Lumen Data Science

Team: MedMax

Technical documentation

Haralović Marko

Štolfa Duje

A technical report submitted as part of the solution for the Lumen Data Science task in year 2024./2025.

April 2025

Contents

Chapter 1

Introduction

This technical documentation provides an overview of our melanoma classification pipeline, developed as part of the Lumen Data Science COMPETITION during the 2024/2025 academic year. It is intended to serve both as a user guide and a developer reference for understanding, executing, and reproducing our solution.

The report is structured to walk the reader through every aspect of the system—from environment setup and modular code architecture, to training, evaluation, and deployment workflows. We explain how to use the CLI interface, modify configurations, and run trainings. Special attention is given to environment reproducibility using Docker and Conda, and to the modular design of the training components (data loading, model configuration, loss functions, and evaluation metrics).

Finally, we include a detailed reproducibility checklist—giving the exact configuration, system setup, and commands required to reproduce our best-reported results.

Chapter 2

Project Structure and Organization

In this section, we describe the technology stack used for the project, how the codebase was organized to ensure modularity, how multi-GPU training was enabled, and how our custom DataLoader handles image loading and preprocessing. We also provide usage details for key components of the code and explain how its functionality can be easily extended.

2.1 Technology Stack

We used Python 3.10 for this project, `conda` for environment setup as a package manager, and `PyTorch`[1] as the main deep learning library. We used `submitit`[2] for submitting jobs to `Slurm`, `Docker`[3] for code containerization, and `tmux`[4] for running code in detached terminals and avoiding SSH interruptions. `GitHub` was used for version control and collaboration, while `Kaggle`[5] was used for initial model exploration.

Other important libraries used include `kneed`[6] for detecting the knee point of a function (useful for ITA prediction), `fairlearn`[7] for fairness metric calculation, `torchsampler`[8] for efficient oversampling, `timm`[9] for model creation, `seaborn`[10] and `matplotlib`[11] for visualization, and `tensorboardX`[12] for experiment logging. Export to `ONNX`[13] and `TorchScript`[14] is supported and described in a later section.

We also used `HuggingFace`[15] to deploy our model and our code, making our model available for inference using Hugging Face transformers library.

[1] https://pytorch.org
[2] https://github.com/facebookincubator/submitit
[3] https://www.docker.com
[4] https://github.com/tmux/tmux
[5] https://www.kaggle.com
[6] https://github.com/arvkevi/kneed
[7] https://fairlearn.org
[8] https://github.com/ufoym/imbalanced-dataset-sampler
[9] https://github.com/huggingface/pytorch-image-models
[10] https://seaborn.pydata.org
[11] https://matplotlib.org
[12] https://github.com/lanpa/tensorboardX
[13] https://onnx.ai
[14] https://pytorch.org/docs/stable/jit.html
[15] https://huggingface.co/

2.2 ISIC 2020 Dataset

We use the official ISIC 2020 classification dataset[16], which includes dermoscopic images labeled as benign or malignant. The dataset must be downloaded and structured in the following format:

```
/path/to/isic2020_challenge/
  train/
    class1/
      img1.jpeg
    class2/
      img2.jpeg
  val/
    class1/
      img3.jpeg
    class2/
      img4.jpeg
  masks/ #optional, dataloader works without them.
    train/
      class1/
        mask1.png
      class2/
        mask2.png
    val/
      class1/
        mask3.png
      class2/
        mask4.png
```

In addition to the raw images, we provide a CSV file with following required fields: `image_name`, `group`, `target`, `group_str`, and `split`. This file enables metadata-based grouping (e.g., by skin tone) and train/validation splitting.

Utilities. To prepare the dataset, we provide the following scripts:

- `preparer/move.py` – for moving images to the correct class folders.
- `preparer/prepare_split.ipynb` – an notebook for assigning splits and verifying data organization.
- `src/scripts/generate_skin_masks.py` – script for generating segmentation masks based on skin detection, given a source path.

For reproducibility, the metadata CSV file used during training is also included in our GitHub repository.

2.3 Code Structure

The project is organized in a modular and scalable structure. Below is a simplified directory tree (up to depth 3) that outlines the key components.

