UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ROMÂNĂ

LUCRARE DE LICENŢĂ

MammoDetect: Deep Learning Techniques for Breast Cancer Detection: A Comparative Study of GoogLeNet and EfficientDet in 3D Imaging

Conducător ştiinţific
prof. Dioşan Laura

Absolvent
Baciu Ioana

2024

ABSTRACT

The purpose of this thesis is to experiment and compare how a GoogLeNet architecture will perform in classifying breast images into normal or cancerous. Along with the classification problem, a second model representing an EfficientDet architecture focused on detecting the tumors in the aforementioned images and also classifying them into benign or malign. The best performing models from each problem have been selected and integrated into an intelligent system that aims to showcase the use of the two models in real life scenarios.

In order to get the best configuration for each architecture I conducted many experiments on each one and documented them in chapter 4. The detailed description of the architecture structure for both GoogLeNet and EfficientDet have been written in chapter 3. Other ways of solving breast cancer classification and detection are presented in chapter 2. Chapter 5 explains the evolution of the system and finally, chapter 6 encompass the conclusions of the thesis and how to improve the performance of the models resulted after the experiments.

Contents

Chapter 1

Introduction

1.1 Motives for studying breast cancer detection

Breast cancer is known to be the most common cancer among women [18]. The American Cancer Society's estimates for breast cancer in the United States alone for 2023 are that about 297,790 new cases of invasive breast cancer will be diagnosed in women and about 43,700 women will die from breast cancer [19]. Since 1989, breast cancer death rates have been decreasing, believed to be a consequence of early detection and increased awareness, as well as better treatments. This progress seems to have slightly stopped [19].

The main objective of this project is to understand and implement ways to detect the presence of cancerous, benign, or precancerous tumors in the breast in an efficient way using machine learning. This would minimize the time doctors spend studying thousands of breast screenings to label them accordingly and aid early detection.

1.2 Paper structure

The definition of the two problems, classification and detection models, are represented in chapter 2. Along with the problem definition, the documentation of other papers and datasets focused on detecting breast cancer is also present in chapter 2. Chapter 3 contains the details of the two architectures used for experimenting with, GoogLeNet and EfficientDet, by showcasing original concepts and layers used in the structure of the models. The experiments made are documented in chapter 4, along with presenting the dataset used and data processing. The design and implementation of the intelligent system are written in chapter 5, and the conclusion of the experiements and possible improvements for achieving better performance are presented in chapter 6.

1.3 Original contributions

The following thesis provides original contributions to the world of machine learning. Some of these contributions are: comparative analysis of the literature, by comparing datasets, models and performances and training and testing of various classification and detection models, followed by a bi-objective comparison. One criterion is the loss function, another possible criterion is the input type(2D or 3D). Along with these, the design and implementation of the system are another contribution, because of the intelligent system that integrates the best classification and detection models. Also, the comparative analysis of the trained models with SOTA models represents another original contribution of this thesis.

1.4 Use of generative AI instruments

During the process of this thesis, I used a Grammar Checker, Quillbot, to flag the grammatical errors made in writing this thesis. Each modification proposed by the tool were aproved by me.

Chapter 2

Problem definition and existing methods of detecting breast cancer

2.1 Problem definition

The problem of detecting breast cancer consists of two main steps: classification and detection. Classification is the process in which, when given an image, the trained model returns a integer value label, representing the class in which the image is predicted to belong to. For the specific problem of detecting breast cancer, a label would take values from 0 to 1, representing normal and cancerous, or to 2, representing cancer, benign and malign. Detection also categorizes the images, but also returns bounding boxes that are predicted to contain the tumor in the image.

For testing the predictions of a classification model, the most used methods are accuracy, precision and recall, or confusion matrices from which all three values can be calculated from. The formula for these values:

$$acc = \frac{TP}{TP + TN + FP + FN},\ precision = \frac{TP}{TP + FP},\ recall = \frac{TP}{TP + FN}$$

.

The detection model on the other hand, can be tested bu computing IoU, Intersection over Union, that calculates the difference between the predicted image and the true image. For this particular problem, this can be achieved by comparing the original image with the true bounding boxes drawn onto it and the same original image with the predicted bounding boxes drawn. The formula for this metric is:

$$IoU = \frac{Area\ of\ overlap}{Area\ of\ union}$$

.

2.2 Databases used in detecting breast cancer

In the "Existing methods of detecting breast cancer" section of this thesis I mentioned the WDBC dataset that was used in detecting breast cancer in other research papers. This dataset, also known as Breast Cancer Wisconsin Diagnostic, is found on the website Kaggle, and consists of features extracted from digitized images of a fine needle aspirate, FNA, of a breast mass. The characteristics described, those being ten real values recorded with four significant digits, belong to the cell nuclei found in the images [17]. Figure 2.1 shows the distribution of the classes among the 569 instances.

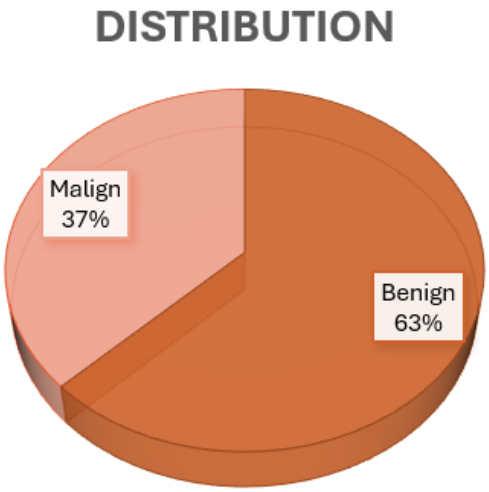


Figure 2.1: Distribution of the classes of the WDBC database among the 569 inputs

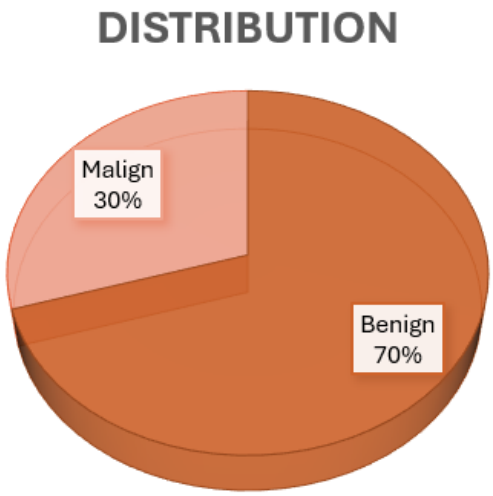


Figure 2.2: Distribution of the classes of the Breast Cancer database among the 286 inputs

Other datasets used in breast cancer detection include the Breast Cancer dataset obtained form the University Medical Center, Institute of Oncology, Ljubljana, Yugoslavia. It contains 286 inputs, each with 9 features: class, age, menopause, tumour-size, inv-nodes, node-caps, deg-malig, breast, breast-squad and irradiat [10]. Figure 2.2 shows the distribution of the classes.

2.3 Existing methods of detecting breast cancer

There are several articles related to studying various ways of implementing breast cancer detection using machine learning. There is "Machine Learning Techniques for Breast Cancer Prediction" by Varsha Nemade and Vishal Fegade from Mukesh Patel School of Technology Management and Engineering, NMIMS Shirpur Campus, India [1]. They have documented different ML classification techniques and evaluated each of them using different performance measures, such as accuracy, precision, and recall. These techniques include Näive Bayes, Logistic Regression, Support Vector Machine, K-Nearest Neighbour and Decision Tree, the latter being found to have the highest accuracy, 97%. The researchers who conducted this experiment worked with benign and malign classes.

The dataset used in their experiments was the WDBC dataset, which contains features from 569 digitized images of a fine needle aspirate of a breast mass [20]. The algorithm implemented uses features from the image rather than the image itself, some of which include radius, texture, area, perimeter, smoothness, compactness, concavity, concavity points, symmetry and fractal dimension. The following charts represent the performance of the classification techniques used.

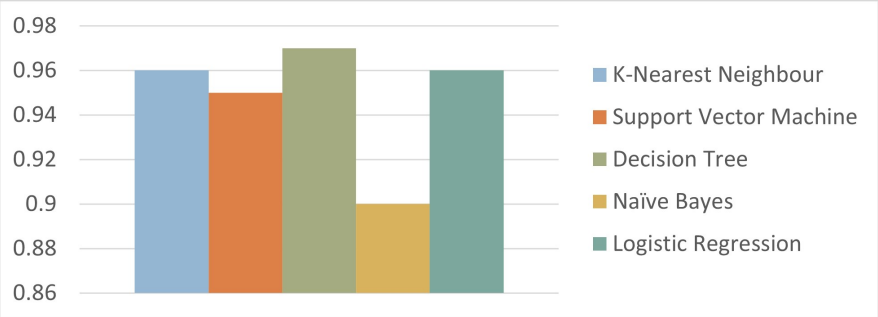


Figure 2.3: Accuracy of classification techniques obtained in [1]

Other relevant studies include "An enhanced Predictive heterogeneous ensemble model for breast cancer prediction" concluded by S. Nanglia et al. which got 78% accuracy using KNN, SVM and DT [13]. Islam et al. got a 98.75% using ANN on the WDBC [7]. Amrane et al. proposed an approach using KNN and NB with

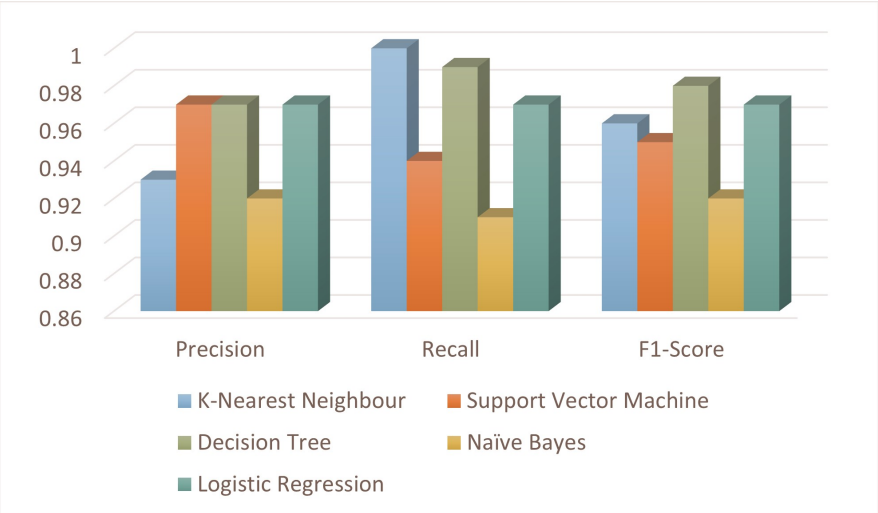


Figure 2.4: Performance of classification techniques for class benign obtained in [1]

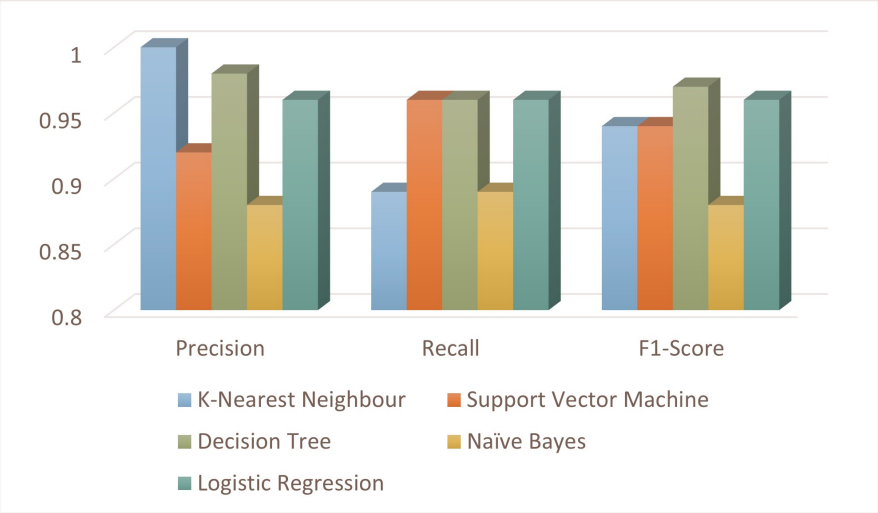


Figure 2.5: Performance of classification techniques for class malignant obtained in [1]

97.51% accuracy [5]. Dhahri et al. studied the usage of genetic programming techniques for the selection of the best features and parameters for the machine learning classifier [4].

