Pneumonia_Analysis

General Strategy

I - Analyse des données.

On suit les étapes standards d'analyses des données :

Dans un premier temps, on se familiarise avec les données.

- On regarde les caractéristiques des données (width, height …)
- Les différentes distributions.
- Visualiser les différents types de radiographies.
- Compter la proportion de chaque type et leur nombre respectifs.
- On traite les données pour transformer les types.
- On enregistre et sauvergarde les données traitées dans des fichiers spécifiques.

II- Comparaison de modèles.

- define if using gpu, tpu, cpu.
- Fetch train et val paths.
- Commencer à reduce, shuffle, take.
- Batcher et prefetcher
- compute class weights
- train
- save training
- try multiple models
-
- Compare results and selDécrire les boites à moustaches des données.
- ect best model
-

Purpose of analysis in data_processing

dataprocessing.ipyng - analysis.ipynb

on importe la classe FileManager qui gère les dossiers de données du projet. Elle comprend des méthodes pour créer, récupérer et vérifier le jeu de données.

On importe également de la classe tf_utils les méthodes save_image_dataset_to_tfrecord et load_image_dataset_from_tfrecord qui respectivement enregistre un ensemble d'images dans un fichier TFRecord et charge un ensemble d'images à partir d'un fichier TFRecord.

D'abord, à travers la méthode fetch_full_dataset() qui se trouve dans la classe FileManager(), on récupère l'ensemble des données du répertoire des données raw se trouvant dans le répertoire des données intermédiaires, créant ainsi le jeu de données complet.

Ainsi, nous avons au préalable réparti chaque image dans une classe différente (normal, virus, bactérie).

Nous devons ensuite les rassembler pour les analysées toutes esemble. ***Stratégie derrière :*** Après avoir séparé les images en fonction de leur types, pour ne pas les analyser séparément puis faire une moyenne ensuite? Pourquoi les séparer puis les rassembler tout de suite après?

La séparation des images en fonction de leur type, c'est-à-dire en classes "normal", "virus" et "bactérie", permettrait d'organiser les données en groupes distincts pour une analyse plus spécifique. Chaque classe peut avoir ses propres caractéristiques ou comportements distincts. En les séparant, on peut appliquer des analyses, des prétraitements ou des modèles spécifiques à chaque classe pour mieux comprendre et traiter les données.

Or, cela ne correspond pas à notre besoin. Nous rassemblons les données à nouveau pour effectuer des analyses globales ou pour entraîner un modèle sur l'ensemble des données. La réunion des classes permet de créer un jeu de données complet, représentant l'ensemble des types de pneumonie que nous souhaitons classifier. Ainsi, nous pouvons appliquer aux différengtes catégories la même fonction. Par exemple, nous pouvons visualiser le déséquilibre de chaque catégorie pour déterminer si nous devons mesurer les métriques à travers des méthodes qui prennent en compte les déséquilibre ou non.

En somme, la réunion ultérieure des classes permet de traiter l'ensemble des données et de développer un modèle global pour la classification des pneumonies.

C’est pourquoi nous avons créer une liste metadata = [], auquel nous allons ajouter à travers la méthode append() l’ensemble des images avec leur label, caractéristiques et channels.

Avant de tirer des renseignements de ces images, il faut pouvoir les ajuster pour l’entraînement du CNN. Pourquoi? ***Stratégie :***

Le redimensionnement des images et l'ajustement du nombre de canaux sont des étapes de prétraitement courantes dans la préparation des données d'image pour les réseaux neuronaux convolutionnels (CNN). Voici pourquoi ces étapes sont importantes :

1. Des tailles d'image cohérentes : Les réseaux neuronaux convolutifs s'attendent généralement à ce que les images d'entrée aient une taille cohérente. Le redimensionnement des images permet de s'assurer que toutes les images de l'ensemble de données ont les mêmes dimensions, ce qui est nécessaire pour alimenter le CNN. Ce point est important car les CNN sont conçus pour traiter efficacement des données d'entrée de taille fixe. Le redimensionnement des images à une taille spécifique permet de définir en conséquence l'architecture du réseau, notamment le nombre de filtres et la taille des couches convolutives.
2. Réduire la complexité des calculs : En redimensionnant les images, vous pouvez réduire la complexité de calcul et les besoins en mémoire du CNN. Les grandes images ont plus de pixels, ce qui entraîne plus de calculs lors des passages avant et arrière du réseau. Le redimensionnement des images à une taille plus petite réduit le nombre de calculs nécessaires, ce qui permet d'accélérer l'apprentissage et l'inférence.
3. Ajustement des canaux : Les CNN travaillent généralement avec des images RVB qui comportent trois canaux (rouge, vert et bleu). Cependant, les images en niveaux de gris n'ont qu'un seul canal. En ajustant le nombre de canaux, vous vous assurez que l'entrée du CNN correspond au nombre de canaux attendu. Cette cohérence est essentielle pour que le CNN puisse apprendre des modèles et des caractéristiques significatifs à partir des données.

En redimensionnant les images et en ajustant le nombre de canaux, on rend les données d'image compatibles avec les exigences d'entrée du CNN, ce qui améliore l'efficacité du processus d'apprentissage du réseau.

Maintenant que nous avons rassembler et ajuster les images dans métadonnées, il est temps d’en tirer des renseignements.

Mais par où commencer pour tirer des insights?

Prochaine étape :

a) analyse de la distribution des variables continues.

L’étape qui suit le rassemblement de données est l’analyse de la distribution des variables continues.

