Wroclaw University of Science and Technology

Faculty of Electronics, Photonics and Microsystems

Field of study: Electronic and Computer Engineering

Engineering Thesis

Title:

EN: Application of active learning and self-labeling in pattern classification

PL: Wykorzystanie metod aktywnego uczenia oraz self-labelingu w klasyfikacji danych

Author:

Beniamin Weyna

Supervisor:

prof. dr hab. inż. Michał Woźniak

WROCŁAW 2023

Abstract:

This research builds upon the foundation laid out by the paper *Plug and Play Active Learning for Object Detection* [1], that introduced active learning pipeline, utilizing uncertainty-based sampling and diversity-based sampling, for object detection neural network models. My work implements the whole active learning algorithm in Detectron2 library from scratch, and it aims at deepening the understanding of why this method is so effective and at answering questions such as: How do uncertainty-based and diversity-based sampling individually contribute to the training process? Which of these methods should be given priority in different contexts? Will diversity in machine learning have similar characteristics to interleaved studying method used in education?

Table of Contents

0. Project overview 4
  0.1. Functional overview 4
  0.2. Aim of this research 4
1. Theoretical introduction – active learning background 7
  1.1. Importance of efficient data usage 7
  1.2. Active learning 8
    1.2.1. What is active learning? 8
    1.2.2. Uncertainty-based sampling 9
    1.2.3. Diversity-based sampling 10
  1.3. "Plug and Play Active Learning for Object detection" algorithm overview 11
  1.4. My work 13
    1.4.1. Failure of initial plan – problems with PPAL project 13
    1.4.2. Final approach 16
2. Implementation 17
  2.1. Project core – training a model 17
    2.1.1. Main program 17
    2.1.2. Detectron2 basic functions 18
    2.1.3. Modularity with a config file 18
  2.2. Uncertainty-based sampling 19
    2.2.1. Image uncertainty – Entropy calculation 19
    2.2.2. Class uncertainty 20
    2.2.3. Probability based uncertainty selection 20
  2.3. Diversity-based sampling 21
    2.3.1. Cosine Similarity 22
    2.3.2. CCMS 22
    2.3.3. Distance matrix 23
    2.3.4. Speeding up the compute 23
    2.3.5. Clustering 24
  2.4. Ensuring reproducibility 26
  2.5. Final pipeline 28
3. Experiments 29
  3.1. Experimental setup 29
  3.2. Results 31
4. Conclusions 33
  4.1. Summary 33
  4.2. Detailed conclusions 34
Bibliography 36
Appendix A – Numerical results of the main experiment 37
Appendix B – Email sent to the main PPAL author 38

0. Project overview

0.1. Functional overview

This project implements an active learning algorithm for training object detection models, described in the paper *Plug and Play Active Learning for Object Detection* [1]. I built the project in Detectron2 [2] from scratch, while the source code for the paper has been based on MMDetection [3].

It can train any object detection model based on any backbone network, that are both supported and maintained in Detectron2 library [2]. For now, I support any dataset with annotations in YOLO format.

Each model training consists of initial training round, and then a specified number of active learning rounds, for which the percentage factor of images picked by uncertainty has to be specified. That enables the user to experiment and compare final performances of models, that were trained on the same dataset, but with different sampling methods: all round training images picked by uncertainty sampler, all picked by diversity or half-half between these 2 methods, or any combination in between.

0.2. Aim of this research

Algorithm implemented in the publication *Plug and Play Active Learning for Object Detection* [1] introduced a new algorithm, that beats the performance of prior active learning algorithms by picking uncertain images, that are "problematic" for the given model, and then picks diverse samples from that pool to decrease the probability of a model's weights being stuck in various local minima.

So, simplifying it, we can say that PPAL project introduced a truly plug and play, ready algorithm that beats the performance of other training methods.

My work, however, focuses more on the research aspects of it. To examine the learning methods, to investigate why PPAL algorithm beats competitors. I wanted to dig deeper, understand what makes the combination of uncertainty and diversity so effective.

While studying machine learning, one can notice the accurate analogies between how humans learn and how machines learn and how much they can teach us about ourselves through studying Artificial Intelligence. Just as there are many different active learning methods in machine learning, there are also many methods used in human learning.

There is a book, that I read about 6 years ago, in 2017, about efficient studying methods *Hack into the brain* (Original in polish: *Włam się do mózgu*) written by Radosław Kotarski [4]. In one chapter Kotarski examined a training routine of Muhammad Ali, regarded as the greatest heavyweight boxer of all time [5]. Ali's approach to training was unusual, since he was not perfecting each move one by one, as was done traditionally. He was constantly changing what he was training, which made him seem like he was constantly bored.

This style of learning is called interleaved practice, which is the opposite of blocked practice.

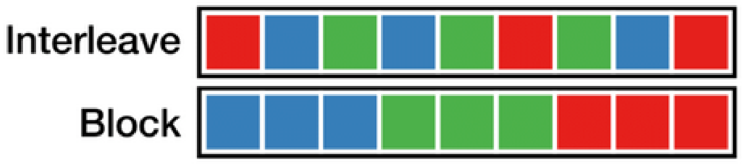


Figure 1. Interleaved vs blocked practice [6] – completing tasks in mixed order or tasks blocked by category.

The discussion about which method of learning is more effective has been going on for years, but interleaved practice does really seem to work better in the long term. Even though, we know for a fact, that students do not prefer interleaved practice and they prefer blocked studying, as proved by the source presented on the Figure 2.

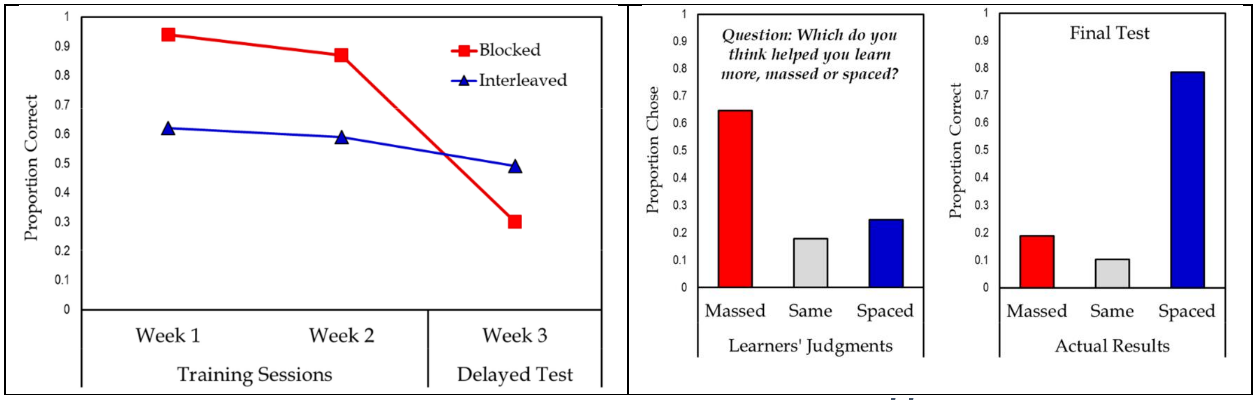


Figure 2. Findings on interleaved vs blocked learning [7].

The explanation for that might be a, so called, discriminative-contrast hypothesis, which suggests that people during blocked training roughly know what to expect, so the task itself becomes easier. It gives them a false sense of security, that they really grasped the concept. Interleaved practitioners report much lower confidence in their knowledge, but they score higher, since they had to put a lot of effort to complete a task.

Why am I talking about all this? The methods of active learning in machine learning might have exactly the same nature. Diversity sampling is analogous to interleaved learning since we prioritize the diversity of given problems. Uncertainty learning does not have that strong an analogy with blocked practice, but we can still think of it as just prioritizing hard problems, without caring about how different they are from each other. It should still be better than random sampling, but it does not offer any mechanisms for dealing with certain, but wrong answers.

Significance of this analogy and of the discussion around it cannot be overstated, especially when we think of how modern education system works. It is completely normalized that schools teach one subject after another, in carefully chosen order. Even despite the fact that great evidence has been shown that interleaved studying is more effective in the long-term.

I understand that it is definitely much harder to organize the school around such learning style, just how we know that individual tutoring is much effective than classroom teaching, but as society we cannot simply provide multiple tutors for every student.

However, are the same limitations applicable to machine learning, where most work is performed by computers? Does the diversification of encountered problems also benefit much simpler models than human brains?

One of my theses to examine is that **just as people perform better after interleaved practice, object detection models will also perform better when subjected to diversified training data.** We will see if the analogy is viable and to what extent we should prioritize diversity-based sampling.

On top of deepening our understanding of current active learning methods, **my research also aims at helping to develop semi-supervised learning methods**, which can take advantage of orders of magnitude more data because of how time, energy and money consuming it is to properly label it.

Thus, semi-supervised methods, that include active learning, are predicted to be one of the most prospective branches of machine learning. They enable models to not only use more data, but also to build intuition and common sense, that drastically speed up learning process. After all, this is how human babies learn, when they do not yet fully use language: they find patterns in lots of data, but very few labels. Figure 3 visualizes how even unlabeled data can help in properly classifying datapoints.

More information on significance of developing active learning has also been provided in the chapter “1.1. Importance of efficient data usage”.

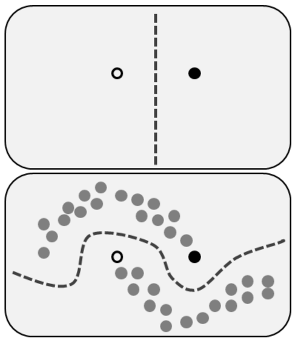


Figure 3. “An example of the influence of unlabeled data in semi-supervised learning. The top panel shows a decision boundary we might adopt after seeing only one positive (white circle) and one negative (black circle) example. The bottom panel shows a decision boundary we might adopt if, in addition to the two labeled examples, we were given a collection of unlabeled data (gray circles)” – cited from [8].

1. Theoretical introduction – active learning background

1.1. Importance of efficient data usage

Current state of deep neural networks requires providing the model with a lot of labeled data, especially for complex environments, and especially when we contrast it with how quickly humans can learn to perform the same task. The ability to learn the most out of little information will be one of the most significant areas of future research on Artificial Intelligence, as pointed out by Yann LeCun, a Chief AI Scientist at Meta [9], who said:

“Current LLMs are trained on text data that would take 20,000 years for a human to read. […] Humans get a lot smarter than that with comparatively little training data.” [10]

In a separate post, he wrote:

“My money is on new architectures that would learn as efficiently as animals and humans. Using more data (synthetic or not) is a temporary stopgap made necessary by the limitations of our current approaches.” [11]

LeCun said it in a context of Large Language Models, but the idea might be generalized to most, if not all, deep neural networks. The main point, however, is that we quickly approach the limit of how much data we can feed the model for it to become better.

The most capable state-of-the-art models, like ChatGPT-4, have already been trained on vast amounts of human-generated content, and have started to be colloquially referred to as internet compressors [12], since it becomes more and more convenient to generate a response to a query or generate an image instead of searching for it with “traditional” methods, like using a search engine.

The sheer scale of training content needed to train these models is what made them a breakthrough and a step forward in the capabilities of AI, but, as LeCun pointed out [11], it is also what limits them from further innovations, as we simply run out of high-quality, informative data.

The way forward seems to be a highly curated synthetic data, to push the current approach to its limits. However, another path will be to design new architectures, that can get closer to the learning performance of humans, and then eventually surpass it.

For these reasons, it is crucial to investigate and formalize efficient learning. Active learning is merely an attempt at the beginning of this investigation. A small part of an underinvested field, compared to the potential it holds.

1.2. Active learning

1.2.1. What is active learning?

Active learning by definition is a:

> *special case of machine learning in which a learning algorithm can interactively query a user (or some other information source) to label new data points with the desired outputs* [13].

Which means that instead of training the model on as much data as we possibly can within our budget, we approach it strategically and determine which datapoints would give us the most benefit. In other words: we actively pick how to wisely spend our “labeling budget”.

For testing purposes, of course, we have all samples annotated and we just pretend not to have them. When an image is picked for annotation we simply pull the annotation file, pretending that someone has labeled it in the meantime.

