

Céline CONSTANT
Sherine BAYA KOCHKAR
Jules BONHOTAL
Joe TOUBIA

DEEP LEARNING :

Classification des

151 Premiers

Pokémons



Introduction

Dans le cadre de notre projet de fin de semestre en deep learning, nous avons choisi de nous concentrer sur la classification des 151 premiers Pokémon. Ce choix s'appuie sur deux motivations principales : d'une part, il s'agit d'un sujet ludique et attrayant, capable de susciter l'engagement, et d'autre part, il pose un défi technique intéressant dans le domaine de la classification d'images. Cette tâche nécessite une approche rigoureuse impliquant le traitement d'images et l'utilisation de modèles avancés en apprentissage automatique.

La classification des Pokémon présente une opportunité unique d'explorer des concepts clés du deep learning. Les 151 Pokémon de la première génération se distinguent par leurs formes, couleurs et motifs variés, offrant une complexité visuelle qui met à l'épreuve la capacité des modèles à extraire et discriminer les caractéristiques visuelles. En parallèle, leur nombre relativement limité permet de concevoir un projet réalisable dans un cadre académique, tout en permettant une évaluation approfondie des performances.

Notre objectif est de comparer l'efficacité de plusieurs algorithmes de classification d'images en termes de précision et de coût computationnel. Pour ce faire, nous exploiterons des datasets existants, issus principalement de plateformes telles que Kaggle, en y appliquant des traitements adaptés : nettoyage, fusion et augmentation des images, afin de garantir des données optimales pour l'entraînement et l'évaluation.

Nous avons sélectionné cinq modèles pour leur complémentarité en termes de structure, de robustesse et de performance dans des tâches similaires :

1. **Un CNN personnalisé** : conçu manuellement, ce modèle nous permet de tester une architecture créée spécifiquement pour cette tâche et d'explorer ses capacités d'apprentissage sur un dataset limité.
2. **EfficientNet** : choisi pour son rapport optimal entre performance et coût computationnel, avec deux variantes — un modèle standard pré-entraîné et une version adaptée à notre tâche spécifique via le transfert d'apprentissage.
3. **ResNet** : connu pour sa capacité à résoudre les problèmes de dégradation grâce à ses connexions résiduelles, ce modèle pré-entraîné sera optimisé pour notre tâche via le transfert d'apprentissage.
4. **DenseNet121** : une architecture réputée pour sa connectivité dense, permettant de limiter les pertes d'information tout en restant relativement légères.
5. **VGG16/VGG19** : ces modèles, bien qu'anciens, restent performants pour des tâches de classification visuelle et seront également adaptés via le transfert d'apprentissage.

Ces modèles ont été choisis pour leur capacité à couvrir un large spectre de performances et de complexité. En comparant leurs résultats, nous visons à identifier celui qui offre le meilleur compromis entre précision et efficacité computationnelle, tout en mettant en évidence leurs forces et limites respectives.

Ce projet constitue une opportunité d'approfondir notre compréhension des étapes clés d'un pipeline de classification d'images : préparation des données, choix des modèles, entraînement, et évaluation. En abordant ces étapes de manière méthodique, nous visons à contribuer à une meilleure compréhension des applications du deep learning dans des problématiques concrètes et engageantes.

Information importante: pour respecter les consignes du rendu nous avons uniquement rendu le fichier de poids et le notebook de notre meilleur modèle. Mais si vous souhaitez voir le reste des notebooks qui ont été créés dans le cadre de ce projet vous pouvez vous rendre sur <https://github.com/jules-bonhotal/ProjetClassificationImagePokemon>

I. Création et préparation du data set

Avant de passer à la phase d'entraînement des modèles, il était essentiel de constituer un dataset adéquat. Nous avons commencé par effectuer des recherches sur diverses plateformes, telles que Kaggle et GitHub, pour identifier un dataset robuste et adapté à la classification des 151 premiers Pokémon. Cependant, aucun dataset unique n'était suffisamment complet ou de qualité pour répondre à nos besoins. Cela nous a conduits à adopter une approche hybride en utilisant et en traitant plusieurs datasets pour construire une base de données d'apprentissage et de test convenable.

Nous avons d'abord travaillé avec le dataset suivant :

[Original 150 Pokémon Image Search Results](#), qui contient environ 250 images pour chacun des 151 premiers Pokémon, obtenues via des recherches Google. Bien que riche en volume, ce dataset présentait des limitations importantes en termes de pertinence des images. Par exemple, pour le Pokémon **Persian**, de nombreuses images représentaient des chats de race persane au lieu du Pokémon lui-même. Ce problème nécessitait un tri manuel rigoureux pour garantir la qualité et la cohérence des données.

A. Tri manuel et sélection des images

Le tri manuel, réalisé sur les 38 000 images du dataset, représentait une tâche conséquente et chronophage. Même en travaillant à un rythme rapide, trier le répertoire de chaque Pokémon nécessitait au minimum 4 minutes, ce qui correspondait à un minimum de 10 heures de travail pour passer en revue les 151 répertoires. Cette tâche ne pouvait pas être automatisée, car elle exigeait une capacité humaine à reconnaître précisément les différents Pokémon, notamment pour distinguer les images pertinentes des éléments erronés ou ambigus.

Le tri a consisté à :

- **Supprimer les images non pertinentes**, c'est-à-dire celles ne représentant pas le Pokémon cible ou présentant une ambiguïté (par exemple, des images d'évolutions ou d'autres formes du Pokémon).
- **Évaluer la qualité des images**, en éliminant celles trop floues ou de résolution insuffisante, tout en évitant un surfiltrage qui aurait réduit la diversité nécessaire pour assurer une bonne généralisation du modèle.
- **Maintenir une diversité visuelle**, en sélectionnant des images légèrement différentes mais toujours représentatives, afin de limiter les risques d'overfitting sans introduire de contradictions dans les données.

De plus, la popularité des Pokémon a influencé cette phase : les Pokémon très populaires comme Pikachu disposaient d'un grand nombre d'images variées et de bonne qualité, facilitant le tri, tandis que d'autres, moins connus, comme Persian, étaient sous-représentés, rendant le processus encore plus délicat.

B. Fusion avec d'autres datasets

Une fois le tri manuel terminé, nous avons enrichi notre base de données en fusionnant le dataset principal avec deux autres datasets :

- Un data set composé des Pokémons de la première génération uniquement avec environ 60 images par pokémon : <https://www.kaggle.com/datasets/thedagger/pokemon-generation-one>
- Un dataset comprenant entre une et huit images pour chaque Pokémon, couvrant les 8 générations, soit environ 800 Pokémons : <https://www.kaggle.com/datasets/hlrhegemony/pokemon-image-dataset?resource=download>

Ces deux datasets contenaient moins d'images que le premier dataset (qui nécessitait une importante phase de tri) mais elles étaient de très bonne qualité, ce qui a limité la nécessité d'un tri manuel supplémentaire.

J'ai commencé par fusionner le dataset contenant exclusivement les Pokémons de la première génération avec celui précédemment trié. Étant donné que les deux datasets ne comportaient que des Pokémons de la première génération, il suffisait simplement d'unifier les répertoires portant le même nom.

Pour le deuxième dataset, qui inclut des Pokémons des huit premières générations, il a fallu extraire uniquement les Pokémons de la première génération parmi les 800 représentés. Plus précisément, j'ai commencé par identifier les répertoires communs entre eux. Chaque répertoire représente un Pokémon, et le but était de conserver uniquement ceux appartenant à la première génération. Une fois les répertoires communs identifiés, un processus de fusion a été mis en place (cf script ci-dessous).

Pour commencer, les deux datasets ont été parcourus pour obtenir la liste des répertoires correspondant aux Pokémons. Une intersection a permis de repérer les noms de répertoires partagés, garantissant que seules les données communes seraient fusionnées.

Puis, un nouveau dossier de sortie a été généré, avec une structure identique à celle des répertoires communs, afin d'organiser les fichiers fusionnés.

Également, lors de la copie des fichiers, une attention particulière a été portée aux doublons éventuels. Si un fichier du deuxième dataset portait le même nom qu'un fichier déjà présent, un suffixe a été ajouté pour différencier les deux versions.

Enfin, le script a inclus des vérifications pour s'assurer que les chemins des datasets et du dossier de sortie étaient valides, et il a créé les dossiers manquants si nécessaire.

```

def merge_datasets(dataset1_path, dataset2_path, output_path):
    """
    Fusionne deux datasets en prenant uniquement les dossiers communs.

    dataset1_path (str): Chemin du premier dataset.
    dataset2_path (str): Chemin du deuxième dataset.
    output_path (str): Chemin du dossier de sortie.
    """

# Vérification des chemins
if os.path.exists(dataset1_path):
    print(f"Chemin du premier dataset existe : {dataset1_path}")
else:
    print(f"Chemin du premier dataset n'existe pas : {dataset1_path}")

if os.path.exists(dataset2_path):
    print(f"Chemin du deuxième dataset existe : {dataset2_path}")
else:
    print(f"Chemin du deuxième dataset n'existe pas : {dataset2_path}")

if os.path.exists(output_path):
    print(f"Chemin du dossier de sortie existe déjà : {output_path}")
else:
    print(f"Chemin du dossier de sortie sera créé : {output_path}")

# Obtenir la liste des dossiers pour chaque dataset
dataset1_folders = set(os.listdir(dataset1_path))
dataset2_folders = set(os.listdir(dataset2_path))

# Identifier les dossiers en commun
common_folders = dataset1_folders.intersection(dataset2_folders)

# Créer le dossier de sortie s'il n'existe pas
os.makedirs(output_path, exist_ok=True)

for folder in common_folders:
    folder_path1 = os.path.join(dataset1_path, folder)
    folder_path2 = os.path.join(dataset2_path, folder)
    output_folder_path = os.path.join(output_path, folder)

    # Créer le dossier de sortie pour ce Pokémon
    os.makedirs(output_folder_path, exist_ok=True)

    # Copier les images du premier dataset
    for file in os.listdir(folder_path1):

        source_file = os.path.join(folder_path1, file)
        dest_file = os.path.join(output_folder_path, file)
        shutil.copy2(source_file, dest_file)

    # Copier les images du deuxième dataset
    for file in os.listdir(folder_path2):
        source_file = os.path.join(folder_path2, file)
        dest_file = os.path.join(output_folder_path, file)

        # Éviter les doublons en cas de fichiers avec le même nom
        if os.path.exists(dest_file):
            file_name, file_ext = os.path.splitext(file)
            dest_file = os.path.join(output_folder_path, f"{file_name}_from_dataset2{file_ext}")

        shutil.copy2(source_file, dest_file)

```

Figure 1: Script python utilisé pour la fusion des data set

C. Préparation des images et des ensembles

Après la fusion, nous avons préparé les images afin de les rendre compatibles avec les exigences des modèles de deep learning. Tout d'abord, toutes les images ont été uniformisées à une taille de 224x224 pixels pour garantir leur cohérence et leur compatibilité avec les architectures choisies. Ensuite, pour améliorer la robustesse du modèle, nous avons appliqué différentes techniques d'augmentation des données. Ces transformations comprenaient des rotations aléatoires allant jusqu'à 20°, des décalages horizontaux et verticaux de 20 %, ainsi que des transformations de cisaillement et de zoom. Nous avons également introduit un renversement horizontal aléatoire pour enrichir davantage la diversité visuelle des images.

