DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Neural Networks Solving Linear Systems

Iremnur Kidil

DEPARTMENT OF INFORMATICS

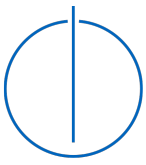
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Neural Networks Solving Linear Systems

Neurale Netzwerke, die lineare Systeme lösen

Author:	Iremnur Kidil
Supervisor:	Prof. Dr. Christian Mendl
Advisor:	Dr. Felix Dietrich
Submission Date:	15.04.2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

15.04.2021 Iremnur Kidil

Acknowledgments

I would like to thank my advisor, Dr. Felix Dietrich, for the guidance he has given me throughout this thesis.

Abstract

Solving linear systems is a fundamental part of engineering and computational science problems since it is relied on in many fields such as numerical simulations, image and signal processing, and fluid dynamics. The linear equations can be collectively represented as $Ax = b$, a system consisting of a matrix A, a solution x, and data b. In the problem, matrix A and data b are given, and we need to find x. In this work, we explore two different computation methods for solving linear systems, firstly using a linear solver and secondly implementing a neural network. For the linear solvers we benefit from the NumPy linear algebra functions `solve()` and `lstsq()`. As a second method for solving linear systems, we build neural networks that represent the solution vector x. A and b are fixed and the goal of the network is to find an estimation of x that minimizes the error r in $r = Ax - b$. As an advantage neural networks can estimate solutions for large-scale problems. After the direct solution of linear systems with neural networks, we discuss and explore special settings, like applying custom loss functions, in which using neural networks to solve a linear system is beneficial against the standard solvers.

Contents

1 Introduction

There are several methods that can be utilized to solve linear systems. One of the options is to use pre-existing linear solvers. In this thesis, we propose a second option, which is building neural networks. These neural networks are created because of the ability to work with custom loss functions and to compute estimations of x for large-scale problems. Thus, the aim of the thesis is to solve a linear system $Ax = b$ with a neural network and discuss the advantages of using neural networks against linear solvers.
The thesis starts with a brief explanation of linear systems. We then demonstrate different linear solvers to solve these linear systems. For the implementation of the neural networks we use Python. Therefore, to implement linear solvers we benefit from NumPy, which is a package in Python that enables the use of operations for scientific computing. We take a deeper look at the NumPy linear algebra functions that help us to find a solution x for the problem $Ax = b$. The linear algebra functions that are discussed consist of three options. First option is inverting the matrix A with the method `inv()` and then multiplying it with b with the help of `dot()`, since $x = A^{-1}b$. Due to the high complexity of inverting a matrix, we work with the `solve()` or the least squares method `lstsq()`. If A is square and has full rank, the `solve()` method can compute an exact solution with the help of a LAPACK routine. Nevertheless, if either of the conditions are not true, we use the `lstsq()` method. The least squares method uses a regression procedure to determine the best fit hyperplane to a given dataset and returns least-squares solution x to our linear system $Ax = b$. The working principles of the methods `solve()` and `lstsq()` are explained comprehensively.
We then represent the features of the nonlinear networks in depth because our linear network adopts some of the main features of nonlinear networks. The reason why we refer to the neural network specifically as linear is that usually neural networks represent solutions to nonlinear problems. However, our network represents a solution for a linear system. How we approach building neural networks is primarily based on the paper "Neural networks for solving systems of linear equations and related problems" [5] that was written in 1992 by Cichocki and Unbehauen. The theory behind the paper is to solve a linear equation $Ax = b$ iteratively in order to converge to a solution x as close as possible to the true solution. Normally, we expect from a neural network to train with a dataset that has numerous information. The reason for that is to acquire a network that is as accurate as possible for solving any other related problem. In our implementation, we work with a different method. Rather than feeding the network different matrices A and b, we have a fixed matrix A and a fixed matrix b as training samples.
After explaining the approach of Cichocki and Unbehauen, we demonstrate the reimple-

mentation of the iterative solution method. We implement the mentioned method in two different ways. At first, we used only numerical operations with respect to the paper, because at that time there were no open source platforms for machine learning like TensorFlow [3]. We then implement the same iterative solution method with TensorFlow to benefit from its library. With TensorFlow, we can deal with large datasets and for our problem $Ax = b$, a large matrix is treated as a large dataset. Hence, every row represents one data point. We then use software frameworks to solve a linear system iteratively.
One of the most crucial advantages of using the iterative solution method is that we can benefit from custom loss functions. In the thesis four different energy functions, being ordinary least squares, iteratively reweighted least squares, least absolute value and chebyshev problems, are thoroughly explained. These energy functions used in the work of Cichocki and Unbehauen serve the duty of a loss function, and we implement these different loss functions without using TensorFlow as well as using TensorFlow. After we discuss the advantages and the disadvantages of each energy function, we compare behaviours of the losses for a small-scale problem.
Before concluding the thesis, we also introduce the results of a large-scale problem solved by our iterative method. The implemented large-scale problem can not be solved by linear solvers, at least on our laptops. The problem consists of a matrix A that has the shape (10000×10000). We then compare the computation times for both implementations with and without using TensorFlow. Further, we examine which loss function performed the best on our example.

2 State of the Art

2.1 Linear Systems

In this section, we firstly describe what a linear equation is. Further, we proceed to systems of linear equations. We provide the matrix form of a linear system as well.

- Linear Equations
  A linear equation with variables $x_1, x_2, \ldots, x_n$ has the form:

  $$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b,$$

  where $a_i \in \mathbb{R}$ represents the coefficients of x_i and $b \in \mathbb{R}$ represents the constant term [4]. It is called a linear equation because the set of solutions forms a straight line in the hyperplane.

- System of Linear Equations
  A finite collection of the linear equations in same variables is a system of linear equations. An example of a linear system with n different variables and m different equations can be defined as:

  $$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}.$$

  We can demonstrate the same linear system as $Ax = b$. In matrix notation the equation has the following form:

  $$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

  A solution set $x = (x_1, x_2, \ldots, x_n)$ is a tuple that makes each equation a true statement [4].

2.2 Linear Solvers

One option to solve linear systems is to use linear solvers. For the implementation of the neural networks solving linear systems, we use Python. Therefore, we benefit from the NumPy linear algebra functions. In this section we discuss different methods to solve a linear system $Ax = b$ with the help of NumPy. Then, we explain LU decomposition and least squares method in depth to clarify how the linear algebra functions `solve()` and `lstsq()` work respectively.

2.2.1 NumPy Linear Algebra Functions

- `x = np.linalg.inv(A).dot(b)`
  The logic behind the solution method is as follows:

  $$Ax = b \implies x = A^{-1}b.$$

  The `inv()` method finds the inverse of a matrix, in our case A^{-1} [22] and the `dot()` method calculates the dot product of A^{-1} and b. An exact solution of x is computed [2].
  If the inverse of a matrix exists, a common technique is to use Gauss-Jordan elimination. Nevertheless, Gauss-Jordan elimination is an inefficient and expensive mechanism when it comes to large matrices. Depending upon the size of the matrix, the cost of the operation scales cubically. The complexity of finding an inverse is $O(n^3)$. In many scientific computing applications, it is sufficient to compute approximations of x rather than computing the exact result.
  The method computes the inverse of the matrix A in:

  $$AA^{-1} = I \tag{2.1}$$

  by solving for A^{-1}. I is a $n \times n$ identity matrix that has ones in the diagonal and zeros elsewhere. To solve 2.1 A is factorized with LU decomposition. We demonstrate how LU decomposition works in 2.2.2. In order to compute x with the found A^{-1}, we need an extra matrix multiplication $x = A^{-1}b$.

- `x = np.linalg.solve(A, b)`
  Instead of chaining both methods `inv()` and `dot()`, the `solve()` method can be performed directly. It computes an exact solution as well. However, in order to benefit from the `solve()` method, A must be square and well-determined, meaning it needs to have full rank. A square matrix has full rank either when its rows or its columns are linearly independent [2].
  The method does not compute the inverse of the matrix A. Computation of the solution is provided by LAPACK's LU decomposition. Thus, A is factorized with LU decomposition and then x is solved with forward and backward substitutions. LAPACK, abbreviation for Linear Algebra Package, is a software library for numerical

linear algebra. With the help of a routine in gesv, the solution to a linear system with a square matrix A and multiple right hand sides is computed. The reason why `solve()` is preferred over the `inv()` method is that more floating point operations are used to solve for A^{-1}, an $n \times n$ matrix, than for x, a vector that has n elements. Thus, more floating point operations cause more numerical errors and slower performance.

- `x = np.linalg.lstsq(A, b, rcond='warn')`
  If matrix A is not square and has not full rank, we use the least squares method `lstsq()`. It returns the least-squares estimation to a linear matrix equation, $Ax = b$. Hence, instead of computing an exact solution, we obtain the best solution. Matrix A can have the amount of linearly independent rows that are less than, equal to, or greater than A's number of linearly independent columns. Found x minimizes the squared Euclidean norm of the term $(b - Ax)$. Hence, we take the square of each element and sum them. Later in 2.2.3, we take a deeper look at the least squares method. To have a better understanding of the NumPy linear algebra function `lstsq()`, we observe the parameters and the output values.
  Parameters:
  - A: the $m \times n$ coefficient matrix.
  - b: the dependent variable values. b can be a vector or a matrix. Assume b is a $m \times k$ matrix, then the least squares solution is calculated for each column k. Note that different types of regularizations are discussed later in the thesis.
  - `rcond`: a cut-off ratio for small singular values of A.