[16] https://challenge.isic-archive.com/data/

```
        configs/                          # Model-specific training
configs
            convnext_base.py
            convnext_large.py
        docker/                           # Dockerfiles for
containerization
            Dockerfile
            Dockerfile_cpu
        environment.yml                   # Conda environment definition
        isic2020_challenge/               # Dataset: splits, masks, and
labels
            ISIC_2020_full.csv
            masks/
                train/
                valid/
            train/
                benign/
                malignant/
            valid/
                benign/
                malignant/
        melanoma_classifier_output/       # Training logs, configs, and
checkpoints
        melanoma_train.py                 # Main training script
        melanoma_eval.py                  # Evaluation script
        notebooks/                        # Exploratory notebooks
            skin_tone_analysis.ipynb
            skin_tone_estimation.ipynb
        preparer/                         # Data preprocessing scripts
            move.py
            prepare_split.ipynb
        src/                              # Core project modules
            datasets/
                data_processing.py
                datasets.py    # custom datasets
                sampler.py     # custom samplers
            engine/
                engine.py      # train and test logic
                scheduler.py
            evaluation/
                metrics.py     # metric definition
            models/
                backbones/     # models used for feature extraction,
e.g. ConvNeXt, DinoV2
                layers/        # layers needed for models
initialization
                losses/        # loss function definitions
                melanoma_classifier.py # our classification model
wrapper
                optim_factory.py     # optimizer factory
                utils/
            scripts/
                estimate_ita.py             # script for ITAS
estimation from images
                generate_skin_masks.py      # mask generator script
            utils/
                argparser.py       # argument parsing utilities
                distributed.py     # distributed training utilities
                logging_utils.py   # logging utilities
                utils.py           # model saving logic utilities,
etc.
```

```
53                 visualization/
54                     visualize_images.py
55             test.sh                          # Testing entrypoint
56             train.sh                         # Training entrypoint
57             weights/                         # Model weights directory
```

Listing 2.1: Project Directory Structure

2.4 Modular Design

The main idea behind the code is modularity. We wanted to iterate quickly and try different experiment setups, which required from us to build a code that is easy to maintain and even easier to extend.

In the following sections, we will go through main components of our system, focusing on how our custom DataLoader operates, how we made it possible to add a completely new architecture with few lines of code, how we managed to handle multiple options for samplers, optimizers, losses, loss weights, metric calculations,etc.

2.4.1 Dataloader design

`LocalISICDataset`, as we named it, is a highly modular PyTorch `Dataset` implementation designed specifically for the ISIC 2020 skin lesion dataset. It supports advanced preprocessing, skin tone–aware augmentation, and segmentation-based manipulation of images, as we wanted to explore whether and how colorspace (RGB vs LAB), augmentations (skin color transformations), segmenting out skin, and oversampling helps our model to perfrom better.

Key Features:

- **Input configuration:** Supports directory-based dataset structure with separate folders for `train/test`, `benign/malignant` images and corresponding segmentation masks. Masks may only be used/present if we want masking out skin and learning only on the masked images.
- **Skin tone integratio:** When a `skin_color_csv` is provided, the loader incorporates:
  - ITA (Individual Typology Angle)
  - Fitzpatrick skin scale
  - Group labels for skin-type-aware augmentation
- **Oversampling with augmentations:** For malignant samples, user-defined augmentations (e.g., rotations, brightness shifts) can be selectively applied. The loader supports sampling with an oversampling ratio derived from the number of augmentations. This was used to artificially increase the number of positives.
- **Color space transformation:**
  - Standard RGB pipeline
  - Optional conversion to CIELAB color space
  - Group-specific pixel-level transformations to simulate skin tone shifts (“skin-former” mode) - idea is to mask out the skin from the image and with certain probability to darken it, as most of the skin color types are of lighter colors.

- **Mask-based processing:** If `segment_out_skin=True`, the loader uses segmentation masks to isolate lesions and mask out all the other pixels.

Return Values: Each call to `__getitem__` returns a triplet:

```
(image: Tensor, label: int, group: int)
```

where `group` encodes the patient's skin tone classification, and is used for group-aware training and fairness analysis.

