Federated Learning for Mobile and Embedded Systems

by

Stefan Hofman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday November 27, 2020 at 1:00 PM.

Student number:	4887336	
Project duration:	February 1, 2020 – November 27, 2020	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. T. G. R. M. van Leuken,	TU Delft
	Dr. ir. J. Hoozemans,	TU Delft

TU Delft

Abstract

An increase in the performance of mobile devices has started a revolution in deploying artificial intelligence (AI) algorithms on mobile and embedded systems. In addition, fueled by the need for privacy-aware insights into data, we see a strong push towards federated machine learning, where data is stored locally and not shared with a central server. By allowing data to stay on client devices and do training locally, we work towards a more privacy-friendly future. Furthermore, utilizing federated machine learning enables machine learning in data-constrained environments where bandwidth is not sufficient to upload the entire dataset.

In this thesis, we look at the recent trend into less complex machine learning models. These models optimize resource usage while reducing accuracy loss. We investigate how these simpler models hold up within a federated setting. We also look into the developments of AI frameworks and their capabilities for mobile platforms.

Based on these findings, we propose that model-hyper-parameter optimization is possible to maximize accuracy for smaller networks during federated learning. We show that it is possible to reduce the accuracy loss from 15% to only 0.04%. We then demonstrate what a mobile implementation looks like and the performance we see from an iPhone X. We show that an iPhone implementation takes less than $2\times$ the amount of a regular laptop implementation. Finally, we demonstrate that we can reduce the model-size by up to $7\times$ using modern weight quantization methods.

Preface

Dear reader,

With this report, containing my MSc thesis, I am concluding my time at TUDelft. I have gained a tremendous amount of knowledge that has prepared me to start my professional career with confidence. The work in this report is the result of a nine-month effort within the ABS group and in collaboration with Huawei under the supervision of Dr. Ir. Zaid Al-Ars.

First of all, I would like to thank Zaid for his continued support and constructive feedback. We had great discussions that lead to new insights and creative solutions. Even while the world shut down, and all the discussions moved online, Zaid was always open for a quick chat.

Secondly, I want to thank Dr. Ir. Joost Hoozemans and Dr. Ir. René van Leuken for being on my thesis committee and giving me feedback along the way. Furthermore, a special thanks to David Enthoven, who has given me many insights into the process of federated learning.

I also want to thank Ákos Hadnagy, his feedback on my report and his input during our lengthy coffee ~~breaks~~ meetings were invaluable and have driven this research to a higher standard.

Finally, I would like to express my deepest appreciation to my entire family. In particular, my parents, *Ingeborg* and *Dick*, who have supported me the entire duration of my studies. I could not have achieved this without them, so for that, I will be forever grateful.

Stefan Hofman
Den Haag, November 2020

Contents

1 Introduction

1.1. Context

Artificial intelligence (AI) is a useful and powerful technology. It enables a wide variety of applications ranging from image classification, word prediction, to voice recognition. The common factor in all of these applications is that a lot of data is necessary to train the models to provide accurate predictions. Data to train these models might not always be readily available for developers of those models and might live on devices of customers. Getting that data to developers to train neural networks is an ever-growing challenge. Here we run into privacy and bandwidth related issues. It is undesirable to distribute a user's private data to a central place where a third-party might read that data. Also, bandwidth limitations might arise where sending a large amount of data is simply impractical.

Another concern that might emerge is the amount of computational power that is needed to train neural networks. Training a network at a central location requires a large amount of GPUs which are expensive to purchase and to run due to electricity cost. This problem escalates with network size, where bigger networks give higher accuracy but require even more computations to train. And where more complex networks require more computing power to train, they also need more computing power to do inference tasks. The increase in computing power to do inference might result in undesirable scenarios where client devices might not have powerful GPUs to do these computations or are very limited by power usage.

Especially now, we see more and more intelligent features coming into consumer devices. These can be smart assistants on mobile phones, smart fridges, or even self-driving cars. All of these applications might run into the problems mentioned above, where it is undesirable to send data from a mobile phone to a central place, or cars that do not have the bandwidth to send all data to a central place. This is why new techniques are popping up to combat these problems. One of the most promising of these is "Federated Machine Learning" which was proposed in [35]. And in this research, we will investigate whether and how we can combine federated machine learning with the efficiency of mobile-oriented machine learning models like MobileNetV2 [36].

1.2. Challenges

The main problem is that there has been no research into using small machine learning models in a federated environment. It is proven that a large machine learning model can be used effectively in a federated setting without a significant accuracy loss, but it is unclear if this is also the case for small (mobile) machine learning models.

One of the more challenging problems we might face along the way is support for on-device training by artificial intelligence frameworks. This is something that would not be possible to solve in the time span of this research.

1.3. Problem statement

This thesis aims at addressing the void that exists in federated machine learning with regards to its effectiveness on smaller (mobile) networks. Based on the problems discussed above, the following

research questions can be formulated:

- How do smaller (mobile) machine learning networks perform in a federated environment?
- How can we ensure that smaller (mobile) machine learning networks are able to perform well in a federated environment?
- Are frameworks ready for mobile (on-device) machine learning training and inference?
- How well do mobile machine learning networks perform on mobile devices?

1.4. Thesis outline

This thesis is organized as follows:

- **Chapter 2: (Background)** provides background information regarding regular machine learning and federated machine learning.
- **Chapter 3: (Mobile machine learning models)** provides an overview of multiple mobile machine learning networks.
- **Chapter 4: (Evaluating federated mobile machine learning networks)** shows the experimental results from federating a mobile machine learning network.
- **Chapter 5: (Mobile machine learning frameworks)** provides an extensive overview of multiple machine learning frameworks with an orientation towards mobile and embedded devices.
- **Chapter 6: (Experimental results - Mobile implementation)** shows the experimental results from running mobile machine learning networks on actual mobile hardware.
- **Chapter 7: (Conclusion & further research)** summarizes this thesis, provides some lessons learned, and suggests some areas for further research.

2

Background

This chapter provides background regarding machine learning, distributed machine learning, and mobile machine learning models.

2.1. Machine learning

Machine learning (ML) is a common term for the study of self-improving computer algorithms. These computer algorithms, which we also call machine learning models, improve themselves by using what is called *training data* in a procedure which is called *training*. During training, we try to fit a machine learning model in such a way that with an input x we can predict an output y. We do this fitting by presenting the model with an input and compare the output of the model with the ground truth. The distance between our prediction and the ground truth is called the loss and can be calculated using a loss-function. Two of the more common loss-functions are mean squared error loss Equation (2.1) and cross-entropy loss Equation (2.2).

$$loss(x, y) = \overline{(x - y)^2} \tag{2.1}$$

$$loss(x, y) = -\sum x \log y \tag{2.2}$$

This loss will then be propagated over the *weights* of the model in a step we call *backpropagation*. A common representation of a machine learning model is shown in Figure 2.1, where we have a single input layer, at least one hidden layer, and an output layer. These layers are comprised out of weights that we update during backpropagation using the loss function to fit the model to predict the right output.

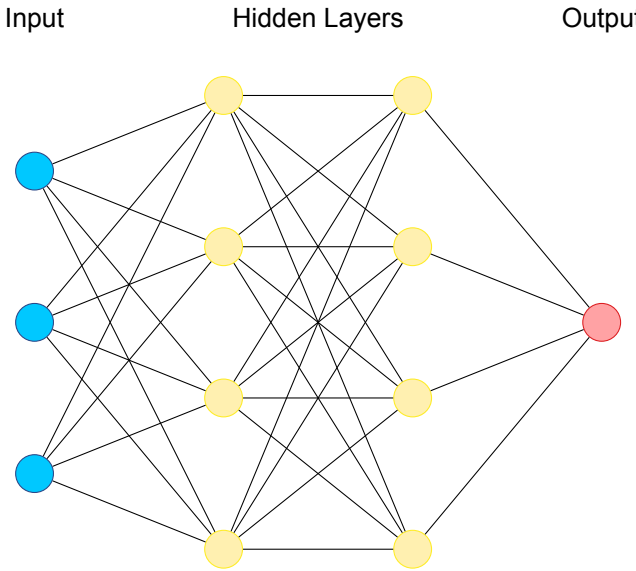

Figure 2.1: A schematic overview of a machine learning model with two hidden layers.

The ultimate goal for a machine learning model is to predict the correct output with 100% accuracy. For image classification, which we will be looking at in this thesis, this means that the model is supposed to recognize the class of the object in the given input image. On our way to this goal, two unwanted scenarios [24] might occur:

- **Overfitting:** this shows when a classifier is 100% accurate on the test dataset but significantly less accurate on the *validation dataset* (this is a dataset that is not used during training, so we can verify without bias that our model is accurate). It means that we might be learning too specific details about images that might not occur in all images of the same class. For example, we are training our model to classify cars from trucks, and in every training image of a car there is also a tree visible. In this case, overfitting can mean that we can classify an image of a truck next to a tree as a car. This can be solved by using more training images with a higher variety of contexts in the images and numerous *generalization* techniques. [37]
- **Underfitting:** this is the opposite of overfitting, and occurs when the model is not complex enough. The model will not be able to distinguish subtle differences, and thus not be able to make accurate predictions. Solutions to underfitting often lie in pre-processing or adding more complexity to the machine learning model by adding new layers. [40]

A schematic overview of what underfitting and overfitting look like can be found in Figure 2.2. Here we see that underfitting clearly misses some detail, and overfitting has too much detail. With the right amount of generalization, we can get the right amount of fitting and this will give the best result on validation images and thus on general prediction use cases.

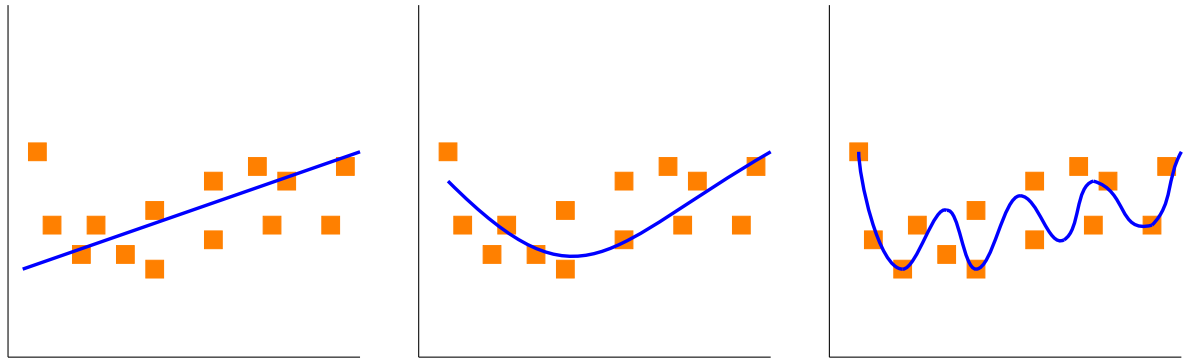

Figure 2.2: An example of what overfitting, underfitting, and a correct fitting look like.

Data distribution: The training data can be presented to our machine learning model in two distinct ways:

- **Independent and Identically Distributed (IID):** if the training data is IID, every category has the same probability distribution as the other categories and all are mutually independent. To put this into an example, if we have a training dataset of pictures with dogs and cats, it means that we have the same amount of dog pictures as we have cat pictures.
- **Not Independent and Identically Distributed (Non-IID):** if the training data is Non-IID, we do not have the same probability distribution, nor are they all mutually independent. This means, for the same example, that we might have more pictures of dogs in our training than pictures of cats.

Hyper-parameters: When we talk about machine learning, we come across so-called hyper-parameters. These hyper-parameters need to be set by the developer, as opposed to the other parameters like weights that are derived during training. Some of these hyper-parameters are:

- **Learning rate (η):** the learning rate indicates how much each epoch counts toward the final weight of the layers. The learning rate is one of the most important hyper-parameters, as using

a learning rate that is too small would result in impractical training times, but using a learning rate that is too big would result in a machine learning network that does not converge. A variable learning rate might be used to make quick progress in the beginning with a large learning rate, and a small learning rate at the end to end up with high accuracy.