  Returns:
  - x: the least-squares solution.
  - residuals: an array for sums of squared residuals. It is empty, if the rank of the matrix A is $< n$ or $m \leq n$.
  - rank of the matrix A.
  - s: an array that holds singular values of A.

  Finding an exact solution is also possible. If A is square and of full rank, and when the rcond parameter is set to zero, then the `lstsq()` method acts as the `solve()` method [2].

2.2.2 LU Decomposition

LU decomposition process computes U, which is an upper triangular matrix, and L, which is a lower triangular matrix, such that $A = LU$ [26]. A demonstration can be given with

matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}, U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

We first modify A and convert it to U. We do this by performing only one type of row operation, which is replacing a row R_i by $R_i - kR_j$. Next step is to compute L. L records the k-values with respect to the positions in the matrix U. Hence, we compute U using Gaussian elimination (only the specified row operation) and L by saving the steps in Gaussian elimination.
To give a better understanding we perform LU decomposition on the matrix A that we used in the implementation of our neural network described in 3.2.2.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \xrightarrow{R_2 - 0R_1 \to R_2} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \xrightarrow{R_3 - 2R_1 \to R_3}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & l_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 0 & 3 & -3 \end{bmatrix} \xrightarrow{R_3 - \left(\frac{3}{2}\right) R_2 \to R_3} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 0 & 0 & \frac{-21}{2} \end{bmatrix} \tag{2.2}$$

After completing the row operations, we achieved the factorized A in form $A = LU$ in the last step 2.2, which is:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 0 & 0 & \frac{-21}{2} \end{bmatrix}.$$

2.2.3 Least Squares Method

The least squares method is a solution technique for an overdetermined set of equations $Ax \approx b$, where $A \in R^{m \times n}$ is the matrix model, $b \in R^m$ is the measurement vector, and $x \in R^n$ is the unknown vector [5, 11, 12, 21]. A and b are given in advance and x is unknown. We can define the linear estimation model as follows:

$$Ax = b + r = b_{true}, \tag{2.3}$$

where $r \in R^m$ is the unknown vector of measurement errors that we want to minimize and $b_{true} \in R^m$ is the vector of true values [5, 21].

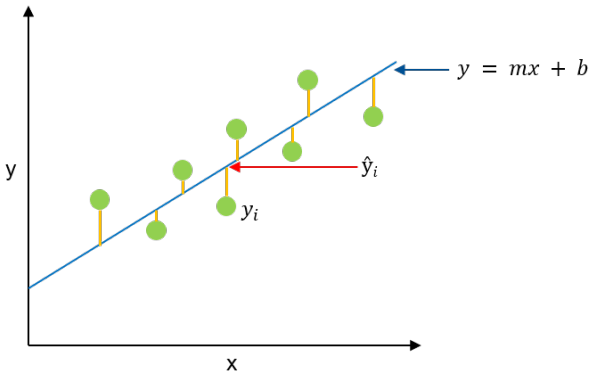


Figure 2.1: Least Squares Regression

Figure 2.1 demonstrates how the least squares method is computed for a two-dimensional problem. Assume, our dataset consists of n data points (x_i, y_i), $i = 1, \ldots, n$. The coefficients of x represent A and y represents b_{true} in 2.3. The dataset is shown with the green dots. x_i is an independent and y_i is a dependent variable that is computed by observation. The goal is to obtain a best fit line to a given dataset that minimizes the residuals r. The difference between the actual output y_i and the output predicted by the model $\hat{y}_i$, demonstrated with orange lines, gives us the r_i. According to estimation model 2.3, $\hat{y}$ corresponds to b. The value predicted by the model can be also defined as $f(x_i, \beta)$. For a two-dimensional problem, the y-intercept can be denoted with β_1 and the slope with β_0. Thus, $f(x_i, \beta) = \beta_0 x + \beta_1$. In order to find the optimal parameters, for instance β_0 and β_1 for the given problem, the sum of the squared residuals is computed:

$$S = \sum_{i=1}^{n} r_i^2$$

and the method tries to minimize S. Least squares method is sensitive to the outliers [5]. If there are outliers in the data, the line will be shifted in the direction of the error.

2.3 Neural Networks

In this section we take a deeper look at the nonlinear neural networks. The iterative method that we proposed in this thesis to solve linear systems adopts the general features of the nonlinear neural networks. Therefore, it is crucial to discuss the concepts regarding nonlinear neural networks at first.

2.3.1 Introduction to Neural Networks

Artificial neural networks are attracting attention due to the outstanding performance inpattern recognition and modelling of nonlinear relationships involving large numbers of variables. The human brain is the inspiration for creating artificial neural networks. Thus, a neural network is a computer model with an architecture that essentially mimics the organizational skills of the human brain and the knowledge acquisition. Layers in an artificial neural network represent the network of neurons in the brain [27]. The operating principle of a biological neuron is to receive electrical signals from other neurons that have different weights. The neuron is fired if the value of all the electrical signals is big enough, otherwise, the neuron is in an inactive state. Based on the principle of how the biological neurons work, artificial neurons are designed to have dynamic states that can hold values [7, 10].
There are three fundamental components of an artificial neural network, being: the input layer, the hidden layer, and the output layer. Input layer is where the network meets the presented input and output layer keeps the response of the network to the input. The intermediate layers, also called hidden layers, enables us to compute complicated mappings between patterns. These layers are interconnected processing elements, referred to as neurons. Neurons in an artificial neural network interact with each other via weighted connections and they together form layers. Each neuron in the former layer is connected with all the neurons in the next layer [10].
The number of neurons and layers depends on the user and on the specific application. For instance, if the hidden layer has few nodes, the network will not be able to properly perform the task it is trained on. In contrast, if the hidden layer has too many neurons, the network will not be able to generalize the given data and the training time will be longer [10]. This is because the patterns will be memorized by the network, which is a common issue in deep learning [8]. We want from our network to estimate outputs generally from the dataset, rather than memorizing them.

2.3.2 General Overview on Training and Testing

In order to train a network, a dataset is given with the actual input and corresponding output. During training, information is passed from the input layer, to the hidden layers and the output is given by the outermost layer [8, 27]. The goal of the network is to make an estimation as accurately as possible for a given input. Thus, a high number of data is fed to the model venturing the increase of time. The number of the training data should be less than five to ten times the number of connections in the network [10, 14]. One more aspect to achieve an optimal solution is to feed data randomly in each iteration while training. The network will generalize the information and will not be biased by the order of it [8].
The entire dataset is fed to the network during training with respect to the number of epochs. One epoch, also called a cycle, is a pass through the set of training data along with the update of the weights. The number of epochs depends on the specific application. At

the end of each cycle, weights are updated. After training is finished, the network is expected to produce accurate outputs with minimum errors regarding the given dataset. To use the model for different datasets, weights are stored in the network [10].
The network is tested with a testing dataset, also called a testset, to evaluate how well the trained network with trained weights performs. During the testing phase, no learning and weight adjustment take place. In order to evaluate results, at least 10-20% of the dataset [14] should be kept. The predictions of the model are compared to the target output values. They should be reliable considering the input values are in the range of the training dataset. If the model predicts the output accurately, it is established that the model generalizes the information successfully and it can be trusted. After acquiring convenient results, the model can be used in practical applications [8, 10].