Les données ont ensuite été séparées en trois ensembles distincts : un ensemble d'entraînement (80 % des données), destiné à apprendre les caractéristiques des Pokémon ; un ensemble de validation (20 % des données), utilisé pour ajuster les hyperparamètres et prévenir l'overfitting ; et enfin, un ensemble de test, prévu pour une évaluation finale. Ces ensembles ont été organisés en batchs de 32 images, ce qui permet d'optimiser l'entraînement tout en limitant la charge mémoire.

Voici un extrait du code Python utilisé pour cette préparation :

```
dataset_path = "./fusion2"
output_image_size = (224, 224)
batch_size = 32

# générateurs de données
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255, # Normalisation
    rotation_range=20, # Augmentation des données
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    validation_split=0.2 # Fraction de validation
)

# Générateur pour l'ensemble d'entraînement
train_generator = train_datagen.flow_from_directory(
    dataset_path,
    target_size=output_image_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='training'
)

# Générateur pour l'ensemble de validation
validation_generator = train_datagen.flow_from_directory(
    dataset_path,
    target_size=output_image_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation'
)
```

Figure 2: Script Python utilisé pour la préparation des images

À l'issue de ce processus rigoureux de sélection, fusion et préparation, nous avons constitué un dataset contenant un total de **13 612 images**. Ce volume conséquent offre une diversité significative, essentielle pour entraîner nos modèles de classification. Grâce à la diversité des sources et aux techniques d'augmentation appliquées, le dataset est suffisamment représentatif pour permettre à nos modèles d'apprendre à distinguer les 151 premiers Pokémon tout en généralisant efficacement à de nouvelles images.

La séparation des données en ensembles d'entraînement et de validation, combinée à une organisation en batchs, garantit une utilisation optimisée des ressources et une évaluation précise des performances. Ce dataset constitue une base solide et équilibrée pour les prochaines étapes de notre projet, qui consistent à entraîner et comparer plusieurs architectures de deep learning.

1. Modèle CNN

Dans ce projet, nous avons exploré plusieurs algorithmes pour résoudre une tâche de classification d'images. Nous avons commencé par concevoir un réseau neuronal convolutif (CNN) simple. Chaque couche du réseau joue un rôle précis dans l'extraction et la transformation des caractéristiques visuelles, ce qui permet au modèle de généraliser efficacement sur des données inédites. Cette section du rapport détaille l'architecture utilisée, en se concentrant sur le rôle de chaque couche dans des paragraphes dédiés.

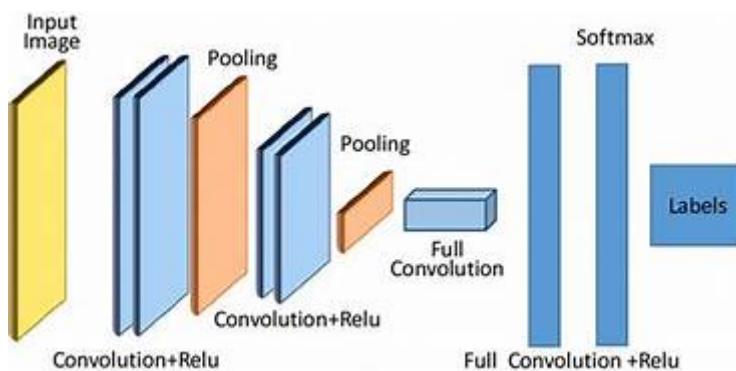


Figure 3 : Illustration de l'architecture d'un CNN

1.1 Couche d'entrée

La couche d'entrée est configurée pour accepter des images au format **(224, 224, 3)**. Ces dimensions correspondent à une largeur et une hauteur de 224 pixels, avec trois canaux pour les couleurs **Rouge, Vert et Bleu (RGB)**. Cela garantit que toutes les images, indépendamment de leur résolution initiale, sont standardisées pour être compatibles avec les couches convolutives suivantes. Cette étape est cruciale pour traiter des images provenant de diverses sources, tout en offrant une base stable pour extraire des motifs visuels de manière uniforme.

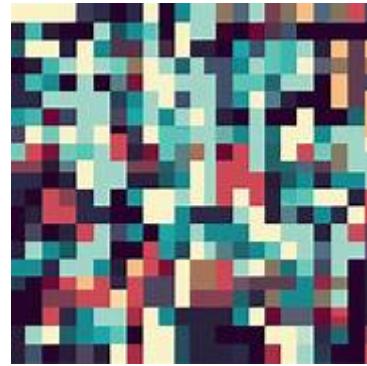


Figure 4 : Illustration des pixels d'une image

1.2 Première couche convolutionnelle

Cette couche applique **32 filtres** de taille **(5x5)** avec un **stride de 1** et un **padding same**, ce qui garantit que les dimensions d'entrée sont conservées après l'opération. Chaque filtre analyse des parties locales de l'image pour détecter des motifs simples comme des contours ou des textures. Les **32 cartes de caractéristiques** générées par cette couche représentent une première abstraction des motifs visuels. La fonction d'activation utilisée est **ReLU (Rectified Linear Unit)**, qui accélère la convergence en éliminant les activations négatives.

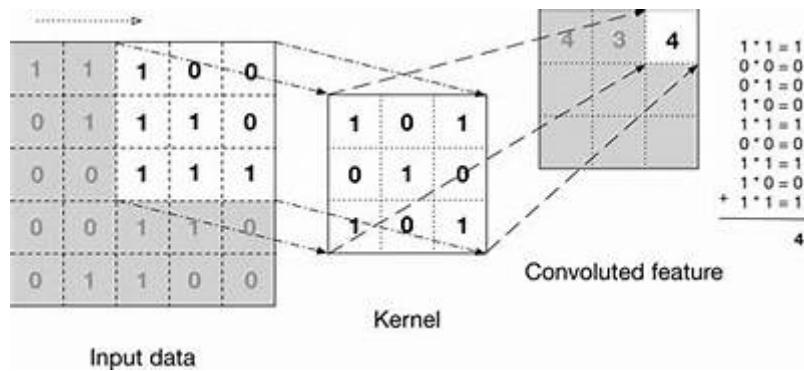


Figure 5 : Illustration du fonctionnement d'une couche convulsive

3.3 Première couche de pooling

Un **max pooling** avec une fenêtre de **(2x2)** est appliqué après la première convolution. Cette opération réduit les dimensions spatiales de moitié, simplifiant les calculs tout en conservant les informations essentielles. Le pooling aide également le modèle à devenir invariant aux petites translations dans les images, améliorant ainsi sa robustesse face aux variations d'entrée.

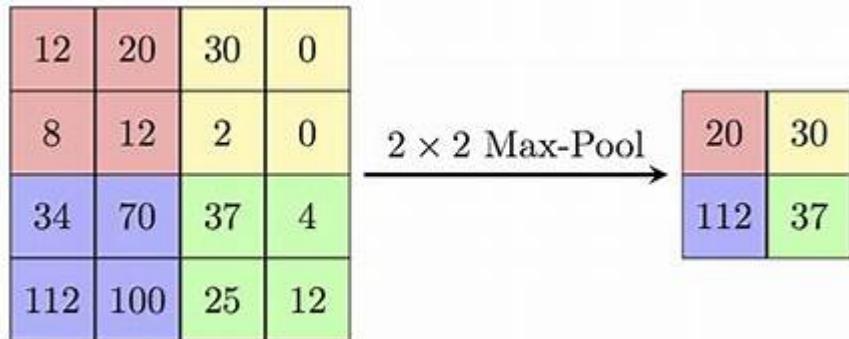


Figure 4 : Exemple du fonctionnement du max pooling

3.4 Deuxième couche convolutionnelle

La deuxième couche convolutionnelle utilise **64 filtres** de taille **(3x3)** avec un **stride de 1** et un **padding same**. Contrairement à la première couche, elle se concentre sur des motifs plus complexes en combinant les informations extraites précédemment. Chaque filtre est optimisé pour détecter des structures telles que des formes ou des objets partiels. La fonction d'activation ReLU est utilisée ici pour continuer à modéliser les relations non linéaires dans les données.

3.5 Deuxième couche de pooling

Un second **max pooling** avec une fenêtre de **(2x2)** est appliqué pour réduire encore les dimensions spatiales. Cela permet de simplifier les calculs tout en préservant les motifs visuels les plus pertinents pour la tâche de classification.

3.6 Troisième couche convolutionnelle

Cette couche utilise **128 filtres** de taille **(3x3)**, avec un **stride de 1** et un **padding same**. Elle se concentre sur l'extraction de motifs complexes et abstraits, en combinant les motifs simples et complexes identifiés dans les couches précédentes. Ces motifs incluent souvent des relations globales dans l'image, permettant au modèle de mieux comprendre la structure globale des objets.

3.7 Troisième couche de pooling

Un troisième **max pooling** est appliqué avec une fenêtre de **(2x2)**. Cela réduit davantage les dimensions, préparant les cartes de caractéristiques pour les couches entièrement connectées. Cette étape garantit que seules les informations les plus pertinentes sont transmises aux couches suivantes, tout en diminuant la complexité computationnelle.

3.8 Couche Flatten

La couche Flatten transforme les cartes de caractéristiques 3D obtenues après les convolutions et le pooling en un vecteur 1D. Ce vecteur linéaire regroupe toutes les informations extraites et sert de passerelle entre les couches convolutives et les couches denses. Cette transformation est essentielle pour que les données puissent être traitées par des couches entièrement connectées.

3.9 Première couche dense

Cette couche dense contient **256 neurones**, chacun connecté à toutes les sorties de la couche Flatten. Elle permet d'apprendre des relations complexes entre les caractéristiques extraites, en combinant les informations pour les rendre exploitable par la couche finale. La fonction d'activation **ReLU** continue d'aider à modéliser des relations non linéaires.

3.10 Couche Dropout

La couche Dropout applique un taux de désactivation de **50 %**, désactivant aléatoirement certains neurones pendant l'entraînement. Cela force le réseau à apprendre des motifs globaux plutôt que de s'appuyer sur des caractéristiques spécifiques, améliorant ainsi la généralisation du modèle.

3.11 Couche de sortie

La couche finale est une couche dense avec une activation **softmax**, qui génère une distribution de probabilités sur les classes cibles. Cela permet au modèle de prédire la classe la plus probable pour chaque image, tout en offrant une vue quantitative de la confiance du modèle dans ses prédictions.