Chapter 3

Backgrounds in classification and detection architectures

In the interest of improving image classification, there have been many models documented, each trying to be better than the previous ones. The most commonly used include ResNet, DenseNet, VGGNet, AlexNet and GoogLeNet [3]. The same can be said about detection models; some of the most popular in 2024 are YOLO (You Only Look Once), RetinaNet, Faster-RCNN and EfficientDet [16].

Of all of these models, I chose to integrate into my system GoogLeNet for classification and EfficientDet for detection.

3.1 GoogLeNet Architecture

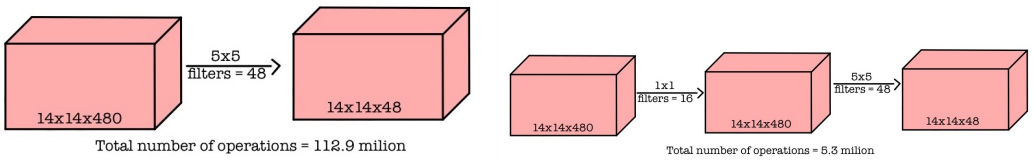
This models first appereance was in an research paper published in 2014 called "Going Deeper with Convolutions" [14]. The researches at Google, along with other universities, received that year the first place in the ILSVRC 2014 image classification competition, by achieving a less error rate than the other contestants. They managed this by introducing new concepts, such as 1x1 convolution, global average pooling, inception module and auxiliary classifiers. Table 3.1 shows the distribution of the layers in the GoogLe Net architecture.

Type	Path size/Stride	Output size	Depth	#1x1	#3x3 reduce	#3x3	#5x5 reduce	#5x5	pool proj	params	ops
Convolution	7x7/2	112x112x64	1							2.7K	34M
Maxpool	3x3/2	56x56x64	0								
Convolution	3x3/1	56x56x192	2		64	192				112K	360M
MaxPool	3x3/2	28x28x192	0								
Inception(3a)		28x28x256	2	64	96	128	16	32	32	159K	128M
Inception(3b)		28x28x480	2	128	128	192	32	96	64	380K	304M
Maxpool	3x3/2	14x14x480	0								
Inception(4a)		14x14x512	2	192	96	208	16	48	64	364K	73M
Inception(4b)		14x14x512	2	160	112	224	24	64	64	473K	88M
Inception(4c)		14x14x512	2	128	128	256	24	64	64	463K	100M
Inception(4d)		14x14x528	2	112	144	288	32	64	64	580K	119M
Inception(4e)		14x14x832	2	256	160	320	32	128	128	840K	170M
Maxpool	3x3/2	7x7x832	0								
Inception(5a)		7x7x832	2	256	160	320	32	128	128	1072K	54M
Inception(5b)		7x7x1024	2	384	192	384	48	128	128	1388K	71M
Avgpool	7x7/1	1x1x1024	0								
Dropout(40%)		1x1x1024	0								
Linear		1x1x1000	1							1000K	1M

Table 3.1: Layer structure of the GoogLeNet architecture

3.1.1 1x1 convolution layer

The use of 1×1 convolution reduces the overall number of parameters of the model [21]. The effect of this operation is increasing the depth of the model, which can help catch and understand complex patterns in the input images. Images 3.1 showcase how many parameters result from a 5×5 convolution without a 1×1 convolution as intermediate compared to one with the 1×1 convolution.



(a) Without intermediate (b) With intermediate

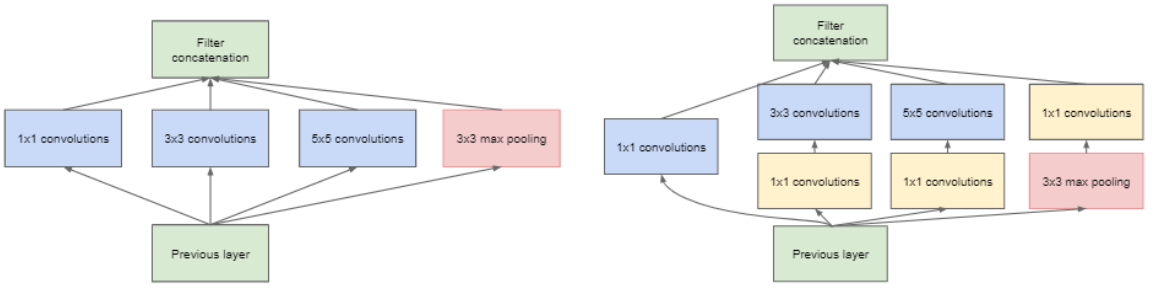
Figure 3.1: Difference in parameters between convolution operations

3.1.2 Global Average Pooling

This method is also a method of decreasing the number of trainable parameters, which then reduces computation costs. Other architectures place their fully connected layers at the end of the network, which contain the largest number of parameters. To combat this, GoogLeNet proposes the global average pooling method, also used at the end of network, which takes a 7×7 feature map and reduces it to 1×1. Along with reducing the number of trainable parameters, this also increases accuracy by 0.6% [21].

3.1.3 Inception Module

While GoogLeNet is not the first architecture to use inception modules in it's configuration, it is the first to set the convolution size for each layer. In these modules, 1×1, 3×3 and 5×5 along with 3×3 max pooling layers are performed in parallel at the input, while the output of these convolutions are stacked in order to obtain one final output. The idea behind this is that convolution layers of different filter sizes will manage objects at multiple scales better [21]. Image 3.2 provides two examples of possible inception module concepts, näive approach and the reduced dimensions one, the latter of which is used in the GoogLeNet architecture.



(a) Inception module, naïve version

(b) Inception module with dimension reductions

Figure 3.2: Difference between the two main concepts of the inception module, obtained from [14]

The main purpose of this second concept, employed by the GoogLeNet architecture is overall reduction of parameters. Adding the reduction of dimensions provides a 60% decrease in number of parameters [2].

3.1.4 Auxiliary Module

These modules, two in total, are used only during the training stage of the model. They help in combating gradient vanishing problem and providing regularization.

3.2 EfficientDet Architecture

The EfficientDet model is a new family of object detectors based on two main optimizations, those being efficient multi-scale feature fusion and model scaling. Along with those optimizations, different models have been tested to serve as the backbone of the model, and for this particular model, EfficientNet has proven to be a sufficient backbone architecture [8].

The first challenge of this model was feature fusion, a concept introduced to aid convolutional neural networks in identifying objects from images when the objects' sizes are either too small or exceed the convolutional kernel's receptive field. [12]. It has been discovered that changing the resolution of the image solved the sensitivity that neural networks have to the images' size, to some extent, as shown in figure 3.3.

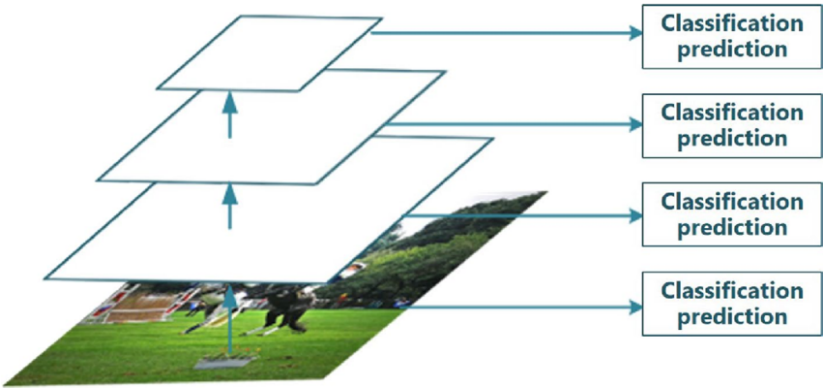


Figure 3.3: Pattern diagram of image pyramid, obtained from [22]

The problem with this discovery was that the cost of system storage for creating a pyramid of the same image in different resolutions and computational resources are too harsh, therefore deeming this method rarely usable. Because each resolution had its advantages, a way to combine features from high-resolution features of shallow networks with the high-level semantic information of high-level network features was researched. Thus, the FPN, or Feature Pyramid Network, has been used to extract from deep-layer networks and get the same features as the shallow-layer features by up-sampling, as shown in figure 3.4.

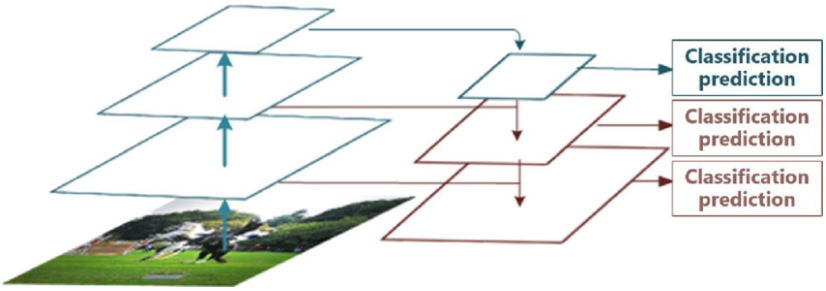


Figure 3.4: Pattern diagram of feature pyramid network, obtained from [23]

In this particular model, a BiPFN has been introduced, Bi-directional FPN, which utilizes learnable weights to learn the importance of different input features from different resolutions.

The second challenge is model-scaling. This method was popularized in [9], and is used to uniformly scale all dimensions of depth, width and resolution using a

compound coefficient. This scaling method, called compound scaling, is looking at how the different dimensions interact with each other and scales them accordingly. For the EfficientDet architecture, this scaling method is used to scale up resolution, width and depth for all layers of the backbone, BiFPN and the bounding boxes and classification network.

Finally, for the backbone, EfficientNet has been observed to achieve better accuracy than other previously used backbones [8]. The baseline strategy for a standard FPN is to aggregate the features from the different levels of resolution in a top-down manner, from the features of the image with the lowest resolution to the ones from the highest one. For example, if a feature is extracted from the lowest resolution image, that feature is passed through a convolutional layer, used for feature processing, and the resulting output is then resized, which could be upsampling or downsampling in order to match the resolution of the following feature and so on [8].

The conventional FPN is, however, limited by the one-directional feature flow. To correct this use, PANet, or Path Aggregation Network, is introduced and adds a path going bottom-up. These methods are still not ideal because of the cost of computations and parameters needed. The strategy proposed in [8] says that nodes that have only one edge going into them should be removed because the lack of multiple input edges corresponds to less contribution to the overall feature network as it does not involve feature fusion. Moreover, between every input and output node at the same level, there are edges added in order to add more feature fusion without much cost, as it is on the same level. Finally, each bidirectional path is treated as one singular layer and more layers are involved in order to achieve more high-level feature fusion, as shown in figure 3.5.

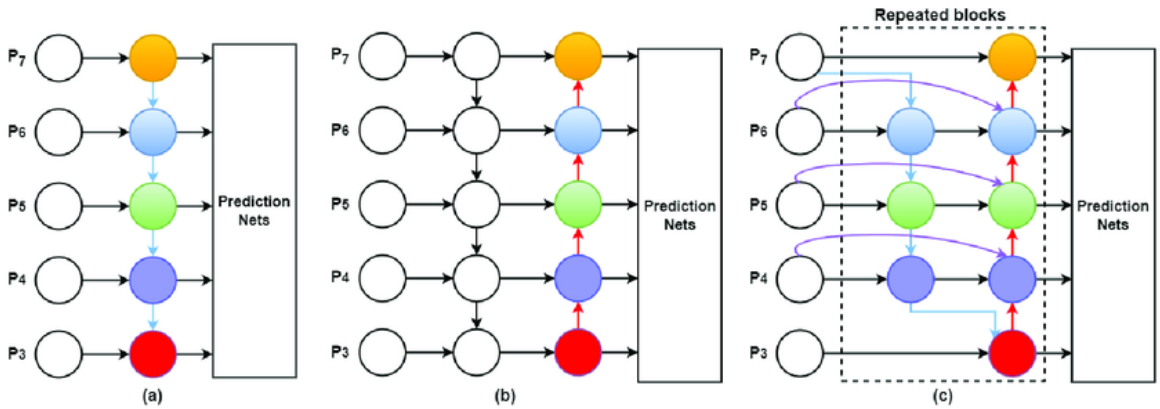


Figure 3.5: (a)-Conventional FPN; (b)-PANet; (c)-BiFPN, obtained from [24]

In addition to the bidirectional cross-scale connections, the EfficientDet model takes into consideration the importance of the different features at different resolu-

tions that can impact the predictions. Therefore, it implements a weighted feature fusion, or more precisely, a fast normalized fusion, that does not affect the performance of the GPU while also maintaining training stability [8]. Thus, the EfficientDet model combines all these concepts into a singular network, using ImageNet pretrained weights for the EfficientNet model, the BiFPN levels, that take the features from levels 3 to 7 from the backbone and applies feature fusion. In the end, these features are sent to a classification head and a regression head to both classify the objects detected and provide bounding boxes for them. Below is shown the EfficientDet architecture (see Figure 3.6).

