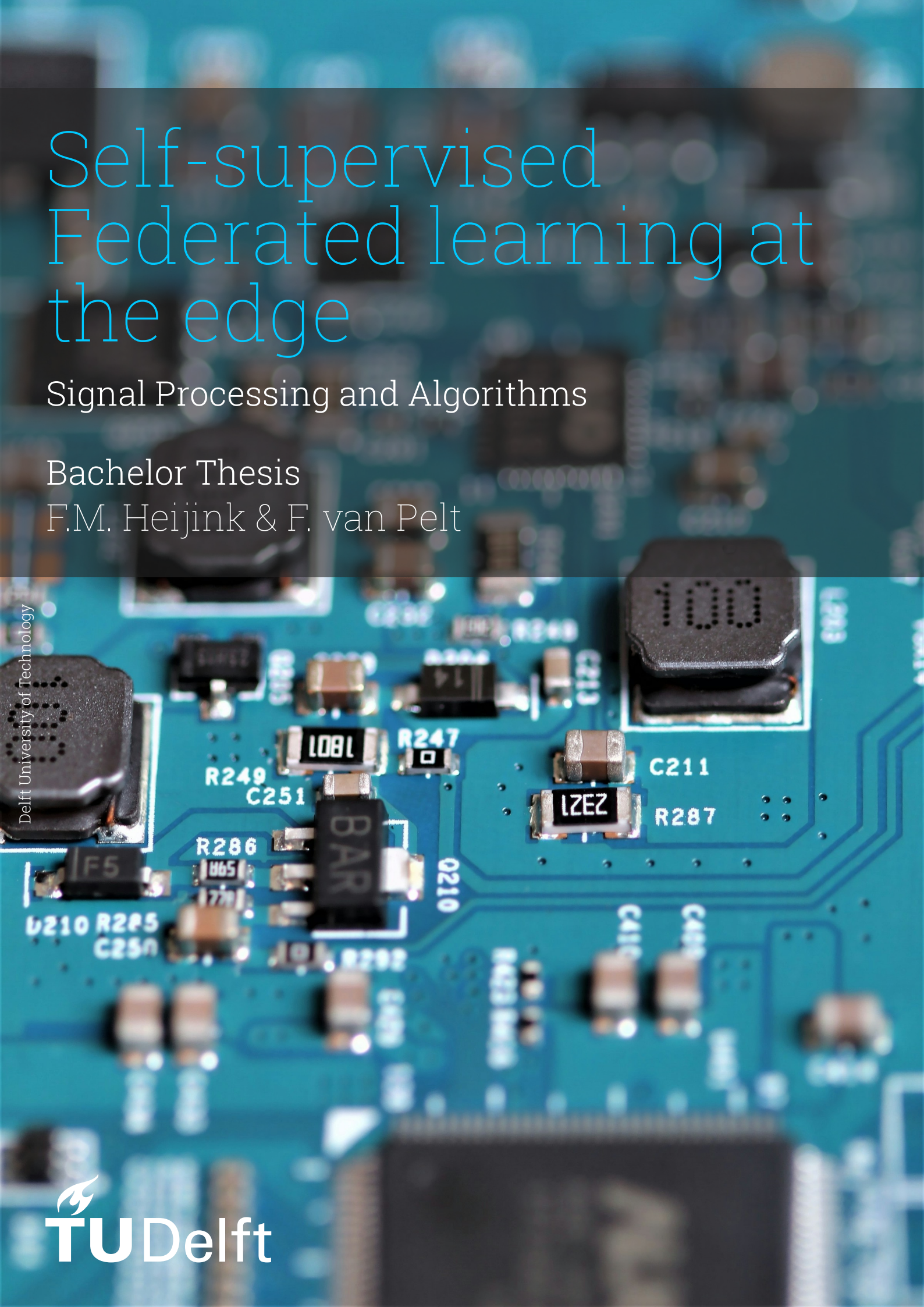
Self-supervised Federated learning at the edge

Signal Processing and Algorithms

Bachelor Thesis

F.M. Heijink & F. van Pelt



Delft University of Technology

TU Delft

Self-supervised Federated learning at the edge

Signal Processing and Algorithms

by

F.M. Heijink & F. van Pelt

Student Name	Student Number
Finn Heijink	5607051
Frank van Pelt	5648378

Supervisors:	Dr. C. Frenkel	TU Delft	EI
	Dr. J.H.G. Dauwels	TU Delft	SPS
Project Assistant:	D. Casnici	TU Delft	EI
Thesis committee:	Dr. C. Frenkel	TU Delft	EI
	Dr. J.H.G. Dauwels	TU Delft	SPS
	Prof. Dr. N. Llombart	TU Delft	THZ
	Dr. L.M. Ramírez-Elizondo	TU Delft	DCES
Thesis defence date:	27th of June, 2024		
Faculty:	EEMCS		
Degree:	BSc Electrical Engineering		

TU Delft

Preface

This report serves to finalize the bachelor graduation project on the topic of self-supervised federated learning, specifically the implementation of the algorithms in Python. The goal of the project is to implement a self-supervised learning setup in a decentralized approach using Field-Programmable Gate Arrays (FPGAs) for the processing of data. This serves as a proof of concept that decentralized machine learning on unlabeled data using FPGAs is possible. Multiple algorithms based on the literature were considered to allow for a low-profile learning setup, with simplifications done to be able to reduce the compute required. The results are promising: scaled-down models that can run on an FPGA show that self-supervised learning functions as expected from the theory. By decentralizing the computations increases in performance are possible in favorable conditions. The authors hope that the concept of self-supervised federated learning can be employed to FPGAs on a larger scale to help in the processing of the abundant yet underutilized unlabeled data present at the edges of information networks.

F.M. Heijink & F. van Pelt
Delft, July 2024

Contents

1 Introduction

In this chapter some fundamental concepts of machine learning classification on unlabeled data will be introduced. Afterwards, the goal of the project itself will be defined and the subgroup division will be explained. Finally, the structure of the thesis will be explained. More explanation of the terms used here and in the remainder of the report can be found in chapter A.

1.1. Introduction to classification

In machine learning, the aim is often to find a model that can give information about new input data based on similar, known data. An example could be the prediction of the outside temperature based on features of the weather, like rainfall and the geographical location. This is done using information about different outside temperature measurements: a relation between the features of the input data and the outside temperature is found. Since the outside temperature can take on an infinite amount of values it is known as a continuous variable, and this form of machine learning is known as regression: the prediction is a continuous value.

Another form of machine learning has to do with the prediction of the label of the input data. An example could be checking whether a given input image contains a dog or a cat. In this case, the prediction is discrete: there is a limited amount of values the prediction can obtain.

Image classification is a form of classification that takes images as input data and performs classification based on this. Since humans rely heavily on their vision to deduce features of their surroundings it is no wonder that artificial intelligence (AI) would evolve to perform the same thing. The applications of image classification are numerous, and with the improvement of imaging- and sensor technology the quality and amount of data is increasing. The structure of image classification is given in figure 1.1.

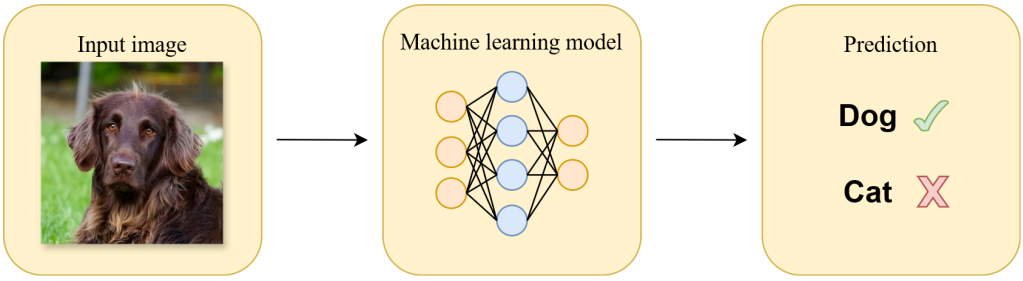


Figure 1.1: General image classification structure. The image is processed by the machine learning network and a prediction is made.

To perform classification, the model must be trained using representative data for the data that needs to be classified. A machine learning model consists of a general structure with parameters that need to be trained: depending on the predictions made, the machine learning model should change these parameters to improve

the accuracy of the predictions. This is known as the training of a machine learning model. A general training approach can look as follows:

1. **Initialize the machine learning model**. Give the trainable parameters an initial value.
2. **Find the prediction based on the input data**. Feed the input data to the machine learning model and find the prediction based on this.
3. **Compare the prediction to the actual label**. This is done by using a cost function. This is a function that quantifies the difference between correct and incorrect predictions, also known as the loss. If the loss increases, the prediction made by the model differs more from the input data.
4. **Compute in what way the parameters should change to reduce the error between the prediction and the label**. This is done by minimizing the loss. To achieve this, most often the derivatives of the cost function with respect to the trainable parameters are calculated to find the effect of changing them on the loss.
5. **Update the parameters such that the error between the prediction and the label decreases**. This step is done using an optimizer that decides in which way to change the weights.

If we apply this approach to figure 1.1, we can implement it as seen in figure 1.2.

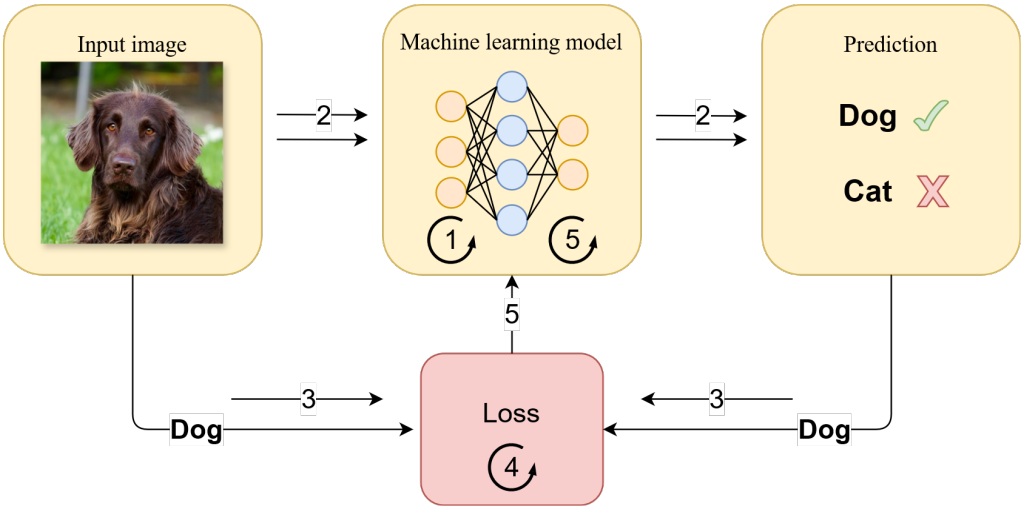


Figure 1.2: Training of an image classifier. The steps are based on the general training approach, with the numbers corresponding to the steps given there.

If we keep repeating steps 2 through 5 using all the data we have, we can improve the machine learning model and increase the accuracy. This is the main idea behind classification, which in this thesis' case is performed on images.

To test the performance of the model, we stop training the model. We use separate representative input data that was not used to train the model itself as the testing set, and compare the predictions generated by the model based on the testing set to the actual labels of the testing set. We find what percentage of the labels of the testing data was predicted correctly to find the accuracy of the machine learning model. If we are satisfied with the accuracy of the model we can use the model to make predictions on different representative data to find labels. This is the main idea behind supervised learning: we use the ground-truth of the data to estimate the performance of our machine learning model.

1.2. Introduction to self-supervised learning

Thus far, we have assumed that supervised learning was possible due to the presence of labels. A very realistic scenario is one in which this is not the case: a very large amount of data can be generated using modern

technologies, way too much to be analyzed and labeled by humans. We thus often use a small amount of the actual data we have since we have a limited amount of data we can actually label. This leaves us with much potentially useful data that can not be used for supervised learning due to the lack of labels. This issue can be tackled by the use self-supervised learning.

Self-supervised learning is a form of unsupervised learning: the input data is unlabeled. Self-supervised learning tries to learn from the data by generating labels itself: the machine learning model in a self-supervised learning setup is tasked with generating representations of the input data that captures the important features of the input data.

The general approach for self-supervised learning is as follows:

1. **Use self-supervised training to train a machine learning model to generate good representations of the input data**. We still follow the same steps as for supervised learning, but now use a new cost function and associated loss that are not reliant on a label of the input data. For this multiple methods exist, and these will be elaborated upon in chapter 3.
2. **Copy this trained model for downstream classification**. If the model is thought to generate sufficiently accurate representations of the input data, the trained model is used for downstream classification: we place another machine learning model, a classifier, behind the trained model.
3. **Use supervised training to train the classifier**. We use a small amount of labeled data to train this classifier, which does not see the data but the output representations of the trained model: the trained model is now fed with labeled data.

The general self-supervised approach is pictured in figure 1.3.

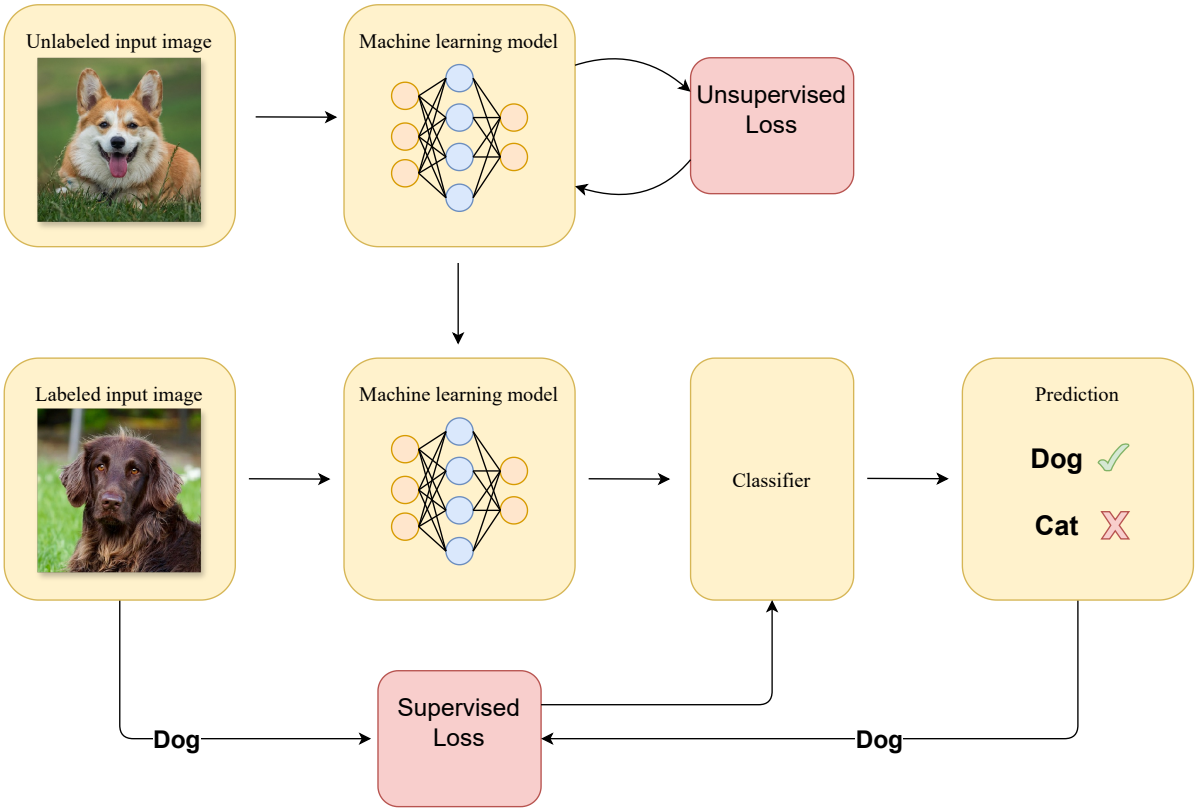


Figure 1.3: General self-supervised learning setup. The self-supervised learning setup is trained using unlabeled data

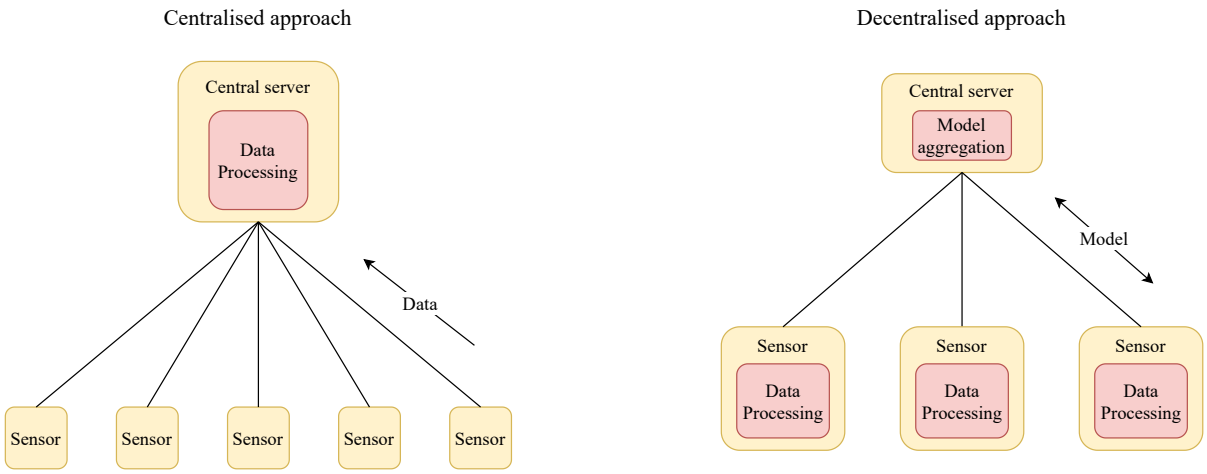


Figure 1.4: Comparison of a centralised decentralised approach in a sensor network.

1.3. Introduction to federated learning

In many cases, many sensors are used at different locations. The effect is that the data is spread throughout a larger network. The centralised approach is to first share the data with a central server that does the data processing, which admits a single point of failure and a large amount of data transfer. In some cases, this data could even be private such that the sharing is prohibited. Federated learning aims to solve this issue.

The main idea of federated learning is to implement the processing of data in a decentralized approach: the data is (partially) processed at the location where it is gathered, such that instead of using a single large server multiple smaller processing units are used [1]. This removes the single point of failure and does not require the sharing of the data itself, but the sharing of (partially) trained machine learning models. Figure 1.4 shows this difference.

To realise this, increased compute is required at the sensors itself. Depending on the application this can result in the constraint that the processing hardware must be more compact and affordable than usual machine learning hardware, like computers with powerful processors (CPUs) and graphics cards (GPUs).

1.4. The goal of the project

Due to the large amount of unlabeled data which potentially can not be shared due to privacy constraints, a decentralized approach to processing unlabeled data is in high demand. There are numerous applications, some of which are listed below:

- Intelligent medical diagnosis systems can be implemented using self-supervised federated learning. Patient confidentiality is challenging to maintain if the data were to be processed. Even when keeping the data private, this typically unlabeled data can be processed by self-supervised federated learning setups [2].
- In modern wireless networks, model-based approaches for system design and analysis are not feasible anymore due to the large sizes of the networks. Instead of relying only on the mathematical models, machine learning models that can learn from the enormous amounts of available data can be used to tackle this problem. Due to the often private nature of the data federated learning can prove to be useful, especially in the case of on-device computation units [3].
- The power grid is becoming increasingly digital and intelligent with the rise of smart grids. This allows for better sensing and sharing of power-related data. A downside of this is the immense amount of data that would need to be shared with central nodes for processing. Multi-party sharing and collaborated learning of the data is done, however this can endanger the private nature of the data. Federated learning can be used to mitigate this risk by processing the data at the grids itself and sharing encrypted model parameters back to a central node. This reduces the necessary bandwidth for sharing this much

data and reduces the risk of leaking private data [4].

By processing unlabeled data directly at the sensor nodes, known as the edges of a network, server-side machine learning models can improve their accuracy and reliability, even in the case of private data which may not be shared between the sensors and the central server. High constraints on the hardware at the sensor sides call for affordable and low-profile hardware that replaces the task classically done at the central server, where constraints on hardware are a lot less stringent.

The goal of the project is to provide a first step in realising this using Field Programmable Gate Arrays (FPGAs). The FPGA would serve as the data processing units that perform self-supervised learning in a federated learning context. A down-scaled implementation of self-supervised learning in conjunction with a federated learning framework will be developed. The FPGAs will perform hardware acceleration to increase the speed performance of the setup. The self-supervised federated learning framework will be created in Python and the hardware design will be done using SystemVerilog. The federated learning setup is aimed to be centered around the self-supervised on-device federated learning (SOFed) algorithm [5]. A readily available FPGA board is used, which contains an on-board microcontroller as well as an FPGA chip.

1.5. Problem definition

To implement a self-supervised federated learning setup, the total group is split up into pairs. Each pair is denoted as a subgroup, and works on one of two parts of the project:

- **Hardware subgroup**. Responsible for the hardware acceleration of the self-supervised learning setup on the FPGA.
- **Signal Processing and Algorithms subgroup**. Responsible for the implementation of the self-supervised learning setup in Python.

This thesis will describe the process and product developed by the Signal Processing and Algorithms group.

1.6. Thesis structure

The thesis has the following structure. In chapter 2 the programme of requirements for the total group and the subgroup will be discussed. In chapter 3 the theoretical background of self-supervised learning and federated learning will be discussed. Chapter 4 discusses a preliminary assessment of the speed bottlenecks in the algorithms that will be used, such that the implementation of the algorithms can take these into account. In chapter 5 the implementation of the self-supervised federated learning framework will be discussed in addition to the testing and results of the implementation. In chapter 6 the delivered product will be reflected upon and recommendations for future work on the topic will be mentioned, and the completion of the requirements will discussed. Chapter 7 gives the conclusion of the thesis.

2

Programme of requirements

In order to delineate the problem statement further, a programme of requirements has been formulated. This list of requirements can be used afterwards to evaluate whether the final result conforms to the initially devised requests and expectations. The programme of requirements has been split up into three parts: a general part, shared by both subgroups, and a further sequence of requirements for each subgroup. Only the requirements relevant for the signal processing subgroup will be stated here.

2.1. General requirements

These requirements make the use of self-supervised learning, federated learning and an FPGA as client explicit.

1. The system should use self-supervised federated learning.
2. The system should be able to perform federated learning on an FPGA on the client side.
3. The system should include a server that coordinates the federated learning among multiple clients. This server does not need to run on an FPGA.
4. The system should be scalable to allow for multiple clients, with 2 clients being the minimum amount.
5. The system's algorithms must be accelerated by using the programmable logic available on the client nodes.

The last requirements is not the task of the signal processing subgroup however, but of the hardware subgroup: it is added for completeness.

2.2. Signal Processing and Algorithms Requirements

The following requirements detail the algorithm that will be implemented, how it will be adjusted to be used in combination with an FPGA and how different network models will be compared. The focus lays firstly on the implementation of the self-supervised and federated learning algorithms, and secondly on the testing and evaluation of the algorithms.

1. The system should implement the basic SOFed setup based on the SOFed paper [5].
2. In this paper, each client has a databuffer containing data for the network to train on. When the client receives new data, this data can be inserted into the databuffer according to its relevance. The system may implement the buffer first in a FIFO (first-in, first-out) manner, with the lazy-scoring buffer detailed in the paper serving as an optional aspect.
3. The system should be optimized and downscaled enough such that it is possible to run the system on the FPGA board (e.g. using simpler machine learning models and algorithms, or using a more coarsely quantized number data format).
4. Several different machine learning models should be tested and compared.

5. The memory usage of the different implementations should be found and compared.
6. The compute requirements (number of multiply-adds and trainable parameters) of the different implementations should be found and compared.
7. The effect of key algorithm parameters should be tested and evaluated.
8. The self-supervised learning should have an effect: the downstream classification accuracy at the server side must be improved by training the model on unlabeled data.

3

Theory

To discuss the relevant background material, a chapter is dedicated to the methods used in the implementation. The concept of self-supervised learning discussed in chapter 1 is built upon with clear examples, followed by the continuation of the concept of federated learning which was also discussed in chapter 1. Different methods will be discussed including pros and cons of each. The goal of this chapter is to help in making the decision of which form of self-supervised learning in conjunction with federated learning to use for the implementation in chapter 5.

3.1. Self-supervised learning

Self-supervised learning is the process of training a machine learning network to classify data without telling the network what the data represents. In this thesis, image classification is considered, and henceforth the data is assumed to consist of images. Given a network $f(x)$, and a constrastive L_2 loss function $\mathcal{L}(a, b) = ||a - b||_2^2$, the network is trained in general by providing either two images x_1 and x_2, called a positive pair, representing the same class, and minimizing the loss $\mathcal{L}(f(x_1), f(x_2))$, or by providing two images x_1 and x_2, called a negative pair, representing different classes, and maximizing the loss $\mathcal{L}(f(x_1), f(x_2))$. The former kind is called non-contrastive learning, and the latter contrastive learning. Using this process, the network f is trained to output similar output vectors for similar input images, and different output vectors for different input images. These are called representations of the input data.

