Alexis Balayre

Artificial Intelligence Assignment

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Jun Li
18th March 2024

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of artificial intelligence in the medical field represents a major development in diagnostics, opening up new avenues for the accurate detection of disease through the analysis of medical images. This report explores the impact of AI on improving the interpretation of chest X-rays, a field characterised by its intrinsic complexity and the crucial need for diagnostic accuracy for effective patient management. This research project was stimulated by a challenge from Vingroup's Big Data Institute to develop automated systems capable of accurately identifying and classifying 14 types of thoracic abnormality from chest X-ray images.

Chest X-rays are essential in the diagnosis of various pathologies, including potentially fatal conditions such as COVID-19, tuberculosis, and pneumonia. However, their interpretation can be difficult, not least because of the subtlety of the pathological signs and the variability of interpretations among radiologists. Computer-aided detection and diagnosis (CADe/CADx) systems, enhanced by AI, offer a solution to these challenges, enabling rapid and accurate analysis of radiographic images, which could significantly improve clinical decisions and, consequently, patient outcomes.

The aim of this research was to design a deep learning algorithm exploiting a comprehensive dataset of 15,000 annotated chest scans provided by the Institute. This model aims to automatically detect abnormalities in chest X-rays, demonstrating the potential of AI not only as a viable diagnostic support tool but also as a means of improving diagnostic accuracy and reducing diagnosis times. By focusing on the detection and classification of thoracic abnormalities, this initiative seeks to address the pressing needs of healthcare professionals, particularly in regions where the lack of experienced radiologists can compromise the quality of patient care.

Chapter 2

literature Review

2.1 Deep Learning in Medical Imaging

Deep learning, a branch of artificial intelligence, is characterised by the use of deep neural networks to model complex representations and perform classification and prediction tasks on large quantities of data. In the context of medical imaging, this represents a rapidly expanding area of research that promises to revolutionise the way imaging data is analysed and interpreted.

2.1.1 Potential of deep learning in medical imaging

Deep learning has demonstrated its potential to improve diagnostic accuracy, automate repetitive tasks and identify subtle features in medical images. Algorithms have been developed for the early detection of diseases, such as cancer, by analysing mammography or magnetic resonance (MR) images.

One of the main strengths of deep learning is its ability to learn directly from data, without the need for explicit programming. This allows models to be adapted to a variety of medical imaging tasks, from segmentation to disease classification (3).

2.1.2 Challenges and outlook

Despite advances, the integration of deep learning into everyday clinical practice faces challenges, including the need for large amounts of annotated data, concerns about data privacy and security, and the need for rigorous validation.

Ongoing research aims to overcome these obstacles and explore new applications, such as improving image quality and predicting disease progression (4).

2.2 Object Detection Methods

Object detection is a fundamental task in computer vision that involves identifying and locating objects of different categories in an image or video. Unlike image classification, which assigns a label to the entire image, object detection aims to provide a label and bounding box for each object of interest in the image.

Object detection generally involves two main tasks: object classification (knowing what objects are) and object localisation (knowing where objects are). To be successful, an object detection system must be able to recognise objects under a variety of conditions, such as different sizes, viewing angles, and occlusion levels. There are two main types of method: two-stage methods and single-stage methods. These approaches differ mainly in the way they combine the proposal of regions of interest and object classification.

2.2.1 Two-Step Methods

2.2.1.1 Description

Two-step methods, such as R-CNN and its variants (Fast R-CNN and Faster R-CNN), start by generating proposals for regions of interest that could contain objects. They then use a convolution neural network (CNN) to classify the objects in each proposed region and refine their bounding boxes.

2.2.1.2 Benefits

- **High precision:**These methods allow detailed analysis of each region, leading to highly accurate object detection.
- **Flexibility:** The separation of tasks allows the integration of advanced CNNs for classification, taking advantage of advances in image classification.

2.2.1.3 Disadvantages

- **Processing speed:** Individual processing of each region can be slow, which is a disadvantage for real-time applications.
- **Computational complexity:** Generating and evaluating region proposals increases overall complexity.

2.2.2 One-Step Methods

2.2.2.1 Description

One-step methods, such as YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector), perform bounding box classification and prediction in a single pass through the network, greatly simplifying the process.

2.2.2.2 Benefits

- **Speed:** Designed to be fast, they facilitate the detection of objects in real time.
- **Simplicity:** Eliminating region proposals reduces complexity and resource requirements.

2.2.2.3 Disadvantages

- **Precision:** These methods may be less accurate for certain types of object, particularly small ones or those in groups.
- **Balance between speed and accuracy:** It is often necessary to fine-tune models to balance these aspects.

2.3 Using metrics to choose the right model

The performance of object detection models is primarily gauged using two critical metrics: Average Precision (AP) and Average Recall (AR). These metrics offer insights into the accuracy and reliability of the model in detecting and correctly labelling objects across different scenarios.

- **Average Precision (AP):** Measures the precision of the object detection model across various recall levels. Precision here refers to the proportion of true positive detections over the sum of true positive and false positive detections. AP is often averaged over multiple thresholds of Intersection over Union (IoU) to provide a comprehensive measure of model precision.
- **Average Recall (AR):** Assesses the model's ability to detect all relevant objects within an image. It is calculated as the proportion of true positive detections over the sum of true positives and false negatives. AR can be particularly informative when evaluating models on datasets with dense object placements.
- **Intersection over Union (IoU):** Fundamental metric used in object detection to evaluate the accuracy of the bounding boxes drawn by the model. IoU measures the overlap between the predicted bounding box and the ground truth bounding box, expressed as the ratio of their intersection over their union. A detection is classified as a true positive or false positive based on whether the IoU exceeds a specific threshold.

In order to choose the best model, the following table can be used.

Table 2.1: Recommended Metrics for Various Object Detection Use Cases (2)

Use Case	Real-world Scenarios	Recommended Metric
General object detection performance	Surveillance, sports analysis	AP
Low accuracy requirements	Augmented reality, gesture recognition	AP@.5
High accuracy requirements	Face detection	AP@.75
Detecting small objects	Small artifacts in medical imaging	AP-S
Medium-sized objects detection	Airport security luggage detection	AP-M
Large-sized objects detection	Detecting vehicles in parking lots	AP-L
Detecting 1 object per image	Single object tracking in videos	AR-1
Detecting up to 10 objects per image	Pedestrian detection in street cameras	AR-10
Detecting up to 100 objects per image	Crowd counting	AR-100
Recall for small objects	Medical imaging for tiny anomalies	AR-S
Recall for medium-sized objects	Sports analysis for players	AR-M
Recall for large objects	Wildlife tracking in wide landscapes	AR-L

Chapter 3

Methodology

3.1 Data Exploration

3.1.1 Overview of the Dataset

The dataset provided in the VinBigData Chest X-ray Abnormalities Detection competition consists of 15,000 postero-anterior chest X-ray scans, meticulously annotated for the presence of various thoracic abnormalities. Each image is labeled with one or more of 14 distinct abnormality classes, with a dedicated class for normal observations without findings. The images are stored in DICOM format, which not only captures the radiographic image but also houses rich metadata that could potentially enhance analysis and model training.

