Technical documentation

1. Project overview

This project presents a deep learning-based system for automatic melanoma detection from dermoscopic images, built using convolutional neural networks (CNNs) and optimized for fast, lightweight inference with ONNX Runtime.

The system is trained and evaluated on a mix of ISIC 2020 and 2019 Challenge datasets, using multiple pretrained CNN architectures (SqueezeNet1_1, MobileNetV3, and ShuffleNetV2). After extensive training and evaluation, the models are exported to the ONNX format for efficient inference.

The project includes:

- A training pipeline.
- A preprocessing pipeline which uses padding, CLAHE (Contrast Limited Adaptive Histogram Equalization) (Ali M. Reza, 2004) and resizes the images into a more compact format.
- An inference script that performs ensemble prediction via voting across multiple ONNX models. This inference script has support for inference on folders of images, with outputs saved in CSV format.
- A Streamlit web application for real-time melanoma prediction through a user-friendly interface.

The goal of this project is to build a modular, reproducible, and efficient pipeline for research and deployment in medical image classification tasks, focusing on melanoma detection as a representative use case.

2. Project structure

The project consists of 3 main parts:

1. The training pipeline
2. The inference script
3. The Streamlit web application

2.1. Training pipeline overview

To address the task of melanoma detection from dermoscopic images, we have built a modular and reproducible training pipeline leveraging PyTorch Lightning, Hydra, and TensorBoard. This setup enables clean abstraction between configuration, training logic, and data processing, while also ensuring consistent reproducibility across training runs and experiments.

2.1.1. Technologies Used

- PyTorch Lightning is used to abstract boilerplate code and structure the training loop, callbacks, and logging, offering a more readable and manageable interface for model training.
- Hydra handles dynamic configuration management, allowing for modular YAML-based instantiation of objects like models, datamodules, loggers, and training strategies through structured and override-friendly configuration files.
- TensorBoard provides real-time logging of metrics such as loss, accuracy, and learning rates during training and evaluation, enabling better insight into model performance.
- Reproducibility is ensured through Hydra's support for fixed seeds and versioned configuration files, and by PyTorch Lightning's deterministic training features.

