Deep Learning Final Project

Course: Deep Learning indenfor det sundhedstekniske domæne

Date Submitted: 12/24/2025

Submitted by: Silas Jul Kierstein

Student ID: 201550608

Institution: University of southern denmark (SDU)

Table of contents

1. Introduction

In this project I implemented a state-of-the-art RF-DETR-Large architecture and used it on a children's playbook to solve a difficult object-detection task. I managed hardware constraints of a 24GB VRAM GPU by using Gradient Checkpointing and Mixed Precision (AMP), achieving a high-precision. This report details the project which is for the Deep Learning course exam. The goal of this project was to apply theoretical concepts learned throughout the course to a chosen problem that we hadn't covered or previously done. My personal goal was to challenge myself and see what I could achieve with the knowledge and skills I had acquired throughout this semester. I had a blast making this project!

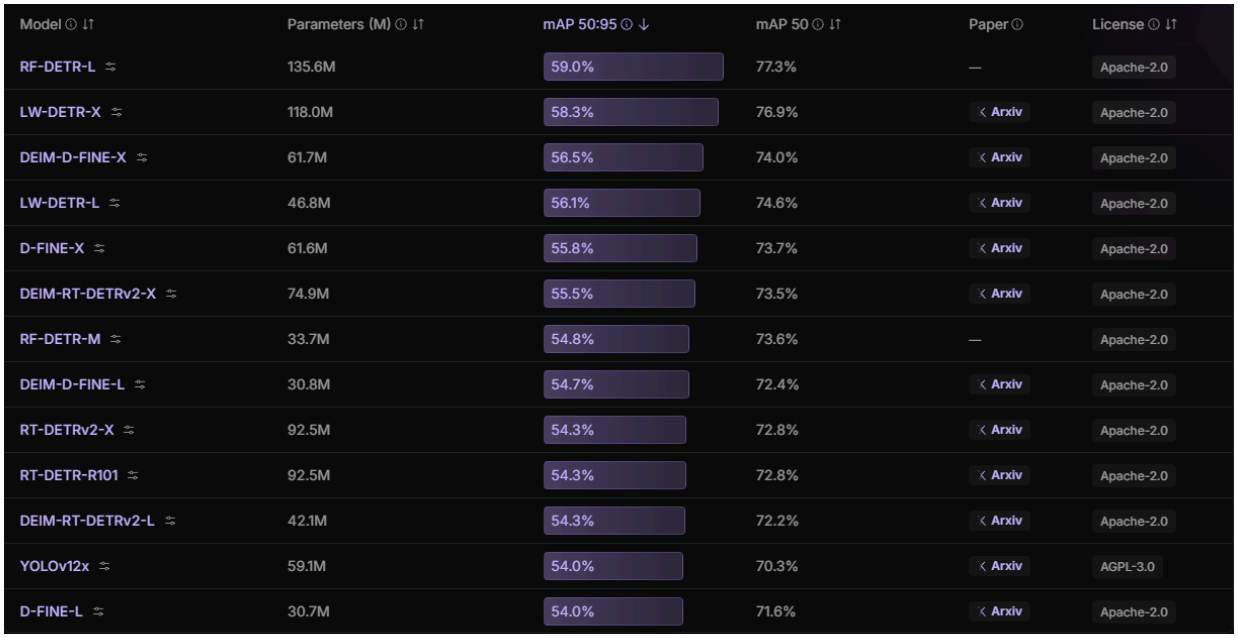
2. Project Overview

2.1 Problem Statement

"Wheres Wally" is an infamous children's playbook, where the goal is to locate tiny characters in a big clutter of drawings and IT IS HARD! I thought it would be really fun to see if I could use object-detection to locate the characters that you have to search for in this book. This task is quite difficult for multiple reasons. The pictures are very big, I only used images from one book so my dataset is small and pictures vary a lot in levels of detail, noise and color. The characters also vary in the way they are drawn. As a human looking at these images, it truly feels like looking for a needle in a haystack, but how will a machine do?

2.2 Transfer-learning (RF-DETR)

This is a difficult task and personally I am much more interested in transfer-learning simply because it feels like having super-powers, when you are able to leverage other people's research to achieve incredible results. I was also curious how the best performing models would handle a task like finding Wally. I researched the best performing architectures and found that the RF-DETR models where the current best performers when looking at precision (COCO):



Model	Parameters (M)	mAP 50:95	mAP 50	Paper	License
RF-DETR-L	135.6M	59.0%	77.3%	—	Apache-2.0
LW-DETR-X	118.0M	58.3%	76.9%	Arxiv	Apache-2.0
DEIM-D-FINE-X	61.7M	56.5%	74.0%	Arxiv	Apache-2.0
LW-DETR-L	46.8M	56.1%	74.6%	Arxiv	Apache-2.0
D-FINE-X	61.6M	55.8%	73.7%	Arxiv	Apache-2.0
DEIM-RT-DETRv2-X	74.9M	55.5%	73.5%	Arxiv	Apache-2.0
RF-DETR-M	33.7M	54.8%	73.6%	—	Apache-2.0
DEIM-D-FINE-L	30.8M	54.7%	72.4%	Arxiv	Apache-2.0
RT-DETRv2-X	92.5M	54.3%	72.8%	Arxiv	Apache-2.0
RT-DETR-R101	92.5M	54.3%	72.8%	Arxiv	Apache-2.0
DEIM-RT-DETRv2-L	42.1M	54.3%	72.2%	Arxiv	Apache-2.0
YOLOv12x	59.1M	54.0%	70.3%	Arxiv	AGPL-3.0
D-FINE-L	30.7M	54.0%	71.6%	Arxiv	Apache-2.0

Figure 1: Screenshot of a table showing a leaderboard of best performing object-detection models [https://leaderboard.roboflow.com/]

The RF-DETR is the current leader in precision scoring 59.0% (mAP 50:95) and 77.3% (mAP 50). The metric mAP 50:95 being the “Mean Average Precision at IoU thresholds from 0.5 to 0.95 (primary COCO metric)”. This leaderboard is also made by the developers of the top performing RF-DETR model, so it might be biased.

