UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA [Secţie]

LUCRARE DE LICENŢĂ

[Titlu lucrare]

Conducător ştiinţific
[Grad, titlu si nume coordonator]

Absolvent
[Baciu Ioana]

2024

ABSTRACT

Abstract: un rezumat în limba engleză cu prezentarea, pe scurt, a conţinutului pe capitole, punând accent pe contribuţiile proprii şi originalitate

Contents

Chapter 1

Introduction

Breast cancer is known to be the most common cancer among women [11]. 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 [12]. 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 [12].
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.

Chapter 2

The scientific problem addressed

Chapter 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 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 [13]. 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.

Other relevant studies include "An enhanced Predictive heterogeneous ensemble

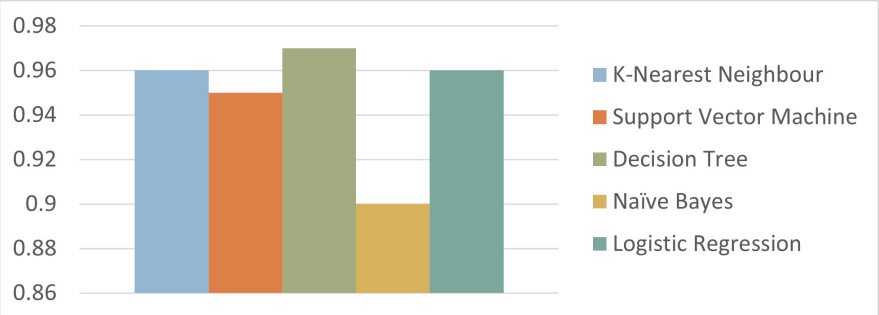


Figure 3.1: Accuracy of classification techniques

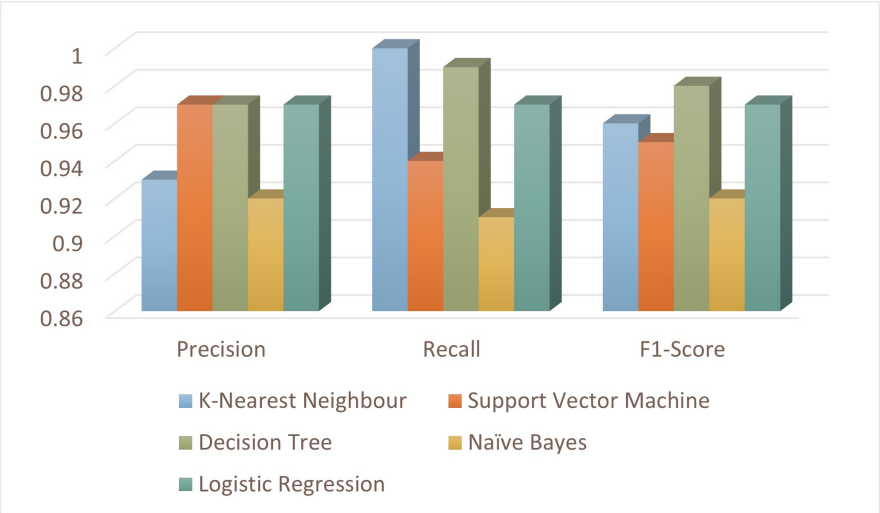


Figure 3.2: Performance of classification techniques for class benign

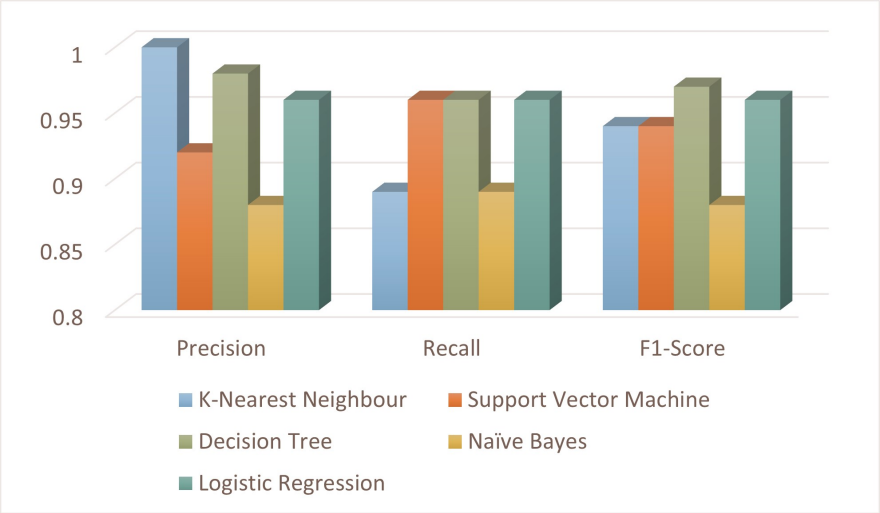


Figure 3.3: Performance of classification techniques for class malignant

model for breast cancer prediction" concluded by S. Nanglia et al. which got 78% accuracy using KNN, SVM and DT [10]. Islam et al. got a 98.75% using ANN on the WDBC [5]. Amrane et al. proposed an approach using KNN and NB with 97.51% accuracy [3]. Dhahri et al. studied the usage of genetic programming techniques for the selection of the best features and parameters for the machine learning classifier [2].

