MLOps
Lab1 - Lab2 - Lab3

Iker Jauregui Elso

Contents

1 Introduction

In this report, we will see a brief summary of the project developed during *Lab1*, *Lab2* and *Lab3* assignments of *MLOps* subject, where the main idea was to develop both a CLI and an API for image classification.

If we look back at *Lab1*, we developed unit and integration tests for the *predict* and *resize* functionalities and we created a continuous integration (CI) workflow on *GitHub* that linted (*pylint*), refactored (*black*) and tested (*pytest*) the code.

After that, *Lab2* focused on deployment, so we could serve an API for public use. Even if our classifier was still a random guesser, we developed a *Gradio* app that successfully used our API.

On the last assignment, *Lab3*, we finally trained a Deep Learning model by correctly tracking each experiment. Here we replaced the dummy random guesser by our trained model, and successfully implemented a full Machine Learning solution service. As image transformation utilities like *resize* were out of the scope of the main goal of the project, in order to make code cleaner and simpler, for this last *Lab3*, I personally decided to remove every hint of it from the project.

On the following sections, on the one hand, you will be able to find the links for every repository generated on this project and, on the other hand, you will find a brief explanation of the tests implemented and the training configurations carried out in the last *Lab3* assignment.

2 GitHub Repositories

Even if the *Labs* are related between them, each one of it has its own *GitHub* repository:

- *Lab1*: https://github.com/Iker-Jauregui/MLOps-Lab1
- *Lab2*: https://github.com/Iker-Jauregui/MLOps-Lab2
- *Lab3*: https://github.com/Iker-Jauregui/MLOps-Lab3

3 Hugging Face Spaces

On both *Lab2* and *Lab3*, a *Gradio* app was implemented and deployed on *Hugging Face Spaces*:

- *HF Space* for *Lab2*: https://huggingface.co/spaces/IkerJau/mlops-lab2-jauregui
- *HF Space* for *Lab3*: https://huggingface.co/spaces/IkerJau/mlops-lab3-jauregui

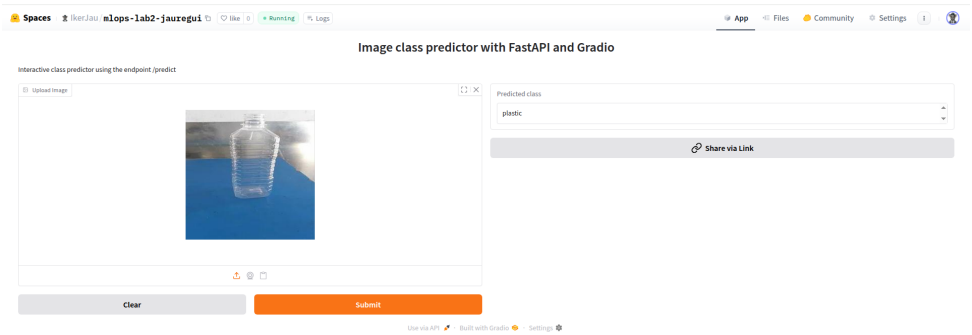

Figure 1: Gradio application for *Lab2*.

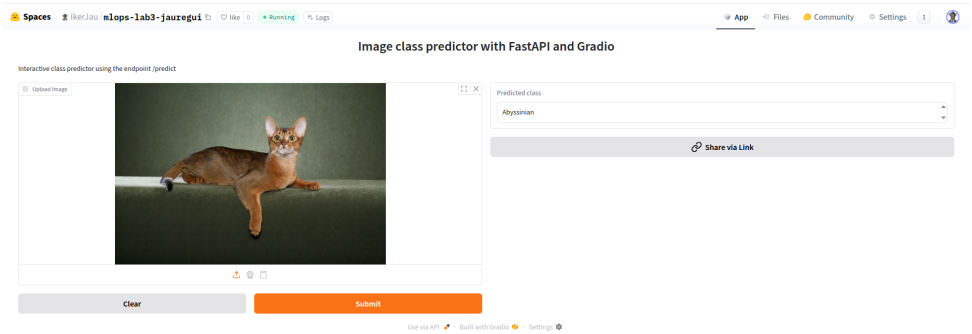

Figure 2: Gradio application for *Lab3*.

4 Tests

Within this project, there are a total of three test scripts, one for each of the services or functionalities implemented in it: the image classification library/module (*test_logic.py*), the command line interface (*test_cli.py*) and the API (*test_api.py*). Even if each of them are important, we can't deny that the most valuable functionality of the project is the API, as it is the actual public service we are exposing. So, for this last part of the project (*Lab3*), most of the tests were focused on assuring that the API worked correctly.

