



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

Client-server web application for tracking birds in videos with deep learning methods

Author:

Zsombor Fülöp

Computer Science BSc

Internal supervisor:

Dr. habil. András Lőrincz

Senior Researcher

External supervisor:

Anna Gelencsér-Horváth

Assistant research fellow

Pázmány Péter Catholic University

Budapest, 2022

Contents

Acknowledgements	3
1 Introduction	4
1.1 Reuseability	4
1.2 Fast running time	5
1.3 Easy usage	5
2 User documentation	6
2.1 System requirements	6
2.2 Installing	7
2.3 Usage	8
2.3.1 Annotation	10
2.3.2 Export frames	12
2.3.3 Create COCO JSON	13
2.3.4 Train	15
2.3.5 Inference	18
2.3.6 Track	19
2.3.7 Statistics	21
3 Developer documentation	23
3.1 Theoretical concepts	23
3.1.1 Artificial Intelligence	23
3.1.2 Machine learning	24
3.1.3 Artificial neural networks	25
3.1.4 Deep neural networks	26
3.2 Used technologies and tools	28
3.2.1 Python	28
3.2.2 Mask R-CNN and Detectron2	28

3.2.3	MiVOS and SegAnnot	29
3.2.4	Augmentation on light conditions	29
3.2.5	Decord	29
3.2.6	TensorBoard	30
3.2.7	NIPGBoard vz-labelvideo plugin	30
3.2.8	Streamlit	31
3.2.9	Program environment	31
3.3	Code components	32
3.3.1	Calling backend	33
3.3.2	Collecting labeled data	34
3.3.3	Creating training data	37
3.3.4	Training model	40
3.3.5	Model inference	42
3.3.6	Track detections and correct mistakes	47
3.3.7	Get statistics	48
3.4	Testing	48
3.4.1	Backend	48
3.4.2	Frontend	49
3.5	Further development	50
4	Conclusion	51
	Bibliography	52
	List of Figures	55
	List of Codes	57

Acknowledgements

I would like to express my gratitude towards my supervisor Dr. habil. András Lőrincz and Anna Gelencsér-Horváth. Throughout the last year spent with researching this topic and writing my thesis, I could dive into the depth of Artificial Intelligence and gain valuable knowledge. I am very thankful for the NIPG research group that supported this project financially.

My appreciation also goes out to all my teachers for not only passing on an enormous amount of knowledge but also for letting me develop essential skills and attitudes.

Finally, I would like to thank my family and my friends for their endless understanding and encouragement. Without them, I could not have completed my studies.

Chapter 1

Introduction

The mating strategy of ruffs is an interesting question in behavioural ecology. At the Max Planck Institute for Biological Intelligence ¹, the research on mating strategies of ruffs involves the annotation and analysis of hundreds of hours of video footage. This is a low-skilled task, but in the lack of automatic tools, it turns out to be an extremely time-consuming process for expert ornithologists.

To speed it up an automatic annotation and tracking approach, driven by state-of-the-art AI methods needs to be developed. However, due to the complexity of the task, the currently available tools such as *idtracker.ai* [1] or *ToxTrac* [2] and end-to-end deep networks are not able to reliably track the bird instances. The aim of our research project is to propose a technique for tracking the instances with preserving the identity labels based on automatic annotation.

In my thesis I present the efficient implementation of the training and evaluation of the object detector neural network *Mask R-CNN* [3] through *Detectron2* [4] API, as well as a framework to bring the whole pipeline together. It is essential to provide the ornithologists with a software with three main aspects besides reliability: reusability, relatively fast running time, and easy usage.

1.1 Reuseability

The behavior ecology research in which the software will be used, lasts for years, as the mating season repeats annually and new questions may arise each year. This also means, that there can be minor changes in the setting of the experiment and the

¹<https://www.bi.mpg.de/en>

software should be able to deal with this. Therefore, the collection of labelled data and the training of the object detection deep network is as relevant as the evaluation and use of the detections.

1.2 Fast running time

Evaluating deep neural networks has huge computational costs, which requires a considerable amount of time even if executed on high-performance GPUs. For this sake, I optimized the evaluation by decoding videos with GPU acceleration using *Decord* [5] and implementing post GPU tasks with multiprocessing. Of course, the runtime also depends on the performance and quantity of the GPU(s), thus, it was also an essential factor to make it scalable.

1.3 Easy usage

Complex AI concepts are implemented, and using these programs can be challenging even for programmers. To help the user, I provide a web-based, easy-to-use graphical user interface, where understanding the underlying concepts is not needed. However, having a bit of knowledge about training and evaluating deep networks (such as amount and distribution of training data, types of annotations) is essential. The user documentation chapter of this document helps with that.

Chapter 2

User documentation

In this chapter, I explain the requirements of my software, how to install it, and how to use it, as well as provide a brief explanation of the key concepts.

2.1 System requirements

Server:

- Hardware:
 - CPU: minimum 2.2 GHz, minimum 6 threads
 - GPU: Nvidia RTX TITAN or similar Nvidia GPU
 - RAM: minimum 32 Gb
 - Disk: minimum 50 Gb + plus disk for videos
 - Gigabit internet if the client will start on a different machine.
- Software:
 - Linux OS
 - installed Apptainer [6] 1.1.0 container

Client:

- Hardware: General usage PC with 500-1000 Mbps internet
- Software: Google Chrome browser

2.2 Installing

1. Download the source code of my program to the server machine and unzip it.
2. Open the following files with a text editor and search for the first occurrence of `boardPath` with `Ctrl+f`.
 - `nipg-board-v3/tensorboard/plugins/modelmanager/modelmanager_plugin.py`
 - `nipg-board-v3/tensorboard/plugins/executer/executer_plugin.py`

It will look like this: `boardPath = '/home/csgergo/nipgboard/nipg-board-v3/'`. Change the part before `nipg-board-v3` to the path to the `nipg-board-v3` folder on the server in both files.

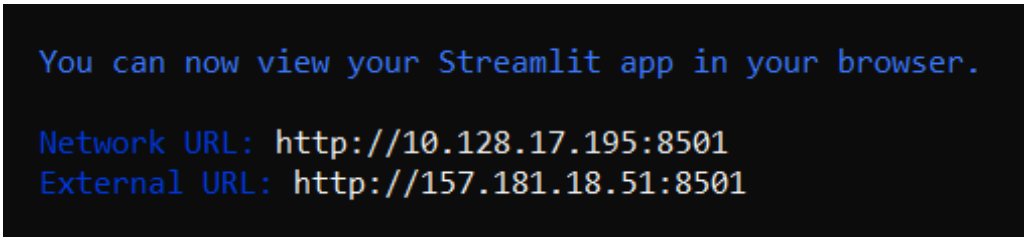
3. Open a terminal window if you have not opened one yet.
4. Run the following commands line by line:
 - `"cd <<abspath>>"` changing «`abspath`» to the absolute path to the root directory of my code.
 - `"export APPTAINER_BIND="<<abspath>>:<<abspath>>"` changing «`abspath`» to the absolute path to the root directory of my code.
 - `"apptainer instance start --nv cu113py38nvidia-codecsdk120bazel0261.sif thesis"`
 - `"apptainer shell --nv instance://thesis"`
5. Run the installer with the following commands:
 - `"chmod +x setup.sh"`
 - `"./setup.sh"`
6. Open `data/nipg-board-logdir/user/cnf.json` with a text editor and change `"/home/zsombor/thesis_zsombor_fulop/data/nipg-board-vidfolder/example"` to `"<<abspath>>/data/nipg-board-vidfolder/example"` changing «`abspath`» to the absolute path to the root directory of my code.
7. NIPGBoard (the relabelling software) is not yet stable; thus, you need to run it manually for the first time. Type this command: `"source .envs/nipg-board-v3/bin/activate && cd nipg-board-v3 && bazel run tensorboard -- --logdir=<<abspath>>/data/nipg-board-logdir --port=<<port>>"` changing «`abspath`» to the

absolute path to the root folder of my code, and «port» to a free port of your server machine. If this fails, run `"bazel clean --expunge && cd .."` and repeat the previous one. Sometimes, even running the first command twice helps. After several attempts, it should start successfully. Hit Ctrl+c to close it.

2.3 Usage

After running the following commands line by line the program will write a message show in Fig. 2.1

- `"cd <<abspath>>"` changing «abspath» to the absolute path to the root directory of my code.
- `"export APPTAINER_BIND="<<abspath>>:<<abspath>>"` changing «abspath» to the absolute path to the root directory of my code.
- `"apptainer instance start --nv cu113py38nvvidcodecsdk120bazel0261.sif thesis"`
- `"apptainer shell --nv instance://thesis"`
- `"source .envs/main/bin/activate && cd gui && streamlit run Start_page.py args --server.fileWatcherType none"`

A terminal window with a black background and blue and yellow text. The text reads: "You can now view your Streamlit app in your browser." followed by "Network URL: http://10.128.17.195:8501" and "External URL: http://157.181.18.51:8501".

```
You can now view your Streamlit app in your browser.  
Network URL: http://10.128.17.195:8501  
External URL: http://157.181.18.51:8501
```

Figure 2.1: The application started successfully

After opening the external URL on the client machine in a browser, the webpage shown in Fig. 2.2 will be seen.

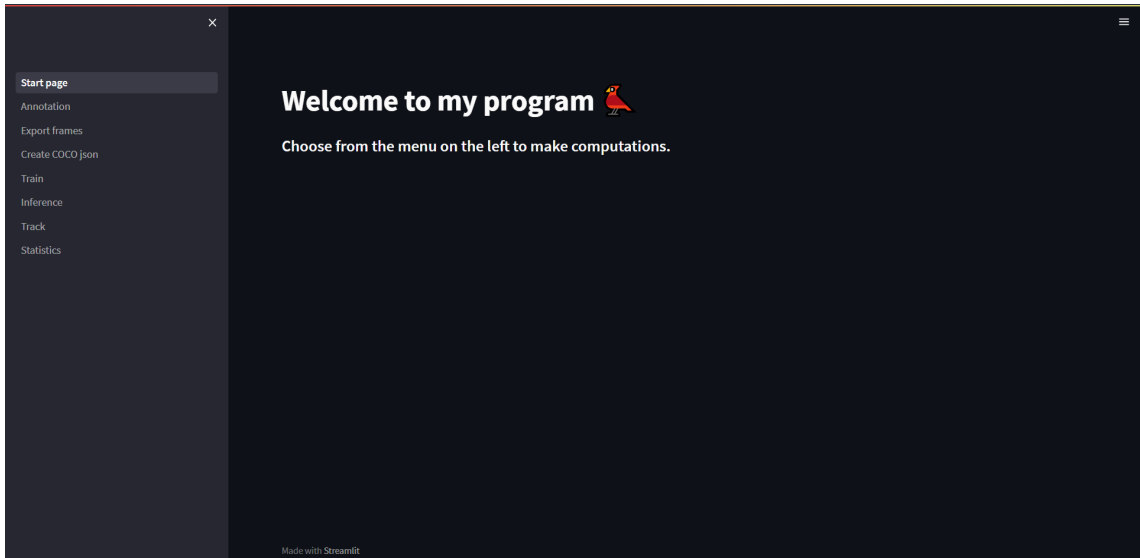


Figure 2.2: Start page

On the left side is a menu bar that lists the different modules. It can be hidden and reopened by clicking on ">" on the top.

A button on the top right opens some advanced options shown in Fig. 2.3. Only the Settings, shown in Fig. 2.4 are relevant here, where the colour theme and the layout size can be modified.

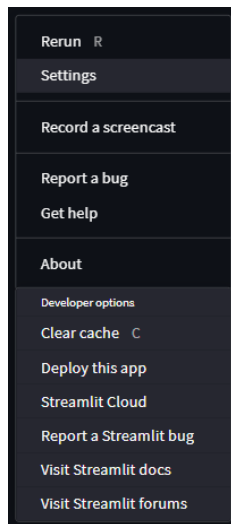


Figure 2.3: Advanced options

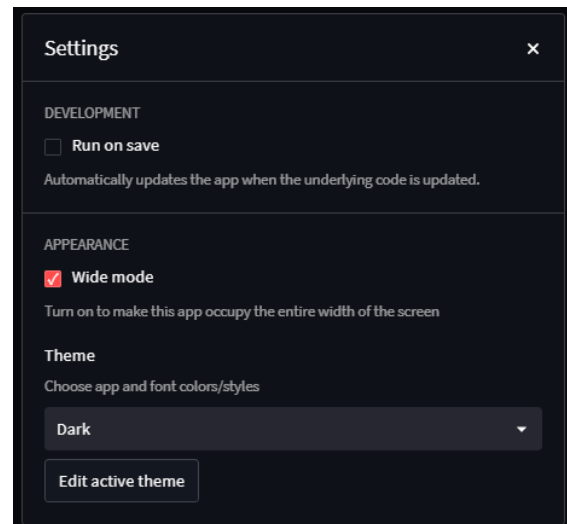


Figure 2.4: Settings

2.3.1 Annotation

The software provides an easy-to-use interface for an ornithologist to detect, track and get automatically computed statistics about videos of the ruff research. Detection is done by an AI model, which has to be shown some samples with the desired outputs to make detections later automatically. This process is called training the model. This has to be done every time there is a different video setting, camera setting, background, or light. Around 5-10 thousand images should be enough for the model to learn.

The process of creating sample outputs from the inputs for training is called annotation. Clicking on annotation in the page menu the user to a page shown in Fig. 2.5.

