Towards Improving Generalizability By Using Self Supervised Learning Techniques for Deep Learning Models in Multi-Sensor-Based Human Activity Recognition

Master Thesis

Helge Hartleb (1684818)

submitted to the
Ubiquitous Computing Group
Prof. Dr. Kristof Van Laerhoven
University of Siegen

Supervisor:
M. Sc. Marius Bock
Computer Vision Group
University of Siegen

October 2025

Statutory declaration

I affirm that I have written my thesis (in the case of a group thesis, my appropriately marked part of the thesis) independently and that I have not used any sources or aids other than those indicated, and that I have clearly indicated citations.

All passages that are taken from other works in terms of wording or meaning (including translations) have been clearly marked as borrowed in each individual case, with precise indication of the source (including the World Wide Web and other electronic data collections). This also applies to attached drawings, pictorial representations, sketches and the like. I take note that the proven omission of the indication of origin will be considered as attempted deception.

_____	_____
Location, Date	Signature

Abstract

Human Activity Recognition (HAR) is a steadily growing field of interest for many sectors of the computer science industry. Whilst originally focused on visual data for activity recognition, the field has since branched into different research categories. With widely available sensors in wearable devices such as smart watches, smartphones, earbuds, etc., a research field has emerged that centers its attention around HAR based on sensor data rather than visual data. Predominantly in sectors like sports and healthcare, the demand for accurate activity tracking has increased over the past decades. Especially, the use of multiple data streams from varying body positions grew in popularity. This brings forth a new challenge to achieve good recognition performance regardless of the placement of the inertial origin sensor.

This thesis extends an existing self-supervised deep learning framework for sensor-based HAR (ColloSSL [17]). The framework applies contrastive training in a multi-sensor scenario. Three modifications are made: multi-anchor training, multi fine-tuning evaluation, and adaptation layers with regulatory loss. Further, a generalizability training option is introduced. The proposed extensions are evaluated on two benchmark datasets, *RealWorld* and *PaMap2*, with a focus on adaptability and generalizability. A measure is proposed for the assessment of these characteristics. The results show that multi fine-tuning enables the training of a single model with strong cross-domain performance, reducing the need for device-specific fine-tuning. Adaptation layers increase adaptability but lead to higher variance, indicating a trade-off between specialization and stability. In contrast, multi-anchor training does not consistently outperform the baseline, particularly on smaller datasets, suggesting limits to its effectiveness.

These findings demonstrate that targeted extensions of self-supervised learning frameworks can improve robustness in HAR and contribute to the development of more reliable, device-independent model for human activity recognition. The code of the implemented changes can be accessed at `https://github.com/hel-har/master_thesis`.

Contents

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

The goal of Human Activity Recognition (HAR) is to detect and correctly categorize activities performed by a person while analyzing sensor data. Initial approaches were mainly aimed at HAR for visual data. But with the growing popularity of wearables (i.e., smart-watches, smartphones, etc.), the field of sensor-based HAR increased in relevance. These wearables contributed to the widespread use of inertial data for HAR [25]. This data consists mostly of accelerometer and/or gyroscopic data, which is recorded in three channels (one for each axis of acceleration/rotation). Even with over a decade of research, the task of accurately classifying this inertial data still remains challenging due to the complexity of human movements and the differences in body motion based on the placement of the sensors [3, 43]. Recording human activity data, whilst made easy with the large supply of wearable sensors, is often a very time-consuming task due to the manual labeling of recoded datasets. Since sensor data is not easily recognizable by the human eye, most recordings have to be accommodated by a camera setup to accurately label the recorded data. To resolve this issue, many recent HAR approaches focus on the use of unsupervised or self-supervised deep learning. This machine learning approach focuses on the use of unlabeled data for the majority of its training. Therefore, enabling the use of widely available unlabeled data for training [44].

Unsupervised deep learning focuses on representation learning of *features*. Features can be understood as a sort of data patterns that are distinct for specific devices and activities. An unsupervised deep learning model makes use of multiple layers of data reduction operations (here we use convolutions) that concentrate input data to a *feature space*. This encoding of data into feature vectors is done by a *feature extractor* [4]. The learned features are then fed into a classifier that learns to match the features to certain activities. When combining both the feature extractor and the classifier, one gets a model that can predict an activity based on

input data with a certain probability [4].

Self-supervised learning serves as a middle ground between supervised and unsupervised deep learning. The main training effort is completed by an unsupervised (pre-) training of the feature extractor with an additional labeled fine-tuning after attaching the classifier [13]. This training approach combines the easier data usage of unsupervised training with the performance benefits of supervised training, whilst keeping the usage of labeled data to a minimum. More recent research [6, 21, 17] focuses on the use of multiple sensor positions to either improve the recognition performance of a specific sensor [17] or strengthen the robustness to data from different sensor positions [6, 21]. There are two main ideas for the use of multiple different data distributions for HAR. The first idea is that the use of data from multiple different sensor positions can act as a sort of data augmentation. Therefore, the training benefit lies in a greater robustness to potential noise or position changes of a given sensor [39]. The second Idea is *Contrastive Learning* which is a deep learning method that was originally proposed for visual representation learning [7]. It was adopted into the commonly used *SimCLR* framework [7], which was later adapted to work with sensor-based data for HAR using different approaches [19, 39]. Contrastive learning tries to leverage commonalities and differences between different sensor data distributions to learn more precise features for one specific device. This is achieved by providing positive and negative pairs during learning. The positive pairs can, for example, be augmented data segments of the same source data. Therefore, encouraging the model to learn a more robust representation of these segments by applying a contrastive loss that rewards close representations of positive pairs and penalizes close representations of negative pairs (e.g., segments from different activities) [19].

We aim to evaluate a framework (ColloSSL [17]) that extends contrastive learning to multi-sensor environments regarding adaptability and generalizability (see Section 1.2 for definitions of both phrases). Afterwards, we target the improvement of both abilities by implementing a variety of changes to the training, fine-tuning, and architecture. These changes focus particularly on the idea of producing a single model with good adaptability and generalizability without the need for training or fine-tuning for specific devices. This "one-model-fits-all" approach is based on the assumption that there is a feature space from which accurate predictions regarding activity classes can be made, regardless of the origin device.

1.1 Related Work

HAR originally started with visual data. In the early 1990s first approaches for the use of inertial body-worn sensors were implemented [5]. With promising results

with reduced data requirements compared to visual HAR, the field rapidly grew in interest. In the 2000s, early results by *Maurer et al.* [28] already showed a significant increase in classification performance from the use of multiple sensors. These results sparked an evolving research domain for the past two decades [5]. Especially for more complex, full-body, activities (e.g., sports or activities of daily living (ADLs) like cooking) the use of multiple sensor positions can provide additional information to improve recognition accuracy of such activities. There have been many approaches regarding the use of multiple sensors for HAR [2], with a substantial amount of them focusing on the use of wearables [30]. However, many of these approaches mainly use sensor fusion by combining different modalities in the input of a model rather than using the entire data from a number of different devices [30].

A novel approach for multi-sensor-based HAR was proposed by *Jain and Tang, et al.* [17]. They extend the contrastive learning idea to multiple devices to an approach called *collaborative learning*. Collaborative learning forms the positive and negative pairs from samples of different devices instead of augmented segments of a single device. All devices are grouped into positive and negative devices. To form positive pairs, time-synchronized samples from the positive devices are used. For negative samples, the data from negative devices is desynchronized to minimize the chance of sampling data from the same activity class. Their collaborative self-supervised framework (*ColloSSL*) receives input data from multiple devices. One of the devices is chosen as the *anchor device* for which the model is trained. All other devices are then categorized as positive or negative based on the similarity of their data distribution to the *anchor devices*. As the similarity measure, ColloSSL uses the maximum mean discrepancy (MMD) [37]. Using these positive and negative samples, feature embeddings are computed for the positive/negative sample as well as for the anchor device. Using these embeddings, a multi-view contrastive loss is computed. This loss computes the cosine similarity of the anchor features and the positive and negative features. The similarity to the negative features is weighted with the inverse MMD to enhance the impact of more dissimilar devices on the loss. For a more detailed explanation of the multi-view contrastive loss, refer to the original source [17]. The loss encourages a high similarity between anchor and positive features and a low similarity between anchor and negative features. Hence, pushing features of positive devices closer to and features of negative devices further from the anchor features [17]. The model achieved state-of-the-art performance on multiple HAR datasets [33, 34, 38].

Another contrastive learning approach was brought forward by *Xu et al.* [42]. It focuses on a novel way of sampling positive pairs from one or multiple data streams. Their model focuses on retrieval-based reconstruction, which is an algorithm that samples pairs based on their similarities in certain *motifs*. Motifs are

patterns within the data distribution that different samples have in common. These patterns are used to reconstruct an *anchor* sample using a second, randomly chosen one. The similarity of the reconstruction to the *anchor* is then measured and positive pairs are formed accordingly [42].

An entirely other HAR research field that also tries to improve performance on multiple devices is domain adaptation. *Chang et al.* [6] and *Khan et al.*[21] propose techniques that aim to enhance the ability of a trained model to adapt to data from different *target* sensor positions (e.g., head, limbs, waist, etc.) while minimizing the loss in performance on the *source* sensor data [6]. To evaluate the performance of a model on a variety of sensor positions, [6] and [20] propose different measurement techniques. Instead of transferring a pre-trained model to an already known target dataset, an approach developed by *Lu et al.* [27] proposes a domain generalization technique. This approach focuses on the transference to a target dataset with an unknown data distribution. This is achieved by using distinct feature extractors to extract local and global features for different activity classes from the source dataset. For the distinction of local and global features, the data is split into different segments. Local features are features that are obtained from the extraction of a *single* segment via the use of 1D convolutions. Global features are obtained from *multiple* segments by using a 2D convolution over a given number of segments. Both feature types are aligned in their respective feature spaces (local/global) by penalizing large distances between features. However, the loss for local and global features is computed separately to not enforce alignment between local and global features. Both feature types are then concatenated to form the output features of the feature extractor. When applying the feature extractor to a target dataset, the newly learned local and global features are aligned to the already known features to improve performance on the unseen dataset [27].

1.2 Definitions

The two most important definitions that are used to a great extent within this thesis are:

- **Adaptability**: We define *adaptability* as the ability of a model with a given training dataset to perform accurate classification on a (sub)set of *known* training devices.
- **Generalizability**: We define *generalizability* as the ability of a model with a given training dataset to perform accurate classification on an (sub)set of *unknown* training devices. *Unknown* in this context means that no data of these devices has been used during training.

For both adaptability and generalizability, we distinguish two variations. For *regular* adaptability/generalizability, we allow the use of fine-tuning of the model for the evaluated devices. For *true* adaptability/generalizability, we do not allow fine-tuning for the evaluated devices. Within the scope of this thesis, when describing either of the two without the mention of *regular* or *true* prefix, it is considered as *regular* adaptability, respectively generalizability. Regarding the title of this thesis, generalizability includes both adaptability and the defined generalizability. The distinction between the two is merely used for ease of discussion for different training setups.

For discussions regarding our implementations and the base ColloSSL framework [17], we define the following phrases:

- **Anchor device**: This device serves as the anchor point for the contrastive sampling and is the device with which each pairwise MMD is computed.
- **Training device(s)**: All devices of which the data is used for the (pre-) training of the feature extractor.
- **Fine-tuning device(s)**: All devices for which the data is used for the fine-tuning process.
- **Evaluation device**: The device for which the model is evaluated (using the F1-score).

After establishing these definitions, we can define our research objectives.

Chapter 2

Technical Background

This chapter provides a more detailed overview of the most important concepts used in this thesis. We start with a deeper look into inertial Human Activity Recognition (HAR).

2.1 Inertial Human Activity Recognition

The field of inertial HAR focuses on the use of body-worn sensors, or otherwise called Inertial Measurement Units (IMU), instead of visual data for the task of activity recognition. These IMUs are often incorporated in wearables such as smart watches/bands, heart rate straps, etc. The two most commonly used types of IMUs are accelerometers and gyroscopes. These provide either acceleration or rotation data for three different axes. Common applications for inertial HAR are in the fields of healthcare, sports monitoring, and smart homes [10].

The use of different sensors, especially in the case of wearable devices within the research field, introduces the challenge of differing data modalities, such as sampling frequency, noise, and the placement of the IMUs. All of these variables have to be considered when investigating inertial HAR approaches. Further challenges are faced due to variability between different subjects when recording activity data. Complex movements will differ in their recorded characteristics from subject to subject or even within a given activity. Additionally, many activities can be performed in similar manners (e.g., climbing/descending stairs) with only subtle differences in movement patterns. And lastly, the performed activities may cause the sensors' positions to shift slightly, which results in somewhat differing inertial data. Some of these challenges can be reduced with the use of multiple sensor positions across the body. The additional data streams can record more subtle differences between activities by recording data from different body parts (e.g.,

ankle and wrist for walking vs. running). More sensor positions also reduce the possibility of orientation changes for the input data by providing redundancy in the case that the data from an IMU might not be considered useful for recognition [5]. After recording inertial activity data, the resulting dataset is preprocessed before it is used for training a model. Frequent preprocessing steps include filtering or interpolating missing data, normalization (per-user or per-sensor), and *windowing* the data. The process of windowing the data serves the purpose of reducing the size of a whole data stream into more manageable chunks. These data snippets are created with a fixed length and overlap with neighboring windows. This *sliding window* approach tries to maximize the expressiveness of the data windows. Since each window has the chance to miss important data parts due to its fixed length, the overlapping aspect of the created windows reduces the loss of data affiliation in the windowing process. The length of the windows can be chosen in accordance with the considered activities and their associated movement patterns (e.g., squats probably are performed in a larger window than steps while running) [5].

Activity recognition on a recorded dataset can be performed using a multitude of methods. Early approaches used handcrafted features from different domains (e.g., time domain: mean, variance, standard deviation, etc., frequency domain: Fast Fourier Transform (FFT)) and applied common machine learning techniques such as Random Forests [14] or K-Nearest Neighbors [9] to predict an activity from the crafted features. [2]. Later research was largely focused on the use of Deep Learning (DL) models that learn data representations, called *features*, to distinguish between activities. For this thesis, we focus on the explanation of DL models for HAR. There are two main classification approaches for DL models in HAR. First, the end-to-end learning. These models consist of a deep architecture that commonly uses convolutions [31], long short-term memory modules (LSTMs) [15], or transformers [16], and output a probability vector that predicts the activity that is most likely to be performed according to the input data. The second category of models splits the architecture and, therefore, the training into two different parts.

These DL models consist of a *feature extractor* which learns to map the input data into a *feature space* of a given dimension. This produces *feature vectors* which represent certain data characteristics that should be distinct for each activity class. The training of this feature extractor is often referred to as a *pretest task*. The generated feature vectors serve as the input for a *classifier* (also called *classification head*), which is trained to predict the activity classes based on the feature embeddings (vectors). This fine-tuning is also referred to as a *downstream task*. Figure 2.1 shows the two-stage training process of self-supervised deep learning models. The main advantage of applying the training in two separate steps lies in the reusability of the trained feature extractor. A pre-trained feature extractor

can be applied to a variety of datasets, which can greatly reduce the training time needed for comparable performance to an end-to-end trained model [26]. Another big advantage is the compatability of feature-embedding-based models with unsupervised and self-supervised deep learning approaches. These approaches reduce the need for labels in training datasets, which significantly reduces the cost/effort of dataset creation for these approaches. Since labels for inertial HAR datasets are almost exclusively produced by manual labor, the reduction of labels in training brings the inherent benefit of using unlabeled data, which is widely available due to the popularity of wearables. Therefore, increasing the potential activity training data by a significant amount. This reduction/omission of labels during training is achieved by leveraging different loss functions [35]. We focus on a prominent self-supervised learning approach that uses the idea of *contrastive learning*.

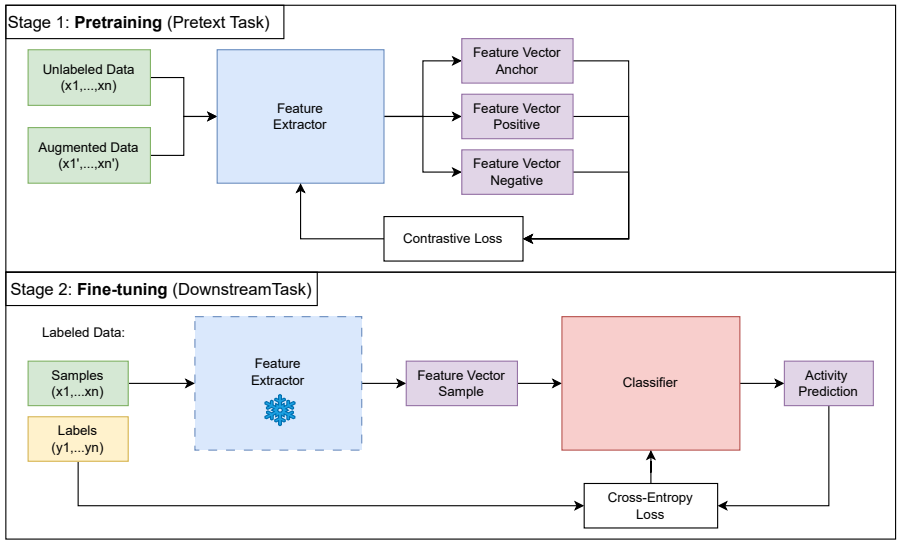


Figure 2.1: Illustration of the two-stage training process of self-supervised learning models for the example of contrastive learning.

2.2 Contrastive Learning

Contrastive learning is a deep learning method that was originally proposed for visual representation learning [7]. It was adopted into the commonly used *SimCLR* framework [7], which was later adapted to work with sensor-based data for HAR using different approaches [19, 39]. Contrastive learning tries to minimize the use

of labels during training of the feature extractor by leveraging commonalities and differences between sensor data samples to learn more precise features. During training, the feature extractor receives two kinds of samples, which are called *positive* and *negative pairs*. The idea behind the creation of the positive pairs is to provide the feature extractor with similar samples to align the encoding of their features. On the other hand, negative pairs should enforce distinct feature encodings for samples that are dissimilar [22]. Contrastive learning approaches often use data augmentation to produce positive samples, i.e., samples that have a different data distribution but relate to the same activity class and subject. These data augmentations include jittering/noising, cropping, permutation, and scaling of the original data, as well as masking, axis rotation, and channel dropout to generate different alterations of the same data segments [41]. Negative samples are mostly generated by picking random samples within a batch, as this randomness reduces the chance of picking a sample from the same activity class by accident. The notion to keep features from positive samples close together and features from negative samples far apart is formulated in the *triplet loss*.

Let x_i be an *anchor* data sample, i.e., the sample for which the positive and negative pairs are generated/sampled. Further, let x_i^+ and x_i^- be the corresponding positive and negative samples. We define

$$z_i = f(x_i),\ z_i^+ = f(x_i^+),\ z_i^- = f(x_i^+)$$

as the feature vectors produced by the feature extractor f for the anchor, positive, and negative samples, respectively. The triplet loss is defined as:

$$\mathcal{L}_{triplet}(x_i, x_i^+, x_i^-) = max(d(z_i, z_i^+) \ - \ d(z_i, z_i^-) \ + \ \alpha,\ 0) \tag{2.1}$$

Where d is a similarity measure for (feature) vectors, such as the cosine similarity, and α is a hyperparameter (margin) that can be set to a suitable value and serves as a lower bound for the dissimilarity between positive and negative embeddings [36]. This loss ensures a lower distance between the feature vectors of anchor and positive samples than between anchor and negative samples. However, the triplet loss only considers a single negative sample. As a solution, the triplet loss was extended to a more general contrastive loss. For each sample x_i in a batch of size B, the loss includes a set of negative samples (of size N) [22]:

$$\mathcal{L}_i = -\log \frac{\exp\left(\mathrm{sim}(z_i, z_i^+)/\tau\right)}{\exp\left(\mathrm{sim}(z_i, z_i^+)/\tau\right) + \sum_{j=1}^{N} \exp\Big(\mathrm{sim}(z_i, z_j^-)/\tau\Big)} \tag{2.2}$$

Where τ is a temperature parameter that regulates the specificity of the learned feature mappings. This sample-specific loss is computed for every sample in the

batch and averaged over the batch size, resulting in the following contrastive loss:

$$\mathcal{L}_{CL} = \frac{1}{B} \sum_{i=1}^{B} \mathcal{L}_i \tag{2.3}$$

During the pretraining, the feature encoder aims to minimize this loss. The following fine-tuning uses the learned feature embeddings to train a classifier. This is mostly done using the Cross-Entropy Loss [11]. The use of the contrastive loss pretraining of the feature extractor shows improvement in the generalization of the learned features across multiple subjects and/or datasets [7]. Open research questions regarding contrastive training include the sampling of positive and negative pairs and domain adaptation to different devices or sensor placements.