Chapter 4

Methods used in detecting breast cancer

4.1 Database used

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 [14]. 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.
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. [8].
Also from the Cancer Imaging Archive website, files containing file paths, ground truth labels and bounding boxes can be accessed. [4]. For reading the images from the dcm files, a github repository has been provided by the publishers of the database. [15]

4.2 GoogLeNet Architecture

4.3 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. [6]

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. [9]. 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 4.1

The problem with this discovery was that the cost of system storage for creating

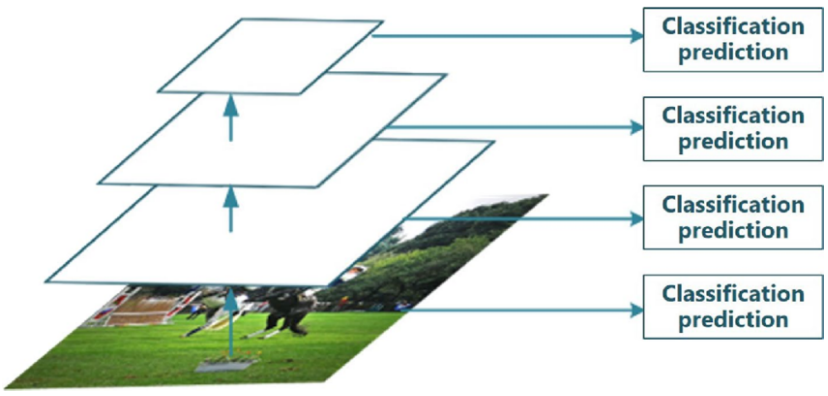


Figure 4.1: Pattern diagram of image pyramid

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 4.2

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 [7], 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

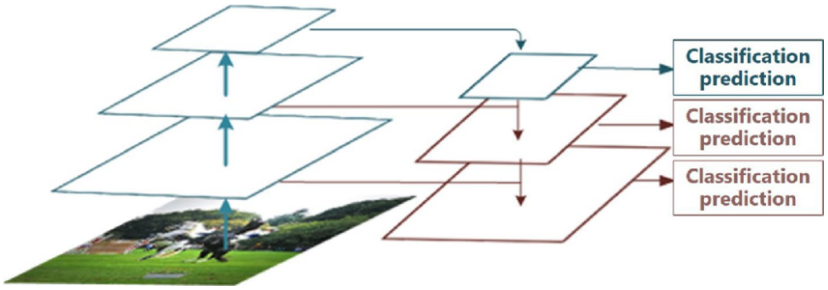


Figure 4.2: Pattern diagram of feature pyramid network

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. [6] 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. [6].

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 [6] 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 4.3.

In addition to the bidirectional cross-scale connections, the EfficientDet model takes into consideration the importance of the different features at different resolutions 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. [6]

Thus, the EfficientDet model combines all these concepts into a singular network, using ImageNet pretrained weights for the EfficientNet model, the BiFPN levels,

