FREIE UNIVERSITÄT BERLIN

**Bachelorarbeit**

# Implementing a simple deep learning neural network in C++ for federate learning: creating synthetic data for simple use case and implementing NN to solve some classification task


Wiktoria Krawczyk


| | |
|---|---|
| Gutachter(in): | Univ.-Prof. Gerhard Wunder |
| Semester: | Sommersemester 2023 |
| Verfasser: | Vorname Nachname |
| Matrikel-Nr.: | 5461523 |
| Adresse: | Kreuzgraben 4, 13156 Berlin |
| Email: | kraw00@zedat.fu-berlin.de |
| Telefon: | 0048666830836 |
| Studienfach: | Bachelor Informatik |
| **Abgabetermin:** | **25. August 2023** |

# Abstract

This thesis focuses on the application of machine learning techniques, specifically federated neural networks, for low resource devices. The aim is to explore and understand the fundamentals of artificial intelligence and classification algorithms as a foundation for this research. The thesis begins by providing an overview of artificial intelligence and its key concepts. Three primary algorithms are studied in detail: logistic regression, neural networks, and xgboost. To facilitate the implementation and comparison of logistic regression and neural networks, the Shark library is employed. Additionally, the xgboost library is utilized to evaluate and analyze the performance of the xgboost algorithm. The implementation process is carried out, and the obtained results are compared and discussed. Furthermore, the thesis examines federated neural networks, considering their suitability and effectiveness for deployment in resource-constrained devices. The conceptual understanding gained earlier is translated into practical application through the implementation of a federated neural network using the C++ programming language. The design choices and implementation are examined, providing valuable insights into the feasibility and performance of federated neural networks for low resource devices.

In summary, this thesis contributes to the existing body of knowledge by investigating the application of machine learning techniques, particularly federated neural networks, for low resource devices. Through an exploration of fundamental concepts, analysis of classification algorithms, implementation using relevant libraries, and comparison of results, the thesis provides valuable insights and understanding in this domain.

# Contents

**Abstract** **2**

**1 Introduction** **4**
1.1 Machine Learning . . . . . 5

**2 Literature Review** **6**

**3 Background** **7**
3.0.1 Cost Function . . . . . 7
3.0.2 Gradient Descent . . . . . 7
3.0.3 Learning Rate . . . . . 7
3.0.4 Activation Function . . . . . 7
3.0.5 Training and Testing . . . . . 8
3.0.6 Regularization . . . . . 8
3.1 Logistic Regression . . . . . 8
3.2 XGBoost . . . . . 9
3.3 Neural Network . . . . . 10
3.3.1 Forward Propagation . . . . . 11
3.3.2 Training . . . . . 11
3.3.3 Backpropagation . . . . . 12

**4 Methodology** **14**
4.1 Neural Network Implementation . . . . . 14
4.2 XGBoost Implementation . . . . . 17
4.3 Logistic Regression Implementation . . . . . 17

**5 Federated Learning** **19**

**6 Results and Analysis** **20**

**7 Conclusion** **21**

**A Appendix Title** **22**

# Chapter 1

# Introduction

The primary focus of this thesis is the implementation and evaluation of machine learning algorithms, with a particular emphasis on the development of a simple deep learning neural network in C++ for federated learning. This research aims to create a robust and efficient implementation of the federated learning model, which allows multiple parties to collaborate and train machine learning models without sharing their private data.

As an illustrative example, we can explore the application of a photo recognition model that learns to identify and categorize various objects such as people, animals, and places. Neural networks are known to excel in scenarios where a vast amount of data is available. In theory, one could collect all the photos from users and train the model on this comprehensive dataset. However, in practice, this approach raises concerns regarding privacy, which is already a significant consideration for many individuals.

To address the privacy concern, an alternative approach known as federated learning can be employed. Federated learning allows the models to be trained locally on each participant's device, thereby eliminating the need for centralized data collection. Instead of sharing raw data, only the model updates are exchanged among the participants. This approach ensures data privacy while facilitating collaborative learning across multiple entities. By training the models on the participants' devices, sensitive data remains within their control, mitigating privacy risks associated with sharing personal information.

In addition to privacy, the limited resources available on devices must also be taken into account. To address these resource limitations, it is advantageous to implement the model using a lower-level programming language, such as C++, rather than relying solely on high-level languages like Python. Lower-level languages provide greater control over system resources and allow for efficient utilization of the device's capabilities, enabling the model to run smoothly on devices with constrained resources.

Before delving into the details of the implemented algorithms, it is essential to provide a comprehensive explanation of machine learning. Machine learning is a field within artificial intelligence (AI) that focuses on developing algorithms and models capable of learning from data and making predictions or decisions without being explicitly programmed. It encompasses a wide range of techniques and approaches, including supervised learning, unsupervised learning, and reinforcement learning.