2.3 Collaborative Learning - ColloSSL

We notice that the discussed contrastive learning approach focuses on the use of a single sensor device. An extension of the contrastive learning approach to multi-sensor setups was proposed by *Jain and Tang et al.* [17]. Their developed collaborative self-supervised learning framework *ColloSSL* can receive multiple input data streams and apply the concept of contrastive learning to achieve more reliable feature mappings and achieve state-of-the-art performance on multiple well-known HAR datasets [38, 33, 34]. ColloSSL implements a form of the two-stage training described in the previous section. The framework takes an *unlabeled*, time-synchronous dataset as input. Let $D = \{D_1, ..., D_n\}$ be a dataset containing time-synchronous data from n different devices. ColloSSL aims to use the data from each device $D_i \in D$ to train a feature extractor that learns feature embeddings for an *anchor device* D^*. Afterwards, a classification head is trained using *labeled* data and the learned feature mappings in a fine-tuning stage. The fine-tuning of the classifier was achieved using cross-entropy loss [17]. The main focus of ColloSSL lies in the training of the feature extractor. To expand contrastive training to multiple devices, the training protocol (especially, the sampling and loss) was extended. Similar to the use of an anchor sample, the *anchor device* serves as the reference against which other devices are compared. Using Maximum Mean Discrepancy (MMD), all devices in $D \setminus D^*$ are ranked. The closest device is chosen as the *positive device* and **all** devices serve as *negative devices* (their experiments showed the best results for this device split) [17].

After grouping the devices into positives and negatives, the data from negative devices is desynchronized. This serves the purpose of simplifying the sampling process. The positive samples are drawn from the still time synchronous positive device. This ensures that positive samples come from the same activity class as the

anchor sample. Negative samples are drawn from the non-time synchronous negative devices, minimizing the chance for negative samples from the same activity class. We notice that in the case of accidentally sampling from the same activity class, the negative sample is only one of multiple samples, which reduces its impact on the learned feature embeddings. Additionally, the impact of each negative sample is weighted by $\frac{1}{\text{MMD}}$. Hence, samples from negative devices that are "more similar" (lower MMD) compared to the anchor device have a higher impact on the computed loss. This ensures that samples from different activity classes but devices with similar data distributions are pushed further apart [17].

Let D^* be the anchor device, $D^+, D^- \subseteq D$, be the corresponding positive and negative devices and $z^*, \{z_i^+\}|_{i=1}^{|D^+|}, \{z_j^-\}|_{j=1}^{|D^-|}$ the feature vectors obtained from the anchor and respectively positive and negative samples. *Jain and Tang et al.* propose a new *Multi-view Contrastive Loss* (MCL) [17]:

$$\mathcal{L}_{MCL} = -\log \frac{\sum_{i=1}^{|D^+|} \exp\left(\text{sim}(z^*, z_i^+)/\tau\right)}{\sum_{i=1}^{|D^+|} \exp\left(\text{sim}(z^*, z_i^+)/\tau\right) + \sum_{j=1}^{|D^-|} \exp\left(\text{sim}(z_i, z_j^-)/\tau\right)} \tag{2.4}$$

The loss is computed for every sample z^* and averaged over the batch size. In comparison to the contrastive loss (Equation 2.3), the MCL includes the negative samples from their associated negative devices. The sum over multiple positive devices allows for flexibility in device selection. However, the use of a single positive device showed the best results according to their experiments [17].

Figure 2.2 shows the whole training pipeline of the ColloSSL framework. The architecture of the feature extractor consists of three one-dimensional convolution layers with dropout layers in between, and a (maximum) pooling layer forms the output. The classification head uses two fully connected (dense) layers. The first has 1024 nodes, and the second one forms the output layer, which is followed by a softmax to produce a prediction vector. During the fine-tuning stage of the collaborative training (stage 4 in the figure), the layers of the feature extractor are frozen with the exception of the last convolution layer. The dataset used for fine-tuning consists of the *labeled* samples from the anchor device. [17]. The ColloSSL framework also implements the option to train the feature extractor for multiple devices. This option causes the anchor device to be switched randomly between batches [18].

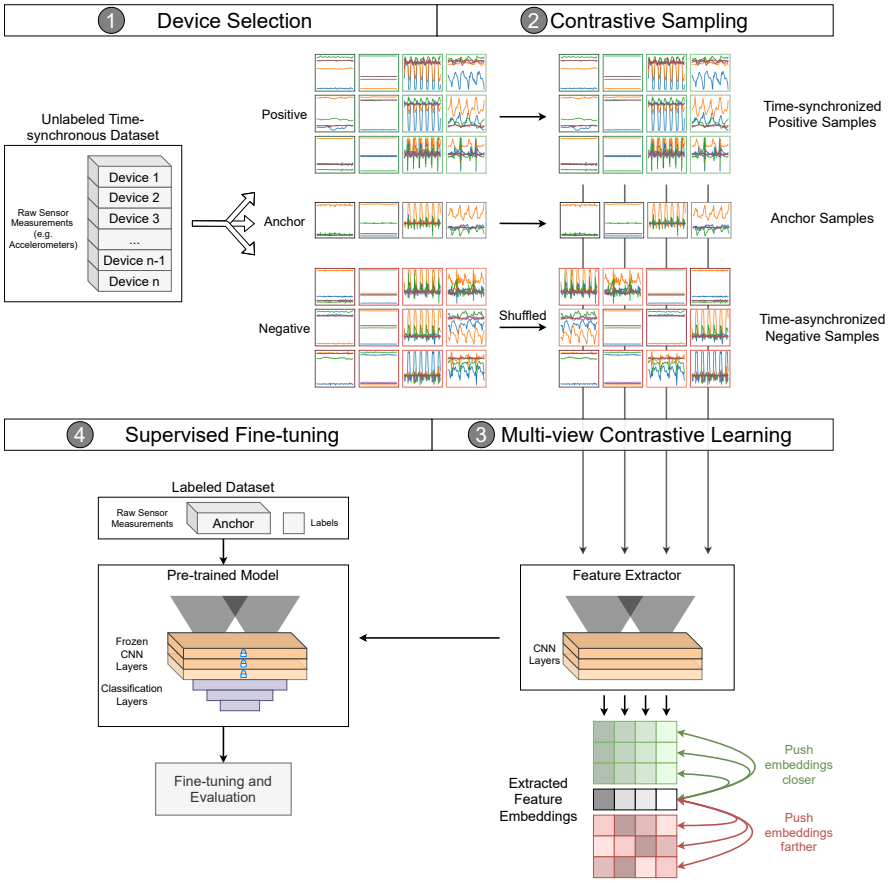


Figure 2.2: Overview of Collaborative Self-Supervised Learning (copied from [17]).

Chapter 3

Implementation

This chapter delves into the details of the extensions that were made to the original ColloSSL model. The code provided in the GitHub repository of the ColloSSL paper [17] was used as the starting point for further implementations. The ColloSSL framework is implemented in Python using TensorFlow 2 [1]. The repository provides a script for setting up a Docker [29] container with all necessary requirements. However, due to the use of the OMNI Cluster [32] of the University of Siegen, which prohibits the use of Docker directly all further implementations were conducted using Python 3.12.11 [12] and TensorFlow 2.16.1. The choice of version 2.16.1 for TensorFlow was made because of its added feature to enable deterministic behavior for GPU operations. Further details regarding this choice are discussed in Section 4.2. For better readability, the source code displayed in this chapter may be slightly adjusted. Though, the core functionalities remain intact.

Our changes focus on both parts of the ColloSSL training. In the pretext task, we implement a multi-device training approach and a new architecture option. In the downstream task, we introduce a new fine-tuning method that aims to train the classification head for multiple devices at once. A full illustration of the training pipeline is provided in Figure 3.1. Our additions to the original implementation are marked in green.

3.1 Multi-Anchor Training

The first addition we made to the ColloSSL framework is the option to train a model using more than one anchor device. The multi-anchor training is inspired by the thought of learning features not only from a single anchor during training, but rather from a multitude of anchors. This has the goal of reinforcing features that are shared by data from multiple devices and therefore increase the gener-

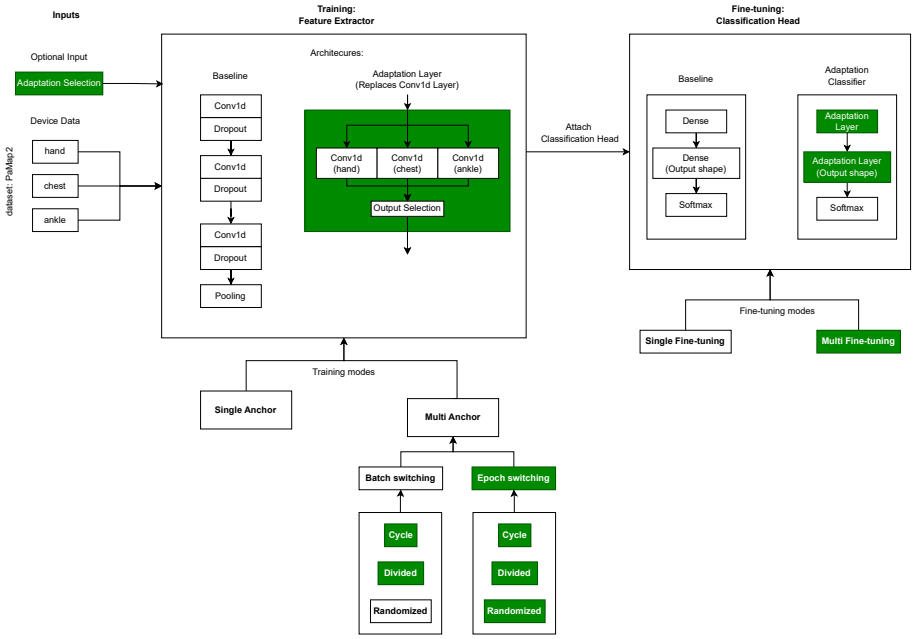


Figure 3.1: Illustration of the ColloSSL training pipeline. Our additions are marked with green color.

alizability of the learned feature embedding. For this, the option to change the anchor device during training was implemented. Another flag was added to the parser *multi_anchor*, which enables this training mode. Additionally, a parameter (*multi_anchor_training_mode*) was added to choose between three different multi-anchor training modes: *cycle*, *divided*, and *randomized*. Lastly, a list of anchor devices can be passed to the network using the *multi_anchor_list* parameter. The source code illustrating the device selection for each of the three modes can be seen in Source Code 3.1.

```
if args.multi_anchor_training_mode == 'divided':
    divided_epochs = epochs_total // num_devices
    for device in args.multi_anchor_list:
        args.multi_anchor_list = group
        args.train_device = args.multi_anchor_list[0]
        args.training_epochs = divided_epochs
        args.trained_model_path = train(dataset_full, args)

```

```
elif args.multi_anchor_training_mode == 'cycle':
    for i in range(0, epochs_total, args.multi_anchor_cycle):
        if i % num_devices == 0:
            rng.shuffle(multi_anchor_list)
        id = (i//args.multi_anchor_cycle)%num_devices
        args.multi_anchor_list = multi_anchor_list[id]
        args.train_device = args.multi_anchor_list[0]
        args.training_epochs = args.multi_anchor_cycle
        args.trained_model_path = train(dataset_full, args)

elif args.multi_anchor_training_mode == 'randomized':
    for i in range(0, epochs_total):
        args.multi_anchor_list = rng.choice(multi_anchor_list)
        args.train_device = args.multi_anchor_list[0]
        args.training_epochs = 1
        args.trained_model_path = train(dataset_full, args)
```

Source Code 3.1: Source code of the device selection for the multi-anchor training mode.

The *cycle* mode switches the anchor device within the given list after a specified number of epochs (using *multi_anchor_cycle*). For each epoch i, the training device *(t_device)* is given by:

$$t\_device(i) = multi\_anchor\_list[(i \;//\; multi\_anchor\_cycle) \;\%\; \#devices]$$

Where $//$ is the integer division, $\%$ is the modulo operator, and *#devices* is the number of devices within the *multi-anchor list*. The list is shuffled between each full cycle to reduce potential bias towards devices that are near the end of the list (since those devices would be used for training near the end).

Divided mode creates training blocks of equal size and therefore divides the training equally between all devices on the list. For each epoch i, the training device is given by:

$$t\_device(i) = multi\_acnhor\_list[(i \;//\; (\#epochs \;//\; \#devices) \;\%\; \#devices)]$$

Where *#epochs* is the total number of epochs in the training. The conscious choice was made not to shuffle the list to allow for different orders of devices for the training. Lastly, the *randomized* mode chooses a random device as the anchor

device and also uses the specified number of epochs from *multi_anchor_cycle* to create the training blocks.

The aforementioned changes are operating on an epoch level, meaning each training block still uses only a single anchor device. However, these training blocks have differing anchor devices in comparison to the training of the original ColloSSL framework, which only uses a single anchor device for the whole training. The intuition behind this implementation was to enhance features that are found in the training data of multiple devices compared to features only found in the data of a single device. After running our experiments, we moved the implementation of anchor switching from outside the training function into the training loop to prevent unnecessary re-instantiations of the optimizer, which could hinder convergence of the feature extractor (see Section 5.2.1 for more details). Yet, the device selection logic remains unchanged.

To further explore this training style, similar changes were performed on the batch level within an epoch. This training mode can be enabled by using the flag *multi_anchor_batches*. The mode switches the anchor device *within* a training epoch rather than *between* epochs. To achieve this, an addition was made to the training call to accommodate the use of multiple anchor devices within an epoch. Comparatively, a list of possible anchor devices is passed to the function, and the three *multi-anchor training modes* are implemented, providing similar functionalities. For the epoch mode of multi-anchor training, we opted to allow for arbitrary lists of lists (groups) as the anchor devices. For example, the *multi-anchor list*:

$$MAL = [[hand, chest, ankle], [hand]]$$

Together with the *divided* training mode and *multi-anchor batch* training enabled, would result in a training that uses *hand*, *chest*, and *ankle* as the anchor devices for the first half of the training and then trains on *hand* as the sole anchor for the second half of the training. When using *cycle* training mode instead of *divided*, the training would switch between the two groups of anchors every (few) epoch(s) (dependent on the value of *multi-anchor cycle*). This change provides a vast amount of customization during training. After customizing the training of the feature extractor, we developed a fine-tuning approach with the goal of fine-tuning for multiple devices in a single model.

3.2 Multi-Fine-tuning

In order to assess a trained model regarding its performance on a variety of sensor devices, a new evaluation method was implemented. The original evaluation method *base_model* provided in the ColloSSL GitHub repository passes a trained

feature extractor to the evaluation function. There, a classification head (consisting of a fully connected layer and an output layer) is attached and this classifier as well as the last convolution layer of the feature extractor are fine-tuned to a specified *fine-tuning device* (*ft_device*) using supervised training and afterwards evaluated on a given *evaluation device* (*e_device*).

Compared to *base_model* evaluation (single fine-tuning), our new method *multi-fine-tuning* approach (*multi_ft*) uses data not only from a single device during fine-tuning but from all devices specified in a *multi_finetune_list*. To achieve this, a dataset is created using the concatenation of labeled data from all devices within the list. This dataset is then used to fine-tune the model (last layer of the feature extractor + classification head). Afterwards, the model can either be evaluated on a single device or on multiple devices using the *multi_eval_list* parameter. Each evaluation is performed using a test set of the associated device. This method is used to assess the "one model fits all" approach discussed in the introduction (Chapter 1).

3.3 Adaptation Layer

After focusing on the implementations regarding training for the feature extractor and classification head (using fine-tuning), we now explore a separate approach that includes a combination of training adjustments and architecture changes. The idea of an *adaptation layer* came from the intuition that data from different devices can have distinct features that are only found in the data of a specific device. Therefore, training the model to learn more general features to improve performance overall (i.e., on all devices) could hinder the performance on specific devices, which may have very distinct data distributions that are not well represented by any other device. This led to the idea of implementing a layer within the model that is specifically trained on certain devices. Our first implementation of this feature replaced a single convolution layer within the feature extractor with multiple convolution layers (one for each device in the dataset). These convolution layers are not included sequentially but rather used in *parallel*. An illustration of these adaptation layers is depicted in Figure 3.2.

During training, all device specific convolution layers were frozen and unfrozen dynamically for each batch based on the given anchor device. This implementation resulted in two major problems. Firstly, the averaging of the outputs leads to training behavior that depends on the weights of the other convolution layers since the gradient is computed using the combined output. Secondly, the dynamic freezing of some layers forced us to re-instantiate the optimizer after each batch due to a change in the number of trainable variables. This caused the loss

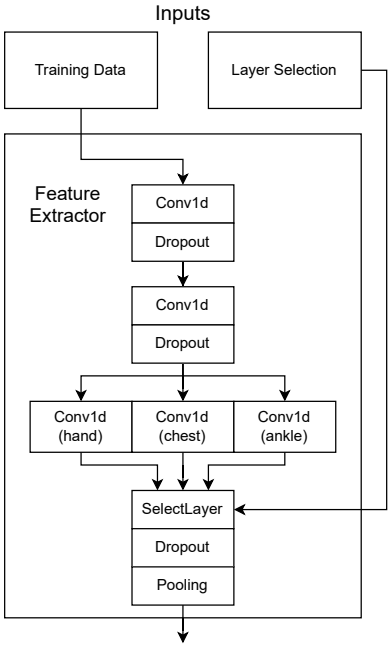


Figure 3.2: Illustration of the architecture changes made to the feature extractor. The adaptation layers replace the last convolution layer of the original ColloSSL model. This example was made for the PamMap2 [33] dataset, which contains three sensor devices: hand, chest, and ankle.

to diverge during training, which is probably caused by constant resetting of adjustable training variables like the momentum within the ADAM [23] optimizer. Therefore, we opted for a different approach to implement the parallel training of multiple convolution layers.

The next approach we chose is a custom Keras [8] layer class *SelectLayer*. The layer receives two inputs. The first input is a list of each device-specific convolution layer. An additional input consisting of a scalar or a list of scalars is used by the *SelectLayer* to choose a single output from the list of outputs based on the scalar it receives. This output selection can be done per batch or per sample. The per-sample selection requires a list of scalars with a length of *batch_size*. The code for this class can be seen in Source Code 3.2. The layer returns the output determined

by the *selection_index*. No trainable parameters are added to the model by the introduction of *SelectLayer*, as it merely serves as an output selection for the parallel convolution layers. The number of parameters that are trained simultaneously during training also remains unchanged compared to the original architecture, since all gradients for device specific convolution layers that are not selected are zero. This behavior is ensured by only using differentiable operations (*tf.gather_nd* and simple indexing) within the *call* function of the *SelectLayer*.

After a few first tests, we revised the design of the *Adaptation Layer* implementations to allow for more freedom in training and improve readability and functionality of the code. The second design follows the same idea of the first approach, but implements the convolution layers as sub-layers of its own instance rather than creating the layers manually during creation of the feature extractor. The full code of this custom Keras layer called *AdaptationLayer* can be seen in Appendix A, Source Code A.1. Each instance of *AdaptationLayer* receives a number of sub layers, the type of sub layer, the sub layer arguments, and a boolean as input. The sub layers are then created as a list with each sub layer having the specified type and arguments. The boolean is used for the new feature of *adapt mode*. This mode allows the layer to switch between regular (general) and adaptation (sub-layer specific) training. Regular training forwards the mean output of all sub layers and passes the gradient equally to all sub layers during backpropagation, therefore training all sub layers simultaneously when *adapt mode* is *disabled*. During adaptation training, the layer forwards only the output of the sub-layer specified by the *selection index* and only passes the gradient to the same layer during backpropagation, while passing zeros to all other sub-layers. Hence, only training one specified sub-layer when *adapt mode* is *enabled*.

```
1  import tensorflow as tf
2
3  @tf.keras.utils.register_keras_serializable()
4  class SelectLayer(tf.keras.layers.Layer):
5      def __init__(self, **kwargs):
6          super(SelectLayer, self).__init__(**kwargs)
7
8      def call(self, inputs):
9          conv_outputs_list, selection_index = inputs
10         # conv_outputs_list: list of tensors [batch, time,
           ↪  channels]
11         # selection_index: scalar tensor or shape [batch]
           ↪  int32/64
12
```

```
        # If selection_index is a scalar:
        if selection_index.shape.rank == 0:
            selected_output = stacked[selection_index]
        else:
            # for per-sample selection use gather.
            # Assumes selection_index shape [batch]
            batch_indices =
            ↪  tf.range(tf.shape(selection_index)[0])
            indices = tf.stack([selection_index, batch_indices],
            ↪  axis=1)  # [batch, 2]
            selected_output = tf.gather_nd(stacked, indices)

        return selected_output

    def get_config(self):
        config = super(SelectLayer, self).get_config()
        return config
```

Source Code 3.2: Source code of the custom Keras layer class *SelectLayer*.