CNN simple

```
import tensorflow as tf

# Définition du modèle CNN
model = tf.keras.Sequential([
    # Entrée du modèle, taille des images (224, 224, 3)
    tf.keras.layers.Input(shape=(224, 224, 3)), # Couche d'entrée spécifiant la taille de l'image d'entrée (224x224 pixels, 3 canaux couleur)

    # Première couche de convolution : 32 filtres, taille de filtre (5x5), activation ReLU
    tf.keras.layers.Conv2D(32, (5, 5), strides=1, padding='same', activation='relu'),
    # Extraction des caractéristiques locales avec 32 filtres de taille 5x5, "same" conserve la taille d'entrée.

    # Première couche de pooling : réduction de la taille (2x2)
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

    # Réduit les dimensions spatiales de moitié (224x224 -> 112x112), conserve les caractéristiques principales.

    # Deuxième couche de convolution : 64 filtres, taille de filtre (3x3), activation ReLU
    tf.keras.layers.Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'),
    # Ajout de 64 filtres pour capturer des caractéristiques plus complexes.

    # Deuxième couche de pooling : réduction de la taille (2x2)
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    # Réduction supplémentaire des dimensions spatiales (112x112 -> 56x56).
```

```

# Troisième couche de convolution : 128 filtres, taille de filtre (3x3), activation ReLU
tf.keras.layers.Conv2D(128, (3, 3), strides=1, padding='same', activation='relu'),
# Capture des motifs plus complexes avec 128 filtres.

# Troisième couche de pooling : réduction de la taille (2x2)
tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
# Réduction finale des dimensions spatiales (56x56 -> 28x28).

# Couche Flatten : aplatissement des dimensions en un vecteur 1D
tf.keras.layers.Flatten(),
# Transforme la sortie 28x28x128 en un vecteur linéaire de 100352 pour la couche dense suivante.

# Première couche dense (Fully Connected Layer) : 256 neurones, activation ReLU
tf.keras.layers.Dense(256, activation='relu'),
# Combine toutes les caractéristiques pour apprendre des motifs complexes.

# Couche Dropout : désactivation aléatoire de 50% des neurones pour éviter le surapprentissage
tf.keras.layers.Dropout(0.5),
# Améliore la généralisation du modèle en désactivant des neurones de manière aléatoire.

# Dernière couche dense : nombre de sorties égal au nombre de classes, activation softmax
tf.keras.layers.Dense(len(train_generator.class_indices), activation='softmax')
# Prédiction des probabilités pour chaque classe (classification multi-classes).
])

```

Figure 6: Implémentation CNN Simple

3.12 Entraînement du CNN

L’entraînement du modèle CNN a été réalisé en utilisant un ensemble de données divisé en **80 % pour l’entraînement et 20 % pour la validation interne**. Les images ont été prétraitées pour garantir une normalisation entre 0 et 1, tandis que les labels ont été convertis en **one-hot encoding** pour correspondre à la fonction de perte choisie (**categorical_crossentropy**).

Le modèle a été configuré avec un **batch size de 32**, permettant un bon équilibre entre efficacité computationnelle et stabilité des gradients. Pendant **20 époques**, l’optimiseur **Adam**, avec un learning rate fixé à **0.0001**, a ajusté les poids du modèle pour minimiser la perte d’entraînement. Une validation interne a été effectuée à chaque époque pour suivre les performances et identifier d’éventuels signes de surapprentissage. La précision sur l’ensemble d’entraînement a atteint **97.08 %** avec une perte réduite à **0.1088**, montrant que le modèle a bien appris à extraire et exploiter les motifs des images.

3.13 Analyse des résultats

L’entraînement du modèle **CNN** a produit des résultats impressionnantes, avec une précision parfaite de **100%** sur l’ensemble d’entraînement dès l’époque 4 et une précision stable de **99,03%** sur l’ensemble de validation tout au long des 20 époques. La perte d’entraînement a rapidement diminué pour atteindre des valeurs quasi nulles (par exemple **2,25e-05** à l’époque 20), indiquant une excellente adaptation du modèle aux données d’entraînement.

Cependant, la perte de validation a légèrement augmenté au fil des époques, passant de **0,1716** à **0,2277**, ce qui pourrait suggérer un début de sur-apprentissage, où le modèle mémorise trop les données d'entraînement au lieu de généraliser davantage.

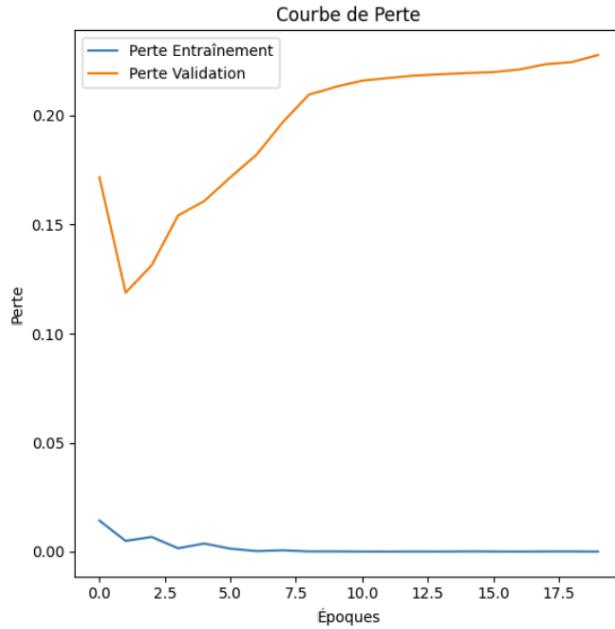


Figure 7: Illustration de courbe de perte

La figure ci-dessus illustre l'évolution de la précision d'entraînement et de validation au cours des 20 époques d'entraînement pour le modèle **CNN**. Dès les premières époques, la précision d'entraînement augmente rapidement, atteignant presque **100%** dès l'époque 4, témoignant de la capacité du modèle à apprendre efficacement les motifs présents dans les données d'entraînement. Après cette convergence rapide, la précision d'entraînement reste parfaitement stable pour le reste des époques, montrant que le modèle est bien ajusté aux données d'entraînement sans erreurs significatives.

En parallèle, la précision de validation se maintient à un niveau constant de **99,03%** tout au long des 20 époques, ce qui reflète une excellente généralisation du modèle sur les données non vues.

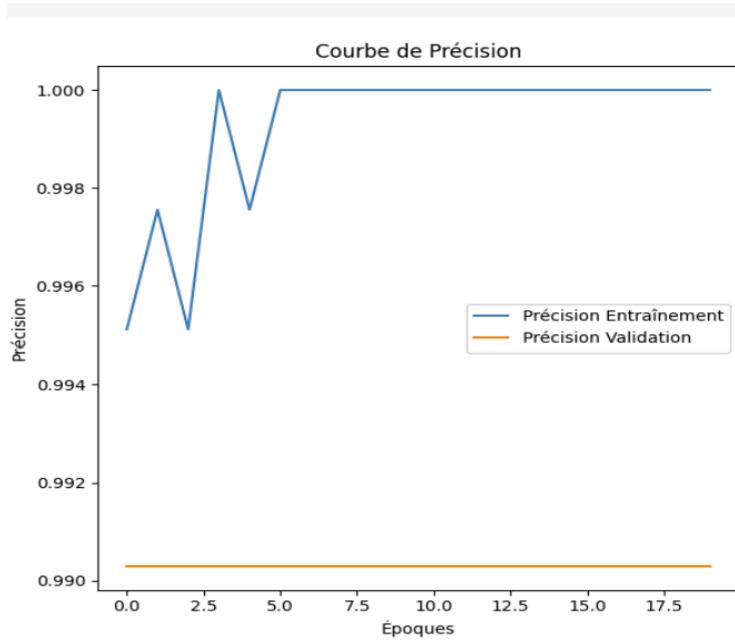


Figure 8: Illustration de la courbe de précision

La courbe de précision (**Figure 8**) met en lumière une évolution rapide et stable des performances du modèle **CNN**. Dès les premières époques, la précision d'entraînement atteint presque **100%** dès l'époque 4, avec une convergence parfaite qui se maintient de manière stable pour le reste des époques. En revanche, la précision de validation reste constante à **99,03%** tout au long des 20 époques, ce qui témoigne d'une bonne capacité de généralisation du modèle sur des données non vues. Cependant, la différence entre les deux courbes, bien que minime, mérite d'être notée : la précision d'entraînement parfaite contraste avec l'absence de progression de la précision de validation, ce qui pourrait indiquer une saturation ou une limite dans la diversité et la représentativité des données de validation.

3 EfficientNet

3.1 Contexte

EfficientNet est une architecture conçue pour optimiser l'efficacité des réseaux neuronaux tout en maintenant une haute précision. L'un des éléments clés de l'architecture d'EfficientNet est sa méthode de mise à l'échelle uniforme, qui vise à optimiser la performance tout en réduisant les besoins en calcul.

3.2 Facteurs Clés de la Mise à l'Échelle Uniforme

Cette approche repose sur trois facteurs principaux qui sont ajustés simultanément :

- **Profondeur** : Augmenter le nombre de couches dans le réseau permet de mieux capturer des représentations complexes et de renforcer la capacité du modèle à apprendre des caractéristiques plus fines.
- **Largeur** : L'augmentation du nombre de filtres par couche améliore la capacité du modèle à détecter des motifs plus subtils, contribuant ainsi à sa capacité à généraliser sur des données diverses.
- **Résolution** : L'augmentation de la taille des images en entrée permet au modèle de capter davantage de détails, bien qu'elle entraîne également une demande plus élevée en termes de ressources computationnelles.

3.3 Stratégies de Mise à l'Échelle

Les stratégies de mise à l'échelle d'EfficientNet, qui visent à optimiser les trois facteurs clés varient en fonction des ressources disponibles et des exigences spécifiques du modèle. Les différentes approches, illustrées ci-dessous, incluent :

- **Ligne de base** : Utilisation du réseau d'origine sans mise à l'échelle.
- **Mise à l'échelle de la largeur** : Augmentation du nombre de canaux dans chaque couche, améliorant ainsi l'extraction des caractéristiques.
- **Mise à l'échelle de la profondeur** : Augmentation du nombre de couches pour mieux capturer des représentations plus profondes et complexes.
- **Mise à l'échelle de la résolution** : Augmentation de la taille des images d'entrée pour capter davantage de détails visuels.
- **Mise à l'échelle composée** : Combinaison simultanée de l'augmentation de la largeur, de la profondeur et de la résolution, optimisée par une formule spécifique pour équilibrer les performances et l'efficacité des ressources.

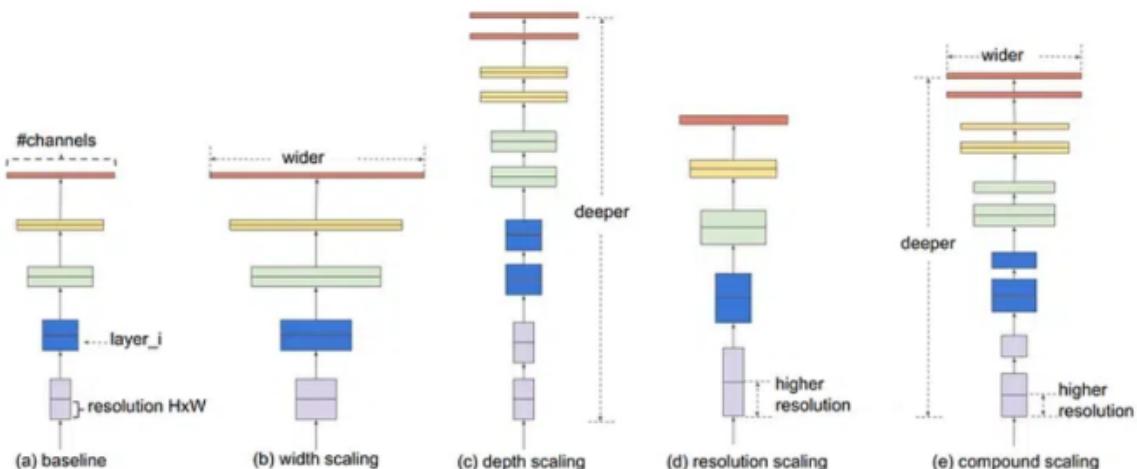


Figure 9: Mise à l'échelle de modèle

3.4 Architecture du Modèle

Comme le montre l'image ci-dessous, EfficientNet-B0 repose sur une architecture légère et efficace, utilisant des blocs MBConv (Mobile Inverted Bottleneck Convolution), qui représentent une amélioration des convolutions classiques. Contrairement aux convolutions standard, les blocs MBConv intègrent une étape de compression pour réduire le nombre de canaux, allégeant ainsi les calculs, suivie d'une expansion pour rétablir les dimensions et enrichir les représentations des données.