The initial step in developing a federated neural network involves comparing various machine learning algorithms for data classification and assessing their results. This comparison allows us to identify the most suitable algorithms for the federated learning framework and understand their performance characteristics. Through thorough evaluation and analysis, we can ensure that the chosen algorithms provide accurate and reliable results while preserving data privacy.

Furthermore, this research aims to create synthetic data for a simple use case and implement the neural network to solve a classification task. Synthetic data generation allows us to create datasets with known properties and characteristics, which can be useful for testing and validating machine learning models. By implementing the neural network on the synthetic data, we can evaluate its

performance and assess its ability to accurately classify and make predictions.

Overall, this thesis endeavors to contribute to the field of machine learning by providing a practical implementation of a deep learning neural network in C++ for federated learning.

## 1.1 Machine Learning

According to Montavon (2023), AI-powered systems have found extensive applications in various domains. The field of AI encompasses four distinct categories: thinking humanly, acting humanly, thinking rationally, and acting rationally. These categories represent different conceptualizations of AI. AI is an interdisciplinary field that draws upon disciplines such as philosophy, mathematics, economics, and neuroscience to establish its foundations. Since its inception in the 1940s, AI has evolved and undergone shifts in direction, with a focus on solving general tasks or addressing specific problems of interest.

# Chapter 2

# Literature Review

# Chapter 3

# Background

Before delving into the details of the implemented machine learning algorithms, it is crucial to clarify several common concepts based on Ng, A., & Ma, T. (2023, June 11). CS229 Lecture Notes, Stanford University.

### 3.0.1 Cost Function

A cost function $J(\theta)$ is a mathematical formula used to quantify the error between predicted values and expected values, measuring the performance of a model. The goal is to minimize the cost function to minimize the error.

### 3.0.2 Gradient Descent

To minimize the cost function $J(\theta)$, an algorithm can be employed that starts with an "initial guess" for $\theta$ and iteratively modifies it to reduce $J(\theta)$ until convergence is achieved and a value of $\theta$ that minimizes $J(\theta)$ is obtained. The gradient descent algorithm is specifically utilized for this purpose, where an initial $\theta$ is chosen, and iterative updates are performed.

Furthermore gradient can be described, so that when calculating the gradient of the loss function at the current point and moving in the opposite direction. In a function with multiple variables, the gradient represents a vector pointing towards the direction of the steepest increase. At this point we can introduce the learning rate.

### 3.0.3 Learning Rate

The learning rate is the magnitude of the movement in gradient descent is determined by the value of the slope $\frac{d}{dw}L(f(x;w),y)$, which is further scaled by a learning rate $\eta$. A higher learning rate, indicating a faster pace, implies that the value of $w$ should be adjusted more significantly at each step.

### 3.0.4 Activation Function

Ng, A. lists several activation functions that map $R$ to $R$, such as:

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad \text{(sigmoid)}$$

$$\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \quad \text{(tanh)}$$

$$\sigma(z) = \max(z, \gamma z), \quad \gamma \in (0,1) \quad \text{(leaky ReLU)}$$

$$\sigma(z) = \frac{z^2}{1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)} \quad \text{(GELU)}$$

$$\sigma(z) = \frac{1}{\beta} \log\left(1 + \exp(\beta z)\right), \quad \beta > 0 \quad \text{(Softplus)}$$

These functions, as the name suggests, either activate or deactivate neurons based on their input value. We use them to introduce non-linearity into the model and prevent the neural network from being limited to linear functions over the original input.

### 3.0.5 Training and Testing

The dataset that the model will be used on needs to be divided into training and testing datasets. This division allows for an accurate evaluation of the model's performance. The purpose of this practice is to train the algorithm to fit the data while avoiding overfitting. In other words, the model needs to work well with data beyond the training set. If the testing error is large while the training error is small, it indicates that the model overfits the data. Conversely, if both errors are large, the model underfits the data.

### 3.0.6 Regularization

Regularization is employed to address the problem of overfitting in machine learning models. Overfitting occurs when the model excessively adapts to the training data, resulting in poor generalization to unseen test data. One common manifestation of overfitting is assigning very high weights to features that happen to be perfectly predictive of the outcome but are not necessarily meaningful.

## 3.1 Logistic Regression

Logistic regression is a widely used algorithm for classification tasks in machine learning. It is a supervised learning method that aims to find an optimal model for predicting the probability of an input belonging to a specific class.

The sigmoid function is employed in logistic regression to map predicted values to probabilities. It is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

To train a logistic regression model, a suitable cost function is required to quantify the error between the predicted probabilities and the true labels. The binary cross-entropy loss is commonly used as the cost function for logistic regression:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

Here, $m$ denotes the number of training examples, $y^{(i)}$ represents the true label of the $i$-th example, $x^{(i)}$ denotes the feature vector of the $i$-th example, and $h_\theta(x^{(i)})$ denotes the predicted probability of $x^{(i)}$ belonging to the positive class.

