Faculty of Engineering and Technology
Department of Computer and Informatics Engineering

"PCOS Detection in Ultrasound Images via Deep Learning and CNNs"

Submitted in partial fulfillment of the requirement of degree of bachelor in informatics Engineering

BY

SEYED YOUSSEF SEYED LOEI HEYDARI

Supervised By

Dr. Mohammed Hayyan ALSIBAI

Senior Project II
2022-2023

Abstract:

This thesis presents a study on the use of deep convolutional neural networks (CNN) models such as DenseNet models to classify ovarian ultrasound images into healthy and infected cases. A much smaller model (IustNet) is also built from scratch in which its performance is compared to the DenseNet models. The datasets used in the study consisted of healthy ovaries and ovaries with Polycystic Ovary Syndrome (PCOS). Using transfer learning, a pre-trained DenseNet201 model was fine-tuned to train on the available dataset. Then, a DenseNet121d model is trained from scratch on the same dataset. Finally, IustNet is built from scratch and trained also on the dataset. Performance of the three models on the dataset was evaluated using accuracy, precision, and recall. Results of the study showed that DenseNet121d achieved the most desired outcome due to obtaining the highest recall which of utmost importance in medical classification tasks. The accuracy of the proposed model was able to achieve a somewhat satisfactory accuracy of roughly 72% on the test set. The suggested model was also deployed on the web for accessibility by doctors for usage as a decision support system. Lastly, the study also highlights the importance of the quality of the dataset in achieving more accurate results. This research contributes to the growing body of literature on the application of deep learning in medical imaging and has the potential to be applied to other medical imaging tasks.

Table of Contents

List of Figures:

List of Tables

Table of Abbreviations

DL	Deep Learning
CNN	Convolutional Neural Network
AI	Artificial Intelligence
PCOS	Polycystic Ovary Syndrome
ML	Machine Learning
ANN	Artificial Neural Network
TF	Transfer Learning
DenseNet	Densely Connected Convolutional Networks
SGD	Stochastic Gradient Descent
BCE	Binary Cross Entropy
GPU	Graphics Processing Units

Acknowledgement:

I would like to express my sincere gratitude to Dr. Mohammed Hayyan Al-Sibai, for his invaluable support and guidance throughout the course of this project. His expertise and knowledge have been instrumental in the success of this project, and I am deeply grateful for his mentorship. I would also like to thank him for his patience and understanding, as well as for always being available to offer support and answer questions. This project would not have been possible without his guidance and support.

شكر :

أود أن أعرب عن خالص امتناني للدكتور محمد حيان السباعي على دعمه وتوجيهه القيمين طوال فترة هذا المشروع. لقد لعبت خبرته ومعرفته دوراً أساسياً في نجاح هذا المشروع ، وأنا ممتن للغاية لتوجيهه. كما أود أن أشكره على صبره وتفهمه ، وكذلك على تواجده دائمًا لتقديم الدعم والإجابة على الأسئلة. لم يكن هذا المشروع ممكناً لولا توجيهه ودعمه.

Chapter 1 Introduction

1.1 Background

Polycystic ovary syndrome (PCOS) is one of the most prevalent conditions affecting women of reproductive age. It affects 6%–20% of premenopausal women globally [1]. Ovarian dysfunction and an excess of androgen are the two main symptoms of PCOS. Menstrual abnormalities, hirsutism, obesity, insulin resistance, cardiovascular disease, in addition to emotional symptoms like depression [2] are all common among PCOS patients [3]. As a result, it is crucial for the accurate diagnosis and treatment of PCOS. In 1985, Adams et al. [4] discovered that polycystic ovaries have an abnormally high number of follicles, also termed multifollicularity. It was suggested that PCOS be diagnosed when at least two of the three following features were present at an expert meeting in Rotterdam in 2003: (i) oligo- or anovulation, (ii) clinical and/or biochemical hyperandrogenism, or (iii) polycystic ovaries [5]. The latter of which can be detected using ultrasonography which offers the highest contribution to the diagnosis of PCOS [6]. Early detection of PCOS is important because it can help to manage the symptoms and reduce the risk of long-term health issues.

Although ultrasound images have some disadvantages of strong artifacts, noise and high dependence on the experience of doctors, they are still considered as one of the most widely used modalities in medical diagnosis. Many artificial intelligence systems have been developed to help doctors. Convolutional Neural Networks (CNNs) and deep learning has achieved great success in computer vision with its unique advantages [7]. Many diseases are diagnosed using different Deep Learning Models [7]. Some examples include the detection of COVID-19 using lung ultrasound imagery achieving 89.1% accuracy using InceptionV3 network [8], the use of deep learning architectures for the segmentation of the left ventricle of the heart [9], and the classification of breast lesions in ultrasound images obtaining an accuracy of 90%, sensitivity of 86%, and specificity of 96% by utilizing the GoogLeNet CNN [10]. As we can see, deep learning has proved its potential and the vital role it can provide in benefitting and assisting practitioners that use ultrasonography as a tool for diagnosis. This thesis is discussing the potential of deep

learning in diagnosing PCOS. Since AI and deep learning algorithms can quickly and reliably assess vast volumes of data, they can be utilized to diagnose PCOS in ultrasound scans. AI and deep learning algorithms can examine ultrasound images to find patterns and traits that are indicative of PCOS in the case of PCOS detection. This can help to increase the speed and accuracy of diagnosis as it can be done more accurately and efficiently than by manual analysis. Furthermore, the application of AI and deep learning in the diagnosis of PCOS can decrease the workload for medical professionals and free them up to concentrate on other responsibilities. Overall, the use of AI and deep learning in the detection of PCOS in ultrasound images has the potential to improve the accuracy, efficiency, and accessibility of healthcare. This was the motive to tackle such an important health issue that affects millions of women worldwide and apply the potential of deep learning to help nullify this crucial problem. Obtaining a viable and correct ultrasound dataset for this task is difficult and time-consuming as the annotation of medical images requires significant professional medical knowledge, which makes the annotation very expensive and rare as well as the ethical issues and sensitivity of such dataset which can pose another problem. Therefore, resorting to a publicly available dataset can hugely accelerate the work on this project. After observing some of the related work, one publicly available PCOS dataset was utilized in the training of PCONet, a CNN developed by Hosain AK et al. that detects PCOS from ovarian ultrasound images with accuracy of 98.12% on test images as well as fine-tuning InceptionV3 model achieving 96.56% accuracy [11]. The PCOS dataset is publicly available on Kaggle [12]. Other related work includes Wenqi Lv et al. who utilized image segmentation technique using U-Net on scleral images then a Resnet model was adapted for feature extraction of PCOS achieving classification accuracy of 0.929, and AUC of 0.979 [13]. Subrato Bharati et al. used clinical attributes of 541 women in which 177 are infected with PCOS to be utilized in a machine learning model that uses random forest and logistic regression to predict PCOS disease for which the testing accuracy achieved is 91.01% [14]. Sakshi Srivastava et al. employed a fine-tuned VGG-16 model to train on their dataset that consists of ultrasound images of the ovaries to detect the presence ovarian cyst with 92.11% accuracy obtained [15].

1.2 Problem Statement:

Polycystic Ovary Syndrome (PCOS) is a common hormonal disorder that affects approximately 6%-20% of women of reproductive age. It is often characterized by irregular menstrual cycles, insulin resistance, and the presence of multiple cysts on the ovaries. Currently, the diagnosis of PCOS is based on a combination of clinical and laboratory findings, including ultrasound imaging. However, the diagnosis of PCOS based on ultrasound images can be subjective and may vary among different practitioners. This study aims to develop a deep learning-based classification system using convolutional neural networks (CNNs) to accurately detect and classify ultrasound images of ovaries with and without PCOS. The specific research question that this study aims to answer is: Can a deep learning-based classification system using CNNs accurately detect and classify ultrasound images of ovaries with and without PCOS?

1.3 Objectives

To achieve the main target of the project, the following objectives are considered:

1. To apply transfer learning to a DenseNet model for the classification of ultrasound images of ovaries.
2. To evaluate the accuracy of the DenseNet model in classifying ultrasound images of healthy ovaries versus ovaries with PCOS.
3. To identify any factors that may influence the performance of the DenseNet model, such as the size of the training dataset or the quality of the dataset.
4. Improve the accuracy and efficiency by trying different approaches such as training different variations of the DenseNet architecture from scratch as well as building an entire model from scratch and training it on the available dataset.
5. Deploy the model and make it accessible to the public.

Chapter 2 Methodology

2.1 Introduction

Figure 1 shows a basic block diagram of the entire system. It consists of the main components that needs to be addressed when building a deep learning model that predicts on ultrasound images of ovaries.

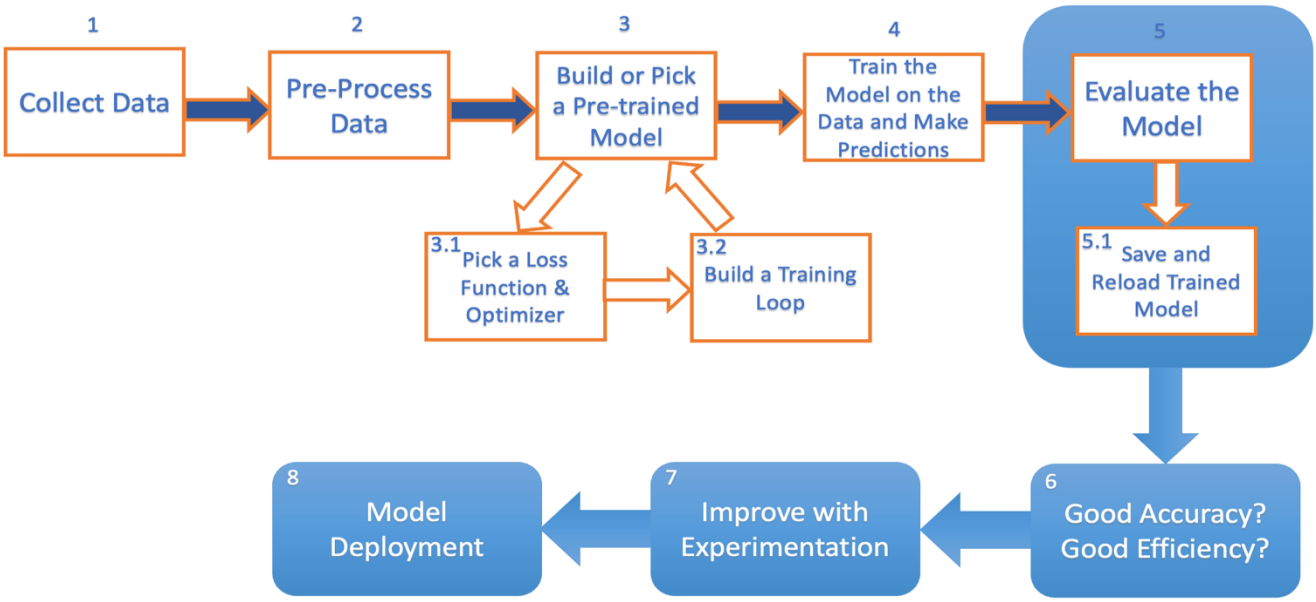


Figure 1: General block diagram of the entire system

1. **Collect Data:** The first step in any deep learning project is to collect the data that will be used to train and evaluate the model. This may involve scraping web data, collecting data from sensors or other devices, or using a publicly available dataset.
2. **Pre-Process Data:** Once the data has been collected, it is usually necessary to pre-process it in order to prepare it for use with a deep learning model. This may involve tasks such as cleaning the data, selecting a subset of the data to use, or transforming the data into a suitable format.
3. **Build or Pick a Pre-trained Model:** The next step is to either build a deep learning model from scratch or use a pre-trained model that has already been trained on a large dataset. Building a model from scratch requires a good understanding of deep learning concepts and techniques, as well as a strong ability to implement

these techniques using a deep learning framework such as TensorFlow or PyTorch. Using a pre-trained model can be a good option for those who are less familiar with deep learning, as it allows them to leverage the knowledge and expertise of others who have already invested the time and effort to build a high-quality model.

3.1. **Pick a loss function & optimizer:** In order to train a deep learning model, it is necessary to define a loss function and an optimizer. The loss function measures how well the model is performing on the training data, and the optimizer adjusts the model's parameters in order to minimize the loss. There are many different loss functions and optimizers to choose from, and the appropriate choice will depend on the specifics of the problem at hand.

3.2. **Build a training loop:** The training loop is the mechanism by which the model is actually trained on the data. It typically involves repeating the following steps multiple times: (1) reading a batch of data from the training set, (2) passing the data through the model to make predictions, (3) computing the loss between the predictions and the ground truth labels, (4) backpropagating the loss through the model to update the model's parameters, and (5) updating the optimizer with the updated parameters.

4. **Train the Model on the Data and Make Predictions:** Once the training loop is set up, the model can be trained by repeatedly executing the training loop. After training is complete, the model can be used to make predictions on new data by simply passing the data through the model and using the output as the prediction.
5. **Evaluate the Model:** After the model has been trained and used to make predictions, it is important to evaluate its performance in order to determine how well it is able to generalize to unseen data. This may involve calculating metrics such as accuracy, precision, and recall on a separate evaluation dataset, and constructing a confusion matrix.

5.1. **Save and Reload Trained Model:** It is often useful to save a trained model so that it can be used later without the need to retrain it from

scratch. This can be done using a deep learning framework's built-in model saving utilities. The saved model can then be reloaded at a later time and used to make predictions or to fine-tune it on additional data.

6. **Good Accuracy? Good Efficiency?**: Evaluating the model's performance in terms of accuracy and efficiency. Does the model provide acceptable level of accuracy, and could it be improved? Is it ready for deployment? Or are further improvements with experimentations necessary? Efficiency considerations will involve analyzing factors like inference time, and computational complexity to ensure the model is feasible for real-world deployment.
7. **Improve with Experimentation:** Explore techniques to enhance the model's performance through experimentation. This will involve modifying the model architecture such as reduce/increase the layers in the deep neural network, training another model scratch rather only the classifier layers, building another new smaller model from scratch, adjusting hyperparameters…
8. **Model Deployment:** The final section focuses on deploying the trained model for real-world use. Considerations such as scalability, reliability, ease of integration, data handling, model updates, and performance monitoring will be discussed. The goal is to enable the practical application of the PCOS detection model in healthcare settings, benefiting both patients and healthcare providers.