2.4.2 Classifier design

The classification model is implemented in `src/models/melanoma_classifier.py`. It is structured as a modular wrapper around backbone feature extractors, enabling flexibility in switching between different model architectures. Each backbone is followed by a classification head — a linear layer — making the overall model suitable for 2-class melanoma classification.

Skin Tone Group-Aware Extension. To support fairness-aware training, the classifier can optionally be extended to produce `num_classes` $\times$ `num_groups` outputs. This is used when performing group-aware loss computation (e.g., skin tone–sensitive learning), and allows the same model to adapt to fairness constrained tasks.

Backbone Support. The classifier supports the following pretrained architectures:

- **ConvNeXt** – using `create_convnext_model`[17]
- **ConvNeXtV2** – using `create_convnext_v2_model`[18]
- **EfficientNetV2** – using `crete_efficientnet_v2_model`[19]
- **DINOv2** – through `create_dinov2_model`[20]

Each model returns either a classification token (e.g., in ViT/DINOv2) or pooled feature vector (e.g., in CNNs), which is then passed to a fully connected `nn.Linear` head layer. The head is dynamically adjusted to match the number of target classes.

Training Flexibility. The classifier accepts parameters to control:

- Whether to use pretrained weights
- Whether the backbone was trained on ImageNet-22K (via `in_22k`)
- Whether to freeze the backbone and only fine-tune the head (for linear probing)

The model is used in both standard and fairness-aware training setups. To add a new backbone, one must define a custom `create_model_x` function inside `src/models/model_x.py`, and extend the model selection logic in `src/models/melanoma_classifier.py` to include the corresponding `model_name`.

[17] https://arxiv.org/abs/2201.03545
[18] https://arxiv.org/abs/2301.00808
[19] https://arxiv.org/abs/2104.00298
[20] https://arxiv.org/abs/2304.07193

The logic can be summarized as the following pseudocode:

Algorithm 1 Backbone Selection in `MelanomaClassifier`

function MELANOMACLASSIFIER(*model_name*, *num_classes*, *pretrained*, *in_22k*, *freeze*)
    if *model_name* contains "convnext_" **then**
        Load ConvNeXt backbone
        Replace head with `Linear(num_features, num_classes)`
    else if *model_name* contains "efficientnet" **then**
        Load EfficientNetV2 with specified parameters
    else if *model_name* contains "convnextv2" **then**
        Load ConvNeXtV2 with specified parameters
    else if *model_name* contains "dinov2" **then**
        Load DINOv2 backbone
        Append `Linear(num_features, num_classes)` to CLS output
    else
        Raise error: Unsupported model
    end if
    return wrapped model
end function

2.4.3 Criterion design

Inside `src/models/losses/criterion.py`, we define several custom loss functions: `OhemCrossEntropy`, `RecallCrossEntropy`, `DomainIndependentLoss`, `DomainDiscriminativeLoss`, and `FocalLoss`, along with support for `InverseFrequencyWeighting`.

The choice of loss function and whether to enable inverse frequency weighting is controlled via command-line arguments, making it easy to experiment with different training objectives. This modular design also allows us usage of domain-aware and domain-independent strategies during training and evaluation. Again, to add a criterion, one should add a class that inherits `nn.Module` with forward method and add a command line argument, and the code would work for both binary and domain aware classification.

2.4.4 Sampler design

Inside `src/models/losses/sampler.py`, we define two custom data samplers: `BalancedBatchSampler` and `UnderSampler`. These are built by subclassing PyTorch's `Sampler` class and are used to mitigate class imbalance during training.

BalancedBatchSampler. This sampler ensures that all classes are equally represented within each batch. It works by oversampling underrepresented classes to match the size of the majority class.

UnderSampler. This sampler performs dataset-level under-sampling by keeping all samples from the minority class and randomly sampling a fraction (controlled via `under_sample_rate`) of samples from majority classes.

Modularity. Both samplers operate independently of any specific dataset class, but expect the dataset to either expose a `get_labels()` method or return labels in the form `(image, label, group)` from `__getitem__`.

Samplers can be plugged directly into the PyTorch `DataLoader` via the `sampler=` argument, replacing the default shuffle behavior.

