CropSegmentation: M5

1. M4 Improvements

1.1 API Documentation Improvement

To enhance the readability and maintainability of the code, as well as to improve the API documentation, comments have been added to each FastAPI endpoint. These comments provide a comprehensive description of the endpoint, its inputs, and outputs. This practice is crucial for facilitating the understanding of our codebase by providing detailed descriptions of parameters and returns in API documentation. This can help bridge the gap between the API creators and API users by providing clear and concise information about the API's capabilities, inputs, outputs, and error handling. Additionally, this improved API documentation can aid in code maintenance, since it makes it easier to update or modify the API as it evolves. After these changes, we re-ran the script to create the HTML file containing the ReDoc API documentation, ensuring that the documentation is up-to-date and accurately reflects the current state of the API.

1.2 Separation of Development and Production Requirements

Additionally, the requirements for the project were separated into development requirements and production requirements, in order to abide by the best practices for requirement definitions in software development. This approach allows for a clear distinction between the dependencies needed for development and those needed for the production environment. It also helps to manage the project's dependencies more efficiently and reduces the risk of unnecessary dependencies in the production environment.

The requirements-dev.txt file includes additional tools necessary for development, such as coverage for code coverage analysis, pylint and flake8 for code linting, and pytest for running tests. The deepchecks package is also included for data integrity checks.

On the other hand, the requirements-prod.txt file lists the packages necessary for the application to run in the production environment. These include packages for data processing (numpy, pandas, scikit-learn, torch, opencv-python), web application (fastapi, uvicorn, python-multipart), and data visualization (matplotlib, matplotlib-inline).

By separating the requirements, it becomes easier to manage the project's dependencies and ensure that only necessary packages are included in the production environment. This approach also makes it easier to update or modify dependencies for development or production independently.

An important aspect to note is that the requirements-dev.txt file imports the requirements-prod.txt file. This means that instead of rewriting the same requirements, the development requirements file reuses the production requirements. This practice is beneficial for several reasons:

- **Clean Code:** It avoids code duplication, which is a fundamental principle of clean code. Duplication can lead to errors and inconsistencies, especially when updates are needed.
- **Modularity:** It enhances modularity as each file has a specific purpose and can be updated independently.
- **Maintainability:** It improves maintainability as changes in the production requirements automatically reflect in the development requirements.
- **Efficiency:** It increases efficiency as there's no need to manually synchronize the two files.

This approach of reusing code aligns with the DRY (Don't Repeat Yourself) principle, a best practice in software development. It contributes to a more manageable, efficient, and error-free project.

2. Backend and Frontend Containerization with Docker

Docker is a platform that enables developers to package and distribute applications in a lightweight, portable manner. This process, known as containerization, encapsulates an application with its environment and dependencies into a container. This ensures that the application works uniformly across different environments. Docker Compose further extends Docker's capabilities by allowing the definition and orchestration of multi-container Docker applications.

During the implementation of Docker and Docker Compose in our project, we followed established best practices in order to utilize containerization benefits to the fullest. For our project, two different Dockerfiles were implemented in order to create two separate containers for the backend and frontend with the goal of creating a scalable, maintainable, and efficient development and deployment workflow.

Below is a detailed description of the backed Dockerfile:

1. **Python Environment Management and Version Control:** The Dockerfile begins by specifying a Python slim base image, an efficient choice due to its smaller size. The Python version is defined as an argument (ARG PYTHON_VERSION=3.11.4), providing flexibility in upgrading or downgrading Python versions without altering the Dockerfile's main structure.

2. **Working Directory Set-Up:** Initially, the working directory is set to the root and later changed to /src/api. This facilitates a clear and organized structure within the Docker container and segregates the application code from other files, enhancing readability and maintainability.

3. **Environment Variables:** Two critical environment variables are set: PYTHONDONTWRITEBYTECODE to prevent Python from writing .pyc files and PYTHONUNBUFFERED to ensure real-time log outputs. This aids in debugging and monitoring.

4. **System Dependency Installation:** The Dockerfile installs necessary system dependencies (like gcc, ffmpeg, libsm6, and libxext6), indicative of the project's reliance on media processing and advanced computational functionalities.

5. **Optimized Python Dependency Handling:** The Dockerfile employs a layered approach to manage Python dependencies. By separating the downloading of dependencies from the installation, the build process leverages Docker's cache. This method significantly reduces build time, especially when there are no changes in the requirements.txt file.