Since the images are unlabeled it is difficult to recognize which images should generate the same representations: the model does not know if two images are part of the same image class. Non-contrastive learning attempts to solve this problem by taking the same input image x, and generating two views $x_1 = U(x)$ and $x_2 = V(x)$ of this image. U and V are image transformations like rotation, blurring and resizing. These transformations change the image, but preserve the semantic information and the class that the image represents. These two views, which by definition represent the same image class, can then be used to minimize $\mathcal{L}(f(x_1), f(x_2))$, thereby training the network f to generate the same representation for the same image class. It is evident that the degree to which this method works is dependent on the quality of the image transformations U and V. This method does not work however for contrastive learning, where labels are needed to determine whether two images indeed do not represent the same image class. To circumvent this, negative pairs are randomly chosen by picking two different images. However, this means that there is still a chance that a negative pair represents the same class, since those two images could belong to the same class.

3.1.1. Different methods & models

Over the past years, several different variations of this basic concept have been devised. This section will go through the most important of these.

BYOL

The Bootstrap Your Own Latent (BYOL) [6] approach is visualized in figure 3.1.

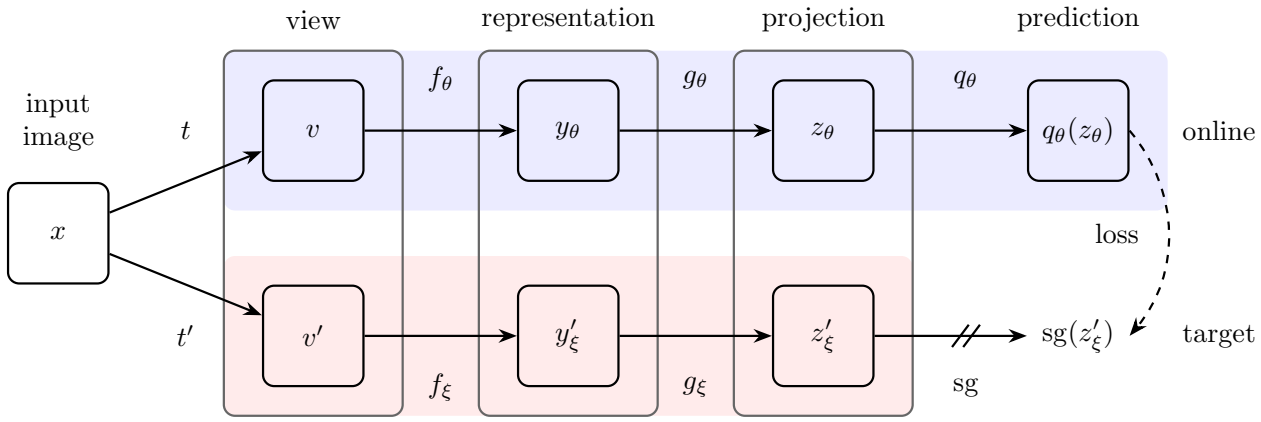


Figure 3.1: A schematic depiction of the BYOL algorithm. [6]

The BYOL algorithm starts with the same input image x, which gets transformed into two views v and v' by a random transformation t. These views are then fed into two separate branches: an online branch and a target branch. The online branch contains the encoder network f_θ that is actively trained, along with a *projector* g_θ and *predictor* q_θ. The target branch also contains a network f_ξ and a projector g_ξ. These target networks are however not actively trained, due to a stop-gradient (sg) element at the end of the target branch. Instead, the target branch is defined as a moving average of the online branch. If w_ξ is the set of weights of the target branch, and w_θ the weights of the online branch, the new set of weights $w_{\xi,new} = \tau w_\xi + (1-\tau)w_\theta$ is calculated after every training iteration. By feeding the two views into the two network branches, the loss function can be defined as follows: $\mathcal{L}(q_\theta(g_\theta(f_\theta(v)), g_\xi(f_\xi(v'))))$. As a loss function, it uses $\mathcal{L}(a,b) = ||\frac{a}{||a||_2} - \frac{b}{||b||_2}||_2^2$. Because BYOL only operates on positive pairs, it is a form of non-contrastive learning.

The encoder network is a convolutional neural network (CNN) generally consisting of a sequence of convolutional layers, batch normalization layers, and activation layers. The functioning of a CNN and batch normalization layers is explained in appendix A.

It should be noted that, along with the encoder network f, two other networks are included: a projector and a predictor. The projector, present in both the online and target branch, is a Multi-Layer Perceptron (shortened as MLP, a multi-layer neural network consisting of $n \geq 2$ linear layers and an activated layer), reducing the dimension of the output of the network f. The predictor is an MLP as well, which tries to map the output of the online branch to the output of the target branch.

Three counter-measures taken to prevent collapse are taking the exponential moving average (EMA) of the online branch for the target branch, the projector, and the predictor. Theoretically, in non-contrastive learning, it would be possible for the network f to always output a zero vector; an optimal loss $\mathcal{L}(f(x_1), f(x_2)) = \mathcal{L}(0,0) = 0$ would then be achieved. However, this is undesirable: the network has collapsed, and the output of the does network f does not contain a representation of the input image class. Every image would generate the same representation. The EMA, projector, predictor, and stop gradient all help to prevent this collapse. An (empirical) argument for the need of these elements is given in the next section.

A brief explanation of the need for a projector is given in the 'SSL Cookbook' [7]. Although not strictly necessary for preventing collapse, it can give ~20% accuracy gains under special conditions. Furthermore, it is postulated that it can help with noisy augmentations such that a positive pair whose two views are quite different can still be projected onto the same representation.

SimSiam

SimSiam [8] is a modification of BYOL. It removes the EMA and sets the target network equal to the online network at all times. This showed that the EMA was not needed to prevent collapse. As a loss function, it uses negative cosine similarity: $\mathcal{L}(a,b) = -\frac{a}{||a||_2} \cdot \frac{b}{||b||_2}$. Note that SimSiam in fact removes the distinction between

the online and target branch. However, these names will still be used to designate the two branches.

Because the networks in the two branches of SimSiam are equal, SimSiam requires less memory than BYOL, since the weights of the two branches are shared. It can also require less computation, since feeding an image through the network immediately computes both the online and target result of that image. The predictor output can then be calculated to find the output of the online branch using the output of the target branch.

One of the purposes of the SimSiam paper is to show that EMA is not strictly necessary to prevent collapse. SimSiam does achieve a slightly lower final accuracy however, although it converges faster in the beginning [8].

The paper also performs an empirical study to show what effect the other network parts have. It showed that removing the stop gradient causes a quick collapse [8]. Hence, the stop gradient seems instrumental to prevent collapse. Removing the batchnorm layers in the network did not cause collapse, but it did cause a great loss of accuracy. Hence, batch normalization is instrumental to achieve good accuracy. Removing the predictor also caused collapse; hence the predictor is also a necessary element.

SimCLR

SimCLR [9] is a predecessor to BYOL. Like SimSiam, it does not use an EMA encoder but the same network for both branches. It does use negative pairs: when training the network using one minibatch at a time, the two augmentations in the two branches from the same image in the minibatch count as a positive pair, but the same image with respect to every other image in the minibatch count as negative pairs. This removes collapse, considering that besides the minimisation of one loss, the maximization of another type of loss is needed: the representations for the negative pair should differ as much as possible, while the representations of the positive pair should be as close as possible like before.

The paper shows that contrastive learning benefits from large batch sizes and longer training. It also shows that it is able to achieve accuracies only slightly lower (a few %) than equivalent supervised learning on the ImageNet dataset [10]. Note that BYOL and SimSiam generally achieve better results than SimCLR [6] [8], although the risk of collapse exists for those algorithms.

BarlowTwins

BarlowTwins [11] is a successor to SimSiam and BYOL. It has a symmetrical architecture, with the online and target branch being the same, containing an encoder and a projector. BarlowTwins does not contain a predictor, nor a stop gradient. In order to prevent collapse without a predictor or a stop gradient, BarlowTwins defines its loss function as trying to get the cross-correlation of the output of the two branches as close as possible to an unity matrix. Given the two projections z_θ and z_ξ:

$$C_{ij} = \frac{\sum_b z_{\theta,b,i} z_{\xi,b,j}}{\sqrt{\sum_b (z^2_{\theta,b,i})} \sqrt{\sum_b (z^2_{\xi,b,j})}}, \tag{3.1}$$

it defines the loss function $\mathcal{L}(z_\theta, z_\xi) = \sum_i (1 - C_{ii})^2 + \lambda \sum_i \sum_{j \neq i} C^2_{ij}$. With this loss function, $\mathcal{L}(\mathbf{0}, \mathbf{0}) \neq 0$ is not an optimal solution anymore, hence collapse is prevented inherently.

On ImageNet, it achieves an accuracy slightly below BYOL, and slightly above SimSiam. It seems robust to small batch sizes, like BYOL. BarlowTwins seems to profit from a large projector output size.

3.1.2. Relevant methods

Having explained the most popular SSL methods, a summary of the characteristics of each can be seen in table 3.1. In order to choose which SSL method to employ, several considerations could be made. For example, [12] details the considerations in choosing between contrastive and non-contrastive learning. But given the scope of this project, implementation simplicity and computational complexity are key considerations due to time and hardware constraints. Non-contrastive learning is easier to implement since only positive pairs need to be considered, simplifying the loss function. Both BarlowTwins and SimSiam have lower memory needs since both SSL branches are equal, while there is a distinction in BYOL between the online and target

branch, hence both need to be stored in memory. Between BarlowTwins and SimSiam, SimSiam is easier to implement since, again, it has a simpler loss function. BYOL is similar in this regard, but additionally does not collapse as quickly and can achieve slightly higher accuracies, albeit at the cost of higher memory usage. In lieue of these considerations, SimSiam seems the most promising SSL method, with BYOL serving as a baseline. Since SimSiam is akin to BYOL but without EMA, it is trivial implementation-wise to change BYOL to SimSiam. Thus, BYOL and SimSiam will be the two SSL methods that will be considered from now on, and their performance will be compared and discussed in chapter 5.

Table 3.1: An overview of each of the SSL methods.

Name	Contrastive	EMA	Projector	Predictor	Liable to collapse
BYOL	No	Yes	Yes	Yes	Yes
SimSiam	No	No	Yes	Yes	Yes
SimCLR	Yes	No	Yes	No	Yes
BarlowTwins	No	No	Yes	No	No

3.1.3. Classification

After training the model in a self-supervised manner, it can be converted into a classification model by taking only the (online) encoder of the SSL model and attaching it to a linear evaluation head, which consists of a single linear layer. This layer takes the output vector of the encoder and converts it into scores for each of the classes in the dataset. This classifier head can then be trained in a supervised manner, although with much less (labeled) data than the amount of unlabeled data that was available to the encoder using SSL. This method of evaluating a self-supervised model using a linear layer is called linear evaluation, and shows if the encoder generates proper representations of the different image classes when used for testing [13].

3.2. Federated learning

With the discussion of self-supervised learning, the ability to process unlabeled data is explained and explored. This allows for real-time data processing at the edges of a data-gathering network. To further build upon this concept, a decentralized approach is needed such that the data processing at the sensor nodes can be incorporated into a collective machine learning training protocol without the sharing of data. The answer to this is the concept of federated learning.

As mentioned in chapter 1, federated learning is a machine learning approach that relies on multiple clients with data-processing capabilities to train the models, relieving the central server of being solely responsible for this task [1]. In this section, multiple approaches to achieve this are explained and compared.

In general, a decentralized processing network consists of N clients, with client i having acces to a subset D_i of the total data in the network D. In general, the data is not identically and independently distributed (IID): some sensors may generate data that differs tremendously from other sensors. In context: a medical image sensor that tries to detect the presence of diseases in organs would generate more data of damaged organs if a sick patient is imaged. Another sensor in the same network that is imaging a healthy patient will clearly have a different data distribution. This means that D_i and D_{i+1} will not necessarily have the same distribution.

Now take a general machine learning model $M(D)$ that is used at the server for making data predictions on data D. In a decentralized approach, we try to create this model by taking the models of the clients $M_i(D_i)$ which are trained on the local data. We thus have a setup of N clients each with a model $M_i(D_i)$ trained on the subset D_i of the total data D, and we are looking for a way to combine these to generate $M(D)$: the goal of federated learning is to perform the function $f(\cdot)$ such that $M(D) = f(M_1(D_1), M_2(D_2), ..., M_N(D_N))$ predicts the total dataset sufficiently well.

There are two main benefits to this approach when compared to centralized machine learning: the data does not have to be present at the server itself, and the server does not have to train a machine learning model. This is an ideal solution to issues with private or very abundant data: the clients can not (realistically) share their data with the server due to regulations or low communication bandwidths, and the server has no/little access to any data. The server only has to find $M(D)$ by performing $f(\cdot)$.

3.2.1. FederatedAveraging

The initial approach to federated learning was to take the average of the weights of all models, known as FederatedAveraging [1]. The critical assumption for this to work is that $M_i(D_i)$ has the exact same architecture for each of the clients, and that the starting models are all initialized with the same weights. We denote the weights of $M_i(D_i)$ as w_i. The weights w of $M(D)$ can then be found using equation 3.2, which serves as $f(\cdot)$ in the case of this approach.

$$w = \sum_{i=1}^{N} \frac{|D_i|}{\sum_{j=1}^{N} |D_j|} w_i \tag{3.2}$$

Where $|D_i|$ is denoted as the amount of images in $|D_i|$. The emphasis of this approach lies on the unbalanced nature of the data per client: some clients are likely to have way more data available than others. The algorithm takes care of this by taking a weighted average based on the amount of data present at the client: more data means a higher weight and thus a larger influence on the total model update. Note that the data itself does not have to be shared: the client shares the model weights and the amount of images used for training, not the images itself

This model is based on Stochastic Gradient Descent (SGD) as the optimizer. SGD works by updating the weights of the model for every small batch of training images [14]. When compared to normal Gradient Descent (GD), where the update is only done after the entire training dataset, we see a more fluctuating convergence to the minimum: outliers in the data can cause larger changes in the model parameters. While the exact convergence is more complicated due to this, potentially better minima of the loss function can be achieved.

In the case of FederatedAveraging, the data at each client is not necessarily representative of the total data but could be distributed differently. By using SGD on this data, the model is updated more often when compared to GD, meaning that the model can converge quicker but the result is less stable. This is ideal for federated learning: each client can converge quicker to the representative data but with more variance of the datasets taken into account, and the averaging of the weights that takes place at the server side helps resolve these issues. FederatedAveraging adds to this concept by letting the clients train their data for multiple epochs before sending the model to the server.

In summary, the training loop looks as follows.

1. **Server**: initialize central model $M_{t-1}(D)$ at timestep $t-1$.
2. **Server**: send central model to clients.
3. **Clients**: perform training using SGD for multiple epochs on the received model using the local data D_i.
4. **Clients**: send $M_{t-1,i}(D_i)$ and the amount of data used $|D_i|$ to the server.
5. **Server**: finds $M_t(D) = f(M_{t-1,1}(D_1), M_{t-1,2}(D_2), ..., M_{t-1,N}(D_N))$ as described in equation 3.2. The loop will continue from 2.

A strong point of this approach is the little amount of communication rounds needed: instead of updating the central model after every local model update, more training is done per communication round. This does not only decrease the amount of communication rounds needed for convergence, but also outperforms the approach of updating the central model after every single training epoch of the clients [1].

What this method does not address is the limited amount of storage at the clients: the datasets D_i can not be too large due to this. Since there is typically a lot of data being generated, part of this data must be dropped to allow for realistic training-data storage. If we were to just drop the data in the context of non-IID data, catastrophic forgetting could occur. This leads to the incorrect classification of images that have not been seen for a certain amount of time, which is realistic in the case of non-IID data.

3.2.2. SOFed

To account for the limited amount of storage at the clients, data must be dropped. Section 3.2.1 detailed that dropping data without any consideration of the content itself, catastrophic forgetting could occur in the case of

non-IID data. Self-supervised On-device Federated learning (SOFed) aims to solve this, in addition to omitting the use of labeled data by focusing specifically on the use of self-supervised learning for the training of the local models [5].

In SOFed the data will be stored in a data buffer that can be implemented on a device. This buffer serves to hold the most important data for model training, where importance is defined using a score. This score indicates if the self-supervised machine learning model has difficulty creating representations based on this data, which can be formulated as the SSL loss for the input image, as described in section 3.1. When new data arrives, the data entries with the highest scores are chosen to fill the buffer. This way, we try to keep the data that is the most useful to train on since the model has not yet managed to process the data sufficiently. The batch size during training is set equal to the buffer size, and the buffer is updated as soon as data comes in.

With the current setup, every time the weights are updated the score of an image in the buffer becomes invalid. Since typical updates do not change the loss of a single image dramatically in typical systems, a lazy scoring approach is used. By only calculating the score after a certain amount of weight updates, less computations are needed with minimal changes in accuracy [5]. This is done by giving a timer to each image indicating the amount of epochs since the last scoring. If this timer exceeds a chosen amount of epochs, it is updated to better represent the score of the image for the current model.

SOFed also makes use of equation 3.2 to find the new weights, however now the weighted average is dependent on the buffer size (which is typically equal for every client, in which case the averaging becomes uniform). The training approach is equal to that of FederatedAveraging, besides the lazy scoring of images which takes place after every weight update of every client.

Figure B.2 in appendix B shows the approach used by SOFed.

4

Identification of bottlenecks

This chapter will highlight the bottlenecks of the system by the profiling of a formative subsystem.

Before simplifications can be made, the bottlenecks of the system must be analyzed. This is done by profiling the system: the percentage of the runtime of each section of the subsystem for an epoch of training is done. For more information on how this is achieved the reader is referred to the thesis of the hardware subgroup.

The tests are performed on BYOL with an encoder that contains of 2 convolutional layers with 16 channels in the first layer and 32 channels in the second layer. The hyperparameters of the system and other models used for the setup are discussed in section 5.1.11 and section D.1. The results are given in figure 4.1.

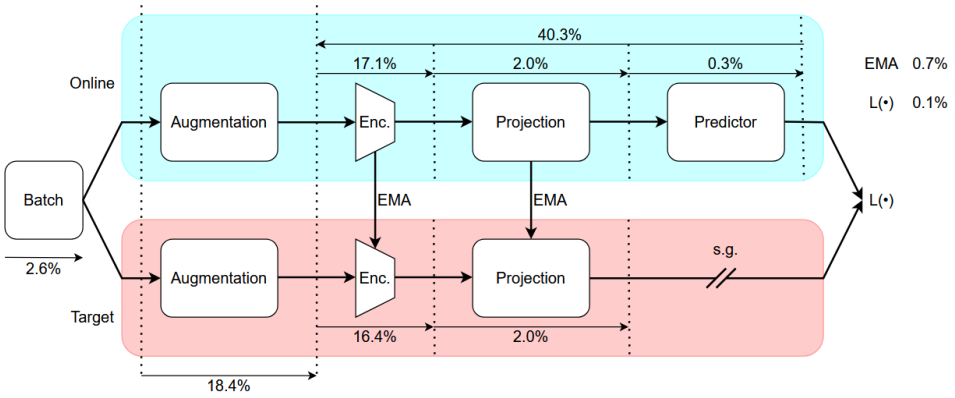


Figure 4.1: Results of the profiling done by the Hardware subgroup on an early implementation of BYOL. The percentages indicate what part of the runtime is needed for that part of the setup.

We note that the three most taxing elements of the system are the backpropagation, the forward pass of the system (calculation of the output of the system for a given input image), and the augmentations of the images. The focus will thus lie on simplifying the models by accelerating these in the hardware. Due to the scope and time-frame of the project the backpropagation algorithm in hardware was not deemed to be feasible. The hardware subgroup goes into more detail on this topic, with the conclusion being that the augmentations and the forward pass of the encoder will be accelerated in hardware. This will be taken into consideration on the software side of the project too by taking simplified encoder structures and researching alternatives that require less computations (e.g. SimSiam). The results of this will be explained in chapter 5.

5

Implementation

This chapter firstly gives an overview of the implementation of the self-supervised learning algorithms by outlining the high-level algorithms and the network models used. These algorithms and network models have been implemented in Python using PyTorch [15]. All code is included in appendix E, and some implementation details are given in appendix D. Afterwards, the interaction with the hardware/FPGA-side of this project is detailed and the simplifications that had to be performed. These were based on the profiling of the system mentioned in chapter 4. Finally, the test results of the then complete SSL setup will be given. Different SSL methods and network models will be compared, and an ablation study will be performed to test the effect of some of the hyperparameters on evaluation accuracy.

The second part of this chapter discusses the federated learning setup in the same structure: a detailed explanation of the algorithms used, followed by the test results.

5.1. SSL Implementation

The implementation of the SSL implementation will be discussed part-by-part using the heartbeat of the program: the training loop, as seen in pseudocode in algorithm 1. The network is trained on a dataset which is split up into small batches of images and looped over in N epochs. For every batch, the images are augmented into two different views of the same input images. These input images are fed into the model, which returns a loss. Backpropagation (see appendix A) is performed on this loss, thereby minimizing it. After every batch, the online and target network are updated in the case of the BYOL model. Firstly, the 'model' part will be discussed, then the dataset ('loadBatch'), the image augmentations ('augment'), the optimizer ('backward'), the EMA ('updateEMA'), the changing of τ and the learning rate over time ('updateSchedule'), and finally how the trained model can be used to perform classification tasks and how the accuracy of a trained model is evaluated.

Algorithm 1 Main Loop

```
n ← 0
while n < N_epochs do
    startEpoch()
    while dataLeft() do
        x ←loadBatch()
        x_1, x_2 ←augment(x)
        L ←model(x_1, x_2)
        backward(L)
        updateEMA()
        updateSchedule()
    end while
    n ← n + 1
end while
```

5.1.1. Models

As discussed in chapter 3, two main SSL methods will be discussed: BYOL and SimSiam. Firstly, the BYOL algorithm is considered, then its constituent parts: the encoder, projector and predictor, and finally the SimSiam algorithm (which utilises the same encoder, projector and predictor, but in a slightly different manner).

BYOL

The implementation of the BYOL algorithm is given in pseudocode in algorithm 2. Both images views are fed into both the online and target branch, and the loss is computed both between the online result of view 1 and the target result of view 2, and the online result of view 2 and the target result of view 1. This effectively trains the network twice, without having to recompute augmentations.
In the algorithm, it can be seen that the BYOL makes use of several other submodels. The implementation of each of the subparts of the network (the encoder f, the projector g and the predictor q) will now be detailed.

Algorithm 2 BYOL

Input
- x_1 First augmented image view
- x_2 Second augmented image view

Output
- $\mathcal{L}$ Loss

$z_{\theta,1} \leftarrow g_\theta(f_\theta(x_1))$
$q_{\theta,1} \leftarrow q_\theta(z_{\theta,1})$
$z_{\xi,1} \leftarrow sg(g_\xi(f_\xi(x_1)))$

$z_{\theta,2} \leftarrow g_\theta(f_\theta(x_2))$
$q_{\theta,2} \leftarrow q_\theta(z_{\theta,2})$
$z_{\xi,2} \leftarrow sg(g_\xi(f_\xi(x_2)))$

$\mathcal{L} \leftarrow \frac{1}{2}\left|\left|\frac{q_{\theta,1}}{||q_{\theta,1}||_2} - \frac{z_{\xi,2}}{||z_{\xi,2}||_2}\right|\right|_2^2 + \frac{1}{2}\left|\left|\frac{q_{\theta,2}}{||q_{\theta,2}||_2} - \frac{z_{\xi,1}}{||z_{\xi,1}||_2}\right|\right|_2^2$

Encoder

The encoder is a Convolutional Neural Network (CNN). Several CNN implementations have been considered: the BYOL paper [6] itself uses different ResNet models [16] for the encoder; MobileNet [17], and its different versions, are intended to be used in mobile (hence low-power) vision applications. These models still have upwards of 2M+ parameters [17]. Due to the hardware constraints the decision was made to stick to CNN models with less than 50K parameters. In the end, three different CNN models were used: a 'simple', 'medium' and an 'advanced' network. These can be seen in figures B.1a, B.1b & B.1c in appendix B. Note, that each BatchNorm layer also includes a Rectified Linear Unit (ReLU) activation at the end. Batchnorm is included, which is very helpful for the network to converge and achieve decent accuracies [18] [6]. The configuration of the convolutional layers can be seen in table 5.1. These configurations have differing amounts of layers, with channel counts gradually increasing, as is commonly seen in literature [17]. The size of each encoder in terms of parameters and its computational cost can be found in table B.1 in appendix B.

Table 5.1: Configurations of the three different encoder types.

Encoder type	Input channel #	Output channel #'s				
Simple	1	2	4			
Medium	1	4	8	12		
Advanced	1	6	12	18	24	30

Projector

The projector is an MLP [6]. All MLP's used and mentioned in this thesis are regular two-layered networks, consisting of, in sequence: a linear layer (the hidden layer), batch normalization, ReLU activation, and at the end another linear layer (the output layer).

The input of the projector is the image output of the encoder, flattened to a singled one dimensional vector. The hidden layer size and the output size are hyperparameters, whose values are stated in section 5.1.11.

Predictor

The predictor is an MLP as well [6]. The output of the online projector is fed into the predictor, and the output of the predictor has the same size as its input considering that it has to be compared (in the loss function) against the output of the target projector. The hidden layer size of the MLP is a hyperparameter stated in section 5.1.11.

SimSiam

The implementation of the SimSiam algorithm can be seen in algorithm 3. Again, both images views are fed into both the online and target branch, and the loss is computed both between the online result of view 1 and the target result of view 2, and the online result of view 2 and the target result of view 1. Since the target branch is equal to the online branch, the encoded and projected result of the one branch can be reused for the other: $z_{\theta,1} = z_{\theta,2}$. The same encoder ($f$), projector ($g$) and predictor ($q$) networks are used as with BYOL.