RF-DETR differs from YOLO as it is a Transformer-based model whilst YOLO is a CNN model. YOLO uses scanning looking at only a small window and determines what class it belongs to, whilst RF-DETR looks at the whole image together by comparing each patch to other patches to see if they relate. YOLO will draw lots of bounding-boxes and use NMS (Non-Maximum Suppression) to delete “duplicates”. This means it might accidentally delete the wrong Wally mistaking him for the tiny guy right beside him. RF-DETR uses Object Queries (usually 100-300). Each query can only have one answer/value, meaning it will not have any duplicates. This makes RF-DETR better for cases like “Wheres Wally”, as there are a bunch of tiny characters closely-packed where YOLO might struggle. RF-DETR is built on DINOv2 from Meta/Facebook trained on the LVD-142M dataset consisting of 142 million images. It was trained using self-supervision training, meaning that it had to distinguish between objects, textures and details by itself. Using these pretrained weights is a big advantage when trying to identify Wally as it already knows most patterns.

2.3 Dataset

I found a free PDF download for one of his books which seemed to be very high quality. From this PDF I extracted 12/16 images as the rest were not task related and saved them. For each page the task is to find a total of 4 characters:

Figure 2: Showing the characters: Wally, Wenda, Wizard Whitebeard and Odwald

The images converted from the PDF “Wheres Wally” (1st book in the series) are large 2480×1628 images. For me to train a model I can’t simply use the images as is, because it would just shrink them down to fit the model making it lose all its detail. This would compress Wally to some noisy looking pixels, which the model can’t use to generalize. RF-DETR models use a much smaller input size than the raw large images. This is good because it takes more computational power to process larger images. Time, math and memory are the biggest constraints regarding image sizes. The amount of pixels in an image does not just double when the size is doubled it quadruples:

- **Standard (640×640):** ~0.4 Million pixels.
- **High Res (1280×1280):** ~1.6 Million pixels.
- **Ultra Res (4K):** ~8.3 Million pixels.

To prevent my image from shrinking and losing all its detail I used a “slicing” technique. I sliced each image into lots of smaller ones using the slice_coco method from the SAHI library. This also automatically rewrites the coco.json annotations.

For creating the annotations I used Roboflow to mark the bounding boxes for each character in my dataset. Roboflow makes creating datasets and drawing boxes very easy and fast. This meant that I had to find all four characters in each page, which was a pain,

so I cheated and found the solutions online 🙂. Here is what the annotations look like in Roboflow:

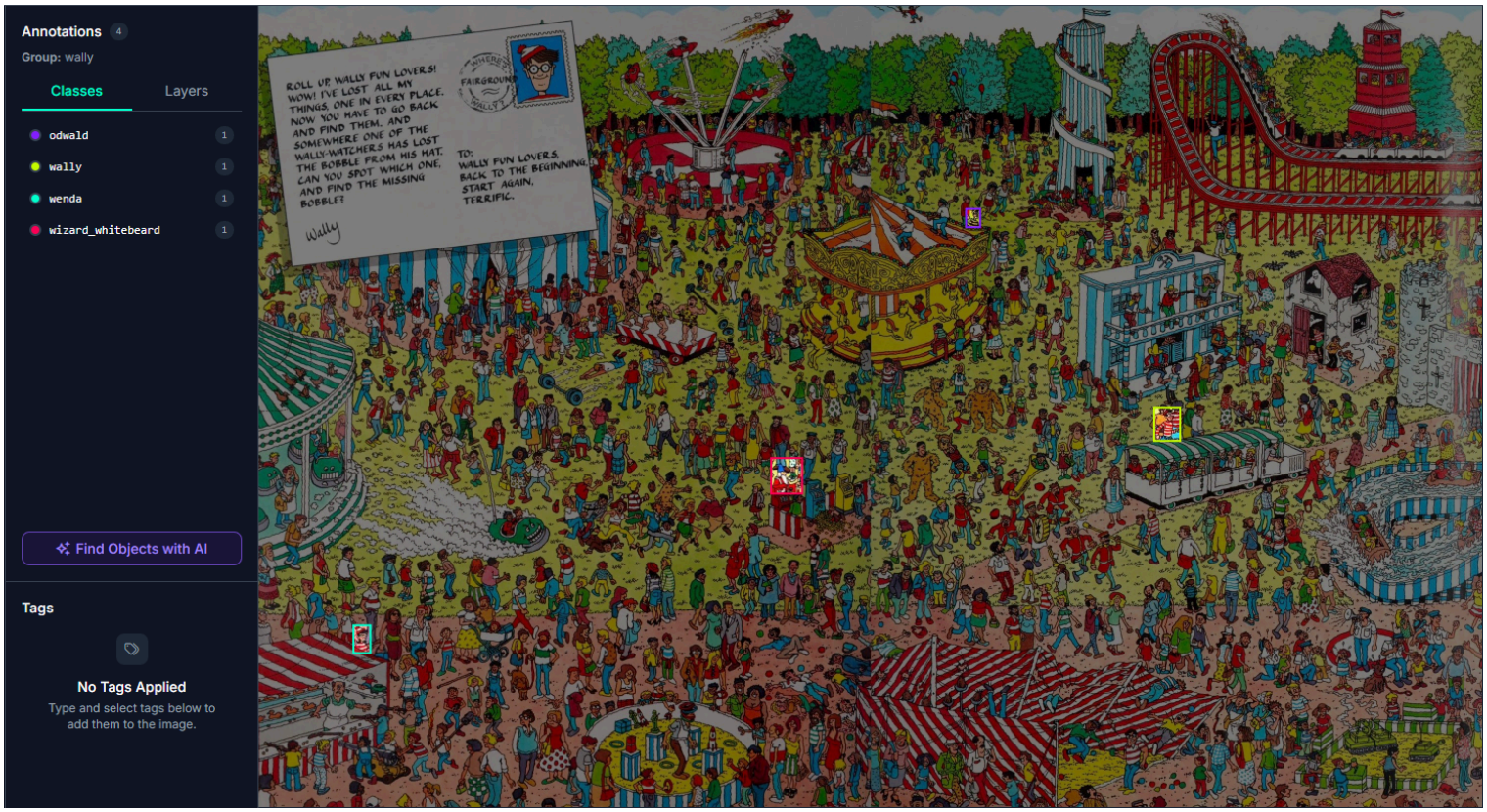


Figure 3: Image of annotating bounding boxes in Roboflow

Since I only had 12 images, I used data augmentation to create more training data. In my dataset split I used 8 images for training, 2 for validation and 2 for testing. I augmented the 8 images to 80 images, using varying brightness, noise and horizontal flipping. After having sliced those 80 images, using the coco_slice method, the dataset became filled with empty background images that didn't include the characters I wanted to classify. To negate the model becoming shy and learn that the characters don't exist I balanced the dataset by removing most of the background images. I found online (StackOverflow) that the optimal ratio was anywhere from 0-10%. Being in this range should teach the model what the background looks like without becoming "shy", failing to classify the characters. A screenshot of the output of my slicing script can be seen in appendix A. After slicing/pruning most of the background images, the dataset consisted of 563 training images and 13 validation images (no data augmentation for validation data).