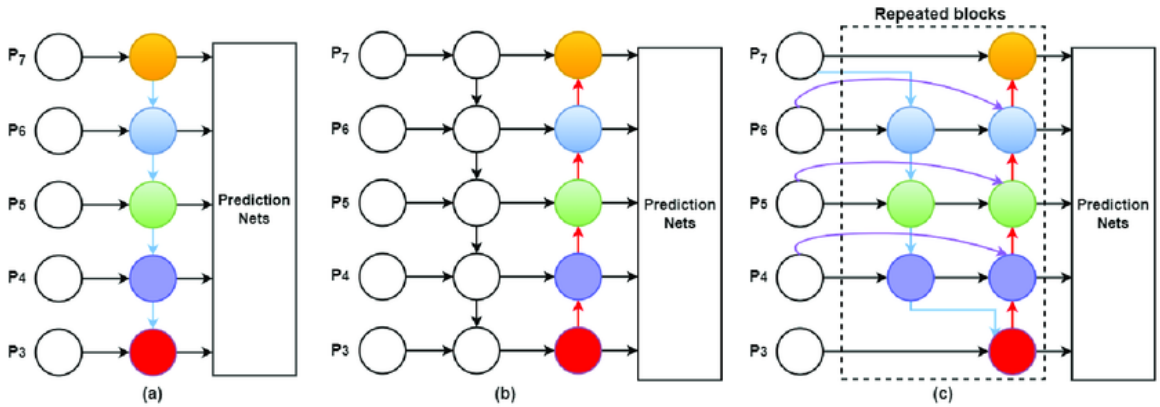


Figure 4.3: (a)-Conventional FPN; (b)-PANet; (c)-BiFPN

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 4.4

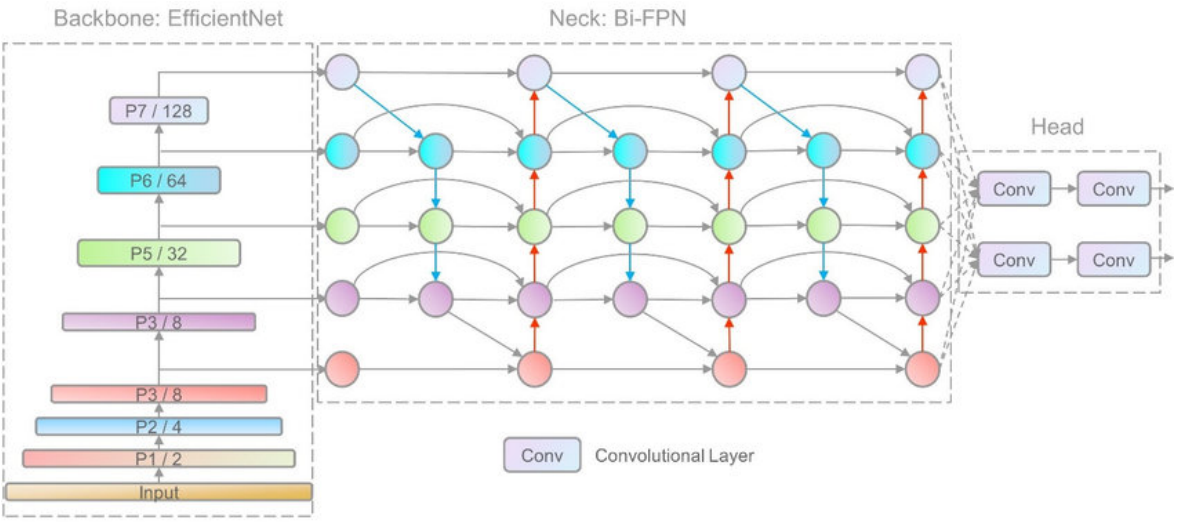


Figure 4.4: EfficientDet architecture

Chapter 5

Experimental results obtained

5.1 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 [6] 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. I was training images of size 2457,1890 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.

5.2 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 accepted inputs of 3xWxH.
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. 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.
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 below 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

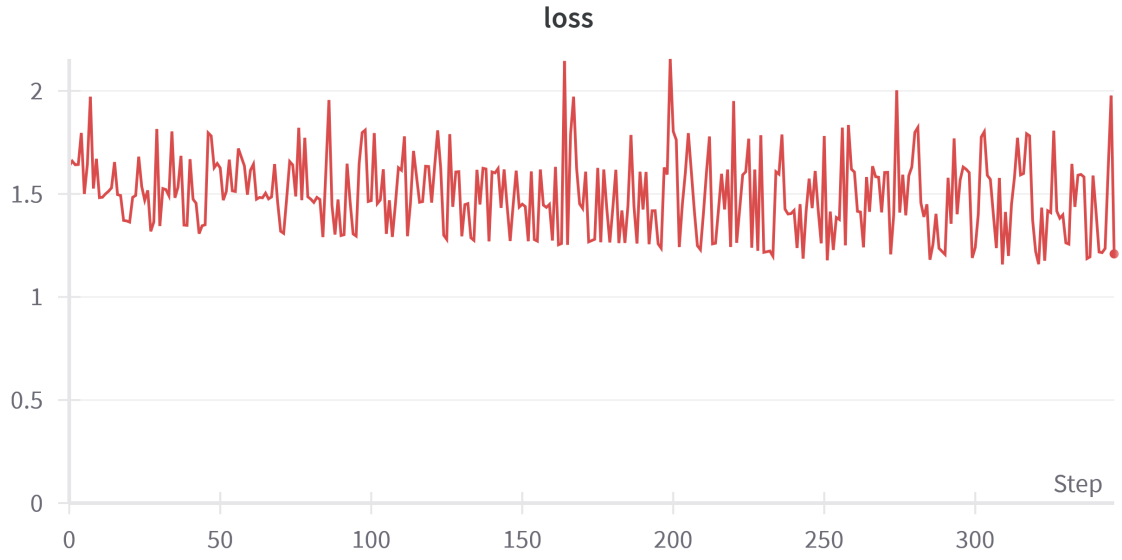
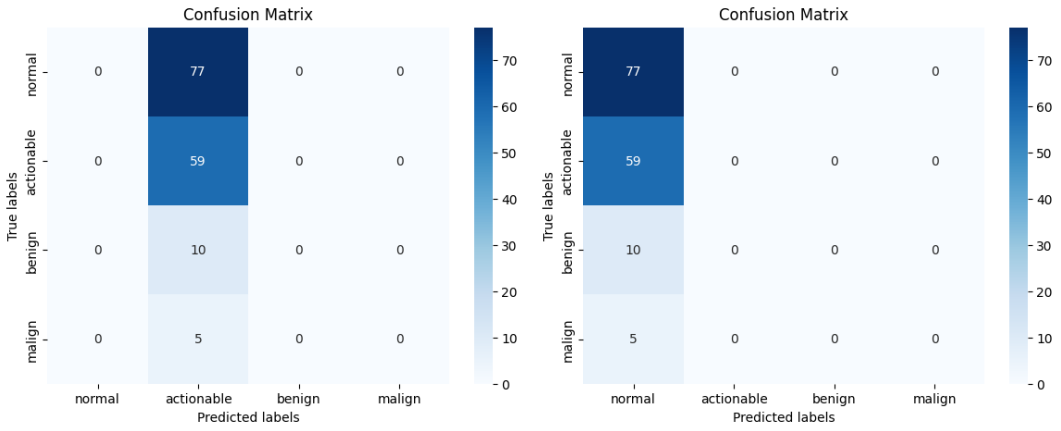


