UFR des Sciences et Technologies

Master 2 des signaux et image en médecine

Rapport du Projet :

Segmentation d'Images d'un papillon zebra: Développement d'un Modèle de Segmentation Basé sur Mask R-CNN

Réalisé par Imane EL BLAGE

Encadré : Delphine Maugars

Table des matières

Introduction :

La vision par ordinateur, un sous-domaine de l'intelligence artificielle, s'intéresse à la capacité des machines à comprendre et analyser des images ou des vidéos. Parmi les tâches les plus complexes en vision par ordinateur figurent la **détection** et la **segmentation d'objets**, qui consistent respectivement à localiser les objets dans une image et à définir leurs limites précises au niveau des pixels. Ces techniques jouent un rôle clé dans de nombreuses applications, allant des véhicules autonomes, où il est crucial de détecter et segmenter des objets en temps réel, à la médecine, pour identifier des anomalies sur des images médicales.

Avec l'émergence des modèles d'apprentissage profond, comme **Mask R-CNN (Region-based Convolutional Neural Network)**, il est désormais possible de traiter efficacement ces tâches complexes. Mask R-CNN est une architecture avancée qui combine la détection d'objets et la segmentation d'instances, en fournissant des résultats précis, même dans des environnements visuellement complexes.

Dans ce projet, Mask R-CNN est utilisé pour entraîner un modèle capable de détecter et de segmenter des **papillons zebra** dans des images. Ces papillons, connus pour leurs motifs complexes et leurs contours irréguliers, représentent un défi particulièrement intéressant pour la segmentation. Les ailes des papillons zebra sont ornées de rayures noires et blanches caractéristiques, parfois accompagnées de détails fins et de variations subtiles de texture, ce qui les rend idéaux pour tester la robustesse et la précision du modèle.

I.2- Choix de l'Architecture Mask R-CNN

Mask R-CNN a été choisi pour ce projet en raison de ses capacités uniques :

- **Précision élevée** : Il combine la détection des boîtes englobantes avec la segmentation des objets, offrant des résultats précis à l'échelle des pixels.
- **Adaptabilité** : Il peut être facilement ajusté pour travailler sur des classes d'objets spécifiques, comme les papillons zebra, en modifiant les couches finales et en ajustant les hyperparamètres.
- **Performances robustes** : Grâce à sa capacité à gérer des motifs complexes et des objets aux contours irréguliers, Mask R-CNN est idéal pour segmenter les papillons zebra.

I.3- Création et Préparation des Données

Pour entraîner Mask R-CNN, il est essentiel de disposer d'un **jeu de données annoté avec précision**. Dans ce projet :

- Les images de papillons zebra ont été collectées à partir de sources publiques variées, en veillant à inclure différentes orientations, conditions d'éclairage, et contextes pour assurer la diversité.
- Les annotations ont été réalisées à l'aide de **Roboflow**, un outil performant pour créer des masques de segmentation précis. Chaque contour de papillon a été délimité point par point pour capturer les motifs complexes et les détails des ailes.
- Les annotations ont ensuite été exportées au **format COCO**, un standard en vision par ordinateur qui permet une compatibilité avec le modèle Mask R-CNN.

I.4- Objectif du Projet

Ce projet a pour objectifs de :

1. **Former un modèle Mask R-CNN** capable de détecter et segmenter des papillons zebra dans des images.
2. **Créer un dataset robuste et diversifié**, annoté avec précision pour capturer les caractéristiques complexes des papillons.
3. **Évaluer les performances du modèle** en utilisant des métriques telles que la précision, le rappel, le score F1, et le mAP (Mean Average Precision).
4. **Sauvegarder les résultats** et produire une analyse complète des performances pour des utilisations futures ou des comparaisons avec d'autres modèles.

En utilisant Mask R-CNN, ce projet démontre l'application concrète des algorithmes de vision par ordinateur à la segmentation d'objets complexes comme les papillons zebra. La prochaine section détaillera les étapes suivies pour la préparation du jeu de données et sa conversion au format COCO.

1. Création du DATASET :

1.1. Choix

Pour concevoir ce DATASET, j'ai décidé d'utiliser des images de papillons zebra. Ce choix repose sur la nécessité de tester la segmentation d'objets sur un sujet présentant des motifs complexes et des contours variés. Chaque image a été annotée avec soin à l'aide de masques de segmentation précis, créés manuellement à l'aide d'outils dédiés. L'objectif était d'assurer une délimitation exacte de chaque papillon pour obtenir des annotations de haute qualité.

Pourquoi ce choix ?

• **Complexité de la segmentation d'objets :** Les motifs des ailes de papillons, souvent riches en détails et en couleurs, associés à des contours parfois irréguliers, constituent un défi pour les modèles de segmentation. Tester sur un tel sujet permet d'évaluer la capacité du modèle à apprendre et à segmenter avec précision des objets dans des contextes variés.

• **Utilisation de Roboflow pour l'annotation :** J'ai choisi **Roboflow**, une plateforme reconnue pour son efficacité dans la création et la gestion de datasets annotés. Cet outil offre des fonctionnalités avancées de dessin, des options d'annotation manuelle précises et des formats d'exportation adaptés à divers **frameworks**, notamment COCO, ce qui simplifie l'intégration dans les pipelines d'apprentissage automatique.