Algorithm 3 SimSiam

Input

x_1 First augmented image view

x_2 Second augmented image view

Output

$\mathcal{L}$ Loss

$z_{\theta,1} \leftarrow g_\theta(f_\theta(x_1))$
$q_{\theta,1} \leftarrow q_\theta(z_{\theta,1})$
$z_{\xi,1} \leftarrow sg(z_{\theta,1})$

$z_{\theta,2} \leftarrow g_\theta(f_\theta(x_2))$
$q_{\theta,2} \leftarrow q_\theta(z_{\theta,2})$
$z_{\xi,2} \leftarrow sg(z_{\theta,2})$

$\mathcal{L} \leftarrow \frac{1}{2}\left|\left|\frac{q_{\theta,1}}{||q_{\theta,1}||_2} - \frac{z_{\xi,2}}{||z_{\xi,2}||_2}\right|\right|_2^2 + \frac{1}{2}\left|\left|\frac{q_{\theta,2}}{||q_{\theta,2}||_2} - \frac{z_{\xi,1}}{||z_{\xi,1}||_2}\right|\right|_2^2$

5.1.2. Dataset

In order to choose an adequate dataset of images to test the SSL framework on it must adhere to the following criteria:

1. It must be sufficiently difficult to train on such that a simple linear layer without a CNN cannot already achieve high accuracy.
2. It must not be too difficult: the encoders defined above must be able to achieve sufficiently high accuracy such that the setup can find proper representations of the data.
3. The format of the images in the dataset must not be too big: the images must fit in the storage on the FPGA board.

Three common datasets used for testing are MNIST [19], CIFAR-10 [20] and ImageNet [10]. These datasets vary wildly in difficulty, and also in the format of the images. The buffer storing the images mentioned in chapter 3.2.2 should be big enough for the algorithms to function properly without taking too much space in the memory or on the chip, hence the third point in the list

ImageNet has coloured images that have an average resolution of around 256x256 pixels after cropping [10], CIFAR-10 has coloured images that are 32x32 pixels, while MNIST has gray-scale images that are 28x28 pixels. Examples can be found in figure 5.1. Since a coloured image requires three channels to represent the RGB values, MNIST would require less than a third of the amount of storage when compared to CIFAR-10, and almost 250 times less storage than an ImageNet image. For this reason, MNIST is chosen as a preliminary dataset. The main issue with MNIST lies in the ease of classification: the accuracy typically lies well

above 90% even for simple models [19]. Alternatives that are harder to classify exist: FashionMNIST [21] and KMNIST [22] have the exact same image format (size, resolution, and number of colours) while typically being harder to classify. For this reason a test was done to find baseline accuracies for each of these sets to find which is most difficult to classify. The results can be found in chapter 5.1.12, table 5.2, which show that KMNIST is the best fit for the needs of the project. KMNIST will thus be used as the dataset for training and testing.

(a) An image in the MNIST dataset.

(b) An image in the KMNIST dataset.

(c) An image in the CIFAR-10 dataset.

(d) An image in the ImageNet dataset.

Figure 5.1: Examples of images in the different datasets that were considered.

5.1.3. Augmentations

Three different image augmentations will be considered: Gaussian blur, rotation, and resized crop. The effect of these three image augmentations can be seen in figure 5.2. A Gaussian blur and resized crop stem from the BYOL paper [6], while (slight) rotations are very effective image augmentations that can be replicated relatively well on low-hardware [23]. The BYOL paper also uses color augmentations, but since the KMNIST dataset is in grayscale, these augmentations do not apply. The paper also uses a horizontal flip augmentation, but since a flipped character in the KMNIST dataset does not have the same semantic meaning, it is also not applicable for this dataset: KMNIST consists of Japanese characters, and unbounded rotations and flips result in ambiguity of the letters in extreme cases. Note also that each image is normalized before it is sent to the network, which helps to train the network and improve accuracy [24].

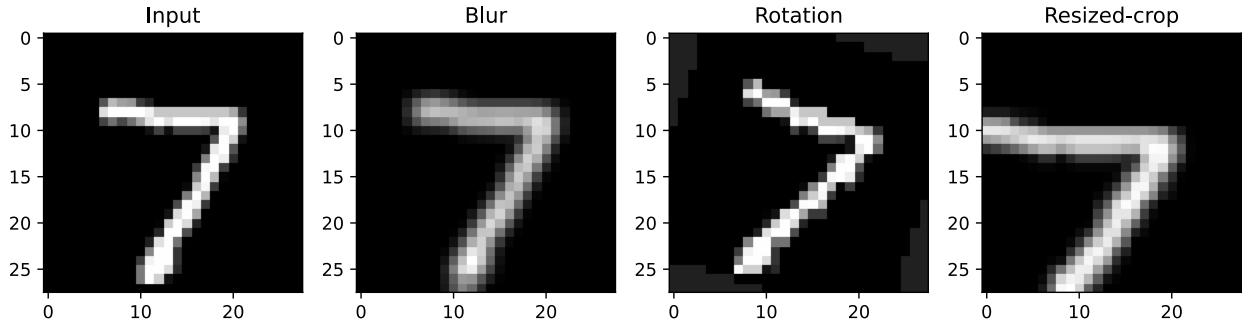


Figure 5.2: A visualization of the different image augmentations that will be considered. Note that this is an image from the MNIST dataset, not from the KMNIST dataset.

Three different ways of augmenting images will be considered:

1. Augmenting the input twice to create two different, augmentated views: $x_1 = U(x), x_2 = U(x)$.
2. Augmenting the input once for one view, and using the input directly for the other view: $x_1 = U(x), x_2 = x$.
3. Augmenting the input once, using a weak augment (only the resized crop), and using the input directly for the other view: $x_1 = V(x), x_2 = x$.

While the first method is the most thorough way of creating two different views of the same image, methods two and three are much faster to execute. As a baseline, the first method will be used, and the effect of the

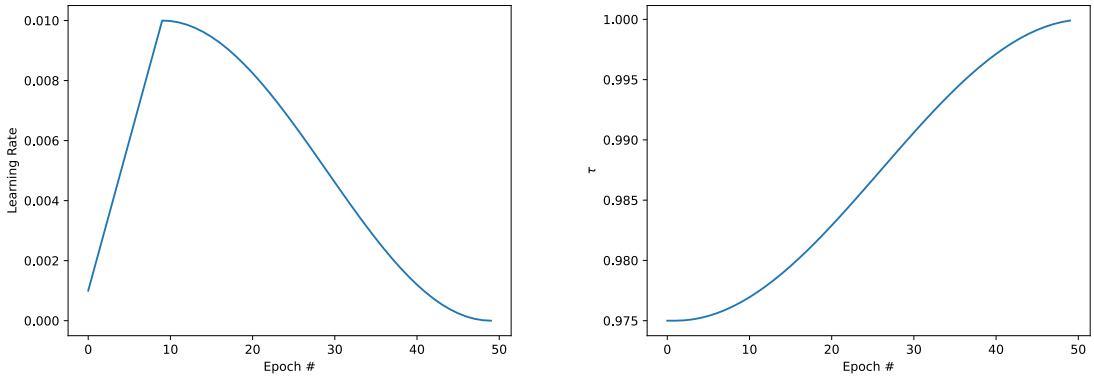
second and third method on evaluation accuracy will be evaluated in the ablation study in section 5.1.12.

5.1.4. Optimizer

The process of backpropagation (see appendix A for more information) mentioned in algorithm 1 as 'backward()' is implemented in PyTorch itself [15]. What remains is choosing an optimizer.
A simple SGD optimizer has been used as a baseline, the reason for which is detailed in section 3.2.1. Its parameters (learning rate, momentum, and weight decay) are stated in section 5.1.11.

5.1.5. Scheduling

The process of scheduling entails changing certain hyperparameters over time, commonly the learning rate. By employing a high, coarse learning rate, the model first finds a rough estimate of the global minimum of the loss function, and by slowly changing this to a lower, finer learning rate, the model is fine-tuned to the global minimum. In this implementation, a linear warm-up with cosine decay schedule is used, in order to modulate the learning rate as seen in figure 5.3a. This is the same schedule as used in the BYOL paper [6].
Another parameter whose value could be changed according to a schedule is the EMA parameter τ. It will be varied according to cosine decay schedule based on the BYOL paper [6]. The scheduler is shown in figure 5.3b.



(a) The linear startup, cosine decay schedule of the learning rate.

(b) The cosine decay schedule of the EMA parameter τ.

Figure 5.3: The two different schedules employed while training.

5.1.6. Exponential Moving Average

As explained in chapter 3, the weights of the target branch of the BYOL network are updated according to equation 5.1:

$$w_{\xi,new} = \tau w_\xi + (1-\tau) w_\theta. \tag{5.1}$$

Here w_ξ are the weights of the target network (belonging to the encoder and projector), and w_θ are the weights of the online network (belonging to the encoder and projector). The predictor is not taken into account here since the target network contains no predictor. τ is a hyperparameter controlling how fast the target network is updated using the weights of the online network, whose value is stated in section 5.1.11. Note that this process of updating the target branch using an EMA of the online network is not performed when using SimSiam since SimSiam does not have a distinct online and target branch. Note that when setting $\tau = 0$, the target and online branch are set equal to each other as well: SimSiam is the same as BYOL with $\tau = 0$.

5.1.7. Classification

After having trained the BYOL network (algorithm 2) or the SimSiam network (algorithm 3), the process of which has been detailed in the preceding chapters, the network as of yet only operates on unlabeled data, and hence it is not yet able to correctly guess the label of input images. Allowing the network to perform actual classification is done by training a separate classification network. This classification reuses the encoder part

of the original network, but discards the projector and predictor. After the encoder, a classification head is placed: a single linear layer. The input of this layer is the flattened output of the encoder, and the output size is equal to the number of classes in the dataset (10 for KMNIST). Finally, a LogSoftmax [25] activation is used at the output.

This network is trained in the same manner as in algorithm 1, but with a few changes:

1. The weights of the encoder network stay fixed, to ensure that the encoder has only been trained in a self-supervised manner without using labeled data.
2. There is generally much less labeled data. To simulate this difference between the amount of unlabeled data and the amount of labeled data, the classifier is only trained on a subset of the dataset. The exact percentage of image data used as labeled is a hyperparameter, stated in section 5.1.11, and has been taken as 10%. The other 90% of the dataset is used for self-supervised training: training of the BYOL/SimSiam network and training of the classification network happen on distinct parts of the dataset, there is no data overlap between the two training processes. It must be noted that this is only a simulation of the effects of having a large portion of unlabeled data, since in fact all of the image data in the KMNIST dataset is labeled.

5.1.8. Evaluation

Now that a full network has been obtained fit for classification tasks, the accuracy of the trained network needs to be assessed. The accuracy of a network is obtained by calculating the percentage of images in the dataset it is able to classify correctly. Generally, a separate evaluation part of the dataset, apart from the training part, is used for this evaluation of the network, such that a network that has been able to memorize all data from the training dataset without actually learning to classify the data will not be able to score the 100% on the evaluation dataset. For more information see appendix A.

Note that the classification network does not directly output a label but rather outputs a score for each class. The higher the score, the higher the probability that the input image is of that class. The class with the highest score is taken as the final prediction of the model.

5.1.9. Quantization

This project, as discussed, also contains a hardware side, which aims to perform the forward passes through the CNN on the FPGA in programmable logic in order to speed up the training time. The interface between the hardware and software group has been determined (see section 5.1.10) to be a function that takes the input and weights of one convolutional layer and calculates and returns the convolved output. Furthermore, the hardware part performs it calculations in fixed-point arithmetic (see also section 5.1.10), instead of the floating-point arithmetic that is used in the rest of the system.

This software/signal processing side of the project hence has to accommodate for a mixed signal path: part of the computation done is by the microcontroller unit (MCU) on the FPGA board, and part on the programmable logic. Furthermore, in order to test the software part independently from the hardware part, a drop-in replacement of the hardware forward convolutional function should be made. This replacement function should also emulate the effects of fixed-point arithmetic by quantizing floating-point values to only the values allowed in the fixed-point format, although the calculations themselves may still be performed in floating-point arithmetic.

As activation function in the encoder, ReLU-1 is used: $\text{ReLU}_1(x) = \min(\max(0, x), 1)$, clamping the input between 0 and 1. This simulates the internal 8-bit hardware data format (see section 5.1.10).

5.1.10. Hardware agreements

The previous section already discussed some of the interaction between signal processing subgroup and the hardware subgroup. This section gives an overview of the agreements that were made. The reader is referred to the thesis by the hardware group for information on the hardware acceleration.

1. Forward convolutional computations are hardware-accelerated and calculated in fixed-point, 12-bit format: 1 sign bit, 4 whole bits (before the dot) and 7 fraction bits (after the dot).
2. Other computations, like the linear layer computations of the projector and predictor MLPs are done on

the MCU and are not hardware-accelerated. The backward pass is also not hardware-accelerated.

3. Weights transported to and from hardware are 12 bit fixed-point: 1 sign bit, 4 integer bits, 7 fraction bits.
4. Input and output values of each convolutional section (including batchnorm and activation) are 8 bit: 1 integer bit and 7 fraction bits. Values are thus between 0 and $\sim$2. This effect was intended to be simulated by using ReLU-1 as an activation function. However, the correct activation function would be ReLU-2. This is considered further in the discussion in chapter 6.
5. The interaction mechanism between hardware and software is, at it most simplest, one function that performs the forward pass of a single convolutional layer. If time permits, this can be extended to performing the whole forward encoder pass, including augmentations, in hardware. In the end, this was not achieved. This will be elaborated upon in chapter 6.

5.1.11. Hyperparameters

The hyperparameters of a machine learning network are specific parameters that can be used to tune a machine learning model. They can change how quickly the model will converge or how high the classification accuracy will be. These need to be tuned to allow for the best outcomes.

This can often be a tricky task to achieve, since each combination generally results in a different outcome and the parameters are often not uncorrelated: the effect on the performance of a parameter can change as soon as another parameter changes. We first start by listing all the hyperparameters of the setup and then move on to the selection- and tuning process. For any other hyperparameters that are part of the PyTorch library the default values are used unless otherwise specified

The first set of hyperparameters hold for any chosen SSL method in this setup.

1. **Learning rate**. This hyperparameter influences the amount of change that can occur in a single weight update. Increasing the learning rate results in higher jumps in the values of the weights per optimization step.
2. **Weight decay**. This hyperparameter influences the loss function of the system: it adds a penalty term related to the magnitudes of the weights to the loss function. A higher weight decay forces the system to reduce the magnitude of the weights of the system.
3. **Momentum**. This hyperparameter influences the convergence of the model by taking previous weight updates in account during the next update.
4. **Batch size**. The total training data is split into smaller batches. Changing the size of this batch can influence the convergence and variance of the updates of the weigths [14]. In general, the learning rate should be scaled based on the batch size: higher batch sizes call for a larger learning rate [6].
5. **Classification split**. Datasets often provide a train and a test set for the training and testing of a model respectively. In the case of self-supervised learning, this is not enough. A large train set should be utilized to train the network using a large amount of unlabeled data, while a smaller train set should be used to train the downstream classifier using a small amount of labeled data. Lastly, the test set is used to evaluate the total system. The percentage split between the labeled training data and the unlabeled training data can be varied and is denoted as the classification split.
6. **Size of the hidden layers**. For the projector and predictor networks, a MLP is used. This gives freedom in the number of nodes in the hidden channel of the network. Increasing this channel in size requires more computations but can increase performance. Note: the output of the projector is also a hyperparameter that can be tuned.
7. **For BYOL:** τ. This parameter described in section 5.1.6 is also treated as a hyperparameter. This parameter can have major effects on the convergence of the SSL setup and can help prevent collapse [26].

Optimization methods for hyperparameter tuning exist, which often consist of checking the performance of the model in the case of a selection of parameters, and choosing the setup with the highest accuracy. A setup based on this was created using the SciKit-Optimize library [27]: by approximating the distribution of the hyperparameters as a multivariate normal distribution, an estimation is made of the distribution of the parameters based on the accuracies for different combinations of the parameters. After this a selection is made

for the optimal selection of the hyperparameters in the regard of accuracy. This setup was implemented in Python but due to the stochastic convergence of the models many epochs of training were needed for proper accuracy measurements. Due to the short time-frame of the project, this was not feasible to perform (more on this in chapter 6). For this reason most of the hyperparameters are chosen based on the literature of the chosen methods or on empirical results.

We choose a learning rate equal to $0.05 \cdot \frac{B}{64}$, where B is the batch size. This is based on the SimSiam paper [8] besides a scaling factor: the learning rate was empirically found to perform better when increasing the learning rate found in the papers, which is equal to $0.05 \cdot \frac{B}{256}$.

We also base the weight decay on the SimSiam paper [8], which yields a weight decay of 0.0001. For the momentum, a value of 0.9 is used as stated in the BYOL paper [6].

The batch size is more difficult to find considering that the size and depth of the neural networks used can influence what batch size is optimal. In the SOFed scenario depicted in chapter 3.2.2, the batch size corresponds to the data buffer on the clients: a smaller buffer will be helpful for the hardware implementation considering the decrease in storage required. We start off with a slightly smaller value than mentioned in the SOFed paper [5]: instead of 128, a batch size of 64 is chosen as a starting point.

The classification split is also based on the SOFed paper [5]. Tests are done where either 1% or 10% of the total training data is used as labeled training data for the classifier, where the latter achieved higher accuracies. Considering that we want to show the effect of training the SSL setup on the classification accuracy, we do not want to take the risk of supplying too little data to the classifier such that it bottlenecks. Choosing a lower classification split may allow for a higher increase in performance due to the increase in data used for the SSL setup. This choice is hard to make, but ultimately a classification split of 10% was chosen based on empirical tests: using this split often resulted in enough data for proper convergence of the classifier, which reduces the uncertainty for testing.

The size of the hidden layers of the projector and the predictor were decreased when compared to the BYOL and SimSiam [6] [8] considering that the used dataset is considerably smaller than the dataset used in the papers. In the papers, ImageNet is used a dataset, which consists of more than 5000 different object classes [10]. Considering that KMNIST consists of only 10 object classes and the general model complexity necessary for classifying ImageNet calls for more advanced models, the layers where arbitrarily chosen as 128 nodes for the hidden layers and 32 nodes for the output layer of the predictor.

For the value of τ for BYOL, the value in the paper was chosen [6]. This results in a value of 0.99 for this parameter. This value is of importance for the convergence and should be chosen carefully: some values can result in the collapse of the system as described in chapter 3.1.1. The prevention of collapse was ensured before testing.

5.1.12. Results

We first start with the comparison of two forms of SSL: BYOL and SimSiam. The theory tells us that we expect SimSiam to converge slightly faster, but that BYOL can achieve an overall higher accuracy.

We use the three separate NNs mentioned in chapter 5.1.1 for the encoder in both setups. This thus requires six separate tests to be done, one for each combination of SimSiam or BYOL and an encoder. The total sizes of each of the models are given in table B.2 in appendix B.

We aim to compare the two methods by comparing the accuracy of the classifier in the case of an encoder that is not trained and an encoder that is trained using SSL as described in sections 5.1.7 and 5.1.8. Before we do this however, we find the average classifier accuracy for each dataset.

In general, we train the classifier for 50 epochs using the chosen classification split of 0.1/10%. We use a batch size of 64 and SGD as the optimizer. Since the classifier starts to converge after around 20 epochs, we take the average of the last 30 epochs to estimate the average classifier accuracy for each encoder. We do this for three datasets: MNIST, FashionMNIST, and KMNIST. The accuracy for each type of encoder and

dataset is given in table 5.2.

Table 5.2: Accuracies of the different (randomly initialized) encoders on different MNIST-like datasets.

Classifier accuracy (untrained encoders)			
Encoder name	MNIST dataset	FashionMNIST dataset	KMNIST dataset
Simple encoder	92.2%	79.3%	66.2%
Medium encoder	94.6%	80.1%	74.0%
Advanced encoder	94.6%	77.9%	73.5%

We note that the KMNIST dataset clearly shows the lowest base accuracy. We use the results to justify the choices made in section 5.1.2.

We now move to the testing of the classifier accuracy in the case of trained encoders for both BYOL and SimSiam. We use 50 epochs of self-supervised training and train the classifier for 1 epoch and directly test it afterwards after every epoch of self-supervised training. We plot the results per epoch of this test in figure B.3 in the appendix. The model training was done using the DelftBlue High-performance Cluster [28], using a single NVIDIA Tesla V100S GPU card.

We first note the stochastic convergence: after 30 epochs, accuracy ceases to improve, although there is a lot of variance in the result. We thus calculate the final accuracy using the final 10 epochs and taking the average of these values for every method. We also calculate the relative increase of the accuracy compared to the reference accuracies for each type of encoder. The results are given in table 5.3.

Table 5.3: Accuracies of the different trained encoders in different SSL setups on KMNIST.

Classifier accuracy (trained encoders)			
Encoder type (SSL type)	Reference	Obtained accuracy	Relative increase
Simple encoder (BYOL)	66.2%	71.1%	7.4%
Simple encoder (SimSiam)	66.2%	69.7%	5.3%
Medium encoder (BYOL)	74.0%	80.3%	8.5%
Medium encoder (SimSiam)	74.0%	79.8%	7.8%
Advanced encoder (BYOL)	73.5%	81.9%	11.4%
Advanced encoder (SimSiam)	73.5%	80.4%	9.4%

We see promising results: by choosing an encoder with more channels and layers, the accuracy increases. BYOL typically gives a slightly higher result but also requires more computations to perform. The amount of epochs needed for convergence seems to stay relatively close to that for SimSiam, which could be due to the smaller NNs used for the encoder. The relative increases, defined as the increase of the SSL setup on the classifier trained on an initialized encoder, also increase as the encoder becomes more complex: using complexer encoders help the model in achieving better results.

We now move on to the choice of a single encoder for the FPGA setup. We base this on two factors: the setup should be computationally efficient and the effect of SSL should be clear: there should be a high (relative) increase in accuracy. Considering that the forward pass requires a lot of computations based on the profiling, we would prefer using SimSiam: the encoder and projector forward pass for the online and target branch are equal, heavily reducing the amount of computations necessary for the forward pass. We thus pick the encoder with the largest relative accuracy increase in a SimSiam setup, which results in using the advanced encoder in conjunction with SimSiam. Although the other setups can be used just fine, the chosen setup is picked for the ablation study to show the influence of certain parameters and components.

Ablation study

In the ablation study, the chosen SSL setup will be further analyzed by changing parameters and parts of the system. We will focus on two parts of the system that play a large role for the hardware group and implementation of the system on a device: the batch size and the augmentation of the input images. The batch size

decides the size of the buffer, which should ideally be small. The augmentation of the input images is important for the generation of good representations of the data (see chapter 3.1) but could perhaps be simplified. We research these two critical parts individually.

We start with studying the influence of the batch size. To do this, we do the same tests as before but we now change the batch size (and thus the learning rate too, see chapter 5.1.11) from 16 to 256, doubling the batch size for every step. We use the advanced encoder with SimSiam as mentioned before. We analyze the average final accuracy and relative increase in the same ways. We now also analyze the stability of the final solution: we find the standard deviation of the accuracy in the last 15 epochs of training for every batch size used. The results are plotted in figure B.4 in the appendix, and are also given in the table below.

Table 5.4: Accuracies of the advanced encoder using SimSiam using different batch sizes on KMNIST.

Classifier accuracy (different batch sizes)			
Batch size	Obtained accuracy	Relative increase	Standard deviation
B = 16	81.2%	10.5%	0.21%
B = 32	80.1%	9.0%	0.24%
B = 64	80.4%	9.3%	0.41%
B = 128	82.2%	11.9%	1.3%
B = 256	80.8%	9.9%	1.4%

The results are very interesting: we see that there is not necessarily a clear optimum. When increasing the batch size, we notice an increase in accuracy first but then a decrease in accuracy afterwards. We hypothesize that the increase in batch size increases the capability of high performance of the model by achieving high accuracies but also the standard deviation of the accuracy. There is thus a balance between good accuracies and a low standard deviation. In the ablation study we have done, a batch size of 16 proved to be most viable for an on-device setup considering that we would like to keep the buffer (and thus the batch size) as small as possible. A more optimal batch size could be chosen but this would have effects on the hardware needed for the buffer. The key takeaway is that a higher or lower batch size is not better necessarily. This agrees with the literature [8].

The second ablation study is done by varying the means of image augmentation. In standard SSL schemes, both the online and target branch apply a different augmentation to the same image. We try to analyze the effect of reducing this to only a single augmentation of the input image of the target branch, and even simplifying the augmentation itself. We base the types of augmentations on section 5.1.3. We use the batch size of 16 from the previous ablation test for these tests, and keep the rest of the settings the same as before. The results are plotted in figure B.5 in the appendix. The results are also given in table 5.5.

Table 5.5: Accuracies of the advanced encoder using SimSiam with a batch size of 16 using different augmentations, tested on KMNIST.

Classifier accuracy (different augmentations)		
Type of augmentation	Obtained accuracy	Relative increase
Double augmentation	81.2%	10.5%
Single augmentation	79.6%	8.3%
Single weak augmentation	76.8%	4.5%

The results are as expected: decreasing the amount of augmentations and the quality of the augmentation decreases the obtained accuracy. This trade-off in accuracy could be made to decrease the computations necessary for the augmentations.