4.1 Tests for the Classification Module

The implemented tests aimed to assure the following aspects:

- **Model Loading and Initialization**: These tests were implemented to assure that all model related artifacts (the ONNX model and the JSON containing class names) were correctly read and instantiated.
- **Preprocessing Transformations**: These tests were implemented to assure that images were correctly resized and normalized before feeding to the model.
- **Image Inputs**: These tests were implemented to assure the following situations:
  - **Image File Path**: This is a normal use case. The model is fed with a valid and an existing path of an image.
  - **PIL Image**: This is a normal use case. The model is fed with a valid PIL image.
  - **Non Existing Image**: The model is fed with the path of a non existing file. The aim of this test is to assure the correct error handling under this common but problematic scenario.

4.2 Tests for the CLI

In this case, as this functionality isn't that important and valuable compared to the proper classifying module or the API, a basic testing was implemented:

- **Basic Information**: This test assures that *–help* option works correctly, showing the basic information of the command-line interface.
- **Image Inputs**: Same as before, these tests were implemented to assure the following common situations:
  - **Image File Path**: The *predict* command gets a valid image file path from the command-line.
  - **Non Existing Image**: The *predict* command gets an invalid path of a non existent image.

4.3 Tests for the API

The API is one of the most valuable parts of the project, but as it is a public service, it is also the most exposed and vulnerable one. So, extended tests were implemented in this part:

- **Valid Image File**: This is the most normal and expected use case. A valid image file is sent to the */predict* endpoint.
- **Invalid File**: This test assures that the API correctly handles the case when a non-image file is sent through it.
- **Deterministic Response**: This test assures that the model has a deterministic behavior by comparing the results of two identical images.
- **Different Image Formats**: These tests aim to assure the correct performance of the service when unexpected or uncommon images are employed:
  - **Different Image Sizes**: These tests use different image resolutions to assure that both API and model correctly handle and resize these images.
  - **Different Image Formats**: These tests assure that the system performs correctly with various image formats (PNG, JPG, TIFF), with alpha channels (RGBA) and with grayscale images.

5 Training Experiments

In order to train the best model, a total of three sequential experimental steps were conducted. In each one of them, an specific train parameter was analyzed, and based on the obtained results, the best value was fixed to be used on the next experiments. Even though, as it can be seen in Table 1, all experiments share a common parameter configuration.

Parameter	**Value for 1st Experiment**	**Value for 2nd Experiment**	**Value for 3rd Experiment**
batch_size	[32, 64, 128]	64	64
learning_rate	1e-3	1e-3	1e-3
max_epochs	5	5	15
num_workers	2	2	2
train_val_split	0.8	0.8	0.8
val_test_split	0.5	0.5	0.5
optimizer	Adam	Adam	Adam
seed	42	[42, 1234, 4321]	42
weights	IMAGENET1K_V1	IMAGENET1K_V1	IMAGENET1K_V1
frozen_backbone	True	True	[True, False]

Table 1: Hyperparameters configuration during experiments.

Here is a brief explanation of the experiments:

- **Batch Size**: First, different training batch sizes were tested (32, 64 and 128). A batch size of 64 achieved the best performance (Figure 3), so this value was fixed for the next experiments.

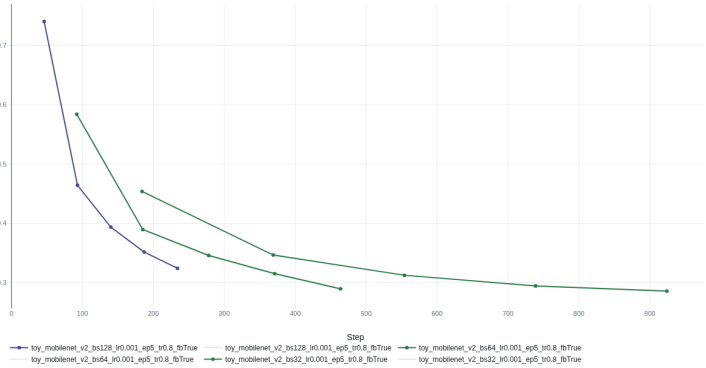

Figure 3: Validation losses for the batch size experimental study. (Taken from *MLFlow UI*)

- **Random Seed**: Second, in order to see the impact of the randomization on data splits and model's structure, different random seeds were tested (42, 1234 and 4321). 1234 and 4321 values showed almost no difference on the results, but 42 seed value achieved a slightly better performance (Figure 4), so this seed was fixed for the next experiments.

