University Master's Degree in
Machine Learning

Assignment: Experiment tracking and versioning with MLFlow

The main objective of this assignment is to learn how to track the experiments and version models in MLOps. To do it, you must create a new repository for this lab named MLOps-Lab3. In this third part of the series (Lab1-Lab2-La3) you must replace the random model with a deep learning model, so that the class label is predicted by a real classifier. Furthermore, we will also track the experiments carried out to select the model to be pushed into production. Specifically, we will use MLFlow as the experiment tracking and versioning tool. You can find information about MLFlow in the following links

- MLFlow Tracking Quickstart
- MLFlow Tracking
- MLFlow for Deep Learning
- MLFlow integration for PyTorch

Specifically, you must develop three scripts to:

1. Train deep learning models using transfer learning and track the experiments with MLflow (registering the trained models).
    - NOTE: The aim of this lab is to learn how to track and version models. Therefore, you do not need to train the models for many epochs.
2. Query the registered models to select the best one and serialize it in ONNX format (the serialized model will be deployed in production).
3. Modify the inference script so that it uses the serialized model to make predictions on the images.

The details of each phase are as follows. **Train deep learning models** using transfer learning

- You must use lightweight models such as *MobileNet_v2*.
    - It is included in *torchvision.models* (named *mobilenet_v2*, you can use the "*IMAGENET1K_V1*" for the weights)
    - You can try other small models
- You must adapt the classifier of the deep learning model to tackle the *Oxford-IIIT Pet dataset*
    - Resizing the images to 224x224
    - Create a training and validation set (for instance 80/20 of the examples to each set)
        - Ensure reproducibility (seeds, shuffle options, etc.).
- To accomplish the transfer learning task
    - Freeze the parameters of the feature extractor.
    - Modify the classifier so that its final layer contains as many neurons as there are classes.

- Use cross entropy as the loss function and the Adam optimizer
  - You can use other options if you want

You can **train several models by using different models**, different configuration for these models (play, for instance, with batch size or learning rate). You must log each experiment using *MLFlow*. To do it, you must

- Specify the experiment name (it can be common for all the runs)
- Specify the run name (it must be different for each run)
  - You can compose it by concatenating the model's name with the names and values of the configuration you are using ?
- In each run
  - You must log the model's hyperparameters using *log_params*.
    - As well as all the details to ensure reproducibility (seeds, model name, name of the dataset used, ...)
  - You must log the metrics you want to track using *log_metrics*. For instance,
    - The final accuracy achieved on the training and validation sets.
      - You can also log it by epoch to visualize a curve (you can do it using *step=epoch*)
    - The training and validations loss by epochs
      - You may also log the loss curves (PNG files) as artifacts using *log_artifact*.
  - You must log the class labels as a JSON file (you can log them as an artifact)
  - You must register the model using *log_model*
    - Use the same name for all the registered models (the version will be automatically increased)
- You can visualize all logged information using the *MLflow UI*.
  - Run in a new terminal: *mlflow ui*
  - You can also see the progress of the training process in the GUI
  - You can easily compare several executions to make decisions or reports

Select the best registered model and serialize it so that it can be pushed into production. To do it, you must develop another script where you

- Use the *MLflowClient* to query the registered model versions you want to compare.
  - Using the *search_model_versions* function where you can filter the models, for instance, by its name: f"name='{MODEL_NAME}'"
    - This function returns a list of models
- Compare the list of models to select the best one according to the accuracy in the validation set
  - For each model you can obtain its run ID (*run_id* property)
  - You can use this run ID to obtain the metrics of this run
    - *MlflowClient* instance using the *get_run* method and then, accessing the *data* property and finally the *metrics* property
      - You will have all the registered metrics (accessible with the *get* method)
  - Save the best model (you will use it later)
- Load the model associated with the best version using the *load_model* method of *mlflow.pytorch*