5.2. FL Implementation

With SSL being fully explained and tested, federated learning will now be considered. As explained in section 3.2, federated learning (FL) entails training a model on several clients simultaneously, with a server coordinating the process and training a classifier upon the self-supervised models that the clients train. In this project,

the SOFed algorithm will be implemented [5]. This algorithm consists, on the one hand, of FederatedAveraging (see section 3.2.1) as a method of updating the global model using the local model of each client, and a special data buffer on the other hand which does not contain the full dataset, but only relevant portions. By keeping a small data buffer, memory usage is kept small. The implementation of these two parts will be detailed in their respective order.

5.2.1. Communication Protocol

The FederatedAveraging algorithm has been explained in global terms in section 3.2.1. Algorithm 4 shows the implementation of this algorithm im more detailed terms. The algorithm has been adjusted such that a client can enter and leave the training process at any time, which simplifies testing.

Algorithm 4 Federated Averaging: Server

$w_{\text{global}} \leftarrow \text{randomModel}()$
while True do
  while $\sum_{c \in \text{clients}} \text{receivedModel}(c) < \text{len(clients)} \vee \text{len(clients)} = 0$ **do**
    listenForClients()
    for all $c \in$ clients **do**
      updateCommunication(c)
    end for
  end while
  $w_{\text{global}} \leftarrow \frac{1}{\text{len(clients)}} \sum_{c \in \text{clients}} w_c$
  sleep($t_{timeout}$)
  for all $c \in$ clients **do**
    updateCommunication(c)
  end for
  trainClassifier()
  evaluateClassifier()
end while

Algorithm 5 Federated Averaging: Client

startCommunication()
$n \leftarrow 0$
while $n < N_{epochs}$ **do**
  if $n \bmod N_{serversync} = 0$ **then**
    receiveModel()
  end if
  trainModel()
  if $n \bmod N_{bufferupdate} = 0$ **then**
    updateBuffer()
  end if
  $n \leftarrow n + 1$
  if $n \bmod N_{serversync} = 0$ **then**
    sendModel()
  end if
end while
stopCommunication()

The server starts with a randomly initialized model. It runs a main loop, which performs a full averaging process every iteration. At the start of the iteration, the server waits for new clients, and waits until each of these new clients has sent its updated model. Then, it averages the model, waits a while (to give each of the clients time to send a request for the averaged model after they have sent their updated model. For a production environment, this could be made more robust), and sends the new global model. Then, it trains a classifier upon the global model and evaluates the performance.

The client-server communication works using a messaging system: the client sends one of three possible messages:

1. Request for global model. Upon receiving this message, the server sends back the global model.
2. Updated local model available. Immediately after sending this message, the client sends its own local model
3. Client stopped. Upon receiving this message, the server sends an acknowledge message, such that it knows that the server has successfully received the stop message and the communication channel can be closed.

The function updateCommunication() checks if messages have been received from a client, and if so, it processes them.

The function listenForClients() checks whether new clients are available to join the training process, and if so, it appends them to the 'client' list.

The functioning of the client can be seen in pseudocode in algorithm 5. The training starts by connecting the server, after which the training loop is entered. At the first epoch of the training loop the global model is

requested from the server such that the client starts at the most up-to-date version of the global model. Then, the model is trained ('trainModel()'). Note that the model is trained not on a dataset, like in the SSL implementation discussed before, but on a data buffer, whose implementation is discussed in the next section. Every $N_{bufferupdate}$ epochs, this data buffer is updated, allowing new data to enter the buffer. Every $N_{serversync}$ epochs, the trained client model is sent to the server, after which the server averages the model and sends the model back at the start of next client epoch. $N_{bufferupdate}$ and $N_{serversync}$ are two hyperparameters whose value will be discussed in section 5.2.3.

Training, however, is done in quite the same way as detailed in section 3.1, apart from a few differences. For example, the FL client does not contain learning-rate and τ scheduling; this was left out due to time constraints. Furthermore, note that an epoch in the sense taken here is different from its meaning in SSL training. In SSL training, an epoch meant iterating over the whole KMNIST dataset; here it entails iterating over just the data buffer, which contains much less data. Consequently, fully training a model requires many more epochs.

5.2.2. Databuffer

The SOFed algorithm [5] includes a data buffer which keeps a small subset of images instead of a full dataset to conserve memory in low-footprint devices. This data buffer can be implemented in a First-In First-Out (FIFO) manner, where each time a device receives a new image, it is immediately added to the buffer, and the oldest image in the buffer is discarded. However, the SOFed paper details an importance scoring algorithm, where each received image receives a score according to how 'important' the image is to train on. The most important images are then kept in the data buffer. As a measure of importance, the loss of the BYOL/SimSiam network is used when the two branches are fed the same image, only weakly augmented: $v = \mathcal{L} = f(x, V(x))$, where v is the importance score, f the whole network including encoder, projector and predictor as seen in algorithms 2 & 3, and V a weak augmentation. In this implementation, a resized crop augmentation was used as a weak augmentation. The implementation of the data buffer can be seen in algorithm 6, and is used in algorithm 5 as 'updateBuffer()'.

Algorithm 6 Data Buffer

Input
  newImages Newly received images
for all $x \in$ buffer **do**
  timeInBuffer$(x) \leftarrow$ timeInBuffer$(x) + 1$
  if timeInBuffer$(x) \geq N_{rescore}$ **then**
    score$(x) \leftarrow f(x, V(x))$
    timeInBuffer$(x) \leftarrow 0$
  end if
end for
for all $x \in$ newImages **do**
  score$(x) \leftarrow f(x, V(x))$
  timeInBuffer$(x) \leftarrow 0$
  add(buffer, x)
  if len$(\text{buffer}) > N_{buffersize}$ **then**
    remove$(\text{buffer}, \text{lowestScoreItem}(\text{buffer}))$
  end if
end for

First, images in the buffer receive new scores when necessary. New scores are calculated ever $N_{rescore}$ (the lazy-scoring interval, see section 5.2.3 for its value and section 3.2.2 for a more thorough explanation) times this function gets called. It would be possible to calculate new scores every time but this would require much computation time. Since the network does not change too much between iterations, the scores are expected to stay mostly the same, and only a periodic update is required. This saves much computation time. Then, the newly received images and scored. If the score is high enough, they are kept in the buffer and another image is discarded.
The buffer starts out empty, but images are gradually added. When the buffer reaches it maximum size $N_{buffersize}$, an image must be discarded for every new image that is added.

5.2.3. Hyperparameters

Most of the hyperparameters corresponding to FL are equal to those in the SSL setup: please refer to chapter 5.1.11 for these. Besides the values given here, we update the batch size following the ablation study: the batch size used is equal to 16, which also equals the size of the buffer. In addition to this, we use SimSiam for the SSL in conjunction with the advanced encoder. The lazy-scoring interval of the buffer is arbitrarily set to 10 epochs: the score is updated every 10 epochs at the client.

5.2.4. Results

The testing of the federated learning setup is done using two clients as a starting point. To simulate the SSL setup, the clients perform 50 epochs of training on the data buffer every training round. 16 new images are provided to the setup during every epoch of training, of which the client picks a selection using the scoring mechanism. After training and model-sharing by the clients, the server finds the new global model and trains the classifier for 5 epochs. We analyze the accuracy as a function of the number of communication rounds.

A critical detail is that the data distribution is identical for each client: each client receives a random selection of the total data. This is not analyzed in the literature for SOFed [5] considering that it is a simplification of the real scenario, which means an analysis and check of the results will be more difficult.

The results of the experiment are visible in figure B.6 in the appendix. We see a stark increase in accuracy that does not seem to slow down: the results do not give a definite result of the final accuracy considering that it is not reached. Due to the time frame of the project, no further tests were possible to find the final accuracy of the setup.

From the results we can already see that the accuracy presented in chapter 5.1.12 for a batch size of 16 is exceeded: the accuracy almost reaches 85% in the SOFed setup, while the SSL setup only managed to achieve an accuracy of 81.2%. These results thus tell us that the FL setup is able to outperform the SSL setup.

This contradicts the results in the paper: a decrease in accuracy when compared to the SSL setup was achieved [5]. One critical difference is that the data used for the data buffer was not distributed equally for every client: each client had access to data classes in different amounts, which was shown to negatively impact the accuracy if the differences in distribution increase in the ablation tests. We thus hypothesize that the equal distribution of the data for every data buffer helps in achieving higher accuracies than standalone SSL.

As to why higher accuracies may occur, the parallel nature could very much help in this regard. By using multiple clients with different data which we train on extensively, we allow the clients to train the model to create very good representations of the small amount of data in the data buffers. By averaging the results of the encoders of every client, we are essentially averaging the models that are trained on data with the same distribution. This could allow for a decrease in the variance of the model updates performed by the clients. This argument is backed by the very stable convergence of the models: comparing figure B.6 to figure B.3 in the appendix, we see much less large fluctuations in the accuracy, with drops of more than a percent in accuracy being very uncommon in the case of SOFed.

To build upon this result, tests on a higher number of clients must be performed in addition to longer tests on two clients to reveal the final achieved accuracy. The literature tells us that a larger amount of clients is expected to result in a lower accuracy; however, this is in the case of clients with access to different data distributions which does not correspond to the implemented setup. It thus remains to be seen what the effect of increasing the number of clients will be on the performance of the FL setup.

6

Discussion and future work

In this chapter the results of the project are analyzed. The choices made are discussed and compared to the literature. The results are then compared to the programme of requirements. Finally the continuation and recommended future work of the project is discussed.

6.1. Self-supervised learning

The results of the self-supervised learning part of the project were largely positive: the framework yields results which seem to fit the theory. Simplifications had to be done to allow for a decrease in available computational power but it is suspected that due to the use of a simpler dataset this did not influence the comparison of expected results to theory too much.

What could have been improved was the tuning of the setups. It was seen during the ablation study that the optimal batch size was not found before the start of the tests, meaning that the initial tests were not completely representative for the setup in the case of changing the batch size. This could have been solved by tuning the hyperparameters to find the optimal set first. For the purpose of the tuning a Python setup was created, however the time necessary for testing the parameters would have taken too long due to time constraints. (It would have taken in the range of 50 hours per encoder and SSL combination, and could only be done after the SSL setup was finalized.)

Implementing quantized computations proved more difficult than expected. It was agreed upon with the hardware group that the convolutional computations would be done in 12 bit fixed-point precision. However, PyTorch, the Python framework that was used for the implementation, does not allow the usage of fixed-point data in its networks, let alone custom data formats (like the 12 bit format). Hence, the quantization effects that the hardware would encounter had to be simulated in software. Before and after every convolution, the input, weights and output floating-point data had to be quantized to fixed-point values (e.g. 0.246 gets rounded off to 0.25, although still staying a floating-point value). PyTorch did allow using 16-bit floating-point values instead of 32-bit values however, which got made use of, since would still save the microcontroller on the FPGA-board computation time. Furthermore, a mistake slipped in while discussing the quantization formats with the hardware subgroup; at the one hand, ReLU-1 activation was agreed upon to simulate the effects of hardware quantization, but on the other hand the actually 8 bit quantized values have a range of 0-2; ReLU-2 should have been used. Hence, the test results are not completely accurate. The effect of this discrepancy is expected to be minimal however.

6.2. Federated learning

The results of the federated learning setup are not as complete as hoped for. Since the results in the case of two clients are harder to compare to the literature due to the simplifications made it is harder to confirm the functioning of the system. To properly show the functionality of the system, extra tests with more clients and longer tests are needed. This sadly did not fit in the time frame of the project.

6.3. Possible continuation

Due to the time constraints of the project some of the tests and additions were not completed. We offer a list of things that should be analyzed and tested in the case of a continuation of the completed work.

- **Implementation of the complete quantization**. The quantization of the whole system should be implemented such that the complete forward and backward pass of the system can be performed in hardware. This would require quantizing all the weights and the backpropagation results. While attempts were made to implement this, no viable results were generated due to issues in causing stable results.
- **Tuning and further ablation testing**. The setups should be tuned properly using the created hyperparameter optimizer before further testing. After this the ablation testing can be extended: besides the batch size and the augmentations, also looking at changing the hidden layer sizes of the projector and predictor: this could result in accuracy gains without adding too many computations.
- **Further testing of the federated learning setup**. The tests performed were incomplete for a conclusive answer on the effect of scaling the setup to multiple clients. In addition to this, further tests should be performed to research the convergence time and accuracies achievable using the created setup. If the quantization is implemented correctly the results should be representative for the actual setup using FPGAs.
- **Actual implementation in hardware**. Although this is mostly the task of the hardware group, the signal processing and algorithms group could implement a version of the framework that can properly communicate with the hardware acceleration unit on the FPGA. The framework should be adapted to allow for this. Efforts were made in this regard but due to the issues in the hardware group actual integration and testing was sadly not yet possible.
- **Further improvement of the implementation**. Some simplifications like IID data were assumed, which is less realisic when compared to actual sensing scenario's [1]. Considering that the algorithms used are relatively resilient to the use of non-IID data, testing can be done to confirm this. If this proves to be difficult, improvements can be made like replacing the averaging of the model weights at the server side in SOFed and using more complex algorithms currently treated in the literature, which can allow for dynamic data distributions and increased privacy [29].

6.4. Requirements revisited

Most of the requirements have been successfully completed. Some requirements still remain to be completed due to integration issues with the hardware group. To complete all requirements of the Signal Processing and Algorithms subgroup, further testing would be necessary. The reflection on the fulfillment of each requirement can be found in appendix C.

6.5. Problem statement revisited

The problem that this thesis set out to solve was: how can the discrepancy between the amount of labeled and unlabeled data be leveraged to train a network using unlabeled data, such that it performs better than if it were trained purely on labeled data, but without sharing the (private) unlabeled data?
The first part of this problem, training a network using unlabeled data, was solved using self-supervised algorithms, BYOL and SimSiam. The results in section 5.1.12 have shown that training an encoder using both BYOL and SimSiam on unlabeled data allowed for a better downstream classification accuracy.
The second problem, of training the network without sharing the unlabeled data with a central server, has been solved by federated learning. Two clients have been able to separately train a common network in parallel, without needing to share their data, and achieving an accuracy on par with training the SSL model centrally on one single computer. Although running the system with more than two clients has not been tested, there is no reason to expect why this system would not be scalable to more clients. For a conclusive answer, tests must be done.

This shows that both SSL and FL are promising methods for companies who have a lot of unlabeled or highly distributed, privacy sensitive data respectively.

7

Conclusion

The goal of the bachelor graduation project was to implement self-supervised federated learning on a Field-Programmable Gate Array (FPGA), with this report being focused on the implementation of the machine learning framework in Python.

Multiple self-supervised learning setups were considered with two approaches being most suited for the setup: Bootstrap Your Own Latent (BYOL) and Simple Siamese networks (SimSiam). These were implemented in Python and simplified to reduce computations needed for model training. Both approaches were tested for a selection of different compact neural networks to find a solution that balances the effectiveness of the setup and the computations needed for training. The setup was further improved via an ablation study. The end result was a lightweight implementation of SimSiam with quantization.

The self-supervised learning approach was combined with Self-supervised On-device Federated learning (SOFed) to test the performance. A client and server were simulated to test the performance of the setup. In the case of two clients, the results show an increase in performance and a more stable convergence of the algorithm. The results for multiple clients remain to be tested.

The final result is an efficient Python implementation of self-supervised federated learning that simulates the computational environment of the envisioned hardware acceleration that could be implemented on an FPGA.

The implementation of self-supervised federated learning on compact and affordable hardware can improve privacy and processing of unlabeled data which is otherwise difficult to utilize. Data can remain local and private while still being used to improve machine learning models at central locations. This has numerous applications in for example the fields of wireless communications, health, and power grid management.

References

[1] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," *CoRR*, vol. abs/1602.05629, 2016. arXiv: 1602.05629. [Online]. Available: http://arxiv.org/abs/1602.05629.

[2] S. Bharati, M. R. H. Mondal, P. Podder, and V. S. Prasath, "Federated learning: Applications, challenges and future directions," *International Journal of Hybrid Intelligent Systems*, vol. 18, no. 1–2, pp. 19–35, May 2022, ISSN: 1875-8819. DOI: 10.3233/his-220006. [Online]. Available: http://dx.doi.org/10.3233/HIS-220006.

[3] S. Niknam, H. S. Dhillon, and J. H. Reed, *Federated learning for wireless communications: Motivation, opportunities and challenges*, 2020. arXiv: 1908.06847 [eess.SP].

[4] H. Liu, X. Zhang, X. Shen, and H. Sun, *A federated learning framework for smart grids: Securing power traces in collaborative learning*, 2021. arXiv: 2103.11870 [cs.LG].

[5] J. Shi, Y. Wu, D. Zeng, J. Tao, J. Hu, and Y. Shi, "Self-supervised on-device federated learning from unlabeled streams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4871–4882, Dec. 2023, ISSN: 1937-4151. DOI: 10.1109/tcad.2023.3274956. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2023.3274956.

[6] J.-B. Grill, F. Strub, F. Altché, *et al.*, *Bootstrap your own latent: A new approach to self-supervised learning*, 2020. arXiv: 2006.07733 [cs.LG].

[7] R. Balestriero, M. Ibrahim, V. Sobal, *et al.*, "A cookbook of self-supervised learning," 2023.

[8] X. Chen and K. He, *Exploring simple siamese representation learning*, 2020. arXiv: 2011.10566 [cs.CV].

[9] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, *A simple framework for contrastive learning of visual representations*, 2020. arXiv: 2002.05709 [cs.LG].

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[11] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny, *Barlow twins: Self-supervised learning via redundancy reduction*, 2021. arXiv: 2103.03230 [cs.CV].

[12] R. Balestriero and Y. LeCun, *Contrastive and non-contrastive self-supervised learning recover global and local spectral embedding methods*, 2022. arXiv: 2205.11508 [cs.LG].

[13] A. Kolesnikov, X. Zhai, and L. Beyer, *Revisiting self-supervised visual representation learning*, 2019. arXiv: 1901.09005 [cs.CV].

[14] S. Ruder, *An overview of gradient descent optimization algorithms*, 2017. arXiv: 1609.04747 [cs.LG].

[15] A. Paszke, S. Gross, F. Massa, *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG].

[16] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[17] A. G. Howard, M. Zhu, B. Chen, *et al.*, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV].

[18] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167.

[19] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

[20] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *University of Toronto*, May 2012.

[21] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms,” *CoRR*, vol. abs/1708.07747, 2017. arXiv: 1708.07747. [Online]. Available: http://arxiv.org/abs/1708.07747.

[22] T. Clanuwat, M. Bober-Irizar, A. Kitamoto, A. Lamb, K. Yamamoto, and D. Ha, “Deep learning for classical japanese literature,” *CoRR*, vol. abs/1812.01718, 2018. arXiv: 1812.01718. [Online]. Available: http://arxiv.org/abs/1812.01718.

[23] D. D. Thang and T. Matsui, *Image transformation can make neural networks more robust against adversarial examples*, 2019. arXiv: 1901.03037 [cs.CV].

[24] K. Cabello-Solorzano, I. Ortigosa de Araujo, M. Peña, L. Correia, and A. J. Tallón-Ballesteros, “The impact of data normalization on the accuracy of machine learning algorithms: A comparative analysis,” in *18th International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2023)*, P. García Bringas, H. Pérez García, F. J. Martínez de Pisón, *et al.*, Eds., Cham: Springer Nature Switzerland, 2023, pp. 344–353, ISBN: 978-3-031-42536-3.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[26] Y. Tian, X. Chen, and S. Ganguli, *Understanding self-supervised learning dynamics without contrastive pairs*, 2021. arXiv: 2102.06810 [cs.LG].

[27] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, *Scikit-optimize/scikit-optimize*, 2021. DOI: 10.5281/ZENODO.5565057. [Online]. Available: https://zenodo.org/record/5565057.

[28] D. H. P. C. C. (DHPC), *DelftBlue Supercomputer (Phase 2)*, https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[29] B. Liu, N. Lv, Y. Guo, and Y. Li, *Recent advances on federated learning: A systematic survey*, 2023. arXiv: 2301.01299 [cs.LG].

[30] N. Sharma, V. Jain, and A. Mishra, “An analysis of convolutional neural networks for image classification,” *Procedia Computer Science*, vol. 132, pp. 377–384, 2018, International Conference on Computational Intelligence and Data Science, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.05.198. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918309335.

A

Machine learning terminology

In machine learning, lots of terms exist that can be confusing and not entirely self-explanatory. This list is provided to help the reader understand terms used in the report and literature.

- **Epoch**: an epoch is a single training round in which all of the data is used for training. The convergence time of a machine learning model is often denoted in number of epochs instead of computation time.
- **Batch**: a batch is a smaller part of the total training data on which training is done. For every piece of data in a batch, training is done to find the necessary update needed to the weights. Afterward the average value of all these updates is taken. The update to the weights is thus only done once per batch instead of after every image.
- **Gradient descent**: gradient descent is an optimizer that works by updating the weights in the opposite direction of the gradient of the loss function. The maximal decrease in loss is taken using the derivative of the loss function with respect to all the model weights.
- **Training-, test-, and validation sets**: the total data is split up into several categories. The training set is used to train the model. The testing set is used to find how many correct predictions the trained model can make on data it has never seen before. The validation set is a part of the training set which is also tested to find the number of correct predictions: the model has seen this data, thus a higher result is typically expected.
- **Hyperparameters**: the hyperparameters of a machine learning model are parameters that can be tuned to change how the learning is done. These are related to the architecture of the models but also the convergence rate.
- **Overfitting**: the machine learning model is too closely trained on the training data. This results in the noise and peculiarities of the training data becoming part of the model. This is often seen as a very high accuracy on the validation set but a much lower accuracy on the testing set. This can be handled by simplifying the model or tuning the hyperparameters.
- **Underfitting**: the machine learning model can not find a good relation between the data and the result. This is often seen as very low accuracies for both the testing and the training set. This can be handled by complexifying the model or tuning the hyperparameters.
- **Neural network**: a neural network is a special type of machine learning model. It consists of layers of nodes. Each layer receives a number of input values. In an activation layer, each node in this layer obtains a linear combination of the inputs (scaled by the weights and added) and a unique bias. These weights and biases can be updated via training.

  After this summation, the output is scaled by a non-linear function: the activation function. This can be a unit step function, a rectifying function (known as ReLU) or any other non-linear function. The goal of the network is to train each weight such that the activation functions can be used to decipher non-linear relations in the data. Each node has a related activation function, although commonly these are equal for every node in the layer. The outputs of the nodes are then the output of the activation function.

If there is no activation function, the layer is called a linear layer: there is a linear relation between the inputs and the output.

- **Batch normalization layer**: a special type of layer that normalizes the data in a batch by setting the expected value to zero and the variance to 1. These are commonly used since they cause an increase in performance in neural networks [18].
- **Convolutional layer**: a special type of layer that takes a convolution of the previous layer but applies an activation function to the result of each kernel. Each node in the layer corresponds to a single result of a convolution. This reduces the number of weights and has proven to be extremely useful for image processing classification [30].
- **Convolutional neural network**: a neural network that uses convolutional layers. An example of a simple CNN is given in figure A.1. If we denote the activation function as $f(x)$ and the weights of the 3x3 kernel pictured in the figure by $w_0, w_2, ..., w_9$, we can find the following relation: $y_1 = f(w_0 + \sum_{i=1}^{9} w_i x_i)$. The bias in this case is equal to w_0. Each channel has a unique sets of weights per previous input channel, and if the previous layer has multiple channels, next channel takes the sum of the convolutions over each previous layer. Related to the figure, the first convolutional layer sees 3 input layers (the image is in colour, thus a channel for red, green, and blue respectively), thus it needs 3 sets of 10 weights (3x3 kernel weights plus a bias weight) per channel in the first convolutional layer.
- **Backpropagation**. This is an efficient way to find the derivatives of the cost function with respect to the weights by making use of the chain rule. Instead of calculating the derivative of every weight with respect to the input individually, the loss function is differentiated with respect to the weights closest to the output first, followed by differentiating these weights with respect to the weights one step back to the input. This cycle repeats until the input is reached. Using multiplications, the derivative of the cost function with respect to every weight can be found. The algorithm propagates back through the network: the calculations are started close to the output and finish at the input. This is the general way in which the derivative of the loss function with respect to the weights is taken in the case of deeper machine learning models like typical neural networks. After this, the optimizer can be applied.

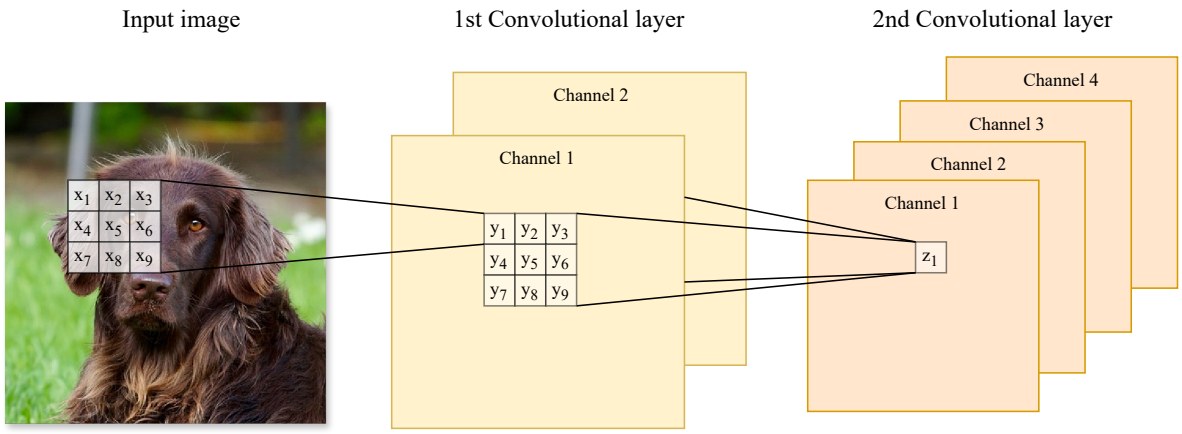


Figure A.1: Example of a 2-layer CNN, with the first layer having 2 channels and the second layer having 4 channels.

