Thesis-Summary

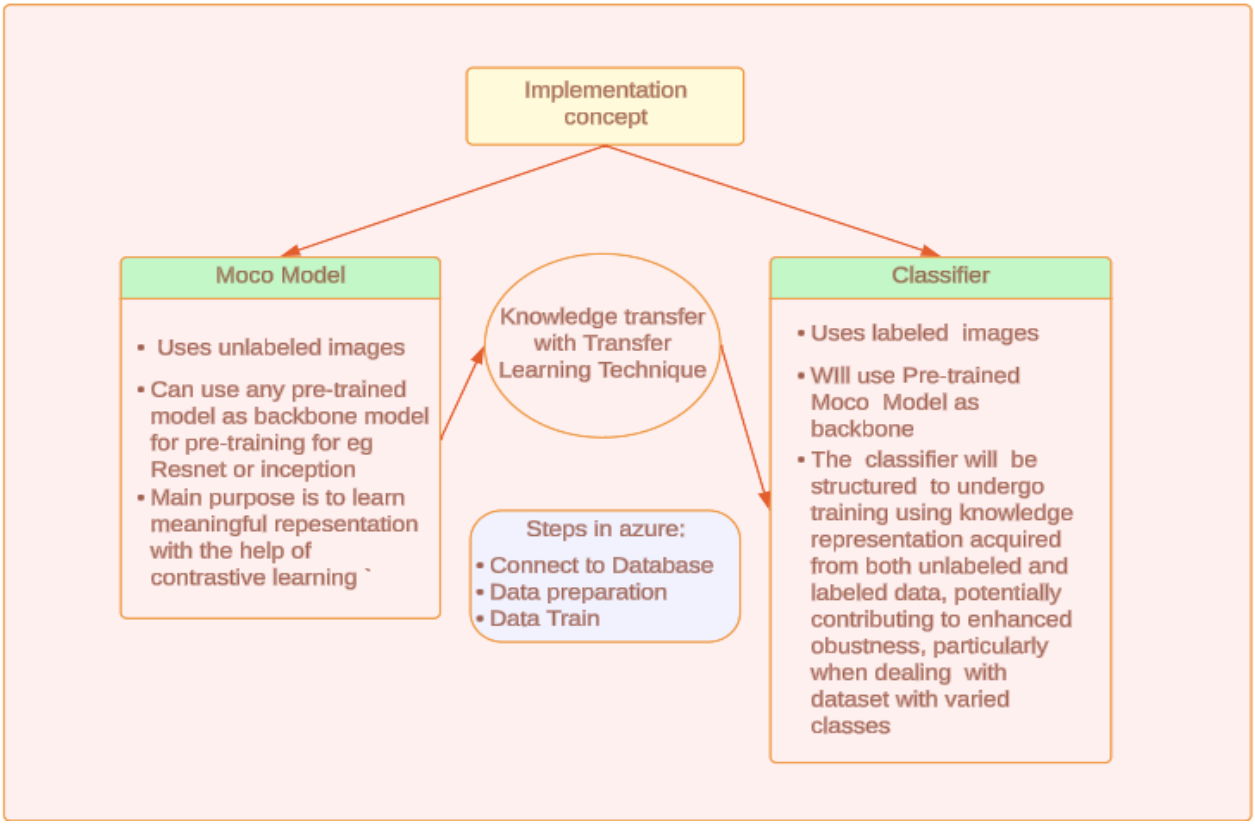
This document consists of summarized version of my thesis to help it understand faster. The full report can also be found under Full_report folder which gives more information about the literature review and pre-processing step needed to train the model.

ABSTRACT

This master's thesis investigates the potential of self-supervised pre-training for feature extraction in hierarchical image classification tasks, focusing on the utilization of large amounts of unlabeled images. Current practices involve training separate models on limited data at each hierarchy level, leading to increased computational effort and potentially reduced performance. We propose a domain-familiar backbone Convolutional Neural Network (CNN) that can be pre-trained on a vast dataset of unlabeled images to produce high-quality embeddings, thereby reducing computational efforts across all hierarchy tiers. Our approach involves gathering and pre-processing all available unlabeled images, training a self-supervised image embedding's model such as Momentum Contrast V2 (MoCoV2), and employing this model for fine-tuning in the hierarchical image classifier. This study highlights the robustness of the trained embedding space, which is less sensitive to class distribution imbalances and can be used for image similarity comparisons to aid in labeling. The proposed method seeks to enhance the efficiency and performance of hierarchical image classification tasks by leveraging the power of self-supervised pre-training.