2.3.3 The Perceptron: Forward Propagation

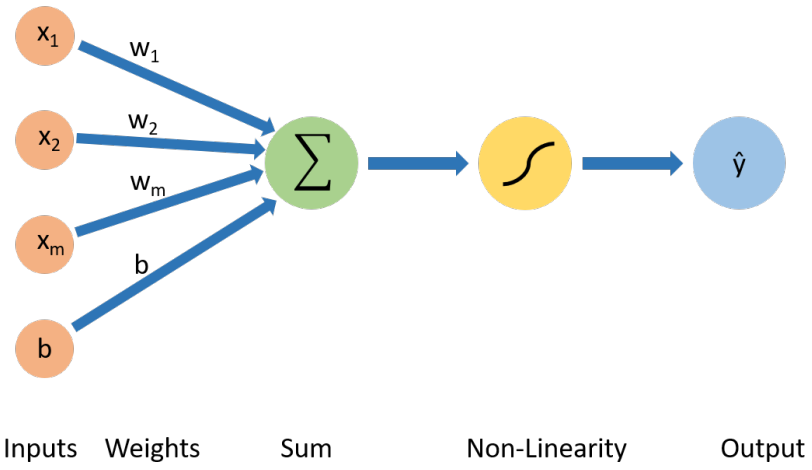


Figure 2.2: Perceptron

The information processing in a neural network starts with data being fed to the input layer. Due to the structure of the interconnected layers, input data is processed with the associated weights. Output from each neuron is produced by multiplying the neuron's input and corresponding weight vector. Then, the result is passed through an activation function and it is supplied to the next layer as the input [5, 27]. Hence, the number of neurons in the output from the prior layer equals the input into the current layer [8]. At the end, the data we want to obtain is the values of the neurons in the last layer. We can demonstrate the mentioned process with a model, called the perceptron.
Perceptron is the structural building block of deep learning that demonstrates the model of a biological neuron. Figure 2.2 shows an example of a perceptron. The input is denoted by $x = [x_1, x_2, \ldots, x_m] \in \mathbb{R}^m$ with m entries and there is also a bias b. Bias is a special

form of weight and it is optimized during the training process along with the weights. Weights, defined as $w = [w_1, w_2, ..., w_m] \in \mathbb{R}^m$, connect the corresponding input to the next layer. Strength of the relationships between the interconnected neurons are determined by the weights. In multilayered neural networks there are two fundamental operations for the modeling and the training process, namely forward propagation and backpropagation [7, 8, 10].
Here, we elaborate the forward propagation. The logic behind the forward propagation is to sum the multiplication of inputs with the corresponding weights and the bias. We can denote the sum with z:

$$z = b + \sum_{i=1}^{m} x_i w_i.$$

Bias is a constant that helps to shift the activation function. The value z is passed to an activation function denoted by g, where the response state of the neuron [7] is simulated. Thus, we obtain the output y:

$$y = g(z).$$

Overall, the output of a single perceptron is calculated with:

$$y = g(b + \sum_{i=1}^{m} x_i w_i). \tag{2.4}$$

The goal of the forward propagation is to generate a prediction and calculate the loss at the meantime. Computation of loss is described in 2.3.5 in depth.

2.3.4 Activation Functions

Activation functions have a crucial role for artificial neural networks. They enable the network to learn by providing nonlinear mappings between the input and the corresponding output. Aside from the nonlinear activation functions, there are also linear activation functions. Linear functions are usually not preferred because the network can only adapt to the linear changes in the input. However, in the real world, most relations are nonlinear. Nonlinear activation functions allow the network to learn the nonlinear dependencies in data. In other words, data that can not be classified linearly can be differentiated using nonlinear activation functions [7, 27].
In this thesis, we will focus on linear activation functions only, because we concentrate on the solution to linear problems. Nevertheless, the loss functions we discuss for these linear systems can be nonlinear, and in principle can be treated as a nonlinear activation function as well. Therefore, it is useful to also discuss nonlinear activations at this point.
In backpropagation derivatives of the activation functions in each layer need to be calculated [7]. Therefore, it is crucial for activation functions to be differentiable almost everywhere with $h = \mathbb{R} \to \mathbb{R}$ [13].
There are several activation functions that are used in different scenarios and each of them

has different features. For almost all settings, choosing the right function for a specific network requires trial-and-error, since there exists no rules about which activation function will perform better in different scenarios. Here we elaborate the most common activation functions and examine the advantages and disadvantages.

1. Linear Activation Function
   Activation function is called linear because it is proportional to the given input. Ad-

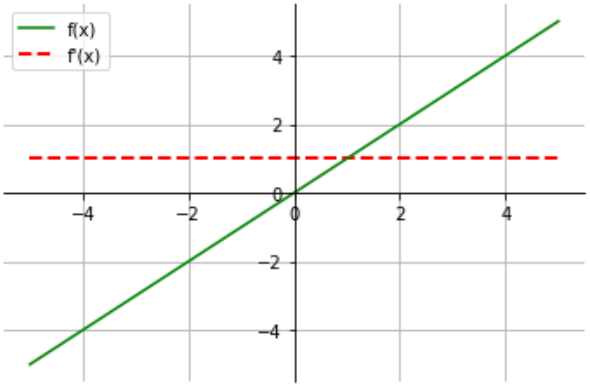


Figure 2.3: Linear Activation Function and the Derivative

   vantage:

   - A linear activation function has a derivative m, considering that a linear line has the function $y = mx + c$. The constant value m can be selected arbitrarily. Figure 2.3 is an example of a linear function with slope $m = 1$ and $c = 0$.

   Disadvantage:

   - The derivative of the function will always be the same. It is not sufficient enough to adjust weights and biases during backpropagation with a constant value that is independent from the input. Hence, linear activation functions will perform poorly for nonlinear tasks, since the error of the model will not be improved on each iteration [27].

2. Nonlinear Activation Functions

   a) Sigmoid
   Sigmoid activation function, also known as the logistic curve, is a nonlinear s-shaped function [10]. It is one of the most common activation functions [7, 10, 30]. As seen in Figure 2.4, the output is scaled between 0 and 1 [8] like a probability distribution.
   Sigmoid function is defined as:

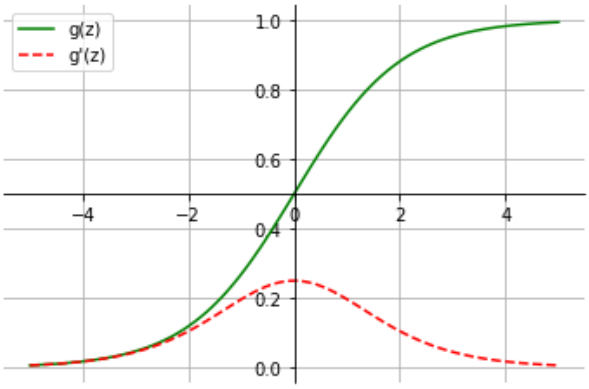


Figure 2.4: Sigmoid and the Derivative

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

and the derivative of the function is:

$$\frac{\mathrm{d}}{\mathrm{d}x}\, g(z) = g(z)(1 - g(z)).$$

The output values have the same sign. Most probable outcome for the given input pattern is the output with the largest value [8].
Advantage:

- It is well suited for problems, where we need to predict the probability as an output.

Disadvantage:

- There is a vanishing gradient problem [20] because it results zero gradient in the limit [7, 13]:

$$\lim_{x \to +\infty} g'(z) = 0$$

$$\lim_{x \to -\infty} g'(z) = 0.$$

  While backpropagation the outputs are chained with respect to the gradient descent algorithm one another in the direction of inner layers. Hence, the inner layers have reducing gradients towards 0 causing them to contribute less to the learning process. At the end, the network can have inaccurate results and it can take longer to compute outputs. Therefore, sigmoid activation function is used in the output level rather than hidden layers [7].

b) Tanh
Tanh or hyperbolic tangent activation function has similarities with sigmoid,

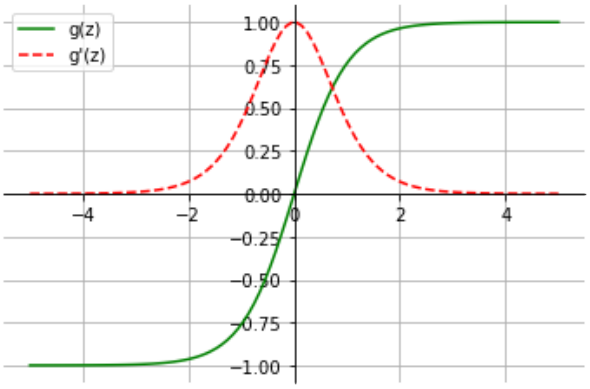


Figure 2.5: Tanh and the Derivative

such as being s-shaped. However, as seen in Figure 2.5 this function is symmetrical to the origin [27], which means the output values of the neurons can have different signs. The output is scaled between -1 to 1.
Tanh function is defined as:

$$g(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

and the derivative of the function is:

$$g'(z) = 1 - \tanh^2(z).$$

Advantage:

- The mean of the input data is approximately zero. In other words, the input data is normalized. Therefore, the convergence is faster than sigmoid [18] making tanh activation function more preferable. Also, tanh achieves a lower classification error [7, 9].

