BUT SCIENCE DES DONNÉES

DECOUST, DELEZINIER

RAPPORT

PROJET PERCEPTRON

DISCRIMINATION D'UNE IMAGE

Campus
de Niort

ANNÉE 2025/2026 - S5

Table des matières

I. Introduction 3

II. Approche basée Descripteurs 3

1. Calcul des descripteurs 3

2. Système de discrimination basée structure Full-Connected 4

III. Approche Deep 8

1. Modèle CNN 8

2. Construction du modèle 9

3. Configuration des paramètres 10

4. Entraînement et évaluation du modèle 11

5. Problématiques et solutions abordées 13

Conclusion 14

Annexes 15

I. Introduction

Dans le cadre du module *Introduction aux réseaux de neurones* du BUT Science des données de l'Université de Poitiers, nous avons réalisé un projet portant sur la discrimination automatique d'images.

L'objectif principal de ce travail est de concevoir un système capable de reconnaître la classe d'une image inconnue parmi un ensemble de dix catégories (Jungle, Plage, Monuments, Bus, Dinosaures, Éléphants, Fleurs, Chevaux, Montagne et Plats), issues de la base de données d'images Wang.
Pour atteindre cet objectif, nous avons mis en œuvre et comparé deux approches complémentaires :

- Une approche dite *basée descripteurs*, reposant sur des caractéristiques visuelles pré-calculées (notamment le descripteur FCTH) et un modèle de type *Perceptron multi-couches*.
- Une approche *Deep Learning*, dans laquelle les descripteurs sont appris automatiquement par un réseau de neurones convolutif (CNN).

Ce projet nous a permis de nous familiariser avec les concepts fondamentaux du traitement d'images, du Machine Learning et du Deep Learning, tout en expérimentant la mise en œuvre pratique d'un système complet de classification supervisée. Le rapport qui suit présente les différentes étapes de développement, les choix techniques effectués, ainsi que l'analyse et la comparaison des performances obtenues.

II. Approche basée Descripteurs

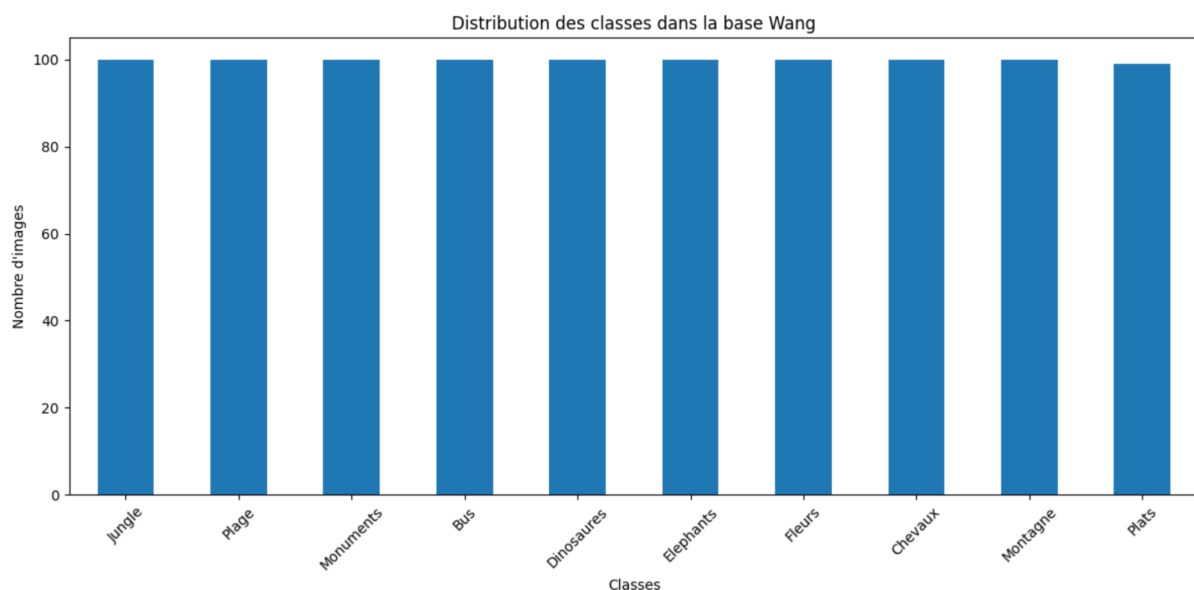
1. Calcul des descripteurs

La première étape de l'approche basée descripteurs est de créer un vecteur de label pour ranger les images dans le vecteur correspondant à leur catégorie d'appartenance. Comme expliqué en introduction, nous avons 1000 images, dans un dossier "wang", qui sont réparties par 100 dans 10 classes différentes. De plus, dans le dossier wang, qui était à notre disposition, les images sont déjà plus ou moins triées puisque les 100 premières images (0 à 99) appartiennent à la même classe, les 100 qui suivent (100-199) appartiennent également à la même classe et ainsi de suite. Nous avons aussi à notre disposition un fichier Excel, 'WangSignatures.xls', qui contient une ligne pour chaque image, où dans la première colonne on

