Redesign and Extension of Source Code Package for Open-Set Classification on ImageNet

Master's project report

Dean Heizmann, Nicolas Fazli Kohler and David Pierre Sébastien Lebrec
16-704-033, 16-712-887, 21-739-362

Submitted on
February 8 2024

Thesis Supervisor
Prof. Dr. Manuel Günther

University of Zurich
Department of Informatics

Declaration of Independence for Written Work

I hereby declare that I have **composed** this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT) – the use of generative AI to **improve** my composed work was permitted by the thesis supervisor. I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used. All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zurich, 08.02.2024
Place, Date

Dean Heizmann N. Kohler [illegible]
Dean Heizmann, Nicolas Fazli Kohler and David Pierre Sébastien Lebrec

Master's project report

Author:	Dean Heizmann, Nicolas Fazli Kohler and David Pierre Sébastien Lebrec
Project period:	01.08.2023 - 08.02.2024

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

Acknowledgements

We would like to thank Prof. Manuel Günther for giving us the opportunity to work on this project and providing us with guidance and advice along the way.

Abstract

Open-set classification is a problem in machine learning, where models are required to perform classification of known classes, while simultaneously identifying unknown samples, i.e. inputs that are not part of any class the model has been trained on. This project aims to improve and extend an open-source package that allows the experimentation and comparison of different approaches for open-set classification. To do this, the project refactors the package to make it more intuitive to understand its structure and functionality, as well as simplifying potential future modifications and extensions. Additionally, the package first is extended with the EMNIST dataset, as an alternative to the already implemented ImageNet dataset. Second, it is expanded with two additional neural network architectures, LeNet and AlexNet.

Contents

Chapter 1

Introduction

1.1 Motivation and Definition of Goals

The field of deep learning for image classification has seen great improvement over the past decade. The breakthroughs regarding deep neural networks delivered in 2012 and 2015 (Krizhevsky et al., 2012; He et al., 2015), and the introduction of the ImageNet dataset (Deng et al., 2009) to the International Large Scale Visual Recognition Challenge (ILSVRC) (Russakovsky et al., 2015) accelerated the development in this field dramatically. Nevertheless, the advances also highlighted some limitations of current approaches. A crucial limitation is the issues that classifiers have, when faced with samples of previously unseen classes, often confidently misjudging them for seen classes (Dhamija et al., 2018, 2020; Palechor et al., 2023).

Since then, researchers have dedicated their efforts to understand and tackle the problem of network agnostophobia (Dhamija et al., 2018), called open-set classification, and provided different ways to prepare a classifier to detect and reject unknown samples. For example, thresholding the softmax probabilities, or the logits that are given to the softmax function (Hendrycks and Gimpel, 2017; Hendrycks et al., 2022). Other methods focus on estimating the likelihood of unknown classes by modeling deep feature representations derived by known classes and providing a probability for the rejection of a sample (Bendale and Boult, 2016a; Rudd et al., 2017a; Lyu et al., 2023). Alternatively, the network can be prepared for open-set classification by training it with an additional output for unknown classes or incorporating different loss functions to enhance the thresholding effectiveness of outputs (Dhamija et al., 2018).

Palechor et al. (2023) have developed an open-source package that allows the comparison of approaches for open-set classification based on ImageNet. This project focuses on restructuring and developing this open source package. It does so by implementing clean design choices, encapsulating functionality and promoting an intuitive understanding of the code and its structure. Finally it allows the simplified extension of the code by integrating new components, such as new network architectures or datasets. To demonstrate that, new such parts have been added to extend the current implementation: The Emnist (Cohen et al., 2017) dataset was added as an alternative to the ImageNet dataset. Furthermore, a customized LeNet (LeCun et al., 1998) and AlexNet (Krizhevsky et al., 2012) backbone were integrated alongside the already existing ResNet (He et al., 2016) architecture. In the following, the report will briefly discuss some theoretical concepts related to open-set image classification, before jumping into the documentation of the new codebase. The repository can be found on GitHub[1].

[1]Repository of the refactored codebase: https://github.com/AIML-IfI/osei

1.2 A Primer in Open-Set Classification

Techniques such as Decision Trees, Support Vector Machines and Neural Networks have achieved great success in areas such as image classification and image recognition (Varghese, 2018). While the breakthroughs in deep neural networks (Krizhevsky et al., 2012; He et al., 2015) accelerated the development in this field dramatically, it is important to consider that the performance of classifiers depend on whether they operate in a closed- or open-set environment:

Closed-Set Classification: When conducting closed-set classification, one generally follows the closed-set assumption, which states that all testing classes are encountered at training time (Scheirer et al., 2012; Ryota Yoshihashi, 2019). As such, a closed-set classification model only needs to classify at test time what it has previously seen at train time (Geng et al., 2021; Sun and Dong, 2023). Thanks to this simplification, closed-set models are efficient at detecting classes from a finite amount of known classes that remain unchanged between training and testing (Vaze et al., 2022). However, such closed-set models encounter limitations when faced with data outside their training distribution (Sun and Dong, 2023).

Open-Set Classification: To deal with data that lies outside the training distribution Scheirer et al. (2012) defined and formalized the task of open-set classification. They state that open-set classification is a more realistic scenario, given that not everything is known at training time. Hence an algorithm has to be able to deal with classes that it has not seen during training. Therefore, the goal of open-set classification is to maximize closed and open-set recognition capabilities by not only accurately classifying and identifying known classes but also unseen/unknown classes (Bisgin et al., 2023). In other terms, the main purpose of Open-set classification is to classify known instances and identify unknown instances (Geng et al., 2021).

1.2.1 How to approach Open-Set Image Classification

There already exists a plethora of open-set image classification approaches. At this point, the report will refrain from providing a complete list of approaches and instead focus on those that are provided by the repository while eventually mentioning some notable other contributions.

Threshold: This approach defines a confidence threshold, and follows the assumption that the probability for an unknown input would be low (Matan et al., 1990). As such, if the probability of an input is below this threshold, it gets classified as unknown. However, misclassification, even with high confidence, is frequent (Dhamija et al., 2018)! Therefore, other techniques have been investigated. Thresholding has been implemented in this project. An explanation can be found in subsection 3.3.2.

Garbage: In open-set image classification, it is possible to integrate negative classes (sometimes also simply referred to as unknown classes (Bisgin et al., 2023)), in the training set. These classes are unrelated to the target classes and thus help to increase the classifier's robustness when encountering unknown classes at test time (Geng et al., 2021). Such negative classes can either be added or generated by modifying or combining known samples (Bisgin et al., 2023). The project implements this idea and generally refers to it as the Garbage method. When applying the Garbage method, a model is trained with additional "unknown" or "garbage" classes. While the model sees the garbage classes at train time, they are removed during validation and testing to decrease overfitting (Palechor et al., 2023). This approach allows the model to better learn to differentiate between known classes and possible outliers. More information about the Garbage method and how it is implemented can be found in the subsection 3.3.2.

Proser: With the "Placeholders for Open-Set Recognition" (Proser) approach, introduced by Zhou et al. (2021), placeholders are learned for the data and the classifier during training. Proser enables the generation of new classes through the manifold mixup technique. At the same time, it is able to adjust the parameters of the classifier during the training process. Eventually, it allows the model to better separate known and unknown classes during inference. The implementation of Proser is relatively complex, please see the subsection 3.3.2 for details about Proser itself and its implementation in this project.

Extreme Value Machine (EVM): The Extreme Value Machine (EVM), formulated by Rudd et al. (2017b), models the decision limit for each class using extreme vectors and is great for recognizing outliers. It is inspired by the statistical extreme value theory (EVT), which helps to understand the likelihood of a data point belonging to a certain class, based on its distance from the class's defining points (Rudd et al., 2017b). It is possible to construct a probabilistic outline of each class's boundaries by selecting specific points and their distributions that most represent each class. As a result, it can minimize overlap. Extreme vectors (EVs) limit the risk associated with new, unseen data categories. The EVM can adapt to new information by updating these EVs, and can be considered an efficient nonlinear classifier. (Rudd et al., 2017b). EVM is covered by this project. Further information is available in section 3.4 and in the in subsection 3.6.4.

OpenMax: OpenMax, introduced by Bendale and Boult (2016b), adjusts the outputs and recomputes softmax probabilities to take unknown classes into consideration. Openmax is used to estimate the probability of an unknown class input based on the distance of its features from the features of known classes in the feature space (Bendale and Boult, 2016b). Openmax is part of this project, more information can be found in the worker section 3.4 and in the in subsection 3.6.4.

General Adversarial Networks Neal et al. (2018) used the image generation capability of Generative Adversarial Networks (GAN) to generate "unknown" images similar to those in the training set. Ge et al. (2017) followed a similar approach, by extending OpenMax and creating what they call Generative OpenMax (G-OpenMax). While this project does not support GANs at the moment, they could be added in the future.

Autoencoders While Autoencoders are typically used for tasks such as image denoising and compression, they have also found their way into the realm of open-set image classification (Oza and Patel, 2019; Ryota Yoshihashi, 2019). For instance, Oza and Patel (2019) propose an open-set classification algorithm that uses class-conditioned auto-encoders. They divide the training procedure into two sub-tasks: Closed-set classification (Encoder) and open-set identification (Decoder). Their encoder learns to do closed-set classification, while the decoder learns to reconstruct the image. To decide whether or not an image belongs to a known class, they model the reconstruction error. If said error is below or above a certain threshold, the image gets classified as either belonging to a known or unknown class (Oza and Patel, 2019). Autoencoders are currently not supported by this project.

Chapter 2

Evolution of Architecture High Level Overview

This chapter describes the state of the project's code at the start of the project in comparison to its final version. The focus is to provide a very high level overview of how separate parts of the code are encapsulated and how the different parts interact with each other. The goal is to highlight the advantages of the new implementation regarding the encapsulation of system components and the adherence to the Single Responsibility Principle of these components, compared to the previous implementation.

2.1 Initial Implementation

As it can be seen in Figure 2.1, the original state of the package was comprised of three main components:

- **Config Directory:** Contained YAML files for each distinct approach. The files held the configuration data for the training process and the dataset.
- **Script Directory:** Contained the scripts for training, evaluation, plotting, etc. that would be executed on command. They executed the code using the parameters given by the indicated YAML file.
- **Openset_ImageNet Directory:** Contained all the classes and functions necessary for the executable processes.

While the initial organization provided clarity and efficiency in effectively differentiating between the configurations, scripts, and the third directory that contained core functionalities and complex logic, it also raised concerns about scalability. As the package is expected to be extended in the future, it became necessary to develop a more sophisticated structure that adheres to general Object Oriented Programming Principles. This is essential not only to provide a more intuitive overview of the code but also to ensure smooth encapsulation of functionality and tasks within and across the various components.

2.2 Final Implementation

The final version, after all the refactoring has been done, keeps some core ideas of the initial implementation. The configurations are kept in their own directory and the script directory contains

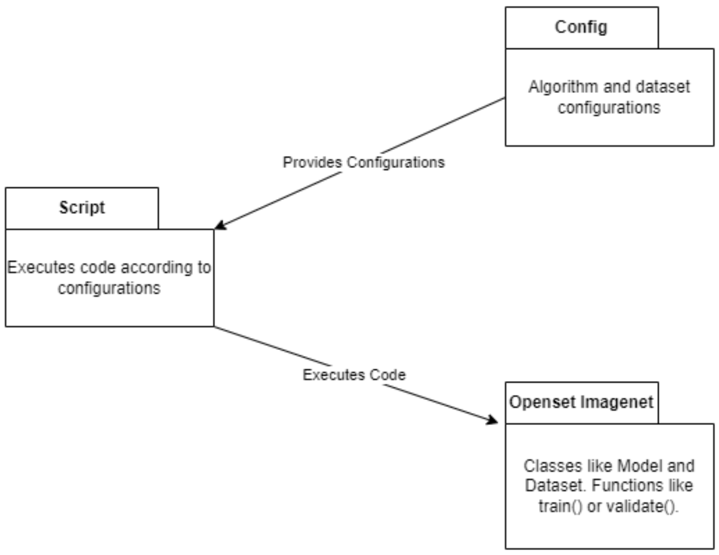

Config
Algorithm and dataset configurations
Provides Configurations
Script
Executes code according to configurations
Executes Code
Openset Imagenet
Classes like Model and Dataset. Functions like train() or validate().

