Chapter 1

Introduction

1.1 Deep Learning Models

1.1.1 Multilayer Perceptrons

A Multilayer Perceptron (MLP), also known as feedforward neural network, is a mathematical model mainly used in supervised learning. Essentially, an MLP is a directed acyclic graph which represents the composition of functions. Each function or **layer** is a collection of neurons. A neuron is defined as:

$$y = f(\mathbf{x}; \theta) = \omega^T \mathbf{x} + b, \qquad \theta = (\omega, b) \tag{1.1}$$

The term *perceptron* refers to a linear classifier consisting of one layer [16], $f(\mathbf{x}; \theta) = \mathbb{I}(\omega^T \mathbf{x} + b > 0)$. Here, $\mathbb{I}(a > 0)$ is the Heaviside function, which is non-differentiable. In MLP, the activation function from the original perceptron $\mathbb{I}$ is usually replaced by another differentiable function $\psi \colon \mathbb{R} \to \mathbb{R}$. The internal layers of MLP are usually named **hidden layers**, the last one is called **output layer**. The dimensionality of the hidden layers determines the **width** of the neural network [6]. Each layer l consists of many units $\mathbf{z}_l$ which are computed as a linear transformation of the units from the previous layer $l-1$ passed element-wise through the activation function [13]:

$$\mathbf{z}_l = \psi_l(\mathbf{W}_l \mathbf{z}_{l-1} + \mathbf{b}_l)$$

We can choose different activation functions that will define our model and impact in the performance of the training. If we use a linear activation function $\psi_l(x) = K_l x$ then our neural network becomes just a linear model [13], that's why usually non-linear activation functions are used, since we would like to capture non-linear relationships between the input data and the output. Also, the Universal Approximation Theorem states that a neural network with a single hidden layer and non-linear activation functions can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to any desired degree of accuracy. Since the goal of an MLP is to approximate some function f^* (for example, for a classifier $y = f^*(\mathbf{x})$), non-linear activation functions are necessary to achieve this level of approximation. However, they can be used when we know the problem is linearly separable (think of two separate clusters of points in $\mathbb{R}^2$ in a classifier model) or in the output layer, for example when the MLP acts as a regression model.
If we change the Heaviside function by the sigmoid (logistic) function $\sigma(x) = \frac{1}{1+e^{-x}}$ we get a smooth approximation of $\mathbb{I}$. Another choice of activation function could be $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ which has a similar shape but its image is $(-1, 1)$. Both are valid activation function and have been used. However, since we need to compute gradients in the optimization process in order to update the weights, a $\mathbf{0}$ gradient would be a problem, since it will make hard to train a model using gradient descent (vanishing gradient problem). Both functions have an almost horizontal slope (gradient near 0) for large positive and negative inputs. Also, $\|\frac{d}{dx}\sigma(x)\| = \|\sigma(x)(1 - \sigma(x))\| \leq \frac{1}{4} < 1$, therefore several multiplications will quickly approximate to zero. Although they are used in practice (for example, $\sigma(\cdot)$ is used in the output layer for binary regression problems), in order to train deep models we need non-saturating activation functions for the hidden layers. One

of the most used is *rectified linear unit*: $ReLU(x) = \max(a, 0) = a\mathbb{I}(a > 0)$. The gradient of $ReLU'(z) \neq 0$ as long as z is positive: this function don't require input normalization to prevent them from saturating. As stated in [5], the rectifier activation function allows a network to easily obtain sparse representations, the *hard* saturation (gradients being exactly 0) help supervised training: experimental results suggest that networks with ReLU activation functions in the hidden layers have better convergence performance than using sigmoid [9]. There has been a lot of activation functions proposed in the last years, [TERMINAR ESTO] .

The neural network tries to approximate the target function $\mathbf{y} = F(\mathbf{x})$, the *true* relation between the variables. The network, $f(x; \theta)$, *learns* by searching the parameters $\boldsymbol{\theta}$ that minimizes a loss function $J(\boldsymbol{\theta})$, which measures the distance between the output and the target or the proximity between probability densities of random variables [1]. If we consider a k-class classification problem with $\mathcal{X} \subset \mathbb{R}^d$ the feature space and $\mathcal{Y} = \{1, \ldots, k\}$ the label space, given the dataset $D = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)\}$ and a MLP $f \colon \mathcal{X} \to \mathbb{R}^k$ with a softmax as the output layer. For any loss function $J(\boldsymbol{\theta}) = J(f(\mathbf{x}, \boldsymbol{\theta}), y)$, assuming there is a joint distribution $P(\mathbf{x}, y)$ over $\mathcal{X}$ and $\mathcal{Y}$, the **risk** of f is defined as $R_J(f) = \mathbb{E}\Big[J(f(\mathbf{x}; \boldsymbol{\theta}), y)\Big] = \int J(f(\mathbf{x}; \theta), y) dP(\mathbf{x}, y)$ and the **empirical risk** is defined as $\hat{R}_J(f) = \mathbb{E}_D\Big[J(f(\mathbf{x}; \boldsymbol{\theta}), y)\Big]$ assuming that $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ are IID samples from $P(\mathbf{x}, y)$. Since the nonlinearity of an MLP makes the loss function to become non-convex, it's trained using iterative, gradient-based optimizers. Most neural networks, and in particular the models that will be used in this work, are trained using maximum likelihood: the loss function is the negative log-likelihood [1] [6]. That is, we will compute the *cross-entropy*[2] between the training data and the model distribution:

$$J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim D} \log_{P(\mathbf{x}, y)}(y|\mathbf{x}) \tag{1.2}$$