A.1. Abbreviations

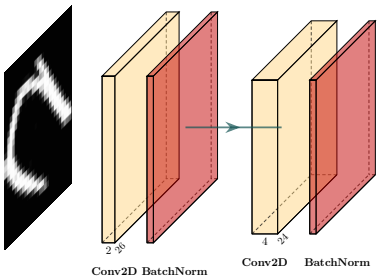
Abbreviation	Definition
BYOL	Bootstrap Your Own Latent
CNN	Convolutional neural network
EMA	Exponential moving average
FedAvg	Federated Averaging
FL	Federated learning

Abbreviation	Definition
FPGA	Field-programmable Gate Array
GD	Gradient Descent
IID	Identically and independently distributed
MCU	Microcontroller unit
MLP	Multilayer perceptron
NN	Neural network
SGD	Stochastic Gradient Descent
SimSiam	Simple Siamese representation learning
SOFed	Self-supervised On-device Federated learning
SSL	Self-supervised learning
WD	Weight decay

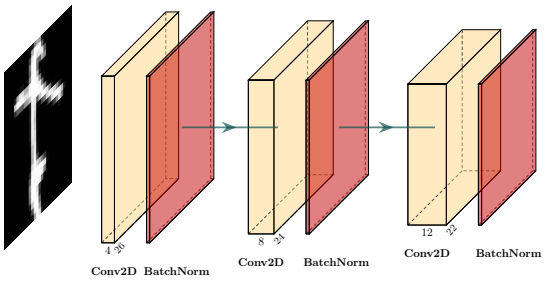
B

Extra figures and tables

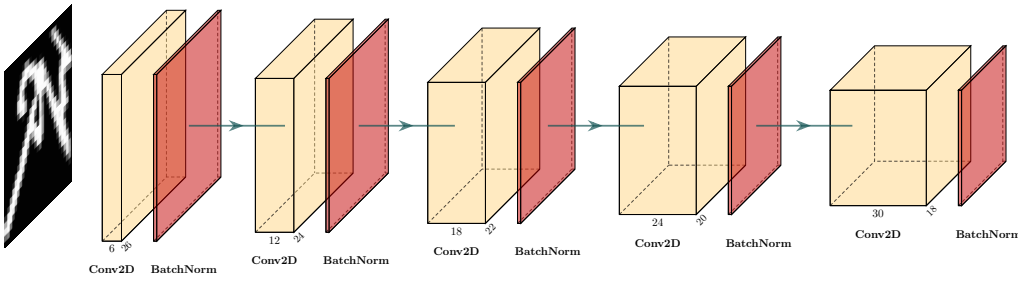
B.1. Self-supervised learning



(a) The structure of the 'simple' encoder.



(b) The structure of the 'medium' encoder.



(c) The structure of the 'advanced' encoder.

Figure B.1: The structures of the different encoder types.

B.2. Federated Learning

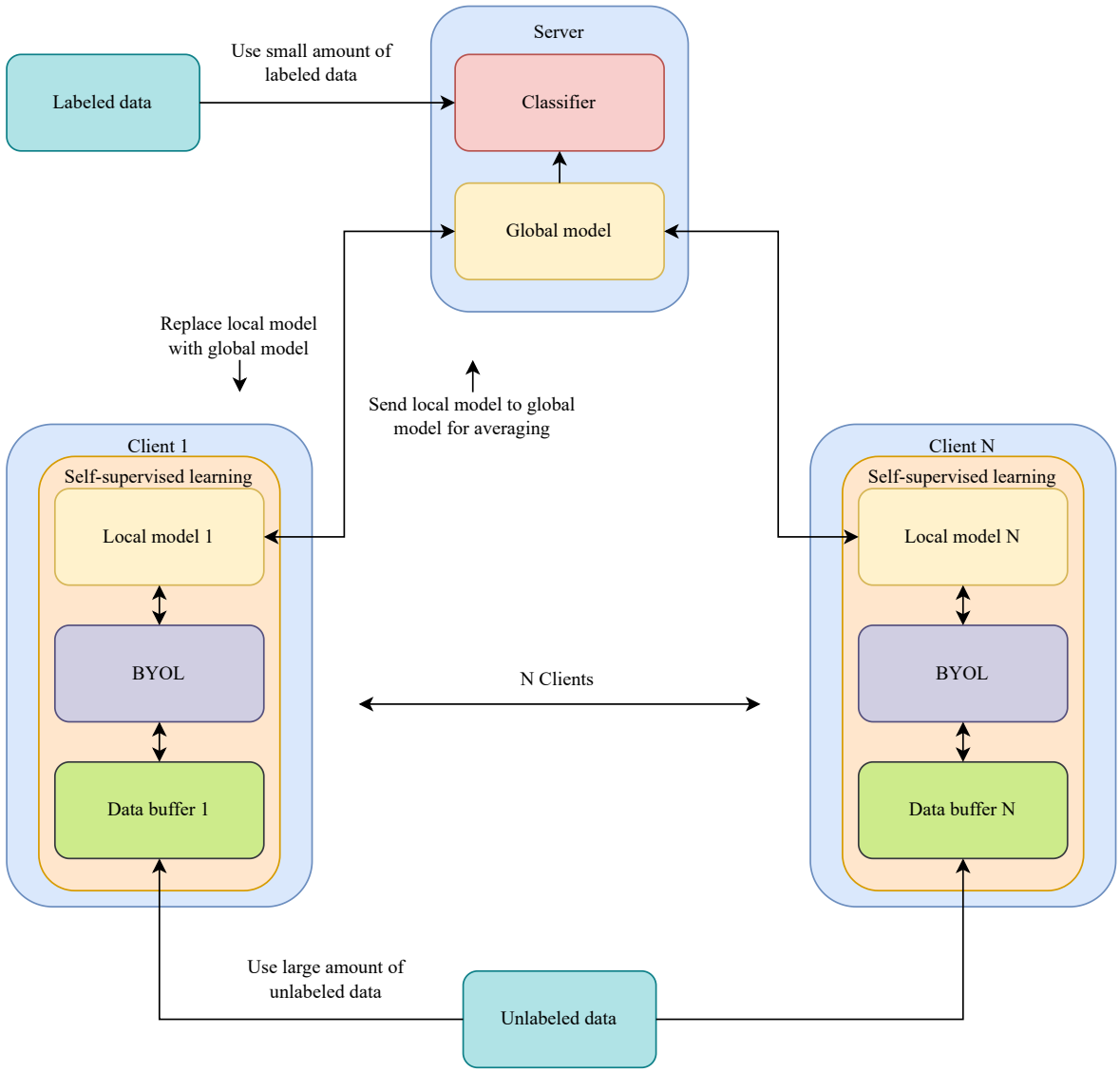


Figure B.2: Schematic diagram of SOFed. Each client performs training using data present in the data buffer on a global model, and the server combines these models every communication round. After training the model is used for downstream classification. The self-supervised learning method used is BYOL as based on the literature [5]

B.3. Model sizes

The sizes of the models in the case of only the encoder is given in table B.1 while the size including the predictor and projector is given in table B.2. These encoders correspond to the ones mentioned in chapter 5.1.1 and pictured in figures B.1a, B.1b, and B.1c.

Table B.1: Sizes of the three different encoders

Encoder name	Trainable parameters	Multiply-adds needed	Total size (MB)
Simple encoder	108	0.05M	0.15 MB
Medium encoder	1.26k	0.61M	0.30 MB
Advanced encoder	13.3k	0.74M	0.74 MB

Table B.2: Sizes of the three different setups (incl. encoder)

Encoder name	Trainable parameters	Multiply-adds needed	Total size (MB)
Simple encoder	86.9k	0.14M	0.56 MB
Medium encoder	200.2k	0.81M	1.25 MB
Advanced encoder	337.4k	5.35M	2.28 MB

B.4. Self-supervised learning tests

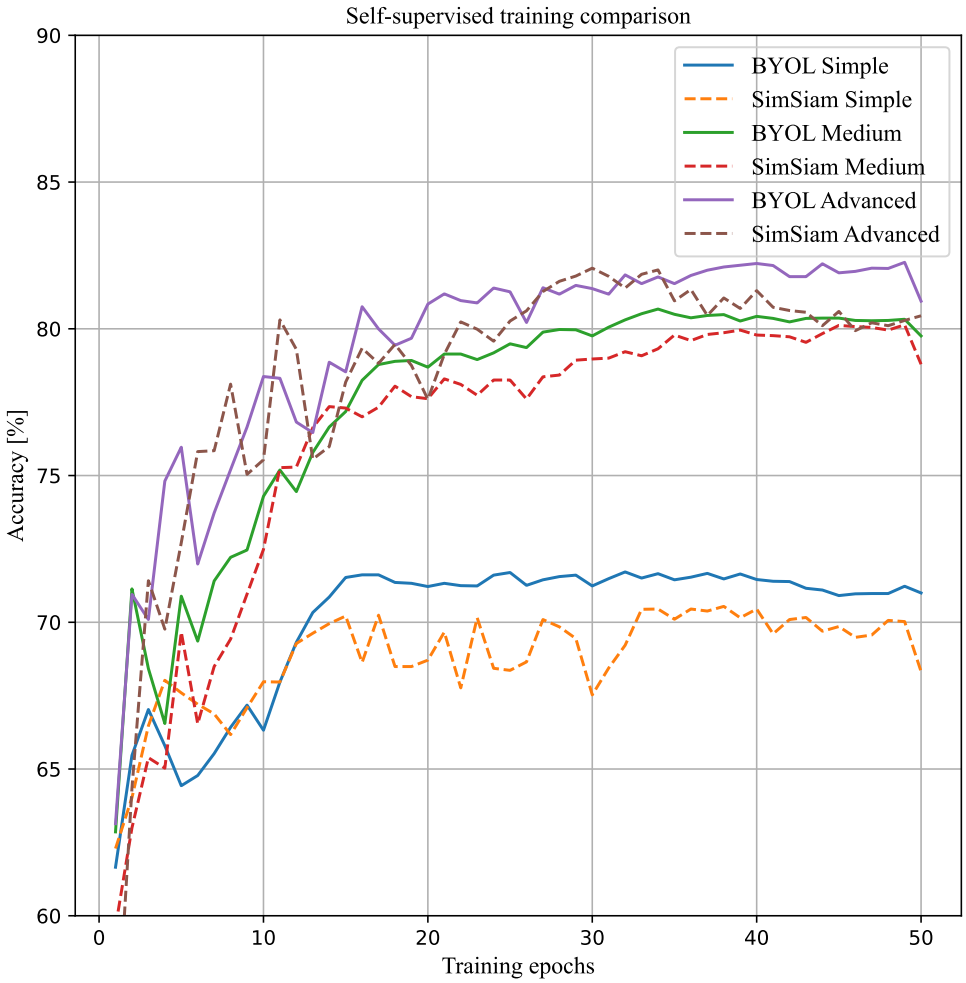


Figure B.3: Results of the SSL testing setup using the 3 different encoders for both BYOL and SimSiam.

B.4.1. Ablation study

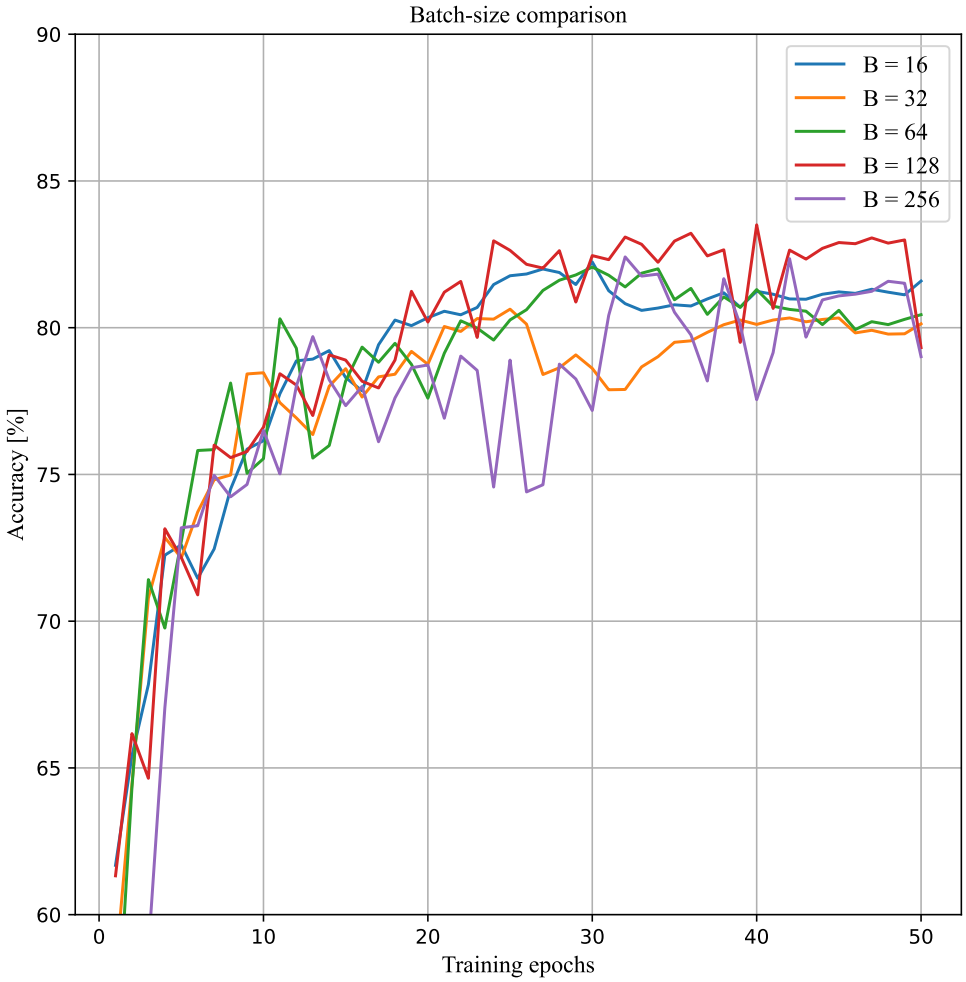


Figure B.4: Results of the ablation test for the batch size. The encoder used is the advanced encoder using SimSiam for the SSL algorithm.

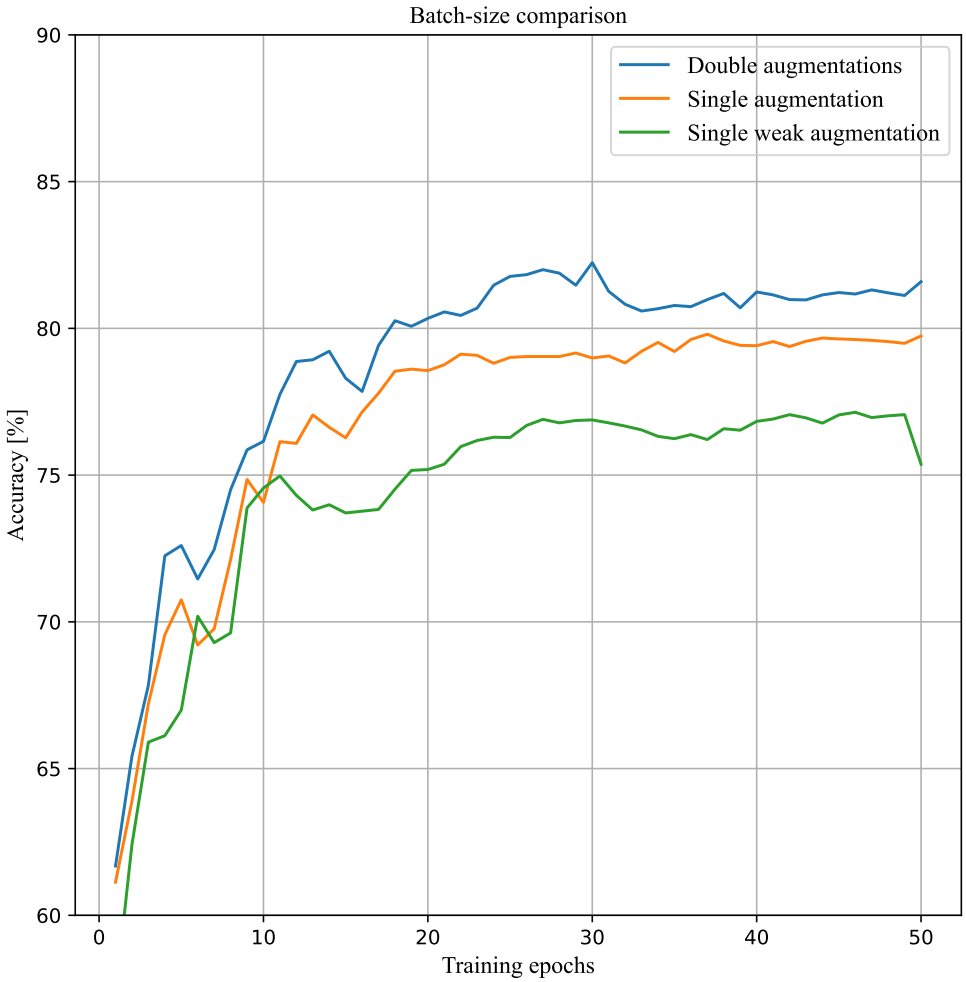


Figure B.5: Results of the ablation test for the augmentations. The encoder used is the advanced encoder using SimSiam for the SSL algorithm.

B.5. Federated learning tests

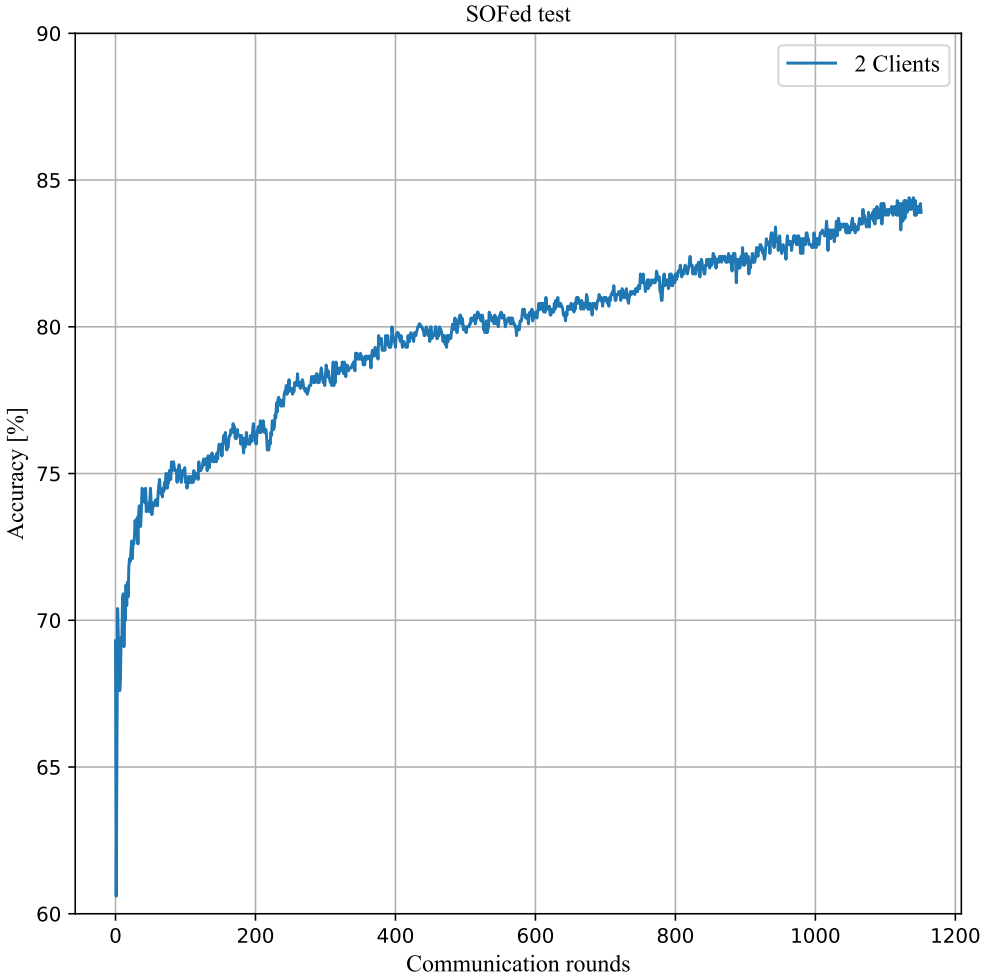


Figure B.6: Results of the SOFed testing setup using 2 clients.

C

Fulfillment of the requirements

In this chapter the requirements are revisited and the completion of each is discussed.

C.1. General requirements

1. **The system should use self-supervised federated learning.**
   The implemented software framework successfully performs self-supervised federated learning, indicating that this requirement is fulfilled.
2. **The system should be able to perform federated learning on an FPGA on the client side.**
   This requirement is not met completely: while the software framework can run on an FPGA, the hardware acceleration is not yet integrated with the software framework.
3. **The system should include a server that coordinates the federated learning among multiple clients. This server does not need to run on an FPGA.**
   The server is successfully implemented in the software framework. While minor improvements can be made, it functions as expected. This requirement is fulfilled.
4. **The system should be scalable to allow for multiple clients, with 2 clients being the minimum amount.**
   The tests show the functionality in the case of 2 clients. Considering that the results are hard to verify since the literature does not cover the expected result, it is unclear if the results are correct. The created software framework allow for the selection of more clients, thus this requirement would be met if the results of the system can be verified via further testing.
5. **The system's algorithms must be accelerated by using the programmable logic available on the client nodes.**
   This requirement is not part of the Signal Processing and Algorithms subgroup. For the reflection on the completion of this requirement the reader is referred to the thesis of the Hardware subgroup.

C.2. Signal Processing and Algorithms Requirements

1. **The system should implement the basic SOFed setup based on the SOFed paper [5].**
   The software framework implements the SOFed setup successfully. This requirement is fulfilled.
2. **In this paper, each client has a databuffer containing data for the network to train on. When the client receives new data, this data can be inserted into the databuffer according to its relevance. The system may implement the buffer first in a FIFO (first-in, first-out) manner, with the lazy-scoring buffer detailed in the paper serving as an optional aspect.**
   The software framework makes use of the lazy-scoring buffer for the storing of data at the clients. This requirement is fulfilled.
3. **The system should be optimized and downscaled enough such that it is possible to run the system on the FPGA board (e.g. using simpler machine learning models and algorithms, or using a more coarsely quantized number data format).**

While it is not entirely sure if the simplifications done are sufficient for the functionality on an FPGA, efforts were done to achieve this. Considering that the system is unable to run using the hardware acceleration, the requirement is not met completely.

4. **Several different machine learning models should be tested and compared.**
   The tests were done using multiple different encoders. This requirement is met.
5. **The memory usage of the different implementations should be found and compared.**
   The storage space in memory necessary for the training of each SSL setup is successfully found. This requirement is fulfilled.
6. **The compute requirements (number of multiply-adds and trainable parameters) of the different implementations should be found and compared.**
   The number of multiplies and additions needed in addition to the number of trainable parameters for each SSL setup were successfully found. This requirement is fulfilled.
7. **The effect of key algorithm parameters should be tested and evaluated.**
   For some of the parameters (batch size and type of augmentations) the effects on the performance were found. Other parameters (hidden sizes of the projector and predictor) remain to be tested. This requirement is not fulfilled completely.
8. **The self-supervised learning should have an effect: the downstream classification accuracy at the server side must be improved by training the model on unlabeled data.**
   The results show definite increases when SSL training is informed, and this is in-line with the theoretical results stemming from the literature. This requirement is fulfilled.

D

Implementation Details

This chapter details the Python implementation of the SSL and FL algorithms. The Python files can be found in appendix E.

D.1. Self-supervised learning

The full process of training and evaluation as well as the definition of the network models used has been explained now in general terms. This process and these definitions have been translated into and implemented as Python code. A complete walk-through of the code will not be given, since the explanation above should suffice as a blueprint of any implementation. However, some codebase specific things will be detailed in this section.

As a general remark: the code uses PyTorch [15] as a machine-learning framework, for its ease of use. The main files of the codebase will be regarded one-by-one.

D.1.1. Config.py

All hyper-parameters, as well as other configuration settings are set centrally in Config.py. This file contains a large dictionary of settings, whose parts are distributed to relevant portions of the program, and then expanded as a list of function arguments. In this manner, variables all over the program can be configured centrally. These configuration variables can also be set using the command-line.

D.1.2. Model.py

This file contains, aside from some helper functions, the model definitions, of the MLP, the Classifier, a generic encoder implementation which is used to create models of the 'simple', 'medium' and 'advanced' encoders, an implementation of other, more advanced encoders like MobileNetV2, and an implementation of the BYOL network. The BYOL class functions as SimSiam when τ is set to zero.