Figure 2.1: The initial state of the package containing three major components; Config, Script and Openset_Imagenet.

the files to execute the rest of the package. The final composition of the package is defined by the following components:

- **Config Directory:** Contains YAML files for each distinct approach. In comparison to the old implementation, there are now three subdirectories. One for configurations regarding the algorithm, one for the dataset configurations, and one for the test/evaluation process. More information about the Config Directory are provided in 3.1.
- **Datasets:** The Datasets are now made up of a base dataset class and specific dataset implementations such as ImageNet and EMNIST with their specific logic, that inherit from the base dataset class. As this refactoring is especially extensive, there is a specific section for it in this paper, namely section 3.2.
- **Networks:** Handles everything related to Backbones and Models. Similar to the Datasets, this refactoring is extensive, as such please see section 3.3 for more details.
- **Workers:** Contains the worker functions that train models with different approaches and is described in section 3.4.
- **Script Directory:** While the functionality remains the same, the containing files were heavily restructured and reshuffled. See section 3.5 for more information.
- **Evaluation:** Contains services needed for the evaluation process. See subsection 3.6.1.
- **Losses:** Contains classes to initialize different losses as well as helper functions. See subsection 3.6.2.
- **Metrics:** Contains metrics required for plotting and evaluation. See subsection 3.6.3.
- **OSR_algorithms:** Contains two classes, one for OpenMax and one for EVM. See subsection 3.6.4.

- **Parameter_selection:** Contains services for the parameter selection process. See subsection 3.6.5.
- **Plotting:** Contains classes and services for plotting. See subsection 3.6.6.
- **Util:** Contains services used over the whole package. See subsection 3.6.7.

Figure 2.2 shows the overview and dependencies of the components in the new architecture. In the final implementation, the structure is more sophisticated and enhances the encapsulation of functionality and single responsibility (For a simplified overview, check out figure B.1). Even though the previous version was efficient, it was limited in terms of scalability and extensibility. However, with the introduction of new subdirectories such as Datasets, Networks, Workers, and more, each aspect of the package is separated in its own space, each being responsible for its tasks in the specific processes. This ensures that changes in one part of the system have no to minimal impact on other components, making it much easier to not only modify the code but also understand it more intuitively. Additionally, adding new components or extending existing ones by e.g. a new dataset or network architecture, can be done with none or very little consideration about the rest of the package.

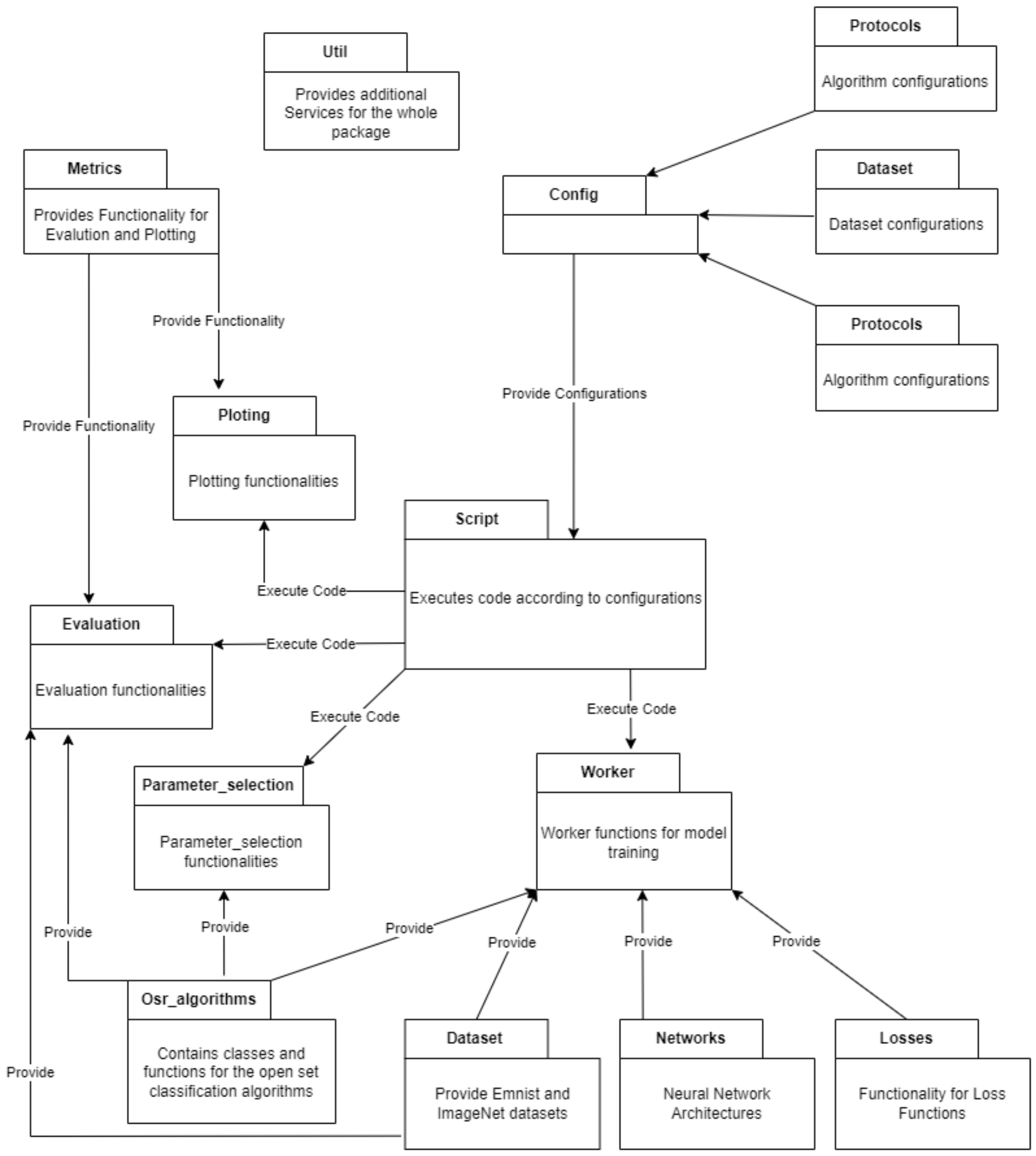

Util
Provides additional Services for the whole package
Protocols
Algorithm configurations
Metrics
Provides Functionality for Evalution and Plotting
Config
Dataset
Dataset configurations
Protocols
Algorithm configurations
Provide Functionality
Provide Functionality
Ploting
Plotting functionalities
Provide Configurations
Script
Executes code according to configurations
Execute Code
Execute Code
Evaluation
Evaluation functionalities
Execute Code
Execute Code
Parameter_selection
Parameter_selection functionalities
Worker
Worker functions for model training
Provide
Provide
Provide
Provide
Provide
Provide
Osr_algorithms
Contains classes and functions for the open set classification algorithms
Provide
Dataset
Provide Emnist and ImageNet datasets
Networks
Neural Network Architectures
Losses
Functionality for Loss Functions

Figure 2.2: The final state of the package, after refacturing the initial three parts into many.

Chapter 3

Documentation of new Implementation

Chapter 3 documents and explains the new project architecture as of February 2024. The refactored codebase can be found on GitHub[1].

3.1 Configuration Files

Configuration files carry information about how a process should be run. They define settings like the dataset, model type, protocols, loss function, etc. As such, there are three categories of configuration files: Protocol files contain general, as well as loss, optimizer, and algorithm parameters. The dataset configurations hold information about the used dataset and the test configuration files carry parameters for the evaluation process. When executing commands, these files are then passed as parameters for the execution. For instance, a training process can be triggered as follows: `train.py protocols/threshold.yaml dataset/imagenet.yaml`. More information about how the individual processes, in combination with their respective configuration files, can be triggered, is documented in the README A.

3.1.1 Protocol Config Files

Protocol files are structured to provide a flexible way to easily change the parameters of experiments. The general structure is as follows:

- **General parameters:** Non-technical details such as the output directory, the GPU index, and others.
- **Common parameters:** Common machine learning parameters such as batch size, number of epochs, etc.
- **Loss parameters:** Settings concerning the loss function. At the moment the following losses are supported: Entropic Openset Loss, SoftMax, Garbage Class Loss.
- **Optimizer parameter:** Settings for the optimizer.
- **Algorithm type:** Indicates which algorithm the config file revolves around.

[1]Repository of the refactored codebase: https://github.com/AIML-IfI/osei

- **Network:** Defines which backbone architecture shall be used. At the moment three backbone types are supported: LeNet, AlexNet and Resnet50. More information about the specific backbones can be found in 3.3.

3.1.2 Dataset Configuration Files

For each specific dataset, there is a configuration file. The specified attributes differ from dataset to dataset and depend on the implementation. For more information about the datasets, please see section 3.2. In general, only two parameters are mandatory in all dataset configurations:

- `dataset`: Name of the dataset as a string in lowercase.
- `image_size`: Size of an image of the dataset in pixels.

3.1.3 Test File

Any parameters necessary for the evaluation process are defined here. Some important ones include:

- `model_path`: Structure of the path where the algorithm expects to find the models to evaluate. The path is then dynamically composed at runtime.
- `output_directory`: Output folder defining where the algorithm saves the results.
- `network`: It is necessary to define which network architecture is used in the model to be evaluated.

3.2 Datasets

As of writing this report, two datasets are supported. ImageNet, which was part of the original codebase, and EMNIST. While ImageNet is fully supported, EMNIST is still considered to be in an experimental state. Generally speaking, EMNIST is smaller and is great for testing code changes fast and locally, while ImageNet fully supports the use of different protocols. In this section, the new inheritance-based architecture of the dataset implementation will be discussed.

3.2.1 General Architecture

At the top of the hierarchy is the Dataset class of the torch.utils library[2]. The parent class `Base_Dataset`, inherits from it, while the classes `EMNIST_Dataset` and `Imagenet_Dataset`, in turn, inherit from the `Base_Dataset` class. The `Base_Dataset` class contains some class variables as well as the functionality to create and return PyTorch data loaders.

The classes `EMNIST_Dataset` and `Imagenet_Dataset` then extend the inherited properties with dataset-specific properties. They use lazy initialization for almost all instance variables; this means that at the first initialization, these instance variables are set to None and are only initialized upon the first call and stored as instance variables. This becomes especially important for the training, validation, and test data.

The overarching principle for generating the training, validation, and test sub-datasets, as well as accessing them, is the same for both datasets. A specific dataset instance, e.g., of `Imagenet_Dataset`, let's call it **imagenet_container**, serves as an initially empty container for the subsets.

[2] Information about the torch.utils library can be found here: https://pypi.org/project/torch-utils/

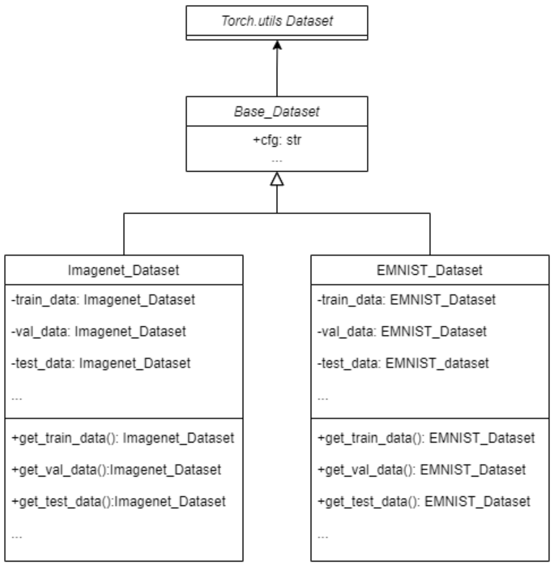

Torch.utils Dataset
Base_Dataset
+cfg: str
...
Imagenet_Dataset
-train_data: Imagenet_Dataset
-val_data: Imagenet_Dataset
-test_data: Imagenet_Dataset
...
+get_train_data(): Imagenet_Dataset
+get_val_data():Imagenet_Dataset
+get_test_data():Imagenet_Dataset
...
EMNIST_Dataset
-train_data: EMNIST_Dataset
-val_data: EMNIST_Dataset
-test_data: EMNIST_dataset
...
+get_train_data(): EMNIST_Dataset
+get_val_data(): EMNIST_Dataset
+get_test_data(): EMNIST_Dataset
...