Stochastic gradient descent (SGD) is a commonly used optimization algorithm to minimize the cost function and determine the optimal parameters $\theta$ for the logistic regression model. The steps to implement SGD for logistic regression are as follows:

In this algorithm, the training data is iterated over in a random order for a maximum of $N$ iterations. For each training example, the predicted probability is computed, and the parameter vector $\theta$

**Algorithm 1:** Stochastic Gradient Descent for Logistic Regression

**Input:** Training data: $X$, labels: $y$, learning rate: $\eta$, maximum number of iterations: $N$
**Output:** Optimal parameter vector: $\theta$

```
Initialize the parameter vector θ randomly or with zeros;
for n ← 1 to N do
    Randomly shuffle the training data;
    for i ← 1 to m do
        Compute the pre dicted probability: h_θ(x^(i));
        Update the parameter vector: θ ← θ − η∇_θ J(θ), where ∇_θ J(θ) denotes the gradient
          of the cost function with respect to θ;
    end
end
```

is updated using the gradient of the cost function. The learning rate $\eta$ controls the step size in each parameter update.

The resulting parameter vector $\theta$ represents the optimized weights for the logistic regression model, which can then be used for making predictions on new, unseen data.

## 3.2 XGBoost

Let us start with the regular decision trees, that make predictions by splitting the input space into regions (leafs) based on feature thresholds, it means assigning a class or value to each leaf.

The next step, boosting, will be based on Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. Boosting is a machine learning technique used to obtain a strong predictive model, by combining multiple weak ones. It aims to better the models sequentially,by training it so that with each subsequent model aiming to correct the mistakes made by the previous models. In each iteration of gradient boosting, the difference between the predicted output and the actual target values is calculated. This difference is called the residual. In each of those iteration a model is trained to predict those residuals and minimize them. Each weak model is assigned a weight based on its performance, which determines its contribution to the final prediction, that needs to be regularized to avoid overfitting.

Extreme Gradient Boosting (XGBoost) is just an optimized version pf the previously discussed gradient boosting. It includes techniques such as parallelization, efficient tree construction, and regularization improvements to enhance performance and scalability.

For each training example (i.e., each data point), gradient value based on the loss function and the current predictions needs to be calculated as well as the Hessian value, that represents the second-order derivative of the loss function with respect to the predicted values, based on the loss function and the current predictions. They are used to optimize the model during each iteration.

**Algorithm 2:** XGBoost Algorithm

**Input:** Training data: $\{(x_i, y_i)\}_{i=1}^{n}$, number of iterations $T$
**Output:** Ensemble model $F(x)$

Initialize ensemble model: $F_0(x) = 0$
Initialize gradients: $g_i = \frac{\partial}{\partial F}\mathcal{L}(y_i, F_0(x_i))$
Initialize Hessian: $h_i = \frac{\partial^2}{\partial F^2}\mathcal{L}(y_i, F_0(x_i))$
**for** $t = 1$ *to* $T$ **do**
    Compute pseudo-residuals: $r_i = -\left[\frac{\partial}{\partial F}\mathcal{L}(y_i, F_{t-1}(x_i))\right]_{F=F_{t-1}(x_i)}$
    Fit a weak learner to the pseudo-residuals: $f_t(x)$
    Update the ensemble model: $F_t(x) = F_{t-1}(x) + \eta \cdot f_t(x)$
    Update gradients: $g_i = g_i + \frac{\partial}{\partial F}\mathcal{L}(y_i, F_t(x_i))$
    Update Hessian: $h_i = h_i + \frac{\partial^2}{\partial F^2}\mathcal{L}(y_i, F_t(x_i))$
**end**
**return** *Ensemble model* $F(x) = F_T(x)$

## 3.3 Neural Network

The neural network is a deep learning model that encompasses a diverse range of non-linear models or parametrizations. These models utilize combinations of matrix multiplications along with other element-wise non-linear operations.

According to Portland State University **?**, the fundamental components of a neural network are the **neurons** organized in **layers**. Neurons within a layer are not connected to each other; they only connect to neurons in different layers. Each neuron receives one or multiple inputs and produces an output. Additionally, each input is assigned a **weight** that represents its significance within the neural network.

The term "neural network" is broad and is also known as "fully-connected networks" or historically as "multi-layer perceptrons" (MLP). The perceptrons were the first deep learning algorithms assumed to function similarly to the human brain, with neurons and electrical impulses (activation functions). However, perceptrons are simpler, taking several binary inputs $x_1, x_2, ..., x_n$ and producing a single binary output.

Training a neural network involves the following steps:

1. Initialize weights for all neurons.
2. Present the input layer with, for instance, spectral reflectance.
3. Calculate the outputs of the neural network.
4. Compare the outputs with, for example, biophysical parameters.
5. Update the weights of the neurons in an attempt to achieve a better match.
6. Repeat these steps until all examples have been presented.