3. Methodology

3.1 Model

RF-DETR comes in different sizes. The large model has 135.6m parameters, whilst the medium version has 33.7M. I started training with RF-DETR-M as this one is faster to train and experiment with so I don't waste my precious google-colab credits.

3.2 Training Parameters

The model was trained using the following hyperparameters:

RF-DETR-M

Parameter	Value
Optimization Algorithm	AdamW (default) [1]
Learning Rate	lr-4 (0.0001)
Loss Function	cross-entropy for classification and a combination of L1 and GIoU losses for bounding box regression (default) [1]
Batch Size	8
Gradient Accumulation Steps	2
Number of Epochs	10

The large model was trained with a couple of tweaks to allow me to train it on the same GPU and save credits:

RF-DETR-L

Parameter	Value
Optimization Algorithm	AdamW (default) [1]

Parameter	Value
Learning Rate	lr-4 (0.0001)
Loss Function	cross-entropy for classification and a combination of L1 and GIoU losses for bounding box regression (default) [1]
Batch Size	4
Gradient Accumulation Steps	4
Number of Epochs	100
Patience	10
AMP	True
Gradient Checkpointing	True

3.3 Implementation Details

Transfer-learning a highly documented model like RF-DETR is very easy as there are tons of YouTube, docs and notebooks on the internet to guide you. For training, evaluating and making predictions I used a notebook found on their github as inspiration [appendix C].

4. Results and Evaluation

Both of the models trained quickly in minutes with a standard learning rate. This is because my dataset is so small. It's a tradeoff between speed and precision. In training checkpoints are saved where it peaks in performance. For the medium model it went from understanding nothing in epoch 0 with an mAP@50 of 0.003 to 0.814 after just 4 epochs understanding and nailing most of the patterns. Looking at the loss_bbox and loss_ce (classification error) the loss_bbox drops very fast whilst the loss_ce took a little longer. This means that it was able to find the characters almost immediately, but took a little

longer before classifying the characters correctly. Here is a plot training metrics for the medium model:

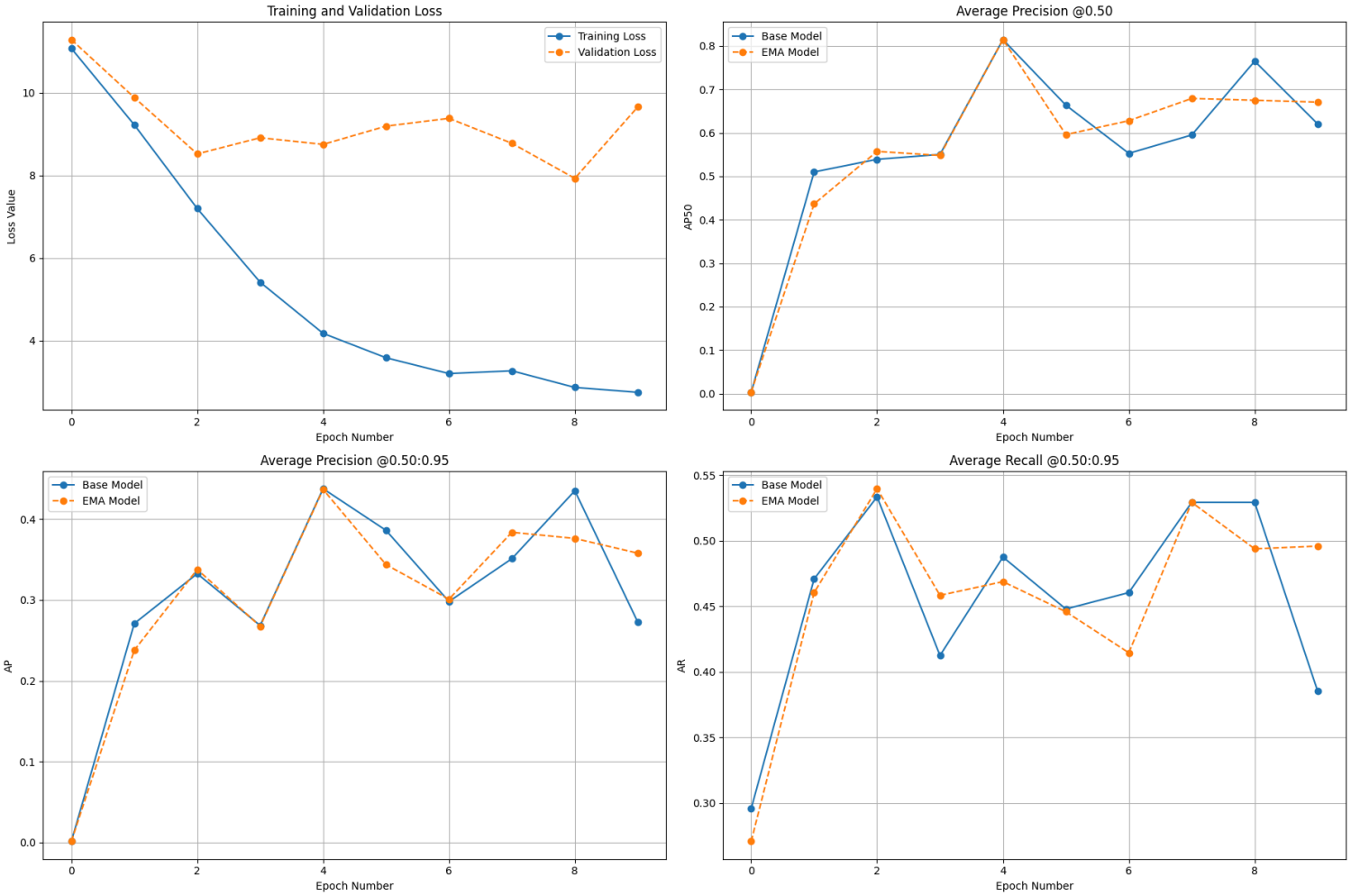


Figure 4: metrics plot from training the medium model

The large model seemed to learn much faster. Even with a patience of 10 epochs it maxed out its performance after epoch 7. Going from an mAP@50 of 0.735 to 0.843. The same metrics for training the large model can be seen here:

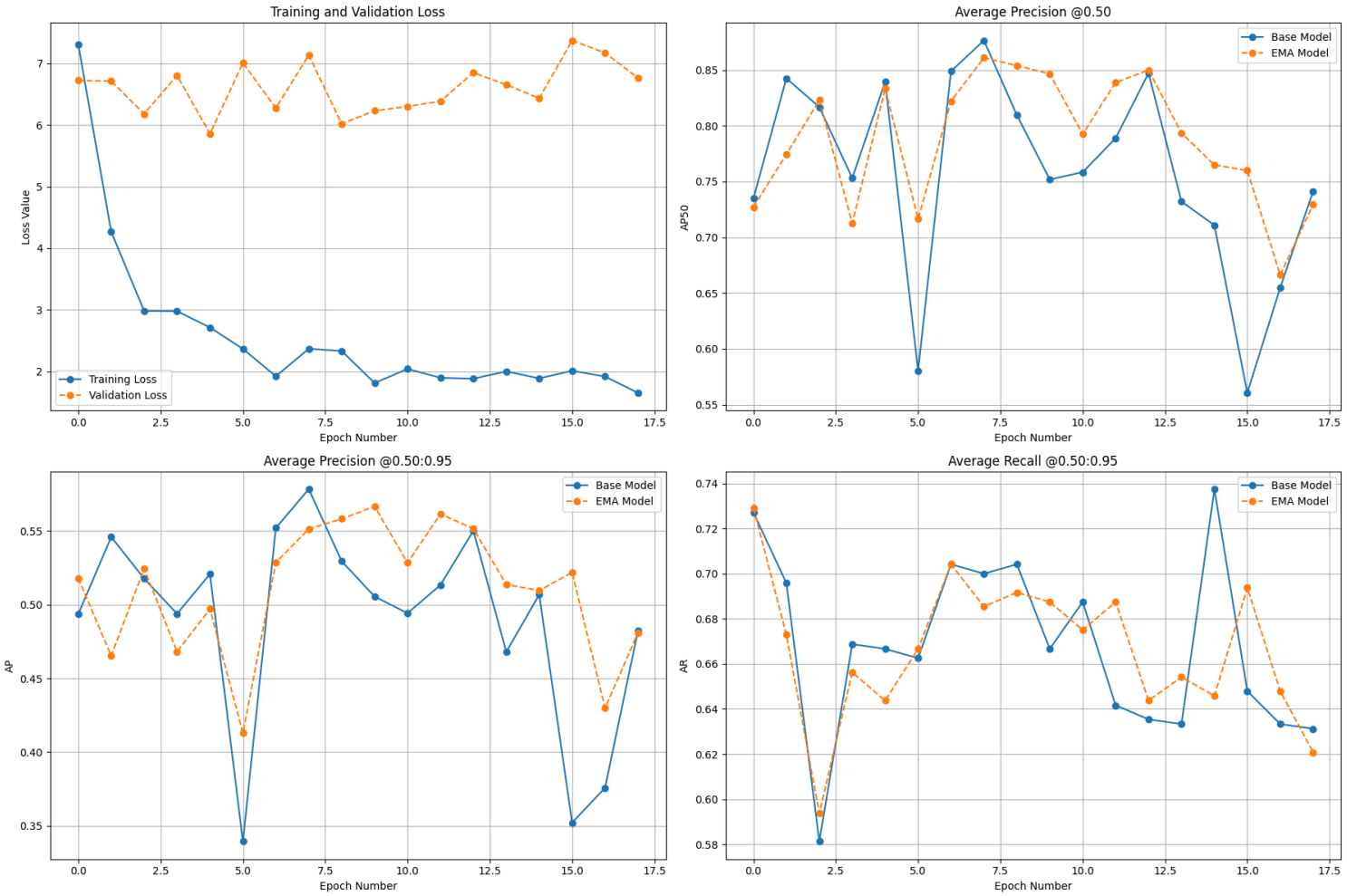


Figure 5: metrics plot from training the largemodel

4.1 Performance Metrics

Performance metrics were also created for each of the models when they peaked in performance. I acquired some very interesting insights when looking at precision, mAP and recall for the characters/classes separately and together. The full test results can be seen in appendix B.

Medium model

For the medium model the most noteworthy conclusions from the metrics were:

- **In validation the average precision was 1 (validation):** It is very good at predicting the right class and did it correctly 100% of the time on validation data.

Unfortunately the recall was only 0.5 on validation so it is a little “shy”, but when it guesses it is always correct.

- **Wenda was most precisely recognized (validation):** She has a precision of 0.74 mAP@50:95, she is also the character which is drawn mostly the same way and doesn't change poses much.
- **Wizard was most reliable (test):** The wizard had mAP@50 of 1.0 and mAP@95 of 0.67 which was the highest for all of the characters in the test data.
- **Odwald was the hardest class to identify (test):** Odwald had a precision of 0.0 but a recall of 1.0 and a mAP@50 of 0.03. This means it just fires in a bunch of random places because he is so similar to the other characters and background that it fails to distinguish between them. This would explain why it still has a little bit of mAP as it likely just fired in a bunch of places and randomly hit by a little.