Figure 3.1: Inheritence of dataset classes and type of train, validation and test datasets.

The **imagenet_container** has its instance variables test_data, validation_data, and test_data, all set to None.

To get the training data, the classes getter function is invoked: `imagenet_train_data = imagenet_container.get_train_data()`. The **imagenet_container** will now create the training dataset, assign it to itself as instance variables, and then return it. If the function `get_train_data()` of the **imagenet_container** is called in the future, the training dataset will be returned immediately, since it is stored as instance variable of the **imagenet_container**. Figure 3.1 shows the inheritance of the dataset classes and the principle of having a dataset subclass as a container, which has instances of itself assigned as train, validation and test data.

It becomes clear how well this works when evaluating a trained model. For this process, an empty container instance e.g. **imagenet_container** is created again. For the validation process, only the test dataset is needed, but not the training and validation data. This time around the test data subdataset is initialized and accessed with the function `imagenet_container.get_test_data()`. The test dataset is created and assigned to the **imagenet_container** as an instance variable for later retrieval. However, training and validation datasets are not created.

3.2.2 Dataset Specific Documentation

A major distinction between the two datasets, EMNIST and Imagenet, in the current implementation, is that it is assumed that the dataset is already present on the hard drive while EMNIST needs to be downloaded. For EMNIST, this means that an initialized `EMNIST_Datset` instance must download the data first, for each subset train, validation, and test. However, these subsets are stored locally and are directly accessible afterward.

Imagenet

The class `Imagenet_Dataset` consists of a getter function for each sub-dataset: **train**, **validation**, and **test**. The getter first checks whether the `Imagenet_Dataset` (container)-instance, from which it is called, already has the desired sub-dataset saved as an instance variable.

If not, the getter function first calls the corresponding class function, for example the function `get_train_file()`. This function returns the CSV file from the protocols folder, according to the protocol specified in the command-line input. The CSV file, for example, `p1_train.csv`, contains the path of each image and its corresponding label to be used for creating the dataset.

The getter for the sub-dataset continues execution and creates a variable with image transformations. It creates an `Imagenet_Dataset` instance based on the information from the CSV file and the parameters for image transformation.

Next, the getter checks whether adjustments need to be made due to the employed loss function:

Train data preprocessing:

- **Garbage loss:**
  - Proser algorithm: Removes all negative labels from the dataset.
  - Other algorithms: Replace all negative labels in the dataset with the highest existing label plus one.
- **Softmax loss:** Removes all negative labels from the dataset.

After the initial preprocessing based on loss type and algorithm, the following operations are applied:

- Update the label count and unique classes in the dataset.
- **Entropic loss:** The total label count is decreased by one.
- Only for the combination of the Proser algorithm with **garbage loss**: The label count is increased by one.

Validation data:

- **Garbage loss**: Replaces all negative labels in dataset with highest label.

Test data

- **Garbage loss**: Replaces all negative labels in dataset with highest label.

Finally, the sub-dataset instance is assigned the number of labels and unique labels as instance variables before being assigned as instance variable to the `Imagenet_Dataset` instance, which serves as container.

EMNIST

Similar to the `Imagenet_Dataset` class, the `EMNIST_Dataset` class contains getter functions for training and validating sub-datasets. The getter first checks whether the `Imagenet_Dataset` instance, from which it is called, already has the desired sub-dataset saved as an instance variable. During the initial initialization, all variables are set to None.

If not, the dataset is downloaded, which can be easily done using the `torch.datasets-.EMNIST` class. Before that, the `get_data_depending_on_loss()` function is called, which determines whether it is training or validation data and what parameters need to be set based on the loss function. The same parameters apply to both training and validation data.

- **Softmax loss**: Does not include unknown class, hence remove all negative labels from the dataset. The label count is currently hard-coded to 11.
- **Garbage**: Includes unknown classes and adds an additional garbage class.
- **Proser algorithm**: Only for validation data and in case of entropic and softmax loss, the label counts is increased by 1.

Then either the training or validation set is downloaded through the `torch.datasets.EMNIST` class, and an `EMNIST_Dataset` instance is created along with the other parameters. For example: {`which_set` = "train", `has_garbage_class` = True, `include_unknown` = False}.

During the initialization of an `EMNIST_Dataset` instance, the constructor checks if a value for '`which_set`' has been passed. If yes, it automatically loads the dataset and calculates other variables like `letters`, `unique_labels`, and `labels_count`. Then, the black and white images (1 channel) are converted to RGB (3 channels). This step is necessary to ensure that the data has the correct dimensions for the input layers of the defined neural networks. Finally, the created train or validation `EMNIST_Dataset` instance is added as an instance variable to the overarching `EMNIST_Dataset` instance.

3.3 Networks

In Computer Vision, Convolutional Neural Networks (CNNs) are used to extract low and high-level features from raw image data (Elharrouss et al., 2022) and are commonly referred to as backbones. On top of the backbone lay the "Neck" and the "Head", which use these learned features for various subsequent tasks, such as Image Classification or Object Detection (Li et al., 2022). Since Open Set Image Recognition is a subset of Computer Vision the design philosophy also applies. The backbone can be a simple network like LeNet or more advanced like Resnet50. Regardless of the chosen backbone, there will be a head on top of it, that acts as a classifier which decides whether a given input belongs to a class or is unknown.

Following this general design philosophy, this project uses backbones too. These backbones are then used by the models, which add/adapt layers and define train/evaluation loops as well as other model specifics. This separation allows to easily mix and match backbones with models as needed. To prove this modularity, the existing Resnet50 network was adapted and two additional backbones have been added, namely, Alexnet and a customized LeNet variant. Thanks to the modular design these backbones can now be paired with the new Threshold, Garbage and Proser model. In the following, the report provides a closer look at the general network architecture depicted in Figure 3.2. To do this, it will explain how these backbones and models are implemented and how new backbones and models can be added in the future.

3.3.1 Backbone

One goal of the project was to make adding new backbones easy. As such, a modular design with a great emphasis on Object Oriented Programming, especially with regards to Polymorphism and Inheritance, has been chosen for the backbones. Hence a general parent Class, simply named `Backbone`, and various Child classes, that represent the specific network architectures, are defined.

The `Backbone` class defines two abstract methods: `first_blocks()` and `last_blocks()`. These two functions are implemented by the respective child classes. `first_blocks()` should pass the input through the first few blocks of the network while `last_blocks()` passes it through the last few layers and returns the features. Additionally, there is a concrete function

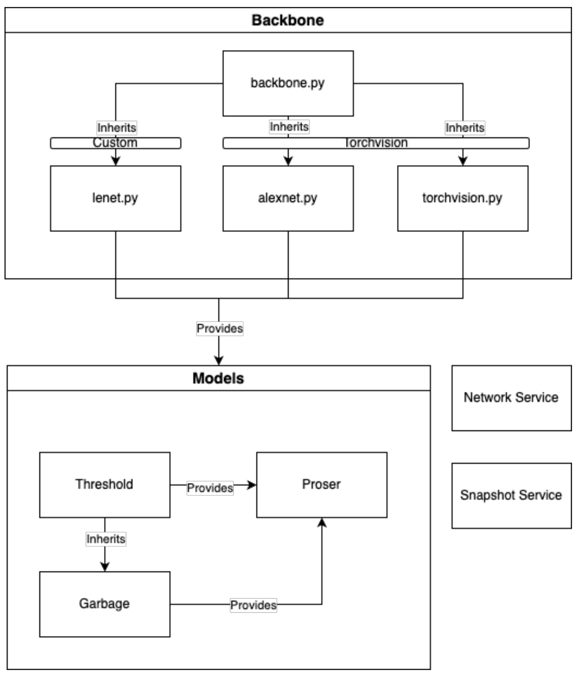


Figure 3.2: Network Architecture

`forward()`, which calls `first_blocks()` and `last_blocks()` sequentially. The differentiation into `first_blocks()` and `last_blocks()` is essential, as it enables Manifold MixUp required by Proser (Zhou et al., 2021). Lastly, there is also a `calculate_out_feature_size()` function that can be used by child classes to calculate the output size. During initialization, the backbone expects a `base` and `out_features`. The base contains the layers of a specific backbone network architecture, for instance, Resnet50, while the `out_features` defines the dimensionality of the last backbone layer. As these two variables are specific to the chosen backbone network architecture they are defined in their respective subclasses and passed to the Backbone class during initialization. This approach ensures a common setup regardless of the underlying backbone network architecture.

As of writing this paper, three different specific backbone network architectures are supported: Resnet50, AlexNet, and a customized LeNet variant. For adding Resnet50 and AlexNet the base torchvision implementation was used. While for the customized LeNet variant the layers have been defined "manually". Figure 3.3 visualizes the architecture as an UML Diagram.

LeNet

LeNet, was first introduced in 1989 by LeCun et al. (1989). Thus, it is a well-known and simple architecture that has been proven to work well with the EMNIST dataset (Schaub and Hotaling, 2023). It is simple and training with it is usually very fast. Thanks to its lightweight nature, it's a great network for this project as it allows to quickly test code changes. As part of this project, LeNet consists of two layers, where each layer uses 2d Convolution, followed by a RELU activation function and Maxpooling. As input, three channels (RGB) are expected. This allows to easily apply it to the ImageNet dataset. The concrete `first_blocks()` and `last_blocks()` functions, run the first and last layer respectively.

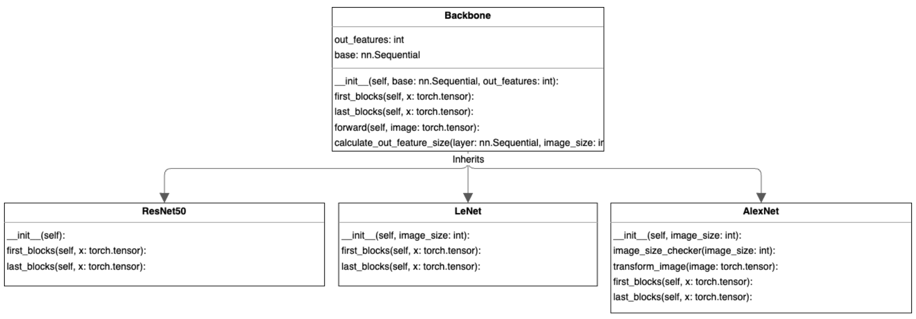

Backbone
out_features: int
base: nn.Sequential
__init__(self, base: nn.Sequential, out_features: int):
first_blocks(self, x: torch.tensor):
last_blocks(self, x: torch.tensor):
forward(self, image: torch.tensor):
calculate_out_feature_size(layer: nn.Sequential, image_size: ir
Inherits
ResNet50
__init__(self):
first_blocks(self, x: torch.tensor):
last_blocks(self, x: torch.tensor):
LeNet
__init__(self, image_size: int):
first_blocks(self, x: torch.tensor):
last_blocks(self, x: torch.tensor):
AlexNet
__init__(self, image_size: int):
image_size_checker(image_size: int):
transform_image(image: torch.tensor):
first_blocks(self, x: torch.tensor):
last_blocks(self, x: torch.tensor):

Figure 3.3: Backbone Architecture

AlexNet

Alexnet is another well-established network and has been used for ImageNet classification in the past (Krizhevsky et al., 2012). Like LeNet, Alexnet is a CNN, however it is more complex and has more layers. For this project the non-pretrained Alexnet from Pytorch[3] is used. Using a torchvision model makes it easy to use a pretrained model in a future step. While the Alexnet implementation also possesses `first_blocks()` and `last_blocks()`, it has two additional functions: `image_size_checker()` and `transform_image()`. The reason being, that the standard Alexnet implementation does not work on datasets with small images, like EMNIST. That is because, typically, each convolution and pooling layer reduces the dimension of the input, which can eventually result in a zero or even negative dimension. As such the implementation first checks whether the input size is too small. If it is, it asks the user whether or not the images should be automatically resized. ImageNet is not affected as its standard image size is 224px*224px compared to EMNIST's 28px*28px.

Resnet50

Resnet50 (He et al., 2016) is another popular CNN. Like Alexnet, the torchvision Resnet50[4] is used, and the `first_blocks()` and `last_blocks()` are defined.

