Convolutional Neural Networks in Pulmonary Embolism detection on Computer Tomography Pulmonary Angiogram scans

George-Ionut Buleandra

(P2625446)

Module: CTEC3451_2223_520 Development Project
Supervisor: Dr Hossein Malekmohamadi
Module Leader: Dr Hossein Malekmohamadi

Abstract

The paper at hand explores the training of VGG19, Xception and InceptionResNetV2 on a balanced sample extracted from RSNA Pulmonary Embolism Dataset for PE detection in CTPA scans and creation of a GUI to facilitate the use of the model champion model chosen between the three models above. The training was done using Keras and Kaggle notebooks to speed up the process using the provided accelerators. Also, the benefits of using Kanban and SCRUM in development management were presented. The study concludes by critiquing the results obtained and recommending improvements in the training process as well as the ideal integration of the model into the workflow.

Acknowledgements

I would like to thank my supervisor Dr Hossein Malekmohamadi for sharing his knowledge in computer vision and for guidance and patience through the project. Also, for the idea to meet with Romans Kononovs who chose the same project as me and to debate around CNN in PE detection and the difference between the datasets used that each one used in their projects. And nonetheless for the idea to continue the research of CNN in PE detection if we obtain models with competitive performance which motivate me to put even more effort into it.

Table of Contents

List of Table and Figures

List Of Abbreviations

CTPA – Computer Tomography Pulmonary Angiogram

PE – Pulmonary Embolism

CNN – Convolutional Neural Networks

GUI – Graphic User Interface

AI – Artificial Intelligence

GPU – Graphic processor unit

RSNA - Radiological Society of North America

Introduction

Topic and Scope

The research study at hand has a primary objective of examining the feasibility, consequences, and efficacy of utilizing Convolutional Neural Networks (CNNs) for the detection of Pulmonary Embolism (PE) on Computed Tomography Pulmonary Angiography (CTPA) scans. The study is specifically interested in assessing three CNN models developed from 2014 until the current period and providing a graphic user interface (GUI) that can facilitate the use of them.

Pulmonary Embolism is a relatively common cardiovascular disorder which despite the developments of technology in the last 30 years still has a high mortality rate, patients die in the first hours of presentation. Because of that the timely diagnosis and treatment is paramount (Bělohlávek et al. 2013).

Bělohlávek et al. (2013) cite Torbicki et al. (2008), who confirm that CT pulmonary angiogram (CTPA) has now become the preferred method to evaluate patients with pulmonary embolism. However, as noted by Sofer et al. (2021), a major disadvantage of CTPA is that it requires significant time and expertise from radiologists to assess. Burs et al. (2021) cite McDonald et al. (2015) who estimated that a radiologist has to evaluate an image every three seconds. Additionally, Burs et al. (2021) cite Brady et al. (2017) who demonstrated that due to this high workload, radiologists may miss some diagnoses, resulting in a considerable number of false negatives and misinterpretations.

Convolutional neural networks (CNNs) are a type of deep learning model that are commonly used for image recognition tasks. CNNs are designed to automatically learn and extract relevant features from images through a series of convolutional layers.

In a CNN, the input image is fed through multiple convolutional layers, which apply filters or kernels to the image to extract relevant features. These features are then down sampled or pooled, which reduces the dimensionality of the image and helps to maintain the important

features while discarding the irrelevant ones. The output of the convolutional layers is then passed through one or more fully connected layers, which perform classification or regression tasks based on the learned features.

CNNs have been shown to be highly effective for image recognition tasks such as object detection, face recognition, and medical image analysis. They have been used in a wide range of applications, including self-driving cars, image search engines, and medical diagnosis systems.

Relevance and Importance

Development of computer applications that can detect PE in CTPA scans may improve the radiologists' diagnosis time and precision by comforting them in their decisions.

Burs et al. (2021) provide a summary of studies by Arbabshirani et al. (2017), Prevedello et al. (2017), and Chang et al. (2018), which suggest that Artificial Intelligence (AI) has the potential to reduce the workload of radiologists and enhance the precision of diagnoses. Similarly, Huhtanen et al. (2022) arrived at a comparable conclusion, stating that implementing an automated system to detect PE could assist radiologists in avoiding errors and prioritizing positive cases for swift evaluations.

Convolutional Neural Networks have demonstrated encouraging outcomes both theoretically and practically in the detection of PE in CTPA scans.

In 2021, Sofer et al. conducted a survey of papers that assessed the performances of deep learning models for detecting pulmonary embolism (PE) on computed tomography pulmonary angiography (CTPA) images. They identified seven papers that met the inclusion criteria of being peer-reviewed, and containing measurable outcomes. The studies indicated that CNNs performed similarly to radiologists, with pooled sensitivity and specificity values of 0.88 and 0.86, respectively. In comparison, radiologists had sensitivities ranging from 0.67 to 0.87 and specificities ranging from 0.89 to 0.99.

According to recent research, convolutional neural networks (CNNs) can be smoothly incorporated into real-time workflows and perform comparably to radiologists. In a real-world radiology environment at Universitair Ziekenhuis Brussel, Buls et al. (2021) evaluated the accuracy of Aidoc version 1.3, an FDA-approved CNN-based tool. The model demonstrated an accuracy of 98% and a significant concordance with expert readings, as evidenced by a kappa value of 0.78. However, the false positive rate was 24%, which was largely attributable to factors that made the readings challenging for both the AI and the experts, such as body anomalies, comorbidities, and imaging artifacts.

Methodologies

Development framework

Effective planning is imperative when undertaking software development that employs machine learning techniques, as the latter can entail a significant expenditure of time and resources. In particular, the scarcity and costliness of specialized hardware presents a critical challenge, as it may compromise the quality of model training and subsequently, its overall performance. Failure

to adequately plan and allocate resources can result in a suboptimal outcome, escalated training costs due to the need for additional resources, or inadequate handling of unforeseen issues that may arise during development. Timely delivery of the software product within the agreed upon timeframe is also essential to ensure optimal performance and to attain the best possible evaluation.

A hybrid agile development framework, known as Scrumban, combining elements of Kanban and Scrum development lifecycles, was selected after thorough examination of implementation requirements and overarching aims and objectives. Scrumban enabled the division of work into sprints while ensuring effective task organization. The Kanban component of the framework facilitated task clarification and identification of task accumulation, while the categorization of tasks into “To do”, “Doing”, and “Done” provided insight into project progress. The Scrum component of the framework ensured timely completion of tasks through sprint time frames and facilitated the evaluation of model training through sprint retrospection, which informed future training strategies and architectures.

The integration of both SCRUM and Kanban frameworks was deemed indispensable to circumvent the perils and likely project mismanagement that might ensue if a solitary developer solely employed the SCRUM framework.

The Implementation

During the sprint planning phase, a comprehensive review of objectives was conducted, which included assessing hardware requirements, acquiring equipment, evaluating architecture performance, calculating training possibilities, brainstorming dataset arrangements and model architectures, and reviewing mock-ups for the software's graphical user interface.

In the sprint build phase, training strategies were implemented using the APIs presented below, with the main tasks being training, programming, and writing the current paper, although these were carried out in parallel. The training phase involved an initial programming step, where learning steps were translated into code, followed by actual training, during which notes were taken on the model's evolution. The training time also allowed for other parallel activities, such as GUI implementation, thereby maximizing time efficiency.

During the sprint retrospective, notes taken during the training were reviewed, and insights gleaned from them, that they were used to inform planning for the next sprint. This was an important aspect of development given the imprecise nature of finding a highly performing architecture, and also involved a research component for discovering new aspects of the problem.

Programming Language

Python3 is a high-level programming language that has gained immense popularity in the field of machine learning due to its simplicity, flexibility, and extensive library support. It provides a vast range of powerful tools and frameworks that enable developers to build robust machine learning models with ease.

One of the key advantages of Python3 in machine learning is its ease of use. Python3 has a simple syntax that is easy to read and understand, making it ideal for beginners and experts alike. It also supports a wide range of data structures, including lists, dictionaries, and tuples, which makes it easier to manipulate data.