In a k-class classification model with a cross-entropy loss function, the empirical risk to minimize is $\hat{R}_J(f) = -\frac{1}{n}\sum_{i=1}^n \sum_{j=1}^k \mathbf{y}_{ij} \log f_j(\mathbf{x}_i; \boldsymbol{\theta})$, where $\mathbf{y}_{ij}$ is the j-th element of the one-hot encoded label of $\mathbf{x}_i$ such that $\mathbf{1}^T \mathbf{y}_i = 1$, $\forall i$, and f_j is the j-th element of the softmax output layer of f [27].
Typically, the minimization of $J(\boldsymbol{\theta})$ is achieved using a gradient descent algorithm. Large training datasets are good for generalization, but since $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \frac{1}{n} \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}; \boldsymbol{\theta}), y)$ has a computational complexity of $\mathcal{O}(n)$ for each step [6], this approach is prohibitive for n in a scale of millions. Optimization algorithms that use the whole dataset are called **batch**[3] gradient methods. When only a single sample is used each step, the method is called **stochastic**. The **Minibatch Stochastic Gradient Descent** (MSGD) algorithm (and its variants) is preferred, since the gradient direction in SGD oscillates because of the additional noise added by random sampling [21]. The MSGD algorithm makes and estimation of the gradient selecting randomly a *minibatch* $b = \{(\mathbf{x}, y)^{(1)}, \cdots, (\mathbf{x}, y)^{(m)}\}$, $m < n$ such as:

$$\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^m l(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}) \tag{1.3}$$

Each step of the MSGD algorithm, the parameters $\boldsymbol{\theta}$ are updated given a learning rate $\eta > 0$: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \mathbf{g}$. A constant learning rate during training may lead to a zone where the gradient estimate "jumps around" a local minimum, although if the network is complex enough to represent the underlying function, a constant learning rate usually works well in practice [20]. In some implementations, the learning rate is updated during training, an example could be multiplying by a factor $0 < \gamma < 1$ each T number of steps[4]. Here, we encounter some challenges [17]:

- A small η leads to slow convergence of vanilla MSGD, while if it's too large the loss function will fluctuate around the minimum or diverge.

[1] Empirical risk minimization is equivalent to Maximum Likelihood Estimation when the risk is defined as the negative of the likelihood. Note that ERM does not limit itself to that particular class of risk functions.
[2] This is valid for a loss consisting of a negative log-likelihood, not only for the negative log-likelihodd of a softmax distribution.
[3] The term *batch* may also refer to a subset of the training set. The size of a minibatch is usually called *batch size*.
[4] This is actually what the *StepLR* class does in the popular Python package torch.optim.

- A learning rate scheduler needs to be defined at the beginning and it's not adapted to dataset's characteristics.

- Since $J(\boldsymbol{\theta})$ is non-convex (recall that affine transformations followed by an activation function such as ReLU is non-covex). A key challenge is avoiding getting trapped in suboptimal local minima. Also, according to [3], the difficulty comes from saddle points present in high-dimensional non-convex optimization that are hard for SGD to scape, since the gradient is close to zero in all dimensions.

There are several gradient-based variants that try to tackle these challenges. For example, a momentum term can be used to help accelerate MSGD and dampen oscillations [15]. Momentum stores the gradient of the past step to adjust the new direction:

$$\begin{aligned} \mathbf{v}_t &= \alpha \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \qquad \eta > 0,\ \alpha \in [0,1] \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \mathbf{v}_t \end{aligned} \tag{1.4}$$

The momentum term increases the updates for dimensions whose gradients point in the same direction, allowing the algorithm to build momentum and move more efficiently towards the minimum. Conversely, it reduces updates for dimensions whose gradients change direction, dampening out oscillations and preventing the algorithm from getting stuck in local minima [17].

Another variant is **Nesterov accelerated gradient** [24], which is an extension of Gradient Descent with momentum where the computation of the gradient is performed using projected parameters and not the actual values:

$$\begin{aligned} \mathbf{v}_t &= \alpha \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} - \alpha \mathbf{v}_{t-1}) \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \mathbf{v}_t \end{aligned} \tag{1.5}$$

Here, similar to GD with momentum, $\mathbf{v}_t$ is the direction and speed at which the parameters should be tweaked and α determines how quickly the previous gradients will decay. Nesterov Accelerated Gradient tries to tackle the exploding gradient case: where the velocity added to the parameters gives an unwanted high loss. In this case, if the velocity update lead to a bad loss, the gradients will redirect the update back to $\boldsymbol{\theta}$.

[PONER IMAGEN QUE COMPARE MOMENTUM CON NAG]

Still, with the previous algorithms the hyperparameters η and α need to be fixed or changed during training a heuristic way. With the **Adaptative Gradient Algorithm** (**Adagrad**), the learning rate change depending on the parameters: it performs larger updates for infrequent parameters and smaller updates for frequent parameters. That's it, each parameter is updated with a different learning rate each step. We set $g_{t,i}$ the gradient estimate w.r.t the parameter θ_i at step t. The Adagrad update rule is:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} g_{t,i} \tag{1.6}$$

Where $G_{t,i}$ is the sum of squares of the gradients w.r.t θ_i up to step t, and $\epsilon > 0$ is the decimal smallest number in the machine, in order to avoid division by zero. With Adagrad, the selection of η is irrelevant since the term will be scaled by the root. Most implementations just fix it at 0.01. The problem with this algorithm is that the accumulated sums will increase during training and eventually the learning rate will be close to zero, therefore the parameters are not updated at a certain point. In order to fix this, **Adadelta** restricts the number of accumulated gradients to some fixed size w [25]. Since storing w previous squared gradients is inefficient, it was proposed to compute an exponentially decaying average of the squared gradients, given a decay constant $\rho \in (0,1)$, similar to that used in Momentum. So, let $\mathbf{g}_t$ be the estimate of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$, $\mathbf{v}$ and $\mathbf{u}$ the square average and the accumulated variable resp., both initialized at $\mathbf{0}$ and $\lambda > 0$ the weight decay, the Adadelta algorithm is:

$$
\begin{aligned}
\mathbf{g}_t &\leftarrow \mathbf{g_t} + \lambda \boldsymbol{\theta}_{t-1} \\
\mathbf{v}_t &\leftarrow \mathbf{v}_{t-1}\rho + \mathbf{g}_t^2(1-\rho) \\
\Delta \mathbf{x}_t &\leftarrow \frac{\sqrt{\mathbf{u}_{t-1}+\epsilon}}{\sqrt{\mathbf{v}_t+\epsilon}}\mathbf{g}_t \\
\mathbf{u}_t &\leftarrow \mathbf{u}_{t-1}\rho + \Delta\mathbf{x}_t^2(1-\rho) \\
\boldsymbol{\theta}_t &\leftarrow \boldsymbol{\theta}_{t-1} - \gamma\Delta\mathbf{x}_t
\end{aligned}
\tag{1.7}
$$

Where the operations between vectors like the square is elementwise, i.e. $\mathbf{g}^2 = \mathbf{g} \odot \mathbf{g}$.
Another first-order gradient-base optimization algorithm for stochastic objective functions that is computationally efficient and compares favorably to other adaptive learning-method algorithms is **Adaptive Moment Estimation** (**Adam**) [8]. In addition to storing an exponentially decaying average of squared gradients $\mathbf{v}_t$ like Adadelta, this algorithm also keeps an exponentially decaying average of past gradients $\mathbf{m}_t$, similar to momentum. Let $\beta_1, \beta_2 \in [0, 1)$ be the exponential decay rates for the moment estimates:

$$
\begin{aligned}
\mathbf{g}_t &\leftarrow \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}) \\
\mathbf{m}_t &\leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \quad \text{(Biased first moment estimate)} \\
\mathbf{v}_t &\leftarrow \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \quad \text{(Biased second raw moment estimate)} \\
\hat{\mathbf{m}}_t &\leftarrow \mathbf{m}_t/(1-\beta_1^t) \quad \text{(Bias-corrected first moment estimate)} \\
\hat{\mathbf{v}}_t &\leftarrow \mathbf{v}_t/(1-\beta_2^t) \quad \text{(Bias-corrected second raw moment estimate)} \\
\boldsymbol{\theta}_t &\leftarrow \boldsymbol{\theta}_{t-1} - \alpha\hat{\mathbf{m}}_t/\left(\sqrt{\hat{\mathbf{v}}_t}+\epsilon\right)
\end{aligned}
\tag{1.8}
$$

According to [8], good default settings (in their machine learning experiments) are $\alpha = 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

In order to use this gradient-based algorithms we need to compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is done using the **Backpropagation** algorithm [18], which leverages the composite structure of the neural network to efficiently compute the gradient [2].

1.1.2 Convolutional Neural Networks

When the input data is an image, a better approach for learning about $p(y|\mathbf{x})$ is to use Convolutional Neural Networks (CNN), since MLP tend to struggle with the computational complexity required to compute image data and the width required to train this kind of data may lead to overfitting [14]. CNNs are comprised of three types of layers: convolutional layers, pooling layers and fully-connected layers (which are essentially a MLP).
The convolutional layer use learnable kernels:

1.2 Other ML models

Chapter 2

Federated Learning

2.1 Introduction

In the rapidly evolving landscape of artificial intelligence and machine learning, Federated Learning (FL) has emerged as a paradigm that addresses key challenges related to privacy, data security, and decentralized computing. Federated Learning represents a novel approach to model training, allowing machine learning models to be trained collaboratively across multiple decentralized devices or servers without exchanging raw data [12].

Unlike traditional centralized approaches, where data is collected and processed in a central server, FL enables training on local devices following a scheme of decentralized model training. This decentralization ensures that data remains on the device, granting a certain degree of privacy. In a centralized setting, the trained model is updated based on the complete dataset, which is stored in a unique server. In the federated setting, data is distributed across local devices (parties) and training happens locally. This can be problematic, since the source of the data differ, the data can differ in various ways: unbalanced datasets, different distributions, etc. This is one of the main challenges of FL: **non-IID data**, since this will negatively affect the performance of the model [11], [28], [10].

Horizontal Federated Learning (HFL) and Vertical Federated learning (VFL) are two variations of the federated learning paradigm that differ in how they distribute and collaborate on data.

- **HFL:** Each party has a portion of the overall dataset, each party holds a different subset of samples but for the same features.
- **VFL:** The data is vertically partitioned, each party has different features for the same set of samples.

[PONER TABLAS QUE ILUSTREN HFL Y VFL]

HFL and VFL are not mutually exclusive, in some cases a combination of both schemes may be applied. This work will focus on HFL. Also, we will only study *Cross-Silo Federated Learning (Cross-Silo FL)*, which is a variation of FL that addresses the scenario where data is distributed across different organizations, usually few parties, each maintaining control over its own data. This setting is particularly relevant in industries where different organizations need to collaborate on machine learning task, such as healthcare (hospitals collaborating on medical research), finance (banks collaborating on fraud detection), epidemiological studies (international public health agencies studying disease spread), smart cities (urban planning authorities collaborating on public services optimization), etc. Ensuring interoperability between different silos is a huge challenge, since there needs to be a fixed standard in data format, structures and processing capabilities accross different organizations.

