Impact of model architecture and data distribution on self-supervised federated learning

Karin Bergdahl

28th July 2022

Abstract

Data is a crucial resource for machine learning. But in many settings, such as in healthcare or on mobile devices, there are obstacles that make it difficult to utilize the available data. This data is often distributed between many clients and private, meaning that central storage of the data is inadvisable. Further, image data is often unlabeled and external labelling is impossible due to its private nature. This project aims to train and examine a self-supervised representation encoder on distributed and unlabeled image data. We create a federated implementation of the contrastive learning framework SimCLR and compare its performance to the traditional central version. We use federated averaging to create a federated implementation of SimCLR. Within the SimCLR framework, we test two different model types for the encoder (ResNet-18 and AlexNet). The encoders are trained in two different federated settings: i.i.d., where all clients have data from the same distribution, and non-i.i.d., where the client data distributions are completely disjoint. We also create a non-federated implementation trained on the same data, to compare the impact of federation on SimCLR. The quality of the representations is measured by the accuracy of a linear classifier trained on a small, labelled data set. We find that the best type of federated encoder has an average classifier accuracy of 67.0 % in the i.i.d. setting. This is only a small drop from the non-federated implementation, which reaches 69.0 %. However, the encoders trained in a non-i.i.d. setting have a lower average accuracy at 62.3 %. So, while a federated model has the capacity to perform on the level of a central one, a challenge in real world federated applications may be unbalanced data distributions.

Preface

This thesis project was carried out during the spring of 2022 at the Centre of Mathematical Sciences, Lund Univeristy, together with RISE. This thesis marks the (near) end of my time at LTH, studying Engineering Mathematics.

Of course, there are many people to thank after a project like this. Firstly, the deep learning group at RISE. Thank you for your support, knowledge and inspiring seminars. An especially big thanks to my co-supervisor Edvin Listo Zec, who made this project fun, interesting and much less stressful than I thought it would be. I would also like to thank my supervisor and examiner at the university, Carina Geldhauser and Alexandors Sopasakis. Thanks to my friends in π16 for answering all my questions about your own projects, for all your help during our studies, and for all the fun. And finally, thank you to my amazing partner Filip. Without you, I would never have gotten to my final year at all.

Karin Bergdahl
Lund 2022

Contents

1 Introduction

1.1 Background

Most machine learning (ML) methods need large amounts of data to train high performing models. Traditionally, all this data is stored in a central server. But there are many potential applications for ML where it is not possible to store all data in one place, usually because the data contains a lot of sensitive personal information.

During the last few years, *federated learning* (FL) has emerged as a method to learn from decentralized data while preserving data privacy. In FL, a model is sent to each client and trained there, then all client models are sent back and combined. This repeats until the model is sufficiently trained. By training the model locally at each client's device, there is no need for central data storage. FL has applications in fields such as mobile devices [1], healthcare [2, 3], autonomous driving and intelligent transport systems [4], and Internet of Things [5].

A challenge in some of these fields is that most of the available data is unlabeled. As a simple example, most mobile phones have hundreds of photos on them but very little information about what those photos depict. Users are unlikely to annotate their own images, and due to its private nature the data cannot be annotated centrally.

Self-supervised learning methods are a well established way to tackle problems where none or only a small portion of the data is labeled. Self-supervised learning aims to learn useful representations of complex data, such as images or sound, that can be used for other tasks. The networks used to create these representations are usually called encoders. We will try two different encoder architectures, to see how the representation quality is impacted by encoder design. *AlexNet* is a small CNN with only 5 layers, while *ResNet-18* is a much more complex structure.

But how can self-supervised learning be applied in a federated setting? Self-supervised federated learning (SSFL) would be very useful for real world applications, but it is still a relatively understudied field. This thesis will investigate one way to combine self-supervised and federated learning. We will work in the image domain, simulating a problem where we want to classify phone images. To do this, we will create a federated version of a popular self-supervised method for representation learning, SimCLR.

1.2 Aim

The aim of this work is to create federated implementations of SimCLR, using two different encoders, and

1. assess the impact of federation by comparing their performance to those of non-federated implementations.
2. compare performance between two federated data settings. One simple, with even label distribution between clients, and one more realistic, where the label distributions vary among clients.

2 Theory

2.1 Self-supervised learning

Usually, the data on phones has no labels. Annotating the data centrally is both expensive and often impossible when the data is private, and users usually do not annotate their own data. A common approach with unlabeled data is to do some form of *self-supervised learning*. There are many different frameworks for self-supervised learning, suitable for different data types and tasks. We will use SimCLR [6], which is described in more detail later in this chapter.

The aim of self-supervised representation learning is to transform the input $\boldsymbol{x}$ into a representation vector $\boldsymbol{h}$. We are working with images so the input $\boldsymbol{x}$ has a height H and width W. If it is a color image it is usually represented in RGB, 3 channels with integer values between 0 and 255. So, we have $\boldsymbol{x} \in \mathbb{Z}_{256}^{H \times W \times 3}$. We want to create some transformation that maps $\boldsymbol{x}$ to a representation of length l,

$$\boldsymbol{x} \to \boldsymbol{h} \in \mathbb{R}^l.$$

The representations $\boldsymbol{h}$ are then used to solve other *downstream tasks*, more efficiently than if we had used the inputs directly. In the ideal case, the representations contain important features of the data, and throw away less useful information.

This transformation will be created by a neural network called an *encoder*. The encoder is trained to preform some task that does not require labels. This *auxiliary task* can be tailored to the data type and what the representations will be used for. A common objective is that if we have similar inputs, their vector representations should be close in the representation space.

2.1.1 Semi-supervised learning

Related to self-supervised learning is *semi-supervised learning*. As the name suggests, this concerns ML that combines unsupervised and supervised learning. Imagine a case where we have access to a large number of unlabeled images, and a much smaller number of labeled images. We want to create a classifier, but the small, labeled dataset is not enough to get good results. If the data was "better" with clearer features, we could get a more accurate classifier.

So, we use the unlabeled data to train an encoder using self-supervised learning. The representations created by the encoder should bring out useful features for the classifier. We then compute the representations of the labeled images. These representations and their labels can be used to train the classifier in a supervised manner, with better results than if we had used the images directly.

2.1.2 SimCLR

SimCLR is a self-supervised learning framework for images presented by Ting Chen et al in 2020 [6]. The first component of the framework is *augmentation*. "Augmenting an image" is when you change it in some way, for example cropping or distort colors. By augmenting a photo in different ways, we get multiple images we know are "similar", since we use augmentations that preserve semantics.

Given two representations generated from different augmentations of the same photo, the goal of the network is to place the representations near each other. This will hopefully make it so images that are similar (but not originally the same photo) also have representations that are close in the representation space.

The augmentations used in SimCLR are random crop and resize, flipping, color distortion, blurring and grayscale transformation. Details on these augmentations can be found in Appendix A. The augmentations are randomly applied to an image $\boldsymbol{x}$ twice, yielding two augmented versions, $\tilde{\boldsymbol{x}}_i$ and $\tilde{\boldsymbol{x}}_j$. An example can be seen in Figure 2.1.

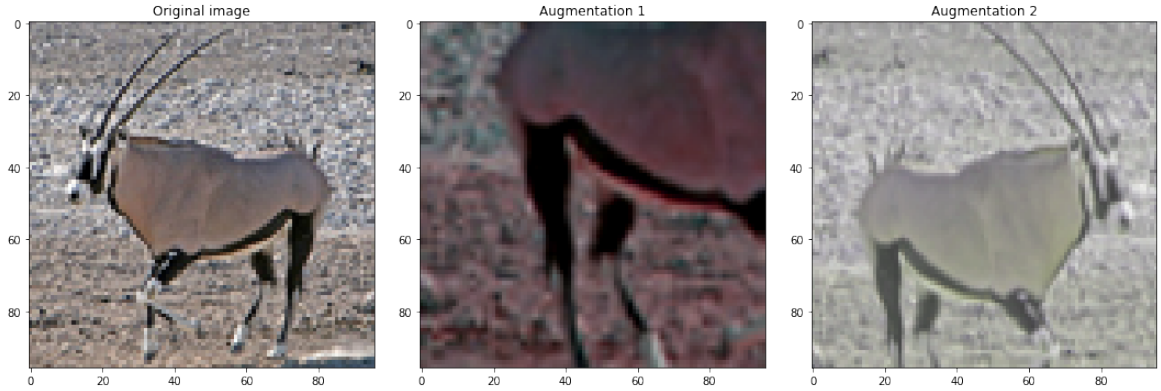


Figure 2.1: Example of augmentations.