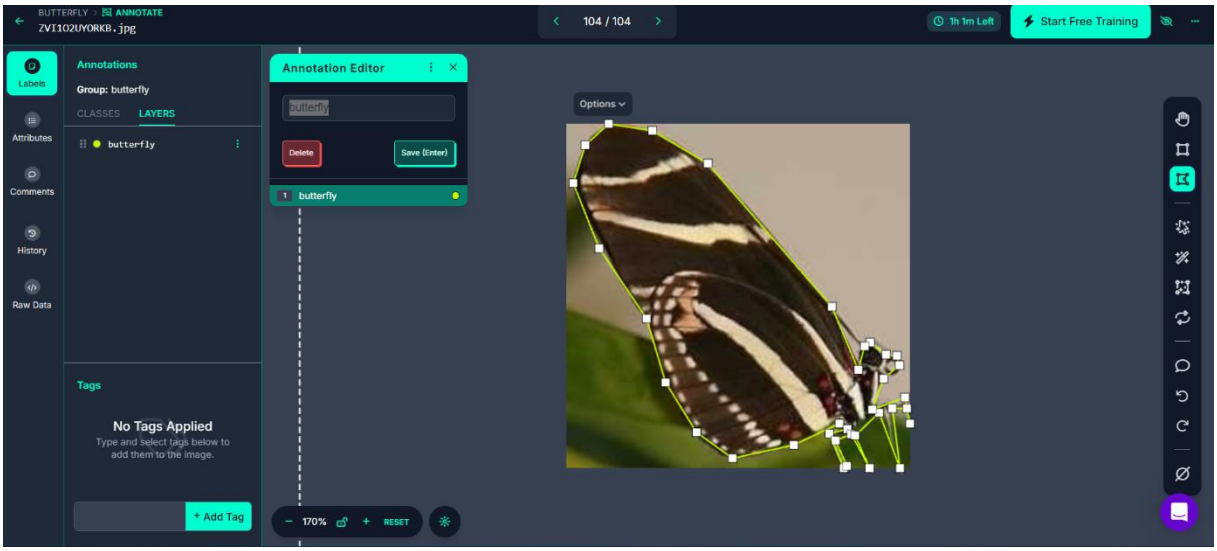
1.2. Méthodes utilisées

a) Collecte des images :

Les images de papillons ont été collectées à partir de sources publiques en ligne, en veillant à diversifier les espèces, les environnements, les angles de vue et les conditions d'éclairage. Cette diversité garantit un dataset varié et représentatif, favorisant une meilleure généralisation des modèles de segmentation.

b) Annotation des images avec Roboflow :

Pour chaque image, j'ai utilisé Roboflow afin de créer des masques précis en traçant des polygones point par point autour des contours du papillon. Cette méthode permet de capturer les motifs complexes et les détails spécifiques des ailes, produisant ainsi une annotation de haute qualité. L'utilisation de points de segmentation garantit une représentation fidèle et détaillée des papillons, essentielle pour tester la robustesse du modèle face à des textures complexes et des formes variées.



1.3. Conversion au format COCO

Après l'annotation des images dans Roboflow, j'ai procédé à l'exportation des annotations au format COCO. Ce format est un standard en vision par ordinateur, largement utilisé pour les tâches telles que la détection d'objets et la segmentation. Il offre une structure bien définie qui inclut :

• Les informations sur **les masques de segmentation**, permettant de délimiter précisément les objets annotés,

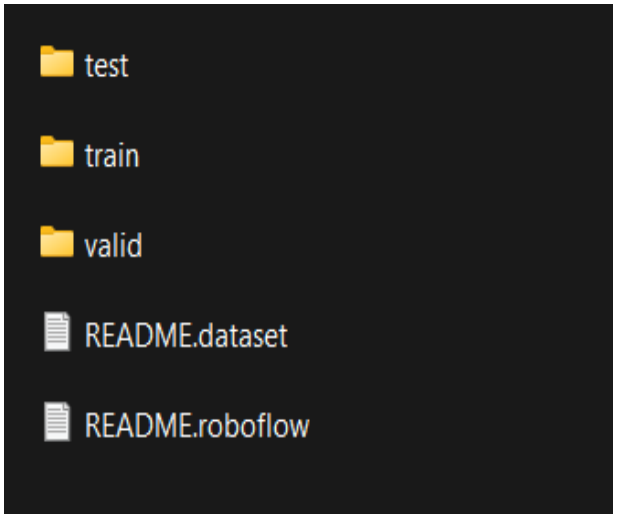
• **Les bounding boxes**, qui encadrent chaque objet pour un repérage global,

• Des métadonnées telles que les catégories des objets, les dimensions des images, et les identifiants uniques.