Class ID	Name
0	Aortic Enlargement
1	Atelectasis
2	Calcification
3	Cardiomegaly
4	Consolidation
5	ILD
6	Infiltration
7	Lung Opacity
8	Nodule/Mass
9	Other Lesion
10	Pleural Effusion
11	Pleural Thickening
12	Pneumothorax
13	Pulmonary Fibrosis
14	No Finding

Table 3.1: Classes and associated findings Abnormalities Detection dataset

3.1.2 Annotation Distribution Across Classes

Initial examination of the dataset revealed a notable class imbalance. The number of annotations per class was visualized in a Figure 4.2, showing that certain conditions such as Aortic Enlargement (Class 0) are more commonly annotated compared to others. The class 'No finding' (Class 14) had the most annotations, indicating a large proportion of normal cases.

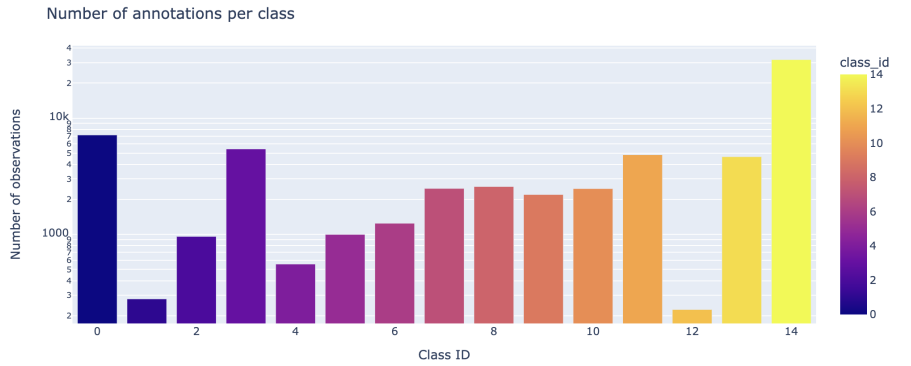

Figure 3.1: Number of annotations per class. The y-axis is on a logarithmic scale to account for the wide range of annotation counts.

3.1.3 Annotation Distribution Across Radiologists

The dataset annotations are also characterized by variability across radiologists. Figure 3.2 highlights the number of annotations contributed by each radiologist, with some radiologists annotating more extensively than others.

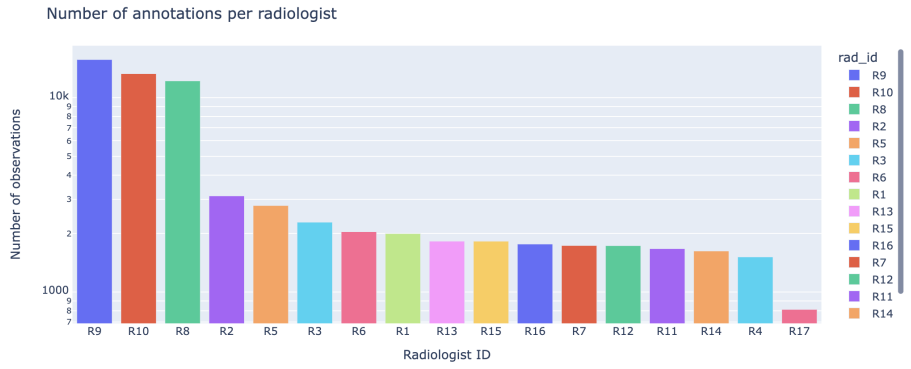

Figure 3.2: Number of annotations per radiologist. The distribution shows significant variability in the number of annotations made by different radiologists.

3.1.4 Inter-observer Variability

To further explore the inter-observer variability, a heatmap was constructed (Figure 3.3), showing the interplay between radiologist IDs and class annotations. This visualization underscored the 'No finding' class's dominance and revealed discrepancies in

the frequency of annotations per class by different radiologists, suggesting differences in diagnostic criteria or individual radiologist experience.

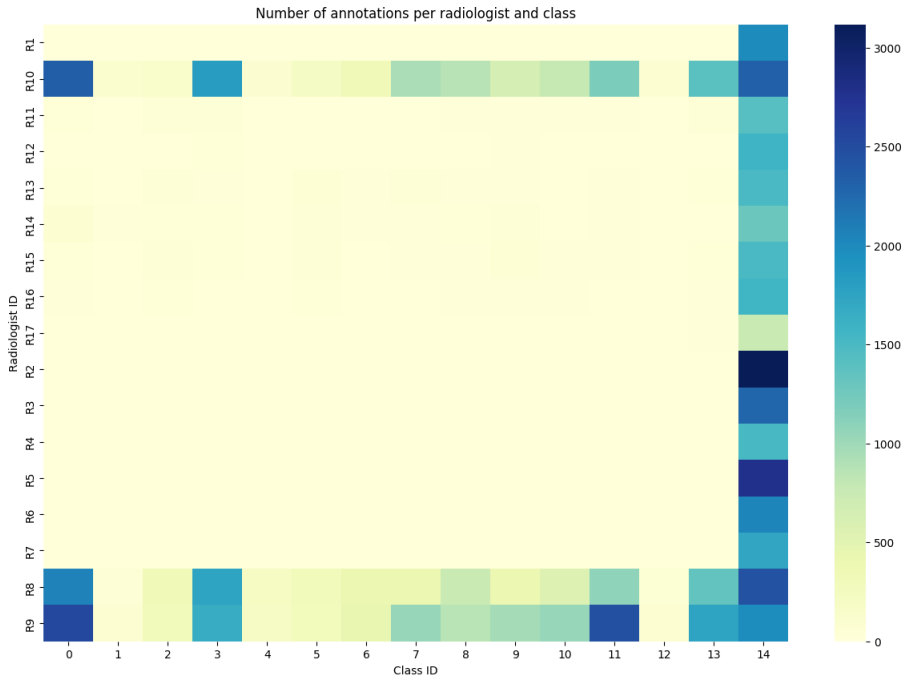

Figure 3.3: Number of annotations per radiologist and class. Darker colors indicate a higher number of annotations, showing a strong prevalence of 'No finding' annotations across all radiologists.

The initial data exploration indicates a rich and complex dataset that presents certain challenges for machine learning tasks, including class imbalance and significant inter-observer variability. These insights set the stage for careful preprocessing and the necessity of balancing techniques.