The next component is the network, which is split into two parts. First is the *base encoder*, which creates the representations $\boldsymbol{h}_i$ that will be used later. This can be any type of network that suits the data type. Connected to that is the *projection head*, which is a small MLP that maps the representations $\boldsymbol{h}_i$ to embeddings $\boldsymbol{z}_i$. These embeddings will be used to calculate loss. Previous results show that adding a projection head improves the quality of the representations compared to when the representations $\boldsymbol{h}_i$ are used directly to calculate loss [6]. An illustration of the SimCLR structure can be seen in Figure 2.2.

The final component is the task that the network should solve. The aim of this network is to identify *positive pairs*, which augmented images come from the same original image. If we have a batch of N images, there will be $2N$ data points (two augmented versions of each image). This gives one positive and $2(N-1)$ negative examples for each point. We want each embedding $\boldsymbol{z}_i$ to be close to the embedding of its positive example $\boldsymbol{z}_j$, and far from all other embeddings.

This leads to the loss function that SimCLR uses, *normalized temperature-scaled cross entropy loss*, NT-Xent. To measure how "close" two embeddings are it uses cosine similarity, $\text{sim}(u, v) = u^T v / \|u\| \|v\|$. For a positive pair (i, j), the loss $\ell(\boldsymbol{z}_i, \boldsymbol{z}_j)$ is

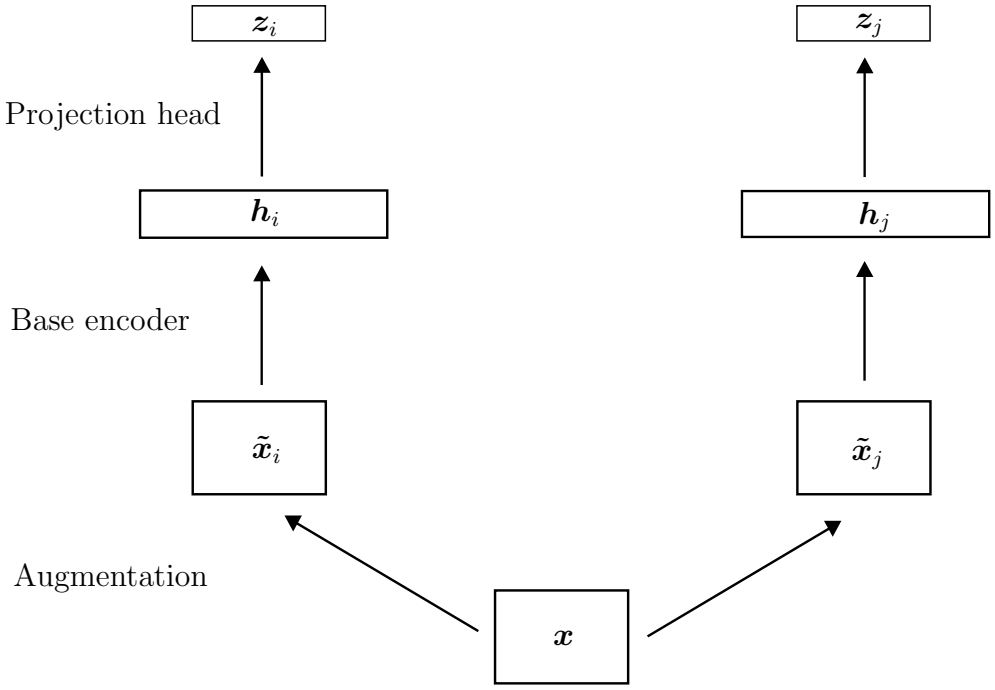


Figure 2.2: SimCLR structure

defined as

$$\ell(\boldsymbol{z}_i, \boldsymbol{z}_j) = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}.$$

Here $\mathbb{1}_{[k \neq i]}$ is 0 when $k = i$ and 1 otherwise. The temperature parameter τ can be tuned to adjust the impact of *hard negatives*, embeddings that is not in the positive pair but has a high similarity to $\boldsymbol{z}_i$. If τ is small, hard negatives will dominate in the denominator. When τ is large, all elements in the denominator are more similar. So, a τ below 1 leads to a loss that gives more emphasis to hard negatives than other examples.

Note that in a batch the loss is computed on all positive pairs, i.e both (i, j) and (j, i). To simplify notation forwards, we will define loss per input. For an input image $\boldsymbol{x}^n$ and network weights $\boldsymbol{w} \in \mathbb{R}^d$ that generate the embeddings $\boldsymbol{z}_i^n, \boldsymbol{z}_j^n$, we define $\ell(\boldsymbol{w}, \boldsymbol{x}^n)$ as

$$\ell(\boldsymbol{w}, \boldsymbol{x}^n) = \ell(\boldsymbol{z}_i^n, \boldsymbol{z}_j^n) + \ell(\boldsymbol{z}_j^n, \boldsymbol{z}_i^n).$$

2.2 Federated learning

In a traditional ML environment, the training data is stored in a central location. But phone data is often private, and storing it in a central location, vulnerable to attacks, is inadvisable. A solution could be to train a model directly on the client, but usually a single user does not have the amount of data necessary for a good model. Federated learning was proposed by McMahan et al. in 2017 as a solution to these challenges [1]. The idea is to have a central model that is trained locally on clients for a few epochs, then all clients communicate their updates, creating a new central model.

The basic steps of federated learning are as follows.

1. A central model is initialized.
2. The weights w of the central model is sent from the server to each client.
3. Each client trains the received model for a number of local epochs, E.
4. All the local models are sent back to the server. The new central model weights are computed as the average of the local model weights, weighted by the amount of data each client has.
5. Repeat step 2-4 (called a *communication round*) a number of times.

In federated learning, we aim to find the weights w that minimize the loss over all clients. If client k has N_k training samples, the total loss for that client is

$$L_k(\boldsymbol{w}) = \frac{1}{N_k} \sum_{n=1}^{N_k} \ell(\boldsymbol{w}, \boldsymbol{x}^n)$$

and the loss over all K clients is

$$L(\boldsymbol{w}) = \sum_{k=1}^{K} \frac{N_k}{N} L_k(\boldsymbol{w}) \quad \text{where} \quad N = \sum_{k=1}^{K} N_k.$$

Each client's loss is weighted by the number of samples that client has. The problem we are trying to solve can now be formulated as

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} L(\boldsymbol{w})$$

2.3 Convolutional Neural Networks

This project will make heavy use of *convolutional neural networks*, CNNs. This type of network has become very popular during the last two decades due to its high performance, especially on time series and image data. We will give a brief overview of CNNs. All the information comes from "The Deep Learning Book" by Goodfellow et al. [7], which we also recommend to any readers looking for a more in-depth explanation.

Convolutional neural networks get their name from the central operation in the network, the *convolution*. In two dimensions, the discrete convolution between an image I and a kernel K is defined as

$$(I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n).$$

In this definition the kernel is flipped relative to the image, such that an increase in the index of I means a decrease in the index of K. This makes convolution commutative,

which is important for theoretical computations and proof. But in practice *cross-correlation* (convolution without the kernel flipped) is often used, since it is easier to understand and makes no difference for the network performance. So, we will use

$$(I * K)(i, j) = \sum_m \sum_n I(m, n)K(i + m, j + n).$$

An illustration of convolution without flipping the kernel can be seen in Figure 2.3[1].