Disadvantage:

- Like the sigmoid activation function, there is a vanishing gradient problem. Therefore, using sigmoid and tanh activation functions on the hidden layers is not favored.

c) ReLU
According to researches in the field of neuroscience, only one to four percent of neurons are in an active state at the same time. In contrast, the activation functions such as sigmoid and tanh activate almost half of the neurons in the network at the same time. Since each neuron in an active state goes through operations like forward propagation and backpropagation, it becomes difficult for the network to train. Rectified linear unit is introduced to improve efficiency

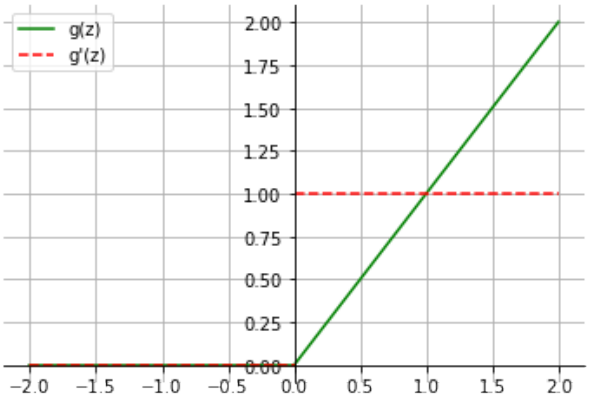


Figure 2.6: ReLU and the Derivative

regarding the mentioned issue [7].
ReLU is one of the most widely used activation functions [7, 27] and it is defined as:

$$g(z) = max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}.$$

The derivative of ReLU is:

$$g'(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}.$$

Advantages:

- Neurons are activated partially [27]. Therefore, the efficiency is higher, since each iteration the network does not operate all the nodes. If the output from the activation function is 0, the neuron is deactivated as seen in Figure 2.6, meaning the output from the neuron will not be delivered as the input to the next layer. ReLU activation function can only be used in the hidden layers of the network.
- Since the derivative of the function is 1, when the input $x > 0$, the vanishing gradient problem is solved [6, 7, 20]. The training process is faster and that is why ReLU is preferred most of the time [15, 27]. In deep neural networks sigmoid and tanh activation functions are replaced by ReLU and ELU to solve the vanishing gradient issue and to make convergence speed faster [7].
- As opposed to sigmoid and tanh, it is cheaper to compute ReLU because the activation function does not contain an exponential function.

Disadvantages:

- A neuron is deactivated when the input x is less than 0 preventing updates on the relative weights [7, 20].
- Since the function has a linear nature, it is not well suited for classification problems like tanh and sigmoid.

d) ELU
ELU is a variant of ReLU, which stands for exponential linear unit. The goal

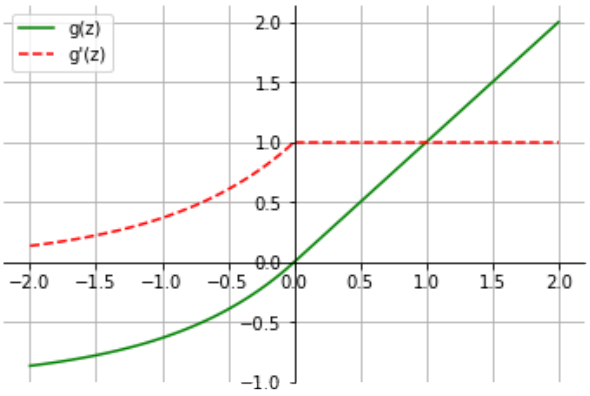


Figure 2.7: ELU and the Derivative, $\alpha = 1$

is to decrease the bias shift effect of the ReLU by pushing the activation means closer to zero [6, 7]. Negative outputs can be produced with $\alpha(e^z - 1)$. An example is demonstrated in Figure 2.7 with $\alpha = 1$. ELU is defined as:

$$g(\alpha, z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$$

and the derivative of the function is:

$$g'(\alpha, z) = \begin{cases} 1 & \text{if } z > 0 \\ \alpha e^z & \text{if } z \leq 0. \end{cases}$$

Advantages:

- Like ReLU, vanishing gradient issue is prevented because of the constant derivative in the positive part.
- The average of the output comes close to zero. As a consequence ELU converges faster.
- Experimental results [6] shows that when there are more than five layers in the network, ELU generalizes better than ReLU. Furthermore, ELU enables faster learning [7].

Disadvantage:

- It is time consuming for the network to search for an optimal α.

2.3.5 Loss functions

In this thesis, we will elaborate different energy functions in 2.4.2 that we used as loss functions in our networks. The logic of the energy functions is the same as the loss functions that are used in nonlinear neural networks. Therefore, we discuss different loss functions in order to demonstrate their role in the learning process.
A loss function can also be referred to as a cost function, an empirical risk or an objective function. The cost incurred from incorrect predictions is measured by the loss of our network. We mark the loss as L and it is computed by $L(f(x_i; W), y_i)$ [31], where $f(x^{(i)}; W)$ is the predicted output of the model with respect to the weights. The actual output that we want to acquire is $y^{(i)}$ and error is the difference between the actual output and the predicted output of the model. According to the error, weights of the neurons are modified. The loss is carried out through optimization algorithms [25], which are explained in 2.3.6. To measure the total loss over the entire dataset we compute the empirical loss:

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i; W), y_i).$$

For each input x_i the loss is calculated and summed up. To find the average loss of the network the sum is divided by the number of training training samples n [19].
There are different types of loss functions serving different purposes. Here, we discuss some of the most common loss functions and which loss function to opt for in different scenarios.

- Binary Cross Entropy Loss
  The loss function is defined as:

$$J(W) = -\frac{1}{n} \sum_{i=1}^{n} y_i \log(f(x_i; W)) + (1 - y_i) \log(1 - f(x_i; W)). \tag{2.5}$$

  When there are only two label classes (e.g. a binary classification problem) using binary cross entropy is beneficial. The output can be left or right, 1 or 0, A or B. The activation function must be compatible with the loss function. For binary cross entropy, the compatible activation function can be selected as sigmoid.
  The reason for using logarithmic operations is to penalize bad predictions. For instance, if the associated probability belongs to the true class 1, then we need the loss to be 0. However, if that probability is falsely a lower value such as 0.1, the model should penalize the bad prediction. Thus, the loss needs to be significantly higher.
  Log of a value that is between 0 and 1 is negative. Hence, taking the negative log serves the above mentioned purpose perfectly. $-\log(x)$ is equal to 0 when x is 1. For

probabilities that are closer to zero, the loss increases exponentially. Say a training example supposed to be 1, as the target value. However, the model predicted the output as 0.6. That indicates there is a probabilistic false negative of 40%, meaning the model has 40% confidence in the wrong result from a Bayesian perspective. Hence, this 40% is penalized by returning the value $-\log(0.6) \approx 0.22$. For the second term of the equation 2.5, the same logic applies as probabilistic false positives [16].

- Mean Squared Error Loss
  For regression models that output continuous real numbers, mean squared error loss is one of the most common loss functions. The function is defined as:

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(y - f(x_i; W))^2.$$

  The error is calculated as the average of squared differences between the actual and predicted values [28]. The loss function penalizes large errors by taking the square of them.
  The loss function is sensitive to outliers, noisy data. The mean target value is the optimal prediction for the input data in mean squared error loss function. In contrast to mean squared error, for mean absolute error the optimal prediction is the median of the data. One can utilize mean squared error loss function, when the target data is distributed normally around the mean value with respect to the input [5].

- Mean Absolute Value Error Loss
  The difference from the mean squared error loss is that instead of taking the square of the error, we take the absolute value. The function is defined as:

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}|y - f(x_i; W)|.$$

  Mean absolute value error loss is insensitive to the outliers. The median target value of the inputs is the optimal prediction for this loss function. Because of the function not being quadratic, the gradient magnitude does not depend on the error size, rather it depends on the sign of the error. Therefore, convergence problems might occur, since the gradient magnitude will be large even though the error is small. Nevertheless, it is rational to use the mean absolute value error loss, when we do not want our regression to be affected by the outliers as much as the mean squared error loss [5].

2.3.6 Loss optimization

To achieve the lowest loss, we need to find the weights W^* that minimize the loss function shown below:

$$W^* = \underset{W}{argmin}\frac{1}{n}\sum_{i=1}^{n}L(f(x_i; W), y_i),$$

$$W^* = \underset{W}{argmin} J(W).$$

$W = w_0, w_1, \ldots, w_n$ is the collection of weights across the neural network from all the layers. In an optimization problem we want to optimize all of the weights to minimize the empirical loss. In 2.3.5 we described loss as a function of the network weights with $J(W)$. To give an example, we demonstrate a loss landscape with a loss function that depends on only two weights $J(w_0, w_1)$ in Figure 2.8.
First step is to initialize random weighting factors for each node interconnection and a bias

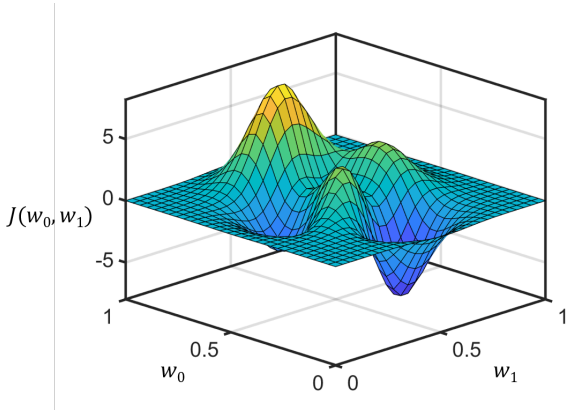


Figure 2.8: Landscape of a Loss Regarding Two Weights

[10]. Then, we feed the network given dataset and produce outputs with respect to randomly assigned weights. Starting from the random point on the landscape, we compute the gradient of the loss function with respect to the weights $\delta J(W)/\delta(W)$. The result of the gradient indicates the steepest ascent. Hence, the direction of the gradient is upwards. Instead of going upwards, we want to take steps in the direction, where the minimum is located. Therefore, we opt for the opposite direction of the gradient. New weights and biases are computed in order to minimize the total error calculated for the initial iteration. We repeat the process until we converge to a minimum [8].
Gradient descent is one of the most common ways to optimize neural networks. There are different options for applying this algorithm. We explain the batch, stochastic and mini-batch gradient descent algorithms. They differ from each other in the amount of data used to compute the gradient of the loss function. A trade-off is made between the time it takes to perform a parameter update and the accuracy of this update [23].
Usually, the gradient descent algorithms are performed by the software frameworks that help us to build deep learning models such as Keras, which is the high-level API of TensorFlow [1]. Therefore, the steps of the gradient descent can be overlooked. Here, we discuss explicitly how different types of gradient descent algorithms work.