Large model

Switching from a medium to a large model gave better results in some ways. Looking at validation most metrics moved in the right direction, but in the test it struggled in some aspects more than the medium model. Some key insights after having trained the large model and testing it were:

- **Massive performance boost (validation):** Recall stayed at 0.5 for all, but precision remained 1.0 and both mAP@50 and @50:95 went up, which is great. This shows that the large model is more accurate at drawing bounding boxes precisely.
- **Worse at drawing Wally (test):** Even though it drew Wally better in validation it got worse on test data. This is sad because he is the main character. mAP@50 got cut in half and @50:95 went down by ~0.8 too.
- **Improved at finding Odwald (test):** In both test and validation data the large model was better at finding Odwald, the hard class. Recall went down so it's no longer firing in a bunch of random places. Precision went up to 0.16. The mAP@50 also went up to 0.1.
- **More precise on average (test):** Whilst the medium model had an average precision of 0.4 on test data, the large model got 0.79. The recall decreased to 0.33. This is good as it recognizes the complex pattern of very well hidden characters, which the medium model didn't.

4.2 Predictions

I imported the best checkpoints for each of the models so I could load and run it on some of the original images locally and see how they do in action. I created a class for loading the modal and a predict method that takes an image-path as an argument. Here are some images predicted:

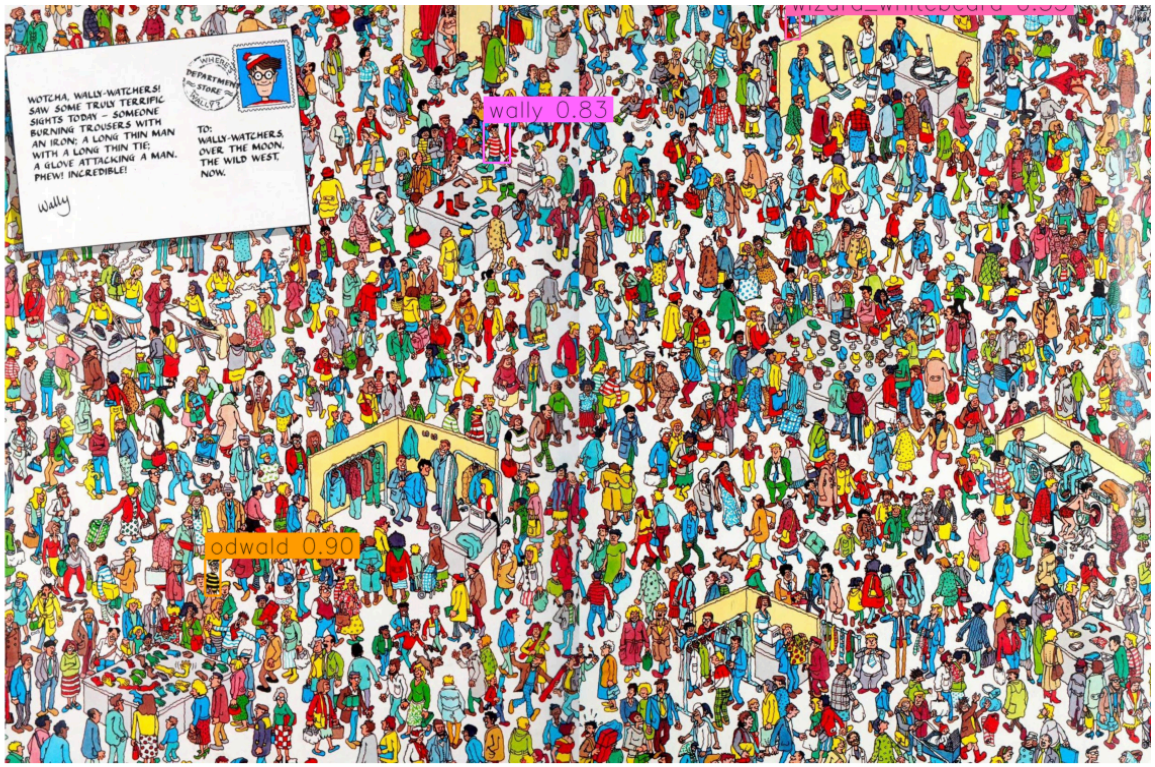


Figure 6: Prediction of a validation image using the medium model checkpoint

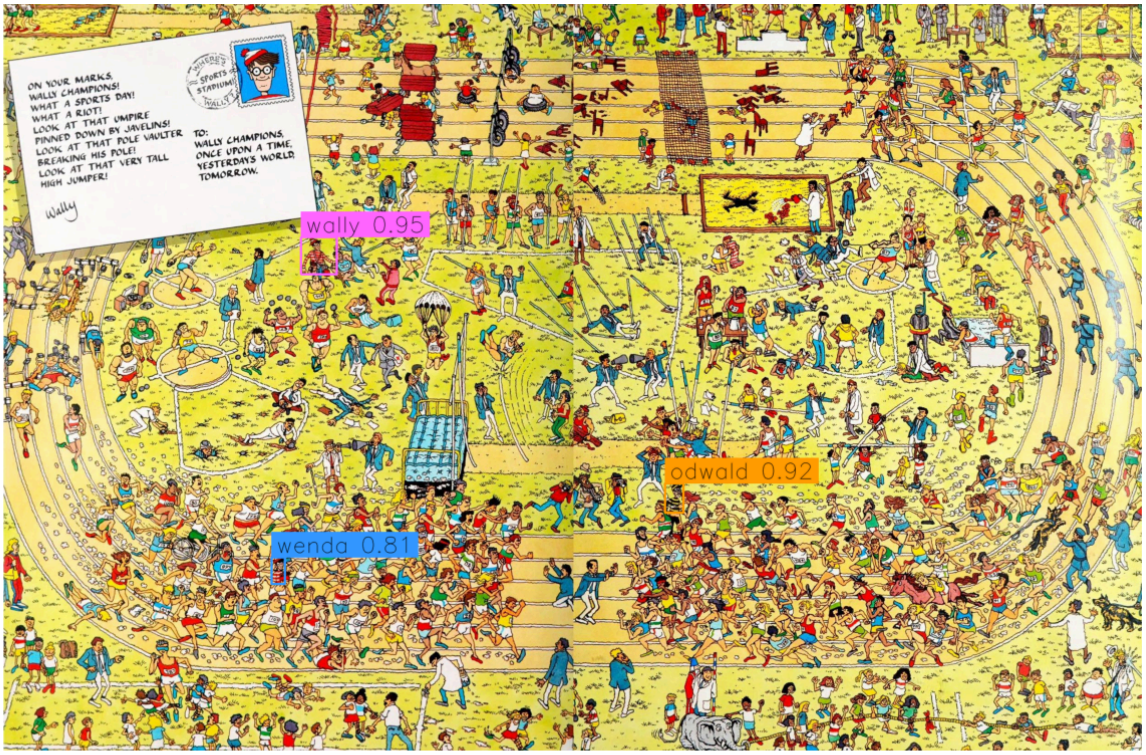


Figure 7: Prediction of a validation image using the large model checkpoint

Plotting an image side by side with the actual bounding boxes is a great way to visually test the model. Here is an image of a medium model prediction on the right and the correct annotations on the left for a test image:

Figure 8: Image of a prediction using the medium model on the right and the true annotations on the left

5. Discussion and Conclusion

The deep learning model achieved a lot better results than I had expected, with the complexity of the task and the small dataset I had gathered. The medium model achieved an average mAP@50 of 0.558 on the test data and the large one scored 0.428. Looking at those numbers only makes it look like the medium model is better, but looking closer at precision and recall, those numbers favor the large model. Using prediction from the checkpoints proves that it is usually able to correctly identify 2 characters per image.

This project shows how effective transfer learning is and how you can achieve impressive results from minimal training. Since I had a small dataset I chose to also show images from validation predictions (which is wrong because you want to avoid bias). The model did correctly identify characters in both test images, but these images were harder for the model to predict because of the new colors and some of the characters being hidden behind bars (new ways of hiding characters), making it very challenging for both models.

In the past I have used my own gpu because it was powerful enough, but this time I was resource limited and I tried using Google Colab and purchased some credits. Having access to a powerful GPU with 24GB VRAM was a blast and I did not expect it to finish training so quickly. This is a great benefit of a small dataset and shows how quality > quantity in a dataset can save you money. At first training the large model was a struggle, but I learned some neat tricks like using AMP and Gradient Checkpoints to reduce VRAM usage by 3-6GB making it go from almost maxing out to training smoothly on the large model. This can be seen in Appendix D.

5.1 Future work

If I had more time to work on this project I know lots of ways to improve the model. Given more time I could train the greatest Wheres-Wally model the world has ever seen! Next steps I considered to improve the model where:

1. **More training data:** There are 6 or 7 books in total and increasing the data would make it able to generalize better.
2. **Uncompressed data:** I found out that using Roboflow actually compressed my images by a little without telling me. I noticed this when I predicted an original

image (uncompressed 2480×1628) beside a test image (compressed 2048×1344) and I saw the original image had another correct character identified. This is quite a bit of detail that it is missing. This might explain why it fails to identify harder characters like Odwald which only has his small head sticking out at times.

3. **Optimize lr and batch-sizes:** The first first intuition since it finished training so quickly was that reducing the learning rate and playing with batch sizes as it is very easy to tweak these hyperparameters and they can have a big impact on results.

This project successfully demonstrates how you can use any dataset and use transfer-learning on the best performing models to solve most tasks. Transfer learning is a powerful tool and makes single developers like myself able to utilize incredible models and get incredible results, even when facing a difficult challenge and bottlenecked by a small dataset.

Appendix: References

Here you can see all of my external resources and tools used for this project.

Sources

[1] Report on RF-DETR by Ranjan SapkotaRahul Harsha CheppallyAjay ShardaManoj Karkee, titled: “RF-DETR Object Detection vs YOLOv12 : A Study of Transformer-based and CNN-based Architectures for Single-Class and Multi-Class Greenfruit Detection in Complex Orchard Environments Under Label Ambiguity
” - https://arxiv.org/html/2504.13099v1

Appendix A - code screenshots

This screenshot shows running and balancing my dataset to make it fit the model.

```
PS C:\Users\silas\Documents\Projects\automations\wheres-wally> uv run .\main.py build-dataset
Running slice_coco_dataset() ...
Loading coco annotations: 100%|
100%|
Loading coco annotations: 100%|
100%|
Loading coco annotations: 100%|
100%|
Renaming data/train_sliced\_annotations_coco.json -> data/train_sliced/_annotations.coco.json
Updated data/train_sliced/_annotations.coco.json: copied info, licenses.
Renaming data/valid_sliced\_annotations_coco.json -> data/valid_sliced/_annotations.coco.json
Updated data/valid_sliced/_annotations.coco.json: copied info, licenses.
Renaming data/test_sliced\_annotations_coco.json -> data/test_sliced/_annotations.coco.json
Updated data/test_sliced/_annotations.coco.json: copied info, licenses.
Running balance_dataset() ...
Processing data/train_sliced...
  Positives: 507 | Backgrounds: 453 -> Target: 56
  Deleting 397 excess background images...
  Successfully pruned data/train_sliced. Final total: 563 images.
Processing data/valid_sliced...
  Positives: 12 | Backgrounds: 12 -> Target: 1
  Deleting 11 excess background images...
  Successfully pruned data/valid_sliced. Final total: 13 images.
Processing data/test_sliced...
  Positives: 13 | Backgrounds: 11 -> Target: 1
  Deleting 10 excess background images...
  Successfully pruned data/test_sliced. Final total: 14 images.
Dataset build completed.
```