6. **Code Copying Strategy:** It copies the entire source code into the container, as opposed to selectively adding files, simplifying the deployment process.

7. **Exposing and Running the Application:** The Dockerfile exposes port 80 and specifies the command to run the application using uvicorn, thus making the project's web application ready for production.

The frontend Dockerfile uses Nginx, a popular web server, for the containerization of the project's web-based frontend application. It copies the Nginx configuration and the web content into the container. Subsequently, it exposes port 8080, which aligns with standard web application practices and ensures the application is accessible.

The dimensions of the backend and frontend Docker images are provided in the screenshot below.

Name	Tag	Status	Created	Size
cropsegmentation-api 4c67151c89fd	latest	In use	14 hours ago	2.36 GB
nginx 9dbd2fdc9b20	latest	In use	14 hours ago	194.15 MB

3. Docker Compose Implementation

The compose.yaml file for our project defines two services: the API and the frontend application. This setup demonstrates a microservices architecture, ensuring scalability and maintainability. The service configuration within the compose.yaml is concise yet comprehensive, detailing the build context and port mappings. This clarity facilitates easy updates and maintenance. The frontend Dockerfile maps the container ports to the host machine's ports (80 and 8080) ensuring accessibility of the services.

The Docker and Docker Compose implementation in this project follows established best practices:

- **.dockerignore:** The use of a .dockerignore file is a best practice that enhances build efficiency by excluding unnecessary files from the build context.
- **Layer Caching:** The Dockerfiles demonstrate best practices in layer caching, a method crucial for optimizing build times and resource usage.
- **Security and Slim Images:** Using slim images and minimizing the number of layers reduces the security risk and the overall size of the images.

4. Building Workflows with GitHub Actions

GitHub Actions is a tool provided by GitHub that allows developers to automate, customize, and execute their software development workflows right in their repositories. Github Actions are custom automated sequences organized into workflows, where each workflow consists of one or more jobs that can run in sequence or in parallel. Within a job, a series of steps execute commands, scripts, or actions. GitHub Actions are event-driven, meaning they can be triggered by specified events such as a push or pull request. They are defined in YAML

files that are saved in the .github/workflows folder of the project and can be used for a variety of tasks such as testing code, deploying software, and much more. Another important feature is represented by Artifacts, which are files that are produced by a step in a GitHub Actions workflow. They can be used to share data between jobs in a workflow and can be downloaded after a workflow run finishes.

In our project, we have utilized GitHub Actions to automate several aspects of our software development process. We have implemented workflows for testing our model, testing our API, and checking the quality of our code with Pylint, Flake8, and Pynblint. These workflows are triggered by specific changes in our repository.

The following workflows were implemented to automate and streamline our development process:

1. **Testing Model Workflow:** The testing_model.yaml workflow is triggered when changes are made to the src folder, excluding the src/api folder, or to any of the testing folders. This workflow sets up Python, installs the necessary dependencies, sets up the data, and then runs behavioral, model training, and utility tests using Pytest.

2. **Testing API Workflow:** The testing_api.yaml workflow is triggered when changes are made to the src/api folder or the tests/api_testing folder. This workflow also sets up Python, installs the necessary dependencies, sets up the data, and then runs API tests using Pytest. The integration of DVC data with GitHub Actions was accomplished by setting up DVC in the workflow environment and establishing a Google Drive Remote by using a service account for authentication, using the GDRIVE_CREDENTIALS_DATA secret. The workflow then retrieves the necessary datasets with dvc pull -r myremote and extracts the zipped datasets into the specified directory for further use in testing.

3. **Flake8 Code Quality Check Workflow:** The flake8_code_quality_check.yaml workflow is triggered when changes are made to the src folder. This workflow sets up Python, installs Flake8, and then checks the quality of the code in the src folder using

Flake8. The results of the Flake8 check are saved in a JSON file, flake8_report.json, which is then uploaded as an artifact.

4. **Pylint Code Quality Check Workflow:** The pylint_code_quality_check.yaml workflow is triggered when changes are made to the src folder. This workflow sets up Python, installs Pylint, and then checks the quality of the code in the src folder using Pylint. The results of the Pylint check are saved in a JSON file, pylint_report.json, which is then uploaded as an artifact.