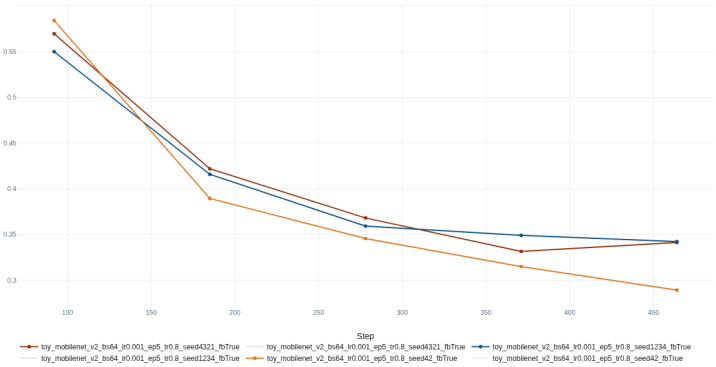

Figure 4: Validation losses for the random seed experimental study. (Taken from *MLFlow UI*)

- **Frozen vs Unfrozen Backbone**: Until now, as requested on the assignment, all but last network layers were froze during training. So, in order to test a parameter that could make a big difference, this freezing aspect was studied by training the frozen model and a completely unfrozen model. To give the unfrozen model more time to fit, training epochs (*max_epochs*) were extended to 15. The obtained results showed that the unfrozen model achieved the worst results by far (Figure 5), so letting the network overwrite its weights (at least the way it was done) probed to be a bad idea.

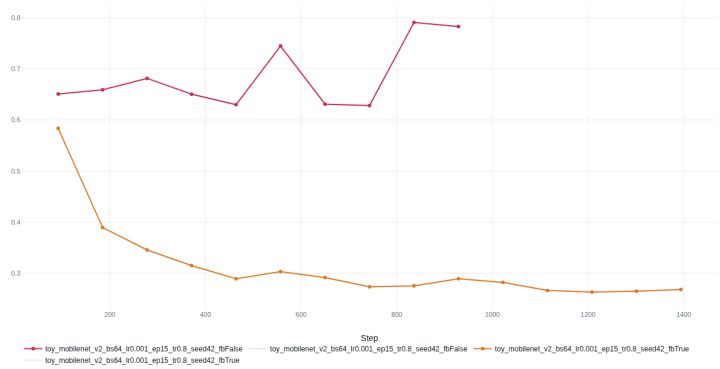

Figure 5: Validation losses for the random seed experimental study. Unfrozen model aborted training on epoch 10 due to early stopping. (Taken from *MLFlow UI*)

Regarding to the logged artifacts, for each run, training hyperparameters and different metrics (train and validation loss and accuracy) were logged. But, apart from that, some interesting artifacts were saved too:

- **Model Checkpoint and Class Labels**: Trained model's weights and used class names and indexes were saved in order to use for prediction or to resume training from that point.
- **Train and Validation Curves**: Loss and accuracies curves were plotted and saved using *matplotlib.pyplot*. This may seem to be redundant with the tracked metrics, but this plots were generated referred to epochs instead of training steps. This is a more natural way of analyzing train curves and provides a common ground for all experiments, as the use of different batch sizes modifies the number of steps and makes difficult to compare plots of different runs (see Figure 6).

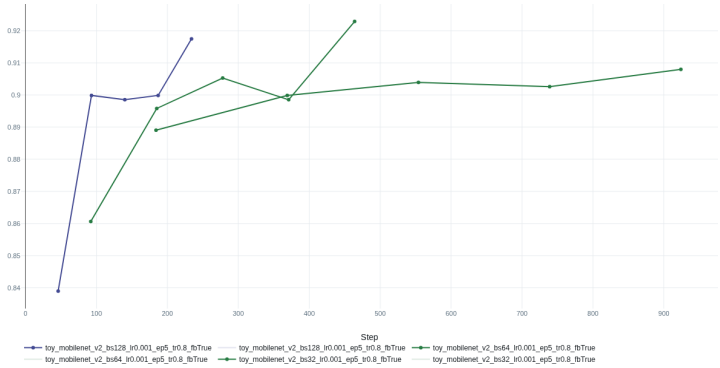

Figure 6: Validation accuracy curves for the experiments of the batch size study. Notice how each run starts and ends at different steps even if they were trained with the same number of epochs. (Taken from *MLFlow UI*)

- **Test Accuracy and Confusion Matrix**: Finally, models were evaluated on test set and both the accuracy and the confusion matrix were stored. The confusion matrix would help us to identify which classes are being misclassified and the test accuracy would be used to rank the models and select the best one (Figure 7).