Elle est importante car elle donne un aperçu des données sous-jacentes et de leurs caractéristiques. Elle fournit des informations précieuses sur les données, facilite la comparaison entre les groupes, aide à la détection des valeurs aberrantes et oriente les processus de prise de décision.

La compréhension de la distribution est utile à plusieurs égards :

- Statistiques descriptives : La distribution fournit des statistiques descriptives clés telles que la moyenne, la médiane, le mode, la variance et l'asymétrie, qui sont fondamentales pour résumer et comprendre la tendance centrale et la dispersion de la variable.
- Hypothèses des tests statistiques : De nombreux tests et modèles statistiques supposent certaines propriétés de la distribution de la variable, telles que la normalité. La vérification de la distribution vous permet d'évaluer si ces hypothèses sont respectées,

ce qui garantit la validité de l'analyse.

- Ingénierie des caractéristiques : La forme et les caractéristiques de la distribution peuvent guider les décisions en matière d'ingénierie des caractéristiques. Par exemple, si la variable suit une distribution asymétrique, il peut être nécessaire de la transformer (par exemple en prenant le logarithme) pour la rendre plus symétrique et améliorer les performances du modèle.
- Prise de décision : La compréhension de la distribution permet de prendre des décisions éclairées et de tirer des conclusions précises. Elle donne un aperçu de l'éventail des valeurs, de la probabilité d'occurrence des différentes valeurs et des valeurs aberrantes ou extrêmes potentielles qui pourraient avoir un impact sur les processus de prise de décision.

Quel est le meilleur et plus simple moyen d'y parvenir?

b) Violin plot

C'est le diagramme de violon. Il offre une représentation visuelle efficace de la distribution, permettant une compréhension globale des caractéristiques de la variable.

Un diagramme de violon est une visualisation utile pour plusieurs raisons :

- Visualisation de la distribution : Le diagramme en violon fournit une représentation claire et intuitive de la distribution d'une variable continue. Il vous permet de voir la forme, la dispersion et la tendance centrale des données pour différentes catégories ou groupes. Ces informations sont précieuses pour comprendre les caractéristiques et les tendances de la variable au sein de chaque groupe.
- Comparaison de groupes : Les diagrammes en violon permettent de comparer visuellement les distributions entre différentes catégories ou différents groupes. En superposant les violons ou en les plaçant côte à côte, vous pouvez observer les variations des données entre les groupes. Cela permet d'identifier les différences, les similitudes ou les modèles dans les distributions, et d'obtenir des informations sur les relations ou les tendances potentielles.
- Détection des valeurs aberrantes : Les diagrammes en violon permettent également d'identifier les valeurs aberrantes, c'est-à-dire les points de données qui s'écartent de manière significative du reste de la distribution. Les valeurs aberrantes peuvent fournir des informations précieuses sur des observations inhabituelles ou extrêmes au sein d'un groupe et peuvent justifier un examen plus approfondi.
- Représentation compacte : Les diagrammes en violon combinent les aspects des diagrammes en boîte et des diagrammes de densité de noyau en une seule visualisation. Cette représentation compacte fournit un résumé concis de la distribution, ce qui facilite la transmission des informations à d'autres personnes et la comparaison simultanée de plusieurs distributions.

A violin plot is a data visualization tool that combines aspects of a box plot and a kernel density plot. It is used to visualize the distribution of a continuous variable or a categorical variable across different categories or groups.

```
What we have plot is a violin plot : horizontal box corresponds to quantiles with the mean at the white dot. The colored space is the kerne
According to this violin plot, it appears that we will be able to resize images around
 without having to oversize too many images (oversizing can create blur in the image so we prefer to avoid it as much as possible).
```

c) Labels

Enfin, parlons des labels. La chose la plus importante que nous voulons savoir est s'il y a un déséquilibre entre les classes (c'est-à-dire si une classe a beaucoup plus de données que les autres). Le déséquilibre des classes peut rendre la classification plus difficile en biaisant le modèle vers la classe majoritaire, ce qui conduit à de mauvaises performances sur la classe minoritaire et à des mesures d'évaluation trompeuses.

Nous observons un déséquilibre des données. En effet, la classe des bactéries contient environ deux fois plus d'enregistrements que la classe des virus et celle des normaux !Nous devons donc gérer le déséquilibre des classes. Nous le ferons par :

- Ne pas choisir la précision comme métrique mais préférer la métrique du coefficient Kappa.
- Définir les poids des classes dans notre fonction de perte.

Training :

On réduit la taille des test et val splits car elle est dans sa forme actuelle beaucoup trop large. Ainsi:

```
test_split = 0.2
val_split = 0.25
```

Nous utiliserons la fonction image_dataset_from_directory pour séparer la formation et le test et les charger dans les datasets de tensorflow. Nous chargeons les images en mode rgb (3 canaux) et les redimensionnons.

Tout d'abord, nous comptons la taille des valeurs.
Ensuite, nous mélangeons les données de la taille du test complet par taille_de_tampon.
Taille_tampon : Spécifie le nombre d'éléments de l'ensemble de données à utiliser pour le brassage. Dans le cas présent, elle est fixée à num_train_img, qui représente vraisemblablement le nombre d'images d'apprentissage. Le fait de fixer la taille de la mémoire tampon au nombre total d'éléments garantit que tous les éléments sont mélangés.

val_ds = full_test_ds.take(val_size)
test_ds = full_test_ds.skip(val_size)

Le code `val_ds = full_test_ds.take(val_size)` crée un nouveau jeu de données `val_ds` en prenant les premiers éléments `val_size` du jeu de données `full_test_ds`. Cette méthode est couramment utilisée pour extraire un jeu de validation d'un jeu de données plus important.