5. **Pynblint Notebook Quality Check Workflow:** The pynblint_code_quality_check.yaml workflow is triggered when changes are made to the project notebook. This workflow sets up Python, installs Pynblint, and then checks the quality of the project notebook using Pynblint. The results of the Pynblint check are saved in a JSON file, pynblint_report.json, which is then uploaded as an artifact.

In the Flake8, Pylint, and Pynblint workflows, code quality reports are generated and saved as artifacts - outputs of each GitHub action check. These artifacts can be downloaded after the execution of the GitHub action, providing a detailed analysis of the code quality and notebook standards. This feature is particularly beneficial for continuous integration, as it allows us to review the code quality check results after each workflow run and promptly address any issues highlighted in these reports.

5. Carbon Emissions Tracking with CodeCarbon

In recent years, the environmental impact of computing, particularly in data-intensive fields like machine learning, has garnered increasing attention. Carbon emission tracking is a critical aspect of sustainable software development, allowing developers to quantify and minimize the environmental footprint of their computing processes. CodeCarbon, a library

designed for this purpose, offers a straightforward way to measure and analyze the carbon emissions associated with computational tasks.

In our project, CodeCarbon was integrated for tracking carbon emissions during the training phase, a period known for high computational demand and, consequently, potentially significant carbon emissions. The carbon emission tracking was set up to encompass each epoch of the model's training phase. The following points highlight the key aspects of this implementation:

1. **Tracking Setup:** CodeCarbon was configured to track emissions from the start to the end of each training epoch. This granularity allowed for a detailed analysis of emissions over the course of the model's development.

2. **Data Collection:** The emissions data collected during each epoch were saved in a CSV file. This method provided a straightforward way to aggregate and analyze the total emissions resulting from the training of the model.

3. **Output Information:** The output data from CodeCarbon were categorized into three types, each contributing to the overall calculation of CO2 emissions:

   ○ **Consumption Information:** Data on the energy consumed during the computational process.

	duration	emissions	cpu_power	gpu_power	ram_power	cpu_energy	gpu_energy	ram_energy
44	216.815621	0.000206	0.1177	0.0612	0.279013	0.000115	0.000446	0.000044
24	215.101522	0.000214	0.0588	0.0191	0.255592	0.000056	0.000536	0.000037
68	215.810879	0.000226	0.1174	0.0631	0.259741	0.000092	0.000535	0.000036

   ○ **Geographical Information:** Details about the geographic location of the computing resources, which is relevant due to variations in energy sources and their carbon footprints across different regions.

|	country_name	country_iso_code	region	longitude	latitude
37	Italy	ITA	apulia	16.8547	41.1122
36	Italy	ITA	apulia	16.8547	41.1122
0	Italy	ITA	apulia	16.8547	41.1122

- ○ **Hardware Information:** Specifications of the hardware used, such as CPU, GPU, and RAM, as these components have different energy efficiencies and impact the overall carbon emissions.

|	os	cpu_count	cpu_model	gpu_count	gpu_model	ram_total_size
10	macOS-13.6.2-arm64-arm-64bit	12	Apple M2 Max	1	Apple M2 Max	64.0
63	macOS-13.6.2-arm64-arm-64bit	12	Apple M2 Max	1	Apple M2 Max	64.0
21	macOS-13.6.2-arm64-arm-64bit	12	Apple M2 Max	1	Apple M2 Max	64.0