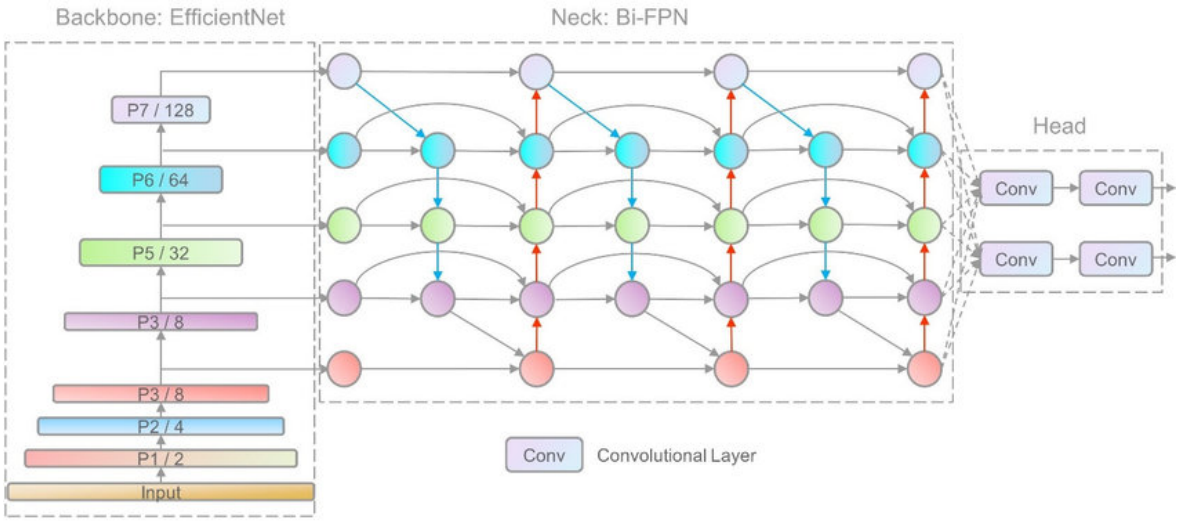


Figure 3.6: EfficientDet architecture, obtained from [25]

Chapter 4

Experimental results obtained

The figure 4.1 bellow shows the pipeline of the system and how the image flows from the classification model to the detection model.

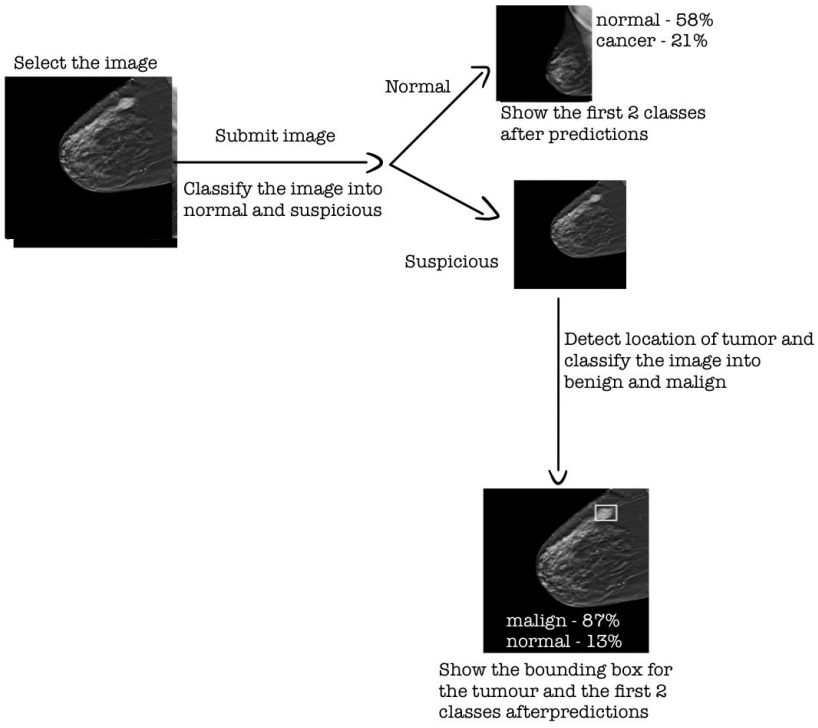


Figure 4.1: System flow

4.1 Breast Cancer Screening

The purpose of this thesis is to test the accuracy and precision resulting from training a GoogLeNet model on the Breast-Cancer-Screening-DBT database [26]. From this database, only a part of the data has been selected for training and validation. Thus, the database I will be working on contains 719 screenings, of which 350 don't present any cancer, 280 have actionable skin neoplasms, 42 have benign tumors and

47 show signs of malignant cancerous tumors. Figure 4.2 shows the distribution of the classes.

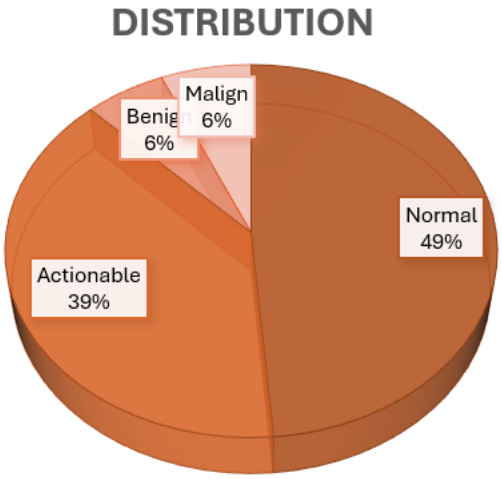


Figure 4.2: Distribution of the classes of the Breast Cancer Screening DBT database among the 719 inputs

The Cancer Imaging Archive site provides a download link to a .tcia file that contains the screening files, which can be further downloaded using the NBIA Data Retriever. The images come in the form of dcm files, which are DICOM files known in the medical forum, the abbreviation coming from Digital Imaging and Communications in Medicine.

This format is different from other image formats because the information is grouped into data sets. For the sake of the patient's confidentiality, all sensitive information regarding the patient is removed before making the DICOM file available to the public [11].

Also from the Cancer Imaging Archive website, files containing file paths, ground truth labels and bounding boxes can be accessed [6]. For reading the images from the dcm files, a github repository has been provided by the publishers of the database [27].

4.2 Data processing

The process of defining and training both the classification model and the detection model started with gathering the data and learning how to work with dcm files because this was the first time I had an encounter with them. I quickly discovered that reading the images from the dcm files took a lot of time, which meant that training the models would also take a lot of time. These files contained the pixel data of about 50 slices, some had more and some had less, each representing images

of width 2457 and height 1890, or 1996. I searched for ways to reduce the time for reading the files, at first by saving the slices as numpy arrays in file format, but that proved to not be ideal because it occupied a lot of disk storage. The dcm files themselves were not light either, totaling 120 GB of memory on my laptop, but the numpy arrays would have been impossible to store. Then, I arrived at the solution that, in time, proved to be the best: to save each individual slice as a png file. This meant that the time for reading the images was much faster, reading from dcm compared to png, and also that disk space was better utilized.

In [8] it is said that the input images for the object detection model should have a size divisible by 128, which I was not aware of when training the classification model, for which the images were not resized at all. For the images of size 2457×1996=4,904,172 I clipped the images to still have a height of 1890. I was training images of sizes 2457×1890=4,643,730 on the GoogleNet model, but only taking a maximum of 5 slices from each image. After processing the dcm files, I found out that the lowest number of slices for an image in my entire subsection of the database was 22 therefore, for the purpose of properly balancing the database, I took into consideration only 22 slices of each image.

For the images that had cancer, the ones that I had a file of bounding boxes on, I took the slice that had the tumor, 10 slices before that, and 11 after.

4.3 Classification experiments

I performed many experiments on the GoogleNet model for classification, and so I was able to understand how to work with 3D images. I first thought I had to stack them, so I sent the 5 slices that I selected from each image as 5 channels for the same input. That meant modifying the first convolutional layer of the GoogleNet architecture, which originally accepted inputs of 3×W×H and now, in the current implementation, accepts inputs of 5×W×H. In the end I came to the conclusion that a 2D approach would be better and so I returned to the original configuration for the GoogLeNet architecture; in order to have inputs of 3×W×H, considering the fact that the images were grayscale, I copied the gray layer onto two more layers.

4.3.1 Experiment 1

The first experiment was made with a learning rate of 0.001, no learning rate decay, the optimizer was SGD, and the loss for every experiment, including this one, was cross-entropy. However, for the first few runs, I considered the problem of classification as a 4 label distinction, for which I attributed a different weight to the 4 labels: 1 for both normal and actionable, and 4 for benign and malign. These weights were

used in initializing the Cross Entropy Loss Function this way:

$$torch.nn.CrossEntropyLoss(torch.tensor((1, 1, 4, 4)))$$

The batch size was 10, and the images were not resized, so I was training images of width 2457 and height 1890. The batches were also not properly distributed, meaning that one single batch could have images of only one type. Each input consisted of 5 slices stacked on top of each other, representing 5 channels. These 5 slices were chosen so that the middle slice from the whole 3D scan was the third channel, two more before and two more after. In order to better track the differences between each experiment, table 4.1 shows the distribution between the train and test classes.

	Train set	Test set
Size	568	151
Normal	273	77
Actionable	221	59
Benign	32	10
Cancer	42	5

Table 4.1: Experiment 1 — data distribution

In order to track and compare each experiment, I used Weights & Biases, also known as wanb, and logged the loss for each step, meaning that after each batch, the loss was calculated and added to the overall graph. Therefore, the image from Figure 4.3 shows the loss graph for the experiment described above.

Evidently, the loss is not ideal. It does not appear to reduce its value after each step. This image from Figure 4.3) consists of the loss of the model calculated for 6 epochs, and the predictions made on the test data was not ideal either, because it seemed to only predict normal or actionable types, disregarding the latter ones entirely. Figure 4.4 shows us how the confusion matrix looked after the first epochs, and after the sixth one.