2.4.5 Metric and optimizer setup

Metric calculation differs depending on whether a standard binary classifier is trained or a domain-aware classifier is used. In the latter case, the model produces a `num_classes * num_groups` output, which must be reduced to `num_classes` before evaluation. Metrics can be seamlessly extended to support both training paradigms, and we define several prediction-handling functions in `src/evaluation/metrics.py`.

The optimizer is created using a utility from `src/models/optim_factory.py`, adapted from Facebook's official ConvNeXt repository.

2.4.6 Development workflow

We used GitHub for version control and code management. To maintain consistent code style, we configured `pre-commit` hooks for automatic import sorting and code formatting using `isort` and `black`. These tools are included as optional development dependencies in `requirements_dev.txt` on the main branch.

In addition, we defined a `pyproject.toml` file to centralize formatting rules for both tool, which is automatically applied when the tools are executed via the command line or through the pre-commit.

Throughout development, we maintained a feature-branch workflow, with separate branches for individual features or experiments.

2.4.7 Training and infrastructure

Training was conducted on a remote GPU server, which we accessed via SSH. Each training session was initiated through a shell script that passed command-line arguments to our main Python training script. To ensure uninterrupted execution, we used `tmux`, a terminal multiplexer that allows terminal sessions to be detached and reattached. This made it possible to run long sequences of experiments and monitor them as needed.

To monitor training progress, we implemented a custom logging system that recorded:

- Per-batch and per-epoch training time
- Learning rate and minimal learning rate
- Loss values and per-class accuracy
- Weight decay and memory consumption

Command-line arguments were also saved, and the system supported per-epoch checkpointing, as well as saving the best-performing model based on a target metric such as F1-score or recall for the malignant class. Evaluation results were logged per epoch into a `training.log` file, which allowed us to track model performance across runs.

We also supported mixed-precision training using PyTorch's Automatic Mixed Precision (AMP), which can be enabled with the `-use_amp true` flag. The training pipeline supports both single-GPU and multi-GPU setups via `torch.distributed.launch`, and we extended it with multi-node SLURM support using `submitit`, as provided by the official ConvNeXt repository.

Experiments were executed on a cluster of 8 NVIDIA L4 GPUs, each with 24GB of memory. We stored the following outputs for each run:

- `events.out.tfevents.*` (TensorBoard logs)
- `config.json` (command-line configuration)
- `training.log` (epoch-level metrics)
- `checkpoint_epoch_X.pth` and `best_checkpoint.pth`

2.4.8 ONNX and TorchScript support

To ensure compatibility with various deployment environments and inference frameworks, we support model export in both **TorchScript** and **ONNX** formats.

TorchScript. TorchScript is an intermediate representation of a PyTorch model that can be run independently of Python. We provide the function `convert_to_torchscript` for this:

```
def convert_to_torchscript(model, input_tensor, output_path):
    model.eval()
    scripted_model = torch.jit.trace(model, input_tensor)
    scripted_model.save(output_path)
```

Listing 2.2: TorchScript conversion

Link: https://pytorch.org/docs/stable/jit.html

ONNX. ONNX (Open Neural Network Exchange) is an open format built to represent machine learning models. Export is done via the `export_model_to_onnx` function:

```
def export_model_to_onnx(model, input_tensor, output_path):
    torch.onnx.export(model, input_tensor, output_path, export_params=True, opset_version=11,
                      do_constant_folding=True, input_names=['input'], output_names=['output'],
                      dynamic_axes={'input': {0: 'batch_size'}, 'output': {0: 'batch_size'}})
```

Listing 2.3: ONNX export

Link: https://onnx.ai/

Supporting both formats allows portability and speed during deployment, regardless of the serving infrastructure, which is quite important for our task.

We used onnx and onnxruntime for model inference. We wanted to remove the need for users to know anything about our codebase when using our models. We will discuss this in details in Chapter 5,

Chapter 3

Environment Setup

We ensured our code works reliably by using Conda environments. We separated dependencies into: device-specific packages (e.g., `torch`, `torchaudio`), development dependencies (e.g., `isort`, `black`), and core library requirements.