Appendix B - model result snippets

RF-DETR-Medium


```
{
    "class_map": {
        "valid": [
            {
                "class": "odwald",
                "map@50:95": 0.24334433443344333,
                "map@50": 0.504950495049505,
                "precision": 1.0,
                "recall": 0.5
            },
            {
                "class": "wally",
                "map@50:95": 0.13309804166536532,
                "map@50": 0.6649592404035356,
                "precision": 1.0,
                "recall": 0.5
            },
            {
                "class": "wenda",
                "map@50:95": 0.2595684793565756,
                "map@50": 0.8073296340623078,
                "precision": 1.0,
```

```
            "recall": 0.5
        },
        {
            "class": "wizard_whitebeard",
            "map@50:95": 0.45445544554455436,
            "map@50": 0.5049504950495048,
            "precision": 0.9999999999999998,
            "recall": 0.5
        },
        {
            "class": "all",
            "map@50:95": 0.27261657524998467,
            "map@50": 0.6205474661412133,
            "precision": 1.0,
            "recall": 0.5
        }
    ],
    "test": [
        {
            "class": "odwald",
            "map@50:95": 0.008899217896066132,
            "map@50": 0.03099055886296025,
            "precision": 0.0,
            "recall": 1.0
        },
        {
            "class": "wally",
            "map@50:95": 0.46019801980198016,
            "map@50": 0.865346534653465,
            "precision": 0.6,
            "recall": 1.0
        },
        {
            "class": "wenda",
            "map@50:95": 0.20278017646127589,
            "map@50": 0.3383590037942041,
            "precision": 0.0,
            "recall": 1.0
        },
```

```
            {
                "class": "wizard_whitebeard",
                "map@50:95": 0.6714521452145215,
                "map@50": 1.0,
                "precision": 1.0,
                "recall": 1.0
            },
            {
                "class": "all",
                "map@50:95": 0.3358323898434609,
                "map@50": 0.5586740243276574,
                "precision": 0.4,
                "recall": 1.0
            }
        ]
    },
    "map": 0.6205474661412133,
    "precision": 1.0,
    "recall": 0.5
}
```


RF-DETR-large


```
{
    "class_map": {
        "valid": [
            {
                "class": "odwald",
                "map@50:95": 0.47701395139513947,
                "map@50": 0.5634563456345635,
                "precision": 1.0,
                "recall": 0.5
            },
            {
                "class": "wally",
                "map@50:95": 0.22019801980198014,
                "map@50": 0.865346534653465,
                "precision": 1.0,
```

```
            "recall": 0.5
        },
        {
            "class": "wenda",
            "map@50:95": 0.7384158415841584,
            "map@50": 0.9326732673267323,
            "precision": 1.0,
            "recall": 0.5
        },
        {
            "class": "wizard_whitebeard",
            "map@50:95": 0.49405940594059405,
            "map@50": 0.603960396039604,
            "precision": 0.9999999999999998,
            "recall": 0.5
        },
        {
            "class": "all",
            "map@50:95": 0.4824218046804681,
            "map@50": 0.7413591359135914,
            "precision": 1.0,
            "recall": 0.5
        }
    ],
    "test": [
        {
            "class": "odwald",
            "map@50:95": 0.08372213026165884,
            "map@50": 0.10465266282707354,
            "precision": 0.16666666666666666,
            "recall": 0.33
        },
        {
            "class": "wally",
            "map@50:95": 0.3809355425338452,
            "map@50": 0.40139422105475864,
            "precision": 0.9999999999999998,
            "recall": 0.33
        },
```

```
            {
                "class": "wenda",
                "map@50:95": 0.24185724789964338,
                "map@50": 0.35157821828159525,
                "precision": 1.0,
                "recall": 0.33
            },
            {
                "class": "wizard_whitebeard",
                "map@50:95": 0.624068835454974,
                "map@50": 0.8556105610561057,
                "precision": 1.0,
                "recall": 0.33
            },
            {
                "class": "all",
                "map@50:95": 0.3326459390375304,
                "map@50": 0.42830891580488323,
                "precision": 0.7916666666666666,
                "recall": 0.33
            }
        ]
    },
    "map": 0.7413591359135914,
    "precision": 1.0,
    "recall": 0.5
}
```

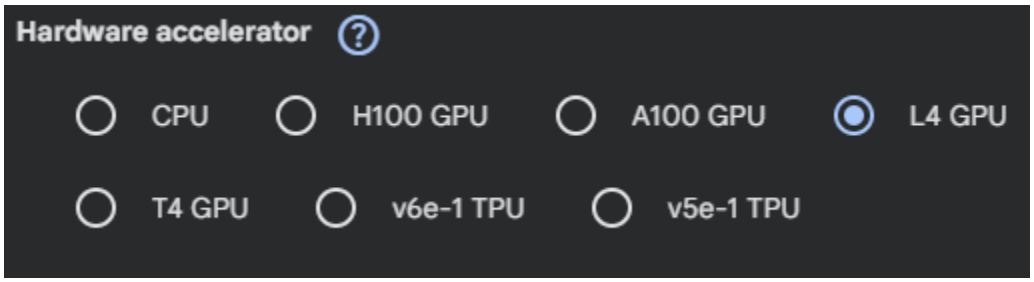

Appendix C - other

- Leaderboard for object-detection models on roboflow - https://leaderboard.roboflow.com/
- Training documentation roboflow - https://docs.roboflow.com/train/train
- RF-DETR notebook - https://github.com/roboflow/notebooks
- My github for testing code, slicing data and creating predictions - https://github.com/silasjul/wheres-wally

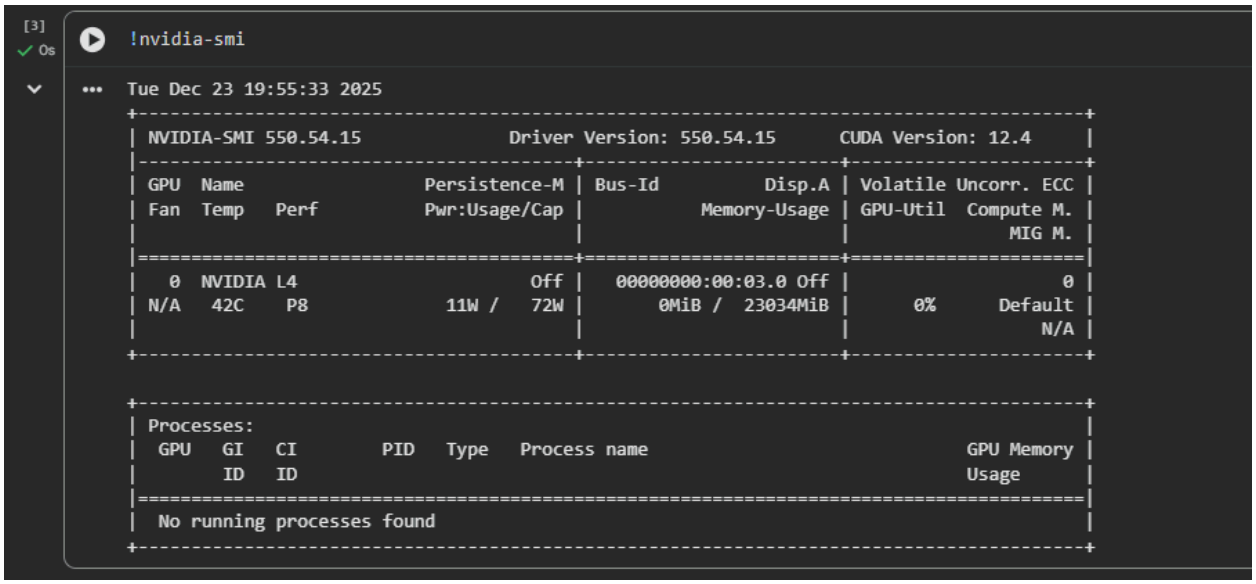
- Wheres wally #1 download - https://www.d-pdf.com/book/-download-pdf--where-s-wally--9781406313185