Le code `test_ds = full_test_ds.skip(val_size)` crée un nouveau jeu de données `test_ds` en sautant les premiers éléments `val_size` du jeu de données `full_test_ds`. Ceci est typiquement fait pour créer un jeu de test séparé à partir des données restantes après avoir extrait le jeu de validation.

En divisant le jeu de données `full_test_ds` en `val_ds` et `test_ds`, vous pouvez avoir des jeux de données distincts pour la validation et le test. L'ensemble de validation (`val_ds`) est utilisé pour évaluer et ajuster le modèle pendant la formation, tandis que l'ensemble de test (`test_ds`) est utilisé pour évaluer la performance finale du modèle formé.

Puis, count_img_by_class() compte le nombre d'images dans chaque classe ou catégorie d'un ensemble de données. La fonction itère probablement sur l'ensemble de données, examine les étiquettes ou les affectations de classe de chaque image et calcule le nombre d'images pour chaque classe.

Finalement, nous enregistrons les ensembles de données de formation, de validation et de test dans un fichier différent du dossier traité.

src/data/evaluation.py

a) class FPRNormal(keras.metrics.Metric):

Cette classe commence par ajouter de poids aux vrais négatifs et aux faux négatifs car comme expliqué précédemment il y a un problème de classification qui est déséquilibré. En attribuant des poids aux vrais négatifs et aux faux négatifs, le modèle peut accorder plus d'importance ou pénaliser certains types d'erreurs au cours de la formation.

Déséquilibre des classes : Dans de nombreux problèmes de classification, la distribution des classes est déséquilibrée, ce qui signifie qu'une classe a beaucoup plus d'échantillons que les autres. Dans ce cas, le modèle peut être biaisé en faveur de la classe majoritaire et avoir du mal à apprendre et à prédire efficacement la classe minoritaire. En attribuant des poids plus élevés à la classe minoritaire (tels que les vrais négatifs ou les faux négatifs), le modèle est encouragé à accorder plus d'attention à ces échantillons et à améliorer ses performances sur la classe sous-représentée.

Il est important de noter que l'ajout de poids aux différents types d'erreurs doit être effectué avec une attention particulière et une connaissance du domaine. Les pondérations doivent refléter l'importance relative ou les coûts associés à chaque type d'erreur. En attribuant des poids appropriés, le modèle peut être entraîné à donner la priorité à certains types d'erreurs, à obtenir de meilleures performances globales et à s'aligner sur les besoins et les objectifs spécifiques du problème en question.

Ce module fournit des classes et des fonctions pour évaluer la performance d'un modèle de
modèle de classification multi-classes dans TensorFlow. Il inclut une classe métrique
personnalisée appelée FPRNormal, qui calcule le taux de faux positifs pour une classe
classe spécifique (dans ce cas, la classe "normale"). Il comprend également une classe Evaluation

qui peut calculer diverses mesures d'évaluation. Le module utilise `numpy`, `TensorFlow`, `Keras`, `TensorFlow Addons`, et `scikit-learn`.

Using `*kwargs` allows you to create functions that can accept any number of keyword arguments, providing flexibility and extensibility. It is particularly useful when you want to pass a variable number of keyword arguments to a function without explicitly defining all the parameter names.

```
self.false_positives = self.add_weight(name='fn',
                                       initializer='zeros')
```

On ajoute des poids pour les false positives ainsi que les true positives.

En utilisant self.add_weight(...), le code crée une variable de poids appelée "false_positives" avec une valeur initiale de zéro. Cette variable de poids peut ensuite être utilisée dans la couche ou le modèle personnalisé à diverses fins, telles que le suivi et la mise à jour du nombre de faux positifs au cours de la formation ou de l'évaluation.

self.add_weight(...) : Il s'agit d'une méthode fournie par la classe de couche ou de modèle TensorFlow pour ajouter des poids entraînables. Elle est utilisée pour définir et initialiser une nouvelle variable de poids. La méthode prend plusieurs arguments, tels que name (le nom de la variable de poids) et initializer (la stratégie d'initialisation de la variable).

def update_state(self, y_true, y_pred, sample_weight=None):

```
def update_state(self, y_true, y_pred, sample_weight=None):
        """
        Accumulates statistics for computing the FPR.

        Args:
            y_true (tf.Tensor): The true labels.
            y_pred (tf.Tensor): The predicted labels.
            sample_weight (tf.Tensor): Optional weighting of individual
                samples. Default is None.

        Returns:
            None
        """
        y_pred = tf.argmax(y_pred, axis=-1)
        y_true = tf.argmax(y_true, axis=-1)
        false_positives = tf.reduce_sum(
            tf.cast(tf.logical_and(tf.equal(y_pred, self.normal_idx),
                                   tf.not_equal(y_true, y_pred)),
                    dtype=tf.float32)
            )
        actual_negatives = tf.reduce_sum(
            tf.cast(tf.not_equal(y_true, self.normal_idx),
                    dtype=tf.float32)
            )
        self.false_positives.assign_add(false_positives)
        self.actual_negatives.assign_add(actual_negatives)
```

1. `y_pred = tf.argmax(y_pred, axis=-1)` and `y_true = tf.argmax(y_true, axis=-1)` are used to convert the predicted and true labels from one-hot encoded representations to class indices. The `tf.argmax()` function is used to find the index of the maximum value along the specified axis, which corresponds to the predicted or true class.

what is a class indices?