Figure 5.1: Loss Graph

step. This image 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. This is how

the confusion matrix looked after the first epochs, and after the sixth one.

The second experiment proved to be a little more fruitful. I changed the learning



(a) (b)

Figure 5.2: (a) After epoch 1 (b) After epoch 6

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 GoogleNet provides. 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. Image 5.3 show 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 only after the first epoch. Curiously, it did not predict any images to be of type benign, even after eight epochs. Image 5.2 show how the predictions looked after the first and eighth epochs.

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

The spikes in image 5.3 represent the batches that had more input of classes benign

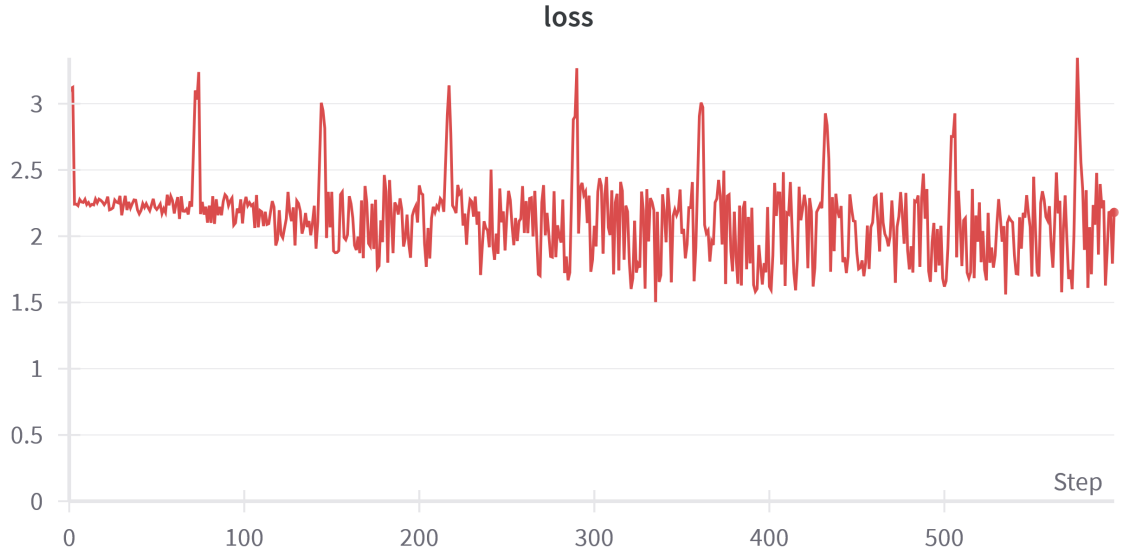
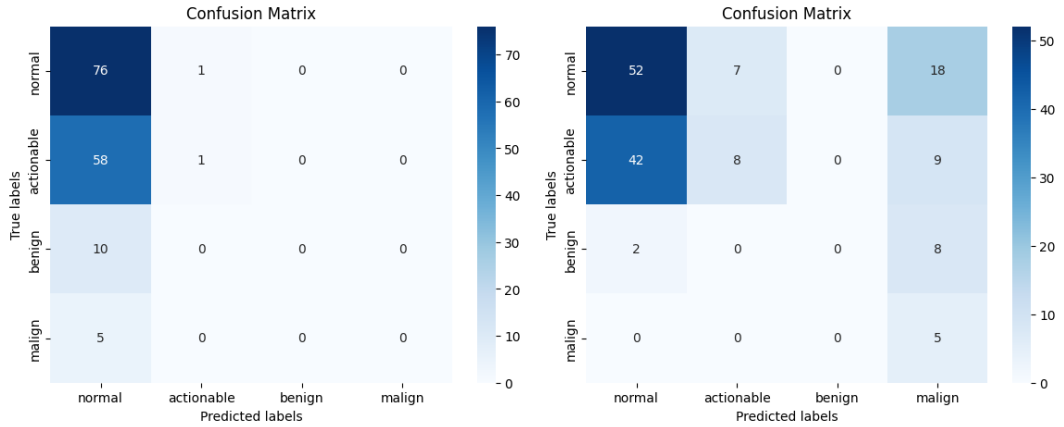


Figure 5.3: Loss graph



(a) (b)

Figure 5.4: (a) After epoch 1 (b) After epoch 8

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 5.5 shows how the experiment turned out, so not much was changed, and the predictions overall on the test data were not different.