- **Batch size (B):** the batch size indicates how much of the training data we process at a time. Historically, one would often use only one single example of the dataset to fit on the model. But more commonly these days, we give a ”batch” of examples to fit on the model. Common batch sizes are 2, 4, 8, 16, 32, 64, 128, 256. Using a larger batch size is more computationally efficient, but requires more memory.
- **Number of Epochs:** the number of epochs is the number of times we fit a single example or batch on the network. If this number is too low, our network will not be fully trained yet, but if it is too high, it may take a long time to finish. A technique used is early stopping, which terminates the learning process if the accuracy does not improve in x rounds.

2.2. Federated machine learning

2.2.1. Overview

Federated machine learning is a type of distributed machine learning [33]. Distributed machine learning is a collective name used to bundle machine learning techniques where the training of a machine learning model is not done on the same compute node. This can indicate that we split our training dataset into multiple (more or less) even parts that we will send to multiple compute nodes, these compute nodes will then collectively train the machine learning model. Distributed machine learning is used because training a machine learning model is extremely compute-intensive, and splitting this computational intensity will significantly decrease training times. Federated machine learning is, as stated above, a form of distributed machine learning where the main difference is that the data to train the machine learning model is not available at the central server. While in distributed machine learning we split the dataset into multiple parts, it is not possible to do that in federated machine learning, because we do not have this data. Instead, every device with data will learn from that data locally and send the machine learning model to a central server. The central server will then average the models it receives from various clients and sends back the averaged model to the clients. Both a general distributed learning and a federated learning sequence is shown in Figure 2.3.

Some hyper-parameters used in federated machine learning are:

- **Local epoch size (E):** the local epoch size is similar to the number of epochs from the general hyper-parameters, but here it represents the number of epochs that are done on a client device before sending the updated machine learning network.
- **Number of Clients (u):** this is the number of client devices being utilized during the federated learning.
- **Client fraction (C):** this is the fraction of clients being used within a single ”communication round” or global epoch.

By using federated machine learning, we can identify three main advantages which we will discuss in the next sub-chapters: privacy, computational efficiency, and data efficiency.

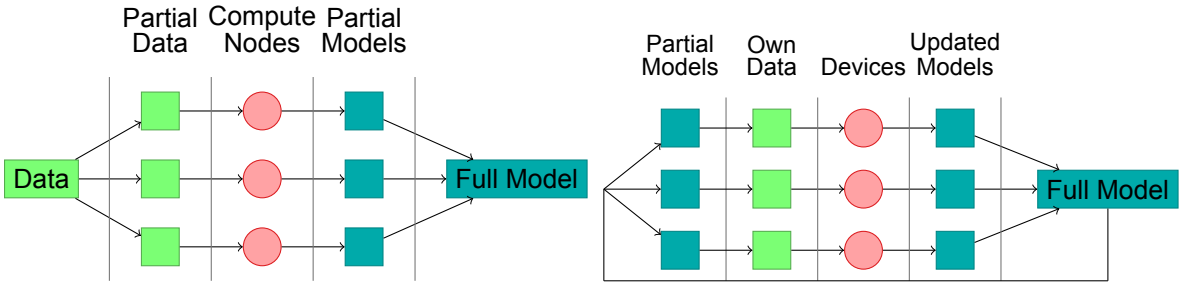

Figure 2.3: A schematic overview of a general distributed learning process (left) and a federated learning process (right).

2.2.2. Privacy

Privacy is a big part of why we use federated machine learning. People are getting more privacy-aware, and even laws like the general data protection regulation (GDPR) in the European Union might mandate these privacy precautions. In regular centralized machine learning or general distributed learning, it is necessary to have all the training data at the central server to start training. Sending the training data to the central server means that, for example, private images on a mobile device will have to be sent off-device where we do not have any control over what happens to that image anymore.

A more elegant solution would be to use federated machine learning. In federated machine learning, we do not have to send the training data to a central server. Instead, we can train on devices locally and only send the averaged models to the central server. Now our private information stays private on our own devices [45].

While this thesis research does not focus on privacy, specifically, it is worth noting that federated machine learning is not inherently private [25]. There are attack methods that enable the reconstruction of images from the gradients in the machine learning model itself [26]. While there are attack methods to extract information from the model, there are also defense methods to defend against such attacks. These defense methods do usually come with an accuracy trade-off, so it is possible to make a model more private but at the cost of lower model accuracy.

2.2.3. Computational efficiency

Federated machine learning also has a significant impact on computational efficiency compared to regular machine learning. As we have said earlier, training a machine learning model requires a lot of computing power, and training is therefore mostly done on large GPU (Graphical Processing Unit) clusters or sometimes even TPU (Tensor Processing Unit) clusters. It can be an incredible cost-saving measure if machine learning models were to be trained on client-devices instead of a central GPU cluster. Training on client-devices comes with its own set of problems since client-devices often do not have powerful GPUs to train machine learning models. Also, client-devices might be power-constrained and might have to run on batteries or are running in low-power settings. The key element which allows the client-devices to train the machine learning models themselves is that they only work on their data. Because they only work on their data, they only have to process a subset of the total data, and thus require less computation to train the model.

The viability of training on client-devices is also expected to increase over time. As a consequence of Moore's law, which predicts that the number of transistors in a dense integrated circuit double every two years, computational power in client-devices is predicted to only go up. And especially for mobile phones which, are getting fitted with dedicated neural engines to do machine learning tasks efficiently.

2.2.4. Data efficiency

The last main advantage of using federated machine learning is data efficiency. With data efficiency, we mean that we can reduce the amount of bandwidth that is required to train a machine learning model. During regular distributed machine learning, all the training data has to be sent to the central server for training. After training, the trained model has to be sent back to the client devices. This process can be seen in Figure 2.4. From the image, we can see that the clients either send their data to the central server (CS) or just the models which they trained using their data locally. In this thesis, we will explore how to use small machine learning models, and the smaller a model is, the more data-efficient we can get.

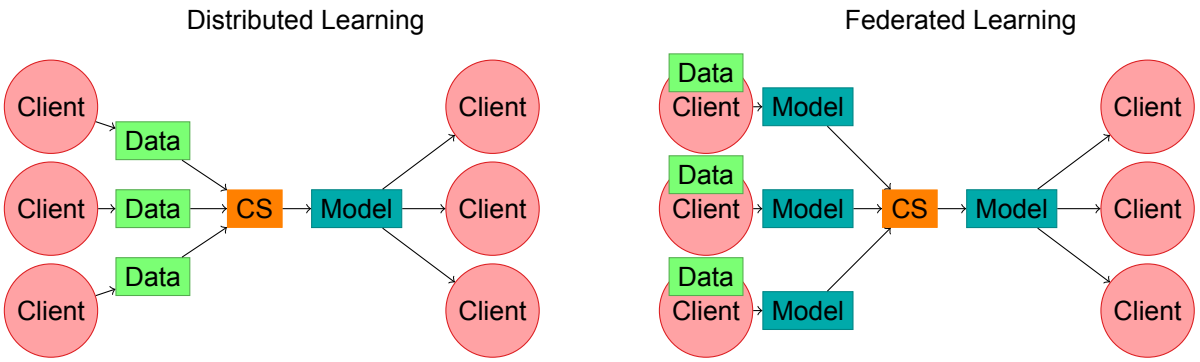

Figure 2.4: A schematic overview of the data-flow of distributed learning (left) and federated learning (right).

The reason data-efficiency is important is because most client-devices have a restricted bandwidth capacity. Mobile phones might be on a cellular or WiFi connection, and self-driving cars might only have a cellular connection. These wireless connections have a restricted bandwidth and might have difficulties sending all the data to the central server. By only sending the machine learning model sometimes, we limit the bandwidth needed to train such a model.

3

Mobile machine learning models

The general trend in machine learning is to create deeper, more accurate convolutional neural networks. [30, 38, 39] These improvements in accuracy, however, create machine learning networks that are bigger in size and show lower inference throughput. For real life implementations in mobile and embedded environments, it is often not possible to use these big networks. So what we are seeing now is that more resource efficient networks are being developed. [31, 32, 36, 41, 43] These resource-efficient networks try to reduce model complexity while retaining as much accuracy as possible. Reducing model complexity might, besides lower accuracy, also result in less resilient networks. This means that accuracy can drop fast when only changing things like hyper-parameters or the training environment. In this chapter, we will go over multiple mobile machine learning networks to highlight what they do to decrease network size, improve performance, and minimize accuracy loss.

3.1. SqueezeNet

SqueezeNet [32] is one of the earlier mobile machine learning networks. SqueezeNet lays out a 3 step strategy on how to decrease the size of the network by decreasing the number of parameters, while keeping the accuracy reasonable.

- **Step 1:** Replace as many 3×3 filters with 1×1 filters as possible, this will decrease the network size significantly, since 1×1 filters have $9\times$ fewer parameters than 3×3.
- **Step 2:** Decrease the number of input channels to 3×3 filters. When a convolutional layer is fully composed out of 3×3 filters, the number of parameters is equal to Equation (3.1), where M is the number of input channels, and F is the number of filters.

$$M \cdot F \cdot 3 \cdot 3 \tag{3.1}$$

  As we can derive from Equation (3.1), it is not only beneficial to decrease the filter size, but it can also be advantageous to decrease the number of input channels of the 3×3 filters. This is done in SqueezeNet with so-called *squeeze layers*.

- **Step 3:** Delay downsampling in the network to maintain large activation maps. As we can see in [28], delaying downsampling leads to higher accuracy by maintaining large activation maps. Downsampling is done by using a $stride > 1$. If downsampling is done at the beginning of the network, the subsequent layers in the network will have small activation maps which will lead to worse accuracy.

While Step 1 and 2 will decrease the network size while trying to preserve accuracy, step 3 is about maximizing the accuracy while keeping the size of the network in mind.

SqueezeNet proposes, based upon the three steps, a new building block called the *Fire module*. The Fire module is made up of a *squeeze* convolution layer which feeds into an *expand* layer. The squeeze layer only contains 1×1 convolution filters, and the expand layer contains both 1×1 and 3×3 convolution filters. The Fire module exposes the following new hyper-parameters:

- $s_{1\times1}$ - the number of 1×1 filters in the squeeze layer.
- $e_{1\times1}$ - the number of 1×1 filters in the expansion layer.
- $e_{3\times3}$ - the number of 3×3 filters in the expansion layer.

The hyper-parameters can be set using the constraint Equation (3.2) to follow Step 2 where the squeeze layer limits the number of input channels in the expansion layer. A schematic overview of the Fire module can be found in Figure 3.1 and an overview of the SqueezeNet architecture can be found in Table 3.1

$$s_{1\times1} < (e_{1\times1} + e_{3\times3}) \tag{3.2}$$

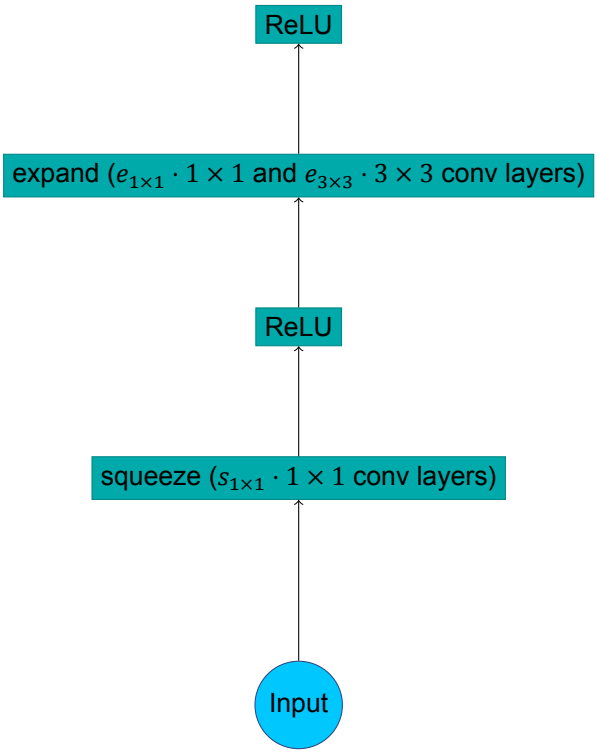

Figure 3.1: Fire module.

