IT Business School

Mini-project report

Deep learning

Breast Cancer Classification

Prepared by

HAZEM AYACHI

Under the supervision of

Mr. AHMED BEN TALEB

ACADEMIC YEAR:

2024-2025

Acknowledgements

The completion of this project was made possible thanks to the support and assistance of several individuals to whom I wish to express my sincere gratitude.

First and foremost, I would like to extend my warmest thanks to AHMAD BEN TALEB, my academic supervisor, for their valuable guidance, insightful advice, availability, and constant support throughout this work. Their guidance was instrumental in structuring my approach and deepening my knowledge.

My thanks also go to my family and friends for their unconditional moral support, encouragement, and patience during this period of intense work.

Finally, I thank everyone who, directly or indirectly, contributed to the success of this project.

List of Acronyms and Abbreviations

AI – Artificial Intelligence

ANN – Artificial Neural Network

BP – Backpropagation

CAD – Computer-Aided Diagnosis

CNN – Convolutional Neural Network

CPU – Central Processing Unit

CT – Computed Tomography

DL – Deep Learning

F1-Score – Harmonic mean of Precision and Recall

GCP – Google Cloud Platform

HTTP – Hypertext Transfer Protocol

I/O – Input/Output

Keras – High-level neural networks API (now integrated into TensorFlow)

ML – Machine Learning

MRI – Magnetic Resonance Imaging

NN – Neural Network

PNG – Portable Network Graphics

RAM – Random Access Memory

ReLU – Rectified Linear Unit (activation function)

RGB – Red, Green, Blue (color model)

SDK – Software Development Kit

SGD – Stochastic Gradient Descent

SSD – Solid State Drive

tf – TensorFlow

VGG – Visual Geometry Group (a CNN architecture)

VM – Virtual Machine

WSGI – Web Server Gateway Interface

Table of Contents

Table des matières

List of Figures

1. General Introduction

The field of artificial intelligence, and more specifically Deep Learning, has revolutionized numerous sectors in recent years, notably thanks to significant advancements in image processing. The ability of deep neural networks, and particularly Convolutional Neural Networks (CNNs), to automatically extract complex features from raw data has opened up new perspectives for applications in critical domains such as medicine.

Medical diagnosis, and specifically the analysis of histological images, is a field where precision and speed are paramount. Breast cancer, being one of the most common cancers in women, requires early and reliable diagnosis to optimize treatment success rates. Computer-aided diagnosis (CAD) systems using Deep Learning techniques show considerable potential to assist pathologists in analyzing large quantities of images, thereby reducing workload and potentially improving accuracy.

This report documents a project focused on applying Deep Learning techniques for the classification of breast cancer histological images. Using the BreaKHis dataset, the objective was to explore different CNN architectures, evaluate the impact of performance enhancement techniques like data augmentation and transfer learning, and finally deploy the most performant model as an accessible web service.

2. Project Context and Motivation

This project was undertaken as part of a university mini-project aimed at practically applying Deep Learning concepts to a real and significant image classification problem. The choice of breast cancer histological image classification was motivated by the medical importance of this field and the availability of a relevant public dataset (BreaKHis).

The main motivation was to master the entire process of developing a Deep Learning-based solution, from data preparation and model design to their rigorous evaluation and operational deployment. The project aimed to compare the effectiveness of different approaches (simple model vs. augmentation vs. transfer learning) and to understand the practical challenges related to putting a Deep Learning model into production.

The specific objectives of this project were to:

- Set up a complete pipeline for loading, preprocessing, and managing the BreaKHis dataset.
- Design and train a baseline CNN model to establish a performance reference.
- Evaluate the impact of data augmentation on the performance of the baseline model.
- Implement and evaluate a transfer learning approach using a pre-trained model (DenseNet121) with a fine-tuning and feature fusion strategy.
- Compare the performance of different approaches using appropriate evaluation metrics.

- Deploy the most performant model as a containerized web API on a cloud platform (GCP Cloud Run).

3. State of the Art: Fundamental Concepts in Deep Learning and Medical Applications

This section presents the essential theoretical concepts underlying the work carried out in this project, as well as an overview of the applications of Deep Learning in the field of medical imaging.

3.1. Foundations of Deep Learning

Deep Learning is a subcategory of Machine Learning that uses artificial neural networks composed of multiple layers (hence the term "deep"). Unlike traditional Machine Learning algorithms that require manual feature extraction, Deep Learning allows the model to automatically learn hierarchical representations of data.

3.1.1. Artificial Neural Networks

An artificial neural network is composed of layers of interconnected neurons. Each neuron receives inputs, applies a non-linear activation function, and produces an output. Deep networks include an input layer, several hidden layers, and an output layer. Learning involves adjusting the weights (synapses) and biases of the neurons to minimize the error between predicted and actual outputs.

3.1.2. The Training Process (Backpropagation, Optimizers)

Training a deep neural network is generally done using the backpropagation of gradient method. This method calculates the error at the network's output and propagates it back through the preceding layers to adjust weights and biases with the goal of reducing the error.

Optimizers (like Adam, SGD, RMSprop) are algorithms that guide the weight adjustment process by determining how to modify the model's parameters based on the gradient calculated by backpropagation. The objective is to find the minimum of the loss function.

3.2. Convolutional Neural Networks (CNNs)

CNNs are a specific type of deep neural network particularly effective for processing structured data like images. Their architecture is inspired by the visual cortex of animals.

