Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Federated Machine Learning: A Distributed Approach to Pain Expression Recognition in Healthcare

Author:
Nicolas TOBIS

Supervisors:
Dr. Ognjen RUDOVIC
Prof. Björn SCHULLER

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

September 6, 2019

Declaration of Authorship

I, Nicolas TOBIS, declare that this thesis titled, "Federated Machine Learning: A Distributed Approach to Pain Expression Recognition in Healthcare" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a MSc degree at Imperial College London.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

"Deep-learning will transform every single industry. Healthcare and transportation will be transformed by deep-learning. I want to live in an AI-powered society. When anyone goes to see a doctor, I want AI to help that doctor provide higher quality and lower cost medical service. I want every five-year-old to have a personalised tutor."

Andrew Ng

"Geteiltes Leid ist halbes Leid."

German proverb

IMPERIAL COLLEGE LONDON

Abstract

Federated Machine Learning: A Distributed Approach to Pain Expression Recognition in Healthcare

by Nicolas TOBIS

Pain-monitoring is an essential task that hospital staff is required to perform on an ongoing basis. While evidence suggests, however, that improved pain monitoring yields better patient outcomes, competing demand for nursing staff has put a toll on the practical implementation of manual routine check-ups. In this work, we, therefore, address the problem of automated pain recognition from facial expressions. Pain recognition from image data is challenging since a classifier relies on very subtle changes in a test subject's facial expressions. Leveraging the "UNBC-McMaster shoulder pain expression archive database", a dataset consisting of >48k annotated video frames, we propose a lightweight CNN architecture that can be learned to recognize pain from image data to tackle this problem.

Building on this architecture, we show how federated learning, a distributed approach to machine learning, can be employed to allow multiple clients (e.g., hospitals) to jointly train such a model, without ever sharing their local data. Federated learning is very beneficial in a healthcare setting, where data regulations are strong, and data is often sparse.

We finally propose a novel algorithm that adds another level of privacy to the federated learning algorithm by further reducing the amount of information shared with a central server. Despite the limited amount of information shared between clients, our algorithm performs comparably to the standard federated learning algorithm and outperforms purely local models with no information sharing.

Submissions

This work has been submitted to the Workshop on Federated Learning for Data Privacy and Confidentiality (in Conjunction with NeurIPS 2019).

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Ognjen (Oggi) Rudovic, whose continued mentorship throughout the project proved invaluable to me. Engaging whiteboard-brainstorming sessions and strategy lunch-meetings continuously challenged me to explore new ideas, and helped shape this project substantially.

I would also like to thank my family and friends for feigning initial interest in stories about the latest model architecture I managed to implement and then gently redirecting the conversation to different topics.

Finally, I would like to thank the staff at Chapter Coffee, West Kensington for providing providing the most welcoming working environment and putting up with me every day for the past 3 months ordering a lavish 2 Americanos a day.

Ethical Considerations

This project involves video data collected from human participants. A team of researchers collected this data to compile a database for advancing pain research (see Chapter 3 for a more detailed outline). As such, we promise that the data has exclusively been processed for its intended use: Advancing the field of studying pain recognition from facial expressions. Since the UNBC-McMaster shoulder pain expression archive database was compiled exclusively for research purposes and is only available to researchers on request, we do not include it in any publicly available repositories of our work.

Moreover, we are aware of the growing environmental implications of machine learning research. Training machine learning models over many hours and days often require power-hungry GPUs that contribute to a growing demand for electricity worldwide. We therefore carefully designed our code to (a) leverage computing resources efficiently, and (b) launched experiments on large data sets, only after prototyping on smaller subsets of data.

Finally, rising privacy concerns regarding machine learning are, in part a motivation for this work, and we contribute to preserving individuals' privacy while training robust machine learning classifiers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Machine Learning

The not-so-recent-anymore rise of machine learning (ML) models has led to unprecedented advances in a broad array of fields. In the healthcare space, for example, employing deep learning has shown promise to increase the accuracy of pathological diagnoses[66][33]. In defeating Lee Sedol, widely regarded as the world's best player of the traditional Chinese game *Go*, Google's AlphaGo computer successfully showcased that computers powered by deep learning can achieve super-human performance[55]. Subsequent experiments with an enhanced algorithm dubbed AlphaGo Zero that defeated the original algorithm 100-0, and another algorithm defeating human champions in the highly complex real-time computer game StarCraft, helped further publicize the potential power of machine learning algorithms as a whole [56][64]. ML-powered algorithms are also becoming ever more present in our everyday lives, with voice assistants using speech recognition on mobile phones and in the home[15], and self-driving cars employing computer vision to guide us - most of the time - safely to our destination[4].

Today's popularity of machine learning in research and industry can largely be attributed to the unprecedented amounts of data people generate daily using their computers, credit cards, and most recently - mobile phones. According to one study, the average smartphone user interacts with his or her device a staggering 2,600 times per day[69], generating valuable data for advertisers [26], developers [32] and even medical researchers [23] with every tap.

1.1.2 Privacy

However, with billions of computing devices generating new data every day, new challenges and concerns arise. One example is privacy. Large centralized datasets that fuel modern ML models present a lucrative target for data breaches. The recent Facebook/Cambridge Analytica scandal has shown the impact that poorly protected user data can have [59]. As a consequence, governments have already started to act and passed legislation to protect their citizens, with *GDPR* in the European Union, the *Personal Information Security Specification* in China and the definitions of *focused collection* and *data minimization* proposed by the 2012 White House report on the privacy of consumer data in the United States. As a result, clients ordinarily generating data for an ML model might now object to, or be legally prohibited from providing even anonymized data to a central entity for data processing. Clients may include individuals generating data passively through interaction with their devices, as well as businesses deliberately sharing data with a centralized server for evaluation.

Moreover, also the practicality of training ML models on large raw datasets is running into constraints. Datasets are continually increasing in size as - in the case of mobile devices - millions of users generate information every single day. Compiling such datasets for evaluation requires a massive infrastructure on the server-side and strong upload capacity on the client-side. Besides, training a classifier on e.g., raw image data for a computer-vision algorithm from millions of users in a centralized location requires exceptionally high processing power, where even the largest tech companies such as Google and Facebook run into limitations.

At the same time, data in most industries is siloed. Owing to industry competition, privacy concerns, and bureaucratic administrative procedures, even data exchange within the same company is often heavily constrained. As a consequence, a new machine learning technique - federated learning (FL) - is gaining in popularity. First introduced by Google in 2016 [39][28][29], unlike traditional server-side machine learning models FL models are deployed to distributed devices (e.g. mobile phones), and learn locally. Since the training data never leaves the device, federated learning bears great promise for both increased privacy, as well as the distribution of computationally expensive tasks, increasing the training speed of ML models trained on large amounts of data.

1.1.3 Healthcare

While the potential applications for federated learning are numerous and highly diverse, the promise of a privacy-preserving, less computationally hungry machine learning approach can be particularly valuable for the healthcare space. Healthcare patient data are typically among the most strictly regulated with HIPAA[52] in the US and GDPR[16] in the European Union governing the rules by which such data can be accessed. At the same time, pooling healthcare data from healthcare facilities, insurance providers, and government agencies on a regional, national or even global scale holds enormous potential for example for rare diseases research[67] or treatment best-practices.

Pain

One example, where hospitals could work together to improve the lives of the patients they treat is the identification of pain. Detecting pain in patients to provide effective treatment is a critical job that hospital staff needs to perform on an ongoing basis, and the automation of this task has been of interest to researchers for quite some time [2][36][35]. Shortcomings so far, among other things, have included a lack of labeled training data for machine learning classifiers. If different hospitals, senior-care facilities, and other healthcare institutions collaborated in training a shared model for this task, the amount of available data would increase significantly, likely improving the performance of any classifying algorithm, in the process.

In "RoboChain: A Secure Data-Sharing Framework for Human-Robot Interaction" [10] the authors propose a framework to jointly learn a machine learning model on private, local data, building upon the latest advances in blockchain technology, and federated machine learning. Building on their work, this thesis further discusses the potential of federated learning in general, and as a catalyst for future breakthroughs in the medical space.

1.2 Contributions

Our work contributes to the field of pain study by way of monitoring *Facial Action Units*, as well as to the field of federated machine learning, by introducing a novel algorithm we dub *federated personalization*. More specifically, we:

1. Show that a lightweight CNN architecture can learn to recognize pain from the facial expressions of individuals.
   Based on a data set labeled according to the standard of the "Facial Action Coding System" introduced in chapter 3 we successfully train a convolutional neural network to predict when a person is in pain, using a video stream of that person's face as an input.

2. Show that the federated learning algorithm is robust even in a "production-level" setting, learning a challenging data set that changes over time.
   Many papers discuss the benefits of federated learning in benchmarking performance on toy-data sets such as MNIST or CIFAR-10[39][12]. In our case, we experimented with a highly unbalanced dataset, where positive and negative examples are not easily distinguishable. In our experiments, the underlying data distribution also evolves, as would be expected if we put the model into deployment.

3. Introduce a new federated learning algorithm that adds additional privacy preservation, at only a modest expense of performance.
   Our "federated personalization" algorithm only shares some layers with the central server, but not all, which makes it harder for an adversary to learn previously unknown information about an "honest" client participating in jointly learning a model. While these "global" layers are still averaged between participants, the "local" layers only continue training on each client's local dataset.

4. Provide specific directions for future research.
   We suggest a more sophisticated validation-set algorithm that leverages more precise early-stopping for a sparse data set such as the pain data set. We also recommend implementing a "fallback"-model in a federated setting that kicks in if an updated global model would likely worsen the performance of a given client.

1.3 Outcomes

We evaluated 24 different test subjects experiencing pain, split into two groups (group 1 and group 2). Group 1 was used to pre-train a model that we used as a baseline. Group 2 was used to continue training on the pre-trained model and evaluate model performance. Table 1.1 shows a summary of our key findings. The table compares the following methods for classifying the pain data set introduced in chapter 3:

RANDOM: A CNN where weights have been randomly initialized using the glorot uniform distribution [13].

BC-CNN: The **B**aseline **C**entralized **CNN**, a model that was pre-trained on the 12 test subjects of group 1.

	Weighted AVG + STD		
Experiment	**ACC**	**PR-AUC**	**F1**
RANDOM	43 ± 14	30 ± 15	31 ± 2
BC-CNN	72 ± 12	54 ± 23	47 ± 24
C-CNN (C)	75 ± 13	57 ± 21	49 ± 21
F-CNN (C)	74 ± 11	59 ± 23	51 ± 25
FP-CNN (C)	76 ± 12	56 ± 21	49 ± 23
FL-CNN (C)	75 ± 13	54 ± 20	47 ± 23

TABLE 1.1: Comparison of aggregated results on group 2 data for all learning algorithms with centralized pre-training in (%). Standard deviation is computed between test subjects.

C-CNN (C): The **C**entralized **CNN** with **C**entralized pre-training, a model that was initialized with the weights of the BC-CNN and trained with centralized learning and vanilla SGD.

F-CNN (C): The **F**ederated **CNN** with **C**entralized pre-training, a model that was initialized with the weights of the BC-CNN and trained with federated leraning.

FP-CNN (C): The **F**ederated, **p**ersonalized **CNN** with **C**entralized pre-training, a model that was initialized with the weights of the BC-CNN and trained on the federated personalized learning algorithm, introduced in chapter 5.

FL-CNN (C): The **F**ederated, **l**ocal learning **CNN** with **C**entralized pre-training, 12 individual models (one for each client), each initialized with the weights of the BC-CNN and trained separately, with the performance averaged across models.

As we can see from this table, the baseline for accuracy is beaten by every learning algorithm. When looking at PR-AUC and F1, both measures for identifying how well the model identifies positive examples, we find that FL-CNN (C) struggles to beat the baseline, while all other learning algorithms manage to outperform it. F-CNN (C) performs the best, followed by C-CNN (C) and FP-CNN (C), and finally, FL-CNN (C).

1.4 Outline

See the following for an outline of the remainder of this thesis:

Chapter 2: Background and related work This chapter introduces the necessary machine learning preliminaries that this thesis builds on. It explores neural networks in general, their strengths and weaknesses, convolutional neural networks, and finally, the federated learning algorithm.

Chapter 3: Data In this chapter we take a deep-dive into the data, and discuss the challenges the data set presented, as well as the techniques we used to augment the data and tackle these challenges.

Chapter 4: Model Architecture In this chapter, we present the underlying model architecture we used to assess our different learning algorithms. We show the original model architecture we started experimenting with as well as the final architecture that produced the best results. We also comment on the general viability of commonly employed model architectures for image recognition tasks such as ResNet50 and VGG19.

Chapter 5: Experiments This chapter discusses the different learning algorithms we tested, as well as the experimental settings we designed. It also introduces a novel algorithm we dub "federated personalized learning" that we designed for additional privacy protection.

Chapter 6: Results & Evaluation In this chapter we discuss the results that the different learning algorithms achieve, and analyze the absolute performance of each algorithm as well as their relative performance to each other.

Chapter 7: Conclusions & Future Work In the final chapter, we conclude our work and point to directions that future research can take. Expressly, we point towards possible advancements of our validation algorithm, as well as the federated personalized learning algorithm.

Chapter 2

Supervised machine learning

2.1 Machine Learning Preliminaries

Machine learning tasks can generally be grouped into two different paradigms: *Supervised* and *Unsupervised Learning*. In unsupervised learning, a typically unlabeled data set is fed to an algorithm, which is designed to detect previously unknown patterns in the data. Clustering algorithms such as *K-Nearest Neighbor*, or *Gaussian Mixture Models*[47] are examples of unsupervised learning, where unlabeled data is being grouped based on some shared properties. With supervised learning, on the other hand, labeled training data fed to an algorithm to learn a function that can map an input X to an output Y. Such a function can serve to solve a *classification* task (e.g., does an image contain a red car or a blue bus), or a *regression* task where a continuous variable is predicted (e.g., given X liters of gasoline, we expect a car to be able to drive for Y kilometers). The federated learning algorithm (described in detail in section 2.2) advances the field of *supervised learning*, as it allows multiple *clients* to jointly learn such a function. In the following, we will thus focus on supervised learning.

2.1.1 Multilayer Perceptron

While determining the amount of gasoline required to travel a certain distance could potentially be solved with a simple linear regression model, image classification tasks such as the one mentioned above typically require more sophisticated models. This has led to the increasing popularity of artificial neural networks (ANNs). Due to the typically much larger number of tunable parameters, ANNs can approximate significantly more difficult functions, qualifying them well for non-linear tasks such as speech recognition or object detection.

Multilayer perceptrons (MLP) are a type of feedforward ANN. The perceptron was first proposed by Rosenblatt in 1957 [49]. A multilayer perceptron consists of one input layer $\boldsymbol{x}$, one to many to hidden layers $\boldsymbol{h_i}$ and an output layer $\boldsymbol{y}$. For simplicity, we will assume a variant of the MLP, the single-layer perceptron in the following. Figure 2.1 shows an example of such an architecture. When training the MLP, we feed a sample of our training data to the input layer, where each neuron represents a feature (e.g., one pixel of an image, or one column labeled "age" of a table containing user-data). Each of the input layer (an n-dimensional vector where $\boldsymbol{x} \epsilon \mathbb{R}^n$) is connected to each neuron of the first hidden layer h_i (an m-dimensional vector where $\boldsymbol{h_i} \epsilon \mathbb{R}^m$), which is again connected to the output layer $\boldsymbol{y}$, a k-dimensional vector $\boldsymbol{y} \epsilon \mathbb{R}^k$, where $\boldsymbol{k}$ is the number of classes the perceptron is designed to predict. In a binary case, one neuron (instead of 2) is enough, as that neuron could output a value closer to 0 for class one and a value closer to 1 for class two. The neurons

are connected by typically randomly initialized weights $\boldsymbol{w} \epsilon \mathbb{R}^m$ for the input layer to the hidden layer and $\boldsymbol{w} \epsilon \mathbb{R}^k$ to the output layer.

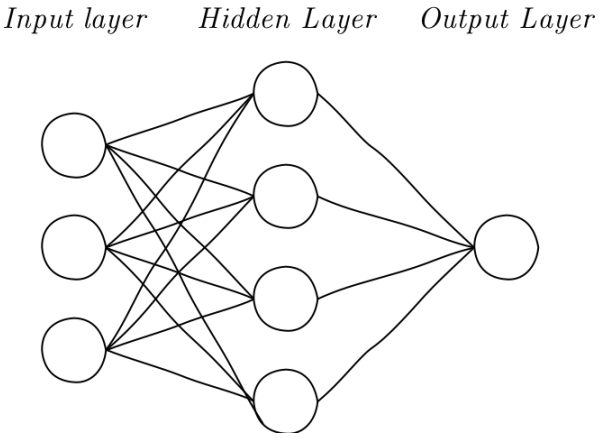

FIGURE 2.1: Multi-layer perceptron

Neuron

As the name suggests, neural networks such as the MLP are made up of many interconnected neurons. In the "forward-pass", i.e. when the network is asked to make a prediction based on some given input, neurons receive $\boldsymbol{n}$ inputs, which are multiplied by a corresponding weight, and summed up to get a pre-activation value $\boldsymbol{z}$, which is then passed to the activation function $\boldsymbol{f}$, which typically performs a non-linear transformation on the value (explained more in detail below). This is shown in figure 2.2 and can be summarized as:

$$y = f(z) = f(\sum_i w_i x_i) = f(w^T x) \tag{2.1}$$

Activation Function

Activation functions are required to introduce non-linearity to an ANN. If we constructed an ANN without activation functions we would simply be chaining a number of linear neurons of the form $y = w^T x$, the result of which would just be another

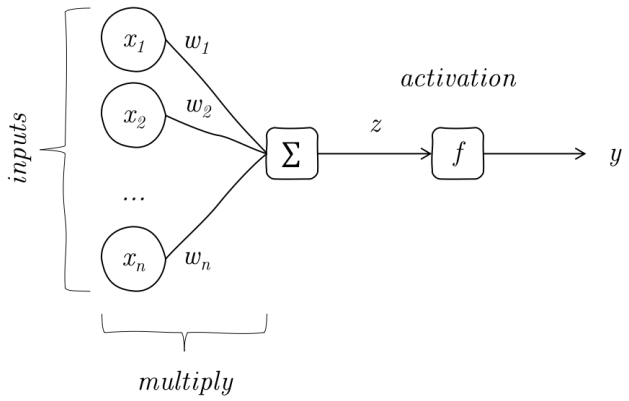

FIGURE 2.2: An exemplary neuron in a neural network

linear function. As discussed at the beginning of section 2.1, this would be inadequate for many applications. Some of the most popular activation functions are:

Linear (identity) Does not transform x.

$$identity(x) = x \tag{2.2}$$

Sigmoid Compresses the output to the range between 0 and 1. Often used in the output layer for binary classification tasks[43].

$$sigmoid(x) = \frac{1}{1 + (e^{-}x)} \tag{2.3}$$

Tanh Adjusts sigmoid such that it ranges between -1 and 1[43].

$$tanh(x) = \frac{2}{1 + (e^{-}2x)} - 1 \tag{2.4}$$

ReLU Short for "Rectified linear unit". A piece-wise linear function returning $\boldsymbol{x}$ if $\boldsymbol{x}$ is larger than 0, else returns 0[43].

$$relu(x) = \begin{cases} \text{x} & \text{if x} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Softmax n-dimensional sigmoid, compressing the sum of the output vector to 1, often used in the output layer for multi-class classification tasks[43].

$$softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}};\ z\, \epsilon\, \mathbb{R}^n \tag{2.6}$$

Loss Function

During the training of the network, once the forward-pass is complete and the network produced some numerical result, this result is then compared to the actual, known labels, and the error is calculated. For example, in a multi-class classification task where the model needs to differentiate between cars, planes, and trains, the model might produce an output vector yielding probabilities of [0.5, 0.2, 0.3] for a single image. Assuming the image shows a car, the corresponding one-hot encoded vector will be [1, 0, 0]. These vectors are then passed into a *Loss Function*, which calculates the prediction error, which is used to update the model weights connecting the model's neurons, in a subsequent step discussed in the next paragraph. Common loss functions are:

Binary Crossentropy This is typically used for binary classification tasks[25].

$$l(y, \hat{y}) = -\frac{1}{N}\sum_{i=0}^{N}(y \times log(\hat{y}_i) + (1 - y) \times log(1 - \hat{y}_i)) \tag{2.7}$$

Categorical Crossentropy Similar to binary cross-entropy but N > 2, used for multi-class classification tasks[25].

$$L(y, \hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} \times log(\hat{y}_{ij}) \tag{2.8}$$

Mean Squared Error Typically used for regession tasks as it computes the euclidean distances between the prediction vector and the labels vector[25].