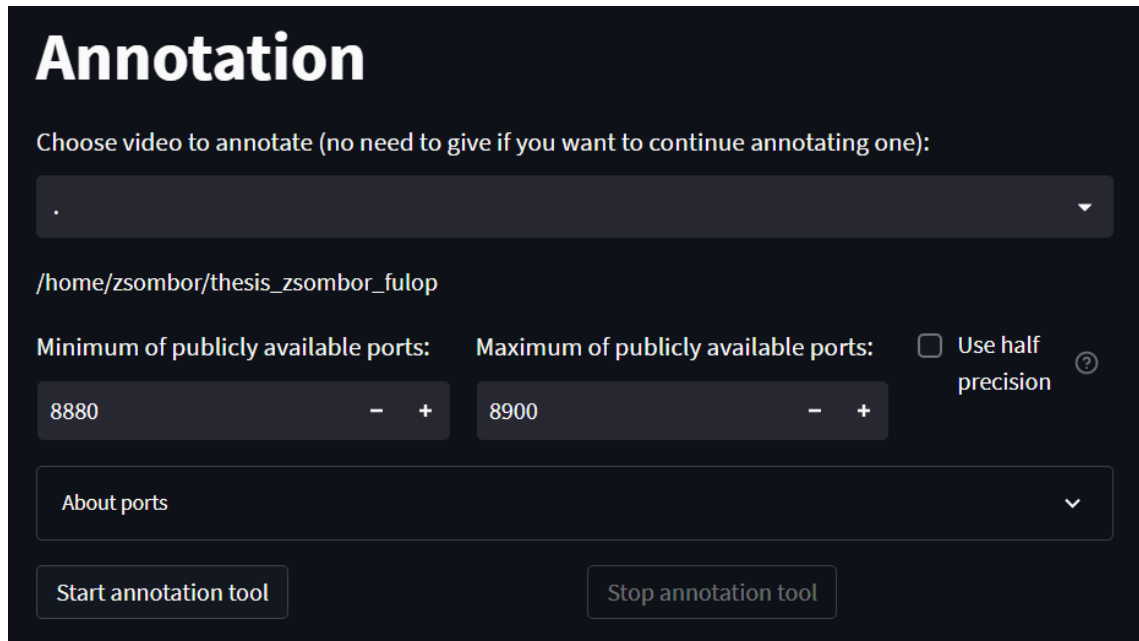
The screenshot shows a web interface titled "Annotation" in large white text on a dark background. Below the title is a text prompt: "Choose video to annotate (no need to give if you want to continue annotating one):". Underneath is a dark dropdown menu with a single visible option: ".". Below the dropdown is the file path "/home/zsombor/thesis_zsombor_fulop". Further down are two input fields for port ranges. The first is labeled "Minimum of publicly available ports:" and contains the value "8880", with minus and plus buttons on either side. The second is labeled "Maximum of publicly available ports:" and contains the value "8900", also with minus and plus buttons. To the right of these fields is a checkbox labeled "Use half precision" with a question mark icon. Below the port fields is another dropdown menu labeled "About ports" with a downward arrow. At the bottom of the interface are two buttons: "Start annotation tool" on the left and "Stop annotation tool" on the right.

Figure 2.5: Annotation page

One can choose a video from the server to label (annotate). If the video has already been chosen once and the annotation was started, there is no need to choose the video again. A port range has to be selected for the annotation tool; reading the "About ports" dropdown text helps understand what exactly is needed. When using Nvidia GeForce 20xx, 30xx and RTX TITAN GPU (the recommended one), checking the half-precision checkbox is recommended as it will provide faster labelling. After some seconds, the page shown in Fig. 2.6 will be seen.

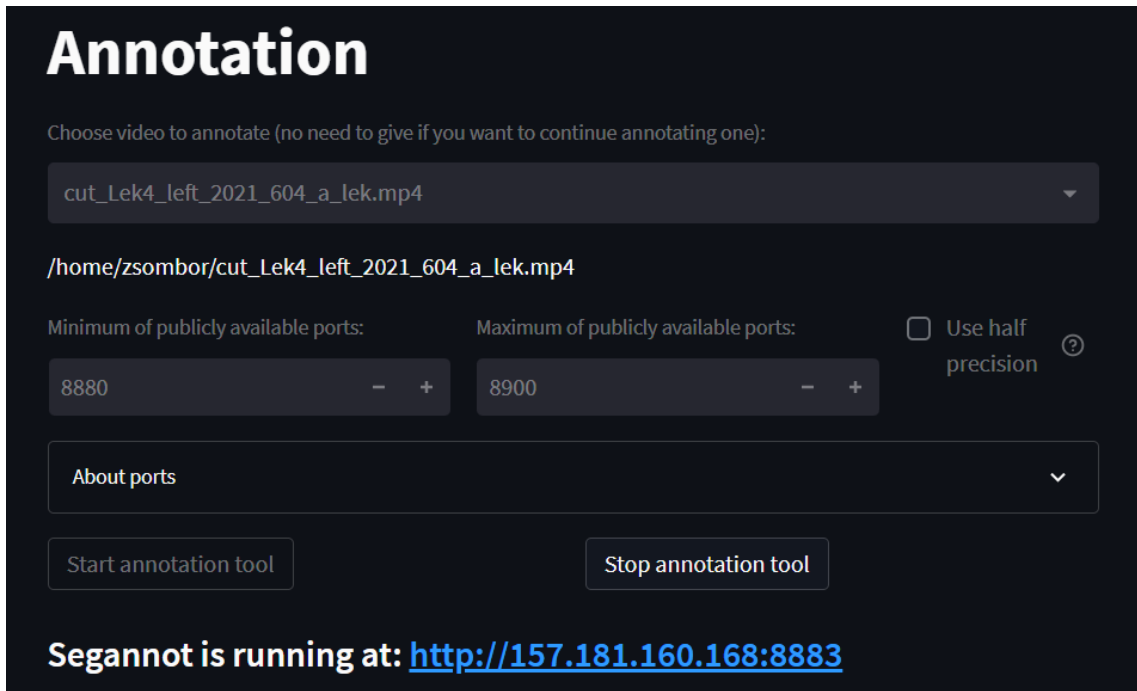


Figure 2.6: Annotation page when SegAnnot is running

The program has launched the annotation software SegAnnot which can be opened by clicking on the link written. When SegAnnot runs, the input elements are disabled except for the "Stop annotation tool" button. After clicking on the URL, the site might not load immediately. Refresh the page after a couple of seconds, and SegAnnot will load, as shown in Fig. 2.7.

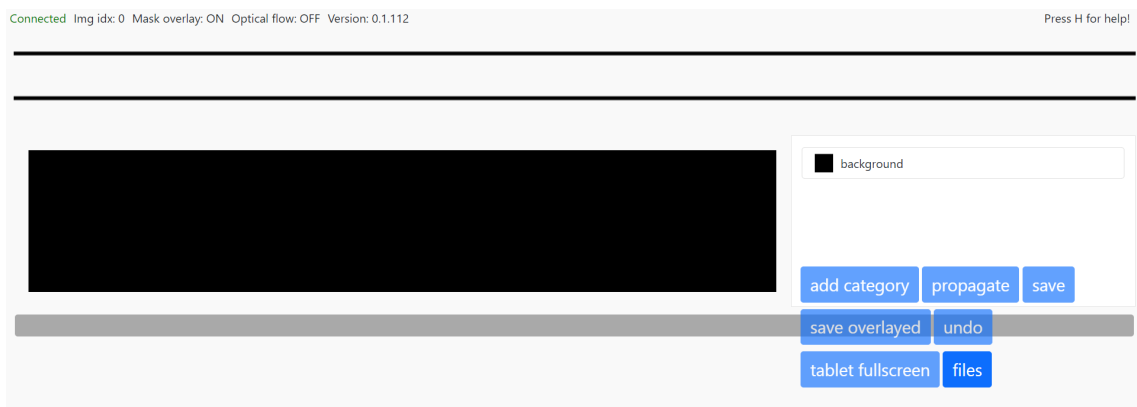


Figure 2.7: Annotation tool opened

In the top left corner, it should be written: "Connected". If not, then your client probably has lost the internet connection. Pressing "H" opens a description about SegAnnot, which should be read before the first usage. If you are ready with annotation, close SegAnnot's tab and click on the "Stop annotation tool" button. This will result in returning to the state shown in Fig. 2.5

2.3.2 Export frames

The training dataset for the model must be in a certain form called COCO JSON. The first step to creating this JSON file is exporting the input frames labelled before. Clicking on "Export frames" in the page menu takes the user to a page shown in Fig. 2.8.

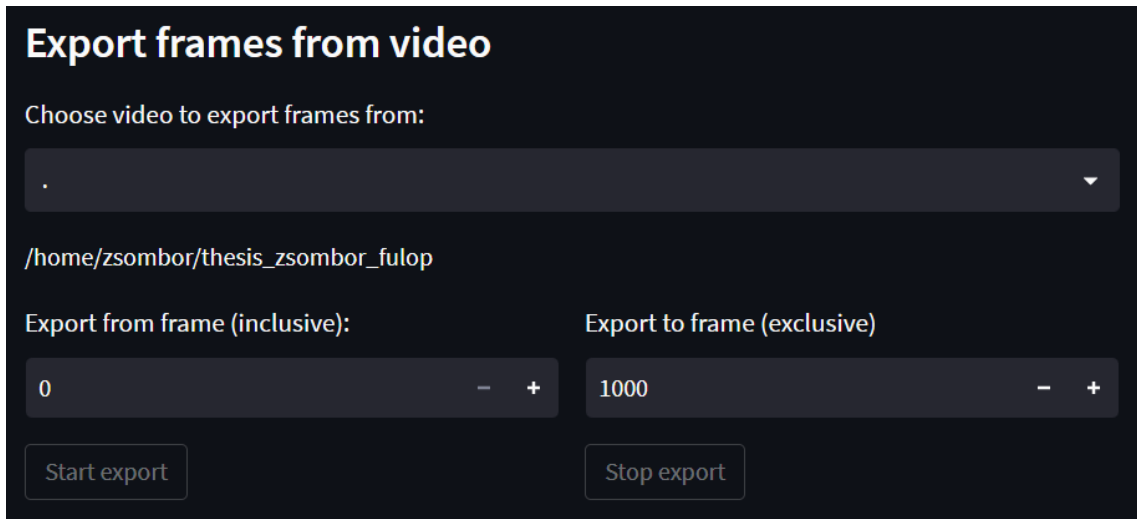


Figure 2.8: Export frames page

The "Start export" button will be disabled until a .mp4 file is not selected. After starting the exporting, a progress bar will be shown alongside the save path as shown in Fig. 2.9.

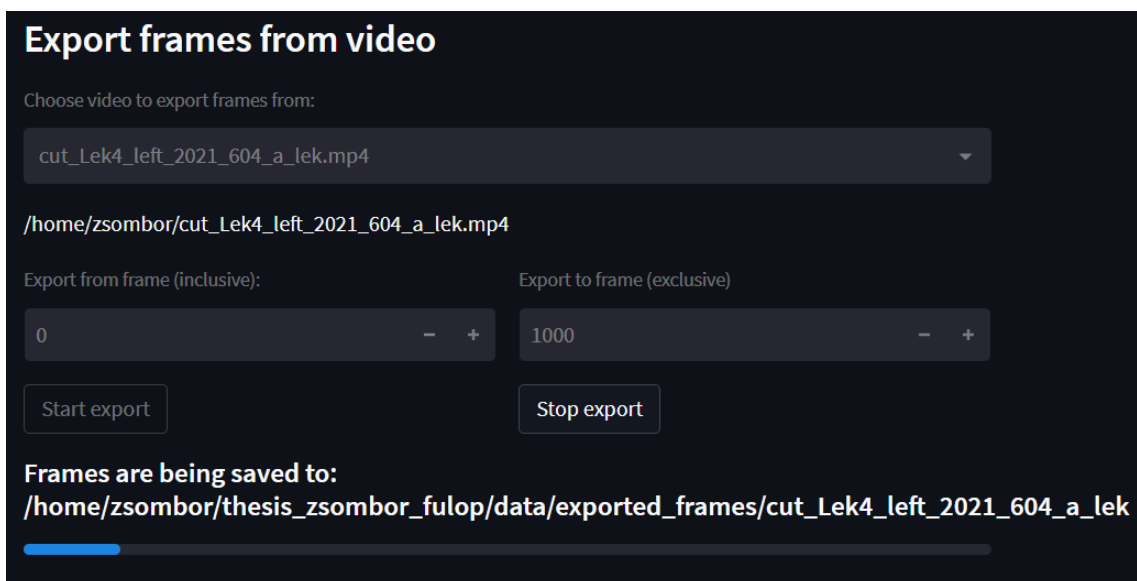


Figure 2.9: Export frames exporting running

"Stop export" can interrupt exporting, but do not forget to delete the folder with the already exported frames manually. While exporting is running, you can use the other pages on the website. If the progress bar has finished, click on "Stop export", and the message and progress bar will be deleted; you can start a new exporting. If you want to export a more extensive interval of frames from a video that you have already exported from, delete the folder with the already exported frames. Otherwise, the progress bar may not work. Always give an existing interval of frames, or the exporting will fail, which can be understood by a stuck progress bar or a message shown in Fig. 2.10. If this happens, stop exporting and delete the frames' folder. You can note the frame range when labelling with SegAnnot to ensure you give the correct frame range.

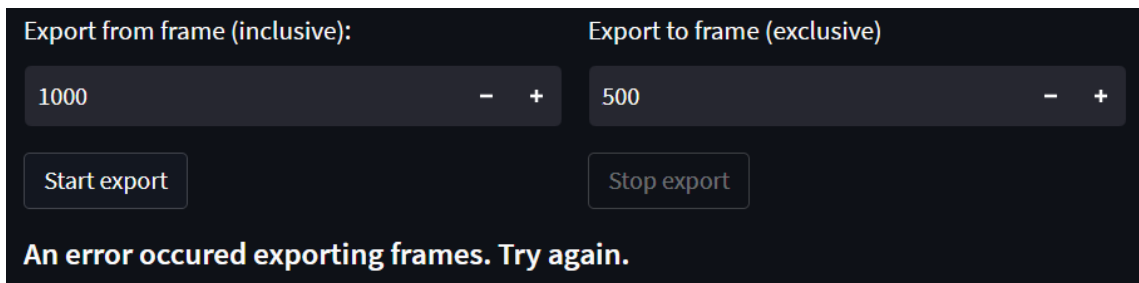


Figure 2.10: Error while exporting frames due to invalid range of frames

2.3.3 Create COCO JSON

After having the frames labelled and exported, you can create the COCO JSONs. Clicking on "Create COCO json" in the page menu takes the user to a page shown in Fig. 2.11.