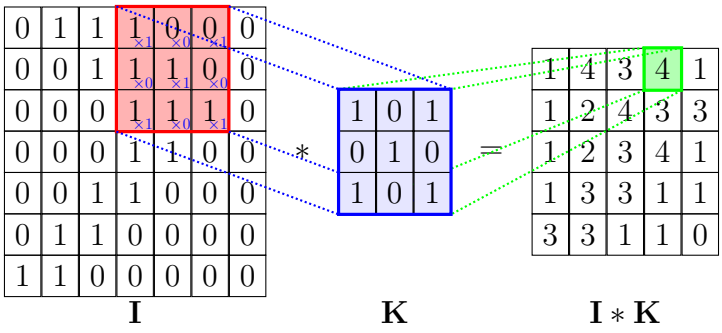


Figure 2.3: Example of convolution.

In images, the context of a pixel is important. Knowing the value of a single pixel does not tell us a lot. Further, knowing the values of ten randomly chosen pixels in the image will probably not tell us much either. But knowing that we few dark pixels right next to a few light pixels can indicate an edge. The most interesting thing to know about a pixel is its relation to the pixels around it, its immediate context.

In a traditional dense network, an output node is connected to every input node. However, in convolutional layers, each output node is only connected to a small region of input nodes. Consider Figure 2.3. The output node marked in green is only connected to a small portion of the input nodes, the ones marked in red. This property means that a focus on context is directly built into CNNs. A dense network would have to learn from scratch which pixel relations are important, but a CNN already knows.

A CNN is usually built of multiple types of layers. *Convolutional layers* consists of a number of kernels. Each kernel is convoluted with the input image, to produce an output called a feature map. These kernels are the parameters that the network trains.

An important parameter is the *stride*, which dictates how far the kernel moves between each operation. Figures 2.4 and 2.5 shows examples of convolutions with stride 1 and 2 respectively. The coloured outlines mark where the kernel is applied for the first three operations and their corresponding outputs. As we can see, the larger the stride, the smaller the output, since the kernel is applied fewer times.

Another technique used is in CNNs is *padding*, which is when additional pixels are added to the edges to the input, usually zeros. A matrix with one pixel wide padding is depicted in Figure 2.6. Padding is applied for two reasons. Firstly, convolution decreases the size of the matrix from input to output. Consider a convolution with a

[1]Image by Petar Veličković from `https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution`

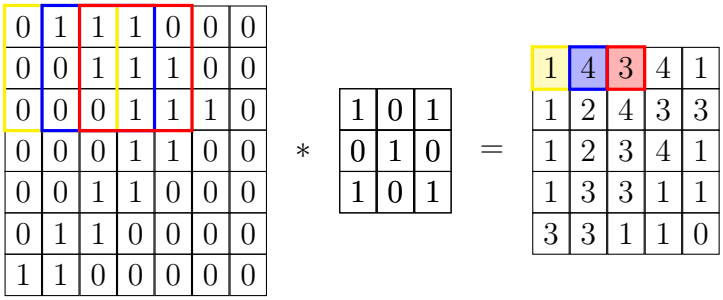


Figure 2.4: Convolution with stride 1. The coloured outlines show where the first 3 kernel operations are applied and the placement of the corresponding outputs.

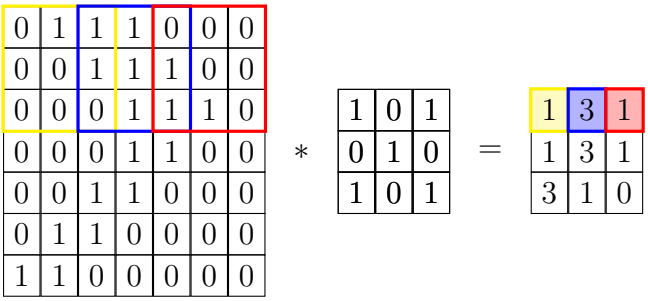


Figure 2.5: Convolution with stride 2. The coloured outlines show where the first 3 kernel operations are applied and the placement of the corresponding outputs.

3×3 kernel and stride 1, applied to a 7×7 matrix. The output will be of size 5×5. By adding padding to the input, it becomes a 9×9 matrix, which will give an output of size 7×7. The padding makes it so that the matrix size decreases slower throughout the convolutional layers.

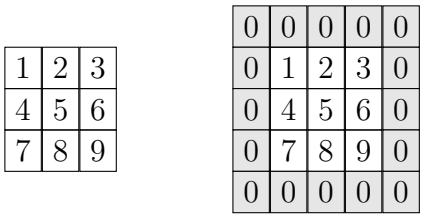


Figure 2.6: Matrix without and with padding, padding marked in gray.

The second effect of padding concerns the edges and corners of matrices. Consider the upper left and the central pixels of the non-padded matrix in Figure 2.6, the 1 and 5. There is only one 2×2 segment that contains the 1, but four that contains the 5. If we were to convolve this matrix with a 2×2 kernel, only one of the operations would operate on the 1. However, if we consider the padded matrix, there are now 4 different 2×2 segments that contain the 1. So, padding gives more opportunities for the convolutional layers to interact with the corners and edges of a matrix.

CNNs also use *pooling layers*. The most common are max pooling and average pooling, which output the maximum or average of an area of pixels respectively. An illustration of maxpooling is shown in Figure 2.7. Here, we use a 2×2 maxpool with stride 2. This means that the output will have the maximum pixel value in a 2×2 square in

the input.

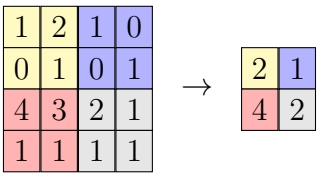


Figure 2.7: Illustration of maxpooling with a 2×2 kernel and stride 2.

Pooling can be viewed as a way of condensing information and making the network more robust to small shifts in the input. Imagine that the left matrix in Figure 2.7 is a feature map, an output from a convolution layer that detects some specific feature. The best match for this feature is located where the highest value is, the 4 in the third row, column 1. But the output from the maxpooling would have been the same even if the feature had shifted slightly down, to row 4. This is important when we work with images since we want the network to recognize when two images contain the same object even if they are not in the exact same location in the image.

3 Methods

3.1 Encoder

3.1.1 ResNet-18

For one version of the base encoder in the encoder network, we will use ResNet-18. ResNet stands for *residual neural network* and is an architecture first proposed by He et al. [8]. ResNet are a class of CNNs with *skip connections*. The skip connections are, like the name would suggest, connections that skip one or a few layers of the network. An illustration of a block with a skip connection can be seen in Figure 3.1.

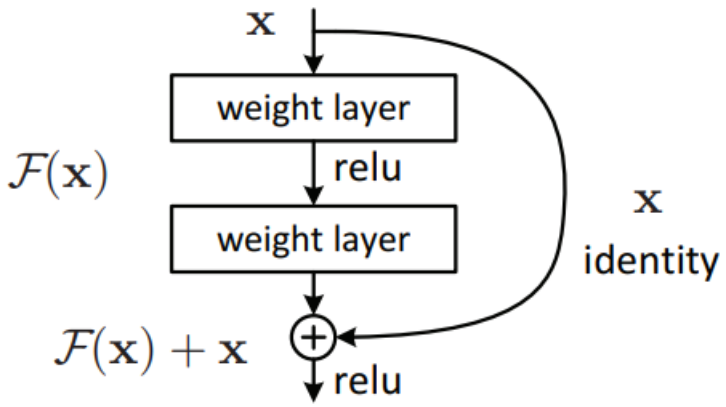


Figure 3.1: Illustration of a block with a skip connection that jumps over two layers. Image from [8].