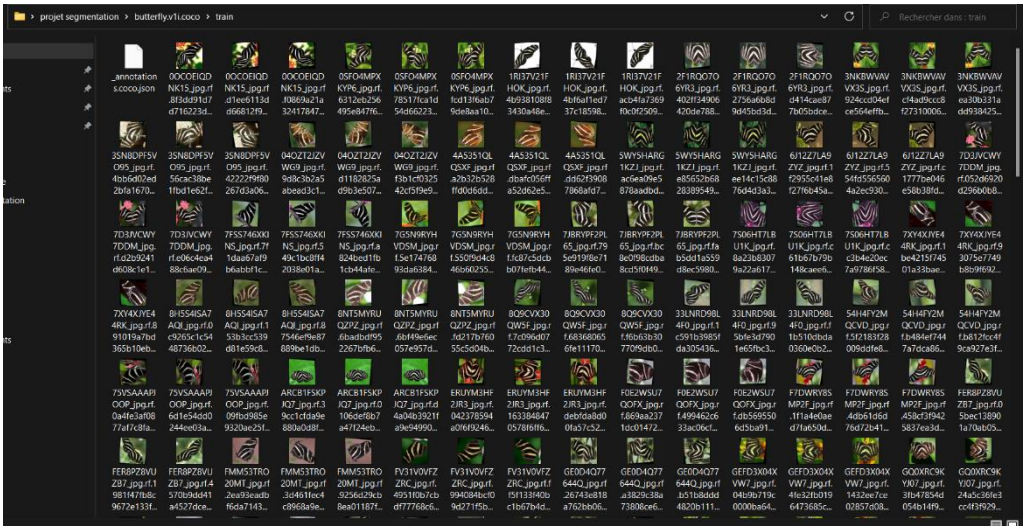
Ce choix garantit une compatibilité optimale avec de nombreux frameworks d'apprentissage automatique, facilitant ainsi l'entraînement et l'évaluation du modèle.

1.4. Répartition du DATASET

Afin de maximiser les performances du modèle et d'assurer une évaluation rigoureuse, le dataset a été divisé en trois sous-ensembles distincts : train set, valid set, et test set. Cette répartition a été effectuée avec soin pour garantir une représentativité équilibrée des données dans chaque ensemble.



a. Train Set (Ensemble d'entraînement)

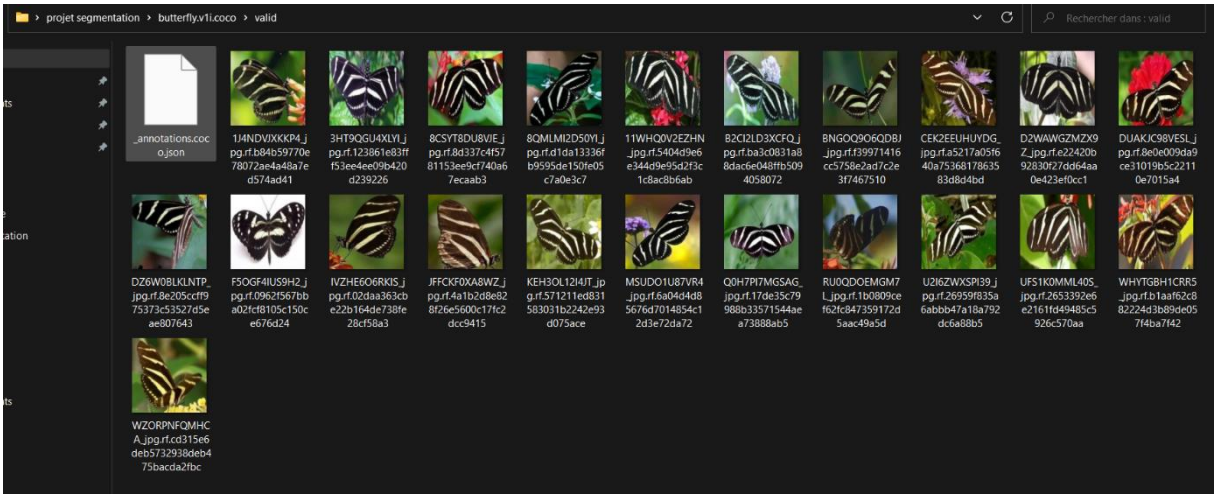


• 216 images ont été utilisées pour l'entrainement du modèle.

• Objectif : Cet ensemble est utilisé pour entraîner le modèle. Il permet au modèle d'apprendre les caractéristiques des papillons, notamment leurs contours, motifs et formes variées, afin d'optimiser ses paramètres.

• Importance : Ce lot de données est essentiel pour que le modèle développe sa capacité à effectuer des prédictions précises en se basant sur des exemples divers et riches.

b. Valid Set (Ensemble de validation)



• 22 images ont été utilisées pour la validation du modèle.