retrouve son nom et dans celles qui suivent des données sur l'image. Pour la création de label je suis donc partie du fichier Excel.

```python
# === 1.1.1 CRÉATION DU VECTEUR DE LABELS ===
def create_label_vector(filenames):
    """
    Crée un vecteur de labels basé sur les noms de fichiers.
    La classe de l'image "i.jpg" est contenue dans label[i].
    Dans le dataset Wang, chaque groupe de 100 images consécutives forme une classe.
    """
    labels = []

    for filename in filenames:
        # Extraire le numéro du fichier (ex: "123.jpg" -> 123)
        image_number = int(filename.split('.')[0])

        # Dans le dataset Wang, les classes sont organisées par groupes de 100
        # Image 0-99 = classe 0, 100-199 = classe 1, etc.
        classe = image_number // 100
        labels.append(classe)

    return np.array(labels)

# Créer le vecteur de labels
labels = create_label_vector(filenames)
```

Voici le morceau de code de la fonction qui nous permet de créer les vecteurs. Dans un premier temps, nous chargeons le fichier qui contient les images, puis pour chaque nom d'image nous extrayons la partie numérique avant le point (ex : 155.png □ 155). Ensuite, nous divisons le nombre récupéré par 100 en division entière et le chiffre obtenu correspond à la classe de l'image (ex : 155 / 100 = 1,55 □ la partie entière = 1). Ensuite, pour associer à chaque classe chiffrée le bon nom de la classe, j'utilise un fichier json 'class_names.json' qui allie à chaque chiffre sa classe. J'enregistre tout ça dans le fichier 'labels.npy'. Pour finir, nous avons créé un graphique représentant la distribution des classes.



2. Système de discrimination basée structure Full-Connected

L'objectif de ce système de discrimination est de déterminer à partir d'un ensemble d'apprentissage la catégorie de l'image inconnue. La structure de discrimination à mettre en

place est de type Perceptron Multi-couches et nous avons utilisé le framework ‘Keras/TensorFlow’.

a. Préparation des données :

Dans un premier temps, nous avons commencé par créer l’ensemble d’apprentissage (train) et l’ensemble de test (test) qui représentent respectivement 80% et 20% en respectant les proportions par classe des labels (stratify=labels). Ensuite nous créons l’ensemble de validation (val) à partir de l’ensemble d’apprentissage en 20 %, 80 % et toujours en respectant les proportions mais celle d’apprentissage (stratify=y_train) cette fois-ci.

```
# === 3. SPLIT TRAIN / TEST (80% / 20%) ===
X_train, X_test, y_train, y_test = train_test_split(
    features,
    labels,
    test_size=0.2,
    stratify=labels,
    random_state=42
)
```

```
# === Split TRAIN / VAL (20% de train) ===
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train,
    test_size=0.2,
    stratify=y_train,
    random_state=42
)
```

Ensuite, nous avons normalisé nos données pour les mettre à la même échelle. Ce qui permet d’éviter la domination de certaines features, d’accélérer l’entraînement et de stabiliser l’apprentissage.

```
# === 4. NORMALISATION ===
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val   = scaler.transform(X_val)
X_test  = scaler.transform(X_test)
```