Create COCO json

Add parameters of COCO json

You should add images folders and segmentations folders in the same order.

If you don't see the folder of the video whose frames you have just exported, refresh the page.

Choose the folder of the images to add to COCO json:

/home/zsombor/thesis_zsombor_fulop/data/exported_frames

Choose the folder of the masks to add to COCO json:

/home/zsombor/thesis_zsombor_fulop/segannot/working_dir

Give the names of the classes in Segannot:

Train samples ratio:

0.00 1.00

Name of the train json file:

Name of the validation json file:

Figure 2.11: Create COCO JSON page

First, you need to add the folders of the exported images and the masks that SegAnnot has saved. If you select a folder with .png files, the "Add" and "Remove" buttons will be available. When you add the first folder, a list will appear underneath the already-added folders, as shown in Fig. 2.12.

```
[
  0 : "/home/zsombor/thesis_zsombor_fulop/data/exported_frames/cut_Lek4_left_2021_604_a_lek"
  1 : "/home/zsombor/thesis_zsombor_fulop/data/exported_frames/cut_Lek4_right_2021_604_a_lek"
]
```

Figure 2.12: Multiple frame folders added

Add the class names as well. The train samples ratio is 0-1, representing a percentage. The webpage will make a training COCO JSON and a validation COCO JSON. Validation is a process that will be made during training. Some metrics will be calculated, which intend to show how well and how fast the model is learning. These metrics are not always representative if calculated on the training set; therefore, a part of the data will be separated and used as validation data. Keep it around 0.7-0.9, or 1 if you do not want validation. Lastly, the names of the two JSON files can be given without the extension.

After clicking on "Generate COCO json" the inputs will be disabled, and the save path will be written. Keep in mind that this is a long process. You can do other things with the software meanwhile. The webpage does not give an explicit notification when generating finishes; you can understand it by seeing the two .json files in the given folder.

2.3.4 Train

Clicking on "Train" in the page menu takes the user to a page shown in Fig. 2.13. On this page, you can train a model called Mask R-CNN through a tool called Detectron2.

The screenshot shows a web interface titled "Train Mask R-CNN through Detectron2 API". It features several input fields and controls for training a model. At the top, there are two dropdown menus for selecting training and validation COCO JSON files, with "train.json" and "val.json" selected. Below these, the file paths are displayed: "/home/zsombor/thesis_zsombor_fulop/data/COCO_jsons/train.json" and "/home/zsombor/thesis_zsombor_fulop/data/COCO_jsons/val.json". The interface includes sliders and input fields for "Number of classes" (set to 1), "Number of epochs" (set to 10), "Learning rate" (set to 0.00010), "Number of workers" (set to 4), and "Batch size" (set to 2). A red progress bar is visible above the learning rate slider, with a value of 0.00075. At the bottom, there are input fields for "Minimum of publicly available ports" (set to 8880) and "Maximum of publicly available ports" (set to 8900). A dropdown menu labeled "About ports" is also present. Finally, there are "Start training" and "Stop training" buttons at the bottom.

Figure 2.13: Train page

First, you need to choose the training and validation JSON files that you created before. Training can only be started after they are chosen. The number of classes given should match the number of classes in the .json files; otherwise, the training

will not work correctly. Epoch means how many times the training data will be shown to the model. If this is set too high, training will last very long, and the model will overfit. This means it will perform very well on the training data but poorly on other videos. You should set it around 5-15.

The learning rate influences how fast the model learns. If it is set lower, the training will need more epochs to reach the same performance because it will learn smaller step by step. If it is higher, the model will try to learn bigger step by step, which would naturally mean faster learning. However, if it is too high, the model will not learn at all as it will act like someone stepping a big step forward and then a big step backwards while the goal is somewhere in the middle of the two endpoints.

The number of workers influences how fast the model will get the input data. If it is set lower, the training will last longer. A higher number, however, requires more RAM and GPU memory. For an NVIDIA RTX TITAN 24 Gb, 60 Gb RAM and 1520x2688 images, 4 is a good number. Batch size means how many input images are shown to the model simultaneously. The higher it is, the faster the model will learn. However, if it is higher, it requires more RAM GPU memory. For an NVIDIA TITAN 24 Gb, 60 Gb RAM and 1520x2688 images, 2 is a good number. After starting training, a message will appear, as shown in Fig. 2.14.

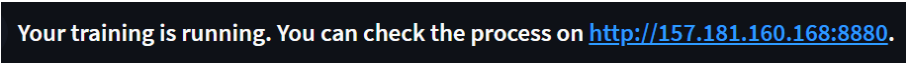


Figure 2.14: Tensorboard URL

This message means the training has successfully started. If you click on the URL, it will open TensorBoard, a graphical surface to analyze the training process. You see a page shown in Fig. 2.15.

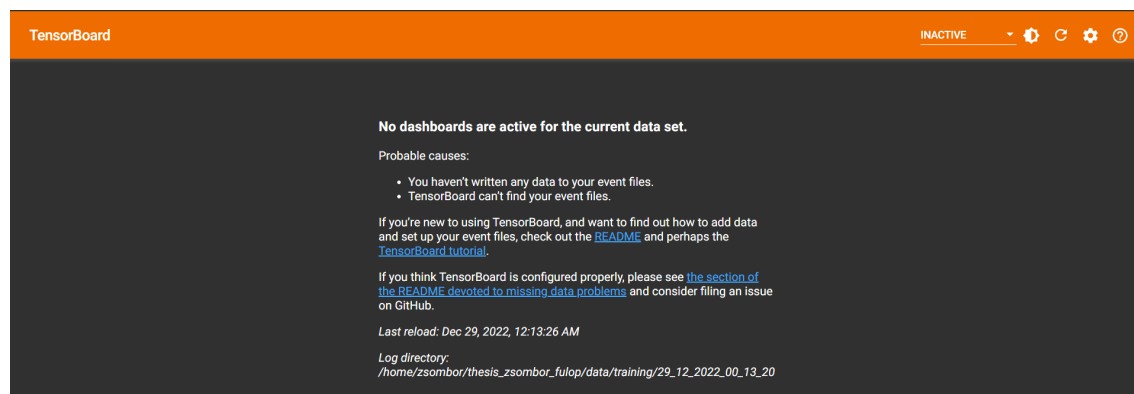


Figure 2.15: Tensorboard no data exists to display

This message means that the training has yet to display. Wait a couple of minutes and refresh the page. If the same page appears, an error has occurred during training. This can be due to too many workers or batch size, for example. Try experimenting with different numbers. When TensorBoard finds data to display, a page shown in Fig. 2.16 will be seen.

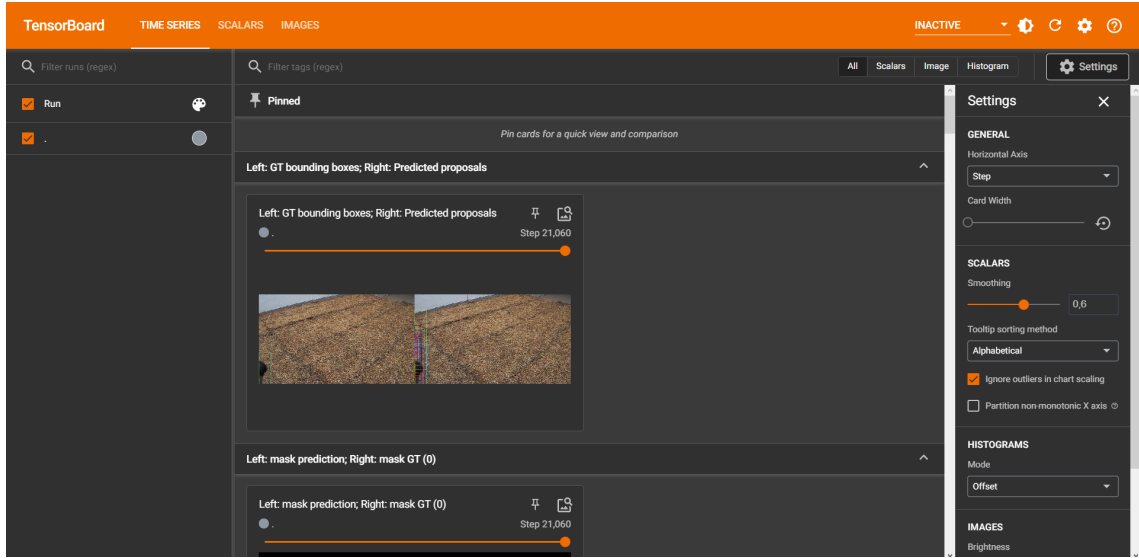


Figure 2.16: Tensorboard has data to display

We will only care about the "SCALARS" tab now, as the others require more advanced knowledge. At the bottom, when opening "total_loss" and "validation_loss" we will see two graphs. Loss is something that shows how many mistakes the model is making. During training, this should decrease; otherwise, the model is not learning. If the model overfits, we will see that training loss is very low, but the validation loss is increasing. In Fig. 2.17 and Fig. 2.18, we can see good losses.

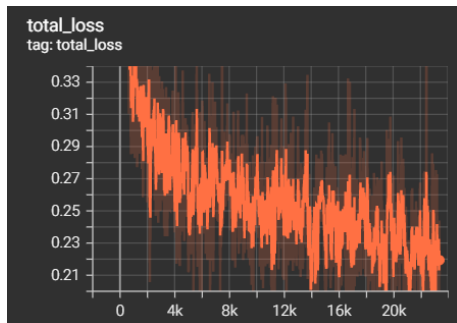


Figure 2.17: Good training loss

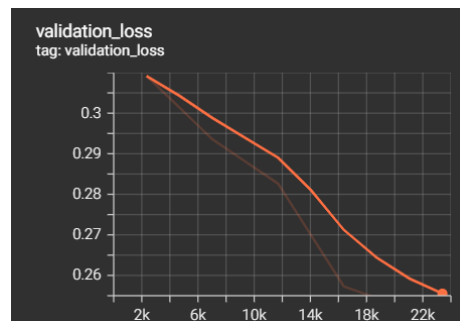


Figure 2.18: Good validation loss

The other metric which is even more representative is the average segmentation precision. This shows how precisely the model could predict the masks of the in-

stances in the validation set compared to the SegAnnot labelled masks. If we had a perfect model, this would be 100, meaning 100% precision. A reasonable average segmentation precision can be seen in Fig. 2.19.

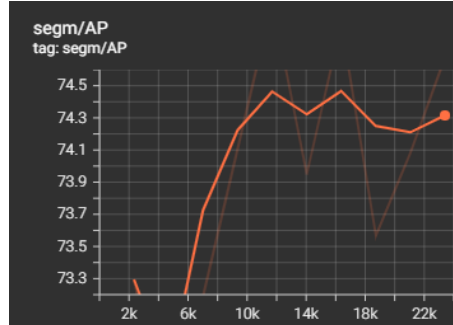


Figure 2.19: Tensorboard shows reasonable average segmentation precision

2.3.5 Inference

After having a model trained, you can inference it, which means that you can use it for detecting automatically. Clicking on "Inference" in the page menu takes the user to a page shown in Fig. 2.20.

Inference trained Mask R-CNN through Detectron2 API

Choose video to inference on:

.

▼

/home/zsombor/thesis_zsombor_fulop

Choose model folder to use:

.

▼

/home/zsombor/thesis_zsombor_fulop/data/training

Number of classes:

1

-

+

ID of GPU to use:

0

-

+

Number of saver processes:

3

-

+

Start inferencing

Stop inferencing

Figure 2.20: Inference page

After choosing a .mp4 video file for getting detections on, you can choose your trained model's folder. Keep in mind that if you do not choose a model, a default model will be used, which was trained on general objects. Set the number of classes based on how many classes the model was trained on.

When you start inferencing, a similar progress bar will be shown, like when exporting frames. If you see a stuck progress bar, it means you either gave a false GPU ID or too many saver processes. While inference is in progress, you can use other things in the program.

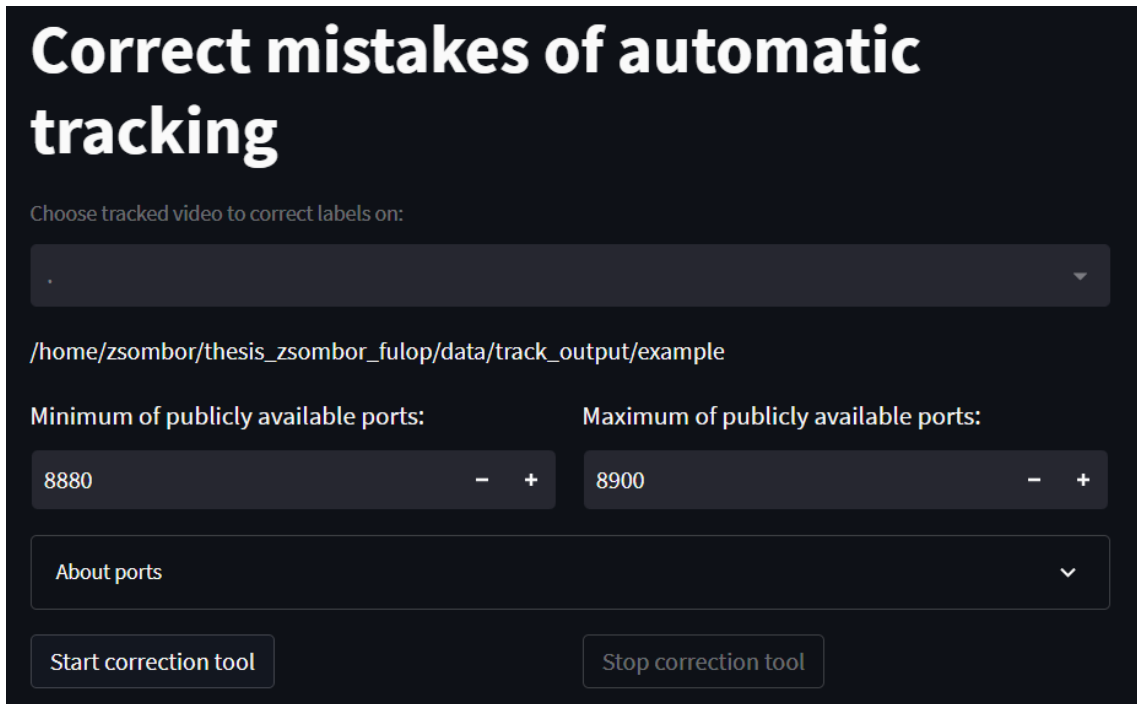
2.3.6 Track

Clicking on "Track" in the page menu takes the user to a page shown in Fig. 2.21. On this page, you can track the detections given to every frame. Unfortunately, the effective tracking method is still under research, so now, only an example can be tracked for demonstration.



