Delft University of Technology

CS4240 Deep Learning

# Reproducibility Project: Self-Supervised Learning of Object Parts for Semantic Segmentation

*Authors:*

Group 14

Junhwi Mun (5359511)
Ahmed Alazzawy (4564995)
Samuel Karskens(4962958)

April 14, 2024

# Part 1: Abstract

This blog explains the process of reproducing the deep learning paper titled "Self-Supervised Learning of Object Parts for Semantic Segmentation," authored by Adrian Ziegler and Yuki M. Asano, published on April 27, 2022 [2]. The paper proposes a self-supervised learning approach for object parts, aimed at guiding the model to acquire representations corresponding to various potential object categories in unsupervised image segmentation. It primarily showcases enhancements in the object extraction capabilities of a ViT (Vision Transformer) and establishes new state-of-the-art results for fully unsupervised semantic segmentation, commencing from DINO [1] initialization.

# Part 2: Methods introduced in the paper

The paper introduces three main methods to demonstrate improvements compared to DINO. Before delving into these methods, let's briefly discuss the concept of DINO. DINO, an acronym for "Distillation of Images for Neural Networks with Overlapping," is a deep learning technique presented in a paper by Touvron et al. in 2021. DINO is tailored for training vision transformers without the use of labeled data, relying solely on a form of self-supervision. The process of labeling data is expensive which is not required when using DINO.
The core concept behind DINO lies in training a vision transformer to predict the relative positions of patches within an image. This involves comparing representations of different patches within the same image, rather than contrasting them with fixed representations of classes as in supervised learning. By employing a contrastive loss function, DINO incentivizes the model to learn meaningful representations of input images without explicit labels.
In our exploration, we delve into three distinct methods: Leopart, Cluster Based Foreground Extraction (CBFE) and overclustering with community detection (CD). Let's start with Leopart, which focuses on learning object parts through a dense image patch clustering task. This method offers a systematic approach to self-supervised dense representation learning. After an initial phase of generic pretraining, Leopart allows for the assembly of learned object parts into complete objects, aligning with benchmarks.

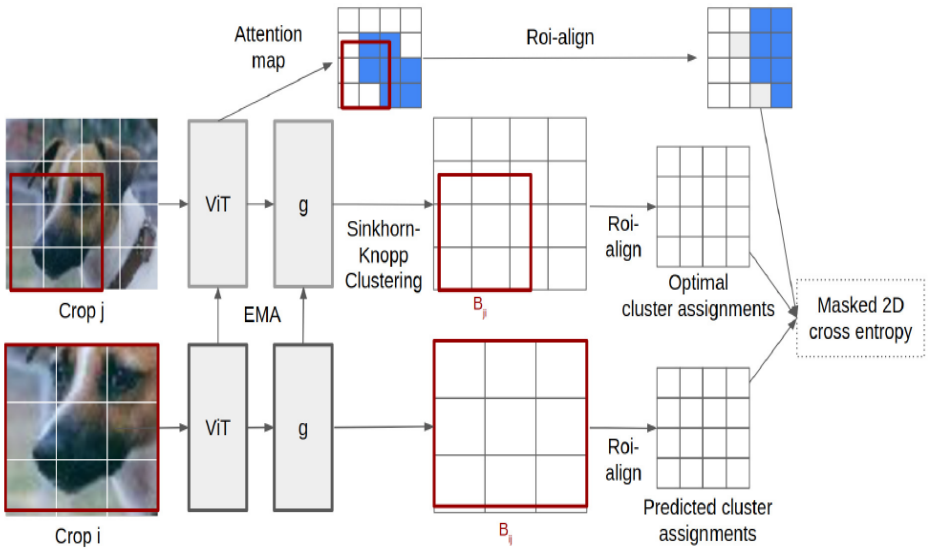



Figure 1: Architecture of Leopart [2]

The Leopart training journey, depicted in Fig. 1, kicks off with DINO initialization. Here, different image crops are fed into both the student and teacher networks to generate predictions for patch-level clusters and optimal assignments. An alignment process follows to ensure coherence between cluster targets and assignments. Moreover, attention is directed towards foreground patches using the ViT's attention map. While Leopart exhibits higher semantic segmentation mIoU values compared to DINO, its object extraction capability slightly lags behind.
Additionally, leveraging CBFE and CD methods, as illustrated in Fig. 2, further enhances foreground extraction

and semantic segmentation. These methods serve as crucial extensions to Leopart, contributing to the overall improvement in performance.