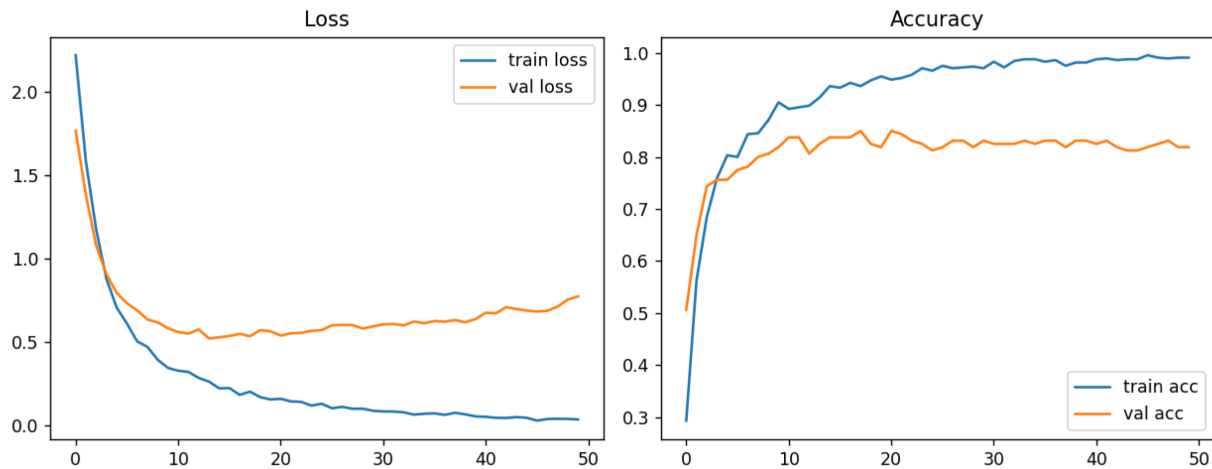
```
# === 5. CALCUL DES POIDS DE CLASSES ===
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train
)

class_weight_dict = {cls: float(w) for cls, w in zip(np.unique(y_train), class_weights)}
```

Pour finir avec la préparation des données, nous avons procédé au calcul des poids des classes afin de limiter le déséquilibre entre elles. Dans notre cas, nous ne sommes pas obligées de faire cette étape étant donné que nous savons que nos classes sont toutes composées de 100 images, ce qui signifie qu’elles sont équilibrées. Nous l’avons tout de même fait dans notre programme, et comme les classes sont équilibrées, les poids seront alors d’environ 1.

b. Choix du modèle :

Pour choisir le modèle le plus efficace, j’ai effectué plusieurs tests en adaptant les différents paramètres et le nombre d’epochs. J’ai commencé avec un modèle assez simple (Dense=128, Dropout=0.3, Dense=64 et dense=nb_classe (correspond au nombre de classes que l’on a soit 10)) sur 50 epochs. Voici la courbe de l’accuracy et du loss (la perte). On peut voir

qu'en entraînement le modèle a l'air plutôt bon, avec une accuracy de 0,9906 et une perte de 0,0370. Cependant en test, il y a un grand écart, l'accuracy est de 0,8188 et la perte est de 0,7745 ce qui fait un gap, une différence entre l'entraînement et le test, de 0,1719. Ce gap étant très largement supérieur à 0,10 indique qu'il y a un fort risque de surapprentissage, ce qui peut se confirmer par la différence de courbe entre train et val.



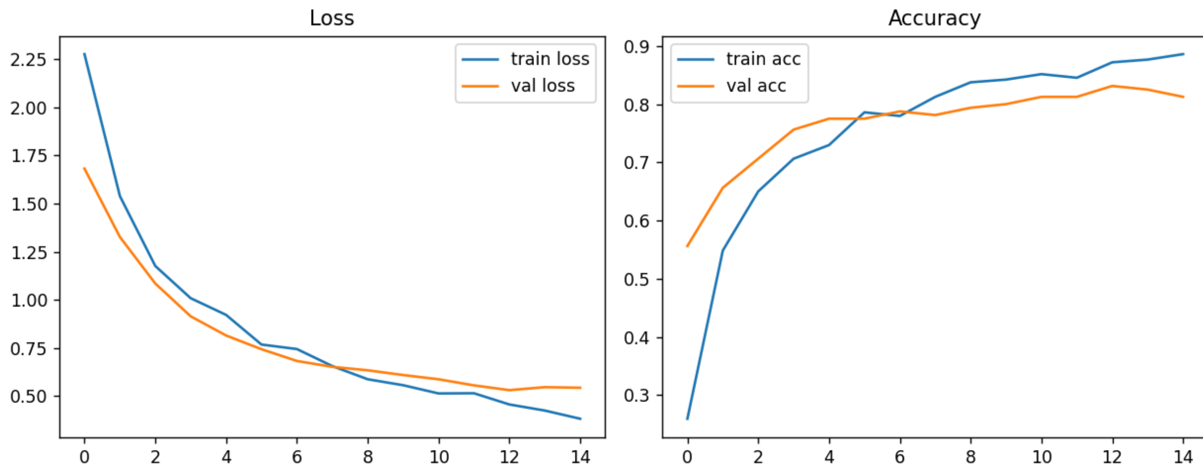
Pour améliorer notre modèle, nous avons donc pris la décision de diminuer le nombre d'epochs à 20 dans un premier temps parce qu'on peut voir sur les courbes qu'à partir de la 20ème epoch il n'y a pas de grande différence sur les résultats. Après plusieurs essais avec des réglages différents, nous avons conclu que le meilleur paramétrage pour obtenir le modèle le plus pertinent est celui-ci :

```
model = Sequential([

    Dense(256, input_dim=input_dim, kernel_regularizer=regularizers.l2(2e-5)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.35),

    Dense(128, kernel_regularizer=regularizers.l2(2e-5)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.25),

    Dense(64, kernel_regularizer=regularizers.l2(2e-5)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.15),

    Dense(nb_classes, activation='softmax')
])
```