3.2 Data Extraction and Preprocessing

The workflow for preparing DICOM files for analysis encompasses three primary stages, each essential for transforming raw medical images into structured data amenable to analysis or machine learning applications. These stages are outlined as follows:

1. **DICOM File Metadata Extraction:** The process begins with the extraction of metadata from DICOM files, which are rich in information such as patient demographics, study specifics, and imaging parameters. This extraction is facilitated by the `pydicom` library, enabling the reading of each DICOM file and the retrieval of pertinent metadata fields. The collected metadata is subsequently stored in a CSV file, offering straightforward access and the ease of subsequent analysis.

2. **DICOM File Pixel Array Processing and Extraction:** Following metadata extraction, attention turns to the DICOM files' pixel data. This phase includes applying the Value of Interest (VOI) Look-Up Table (LUT) for image normalization, adjusting images based on their photometric interpretation (e.g., inverting "MONOCHROME1" images), and scaling the pixel values to an 8-bit format. The processed pixel data is stored in an HDF5 file, chosen for its efficiency in managing sizable datasets and facilitating fast access to individual images.

3. **DICOM File Features Extraction:** The concluding stage involves extracting salient features from the processed images. This encompasses computing features related to texture, shape, and intensity histograms to quantitatively describe each image's essential characteristics. These features are vital for training machine learning models, as they provide a numeric representation of the images. The extracted features are assembled into a structured dataset, usually saved in a CSV file, ready for in-depth analysis and model training.

In addition, the training dataset has been split into 3 datasets (training, validation and test). This division was made according to the 'class_id' label value to avoid any training problems.

3.3 Model Architecture (Faster R-CNN)

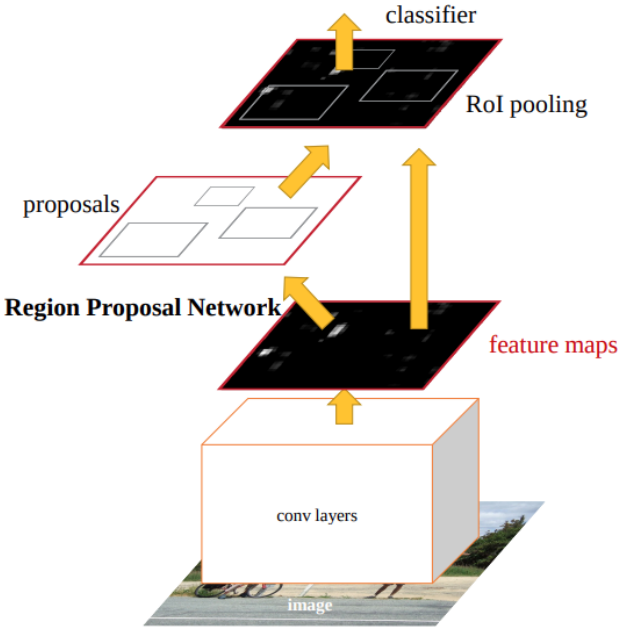

Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the 'attention' of this unified network.

Figure 3.4: The RPN module works as 'attention' in Faster RCNN (1)

The Faster R-CNN model is a deep learning architecture for object detection, consisting of two main components: a Region Proposal Network (RPN) and a Fast R-CNN object detection network. The workflow of this model can be described as follows (5):

3.3.1 Region Proposal Network (RPN)

The RPN is a fully convolutional network that operates on the feature maps $\mathbf{X}$ generated by the backbone network (here ResNet-50). It generates object proposals by classifying anchor boxes $\mathbf{a}$ as either containing an object or not, and also refines the bounding box coordinates for positive proposals.

The RPN outputs a set of object proposals $\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n\}$, where each proposal $\mathbf{r}_i = (p_i, b_i)$ consists of a probability score p_i indicating the likelihood of containing an object, and a bounding box b_i represented by its coordinates.

The RPN is trained using a multi-task loss function:

$$L_{\text{RPN}} = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_i, p_i^*) + \lambda \frac{1}{N_{\text{reg}}} \sum_i p_i^* L_{\text{reg}}(b_i, b_i^*) \tag{3.1}$$

where L_{cls} is the classification loss (e.g., cross-entropy loss), L_{reg} is the bounding box regression loss (e.g., smooth L1 loss), p_i^* and b_i^* are the ground truth labels for proposal i, N_{cls} and N_{reg} are normalization factors, and λ is a balancing weight.

3.3.2 Fast R-CNN Network

The Fast R-CNN network is responsible for classifying the proposed RoIs and refining their bounding box coordinates. It takes the feature maps $\mathbf{X}$ from the backbone network and the proposed RoIs $\mathbf{R}$ from the RPN as input.

The RoI pooling layer extracts a fixed-size feature map $\mathbf{x}_i$ from each RoI $\mathbf{r}_i$, which is then fed into fully connected layers for classification and bounding box regression:

$$p_{\text{cls}}(c|\mathbf{x}_i) = \text{Softmax}(W_{\text{cls}}^T \mathbf{x}_i + b_{\text{cls}}) \tag{3.2}$$

$$b_{\text{reg}}(\mathbf{x}_i) = W_{\text{reg}}^T \mathbf{x}_i + b_{\text{reg}} \tag{3.3}$$

where $p_{\text{cls}}(c|\mathbf{x}_i)$ is the predicted probability of RoI $\mathbf{x}_i$ belonging to class c, and $b_{\text{reg}}(\mathbf{x}_i)$ is the predicted bounding box regression offsets for $\mathbf{x}_i$. W_{cls}, W_{reg}, b_{cls}, and b_{reg} are learnable parameters.

The Fast R-CNN network is trained using a multi-task loss function similar to the RPN:

$$L_{\text{Fast R-CNN}} = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_{\text{cls}}(c|\mathbf{x}_i), c_i^*) + \lambda \frac{1}{N_{\text{reg}}} \sum_i c_i^* L_{\text{reg}}(b_{\text{reg}}(\mathbf{x}_i), b_i^*) \tag{3.4}$$

where c_i^* and b_i^* are the ground truth labels for RoI $\mathbf{x}_i$, and λ is a balancing weight.

3.3.3 Model Training and Evaluation

The overall loss function for the Faster R-CNN model is the sum of the RPN loss and the Fast R-CNN loss:

$$L = L_{\text{RPN}} + L_{\text{Fast R-CNN}} \tag{3.5}$$

The model is trained by minimizing this loss function using an optimization algorithm, such as Stochastic Gradient Descent (SGD) with momentum and weight decay. Here we use the transfer learning technique, using a model pre-trained on a large dataset. The features learned can be transferred to a new object detection task with less data. This approach allows the model to benefit from general visual knowledge, making it easier to learn specific object detection tasks with better generalisation and less training data. Only the last part of the pipeline is modified (classifier).