L'architecture inclut également des couches de normalisation par batch, des activations non linéaires (Swish) et des mécanismes de régularisation comme le dropout pour optimiser la performance. Enfin, des couches fully-connected sont utilisées à la fin du modèle pour la classification[1,2].

EfficientNet est décliné en plusieurs versions, de B0 (la plus légère) à B7 (la plus complexe). Chaque version adapte la profondeur, la largeur et la résolution pour répondre à des exigences spécifiques, garantissant un équilibre optimal entre performance et consommation des ressources.

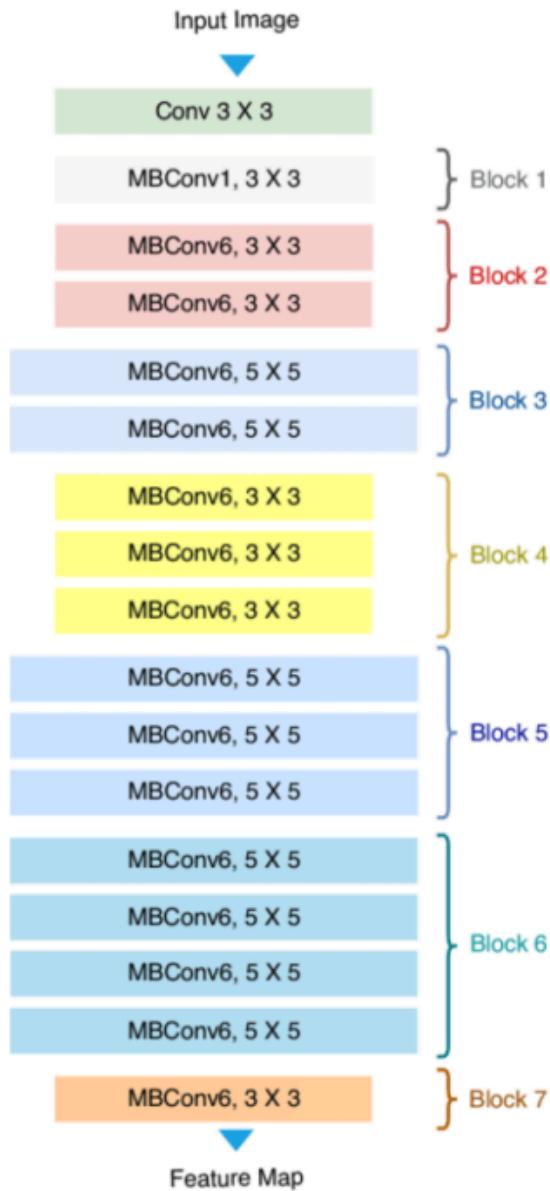


Figure 10: Architecture du Modèle

3.5 Application à la Classification de Pokémons

Grâce à sa mise à l'échelle uniforme, le modèle peut s'ajuster pour traiter des images de différentes résolutions, ce qui est essentiel pour capturer les détails visuels fins des Pokémons, tout en réduisant les besoins en calcul. Nous avons appliqué le transfert de learning en chargeant un modèle EfficientNet-B0 pré-entraîné et en modifiant la dernière couche fully-connected pour qu'elle corresponde au nombre de classes dans notre dataset.

3.5.1 Modèle 1 : EfficientNet-B0 avec Images 224x224

Pour ce modèle, nous avons utilisé EfficientNet-B0. Les images ont été redimensionnées à 224x224 pixels. Cette transformation garantit que toutes les images sont homogènes en termes de dimensions, ce qui est crucial pour un traitement cohérent par le réseau de neurones.

Hyperparamètres Utilisés

- **Nombre d'Époques** : 10
- **Taille du Lot** : 32
- **Taux d'Apprentissage** : 0.001
- **Transformations des Images** : Les images ont été redimensionnées, converties en tenseurs et normalisées. Les valeurs de normalisation utilisées sont la moyenne et l'écart type basés sur le dataset ImageNet, correspondant respectivement à [0.485, 0.456, 0.406] et [0.229, 0.224, 0.225]

3.5.2 Modèle 2 : EfficientNet-B0 avec Ajustements

Après avoir observé les résultats du Modèle 1, des ajustements ont été apportés pour améliorer les performances.

Modèle et Prétraitement des Données

- **Redimensionnement** : Les images ont été réduites à 160x160 pixels.
- **Flip Horizontal Aléatoire** : Pour augmenter la variété des données d'entraînement.
- **Rotation Aléatoire** : Pour améliorer la robustesse du modèle face aux orientations variées des images.
- **Normalisation** : Utilisation des moyennes et des écarts types basés sur le dataset ImageNet ([0.485, 0.456, 0.406] et [0.229, 0.224, 0.225]).
- **Dropout** : Ajout d'une couche de dropout avec un taux de 50% avant la dernière couche fully-connected.

Hyperparamètres Utilisés

- **Nombre d'Époques** : 10
- **Taille du Lot** : 32
- **Taux d'Apprentissage** : 0.001

Résultats

Les résultats ci-dessous montrent les performances des modèles après 10 époques d'entraînement

	Modèle 1	Modèle 2
Perte d'Entrainement	0.0737	0.0509
Perte de Validation	0.6248	0.2574
Perte de Test	0.5994	0.2375
Precision d'Entrainement	97.97%	98.69%
Precision de Validation	84.53%	94.75%
precision de Test	85.74%	95.27%

En observant les histogrammes de perte ci-dessous, on peut suivre l'évolution de la perte d'entraînement et de validation au fil des époques. Afin d'obtenir de meilleurs résultats, il est crucial de trouver un équilibre optimal entre l'entraînement et la validation pour les deux modèles. Pour le Modèle 1 (Figure 11), la meilleure valeur de perte est obtenue à la 8ème époque.

Pour le Modèle 2 (Figure 12), la perte d'entraînement et de validation atteint une valeur plus faible après la 8ème époque, et reste ensuite relativement constante, ce qui indique qu'il est possible de réduire le nombre d'époques de notre modèle à 8, tout en maintenant des performances optimales.

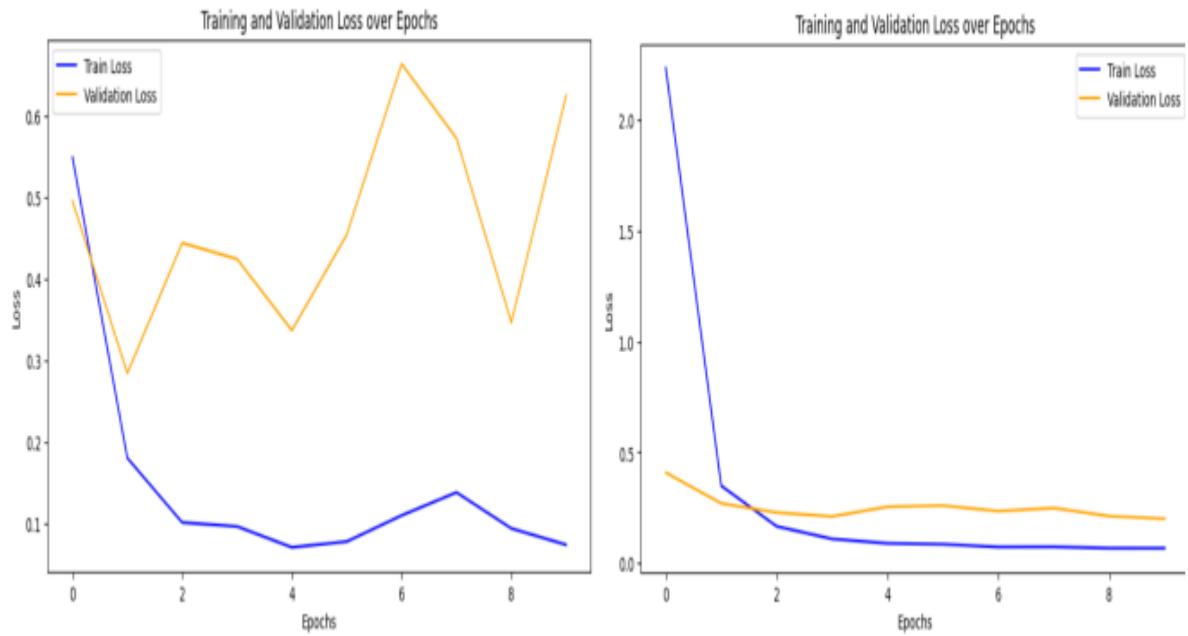


Figure 11: Train et validation loss pour modèle 1

Figure 12: Train et validation loss pour model 2

Analyse des résultats de classification sur le modèle 2

D'après les résultats précédents, j'ai décidé de continuer avec le **modèle 2**. Sur l'échantillon de 30 images tirées du jeu de données de test, il est observé que deux images ont été mal classifiées. Malgré cela, la performance globale du modèle reste satisfaisante, avec une majorité d'images correctement prédites.

