ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RSL Robotic Systems Lab
Prof. Dr. Marco Hutter

Semester Thesis

Object tracking and segmentation using Video Transformers

Semester project

Spring Term 2022

Supervised by:
Lorenzo Terenzi
Julian Nurbert
Prof. Dr. Marco Hutter

Author:
Roberto Pellerito

Declaration of Originality

I hereby declare that the written work I have submitted entitled

Object tracking and segmentation using Video Transformers

is original work which I alone have authored and which is written in my own words.[1]

Author(s)

Roberto	Pellerito

Student supervisor(s)

Lorenzo	Terenzi
Julian	Nubert

Supervising lecturer

Marco	Hutter

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf`). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

[1] Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Intellectual Property Agreement

The student acted under the supervision of Prof. Hutter and contributed to research of his group. Research results of students outside the scope of an employment contract with ETH Zurich belong to the students themselves. The results of the student within the present thesis shall be exploited by ETH Zurich, possibly together with results of other contributors in the same field. To facilitate and to enable a common exploitation of all combined research results, the student hereby assigns his rights to the research results to ETH Zurich. In exchange, the student shall be treated like an employee of ETH Zurich with respect to any income generated due to the research results.
This agreement regulates the rights to the created research results.

1. Intellectual Property Rights

1. The student assigns his/her rights to the research results, including inventions and works protected by copyright, but not including his moral rights ("Urheberpersönlichkeitsrechte"), to ETH Zurich. Herewith, he cedes, in particular, all rights for commercial exploitations of research results to ETH Zurich. He is doing this voluntarily and with full awareness, in order to facilitate the commercial exploitation of the created Research Results. The student's moral rights ("Urheberpersönlichkeitsrechte") shall not be affected by this assignment.
2. In exchange, the student will be compensated by ETH Zurich in the case of income through the commercial exploitation of research results. Compensation will be made as if the student was an employee of ETH Zurich and according to the guidelines "Richtlinien für die wirtschaftliche Verwertung von Forschungsergebnissen der ETH Zürich".
3. The student agrees to keep all research results confidential. This obligation to confidentiality shall persist until he or she is informed by ETH Zurich that the intellectual property rights to the research results have been protected through patent applications or other adequate measures or that no protection is sought, but not longer than 12 months after the collaborator has signed this agreement.
4. If a patent application is filed for an invention based on the research results, the student will duly provide all necessary signatures. He/she also agrees to be available whenever his aid is necessary in the course of the patent application process, e.g. to respond to questions of patent examiners or the like.

2. Settlement of Disagreements

Should disagreements arise out between the parties, the parties will make an effort to settle them between them in good faith. In case of failure of these agreements, Swiss Law shall be applied and the Courts of Zurich shall have exclusive jurisdiction.

Place and date	Signature

Contents

Preface

What follows is the report of the project *Object tracking and segmentation using Video Transformers*. The main idea behind this scientific work is to report all the work done for the semester project entitled *Object tracking and segmentation using Video Transformers*.
The report focus on the process through which we enhanced the capabilities of the existing machine learning model for panoptic segmentation and how we trained an object detection model for people tracking to be used on *RSL autonomous excavator*.

Abstract

We refers to **panoptic segmentation** as one of the computer vision tasks that aims at separating all different objects in the image into individual segments. Similarly **Object tracking** is referred as one of the computer vision tasks that focus on following the object on the scene.
Object tracking and panoptic segmentation are two common tasks that aims at solving challenging situations that we commonly encounter in a robotic platform. we often need to recognize objects on stage and understand their shapes and positions for planning robot trajectory or guiding the end-effector.
Tipically panoptic segmentation algorithms and object tracking algorithms cannot be used efficiently because they are computationally expensive, they run at low frequency and they require a substantial amount of memory.
In this work, we investigate a solution to speed up model inference of novel vision Transformers for segmentation and object tracking.
First, we focus on state of the art network used for object detection and image segmentation: DETR. We develop an approach to convert the model from PyTorch to TensorRT that can reduce the computational cost and memory footprint significantly.
Second, we investigate DETR's potential towards online object instance segmentation by investigating TrackFormer, a model that by itself makes use of DETR. We can show promising results in the video segmentation domain, and train the model using small datasets for tracking a specific objects.

Chapter 1

Introduction

Throughout this work we look into the latest neural networks for images and video segmentation[1] and tracking.
An example of human tracking can be seen in 1.1.B.
We focus specifically on transformer models [2], a kind of machine learning model developed for Natural Language Processing that is recently becoming more pervasive inside computer vision algorithms.
Different Transformers are currently implied to solve computer vision tasks, some of them are Swin Transformer [3] and DEtection TRansformer [4]. Throughout this work we will focus on DETR [4].
We explore panoptic segmentation 1.1.A because we often encounter similar challenges in robotics: recognize objects on the scene and understand their shapes and positions through onboard cameras.
Solving these problems is particularly useful in a robot like *RSL autonomous excavator*, for planning the trajectory on terrain or guiding its end-effector.
Often additional Information are required for an efficient perception of the environment, such as position of an object on the scene at each time step.
A panoptic segmentation model that uses only images can not encode temporal Information, we can lost the object track and the object identity is not unique.
For these reasons we firstly enhance the existing panoptic segmentation model and then we focus on object instance segmentation and tracking.
An example of object instance segmentation and tracking can be seen in 1.1.C
The currently used panoptic segmentation model is an implementation of a DEtection TRansoformer [4].
This model was firstly converted from PyTorch to Open Neural Network Exchange format and then to TensorRT format. The Objective of these operations is to obtain a converted model, faster at inference time on GPU and that uses a smaller amount of memory.
The pipeline to transform a DETR [4] model in PyTorch to TensorRT is fundamental to be able to convert the second model that we are using to tackle object instance segmentation and tracking : Trackformer [5].
Trackformer [5] is particular convenient for our scopes since it uses DETR [4] with slight modifications: a particular choice of non-random queries. This queries have the task of encoding the appearance of the object that is being tracked.