2.2 Collecting Data

Data is crucial in machine learning and deep learning because it is the input that drives the training and evaluation of models. Without sufficient and high-quality data, it is difficult to build accurate and reliable models. In the medical field, collecting data for a classification project on healthy ovaries versus ovaries with PCOS can be challenging due to the sensitive nature of the data and the need to ensure patient privacy. To collect data for this project, one approach could be to obtain data from publicly available medical datasets, with the appropriate permissions and safeguards in place. It may also be possible to collect data through clinical studies or by partnering with hospitals or clinics. It is important to carefully plan and execute the data collection process to ensure that the data is representative and relevant to the project goals.

In this project, the Kaggle dataset [12] highlighted previously is used to conduct training by enhancing the power of transfer learning to train an existing model architecture that is pre-trained on thousands of images in advance. This same dataset is referred to as *dataset A* in the highlighted paper of Hosain AK et al. [11] mentioned previously and will be referred to as dataset A in this project as well.

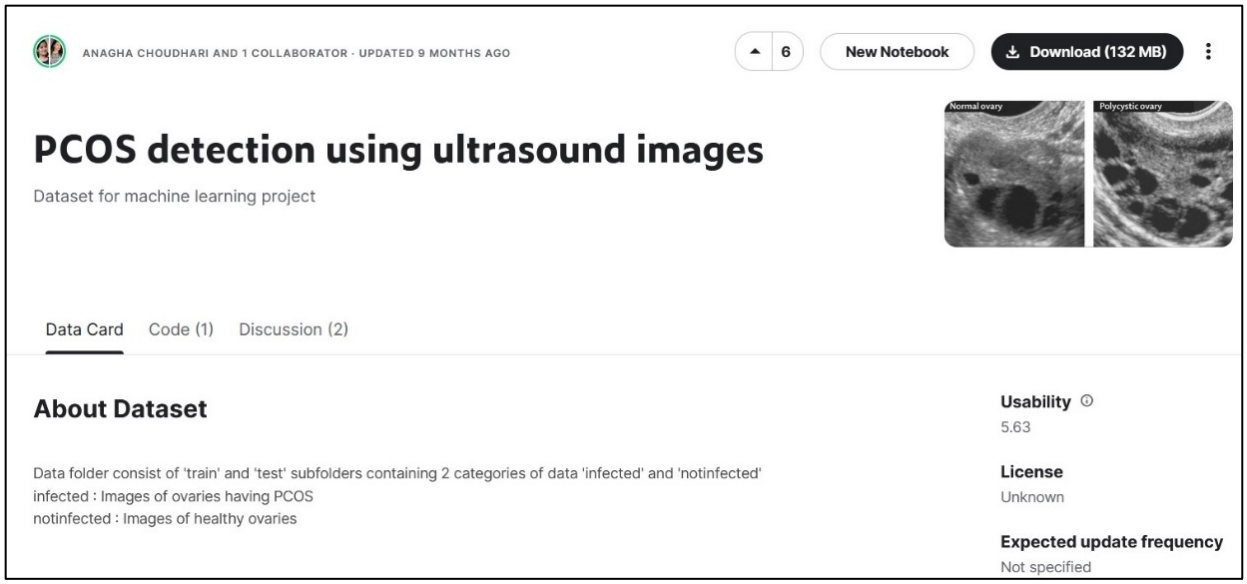


Figure 2: Screenshot of the publicly available PCOS dataset on Kaggle consisting of `infected` and `notinfected` ovarian ultrasound images referred to as Dataset A.

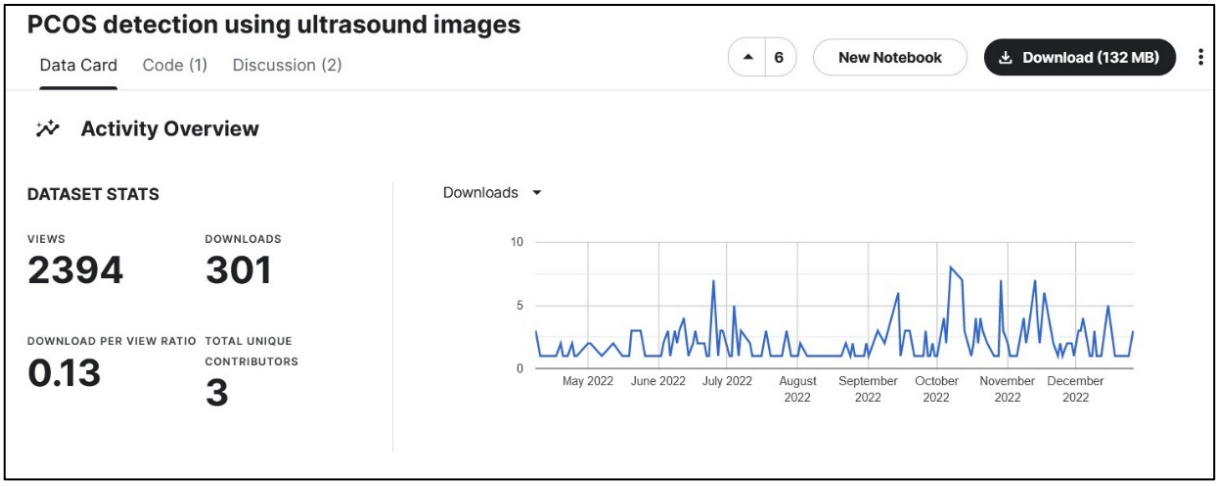


Figure 3: Statistics of Dataset A shows that it is downloaded 301 times out of 2394 views. This means that almost 1 out of every 10 viewers downloads this dataset for utilization in research/projects.

The dataset consists of 3856 ultrasound images divided into 2 classes which are `infected` and `notinfected`. The latter depicts healthy ovaries and the first indicates the presence of PCOS. These images are partitioned into train and test sets in which 1932 images belong to the test set and the rest belong to the train set. However, the same images seem to be repeated in the test and train directories. Therefore, one of the directories will be neglected. Figure 4 depicts a sample of the ultrasound images present in this dataset.

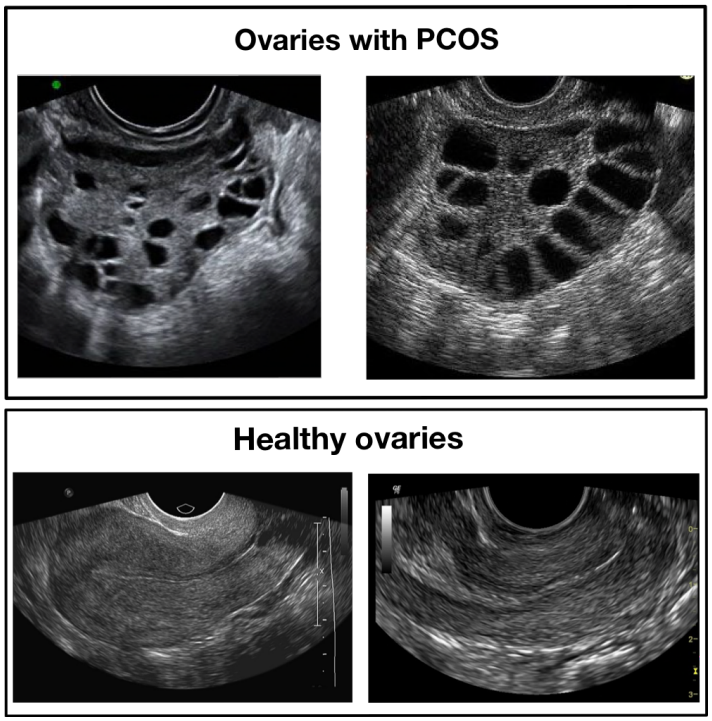


Figure 4: Samples of infected and healthy ovaries in Dataset A

2.3 Pre-Processing Data

There are several steps that may be taken to preprocess medical ultrasound images before they are input into a deep learning model for training. Some common preprocessing steps include:

1. **Image resizing:** The images may be resized to a uniform size to make it easier for the model to process them.

2. **Image normalization:** The intensity values of the pixels in the images may be normalized to a range of [0,1] or [-1,1] to improve the model's ability to learn from the data.

3. **Image augmentation:** The images may be augmented with additional data to help the model generalize better. This can be done by applying various transformations such as rotation, scaling, and cropping to the images.

4. **Turn the data into batches**: Data is typically grouped into smaller batches when training a deep learning model to improve memory efficiency, computational efficiency, and regularization. This allows the model to learn from the data more efficiently and can improve its performance.

It's important to note that the specific preprocessing steps will depend on the characteristics of the images and the specific requirements of the deep learning model being used. Since the model chosen for this project is the DenseNet201 architecture which is a pre-trained model on the ImageNet dataset, the ovarian ultrasound images that will be the input to this model need to be pre-processed the same way as the images this model was originally trained on. These images vary in size and dimensions therefore they need to be preprocessed so that they are uniform in this aspect. The preprocessing pipeline include:

1) **Resizing** all the images into the same size of (224, 224), which is the image size that DenseNet201 was trained on.
2) Transforming the images into **tensors**. In Pytorch, a tensor is a multi-dimensional array that is similar to a Numpy array, but can be operated on by the GPU, which makes it more efficient for certain types of computations. Tensors can be used to store a wide variety of data, including images, videos, and audio, and are an important building block in PyTorch. The resulting tensor has the same number of dimensions as the original image, with the size of each dimension being the size of the corresponding image dimension. The pixel values of the image are also normalized to the range [0,1] when converted to tensors.
3) **Normalize the images**: It is common to normalize the images by subtracting the mean and dividing by the standard deviation of the image pixels. This helps to

center the data and improve the model's ability to learn from it. The mean and standard deviation values for each color channel are:

`mean = [0.485, 0.456, 0.406] std = [0.229, 0.224, 0.225]`

This is done due to it being mentioned as a necessary pre-process in the documentation [16] of DenseNet201.

4) Transforming all ultrasound images to **grayscale** as this will eliminate the unnecessary color channels if they exist and the resultant image will have only 1 color channel due to it being grayscale.
5) The images and their corresponding labels are turned into **batches** where each batch consists of 32 images. This is very beneficial for a variety of reasons:
    a. Memory constraints: Training a deep learning model on a large dataset can require a lot of memory. Grouping the images into batches allows the model to train on a subset of the data at a time, which can be more memory efficient.
    b. Computational efficiency: Processing the images in batches can be more computationally efficient, as it allows the model to make better use of the parallel processing capabilities of modern hardware such as GPUs.
    c. Stochastic gradient descent: When training a model using stochastic gradient descent, it is common to update the model's weights using the gradients computed on a small batch of data rather than the entire dataset. This is because computing the gradients on the full dataset can be computationally expensive, and using a smaller batch of data can provide a good approximation.
    d. Regularization: Using small batches of data can also introduce noise into the learning process, which can act as a form of regularization and help the model generalize better to new data.

2.4 Build or Pick a Pre-Trained Model

After pre-processing the data, it is time to either build a model from scratch or use transfer learning. From reviewing related work, almost all the research done in the medical field that uses deep learning utilizes transfer learning to train on the data through a pre-trained

model. In fact, the conclusion of this paper [17] mentions that transfer learning is the best available option for training even if the data the model is pre-trained on is weakly related to data in hand.

Thus, in order to train on this dataset, the DenseNet architecture [18] was chosen. DenseNet is one of the leading architectures used in ultrasonography analysis when conducting transfer learning as *table* 3 in this paper [19] shows. DenseNet is a variation of the traditional CNN architecture that uses dense blocks, where each layer in a block is connected to every other layer in the block. This allows the network to pass information from earlier layers to later layers more efficiently, alleviate the problem of vanishing gradients in deeper networks, and can improve performance on tasks such as image classification. There are several variations of the DenseNet architecture, such as DenseNet121, DenseNet201, and DenseNet264. Each variation refers to the number of layers it consists of. DenseNet201 is the variation chosen for this project as it has a moderate number of layers and is not too computationally demanding. DenseNet and many other pre-trained architectures are pre-trained on the ImageNet dataset which is long-standing landmark in computer vision [20]. It consists of 1,281,167 training images, 50,000 validation images, and 100,000 test images belonging to 1000 classes. These images consist of a wide variety of scenes, covering a diverse range of categories such as animals, vehicles, household objects, food, clothing, musical instruments, body parts, and much more. A question might arise from this information, why would a model trained on a such unrelated dataset be used for training on and predicting medical images?

In fact, the paper *"A scoping review of transfer learning research on medical image analysis using ImageNet"* [19] discusses this very topic and after inspecting tens of research papers and studies that utilize ImageNet models to train on medical datasets, proves that transfer learning of ImageNet models is a viable option to train on medical datasets. The idea behind transfer learning is that although medical datasets are different from non-medical datasets, the low-level features (e.g., straight and curved lines that construct images) are universal to most of the image analysis tasks [21]. Therefore, transferred parameters (i.e., weights) may serve as a powerful set of features, which reduce the need for a large dataset as well as the training time and memory cost [21].

The structure of DenseNet is shown in figure 5:

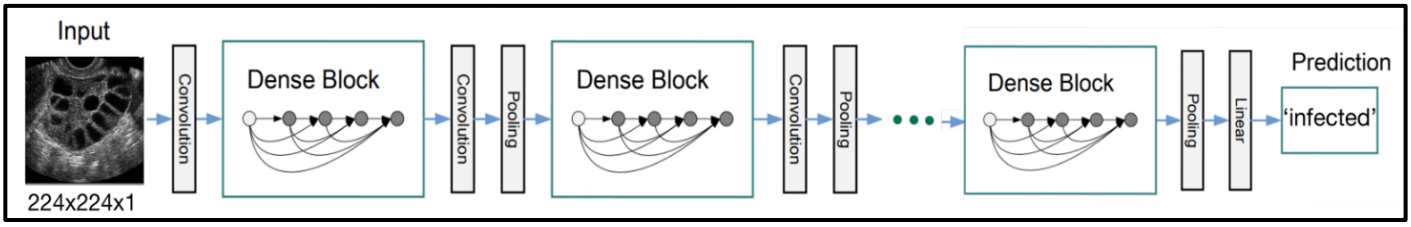


Figure 5: An ultrasound image with size (224, 224, 1) as an input to the DenseNet model using its weights and architecture to make a prediction.