Table 3.1: Architecture of SqueezeNet.

Type	**Stride**	**Input Size**	$s_{1\times1}$	$e_{1\times1}$	$e_{3\times3}$
Conv	2	$224 \times 224 \times 3$			
Maxpool	2	$111 \times 111 \times 96$			
Fire		$55 \times 55 \times 96$	16	64	64
Fire		$55 \times 55 \times 128$	16	64	64
Fire		$55 \times 55 \times 128$	32	128	128
Maxpool	2	$55 \times 55 \times 256$			
Fire		$27 \times 27 \times 256$	32	128	128
Fire		$27 \times 27 \times 256$	48	192	192
Fire		$27 \times 27 \times 384$	48	192	192
Fire		$27 \times 27 \times 384$	64	256	256
Maxpool	2	$27 \times 27 \times 512$			
Fire		$13 \times 12 \times 512$	64	256	256
Conv	1	$13 \times 13 \times 512$			
Avg pool	1	$13 \times 13 \times 1000$			

3.2. MobileNet

MobileNets are a small type of network that are developed at Google Inc. These networks aim at cutting computational power, memory usage, and storage utilization. Currently, two MobileNets are getting a lot of attention, MobileNetV1, and MobileNetV2. We will go over both, and see what they do and what is different between them.

3.2.1. MobileNetV1

The core layers of MobileNetV1 [31] are called depthwise separable filters. These filters are based on depthwise separable convolutions which split a standard convolution (Figure 3.2) into a depthwise convolution (Figure 3.3) and a 1×1 (pointwise) convolution (Figure 3.4). The parameters used in these images are as follows:

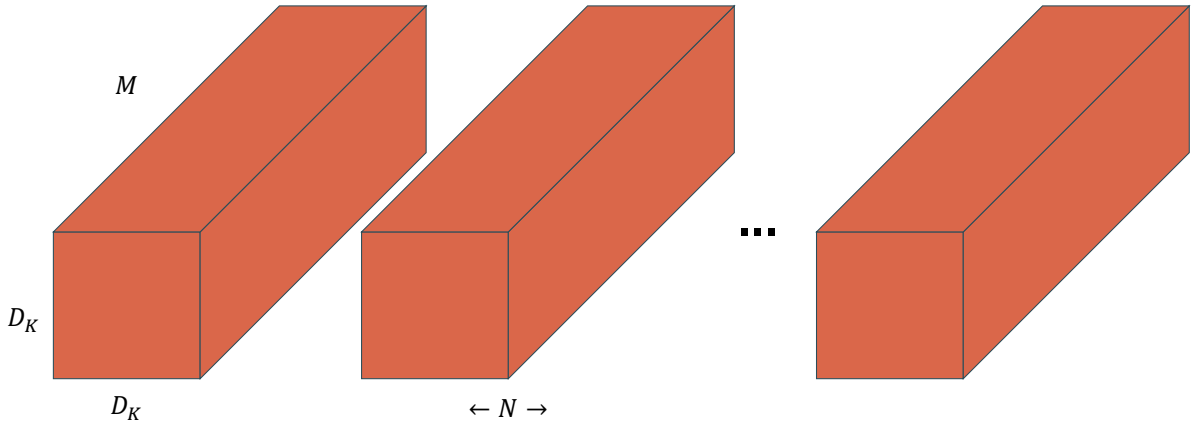

Figure 3.2: Standard Convolution.

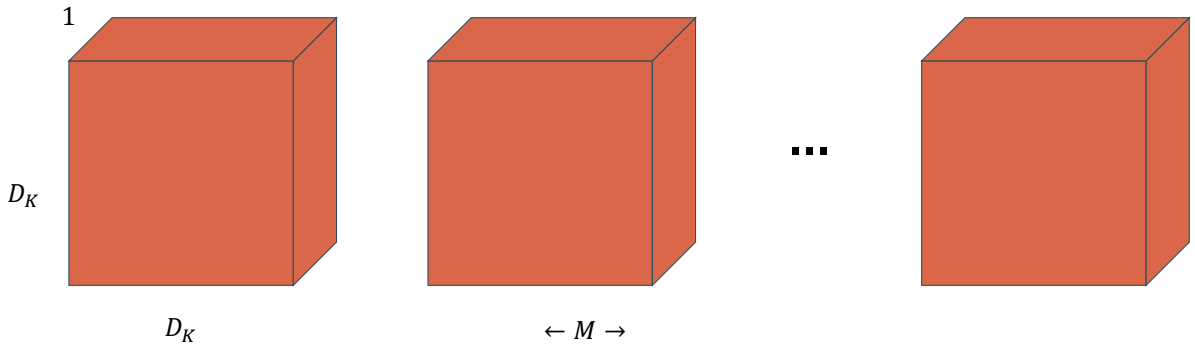

Figure 3.3: Depthwise Convolution.

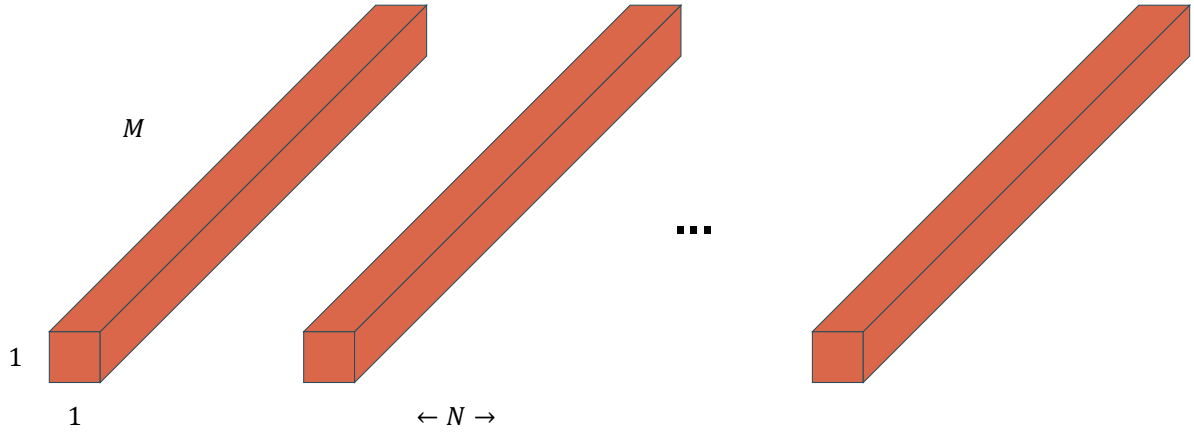

Figure 3.4: 1×1 (pointwise) Convolution.

- M - The number of input channels.
- D_K - The kernel size.
- N - The number of output channels.

A standard convolution takes $D_F \times D_F \times M$ as a feature map and generates a $D_F \times D_F \times N$ feature map. D_F represents the spatial width and height of the feature map. This results in a standard convolution that has a computational cost of Equation (3.3), which is dependent on the kernel size (D_K), the number of input channels (M), the number of output channels (N), and the feature map size (D_F).

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{3.3}$$

To reduce the computational intensity, it is possible to split this normal convolution via the use of factorized convolutions, which are called depthwise separable convolution. This depthwise separable convolution exists of two convolutional layers, depthwise convolution, and pointwise convolution.

The computational intensity of depthwise convolution is given in Equation (3.4) and is extremely efficient relative to normal convolution, however, it does not combine input channels to create new features, it only filters them.

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \tag{3.4}$$

To generate new features a second layer is introduced called the 1×1 or pointwise convolutional layer. This pointwise convolutional layer has a computational intensity of Equation (3.5), and introduces new features.

$$M \cdot N \cdot D_F \cdot D_F \tag{3.5}$$

If we combine these two layers, so we filter and generate new features, we get depthwise separable convolution which has a computational intensity of Equation (3.6).

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{3.6}$$

So by splitting our normal convolutional layer into two separate layers we can achieve a reduction in the computational intensity of Equation (3.7).

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{3.7}$$

With MobileNet using 3×3 kernels, this results in an 8 to 9-time reduction in the computational intensity with only a small reduction in accuracy of about 1%.

MobileNet has also two hyper-parameters called a *width multiplier* (α) and a *resolution multiplier* (ρ). The width multiplier can be used to thin the MobileNet even more extensively. This is done uniformly across all layers. With the number of input channels M of a layer and a width multiplier α, we get

αM. Similarly, we see the number of output channels N with width multiplier α becomes αN. We can then derive the computational intensity of a separable convolutional layer with width multiplier α in Equation (3.8).

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F \tag{3.8}$$

The baseline MobileNet has $\alpha = 1$, but with $\alpha \in (0, 1]$, reduced MobileNets are possible with common values being 0.75, 0.5, and 0.25. These lower width multipliers will reduce the computational intensity, and size, but also accuracy.

The second hyper-parameter introduced, the resolution multiplier, is applied to the input image, and the internal representation of all the layers. With the resolution multiplier $\rho \in (0, 1]$ and the width multiplier we have a computational intensity for the separable convolutional layers of Equation (3.9).

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F \tag{3.9}$$

The general architecture of the MobileNet network can be found in Table 3.2. From this table, it can be seen that MobileNet is build using normal convolutional layers combined with separable depthwise convolutional layers. Also after each convolutional layer there is a batch-normalization layer and a ReLU layer as shown in Figure 3.5.

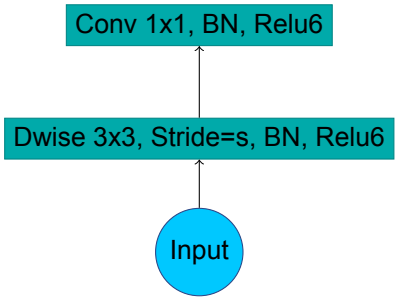

Figure 3.5: Depthwise Separable Convolution module.

Table 3.2: Architecture MobileNetV1.

Type	Stride	Filter Shape	Input Size
Conv	2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw	1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv	1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw	2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv	1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw	1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv	1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw	2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv	1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw	1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv	1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw	2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv	1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw	1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw	2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv	1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw	2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv	1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool	1	Pool 7×7	$7 \times 7 \times 1024$
FC	1	1024×1000	$1 \times 1 \times 1024$
Softmax	1	Classifier	$1 \times 1 \times 1000$

3.2.2. MobileNetV2

The new MobileNetV2 [36] continues with the general idea of MobileNetV1 as it also uses depthwise separable convolution layers and the two hyper-parameters, width multiplier and resolution multiplier. The main novel improvement in MobileNetV2 is the addition of a new module: the inverted residual with linear bottleneck. The general idea behind this module is that it takes a low-dimensional compressed input which is expanded into a high-dimensional input and filtered with lightweight depthwise convolution. The features are then projected back to a low-dimensional representation using a linear convolution. This works on the principle that ”manifolds of interest” could be embedded in the low-dimension subspace of a neural network. Two properties are indicative of the requirement this manifold of interest should lie in a low-dimensional subspace of higher-dimensional activation spaces. The properties are [36]:

- If the manifold of interest remains non-zero volume after the ReLU transformation, it corresponds to a linear transformation.
- ReLU is capable of preserving complete information about the input manifold, but only if the input manifold lies in a low-dimensional subspace of the input space.

Based on these two properties, we can insert linear bottleneck layers into our convolutional blocks. And we use linear layers because non-linear layers would destroy too much information as shown in various works [27, 36].

Finally, just as in classical residual networks, shortcuts are added to the module. These shortcuts are added to improve the propagation of the gradient, but in contrast to classical residual networks, these shortcuts are inverted which greatly reduces memory usage and even result in better accuracy. Figure 3.6 shows a schematic overview of this inverted residual with linear bottleneck and Table 3.3 shows the layers of which MobileNetV2 consists of.