Another advantage of Python3 is its extensive library support. Python3 has numerous libraries specifically designed for machine learning, including TensorFlow, Scikit-learn, and Keras, which provide powerful tools for building and deploying machine learning models. These libraries provide an array of pre-built models and algorithms, saving developers significant amounts of time and effort.

Python3 also supports object-oriented programming, which allows developers to create reusable code and build complex applications with ease. Additionally, Python3 has a vast online community that provides support, resources, and documentation, making it easier for developers to learn and build machine learning models.

In summary, Python3 is a powerful language that provides developers with the necessary tools and frameworks to build robust machine learning models. Its simplicity, flexibility, extensive library support, and community make it an ideal language for machine learning projects. Utilizing C++ or Javascript for training would have resulted in significant slowing down of the process, as every model would have had to be built from scratch, among other smaller drawbacks.

Environment

Kaggle is a popular platform for data scientists and machine learning enthusiasts that provides a diverse set of datasets and challenges to solve using machine learning techniques. Using Kaggle can offer several benefits to machine learning practitioners, including:

Access to Diverse Datasets: Kaggle offers a vast collection of public datasets that cover a broad range of topics and industries, such as healthcare, finance, and transportation. These datasets are an excellent resource for machine learning practitioners to explore and build models on.

Learning Resources: Kaggle also provides an extensive library of notebooks, tutorials, and other learning resources to help users develop their skills and stay up-to-date with the latest machine learning techniques.

Hardware: Kaggle offers four accelerator options to speed up your training time as machine learning can be a demanding activity in terms of processing power and not only.

Frameworks

Tensorflow

TensorFlow is a popular open-source library for machine learning that offers a range of tools and capabilities to develop and train CNNs . Here are some reasons why TensorFlow was used for training the CNNs:

High Performance: TensorFlow's computational graph system is designed to optimize computation for high performance, making it an excellent choice for training large and complex CNN models. TensorFlow is also optimized for GPUs, allowing for faster training times.

Flexibility: TensorFlow offers a high degree of flexibility, allowing developers to build custom CNN models that fit their specific needs. It also provides a wide range of pre-built CNN models, including popular architectures such as VGG, ResNet, and Inception. Throughout this feature, different architectures were created.

Ease of Use: TensorFlow has a user-friendly API that makes it easy for developers to build, train, and deploy CNN models. The library provides a range of functions and tools that simplify the process of building and training CNN models. It was fast and easy to acquire the knowledge needed in using the library through simple examples but also offering comprehensive details were interested.

Extensive Community Support: TensorFlow has a vast community of users who contribute to the development and improvement of the library. This community offers extensive support through forums, documentation, and tutorials, making it easier for developers to learn and use TensorFlow. The code was easy to debug finding topics on the forums quickly.

Integration with Other Libraries: TensorFlow integrates seamlessly with other popular machine learning libraries such as Keras and PyTorch. This integration allows developers to take advantage of the unique features of each library and build more powerful CNN models.

In summary, TensorFlow is an excellent choice for training convolutional neural networks due to its high performance, flexibility, ease of use, extensive community support, and integration with other popular machine learning libraries.

Keras

Keras is a high-level neural network API written in Python that allows developers to easily build, train, and deploy deep learning models. It is designed to be user-friendly, modular, and extensible, making it a popular choice among researchers and developers in the machine learning community. Keras provides a simple, intuitive interface for constructing and configuring neural networks, as well as a wide range of pre-built layers and models that can be easily customized and combined. It also supports a range of popular backends, including TensorFlow, Theano, and Microsoft Cognitive Toolkit, allowing users to choose the best backend for their specific use case. Additionally, Keras supports a range of advanced features such as multi-GPU training, distributed training, and custom callbacks, making it a versatile tool for developing and deploying deep learning models. Overall, Keras is a powerful and flexible tool for building deep learning models, whether for research or production use cases.

Keras provided access to the architecture's code of the tested CNN while offering the possibility to use the hardware resources at maximum being GPUs or TPUs. It offered tools in organizing the dataset for training and validation, classes along with comprehensive set of optimizes and performance metrics. Without it the development of this project would have been impossible in the time frame given.

PyQt5

PyQt5 is a Python binding of the popular GUI toolkit, Qt. It allows developers to create desktop applications with a modern and intuitive user interface. PyQt5 provides a set of Python modules that can be used to create graphical interfaces using the Qt framework. It offers a wide range of features, such as drag and drop, signals and slots, native-looking widgets, internationalization, and more.

With PyQt5, developers can create applications that run on multiple platforms, including Windows, Linux, and macOS. The framework also supports different programming paradigms, including imperative, object-oriented, and event-driven programming.

PyQt5 is easy to learn and use, with comprehensive documentation and a large community of users and contributors. It is also highly customizable, allowing developers to create their own custom widgets and extend the functionality of the framework.

Overall, PyQt5 is a powerful and versatile framework for building desktop applications with a modern user interface in Python.

Models

Three models were chosen for comparison, Xception, VGG19 and InceptionRestV2.

Xception: Xception is a deep convolutional neural network that was introduced by Francois Chollet in 2016. It is a modified version of the Inception architecture that uses depth wise separable convolutions, which are more computationally efficient and require fewer parameters. Xception has achieved state-of-the-art performance on several image classification tasks, including the ImageNet dataset. Xception has 22 layers and 36 million parameters.

VGG19: VGG19 is a deep convolutional neural network which was introduced by Karen Simonyan and Andrew Zisserman in 2014. VGG19 has 19 layers, including 16 convolutional layers and 3 fully connected layers. VGG19 has achieved state-of-the-art performance on several image classification tasks, including the ImageNet dataset. VGG19 has approximately 143 million parameters.

InceptionResNetV2: InceptionResNetV2 is a convolutional neural network that was introduced by Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke in 2017. It is a hybrid architecture that combines the Inception and ResNet architectures, which are both well-known for their high performance on image classification tasks. InceptionResNetV2 has achieved state-of-the-art performance on several image classification tasks, including the ImageNet dataset. It has approximately 55 million parameters.

In summary, Xception, VGG19, and InceptionResNetV2 are all powerful convolutional neural network models that have achieved state-of-the-art performance on several image classification tasks. These models have different architectures, depths, and numbers of parameters, which may affect their performance on specific tasks.

Metrics

In the detection of pulmonary embolism, it is crucial to have a high-level precision and recall.

Precision refers to the proportion of true positives identified by the model to the total number of positive results reported by it. In other words, it indicates how little of an increased workload the model generates for doctors. If the model has a high level of false positive that increases the workload of the doctor by having to reassess the image. A 100% precision indicates that all the cases predicted with pulmonary embolism were true. This can reduce the treatment time by avoiding putting in treatment healthy patients.

Recall indicates the proportion of positive cases of pulmonary embolism that the model can predict out of the total number of positive cases. A 100% recall means that out of 100 individuals with pulmonary embolism all were identified, which can reduce the mortality rate of the disorder.

Therefore, precision and recall are important metrics to consider when selecting a champion model.

Dataset

The dataset utilized in this study is a sample extracted from The RSNA Pulmonary Embolism CT Dataset, a publicly available resource for researchers interested in developing machine learning techniques for identifying pulmonary embolism (PE). Owned by the Radiological Society of North America, this dataset is considered one of the most complex in its field, with data collected from institutions in five different countries to ensure a diverse population, imaging equipment, and anatomic coverage. Comprised of axial soft-tissue window images from chest CT scans performed using a pulmonary angiography protocol, this dataset contains 1,790,594 images from 7279 patients. To label the data, a group of 86 volunteers were selected based on their previous experience and thoroughly tested. The data has been curated to protect patient identity.

The original dataset was delivered as a zipped file with a size of 450 gigabytes and contained both the training and testing sets. However, only the training set was used in this study, as the labels were only provided for this portion. A sample of 174,445 images was extracted, organized into two classes: PE_PRESENT and PE_NOT_PRESENT, as required by the API used for training. Due to the difficulty in handling the full dataset, which had a size of 980 gigabytes, only approximately 10% of the dataset was utilized for this study.