2.4.1 Fine-Tuning DenseNet201

In order to utilize transfer learning for importing DenseNet201 and using it on the ovarian ultrasound dataset, the model must be fine-tuned. This fine-tuning involves 2 stages:

1. **Adjusting the very first layer** to make it accept grayscale images that are composed of 1 color channel as opposed to 3 color channels that the ImageNet dataset consists of which is used to train the DenseNet201 model on. This is a better method than expanding a single channel to 3 channels because it requires additional resources to store and process the additional channels that don't provide any new information.
2. ImageNet dataset consists of 1000 classes. Therefore, the DenseNet201 model also has 1000 corresponding outputs, 1 output probability for each class. This dataset consists of 2 classes only. Thus, **the very last layer is adjusted to output 1 probability** only which will be rounded to either 0 if the value is below 0.5 which indicates that the image is `infected` or 1 if the value is above 0.5 which signifies that the image is `not infected`. The raw output of the model i.e., the logit which is the final unnormalized score of the model (unbounded real number) is converted into a probability (range 0→1) using the sigmoid function: $\boldsymbol{\sigma(x) = \frac{1}{1+e^{-x}}}$ where x is the model logit. After the logit is converted into a probability, it is then converted into label (0 or 1) by rounding the probability into either 0 or 1 to indicate the prediction of the model.

After fine-tuning the model, the total number of parameters in the model is 18,088,577 in which 1,921 is trainable, and 18,086,656 is non-trainable (frozen).

2.4.2 Pick a Loss Function and Optimizer

2.4.2.1 Loss function

When working on binary classification in PyTorch, the most common loss function to use is binary cross-entropy loss, also known as log loss. This loss function is appropriate for binary classification problems where the output of the model is a probability, and the goal is to minimize the difference between the predicted probabilities and the true labels. There are 2 variations of the binary cross-entropy loss in PyTorch, **BCELoss()** and **BCEWithLogitsLoss().** The main difference is that the first one expects the logits to be inputted into a sigmoid function then the output of the sigmoid is the input into this function to calculate the loss. The loss is calculated as:

loss = -(y * log(p) + (1 - y) * log(1 - p))

The latter expects the logits to be inputted directly as it has the sigmoid function integrated internally. Thus, there is no need for the logits to be inputted into the sigmoid beforehand. It is calculated as:

loss = -(y * log(sigmoid(input)) + (1 - y) * log(1 - sigmoid(input)))

So, the main difference between the two is the input format. **BCEWithLogitsLoss()** was chosen due to it being more numerically stable [22].

2.4.2.2 Optimizer

There are multiple options to choose from for an optimizer. The two most common optimizers used are Adam and Stochastic Gradient Descent (SGD). The latter was the choice for this project as this paper [23] mentions that SGD generally performs better on image classification tasks. The optimizer and the learning rate are closely related, as the optimizer uses the learning rate to determine the step size when making updates to the model parameters. The learning rate opted for is 0.01.

2.4.3 Build a Training Loop

In PyTorch, building a training loop for image classification involves the following steps:

1. Creating an instance of the model used for the classification task, named *model0* in the code which is an instance of the pre-trained DenseNet201.
2. Defining the loss function and optimizer which has already been discussed in the previous section. They are **BCEWithLogitsLoss()** and **SGD** respectively.

3. Define the data loaders to create batches of training and test images that the model will train on during each iteration of the training loop. Each batch consists of 32 images and labels. The training loop consists of 15 epochs to be iterated on. An epoch is a one complete pass through the entire training dataset. During one epoch, the model processes every sample in the training dataset once, and updates its parameters based on the computed gradients from the loss function. After each pass through the dataset, the model's parameters are updated, and the process is repeated for a specified number of epochs. The number of epochs is a hyperparameter that determines how many times the model will process the entire dataset.
4. Define the training loop. In each iteration, the following will take place:
    a. Iterate over the data loaders to get a batch of images and labels
    b. Pass the images through the model to get predicted labels
    c. Compute the loss by comparing the predicted and true labels
    d. Compute the gradients of the loss with respect to the model's parameters
    e. Update the model's parameters using the optimizer
    f. Record the loss and any other metrics needed to be tracked, such as accuracy
    g. At the end of each epoch, run the validation dataset through the model and record the metrics on that set
5. Run the loop for a certain number of epochs, 15 in this case.

2.5 Train the Model on the Data and Make Predictions

During training, the pre-trained weights of the model are frozen except for the final linear layer which is a method recommended in this paper [19] for models trained on ImageNet architectures. In order to organize the code into smaller, more manageable units that can be reused repeatedly, it is divided into scripts. One script called *engine.py,* contains functions for training and test a PyTorch model. These functions are:

→ **`train_step()`**: Trains a PyTorch model for a single epoch. Turns a target PyTorch model to training mode and then runs through all of the required training steps (forward pass, loss calculation, optimizer step). Arguments passed into this function are:

- *model:* A PyTorch model to be trained.
- *dataloader*: A DataLoader instance for the model to be trained on.
- *loss_fn:* A PyTorch loss function to minimize.
- *optimizer:* A PyTorch optimizer to help minimize the loss function.
- *device:* A target device to compute on (e.g. "cuda" or "cpu").

This function returns a tuple of training loss and training accuracy metrics in the form (train_loss, train_accuracy). For example: (0.1112, 0.8743)

→ **`test_step()`:** Tests a PyTorch model for a single epoch. Turns a target PyTorch model to evaluation mode and then performs a forward pass on a testing dataset. Arguments passed into this function are:

- *model:* A PyTorch model to be tested.
- *dataloader:* A DataLoader instance for the model to be tested on.
- *loss_fn:* A PyTorch loss function to calculate loss on the test data.
- *device:* A target device to compute on (e.g. "cuda" or "cpu").

This function returns a tuple of testing loss and testing accuracy metrics in the form (test_loss, test_accuracy). For example: (0.0223, 0.8985)

→ **`train()`:** Trains and tests a PyTorch model by combining the two methods above. Passes a target PyTorch model through **`train_step()`** and **`test_step()`** functions for a number of epochs, training and testing the model in the same epoch loop. Calculates, prints and stores evaluation metrics throughout. Arguments passed into this function are:

- *model*: A PyTorch model to be trained and tested.
- *train_dataloader*: A DataLoader instance for the model to be trained on.
- *test_dataloader*: A DataLoader instance for the model to be tested on.
- *optimizer*: A PyTorch optimizer to help minimize the loss function.
- *loss_fn*: A PyTorch loss function to calculate loss on both datasets.
- *epochs*: An integer indicating how many epochs to train for.
- *device*: A target device to compute on (e.g. "cuda" or "cpu").

This function returns a dictionary of training and testing loss as well as training and testing accuracy metrics. Each metric has a value in a list for each epoch. In the form:

{train_loss: [...],

train_acc: [...],
test_loss: [...],
test_acc: [...]}

For example if training for epochs=2:

{train_loss: [2.0616, 1.0537],
train_acc: [0.3945, 0.4912],
test_loss: [1.2641, 1.5706],
test_acc: [0.3400, 0.2973]}

In the main Python Jupyter notebook file, the *engine* script which contains the `train()` method is imported and utilized to train the fine-tuned DenseNet201 model on the dataloaders for 15 epochs. This training was done on Google Colab [24] which is a free Jupyter notebook environment that runs entirely in the cloud. It supports many popular machine learning libraries including PyTorch which can be easily loaded in the notebook. More importantly, it offers GPU use for free which drastically accelerates the training times for models. The Tesla T4 GPU offered by Colab was utilized in training in which the total time taken for the model to train on the images is 666.934 seconds whereas on a CPU, training would've taken several hours.

2.5 Evaluate the Model

Another script used in this project is the *helper_functions.py.* It offers multiple helper functions, one of which is the `plot_loss_curves()` which as the name suggests plots the training and testing curves of a results dictionary. It makes use of the open source library *matplotlib* for visualizing the model performance. The only argument to this function is a dictionary containing lists of value. The `train()` function discussed in the previous section will return a dictionary of training accuracy and loss, testing accuracy and loss for each epoch in which this returned dictionary will be the input argument to `plot_loss_curves()` function where it will use the values in dictionary to plot the required curves.

After training the model for 15 epochs, the results can be seen in the following figure:

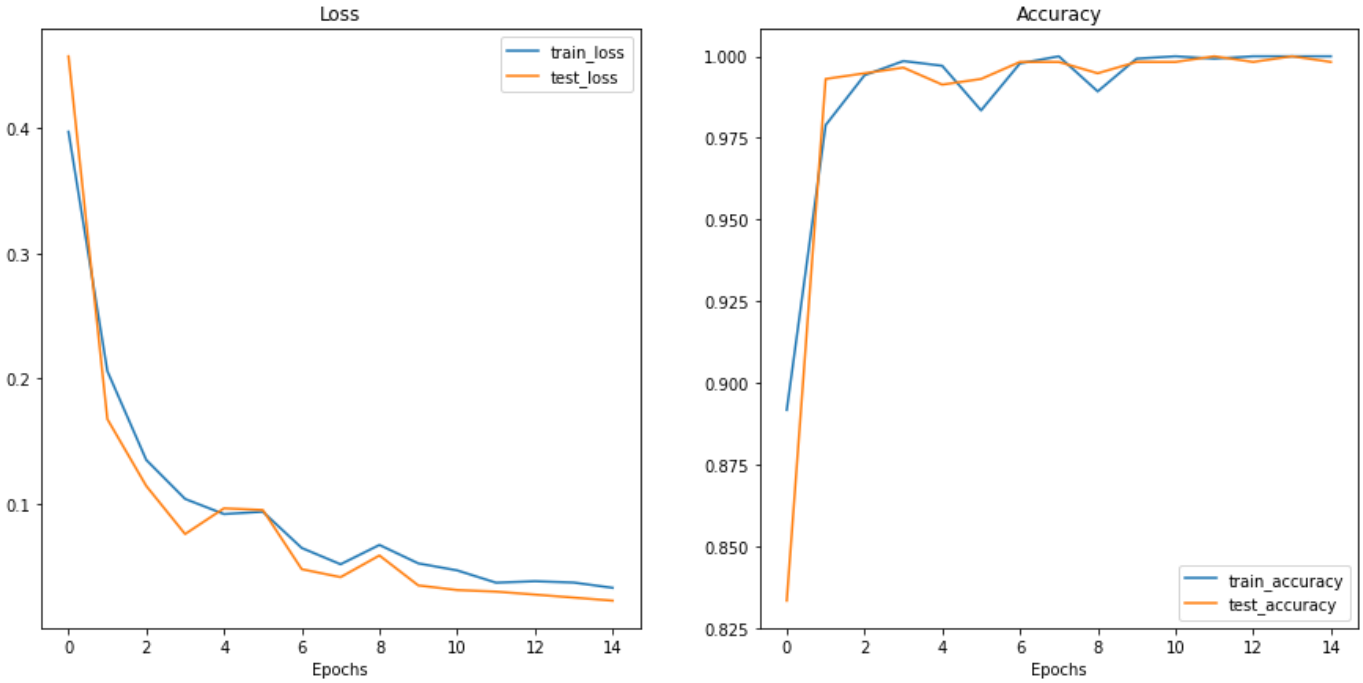


Figure 6: Results of accuracy and loss during training epochs on dataset A. Visualization is done by Matplotlib.

As observed in figure 6, exceptional results were achieved with 100% accuracy on the train set and 99.83% accuracy on the test set. Both the train and test losses are heading in the right direction. However, the accuracy values are both trending upwards. This shows the power of transfer learning where a pre-trained model leads to great results with a relatively small amount of data in a relatively short training time. A function called `pred_and_plot_image()` is created to take an image path as an argument, plot that image, and predict on it using a trained model. This function also utilizes *matplotlib* for visualization. Here's some of the outputs this function has to offer:

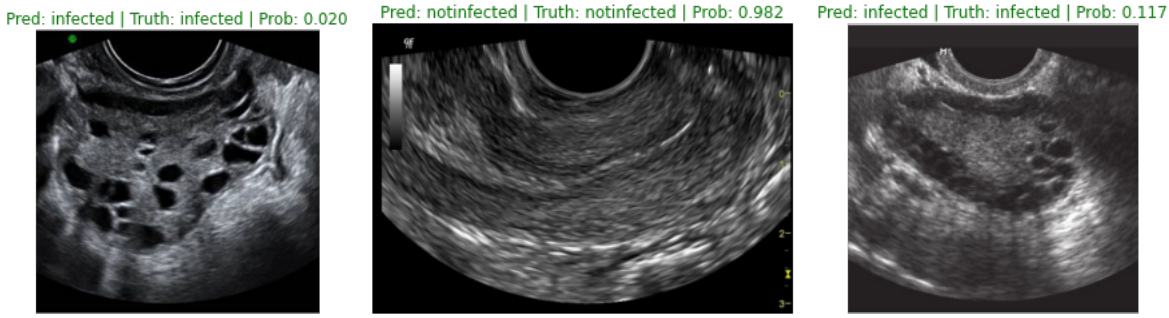


Figure 7: using pred_and_plot_image() to visualize and predict on images. Visualization is done by Matplotlib.

As seen in figure 7, the predictions are all correct since the caption is in green and as shown by the ground truth compared to the prediction in the caption. Also, the model

seems to be confident when predicting on these images judging by the probabilities which are either close to 0 for infected cases or to 1 for not infected cases. This means that the model is not just predicting randomly.