Another flavour of FL is *Cross device FL*, which was the original topology proposed [12]. In this type of setting, the scale of the parties is potentially much bigger (order of millions) since it's aimed to mobile devices, IoT, apps... where the data format and architecture is usually already defined by an organization. For example, this is the kind of FL that Google uses for training Gboard's models [26]. With *Cross device FL*,

it's needed to take into account some technical difficulties like client selection strategy [19], connection overheard, connection dropouts, device heterogeneity, etc. Some of these challenges are shared with *Cross-Silo FL*. The main differences are the scale of parties, the computing resources (mobile devices vs data centers or computers) and the partition of data: *cross device* tends to be HFL while *cross-silo* could be HFL or VFL. Since we are focusing in *cross-silo FL*, it won't be studied the implications of client selection (since we are assuming the parties are few and well-defined), the connection overheard or dropouts. The focus of this work will be mainly the statistical heterogeneity challenge, since this work serves as the final project for obtaining a MSc in Statistics.

We will begin by studying the FedAvg algorithm [12], which is de facto approach for Federated Learning (FL). We will establish notation, examine some of its properties, and explore issues that arise when data is not independently identically distributed (statistical heterogeneity). Following that, various approaches that have been proposed to address this problem will be developed, and the performance of each will be analyzed across different training architectures.

2.2 Objective function

Let $D = \{(\mathbf{x}, y)\}$ be the global dataset[1] and $D^i \subset D$ the i-th party's local dataset, $i = 1, ..., N$. Let ω_g^t and ω_i^t be the global model and the local model of the i-th party in round $t \in \{1, ..., T\}$, respectively. Since we are working in a *Cross-silo FL* setting, the main differences between a federated optimization scheme and a distributed one would be the unbalanced data and non-IID data.
In a general FL optimization framework, we want to minimize the objective function:

$$F(\omega) = \mathbb{E}_{i\sim D}\Big[F_i(\omega_g)\Big], \qquad F_i(\omega_g) = \mathbb{E}_{z\sim D^i}\Big[l_i(\omega_g, z)\Big] \tag{2.1}$$

Here, $l_i(\omega_g, z)$ represents the local loss function of client i (with local dataset D^i). As stated in [22], the objective function in 2.1 can take the form of an empirical risk minimization objective function:

$$F(\omega_g) = \sum_{i=1}^{N} \alpha_i F_i(\omega_g) \text{ where } F_i(\omega_g) = \frac{1}{|D^i|} \sum_{z\in D^i} l_i(\omega_g, z) \text{ and } \sum_{i=1}^{N} \alpha_i = 1 \tag{2.2}$$

If $\alpha_i = \frac{|D^i|}{\sum_{i=1}^{N} |D^i|}$, the objective function in 2.2 would be the empirical risk minimization objective function of $D = \cup_i D^i$. We recall that 2.1 is an optimization problem that could be solved using Stochastic Descent: $\omega_g^{t+1} = \omega_g^t - \eta_t \nabla F(\omega_g^t)$ where η_t is the learning rate at time t. In practice, we will use an unbiased estimator of the gradient of the local loss $g_i(\omega_g^t)$ such that $\mathbb{E}_{z\sim D^i}\Big[g_i(\omega_g^t)\Big] = \nabla F_i(\omega_g^t)$ using Stochastic Gradient Descent (SGD).

2.3 Statistical heterogeneity

In a real-world scenario, the data distributed among parties is Non-IID, the distribution from each party differs from the global distribution. Each client has its own local optima, which may be very different from the actual global optima. Under a IID setting, the global optima is *close* to the local optima, but under a Non-IID settig, there is a *drift* in the local updates, especially if the number of local epochs changes between parties.

In order to simulate a Non-IID setting, it has been chosen the same strategy as [10], the datasets will be partitioned into multiple subsets. Let the local data distribution be $P(x_i, y_i) = P(x_i|y_i)P(y_i) = P(y_i|x_i)P(x_i)$. From a distribution perspective, the following Non-IID cases summarized in [7] are considered:

[1]Here, the global dataset is the union of the different local datasets, $D = \cup_{i=1}^{N} D^i$. In practical cases, there is no such dataset in order to ensure data privacy. However, we will consider it to conduct a performance study of the various algorithms.

- **Label distribution skew:** Different $P(y_i)$ across parties.
- **Feature distribution skew:** Different $P(x_i)$ across parties.
- **Same label but different features:** Different $P(x_i|y_i)$ across parties.
- **Same features but different labels:** Different $P(y_i|x_i)$ across parties.
- **Quantity skew:** Although the parties have the same $P(x_i, y_i)$, $|D^i|$ differs.

For the **label distribution skew**, it will be used two settings. The first one, is a quantity-based label imbalance, where each party possesses data samples corresponding to a fixed number of labels. An extreme case is where each party only has data from a unique label.

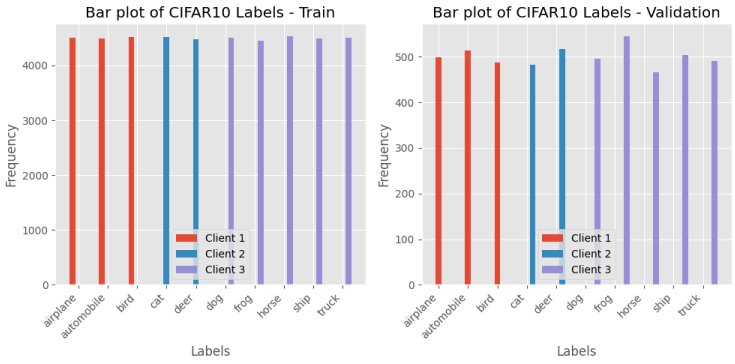

Figure 2.1: Example of label distribution skew with 3 clients, each one holding data from different labels.

The second setting is a distribution-based label imbalance, where the Dirichlet distribution $Dir(\boldsymbol{\beta})$ is used to assign different proportion of each label across parties. Given N parties, the probability density function from the Dirichlet distribution is defined as:

$$f(\mathbf{x}, \boldsymbol{\beta}) = \frac{1}{\boldsymbol{\beta}} \prod_{i=1}^{N} x_i^{\beta_i - 1}, \quad \sum_{i=1}^{N} x_i = 1, \; x_i \in [0, 1], \; \beta_i > 0$$

where the normalizing constant $B(\boldsymbol{\beta})$ is the multivariate beta function.