4.3.2 Experiment 2

The second experiment proved to be a little more fruitful. I changed the learning rate to 0.0001, the optimizer to Adam and employed learning rate decay, more exactly a step learning rate decay with a step size of 4 and a gamma of 0.1. The batch size was changed to 8 and I utilized a batch sampler class to correctly distribute the different input types into the batches in order to have a better proportion of classes. I was also under the impression that the GoogleNet architecture was maybe too deep to predict the images, so I experimented with the two auxiliary modules that

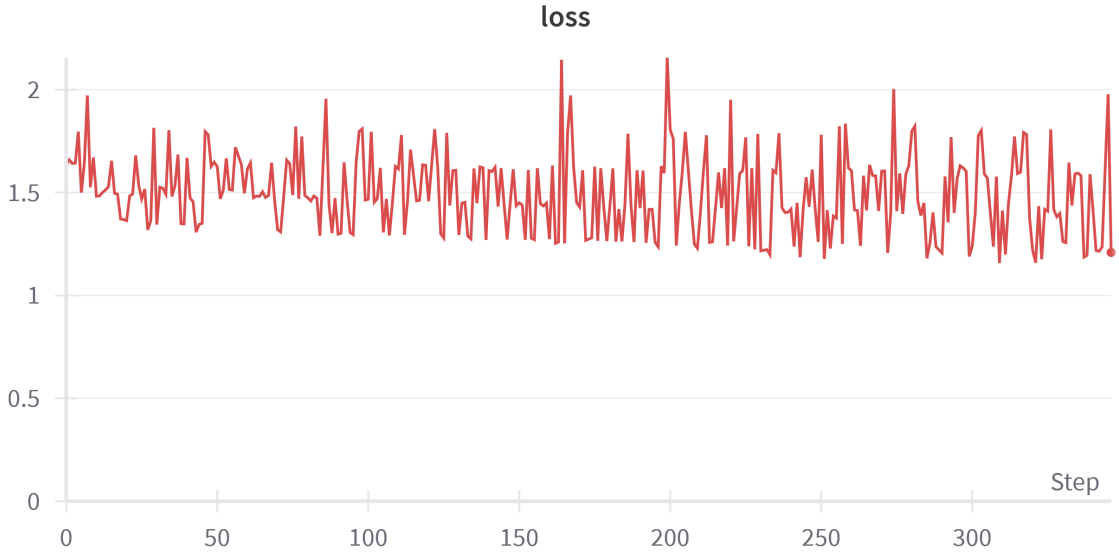
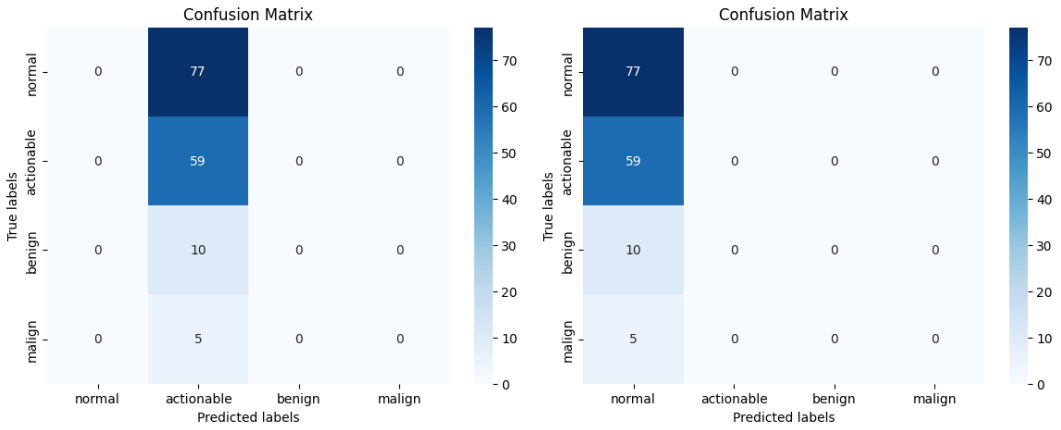


Figure 4.3: Loss Graph on train set using original GoogLeNet with a 5 channel input for the first layer



(a) After epoch 1

(b) After epoch 6

Figure 4.4: Confusion matrices for the test set ran on an original GoogLeNet architecture with a 5 channel input for the first layer

GoogleNet provides. Therefore, the model for this experiment is the same one as the GoogLeNet architecture, but moving the first auxiliary model higher, as shown in figure 4.5. For this experiment, I trained the model using the first auxiliary while also modifying the forward-pass function to only include two inception layers.

Additionally, I changed the weights for the benign and malign classes in the loss function, substituting 4 for 6 for both of them. Figure 4.6 shows how the loss graph looked after training eight epochs and another 21 batches in the ninth epoch.

This time around, the model started to also predict images of class malign, but

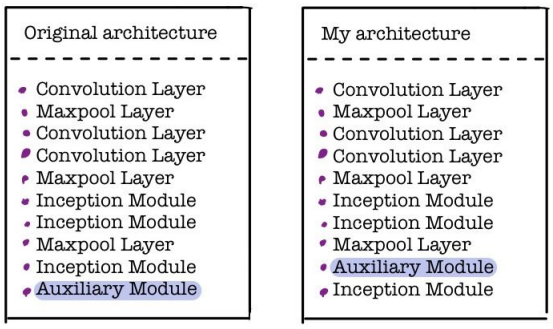


Figure 4.5: Difference between the architecture of the original GoogLeNet and the configuration for the second experiment

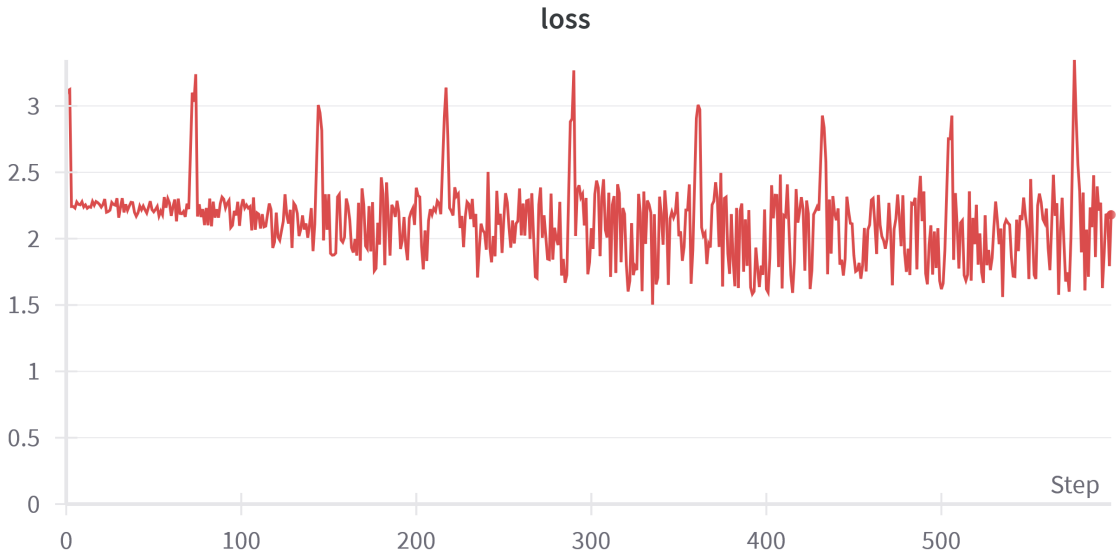


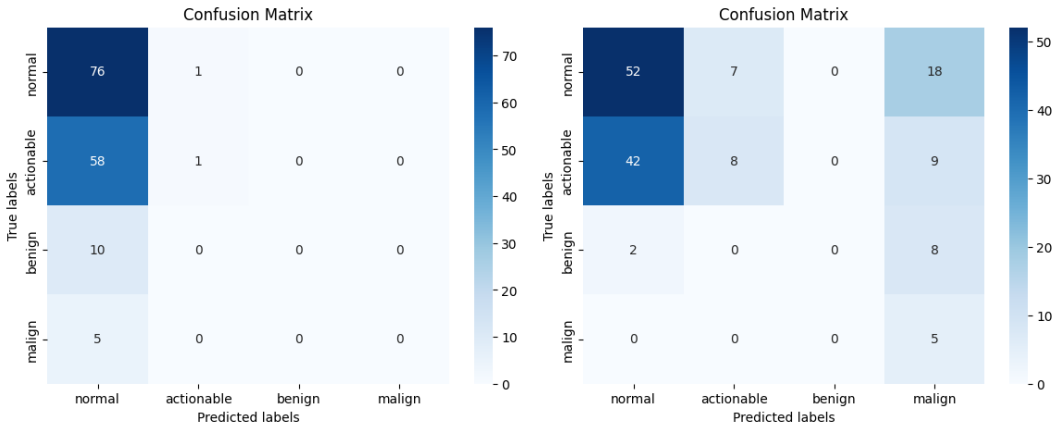
Figure 4.6: Loss Graph on train set using the first auxiliary module from the modified GoogLeNet architecture with a 5 channel input for the first layer

only after the first epoch. Curiously, it did not predict any images to be of type benign, even after eight epochs. Figure 4.7 shows how the predictions looked after the first and eighth epochs.

4.3.3 Experiment 3

The only difference between this second experiment and the third was that the learning rate decay was changed from step to exponential with a gamma of 0.8. This means that the configuration for the forward-pass method is also the same, as shown in figure 4.5 .In Figure 4.8 is shown on the same graph the difference in loss between these two latter experiments.

The predictions for the third experiment were very similar to the ones from the



(a) After epoch 1 (b) After epoch 2

Figure 4.7: Confusion matrices for test set ran on the first auxiliary module from the modified GoogLeNet architecture with a 5 channel input for the first layer

Figure 4.8: Loss Graph on train set using the first auxiliary module from the modified GoogLeNet architecture with a 5 channel input for the first layer

second experiment, meaning the model would predict only three classes out of all four of them.

The spikes in Figure 4.6 represent the batches that had more input of classes benign and malign because the number of classes did not allow for equally proportionate batches of eight. I thought that the randomness in which these batches appeared could negatively impact the overall accuracy of the model, so in further experiments, I changed this way of thinking and made these batches appear first. Figure 4.8 shows how the experiment turned out, so not much was changed, and

the predictions overall on the test data were not different.

4.3.4 Experiment 4

For the next experiment, I stopped considering this a problem of 4-label detection but rather a 2-label detection. Therefore, I grouped the actionable, benign and malign types into one and trained the model using this group and the normal ones. I kept everything else in terms of configuration, including the structure of the architecture, from the third to the fourth experiment, except for batch size, which I changed to 64. Figure 4.9 shows how the loss looked after training three epochs on the training data, and Figure 4.10 shows how the confusion matrix looked after those three epochs on the testing data. Table 4.2 shows the distribution between the classes in the train set and test set.

	Train set	Test set
Size	568	151
Normal	273	77
Cancer	295	74

Table 4.2: Experiment 4 — data distribution

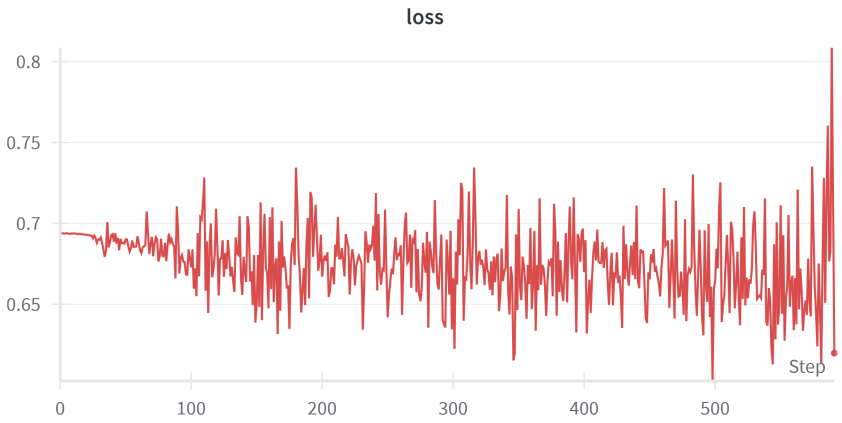


Figure 4.9: Loss Graph on train set using the first auxiliary module from the modified GoogLeNet architecture with a 5 channel input for the first layer

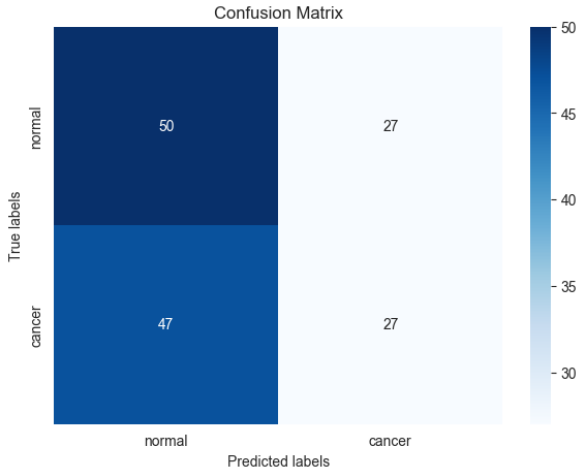


Figure 4.10: Confusion matrices for test set ran for three epochs on the first auxiliary module from the modified GoogLeNet architecture with a 5 channel input for the first layer

4.3.5 Experiment 5

I resized the images to be 300×300 and instead of adding the slices as channels of the same image, I added the slices as individual inputs. In doing so, I transformed a 568-length dataset into a 2,840-length one, and used a batch size of 64.

Upon figuring out the number of learnable parameters, which was around 3 milions, I wanted to increase the length of the dataset even more. With this purpose in mind I considered 22 slices of each input, and adding each of them individually resulted in a 12,496-length dataset. This also meant that I stopped providing the model with 3D images, instead adding each input as an image of size 3×W×H.