- Batch Gradient Descent
  The gradient of the loss function is calculated for the entire dataset. The disadvantage

of the batch gradient descent is that it can be slow since the gradients for the entire dataset needs to be computed. Below, we describe step by step the batch gradient descent [23].

1. Weights are randomly initialized.
2. Until convergence, the following two steps are looped through:
   - Computation of the gradient according to the weights $\frac{\delta J(W)}{\delta(W)}$,
   - Update of the weights with $W \leftarrow W - \eta \frac{\delta J(W)}{\delta(W)}$.
3. Weights are returned.

The gradient computed in step 2 determines how the loss is changing with respect to the weights. Computation of the gradients to minimize the loss is performed by backpropagation. Later in 2.3.7, we examine backpropagation and give an example to demonstrate how the algorithm works.
Weight update occurs by subtracting the current weight from the gradient multiplied by a small value η, which is the learning rate. Learning rate determines how large each step we take is with regard to the gradient. Thus, the gradient indicates the direction we want to take in order to converge and the learning rate indicates the magnitude of change of the parameter set at each iteration. Learning is the process of improving the properties, such as increased convergence rate, decreased settling time, and the functionality of the neural network by adapting the connection weights in the course of time [5]. The speed of the learning process is controlled by the learning rate that is adjustable [8].
Setting a learning rate is a difficult problem. If the learning rate is set to a high number, the learning process will be faster. However, if the learning rate is set to a number that is too high, then weight changes will have oscillations that will prevent convergence to the optimal solution set. Hence, high learning rates can become unstable and diverge. On the contrary, it is possible for the network to converge too slow that it stucks in a local error minimum instead of the global minimum, if the learning rate is too low. Once again, this situation can result in a solution set that is not optimal. In order to obtain the optimal solution set, learning rate can be shifted from a higher number to a lower one during training. This process reduces the probability of settling in a local minimum and speeds up the convergence to the optimal solution set [5, 8, 23].

- Stochastic Gradient Descent
  A variant of the gradient descent is the stochastic gradient descent. The only different step from batch gradient descent is the step 2.
  1. Weights are randomly initialized.
  2. Until convergence, the following steps are looped through:

- Picking a single data point i,
- Computation of the gradient according to the weights $\frac{\delta J_i(W)}{\delta(W)}$,
- Update the weights with $W \leftarrow W - \eta \frac{\delta J(W)}{\delta(W)}$.

3. Weights are returned.

For each training example a parameter update is performed. In contrast to batch gradient descent, stochastic gradient descent performs only one update at a time preventing redundancy. The trade-off here is that we sacrifice the accuracy that we can get from batch gradient descent to obtain lower computation time. However, it is shown that, when we slowly decrease the learning rate while training, we can obtain the same accuracy as the batch gradient descent with a lower computation time [23].

- Mini-Batch Gradient Descent
  Mini-batch gradient descent performs an update for every n training examples instead of performing one as shown in the steps of the algorithm below.
  1. Weights are randomly initialized.
  2. Until convergence, the following steps are looped through:
     - Picking a batch of data points B,
     - Computation of the gradient according to the weights

$$\frac{\delta J(W)}{\delta(W)} = \frac{1}{B} \sum_{k=1}^{B} \frac{\delta J_k(W)}{\delta W},$$

     - Update the weights with $W \leftarrow W - \eta \frac{\delta J(W)}{\delta(W)}$.
  3. Weights are returned.

  The common batch sizes are between 50 and 256. The advantages of mini-batch gradient descent algorithm is that it achieves more stable convergence and it can optimize large-scale matrix operations. This is one of the crucial reasons why we use neural networks for solving linear systems. For large-scale problems $Ax = b$, instead of working with full matrix A and b, we can operate on batches of data. The computation of the gradient is again faster, since we calculate the gradient as the average of the batch rather as all the data points. Additionally, estimations are more accurate too since we work with more than one point each iteration [23].

2.3.7 Backpropagation

Currently backpropagation is the most popular technique to perform supervised learning tasks such as pattern recognition. The algorithm can be applied to neural networks that

represent solutions to nonlinear problems. To backpropagate, at least one hidden layer is necessary and the researches have shown that networks with a single hidden layer are capable of providing accurate approximations of any continuous function [10, 17]. Nevertheless, we also apply a highly simplified version of backpropagation in the iterative solution method because we need to calculate the gradients to update the solution x. The aim of the algorithm is to obtain an accurate network that can perform well on unseen data by adapting the parameters. Therefore, we need to adjust the weights in a way that if decreasing one would cause the error to be higher than it is increased and vice versa. This method is performed on all of the weights in the network and then we start all over again. For one pass through the network the derivatives for all of the weights are calculated using the chain rule. We continue to minimize the error by adjusting the weights until the error and the weights are stabilized [24, 29].
To visualize what backpropagation does we benefit from Figure 2.9. It is a demonstration

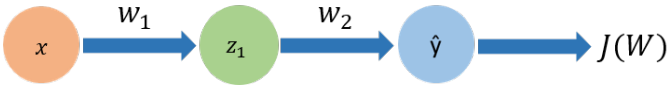


Figure 2.9: Backpropagation

of a highly simplified neural network that contains one input neuron x, one hidden layer existing from only a neuron z_1 and an output neuron $\hat{y}$. To determine, how a small change in one weight w_2 affects the final loss $J(W)$, we compute:

$$\frac{\delta J(W)}{\delta(w_2)} = \frac{\delta J(W)}{\delta \hat{y}} \times \frac{\delta \hat{y}}{\delta w_2}.$$

We apply the chain rule in order to compute the gradient according to w_2. If we want to compute the gradient according to w_1, then we apply the chain rule recursively:

$$\frac{\delta J(W)}{\delta(w_1)} = \frac{\delta J(W)}{\delta \hat{y}} \times \frac{\delta \hat{y}}{\delta z_1} \times \frac{\delta z_1}{\delta w_1},$$

where z_1 is the activation function of the first unit. As the name backpropagation implies, when we compute the gradient with respect to w_1, we take a step back and we include the derivative of the output $\hat{y}$ with respect to z_1 and derivative of z_1 with respect to w_1. To complete backpropagation, the algorithm is repeated for every weight in the network using the gradients from outer layers.

2.4 Neural Networks for Solving Systems of Linear Equations

At this point, we switch from general nonlinear neural network theory to solving linear problems. The reason we discussed neural networks before is that they allow us to represent solutions to (usually: nonlinear) problems, and we will use this property now to

represent solutions to linear systems instead. Additionally, we discuss specific regularization methods for linear systems.

2.4.1 Introduction to the Paper "Neural Networks for Solving Systems of Linear Equations and Related Problems"

The paper "Neural Networks for Solving Systems of Linear Equations and Related Problems" written by Cichocki and Unbehauen in 1992 [5] proposes to build neural networks to solve a linear system iteratively. The underlying ideas of the authors are still valid, even though the specific computational methods are no longer relevant because of the later improvement in machine learning frameworks like TensorFlow.
According to the paper, the linear estimation model is:

$$Ax = b + r = b_{true}.$$

$A = [a_{ij}] \in R^{m \times n}$ is our matrix model, $b \in R^m$ is a vector of observations or measurements, $b_{true} \in R^m$ is a vector of true values and lastly, $r \in R^m$ is an unknown vector of measurement errors. What we search for is an estimation of $x = [x_1, x_2, ..., x_n] \in R^n$.
A hypothetical energy function is built in order to find a vector x, which minimizes the energy function. The energy functions is defined as follows:

$$E(x) = \sum_{i=1}^{m} \sigma_i(r_i(x)).$$

σ_i represents the convex function and several options are examined in the paper, being:

- Ordinary Least Squares Problem: $\sigma_i(r_i) = r_i^2/2$,
- Iteratively Reweighted Least Squares Problem: $\sigma_i(r_i) = (\beta/\alpha)\ln(\cosh(\alpha r_i))$,
- Least Absolute Value Problem: $\sigma_i(r_i) = |r_i|$,
- Chebyshev Problem: $\sigma_i(r_i) = |r_i|$.