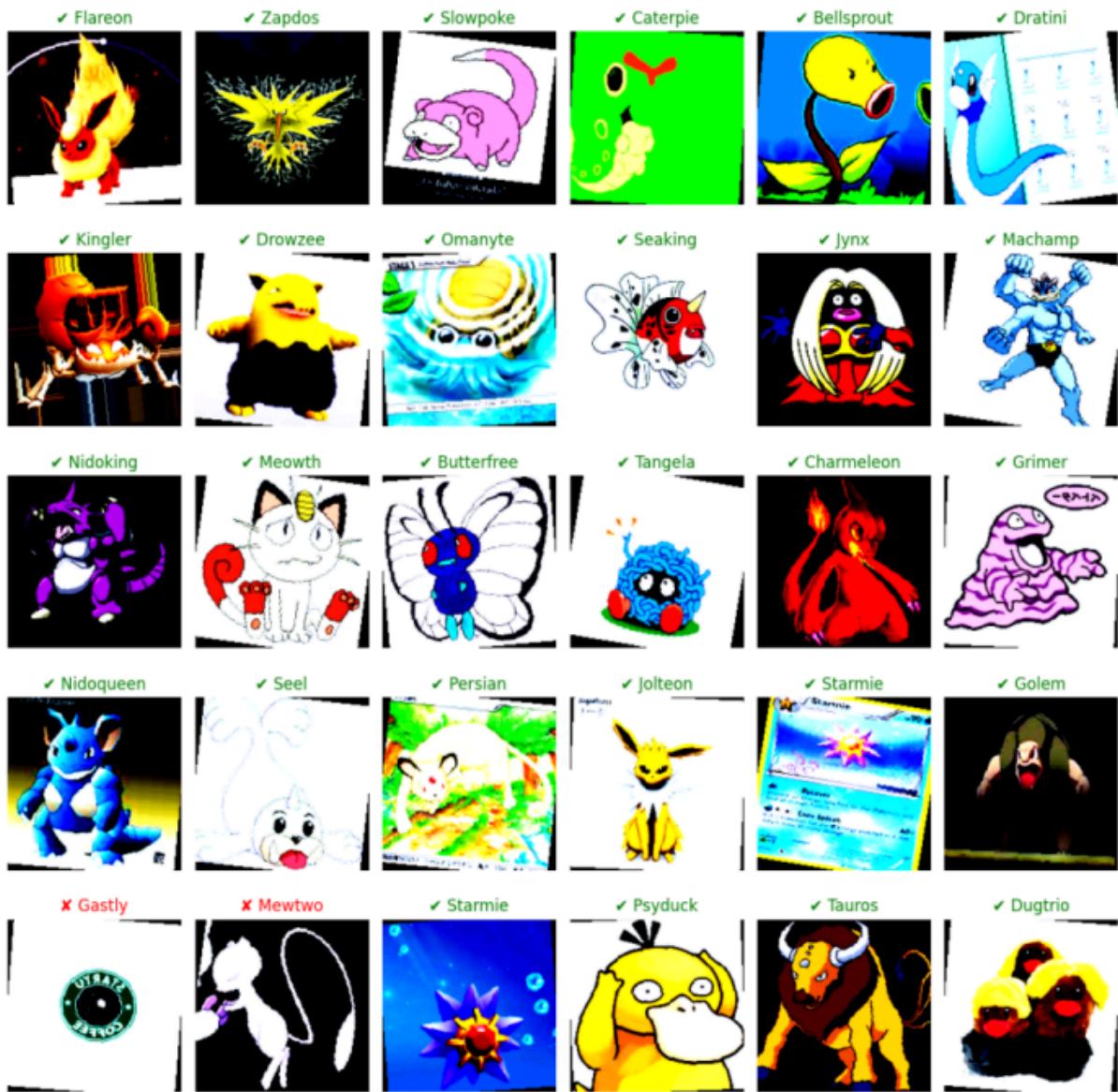


Figure 13: Résultats de classification

3.5.3 Modèle simple pour la classification

Pour comparer un modèle développé à partir de zéro avec un modèle préexistant comme EfficientNet, une architecture CNN basique a été utilisée.

Ce modèle comprend quatre couches convolutionnelles de tailles croissantes (64, 128, 256, 512 filtres) et de la normalisation par lot pour améliorer l'entraînement. Après chaque couche, un pooling réduit les dimensions, suivi par une Global Average Pooling pour compresser l'information. Le modèle utilise deux couches entièrement connectées avec un dropout pour éviter le surapprentissage.

En voyant les résultats, notamment avec une perte d'entraînement de 4.9973 et une précision de 0.99% pour l'entraînement, contre une perte de validation de 5.0237 et une précision de 0.60% pour la validation après la dernière époque, il est évident que la perte de validation est trop élevée. Cela suggère que ce modèle n'est pas adapté pour notre problème de classification.

4 ResNet 50

4.1 Contexte

ResNet50, ou "Residuary Network 50", est une architecture de réseau de neurones profonds qui introduit des blocs résiduels pour faciliter l'entraînement de réseaux très profonds. L'innovation clé de ResNet50 réside dans l'utilisation de connexions résiduelles, qui permettent de combiner les informations d'entrée directement avec les informations de sortie d'une couche, en sautant une ou plusieurs couches intermédiaires.

4.2 Architecture du Modèle

ResNet50 repose sur une architecture qui utilise des blocs résiduels, où chaque couche reçoit en entrée la concaténation de toutes les couches précédentes.

Contrairement aux architectures traditionnelles, ces blocs intègrent une étape de compression pour réduire le nombre de canaux, allégeant ainsi les calculs, suivie d'une expansion pour rétablir les dimensions et enrichir les représentations des données.

L'architecture inclut également des couches de normalisation par batch [3], des activations non linéaires (ReLU) et des mécanismes de régularisation comme le dropout pour optimiser la performance. Enfin, des couches fully-connected sont utilisées à la fin du modèle pour la classification. ResNet50 est conçu pour améliorer l'efficacité de l'apprentissage tout en maintenant une haute précision.

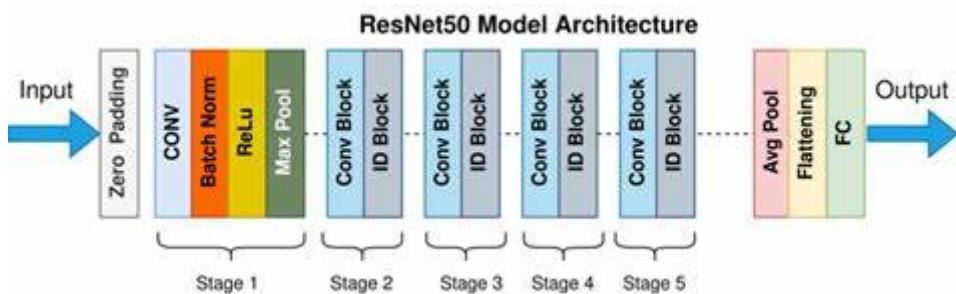


Figure 14: Illustration de l'architecture de ResNet50

4.3 Application à la Classification de Pokémon

Nous avons mis en œuvre un modèle **ResNet50** pré-entraîné pour classifier un ensemble d'images en tirant parti de l'écosystème TensorFlow et Keras. Le modèle a été initialisé avec des poids pré-entraînés issus du dataset ImageNet , tout en excluant ses couches entièrement connectées d'origine (option `include_top=False`). Cela nous a permis de personnaliser les couches finales pour les adapter aux spécificités de notre jeu de données. Les images ont été redimensionnées à 224x224 pixels , conformément aux dimensions d'entrée attendues par ResNet50, garantissant ainsi une homogénéité des données et un traitement compatible avec le réseau.

Pour compléter le modèle, nous avons ajouté une couche de pooling global (GlobalAveragePooling2D) qui a permis de réduire les dimensions des cartes de caractéristiques tout en conservant les informations les plus importantes. Une couche dense de 1024 neurones a ensuite été ajoutée, activée par une fonction ReLU , afin de capturer des relations complexes entre les caractéristiques extraites par les couches convolutives. Enfin, une couche de sortie composée de 151 neurones , correspondant au nombre de classes dans notre jeu de données, a été intégrée. Cette dernière utilise une activation softmax pour produire une distribution de probabilités et effectuer une classification multi-classes.

Pour traiter efficacement le déséquilibre des classes dans notre jeu de données, le poids de classe a été calculé à l'aide de la bibliothèque sklearn. Ces poids, obtenus grâce à la méthode `class_weight='balanced'`, permettent d'accorder une importance proportionnelle aux classes minoritaires, notamment l'impact du déséquilibre sur l'entraînement. Ces poids ont été intégrés dans l'entraînement via l'argument `class_weight` de la méthode `model.fit()`.

Le modèle a été compilé avec l'optimiseur **Adam** , configuré avec un taux d'apprentissage initial de **0.0001** , afin d'assurer une convergence efficace. La fonction de perte `categorical_crossentropy` a été utilisée, car elle est adaptée aux problèmes de classification multi-classes. La précision métrique a été spécifiée pour suivre les performances pendant l'entraînement **20 époques** , avec une taille de **lot32** , en utilisant des images normalisées.

```
25] # Importation des bibliothèques nécessaires
from tensorflow.keras.applications import ResNet50 # Modèle pré-entraîné ResNet50
from tensorflow.keras.models import Model # Pour personnaliser le modèle
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense # Couches personnalisées
from tensorflow.keras.optimizers import Adam # Optimiseur pour entraîner le modèle
from tensorflow.keras.preprocessing.image import ImageDataGenerator # Prétraitement des données
from sklearn.utils import class_weight # Calcul des poids pour les classes déséquilibrées
import numpy as np

# Chargement du modèle ResNet50 pré-entraîné
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Ajout des couches personnalisées
Y = base_model.output
Y = GlobalAveragePooling2D()(Y) # Pooling global pour réduire les dimensions
Y = Dense(1024, activation='relu')(Y) # Couche dense avec 1024 neurones
predictions = Dense(train_generator.num_classes, activation='softmax')(Y) # Couche de sortie pour la classification

# Création du modèle final
model = Model(inputs=base_model.input, outputs=predictions)
```

Figure 15: Implémentation ResNet

4.4 Analyse des résultats obtenus

Le modèle **ResNet50** a été entraîné sur 20 époques, affichant une progression rapide des performances avec des résultats globalement prometteurs, mais aussi des signes de surapprentissage dès l'époque 7. La précision d'entraînement, initialement à **70,15%** (époque 1), a atteint **97,67%** à l'époque 10, accompagnée d'une diminution significative de la perte d'entraînement, de **1,4765** à **0,0815**, montrant que le modèle a efficacement appris les motifs des données d'entraînement.

En validation, la précision a commencé à un niveau bas (**5,97%** à l'époque 1), progressant rapidement pour atteindre **96,58%** à l'époque 10, tandis que la perte diminuait de **4,5969** à **0,1344**. Cependant, à partir de l'époque 7, des signes de surapprentissage sont apparus, avec une précision

de validation chutant à **93,50%** et une perte augmentant de **0,1424** (époque 6) à **0,2837** (époque 8). Cela reflète une spécialisation excessive du modèle sur les données d'entraînement, au détriment de sa capacité à généraliser sur des données non vues.

Une récupération partielle a toutefois été observée après l'époque 8, avec une précision de validation remontant à **96,80%** à l'époque 9 et une perte réduite à **0,1368**, indiquant une meilleure stabilisation grâce aux mécanismes d'optimisation. Les résultats finaux montrent une précision de validation atteignant **97,54%** (époque 16), mais avec des fluctuations dans les dernières époques, notamment une perte de validation atteignant **0,3016** à l'époque 14.

Le modèle **ResNet50** a également été évalué qualitativement à travers la visualisation des prédictions pour chaque classe. Une méthode a été mise en place pour sélectionner aléatoirement une image représentative de chaque classe parmi les données d'entraînement, en utilisant le générateur de données. Le modèle a ensuite prédit la classe associée à chaque image, permettant de comparer directement les prédictions avec les labels réels. Comme illustré dans la **Figure 16**, certaines confusions sont apparentes : pour la classe **Abra**, le modèle a prédit **Golduck** au lieu de la classe correcte. De même, pour **Bulbasaur**, la prédiction était incorrectement attribuée à **Articuno**, bien que ces deux Pokémon soient visuellement distincts.

Lors de cette évaluation, un total de **151 images**, correspondant à une image par classe, a été affiché dans une grille de dimensions **19x8**. Les prédictions correctes ont été annotées en **bleu**, tandis que les erreurs ont été mises en évidence en **rouge**, comme illustré dans la **Figure 16**. Certaines classes, comme **Charizard**, ont également été mal classées comme **Golduck**, montrant une tendance à confondre des classes sans similarités visuelles évidentes. Ces erreurs soulignent des faiblesses spécifiques du modèle qui pourraient être dues à un déséquilibre dans les données d'entraînement ou à des caractéristiques sous-représentées dans certaines classes.