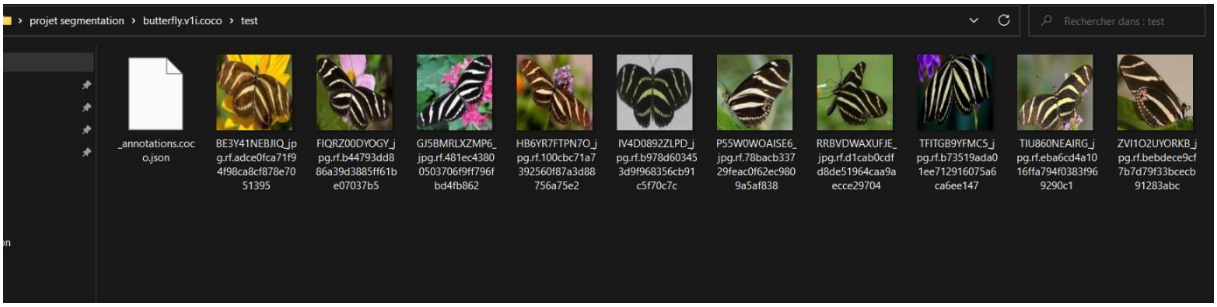
• Objectif : Cet ensemble est utilisé pour évaluer les performances du modèle pendant l'entraînement, mais sans qu'il apprenne directement à partir de ces images.

• Importance : La validation permet de :

• Détecter d'éventuels problèmes de sur-apprentissage (overfitting),

• Ajuster les hyperparamètres (par exemple, le taux d'apprentissage, le nombre d'époques, etc.) pour garantir un meilleur équilibre entre généralisation et précision.

c. Test Set (Ensemble de test)



• 10 images ont été mises de coté pour le test final du modèle.

• Objectif : Cet ensemble est réservé à l'évaluation finale des performances du modèle, une fois l'entraînement terminé.

• Importance : Le test final permet d'obtenir une évaluation objective et réaliste de la capacité du modèle à généraliser ses prédictions sur des données totalement inédites. Les résultats obtenus à ce stade ne sont pas utilisés pour affiner le modèle, garantissant ainsi une mesure fiable de sa performance.

Cette répartition stratégique assure une couverture complète des besoins d'entraînement, de validation et de test, tout en évitant les biais potentiels liés à un manque de diversité ou de représentativité dans les données.

2.Étapes de Développement sur Google Collab :

Google Collab a joué un rôle essentiel dans mon projet en fournissant un environnement de développement puissant avec un accès gratuit à des GPU. Cela a permis d'entraîner le modèle Mask R-CNN de manière efficace tout en optimisant les ressources pour traiter les données complexes du DATASET de papillons Zebra.

Voici une description détaillée de chaque étape réalisée sur Google Collab.

1-Préparation de l'Environnement

1.1. Clonage du Dépôt Mask R-CNN

```
[1] !git clone https://github.com/z-mahmud22/Mask-RCNN_TF2.14.0.git

Cloning into 'Mask-RCNN_TF2.14.0'...
remote: Enumerating objects: 246, done.
remote: Counting objects: 100% (54/54), done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 246 (delta 27), reused 15 (delta 15), pack-reused 192 (from 1)
Receiving objects: 100% (246/246), 74.90 MiB | 11.95 MiB/s, done.
Resolving deltas: 100% (75/75), done.
Updating files: 100% (86/86), done.
```

Ce code clone une version adaptée de Mask R-CNN compatible avec TensorFlow 2.14.0. Le dépôt contient les fichiers nécessaires, y compris les dépendances et les modules spécifiques au modèle.

2.1. Montage de Google Drive

```
[5] from google.colab import drive
    drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Connecte Google Drive à l'environnement Colab pour accéder facilement au dataset, aux poids pré-entraînés (mask_rcnn_coco.h5) et pour sauvegarder les résultats.

3.1. Installation des Dépendances

```
[ ] # Install the Mask R-CNN-specific dependencies
    !pip install -r requirements.txt
```

Installe les bibliothèques requises comme pycocotools pour gérer les annotations COCO et tensorflow pour l'entraînement.

2. Chargement et Préparation des Données

2.1. Classe ButterflyDataset

La classe ButterflyDataset, héritée de Dataset, gère le chargement des données annotées et leur préparation.

```
[10] import os
     import numpy as np
     from pycocotools.coco import COCO
     from mrcnn.utils import Dataset

     class ButterflyDataset(Dataset):
         def load_butterfly(self, dataset_dir, subset):
             """Load a subset of the butterfly dataset."""
             self.add_class("butterfly", 1, "butterfly")
             annotations_path = os.path.join(dataset_dir, subset, '_annotations.coco.json')
             coco = COCO(annotations_path)

             for image_info in coco.loadImgs(coco.getImgIds()):
                 self.add_image(
                     "butterfly",
                     image_id=image_info['id'],
                     path=os.path.join(dataset_dir, subset, image_info['file_name']),
                     width=image_info['width'],
                     height=image_info['height']
                 )
```

Fonctionnalités :

- Charge les annotations COCO pour chaque image dans le dataset.
- Ajoute les métadonnées des images, comme leur taille et chemin, pour une utilisation ultérieure par Mask R-C