This approach was further extended using Docker to containerize our code. We created two Dockerfiles—one for GPU and one for CPU execution—depending on the availability of a CUDA-capable device. This separation of concerns was motivated by the fact that, as a two-member team with different hardware, we wanted to avoid creating custom Dockerfiles for each user. The only user-specific setting that needs to be changed is the CUDA version.

Our environment can be created by:

1. Using Conda and the `environment.yaml` dependency file.
2. Using Conda and following the instructions in `INSTALL.md`.
3. Building a GPU or CPU Docker image.
4. Pulling the prebuilt GPU or CPU Docker image from Docker Hub.

In the following sections, we provide detailed instructions on setting up the environment.

3.1 Conda install using environment YAML file

On the main branch of the code, in the root directory, you can find the environment.yml file. The only prerequisite is to have Conda installed, and the environment can be recreated with a single line of code. However, within that YAML file, the user must specify the correct versions of dependencies based on the CUDA and cuDNN versions being used.

```
conda env create -f environment.yml
```

Listing 3.1: Creating Conda environment from `environment.yml`

3.2 Conda install following the instructions from INSTALL.md

For a more controled approach, follow these instructions:

```
# Create and activate Conda environment
conda create -n melanoma python=3.10 -y
conda activate melanoma

# Install dependencies from requirements file
```

```
pip install -r requirements.txt

# For CPU-only installation of PyTorch and related libraries:
pip install \
    torch==2.2.0+cpu \
    torchvision==0.17.0+cpu \
    torchaudio==2.2.0+cpu \
    -f https://download.pytorch.org/whl/cpu/torch_stable.html
```

Listing 3.2: Environment setup using Conda and pip

Note: If you are using a GPU, make sure to install the appropriate version of PyTorch that matches your CUDA version. You can find the correct installation command for your system on the official PyTorch website.

3.3 Docker installation

To build and run the project using Docker, follow the instructions below depending on whether you are using a CPU or a GPU.

CPU Setup

Place yourself in the project's root directory and run the following commands:

```
# Build the Docker image using the CPU Dockerfile
docker build -t melanoma -f docker/Dockerfile_cpu .

# Run the Docker container interactively
docker run -it melanoma
```

Listing 3.3: Build and run Docker container (CPU version)

GPU Setup

If you have a CUDA-compatible GPU and NVIDIA Docker runtime installed, use the GPU-specific Dockerfile:

```
# Build the Docker image using the GPU Dockerfile
docker build -t melanoma -f docker/Dockerfile .

# Run the Docker container interactively
docker run -it --gpus all melanoma
```

Listing 3.4: Build and run Docker container (GPU version)

3.4 Pull prebuilt image from Docker Hub

If you prefer not to build the image locally, you can pull the latest version from Docker Hub. We pushed already buitl images there, both for CPU and GPU devices.

```
# Pull the CPU image from Docker Hub
docker pull haralovicmarko/melanoma_cpu:latest

# Run the container (CPU version)
docker run -it haralovicmarko/melanoma_cpu:latest

```

```
# Run the container (CPU version) with mounted dir (of images and weights
    for example)
docker run -it \
  -v /path/to/local/data:/melanoma-classification/data \
  -v /path/to/local/weights:/melanoma-classification/weights \
  haralovicmarko/melanoma_cpu:latest

# Pull the GPU image from Docker Hub
docker pull haralovicmarko/melanoma_gpu:latest

# Run with GPU support
docker run -it --gpus all haralovicmarko_gpu/melanoma_cpu:latest

# Run with GPU support and mounted directories (again for weights and
    images dir)
docker run -it --gpus all \
  -v /path/to/local/data:/melanoma-classification/data \
  -v /path/to/local/weights:/melanoma-classification/weights \
  haralovicmarko/melanoma_gpu:latest
```

Listing 3.5: Pull and run the Docker image from Docker Hub

Chapter 4

Codebase usage

Our codebase is designed to be controlled via command-line arguments. An argument parser captures all user-provided parameters and initializes relevant classes and components accordingly. The main training entry point is `melanoma_train.py`, which supports both training and evaluation on CPU, single GPU, multi-GPU, and SLURM-based distributed setups using `submitit`.

In addition, we provide a dedicated evaluation script, `melanoma_eval.py`, which includes its own argument parser and can be used independently for model evaluation.