Les résultats qualitatifs confirment globalement les performances observées dans les métriques quantitatives : la majorité des prédictions sont correctes, mais des confusions récurrentes subsistent. Les exemples dans la **Figure 16** mettent en évidence des tendances spécifiques, comme les confusions fréquentes avec **Golduck** et **Articuno**, qui pourraient être corrigées par une meilleure représentation de ces classes dans l'entraînement ou par des ajustements dans les hyperparamètres.

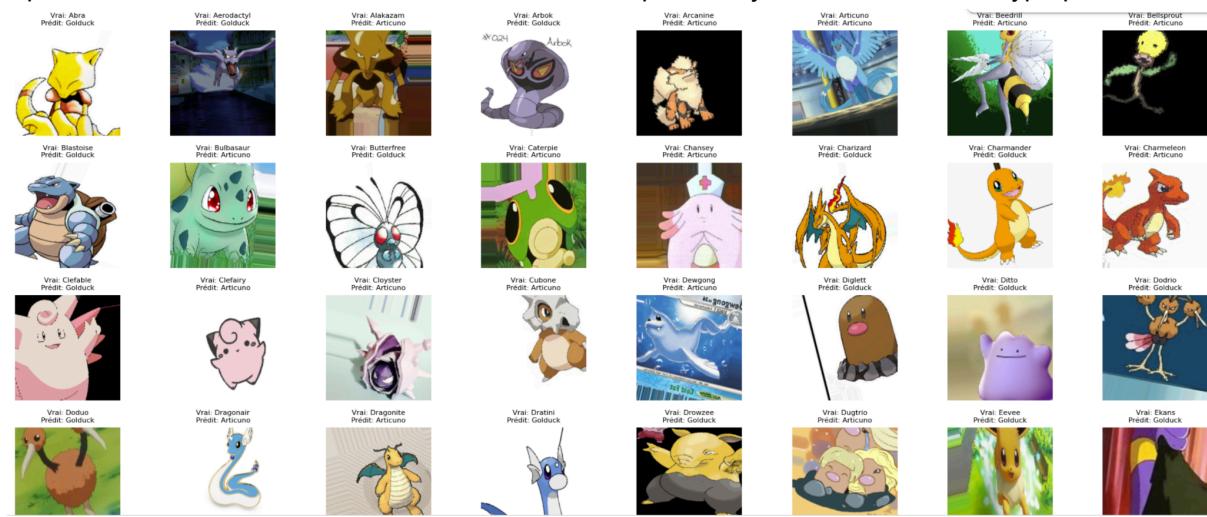


Figure 16: Résultats des classification de prédiction

4. DenseNet

4.1 Contexte

DenseNet est une architecture de réseau de neurones conçue pour maximiser l'efficacité de la transmission d'informations tout en impliquant la redondance[4]. Un des aspects clés de DenseNet est l'utilisation de connexions denses, où chaque couche est connectée à toutes les couches précédentes. Cette approche permet une meilleure réutilisation des caractéristiques, facilite la propagation des gradients et réduit le nombre de paramètres par rapport à d'autres architectures profondes.

4.2 Architecture du Modèle

DenseNet repose sur une architecture composée de blocs denses. Dans ces blocs, chaque couche reçoit en entrée la concaténation des sorties de toutes les couches précédentes, favorisant un apprentissage hiérarchique et une utilisation efficace des caractéristiques. Entre les blocs denses, des couches de transition sont intégrées, combinant convolutions 1x1 et pooling pour réduire les dimensions spatiales et le nombre de canaux, limitant ainsi les coûts informatiques.

Chaque couche du modèle inclut des éléments classiques tels que la normalisation par batch, des activations non linéaires (ReLU), et parfois des mécanismes de régularisation comme le dropout en fonction de l'application. DenseNet se termine généralement par une couche de pooling globale avant une couche entièrement connectée utilisée pour la classification finale.

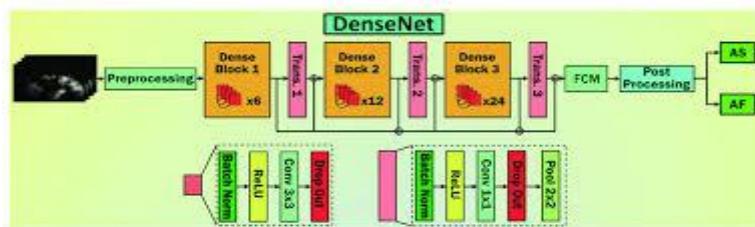


Figure 17: Architecture DenseNet

```
# Charger le modèle DenseNet121 pré-entraîné avec les poids d'ImageNet
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