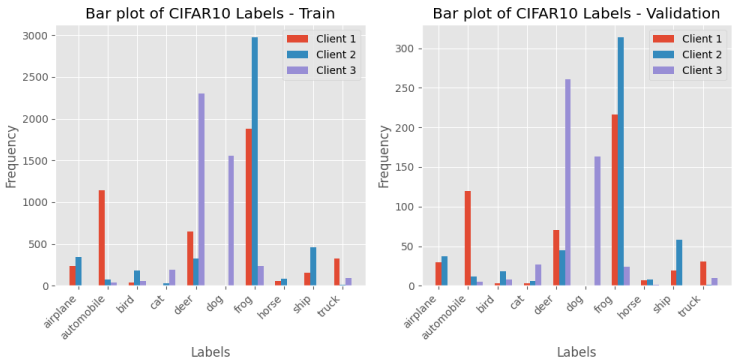

Figure 2.2: Example of label distribution skew with 3 clients using the Dirichlet distribution with $\boldsymbol{\beta} = (0.5, 0.5, 0.5)$.

For the **feature distribution skew**, it will be used a noise-based feature imabalance, where different levels of Gaussian noise will be added for each party. Given $\sigma > 0$, for party i, a noise $N(0, \sigma\frac{i}{N})$ will be added to its local features. Increasing σ will increase the dissimilarity among parties. Note that although $P(x_i)$ differs, $P(y_i|x_i)$ is the same across parties.

Figure 2.3: Example of feature distribution skew with $\sigma = 0.3$

It will be not considered the case where $P(y_i|x_i)$ differs, because then a distributed learning setting would make no sense (same feature should be classified as a different label across parties). Different $P(x_i|y_i)$ will neither be considered since we are focusing in a Horizontal FL setting. For the quantity skew, it will be used $Dir(\boldsymbol{\beta})$ to assign different proportion of the whole dataset to each party.

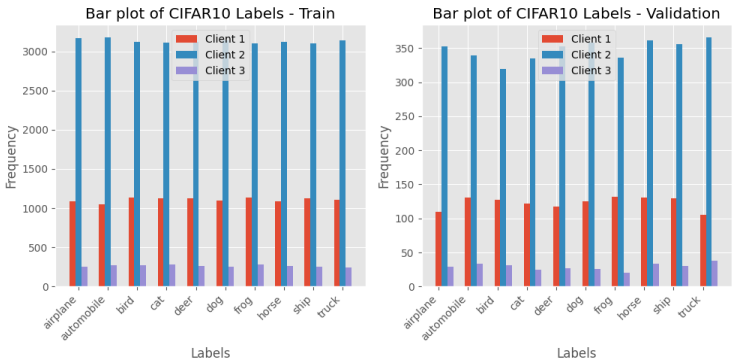

Figure 2.4: Example of quantity skew with 3 clients using the Dirichlet distribution with $\boldsymbol{\beta} = (0.5, 0.5, 0.5)$.

For the rest of the chapter, these Non-IID Data will be applied in order to see the effect in the model's accuracy. The model used is a very simple[2] CNN described in Figure 2.5

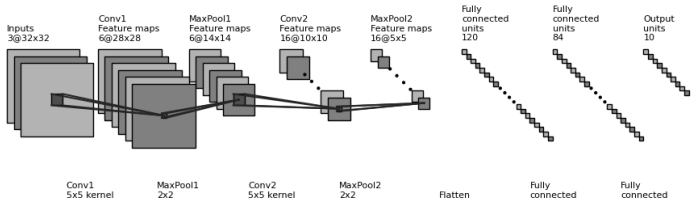

Figure 2.5: CNN used for all the following experiments.

2.4 FedAvg

In [12], it was introduced the **FederatedAveraging** algorithm (FedAvg).

[2]The aim of these experiments is to check the accuracy degradation between different Non-IID settings. It is not necessary to train the best model in order to achieve a high test accuracy, since the IID settings will serve as a baseline to compare between Non-IID cases. Note that the number of possible experiments combining all these settings is huge, and the whole project has been made with a low-spec laptop without a GPU.

Algorithm 1 FedAvg

Input: local datasets $D^i \; \forall i \in \{1, \dots, N\}$, number of parties N, number of communication rounds T, number of local epochs E, learning rate η for SGD, local mini-batch size B.
Output: global model ω_g^T.

procedure SERVER EXECUTION
  Initialize ω_g^0
  for round $t = 1, \dots, T$ **do**
    S_t (Selection of clients)
    for client $k \in S_t$ **in parallel do**
      $\omega_k^{t+1} \leftarrow ClientUpdate(k, \omega_g^t)$
    $\omega_g^{t+1} \leftarrow \sum_{k \in S_t} \frac{|D^k|}{\sum_{k \in S_t} |D^k|} \omega_k^{t+1}$
procedure $ClientUpdate(k, \omega_g^t)$
  $\omega_k^t \leftarrow \omega_g^t$
  $\mathcal{B} \leftarrow$ Batches of D^k of size B
  for local epoch $i = 1, \dots, E$ **do**
    for batch $\mathbf{b} \in \mathcal{B}$ **do**
      $\omega_k^t \leftarrow \omega_k^t - \eta \nabla l(\omega_k^t; \mathbf{b})$
  return ω_k^t to the server.

As we see, *FedAvg* took into account a client selection strategy. Since we are focusing in a framework where the clients are limited, we will not consider any subset of clients each round: all the clients will participate.
The algorithm is quite simple, the server is averaging the local models' weights, with a constant parameter determining the contribution of each client given the size of the local dataset.