Least absolute value and Chebyshev problems have the same convex functions yet their energies are different. One more aspect that we need to know to understand the energy function is the residual vector r defined as:

$$r_i(x) = [r_1(x), r_2(x), ...r_m(x)]^T = Ax - b. \tag{2.6}$$

Each member of the vector r in 2.6 can be calculated with:

$$r_i(x) = a_i^T x - b_i = \sum_{j=1}^{n} a_{ij}x_j - b_i.$$

The paper discusses how well the estimation of x is made using different convex functions in the training process. In the present time, the energy function used in Chichocki and Unbehauen's work corresponds to the loss function.

2.4.2 Energy Functions

We will examine four different energy functions throughout this thesis. Which energy function to use depends on the error distribution in the measurement vector b and on the specific application.

- Ordinary Least Squares Problem: $\sigma_i(r_i) = r_i^2/2$
  The energy function for the problem is defined as:

$$\begin{aligned} E_2(x) &= \frac{1}{2}\sum_{i=1}^{m} r_i^2(x) \\ &= \frac{1}{2} r^T(x) r(x) \\ &= \frac{1}{2}(Ax-b)^T(Ax-b) \\ &= \frac{1}{2}||Ax-b||_2^2. \end{aligned} \tag{2.7}$$

  If the noise in the measurement vector b has a Gaussian distribution, it is optimal to use ordinary least squares criterion. However, assuming the error has a Gaussian distribution is not quite realistic. There could be different sources of errors, such as modeling errors, sampling errors or human errors.
  As mentioned earlier, our goal is to find an x that minimizes the energy function. Therefore, we need to calculate the gradient of the energy function $\nabla E_2(x)$. To compute the gradient, derivatives of the energy function are calculated with respect to each element in x as shown below:

$$\begin{aligned} \nabla E_2(x) &= A^T(Ax-b) \\ &= \left[\frac{\delta E_2(x)}{\delta x_1}, \frac{\delta E_2(x)}{\delta x_2}, \ldots, \frac{\delta E_2(x)}{\delta x_n}\right], \end{aligned} \tag{2.8}$$

  where $x(0) = x^0$. After we compute the gradient with respect to the specified energy function, we scale the gradient by multiplying it with $-\mu(t)$:

$$\frac{dx}{dt} = -\mu(t)\nabla E_2(x). \tag{2.9}$$

  $\mu(t) = [m_{ij}(t)]$ is a $n \times n$ positive definite matrix that is often diagonal. The entries of the matrix are dependent on the time t.
  The learning procedure of the whole system with respect to the ordinary least squares is:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp}\left(\sum_{p=1}^{m} a_{ip}\left(\sum_{k=1}^{n} a_{ik}x_k - b_i\right)\right). \tag{2.10}$$

with $x_j(0) = x_j^0, m \geq n$ (because the equation is assumed to be overdetermined), for $j = 1, 2, ..., n$. x is calculated as a limit point starting from x^0, which can be chosen random at the initial state.
The goal when building a matrix $\mu(t)$ is to provide convergence speed and stability of differential equations. On the one hand, if the μ_{ij} is chosen small, the convergence speed to the desired solution x will be rather slow. On the other hand, if the μ_{ij} is chosen large, it can cause an unstable behaviour of the neural network. Hence, $\mu(t)$ serves here the duty of the learning rate that we discussed in 2.3.6. There are two possibilities deciding on the entries μ_{ij}. Firstly, designating constant entries for the weights. Secondly, selecting the entries adaptively. Adaptive selection can increase the convergence rate without causing stability problems, as opposed to choosing large constants directly.

- Iteratively Reweighted Least Squares Problem: $\sigma_i(r_i) = (\beta/\alpha)\ln(\cosh(\alpha r_i))$
The energy function for the problem is defined as:

$$(x) = E(x, \alpha, \beta) = \frac{\beta}{\alpha} \sum_{i=1}^{m} \ln(\cosh(\alpha r_i), \tag{2.11}$$

where $\alpha > 0, \beta > 0$ are problem-dependent variables.
The criterion is more robust towards the outliers than ordinary least squares criterion. Therefore, we achieve a better quality from a continuous-valued solution when there are spiky noise or outliers in the measurement vector b. What makes this possible is the nonlinear activation function g that we pass the error into:

$$g_i[r_i] = g_i \left[\sum_{k=1}^{n} a_{ik} x_k - b_i \right]. \tag{2.12}$$

g performs the following operations to the error:

$$g_i[r_i] = \frac{\delta \sigma_L(r_i)}{\delta r_i} = \frac{\delta \left[\frac{\beta}{\alpha} \sum \ln(\cosh(\alpha r_i) \right]}{\delta r_i} = \beta \tanh(\alpha r_i). \tag{2.13}$$

Here, the sigmoid nonlinearities are not used with the purpose of binary classification as described in 2.3.4, rather they are used for enabling the network to achieve more robust solutions by preventing the absolute values of residuals from being greater than the cut-off parameter β and by compressing large residuals. If α and β are selected in a way that $\alpha = 1/\beta$, then the iteratively reweighted least squares problem converges to the ordinary least squares problem. In contrast, if $\beta = 1$ and α is chosen large (e.g. 100), then the problem converges to the least absolute value problem.

The learning procedure of the whole system with respect to the iteratively reweighted least squares is:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp} \left(\sum_{p=1}^{m} a_{ip} g \left[\sum_{k=1}^{n} a_{ik} x_k - b_i \right] \right).$$

- Least Absolute Value Problem: $\sigma_i(r_i) = |r_i|$
  The energy function is defined as:

$$E_1(x) = \sum_{i=1}^{m} |r_i(x)|, \tag{2.14}$$

  where the absolute values of the residuals are summed up. Like iteratively reweighted least squares criterion, least absolute value criterion can be used as an alternative when there are outliers in the measurement vector b.
  The learning procedure of the whole system with respect to the least absolute value problem is:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp} \left(\sum_{i=1}^{m} a_{ip} sign \left(-b_i + \sum_{j=1}^{n} a_{ij} x_j \right) \right).$$

- Chebyshev Problem: $\sigma_i(r_i) = |r_i|$
  Chebyshev criterion is formulated as a minimax problem:

$$E(x) = \min_{x \in R^n} \max_{1 \le i \le m} \{|r_i(x)|\}. \tag{2.15}$$

  First the absolute values of the residuals are computed. Then, the maximum along the residuals is selected. If the errors are uniformly distributed, using Chebyshev criterion is suggested. However, it is unlikely that the measurement vector b has uniformly distributed errors.
  The learning procedure of the whole system with respect to the Chebyshev problem is:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp} \left(a_{ip} sign \left(-b_i + \sum_{j=1}^{n} a_{ij} x_j \right) \right).$$

The entire approach with the differential equations can be performed in three essential steps and each step represents a layer in the network demonstrated in Figure 2.10. The architecture of the iterative solution method resembles the perceptron in 2.3.3. The difference between the models is that the network built for the iterative solution method stores x, the solution that we want to estimate, as weights. Also, the network does not process any inputs in contrast to usual neural networks. We have a fixed A and a fixed b.

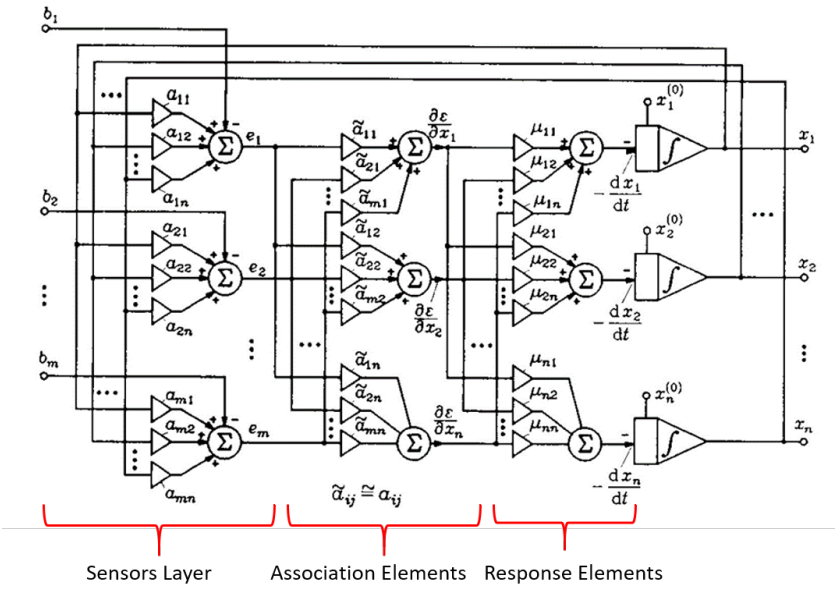


Figure 2.10: Architecture of a Neural Network for Solving a Linear System With Ordinary Least Squares Criterion. Adapted from [5].

1. Computation of the Error
   For all the above mentioned different energy functions, first step is to calculate the error:
   $$e_i(x) = \sum_{k=1}^{n} a_{ik}x_k - b_i, i = 1, 2, ..., m.$$
   Exceptionally, a nonlinearity with g is added to the energy function of iteratively reweighted least squares problem:
   $$e_i(x) = g_i \left[\sum_{k=1}^{n} a_{ik}x_k - b_i \right], i = 1, 2, ..., m.$$
   In the first layer, namely the sensors layer, the error signals are produced as seen in Figure 2.10.