Figure 2.21: Track detections

Tracking will make some mistakes, for example, when birds fly out of the camera. Human relabelling will be needed to handle these cases. It can be done with a tool called NIPGBoard, which can be started from an interface shown in Fig. 2.22.



Correct mistakes of automatic tracking

Choose tracked video to correct labels on:

/home/zsombor/thesis_zsombor_fulop/data/track_output/example

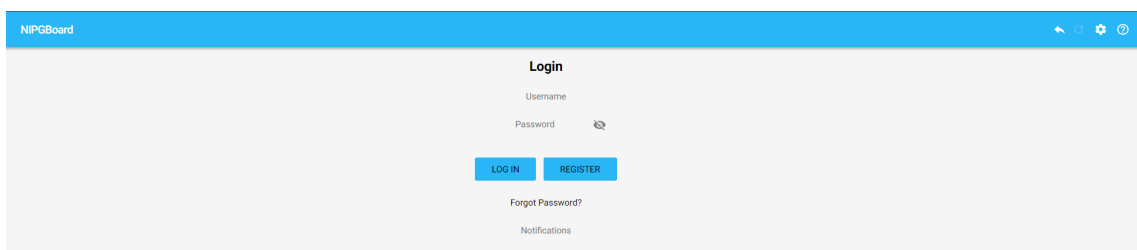
Minimum of publicly available ports: - +

Maximum of publicly available ports: - +

About ports

Figure 2.22: Correct detections mistakes

Choosing a custom tracking output is not yet possible because of lacking an effective tracking method. By now, the correction tool can be used on an example which is already set in the file picker. Start the correction tool, and the software will display an URL. Starting NIPGBoard may take some minutes. NIPGBoard is currently supported only in Google Chrome, so be sure you open it in that. You should see a page like shown in Fig. 2.23.



NIPGBoard

Login

Username

Password

[Forgot Password?](#)

[Notifications](#)

Figure 2.23: NIPGBoard login

If you cannot see the login surface, refresh the page. Do not register; use these credentials: user: asdASD123, password: asdASD123. After having logged in you will see a page shown in Fig. 2.24.

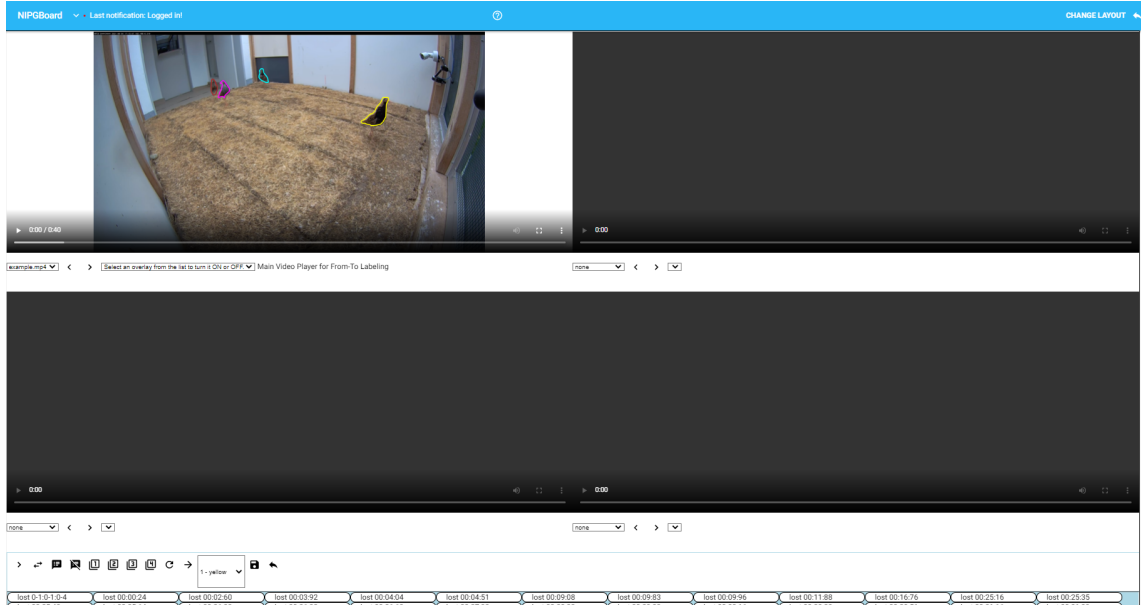


Figure 2.24: NIPGBoard after login

Clicking on the button "1" you can remove the unused video players. You can play the video and see how well the model found the instances. Below there are labels that show when the system has lost tracking. Going one by one from first to last you can relabel the birds. Click on the button next to "4" which will make it a pink background as shown in Fig. 2.25. Once you do it, you will be able to choose a colour from the dropdown menu, and by clicking on the bird relabel it. If you finished relabeling a frame push the button between the pink button and the dropdown menu. This will propagate the changes. If your relabelling is not correct, the system throws an error message. After correcting every label save it by clicking on the save button.



Figure 2.25: NIPGBoard relabelling toggled

2.3.7 Statistics

After having a video's tracking corrected, you can get statistics about how many birds are in which zone on average. Currently, they can be computed only for the example video, therefore, the file picker is disabled. Clicking on "Statistics" in the page menu takes the user to a page shown in Fig. 2.26.

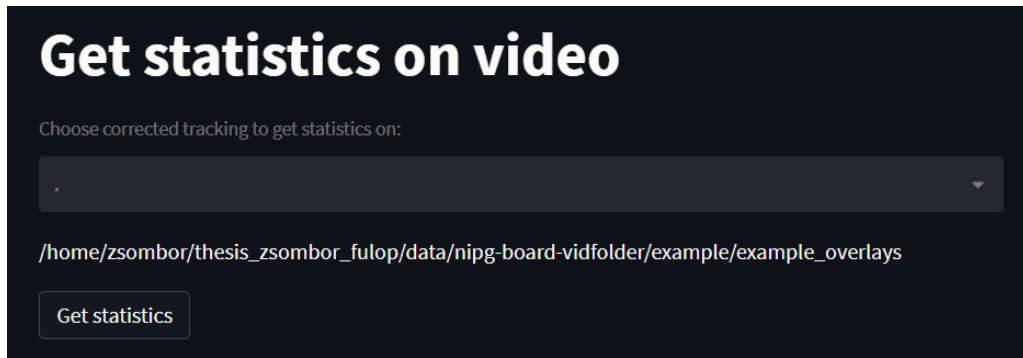


Figure 2.26: Statistics page

After getting statistics they will be written as shown in Fig. 2.27.

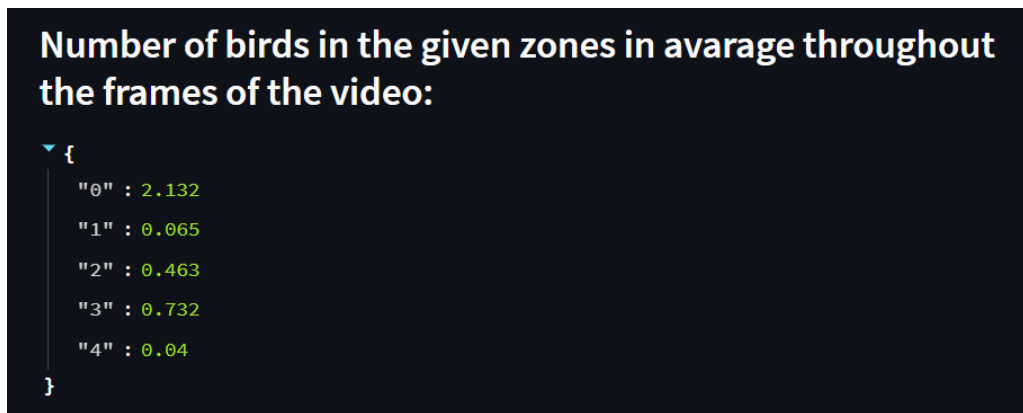


Figure 2.27: Statistics on video

The reason why so many birds are in zone "0" is that the tracking method is not good enough yet and it has lost the birds.

Chapter 3

Developer documentation

This chapter covers the software used, describes the underlying theoretical and technical concepts, and details the structure of my program. I present the various challenges that arised and my approaches to dealing with them.

3.1 Theoretical concepts

3.1.1 Artificial Intelligence

Artificial Intelligence (AI) has grown from several different fields, shown by Figure: 3.1. Most of these fields had already been dealing with intelligence concepts by the time the term AI was coined in a workshop in 1956.

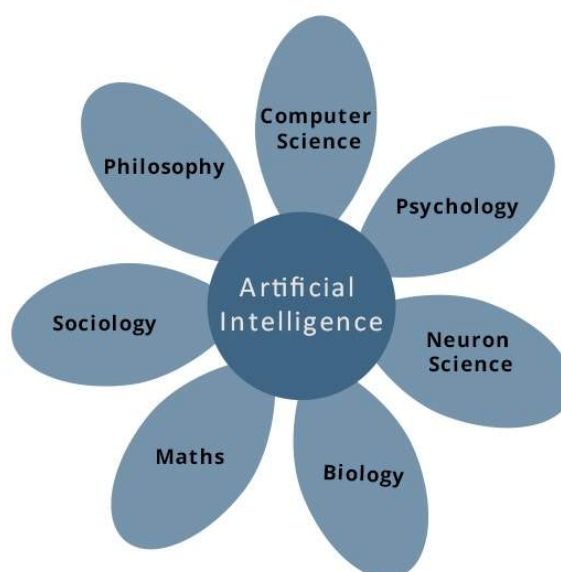


Figure 3.1: Foundations of AI, as presented in [7]

Artificial Intelligence: A modern approach [8], probably the most comprehensive book on AI, defines AI this way: *"The field of artificial intelligence, or AI, is concerned with not just understanding but also building intelligent entities—machines that can compute how to act effectively and safely in a wide variety of novel situations."*

Throughout the decades, numerous subfields were born, for example, search algorithms, knowledge representation, reasoning, machine learning, AI ethics, etc. Ever since AI as a term was made, its influence got more and more significant not only in pop culture but also in different branches of science.

My thesis deals with advanced AI concepts, so it is essential to have a brief knowledge of the theoretical background of these concepts.

3.1.2 Machine learning

Machine learning (ML) is a subfield of AI. Its main aim is to build models that can find a way to solve a given task just by analyzing some sample data. This imitates the way human learning works. [8]. There are three main approaches in traditional ML: supervised, unsupervised and reinforcement learning. In supervised learning, we provide input and corresponding output data (ground truth), so the model can generalise. In unsupervised learning, the model is not provided with any ground truth; it uses only the input data to learn how to find specific patterns or structures within its distribution. Reinforcement learning is a more complex paradigm in which we only give reinforcements to the model, which are certain rewards or punishments. [8]

In this project, we mainly focus on supervised learning methods. There are numerous approaches, such as decision trees, support vector machines, Bayesian networks, artificial neural networks, etc. It is important that we cannot call them algorithms in the traditional meaning as they do not explicitly describe how to produce the desired output.

Deciding whether the training was successful is not always an easy process. Before using the solution in production, it must be tested. There are specific ways to calculate several metrics to understand how well a model performs.

Machine learning was born for two reasons: some tasks are too complex to design a traditional algorithm for, and all possible future situations cannot always be foreseen. [8] Machine learning methods use computational power not only to solve

the task but also to find a solution for the task. By now, ML has become a standard part of software engineering.

3.1.3 Artificial neural networks

Artificial neural networks, or shortly ANNs, are a method within ML. They are a simplified model of the human nervous system. Their foundations were already laid in the 1940s and 50s [9, 8].

The building block of an ANN is the artificial neuron which can be described as a function $g: \mathbb{R}^n \rightarrow \mathbb{R}$. $x_1 \dots x_n$ are the inputs of the neuron and y the output. $w_1 \dots w_n \in \mathbb{R}$ are the so-called weights and $b \in \mathbb{R}$ is the bias. The basic concept of an artificial neuron is illustrated in Fig. 3.2. The function multiplies the input with the weights element-wise (vector multiplication), sums it up, adds the bias and calls function f on it. Function f is called the activation function, and it depends on the task of the neuron.