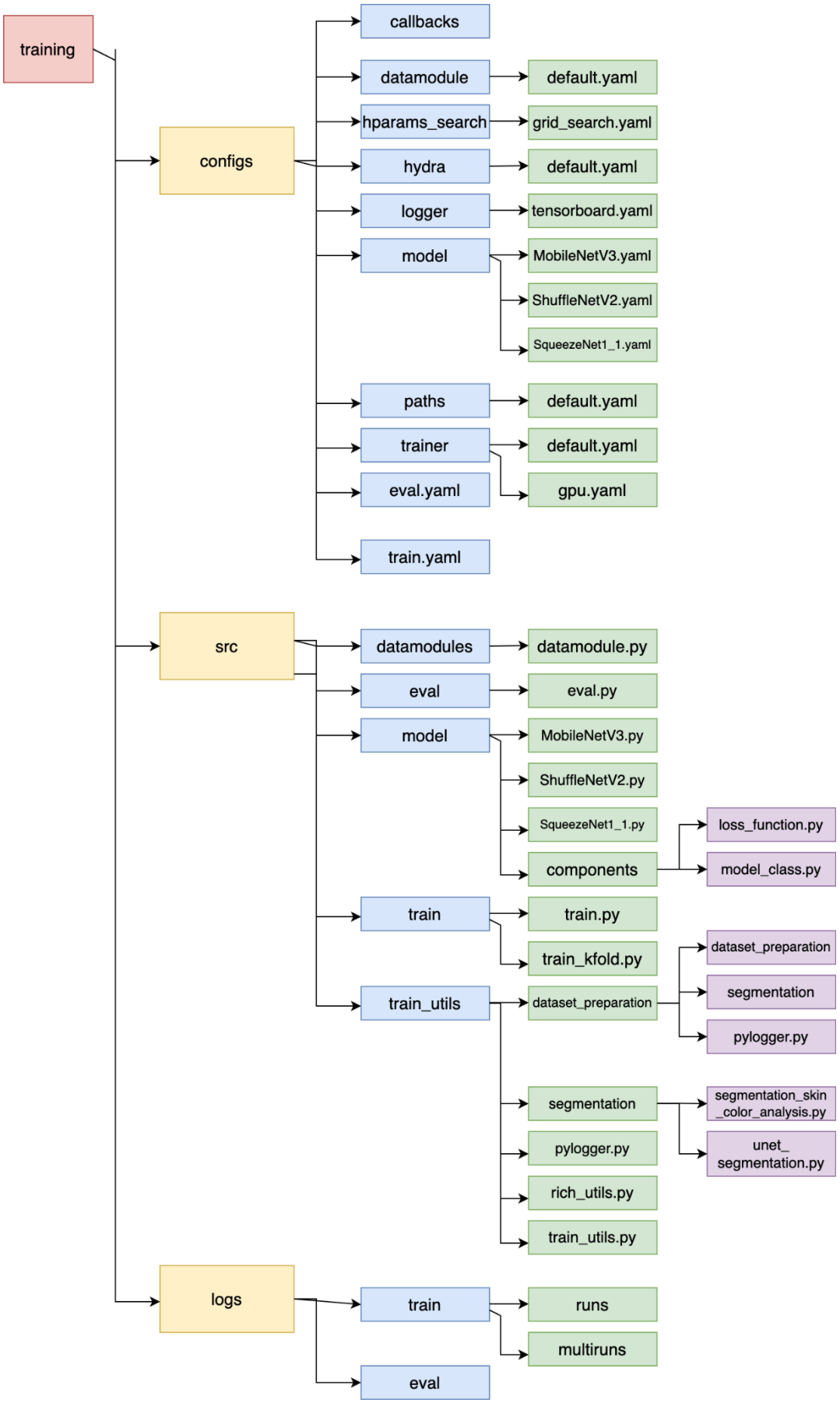
2.1.2. Folder Structure

The pipeline is organized into the training/ directory, which includes two major subdirectories:

1. configs/
2. src/

There is also a third one, which is logs/, but this one is generated upon running training/evaluation. The entire project structure is shown in Figure 1.

Figure 1. Folder Structure



2.1.2.1. configs/z

This folder contains all YAML configurations for Hydra-based object instantiation. It is organized into multiple subfolders, each dedicated to a specific component of the pipeline:

- callbacks/: Configuration for model checkpointing, early stopping, and other training callbacks.
- datamodule/: Settings for dataset-related parameters such as data paths, batch size, and augmentation flags.
- hparams_search/: Templates for hyperparameter sweeps and optimization.
- hydra/: Configuration of Hydra's runtime behavior, including output directory structure and job logging.
- logger/: Definitions for loggers, primarily for TensorBoard.
- model/: Configuration for selecting and initializing different model architectures (e.g., MobileNetV3, ShuffleNetV2, SqueezeNet1_1). This also includes hyperparameters.
- paths/: File system paths used throughout the pipeline.
- trainer/: PyTorch Lightning Trainer configuration, such as minimum and maximum number of epochs, device accelerators, frequency of validation checking and determinism.

Two key files aggregate all of the above configurations:

- train.yaml: The main configuration entry point for training.
- eval.yaml: Configuration for model evaluation.

2.1.2.2. src/

This directory contains the implementation code that drives training, evaluation, data preparation, and utilities.

- datamodules/: Contains the LightningDataModule implementation used by PyTorch Lightning. This module handles dataset splits, augmentations, and DataLoader creation.
- eval/: Contains the evaluation script (eval.py) that runs evaluation using a trained model, with Hydra integration for modular configuration.
- models/: Hosts all model architecture classes along with a shared base class that defines common behaviors such as forward pass logic and loss calculation. Specific architectures supported include MobileNetV3, ShuffleNetV2, and SqueezeNet1_1.
- train/: Includes train.py for standard training and train_kfold.py for training using k-fold cross-validation.
- train_utils/: Provides:

- Utility functions for PyTorch Lightning and Hydra integration.
- Dataset preparation scripts for:
    - Downloading the raw dataset.
    - Extracting the archive contents.
    - Formatting and organizing data into a directory layout compatible with training and validation procedures.