To improve flexibility, we extended the system to support YAML configuration files. Instead of specifying all parameters via command-line arguments, users can pass a `-config` flag with a path to a YAML file. This file is automatically parsed, and its values override the defaults defined in the argument parser.

In the following sections, we provide:

- a description of key arguments,
- a sample bash script for running training and evaluation,
- an example configuration file,
- usage of `torch.distributed` for multi-GPU setups, and
- guidelines on writing custom YAML configuration files.

4.1 Key CLI arguments

Our training and evaluation scripts are fully configurable through command-line arguments. Below we highlight some of the most important options for customizing the training pipeline:

- **Dataset**
  - `--skin_color_csv` *(default: None)*
    Path to the CSV containing image metadata.
  - `--data_path` *(default: ./isic2020_challenge)*
    Path to the images folder.
- **Model configuration**
  - `--model` *(default: convnext_tiny)*
    Backbone architecture to use.

  - `--num_classes` *(default: 2)*
    Number of output classes for classification.
  - `--num_groups` *(default: 1)*
    Number of skin tone groups.
  - `--pretrained` *(default: True)*
    Load pretrained weights (automatically handled).
  - `--freeze_model` *(default: False)*
    Freeze model backbone for linear probing.
  - `--input_size` *(default: 224)*
    Input resolution of training images.

- **Loss function**
  - `--ohem` *(default: False)*
    Use Online Hard Example Mining loss.
  - `--ifw` *(default: False)*
    Apply inverse frequency weighting.
  - `--recall_ce` *(default: False)*
    Use recall-weighted Cross Entropy loss.
  - `--focal_loss` *(default: False)*
    Use Focal loss for class imbalance.
  - `--domain_independent_loss` *(default: False)*
    Ignore group info in loss function.
  - `--domain_discriminative_loss` *(default: False)*
    Separate classes across domain groups.

- **Sampling and class balancing**
  - `--oversample_malignant` *(default: False)*
    Oversample malignant lesions during training.
  - `--undersample_benign` *(default: False)*
    Undersample benign lesions during training.
  - `--undersample_benign_ratio` *(default: -1)*
    Ratio for undersampling benign cases.

- **Preprocessing options**
  - `--cielab` *(default: False)*
    Convert input images to CIELAB color space.
  - `--skin_former` *(default: False)*
    Apply skin tone shifting transformation.
  - `--segment_out_skin` *(default: False)*
    Use skin segmentation to mask background.
  - `--conditional_accuracy` *(default: False)*
    Report per-group conditional accuracy.

- **Training setup**

- `--use_amp` *(default: False)*
  Use PyTorch AMP for mixed precision.
- `--config` *(default: None)*
  Path to YAML configuration file.

These are the most important arguments. All available arguments can be found in the argument parser defined in `src/utils/argparser.py`.

4.2 Example bash script

To simplify training execution,we used bash scripts to run our experiments. Here we provide example bash scripts for both CPU and multi-GPU setups.

4.2.1 CPU setup

This is a bash script designed to run linear probing using pretrained DinoV2 ViT-s/14 on resolution 224x224, with 2 classes, using recall based cross entropy and inverse frequency weighting.

```
python melanoma_train.py \
    --data_path "./isic2020_challenge" \
    --skin_color_csv ".isic2020_challenge/ISIC_2020_full.csv" \
    --model dinov2_vit_small \
    --batch_size 8 \
    --epochs 10 \
    --device cpu \
    --freeze_model True \
    --input_size 224 \
    --num_classes 2 \
    --pretrained True \
    --log_dir "./melanoma_logs" \
    --warmup_epochs 0 \
    --use_amp False \
    --lr 0.01 \
    --weight_decay 0.0001 \
    --update_freq 1 \
    --ifw \
    --recall_ce
```