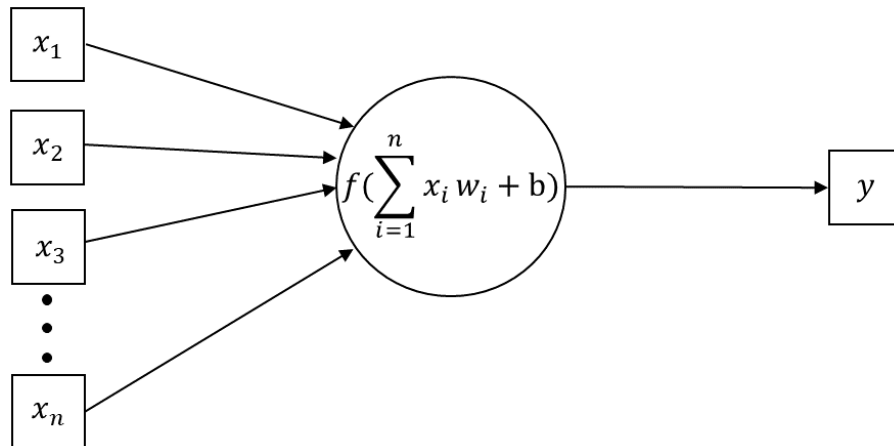


Figure 3.2: Illustration of an artificial neuron. x_i represent the inputs, y the output, function f stands for the activation functions. The w weights are used to find the mapping between the input and the output, with considering a b bias.

An ANN (Artificial Neural Network) consists of multiple layers of neurons and also multiple neurons in the layers. One neuron is not powerful enough, which is why we create a network of them, just like the human nervous system. The number of layers and neurons depends on the task, as well as the number of inputs and outputs. See a basic illustration in Fig. 3.3.

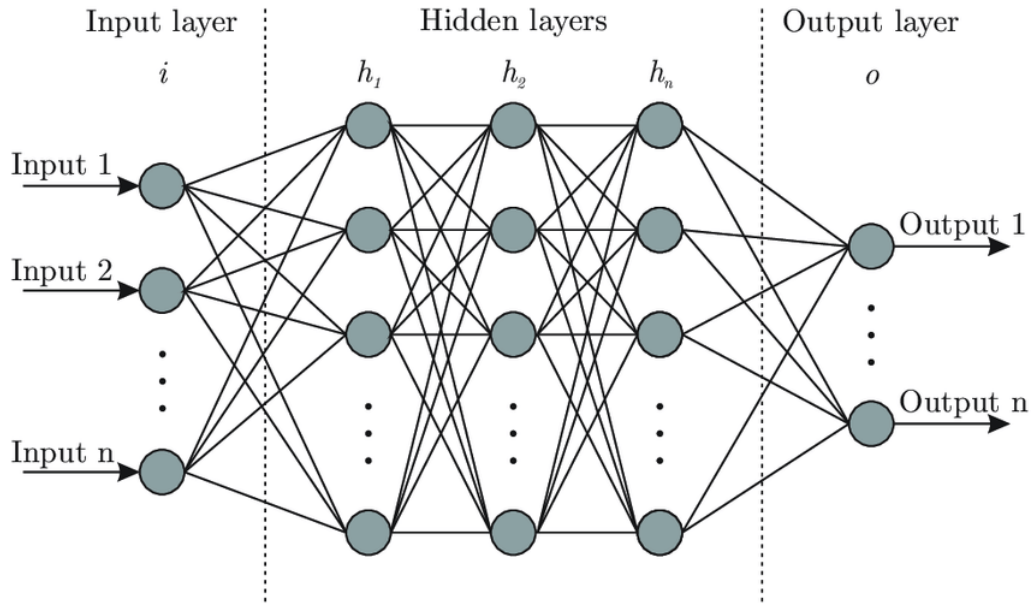


Figure 3.3: Illustration of an artificial neural network [10].

The central concept of ANNs differs very much from the traditional way of algorithms, which are usually a sequence of instructions that can be considered as a recipe to the machine on how to prepare the output. Training is the automatic process of setting the network's parameters such that it will solve the task for an unseen input with reasonable accuracy. Gradient descent [11] and backpropagation [12] are the most common strategies how to modify parameters during training.

In general, the more data we have (with a distribution similar to that of the expected unseen data), the better quality forecast we can expect from a trained ANN. Nevertheless, unfortunately, there are often limitations on the amount of available training data. A method called data augmentation is an approach to overcome this as a synthetic way to multiply our data. By augmentation, we traditionally mean a slight automatic modification of an input sample, which transforms it into another sample likely to stem from the same source. This often means cropping, rotating, flipping, blurring, or changing the image's colours in computer vision tasks.

3.1.4 Deep neural networks

The first implemented working ANN was made by Frank Rosenblatt in 1957 and could soon solve an elementary image classification task [13]. However, despite the early success, ANNs stayed for long theoretical concepts only. The problem was that solving real-life tasks end-to-end requires thousands of neurons in the network. Depending on the task, it means that millions of parameters must be optimised at

the same time during training. The rise of GPUs in 2000-2010 (and ever since) by leaps and bounds provided the required computational power for the ANNs and allowed them to spread widely. See a comparison between the floating point operation numbers of GPUs and CPUs in Fig. 3.4.

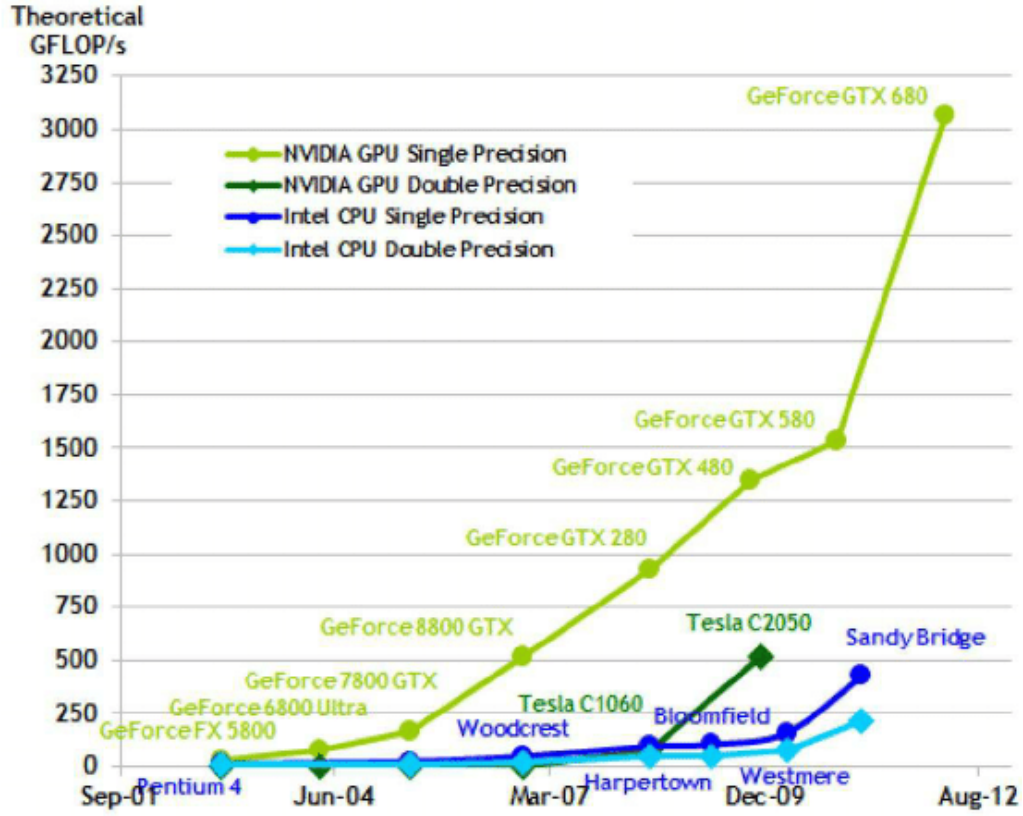


Figure 3.4: GPU floating point operation compared to CPU[14]

Although GPUs were designed to accelerate graphical computing, neural networks and graphical programs have much in common, namely a lot of parallelisable linear algebra operations. Nvidia recognized the potential of using GPUs for purposes other than graphical computing and created the Compute Unified Device Architecture (CUDA) in 2007. CUDA allows writing general-purpose parallel computing programs running on Nvidia GPUs.

Deep neural networks (DNNs) are a subset of ANNs. Roughly speaking, the more complex a task is, the more layers the ANN needs to solve it, meaning the network will be deeper. However, such complex networks could not be executed for long due to hardware limitations. The first implemented DNN is considered to be *AlexNet* [15] developed in 2012, containing 8 layers, 650000 neurons and 60 million parameters. *AlexNet* [15] won the *ILSVRC* [16] challenge, which was an annual competition for identifying objects in pictures. In 2015, the winner reached

superhuman performance, and since 2017 the competition is not held anymore as the task is now too easy.

Nowadays, DNNs are used in several complex tasks, including applications for image processing, natural language processing, medical data analysis, and recommendation systems.

3.2 Used technologies and tools

3.2.1 Python

Python is an interpreted, high-level, general-purpose programming language made in 1991. By the end of the 10s, Python became one of the most popular programming languages and, in 2022, the most popular one according to both of the two most significant rankings *PYPL* [17] and *TIOBE* [18].

Due to its simple syntax and very high-level features, such as dynamic typing and garbage collection, it has also become viral in scientific computing. Hiding such concepts as memory management and typing makes it suitable for non-programmer scientists to run simulations and experiments quickly and easily. Numerous packages have been made for several domains, both by organizations and communities.

However, high-level abstraction has its price; native Python structures are not very efficient. They are much slower than C/C++, for example. This is why Python is often used as a scripting language. Packages like NumPy [19], OpenCV [20], PyTorch [21] and TensorFlow [22] are all using fast precompiled C/C++, or even Fortran codes (on Nvidia GPU CUDA, on AMD GPU ROCm). Providing an easy-to-use API and still running efficiently with the help of these languages has provenly become the best practice. Due to all of the aforementioned, Python is used in plenty of ML and DNN projects.

3.2.2 Mask R-CNN and Detectron2

My task was instance segmentation, which aims to find different instances of a class (in my case, ruff) and label the pixels corresponding to each object with a unique ID. Instance segmentation is an area where DNNs can be exploited. As designing a proper network architecture is a difficult research question, I used an existing one. *Mask R-CNN* [3] is a DNN for object detection and segmentation in

images, designed in 2017. As the segmentation masks are predicted, the position tracking of the objects can be feasible.

Training and evaluating such large networks are very difficult for several reasons, even with high-level libraries like PyTorch or TensorFlow. That is why I did not implement them from scratch but used a good working framework that I could extend with the desired features. I used *Detectron2* [4], which supports many DNN architectures for various computer vision tasks, including keypoint detection, semantic, panoptic and instance segmentation. The architectures are implemented in PyTorch.

3.2.3 MiVOS and SegAnnot

Detectron2 is a supervised instance segmentation network, thus training it requires thousands of labelled images. However, labelling each frame can be a very time-consuming task, even with tools that enable drawing a polygon around the objects.

MiVOS [23] is a method to speed up the annotation of videos drastically by using DNNs to predict the mask of an object from a scribble drawn on it and propagating it through frames. The implementation provided alongside the article had although some severe limitations, such as failing with memory errors due to a lack of tracking memory usage and thus loading too much data. In *SegAnnot* [24], an extended *MiVOS* is introduced to resolve these limitations and to provide an easy-to-use UI.

3.2.4 Augmentation on light conditions

To improve the network’s generalisation and prediction accuracy, we apply colour augmentation on the training images. The applied augmentation is related to the project of [25] and is used without modification in this project. Transformations include gamma correction, a sepia transform, histogram equalization, and white balance adjustments using WBEmulator [26]. WBEmulator is an approach to change the white balance of an image by avoiding any non-realistic colours in the output, trained with deep learning methods.

3.2.5 Decord

Decord [5] is a video reader tool whose features, among many, are frame-accurate seeking and decoding on GPU with an easy-to-use API. Videos do not store each

frame as a separate image but rather encode the changes between frames. Decoding needs computational power, and with current CPUs, this is not notable if we only want to watch a video.

However, when using each frame for further heavy computation, we want our software to wait for the frames for the least time possible. Thus real-time decoding speed, in this case, may be too slow. To accelerate this, most current Nvidia GPUs have special hardware for video encoding and decoding. These are separate from the central GPU part (called CUDA cores). Using this hardware for video operation is very beneficial because it is not only faster than the CPU but also frees many CPU resources. Moreover, using GPU's decoding for DNNs has the advantage that the frames do not need to be copied from RAM (where the CPU would put it) to GPU memory, as they are already there after decoding. As copying between RAM and GPU memory is costly, it is wise to minimise it.

Another issue is to seek an exact frame accurately. Most libraries, including the widely used OpenCV's VideoCapture [27], do not guarantee perfect accuracy unless the video is read from the beginning to the end frame by frame.

3.2.6 TensorBoard

TensorBoard is a tool for visualising ANN training. During training, several metrics can be calculated, representing the accuracy of the model. These numbers, for example, can be written to the standard output but plotting them to a graph is more helpful for understanding, as provided by TensorBoard on a browser-based interface. Detectron2 by default sends some of the metrics to TensorBoard, though it can be extended as well. TensorBoard can also display input images, masks, visualisations and many other things.

3.2.7 NIPGBoard vz-labelvideo plugin

NIPGBoard [28] is a modified version of Tensorboard designed for quickly annotating based on the positive or negative association between object pairs, designed to allow annotation and training alike through ease of use. It also has a plugin called *vz-labelvideo*, which is suitable for quickly relabeling instances. This is needed in my project because perfect end-to-end tracking cannot be produced, so the user occasionally must correct the switched instance labels.

3.2.8 Streamlit

Streamlit is a Python package explicitly made to turn machine learning and data science Python scripts into responsive web applications. Using it does not require any web experience. The page elements (called widgets) can be created with Python function calls, which create the Javascript, HTML and CSS files in the background. Streamlit provides an executor program, so one does not run the script directly by a Python interpreter. However, Streamlit accepts everything that the Python interpreter would. With Streamlit, machine learning engineers and data scientist can quickly deploy their data-driven solutions into production without any web expertise. Of course, higher abstraction comes at a price; things that can be done in Streamlit are limited. The good thing is that if a needed widget is yet to exist in the framework, one can develop it with any web frontend technology. Adding a given interface will enable using it in Streamlit.

