Lumen Data Science 2023

Harmony Is All You Need

Technical documentation

Version 1.0

Tin Ferković, Dorian Smoljan
University of Zagreb
May 20, 2023

Contents

List of Code Listings

1 Introduction

The goal of this technical documentation is to describe the technical implementation of the solution developed by our team as a part of the Lumen Data Science 2023 competition, on the subject of instrument and human voice detection in audio files.

2 Project structure and organization

This section describes the structure of our project, including the technologies and tools used, directory structure, and experiments setup and tracking.

2.1 Languages and libraries

We used a number of software tools to develop our solution. As a programming language we, of course, used Python, alongside Conda as a package manager. To load, analyze and transform audio files we used Librosa, a Python library for music and audio analysis which proved exceptionally powerful and simple to use. For data analysis, Pandas came in handy, alongside Matplotlib and its wrapper, Seaborn.

As for the implementation of our deep learning models, which made up the majority of the models we experimented with, we first tried using plain PyTorch, but soon decided to switch to PyTorch Lightning, a high-level wrapper around PyTorch. It allowed us to better organize our code, reduce boilerplate and implement new models faster. It was the first time either of us worked with Lightning, and we were pleasantly surprised by the ease of setup and the improvements it brought, and we plan to keep using it in our future experiments.

To develop an application to showcase our solution, we used FastAPI[1] for backend, and Streamlit[2] for frontend. We also used Docker[3] to containerize our application and enable it to be run virtually anywhere by using a single command.

2.2 Code structure and organization

One very important, but sometimes overlooked key to good projects is their structure. If badly structured, no matter how good the project's idea or implementation is, it is more likely to fail than to succeed. With that in mind, we paid special attention to the structure and organization of our project.

In order to achieve this, several things were done. Firstly, we followed PyTorch Lightning's guidelines and extracted every model architecture we developed into its own Lightning Module, while the logic common to all models (train and evaluation loop, optimizer configuration, and logging) was extracted to a single module. This module would then be able to take any of our model

[1] https://fastapi.tiangolo.com/
[2] https://streamlit.io/
[3] https://www.docker.com/

architectures and use them for training/evaluation, increasing the modality and decoupling our code. It also meant that, if we wanted to experiment with a new model architecture, all we had to do was to define it in its own module and inject it into the base module, which would take care of the training/evaluation.

Secondly, as we run a lot of different experiments using a plethora of different models, we needed to constantly manage a lot of configuration, as each model or experiment required some different parameters in order to achieve the best results. In order to be able to do this in an organized way, we used Hydra[4], a Python library for configuration handling. This enabled us to split every part of our configuration into a separate .yaml file, which could then be easily altered or exchanged for a different configuration file to achieve different behavior. For example, we could define one configuration file for training a ResNet model, complete with definitions for its specific learning rate and optimizers, and another for the audio transformer model, with different hyperparameters. Then, once we have these two different configs, experimenting with different models is just a matter of referencing a different model config file in the master configuration file. An example of a Hyrdra .yaml configuration file can be seen in Listing 1. Furthermore, Hydra also supports object instantiation, which meant that everything we need to define an experiment could be setup and instantiated directly for Hydra, eliminating the need for changing configurations and parameters in 10 different files each time a new experiment is run.

Finally, in order to logically organize our code and configuration into proper directories and subdirectories, we used a slightly modified version of the excellent Lightning-Hydra-Template[5], a template for organizing deep learning projects using PyTorch Lightning and Hydra. It provided an invaluable resource for code and experiment organization, helping us efficiently organize all our components into a whole.

A Github project repository was used to keep track of all changes made during development.

2.3 Data organization

IRMAS dataset was provided to us in a train and evaluation split; we futher split we evaluation dataset into a validation and test datasets with the ratio

[4] https://hydra.cc/docs/intro/
[5] https://github.com/ashleve/lightning-hydra-template

```yaml
_target_: src.models.audio_model_lightning.AudioLitModule

net:
  _target_: src.models.components.cnn_spectogram_net.CNNSpectogramNet
  no_classes: ${data.general.no_classes}

optimizer:
  _target_: torch.optim.Adam
  _partial_: true
  lr: 0.0002
  weight_decay: 0.00001

scheduler:
  _target_: transformers.get_polynomial_decay_schedule_with_warmup
  _partial_: true
  lr_end: 0.000002
  power: 0.7

scheduler_warmup_percentage: 0.05
no_classes: ${data.general.no_classes}
threshold_value: 0.5
aggregation_function: S2
```

Listing 1: Example Hydra configuration .yaml file

30-70. Audioset was simply split into train-val-test using the standard 70-10-20 split. The same splits were used for every model we evaluated. Each dataset and its train/test/val was referenced by a .csv file, which contained all the information required for the dataloader to load the dataset examples into memory and pass them to the training module.