[PONER AQUI EXPERIMENTOS HECHOS CON FEDAVG]

Each client performs multiple local epochs, which reduces the communication rounds. However, these local updates can lead to a worse performance since the local objective functions differ from each other. This makes the algorithm not robust against non-IID data. There are several works that try to tackle this problem and a lot of FedAvg variants have been proposed in order to mitigate the divergence between the local updates. We will review some of the more relevant of them, starting with *FedProx*.

2.5 FedProx

In [11], a generalization of *FedAvg* was proposed where the local function to minimize $F_k(\omega_k)$ was modified adding a proximal term:

$$\min_{\omega_k} h_k(\omega_k, \omega_g^t) = F_k(\omega_k) + \frac{\mu}{2} \|\omega_k - \omega_g^t\|^2, \qquad \mu \geq 0 \tag{2.3}$$

The basic idea, is to restrict the local updates to be closer to the actual global model. In the original proposal, it was stated that a constant number of local epochs per round each time could be not feasible because of the device / system heterogeneity. Since this work is not focused on system heterogeneity, it will not be mentioned the theoretical convergence results.

FedAvg is a particular case when $\mu = 0$, a fixed number of local epochs E is set and all the local solvers are SGD. As μ increases, the function becomes more restrictive meaning that it will take longer to converge. This algorithm doesn't restrict us to any local solver, so it's not necessary to compute the estimation of the gradient using SGD. When the data across clients is IID, a positive μ could decelerate the convergence, some heuristics could be applied such as decreasing μ as long as the loss functions continues to decrease.

Following the experiments in [11], the possible values of μ that will be used in this work are selected from the finite set $\{0.001, 0.01, 0.1, 1\}$

2.6 FedNova

This algorithm was first introduced in [23], in order to study its main contributions we are changing the original notation of *FedAvg*. We have seen that the client updates its model at round t starting with a shared, global model ω_g^t. Then, it applies SGD with its local data for a fixed number of epochs E. We recall that, for each epoch, and for each batch $\mathbf{b} = \{\mathbf{x}, y\} \in \mathcal{B} \subset D^i$, the i-th client performs $\omega_i^t = \omega_i^t - \eta \nabla l_i(\omega_i^t, \mathbf{b})$. After the round, the client will have a locally updated model ω_i^{t+1}. After all the clients have updated their models, it aggregates them following the formula: $\omega_g^{t+1} = \sum_{k \in S_t} \frac{|D^k|}{\sum_{k \in S_t} |D^k|} w_k^{t+1}$. We can now consider the difference between the global client's starting model ω_g^t and its locally updated model ω_i^{t+1}, we will denote $\Delta\omega_i^t \coloneqq \omega_g^t - \omega_i^{t+1}$, the difference between the global model at time t and the local model of client i at time $t+1$. The server would then collect $\Delta\omega_k^t \ \forall k \in \{1, \ldots, N\}$ and update the global model. For example, in *FedAvg*: $\omega_g^{t+1} = \omega_g^t + \sum_{k \in S_t} \alpha_k \Delta\omega_k^t = \omega_g^t - \sum_{k \in S_t} \alpha_k \eta \sum_{j=1}^{\tau_k} g_k(\omega_k^{t,j})$ where $\alpha_k = \frac{|D^k|}{\sum_{k \in S_t} |D^k|}$, $\tau_k = \lfloor \frac{E_k |D^k|}{B} \rfloor$ with B being the mini-batch size and E_k the local number of epochs of client k, $\omega_k^{t,j}$ is the client k's model after the j-th SGD update at time t, η is the learning rate (same for all clients) and g_k is the stochastic gradient over a mini-batch $\mathbf{b} \in \mathcal{B}$.
As stated in [23], different E_k across clients and rounds means that $FedAvg$ algorithm converges to a surrogate objective instead of $F(\omega)$ from (2.1). This algorithm considers that different parties compute a different number of local steps (τ_k) because of computation constraints or different sizes of local datasets. When $\tau_k > \tau_{k'}$, client k will have a major influence over the global update rule compared to client k'. That's why the authors proposed to normalize and scale the local updates according to the number of local steps.

[PONER IMAGEN QUE SE VEA LAS DIRECCIONES DE LOS GRADIENTES CUANDO NO SE NORMALIZA Y CUANDO SE NORMALIZA]

Algorithm 2 FedNova

Input: local datasets $D^i \; \forall i \in \{1, \ldots, N\}$, number of parties N, number of communication rounds T, number of local epochs E, learning rate η, local mini-batch size B.
Output: global model ω_g^T.

procedure SERVER EXECUTION
  Initialize ω_g^0
  for round $t = 1, \ldots, T$ **do**
    S_t (Selection of clients)
    for client $k \in S_t$ **in parallel do**
      $\Delta\omega_k^t, \tau_k \leftarrow ClientUpdate(k, \omega_g^t)$
    $n \leftarrow \sum_{k \in S_t} |D^k|$
    $\omega_g^{t+1} \leftarrow \omega_g^t - \eta \frac{\sum_{k \in S_t} |D^k| \tau_k}{n} \sum_{k \in S_t} \frac{|D^k| \Delta\omega_k^t}{\tau_k n}$
procedure $ClientUpdate(k, \omega_g^t)$
  $\omega_k^t \leftarrow \omega_g^t$
  $\tau_k \leftarrow 0$
  $\mathcal{B} \leftarrow$ Batches of D^k of size B
  for local epoch $i = 1, \ldots, E$ **do**
    for batch $\mathbf{b} \in \mathcal{B}$ **do**
      $\omega_k^t \leftarrow \omega_k^t - \eta \nabla l(\omega_k^t; \mathbf{b})$
      $\tau_k \leftarrow \tau_k + 1$
  $\Delta\omega_k^t \leftarrow \omega_g^t - \omega_k^t$
  return $\Delta\omega_k^t$, τ_k to the server.