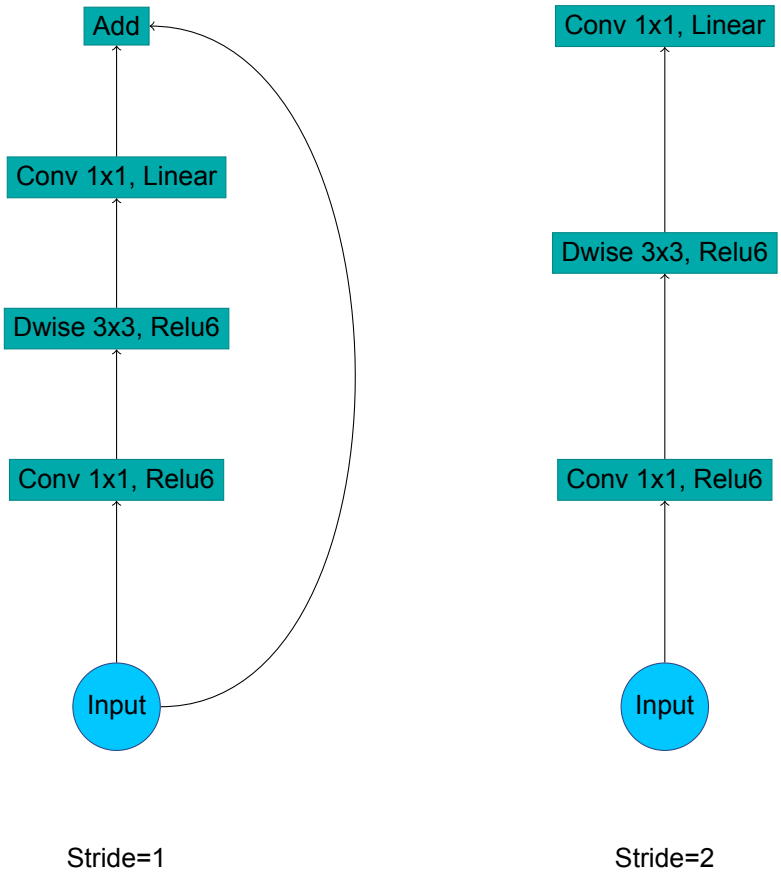

Figure 3.6: Inverted residual with linear bottleneck for stride = 1 (left) and stride = 2 (right).

Table 3.3: Architecture MobileNetV2 with n indicating the number of identical repetitive blocks.

Type	Stride	n	Input Size
Conv	2	1	$224 \times 224 \times 3$
Bottleneck	1	1	$112 \times 112 \times 32$
Bottleneck	2	2	$112 \times 112 \times 16$
Bottleneck	2	3	$56 \times 56 \times 24$
Bottleneck	2	4	$28 \times 28 \times 32$
Bottleneck	1	3	$14 \times 14 \times 64$
Bottleneck	2	3	$14 \times 14 \times 96$
Bottleneck	1	1	$7 \times 7 \times 160$
Conv	1	1	$7 \times 7 \times 320$
Avg Pool	-	1	$7 \times 7 \times 1280$
Conv		-	$1 \times 1 \times 1280$

3.3. ShuffleNet

ShuffleNets [34, 41] utilize the efficient model design from SqueezeNet, depthwise separable convolution from MobileNet, and adds a *channel shuffle* operation to help information flow across feature channels.

Just like MobileNet, ShuffleNetV1 [41] uses depthwise separable convolution, but ShuffleNet also utilizes *group convolutions* which are significantly more efficient than normal convolution. The idea of group convolution is to increase the number of kernels per layer to learn more intermediate features. This increases the number of channels in the next layer. The improved efficiency comes from the possibility to use these kernels in parallel.

One way of implementing group convolution in pointwise convolutional layers is by stacking multiple group convolution layers together, as can be seen in Figure 3.7. The issue with this is that the outputs from a group only relate to the inputs from that same group and thus weakens the representation.

Figure 3.7: Regular Grouped Convolution.

The proposed solution in ShuffleNet is to add a *channel shuffle* operation. The shuffle operation, as shown in Figure 3.8, allows information from different groups to flow to other groups. This way the input and output channels will be fully related.

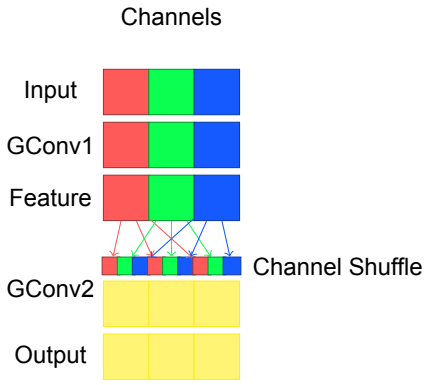

Figure 3.8: Grouped Convolution with a channel shuffle operation.

ShuffleNet proposes a new *ShuffleNet Unit* which is derived from a standard bottleneck module (similar to Figure 3.6 with stride = 1). We then replace the first pointwise convolution with a group convolution followed by a channel shuffle. This is then followed by a standard 3×3 depthwise convolution and finally, the linear bottleneck layer is replaced with a group convolution to recover the channel dimension to match the shortcut path. This is shown in Figure 3.9 (left). When ShuffleNet is used with a stride, two changes have to be made, first, an average pool is added on the shortcut path, and second, the addition is replaced with a concatenation Figure 3.9 (right). A full ShuffleNet is comprised out of layers visible in Table 3.4.

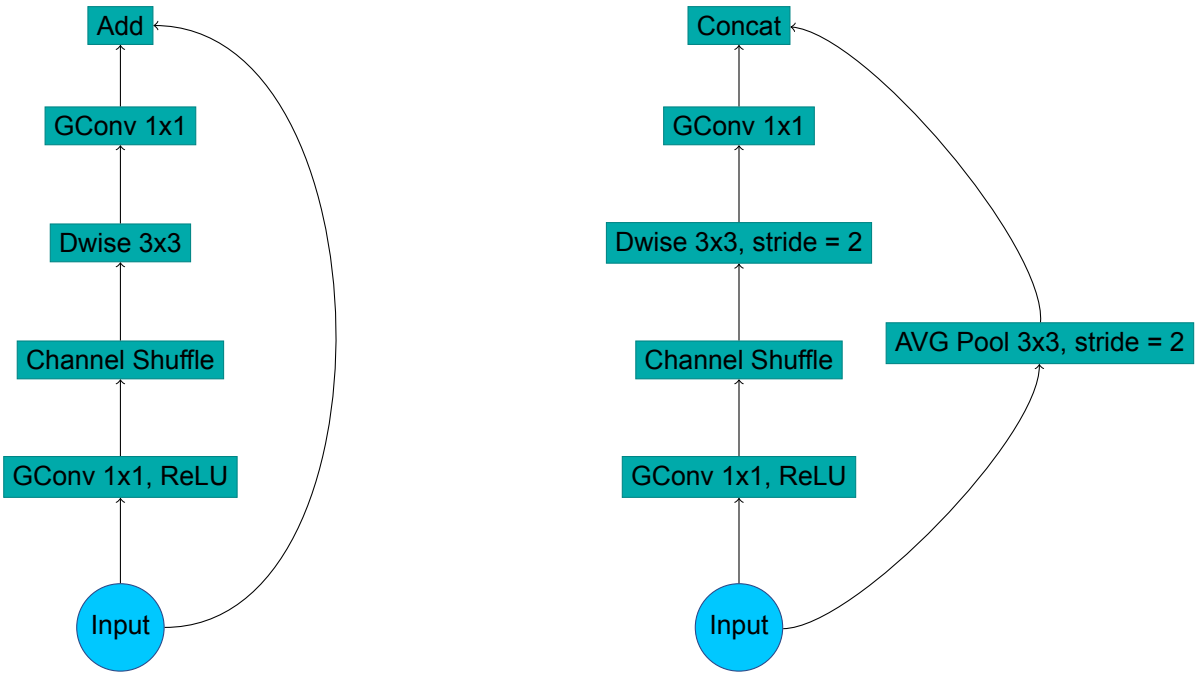

Figure 3.9: Schematic overview of the ShuffleNet unit for stride = 1 (left) and stride = 2 (right).

Table 3.4: Architecture of ShuffleNet with n being the number of repetitive units and the complexity given in number of trainable parameters.

Type	**Stride**	**Output Size**	n	**Output channels (g groups)**				
				$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 8$
Conv	2	112×112	1	3	3	3	3	3
Maxpool	2	56×56	1	24	24	24	24	24
ShuffleNet Unit	2	28×28	1	144	200	240	272	384
	1	28×28	3	144	200	240	272	384
ShuffleNet Unit	2	14×14	1	288	400	480	544	768
	1	14×14	7	288	400	480	544	768
ShuffleNet Unit	2	7×7	1	576	800	960	1088	1536
	1	7×7	3	576	800	960	1088	1536
GlobalPool		1×1						
FC				1000	1000	1000	1000	1000
Complexity				143M	140M	137M	133M	137M

Finally, it is also possible to give ShuffleNet a scale factor s which applies to the number of channels. Common given scale factors are $s = 1$, $s = 0.5$, and $s = 0.25$. A network is denoted as ShuffleNet $s\times$, so ShuffleNet $1\times$ has a $s = 1$.

4

Evaluating federated mobile machine learning networks

In this chapter, we will evaluate the implementation of MobileNetV2 and ShuffleNet into a federated environment. We will report mainly on their performance in regards to accuracy, and what we had to do to achieve these results.

4.1. Setup

Framework: For this evaluation, we opted to use PyTorch, because of its versatility and wide adoption in the academic community. We used, and adapted a small framework that was used to reproduce the results from [8].

Dataset: Three datasets have been used in this evaluation which are:

1. **MNIST:** MNIST is the easiest dataset and contains 60.000 images with handwritten digits for training and 10.000 images for testing/ verification. These images have a dimension of 28×28 and are black and white resulting in having only one color channel.
2. **CIFAR-10:** CIFAR-10 is a dataset that contains 60.000 images in ten different classes. The ten different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The number of images for each class is distributed equally thus there are 6.000 images per class. The CIFAR-10 images have a size of 32×32 and are colored resulting in CIFAR-10 images having three channels.
3. **ImageNette:** ImageNette is a subset of the more famous ImageNet dataset. ImageNet is a large database of images with more than 14 million images distributed over more than 20.000 categories. These 20.000 categories are distributed over several hundred top-level categories. An example of a top-level category might be *dog* with subclasses for each breed. ImageNette takes images from ImageNet with the same classes as CIFAR-10, but instead of 32×32, these images have a variable size and are multiple times bigger. These images have color resulting in three channels.

Machine learning models: Most of the evaluation is done on MobileNetV2 that we adapted for better performance in federated environments. Besides MobileNetV2, ShuffleNet was used to verify our approach to make MobileNetV2 ready for federated environments.

Platforms: The evaluation with the MNIST and CIFAR-10 datasets were done on Google Colab [5]. On Google Colab we used a GPU (Nvidia Tesla K80) as an accelerator to speed up training. For ImageNette we used the TUDelft GPU compute infrastructure for more performance during our training. This computing infrastructure was necessary because of the larger images which take more time to train a network.

4.2. Accuracy

MNIST: We first started with a plain MobileNetV2 that is pre-implemented in PyTorch [12]. A couple of changes were made to allow MNIST images to be processed because MobileNetV2 was built for the larger images from ImageNet. After training on the network, we instantly saw results similar to those from McMahan et al. [35] which trained a general-purpose convolutional network on the MNIST dataset. Both reach and accuracy of around $98\% - 99\%$, which is on par with non-federated environments.