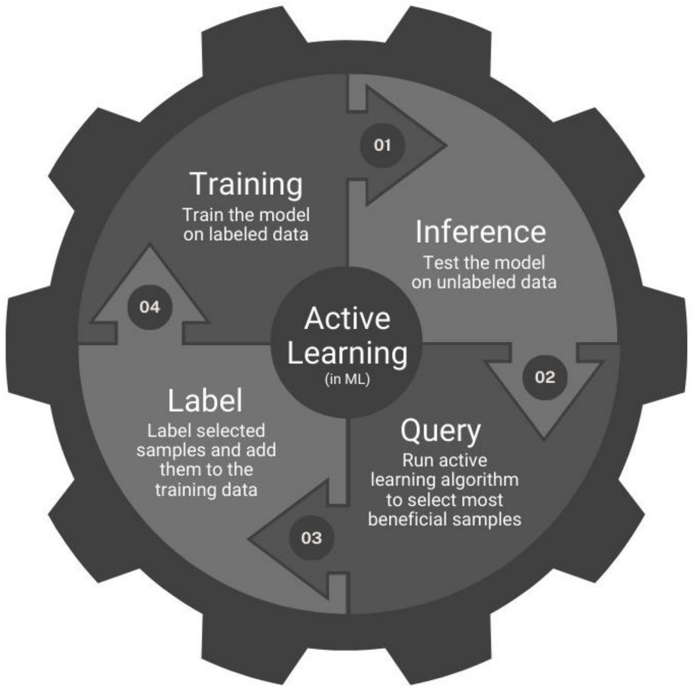

01
Training
Train the model on labeled data
Inference
Test the model on unlabeled data
04
Active Learning
(in ML)
02
Label
Label selected samples and add them to the training data
Query
Run active learning algorithm to select most beneficial samples
03

Figure 4. Diagram visualizing active learning loop.

Labeling is among the most expensive steps of training such models since it requires a human to do it. Often a specialized, costly human. It also takes a lot of time, especially for professional use-cases with a small margin of error.

While more training examples lead to better performance of the model, this labeled data will always be a precious resource. Google is one of the companies historically known for understanding the significance of it.

Google wanted to “organize the world’s information”, which is one of their slogans till this day, and they invested in CAPTCHA security measure [14], that used to train their machine learning models to decipher scans of old books and house numbers from Street View’s images [15]. The scale of this labeling process eventually led to satiating the learning capabilities of the model, that got much better at reading house numbers than an average human, as proven in the paper *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks* [16].

So, essentially, active learning helps in achieving best possible model performance, without having as much labeled data as Google, who got vast amounts of it through using unconventional and hard to replicate methods.

1.2.2. Uncertainty-based sampling

One of the first active learning methods,, and very popular due to its simplicity, is uncertainty sampling, that focuses on picking the most uncertain, the most ambiguous samples, that in theory should be the hardest to classify.

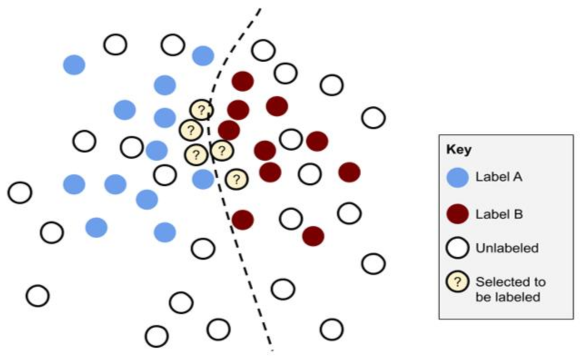


Figure 5. Uncertainty-based sampling simplified visualization [17]. Picked for the training are the most uncertain samples – the ones close to the decision border, that could be swayed to any label by a minor difference.

In information theory there is a term closely related to uncertainty, which is “entropy”. The measurement of entropy of a specific sample is often called a measure of how “informative” the sample is. This seems to be a great and simple approach – putting emphasis on the hardest samples. It is an intuitive approach that people use themselves during studying. However, it’s not all that optimal.

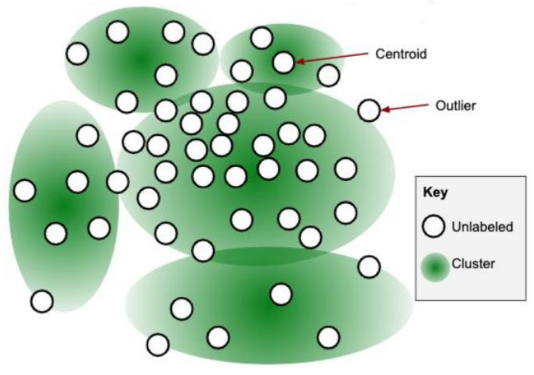
There are various different implementations of this method, and I’ll get into the specifics in future chapters, but the main problem here is that we prioritize being certain, not necessarily being correct. The connection is not obvious, but if the model is confident about a wrong label, uncertainty sampling has no mechanism for correcting that mistake, which takes considerable toll on the final model evaluation. Another problem is that a model can train on a narrow set of problematic samples, while deteriorating its performance on easier samples.

1.2.3. Diversity-based sampling

Diversity sampling is an active learning method, which prioritizes diverse samples, where the word “diverse” can also mean quite a few different things. There are a few known, established methods of calculating the diversity, like the ones on the Figure 6.

Cluster-based Sampling

For capturing all the meaningful trends in data’s feature space



Representative Sampling

For representing the specific target domain and adapt to it as fast as possible

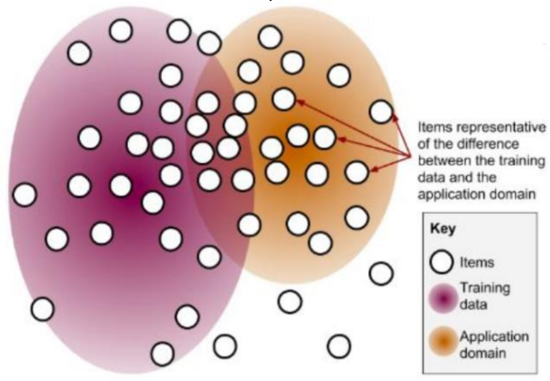

Items representative of the difference between the training data and the application domain
Key
Items
Training data
Application domain

Figure 6. Example 2 diversity sampling methods [18].

I could get into the specifics, exact clustering algorithms, different algorithms parameters, but I provide these examples just to show the rough idea of diversity sampling.

The proposed new method described and implemented in this paper will have similar goals, but different calculation methods, that are discussed in greater detail in a chapter “2.3. Diversity-based sampling”.

1.3. “Plug and Play Active Learning for Object detection” algorithm overview

The paper *Plug and Play Active Learning for Object Detection* [1] introduced a state-of-the-art 2-step active learning process, combining uncertainty-based and diversity-based sampling in series.

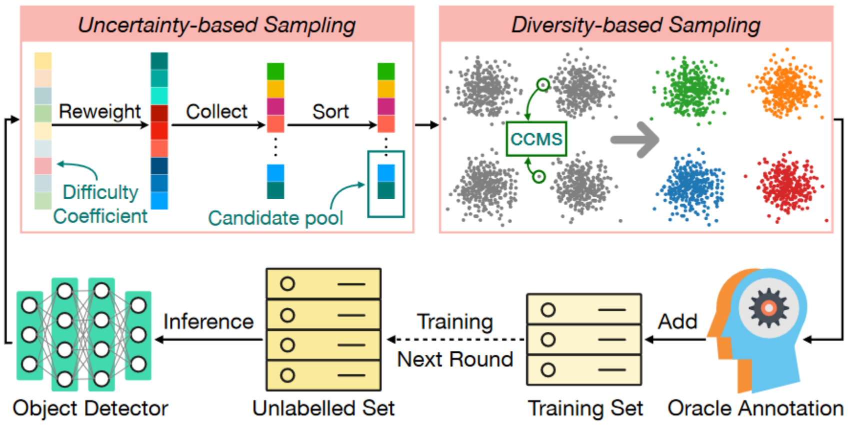


Figure 7. An overview of two-stage PPAL algorithm [1].

First step – Difficulty Calibrated Uncertainty Sampling (DCUS) introduces an easy solution to measuring uncertainty of both classification and localization, which was challenging to implement, since previous solutions relied on “modifying the architecture of an object detector as well as the training pipeline” [1]. With DCUS, the algorithm can be easily integrated into existing object detector frameworks.

The algorithm here also calculates the combined uncertainty of all objects from a certain category, which enables it to more frequent sampling of objects with challenging categories. This is an effective solution to representation imbalance of certain classes in our dataset, which usually causes the model to be accurate with some classes but underperform with others.

Second step – Category Conditioned Matching Similarity (CCMS) is another authors’ original method of diversity sampling. Its main presupposition is that images are as similar as similar are objects they contain. But the objects are determined mostly by the model itself, so it’s not an objective determinant of similarity of images, but rather subjective for every model, which in theory should get closer to the objective similarity as the model gets better.

Following the pipeline depicted in the Figure 7, the uncertainty sampling’s output is a candidate pool of images that will be considered in diversity-based sampling.

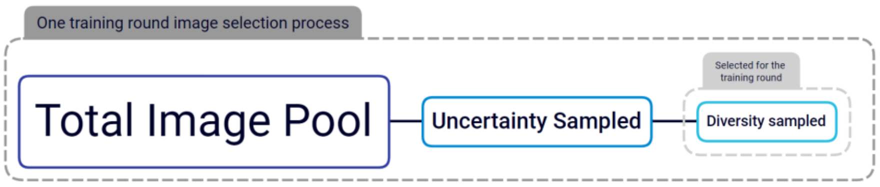

One training round image selection process
Total Image Pool
Uncertainty Sampled
Selected for the training round
Diversity sampled

Figure 8. Diagram of image selection process.

The model was trained on a fraction of the entire available image dataset (like COCO), thus, there was always a wide pool of images to choose from. How many exactly? The number has not been directly stated in the publication [1], but checking its source code [19] I found out that first, of the whole image pool there are 4 times the target amount images left after uncertainty sampling, and out of this pool, the final samples are chosen by diversity.

From what authors have described in the paper, their experiments involved an initial training round on random images, that were different every time, but they ran the experiments 3 times and reported the average performance of the three to minimize the influence of randomness. It is not the greatest solution, and I, for my experiments, will ensure the replicable behavior of my algorithm more strictly, so every model's performance will be roughly the same after initial training (more in the chapter: "2.4. Ensuring reproducibility").

After this initial round, PPAL authors ran from 4 to 9 rounds of active learning, with a constant fraction of the whole dataset selected for training. Figure 9 depicts the example results presented in the *PPAL* paper [1], to get a sense of their benchmarking process. For RetinaNet model used on COCO dataset, we can see that after initial training on 2% of the dataset, another 2% of images has been selected for each training round up to 4 active learning rounds.

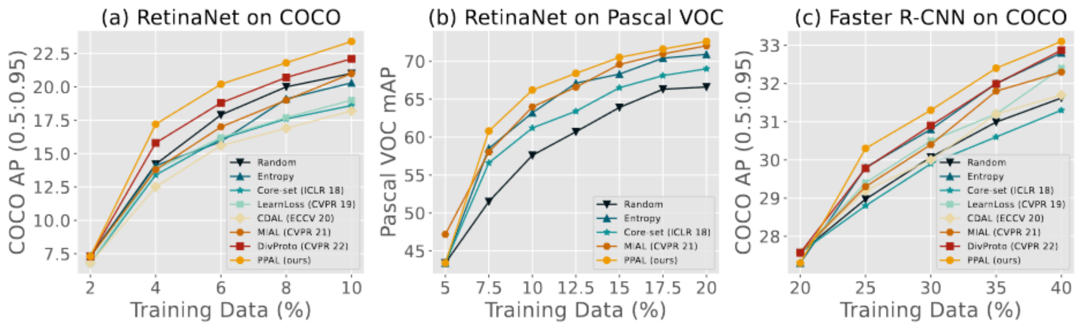


Figure 9. Example results obtained by the PPAL algorithm [1].

After understanding the high-level benchmarking process, the only thing left is diving into the details of the algorithms alongside its equations, but I will describe it more profoundly in the section dedicated to implementation (see chapter: "2. Implementation") when it will be relevant to mention the PPAL algorithm details.

At the end, it is worth mentioning that the project was based on the MMDetection library [3] and its source code has also been provided on GitHub [19].

1.4. My work

1.4.1. Failure of initial plan – problems with PPAL project

My initial plan assumed running the Plug and Play Active Learning for Object Detection project [19] and modifying it or adding features. The availability of the source code has been one of the main factors, based on which I chose to work on this project.

Unfortunately, despite the provided code, I did not manage to run it due to countless and never-ending problems.