2.1.2.3. logs/

In the logs folder, the runs for training and evaluation are stored as datetime folders from when the run was started. This is also where the Tensorboard log files are stored, respective to each run of training/evaluation.

2.1.3. Implementation and Integration

Each component is designed to work independently and harmoniously with others via Hydra-based object instantiation. For example:

- Switching from one model architecture to another involves changing just a single line in train.yaml.
- Modifying batch size or data augmentations is isolated to the datamodule/ config.
- Running evaluation or training with different seeds, models, data directories, loss functions can be easily managed via command-line overrides.

Training logs and metrics are saved in a versioned directory structure, with each experiment reproducible by referencing its configuration snapshot saved automatically by Hydra.

3. Training Pipeline

3.1. Dataset acquisition and preparation

Dataset acquisition and preparation consists of two scripts:

1. get_data.py
2. dataset_preparation.py

The former serves in getting the data and formatting it in a way which is appropriate for the dataset_preparation.py script.

The dataset_preparation.py script fulfills the following functionalities:

1. Resizes images to the desired dimensions.
2. Splits data into training, validation, and test sets using options like K-Fold or fixed ratios.
3. Applies optional preprocessing steps such as CLAHE enhancement, hair removal, and padding.

3.2. Data augmentation and preprocessing techniques

3.2.1. Augmentations

The following augmentations are applied to enhance model generalization (Shorten & Khoshgoftaar, 2019):

Table 1 Augmentation explanations

Augmentation	What it Does	Why it is Used in This Task
Random Horizontal Flip	Flips the image horizontally with a certain probability.	Lesions can appear on either side of the body; this helps the model generalize better.
Random Rotation	Rotates the image randomly within a 15-degree range.	Helps the model become invariant to slight angular differences in lesion orientation.
Color Jitter	Randomly changes brightness, contrast, saturation, and hue.	Simulates diverse lighting conditions often present in dermoscopic images.
Random Affine	Applies small translations and affine transformations.	Mimics variations in camera position and angle during image capture.
Random Vertical Flip	Flips the image vertically with a certain probability.	Augments limited data with mirrored versions to avoid positional bias.
Random Adjust Sharpness	Randomly sharpens or blurs the image.	Improves robustness to image sharpness variations from different capture devices.

Augmentation	What it Does	Why it is Used in This Task
Random Perspective	Distorts the image using a perspective transformation.	Simulates natural 3D distortion, improving spatial generalization.

3.2.2. Preprocessing techniques

Preprocessing techniques are particularly beneficial in the context of melanoma classification because dermoscopic images can vary widely in terms of lighting, orientation, and background noise. Techniques like CLAHE improve contrast in low-light areas, making subtle lesion features more distinguishable. Padding ensures that important lesion features are preserved without distortion, allowing the model to focus on relevant regions. Hair removal is crucial in medical imaging because body hair can obscure lesions, potentially confusing the model. Overall, preprocessing enhances image quality, standardizes input, and reduces noise, allowing the model to focus on the most relevant features for melanoma detection. This ultimately improves the model's ability to generalize and make accurate predictions in real-world settings.

It’s important to maintain consistency in the preprocessing pipeline, across dataset preparation and later inference is crucial for ensuring that the model sees the data in the same way during both training and evaluation. If preprocessing differs between these stages, the model might learn features in one context that don't generalize well when applied to new, unseen data. For example, if augmentations or preprocessing steps like CLAHE or padding are applied differently during inference than during training, the model may encounter data that deviates from what it was trained on, leading to a drop in performance. By ensuring consistency in preprocessing across both phases, the model is more likely to recognize patterns it learned during training, resulting in more reliable and robust predictions.