Appendix D - colab screenshots

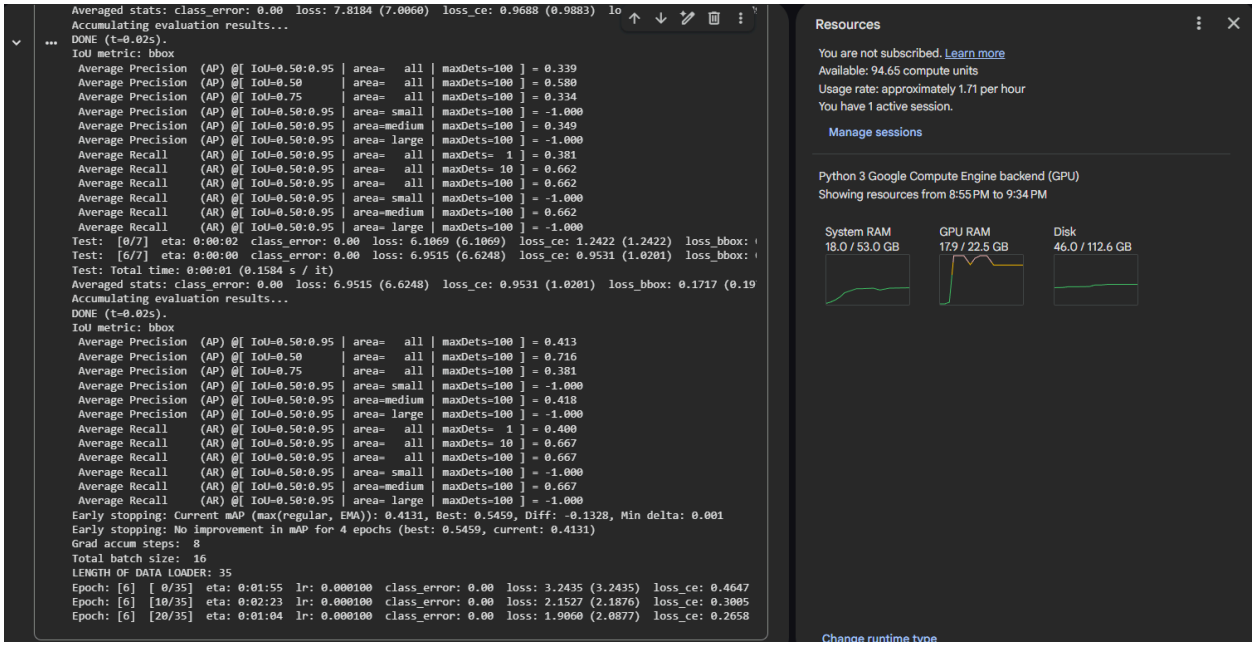
This shows the gpu i used for training on colab



This screenshot shows the details of the gpu (for nerds)



This screenshots shows me training the large model on google colab. Looking at the VRAM graph you can see how using AMP and Gradient Checkpoints drastically improved VRAM usage and made it much more stable:


Averaged stats: class_error: 0.00 loss: 7.8184 (7.0060) loss_ce: 0.9688 (0.9883) lo
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.339
Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] = 0.580
Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100] = 0.334
Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100] = -1.000
Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.349
Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100] = -1.000
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.381
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.662
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.662
Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100] = -1.000
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.662
Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100] = -1.000
Test: [0/7] eta: 0:00:02 class_error: 0.00 loss: 6.1069 (6.1069) loss_ce: 1.2422 (1.2422) loss_bbox:
Test: [6/7] eta: 0:00:00 class_error: 0.00 loss: 6.9515 (6.6248) loss_ce: 0.9531 (1.0201) loss_bbox:
Test: Total time: 0:00:01 (0.1584 s / it)
Averaged stats: class_error: 0.00 loss: 6.9515 (6.6248) loss_ce: 0.9531 (1.0201) loss_bbox: 0.1717 (0.19
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.413
Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] = 0.716
Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100] = 0.381
Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100] = -1.000
Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.418
Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100] = -1.000
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.400
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.667
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.667
Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100] = -1.000
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.667
Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100] = -1.000
Early stopping: Current mAP (max(regular, EMA)): 0.4131, Best: 0.5459, Diff: -0.1328, Min delta: 0.001
Early stopping: No improvement in mAP for 4 epochs (best: 0.5459, current: 0.4131)
Grad accum steps: 8
Total batch size: 16
LENGTH OF DATA LOADER: 35
Epoch: [6] [0/35] eta: 0:01:55 lr: 0.000100 class_error: 0.00 loss: 3.2435 (3.2435) loss_ce: 0.4647
Epoch: [6] [10/35] eta: 0:02:23 lr: 0.000100 class_error: 0.00 loss: 2.1527 (2.1876) loss_ce: 0.3005
Epoch: [6] [20/35] eta: 0:01:04 lr: 0.000100 class_error: 0.00 loss: 1.9060 (2.0877) loss_ce: 0.2658
Resources
You are not subscribed. Learn more
Available: 94.65 compute units
Usage rate: approximately 1.71 per hour
You have 1 active session.
Manage sessions
Python 3 Google Compute Engine backend (GPU)
Showing resources from 8:55 PM to 9:34 PM
System RAM
18.0 / 53.0 GB
GPU RAM
17.9 / 22.5 GB
Disk
46.0 / 112.6 GB
Change runtime type