2.4 Running and monitoring experiments

All the experiments were run on a computer equipped with an NVIDIA RTX 2080TI GPU with 11 GB of VRAM. In order to monitor and visualize the progression of our experiments, we initially used Tensorboard[6], but after a bit of research, we decided to try out Weights&Biases[7], as neither of us had used it before and we wanted to give it a try. And we were pleasantly surprised - in general, we found it more informative and easier to use than Tensorboard, especially when comparing different experiments. It also enabled us to create a team space, where we could easily monitor all experiments, regardless on whose computer the experiment was run. An example of a Weights&Biases experiment overview can be seen in Figure 1.

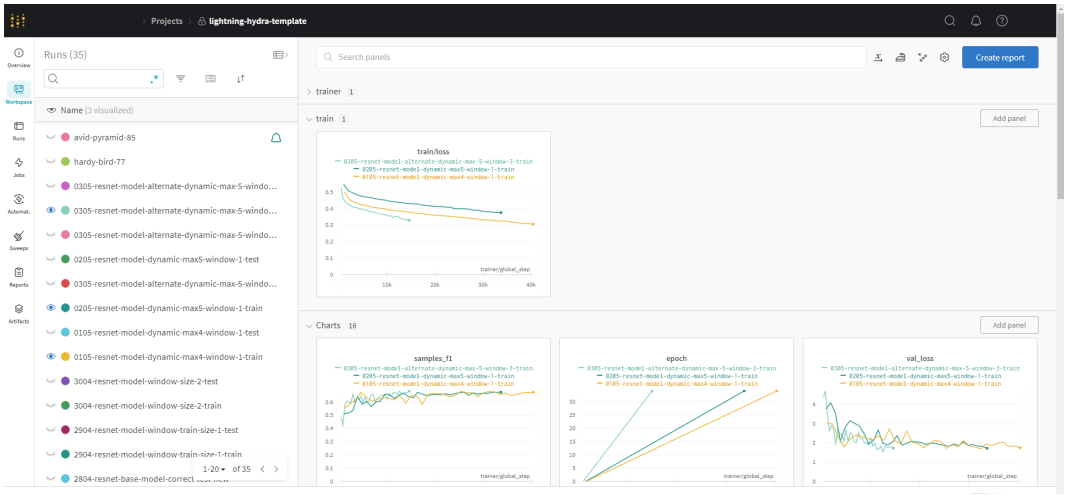

Figure 1: Weights&Biases experiments overview

[6] https://www.tensorflow.org/tensorboard
[7] https://wandb.ai/home

2.5 Exporting trained models

Once our models were trained and evaluated, we chose the best models and exported them to an archive format using Torchscript[8]. The archived model could then be easily used by the backend service in order to make predictions. We believe this is a better approach compared to using regular PyTorch checkpoints, as Torchscript exported models do not require the source code of the original model to run, while regular PyTorch checkpoint files do, thus creating a dependency between the final application and the source code of the model. Also, interestingly enough, Torchscript checkpoints are substantially smaller, for example, the checkpoint for the final model (ResNet-50), saved as a regular PyTorch .ckpt file had a size of 276MB, while the same model saved as a Torchscript checkpoint, .pt, was only 92MB in size.

2.6 Group organization and communication

For team communication, we used Discord, WhatsApp, and Google Meet. We also used Evernote to keep track of new ideas and take notes for the project documentation. As we already used Github for version tracking and syncing changes to our code, we also took advantage of its pull requests feature for code reviews.

[8] https://pytorch.org/docs/stable/jit.html

3 Application startup and usage guide

In this section, we will give a short description of the application we developed to showcase the functionality of our models, as well as a guide on how to set up and use the application.

3.1 Application startup

There are two main ways to run the backend and frontend applications: using Docker, and running the applications as Python programs directly.

3.1.1 Running the application using Docker

The easiest way of starting the application is by using docker-compose, which starts both the backend and frontend by default. Simply make sure that you have Docker installed and that your Docker daemon is running, and then follow the steps described below.