When enabling adaptation layers during training using the *adaptation_layer* flag, the desired architecture can be chosen using *adaptation_layer_architecture*. We allow full freedom to replace any convolution layer with an *AdaptationLayer* by specifying an architecture string. The string is specified as

$$s = ''[c|a] - [c|a] - [c|a]''$$

where c stands for the use of the *Conv1d* layer and a represents the substitution with an *AdaptationLayer*.

After implementing the *AdaptationLayer* class, we also revised the creation of the sub-layers to allow more flexibility. The number of sub-layers and the devices which contribute to its training can be adjusted by using a new parser argument: *adaptation_groups*. This argument receives a list of lists, where each element of the list represents a group of devices that contribute to training the associated sub-layer. For example, using

$$adaptation\_groups = [['hand'], ['chest], ['ankle']]$$

results in a sub-layer structure similar to the one shown in Fig.3.2. When passing $[['hand','ankle'], ['chest']]$ as the argument, only two sub layers are created within

an *AdaptationLayer* and both *hand* and *ankle* data is used to train their associated sub-layer (as long as *adapt mode* is *enabled*). During training, the index of the *first* group containing the current anchor device is computed and used as the selection input for the adaptation layer(s). This ensures that the behavior of the model remains consistent even if a device is part of multiple groups. If a device is used as a training device and not part of any adaptation group, the function throws an error.

During multi-fine-tuning evaluation, the selections of the adaptation layers are synchronized with the batches of labeled data for each device to achieve meaningful fine-tuning behavior. Source Code 3.3 contains the code for creating the multi-fine-tuning dataset and synchronizing the layer selections. The revised implementation also allows us to use Dense (fully-connected) layers as sub-layers of the *AdaptationLayer*, which we used to implement an adaptation version of the classification head to use during multi-fine-tuning. It can be enabled using the flag *adaptation_classifier*. The classification head then substitutes the fully-connected layer as well as the output layer with *AdaptationLayers* of the same dimensions, hence functioning as separate classification heads for each sub-layer.

The introduction of different sub-layers for specific (groups of) devices can result in outputs of these sub-layers that diverge significantly. Especially with the use of multi-fine-tuning, this can cause problems (see Section 5.2.3 for more details). Therefore, we introduce the option to enable a regulatory loss during training of the feature extractor to penalize differences in the sub-layer outputs. Hence, reducing divergence of sub-layer outputs and resulting in increased multi-fine-tuning performance. We opted to use *KL-divergence* due to better results than standard *least squares* for deep learning applications [24]. The regulatory loss is computed for each adaptation layer within the feature extractor. For each adaptation layer, the outputs of the sub-layers are compared to the mean of the outputs. These two distributions are then used to compute the *KL-divergence*. The loss for each adaptation layer is summed up and added to the existing contrastive loss for each batch. This loss is *not* used during fine-tuning, therefore not encouraging convergence of the adaptation classifier, which could be undesirable. The loss can be optionally enabled using the *reg_loss* flag (only possible in conjunction with adaptation layers).

As a last addition to the adaptation layer, we implemented the option to train the adaptation layers using *adapt mode switching*. This feature allows the selective training of models with *more than one* adaptation layer using three different switching modes. We took inspiration from our implementation of the multi-anchor training for the implementation of these modes. In *cycle* mode, the adaptation layer with enabled adapt mode is given by the current epoch modulo the adaptation layer ID. This means that in the first epoch, the first adaptation layer is chosen, then the second, and so forth. This ensures an even distribution of adaptation mode training for

all three adaptation layers with only short training intervals. *Divided* mode divides the training epochs into equal parts and trains one of the adaptation layers using adapt mode in each training section. However, due to patience and the resulting early stopping of the training, this number can be smaller for the third adaptation layer. Though, the number of actual training epochs is not known beforehand and can therefore not be used for the equal segmentation of the training into equally sized blocks. *Random* mode chooses one of the three adaptation layers for adapt mode training randomly every epoch. This adapt mode training can be activated using the *adapt_mode_training* option and pass one of the three modes to it.

```

def get_closest_neighbor(dataset, anchor_id):
    positive_indices, negative_indices, distances =
    ↪  get_pos_neg_apriori(dataset, anchor_id)
    return positive_indices[0]

def copy_weights(source, target):
    target.set_weights(copy.deepcopy(source.get_weights()))

def handle_generalizability(model, dataset, anchor_id):
    neighbor_id = get_closest_neighbor(dataset_full, ft_id)
    for layer in model.layers:
        if isInstance(layer, AdaptationLayer):
            copy_weights(layer.sub_layers[neighbor_id],
            ↪  layer.sub_layers[anchor_id])

# training_set_X, training_set_Y are combined sets of data/labels
↪  for all fine-tuning devices

adapt_sel = []
for ft_id, ft_device in enum(fine_tune_devices):

    if args.generalizability and ft_id == eval_id:
        # ft_device was not present in training
        handle_generalizability(eval_model, dataset_full, ft_id)

    num_batches = training_set_X[ft_id].shape[0]
    device_selections = np.full((num_batches,), ft_id)
    adapt_sel.append(device_selections)

```

```
28 # combine into a single list
29 adapt_sel = np.concatenate([du for du in adapt_sel])
30
31 # shuffle and data and keep adapt_sel synchronized
32 data_size = training_set_Y.shape[0]
33 shuffle_indices = get_random_shuffle_indices(data_saize)
34 training_set_X = training_set_X[shuffle_indices]
35 training_set_Y = training_set_Y[shuffle_indices]
36 adapt_sel = adapt_sel[shuffle_indices]
```

Source Code 3.3: Source code of creation of the multi-fine-tuning dataset and the synchronization of the layer selections.

3.4 Generalizability Option

The final addition to the ColloSSL framework was the option to train and evaluate the model regarding its generalizability. The argument parser was extended with the flag *generalizability*. Generalizability training automatically removes the *evaluation device* from the list of training devices, hence it ensures that no data from the *evaluation device* is used during training. This option also ensures that the training device and evaluation device can not be the same.

When generalizability is enabled while using multi-anchor training, the chosen evaluation device is removed from each sub-list of the *multi-anchor list*. Sub-lists that are empty because of this removal are taken out of the *multi-anchor list* entirely. By design, the *multi-anchor list* is a list of lists, even for single elements in each sub-list. This is done to ensure correct behavior in all cases. The generalizability option does not affect the multi-fine-tuning, as our main focus during research is (*regular*) generalizability rather than *true* generalizability (refer to Section 1.2 for full details on the distinction). For single fine-tuning, we set the fine-tuning device to the evaluation device to evaluate the interaction of data missing from feature extractor training and fine-tuning for the unknown device.

For the discussion of generalizability with the use of adaptation layers, we have to distinguish between two different cases. The first case has a sub-layer dedicated to only the evaluation device. In this case, the sub-layer does not receive any training as the evaluation device is not contained in the training data and therefore is never chosen as the anchor device. After the training of the feature extractor, we choose a *replacement device* by computing the pairwise MMD for all devices and

choosing the device with the least MMD. The index of the *replacement device's* sub-layer is computed, and the weights of the corresponding sub-layer are then copied to the sub-layer of the evaluation device before fine-tuning. Source Code 3.3 shows this replacement process within the *handle_generalizability(...)* function. The second case uses a group of devices for the sub-layer of the evaluation device. In this case, no replacement of weights is necessary as the associated sub-layer is trained on the remaining elements of this group.

After examining the additions and changes to the original ColloSSL source code, we establish the baselines against which we compare our experiments regarding the effects on generalizability of these implementations.

Chapter 4

Baselines

In order to evaluate new ideas and approaches that are implemented in the scope of this thesis, we first need to establish baselines that we compare our work against. This is important to evaluate results and draw conclusions for any experiments that are performed during our research. m

4.1 Methodology

The framework we chose to work with is the ColloSSL framework, due to its state-of-the-art performance using self-supervised learning for inertial sensor data in a single evaluation device setup [17]. The code provided in their GitHub repository [18] serves as the foundation of our own implementations. The provided model architecture is evaluated using ColloSSL's collaborative training method. Additionally, we chose to train and evaluate the model using fully supervised training as it serves a measurement for how much labeled data effects the cross device performance.

To measure the effects of future adjustments in different scenarios and prevent bias towards certain results caused by data anomalies, we ran all baseline experiments on two different datasets. Firstly, the *RealWorld* dataset consisting of seven device positions (head, chest, waist, forearm, upper arm, shin and thigh) and eight activities (walking, running, sitting, standing, lying, stairs ascending, stairs descending and jumping). The data was collected from 15 subjects, eight males and seven females [38]. Secondly, a subset of the *PaMap2* dateset. The dataset contains three different sensor positions (wrist, in the following also called: hand, chest and ankle). Data was collected from nine subjects on 18 different activities [33]. The following activities were included in our experiments: lying, sitting, standing, walking, running, cycling, nordic walking, ascending stairs, descending

stairs, vacuum cleaning, ironing and rope jumping. This is in accordance with the version of the dataset that is provided in the GitHub repository of ColloSSL. However, while trying to reproduce the results of the paper on the PaMap2 dataset, we encountered considerably differing results for the chest sensor. Therefore, we created our own version of the PaMap2 dataset (see Appendix A *create_pamap2.py* for full details) in accordance to the version provided by the ColloSSL repository. This includes some preprocessing which mainly focuses on interpolating values if missing and reduce the actions that are considered for classification to the previously mentioned set of activities. The results on the chest position using our own version of PaMap2 were still differing from the ones achieved by *Jain and Tang et al.*[17] but slightly improved compared to the provided version. Hence, we used our own version for all experiments.

4.2 Limitations and Reproducibility

Following up on the previous paragraph on the difficulties we encountered using the PaMap2 dataset provided in the ColloSSL GitHub [18], we conducted our own experiments to establish a baseline rather than using the results provided in the ColloSSL paper [17]. Because of the large amount of experiments conducted during our research, we ran each experiment only once. However, to reduce chance bias we mostly focus on averages over all devices in the form of ADI and GI rather than single evaluation results. This should mostly reduce this bias and make the results comparable to our own baseline as well as the results published by *Jain and Tang et al.* [17].

To ensure reproducibility, we opted to use TensorFlow version 2.16.1 [1] as it implements an option for deterministic operations on GPUs using:

tf.config.experimental.enable_op_determinism() [40]

Though, because all of our experiments are performed on the OMNI Cluster [32] of the University of Siegen and therefore mostly use different (GPU) nodes, we did not manage to achieve fully deterministic behavior for the majority of the experiments. To reproduce experiments reliably we opted to re-run the best performing model on a separate machine with a single GPU which allowed us to get a fully reproducible result. Before we can set up a baseline we first have to discuss the metrics that are used to compare different experiments.

4.3 Evaluation Metrics

As we are interested in the performance of a model on multiple sensor positions, we focus on averages over multiple devices rather than comparing single values. The scores we chose for this purpose are the average F1-score of a model evaluated on **all** devices in a given dataset, the standard deviation (σ) over the F1-scores and the minimum F1-score. To evaluate the adaptability of a model across multiple inertial sensor positions, we define the *Adaptability Index (ADI)* as a affine combination of three components:

Let M a deep learning model and$D = \{d_1, ..., d_P\}$ a dataset containing P different devices. We define

$$F_1(M_D(d_i)),\ 1 \leq i \leq P$$

as the *F1-score* of model M trained on dataset D and evaluated on device d_i. This leads to the following definitions:

1. **Overall accuracy** — the mean F1 score across devices:

$$\mu_{ad}(M, D) := \frac{1}{P} \sum_{p=1}^{P} F_1(M_D(d_p)) \tag{4.1}$$

2. **Robustness to variability** — the standard deviation of F1 scores:

$$\sigma_{ad}(M, D) := \sqrt{\frac{1}{P} \sum_{p=1}^{P} \big(F_1(M_D(d_p)) - \mu_{ad}(M, D)\big)^2} \tag{4.2}$$

3. **Worst-case resilience** — the minimum F1 score across devices:

$$F_{1_{ad}}^{min}(M, D) := \min_{p \in \{1,...,n\}} F_1(M_D(d_p)) \tag{4.3}$$

The Adaptability Index is then defined as:

$$\boxed{ADI(M, D) := w_1 \mu_{ad}(M, D)\ -\ w_2 \sigma_{ad}(M, D)\ +\ w_3 F_{1_{ad}}^{min}(M, D)} \tag{4.4}$$

where $w_1, w_2, w_3 \geq 0$ and $w_1 + w_2 + w_3 = 1$. This formulation rewards models that achieve high average performance (μ), maintain consistency across sensor positions (low σ), and guarantee strong performance in the worst-case scenario ($F_1^{\min}$). In all following chapters we use values $0.5, 0.2$ and 0.3 for w_1, w_2 and w_3 respectively.

Similarly, we introduce the *Generalizability Index (GI)* using the following definitions:

$$\mu_{gen}(M, D) := \frac{1}{P} \sum_{p=1}^{P} F_1(M_{D \setminus \{d_p\}}(d_p)) \tag{4.5}$$

$$\sigma_{gen}(M, D) := \sqrt{\frac{1}{P} \sum_{p=1}^{P} \left(F_1(M_{D \setminus \{d_p\}}(d_p)) - \mu_{gen}(M, D)\right)^2} \tag{4.6}$$

$$F_{1_{gen}}^{min}(M, D) := \min_{p \in \{1,\ldots,n\}} F_1(M_{D \setminus \{d_p\}}(d_p)) \tag{4.7}$$

$$\boxed{\text{GI}(M, D) := w_1 \mu_{gen}(M, D) \; - \; w_2 \sigma_{gen}(M, D) \; + \; w_3 F_{1_{gen}}^{min}(M, D)} \tag{4.8}$$

ADI and GI are defined very similarly. The key difference consists of the models training. Adaptability models are trained on all devices of a given dataset and generalizability models are trained using the *leave-one-out* approach. The missing device is then used during fine-tuning to evaluate how the model generalizes to a previously unseen device.

After establishing our evaluation metrics, we now show the results regarding *adaptability* and *generalizability* of the ColloSSL framework using *self-supervised collaborative* and *fully supervised* training respectively. For every result, the evaluation device is the same as the fine-tuning device (if applicable) unless stated otherwise.

4.4 Evaluation Methods

The F1-scores that are used to compare different models to each other are obtained through the evaluation methods provided in the ColloSSL GitHub. For our baselines, the two used methods are called *base model* evaluation and *full model* evaluation. The *base model* evaluation uses a pre-trained feature extractor and attaches a classification head consisting of a fully connected layer and an output layer. The classification head is initialized with random weights. Then, the last layer of the feature extractor and the classification head are then fine-tuned to a specific *fine-tuning device*. Afterwards, the resulting model is applied to a test set and the results are stated as an F1-score. The *full model* evaluation receives a model already consisting of a feature extractor as well as a classification head both with pre-trained weights. The fine-tuning and evaluation process then remains the

same as for the *base model* evaluation. However, since the classification head is not trained for self-supervised collaborative training, the *full model* evaluation can only be applied to model trained using fully-supervised training. Hence, for all baseline experiments, we use *full model* evaluation on models trained using supervised training and *base model* evaluation for model trained using self-supervised collaborative training.

4.5 Baseline - Adaptability

The adaptability experiments are split in single device runs and multi device runs for both training techniques (self-supervised collaborative training and fully supervised training). For the single device experiments, we trained a model for each of the anchor devices in both datasets (RealWorld - 7 devices, PaMap2 - 3 devices). The results can be found in the according Subsections 4.5.1 and 4.5.2.

4.5.1 Self-supervised Collaborative Training

For all experiments in this thesis, we used 100 training and fine-tuning epochs to preserve comparability to the results published by *Jain and Tang et al.* [17]. A full list of all parameters can be found in Appendix A. We did not conduct any experiments with differing values for any additional parameters due to the already high number of experiments and time constraints of this thesis. We assumed these values to perform best for the base ColloSSL model because of their use in the mentioned script that is provided to reproduce the results from the ColloSSL paper [17].

For the collaborative training we trained a feature extractor using each of the devices from the respective datasets as the single anchor device. Those feature extractors were then fine-tuned and evaluated on each sensor position, totaling $2 * ((7 * 7) + (3 * 3)) = 116$ F1-scores. The results are shown in Table 4.1 for RealWorld and Table 4.2 for PaMap2. These scores were then averaged for each anchor device, giving a measurement of the performance of a single model on all devices of the according dataset. Additionally, the the standard deviation (σ) for each model is computed. Lastly, the *Adaptability Index* (ADI) is computed for each model using the two previous measures as well as the minimum over the F1 scores.

For the collaborative training of ColloSSL on RealWorld, we found that training on *forearm* as the anchor device, the model achieves the best results in μ_{ad} and ADI with scores of 0.765, 0.570 respectively. However, the best σ_{ad} performance is observed when using *shin* as the anchor device, indicating a more even

performance compared to the *forearm* model. For PaMap2, Training on *hand* as the anchor device resulted in the best *Adaptability Index* with 0.448 despite *chest* achieving a notably lower σ_{ad} (0.116 compared to 0.149). This is mostly caused by significantly higher scores for hand and ankle on the *hand* model. This suggests a higher similarity for features of hand and ankle than for chest and any of the two other devices.

		anchor devices						
	F1-scores	**head**	**chest**	**waist**	**forearm**	**upper arm**	**shin**	**thigh**
evaluation devices	**head**	0.667	0.700	0.692	0.724	0.647	0.742	0.656
	chest	0.681	0.686	0.720	0.700	0.680	0.703	0.710
	waist	0.886	0.889	0.878	0.839	0.886	0.858	0.864
	forearm	0.810	0.771	0.803	0.798	0.748	0.749	0.746
	upper arm	0.786	0.766	0.788	0.755	0.794	0.773	0.803
	shin	0.583	0.660	0.662	0.672	0.675	0.671	0.674
	thigh	0.858	0.783	0.805	0.866	0.693	0.788	0.787
	$\mu_{\mathbf{ad}}$	0.753	0.751	0.764	**0.765**	0.732	0.755	0.749
	$\sigma_{\mathbf{ad}}$	0.111	0.077	0.076	0.072	0.084	**0.061**	0.075
	ADI	0.529	0.558	0.565	**0.570**	0.543	0.567	0.556

Table 4.1: Run of the ColloSSL [17] model using self-supervised collaborative training on the *RealWorld* [38] dataset. Each combination of *anchor device* and *evalutation device* was evaluated. The best results are **bold**.

		anchor devices		
	F1-scores	**hand**	**chest**	**ankle**
eval devices	**hand**	0.787	0.731	0.605
	chest	0.495	0.500	0.490
	ankle	0.694	0.640	0.723
	$\mu_{\mathbf{ad}}$	**0.659**	0.624	0.606
	$\sigma_{\mathbf{ad}}$	0.149	**0.116**	0.117
	ADI	**0.448**	0.439	0.427

Table 4.2: Run of the ColloSSL [17] model using self-supervised collaborative training on the *PaMap2* [33] dataset. Each combination of *anchor device* and *evalutation device* was evaluated. The best results for each experiment are **bold**.

The same experiments were repeated with the built-in *multi-anchor* option enabled. This overrides the training device and uses all available devices for training, by switching the anchor device randomly each batch. Thus, resulting in two different feature extractors which were fine-tuned and evaluated on all devices of

their associated datasets. Table 4.3 shows the results for *multi-anchor* collaborative training. With *ADIs* of 0.540 and 0.419 for RealWorld and PaMap2 respectively, compared to their best performing models with scores of 0.570 and 0.448, we see no improvement over using a single anchor device regarding their adaptability. However, due to the small margins and only a single run of both experiments, we can not draw any substantial conclusions regarding their performances.

	F1-scores	**RealWorld**		**PaMap2**
evaluation devices	**head**	0.720	**hand**	0.714
	chest	0.677	**chest**	0.469
	waist	0.882	**ankle**	0.635
	forearm	0.794		
	upper arm	0.791		
	shin	0.606		
	thigh	0.811		
	$\mu_{\mathbf{ad}}$	0.754		0.606
	$\sigma_{\mathbf{ad}}$	0.093		0.125
	ADI	0.540		0.419

Table 4.3: Run of the ColloSSL [17] model using self-supervised collaborative **multi-anchor** training on the RealWorld and *PaMap2* datasets. Each model is trained on all devices of the associated dataset.