```
# Ajouter des couches personnalisées
custom_layers = base_model.output
custom_layers = GlobalAveragePooling2D()(custom_layers) # Pooling global pour réduire les dimensions
custom_layers = Dense(1024, activation='relu')(custom_layers) # Couche dense avec 1024 neurones
custom_layers = Dropout(0.5)(custom_layers) # Dropout pour réduire le surapprentissage
custom_layers = Dense(512, activation='relu')(custom_layers) # Une autre couche dense
custom_layers = Dropout(0.5)(custom_layers) # Second Dropout
predictions = Dense(151, activation='softmax')(custom_layers) # Couche de sortie avec 151 classes
```

Figure 18: Implémentation DenseNet121

4.3 Application à la Classification de Pokémons

Nous avons utilisé l'architecture DenseNet121 pré-entraînée avec les poids d'ImageNet, adaptée pour une tâche de classification d'images à 151 classes. Les modifications apportées incluent une couche de global average pooling pour réduire les dimensions des caractéristiques extraites, suivie de couches denses activées par ReLU (1024 et 512 neurones respectivement), ainsi que des couches de dropout (taux de 0,5) pour limiter le surapprentissage. L'entraînement du modèle a été effectué sur 20 époques, en utilisant l'optimiseur **Adam** avec un **taux d'apprentissage de 0,0001**, choisi pour garantir une convergence stable tout en évitant les oscillations.

Une attention particulière a été portée à l'équilibre des données grâce **au calcul des poids des classes**, permettant de compenser les déséquilibres éventuels dans la répartition des classes. Ces poids ont été intégrés directement dans la fonction de perte pour s'assurer que toutes les classes contribuent équitablement à l'apprentissage. Les performances du modèle ont été suivies à chaque époque, avec une évaluation basée sur des métriques telles que la précision et la perte, appliquées sur les ensembles d'entraînement et de validation.

4.4 Analyse des résultats obtenus

Le modèle **DenseNet121** a démontré des performances exceptionnelles, atteignant une précision de validation finale de **97,82%**, témoignant de sa capacité à généraliser efficacement sur des données non vues.

Dès l'époque 1, le modèle a commencé avec une précision de validation de **55,04%**, progressant régulièrement jusqu'à un pic de **98,88%** à l'époque 18, avant de légèrement diminuer à **97,42%** à l'époque 20, suggérant une possible saturation ou une légère sur-adaptation aux données d'entraînement. En parallèle, la perte de validation a diminué de manière cohérente, passant de **2,5416** à l'époque 1 à **0,0854** à l'époque 20, reflétant une minimisation efficace de l'erreur.

Le rapport de classification met en lumière des **f1-scores élevés** pour la plupart des classes, avec des performances parfaites (précision, rappel, et f1-score à **100%**) pour des classes comme *Bulbasaur*, *Wigglytuff*, et *Zubat*, ce qui indique une excellente capacité du modèle à capturer leurs caractéristiques distinctives. Cependant, certaines classes, telles que *Alakazam* (92%) et *Weezing* (96,67%), présentent des scores légèrement inférieurs, probablement en raison d'une représentation moindre ou d'une diversité limitée dans les données d'entraînement.

Les métriques utilisées, notamment la **précision**, le **rappel**, et le **f1-score**, ont permis une évaluation fine et approfondie des performances du modèle, soulignant son équilibre entre exactitude et sensibilité. L'entraînement a été optimisé grâce à l'utilisation de l'optimiseur **Adam**, configuré avec un taux d'apprentissage de **0,0001**, garantissant une convergence stable et progressive. De plus, l'intégration de couches denses personnalisées et de **dropout** a permis de réduire efficacement le surapprentissage, renforçant ainsi la robustesse du modèle. Ces résultats illustrent parfaitement la puissance et la fiabilité de l'architecture DenseNet121 pour une tâche de classification multi-classes exigeante, mettant en avant son efficacité dans l'apprentissage des caractéristiques complexes des données.

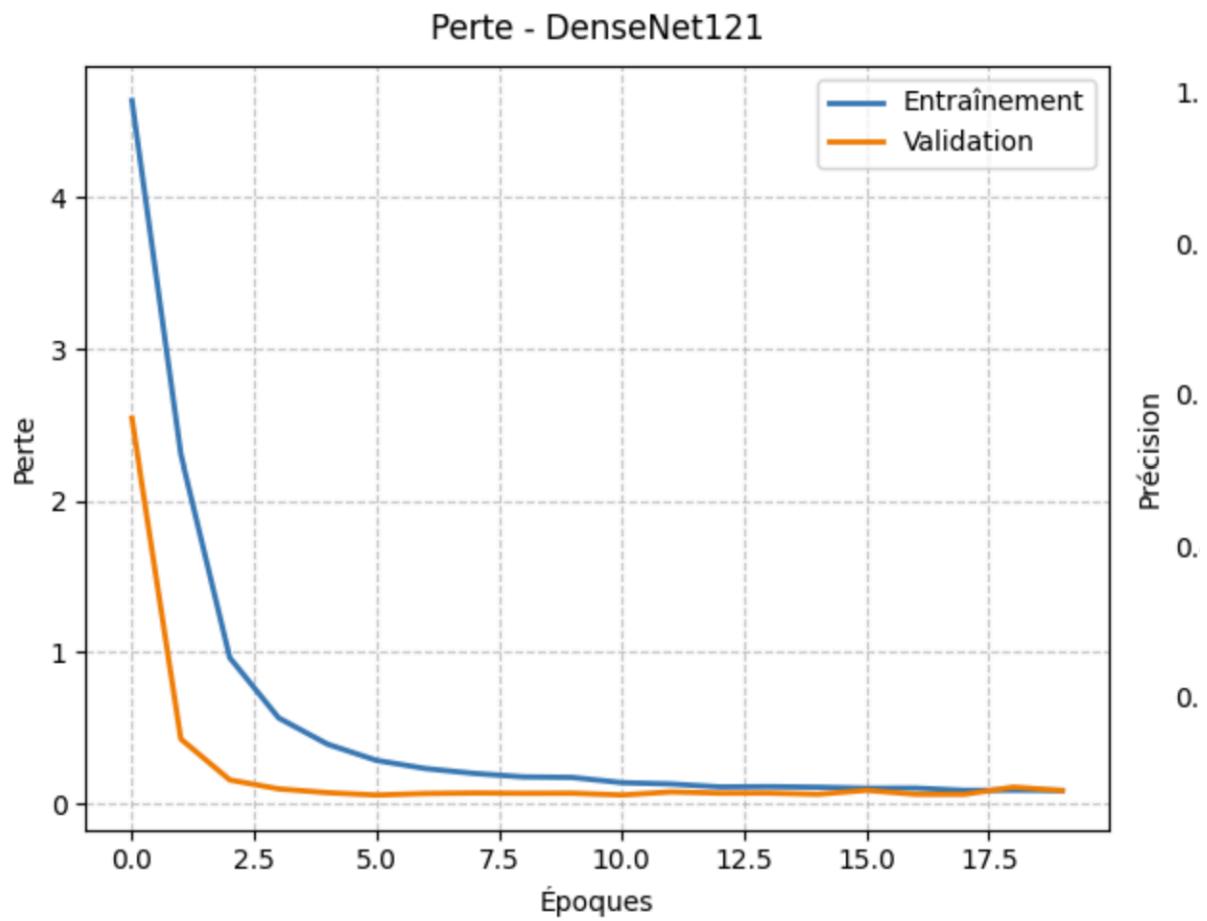


Figure 19: Illustration de courbe de perte de DenseNet121

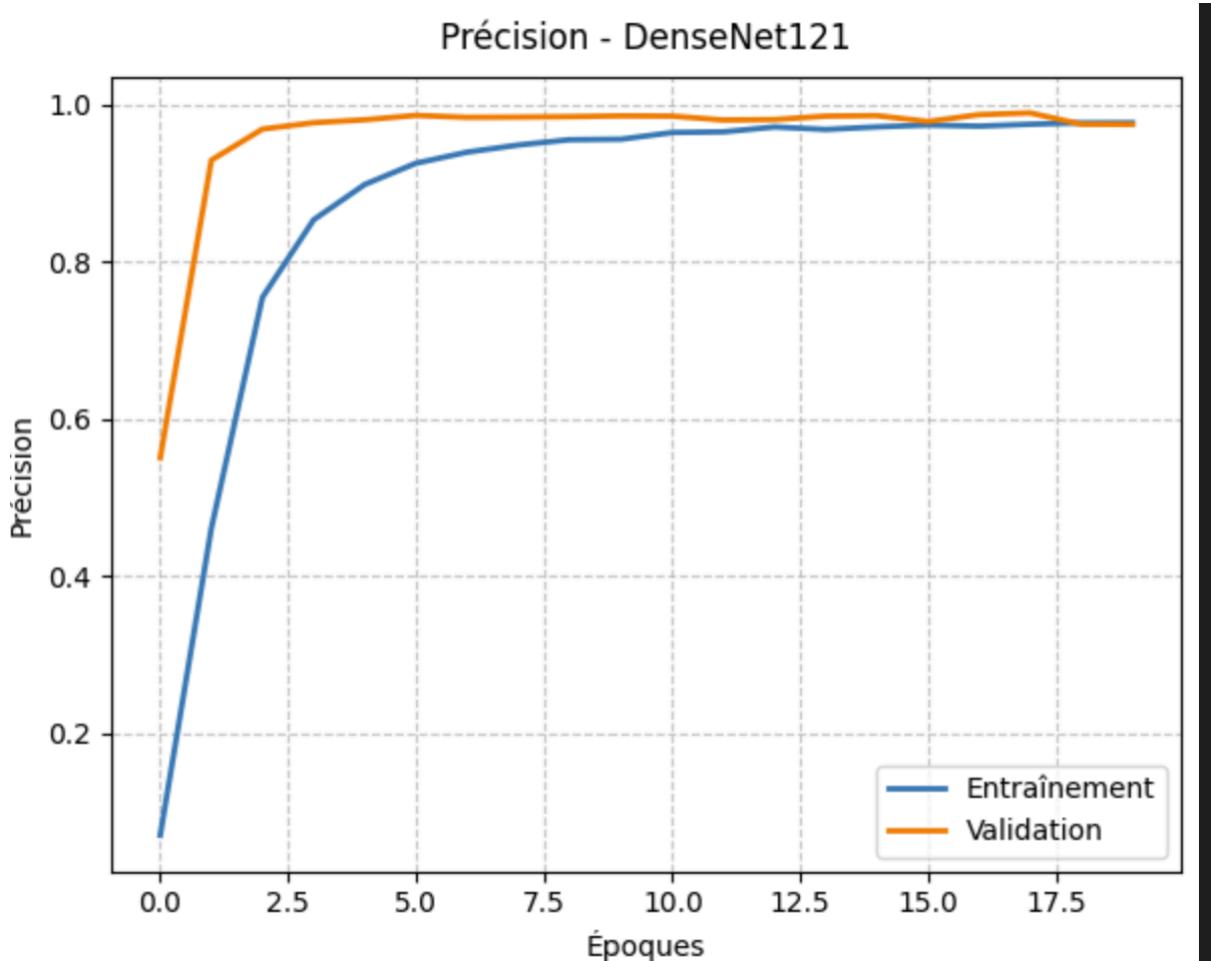


Figure 20: Illustration de courbe de précision de DenseNet121

Pour approfondir l'analyse des performances du modèle **DenseNet121**, les figures 19 et 20 illustrent l'évolution de ses performances au cours de 20 époques. La **figure 19**, consacrée à la perte, met en évidence une diminution rapide pour les ensembles d'entraînement et de validation, passant de plus de **4** à des valeurs proches de **0** dès les premières époques, traduisant une convergence efficace.

Quant à la **figure 20**, représentant la précision, elle montre une augmentation rapide atteignant près de **100%** pour la validation dès l'époque 5, suivie d'une stabilisation. L'absence d'écart significatif entre les courbes d'entraînement et de validation dans les deux figures confirme l'efficacité du modèle à généraliser, sans surapprentissage notable. Ces résultats mettent en lumière la robustesse de l'architecture DenseNet121, combinée à l'utilisation de l'optimiseur **Adam** avec un taux d'apprentissage ajusté, favorisant une convergence rapide et un apprentissage performant des motifs complexes, rendant ce modèle particulièrement adapté aux tâches de classification multi-classes.

6.VGG16/VGG19

6.1 Contexte

Les architectures VGG16 et VGG19 ont été développées par le Visual Geometry Group de l'Université d'Oxford. Ces architectures se distinguent par leur approche systématique de l'apprentissage profond, privilégiant une structure simple mais profonde qui a prouvé son efficacité dans de nombreuses tâches de computer vision. La différence principale entre ces deux versions réside dans leur profondeur : VGG16 comprend 16 couches au total, dont 13 couches convolutives et 3 couches entièrement connectées, tandis que VGG19 pousse cette architecture plus loin avec 19 couches, dont 16 convolutives, offrant ainsi une capacité accrue à capturer des caractéristiques complexes dans les images.

6.2 Architecture des Modèles

L'ensemble du réseau utilise exclusivement des filtres de convolution de taille 3x3, avec un stride d'un pixel. Ce choix de conception permet une extraction progressive et fine des caractéristiques, tout en maintenant un nombre de paramètres raisonnable. Le padding est configuré pour préserver les dimensions spatiales après chaque convolution, assurant ainsi une transition fluide entre les couches.

La réduction de la dimensionnalité spatiale est gérée par des couches de max pooling, utilisant des fenêtres de 2x2 pixels avec un stride de 2. Cette configuration permet de réduire progressivement la taille des cartes de caractéristiques tout en conservant les informations les plus pertinentes. Cette réduction progressive de la dimensionnalité spatiale, combinée à l'augmentation du nombre de filtres dans les couches plus profondes, permet au réseau de construire une représentation hiérarchique de plus en plus abstraite des caractéristiques de l'image.

La partie finale du réseau comprend trois couches entièrement connectées (fully connected). Les deux premières comportent chacune 4096 neurones, permettant une intégration complète des caractéristiques extraites par les couches convolutives. La dernière couche, adaptée à notre tâche spécifique, comprend 151 neurones correspondant à nos classes de Pokémons, utilisant une activation softmax pour produire une distribution de probabilités sur l'ensemble des classes.

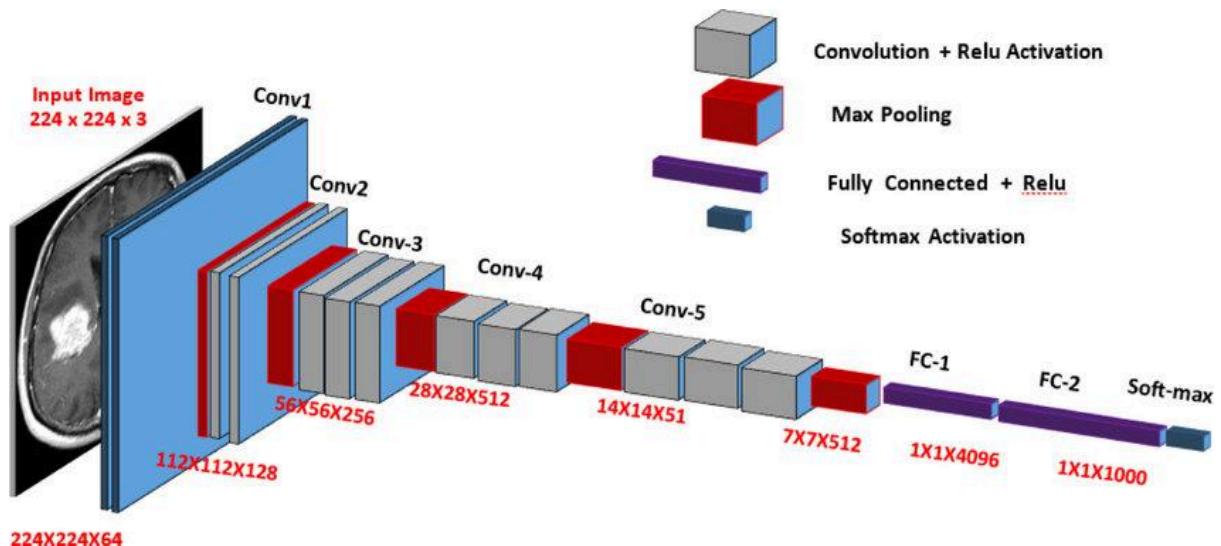


Figure 21: Modèle VGG 16

6.3 Application du Transfer Learning

Pour notre tâche de classification des Pokémon, nous avons testé une approche de transfer learning en utilisant les modèles VGG16 et VGG19 pré-entraînés sur la base de données ImageNet. Cette stratégie permet de bénéficier des connaissances générales acquises sur un vaste ensemble de données tout en adaptant le modèle à notre cas spécifique.

Notre processus commence par le chargement des modèles pré-entraînés, en excluant les couches fully connected originales qui étaient adaptées aux 1000 classes d'ImageNet. Nous avons délibérément choisi de conserver les poids pré-entraînés des couches convolutives, reconnaissant leur capacité à extraire des caractéristiques visuelles pertinentes et généralisables. Ces couches sont "gelées" pendant l'entraînement, ce qui signifie que leurs poids ne seront pas modifiés, permettant ainsi de préserver les filtres sophistiqués appris sur ImageNet.

6.4 Class Weighting