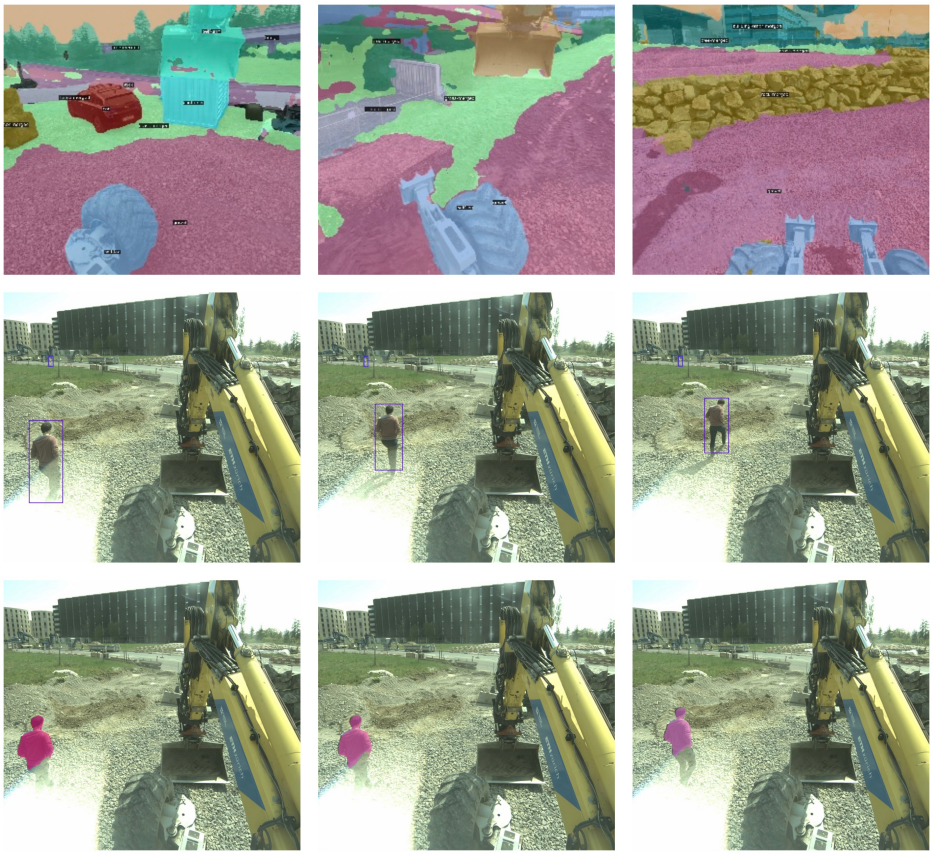

Figure 1.1: Three tasks (top-bottom): (A)panoptic segmentation (with a segmentation produced by DETR on construction site dataset), (B)object tracking, (C)instance level segmentation and tracking (with bounding boxes and segmentation mask produced on a video from RSL autonomous excavator cameras).

Chapter 2

Model speed enhancement

We start by enhancing the existing panoptic segmentation model and then we focus on object instance segmentation and tracking 1.1.C.
We will refer to DETR for panoptic segmentation as currently used panoptic segmentation model. This model is, as mentioned in introduction, an implementation of a DEtection TRansoformer [4].
The current panoptic segmentation model was firstly converted from PyTorch to Open Neural Network Exchange format and then to TensorRT format.
The Objective of these operations, as mentioned earlier, is to obtain a converted model, faster at inference time on GPU that would use a smaller amount of memory.
In the following we will show the starting PyTorch baseline, our methodology and for each framework a brief explanation of what it is, how we converted the current panoptic segmentation model, and finally how to perform inference with them.
Finally we will evaluate the converted model, using a dataset for panoptic segmentation in a construction site environment.

2.1 PyTorch DETR baseline for panoptic segmentation

The existing panoptic segmentation model is a DEtection TRansformer [4], implemented with PyTorch, where Facebook research [1] and Huggingface [2] offer some implementation of this model specifically for panoptic segmentation.
In particular we started converting from a DETR implemented similarity to the one offered by Facebook research 1.
The issues with using directly a transformer based model like DETR [4] is that it was particularly slow, preventing it from having an even wider usage for planning, detection, or tracking on our robot.
As we can see from the results on table 2.1 NVIDIA TITAN X GPU, Pytorch DETR had mainly two bottlenecks, inference time and post-processing time.
After studying community standard for similar conversion problems [3] [4] [5]
we understood that model inference time can be improved using a more efficient GPU computation at each layer. This can be achieved by converting the model and using established frameworks like ONNX and TensorRT.