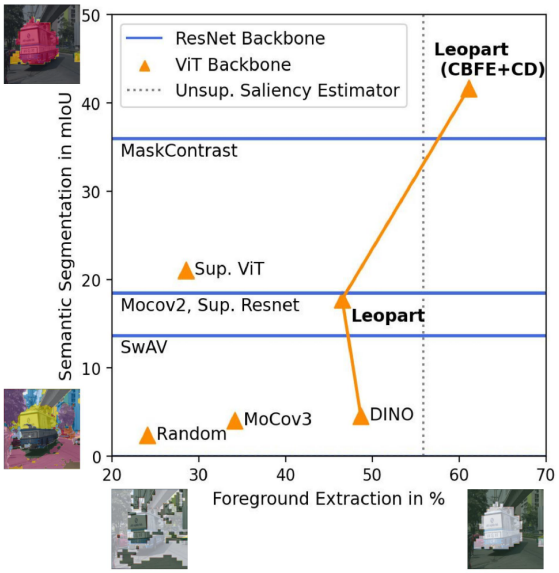



Figure 2: Metric scores from several methods [2]

## Part 3: Target and method of reproduction

The main focus of the reproduction is getting the mean Intersection over Union (mIoU) values depicted in the table represented in figure 4. The table provides us with mIoU values for several steps within the process. To get an intuition of the IoU metric, we refer to Figure 3.

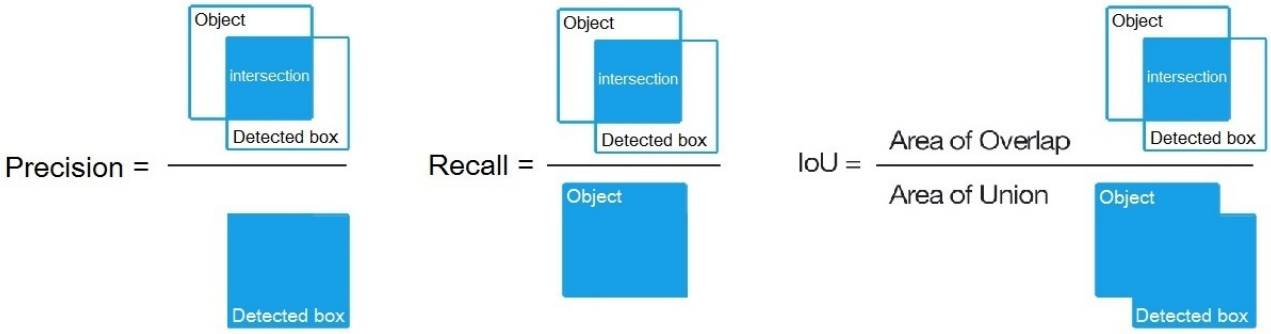



Figure 3: Precision, Recall and IoU metrics

| | mIoU |
|---|---|
| K=150 | 48.8 |
| DINO | 4.6 |
| + Leopart | 18.9 (+14.3%) |
| + CBFE | 36.6 (+17.7%) |
| + CD | 41.7 (+5.1%) |

Figure 4: Table to be reproduced [2]