3.2.1. Convolutional Layers

The convolutional layer is the core of a CNN. It applies a filter (or kernel) over the input image to detect local patterns (edges, textures, etc.). The filter slides over the image, performing a convolution operation and producing a feature map that highlights the presence of these patterns.

3.2.2. Pooling Layers

Pooling layers (like Max Pooling or Average Pooling) reduce the spatial dimension of feature maps, which helps reduce the number of parameters and computations, and makes the model more robust to small spatial variations in the image.

3.2.3. Common CNN Architectures (General Overview)

Numerous high-performing CNN architectures have been developed over time, such as LeNet, AlexNet, VGG, GoogLeNet (Inception), ResNet, and DenseNet. These architectures vary in their depth, the type of layers used, and how connections are established between layers.

3.3. Performance Enhancement Techniques

To improve the performance of Deep Learning models, especially when dealing with limited-size datasets or to prevent overfitting, various techniques are used.

3.3.1. Data Augmentation

Data augmentation involves creating new synthetic training images from existing ones by applying random transformations (rotations, translations, zooms, flips, brightness changes, etc.). This increases the size and diversity of the training dataset, helping the model generalize better and be less sensitive to variations in unseen data.

3.3.2. Transfer Learning (Fine-tuning vs Feature Extraction)

Transfer learning is a technique that involves reusing a model that has already been trained on a similar task and a large dataset (like ImageNet) for a new task. The idea is that the pre-trained model has already learned generic features (edge detection, shape detection, etc.) that can be useful for the new task, even if the domain is different.

There are two main approaches:

- **Feature Extraction:** Use the pre-trained model as a feature extractor by freezing its weights and training only a new classification head on the extracted features.
- **Fine-tuning:** Use the pre-trained model as a starting point, but allow the adjustment (with a low learning rate) of the weights of the last layers (or all layers) of the pre-trained model to better adapt it to the new task and dataset.

3.4. Applications of Deep Learning in Medical Imaging

Deep Learning has had a major impact on the field of medical imaging, offering new possibilities for diagnosis and analysis.

3.4.1. Computer-Aided Diagnosis

Deep Learning models can be trained to detect anomalies or signs of diseases in various imaging modalities (X-rays, MRIs, CT scans, histological images). They can serve as a "second opinion" for clinicians, helping to identify suspicious areas that might be missed by the naked eye.

3.4.2. Segmentation and Object Detection

Deep Learning is used to segment anatomical structures (organs, tumors) or detect specific objects (lesions, cells) in medical images. This is crucial for quantification, treatment planning, and morphological analysis.

3.4.3. Medical Image Classification

Medical image classification aims to assign a category (e.g., benign or malignant, tumor type) to an entire image or a region of interest. This is the specific application explored in this project for breast cancer histological images.

4. Technical Study and Choices: Models, Dataset, and Deployment Platform

This section details the technical choices made for this project, including the dataset selection, the explored model architectures, and the justification for the cloud deployment platform.

4.1. Detailed Presentation of the BreaKHis Dataset

The project relied on the BreaKHis dataset, a widely used public benchmark for breast cancer histological image classification.

4.1.1. Dataset Origin and Characteristics

The BreaKHis v1 dataset contains digitized images of breast tissue biopsies. The images are captured at different magnification factors (40X, 100X, 200X, 400X).

4.1.2. Structure and Classes

The dataset is organized into two main classes: Benign and Malignant. Each class is subdivided into several subtypes. For this project, the focus was on binary classification (Benign vs. Malignant). The dataset includes approximately 7900 images in PNG format.

4.2. Analysis of Explored Model Approaches

The project explored three distinct approaches for classification, allowing for a comparison of their performance.

4.2.1. Baseline CNN Model

A simple and custom CNN model was built to serve as a starting point and establish a performance baseline. This approach helps understand the capabilities of a basic network without advanced techniques.

4.2.2. Impact of Data Augmentation

Data augmentation was applied to the baseline model to evaluate its effectiveness in improving generalization and reducing overfitting on the moderately sized dataset.

4.2.3. Transfer Learning Model (DenseNet121)

The use of a pre-trained model (DenseNet121) via transfer learning was explored to leverage knowledge gained on a large dataset (ImageNet) and assess whether this knowledge is transferable to the domain of histological images. A fine-tuning strategy with feature fusion was preferred.

4.3. Justification of Technical Choices

Technical choices were guided by the project's objectives, time and resource constraints, and the desire to explore relevant Deep Learning techniques.

4.3.1. Choice of Baseline Architecture

The baseline architecture was designed to be relatively simple while including the fundamental blocks of a CNN (convolutional layers, pooling, dense layers) to provide a clear comparison basis.

4.3.2. Choice of Augmentation Techniques

Rotation and flipping techniques were chosen because they are common and effective for images, and particularly relevant for histological images where orientation can vary without altering clinical meaning.

4.3.3. Choice of DenseNet121 for Transfer Learning

DenseNet121 was selected for several reasons: it is an architecture known for its performance on ImageNet, it is relatively efficient in terms of parameters compared to other very deep models, and its structure with dense connections facilitates feature propagation. Feature fusion from different layers aims to exploit both low-level and high-level features.

4.3.4. Choice of Flask for the API

Flask was chosen for web API development due to its lightness, simplicity, and ease of use for quickly creating HTTP endpoints, which was sufficient for the needs of this mini-project.