[1] detectron2 : `https://github.com/facebookresearch/detectron2`
[2] huggingface DETR : `https://huggingface.co/facebook/detr-resnet-50-panoptic`
[3] Hugging face repo: `https://github.com/ELS-RD/transformer-deploy`
[4] Horace blog: `https://horace.io/brrr_intro.html`
[5] Tulloch blog: `https://tullo.ch/articles/pytorch-gpu-inference-performance`

Table 2.1: Inference results on NVIDIA TITAN X gpu

Inference components	units	time	time percentage
Preprocessing time	ms	5.00(26)	1%
Inference time	ms	384(43)	81%
Postprocessing	ms	80(89)	18%

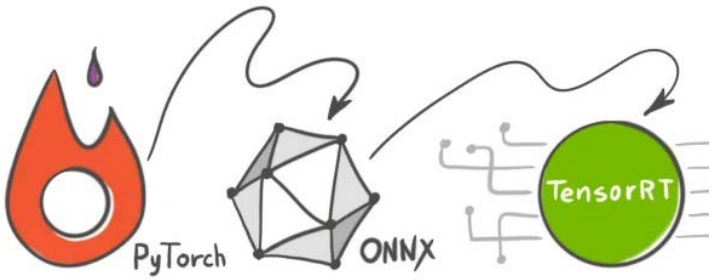

Figure 2.1: Suggested conversion pipeline[8].

2.2 Exporting the model: Method

The suggested conversion pipeline[6] as identified from literature involve exporting the PyTorch model to onnx format and then from onnx to TensorRT format, as represented in 2.1.
Another possible route is to directly convert from Pytorch to Tensorrt [7], for which we compormise the possibility of using onnx tools to change model structure graphically with GraphSurgeon.
We will now analyze single components of the conversion pipeline.

2.3 Using Onnx

We firstly went from a Pytorch model, typically saved as a state dictionary under the extension $< file - name > .pth$, to an Onnx model saved similarly under $< file - name > .onnx$ extension.
We use specifically Onnx since it is an open source standard framework to optimize machine learning inferencing. In particular Onnx offers different advantages:

1. Converting to onnx is easy because most of the layers are supported unlike in TensorRT

2. Using onnx-runtime alone, we can lower inference time and shorten the memory footprint of our models with respect to the same PyTorch model. This is evident looking at our results after conversion 2.2

3. We can use tools like GraphSurgeon that allow us to change the model structure without changing the layer's implementation

4. ONNX is a common intermediate layer. Indeed it is possible to use, Tensorflow instead of Pytorch and we would still be able to convert our saved checkpoint to onnx and then to TensorRT.

[6] OpenCV blog: `learnopenCV.com`
[7] Torch-TensorRT: `https://pytorch.org/TensorRT/`

2.4 Going from Pytorch to Onnx

To export a model from Pytorch to Onnx via tracing you should call the function

`torch.onnx.export()`

. This will execute the model, and record a trace of the operators that are used to compute the outputs. Because export runs the model and record a trace of each used operator, we need to provide an input tensor. We used a random tensor as input while being careful to match the right type and size requested by the model. Following this procedure, the input size will be fixed in the exported ONNX model. If the input size changes an out-of-memory error is thrown. Since we use Onnx for inference, we exported the model with an input of $batch - size = 1$.
This would be given as a parameter in

`torch.onnx.export()`

. For this command we used the following list of parameters.

1. model : Pytorch model in eval mode from torch.load(...)
2. model-input-size : size of the input as a list [batch-size, 3, 955, 800]
3. model-path : Path where to save the model
4. output-names : ["output"] name to be given to the output(s) tensors of onnx model, useful when debugging the model
5. opset-version : Onnx converter version, we used 13 as default, use the most recent one, since newer version support more types of network layers
6. do-constant-folding : Whether to execute constant folding for optimization
7. export-params : Whether to store the trained parameter weights inside the model file

Once the model $< file-name > .onnx$ is exported it can now be inspected visually with Netron and used for inference with OnnxRuntime. Netron[9] is a simple online viewer for neural networks. We then use OnnxRuntime to perform inference with our Onnx model.

2.4.1 Inference with Onnx

Performing inference with OnnxRuntime is convenient and can shorten time and memory footprint. To start predicting, for a generic model, you need to first initialize session options through

`rt.SessionOptions(...)`

Then start a new session with

`rt.InferenceSession(...)`

specifying model path, session options and providers i.e. CUDAExecutionProvider or CPUExecutionProvider. Finally, after running

`session.run(...)`

[9] Netron: `https://netron.app`

with the field $inputname = your-model-input$ the model will provide the desired outputs. It's important to notice that in order to have the same accuracy of the PyTorch implementation of the network, it should be fed as input tensor, an image of exactly the same size as the image given to the Torch model. If everything is done right, the output of your model should be the same as the Pytorch / Tensorflow model, since Onnx does not introduce any approximation (if the same weights' precision is kept i.e. FP32, FP16). The relative divergence across all layers output is negligible, in the order of 10^{-5}.
Many examples of deployed networks can be found on the suggested repository for onnxruntime [10]

2.5 Using TensorRT

TensorRT is a C++ library developed by NVIDIA for fast inference on NVIDIA GPUs. TensorRT is built on CUDA, NVIDIA's parallel computing platform and application interface. To achieve better performance TensorRT uses five types of **optimizations**: Precision calibration, Layer and tensor fusion, kernel auto-tuning, multi-stream execution and dynamic tensor memory, which are outlined in the following: [11]

1. **Weight and Activation Precision Calibration**: During training, parameters and activations are expressed in FP32 (Floating Point 32) precision. They can be converted in FP16 or INT8 precision. This optimization not only reduces latency but also gives a significant reduction in model size because FP32 weights get converted into FP16 or INT8 precision. It is important to notice that, as for Onnx, shrinking due to numerical overflow FP32 weights into FP16 doesn't affect the accuracy significantly, indeed clipping weights from FP32 to FP16 preserve the output and increase inference speed. Performing operation with lower precision is typically faster, which justifies model speed-up. But converting from FP32 to FP16 or to INT8 in TensorRT is not straightforward since not all layers can be converted with such precisions.