The input $\boldsymbol{x}$ is not only sent to the first weight layer, but also added to the output from the block. The output $\mathcal{F}(\boldsymbol{x})$ from the intermediate layers is called a residual, and we can denote the whole mapping between input and output of the block as $\mathcal{H}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$.

In ResNet, the skip connections are intended to help with the *degradation problem*. In general, a deep network can learn complex functions more efficiently than a shallower one, but deeper networks are also harder to train. The degradation problem refers to the situation where deep networks perform worse during training than shallower counterparts. This is quite a common problem in deep learning. He et al. theorize that it is hard for a deep network to learn the optimal mapping $\mathcal{H}(\boldsymbol{x})$, but easier to learn the optimal residual $\mathcal{F}(\boldsymbol{x})$. They also show empirically that skip connections makes the network easier to train and optimize.

Multiple different sizes of ResNet exists, ResNet-18 being one of the smaller versions. It has 18 layers and about 11 million trainable parameters. We will use an implementation from PyTorch. The output (which will be the representations used later) is of

dimension 512.

Batch normalization

ResNet also uses *batch normalization* [9], which adjusts the mean and variance of inputs to the layers. By normalizing each batch before it is sent to the next layer, learning becomes more stable since the overall characteristics of the data stay constant. Given a layer output $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(d)})$, each dimension k can be normalized as

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}. \tag{3.1}$$

Here, the expectation and variance are calculated over the current batch. This will give a zero mean and standard deviation of 1, which is what we usually mean when we talk about normalization. However, this limits what the output can represent. It might be that a different mean or standard deviation would work better in the network. So, we introduce two parameters that can adjust the mean and standard deviation, $\gamma^{(k)}$ and $\beta^{(k)}$. We combine them with the normalized output from Equation 3.1 and get the input to the next layer

$$y^{(k)} = \hat{x}^{(k)} * \gamma^{(k)} + \beta^{(k)}. \tag{3.2}$$

These are initialized with $\gamma^{(k)} = 1$, $\beta^{(k)} = 0$, which gives the standard normalization. But, during training $\gamma^{(k)}$ and $\beta^{(k)}$ are updated by the optimizer, just like the network weights and biases. So, if it turns out that zero mean and unit variance is suboptimal, the network can learn a better version.

3.1.2 AlexNet

We will also try a modified version of AlexNet as the base encoder [10]. AlexNet, named after it's designer Alex Krizhevsky, was the winner of the ImageNet Large Scale Visual Recognition Challenge in 2012. It is a CNN with 5 convolutional layers and 3 fully connected layers.

For this network to function in our setting, we need to modify it slightly. Table 3.1 shows the details of the original AlexNet compared to our smaller version. Firstly, we are only interested in the convolutional layers since they are the part that efficiently extract features from images. So, we remove the fully connected layers.

Secondly, AlexNet is designed for images of size 3×224×224, but the images we will use are only 3×32×32. This makes it impossible to use the same size for the convolution kernels since each convolution decreases the size of the last two dimensions and we eventually get to 0×0. The kernel sizes have to be decreased, and some other parameters modified accordingly.

Finally, we want the output to be of dimension 2048, to make it comparable to results from other studies. The simplest way to do this is to change the number of filters in the last convolutional layer. Our version has just under 4 million trainable parameters.

	Original AlexNet	Smaller AlexNet
Conv1	64 filters, kernel 11, stride 4, padding 2	64 filters, kernel 3, stride 2, padding 1
MaxPool1	kernel 3, stride 2	kernel 2, stride 2
Conv2	192 filters, kernel 5, stride 1, padding 2	192 filters, kernel 3, stride 1, padding 1
MaxPool2	kernel 3, stride 2	kernel 2, stride 2
Conv3	384 filters, kernel 3, stride 1, padding 1	384 filters, kernel 3, stride 1, padding 1
Conv4	256 filters, kernel 3, stride 1, padding 1	256 filters, kernel 3, stride 1, padding 1
Conv5	256 filters, kernel 3, stride 1, padding 1	512 filters, kernel 3, stride 1, padding 1
MaxPool3	kernel 3, stride 2	kernel 2, stride 2

Table 3.1: Parameters of convolutional layers for original and modified AlexNet, differences highlighted.

3.1.3 Projection head

In the paper that introduces SimCLR, the architecture of the projection head is not specified beyond "an MLP with one hidden layer" (which uses ReLU as its activation function) and that the embeddings $\boldsymbol{z}_i$ are of dimension 128. Since the two base encoders create representations of different sizes (512 for ResNet-18, 2048 for AlexNet), the projection heads need to be slightly different.

We choose to decrease the size of each output evenly through the projection head. For ResNet-18 that gives 256 nodes in the hidden layer, the output from each layer is half the size of the previous layer (512 to 256 to 128). For AlexNet the hidden layer has 512 nodes, which means each layer decreases the output to 1/4 of the previous size (2048 to 512 to 128).

We want to remind the reader that the projection head is only used during training of the encoder. For evaluation, the projection head is disconnected and only the base encoder remains.

3.2 Evaluation

To evaluate the quality of the representations, we use a *linear evaluation protocol*. Linear classifiers are a common evaluation metric for unsupervised methods [6, 11, 12, 13]. If the accuracy of the classifier is high, it is a sign that the representations are good.

After training the unsupervised network, the projection head is disconnected and the remaining network is frozen. Then a single layer linear classifier is connected. This is trained and tested on previously unseen, labeled data. Note that we do not augment the images during training and testing of the classifier.

3.3 Data

This project will use CIFAR-10 [14]. It is a set of 60 000 labeled images with 10 classes, depicting different species of animal and types of vehicles. The pictures are 3 channel color images with 32×32 pixels. This dataset was chosen since it has been used in previous related research, making it simple to compare results [6, 11].

3.3.1 Data partition

Since there are two different models to train and optimize (the encoder and the classifier) a simple split into training, validation and test set is not enough. We want to measure validation loss during training of the encoder, so we need a training and a validation set for that model. In addition, we want to train the classifier, optimize it based on validation performance, and then report a final performance on test data. Finally, to make sure the generalization performance is as good as possible, the unsupervised model and the classifier should not train on the same data.

The 60 000 images in CIFAR-10 are originally split into 50 000 training images and 10 000 test images. A natural division is to use the training set for the unsupervised model and the test set for the classifier, then split those further into the sets needed. Table 3.2 shows the final partition of the dataset. To make the partitions, we use `torch.utils.data.random_split`, which creates a random split of a dataset.

Encoder		Classifier		
Training	Validation	Training	Validation	Test
40 000	10 000	4 000	1 000	5 000

Table 3.2: Number of images in each data set.

Comparison to Zhang et al.

A significant paper for this project is [11] by Zhang et al. They have also implemented a federated version of SimCLR as a comparison to their own method for SSFL. Thus, their results can serve as an interesting point of comparison. When it comes to hyperparameters and other settings, we will often use their choice as a starting point. An important difference is with the data partition. They seem to train both their encoder and classifier on the same set (the training set of CIFAR-10), test on the test set, and make no mention of validation. We make the choice to train the classifier on different images from the ones used to train the encoder, in order to maximize generalization performance. This means that their classifier is trained on a much larger set compared to ours (50 000 data points, compared to 4 000). This will have some consequences for our classifier settings.

3.4 Central training

To give the federated models some context we will train two *central models*, one with each base encoder. Rather than splitting the training and validation data over 5 clients, these non-federated models have all the data in one place. Central models can give an indication of how the performance of SimCLR is affected by the federated setting.

3.5 Optimizer

Adam is an optimization algorithm introduced by Kingma and Ba in [15], which we will use to train both the encoder and the classifier. Adam uses individual adaptive learning rates for each parameter, which are computed from the moments of the gradients. It is computationally efficient and empirically shown to work well.

PyTorch's implementation of Adam also includes an option for *weight decay*. This is a regularization technique that includes a penalty for large weights in the loss function, which improves Adams generalization performance [16].