4.5.2 Fully-supervised Training

After stating the results of our collaborative training runs, we now investigate the baseline for fully-supervised training. In comparison to the collaborative training where feature extractor and classification head are trained respective fine-tuned *separately*, the fully-supervised approach trains a full classification model already consisting of feature extractor *and* classification head. Though, the training differs, the architecture of the model remains unchanged. Regarding the adaptability experiments, we repeated the same experiments from the previous section with supervised training. The resulting F1-scores and computed measures are shown in Tables 4.4 and 4.5 for RealWorld and PaMap2 respectively. For RealWorld we see that *chest* as the training device results in the best *Adaptability Index* with 0.187. For PaMap2 with a score of 0.110, the best performing model is the one trained on hand data. We also notice that despite *hand* outperforming *chest* in the average F1-scores with 0.332 compared to 0.279, *chest* achieves a considerably lower σ_{ad} (0.216 compared to *hand* 0.329) which results in ADIs which only differ by

0.005. This result underlines the significance of including the standard deviation and minimum F1 into the *Adaptability Index* to benefit models with lower averages but more equal performance and higher worst case results when evaluating for multiple devices.

		anchor devices						
	F1	**head**	**chest**	**waist**	**forearm**	**upper arm**	**shin**	**thigh**
evaluation devices	**head**	0.678	0.388	0.389	0.104	0.146	0.122	0.167
	chest	0.373	0.685	0.193	0.055	0.502	0.239	0.398
	waist	0.057	0.109	0.855	0.144	0.117	0.036	0.260
	forearm	0.154	0.268	0.153	0.788	0.191	0.034	0.028
	upper arm	0.428	0.357	0.052	0.056	0.776	0.203	0.409
	shin	0.248	0.309	0.125	0.037	0.408	0.663	0.323
	thigh	0.501	0.599	0.089	0.045	0.582	0.342	0.848
	$\mu_{\mathbf{ad}}$	0.348	0.388	0.265	0.176	**0.389**	0.234	0.348
	$\sigma_{\mathbf{ad}}$	0.213	**0.197**	0.282	0.273	0.249	0.219	0.258
	ADI	0.149	**0.187**	0.092	0.044	0.180	0.083	0.131

Table 4.4: Run of the ColloSSL [17] model using fully-supervised training on the *RealWorld* [38] dataset. Each combination of *anchor device* and *evalutation device* was evaluated. The best results for each experiment are **bold**.

		anchor devices		
	F1-scores	**hand**	**chest**	**ankle**
eval devices	**hand**	0.686	0.406	0.007
	chest	0.274	0.401	0.034
	ankle	0.035	0.029	0.644
	$\mu_{\mathbf{ad}}$	**0.332**	0.279	0.228
	$\sigma_{\mathbf{ad}}$	0.329	**0.216**	0.360
	ADI	**0.110**	0.105	0.044

Table 4.5: Run of the ColloSSL [17] model using fully-supervised training on the *PaMap2* [33] dataset. Each combination of *anchor device* and *evalutation device* was evaluated. The best results for each experiment are **bold**.

For the multi device training, the data for all devices from the associated dataset is concatenated and used for the supervised training. The resulting F1-scores are shown in Table4.6. This setup performs significantly better than the single device training. This is to be expected since single device training on a full classification model means evaluation devices that are not the training device are never seen

before evaluation, hence the results are obtained for *true adaptability* compared to collaborative training, where data from evaluation devices is seen during fine-tuning. The multi device training results in an ADI of 0.400 for RealWorld and 0.405 for PaMap2, thereby increasing by 113.9% and 268.1% compared to single device training, resulting from the use of all devices data within the training.

	F1-scores	RealWorld		PaMap2
evaluation devices	**head**	0.478	**hand**	0.760
	chest	0.670	**chest**	0.428
	waist	0.645	**ankle**	0.676
	forearm	0.445		
	upper arm	0.538		
	shin	0.562		
	thigh	0.645		
	$\mu_{\mathbf{ad}}$	0.569		0.621
	$\sigma_{\mathbf{ad}}$	0.088		0.173
	ADI	0.400		0.405

Table 4.6: Run of the ColloSSL [17] model using fully-supervised **multi device** training on the *RealWorld* and *PaMap2* datasets. Each model is trained on all devices of the associated dataset.

4.6 Baseline - Generalizability

This section shows the results from the initial runs of the ColloSSL framework using self-supervised collaborative training and fully-supervised training regarding *generalizability*. For each of the two training methods, we chose only to evaluate multi device training approaches due to the fact that for every run, data for the *evaluation device* was removed from the training. For single device trainings, this would not cause significant changes for the collaborative case and no changes at all for the supervised case. Therefore, we opted only to investigate multi device trainings when analyzing the *generalizability* of a model.

4.6.1 Self-supervised Collaborative Training

For the self-supervised collaborative training, the built-in *multi-anchor* training mode of the ColloSSL framework was used as the baseline. This randomly chooses a anchor device from all available devices every batch. For each device d_p of a dataset $D = \{d_1, ..., d_n\}$, a model was trained that uses the data from devices

$d_i \in D, i \neq p$ for the training of the feature extractor. A classification head is then attached to the feature extractor and fine-tuned to d_p. The resulting model is then only evaluated on d_p since this is the only device not present in the training of the feature extractor. Table 4.7 shows the results for this training on both datasets. This experiment served as an extension of the experiments already conducted by *Jain and Tang et al.* [17] regarding the *generalizability* of their framework. Even though we were only able to conduct each run only a single time, we can with reasonable certainty confirm their findings about the *generalizability*. Our results are within the margins described in their paper for all of the devices of both datasets [17]. In our single test, this training method even outperforms the training using *all* devices on RealWorld in both average F1-score of 0.763 compared to 0.754 and σ_{ad} with 0.083 to 0.093, resulting in a *Generalizability Index* of 0.560. One explanation could be the decrease in different data distributions. For PaMap2 the results are almost identical with $ADIs$ of 0.397 compared to 0.400, which could support this theory since PaMap2 has less devices and therefore fewer different data distributions to learn distinct features from in the first place. However, all results may also differ because of statistical variance through in-deterministic training behavior (refer to Section 4.2 for more details).

	F1-scores	**RealWorld**		**PaMap2**
evaluation devices	**head**	0.715	**hand**	0.817
	chest	0.697	**chest**	0.424
	waist	0.881	**ankle**	0.616
	forearm	0.751		
	upper arm	0.796		
	shin	0.652		
	thigh	0.847		
	$\mu_{\mathbf{gen}}$	0.763		0.619
	$\sigma_{\mathbf{gen}}$	0.083		0.197
	GI	0.560		0.397

Table 4.7: Run of the ColloSSL [17] model using self-supervised collaborative **multi-anchor** training on the *RealWorld* and *PaMap2* datasets. Each model is trained on all devices except the evaluation device of the associated dataset. Afterwards, it is fine-tuned and evaluated on the evaluation device.

4.6.2 Fully-supervised Training

The training approach described in the previous section was repeated using supervised training on a full classification model. Training data from all devices except of the *evaluation device* was concatenated and used for training. The resulting model was then evaluated on the *evaluation device*. The results can be found in Table 4.8. We see a significant decrease in all measures compared to training on all devices 4.6. However, this should be expected, since the lack of fine-tuning compared to the collaborative approach means that these values represent *true generalizability*.

	F1-scores	**RealWorld**		**PaMap2**
evaluation devices	**head**	0.275	**hand**	0.322
	chest	0.579	**chest**	0.247
	waist	0.265	**ankle**	0.170
	forearm	0.229		
	upper arm	0.440		
	shin	0.422		
	thigh	0.507		
	$\mu_{\mathbf{gen}}$	0.388		0.246
	$\sigma_{\mathbf{gen}}$	0.134		0.076
	GI	0.236		0.159

Table 4.8: Run of the ColloSSL [17] model using fully-supervised **multi device** training on the *RealWorld* and *PaMap2* datasets. Each model is trained on all devices except the evaluation device of the associated dataset.

Chapter 5

Experiments and Results

In this chapter, we take a look at the different experiments we conducted regarding our implemented additions to the ColloSSL [17] framework and discuss the results. We evaluate a variety of models regarding adaptability and generalizability. Our methodology for conducting the experiments remains unchanged from the baselines chapter (see Section 4.1 for full details). Each experiment was performed by running the code from *collaborative_training.py* (see Appendix A for the full source code) with the associated parser flags for each training mode. As input parameters, the same parameters listed in Section 4.5.1 were used for all experiments to maintain comparability. Any additional parameters used for experiments will be stated explicitly in the corresponding sections. To evaluate the best-performing models accurately, we repeated experiments for those models two times (for a total of three runs) and averaged the results. The results of these repeats will be explicitly stated in the corresponding section 5.2.5.

5.1 Multi-fine-tuning

The first alteration examined is the method used for fine-tuning the classification head of a model. The goal remains to find a model that performs well on all devices of a given dataset. To achieve this, we started by replacing the single fine-tuning method (*base model*) of the ColloSSL framework [17] with a new approach that fine-tunes the model to multiple devices at the same time. We took inspiration from the multi-device training mode for the supervised model (refer to Sections 4.5.2 and 3.2 for more details) and replaced the single device dataset used in the *base model* evaluation method with a multi-device dataset. The multi-device dataset consists of the concatenation of the different sets for each fine-tuning device (specified by the *multi finetune list*). This then serves as the training set for the supervised fine-

tuning. The resulting model "combines" the different models for each device in a dataset into a single model. We first evaluated the baselines defined in Chapter 4 using this method. For single anchor training, this change resulted in an average ADI of 0.532 (across all possible anchor devices) compared to 0.555 using fine-tuning for each evaluation device separately on the RealWorld dataset. On PaMap2 the resulting average ADI is 0.426, compared to the baseline's ADI of 0.438. This is a decrease of 4.2% and 2.7% on RealWorld and PaMap2 respectively. This is a tradeoff between a small performance decrease to gain the advantage of having a single model compared to seven, respectively three different models for all devices of the associated datasets. The table containing all Adaptability Indices is shown in Table 5.1.

When analyzing the individual F1-scores (see Appendix B, Table B.2 for full details) over all 7 different models for RealWorld, we notice that the F1-scores for *forearm* and *head* decrease on each model and the F1-scores for *chest* and *shin* increase on almost all models (regardless of the used anchor device). This observation proposes the differing effectiveness of this fine-tuning for particular devices. We hypothesize that these increases and decreases are caused by the features that are shared across devices. Since the classification head is fed data from all devices during fine-tuning, this fine-tuning method emphasizes the use of shared features more prominently than single fine-tuning. These shared features are probably more beneficial for the classification of *chest* and *shin* than (device) specific features, and less reliable for the classification of activities for *forearm* and *head* data.

		RealWorld			**PaMap2**	
	Adapt.-scores	**Single ft**	**Multi ft**		**Single ft**	**Multi ft**
evaluation devices	**head**	0.529	0.532	**hand**	0.448	0.400
	chest	0.558	0.541	**chest**	0.439	0.432
	waist	0.565	0.534	**ankle**	0.427	0.448
	forearm	0.570	0.532			
	upper arm	0.543	0.515			
	shin	0.567	0.540			
	thigh	0.556	0.531			
	$\mathbf{Avg(ADI)}$	0.555	0.532		0.438	0.426

Table 5.1: Multi-fine-tuning compared to single fine-tuning (baselines 4 using self-supervised collaborative training on the *RealWorld* and *PaMap2*. Table shows $ADIs$ of the corresponding models.

The same process was repeated for the *generalizability* baseline. Multi-fine-tuning achieves GIs of 0.503 and 0.439 compared to single fine-tuning with scores

of 0.560 and 0.397 for RealWorld and PaMap2, respectively. Resulting in a decrease of 10.2% for RealWorld and a 10.6% increase for PaMap2. This could indicate a better performance of this fine-tuning method when fewer devices are present. Regarding the previous observation, similar findings were detected as in the adaptability case. *Forearm* and *head* again achieve significantly lower F1-scores than with single fine-tuning and *chest* and *shin* data is classified more precisely. This supports our hypothesis concerning the practicality of shared features regarding the classification of different devices. Further experiments concerning this behavior will be discussed as part of the following sections.

The results show that *multi-fine-tuning* could lead to a single model that performs on multiple sensors with comparable performance to models that are fine-tuned for each sensor separately in the for both *adaptability* and *generalizability*. In generalizability scenarios, the benefit of *multi-fine-tuning* may depend on the evaluated dataset. We use *multi-fine-tuning* to evaluate further experiments in addition to *single fine-tuning*, as the goal is ultimately to train a single model to perform well on multiple sensors. This should also give us more data points, which can be used to assess the general performance of *multi-fine-tuning*.

5.2 Collaborative Training

This section focuses on the collaborative approach of the ColloSSL framework [17]. The experiments are separated into two parts. Firstly, alterations were made to the training process of the feature extractor and the fine-tuning process. Those include the use of *multi-anchor epoch* and *multi-anchor batch* training (details can be found in Section 3.1) as well as *warm-up training* combining both multi- and single anchor training. Secondly, alterations were made to the architecture of the model. This is mainly focused on the use of the *adaptation layer* (detailed explanation provided in Section 3.3). We now discuss the test setups that were used to assess the impact of our implemented augmentations on the training process of the ColloSSL framework. To start, we focus on the multi-anchor training.

5.2.1 Multi-Anchor Training

Adaptability

The following experiments were conducted regarding multi-anchor training on the ColloSSL framework. Note that the batch training mode with the *randomized* device selection option is the *multi-anchor mode* used by *Jain and Tang et al.* in their analysis of the generalizability of ColloSSL [17] and serves as the baseline for multi-anchor training. The experiments performed in Chapter 4 used this mode

for multi-anchor training. We chose to include the results of these tests in this section as a comparison to our own implementations. To evaluate the effect of our multi-anchor training modes (switching between epochs or batches) and its three device selection options (*cycle*, *divided* and *randomized*) on the *adaptability* of the model, we trained a feature extractor for each combination of multi-anchor modes and device selection options. This resulted in $2 \cdot 3 = 6$ different feature extractors. We then used those feature extractors and attached a classification head for each device in our two datasets (RealWorld with seven devices [38], and PaMap2 with three devices [33], for a total of $6 \cdot (7 + 3) = 60$ different fine-tuned models. An illustration of these training setups is shown in Figure 5.1.

For the *cycle* device selection option, we set the number of epochs/batches between choosing the next device to 1. For reasons of time constraints, we were not able to experiment with such hyperparameters. The resulting F1 scores for all 60 fine-tuned models as well as their σ_{ad} values and $ADIs$ are depicted in Tables 5.2 and 5.3 for the RealWorld and PaMap2 datasets, respectively. The results show poor performances for *multi-anchor epoch* training using *cycle* and *randomized* device selection on RealWorld. *Divided* whilst performing considerably better, still results in a decreased ADI of roughly 8.1% compared to the best performing single anchor model which uses *forearm* as the anchor device (0.524 vs. 0.570).

These decreases in ADI for *cycle* and *randomized* multi-anchor *epoch* training could mean that an entire epoch per anchor device is too long compared to per batch switching and too short compared to *divided's* longer training blocks, for the feature extractor to converge the weights in a meaningful manner. Each anchor device results in distinct gradients that influence the learned weights in a specific direction. When training the model for one epoch on the same anchor device, the changes in the weights could be rather large compared to training on a single (or multiple) batches for each anchor device. Thereby, impeding the convergence of the weights. In comparison, the *divided* epoch training might give the model enough time in between the switches to converge its weights towards the current anchor devices' data distribution.

Another explanation could be that the decreased performance might be caused by a frequent re-instantiation of the optimizer. This can be attributed to the implementation of the multi-anchor epoch training, which starts a new *training call* after each epoch, therefore basically restarting the training after each epoch with a different anchor device. Unfortunately, we were only able to conduct a single test regarding the second explanation, as the hypothesis was only developed late into the experiments. We adjusted the multi-anchor epoch training to switch the anchor device within the training loop instead of restarting the entire training loop with a different anchor device and repeated the adaptability training on RealWorld using the *cycle* device selection mode. This resulted in a significantly increased perfor-

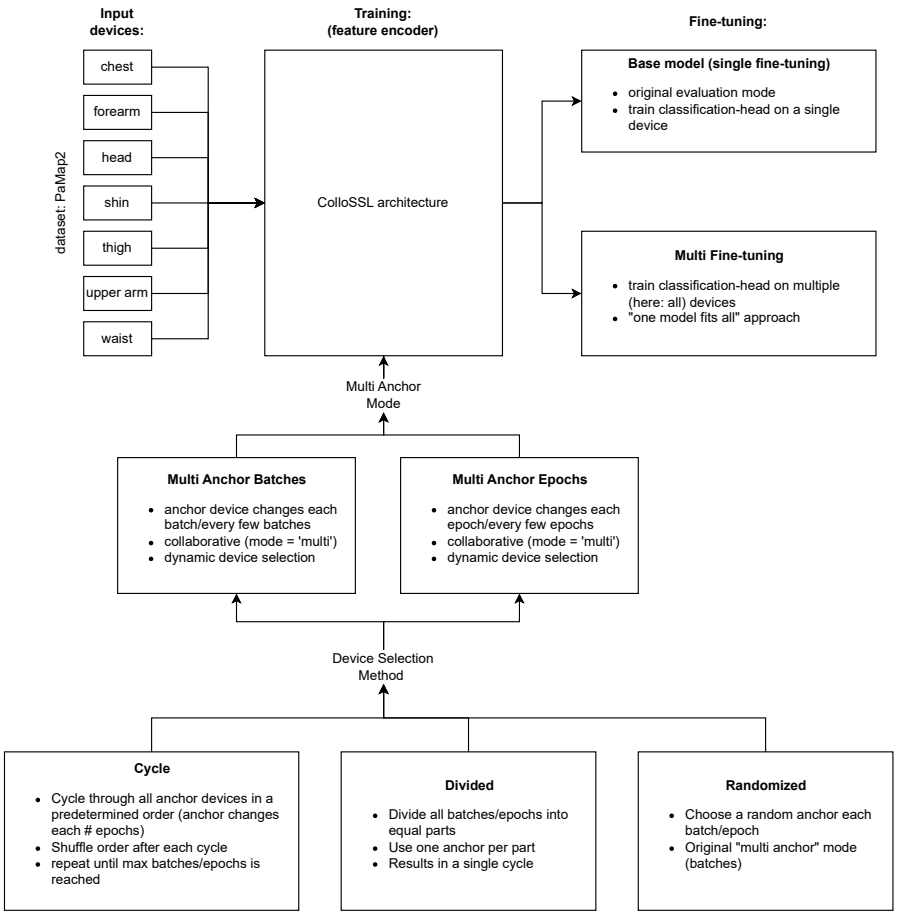


Figure 5.1: Illustration of different experiment setups to evaluate our implemented multi-anchor training modes and device selection methods.

mance compared to the previous implementation for both fine-tuning methods with $ADIs$ of 0.549 and 0.497 compared to 0.444 and 0.422. Full results of this test are depicted in Table B.1 in Appendix B. However, since we were not able to repeat all experiments using this change in implementation, we still discuss the results of the first implementation in the following paragraphs.

The *multi-anchor batch* training performs better than its *multi-anchor epoch* counterpart for all examples, with one exception being the *divided* mode on the PaMap2 dataset. With all three device selection methods outperforming the epoch-

dependent training on RealWorld. *Cycle* and *randomized* also achieve a higher ADI for batch training than for epoch dependent training on PaMap2. However, only *cycle* multi-anchor batch training results in a model with almost the same ADI as the best performing baseline with an ADI of 0.565. We repeated the experiments using *multi-fine-tuning* and found that the average F1-score (i.e., across all 6 training modes) is lowered by 0.036 or 5.2% on RealWorld and increased by 0.022 or 4.5% on PaMap2 using multi-fine-tuning compared to single fine-tuning. The average ADI is lowered by 0.047 or 9.3% on RealWorld and increased by 0.018 or 5.0% on PaMap2 with the largest improvements noted for the μ_{ad} values of *cycle* and *divided* multi epoch training. This could strengthen our previous hypothesis that multi-fine-tuning achieves better results on datasets with fewer devices since PaMap2 only contains three sensor positions compared to seven devices in RealWorld. Additionally, the larger amount of labeled data might benefit the classifier to make more accurate predictions even with less optimized features in the case of *cycle* and *randomized* multi-anchor epoch training. Hence, explaining the improvement compared to single fine-tuning. In summary, the average F1-score over both datasets decreases by 0.007 or 0.4%, and the average ADI decreases by 0.015 or 2.1% when using multi-fine-tuning instead of single fine-tuning. When considering the decreases in ADI, we have to take into consideration that these losses come with the advantage of using a single model compared to multiple models for single fine-tuning. And for smaller datasets, multi-fine-tuning might even increase F1-scores and $ADIs$. Further experiments regarding the number of devices are discussed in section 5.2.3.

		multi-anchor epochs			multi-anchor batches		
	F1-scores	**cycle**	**divided**	**randomized**	**cycle**	**divided**	**randomized**
evaluation devices	**head**	0.531	0.650	0.543	0.740	0.674	0.720
	chest	0.560	0.622	0.526	0.680	0.672	0.677
	waist	0.669	0.825	0.575	0.870	0.860	0.882
	forearm	0.545	0.660	0.566	0.798	0.777	0.794
	upper arm	0.590	0.734	0.585	0.795	0.784	0.791
	shin	0.581	0.656	0.549	0.657	0.651	0.606
	thigh	0.652	0.814	0.692	0.837	0.767	0.811
	$\mu_{\mathbf{ad}}$	0.590	0.709	0.576	**0.768**	0.741	0.754
	$\sigma_{\mathbf{ad}}$	**0.052**	0.083	0.055	0.079	0.077	0.093
	ADI	0.444	0.524	0.435	**0.565**	0.550	0.540

Table 5.2: F1-scores for each fine-tuned model for both multi-anchor training modes (epoch, batch) and three device selection options (cycle. divided, randomized) evaluated using *single* fine-tuning on the *RealWorld* dataset. The best results are **bold**.

		multi-anchor epochs			multi-anchor batches		
	F1-scores	**cycle**	**divided**	**randomized**	**cycle**	**divided**	**randomized**
eval devices	**hand**	0.478	0.600	0.554	0.712	0.604	0.714
	chest	0.432	0.516	0.397	0.488	0.458	0.469
	ankle	0.537	0.641	0.475	0.594	0.726	0.635
	$\mu_{\mathbf{ad}}$	0.482	0.586	0.475	0.598	0.596	**0.606**
	$\sigma_{\mathbf{ad}}$	**0.053**	0.064	0.079	0.112	0.134	0.125
	ADI	0.360	**0.435**	0.341	0.423	0.409	0.419

Table 5.3: F1-scores for each fine-tuned model for both multi-anchor training modes (epoch, batch) and three device selection options (cycle. divided, randomized) evaluated using *single* fine-tuning on the *PaMap2* dataset. The best results are **bold**.