In the context of classification tasks, class indices refer to the numerical representations or labels assigned to different classes within a dataset. These indices are used to uniquely identify and distinguish between the classes.

When working with categorical data, it is common to encode the class labels using integer values. For example, in a dataset with three classes (e.g., "cat", "dog", "bird"), the class indices could be assigned as follows:

- "normal" → class index 0
- "bacteria" → class index 1

- "virus" → class index 2

The class indices serve as a convenient way to represent the different classes numerically, allowing machine learning models to process and make predictions on the data. During training and evaluation, the models typically output predictions as class indices, which can be mapped back to their respective class labels for interpretation.

The code snippet provided shows a method called `update_state()` that is used to accumulate statistics for computing the false positive rate (FPR) in a custom metric or loss function. Let's break down the code:

1. The method takes three arguments: `y_true` (the true labels), `y_pred` (the predicted labels), and `sample_weight` (optional weighting of individual samples).
2. `y_pred = tf.argmax(y_pred, axis=-1)` and `y_true = tf.argmax(y_true, axis=-1)` are used to convert the predicted and true labels from one-hot encoded representations to class indices. The `tf.argmax()` function is used to find the index of the maximum value along the specified axis, which corresponds to the predicted or true class.
3. `false_positives` is computed using `tf.reduce_sum()` and `tf.cast()`. It counts the number of false positives by checking if the predicted label is the normal class (`self.normal_idx`) and the true label is not equal to the predicted label.

Question : Why do that ?

Computing the number of false positives in a classification task serves as a useful metric to evaluate the performance of a model. False positives represent instances where the model incorrectly predicts a positive outcome (e.g., classifying a sample as abnormal or positive) when the true label is negative or normal.

The specific computation of false positives using `tf.reduce_sum()` and `tf.cast()` involves the following steps:

1. `tf.equal(y_pred, self.normal_idx)`: This checks if the predicted label is equal to the normal class index (`self.normal_idx`). The result is a boolean tensor where `True` indicates that the predicted label is the normal class and `False` indicates otherwise.
2. `tf.not_equal(y_true, y_pred)`: This checks if the true label is not equal to the predicted label. The result is a boolean tensor where `True` indicates that the true label is different from the predicted label, implying a misclassification.
3. `tf.logical_and(tf.equal(y_pred, self.normal_idx), tf.not_equal(y_true, y_pred))`: This combines the two boolean tensors from the previous steps using logical AND. The resulting tensor contains `True` values for instances where the predicted label is the normal class and the true label is not equal to the predicted label, indicating a false positive.
4. `tf.cast(..., dtype=tf.float32)`: This casts the boolean tensor into a float tensor, where `True` is represented as 1.0 and `False` is represented as 0.0.
5. `tf.reduce_sum(...)`: This computes the sum of all the elements in the tensor, resulting in the total count of false positives.

By counting the false positives, we gain insights into the model's tendency to incorrectly classify normal instances as abnormal. This information is valuable for evaluating the model's specificity and understanding its performance in differentiating between normal and abnormal classes.

Monitoring the false positive count during training and evaluation helps in assessing the model's ability to minimize false positives, which is particularly important in applications where false positives carry significant consequences or costs, such as medical diagnostics or anomaly detection.

1. `actual_negatives` is computed similarly using `tf.reduce_sum()` and `tf.cast()`. It counts the number of actual negatives by checking if the true label is not equal to the normal class.
2. `self.false_positives.assign_add(false_positives)` and `self.actual_negatives.assign_add(actual_negatives)` update the accumulated values of false positives and actual negatives. These values are stored in variables (`self.false_positives` and `self.actual_negatives`) defined within the class or object.

The purpose of this code is to accumulate statistics needed for computing the false positive rate (FPR) during the training or evaluation of a model. By updating the state of false positives and actual negatives, the metric can be computed later using these accumulated values.

Note: This code snippet provides only a part of the implementation. The complete implementation may include other methods or functions for computing the FPR or using these statistics in a larger context.

class Evaluation():

https://discuss.mxnet.apache.org/t/custom-mcc-loss-function/5857/2

"""

La classe Evaluation fournit des méthodes pour calculer les métriques d'évaluation et d'afficher les résultats. Elle fournit également la fonction de perte.

```
Attributs :
    étiquettes : Une liste d'étiquettes pour les différentes classes.
    full_metrics : Un dictionnaire contenant les métriques à calculer.
    loss_function : La fonction de perte utilisée dans le modèle.
    training_metrics : Une liste de métriques d'apprentissage.

Méthodes :

__decode_one_hot(y_true, y_pred) :
    Une méthode privée qui décode les étiquettes vraies et prédites encodées à un coup.
    prédites encodées à un coup.
compute_confusion_matrix(y_true, y_pred, display=False) :
    Une méthode qui calcule la matrice de confusion pour les étiquettes vraies et prédites.
    prédites.
get_training_metrics(metrics='BASE') :
    Une méthode qui renvoie les métriques d'apprentissage.
compute_full_metrics(y_true_oh, y_pred_oh, sample_weight=None,
display=False, digits=2) :
    Une méthode qui calcule les métriques d'évaluation complètes.
"""
```

```
def __init__(self, strategy, labels=['bacteria', 'normal', 'virus']):
        """
        Initializes a new instance of the Evaluation class.

        Args:
            labels (list): A list containing the label names for each
                class. Default is ['bacteria', 'normal', 'virus'].
        """
        self.strategy = strategy
        self.labels = labels
        self.full_metrics = {
            'Precision': keras.metrics.Precision,
            'Recall': keras.metrics.Recall,
            'Kappa': CohenKappa,
            'FPR normal': FPRNormal,
        }
        self.loss_function = 'categorical_crossentropy'
```

Dans le contexte de l'apprentissage profond et des réseaux neuronaux, "categorical_crossentropy" fait référence à une fonction de perte spécifique utilisée pour les problèmes de classification multi-classes. Elle est couramment employée lorsque la variable cible comporte plusieurs classes et que chaque échantillon d'entrée appartient à une seule classe.