2. **Layers and Tensor Fusion**: While executing a network, layers and tensors fusion needs to be performed routinely. Therefore, TensorRT uses layer and tensor fusion to optimize the GPU memory and bandwidth by fusing nodes in a kernel vertically (between different layers) and horizontally (within the same layer), which reduces the overhead and the cost of reading and writing the tensor data for each layer. I.e. in the case of a convolutional network TensorRT recognizes all layers with similar input and filter size and combines them to form a single node.

3. **Kernel auto-tuning**: While optimizing models, there is some kernel-specific optimization that can be performed during the process. This selects the best layers, algorithms, and optimal batch size based on the target GPU platform. For example, there are multiple ways of performing convolution operations and TensorRT can select the best strategy automatically.

4. **Dynamic Tensor Memory**: TensorRT improves memory reuse by allocating memory to the tensor only for the duration of its usage. It helps in reducing the memory footprint and avoiding allocation overhead for fast and efficient execution.

[10] onnxruntime inference examples: `https://github.com/microsoft/onnxruntime-inference-examples`
[11] article on TensorRT: `https://medium.com/@abhaychaturvedi_72055/understanding-nvidias-tensorrt-for-deep-learning-model-optimization-dad3eb6b26d9`

5. **Multiple Stream Execution**: TensorRT is designed to process multiple input streams in parallel.

2.6 Going from Onnx to TensorRT

We can perform inference with TensorRT with two different modalities.

1. perform inference directly with the Onnx model letting TensorRT do all of its optimizations online
2. Perform inference with a serialized model from Onnx, where Optimizations like Layer and tensor fusion and Kernel auto-tuning are already being incorporated.

We select the second option to perform predictions and the results for inference time/memory reduction can be seen in table 2.2.
TensorRT gives the best results but it is not straightforward to implement indeed not all the layers are supported. For this reason, we modified several nodes of the original DETR.

1. We modified how Pytorch performs tensor padding: when calling the nested-tensor-from-tensor-list() function, Pytorch adds zeros iterating over the tensor, this is not supported by Onnx and TensorRT. We instead padded the tensor, adding a row or a column of zeros of the size of the tensor dimension to be padded.
2. We erased some other padding on the backbone of our model since they were using ND-shape tensors, that are not yet supported by TensorRT engine serializing tools [12]
3. TensorRT engine serializing tools do not support Gathering operations with the first input (the tensor to be reshaped) made of Boolean data. Therefore we cut with the help of Onnx Graphsurgeon, the Casting layer performing the conversion from binary to float numbers after the gathering layer, and we attached it before the gathering operation.

All layers that are used by our DETR model are made of unit components that are naturally supported by TensorRT. TensorRT gives the possibility to define a custom layer in our network, but to allow for a smooth conversion, you should write a tactic i.e. a function to allow TensorRT to serialize the layer in C++. The latter can be achieved by implementing the IPluginV2 and IPluginCreator classes using C++ for Caffe or using the python API calling the function

```
add-plugin-v2
```

that enables the addition of a plug-in node to a network[13].
To generate a serialized engine from the Onnx model, we use the **Trtexec** tool. Trtexec is a command line tool, that can be used to build engines, using different TensorRT features through command line arguments, and run inference. Trtexec also measures and reports execution time and can be used to understand performance and locate bottlenecks. For our scope we used the following conversion command :

[12] GitHub issue we raised regarding padding : `https://github.com/NVIDIA/TensorRT/issues/1882#issuecomment-1107438254`

[13] Official guide to define a custom layer with python API : `https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#add_custom_layer_python`

```
opt/tensorrt/bin/trtexec
--onnx=model.onnx
--saveEngine=model.plan
--buildOnly
--workspace=5000
```

Here, it is important to notice that...

1. We should allow the always maximum workspace possible, by specifying a high value of allocated GPU memory when converting. This should enable the tactics to convert the layer, such that the model is better optimized.

2. The workspace size set an upper bound on the memory, that the model use at inference time. Typically the model uses much less than the workspace.

3. Shrinking too much the workspace would eventually raise a conversion error, since there is a minimum memory requirement when serializing the model.

4. It is possible to specify various parameters if required for some special use case, the most important are: –batch to specify the batch size of the input, –int8 to clip the weights into int8 precision, –shapes to specify fixed input shape (do not specify it to use variable size inputs), –streams number of the stream to which the process is assigned

2.6.1 Inference with TensorRT

Inferencing with TensorRT is typically more complex than Onnx and in order to have a clear understanding of the steps involved, we report the macro actions and functions needed to obtain predictions.

1. Load and deserialize .plan model to obtain the CUDA engine.

2. Allocate memory buffers on GPU, to place the deserialized model, inputs, and outputs and attach the buffer to a Cuda stream. The latter mandatory step allows to use of multiple streams simultaneously.

3. Initialize execution context with

   ```
   cuda-engine.create-execution-context()
   ```

4. Use allocated memory to attach the actual model input. This can be achieved by sending the input to the host memory buffers, through similar instructions:

   ```
   inputs[0].host = your-input
   ```

   . It's important to specify the input as a contiguous packed array in memory, to avoid out-of-memory errors.