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i_0}^{N}(y_i - \hat{y}_i)^2 \tag{2.9}$$

Gradient Descent

In order to learn a model that can make accurate predictions, we aim to minimize the loss of the model by adjusting our model's parameters (or weights). While this is generally done by optimizing them for the training data, the model is also often validated on otherwise unused data. In the end, the parameters with the best performance on the validation set are selected.

For minimizing L a large part of machine learning research is dedicated to a learning algorithm called *gradient descent*[5].

In gradient descent, we iteratively adjust the model parameters such that in each iteration the value computed by the loss function is brought closer to a local or global minimum. An illustration of this technique can be found in figure 2.3, where the loss (indicated by the arrows) is gradually improved by updating two parameters. Starting with randomly initialized parameters, we compute the partial derivatives

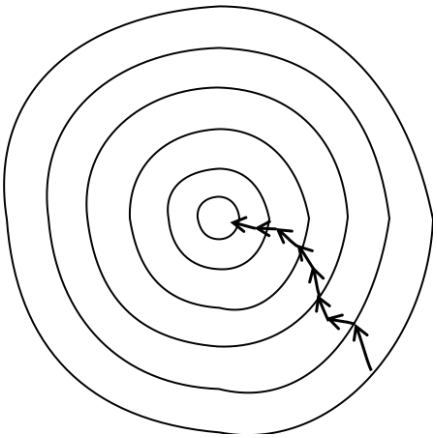

FIGURE 2.3: A two-dimensional illustration of gradient descent

of the loss function with respect to the model parameters and store the result in a gradient. The gradient is an indication of the slope of the loss function given the current values of our parameters, as well as the direction in which the parameters should be updated. After each parameter w_i is updated, this process is repeated, until a local or global minimum of the function we are trying to approximate is found. Formally these steps can be defined as:

$$w_i \leftarrow w_i - \eta \times \frac{\partial L}{\partial w_i}, \tag{2.10}$$

where η represents the algorithm's "learning rate", discussed more in-depth below. In practice, one of two variants of gradient descent is usually used. In Stochastic Gradient Descent, one data point is used to update the weights of the model, while in Mini-batch gradient descent small batches of data points are used instead of the whole data set. These variants are applied since having to iterate through the entire data set for each step of gradient descent would be too computationally expensive. SGD works under the assumption that much of training data is similar, and thus $\widetilde{\nabla} L$ can be called an unbiased estimator of ∇L. This property implies that while individual estimates on a batch might be inaccurate, the randomness will average out over time and the parameters are updated in the correct direction.

Finally, we need to mention that with gradient descent, we can only guarantee that the model converges to a local optimum. As there are generally few local optima in high-dimensional spaces, this is seldom a problem in real-world applications. However, gradient descent can get stuck on a saddle point or a plateau, which occurs more frequently[7]. While on a saddle point the gradient might be zero in all directions, there may still be a better point somewhere in the vicinity. See figure 2.4 for an example.

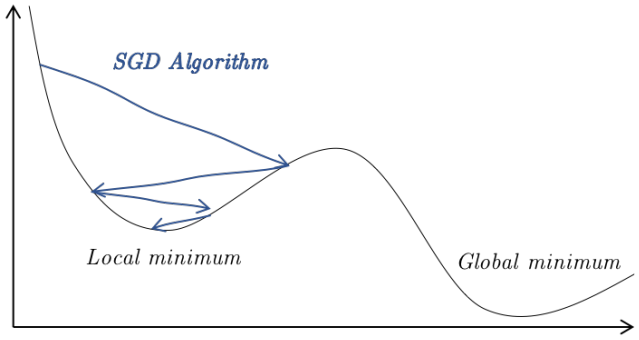

FIGURE 2.4: The stochastic gradient descent algorithm finding a local optimum

Computational Graph

As eluded to above, the gradient descent algorithm requires calculating partial derivatives. While derivatives for simpler methods like linear or logistic regression are well described in the literature, deriving the gradient for more complex functions becomes increasingly difficult. For a neural network with many layers and even more neurons, setting up a long function that describes the entire network is nearly impossible.

Computational graphs represent an abstraction that allows machine learning researchers to circumvent this problem. In place of attempting to construct a functional representation of an entire neural network, of which a derivative can be computed, the network is broken down into smaller more manageable pieces, such as multiplication or the exponential function, where the direct derivative is known. These smaller functions are connected into a graph where each node represents a function, and each edge shows how information moves between nodes. An example of this is shown in figure 2.5.

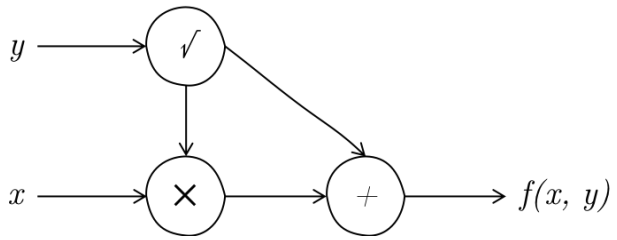

FIGURE 2.5: A computational graph of the function $f(x, y)$

This graph represents the function

$$f(x, y) = x \times \sqrt{y} + \sqrt{y}. \tag{2.11}$$

It also shows how constructing a graph can be more computationally efficient as the term $\sqrt{y}$ only needs to be calculated once. Representing our model as a graph allows us to employ back-propagation, an algorithm that applies the chain rule of derivation to find the partial derivative of the loss function with respect to the model weights. For the last layer in the model before the output layer, this partial derivative can be described as:

$$\frac{\partial L}{\partial W^{(L)}} = \frac{\partial L}{\partial A^{(L)}} \times \frac{\partial A^{(L)}}{\partial Z^{(L)}} \times \frac{\partial Z^{(L)}}{\partial W^{(L)}}, \tag{2.12}$$

where L describes the loss, W the weights, A the activation value and Z the pre-activation output. For additional layers, we need to add to this function by multiplying it with the partial derivatives of the weights of those layers with respect to the loss.

Overfitting

Overfitting is a common problem with training neural networks[6]. A neural network is an *eager learner*, meaning that it stores many parameters that were optimized on an underlying training data set. Overfitting refers to the processes of learning particular pieces of information about the training data, which do not generalize well to the overall population. The result is typically a model that yields a low loss when evaluated on the training data itself, but a much higher loss when evaluated on unseen test data. An example can be seen in figure 2.6, where the grey line represents a model that overfit on a specific set of features and maps the training data distribution very closely whereas the model represented by the green line was trained to learn more general features.

The opposite problem is "underfitting" when the model is not strong enough to capture discriminative information on the training data and yields a high training and test loss. While this can often be addressed by increasing the number of learnable parameters in the network and training the model for longer on the training data, overfitting is more challenging to tackle.

Early Stopping

Neural Networks tend to overfit on the training data when they train on it for too many iterations (also referred to as "epochs"). To address overfitting, we can apply early stopping[3]. With early stopping, we split the training data into a training set and a validation set. After each epoch (a full pass over the training data), the quality of the model is evaluated on the validation set. If the validation loss does

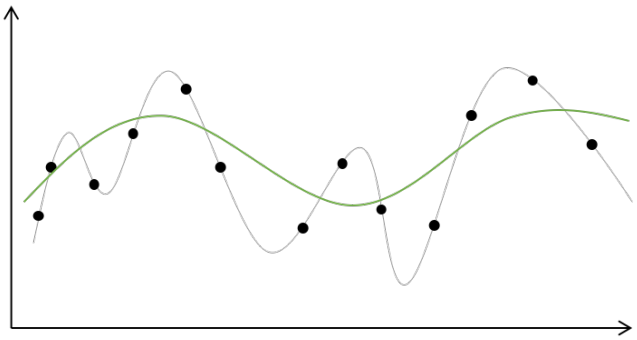

FIGURE 2.6: An example of overfitting. The grey line represents a model that overfits, whereas the green line represents a model that learnt more general features of the population.

not improve within a specified number of epochs, training is stopped. Typically, the model weights that yielded the lowest validation loss are then restored at this point.

Regularization Techniques

Dropout Dropout is a regularization technique, whereby each neuron except the output neurons has a probability $\boldsymbol{p}$ of being ignored during a given training step[60]. During the forward- and backward-pass during this training step, the neuron is shut off and does not perform any calculations. As a result, the neural network tends to converge slower, but inter-dependency between neurons across layers is reduced, allowing the model to generalize better on unseen data. We can see in figure 2.7 how this process looks in practice. Dropout is only applied during the training stage. At

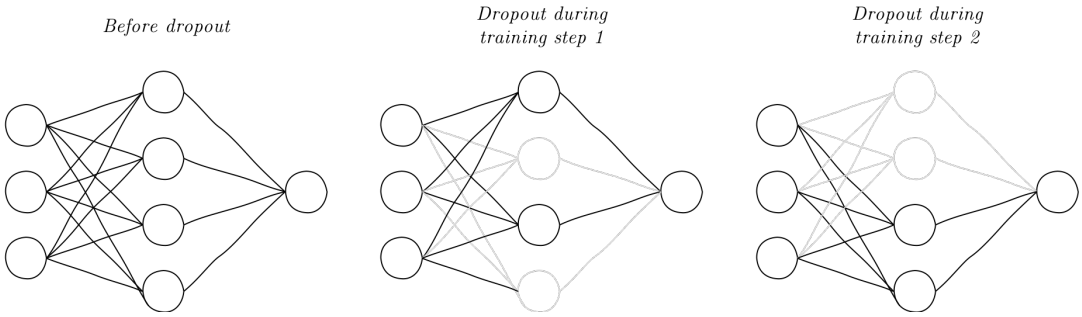

FIGURE 2.7: An example of dropout for two consecutive training steps

prediction time, all neurons are active; however, their weights and activations are typically scaled to the dropout factor as otherwise, the input that a neuron receives during prediction time would on average be $\frac{1}{p}$ higher than during training time.

L1 and L2-Regularization L-Regularization combats the issue of model weights growing out of proportion[41]. In both techniques, the objective function is modified by adding the model weights to the model loss. In this case, the model Ois penalized if it achieves a low loss at the cost of large weights. Thus, in order to reduce the loss, the weights need to be kept small as well. With L1-regularization, the absolute value of the weight is added to the objective function. The parameter λ indicates

how much the model should be regularized.

$$J(W) = L(Y, A) + \lambda \sum_{w} |w| \tag{2.13}$$

This means that for the update rule in the backward pass, a fixed movement towards 0 is considered.

$$w \leftarrow w - \eta(\frac{\partial L}{\partial w} + \lambda sign(w)) \tag{2.14}$$

For L2-regularization, the squared weight w^2 is added to the objective function alongside the loss.

$$J(W) = L(Y, A) + \lambda \sum_{w} w^2 \tag{2.15}$$

The update rule shows that in L2-regularization the update is now proportional to the weight itself, indicating that large weights shrink proportionally faster.

$$w \leftarrow w - \eta(\frac{\partial L}{\partial w} + 2\lambda w) \tag{2.16}$$

The L1 regularizer tends to produce sparse weights, as most weights are pushed to 0, and consequently only the most useful weights will be non-zero to make predictions. As a result of this sparsity, *feature selection* occurs, where L1 regularisation forces each layer to select only a few inputs in order to keep the weights small. By contrast, with L2 regularization, layers benefit from taking in a combination of features, as weights are not pushed as strongly towards 0 when they already have small values.

Batch normalization Batch Normalization is a normalization technique introduced in 2015 by Sergey Ioffe and Christian Szegedy in their paper *'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'*[42] and today it is used in almost all convolutional neural network architectures (see section 2.1.2). According to the original paper, batch normalization helps reduce the internal covariate shift of the hidden layers of the network. However, in a more recent paper titled *"How Does Batch Normalization Help Optimization?"*[51], the authors suggest that batch normalization actually "makes the optimization landscape significantly smoother." The result is a changed behavior of the gradients, which becomes more predictive and stable.

Historically, research has focused on uniformly distributing the data fed into the input layer of the neural network. For example, in a case where the model is trained to separate cats from dogs, it makes intuitive sense to feed cats and dogs of all shapes and colors to the model in a given minibatch during training, rather than feeding black cats and dogs in one mini-batch to the model and brown cats and dogs in the next because these subsets of data have different distributions.

For the hidden layers, however, this input distribution changes every time there is a parameter update in the previous layer. This challenge is addressed by batch normalization. BN replaces the incoming vector of pre-activation values of a given minibatch in a given layer with its normalized version. Formally this process can be summarized in four steps (simplified for one layer to limit the number of superscripts):

1. Calculate the mean μ of the mini-batch.

$$\mu = \frac{1}{m}\sum_i z^{(i)} \tag{2.17}$$

2. Calculate the variance σ^2 of the mini-batch.

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2 \tag{2.18}$$

3. Calculate the normalized value of z, z_{norm}.

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2}} \tag{2.19}$$

4. Calculate $\widetilde{z}_{norm}$ by multiplying z_{norm} with a scale γ and adding a shift β and replace the pre-activation z with $\widetilde{z}_{norm}$.

$$\widetilde{z}^{(i)}_{norm} = \gamma z^{(i)}_{norm} + \beta \tag{2.20}$$

Throughout the experiments conducted in light of this thesis, we experimented with all of the regularization techniques mentioned above and found batch-normalization to be the most effective.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks are another type of feedforward neural network, typically used for image recognition/computer vision tasks in deep learning[30]. The purpose of employing convolutional layers is to learn useful features from an input image such as edges (horizontal, vertical, or diagonal) and spatial relationships between elements in an image (a face usually consists of eyes, a nose, ears, and a mouth). This is done through applying one or more filters to the input image (and in the case of further hidden convolutional layers applying additional filters on the outputs from the first convolutional layer).

Filters

In image processing, a filter refers to an operation that transforms an image in a meaningful way, i.e., by increasing the image's contrast or grey-scaling an RGB image. This is done by applying a standard, predefined mathematical operation on each pixel or a group of pixels. To grey-scale an image, for example, a filter would multiply each pixels' image channels (**R**ed, **G**reen, and **B**lue) by $\frac{1}{3}$ and sum the results to output a single channel that held the numerical average of the three color channels. In this example, the filter is a **fixed** 3-dimensional vector of shape (1 x 1 x 3), with each dimension holding the value $\frac{1}{3}$. See figure 2.8 for a pictorial description of this process.

In convolutional NNs, rather than specifying a filter's values in advance, the filter's values are randomly initialized and then *learned* over consecutive training iterations. While the last dimension of the filter (the depth, or number of *kernels*) will always need to equal the last dimension of the input shape, the height and width of the filter are hyperparameters that can be freely tuned.

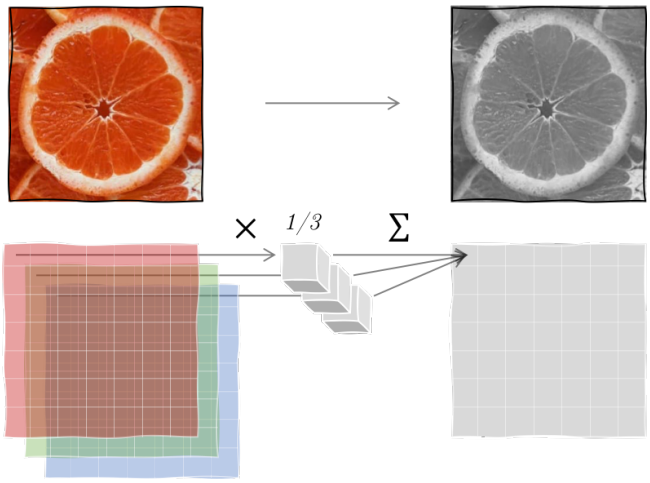

FIGURE 2.8: Example of applying a filter consecutively to each pixel of an image to greyscale the image

Usually, a convolutional layer holds more than one filter, each of which learns a separate feature of the input image (e.g., filter 1 learns horizontal edges, while filter 2 learns round edges. All learned features are subsequently combined in the forward pass to make a prediction.

Kernel

One filter is typically made up of several kernels. The term *kernel* refers to a 2D array of weights, which are the parameters that are being tuned in a convolutional layer. See figure 2.9 for an example. Here the filter is a (3 x 3 x 3) matrix, meaning that it consists of 3 kernels of the shape (3 x 3). First, each kernel is applied to one input channel, without *padding* (explained below). The three convolutions that are performed result in three channels, each with a size of (3 x 3). These resulting channels are then summed together, forming one single channel with dimensions (3 x 3 x 1), which is the result of the convolution.

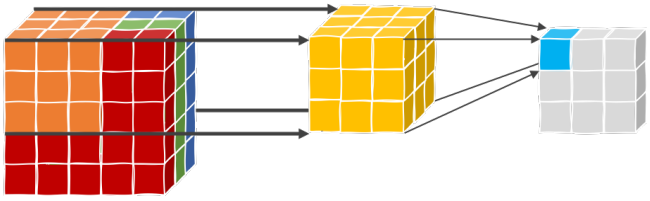

FIGURE 2.9: An example of a filter consisting of three kernels sliding through an RGB image translating it into one output channel

Convolution

We can describe a convolution as follows:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.21}$$

The function above describes the process of taking a two-dimensional input image I as an input, and applying a two-dimensional kernel K on the image. The kernel is

applied to a region of the image that matches its shape. Then, an element-wise sum-product is calculated. The kernel is then moved by a predefined number of steps (called *stride*), and the operation is repeated. Applying this algorithm has the same effect as multiplying the input image by a sparse matrix.