Even the data_setup.sh and data_setup.py that were supposed to be run at the very beginning to prepare datasets had some issues:

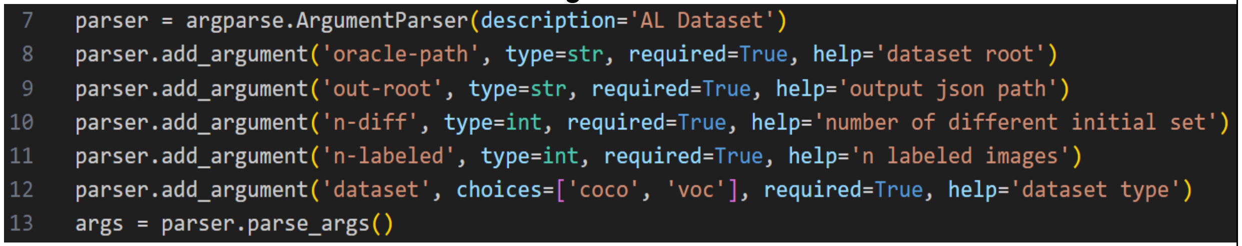
data_setup issue 1: Error 'required' is an invalid argument for positionals.

It was a problem caused by a ‘required’ keyword given to a positional argument in argument parser, which are always required by definition.

I made a quick, superficial search of historical changes of argparse, and it appears like the original project syntax had never been the correct syntax to begin with, which is odd, to say the least.

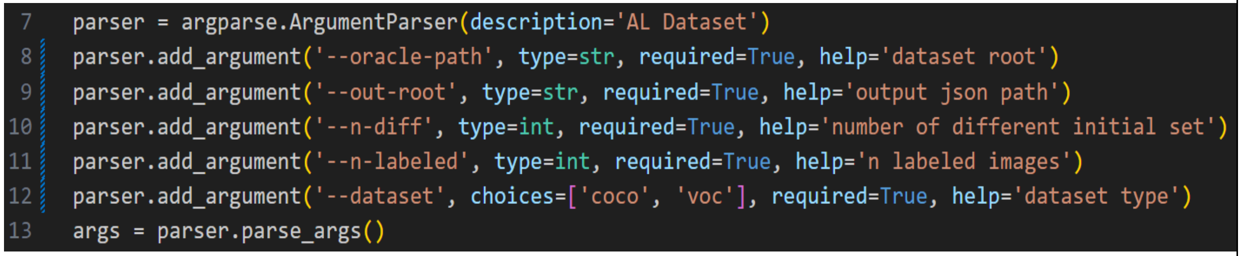
I managed to fix the issue by adding dashes, that would turn all these arguments into required option arguments, so they have to be provided, although in any order.

Original code



```
parser = argparse.ArgumentParser(description='AL Dataset')
parser.add_argument('oracle-path', type=str, required=True, help='dataset root')
parser.add_argument('out-root', type=str, required=True, help='output json path')
parser.add_argument('n-diff', type=int, required=True, help='number of different initial set')
parser.add_argument('n-labeled', type=int, required=True, help='n labeled images')
parser.add_argument('dataset', choices=['coco', 'voc'], required=True, help='dataset type')
args = parser.parse_args()
```

Fixed code



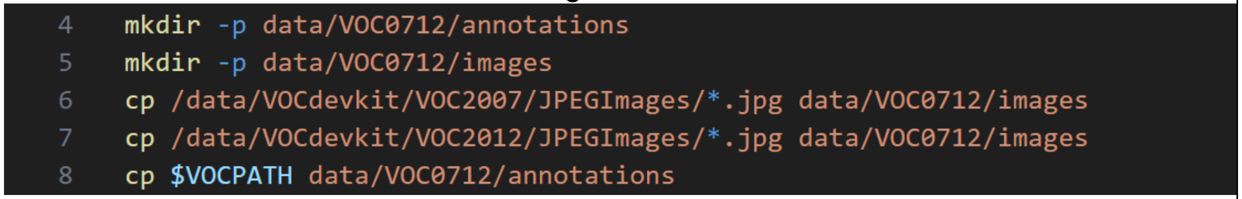
```
parser = argparse.ArgumentParser(description='AL Dataset')
parser.add_argument('--oracle-path', type=str, required=True, help='dataset root')
parser.add_argument('--out-root', type=str, required=True, help='output json path')
parser.add_argument('--n-diff', type=int, required=True, help='number of different initial set')
parser.add_argument('--n-labeled', type=int, required=True, help='n labeled images')
parser.add_argument('--dataset', choices=['coco', 'voc'], required=True, help='dataset type')
args = parser.parse_args()
```

Figure 10. Original and fixed code, that solved issue 1 of PPAL data_setup source code.

data_setup issue 2: Misplaced dashes in dataset path

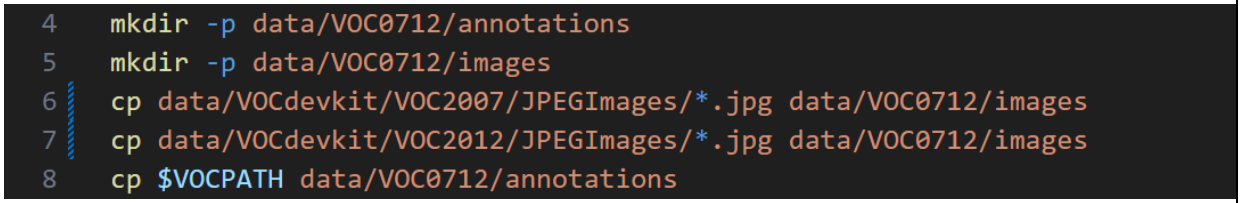
Next problem were misplaced dashed in the paths of dataset images, which did not agree with other paths, written one line above it.

Original code



```
mkdir -p data/VOC0712/annotations
mkdir -p data/VOC0712/images
cp /data/VOCdevkit/VOC2007/JPEGImages/*.jpg data/VOC0712/images
cp /data/VOCdevkit/VOC2012/JPEGImages/*.jpg data/VOC0712/images
cp $VOCPATH data/VOC0712/annotations
```

Fixed code



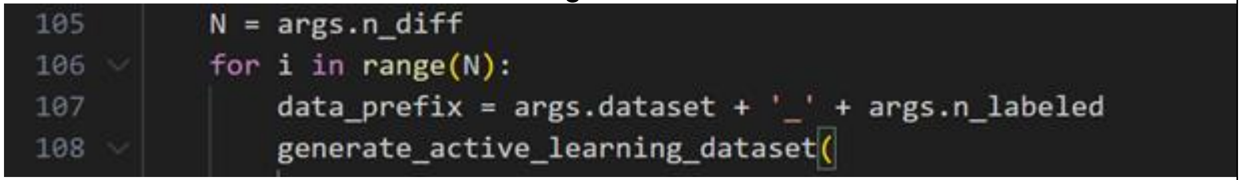
```
mkdir -p data/VOC0712/annotations
mkdir -p data/VOC0712/images
cp data/VOCdevkit/VOC2007/JPEGImages/*.jpg data/VOC0712/images
cp data/VOCdevkit/VOC2012/JPEGImages/*.jpg data/VOC0712/images
cp $VOCPATH data/VOC0712/annotations
```

Figure 11. Original and fixed code, that solved issue 2 of PPAL data_setup source code.

data_setup issue 3: Lack of casting integer to a string

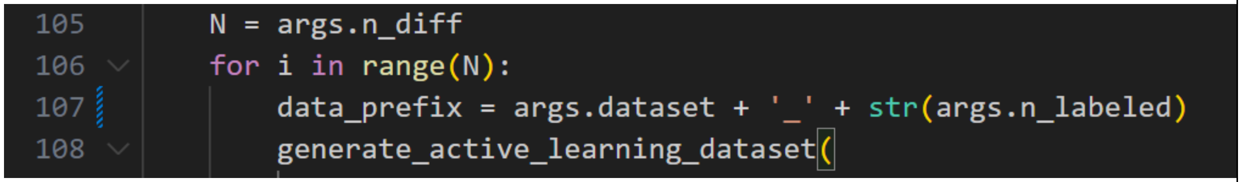
Last issue with the data_setup program, was the lack of casting of an ‘n_labeled’ argument to a string, without which the concatenation of strings would not be possible.

Original code



```
    N = args.n_diff
    for i in range(N):
        data_prefix = args.dataset + '_' + args.n_labeled
        generate_active_learning_dataset(
```

Fixed code

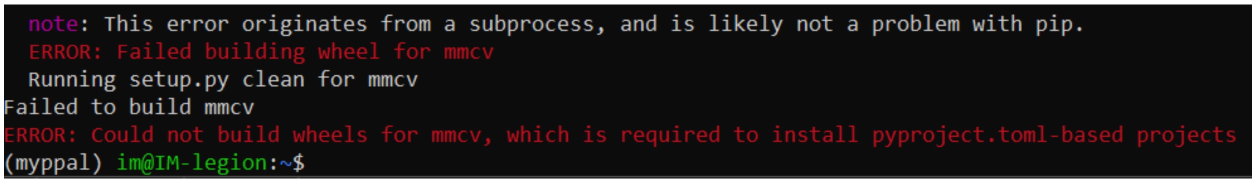


```
    N = args.n_diff
    for i in range(N):
        data_prefix = args.dataset + '_' + str(args.n_labeled)
        generate_active_learning_dataset(
```

Figure 12. Original and fixed code solving issue 3 of PPAL data_setup source code.

After finally running the data setup, I tried running the main active learning program with no success.

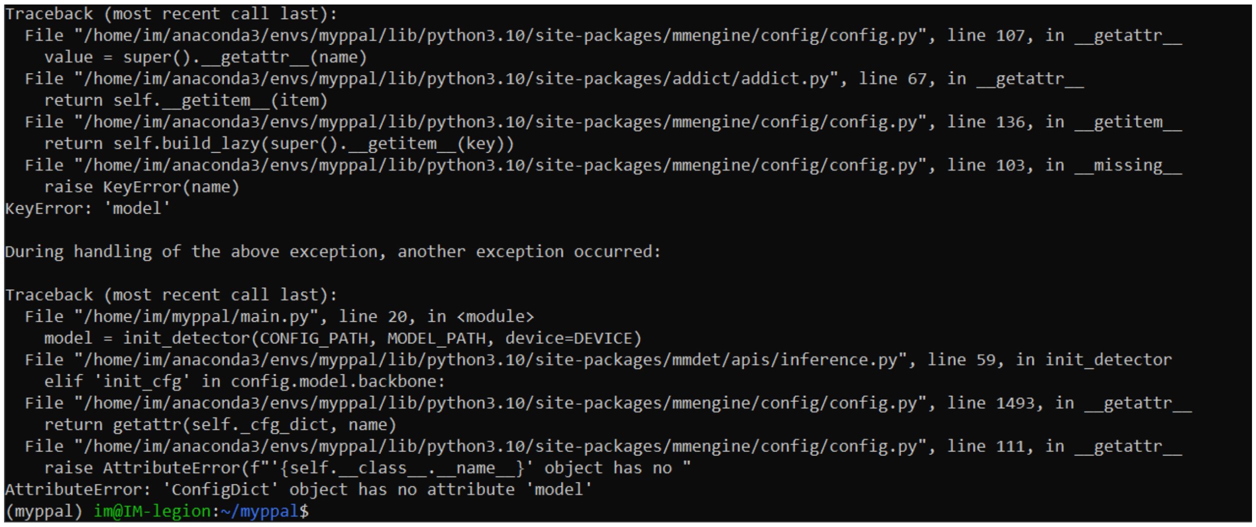
The first error, but also the relatively easy one to fix was "Failed building wheel for mmcv". With time I figured out it is caused by incompatibilities between mmcv library versions, PyTorch and/or CUDA versions.



```
  note: This error originates from a subprocess, and is likely not a problem with pip.
  ERROR: Failed building wheel for mmcv
  Running setup.py clean for mmcv
Failed to build mmcv
ERROR: Could not build wheels for mmcv, which is required to install pyproject.toml-based projects
(myppal) im@IM-legion:~$
```

Figure 13. Failed building wheel for mmcv.

But that was only the beginning, since after fixing this I got an attribute error ConfigDict object has no attribute 'test_dataloader' alongside another error in a config file.



```
Traceback (most recent call last):
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmengine/config/config.py", line 107, in __getattr__
    value = super().__getattr__(name)
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/addict/addict.py", line 67, in __getattr__
    return self.__getitem__(item)
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmengine/config/config.py", line 136, in __getitem__
    return self.build_lazy(super().__getitem__(key))
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmengine/config/config.py", line 103, in __missing__
    raise KeyError(name)
KeyError: 'model'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/im/myppal/main.py", line 20, in <module>
    model = init_detector(CONFIG_PATH, MODEL_PATH, device=DEVICE)
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmdet/apis/inference.py", line 59, in init_detector
    elif 'init_cfg' in config.model.backbone:
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmengine/config/config.py", line 1493, in __getattr__
    return getattr(self._cfg_dict, name)
  File "/home/im/anaconda3/envs/myppal/lib/python3.10/site-packages/mmengine/config/config.py", line 111, in __getattr__
    raise AttributeError(f"'{self.__class__.__name__}' object has no "
AttributeError: 'ConfigDict' object has no attribute 'model'
(myppal) im@IM-legion:~/myppal$
```

Figure 14. Errors while running PPAL after fixing initial problems.