In this thesis, we will work with supervised learning, where our training dataset will have desired outputs. The errors will be calculated, and the parameters will be updated through backpropagation.

The neural network consists of several types of layers: the input layer, hidden layers, and the output layer. Hidden layers are sometimes referred to as a "black box" because the reasons behind the neural network's selection of neuron weights may not be fully understood. However, these connections may still provide meaningful relationships for the algorithm.

The following graph illustrates a simple neural network with 8 neurons: 3 in the input layer, 3 in the hidden layer, and 2 in the output layer. For example, in a binary classification problem, the output in a spam detection system could indicate a spam email in one neuron and a non-spam email in the other neuron.

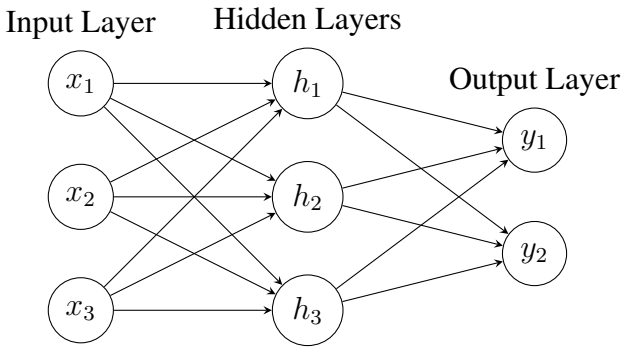


Figure 3.1: Graph of a simple neural network with 8 neurons

### 3.3.1 Forward Propagation

After designing the neural network architecture, the learning process begins. We utilize an activation function to calculate the output of the neural network.

The forward propagation process can be described using the sigmoid function as the activation function. The equations below demonstrate how the forward propagation works, where $(1)$ represents the input layer, $(k)$ represents the output layer, $x$ represents the input data, and $a$ represents the output activations. $W^{(i)}$ denotes the weight matrix of the $i$-th layer, and $b^{(i)}$ represents the bias vector of the $i$-th layer.

$$
\begin{aligned}
z^{(1)} &= W^{(1)}x + b^{(1)} \\
a^{(1)} &= \sigma(z^{(1)}) \\
z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\
a^{(2)} &= \sigma(z^{(2)}) \\
&\vdots \\
z^{(k)} &= W^{(k)}a^{(k-1)} + b^{(k)} \\
a^{(k)} &= \sigma(z^{(k)})
\end{aligned}
$$

Additionally, a **bias** refers to a learnable constant that allows for shifts in the data to better align the function with the patterns in the data. It enables the function to fit the data more accurately, even if the initial shape of the function is not well-suited for the data.

### 3.3.2 Training

The training phase of a neural network is a crucial and challenging process that involves gradient descent and backpropagation. In this section, we will focus on gradient descent, which iteratively updates the network's parameters in the direction of the gradient. While an optimized variant of this algorithm, known as Stochastic Gradient Descent, has been explained earlier, we will briefly introduce the mathematical formula for regular gradient descent to highlight its simplicity. It involves updating the parameters by subtracting the gradient of the cost function multiplied by the learning rate from the current parameter value.

$$\theta := \theta - \alpha \nabla J(\theta)$$

where:

$\theta$ represents the parameters that need to be updated.

$\alpha$ is the learning rate, determining the step size of each parameter update.

$\nabla J(\theta)$ denotes the gradient of the loss function with respect to the parameters, indicating the direction of steepest descent.

### 3.3.3 Backpropagation

This section provides an academic discussion on backpropagation, based on the notes from Stanford. Backpropagation is an efficient method for computing the gradient of the loss function $\nabla J(\Theta)$.

**Theorem:** [Backpropagation or auto-differentiation, informally stated] Suppose a differentiable circuit of size $N$ computes a real-valued function $f : R' \to R$. Then, the gradient $\nabla f$ can be computed in time $O(N)$ by a circuit of size $O(N)$. If $N \leq R'$, the gradient can potentially be computed in even less than $R'$ time.

The theorem of backpropagation (or auto-differentiation) suggests that if a real-valued function can be computed by a differentiable network or circuit, its gradient can be efficiently computed as well. A differentiable network or circuit is defined as a composition of differentiable arithmetic operations and elementary differentiable functions. By composing the circuit using arithmetic operations (e.g., addition, subtraction, multiplication, and division) and elementary functions (e.g., ReLU, exponential, logarithm, sine, cosine, etc.), the derivatives can be easily computed. Backpropagation relies on the **chain rule** to propagate the gradients through the circuit by differentiating them.

The applicability of the theorem is not limited to fully-connected neural networks discussed thus far, but also extends to other types of neural networks with advanced modules.

In the next section, a deeper understanding of the chain rule will be explored, along with the introduction of the concept of a backward function.