3.6 Experiment set-up

Below, we present an overview of the whole training and testing process.

1. Train encoder (including projection head) on unlabeled data with augmentation
2. Disconnect projection head and freeze base encoder
3. Connect a classifier to the base encoder
4. Train the classifier on labeled training data without augmentation
5. Evaluate the classifier on labeled test data without augmentation, report accuracy

3.6.1 Encoder training settings

We will train both types of encoder in three different data settings: central, federated i.i.d. and federated non-i.i.d.. The two encoders are ResNet-18, which has 18 layers and around 11 million parameters, and AlexNet, which has 5 layers and just under 4 million parameters.

Zhang et al. uses a learning rate of 10^{-3} in their implementation. However, early experiments indicated that a smaller learning rate can be beneficial for AlexNet. So, for each combination of encoder and data setting we will train the encoder with two learning rates; 10^{-3} and 10^{-4}. The temperature parameter τ in the NT-Xent loss will

be set to 0.5 since that is found to be optimal by Chen et al [6]. All models will use Adam with weight decay 10^{-6}. All models will be trained with a batch size of 128.

Finally, we want to investigate the variability of the models. Does different random initializations affect the performance? For this reason, we will train 3 encoders for each setting and report mean and standard deviation of the classifier accuracy. Further details for the different data settings follow below.

Large Gaussian blur

Very late into the project, it was discovered that the Gaussian blur kernel used in the augmentation of images was too large. It is meant to be 10 % of the image size but was actually set to 30 % in our code. This means that many inputs have been more blurred than necessary. Unfortunately, there was not enough time to rerun the experiments, so all results in this report is affected by that.

Federated encoders

The federated encoders will be trained for 300 communication rounds, with 5 local epochs in each round. There are 5 clients, and all clients participate in every round.

In the i.i.d. (independent and identically distributed) setting the CIFAR-10 training set is evenly and randomly split over the clients. Each client has 10 000 images, which are the randomly split into 8 000 training images and 2 000 validation images.

The non-i.i.d. setting is meant to simulate the more realistic case where different users have different data distributions. One user might mostly photograph their cat, while another takes pictures of landscapes. In the non-i.i.d. setting each client get images from 2 of the 10 classes in CIFAR-10. This means that the client data distributions are completely disjoint. One client might only have the cars and birds, while another only has the dogs and planes. Note that this split is deterministic, client 1 will always have images from class 0 and 5, client 2 will have classes 1 and 6, and so forth. As in the i.i.d setting, each client set is randomly split into training and validation.

Central encoders

Central encoders will be trained for 300 epochs. Note that this is *not* the same as training the federated encoders for 300 rounds. In a round there is 5 epochs, meaning the federated models pass over the data 1500 times total. Meanwhile, the central models pass the data 300 times.

3.6.2 Classifier training settings

We initially used the same settings for the classifier as Zhang et al. in [11]. They have also implemented a federated a version of SimCLR, so we want to be able to compare their results to ours. They train their classifier for 100 epochs. However, early experiments showed a tendency to overtrain the classifier. This could be a problem that Zhang et al. did not have, since their classifier trains on a much larger dataset (as mentioned in Section 3.3.1) and thus is not as vulnerable to overtraining. So, for the results given in the rest of this report, the classifier has only been trained for 75 epochs.

For the classifier training we use the standard loss for classifiers, cross entropy loss. The classifier will be trained with Adam using a learning rate of 10^{-3} and weight decay 10^{-6}. We use a batch size of 128.

4 Results

4.1 Central setting

The results of training a classifier on CIFAR-10 using the representations from the central encoder Table 4.1. AlexNet with learning rate 10^{-4} and ResNet-18 with learning rate 10^{-3} perform the best.

Base encoder	Learning rate	Mean accuracy	95 % C.I
ResNet-18	10^{-3}	67.7 %	[64.9, 70.5]
	10^{-4}	64.3 %	[63.6, 65.0]
AlexNet	10^{-3}	60.7 %	[58.4, 63.1]
	10^{-4}	**69.0 %**	[66.7, 71.3]

Table 4.1: Mean accuracy and standard deviation of the classifier using encoders trained in the central setting. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder training time: 300 epochs; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

To investigate the impact of training duration, an AlexNet-encoder with learning rate 10^{-4} was trained for 200, 300, 400 and 600 epochs. The result can be seen in Figure 4.1. The performance initially improves with longer training, but seems to plateau around 400.

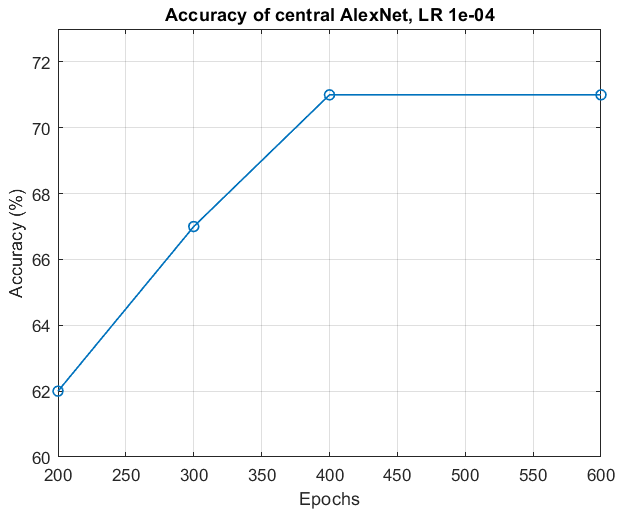


Figure 4.1: Accuracy by number of epochs trained for an AlexNet-encoder with learning rate 10^{-4} in the central setting. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

4.2 Federated models

4.2.1 I.i.d. setting

Table 4.2 show the mean accuracy and confidence intervals for each base encoder and learning rate in the i.i.d. setting, and Figure 4.2 show an illustration. The AlexNet-encoder with a lower learning rate beats the other models with 5 percentage points. However, it has an overlapping confidence interval with one of the ResNet-18-encoders, and thus we can't say with confidence that it is better.

Comparing these results with those of the central models in Table 4.1, we see that the mean performance drops with around 2-4 percentage units in the federated i.i.d. setting.

Base encoder	Learning rate	Mean accuracy	95 % C.I
ResNet-18	10^{-3}	64.0 %	[62.0, 66.0]
	10^{-4}	62.0 %	[60.9, 63.1]
AlexNet	10^{-3}	57.0 %	[52.6, 60.4]
	10^{-4}	**67.0 %**	[64.7, 69.3]

Table 4.2: Mean accuracy and standard deviation for the classifiers using federated encoders trained on i.i.d data. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

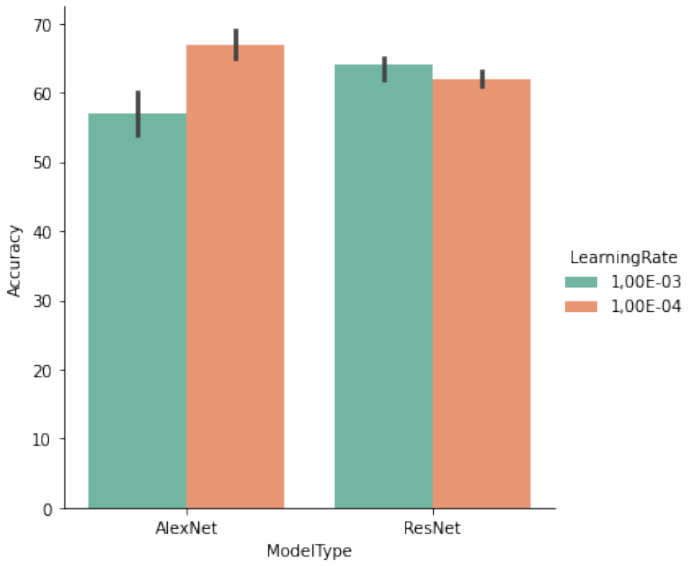
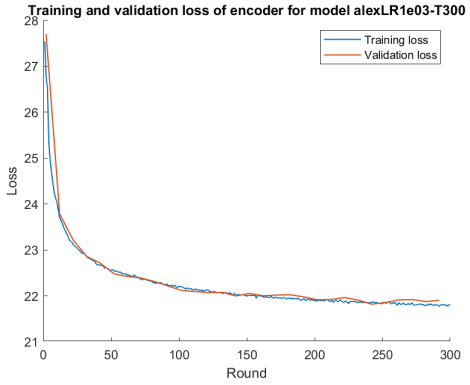