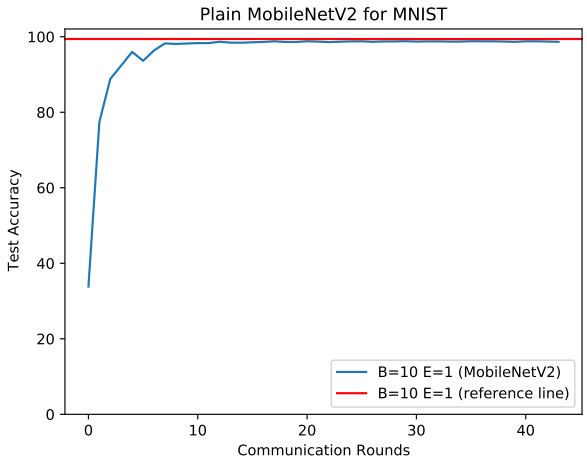

Figure 4.1: The accuracy during training of MobileNetV2 in a federated environment on MNIST. Included is a reference line from [35].

The results from this experiment are shown in Figure 4.1 where we have the accuracy in percentages on the y-axis, and the number of communication rounds on the x-axis. The results show us that for MNIST it is very straightforward to implement MobileNetV2 in a federated environment.

CIFAR-10: After a successful implementation with MNIST, it was now time to move on to CIFAR-10. The CIFAR-10 dataset is normally seen as a little bit more difficult to learn than MNIST, so this is the next logical step. So after modifying MobileNetV2 again to make sure it would now be compatible with CIFAR-10 images, we started training in a federated environment. The results of this were surprisingly bad, this can be seen in Figure 4.2 (red and magenta lines). After training, an accuracy of 70% for 10 clients, and 66% for 100 clients was reached. Even with only 1 client, which is a non-federated environment, we were only able to achieve an accuracy of 85%.

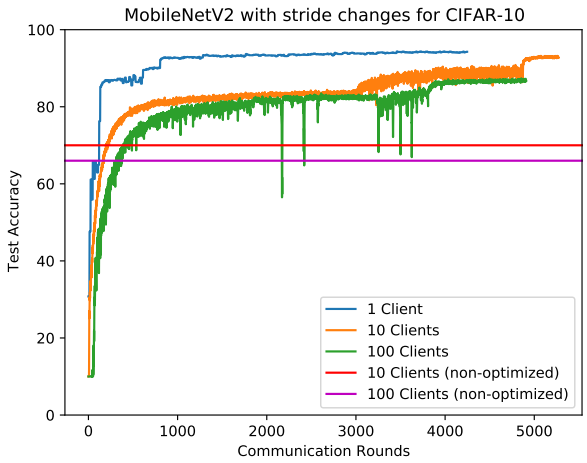

Figure 4.2: The accuracy during training of MobileNetV2 in a federated environment on CIFAR-10 for 1, 10, and 100 clients. Included are two reference lines from the non-optimized MobileNetV2.

To increase accuracy with the CIFAR-10 dataset, we looked at MobileNets that are optimized for training on CIFAR-10 and found [13]. This network can achieve an accuracy of 94.43% outside a federated environment. This new accuracy is significantly higher than the accuracy received with the original MobileNetV2, so we implemented this into our federated framework. However, the same happened again and our accuracy dropped from 94.43% to 79% and 73% for 10 clients, and 100 clients, respectively.

We then performed various experiments to identify the source of the drop in accuracy when using more than one client. One of the directions we investigated was to inspect the optimizations that were made in the CIFAR-10 optimized MobileNetV2. From this optimized network, we identified one specific optimization that might be particularly beneficial for accuracy. These are the stride changes in the first layer, the second block, and the third block as can be seen in Table 4.1. From this point, we refer to this new configuration *FedMobileNetV2*.

Table 4.1: Architecture MobileNetV2 compared our new FedMobileNetV2 architecture.

Type	**Old Stride**	**New Stride**
Conv	2	1
Bottleneck	1	1
Bottleneck	2	1
Bottleneck	2	1
Bottleneck	2	2
Bottleneck	1	1
Bottleneck	2	2
Bottleneck	1	1
Conv	1	1
Avg Pool	-	-
Conv		

Now only using this specific optimization, we do the same experiment again and see much better results. The results are visualized in Figure 4.2 where we see the accuracy on the y-axis and the number of communication rounds on the x-axis. Where our one client implementation got an accuracy of 94.39% which is only a 0.04% drop from the ideal non-federated MobileNetV2 we spoke about a bit earlier. If we look at 10 and 100 clients, we see that we can achieve accuracies of 93.1% and 87.2%, respectively. One thing to point out is that we do require a significant increase in the number of communication rounds compared to the MNSIT example in Figure 4.1. Another thing to point out is that we can see in Figure 4.2 that the lines are going up in terraces, this is because the learning rate has been lowered manually during the training process when the accuracy would not increase anymore.

ImageNette: One of our hypotheses for the accuracy jump with the increase in the stride size was that the smaller would lose too much information. The loss of information can be a problem for smaller images because they contain fewer data points overall. To test this hypothesis, we started training the original MobileNetV2 on the ImageNet dataset. Our hypothesis would hold when we notice a smaller drop in accuracy when doing federated training compared to the CIFAR-10 scenario.

The results of this experiment are shown in Figure 4.3. In this figure, we have the accuracy on the y-axis and the communication rounds on the x-axis. The results show what we expected, a smaller accuracy drop when training with ten clients. Where we have 88% accuracy for a one client configuration, we see, from our results, 80% accuracy for a ten client configurations and is only an 8% drop. This accuracy drop is significantly less than observed with the CIFAR-10 experiments.

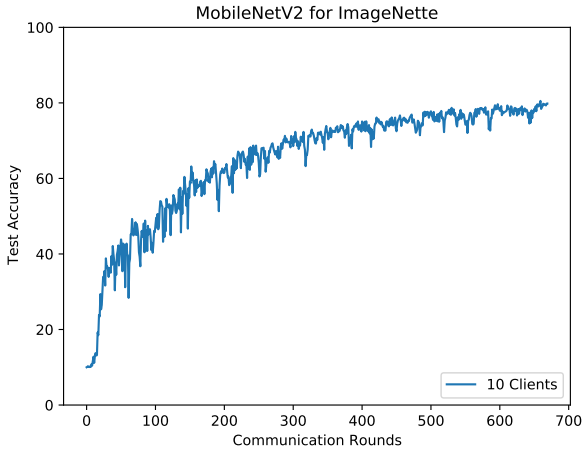

Figure 4.3: The accuracy during training of MobileNetV2 in a federated environment on ImageNette for 10 clients.

4.3. Designing mobile machine learning networks

It now appears to be possible to use MobileNetV2 in a federated setting, but is our technique only applicable on MobileNet, or is this technique universal, and can we apply this to other mobile machine learning networks too. Also, what are the consequences of increasing the stride this early in the network?

ShuffleNet: To test if changing the stride might hint towards a universal solution, we looked at ShuffleNet and tested its performance on MNIST and CIFAR-10. As expected, we saw that MNIST was no problem at all, but CIFAR-10 was losing a significant amount of accuracy again as can be seen in Figure 4.4 (left).

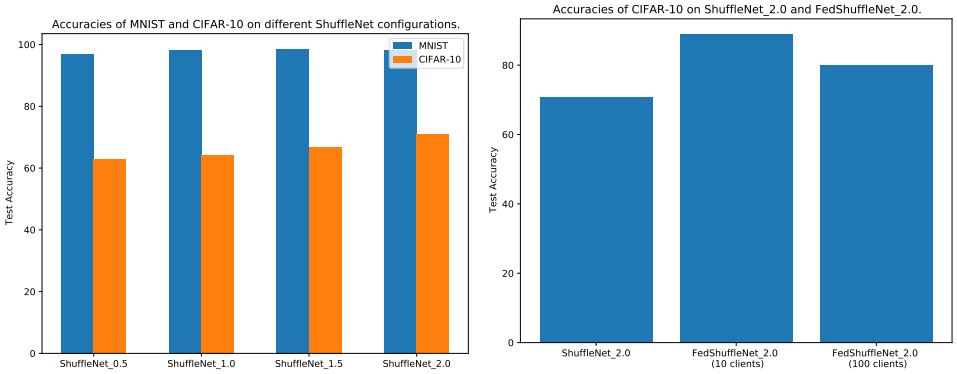

Figure 4.4: The accuracy during training of ShuffleNet on MNIST and CIFAR-10 for 1 client (left) and the accuracy during training of ShuffleNet_2.0 vs. FedShuffleNet_2.0 with 10 and 100 clients (right).

If we change the stride of the first layer as we did in FedMobileNet, we create a new configuration called FedShuffleNet. The CIFAR-10 performance of this new configuration can be seen in Figure 4.4 (right). We see an instant increase in the accuracy even over non-federated ShuffleNets. ShuffleNet_2.0 achieves an accuracy of 70.9% with only one client, in comparison, FedShuffleNet_2.0 achieves an accuracy of 88.9% and 79.9% for 10 and 100 clients, respectively.

Disadvantages: The disadvantage of the method we apply is that we increase the amount of memory needed to do training and inference. For FedMobileNet we require more than $10\times$ the amount of memory and go from $3MB$ to $37MB$. For FedShuffleNet, the difference is less extreme, because we only changed the stride for the first layer and did not touch the Shuffle Unit. For ShuffleNet we go from $20MB$ to $28MB$.

The increase in memory usage is not necessarily an issue because modern devices normally have gigabytes available. And this does give us control over accuracy within a resource budget. For memory-constrained environments, it is possible to reduce accuracy and improve resource usage. If accuracy is extremely important, it is possible to increase the memory budget and achieve higher accuracies.

5

Mobile machine learning frameworks

Mobile machine learning frameworks are frameworks that are capable of performing machine learning tasks on *mobile devices*. Mobile devices in this context will relate to mobile phones and tablets. In this chapter, we will go over a couple of mobile machine learning frameworks we have tried, give some background information, and tell you what they can or cannot do at this moment. These frameworks are being actively developed, and therefore might at the moment of reading be more capable than described here.

5.1. HiAi

HiAi [9] is being developed by Huawei and proposes a ”three-layer” open architecture. These three layers are:

- **Chip:** it is possible to convert existing models and achieve high performance by utilizing NPU (Neural Processing Unit) optimizations.
- **Device:** it allows the app to use AI and make it more intelligent.
- **Cloud:** it can push services that are based on users' requirements and do this in a proactive way.

HiAi also provides tools to convert, optimize, visualize, and simulate models. One of these tools enables the possibility to execute the model on a hosted mobile device.

Operational information: HiAi can only be used for Android devices and for this reason, HiAi can be used with either Java or Kotlin. HiAi does have the capability to convert models from the following formats to use in HiAi applications:

- **Huawei MindSpore [11]:** this is an AI framework developed by Huawei and is intended to ”bridge the gap between AI and applications”.
- **TensorFlow [21]:** this is an AI framework developed by Google and is one of the most complete and most used AI frameworks. It lets researchers push the state of the art while letting developers build and deploy models with ease. See more about TensorFlow at Section 5.2.
- **Caffe [4]:** this is an AI framework developed by Berkeley AI Research and community contributions. Caffe is developed using an expressive architecture, extensible code, optimized for speed, and community.
- **Paddle [1]:** this is developed by PadlePadle and they pride themselves on being the only independent R&D deep learning platform in China. This platform advertises four *leading technologies*, which are: an agile framework for industrial development of deep neural networks, support ultra-large-scale training of deep neural networks, accelerated high-performance inference over ubiquitous deployments, and industry-oriented models and libraries with open source repositories.

Capabilities: For capabilities we will look at three different areas. First, we will look at inference capabilities, then, we will look at on-device training capabilities, and finally, we will evaluate quantization.

- **Inference:** HiAi fully supports inference on mobile devices. All the networks we tested fully worked, so we feel strongly that most if not all networks will work. For some model conversions, it is however necessary to have the source framework at a specific target level.
- **On-device training:** HiAi at the moment does not support any on-device training. After contacting Huawei, they mention this is an area that they are investigating, and quite likely, on-device training will be supported in the (near) feature.
- **Quantization:** at this point, post-training quantization is supported with convolution, fully connected, and depthwise convolution operators. This includes weight quantization, offset quantization, and data quantization. There is one weight quantization level available, which is fp32 to int8.