#### Chain Rule and Backward Function

To introduce the chain rule, consider a scenario where a variable scalar $J$ depends on some variables $z$. In this case, $\frac{\partial J}{\partial z}$ represents the partial derivatives of $J$ with respect to the variable $z$.

The chain rule is crucial for computing the partial derivatives of $J$ with respect to $z$ from the partial derivatives of $J$ with respect to $u$.

Let $u = (u_1, \ldots, u_n)$ and $g(z) = (g_1(z), \ldots, g_n(z))$. Then, the standard chain rule states that for every $i \in \{1, \ldots, m\}$, $\frac{\partial J}{\partial z_i} = \sum_{j=1}^{n} \frac{\partial J}{\partial u_j} \cdot \frac{\partial g_j}{\partial z_i}$.

Furthermore, the backward function $B[g, z]$ can be introduced, where $g$ represents a module or function and $z$ is its input. The backward function is always a linear map from $R^n$ to $R^m$, where $n$ and $m$ are the dimensions of $u$ and $z$, respectively. It maps $\frac{\partial J}{\partial u}$ to $\frac{\partial J}{\partial z}$ and represents the backward propagation of gradients through the computational graph. The backward function allows efficient computation of gradients when $z$ is changing while $g$ remains fixed. This is particularly useful when optimizing neural networks or complex models, where the gradients need to be computed efficiently during training.

Backpropagation is a key component of auto-differentiation. Therefore, let us examine its general strategy. We can consider a neural network as a complex composition of smaller building blocks, such as matrix multiplication (MM), activation functions ($\sigma$), convolutional layers (Conv2D), and layer normalization (LN), among others. The loss function can also be viewed as a more complex function composed of additional $k$-modules denoted as $M_k(M_{k-1}(\ldots M_1(x)))$. The computation of the loss function involves sequentially applying these modules to the input data.

Backpropagation consists of two passes: a forward pass, where intermediate variables are computed sequentially, and a backward pass, where derivatives with respect to the intermediate variables are computed in reverse order. Let us consider an example with three modules: $M_1$, $M_2$, and $M_3$.

In the forward pass, the computation is as follows:

$$u[0] = x \quad \text{(the input)}$$

$$u[1] = M1(u[0])$$

$$u[2] = M2(u[1])$$

$$u[3] = M3(u[2])$$

In the backward pass, the derivatives are computed in reverse order, starting from the derivative of the loss function with respect to $u[3]$. The computation is as follows:

$$\frac{\partial J}{\partial u[3]} = B[M3, u[3]] \left( \frac{\partial J}{\partial u[3]} \right)$$

$$\frac{\partial J}{\partial u[2]} = B[M2, u[2]] \left( \frac{\partial J}{\partial u[3]} \right)$$

$$\frac{\partial J}{\partial u[1]} = B[M1, u[1]] \left( \frac{\partial J}{\partial u[2]} \right)$$

$$\frac{\partial J}{\partial u[0]} = B[M0, u[0]] \left( \frac{\partial J}{\partial u[1]} \right)$$

Here, we observe that the chain rule enables efficient computation of derivatives with respect to earlier intermediate variables using derivatives with respect to later ones. The efficiency relies on the implementation of small modules with efficiently implementable backward functions.

# Chapter 4

# Methodology

The implementation begins by utilizing two libraries: XGBoost and Shark. The Shark library is an open-source C++ library that facilitates the implementation of machine learning algorithms. Among the available algorithms, the neural network use case is extensively documented. For this implementation, the MNIST dataset is chosen for training and testing. The MNIST dataset is well-known for its use in handwritten digit recognition, comprising 10 classes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) with 784 features.

## 4.1 Neural Network Implementation

The first step involves initializing the variables necessary for the network configuration:

```
std::size_t hidden1 = 200;
std::size_t hidden2 = 100;
std::size_t iterations = 1000;
std::size_t batchSize = 256;
```

This configuration includes two hidden layers with 200 and 100 nodes, respectively. The network will undergo training for a specified number of iterations, and the network's parameters will be updated using the average gradient computed over each batch. In this implementation, Batch Gradient Descent is employed.