1. Position yourself in the src/app directory
2. Run `docker-compose -f docker-compose.yml up`
3. Acess the UI on `localhost:8501`, or the backend API on `localhost:8080`

It is also possible to run only the backend service, or only the frontend service. To do this, you can do the following:

Backend service

1. Position yourself in the src/app/backend directory
2. Run `"docker run –gpus=all -p 8080:8080 app_backend"`
3. Acess the backend API on `localhost:8080`

Frontend service

1. Position yourself in the src/app/frontend directory
2. Run `"docker run -p 8501:8501 app_frontend"`
3. Acess the frontend app on `localhost:8501`

When running an application using Docker, please make sure you have Docker installed and Docker daemon running. Furthermore, make sure you have NVIDIA Container Toolkit[9] installed, as it is possible that without it, Docker will not be able to recognize your NVIDIA driver.

Note: we ran into a couple of issues when starting our app using Docker and Docker compose, relating to the container not being able to recognize the NVIDIA driver installed on our system. The first time we set up Docker images and started the app using docker-compose.yaml, everything worked flawlessly. However, when we (purposely) deleted both images and reinstalled them, to make sure everything worked fine, we got the error described above, even though there were no changes made to the Docker files or Python scripts. We tweaked the Docker files a bit, and made sure Docker's default runtime is NVIDIA[10], and it started working again. We tested it on two different computers, one running Linux Ubuntu and one Ubuntu in WSL, and it worked, even after running "`docker system prune -a`" and reinstalling the images, but as we're still unsure what caused the original error, we are aware you might run into the same issue. If you do, you can try some of the troubleshooting steps we listed here, but if none work, you can start the applications without using Docker using steps from 3.2

3.1.2 Running the application without using Docker

Frontend service

1. Position yourself in the src/app/frontend directory
2. Run "`pip install -r requirements.txt`" to install all the necessary requirements for the app to run
3. Run "`streamlit run main.py`"
4. Acess the frontend app on `localhost:8501`

Backend service

1. Position yourself in the src/app/backend directory
2. Run "`pip install -r requirements.txt`" to install all the necessary requirements for the app to run

[9] `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html`

[10] `https://stackoverflow.com/questions/59008295/add-nvidia-runtime-to-docker-runtimes`

3. Run "`python main.py`"
4. Acess the backend app on `localhost:8080`

3.2 Using the application

There are two main ways to use the application - through the UI, or by sending requests to the application API. In this section, we are going to give a short introduction to how to use both ways.

3.2.1 Using the application through the UI

If you followed the instructions from section 3.1, you should now have the backend service running on `localhost:8080` and the frontned service running on `localhost:3000`. You can go ahead and open the frontend URL in your favorite browser. Once you open it, you should see the screen shown in Figure 2.