3.2.9 Program environment

A specific operating system, CUDA, Python, and other system packages are needed to run my program. I developed the project in a container to avoid dependency conflicts, make installation easier, and support portability.

Containers are similar to virtual machines (VM), providing system-level virtualisation. However, while VMs can easily use a different guest OS on the host OS, with containers, it is not natively supported. Containers can virtualise system libraries and anything lower than that. Container states can be saved to image files, which will have most of the things set up for the application. This grants fast app sharing. I used Apptainer [6], which is specially made for high-performance computing.

As opposed to Docker [29], which is also a widely spread container technology, Apptainer is more secure on user permissions. Still, the beneficial compatibility to Docker can be exploited by using a Docker image as a base image that already contains CUDA 11.3.1 and CUDNN 8 installation. The definition file of the image can be found in `cu113py38nvvidcodecsdk120baze10261.def`, and the image file `cu113py38nvvidcodecsdk120baze10261.sif`. Other system packages, such as Python, are also installed in it.

Dependency issues had to be solved between the software components, as SegAnnot, NIPGBoard, and the detection and tracking software components have

different dependencies. SegAnnot is suggested to run in an Anaconda environment [30], NIPGBoard is recommended running in a Python virtual environment(VE) [31], and the back-end codes also need a different python environment and package versions.

To overcome these issues, two separate VE and one Anaconda environment will be created when running `setup.sh`. Anaconda environment `segannot` will include every package needed for SegAnnot, whereas `nipg-board-v3` VE everything that is needed for NIPGBoard. In the second VE `main` every other python package will be included that is necessary for running my software.

SegAnnot, NIPGBoard, and WBEmulator will be used directly from source. Decord will be installed from source to enable GPU acceleration automatically in `setup.sh`.

3.3 Code components

The main structure of the project is illustrated in Fig. 3.5.

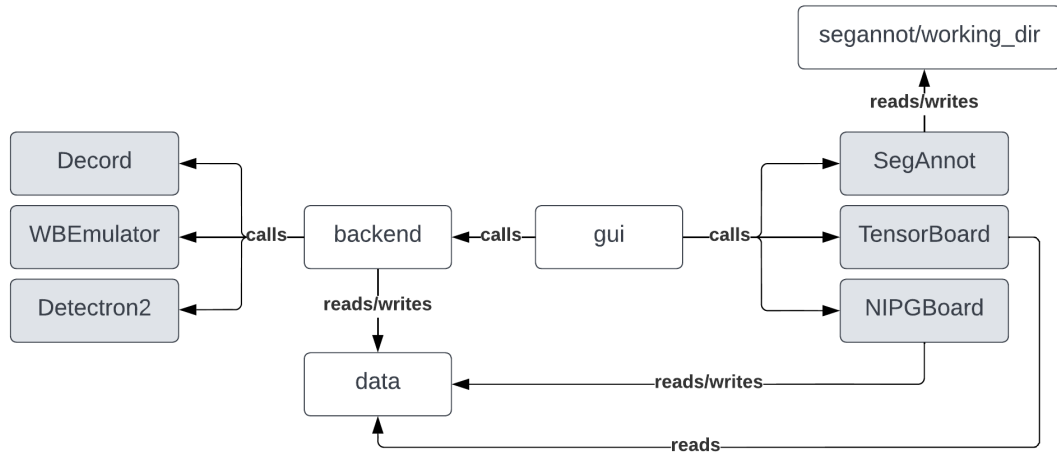


Figure 3.5: General structure of the code. Boxes with a grey background are the components from other software libraries.

Streamlit is designed to make an existing data-driven script into a functional web page only by adding some extra lines to the code. This approach treats the front-end and the back-end as one component, which is a straightforward approach, yet if the code is complex, it can make it messy and hard to understand. Therefore, I have separated front-end scripts from the back-end ones. Scripts in `gui` folder

are only responsible for making the user interface and calling the scripts in the *backend* folder. This way, the UI is involved only indirectly in the computations and data manipulation through the scripts of *backend*. The *gui* also calls SegAnnot as a component, which reads and writes data from its own working directory, and NIPGBoard, which reads and writes from the *data* directory.

3.3.1 Calling backend

Streamlit's default operating principle is reloading the whole page by rerunning the script from top to bottom whenever a widget's state changes due to user interaction. Of course, this would make Streamlit inappropriate for data apps, as it would unnecessarily rerun complex computations using large datasets. That is why Streamlit has a caching functionality. Whichever function is decorated with the `@streamlit.cache` decorator will tell Streamlit to run that function only once and cache its return value. Streamlit will only rerun the function once the parameters have changed.

However, this caching functionality is not suitable for my application. Whenever a directly called computation runs, the page is blocked; it is not interactive. We need to run multiple computations simultaneously, as for example inferencing the DNN model should be possible while the user corrects tracking mistakes. This lets the user process data much faster, as the machine can prepare the next video's predictions while the user is working on the previous one. The time needed for processing and human attention is relevant because the video footage is about 500 hours. Thus, instead of caching, my program starts the computations in processes fully independent of the Streamlit app. Calling background this way also grants that computations will still run on the server even if the client has disconnected. This is beneficial, for example, if the user wants to run long computations during the night, and process the outputs only the next day.

Although, to manipulate these processes from the Streamlit app their process IDs have to be determined, see function `gui\helper_functions.py` in Code 3.1.

```
1 from subprocess import check_output, CalledProcessError
2
3 def getPIDsWithParameters(command_with_parameters):
4     try:
```

```
5     # there might be more processes so we get them all
6     processes = check_output(["pgrep", "-af",
7                               command_with_parameters]).decode('utf8').split('\n')
8     processes = list(filter(None, processes)) # there would be a
9     # split by ' ' to get PID at first id, and also we need to
10    # remove again ' ' -> map it to all the lines
11    pidlist = list(map(lambda line: int(list(filter(None, line.
12    split(' '))))[0]), processes))
13
14    except CalledProcessError: #throws it if no process found
15        pidlist = []
16
17    return pidlist
```

Code 3.1: Getting PIDs of started processes

- Input: A string which is the prefix of the searched process.
- Output: A list of int, containing PIDs.
- Description: The function gets the PID(s) of the process(es) whose program name and program parameters match the provided prefix. It would be easy to get the PIDs with the `psutil` command; however, it cannot filter by program parameters, meaning that it could only return PIDs of every Python process. Instead, `pgrep -af` can search by the program's name and also its parameters. As the searched program might use multiprocessing, it may find more matches, thus returning a list of PIDs. If none is found, it returns an empty list.

In the next section, I explain through the example of starting and stopping SegAnnot how my code starts and stops the backend and other modules as subprocesses with the help of `subprocess.Popen` and `getPIDsWithParameters()`.

The SegAnnot, TensorBoard and NIPGBoard components are using ports to display their UIs. The program will not request the user to specify exact port numbers for the components to provide a user-friendly solution, just a range of ports within which the ports will be automatically assigned as needed.

3.3.2 Collecting labeled data

The structure of the code for labelling data is illustrated in Fig. 3.6.



Figure 3.6: Structure of the code for collecting labeled data

SegAnnot reads and writes from and to its own working directory. However, copying large video files costs time, and storing them redundant wastes storage. Therefore, I create a symbolic link to the video and store only the link in *segannot/working_dir*.

My code starts the scripts in the backend and other modules as shown in Code 3.2.

```

1 segannot_starter_proc = subprocess.Popen(f'cd {PROJ_DIR}/segannot
2     && {SEGANNOT_CONDA_PYTHON} app.py\
3     --host {external_ip} --port {port}
4     {"--half_precision" if half_precision else ""}',
5     shell=True)

```

Code 3.2: Starting a script as a subprocess with SegAnnot's example

Firstly, I had to `cd` into SegAnnot's directory, as with an absolute path, it failed due to some import errors. Every script will be started with its specific Python interpreter, in this case with `SEGANNOT_CONDA_PYTHON` which is the interpreter in SegAnnot's environment. This grants that every needed package for the script will be available as the interpreter is either from a Python virtual environment or an Anaconda environment. The program parameters will be provided after based on user inputs.

We cannot be sure whether the script started successfully. If it failed, its error message is not propagated to the Streamlit app because it would start as an independent subprocess. Therefore, I ensure I can find this process among the running processes as shown in Code 3.3.

```

1 from helper_functions import getPIDsWithParameters
2

```

```
3 def isSegannotRunning():
4     return len(getPIDsWithParameters(f"{SEGANNOT_CONDA_PYTHON} app.
        py")) != 0
5
6 def startSegannot(min_port, max_port, half_precision):
7     ...
8     running = False
9     for _ in range(10):
10         time.sleep(6)
11         if isSegannotRunning():
12             running = True
13             break
14     segannot_starter_proc.kill() # we kill the starter process
        because it would run till streamlit runs
15     segannot_starter_proc.communicate() # we need to call this
        otherwise it would remain a zombie process
16     ...
```

Code 3.3: Checking if script has started with SegAnnot's example

The code searches for the PID(s) of the started process until a time limit. If not found, an error has happened, which will be told to the user. Searching for only `SEGANNOT_CONDA_PYTHON app.py` without the parameters of `app.py` is enough because even if multiple users use the program on the same machine, `SEGANNOT_CONDA_PYTHON` will differ. It is an absolute path, and different installations will have different absolute paths. Also, searching with program parameters would only be possible by saving them to the disk, as the variables would be lost between reloads.

Moreover, the processes can be killed by sending `SIGINT` or `SIGKILL`. If the program is multiprocess, we kill all of the processes as we cannot ensure which is the parent process as shown in Code 3.4.

```
1 for pid in getPIDsWithParameters(f"{SEGANNOT_CONDA_PYTHON} app.py")
   :
2     os.kill(pid, signal.SIGINT)
```

Code 3.4: Killing running subprocesses with SegAnnot's example

3.3.3 Creating training data

The structure of the code for creating training data is illustrated in Fig. 3.7.

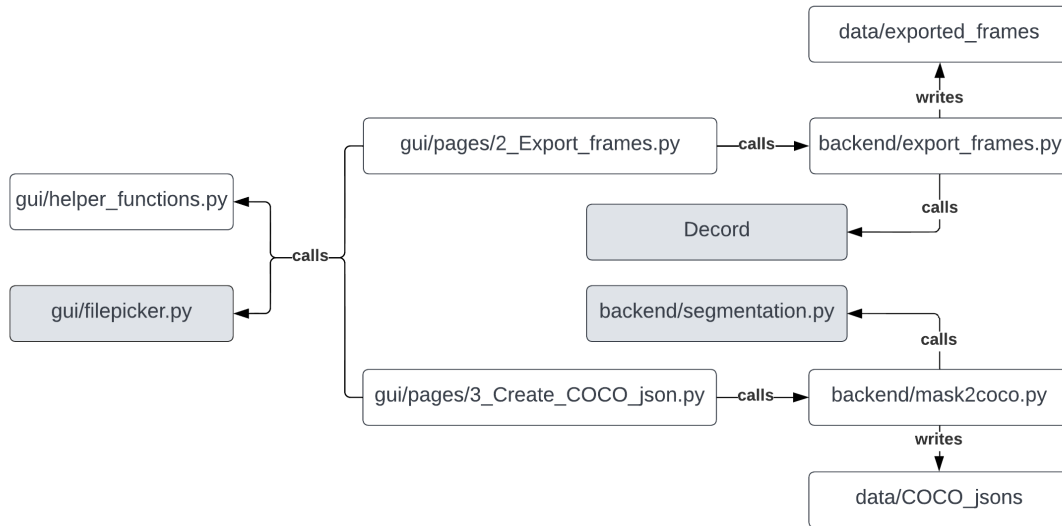


Figure 3.7: Structure of the code for creating training data

In order to train Detectron2, we need to have our training data either in a particular format or write a mapper function. I chose to have it in the particular format called COCO JSON. COCO format is the format of the Common Object in Context [32] dataset, which has become a well-used benchmark dataset for computer vision solutions. Thus, its format has also spread; many frameworks use it. The format is shown in Code 3.5.

```

1 {  "info": {
2      "year": "2022",
3      "version": "1.0",
4      "description": "Sample dataset",
5      "contributor": "Zsombor Fulop",
6      "url": "https://url.of.this.dataset.com",
7      "date_created": "2022-12-19T10:22:45"},
8  "licenses": [ {
9      "url": "https://url.of.licence.com",
10     "id": 0,
11     "name": "Name-of-licence"},
12     ... ],
13  "categories": [ {
14     "id": 0,

```

```
15         "name": "dog",
16         "supercategory": "animal" },
17     ... ],
18     "images": [ {
19         "id": 0,
20         "license": 1,
21         "file_name": "<filename0>.<ext>",
22         "height": 1520,
23         "width": 2688,
24         "date_captured": null },
25     ... ],
26     "annotations": [ {
27         "id": 0,
28         "image_id": 0,
29         "category_id": 2,
30         "bbox": [x1, y1, x2, y2],
31         "segmentation": [contour list of mask],
32         "area": area-of-bbox,
33         "iscrowd": 0 },
34     ... ]
35 }
```

Code 3.5: COCO JSON format

The user labelled frames of a video, so firstly, I need to export the frames of the video to be able to put their paths to the JSON. I use Decord for decoding the frames and save with cv2, as shown in Code 3.6. Decoding is really fast with GPU; saving is the bottleneck. To make saving faster, I used some multiprocessing. I save images into .png extension. This costs more storage than .jpg would. However, depending on the video's encoding, saving it into .jpg might result in (further) quality loss, which might influence the training of the neural network. To answer if it actually influences, further experimenting would be needed. Now, I assume there is enough storage to store a couple of thousands of training images and save them in .png to ensure the DNN gets the best quality pictures.

```
1 def export_frames(src, dest, from_frame, to_frame):
2     pool = multiprocessing.Pool(6)
3     vr = VideoReader(src, ctx=gpu(0))
4     for i in range(from_frame, to_frame):
5         frame = vr[i]
```

```
6         # Multiprocessing makes the saving much faster, approx 3x.
7         pool.apply_async(cv2.imwrite, args=(dest+'/' + str(i).zfill
            (6) + ".png", frame.asnumpy()[:, :, :-1]))
8     pool.close()
9     pool.join()
```

Code 3.6: Exporting frames of a video

I think it could be sped up more by parallelizing the data carriage from the GPU memory to the RAM as well. In the code, Decord returns the frame as an `NDArray` (Decord's default type) which is stored in the RAM. However, it can be set to return a GPU Torch tensor. Although, implementing this way would probably require a `Queue` and its size set appropriately depending on the video's size, GPU memory's size and the speed of the decoding unit. For simplicity, I use the method above, which is relatively fast (namely 0.57fps with NVIDIA TITAN RTX on a 1520x2688 H.264 encoded video).

backend/export_frames.py

- Input:
 - `src` : `string`, path of source video.
 - `dest` : `string`, path of folder to save exported frames.
 - `from_frame` : `int`, frame number to start exporting frames from (inclusive).
 - `to_frame` : `int`, frame number to export frames to (exclusive).
- Output: .png files in the destination folder specified.
- Description: Exports frames from a given video from a given range into .png files in the given folder. It uses a fast implementation using GPU decoding.