5. Run inference based on provided context, stream, and bindings.

6. Finally transfer predictions back from the GPU and return them.

2.7 Results

We evaluated all three models: Torch, Onnx, and TensorRT comparing inference time and memory usage on two different GPUs. As introducted before, the objective of this conversion was to obtain the maximum speedup possible.
With TensorRT we achieved such goals, and we gained a 2.56 factor of time advantage with respect to the Torch model. On GeForce RTX 3090, we reached 18hz for inference alone, see table 2.2. With Onnx-runtime we achieved a 1.52 factor of time advantage for inference, see table 2.2.
The following tables summarize the inference results on two different GPUs.
The numbers under brackets represent the uncertainty measurement $(u) = 1 * \sigma$ *following the latest conventions of the International System of Units*[14]

Table 2.2: Mean Inference time results for different models

Inference components	units	GeForce RTX 3090	GeForce GTX 1080
Torch model	s	0.1280(787)	0.3840(435)
Onnx model	s	0.0842(211)	0.252(012)
TensorRT model	s	0.053830(509)	0.150(005)

Preprocessing time scale as we switch to a better performing GPU, GeForce RTX 3090, instead postprocessing time variation is much less accentuated since it mostly uses CPU, 2.3. As we decrease the inference time, postprocessing became the new bottleneck of the pipeline, the following table 2.3 shows the result on GeForce RTX 3090.

Table 2.3: Mean Inference time on GeForce RTX 3090

Inference components	units	Mean Inference time(σ)	time percentage
Preprocessing	s	0.00696(644)	4%
Inference	s	0.05383(509)	36%
Postprocessing	s	0.08958(723)	60%

We tested different configurations when converting to Onnx 2.4. Specifically, the vanilla model corresponds to the Torch model but is directly converted to Onnx format without changing the model structure. The simplified model is the Onnx model, after using the Onnx graph surgeon to remove and change some layers, as reported previously, and after simplifying and folding the model. We also tested int8 quantization of the weights with Onnx and the Quantized model refers to this model. All this results use same measurement of uncertainty that is $\sigma = 0.509$s

Table 2.4: Mean Inference time on GeForce GTX 1080 for different onnx models

Inference components	units	Mean Inference time
Vanilla model	s	0.282
Simplified model	s	0.252
Quantized model	s	0.292

When using the same precision of the weights, i.e. FP32, no drop in accuracy is supposed to happen. Keeping the same dimension for the input and using the very

[14]Simple guide to uncertanty : `https://www.bipm.org/documents/20126/50065290/JCGM_GUM_6_2020.pdf/d4e77d99-3870-0908-ff37-c1b6a230a337n`

same preprocessing steps (those two are fundamental to observe no accuracy drop) we achieved the following results on the construction site dataset 2.5. *PQ refers to panoptic quality, SQ refers to segmentation quality, RQ refers to recognition quality, and N to the number of classes*

Table 2.5: Network accuracy for construction site dataset

Inference components	PQ	SQ	RQ	N
All	82.6	94.9	86.3	26
Things	66.8	83.8	79.5	8
Stuff	89.6	99.8	89.4	18

Lastly, we focused on the memory usage of our models. The considered models should in the future also be deployed on more lightweight edge-devices, an example could be AnyMal. The results, 2.6 show a diminished memory requirement to perform Inference using TensorRT and even more using Onnx. Our results are especially focused on the use of two different precision fp32 and best, where best refers to the optimal precision for each layer. When doing Inference for the first time, we require to load the model on GPU memory, this is the memory to start the engine. Then for new each image fed on the model a small part of the memory is mounted and dismounted, this is what we refer as memory to predict.

Table 2.6: Memory usage

Model	Config	Start the engine	predict
Torch model	fp32	7.4 GB	-
Onnx model	fp32	1.5 GB	227.8 Mb
TensorRT model	12gb workspace, fp32	4.6 GB	558.0 Mb
TensorRT model	4gb workspace, fp32	3.9 GB	54.5 Mb
TensorRT model	4gb workspace, best	3.9 GB	54.5 Mb

The variation observed when using the TensorRT model, is mainly due to workspace size. As explained above, when converting with

```
trtexec
```

is possible to specify the maximum workspace to be allowed during conversion.
In general, we should allow for the maximum workspace possible, since the tactics to convert layers from python work better when they have more memory to use. TensorRT typically uses less memory than the workspace allowed when performing inference, and maximum workspace puts an upper limit on the memory that could be used when Inferencing. As shown in the table we can diminish the memory footprint of our TensorRT model by restricting the maximum Workspace, at a cost of longer inference times. Using a maximum Workspace that exceeds the necessary memory to convert the model, in general, does not shorten further the inference time. For instance, our model requires almost 5 GB of workspace to be converted, using a wider Workspace of 12 GB is shown to not affect the model performances.

Chapter 3

Video object tracking

Perception is one of the fundamental blocks that any robot should have. To perceive the world in a meaningful way, we need to understand the objects we are seeing (panoptic segmentation), where the objects are going, through space and time (object tracking across frames), and finally what are the actual boundaries of the objects(object instance segmentation and panoptic segmentation). Position, type and boundaries of the objects are useful when performing grasping and locomotion. In the first part of this thesis, we developed a pipeline to speed up our models The following part of the thesis investigate how can we make use of video object tracking and segmentation models, typically computationally expensive.