In order to plot the confusion matrix which is very important for evaluation because it provides a clear and concise summary of the performance of a classifier, two open source libraries were used. These libraries are *torchmetrics* and *mlxtend.* Here's the confusion matrix obtained when predicting on the entire test set:

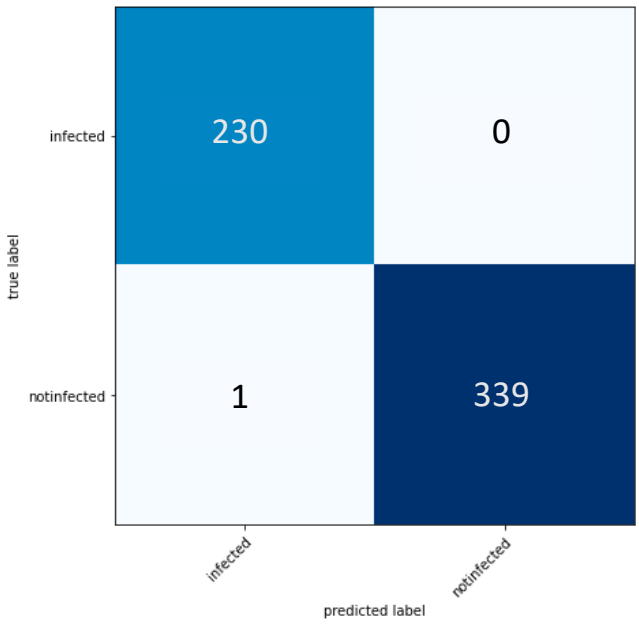


Figure 8: Confusion matrix of test set in Dataset A.

As observed from the confusion matrix, the number of true positives and true negatives is perfect bar 1 image. From figure 8, we can derive the precision, recall, and F1-score for the infected class. Table 1 depicts these three values:

	Precision (%)	Recall (%)	F1-score (%)
Dataset A	99.57	100	99.78

Table 1: Precision, recall, and F1-score for the infected class (dataset A)

The model can be saved for later use and deployment. This is done using `torch.save()` and can be reloaded using `torch.load()` in order to utilize the model elsewhere.

However, since the results are suspiciously perfect, and after further inspection. It turned out that dataset A that was used to train the fine-tuned DenseNet201 model is highly erroneous and misleading. By consulting a professional specialist in this medical field, it was discovered that the images in `notinfected` class which is supposed to represent the healthy ovaries having no sign of PCOS are in fact not images of ovaries at all. Rather, they are ultrasound images of uterus which completely falsifies this dataset. Therefore, another reliable dataset should be obtained.

Chapter 3 Different Data

3.1 Collecting New Data

Another publicly available ovarian ultrasound dataset is published by Telkom University Dataverse [25]. This new dataset is allegedly annotated by specialist doctors, and will be referred to as Dataset B. It consists of 54 ultrasound images 14 of which are classified as PCOS and the rest are normal.

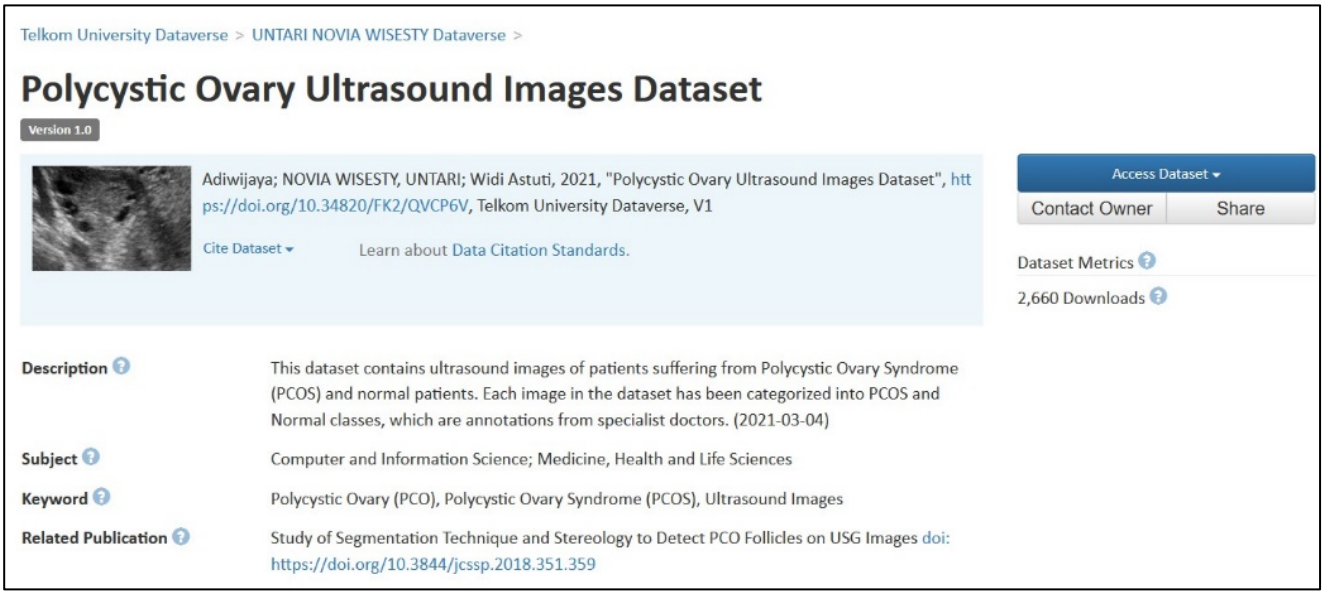


Figure 9: Screenshot of the publicly available PCOS dataset by Telkom University Dataverse referred to as Dataset B.

Figure 10 shows a sample from this dataset.

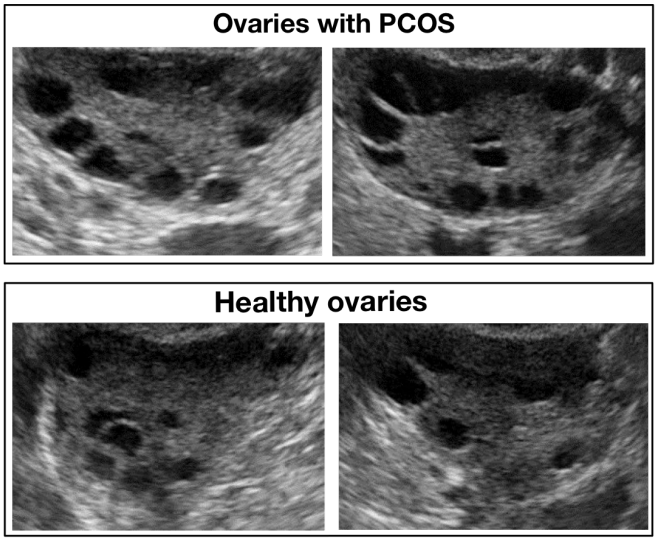


Figure 10: Sample of infected and healthy ovaries in Dataset B.

3.2 Training on the New Data

This dataset poses two new challenges that need to be addressed:

1. The dataset is small: Tackled through data augmentation.
2. The dataset is imbalanced: Increasing the data points of the minority class when artificially creating new data points through data augmentation.To keep the two experiments on both datasets A and B fair and due to this dataset being relatively small in numbers, data augmentation techniques such as random horizontal flipping, random vertical flipping, random brightness alteration, and random rotation are used to artificially increase the number of ultrasound images using the Python library `*imgaug*` as this will help the training process massively. After training the same model again for 15 epochs on this new dataset and keeping all the hyperparameters fixed, the following results and confusion matrix are obtained:

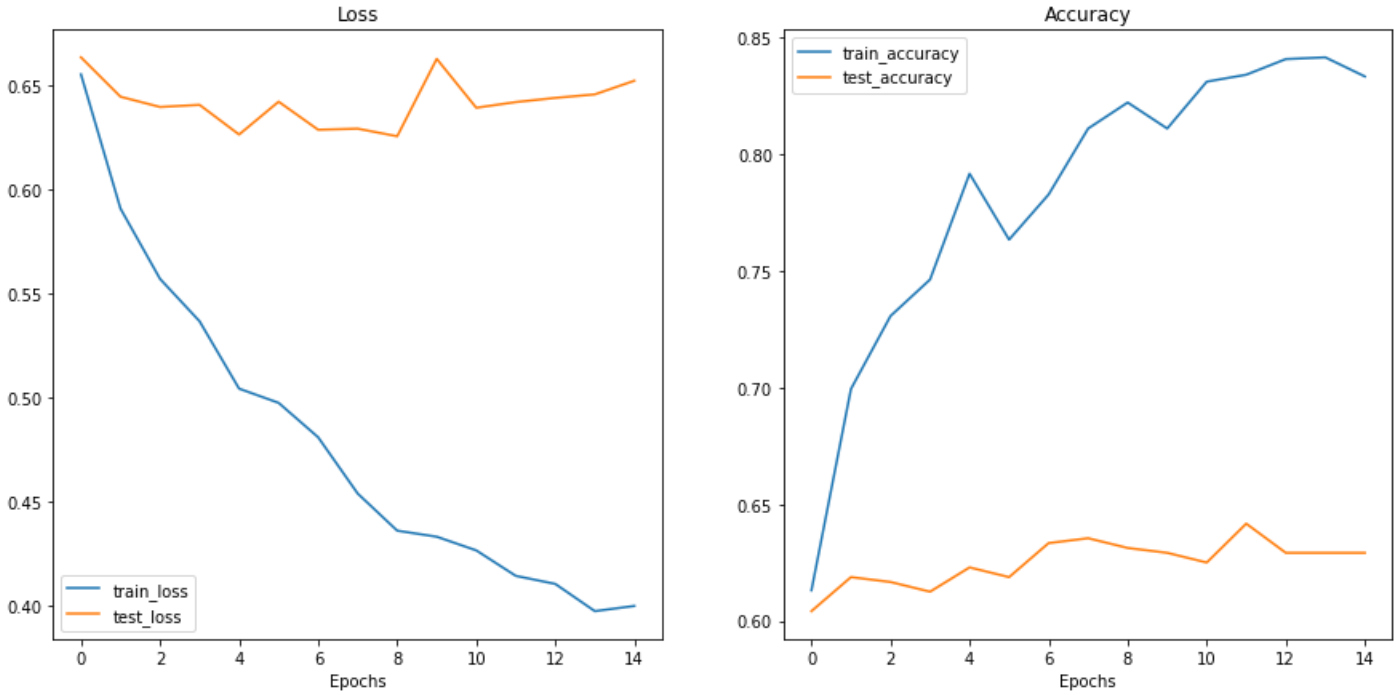


Figure 11: Results of accuracy and loss during training epochs on dataset B. Visualization is done by Matplotlib.

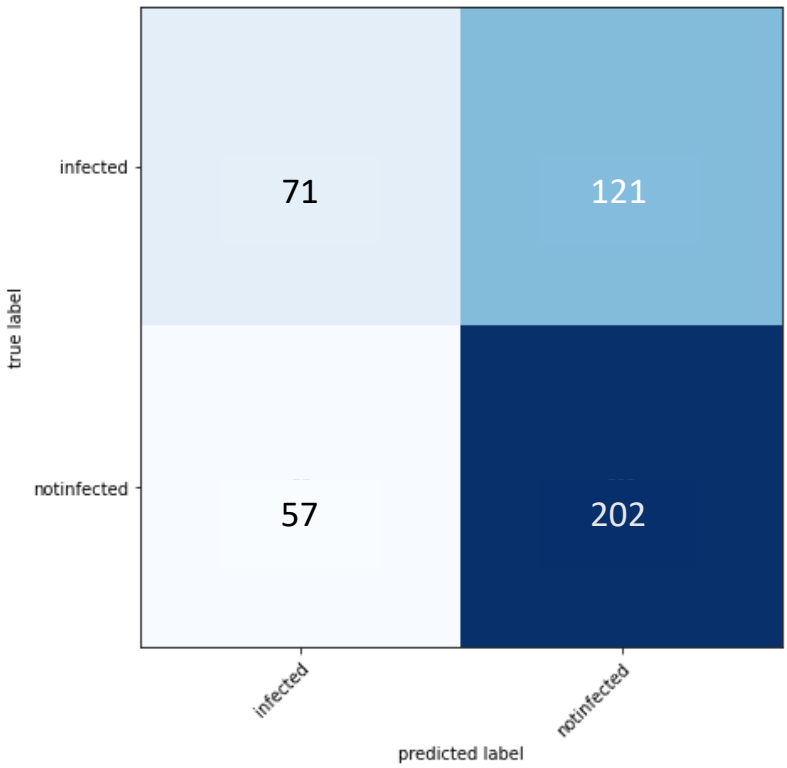


Figure 12: Confusion matrix of test set in Dataset A.

As figure 11 shows, the train accuracy increases steadily until it reaches 83.33% while the test accuracy remains relatively unchanged at 62.92%. This means that the model did not learn any features and is not able to generalize to the test images since the test accuracy remained relatively unchanged. The same can be observed for train and test loss where the train loss steadily decreases but not for the test loss. This is a clear indication of overfitting and the inability of the model to generalize well on unseen data. The poor model performance is also confirmed by the confusion matrix where the number of true positives and true negatives is unsatisfactory. Table 2 exhibits the precision, recall, and F1-score for the infected class of the test set in dataset B:

	Precision (%)	Recall (%)	F1-score (%)
Dataset B	55.47	36.98	44.38

Table 2: Precision, recall, and F1-score for the infected class (dataset B)

Here are some images being predicted on and plotted using the previously mentioned function `pred_and_plot_image()`:

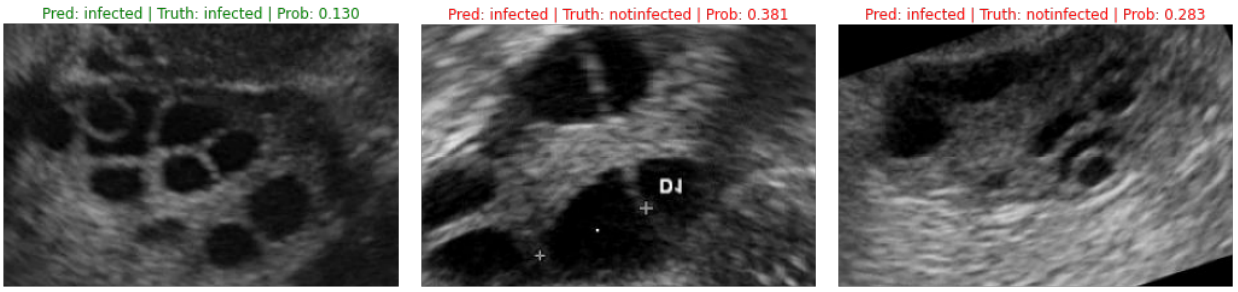


Figure 13: using pred_and_plot_image() to visualize and predict on images in dataset B. Visualization is done by Matplotlib. Note: If the caption is in green, it means that the prediction is correct. If the text is in red, it means that the prediction is incorrect

As shown by the previous 2 experiments, when the model was presented with a true dataset rather than a false one, it struggled to generalize well on it as the task got significantly harder and it overfitted on the training data. Thus, data quality and real data is of utmost importance when training deep learning models, especially in the medical and health fields. If the data is flawed or biased, the model will likely produce inaccurate or unreliable results even if the results appear to be satisfactory. In the medical and health field, this can have serious consequences as it can lead to incorrect diagnoses or treatment recommendations, potentially causing harm to patients. It is therefore essential to ensure that the data used to train these models is of the highest quality and accurately represents the population it is intended to serve.