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 from the third to the fourth experiment, except for batch size, which I changed to 64. Image 5.6 shows how the loss

Figure 5.5: Loss graph

looked after training three epochs, and image 5.7 shows how the confusion matrix looked after those three epochs.

I resized the images to be 300x300 and instead of adding the slices as channels of

Figure 5.6: Loss graph

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 millions, 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

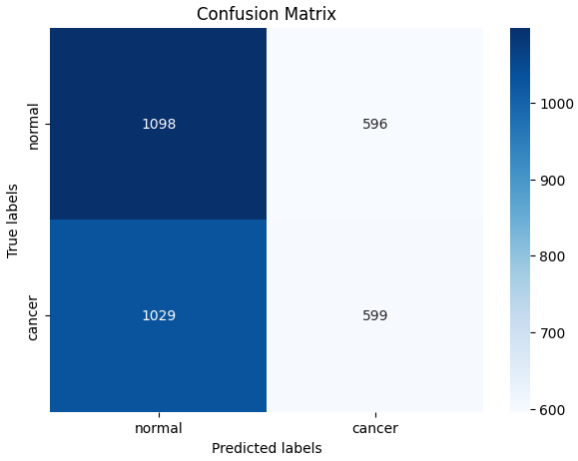


Figure 5.7: After 3 epochs

resulted in a 12,496-length dataset.
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 5.8
Even with all these experiments and improvements, the best accuracy I managed

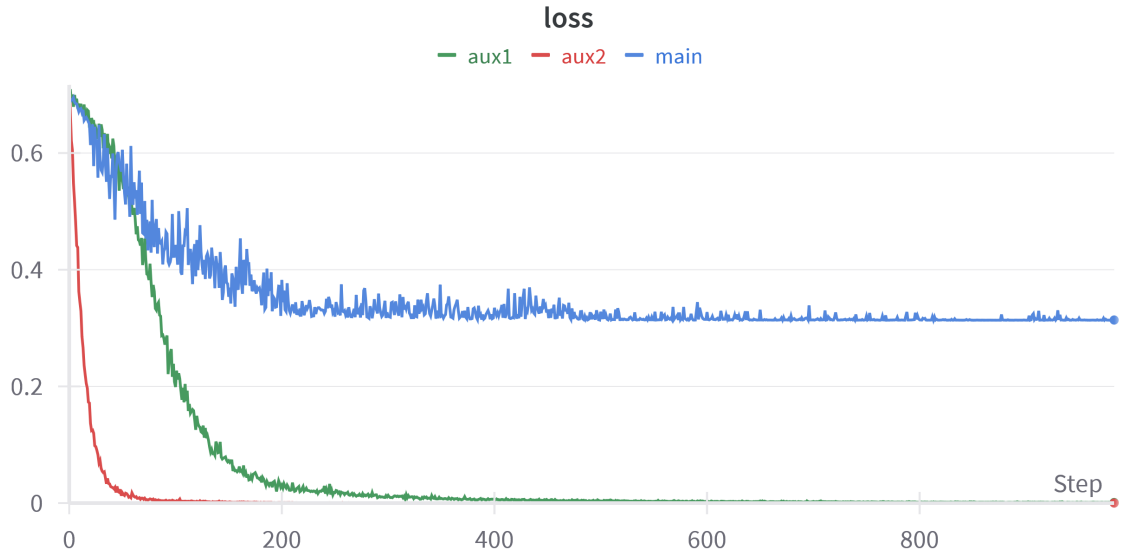


Figure 5.8: Loss graph

to obtain was 0.5409. Towards the end of the process of training the classification model, I decided to use already trained weights from the Torchvision library. These

weights had been trained on images from ImageNet. I also resized the images again and made them 512x512 in order to also fit the detection model. Figure 5.9 shows the loss graph after training the model for 5 epochs, while figure 5.10 shows the confusion matrix at the end of those epochs.