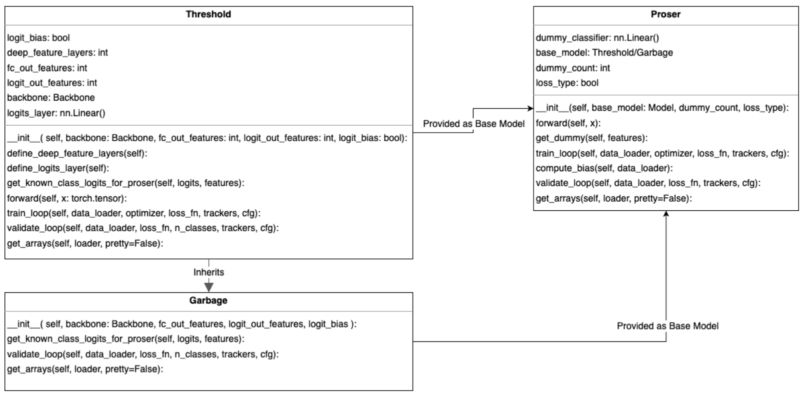
How to add a new Backbone

The current implementation can easily be extended with additional backbones. For this, one has to simply take a backbone from pytorch or create a custom one and place it in the respective folder. Generally speaking, the required steps are as follows:

1. The new file must be added to either `/networks/backbone/custom` or `/networks/backbone/torchvision`.
2. The new backbone class must inherit from the `Backbone` class and add `first_blocks()` and `last_blocks()` functions to make it compatible with Proser.
3. Adapting the imports within the `backkbone.__innit__.py` file.
4. Adapting the `get_backbone()` function in `network_services.py`.

[3]The AlexNet Model from Pytorch can be found here: https://pytorch.org/vision/main/models/generated/torchvision.models.alexnet.html

[4]The Resnet50 Model from Pytorch can be found here: https://pytorch.org/vision/main/models/resnet.html



```
Threshold
logit_bias: bool
deep_feature_layers: int
fc_out_features: int
logit_out_features: int
backbone: Backbone
logits_layer: nn.Linear()
__init__( self, backbone: Backbone, fc_out_features: int, logit_out_features: int, logit_bias: bool):
define_deep_feature_layers(self):
define_logits_layer(self):
get_known_class_logits_for_proser(self, logits, features):
forward(self, x: torch.tensor):
train_loop(self, data_loader, optimizer, loss_fn, trackers, cfg):
validate_loop(self, data_loader, loss_fn, n_classes, trackers, cfg):
get_arrays(self, loader, pretty=False):
```

```
Proser
dummy_classifier: nn.Linear()
base_model: Threshold/Garbage
dummy_count: int
loss_type: bool
__init__(self, base_model: Model, dummy_count, loss_type):
forward(self, x):
get_dummy(self, features):
train_loop(self, data_loader, optimizer, loss_fn, trackers, cfg):
compute_bias(self, data_loader):
validate_loop(self, data_loader, loss_fn, trackers, cfg):
get_arrays(self, loader, pretty=False):
```

```
Garbage
__init__( self, backbone: Backbone, fc_out_features, logit_out_features, logit_bias ):
get_known_class_logits_for_proser(self, logits, features):
validate_loop(self, data_loader, loss_fn, n_classes, trackers, cfg):
get_arrays(self, loader, pretty=False):
```

Figure 3.4: Model Architecture

3.3.2 Models

As previously introduced, a model takes a backbone, adds/adapts specific layers, and defines the train/validation loops as well as other model specifics. At the moment, there are three different models, each with its own approach as to how the problem of open-set image classification shall be tackled. The models are the Threshold Model, The Garbage Model which inherits from the Threshold Model, and the Proser Model. Figure 3.4 shows how these models relate to each other and which variables and methods they carry. Please note that, while all three approaches already existed in the base implementation, they have been adapted to bring them in-line with the general model architecture. As such, these models, and how they are implemented, will be introduced in more detail.

Threshold Model

As part of this project, the Threshold model can be considered the base approach. The idea is to train the model on known classes only and then classify whether the class belongs to a known class or an outlier (Sun and Dong, 2023). These class probabilities are then predicted via an appropriate loss function like Softmax. If the probability is below a certain threshold the image is rated to be unknown (Sun and Dong, 2023).

In the current implementation the Threshold model expects a backbone, the number of output features of the fully connected layer (`fc_out_features`), the number of output features of the logits (`logits_out_features`) and whether or not a `logit_bias` shall be used. During the initialization, the `deep_feature_layer` as well as the `logits_layer` are defined by calling their respective "define" methods: `define_deep_feature_layer()` and `define_logits_layer()`. This setup allows a potential child class of Threshold to simply override these "define" methods with its own specification. The `forward()` function, called from within the `train_loop()`, `validation_loop()` and `get_arrays()` methods, first calls the `forward()` function of the backbone. It then passes the resulting `backbone_features` into the `deep_feature_layers()`. Afterwards, it takes these `deep_features` and passes them through the `logits_layer()` to get the `logits`. Finally, the `logits`, along with the `deep_features` are returned. The report will not go into detail about how the `train_loop()`, `validation_loop()` and `get_arrays()` are defined, as there were little changes made to them, compared to the original codebase. However, there is one additional function that has not been discussed so far, the `get_known_class_logits_for_proser()`, which will be covered in detail as part of the Proser model.

Garbage Model

The general idea of the Garbage model is to train with so-called unknown/garbage classes. These unknown/garbage classes are then removed at test time to prevent overfitting (Palechor et al., 2023). With this technique the presence of unknown classes during train time is simulated, effectively allowing the model to better account for them. Since the Threshold model can already handle unknown classes at train time, the Garbage model inherits from it. While the `train_loop()` can stay the same, garbage specific code is needed for the `validate_loop()` and `get_arrays()` functions. Hence these functions are overridden. Lastly `get_known_class_logits_for_proser()` is also overridden, but as previously mentioned, more details on that will follow when the Proser model is described.

Proser Model

Proser (PlaceholdeRs for Open-Set Recognition) is a specific method to improve open-set recognition and was introduced by Zhou et al. (2021). Proser introduces placeholders for data and classifiers and tries to learn invariant information between target and non-target classes. It does so by using Manifold Mixup to create new classes during runtime and trying to balance the classification of known classes and the detection (i.e. not-classification) of unknowns. For more detailed information please see the paper by Zhou et al. (2021).

However, it is worth noting that, to facilitate this, Proser calls the `first_blocks()` and `last_blocks()` functions separately in the main `train_loop()`. This is the reason, why these two functions have to be present in the specific Backbones. During its initialization, the Proser Model expects a base model (Threshold or Garbage), as well as the `loss_type` and `dummy_count` in order to create the `dummy_classifier`. The `forward()` function gets the logits and features by calling `base_model.forward()`, i.e. the `forward()` method of the base model. It then uses the retrieved features to acquire the dummy. Eventually the `forward()` function returns the logits, features and dummy.

However, if Proser is paired with a Garbage base model, it has to be ensured that only the known class logits are retrieved, i.e. the logits without the unknown class. As such, in the `get_arrays()` and the `validate_loop()` methods, the `forward()` function is triggered to retrieve the logits, features and dummy. The logits and features are then passed to the `base_model.get_known_class_logits_for_proser()` method, which eventually returns the known class logits. Thanks to the Object Oriented Programming concept of Method Overriding, accounting for whether or not the base model is Threshold or Garbage, is trivial. As the `get_known_class_logits_for_proser()` function is defined twice, once as Identity Function in the Threshold model and once in the Garbage model, where the unknown class `logits` are removed.

For more details on how Proser is implemented, it is recommended to study the code, as going into the full detail could, and is, a paper on its own (Zhou et al., 2021).

How to add a new Model

Adding a completely new model is more complex than just simply adding a new backbone. However, the main procedure can be broken down into four steps:

1. Adding the new model to `/models`.
2. Adapting the `__init__.py` `/models`.
3. Adapting the `network_services.py` as needed.
4. Potentially creating specific worker classes etc.

3.3.3 Services related to Networks

Network Services

The purpose of the network_services is to provide a central point to handle model creation and retrieval. As such network_services is inspired by the Mediator Design Pattern [5]. By applying this design pattern, loose coupling is achieved, which is essential for streamlining the future addition of new Backbones and Models. The two main functions that are made accessible by the network_services are the get_model() and extract_arrays(). The extract-_arrays() function simply calls the get_arrays() function of a specific model and returns the result, while the get_model() function is concerned about the instantiation and retrieval of the model architectures. This model instantiation is important for the workers as well as for the post-processing. To do this, the get_model() function takes a Configuration file and the amount of classes. It then checks whether, based on the Configuration, a Proser model is needed or a Threshold/Garbage model is enough. Following this assessment, it then calls the appropriate private _get_base_model() function, and depending on the result of the assessment, the _get_proser() function as well. In order to simplify the retrieval of Backbones, there is also the _get_backbone() function, which is used by the aforementioned private _get_base-_model() function.

Snaphsot Services

The snapshot_services is concerned about the saving and loading of model checkpoints. For this it offers a save_checkpoint() and a load_checkpoint()function. In addition it also has a helper function that manage checkpoint naming. The get_output_model_checkpoint-_name() produces the correct checkpoint name, given the Configuration. Let's assume the provided Configuration specifies a Threshold Algorithm, a Softmax Loss, a LeNet Network and Imagenet as dataset. Then the generated name would look as follows: imagenet_lenet_softmax-_threshold_best.pth. While this is a very long name, its expressiveness allows for no ambiguity, making it clear which parameters were used during model training. Eventually, the model will be saved in the appropriate directory. For instance, if it used Protocol 1 it will be saved in /experiments/imagenet/protocol_1.

3.4 Workers

The worker is one of the core feature of our implementation. One worker function is used for one or more algorithms (EVM / OpenMax / Proser / Threshold). The OSEI project's root directory contains a "workers" folder where three files regarding the workers are located: worker-_services.py, worker_openmax_evm.py and worker_threshold_proser.py.

The worker_services.py file contains functions used by both, the worker_openmax-_evm.py and the worker_threshold_proser.py. The worker_openmax_evm.py file contains the worker for OpenMax and EVM, while the worker_threshold_proser.py file holds the worker function for Threshold and Proser. The worker functions organize the training process, which consists of six different main steps:

1. **Initialization and configuration loading**
    - **SnapshotManager**: Manages model checkpoints for the training phase.

[5] An Explanation of the Mediator Design Pattern can be found here: https://www.patterns.dev/vanilla/mediator-pattern/

   - **Configuration**: Cfg contains settings like the type of loss, the type of model, the number of workers, the number of epochs, the batch size as well as the number of batches for EVM, OpenMax, Proser or Threshold.
   - **Logger setup**: Documents the training progress.
   - **Device configuration**: Configures the training device (CPU/GPU) as specified in the settings.

2. **Data preparation**
   - **Data loading**: Uses `dataset_services.py` to obtain the training and the validation datasets, and to create data loaders for model feeding during training and validation.

3. **Model preparation**
   - **Model initialization**: Initializes a model with the correct number of classes and selects an appropriate loss function based on the model.
   - **Optimizer and scheduler**: Sets up objects like early_stopping to prematurely stop the worker function. Furthermore, StepLR can adjust the learning rate during the training process.

4. **Training loop**
   - Each **Epoch** involves a:
     - training loop, which trains the model on the training set to optimize the loss.
     - performance tracking step, which logs a score corresponding to best metrics for an epoch after a certain point in time.

5. **Model saving**
   - **Checkpoints**: Saves the current model state at the end of each epoch. For each epoch the state of the model is saved in a pth file.

6. **Clean-up**
   - After completing the training process (after completing all epochs or before due to early stopping), the script performs a resource clean-up and removes the current model from memory.

While overall, The `worker_openmax_evm.py` and `worker_threshold_proser.py` have a similar "architecture" they still hold a few differences:

- They manipulate different objects.
- The clean-up step is not done in `worker_openmax_evm.py`.
- There is no early_stopping in `worker_openmax_evm.py`.
- There are more logs in `worker_threshold_proser.py`.
- The custom scheduler is only present in `worker_threshold_proser.py`.