In my notes from these attempts, I noted an MMDetection GitHub issue (https://github.com/open-mmlab/mmdetection/issues/10127) with discussion about this attribute error, with no conclusion or solution, that would help me run the project without updating the MMDetection library itself, as that was rendering PPAL project obsolete and still unusable.

After more digging, I noted: "There is no solutions. It turns out that the officially maintained ResNet model and config files are outdated and incompatible". It was a mess of co-dependencies, where the new library versions were released, without making sure that all the config files and models are compatible with it. The issue was even harder to debug in a bigger project like this with a hard-to-follow command order execution stack.

Ultimately, I spent a whole month on almost every day work trying to figure it out. There is barely any library versions configuration I have not tried. The project did not run with all the libraries depreciated to the past versions, that were also picked by guessing, since authors did not include versions of their dependencies in the requirements.

As a last resort, I personally emailed the main author of the paper, who was the owner of the project on GitHub. That email has been attached on the bottom of this document as “Appendix B – Email sent to the main PPAL author”.

The email was met with silence, but, of course, it was worth giving a try.

My further plan was to rewrite the main parts of the original code using the much newer versions of the library MMDetection and others, that should all be compatible with each other. But a few hours of trying to run anything more complex than one demo without encountering any errors were enough to motivate me to start searching for alternatives.

The biggest competitor to MMDetection turned out to be Detectron2 [2] built and maintained by Meta (Facebook AI Research team rebranded to Fundamental AI Research team). Due to my willingness to explore new paths, I installed the library and tried running example code, which immediately ran without any problems.

On one hand, that was great news because I could start working on my code and build a project without hopelessly trying to make anything work. But on the other hand, it meant that I had to remake a whole project without much help, since I had to figure out most things on my own (learning resources were not impressive) and that meant much more work to be done than the already-demanding project has assumed.

1.4.2. Final approach

After many setbacks, I managed to build a whole project using Detectron2. I have not even taken inspiration from the PPAL project code, since my code was meant to be much simpler and smaller. I could not even work on the same standard datasets, since my available computation power enclosed in the laptop of mine was not capable of competing with 8 NVIDIA RTX 2080Ti GPUs [1].

My main guide was the PPAL paper itself, that included equations and descriptions of the algorithm, that I implemented as faithfully as I could, with a few minor differences due to either a lack of technical possibility or a lack of details in the paper. In such cases I filled the missing parts in my own, unique, creative ways.

2. Implementation

2.1. Project core – training a model

Core of the project, that had all its features built around it, inherently stems from how the Detectron2 library is meant to be used, which is shown in the main official tutorial in the form of a Google Colab Notebook [20].

This main code worked as a starting point for implementing my algorithm. I wanted to be able to train any model on any dataset without problems, and this is mainly what this part of the program does.

2.1.1. Main program

Main program, which is the only program that has to be ran to train a model, is fairly short, and thus, simple to follow and modify, without messing with the backstage calculations and the whole algorithm implemented in utils.py file.

Main program does the following things in a sequence:

1. Parsing all the arguments provided by the user
2. Determining how many samples will be needed for training each round and moving this number of them to a separate folder
3. Starting initial training on these images by treating them as a separate dataset
4. Calling a function to determine what images should be used for the next round, depending on the provided parameters
5. Moving selected images to a new folder
6. Training on these images
7. Repeat from step 4, until all training rounds are successfully completed

From what I know, that was the easiest way to organize the active learning because Detectron2's functions for registering datasets and executing training on them mostly require a folder name with the training data and with annotations. The documentation itself is fairly limited when it comes to severe interventions into the inner workings of a training process, such as training on selected images from a registered dataset.

So, to make things easier for myself and focus on the active learning algorithm, I simply copy the whole dataset and move selected images into new folders, that are being regularly deleted as the program progresses.

2.1.2. Detectron2 basic functions

In the final version the basic functions have been changed a lot, but they serve the same purpose as they did at the beginning, so I will briefly go over them.

From the main program a “train” function is called, which is responsible for calling all other basic functions, that are responsible for:

1. Reading annotations in the specified format and converting them into Detectron2’s standard format. For COCO dataset there are built-in functions, that could read COCO format annotations in 2 lines – each for training and validation dataset, but for YOLO format used in my project there is just a function that reads annotations line by line and returns required dictionaries.
2. Registering a dataset through Detectron2 API.
3. Getting a config file for the training itself that includes information like the score threshold for detecting an object, device used to perform calculations, path to a weights file and so on. This is the standard for of holding all parameters used during the training loop.
4. Training the model on selected datasets.
5. Saving the model weights and dataset metadata for running inference and/or evaluation later on.

With these functions working properly. I could successfully train a model.

There is also another program dedicated to running inference on saved models for visual inspection of the results, that was useful while debugging and working on the project, but it is not a necessary part of it.

2.1.3. Modularity with a config file

The whole program was built with modularity in mind. At the beginning the whole folder structure was hard-coded, as well as specific folder and file names. For ease of use and clearer code, I made a config file for the whole program, that also serves as a glossary and overview of all the parameters having to be provided by the user.

That was a way to make my algorithm concise, clear, and modular, with how easy it is to modify the parameters either in config file to change default values, or through manual passing just for specific training.

Thanks to this setup, everything should run as intended even without any additional parameters from the user, if everything else has been set up properly.

2.2. Uncertainty-based sampling

Uncertainty sampling plays a role in determining what samples are ambiguous and hard for our model to work on. This step is crucial, as it filters available samples, to only include challenging ones and consider it for training.

In the original paper, authors called this step Difficulty Calibrated Uncertainty Sampling (DCUS), that represents their unique approach of easily defining uncertainty measure based on both classification and localization. Their algorithm also calculates an exponential moving average of uncertainty of specific classes, which allowed them to downweigh the uncertainty of images from a relatively certain classes. As a result, the training will focus much more on harder classes, and possibly eliminate the problem of unbalanced class representation in the training dataset.

My solution had to be different, since I did not manage to extract localization information from the model's output, so I had to base my calculations solely out of curated and more abstract model outputs: bounding boxes, predicted classes and scores (probability/certainty).

First step was to calculate the uncertainty of every image output.

2.2.1. Image uncertainty – Entropy calculation

Just like the authors of PPAL paper, I based the image-wise uncertainty calculation on Shannon Entropy [21], given by the equation below, that depends on the probability of multiple events.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \cdot log\, p(x_i) \tag{1}$$

Where: H – entropy, X – A list of events, n – event number, p – probability, x – individual event

So, I adapted the equation to calculate the sum of entropy of every detected object, weighted by the class uncertainty for that class.

$$UN(I) = -\left(\sum_{i=1}^{n} s_{o_{ij}} \cdot log\, s_{o_{ij}} \cdot classUN_{c_{o_i}}\right) \cdot \frac{1}{classUN_{MAX}} \tag{2}$$

Where: UN – uncertainty, I – image, n – number of detected objects, o – detected object, s – score, classUN – class uncertainty, c – class, $\mathrm{classUN_{MAX}}$ – the highest class uncertainty.

Which means that the final entropy of an image is an entropy of each detection weighted by the given detection's class uncertainty summed up. At the end, the whole sum is divided by the uncertainty of the most uncertain class, so that the entropy of an object belonging to that class remains unchanged and only the more certain classes will have lower probability to be picked for training.

2.2.2. Class uncertainty

For class uncertainty calculation I used a method that is quite straightforward and much simpler than using exponential moving average used by the authors of PPAL algorithm.

Conveniently I used the loop that does inference and calculates uncertainty for every image, to make use of already calculated entropy of an object detection. It works in the way that for every object detected in every image, besides adding the detection's entropy to total image entropy, I also add it to that class's entropy.

$$UN(c) = -\sum_{j=1}^{n}\left(\sum_{i=1}^{n} s_{o_{ij}} \cdot log\, s_{o_{ij}}, \qquad if\ c_{o_{ij}} = c\right) \tag{3}$$

Where: UN – uncertainty, c – class, m – total number of images, n – number of detections in an image, s – score, o_{ij} – specific object detected in a specific image.

This method might be much simpler, but it still captures the essence of method from PPAL algorithm, and it fulfills its goal on making more uncertain classes be picked with a higher probability.

2.2.3. Probability based uncertainty selection

After calculating all the class and image uncertainties and scaling them down to the range $(0; 1)$, it is time for final selection of these images.

At the earlier stages of my project, I was simply sorting the images by the final, class weighted uncertainty and picking the most uncertain images for further steps.

However, I upgraded the algorithm to pick final images based on the probability, if we treat the uncertainty as probability, so high uncertainty means a high chance of being chosen by the sampler.

This approach adds a level of resiliency to the algorithm, so that a small fraction of images are also selected from other parts of the uncertainty distribution.

2.3. Diversity-based sampling

Diversity sampler is a second step of our active learning algorithm, which takes the output of uncertainty sampler and outputs final images, that will be used for training. It ensures that the images we train are dissimilar from each other, hopefully preventing the model weights from getting stuck in local optima and exploring wider variety of parameter space.

The solution proposed in the paper introducing PPAL is called Category Conditioned Matching Similarity (CCMS) and it is based on the principle that “similarity of two multi-instance images can be computed by measuring how similar the objects they contain are” [1], which seems to be a better approach for multi-instance images than using global similarity, which works much better on single-object images, and which is computed using averaged convolutional feature maps.

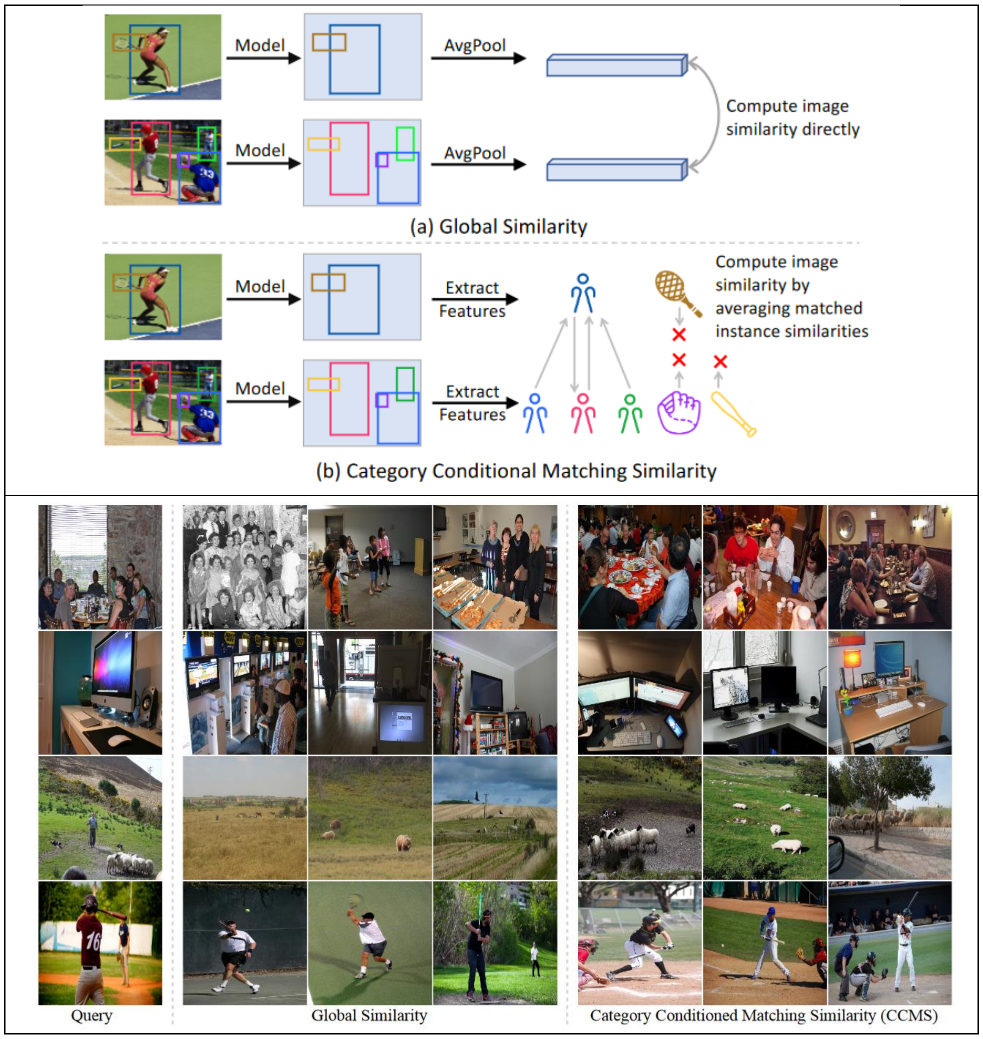

Model
AvgPool
Compute image similarity directly
(a) Global Similarity
Extract Features
Compute image similarity by averaging matched instance similarities
(b) Category Conditional Matching Similarity
Query
Global Similarity
Category Conditioned Matching Similarity (CCMS)

Figure 15. CCMS comparison to Global Similarity from the PPAL publication [1]. At the top, diagram shows differences in computing these similarities, at the bottom results are shown for the most similar matches to the query image from COCO mini-val dataset using RetinaNet.