The following preprocessing techniques have been used in our task:

CLAHE enhances local contrast in dermoscopic images by applying histogram equalization in small image regions (tiles), limiting noise amplification. This is particularly beneficial in medical imaging, where subtle variations in lesion texture and structure are clinically relevant. CLAHE improves feature visibility in low-contrast regions, aiding better model discrimination between malignant and benign lesions.

Padding is used to maintain the aspect ratio of lesions when resizing images to the input size required by neural networks (e.g., 224×224). Instead of distorting lesions through resizing, padding adds borders to the image to preserve anatomical proportions. This reduces spatial distortion and ensures that lesion features remain consistent and centered, which improves model generalization.

Hair removal is used in this task because dermoscopic images often contain occlusions from body hair, which can introduce noise and confuse learning algorithms. Hair removal techniques, in our case the Dull-Razor algorithm, is applied to detect and inpaint hair regions. The algorithm has promising performance in hair removal, but it produces a large blur in most of the image. For that reason it was left out of the final preprocessing pipeline.

3.3. Model architectures

The models that were used in this task are MobileNetV3, ShuffleNetV2 and SqueezeNet1.1. These models were picked because of their efficiency in execution, making them great for running on edge or mobile devices.

3.3.1. MobileNetV3

MobileNetV3 (Howard et al., 2019) is part of the MobileNet family of networks designed for mobile and embedded devices. It is optimized for both accuracy and efficiency by using a combination of depthwise separable convolutions and lightweight attention mechanisms (like the Squeeze-and-Excitation block). MobileNetV3 comes in two variants: MobileNetV3-Large and MobileNetV3-Small, with the large variant being optimized for better accuracy and the small variant for lower latency and fewer computations. The architecture uses a novel "h-swish" activation function and includes optimizations based on platform-specific searches.

3.3.2. ShuffleNetV2

ShuffleNetV2 (Ma et al., 2018) is an improved version of the ShuffleNet architecture, designed to be even more efficient than its predecessor. The key feature of ShuffleNetV2 is its use of channel shuffle operations, which reduce the computational cost of convolutional layers while maintaining performance. By improving the design of depthwise separable convolutions and introducing lightweight block structures, ShuffleNetV2 achieves faster inference speeds and higher efficiency compared to other networks with similar performance. The architecture includes a carefully designed building block that balances speed and accuracy, making it particularly suitable for mobile and edge devices.

3.3.3. SqueezeNet1.1

SqueezeNet (Iandola et al., 2016) is a lightweight convolutional neural network (CNN) designed to achieve AlexNet-level accuracy with fewer parameters, making it efficient for use in resource-constrained environments. The key innovation of SqueezeNet is the use of "fire modules," which consist of a squeeze layer (with 1x1 convolutions) followed by an expand layer (with both 1x1 and 3x3 convolutions). This structure drastically reduces the number of parameters while maintaining high accuracy. SqueezeNet 1.1 improves upon the original SqueezeNet by further reducing the model size, making it even more efficient for tasks where memory and computation power are limited, such as mobile and embedded devices. This network is well-suited for tasks where model size and speed are more critical than absolute accuracy.

3.3.4. Summary

In summary, these networks (MobileNetV3, ShuffleNetV2 and SqueezeNet 1.1) offer benefits in terms of efficiency, speed, and accuracy, making them good candidates for the melanoma detection tasks, especially if inference needs to be performed in constrained environments.

3.4. Training strategy

3.4.1. K-fold vs training on the entire set

When doing the stratified k-Fold training (Anguita et al., 2012), we split the dataset into 5 folds, each consisting of training, validation and an independent test set. This is to check that the performance over each fold is consistent, and that the hyperparameter combination is in line with expectations. When the on average best performing combination of hyperparameters is found, we can train the models on the entire training set, in order to get the full power of our training. We also use the k-Fold trained models in the ensemble. The purpose of taking models from the k-Fold training is to get models trained/validated on different data. This will in turn make our decision-making in inference more robust.