La fonction de perte "categorical_crossentropy" est conçue pour mesurer la dissimilarité entre les probabilités de classe prédites et les probabilités de classe réelles. Elle quantifie la différence entre la distribution de probabilité prédite entre les classes et la distribution de probabilité réelle, généralement représentée sous la forme de vecteurs codés à un point.

Mathématiquement, la fonction de perte d'entropie croisée catégorielle calcule le logarithme négatif moyen de la vraisemblance des étiquettes de classe compte tenu des probabilités prédites. Elle encourage le modèle à attribuer correctement des probabilités élevées aux vraies classes et pénalise les écarts par rapport aux étiquettes des vraies classes.

Lors de l'utilisation de la fonction de perte "categorical_crossentropy", il est important de s'assurer que la variable cible est codée à un seul point, ce qui signifie que l'étiquette de la vraie classe de chaque échantillon est représentée sous la forme d'un vecteur binaire avec une valeur de 1 à l'index correspondant à la vraie classe et de 0 ailleurs.

L'optimisation de la fonction de perte "categorical_crossentropy" pendant la formation aide le réseau neuronal à apprendre et à améliorer ses performances de classification, ce qui lui permet de faire des prédictions précises pour plusieurs classes.

def get_training_metrics(self, metrics='BASE'):

The code you provided allows for three options (`'BASE'`, `'FULL'`, or `'FPR normal'`) to be specified for the `metrics` argument. Let's discuss the possible reasons for these options:

1. `'BASE'`: This option is likely provided to offer a minimal set of metrics for monitoring during training. It includes the Kappa metric, which is a measure of inter-rater agreement, commonly used in classification tasks. By including the Kappa metric, you can assess the performance of the model based on its agreement with the true labels.
2. `'FULL'`: This option is likely provided to include all available metrics. It allows for a more comprehensive evaluation of the model's performance during training. By including all available metrics, you can monitor multiple aspects of the model's performance and gain a more comprehensive understanding of its behavior.
3. `'FPR normal'`: This option specifically focuses on the false positive rate (FPR) for the normal class. The FPR is a metric that measures the rate at which the model incorrectly predicts the normal class as a positive (abnormal) class. This metric is particularly relevant in the context of binary classification problems, where distinguishing between the normal class and abnormal classes is crucial. By including this option, you can monitor and evaluate the model's performance specifically in terms of the FPR for the normal class.

The availability of these options suggests that the code is designed to provide flexibility in selecting the desired set of metrics for training monitoring. This allows you to tailor the monitoring to your specific needs and focus on the metrics that are most relevant to your problem domain.

```
def get_training_metrics(self, metrics='BASE'):
        """

        Returns a list of training metrics to monitor during training.

        Args:
            metrics (str or list, optional): The metrics to include. Can be
                one of 'BASE', 'FULL', or a list of specific metrics. 'BASE'
                includes only the Kapa metric and the FPR for the normal class,
                which is the default. 'FULL' includes all available metrics.
                A list of specific metrics can be passed as a list of strings.
                Defaults to 'BASE'.

        Returns:
            list: A list of the selected training metrics.
        """
        with self.strategy.scope():
            if metrics == 'BASE':
                training_metrics = [
                    self.full_metrics['Kappa'](num_classes=3, name='Kappa')
                ]

            if metrics == 'FULL':
                training_metrics = []
                for name, metric in self.full_metrics.items():
                    if name == 'Kappa':
                        training_metrics.append(
                            metric(num_classes=3, name=name)
                        )
                    elif name == 'FPR normal':
                        training_metrics.append(
                            metric(
                                labels=tf.constant(self.labels),
                                name=name
                            )
                        )
                    else:
                        training_metrics.append(
                            metric(name=name)
                        )
```

The information you provided appears to be part of a method called `get_training_metrics()`. Let's break down the code:

1. The method takes an optional argument `metrics`, which specifies the desired metrics to include. It can be one of three options:
    - `'BASE'`: Includes only the Kappa metric and the false positive rate (FPR) for the normal class.
    - `'FULL'`: Includes all available metrics.
    - A list of specific metrics: Allows you to pass a list of metric names as strings.

2. Within the method, the `self.strategy.scope()` context manager is used. This suggests that the code is executed within a distributed training strategy, which can be beneficial for training deep learning models on multiple devices or machines.
3. The code then checks the value of `metrics` and constructs the `training_metrics` list accordingly.
    - If `metrics` is `'BASE'`, it adds the Kappa metric (`self.full_metrics['Kappa']`) to the `training_metrics` list.
    - If `metrics` is `'FULL'`, it initializes an empty `training_metrics` list and iterates over the items in `self.full_metrics`. For each metric, it checks the name and appends the corresponding metric to the `training_metrics` list. Specifically, for the Kappa metric, it initializes it with `num_classes=3` (indicating a multi-class classification problem). For the FPR metric, it is initialized with the labels as a constant tensor (`tf.constant(self.labels)`) and the metric name.
    - If `metrics` is a list of specific metrics, it iterates over the metric names and appends the corresponding metric from `self.full_metrics` to the `training_metrics` list.
4. Finally, the method returns the `training_metrics` list, which contains the selected training metrics based on the specified `metrics` argument.