```
        )

    def load_mask(self, image_id):
        """Load instance masks for a given image."""
        info = self.image_info[image_id]
        coco = COCO(os.path.join(os.path.dirname(info['path']), '_annotations.coco.json'))
        annotations = coco.loadAnns(coco.getAnnIds(imgIds=info['id']))

        masks = [coco.annToMask(ann) for ann in annotations]
        class_ids = [ann['category_id'] for ann in annotations]

        if not masks:
            return np.zeros((info['height'], info['width'], 0)), np.array([])

        return np.stack(masks, axis=-1), np.array(class_ids)

    def image_reference(self, image_id):
        """Return the file path of the image."""
        return self.image_info[image_id]['path']
```

Rôle : Crée des masques binaires pour chaque objet dans une image.

Fonctionnement :

- Convertit une annotation en un masque binaire.
- Empile tous les masques en un tableau multidimensionnel (hauteur x largeur x nombre d'objets).
- Gestion des cas vides : Si une image ne contient pas d'annotations, la méthode retourne un masque vide et un tableau vide de classes.

2.2. Création d'une Classe Personnalisée pour le Dataset

La classe ButterflyDataset a été définie pour gérer le jeu de données. Elle charge les images et annotations au format COCO et génère les masques binaires pour chaque papillon zebra.

```
[11] # Initialize and load the training dataset
dataset_train = ButterflyDataset() # Changed 'butterflyDataset' to 'ButterflyDataset'
dataset_train.load_butterfly(data_dir, 'train')
dataset_train.prepare()

loading annotations into memory...
Done (t=0.01s)
creating index...
index created!

[12] # Initialize and load the validation dataset
dataset_val = ButterflyDataset()  # Changed 'butterflyDataset' to 'ButterflyDataset'
dataset_val.load_butterfly(data_dir, 'valid')
dataset_val.prepare()

loading annotations into memory...
Done (t=0.00s)
creating index...
index created!

[13] # Initialize and load the testing dataset
dataset_test = ButterflyDataset() # Changed 'butterflyDataset' to 'ButterflyDataset'
dataset_test.load_butterfly(data_dir, 'test')
dataset_test.prepare()

loading annotations into memory...
Done (t=0.00s)
creating index...
index created!
```

- Étape 1 : Chargement des données d'entraînement et de validation à partir des sous-dossiers.
- Étape 2 : Appel à prepare() pour convertir les annotations JSON en structures exploitables par Mask R-CNN.

3. Configuration et Initialisation du Modèle

3.1. Définition de la Configuration

Une classe de configuration personnalisée a été créée pour adapter Mask R-CNN au dataset de papillons zebra. Les paramètres incluent :

```python
# Import necessary libraries
from mrcnn.model import MaskRCNN
from mrcnn.config import Config

# Define your custom configuration class
class CustomConfig(Config):
    NAME = "butterfly"  # Name of the dataset or task
    IMAGES_PER_GPU = 2  # Adjust this based on the memory of your GPU
    NUM_CLASSES = 2  # Including background (1) + butterfly (1)
    STEPS_PER_EPOCH = 100  # Set based on the number of images in your dataset
    VALIDATION_STEPS = 50  # Set validation steps based on the dataset size
    LEARNING_RATE = 0.001  # Learning rate for training
    IMAGE_MIN_DIM = 512  # Minimum image dimension for resizing
    IMAGE_MAX_DIM = 512  # Maximum image dimension for resizing
    GPU_COUNT = 1  # Number of GPUs (adjust if you have more)

# Instantiate your custom configuration
config = CustomConfig()
```

- **Nombre de Classes** : Inclut l'arrière-plan et la classe "papillon".
- **Images par GPU** : Optimisé en fonction de la mémoire GPU disponible.
- **Dimensions des Images** : Fixées à 512x512 pour normaliser les entrées.

3.2. Initialisation du Modèle

```python
# Step 1: Create the model in training mode
model = MaskRCNN(mode="training", config=config, model_dir='/content')

# Step 2: Load pre-trained weights (COCO weights) but exclude the conflicting layers
model.load_weights('mask_rcnn_coco.h5', by_name=True, exclude=["mrcnn_bbox_fc", "mrcnn_class_logits", "mrcnn_mask"])

# Step 3: Train the model (fine-tuning only the heads)
model.train(dataset_train, dataset_val, learning_rate=config.LEARNING_RATE, epochs=10, layers='heads')
```

Le code configure et entraîne un modèle Mask R-CNN pour segmenter des papillons zebra à partir de votre dataset annoté.

- ✓ **Etape 01 : Création du modèle** : Le modèle est initialisé en mode "training" avec une configuration adaptée (2 classes : arrière-plan et papillons zebra.
- ✓ **Etape 02 : Chargement des poids pré-entraînés** : Les poids COCO sont utilisés pour les couches génériques (détection de textures et de formes), tandis que les couches spécifiques au dataset (classification et segmentation) sont réinitialisées et prêtes à être ajustées.

- ✓ **Etape 03 : Entraînement des couches supérieures** : Seules les couches finales sont ajustées sur les données spécifiques de votre dataset. L'entraînement est effectué sur 10 époques, en utilisant 88 % des données pour l'entraînement et 8 % pour la validation.

Ce pipeline optimise le modèle pour apprendre efficacement à détecter et segmenter les papillons, en utilisant les données annotées et augmentées fournies via Roboflow.

3.3. Enregistrement et Visualisation de l'Historique d'Entraînement