    - You must send the model URI: *f"runs:/{best_version.run_id}/model"*
    - Move the model to the CPU using *model.to('cpu')*.
        - Because Render does not support GPU
    - Set the model to evaluation mode using *model.eval()*.
- Serialize the model using ONNX as format. Documentation about ONNX from PyTorch.
    - *torch.onnx.export* (use 18 as *opset_version*)
- Save the class labels (JSON file). This is necessary because when loading a serialized model, we do not know the class labels, just the class position. This way, we could output a class label instead of a class number (position).
    - Load the class labels of the best version (the one you saved before) using the *download_artifacts* method of the *MLFlowClient*
        - You must send as input arguments of this function both the *run_id* of the best version and the file name of the JSON file you saved when training the model
    - Open the local path (to the registered model) obtained in the previous step to load the JSON file content and save the class labels in a JSON file

Finally, you must **replace the random prediction** we used in the previous labs **by the prediction using the serialized model**. To do it, you will use the *onnxruntime*, you can find information in these links

- Documentation about onnxruntime for Python
- Blog with information about how to export and import a model into ONNX format
- Article from Azure about the usage of ONNX models

To develop the inference, you can create a classifier wrapper including the code to load and use the ONNX model or you can use different functions without a class. Anyway, you must

- Start the ONNX Runtime session
    - Instantiate the *InferenceSession* class using as input arguments
        - The model path (to the .onnx file)
        - *providers=["CPUExecutionProvider"]*
        - Assing the *sess_options* to an instance of *SessionOptions()*
            - Setting the *intra_op_num_threads* to 4
    - Obtain the session name: *session.get_inputs()[0].name*
- Get the class labels (read the JSON file)
- Define a function/method to preprocess the data, which receives the input image and it preprocess it to accommodate it to the format of the images used to train the model (RGB, size, normalization, expansion to include the batch dimension, etc..)
- Define a function/method to predict the class label for the image
    - It creates the inputs, which is a dictionary composed of
        - The session name as key
        - The result of preprocessing the input image with the previous method/function as value
    - It obtains the output of the model: *self.session.run(None, inputs)*
    - Obtains the logits of the prediction: first dimension of the outputs obtained in the previous step (*outputs[0]*)

    - Obtain the class label based on the logits and the class labels loaded from the JSON file.
- If you created a class (wrapper classifier) you can instantiate it so that you can directly import it in the API

You can add tests to check whether the serialized model (.onnx file) and the class labels (.json file) exist before the containerization of the project. You should **modify the Docker file** to include the artifacts derived from the serialization of the model and the JSON file containing the class labels so that the prediction endpoint of the API works properly.

You may find some problems when using PyTorch (*torch* and *torchvision*) in your project. If you have used the *pyproject.toml* file of the previous lab and it used **Python 3.13**, to solve these problems, you can **update the Python version to 3.11** by running

```
uv python pin 3.11
```

Then, you must **remove the previous virtual environment and create it again**

```
rm -rf .venv

uv sync
```

***Pylint* sometimes sends warnings** (and even errors) when used in projects where **PyTorch** is used, because it does not find some of its members. To avoid them, we can **create a file named *.pylintrc* where you can** configure global rules for the linting process. In this case, you can add the following content to this file (or something similar)

```
[MASTER]

extension-pkg-whitelist=torch

[TYPECHECK]

# List of members which are set dynamically and missed by Pylint inference

  # system, and so shouldn't trigger E1101 when accessed.

generated-members=torch.*, numpy.*

[MESSAGES CONTROL]

# Optional: disable refactor (R) and conventions (C)

disable=R,C
```

Then, once you have this file, you can **ensure that these rules are used** by calling the linter using this instruction

```
uv run pylint --rcfile=.pylintrc --ignore-patterns=test_.*\.py <files_to_lint>
```

On the other hand, **the experiment tracking tool as well as the transfer learning process can create data locally**, we may not want to include in our GitHub repository. To avoid it, we can **modify the *.gitignore* file** and include the newly created folders such as *data*, *mlruns*, *plots* and *results*.

Deliverables

- Report including
    - Links to the GitHub repositories related to Labs 1, 2 and 3.
    - Links to the HuggingFace spaces related to Labs 2 and 3.
    - An explanation of the logic of the testing of the project
    - An explanation of the experiments conducted
        - Including the election of the logged artifacts
    - An analysis of their results where you must use the MLFlow GUI