3.4 Training Pipeline

3.4.1 Framework (PyTorch Lightning)

Training an object detection model such as Faster R-CNN requires a structured approach to efficiently manage data loading, training and inference. In this project, the PyTorch Lightning framework was used to perform the transfer learning operation on the Faster-R-CNN model.

PyTorch Lightning is a framework that builds on PyTorch to provide a more organized code structure and reduce the amount of boilerplate code required. It provides a clear separation between model training logic and automation code, facilitating code development, debugging and reuse. PyTorch Lightning supports advanced features such as multi-GPU, TPU, distributed training and mixed precision, making the training process more efficient and scalable.

3.4.2 Data Loading

Data loading is an essential step in the training pipeline, requiring special attention to object detection due to the complexity of annotations and data formats. PyTorch Lightning uses PyTorch's DataLoader to automate data loading, blending and batch distribution, while allowing customization for specific cases such as object detection datasets. Data preparation often involves transformations such as resizing, normalizing and augmenting data to improve model generalization.

3.4.3 Training and Inference

Training an object detection model with PyTorch Lightning begins with the definition of the LightningModule, which encapsulates the model, data and optimization logic. During training, the model learns to predict bounding boxes and object classes in training images, using metrics such as loss to guide the optimization process. PyTorch Lightning facilitates the implementation of customized training routines and provides integrated tools for tracking performance, saving checkpoints, and resuming training.

Inference, or prediction from unseen data, involves using the trained model to detect objects in new images. The framework provides a simplified interface for model evaluation, facilitating the generation and evaluation of predictions on the validation or test set.

3.5 Model Optimization and Improvements

Model optimization and enhancements are crucial for maximizing the performance of an object detection model like Faster R-CNN. This section explores the key techniques involved in this process, including optimizer and learning rate planner selection and data augmentation.

3.5.1 Optimizer and Learning Rate Scheduler

The choice of optimizer and learning rate planner plays a significant role in the effectiveness of model training.Optimizers such as SGD (Stochastic Gradient Descent) with momentum or Adam are commonly used due to their ability to efficiently navigate the complex loss landscapes often encountered in object detection model training.

- **Optimizer:** The model implemented in this project uses Stochastic Gradient Descent (SGD) optimizer, which is renowned for producing better generalized models.
- **Learning rate planners:** The scheduler used was CosineAnnealingLR, which adjusts the learning rate according to a cosine function, can improve model performance by dynamically adjusting the learning rate during training. This approach helps to avoid local minima and refine convergence towards the global minimum (6).

3.5.2 Data Augmentation

Data augmentation is an essential technique for improving model robustness and generalization by artificially increasing the diversity of training data. In the context of object detection, common transformations include:

- **Rotate, Resize and Crop:** Modifies the perspective and size of objects in images, helping the model learn to recognize objects at different angles and scales.
- **Horizontal/Vertical Inversion:** Helps generalize detection regardless of object orientation.
- **Color Change:** Adjusts brightness, contrast and saturation to enhance the model's ability to recognize objects under different lighting and color conditions.

3.6 Post-Validation

In order to avoid false positives, a binary classification Tree decision model was trained to predict whether a DICOM file has an anomaly or not.

3.7 Evaluation Metrics

Model performance is assessed using mean Average Precision (mAP) and mean Average Recall (mAR) across different Intersection over Union (IoU) thresholds, providing a comprehensive evaluation of detection accuracy.

3.7.1 Intersection over Union (IoU)

Intersection over Union (IoU) is a key metric for assessing the accuracy of of the bounding boxes predicted by the model. It is defined as the ratio between the area of the intersection and the area of the union of the predicted and ground truth:

$$IoU = \frac{\text{Intersection area}}{\text{Union area}} \tag{3.6}$$

A prediction is considered correct if the IoU with a ground truth box field exceeds a certain threshold. In the context of the competition, this threshold is **0.4**.

3.7.2 Precision and Recall

Precision is the proportion of positive identifications that are correct, while recall is the proportion of actual ground truths that are truths that are correctly identified. They are calculated as follows:

$$\text{Precision} = \frac{TP}{TP+FP} \tag{3.7}$$

$$\text{Recall} = \frac{TP}{TP+FN} \tag{3.8}$$

where TP represents true positives, FP false positives, and FN false negatives.

3.7.3 Mean Average Precision (mAP)

The mean Average Precision (mAP) is the average of the APs calculated for each class of objects over different IoU thresholds. The AP for a class is the area under the precision-recall curve, and the mAP is an average of these values for all classes:

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i \tag{3.9}$$

where N is the number of classes and AP_i is the Average Precision for class i.

Chapter 4

Results and Discussion

4.1 Results

The following results were obtained using the computing power of Cranfield university's HPC cluster. Two 32GB GPUs were used to train the models.

4.1.1 Training Loss Evolution

Several tests were ran using different setup (training images sizes, learning rate, optimiser, scheduler). As you can see in figure 4.1, noise decreases during the training with the number of epochs and tests performed.

The best hyperparameters found were:

- **Image Size:** 800 * 1000
- **Learnint Rate:** 3e-3

At the start of the tests, the model converged after 6-7 hours of training, but as the training progressed, convergence began earlier (around 4 hours of training).

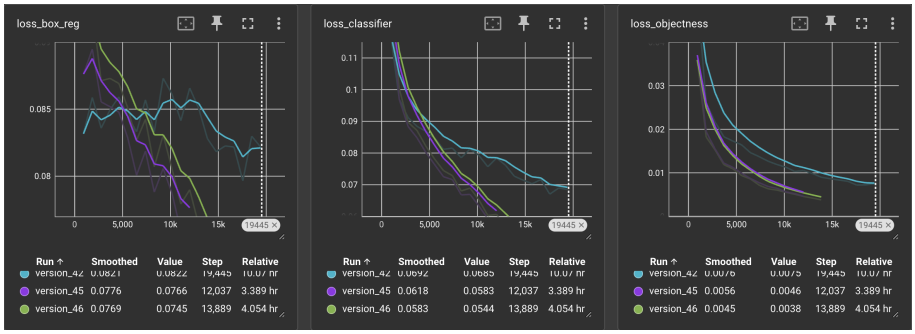

Figure 4.1: Training Loss Evolution

4.1.2 Validation Accuracy Evolution

Here is the evolution of accuracy on validation set.