3.5 Scripts

The package provides four base capabilities that form a pipeline of processes from training a model to plotting its performance on open-set tasks. These processes, in short, involve:

1. The training of the base model as well as the application of an algorithm on top of that base.
2. (Optional for EVM and OpenMax) Parameter optimization.
3. Evaluation to extract performance statistics.
4. Plotting using extracted statistics.

When the user interacts with the command line, the Entry Points defined in the `setup.py` trigger the run method in the specified script. As such the scripts can be considered the start and endpoint of code executions, triggered by a user. All scripts located in the script folder share the same structure. They all possess only two static methods, a method called `command_line_options()` and a method called `run()`. The `command_line_options()` function is responsible for argument parsing, while the `run()` method prepares the configuration files and kickstarts the actual training, plotting etc.

For all Scripts discussed in this section, additional information, including example commands, can be found in the README, either on the repository or in chapter A. In addition, at time of submission, some combinations of commands with specific losses/algorithms/datasets fail. For more information about which command and combinations work, see section 5.1 or the excel in the repository.

3.5.1 Train and Train All

With the `train` and `train_all` commands a model (or multiple models) can be trained. This training is influenced by parameters specified in both the dataset and the algorithm configuration files. Selecting these configuration files is achieved through command line inputs, which is done by specifying the configuration file's name, for example, `datasets/imagenet.yaml` or `algorithms/threshold.yaml`. Adjustments to specific training parameters, such as the loss function, step size, or network architecture, are made by modifying the corresponding values in the algorithm configuration file located in `configs/algorithms`. While the `train` command requires choosing a protocol, the `train_all` command allows the training with multiple protocols, as well as with multiple algorithms and loss functions. When training a model it is also important to keep in mind that training a Proser, EVM or OpenMax model requires a Threshold Model that has been trained on the same protocol, loss function and network architecture. As some combinations models/losses/algorithms/datasets might fail, it is generally advised to only train what is actually needed.

3.5.2 Evaluate

The evaluate script is executed by the the `evaluate.py` command. If executed without any specified parameters, the evaluation script will, by default, assess all valid trained models it detects. Alternatively, there is the option to narrow down the evaluation to a specific subset of models trained with specific algorithms, loss functions, and protocols by providing the corresponding parameters in the command. There it is also possible to configure the process to evaluate the current model or the best performing.

When initiated, the script first checks if it has to load current models, or the best performing ones. It then iterates through all the open-set classification algorithms and all loss functions it should consider. For each combination it calls the `process_model_evaluate` function from the evaluation directory. It is always necessary to provide the dataset parameter via a configuration file. During that, the evaluation process will extract features, logits and scores from the test dataset and store the results in the `osei/experiments` directory. Furthermore, the evaluation script takes into account a configuration file, which is, by default, set to `configs/tests/test-.yaml`. This configuration file contains additional parameters, including network architecture specifications and optimized parameters for validation, specific to each individual algorithm.

3.5.3 Select Parameters

Some of the algorithms necessitate adjustments to their parameters to align with various loss functions and protocols. In particular, the EVM and OpenMax algorithms come with a predefined set of parameters that require optimization. Prerequisite for parameter optimization, is that a model has already been trained and is available in the directory (just as for evaluation).

The `select_parameters` script first initializes a dictionary to store all maximum values related to the evaluation. It considers parameters and iterates over all required combinations of algorithms, protocols and loss functions. It then calls `process_model_parameter_selection()` from the `parameter_selection` directory for each combination. The results are stored in a `LaTeX` table.

Finally, the code generates a summary table of maximum values for each combination of algorithms, losses, and protocols. It prints the best results for each algorithm, protocol, and loss combination, including detailed information about the selected parameters and their values.

3.5.4 Plot All

The `plot` command uses the evaluation data (scores and ground truths) generated by the `evaluate` command for the creation of plots and tables. Note that this process makes use of the `configs/test/test.yaml` file! As such it should be ensured that the appropriate network architecture is specified in the configuration file. For retrieving the scores and for plotting itself the `PlotScoring` class, the `PlotGraph` class and the `PlotReporting` class are used. These classes are all located in files within the `plotting` directory. Eventually, the results are compiled into a PDF file, named `Results_last.pdf`. This PDF contains multiple pages. The first page focuses on presenting the OSCR plots for all algorithms applied to networks trained with all loss functions, where both negative and unknown samples are evaluated for all protocols. The second and following pages show the score distribution plots for each algorithm, but page wise separated for each loss function.

At this point it is worth noting that the current implementation of the plotting mechanism is buggy. See section 5.1 and the excel provided in the repository, for more details.

3.6 Other Code Directories

In this section, the remaining directories will be introduced. These directories do not warrant their own section, as they either just provide some functionalities to parts of the code that has already been discussed, or are simply not as complicated or extensive. Hence, they are all grouped together here as part of the "Other Code Directories". As always more details about the implementation can be found in the code itself as well as in the accompanying docstrings.

3.6.1 Evaluation Directory

The evaluation_services.py file in the evaluation directory, contains all the code that was previously located in the evaluate.py script, except the command_line_options() and run() method. This was done to keep all the script files consistent with regards to their structure, as each of them only offers the command_line_options() and run() method. While the write_scores(), load_scores(), process_model_evaluate() and _post_process-_evaluate() methods are now named differently and were moved to the evaluation_ser-vices.py file, their functionality are, for the most part, unchanged. Hence please see the code itself for more details.

3.6.2 Losses Directory

In the losses folder, loss_services.py defines two functions that create and return loss functions when using threshold or proser. These loss functions measure the difference between the predicted outputs and the true targets. Furthermore, these two functions guide the training process by updating the model's weights.

The loss_threshold function is chosen based on configuration settings. The settings can be the entropic loss for open-set classification, the standard cross-entropy loss for multi-class classification, or a version of cross-entropy with balanced class weights to address class imbalance. The loss_proser function returns the standard cross-entropy loss. loss_threshold and loss-_proser are useful for different types of classification problems.

The class loss_early_stopping.py is used to stop the training process if there is no improvement in the validation loss or other specified metric after a certain number of epochs. The patience parameter indicates the number of epochs to wait before there is an improvement and the delta parameter indicates the minimum change to consider as an improvement. These two parameters help to define the moment to stop the process.

The class loss_average_meter.py is used to track the average and current values of a metric (e.g. loss) over time. It is often used in training loops to monitor performance.

The class entropic_openset_loss.py creates a custom loss function designed for open-set learning. The entropic part of the loss function incites the model to have a uniform distribution over classes for unknown samples. As a result, overconfident predictions are less likely.

An Overview of the three different losses can be seen in Figure B.2.

3.6.3 Metrics Directory

Within the metrics folder there are two files that define a ValidationMetrics class and a EvaluationMetrics class each. The EvaluationMetrics class is used for tasks related to plotting and parameter selection and offers two functions: calculate_oscr() and ccr_at-_fpr(). The validation_metrics.py file contains the ValidationMetrics class, which has two functions too, as well as method called confidence which is used by the Models (Threshold, Garbage and Proser) to get their respective confidence values. The two methods of the ValidationMetrics class, namely auc_score_binary.py and auc_score_multiclass-.py are currently not used in the project! It was decided against removing them as they might be beneficial in fixing bugs related to plotting in the future.

For the validation part, confidence levels are used in the whole project. the main confidence's properties are:

- kn-conf: confidence of known samples.
- kn-count: count of known samples.

- neg-conf: confidence of negative samples.
- neg-count count of negative samples.

3.6.4 OSR Algorithms Directory

The `osr_algorithms` (Open Set Recognition Algorithms) folder contains three files: `evm.py`, `openmax.py` and `postprocessing.py`, which all are needed by the `worker_openmax_evm-.py` file. The `evm.py` and `openmax.py` both define their respective classes `EVM` and `OpenMax` while inheriting from the `PostProcessing` class defined in `postprocessing.py`. This inheritance is motivated by the fact that both, the `EVM` and `OpenMax` classes make use of the `postprocess_train_data()` and `compose_dicts()` methods, defined within the `Post-Processing` class. As the saving of `EVM` and `OpenMax` depends on different hyperparameter combinations, a `save_models()` method has been added. For more details about the methods within `postprocessing.py`, `evm.py` and `openmax.py`, please see the Docstrings and code itself.

3.6.5 Parameter Selection

The `parameter_selection` directory consists of one file, named `parameter_selection-_services.py`. It basically contains the code that previously was located within the `select-_parameter` script, placed in the `script` folder. This was done to keep the style consistent between the `select_parameter` script and the other entry point scripts. As such, in the current implementation the `parameter_selection_services.py` only contains code that is needed by the `select_parameter` script. The methods provided are: `result_table(), summary-_table(),process_model_parameter_selection()` and `post_process_model_para-meter_selection()`. For more details about what these functions do, please see the Docstrings in the code.

3.6.6 Plotting Directory

The plotting folder, includes three files: The `plot_export.py` file focuses on generating LaTeX tables, showing the Correct Classification Rate (CCR) across various False Positive Rates (FPRs). Additionally, the `plot_graph.py` file is designed to plot Open Set Classification Rate (OSCR) graphs for different protocols, creating figures with subplots for each protocol and displaying OSCR for 'Negative' and 'Unknown' classifications using a semilog scale for FPR. Lastly, the `plot_scoring.py` file with the `PlotScoring` class, focuses on outputting protocol scores directly in the console.

3.6.7 Util Directory

The `Util` directory contains files which are accessed from many different locations within the codebase, or where its generally ambigious as to where to place them. For instance, one could make the argument for putting the `plot_service()` method into the plotting folder. However, it is also needed in the `parameter_selection_service()` method, located in the `paramet-er_selection` folder. Similar arguments can be made about the `hyperparameters.py`, `load-_post_process_model.py` and the `config_services.py` files.

The last file deserves some special attention as it is responsible for managing the configuration files. It offers the `find_configs()` method which is concerned about retrieving configs. In addition, it also provides the `combine_yamls()` function, which calls the yamlparser [6] and combines two yaml dictionaries. The `find_configs()` and the `combine_yamls()` methods are both used by the `prepare_config()` function. This function is called at the beginning of the `run()` methods within the `train.py`, `select_parameters.py` and `evaluate.py` scripts to find and combine the required configuration files.

[6]The YamlParser can be found here: `https://github.com/AIML-IfI/yamlparser`

Chapter 4

Process

4.1 Collaboration

We followed an agile working approach, with two- to three-week sprints. During the sprint, we held several team meetings to align, define the tasks, split the workload, and discuss success moments and problems. At the end of each sprint, we held a meeting with Professor Manuel Günther. We presented our progress and answered open questions. Finally, we set the goals for the next spring. At a certain stage, we started documenting our process in this report. We used GitHub as platform for version control. Initially, we worked in a separate repository until we felt comfortable enough to push the first version to the AIML-IfI repository. This is the reason why, for the first changes in the project, the commits are not visible in the current repository.

4.2 Challenges faced during the project

One major challenge of the project was the inherent complexity of the topic and algorithms involved. This made it difficult to understand and modify code without the risk of breaking functionalities. In Addition, there were also challenges related to the University server Rolf. Rolf was used to test and run code with the ImageNet dataset. However, getting access to Rolf took longer than anticipated, so the team decided to implement the EMNIST dataset. This dataset, allowed quick testing on local machines. Once access was granted, new challenges arose. When testing on the Rolf server, one shares the resources with other students. As such, depending on the time of day, the "CUDA out of Memory" error was thrown, especially when training Resnet50 in combination with ImageNet.

4.3 Contributions

The following section provides a brief list of team members and their respective contributions to the package. It's important to note that there may be overlaps, and multiple team members may have contributed to certain parts. As such this is not a complete list of tasks done. It simply highlights the main/focus contributions of the team members. For a more detailed overview, the commit history on Github[1]. can be examined.