3.3 More experimentations…

Achieving bad results on dataset B doesn't mean we have to stop there. Experimentation in the field of deep learning is crucial because it allows for the testing and fine-tuning of different model architectures, hyperparameters, and data pre-processing techniques. This experimentation process helps to identify the most effective combination of these elements for a given task, and can lead to significant improvements in model performance. Therefore, **three modifications** are carried out in this third experiment. First, the pre-processing in terms of data augmentation on the images has been slightly adjusted where vertical flipping has been neglected, random rotations' angles has been reduced, the brightness alteration has also been slightly reduced, and cropping has been dropped. This is all done to hopefully better represent the target classes when artificially creating new images for the classes in question. Secondly, the number of data points has

been reduced in both the training and test sets to almost half as compared to the dataset used in the second experiment (dataset B). Finally, the way the first layer is fine-tuned now is different. Previously, the method to adjust the very first layer was to replace the entire layer with another layer that accepts 1 color-channel input. The method is shown in the figure below:

```
[24]  print(model0.features.conv0) # We need to modify this                    Python
...   Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

      # Adjusting input channels from 3 to 1 to match grayscale images color channels
[25]  model0.features.conv0 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)  Python
```

Figure 14: Adjusting the very first layer of the DenseNet model named model0.

The problem with this method is that by replacing the entire first layer, some of the features learned by the model in the first layer are also being eliminated and replaced with a new layer that has the same number of parameters, but these parameters are completely random as opposed to the ones removed. After coming across this blog post [26], a new method for adjusting the first layer is demonstrated without discarding the entirety of the first layer, and with only adjusting the number of channels in the first layer, the same goal of accepting 1 color-channel images was achieved more efficiently. After these changes has been carried out, the model is trained with the same remaining hyperparameters as before. Much better results have been obtained, with accuracy of roughly 70% on the test set and the following confusion matrix:

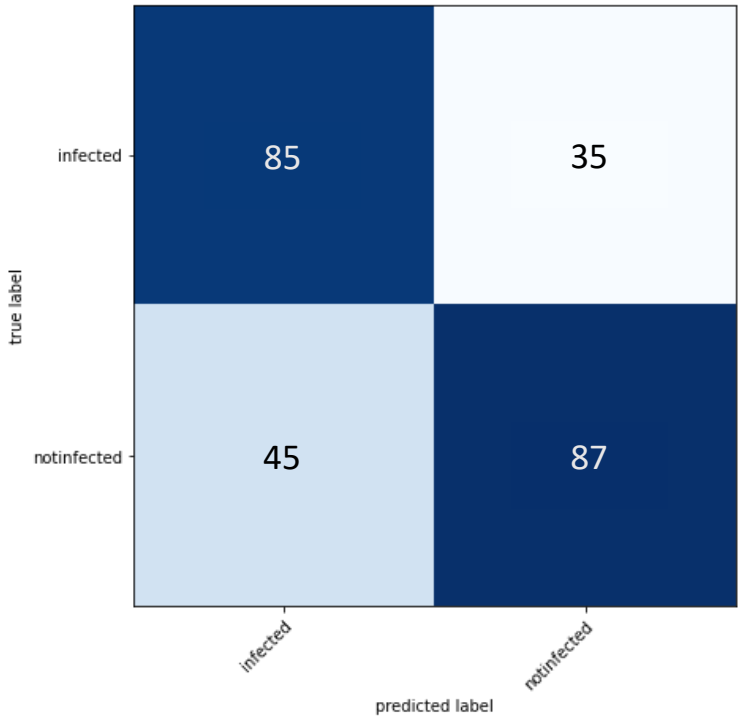


Figure 15: Confusion matrix of test set in the new experiment.

Table 3 shows the precision, recall, and F1-score for the infected class of the test set in this new dataset:

	Precision (%)	Recall (%)	F1-score (%)
Modified Dataset B	65.4	70.8	68.0

Table 3: Precision, recall, and F1-score for the infected class in the new dataset

As observed from the confusion matrix above, much improvement has been achieved and the results are somewhat satisfactory.

Chapter 4 Model Improvement with Experimentations

Improving the accuracy and efficiency of a model is important for several reasons:

1. **Enhanced Performance:** Higher accuracy means the model is more reliable and makes fewer errors in its predictions. This is particularly crucial in applications like medical diagnosis, where incorrect predictions can have serious consequences. Improved accuracy ensures that the model provides trustworthy results, leading to better decision-making and patient care.
2. **Resource Optimization:** Improving efficiency reduces the computational resources required to execute the model. This translates to faster inference times, reduced memory usage, and lower energy consumption. Efficient models are more suitable for real-time applications, large-scale deployments, or scenarios with limited computational capabilities.
3. **Cost-Effectiveness:** Efficient models help reduce the costs associated with deploying and maintaining the system. By optimizing resource usage, organizations can scale their infrastructure more effectively, saving on hardware expenses and operational costs. Moreover, increased efficiency allows for faster processing, leading to higher throughput and productivity.
4. **Scalability:** A highly accurate and efficient model is more scalable, capable of handling larger datasets and increasing prediction throughput. Scalability is essential for applications that deal with massive amounts of data, such as analyzing medical records or monitoring a large population for PCOS. By improving scalability, the model can handle increased workloads without sacrificing performance.
5. **Deployment Feasibility:** Models with good accuracy and efficiency are more likely to be successfully deployed in real-world settings. When a model meets the required performance criteria, it becomes more feasible to integrate it into existing systems, such as electronic health record systems or ultrasound machines. The

practical application of the model becomes viable, benefiting patients and healthcare providers.

The above requirements could be possibly achieved by trying out different variations of the model used, different methods of training, or entirely different model architectures. One of the criteria for our deployed model is to be fast. In our case, fast meaning as close to real-time (30 FPS - 0.033 sec) as possible but without sacrificing model accuracy.

4.1 DenseNet201 vs. DenseNet121d

Ideally, the model should perform predictions as fast as possible and with the most accuracy achievable. In the previous sections, all training and evaluation was carried out using DenseNet201 architecture where transfer learning was utilized to import the model along with its weights and parameters. However, during the 15 training epochs, **only the classifier was being trained** while the rest of model parameters were frozen. Now, a different smaller variation of the DenseNet architecture family will be imported and trained on the same available PCOS dataset. This variation is the DenseNet121 which is a much smaller model than DenseNet201. DenseNe201 has roughly 18 million parameters.

```
==========================================================
Total params: 18,088,577
Trainable params: 1,921
Non-trainable params: 18,086,656
Total mult-adds (G): 134.75
==========================================================
Input size (MB): 6.42
Forward/backward pass size (MB): 8334.84
Params size (MB): 72.35
Estimated Total Size (MB): 8413.61
==========================================================
```

Figure 16: DenseNet201 statistics. Only 1,912 parameters (classifier) are trainable. Rest are frozen.

As we on the other hand, DenseNet121d has roughly 7 million parameters, which makes it almost only 38% the size of DenseNet201. The smaller size will lead to faster inferences and more efficient use of the hardware. Though, this increase in efficiency should not be at the expense of the accuracy of the model. Building on this, training DenseNet121d on the latest modified PCOS dataset will be initiated with one major difference this time. Only the architecture of DenseNet121d will be imported and the entire model will be trained rather than only the classifier. Training will be initiated with **random weights**.

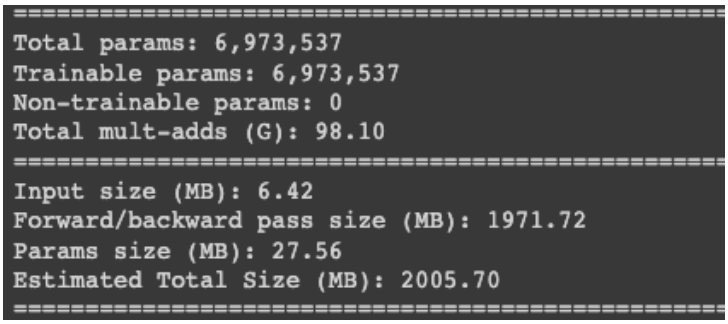


Figure 17: DenseNet121 statistics. The entire model is trainable meaning no parameters are frozen.

4.2 Training DenseNet121d from scratch

As training was conducted before, the same pre-processing pipeline on the dataset was carried out along with choosing the same loss function **(BCEWithLogitsLoss)** and the same optimizer **(SGD).** A slight difference is the learning rate chosen for this particular training process. It is set to 0.001 rather than 0.01 that was set previously. **Engine.py** script that contains the **train** function was used to train this model for **217 epochs**. Note that this training process is much longer than what was done previously due to the fact that this architecture is not pre-trained and was trained from randomly initiated weights. By starting with pre-trained weights, the model already has a good initialization, and the learned features can be fine-tuned to a new dataset. This process is referred to as transfer learning. The pre-trained weights act as a useful starting point, capturing low-level features like edges, corners, and textures, which are applicable to a wide range of images. As a result, the model requires fewer iterations to converge and can achieve good performance faster. Random initialization starts with no prior knowledge of the data, so the model needs to learn relevant features from scratch. This process takes much more time hence the 217 epochs and is **not** referred to as transfer learning as it is not leveraging the learned parameters of an existing model. This is instead referred to as training from scratch from randomly initiated weights.

The following are the results and confusion matrix obtained after training DenseNet121d:

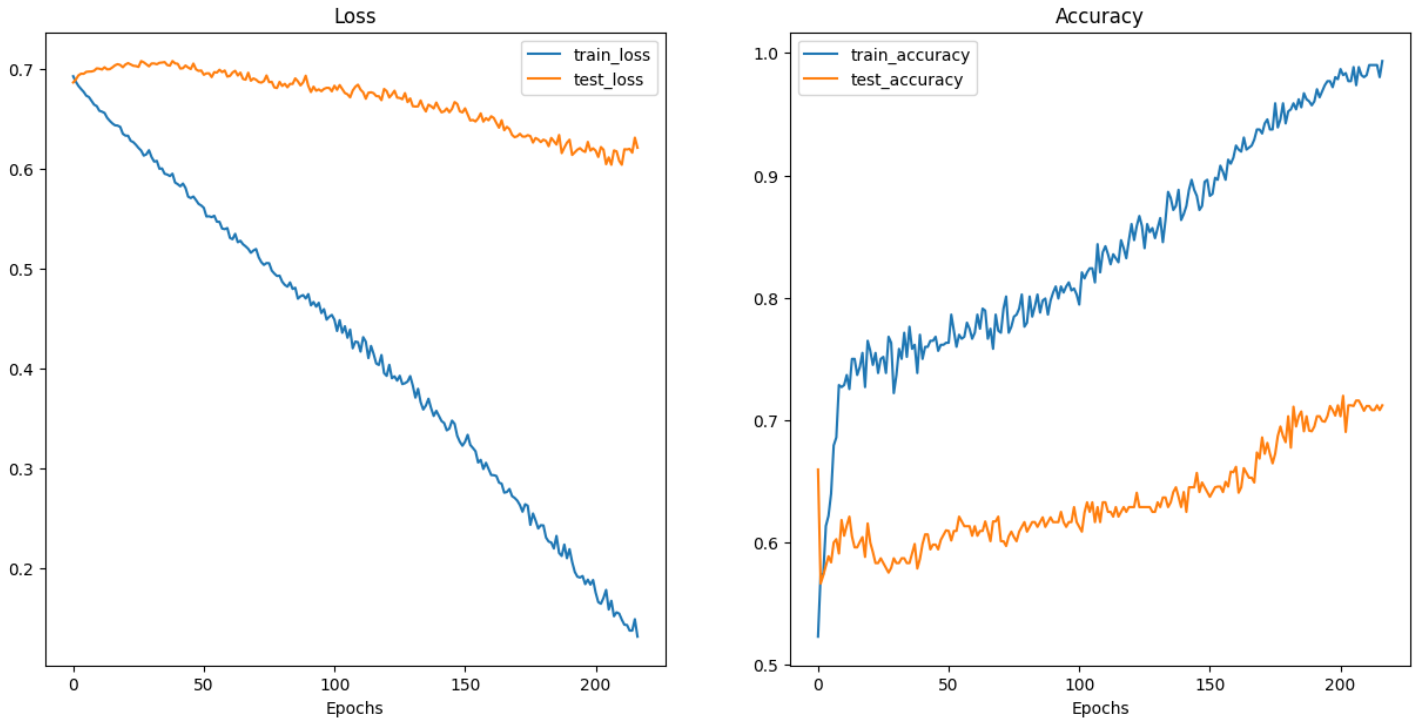


Figure 18: Results of accuracy and loss during training epochs of DenseNet121d. Visualization is done by Matplotlib

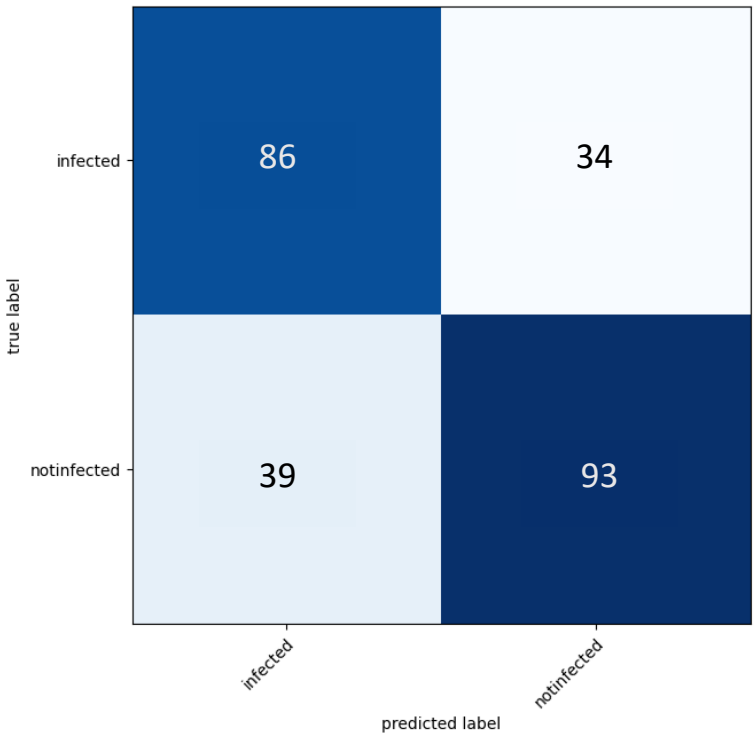


Figure 19: Confusion matrix on the test set using DenseNet121d

Table 4 shows the precision, recall, and F1-score for the infected class of the test set obtained using DenseNet121d:

	Precision (%)	Recall (%)	F1-score (%)
Modified Dataset B	68.8	71.1	70

Table 4: Precision, recall, and F1-score for the infected class obtained using DenseNet121d

As figure 16 shows, after 217 epochs, training accuracy reached 99% and testing accuracy reached roughly 72% as opposed to 70% achieved previously. The confusion matrix also confirms this improvement in performance as more true positives and true negatives were correctly predicted, specially in the case of true negatives where the correct predictions is raised from 87 (figure 15) to 93 in this case. Finally, table 4 shows improvements in all 3 metrics over table 3. Therefore, there are improvements in both accuracy since higher accuracy was obtained and in efficiency since the model is considerably smaller than DenseNet201.