The initial step has a mIoU of 4.6 % and represents only using DINO. The second step is using Leopart to fine-tune DINO's parameters. We see an mIoU of 18.9 % which is an increase of 14.3 %. The subsequent steps of CBFE and CD add an additional 17.7% and 5.1% improvement respectively. Starting from DINO, taking the approach described in the paper, an overall improvement of mIoU of 37.1 % is presented. These steps are all derived from evaluating the models on the Pascal VOC dataset. Other possible datasets to be used are Imagenet or Coco. Unfortunately using these datasets requires access to them and sufficiently more compute. Pascal VOC is a smaller dataset with fewer classes (K = 21, 20 classes and 1 background) and simpler scenes. To reproduce the table above, the given code from the authors for this paper is used. The provided code is extensive but lacks sufficient documentation. In the given code in GitHub, the file called `"sup_overcluster.py"` and `"fully_unsup_seg.py"` are mainly used.

For DINO, the paper presents different models that vary in size and patch size. Specifically, the paper uses both the vits16 and vitb8 models. For the table presented in Fig. 4, the authors of the paper utilized the vits16 model. However, they also discuss the improvements observed when using the larger base model, vitb8, in subsequent sections of the paper. In our analysis, we focus solely on reproducing the results obtained using the vits16 model, as presented in Fig. 4.

In the subsequent section, we will present the reproduced output mIoU of each method as depicted in Fig. 4. This demonstration aims to substantiate the actual improvements achieved by showing all intermediate steps.

## Part 4: Output

Before reproducing the table 4, we started out by running the given code. Unfortunately, there are several small errors that required some digging into the code to fix. We will start by denoting these errors and our fixes.

1. The repository provides an `environment.yml` file to recreate the environment. Unfortunately, the environment file not only lists the required packages but is made for the exact same setup as the authors. Since we don't have this exact setup, we could not use their environment file initially. To fix this we approached it in two ways. For the environment running locally on Windows, we ran the code and fixed the environment errors as they came up. Since we did want to use their provided environment file, we also started a cloud approach where we used Google Cloud to create a virtual machine. In the virtual machine running a Google Cloud image with Debian 11, Python 3.10. With CUDA 11.8 preinstalled we were able to setup the environment without any problems.

2. The root folder contains several folders which have Python files within them. These Python files hold functions and classes which need to be imported by experiment files. Unfortunately, the required folders are not added to the code as a module. Therefore we need to add these folders as a module package. There are several ways to do so. The provided readme is not extensive in describing this step: "Export the module to PYTHONPATH with export PYTHONPATH="$PYTHONPATH:PATH_TO_REPO"".

   We fixed the error by first adding an empty `__init__.py` files in the folders. This tells Python that these folders can be viewed as modules. Another approach is to add the following code before importing the modules.

   ```
   ## Code to fix the data module errors
   import sys

   # Set the root folder to the current working directory
   root_folder = os.getcwd()

   # Add the root folder to the Python path
   sys.path.append(root_folder)
   ```

   This code block will add the root folder to the system PATH resulting in fixing the module not found errors. It is important that the working directory (the location from which the program is run) is set up as the root folder of the project.

3. The experiments come with config files which will be loaded when running an experiment. The config file holds the values for the hyperparameters and directories. After updating our directories within the config

files there was still an error. The `arch` was in some config files denoted as `vit` but you could only use `vit-small, vit-base, vit-large`. Correcting the string for the `arch` fixes this error.