D.1.3. ImageAugmenter.py

This file contains a class and methods to augment a batch of images. Note the different augmentation types, as mentioned in section 5.1.3.

D.1.4. CheckPointer.py

This file contains a utility class used to save and load pretrained networks, such that training may be paused temporarily and resumed later.

D.1.5. Dataset.py

This file uses PyTorch vision to load different image datasets. Note that each dataset is splitted into three sections: training, classification and testing.

D.1.6. KRIAInterface.py
This file implements a function that emulates a quantized forward pass as a replacement for the actual hardware implementation. Weights and inputs are quantized to values allowed by a preconfigured fixed-point format, and the output is again quantized. The file then uses this function to implement a convolutional layer class, that is then used in Model.py as a direct replacement for the default PyTorch Conv2D class.

D.1.7. main.py
This file contains functions for the main SSL training loop, for the classification training loop, and a function for model evaluation. The main function itself initialises the different parts of the program, distributes, as mentioned, the configuration to each part, and then, depending on the configured mode, performs different training types: either SSL training ('pretrain'), and optionally training the classifier head every epoch, or evaluating the network every N epochs; evaluating a model ('eval'); or training the classifier after having trained the encoder ('evaltrain'/'evaltrainfrombyol'); or, a special mode, training the classifier, *without* fixing the weights of the encoder ('classtrain'). After having performed training, the program exits and outputs several training statistics like the final accuracy.

D.2. Federated Learning
The implementation of the SOFed algorithm in code reuses most of the SSL codebase detailed in section D.1. Three main files have been added however, which will be discussed in order.

D.2.1. Communication.py
This file implements the communication channel necessary for the server and clients to communicate. The TCP protocol is used, using the Python 'socket' library. Since the TCP protocol exposes an unstructured stream of byte data, a separate protocol has to be used on top of TCP. This protocol has been explained in section 5.2.1. The client and server communicate via messages, which consists of a 4-byte integer and a UTF-8 string. The integer represents the length of the string in bytes.

D.2.2. Client.py
The client implements the client SOFed algorithm as seen in algorithm 5. The client trains on a dataset exposed via an abstract base class 'DataSource', such that multiple data source implementations can be easily tested and compared. Two different data sources have been implemented: one allowing the client to train regularly on the whole KMNIST dataset, and one implementing the SOFed databuffer algorithm as seen in algorithm 6. Every time 'updateBuffer()' (see algorithm 5) is called, this implementation simulates the arrival of new image data by sampling $N_{streamcount}$ new images from the KMNIST dataset. This parameter has been set arbitrarily to $N_{streamcount} = 16$.

D.2.3. Server.py
This file implements the server according to algorithm 4. The timeout parameter has been set to $t_{timeout} = 1$s.

E

Python code

This appendix contains the Python code of the SSL and FL implementations. A brief explanation of the structure and each file can be found in appendix D.

E.1. Checkpointer.py

```
import os.path as path
import time
import datetime
import glob
import os
from enum import Enum

import torch

class CheckpointMode(Enum):
    DISABLED = -1,
    EVERY_EPOCH = 0,
    EVERY_BATCH = 1,
    EVERY_N_BATCHES = 2,
    EVERY_N_SECS = 3,

class Checkpointer:

    def __init__(self, directory, checkpointMode, saveOptimizerData=False, checkPointEveryNSecs=0,
    checkPointEveryNBatches=0, prefix=""):
        self.directory = directory
        self.mode = checkpointMode
        self.saveOptimizerData = saveOptimizerData
        self.checkPointEveryNSecs = checkPointEveryNSecs
        self.checkPointEveryNBatches = checkPointEveryNBatches

        self.lastEpoch = -1
        self.lastBatch = -1
        self.lastCheckpointTime = -1

        self.prefix = prefix
        self.runIdentifier = datetime.datetime.now().strftime("%d%m%y_%H%M%S")

    def getModelCheckpointPath(self):
        return path.join(self.directory, self.prefix + "Model_" + self.runIdentifier + "_" +
    str(self.lastEpoch) + ".pt")

    def getOptimizerCheckpointPath(self):
        return path.join(self.directory, self.prefix + "Optimizer_" + self.runIdentifier + "_" +
    str(self.lastEpoch) + ".pt")

    def update(self, model, optimizer, currentEpoch, maxEpochs, currentBatch, maxBatches):
        currentTime = time.time()
```

```
41
42          if self.mode == CheckpointMode.EVERY_EPOCH:
43              if currentEpoch != self.lastEpoch:
44                  self.lastEpoch = currentEpoch
45                  self.saveCheckpoint(model, optimizer)
46          elif self.mode == CheckpointMode.EVERY_BATCH:
47              if currentEpoch != self.lastEpoch or currentBatch != self.lastBatch:
48                  self.lastEpoch = currentEpoch
49                  self.lastBatch = currentBatch
50                  self.saveCheckpoint(model, optimizer)
51          elif self.mode == CheckpointMode.EVERY_N_SECS:
52              if currentTime > self.lastCheckpointTime + self.checkPointEveryNSecs:
53                  self.lastCheckpointTime = currentTime
54                  self.saveCheckpoint(model, optimizer)
55          elif self.mode == CheckpointMode.EVERY_N_BATCHES:
56              raise NotImplementedError
57          elif self.mode == CheckpointMode.DISABLED:
58              pass
59          else:
60              raise NotImplementedError
61
62          self.lastEpoch = currentEpoch
63
64      def loadLastCheckpoint(self, model, optimizer):
65          listOfFiles = glob.glob(self.directory + "/*")
66          latestFile = max(listOfFiles, key=lambda f: os.path.getctime(f) if
    f.split("\\")[-1].startswith(self.prefix) else 0).split("\\")[-1]
67
68          if not latestFile.startswith(self.prefix):
69              print(f"No {self.prefix} checkpoint found")
70              return
71
72          postfix = "_".join(latestFile.split("_")[1:])
73
74          return self.loadCheckpointFromPostfix(postfix, model, optimizer)
75
76      def loadCheckpointFromPostfix(self, postfix, model, optimizer):
77          print(f"Loading {self.prefix} checkpoint: {postfix}")
78
79          modelPath = path.join(self.directory, self.prefix + "Model_" + postfix)
80          if os.path.exists(modelPath):
81              model.load_state_dict(torch.load(modelPath))
82
83          if optimizer and self.saveOptimizerData:
84              optimizerPath = path.join(self.directory, self.prefix + "Optimizer_" + postfix)
85              if os.path.exists(optimizerPath):
86                  optimizer.load_state_dict(torch.load(optimizerPath))
87
88          return int(postfix.split("_")[-1].split(".")[0])
89
90      def loadCheckpoint(self, specificCheckpoint, model, optimizer):
91          if specificCheckpoint == None:
92              return self.loadLastCheckpoint(model, optimizer)
93          else:
94              return self.loadCheckpointFromPostfix(specificCheckpoint, model, optimizer)
95
96      def saveCheckpoint(self, model, optimizer):
97          torch.save(model.state_dict(), self.getModelCheckpointPath())
98
99          if optimizer and self.saveOptimizerData:
100             torch.save(optimizer.state_dict(), self.getOptimizerCheckpointPath())
```

E.2. Client.py

```
1 import torch
2 import torch.optim as optim
3
4 import time
5
6 import Dataset
```

```
import Config
import Model
import ImageAugmenter
import Communication
import Util

class DataSource:

    def initBuffer(self, model, device):
        pass

    def updateBuffer(self, model, device):
        pass

    def startEpoch(self):
        pass

    def getDataBatch(self): # Should not return target labels
        pass

    def isEpochFinished(self):
        pass

class DataBufferDataSource(DataSource):

    class DataBufferImage:
        def __init__(self, image, score, timeout):
            self.image = image
            self.score = score
            self.timeout = timeout

    def __init__(self, config):

        config = config.copy()
        config["dataset"]["batchSize"] = config["dataBuffer"]["datasetLoadBatchSize"] #Don't load complete batches from the dataset for filling the image buffer

        self.config = config

        self.dataset = Dataset.Dataset(**config["dataset"])
        self.enumeration = self.dataset.trainingEnumeration()

        self.augmenter = ImageAugmenter.ImageAugmenter(**config["augmenter"])

        self.buffer = [] # array of (image, score)
        self.bufferTargetSize = config["dataBuffer"]["bufferSize"]
        self.lazyScoringInterval = config["dataBuffer"]["lazyScoringInterval"]
        self.batchSize = config["dataBuffer"]["batchSize"]
        self.epochStreamCount = config["dataBuffer"]["epochStreamCount"]

        self.index = 0

    def calculateScore(self, model, device, image):
        image = torch.unsqueeze(image, 0)
        dataView1, dataView2 = self.augmenter.weaklyAugment(image)
        dataView1, dataView2 = dataView1.to(device), dataView2.to(device)

        model.eval()
        with torch.no_grad():
            loss = model(dataView1, dataView2)

        return loss.item()

    def updateAndStreamNewData(self, model, device, newImageCount):

        # Update the score of the images in the buffer when necessary (lazy scoring)
        rescoreCount = 0
        for bufferImage in self.buffer:
            bufferImage.timeout = bufferImage.timeout - 1
            if bufferImage.timeout == 0:
                bufferImage.score = self.calculateScore(model, device, bufferImage.image)
```

```
77                  bufferImage.timeout = self.lazyScoringInterval
78                  rescoreCount = rescoreCount + 1
79          print(f"Rescored {rescoreCount} images")
80
81          # Load newImageCount new images in batches, and add them to the buffer, keeping the images
    in the buffer with the highest scores
82          imagesProcessed = 0
83          newImagesAcceptedCount = 0
84          while imagesProcessed < newImageCount:
85              batchData, batchLabels = next(iter(self.enumeration))
86              for image in batchData:
87                  bufferImage = self.DataBufferImage(image, self.calculateScore(model, device,
    image), self.lazyScoringInterval)
88                  imagesProcessed = imagesProcessed + 1
89
90                  self.buffer.append(bufferImage)
91
92                  if len(self.buffer) > self.bufferTargetSize:
93
94                      # Buffer is now too large, remove the item with the lowest score
95
96                      lowestIndex = -1
97                      lowestScore = 100000000
98                      for index, bufferImage in enumerate(self.buffer):
99                          if bufferImage.score < lowestScore:
100                             lowestScore = bufferImage.score
101                             lowestIndex = index
102
103                     if lowestIndex != len(self.buffer) - 1:
104                         newImagesAcceptedCount += 1
105
106                     self.buffer.pop(lowestIndex)
107                 else:
108                     newImagesAcceptedCount += 1
109         print(f"Accepted {newImagesAcceptedCount} new images")
110
111     def printBuffer(self):
112         for index, bufferImage in enumerate(self.buffer):
113             print(f"Image #{index}: score={bufferImage.score}, timeout={bufferImage.timeout}")
114
115     def initBuffer(self, model, device):
116         self.updateAndStreamNewData(model, device, self.bufferTargetSize)
117
118     def updateBuffer(self, model, device):
119         self.updateAndStreamNewData(model, device, self.epochStreamCount)
120
121     def startEpoch(self):
122         self.index = 0
123
124     def getDataBatch(self):
125         dataAmountToFetch = min(self.batchSize, len(self.buffer) - self.index)
126         images = torch.stack([bufferImage.image for bufferImage in self.buffer[-self.index -
    dataAmountToFetch-1:-self.index-1]])
127         self.index += dataAmountToFetch
128         return images
129
130     def getProgress(self):
131         return self.index / len(self.buffer)
132
133     def isEpochFinished(self):
134         return self.index == len(self.buffer)
135
136 class DatasetDataSource(DataSource):
137     def __init__(self, config):
138         self.dataset = Dataset.Dataset(**config["dataset"])
139         self.enumeration = self.dataset.trainingEnumeration()
140
141         self.len = self.dataset.trainBatchCount() // self.dataset.batchSize
142         self.index = 0
143
144     def startEpoch(self):
```

```
145         self.index = 0
146 
147     def getDataBatch(self):
148         data, target = next(iter(self.enumeration))
149         self.index = self.index + 1
150         return data # Note: we don't return the target labels!
151 
152     def getProgress(self):
153         return self.index / self.len
154 
155     def isEpochFinished(self):
156         return self.index == self.len
157 
158 class Client:
159     def __init__(self, device, config, dataSource : DataSource):
160         self.device = device
161         self.config = config
162         self.dataSource = dataSource
163 
164         self.emaScheduler = Util.EMAScheduler(**config["EMA"])
165         self.model = Model.BYOL(self.emaScheduler, **config["BYOL"]).to(device)
166         self.optimizer = getattr(optim,
    config["optimizer"]["name"])(self.model.trainableParameters(),
    **config["optimizer"]["settings"])
167 
168         self.augmenter = ImageAugmenter.ImageAugmenter(**config["augmenter"])
169 
170         self.communication = Communication.Communication()
171 
172     def connect(self, ip, port):
173         print(f"Connecting to {ip}:{port}")
174         self.communication.connect(ip, port)
175 
176     def run(self):
177         try:
178             self.run_()
179         except KeyboardInterrupt:
180             pass
181 
182         self.communication.sendMessage("stop")
183         while not self.communication.isDataReady():
184             time.sleep(1)
185         self.communication.receiveMessage()
186         self.communication.close()
187 
188     def run_(self):
189         shouldStop = False
190 
191         epoch = 0
192 
193         self.dataSource.initBuffer(self.model, self.device)
194 
195         while not shouldStop and epoch < self.config["training"]["epochs"]:
196 
197             if epoch % self.config["client"]["serverSyncEveryNEpochs"] == 0:
198                 print("Loading model from server")
199                 self.communication.sendMessage("requestSend")
200                 self.communication.receiveModel(self.model)
201 
202             print(f"Training BYOL Epoch {epoch + 1}: lr={self.optimizer.param_groups[0]['lr']}")
203             self.dataSource.startEpoch()
204 
205             self.model.train()
206 
207             batchIndex = 0
208             while not self.dataSource.isEpochFinished():
209                 data = self.dataSource.getDataBatch()
210                 dataView1, dataView2 = self.augmenter.createImagePairBatch(data)
211                 dataView1, dataView2 = dataView1.to(self.device), dataView2.to(self.device)
212 
213                 self.optimizer.zero_grad()
```

```
                loss = self.model(dataView1, dataView2)
                loss.backward()
                self.optimizer.step()
                self.model.stepEMA()

                if batchIndex % 1 == 0:
                    print(f"Epoch {epoch + 1}, batch {batchIndex}/{self.dataSource.getProgress() * 100:.1f}%: loss={loss:.4f}")
                batchIndex = batchIndex + 1

            if epoch % self.config["client"]["updateBufferEveryNEpochs"] == 0:
                print("Updating buffer...")
                self.dataSource.updateBuffer(self.model, self.device)

            epoch = epoch + 1

            # Note: checking after epoch+1! We load from server at the first epoch in a sequence, and send to the server at the last of the sequence
            if epoch % self.config["client"]["serverSyncEveryNEpochs"] == 0:
                print("Sending model to server")
                self.communication.sendMessage("update")
                self.communication.sendModel(self.model)

def main():
    config = Config.GetConfig()

    #torch.manual_seed(0) # Don't seed, such that all clients train in a different way.
    device = Util.GetDeviceFromConfig(config)

    Model.SetUseReLU1(config["useReLU1"])

    #dataSource = DatasetDataSource(config)
    dataSource = DataBufferDataSource(config)
    client = Client(device, config, dataSource)

    client.connect("localhost", 1234)
    client.run()

if __name__ == "__main__":
    main()
```

E.3. Communication.py

```
import socket
import time

import select
import torch
import struct
import io

class Server:
    def __init__(self):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    def bind(self, ip, port):
        self.socket.bind((ip, port))
        self.socket.listen()
        self.socket.setblocking(False)

    def close(self):
        self.socket.close()

    def tryAcceptClient(self):
        readable, writable, errored = select.select([self.socket], [], [], 0)
        if len(readable):
            clientSocket, addr = self.socket.accept()
            return Communication(clientSocket), addr
        else:
            return None, None
```

```
28
29 class Communication:
30     def __init__(self, initialSocket = None):
31         if initialSocket == None:
32             self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
33         else:
34             self.socket = initialSocket
35
36     def connect(self, ip, port):
37         self.socket.connect((ip, port))
38
39     def close(self):
40         self.socket.close()
41
42     def sendModel(self, model):
43         bytesStream = io.BytesIO()
44         torch.save(model.state_dict(), bytesStream)
45
46         data = bytesStream.getvalue()
47         length = len(data)
48
49         packedLength = struct.pack("!i", length)
50
51         self.socket.sendall(packedLength)
52         self.socket.sendall(data)
53
54     def receiveModel(self, model):
55         packedLength = recvall(self.socket, 4)
56         length = struct.unpack("!i", packedLength)[0]
57
58         data = recvall(self.socket, length)
59
60         bytesReadStream = io.BytesIO(data)
61         statesDict = model.state_dict()
62         model.load_state_dict(torch.load(bytesReadStream))
63         bytesReadStream.close()
64
65     def sendMessage(self, text):
66         data = text.encode("utf-8")
67         length = len(data)
68
69         packedLength = struct.pack("!i", length)
70
71         self.socket.sendall(packedLength)
72         self.socket.sendall(data)
73
74     def receiveMessage(self):
75         packedLength = recvall(self.socket, 4)
76         length = struct.unpack("!i", packedLength)[0]
77
78         data = recvall(self.socket, length)
79         return data.decode("utf-8")
80
81     def isDataReady(self):
82         read_sockets, write_sockets, error_sockets = select.select([self.socket], [], [], 0)
83         return len(read_sockets) == 1
84
85 def recvall(socket : socket.socket, n):
86     # Helper function to recv n bytes or return None if EOF is hit
87     data = bytearray()
88     while len(data) < n:
89         try:
90             packet = socket.recv(n - len(data))
91             if not packet:
92                 return None
93             data.extend(packet)
94         except BlockingIOError: #No data available yet, wait a bit then try again
95             time.sleep(0.010)
96     return data
```

E.4. Config.py

```
import Checkpointer

def GetConfig(doPostConfig=True):
    config = dict(
        device="cuda",
        mode="pretrain",
        loadFromCheckpoint=False,
        loadFromSpecificCheckpoint=None,
        printStatistics=True,
        useHalfPrecision=True,
        useReLU1=True,

        augmenter=dict(
            imageDims=(0, 0), #autoset
            applyColorAugments=False,
            applyFlips=False,
        ),

        training=dict(
            epochs=50,
            warmupEpochs=1,
            evaluateEveryNEpochs=1,
            classifierEpochs=1,
            finalclassifierEpochs=20,
        ),
        dataset=dict(
            datasetName="KMNIST",
            normalization=None, #autoset
            batchSize=64,
            classificationSplit=0.1,
        ),
        EMA=dict(
            initialTau=0.99,
            epochCount=None, #autoset
            enableSchedule=True
        ),
        classifier=dict(
            hiddenSize=128,
            batchNorm=None #autoset
        ),
        BYOL=dict(
            encoderName="EncoderType1",
            projector=dict(
                hiddenSize=128,
                outputSize=32,
            ),
            predictor=dict(
                hiddenSize=128
            ),
            encoder=dict(
                imageDims=(0, 0), #autoset
                imageChannels=0 #autoset
            ),
            batchNorm=None #autoset
        ),
        #optimizer=dict(
        #    name="AdamW",
        #    settings=dict(
        #        lr=0.0003,
        #        weight_decay=0.0001
        #    )
        #),
        optimizer=dict(
            name="SGD",
            settings=dict(
                lr=0.05,
                weight_decay=0.0001,
                momentum=0.9
            )
```

```
70          ),
71          batchNorm=dict(
72              eps=1e-5,
73              momentum=0.1
74          ),
75          quantization = dict(
76              enabled=True,
77              nb=12,
78              nf=7,
79              quantizeWeights=False,
80              useCustomConv=True
81          ),
82          checkpointer=dict(
83              directory="src/checkpoints",
84              #checkpointMode=Checkpointer.CheckpointMode.EVERY_N_SECS,
85              checkpointMode=Checkpointer.CheckpointMode.EVERY_EPOCH,
86              checkPointEveryNSecs=30,
87              saveOptimizerData=True
88          ),
89          dataBuffer=dict(
90              datasetLoadBatchSize=16,
91              bufferSize=128,
92              batchSize=None, #autoset
93              lazyScoringInterval=50,
94              epochStreamCount=16,
95          ),
96          client=dict(
97              serverSyncEveryNEpochs=100,
98              updateBufferEveryNEpochs=1
99          ),
100         server=dict(
101             classifierTrainEpochs=25,
102         )
103     )
104 
105     if doPostConfig:
106         DoPostConfig(config)
107 
108     return config
109 
110 def DoPostConfig(config):
111     config["BYOL"]["batchNorm"] = config["batchNorm"]
112     config["BYOL"]["dtypeName"] = "float16" if config["useHalfPrecision"] else "float32"
113     config["BYOL"]["quantization"] = config["quantization"]
114     config["classifier"]["dtypeName"] = "float16" if config["useHalfPrecision"] else "float32"
115     config["classifier"]["batchNorm"] = config["batchNorm"]
116     config["classifier"]["encoder"] = config["BYOL"]["encoder"]
117     config["classifier"]["encoderName"] = config["BYOL"]["encoderName"]
118     config["classifier"]["quantization"] = config["quantization"]
119     config["dataBuffer"]["batchSize"] = config["dataset"]["batchSize"]
120     config["EMA"]["epochCount"] = config["training"]["epochs"]
121     config["augmenter"]["useHalfPrecision"] = config["useHalfPrecision"]
122 
123     config["optimizer"]["settings"]["lr"] = config["optimizer"]["settings"]["lr"] *
    config["dataset"]["batchSize"] / 64
124 
125     if config["EMA"]["initialTau"] > 0.01:  # Tau=0 means EMA disabled, so don't scale it.
    Otherwise, do scale.
126         config["EMA"]["initialTau"] = 1 - (1 - config["EMA"]["initialTau"]) *
    (config["dataset"]["batchSize"] / 64)
127 
128     if config["dataset"]["datasetName"] == "MNIST":
129         config["augmenter"]["imageDims"] = (28, 28)
130         config["dataset"]["normalization"] = ((0.1307,), (0.3081,))
131         config["BYOL"]["encoder"]["imageDims"] = (28, 28)
132         config["BYOL"]["encoder"]["imageChannels"] = 1
133         config["classifier"]["classCount"] = 10
134     elif config["dataset"]["datasetName"] == "CIFAR10":
135         config["augmenter"]["imageDims"] = (32, 32)
136         config["augmenter"]["applyColorAugments"] = True
137         config["augmenter"]["applyFlips"] = True
```

```
        config["dataset"]["normalization"] = ((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
        config["BYOL"]["encoder"]["imageDims"] = (32, 32)
        config["BYOL"]["encoder"]["imageChannels"] = 3
        config["classifier"]["classCount"] = 10
    elif config["dataset"]["datasetName"] == "FashionMNIST":
        config["augmenter"]["imageDims"] = (28, 28)
        config["dataset"]["normalization"] = ((0.1307,), (0.3081,))
        config["BYOL"]["encoder"]["imageDims"] = (28, 28)
        config["BYOL"]["encoder"]["imageChannels"] = 1
        config["classifier"]["classCount"] = 10
    elif config["dataset"]["datasetName"] == "KMNIST":
        config["augmenter"]["imageDims"] = (28, 28)
        config["dataset"]["normalization"] = ((0.1917,), (0.3483,))
        config["BYOL"]["encoder"]["imageDims"] = (28, 28)
        config["BYOL"]["encoder"]["imageChannels"] = 1
        config["classifier"]["classCount"] = 10
    elif config["dataset"]["datasetName"] == "EMNIST":
        config["augmenter"]["imageDims"] = (28, 28)
        config["dataset"]["normalization"] = ((0.1307,), (0.3081,))
        config["BYOL"]["encoder"]["imageDims"] = (28, 28)
        config["BYOL"]["encoder"]["imageChannels"] = 1
        config["classifier"]["classCount"] = 47
    else:
        raise NotImplementedError
```

E.5. Dataset.py

```
import torch
from torchvision import datasets, transforms
import torchvision.transforms.functional as F

class ToHalfTensor(torch.nn.Module):
    def forward(self, img):
        return F.to_tensor(img).half()

class Dataset:

    def __init__(self, datasetName, batchSize, normalization, classificationSplit):

        transform = transforms.Compose([
            #ToHalfTensor() if useHalfPrecision else transforms.ToTensor(),
            transforms.ToTensor(),
            transforms.Normalize(normalization[0], normalization[1])
        ])

        if (datasetName == 'EMNIST'):
            self.train = getattr(datasets, datasetName)('datasets', train=True,
    download=True,split='balanced', transform=transform)
            self.test = getattr(datasets, datasetName)('datasets', train=False, split='balanced',
    transform=transform)
        else:
            self.train = getattr(datasets, datasetName)('datasets', train=True, download=True,
    transform=transform)
            self.test = getattr(datasets, datasetName)('datasets', train=False,
    transform=transform)

        self.train, self.classification = torch.utils.data.random_split(self.train, [1 -
    classificationSplit, classificationSplit])

        self.trainLoader = torch.utils.data.DataLoader(self.train, batch_size=batchSize)
        self.classificationLoader = torch.utils.data.DataLoader(self.classification,
    batch_size=batchSize)
        self.testLoader = torch.utils.data.DataLoader(self.test, batch_size=batchSize)

        self.batchSize = batchSize

    def trainingEnumeration(self):
        return self.trainLoader

    def classificationEnumeration(self):
```

```
        return self.classificationLoader

    def testingEnumeration(self):
        return self.testLoader

    def trainBatchCount(self):
        return len(self.trainLoader.dataset)

    def classificationBatchCount(self):
        return len(self.classificationLoader.dataset)

    def testBatchCount(self):
        return len(self.testLoader.dataset)
```