Obtaining and preparing the dataset for training was a time-consuming process, taking approximately 24 hours to download and another 12 hours to unzip the file. The images were delivered in DICOM format and needed to be converted to png files for ease of use. Although the API used offers tools for conversion to the required format for training, converting them to

images reduced training time and dataset size, as DICOM files contain additional information about the patient, such as sex and age.

After sorting the images into two classes, the dataset was highly unbalanced, with around 80,000 images for the PE_PRESENT class and 1,000,000 images for the PE_NOT_PRESENT class. Given that the dataset needed to be uploaded to Kaggle for training, which would take a considerable amount of time, balancing the dataset by reducing the size of the majority class was deemed necessary.

Another reason for which the reduction in the size of dataset was necessary was due to limitations in access time to the specialized hardware for machine learning on Kaggle. That means that the training time was limited thus balancing the dataset was a good way to ensure that the model is trained on positive cases as much as on negative cases in the time given.

In the end the dataset has a size of fifteen gigabytes and 87 222 images per class. The dataset was divided 80% for training and 20% for validation.

Hardware

An initial attempt at training was done using a Nvidia RTX 3070 Laptop GPU 8 gigabytes and 16 gigabytes of RAM but the training time was too long to obtain satisfactory results in the time frame allocated. A faster alternative was necessary. In the end Kaggle platform was used because it offers free but limited processing power. It offers 4 different accelerators, the fastest one and the one that was chosen for the training of the models was TPU VM v3-8. The TPU VM v3-8 is a type of virtual machine (VM) provided by Google Cloud Platform (GCP) that is optimized for running machine learning workloads using Google's Tensor Processing Unit (TPU) hardware accelerators. This VM offers 8 vCPUs, 64 GB of memory, and can be configured with up to 8 TPUs per VM. The TPUs are specialized hardware accelerators designed to perform tensor operations commonly used in deep learning, which can dramatically speed up the training and inference of machine learning models. The TPU VM v3-8 is ideal for large-scale machine learning workloads that require high-performance computing and efficient use of resources. Using this accelerator, the training time per epoch went down to an average of 5 minutes per epoch from one hour in the first case.

Training

When training a model different configuration in the structure of it must be considered as there is no rule in how many hidden layers a model should have. Because the model is not trained from scratch due to the high requirements in hardware, time and training datasets, transfer learning was used. Both methods are common practices where a model is pretrained on a dataset which has a significant number of records for feature extraction and then trained further on the dataset of interest, Menegola et al. (2016) as cited by Litjens et al. (2017) . The weights obtained during the training on ImageNet dataset were used for the pretrained model. The reason for using these weights is because ImageNet contains a diverse set of images and object classes summing up more than 1 million labeled images, so the pre-trained weights can capture a wide range of features that can be useful for this task.

For each model the top layer was removed as it contains the neurons for the labels for which the model was initially trained and the rest of the weights frozen. After removing the top an average pooling layer was added on top to convert the output of the model in scalars that can be used as input for the final layer. The final layer or the output layer of the CNN is composed out of a single neuron as expected result is a number between 0 and 1 representing the probability of existence of PE in the image. The sigmoid function was used for the output layer activation function as it produces values between 0 and 1, which can be easily interpreted as percentages.

To explore the effect of adding a new layer on performance, a 1024 neuron layer with the ReLU activation function was added between the average pooling layer and the output layer, and the models were trained and compared again. The training was extended to 20 epochs, as the results obtained during the initial 10 epochs were promising.

A learning rate of 0.0005 was used for both training sessions to ensure a slow and accurate convergence of the models, avoiding overfitting.

The optimizer used was Adam, which is a stochastic gradient descent optimization algorithm that combines the benefits of AdaGrad and RMSProp. Adam is a computationally efficient and effective optimization algorithm for deep learning tasks, including image classification, object detection, and natural language processing.

Code

The objected oriented paradigm is used in coding the application. This paradigm was chosen for a better code organization and for possibilities of expanding to current features of the application.

Application

Five classes were created for handling each aspect of it.

Main Window class inherits from QWidget class. QWidget is a fundamental building block of PyQt5 GUI applications, and it represents the main component of the graphical user interface that users can interact with. It is a basic element that represents a rectangular region on the screen. The QWidget class provides many methods that allow to add the features as drag and drop, resizing the image among the window size and other design elements as the image, progression bar and the file name.

QLabel Class is a GUI widget that displays text or an image on the screen and was used as parent class for CTPAImage, Title and SmallText classes. CTPAImage has the duty of displaying the black area when no CTPA is loaded or the Image otherwise. It also displays the text in the black area according to the state of the application.Title and SmallText classes handle the file name displaying and updating. Using QProgressBar class the PE prediction is displayed.

The extra methods developed allow the conversion of the DICOM in PNG format, enables the resizing of the window and the elements contained, and creates the drag and drop feature along with the validation of the file format.

The complete code can be found in Appendix: Application Code.

Training

Keras made the training of the models and configuration of them simple and quick. It offered the necessary tools to import the dataset, split it into training and validation sets also in classes based on the directory organization of the dataset and organize it into batches. All of this just using one function “tf.keras.utils.image_dataset_from_directory()”.

```
import tensorflow as tf
import os
base_dir="/kaggle"
input_dir = os.path.join(base_dir, "input")
project_dir = os.path.join(input_dir, "binary-adjusted-pe")
dataset_dir = os.path.join(project_dir, "Binary_Adjusted")
batch_size=128
dataset = tf.keras.utils.image_dataset_from_directory(
  directory=dataset_dir,
  validation_split=0.2,
  subset="both",
  seed=123,
  image_size=(512, 512),
  color_mode="rgb",
  label_mode="binary",
  batch_size=batch_size)
AUTOTUNE=tf.data.AUTOTUNE
training_data = dataset[0].prefetch(buffer_size=AUTOTUNE)
valid_data= dataset[1].prefetch(buffer_size=AUTOTUNE)
```

Figure 1:Loading the Dataset

It enabled the possibility to use one of the most recent hardware in deep learning with just three additional lines of code.

```
tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()

# instantiate a distribution strategy
tpu_strategy = tf.distribute.TPUStrategy(tpu)
```

Figure 2:Setting up the TPU 1

```
import tensorflow as tf
# instantiating the model in the strategy scope creates the model on the TPU
with tpu_strategy.scope():
    base_model = tf.keras.applications.InceptionResNetV2(input_shape = (512, 512, 3), include_top = False, weights = 'imagenet')
    base_model.trainable= False
    inputs = keras.Input(shape=(512, 512, 3))
    x=base_model(inputs, training=False)
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    #x = tf.keras.layers.Dropout(0.2)(x)
    layerinfo="base model to output no dropout"
    x = tf.keras.layers.Dense(1024,activation="relu")(x)
    outputs = tf.keras.layers.Dense(1,activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
```

Figure 3:Setting up the TPU 2.

The process of importing and modifying the model was made simple due to the convenience of using pre-existing models with customizable parameters. In just a few lines of code, a variety of models can be imported and configured based on the specific needs of the project. One can easily modify the model by removing the top layer, choosing the pre-trained weights, or freezing the model for transfer learning. The addition of new layers is also facilitated using the provided

functions, such as tf.keras.layers.Dense(), which allows for the specification of parameters such as the number of neurons and the activation function. For a visual representation, please refer to Figure "Creating the architecture of the model".

```
base_model = tf.keras.applications.InceptionResNetV2(input_shape = (512, 512, 3), include_top = False, weights = 'imagenet')
base_model.trainable= False
inputs = keras.Input(shape=(512, 512, 3))
x=base_model(inputs, training=False)
x = tf.keras.layers.GlobalAveragePooling2D()(x)
#x = tf.keras.layers.Dropout(0.2)(x)
layerinfo="base model to output no dropout"
x = tf.keras.layers.Dense(1024,activation="relu")(x)
outputs = tf.keras.layers.Dense(1,activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

Figure 4:Creating the architecture of the model

Before the training, the optimizer, loss function and the metrics of the models have to be set which can be achieved using only this one function “model.compile()”

```
### MODEL FITTING #######
    lr=0.0005
    optimizer="Adam"
    model.compile(tf.keras.optimizers.Adam(learning_rate=lr),
                  loss=tf.keras.losses.BinaryCrossentropy(), metrics=[
                      tf.keras.metrics.AUC(),
                      tf.keras.metrics.BinaryAccuracy(),
                      tf.keras.metrics.Precision(),
                      tf.keras.metrics.Recall(),
                      tf.keras.metrics.TruePositives(),
                      tf.keras.metrics.TrueNegatives(),
                      tf.keras.metrics.FalsePositives(),
                      tf.keras.metrics.FalseNegatives()])
    model.summary()
```

Figure 5:Setting up the loss function, optimizer and metrics

.”

The model was saved every epoch for later comparation and fine-tuning. This was achieved using the function “tf.keras.callbacks.ModelCheckpoint()”. There were other options as saving the model with the highest chosen metric but to be able to choose the model based on overview of the metrics the model is saved after every epoch.