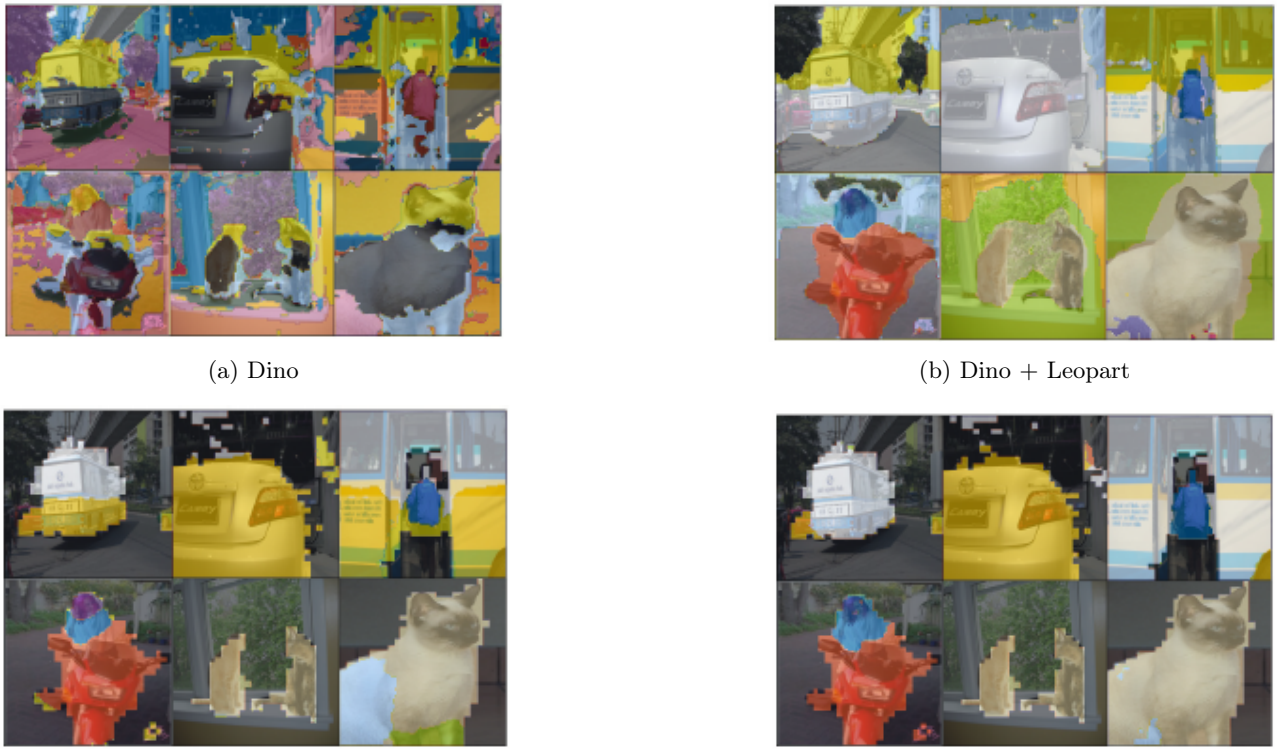
4. When running the file `fully_unsup_seg.py` it is required to provide a data directory and a save directory as arguments. This was not made clear in the readme. All other experiments take the data directory from the config file.

5. The `tqdm` code used was not working. There are several lines which used `tqdm.auto.tqdm`. This should be `tqdm.tqdm`.

Now that we have fixed the initial errors. We can start reproducing the paper's results. We start with a visual representation of the sub-steps after which we describe the commands and outputs.

## 4.1 Visual overview of mIoU improvements for each substep

To get an intuition for how each step improves the segmentation, the authors provide four images that showcase the improvements. Figure 5 shows how only using DINO 5a does segment the images but very poorly. We see a large improvement by just fine-tuning DINO with Leopart 5b. Additional improvements are done with CBFE 5c and CD 5d.



(a) Dino

(b) Dino + Leopart

(c) Dino + Leopart + CBFE

(d) Dino + Leopart + CBFE + CD

Figure 5: Visual overview of segmentations per step, [2]

## 4.2 sup_overcluster.py

The `fully_unsup_seg.py` file does an evaluation of all the steps. Nevertheless, it will only print out mIoU results for the CBFE and CD steps. Therefore to obtain the results for DINO and Leopart, we use the `sup_overcluster.py` file. To obtain the evaluation scores, we use the `dino.yml` file for the DINO results and the `leopart-vits16.yml` for the Leopart results. It is worth noting that while segmentation and clustering are not exactly the same, segmentation can be achieved through clustering or by using a linear classifier.