3.4.2. Hyperparameters

For training the architectures, we used the Adam optimizer with a set initial learning rate and an initial weight decay to prevent overfitting. To improve training stability and convergence, in some iterations of the model training, we applied layer freezing, allowing the model to retain pretrained low-level features while fine-tuning the classifier for the melanoma detection task.

Learning rate scheduling was handled using the ReduceLROnPlateau scheduler, which monitors the validation loss and reduces the learning rate by a factor of 0.1 when no improvement is observed for 5 consecutive epochs. This helps prevent stagnation during training and ensures more precise convergence, especially in the later stages of optimization.

3.4.3. Loss functions

Two loss functions were considered: Cross-Entropy Loss and Focal Loss (Mukhoti et al., 2020). While Cross-Entropy is a standard choice for classification tasks, Focal Loss was adopted as the default due to its effectiveness in handling class imbalance—an inherent issue in melanoma detection where malignant samples are often underrepresented.

Focal Loss dynamically scales the contribution of each example, focusing more on hard-to-classify instances and less on easy ones. This leads to improved sensitivity towards minority classes (malignant lesions), ultimately boosting model robustness in real-world deployment scenarios.

3.5. Logging using Tensorboard

During training, TensorBoard is used for real-time logging and visualization of key metrics such as training and validation loss, accuracy, AUC, learning rate changes, etc. These metrics are logged using PyTorch Lightning's built-in logging mechanism, which integrates seamlessly with TensorBoard and automatically records metrics at the end of each epoch or step.

This logging setup enables:

- Transparent training progress monitoring, allowing you to visually inspect learning curves and detect potential issues like overfitting or vanishing gradients.

- Comparative analysis between multiple training runs or models, especially useful during hyperparameter tuning and architecture selection.
- Convenient debugging by tracking how the model responds to learning rate changes or loss fluctuations.

By using TensorBoard, experiment tracking becomes systematic and visually intuitive, which is critical for training workflows and model selection.

4. Evaluation

4.1. Metrics in evaluation

The metrics that are used to evaluate models are described in the following table.

Table 2 Metrics for model traning and evaluation

Metric	Description
Accuracy	Measures the overall proportion of correct predictions out of all samples. While useful, it can be misleading in imbalanced datasets.
Precision	Indicates how many of the predicted positives are true positives. High precision reduces false positives, which is important when false alarms (e.g., benign labeled malignant) are costly.
Recall (Sensitivity)	Measures how many actual positives are correctly identified. Important for detecting as many true melanoma cases as possible.
F1 Score	Harmonic mean of precision and recall. Useful when you need a balance between false positives and false negatives.
Specificity	Measures the proportion of actual negatives correctly identified. Critical for ensuring benign cases are not misclassified as malignant.
AUROC (Area Under ROC Curve)	Measures the model's ability to distinguish between classes across all thresholds. Robust to class imbalance and often used in medical diagnostics.
AUPRC (Saito & Rehmsmeier, 2015)	Particularly useful in imbalanced datasets, focusing on performance for the positive class. Helps evaluate model's effectiveness at ranking true positives higher.
Mean Loss	Tracks average loss per epoch. Provides an objective measure of how well the model fits the data.
Confusion Matrix	Summarizes true positives, false positives, true negatives, and false negatives. Offers insight into class-specific errors.
ROC AUC Best	Keeps track of the best validation AUROC achieved during training. Used for checkpointing the best model.

4.2. How to interpret the results

In the case of most of these metrics (apart from loss), the rule is simple, the higher the better. But comparing metrics through the entire training process is crucial as well, so we can ensure that models are properly converging. We'll give a visualization of what a good training looks like and what a poor one is like.

Below is a good training/validation loss example is shown.