E.6. ImageAugmenter.py

```
import torch
import torchvision.transforms.v2 as transforms
import torchvision.transforms.functional as F

class ToHalfTensor(torch.nn.Module):
    def forward(self, img):
        return img.half()

class UnityTransform(torch.nn.Module):
    def forward(self, img):
        return img

class ImageAugmenter:
    def __init__(self, imageDims, applyFlips=False, applyColorAugments=False, useHalfPrecision=False):
        self.transform = transforms.Compose([
            transforms.RandomApply(torch.nn.ModuleList([transforms.GaussianBlur((3, 3), (1.0, 2.0))]), p=0.2),
            transforms.RandomRotation(degrees=30),
            transforms.RandomResizedCrop(size=imageDims, antialias=True, scale=(0.5, 1)),
        ])

        if applyColorAugments:
            self.transform = transforms.Compose([
                self.transform,
                transforms.ColorJitter(0.4, 0.4, 0.2, 0.1),
                transforms.RandomGrayscale(p=0.2),
                #transforms.GaussianBlur(9, sigma=(0.1, 0.2)),
                transforms.RandomSolarize(threshold=0.5, p=0.2)
            ])

        if applyFlips:
            self.transform = transforms.Compose([
                self.transform,
                transforms.RandomHorizontalFlip(p=0.5),
                #transforms.RandomVerticalFlip(p=0.5),
            ])


        if useHalfPrecision:
            self.transform = transforms.Compose([
                self.transform,
                ToHalfTensor()
            ])
            self.weakTransform = transforms.Compose([
                transforms.RandomResizedCrop(size=imageDims, antialias=True, scale=(0.8, 1)),
                ToHalfTensor()
            ])
            self.noTransform = ToHalfTensor()
        else:
            self.weakTransform = transforms.RandomResizedCrop(size=imageDims, antialias=True, scale=(0.8, 1))
            self.noTransform = UnityTransform()

```

```
52      def createImagePairBatch(self, imageBatch):
53          return torch.stack([self.transform(image) for image in imageBatch]),
    torch.stack([self.transform(image) for image in imageBatch])
54
55      def createImagePairBatchSingleAugment(self, imageBatch):
56          return self.noTransform(imageBatch), torch.stack([self.transform(image) for image in
    imageBatch])
57
58      def weaklyAugment(self, image):
59          return self.noTransform(image), self.weakTransform(image)
```

E.7. KRIAInterface.py

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  import math
5
6  import QNN
7
8  class FConv2D_3x3(torch.autograd.Function):
9      """
10     We can implement our own custom autograd Functions by subclassing
11     torch.autograd.Function and implementing the forward and backward passes
12     which operate on Tensors.
13     """
14
15     @staticmethod
16     def forward(ctx, input, weight, bias):
17         """
18         In the forward pass we receive a Tensor containing the input and return
19         a Tensor containing the output. ctx is a context object that can be used
20         to stash information for backward computation. You can cache arbitrary
21         objects for use in the backward pass using the ctx.save_for_backward method.
22         """
23         ctx.save_for_backward(input, weight, bias)
24
25         result = QNN.quantize(F.conv2d(QNN.quantize(input), QNN.quantize(weight)))
26
27         if bias is not None:
28             reshapedBias = bias.reshape(1, bias.shape[0], 1, 1).repeat(result.shape[0], 1,
    result.shape[2], result.shape[3])
29             result += reshapedBias
30
31         return result
32
33     @staticmethod
34     @torch.autograd.function.once_differentiable
35     def backward(ctx, grad_output):
36         """
37         In the backward pass we receive a Tensor containing the gradient of the loss
38         with respect to the output, and we need to compute the gradient of the loss
39         with respect to the input.
40         """
41         input, weight, bias = ctx.saved_tensors
42
43         grad_input = grad_weight = grad_bias = None
44         if ctx.needs_input_grad[0]:
45             grad_input = F.conv_transpose2d(grad_output, weight)
46         if ctx.needs_input_grad[1]:
47             grad_weight = F.conv2d(input.transpose(0, 1), grad_output.transpose(0,
    1)).transpose(0, 1)
48         if bias is not None and ctx.needs_input_grad[2]:
49             grad_bias = grad_output.sum((0, 2, 3)).squeeze(0)
50
51         return grad_input, grad_weight, grad_bias
52
53 class Conv2D_3x3(nn.Module):
54     def __init__(self, inChannels, outChannels, dtype=torch.float32, bias=True):
55         super(Conv2D_3x3, self).__init__()
```

```
56
57          self.inChannels = inChannels
58          self.outChannels = outChannels
59          self.dtype = dtype
60
61          self.weight = torch.nn.Parameter(torch.empty((outChannels, inChannels, 3, 3), dtype=dtype))
62
63          if bias:
64              self.bias = torch.nn.Parameter(torch.empty(outChannels, dtype=dtype))
65          else:
66              self.register_parameter('bias', None)
67
68          # Initialize parameters, from ConvNd.reset_parameters
69          torch.nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
70          if self.bias is not None:
71              fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.weight)
72              if fan_in != 0:
73                  bound = 1 / math.sqrt(fan_in)
74                  torch.nn.init.uniform_(self.bias, -bound, bound)
75
76      def forward(self, input):
77          return FConv2D_3x3.apply(input, self.weight, self.bias)
```

E.8. main.py

```
1  import torch
2  import torch.optim as optim
3
4  import argparse
5
6  import Checkpointer
7  import Model
8  import Config
9  import Dataset
10 import ImageAugmenter
11 import Util
12
13 def TrainBYOLEpoch(byol, device, dataset, optimizer, augmenter, checkpointer, epoch, maxEpochs):
14     byol.train()  # Enables dropout
15
16     print(f"Training BYOL Epoch {epoch + 1}: lr={optimizer.param_groups[0]['lr']},
       tau={byol.emaScheduler.getTau()}")
17
18     maxTrainBatches = dataset.trainBatchCount() / dataset.batchSize
19     for batchIndex, (data, target) in enumerate(dataset.trainingEnumeration()):
20         data = data.to(device)
21         dataView1, dataView2 = augmenter.createImagePairBatch(data)
22         #dataView1, dataView2 = dataView1.to(device), dataView2.to(device)
23
24         optimizer.zero_grad()
25         loss = byol(dataView1, dataView2)
26         loss.backward()
27         optimizer.step()
28         byol.stepEMA()
29         byol.quantizeParameters()
30
31         checkpointer.update(byol, optimizer, epoch, maxEpochs, batchIndex, maxTrainBatches)
32
33         if batchIndex % 10 == 0:
34             print(f"Epoch {epoch + 1}, batch {batchIndex}/{batchIndex / maxTrainBatches *
       100:.1f}%: BYOLLoss={loss:.4f}")
35
36 def TrainClassifierEpoch(classifier, device, dataset, optimizer, checkpointer, epoch, maxEpochs,
       useHalfPrecision):
37     classifier.train()
38
39     print(f"Training Classifier Epoch {epoch + 1}: lr={optimizer.param_groups[0]['lr']}")
40
41     maxClassifierBatches = dataset.classificationBatchCount() / dataset.batchSize
42     for batchIndex, (data, target) in enumerate(dataset.classificationEnumeration()):
```

```
43          data, target = data.to(device), target.to(device)
44
45          if useHalfPrecision:
46              data = data.half()
47
48          optimizer.zero_grad()
49          loss = classifier.loss(data, target)
50          loss.backward()
51          optimizer.step()
52
53          checkpointer.update(classifier, optimizer, epoch, maxEpochs, batchIndex,
    maxClassifierBatches)
54
55          if batchIndex % 10 == 0:
56              print(
57                  f"Epoch {epoch + 1}, batch {batchIndex}/{batchIndex / maxClassifierBatches *
    100:.1f}%: classificationLoss={loss:.2f}")
58
59
60  def TestEpoch(classifier, device, dataset, useHalfPrecision):
61      classifier.eval() # Disable dropout
62
63      testLoss = 0
64      accuracy = 0
65      batchCount = 0
66      with torch.no_grad():
67          for batchIndex, (data, target) in enumerate(dataset.testingEnumeration()):
68              data, target = data.to(device), target.to(device)
69
70              if useHalfPrecision:
71                  data = data.half()
72
73              loss, output, prediction = classifier.predictionLoss(data, target)
74
75              testLoss += loss.item()
76              accuracy += prediction.eq(target.view_as(prediction)).sum().item() / len(data)
77
78              batchCount += 1
79
80      testLoss /= batchCount
81      accuracy /= batchCount
82
83      print(f"Evaluation: loss={testLoss:2f}, accuracy={accuracy * 100:.1f}%")
84
85      return testLoss, accuracy
86
87  def ParseArgs(config):
88      parser = argparse.ArgumentParser()
89
90      # From: https://stackoverflow.com/questions/15008758/parsing-boolean-values-with-argparse
91      def t_or_f(arg):
92          ua = str(arg).upper()
93          if 'TRUE'.startswith(ua):
94              return True
95          elif 'FALSE'.startswith(ua):
96              return False
97          else:
98              pass  # error condition maybe?
99
100     def PopulateDict(prefix, dictionary):
101         for key, value in dictionary.items():
102             if isinstance(value, dict):
103                 PopulateDict(prefix + key + ".", value)
104             else:
105                 if type(value) == bool:
106                     parser.add_argument("--" + prefix + key, required=False, type=t_or_f)
107                 else:
108                     parser.add_argument("--" + prefix + key, required=False, type=type(value))
109
110     PopulateDict("", config)
111     result = parser.parse_args()
```

```
112
113     def RetrieveDict(prefix, dictionary):
114         for key, value in dictionary.items():
115             if isinstance(value, dict):
116                 RetrieveDict(prefix + key + ".", value)
117             else:
118                 resultValue = getattr(result, prefix + key)
119                 if not (resultValue is None):
120                     dictionary[key] = resultValue
121
122     RetrieveDict("", config)
123
124 def main():
125     config = Config.GetConfig(doPostConfig=False)
126     ParseArgs(config)
127     Config.DoPostConfig(config)
128
129     torch.manual_seed(0)
130     device = Util.GetDeviceFromConfig(config)
131     statistics = []
132
133     dataset = Dataset.Dataset(**config["dataset"])
134     emaScheduler = Util.EMAScheduler(**config["EMA"])
135     byol = Model.BYOL(emaScheduler, **config["BYOL"]).to(device)
136     classifier = Model.Classifier(**config["classifier"]).to(device)
137     byolCheckpointer = Checkpointer.Checkpointer(**config["checkpointer"], prefix="BYOL")
138     classifierCheckpointer = Checkpointer.Checkpointer(**config["checkpointer"],
    prefix="Classifier")
139
140     Model.SetUseReLU1(config["useReLU1"])
141
142     if config["mode"] == "pretrain":
143         byolOptimizer = getattr(optim, config["optimizer"]["name"])(byol.trainableParameters(),
    **config["optimizer"]["settings"])
144         classifierOptimizer = getattr(optim,
    config["optimizer"]["name"])(classifier.trainableParameters(),
    **config["optimizer"]["settings"])
145
146         startEpoch = 0
147         if config["loadFromCheckpoint"]:
148             startEpoch = byolCheckpointer.loadCheckpoint(config["loadFromSpecificCheckpoint"],
    byol, byolOptimizer)
149             classifierCheckpointer.loadCheckpoint(config["loadFromSpecificCheckpoint"],
    classifier, classifierOptimizer)
150
151         lrScheduler = Util.WarmupCosineScheduler(byolOptimizer, startEpoch,
    config["training"]["epochs"], config["training"]["warmupEpochs"],
    config["optimizer"]["settings"]["lr"])
152         emaScheduler.startStep(startEpoch)
153
154         augmenter = ImageAugmenter.ImageAugmenter(**config["augmenter"])
155
156         try:
157             for epoch in range(startEpoch, config["training"]["epochs"]):
158                 TrainBYOLEpoch(byol, device, dataset, byolOptimizer, augmenter, byolCheckpointer,
    epoch, config["training"]["epochs"])
159                 classifier.copyEncoderFromBYOL(byol)
160                 for i in range(config["training"]["classifierEpochs"]):
161                     TrainClassifierEpoch(classifier, device, dataset, classifierOptimizer,
    classifierCheckpointer, epoch, config["training"]["epochs"], config["useHalfPrecision"])
162
163                 if config["training"]["evaluateEveryNEpochs"] != 0 and epoch %
    config["training"]["evaluateEveryNEpochs"] == 0:
164                     testResults = TestEpoch(classifier, device, dataset,
    config["useHalfPrecision"])
165                     statistics.append((*testResults, epoch))
166
167                 lrScheduler.step()
168                 classifierOptimizer.param_groups[0]["lr"] = byolOptimizer.param_groups[0]["lr"]
169                 emaScheduler.step(epoch)
170         except KeyboardInterrupt:
```

```
171             pass
172 
173         for i in range(config["training"]["finalclassifierEpochs"]):
174             TrainClassifierEpoch(classifier, device, dataset, classifierOptimizer,
    classifierCheckpointer, config["training"]["epochs"], config["training"]["epochs"],
    config["useHalfPrecision"])
175         testResults = TestEpoch(classifier, device, dataset, config["useHalfPrecision"])
176         statistics.append((*testResults, config["training"]["epochs"]))
177 
178     elif config["mode"] == "eval":
179         if config["loadFromCheckpoint"]:
180             classifierCheckpointer.loadCheckpoint(config["loadFromSpecificCheckpoint"],
    classifier, None)
181 
182         TestEpoch(classifier, device, dataset, config["useHalfPrecision"])
183     elif config["mode"] == "evaltrain" or config["mode"] == "evaltrainfrombyol":
184         classifierOptimizer = getattr(optim,
    config["optimizer"]["name"])(classifier.trainableParameters(),
    **config["optimizer"]["settings"])
185 
186         startEpoch = 0
187 
188         if config["mode"] == "evaltrainfrombyol":
189             if config["loadFromCheckpoint"]:
190                 startEpoch = byolCheckpointer.loadCheckpoint(config["loadFromSpecificCheckpoint"],
    byol, None)
191             classifier.copyEncoderFromBYOL(byol)
192         else:
193             if config["loadFromCheckpoint"]:
194                 startEpoch =
    classifierCheckpointer.loadCheckpoint(config["loadFromSpecificCheckpoint"], classifier,
    classifierOptimizer)
195 
196         lrScheduler = Util.WarmupCosineScheduler(classifierOptimizer, startEpoch,
    config["training"]["epochs"], config["training"]["warmupEpochs"],
    config["optimizer"]["settings"]["lr"])
197 
198         try:
199             for epoch in range(startEpoch, config["training"]["classifierEpochs"]):
200                 TrainClassifierEpoch(classifier, device, dataset, classifierOptimizer,
    classifierCheckpointer, epoch, config["training"]["classifierEpochs"],
    config["useHalfPrecision"])
201 
202                 if config["training"]["evaluateEveryNEpochs"] != 0 and epoch %
    config["training"]["evaluateEveryNEpochs"] == 0:
203                     testResults = TestEpoch(classifier, device, dataset,
    config["useHalfPrecision"])
204                     statistics.append((*testResults, epoch))
205 
206             lrScheduler.step()
207         except KeyboardInterrupt:
208             pass
209     elif config["mode"] == "classtrain":
210         classifier.setAllowTrainingEncoder() #Before classifier.trainableParameters()
211         classifierOptimizer = getattr(optim,
    config["optimizer"]["name"])(classifier.trainableParameters(),
    **config["optimizer"]["settings"])
212 
213         startEpoch = 0
214         lrScheduler = Util.WarmupCosineScheduler(classifierOptimizer, startEpoch,
    config["training"]["epochs"], config["training"]["warmupEpochs"],
    config["optimizer"]["settings"]["lr"])
215 
216         try:
217             for epoch in range(startEpoch, config["training"]["classifierEpochs"]):
218                 TrainClassifierEpoch(classifier, device, dataset, classifierOptimizer,
    classifierCheckpointer, epoch, config["training"]["classifierEpochs"],
    config["useHalfPrecision"])
219 
220                 if config["training"]["evaluateEveryNEpochs"] != 0 and epoch %
    config["training"]["evaluateEveryNEpochs"] == 0:
```

```
221                     testResults = TestEpoch(classifier, device, dataset,
    config["useHalfPrecision"])
222                     statistics.append((*testResults, epoch))
223
224             lrScheduler.step()
225         except KeyboardInterrupt:
226             pass
227     else:
228         raise NotImplementedError
229
230     if config["printStatistics"]:
231         computeCost = byol.getForwardComputeCost()
232         print("Multiplies:", computeCost[0])
233         print("Adds:", computeCost[1])
234         print("Memory:", computeCost[2] * (2 if config["useHalfPrecision"] else 4))
235
236     if len(statistics):
237         if config["printStatistics"]:
238             Util.PlotStatistics(statistics)
239
240         print("Final accuracy:", statistics[-1][1])
241
242 if __name__ == "__main__":
243     main()
```

E.9. Model.py

```
1  import math
2  import numpy as np
3
4  import torch
5  import torch.nn as nn
6  from torch.nn.functional import relu, max_pool2d, log_softmax, nll_loss, normalize, relu6,
       adaptive_avg_pool2d
7  from itertools import chain
8
9  import KRIAInterface
10 import QNN
11
12 def relu1(x, inplace=False):
13     return torch.clamp(x, 0, 1)
14
15 class ReLU1(nn.Module):
16     def __init__(self):
17         super(ReLU1, self).__init__()
18
19     def forward(self, x):
20         return relu1(x)
21
22 reluFunctionToUse=relu
23 reluModuleToUse=nn.ReLU
24
25 def SetUseReLU1(use):
26     global reluFunctionToUse
27     global reluModuleToUse
28     if use:
29         reluFunctionToUse = relu1
30         reluModuleToUse = ReLU1
31     else:
32         reluFunctionToUse = relu
33         reluModuleToUse = nn.ReLU
34
35 def GetLinearComputeCost(inputSize, outputSize, bias):
36     if bias:
37         inputSize += 1
38
39     multiplies = inputSize * outputSize
40     adds = outputSize * (inputSize - 1)
41     params = inputSize * outputSize
42     return np.array((multiplies, adds, params))
```

```
43
44 def GetConvolutionalComputeCost(inputDims, inputChannels, outputChannels, kernelSize, stride=1):
45     perPixel = np.array((kernelSize ** 2, kernelSize ** 2 + 1))
46
47     pixels = (inputDims[0] - (kernelSize - 1))/stride * (inputDims[1] - (kernelSize - 1))/stride
48
49     totalMA = perPixel * pixels * inputChannels * outputChannels
50     params = kernelSize ** 2 * inputChannels * outputChannels
51     return np.concatenate((totalMA, [params]))
52
53 def GetBatchNormComputeCost(size):
54     return np.array((0, 0, 0))
55
56 class MLP(nn.Module):
57     def __init__(self, inputSize, hiddenSize, outputSize, batchNorm, dtype, quantization):
58         super(MLP, self).__init__()
59
60         self.hiddenLayer = nn.Linear(inputSize, hiddenSize, bias=True, dtype=dtype)
61         self.outputLayer = nn.Linear(hiddenSize, outputSize, bias=False, dtype=dtype)
62         self.batchNorm = nn.BatchNorm1d(hiddenSize, **batchNorm)
63
64         self.quantizationEnabled = quantization["enabled"]
65
66         self.inputSize = inputSize
67         self.hiddenSize = hiddenSize
68         self.outputSize = outputSize
69
70     def forward(self, x):
71         x = self.hiddenLayer(x)
72         if self.quantizationEnabled:
73             x = QNN.quantize(x)
74         x = self.batchNorm(x)
75         if self.quantizationEnabled:
76             x = QNN.quantize(x)
77         x = reluFunctionToUse(x, inplace=True)
78         x = self.outputLayer(x)
79         if self.quantizationEnabled:
80             x = QNN.quantize(x)
81         return x
82         #return self.outputLayer(reluFunctionToUse(self.hiddenLayer(x), inplace=True))
83
84     def getOutputSize(self):
85         return self.outputSize
86
87     def getInputSize(self):
88         return self.inputSize
89
90     def getHiddenSize(self):
91         return self.hiddenSize
92
93     def getComputeCost(self):
94         return GetLinearComputeCost(self.inputSize, self.hiddenSize, True) +\
95             GetLinearComputeCost(self.hiddenSize, self.outputSize, False) +\
96             GetBatchNormComputeCost(self.hiddenSize)
97
98 class Classifier(nn.Module):
99     def __init__(self, classCount, hiddenSize, encoder, encoderName, batchNorm, dtypeName,
    quantization):
100         super(Classifier, self).__init__()
101
102         dtype = getattr(torch, dtypeName)
103
104         QNN.QuantizeTensor.nb = quantization["nb"]
105         QNN.QuantizeTensor.nf = quantization["nf"]
106         self.quantizationEnabled = quantization["enabled"]
107
108         self.encoder = globals()[encoderName](dtype=dtype, batchConfig=batchNorm,
    quantization=quantization, **encoder)
109         # self.outputLayer = MLP(self.encoder.getOutputSize(), hiddenSize, classCount,
    batchNorm=batchNorm, dtype=dtype, quantization=quantization)
110         self.outputLayer = nn.Linear(self.encoder.getOutputSize(), classCount, dtype=dtype)
```

```
111
112         for param in self.encoder.parameters():
113             param.requires_grad = False
114
115         self.allowTrainingEncoder = False
116
117     def setAllowTrainingEncoder(self):
118         for param in self.encoder.parameters():
119             param.requires_grad = True
120
121         self.allowTrainingEncoder = True
122
123     def forward(self, x):
124
125         if self.allowTrainingEncoder:
126             if self.quantizationEnabled:
127                 x = QNN.quantize(x)
128             encoded = self.encoder(x)
129         else:
130             with torch.no_grad():
131                 if self.quantizationEnabled:
132                     x = QNN.quantize(x)
133                 encoded = self.encoder(x).detach()
134
135         return log_softmax(self.outputLayer(encoded), dim=1)
136
137     def loss(self, x, target):
138         return nll_loss(self(x), target)
139
140     def predict(self, x):
141         output = self(x)
142         prediction = output.argmax(dim=1, keepdim=True)
143         return output, prediction
144
145     def predictionLoss(self, x, target):
146         output = self(x)
147         prediction = output.argmax(dim=1, keepdim=True)
148         loss = nll_loss(output, target)
149         return loss, output, prediction
150
151     def copyEncoderFromBYOL(self, byol):
152         for classifierParam, onlineParam in zip(self.encoder.parameters(),
    byol.onlineEncoderParameters()):
153             classifierParam.data = onlineParam.data
154
155     def trainableParameters(self):
156         if self.allowTrainingEncoder:
157             return self.parameters()
158         else:
159             return self.outputLayer.parameters()
160
161     def getComputeCost(self):
162         return self.encoder.getComputeCost() + self.outputLayer.getComputeCost()
163
164 class GenericEncoder(nn.Module):
165     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization, channels):
166         super(GenericEncoder, self).__init__()
167
168         self.imageDims = imageDims
169         self.channels = channels
170
171         self.quantizationEnabled = quantization["enabled"]
172         self.useCustomConv = quantization["useCustomConv"]
173
174         sequence = []
175         lastChannelCount = imageChannels
176         computeCost = np.zeros((3,))
177         currentImageDims = list(imageDims)
178
179         for channel in channels:
180             if self.useCustomConv:
```

```
181                 sequence.append(KRIAInterface.Conv2D_3x3(lastChannelCount, channel, dtype=dtype))
182             else:
183                 sequence.append(nn.Conv2d(lastChannelCount, channel, 3, 1, dtype=dtype))
184             if self.quantizationEnabled:
185                 sequence.append(QNN.QuantizeModel())
186 
187             sequence.append(nn.BatchNorm2d(channel, **batchConfig))
188             if self.quantizationEnabled:
189                 sequence.append(QNN.QuantizeModel())
190 
191             sequence.append(reluModuleToUse())
192 
193             computeCost += GetConvolutionalComputeCost(currentImageDims, lastChannelCount,
    channel, 3)
194 
195             currentImageDims[0] -= 1
196             currentImageDims[1] -= 1
197             lastChannelCount = channel
198 
199         self.computeCost = computeCost
200 
201         self.sequence = nn.Sequential(*sequence)
202 
203     def getOutputSize(self):
204         return self.channels[-1] * (self.imageDims[0] - 2 * len(self.channels)) // 2 *
    (self.imageDims[1] - 2 * len(self.channels)) // 2
205 
206     def forward(self, x):
207         x = self.sequence(x)
208         x = max_pool2d(x, 2)
209         return torch.flatten(x, 1)
210 
211     def getComputeCost(self):
212         return self.computeCost
213 
214 class Encoder(GenericEncoder):
215     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization,
    outputChannels=64, hiddenChannels=32):
216         super(Encoder, self).__init__(imageDims, imageChannels, batchConfig, dtype, quantization,
    channels=[hiddenChannels, outputChannels])
217 
218 class EncoderType1(GenericEncoder):
219     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization):
220         super(EncoderType1, self).__init__(imageDims, imageChannels, batchConfig, dtype,
    quantization, channels=[2, 4])
221 
222 class EncoderType2(GenericEncoder):
223     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization):
224         super(EncoderType2, self).__init__(imageDims, imageChannels, batchConfig, dtype,
    quantization, channels=[4, 8])
225 
226 class EncoderType3(GenericEncoder):
227     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization):
228         super(EncoderType3, self).__init__(imageDims, imageChannels, batchConfig, dtype,
    quantization, channels=[4, 8, 12])
229 
230 class EncoderType4(GenericEncoder):
231     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization):
232         super(EncoderType4, self).__init__(imageDims, imageChannels, batchConfig, dtype,
    quantization, channels=[6, 12, 18, 24, 30])
233 
234 class MobileNetV2Block(nn.Module):
235     def __init__(self, imageDims, inputChannels, outputChannels, batchConfig, dtype, quantization,
    expansionFactor=6, downSample=False):
236         super(MobileNetV2Block, self).__init__()
237 
238         self.downSample = downSample
239         self.shortcut = (not downSample) and (inputChannels == outputChannels)
240         self.imageDims = [*imageDims]
241 
242         internalChannels = inputChannels * expansionFactor
```