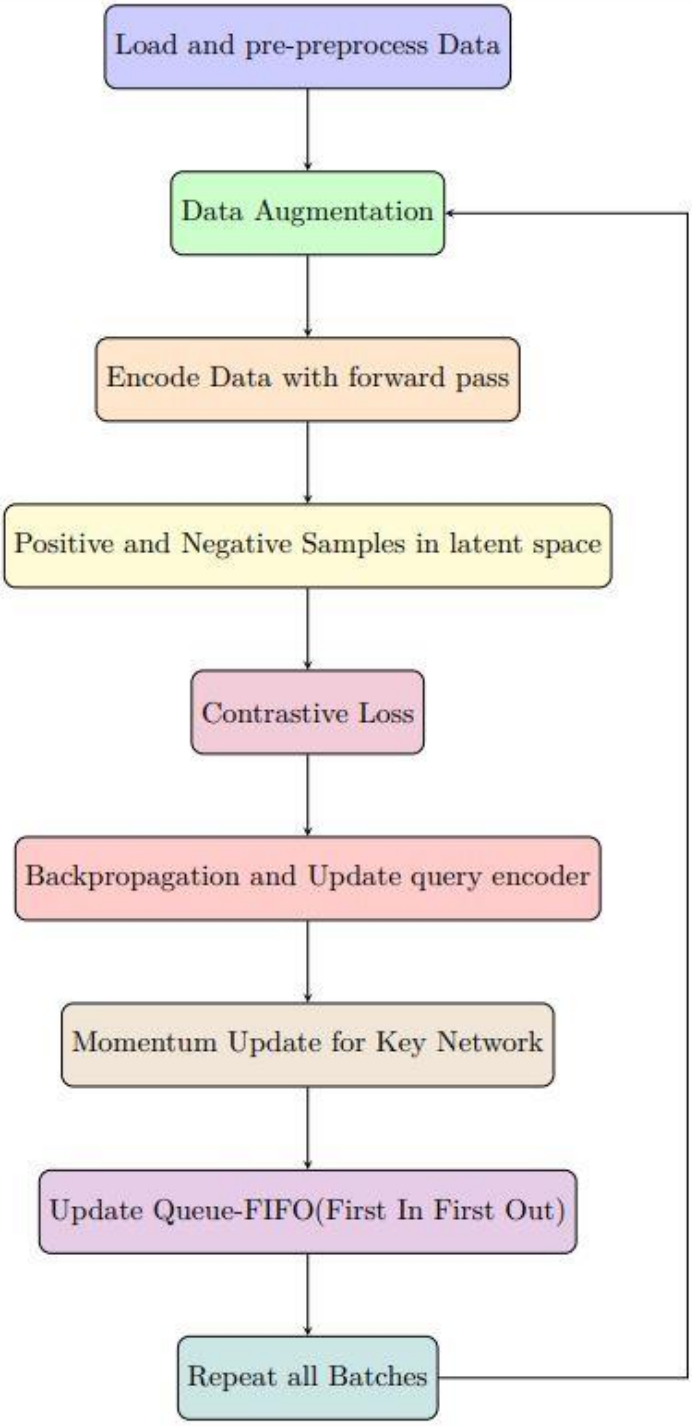
How will It be done?

The implementation phase consists of creating of Momentum contrast model and modifying the existing classifier model along with the application of judicious transfer learning methodologies.



What is Momentum Contrast Model?