I also experimented with different batch sizes for the model. Therefore, I trained

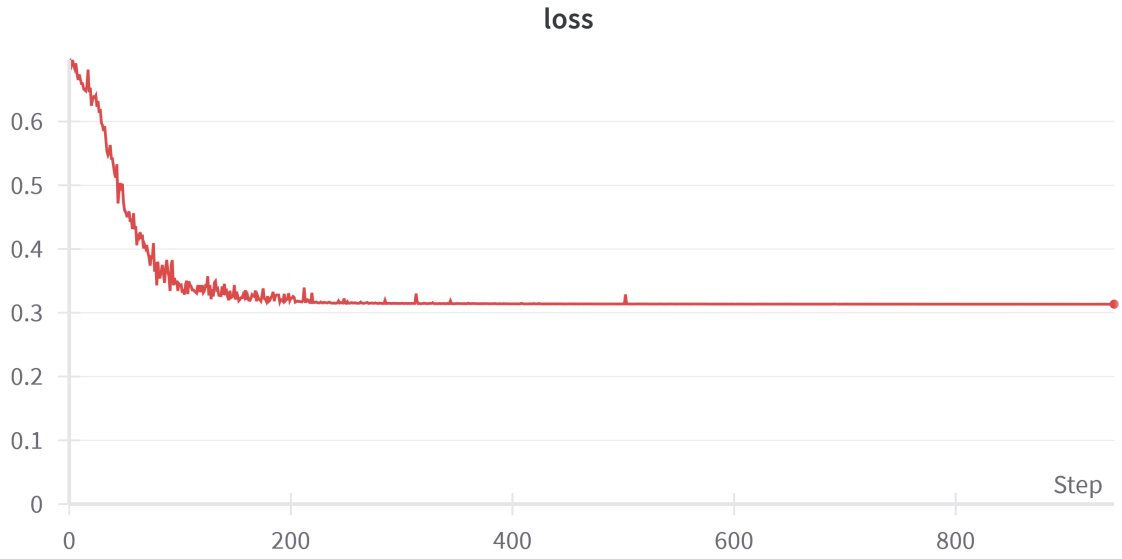


Figure 5.9: Loss graph

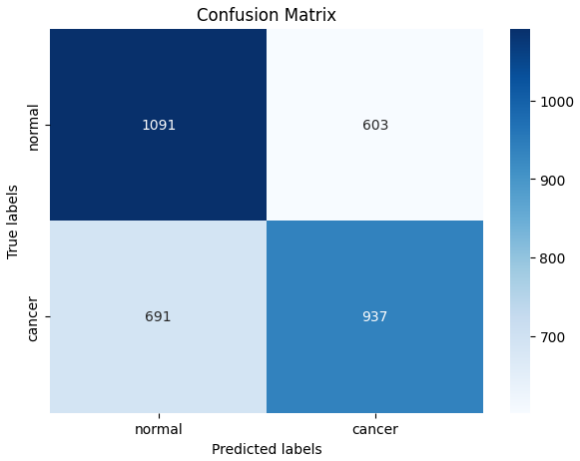


Figure 5.10: After epoch 5

the model with the same configuration as the previous run, except I modified the size of the batches from 64 to 128. Figure 5.11 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.

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

Figure 5.11: Loss graph

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 5.12.

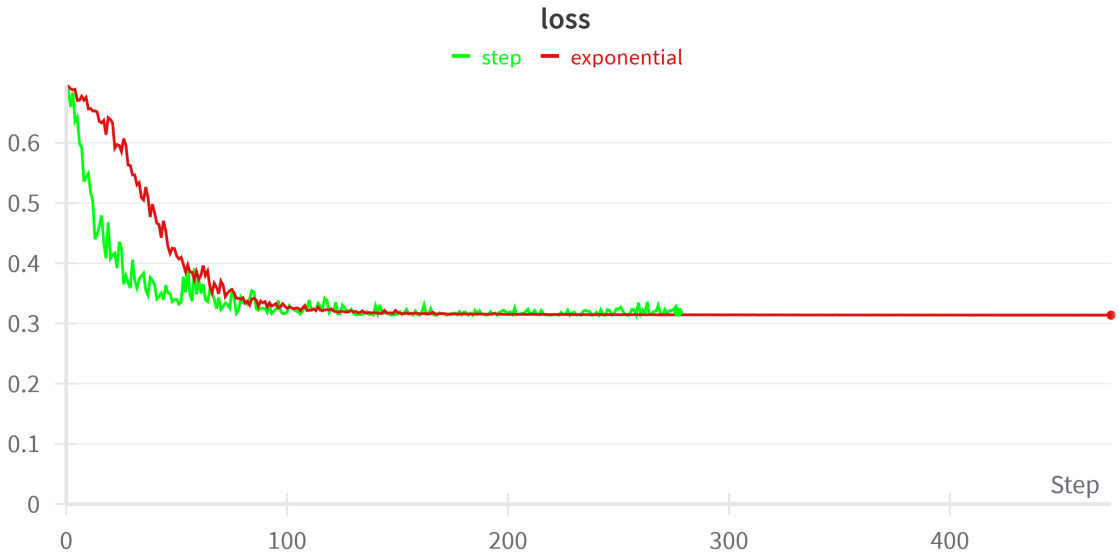
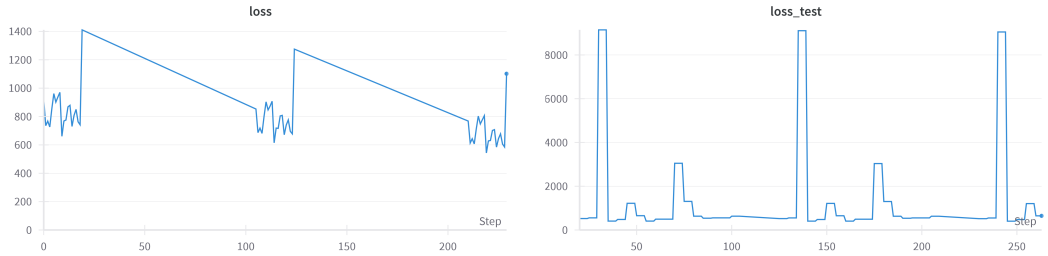