Generalizability

To assess the *generalizability* of the multi-anchor methods, a similar approach was used to the evaluation of the *adaptability*. Yet, there is one key difference in the training. For each device $d_p \in D$ of both datasets, a feature extractor was trained for each combination of multi-anchor mode and device selection method. For the training, an adjusted dataset $D' = D \setminus d_p$ was used for each experiment that left out one device from the dataset. This feature extractor was then fine-tuned using both single fine-tuning and multi-fine-tuning. This resulted in $2 \cdot 3 \cdot (7 + 3) = 60$ different feature extractors and a total of $2 \cdot 60 = 120$ fine-tuned models, as each feature extractor was fine-tuned to the evaluation device d_p as well as multi-fine-tuned to all devices in the dataset. Testing generalizability setups with more than one device missing during training is beyond the scope of this thesis and could be part of future work (especially, the use of multi-fine-tuning in such a scenario could be a topic of interest).

Tables 5.4 (RealWorld) and 5.5 (PaMap2) show the results of multi-anchor training with single fine-tuning. The first thing to notice is that the average μ_{gen} and average GI improve for the generalizability experiments compared to their adaptability conterparts on both datasets. This means that for T beeing the set of all 6 different multi-anchor modes:

$$\frac{1}{|T|} \cdot \sum_{t \in T} \mu_{gen}^{t} > \frac{1}{|T|} \cdot \sum_{t \in T} \mu_{ad}^{t}$$

holds for *almost* all test setups (combinations of datasets and fine-tuning methods). Similarly, the same is true for the average GI compared to the average ADI. The only exception is single fine-tuning on RealWorld where both average μ_{gen} and average GI are only marginally lower ($\mu_{gen} = 0.668$ vs. 0.690 and $GI = 0.504$

vs. 0.510). This increase is most significant on PaMap2 for single fine-tuning with an increase in average μ_{gen} of 0.081 (14.5%) over average μ_{ad}. This resulted in an increase of 0.019 (4.8%) in GI compared to ADI despite significantly higher standard deviations (average $\sigma_{gen} = 0.182$ vs. average $\sigma_{ad} = 0.094$). This suggests a benefit in leaving data out from multi-anchor training when training for a specific device. However, we do not have a convincing explanation for this observation, and further experiments are needed to confirm this trend. On both datasets, the most significant increases are observed for the *cycle* and *randomized multi-anchor epoch* training. This could indicate that those methods work better with fewer devices. This is supported by the fact that both $ADIs$ (0.360 and 0.341) for these methods are (relatively) closer to the highest ADI (0.435) on PaMap2 compared to RealWorld (0.444 and 0.435 vs. 0.565). For *generalizability*, this effect is even more noticeable since *cycle* multi-anchor epoch training achieves the highest GI for single fine-tuning with a value of 0.430. To emphasize the benefit of fewer devices during *multi-anchor* training for our implemented training modes and device selection methods (*multi-anchor epochs, cycle, divided*), the results show that all methods outperform the baseline of *randomized multi-anchor batch* training by significant amounts for the *generalizability* models on PaMap2 using single fine-tuning. The increases range from 3.8% for *divided multi-anchor epoch training* to 8.3% for *cycle multi-anchor epoch* training.

When evaluating all models using multi-fine-tuning, the μ_{gen} and GIs are lowered by 0.021 (3.1%) and 0.006 (0.9%) respectively when averaging over both datasets. Showing less decrease in GI on PaMap2 (1.5%) than on RealWorld (4.6%). However, we have to take into consideration that multi-fine-tuning should be expected to perform worse than single fine-tuning in a *generalizability* scenario since the fine-tuning is performed on all devices equally. In comparison, for single fine-tuning, the evaluation device is the only device for which the model is fine-tuned. Hence, in the scenario of not seeing data from the evaluation device during training, this specific fine-tuning would be expected to perform better.

Analyzing the result from both the adaptability and generalizability experiments combined, we see *cycle* multi-anchor batch training performing best out of all six training modes. It achieves the best results in three out of the eight different testing scenarios (adaptability, generalizability on both RealWorld and PaMap2 for single fine-tuning and multi-fine-tuning). Most significantly, it outperforms the multi-anchor baseline (randomized multi-anchor batch training) regarding ADI/GI in most scenarios (6 out of 8) and closely matches it in the other scenarios. It also closely matches the other training methods in the setups where it got outperformed. Subsequently, we use *cycle* multi-anchor batch training for all following experiments that involve multi-anchor training to limit the number of experiments to a reasonable amount.

		multi-anchor epochs			multi-anchor batches		
	F1-scores	**cycle**	**divided**	**randomized**	**cycle**	**divided**	**randomized**
evaluation devices	**head**	0.451	0.698	0.562	0.706	0.714	0.715
	chest	0.538	0.663	0.524	0.695	0.703	0.697
	waist	0.742	0.848	0.826	0.882	0.871	0.881
	forearm	0.629	0.747	0.636	0.772	0.728	0.751
	upper arm	0.614	0.767	0.620	0.790	0.787	0.796
	shin	0.622	0.653	0.588	0.599	0.660	0.652
	thigh	0.676	0.756	0.681	0.866	0.803	0.847
	$\mu_{\mathbf{gen}}$	0.610	0.733	0.634	0.759	0.752	**0.763**
	$\sigma_{\mathbf{gen}}$	0.094	**0.068**	0.099	0.100	0.072	0.083
	GI	0.422	0.549	0.454	0.539	**0.560**	**0.560**

Table 5.4: F1-scores for each fine-tuned model for both multi-anchor training modes (epoch, batch) and three device selection options (cycle. divided, randomized) evaluated using *single* fine-tuning on the *RealWorld* dataset (evaluation device missing from training). The best results are **bold**.

		multi-anchor epochs			multi-anchor batches		
	F1-scores	**cycle**	**divided**	**randomized**	**cycle**	**divided**	**randomized**
eval devices	**hand**	0.782	0.814	0.798	0.806	0.812	0.817
	chest	0.472	0.443	0.440	0.444	0.454	0.424
	ankle	0.662	0.638	0.699	0.696	0.670	0.616
	$\mu_{\mathbf{gen}}$	0.639	0.632	0.646	**0.649**	0.645	0.619
	$\sigma_{\mathbf{gen}}$	**0.156**	0.186	0.185	0.186	0.180	0.197
	GI	**0.430**	0.412	0.418	0.420	0.423	0.397

Table 5.5: F1-scores for each fine-tuned model for both multi-anchor training modes (epoch, batch) and three device selection options (cycle. divided, randomized) evaluated using *single* fine-tuning on the *PaMap2* dataset (evaluation device missing from training). The best results are **bold**.

5.2.2 Warm-up Training

The next idea we wanted to pursue is the concept of *warm-up* training. The core concept behind this approach is to learn more general features at an early stage during training and afterwards to concretize those features by more specific training. The idea is similar to the approach of fine-tuning. For our experiments, we divided the training time into two stages. The first half of the training consists of multi-anchor training on all devices in the dataset. The second half is then used to tailor the feature extractor to a specific device by using single anchor training for this device.

This approach, however, is not limited to all devices or a single device. Our

implementation of multi-anchor training would allow us to use arbitrary groups of devices and customize training times and iterations. For example, one could separate a dataset $D = \{d_1, ..., d_n\}$ into arbitrary overlapping or non-overlapping groups of devices $D'_1, ..., D'_m \subseteq D$ and train the model on $D'_i, 1 \leq i \leq m$ using the three device (here device-group) selection methods *cycle*, *divided* and *randomized*. Conducting experiments for more than the split of all devices and a singular device is beyond the scope of this thesis. The multi-device training of choice for these experiments is *cycle* multi-anchor batch training to limit the number of experiments.

Adaptability

To evaluate warm-up training regarding its *adaptability*, we trained a feature extractor for each device in RealWorld and PaMap2 using the proposed training split. Each model was then fine-tuned to each device in the associated datasets using both single fine-tuning and multi-fine-tuning to assess the $ADIs$. Hence, resulting in $(7 \cdot 7 + 3 \cdot 3) + (7 + 3) = 68$ different models. Tables 5.6 and 5.7 show the results for RealWorld and PaMap2 respectively. Comparatively, this training method works considerably better on *RealWord* than on PaMap2. And single fine-tuning appears to pair better with this training approach. The results using single fine-tuning closely match the single anchor baselines with average values for μ_{ad}, σ_{ad}, and ADI of $0.750, 0.078$ and 0.553 over all trained models on RealWorld (baseline values for the same measures are $0.753, 0.079, 0.555$). When multi-fine-tuning is applied, the warm-up training performs considerably worse than using single anchor training with multi-fine-tuning, for both RealWorld and PaMap2, with decreases in ADI of 3.4% and 8.0% respectively. The decrease on PaMap2 is caused to a large degree by significantly lower F1 scores (baseline: $0.608, 0.617, 0.639$ vs. warm-up: $0.566, 0.522, 0.556$). These findings could suggest that a feature extractor trained using this approach learns features that are easily adjustable to suit the classification of a specific device but not generally applicable for the classification of all devices. Hence, achieving good performance for single fin-tuning and significantly decreased performance for multi fine-tuning.

When comparing warm-up training to 100% *cycle* multi-anchor (batch) training, the training approach achieves close results on RealWorld for ADI using single fine-tuning (ADI of 0.565) on *chest* (0.556), *waist* (0.568), *forearm* (0.567) and *shin* (0.564) while performing slightly worse on the other devices. On PaMap2, the only device that shows close results is *chest* with an ADI of 0.425 compared to pure multi-anchor training ADI of 0.423. This implies a inherent dependence between the generalization of learned features and the used single anchor device in the second training stage. Therefore, some devices lead to more generally applicable features than others.

For multi-fine-tuning, we observe similar results on RealWorld with *chest*, *waist*, *forearm*, and *shin* outperforming 100% multi-anchor training by small margins (reaching from $2-3\%$ on forearm to 5.1% on waist). On PaMap2, the trend of lower results continues with decreases for all three training devices. A possible explanation for the lower performance of this training approach on PaMap2 compared to RealWorld could be the higher discrepancy between data distributions for the sensor locations in PaMap2. If hand, chest, and ankle have pretty distinct features, the warm-up training could encourage learning features that are shared between all devices, and the subsequent single device training would encourage the model to learn device-specific features, which might be harder from an already pre-trained state than from a randomly initialized one. RealWorld, on the other hand, provides more devices to learn features from and therefore might already learn features in warm-up training that are more closely related to the features the model wants to enhance during the subsequent single anchor training, therefore making it easier to adapt. This is supported by the fact that warm-up training shows significant differences between the two groups of devices ($G1 = \{chest, waist, forearm, shin\}, G2 = \{head, upperarm, thigh\}$). Group 1 shows considerably better performances in both single and multi-fine-tuning compared to Group 2. This suggests features learned from these devices are more helpful for classification on *all* devices than features learned from devices in Group 2. This observation will be part of tests in the Adaptation Layer section of this chapter (see Section 5.2.3 for more details).

		anchor devices (last 50% of training)						
	F1-scores	**head**	**chest**	**waist**	**forearm**	**upper arm**	**shin**	**thigh**
evaluation devices	**head**	0.714	0.691	0.695	0.721	0.642	0.713	0.616
	chest	0.708	0.648	0.701	0.744	0.654	0.709	0.641
	waist	0.846	0.862	0.854	0.874	0.880	0.876	0.868
	forearm	0.809	0.779	0.786	0.801	0.763	0.780	0.742
	upper arm	0.786	0.793	0.791	0.778	0.791	0.781	0.786
	shin	0.605	0.676	0.682	0.656	0.677	0.669	0.693
	thigh	0.802	0.852	0.754	0.807	0.693	0.743	0.813
	$\mu_{\mathbf{ad}}$	0.753	0.757	0.752	**0.769**	0.729	0.753	0.737
	$\sigma_{\mathbf{ad}}$	0.082	0.086	**0.063**	0.070	0.087	0.067	0.092
	ADI	0.541	0.556	**0.568**	0.567	0.540	0.564	0.535

Table 5.6: Run of the ColloSSL [17] model using warm-up training on the *RealWorld* [38] dataset. Each combination of *anchor device* for the last 50% of the training and *evaluation device* was evaluated. The best results for each experiment are **bold**.

		anchor devices (last 50%)		
	F1-scores	**hand**	**chest**	**ankle**
eval devices	**hand**	0.719	0.563	0.550
	chest	0.468	0.502	0.492
	ankle	0.431	0.718	0.600
	$\mu_{\mathbf{ad}}$	0.539	**0.594**	0.547
	$\sigma_{\mathbf{ad}}$	0.157	0.111	**0.054**
	ADI	0.368	**0.425**	0.411

Table 5.7: Run of the ColloSSL [17] model using warm-up training on the *PaMap2* [33] dataset. Each combination of *anchor device* for the last 50% of the training and *evaluation device* was evaluated. The best results for each experiment are **bold**.

Generalizability

Our idea behind evaluating the generalizability of warm-up training was to assess the effects of different devices replacing features that would otherwise have been learned during training on the evaluation device. To see these effects, we used the proposed 50/50 training split to train a model for each combination of *different* devices, using one device as the single training device in the second half of the training and the other device as the evaluation device. The data of the evaluation device was omitted from the multi-anchor training in the first half of the training process. The evaluation was performed using both single and multi-fine-tuning, hence resulting in a total of $2 \cdot (7 \cdot 6 + 3 \cdot 2) = 96$ distinct models. The results are shown in Tables 5.8 (RealWorld) and 5.9 (PaMap2). Values marked in red are values that were unobtainable by the design of our generalizability evaluation and therefore were taken from the corresponding adaptability experiment to preserve comparability between trained models (since missing values would impact μ_{gen}, σ_{gen} and $F1^{min}_{1_{gen}}$ and therefore GI differently based on the training device).

When comparing the results to the adaptability experiments on RealWorld, we notice a much more even distribution of GIs compared to $ADIs$, with only *forearm* as the single anchor standing out with a notably higher GI (0.574) than any other device. This suggests a pretty even spread of best replacement devices for each evaluation device. Supporting this fact, we see that every device except *head* and *shin* serves as the best replacement for another device in this generalizability experiment, indicated by **bold** numbers in the associated columns (highest F1 score in each row without considering red values). An explanation could be a relatively even distribution of useful features learned from each device. Meaning, that data from every device contributes to learning features that are useful for classification

on all devices, with *forearm* data providing the most widely spread features, as already seen in the self-supervised collaborative baseline (Section 4.5.1). The same training approach once again performs worse on PaMap2, with large differences between the choices for the single anchor. *Chest* however, performs once again notably better than *hand* and *ankle*. Especially, when looking at the model that was trained using a 50/50 split of 50% multi-anchor with *chest* and *ankle* and 50% on *ankle*. It achieves an F1 score of 0.804, compared to the similar model in the adaptability experiment, which also used *hand* data during multi-anchor training ($F1 = 0.550$). A similarly large difference can be observed in the 50/50 training using $\{chest, ankle\}, \{chest\}$ with an increase of 0.229 (40.7%) compared to its adaptability counterpart, which also included *hand* data in the multi-anchor training. These examples suggest that a mix of more general (multi-anchor) and specific (single anchor) training can negatively impact the ability of a model to enhance features from known data distributions. A reason for this could be the difficult process of unlearning useless features from the warm-up training. Therefore, it can be beneficial in many cases to completely exclude data from training, even in adaptability applications. This is supported by the previous results of pure multi-anchor training, where similar observations were made. This effect might be more severe on smaller datasets.

Similar to pure multi-anchor training, we can see the trend that multi-fine-tuning results in slightly lower GIs for RealWorld (6.3%) and slightly increased GIs on PaMap2 (3.6%), supporting our previous hypothesis that its performance being anti proportional to the number of devices. This would also imply that multi-fine-tuning could perform better in generalizability applications than adaptability applications.

After analyzing the warm-up training and gaining new insights into effects of different training methods, we now present the results for out implemented *Adaptation Layer*. Further, we investigate the different observations we made in the previous experiments, e.g. better performance of multi-fine-tuning on smaller datasets, benefits of multi-fine-tuning on specific devices (lowest, highest results) and the performances of different device groups in training (similar to G1, G2 from adaptability experiments) using the results from these experiments.

5.2.3 Adaptation Layer

The experiments we performed using our implemented *Adaptation Layer* mainly focus on the architecture of the resulting models feature extractor. The first parameter that we tracked was the placement of a single adaptation layer. We investigated the replacement of each of the three convolution layers in the feature extractor regarding the effect on adaptability and generalizability. During the evaluation pro-

		anchor devices (last 50% of training)						
	F1-scores	**head**	**chest**	**waist**	**forearm**	**upper arm**	**shin**	**thigh**
evaluation devices	**head**	0.714	0.687	0.664	**0.743**	0.657	0.678	0.709
	chest	0.739	0.648	0.683	**0.780**	0.689	0.682	0.756
	waist	0.886	**0.903**	0.854	0.889	0.874	0.877	0.882
	forearm	0.718	0.722	0.722	0.801	0.747	0.748	**0.761**
	upper arm	0.797	0.794	**0.808**	0.799	0.791	0.803	0.791
	shin	0.652	0.657	0.658	0.656	**0.670**	0.669	0.648
	thigh	0.844	**0.861**	0.783	0.812	0.775	0.804	0.813
	$\mu_{\mathbf{gen}}$	0.764	0.753	0.739	**0.783**	0.743	0.752	0.766
	$\sigma_{\mathbf{gen}}$	0.082	0.101	0.077	**0.071**	0.078	0.080	0.075
	GI	0.561	0.551	0.551	**0.574**	0.553	0.561	0.562

Table 5.8: Run of the ColloSSL [17] model using warm-up training on the *RealWorld* [38] dataset. Each combination of *anchor device* for the last 50% of the training and *evaluation device* was evaluated for **different** devices. The best results for each experiment are **bold**. Values in red are obtained from the corresponding adaptability experiment.

		anchor devices (last 50%)		
	F1-scores	**hand**	**chest**	**ankle**
eval devices	**hand**	0.719	0.792	**0.804**
	chest	0.407	0.502	**0.416**
	ankle	0.707	**0.731**	0.600
	$\mu_{\mathbf{gen}}$	0.611	**0.675**	0.607
	$\sigma_{\mathbf{gen}}$	0.177	**0.153**	0.194
	GI	0.392	**0.458**	0.389

Table 5.9: Run of the ColloSSL [17] model using warm-up training on the *PaMap2* [33] dataset. Each combination of *anchor device* for the last 50% of the training and *evaluation device* was evaluated for **different** devices. The best results for each experiment are **bold**. Values in red are obtained from the corresponding adaptability experiment.

cess of our different implementations of the Adaptation Layer, we ran into some troubles that we want to mention briefly. Using our first implementation of the Adaptation Layer concept using the *SelectLayer* class and manually placing the required convolution layers, we ran tests using *cycle* multi-anchor batch training. However, while analyzing the results, we discovered a mistake in the fine-tuning process of the trained models. The last freeze layer during the fine-tuning process

was not adjusted in accordance with the newly added convolution layers, therefore making the results not entirely comparable to our established baselines. Hence, we repeated the experiments. However, at this point, we had already adjusted our implementation to the revised version and therefore chose to only repeat the experiments with the revised version (using the *AdaptationLayer* class) due to time constraints.