Next, the data is imported and divided into training (70%) and testing (30%) sets:

```
LabeledData<RealVector, unsigned int> data;
importSparseData(data, argv[1], 0, batchSize);
data.shuffle();
auto test = splitAtElement(data, 70 * data.numberOfElements() / 100);
std::size_t numClasses = numberOfClasses(data);
```

The neural network model is defined as a series of dense layers, with each layer based on the output of the previous layer:

```
DenseLayer layer1(data.inputShape(), hidden1);
DenseLayer layer2(layer1.outputShape(), hidden2);
LinearModel<RealVector> output(layer2.outputShape(), numClasses);
auto network = layer1 >> layer2 >> output;
```

The error function and loss function are then defined. The cross-entropy function is chosen as the loss function, and the ErrorFunction class calculates the error for the given dataset and network. It computes the average loss by evaluating the loss function for each sample and taking the mean.

```
CrossEntropy<unsigned int, RealVector> loss;
```

```
ErrorFunction<RealVector> error(data, &network, &loss, true);
```

The training process commences with the initialization of the network's parameters, assigning a weight value of 0.001. The error function is initialized, and the Adam optimizer is employed for error optimization.

```
initRandomNormal(network, 0.001);
Adam<RealVector> optimizer;
error.init();
optimizer.init(error);
```

The subsequent loop constitutes the training loop. It performs the optimization step based on the current error function for each iteration and updates the iterations vector and error vector with the optimized values.

```
for (std::size_t i = 0; i != iterations; ++i) {
    optimizer.step(error);
    iterationsVec.push_back(i + 1);
    errorVec.push_back(optimizer.solution().value);
}
```

After the loop, the network's parameter vector is updated with the optimized values obtained from the optimizer, enabling prediction generation.

```
network.setParameterVector(optimizer.solution().point);
```

To evaluate the classification error, the 'ZeroOneLoss' function is employed. It takes the true labels and predicted labels as input, calculating the classification error.

```
ZeroOneLoss<unsigned int, RealVector> loss01;
Data<RealVector> predictionTrain = network(data.inputs());
std::cout << "classification error, train: " <<
   loss01.eval(data.labels(), predictionTrain) << std::endl;
```

The same procedure is applied to the test data. For further analysis, the results are plotted, comparing different network models.

Table 4.1: Neural Network Variations

| Variation | H. Layers | Hidden Nodes | Iter. | Batch Size | Error (Train) | Error (Test) |
|---|---|---|---|---|---|---|
| Layer Variations | | | | | | |
| Test 1 lr:0,001 | 1 | 100 | 1000 | 256 | 0.0003 | 0.084 |
| Test 2 lr:0,001 | 2 | 200, 100 | 1000 | 256 | 0 | 0.078 |
| Test 3 lr:0,001 | 3 | 300, 200, 100 | 1000 | 256 | 0 | 0.068 |
| Iteration Variations | | | | | | |
| Test 1 lr:0,001 | 2 | 200, 100 | 500 | 256 | 0.0017 | 0.083 |
| Test 2 lr:0,001 | 2 | 200, 100 | 1000 | 256 | 0 | 0.075 |
| Test 3 lr:0,001 | 2 | 200, 100 | 2000 | 256 | 0 | 0.072 |
| Batch Size Variations | | | | | | |
| Test 1 lr:0,001 | 2 | 200, 100 | 1000 | 128 | 0.0003 | 0.079 |
| Test 2 lr:0,001 | 2 | 200, 100 | 1000 | 512 | 0 | 0.079 |
| Test 3 lr:0,001 | 2 | 200, 100 | 1000 | 1024 | 0 | 0.082 |
| Learning Rate Variations | | | | | | |
| Test 1 lr:0,01 | 2 | 200, 100 | 1000 | 256 | 0.0486 | 0.093 |
| Test 2 lr:0,001 | 2 | 200, 100 | 1000 | 256 | 0 | 0.078 |
| Test 3 lr:0,0001 | 2 | 200, 100 | 1000 | 256 | 0.0034 | 0.061 |

**Analysis:** The greater number of layers and nodes generally seems to improve both the train and test classification error. Increasing the iteration number may improve performance, but there may be a convergence or overfitting point. Batch size variations do not significantly impact performance, and a learning rate of 0.001 appears to be suitable for this problem.