3.1 Related work

Firstly we investigated the most recent literature on video object tracking/segmentation.
The faster approach is ByteTrack [6] which uses no learned features to perform tracking by regression, 3.1.
Among models that uses no-learned features and perform tracking by detection we recognize StrongSort [7], 3.1
Other End-to-end models that uses learned trackers are: TrackFormer [5], Mask2Former [8], Global tracking transformer [9], Vivit [10] and Mask R-CNN [11], see 3.1 .
Where the last two are models that do not represent current state-of-art. An updated list of the leaderboard of best performing trackers is available under multiple object tracking challenge benchmark [1]
TrackFormer [5] is a DETR [4] based learned approach, easily improvable with more training, more data, and using our speedup pipeline.
Mask2Former [8] is also a valid approach, originally proposed for images, can be extended to video sequences and despite being an offline algorithm it can be made online with some frame instances matching. Mask2Former [8] offers the possibility to train a DETR [4] model on images and use it for video scene segmentation.
After the literature study, we selected the last two approaches: Trackformer [5] and Mask2Former [8]. Typically those two models in a supervised training regime would require a labeled video dataset. Where the labels are either bounding boxes around the object of interest i.e. humans, rocks, etc., or a continuous segmentation of the scene. A labeled video dataset for a construction site environment is not available, and its generation should be avoided, so most of our efforts focused on doing **transfer learning** with pre-trained models on MOTS20 3.2 or adapting a DEtection TRansformer [4] trained on images of a construction site setup.

[1] MOT challenge benchmark `https://motchallenge.net/results/MOTS/`

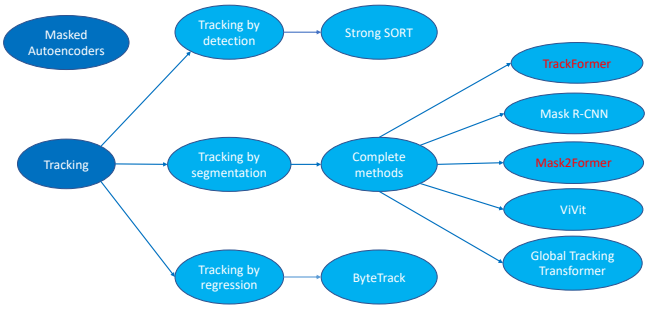

Figure 3.1: Usual classification of detection algorithms

3.2 Trackformer

We investigate how to enhance the perception capabilities of Menzi Muck, using object tracking based on a learned approach. We select Trackformer [5] since it achieves state-of-art results, in terms of speed and accuracy on the Multi-Object-Tracking task.
The innovations introduced by Trackformer [5] can be summarized as follows.

1. Tracking initialization: New objects appearing in the scene are detected by a fixed number of object output embeddings each initialized with a static and learned object encoding referred as object queries, 3.3. Each object query learns to predict objects with certain spatial properties, such as bounding box size and position.

2. Online Tracking: the track queries follow objects through a video sequence carrying by utilizing their identity information, 3.3. Computing self-attention over all queries allows for joint reasoning about the objects in a scene.

3. Track query re-identification: previously removed track queries are kept for a maximum number of frames. During this time track queries are considered to be inactive and do not contribute to the trajectory until a classification high score triggers a re-identification

Trackformer was select since it was implemented with DETR [4], where DETR[4] is the same model we used for panoptic segmentation of images on the construction site dataset and the one for which we developed our conversion pipeline to TensorRT.
Trackformer[5] was implemented in several different ways, in particular, 8 different models were available, 4 with deformable DETR and 4 with DETR. These 4 models were designed for, respectively, simultaneous object segmentation and tracking, only object tracking, object segmentation, and segmentation without tracking.

3.2.1 Method

We started by looking at the model capable of object segmentation and tracking. Specifically, we approached the problem using two kinds of pre-trained DETR, since it would be time expensive and unjustified to train DETR from scratch.
We have multiple possibilities to train our model, in order to track different objects, for instance, people, piles of sand, rocks, cars, etc. We have chosen people since they

Figure 3.2: Tracking people on MOTS20

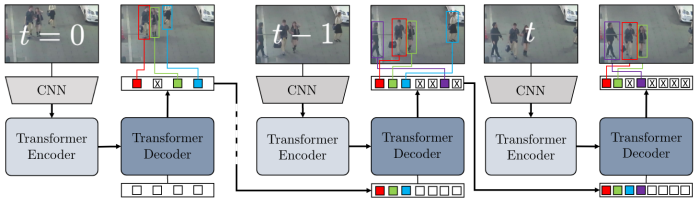

Figure 3.3: Representation of Trackformer [5] Network. The queries are high dimensional representation of the features around the object. These queries are fed in an autoregressive manner to the next step of tracking.

are the most important "objects" to be followed or avoided and for the availability of datasets for human tracking.
Different datasets are available in the domain of tracking of people: MOT17, MOT20, MOTS20, CrowdHumans, etc.
We trained using MOTS20 (multiple object tracking for segmentation dataset) since this dataset contains segmented object instances. It could be possible to train the same model on other datasets such as MOT17 and MOT20 with minor changes but without the possibility of predicting and computing losses for segmentation masks, but only for bounding boxes.
All these datasets are designed for people tracking. In a classical supervised learning setting, in order to track other objects of interest such as a pile of sand, cars, and rocks, it would be required a labeled dataset. Most recent approaches show that heavy hand labeling is not always required. Our objective is to prove that we can train an object detector like Trackformer without heavy hand labeling of images.