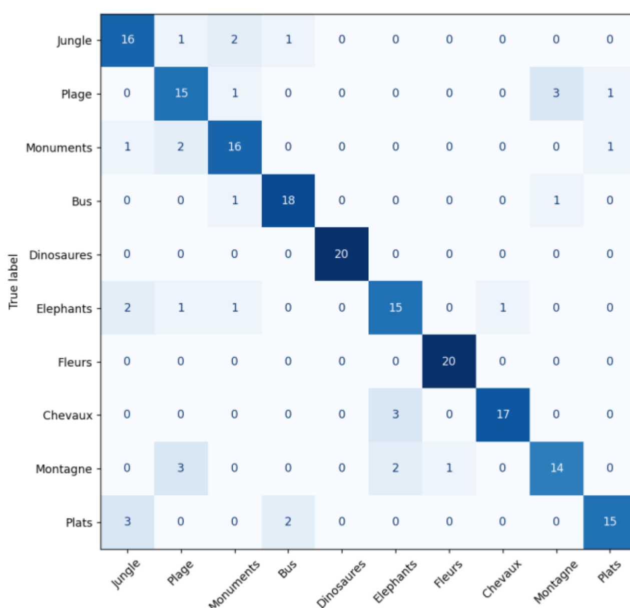
Nous avons trouvé que le modèle est le plus performant et le plus précis avec ces paramètres et sur 15 epochs.

c. <u>Analyse des résultats du modèle :</u>

Le modèle atteint une accuracy de 83 % sur le jeu de test, ce qui signifie qu'il classifie correctement 83 % des images. Ce résultat est très satisfaisant pour un problème de classification à 10 classes, où le hasard donnerait seulement 10 % de réussite.



Ces deux graphiques montrent bien que le modèle est assez efficace. Les deux courbes de loss nous confirment que le modèle est efficace puisqu'elles ont une pente légère avec des valeurs assez proches, ce qui démontre le fait qu'il n'y ait pas de surapprentissage majeur. Quant à elle, les d'accuracy nous confirment que le modèle apprend bien étant donné que la courbe en apprentissage a atteint 90 % tandis que celle de validation s'est stabilisée à 82 %. Il ne faudrait pas que cet écart ne s'agrandisse trop sinon on pourrait suspecter un surapprentissage de la part de notre modèle. Néanmoins, comme les courbes restent bien équilibrées il n'y a pas de surapprentissage critique.



Maintenant on peut constater que le modèle arrive à classer certaines classes à la perfection comme celles des « dinosaures » et des « fleurs ». Il y en a d'autres, qu'il arrive à prédire sans trop de difficulté : « bus », « chevaux », « monuments » et « jungles ». Pour d'autres classes, « plage », « éléphant », « plats » et « montagne ». On peut donc en conclure que le modèle a plus de mal à classer les images qui se rapportent au domaine de la nature.

Finalement, le modèle présente un bon équilibre entre performance (83 % d'accuracy) et généralisation, avec des marges d'amélioration identifiées principalement sur les classes naturelles similaires.

III. Approche Deep

1. Modèle CNN

Un réseau neuronal convolutif (CNN), est un type spécialisé d'algorithme d'apprentissage profond principalement conçu pour les tâches qui nécessitent la reconnaissance d'objets, notamment la classification, la détection et la segmentation d'images.

Dans cette partie, nous utilisons des images, et nous proposons une stratégie basée sur l'apprentissage, où les descripteurs sont construits par des couches de convolution.

Le CNN est utilisé afin de classer des images en catégories correspondant à leur numéro (divisé par 100). Notre pipeline de traitement et d'entraînement repose sur plusieurs étapes essentielles, directement visibles dans le code implémenté : l'organisation des données, le prétraitement (redimensionnement et normalisation), la division des données et la construction du modèle CNN.