```
243
244         self.inputChannels = inputChannels
245         self.internalChannels = internalChannels
246
247         self.quantizationEnabled = quantization["enabled"]
248
249         self.conv1 = nn.Conv2d(inputChannels, internalChannels, 1, bias=False, dtype=dtype)
250         self.bn1 = nn.BatchNorm2d(internalChannels, **batchConfig, dtype=dtype)
251         self.conv2 = nn.Conv2d(internalChannels, internalChannels, 3, stride=2 if downSample else
    1, groups=internalChannels, bias=False, padding=1, dtype=dtype)
252         self.bn2 = nn.BatchNorm2d(internalChannels, **batchConfig, dtype=dtype)
253         self.conv3 = nn.Conv2d(internalChannels, outputChannels, 1, bias=False, dtype=dtype)
254         self.bn3 = nn.BatchNorm2d(outputChannels, **batchConfig, dtype=dtype)
255
256     def forward(self, x):
257         y = self.conv1(x)
258         if self.quantizationEnabled:
259             y = QNN.quantize(y)
260         y = self.bn1(y)
261         if self.quantizationEnabled:
262             y = QNN.quantize(y)
263         y = relu1(y, inplace=True)
264         y = self.conv2(y)
265         if self.quantizationEnabled:
266             y = QNN.quantize(y)
267         y = self.bn2(y)
268         if self.quantizationEnabled:
269             y = QNN.quantize(y)
270         y = relu1(y, inplace=True)
271         y = self.conv3(y)
272         if self.quantizationEnabled:
273             y = QNN.quantize(y)
274         y = self.bn3(y)
275         if self.quantizationEnabled:
276             y = QNN.quantize(y)
277
278         if self.shortcut:
279             return y + x
280         else:
281             return y
282
283     def getComputeCost(self):
284         hiddenDims = [*self.imageDims]
285         if self.downSample:
286             hiddenDims[0] /= 2
287             hiddenDims[1] /= 2
288
289         return GetConvolutionalComputeCost(self.imageDims, self.inputChannels,
    self.internalChannels, 1) +\
290             GetConvolutionalComputeCost(self.imageDims, self.internalChannels,
    self.internalChannels, 3, stride=2 if self.downSample else 1) +\
291             GetConvolutionalComputeCost(hiddenDims, self.internalChannels, self.outputChannels, 1)
292
293 class MobileNetV2(nn.Module):
294     def __init__(self, dtype, imageDims, imageChannels, batchConfig, quantization):
295         super(MobileNetV2, self).__init__()
296
297         imageDims = [*imageDims]
298
299         self.conv0 = nn.Conv2d(imageChannels, 32, 3, padding=1, bias=False)
300         self.bn0 = nn.BatchNorm2d(32)
301
302         blocks = [
303             MobileNetV2Block(imageDims, 32, 16, batchConfig, dtype, quantization,
    expansionFactor=1, downSample=False),
304             MobileNetV2Block(imageDims, 16, 24, batchConfig, dtype, quantization,
    downSample=False),
305             MobileNetV2Block(imageDims, 24, 24, batchConfig, dtype, quantization),
306             MobileNetV2Block(imageDims, 24, 32, batchConfig, dtype, quantization,
    downSample=False),
307             MobileNetV2Block(imageDims, 32, 32, batchConfig, dtype, quantization),
```

```
308             MobileNetV2Block(imageDims, 32, 32, batchConfig, dtype, quantization),
309             MobileNetV2Block(imageDims, 32, 64, batchConfig, dtype, quantization, downSample=True),
310             MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
311             MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
312             MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
313             MobileNetV2Block(imageDims, 64, 96, batchConfig, dtype, quantization,
    downSample=False),
314             MobileNetV2Block(imageDims, 96, 96, batchConfig, dtype, quantization),
315             MobileNetV2Block(imageDims, 96, 96, batchConfig, dtype, quantization),
316             MobileNetV2Block(imageDims, 96, 160, batchConfig, dtype, quantization,
    downSample=True),
317             MobileNetV2Block(imageDims, 160, 160, batchConfig, dtype, quantization),
318             MobileNetV2Block(imageDims, 160, 160, batchConfig, dtype, quantization),
319             MobileNetV2Block(imageDims, 160, 320, batchConfig, dtype, quantization,
    downSample=False)
320         ]
321
322         for block in blocks:
323             if block.downSample:
324                 imageDims[0] /= 2
325                 imageDims[1] /= 2
326                 block.imageDims = imageDims
327
328         self.blocks = nn.Sequential(*blocks)
329
330         # last conv layers and fc layer
331         self.conv1 = nn.Conv2d(320, 1280, 1, bias=False)
332         self.bn1 = nn.BatchNorm2d(1280)
333
334     def getOutputSize(self):
335         return 1280
336
337     def forward(self, x):
338         y = relu1(self.bn0(self.conv0(x)))
339         y = self.blocks(y)
340         y = relu1(self.bn1(self.conv1(y)))
341         y = adaptive_avg_pool2d(y, 1)
342         y = torch.squeeze(torch.squeeze(y, -1), -1)
343         return y
344
345     def getComputeCost(self):
346         computeCost = GetConvolutionalComputeCost(self.imageDims, self.imageChannels, 32, 3)
347
348         for block in self.blockList:
349             computeCost += block.getComputeCost()
350
351         computeCost += GetConvolutionalComputeCost(self.lastLayerImageDims, 320, 1280, 1)
352
353         return computeCost
354
355 class MobileNetV2Short(nn.Module):
356     def __init__(self, imageDims, imageChannels, batchConfig, dtype, quantization):
357         super(MobileNetV2Short, self).__init__()
358
359         self.imageDims = [*imageDims]
360         self.imageChannels = imageChannels
361
362         self.conv0 = nn.Conv2d(imageChannels, 32, 3, padding=1, bias=False)
363         self.bn0 = nn.BatchNorm2d(32)
364
365         self.blockList = [
366             MobileNetV2Block(imageDims, 32, 16, batchConfig, dtype, quantization,
    expansionFactor=1, downSample=False),
367             MobileNetV2Block(imageDims, 16, 24, batchConfig, dtype, quantization,
    downSample=False),
368             # MobileNetV2Block(imageDims, 24, 24, batchConfig, dtype, quantization),
369             MobileNetV2Block(imageDims, 24, 32, batchConfig, dtype, quantization,
    downSample=False),
370             # MobileNetV2Block(imageDims, 32, 32, batchConfig, dtype, quantization),
371             # MobileNetV2Block(imageDims, 32, 32, batchConfig, dtype, quantization),
372             MobileNetV2Block(imageDims, 32, 64, batchConfig, dtype, quantization, downSample=True),
```

```
373             # MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
374             # MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
375             # MobileNetV2Block(imageDims, 64, 64, batchConfig, dtype, quantization),
376             MobileNetV2Block(imageDims, 64, 96, batchConfig, dtype, quantization,
    downSample=False),
377             # MobileNetV2Block(imageDims, 96, 96, batchConfig, dtype, quantization),
378             # MobileNetV2Block(imageDims, 96, 96, batchConfig, dtype, quantization),
379             MobileNetV2Block(imageDims, 96, 160, batchConfig, dtype, quantization,
    downSample=True),
380             # MobileNetV2Block(imageDims, 160, 160, batchConfig, dtype, quantization),
381             # MobileNetV2Block(imageDims, 160, 160, batchConfig, dtype, quantization),
382             MobileNetV2Block(imageDims, 160, 320, batchConfig, dtype, quantization,
    downSample=False)
383         ]
384 
385         imageDims = [*imageDims]
386         for block in self.blockList:
387             if block.downSample:
388                 imageDims[0] /= 2
389                 imageDims[1] /= 2
390                 block.imageDims = [*imageDims]
391 
392         self.blocks = nn.Sequential(*self.blockList)
393 
394         self.lastLayerImageDims = imageDims
395 
396         # last conv layers and fc layer
397         self.conv1 = nn.Conv2d(320, 1280, 1, bias=False, dtype=dtype)
398         self.bn1 = nn.BatchNorm2d(1280)
399 
400     def getOutputSize(self):
401         return 1280
402 
403     def forward(self, x):
404         y = relu1(self.bn0(self.conv0(x)))
405         y = self.blocks(y)
406         y = relu1(self.bn1(self.conv1(y)))
407         y = adaptive_avg_pool2d(y, 1)
408         y = torch.squeeze(torch.squeeze(y, -1), -1)
409         return y
410 
411     def getComputeCost(self):
412         computeCost = GetConvolutionalComputeCost(self.imageDims, self.imageChannels, 32, 3)
413 
414         for block in self.blockList:
415             computeCost += block.getComputeCost()
416 
417         computeCost += GetConvolutionalComputeCost(self.lastLayerImageDims, 320, 1280, 1)
418 
419         return computeCost
420 
421 class BYOL(nn.Module):
422     def __init__(self, emaScheduler, encoderName, predictor, projector, encoder, batchNorm,
    dtypeName, quantization):
423         super(BYOL, self).__init__()
424 
425         self.emaScheduler = emaScheduler
426 
427         dtype = getattr(torch, dtypeName)
428 
429         QNN.QuantizeTensor.nb = quantization["nb"]
430         QNN.QuantizeTensor.nf = quantization["nf"]
431         self.quantizationEnabled = quantization["enabled"]
432         self.weightQuantizationEnabled = quantization["quantizeWeights"]
433 
434         self.onlineEncoder = globals()[encoderName](dtype=dtype, batchConfig=batchNorm,
    quantization=quantization, **encoder)
435         self.targetEncoder = globals()[encoderName](dtype=dtype, batchConfig=batchNorm,
    quantization=quantization, **encoder)
436         self.onlineProjector = MLP(dtype=dtype, inputSize=self.onlineEncoder.getOutputSize(),
    batchNorm=batchNorm, quantization=quantization, **projector)
```

```
437         self.targetProjector = MLP(dtype=dtype, inputSize=self.targetEncoder.getOutputSize(),
    batchNorm=batchNorm, quantization=quantization, **projector)
438         self.predictor = MLP(dtype=dtype, inputSize=self.onlineProjector.getOutputSize(),
    outputSize=self.targetProjector.getOutputSize(), batchNorm=batchNorm,
    quantization=quantization, **predictor)
439 
440         # Make sure the target network starts out the same as the online network
441         for onlineParam, targetParam in zip(self.onlineParameters(), self.targetParameters()):
442             targetParam.requires_grad = False
443             targetParam.data = onlineParam.data
444 
445         if self.emaScheduler.getTau() == 0:
446             print("Using SimSiam")
447 
448     def forward(self, dataView1, dataView2):
449         # dimensions of dataView1,2: [batchSize, channelCount, imageWidth, imageHeight]
450 
451         if self.quantizationEnabled:
452             with torch.no_grad():
453                 dataView1 = QNN.quantize(dataView1)
454                 dataView2 = QNN.quantize(dataView2)
455 
456         # Standard BYOL approach
457         if self.emaScheduler.getTau() != 0:
458             image1OnlineEncoded = self.onlineEncoder(dataView1)
459             image1Online = self.onlineProjector(image1OnlineEncoded)
460             image1Predicted = self.predictor(image1Online)
461             with torch.no_grad():
462                 image1Target = self.targetProjector(self.targetEncoder(dataView1)).detach()
463 
464             image2OnlineEncoded = self.onlineEncoder(dataView2)
465             image2Online = self.onlineProjector(image2OnlineEncoded)
466             image2Predicted = self.predictor(image2Online)
467             with torch.no_grad():
468                 image2Target = self.targetProjector(self.targetEncoder(dataView2)).detach()
469 
470         # Simplified SimSiam approach
471         else:
472             image1OnlineEncoded = self.onlineEncoder(dataView1)
473             image1Online = self.onlineProjector(image1OnlineEncoded)
474             image1Predicted = self.predictor(image1Online)
475             with torch.no_grad():
476                 image1Target = image1Online.detach()
477 
478             image2OnlineEncoded = self.onlineEncoder(dataView2)
479             image2Online = self.onlineProjector(image2OnlineEncoded)
480             image2Predicted = self.predictor(image2Online)
481             with torch.no_grad():
482                 image2Target = image2Online.detach()
483 
484         def MSELoss(a, b):
485             return torch.mean(torch.square(a - b))
486 
487         def RegressionLoss(a, b):
488             return MSELoss(normalize(a, dim=1), normalize(b, dim=1))
489 
490         onlineLoss = (RegressionLoss(image1Predicted, image2Target) +
    RegressionLoss(image2Predicted, image1Target)) / 2
491 
492         if torch.isnan(onlineLoss).any():
493             breakpoint()
494 
495         return onlineLoss
496 
497     def stepEMA(self):
498         tau = self.emaScheduler.getTau()
499 
500         # Standard BYOL approach
501         if tau != 0:
502             for onlineParam, targetParam in zip(self.onlineParameters(), self.targetParameters()):
503                 targetParam.data = targetParam.data + (onlineParam.data - targetParam.data) * (1.0
```

```
    - tau)
504                 targetParam.requires_grad = False
505
506         # Simplified SimSiam approach
507         else:
508             for onlineParam, targetParam in zip(self.onlineParameters(), self.targetParameters()):
509                 targetParam.data = onlineParam.data
510                 targetParam.requires_grad = False
511
512     def trainableParameters(self):
513         return chain(
514             self.onlineEncoder.parameters(),
515             self.onlineProjector.parameters(),
516             self.predictor.parameters(),
517         )
518
519     def onlineParameters(self):
520         return chain(
521             self.onlineEncoder.parameters(),
522             self.onlineProjector.parameters()
523         )
524
525     def targetParameters(self):
526         return chain(
527             self.targetEncoder.parameters(),
528             self.targetProjector.parameters()
529         )
530
531     def onlineEncoderParameters(self):
532         return self.onlineEncoder.parameters()
533
534     def predictorParameters(self):
535         return self.predictor.parameters()
536
537     def getForwardComputeCost(self):
538         computeCost = 2 * (self.onlineEncoder.getComputeCost() +
    self.onlineProjector.getComputeCost() + self.predictor.getComputeCost() +
    self.targetEncoder.getComputeCost() + self.targetProjector.getComputeCost())
539         computeCost[2] /= 2
540         return computeCost
541
542     def quantizeParameters(self):
543         if self.quantizationEnabled and self.weightQuantizationEnabled:
544             for param in self.parameters():
545                 with torch.no_grad():
546                     param.data = QNN.quantize(param.data)
```

E.10. QNN.py

This file contains a (modified) function to quantize a PyTorch Tensor. Credits to Charlotte Frenkel, Institute of Neuroinformatics, 2021-2022.

```
1 #
2 # Charlotte Frenkel, Institute of Neuroinformatics, 2021-2022
3 #
4 # Credit for the surrogate gradient function: F. Zenke,
    https://github.com/fzenke/spytorch/blob/main/notebooks/SpyTorchTutorial1.ipynb
5
6 # The quant_fraction function is an addition to the original file contents
7 #
8
9 import math
10
11 import torch
12 import torch.nn as nn
13 from torch.autograd import Variable
14 from torch.nn.modules.utils import _pair
15 import torch.nn.functional as F
16 from torch.autograd import Function
17
18
```

```
def quant_one(W, nb):
    if nb == 1:
        return (W >= 0).float() * 2 - 1
    elif nb == 2:
        return torch.clamp(torch.floor(W), -1, 1)
    non_sign_bits = nb - 1
    m = pow(2, non_sign_bits)
    return torch.clamp(torch.floor(W * m), -m, m - 1) / m

def quant_fraction(W, nb, nfract):
    if nb == 1:
        return (W >= 0).float() * 2 - 1
    elif nb == 2:
        return torch.clamp(torch.floor(W), -1, 1)

    non_sign_bits = nb - 1

    f = pow(2, nfract)
    m = pow(2, non_sign_bits)

    return torch.clamp(torch.floor(W * f), -m, m - 1) / f

class SurrGradSpike(torch.autograd.Function):
    """
    Here we implement our spiking nonlinearity which also implements
    the surrogate gradient. By subclassing torch.autograd.Function,
    we will be able to use all of PyTorch's autograd functionality.
    Here we use the normalized negative part of a fast sigmoid
    as this was done in Zenke & Ganguli (2018).
    """

    scale = 100.0  # controls steepness of surrogate gradient

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we compute a step function of the input Tensor
        and return it. ctx is a context object that we use to stash information which
        we need to later backpropagate our error signals. To achieve this we use the
        ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        out = torch.zeros_like(input)
        out[input > 0] = 1.0
        return out

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor we need to compute the
        surrogate gradient of the loss with respect to the input.
        Here we use the normalized negative part of a fast sigmoid
        as this was done in Zenke & Ganguli (2018).
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad = grad_input / (SurrGradSpike.scale * torch.abs(input) + 1.0) ** 2
        return grad


class QuantizeTensor(torch.autograd.Function):
    nb = 8
    nf = 4

    @staticmethod
    def forward(ctx, input):
        return quant_fraction(input, QuantizeTensor.nb, QuantizeTensor.nf)

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output
```

```

class QuantizeModel(nn.Module):
    def forward(self, input):
        return quantize(input)

spikeAct = SurrGradSpike.apply
quantize = QuantizeTensor.apply
```

E.11. Server.py

```
import torch
import torch.optim as optim

import time

import Config
import Model
import Communication
import Checkpointer
import Dataset
import Util
import main as mainModule
class Server:

    class ServerClient:
        def __init__(self, comm, addr, model):
            self.comm = comm
            self.name = str(addr)
            self.model = model
            self.hasReceivedModel = False

    def __init__(self, device, config):
        self.device = device
        self.config = config
        self.checkpointer = Checkpointer.Checkpointer(**config["checkpointer"])

        self.classifier = Model.Classifier(**config["classifier"]).to(device)
        self.classifierOptimizer = getattr(optim,
    config["optimizer"]["name"])(self.classifier.trainableParameters(),
    **config["optimizer"]["settings"])
        self.emaScheduler = Util.EMAScheduler(**config["EMA"])

        self.dataset = Dataset.Dataset(**config["dataset"])

        self.communicationServer = Communication.Server()

        self.clients = []
        self.currentModel = Model.BYOL(self.emaScheduler, **self.config["BYOL"])

    def bind(self, ip, port):

        self.communicationServer.bind(ip, port)

    def listenForClients(self):

        while True:
            comm, addr = self.communicationServer.tryAcceptClient()
            if comm is None:
                break
            else:
                print(f"Accepted client at {addr}")

                client = self.ServerClient(comm, addr, Model.BYOL(self.emaScheduler,
    **self.config["BYOL"]))
                self.clients.append(client)
                self.updateClientCommunication(client)

    def updateClientCommunication(self, client):
        if client.comm.isDataReady():
            message = client.comm.receiveMessage()
```

```
58              if message == "stop": # Clients wants to disconnect
59                  print(f"Closing client {client.name}")
60                  self.clients.remove(client)
61                  client.comm.sendMessage("stopAcknowledged")
62                  client.comm.close()
63              elif message == "requestSend": # Should send current model to client
64                  print(f"Sending model to client {client.name}")
65                  client.comm.sendModel(self.currentModel)
66              elif message == "update":
67                  print(f"Receiving model from client {client.name}")
68                  client.comm.receiveModel(client.model)
69                  client.hasReceivedModel = True
70              else:
71                  print(f"Received unknown message {message} from client {client.name}")
72
73      def run(self):
74
75          shouldStop = False
76          while not shouldStop:
77
78              print("Waiting for updated models...")
79              while True: # Communicate with clients until all clients have sent an updated model
80                  self.listenForClients()
81                  for client in self.clients:
82                      self.updateClientCommunication(client)
83
84                  clientsWithModelCount = 0
85                  for client in self.clients:
86                      if client.hasReceivedModel:
87                          clientsWithModelCount += 1
88
89                  if len(self.clients) == 0:
90                      print("Waiting for clients...")
91                  elif clientsWithModelCount == len(self.clients):
92                      break
93
94                  time.sleep(1)
95
96              print("Averaging")
97
98              modelParameters = [list((client.model.parameters())) for client in self.clients]
99
100             for idx, param in enumerate(self.currentModel.parameters()):
101                 param.data = torch.mean(torch.stack([modelParameters[modelIdx][idx].data for
    modelIdx in range(len(modelParameters))]), dim=0)
102
103             # Clear the received models to indicate we processed them
104             for client in self.clients:
105                 client.hasReceivedModel = False
106
107             # Communicate with clients to send the averaged model
108             time.sleep(1) # Give clients time to process
109             for client in self.clients:
110                 self.updateClientCommunication(client)
111
112             print("Saving model")
113             self.checkpointer.saveCheckpoint(self.currentModel, None)
114
115             self.classifier.copyEncoderFromBYOL(self.currentModel)
116             for i in range(self.config["server"]["classifierTrainEpochs"]):
117                 mainModule.TrainClassifierEpoch(self.classifier, self.device, self.dataset,
    self.classifierOptimizer, self.checkpointer, -1, -1, self.config["useHalfPrecision"])
118             mainModule.TestEpoch(self.classifier, self.device, self.dataset,
    self.config["useHalfPrecision"])
119
120 def main():
121     config = Config.GetConfig()
122     config["device"] = "cpu"
123
124     torch.manual_seed(0)
125     device = Util.GetDeviceFromConfig(config)
```

```
126
127     Model.SetUseReLU1(config["useReLU1"])
128
129     server = Server(device, config)
130     server.bind("localhost", 1234)
131     server.run()
132
133 if __name__ == "__main__":
134     main()
```

E.12. Util.py

This file contains a few utility functions.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import torch
4  import math
5
6  def PlotImage(image):
7      plt.imshow(np.squeeze(torch.movedim(image * 0.5 + 0.5, 0, 2).detach().cpu().numpy()))
8      plt.show()
9
10 def PlotStatistics(statistics):
11     statistics = np.array(statistics)
12
13     print("Statistics:", statistics)
14
15     loss = statistics[:,0]
16     accuracy = statistics[:,1]
17     epochs = statistics[:,2]
18
19     plt.figure(figsize=(10,10))
20     plt.plot(epochs, loss, label="Loss")
21     plt.plot(epochs, accuracy, label="Accuracy")
22     plt.grid()
23     plt.xlabel("Epochs #")
24     plt.ylim(0)
25     plt.legend()
26     plt.show()
27
28 def GetDeviceFromConfig(config):
29     deviceName = config["device"]
30     if deviceName == "cuda" and torch.cuda.is_available():
31         print("Using CUDA")
32         return torch.device("cuda")
33     else:
34         print("Using CPU")
35         return torch.device("cpu")
36
37 class WarmupCosineScheduler(torch.optim.lr_scheduler.LRScheduler):
38     def __init__(self, optimizer, startEpoch, epochCount, warmupEpochs, baseLearningRate):
39         self.baseLearningRate = baseLearningRate
40         self.epochCount = epochCount
41         self.warmupEpochs = warmupEpochs
42
43         super(WarmupCosineScheduler, self).__init__(optimizer)
44
45         for i in range(startEpoch):
46             self.step()
47
48     def get_lr(self):
49         afterWarmupEpoch = self._step_count - self.warmupEpochs
50         nonWarmupEpochCount = self.epochCount - self.warmupEpochs
51
52         if self._step_count < self.warmupEpochs:
53             return [self.baseLearningRate * self._step_count / self.warmupEpochs] # Linear warm-up
54         elif self._step_count < self.epochCount:
55             return [self.baseLearningRate * 0.5 * (1 + math.cos(math.pi * afterWarmupEpoch /
       nonWarmupEpochCount))]
56         else:
```

```
57             return [0]
58
59 class EMAScheduler:
60     def __init__(self, initialTau, epochCount, enableSchedule):
61         self.initialTau = initialTau
62         self.epochCount = epochCount
63         self.enableSchedule = enableSchedule
64
65         self.tau = self.initialTau
66
67     def step(self, currentEpoch):
68         if not self.enableSchedule:
69             return
70
71         if self.initialTau == 0:
72             self.tau = 0 # SimSiam
73         else:
74             self.tau = 1 - (1 - self.initialTau) * 0.5 * (1 + math.cos(math.pi * currentEpoch /
    self.epochCount))
75
76     def startStep(self, steps):
77         for i in range(steps):
78             self.step(i)
79
80     def getTau(self):
81         return self.tau
```