```
import os

working_dir=os.path.join(base_dir,"working")
MODEL_NAME="InceptionResNetV2"
MODEL_SERIES="-1024"
MODELS_PATH=os.path.join(working_dir,MODEL_NAME+MODEL_SERIES+"second-ep-{epoch:04d}.h5")
working_dir
save_every_epoch=tf.keras.callbacks.ModelCheckpoint(
    MODELS_PATH,
    save_freq="epoch",
    verbose=1
)
```

Figure 6:Saving the model every epoch

”

Finally, the model was trained or fitted using the function “model.fit()” allowing to choose the number of epochs for which the model is trained, the dataset, the callback used and how to display the metric values in the console through “verbose” parameter.

```
ep=20
history = model.fit(training_data,validation_data=valid_data,epochs=ep,callbacks=[save_every_epoch], verbose=1)
```

Figure 7:Training function

For fine-tuning the model was loaded using “tf.keras.models.load_model()” and then the weights of the model were unfrozen and trained as regular.

The console played a vital role during training by displaying the model's evolution, which was a key feature of the API, enabling the collection of data required for evaluating the model's progress during the sprint retrospective.

```
Epoch 1: saving model to /kaggle/working/InceptionResNetV2-1024second-ep-0001.h5
1091/1091 [==============================] - 660s 560ms/step - loss: 13.9986 - auc: 0.5858 - binary_accuracy: 0.5758 - precision: 0.5768 - recall: 0.5760 - t
rue_positives: 40271.0000 - true_negatives: 40092.0000 - false_positives: 29553.0000 - false_negatives: 29640.0000 - val_loss: 1.9569 - val_auc: 0.7494 - val
_binary_accuracy: 0.6813 - val_precision: 0.6198 - val_recall: 0.9252 - val_true_positives: 16017.0000 - val_true_negatives: 7752.0000 - val_false_positives:
9826.0000 - val_false_negatives: 1294.0000
Epoch 2/20
1091/1091 [==============================] - ETA: 0s - loss: 4.1369 - auc: 0.6335 - binary_accuracy: 0.6081 - precision: 0.6089 - recall: 0.6086 - true_posit
ives: 42547.0000 - true_negatives: 42316.0000 - false_positives: 27329.0000 - false_negatives: 27364.0000
Epoch 2: saving model to /kaggle/working/InceptionResNetV2-1024second-ep-0002.h5
1091/1091 [==============================] - 516s 472ms/step - loss: 4.1369 - auc: 0.6335 - binary_accuracy: 0.6081 - precision: 0.6089 - recall: 0.6086 - tr
ue_positives: 42547.0000 - true_negatives: 42316.0000 - false_positives: 27329.0000 - false_negatives: 27364.0000 - val_loss: 4.4612 - val_auc: 0.6226 - val_
binary_accuracy: 0.5100 - val_precision: 0.5031 - val_recall: 0.9995 - val_true_positives: 17302.0000 - val_true_negatives: 490.0000 - val_false_positives: 1
7088.0000 - val_false_negatives: 9.0000
Epoch 3/20
1091/1091 [==============================] - ETA: 0s - loss: 2.2541 - auc: 0.6441 - binary_accuracy: 0.6150 - precision: 0.6152 - recall: 0.6176 - true_posit
ives: 43178.0000 - true_negatives: 42642.0000 - false_positives: 27003.0000 - false_negatives: 26733.0000
Epoch 3: saving model to /kaggle/working/InceptionResNetV2-1024second-ep-0003.h5
1091/1091 [==============================] - 515s 472ms/step - loss: 2.2541 - auc: 0.6441 - binary_accuracy: 0.6150 - precision: 0.6152 - recall: 0.6176 - tr
ue_positives: 43178.0000 - true_negatives: 42642.0000 - false_positives: 27003.0000 - false_negatives: 26733.0000 - val_loss: 0.7356 - val_auc: 0.7563 - val_
binary_accuracy: 0.6475 - val_precision: 0.7228 - val_recall: 0.4695 - val_true_positives: 8128.0000 - val_true_negatives: 14461.0000 - val_false_positives:
3117.0000 - val_false_negatives: 9183.0000
Epoch 4/20
 799/1091 [====================>.........] - ETA: 1:48 - loss: 1.2648 - auc: 0.6609 - binary_accuracy: 0.6200 - precision: 0.6193 - recall: 0.6204 - true_pos
itives: 31683.0000 - true_negatives: 31725.0000 - false_positives: 19478.0000 - false_negatives: 19386.0000
```

Figure 8:Console output during training

Kaggle Notebooks proved to be a valuable tool not only for accelerating the execution of the code but also for facilitating its organization. The code was structured in sections, and the global variables remained in memory at the end of each execution, allowing the dataset to be loaded once and reused for different convolutional neural networks (CNNs). By simply changing the model function, it was possible to switch between models, and only the section containing it had to be re-run. The same approach was taken when changing the number of layers, leading to a more efficient workflow.

Dataset preparation

The original dataset was structured as studies comprised of DICOM files, while the corresponding labels were provided in a separate CSV file. To prepare the dataset for training, it was necessary to restructure it into separate directories for each class. Leveraging the capabilities of Python and existing libraries for managing DICOM and CSV files, small pieces of code were developed to convert the data to PNG format and organize it into two distinct directories based on the provided CSV file. The dataset was first converted to PNG format and then organized into classes according to the location of the pulmonary embolism.

```python
def save_to_png(filename, save_path):
    """ converts and saves """
    try:
        img = pydicom.dcmread(filename)
        myImg=img.pixel_array.astype(float)
        rescaled_image=(np.maximum(myImg,0)/myImg.max())*255
        final_image=np.uint8(rescaled_image)
        final_image=Image.fromarray(final_image)
        final_image.save(save_path+".png")
    except:
        return False
    return True
with open(r"F:\CTPA\train.csv", "r") as csvFile:
    csvReader=csv.reader(csvFile)
    next(csvReader)
    for row in reversed(list(csvReader)):
        try:
            dicomFile=pydicom.data.data_manager.get_files(os.path.join(ROOT_PATH,row[0],row[1]),row[2]+".dcm")
            if len(dicomFile)!=0:
                if int(row[3]) == 1:
                    #put it in PE_PRESENT_PATH
                    if int(row[10]) == 1:
                        save_path=os.path.join(PE_LEFT_SIDE_PATH,row[2])
                        save_to_png(dicomFile[0],save_path)
                        #make a copy in the PE_LEFT_SIDE_PATH
                    if int(row[13]) == 1:
                        #make a copy in the PE_RIGH_SIDE_PATH
                        save_path=os.path.join(PE_RIGHT_SIDE_PATH,row[2])
                        save_to_png(dicomFile[0],save_path)
                    if int(row[15]) == 1:
                        save_path=os.path.join(PE_CENTRAL_SIDE_PATH,row[2])
                        save_to_png(dicomFile[0],save_path)
                        #make a copy
                    elif int(row[4])== 1:
                            save_path=os.path.join(PE_NOT_PRESENT_PATH,row[2])
                            save_to_png(dicomFile[0],save_path)
                    #put it in the PE_NOT_PRESENT_PATH
        except:
            pass
csvFile.close()
```

Figure 9:Converting to PNG then separate them accordingly to the PE

After organizing the files in four classes based on the position of the PE a problem was discovered with the organization of the files; a class had to be created for each combination possible with the positions of the PE. Using shutil library the files were copied in the right directories.