2. Computation of the Gradient
   Second layer consists of association elements that compute the gradient components of the function with the contribution of the first layer. For least squares and iteratively reweighted least squares problems, the gradients are calculated in the same manner, which correspond to:
   $$\frac{\delta\epsilon(x)}{\delta x_p} = \sum_{i=1}^{m} a_{ip}e_i(x), p = 1, 2, ..., n.$$

For least absolute value problem, the gradient is:

$$\frac{\delta\epsilon(x)}{\delta x_p} = \sum_{i=1}^{m} a_{ip} sign\left(-b_i + \sum_{j=1}^{n} a_{ij}x_j\right), p = 1, 2, ..., n. \tag{2.16}$$

Lastly for the Chebyshev problem, the index i corresponds to the maximum element from the residuals vector r, as shown in 2.15. Hence, the gradient is defined as:

$$\frac{\delta\epsilon(x)}{\delta x_p} = a_{ip} sign\left(-b_i + \sum_{j=1}^{n} a_{ij}x_j\right), p = 1, 2, ..., n. \tag{2.17}$$

3. Constitution of the Proper Learning System
   Third and last layer is formed of response elements. With this step, the whole learning process is completed for one iteration. Last step is applied to all the different energy functions in the same manner. The gradient is scaled with $-\mu$:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp} \frac{\delta\epsilon(x)}{\delta x_p}, x_j(0) = x_j^{(0)}, j = 1, 2, ..., n.$$

3 Neural Networks Solving Linear Systems

In this chapter the implementation of neural networks that can solve linear systems are explained in depth. The work in the thesis is mainly based on the paper "Neural Networks for Solving Systems of Linear Equations and Related Problems" [5], where their approach for building neural networks is discussed thoroughly in Section 2.4. We start with the reimplementation of the iterative solution method adopted from the paper. At first, we do not benefit from any software frameworks for deep learning. Afterwards, with the same conditions, such as the same learning rate, same number of iterations/epochs and same loss functions, we implement the same solution method with TensorFlow. TensorFlow is an open source platform that has extensive tools and libraries to train deep neural networks [3]. We utilize these properties while creating our own model. Furthermore, the implementations also differ in the used loss functions. We build the networks using different regularization techniques discussed in 2.4.2.
Later in the thesis, we compare the implementations of both neural networks, with and without using Tensorflow. Additionally, a comparison of networks adopting different loss functions is also included. We use a small-scale problem $Ax = b$, where we selected A as a (3×3) matrix at first, to demonstrate the solution method explicitly. Then, we provide results for a large-scale problem, where we opted for a (10000×10000) matrix A. We also compute the same problems with linear solvers. Furthermore, we discuss the advantages and the disadvantages of using linear solvers for solving linear systems.

3.1 Structure of the Neural Network

Before explaining the implementation of the iterative solution method, we give a general overview at the structure of the network. In this section, we examine the layering of the network and the use of activation functions in advance by taking the structure of nonlinear neural networks into consideration.

3.1.1 Layers

As opposed to the nonlinear neural networks, our network does not represent nonlinear mappings. In Figure 2.10 from the last Section 2.4, the linear model for the iterative solution method is demonstrated. According to Cichocki and Unbehauen, the model consists of three layers and in each layer different computations for building the learning system take place. Thus, the logic of layering is different from today as we discussed in 2.3.1.

The mentioned three layers correspond to one linear layer at the present time because in our implementation neither we use any other hidden layers nor we have separate input and output layers. Our network does not process any inputs rather it stores the solution x in one single linear layer as weights. The reason why they divide the network into three different layers is to distinguish different calculation steps from each other.

3.1.2 Activation Function

In our network we do not use any activation functions like the usual nonlinear neural networks discussed in 2.3.4. The only time we apply a nonlinear activation function g is when we opt for the iteratively reweighted least squares problem. Nevertheless, this activation function is not applied to the network directly like the nonlinear neural networks, where at each layer the multiplication of weights and inputs are passed into it as described in 2.3.3. This activation function is a part of the loss function. Hence, we do not need to implement the activation function explicitly for the training process.

3.2 Implementation of the Neural Networks

3.2.1 Libraries

For the implementation of the iterative solution method we used Python. Thus, we need to import several Python libraries that are demonstrated in 3.1. We benefit from NumPy to perform linear algebra operations. Moreover, we need to have a library for visualization of our results and Matplotlib fulfill the need. Since we implement two types of neural networks, being a neural network that we built only with numerical computations and a network built with TensorFlow, we import the TensorFlow package as well.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import tensorflow as tf
```

Code Listing 3.1: Imported Libraries for the Implementation of the Neural Networks

3.2.2 Training Samples

The theory behind Cichocki and Unbehauen's paper is to solve a linear system $Ax = b$ iteratively in order to converge to a solution x as close as possible to the actual solution. Normally, as we examine in 2.3.2, neural networks are trained with datasets that have numerous information. The goal is to obtain a network with trained weights that can predict accurate results on unseen data. In our implementation, we adopt a different solution method. We feed our network a fixed matrix A and a fixed matrix b instead of feeding the

network various matrices A and b. To be concrete, we work with an example to demonstrate the approach better:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix}, b = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}.$$

In this case, the actual x that we want to converge after the training is:

$$x\_actual = \begin{bmatrix} 5 \\ 3 \\ -2 \end{bmatrix}.$$

However, x is unknown at the start. The problem can be defined as:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}.$$

The goal of our network is to find an approximation for x_1, x_2, and x_3. We do not use any testing samples, since the responsibility of our network is just to find an approximation x for this specific problem. In other words, our network is specialized for solving only one problem.

3.2.3 Training Procedure

- Neural Network Implemented Without TensorFlow

```
x = np.array([0, 0, 0])
energy_function = 0
lstsq_energies = []

for i in range(1000):

    error_vector = A @ x - b

    energy_function = (1/2) * error_vector.T @ error_vector
    lstsq_energies.append(energy_function)

    gradient_vector = A.T @ error_vector

    scaled_gradient = -m @ gradient_vector

    x_new = x + scaled_gradient
    x = x_new
```

Code Listing 3.2: Neural Network Implemented Without TensorFlow Using Ordinary Least Squares Loss

Above, a code excerpt 3.2 is provided to give a better understanding of the training procedure. We firstly initialize a random x and we opted for a vector that consists of zeros:

$$x = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.1}$$

We implement the approach with three essential steps mentioned in Section 2.4. At first, error r is calculated with $Ax - b$ in line 7. Then, the energy function using the error r is computed. The energy function differs for each regularization technique. In code excerpt 3.2, the energy function represents the ordinary least squares problem. We append the computed energy function to a list because after the training, we want to plot the loss and observe its behaviour.
Next step is to calculate the gradient of the energy function with respect to the x vector in line 12. The reason is to detect the rate of change in the energy function to the change in each $x_i = [x_1, x_2, ..., x_n]$. The process is a simplified version of backpropagation described in 2.3.7. Here, our weights are the elements of the x vector that we want to estimate. Therefore, the gradient is computed with respect to x. We also do not chain derivatives as we move towards outer layers because we only have one layer in our model.
For μ we designate a matrix defined as:

$$\mu = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 0.01 \end{bmatrix}. \tag{3.2}$$

We opt for entries 0.01 for the diagonal. Thus, the matrix μ scales the gradient instead of updating x with the full amount like a learning rate described in 2.3.6. After computing the step size for each x_i, we need to integrate the gradient in order to obtain the updated x. We perform integration by using the rectangle method. For each iteration, the scaled gradient is added to x in order to achieve the area of the rectangle below the function as in line 16. The updated x will be the new input for the next iteration. We iterate all the steps over a 1000 times.

- Neural Network Implemented With TensorFlow

```
1 n_dim = 3
2 n_training_steps_per_epoch = 1
3 x_train = np.zeros((n_training_steps_per_epoch,n_dim))
4 NET_LEARNING_RATE = 1e-2
5 NET_EPOCHS = 1000
6 NET_BATCH_SIZE = 1
```

Code Listing 3.3: Extra Parameters for the TensorFlow Implementation

For the implementation with TensorFlow we have additional variables as shown above in 3.3 that we will utilize in the training process. We set the dimension of the problem to 3, since we have a 3×3 matrix A. Moreover, we use a training variable `x_train` to define the number of training steps per epoch and the dimension of the problem. The number of training steps per epoch is set to 1 and as mentioned, the dimension is set to 3, which stands for the number of rows and number of columns in `x_train`.
Our network performs the stochastic gradient descent algorithm described in 2.3.6. As we already mentioned above, we set 1 for the `n_training_steps_per_epoch` and 1 for the `NET_BATCH_SIZE` because in each iteration our goal is to accomplish one pass through the operations. The reason why we do this is to have the same plot as the implementation without using Tensorflow at the end. If we set the number of training steps per epoch to a larger number, for instance to 1000, and the batch size to 256, then we can converge faster to the solution. Note that with those parameters, stochastic gradient descent acts as mini-batch gradient descent.