```
history_path = os.path.join('/content', 'training_logs.txt')
with open(history_path, 'w') as log_file:
    log_file.write(str(model.keras_model.history.history))  # Save training history

# Step 5: Generate a Plot of Losses (Manual Implementation)
import matplotlib.pyplot as plt

loss = model.keras_model.history.history['loss']
val_loss = model.keras_model.history.history['val_loss']
epochs = range(1, len(loss) + 1)

plt.figure()
plt.plot(epochs, loss, 'bo-', label='Training Loss')
plt.plot(epochs, val_loss, 'r*-', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Le code enregistre l'historique d'entraînement (pertes d'entraînement et de validation) dans un fichier texte pour analyse ultérieure. Ensuite, il génère un graphique comparant ces pertes au fil des époques, permettant d'évaluer la progression de l'apprentissage et de détecter d'éventuels problèmes comme le surapprentissage ou une mauvaise généralisation.

4.Matrice de confusion et visualisation de prédictions :

Step04 : Génération de la matrice de confusion

Ce code calcule et visualise une matrice de confusion après l'entraînement pour évaluer les performances de classification du modèle. Cela inclut la création d'un modèle en mode inférence, la collecte des étiquettes réelles et prédites, et l'affichage de la matrice.

4.1 Création du modèle en mode inférence

```
# Step 4: Generate Confusion Matrix after training
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Step 4.1: Create a model for inference
class InferenceConfig(Config):
    NAME = "butterfly"
    NUM_CLASSES = 1 + 1  # Background + butterfly
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

inference_config = InferenceConfig()
inference_model = MaskRCNN(mode="inference", config=inference_config, model_dir='/content')

# Load the trained weights into the inference model
inference_model.load_weights(model.find_last(), by_name=True)
```

- Classe InferenceConfig : Spécifie les paramètres pour le modèle en mode inférence :
  - NUM_CLASSES : Définit deux classes : le fond et le papillon zebra.
  - IMAGES_PER_GPU : Une image par GPU est traitée en inférence pour maximiser la précision.
  - mode="inference" : Active uniquement les calculs nécessaires pour prédire les masques et les classes.
- Chargement des poids :
  - Les poids du modèle entraîné sont chargés via model.find_last() pour évaluer le modèle sur le dataset de test.

4.2 Collecte des étiquettes (vérités terrain et prédictions)

```python
# Step 4.2: Collect true and predicted labels
true_labels = []
pred_labels = []

for image_id in dataset_test.image_ids:
    # Load true labels (class IDs) from the dataset
    true_masks, true_classes = dataset_test.load_mask(image_id)

    # Predict the image using the inference model
    image = dataset_test.load_image(image_id)
    pred = inference_model.detect([image], verbose=0)[0]

    # Align true and predicted classes
    if len(true_classes) == len(pred['class_ids']):
        true_labels.extend(true_classes)
        pred_labels.extend(pred['class_ids'])
    else:
        # Handle mismatched cases
        true_labels.extend(true_classes)
        pred_labels.extend(pred['class_ids'])

        # Fill with '0' (background class) for unmatched entries
        if len(true_classes) > len(pred['class_ids']):
            pred_labels.extend([0] * (len(true_classes) - len(pred['class_ids'])))
        elif len(pred['class_ids']) > len(true_classes):
            true_labels.extend([0] * (len(pred['class_ids']) - len(true_classes)))
```

- Vérités terrain (true_labels) :
  - Les masques réels et les classes associées sont extraits via load_mask.
  - Ces données correspondent aux annotations originales du dataset COCO.
- Prédictions (pred_labels) :
  - Les masques prédits, boîtes englobantes et identifiants de classe sont générés via inference_model.detect.
- Alignement des étiquettes :
  - Les prédictions et les vérités terrain peuvent ne pas correspondre en nombre. Le code corrige cela en ajoutant des "0" (classe arrière-plan) pour les désalignements.

4.3 Génération et visualisation de la matrice de confusion :

```
# Step 4.3: Generate and plot the confusion matrix
cm = confusion_matrix(true_labels, pred_labels, labels=[0, 1])  # Add '0' for background
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['background', 'butterfly'])
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

- Matrice de confusion (confusion_matrix) :
  - Compare les classes prédites et les vérités terrain.
  - labels=[0, 1] : Inclut le fond (0) et le papillon (1).
  - Chaque cellule indique combien de fois une classe a été correctement ou incorrectement prédite.
- Affichage avec ConfusionMatrixDisplay :
  - Axes :
  - Ligne : Classes réelles (truth).
  - Colonne : Classes prédites (predictions).
  - Les couleurs (Bleues ici) indiquent l'intensité : une valeur élevée signifie de meilleures prédictions.
- Interprétation :
  - Diagonale principale : Correspond aux prédictions correctes (True Positives).
  - Hors diagonale : Correspond aux erreurs (False Positives ou False Negatives).

Step 05. Visualisation des prédictions (Succès et erreurs)