Adaptability

The adaptability of different models using adaptation layers was assessed by training a feature extractor while replacing one of its convolution layers with an *AdaptationLayer* instance. The adaptation layers contain sub-layers for each device in the associated dataset (7 for RealWorld, 3 for PaMap2). The feature extractors were then fine-tuned using both single and multi-fine-tuning. This resulted in a total of $3 \cdot (7 + 3) + (3 + 3) = 36$ distinct models. The results of the single fine-tuning evaluation for both datasets are shown in Table 5.10. With an ADI of 0.563, *c-a-c* performs best on RealWorld, whilst being surpassed by *c-c-a* on PaMap2 by a small margin (0.008). Particularly, *a-c-c* seems to perform poorly on both datasets, which would suggest that distinct features at a low level within the feature extractor tend to hinder the model to learn more general features on a higher level. When comparing *a-c-c* and *c-a-c* to *c-c-a*, we have to take into consideration that only in the *c-c-a* architecture are the adaptation layers unfrozen during fine-tuning. This could influence the behavior of the adaptation layer by receiving more training data than the other two architectures. However, despite this distinction, *c-c-a* does not appear to perform to a higher degree than *c-a-c* when looking at the combined results from both datasets. Rather, the opposite can be observed on RealWorld. This could imply that the classification head can work better with the output of a combined convolution layer rather than the specific features that are learned within the sub-layers of the adaptation layer.

When analyzing the results of the multi-fine-tuned models (see Table 5.11), we observed large discrepancies in F1-scores between the different devices. On RealWorld, using multi-fine-tuning results in a decrease of 34.7% in ADI on average, and on PaMap2 the decrease is 9.7%. These results suggest that the training of one classification head for all devices does not work well with a feature extractor that learned device-specific features at any level of its architecture. Despite our effort to align the sub-layer choice with the supervised training (see Section 3.3 for more details), the classification head appears not to be able to learn from the distinct features in a meaningful manner. Our hypothesis is that the specific features provided by the adaptation layer could cause contradicting gradients during the fine-tuning, which would hinder the model's convergence. In Section 5.2.3, we

		RealWorld				PaMap2		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.669	0.696	0.638	**hand**	0.586	0.764	0.790
	chest	0.699	0.700	0.632	**chest**	0.414	0.491	0.508
	waist	0.861	0.894	0.865	**ankle**	0.621	0.710	0.680
	forearm	0.790	0.783	0.782				
	upper arm	0.798	0.789	0.781				
	shin	0.582	0.655	0.646				
	thigh	0.812	0.876	0.796				
	$\mu_{\mathbf{ad}}$	0.744	**0.770**	0.734		0.540	0.655	**0.659**
	$\sigma_{\mathbf{ad}}$	0.098	**0.092**	0.094		**0.111**	0.145	0.142
	ADI	0.527	**0.563**	0.538		0.372	0.446	**0.454**

Table 5.10: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *single* fine-tuning. The best results for each experiment are **bold**.

explore a possible solution to this problem by adding a regulatory term in addition to the contrastive loss during the training of the feature extractor.

		RealWorld				PaMap2		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.380	0.340	0.389	**hand**	0.535	0.582	0.822
	chest	0.652	0.647	0.499	**chest**	0.395	0.401	0.518
	waist	0.474	0.677	0.664	**ankle**	0.476	0.568	0.620
	forearm	0.491	0.425	0.260				
	upper arm	0.548	0.790	0.688				
	shin	0.639	0.654	0.656				
	thigh	0.746	0.706	0.731				
	$\mu_{\mathbf{ad}}$	0.561	**0.606**	0.555		0.469	0.517	**0.653**
	$\sigma_{\mathbf{ad}}$	**0.125**	0.161	0.177		**0.070**	0.101	0.155
	ADI	0.370	**0.372**	0.320		0.339	0.359	**0.451**

Table 5.11: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *multi* fine-tuning. The best results for each experiment are **bold**.

As another solution, we integrated the possibility to also adapt the classification head using sub-layers for its fully connected layers. We repeated the previous experiments using this approach. The results are depicted in Table 5.12. We see

a significant increase compared to multi-fine-tuning with a regular classification head. The adaptation classifier even outperforms single fine-tuning by 2.7% on RealWorld and 5.8% on PaMap2 when averaging over all three architectures. This increase in ADI is mostly caused by increased minimum F1-scores and reduced σ_{ad} since the average F1-score is almost identical for both fine-tuning methods.

		RealWorld				PaMap2		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.732	0.682	0.726	**hand**	0.823	0.784	0.836
	chest	0.692	0.659	0.763	**chest**	0.502	0.504	0.507
	waist	0.750	0.829	0.800	**ankle**	0.530	0.753	0.666
	forearm	0.733	0.766	0.753				
	upper arm	0.789	0.781	0.791				
	shin	0.648	0.694	0.657				
	thigh	0.792	0.832	0.752				
	$\mu_{\mathbf{ad}}$	0.734	**0.749**	**0.749**		0.618	**0.680**	0.670
	$\sigma_{\mathbf{ad}}$	0.051	0.071	**0.048**		0.178	**0.153**	0.165
	ADI	0.551	0.558	**0.562**		0.424	**0.461**	0.454

Table 5.12: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *multi* fine-tuning with an adaptation classifier. The best results for each experiment are **bold**.

However, when comparing these results to models that were trained using multi-anchor training without any adaptation layers (refer to Tables 5.2 and 5.3 for exact numbers), we see no significant improvement in ADI. This is supported by the comparison between single fine-tuning, regular multi-fine-tuning, and adaptation multi-fine-tuning, which we investigated using the same feature extractor trained using *cycle* multi-anchor (batch) training. The recoded measures are outlined in Tables 5.13 and 5.14for RealWorld and PaMap2, respectively. We only note a significant difference to the adaptation layer experiments for PaMap2, where the *c-a-c* and *c-c-a* architectures increase the ADI by 9% and 7.3% compared to multi-anchor training without any adaptation layers, respectively. This indicates a similarity to the multi-fine-tuning, where we observed better performances with fewer devices in a dataset.

Another consideration might be the data distributions of the two datasets. The MMD, which is used for the collaborative training of the feature extractor, is lower on average between arbitrary pairs of devices in PaMap2 than in RealWorld (average $MMDs$ of 0.2545 and 0.3335 respectively). This can mean that the sub-layers of the *AdaptationLayers* do not diverge as much in their learned feature represen-

tation. Therefore, simplifying the training for subsequent layers in the network. We try to emulate this reduction in feature distance with the previously mentioned regulatory term in the "Preliminary Results" section 5.2.4.

			Multi-fine-tuning	
	F1-scores	**Single ft**	**Regular**	**Adapt.**
evaluation devices	**head**	0.740	0.597	0.706
	chest	0.680	0.744	0.708
	waist	0.870	0.856	0.876
	forearm	0.798	0.573	0.818
	upper arm	0.795	0.822	0.786
	shin	0.657	0.689	0.662
	thigh	0.837	0.776	0.817
	$\mu_{\mathbf{ad}}$	**0.768**	0.722	**0.768**
	$\sigma_{\mathbf{ad}}$	0.079	0.108	**0.077**
	ADI	0.565	0.511	**0.567**

Table 5.13: Multi-fine-tuning using an adaptation classifier compared to multi-fine-tuning on a single classification head using self-supervised collaborative *cycle* **multi-anchor** training on *RealWorld*.

			Multi-fine-tuning	
	F1-scores	**Single ft**	**Regular**	**Adapt.**
eval devices	**hand**	0.712	0.744	0.801
	chest	0.488	0.494	0.469
	ankle	0.594	0.553	0.611
	$\mu_{\mathbf{ad}}$	0.598	0.597	**0.627**
	$\sigma_{\mathbf{ad}}$	**0.112**	0.131	0.167
	ADI	**0.423**	0.421	0.421

Table 5.14: Multi-fine-tuning using an adaptation classifier compared to multi-fine-tuning on a single classification head using self-supervised collaborative *cycle* **multi-anchor** training on *PaMap2*.

In summary, we gathered several insights from applying adaptation layers to the feature extractor. Firstly, placing an adaptation as the first layer within the feature extractor hinders the model to learn shared features through collaborative training. Secondly, regular multi-fine-tuning does not work well with adaptation layers, probably because of very distinct features learned in each sub-layer, which results in contradicting gradients during fine-tuning. A fix for this problem is the

introduction of an adaptation classifier, which creates a separate classification head for each sub-layer. This emulates single fine-tuning on all devices at the same time. And lastly, the benefit of adaptation layers gets lost when learned features deviate too much. Therefore, adaptation layers (especially in combination with multi-fine-tuning) could perform better with fewer devices and possibly more similar data distributions within a dataset.

Generalizability

To assess the generalizability potential of adaptation layers, we trained models using the three architectures used for the adaptability (*a-c-c*, *c-a-c*, *c-c-a*). For each of the architectures, we trained a feature extractor using *cycle* multi-anchor batch training with one device held out from the training data. The resulting feature extractors were then fine-tuned using both single and multi-fine-tuning, for a total of $3 \cdot (2 \cdot (7 + 3)) = 60$ distinct models. The resulting measures for the single fine-tuned models are shown in Table 5.15. We can observe the continuation of the trend that the method scores higher values in GIs than its comparable ADI counterpart. This strengthens our hypothesis that leaving data from the evaluation device out while training can improve performance when using multi-anchor training.

		RealWorld				**PaMap2**		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.676	0.707	0.647	**hand**	0.785	0.796	0.760
	chest	0.662	0.700	0.666	**chest**	0.526	0.509	0.435
	waist	0.868	0.874	0.780	**ankle**	0.700	0.704	0.643
	forearm	0.806	0.782	0.751				
	upper arm	0.809	0.780	0.727				
	shin	0.667	0.683	0.673				
	thigh	0.762	0.798	0.771				
	$\mu_{\mathbf{gen}}$	0.750	**0.761**	0.716		**0.670**	**0.670**	0.613
	$\sigma_{\mathbf{gen}}$	0.082	0.068	**0.054**		**0.132**	0.147	0.165
	GI	0.557	**0.572**	0.541		**0.467**	0.458	0.404

Table 5.15: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *single* fine-tuning. The best results for each experiment are **bold**.

As a second observation we can notice that placing the adaptation layer as the middle layer of the feature extractor (*c-a-c* architecture) scores (similar to adaptability) the highest combined GI over both datasets with a small margin over the *a-*

c-c architecture (average GIs over both datasets of 0.515 and 0.512 respectively). Also, *c-c-a* performs worst of all three architectures on both datasets (average GI over both sets of 0.473). These results support the hypothesis that the most useful distinct features for specific devices are learned at a medium data scope (with each convolution layer increasing the amount of data influencing the output).

When comparing the generalizability of adaptation layers to the multi-anchor training without adaptation layers, we see an increase in all three architectures over (regular) *cycle* multi-anchor batch training for RealWorld and an increased GIs for *a-c-c* and *c-a-c* on PaMap2. In contrast to the adaptability, where we saw no noticeable difference between models with or without adaptation layers on RealWorld, the approach shows consistent improvements for generalizability on both datasets. This could indicate that the *replacement* features, which are learned by the sub-layer that replaces the evaluation devices' sub-layer (its weights to be specific) for fine-tuning, help the model to learn the evaluation devices' features with higher accuracy. As our choice of replacement device, we use the device with the lowest (pairwise) MMD regarding the evaluation device. However, other replacement strategies, such as a weighted sum (weighted by inverse MMD) of some or all devices, could be the subject of future research to try to increase the benefits of the adaptation layer. The improvements using an adaptation layer regarding GI are more prominent on PaMap2, which supports our theory that fewer devices and less difference in data distributions could favor the use of adaptation layers.

When analyzing the results for multi-fine-tuning (see Appendix B, Table B.3 for details), we see similar results to the adaptability experiments. The average GI over all three architectures is significantly reduced compared to multi-anchor training without adaptation layers, with the only exception being the *a-c-c* architecture on PaMap2. This underlines the dissonance between learning device-specific features within the adaptation layer and the training of a single classification head to make accurate predictions for each device. This again encourages us to include a regulatory loss for the features learned in the adaptation layer to counteract this divergence in sub-layer outputs.

We repeated all generalizability experiments using the adaptation classifier discussed in the previous section in combination with multi-fine-tuning. Table 5.16 depicts the resulting measures. The first thing to spot is the recurring trend of GIs being slightly higher than $ADIs$ of the similar adaptability experiments. When comparing the results to single fine-tuning for the same models, we see a slight increase in GIs, which is caused by higher values for the *c-c-a* architecture (for both RealWorld and PaMap2). This increase is hard to explain, since multi-fine-tuning with an adaptation classifier would logically be almost indistinguishable from single fine-tuning for each device separately by design. Therefore, these differences are most likely caused by nondeterministic training behavior. However, the simul-

taneous occurrence in both datasets for the same architecture could also point to another interaction with the fine-tuning of the sub-layers, which we are currently unaware of (since *c-c-a* is the only architecture where the sub-layers are not frozen during fine-tuning).

		RealWorld				PaMap2		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.689	0.679	0.729	**hand**	0.770	0.791	0.824
	chest	0.694	0.694	0.722	**chest**	0.505	0.494	0.521
	waist	0.895	0.873	0.827	**ankle**	0.741	0.649	0.694
	forearm	0.715	0.772	0.759				
	upper arm	0.779	0.791	0.791				
	shin	0.656	0.678	0.656				
	thigh	0.772	0.869	0.805				
	$\mu_{\mathbf{gen}}$	0.743	**0.765**	0.756		0.672	0.645	**0.680**
	$\sigma_{\mathbf{gen}}$	0.081	0.085	**0.059**		**0.145**	0.149	0.152
	GI	0.552	**0.569**	0.563		0.458	0.441	**0.466**

Table 5.16: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *multi* fine-tuning with an adaptation classifier. Data from the evaluation device was left out during training of the feature extractor. The best results for each experiment are **bold**.

To summarize our findings from the generalizability experiments, we can note three main observations. The first is in accordance with previous sections, where we noticed a trend of generalizability models achieving higher GIs than $ADIs$ for similar adaptability models. This effect was repeatedly observed for adaptation layers regarding multiple fine-tuning methods. This emphasizes the potential of looking deeper into this phenomenon.
Secondly, adaptation layers seem to increase the generalizability of a model. This is underlined by the comparison to multi-anchor training without any adaptation layers. The addition of device-specific features from a *replacement* device appears to enhance the learning of features for the evaluation device during fine-tuning and therefore increases classification performance on the evaluation device. This increase seems to be more significant on a dataset with fewer devices (larger differences in GI on PaMap2).
And lastly, the addition of an adaptation classifier in combination with multi-fine-tuning enables us to have the benefits of single fine-tuning within a single model for an increase in training time and model size. Additionally, it also performs well with any of the three tested architectures (in contrast to multi-fine-tuning and pos-

sibly single fine-tuning).

Regulation Loss

This section discusses the results of repeating the previous experiments regarding adaptability and generalizability with the added regulatory loss. Details about the implementation of this loss can be found in Section 3.3. The main point of the addition of the regulatory loss was the interaction of adaptation layers and multi-fine-tuning. We stated the hypothesis that learning useful classification features during multi-fine-tuning is impeded by the introduction of device-specific sub-layer outputs. Therefore, we focused this evaluation on the effects of this loss for multi-fine-tuning specifically and only briefly mention its impacts on single fine-tuning. We trained a model for each of the three architectures on both datasets and fine-tuned them using multi-fine-tuning. The results are depicted in Table 5.17. When comparing these results against the multi-fine-tuning results without a regulation loss, we see significant improvements in ADI for almost all architectures on both datasets. The average ADI increases by 32.7% by using a regulation loss. In contrast to these large increases, we see a **decrease** of 6.1% in comparison to multi-fine-tuning with an adaptation classifier on RealWorld and an **increase** of 2.0% on PaMap2. However, this slight decrease come with the upside of having a smaller model and therefore faster training times.

Additionally, we repeated the generalizability experiments from the previous section with the regulatory loss. When compared to the associated experiments without the regulatory loss, the GI increases by 32.4% and 3.1% for RealWorld and PaMap2 respectively. This large difference in increases suggests a more significant benefit of the regulatory loss for a larger number of sub-layers (i.e., number of devices in the dataset for those tests). This correlation seems sensible as the classifier has to adapt to more distinct outputs on RealWorld than on PaMap2 with the same number of weights. Since the regulatory loss helps to align the outputs more closely, this addition should be expected to be more beneficial with more sub-layers. A Table with all measures is provided in Appendix A (Table B.4). The results again show an increased GI compared to ADI for RealWorld. However, for PaMap2 this is not the case. Thus, it is the first experiment with a higher ADI than GI (on average) in all experiments regarding our different implementations. The main difference between the adaptability and generalizability experiments was observed in the F1-scores of *chest*, which were reduced by 8.9% on average (across all three architectures). However, when comparing the F1-scores of *chest* to other experiments, we notice that these values are not statistical outliers but rather the opposite is the case. The F1-scores of *chest* in the adaptability experiments are significantly higher than in all other experiments on PaMap2 (on two of the three

architectures). This could be attributed to statistical variance between runs (see Section 4.2 for details about in-deterministic behavior). Another explanation would be a more precise classification caused by closer outputs of sub-layers due to the regulatory loss. This would benefit the classification of *chest* data in particular if the distribution of the mean output of all sub-layers (towards which the other outputs are encouraged to converge by the loss) is close to the data distribution of the learned *chest* features. The same benefit would not occur for generalizability training, as the *chest* sub-layer would not receive any training and its weights are replaced by the ones of the closest neighboring device before fine-tuning.

When comparing the generalizability of the regulatory loss models to the adaptation classifier models, we notice a decrease in GI by 4.0% on average over both datasets. While both approaches produce models that can efficiently classify actions for all devices within a dataset, the introduction of the regulatory loss allows for the training of a single classification head, resulting in a smaller model size with comparable results.

We also applied the regulatory loss to the training of single fine-tuned models to see the effect on the performance. Table B.5 stating all results is provided in Appendix A. Compared to adaptation layer training without the regulation loss, we see a decrease in performance for the *c-a-c* architecture for both adaptability and generalizability ($ADIs : 0.563$ vs. 0.533 and $GIs : 0.572$ vs. 0.546). For both other architectures, we see either no significant change or even increases in ADI or GI. Especially on PaMap2, the regulation loss seems to produce more stable results for any architecture, as the performance differences are significantly smaller than without the regulatory loss.

All following adaptation layer experiments were performed without the regulatory loss unless explicitly stated otherwise.

5.2.4 Preliminary Results

This section contains preliminary results from experiments that were performed in accordance with different observations or theories, but were not tested in a fully structured manner because of time constraints. These additional experiments mainly focus on adaptability rather than generalizability due to the number of models that must be trained being significantly reduced for adaptability experiments.

The first experiments we discuss focus on the use of groups of devices for the adaptation sub-layers rather than specific devices. With these experiments, we wanted to investigate two observations from previous sections. First, the finding that the classification performance on different devices within the RealWorld dataset benefited or was harmed by the use of multi-fine-tuning. Different group layouts in accordance with the results from Section 5.1, where we recorded signif-

		RealWorld				PaMap2		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.641	0.656	0.745	**hand**	0.748	0.756	0.810
	chest	0.603	0.693	0.702	**chest**	0.582	0.550	0.473
	waist	0.728	0.846	0.846	**ankle**	0.569	0.665	0.624
	forearm	0.619	0.666	0.753				
	upper arm	0.679	0.755	0.800				
	shin	0.658	0.619	0.613				
	thigh	0.721	0.763	0.710				
	$\mu_{\mathbf{ad}}$	0.664	0.714	**0.738**		0.633	**0.657**	0.636
	$\sigma_{\mathbf{ad}}$	**0.048**	0.078	0.075		**0.100**	0.103	0.169
	ADI	0.503	0.527	**0.538**		0.467	**0.473**	0.426

Table 5.17: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using *multi* fine-tuning with the inclusion of a regulatory loss during training of the feature extractor. The best results for each experiment are **bold**.

icant losses in classification performance for *forearm* and *head*, as well as significant gains for *chest* and *shin*. This observation resulted in the following groups for the adaptation sub-layers:

$$G1 = [forearm],\ G2 = [head],\ G3 = [chest, shin, waist, upperarm, thigh]$$

Since classification performance decreased on *forearm* and *head*, we opted to assign them their device-specific sub-layers. Due to *chest* and *shin* benefiting from multi-fine-tuning, we chose to keep them in a large group with the remaining devices (which showed no significant difference between multi-fine-tuning and single fine-tuning). We trained a model using this adaptation group layout with an adaptation classifier. The adaptation classifier should therefore allow single fine-tuning for *forearm* and *head* whilst using multi-fine-tuning for the other devices. The model was trained using the known parameters with *cycle* multi-anchor training on the *c-a-c* architecture. The resulting model achieved values of $\mu_{ad} = 0.714, \sigma_{ad} = 0.075, F1_{ad}^{min} = 0.630$ and $ADI = 0.531$. This is a slight improvement compared to the multi-anchor baseline (using multi-fine-tuning) with $ADI = 0.517$, a significant increase compared to adaptation layers with multi-fine-tuning ($ADI = 0.372$), and a slight decrease compared to adaptation layers with regulation loss (this experiment was performed before the introduction of the new loss). The desired effect was noticeable with both *forearm* and *head* improving significantly compared to regular multi-anchor training. However, performance

on *chest* decreased significantly, which suggests that data from either *forearm* or *head* (or both) was part of the increase in classification performance for *chest*. This observation outlines a challenge for the introduction of groups, as the benefits for specific devices from multi-anchor training and single anchor training have to be considered for the group selection. Yet, this also leaves room for more research into this topic. Further experiments using adaptation groups were performed by grouping the devices according to their body position (limbs vs. torso), with the following groups:

$$G1 = [forearm, upperarm],\ G2 = [shinthigh],\ G3 = [head, chest, waist]$$

and

$$G1 = [forearm, upperarm, shinthigh],\ G2 = [head, chest, waist].$$

These experiments showed similar results to the previous one. Though the performance improved significantly compared to multi-fine-tuning with different sublayers for each device, the performance on individual devices (mostly *forearm*) suffered from the formed groups. This further strengthens the causation between the number of devices (here, groups) and the performance of multi-fine-tuning with a single classification head.