Figure 4.2: Mean accuracy of classifiers using each base encoder and learning rate in the i.i.d. setting, with 95 % confidence intervals in black. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

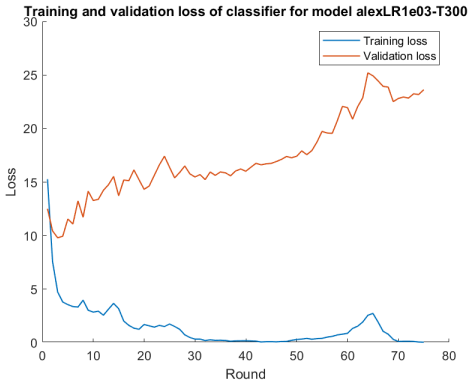
A notable behaviour also appears in the training of the classifiers. Figure 4.3 shows an example of training and validation loss during training of both encoders (on the left) and classifiers (on the right). The only difference between the two models in the figure is the learning rate used to train the encoder, all other parameters are identical in both encoder and classifier. When the encoder is trained with a learning rate of 10^{-3}, the performance during training of the classifier is much worse than otherwise.

Figure 4.3 only show one example of each model, but this behaviour was consistent for all AlexNet-encoders in both federated and central settings; a learning rate of 10^{-3} gives a diverging loss during training of the classifier. Additionally, despite the difference in the classifier loss, the encoder loss looks very similar for both learning rates.

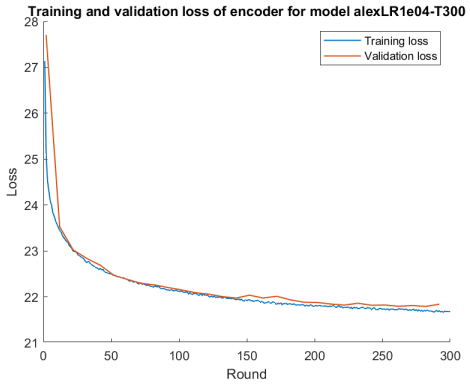
Figures 4.4 and 4.5 show confusion matrices for two AlexNet encoders trained with learning rates of 10^{-3} and 10^{-4} respectively. Note htat the predictions are skewed, in general there is more confusion among the animal classes. This is a trend that we see through all models in this project, but it is more pronounced in the AlexNet-encoder with a larger learning rate (the type that diverges during classifier training).



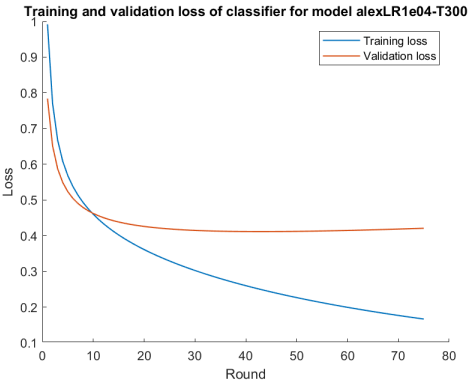
(a) Training and validation loss of **encoder**, AlexNet trained with learning rate 10^{-3}.



(b) Training and validation loss of **classifier**, encoder is AlexNet trained with learning rate 10^{-3}.



(c) Training and validation loss of **encoder**, AlexNet trained with learning rate 10^{-4}.



(d) Training and validation loss of **classifier**, encoder is AlexNet trained with learning rate 10^{-4}.

Figure 4.3: Training and validation loss of encoder and classifier for AlexNet trained with two different learning rates in the i.i.d setting. Note the difference in scale on the y-axis in subfigures b and d. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

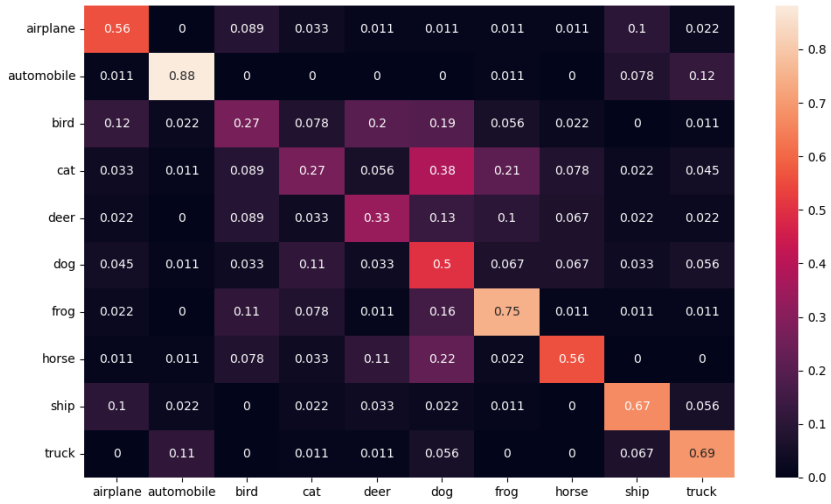


Figure 4.4: Confusion matrix for classifier using an AlexNet-encoder trained in i.i.d. setting with learning rate 10^{-3}. Predicted label along the x-axis, true label along the y-axis. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

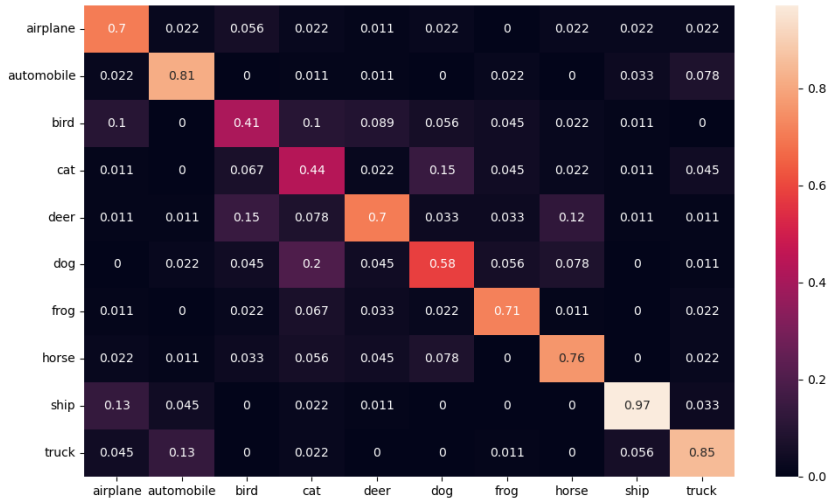


Figure 4.5: Confusion matrix for classifier using AlexNet-encoder trained in i.i.d. setting with learning rate 10^{-4}. Predicted label along the x-axis, true label along the y-axis. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

4.2.2 Non-i.i.d. setting

Finally, the setting meant to be the most realistic: non-i.i.d.. The accuracy and confidence intervals are presented in Table 4.3 and plotted in Figure 4.6. As in the previous cases, AlexNet with a 10^{-3} learning rate performs the best. In this case, with a significant margin.

If we compare the non-i.i.d. results with the i.i.d. results from Table 4.2, we can see that ResNet-18 is more affected by the data being non-i.i.d. The ResNet-18 models drop 14 and 10 percentage units in mean accuracy, while AlexNet only drops 4.3 and 4.7 percentage units.