[1]Repository of the refactored codebase: https://github.com/AIML-IfI/osei

Nicolas Kohler

- Designed and Implemented the modular and extensible model-backbone architecture as described in section 3.3. To prove the modularity, Nicolas also added two alternative backbone architectures, namely Alexnet and LeNet, which are described in the subsection 3.3.1.
- Developed a dynamic way of writing and retrieving checkpoint locations and filenames for model and evaluation files. In Addition, Nicolas also handled platform dependent issues and completed a final code-refactor in January, which touched many parts of the project again.
- Tested the training, evaluation, parameter selection and plotting implementations on the university server "Rolf" and collected the results as seen in Figure 5.1 of subsection 5.1.
- Main Writing contributions:
  - Primer 1.2
  - Network section 3.3, incl. all subsections
  - The subsections 3.5.1 and 3.5.4 in the Scripts section 3.5
  - The subsections 3.6.1, 3.6.3, 3.6.4, 3.6.5 and 3.6.7 in the section Other Code Directories 3.6
  - Overview of the current execution status section 5.1

Dean Heizmann

- Overhauled and developed the new Dataset architecture and implemented EMNIST as a new alternative for a dataset. The new dataset architecture is elaborated on in the subsection 3.2.
- Refactored code related to the configuration files, as described in 3.1.
- Refactored the `train.py` and `train_all.py` scripts. They had to be transformed to work with all the new implementations of other parts.
- Main Writing contributions:
  - Abstract and Motivation 1.1
  - Evolution of architecture 2
  - The sections about configuration files 3.1 and datasets 3.2
  - The subsections 3.5.2, 3.5.3, 3.5.4,
  - The collaboration process 4.1
  - The Readme A
  - The conclusion 5

David Lebrec

- Adapted `workers` functions to EVM, OpenMax, Proser, Threshold. More information about the workers can be found in the section 3.4.
- Refactored the `losses` part as described in 3.6.2.

- In Addition, David also conducted various refactorings throughout the project. These include parts of the `script` 3.5, `metrics` 3.6.3, `util` 3.6.7, `plotting` 3.6.6, `parameter_selection` 3.6.5 and `osr_algorithms` 3.6.4 folders.
- Main Writing contributions:
  - Parts of the Introduction 1.2.1
  - Losses 3.6.2
  - Workers 3.4
  - Challenges faced during the project 4.2
  - Next Steps 5.2
  - Diagrams

Chapter 5

Conclusion

5.1 Overview of current execution status

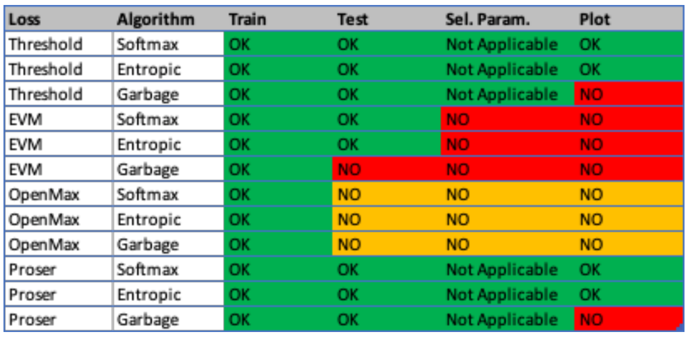
As was hinted at during previous sections (especially in the script section 3.5), not all user commands that trigger the execution of a script and corresponding processes work. Figure 5.1 provides a general overview of commands in combinations with different Losses and Algorithms. In order to save on computing resources and time, it was decided to only train with the small LeNet backbone for one epoch. All the experiments were done on the ImageNet dataset and training was done solely by using the train command (i.e. the train_all command was not used). To indicate what works and what does not work, three different colors were used:

- **Green** indicates that the command is working and runs through without errors.
- **Orange** means that the command is not working. In Addition, testing the command on the baseline implementation also yielded errors. As such, the command **might** have already been broken when starting the project.
- Similar to Orange, **Red** also means that the command is not working. However, for these commands we were not able verify whether or not they previously worked on the baseline implementation. As such their functionality **might** have been broken by the refactoring conducted as part of this project.

While it was tried to get everything to work, achieving a fully working implementation was not possible. This is partly due to the fact that it was not manageable to fully verify, whether or not all parts of the baseline worked as intended. Also, for verifying whether or not the recent changes introduced bugs, it would have been required to test on the University server. This often was not possible, due to lack of available server resources. Another reason is the complexity of the code and algorithms involved, as properly integrating them could be a thesis on its own. However, training seems to work for all combinations (using the standard train command). That is key, as this was our focus and since testing, evaluation and plotting errors might be easier to address.

Given the complexity of certain parts of the code, it is recommended that the specific bugs are addressed by people who have an in-depth knowledge about how the approach should work and how it should be implemented. This way it can be ensured that the fixes deliver results that make sense and not simply make the code run. If efforts are undertaken to further fix and improve the code, it is highly recommended to take the more detailed version of Figure 5.1 into account. It describes exactly which commands where used and which errors occurred. It is named `command_line_inputs_status.xlsx` and can be found on the Github repository[1].

[1]Repository of the refactored codebase: https://github.com/AIML-IfI/osei



Loss	Algorithm	Train	Test	Sel. Param.	Plot
Threshold	Softmax	OK	OK	Not Applicable	OK
Threshold	Entropic	OK	OK	Not Applicable	OK
Threshold	Garbage	OK	OK	Not Applicable	NO
EVM	Softmax	OK	OK	NO	NO
EVM	Entropic	OK	OK	NO	NO
EVM	Garbage	OK	NO	NO	NO
OpenMax	Softmax	OK	NO	NO	NO
OpenMax	Entropic	OK	NO	NO	NO
OpenMax	Garbage	OK	NO	NO	NO
Proser	Softmax	OK	OK	Not Applicable	OK
Proser	Entropic	OK	OK	Not Applicable	OK
Proser	Garbage	OK	OK	Not Applicable	NO

Figure 5.1: Status of Commands

5.2 Next Steps

The following three subsections lay out potential steps for further improvements to the repository. They are divided into open tasks that should be tackled in the short-, medium- and long term.

5.2.1 Short Term

Validation Metrics The `ValidationMetrics` class, is currently not used by the implementation. It is not exactly clear why that is the case. It is strongly suspected that the methods `auc_score_binary()` and `auc_score_multiclass()` are important for the plotting step. As such, these parts require further investigation.

Minor Naming Conventions: After freezing the code some minor naming errors popped up:

- Filename: plot_all.py - Command: plot.py
- Filename: select_parameters.py - ClassName: ParameterSelection
- Filename: plot_services.py - ClassName: PlotHelper
- Filename: plot_exporting.py - ClassName: PlotReporting

Pathing and Checkpoint Naming At the moment, model checkpoints are saved based on a dynamically created path (if `use_dynamic_output_directory` in the config file is set to true, which is the default). As such a potential model checkpoint might be saved as follows: `experiment/imagenet/protocol_1/imagenet_lenet_softmax_threshold_best.pth`. While the `snapshot_services.py`, currently acts the "Naming" Manager, the `config_services.py` is handling the file location. It would be nicer, to have all things related to pathing and the naming of generated files in one point. By doing so, one could also incorporated the naming for results and plots.

Investigating Util As elaborated on in the util section 3.6.7, the directory currently holds files that can not be attributed to one specific task. In general, they do not fit properly into the project structure. As such, it might make sense to investigate these files in the future.

5.2.2 Medium Term

Datasets: Encapsulating the logic that adapts datasets according to loss functions and open-set classification algorithms would significantly improve the simplicity of adding new datasets. At the same time, however, it is also difficult to implement this conceptually and practically, since the appearance, structure, and finally, implementation of each dataset heavily differs from one to another. Because of these difficulties, the issue fell out of scope for the project.

EMNIST: In the current implementation, the usage of the EMNIST dataset has not proven useful in all cases. This would have to be investigated.

Workers: Currently, two worker functions carry out the training of the models. In the future, this could be either combined into one worker function that handles every training, or the worker function for training base models is separated from the worker that trains with open-set classification algorithms. This would especially be important to extend the code with new open-set classification methods.

Bugs: There are bugs related to the evaluation and plotting of trained models with certain combinations of algorithms and loss functions. While some of the issues might have been transferred from the initial state of the package, others might have been introduced by the refactoring itself. Fixing these issues was out of the scope of the project and also not part of the main objectives. For an overview of all detected bugs, consider Figure 5.1.

5.2.3 Long Term

Automated Testing and Logging: At the moment the project does not contain any test cases. This can obviously lead to untested code which can break functionality. As such, one might introduce automated test cases and in-depth logging information in the future. One might also consider to make use of Github Actions[2]. This would allow to automatically run tests and conduct formatting/linting when pushing code to the main branch.

Introduction of new Networks, Datasets and Algorithms: In the future, more datasets and neural network architectures (for instance: EfficientNet (Tan and Le, 2019)) can easily be added, as discussed throughout this report. Additionally, new open-set classification approaches can be introduced. However, this would eventually need further modifications to the codebase. In the same manner, new loss functions can be added.

5.3 Conclusion

The primary objective of this project was to restructure the codebase to make it more intuitive, user-friendly and easy to update, while also simplifying future modifications and extensions. This was achieved by moving different parts into their own directories based on functionality. To accomplish this, the previously rather concentrated and simply structured codebase was transformed. This involved breaking down the code into well-defined classes, functions, and helper services. Each component was organized into its own directory, ensuring encapsulation, and assuring single responsibility. To achieve this, some parts of the code were partly or completely

[2]Github Actions: `https://github.com/features/actions`

rewritten or extended. As a result, the reorganization not only enhanced code clarity but also facilitated easier maintenance and scalability.

The secondary objective was to extend the code by a new dataset and two new neural network architectures. The EMNIST dataset is now available for training, alongside the refactored ImageNet dataset implementation. EMNIST is much smaller, allowing for test runs and experiments without having access to the much larger ImageNet dataset.

Additionally, customized LeNet and pytorch AlexNet backbones are now supported as well. They offer a less time- and resource-consuming option compared to the existing ResNet50 network, allowing for faster more diverse experiments in the future. However, it is important to mention that specific model/loss/algorithm combinations result in execution failures, as discussed throughout the report and especially in 5.1.

In conclusion, the project was successful in taking the codebase to a state where it is ready for future commitments and improvements. However, there is still room to further streamline expansion efforts, which is unsurprising given the current package's size. In addition, future work should also focus on resolving bugs concerning specific model / loss / algorithms. The current state of the implementation can be found on Github[3].

[3]Repository of the refactored codebase: `https://github.com/AIML-IfI/osei`

Appendix A

README - Open-Set Image Classification Comparison

Please Note:
The following README can also be found in the Github Repository on: https://github.com/AIML-IfI/osei

A.1 README

This open source package allows the investigation and comparison of various approaches for open-set image classification, as introduced in the paper "Large-Scale Open-Set Classification Protocols for ImageNet" presented in WCAV 2023 (Palechor et al., 2023).

The implementation currently supports training models on `ImageNet` with three different protocols. Additionally, an experimental `EMNIST` Dataset is included. This smaller dataset allows for the quick testing of code changes on local machines, without having to access the huge `ImageNet` dataset. The available algorithms are `EVM`, `Garbage`, `OpenMax` and `Proser`. Applicable loss functions are `Softmax`, `Entropic Open-Set Loss` and `Garbage Loss`. Furthermore, `Resnet50`, `Alexnet` and a customized `LeNet` are supported network architectures. The package structure provides an easy way to extend it with additional datasets, algorithms, loss functions or neural network architectures in the future. If this package, or any of the evaluation protocols provided by it, are used, please cite as follows:

> @inproceedingspalechor2023openset,
> author = Palechor, Andres and Bhoumik, Annesha and Günther, Manuel,
> booktitle = Winter Conference on Applications of Computer Vision (WACV),
> title = Large-Scale Open-Set Classification Protocols for ImageNet,
> year = 2023,
> organization = IEEE/CVF

More information about the aforementioned Algorithms, such as `OpenMax`, `EVM` and `PROSER` can be found in the following paper:

@articlebisgin2023large,
title = Large-Scale Evaluation of Open-Set Image Classification Techniques,
author = Bisgin, Halil and Palechor, Andres and Suter, Mike and Günther, Manuel,
journal = **under submission**,
year = 2023

A.1.1 LICENSE

This code package is open-source based on the BSD license. Please see LICENSE A.1.1 for details.

A.1.2 Disclaimer

As of Early 2024, not all script commands work for all dataset/networks/losses etc. combinations. Please find an overview of the current state in the `current_functionality.xlsx`.