Les images sont importées depuis le répertoire « Wang », qui contient l'ensemble des données. Chaque image est identifiée par un nom de fichier numérique, ce qui permet d'attribuer automatiquement une étiquette de classe. Le label est calculé en divisant le numéro de l'image par 100, ce qui regroupe les images par tranches de cent. Cette méthode permet d'organiser les données de manière simple et efficace sans modifier la structure physique des fichiers. L'importation des chemins des images et la création des labels sont réalisées par la fonction "charger_paths_et_labels".

```
# redimensionnement, normalisation
def parse_image(filename, label, img_size=(256, 256)):
    image = tf.io.read_file(filename)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, img_size)
    image = image / 255.0
    return image, label
```

Afin d'assurer une entrée homogène des données dans le réseau de neurones, toutes les images sont redimensionnées à une taille fixe de **256 × 256 pixels**. Cette étape est indispensable, car les images du jeu de données peuvent présenter des dimensions variables, incompatibles avec l'entraînement d'un modèle de Deep Learning.

Le redimensionnement est effectué automatiquement lors du chargement des images, après leur lecture et leur décodage, à l'aide des fonctions de traitement d'images de TensorFlow. En parallèle, les valeurs des pixels sont normalisées par une division par 255 afin de les ramener

dans l'intervalle **[0, 1]**. Ces opérations de redimensionnement et de normalisation sont réalisées par la fonction "parse_image(filename, label, img_size=(256, 256))". Cette normalisation améliore la stabilité numérique de l'apprentissage et facilite la convergence du modèle.

Après l'importation et le prétraitement des images, le jeu de données est divisé en deux sous-ensembles distincts afin d'évaluer correctement les performances du modèle. **80 % des données sont réservées à l'entraînement** et **20 % à la validation**. Cette séparation est réalisée de manière aléatoire mais stratifiée, ce qui permet de conserver une répartition équilibrée des classes dans chaque sous-ensemble. La division des données est effectuée à l'aide de la fonction "train_test_split(paths, labels, train_size=0.8, stratify=labels, random_state=42)" de la bibliothèque *scikit-learn*. Cette approche garantit une évaluation fiable du modèle en limitant les biais liés à une mauvaise distribution des classes.

2. Construction du modèle

Le modèle développé dans ce projet a pour objectif la **classification d'images en plusieurs catégories**, en s'appuyant sur les capacités des réseaux de neurones convolutionnels (CNN) à extraire automatiquement des caractéristiques pertinentes. Pour ce faire, nous avons utilisé l'API Keras de TensorFlow, qui permet une construction modulaire et lisible du pipeline d'apprentissage. L'architecture choisie suit une logique hiérarchique en couches, chaque étape contribuant à la complexité croissante des informations extraites à partir des images.

Les premières couches sont des **couches convolutionnelles**, responsables de la détection de motifs locaux tels que les contours, textures ou formes. Dans ce modèle, trois blocs convolutionnels sont utilisés avec un nombre croissant de filtres : 32 filtres pour la première couche, 64 pour la deuxième et 128 pour la troisième. Cette augmentation progressive permet au modèle de passer de la détection de motifs simples à l'extraction de caractéristiques de plus en plus abstraites et complexes, améliorant ainsi sa capacité à reconnaître des objets variés au sein des images.

Chaque couche de convolution est suivie d'une **couche de MaxPooling**, dont le rôle est de réduire la dimension spatiale des cartes de caractéristiques tout en conservant l'information la plus pertinente. Cette opération diminue le nombre de paramètres du réseau, accélère les

calculs et rend le modèle moins sensible aux variations locales, telles que de légères translations ou rotations des objets dans l'image.

```python
model.add(layers.Conv2D(32, (3, 3), input_shape=input_shape))  # Première couche de convolution
model.add(layers.Activation('relu'))  # Activation ReLU
model.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Pooling

model.add(layers.Conv2D(64, (3, 3)))  # Deuxième couche de convolution
model.add(layers.Activation('relu'))  # Activation ReLU
model.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Pooling

model.add(layers.Conv2D(128, (3, 3)))  # Troisième couche de convolution
model.add(layers.Activation('relu'))  # Activation ReLU
model.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Pooling

model.add(layers.Flatten())  # Aplatissement des sorties pour la couche dense
model.add(layers.Dense(128, activation='relu'))  # Couche dense avec activation ReLU
model.add(layers.Dropout(0.5))  # Dropout pour régularisation

# Couche de sortie avec activation Softmax
model.add(layers.Dense(num_classes, activation='softmax'))
```

Après les blocs convolutionnels, les cartes de caractéristiques sont **aplaties** et transmises à des **couches entièrement connectées (Dense)**. Ces couches permettent de combiner les informations extraites précédemment afin de produire la décision finale de classification. Pour limiter le risque de surapprentissage, une **couche Dropout** est ajoutée. Elle consiste à désactiver aléatoirement un certain pourcentage de neurones lors de l'entraînement, ce qui force le modèle à ne pas dépendre d'une caractéristique unique et améliore sa capacité de généralisation sur de nouvelles images.