2.7 SCAFFOLD

With data heterogeneity, *FedAvg* suffers from client-drift, since each local updates it's minimizing its own local objective. SCAFFOLD tries to solve this introducing control variables for the server c and the clients $c_k \; \forall k \in \{1, \ldots, N\}$.

Algorithm 3 SCAFFOLD

Input: local datasets $D^i \; \forall i \in \{1, \ldots, N\}$, number of parties N, number of communication rounds T, number of local epochs E, learning rate η, local mini-batch size B.
Output: global model ω_g^T.

1: **procedure** SERVER EXECUTION
2: Initialize ω_g^0
3: $c^t \leftarrow \mathbf{0}$, $c_k \leftarrow \mathbf{0}$
4: **for** round $t = 1, \ldots, T$ **do**
5: S_t (Selection of clients)
6: **for** client $k \in S_t$ **in parallel do**
7: $\Delta\omega_k^t, \Delta c_k \leftarrow ClientUpdate(k, \omega_g^t, c^t)$
8: $n \leftarrow \sum_{k \in S_t} |D^k|$
9: $\omega_g^{t+1} \leftarrow \omega_g^t - \eta \sum_{k \in S_t} \frac{|D^k|}{n} \Delta\omega_k^t$
10: $c^{t+1} \leftarrow c^t + \frac{1}{N} \sum_{k \in S_t} \Delta c_k$
11: **procedure** $ClientUpdate(k, \omega_g^t, c_k)$
12: $\omega_k^t \leftarrow \omega_g^t$
13: $\tau_k \leftarrow 0$
14: $\mathcal{B} \leftarrow$ Batches of D^k of size B
15: **for** local epoch $i = 1, \ldots, E$ **do**
16: **for** batch $\mathbf{b} \in \mathcal{B}$ **do**
17: $\omega_k^t \leftarrow \omega_k^t - \eta(\nabla l(\omega_k^t; \mathbf{b}) - c_k + c)$
18: $\tau_k \leftarrow \tau_k + 1$
19: $\Delta\omega_k^t \leftarrow \omega_g^t - \omega_k^t$
20: $c_k^+ \leftarrow$ (i) $\nabla l(\omega_g^t)$ or (ii) $c_k - c + \frac{1}{\tau_k \eta}(\omega_g^t - \omega_k^t)$
21: $\Delta c \leftarrow c_k^+ - c_k$
22: $c_k = c_i^+$
23: return $\Delta\omega_k^t$, Δc to the server.

From the SCAFFOLD's algorithm, it can be noted that the clients maintain the state of c_k and the server maintain c, which are initialized at 0. In comparison with the previous algorithms, here the local update formula is: $\omega_k^t = \omega_k^t - \eta(\nabla l(\omega_k^t, \mathbf{b}) - c_k + c)$. Intuitively, the local gradient $\nabla l(\omega_k^t, \mathbf{b})$ moves towards the local optimum for client k, the correction $c - c_k$ ensures that the update change its direction towards the global optimum. As we can see, in line 20, there are two options for computing c_i^+: $\nabla l(\omega_g^t)$ or $c_k - c + \frac{1}{\tau_k \eta}(\omega_g^t - \omega_k^t)$.

Chapter 3

Multi-Party Computation

3.1 Basic concepts

Multi-Party Computation (MPC) is a cryptographic framework where n parties want to compute a function using their data but without revealing its own data to any other party. More precisely, we consider P_i, with its input x_i, $i = 1, \cdots, n$. An MPC protocol aims to compute a function $z = f(x_1, \cdots, x_n)$ and after execution, the party P_i only knows x_i (its own input data) and z (the output).
It could be the case that some parties decide to share its input with other parties, in order to break the privacy of all parties. In this theoretical framework, we say that an *adversary* corrupts a subset of the parties. The more parties are corrupted, the easier to break the privacy of the protocol. In order to classify the security of certain MPC protocol, it is said that the protocol is secure as long as the adversary corrupts at most t parties, usually $t < n/2$ or $t < n/3$. In this chapter, we will focus on *linear secret-sharing schemes*, where parties hold distributed versions of intermediate results of the computations.
A secret-sharing scheme is a method to split a secret into shares in such way that it's not possible to learn anything from the shares as long as each party only holds a maximum of these. Surpass the maximum, and the shares can reveal the secrets.

Se consider a finite field $\mathbb{F}$, let $s \in \mathbb{F}$ and denote $[n]$ as the index set $\{1, \cdots, n\}$, A secret-sharing scheme of n parties with threshold t provides method for computing a set of values (shares) $[[s]] = (s_1, \cdots, s_n) \in \mathbb{F}^n$ such that, for any set $A \subseteq [n]$, $|A| \leq t$, the set of shares $\{s_i\}_{i \in A}$ does not leak anything about s. And for any set $B \subseteq [n]$, $|B| > t$, s can be reconstructed using the set $\{s_i\}_{i \in B}$ [4].

Since we are dealing with *linear* secret-sharing schemes, it holds that if $[[a]] = (a_1, \cdots, a_n)$ and $[[b]] = (b_1, \cdots, b_n)$ then $[[a \pm b]] = (a_1 \pm b_1, \cdots, a_n \pm b_n)$. Given a linear secret-sharing scheme $[[\cdot]]$, it could be possible to define the product of two shared values $[[a \cdot b]]$, but it would possibly require some interaction between parties.