4.3.5. Choice of Docker for Containerization

Docker was selected for containerization to ensure the portability of the application and its dependencies. Containerization simplifies deployment by ensuring a consistent execution environment, regardless of the target environment.

4.3.6. Choice of Google Cloud Run for Deployment

Google Cloud Run was chosen as the deployment platform for its advantages in serverless capabilities, automatic scaling (including to zero), and native support for Docker containers. This allowed for quick and efficient deployment without having to manage underlying infrastructure (VMs, Kubernetes clusters), which was well-suited to the project's scope.

4.4. General Architecture of the Solution

The overall architecture of the developed solution can be diagrammed as follows:

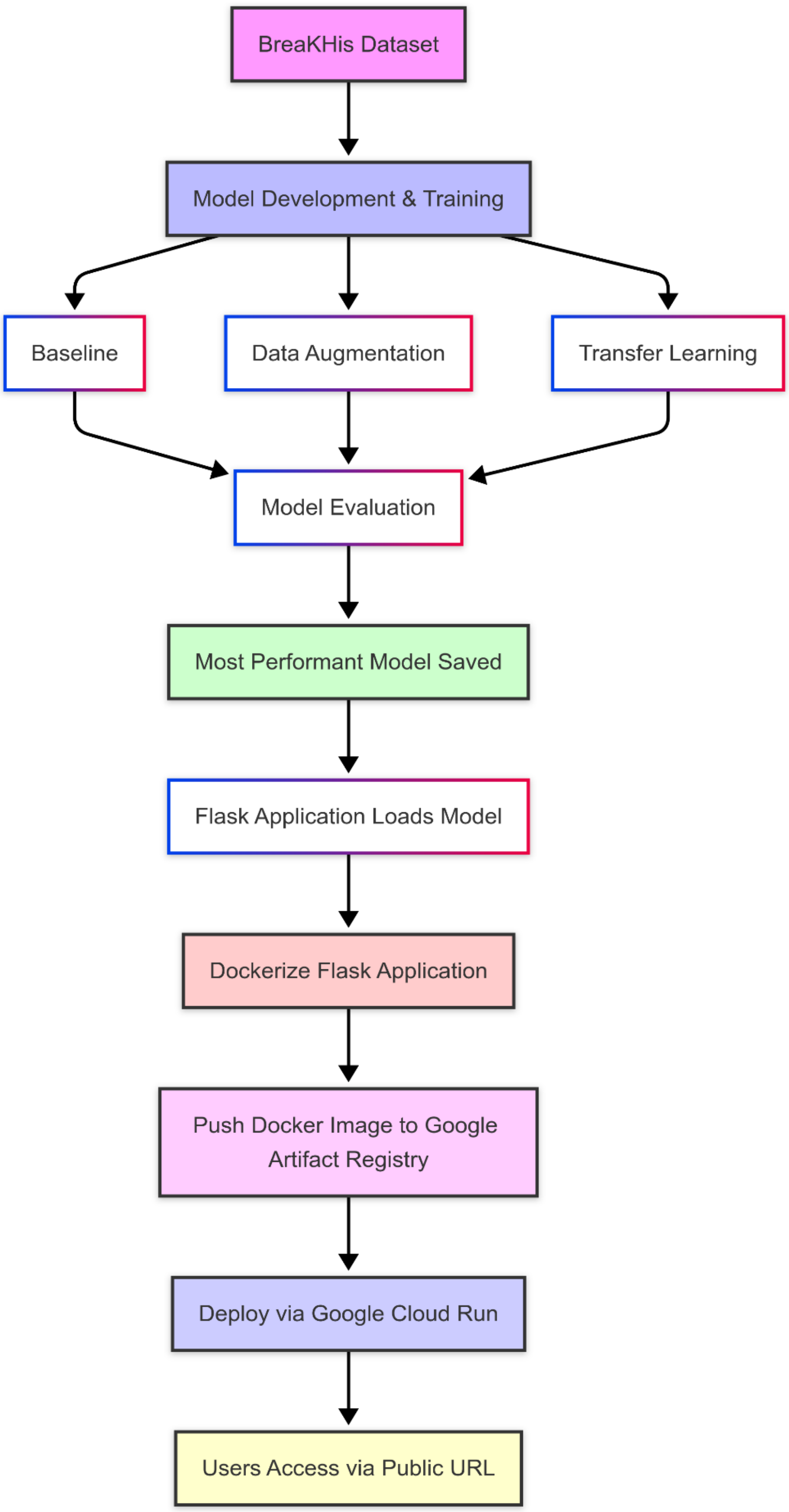
BreaKHis Dataset
Model Development & Training
Baseline
Data Augmentation
Transfer Learning
Model Evaluation
Most Performant Model Saved
Flask Application Loads Model
Dockerize Flask Application
Push Docker Image to Google Artifact Registry
Deploy via Google Cloud Run
Users Access via Public URL

Figure 1: The Project architecture

This architecture illustrates the workflow from raw data to the accessible classification service via the cloud.

4.5. Detailed Working Environment and Technical Prerequisites

The project was developed and executed in an environment including the following elements:

- **Hardware:**

CPU	AMD Ryzen 7 5800H CPU @ 3.20GHz (8 cors, 16 threads)
RAM	16 Go
Storage	387Go SSD

- **Software:**
  - Operating System: Windows 11
  - Python development environment: Anaconda
  - Python Libraries: TensorFlow, Keras, scikit-learn, Pillow, Flask, Gunicorn, etc. (Complete list in `requirements.txt`)
  - Docker Desktop (for containerization)
  - Google Cloud Platform (GCP) account with Cloud Run and Artifact Registry services enabled.
  - Google Cloud SDK (for interacting with GCP via the command line).