Let's delve into the specific output of the DINO and Leopart configs. A command to evaluate a DINO model trained on ImageNet-100 using Pascal VOC configuration can be executed as follows:

```
python experiments/overcluster.py --config_path
    experiments/overcluster/configs/pascal/k=21/dino.yml
```

Similarly, for Leopart, we run the following command, replacing dino.yml with a different configuration named `leopart-vits16.yml`:

```
python experiments/overcluster.py --config_path
    experiments/overcluster/configs/pascal/k=21/leopart-vits16.yml
```

The mIoU results can be found in Tables 1 and 2. Please note that the evaluation is done in 5 separate experiments. We verify the reproduced values by comparing them with the table in fig 4. The mean mIoU values are 4.6% and 18.9% for DINO and Leopart, respectively, as stated in the paper. The differences with the reproduced values are only 0.03% and 1.1%, respectively. Overall, the reproducibility indicates slightly lower performance compared to the paper, but as the differences are small, they are deemed acceptable overall.

| **mIoU** | **Value** | **Expected value from paper** |
|---|---|---|
| mIoU 1 | 0.04891530062661679 | |
| mIoU 2 | 0.050452893420321235 | |
| mIoU 3 | 0.0446251508086259 | |
| mIoU 4 | 0.039160083462912165 | |
| mIoU 5 | 0.04554153668157963 | |
| **Mean mIoU** | 0.04573899300001114 **(4.57%)** | **4.6%** |

Table 1: DINO

| **mIoU** | **Value** | **Expected value from paper** |
|---|---|---|
| mIoU 1 | 0.18703784483065258 | |
| mIoU 2 | 0.20604395630665234 | |
| mIoU 3 | 0.17224440554034937 | |
| mIoU 4 | 0.1541286641365625 | |
| mIoU 5 | 0.16989675420805705 | |
| **Mean mIoU** | 0.17787032500445477 **(17.79%)** | **18.9%** |

Table 2: DINO + Leopart

### 4.3 fully_unsup_seg.py

To obtain the evaluations for the CBFE and CD steps we use the `fully_unsup_seg.py` file. This file does not run with a config file but requires inputting parameter arguments. The file does run without the $best_k$, $best_{mt}$ and $best_{wt}$ parameters. When that is the case it will run an additional function to find the best parameters. However, since we want to reproduce the table from the paper, we use the same parameters as the authors. We run the following command:

```
python experiments/fully_unsup_seg/fully_unsup_seg.py
--ckpt_path "\checkpoints\leopart_vits16.ckpt"
--data_dir "\data\VOCdevkit\VOC_data"
--save_folder "\masks"
--batch_size 4
--experiment_name vits16
--best_k 149
--best_mt 2
--best_wt 0.09
```

The batch_size was changed to 4 to be able to run the `fully_unsup_seg.py` locally. The default batch_size was 15.
The results of the evaluations can be found in tables 3 and 4. Please note that the final evaluation is done for 10 runs. We chose to leave it like that to reproduce the results from the authors.

| mIoU | Value | Expected value from paper |
|---|---|---|
| mIoU 1 | 0.34838576252064424 | |
| mIoU 2 | 0.32963564903839765 | |
| mIoU 3 | 0.3479762398498129 | |
| mIoU 4 | 0.3547392398825335 | |
| mIoU 5 | 0.3569189340856602 | |
| **Mean mIoU** | 0.3475311650754097 **(34.75%)** | **36.6%** |

Table 3: DINO + Leopart + CBFE

| mIoU | Value | Expected value from paper |
|---|---|---|
| mIoU 1 | 0.3807541364770226 | |
| mIoU 2 | 0.40159298148409145 | |
| mIoU 3 | 0.40159298148409145 | |
| mIoU 4 | 0.40159298148409145 | |
| mIoU 5 | 0.3793434737663346 | |
| mIoU 6 | 0.34503206019853394 | |
| mIoU 7 | 0.40159298148409145 | |
| mIoU 8 | 0.34503206019853394 | |
| mIoU 9 | 0.3807541364770226 | |
| mIoU 10 | 0.3299710240409878 | |
| **Mean mIoU** | 0.37373622296898936 **(37.37%)** | **41.7%** |