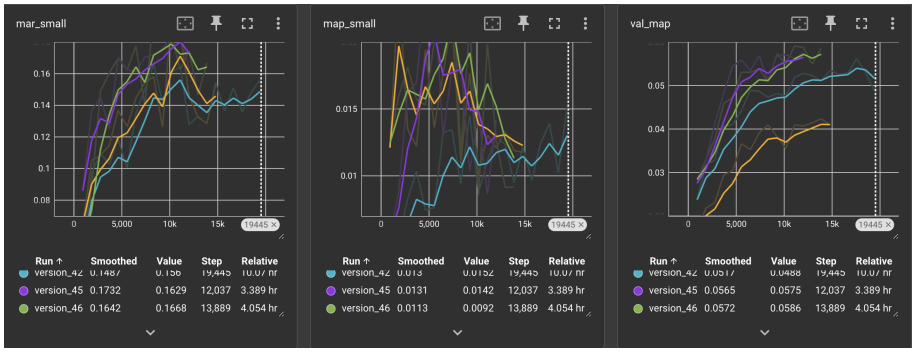

Figure 4.2: Validation Accuracy Evolution

4.1.3 Evaluation Metrics per Class

The total MAp value obtained was 0.1868. Therefore the mean accuracy of the model is 18,68%, which is correct due to the provided dataset. As expected, accuracy is very high for classes with high representation, such as class 14.

Here is a table summarising the accuracy of the results by class:

Class	TP	FP	FN	Precision	Recall	F1 Score	mAP	mAR@100
0	1031	1810	219	0.363	0.825	0.504	0.3933	0.7924
1	24	156	25	0.133	0.490	0.210	0.0555	0.3333
2	64	806	94	0.074	0.405	0.125	0.0252	0.3472
3	694	1802	159	0.278	0.814	0.414	0.4062	0.8071
4	45	485	48	0.085	0.484	0.144	0.0424	0.3614
5	139	1080	66	0.114	0.678	0.195	0.0906	0.5267
6	178	1512	70	0.105	0.718	0.184	0.1054	0.6096
7	454	4786	143	0.087	0.760	0.156	0.0661	0.5791
8	268	2544	166	0.095	0.618	0.165	0.1339	0.5633
9	215	2800	205	0.071	0.512	0.125	0.0539	0.3444
10	498	2431	101	0.170	0.831	0.282	0.2124	0.6935
11	912	10079	269	0.083	0.772	0.150	0.0646	0.5868
12	55	885	9	0.059	0.859	0.110	0.3272	0.7353
13	792	4841	312	0.141	0.717	0.235	0.0971	0.5115
14	3799	345	974	0.917	0.796	0.852	0.7276	0.7959

Table 4.1: Evaluation Metrics per Class

4.1.4 Predictions Visualisation

Here are the predicted labels without score threshold filter. As visible, there are a lot of false positive, however the results are consistent with the referent.

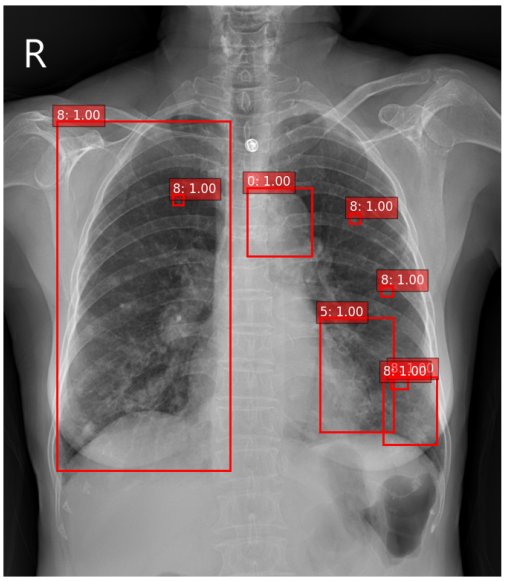

Figure 4.3: True Labels

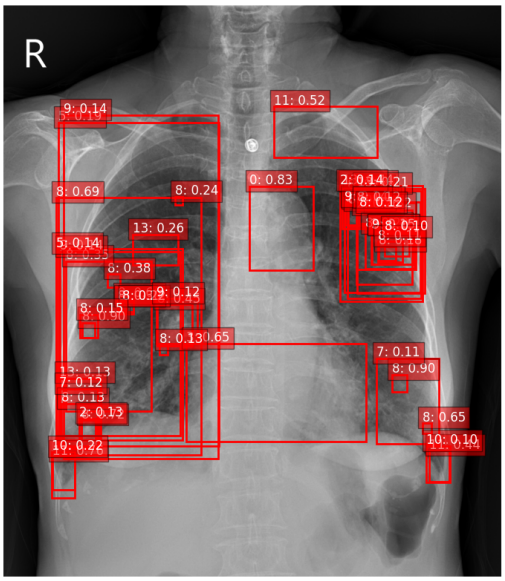

Figure 4.4: Predicted labels

Here are predicted labels with the score threshold filter. The results are more satisfying with filter!

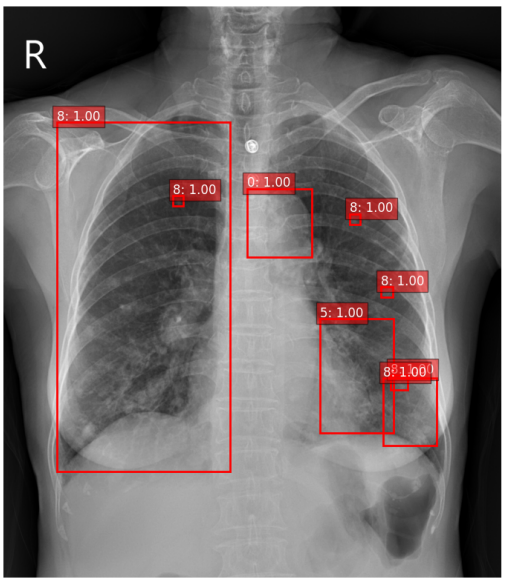

Figure 4.5: True Labels

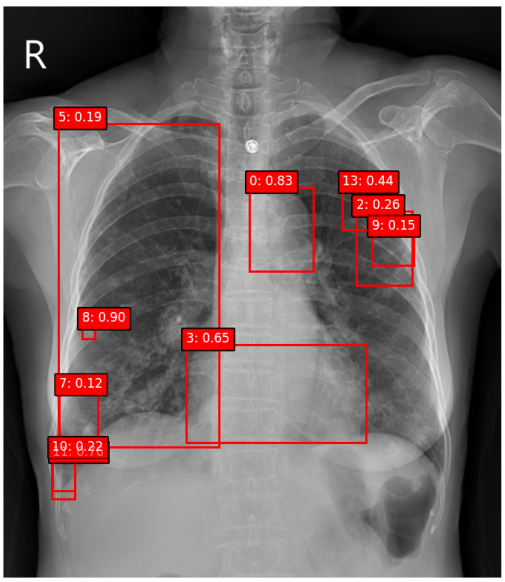

Figure 4.6: Filtered Predicted labels

4.2 AI Ethical challenges in the medical sector

While the use of artificial intelligence in radiology holds enormous transformative potential for the medical sector, this application nevertheless raises a number of significant ethical issues and challenges, requiring careful thought and innovative solutions to ensure that the development and use of AI in medicine is done in an ethical and responsible manner.