Nous avons aussi essayé d'appliquer une pondération des classes pour atténuer l'effet des déséquilibres dans le jeu de données. Cette méthode attribue des poids inversément proportionnels à la fréquence des classes lors du calcul de la fonction de perte. En donnant plus d'importance aux classes rares, le modèle est encouragé à accorder une attention équivalente à toutes les catégories, améliorant ainsi sa capacité à bien généraliser même sur les classes minoritaires. Cette approche réduit le biais en faveur des classes majoritaires, contribuant à une meilleure performance globale sur l'ensemble des classes..

6.5 Analyse des Résultats

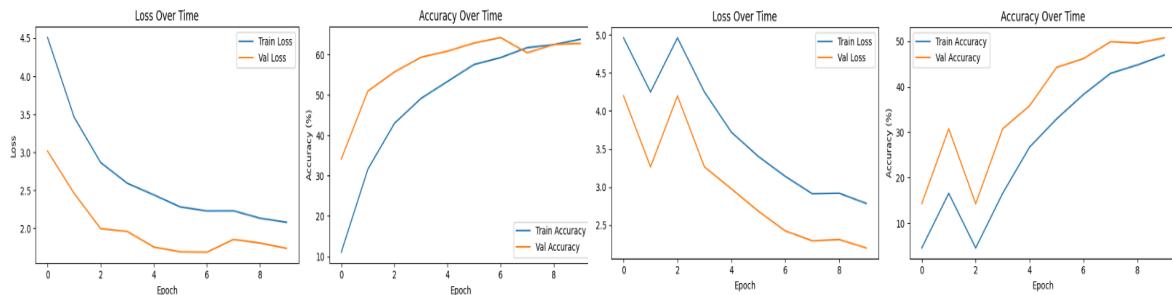


Figure 22: VGG 16

Figure 22: VGG 16 CW

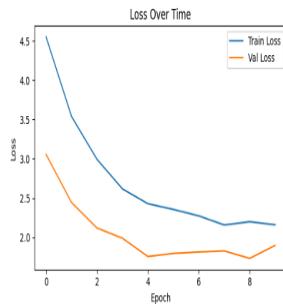


Figure 23: VGG 19

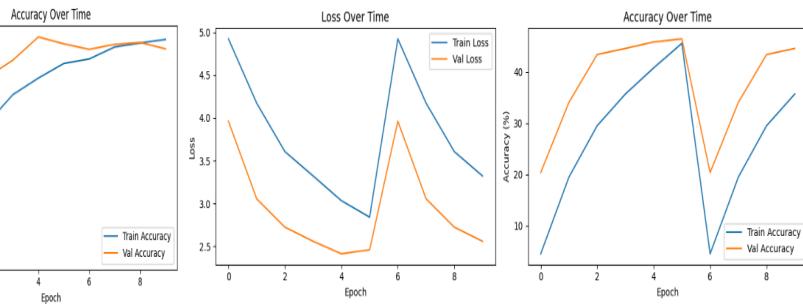
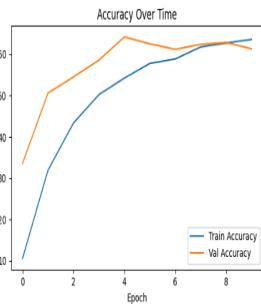


Figure 24: VGG 19 CW

L'analyse des performances de nos modèles VGG adaptés à la classification des Pokémon révèle des résultats encourageants.

On voit une progression stable de l'entraînement pour les deux modèles sans Class Weighting. La convergence s'effectue de manière régulière, sans présenter de signes marqués de surapprentissage, ce qui valide notre choix de paramètres d'entraînement. Les performances sur l'ensemble de test démontrent une bonne capacité de généralisation, suggérant que nos modèles ont réussi à capturer les caractéristiques distinctives des différentes espèces de Pokémon sans se limiter aux particularités du jeu d'entraînement.

Les courbes des correspondant au entraînement pour les versions de VGG avec du class weighing elle son plus chaotique mais on une progression globale satisfaisante a par un pic de perte de performance vers la fin de l'entraînement.

	VGG 16	VGG 19	VGG 16 CW	VGG 19 CW
Test Accuracy (en %)	65.15	74.09	52.60	46.75
Precision	0.78	0.80	0.70	0.68
Recall	0.65	0.74	0.53	0.47
F1-score	0.68	0.75	0.56	0.52

6.5.1 Comparaison entre VGG16 et VGG19

Globalement, VGG19 surpassé VGG16 sur l'ensemble des métriques. La précision de VGG19 atteint 74,09 %, contre 65,15 % pour VGG16, reflétant une meilleure capacité à prédire correctement les classes. De plus, le F1-score de 0,75 pour VGG19 est supérieur à celui de 0,68 obtenu par VGG16, indiquant un équilibre plus efficace entre la précision et le rappel. Cette amélioration s'explique par la profondeur accrue de VGG19, qui permet une extraction plus fine des caractéristiques complexes des images de Pokémon.

6.5.2 Impact de la pondération des classes (CW)

L'ajout de la pondération des classes a eu un impact négatif sur les performances des deux modèles. Pour VGG16, la précision diminue de 65,15 % à 52,60 %, tandis que le F1-score passe de 0,68 à 0,56. Une baisse similaire est observée pour VGG19, où la précision chute de 74,09 % à 46,75 %, et le F1-score passe de 0,75 à 0,52. Cela pourrait être dû à une surcompensation des classes rares ou à une distribution des poids inappropriée.

6.5.3 Conclusion sur les résultats

En résumé, VGG19 sans class weighting se distingue comme la version de VGG la plus performante, équilibrant correctement précision et rappel. Une approche future pourrait impliquer une meilleure calibration des poids ou l'exploration d'autres techniques pour gérer le déséquilibre des classes sans compromettre les performances globales, comme le l'over/under sampling.

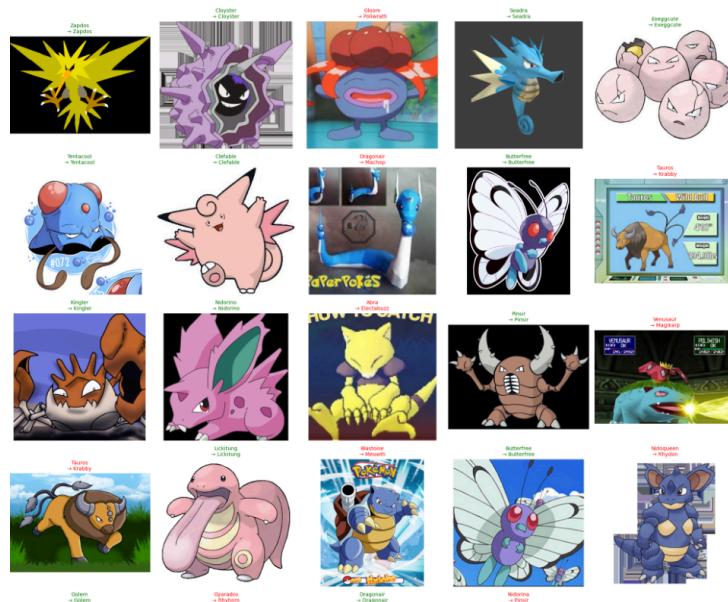


Figure 25: Résultats du modèle VGG 19 sans CW

7. Problèmes rencontrés

Un des principaux problèmes rencontrés a été le temps d'entraînement extrêmement élevé. Par exemple, le modèle EfficientNet a besoin d'environ 5 heures d'entraînement, tandis que ResNet50 a pris près de 11 heures. Malheureusement, j'ai oublié (Sherine) que c'était demandé d'enregistrer les modèles après les premiers entraînements, et je m'en suis rendu compte tardivement, la veille du rendu, en relisant la consigne de ce qu'il fallait soumettre. Cela m'a obligée à relancer les entraînements. Compte tenu du temps limité restant et du fait que l'entraînement des algorithmes demande énormément de temps, j'ai décidé d'envoyer mon code pour DenseNet121 à Jules afin qu'il puisse réentraîner le modèle de son côté afin de gagner du temps . En parallèle, j'ai pris en charge le réentraînement du CNN simple et de ResNet50 .

8. Répartition des tâches

Céline s'est occupée de la création et la préparation du dataset utilisé pour ce projet. Cette mission incluait la recherche et la collecte de données à partir de plusieurs sources, le tri et le nettoyage des images pour garantir leur pertinence, ainsi que la fusion de plusieurs datasets en un ensemble homogène et cohérent. Elle a également standardisé les images et appliqué des techniques d'augmentation des données pour renforcer la diversité visuelle. Enfin, elle a organisé les données en ensembles d'entraînement, de validation et de test, en assurant leur compatibilité avec les modèles que nous avons développés.

Joe s'est chargé de la comparaison entre deux modèles basés sur EfficientNet : le modèle EfficientNet-B0 standard et une version ajustée avec des modifications de paramètres et des transformations d'images pour optimiser les performances. Ces ajustements incluaient des changements sur les hyperparamètres et des prétraitements pour améliorer la qualité des données en entrée. Par la suite, les résultats obtenus ont été comparés à ceux d'un modèle CNN simple, mettant en évidence la complexité de la tâche et l'importance d'utiliser un modèle bien adapté pour atteindre des performances optimales (*le code de cette partie a été réparti sur trois notebooks*).

Jules a travaillé sur la partie concernant le transfert learning à partir des modèles VGG16 et VGG19, ainsi que sur la mise en place du class weighting pour ces deux modèles. Chaque version de ces modèles dispose d'un notebook associé et d'un fichier .pth contenant le modèle.

Sherine a développé un CNN simple en utilisant les couches fondamentales vues en cours, telles que les convolutions pour l'extraction de caractéristiques, le pooling pour la réduction dimensionnelle, et des couches denses pour la classification. Elle a également implémenté un modèle avancé basé sur l'architecture préentraînée ResNet50, en y intégrant des couches personnalisées, notamment la global average pooling et des fully connected adaptées au problème traité. En complément, elle a travaillé sur l'algorithme DenseNet121, en évaluant son performance à l'aide de métriques telles que la précision, le rappel et le F1-score, permettant ainsi une compréhension approfondie de l'architecture.

9. Conclusion

Pour conclure, ce projet a été une opportunité précieuse pour appliquer et analyser diverses architectures de réseaux neuronaux convolutifs (CNN) dans le cadre de la classification de Pokémon. Il a permis de mettre en évidence les forces et les limites des différentes approches testées. Le modèle CNN simple a servi de base pour comprendre les mécanismes fondamentaux de la classification, tandis que les architectures avancées, comme DenseNet121 et ResNet, ont démontré leur efficacité dans l'apprentissage de caractéristiques visuelles complexes. Parmi celles-ci, DenseNet121 s'est distingué par sa capacité à généraliser efficacement, obtenant des résultats remarquables sur la majorité des classes, ce qui a motivé notre choix comme modèle final à rendre.

Cependant, certains défis persistent, notamment pour les classes sous-représentées ou difficiles à distinguer visuellement, soulignant la nécessité d'améliorer la qualité et la diversité des données d'entraînement. Ces enseignements mettent en avant l'importance de continuer à optimiser les modèles et d'explorer des approches complémentaires, telles que l'augmentation des données ou l'utilisation d'architectures plus récentes. Ainsi, ce projet illustre à la fois la puissance des CNN et les

opportunités qu'ils offrent pour des tâches de classification exigeantes, tout en ouvrant la voie à des améliorations futures pour relever des défis encore plus complexes.

Bibliographie

- [1] GeeksforGeeks *EfficientNet Architecture* from
<https://www.geeksforgeeks.org/efficientnet-architecture/>
- [2] Danushi, D. (n.d.). *EfficientNet: Scaling Depth, Width, Resolution.*
<https://medium.com/@danushidk507/efficientnet-scaling-depth-width-resolution-11e2d4311357>
- [3] <https://www.innovatiana.com/post/discover-resnet-50>
- [4] <https://www.geeksforgeeks.org/densenet-explained/>

SOURCE : Figure 21

[https://www.researchgate.net/figure/GG-16-CNN-model-architecture-layer-wise_fig8_346259768]