Base encoder	Learning rate	Mean accuracy	95 % C.I
ResNet-18	10^{-3}	50.0 %	[48.9, 51.1]
	10^{-4}	52.0 %	[50.0, 54.0]
AlexNet	10^{-3}	52.7 %	[51.4, 54.0]
	10^{-4}	**62.3 %**	[61.6, 63.0]

Table 4.3: Mean accuracy and standard deviation for the classifiers using federated encoders trained on non-i.i.d data. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

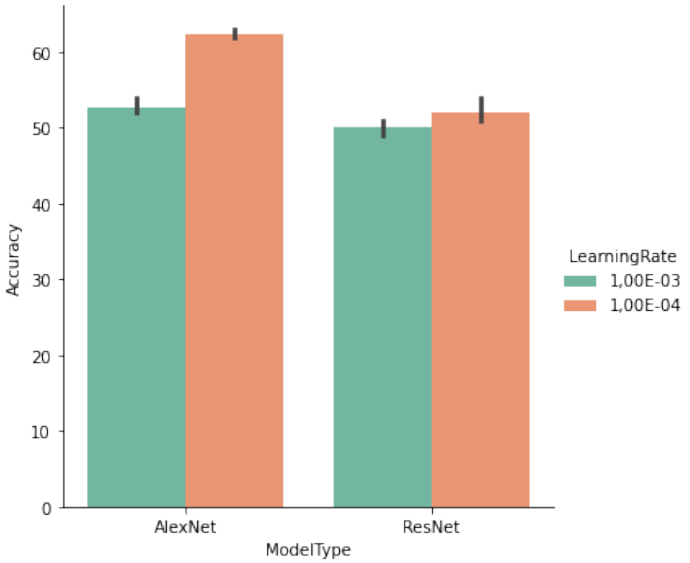


Figure 4.6: Mean accuracy of classifiers using each base encoder and learning rate in the non-i.i.d. setting, with 95 % confidence intervals in black. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; encoder local epochs: 5; encoder communication rounds: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

4.3 Overview

The mean accuracy for all model types and settings are gathered in Table 4.4 and illustrated with confidence intervals in Figure 4.7. The top performer is consistent in each data setting, AlexNet with 10^{-4} learning rate. In both central and federated i.i.d., it is followed by ResNet-18 with 10^{-3} learning rate, however that model performs the worst in the non-i.i.d. setting.

Data setting	Learning Rate	ResNet-18	AlexNet
Central	10^{-3}	67.7 %	60.7 %
	10^{-4}	64.3 %	**69.0 %**
Federated i.i.d	10^{-3}	64.0 %	57.0 %
	10^{-4}	62.0 %	**67.0 %**
Federated non-i.i.d	10^{-3}	50.0 %	52.7 %
	10^{-4}	52.0 %	**62.3 %**

Table 4.4: Mean accuracy of classifiers with each base encoder and setting. The highest performance in each data setting is marked. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; federated encoder local epochs: 5; federated encoder communication rounds: 300; central encoder epochs: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

From Figure 4.7, we note that ResNet-18 in general seems to have a more consistent performance with smaller confidence intervals, but the difference is quite small. We can also see that for AlexNet, a smaller learning rate is clearly preferable, while it is not as clear cut for ResNet-18.

4.3.1 Comparison to Zhang et al.

In their federated SimCLR-implementation, Zhang et al. reach an accuracy of 61.62 % in an i.i.d setting and 59.21 % in a non-i.i.d setting with AlexNet and learning rate 10^{-3} [11]. Note that we do not know if/how they have modified AlexNet to work with CIFAR-10. This is higher than our AlexNets trained with that learning rate, but lower than our best AlexNets, which use a learning rate of 10^{-4}.

They have also trained ResNet-50 as a base encoder, which perform better than our ResNet-18 (68.1 % and 64.06 % for i.i.d. and non-i.i.d. respectively).

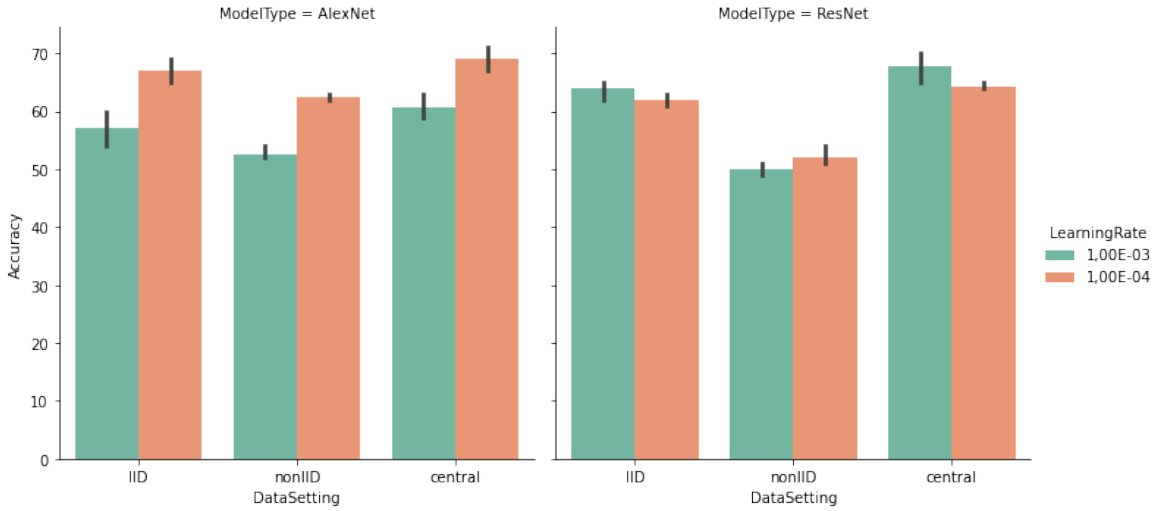


Figure 4.7: Overview of mean accuracy of classifiers with each encoder type and data setting, 95 % confidence intervals in black. Data set: CIFAR-10; encoder loss: NT-Xent; temperature τ: 0.5; encoder optimizer: Adam; encoder weight decay: 10^{-6}; federated encoder local epochs: 5; federated encoder communication rounds: 300; central encoder epochs: 300; encoder batch size: 128; classifier loss: cross entropy; classifier optimizer: Adam; classifier LR: 10^{-3}; classifier weight decay: 10^{-6}; classifier training time: 75 epochs; classifier batch size: 128.

5 Discussion and conclusions

5.1 Discussion of results

5.1.1 AlexNet vs ResNet-18

In our results, AlexNet (with a suitable learning rate) generally outperforms ResNet-18. This might seem unexpected at first. In general, one would expect larger models to outperform smaller ones. AlexNet is a simple 5-layer CNN, while ResNet-18 is much larger and more complex. In addition, other implementations of SimCLR shown higher performances from ResNet-50 than from AlexNet [11].

The reason representations created by AlexNet has a higher quality is probably simply due to the sizes of the representations. As we have previously mentioned, the representations created by ResNet-18 are of length 512 while the AlexNet outputs are of length 2048. It is of course possible to retain more information in the larger representations, which might make the classification easier.

This situation could be described as 'quantity over quality'. We would expect representations created by a more advanced model to contain data that is more precise or of higher quality. If the representations from both models where of the same size, we would expect ResNet-18 to perform the best. But in this case, it seems to be more useful with representations containing a large volume of data, rather than representations that contain more precise information.

5.1.2 AlexNet and large learning rates

As shown in Section 4.2.1, Figure 4.3, the learning rate during training of the base encoders has a large impact on the performance of the classifier. In all settings, the AlexNet-encoder trained with a larger learning rate has around 9 percentage points lower classifier accuracy. The divergence of the classifier training and validation loss indicates overtraining in the classifier. Since the classifier has identical settings for all base encoders, the only explanation is that there is a fundamental difference in the representations fed to the classifier. One possibility is that the *encoder* is overtrained, which leads to further overtraining in the classifier and a poor generalization performance.