Enfin, la couche de sortie applique une activation **Softmax**, qui convertit les résultats en probabilités pour chaque classe. Cela permet d'obtenir une prédiction interprétable, chaque image se voyant attribuer la classe la plus probable selon le modèle. L'ensemble de cette architecture constitue un **CNN simple mais robuste**, capable de réaliser une classification multi-classes efficace.

3. Configuration des paramètres

Optimiseur Adam et Fonction de Perte (Entropie Croisée Catégorielle) :

```python
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

L'optimiseur **Adam** a été choisi pour l'entraînement du modèle, car il combine les avantages de la descente de gradient adaptative et de la méthode du moment. Il permet une convergence rapide et stable du modèle, même avec un grand nombre de paramètres, comme c'est le cas avec un réseau de neurones convolutifs basé sur VGG16.

La fonction de perte utilisée est l'**entropie croisée catégorielle sparse** (*Sparse Categorical Crossentropy*). Elle mesure l'écart entre les prédictions du modèle et les étiquettes réelles, codées sous forme d'entiers. L'objectif de l'entraînement est de minimiser cette perte afin d'améliorer la précision des prédictions.

<u>Nombre d'Époques :</u>

Le modèle a été entraîné sur un maximum de **20 époques**. Une époque correspond à un passage complet sur l'ensemble des données d'entraînement.