5. Implementation: Development, Training, and Deployment of the Solution

This section describes the practical steps taken to implement the project, from data preparation to the deployment of the final model.

5.1. Data Preparation and Preprocessing

A crucial step was preparing the BreaKHis dataset for model training.

5.1.1. Image Loading and Label Extraction

Images were loaded using the `glob` library to traverse directories. Labels (Benign/Malignant) were extracted from filenames or folder structure.

5.1.2. Preprocessing Pipeline (Decoding, Resizing, Normalization)

A consistent preprocessing pipeline was applied to all images:

- Decoding PNG images into 3-channel RGB tensors.
- Resizing images to a uniform size of 128x128 pixels using `tf.image.resize`.
- Normalizing pixel values to the range [0, 1] by dividing by 255.0.

5.1.3. Dataset Splitting (Train, Validation, Test)

The dataset was split into training (60%), validation (20%), and test (20%) sets using `sklearn.model_selection.train_test_split`. Stratification (`stratify=labels`) was used to ensure that the class distribution was similar in each set, thus avoiding bias.
[Table: Dataset Sizes (Train, Val, Test)]

5.1.4. Creation of `tf.data.Dataset` Input Pipelines

The `tf.data.Dataset` library was used to create efficient input pipelines for training and evaluation. These pipelines include shuffling (`shuffle`), batching (`batch`) with a batch size of 64, and prefetching (`prefetch`) to optimize input/output performance. Binary labels were one-hot encoded to be compatible with the categorical_crossentropy loss function.

5.2. Implementation and Training of the Baseline CNN Model

The baseline CNN model was built using the Keras Sequential API.

5.2.1. Architecture Description (Layers, Parameters)

The architecture included:

- An input layer specifying the image shape (128, 128, 3).
- Three convolutional blocks, each consisting of a Conv2D layer (with ReLU activation) followed by a MaxPooling2D layer. The number of filters increased in successive blocks (32, 64, 128).
- A Flatten layer to flatten the output of the convolutional layers.
- A Dense layer with ReLU activation, followed by a Dropout layer (rate 0.5) for regularization.
- A final Dense output layer with 2 units (for the two classes) and a softmax activation to produce class probabilities.

5.2.2. Training Configuration (Optimizer, Loss Function, Metrics)

The model was compiled with the Adam optimizer (initial learning rate of 0.001), the categorical_crossentropy loss function (suitable for one-hot encoded labels and multiclass classification), and the accuracy metric for tracking performance during training.

5.2.3. Training Process (Epochs, Callbacks)

Training was performed on the training dataset *without augmentation* for 20 epochs. Keras callbacks were used:

- EarlyStopping with a patience of 10 epochs to stop training if performance on the validation set stops improving, thus preventing overfitting.
- ReduceLROnPlateau with a patience of 5 epochs to reduce the learning rate if performance on the validation set plateaus, helping the model converge.

5.3. Implementation of Data Augmentation and Retraining of the Baseline

To evaluate the impact of data augmentation, the same baseline CNN model was retrained using an augmented training dataset.

5.3.1. Description of Implemented Augmentation Techniques

A custom augmentation function (augment_image_tf) was created using tensorflow.image operations. For each image, it generated augmented versions by rotating 90 and 180 degrees, followed by a horizontal flip. The original image was also kept, effectively tripling the number of training samples.

5.3.2. Integration into the Input Pipeline

The augmentation function was integrated into the tf.data.Dataset creation function to be applied dynamically to images *only* during training.

5.3.3. Training with Augmentation

The baseline model was compiled and trained with the same hyperparameters (optimizer, loss, metrics) and callbacks as before, but using the augmented input pipeline on the training set.

5.4. Implementation and Training of the Transfer Learning Model

The transfer learning approach used the DenseNet121 model pre-trained on ImageNet.

5.4.1. Loading the Pre-trained Model (DenseNet121)

The DenseNet121 model was loaded using `tf.keras.applications.DenseNet121`, specifying `include_top=False` to exclude the final ImageNet classification layer.

5.4.2. Fine-tuning Strategy

Initially, the weights of the pre-trained model were loaded. Unlike simple feature extraction where weights are frozen, a fine-tuning strategy was adopted by making the layers of the base model trainable (`base_model.trainable = True`). This allows the model to adapt the features learned on ImageNet to the specific characteristics of histological images, although often with a lower learning rate.

5.4.3. Implementation of Feature Fusion

Instead of using only the output of the last layer of the pre-trained model, features were extracted from three intermediate convolutional blocks (`conv3_block12_concat`, `conv4_block24_concat`, `conv5_block16_concat`). Each intermediate output was passed through a `GlobalAveragePooling2D` layer, a `Dense` layer (64 units, ReLU), and a `BatchNormalization` layer. The outputs of these three branches were then concatenated. This fusion aims to combine features from different levels of abstraction.

5.4.4. Adding the Classification Head

The fused features were passed through a `Dense` layer (16 units, ReLU), a `BatchNormalization` layer, and a `Dropout` layer (rate 0.45) before the final output layer (`Dense` with 2 units and `softmax`).