Padding

As can be seen in figure 2.9 a filter with kernels of dimensions larger than (1 x 1) reduces the size of the input image. If this is not desired, it can be prevented by adding some padding (pixels with 0 values) around the input image, as seen in figure 2.10.

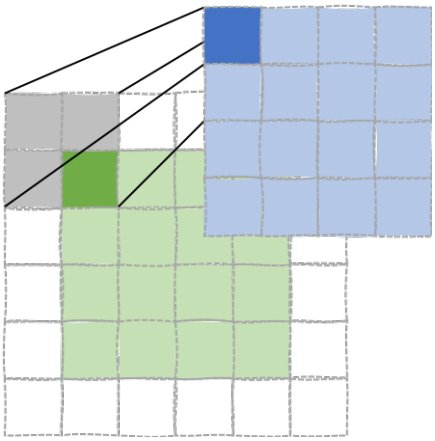

FIGURE 2.10: Example of padding, where a (2x2) kernel and a (4x4) input image (green) produce a (4x4) output image (blue)

Pooling

Still, with images larger than 100 x 100 pixels padding only has a negligible impact on the size of the output. If a reduction in the height and width of the input channels is desired, we can apply a pooling mechanism, which computes a summary statistic of a group of adjacent pixels. One of the most common techniques is called *max-pooling*. In max-pooling, a pool-size (height and width) is defined, which is then applied to the output of the convolutional layer. The max-pooling layer selects the highest activation value from its pool, which then becomes part of the output matrix. See figure 2.11. Pooling generally makes the network more translation invariant,

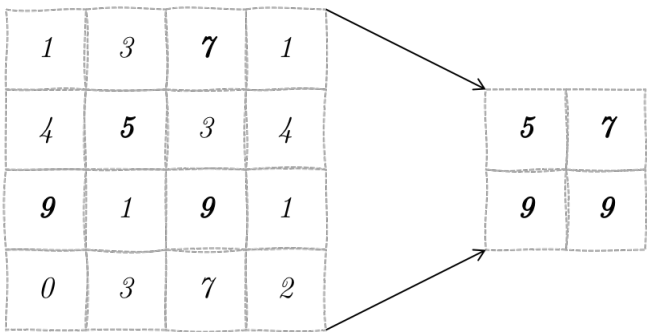

FIGURE 2.11: An example of max-pooling with a 2x2 pool-size. Only the highest activation value in a given 2-by-2 quadrant is added to the output.

meaning that a slight rotation or shift in the image will not substantially alter the model's prediction.

Parameter Sharing

One final reason why convolutional networks have become increasingly popular for many deep learning tasks is parameter sharing. In a traditional multilayer perceptron as discussed in section 2.1.1 every weight in the model is used exactly once, when the output of a layer is computed, by multiplying it with one element of the input. In a convolutional neural net, each member of the kernel (see above), is used at every input position, except for the boundary pixels, if no padding is used. This has no impact on the forward propagation run-time, but drastically reduces the spatial requirements of the model as significantly fewer parameters need to be stored. Also, convolution is dramatically more memory efficient than dense matrix multiplication as a result.

2.1.3 Transfer Learning and Domain Adaptation

As mentioned in section 2.1.1, the parameters of a neural network are typically initialized to random values, when learning a new task. Translating random initialization to how humans learn would imply completely resetting the brain each time we learned a new task. Humans, however, have the innate ability to transfer knowledge about one domain to another related domain. For example, pre-existing knowledge about how to ride a bike can help when learning how to ride a motorcycle.

Transfer Learning

In deep learning, the idea of applying pre-existing knowledge learned for a specific task on one data distribution to a new task on another data distribution is referred to as transfer learning. In computer vision, for example, if the task is to identify cars in images, initializing a model with the parameters of another model originally designed to recognize trucks, can speed up training significantly over alternatively initializing parameters completely randomly. The reasoning behind this approach is that in computer vision tasks, as discussed in the previous section, objects in images share low-level features, extracted by the lower levels of the neural network. The upper layers, - often dense, fully-connected layers - take these extracted features in, and learn the actual classification task.

Popular deep learning libraries such as Tensorflow and PyTorch allow users via an API to download popular deep learning architectures such as ResNet50[19] or VGGNet[57]. Rather than randomly initializing and training these architectures from scratch, which due to the many millions of trainable parameters would be cost-prohibitive and results in long training times, they can be initialized with parameters learned on the Imagenet data set, a dataset containing images of 1,000 different objects[50]. Due to the diversity of the Imagenet data set, these trained model parameters have been shown to serve as a good starting point for another computer vision task, underscoring the viability of transfer learning[54]. In practice, as the upper layers are typically fine-tuned for a specific classification task, they are commonly replaced with randomly initialized layers, when a new task is to be performed. To train the new final layers, the lower, convolutional layers are typically "frozen", meaning that their parameters do not change during training so that the

low-level features learned on the Imagenet data set are preserved. Only the final layers are then freely trained to learn the new classification task.

In transfer learning, however, we are still assuming that our initial training set distribution is representative of the underlying distribution. I.e., if we initialized a model architecture such as VGGNet on the trained parameters of the Imagenet dataset and subsequently trained the final layers to recognize taxi cabs in New York City, we would expect the model also to recognize taxi cabs in Berlin. However, while the model might still perform better than the original VGGNet architecture due to some similarity between New York and Berlin taxi cabs, it would likely still not perform as well as expected. The reason is that the problem domain changed. In this particular case, the domain of the input data changed, while the label's domain (the task domain) stayed the same. Enter domain adaptation.

Domain Adaptation

Domain adaptation can be considered a sub-field of transfer learning and is employed when a model trained on a source distribution is put into practice in the context of a different (but related) target distribution [14]. Generally speaking, the level of relatedness between the source and the target domain determines how successful the domain adaptation will be. Returning to the example of taxi cabs in New York and Berlin, the next step would require to continue training the modified VGGNet model, which has already been trained on images of New York taxi cabs. Two methods are typically used for continued training: Reweighing the source samples, which would imply training only on Berlin taxi cabs, or learning a shared space between the distributions, i.e., training on a joint data set of New York and Berlin cabs. Either approach, however, would likely decrease training time and yield better results faster, compared to randomly initializing the final layers of the original VGGNet architecture again and training them from scratch.

2.2 Federated Machine Learning

2.2.1 Overview

Federated Machine Learning was first introduced by Google in 2016 [39][28][29]. Different from a centralized setting, in a federated learning setting, multiple devices e.g., end-user devices such as mobile phones, or business infrastructure such as hospital servers, contribute to learning a machine learning classifier. The classifier can be a deep neural network, but also a simpler model, with fewer parameters such as a support vector machine, or a logistic regression model[17]. Federated learning models are distinct in that the original training data never leaves the respective local device that collected it. Each device (also dubbed *client*) maintains a version of the same model, which is updated with every new observation. The updates to the model (e.g., the updated weights and bias of neurons in a neural network), not the observations themselves, are then shared with a central server, which averages the new models from all participating devices. Once a new version of the model has been trained, it is pushed back down to all clients. This process repeats continuously until the model converges.

Figure 2.12 displays a graphical representation of this process. In (A) the server-side model is pushed down to a mobile phone, which subsequently trains the model on local data. Training happens on several devices, as depicted in (B). Subsequently,

the new models are pushed to the cloud (the central server), and averaged, to arrive at the model in (C). This model is then pushed to all devices, and the process starts again.

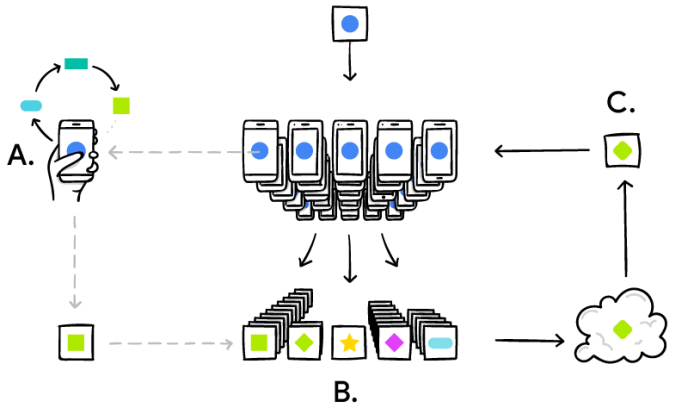

FIGURE 2.12: Federated Machine Learning: Conceptual Architecture[37]

Three primary benefits emerge from this approach, the first of which will be discussed more in-depth in the remainder of this work:

Privacy: In an FL approach, the central server only aggregates ephemeral parameter updates, meaning model updates that last only long enough to be transmitted to the central server and incorporated into the central model. This implies that clients still need to trust the entity aggregating different models enough to receive the individual parameter deltas, but clients only receive the final trained model for inference. As a result, the attack surface for gaining access to personal data is limited to the device only, as opposed to the device and the cloud.

Computing power: Shifting computation down to the devices also significantly reduces the processing power required in a central location, since the role of the central entity is merely to average the updates from all participating devices, as opposed to continuously retraining a global model on new sets of data. Today's mobile devices are becoming increasingly powerful, especially with the emergence of AI chipsets[24]. Thus, considering that there are billions of mobile devices worldwide, the accumulated computing power from these devices far-surpasses that of even the most potent datacenter.

Real-time learning: Finally, since the models are trained locally, updates are instant, enhancing time-to-prediction, and as a consequence, user-experience. Moreover, typical implementations to date have ensured that model updates are only pushed to and pulled from a central server once a device was idle, plugged into power and connected to a WiFi connection. Limiting updates to such a setting addresses the issue of unstable internet connections and ensuring that user-experience is not affected detrimentally due to power-consuming up- and download processes.[70].

2.2.2 The Federated Averaging Algorithm

As discussed above, in federated learning, we assume that our data is not centrally stored, but partitioned over K number of clients. Assume that

- Each partition can be represented as a set of indices $\mathcal{P}_k$ of data points that a given client k holds
- n represents the number of all data points collected by all clients and thus n_k represents the number of data points that the client holds
- $n_k = |\mathcal{P}_k|$

If the standard definition of minimizing a loss function is given by

$$\min_{\theta \in \mathbb{R}^d} f(\theta) \tag{2.22}$$

where

$$f(\theta) \stackrel{def}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(\theta) \tag{2.23}$$

and $f_i(\theta)$ represents the loss for a prediction of one observation $(x_i, y_i$ given model parameters θ, this loss function can be rewritten to represent K clients in a federated setting, such that

$$f(\theta) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\theta) \tag{2.24}$$

where

$$F_k(\theta) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\theta) \tag{2.25}$$

.

To break this down: Instead of computing our average loss (e.g. our MSE) as an average over n number of samples from a centralized data set as $\frac{1}{n} \sum_{i=1}^{n} f_i(\theta)$, we compute the average loss $F_k(\theta)$ for a specified client k as $\frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\theta)$ and then group the loss of all participating clients K, by computing a weighted average loss based on the number of data points n_k that each client holds.

Analog to computing the loss, we also compute the gradients of the federated model. Keeping equation (2.10) in mind, in a federated setting each client computes the average gradient g_k on its local data as

$$g_k = \nabla F_k(\theta)\,. \tag{2.26}$$

Then, each client takes a step of gradient descent and updates its parameters accordingly, formalized as

$$\forall k, \theta_k \leftarrow \theta - \eta g_k\,. \tag{2.27}$$

This step can be repeated multiple times, i.e., for multiple epochs E, until a central server then computes the weighted average of these gradients similar to the weighted average of the loss above as

$$\theta \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} \theta_k \tag{2.28}$$

to update the model parameters of the overall model, stored on the central server. This concludes a full round of updates to the global model.

Assuming mini-batch stochastic gradient descent, in such a federated setting the computational effort for one full update is controlled by three parameters:

1. The fraction C of clients K that participate in a given update round.
2. The number of steps of gradient descent (or epochs) E that each client performs
3. The batch size B used for all client updates

While C impacts the computational power required at the server level (more participating clients requires more transfer of data to the server and more effort in aggregating information), E and B impact the computational effort required on the client-side. The complete pseudo-code for this approach was first proposed by [39] and is provided for convenience in Algorithm 1.

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate. w denotes the model parameters.

procedure SERVER EXECUTES:
  initialize w_0
  for each round $t = 1, 2, \ldots$ **do**
    $m \leftarrow \max(C \times K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for each client k $\in S_t$ **in parallel do**
      $w_{t+1}^k \leftarrow \text{CLIENTUPDATE}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

procedure CLIENTUPDATE(k, w) ▷ Run on client k
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to E **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w to server

2.2.3 Applications for Federated Learning

Potential applications for federated learning are vast and differ substantially in vertical and specific use-case, but typically bear three common traits[70]:

1. Task labels do not necessarily need to be provided by humans but can be derived naturally from user interaction.
2. Training data is privacy sensitive
3. Training data is large, and is difficult to be feasibly collected in a central location

Not all of these conditions strictly need to hold when applying federated learning, but it is under these circumstances, that federated learning provides the most significant value over other machine learning techniques. To illustrate the potential of federated learning, in the following, we briefly review some of the recent applications of FL models in practice.

Brain Tumor Segmentation Without Sharing Patient Data Computer chip-maker intel [53] leverages FL to showcase how multiple healthcare institutions can collaborate in a privacy-preserving manner, leveraging each institution's electronic health records (EHR). The authors argue that while collaboration between institutions could address the challenge of acquiring sufficient data to train machine learning classifiers, but the sharing of medical data is heavily regulated and restricted. They present the first use of an FL classifier for multi-institutional collaboration and find that they can learn a similarly performant federated semantic segmentation model (Dice=0.852) compared to that of a model trained on centralized data (Dice=0.862). This strengthens the hypothesis that FL can lead to breakthroughs in the medical space without compromising patient privacy.

Improving Firefox Search Bar Results In [18], the author leverages federated learning in a production level setting using data from 360,000 users to improve the search results in the Firefox Search Bar, without collecting the users' actual data. Millions of URLs are entered into Firefox daily, thus notably improving the auto-complete feature for users enhances user experiences and can increase customer retention.

Improving Google Keyboard Suggestions Google describes one of the first implementations of federated learning on a large scale, training a global model to "to improve virtual keyboard search suggestion quality"[70]. In their paper, they address many of the technical challenges of coordinating training on millions of devices worldwide. Examples include connectivity issues, the bias of training a model across different time zones, and minimizing the impact on user experience that training a machine learning model locally has (e.g., battery-life and device-speed). They note that future work on privacy still needs to be done and cautiously call their method "privacy-advantaged", vs. "privacy-preserving".

2.2.4 Practical Challenges for Federated Learning

Federated Learning is not without its challenges. A few key-properties describe a typical federated optimization problem:

Non-IID A dataset's data points are said to be IID if they are independent and identically distributed. If the IID assumption holds, the underlying mathematical and statistical techniques can often be simplified. For example, if we draw a sufficiently large sample *at random* from the overall distribution, we can with a specified level of confidence state that the sample is representative of the overall population. In Federated Learning, clients' datasets will often differ substantially from those of other clients (e.g., in the case of mobile phones the phone's dataset is dependent mostly on the interaction with one particular user). Thus, sampling a client at random will likely not yield a dataset that is representative of the global population distribution. Other sampling techniques, such as *stratified sampling*, can be employed to mitigate this problem.

Unbalanced In a similar fashion, clients' datasets may vary substantially in size. Again using the example of mobile phones, some people may use their phones significantly more than others, creating larger datasets that potentially skew the resulting weighted average in their direction, while penalizing users generating less data.

Limited Communication Mobile phones are frequently offline or are connected to flaky or expensive internet connections. Healthcare facilities, especially in rural areas in the United States, often have only slow internet connections [38] or a minimal number of computers that are linked to the internet. While it is usually cheap to compute updates locally, since the amount of training data is low, communicating these results becomes much more time-consuming, making communication-speed and averaging the bottleneck in federated learning.

Maintaing performance Finally, and perhaps most importantly, since a global federated model is not trained on the raw data, but rather a proxy (the clients' parameters), and only local models have access to this data, care needs to be taken in achieving similar performance in a federated setting compared to a centralized machine learning approach. If a model fully preserves the privacy of a client but produces inaccurate predictions, it can essentially be rendered useless.

In the remainder of this work, we will focus on the issues of handling *non-IID* and *unbalanced* data, as well as *maintaining performance*, while leaving the issue of *limited communication* to future work.

Chapter 3

Data

3.1 Overview

As eluded to in chapter 2 one of the most promising applications of federated learning is the healthcare space, where many different entities can jointly learn a model without sharing sensitive raw data, thereby adhering by privacy regulation such as GDPR in the EU or HIPAA in the US.

To demonstrate the effectiveness of Federated Learning in a healthcare setting, we chose to work with the **UNBC-McMaster shoulder pain expression archive database** [34], a database comprising of 200 video sequences containing spontaneous facial expressions of 25 individuals. The videos' frames are labeled individually and constitute a data set that could just as well have been gathered outside of an experimental setting by multiple hospitals cooperating to train a model that recognizes pain in individuals. The importance of regularly checking on a patient's well-being is described in Atul Gawande's "The Checklist Manifesto" [11]. In his work, he describes the significant improvements that compliance with standardized hygiene and a priori checklists yield in intensive care units. Among these compliance-measures is pain monitoring, where a nurse checks on a patient every four hours and makes adjustments to medication, if the patient is found to be suffering from pain.

Although evidence suggests that improved pain monitoring yields better patient outcomes[68], such measures have been difficult to implement due to the competing demand for nursing staff[1]. Therefore, automatic pain monitoring could improve the care environment for patients, positively impact patient outcomes, and relieve some of the pressure on nursing staff.

3.2 Description

To compile the UNBC-McMaster shoulder pain expression archive database researchers recruited a total of 129 participants (63 male, 66 female).

The publicly available subset of this database holds a total of 48,106 video frames from 25 test subjects, each labeled with the test-subject number, the session number, the video frame number, and the level of pain that an individual is feeling in a given frame. Each individual participated in a different number of sessions.

3.2.1 FACS coding

The pain level is determined by a professional "Facial Action Coding System" (FACS) [9] coder. In FACS, facial actions are compartmentalized into 44 individual action units (AUs). To compile the shoulder-pain database, the researchers only focused on

the AUs that are known to be most closely associated with pain, including: "brow-lowering (AU4), cheek-raising (AU6), eyelid tightening (AU7), nose wrinkling (AU9), upper-lip raising (AU10), oblique lip raising (AU12), horizontal lip stretch (AU20), lips parting (AU25), jaw dropping (AU26), mouth stretching (AU27) and eye-closure (AU43)" [34].