```
# Callback EarlyStopping pour limiter le sur-apprentissage
early_stop = EarlyStopping(
    monitor='val_loss',           # Surveille la perte de validation
    patience=2,                   # Arrête si pas d'amélioration pendant 2 époques
    restore_best_weights=True,    # Restaure les meilleurs poids
    verbose=1
)
```

Afin d'éviter le sur-apprentissage, un mécanisme d'**arrêt anticipé (Early Stopping)** a été mis en place. Celui-ci interrompt automatiquement l'entraînement lorsque la performance sur les données de validation n'évolue plus, tout en restaurant les meilleurs poids obtenus.

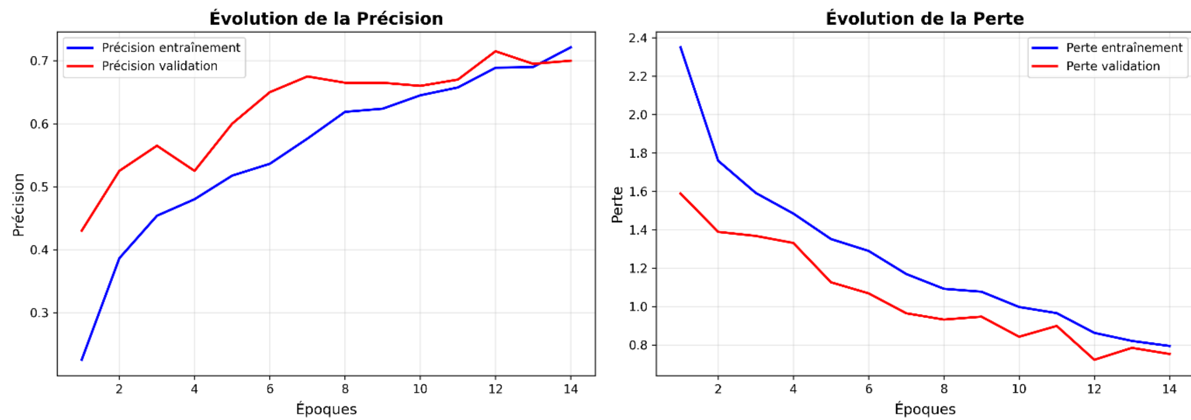
<u>Taille du Lot (Batch Size) :</u>

La taille du lot a été fixée à **32 images par itération**. Ce choix représente un compromis efficace entre la vitesse d'entraînement et la stabilité de l'apprentissage. Une taille de lot modérée permet également une meilleure généralisation du modèle sur des données non vues.

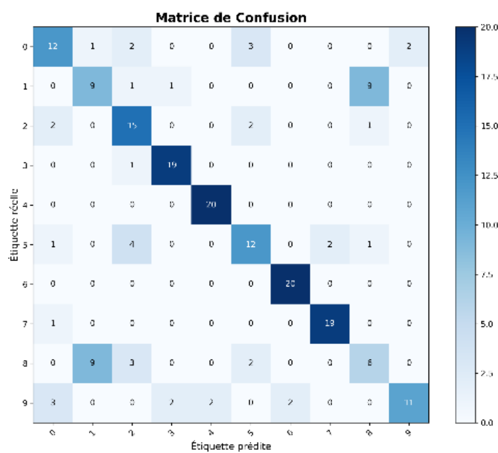
4. Entraînement et évaluation du modèle

Le modèle a été entraîné sur le jeu de données d'entraînement et évalué sur le jeu de validation, ce qui a permis de mesurer sa capacité à généraliser sur des images qu'il n'avait jamais vues. À chaque époque, le réseau ajustait ses poids pour améliorer progressivement sa capacité à prédire correctement la catégorie des images.

Pour suivre l'évolution des performances, des **courbes de précision et de perte** ont été tracées. La précision indique la proportion d'images correctement classées, tandis que la perte quantifie l'écart entre les prédictions et les labels réels. L'analyse de ces courbes permet de détecter d'éventuels problèmes de surapprentissage ou de sous-apprentissage et de juger de l'efficacité de l'apprentissage.



Les résultats indiquent une bonne performance du modèle sur les données d'entraînement avec une précision de 78.25%. Toutefois, la précision obtenue sur les données de test (71.50%) suggère un léger phénomène de surapprentissage (overfitting), ce qui signifie que le modèle se comporte mieux sur les données sur lesquelles il a été entraîné par rapport à des données inconnues.



En complément, une **matrice de confusion** a été générée afin d'évaluer la performance du modèle pour chaque classe. Elle permet d'identifier quelles catégories sont correctement ou incorrectement prédites, offrant une vision détaillée de la robustesse du modèle et des éventuelles confusions entre classes.

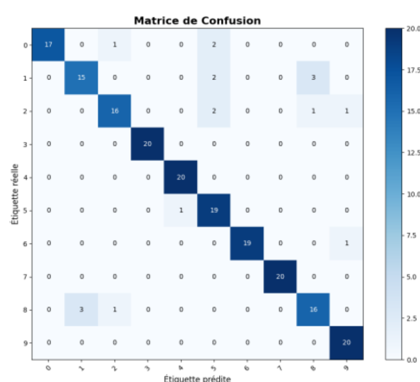
Les résultats montrent une forte concentration des prédictions sur la diagonale principale, indiquant que le modèle reconnaît correctement la majorité des classes, notamment *Jungle*, *Monuments*, *Bus*, *Dinosaures*, *Fleurs*, *Chevaux* et *Plats*. Toutefois, certaines confusions subsistent : les images de *Plage* et de *Jungle* sont parfois confondues avec la classe *Montagne*, probablement en raison de similarités visuelles entre les paysages naturels. Les classes *Éléphants* et *Chevaux* présentent également de légères confusions avec d'autres catégories animales, bien que leurs performances restent globalement satisfaisantes.

L'ensemble de ces étapes – entraînement, suivi des courbes de performance et analyse via la matrice de confusion a été **crucial** pour construire un modèle CNN capable d'**apprendre efficacement** et de **prédire avec précision** les catégories d'images.

5. Problématiques et solutions abordées

```python
def augment_image(image, label):
    # Rotation aléatoire (0°, 90°, 180° ou 270°)
    image = tf.image.rot90(image, k=tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32))

    # Zoom aléatoire via recadrage et redimensionnement
    if tf.random.uniform([]) > 0.5:
        # Zoom in: prend 80-100% de l'image puis redimensionne
        scale = tf.random.uniform([], 0.8, 1.0)
        h, w = tf.shape(image)[0], tf.shape(image)[1]
        new_h = tf.cast(tf.cast(h, tf.float32) * scale, tf.int32)
        new_w = tf.cast(tf.cast(w, tf.float32) * scale, tf.int32)
        image = tf.image.random_crop(image, [new_h, new_w, 3])
        image = tf.image.resize(image, [h, w])