	Train set	Test set
Size	12496	3322
Normal	6006	1694
Cancer	6490	1628

Table 4.3: Experiment 5 distribution

Additionaly, I wanted to compare the loss of the two auxiliary models that GoogLeNet has to offer with the main output that it returns. In doing so, I returned to the initial implementation for the forward-pass method. However, I made a mistake in that I forgot that the original implementation doesn't include either activation function on the last layer, instead adding one only for the main output, not for the two auxiliary modules. Curiously, these two modules proved to obtain the lowest loss of all the experiments I performed, as shown in figure 4.11.

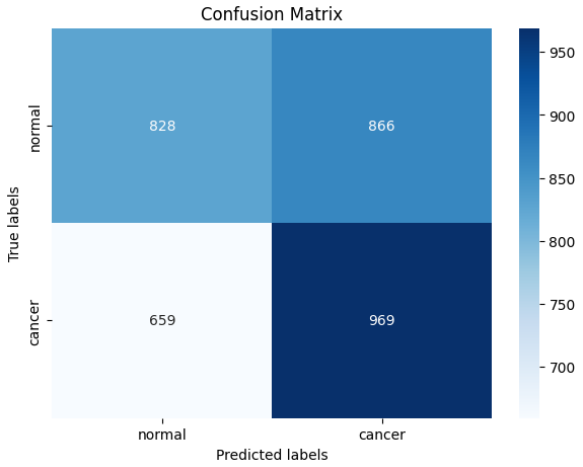


Figure 4.12: Confusion matrices for test set ran for 5 epochs on the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer

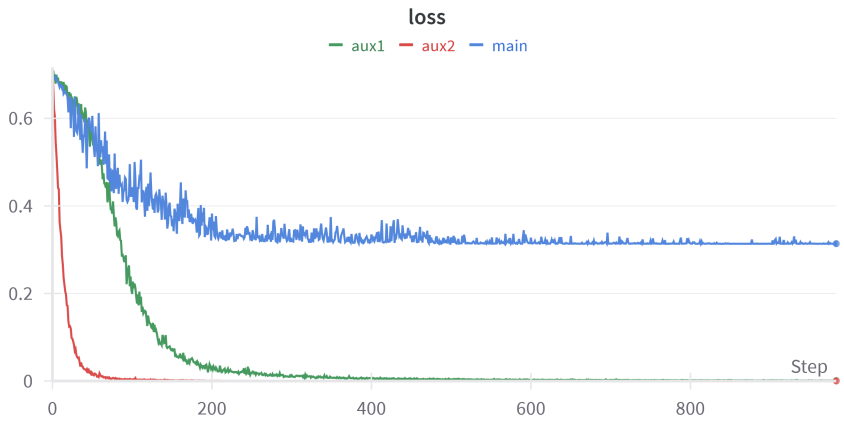


Figure 4.11: Loss Graph on train set using both the auxiliary modules and the main module from the original GoogLeNet architecture with a 3 channel input for the first layer

Even with all these experiments and improvements, the best accuracy I managed to obtain was 0.5409, obtained using the main module from the fifth experiment configuration, after five epochs, shown in figure 4.12.

4.3.6 Experiment 6

Towards the end of the process of training the classification model, I decided to use already GoogLeNet trained weights from the Torchvision library. The nature of this pre-trained weights does not allow direct access to the auxiliary modules, though the forward-pass method could have been overwritten. I however chose not to do this, instead opting to only examine the output from the main module.

These weights had been trained on images from ImageNet. Therefore, the experiments to come represent a fine-tunning of the pre-trained model aquired. The model was trained on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) contest in 2014 [15], which contains 456567 images for training [28]. I also resized the images again and made them 512x512 in order to also fit the detection model. Figure 4.13 shows the loss graph after training the model for 5 epochs, while figure 4.14 shows the confusion matrix at the end of those epochs.

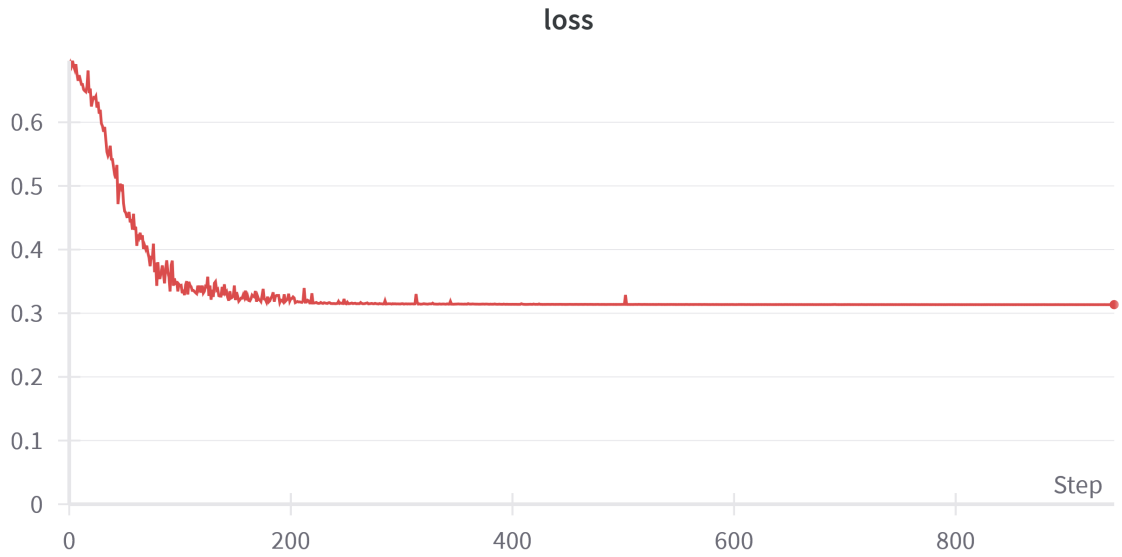


Figure 4.13: Loss Graph on train set using the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer

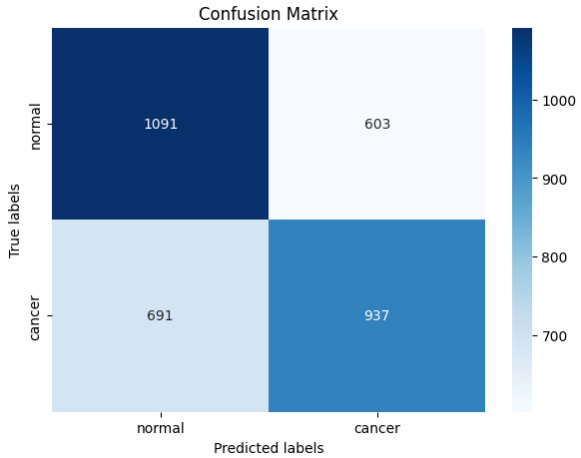


Figure 4.14: Confusion matrices for test set ran for 5 epochs on the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer

I also experimented with different batch sizes for the pre-trained model on ImageNet weights.

Therefore, I trained the model with the same configuration as the previous run, except I modified the size of the batches from 64 to 128. Figure 4.15 shows the comparison in loss between these two experiments. This second model I also trained for 5 epochs; however, the difference in steps in the graph comes from increasing the number of inputs in each batch, which means fewer batches over the dataset and overall fewer steps.

Figure 4.15: Comparison between the losses obtained on train set using the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer, with different sized batches

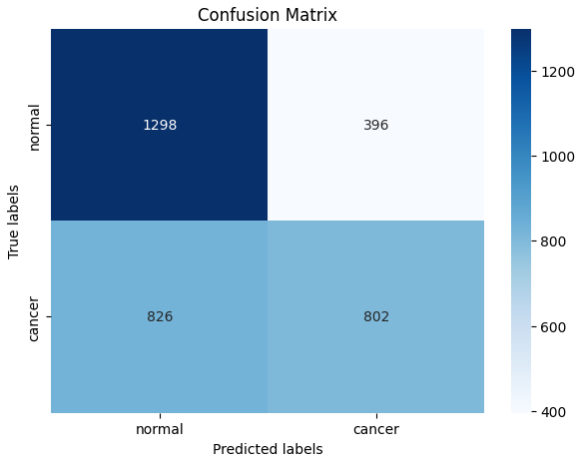


Figure 4.17: Confusion matrices for test set ran for 3 epochs on the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer

The final configuration that I came to included: batches of size 128, pre-trained

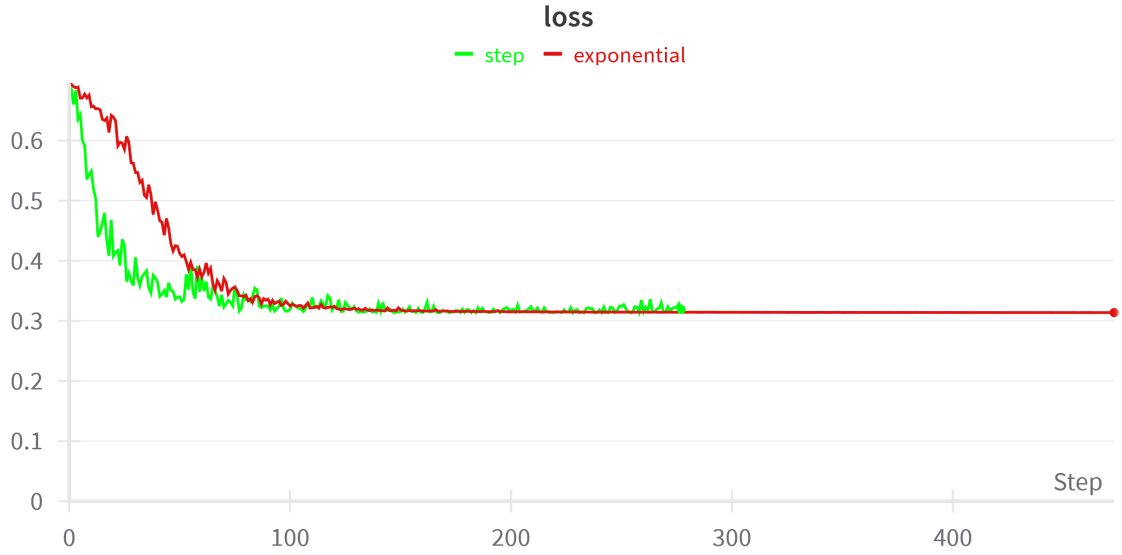


Figure 4.16: Comparison between the losses obtained on train set using the main module from the pre-trained GoogLeNet architecture with a 3 channel input for the first layer, with different types of learning rate decay

weights trained with images from ImageNet, a learning rate of 0.0001, learning rate decay using the StepLR implementation with a step size of 5 and gamma of 0.1, an image size of 512, and 22 inputs of each image representing different slices from the 3D scans. The loss obtained using a learning rate of Step instead of Exponential is shown in figure 4.16. Figure 4.17 shows how the confusion matrix looked after 3 epochs using this configuration.

Tables 4.4, 4.6 and 4.5 shows the conclusions of these experiments.

Experiment	#classes	Image size	Model	trained/pre-trained
Exp1	4	5×2457×1890	GoogLeNet	trained
Exp2	4	5×2457×1890	aux1	trained
Exp3	4	5×2457×1890	aux1	trained
Exp4	2	5×300×300	aux1	trained
Exp5	2	3×300×300	GoogLeNet	trained
Exp6	2	3×512×512	GoogLeNet	pre-trained

Table 4.4: Classification models

Experiment	Performance					
	Accuracy	Recall		Precision		
Exp1	0.5099	normal	1	normal	0.51	
		actionable	0	actionable	0	
		benign	0	benign	0	
		malign	0	malign	0	
Exp2	0.4304	normal	0.675	normal	0.54	
		actionable	0.136	actionable	0.53	
		benign	0	benign	0	
		malign	1	malign	0.125	
Exp3	0.4304	normal	0.675	normal	0.54	
		actionable	0.136	actionable	0.53	
		benign	0	benign	0	
		malign	1	malign	0.125	
Exp4	0.5099	normal	0.649	normal	0.515	
		cancer	0.365	cancer	0.50	
Exp5	0.5409	normal	0.489	normal	0.557	
		cancer	0.596	cancer	0.528	
Exp6	0.6322	normal	0.7659	normal	0.6106	
		cancer	0.4922	cancer	0.6694	

Table 4.5: Classification models(continuation)

Experiment	Hyper parameters				
	learning rate	optimizer	learning rate decay	step	gamma
Exp1	0.001	SGD	no		
Exp2	0.0001	Adam	Step	4	0.1
Exp3	0.0001	Adam	Exponential		0.8
Exp4	0.0001	Adam	Exponential		0.8
Exp5	0.0001	Adam	Exponential		0.8
Exp6	0.0001	Adam	Step	5	0.1

Table 4.6: Classification models(continuation)

4.4 Detection experiments

After classifying the images into those with cancer and without, I wanted to see I could use a model to detect exactly where the tumor is thought to be. In doing so, I trained the model only on the images containing tumors.