After having the frames exported, the user can create the COCO JSON. The GUI calls `mask2coco.py`, which is based on a solution of a previous project of the NIPG. I modified this script by allowing to split the whole dataset into training and validation set, as validating while training the network can be very beneficial. I also corrected some minor mistakes in the code.

backend/mask2coco.py

- Input:
 - `img_folder_path` : list of `string`, paths to the folders of images.
 - `mask_folder_paths` : list of `string`, paths to the folders of masks.
 - `json_path_train` : `string`, path to file to put training JSON to.
 - `json_path_val` : `int`, path to file to put validation JSON to.
 - `train_ratio` : `float`, 0-1 ratio between train and validation samples.
 - `annot_names` : list of `string`, the list of names for the annotations.
- Output: Two COCO formatted JSON files in the given location.
- Description: Reads SegAnnot masks, transforms them into COCO annotations by transforming masks to segmentation lists, calculating bounding boxes and areas. Creates training and validation COCO JSON to the given locations, splitting samples randomly to the training and validation set based on the given ratio.

3.3.4 Training model

The structure of the code for training the model is illustrated in Fig. 3.8.

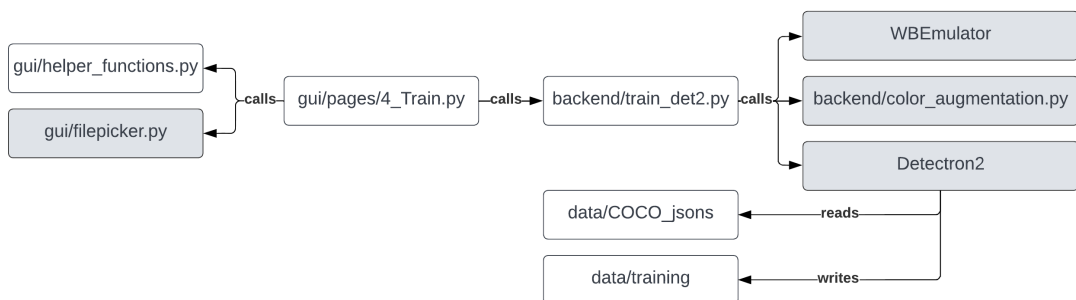


Figure 3.8: Structure of the code for training model

I based my code and extended Detectron2 API's sample training script `train_net.py` with the following:

- I added several types of augmentation to the training data including a light augmentation based on `WBEulator` and `backend/color_agumentation.py` which was made by my supervisor Anna Gelencsér-Horváth.
- I extended the API's `COCOEvaluator` class to not only calculate the metrics on the validation set but also save the predictions from which the metrics were calculated.
- I added loss calculation on the validation set based on a solution found on internet. [33]
- I added program parameters to customize training.

I fixed some parameters; the user is not able to set all of them as it is unnecessary.

`backend/train_det2.py`

- Input:
 - `train_annot` : `string`, path to the JSON file containing the annotations of the training set.
 - `val_annot` : `string`, path to the JSON file containing the annotations of the validation set.
 - `out_dir` : `string`, path to the folder where *Detectron2* [4] saves the output files.
 - `model_dir` : `string`, path to folder containing base model's checkpoints.
 - `model_ckpt` : `string`, base model's checkpoint.
 - `n_epoch` : `int`, number of epochs to train for.
 - `lr` : `float`, learning rate.
 - `n_classes` : `int`, number of classes to train for.
 - `n_workers` : `int`, number of `DataLoader` workers.
 - `batch_size` : `int`, size of batch.
 - `eval_period` : `int`, frequency of evaluation.
 - `ckpt_period` : `int`, frequency of saving model.
 - `vis_period` : `int`, frequency of minibatch visualisation.

- resume : if set training continues.
 - eval-only : if set only evaluation happens.
 - num-gpus : `int`, number of GPUs to use. Currently only one is supported.
 - num-machines : `int`, number of machines to use. Currently only one is supported.
 - machine-rank : `int`, rank of machine. Currently only default is supported.
 - dist-url: `string`, distributed backend URL. Currently only default is supported.
- Output: Models, log files, and evaluation results saved to the specified folder.
 - Description: Trains or evaluates a Mask R-CNN model with a setting depending on the input parameters.

3.3.5 Model inference

The structure of the code for inferencing the model is illustrated in Fig. 3.9.

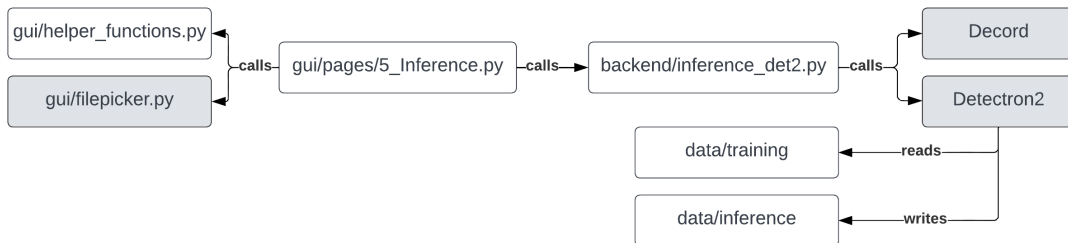


Figure 3.9: Structure of the code for inferencing model

Model inference is the machine learning pipeline’s end; after training it, we want to use it in production. As the inference runs many times, it is essential to have it highly optimized.

Detectron2 API has a `DefaultPredictor` class which does model inference for a single image. As it is written in the documentation, this class should only be used for demo purposes because it is not optimised. However, I use it as a baseline to understand how much my code could optimise. I want to call it on a video’s

frames, but exporting them in advance takes much time and storage, so I get them on-demand with OpenCV's `VideoCapture`, as it is the most common tool in most tutorials available.

In such an example, the following things happen sequentially for every frame:

1. Reading a frame from the video (decoding from video).
2. Moving the frame from RAM to GPU memory.
3. Doing a model inference on the frame on GPU.
4. Moving predictions from GPU memory to RAM.
5. Transforming predicted masks to segmentation lists for saving.
6. Writing predictions to disk.

During execution, we can see the things shown in Fig. 3.10 and Fig. 3.11 when checking GPU performance with `watch -n 0.1 nvidia-smi`.

1	NVIDIA TITAN RTX	Off	00000000:41:00.0 Off	N/A
40%	51C	P2	68W / 280W	6369MiB / 24219MiB
				0%
				Default
				N/A

Figure 3.10: GPU does not work

1	NVIDIA TITAN RTX	Off	00000000:41:00.0 Off	N/A
41%	52C	P2	94W / 280W	6369MiB / 24219MiB
				46%
				Default
				N/A

Figure 3.11: GPU works

Most of the time, the GPU is not in use at all. Predicting on a 1520x2688 H.264 encoded video using an Nvidia RTX TITAN, the GPU is used as in the second picture only every 5-5.5 seconds, and even then, only around 46% exploited. So it means that the GPU is waiting for the CPU the rest of the time, and when it gets the task, it is still too "easy" for it.

Checking CPU usage with `htop` shows that only one CPU thread works from the available twenty-four being heavily overloaded. It can be seen in Fig. 3.12.

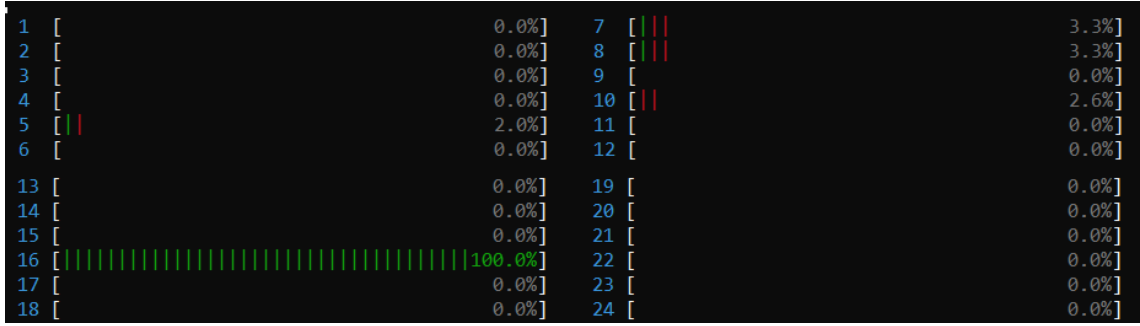


Figure 3.12: Single CPU thread works

As expected, the overall runtime is not very satisfying, 6100 seconds \approx 102 minutes for the sample video. The video was 25 FPS, which means that 40 seconds of video was processed in 102 minutes. Therefore, processing was approximately 152 times slower than real-time, which is a very poor performance considering that Nvidia RTX TITAN is a strong GPU.

There are several methods to optimise the code:

1. Separate decoding frames from the main process, so GPU CUDA cores do not need to wait for it.
2. Separate data post-processing and writing to disk from the main process, so GPU CUDA cores do not need to wait for it.
3. Minimise data transfer between GPU memory and RAM.
4. Inference model on batches.

I used Decord in my solution for decoding the video on GPU. The hardware that does this computation is separate from the CUDA cores, so it does not affect the model inference. Furthermore, with this approach, the frames, after being decoded, will already be in form of CUDA tensors in the GPU memory, so I could spare moving the input frames from RAM.

However, moving the output data from the GPU memory to the RAM cannot be spared, as the CPU needs to save them. Also, the predicted masks need to be encoded into segmentation lists so that the disk usage will also be minimised. Therefore, I separated it from the main process and used multiple processes to handle the model's outputs.

I did not separate the decoding from the main process; decoding a frame and doing model inference is still sequential. However, GPU-acceleration has sped up so much the decoding that its runtime is no longer significant. For example, decoding one frame from the sample 1520x2688 H.264 encoded video was on average 0.0016 second, which means that for a 25 FPS 1 hour long video, the CUDA cores will only wait for about $0.0016 * 25 * 60 * 60 = 144$ seconds. This, compared to the whole runtime, is relatively insignificant.

Some models, including *ResNet-50* [34], which is a backbone of the *Mask R-CNN* [3] implementation that we use, can have a higher throughput if they do batch inference. [35] This results from the fact that the GPU can cache some things when inferencing in batch, and does not need to read it again and again from GPU memory. Of course, having batches increases the GPU memory usage, as multiple input data must be stored at the same time, but in Fig. 3.13, it can be seen that more than half of the GPU memory is still free. However, in our project, my code's performance was enough, so I did not further investigate it.

On average, my code's overall runtime for the sample video is 159 seconds, about 38 times faster than the baseline solution and only around four times slower than real-time.

During execution, we can see something like what is shown in Fig. 3.13 when checking GPU performance with `watch -n 0.1 nvidia-smi`.

0	NVIDIA	TITAN	RTX	Off	00000000:09:00.0	Off	N/A
55%	77C	P2	299W / 280W		10765MiB / 24220MiB	97%	Default
							N/A

Figure 3.13: GPU works heavily

The GPU usage does not fluctuate (to be precise, it does fluctuate a bit because of the video decoding, but that is not significant). The CUDA cores work almost all of the time.

CPU can also work more without waiting for GPU; moreover, multiple threads are active, meaning successful parallelisation as shown in Fig. 3.14.

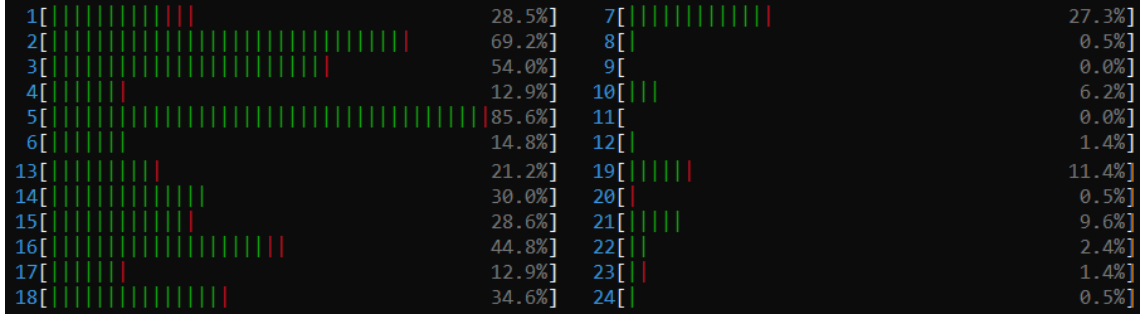


Figure 3.14: Multiple CPU threads work

It is also vital that my code is also scaleable. If there were more GPUs in the machine, with minimal code changing, they could be used. Decord plays an essential role as it provides frame-accurate seeking. Say there were two GPUs available. In this case, one GPU would process the first half of the video while the second GPU the second half. CPU resources are far not fully exploited, so they could easily handle outputs of multiple GPUs.

backend/inference_det2.py

- Input:

- video_path : **string**, path of the video to run model inference on.
- base_out_dir : **string**, path to the folder where predictions will be saved. Detectron2 saves the output files.
- model_dir : **string**, path to folder containing the checkpoints of the model to be used.
- model_ckpt : **string**, checkpoint name of the model to be used.
- frame_start : **int**, frame number to start predicting from. (inclusive)
- frame_end : **int**, frame number to predict to. (exclusive)
- n_classes : **int**, number of classes to train for.
- class_names : list of **string**, class names for visualisation.
- crop_from_x : **int**, X coordinate of the top left edge of the area that you should be predicted on.
- crop_from_y : **int**, Y coordinate of the top left edge of the area that you should be predicted on.