Listing 4.1: Example CPU Training Script

4.2.2 Multi-GPU setup

For distributed training using multiple GPUs, the following script leverages `torch.distributed.launch`. It automatically detects the environment and configures the device accordingly, and perform same linear probing on the features extracted by DinoV2 ViT-s/14.

```
#!/bin/bash

python -m torch.distributed.launch \
    --nproc_per_node=4 \
    --master_port=29500 \
    --use_env \
    melanoma_train.py \
    --data_path "./isic2020_challenge" \
    --skin_color_csv "./isic2020_challenge/ISIC_2020_full.csv" \
```

```
    --model dinov2_vit_small \
    --batch_size 32 \
    --epochs 10 \
    --device "cuda" \
    --freeze_model True \
    --input_size 224 \
    --num_classes 2 \
    --num_workers 4 \
    --pretrained True \
    --log_dir "./melanoma_logs" \
    --warmup_epochs 0 \
    --use_amp False \
    --lr 0.01 \
    --weight_decay 0.0001 \
    --mixup 0.0 \
    --update_freq 1 \
    --ifw \
    --recall_ce \
    --distributed
```

Listing 4.2: Example Multi-GPU Training Script

These scripts can be easily adapted by changing model architecture, dataset paths, or enabling additional options such as skin tone preprocessing or different loss functions.

4.2.3 Evaluation script

To evaluate a trained model checkpoint, we use the same `melanoma_train.py` entry point with the `-test` flag. Below is an example bash command to run evaluation on a saved model.

```
python melanoma_train.py \
    --data_path "./isic2020_challenge" \
    --skin_color_csv "./isic2020_challenge/ISIC_2020_full.csv" \
    --model dinov2_vit_small \
    --batch_size 8 \
    --device $DEVICE \
    --input_size 224 \
    --num_classes 2 \
    --checkpoint <PATH_TO_CHECKPOINT> \
    --ifw \
    --test
```

Listing 4.3: Example Evaluation Script

This script loads the specified checkpoint and evaluates it on the validation split. Results such as loss, accuracy, and per-group metrics will be logged and saved to the output directory. The use of the `-ifw` flag ensures evaluation loss is consistent with the training configuration.

4.3 Running with configuration files

In addition to command-line arguments, our training pipeline supports YAML-based configuration files for improved reproducibility and cleaner experiment setup.

Usage

To run the training script using a YAML configuration file, pass the file path using the `-config` argument:

```
python melanoma_train.py \
    --config "configs/dino_vit_small.yaml" \
    --data_path "./isic2020_challenge" \
    --skin_color_csv "./isic2020_challenge/ISIC_2020_full.csv" \
    --device $DEVICE \
    --num_workers 4 \
    --log_dir "./melanoma_logs" \
```

Listing 4.4: Run training using a config file

The `-config` file can define any argument accepted by the parser. Command-line arguments provided alongside the config file will override values defined in the YAML file.

Example YAML Configuration

Below is an example of a complete YAML configuration file (e.g., `configs/dino_vit_small.yaml`):

```
data_path: "./isic2020_challenge"
skin_color_csv: "./isic2020_challenge/ISIC_2020_full.csv"
model: "dinov2_vit_small"
batch_size: 8
epochs: 10
input_size: 224
num_classes: 2
num_workers: 4
pretrained: true
log_dir: "./melanoma_logs"
warmup_epochs: 0
use_amp: false
lr: 0.01
mixup: 0.0
update_freq: 1
ifw: true
weight_decay: 0.0001
output_dir: "./melanoma_classifier_output"
```

Listing 4.5: YAML config file example

This approach makes it easy to manage multiple experimental configurations and share setups. Configs can be found inside folder ./configs on the main branch.

Chapter 5

Inference

5.1 Inference script

To enable seamless usage of our models during inference, we converted all of our best-performing models into both ONNX and TorchScript formats using the corresponding converters. For deployment and runtime inference, we opted to use the ONNX format in combination with ONNX Runtime.