Figure 5.12: Loss graph

5.3 Detection experiments

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. The number of slices used from each input remained at 22, as for the final configuration for the classification model, and the size of the batches was changed to 16. Considering the fact that the images were resized in order to be 512x512, I scaled the coordinates accordingly. 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. Figure 5.3 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 are in the complementary graph. The figure does not indicate any loss reduction, so I changed the configuration. I



(a) (b)

Figure 5.13: (a) Train loss (b) Test loss

chose not to normalize the images 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 [16] and I reset the Head Nets in order to fit my classification problem. I stopped the run before it had a chance to validate the model, so there is no graph for the test loss. Figure 5.14 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 run 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. In the first 10 epochs, shown in figure 5.3, the loss decrease is steep, and so is the test loss. However, after 40 epochs, the decline gets

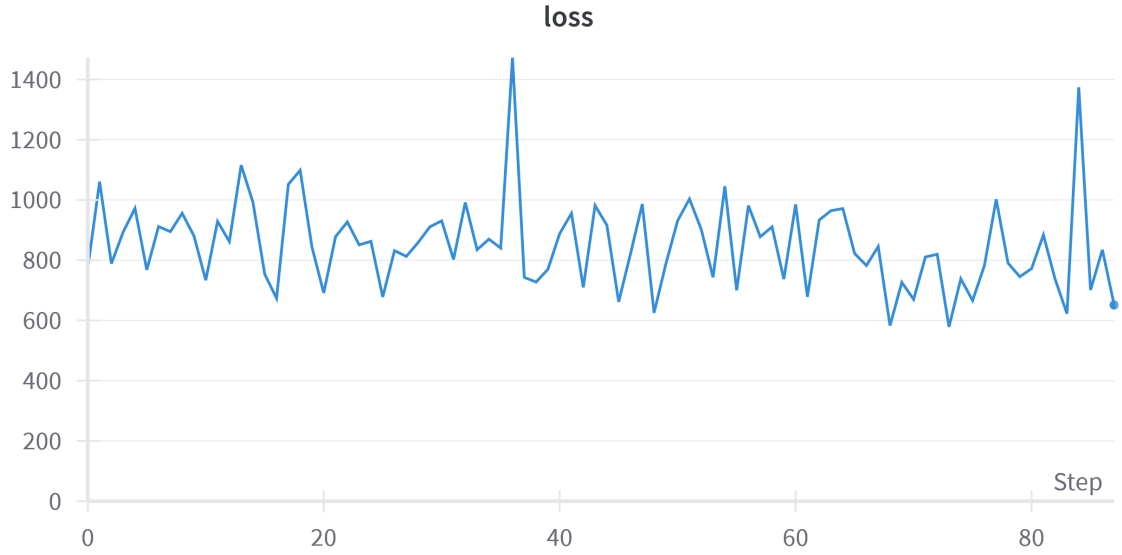
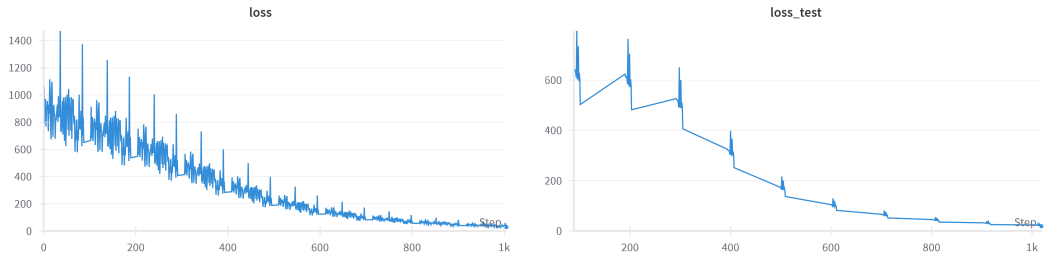