But by what margin is this model exactly more efficient?

4.3 Difference in Efficiency

To find out, a function called `pred_and_store()` is used to iterate over each of the test dataset images one by one and perform a prediction. Each of the predictions will be timed and the results will be stored in a list of dictionaries where each element in the list is a single prediction and each single prediction is a dictionary. The predictions are timed one by one rather than by batch because when the model is deployed, it will be making a prediction one image at a time. As in, doctor/patient uploads a photo and the model predicts on that single image. After collecting the test images (paths) into a list, the `pred_and_store()` is built the following way:

1. Takes a list of paths (test images), a trained PyTorch model, a series of transforms (for image pre-processing), a list of target class names and a target device that will perform the prediction (CPU, GPU…)
2. Create an empty list to store prediction dictionaries. This is what will be returned by the function. A list which will contain a dictionary for each prediction.
3. Loop through the input list of test image paths (steps 4-14 will happen inside the loop).

4. Create an empty dictionary for each iteration in the loop to store prediction values per test image path.
5. Store the image path and class label which can be inferred from the path inside the dictionary.
6. Start the prediction timer using Python's `timeit.default_timer()`
7. Open the image using `PIL.Image.open(path)`.
8. Transform (pre-process) the image so it's capable of being used with the model as well as add a batch dimension and send it to the target device.
9. Prepare the model for inference by sending it to the target device and turning on `eval()` mode.
10. Turn on `torch.inference_mode()`, pass the transformed image to the model and calculate the prediction probability using `torch.sigmoid()` and the target label using `torch.round()`.
11. Add the prediction probability and prediction class to the prediction dictionary created in step 4. Also make sure the prediction probability is on the CPU so it can be used with non-GPU libraries such as NumPy and Pandas for later inspection.
12. End the prediction timer started in step 6 and add the time to the prediction dictionary created in step 4.
13. See if the predicted class matches the ground truth class from step 5 and add the result to the prediction dictionary created in step 4.
14. Append the updated prediction to the list of predictions created in step 2.
15. Return the list of prediction dictionaries.

Now that we defined the function, we can use it with any PyTorch model of choice. Starting with using `pred_and_store()` in making predictions and timing them with DenseNet201:

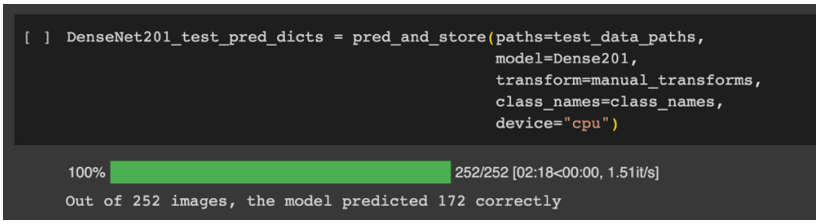


Figure 20: pred_and_store() used with the trained DenseNet201 model

Figure 18 shows that the model predicted 172 images out of 252 correctly. This is consistent with the confusion matrix shown in figure 15 (page 31) as the same model was used to predict on the same test dataset. 172 consists of the number of true positives + true negatives. Now that a list of dictionaries of all the necessary prediction values is obtained, Pandas library can be used to convert the this list into a Pandas Dataframe to a structured format for further analysis such as average time for predictions and how many wrong predictions the model got. After turning the list of dictionaries that contains the prediction information of DenseNet201 into a Dataframe, we obtain the following useful format of the data: (Note, only the last 5 rows of the Dataframe are printed here)

```
import pandas as pd
DenseNet201_test_pred_df = pd.DataFrame(DenseNet201_test_pred_dicts)
DenseNet201_test_pred_df.tail()
```

	image_path	class_name	pred_prob	pred_class	time_for_pred	correct
247	/content/driveG/MyDrive/GP1/After_Research_pap...	notinfected	0.7081	notinfected	0.6507	True
248	/content/driveG/MyDrive/GP1/After_Research_pap...	notinfected	0.8859	notinfected	0.6852	True
249	/content/driveG/MyDrive/GP1/After_Research_pap...	notinfected	0.6413	notinfected	0.7197	True
250	/content/driveG/MyDrive/GP1/After_Research_pap...	notinfected	0.6130	notinfected	0.5847	True
251	/content/driveG/MyDrive/GP1/After_Research_pap...	notinfected	0.6026	notinfected	0.6643	True

Figure 21: Dataframe of predictions obtained from DenseNet20

Since the main focus is on model efficiency which is closely tied to prediction time, the average prediction time is calculated from the Dataframe:

```
# Find the average time per prediction
Dense201_average_time_per_pred = round(DenseNet201_test_pred_df.time_for_pred.mean(), 4)
print(f"DenseNet201 average time per prediction: {Dense201_average_time_per_pred} seconds")
```

```
DenseNet201 average time per prediction: 0.5476 seconds
```

Figure 22: Average prediction time using DenseNet201

Thus, the average prediction time using DenseNet201 is 0.5476 seconds.

Now repeating the same steps again but for DenseNet121d model yields the following:

```
# Find the average time per prediction
Dense121_average_time_per_pred = round(DenseNet121_test_pred_df.time_for_pred.mean(), 4)
print(f"DenseNet121 average time per prediction: {Dense121_average_time_per_pred} seconds")
```

```
DenseNet121 average time per prediction: 0.25 seconds
```

Figure 23: Average prediction time using DenseNet121d

As shown, the average prediction time using DenseNet121d is 0.25 seconds. More than twice as fast as DenseNet201 with having better accuracy simultaneously and while being a much smaller model. Thus, the two goals of accuracy and efficiency are achieved.

Perhaps this can be taken further with an even smaller model? This will be discussed in detail within the next section.

4.4 Building a model from scratch (IustNet)

To build a small model from scratch, the TinyVGG architecture obtained from the CNN explainer website [27] shown in the following figure is replicated with a slight modification and used for training.

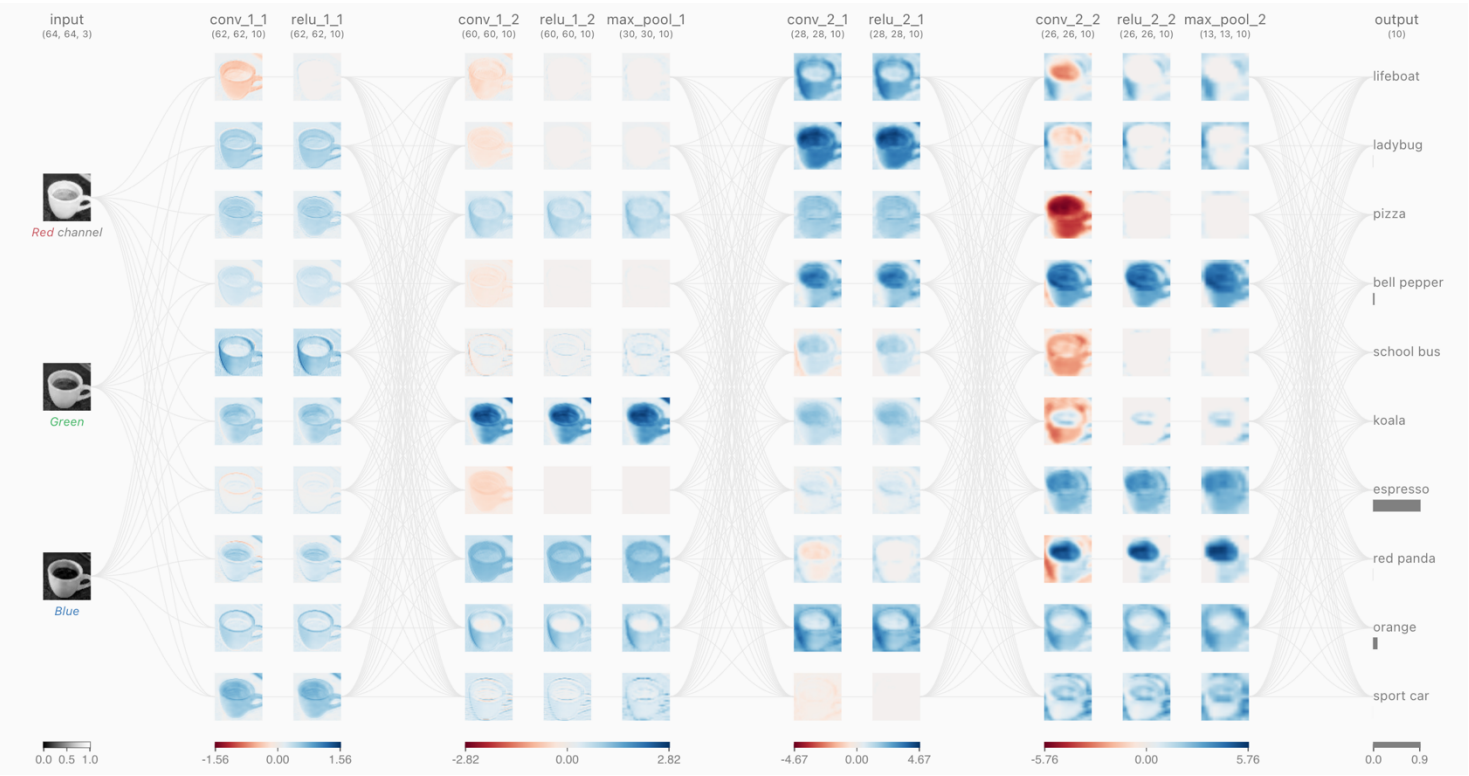


Figure 24: Reference architecture used for IustNet

Using the above model architecture, IustNet will be built and trained using PyTorch. However, the figure shows that the number of hidden units in the model is 10. IustNet will be using 45 hidden units instead with ReLU as activation function. There are two main blocks in this architecture. Each one starting with a convolution layer and ending with

max_pool layer. Finally, there's the classifier which flattens the image into a one-dimensional vector and uses a fully connected network to make predictions.

IustNet is much smaller than DenseNet121d and DenseNet201 as it only has 197,191 total parameters as opposed to 7 million and 18 million respectively.

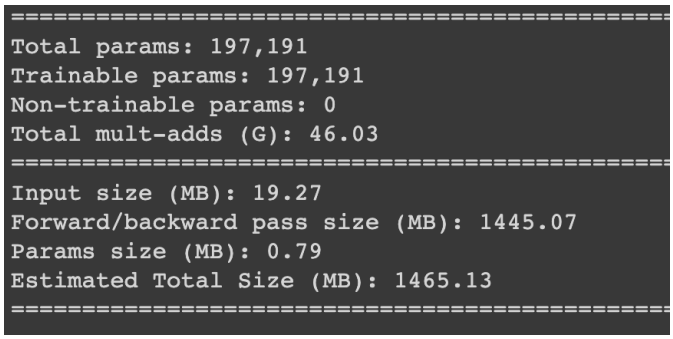


Figure 25: IustNet statistics. Entire model is trainable.

Similar to previous work, **(BCEWithLogitsLoss)** and **(SGD)** are used as loss function and optimizer. Though, there's a change to the pre-processing of images inputted to the model. This time, ultrasound images will not be turned into grayscale during the preprocessing pipeline and will be left as images with 3 color channels. After training for 45 epochs, the model achieved an accuracy of 88% on the training images and 77% on the test dataset. The following is the obtained confusion matrix from IustNet predicting on the test set:

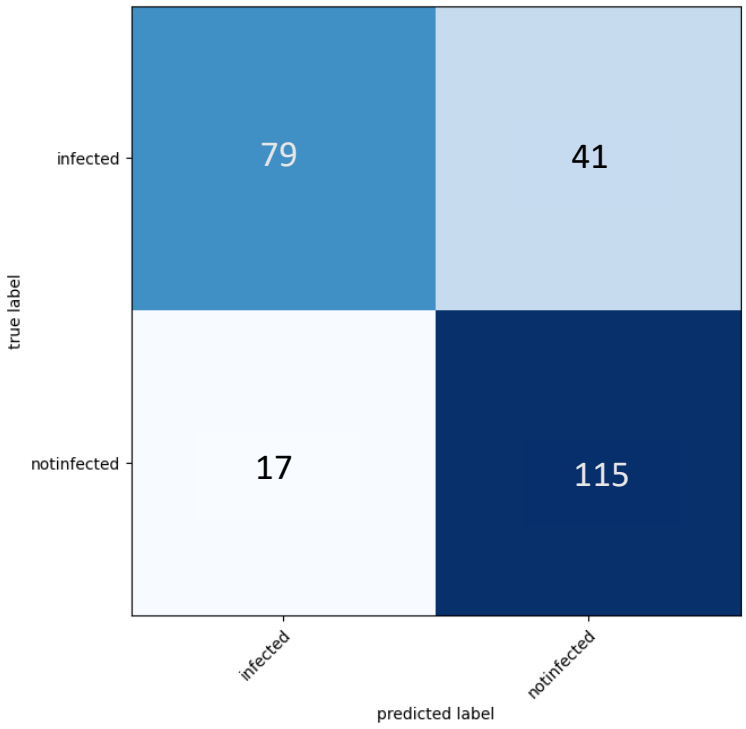


Figure 26: Confusion matrix on the test set using IustNet

Even though on the surface of it seem might seem like this model performs better than DenseNet121d and DenseNet201 due to higher testing accuracy. That is not the case because the actual improvement in predictions is only on the `not infected` class and the increase in accuracy is due to higher bias towards the `not infected`. By paying attention to the number of true positives (infected class), it has actually fallen down from 86 (DenseNet121d) to 79 in this case. Of course, it is of extreme importance to correctly classify the `infected` class cases due to how sensitive the subject matter is. Now checking how much faster this model by using `pred_and_store()` and then converting the returned list of dictionaries into a Dataframe like discussed previously, it is discovered that the average time for predictions using IustNet is 0.0958 seconds which is closer to real-time and is almost 0.15 seconds faster than DenseNet121d.