```python
import os
import csv
import shutil



csv_train=r"D:\Dataset\train.csv"

PE_PRESENT_PATH=r"D:\Dataset_Binary\PE_PRESENT"
DATASET_FOLDER=r"D:\Dataset_Separated"
PE_ALL=os.path.join(DATASET_FOLDER,"PE_ALL")
PE_LEFT_RIGHT=os.path.join(DATASET_FOLDER,"PE_LEFT_RIGHT")
PE_CENTRAL_RIGHT=os.path.join(DATASET_FOLDER,"PE_CENTRAL_RIGHT")
PE_CENTRAL_LEFT=os.path.join(DATASET_FOLDER,"PE_CENTRAL_RIGHT")
PE_LEFT=os.path.join(DATASET_FOLDER,"PE_LEFT")
PE_RIGHT=os.path.join(DATASET_FOLDER,"PE_RIGHT")
PE_CENTRAL=os.path.join(DATASET_FOLDER,"PE_CENTRAL")

def copy_file(filename,destination):
    try:
        shutil.copy2(os.path.join(PE_PRESENT_PATH, filename+".png"), destination)
    except:
        pass

def sort_file(details):
    if details["left"] and details["right"] and details["central"]:
        copy_file(details["filename"],PE_ALL)
    elif details["left"] and details["right"]:
        copy_file(details["filename"],PE_LEFT_RIGHT)
    elif details["right"] and details["central"]:
        copy_file(details["filename"],PE_CENTRAL_RIGHT)
    elif details["left"] and details["central"]:
        copy_file(details["filename"],PE_CENTRAL_LEFT)
    elif details["left"]:
        copy_file(details["filename"],PE_LEFT)
    elif details["right"]:
        copy_file(details["filename"],PE_RIGHT)
    elif details["central"]:
        copy_file(details["filename"],PE_CENTRAL)

with open(csv_train, 'r') as csvfile:
    csvreader=csv.reader(csvfile)
    next(csvreader)# step over header
    for row in csvreader:
        details = {
            "filename":row[2],
            "left": bool(int(row[10])),
            "right": bool(int(row[13])),
            "central": bool(int(row[15]))
        }
        sort_file(details)
csvfile.close()
```

Figure 10:Re-organizing the images

During the sprint retrospective, it was noted that certain classes were underrepresented, with some of them containing no files. To address this issue, a feasible solution within the given circumstances was to create a binary dataset consisting of two classes: "PE_PRESENT" and "PE_NOT_PRESENT". Due to time constraints, searching for additional files was deemed impractical, but remains a viable approach for future model development. To create the binary dataset, files for the "PE_PRESENT" class were manually copied, while for the "PE_NOT_PRESENT" class, a short piece of code was used to select an equal number of images as in the "PE_PRESENT" class.

```
import os
import shutil

dst = 'D:\Binary_Adjusted\PE_NOT_PRESENT'