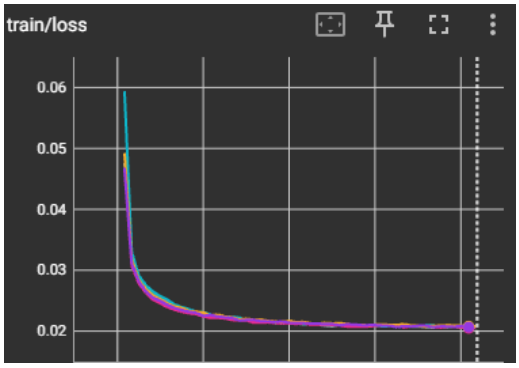


Figure 2 Training loss function during epochs

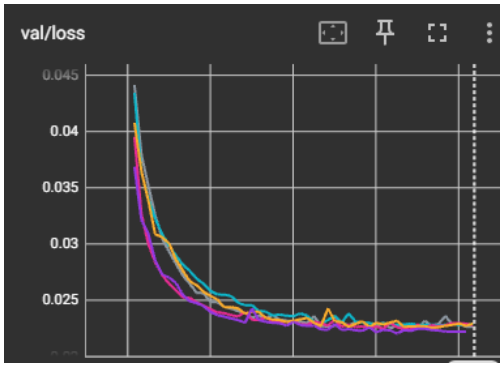


Figure 3 Validation loss function during epochs

As can be seen, the loss functions are in sync with each other, going through the epochs. A bad example would be where they diverge or be unreasonably unstable, indicating overfitting (in the case where training loss is good, but validation loss is diverging) or underfitting (where both are inconsistent across epochs).

The confusion matrices are easy to interpret, just by taking a look at the individual cells, it will tell you about the percentage and count of True Positives, True Negatives, False Positives and False negatives.