The momentum contrast model created uses unlabelled data for learning meaningful representations with the help of contrastive learning. It can be viewed as a dynamic dictionary look-up process by integration of a queue and moving-averaged encoder. The dynamic dictionary created on the fly with these encoders provides an evolving source of negative images crucial for contrastive learning as well as providing the stability to model with a moving average encoder enabling the model to learn intricate visual patterns without too much dependence on labeled data. In rather simple terms, MoCo is a technique enabling a machine learning model to glean insights from unlabeled images by constructing a visual pattern "dictionary" and employing an intelligent approach to image comparison for learning. The figure shown below illustrates the steps involved in creating the Moco V2 model and its detailed implementation is discussed in model training.


Load and pre-preprocess Data
Data Augmentation
Encode Data with forward pass
Positive and Negative Samples in latent space
Contrastive Loss
Backpropagation and Update query encoder
Momentum Update for Key Network
Update Queue-FIFO(First In First Out)
Repeat all Batches

The actual implementation aligns with the three primary stages in Azure :

- Connecting to Database (connect_db_MoCo.py)
- Data preparation (‘data _prep_ moco.py‘)
- Model training

To orchestrate the flow of the model for these steps an execute_moco.py file is created. It also manages pipeline parameters, authentication, configuration settings, and the creation of a docker container for setting up the Python environment for the machine learning job.

Connecting to Database (‘connect_db.py‘)

This module or script plays an important role in preparing unlabelled dataset effectively by connecting to Azure Cosmos Database. With the help of command line arguments, a custom query for Cosmos DB is passed as a pipeline parameter from the execute file. The output of this step is a training set (CSV file) containing image ids of unlabeled data.

Data preparation (‘data_prep_moco.py‘)

The output of ‘connect db‘ is essentially input for ‘data prep‘ step. This script as shown in figure below downloads and pre-processes images from an Azure Blob Storage container which will be used in the training step. The script has ’load data function’ which reads CSV file containing image IDs from the previous step and a ’get image size’ function which determines the image size required for model training based on the PyTorch model. Furthermore more ’get image’ function asynchronously downloads and resizes images from Azure Blob Storage.

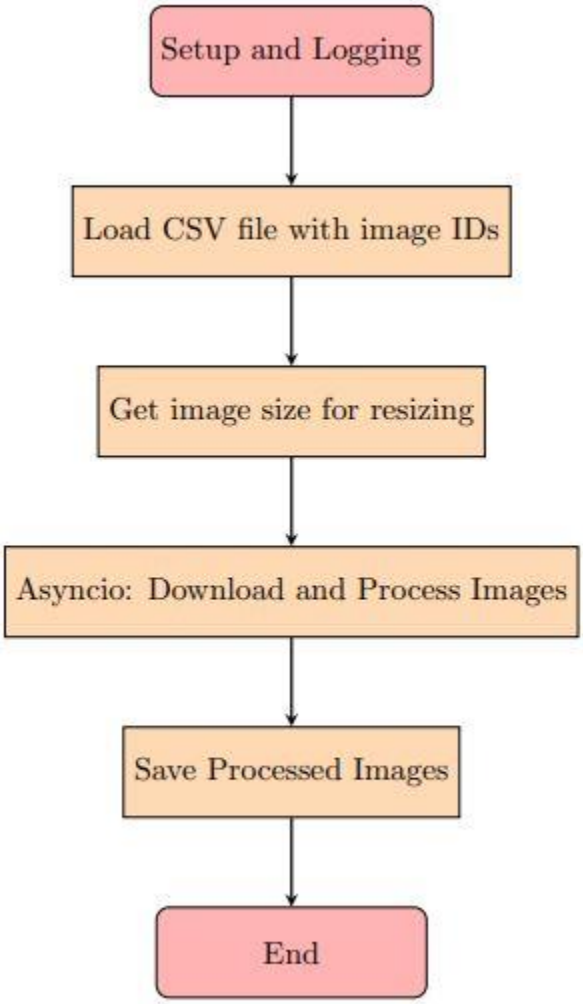
Model Training:

After the preparation of data, this module focuses on training and building of MoCo V2 model. The workflow is segmented into three distinct stages, namely:

1. Data loading and Transformation (‘datamodule_moco.py‘)
2. Momentum Contrast model (‘moco_model.py‘)
3. Encapsulating and saving the model (‘training_moco_pipeline.py‘)
4. Data loading and Transformation (‘datamodule_moco.py‘)

datamodule_moco.py

The Data module script handles the loading of images from (‘data prep‘) step and prepare batches of images after Moco V2 transformations. These transformations are applied randomly to each image in the dataset.