3.2.2 Prkachin and Solomon Pain Intensity Scale

According to Prakchin's article "The consistency of facial expressions of pain: a comparison across modalities" [45] from 1992, the bulk of what humans feel as pain, is expressed by four of the 44 actions determined in FACS coding, namely brow lowering (AU4), orbital tightening (AU6 and AU7), levator contraction (AU9 and AU10) and eye closure (AU43). In a follow-up paper, Prkachin and Solomon [46] defined pain as the function of the following parameters:

$$Pain = AU4 + max(AU6, AU7) + max(AU9, AU10) + AU43. \tag{3.1}$$

The result is a 16-point scale, where the first three components are measured on a 6-point scale (0 = absent to 5 = maximum intensity), and the final element "eyes open/closed" is binary.

3.2.3 Distribution

The 200 available sequences are collected from 25 test subjects. As figure 3.1 shows, this publicly available subset of the pain data holds individuals who are experiencing pain levels ranging from 0 to 9, with nearly 90% of images representing either a 0 or a 1 on the pain intensity scale.

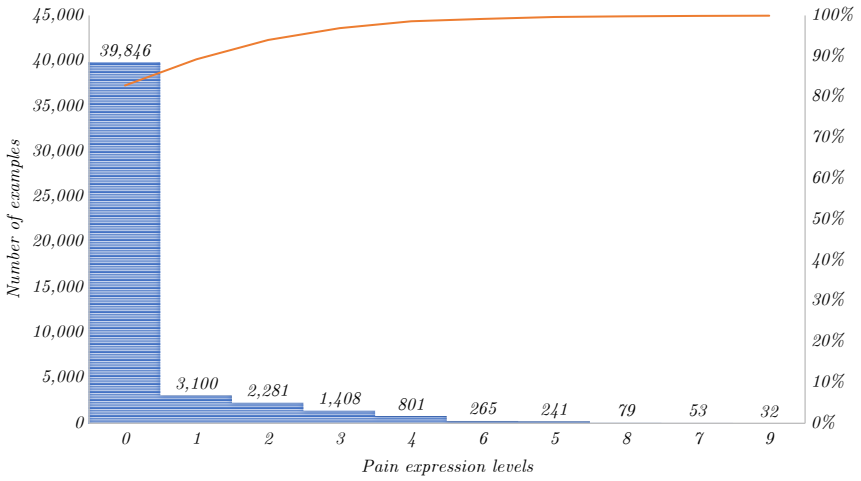

FIGURE 3.1: Pain Intensity Distribution, UNBC-McMaster shoulder pain expression archive database

Figure 3.2 shows some examples of different individuals experiencing pain and paired with a corresponding pain-level. As can be seen in this figure, the differences in pain levels based on the images are quite nuanced. The evident difficulty of separating examples of "pain" from one another as well as from examples of "no-pain", paired with the heavy skewness of the data towards "no-pain" prompted us to perform the pre-processing and data augmentation steps outlined below.

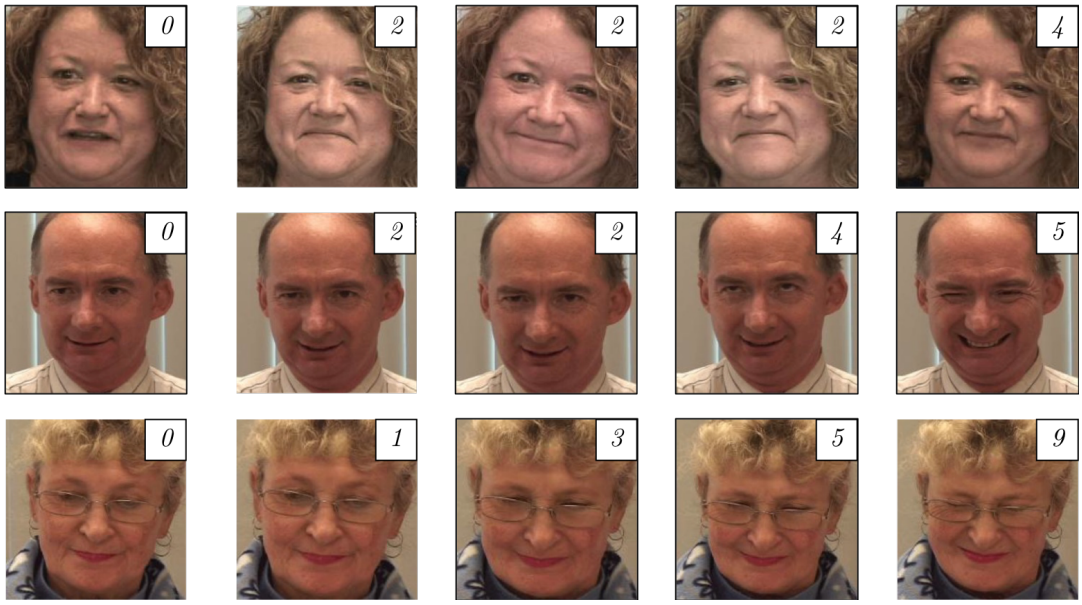

FIGURE 3.2: An example of max-pooling with a 2x2 pool-size. Only the highest activation value in a given 2-by-2 quadrant is added to the output.

3.3 Pre-Processing

In a first step, we wanted to ensure that the relevant features in a person's face that are an indicator of pain are as easily identifiable for a machine learning algorithm as possible and applied the following pre-processing steps.

3.3.1 Greyscaling

OpenCV's *imread()* function provides the option to read in an image with 3 channels or 1 channel. Selecting 3 channels will load a colored image (provided that the input image is colored) while selecting 1 channel will automatically greyscale the image. We chose to greyscale the image in order to reduce the amount of information that is passed to the network for learning. Color is not relevant to detecting pain using the FACS system, and so greyscaling can reduce the number of input features passed to the network by two thirds. An example of the results of this step can be found in figure 3.3.

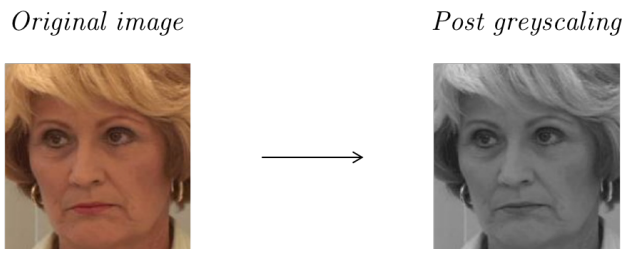

FIGURE 3.3: An example of a greyscaled image, using the OpenCV imread() function

3.3.2 Histogram equalization

Histogram equalization is a technique that helps enhance the contrast in images[44]. Since the Prkachin and Solomon Pain Intensity Scale is measured by looking at only a limited number of features in a person's face, we want the appearance of these features to be as poignant as possible. Increasing the image's contrast makes features like the person's eyes or eyebrows stand out further. In histogram equalization, we construct a histogram of the pixel values of a black-and-white image, as seen in figure 3.4. We then spread out the most frequent intensity values of the image, i.e., the intensity range of the image is stretched out, meaning that light pixels become lighter and dark pixels become darker. As figure 3.4 shows, the intensity of the

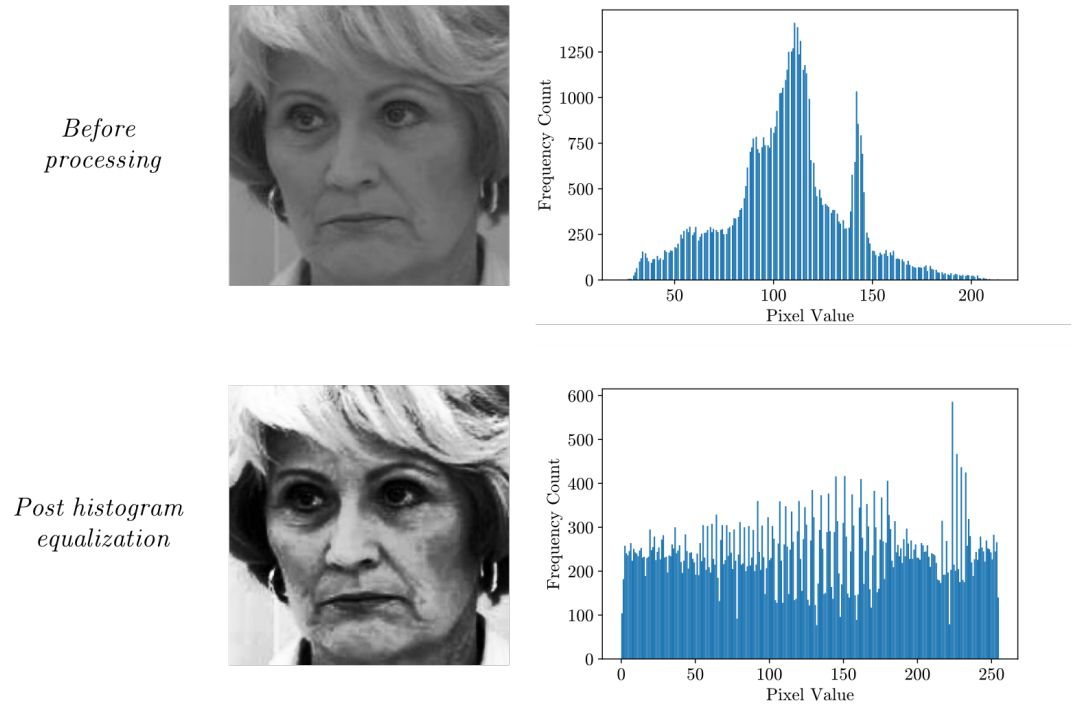

FIGURE 3.4: An example of an image where histogram equalization has been applied

relevant edges that we want our neural network to detect in this test subject's face increases by applying histogram equalization.

3.3.3 Normalization

Finally, our images are represented as 2-dimensional arrays holding integer values from 0-255. To ensure that our neural network does not suffer from the "exploding gradient" problem, we normalize this range to a range of 0-1 by converting all integers to 32-bit floating-point numbers and dividing these numbers by 255. The relative distances between features are thereby maintained, but the absolute values are rescaled, which helps to keep the values of our gradients small during the training phase.

3.4 Augmentation and Sampling

As already seen in figure 3.1, the distribution of our training data is heavily skewed. There are many more examples of test-subjects experiencing no pain rather than experiencing pain, as it would also be expected in a 'real-world' example, where patients may only experience pain sporadically during their hospital visit. If this unbalanced data were fed to a neural network during training, the network would be biased towards the images labeled *'0'*, since correctly identifying these images can be one strategy for the network to minimize the loss function.

To deny our model this strategy, since correctly classifying the minority group of images labeled as "pain" is crucial, we resort to three strategies:

1. Binarize the training data
2. Perform data augmentation to upsample the number of positive training examples
3. Balance the training data by downsampling the negative training examples

3.4.1 Binarizing the training data

Outside of an experimental setting, it will often not be relevant to identify what exact level of pain a patient is feeling, but whether a patient is experiencing pain at all. We, therefore, decided to binarize the training labels by bounding the vector of labels by a "min" function that returns 0 for all labels smaller than 1, and 1 otherwise. From here on we will call 0, no pain, "negative example" and 1, pain, "positive example".

$$y_{bin} = min(y_{ord}, 1) \tag{3.2}$$

This yields the distribution in figure 3.5. For a more detailed picture, see table 3.1,

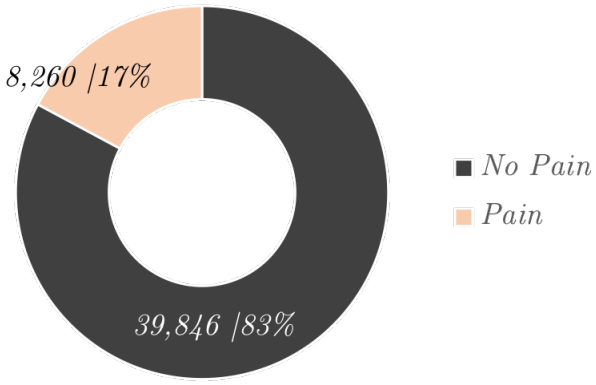

FIGURE 3.5: Binary pain-label distribution

with test subjects split into groups, as further discussed in chapter 6.

3.4.2 Upsampling and Downsampling

Upsampling

To upsample (i.e., increase) the number of positive examples, we applied two data augmentation techniques: We first created a flipped copy of all images. We then

Group 1				Group 2			
Person	No Pain	Pain	% Pain	**Person**	No Pain	Pain	% Pain
42	1,895	239	11%	**43**	1,028	92	8%
47	1,544	64	3%	**48**	798	84	9%
49	2,194	524	19%	**52**	2,503	106	4%
66	1,947	512	20%	**59**	640	133	17%
95	304	498	62%	**64**	1,394	155	10%
97	3,212	147	4%	**80**	896	1,068	54%
103	2,738	824	23%	**92**	1,031	471	31%
106	2,281	517	18%	**96**	2,175	178	7%
108	2,453	455	15%	**107**	1,599	442	21%
121	478	40	7%	**109**	1,724	179	9%
123	822	361	30%	**115**	1,184	99	7%
124	699	996	58%	**120**	1,490	76	4%

TABLE 3.1: Positive and negative examples by test subject and training group before any up- or downsampling. Test subject 101 was removed from the data altogether, as there were no positive ("Pain") examples of this test subject at all.

created another copy of all originals and flipped copies, respectively, that was randomly rotated by either 10 degrees to the left or the right. After the rotation, each image was cropped from 250 x 250 pixels to 215 x 215 pixels in order not to have any whitespace or artificial filling around the images. To make the input shape consistent across mutations, all images were cropped to 215 x 215 pixels. We underscore the importance of applying these steps to both positive and negative examples, as the network might otherwise learn that a specific image mutation always represents a positive example. Figure 3.6 shows the effect of applying these data augmentation steps to one exemplary image. This technique is effective because although the un-

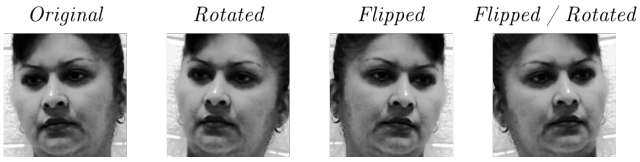

FIGURE 3.6: Example of one image being augmented

derlying image is fed to the model 4 times, the model doesn't recognize it as such, as the pixel values are shifted to different positions within the 2D array representing the image.

Downsampling

After upsampling all images, we had compiled a dataset that had four times as many positive examples as the original data set, but also four times as many negative examples. Therefore, we generally resorted to downsampling the negative examples. In downsampling, we sample from the majority class, without replacement, until enough examples matching the number of the minority class are sampled.

The precise algorithm by which examples were upsampled or downsampled depended on the experimental setting and is described further in detail in chapter 6.

Chapter 4

Model Architectures

In this chapter, we describe the initial CNN architecture used to classify our dataset, as well as some variants that led to our final architecture.

4.1 Baseline CNN

Ever since AlexNet [31] helped popularize deep CNNs through winning the ImageNet Challenge [50], convolutional neural networks have become the default for computer vision tasks. While the general trend has become to make CNNs deeper and more complex to achieve higher accuracy [58][62][61][20], this often comes at the expense of speed and hardware requirements.

These large models are often trained and used to make predictions on powerful cloud-computing infrastructures with many GPUs and large amounts of memory. In a federated learning setting, however, we cannot make any assumptions concerning the hardware that will power our model. In a healthcare setting, in particular, we can expect a very heterogeneous landscape of hardware infrastructures. Leading healthcare facilities would be equipped with modern computers harboring powerful CPUs or even GPUs, while facilities in particular in rural areas might only have access to significantly slower devices.

In part inspired by *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* [22] we decided to start experimenting with a lightweight architecture that could also be trained for a limited number of epochs on older and slower computing devices. In *Deep Structured Learning for Facial Expression Intensity Estimation* [65] the authors propose such a lean CNN structure as part of a more complex algorithm for working with the *Pain Expression Database*, among other datasets. Following this architecture, we designed our initial architecture, as seen in table 4.1. For the initial architecture, we employed a stride of (2, 2) in the convolutional layers and no max-pooling. We chose this approach under the hypothesis that compared with a stride of (1, 1) with subsequent max-pooling we can achieve a similar reduction in the surface area while reducing the computational effort, at the expense of making our feature extraction slightly coarser. The final convolutional layer is then followed by a 2x2 max-pooling layer, reducing the number of learnable parameters in the following dense layer by a factor of 4. The model's final dense layer is followed by a sigmoid activation function, as commonly used for a binary classification task with one output neuron, as discussed in chapter 2. This simple initial architecture was mainly used for quick experiments and tweaking the federated algorithm, early-stopping mechanisms, and evaluation procedures.

Layer Type	Output Shape	Param #
Input	(None, 215, 215, 1)	0
Conv2D	(None, 106, 106, 32)	832
ReLU	(None, 106, 106, 32)	0
Conv2D	(None, 51, 51, 64)	51,264
ReLU	(None, 51, 51, 64)	0
Conv2D	(None, 24, 24, 128)	204,928
ReLU	(None, 24, 24, 128)	0
MaxPooling2D	(None, 12, 12, 128)	0
Flatten	(None, 18432)	0
Dense	(None, 128)	2,359,424
BatchNormalization	(None, 128)	512
ReLU	(None, 128)	0
Dense	(None, 1)	129
Sigmoid	(None, 1)	0
Total		**2,617,089**

Optimizer:	Stochastic Gradient Descent
Loss Function:	Binary Cross Entropy

TABLE 4.1: Initial model architecture. Convolutional layers use VALID padding and a stride of 2x2. 'None' is a placeholder parameter for the batch size of the input batch.

4.2 Revised Architecture

After we were certain that all algorithms worked as expected, we also started modifying our initial architecture, to achieve the best model performance. We experimented with the following variables:

Regularization We experimented with adding dropout, L1 and L2 regularization as well as batch regularization to our model architecture, to prevent gradients from exploding or vanishing. Of these methods, batch normalization was the most effective. As previously discussed, batch-normalization helps smoothen the training landscape and tends to increase training speed / allow for larger learning rates. Adding batch normalization in-between the kernel and the ReLU activation layer for all convolutional layers yielded the best results.

Padding Changing convolutional layer padding from *VALID* to *SAME* was only a minor change to prevent unwanted shrinkage of the input shape of layers. Using **"SAME"** padding resulted in a minor performance improvement.

Stride vs. Max-Pooling We also experimented with swapping a stride of (2, 2) in convolutional layers with a (2 x 2) max-pooling layer. Stride and 2D-max-pooling are two very different operations, accomplishing the same goal: Decreasing the dimensions of the convolutional layer's output. Increasing the stride downsamples the "input features" of a convolutional layer, by letting the kernel "skip" calculations. Introducing a max-pooling to a convolutional layer downsamples the "output features", by only taking the maximum value from a given surface area. Thus, employing max-pooling instead of stride increases the computational effort but also the

amount of information carried forward into the next layer. In our specific use-case, however, we found that using a **stride of (2, 2) and no max-pooling** yielded the best results.

Additional parameters To increase the number of learnable parameters, we experimented with removing the final max-pooling layer and changing stride to (1, 1) in the convolutional layer. We also made the model deeper by adding additional dense and convolutional layers. None of these changes, however, yielded any notable improvements, but increased training time, and were thus discarded.