To explore the idea of training a single model to classify all devices with high accuracy, we constructed a feature extractor consisting purely of adaptation layers (corresponding architecture: *a-a-a*). Additionally, we used an adaptation classifier to fully emulate single anchor training on all devices simultaneously. We only trained this model on RealWorld due to long training times. The results were as expected. The model performs very similarly to our single anchor baseline and the results from *Jain and Tang et al.* [17]. It achieved an $ADI = 0.577$, which is the highest we recorded for RealWorld. However, in comparison to training 7 separate models for each device, the *a-a-a* feature extractor uses the same amount of training data as single anchor training for one device. This is caused by always using the same (full) dataset in training. The only variable that changes for the *a-a-a* model is the number of positive pairs it receives for each device, since we cyclically switch the anchor device and therefore the device that determines the positive pairs in between batches. Hence, achieving a faster training time and possibly better results on datasets with fewer examples than single anchor training. To fully test the learning capability of the model, we conducted an experiment that also reduced the number of labeled data during fine-tuning to the same number as single fine-tuning (meaning $\frac{1}{7}$ of multi-fine-tuning). Interestingly, the resulting model still achieves an $ADI = 0.531$, which is comparable to the regular multi-anchor baseline ($ADI = 0.540$), which uses *full* multi-fine-tuning. This concept

could be further explored to assess the impact of adaptation layers in low data scenarios.

The last experiments we conducted focus on the utilization of the *adapt mode* to vary the training of adaptation layers between device/group-specific training and general training. When *enabled*, the adaptation layer only trains the sub-layer of the selected input. When the adaptation mode is *disabled*, the adaptation layer trains all sub-layers simultaneously by forwarding the average over all outputs. For full details, refer to section 3.3. For the experiments regarding the use of the adaptation mode, we opted to test the behavior using a fully adaptable architecture (a-a-a). We took inspiration from our implementation of the multi-anchor training and tested three different adapt mode switching methods. For all three methods, the adaptation layers' adapt modes are set to false at the start of training, and the adaptation mode is selectively enabled for only one adaptation layer at a time. We made this choice to limit the impact of adaptation layer training on the resulting feature extractor, as we learned from previous experiments that too much divergence between the outputs of a layer can negatively impact the performance of the model. Therefore, we wanted to start with relatively small changes. We trained one feature extractor for each of the three switching methods *cycle*, *divided*, and *randomized* for both datasets and evaluated the models using single and multi-fine-tuning. Each model was trained using the regulatory loss. Table B.6 showing the full results is provided in Appendix B. We found that this training approach resulted in a decrease in performance on RealWorld across all three modes. For PaMap2, this training showed a very small increase in average ADI for single fine-tuning when compared to average ADI of adaptation layer training with only one adaptation layer (0.459 vs. 0.452). When compared to the best performing single adaptation layer training, it is a decrease of 0.007. This may indicate that the frequent switching of the adaptable layer might reduce the benefits of adaptation layer training that are provided by a constantly adaptable single adaptation layer within the feature extractor. A noticeable observation is that the best results were achieved for three out of the four test scenarios (RealWorld, PaMap with both single and multi-fine-tuning) with the *randomized* adaptation mode training option. This suggests a tendency for less structured training to profit most from multiple adaptation layers. However, further experiments are necessary to draw any conclusions.

After discussing these additional experiments and their preliminary results, we now take a closer look at the best-performing models from each group of experiments and discuss their best-suited use cases.

5.2.5 Best Performing Models

This section serves as a summary of the most noticeable models from each experiment section. To evaluate their performance, we opted to re-run some of the experiments to reduce the chance of them being outliers due to indeterministic training behavior. The following list shows the experiments for each section that we considered to have performed best in their respective categories:

- Single Anchor Training (Baseline ColloSSL):
  - RealWorld: forearm, PaMap2: chest
- Multi-Anchor Training:
  - RealWorld + PaMap2: cycle batches
- Warm-up Training:
  - RealWorld: forearm, PaMap2: chest (as second stage devices)
- Adaptation Layer:
  - RealWorld + PaMap2: c-a-c, c-c-a architectures *with regulation loss*

For each of these models, we ran additional experiments to confirm the results. For most experiments, we were able to get two additional test runs, resulting in averages over three results. Due to unforeseen circumstances regarding the OMNI Cluster [32], however, some experiments did not finish the second round of reevaluations in time, hence giving only two data points instead of three to lay the foundation for our final discussion.

Single Anchor Training

For the baseline training method using a single anchor device, *forearm* and *chest* stood out amongst all single anchor models for RealWorld and PaMap2 respectively. *Forearm* as the anchor device showed the best ADI for single fine-tuning of all baseline models on RealWorld with $ADI = 0.570$. After two additional tests, this ADI further improved to an average of 0.580. For multi-fine-tuning, the ADI also improved by a small margin (0.004). This supports the idea that *forearm* training produces a feature extractor that learns commonly occurring features for most devices in RealWorld. For PaMap2, we chose chest as the best single anchor device due to its consistent performance in both single and multi-fine-tuning in our baseline. Though it did not have the highest ADI for either of the fine-tuning methods, it was the only device in PaMap2 that resulted in similarly high scores for

both fine-tuning methods. Our additional tests, however, showed considerable variance in the single fine-tuning results, with one result being significantly lower than the other two. The average $ADIs = 0.419, 0.425$ for single and multi-fine-tuning, respectively, suggest a less stable performance than previously assumed due to the baseline results. These two single anchor models serve us as a baseline for the upcoming discussions regarding the performance of the implemented changes.

Multi-Anchor Training

During the multi-anchor training experiments, *cycle* multi-anchor batch training showed the most promising results. The method had the highest $ADIs$ (respective GIs) across all test setups amongst all six methods and solid results in all tests. The additional tests confirmed this trend with improvements in both average ADI and GI compared to the single previous experiment. The final results are $ADI = 0.575, 0.524$ and $GI = 0.563, 0.521$ for single and multi-fine-tuning, respectively, on RealWorld. On PaMap2 $ADI = 0.407, 0.396$ and $GI = 0.420, 0.446$ for the respective fine-tuning methods (single, multi) suggest a less stable performance than anticipated. However, this is one of the cases where only two results were obtained. Hence, the average scores are prone to one of the scores being an outlier. These results propose a small performance gain over the baseline multi-anchor training method (*randomized* multi-anchor batches). Further, the *cycle* multi-anchor batch training seems to produce comparable results to the best single anchor trainings (*forearm* on RealWorld and *chest* on PaMap2) and even an improvement over the average single anchor trained model on RealWorld (average $ADI = 0.555$). This result makes the training for a multi-device classification model more straightforward. Rather than testing each single device model regarding its performance, multi-anchor training can be used to produce a model with comparable results to the best single anchor choice.

Warm-up Training

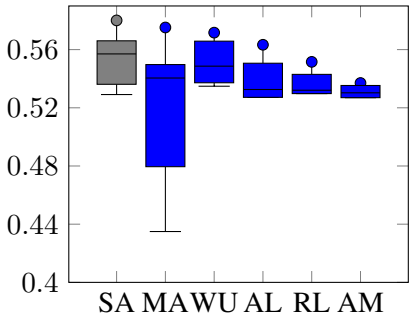
For the warm-up training, we again chose *forearm* and *chest* for the associated datasets as the best performing second stage devices. This decision was based on the combined performance of both adaptability and generalizability models. The additional tests confirmed these choices by showing very similar results to the initial experiments. *Forearm* as the second stage anchor device results in an average $ADIs$ of 0.572 and 0.530 and $GIs = 0.574, 0.524$ for single respective multi-fine-tuning. This closely matches the results from single anchor training. On PaMap2, we record average $ADIs = 0.419, 0.408$ and average $GIs = 0.455, 0.422$. The improvement over regular multi-anchor training is significant for generalizability.

This supports the idea of device-specific features being helpful for the classification of unknown devices. We will conclude on this proposition after analyzing the results of the adaptation layer training.

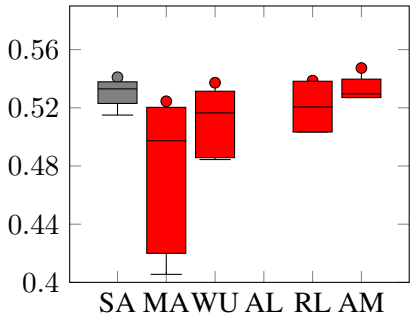
Adaptation Layer

We opted to repeat the tests for both *c-a-c* and *c-c-a* architectures, as both showed promising performances in different scenarios. We opted to use the regulation loss in the repeated runs to benefit the multi-fine-tuning performance. Whilst this could mean a decreased ADI respective GI for specifically the *c-a-c* architecture on RealWorld (see Section 5.2.3 for more details), the overall focus remains on the multi-fine-tuning evaluation. Hence, the use of the regulatory loss. The results confirm the previous observation about the stability of the trained models, as we see very close margins between experiment results for both architectures on both datasets. The results also support the fact that the added loss could benefit the single fine-tuning performance on PaMap2, as we see similar averages to the initial experiment. Overall, the additional tests tend to favor the *c-c-a* architecture, as it achieves a higher average ADI on RealWorld (0.552 compared to *c-a-c* with 0.534). All other results are only marginally different (within 0.001). Therefore *c-c-a* with regulation loss is chosen as the best performing model for the use of adaptation layers. When comparing these results to the best single anchor training models, we see an improvement on the PaMap2 dataset for both single and multi-fine-tuning. Single fine-tuning appears to work particularly well ($ADI = 0.463$) with the use of an adaptation layer (baseline $ADI = 0.419$). Though multi-fine-tuning also shows marginal improvements on PaMap2 for generalizability with a $GI = 0.452$ compared to the baselines $GI = 0.446$.

The results of the additional tests confirm most of our previous observations regarding the benefits of the different training approaches. Especially on PaMap2, the use of an adaptation layer and multi-fine-tuning shows significant improvement over both single and multi-anchor baselines. On RealWorld, the results are mixed. Whilst multi-anchor training shows comparable results to the best single anchor models, the introduction of an adaptation layer does not seem to improve these results. However, the device-specific features from the adaptation layer (with the use of a regulatory loss) tend to benefit the multi-fine-tuning and improve its performance compared to single anchor or regular multi-anchor training.

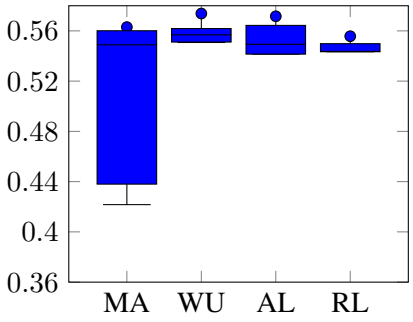


(a) ADI by training method for *single* fine-tuning

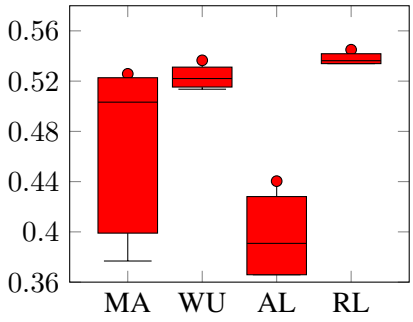


(b) ADI by training method for *multi* fine-tuning

Figure 5.2: Box plots showing ADI for single (a) and multi (b) fine-tuning for the *RealWorld* dataset by training method. Results show improvements for adaptability using multi fine-tuning especially when paired with adaptation layers + regulation loss.



(a) GI by training method for *single* fine-tuning

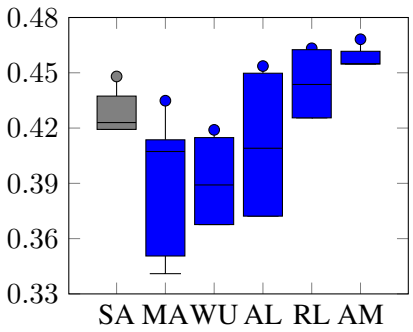


(b) GI by training method for *multi* fine-tuning

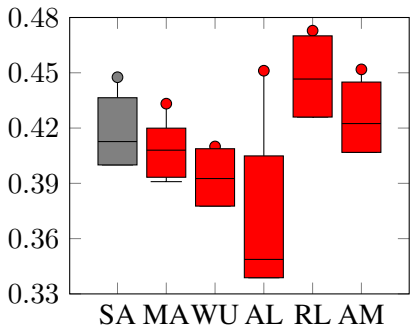
Figure 5.3: Box plots showing GI for single (a) and multi (b) fine-tuning for the *RealWorld* dataset by training method. Results show improvements for generalizability using multi fine-tuning especially when paired with adaptation layers + regulation loss.

5.3 Summary

To place each experiment into the context of all results, we generated multiple box plots (Figures 5.2, 5.3 for RealWorld and Figures 5.4, 5.5 for PaMap2) that show the individual adaptability and generalizability indices side by side. The plots show boxes for each training method. The individual $ADIs$/GIs correspond to

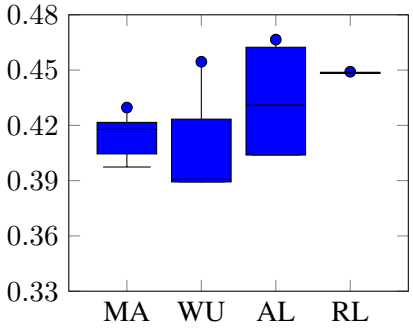


(a) ADI by training method for *single* fine-tuning

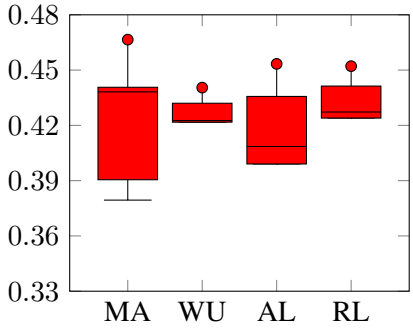


(b) ADI by training method for *multi* fine-tuning

Figure 5.4: Box plots showing ADI for single (a) and multi (b) fine-tuning for the *PaMap2* dataset by training method. Results show improvements for adaptability using multi fine-tuning especially when paired with adaptation layers + regulation loss.



(a) GI by training method for *single* fine-tuning



(b) GI by training method for *multi* fine-tuning

Figure 5.5: Box plots showing GI for single (a) and multi (b) fine-tuning for the *PaMap2* dataset by training method. Results show improvements for generalizability using multi fine-tuning especially when paired with adaptation layers + regulation loss.

the different training variations within each training method. For example, for the multi-anchor training, the adaptability scores for *epoch* and *batch* oriented device switching and all three device selection options (*cycle, divided* and *randomized*) are contributing to the resulting box plot. For the ColloSSL baseline (gray boxes), all devices of the corresponding dataset were used as the *anchor device* and the

resulting $ADIs$ are included into the box plot. All plots include the results from the repetition of experiments from the previous section. The scores of the best performing models for each training method (in accordance with the previous section) are plotted as points for easier comparability.

For both datasets we can clearly see a high variance in $ADIs$ and GIs for multi-anchor training. This is mostly caused by the implementation of the epoch based device switching which resulted in comparably low scores. For warm-up training, we observe a small increase in both ADI and GI for the RealWorld dataset. On PaMap2, the use of warm-up training does not seem to have a positive impact on adaptability nor generalizability. The addition of an adaptation layer, which was inspired by the results of warm-up training show promising results in generalizability scenarios for both datasets. We note that the performance of multi-fine-tuning with the use of adaptation layers is considerably worse then any other result. However, the introduction of the regulation loss especially in conjunction with multi fine-tuning achieves results that are comparable to or even better (in terms of GI) than all single fine-tuned models in generalizability scenarios. This trend can be observed on RealWorld and PaMap2 with regulation loss and multi fine-tuning also achieving high $ADIs$ on PaMap2. This results in a single model that achieves higher scores for both ADI and GI than three separately fine-tuned models for each of the three devices (hand, chest and ankle). Lastly, the introduction of multiple adaptation layers with switching adaptation modes achieve decent results for both datasets. We were only able to conduct experiments for adaptability scenarios. When comparing the results against a single adaptation layer with regulation loss, we do not see any improvement when comparing the best performing models. This seems contra intuitive when considering the vastly higher amount of trainable parameters by replicating all three convolution layers multiple times within each adaptation layer. However, this could be part of future research.

Overall, our experiments suggest that all training methods bring advantages for certain scenarios. Warm-up training whilst useful showing improvements on RealWorld, is dependent on choosing the right device for the second training stage. Adaptation Layers can improve generalizability in both fine-tuning scenarios but also can only cause longer training times without inherent benefit (e.g., adaptability on RealWorld). The addition of the regulation loss improves multi fine-tuning performance significantly but can hinder single fine-tuning performance slightly. And multi fine-tuning can improve both adaptability and generalizability for environments with fewer devices, but reduce performance slightly when more devices are involved. Therefore, multi fine-tuning offers a trade-off between performance and ease of use.

Chapter 6

Conclusion

The goal of this thesis was the improvement of generaliazbility for self-supervised sensor-based human activity recognition. To achieve this goal, we started with a definition of generalizability for the purpose of our research. This was followed by an approach to measure generalizability using statistical quantities of model evaluations. This resulting **generalizability index (**ADI/GI**)** was then used to analyze the collaborative self-supervised learning framework ColloSSL [17] regarding its generalizability. Afterwards, we proposed different changes and additions to the training and fine-tuning process as well as the models architecture to improve these generaliazbility measures. These implemented alterations were then methodically tested and analyzed regarding its impact on ADI/GI through various experiments. The results of these experiments were then compared to the established baselines to evaluate the effects of the adjustments.

The initial evaluation of the ColloSSL framework resulted in two main observations. Firstly, that certain devices in each dataset (e.g., *forearm* for RealWorld [38] and *chest* for PaMap2 [33]) achieve better gerneralizability than other devices. This is most likely caused by the more generally applicable nature of the learned feature from those devices. The second observation was, that leaving training data out for multi device training can improve generalizability compared to training on the full dataset. This circumstance was observed in multiple experiments with few exceptions. As of the writing of this thesis, we do not have a satisfying explanation for this phenomenon and it will be part of future research.

Following the establishing of the baseline, we implemented three major additions to the ColloSSL framework. Firstly, a new fine-tuning method that trains the model for multiple devices at the same time by concatenating the devices training data. This multi-fine-tuning was aimed at the research goal to find a single trained model that performs accurate classification on all devices in a given dataset. The

experiments showed that multi-fine-tuning can achieve an ADI/GI within approximately $5 - 10\%$ of fine-tuning for each device separately. Multi-fine-tuning also showed to increase performance in scenarios with fewer devices (e.g., for three devices on the PaMap2 dataset). This suggests a reasonable result for the *one-model-fits-all* approach. Additionally, multi-fine-tuning appears to benefit single device performance on certain devices. For *chest* and *shin* from the RealWorld dataset, multi-fine-tuning achieves higher F1-scores across all tests compared to single fine-tuning. On the other hand, devices like *head* and *forearm* (RealWorld) lose significant classification performance using multi-fine-tuning. However, this implies that the use of multi-fine-tuning can not only benefit generalizability but also single device classification performance.

The next alteration to ColloSSL was the implementation of different multi-device training modes. We established two different modes (epoch based and batch based switching) and two distinct device selection strategies (*cycle*, *divided*). These were compared to the baseline multi device method (*randomized* batch based switching). We concluded that *cycle* batch based switching provides a small performance benefit over *randomized* switching regarding generalizability. The multi device training was also generalized to allow training on different groups of devices. The proposed warm-up training used this feature to train the model on all devices in the first half of the training and then switch to single device training for the second training phase. This resulted in small performance increases for some devices over multi device training.