3.2.2 Training of Coco pre-trained DETR

Since Trackformer models were trained to start with weights of DETR pre-trained on the COCO dataset (a very large dataset for panoptic segmentation of images depicting animals and common scenarios), we started by loading the official COCO pre-trained weights of DETR from Facebook. The objective was to match the same training scheme of the Trackformer paper and reproduce the results.
But trivially using DETR trained on COCO doesn't work. To perform panoptic segmentation on the COCO dataset, indeed require a number of classes of 251 (that corresponds to the number of possible objects inside the latter dataset). Our objective was instead to track only people, therefore our model had to predict either the None class or the person class.
For this reason, we initialized the last layer of embedding weights that were using 251 output neurons, in order to use only 2 output neurons. This last layer went from a linear layer of size (hidden-layer-size)x251 to (hidden-layer-size)x2.

Therefore the only layer that has been trained from scratch, was this embedding layer.
In order to avoid overfitting to a small dataset like MOTS20, we adopted a special training regime. We used Cosine annealing and a warmup schedule for the learning rate. This means that we decreased the LR linearly starting with an LR = $5*10^{-4}$ for 100 steps to an LR = 10^{-4}. The LR was periodically increased starting with a large value that is slowly decreased in 200 steps to a minimum value before being increased rapidly again
When training we used the following hyper-parameters

Table 3.1: Hyper-parameters table

Parameters	Values
lr backbone	0.0001
lr drop	40
batch size	1
weight decay	0.0001
epochs	20
max norm clip parameter	0.1
position embedding	sine
dropout	0.1
number attention heads	8
number of queries	100
number of encoder/decoder layers	6
transformer dimension	256

We trained on MOTS20 dataset train split, 2262 frames of video representing people close to the camera walking by malls and Zürich streets. We tested the trained model on a video of 637 frames.

3.2.3 Training of excavator pre-trained DETR

Once we had been able to reproduce results, hoping to achieve better performances, we started using the trained DETR on the construction site dataset.
Precisely our DETR was firstly trained on the COCO dataset and then trained on the construction site dataset for panoptic segmentation, this was the model from which we started.
We then loaded the weights of the aforementioned model, and as before we initialized the embedding layer to match the class number. We tested our model with both resnet50 and a resnet101 as the backbone. Without changing any configuration we trained the model for 20 epochs.

Figure 3.4: From top to bottom : (A) DETR pretrained on construction site output and (B) DETR trained on COCO output. Artifacts manifests especially as false positives. It can be noticed the id merging (everyone has the same identity) of the second model output, where the color represents the identity

3.2.4 Results

We finally evaluate the two models both on the MOTS20 dataset, to compute error metrics, and on some unlabeled datasets of a typical construction setting, with some people walking on the stage.
The model is able to track the people in the scene without swapping the instance id, also after the person is occluded for several frames. Nevertheless, the network produces many artifacts.
The problems can be summarized in: track flickering and fragmentation, false positives of segmented sections without humans inside and false negatives of undetected human parts, track lost when the person is far, identity merging and identity swapping, in case of very large occlusions or challenging human poses, see 3.4.A.
We measured multiple objects tracking accuracy for both models. The MOTA score is 14.37%, below the current state of the art, which is around 80.3%, achieved by ByteTrack, one of the engineered techniques in our classification. Given the ease of use and training of this model, our score remains a worthful result.
To evaluate the IOU for bounding boxes and segmentation we measured **Average Precision** and **Average Recall**.
In particular Average Precision measures the percentage of the predictions that are correct. Calculated as the ratio between true positives over true positive + false positive. $AP = \frac{TP}{TP+FP}$
Average Recall measures the probability of ground truth objects being correctly detected. Average Recall ranges from 0 to 1 where a high recall score means that most ground truth objects were detected. E.g, $recall = 0.6$, implies that the model detects 60% of the objects correctly. $AR = \frac{TP}{TP+FN}$
The results were evaluated with variable **MaxDets** number. It corresponds to the maximum number of detection (cocoAPI style) ranging from 1 to 100.
The area parameter refers to the surface of the segmentation mask, used to compute IOU i.e. if IOU for the small parameter is high then the network is particularly good at segmenting tiny objects on stage, the same holds for medium and large areas.

Table 3.2: IoU metric for bounding boxes of DETR pretrained on COCO, evaluated on MOTS20 dataset

IoU metric bbox	area	maxDets	IOU
Average Precision (AP)	all	100	0.406
Average Precision (AP)	all	100	0.694
Average Precision (AP)	all	100	0.415
Average Precision (AP)	small	100	0.000
Average Precision (AP)	medium	100	0.145
Average Precision (AP)	large	100	0.620
Average Recall (AR)	all	1	0.070
Average Recall (AR)	all	10	0.408
Average Recall (AR)	all	100	0.550
Average Recall (AR)	small	100	0.000
Average Recall (AR)	medium	100	0.342
Average Recall(AR)	large	100	0.744

Table 3.3: IoU metric for segmentation of DETR pretrained on COCO, evaluated on MOTS20 dataset

IoU metric segmentation	area	maxDets	IOU
Average Precision (AP)	all	100	0.322
Average Precision (AP)	all	100	0.649
Average Precision (AP)	all	100	0.289
Average Precision (AP)	small	100	0.000
Average Precision (AP)	medium	100	0.113
Average Precision (AP)	large	100	0.492
Average Recall (AR)	all	1	0.058
Average Recall (AR)	all	10	0.347
Average Recall (AR)	all	100	0.423
Average Recall (AR)	small	100	0.000
Average Recall (AR)	medium	100	0.258
Average Recall (AR)	large	100	0.577

From these tables, especially 3.3 and 3.5, we can notice that the model pre-trained on the construction site dataset, 3.2 and evaluated on MOTS20 dataset performs slightly better than the model trained only on the Coco dataset, 3.4. The latter means that using a model pre-trained also on a different dataset for panoptic segmentation gives some advantages. This is confirmed with a visual inspection of the output of the two models, where our model pre-trained on the construction site dataset 3.4.A is capable of generating fewer artifacts and better tracking on similar scenes, 3.4.B .