4.2.1 Respect for Confidentiality and Data Security

The first ethical challenge concerns the confidentiality and security of patient data. AI systems, such as those developed to detect thoracic anomalies from X-rays, require access to large volumes of sensitive medical data. It is imperative to ensure that this data is protected from unauthorised access or misuse, in compliance with regulations such as the RGPD in Europe or HIPAA in the United States. The implementation of advanced cryptographic techniques and access management systems is essential to secure this information (7).

4.2.2 Algorithmic bias and fairness

Another major issue is the risk of algorithmic bias, which can lead to unfair diagnoses. The datasets used to train AI systems may reflect existing biases, resulting in variable performance across demographic groups. This raises the issue of fairness in automated diagnoses, where certain populations could be disadvantaged. To counter this problem, it is crucial to ensure that datasets are diverse and representative, and to develop methodologies for identifying and correcting algorithmic biases (8).

4.3 Transparency and Explicability

The transparency and explicability of decisions made by AI systems is a central ethical challenge. The decision-making mechanisms of complex AI models, such as deep neural networks, are often perceived as a ”black box”, making it difficult to understand and justify the diagnoses proposed. It is therefore imperative to work towards more explainable AI models, enabling healthcare professionals to understand and validate the recommendations provided by these systems before making clinical decisions (9).

4.4 Responsibility and Professional Autonomy

The question of liability in the event of diagnostic errors involving AI is also a cause for concern. Determining the share of responsibility between the creators of AI systems, the healthcare professionals using them, and healthcare institutions requires in-depth legal and ethical reflection. Furthermore, the use of AI should not erode the professional autonomy of radiologists, but rather serve as a complementary tool enabling them to improve their accuracy and efficiency (10).

Chapter 5

Conclusion

To conclude, this study demonstrated the transformative potential of artificial intelligence in the analysis of chest X-rays, offering a promising avenue for improving diagnostic accuracy and reducing interpretation times. Despite an average accuracy of 18.68%, reflecting the challenges inherent in applying AI in this complex field, the results encourage future research to refine and extend these approaches. The moderate performance of the model can largely be attributed to the heterogeneity and imbalance of the dataset used, highlighting the crucial importance of well-balanced and homogeneous datasets in the development of efficient and reliable AI systems in medicine. In addition, the ethical implications highlighted are a reminder of the importance of developing these technologies in a responsible manner, ensuring the security of data, the accuracy of algorithms, and the transparency of decision-making processes. By continuing the collaboration between AI developers, healthcare professionals, and regulators, we can harness the full potential of AI to transform radiology and beyond, while carefully navigating the ethical and practical complexities of this new medical frontier.

References

1. Ren S, He K, Girshick R, Sun J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks; 2016.

2. Padilla R, Roberts A. Object Detection Leaderboard: Decoding Metrics and Their Potential Pitfalls; 2023. Available at: https://huggingface.co/blog/object-detection-leaderboard. Accessed: 2024-03-01.

3. Litjens G, Kooi T, Bejnordi BE, Setio AAA, Ciompi F, Ghafoorian M, et al. A survey on deep learning in medical image analysis. Medical Image Analysis. 2017;42:60-88. Available at: https://www.sciencedirect.com/science/article/pii/S1361841517301135.

4. Shen D, Wu G, Suk HI. Deep Learning in Medical Image Analysis. Annual Review of Biomedical Engineering. 2017 Jun;19:221-48. Epub 2017 Mar 9.

5. Rath SR. Object Detection using PyTorch Faster RCNN ResNet50 FPN V2; 2022. Available at: https://debuggercafe.com/object-detection-using-pytorch-faster-rcnn-resnet50-fpn-v2/. Accessed: 2024-03-01.

6. Smith LN. Cyclical Learning Rates for Training Neural Networks; 2017.

7. Murdoch B. Privacy and artificial intelligence: challenges for protecting health information in a new era. BMC Medical Ethics. 2021;22(1):122. Available at: https://doi.org/10.1186/s12910-021-00687-3.

8. Mittermaier M, Raza MM, Kvedar JC. Bias in AI-based models for medical applications: challenges and mitigation strategies. npj Digital Medicine. 2023;6(1):113. Available at: https://doi.org/10.1038/s41746-023-00858-z.

9. Amann J, Blasimme A, Vayena E, Frey D, Madai VI, the Precise4Q consortium. Explainability for artificial intelligence in healthcare: a multidisciplinary perspective. BMC Medical Informatics and Decision Making. 2020;20(1):310. Available at: https://doi.org/10.1186/s12911-020-01332-6.

10. Naik N, Hameed BMZ, Shetty DK, Swain D, Shah M, Paul R, et al. Legal and Ethical Consideration in Artificial Intelligence in Healthcare: Who Takes Responsibility? Frontiers in surgery. 2022;9:862322.

Appendix A

Source Codes

Appendix A.A Dataset Class

```
import h5py
import pandas as pd
import torch
from torch.utils.data import Dataset
import albumentations as A
from albumentations.pytorch import ToTensorV2
from PIL import Image
import numpy as np
import cv2


class ChestXrayDataset(Dataset):
    """
    A dataset class for chest X-ray images, designed for use with PyTorch data loaders.

    This class handles loading and preprocessing of chest X-ray images stored in an HDF5 file,
    along with their associated annotations provided in a CSV file. It supports customizable
    image resizing and transformations for data augmentation.

    Parameters:
    - labels_df_path (str): Path to the CSV file containing image annotations.
    - hdf5_path (str): Path to the HDF5 file containing image data.
    - target_size (tuple): Desired output size of the images as (width, height).
    - stage (str): The stage of model training ('fit', 'validate', 'test', 'predict'),
                   which determines the set of transformations to apply.

    Attributes:
    - stage (str): Current stage of model training.
    - labels_df (DataFrame): DataFrame containing image annotations.
    - image_annotations (dict): Aggregated annotations for each image.
    - hdf5_path (str): Path to the HDF5 file.
    - target_size (tuple): Target size for image resizing.
    - transform (A.Compose): Composed Albumentations transformations to apply.
    """

    def __init__(self, labels_df_path, hdf5_path, target_size=(224, 224), stage="fit"):
        super(ChestXrayDataset, self).__init__()

        self.stage = stage

        self.labels_df = pd.read_csv(labels_df_path)