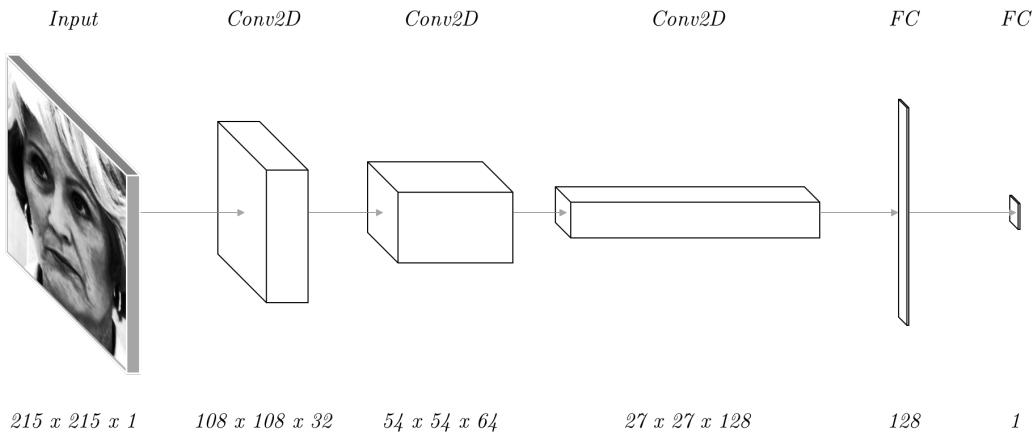

FIGURE 4.1: Final model architecture

The final model architecture which we found to yield the best results is depicted in table 4.2, and can be seen in figure 4.1. Each convolutional layer uses "SAME" padding.

4.2.1 Optimizer and learning rate

The optimizer is the algorithm by which the model performs each step of gradient descent. One example for an optimizer is *Stochastic Gradient Descent (SGD)*, which was explained in detail in chapter 2. We also experimented with other optimizers, such as *RMSProp* [48] and *Adam* [27]. However, we found that the standard Google Tensorflow implementation of these optimizers that we used to set our computational graphs is not suitable for a federated setting. More advanced optimizers such as *RMSProp* and *Adam* store historical information about the weights they optimize, and use this information to compute the magnitude of the step of gradient descent. After each communication round, however, the new, averaged model parameters are completely detached from this information, leading the optimizer to compute inaccurate values. Since we wanted to focus on optimizing each learning algorithm in isolation rather than also introducing a "federated optimizer" algorithm, we chose **SGD** for our experiments.

This merely required us to set an adequate learning rate for our optimizer. The learning rate is one of the most important hyper-parameters and needs to be set carefully. A learning rate that is too large can prevent the model from converging to an optimal solution, and a learning rate that is too small can make the training process too slow, time-consuming, and expensive. We experimented with learning rates ranging from $10e^{-2}$ to $10e^{-5}$ and finally decided on a learning rate of $10e^{-4}$.

Layer Type	Output Shape	Param #
Input	(None, 215, 215, 1)	0
Conv2D	(None, 108, 108, 32)	832
BatchNormalization	(None, 108, 108, 32)	128
ReLU	(None, 108, 108, 32)	0
Conv2D	(None, 54, 54, 64)	51,264
BatchNormalization	(None, 54, 54, 64)	256
ReLU	(None, 54, 54, 64)	0
Conv2D	(None, 27, 27, 128)	204,928
BatchNormalization	(None, 27, 27, 128)	512
ReLU	(None, 27, 27, 128)	0
MaxPooling2D	(None, 13, 13, 128)	0
Flatten	(None, 21632)	0
Dense	(None, 128)	2,769,024
BatchNormalization	(None, 128)	512
ReLU	(None, 128)	0
Dense	(None, 1)	129
Sigmoid	(None, 1)	0
Total		3,027,585

Optimizer:	Stochastic Gradient Descent
Loss Function:	Binary Cross Entropy

TABLE 4.2: Final model architecture. Convolutional layers use SAME padding and a stride of 2x2. 'None' is a placeholder parameter for the batch size of the input batch.

4.3 A note on ResNet50, VGGNet, and other deep model architectures

In trying to identify the best model architecture, we also experimented with common deep learning architectures that have been found to yield good results in standard computer vision tasks like the ResNet50 architecture presented in [19] or the VGGNet architecture presented in [57]. As commonly done when experimenting with these architectures, we loaded these models including their parameters which were pre-trained on the Imagenet [50] data set, discarded the last fully connected layers, and replaced them with the final fully connected layers as in our CNN architecture in table 4.2. We then froze all pre-trained layers and only trained the final fully connected layers. This approach follows the transfer-learning methodology introduced in chapter 2.

As a result, however, we saw training time increase substantially, and model accuracy remain the same or even decrease vs. our simpler architecture in table 4.2. The decrease in performance is commonly referred to as a "negative-transfer" in the literature and occurs when the original data distribution is not similar enough to the new distribution [63]. Concluding that common image recognition model architectures did not seem to be suitable for our specialized task of recognizing pain, we decided to focus solely on training our simple CNN model architecture end-to-end.

Chapter 5

Federated Personalized Learning

In this chapter, we build on the "privacy-advantaged" nature of the federated learning algorithm and propose a modified federated learning algorithm we dub **federated personalization**, designed to protect further the privacy of all clients that participate in jointly training a machine learning model.

5.1 Motivation

One motivation for federated learning, as explained in chapter 2, is increased privacy. Instead of transferring all data to a central server where a central model was learned, in federated learning, we only push the model parameters to the central server. Some research has shown, however, that even when only the abstract parameters of a model are shared with a central server, an adversary can use a generative adversarial network (GAN), to iteratively learn new pieces of information about another client's training data set [21] [40]. The authors of both papers manage to reconstruct prototypical samples of training images by training a GAN on each round of updates of parameters pushed to the central server. For [21], it must be said that the authors assume that the entire training corpus for one class belongs to one client only. They test their method on the MNIST dataset[8], a dataset of black and white images of handwritten digits from 0-9, and they experiment with each client holding the data of one digit only. While we are not aware of a real-world setting where this would be the case, the authors in [40] focus on the more realistic scenario where the training corpus for a given class is distributed across multiple clients. In both papers, however, the authors assume that all model parameters are shared with a central server.

5.2 Intuition

To increase the level of difficulty for an adversary to learn anything meaningful about other clients in a federated learning setting, we introduce a modified federated learning algorithm. Our algorithm discriminates which layers will be shared with a central server, based on a layer's position in the network.

In this novel approach, only the weights of the lower layers are sent to the central server for averaging, while the weights of the upper layers stay local. Without access to all layers in the network, an adversarial misses a critical building block to learn something about an "honest" participating client, making the federated averaging algorithm more secure.

Intuitively, the cost paid in performance for adding this privacy-preserving measure should be minimal: We know from transfer-learning, that the lower convolutional layers are responsible for extracting general features from the image data. Applying the federated optimization algorithm to these layers is imperative for training, to learn general features from as large a population as possible. However, we also know from transfer-learning that the final upper layers are mostly responsible for the final classification task. Thus, they do not necessarily need to learn from the entire population, and could potentially even benefit from learning only a specific client's data distribution, as certain details might only generalize well for that client, but not for the whole population. Putting these ideas into practice, we developed the following **federated personalization** algorithm.

5.3 The Federated Personalization Algorithm

The Federated PPersonalization algorithm is depicted in figure 5.1 and applied as follows:

(0) We initialize all local models to the same set of weights. (1) Each client then trains a local model, on the local data available for a specified number of epochs. (2) Once local training is complete, **only** the weights of the convolutional layers are sent to the central server for averaging. The weights of the final full-connected layers stay locally with the client, just like the data. (3) The convolutional weights are averaged and then used to update each local client model. At this point, however, the convolutional layers are "detached" from the fully-connected layers. Immediately using this updated model, which is a combination of globally averaged and locally tuned weights would yield poor results. We must, therefore, engage in "local fine-tuning". (4) Similarly to transfer-learning, we "freeze" the convolutional layers for each client. (5) We decrease the optimizer's learning rate for each client by some factor in order to avoid overshooting, especially for the first couple of steps of gradient descent. (6) We then train the local models for another specified number of epochs, slowly "reattaching" the fully-connected layers to the convolutional layers. After the fine-tuning is complete, we (7) re-increase the learning rate by the same factor as it was decreased by, and (8) unfreeze the convolutional layers. This approach produces the modified federated learning algorithm, seen in algorithm 2 for reference.

5.4 Local models

In the extreme case, the number of global layers in federated personalization is zero, implying that no weights are shared with the central server. To show that there is a benefit to sharing convolutional layers at all, as opposed to merely initializing parameters locally and never sending any parameters back to the server, we also define a benchmark algorithm. In this benchmark, a group of local models that only share the weight initialization step, and are then separated from one another is evaluated. In this setting, after the model parameters are initialized, each client is shut off from the central server. Thus, after model-initialization, there is no more additional communication between clients, and each client trains its local model in isolation, exclusively on local data.

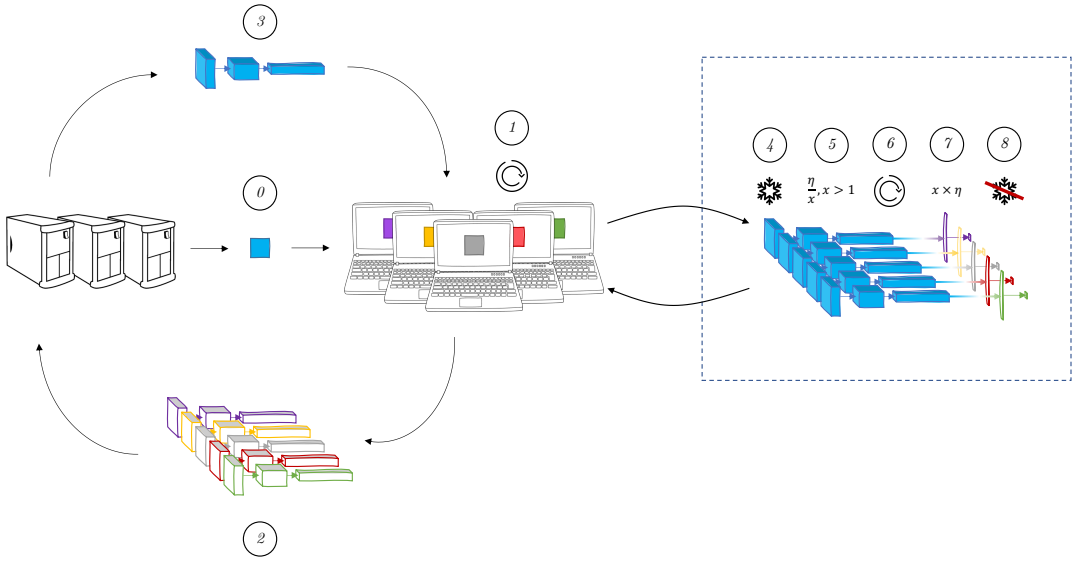

FIGURE 5.1: The Federated Personalization Algorithm

Algorithm 2 `FederatedPersonalization`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, F denotes the rounds that the local model is fine-tuned, and η is the learning rate. w_g denotes the global model parameters, w_l denotes the local model parameters.

procedure SERVER EXECUTES:
  initialize w_{g_0}
  initialize w_{l_0}
  for each round $t = 1, 2, \ldots$ **do**
    $m \leftarrow \max(C \times K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for each client k $\in S_t$ **in parallel do**
      $w^k_{g_{t+1}} \leftarrow$ CLIENTUPDATE(k, w_{g_t}, w_{l_t})
    $w_{g_{t+1}} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w^k_{g_{t+1}}$
    for each client k $\in S_t$ **in parallel do**
      CLIENTFINETUNING(k, w_{g_t}, w_{l_t})

procedure CLIENTUPDATE(k, w_g, w_l) ▷ Run on client k
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to E **do**
    $\{w_g \cup w_l\} \leftarrow \{w_g \cup w_l\} - \eta \nabla \ell(\{w_g \cup w_l\}; b)$ ▷ Update all layers
  return w_g to server

procedure CLIENTFINETUNING(k, w_g, w_l) ▷ Run on client k
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to F **do**
    $w_l \leftarrow w_l - \eta \nabla \ell(w_l; b)$ ▷ Update local layers only

Chapter 6

Experiments

6.1 Pre-training

6.1.1 Domain adaptation: Cold Start vs. Warm Start

We were first interested in learning how initializing model parameters differently would affect the learning process of our models. We experimented with two different settings, hereafter referred to as "cold start" and "warm start". For this purpose, we split our data set into two groups, Group 1 and Group 2. For easy reference, see a copy of table 3.1 introduced in chapter 3 again below:

Group 1				Group 2			
Person	No Pain	Pain	% Pain	**Person**	No Pain	Pain	% Pain
42	1,895	239	11%	**43**	1,028	92	8%
47	1,544	64	3%	**48**	798	84	9%
49	2,194	524	19%	**52**	2,503	106	4%
66	1,947	512	20%	**59**	640	133	17%
95	304	498	62%	**64**	1,394	155	10%
97	3,212	147	4%	**80**	896	1,068	54%
103	2,738	824	23%	**92**	1,031	471	31%
106	2,281	517	18%	**96**	2,175	178	7%
108	2,453	455	15%	**107**	1,599	442	21%
121	478	40	7%	**109**	1,724	179	9%
123	822	361	30%	**115**	1,184	99	7%
124	699	996	58%	**120**	1,490	76	4%

TABLE 6.1: Positive and negative examples by test subject and training group before any up- or downsampling. Test subject 101 was removed from the data altogether, as there were no positive ("Pain") examples of this test subject at all.

Cold Start

A cold start refers to training a model with randomly initialized parameters. Initial predictions of the model on unseen data are expected to be weak, and only gradually do they improve over time. In the "cold start" setting, after random initialization of the model parameters, we always started training and evaluating on **Group 2** immediately, ignoring **Group 1**. In this setting, the model never saw any data from Group 1.

Warm Start

For a "warm start", on the other hand, we employed the idea of **domain adaptation** introduced in chapter 2. We would first randomly initialize a model, and then train that model for a specified number of epochs on all test subjects in **Group 1**. In this step, **Group 1**'s positive examples (augmented and original) were included in the training set in their entirety. The much larger set of negative examples was down-sampled by sampling as many negative examples as there were positive examples at random without replacement into a subset, yielding a balanced training set. 20% of the balanced data set was separated as validation data, while the remaining 80% were used for training.

In a second step, we initialized a new model that would train on and evaluate Group 2, with the learned parameters of the model that was already trained on Group 1, rather than randomly, giving it a "warm start". Using the terminology introduced in chapter 2, while the *target domain* (i.e., predicting pain labels) remained the same using this approach, the input domain shifted from one group of 12 test subjects to another related group of 12 test subjects.

In comparing a cold-start with a warm start, we aimed at validating our hypothesis that also for the pain dataset, domain adaptation would produce more accurate predictions more quickly, and especially in early stages of the training process outperform the same model architecture with randomly initialized parameters, as illustrated in figure 6.1.

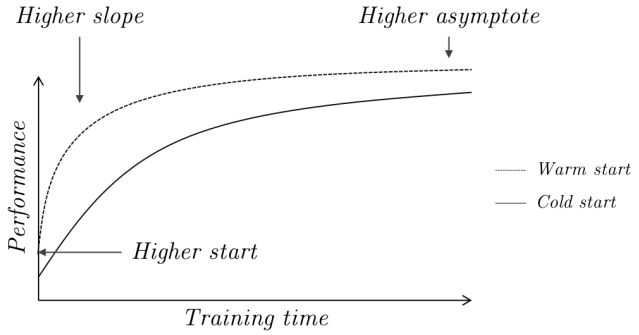

FIGURE 6.1: An exemplary chart showing two hypothetical learning curves, for learning a model with a warm start and a cold start, i.e. with and without transfer learning

6.1.2 Centralized vs. Federated Pre-Training

We also experimented with two different types of pre-training. In a centralized setting, the entire set of **Group 1** data was shuffled and passed to one model. Put into a production-level context, this would imply a central entity - e.g., a company providing machine learning services to healthcare providers - receiving access from its customers to some available training data. The data would be centrally stored and trained on, before it would be discarded, with the company shipping the pre-trained model to its customers. In this setting, we trained the centralized model for a maximum of **30 epochs**, and applied **early stopping** with a **patience of 5**. We also chose to restore the best model weights after early stopping, rather than continuing with the weights updated by the last epoch before stopping. As mentioned above, when training on **Group 1** we chose to always validate model loss on a random sample of

20% of the available data.

In a federated setting, after splitting the **Group 1** data into a training and a validation set, the training set was divided into 12 clients, one client for each test subject in the training data set. Similarly to centralized pre-training, we trained for a maximum of **30 communication rounds**, with **early stopping** and a **patience of 5**. For federated pre-training, the validation loss would be computed on the validation set every time a new global model was available, i.e., just after averaging the parameters of all 12 clients to form one set of parameters.

The federated pre-training setting would be more akin to the same company only providing the infrastructure to its customers for participating in a federated machine learning setting, as well as deploying a randomly initialized model on-site for each client, but never being entrusted with any data set.

6.2 Training

Knowing that our data set was distributed very unevenly, we created three different experimental settings with different levels of difficulty. In all three settings **Group 1** data served exclusively for model pre-training, and **Group 2** data for model training and evaluation.

6.2.1 Randomized Shards: Balanced Test Data

In the first iteration, we decided to train and test on a balanced data set. This set-up was designed purely to assess that our model architecture introduced in chapter 4 and the learning algorithms discussed in chapters 2 and 5 were capable of learning the training data, as well as to identify how much training data our models required in order to achieve strong performance levels.

We first balanced the entire **Group 2** data set to include 50% positive and 50% negative examples, using the up- and downsampling techniques discussed chapter 3. We then randomly split the data set into 60% train and 40% test data, thereby ignoring the temporal correlation of the images (i.e., the split into different therapy sessions). The 60% train data was then further split into cumulative shards of 1%, 5%, 10%, 20%, 30%, 40%, 50%, and 60% of all data, where each shard contained all images of the next smaller shard. Our models were trained for a maximum of 30 epochs on each shard and evaluated on the 40% of test data after each epoch.

6.2.2 Randomized Shards: Unbalanced Test Data

We also evaluated our learning algorithms on an unbalanced test-set. This setting was identical to the preceding one, only that we first split the untouched **Group 2** data set into 60% train and 40% test data. We then balanced the 60% training data, yielding the data distribution seen in table 6.2.

	No Pain	Pain	Total	% of Pain in Group
Train	7464	7464	14928	50%
Test	6601	1217	7818	16%

TABLE 6.2: Balanced training data for experimental setting *Randomized Shards*

This setting allowed us to get a better sense of the performance metrics that our models were able to achieve. Still, as image frames from all therapy sessions and all clients could be found in both the training and the test set, we considered the experimental setting as "artificial", because it made three assumptions that do not hold in a production setting:

1. The model had access to random samples of all training data in advance
2. The model only made predictions for known test subjects
3. All test subjects were present in each training and evaluation round

Conversely, we expect:

1. Training data to only become available sequentially
2. Unknown test subjects to be added to the data set from time to time
3. Test subjects to only participate in some therapy sessions / generate data in irregular intervals

6.2.3 Sessions

Consequently, we designed a third experimental setting that resembles how we would expect our models to be used in practice. In this setting, we did not assume that all data is available upfront, but rather that it is generated sequentially, for example, either in a continuous stream of video data taken from a patient's room or in regular therapy sessions.