To get more context and information about the specific metrics (`Kappa` and `FPR normal`), you may need to refer to the definitions or implementations of these metrics elsewhere in the code.

pneumonia_detection/pneumonia/notebooks/models/file_comparison.ipynb

Tout d'abord, avant d'entraîner les modèles, nous aimerions conserver les données avant de les analyser. Dans le contexte de l'apprentissage automatique, la conservation des données fait référence au processus de préparation et d'organisation des données dans un format approprié avant d'entraîner les modèles ou d'effectuer toute analyse. Cela implique généralement de transformer les données dans un format compatible avec les algorithmes ou modèles d'apprentissage automatique spécifiques utilisés.

Une étape courante de la curation des données consiste à convertir les étiquettes en nombres entiers. Dans de nombreuses tâches d'apprentissage automatique, les étiquettes sont initialement représentées sous forme de variables catégorielles ou sous forme codée à un seul point. Cependant, la plupart des algorithmes et modèles d'apprentissage automatique s'attendent à ce que les étiquettes soient sous la forme de valeurs entières représentant la classe ou la catégorie.

La conversion des étiquettes en nombres entiers facilite le traitement et l'analyse ultérieurs, tels que l'apprentissage ou l'évaluation des modèles, et ce pour plusieurs raisons :

1. **Compatibilité des modèles** : De nombreux algorithmes et bibliothèques d'apprentissage automatique nécessitent des étiquettes entières pour l'apprentissage et l'évaluation. En convertissant les étiquettes en nombres entiers, les données peuvent être facilement utilisées avec un large éventail de modèles d'apprentissage automatique.
2. **Codage des étiquettes** : Les étiquettes entières permettent d'encoder la variable cible de manière compacte et efficace. Au lieu d'utiliser un codage à une touche ou de stocker les étiquettes catégorielles sous forme de chaînes, l'utilisation d'étiquettes entières réduit l'utilisation de la mémoire et simplifie les calculs.
3. **Interprétation des résultats du modèle** : Lors de l'évaluation d'un modèle formé, les étiquettes en nombres entiers facilitent l'interprétation des prédictions du modèle. Il est facile de faire correspondre les étiquettes entières prédites avec les noms de classes ou les catégories correspondantes, ce qui rend les résultats de l'évaluation plus compréhensibles.

En conservant les données et en convertissant les étiquettes en nombres entiers, les données sont mieux adaptées aux tâches ultérieures telles que l'apprentissage et l'évaluation des modèles, ainsi que l'analyse. Cela garantit la compatibilité avec les algorithmes d'apprentissage automatique et permet des calculs et une interprétation efficaces des prédictions du modèle.

Premièrement, on diminue la taille des jeux de données train et split parce que le jeux actuel est trop large. L'entraînement est trop lourd pour pouvoir le conduire tel quel. Cependant, il ne faut pas trop les réduire, car cela risquerait de rendre le résultat trop imprécis. En effet, plus on réduit le jeux de données, plus le risque est grand de perdre en acuité. Après avoir mené plusieurs essaies, nous en sommes venus à la conclusion que c'était les bonnes propotions.

```
BATCH_SIZE = 64
SMALL_TRAIN_SPLIT = 0.2
SMALL_VAL_SPLIT = 0.15
class_names = ['bacteria', 'normal', 'virus']
```

num_train_img = train_ds.reduce(0, lambda x, _: x + 1).numpy()

The purpose of calculating `num_train_img` using `train_ds.reduce()` is to determine the total number of images in the `train_ds` dataset.

1. The `reduce` function is called on the `train_ds` dataset. The `reduce` function applies a specified function to the elements of the dataset and reduces them to a single value.

1. The reduction function is defined as a lambda function `lambda x, _: x + 1`. This lambda function takes two arguments: `x` (the current accumulated value) and `_` (the current element of the dataset, which is ignored in this case). It increments the accumulated value `x` by `1` for each element in the dataset.

why shuffle the da

```
# Shuffle data
train_ds = train_ds.shuffle(buffer_size=num_train_img, seed=42)
```

Shuffling the data is done to introduce randomness and remove any potential biases or patterns that might exist in the order of the data. Here's why shuffling the data is beneficial:

1. Reduce bias: If the data is ordered in a certain way, such as all samples of one class followed by another class, it can introduce bias during training.
2. Enhance generalization: When the data is shuffled, the model is exposed to a varied distribution of samples during training. This helps the model to generalize better by learning patterns and features that are more representative of the overall dataset, rather than relying on the order or structure of the data.
3. Improve optimization: Shuffling the data can improve the optimization process during training. It prevents the model from getting stuck in local minima by providing a more diverse set of training examples in each batch, leading to better convergence and potentially faster training.
4. Minimize overfitting: Shuffling the data helps in reducing overfitting, as the model is less likely to memorize the specific order or patterns in the training data. It encourages the model to learn more robust and generalizable representations that can better generalize to unseen data.

In the code snippet provided, `train_ds.shuffle()` shuffles the training dataset `train_ds` using a buffer size of `num_train_img`. The buffer size indicates the number of elements from the dataset that can be randomly sampled at a time for shuffling. The `seed` parameter ensures reproducibility by setting the random seed for the shuffling operation.