```

```
        self.image_annotations = self.aggregate_annotations()
        self.hdf5_path = hdf5_path
        self.target_size = target_size

        self.transform = (
            A.Compose(
                [
                    A.Resize(
                        width=self.target_size[0], height=self.target_size[1], p
                            =1.0
                    ),
                    A.HorizontalFlip(p=0.5),
                    A.Normalize(
                        mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225],
                        max_pixel_value=255.0,
                        p=1.0,
                    ),
                    ToTensorV2(p=1.0),
                ],
                bbox_params=A.BboxParams(
                    format="pascal_voc",
                    label_fields=["labels"],
                ),
            )
            if stage == "fit"
            else A.Compose(
                [
                    A.Resize(
                        width=self.target_size[0], height=self.target_size[1], p
                            =1.0
                    ),
                    A.Normalize(
                        mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225],
                        max_pixel_value=255.0,
                        p=1.0,
                    ),
                    ToTensorV2(p=1.0),
                ],
                bbox_params=A.BboxParams(
                    format="pascal_voc",
                    label_fields=["labels"],
                ),
            )
        )

    def aggregate_annotations(self):
        """
        Aggregates bounding box and label annotations for each image.

        Parses the annotations DataFrame to compile a dictionary that maps each
            unique
        image ID to its bounding boxes and labels.

        Returns:
        - dict: A dictionary with image IDs as keys, and their "boxes" and "labels"
                aggregated from the annotations DataFrame.
        """
        agg_annotations = {}
        for _, row in self.labels_df.iterrows():
            image_id = row["image_id"]
            if image_id not in agg_annotations:
                agg_annotations[image_id] = {"boxes": [], "labels": []}
            if pd.notna(row["x_min"]):  # If bounding box exists
                agg_annotations[image_id]["boxes"].append(
                    [row["x_min"], row["y_min"], row["x_max"], row["y_max"]]
                )
                agg_annotations[image_id]["labels"].append(row["class_id"] + 1)
        return agg_annotations
```

```
110
111     def __getitem__(self, idx):
112         """
113         Retrieves an item from the dataset at the specified index.
114
115         Parameters:
116         - idx (int): Index of the item to retrieve.
117
118         Returns:
119         - tuple or tuple of (str, torch.Tensor, dict): Depending on the stage,
120           if 'fit', returns (image, target) where 'image' is the transformed image
                tensor
121           and 'target' is a dictionary with boxes, labels, area, and iscrowd.
122           Otherwise, returns (image_id, image, target) including the image ID.
123         """
124
125         image_id = list(self.image_annotations.keys())[idx]
126         annotations = self.image_annotations[image_id]
127
128         with h5py.File(self.hdf5_path, "r") as hdf5_file:
129             image_data = hdf5_file[image_id + ".dicom"][()]
130         image_data = np.repeat(image_data[:, :, np.newaxis], 3, axis=-1)
131
132         transformed = self.transform(
133             image=image_data, bboxes=annotations["boxes"], labels=annotations["
                labels"]
134         )
135         image = transformed["image"]
136         boxes = transformed["bboxes"]
137         labels = transformed["labels"]
138
139         target = {}
140         if boxes:
141             target["boxes"] = torch.as_tensor(boxes, dtype=torch.float32)
142             target["labels"] = torch.as_tensor(labels, dtype=torch.int64)
143             target["area"] = (target["boxes"][:, 3] - target["boxes"][:, 1]) * (
144                 target["boxes"][:, 2] - target["boxes"][:, 0]
145             )
146             target["iscrowd"] = torch.zeros((len(boxes),), dtype=torch.int64)
147         else:
148             target["boxes"] = torch.zeros((0, 4), dtype=torch.float32)
149             target["labels"] = torch.zeros((0,), dtype=torch.int64)
150             target["area"] = torch.zeros((0,), dtype=torch.float32)
151             target["iscrowd"] = torch.zeros((0,), dtype=torch.int64)
152
153         if self.stage == "fit":
154             return image, target
155         return image_id, image, target
156
157     def __len__(self):
158         """
159         Returns the total number of items in the dataset.
160
161         Returns:
162         - int: The total number of images in the dataset.
163         """
164         return len(self.image_annotations)
```

Appendix A.B Data Module Class

```python
import lightning as L
from torch.utils.data import DataLoader

from ChestXrayDataset import ChestXrayDataset


def collate_fn(batch):
    """
    Custom collate function for DataLoader.

    This function prepares a batch by collating the list of samples into a batch,
    where each sample is a tuple of its attributes. It is used to handle cases where
    the dataset returns a tuple of data points. The function rearranges the batch to
    align each element of the tuple across the data points in the batch.

    Parameters:
    - batch (list): A list of tuples, where each tuple corresponds to a data sample.

    Returns:
    - tuple: A tuple of lists, where each list contains all elements of the batch for
      that position in the original tuple.
    """
    return tuple(zip(*batch))


class ChestXrayDataModule(L.LightningDataModule):
    """
    Data module for chest X-ray images, utilizing PyTorch Lightning for structured data loading.

    This module is designed to handle the loading and preprocessing of chest X-ray image datasets,
    facilitating easy integration into a deep learning pipeline. It is specifically configured
    to work with HDF5 datasets and is customizable in terms of image size, batch size, and the
    number of worker threads for data loading.

    Attributes:
    - hdf5_path (str): Path to the HDF5 file containing the datasets.
    - train_dataset_path (str): Path to the training dataset.
    - val_dataset_path (str): Path to the validation dataset.
    - test_dataset_path (str): Path to the test dataset.
    - target_size (tuple): The dimensions to which the images will be resized.
    - batch_size (int): The size of each data batch.
    - num_workers (int): The number of worker threads for data loading operations.
    """

    def __init__(
        self,
        hdf5_path,
        train_dataset_path,
        val_dataset_path,
        test_dataset_path,
        target_size=(224, 224),
        batch_size=8,
        num_workers=8,
    ):
        super().__init__()

        self.hdf5_path = hdf5_path
        self.train_dataset_path = train_dataset_path
        self.val_dataset_path = val_dataset_path
```

```
60          self.test_dataset_path = test_dataset_path
61          self.target_size = target_size
62          self.batch_size = batch_size
63          self.num_workers = num_workers
64
65      def setup(self, stage=None):
66          """
67          Prepares the datasets for the training, validation, testing, and prediction
                stages.
68
69          Depending on the stage, this method initializes the corresponding dataset(s
                )
70          using the provided dataset paths and the HDF5 file. It ensures that
                datasets
71          are ready for use when their respective DataLoader is called.
72
73          Parameters:
74          - stage (str, optional): The stage for which to setup datasets. Can be 'fit
                ',
75            'test', 'predict', or None. If None, datasets for all stages are prepared
                  .
76          """
77          if stage == "fit" or stage is None:
78              self.train_dataset = ChestXrayDataset(
79                  self.train_dataset_path, self.hdf5_path, self.target_size, stage=
                        stage
80              )
81              self.val_dataset = ChestXrayDataset(
82                  self.val_dataset_path, self.hdf5_path, self.target_size, stage=
                        stage
83              )
84          if stage == "test" or stage is None:
85              self.test_dataset = ChestXrayDataset(
86                  self.test_dataset_path, self.hdf5_path, self.target_size, stage=
                        stage
87              )
88
89      def train_dataloader(self):
90          """
91          Creates a DataLoader for the training dataset.
92
93          Returns:
94          - DataLoader: The DataLoader for the training dataset, configured with
95            shuffle, batch size, and the custom collate function.
96          """
97          return DataLoader(
98              self.train_dataset,
99              batch_size=self.batch_size,
100             num_workers=self.num_workers,
101             shuffle=True,
102             drop_last=True,
103             collate_fn=collate_fn,
104         )
105
106     def val_dataloader(self):
107         """
108         Creates a DataLoader for the validation dataset.
109
110         Returns:
111         - DataLoader: The DataLoader for the validation dataset, configured with
112           batch size and the custom collate function.
113         """
114         return DataLoader(
115             self.val_dataset,
116             batch_size=self.batch_size,
117             shuffle=False,
118             num_workers=self.num_workers,
119             collate_fn=collate_fn,
120         )
121
```

```
122     def test_dataloader(self):
123         """
124         Creates a DataLoader for the test dataset.
125 
126         Returns:
127         - DataLoader: The DataLoader for the test dataset, configured with
128           batch size and the custom collate function.
129         """
130         return DataLoader(
131             self.test_dataset,
132             batch_size=self.batch_size,
133             shuffle=False,
134             num_workers=self.num_workers,
135             collate_fn=collate_fn,
136         )
```