basepath = 'D:\Dataset_Binary\PE_NOT_PRESENT'
for index,value in enumerate(os.listdir(basepath)):
    src=os.path.join(basepath, value)
    if os.path.isfile(src):
        if index<87223:
        try:
            shutil.copy(src, dst)
        pass
```

Figure 11: Copying the files for PE_NOT_PRESENT class.

Results

Training

Training the models with the architecture containing only the Average Pooling Layer and the output layer brought good results but not comparable to the other researchers or the radiologists. They were trained for ten epochs. The competition was tight between Xception and VGG19 having similar results.

The values below are the results obtained on the validation set.

Table 1:Results Model simple structure

Model Name	Structure	Loss	Binary accuracy	AUC	Precision	Recall
Xception	Simple	0.4781	0.7651	0.8368	0.7187	0.8652
IncepitonResNetV2	Simple	0.6883	0.7097	0.7692	0.6696	0.8536
VGG19	Simple	0.4809	0.7684	0.8393	0.7157	0.8846

In the second row of training as presented above on hidden layer of 1024 neurons having ReLU as activation functions. Here some satisfying results started to appear with VGG19 standing out by a huge margin. Comparing the models' structure at epoch 10 by adding a hidden layer the performance of the models has improved.

Table 2:Xception Model 1024/ReLU Results

Model Name	Structure/Ac. Func.	No Ep	Loss	Binary accuracy	AUC	Precision	Recall
Xception	1024/ReLU	20	0.3147	0.8621	0.9396	0.8668	0.8532
Xception	1024/ReLU	19	0.3304	0.8547	0.9311	0.8431	0.8689
Xception	1024/ReLU	18	0.3554	0.8428	0.9218	0.8027	0.9060

Xception	1024/ReLU	17	0.3261	0.8550	0.9298	0.8345	0.8827
Xception	1024/ReLU	16	0.3699	0.8406	0.9207	0.7772	0.9516
Xception	1024/ReLU	10	0.3850	0.8219	0.8987	0.7705	0.9130

Table 3:InceptionResNetV2 1024/ReLU Results

Model Name	Structure/Ac. Func.	No Ep	Loss	Binary accuracy	AUC	Precision	Recall
InceptionResNetV2	1024/ReLU	20	0.5873	0.6887	0.7399	0.6237	0.9391
	1024/ReLU	19	0.6140	0.6596	0.7162	0.5975	0.9623
	1024/ReLU	18	0.5671	0.7134	0.7593	0.6584	0.8777
	1024/ReLU	17	0.5740	0.7139	0.7598	0.6714	0.8292
	1024/ReLU	16	0.5709	0.7129	0.7677	0.6779	0.8030

Table 4:VGG19 1024/ReLU Results

Model Name	Structure/Ac. Func.	No Ep	Loss	Binary accuracy	AUC	Precision	Recall
VGG19	1024/ReLU	20	0.2526	0.9002	0.9705	0.8455	0.9774
	1024/ReLU	19	0.2194	0.9111	0.9689	0.8900	0.9365
	1024/ReLU	18	0.2255	0.9059	0.9673	0.8807	0.9374
	1024/ReLU	17	0.2360	0.9005	0.9649	0.9007	0.8987
	1024/ReLU	16	0.2616	0.8914	0.9637	0.8382	0.9679
	1024/ReLU	10	0.3505	0.8362	0.9443	0.9124	0.7410

User Interface

The user interface is intuitive, simple and fast. It is composed of a CTPA area where the image is displayed. A zone with the file name. A prediction bar and a percentage where the probability of PE exitance is converted into a progress bar for a better visualization of it. See FIG. Main Window

A concise warning is displayed shortly after running the application where the terms of use are presented. See picture Fig. Purposes Warning.

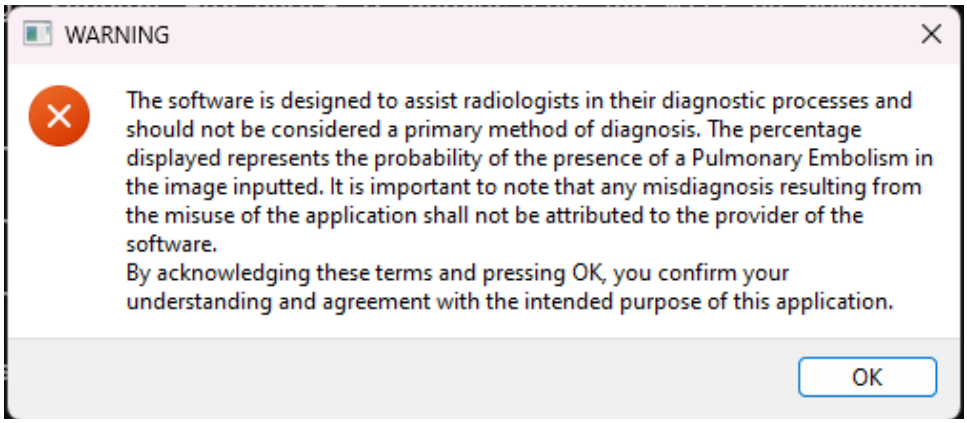

Figure 12:Purposes Warning

The CTPA area is an area marked by the black color and the message “Drop the DICOM file here”. It is meant to display the image of the CTPA for the assessment against the probability provided by the AI. Along with the file name for a better identification of the file for which the prediction is made.

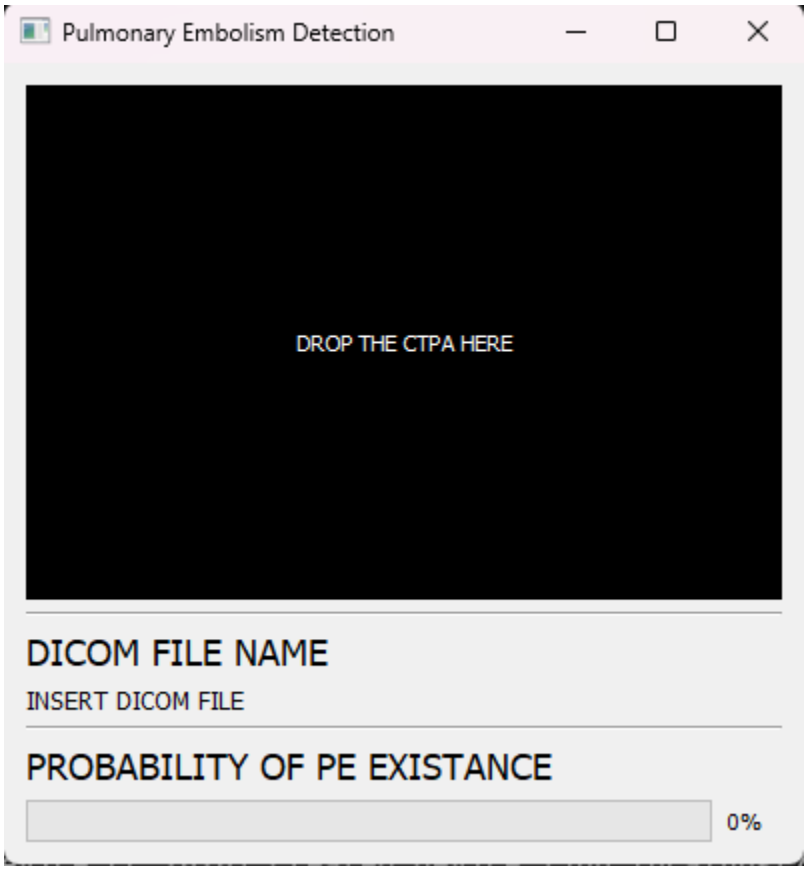

Figure 13:Main Window

The drag on drop feature offers a quick way of inputting the file. The user must insert a DICOM file. If another type of file is inserted the message on the black area is changed to “FORMAT INVALID. ONLY DICOM FILES ARE ACCEPTED”. When a new file is inputted the process repeats; the image is loaded, the file name and the prediction status are updated.

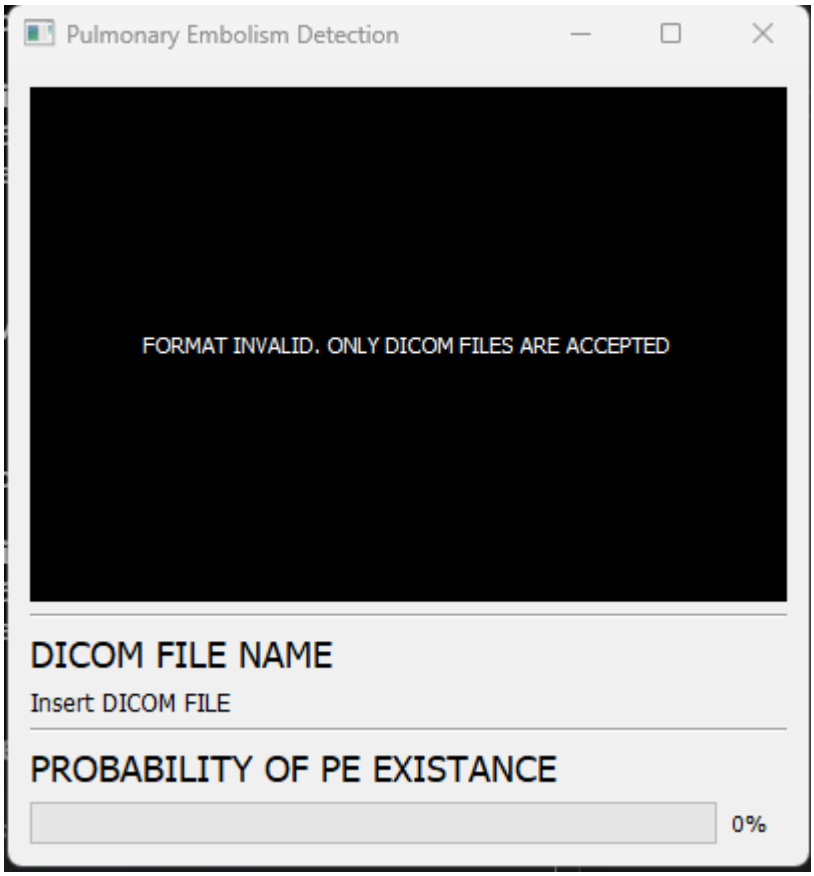

Figure 14:Invalid file inserted

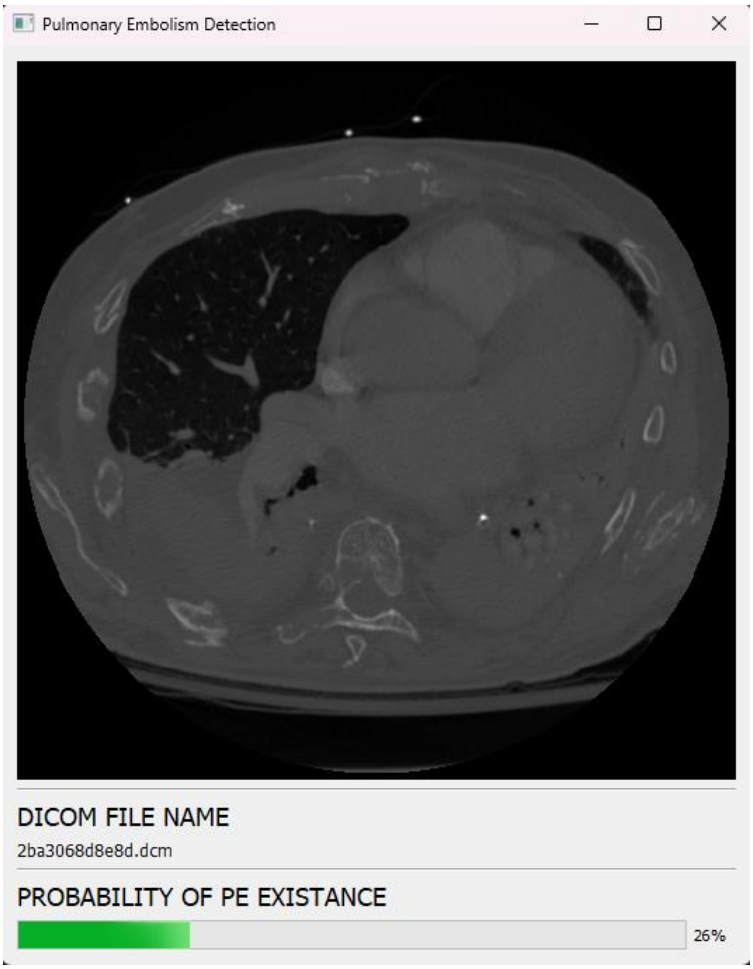

Figure 15: Valid file inserted

The software is initially slow for the first image but for the second image and beyond becomes reasonable fast. The processing time is variable but with a dedicated processor a time under 20 milliseconds can be achieved. Using NVDIA RTX 3070 Laptop the average time per image is 17 milliseconds.

Discussion

The developed application offers a simple means of predicting the presence of pulmonary embolism (PE) in computed tomography pulmonary angiography (CTPA) scans, potentially improving diagnostic turnaround time. To enhance the model's generalization capacity and validity, better training and testing resources are necessary. This entails utilizing the entire dataset, increasing the number of PE_PRESENT CTPA scans, and prolonging training time to experiment with alternative structures, such as multiple hidden layers and varying neuron numbers. Although modest validation testing has been performed, testing the model in a hospital setting in terms of real-time performance, precision, recall, and influence on diagnosis time is necessary to bolster product quality. While the model's graphical interface may slightly prolong diagnosis time, it may also increase accuracy and reassure radiologists in their predictions. Alternatively, integrating the model into the current imaging visualization system rather than a

separate application may be preferable. Having access to a radiology department for testing the final models would be instrumental in evaluating its performance.

The champion model is VGG19 with a hidden layer composed of 1024 neurons. The version obtained at epoch 19 was used for the development of the application as it shows a good balance between precision and recall, 0.8900 and 0.9365, respectively. In other words, out of 100 images the model will have 11 false positives and 6 false negatives. The sensitivity of the model is 0.93%, a specificity of 0.88% performances similar to one of the radiologists of 0.67% to 0.87% sensitivity and 0.89% to 0.99% as discovered by Shoffer et al. 2021 in Kligerman, S. J. et al. 2018 and Eng, J. et al. 2004 research papers.

The performance is comparable as well with the other researchers models found by Shoffer et al. in 2021 and presented in comparation with the champion model in this research in the table below:

Table 5: Performance Comparison

Author	Year	Dataset size	Performance scores
Huang et al.	2020	1997	AUROC of 0.85 Sensitivity and specificity of 75% and 81%
Liu et al.	2020	878	AUC of 0.93 Sensitivity and specificity of 94.6% and 76.5%
Huang et al	2020	1837	AUROC of 0.95 Sensitivity and specificity of 87.3% and 90.2%
Weikert et al.	2019	29,465	Sensitivity and specificity of 92.7% and 95.5%
This paper	2023	174 445	AUC 0.9689 sensitivity 0.93, specificity of 0.88

It can be noticed that the dataset used in this paper is up to 20 times bigger than the smallest one in the list of the previous models. This improves, as pointed out above, the capacity for generalization and it makes a more qualitative model.

The performance of the model could have been slightly improved by cropping out the black zone of the image surrounding the body zone and stretching the resulting image to fit 640 by 640 pixels as this format improves the modeling performance as pointed out by the winner of RSNA STR Pulmonary Embolism Detection 2020 Kaggle Competition GUANSHUO XU in the description of the solution.