The database selected is the same one as the one for the classification problem, the WDBC dataset, from which I selected only the ones representing benign and malign images. Table 4.7 shows the distribution and size of the train and test data for the detection problem.

	Train set	Test set
Benign	704	264
Malign	1012	110

Table 4.7: Detection experiments distribution

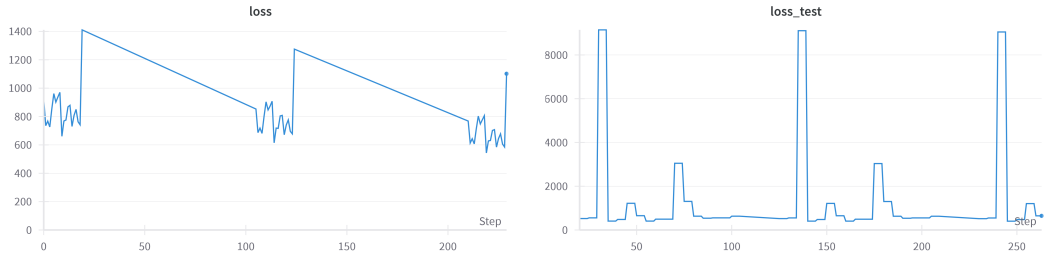
The images were resize to be 512×512, meaning 262,144 features from each image, 22 slices from each 3D scan. Considering the fact that the images were resized in order to be 512×512, I scaled the coordinates of the bounding boxes accordingly. Each image contains a single input, and the coordinates for the bounding box that result from the validation step are ran through a weighted box fusion in order to display one single box. The size of the batches was changed to 16. This configuration remains the same throughout all the detection experiements. Along with these preset conditions, the optimizer was also chosen to be Adam throughout all the experiments.

4.4.1 Experiment 1

In the first implementation of this run, I opted to not use pre-trained weights in order to compare the proper results. With that purpose in mind, the first configuration included a learning rate of 0.0001 and a learning rate decay of ReduceLROnPlateau, with the minimum value for the learning rate being $1e^{-8}$. However, this learning rate decay is only utilized after the test loop has finished running, and the metric used is the loss obtained for the test dataset. Also, the configuration for the EfficientDet is the first one, D-0, as it has the same image size as the size for the inputs used by me. The attributes associated with the D-0 type EfficientDet were obtained by calling the proper function from the library Effdet.

The loss of both the detection problem and classification are calculated directly after running the forward-pass method in the main EfficientDet class, so in order to see exactly how the loss is calculated, I searched for the official implementation of the library on GitHub [29]. This method takes as input the class outputs from the model, the box outputs, the ground truth for the class and box and the positive numbers from the ground truth anchors; this method returns three parameters, the class loss, calculated as the focal loss with the formula $-(1-pt)^{gamma} * log(pt), pt = probability\ of\ being\ classified\ to\ the\ true\ class$, the box regression loss and the total loss, computed as the sum between those two.

Figure 4.18 shows the loss for the train dataset and the test dataset. The big jumps from the steps in both graphs indicate that the missing steps in the train loss graph are in the test loss graph and vice versa.



(a) Train loss (b) Test loss

Figure 4.18: Loss graphs for the EfficientDet D-0 model trained on train and then test sets

4.4.2 Experiment 2

The figure does not indicate any loss reduction, so I changed the configuration of the detector. I chose not to normalize the images, each of size 3×512×512 and test how the model behaves. I also discovered that EfficientDet works better when the bounding box coordinates are given in (y,x,y,x) order, and so I changed it from (x, y, width, height). Additionally, I used pre-trained weights, which I obtained from Alex Shonenkov's Kaggle account [30] and I reset both the classification head and the regression head in order to fit my classification problem.

The hyper-parameters are the same as the previous experiment. The model was trained for one epoch; I stopped the run before it had a chance to validate the model, so there is no graph for the test loss. Figure 4.19 shows the train loss.

I took the same configuration and trained it for 50 epochs to see how the loss evolved. Instead of running the train loop once for 50 epochs, I ran the loop five times with 10 epochs each. Because of this, the graph is not continuous; each batch of 10 epochs starts at the 0 step, but each batch after the first one starts with the weights trained from the previous batches.

In the first 10 epochs, shown in figure 4.20, the loss decrease is steep, and so is the test loss. However, after 40 epochs, the decline gets more gradual, as shown in figure 4.21.

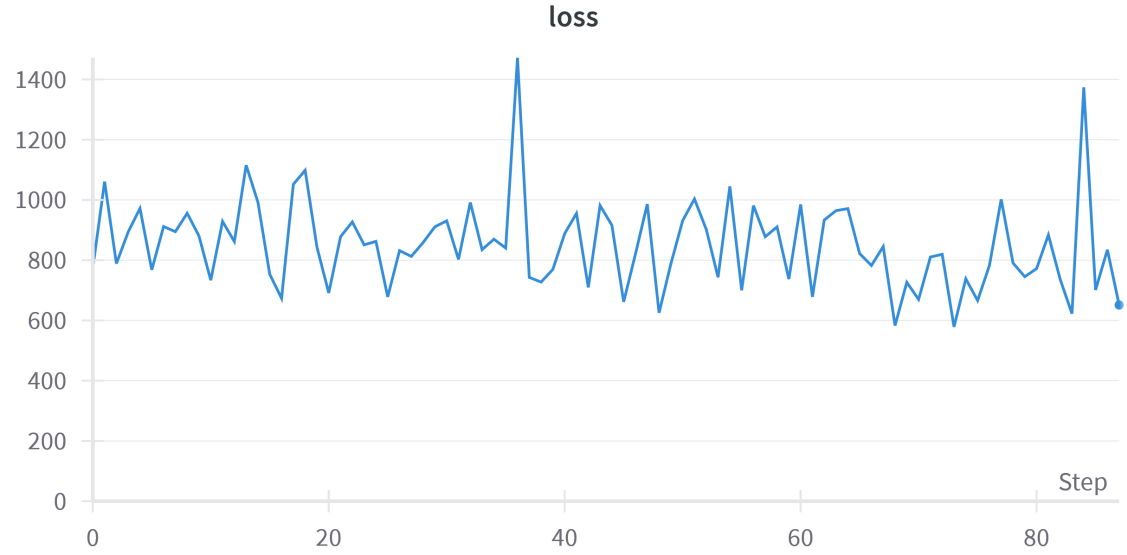
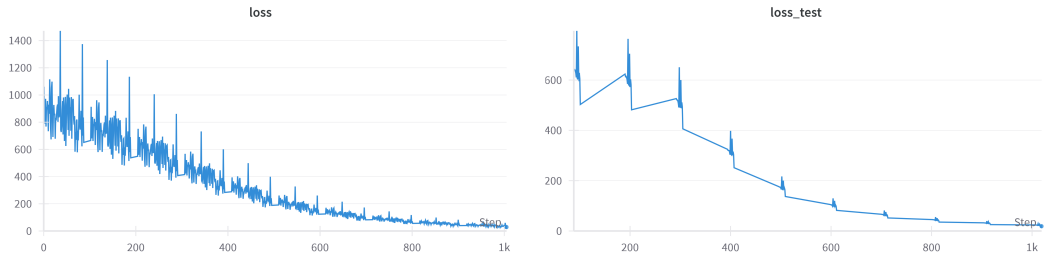
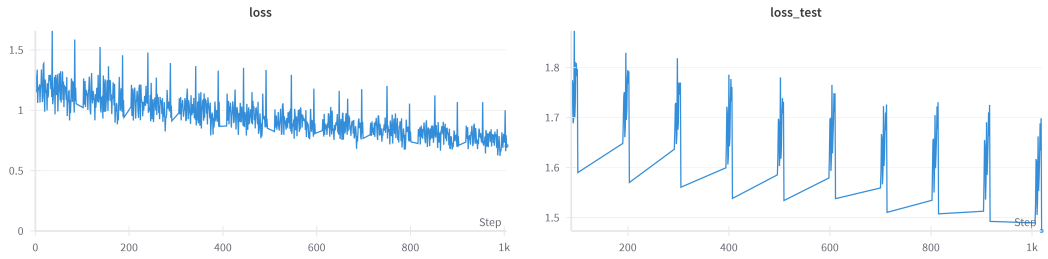


Figure 4.19: Loss Graph on train set using the original EfficientDet with pre-trained weights from Kaggle code



(a) Train loss (b) Test loss

Figure 4.20: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from Kaggle code trained on train and then test sets after 10 epochs



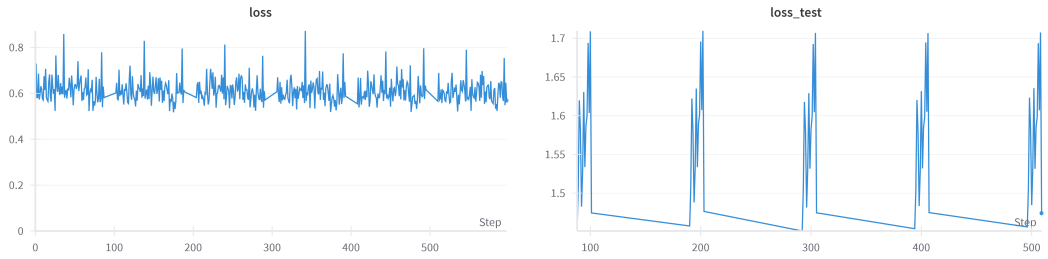
(a) Train loss (b) Test loss

Figure 4.21: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from Kaggle code trained on train and then test sets after 40 epochs

4.4.3 Experiment 3

For the next experiment, I normalized the images again to see if I could get a starting loss lower than the previous ones. I also found another file containing pre-trained weights, this time from the official Effdet library on GitHub [29]. With this configuration, I trained all the layers 13 epochs. The hyper-parameters remained the same as the previous experiments.

Figure 4.4.3 shows how the train and test loss looked after those 13 epochs. While the initial loss was much lower than the previous ones, I was not satisfied with the test loss, so I continued to experiment.



(a) Train loss (b) Test loss

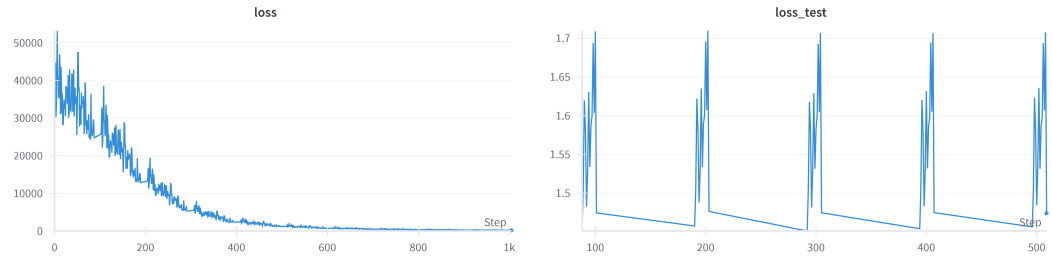
Figure 4.22: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from official GitHub account without anchors trained on train and then test sets

4.4.4 Experiment 4

In the next experiment, I changed the learning rate decay from ReduceLROnPlateau to CosineAnnealingLR, with a minimum learning rate of $1e^{-6}$. I also modified the way the decay is applied, calling it after the each step in the train loop and using the loss as a metric rather than the test. Additionally, I modified the way the model is initialized; instead of manually downloading the file and loading a Pytorh script, I used the create_model_from_config method from the Effdet library, calling it this way:

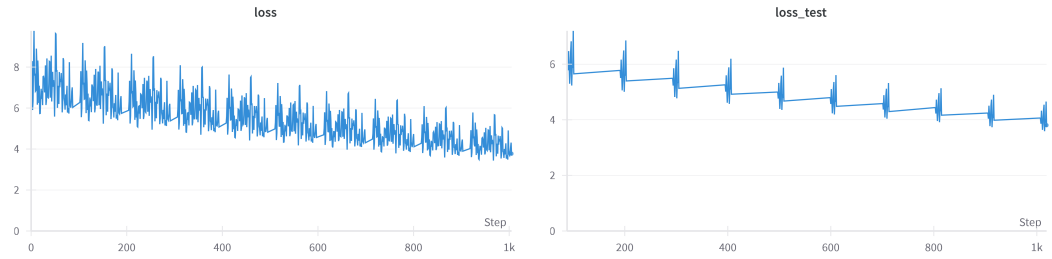
$$create\_model\_from\_config(bench\_task =' train', num\_classes = 2, pretrained = True)$$

I also trained this model for 50 epochs, in the same way as experiment 3. Figure 4.4.4 shows how the train and test looked for the first 10 epochs, and figure 4.4.4 shows how the train and test looked for the 40 to 50 epochs.