```python
# Step 5: Visualize test predictions (Successes and Mistakes)
from mrcnn.visualize import display_instances
from mrcnn import visualize

import random

# Select 5 random image IDs from the test dataset
for i in random.sample(list(dataset_test.image_ids), 5):
    image = dataset_test.load_image(i)  # Load the image
    results = inference_model.detect([image], verbose=0)  # Perform detection

    # Visualize the prediction
    visualize.display_instances(
        image,
        results[0]['rois'],
        results[0]['masks'],
        results[0]['class_ids'],
        dataset_test.class_names,
        results[0]['scores']
    )
```

- Sélection aléatoire d'images :
  - random.sample(dataset_test.image_ids, 5) : Sélectionne 5 images aléatoires du dataset de test.
- Détection des masques et classes :
  - Le modèle d'inférence prédit les masques (results[0]['masks']), les classes (results[0]['class_ids']), et les scores de confiance (results[0]['scores']).
- Affichage avec display_instances :
  - Superpose les prédictions sur les images originales :
    - Masques segmentés.
    - Boîtes englobantes.
    - Classes et scores de confiance.
- Utilité :
  - Permet de visualiser à la fois les succès et les erreurs du modèle sur les images de test.
  - Fournit une validation qualitative pour compléter les métriques numériques.

5. Calcul du Mean Average Precision (mAP) :

```python
from mrcnn.utils import compute_ap
from mrcnn import utils
import numpy as np

# Store Average Precision (AP) for each image
APs = []

# Iterate over all test images
for image_id in dataset_test.image_ids:
    # Load the image
    image = dataset_test.load_image(image_id)

    # Load ground truth data: masks and class IDs
    gt_mask, gt_class_ids = dataset_test.load_mask(image_id)

    # Generate ground truth bounding boxes (compute from masks if not pre-defined)
    gt_bbox = utils.extract_bboxes(gt_mask)

    # Run inference on the image
    results = inference_model.detect([image], verbose=0)
    r = results[0]

    # Skip images with no predictions
    if r['masks'].size == 0:
        continue

    # Ensure predicted masks are the same size as the ground truth masks
    pred_masks = r['masks']
```

```python
    # If the prediction masks are larger than needed, resize them to match ground truth
    if pred_masks.shape[0] != gt_mask.shape[0] or pred_masks.shape[1] != gt_mask.shape[1]:
        pred_masks_resized = []
        for i in range(pred_masks.shape[-1]):
            mask = pred_masks[:, :, i]
            mask_resized = cv2.resize(mask.astype(np.uint8),
                                      (gt_mask.shape[1], gt_mask.shape[0]),
                                      interpolation=cv2.INTER_NEAREST)
            pred_masks_resized.append(mask_resized)
        pred_masks = np.stack(pred_masks_resized, axis=-1)

    # Skip if there are no valid ground truth or predictions
    if gt_class_ids.size == 0 or r['class_ids'].size == 0:
        continue

    # Compute AP for the current image
    AP, precisions, recalls, overlaps = compute_ap(
        gt_bbox,          # Ground truth bounding boxes
        gt_class_ids,     # Ground truth class IDs
        gt_mask,          # Ground truth masks
        r['rois'],        # Predicted bounding boxes
        r['class_ids'],   # Predicted class IDs
        r['scores'],      # Predicted scores (was missing in original)
        pred_masks        # Predicted masks
    )

    APs.append(AP)
```

```
# Compute and display the mean Average Precision (mAP)
if len(APs) > 0:
    mAP = np.mean(APs)
    print(f"Mean Average Precision (mAP): {mAP:.4f}")
else:
    print("No valid predictions found for evaluation")
```

Ce code calcule le Mean Average Precision (mAP) pour évaluer les performances du modèle Mask R-CNN sur le dataset de test. Il charge les vérités terrain (masques, classes, boîtes englobantes), effectue des prédictions avec le modèle d'inférence, aligne les résultats, et calcule l'AP pour chaque image. Enfin, il agrège ces valeurs pour obtenir le mAP, une métrique clé pour évaluer la précision globale du modèle en détection et segmentation.

6. Evaluation des Performances du Modèle et Sauvegarde des Métrique

```
from mrcnn.utils import compute_ap
from mrcnn import utils
import numpy as np
from sklearn.metrics import precision_score, recall_score, f1_score
import cv2

# Store metrics for each image
APs = []
all_true_labels = []
all_pred_labels = []

# Iterate over all test images
for image_id in dataset_test.image_ids:
    # Load the image
    image = dataset_test.load_image(image_id)

    # Load ground truth data: masks and class IDs
    gt_mask, gt_class_ids = dataset_test.load_mask(image_id)

    # Generate ground truth bounding boxes
    gt_bbox = utils.extract_bboxes(gt_mask)

    # Run inference on the image
    results = inference_model.detect([image], verbose=0)
    r = results[0]

    # Skip images with no predictions
    if r['masks'].size == 0:
        continue
```

```
    # Ensure predicted masks are the same size as the ground truth masks
    pred_masks = r['masks']