The innovation of this method and its comparison to previous works are especially important because I managed to replicate this method quite faithfully, with only minimal invention from my side.

2.3.1. Cosine Similarity

Before we get to computing similarity between 2 images, we first need to know how to compute similarity between 2 objects detected on those images. And by objects I mean a bounding box, a vector of 4 values.

One of the most popular and easy methods for computing similarity between 2 vectors is a cosine similarity, given by the equation below.

$$cosine\ similarity = S_C(A,B) = cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4}$$

Where: A – first vector, B – second vector, n – dimensionality of those vectors

2.3.2. CCMS

Then, we follow the CCMS algorithm described by PPAL authors, that to find similarity between images A and B, we should:

1. Find the most similar object (of the same class) in image B, for every object in image A. Similarity is equal to 0 if there is no objects of the same class in the other image.
2. Multiply those maximal similarities by the score of the specific object in image A.
3. Sum up all those weighted maximal similarities.
4. Divide it all by the sum of scores in image A.
5. Now we get a final similarity between image B with respect to A, but we also want to ensure symmetry of our final similarity (to construct a distance matrix, that has to be symmetric), so we calculate the same similarity, but this time of image A with respect to B.
6. Average AB and BA similarity.

The same process can be described with equations 5 and 6.

$$CCMS_{AB}(I_A, I_B) = \frac{1}{\sum_{i=1}^{n} s_{o_{ij}}} \sum_{i=1}^{n} s_{o_{ij}} \cdot \begin{cases} max(S_C) + 1, from\ all\ (o_i, o_j)\ pairs, when\ c_{o_i} = c_{o_j}\ for\ j = 1..m \\ 0, \quad if\ no\ c_{o_i} = c_{o_j}\ for\ j = 1..m \end{cases} \tag{5}$$

$$CCMS(I_A, I_B) = \left(CCMS_{AB}(I_A, I_B) + CCMS_{BA}(I_B, I_A)\right) \cdot \frac{1}{2} \tag{6}$$

Where: I_A and I_B – images A and B respectively, s – score, o – detected object, n – total number of detections in the image A, m – total number of detections in the image B, S_C – cosine similarity

And with that, we can calculate the whole similarity matrix – meaning a matrix showing similarity of every image to every other image.

2.3.3. Distance matrix

After obtaining similarity matrix, in further steps we will want to use clustering algorithms to determine which objects are dissimilar from each other and thus, diverse. But for that, we need to obtain distance matrix, where distance is the reciprocal of similarity.

$$Distance(I_A, I_B) = \frac{1}{CCMS(I_A, I_B) + 0.2} \tag{7}$$

I added 0.2 constant for every similarity value to avoid division by zero, which results simply in the most dissimilar images in my dataset being at a distance of 5 units. The specific scaling should not influence the final results in a considerable way.

It is also worth nothing here, that due to the PPAL source code project [19] being quite complex, I did not manage to find exactly how they constructed their distance matrix. All I had was similarity equations from the paper itself [1] and the clustering algorithms used in diversity sampler, that loaded the distance matrix from a .npy file from working directory but could not find when and where it is computed or saved.

2.3.4. Speeding up the compute

The main problem with calculating my distance matrix was the time needed to compute it. I could not speed up the inference to get the model outputs faster, but I could definitely speed up processing of those outputs.

Initially, my program was calculating 4 distance values a second. With my test dataset of 6000 images, roughly 4800 of them left after preparatory round, the first round of calculating distance matrix (4800x4800 matrix), would take a lot of time, that has been calculated in equation 8.

$$\frac{4800 \cdot 4800\ values}{4\frac{values}{second}} = \ 23\ 040\ 000\ seconds = 244\ days\ of\ nonstop\ calculation \tag{8}$$

My first step was increasing the score threshold to 0.3, from 0.05 used by the authors of *Plug and Play Active learning for Object Detection* paper [1]. This resulted in at least 4x decrease in number of detected objects, which took the time needed to train a model from many months down to 17h. Which was a good sign, but with how many experiments I wanted to conduct, it was still worth trying to make it faster.

The next step was implementing parallel processing into my program. When I think of it, I wonder if there was a way to make it even faster through computations on a GPU, but a default device for computing distance matrix was a CPU, so I just wanted to utilize all CPU cores that I had available.

It turned out to be quite problematic. I used multiprocessing python library, where a lot of simple methods failed. I tried creating a pool of threads that received a copy of outputs of inference, but it turned out to freeze my program due to creating thousands of copies of huge text variable. Then, I tried using a manager with shared memory, so I could pass each thread just a handle to outputs instead of creating a copy of it. But that also did not work due to errors like "Too many open files error" and many more, even though that script did not open any files at a high level. It probably had to do with inner workings of shared memory buffer.

Ultimately, I managed to make it work by using torch.multiprocessing wrapper for multiprocessing library, that was meant for utilizing multiprocessing with tensors, but I only used it to set sharing strategy to "file_system" (unavailable without a wrapper), which fixed the "Too many open files" error, at a cost of possible memory leaks.

All in all, it worked, and my program could utilize 80% of my CPU instead of 10%, which sped up the program 5 times, that took the whole active learning of a model from 17h down to around 3h. That enabled me to leave the experiments running during the night and checking the results in the morning.

2.3.5. Clustering

Clustering part is responsible for taking the distance matrix of all images taken into consideration and picking the specified number of diverse images for further training.

This part was the only part of the original code for PPAL paper authors, that I could directly copy, since the code did not include any references to MMDetection library or other parts of the code. It simply took the distance matrix as an input alongside the desired number of images to be picked.

The whole algorithm is based on k-means++ algorithm initialized by k-Center-Greedy algorithm, that aims at finding diverse samples, that are also representative of a dataset. Proposed solution is a 2-opt solution, which is sufficient for our needs.

K-means is an iterative algorithm, that initializes imperfect centroids, that iteratively move towards the center of gravity of the datapoints they are close to.

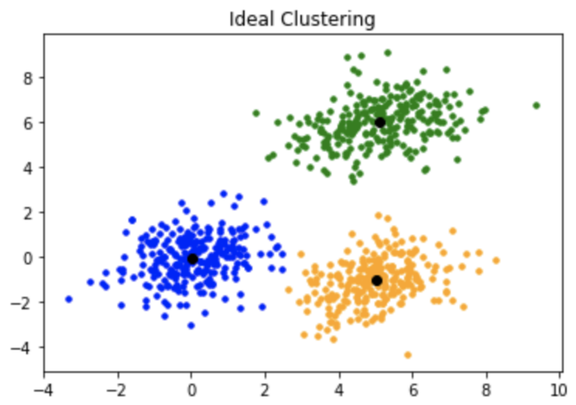


Figure 16. Visualization of k-means++ clustering with 3 centroids on 2-dimensional data [22].

Simplifying, we can say that this algorithm does exactly what is depicted on the Figure 16, but in a much more dimensional space with around 5000 images classified into 1200 clusters.

2.4. Ensuring reproducibility

After first completing the algorithm, I quickly ran a first experiment to see if everything is working as intended. My first result has been shown on Figure 17.

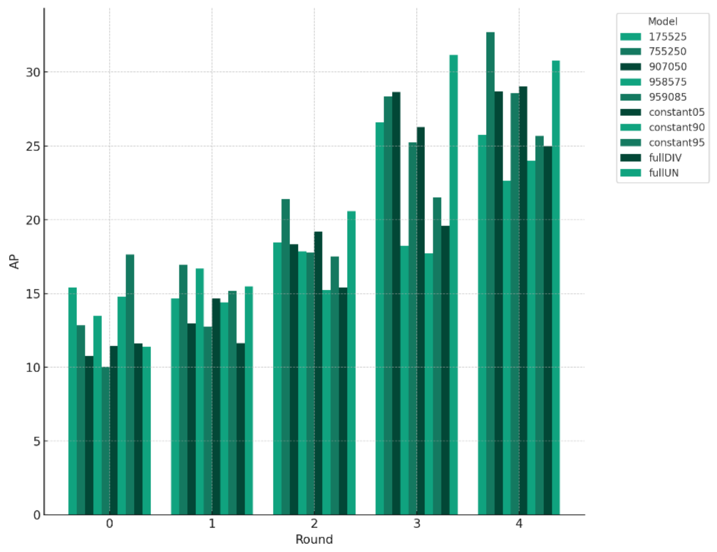


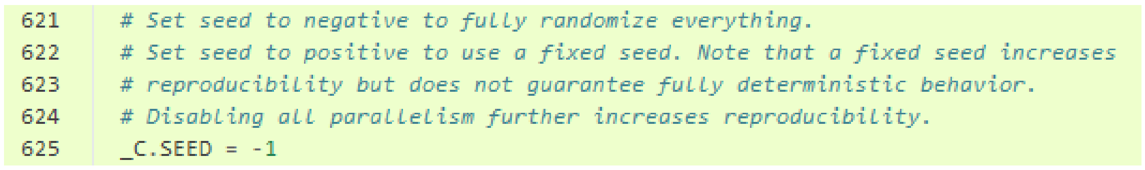
Figure 17. First full-scale experiment results.

The specific parameters and setup are not important (and the repeated colors for different models), but I was comparing models with varying split of images picked by uncertainty and diversity, and one thing immediately struck me.

As described earlier, initial round 0 was trained on the same exact images, starting from the exact same model weights. Despite this, the difference in Average Precision varied from 10 to around 17.5. That was a large gap, that did not meet my expectations. They all should have the same scores, the same weights. Round 0 was just a setup. Without this working properly, I cannot think about directly comparing the models.

It turned out that PPAL paper authors did not ensure the deterministic behavior of their experiments, but they ran the experiment 3 times and averaged the results to minimize the random effect. But I was not satisfied with this solution, and I made a quick research on it.

I found a message in Detectron2 documentations regarding a config file's "SEED" setting to increase reproducibility of experiments, shown on Figure 18.



```
# Set seed to negative to fully randomize everything.
# Set seed to positive to use a fixed seed. Note that a fixed seed increases
# reproducibility but does not guarantee fully deterministic behavior.
# Disabling all parallelism further increases reproducibility.
_C.SEED = -1
```

Figure 18. Part of detectron2.config documentation mentioning "SEED".

To stay safe and to not run many-hours-long experiments in vain, I also used a precautionary step found on the internet [23] of setting all other possible seeds, that could be cause of the problems, in the form of function shown on as Code 1.

```python
def seedEverything(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    print("Set all the seeds")
```

Code 1. Function setting all the seeds beside detectron2.config seed.

With that all set, I ran the experiment again, this time obtaining the result shown on Figure 19.

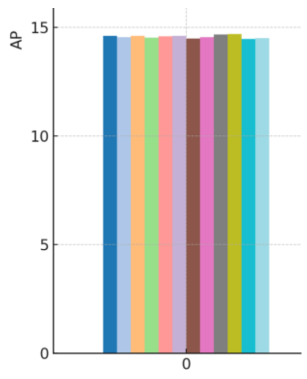


Figure 19. Results for round 0 of various models after ensuring reproducibility of the experiment.

Which completely satisfied my expectations, I could not expect anything more. As written in the documentation, what I showed on Figure 18, the authors do not guarantee deterministic behavior, but merely a reproducibility. The initial variation changed from the range $\langle 10; 17.5 \rangle$ to $\langle 14.48; 14.71 \rangle$.

With that, I considered the problem solved.

2.5. Final pipeline

My final pipeline is exactly the same as the one shown on Figure 7, except the exact numbers of how these 2 methods are stringed together. The main idea is that from the whole available pool – from “totalCount” we want to pick a “targetCount” of images.