Optimizations: HiAi is optimized for high throughput inference on some Huawei devices. The devices currently supported are the newer high-end smartphones that have a dedicated NPU, and it is likely that more Huawei devices in the future will include a dedicated NPU to efficiently do machine learning.

5.2. TensorFlow Lite

TensorFlow Lite [22] is the mobile variant of TensorFlow [21] designed by Google Inc. TensorFlow Lite is designed to be extremely efficient on mobile and embedded devices. It claims that TensorFlow Lite models introduce optimizations and reduce file size while achieving the same accuracy as normal TensorFlow models.

Operational information: TensorFlow Lite can be used on Android with Java, iOS with Objective-C, Embedded Linux with either Python or C++, and Microcontrollers with an experimental C++ framework. While a lot of operations are supported in TensorFlow Lite, some operations might be missing and can be imported using the experimental library TensorFlow Select. Using TensorFlow Select will increase the size of the binary as these operations are not optimized for mobile or embedded use. The TensorFlow Lite model converter can convert the following model types from TensorFlow V2 to its efficient model representation:

- **SavedModel directories:** a TensorFlow SavedModel directory contains a complete TensorFlow Program which includes weights and computation. The advantage is that it does not require the original code to build the model to run, so this is a useful format for sharing or deploying the model.
- **Keras models [10]:** is an AI library that acts as an interface for TensorFlow. It promotes itself as a simplification layer between TensorFlow and the end-user by offering a simple API that minimizes the steps to develop a model, provides clear error messages, and extensive documentation.

Although the number of direct conversion options is limited, it remains possible to convert a model to a TensorFlow model first and then convert it to a TensorFlow Lite model.

Capabilities: For capabilities we will look at three different areas. First, we will look at inference capabilities, then, we will look at on-device training capabilities, and finally, we will evaluate quantization.

- **Inference:** as we mentioned a bit earlier, mobile inference is completely supported. Not everything is supported by TensorFlow Lite, but everything that is not supported will be available in TensorFlow Select. The only disadvantage is that these imported operations are not optimized for mobile execution and increase the model size.
- **On-device training:** TensorFlow Lite, at this moment, does not support on-device training. It is unclear whether TensorFlow is working on implementing on-device training at the moment, but it is unlikely that we see a fully supported version any time soon.
- **Quantization:** TensorFlow Lite does support post-training quantization from fp32 to either half-precision (fp16) or int8.

Optimizations: While TensorFlow Lite does not have any hardware-specific optimizations, it does optimize the network with general optimizations when converting from other networks. Also, TensorFlow has an optimization toolkit that can be used to optimize TensorFlow lite networks.

5.3. PyTorch Mobile

PyTorch Mobile [19] is the mobile variant for PyTorch [18] designed by Facebook Inc. PyTorch tries to remove the friction between developing a model and deploying it on mobile or embedded devices. It tries to provide an end-to-end workflow that simplifies the entire process. It also wants to focus on privacy-preserving learning techniques, like federated machine learning.

Operational information: PyTorch Mobile is available for iOS in Swift, Android in Java, and Linux Embedded devices in C++. PyTorch Mobile can convert all regular PyTorch models by tracing the model using TorchScript. TorchScript is a representation that enables the transformation from Python (the natural language of PyTorch) to C++, Java, or Objective-C.

Although there is no dedicated converter tool to convert machine learning models to PyTorch Mobile format we can still convert all models to the regular PyTroch format, which we can then convert to PyTorch Mobile models.

Capabilities: For capabilities we will look at three different areas. First, we will look at inference capabilities, then, we will look at on-device training capabilities, and finally, we will evaluate quantization.

- **Inference:** PyTorch Mobile is still in beta, so while most operations are supported, some API changes may occur with version upgrades.
- **On-device training:** this is currently not supported, but PyTorch Mobile mentioned privacy-preserving techniques like federated learning as one of its advantages, so it is quite likely they are working on this. So future support is probably imminent.
- **Quantization:** post-training quantization is supported in the form of fp32 to int8 conversion. This includes support for per-channel quantization, dynamic quantization, and per-tensor quantization.

Optimizations: PyTorch Mobile has, besides quantization, dedicated optimization build-in in the form of build level optimizations and selective compilation. They do not optimize for any specific device, however, they mention support for backends like GPU, DSP (Digital Signal Processing), and NPU will be available soon. PyTorch Mobile is also optimized for ease of use with a PyTorch end-to-end workflow.

5.4. PySyft

PySyft [2] is a Python library developed by OpenMined [16] and is specially designed for secure and private Deep Learning. Build-in features enable federated learning, Differential Privacy, and encrypted Computation.

Operational information: PySyft has an implementation for iOS in Swift, Android in Kotlin, desktop in Python, and web in JavaScript. PySyft does not convert models but traces them by hooking into third-party backends. The supported backends are:

- **PyTorch:** as discussed in Section 5.3, PyTorch is a Python AI framework developed by Facebook. It is arguably the most used AI framework with support for almost any operator imaginable.
- **TensorFlow:** as we discussed earlier in Section 5.2, TensorFlow is a very extensive AI framework with support for many operators, lots of tools, and excellent documentation.

PySyft can also use PyGrid [17], another library that is developed by OpenMined. PyGrid can be used to set up peer-to-peer networks, which can be used to share models easily and efficiently. This can be very effective in building federated learning applications.

Capabilities: For capabilities we will look at three different areas. First, we will look at inference capabilities, then, we will look at on-device training capabilities, and finally, we will evaluate quantization.

- **Inference:** PySyft currently supports inference on mobile-devices without limitations.
- **On-device training:** this is currently partially supported. For very simple networks it is possible to on-device learning, but a lot of operations are not supported at the moment. This framework is under active development by OpenMined and the community, and we expect the number of supported operations to keep increasing.
- **Quantization:** at the moment PySyft does not support any form of quantization. Some contributors are looking to implement some form of weight quantization, but nothing is available at the moment of writing.

Optimizations: PySyft does not optimize for performance on mobile devices, it does however optimize for privacy with support for Differential Privacy, and encrypted computation with Multi-Party Computation and Homomorphic Encryption.

5.5. CoreML

CoreML [6] is a mobile machine learning framework developed by Apple Inc. CoreML tries to optimize for on-device performance without making it more complicated for the developer. They also advertise privacy with encrypted models, and easy deployment with Cloud Deployment using CloudKit.

Operational information: CoreML is available for Apple devices only (iOS, MacOS, etc.). The language used for CoreML is Apple's Swift language, and some bindings for Objective-C are available. A helper tool called coremltools [7] is available to build, edit, and convert models from other frameworks. Frameworks it can convert from are:

- **TensorFlow:** (see Section 5.2)
- **PyTorch:** (see Section 5.3)
- **Keras:** (see Section 5.2)
- **Caffe:** (see Section 5.1)
- **Onnx [15]:** this is an open model format for machine learning model representations developed by Facebook and Microsoft. The idea behind ONNX is that you do not have to worry about downstream applications because you are using an open format that should be able to be interpreted.
- **SciKit Learn [20]:** this is an AI framework that includes a lot of machine learning routines that can easily be used by data scientists to carry out some predictive work on NumPy arrays.

If a model is available in another format, it is also possible to first convert it to TensorFlow or PyTorch, and then convert it to a CoreML model.

Capabilities: For capabilities we will look at three different areas. First, we will look at inference capabilities, then, we will look at on-device training capabilities, and finally, we will evaluate quantization.

- **Inference:** CoreML fully supports any inference operators.
- **On-device training:** this is mostly supported on widely used operations, however, on more complicated layers we might find unsupported (support) layers. Our intuition is that the current implementation is not intended to train entire models but only do specific layer retraining. This is impractical for academic research purposes, but this is in line with the expectations from practical applications.
- **Quantization:** this is supported in the form of weight quantization. Normal weights are fp32 and these can be quantized to fp16, int8, in7, in6, int5, int4, int3, int2, or int1. There are also multiple quantization modes available which are linear, linear symmetric, and kmeans LUT. Finally, it is also possible to control which layers in a network need to be quantified.

Optimizations: CoreML is highly optimized for Apple hardware, and especially for the dedicated hardware accelerator in the latest Apple-designed chips called the Neural Engine. The most recent chip, the A14 Bionic, includes 16 dedicated Neural Engine cores and a new machine learning matrix accelerator.

6

Experimental results - Mobile implementation

In this chapter, we will implement some mobile networks using real mobile devices to measure accuracy, classification performance, and on-device learning performance.

6.1. Setup

Framework: After analyzing the mobile machine learning frameworks in Chapter 5, we decided to do our experiments with CoreML. This decision was made because it has the best support for on-device learning, it has model conversion support from most other frameworks, and it has an iOS implementation. The iOS implementation was beneficial because we had an iPhone with accelerator that we can use to run experiments on.

App: The iOS app that is used to run the experiments is a modified version of [3] by Jacopo Mangiavacchi. The original app uses the MNIST dataset, but we had to change it to support CIFAR-10 and ImageNet images. I also changed replaced the network that it was using, with different versions of MobileNet.

Dataset: As mentioned in the previous paragraph, the datasets used in these experiments are CIFAR-10 and ImageNette (which is a subset with ten classes of ImageNet) to test accuracy (Section 6.3), classification performance (Section 6.4), and quantization performance (Section 6.5). For on-device learning performance (Section 6.6), the MNIST dataset was used.

Machine learning models: For accuracy (Section 6.3), classification performance (Section 6.4), and quantization (Section 6.5) MobileNetV2 was used. There is however a difference in the MobileNetV2 we use for CIFAR-10 and ImageNette. The model for CIFAR-10 is converted, using the coremltools conversion tool while the model for ImageNette is downloaded from the CoreML developer website. For on-device learning, a generic convolution neural network is used.

Hardware: The hardware that was used is an iPhone X with an Apple A11 Bionic with a Hexa-core 64-bit ($2\times$ High power core at $2.39GHz$ + $4\times$ low power core at $1.42GHz$) with a dual-core Neural Engine which acts as a hardware accelerator. This Neural Engine is, according to the Apple press release", able to process "600 billion" operations per second. For inference comparisons, we used a MacBook Pro Mid 2014 with a $2.6GHz$ dual-core Intel i5 processor.

Other Frameworks: For our experiments, we used CoreML, but if these experiments would be done using a different framework, performance results could differ. This is especially true for classification performance, because all frameworks tend to optimize mobile machine learning networks in their own way. Out of experiments we ran, most accuracy performances stayed the same when using different

kinds of frameworks. Also, the workflow and setup are likely to remain similar, especially for other frameworks on iOS. Frameworks on Android will need a slightly different setup, but the workflow will remain mainly the same.

6.2. Data usage

We spoke already about data usage at the beginning of this thesis and named it one of the advantages of federated learning. The question that remained is: is it even feasible to send only the model and not the images itself? In other words, how many images do we need to send to make it more efficient to just send the model. It is clear that if we need to send thousands of images, it would become impossible to design a use-case for this federated learning routine and save bandwidth in the process. So what we are looking for, is the intersection between the size of sending the model once and the size of sending x images (see Equation (6.1)).

$$model\_size \leq x \cdot image\_size \tag{6.1}$$

An MNIST image has a size of about $550B$, whilst MobileNet has a size of $9.1MB$. This would mean that we can send 18200 images and be just as efficient as sending the model to the server once. We can do the same for CIFAR-10, whose images are about $2KB$, so that means we can send 4550 images and still be as efficient as sending just the model to the server. This has been plotted in Figure 6.1 (left).

This would indicate that a relatively large amount of images would be required to make federated learning more data-efficient than regular machine learning. MNIST and CIFAR-10, however, contain very small images and it is very unlikely that real-world use cases would use images this tiny. For this reason, we also experimented with ImageNette. ImageNette images have a more realistic size, so this gives us a more realistic representation of a real-live environment.