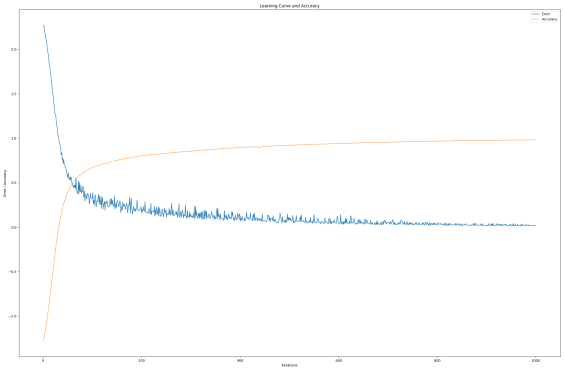

Figure 4.1: 1 Layer

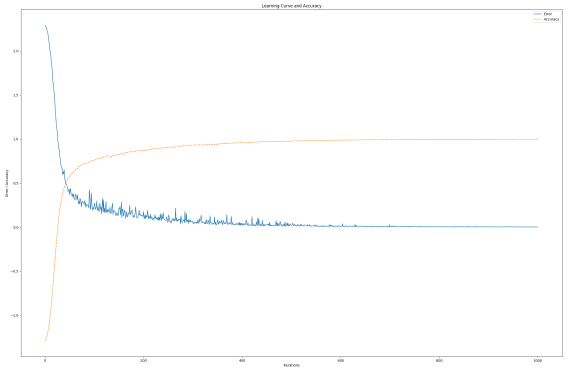

Figure 4.2: 2 Layers

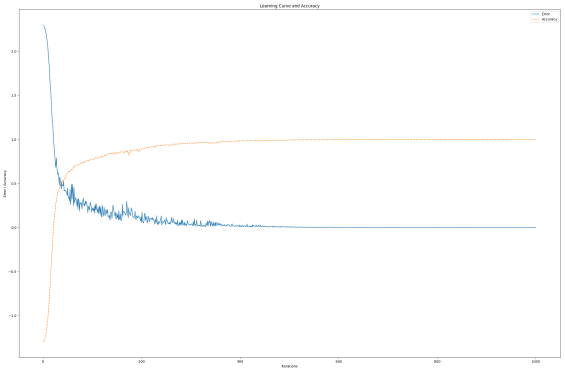

Figure 4.3: 3 Layers

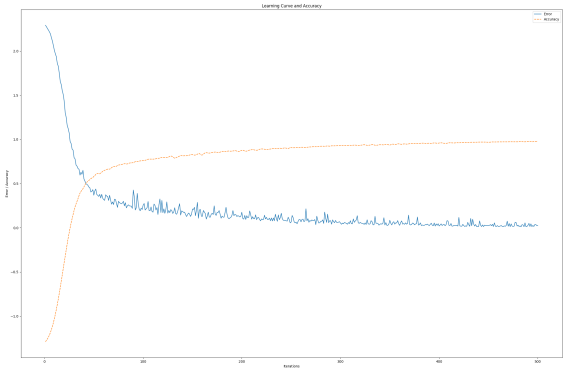

Figure 4.4: 500 Iterations

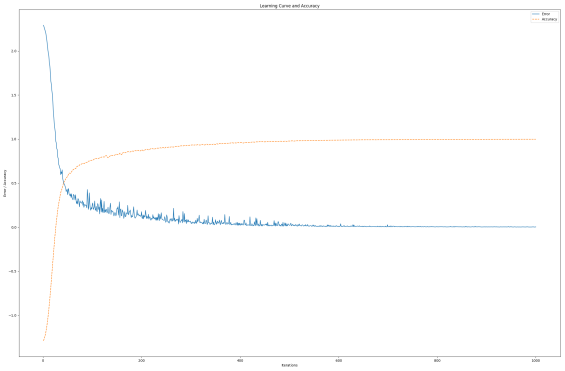

Figure 4.5: 2000 Iterations

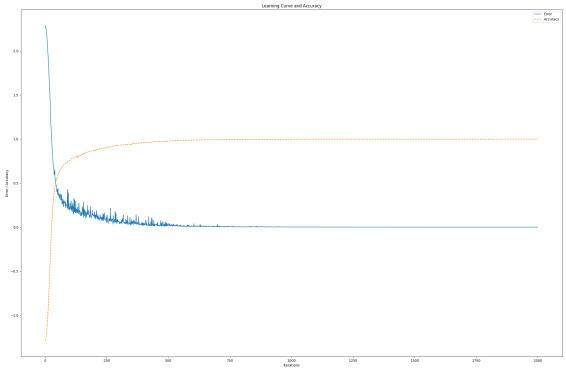

Figure 4.6: 5000 Iterations

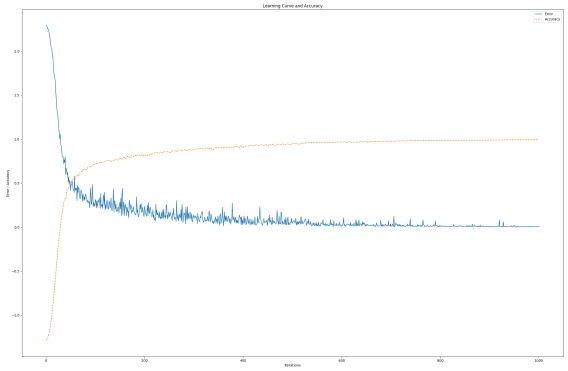

Figure 4.7: Batch Size: 128

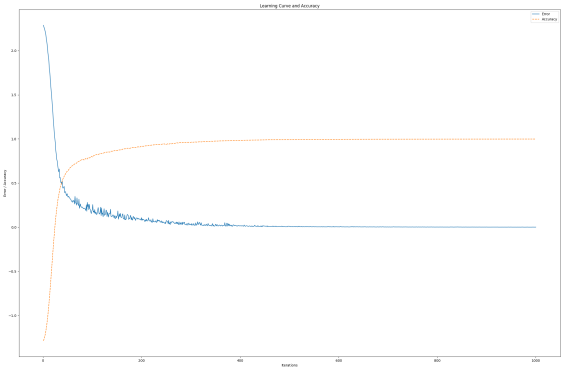

Figure 4.8: Batch Size: 512

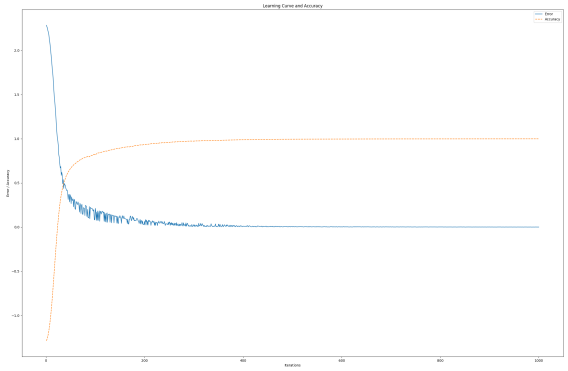

Figure 4.9: Batch Size: 1024

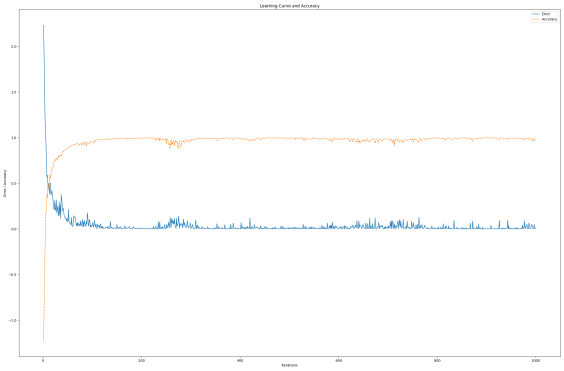

Figure 4.10: Learning rate: 0.0001

## 4.2 XGBoost Implementation

## 4.3 Logistic Regression Implementation

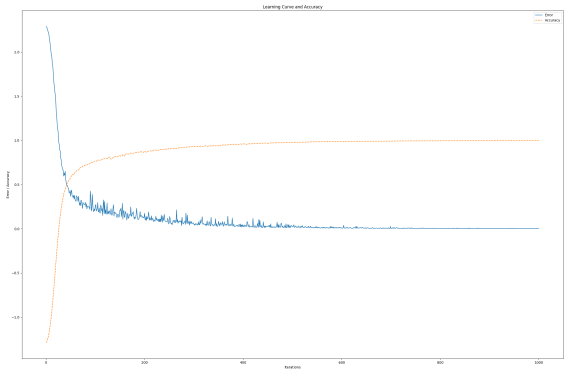


Figure 4.11: Learning rate: 0.001

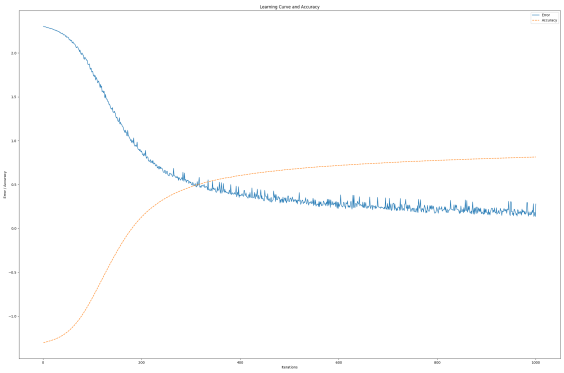


Figure 4.12: Learning rate: 0.01

# Chapter 5

# Federated Learning

The last part of my thesis will be based on the paper "Communication-Efficient Learning of Deep Networks from Decentralized Data", which focuses on a decentralized approach for training machine learning models on remote devices without uploading sensitive data to a central server. This technique is called Federated Learning and focuses on the positive influences of this approach for privacy purposes. It allows users to train the global neural network model by performing their own calculations locally and then sending the updates to the global server. This disables the global access to raw data of the devices, which in turn signifies the increase in security and privacy. An algorithm, FederatedAveraging, will be introduced as an algorithm for practical decentralized learning.

The data that is most suited for federated learning will also be discussed. The ideal problems for said algorithm include real-world data from mobile devices, privacy-sensitive or large data that should not be logged centrally, and tasks with inferred labels from user interactions. This data provides suitable tasks for image classification and language modelling, which align with the task requirements necessary for federated learning and an be computed using it.

In the paper, an approach is shown that minimizes the privacy concerns even more, as the number of required updates is lowered. One of the approaches introduced in the paper is called "Federated Optimization" and distinguished the federated approach from reguoar distibuted optimization problems, as it has to consider issues such as non-independent and identically distributed (non-IID) training data on each client, unbalanced data distributions, a large number of participating clients, and limited communication capabilities of mobile devices.

# Chapter 6

# Results and Analysis

# Chapter 7

# Conclusion

# Appendix A

# Appendix Title