So, I stringed the methods in a way, that after the first filtering – uncertainty-based sampling – there are “countOfImagesForDiversity” images left, where:

$$countOfImagesForDiversity = (totalCount - targetCount) \cdot (1 - UDsplit) + targetCount \tag{9}$$

$$targetCount \leq countOfImagesForDiversity \leq totalCount \tag{10}$$

Where UDsplit is the main program parameter passed by the user that determines how many images should be picked by Uncertainty. As equation 9 shows: when UDsplit is equal to 1, then countOfImagesForDiversity is exactly equal to targetCount, which means that diversity sampling has to pick for example 1200 images out of 1200 possible images, which is pointless to even compute, so program skips that and passes the images further for training. As a consequence, we can say that all images were picked solely by uncertainty.

On the other hand, when the UDsplit parameter is equal to 0, then countOfImagesForDiversity will be equal to totalCount of images, which means that out of the pool of 4800 images, 4800 of them have been passed for diversity sampling step, which means that none of the samples were picked by uncertainty, and everything will be determined by diversity sampling.

This simple system is responsible for chaining these 2 methods through using UDsplit, that can have any value between 0 and 1. UDsplit of 0.5 simply means a half of the difference between total and target image count will be discarded from the pool by uncertainty, and other half by diversity, down to a final target count. The whole process has been shown on Figure 20 for a simplified view of how UDsplit affects what sampling method picks final samples for training.

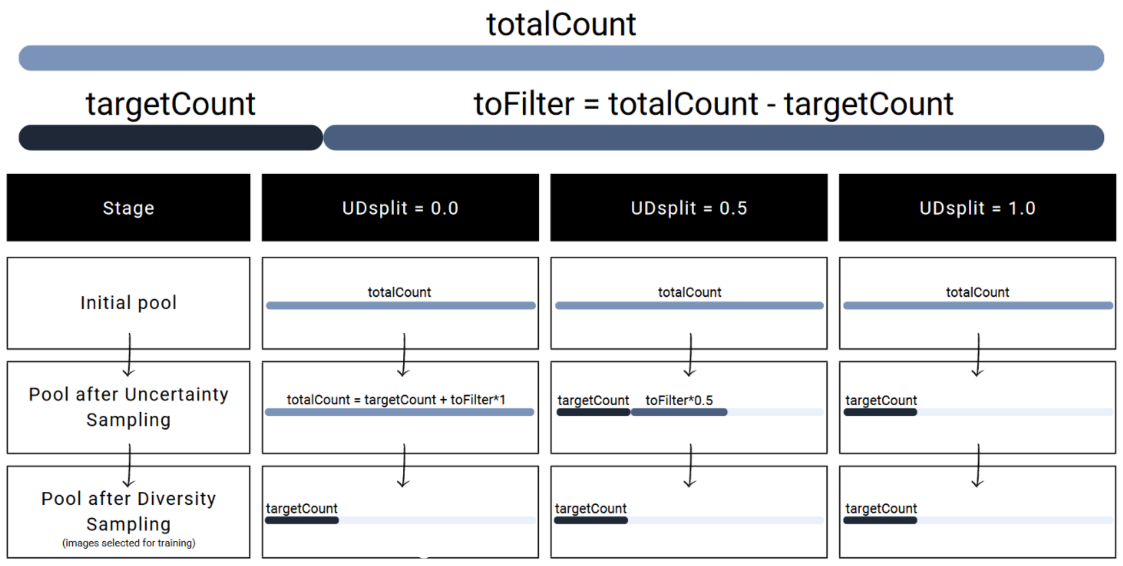


Figure 20. Simplified table of how many images are filtered out at each program stage depending on UDsplit value.

3. Experiments

For a final test, to get a sense of how models perform with different proportions of images picked by uncertainty and diversity (please refer to the chapter “.

2.5. Final pipeline” for more details), I tested various models with UDsplit values staying constant, decreasing with time, and increasing.

3.1. Experimental setup

I was running all the experiments on Faster R-CNN R50-C4 model with 3x learning rate schedule, from the detectron2 model zoo. For all tests I used a dataset called “CSGO TRAIN YOLO V5” [24], from Roboflow Universe service for sharing datasets.

To determine how many iterations to train the model for, I ran one training round with 10k iterations and checked the graph of validation and train losses.

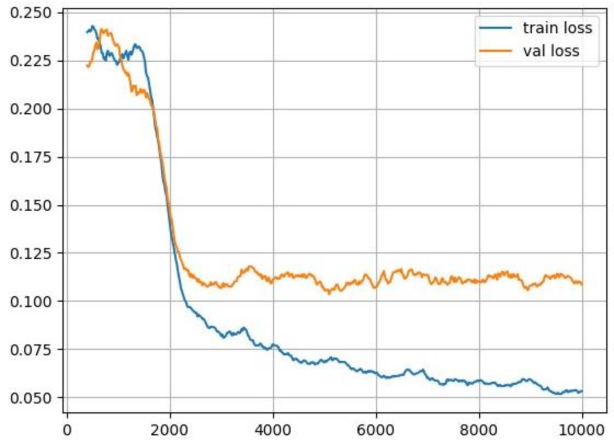


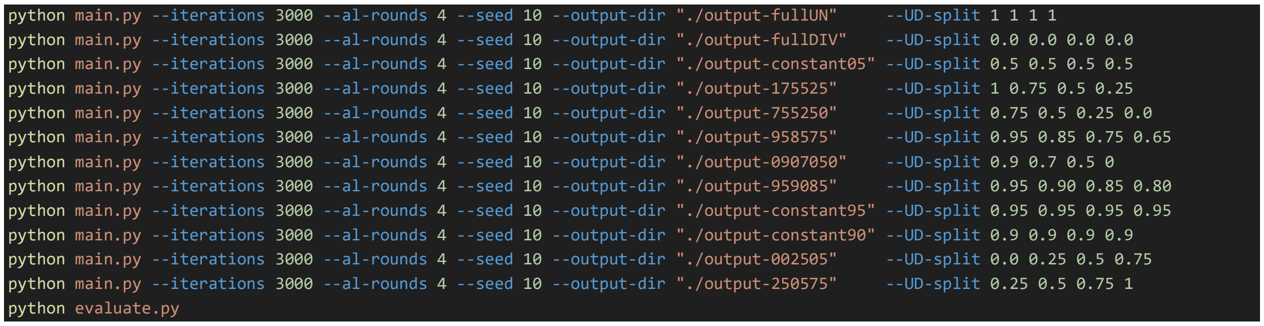
Figure 21. Train and Validation losses plot, that I used to pick the right number of iterations. Validation loss stops going down around iteration 3000, which means that that is the point where the model starts overfitting. All further models were trained with 3000 iteration.

And from the characteristic shown Figure 21, I picked 3000 iterations as a number that would make the model perform decently well, while avoiding overfitting. This number worked only with the specific learning rate I picked arbitrarily: 1e-4.

I picked to train all the models on 5 rounds – 4 active + 1 initial round. More rounds meant more time spent picking the right images, and I wanted to roughly match the experiments that authors of PPAL [1] made, so I picked the lowest number of rounds, that they tested.

Then, final parameter worth mentioning is the array of UDsplit variable, that tells the program what UDsplit value to use for each round.

Final experiment that I ran was the following code shown below:



```
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-fullUN"     --UD-split 1 1 1 1
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-fullDIV"    --UD-split 0.0 0.0 0.0 0.0
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-constant05" --UD-split 0.5 0.5 0.5 0.5
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-175525"     --UD-split 1 0.75 0.5 0.25
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-755250"     --UD-split 0.75 0.5 0.25 0.0
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-958575"     --UD-split 0.95 0.85 0.75 0.65
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-0907050"    --UD-split 0.9 0.7 0.5 0
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-959085"     --UD-split 0.95 0.90 0.85 0.80
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-constant95" --UD-split 0.95 0.95 0.95 0.95
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-constant90" --UD-split 0.9 0.9 0.9 0.9
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-002505"     --UD-split 0.0 0.25 0.5 0.75
python main.py --iterations 3000 --al-rounds 4 --seed 10 --output-dir "./output-250575"     --UD-split 0.25 0.5 0.75 1
python evaluate.py
```

Code 2. Final experiment ran with varying UDsplit variable values. I provided 4 values of UDsplit for each active learning round, but it was only a safety measure because with everything working as intended, the last round has no sample pool to choose from and algorithm simply picks what images are left.

3.2. Results

I ran the experiment described in the previous chapter “3.1. Experimental setup” with seed equal to 10, and then reran the experiment with seed equal to 11 and averaged these 2 results to decrease influence of random factors. The trends and final results were quite similar, so it did not drastically change the observed patterns.

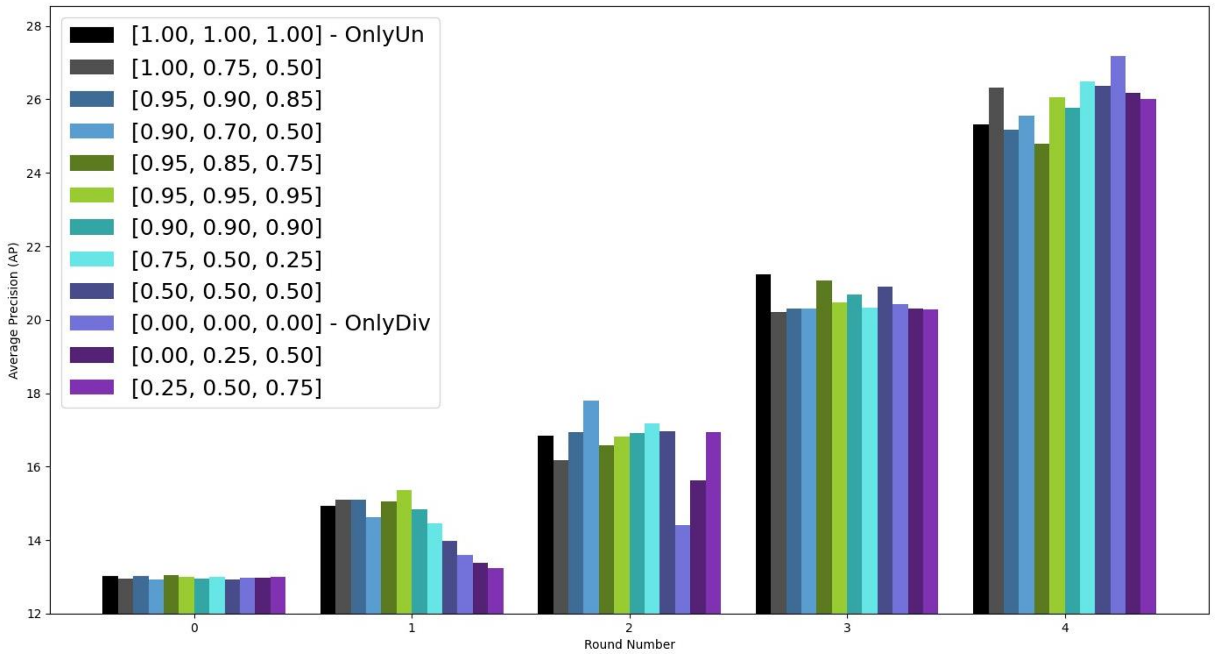


Figure 22. Comparison of Average Precision of different models across 5 rounds of active learning. Names of the models indicate what UDsplit values have been used for each round, meaning the fraction of images for training picked with uncertainty sampling. The rest have been picked by diversity sampling. Diversity oriented models are grouped on the right side for readability. All models trained on exactly the same data – the only difference was the order of training.

Figure 22 shows the Average Precision of each tested model, but Detectron2’s evaluator also provides more metrics like APs, APm and APl indicating objects of small, medium, and large size and average precision among them, that I graphed on the Figure 23.