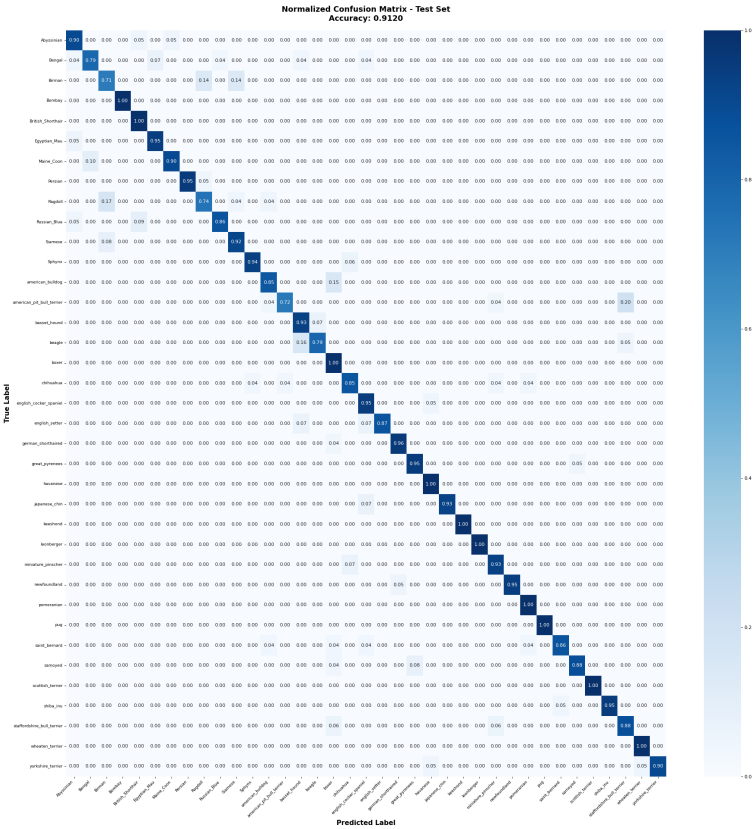

Figure 7: Confusion matrix of the best model. (Downloaded from *MLFlow UI* logged artifacts)

6 Best Model Selection

Finally, in order to serialize and deploy a final model for the whole application, the best model was selected based on the metrics obtained on the test set (Figure 8). This model belongs to the frozen trained model of the last experimental setup, and achieved a global accuracy of **91.20%** on test set.

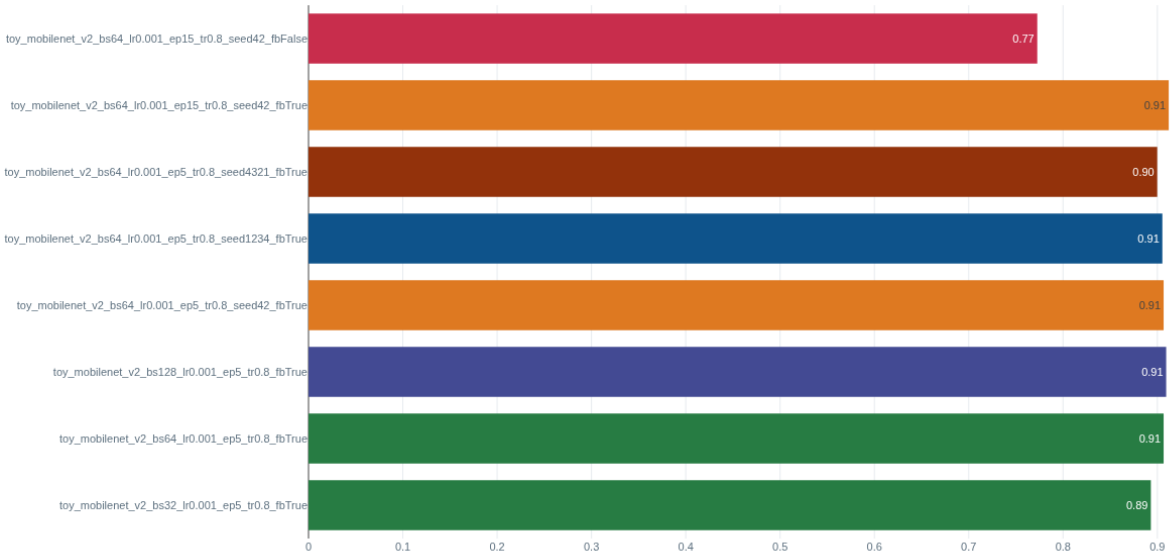

Figure 8: Accuracy on test set for all the trained models. (Taken from *MLFlow UI*)

7 Conclusions

This project successfully demonstrated a complete MLOps pipeline for image classification, integrating CI/CD workflows, API deployment and train experimentation tracking. Through rigorous testing and three sequential experimental studies, we achieved a robust production ready system with a 91.20% test accuracy using a frozen MobileNetV2 network with a batch size of 64 and a random seed of 42.

8 Future Work

Future improvements could focus on reaching better test metrics by training models with data augmentation. Due to *Render's* free plan's computational limitations, a shadow deployment pattern would be discouraged, as maintaining two models working in parallel at the same time would slow down too much the API response. Instead of that, a canary deployment pattern or a blue-green deployment pattern could be used to smoothly integrate new trained models with minimal risks.