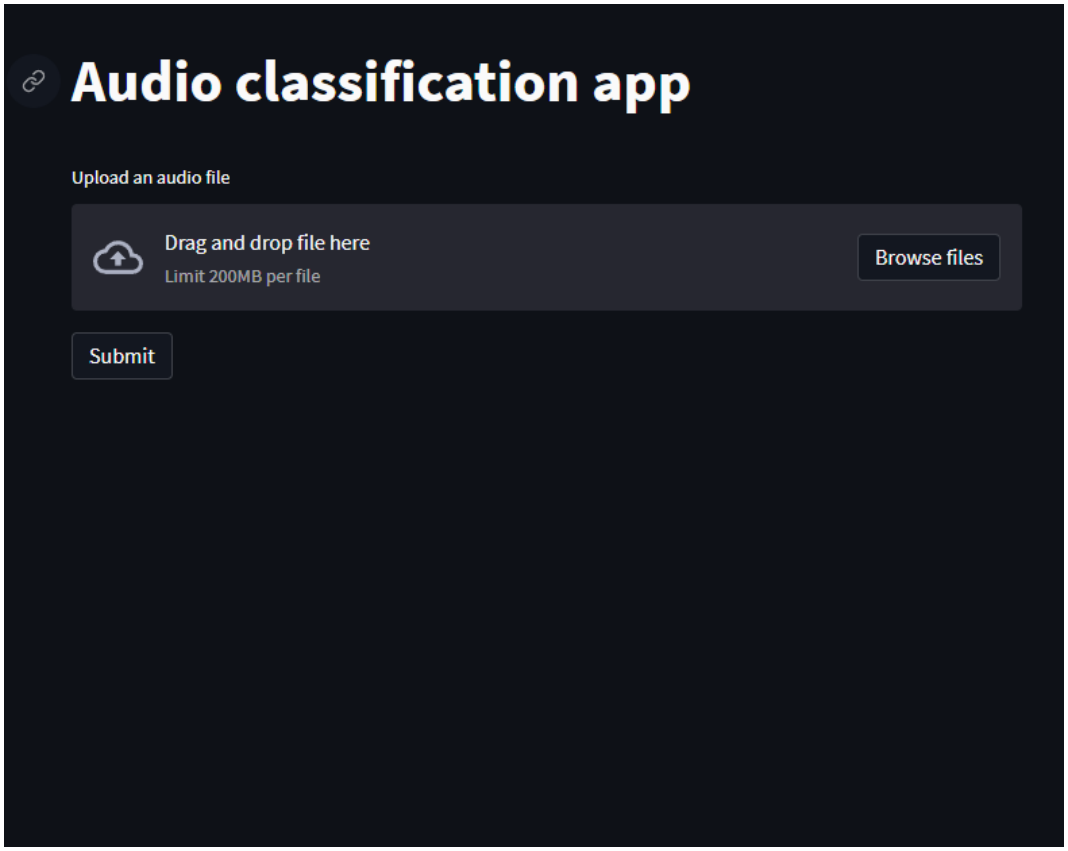

Figure 2: Initial application screen

To get a prediction, first, find an audio file you want to analyze by clicking the "Browse files" button and finding your file.

Now that you found your file, you can also play it to make sure it is the file you really want. If you are satisfied with the chosen file, click "Submit". The application takes a couple of seconds to make a prediction, depending mainly on whether you have a GPU on your system or not.

After a couple of seconds, you should see the screen shown in Figure 3, indicating which instruments were detected in your audio file!