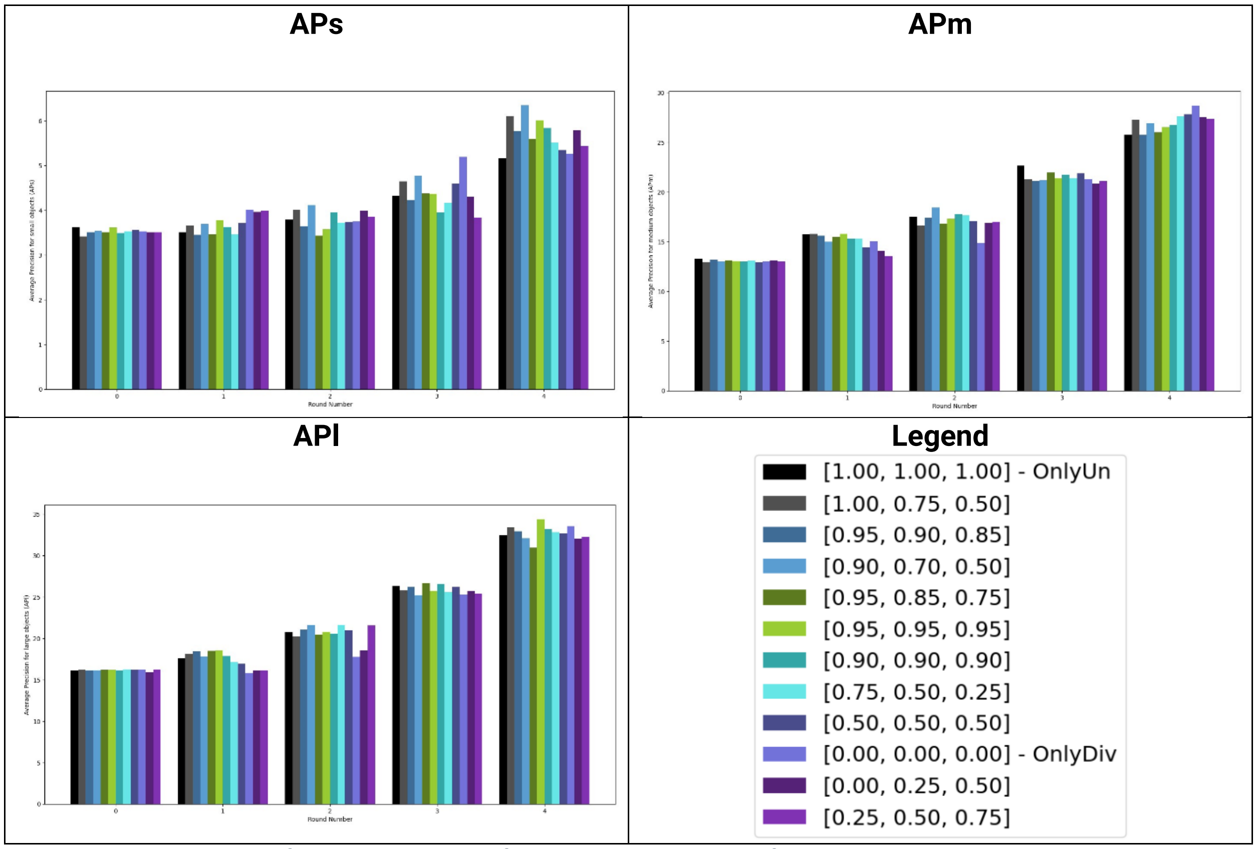


Figure 23. Comparison of average precision after every training round of all models across small, medium, and large objects.

All numeric results have been provided in “Appendix A – Numerical results of the main experiment”, that also include a Random Sampling reference to see how much better active learning is compared to it, but it turned out to overfit at the 4th round and deteriorate to lower Average Precision than after initial round, so I excluded this model for more pleasant axes’ scaling on the graphs. It also could be related to the issues with a dataset that I am using, but I did not manage to investigate the issue and understand its root causes.

4. Conclusions

4.1. Summary

My research fulfilled its purpose of shedding more light on the algorithm introduced in the Plug and Play Active Learning For Object Detection [1] paper. There are quite a few patterns that seem to be universally true, even in wider contexts than merely training an object detection model.

1. Diversity sampling is a key for all active learning methods – it enables the model to explore more solutions and gravitate more towards a global optimum, even if it takes more time. Generally, in the long-term, diversity beats competing active learning methods.

2. Uncertainty-sampling might be thought of as a greedy algorithm prioritizing getting the task done over finding global optimum. It performs well in the short term, which can cause it to overfit the data quicker.

3. Uncertainty-based methods are not needed for effective active learning and might be completely omitted in specific scenarios. Their usefulness comes mostly from how easy it is to compute uncertainty (especially compared to diversity), which can be used to propose candidate pool of challenging samples for diversity-sampling, making the whole system diversity-oriented, but also compute-efficient.

4. Diversity-oriented machine learning is an equivalent of interleaved learning in education, where the "agents" underperform at the beginning and they are not certain of their abilities, but the strategy beats competitors in the long term in all kinds of tests.

The same points with more explanation have been included in the next chapter: 4.2. Detailed conclusions.

However, it is worth keeping in mind that my tests were only performed on a single dataset of around 6500 images, with 2 different random set seeds. My research is a pointer of what might be interesting to explore to advance Artificial Intelligence in the direction of data-efficiency. I would be glad if more people got invested in the topic and confirmed/refined/denied my claims.

4.2. Detailed conclusions

1. Diversity sampling is a key for all active learning methods – it enables the model to explore more solutions and gravitate more towards a global optimum, even if it takes more time. Generally, in the long-term, diversity beats competing active learning methods.

Observing the main experiment results (Figure 22) there is a clear pattern of more diversity oriented models (5 rightmost bars) collectively outcompeted more uncertainty-focused models, while collectively underperforming in the first 2 rounds of active training. I trained the model for 3000 iterations, which is around the moment, where first models might start to overfit. It is an indication that diversity trained models are more resilient to falling into local optima and finding better parameters in the long term.

The best result has been achieved by diversity-only-trained model, which is a clear indicator of how important it is to ensure diversity during training.

2. Uncertainty-sampling might be thought of as a greedy algorithm prioritizing getting the task done over finding global optimum. It performs well in the short term, which can cause it to overfit the data quicker.

After rounds 1 and 2 uncertainty-leaning models clearly perform much better, and even after round 3 most models are relatively equal, instead of only-uncertainty trained model, who positively stood out.

However, diversity trained models by the end of the whole training surpassed on-average even the only-uncertainty trained model.

This is the best explanation I can give, as far as I intuitively understand the topic, that diversity stimulates exploring more possible parameter paths and it eventually will perform better most of the time. Uncertainty training focuses on being certain, leading to prioritizing short over long term.

3. Uncertainty-based methods are not needed for effective active learning and might be completely omitted in specific scenarios. Their usefulness comes mostly from how easy it is to compute uncertainty (especially compared to diversity), which can be used to propose candidate pool of challenging samples for diversity-sampling, making the whole system diversity-oriented, but also compute-efficient.

Model trained only on diverse images, with no uncertainty-sampling involved, achieved the highest average precision, which makes us think more strategically about what role does uncertainty play, and about why and when to apply it.

Uncertainty-sampling seems to have its specific use-cases, mostly thanks to how much easier it is to calculate data sample's uncertainty than it is to calculate diversity with respect to every other image (distance matrix computation is the single longest task while running this algorithm, at least for now, without utilization of a GPU and more advanced optimizations).

But the claim remains, that if finding optimal solution regardless of the cost is concerned, diversity is all you need - paraphrasing the classic line.

4. Diversity-oriented machine learning is an equivalent of interleaved learning in education, where the "agents" underperform at the beginning and they are not certain of their abilities, but the strategy beats competitors in the long term in all kinds of tests.

The performance of diversity-trained models matches the characteristic of performance of "agents" after interleaved studying, shown on Figure 2, where diversity learners seem to fall behind at the beginning, but eventually catch up and surpass competitors. Model scores presented on Figure 22 have been grouped to the right, so it is easier to see how they fall behind initially, but the model trained solely on diverse data have beat every other at the end.

The analogy is not perfect since we do not have a direct comparison to blocked studying. It would probably look similar to training an object detection model of images focusing on one object class at a time and sequentially going through all the classes. There is no surprise that it is hard to find information about such experiments, because of how ineffective it sounds as a learning method. And it is exactly how most schools around the world organize their education process.

Bibliography

[1] C. Yang, L. Huang and E. J. Crowley, *Plug and Play Active Learning for Object Detection,* 2022.

[2] Meta (Facebook AI Research), *Detectron2 detection and segmentation algorithms library.*

[3] OpenMMLab, *MMDetection library source code,* GitHub.

[4] R. Kotarski, *Hack into the brain (Original in polish: Włam się do mózgu),* 2017.

[5] Wikipedia, *Article about Muhammad Ali.*

[6] C. M. M. S. Yana Weinstein, *Teaching the science of learning,* 2018.

[7] R. B. Steven Pan, *Acquiring an Accurate Mental Model of Human Learning: Towards an Owner's Manual (Contributed as Chapter 11.3 for Oxford Handbook of Memory, Vol. II: Applications),* 2020.

[8] Wikipedia, *Article about semi-supervised learning.*

[9] Wikipedia, *Article about Yann LeCun.*

[10] Y. LeCun, *Post on X about efficiency of training LLMs,* 2023.

[11] Y. LeCun, *Post on X about betting on new architectures,* 2023.

[12] G. Verdon, *Post on X about next steps for AI after compressing the internet,* 2023.

[13] Wikipedia, *Article about Active learning in machine learning.*

[14] Google, *What is CAPTCHA?.*

[15] R. Whitwam, *Google's Street View neural network can now decrypt captchas better than a human,* ExtremeTech, 2014.

[16] Google, *Multi-digit Number Recognition from Street View,* 2014.

[17] R. Monarch, *Human-in-the-Loop Machine Learning (Free online fragment),* Manning, 2021.

[18] R. Monarch, *Diversity Sampling Cheatsheet pdf.*

[19] C. Yang, L. Huang and E. J. Crowley, *Code implementation of Plug and Play Active Learning for Object Detection,* 2022.

[20] Meta, *Detectron2 Tutorial Colab Notebook.*

[21] Wikipedia, *Article about Entropy in information theory.*

[22] savyakhosla, *ML | K-means++ Algorithm,* Geeks for Geeks.

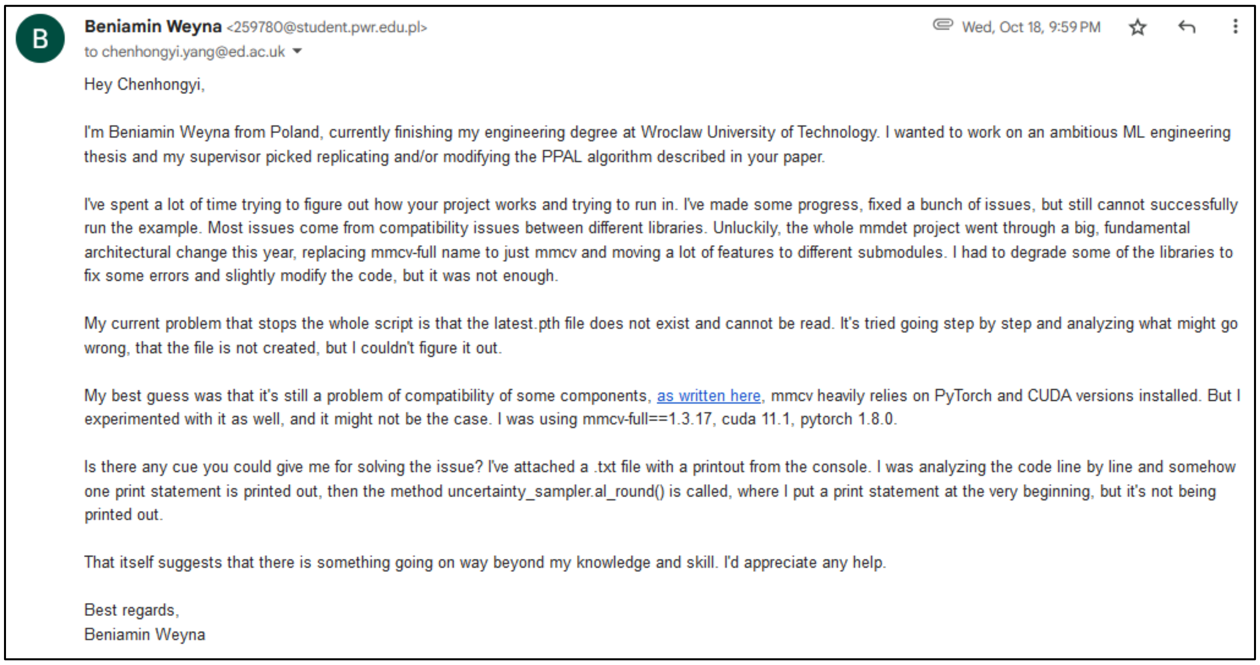
[23] belskikh, *How to properly fix random seed with pytorch lightning? - issue #1565,* GitHub.

[24] new-workspace-rp0z0, *CSGO TRAIN YOLO V5 Dataset,* Roboflow Universe, 2022.