For each image, two transformed versions of the same image are obtained as the output of MoCo V2 transforms. They are known as a query image and a key image. These pairs represent positive samples for contrastive learning while the negative images are the remaining images in the batch.

moco_model.py

This script contains the logic for building moco_model as shown in pseudo-code. To understand the concept in detail please refer to section 5.1.3 in the thesis of kushal shah.

Algorithm 1 Pseudocode of MoCo in a PyTorch-like style.

```
# f_q, f_k: encoder networks for query and key
# queue: dictionary as a queue of K keys (CxK)
# m: momentum
# t: temperature

f_k.params = f_q.params # initialize
for x in loader: # load a minibatch x with N samples
   x_q = aug(x) # a randomly augmented version
   x_k = aug(x) # another randomly augmented version

   q = f_q.forward(x_q) # queries: NxC
   k = f_k.forward(x_k) # keys: NxC
   k = k.detach() # no gradient to keys

   # positive logits: Nx1
   l_pos = bmm(q.view(N,1,C), k.view(N,C,1))

   # negative logits: NxK
   l_neg = mm(q.view(N,C), queue.view(C,K))

   # logits: Nx(1+K)
   logits = cat([l_pos, l_neg], dim=1)

   # contrastive loss, Eqn.(1)
   labels = zeros(N) # positives are the 0-th
   loss = CrossEntropyLoss(logits/t, labels)

   # SGD update: query network
   loss.backward()
   update(f_q.params)

   # momentum update: key network
   f_k.params = m*f_k.params+(1-m)*f_q.params

   # update dictionary
   enqueue(queue, k) # enqueue the current minibatch
   dequeue(queue) # dequeue the earliest minibatch
```

bmm: batch matrix multiplication; mm: matrix multiplication; cat: concatenation.

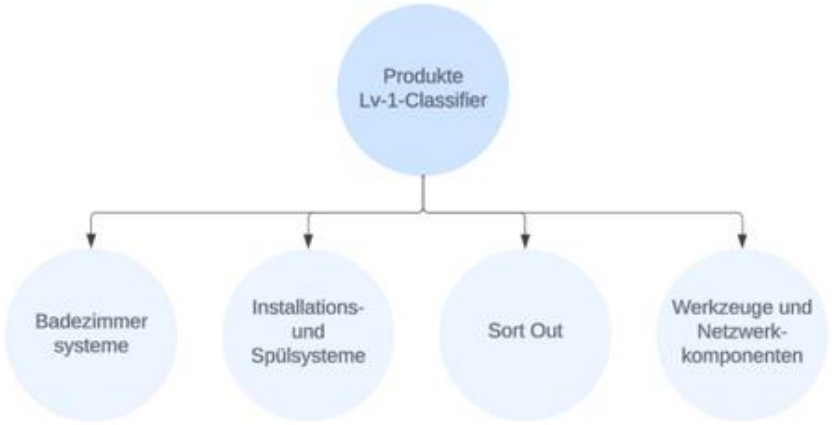
training_moco_pipeline.py:

This script serves as an entry point of the training process. It specifies the input path for data to be used by the MoCo model and configures training epochs. Monitoring the training progress and defining key metrics, such as contrastive loss, is handled within this script. Additionally, it establishes the output path and saves the model after training. The MoCo model created is saved in the Azure model store which can be later used as the base model for the classifier. The output folder contains a checkpoint file with details about the weights and parameters of each layer. The plot for contrastive loss can also be found here.