4.5 Choosing the most suitable model

After calculating precision, recall for the infected class on test set as well as the F1-score, putting everything together, we have:

	DenseNet201	DenseNet121d	IustNet
Transfer Learning	Yes	No	No
# Parameters	18 mil	7 mil	192 k
Avg Prediction Time	0.548 sec	0.25 sec	0.0958 sec
Accuracy	70 %	72 %	77 %
Precision	65.4 %	68 %	82 %
Recall	70.8 %	71 %	65 %
F1-score	68 %	70 %	72%

Table 5: Statistics and metrics of all suggested models

Judging from the above table, it is clear that DenseNet201 is perhaps the worst choice out of the 3 models since it is the largest model which means least efficiency and more prediction time on average as well as worse statistics in almost all the metrics than the other two models. However, when comparing DenseNet121d and IustNet, it might seem that IustNet may be the better choice out of the two. But, regarding medical diagnosis, the recall metric is the most important metric because recall is TP / (TP + FN). This means that recall captures the cases where the model predicts actual infected cases as not infected, this may lead to disastrous consequences as the model might lead the doctor to classify an infected case as not infected which could prevent the patient from carrying out the necessary treatment for the disease. Even though DenseNet121d takes more than double the time to make a prediction on average and has less scores on other metrics than IustNet, the importance of the recall metric outweighs these factors. Thus, **DenseNet121d will be the model of choice for deployment**.

NOTE: Precision captures the cases where the model incorrectly predicts not infected cases as infected. This is much less serious than what the recall captures in this case since false infected predictions can be easily discovered to be incorrect later when starting treatment or carrying out another diagnosis.

Chapter 5 Model Deployment

Machine learning model deployment is the process of making a machine learning (deep learning) model accessible to someone or something else. For example, someone else in the case of the model in this thesis could be a doctor/patient uploading an ultrasound image using their PC/smartphone so that the trained PCOS detector could classify it into infected or not infected. Something else could be another program, app, or even a different model that interacts with this particular model. **Deploying a model is as important as training one** because there's no point in training a machine learning model and improving it if it will not be accessible by another party. PCOS detector could be employed as a decision support system for doctors. Thus, it should be easily accessible from multiple devices and at all times. Based on this, model deployment will be divided into two parts:

1. **Web application development**
2. **Cloud-based hosting**

5.1 Web application development

A web application is developed to integrate the trained PCOS model and make it accessible for everyone to use. The choice of developing a web application is made to several advantages it can provide such as:

1. **Cross-platform accessibility:** Web applications can be accessed from various devices and operating systems, including smartphones, tablets, PCs, and different web browsers. Users can access the application regardless of their device or platform, providing wider accessibility and convenience.
2. **No installation required:** Unlike mobile or desktop applications, web applications do not require installation on the user's device. Users can simply access the application through a web browser, eliminating the need for downloading and installing separate software. This ease of access reduces barriers to entry and makes it simpler for users to start using the application.
3. **Centralized updates and maintenance:** With a web application, updates and maintenance can be performed centrally on the server. When a new version of the

machine learning model or any other changes are made, they can be deployed on the server, and all users accessing the web application will immediately benefit from the updates. This centralized management simplifies the maintenance process and ensures consistent user experience across different devices.

4. **Easier deployment and scalability:** Deploying a web application on a server is often easier and more scalable compared to deploying models on individual devices. With a web application, cloud infrastructure and services can be leveraged to handle high traffic and scale as needed. This scalability allows the application to handle a larger number of concurrent users and adapt to varying demand without compromising performance.
5. **Simplified user interactions:** Web applications can provide a user-friendly interface that guides users through the process of uploading medical images and receiving predictions. They can offer features such as drag-and-drop functionality, progress indicators, and interactive displays of results. This simplified user experience enhances usability and makes the application more accessible to a wider range of users.
6. **Integration with other web services:** Web applications can easily integrate with other web services and APIs, such as electronic health record systems, medical databases, or authentication services. This integration enables seamless data exchange and interoperability, enhancing the functionality and usefulness of the application.
7. **Analytics and tracking:** Web applications can leverage analytics tools to gather data on user interactions, usage patterns, and performance metrics. This data can be valuable for monitoring the application's performance, understanding user behavior, and making informed decisions for further improvements.

To develop this web application, **Gradio** [28] is used. It is a Python library that provides a simple and intuitive interface for quickly creating custom user interfaces (UI) for machine learning models. It allows developers and data scientists to build interactive UIs to showcase their models, enabling users to input data and receive real-time predictions or visualizations. Gradio supports various types of input and output, such as text, images, audio, and video, making it versatile for different types of models. It offers a range of UI

components, including text boxes, sliders, dropdowns, and image uploaders, which can be easily configured to collect user input. Gradio also supports live updates, allowing predictions or visualizations to be displayed dynamically as users interact with the UI.

Using Gradio, developers can create UIs for machine learning models without requiring extensive knowledge of web development or UI frameworks. Gradio abstracts away the complexities of building a web application, providing a simplified interface to connect models with user-friendly interfaces. Additionally, Gradio is compatible with PyTorch, allowing seamless integration of trained PyTorch models. It also supports deployment to various platforms such as cloud-based hosting platforms like Hugging Face Spaces which will be used for hosting the web application.

To create the PCOS detector UI with Gradio, a function is needed to map the inputs to the outputs. The function is made to predict on a single image using DenseNet121 model for reasons discussed in the previous section. This suggested function does the following:

1. Takes an ultrasound image as input
2. Pre-processes it
3. Makes a prediction using DenseNet121
4. Returns the prediction, prediction probability, and the time it took for the prediction

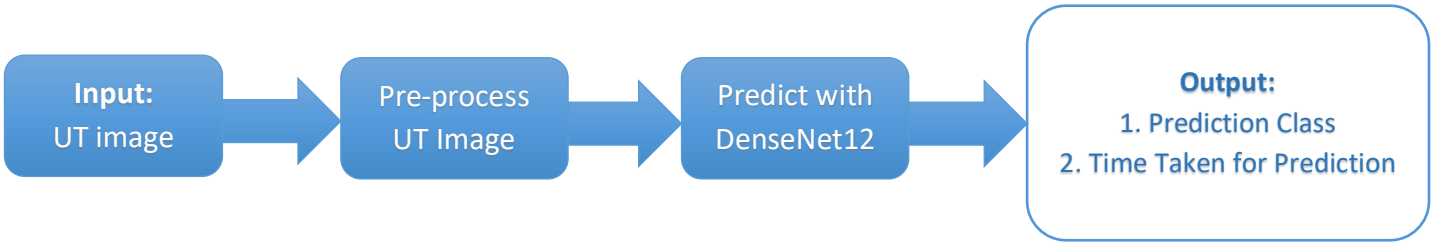


Figure 27: Block diagram for the predict() function

This function will be called `predict()` and it will be set to the `fn` parameter of the Gradio interface function. A list of examples will also be created which will include some UT images that will be used as examples to showcase the user how the interface works without the user having to upload any images of their own, `examples` is an optional parameter in the Gradio interface function. The `gradio.Interface()` class the following parameters:

- `fn`: a pyhton function that maps inputs to outputs. in this case, the `predict()` function is used

- `inputs`: the input to our interface, such as image using `gradio.image()` or `image`
- `Outputs`: the output of our interface once the `inputs` have gone through the `fn`, such as a label using `gradio.Label()` for the PCOS detector class prediction or a number using `gradio.Number()` for the PCOS detector prediction time
- `examples`: a list of examples to showcase how the model works
- `title`: a string title of the UI
- `description`: a string to describe the model briefly

Once the UI is created using an instance of `gr.Interface()`, it can be displayed using `demo.launch()` command.

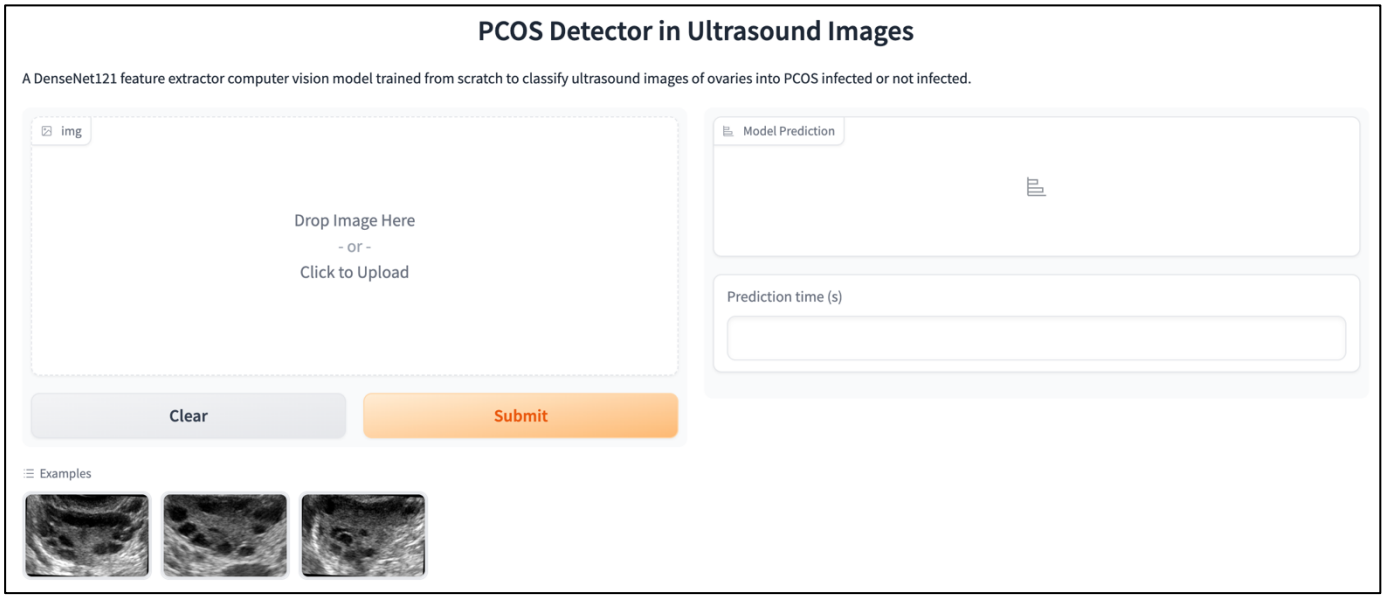


Figure 28: UI for the PCOS detector web application built using Gradio. Note: This is running locally and cannot be accessed through the internet.

5.2 Cloud-Based Hosting

After developing the web application and UI using Gradio, it must be hosted on a cloud service for accessibility for everyone to use. Hosting a web application on a cloud service typically involves utilizing infrastructure or platform services provided by the cloud provider. This allows the web application to be deployed, managed, and accessible to users over the internet. The cloud hosting platform of choice is Hugging Face Spaces [29]. Hugging Face Spaces allows to host and share machine learning applications. It is

chosen due to the ease of use and the direct integration with Gradio library. To upload the Gradio application to Hugging Face Spaces, everything related to it must be in a single directory. For example, the application files might all exist at the path `PCOS_detector/` with the file structure:

```
PCOS_detector/
    ├── DenseNet121d_model.pth
    ├── app.py
    ├── examples/
    │   ├── example_1.jpg
    │   ├── example_2.jpg
    │   └── example_3.jpg
    ├── model.py
    └── requirements.txt
```

Where:

- `DenseNet121d_model.pth` is the previously trained PyTorch model file
- `app.py` contains the code that implements the Gradio interface
- `examples/` contains example images to use our Gradio app
- `model.py` contains the model definition as well as any transforms associated with the model
- `requirements.txt` contains the dependencies to run our app such as `torch`, `torchvision`, and `gradio`

After getting the files ready for deploying the PCOS detector, it is time to upload those files to Hugging Face Spaces and Hugging Face Spaces will take care of the rest by building and running the application and make it available online with a free CPU that can be used to use the model on. The result is a deployed web application available for free use [30] by patients and doctors as a decision support system.

Chapter 6 Conclusion

In conclusion, this thesis has presented a comprehensive examination of the use of CNNs and transfer learning in classifying ovarian ultrasound images. The main objective of this study was to investigate the potential of using multiple DenseNet architectures, both pre-trained, and trained from scratch to classify ovarian ultrasound images into healthy and infected cases as well as comparing their performance to a much smaller model built manually from scratch. In addition to that, deploy the model to be accessible and used by doctors as a decision support system as a tool to help in diagnosis. The datasets used in this study consisted of healthy ovaries and ovaries with PCOS, and the suggested models were trained and evaluated on these datasets.

The results of this study indicate that the proposed method was able to achieve a somewhat satisfactory accuracy of 72% on the test set. This level of accuracy is promising and suggests that the use of ImageNet models and architectures can be a valuable tool for improving the performance of CNNs in medical imaging tasks. However, it is important to note that further research is required to improve the accuracy of this model and to generalize the results to a larger population.