- `crop_to_x : int`, X coordinate of the bottom right edge of the area that you should be predicted on.
 - `crop_to_y : int`, Y coordinate of the bottom right edge of the area that you should be predicted on.
 - `save_mask : if set`, saves masks in an image.
 - `visualize : if set`, saves masks on the input image.
 - `gpu_id : int`, ID of GPU to use.
 - `n_saver_proc : int`, number of saver processes.
- Output: Predictions (visualisations) saved to the given output.
 - Description: Decodes frames of the specified video and does model inference of them with the given model. Saves the predictions as well as visualisations if specified.

3.3.6 Track detections and correct mistakes

The structure of the code for tracking detections and correcting mistakes is illustrated in Fig. 3.15.

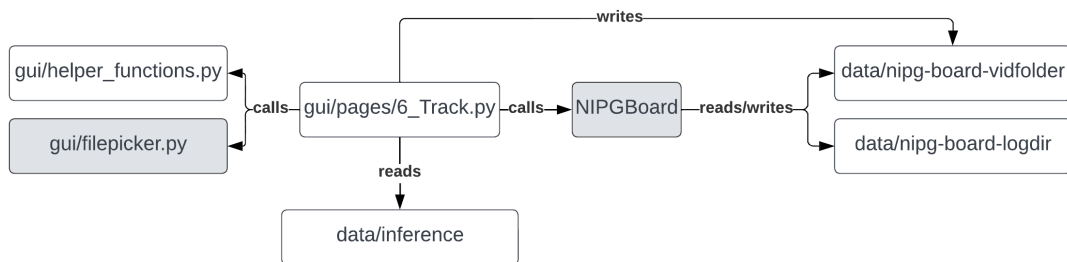


Figure 3.15: Structure of the code for tracking detections and correcting mistakes

A reliable tracking method based on detections is under research as of the time of writing this document. Therefore, the GUI has yet to call any backend code. Instead, it reads a demo tracking output that needs improvement. However, relabelling with NIPGBoard is possible using this file.

3.3.7 Get statistics

The structure of the code for getting statistics is illustrated in Fig. 3.16.

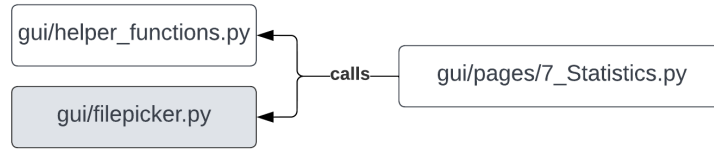


Figure 3.16: Structure of the code for getting statistics

This code works on the demo output of tracking without relabelling. Due to the simplicity of this calculation, the GUI does not call any backend code yet. The code computes how many ruffs are in the zones defined by ornithologists on average. Because of poor tracking and the lack of relabelling, there are too many ruffs in zone 0 which corresponds to the lost state.

3.4 Testing

My thesis program unites multiple complex systems, which limits testing to manual testing. Furthermore, ANN training and inferencing cannot be tested the traditional way.

3.4.1 Backend

`backend/segmentation.py`, `backend/color_augmentation.py`

These codes were taken from another project, and I did not modify them at all, so we assume they work appropriately.

`backend/export_frames.py`

Decord is assumed to work well. The multiprocessing can be tested by checking time performance, and the output folder to ensure all images are saved.

`backend/mask2coco.py`

This code was also taken from another project; however, I extended it with some features. I also corrected a bug that was present before my modification. It stemmed from the fact that SegAnnot sometimes propagates one-pixel big masks, which then

will be transformed into an empty segmentation list. This will later fail training, so I threw such instances. This script can be tested by visualising the annotations that will be in the output COCO JSONS. Drawing bounding boxes and masks on the corresponding images will show if something is malfunctioning in the code.

Also, training validates if this script works well. In the training script, `detectron2.data.datasets.register_coco_instances` is called on the COCO JSONS, which fails if something is wrong with the generated file.

```
backend/train_det2.py
```

I use Detectron2 framework for training. The framework fails if something is not good in the script. Furthermore, if the model is not learning, using logs or TensorBoard can help understand why.

```
backend/inference_det2.py
```

I use Detectron2 framework for inferencing, which shows many errors. The multiprocessing logic can be validated by checking time performance and output files. `torch.multiprocessing.apply_async` calls usually fail silently so if something does not work as expected they should be rewritten to synchronous `torch.multiprocessing.apply` calls to see the error message. If CUDA tensor sharing principles are violated, they are also shown. Uncommenting some print statements in the code may also help find the source of malfunctioning.

3.4.2 Frontend

I assume that Streamlit's frontend generation is correct. The logic behind parameter checking and disabling elements should be tested as well as successful backend launching. This is possible by manually checking if the use cases are handled in the way they are written in the user documentation.

3.5 Further development

- Solving that training script works with multiple GPUs in one machine and possibly multiple machines. Currently, the code runs on deadlock when trying to use multiple GPUs.
- Solving that inference script works with multiple GPUs. This should be a relatively easy task as foundations of scalability are already provided.
- Let the user choose multiple videos for prediction. With this functionality, multiple videos could get predictions overnight, allowing the user to work with complete predictions the next day.
- Make it possible that GUI catches the output of the backend and possibly handle errors.
- Provide a file picker that is easier to use.
- Make `mask2coco.py` faster possibly by multiprocessing.

Chapter 4

Conclusion

I managed to integrate multiple programs with various dependencies into one pipeline, in which they work in harmony. In addition, I have made a simple graphical user interface, which can work from almost any client machine running a browser. These make the software suitable for end-to-end usage without the help of an expert, thus allowing ornithologists to use complex AI methods to accelerate their research.

Furthermore, I have succeeded in optimising crucial parts of the pipeline, making it fast besides reliable. Highly optimal code not only speeds up the runtime but also decreases the hardware needs.

Bibliography

- [1] Alfonso Pérez-Escudero et al. “IdTracker: Tracking individuals in a group by automatic identification of unmarked animals”. In: *Nature methods* 11 (June 2014). DOI: 10.1038/nmeth.2994.
- [2] Alvaro Rodriguez et al. “ToxTrac: A fast and robust software for tracking organisms”. In: *Methods in Ecology and Evolution* 9 (Mar. 2018), 460–464. DOI: 10.1111/2041-210x.12874.
- [3] Kaiming He et al. “Mask R-CNN”. In: (Mar. 2017).
- [4] Yuxin Wu et al. *Detectron2*. <https://github.com/facebookresearch/detectron2>. Accessed: 2022-12-29. 2019.
- [5] Distributed Machine Learning Community. *Decord*. <https://github.com/dmlc/decord>. Accessed: 2022-12-29.
- [6] *Apptainer website*. <https://apptainer.org>. Accessed: 2022-12-29.
- [7] *Artificial Intelligence - Overview*. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_overview.htm. Accessed: 2022-12-29.
- [8] Peter Norvig Stuart Russel. *Artificial Intelligence: A Modern Approach (Pearson Series in Artificial Intelligence)*. 4th US ed. Pearson, 2020. ISBN: 9780134610993.
- [9] *What are neural networks?* <https://www.ibm.com/topics/neural-networks>. Accessed: 2022-12-29.
- [10] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. “Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks”. In: *Energy and Buildings* 158 (Nov. 2017). DOI: 10.1016/j.enbuild.2017.11.045.

- [11] *Gradient descent*. <https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/optimizing-multivariable-functions/a/what-is-gradient-descent>. Accessed: 2022-12-29.
- [12] *Backpropagation*. <https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd>. Accessed: 2022-12-29.
- [13] *Explained: Neural networks*. <https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414>. Accessed: 2022-12-29.
- [14] Ari Harju et al. “Computational Physics on Graphics Processing Units”. In: Oct. 2012. ISBN: 978-3-642-36802-8. DOI: 10.1007/978-3-642-36803-5_1.
- [15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.
- [16] *ILSVRC website*. <https://image-net.org/challenges/LSVRC/>. Accessed: 2022-12-29.
- [17] *PYPL website*. <https://pypl.github.io/PYPL.html>. Accessed: 2022-12-29.
- [18] *TIOBE website*. <https://www.tiobe.com/tiobe-index/>. Accessed: 2022-12-29.
- [19] *NumPy website*. <https://numpy.org>. Accessed: 2022-12-29.
- [20] *OpenCV website*. <https://opencv.org>. Accessed: 2022-12-29.
- [21] *PyTorch website*. <https://pytorch.org>. Accessed: 2022-12-29.
- [22] *TensorFlow website*. <https://www.tensorflow.org>. Accessed: 2022-12-29.
- [23] Ho Cheng, Yu-Wing Tai, and Chi-Keung Tang. “Modular Interactive Video Object Segmentation: Interaction-to-Mask, Propagation and Difference-Aware Fusion”. In: (Mar. 2021).
- [24] *SegAnnot repository*. <https://bitbucket.org/nipg/segannot/src/master/>. Accessed: 2022-12-29.
- [25] Anna Gelencsér-Horváth et al. “Tracking Highly Similar Rat Instances under Heavy Occlusions: An Unsupervised Deep Generative Pipeline”. In: *Journal of Imaging* 8.4 (2022). ISSN: 2313-433X. DOI: 10.3390/jimaging8040109. URL: <https://www.mdpi.com/2313-433X/8/4/109>.

- [26] Mahmoud Afifi and Michael S. Brown. “What Else Can Fool Deep Learning? Addressing Color Constancy Errors on Deep Neural Network Performance”. In: *The IEEE International Conference on Computer Vision (ICCV)*. 2019.
- [27] *OpenCV VideoCapture*. https://docs.opencv.org/3.4/d8/dfe/classcv_1_1VideoCapture.html. Accessed: 2022-12-29.
- [28] *NIPGBoard repository*. <https://bitbucket.org/nipg/nipg-board-v3/src/master/>. Accessed: 2022-12-29.
- [29] *Docker website*. <https://www.docker.com>. Accessed: 2022-12-29.
- [30] *Anaconda website*. <https://www.anaconda.com>. Accessed: 2022-12-29.
- [31] *Python virtualenv website*. <https://docs.python.org/3/tutorial/venv.html>. Accessed: 2022-12-29.
- [32] *COCO dataset website*. <https://cocodataset.org>. Accessed: 2022-12-29.
- [33] *Training Detectron2 with validation loss*. <https://eidos-ai.medium.com/training-on-detectron2-with-a-validation-set-and-plot-loss-on-it-to-avoid-overfitting-6449418fbf4e>. Accessed: 2022-12-29.
- [34] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: 7 (Dec. 2015).
- [35] *Batch inference*. <https://www.eenewseurope.com/en/resnet-50-a-misleading-machine-learning-inference-benchmark-for-megapixel-images/>. Accessed: 2022-12-29.

List of Figures

2.1	The application started successfully	8
2.2	Start page	9
2.3	Advanced options	9
2.4	Settings	9
2.5	Annotation page	10
2.6	Annotation page when SegAnnot is running	11
2.7	Annotation tool opened	11
2.8	Export frames page	12
2.9	Export frames exporting running	12
2.10	Error while exporting frames due to invalid range of frames	13
2.11	Create COCO JSON page	14
2.12	Multiple frame folders added	14
2.13	Train page	15
2.14	Tensorboard URL	16
2.15	Tensorboard no data exists to display	16
2.16	Tensorboard has data to display	17
2.17	Good training loss	17
2.18	Good validation loss	17
2.19	Tensorboard shows reasonable average segmentation precision	18
2.20	Inference page	18
2.21	Track detections	19
2.22	Correct detections mistakes	20
2.23	NIPGBoard login	20
2.24	NIPGBoard after login	21
2.25	NIPGBoard relabelling toggled	21
2.26	Statistics page	22
2.27	Statistics on video	22

3.1	Foundations of AI, as presented in [7]	23
3.2	Illustration of an artificial neuron.	25
3.3	Illustration of an artificial neural network [10].	26
3.4	GPU floating point operation compared to CPU[14]	27
3.5	General structure of the code	32
3.6	Structure of the code for collecting labeled data	35
3.7	Structure of the code for creating training data	37
3.8	Structure of the code for training model	40
3.9	Structure of the code for inferencing model	42
3.10	GPU does not work	43
3.11	GPU works	43
3.12	Single CPU thread works	44
3.13	GPU works heavily	45
3.14	Multiple CPU threads work	46
3.15	Structure of the code for tracking detections and correcting mistakes .	47
3.16	Structure of the code for getting statistics	48

List of Codes

3.1	Getting PIDs of started processes	33
3.2	Starting a script as a subprocess with SegAnnot's example	35
3.3	Checking if script has started with SegAnnot's example	35
3.4	Killing running subprocesses with SegAnnot's example	36
3.5	COCO JSON format	37
3.6	Exporting frames of a video	38