Appendix A – Numerical results of the main experiment

		Seed = 10						Seed = 11					
model	**round**	**AP**	**AP50**	**AP75**	**APs**	**APm**	**APl**	**AP**	**AP50**	**AP75**	**APs**	**APm**	APl
[1.00, 1.00, 1.00]	**0**	14.52	35.07	9.32	4.95	15.10	17.20	11.54	32.90	2.89	2.31	11.42	15.09
[1.00, 1.00, 1.00]	**1**	18.33	43.29	13.34	5.07	19.88	20.48	11.53	36.78	2.06	1.96	11.58	14.81
[1.00, 1.00, 1.00]	**2**	20.06	49.36	10.27	4.76	21.24	24.42	13.64	43.75	1.20	2.82	13.83	17.22
[1.00, 1.00, 1.00]	**3**	22.52	56.07	8.65	4.51	24.14	27.38	19.95	55.50	6.91	4.13	21.19	25.29
[1.00, 1.00, 1.00]	**4**	25.36	55.76	17.11	5.05	25.71	32.63	25.26	55.83	16.58	5.29	25.75	32.27
[1.00, 0.75, 0.50]	**0**	14.62	34.82	9.86	4.54	15.05	17.54	11.29	32.66	2.71	2.30	10.88	14.88
[1.00, 0.75, 0.50]	**1**	18.05	43.40	12.66	5.06	19.16	20.75	12.18	38.13	2.02	2.25	12.38	15.52
[1.00, 0.75, 0.50]	**2**	18.46	48.86	7.36	4.53	18.74	22.94	13.88	44.47	1.55	3.48	14.46	17.49
[1.00, 0.75, 0.50]	**3**	21.02	54.46	9.03	4.53	22.12	26.49	19.41	54.05	5.28	4.75	20.42	25.07
[1.00, 0.75, 0.50]	**4**	26.86	57.31	20.78	5.88	27.30	33.89	25.76	58.68	15.00	6.34	27.28	32.89
[0.90, 0.70, 0.50]	**0**	14.60	35.01	9.47	4.79	15.11	17.40	11.26	32.85	2.78	2.29	10.91	14.94
[0.90, 0.70, 0.50]	**1**	17.35	42.57	11.01	5.04	17.97	20.40	11.89	37.27	1.63	2.35	12.02	15.29
[0.90, 0.70, 0.50]	**2**	21.96	52.04	11.89	5.28	22.89	26.24	13.66	42.79	1.62	2.96	13.98	16.98
[0.90, 0.70, 0.50]	**3**	21.95	55.91	10.23	4.73	22.70	27.25	18.66	53.73	4.95	4.81	19.67	23.17
[0.90, 0.70, 0.50]	**4**	26.46	57.01	18.26	6.15	27.78	32.51	24.66	58.74	12.87	6.54	26.03	31.69
[0.75, 0.50, 0.25]	**0**	14.54	34.51	9.69	4.78	15.01	17.35	11.46	33.05	2.83	2.27	11.16	15.14
[0.75, 0.50, 0.25]	**1**	17.21	42.69	10.58	4.77	18.47	19.64	11.73	36.91	1.86	2.16	12.15	14.70
[0.75, 0.50, 0.25]	**2**	20.12	50.20	9.18	4.54	21.31	24.72	14.23	43.82	1.49	2.91	14.02	18.53
[0.75, 0.50, 0.25]	**3**	20.91	53.23	8.65	3.99	22.13	25.87	19.75	54.34	6.75	4.34	20.63	25.29
[0.75, 0.50, 0.25]	**4**	27.32	58.12	18.69	6.08	28.54	33.46	25.64	57.64	14.65	4.94	26.70	32.23
[0.00, 0.00, 0.00]	**0**	14.48	34.36	9.66	4.78	15.02	17.31	11.48	32.98	3.05	2.28	11.03	15.21
[0.00, 0.00, 0.00]	**1**	15.88	41.96	9.36	5.16	17.99	17.68	11.32	37.23	1.81	2.86	12.12	14.04
[0.00, 0.00, 0.00]	**2**	16.42	44.72	5.64	4.90	17.15	19.83	12.39	41.46	1.11	2.60	12.58	15.74
[0.00, 0.00, 0.00]	**3**	22.61	54.51	11.69	5.73	23.70	27.18	18.25	52.52	4.11	4.66	18.90	23.33
[0.00, 0.00, 0.00]	**4**	27.08	58.19	18.97	4.80	28.99	33.22	27.29	60.64	18.69	5.73	28.42	33.89
[0.50, 0.50, 0.50]	**0**	14.54	34.79	9.58	4.82	14.98	17.36	11.31	32.58	2.86	2.30	10.86	15.04
[0.50, 0.50, 0.50]	**1**	16.40	42.04	9.64	5.14	16.96	19.36	11.58	38.24	1.61	2.28	11.90	14.55
[0.50, 0.50, 0.50]	**2**	20.44	49.95	10.74	4.67	20.92	24.71	13.47	43.69	1.29	2.81	13.23	17.29
[0.50, 0.50, 0.50]	**3**	23.14	55.93	10.88	4.81	23.92	28.58	18.67	52.60	4.71	4.39	19.81	23.81
[0.50, 0.50, 0.50]	**4**	27.49	58.16	21.03	5.20	28.64	34.03	25.26	57.08	14.99	5.49	27.02	31.30
[0.95, 0.90, 0.85]	**0**	14.50	34.64	9.52	4.77	15.09	17.21	11.56	33.24	3.02	2.25	11.30	15.14
[0.95, 0.90, 0.85]	**1**	18.20	43.74	13.34	5.01	19.19	21.03	12.01	38.20	2.45	1.90	12.06	15.83
[0.95, 0.90, 0.85]	**2**	20.36	51.14	9.50	4.76	21.35	24.44	13.53	43.21	1.45	2.53	13.49	17.69
[0.95, 0.90, 0.85]	**3**	21.37	55.44	8.20	4.46	21.84	27.48	19.26	54.07	6.08	4.01	20.35	24.92
[0.95, 0.90, 0.85]	**4**	25.74	56.30	17.85	5.87	25.70	33.63	24.60	58.20	13.52	5.68	25.82	32.22
[0.95, 0.85, 0.75]	**0**	14.62	34.80	9.88	4.75	15.04	17.45	11.50	32.81	3.09	2.28	11.26	14.96
[0.95, 0.85, 0.75]	**1**	17.82	42.68	12.79	4.90	18.66	20.82	12.31	37.95	2.15	2.03	12.31	16.21
[0.95, 0.85, 0.75]	**2**	19.18	48.63	8.63	4.40	19.44	23.57	13.97	43.02	1.69	2.46	14.23	17.42
[0.95, 0.85, 0.75]	**3**	21.92	56.26	9.77	4.29	22.84	27.05	20.21	54.82	7.31	4.47	21.10	26.35
[0.95, 0.85, 0.75]	**4**	26.41	57.45	18.74	5.10	27.30	32.69	23.17	55.60	11.69	6.10	24.74	29.23
[0.90, 0.90, 0.90]	**0**	14.67	34.85	9.95	4.70	15.05	17.66	11.23	32.67	2.76	2.28	10.99	14.66
[0.90, 0.90, 0.90]	**1**	17.93	43.23	12.01	5.03	18.43	20.64	11.76	37.65	1.85	2.23	12.15	15.10
[0.90, 0.90, 0.90]	**2**	20.01	50.13	10.09	4.67	20.97	24.55	13.81	44.16	1.52	3.23	14.62	16.67
[0.90, 0.90, 0.90]	**3**	22.24	54.80	10.50	3.98	23.03	28.64	19.15	53.78	6.31	3.92	20.42	24.56
[0.90, 0.90, 0.90]	**4**	26.15	56.42	18.17	5.83	26.81	33.59	25.40	57.87	14.46	5.84	26.66	32.79
[0.95, 0.95, 0.95]	**0**	14.71	35.15	9.76	4.93	15.15	17.62	11.27	32.68	2.78	2.30	10.96	14.75
[0.95, 0.95, 0.95]	**1**	18.45	44.08	13.22	5.08	19.26	21.17	12.29	38.47	2.49	2.48	12.29	15.91
[0.95, 0.95, 0.95]	**2**	19.32	49.88	8.35	4.59	20.25	23.70	14.30	43.22	1.57	2.56	14.34	17.93
[0.95, 0.95, 0.95]	**3**	22.14	57.06	8.83	4.63	22.82	27.41	18.83	53.52	4.93	4.09	19.99	24.10
[0.95, 0.95, 0.95]	**4**	26.98	57.47	19.82	5.80	26.87	35.79	25.14	58.96	13.56	6.22	26.18	33.01
[0.00, 0.25, 0.50]	**0**	14.55	34.93	9.53	4.75	15.03	17.11	11.39	32.69	2.70	2.27	11.23	14.78
[0.00, 0.25, 0.50]	**1**	15.32	40.55	9.65	4.77	16.76	17.51	11.43	37.46	1.67	3.15	11.42	14.82
[0.00, 0.25, 0.50]	**2**	18.06	47.24	9.26	4.91	19.53	21.01	13.21	42.67	1.29	3.08	14.22	16.15
[0.00, 0.25, 0.50]	**3**	22.64	56.14	9.66	4.50	23.11	28.11	17.95	52.80	4.33	4.10	18.55	23.42
[0.00, 0.25, 0.50]	**4**	26.63	59.47	16.97	4.90	27.41	33.09	25.75	60.46	14.46	6.68	27.62	31.05
[0.25, 0.5, 0.75]	**0**	14.63	34.89	9.70	4.77	14.97	17.63	11.36	32.53	2.94	2.25	11.02	14.89
[0.25, 0.5, 0.75]	**1**	14.54	40.54	5.79	5.18	14.64	17.57	11.96	39.77	1.60	2.80	12.46	14.72
[0.25, 0.5, 0.75]	**2**	20.01	49.26	10.78	5.06	20.70	25.09	13.88	44.19	1.50	2.65	13.34	18.08
[0.25, 0.5, 0.75]	**3**	21.21	54.91	7.90	4.04	22.42	25.56	19.33	54.19	5.12	3.64	19.78	25.27
[0.25, 0.5, 0.75]	**4**	26.47	57.56	18.23	5.24	27.45	32.25	25.57	58.02	15.48	5.62	27.24	32.28
Random Sampling	**0**	14.57	35.19	9.44	5.12	15.08	17.29	11.57	33.02	3.06	2.23	11.17	15.29
Random Sampling	**1**	16.21	38.64	11.63	4.84	16.32	20.19	13.19	32.57	7.15	3.01	12.69	18.47
Random Sampling	**2**	14.18	32.17	7.75	2.85	11.53	21.09	11.70	29.76	4.90	1.84	9.58	18.15
Random Sampling	**3**	25.07	53.51	18.31	6.59	26.98	30.23	21.47	49.81	14.17	4.34	24.58	25.95
Random Sampling	**4**	12.56	24.55	11.40	4.58	11.78	16.72	12.05	24.55	9.93	4.53	11.33	16.87

Appendix B – Email sent to the main PPAL author



Beniamin Weyna <259780@student.pwr.edu.pl>
to chenhongyi.yang@ed.ac.uk

Wed, Oct 18, 9:59 PM

Hey Chenhongyi,

I'm Beniamin Weyna from Poland, currently finishing my engineering degree at Wroclaw University of Technology. I wanted to work on an ambitious ML engineering thesis and my supervisor picked replicating and/or modifying the PPAL algorithm described in your paper.

I've spent a lot of time trying to figure out how your project works and trying to run in. I've made some progress, fixed a bunch of issues, but still cannot successfully run the example. Most issues come from compatibility issues between different libraries. Unluckily, the whole mmdet project went through a big, fundamental architectural change this year, replacing mmcv-full name to just mmcv and moving a lot of features to different submodules. I had to degrade some of the libraries to fix some errors and slightly modify the code, but it was not enough.

My current problem that stops the whole script is that the latest.pth file does not exist and cannot be read. It's tried going step by step and analyzing what might go wrong, that the file is not created, but I couldn't figure it out.

My best guess was that it's still a problem of compatibility of some components, as written here, mmcv heavily relies on PyTorch and CUDA versions installed. But I experimented with it as well, and it might not be the case. I was using mmcv-full==1.3.17, cuda 11.1, pytorch 1.8.0.

Is there any cue you could give me for solving the issue? I've attached a .txt file with a printout from the console. I was analyzing the code line by line and somehow one print statement is printed out, then the method uncertainty_sampler.al_round() is called, where I put a print statement at the very beginning, but it's not being printed out.

That itself suggests that there is something going on way beyond my knowledge and skill. I'd appreciate any help.

Best regards,
Beniamin Weyna