Après on réduit considérablement la taille des données:

```
# Extract a sample
small_train_size = int(num_train_img * SMALL_TRAIN_SPLIT)
small_val_size = int(num_val_img * SMALL_VAL_SPLIT)
```

small_train_ds = train_ds.take(small_train_size)

This process can be useful when you want to create a smaller subset of the training dataset for various purposes, such as testing or debugging code on a smaller sample, or when you have resource limitations and want to work with a smaller dataset for faster experimentation.

1. The `.take()` method is called on `train_ds` with the argument `small_train_size`. This method takes the specified number of elements from the beginning of the dataset and creates a new dataset (`small_train_ds`) with those elements.

def count_img_by_class(dataset, class_names=class_names):

Qu’est-ce que count-img-by-class?

Cette méthode prend un dataset en paramètre, celui envoyé puis en class-names assigné par la valeur train_ds donnée juste auparavant. Il faut pas les prendre tout seul parce que ça contient trop de truc.

Assigner à num_img_by_classes → un nom qui est un numéro pour chaque classe (name dans class_name(label).

idx_label = np.nonzero(labels.numpy())[0][0]

Il est utilisé afin de trouver l’indexe du premier élément qui ne soit pas égale à zéro.

[0][0] est utilisé pour extraire les premiers éléments par les tuples retournés par np.zéro et ensuite extrait le premier élément de cette liste à l’indexe du premier élément non-zéro dans les labels.

idx_label va contenir l’indexe du premier du 1er élément non-zéro.

y_train_iterator = train_ds.map(lambda x, y: y).as_numpy_iterator()

In the code snippet provided, `train_ds.map(lambda x, y: y)` is used to create a new dataset that only contains the labels (target variable) from the `train_ds` dataset. The `.as_numpy_iterator()` function is then called on this new dataset to create an iterator that allows iterating over the labels as NumPy arrays.

```
y_train_iterator = train_ds.map(lambda x, y: y).as_numpy_iterator()
y_train = []
for one_vector in y_train_iterator:
    y_train.append(one_vector)
y_train = np.argmax(y_train, axis=1)
```

In the code snippet provided, `y_train_iterator` is an iterator that provides access to the labels of the training dataset (`train_ds`) as NumPy arrays. The code then iterates over the iterator and appends each label (one_vector) to the `y_train` list. Finally, the `y_train` list is converted to a NumPy array using `np.argmax()`.

The purpose of converting the labels to a NumPy array and using `np.argmax()` is likely to transform the labels from one-hot encoded form (binary vectors representing each class) to integer labels representing the class indices. This is often done to facilitate further processing or analysis, such as model training or evaluation, where integer labels are typically used.

The purpose of converting the labels to a NumPy array and using `np.argmax()` is likely to transform the labels from one-hot encoded form (binary vectors representing each class) to integer labels representing the class indices. This is often done to facilitate further processing or analysis, such as model training or evaluation, where integer labels are typically used.

class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)

```
Why use compute_class_weight() ?
https://stackoverflow.com/questions/69783897/compute-class-weight-function-issue-in-sklearn-library-when-used-in-keras-cl

We want to add weight balancing to the fitting.

`compute_class_weight` is useful in the context of imbalanced classification tasks where the distribution of classes in the training data i

The `compute_class_weight` function helps address this issue by computing class weights that can be used to give higher importance to the m

Here are a few reasons why `compute_class_weight` is useful:

1. **Handling class imbalance**: Class imbalance is a common challenge in real-world datasets, such as fraud detection, medical diagnosis,

2. **Improved model performance**: By assigning appropriate class weights, the model can achieve better performance on the minority class(e

3. **Simplified implementation**: `compute_class_weight` provides a convenient way to calculate class weights without manual calculation or

It's worth noting that using class weights alone may not always be sufficient to address class imbalance. Other techniques like data augmen
```

def save_history(model, history, training_time):

```
The `save_history` function is used to save the training history of a model, along with additional information such as the total training t

1. It takes three parameters:
   - `model`: The trained model.
   - `history`: The history object returned by the `fit` method during model training. It contains the training/validation loss and metric
   - `training_time`: The total time taken for training the model.

2. It creates an empty dictionary, `dic`, to store the information to be saved.

3. It assigns the `history.history` attribute to `dic['history']`. This attribute contains the loss and metric values from the training pro

4. It assigns the `training_time` value to `dic['training_time']`.

5. It calculates the average epoch time by dividing `training_time` by the number of epochs (`len(history.history['loss'])`) and assigns it

6. It defines the directory path (`history_dir`) where the history files will be saved. The path is typically within the model directory an

7. It creates the file path (`history_filepath`) by appending the model name to the history directory path.

8. It uses the `json.dump` function to write the dictionary `dic` into a JSON file at the specified `history_filepath`.

In summary, the `save_history` function saves the training history, training time, and average epoch time of a model in a JSON file for fut
```

def train_model(model, save=False):

1. It trains the model using the `fit` method, specifying the following:
   - `small_train_ds`: The training dataset.
   - `small_val_ds`: The validation dataset.
   - `epochs`: The number of training epochs.
   - `class_weight`: The class weights to be applied during training to handle class imbalance.
   - `callbacks`: Any additional callbacks to be used during training. In this case, it includes a `checkpoint_cb` callback, which is not defined in the given code snippet.

vgg16 = tf.keras.Sequential([], name = 'vgg16')

The manual construction of the VGG16 model architecture using the Keras Sequential API allows for flexibility and customization. By defining the model layer by layer, you have control over the specific architecture, input shape, and layer configurations. This flexibility enables you to modify or extend the model as needed, such as adding additional layers or modifying the existing layers to suit your specific task or requirements.