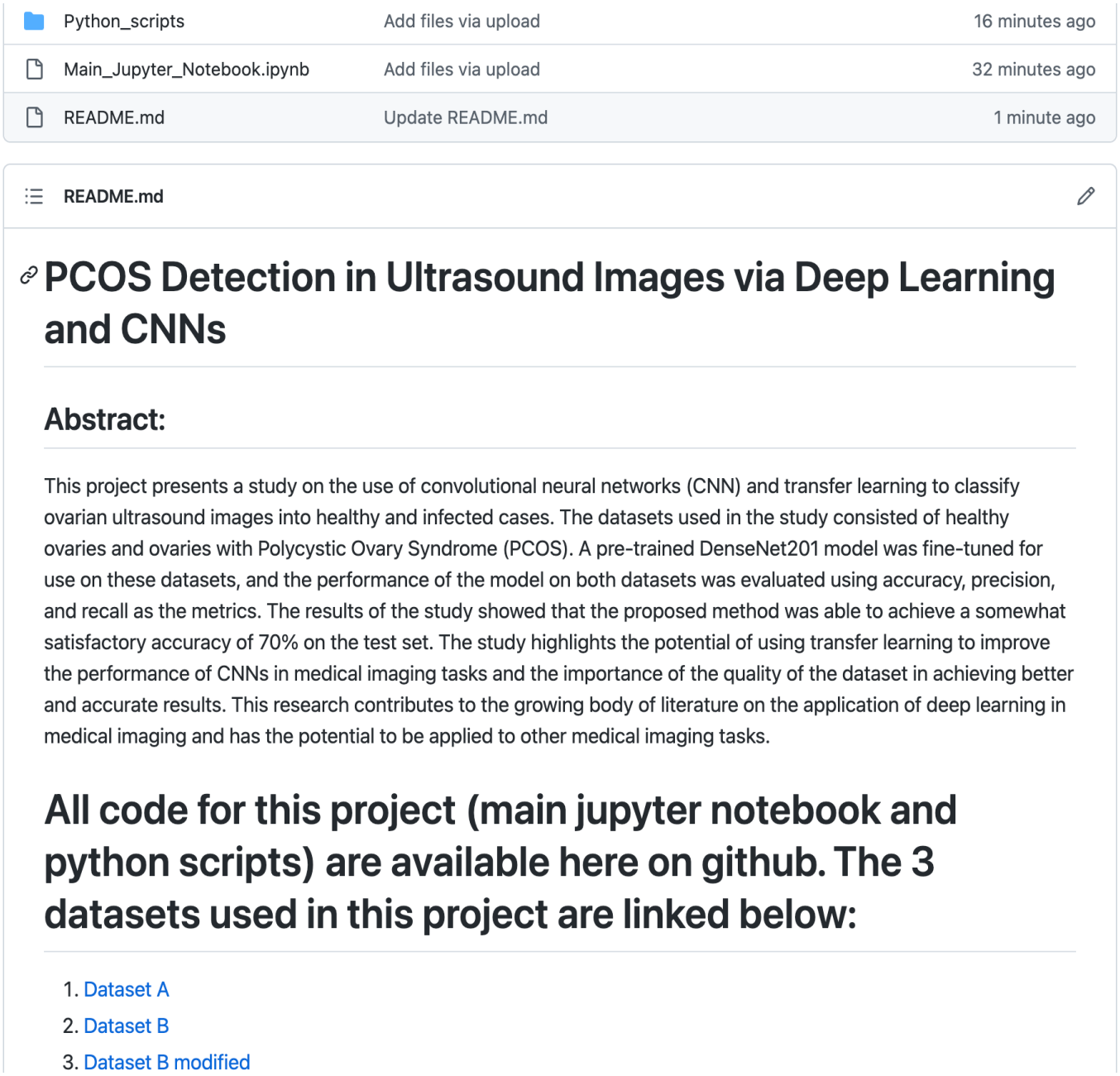
Additionally, as observed from the first two experiments on datasets A and B, data quality is of the utmost importance when training deep learning models, especially in the medical and health fields. If the data is flawed or biased, the model will likely produce inaccurate or unreliable results even if the results appear to be satisfactory. In the medical and health field, this can have serious consequences as it can lead to incorrect diagnoses or treatment recommendations, potentially causing harm to patients. It is therefore essential to ensure that the data used to train these models is of the highest quality and accurately represents the population it is intended to serve.

In summary, this thesis has demonstrated the power of CNNs and deep learning in medical imaging. The proposed method is a promising approach for classifying ovarian ultrasound images and has the potential to be applied to other medical imaging tasks. Furthermore, this study highlights the importance of the pre-processing techniques taken in achieving better results. Overall, this thesis has contributed to the growing body of

literature on the application of deep learning in medical imaging and has shown the potential of using deep learning in ovarian ultrasound images analysis.

Appendix – Code

All code is available on GitHub [31]. Also, links to all of the datasets are provided in the main page of the GitHub link.



Python_scripts	Add files via upload	16 minutes ago
Main_Jupyter_Notebook.ipynb	Add files via upload	32 minutes ago
README.md	Update README.md	1 minute ago

README.md

PCOS Detection in Ultrasound Images via Deep Learning and CNNs

Abstract:

This project presents a study on the use of convolutional neural networks (CNN) and transfer learning to classify ovarian ultrasound images into healthy and infected cases. The datasets used in the study consisted of healthy ovaries and ovaries with Polycystic Ovary Syndrome (PCOS). A pre-trained DenseNet201 model was fine-tuned for use on these datasets, and the performance of the model on both datasets was evaluated using accuracy, precision, and recall as the metrics. The results of the study showed that the proposed method was able to achieve a somewhat satisfactory accuracy of 70% on the test set. The study highlights the potential of using transfer learning to improve the performance of CNNs in medical imaging tasks and the importance of the quality of the dataset in achieving better and accurate results. This research contributes to the growing body of literature on the application of deep learning in medical imaging and has the potential to be applied to other medical imaging tasks.

All code for this project (main jupyter notebook and python scripts) are available here on github. The 3 datasets used in this project are linked below:

1. Dataset A
2. Dataset B
3. Dataset B modified

References

[1] E.-M. HF., "Polycystic ovary syndrome: definition, aetiology, diagnosis and treatment.," *Nat Rev Endocrinol.,* vol. 14(5):270–84., 2018.

[2] R. A. D. Z. Farkas J, "Psychological Aspects of the Polycystic Ovary Syndrome.," *Gynecol Endocrinol,* vol. 30, no. 2, 2013.

[3] L. J. Louwers YV, "Characteristics of polycystic ovary syndrome throughout life.," *Therapeutic Advances in Reproductive Health.,* vol. 14, 2020.

[4] J. e. a. Adams, "Prevalence of polycystic ovaries in women with anovulation and idiopathic hirsutism.," *British Medical Journal,* Vols. 293,6543 (1986): 355-9, 9 Aug 1986.

[5] R. Azziz, "PCOS: a diagnostic challenge," *Reproductive BioMedicine Online,* vol. 8, no. 6, pp. 644-648, 5 April 2004.

[6] F. M. N. P. V. Z. D. d. A. Cesare Battaglia, "Ultrasound evaluation of PCO, PCOS and OHSS," *Reproductive BioMedicine Online,* vol. 9, no. 6, pp. 614-619, 2004.

[7] Y. Wang, X. Ge, H. Ma, S. Qi, G. Zhang and Y. Yao, "Deep Learning in Medical Ultrasound Image Analysis: A Review," *IEEE Access,* vol. 9, pp. 54310-54324, 2021.

[8] O.-G. N. V.-R. S. G.-M. A. K. V. R.-R. R. e. a. Diaz-Escobar J, "Deep-learning based detection of COVID-19 using lung ultrasound imagery," *PLoS ONE 16(8): e0255886.,* 2021.

[9] J. C. N. a. A. F. G. Carneiro, "The Segmentation of the Left Ventricle of the Heart From Ultrasound Data Using Deep Learning Architectures and Derivative-Based Search Methods," *IEEE Transactions on Image Processing,* vol. 21, no. 3, pp. 968-982, 2012.

[10] S. e. a. Han, "A deep learning framework for supporting the classification of breast lesions in ultrasound images," *Physics in medicine and biology,* vol. 62, no. 19, pp. 7714-7728, 2017.

[11] M. M. K. I. Hosain AK, "PCONet: A convolutional neural network architecture to detect polycystic ovary syndrome (PCOS) from ovarian ultrasound images.," *arXiv preprint,* 2 Oct 2022.

[12] "Kaggle," [Online]. Available: https://www.kaggle.com/datasets/anaghachoudhari/pcos-detection-using-ultrasound-images.

[13] W. e. a. Lv, "Deep Learning Algorithm for Automated Detection of Polycystic Ovary Syndrome Using Scleral Images," *Frontiers in endocrinology,* vol. 12, 27 Jan. 2022.

[14] P. P. a. M. R. H. M. S. Bharati, "Diagnosis of Polycystic Ovary Syndrome Using Machine Learning Algorithms," *2020 IEEE Region 10 Symposium (TENSYMP),* pp. 1486-1489, 2020.

[15] P. K. V. C. &. A. S. Sakshi Srivastava, "Detection of Ovarian Cyst in Ultrasound Images Using Fine-Tuned VGG-16 Deep Learning Network," *SN Computer Science,* 2020.

[16] PyTorch, "Documentation for DenseNet201," [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.densenet201.html.

[17] A. S. e. al., "How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers," *Transactions on Machine Learning Research,* vol. https://doi.org/10.48550/arXiv.2106.10270.

[18] Z. L. L. v. d. M. K. Q. W. Gao Huang, "Densely Connected Convolutional Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* pp. 2261-2269, 2016.

[19] M. A. A. B. a. G. D. F. Morid, "A scoping review of transfer learning research on medical image analysis using ImageNet," *Computers in biology and medicine,* vol. 128, p. 104115, 2020.

[20] W. D. R. S. L. -J. L. K. L. a. L. F.-F. J. Deng, "ImageNet: A large-scale hierarchical image database," *IEEE Conference on Computer Vision and Pattern Recognition,* 2009.

[21] R. M. S. Sharma, "Conventional Machine Learning and Deep Learning Approach for MultiClassification of Breast Cancer Histopathology Images—a Comparative Insight," *J Digit Imaging 33.*

[22] "Difference between BCELoss and BCEWithLogitsLoss in PyTorch," [Online]. Available: https://androidkt.com/difference-between-bceloss-and-bcewithlogitsloss-in-pytorch/.

[23] R. R. J. S. S. K. Aman Gupta, "Adam vs. SGD: Closing the generalization gap on image classification," *OPT2021: 13th Annual Workshop on Optimization for Machine Learning.*

[24] "Google Colab," [Online]. Available: https://colab.research.google.com/.

[25] Adiwijaya, U. NOVIA WISESTY and W. Astuti, "Polycystic Ovary Ultrasound Images Dataset," Telkom University Dataverse 2021. [Online]. Available: https://doi.org/10.34820/FK2/QVCP6V.

[26] C. Hughes, "Transfer Learning on Greyscale Images: How to Fine-Tune Pretrained Models on Black-and-White Datasets," [Online]. Available: https://towardsdatascience.com/transfer-learning-on-greyscale-images-how-to-fine-tune-pretrained-models-on-black-and-white-9a5150755c7a.

[27] "CNN Explainer," [Online]. Available: https://poloclub.github.io/cnn-explainer/.

[28] "Gradio," [Online]. Available: https://www.gradio.app/.

[29] "Hugging Face Spaces," [Online]. Available: https://huggingface.co/spaces.

[30] "PCOS Detector," [Online]. Available: https://huggingface.co/spaces/Haidary/PCOS_detector.

[31] "Project Github Repository," [Online]. Available: https://github.com/haidary99/PCOS_classification.

الملخص:

تقدم هذه الرسالة دراسة حول استخدام نماذج الشبكات العصبونية الالتفافية العميقة (CNN) مثل نماذج DenseNet لتصنيف صور الموجات فوق الصوتية على المبيض إلى حالات صحية ومصابة. تم أيضًا إنشاء نموذج أصغر بكثير (lustNet) حيث يتم مقارنة أدائه بنماذج DenseNet المتوفرة. تتكون مجموعات البيانات المستخدمة في الدراسة من المبايض السليمة والمبايض المصابة بمتلازمة تكيس المبايض (PCOS). باستخدام التعلم عن طريق النقل، تم ضبط نموذج DenseNet201 الذي تم تدريبه مسبقًا بشكل دقيق للتدرب على مجموعة البيانات المتاحة. بعد ذلك، تم تدريب نموذج DenseNet121d من البداية على نفس مجموعة البيانات. أخيرًا، تم إنشاء lustNet من البداية وتم تدريبه أيضًا على مجموعة البيانات. تم تقييم أداء النماذج الثلاثة في مجموعة البيانات باستخدام الدقة والاسترجاع. أظهرت نتائج الدراسة أن DenseNet121d حققت أكثر النتائج المرجوة بسبب الحصول على أعلى نسبة استرجاع وهو الأكثر أهمية في مهام التصنيف الطبي. كانت دقة النموذج المقترح قادرة على تحقيق دقة مرضية إلى حد ما بنسبة 72 ٪ تقريبًا في مجموعة الاختبار. تم نشر النموذج المقترح أيضًا على الويب ليتمكن الأطباء من الوصول إليه لاستخدامه كنظام دعم القرار. أخيرًا، تسلط الدراسة الضوء أيضًا على أهمية جودة مجموعة البيانات في تحقيق نتائج أكثر دقة. يساهم هذا البحث في العدد المتزايد من المؤلفات حول تطبيق التعلم العميق في التصوير الطبي ولديه القدرة على تطبيقه على مهام التصوير الطبي الأخرى.

كلية الهندسة والتكنلولوجيا
قسم هندسة الحاسوب و المعلوماتية

" كشف متلازمة تكيس المبايض في صور الموجات فوق الصوتية عبر التعلم العميق والشبكات العصبونية الالتفافية"

يغطي هذا البحث الجزء من متطلبات نيل شهادة البكالوريوس في الهندسة المعلوماتية

اعداد
سيد يوسف سيد لوى حيدرى

بإشراف
د.محمد حيان السباعي

مشروع التخرج الثاني
2022-2023