    # Resize prediction masks if needed
    if pred_masks.shape[0] != gt_mask.shape[0] or pred_masks.shape[1] != gt_mask.shape[1]:
        pred_masks_resized = []
        for i in range(pred_masks.shape[-1]):
            mask = pred_masks[:, :, i]
            mask_resized = cv2.resize(mask.astype(np.uint8),
                                      (gt_mask.shape[1], gt_mask.shape[0]),
                                      interpolation=cv2.INTER_NEAREST)
            pred_masks_resized.append(mask_resized)
        pred_masks = np.stack(pred_masks_resized, axis=-1)

    # Skip if there are no valid ground truth or predictions
    if gt_class_ids.size == 0 or r['class_ids'].size == 0:
        continue

    # Compute AP for the current image
    AP, precisions, recalls, overlaps = compute_ap(
        gt_bbox,          # Ground truth bounding boxes
        gt_class_ids,     # Ground truth class IDs
        gt_mask,          # Ground truth masks
        r['rois'],        # Predicted bounding boxes
        r['class_ids'],   # Predicted class IDs
        r['scores'],      # Predicted scores
        pred_masks        # Predicted masks
    )
```

```
    APs.append(AP)

    # Prepare labels for sklearn metrics
    # Convert masks to binary labels for each pixel
    gt_mask_binary = np.any(gt_mask > 0, axis=2).astype(np.int32)
    pred_mask_binary = np.any(pred_masks > 0, axis=2).astype(np.int32)

    # Flatten masks for sklearn metrics
    gt_mask_flat = gt_mask_binary.flatten()
    pred_mask_flat = pred_mask_binary.flatten()

    # Store flattened labels
    all_true_labels.extend(gt_mask_flat)
    all_pred_labels.extend(pred_mask_flat)

# Convert lists to numpy arrays for sklearn metrics
all_true_labels = np.array(all_true_labels)
all_pred_labels = np.array(all_pred_labels)

# Calculate overall metrics
if len(APs) > 0:
    # Calculate mAP
    mAP = np.mean(APs)
    print(f"Mean Average Precision (mAP): {mAP:.4f}")

    # Calculate sklearn metrics
    precision = precision_score(all_true_labels, all_pred_labels, average='binary')
    recall = recall_score(all_true_labels, all_pred_labels, average='binary')
    f1 = f1_score(all_true_labels, all_pred_labels, average='binary')
```

```
    # Print all metrics
    print("\nOverall Metrics:")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")

    # Calculate per-image averages
    print("\nPer-image AP Statistics:")
    print(f"Min AP: {np.min(APs):.4f}")
    print(f"Max AP: {np.max(APs):.4f}")
    print(f"Median AP: {np.median(APs):.4f}")
    print(f"Standard Deviation AP: {np.std(APs):.4f}")

    # Print detailed distribution of APs
    percentiles = [25, 50, 75, 90, 95]
    print("\nAP Percentiles:")
    for p in percentiles:
        print(f"{p}th percentile: {np.percentile(APs, p):.4f}")
else:
    print("No valid predictions found for evaluation")
```

```
import json
from datetime import datetime

metrics = {
    "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
    "mAP": float(mAP),
    "precision": float(precision),
    "recall": float(recall),
    "f1_score": float(f1),
    "num_images_evaluated": len(APs),
    "ap_statistics": {
        "min": float(np.min(APs)),
        "max": float(np.max(APs)),
        "median": float(np.median(APs)),
        "std": float(np.std(APs)),
        "percentiles": {
            str(p): float(np.percentile(APs, p)) for p in percentiles
        }
    }
}

# Save metrics to JSON file
with open('evaluation_metrics.json', 'w') as f:
    json.dump(metrics, f, indent=4)
print("\nMetrics saved to 'evaluation_metrics.json'")
```

Ce code évalue les performances d'un modèle Mask R-CNN sur un dataset de test en calculant des métriques quantitatives, incluant :

- **Calcul de l'AP (Average Precision)** :

- Pour chaque image, il compare les vérités terrain (masques, classes, boîtes englobantes) et les prédictions générées par le modèle.
- L'AP est calculé et stocké dans une liste pour calculer ensuite le **Mean Average Precision (mAP)**, qui donne une vue globale des performances du modèle.

- **Calcul des métriques globales** :
  - Les masques réels et prédits sont convertis en formats binaires et aplatis pour calculer :
    - **Précision** : Proportion de prédictions correctes parmi toutes les prédictions.
    - **Rappel** : Proportion d'objets correctement détectés par rapport au total d'objets réels.
    - **F1-Score** : Harmonie entre précision et rappel.
- **Statistiques des AP par image** :
  - Min, max, médiane, écart-type et percentiles des AP sont affichés pour une analyse détaillée.
- **Sauvegarde des résultats** :
  - Toutes les métriques calculées (mAP, précision, rappel, F1, statistiques des AP) sont enregistrées dans un fichier JSON (evaluation_metrics.json) pour des analyses futures.

Ce code fournit une évaluation exhaustive des performances du modèle sur des tâches de segmentation et de classification des objets, permettant d'identifier ses forces et faiblesses.

UFR des Sciences et Technologies

Master 2 des signaux et image en médecine