A.1.3 Setup

Version Requirements

- Python Version: Python 3.8.4
- Conda Version: Conda 23.11

Setup Steps

The `environment.yaml` file contains a conda installation script to install all dependencies. The creation and activation commands are as follows:

- `conda env create -f environment.yaml`
- `conda activate osc_comparison`

Also make sure unzip the protocols. More Information about protocols can be found in the corresponding section of this README.

- `unzip protocols.zip`

Setup Pitfalls

- Osc_comparison initialization failure: Make sure to add a pytorch version installed that is compatible with the GPU drivers.
- FileNotFoundError, "Train File not found!": Make sure to unzip the protocols. ‘unzip protocols.zip‘
- ModuleNotFoundError, No module named osc.datasets: Can be fixed by running ‘pip install -e‘
- Local Cuda drivers too old Error: Make sure to have the latest version of conda.[1]

[1]The latest cuda version can be downloaded here: https://docs.conda.io/projects/miniconda/en/latest/

A.1.4 Data

The current implementation supports two datasets – ImageNet ILSVRC 2012 (full support) and Emnist (partial support). ImageNet needs to be downloaded separately and must be accessible by specifying its location in the `data_path` argument of the `imagenet.yaml` file. EMNIST will automatically be downloaded once at runtime. Further datasets can be added, as described later in this README.

ImageNet

The ImageNet used in this package relies on the ILSVRC 2012 data and can be downloaded from Kaggle [2]. In this implementation, the ImageNet dataset class is constructed using a CSV file. The file lists all the paths to the images, as well as the corresponding label. E.g.:

- `val/n02100583/ILSVRC2012_val_00013430.JPEG, 0`

The CSV files are included in the package. The actual images need to be downloaded and made accessible by the code. Look at the protocol section further below for more information.

EMNIST

The EMNIST dataset is downloaded directly from the `torchvision.datasets` library and is saved into a folder in the project directory. The implementation uses lazy initialization, which means the dataset is downloaded only when asked for specific training, validation or test data of the EMNIST dataset. The Dataset uses MNIST as known samples and EMNIST letters as unknowns. EMNIST is a rather experimental dataset for now. Its benefit is its small size compared to ImageNet, which allows to test training approaches quickly.

Adding a new Dataset

To add a new dataset, a new config file has to be created, in which the path to the dataset is specified. This new config should then be added to the `configs/datasets` folder. Additionally, one has to create a new class for the new dataset and let it inherit from the `Base_Dataset` class. The new class must contain certain modifications according to the algorithm and loss functions used.

Caution

Some deep network architectures like AlexNet might not perform well on small image sizes (like the ones from EMNIST). That is because typically, each convolution and pooling layer is reducing the dimension of the input, which can eventually result in a zero or even negative dimension. These issues are addressed in the Backbone implementation.

A.1.5 Protocols

There are three protocols implemented[3], which are designed to create artificial open spaces with different levels of similarity between known and unknown classes in ILSVRC 2012. They also

[2] ImageNet ILSVRC 2012 can be downloaded here: `https://www.kaggle.com/competitions/imagenet-object-localization-challenge/overview)`

[3] More Information about the Protocols can be found here: `https://openaccess.thecvf.com/content/WACV2023/html/Palechor_Large-Scale_Open-Set_Classification_Protocols_for_ImageNet_WACV_2023_paper.html`

contain negative classes for algorithm comparison, creating three levels of difficulty for the open-set classifier. In practice, this results in three sets of CSV files, where each set contains a CSV file for the training, the test and the validation data. The CSV files themselves contain the paths to the ImageNet images, as well as their corresponding label.

- Protocol 1: Focuses on semantically unsimilar known (dogs) and unknown classes (non-animals).
- Protocol 2: Focuses on animal classes in both known (hunting dogs) and unknown classes (Four-legged, fury animals).
- Protocol 3: Focuses on semantically similar known and unknown classes. Involves ancestors of classes as both, known and unknown.

The protocols rely on the robustness library which themselves relies on some files that have been dispensed in the context of the ImageNet dataset. These files are not available anymore and given the considerable time that has passed since, it might prove to be difficult to locate them. The files are:

- imagenet_class_index.json
- wordnet.is_a.txt
- words.txt

If the files can not be found, pre-computed protocol files are provided in the protocols.zip folder and can be extracted via:

- `unzip protocols.zip`

Caution

In the current implementation, only the ImageNet dataset supports the usage of protocols. Training with EMNIST requires a protocol as an input, however it has no practical impact.

A.1.6 Networks

The project currently supports three different Backbone Model Topologies: Resnet50, Alexnet and a customized LeNet variant. They can be found in `/networks/backbone/custom` and `/networks/backbone/torchvision` respectively. These backbones can be paired with the Threshold Model, the Garbage Model or the Proser Model. As part of handling the networks, the `network_services` are provided, which shall be used to create and retrieve models, enabling loose coupling. In Addition, we also provide `snapshot_services`. They main task of these services is to save and load models.

Adding new Backbone

Adding a new Backbone is straight forward and only requires the following few steps:

- Adding the backbone either to `networks/backbone/custom` or `/networks/backbone-/torchvision`.
- Letting the new backbone inherit from the `Backbone` class and add `first_blocks()` and `last_blocks()` functions to make it compatible with Proser.
- Adapting the `__init__.py` in `/backbone`
- Adapting the `_get_backbone()` function in `network_services.py`

Adding new Model

Adding a new Model is a bit more complex but in theory, should only require the following:

- Adding the new model to `/models`.
- Adapting the `__init__.py` in `/models`.
- Adapting the `network_services.py` as needed.
- Potentially creating specific worker classes etc.

A.1.7 Scripts

The package provides four base capabilities that form a pipeline of processes from training a model to plotting its performance on open-set tasks. These processes, in short, involve:

1. The training of the base model as well as the application of an algorithm on top of that base model.
2. Only for EVM and OpenMax: Parameter optimization.
3. Evaluation to extract performance statistics.
4. Plotting using extracted statistics.

All processes are executed via command line inputs. The script files that are run and orchestrate the processes lie in the `osc/script` directory and are automatically installed and runnable. The entry points for the command line inputs are defined in `setup.py`. For more information about input parameters for each individual command, check the according subsections in the following parts.

Train

In order to run an experiment with an algorithm, a threshold model has to be trained first. In a second step, an OSR algorithm can be executed on top of the trained threshold model. Make sure to use the protocol, loss function, and network architecture in the threshold model, that you also want to use later on.

The training process relies on the parameters given in the dataset and the algorithm configuration files. The choice between datasets and algorithms is straightforward and done via the command line input by indicating the name of the config file to use, e.g. `datasets/imagenet.yaml`, `algorithms/threshold.yaml`. Specific parameters about the training process, such as the loss function, the step size, or the network architecture can be set by changing the respective value in the algorithm config file under `configs/algorithms`. The train command is as follows:

```
train.py [algorithm] [dataset] [protocol]
```

Please note that the parameters are order-dependent and can not be left out or re-arranged. Once a base model is trained, other algorithms, such as EVM or OpenMax, can be executed in the same way. The base model does not have to be selected, it will automatically be found. As already mentioned, the network architecture, protocol, and loss function have to match between base model and algorithm configuration.

For an overview of all possible parameters use:

```
train.py -help
```

Caution It is advised to specify a GPU using -g to pass a GPU index. If not provided, the training will use CPU only and the process might take very long.

Usage examples:

- Train a base model on the imagenet dataset with protocol 1 on GPU index 5:

  ```
  train.py algorithm/threshold.yaml dataset/imagenet.yaml 1 -g 5
  ```

- Apply the EVM algorithm on top of the base model on the imagenet dataset with protocol 1 on GPU index 5:

  ```
  train.py algorithm/evm.yaml dataset/imagenet.yaml 1 -g 5
  ```

- Train a base model on the EMNIST dataset with protocol 1 on GPU index 7:

  ```
  train.py algorithm/threshold.yaml dataset/emnist.yaml 1 -g 7
  ```

Train all

Alternatively to the `train.py` script, the `train_all.py` script provides the option to train models with multiple loss functions on different protocols for any algorithms given. The script will access the config file for each given algorithm and execute a training process for each possible value of the loss function parameter (`Softmax`, `Garbage`, `Entropic`) and possible protocol (`1`, `2`, `3`) as default. Currently, it is not implemented to train on various datasets simultaneously, which means it is necessary to indicate on which dataset to train:

```
train_all.py [dataset] -algorithms [algorithms]
```

If no algorithm is specified, it will default to all available algorithms (`Threshold`, `Openmax`, `Evm`, and `Proser`). If only a subset of available loss functions or protocols should be used for processes, it is done by indicating which ones to use in the command line input:

```
train_all.py [dataset] -algorithms [algorithms] -losses [losses]
-protocols [protocols]
```

For an overview of all possible parameters use:

```
train_all.py -help
```

Caution - Training several models needs a lot of computation power. If you don't have several GPUs at hand, expect to wait about a week for results. Running this script on CPU only is practically useless. - As for training single instances, always make sure to train an applicable threshold model first!

Usage example:

- Train base models for all loss functions and all protocols on the imagenet dataset on the GPU on index 7:

  ```
  Train_all.py -algorithms threshold -dataset dataset/imagenet.yaml -g
  7
  ```

- Train EVM and OpenMax models on the imagenet dataset with loss functions softmax and entropic, on protocols 1 and 3, on the GPU on index 7:

  ```
  Train_all.py -losses softmax entropic -algorithms evm openmax
  -protocols 1 3 -dataset dataset/imagenet.yaml -g 7
  ```

Parameter Selection

Some of the algorithms will require adapting the parameters to the different loss functions and protocols. Particularly, EVM and OpenMax have a set of parameters that should be optimized. Due to the nature of the algorithms, the `train_all.py` script has already trained and saved all parameter combinations as provided in the configuration files of these two algorithms. Now the task is only to evaluate the algorithms on unseen data. Naturally, the known and the negative samples of the validation set are used to perform the parameter optimization.

The parameter optimization will be done via the `select_parameters.py` script. It will read the configuration files of the EVM and OpenMax algorithms, load the images from the validation set, extract features with all trained base networks (as given by the `-losses` parameter), and evaluate the different parameter settings of the algorithms. Particularly, the CCR values at various FPR thresholds will be computed. Depending on the protocol, this might require several minutes to hours. Finally, it will write a separate LaTeX table file per protocol/algorithm/loss combination, and summarize LaTeX tables including the best parameters for each algorithm.

Caution Note that also the Proser algorithm has parameters that we might want to optimize. However, since this would require a complete network finetuning for each parameter/protocol/algorithm combination, we do not include Proser in this script.

At the moment some combinations of losses and algorithms fail during parameter selection. Check Figure XYZ for more information.

Usage example:

- Run the parameter selection for the EVM algorithm on the imagenet dataset using the SoftMax loss function, protocol 1, and GPU index 1. In this case different directory for config files was given. The default is the config directory of this package:

  ```
  select_parameters.py -dataset imagenet -protocols 1 -algorithms evm
  -losses softmax -configuration-directory some_directory -g 1
  ```

Evaluation

Evaluation is done using the `evaluate.py` command. Like the `train_all.py` command, if no parameters are given, per default the evaluation script will evaluate every valid trained model it finds. It is always necessary to provide the dataset in the form of `datasets/[datasetname]-.yaml`.

```
evaluate.py [dataset]
```

Alternatively, this can be restricted to a subset of models trained with given algorithms, loss functions, and protocols using the according parameters in the command.

```
evaluate.py [dataset] -algorithms [algorithms] -losses [losses]
-protocols [protocols]
```

The script will extract the features, logits, and scores for the test dataset and save the results in the `osei/experiments` directory. Additionally, the evaluation script considers a configuration file, per default the `configs/tests/test.yaml` file. This file contains more parameters, including the network architecture and optimized parameters for validation for each individual algorithm. For an overview of all possible parameters use:

```
evaluate.py -help
```

Caution At the moment some combinations of losses and algorithms fail during evaluation, so it is recommended to only evaluate what is needed. Check Figure XYZ for more information.

As for training, it is optimal to run the evaluation process on a GPU, even though the task is eventually not as heavy as the training is.

If the error: "Invalid device string: 'cuda:None'", shows up, please select a GPU.