The *UNBC-McMaster shoulder pain expression archive database* was recorded in sessions, with each test subject attending different sessions. Therefore, we assumed that each session only becomes available, once the model has been trained on the previous session. An example for the first two sessions can be seen in figure 6.2.

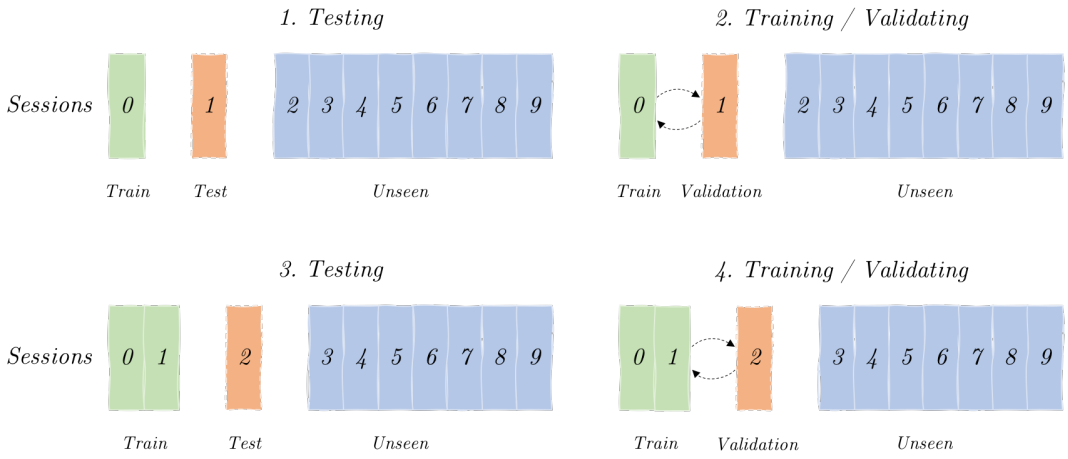

FIGURE 6.2: An example of training a model on session data. Sessions are zero-indexed. In (1) the model is tested on session 1. In (2) it is trained on session 0, using session 1 as a validation set to apply early stopping. In (3) session 1, the model is tested on session 2. In (4), session 1 has become part of the training data. The model is trained on session 0 and 1, and validated on session 2.

Table 6.3 shows that adding this temporal dimension to the experimental setting further adds to the data imbalance. While in a given session some test subjects might

have more positive examples than negative examples, others might have no positive examples at all.

	Positive Examples per Session										Total			
Person	0	1	2	3	4	5	6	7	8	9	# of Sessions	Pain	No Pain	Pain %
43	140				228						9	368	4,112	8%
48		148				188					7	336	3,192	10%
52	72						44		120	188	10	424	10,012	4%
59		532									2	532	2,560	17%
64	244	64	64		248						6	620	5,576	10%
80	1,052	536	484	484	660	792	264				7	4,272	3,584	54%
92	464	696			724						5	1,884	4,124	31%
96					112		512		88		9	712	8,700	8%
107	32		848	60	828						8	1,768	6,396	22%
109		600				116					8	716	6,896	9%
115	60		220	56	60						5	396	4,736	8%
120	116			188							8	304	5,960	5%

TABLE 6.3: Number of positive examples by session and test subject. Each test subject participated in as many consecutive sessions as specified in the column # *of Sessions*, starting with session 0. No number indicates no positive examples for that session (but negative examples, if the session index is smaller than # *of Sessions*).

Consequently, we developed algorithm 3 to ensure that in each session the training data set for each test subject would be balanced. In essence, for each test subject, for each session, the algorithm would check if there were positive examples for this test subject for this or previous sessions. If this were the case, the algorithm would sample at random 200 positive and 200 negative examples, with replacement, for this test subject, from all sessions that were available at this point. This approach would yield a training data set of 400 images per test subject per session, provided that there had been positive examples for this test subject in a previous session.

As seen in table 6.3, a threshold value of 200 strikes a good balance for most test subjects between further upsampling and thereby duplicating positive examples in some sessions, and downsampling and thereby excluding some positive examples from training in a given session.

6.3 Evaluation

To evaluate the effectiveness of our learning algorithms, we considered three sets of metrics: Aggregate average model performance, individual test subject performance, and performance per session/randomized shard of data.

Aggregate Average Model Performance The first set of metrics, aggregate average model performance, allowed us to compare different sets of hyper-parameters for all model architectures, as well as all learning algorithms with one another. However, while in a standard federated setting, the global model parameters are all averaged, resulting in identical models for all clients after each communication round, this is not the case for the **federated personalization** algorithm and the **local models** benchmark. Both of these algorithms produce a different model for each client, which added a layer of complexity to fairly evaluating all learning algorithms relatively to each other.

Algorithm 3 `SessionBalancing`. S denotes the current session and is an integer. P denotes a set of all positive examples. N denotes a set of all negative examples. T denotes the threshold value, set to 200 in our experiments.

procedure SESSIONBALANCING(S, P, N, T)
  $A = set()$
  for each Test Subject $C = \{43, 48, ..., 120\}$ **do**
    $P_C = set()$ ▷ Create two empty sets, for positive and negative examples
    $N_C = set()$
    while s <= S **do**
      P_C.append($P_{C,s}$)
      N_C.append($N_{C,s}$)
    if len(P_C) > 0 **then**
      $P_{sampled,C} = set()$
      $N_{sampled,C} = set()$
      while $len(P_{sampled,C}) < T$ **do**
        $P_{sampled,C}$.append(sample(P_C, replacement=False)
      while $len(N_{sampled,C}) < T$ **do**
        $N_{sampled,C}$.append(sample(P_C, replacement=False)
    A.append($P_{sampled,C}$)
    A.append($N_{sampled,C}$)
  return shuffle(A)

We, therefore, decided to evaluate each client based on their respective local model and computed a weighted average of the results of all clients after each communication round. In some sessions, however, there were no positive examples for certain test subjects, as seen in table 6.3. If we computed a simple weighted average across all test subjects in these sessions, the average of metrics such as precision, recall, and F1-Score would be unfairly heavily biased towards 0. Consequently, we constructed a mask based on table 6.3 and applied this mask on our results to only included a client in the weighted average calculation for a given session, if there were some positive examples for that client in that session. Applying this mask made aggregate performance evaluation metrics comparable across *clients, sessions, hyper-parameters, and learning algorithms*.

Individual Client Metrics and Session/Shard Metrics We also evaluated all learning algorithms on an individual level. We computed training, validation, and test set metrics for (1) performance for individual test subjects, and (2) performance over time, i.e., for each session and randomized shard.

Chapter 7

Results & Evaluation

In this chapter, we discuss the results obtained from the "sessions" experimental setting introduced in chapter 6.

7.1 Metrics

For evaluation, we focus on three metrics: Accuracy (ACC), Precision-Recall Area Under the Curve (PR-AUC), and the F1-Score (F1).

ACC Accuracy describes how many examples the model correctly classifies, across all examples. It is calculated as the number of correctly classified examples, divided by all examples.

PR-AUC The Area Under the Curve for the Precision-Recall curve is a performance metric typically used for imbalanced classes such as the pain data set. AUC summarizes the integral - or an approximation - of the area under the precision-recall curve. Precision refers to the ratio of true positive examples to all examples classified as positive and is calculated as $\frac{TP}{TP+FP}$. Recall refers to the ratio of correctly predicted true positive examples to all positive examples and is calculated as $\frac{TP}{TP+FN}$. The baseline for a random classifier for the PR-AUC is the total number of positive examples in the test set, divided by all examples, i.e., 16% in our case.

F1 The F1-Score is another measure that indicates how well the model classifies positive examples. It is calculated as $\frac{2 \times precision \times recall}{precision+recall}$ While PR-AUC represents the average performance given all possible probability thresholds between 0 and 1, the F1-Score is computed for a specific threshold, 0.5 in our case. I.e., the F1-Score's shown below assume that the model classifies an example as positive if the computed output probability is greater than 0.5.

7.2 Aggregate Results

7.2.1 Test-Set Results

Table 7.1 shows the weighted average performance of all learning algorithms presented in chapter 6, for the "Sessions" experimental setting, which most closely resembles a "real-world" setting that our learning algorithms would be confronted with. The values shown in table 7.1 were computed according to the methodology outlined in section 6.3, and represent and the average of 10 different random seeds, in the range of 123-132. For an acronym disambiguation please see table 7.2.

	Weighted AVG + STD		
Experiment	**ACC**	**PR-AUC**	**F1**
RANDOM	44 ± 15	31 ± 16	32 ± 2
BC-CNN	73 ± 12	54 ± 23	47 ± 24
BF-CNN	74 ± 12	53 ± 23	43 ± 21
C-CNN (N)	69 ± 17	49 ± 23	39 ± 25
C-CNN (C)	75 ± 13	58 ± 21	50 ± 22
F-CNN (N)	66 ± 16	49 ± 23	43 ± 27
F-CNN (C)	75 ± 11	**59 ± 23**	**52 ± 25**
F-CNN (F)	**76 ± 12**	**59 ± 23**	49 ± 25
FP-CNN (N)	69 ± 18	43 ± 19	34 ± 25
FP-CNN (C)	**76 ± 12**	56 ± 21	50 ± 24
FP-CNN (F)	**76 ± 13**	55 ± 22	44 ± 24
FL-CNN (N)	69 ± 18	43 ± 18	34 ± 26
FL-CNN (C)	75 ± 13	55 ± 21	47 ± 23
FL-CNN (F)	75 ± 14	54 ± 21	42 ± 23

TABLE 7.1: Comparison of aggregated results for all learning algorithms in (%). Standard deviation is computed between test subjects. Best results per metric are boldfaced.

As we can see in table 7.1 we achieve a model accuracy of 66% on the low end for a randomly initialized model trained with the federated learning algorithm F-CNN (N) with a standard deviation between test subjects of 16%. On the high end we achieve a model accuracy of 76% for the same algorithm but initialized with pre-trained parameters (F-CNN (F)) as well as both pre-trained federated personalization algorithms FP-CNN (C) and FP-CNN (F) with standard deviations of 12%, 12%, and 13%, respectively.

We also observe that all models outperform the randomly initialized, untrained classifier RANDOM by a wide margin, indicating that we are successfully learning to classify "pain" in individuals.

Furthermore, the table shows that models that have not been initialized to either the federated or the centralized baseline model perform significantly worse than those that have. This difference becomes even more apparent when looking at the PR-AUC and the F1-Score. Both measures show a clear difference between learning algorithms labeled (N) and those that are labeled (C) - centralized pre-training - or (F) - federated pre-training.

This confirms our hypothesis that domain-adaptation can help to build stronger models faster.

For all the following analyses, we will focus on the models that have been initialized with a centrally pre-trained model (C), which relates to the most likely business case as well. An aggregate summary and a condensed version of table 7.1 is shown for reference in table 7.3.

7.2.2 Training & Validation Set Results

Figure 7.1 and figure 7.2 show an example of the development over time of training/validation accuracy and loss, respectively. All graphs are averaged across seeds,

Acronym	Disambiguation	Explanation
RANDOM	Random	Random model weights, not trained
B	Baseline	A model trained on group 1, but not on group 2
C	Centralized	Vanilla centralized learning algorithm
F	Federated	Federated learning algorithm
FP	Federated Personalized	Federated personalized learning algorithm
FL	Federated Local	Local models trained independently of each other (performace averaged)
(N)	No Pre-training	Random parameter initialization
(C)	Centralized Pre-Training	Parameter initialization from a model trained with the centralized algorithm on group 1
(F)	Federated Pre-Training	Parameter initialization from a model trained with the federated algorithm on group 1

TABLE 7.2: Acronym Disambiguation

	Weighted AVG + STD		
Experiment	**ACC**	**PR-AUC**	**F1**
RANDOM	44 ± 15	31 ± 16	32 ± 2
BC-CNN	73 ± 12	54 ± 23	47 ± 24
C-CNN (C)	75 ± 13	58 ± 21	50 ± 22
F-CNN (C)	75 ± 11	**59 ± 23**	**52 ± 25**
FP-CNN (C)	**76 ± 12**	56 ± 21	50 ± 24
FL-CNN (C)	75 ± 13	55 ± 21	47 ± 23

TABLE 7.3: Comparison of aggregated results for all learning algorithms with centralized pre-training in (%). Standard deviation is computed between test subjects. Best results per metric are boldfaced.

with the shaded area representing one standard deviation. Each graph is partitioned into the ten sessions that we used to train and evaluate our models. Sessions differ in width because depending on the session, the early-stopping mechanism took effect after a different number of epochs. Standard deviation is only calculated for epochs where no seed had applied early stopping yet. We make a few observations here:

Training Accuracy and Loss

While generally trending upwards, at the beginning of each session, training accuracy dips (and loss spikes) - for some models more strongly than for others. These dips and spikes occur because the model receives new unseen data at the beginning of each session. The new data effectively works as a regularizer for the model and forces it to readjust its parameters.

Each model's training learning curve also follows a slightly different trajectory. The *centralized model* C-CNN (C) is training directly on all test subjects available, and in each session gets close to 100% training accuracy when early stopping takes effect and prevents the model from overfitting too strongly. The *federated learning*

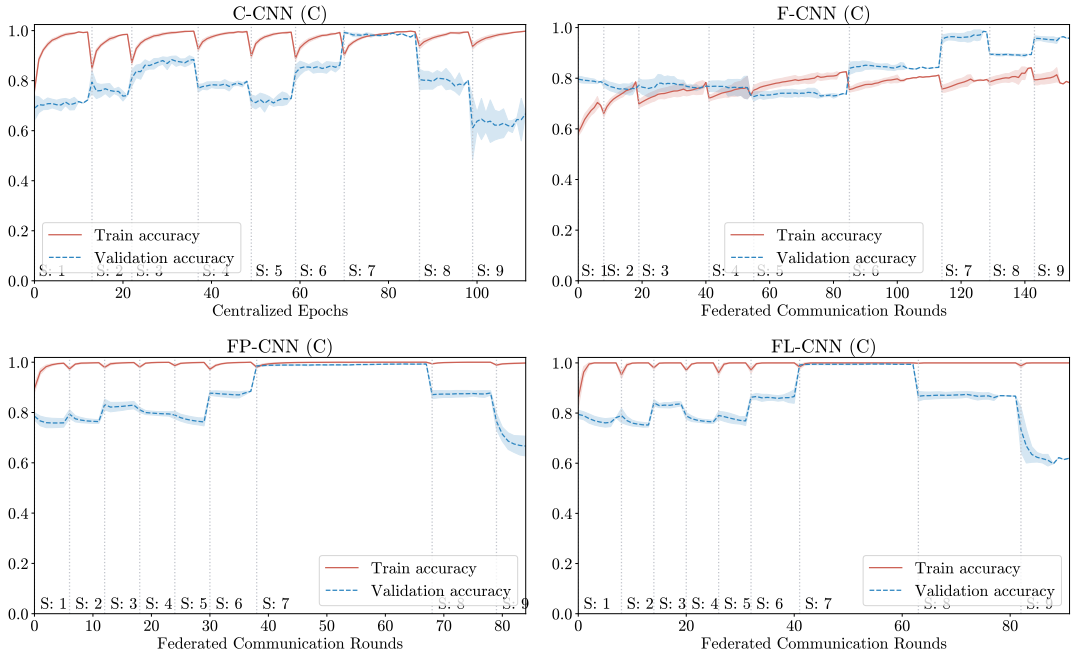

FIGURE 7.1: Mean Training/Validation Accuracy for Seeds 123-132, with 1 Standard Deviation

model F-CNN (C)'s training accuracy improves gradually, but on a lower level than the C-CNN (C). In their original paper [39] the authors conjecture that federated averaging can have a similar regularization effect to dropout, which seems to be the case here. As all 12 clients' local model's parameters are averaged after each communication round, the average parameters are not specifically fit on any test subject's data. Averaging seems to help to learn the underlying task better, however, explaining the best performance out of all models as seen in table 7.1.

The training accuracy curves for FP-CNN (C) and FL-CNN(C) remain very close to 100% for each session throughout the entire training process. Since for both methods the final fully-connected layers are not shared and averaged (and thereby not regularized as well), and are only learned on one test subject, this may hint at a case of overfitting the fully-connected layers on that test subject. A better regularization method might add value in this case.

Validation Accuracy and Loss

For all models validation loss spikes substantially in session 5. In session 5, the share of pain level "1" - the lowest pain level on the 16 point scale introduced in chapter 3, of all examples labeled with a pain level greater than 0 is very high, as can be seen in figure 7.3. Pain level "1" is very hard to separate from pain level "0" as a look at the bottom row in figure 3.2 showed.

Validation loss also drops close to 0 in session 7. Looking at table 6.3, we find that there are no positive examples in session 7. The drop thus indicates that the model is very good at identifying true negative examples.

We also observe that for federated approaches, the validation accuracy is less volatile compared to the centralized approach. Less volatility is another indicator that the federated learning algorithm likely works as a regularization mechanism and leads to less "overshooting" local minima compared vanilla SGD.

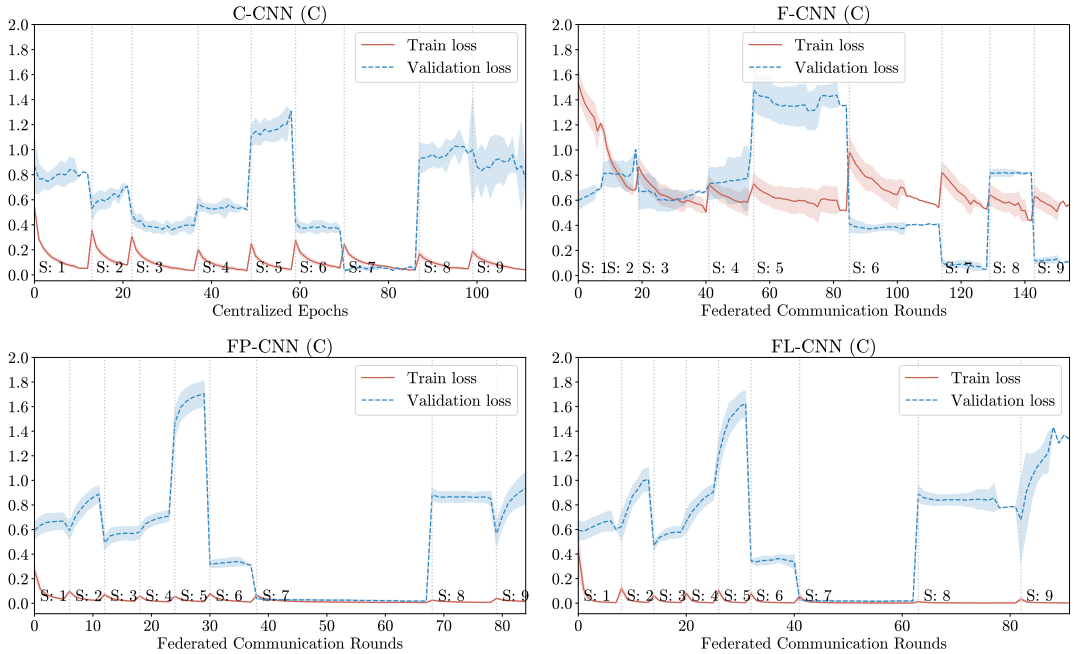

FIGURE 7.2: Mean Training/Validation Loss for Seeds 123-132, with 1 Standard Deviation

7.3 Individual Test Subject Results

Table 7.4 shows average performance across sessions, per test-subject. As outlined in section 6.3, for each test subject, a session was only included in the average, if there were positive examples for that subject in that session, as otherwise the PR-AUC and the F1-Score would be heavily biased towards 0, and not reflective of the true performance of the model.