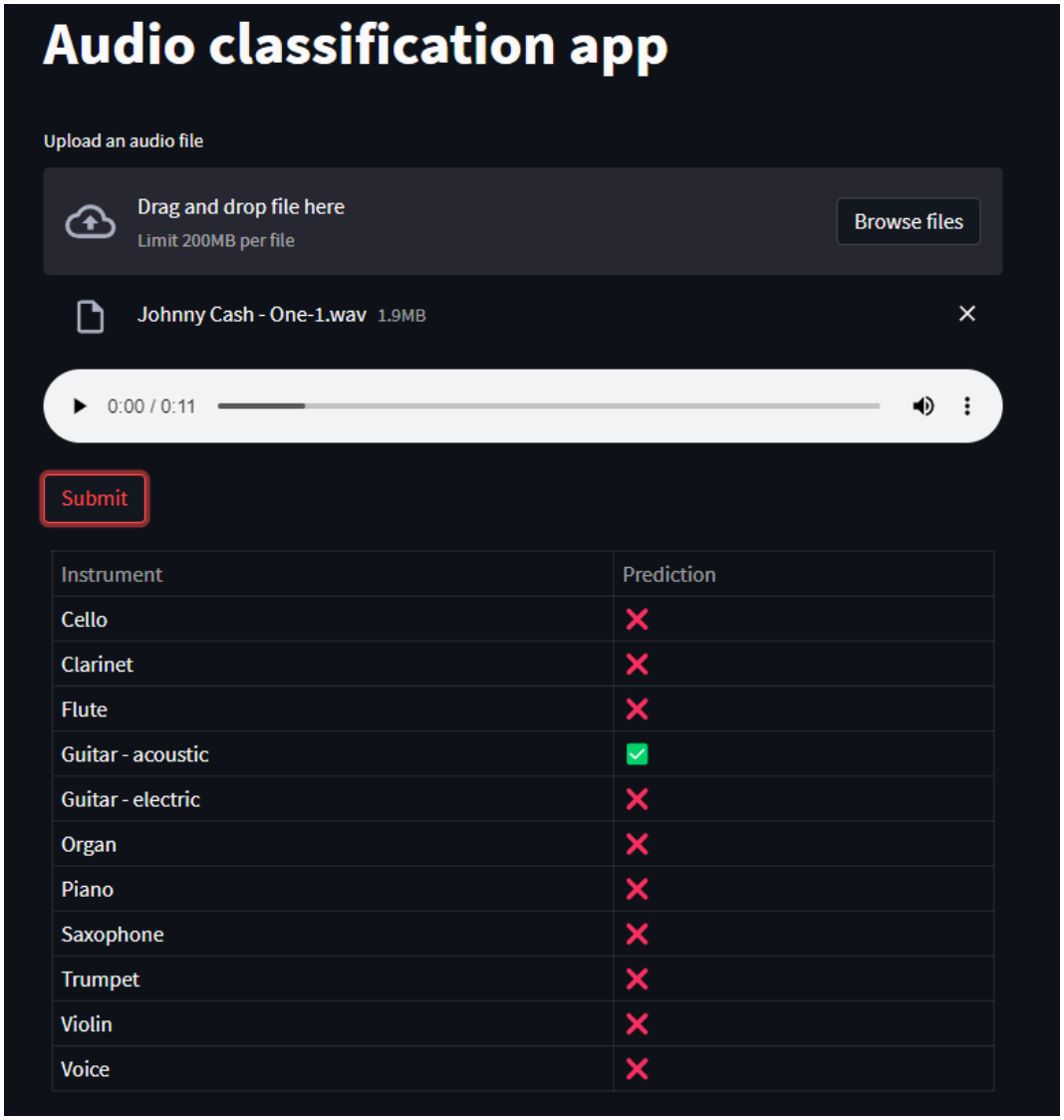

Figure 3: Screen showing the predicted instruments

3.2.2 Using the application API

Using the API is really simple. All you have to do is start the backend service using the steps described in section 3.1, which starts the service on `localhost:8080`. The prediction service is listening on the `/api/prediction` endpoint. This endpoint expects a single image file, encoded as multipart form data in the incoming HTTP request.

The easiest way to test the API is through Postman[11], an API platform for building and testing API's. To do so, run Postman and create a new request of type "POST", and set it's URL to the backend service URL, `localhost:8080/api/predict`. Then, simply fill in the fields in the "Body" section as shown in Figure 4, making sure to select the file you want to send. After that, click send, and you will soon receive the response in JSON format.

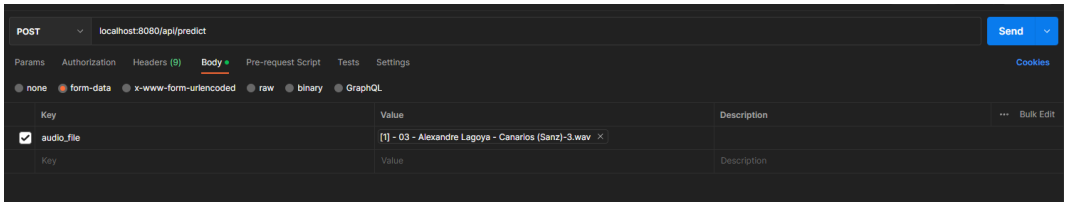

Figure 4: Postman fields setup

[11] `https://www.postman.com/`

4 Reproduciability guide

In this section, we will give instructions on how to setup directories and configuration if you want to replicate our results or continue training the models we developed.

4.1 Cloning the project

Our source code is available at
`https://github.com/dsmoljan/Lumen-Data-Science-2023`. Simply clone our repository using Git, and open it using your favorite IDE.

4.2 Code structure

The cloned repository contains multiple directories, including the documentation, meeting notes, and notebooks used in our project. The code necessary for replication of our results is located in the *Code* directory.

Inside the Code directory, you can find the *configs* directory, containing the Hydra configuration files.

In the *src/model* directory, you can find the code used for training and evaluating models, including the model architecture in *model* package, data-related classes, such as *audio_dataset.py* in *data_utils* package, and various helper methods in the *utils* package.

4.3 Configuration

The main way to define which models and with which parameters to run is by configuring .yaml files in the *configs* directory. Out of these configuration files, the most important ones are *train.yaml* and *eval.yaml* - these are the main configuration files for training and evaluation loops, so any changes you want to make are likely going to be in these files, or by changing the sub-configuration files referenced. For example, if you added a new model and want to use it for training, you would simply define its configuration .yaml file in the *configs/model/* directory, and then reference that configuration file in the field "model" in train.yaml configuration file.

Finally, you'd want to define your project root directory by modifying the appropriate variable in the .env file located in *Code* directory.

4.4 Training and evaluating models

After you have finished setting up the configuration files to suit your experiment, make sure to create a new Conda environment containing all dependencies needed to run our project. You can do this by running the following command:

conda env create -n ENVNAME --file PATH_TO_OUR_PROJECT/src/model/environment.yml

where `ENVNAME` is the name you want to give your new environment, and `PATH_TO_OUR_PROJECT` full system path to the Code folder of our project.

You can start training by positioning yourself in the *Code* directory and running the following command:
`python -m src.model.train`.

Similarly, if you want to evaluate a trained model, run the command:
`python -m src.model.eval`