Figure 5.14: Loss graph

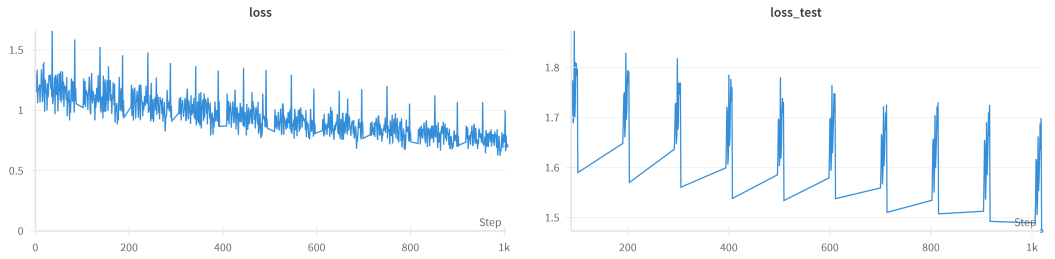
more gradual, as shown in figure 5.3.

For the next experiment, I normalized the images again to see if I could get a starting



(a) (b)

Figure 5.15: (a) Train loss (b) Test loss



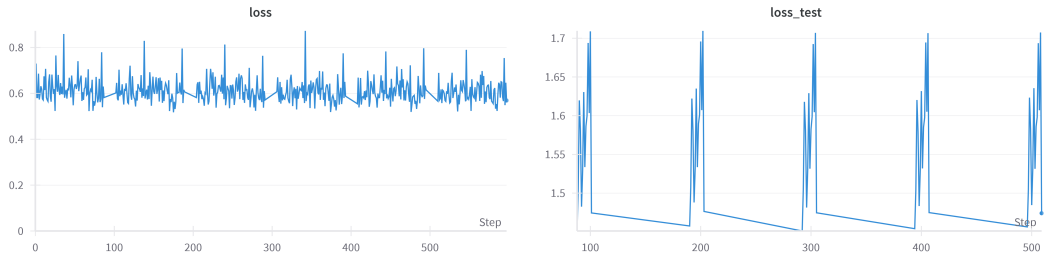
(a) (b)

Figure 5.16: (a) Train loss (b) Test loss

loss lower than the previous ones. I also found another file containing pre-trained

weights, this time from the official Effdet library on GitHub [17]. With this configuration, I trained 13 epochs. Figure 5.3 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.

In the next experiment, I changed the learning rate decay from ReduceLROnPlateau



(a) (b)

Figure 5.17: (a) Train loss (b) Test loss

to CosineAnnealingLR, with a minimum learning rate of $1e^{-6}$. I also modified the way the decay is applied, calling it after the train loss 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: $create\_model\_from\_config(bench\_task = 'train', num\_classes = 2, pretrained = True)$.

Chapter 6

Evolution of the Application Interface

When coming up with designs for the interface and researching various Python libraries for GUIs, I came across the Gradio interface that is used specifically for machine learning models. Using this library, there are two ways to submit an image to the model for predictions: uploading the image from the device or submitting an already uploaded image from an array of them. This array represents images used as examples, which in my case reside in a directory in the project's root.
Implementing such an interface using the Gradio library was fairly simple, but I also wanted to display the correct label of the photo submitted along with the predictions. 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.
Because I modified the problem from a 4-label classification to a 2-label classification, I chose to train another label trained in detecting actionable-type images from the benign and malign ones. This second model is only used if the first model predicts the image as not being normal. I do this because the third model, which detects the bounding boxes, is trained on benign and malign images, and along with the coordinates of the boxes, it provides the classification of the input. The image below shows the interface of the application that detects inputs using the first model.

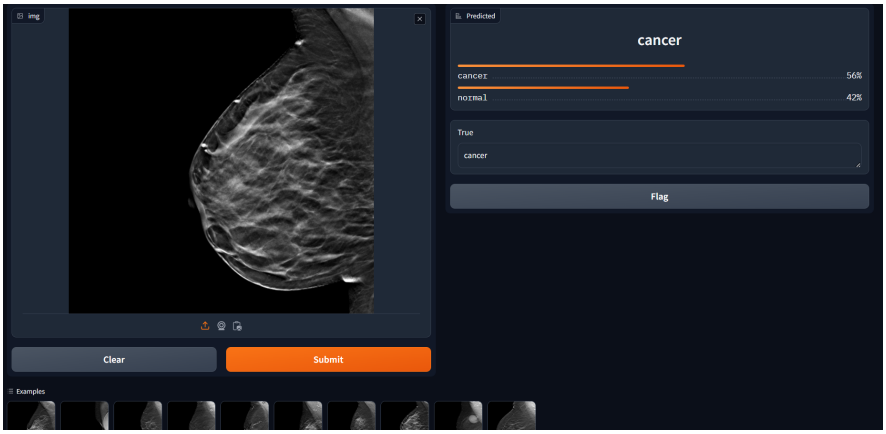


Figure 6.1: Application interface

Chapter 7

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.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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