Table 3.4: IoU metric for bounding boxes of DETR pretrained on Construction site dataset, evaluated on MOTS20 dataset

IoU metric bbox	area	maxDets	IOU
Average Precision (AP)	all	100	0.506
Average Precision (AP)	all	100	0.783
Average Precision (AP)	all	100	0.547
Average Precision (AP)	small	100	0.000
Average Precision (AP)	medium	100	0.264
Average Precision (AP)	large	100	0.699
Average Recall (AR)	all	1	0.074
Average Recall (AR)	all	10	0.501
Average Recall (AR)	all	100	0.653
Average Recall (AR)	small	100	0.002
Average Recall (AR)	medium	100	0.501
Average Recall(AR)	large	100	0.797

Table 3.5: IoU metric for segmentation of DETR pretrained on Construction site dataset, evaluated on MOTS20 dataset

IoU metric segmentation	area	maxDets	IOU
Average Precision (AP)	all	1	0.374
Average Precision (AP)	all	10	0.716
Average Precision (AP)	all	100	0.359
Average Precision (AP)	small	100	0.000
Average Precision (AP)	medium	100	0.174
Average Precision (AP)	large	100	0.533
Average Recall (AR)	all	1	0.061
Average Recall (AR)	all	10	0.399
Average Recall (AR)	all	100	0.484
Average Recall (AR)	small	100	0.002
Average Recall (AR)	medium	100	0.354
Average Recall (AR)	large	100	0.607

Chapter 4

Conclusions and future work

This work has touched on many aspects of a machine learning application. We went from model selection, training, performance validation to efficient and fast deployment.
we have looked into the entire pipeline to obtain an ML application usable in the real world and differently from the usual production cycle of a ML algorithm, we started bottom-up, obtaining a pipeline to efficient deployment of a DETR[4] and then train and validate a similar DETR model [4].
We created a framework to convert a model written in PyTorch to a serialized model in TensoRT and Onnx, validating results with our ready-to-use panoptic segmentation model.
Then we investigated the algorithms currently available. We looked at Trackformer[12], to see if it would adapt to our specifications and if it would be able to perform online instance level segmentation and tracking.
Finally, we selected a model and we proved that is possible to track potentially any object starting with a pretrained model for images, training with a small data set, without heavy hand labeling of videos.
Future work on the topic involve converting the object tracker in Onnx and TensoRT and measuring timing and memory footprint.
In future, it would be required to look more on training: more epochs or different training regimes, considering that the 20 epochs that we adopted are relatively small and given the fact that the order of magnitude of the number of the weight is 10^7.
Finally it would be valuable to construct a labeled data set by uploading a video on i.e. segment.ai, using Mask2Former to obtain the segmentation and test the generalization capabilities of this tracker for different objects.
To conclude, even though robotics perception is made of several blocks, in order to make able agents to reason about the surrounding environment is required a clear understanding of everyday objects characteristics (i.e. object position, velocity, shape and appearance). Therefore further investigation on the topic is required.

Bibliography

[1] A. Kirillov, K. He, R. B. Girshick, C. Rother, and P. Dollár, "Panoptic segmentation," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9396–9405, 2019.

[2] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2021.

[3] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9992–10 002, 2021.

[4] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," 2020.

[5] T. Meinhardt, A. Kirillov, L. Leal-Taixé, and C. Feichtenhofer, "Trackformer: Multi-object tracking with transformers," *ArXiv*, vol. abs/2101.02702, 2021.

[6] Y. Zhang, P. Sun, Y. Jiang, D. Yu, Z. Yuan, P. Luo, W. Liu, and X. Wang, "Bytetrack: Multi-object tracking by associating every detection box," *ArXiv*, vol. abs/2110.06864, 2021.

[7] Y. Du, Y. Song, B. Yang, and Y. Zhao, "Strongsort: Make deepsort great again," *ArXiv*, vol. abs/2202.13514, 2022.

[8] B. Cheng, A. Choudhuri, I. Misra, A. Kirillov, R. Girdhar, and A. G. Schwing, "Mask2former for video instance segmentation," *ArXiv*, vol. abs/2112.10764, 2021.

[9] X. Zhou, T. Yin, V. Koltun, and P. Krähenbühl, "Global tracking transformers," *ArXiv*, vol. abs/2203.13250, 2022.

[10] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lucic, and C. Schmid, "Vivit: A video vision transformer," *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6816–6826, 2021.

[11] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," 2017, cite arxiv:1703.06870Comment: open source; appendix on more results.

[12] C. Feichtenhofer, H. Fan, Y. Li, and K. He, "Masked autoencoders as spatiotemporal learners," *ArXiv*, vol. abs/2205.09113, 2022.

[13] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," *ArXiv*, vol. abs/2005.12872, 2020.

[14] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, "Deformable detr: Deformable transformers for end-to-end object detection," *ArXiv*, vol. abs/2010.04159, 2021.

[2] [10] [13] [8] [7] [12] [1] [3] [5] [14] [6] [4]