The excitement associated with the development of a potentially life-saving product can be intense, leading to ambitious goals. However, a lack of practical experience may impede the realization of these objectives. In the present case, the product under consideration achieves many of its aims by comparing the performance of three models in pulmonary embolism detection provides an interface to ease the use of the model and predicts the presence of the PEs. It falls short in its ability to classify based on embolism localization or to highlight specific areas. While the interface design differs somewhat in terms of elements to the initial design, it includes

crucial features such as presenting the computed tomography pulmonary angiography, the probability percentage of the existence of embolism, and the file name.

This project presented an opportunity to acquire novel insights into machine learning in the Computer Vision domain, leading to a heightened comprehension of the process, key performance drivers, and effective organizational strategies. Employing Keras and Kaggle in project development illuminated the ease of adapting existing models and bolstered motivation to pursue future projects utilizing these tools. Among the widely recognized models, Keras affords the capability to design custom Convolutional Neural Network (CNN) architectures, thereby expanding the potential avenues for exploration.

In Hindsight

If the development project were to be repeated, the final product would likely differ due to the valuable lessons learned. Firstly, a more comprehensive approach to researching datasets would be undertaken, including a use of a mixture of datasets to improve its generalization capabilities. This approach would also involve obtaining more samples for different locations of pulmonary embolisms (PEs), allowing for training on the specification of the position of PE, rather than only predicting its existence.

Another critical element to accelerate the development and improve the training of the models would be to investigate more cloud processing platforms that could enable longer training times on larger datasets, even if it required investment. This would reduce the training time and allow for the testing of more than three models or network configurations.

In addition, a more comprehensive research effort would be conducted to understand the workflow of a radiological department, how images are visualized, and the software used. This would enable the integration of the model into existing software used in hospitals. Moreover, testing the model in a real environment would be the pinnacle of developing such a product.

One key feature for PE identification would be to highlight the areas on the CTPAs scan that represent blood clots. A dataset called FUMPE (Ferdowsi University of Mashhad's PE dataset) would be available for segmentation of the portions representing the blood clots. Segmentation is the process of dividing an image into different regions based on pixel characteristics to identify objects or boundaries, simplifying an image and making it easier to analyze. However, this dataset is relatively small, consisting of only 8792 slices from 35 patients, compared to the dataset used in this project, which includes approximately 160,000 slices from around 12,000 patients across five institutions worldwide. This larger dataset captures more variations in body anatomy, providing a better understanding of the problem. A possible solution to train a more qualitative model capable of highlighting blood clots would be to use both datasets to create an even more comprehensive dataset. However, annotating the regions with blood clots in the Radiological Society of North America (RSNA) dataset would require funding to hire specialists. Additionally, more powerful equipment would be required for this task.

Conclusion

The present development project has successfully delivered a software product that can classify CTPA scans based on the presence of PE. The paper showcases the results of training three different models on a qualitative dataset and demonstrates how a different architecture can enhance their performance. The software includes a GUI that simplifies the model's usage through its minimalistic design. Furthermore, the paper outlines the strengths and weaknesses of the software developed and suggests potential improvements for future research.

The training of this emerging technology involves significant demands for processing power and data acquisition, which can be costly or inaccessible, especially when specialized experts are required for data labeling, and the size of the data and network is considerable. While there are free options for experimenting with deep learning, they are often limited, and as the size of the datasets and models increase, the opportunities for non-funded researchers to experiment become more restricted.

Despite these challenges, the potential of convolutional neural networks (CNNs) in PE detection is substantial, but it is crucial to test this technique in a real-world workflow to assess the effectiveness of using it.

References

SOFER S. et al. (2021) Deep learning for pulmonary embolism detection on computed tomography pulmonary angiogram: a systematic review and meta-analysis. *Sci Rep*. [Online] 11. Available from: https://doi.org/10.1038/s41598-021-95249-3 [Accessed 27/12/22].

BĚLOHLÁVEK, J. et al. (2013) Pulmonary embolism, part I: Epidemiology, risk factors and risk stratification, pathophysiology, clinical presentation, diagnosis and nonthrombotic pulmonary embolism. *Experimental and clinical cardiology*. [Online] 18(2). Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3718593/ [Accessed 27/12/22].

HUHTANEN, H. et al. (2022) Automated detection of pulmonary embolism from CT-angiograms using deep learning. *BMC Medical Imaging*. [Online] 22(1). Available from: https://doi.org/10.1186/s12880-022-00763-z. [Accessed 27/12/2022]

LITJENS G. et al. (2017) A survey on deep learning in medical image analysis.Medical Image Analysis. [Online] 42. Available from: https://doi.org/10.1016/j.media.2017.07.005 [Accessed 27/12/22].

ImageNet (no date). Available at: https://www.image-net.org/about.php. [Accessed 28/02/23].

Sec, I. (2021) "VGG-19 Convolutional Neural Network," All About Machine Learning [Preprint]. Available at: https://blog.techcraft.org/vgg-19-convolutional-neural-network/.

Chollet, F. (2016) Xception: Deep Learning with Depthwise Separable Convolutions. Available at: https://arxiv.org/abs/1610.02357.

Szegedy, C. et al. (2016) "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," arXiv (Cornell University) [Preprint]. Available at: https://doi.org/10.48550/arxiv.1602.07261.

Simonyan K. and Zisserman A. (2015) "VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION". Available at: https://doi.org/10.48550/arXiv.1409.1556.

Riverbank Computing | Introduction (no date). Available at: https://www.riverbankcomputing.com/software/pyqt/. [Accessed 23/04/23].

What is Python? Executive Summary (no date). Available at: https://www.python.org/doc/essays/blurb/. [Accessed 14/02/23].

Abadi M. et al. (2016) “TensorFlow: A System for Large-Scale Machine Learning”. [Online]. Available from: https://www.tensorflow.org/about/bib#tensorflow_a_system_for_large-scale_machine_learning. [Accessed 23/02/23].

Team, K. (no date) Keras documentation: About Keras. Available at: https://keras.io/about/. [Accessed 23/02/23].

Uslu, Ç. (2022) “What is Kaggle?” Available at: https://www.datacamp.com/blog/what-is-kaggle.

Saadatmand-Tarzjan, M. (2018) FUMPE. Available at: https://figshare.com/collections/FUMPE/4107803/1.

Appendix

Application code