The inference script is located at:

`melanoma_classification/predict.py`

Usage: It can be run using this commands:

```
cd melanoma-classification

python predict.py \
    --onnx_model_path /path/to/onnx_model.onnx \
    --input_folder /path/to/images/folder \
    --image_width 224 \
    --image_height 224 \
    --output_csv /path/where/to/save/predictions.csv
```

Listing 5.1: Run ONNX-based inference

Our submission automatically loads the image width and height, as well as the ONNX checkpoint of our best model. To run the model on a folder of images and obtain results, the user must execute the following command:

```
cd melanoma-classification

python predict.py \
    --input_folder /path/to/images/folder \
    --output_csv /path/where/to/save/predictions.csv
```

Listing 5.2: LUMEN submssion inference command

The resulting CSV will contain the following columns:

- `image_name`
- `target`

5.2 Model conversion to onnx

First, navigate to the project root directory:

```
cd melanoma-classification
```

Then, run the script as follows:

```
cd melanoma-classification

python -m src.scripts.export_model \
    --checkpoint_path /path/to/model_checkpoint.pth \
    --model_class MelanomaClassifier \
    --output_dir /path/where/converted/models/are/saved
```

Listing 5.3: Convert model checkpoint to ONNX/TorchScript

Inference progress will be logged, and the resulting predictions should appear

5.3 Hugging Face Inference and Model Access

There are several ways to download and use the melanoma classification models from Hugging Face:

5.3.1 Option 1: Using the `transformers` library

First, install the `transformers` library if you have not already:

```
pip install transformers
```

Listing 5.4: Install Hugging Face Transformers

Then, load the model and feature extractor:

```
from transformers import AutoModelForImageClassification,
    AutoFeatureExtractor

# Load the model
model_name = "Mhara/melanoma_classification"
model = AutoModelForImageClassification.from_pretrained(model_name)
feature_extractor = AutoFeatureExtractor.from_pretrained(model_name)
```

Listing 5.5: Load model and feature extractor from Hugging Face

5.3.2 Option 2: Download a specific file from the Hugging Face Hub

You can download a specific file (such as a model checkpoint) directly:

```
from huggingface_hub import hf_hub_download

model_path = hf_hub_download(
    repo_id="Mhara/melanoma_classification",
    filename="weights/model_0_best.pth",
    repo_type="model"
)

print(f"Model downloaded to: {model_path}")
```

Listing 5.6: Download specific model weight file

5.3.3 Option 3: Manual download via website or Git clone

You can also manually download the files:

- Visit https://huggingface.co/Mhara/melanoma_classification/tree/main/weights
- Click on the specific model file you want to download.
- On the file page, click the download button in the top-right corner.

Alternatively, to download the entire repository:

```
# Install Git LFS
git lfs install

git clone https://huggingface.co/Mhara/melanoma_classification
```

Listing 5.7: Clone the model repository with Git LFS

After cloning, you can access all the model checkpoints locally.

Chapter 6

Reproducibility Guide

To reproduce our results, start by cloning the repository:
https://github.com/MarkoHaralovic/melanoma-classification
Place yourself in the root directory of the cloned repository before running any commands.

6.1 System setup

- **Hardware used**: Most experiments were run on 8 NVIDIA Tesla L4 GPUs. However, we also conducted single-GPU experiments, whose results are reported.
- **Environment setup**: Chapter 3 provides detailed instructions for setting up the Python environment using Conda or Docker, matching the configuration used during training and evaluation.

6.2 Exact configurations

- As shown in Chapter 4, Listing 4.5, we provide an example YAML configuration file and instructions on how to run the project using it.
- The configuration file used for our best-performing model is available in the `./configs` folder and is named `best_run.yaml`.
- To perform evaluation, you can use the same `best_run.yaml` file. Simply modify it by setting `test: True` and provide the path to the checkpoint file via the `checkpoint` parameter.

6.3 Expected output directory structure

- Output files will be saved under the directory specified by the `output_dir` argument, either through the config file or the CLI.
- Logs are saved in a `/logs` subfolder and include:
  - `training.log` – training log + summary of metrics per epoch
  - `log.txt` – detailed training log
  - TensorBoard logs in `events.out.tfevents.*` format
- Checkpoint files are named `checkpoint_epoch_<NUM>.pth`.

- A `config.json` file stores all arguments used during the run for future reference.

6.4 Run the Code

To run the experiment and create our best model, use the following command:

```
python melanoma_train.py --config "./configs/best_run.yaml"
```

To evaluate the newly created model, edit the configuration file by setting the ‘test‘ field to ‘true‘. Then, run:

```
python melanoma_train.py --config "./configs/best_run.yaml" \
                         --checkpoint <PATH_TO_WEIGHTS>
```