ImageNette images are on average $150KB$ (about $75\times$ bigger than CIFAR-10) and because the images of ImageNette are bigger than CIFAR-10 or MNIST images, we also see an increase in the size of MobileNet. A MobileNet for ImageNette is about $14MB$ (about $1.5\times$ larger than for CIFAR-10) and it becomes clear that when the image size increases, the size of the network does not increase linearly with it. So if we then look at how many images we would need to send to be as efficient as sending the model once, we see that the number drops significantly to approximately 93. For Squeezenet1_0, with a size of $4.7MB$, this would drop down even more towards approximately 31 images. To compare this result with a widely used ”normal” network, VGG16 [38] which has a size of about $574MB$, we see that we would require at least 3826 images to be as efficient. This shows that mobile machine learning networks do have the potential to be used in bandwidth-constrained use-cases, where regular networks would struggle to fulfill the requirements.

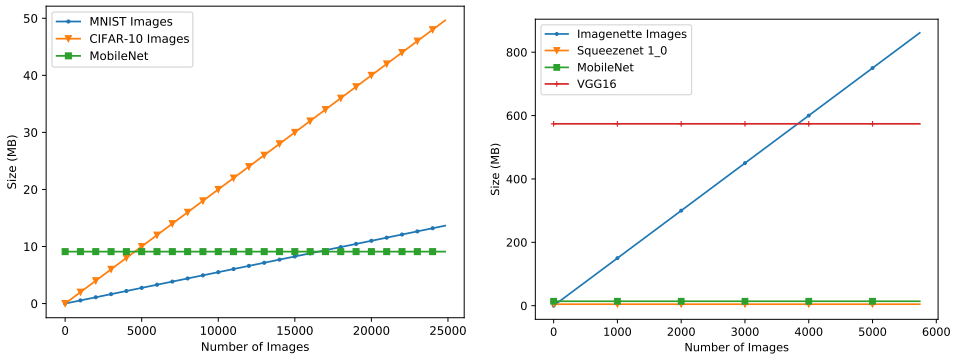

Figure 6.1: A comparison of (left) how many images (MNIST, CIFAR-10) one can send compared to sending the model (MobileNet) once. And a comparison (right) of how many images (Imagenette) one can send compared to sending the model (Squeezenet, MobileNet, VGG16) once.

6.3. Accuracy

To measure accuracy, we implemented two different models into our application. First, we downloaded a MobileNetV2 model from the Apple Developer website [14], which is a pre-trained model in the CoreML format *.mlmodel*. This model is trained on the ImageNet dataset, so for accuracy, we use

ImageNette (a subset of ImageNet). Second, we trained our, for federated learning optimized, MobileNetV2 on CIFAR-10 and converted that model, using CoreMLTools to a *.mlmodel*. In Listing 6.1 you can see a code-sample used to convert PyTorch models to CoreML models.

```
import torch
import coremltools as ct
from torch import nn

# Load the PyTorch model.
model = torch.load(
    "model.pth",
    map_location=torch.device(torch.device('cpu')),
)
model.eval()

# Create a sample input for tracing.
example_input = torch.rand(1, 3, 32, 32)
traced_model = torch.jit.trace(model, example_input)

# Define the image classes.
class_labels = [
    'Airplane', 'Autombile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck'
]


# Convert to Core ML using the Unified Conversion API
model = ct.convert(
    traced_model,
    inputs=[ct.ImageType(
        name="input_1",
        shape=example_input.shape,
        bias=[-1, -1, -1],
        scale=2/255.0,
    )],
    classifier_config = ct.ClassifierConfig(class_labels),
)

model.save("MobileNetV2.mlmodel")
```

Listing 6.1: Conversion script for PyTorch Models to CoreML models.

When we used the model from option 1, the pre-trained model downloaded from the Apple Developer website can achieve a Top-1 accuracy of 88%. This level of accuracy is equal to a similar implementation in PyTorch, so we can say that there is no accuracy loss in this case. If we look at option 2, where we have PyTorch model and convert it to a CoreML model using Listing 6.1, a noticeable drop in Top-1 accuracy occurs. Where we have a Top-1 accuracy of 89% in PyTorch, we only see a Top-1 accuracy of 77% in CoreML after conversion.

A possible explanation for this might be that the model from option 1 is converted from a TensorFlow model. TensorFlow models have a slightly different structure and are more similar to the overall structure of CoreML models. This could result in a more favorable conversion, however, this fell outside of the scope of this research and might be a good question for further research.

6.4. Classification performance

For classification performance, we used the same setup as for the accuracy performance in Section 6.3. We used the model from the Apple Developer website and our model that is converted from PyTorch. As mentioned in Section 6.1, we used an iPhone X to measure the performance that uses its Neural Engine as an accelerator. Having an accelerator means that it has specialized hardware that will *accelerate* the task that it needs to do. In this case, it means that it will do calculations on tensors from the neural network more efficiently than a CPU or GPU.

Table 6.1: The number of images classified per second.

Dataset	Images per Second
CIFAR-10	69
ImageNette	127.5

Table 6.1 shows how many images can be classified per second. In this table, we see that using our converted model from PyTorch on the CIFAR-10 dataset we achieve an average of 69 image classifications per second. This result means that it can be used in real-time photography and videography applications and do instant image recognition on these images. However, as we have mentioned earlier in Section 6.2, CIFAR-10 is a dataset that is too small for real-world scenarios. Because of this, we also experimented with ImageNette on the pre-trained MobileNetV2 model from the Apple Developer website.

Surprisingly, we saw that the performance of this model increased significantly to an average of 127.5 images per second. The increase in speed is unexpected because larger images contain more data and subsequently require more computations. Two factors might play a role in this result. First, we saw an increase in CPU usage for the ImageNette dataset. While with the CIFAR-10 dataset, we only saw a constant CPU usage of 50%, we saw a constant CPU usage of 70% for our ImageNette dataset. The increase in the CPU utilization leads obviously to higher performance. The second reason, which also might explain the increase in the CPU cap, is that the downloaded model is better optimized for CoreML. We already saw that higher accuracy was achieved with this model and now we also see better performance. An example of this might be the way a ReLU6 is performed in both models. Where in the model converted from PyTorch the ReLU6 is consolidated into one single layer, the ReLU6 from the downloaded model is split into four distinct steps. These steps are visible in Figure 6.2 and are:

- **ReLU:** a ReLU function looks as follows: $f(x) = max(0, x)$ and takes the maximum value between 0 and x. This means that we are left with 0 if x is below zero, and x if $x > 0$.
- **-1:** this function looks as follows: $f(x) = -x$ and negates all the values.
- **Threshold -6:** this function looks as follows: $f(x) = max(-6, x)$ and takes the maximum value between -6 and x. This means that we are left with x if the value is above -6, and -6 if the value is below -6.
- **-1:** this function looks as follows: $f(x) = -x$ and negates all the values, but because our values are negative already, they become positive again to complete the ReLU6.

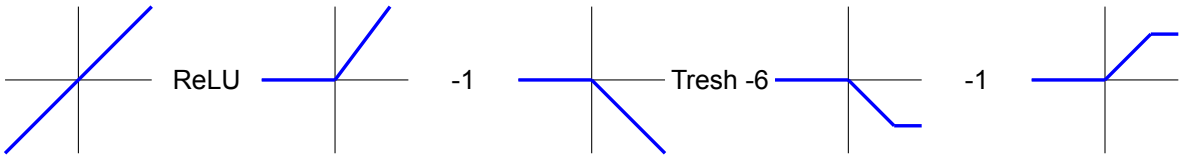

Figure 6.2: ReLU6 from MobileNetV2 for ImageNette dataset with separate steps.

6.5. Quantization

The weights in our models are stored in 32-bit floating-point (32fp) values. These 32fp values allow for precise calculations, but it is often not necessary to use these *full precision* values [23, 42, 44]. While we are now focusing on model size compared to the accuracy of the model, it is also possible to see a performance increase when quantizing a model. This performance increase would come from reduced computation on less complex values, but this is often hardware related. Some hardware is optimized to calculate with 32fp and does not profit from this reduction. On the other side, some hardware is only able to do *half-precision* 16-bit floating-point (16fp) calculations, so it is important to see what quantization does with the accuracy of the model.

With CoreML it is possible to quantize down from 32fp to 16fp, 8-bit integer (8b), 7b, 6b, 5b, 4b, 3b, 2b, and 1b. There are also three quantization modes available that distribute and scale the quantized values in a particular way. The three modes are:

- **Linear:** this uses linear quantization for the weights with a scale and a bias term.
- **Linear Symmetric:** this uses linear quantization for the weights but only using a scale.
- **Kmeans LUT:** this uses the Kmeans [29] algorithm to construct a lookup table for the quantization of the weights.

Figure 6.3 shows the results of quantizing MobileNetV2 with CoreMLTools using the Linear technique. We used the same setup as before, where we have our PyTorch converted model for the CIFAR-10 dataset and our pre-trained downloaded model for ImageNette. We can see that for CIFAR-10 we can go down to 4-bit integer values while still maintaining acceptable accuracy. Once we use 2-bit values, the accuracy goes over a cliff and the model becomes unusable. Quantizing the model reduces the size of the model from $9.1MB$ by $7\times$ to $1.3MB$.

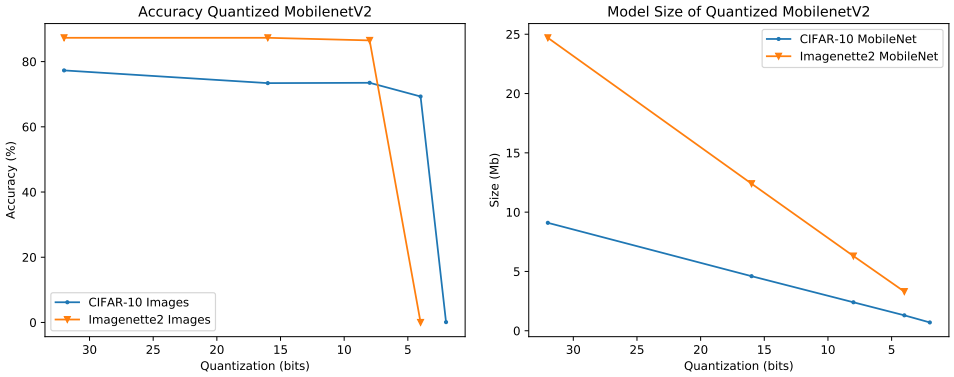

Figure 6.3: Shows the accuracy drop when reducing the number of bits representing the weights (left) and the size of the model with that same reduction of bits representing the weights (right).

Looking at MobileNetV2 for ImageNette, we see that we can reduce the model accuracy to 8-bit integers without losing significant amounts of accuracy. Here we see that if we go to 4-bit integers that the accuracy goes over a cliff, and the model becomes unusable. By Quantizing the model to 8-bit integers, we reduce the model size from $24.7MB$ by $3.9\times$ to $6.3MB$.

6.6. On-device learning

Because it is currently impossible to train a full MobileNetV2 within any of the frameworks we tested, we used a very simple convolutional network to train on the MNIST dataset. An architectural overview of this network can be found in Table 6.2, and this network contains 600.000 trainable parameters. The number of trainable parameters of this network is about $\frac{1}{5}th$ of the trainable parameters on MobileNetV2.

Table 6.2: An overview of the simple convolutional network used for on-device training.

Type	Stride
Conv	1
ReLU	
Pooling	2
Conv	1
ReLU	
Pooling	2
Flatten	
InnerProduct	
ReLU	
InnerProduct	
Softmax	

For training, we again used the iPhone X, and for comparison this time, we used a MacBook Pro Mid 2014. Keep in mind that the MacBook used only its CPU while the iPhone is allowed to use its Neural Engine as an accelerator. Furthermore, we will train the network for 10 epochs on both devices and compare the accuracies and runtime.