We took inspiration of this mixture of general and specific training by implementing a new architecture element. The adaptation layer allows device specific training by the use of sub-layers. With the implementation of an adapt mode, the adaptation layer is also capable of training all sub-layers simultaneously if desired. We used the adaptation layer to replace a standard convolution layer within the feature extractor. In conjunction with a regulatory loss to reduce sub-layer divergence, the resulting model showed promising results especially for the *one-model-fits-all* approach together with the use of multi-fine-tuning by increasing generalizability. Especially for PaMap2, these increases were significant compared to the baseline. This suggests a correlation between the number of device specific sub-layers and the performance increase with higher increases being recorded for fewer sub-layers. Regarding this observation, additional experiments were conducted that split the RealWorld dataset into groups of devices (e.g., arms, legs and torso). However, the preliminary results obtained for those experiments were inconclusive caused by a lack of structured testing due to time restrictions.

Overall, this thesis provided an insight about generalizability in a self-supervised deep learning framework for sensor-based HAR. Different adjustments to the training process were proposed and implemented with varying success of improving

generalizability. In the last section of this thesis we explore additional ideas that could be part of future work.

6.1 Future Work

Due to the time restricted nature of this thesis, there are some adjustments to our implemented additions that we were unable to fully implement and/or test. Firstly, the replacement procedure of the sub-layer weights for generalizability training with adaptation layers. A proposed alternative to the closes neighbor approach of out implementation could be a weighted sum of all other sub-layers weighted by the MMD (similar to the contrastive loss of negative samples but with inverted weights). This could be a more general fit than the replacement by a single device specific sub-layer and therefore increase the generalizability performance even further. Secondly, the use of different regulatory losses. Changes between weight dependent losses and output dependent losses could be interesting to experiment with. Another big research question is the interaction of multi-fine-tuning and multiple unknown devices. This could benefit generalizability to a large degree since single fine-tuning does not allow for the introduction of multiple unknown devices in a single model. Evaluation of this research question could, for example, include cross-dataset generalizability (e.g., training of the feature extractor on ReaWorld and evaluation of PaMap2). And lastly, the sampling process of positive and negative devices. For example the use of retrieval based reconstruction [42] instead of MMD could be of significant interest as it could capture similarities between devices more reliably than pairwise MMD.

6.2 Acknowledgments

This work was supported by the University of Siegen. Special thanks to Chi Ian Tang (University of Cambridge) for his input regarding alterations to the ColloSSL framework [17].

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Antonio A Aguileta, Ramon F Brena, Oscar Mayora, Erik Molino-Minero-Re, and Luis A Trejo. Multi-sensor fusion for activity recognition—a survey. *Sensors*, 19(17):3808, 2019.

[3] Ferhat Attal, Samer Mohammed, Mariam Dedabrishvili, Faicel Chamroukhi, Latifa Oukhellou, and Yacine Amirat. Physical human activity recognition using wearable sensors. *Sensors*, 15(12):31314–31338, 2015.

[4] Yoshua Bengio, Aaron C Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR, abs/1206.5538*, 1(2665):2012, 2012.

[5] Andreas Bulling, Ulf Blanke, and Bernt Schiele. A tutorial on human activity recognition using body-worn inertial sensors. *ACM Computing Surveys (CSUR)*, 46(3):1–33, 2014.

[6] Youngjae Chang, Akhil Mathur, Anton Isopoussu, Junehwa Song, and Fahim Kawsar. A systematic study of unsupervised domain adaptation for robust human-activity recognition. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(1):1–30, 2020.

[7] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PmLR, 2020.

[8] François Chollet et al. Keras. `https://keras.io`, 2015.

[9] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[10] Florenc Demrozi, Graziano Pravadelli, Azra Bihorac, and Parisa Rashidi. Human activity recognition using inertial, physiological and environmental sensors: A comprehensive survey. *IEEE access*, 8:210816–210836, 2020.

[11] Mohamed Elgendy. *Deep learning for vision systems*. Simon and Schuster, 2020.

[12] Python Software Foundation.

[13] Harish Haresamudram, Irfan Essa, and Thomas Plötz. Assessing the state of self-supervised human activity recognition using wearables. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(3):1–47, 2022.

[14] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Saidul Islam, Hanae Elmekki, Ahmed Elsebai, Jamal Bentahar, Nagat Drawel, Gaith Rjoub, and Witold Pedrycz. A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications*, 241:122666, 2024.

[17] Yash Jain, Chi Ian Tang, Chulhong Min, Fahim Kawsar, and Akhil Mathur. Collossl: Collaborative self-supervised learning for human activity recognition. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(1):1–28, 2022.

[18] Yash Jain, Chi Ian Tang, Chulhong Min, Fahim Kawsar, and Akhil Mathur. Collossl: Collaborative self-supervised learning for human activity recognition. `https://github.com/akhilmathurs/collossl`, 2022.

[19] Bulat Khaertdinov, Esam Ghaleb, and Stylianos Asteriadis. Contrastive self-supervised learning for sensor-based human activity recognition. In *2021 IEEE International Joint Conference on Biometrics (IJCB)*, pages 1–8. IEEE, 2021.

[20] Azhar Ali Khaked, Nobuyuki Oishi, Daniel Roggen, and Paula Lago. In shift and in variance: Assessing the robustness of har deep learning models against variability. *Sensors*, 25(2):430, January 2025.

[21] Md Abdullah Al Hafiz Khan, Nirmalya Roy, and Archan Misra. Scaling human activity recognition via deep learning-based domain adaptation. In *2018 IEEE international conference on pervasive computing and communications (PerCom)*, pages 1–9. IEEE, 2018.

[22] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[24] Olga Kosheleva and Vladik Kreinovich. Why deep learning methods use kl divergence instead of least squares: a possible pedagogical explanation. *Mathematical Structures and Modeling*, 46(2):102–106, 2018.

[25] Oscar D Lara and Miguel A Labrador. A survey on human activity recognition using wearable sensors. *IEEE communications surveys & tutorials*, 15(3):1192–1209, 2012.

[26] Frédéric Li, Kimiaki Shirahama, Muhammad Adeel Nisar, Lukas Köping, and Marcin Grzegorzek. Comparison of feature learning methods for human activity recognition using wearable sensors. *Sensors*, 18(2):679, 2018.

[27] Wang Lu, Jindong Wang, and Yiqiang Chen. Local and Global Alignments for Generalizable Sensor-Based Human Activity Recognition. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3833–3837, Singapore, Singapore, May 2022. IEEE.

[28] Uwe Maurer, Asim Smailagic, Daniel P Siewiorek, and Michael Deisher. Activity recognition and monitoring using multiple sensors on different body positions. In *International Workshop on Wearable and Implantable Body Sensor Networks (BSN'06)*, pages 4–pp. IEEE, 2006.

[29] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[30] Henry Friday Nweke, Ying Wah Teh, Uzoma Rita Alo, and Ghulam Mujtaba. Analysis of multi-sensor fusion for mobile and wearable sensor based human activity recognition. In *Proceedings of the international conference on data processing and applications*, pages 22–26, 2018.

[31] Keiron O'shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[32] Gerd Pokorra. Omni cluster university of siegen, Feb 2021.

[33] Attila Reiss and Didier Stricker. Introducing a new benchmarked dataset for activity monitoring. In *2012 16th international symposium on wearable computers*, pages 108–109. IEEE, 2012.

[34] Daniel Roggen, Alberto Calatroni, Mirco Rossi, Thomas Holleczek, Kilian Förster, Gerhard Tröster, Paul Lukowicz, David Bannach, Gerald Pirkl, Alois Ferscha, et al. Collecting complex activity datasets in highly rich networked sensor environments. In *2010 Seventh international conference on networked sensing systems (INSS)*, pages 233–240. IEEE, 2010.

[35] Aaqib Saeed, Tanir Ozcelebi, and Johan Lukkien. Multi-task self-supervised learning for human activity detection. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(2):1–30, 2019.

[36] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[37] Alexander J Smola, A Gretton, and K Borgwardt. Maximum mean discrepancy. In *13th international conference, ICONIP*, volume 6, 2006.

[38] Timo Sztyler and Heiner Stuckenschmidt. On-body localization of wearable devices: An investigation of position-aware activity recognition. In *2016 IEEE international conference on pervasive computing and communications (PerCom)*, pages 1–9. IEEE, 2016.

[39] Chi Ian Tang, Ignacio Perez-Pozuelo, Dimitris Spathis, and Cecilia Mascolo. Exploring contrastive learning in human activity recognition for healthcare. *arXiv preprint arXiv:2011.11542*, 2020.

[40] TensorFlow. tf.config.experimental.enable_op_determinism — TensorFlow v2.16.1 — tensorflow.org. `https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism`, 2024. [Accessed 08-08-2025].

[41] Jinqiang Wang, Tao Zhu, Jingyuan Gan, Liming Luke Chen, Huansheng Ning, and Yaping Wan. Sensor data augmentation by resampling in con-

trastive learning for human activity recognition. *IEEE Sensors Journal*, 22(23):22994–23008, 2022.

[42] Maxwell Xu, Alexander Moreno, Hui Wei, Benjamin Marlin, and James Matthew Rehg. Retrieval-based reconstruction for time-series contrastive learning. In *The Twelfth International Conference on Learning Representations*, 2023.

[43] Ming Zeng, Le T Nguyen, Bo Yu, Ole J Mengshoel, Jiang Zhu, Pang Wu, and Joy Zhang. Convolutional neural networks for human activity recognition using mobile sensors. In *6th international conference on mobile computing, applications and services*, pages 197–205. IEEE, 2014.

[44] Shibo Zhang, Yaxuan Li, Shen Zhang, Farzad Shahabi, Stephen Xia, Yu Deng, and Nabil Alshurafa. Deep learning in human activity recognition with wearable sensors: A review on advances. *Sensors*, 22(4):1476, 2022.

Appendix A

Source Codes

The following parameters were obtained from the *collossl_single_run.sh* within the ColloSSL GitHub repository [17, 18] and were used for all experiments (unless explicitly stated otherwise):

- *training_epochs*: 100
- *fine_tune_epochs*: 100
- *multi_sampling_mode*: unsync_neg
  - Synchronized the samples from positive pairs and unsyncs the samples from negative pairs.
- *held_out_num_groups*: 5, *held_out*: 0
  - Number of groups for leave-one-group-out evaluation and index of the left out group.
- *device_selection_strategy*: closest_pos_all_neg
  - Selects only the closest device (regarding MMD) as the positive device and uses **all** devices as negative devices.
- *neg_sample_size*: 1
  - Number of samples taken from **each** negative device during contrastive training.
- *dynamic_device_selection*: True

  - Whether to select the positive and negative devices dynamically during training or once apriori.

- *device_selection_metric*: mmd_acc_norm
  - Compute MMD normalized over all channels.
- *weighted_collossl*: True
  - Assigns weights to negative samples anti-proportional to their (MMD) distance to the anchor device (higher distance = lower weight).
- *training_batch_size*: 512
- *contrastive_temperature*: 0.05
  - Hyperparameter for the NT-Xent loss during contrastive training
- *optimizer*: Adam [23]
- *learning_rate*: 0.0001

Source Code A.1: Source code of the custom Keras layer class *AdaptationLayer*. To improve readability, some functions and sections of the code may be left out or altered slightly.

```
import tensorflow as tf

@tf.keras.utils.register_keras_serializable()
class AdaptationLayer(tf.keras.layers.Layer):
    def __init__(self,
                 num_layers=3,
                 sub_layer_type=tf.keras.layers.Dense,
                 sub_layer_kwargs=None,
                 adapt_mode=False,
                 **kwargs):
        """
        Adaptation layer with multiple sublayers (Dense or Conv)
        ↪  and optional routing.
        Args:
            num_layers: Number of sublayers.
            sub_layer_type: Layer class, e.g., Dense or Conv1D.
            sub_layer_kwargs: Dictionary of kwargs for sublayers.
```

```
            adapt_mode: If True, route inputs through chosen branch, else average.
        """
        # Create sublayers
        self.sub_layers = [
            sub_layer_type(**self.sub_layer_kwargs, name=f"adapt_sub_layer_{i}")
            for i in range(self.num_layers)
        ]

    def call(self, inputs):
        #Inputs: [data, select_id]
        data, chosen_idx = inputs

        # Compute all sublayer outputs
        branch_outputs =
        [layer(tf.identity(data)) for layer in self.sub_layers]
        stacked = tf.stack(branch_outputs, axis=0)
        # shape: (num_layers, batch, ...)

        @tf.custom_gradient
        def choose_and_route(stacked_out, idx, adapt_mode_bool):
            def forward():
                def adapt_true():
                    if idx.shape.rank == 0: # per batch training
                        return stacked_out[idx]
                    else: # per sample training
                        batch_size = tf.shape(idx)[0]
                        batch_range = tf.range(batch_size, dtype=tf.int32)
                        indices = tf.stack([idx, batch_range], axis=1)
                        return tf.gather_nd(stacked_out, indices)
                def adapt_false():
                    return tf.reduce_mean(stacked_out, axis=0)
                return tf.cond(adapt_mode_bool, adapt_true, adapt_false)

            def grad(dy): # custom gradient computation
                num_branches = tf.shape(stacked_out)[0]
```

```
52                 def grad_true(): # adapt_mode = True
53                     # pass gradient only to selected sub-layer
54                     batch_size = tf.shape(idx)[0]
55                     batch_range = tf.range(batch_size,
                       ↪  dtype=tf.int32)
56                     indices = tf.stack([idx, batch_range],
                       ↪  axis=1)
57                     zero_grads = tf.zeros_like(stacked_out)
58                     mask =
                       ↪  tf.tensor_scatter_nd_update(zero_grads,
                       ↪  indices, dy)
59                     return mask
60
61                 def grad_false(): # adapt_mode = False
62                     # pass gradient to all sub-layers
63                     return tf.tile(tf.expand_dims(dy, 0),
                       ↪  [num_branches] + [1] * len(dy.shape))
64
65                 return grad_true if adapt_mode else grad_false,
                   ↪  None, None
66
67             return forward(), grad
68
69         return choose_and_route(stacked, chosen_idx,
           ↪  self.adapt_mode)
```

The entire source code of this project can be accessed at: `https://github.com/hel-har/master_thesis`

Appendix B

Experiment Data

Table B.1: Results of the adjusted multi-anchor epoch training implementation performed using *cycle* device selection on the *RealWorld* dataset. Models were trained for adaptability.

		cycle multi-anchor epoch	
	F1-scores	**Single ft**	**Multi ft**
evaluation devices	**head**	0.720	0.576
	chest	0.677	0.670
	waist	0.905	0.727
	forearm	0.746	0.583
	upper arm	0.792	0.797
	shin	0.635	0.691
	thigh	0.794	0.723
	$\mu_{\mathbf{ad}}$	0.753	0.681
	$\sigma_{\mathbf{ad}}$	0.089	0.080
	ADI	0.549	0.497

Table B.2: Run of the ColloSSL [17] model using self-supervised collaborative training on the *RealWorld* [38] dataset. Models fine-tuned using multi-fine-tuning. Table shows F1-scores of the according models with the lowest validation error. The best results for each recorded measure are **bold**.

		anchor devices							
	F1-scores	**head**	**chest**	**waist**	**forearm**	**upper arm**	**shin**	**thigh**	**Δ single ft**
evaluation devices	**head**	0.628	0.612	0.598	0.629	0.576	0.632	0.614	-0.077
	chest	0.673	0.803	0.755	0.737	0.717	0.799	0.786	0.056
	waist	0.894	0.867	0.865	0.828	0.864	0.749	0.885	-0.021
	forearm	0.622	0.614	0.647	0.621	0.623	0.629	0.607	-0.152
	upper arm	0.786	0.816	0.805	0.790	0.761	0.794	0.751	0.005
	shin	0.727	0.758	0.724	0.724	0.741	0.736	0.722	0.076
	thigh	0.779	0.829	0.847	0.724	0.774	0.775	0.800	-0.007
	$\mu_{\mathbf{ad}}$	0.730	**0.757**	0.749	0.722	0.722	0.731	0.738	
	$\sigma_{\mathbf{ad}}$	0.098	0.104	0.100	0.076	0.097	**0.072**	0.101	
	ADI	0.532	**0.541**	0.534	0.532	0.515	0.540	0.531	

Table B.3: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using **multi** fine-tuning. Runs performed using the parameters defined in Section 4.5.1. Table shows F1-scores of the according models with the lowest validation error. The best results for each experiment are **bold**.

		RealWorld				**PaMap2**		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.488	0.417	0.382	**hand**	0.781	0.658	0.722
	chest	0.574	0.656	0.576	**chest**	0.520	0.481	0.441
	waist	0.700	0.614	0.577	**ankle**	0.640	0.614	0.607
	forearm	0.521	0.491	0.359				
	upper arm	0.636	0.808	0.777				
	shin	0.755	0.734	0.640				
	thigh	0.739	0.749	0.770				
	$\mu_{\mathbf{gen}}$	0.630	**0.638**	0.583		**0.647**	0.584	0.590
	$\sigma_{\mathbf{gen}}$	**0.106**	0.142	0.167		0.131	**0.092**	0.141
	GI	**0.440**	0.416	0.366		**0.453**	0.418	0.399

Table B.4: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using **multi** fine-tuning with the inclusion of a regulatory loss during training of the feature extractor. Runs performed using the parameters defined in Section 4.5.1. Table shows F1-scores of the according models with the lowest validation error. The best results for each experiment are **bold**.

		RealWorld				**PaMap2**		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.665	0.645	0.710	**hand**	0.809	0.810	0.822
	chest	0.632	0.733	0.798	**chest**	0.490	0.467	0.505
	waist	0.868	0.861	0.811	**ankle**	0.597	0.632	0.667
	forearm	0.675	0.738	0.746				
	upper arm	0.769	0.777	0.782				
	shin	0.694	0.644	0.606				
	thigh	0.741	0.738	0.737				
	$\mu_{\mathbf{gen}}$	0.721	0.734	**0.741**		0.632	0.636	**0.665**
	$\sigma_{\mathbf{gen}}$	0.080	0.075	**0.070**		0.162	0.172	**0.159**
	GI	0.534	**0.545**	0.539		0.431	0.424	**0.452**

Table B.5: Run of the ColloSSL [17] model using different *AdaptationLayer* placements on *RealWorld* and *PaMap2*, evaluated using **single** fine-tuning with the inclusion of a regulatory loss during training of the feature extractor. Runs performed using the parameters defined in Section 4.5.1. Table shows F1-scores of the according models with the lowest validation error. The best results for each experiment are **bold**.

		RealWorld				**PaMap2**		
	F1-scores	**a-c-c**	**c-a-c**	**c-c-a**		**a-c-c**	**c-a-c**	**c-c-a**
evaluation devices	**head**	0.715	0.736	0.729	**hand**	0.838	0.784	0.828
	chest	0.701	0.709	0.644	**chest**	0.458	0.521	0.511
	waist	0.771	0.905	0.888	**ankle**	0.661	0.720	0.712
	forearm	0.691	0.681	0.707				
	upper arm	0.802	0.803	0.796				
	shin	0.598	0.591	0.658				
	thigh	0.881	0.868	0.843				
	$\mu_{\mathbf{ad}}$	0.737	**0.756**	0.752		0.652	0.675	**0.684**
	$\sigma_{\mathbf{ad}}$	**0.091**	0.110	0.093		0.190	**0.137**	0.160
	ADI	0.530	0.533	**0.551**		0.426	**0.466**	0.463

Table B.6: Run of the ColloSSL [17] model using different different adapt mode training options *RealWorld* and *PaMap2* with a-a-a architecture. Evaluated using **single** fine-tuning with the inclusion of a regulatory loss during training of the feature extractor. Runs performed using the parameters defined in Section 4.5.1. Table shows F1-scores of the according models with the lowest validation error. The best results for each experiment are **bold**.

		RealWorld				**PaMap2**		
	F1-scores	**cycle**	**divided**	**randomized**		**divided**	**cycle**	**randomized**
evaluation devices	**head**	0.697	0.640	0.711	**hand**	0.804	0.808	0.809
	chest	0.659	0.635	0.660	**chest**	0.494	0.489	0.516
	waist	0.786	0.855	0.848	**ankle**	0.737	0.760	0.739
	forearm	0.766	0.742	0.776				
	upper arm	0.792	0.795	0.807				
	shin	0.596	0.640	0.598				
	thigh	0.798	0.801	0.816				
	$\mu_{\mathbf{gen}}$	0.728	0.730	**0.745**		0.678	0.686	**0.688**
	$\sigma_{\mathbf{gen}}$	**0.078**	0.092	0.092		0.163	0.172	**0.153**
	GI	0.527	**0.537**	0.534		0.455	0.455	**0.468**