(a) Train loss (b) Test loss

Figure 4.23: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from official GitHub account without anchors trained on train and then test sets for the first 10 epochs



(a) Train loss (b) Test loss

Figure 4.24: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from official GitHub account without anchors trained on train and then test sets for the 40 to 50 epochs

4.4.5 Experiment 5

For the next experiment I wanted to see exactly how the model would operate after I called the initialization method. I again consulted the official implementatin from GitHub [29] and discovered a key parameter, *bench_labeler* that I did not include in my call. This parameter, taking boolean values, True or False, was included in the initialization of the class DetBenchTrain, if I chose to train the model rather than validate it. This goes on to be used in an if statement in the forward-pass method, and if this value proves to not be present in the call, it provides a warning before the code: *target should contain pre-computed anchor labels if labeler not present in bench*. My pre-processing did not include any computing of anchor labels.

Anchors in the context of detection algorithms, also known as prior boxes, are bounding boxes computed before the actual training, that help the model better understand what size the objects in the images would be. For example, if the model was intended to recognize people from an image, the anchor boxes might be chosen to be taller rather than wider, as to match the aspect ratio of the shape of a

person. [31]

The basic configuration that the Effdet library has to offer does include some anchor labels, based on the overall size of the images, so I chose to use those and see how the predictions would turn out. The way the model was initialized is this: $create\_model\_from\_config(config = config, bench\_task =' train', num\_classes = 2, pretrained = True, bench\_labeler = True)$.

This proved to be a very good discovery. The hyper-parameters were the same as the previous experiment. I only ran one epoch of this configuration, so there is no test loss. Figure 4.25 shows how the train loss looked for the first epoch.

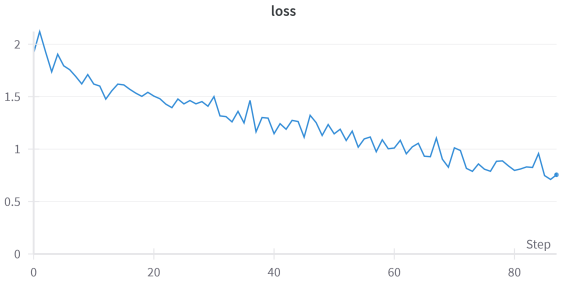
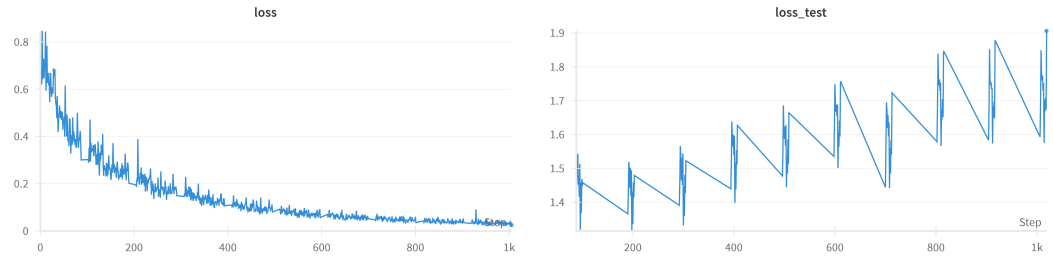


Figure 4.25: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from official GitHub account with anchors trained on the train set for one epoch

This is the first experiment where the initial loss is ideal. Keeping this in mind I kept almost the same configuration for the next experiment, besides training the model for more epochs.

4.4.6 Experiment 6

The learning rate decay in experiment 5 was used after each step, meaning each batch, in the train dataset, and also at the end of all of the batches. In this experiment, I stopped calling the step function after all the batches and kept the one after each batch. Figure 4.4.6 shows how the train and test losses looked for the 10 epochs that the model has been trained for.



(a) Train loss

(b) Test loss

Figure 4.26: Loss graphs for the original EfficientDet D-0 model with pre-trained weights from official GitHub account with anchors trained on train and then test sets for 10 epochs

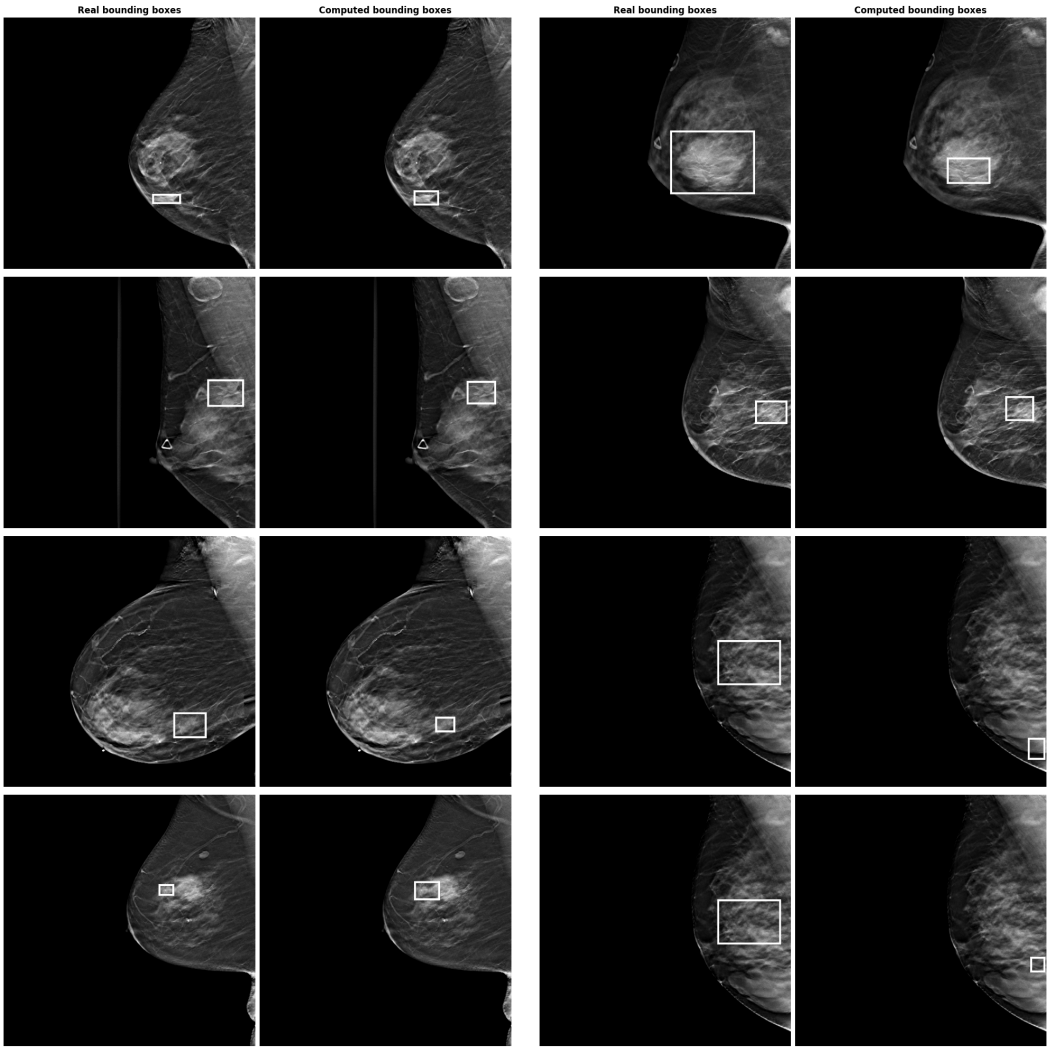
The image 4.4.6 shows real bounding boxes drawn on the picture they belong to versus predicted bounding boxes drawn on the same pictures.

As it appears the test loss increases each epoch, I chose to integrate into the system the model after 2 epochs. Therefore, the final configuration for the detection model is: batch size of 16, image size of 3×512×512, the images were normalized, learning rate of 0.0001 with a CosineAnnealing learning rate decay applied after each batch in the training loop and an Adam optimizer.

Tables 4.8, 4.9 and 4.10 show the conclussions of these experiments

Experiment	Image size	Model	trained/pre-trained
Exp1	3×512×512	EfficientDet	trained
Exp2	3×512×512	EfficientDet without anchors	pre-trained from Kaggle
Exp3	3×512×512	EfficientDet without anchors	pre-trained from GitHub
Exp4	3×512×512	EfficientDet without anchors	pre-trained from GitHub
Exp5	3×512×512	EfficientDet with anchors	pre-trained from GitHub
Exp6	3×512×512	EfficientDet with anchors	pre-trained from GitHub

Table 4.8: Detection models



(a) Train predictions (b) Test predictions

Figure 4.27: List of predictions vs. real bounding boxes on the train and test set

Experiment	Performance			
	Test loss		Train loss	
	Maximum	Minimum	Maximum	Minimum
Exp1	9146.229	402.704	1411.082	543.197
Exp2	795.51	1.473	1471.658	0.622
Exp3	1.712	1.451	0.9926	0.5174
Exp4	33736.91	3.589	53161.637	3.437
Exp5			2.121	0.7108
Exp6	1.906	1.315	0.8469	0.02313

Table 4.9: Detection models(continuation)

Experiment	Hyper parameters			
	learning rate	learning rate decay	minimum lr	step method called
Exp1	0.0001	ReduceLROnPlateau	$1e^{-8}$	after test loop
Exp2	0.0001	ReduceLROnPlateau	$1e^{-8}$	after test loop
Exp3	0.0001	ReduceLROnPlateau	$1e^{-8}$	after test loop
Exp4	0.0001	CosineAnnealingLR	$1e^{-6}$	after each batch after train loop
Exp5	0.0001	CosineAnnealingLR	$1e^{-6}$	after each batch after train loop
Exp6	0.0001	CosineAnnealingLR	$1e^{-6}$	after each batch

Table 4.10: Detection models(continuation)

Chapter 5

MammoDetect System

5.1 Functionalities

The system consists of two main functionalities: selecting an image, either from the array of examples provided in the top right corner of the interface, or the upload button which opens the computer's internal storage. Upon selecting an image, it will be viewed in the main window. The image selected does not necessarily have to be 512×512, because it will be resized anyway. The user then has the choice of either selecting another image or submitting the image to the model to be classified into normal and cancerous, the latter being then sent to the detection model to replace the image in the main window with the same image that has drawn the bounding box resulted from passing the image to the model.

The detection model also classifies the tumors into benign or malign. In the bottom right corner, below the examples image array, there 2 boxes. The top one contains the top two classes the models classified the image as while the bottom one displays the true label of the image, if it was selected from the examples array. If the image has not been selected from the examples array, the bottom box will contain the word "unknown".

Because the way this library sends the data from the image selected doesn't include the actual name of the filename, I found myself thinking of ways to find and display the true label of the images selected from the examples. To do so, I compared the contents of each of the images existing in the directory to the contents of the image selected. In preparation for this, for each image saved as index.png, index being a variable from 1 to 151, I saved an additional file named index-label.txt, where label is either 0 or 1, corresponding to the labels normal or cancer. For the images uploaded from the user's device, there is no way to find the true label of the input, therefore, I added another class, unknown, that is displayed whenever the label of the image cannot be determined.

Comparatively, if the image is detected to be normal, the main window will not change, instead displaying the original image.