5.4.5. Configuration and Training Process (Learning Rate, Callbacks)

The transfer learning model was compiled with the Adam optimizer and the `categorical_crossentropy` loss function. A lower learning rate (e.g., 1e-5) was used for fine-tuning to avoid rapidly disturbing the pre-trained weights. The same `EarlyStopping` and `ReduceLROnPlateau` callbacks were applied. Training was performed on the training dataset *with augmentation*.

5.5. Development of the Web API with Flask

The most performant model was integrated into a simple web application using the Flask microframework to make it accessible via HTTP.

5.5.1. Flask Application Structure (`main.py`)

The application was developed in a single Python file, `main.py`, containing the Flask code to define routes and handle requests.

5.5.2. Loading the Trained Model

The trained and saved Keras model (`transfer_learning_model.keras`) was loaded into memory when the Flask application starts, ready to perform predictions.

5.5.3. Implementation of the `/predict` Endpoint

An HTTP POST endpoint, `/predict`, was created to receive images for classification. It expected a `multipart/form-data` request containing the image file. A GET endpoint `/` was also added for a simple service status check.

5.5.4. Handling Requests and Preprocessing Input Images

When a POST request is received on /predict, the application:

- Retrieves the image file from the request.
- Uses the Pillow library to open and read the image.
- Applies the same preprocessing pipeline used during training (resizing to 128x128, normalization to [0, 1]) using TensorFlow.
- Performs the prediction by calling model.predict() on the preprocessed image.
- Formats the result (predicted label - "Benign" or "Malignant", and confidence score) in JSON format and returns it as a response.

5.6. Containerization of the Application with Docker

To facilitate deployment, the Flask application and all its dependencies were packaged into a Docker container.

5.6.1. Creation of the requirements.txt File

A requirements.txt file listed all the Python libraries required to run the application (tensorflow, flask, gunicorn, pillow, etc.).

5.6.2. Writing the Dockerfile (Base Image, Dependencies, File Copy, Port, Entrypoint)

A Dockerfile was created to define the steps for building the Docker image:

- Using a lightweight Python base image (e.g., python:3.9-slim).
- Copying the requirements.txt file and installing dependencies via pip.
- Copying the application code (main.py) and the saved model.
- Exposing the listening port (8080, required by Cloud Run).
- Defining the entrypoint (CMD) to start the Flask application using the Gunicorn WSGI server (gunicorn -w 4 -b 0.0.0.0:8080 main:app). Gunicorn is used as a more robust production server than Flask's built-in development server.

5.6.3. Building the Docker Image

The Docker image was built locally using the command docker build -t breast-cancer ..

5.7. Deployment on Google Cloud Platform (Cloud Run)

The Docker container was deployed on Google Cloud Run to make it publicly accessible.

5.7.1. Choice of Cloud Run and Advantages

Cloud Run was chosen for its serverless nature (no server management), automatic scaling based on traffic (including to zero instances if idle, reducing costs), and simplicity of deploying an HTTP container.

5.7.2. Pushing the Docker Image to Google Artifact Registry

The locally built Docker image was tagged and pushed to Google Artifact Registry, a private container registry service on GCP.

Enable the Artifact Registry API using the command :

```
# gcloud services enable artifactregistry.googleapis.com
```

Create a Docker repository in Artifact Registry using the command :

```
# gcloud artifacts repositories create mydepot --repository-format=docker --
location=europe-west1 --description="Dépôt Docker pour mon CNN"
```

Configure Docker authentication using the command:

```
# gcloud auth configure-docker europe-west1-docker.pkg.dev
```

Tag the local image with the full registry path using the command:

```
# docker tag breast-cancer europe-west1-docker.pkg.dev/deep-learning-
456710/mydepot/breast-cancer:latest
```

Push the image to Artifact Registry using the command :

```
# docker push europe-west1-docker.pkg.dev/deep-learning-456710/mydepot/breast-
cancer:latest
```

5.7.3. Deploying the Cloud Run Service (gcloud Commands)

The Cloud Run service was deployed using the gcloud run deploy command, specifying the service name, the container image from Artifact Registry, and the deployment region (europe-west1).
Deploy via the gcloud command line using the command:

```
# gcloud run deploy breast-cancer --image europe-west1-docker.pkg.dev/deep-
learning-456710/mydepot/breast-cancer:latest --platform managed --region europe-
west1 --port 8080 --cpu 1 --memory 2Gi --allow-unauthenticated
```

5.7.4. Service Configuration (Region, Resources, Access)

During deployment, the region was specified. The resources allocated to the container (CPU, RAM) were configured (1 CPU, 2 GiB RAM), taking into account TensorFlow's memory requirements. Unauthenticated access was allowed to facilitate testing within the project scope.

6. Evaluation and Performance Comparison

The three model approaches were rigorously evaluated on the unseen test dataset to compare their performance.

6.1. Evaluation Metrics Used (Accuracy, Precision, Recall, F1-Score, Confusion Matrix)

The following evaluation metrics were calculated:

- **Accuracy:** Proportion of correctly classified samples.
- **Precision:** Among samples predicted as positive, what proportion are actually positive. Useful for evaluating the cost of false positives.
- **Recall (Sensitivity):** Among samples that are actually positive, what proportion were correctly detected. Useful for evaluating the cost of false negatives (missing a malignant case).
- **F1-Score:** Harmonic mean of Precision and Recall, providing a balance between the two. The binary F1-score (calculated for each class and then averaged) was used.

- **Confusion Matrix:** A table summarizing classification results, showing the number of true positives, true negatives, false positives, and false negatives.