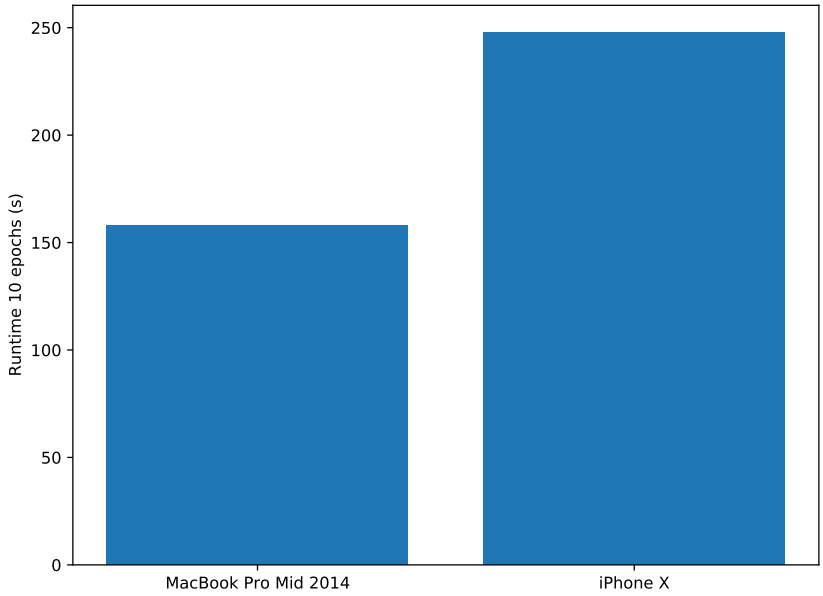

Figure 6.4: Shows the runtime of 10 epochs during training on our convolutional network (smaller is better).

In Figure 6.4 you can find the results of our 10 epoch training. On the y-axis, you see the runtime in seconds, and on the x-axis, you find the two platforms we have experimented with. From this graph, we see that the MacBook had a runtime of $158s$ compared to $248s$ for the iPhone. This is only a $1.6\times$ increase for a battery-powered mobile device.

If we look at the accuracy achieved by both, we see that we achieve a Top-1 accuracy of $99\%+$ for both platforms, which shows there is not any significant difference between these implementations.

7

Conclusion & further research

7.1. Conclusion

In this thesis, we strove at filling the void that exists in current research into mobile machine learning networks in a federated setting. Also, we aimed at providing a comprehensive overview of what is possible in terms of machine learning on mobile and embedded devices, and what performance could we see using mobile machine learning networks.

The thesis shows that mobile networks do not necessarily work, in their original form, in a federated setting, and we find that there is a nice balance that we can tune to optimize a network either for performance, accuracy, or a mix of both. We also see that this optimization is very dataset dependent, and that some datasets have fewer problems than others. This dependency occurs either because a dataset is very trivial like MNIST (which results in little problems for federation), or because images are too small, and thus they do not provide enough context like CIFAR-10. If we then use ImageNet images, which are larger, we see accuracy drops decrease again.

We continue this thesis with an overview of the mobile (on-device) machine learning frameworks. It is clear that this is a new area, and a lot of work still has to be done. While most frameworks fully support on-device inference, most do not support on-device training, and where we do see some support it is more towards doing single layer retraining than full model training. This is indicated by the fact that the main operators are supported while most support layers are not.

Finally, we looked at the performance of mobile networks on mobile devices using CoreML, a framework from the previously mentioned overview. We see that we can do inference with near-real-time performance and training with only a $2\times$ increase in training time.

Based on the findings in this thesis, it becomes possible to answer the research questions stated in Section 1.3:

- *How do smaller (mobile) machine learning networks perform in a federated environment?*

  Smaller mobile machine learning networks are seeing a recent uptick in popularity. Where we used to see an increasing interest in more complex and more accurate machine learning networks, we see that recent trends tend to design smaller machine learning networks that take resource usage into account. We also contemplate that it is possible to achieve acceptable accuracy while significantly reducing the size of the networks and thus also reducing the resource requirements to train such a network. We do see that these accuracies are achieved in perfect environments and that it is not trivial to achieve such accuracies in a federated setting. Especially on smaller images, we notice that there is not enough information there to achieve high accuracies. This, however, is not necessarily a problem caused by federation for mobile machine learning networks, since we also see this problem occur with mobile machine learning networks in a non-federated setting.

  To specifically answer the research question, smaller (mobile) machine learning networks perform generally a bit worse in a federated setting than in a normal training setting. This gets amplified when working on smaller images like CIFAR-10.

- *How can we ensure that smaller (mobile) machine learning networks are able to perform well in a federated environment?*

  We have noticed mobile machine learning networks lose accuracy when placed in a federated environment. To make these small (mobile) networks work in a federated setting, we notice that it might be necessary to tune some (hyper)-parameters and increase resource usage a little bit. The ability to tune these parameters does give us a unique opportunity to have some sort of a balance where we can tune a network to its exact needs. If we have a resource-constrained environment where accuracy is less important, we can start downsampling earlier and reduce the network size. On the other side, if we have some more resources available, we can downsample later in the network and increase the accuracy of the network.

  To specifically answer the research question, we can ensure smaller (mobile) machine learning networks keep performing in a federated environment by changing some of the hyper-parameters of the network. In particular, decreasing the stride parameters of the earlier layers/ blocks in the network proved very effective to improve accuracies.

- *Are frameworks ready for mobile (on-device) machine learning training and inference?*

  We see that on-device machine learning is a relatively new technology. It is not surprising that we only see on-device machine learning popping up now because machine learning is a computational and memory-intensive task. Mobile client devices are only becoming capable of doing such tasks within the last 3 to 6 years. We only have to look at the difference between the first Neural Engine in the iPhone X from 2017 which could perform 600 billion operations per second with the new Neural Engine in the iPhone 12 from 2020 that can perform 11 trillion operations per second. It is then also not surprising most AI frameworks are just starting with implementing on-device inference, and most have not even started with on-device training that is an even more computational intensive task than inference.

  To then explicitly answer the research question, most frameworks support full (on-device) inference or will in the very near future. Looking at on-device training, it is unlikely an application can use on-device training with the current frameworks. We will probably see this gap decrease in the next 2 to 3 years when frameworks are catching up with the new hardware that becomes available.

- *How well do mobile machine learning networks perform on mobile devices?*

  CoreML gave us a platform to run MobileNet on an actual mobile client-device. This gave us insights into how an actual handheld, battery-powered, client-device would fulfill the task of training and inference. As we have mentioned, these tasks were performed on a device from three years ago and new devices have $11\times$ more raw performance. Also, we could not do a full fletched training scenario because MobileNet is not fully supported for on-device training. The lack of this experimentation might influence the end-result because we were unable to test if memory would eventually become a bottleneck.

  To specifically answer the research question, we were positively surprised by the performance of our three-year-old device. Even on this device, we could see that inference would not be a problem at all and might even be used in near-real-time applications. The limited experiments we could do using on-device learning were promising with only doubling the time spend on a training task of 10 epochs.

7.2. Further research

Some areas which would benefit from further research are:

- **CoreML accuracy and efficiency:** we see that performance and accuracy are getting lost during model conversion in CoreML. This shows that the models on the Apple Developer website have undergone some optimizations that are not standard with the conversion tool. It will be interesting to see what operations are more performant than others and to see whether we can combine multiple operations to achieve a more performant bigger operation.

- **On device training:** as we have seen on-device learning is very much in its infant shoes, and it would be extremely useful if more research/ development would be spent on developing these frameworks to support or better support on-device learning. After this has been developed it would be interesting to see what the memory requirements do with training on a mobile device if larger images and bigger networks are used.
- **Distortion in prediction labels:** we got mobile networks working in a federated setting, but we only feed it relatively clean data. It would be interesting to see how these mobile networks would hold up if data with distortion would be provided. This would resemble a more real-life scenario where data is never perfect and distortion is very common.
- **Quantization 2.0:** after speaking with the Circuits and Systems group at TuDelft, it appears there is a big interest in quantization. While we only performed entry-level quantization, it would be interesting to see how more advanced forms of quantization would possibly further decrease network sizes while maintaining the same accuracy.

Bibliography

[1] Paddle github. URL https://github.com/PaddlePaddle/Paddle.

[2] Pysyft github. URL https://github.com/OpenMined/PySyft.

[3] App experiment environment, 2020. URL https://github.com/JacopoMangiavacchi/MNIST-CoreML-Training.

[4] Caffe developer website, 2020. URL https://caffe.berkeleyvision.org.

[5] Google colab, 2020. URL https://colab.research.google.com.

[6] Coreml developer website, 2020. URL https://developer.apple.com/machine-learning/core-ml/.

[7] Coremltools developer website, 2020. URL https://coremltools.readme.io/docs.

[8] Federated learning framework, 2020. URL https://github.com/shaoxiongji/federated-learning.

[9] Hiai developer website, 2020. URL https://developer.huawei.com/consumer/en/hiai.

[10] Keras developer website, 2020. URL https://keras.io.

[11] Huawei mindspore developer website, 2020. URL https://e.huawei.com/en/products/cloud-computing-dc/atlas/mindspore.

[12] Pytroch mobilenetv2 implementation, 2020. URL https://github.com/pytorch/vision/blob/master/torchvision/models/mobilenet.py.

[13] Mobilenetv2 cifar-10 optimized, 2020. URL https://github.com/kuangliu/pytorch-cifar.

[14] Mobilenetv2 apple developer, 2020. URL https://developer.apple.com/machine-learning/models/.

[15] Onnx developer website, 2020. URL https://onnx.ai.

[16] Openmined developer website, 2020. URL https://www.openmined.org.

[17] Pygrid github, 2020. URL https://github.com/OpenMined/PyGrid.

[18] Pytorch developer website, 2020. URL https://pytorch.org/.

[19] Pytorch mobile developer website, 2020. URL https://pytorch.org/mobile/home/.

[20] Scikit learn developer website, 2020. URL https://scikit-learn.org/stable/.

[21] Tensorflow developer website, 2020. URL https://www.tensorflow.org.

[22] Tensorflow lite developer website, 2020. URL https://www.tensorflow.org/lite/.

[23] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020.

[24] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[25] David Enthoven and Zaid Al-Ars. Fidel: Reconstructing private training samples from weight updates in federated learning. *arXiv preprint*, 2020.

[26] David Enthoven and Zaid Al-Ars. An overview of federated deep learning privacy attacks and defensive strategies. *arXiv preprint arXiv:2004.04676*, 2020.

[27] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5927–5935, 2017.

[28] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5353–5360, 2015.

[29] Kaiming He, Fang Wen, and Jian Sun. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2938–2945, 2013.

[30] 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*, pages 770–778, 2016.

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[33] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[35] 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*, pages 1273–1282. PMLR, 2017.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[37] Robert E Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[38] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[39] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016.

[40] Wil MP Van der Aalst, Vladimir Rubin, HMW Verbeek, Boudewijn F van Dongen, Ekkart Kindler, and Christian W Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87, 2010.

[41] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[42] Baozhou Zhu, Zaid Al-Ars, and Peter Hofstee. Nasb: Neural architecture search for binary convolutional neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020. doi: 10.1109/IJCNN48605.2020.9207674.

[43] Baozhou Zhu, Zaid Al-Ars, and Wei Pan. Towards lossless binary convolutional neural networks using piecewise approximation. *European Conference on Artificial Intelligence (ECAI)*, 2020.

[44] Baozhou Zhu, Peter Hofstee, Jinho Lee, and Zaid Al-Ars. Sofar: Shortcut-based fractal architectures for binary convolutional neural networks. *arXiv preprint arXiv:2009.05317*, 2020.

[45] Stefan Zwaard, Henk-Jan Boele, Hani Alers, Christos Strydis, Casey Lew-Williams, and Zaid Al-Ars. Privacy-preserving object detection & localization using distributed machine learning: A case study of infant eyeblink conditioning. *arXiv:2010.07259*, 2020.