7.3.1 Models

BC-CNN Table 7.4 shows that across test subjects the baseline is mostly outperformed, which again indicates that there is a benefit to applying domain adaptation and that an already learned classifier can still benefit from training on data that is more specific to the task. However, this does not hold for subject 120, as well as for most models and metrics for subject 52, where accuracy drops below the baseline initializer BC-CNN for most models. Dropping below the baseline that weights were initialized on is referred to as a **negative transfer** and is one of the limitations of the current implementation of our learning algorithms addressed in section 8.2.2.

C-CNN(C) vs. F-CNN(C) When comparing centralized learning with federated learning across test subjects, we can see that for some test subjects, the federated learning algorithm has an exceedingly positive impact, beating centralized learning by a wide margin. For others, however, federated learning performs much worse, so that on average, these effects even out. These differences are the largest for the F1-Score. If we consider a difference of $\geq 5\%$ as significant, federated learning outperforms centralized learning significantly for test subjects 52, 80, 92, and 109, and performs worse for test subjects 43, 64, and 115. This difference in performance could be related to the number of positive and negative examples per client that are fed to the models. While test subjects 80, 92 and 109 hold 3 of the four largest sets of

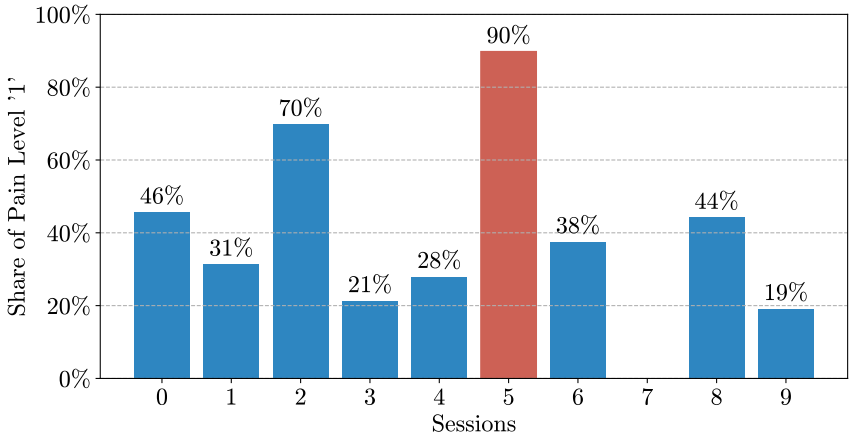

FIGURE 7.3: Share of pain level "1" of all positive examples, per session

positive examples to sample from for training, the number of positive examples for clients 43, 64, and 115 are all in the bottom half of our overall training population. We speculate that while a centralized model might retain some specific information about a test subject even if limited data is available, in a federated setting, clients with a less narrow data distribution generate more significant updates. These updates would nudge the federated averaging algorithm in their direction. While this can lead to better overall generalization (as seen by the overall model performance of F-CNN (C)), it may come at the expense of performing worse for some individuals.

F-CNN (C) vs. FP-CNN (C) When adding the additional privacy measure of only sending the convolutional layers to the central server, PR-AUC, and F1-Scores slightly worsen or stay the same for most test subjects, while accuracy improves. Due to the high class-imbalance of significantly less positive than negative examples, this hints at the fact that the model defaults more often to "negative" in its prediction. This can be an indicator of a decrease in knowledge about how a positive example looks like and a slower learning process overall as less information is shared between models.

FL-CNN (C) This observation is reinforced when looking at local model learning. Both PR-AUC and F1-Score are the worst out of all federated learning algorithms for almost all test subjects while overall accuracy improves for half of the test subjects. This indicates that there seems to be is a clear benefit to jointly learning a model in a federated setting, versus only deploying a pre-trained model, and only continuing to train on a local data set.

7.3.2 Selected Test subjects

48 While accuracy is above average for test subject 48, it performs significantly worse on PR-AUC and F1-Score than other test subjects, indicating that the model has difficulties identifying the test subject's positive examples. A look at figure 7.4 reveals why: All of test subject 48's examples of pain are labeled "1", the lowest pain value on the pain scale, which is very difficult from "0", i.e., "no-pain". The RANDOM classifier actually outperforms all learned classifiers for this test subject on F1-Score.

ACC (%)	**43**	**48**	**52**	**59**	**64**	**80**	**92**	**96**	**107**	**109**	**115**	**120**	**wt. Mean ± SD**
RANDOM	44	44	38	46	37	49	53	41	49	41	38	45	44 ± 15
BC-CNN	70	**78**	**92**	**48**	90	57	68	79	64	76	70	72	73 ± 12
C-CNN (C)	79	**78**	81	**48**	91	**62**	72	**84**	68	**77**	88	67	75 ± 13
F-CNN (C)	71	**78**	91	**48**	**92**	61	72	**84**	**69**	71	74	66	75 ± 11
FP-CNN (C)	82	**78**	87	**48**	91	**62**	**78**	**84**	67	**77**	**90**	**73**	**76 ± 12**
FL-CNN (C)	**83**	**78**	86	**48**	**92**	61	66	**84**	68	**77**	**90**	71	75 ± 13
PR-AUC (%)	**43**	**48**	**52**	**59**	**64**	**80**	**92**	**96**	**107**	**109**	**115**	**120**	**wt. Mean ± SD**
RANDOM	36	21	8	36	12	49	64	24	46	29	12	36	31 ± 16
BC-CNN	70	27	39	**62**	39	65	80	47	67	42	**55**	**70**	54 ± 23
C-CNN (C)	74	26	39	**62**	**46**	**70**	85	52	74	**48**	**55**	65	58 ± 21
F-CNN (C)	75	**28**	**46**	**62**	**46**	**70**	85	54	79	44	52	65	**59 ± 23**
FP-CNN (C)	**79**	27	40	**62**	42	64	87	49	**78**	45	53	65	56 ± 21
FL-CNN (C)	78	26	39	**62**	41	63	83	48	77	46	49	62	55 ± 21
F1 (%)	**43**	**48**	**52**	**59**	**64**	**80**	**92**	**96**	**107**	**109**	**115**	**120**	**wt. Mean ± SD**
RANDOM	35	**24**	10	37	14	44	51	25	42	31	15	36	32 ± 2
BC-CNN	24	5	32	**56**	25	**60**	79	8	59	30	35	**65**	47 ± 24
C-CNN (C)	64	7	30	**56**	**39**	53	67	45	62	32	46	61	50 ± 22
F-CNN (C)	27	7	**44**	**56**	25	58	**81**	**46**	63	**42**	35	62	**52 ± 25**
FP-CNN (C)	72	7	36	**56**	37	52	76	35	**68**	29	**47**	48	50 ± 24
FL-CNN (C)	**75**	7	35	**56**	34	49	53	37	**68**	33	42	46	47 ± 23

TABLE 7.4: Accuracy, Precision-Recall AUC, and F1-Score in (%) by test subject. Best model for each test subject is highlighted in bold.

59 Test subject 59 achieves an identical accuracy, PR-AUC, and F1 score across all models. This can be attributed to the fact that it only appears in session 1. When testing on session 1, all models' parameters are identical, since they were just initialized with the centrally pre-trained model parameters, thus yielding identical test scores for all models for this session.

92 Test subject 92 yields the best PR-AUC and F1 scores for our model. Again, looking at figure 7.4 shows why this is likely the case. All of test subject 92's positive examples are higher than "1" on the pain scale. This allows the models to differentiate more easily between positive and negative examples for this test subject.

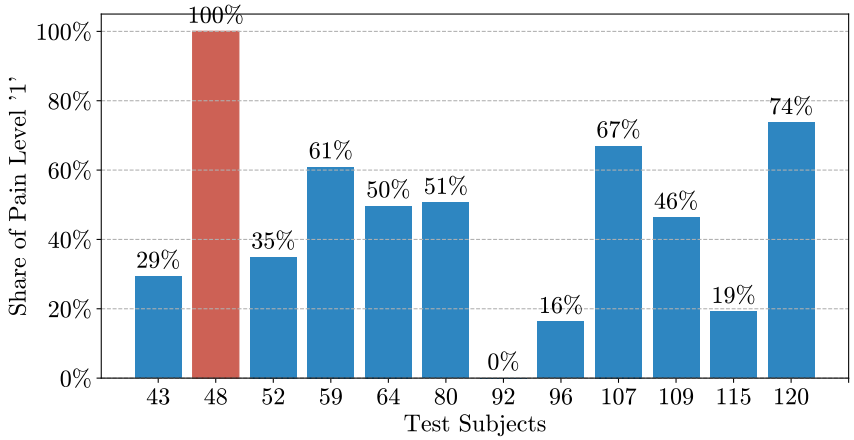

FIGURE 7.4: Share of pain level "1" of all positive examples, per test subject

ACC	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**wt. Mean ± SD**
RANDOM	47	43	44	45	44	44	NA	37	39	44 ± 20
BC-CNN	**68**	63	77	72	**68**	69	NA	**89**	**96**	73 ± 16
C-CNN (C)	**68**	74	82	78	64	79	NA	81	81	75 ± 17
F-CNN (C)	**68**	73	**78**	74	61	**80**	NA	88	**96**	75 ± 15
FP-CNN (C)	**68**	**76**	83	**78**	65	**80**	NA	85	89	**76 ± 16**
FL-CNN (C)	**68**	74	**84**	75	64	79	NA	84	88	75 ± 17
PR-AUC	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**wt. Mean ± SD**
RANDOM	41	34	26	36	35	28	NA	7	10	31 ± 22
BC-CNN	**53**	56	51	73	**39**	48	NA	10	80	54 ± 20
C-CNN (C)	**53**	**68**	56	78	38	48	NA	**14**	79	58 ± 19
F-CNN (C)	**53**	65	60	**79**	**39**	**56**	NA	12	**91**	**59 ± 19**
FP-CNN (C)	**53**	62	**63**	76	37	42	NA	13	78	56 ± 19
FL-CNN (C)	**53**	60	61	74	37	40	NA	13	77	55 ± 19
F1	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**wt. Mean ± SD**
RANDOM	40	34	28	36	**35**	29	NA	9	13	32 ± 6
BC-CNN	**49**	50	49	66	12	36	NA	3	75	47 ± 28
C-CNN (C)	**49**	**56**	50	66	5	29	NA	**28**	50	50 ± 23
F-CNN (C)	**49**	54	51	68	24	**41**	NA	15	**80**	**52 ± 26**
FP-CNN (C)	**49**	55	**56**	**69**	5	25	NA	26	60	50 ± 24
FL-CNN (C)	**49**	48	54	62	8	26	NA	26	58	47 ± 24

TABLE 7.5: Accuracy, Precision-Recall AUC, and F1-Score in (%) by session. Best model for each session is highlighted in bold.

7.3.3 Ranking by person

Table 7.6 displays another view at the data and shows again that the federated personalization algorithm can achieve comparable results to the federated learning algorithm while adding additional privacy measures. In this table, each number is the cumulative count of models that the given model outperformed for all test subjects. This table is computed by assigning a rank from 0 (worst) to 5 (best) to a model for a given test subject (e.g., 43), based on a performance metric, (e.g., accuracy). This is done for all test subjects, and the rank is then summed across test subjects.

7.4 Session Results

Table 7.5 slices the data by session rather than by test subject. As sessions reflect the temporal dimension of our data, it would be desirable that model performance improves by session over time for each learning algorithm.

7.4.1 Session Trends

Session 1 As detailed in section 7.3.2, all models are initialized to the baseline model weights prior to testing on session one. Consequently, all learning algorithms perform equally well on session 1 data.

	Weighted AVG + STD		
Experiment	**ACC**	**PR-AUC**	**F1**
RANDOM	0	0	7
BC-CNN	21	20	21
C-CNN (C)	27	33	30
F-CNN (C)	26	**37**	**36**
FP-CNN (C)	**34**	34	30
FL-CNN (C)	30	23	28

TABLE 7.6: Comparison of model ranking by test subject. Best results per metric are boldfaced.

Session 2 From session 1 to session 2, the performance for all learning algorithms improves, across metrics. Once again, this indicates that *domain adaptation* can be very beneficial to training a machine learning classifier. Since the baseline BC-CNN performs equally well for sessions 1 and 2 we can also conclude that this uptick in model performance for all other models is not merely due to session 2 containing data that is easier to classify, reinforcing the previous statement. Anticipating a little, in session four we see a significant performance uptick across models in terms of PR-AUC and F1-Score, for example, but also baseline performance increases substantially, hinting instead at a more easily classifiable data distribution rather than a substantial improvement in model performance.

Session 3 Session 3 only contains 9% positive examples, explaining the general increase in accuracy, as negative examples are generally easier for the model to identify. Looking at individual model performance and PR-AUC/F1, we notice that FP-CNN (C) and FL-CNN (C) see slight improvements in correctly identifying positive examples, while performance for C-CNN (C) and F-CNN (C) worsens. A critical difference between the two groups is that the former does not share the fully-connected layers with other models, while the latter does. We speculate that the data added to the training set in session two, which contains a large amount of pain level "1" data as seen in figure 7.3, does not help the models generalize very well. Instead, it is "test subject-specific" leading to the diverging performance for both model groups.

Session 4 As eluded to in paragraph **2**, the uptick in performance across all models in session 4 for PR-AUC and F1 is likely due to an easier test set since baseline performance also increases significantly. The low amount of level "1" pain examples (28%) and the high absolute number of positive examples (2,860, the highest out of all sessions) smoothing out "random error" point in that direction.

Session 5 Session 5 sees by far the sharpest drop in performance for PR-AUC and F1 scores. As discussed above, 90% of positive examples in session 5 are level "1" pain, which is exceedingly difficult to classify reliably. When comparing to the baseline, we see that also continued training only leads to modest performance improvements for this class.

Session 6 In session 6, we can observe the limitations of the current implementation of FP-CNN (C), as well as of the local model approach FL-CNN(C). For both algorithms, performance drops substantially below the baseline. Session 6 contains

test subjects 52, 80, and 96. All models have seen only seen 72 positive examples of test subject 52 until this point. It is, therefore misclassified by all models. However, session 6 only contains 44 additional examples for test subject 52, limiting its impact on the session average. Test subject 96 is classified comparatively well by all algorithms as its pain levels are mostly higher than "1" (see table 7.4).

Test subject 80's 264 positive examples, however, are misclassified entirely by FP-CNN (C) and FL-CNN (C). F-CNN (C) does significantly better here, correctly classifying 52 examples on average across seeds. While for other test subjects the reduction in shared information between clients only has a limited impact, client 80, holding the test subject that is most difficult to classify due to a large number of pain-level "1" examples, evidently benefits from receiving additional information from other clients for the last fully-connected layers.

Sessions 7-9 Sessions 7 to 9 hold very few positive examples compared to all other sessions (zero in session 7) and only barely contribute to the overall weighted mean and standard deviation. Moreover, the number of test subjects that participated in sessions 8 and 9 is 3 and 1, respectively, which limits the interpretability of average results for these sessions.

7.4.2 Ranking by session

"Ranking by session" works equivalently to "ranking by person", only that models are compared across sessions. While federated personalization still outperforms all other models on accuracy across sessions, it comes in a clear third after federated learning and centralized learning on PR-AUC and F1-Score.

	Weighted AVG + STD		
Experiment	**ACC**	**PR-AUC**	**F1**
RANDOM	0	0	7
BC-CNN	18	15	15
C-CNN (C)	17	24	17
F-CNN (C)	19	**27**	**24**
FP-CNN (C)	**25**	20	20
FL-CNN (C)	20	15	14

TABLE 7.7: Comparison of model ranking by session. Best results per metric are boldfaced.

7.5 Additional Findings

7.5.1 Improving individual update quality

With federated learning we are introducing one more hyper-parameter: Instead of only choosing a global number of communication rounds or epochs to run through, we can also tweak the local number of epochs that each client iterates over. We found early on that tweaking this parameter can have a positive impact on training. Increasing the number of local training steps is a purely heuristic method. There are no formal guarantees that increasing this number will ultimately yield better results, but in [39] McMahan et al. show in some simulations that it can improve

convergence. On average we found that increasing the number of local epochs from 1 to 5 adds $\tilde{2}\%$ to model accuracy.

In a production setting, adding local epochs would also not come at a high cost, since each client only possesses little data compared to the overall data volumes that are fed to the model.

7.5.2 Adding early stopping

Adding early stopping also dramatically improved convergence. We initially started training our models for a fixed number of 30 epochs on each session, without using a validation set. In doing so, the model tended to overfit on the given session data and would often perform poorly on the next session, and sometimes drop significantly below the baseline. Once we implemented early stopping with a patience of 5 epochs (i.e., training would stop if there were no improvements in validation loss after 5 epochs), and the feature of restoring the best model weights for this round of training, convergence improved significantly.

While for centralized training, we were able to leverage the Tensorflow Keras API, we designed a custom early stopping mechanism for federated learning. This custom mechanism would compute a weighted average loss for all clients, and stop training and restore each client's best model for the given training round, if the weighted loss across all clients stopped improving.

7.5.3 Flipping Group 1 and Group 2

Throughout this work, we always used test subjects in group 1 for pre-training and test subjects in group 2 for continued training, validation, and testing. In order to cross-validate that our findings were not dependent on this specific data distribution, we also flipped the groups and evaluated our learning algorithms on these new data distributions. First experiments indicated that the relative performance between models is not affected by changing the underlying data distribution.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

In this work, we show that we can learn a light-weight convolutional neural network to recognize pain in facial expressions in individuals. Irrespective of the learning algorithm our trained network substantially outperforms a random classifier on average. We also show that domain adaptation can be immensely helpful in accelerating convergence and improving test results compared to building a classifier from the ground up. Furthermore, we show that federated learning, conceived by Google in 2016, can be as effective as vanilla centralized learning in learning a well-performing classifier. Federated learning also tends to produce a more stable learning curve, likely since model averaging has a similar effect as common regularization techniques such as dropout. Our findings on whether standard federated learning can yield substantially better results compared to centralized learning for a majority of clients are inconclusive. However, based on our results, we conjecture that in a federated setting, clients with better data nudge the average model more strongly in their favour.