6.2. Detailed Results of the Baseline Model (Without and With Augmentation)

The performance of the baseline CNN model was evaluated on the test set:

- **Baseline CNN (Without Augmentation):**
  - Accuracy: [0.8578 %]
  - Loss: [0.3527]
  - Precision: [0.8766]
  - Recall: [0.9227]
  - F1-Score: [0.8991]
  - Discussion: The model showed basic classification capability, but with room for improvement

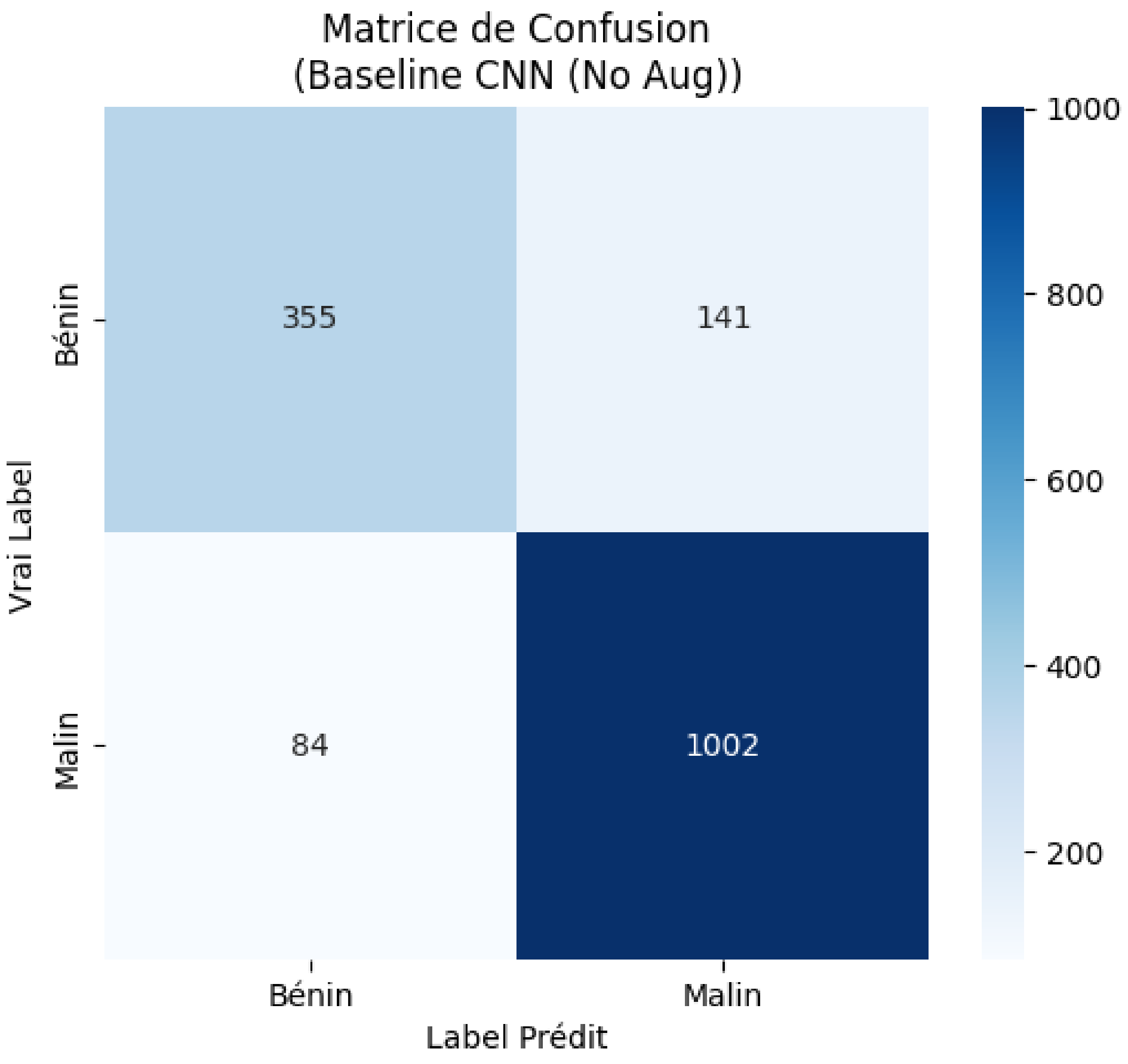

Figure 2: Baseline CNN confusion matrix

- **Baseline CNN (With Augmentation):**
    - Accuracy: [0.9052 %]
    - Loss: [0.2412]
    - Precision: [0.9341]
    - Recall: [0.9273]
    - F1-Score: [0.9307]
    - Discussion: Adding data augmentation significantly improved accuracy and F1-score, indicating better generalization