Usage example:

- Evaluate threshold models trained on ImageNet with softmax loss and protocol 1. Only use the best models:
  ```
  evaluate.py -losses softmax -algorithms threshold -dataset
  datasets/imagenet.yaml -use-best -protocols 1
  ```

Plotting

Finally, the `plot.py` command makes use of the evaluation data extracted by the `evaluate.py` command to create plots and tables. It will use the information from the same `configs/tests-/test.yaml` file for the process. The results are plotted into a PDF file, `Results_last.pdf`, made of several pages:

- Page 1: Contains all OSCR plots for all algorithms applied to networks trained with all loss functions, where both negative and unknown samples are evaluated for all protocols.
- Page 2+: Contain the score distribution plots for each algorithm, but page wise separated for each loss function.

Again, per default the script will search for any possible combination of algorithm, loss function, and protocol and plot the evaluation data of each combination. If the combination is not found, it will simply be skipped. It is necessary to provide the dataset in form `datasets/[dataset-name].yaml`.

```
plot.py [dataset]
```

To have the script only consider a subset of combinations, it can again be done using the according parameters in the command line input:

```
plot.py [dataset] -algorithms [algorithms] -losses [losses]
-protocols [protocols]
```

For an overview of all possible parameters use:

```
plot.py -help
```

Caution

- Plotting always requires an evaluation file.
- At the moment some combinations of losses and algorithms fail during plotting. Check `current_functionality.xlsx` for more information.

Usage example:

- Run the plotting for threshold models trained on imagenet dataset using the softmax loss function, and on GPU index 1. Use the best models only for each combination: `plot.py -protocols 1 -losses softmax -configuration configs/tests/test.yaml -data imagenet -use-best -algorithms threshold -g 1`

Appendix B

Diagram

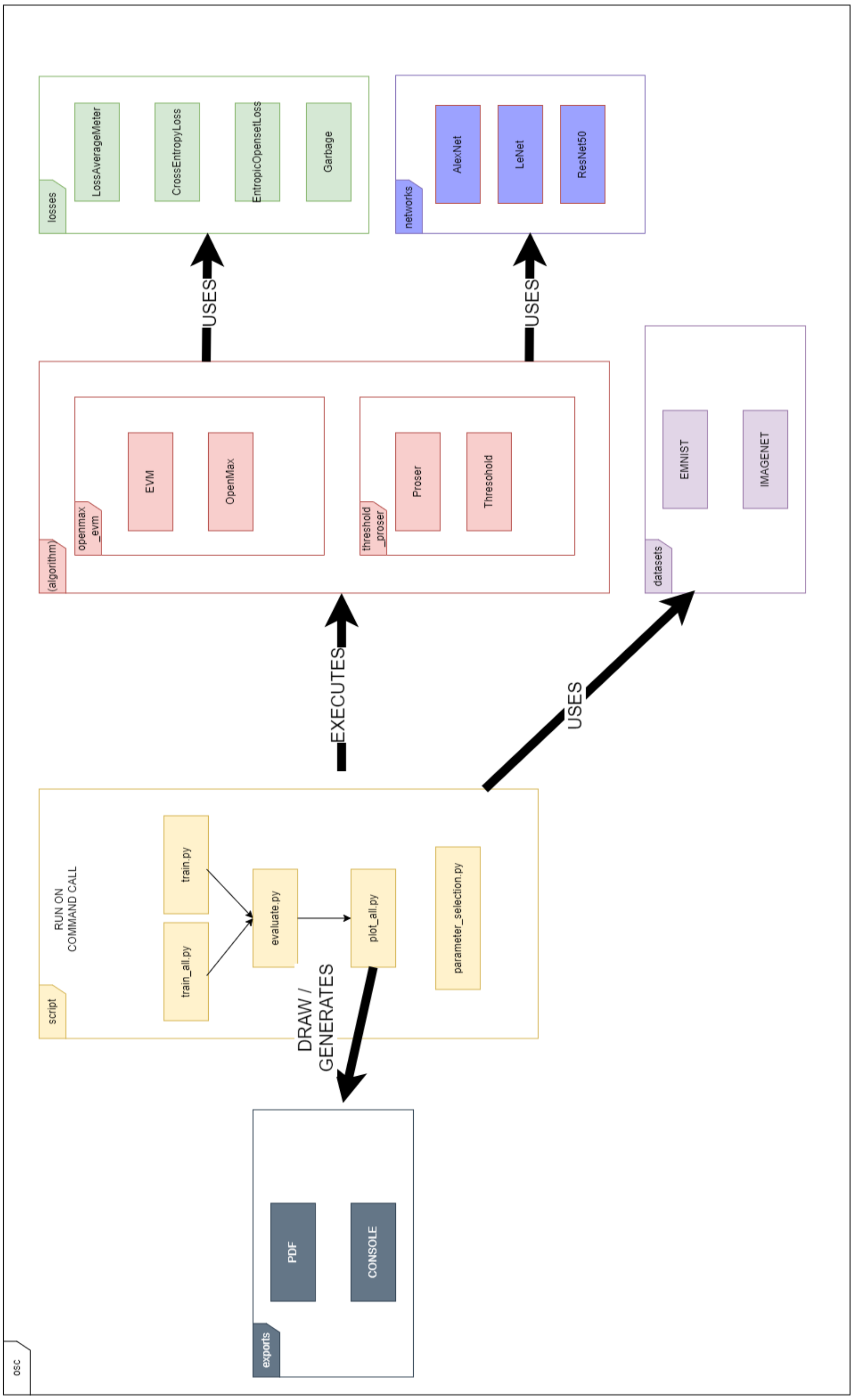


Figure B.1: High-level interaction of scripts, algorithms, networks, losses and dataset components.

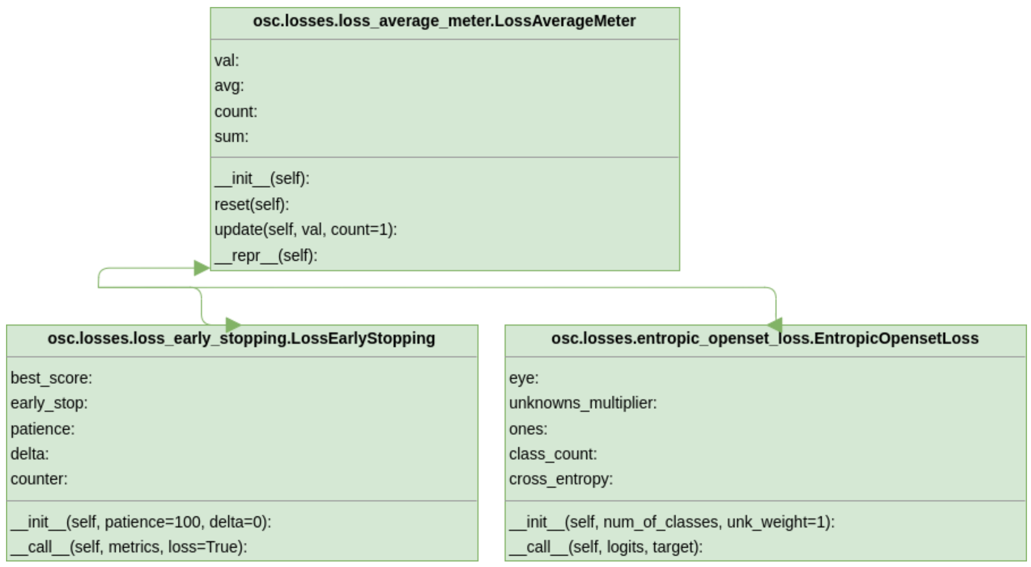

osc.losses.loss_average_meter.LossAverageMeter
val:
avg:
count:
sum:
__init__(self):
reset(self):
update(self, val, count=1):
__repr__(self):
osc.losses.loss_early_stopping.LossEarlyStopping
best_score:
early_stop:
patience:
delta:
counter:
__init__(self, patience=100, delta=0):
__call__(self, metrics, loss=True):
osc.losses.entropic_openset_loss.EntropicOpensetLoss
eye:
unknowns_multiplier:
ones:
class_count:
cross_entropy:
__init__(self, num_of_classes, unk_weight=1):
__call__(self, logits, target):

Figure B.2: Provides an overview of the different losses.

List of Figures

Bibliography

Bendale, A. and Boult, T. E. (2016a). Towards open set deep networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.

Bendale, A. and Boult, T. E. (2016b). Towards open set deep networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.

Bisgin, H., Palechor, A., Suter, M., and Günther, M. (2023). Large-scale evaluation of open-set image classification techniques. *under submission*.

Cohen, G., Afshar, S., Tapson, J., and van Schaik, A. (2017). Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.

Dhamija, A., Günther, M., Ventura, J., and Boult, T. E. (2020). The overlooked elephant of object detection: Open set. In *Winter Conference on Applications of Computer Vision (WACV)*, pages 1021–1030.

Dhamija, A. R., Günther, M., and Boult, T. E. (2018). Reducing network agnostophobia. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 9157–9168.

Elharrouss, O., Akbari, Y., Almaadeed, N., and Al-Maadeed, S. (2022). Backbones-review: Feature extraction networks for deep learning and deep reinforcement learning approaches. *arXiv preprint arXiv:2206.08016*.

Ge, Z., Demyanov, S., and Garnavi, R. (2017). Generative openmax for multi-class open set classification. In *British Machine Vision Conference (BMVC)*.

Geng, C., Huang, S., and Chen, S. (2021). Recent advances in open set recognition: A survey. *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 43(10):3614–3631.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *International Conference on Computer Vision (ICCV)*, pages 1026–1034.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hendrycks, D., Basart, S., Mazeika, M., Zou, A., Kwon, J. M., Mostajabi, M., Steinhardt, J., and Song, D. (2022). Scaling out-of-distribution detection for real-world settings. In *International Conference on Machine Learning (ICML)*. PMLR.

Hendrycks, D. and Gimpel, K. (2017). A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *International Conference on Learning Representations (ICLR)*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, Y., Mao, H., Girshick, R., and He, K. (2022). Exploring plain vision transformer backbones for object detection. In *European Conference on Computer Vision*, pages 280–296. Springer.

Lyu, Z., Gutierrez, N. B., and Beksi, W. J. (2023). Metamax: Improved open-set deep neural networks via weibull calibration. In *Winter Conference on Applications of Computer*.

Matan, O., Kiang, R., Stenard, C., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., Jackel, L., and Le Cun, Y. (1990). Handwritten character recognition using neural network architectures. In *4th USPS advanced technology conference*, volume 2, pages 1003–1011.

Neal, L., Olson, M., Fern, X., Wong, W.-K., and Li, F. (2018). Open set learning with counterfactual images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 613–628.

Oza, P. and Patel, V. M. (2019). C2ae: Class conditioned auto-encoder for open-set recognition. pages 3–4. IEEE.

Palechor, A., Bhoumik, A., and Günther, M. (2023). Large-scale open-set classification protocols for ImageNet. In *Winter Conference on Applications of Computer Vision (WACV)*. IEEE/CVF.

Rudd, E. M., Jain, L. P., Scheirer, W. J., and Boult, T. E. (2017a). The extreme value machine. *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*.

Rudd, E. M., Jain, L. P., Scheirer, W. J., and Boult, T. E. (2017b). The extreme value machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(3):762–768.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3).

Ryota Yoshihashi, Shaodi You, W. S. M. I. R. K. T. N. (2019). Classification-reconstruction learning for open-set recognition. In *The University of Tokyo, Data61-CSIRO*.

Schaub, N. J. and Hotaling, N. (2023). Assessing efficiency in artificial neural networks. *Applied Sciences*, 13(18):10286.

Scheirer, W. J., de Rezende Rocha, A., Sapkota, A., and Boult, T. E. (2012). Toward open set recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(7):1757–1772.

Sun, J. and Dong, Q. (2023). A survey on open-set image recognition. *arXiv preprint arXiv:2312.15571*.

Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR.

Varghese, D. (2018). Comparative study on classic machine learning algorithms. `https://towardsdatascience.com/comparative-study-on-classic-machine-learning-algorithms-24f9ff6ab222`. Accessed: 2024.

Vaze, S., Kai Han, A. V., and Zisserman, A. (2022). Open-set recognition: A good closed-set classifier is all you need? In *conference paper at ICLR 2022*.

Zhou, D.-W., Ye, H.-J., and Zhan, D.-C. (2021). Learning placeholders for open-set recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.