4. **Optimizations for Realistic Results:** To ensure the accuracy and relevance of the results, several optimizations were implemented:
   - ○ The library was set to track the specific process involved in model training, rather than the entire usage of the machine's hardware.
   - ○ Unique identifiers were used for different training runs, especially those with varying hyper-parameter configurations. This approach was critical during the experimental phase for comparing different model iterations.
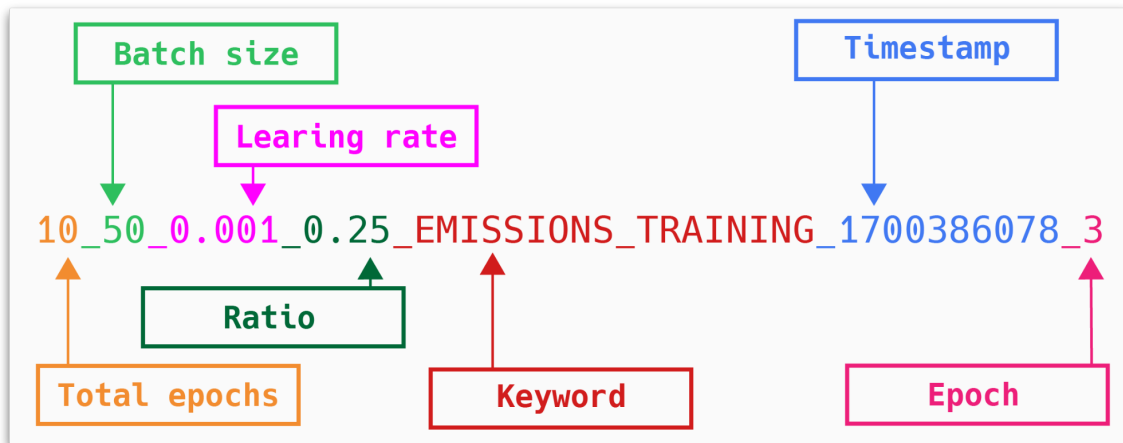

5. **Trade-Off Analysis:** Further insights on the topic have been carried out, developed, and summarized in a Python notebook, where the experimental setting was used to search for the best hyper-parameters for the model based on grid-search, and implemented in the initial phase of the project. A key objective was to evaluate the trade-off between model performance (in terms of accuracy and Intersection Over Union - IOU) and carbon emissions. We found that models with lower performance