Moreover, we present an evolution of the federated averaging algorithm, which we dub *federated personalization*. Our algorithm adds one more layer of privacy preservation to the federated learning algorithm, by only allowing a fixed subset of model parameters to be shared with a central server. We propose that in a neural network, these parameters should be part of the lower levels of the network, which typically extract the input data's general features. The upper levels are kept on local devices exclusively, to prevent the curious client from learning anything meaningful about other honest clients that participate in learning a federated model.

We show that even in limiting the number of shared model parameters in such a deterministic manner, we can still learn a strong model that is only modestly outperformed by the original federated learning algorithm.

Finally, we show that federated learning yields better results than a group of jointly initialized models that are subsequently shut off from one another to only learn on their respective local data sets.

8.2 Future Work

8.2.1 Painful data and model architectures

The painful data set is a difficult data set to learn. Especially the difference between lower levels of pain and images where test subjects do not experience any pain is very nuanced. Moreover, the data set is very imbalanced. To improve the baseline

for all our proposed learning algorithms, we suggest future research to continue experimenting with different neural network architectures, as well as other types of classifiers. For this work, we focused on a lightweight CNN, but due to the temporal correlation of the video data, an LSTM architecture could help improve performance. Due to the binary nature of our classification task, also a simpler classifier like a support vector machine might be worth investigating. To address the strong imbalance of the dataset, implementing a different loss function such as *hinge loss*, or *weighted binary crossentropy* might yield performance improvements. Finally, in "Deep Structured Learning for Facial Action Unit Intensity Estimation" the authors propose a novel Copula CNN architecture to account for the structural dependence of the facial action units used to determine the aggregate pain score[65]. Applying the federated learning and federated personalization algorithm to this architecture is another direction for future research that we propose.

8.2.2 Algorithmic modifications

We also suggest evolving further the federated personalization algorithm proposed in this work. In some sessions, some individuals experienced a decrease in performance compared to the pre-trained baseline. This performance decrease can be attributed to the new information that the models were trained on, commonly referred to as a negative transfer. To prevent the negative transfer, we suggest a few alterations to the federated algorithms with which we experimented.

Random layer sampling and additional privacy measures

While our extension of the federated learning algorithm, federated personalization, offers additional practical privacy benefits, changing the deterministic way by which the averaging layers are chosen for a random approach is an exciting direction for future research. Moreover, adding more formal privacy guarantees, such as differential privacy to our federated algorithms, is another direction for future work.

Validation Buffer

As explained in section 7.5.2, we compute a global weighted average validation loss based on which we decide which set of weights will be pushed from the central server to the clients in order to instate the new local model. However, if a given test subject is included in the training session, but not in the validation session, the average validation loss will not be specific to that test subject. A solution could be to instate a validation buffer, where a test subject's data remains in the validation set until new data for that test subject is generated. Only at this point, the new data becomes part of the validation data, and the old data is moved into the training data set. Implementing this validation buffer also means that clients who only participate in one session never contribute to training the model. This makes sense, as for these clients, we could imagine a scenario where a patient enters the hospital and leaves again after one day. In these cases, the model does not need to learn anything specific about this test subject, but rather be able to identify the patient's pain level once he or she enters the hospital for the first time.

Fallback models

In federated learning, we have the advantage of different clients being able to store different versions of the same model. We think it is worth harnessing this advantage

and experimenting with storing two models locally. Initially, a global model could be initialized and distributed to all clients. During training, after the global averaging step, the updated model would then be pushed to all clients, but instead of replacing the old model, a second model would be created. Both models would then be benchmarked against the local validation set. As long as the new model from the server does not outperform the old local model, the local model is not replaced. Only once the new global model leads to improved performance on the local validation set, the local model is replaced. This would ensure that each local model never drops in performance below the baseline architecture, and thus negative transfer is prevented.

Appendix A

Running the code

A.1 federated-machine-learning

Clone this project from:

```
git clone https://github.com/ntobis/federated-machine-learning.git
```

Go into the directory `federated-machine-learning`. I recommend to create a virtual environment.

```
virtualenv venv
source venv/bin/activate
```

To install all dependencies run:

```
pip install -r requirements.txt
```

If you have the UNBC-McMaster shoulder pain expression archive database, which is required to run this code out-of-the-box, create the following folders

- Data
- Data/Raw Data/
- Data/Preprocessed Data/
- Data/Augmented Data/

and move the images into the Raw Data folder.

Alternatively, you should be able to run the following commands from the project's root directory:

```
mkdir Data
cd Data/
mkdir Raw\ Data
mkdir Preprocessed\ Data
mkdir Augmented\ Data
mv -r [folder where UNBC database is on your computer] Raw\ Data/
```

A.1.1 How to run the code

Data Pre-Processing

First, the image data will need to be pre-processed

1. Navigate to federated-machine-learning/Notebooks and run the notebook Data Pre-Processing.ipynb
2. "Run All", and the pre-processing steps "histogram equalization" and "image flipping", and "image rotation/cropping" will be applied.

Running Experiments

Shell scripts There are 2 shell scripts that can be executed out-of-the-box.

```
./execute_local.sh
./execute_GCP.sh
```

`execute_local.sh` is recommended when running an experiment on an ordinary machine. `execute_GCP.sh` includes 2 sets of additional parameters: If you run this code on the Google Cloud Platform, you can specify

```
--project [your GCP project, e.g., centered-flash-251417]
--zone [your GCP VM zone, e.g., us-west1-b]
--instance [your GCP instance, e.g., tensorflow-1-vm]
```

and the instance will automatically be stopped once your experiment is completed. If you have a Twilio account (see more under www.twilio.com), you can also provide your account credentials, as well as a receiver phone number, to receive a text message once training is completed, or if an error occurs.

```
--sms_acc [your Twilio account, typically of the format ACeabXXXXXXXXXXXXX]
--sms_pw [your Twilio password, typically of the format eab57930XXXXXXXXXX]
--sender [your Twilio sender number, typically of the format +4418XXXXXXXX]
--receiver [your personal phone number, e.g., +4477XXXXXXXX]
```

Most important functions

Experiments.py `Experiments.py` contains the functions responsible for running all experimental settings. See below for a description of the most important functions:

main(seed=123, shards_unbalanced=False, shards_balanced=False, sessions=False, evaluate=False, dest_folder_name='', args=None) The `main()` function initializes the tensorflow optimizer, loss function, and metrics to track. It also executes experiment_pain(), which runs all experiments. We also specify the shards for the "randomized shards" experiment in the main function, all at the top.

The main function then contains 4 blocks, all of which can be controlled with the function parameters. The first three blocks run the experimental settings "randomized shards, unbalanced test data", "randomized shards, balanced test data", and "sessions" respectively. Each experimental block runs the experiment_pain() function 11 times, once for each experimental setting. The final block executes the evaluate_baseline() function.

experiment_pain(algorithm='centralized', dataset='PAIN', experiment='placeholder', setting=None, rounds=30, shards=None, balance_test_set=False, model_path=None, pretraining=None, cumulative=True, optimizer=None, loss=None, metrics=None, local_epochs=1, model_type='CNN', pain_gap=(), individual_validation=True, local_operation='global_averaging') The `experiment_pain()` function allows to fine tune each experimental setting. It defines if a given experiment should be centralized or federated, which type of federated algorithm should be run. It defines if pre-training should be applied, as well as how many global and local epochs should be run.

It is recommended to limit changes to the code to the parameters of this function, if the general features should be maintained and only different experimental settings (optimizers, number of epochs, etc.) are expected to be tried.

run_pretraining(dataset, experiment, local_epochs, optimizer, loss, metrics, model_path, model_type, pretraining, rounds, pain_gap) `run_pretraining()` returns one of 4 models depending on the arguments provided: A Tensorflow model loaded from file, a model that was pre-trained with the centralized algorithm, a model that was pre-trained with the federated algorithm, or a randomly initialized model.

run_shards(algorithm, cumulative, dataset, experiment, local_epochs, model, model_type, rounds, shards, pain_gap, individual_validation, local_operation, balance_test) `run_shards()` runs the randomized shards experiment. It follows the algorithm described in chapter 5 of the thesis.

run_sessions(algorithm, dataset, experiment, local_epochs, model, model_type, rounds, pain_gap, individual_validation, local_operation) `run_shards()` runs the sessions experiment. It follows the algorithm described in chapter 5 of the thesis.

Model_Training.py `Model_Training.py` contains the different learning algorithms described in chapter 5 of the thesis. The two most important functions are:

federated_learning(model, global_epochs, train_data, train_labels, train_people, val_data, val_labels, val_people, val_all_labels, clients, local_epochs, individual_validation, local_operation, weights_accountant) The `federated_learning()` function governs all federated algorithms. It iterates over a specified number of communication rounds, and after each round computes the custom training and validation metrics, based on the algorithm it is currently running. It also implements a custom `EarlyStopping` class, that monitors average validation loss across clients and restores the best model weights, once training has ended.

train_cnn(algorithm, model, epochs, train_data, train_labels, val_data, val_labels, val_people, val_all_labels, individual_validation) `train_cnn()` is the central training function. It implements early stopping if the algorithm is centralized, (for federated algorithms this is handled by `federated_learning()` and allows to individually track training and validation metrics for clients with custom callbacks.

Weights_Accountant.py Finally, the `WeightsAccountant` tracks the weights of all clients in a federated setting. It performs the Federated Averaging algorithm as well as the Federated Personalization algorithm. It also tracks all weights in the Local Model experimental setting.

A.1.2 Evaluation

Two notebooks are helpful for results evaluation. The Notebook "Table Preparation" serves to quickly generate an overview of the results achieved by the experiments stored in the folder "Results". The Notebook "MSc Thesis Visualizations" generates the majority of tables and figures seen in the thesis. Simply "Run All" to generate all figures and tables.

Bibliography

[1] *A shortage of staff is the biggest problem facing the NHS*. 2019. URL: https://www.economist.com/britain/2019/03/23/a-shortage-of-staff-is-the-biggest-problem-facing-the-nhs.

[2] Ahmed Bilal Ashraf et al. "The painful face–pain expression recognition using active appearance models". In: *Image and vision computing* 27.12 (2009), pp. 1788–1796.

[3] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[4] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[5] Léon Bottou, Frank E Curtis, and Jorge Nocedal. "Optimization methods for large-scale machine learning". In: *Siam Review* 60.2 (2018), pp. 223–311.

[6] Rich Caruana, Steve Lawrence, and C Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*. 2001, pp. 402–408.

[7] Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2933–2941. URL: http://papers.nips.cc/paper/5486-identifying-and-attacking-the-saddle-point-problem-in-high-dimensional-non-convex-optimization.pdf.

[8] Li Deng. "The MNIST database of handwritten digit images for machine learning research [best of the web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[9] P Ekman, W Friesen, and J Hager. "Facial action coding system: Research Nexus". In: *Network Research Information, Salt Lake City, UT* 1 (2002).

[10] Eduardo Castelló Ferrer et al. "Robochain: A secure data-sharing framework for human-robot interaction". In: *arXiv preprint arXiv:1802.04480* (2018).

[11] Atul Gawande. *Checklist manifesto, the (HB)*. Penguin Books India, 2010.

[12] Robin C. Geyer, Tassilo Klein, and Moin Nabi. "Differentially Private Federated Learning: A Client Level Perspective". In: *CoRR* abs/1712.07557 (2017). arXiv: 1712.07557. URL: http://arxiv.org/abs/1712.07557.

[13] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Domain adaptation for large-scale sentiment classification: A deep learning approach". In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, pp. 513–520.

[15] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[16] *Guide to the General Data Protection Regulation (GDPR)*. URL: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/.

[17] Stephen Hardy et al. "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption". In: *CoRR* abs / 1711.10677 (2017). arXiv: 1711.10677. URL: http://arxiv.org/abs/1711.10677.

[18] Florian Hartmann. *Federated Learning*. 2018.

[19] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[21] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. "Deep models under the GAN: information leakage from collaborative deep learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 603–618.

[22] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[23] Muzammil Hussain et al. "The landscape of research on smartphone medical apps: Coherent taxonomy, motivations, open challenges and recommendations". In: *Computer Methods and Programs in Biomedicine* 122.3 (2015), pp. 393 –408. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2015.08.015. URL: http://www.sciencedirect.com/science/article/pii/S0169260715002254.

[24] Andrey Ignatov et al. "Ai benchmark: Running deep neural networks on android smartphones". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 0–0.

[25] Katarzyna Janocha and Wojciech Marian Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: *CoRR* abs/1702.05659 (2017). arXiv: 1702.05659. URL: http://arxiv.org/abs/1702.05659.

[26] Yoo Jung Kim and JinYoung Han. "Why smartphone advertising attracts customers: A model of Web advertising, flow, and personalization". In: *Computers in Human Behavior* 33 (2014), pp. 256 –269. ISSN: 0747-5632. DOI: https://doi.org/10.1016/j.chb.2014.01.015. URL: http://www.sciencedirect.com/science/article/pii/S074756321400020X.

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

[28] Jakub Konečnỳ et al. "Federated learning: Strategies for improving communication efficiency". In: *arXiv preprint arXiv:1610.05492* (2016).

[29] Jakub Konecný et al. "Federated Optimization: Distributed Machine Learning for On-Device Intelligence". In: *CoRR* abs/1610.02527 (2016). arXiv: 1610.02527. URL: http://arxiv.org/abs/1610.02527.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[32] Huoran Li et al. "Characterizing Smartphone Usage Patterns from Millions of Android Users". In: *Proceedings of the 2015 Internet Measurement Conference*. IMC '15. ACM, 2015, pp. 459–472. ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815686. URL: http://doi.acm.org/10.1145/2815675.2815686.

[33] Geert Litjens et al. "Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis". In: *Scientific reports* 6 (2016), p. 26286.

[34] P. Lucey et al. "Painful data: The UNBC-McMaster shoulder pain expression archive database". In: *Face and Gesture 2011*. 2011, pp. 57–64. DOI: 10.1109/FG.2011.5771462.

[35] Patrick Lucey et al. "Automatically detecting pain in video through facial action units". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 41.3 (2010), pp. 664–674.

[36] Patrick Lucey et al. "Automatically detecting pain using facial actions". In: *2009 3rd International Conference on Affective Computing and Intelligent Interaction and Workshops*. IEEE. 2009, pp. 1–8.

[37] Brendan McMahan and Daniel Ramage. *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. 2017. URL: http://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[38] Brendan McMahan and Daniel Ramage. *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. 2017. URL: http://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[39] H. Brendan McMahan et al. "Federated Learning of Deep Networks using Model Averaging". In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629.

[40] Luca Melis et al. "Exploiting unintended feature leakage in collaborative learning". In: *arXiv preprint arXiv:1805.04049* (2018).

[41] Andrew Y Ng. "Feature selection, L 1 vs. L 2 regularization, and rotational invariance". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 78.

[42] Batch Normalization. "Accelerating deep network training by reducing internal covariate shift". In: *CoRR.–2015.–Vol. abs/1502.03167.–URL: http://arxiv.org/abs/1502.03167* (2015).

[43] Chigozie Nwankpa et al. "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[44] Omprakash Patel, Yogendra P. S. Maravi, and Sanjeev Sharma. "A Comparative Study of Histogram Equalization Based Image Enhancement Techniques for Brightness Preservation and Contrast Enhancement". In: *Signal & Image Processing : An International Journal* 4 (Nov. 2013). DOI: 10.5121/sipij.2013.4502.

[45] Kenneth M Prkachin. "The consistency of facial expressions of pain: a comparison across modalities". In: *Pain* 51.3 (1992), pp. 297–306.

[46] Kenneth M Prkachin and Patricia E Solomon. "The structure, reliability and validity of pain expression: Evidence from patients with shoulder pain". In: *Pain* 139.2 (2008), pp. 267–274.

[47] Douglas Reynolds. "Gaussian Mixture Models". In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil K. Jain. Boston, MA: Springer US, 2015, pp. 827–832. ISBN: 978-1-4899-7488-4. DOI: 10.1007/978-1-4899-7488-4_196. URL: https://doi.org/10.1007/978-1-4899-7488-4_196.

[48] Martin Riedmiller and Heinrich Braun. "RPROP-A fast adaptive learning algorithm". In: *Proc. of ISCIS VII), Universitat*. Citeseer. 1992.

[49] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[50] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[51] Shibani Santurkar et al. "How does batch normalization help optimization?" In: *Advances in Neural Information Processing Systems*. 2018, pp. 2483–2493.

[52] Office for Civil Rights HHS Office of the Secretary and Ocr. *Summary of the HIPAA Security Rule*. 2013. URL: https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html.

[53] Micah J. Sheller et al. "Multi-Institutional Deep Learning Modeling Without Sharing Patient Data: A Feasibility Study on Brain Tumor Segmentation". In: *CoRR* abs/1810.04304 (2018). arXiv: 1810.04304. URL: http://arxiv.org/abs/1810.04304.

[54] H. Shin et al. "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning". In: *IEEE Transactions on Medical Imaging* 35.5 (2016), pp. 1285–1298. ISSN: 0278-0062. DOI: 10.1109/TMI.2016.2528162.

[55] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[56] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[57] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[58] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[59] Olivia Solon. "Facebook says Cambridge Analytica may have gained 37m more users' data". In: *The Guardian* 4 (2018).

[60] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[61] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: *CoRR* abs/1602.07261 (2016). arXiv: 1602.07261. URL: http://arxiv.org/abs/1602.07261.

[62] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[63] Lisa Torrey and Jude Shavlik. "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global, 2010, pp. 242–264.

[64] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

[65] Robert Walecki et al. "Deep Structured Learning for Facial Action Unit Intensity Estimation". In: *arXiv e-prints*, arXiv:1704.04481 (2017), arXiv:1704.04481. arXiv: 1704.04481 [cs.CV].

[66] Dayong Wang et al. "Deep learning for identifying metastatic breast cancer". In: *arXiv preprint arXiv:1606.05718* (2016).

[67] Zhi Wei et al. "Large Sample Size, Wide Variant Spectrum, and Advanced Machine-Learning Technique Boost Risk Prediction for Inflammatory Bowel Disease". In: *The American Journal of Human Genetics* 92.6 (2013), pp. 1008 – 1012. ISSN: 0002-9297. DOI: https://doi.org/10.1016/j.ajhg.2013.05.002. URL: http://www.sciencedirect.com/science/article/pii/S0002929713002152.

[68] Nancy Wells, Chris Pasero, and Margo Mccaffery. *Chapter 17. Improving the Quality of Care Through Pain Assessment and Management*.

[69] Michael Winnick. *Putting a Finger on Our Phone Obsession*. 2016. URL: https://blog.dscout.com/mobile-touches.

[70] Timothy Yang et al. "Applied Federated Learning: Improving Google Keyboard Query Suggestions". In: *CoRR* abs/1812.02903 (2018). arXiv: 1812.02903. URL: http://arxiv.org/abs/1812.02903.