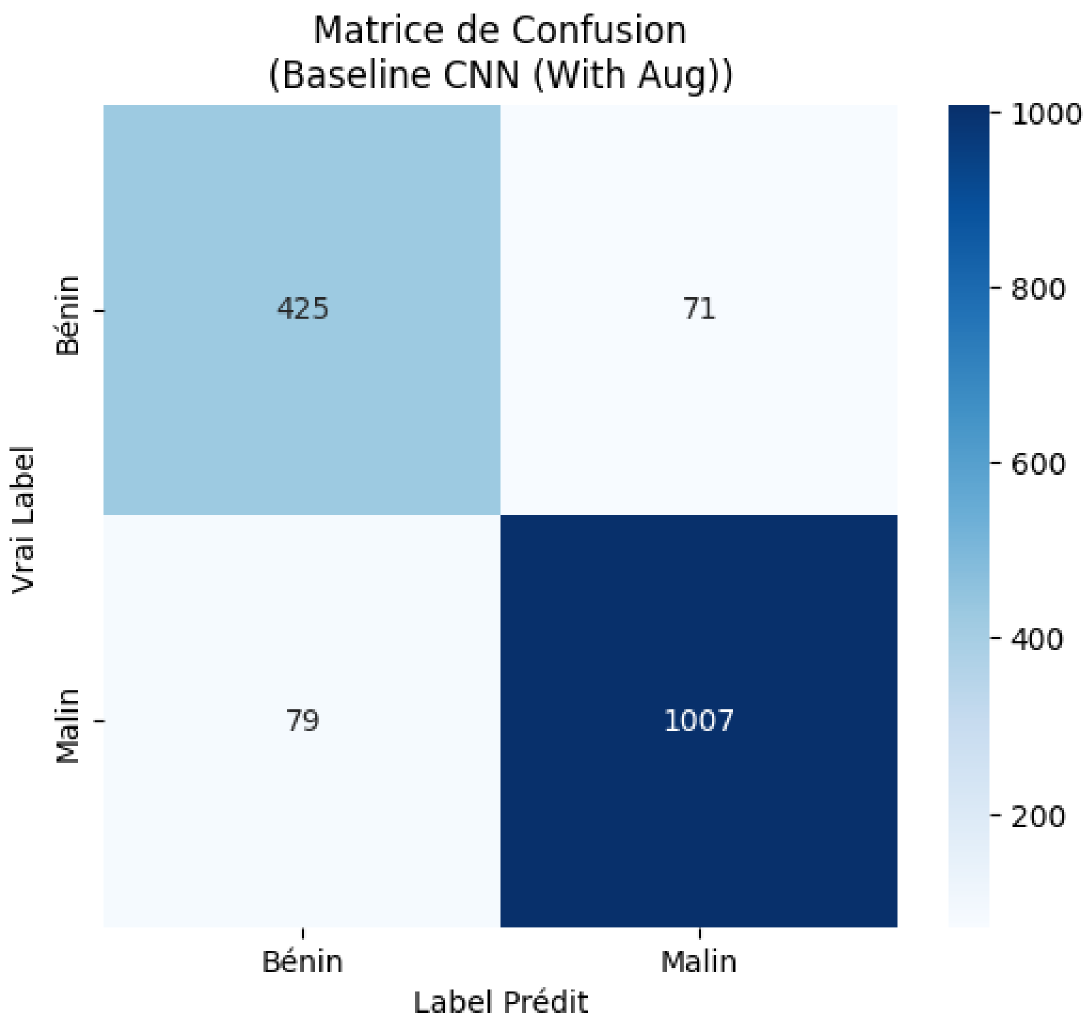

Figure 3: Baseline CNN with augmentation confusion matrix

6.3. Detailed Results of the Transfer Learning Model

The performance of the DenseNet121 model with fine-tuning and feature fusion was evaluated:

- **Transfer Learning (DenseNet121 + Augmentation):**
    - Accuracy: [0.9570 %]
    - Loss: [0.4597]
    - Precision: [0.9611]

- Recall: [0.9770]
- F1-Score: [0.9689]
- [Image: Confusion Matrix of the Transfer Learning model]
- Discussion: This model achieved the best results across all metrics, demonstrating the power of transfer learning