Appendix A.C Faster-R-CNN Lightning Model Class

```python
import lightning as L
from torchvision.models.detection import (
    fasterrcnn_resnet50_fpn_v2,
    FasterRCNN_ResNet50_FPN_V2_Weights,
)
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchmetrics.detection.mean_ap import MeanAveragePrecision
import torch
import torch.nn.functional as F
import torchmetrics


class ChestXrayLightningModel(L.LightningModule):
    """
    A LightningModule for chest X-ray detection using a Faster R-CNN model with a
        ResNet50 backbone.

    This module is designed to be used for the detection of abnormalities in chest
        X-ray images,
    utilizing a pre-trained Faster R-CNN model with custom modifications for the
        task-specific
    number of classes.

    Parameters:
    - num_classes (int): Number of classes for detection, including the background
        class.
    - learning_rate (float, optional): Initial learning rate for the optimizer.
    - cosine_t_max (int, optional): Maximum number of iterations for the cosine
        annealing scheduler.

    Attributes:
    - model (torch.nn.Module): The Faster R-CNN model with a ResNet50 backbone.
    - learning_rate (float): Learning rate for the optimizer.
    - cosine_t_max (int): Maximum number of iterations for the cosine annealing
        scheduler.
    - val_metric (MeanAveragePrecision): Metric for validation, calculating mean
        average precision.
    """

    def __init__(self, num_classes, learning_rate=0.01, cosine_t_max=20):
        super().__init__()

        self.model = fasterrcnn_resnet50_fpn_v2(
            weights=FasterRCNN_ResNet50_FPN_V2_Weights.DEFAULT
        )
        in_features = self.model.roi_heads.box_predictor.cls_score.in_features
        self.model.roi_heads.box_predictor = FastRCNNPredictor(in_features,
            num_classes)

        self.learning_rate = learning_rate
        self.cosine_t_max = cosine_t_max

        self.val_metric = MeanAveragePrecision(iou_type="bbox", class_metrics=True)

        self.save_hyperparameters(ignore=["model"])

    def forward(self, inputs):
        """
        Forward pass of the model.

        Parameters:
        - inputs (list of torch.Tensor): List of images to perform detection on.

        Returns:
        - dict: The model's predictions including detected boxes, labels, and
            scores.
        """
```

```
59          return self.model(inputs)
60
61      def training_step(self, batch, batch_idx):
62          """
63          Defines the training logic for a single batch of data.
64
65          Parameters:
66          - batch (tuple): The batch to train on, containing images and their
                respective targets.
67          - batch_idx (int): The index of the current batch.
68
69          Returns:
70          - torch.Tensor: The aggregated loss from the Faster R-CNN model.
71          """
72          images, targets = batch
73          targets = [{k: v for k, v in t.items()} for t in targets]
74          loss_dict = self.model(images, targets)
75          self.log_dict(
76              loss_dict, on_step=False, on_epoch=True, prog_bar=True, logger=True
77          )
78          train_loss = sum(loss for loss in loss_dict.values())
79          self.log(
80              "train_loss",
81              train_loss,
82              on_step=False,
83              on_epoch=True,
84              prog_bar=True,
85              logger=True,
86          )
87          return train_loss
88
89      def validation_step(self, batch, batch_idx):
90          """
91          Defines the validation logic for a single batch of data.
92
93          Parameters:
94          - batch (tuple): The batch to validate on, containing images and their
                respective targets.
95          - batch_idx (int): The index of the current batch.
96          """
97          images, targets = batch
98          pred = self.model(images)
99          self.val_metric.update(preds=pred, target=targets)
100
101     def on_validation_epoch_end(self):
102         """
103         Called at the end of the validation epoch to log the mean average precision
                (mAP) and
104         mean average recall (mAR) metrics.
105         """
106         mAPs = self.val_metric.compute()
107         map_per_class = mAPs.pop("map_per_class")
108         mar_100_per_class = mAPs.pop("mar_100_per_class")
109         classes = mAPs.pop("classes")
110         map = mAPs.pop("map")
111         self.log(
112             "val_map", map, on_step=False, on_epoch=True, prog_bar=True, logger=
                    True
113         )
114         self.log_dict(mAPs, on_step=False, on_epoch=True, prog_bar=True, logger=
                True)
115         try:
116             for i, class_name in enumerate(classes):
117                 self.log(
118                     f"mAP_{class_name}",
119                     map_per_class[i],
120                     on_step=False,
121                     on_epoch=True,
122                     prog_bar=True,
123                     logger=True,
```

```
                )
                self.log(
                    f"mar_100_{class_name}",
                    mar_100_per_class[i],
                    on_step=False,
                    on_epoch=True,
                    prog_bar=True,
                    logger=True,
                )
        except:
            pass
        self.val_metric.reset()

    def configure_optimizers(self):
        """
        Sets up the optimizer and learning rate scheduler to be used during
            training.

        Returns:
        - dict: A dictionary containing the optimizer and LR scheduler
            configurations.
        """
        optimizer = torch.optim.SGD(
            self.model.parameters(),
            lr=self.learning_rate,
            momentum=0.9,
            weight_decay=0.0005,
        )  # SGD optimizer with momentum and weight decay
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
            optimizer, T_max=self.cosine_t_max, eta_min=0.0001
        )  # Cosine annealing learning rate scheduler

        return {
            "optimizer": optimizer,
            "lr_scheduler": scheduler,
        }
```