```
import sys, os
from PyQt5.QtWidgets import QApplication, QWidget, QLabel,
QVBoxLayout,QProgressBar,QSizePolicy, QFrame, QMessageBox
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPixmap
from PyQt5 import QtCore, QtGui, QtWidgets
import pydicom
import pydicom.data
import numpy as np
from PIL import Image, ImageQt

import tensorflow as tf


class CTPAImage(QtWidgets.QLabel):
    def __init__(self):
        super().__init__()
        self.setAlignment(Qt.AlignCenter)
        self.setText("\n\n DROP THE CTPA HERE \n\n")
        self.setStyleSheet('''
            QLabel{
                    background:black;
                    color:white;
                }

            ''')

    def setPixmap(self, image):
        super().setPixmap(image)

    def updateText(self, text):
        self.setText("\n\n "+text+" \n\n")

class Title(QLabel):
    def __init__(self,text):
        super().__init__()
        self.setText(text)
        self.setStyleSheet('''
            QLabel{
                    font-size: 18px;
```

```
                }

            ''')
        self.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)


class SmallText(QLabel):
    def __init__(self,text):
        super().__init__()
        self.setText(text)
        #self.setAlignment(Qt.AlignBottom)
        self.setStyleSheet('''
            QLabel{
                    font-size: 12px;
                }

            ''')
        self.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)

    def updateText(self, text):
        self.setText(text)


class PredictionBar(QProgressBar):
    def __init__(self):
        super().__init__()
        self.setGeometry(25,25,300,300)
        self.setMaximum(100)
        self.setValue(0)
    def updateValue(self,value):
        self.setValue(value)

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()

        self.currentCTPA=None
        self.modelPath=r".\model.h5"
        self.AI=None

        self.resize(400,400)
        self.setAcceptDrops(True)

        self.show_warning()

        mainLayout=QVBoxLayout()
        self.CTPAViewer=CTPAImage()
        mainLayout.addWidget(self.CTPAViewer)

        horizontal_bar = QFrame()
        horizontal_bar.setFrameShape(QFrame.HLine)  # Set the frame shape to a
horizontal line
        horizontal_bar.setFrameShadow(QFrame.Sunken)  # Set the frame shadow to a
sunken style
        mainLayout.addWidget(horizontal_bar)

        self.IMG_ID_TITLE = Title("DICOM FILE NAME")
        mainLayout.addWidget(self.IMG_ID_TITLE)
```

```
        self.IMG_ID = SmallText("INSERT DICOM FILE")
        mainLayout.addWidget(self.IMG_ID)

        horizontal_bar2 = QFrame()
        horizontal_bar2.setFrameShape(QFrame.HLine)  # Set the frame shape to a
horizontal line
        horizontal_bar2.setFrameShadow(QFrame.Sunken)  # Set the frame shadow to a
sunken style
        mainLayout.addWidget(horizontal_bar2)

        self.PE_PRED_LABEL_PROJECTS_BAR = Title("PROBABILITY OF PE EXISTANCE")
        mainLayout.addWidget( self.PE_PRED_LABEL_PROJECTS_BAR)




        self.PEPrediction=PredictionBar()
        mainLayout.addWidget(self.PEPrediction)
        self.setLayout(mainLayout)
        self.load_model()

    def dragEnterEvent(self, event):
        filename = event.mimeData().urls()[0].fileName()
        if filename[-3:] == "dcm":
            self.CTPAViewer.setText("BON")
            event.accept()
        else:
            self.IMG_ID.updateText("Insert DICOM FILE")
            self.PEPrediction.updateValue(0);
            self.CTPAViewer.setText("FORMAT INVALID. ONLY DICOM FILES ARE ACCEPTED")
            event.ignore()

    def dragMoveEvent(self, event):
        filename = event.mimeData().urls()[0].fileName()
        if filename[-3:] == "dcm":
            self.CTPAViewer.setText("VALID FORMAT")
            event.accept()
        else:
            self.IMG_ID.updateText("Insert DICOM FILE")
            self.PEPrediction.updateValue(0);
            self.CTPAViewer.setText("FORMAT INVALID. ONLY DICOM FILES ARE ACCEPTED")
            event.ignore()
    def dropEvent(self, event):
        url=event.mimeData().urls()[0]
        filename = url.fileName()
        path= url.path()
        if filename[-3:] == "dcm":
            self.IMG_ID.updateText(filename)
            self.CTPAViewer.setText("PROCESSING")
            self.extactCTPA(path,filename)
            event.accept()
            self.set_image(self.currentCTPA)
            self.assess_image()
            self.PEPrediction.updateValue(int(self.results[0][0]*100))
        else:
```

```
            self.IMG_ID.updateText("Insert DICOM FILE")
            self.CTPAViewer.setText("FORMAT INVALID. ONLY DICOM FILES ARE ACCEPTED")

    def resizeEvent(self, event):
        if self.currentCTPA:
            img = ImageQt.ImageQt(self.currentCTPA)  # convert to QImage
            qimage = QtGui.QImage(img)
            qpixmap = QPixmap.fromImage(qimage)
            scaled_pixmap = qpixmap.scaled(self.CTPAViewer.width()-10,
self.CTPAViewer.height()-10,

aspectRatioMode=QtCore.Qt.KeepAspectRatio)
            self.CTPAViewer.setPixmap(scaled_pixmap)

    def show_warning(self):
        msg=QMessageBox()
        msg.setWindowTitle("WARNING")
        msg.setText("The software is designed to assist radiologists in their
diagnostic processes and should not be considered a primary method of diagnosis. The
percentage displayed represents the probability of the presence of a Pulmonary
Embolism in the image inputted. It is important to note that any misdiagnosis
resulting from the misuse of the application shall not be attributed to the provider
of the software. \nBy acknowledging these terms and pressing OK, you confirm your
understanding and agreement with the intended purpose of this application.")
        msg.setIcon(QMessageBox.Critical)
        x = msg.exec_()


    def set_image(self, image):
        pixmap = self.get_pixmap(image, self.CTPAViewer.size())
        self.CTPAViewer.setPixmap(pixmap)
        x = msg.exec_()


    def set_image(self, image):
        img= ImageQt.ImageQt(image)
        self.CTPAViewer.setPixmap(QPixmap.fromImage(img))

    def extactCTPA(self,img_path, filename):
        base=img_path[1:-len(filename)]
        pass_dicom=filename #get the last item in the list which is an string and
eliminate last char.
        filename=pydicom.data.data_manager.get_files(base,pass_dicom)[0]
        img = pydicom.dcmread(filename)
        myImg=img.pixel_array.astype(float)
        rescaled_image=(np.maximum(myImg,0)/myImg.max())*255
        final_image=np.uint8(rescaled_image)
        self.currentCTPA=Image.fromarray(final_image)



    def load_model(self):
        self.AI=tf.keras.models.load_model(
                self.modelPath, compile=False)

    def assess_image(self):
        img_rgb = self.currentCTPA.convert('RGB')
        img_array = tf.keras.preprocessing.image.img_to_array(img_rgb)
```

```
        tensor_img = tf.convert_to_tensor(img_array)
        tensor_img = tf.reshape(tensor_img, [1, 512,512, 3])
        self.results=self.AI.predict(tensor_img)


if __name__ == '__main__':
    app =QApplication(sys.argv)
    mainW = MainWindow()
    mainW.setWindowTitle("Pulmonary Embolism Detection")
    mainW.show()
    sys.exit(app.exec_())
```