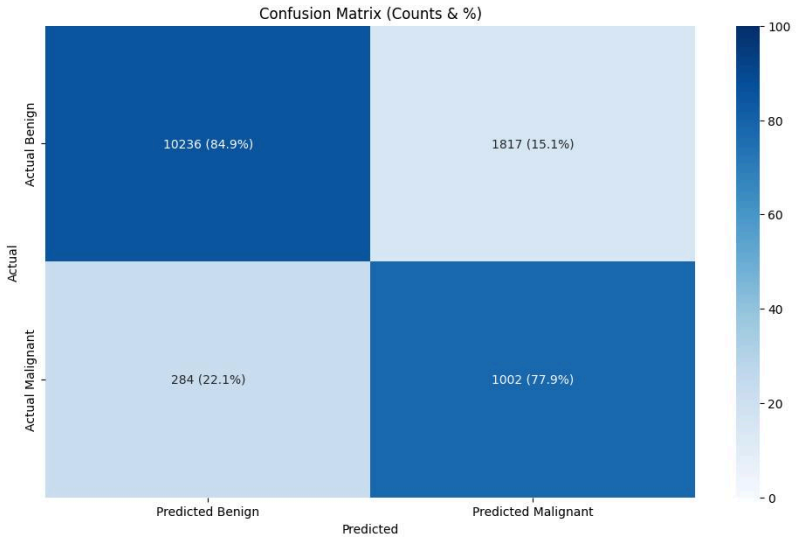


Figure 4 Example confusion matrix

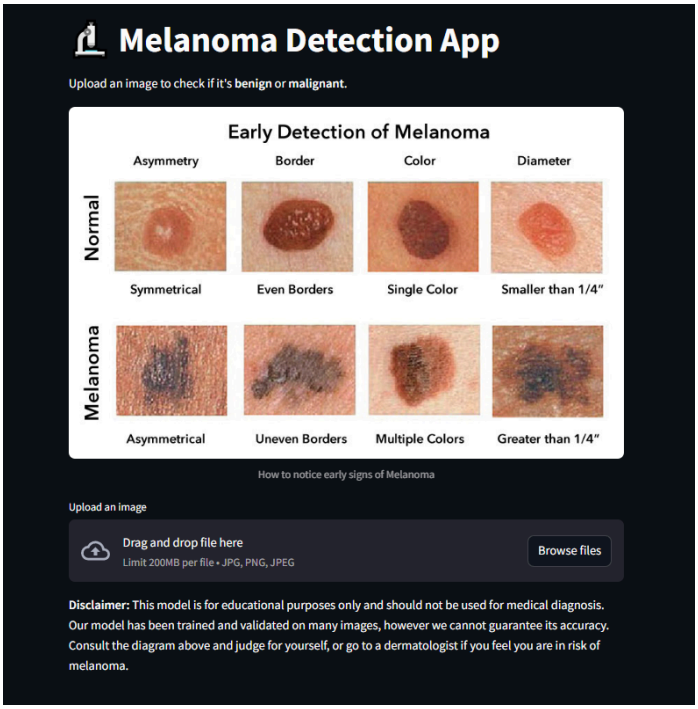
5. Inference

The inference script is designed to run predictions on a batch of input images using one or more ONNX models stored in a specified folder. Each image undergoes consistent preprocessing that includes resizing (with padding) and CLAHE contrast enhancement before being normalized and converted into a tensor compatible with the models. The script loads all available ONNX models and runs inference on each image using every model. The outputs from the models can be aggregated using either majority voting (based on predicted classes) or soft voting (averaging predicted probabilities). Based on the selected voting strategy, the final prediction for each image is determined as either "benign" or "malignant" and saved into a CSV file. Optionally, the preprocessed versions of the input images can also be saved for inspection. The script is configurable via command-line arguments, allowing users to set the input directory, model folder, output CSV path, and whether to use soft voting or save processed images.

6. Deployment on Streamlit

The Streamlit app provides a simple and interactive web interface for melanoma detection using a pretrained ONNX model. When a user uploads an image of a skin lesion, the app processes it in real time—resizing it to the expected input size and applying standard normalization to match the model's training distribution. The ONNX model is loaded using the onnx-runtime backend, and inference is performed by passing the image tensor to the model. The model outputs a prediction in terms of class probabilities, which are then converted to a label ("Benign" or "Malignant") and a confidence percentage using PyTorch's softmax. The result is displayed back to the user with a corresponding message, along with visual feedback. Deployment is handled directly through Streamlit, meaning the app can be launched locally or hosted on platforms like Streamlit Cloud without needing a separate web server or front-end code. This setup makes it easy to share and demo the model, and allows non-technical users to interact with it in a visually guided, user-friendly way.

Figure 5 Streamlit app UI



7. References

1. Reza, A. M. (2004). Realization of the contrast limited adaptive histogram equalization (CLAHE) for real-time image enhancement. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 38*(1), 35–44. https://doi.org/10.1023/B:VLSI.0000028532.53893.7a

2. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data, 6*(1), 60. https://doi.org/10.1186/s40537-019-0197-0

3. Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., & Adam, H. (2019). Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 1314–1324). https://doi.org/10.1109/ICCV.2019.00140

4. Ma, N., Zhang, X., Zheng, H.-T., & Sun, J. (2018). ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 116–131). https://doi.org/10.1007/978-3-030-01264-9_8

5. Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5MB model size. *arXiv preprint* arXiv:1602.07360. https://arxiv.org/abs/1602.07360

6. Anguita, D., Ghelardoni, L., Ghio, A., Oneto, L., & Ridella, S. (2012). The 'K' in K-fold cross validation. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)* (pp. 441–446). i6doc.com. http://www.i6doc.com/en/livre/?GCOI=28001100967420

7. Mukhoti, J., Kulharia, V., Sanyal, A., Golodetz, S., Torr, P. H. S., & Dokania, P. K. (2020). Calibrating deep neural networks using focal loss. In *Advances in Neural Information Processing Systems, 33* (NeurIPS 2020). https://proceedings.neurips.cc/paper_files/paper/2020/file/2ba61cc3f986b21dff0e6e8b1fe26d43-Paper.pdf

8. Saito, T., & Rehmsmeier, M. (2015). The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE, 10*(3), e0118432. https://doi.org/10.1371/journal.pone.0118432