Classifier Model

The classifier model serves as the concluding component of the implementation phase. The learned representations from the pre-trained MoCo model are passed to this model with the help

of transfer learning for the further classification process. It consists of connected layers for the downstream classification task. The goal is to make the first-level classifier for the following categories:



It is important to note that ‘Sort-Out‘ is not any product in the Geberit catalog but these are images with no useful information, hence it is better to sort them out in the first level. The important steps in the classifier model can be aligned with Azure implementation as follows:

1. Connecting to Database (‘connect_db.py‘)

2. Data preparation (‘data_prep.py')

3. Model training ('evaluation_moco.py & training_training.py‘)

connect_db.py and data_prep.py steps work in the same way as it is working currently. In the classifier model, it handles labeled data, unlike the MoCo model.

Model Training:

Model training consists of two important scripts: evaluation_moco.py and training_training.py. The evaluation_moco.py is primarily a copy of the transfer_model.py script. However, instead of building the model from Tim library, it downloads the Moco V2 model from the Azure model store. The model and its specific version can be specified in the ‘build’ function. The model’s desired version can be downloaded as a checkpoint file and saved in the output path of the training step. As described in pseudo-code, only the query encoder without an MLP head is attached to the linear classification head from the checkpoint other parameters are neglected. Pytorch does not save the model’s hyperparameter in the checkpoint file, as a result, while instantiating the query model class, the same hyperparameters used during pre-training were passed as arguments. The resulting class will act as the base model. The weights of this base model are frozen and only fully connected layers of the classification head are trained with the labeled data.

Successful transfer learning of learned representations relies on knowing the architecture of the base model thoroughly, as the output dimensions or features of the last layer of the base model are the input for the first fully connected layer. The classification head consists of one linear layer, ReLU activation, dropout, and another linear layer in the same order and similar to the one used in the conventional model. It is kept relatively simple and unchanged because this project is a preliminary study and the classes to identify are small in number.

To simplify, if the training process uses a base model from the Pytorch library it will evaluate the performance of the conventional model and if it uses the MoCo V2 model it will evaluate results with pre-training. The evaluation metrics or other aspects of both models remain unchanged.

Results and evaluation:

In conclusion, this thesis has successfully demonstrated the effectiveness of MoCo in image classification. Through experimental analysis, the study confirms that MoCo as a self-supervised learning approach, can efficiently learn meaningful representations from unlabelled data and can perform better than the traditional supervised learning method. It outperforms traditional supervised methods in terms of enhanced generalization, F1 score, accuracy, precision, and recall as summarized in the table. For detailed analysis please refer to Chapter 6 of the thesis

Table 7.1: Performance Metrics Comparison with base model as Resnet18

Method	Precision	Recall	F-1 Score	Accuracy
Self-supervised	0.72	0.73	0.72	0.82
Self-supervised with oversampling	0.74	0.75	0.74	0.86
Supervised Learning	0.54	0.50	0.46	0.78

Future work:

In the future, enhanced GPU capabilities and better quality of labeled data can aid in performance. Another potential approach could be to adopt data parallel processing techniques. By scaling up the training script with the Horovod(https://horovod.readthedocs.io/en/stable/summary_include.html#why-horovod) package and distributing batches across a cluster of nodes simultaneously, computing time could be substantially reduced. For example, if training 300,000 images on a single node of a GPU cluster takes 10 days, parallelization could potentially complete the computation in 2 days by utilizing 5 nodes, with the cost remaining constant and dependent on the number of nodes employed.

With the successful completion of the preliminary phase, experiments can be conducted using a higher volume of unlabeled data to enhance overall performance Exploring the fine-tuning of the classification head or MoCo and incorporating batch normalization might be able to enhance the results. Building on findings from SimCLR[32], which indicate that wider networks lead to

improved contrastive loss, it is suggested to use broader networks as backbones for various experiments.

Looking ahead, an immediate future step for this study involves integrating the MoCo pipeline into hierarchical classifiers and conducting inferences to assess performance at different levels of the hierarchy.

Trained Moco Model:

After several iterations finally, MoCo model was trained on all unlabeled data in the cosmos_db database. It took almost 10 days to train on all the unlabeled data 😮.

- ***All the details of code cannot be shared for the development of pipeline because it belongs to geberit, however some of the training part can be found under the code section.***