Additionally, in this case, the VGG16 model architecture is being used as a base and customized for a specific input shape (512x512x3) and number of output classes (3). The resizing and rescaling layers are added to preprocess the input images to match the expected input size of the pre-trained VGG16 model.

While manually constructing the model architecture provides flexibility, it is also possible to use pre-defined model architectures, such as the `tf.keras.applications.VGG16` model, which comes with the predefined architecture, weights, and input preprocessing already implemented. This can save time and effort, especially when the specific customization needs are not extensive.

1. `InputLayer`: Specifies the input shape of the model as (512, 512, 3), indicating images with a height and width of 512 pixels and 3 color channels (RGB).

The code snippet defines a VGG16 model architecture using the Keras Sequential API. Here's a breakdown of the layers in the model:

1. `InputLayer`: Specifies the input shape of the model as (512, 512, 3), indicating images with a height and width of 512 pixels and 3 color channels (RGB).

2. `Resizing`: Resizes the input images to (224, 224) using bilinear interpolation. This step is performed to match the input size expected by the pre-trained VGG16 model.
3. `Rescaling`: Scales the pixel values of the images to a range of [0, 1] by dividing them by 255. This is a common preprocessing step for neural network models.
4. `base_vgg16`: Refers to the VGG16 model architecture without the final fully connected layers. It is a pre-trained model that has learned representations from a large dataset.
5. `Flatten`: Flattens the output from the previous layer into a 1D vector, preparing it to be connected to fully connected layers.
6. `Dense`: Fully connected layers with 1024 and 512 units respectively, both using the ReLU activation function. These layers perform non-linear transformations on the input data.
7. `Dense`: Final dense layer with 3 units and softmax activation. This layer produces the output probabilities for the 3 classes (normal, virus, bacteria) based on the learned features.

The resulting model is named "vgg16" and can be used for tasks such as image classification.

```
history_path = {
    'vgg16' : zoidbergManager.model_dir / 'histories' / 'hty_smallds_vgg16.json',
    'resnet50' : zoidbergManager.model_dir / 'histories' / 'hty_smallds_resnet50.json',
    'inception_resnet' : zoidbergManager.model_dir / 'histories' / 'hty_smallds_inception_resnet.json',
    'xception' : zoidbergManager.model_dir / 'histories' / 'hty_smallds_xception.json',
    'efficientnetb0' : zoidbergManager.model_dir / 'histories' / 'hty_smallds_efficientnetb0.json'
}

checkpoint_path = {
    'vgg16' : zoidbergManager.model_dir / 'checkpoints' / 'ckpt_smallds_vgg16.h5',
    'resnet50' : zoidbergManager.model_dir / 'checkpoints' / 'ckpt_smallds_resnet50.h5',
    'inception_resnet' : zoidbergManager.model_dir / 'checkpoints' / 'ckpt_smallds_inception_resnet.h5',
    'xception' : zoidbergManager.model_dir / 'checkpoints' / 'ckpt_smallds_xception.h5',
    'efficientnetb0' : zoidbergManager.model_dir / 'checkpoints' / 'ckpt_smallds_efficientnetb0.h5'
}

sumup_result_df = pd.DataFrame(columns=['model','size','training_time','max CKS','max val_CKS'])
histories = {}
for model_name, path in history_path.items():
    with open(path) as file:
        history = json.load(file)
    histories[model_name] = history['history']
    sumup = []
    sumup.append(model_name)
    sumup.append(f'{checkpoint_path[model_name].stat().st_size / (1e6):.1f} MB')
    sumup.append(f"{history['training_time']:.3f} s")
    sumup.append(f"{np.max(history['history']['CKS']):.3f}")
    sumup.append(f"{np.max(history['history']['val_CKS']):.3f}")
    idx_model = list(history_path.keys()).index(model_name)
    sumup_result_df.loc[idx_model] = sumup

sumup_result_df.head()
```

The code snippet you provided is used to gather information and summarize the results of training different models. Here's what each part does:

- `history_path`: This dictionary contains the paths where the training history of each model is stored. It specifies the file location for each model's training history in the form of a JSON file.
- `checkpoint_path`: This dictionary contains the paths where the model checkpoints are stored. It specifies the file location for each model's checkpoint in the form of an H5 file.
- `sumup_result_df`: This is a Pandas DataFrame that will store the summary results for each model, including the model name, size of the checkpoint file, training time, maximum CKS (Cohen's Kappa Score), and maximum validation CKS. It starts as an empty DataFrame.
- The `for` loop iterates over each model name and corresponding history path. It reads the training history from the JSON file and stores it in the `histories` dictionary. Then, it extracts the relevant information from the history and appends it to the `sumup`

list.

- `sumup_result_df.loc[idx_model] = sumup`: This line adds a new row to the `sumup_result_df` DataFrame with the collected information for the current model.

Finally, `sumup_result_df.head()` is used to display the first few rows of the resulting DataFrame, providing a summary of the training results for each model.

In summary, this code collects and summarizes the training results of different models, including their file sizes, training times, and performance metrics, and stores the summary in a DataFrame for further analysis and comparison.

Remaining questions :

is 5856 images too short for deep learning?