metrics tended to consume more CO_2, indicating a correlation between model efficiency and environmental impact.
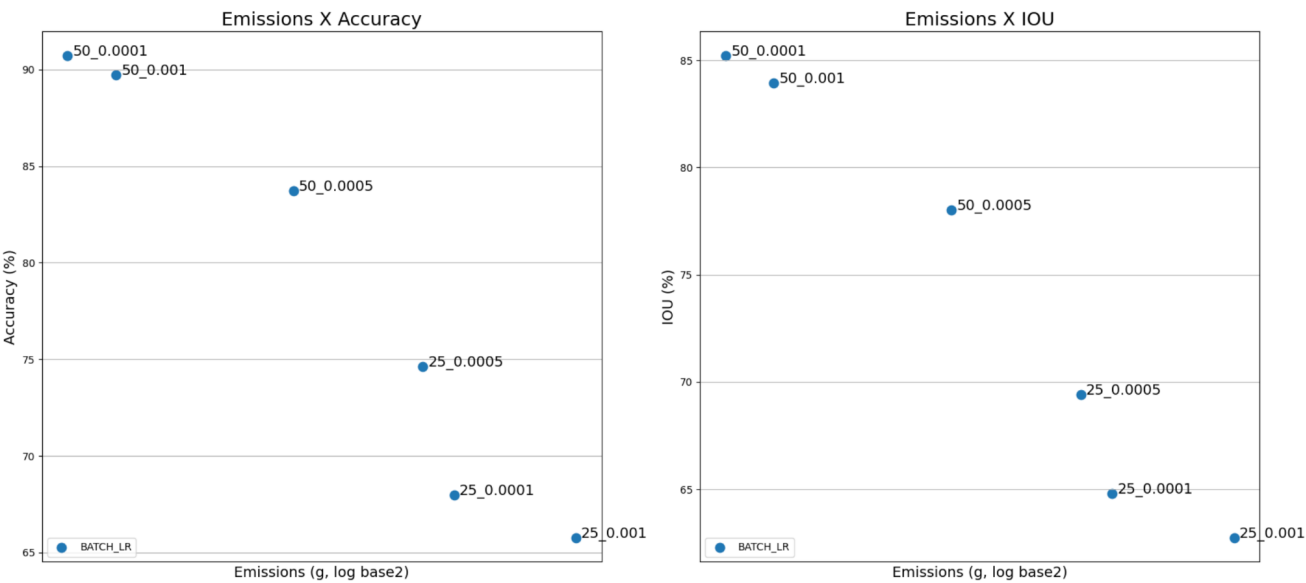
6. **Energy Consumption vs. Computing Power Analysis:** An additional aspect examined was the relationship between electrical energy consumption and the computing power of different machine components. It was observed that while the GPU was less energy-consuming, it provided significantly higher computing power, underscoring its importance in deep learning model training. The figure below showcases the average values obtained during the epochs of the training phase.
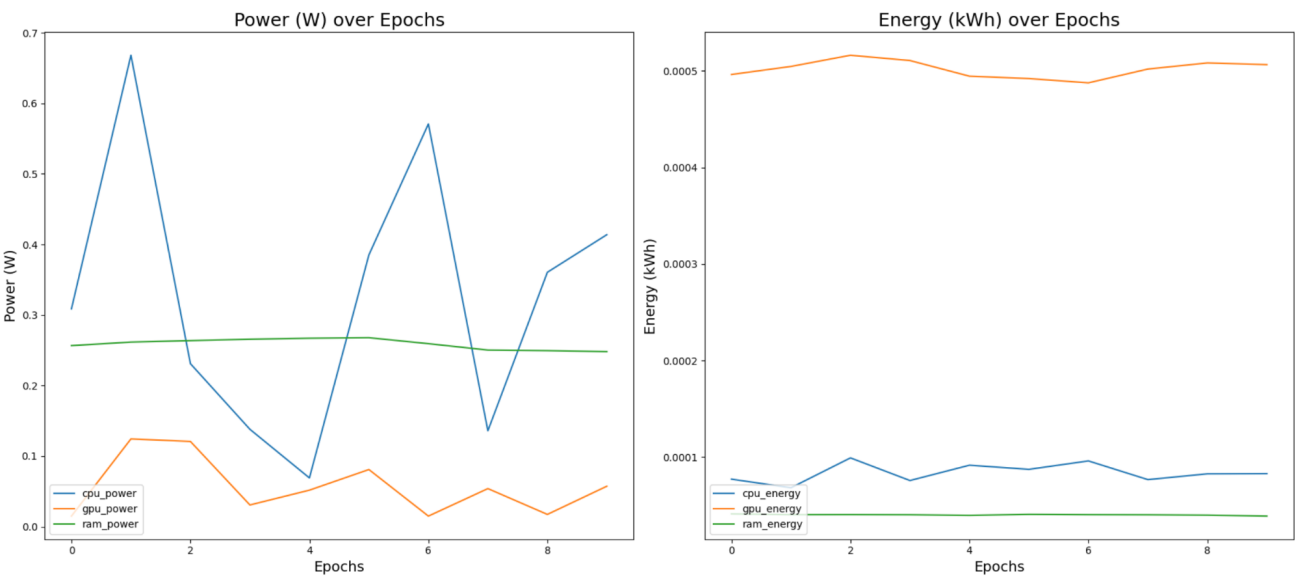
7. **Energy Efficiency Classification:** The best-performing model was further analyzed using the Green AI Dashboard tool to obtain an energy efficiency classification. Our model achieved a Class B rating, reflecting a balance between computational efficiency and environmental responsibility.
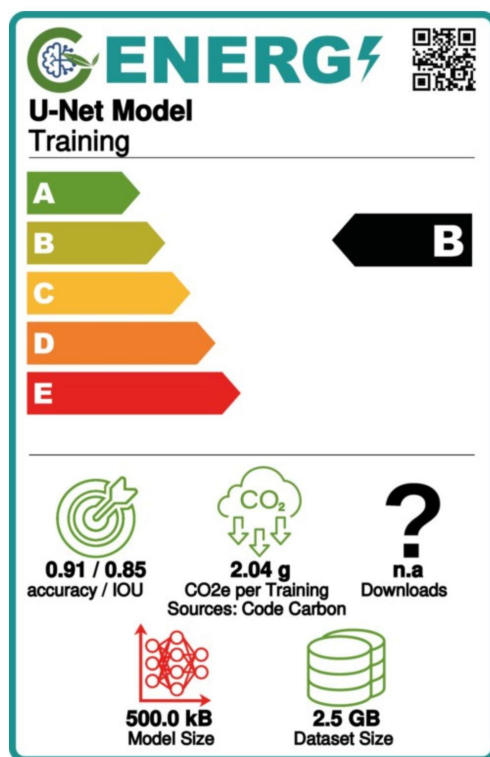
8. **Model Card Update:** The project's model card was updated to include carbon emission metadata, a section describing the carbon footprint, and the energy efficiency label obtained from the Green AI Dashboard.