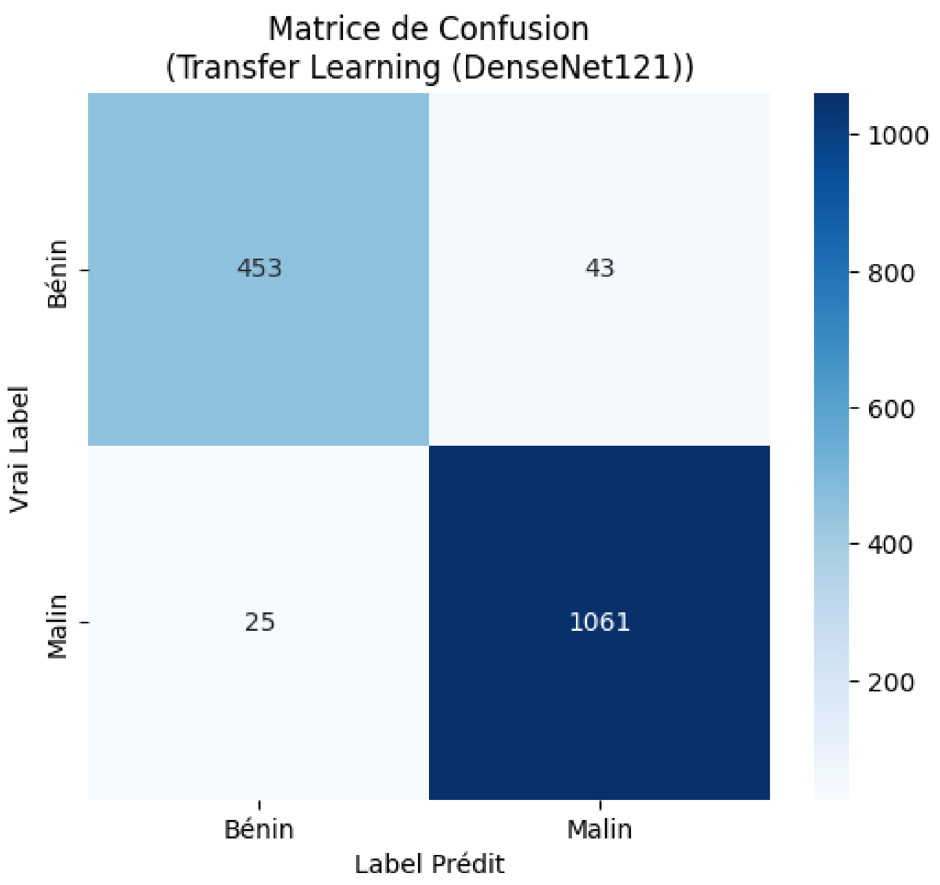

Figure 4: Transfer learning confusion matrix

6.4. Comparative Performance Analysis (Summary Table)

A summary table allows for easy comparison of the performance of the three approaches:

[Table: Performance Summary on the Test Set]

Model	Accuracy	Loss	Precision	Recall	F1-Score
Baseline CNN (Without Aug)	0.8578	0.3527	0.8766	0.9227	0.8991
Baseline CNN (With Aug)	0.9052	0.2412	0.9341	0.9273	0.9307
Transfer Learning (With Aug)	0.9570	0.4597	0.9611	0.9770	0.9689

6.5. Interpretation of Results and Discussion

The analysis of the results confirms the following points:

- The baseline model without augmentation offers moderate performance.
- Data augmentation is a very effective technique for improving model performance on a limited dataset by increasing its robustness and generalization capability.
- Transfer learning, by leveraging knowledge pre-trained on a large dataset, achieves significantly higher performance, even on a different domain like medical imaging. The fine-tuning and feature fusion strategy appears to have been beneficial.
- The transfer learning model achieved the best balance between Precision and Recall (reflected by the high F1-Score), which is crucial in a medical context where minimizing false negatives (missing a malignant case) is paramount.

7. Problems Encountered, Solutions Provided, and Lessons Learned

The completion of this project involved solving various technical problems, particularly during the model development and deployment phases.

7.1. Specific Problems Related to Model Development

- **Problem 1:** Difficulty interpreting training/validation curves (rapid overfitting of the baseline model without augmentation).
- **Problem 2:** Choosing training hyperparameters (learning rate, batch size, number of epochs) for each model.
- **Problem 3:** Fine understanding of the interaction between `tf.data.Dataset`, `.repeat()`, and `steps_per_epoch` when using complex input pipelines with augmentation.

7.2. Specific Problems Related to Cloud Deployment

- **Problem 4:** Errors during Docker image building (missing dependencies, incorrect file paths).
- **Problem 5:** Failure of the container to start on Cloud Run (entrypoint issue, incorrectly exposed port).
- **Problem 6:** Estimating the necessary CPU/Memory resources for the Cloud Run container, as TensorFlow is resource-intensive.

7.3. Diagnosis and Resolution Approach

A systematic approach was adopted for each problem:

- **Analyzing error messages:** Carefully examine the training logs (for models) or Cloud Run container logs (for deployment).
- **Consulting documentation:** Refer to the official documentation for TensorFlow, Keras, Flask, Docker, and Google Cloud Run.
- **Online search:** Use search engines and forums (Stack Overflow, community forums) to find solutions to similar problems.
- **Isolated testing:** Test specific components separately (e.g., build the Docker image locally, run the container locally before deployment).

- **Step-by-step debugging:** Add logging statements or use a debugger to understand code execution.

7.4. Lessons Learned from Practical Experience

Solving these problems was very instructive:

- The crucial importance of good data management and a robust input pipeline in Deep Learning.
- The necessity of understanding regularization mechanisms (Dropout, Early Stopping, Augmentation) to prevent overfitting.
- Learning to read and interpret logs for debugging containerized applications deployed on the cloud.
- Understanding the specifics of deploying containerized applications on serverless platforms like Cloud Run (port configuration, entrypoint).
- The importance of allocating appropriate resources for Deep Learning applications in production.

8. General Conclusion

This project successfully achieved its objectives by demonstrating the application of Deep Learning techniques for breast cancer histological image classification. We designed, trained, and evaluated different CNN architectures, validated the effectiveness of data augmentation and transfer learning, and deployed the most performant model as a web service accessible via Google Cloud Run. The project provided valuable practical experience across the entire lifecycle of a Deep Learning project, from the research and development phase to production deployment.

Bibliography / Webography

- *Dataset of breast cancer histology images (BreaKHis)*. https://www.kaggle.com/datasets/ambarish/breakhis?select=BreaKHis_v1
- Official TensorFlow Documentation. (Accessed April 2025). Available at: https://www.tensorflow.org/
- Official Keras Documentation. (Accessed April 2025). Available at: https://keras.io/
- Official Flask Documentation. (Accessed April 2025). Available at: https://flask.palletsprojects.com/
- Official Docker Documentation. (Accessed April 2025). Available at: https://www.docker.com/
- Official Google Cloud Run Documentation. (Accessed April 2025). Available at: https://cloud.google.com/run/docs
- Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556. (Reference for VGG, a classic CNN architecture)
- He, K., et al. (2016). *Deep residual learning for image recognition*. Proceedings of the IEEE conference on computer vision and pattern recognition, 770-778. (Reference for ResNet)
- Huang, G., et al. (2017). *Densely connected convolutional networks*. Proceedings of the IEEE conference on computer vision and pattern