    # Changements de luminosité (±30%)
    image = tf.image.random_brightness(image, max_delta=0.3)

    # Clip pour garder les valeurs entre [0, 1]
    image = tf.clip_by_value(image, 0.0, 1.0)

    return image, label
```

Pour améliorer la capacité de généralisation du modèle, nous avons appliqué des techniques d'**augmentation de données**. Chaque image du jeu d'entraînement a été dupliquée avec des rotations de 90°, 180° et 270°, ce qui permet de simuler différentes orientations des objets et d'augmenter la taille effective du jeu de données. De plus, la fonction « `augment_image(image, label)` » applique des transformations aléatoires telles que des rotations supplémentaires, des zooms et des variations de luminosité. Ces transformations conservent le contenu informatif des images tout en créant de nouvelles instances, rendant le modèle plus robuste aux variations et améliorant sa capacité à généraliser sur des images inédites.



Dans une approche plus avancée, nous avons également intégré un **modèle pré-entraîné**, le **VGG16,** via la fonction « `creer_modele_vgg16` ». Ce réseau, entraîné sur **ImageNet,** permet d'extraire des caractéristiques visuelles complexes dès les premières couches. En utilisant cette architecture, notre CNN peut apprendre plus rapidement et efficacement, en se concentrant sur l'adaptation de la tête de classification aux données spécifiques de notre projet.

Pour limiter le **risque de surapprentissage**, des techniques de régularisation ont été mises en place. La couche **Dropout**, présente dans notre modèle, désactive aléatoirement un pourcentage de neurones pendant l'entraînement, forçant le réseau à apprendre des caractéristiques générales plutôt que de s'appuyer sur un petit nombre de neurones spécifiques. La régularisation L2, elle reste une stratégie efficace pour réduire la complexité des réseaux et contrôler la magnitude des poids. Par ailleurs, nous avons ajusté des **hyperparamètres clés**, comme le nombre d'époques et le taux d'apprentissage, afin d'assurer une convergence stable et permettre au modèle d'apprendre les caractéristiques complexes des images de manière optimale.

Conclusion

Pour conclure, ce projet a permis de mettre en œuvre et de comparer deux approches complémentaires pour la classification d'images issues de la base Wang : une approche basée sur des descripteurs visuels pré-calculés associée à un Perceptron Multi-Couches (MLP), et une approche Deep Learning reposant sur un réseau de neurones convolutif (CNN) apprenant directement à partir des images brutes.

L'approche basée descripteurs s'est révélée particulièrement efficace, avec une accuracy de 83 % sur le jeu de test, démontrant la pertinence des descripteurs FCTH combinés à une architecture Fully Connected bien paramétrée. Ce modèle présente un bon compromis entre performance et généralisation, malgré certaines confusions observées entre des classes visuellement proches, notamment celles liées aux environnements naturels.

L'approche CNN, quant à elle, met en évidence la capacité des réseaux convolutionnels à extraire automatiquement des caractéristiques discriminantes à partir des images. Bien que plus complexe à entraîner et plus sensible au surapprentissage, cette méthode bénéficie fortement de techniques telles que l'augmentation de données, la régularisation par Dropout et l'utilisation de modèles pré-entraînés comme VGG16. Ces stratégies ont permis d'améliorer la robustesse du modèle et d'explorer le potentiel du Deep Learning dans un contexte de classification multi-classes.

Au-delà des résultats obtenus, ce projet constitue une première approche concrète et progressive du domaine du Machine Learning et du Deep Learning appliqués aux images, réalisée avant l'enseignement approfondi prévu au semestre 6. Il a permis de poser des bases solides en matière de compréhension des architectures de réseaux de neurones, de préparation des données, d'évaluation des modèles et d'analyse critique des performances, facilitant ainsi l'appropriation des concepts plus avancés qui seront abordés par la suite.

Les principales difficultés rencontrées concernent le surapprentissage, la similarité visuelle entre certaines catégories et le coût computationnel des modèles profonds. Néanmoins, ces limitations ont été partiellement surmontées grâce à une analyse fine des courbes d'apprentissage, à l'ajustement des hyperparamètres et à l'intégration de méthodes avancées issues du Deep Learning.

Pour prolonger ce travail, plusieurs perspectives peuvent être envisagées :

1. **Enrichissement des données** : augmenter la taille et la diversité du jeu d’images afin d’améliorer la généralisation des modèles.
2. **Architectures plus avancées** : tester des réseaux plus récents et performants tels que ResNet ou EfficientNet.
3. **Optimisation du modèle** : intégrer des techniques comme la recherche automatique d’hyperparamètres ou des mécanismes d’attention pour affiner l’extraction des caractéristiques pertinentes.

Ces améliorations permettraient d’approfondir l’étude des réseaux de neurones appliqués à la classification d’images et de renforcer les performances obtenues dans ce projet.

Annexes

Partie1.1.1.py : Calcul des descripteurs

Partie1.1.2.py : Système de discrimination basée structure Full-Connected

Partie1.1.3.py : Approche “Deep”

Partie1.1.4.py : Approche « Deep » modèle pré-entrainé