It is also interesting to note that while there is a large difference in the loss curves for the classifiers, the loss curves for training of the base encoders are very similar. So, the NT-Xent-loss is not a very good indicator for the quality of the representations when they are used in a classifier.

5.1.3 In practical applications

From a "real world"-perspective, the fact that AlexNet is as good as or better than ResNet-18 is good news. AlexNet only has about a third as many parameters as ResNet-18. Our implementations use 45.4 and 15.8 MB respectively when they are stored. Of course, an application intended for real phones use would have more efficient implementations, but this gives an indication of the size difference of the models. Since memory space often is a limited resource on phones and other smart devices, it is great if a small model is as good as a big one.

When we examine our results, we see signs of what could be a big hurdle in "real world"-SSFL applications. Although the federated model in the i.i.d. setting perform almost as well as the central one, the non-i.i.d. version is significantly worse. Since most applications of SSFL will have to deal with non-i.d.d. data, it will be challenging to achieve results that are comparable to central models. However, our non-i.i.d. setting is quite an extreme one, with no overlap between clients' labels. A more realistic case would be that different labels dominate the client distributions, but that there still is some overlap. So, while non-i.i.d. data will be a challenge, we cannot say for sure how big that challenge will be.

5.2 Impact of errors and experiment design choices

5.2.1 Gaussian blur

We mentioned in Section 3.6.1, the Gaussian blur kernel was accidentally set too big during augmentation. This makes it so any image that was blurred will be more blurred than intended. There was not enough time to run experiments to investigate the impact of this on the encoder. A possible effect is that it trains the network to latch on to different features than a network trained with less blur would. Maybe more emphasis is put on colours and their placement in the picture, or bigger features rather than edges and corners that would disappear with the blur.

5.2.2 Batch normalization

As mentioned in Section 3.1.1, ResNet uses batch normalization, which incluedes learnable parameters γ and β (see Equation 3.2). In our federated implementations of ResNet-18, we treat these parameters the same as the model weights and average them between all local models in each communication round. However, some research suggests that it is better to keep the batch normalization parameters local during federated training [17]. This would be especially beneficial for situations where the clients have very different data distributions. When all the data is sampled from the same underlying distribution, normalizing it to stabilize training is reasonable. However, normalizing all the data to one distribution when the underlying distributions are different could cause us to hide relevant information from the network and make training harder.

Looking at the overview of the results (Table 4.4), we see that the accuracy drop from the central to the federated i.i.d. setting is about equal for both models. But the drop between the i.i.d. and non-i.i.d settings is much larger for ResNet-18 than for AlexNet. AlexNet does not use batch normalization. If batch normalization hurts training when the clients have heterogeneous data distributions, that could be a possible explanation for the larger drop. Of course there are many other differences between the two model architectures, so we cannot be sure that this is the reason (or only reason) for the drop. Further studies that compare ResNet with and without averaging of the batch norm parameters could be an interesting prospect.

5.3 Limitations and possible future research

5.3.1 Data split

Due to the numbering of classes in CIFAR-10 and our method for dividing them over clients, the non-i.i.d. split results in all clients except one having examples from one vehicle class and one animal class. One could argue that this makes the client distributions more similar than if some clients only had vehicles and some clients only had animals. But one could also argue that even if humans tend to use overarching categories like "animal" before we divide further into "cat", "dog", etc, that is not necessarily how the network is structured. It would be interesting to see if a more "uneven" split would be more challenging for the encoders.

5.3.2 Batch size

In its standard implementation, SimCLR benefits from larger batch sizes [6]. It would be interesting to see if the same is true for a federated implementation. Unfortunately, this project was unable to explore that line of research, first due to limited GPU capacity and then time constraints. The impact of batch size on the representation quality could be an interesting future endeavour.

5.3.3 Training duration

Another factor to examine would be the impact of training duration. The experiment where a single encoder was tested after training for 200, 300, 400 and 600 epochs indicates that longer training can be beneficial in the central setting but that it plateaus after some point. We would have liked to do similar experiments in the federated settings, but there was not enough time.

5.4 Conclusions

Our research only covers a small part of the very big field of SSFL and as mentioned before a lot more research can and should be done. We summarize our findings that could be helpful to keep in mind for future research.

- Training an encoder in a federated setting can result in a representation quality that is almost equal to that of a centrally trained encoder.
- The challenge with SSFL in real world federated applications may be heterogeneous client distributions, rather than federation itself. We observe that a setting where each client has their own data distribution yields significantly lower representation quality than the homogeneous setting.
- Using centrally aggregated parameters for batch normalization does not work well in a setting where label distribution varies among clients.
- The benefits of a large representation size seem to outweigh those of a more advanced model. Representations created by AlexNet, a small CNN, consistently outperforms those created by ResNet-18, a much more advanced model but with a smaller output and thus smaller representation size.

Bibliography

[1] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson and Blaise Aguera y Arcas. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[2] Jie Xu, Benjamin S Glicksberg, Chang Su, Peter Walker, Jiang Bian and Fei Wang. "Federated learning for healthcare informatics". In: *Journal of Healthcare Informatics Research* 5.1 (2021), pp. 1–19.

[3] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus Maier-Hein et al. "The future of digital health with federated learning". In: *NPJ digital medicine* 3.1 (2020), pp. 1–7.

[4] Zhaoyang Du, Celimuge Wu, Tsutomu Yoshinaga, Kok-Lim Alvin Yau, Yusheng Ji and Jie Li. "Federated learning for vehicular internet of things: Recent advances and open issues". In: *IEEE Open Journal of the Computer Society* 1 (2020), pp. 45–61.

[5] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li and H Vincent Poor. "Federated learning for internet of things: A comprehensive survey". In: *IEEE Communications Surveys & Tutorials* (2021).

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

[7] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[9] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[10] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[11] Fengda Zhang, Kun Kuang, Zhaoyang You, Tao Shen, Jun Xiao, Yin Zhang, Chao Wu, Yueting Zhuang and Xiaolin Li. "Federated unsupervised representation learning". In: *arXiv preprint arXiv:2010.08982* (2020).

[12] Chaoyang He, Zhengyu Yang, Erum Mushtaq, Sunwoo Lee, Mahdi Soltanolkotabi and Salman Avestimehr. "SSFL: Tackling Label Deficiency in Federated Learning via Personalized Self-Supervision". In: *arXiv preprint arXiv:2110.02470* (2021).

[13] Xinlei Chen and Kaiming He. "Exploring simple siamese representation learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 15750–15758.

[14] Alex Krizhevsky, Geoffrey Hinton et al. "Learning multiple layers of features from tiny images". In: (2009).

[15] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[16] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[17] Mathieu Andreux, Jean Ogier du Terrail, Constance Beguier and Eric W Tramel. "Siloed federated learning for multi-centric histopathology datasets". In: *Domain Adaptation and Representation Transfer, and Distributed and Collaborative Learning*. Springer, 2020, pp. 129–139.

Appendix A

Details on augmentations

All augmentations are done using methods from `torchvision.transforms`. These are the same settings used in the paper that proposed SimCLR [6]. The transforms are applied in the order they are described below.

Crop and resize - `RandomResizedCrop`

We randomize how big the area of the crop should be, uniformly chosen from $[0.08, 1]$ and the ratio of the crop, uniformly chosen from $[\frac{3}{4}, \frac{4}{3}]$. Then the crop is resized back to the size of the original image.

Flip - `RandomHorizontalFlip`

There is a 50 % probability to flip the image horizontally.

Color distortion - `ColorJitter, RandomGrayscale`

Brightness, contrast, saturation and hue are jittered with strength 0.8. This jitter is applied with a probability of 0.8. In addition, there is a 0.2 probability that the image is transformed to grayscale.

Blur - `GaussianBlur`

There is a 50 % chance to apply a Gaussian blur with kernel size 9×9 (OBS) and standard deviation chosen uniformly from $[0.1, 2.0]$.