5.2 Design

Figure 5.1 shows the architecture diagram. Figure 5.2 shows the class diagram. The Fitter class is used to train the detection model, the classification model is trained in a simple loop. Figure 5.3 shows the sequence diagram.

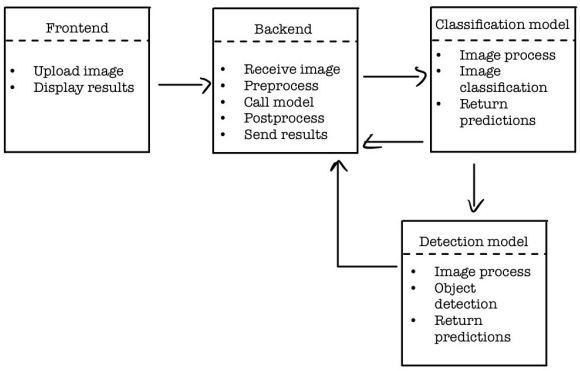


Figure 5.1: System architecture diagram

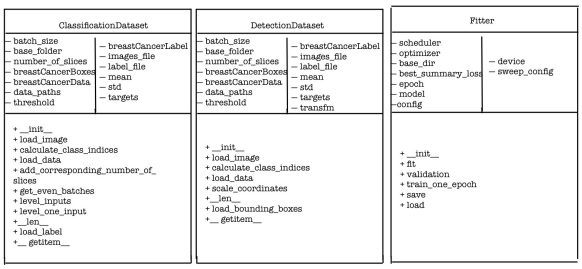


Figure 5.2: System class diagram

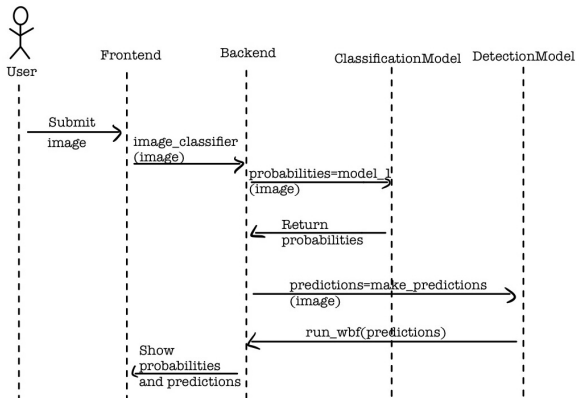


Figure 5.3: System sequence diagram

5.3 Implementation

Both the train and front-end section of the system were implemented in Python. For the front-end, I used the library Gradio. First, I used the gradio Interface method of creating a GUI, however I did not like that the examples array was out of view when uploading an image of 512×512. This can be seen in image 5.4.

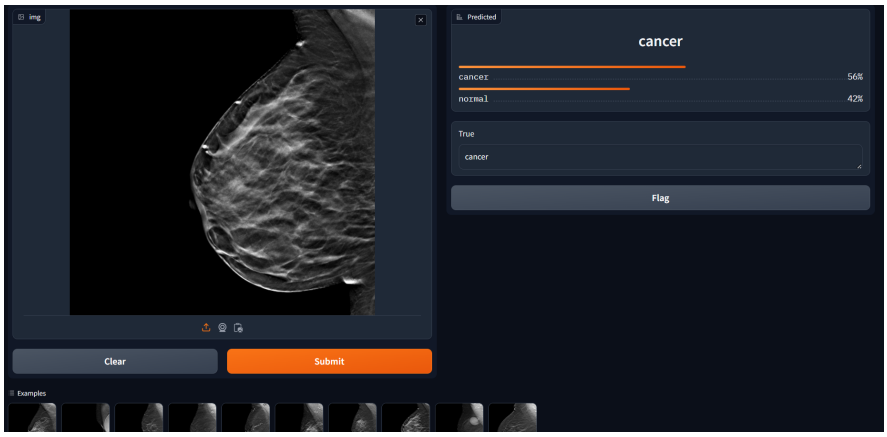
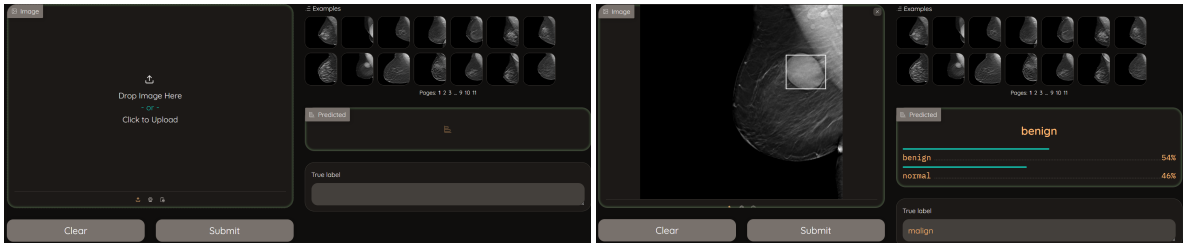


Figure 5.4: Application interface

Then, I experimented with the gradio Blocks method and I also chose a theme to go with the system. Figure 5.5 show how the interface looked with no image selected and how it looks like after submitting.



(a) No image selected

(b) Image selected and submitted

Figure 5.5: GUI

The models' architecture used was the one in the Torch library. I implemented my one datasets for both the classification and detection problems, that inherit methods from $torch.utlis.data.Dataset$. The order of the inputs so that each batch is balanced is determined directly in the dataset classes, so the $torch.utils.data.DataLoader$ does not take any sampler classes as input, instead setting the $shuffle$ attribute to False. The batch size is sent to both the DataLoader and Dataset classes.

5.4 Testing and validation

In order to make sure that the system behavied accordingly, I tested the function *find_image_path* from the directory *utils*, used in the $\_\_getitem()\_\_$ function in both datasets. This method is supposed to remove the name of the file at the end of a long string representing the path to the dcm files and adds the string "NA-" in the last directory name after the "-". The reason for this method existing is that the png images are saved in directories with the name exactly as the dcm files. Also, the name of the files contain the string mentioned before, this might be because I downloaded the dcm files before the latest update of the dataset, which was in January 2024.

For testing this method I used some simply assertions to make sure that the function is fulfilling its intended use.

Chapter 6

Conclusions and possible improvements

In the end, GoogLeNet proved to not be a worthy architecture for the classification of cancer tumors in digital breast tomosynthesis images because of the low accuracy obtained. The highest accuracy obtained was 0.6321 and precision and recall for predicting cancer images were 0.6027 and 0.6087, respectively. I propose a different architecture, maybe DenseNet or ResNet to train and compare the results obtained by GoogLeNet. Additionally, I would increase the length of the dataset even more by including all the slices, keeping in mind the risk that some inputs might overpower others, especially considering the fact that the lowest number of slices an image has in the database is 22, and the highest number is 120.

As for EfficientDet, I am quite happy with the results of the trained model. I would maybe experiment with other types of EfficientDet, from D1 to D5. I do have some concerns, mainly the fact that the test loss seems to slightly increase from epoch to epoch while the train loss decreases. While the CosineAnnealing learning rate decay proved to work well with the model, I would experiment with other types of learning rate decay, such as OneCycle. The test loss obtained was 1.315 and the train loss was 0.2524. I would also experiment with different values for the anchor boxes.

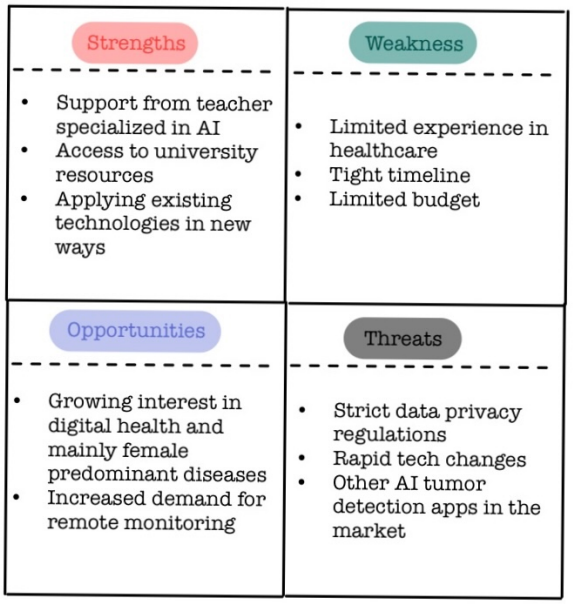


Figure 6.1: SWOT analysis

Bibliography

[1] Naji M. A., El Filali S., Aarika K., Benlahmar E. H., and O. Abdelouhahid R. A.and Debauche. Machine learning algorithms for breast cancer prediction and diagnosis. *Procedia Computer Science*, 191:487–492, 2021.

[2] Gaudenz Boesch. Googlenet explained: The inception model that won imagenet. *Deep Learning*.

[3] Jonas Cleveland. 7 best image classification models you should know in 2023, 2023. https://jonascleveland.com/best-image-classification-models/.

[4] Dhahri H., Al Maghayreh E., Mahmood A., Elkilani W., and Faisal Nagi M. *Automated breast cancer diagnosis based on machine learning algorithms*. Journal of healthcare engineering, 2019.

[5] Amrane M., Oukid S., Gagaoua I., and Ensari T. Breast cancer classification using machine learning. *2018 electric electronics, computer science, biomedical engineerings' meeting (EBBT)*, pages 1–4, 2018.

[6] Buda M., Saha A., Walsh R., Ghate S., Li N., Święcicki A., Lo J. Y., and Mazurowski M. A. A data set and deep learning algorithm for the detection of masses and architectural distortions in digital breast tomosynthesis images. *JAMA Netw Open*, 4, 2021.

[7] Islam M., Haque M., Iqbal H., Hasan M., Hasan M., and Kabir M. N. Breast cancer prediction: a comparative study using machine learning techniques. *SN Computer Science*, 1:1–14, 2020.

[8] Tan M., Pang R., and Le Q. V. Efficientdet: Scalable and efficient object detection. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 0781–10790, 2020.

[9] Tan M. and Le Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *International Conference on Machine Learning*, 2020.

[10] Zwitter Matjaz and Soklic Milan. Breast Cancer. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C51P4M.

[11] Varma D. R. Managing dicom images: Tips and tricks for the radiologist. *Indian J Radiol Imaging*, 22:4–13, 2012.

[12] Lu S., Ding Y, Liu M., Yin Z., Yin L., and Zheng W. Multiscale feature extraction and fusion of image and text in vqa. *International Journal of Computational Intelligence Systems*, 16, 2023.

[13] Nanglia S., Ahmad M., Khan F. A., and Jhanjhi N. Z. An enhanced predictive heterogeneous ensemble model for breast cancer prediction. *Biomedical Signal Processing and Control*, 72, 2022.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[16] Yasas Sri Wickramasinghe. Best object detection models in 2024. *Object Detection*, 2024. https://dagshub.com/blog/best-object-detection-models/.

[17] Wolberg William, Mangasarian Olvi, Street Nick, and Street W. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

[18] https://wiki.cancerimagingarchive.net/pages/viewpage.action?pageId=64685580#6468558050a1e1bdf0de46de92128576e1d3e9b1.

[19] https://www.cancer.org/cancer/types/breast-cancer/about/how-common-is-breast-cancer.html.

[20] https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic.

[21] https://www.geeksforgeeks.org/understanding-googlenet-model-cnn-architecture/.

[22] https://www.researchgate.net/figure/Pattern-diagram-of-feature-pyramid_fig1_369956030.

[23] https://www.researchgate.net/figure/Pattern-diagram-of-feature-pyramid-network_fig2_369956030.

[24] https://www.researchgate.net/figure/Structures-of-FPN-PANet-and-Bi-FPN-a-FPN-64-presents-a-top-down-pathway-b-PANet_fig3_365953412.

[25] https://www.mdpi.com/drones/drones-07-00682/article_deploy/html/images/drones-07-00682-g002-550.jpg.

[26] https://www.cancerimagingarchive.net/collection/breast-cancer-screening-dbt/.

[27] https://github.com/mazurowski-lab/duke-dbt-data.

[28] https://www.image-net.org/challenges/LSVRC/2014/.

[29] https://github.com/rwightman/efficientdet-pytorch/tree/master.

[30] https://www.kaggle.com/code/shonenkov/training-efficientdet/input.

[31] https://encord.com/glossary/anchor-boxes-definition/.