# Table Of Contents



# Replicating the Results of this Project

First, we would recommend using the environment.yml. If you have a CUDA enabled GPU, we strongly recommend installing PyTorch with CUDA. The experiment suite takes ~50 hours on an RTX 3090 with 24GB of VRAM using CUDA; no attempt has been made to run this on a CPU-only system.

To create the environment open the terminal on a Unix-like system;

```
conda env create -f environment.yml
```

Then run,

```
python run_experiments.py
```

This file has been set up to run all the experiments we created.

# Logging into Weights and Biases

To run the scripts, you will need to make a Weights and Biases account on wandb.ai. This is a platform used to track training of deep learning models.

After creating the environment and account, you can log in to Weights and Biases with

```
wandb login
```

After this, training progress will be logged to weights and biases.

# Guide on how to run experiments for DMT

### Architecture for Experiments

Experiments are based on classes. A parent class BaseExperiment defines various default parameters and default ways to run training, as well as providing a uniform interface for running experiments.

The BaseExperiment class contain all the hyper-parameters needed for models within Dynamic Mutual Training (DMT) and for the Baseline models. These are set to some default values that needs to be overridden for experiments to reflect changes in the experiment conducted.

All subclasses of BaseExperiment are registered to a class called Experiments which maintains a registry of all experiments that will be conducted. To run all experiments, all that needs to be done is

```
from path/to/dir/src.experiments import Experiments

Experiments.run_all()
```

or using the run_experiments.py file detailed above.

We do not suggest running all the experiments since it takes a long time; even running a single experiment for a subclass will require significant computing power.

### How the experiments are created

First, we have the Experiments class

```
class Experiments:
    def register:
        # keeps a experiment registry from Base Experiments

    def run_all():
        # runs all registered experiments
```

Then, we have the BaseExperiment class

```
class BaseExperiment(ABC):
    # initialize and holds all hyper-parameters used for
    # running a DMT/Baseline model

    # the properties of each experiment
    @property
    def description():
        # add model description

    @property
```

```
    def model_folder():
        # model folder to save experiment(s)

    # this is the function to override for running
    # constructed experiments
    @abstractmethod
    def run():
        # run some of the below run types

    # the type of runs to do for an experiment
    def _train_baseline_only(**baseline_parameters)
        # trains the baseline model and saves it

    def _base_run(**dmt_parameters):
        # trains and saves dmt model

    def _plabel_run(**plabel_parameters):
        # trains model on a varying proportion of
        # labeled data with pseudo labels in training after
        # pre-training
```

Now, for the base experiment, we have three run cases. These are all different with respect to what your experiment is performing, and they can all be used in combination with each other. The functionality of the seperate run functions are: - **_plabel_run**: Pseudo-label run method for all experiments. This method is called for pseudo label experiments for subclasses. This method trains the PLabel model and saves the best model (See PLabel class for details). If a baseline model is provided, it is also trained and saved.** - _base_run**: Base run method for all experiments. This method is called by the subclass run method. This method trains the DMT model and saves the best model.** - _train_baseline_only**: Training the baseline model only. Used for comparison with DMT. All default parameters are provided, so to modify a parameter, you should call e.g. `self._base_run(label_proportion=0.4)`.

Should you wish to define your own experiment, you should do the following:

```
class MyExperiment(BaseExperiments):
    # create some hyperparameters
    # in here that you want to test

    @property
    def model_folder():
        # set a return string with model folder

    @property
    def description():
```

```
        # set a return description of the experiment

    def run():
        # Implement the experiment performed here
        # inside use the run function from the three cases
        # of run experiment that you want to use
```

**How to run an individual experiment**

Write a Python file which imports a single experiment, e.g.

```
from path/to/dir/src/experiments import VaryLabelProportion

experiment = VaryLabelProportion()
experiment.run()
```

To perform DMT / Baseline experiments in the shell, first navigate:

```
Then, if you want to want to conduct your own experiment you can first import BaseExperiment

```bash
python from /path/to/dir/src.experiments import BaseExperiment
```

Then follow the steps above for creating an experiment, and then do:

```
my_experiment = MyExperiment()
my_experiment.run()
```

**Our Experiments**

We have the following list of conducted experiments in this project: - **TrainBaselines** - **VaryDifferenceMaximization** - **VaryDMTEpochs** - **VaryLabelProportion** - **PLabelVaryLabelProportion** - **PlabelDefault**

To conduct any of them follow the steps from the "**How to run experiments**". If you want to conduct all of them at once, do:

```
/path/to/dir/src/experiments python run_experiments.py
```

## Project structure

```
Plotting scripts to output figures used in the reports.
*_scripts.py

run_experiments.py: Runs all experiments

eval_data/
    |__dmt_loss.npy: The IoU of the DMT with varying label proportion
        on the test set. Sorted by label proportion.
```
```

```
    |__dmt_loss_dms.npy: The IoU of DMT with varying
        difference maximised smapling on the test set.
        Sorted by dms proportion.

    |__dmt_loss_epoch.npy: The IoU of DMT with varying epochs on the
        test set. Sorted by epoch.

    |__plabel_loss.npy: The IoU of PLabel with the default hyperparameters
        repeated 5 times.

    |__plabel_loss_label.npy: The IoU of PLabel with varying label
        proportion on the test set. Sorted by label proportion.

    |__baseline.npy: The IoU of the baseline with the default hyperparameters
         repeated 5 times, Sorted varying label proportion.

final_figs/
    Figures folder.

src/
|__experiments/
        |__experiment.py
            This is the main script for interaction with the code.
            It defines a base class that implements default
            hyperparameters and a wrapper for models,
            and how they are run and evaluated.

|__models/
        The implementation of torch.nn.Modules,
        and their training and evaluation methods.

        |__UNet.py
        |__DMT.py
        |__PLabel.py
|__pet_3/
        |__data.py
            The main interface for data is defined here,
            defining torch datasets, with other convenience methods.

        |__download_utils.py
|__utils/
        Utility functions/classes. Only ref if deeper understanding is
        required.
        |__datasets.py
        |__evaluation.py
        |__loading.py
```

```
        |__mixin.py
        |__training.py
|__plotting/
        |__temporaty_plotting_utils.py
```