Table 4: DINO + Leopart + CBFE + CD

As can be seen in table 5, our reproduced results resemble the results from the authors. Even though there are small differences, we find these negligible.

| mIoU | Expected value from paper | Reproduced value |
|---|---|---|
| DINO | 4.6% | 4.57% |
| + Leopart | 18.9% | 17.79% |
| + CBFE | 36.6% | 34.75% |
| + CD | 41.7% | 37.37% |

Table 5: Comparison between expected values and reproduced values

# Part 5: Individual contribution

Overall, within our team, we initially embarked on individual attempts to run the code, exchanging logs in training via the Neptune AI platform. However, we encountered challenges due to the absence of necessary packages in the provided `environment.yml` file, leading to issues with functionality in specific environments. Consequently, a significant portion of our time was dedicated to configuring the environment and troubleshooting errors. Please note that our team was originally composed of 4 people, but in the beginning of the project, one team member dropped out of the course because of personal circumstances. Unfortunately, we therefore had to redefine our roles. We had two members focusing on the Google Cloud setup and one team member focusing on the local setup. An overview of who did what can be found below.

We also created a separate `showimages.py` file which lets the user select an image from an index. The file will then load the evaluated features (contained in `all_pascal_train.pt` and `all_pascal_val.pt`) and show the images to the user. Figure 6 shows the output of this file.

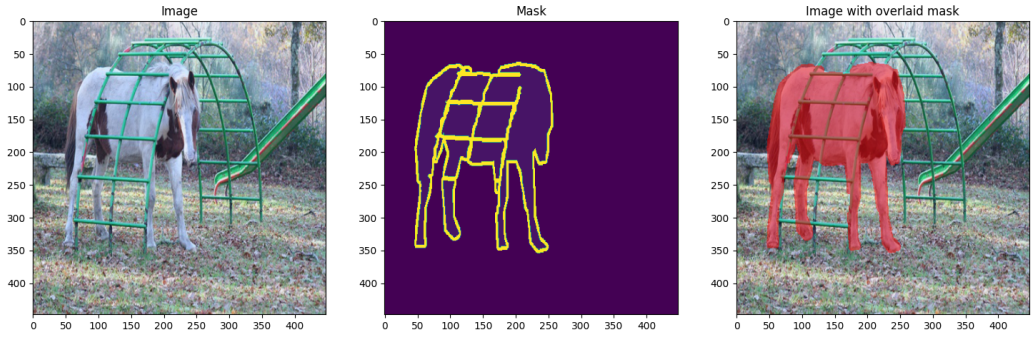



Figure 6: Output of index 8 of `showimage.py`

| **Task** | **Team member** |
|---|---|
| Fix the module not found errors | Ahmed |
| Fix the tqdm error | Ahmed, Junhwi, Samuel |
| Fix the dataset not found / corrupted error | Samuel |
| Fix the "vit" to "vit-small" error | Ahmed, Samuel |
| Initial blog draft | Junhwi |
| Created the local environment | Ahmed |
| Created the google environment | Junhwi, Samuel |
| Set up the neptune environment | Junhwi, Samuel |
| Ran the code locally | Ahmed |
| Ran the code on google cloud | Junhwi, Samuel |
| Visualisation code | Ahmed |
| Finalize the blog post | Ahmed, Junhwi, Samuel |

Table 6: Detailed overview on who did what

# References

[1] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. *CoRR*, abs/2104.14294, 2021. URL https://arxiv.org/abs/2104.14294.

[2] Adrian Ziegler and Yuki M Asano. Self-supervised learning of object parts for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14502–14511, 2022.