We present the widely used **Shamir secret-sharing scheme**

3.2 Protocol Examples

3.3 Private Set Intersection

One widely used application of MPC is **Private Set Intersection** (**PSI**), which allows to compute the common elements between datasets' parties (for simplicity, we will focus on two parties). Given a party P_i with a dataset D^i, $i = 1, 2$, we want to compute $D^1(j) \cap D^2(j)$ being $D^i(j)$ the j-th column (set) from the i-th party's dataset. For example, if D consist in a single list of phone numbers, $D^1 \cap D^2$ would be the common phone contacts between party 1 and party 2. Another example could be that the datasets D^1 and D^2 have the fields:

- $D^1(1) \coloneqq$ National Identity Number
- $D^1(2) \coloneqq$ Name
- $D^1(3) \coloneqq$ Surname
- $D^1(4) \coloneqq$ Annual Salary ($)

- $D^2(1) \coloneqq$ National Identity Number
- $D^2(2) \coloneqq$ Age
- $D^2(3) \coloneqq$ Genre

If we would like to know the mean annual salary by genre, first we would need to *merge* both datasets, mapping the National Identity Number. With PSI, we could compute $D^1(1) \cap D^2(1)$, this gives us the NIN of the people with complete data, i.e. the people whose personal data are in both datasets, so it is known both the annual salary and the genre. With that intersection, we could get two subsets from D^1 and D^2 and then use another MPC protocol (in this case, we are interested in the operation *GroupBy*).

As an informal introduction of the problem, we define two sets: $X = \{x_1, \cdots, x_n\}$ and $Y = \{y_1, \cdots, y_m\}$, $n, m \in \mathbb{N}$

Chapter 4

Advanced strategies for Privacy-Enhancing Machine Learning

Bibliography

[1] Ovidiu Calin. *Deep Learning Architectures: A Mathematical Approach*. Springer Series in the Data Sciences. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-36720-6 978-3-030-36721-3. DOI: 10.1007/978-3-030-36721-3. (Visited on 05/23/2024).

[2] Saeed Damadi, Golnaz Moharrer, and Mostafa Cham. *The Backpropagation Algorithm for a Math Student*. May 2023. arXiv: 2301.09977 [cs, math]. (Visited on 05/26/2024).

[3] Yann Dauphin et al. *Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization*. June 2014. arXiv: 1406.2572 [cs, math, stat]. (Visited on 05/24/2024).

[4] Daniel Escudero. *An Introduction to Secret-Sharing-Based Secure Multiparty Computation*. 2022. (Visited on 07/01/2024).

[5] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, June 2011, pp. 315–323. (Visited on 05/23/2024).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 978-0-262-03561-3.

[7] Peter Kairouz et al. *Advances and Open Problems in Federated Learning*. Mar. 2021. arXiv: 1912.04977 [cs, stat]. (Visited on 05/15/2024).

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 2017. arXiv: 1412.6980 [cs]. (Visited on 05/26/2024).

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. (Visited on 05/23/2024).

[10] Qinbin Li et al. *Federated Learning on Non-IID Data Silos: An Experimental Study*. Oct. 2021. arXiv: 2102.02079 [cs]. (Visited on 01/12/2024).

[11] Tian Li et al. *Federated Optimization in Heterogeneous Networks*. Apr. 2020. arXiv: 1812.06127 [cs, stat]. (Visited on 12/16/2023).

[12] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. Jan. 2023. arXiv: 1602.05629 [cs]. (Visited on 01/12/2024).

[13] Kevin P. Murphy. *Probabilistic Machine Learning: An Introduction*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts London, England: The MIT Press, 2022. ISBN: 978-0-262-36930-5 978-0-262-04682-4.

[14] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2015. arXiv: 1511.08458 [cs]. (Visited on 05/26/2024).

[15] Ning Qian. "On the Momentum Term in Gradient Descent Learning Algorithms". In: *Neural Networks: The Official Journal of the International Neural Network Society* 12.1 (Jan. 1999), pp. 145–151. ISSN: 1879-2782. DOI: 10.1016/s0893-6080(98)00116-6.

[16] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. (Visited on 05/22/2024).

[17] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. June 2017. arXiv: 1609.04747 [cs]. (Visited on 05/24/2024).

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. (Visited on 05/26/2024).

[19] Carl Smestad and Jingyue Li. "A Systematic Literature Review on Client Selection in Federated Learning". In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. June 2023, pp. 2–11. DOI: 10.1145/3593434.3593438. arXiv: 2306.04862 [cs]. (Visited on 05/11/2024).

[20] Ruoyu Sun. *Optimization for Deep Learning: Theory and Algorithms*. Dec. 2019. arXiv: 1912.08957 [cs, math, stat]. (Visited on 05/24/2024).

[21] Shiliang Sun et al. *A Survey of Optimization Methods from a Machine Learning Perspective*. Oct. 2019. arXiv: 1906.06821 [cs, math, stat]. (Visited on 05/24/2024).

[22] Jianyu Wang et al. *A Field Guide to Federated Optimization*. July 2021. arXiv: 2107.06917 [cs]. (Visited on 05/14/2024).

[23] Jianyu Wang et al. *Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization*. July 2020. arXiv: 2007.07481 [cs, stat]. (Visited on 05/15/2024).

[24] Nesterov Y. "A Method of Solving a Convex Programming Problem with Convergence Rate O(1/K**2)". In: *Doklady Akademii Nauk SSSR* 269.3 (1983), p. 543. (Visited on 05/25/2024).

[25] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. Dec. 2012. arXiv: 1212.5701 [cs]. (Visited on 05/26/2024).

[26] Yuanbo Zhang et al. *Private Federated Learning in Gboard*. June 2023. arXiv: 2306.14793 [cs]. (Visited on 05/11/2024).

[27] Zhilu Zhang and Mert R. Sabuncu. *Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels*. Nov. 2018. arXiv: 1805.07836 [cs, stat]. (Visited on 05/24/2024).

[28] Yue Zhao et al. "Federated Learning with Non-IID Data". In: (2018). DOI: 10.48550/arXiv.1806.00582. arXiv: 1806.00582 [cs, stat]. (Visited on 01/12/2024).