```
class LinearSystemSolution(tf.keras.Model):
    def __init__(self, n_input_dimension, A, b, **kwargs):
        super(LinearSystemSolution, self).__init__(**kwargs)
        self.solution = tf.Variable(initial_value=tf.zeros((
                            n_input_dimension,1), dtype=tf.float64),
                                    trainable= True ,
                                    dtype=tf.float64,
                                    shape=(n_input_dimension,1))
    self.A = tf.convert_to_tensor(A)
    self.b = tf.convert_to_tensor(b)

def call(self, x, training=True):
        error = tf.matmul(self.A, self.solution) - self.b
        self.add_loss(tf.reduce_sum(tf.square(error))/2)
    return self.solution

network = LinearSystemSolution(n_dim, A, b)
network.compile(optimizer=tf.optimizers.SGD(learning_rate=
                NET_LEARNING_RATE))

callbacks = [LossAndErrorPrintingCallback()]
lstsq_historian = network.fit(x_train, epochs=NET_EPOCHS,
                verbose=0, batch_size=NET_BATCH_SIZE, callbacks=callbacks)
```

Code Listing 3.4: Neural Network Implemented With Tensorflow Using Ordinary Least Squares Loss

Above, there is a code excerpt 3.4 that demonstrates the implementation of a neural network using TensorFlow. The adopted loss function is again the ordinary least squares problem.
According to the user guide of Keras [1], `tf.keras.Model()` groups layers into an

object with training and inference features. To define our custom single layer and its attributes, we use the constructor `__init__()`. We then pass the input dimension, A, and b described in 3.2.2 to the constructor. To define our solution x in line 4, we use the constructor `Variable()`, which has the arguments shown in 3.5 below.

```
tf.Variable(
    initial_value=None, trainable=None, validate_shape=True,
    caching_device=None, name=None, variable_def=None, dtype=None,
    import_scope=None, constraint=None,
    synchronization=tf.VariableSynchronization.AUTO,
    aggregation=tf.compat.v1.VariableAggregation.NONE, shape=None
)
```

Code Listing 3.5: Arguments of the Variable() Constructor

For building our network, we use the arguments `initial_value`, `trainable`, `dtype`, and `shape`. In our example, `initial_value` is a matrix that consists of zeros with the shape (3×1), since we search for the solution $x \in R^3$. We seek to have a trainable variable because in each iteration x will be updated. Hence, we set the boolean `trainable` to `true`. For numeric stability, we set the default floating-point `dtype` to `float64`. `shape` defines the shape of our solution x, being (3×1). After setting the parameters of the `Variable()`, we convert the matrices A and b from Python objects to Tensor objects in lines 9 and 10.
The `call()` method defines the computation from input to output. At first, the error is calculated with $Ax - b$ in line 13. Then, we use the `add_loss()` method that adds an externally defined loss to the collection of losses. We define the loss function according to the ordinary least squares problem [1].
We create our network in line 17 and then we compile it. To configure our model, we use the method `compile()`. As an optimizer we select the stochastic gradient descent, which is discussed in 2.3.6. After the model is configured, it is ready for training. Method `fit()` trains the model for a fixed number of epochs. The arguments of the method are shown below 3.6.

```
fit(
    x=None, y=None, batch_size=None, epochs=1, verbose=1,
    callbacks=None, validation_split=0.0, validation_data=None,
    shuffle=True, class_weight=None, sample_weight=None,
    initial_epoch=0, steps_per_epoch=None, validation_steps=None,
    validation_batch_size=None, validation_freq=1, max_queue_size=10,
    workers=1, use_multiprocessing=False
)
```

Code Listing 3.6: Arguments of the fit() Method

`x` is the input data and `y` is the target data. Essentially, our network does not process any inputs, it just stores the solution x as a TensorFlow variable. Thus, `x_train` in line 22 is used only for defining the number of training steps per epoch and the

dimension of the problem.
Learning rate serves the same purpose with the matrix μ defined in 3.2. Thus, we set it to 0.01 to create the same environment for both implementations. The number of epochs is correspondingly set to a 1000 as demonstrated in 3.3. Lastly, `verbose` is set to 0. It can have the values 0 or 1. If `verbose = 0`, there is no progress bar as an output and for `verbose = 1` vice versa [1].

3.3 Implementation of Different Loss Functions

The different energy functions that help us to compute loss are discussed in Section 2.4. The described energy functions are used as loss functions in our model. In this section, we elaborate the implementations of the different loss functions for both neural networks that are built with and without using TensorFlow.

3.3.1 Ordinary Least Squares Problem

- Neural Network Implemented Without TensorFlow
  The code excerpt 3.2 demonstrates the implementation of a network that adopts ordinary least squares problem as a loss function. At the beginning of the iteration, the `error_vector` is calculated and passed to the `energy_function`, which is computed with respect to the ordinary least squares problem described in 2.7. Thus, we take the square of the error vector and divide it by two. We calculate the `gradient_vector` with regard to 2.8 by multiplying the transpose of the matrix A with the `error_vector`.

- Neural Network Implemented With TensorFlow
  The full implementation of the neural network using TensorFlow can be seen in 3.4. In the `call()` function, we specify the loss that we want to use during the training process of the network. Therefore, in line 14 we define the ordinary least squares problem. The square of the error is computed with `tf.square()` and `tf.reduce_sum()` sums all the elements. Hence, `tf.reduce_sum()` has the same responsibility as the $\sum$ operator in this situation. Lastly, we divide the result by two to obtain the corresponding loss function.

3.3.2 Iteratively Reweighted Least Squares

- Neural Network Implemented Without TensorFlow
  We introduce two variables before training, namely α and β. As discussed in 2.4.2, α and β symbolizes different shapes of sigmoid nonlinearities. In our example, we selected $\alpha = 0.1$ and $\beta = 10$ so that the iteratively reweighted least squares problem converges to the ordinary least squares problem.

```
for i in range(1000):

error_vector = A @ x - b

sum_total = 0
for j in error_vector:
    res = np.log(np.cosh(alpha*(j)))
    sum_total = res + sum_total

energy_function = beta/alpha * sum_total
iter_energies.append(energy_function)

g = beta * np.tanh(alpha * error_vector)

gradient_vector = A.T @ g

scaled_gradient = -m @ gradient_vector

x_new = x + scaled_gradient
x = x_new
```

Code Listing 3.7: Neural Network Implemented Without TensorFlow Using Iteratively Reweighted Least Squares Loss

Code excerpt 3.7 demonstrates the training procedure of the network with the loss function iteratively reweighted least squares. To implement the energy function defined in 2.11, we iterate through the elements in `error_vector` and apply the operations in line 7. Before, we append the computed energy to the list, we multiply it with β/α.
A nonlinearity with activation function g is added to the problem and in line 13 and we define g as in 2.13. The activation function g contains $\tanh$ activation function that we defined in 2.3.4. `gradient_vector` is calculated in the same manner as the `gradient_vector` from the least squares problem.

- Neural Network Implemented With TensorFlow
  We use the same implementation as the ordinary least squares problem defined in 3.4 for the other loss functions. The only part that needs to be modified is inside the `call()` function, where we define our custom loss functions. Below in code excerpt 3.8, iteratively reweighted least squares problem is implemented with respect to its energy function 2.11.

```
def call(self, x, training=True):
    error = tf.matmul(self.A, self.solution) - self.b
    self.add_loss(tf.reduce_sum(tf.math.log(tf.math.cosh
                        (alpha*(error)))) * (beta/alpha))
    return self.solution
```

Code Listing 3.8: Iteratively Reweighted Least Squares Loss Defined in call()

3.3.3 Least Absolute Value Problem

- Neural Network Implemented Without TensorFlow

```
for i in range(1000):

error_vector = A @ x - b

sum_total = 0
for j in error_vector:
    sum_total = abs(j) + sum_total

energy_function = sum_total
abs_energies.append(energy_function)

sign_0 = np.sign(-b[0] + A[0] @ x)
sign_1 = np.sign(-b[1] + A[1] @ x)
sign_2 = np.sign(-b[2] + A[2] @ x)

sign_vector = np.array([sign_0, sign_1, sign_2])

der_x0 = A[:, 0] @ sign_vector
der_x1 = A[:, 1] @ sign_vector
der_x2 = A[:, 2] @ sign_vector

gradient_vector = np.array([der_x0, der_x1, der_x2]).T

scaled_gradient = -m @ gradient_vector

x_new = x + scaled_gradient
x = x_new
```

Code Listing 3.9: Neural Network Implemented Without Tensorflow Using Least Absolute Value Loss

Code excerpt 3.9 demonstrates the implementation of a network that adopts the least absolute value problem as a loss function. To compute the energy function, we sum up the absolute values of the elements of the `error_vector` in line 7. We calculate the `gradient_vector` with respect to the computation in 2.16. At first, we compute the signs of the term $-b_i + \sum_{j=1}^{n} a_{ij}x_j$ starting at line 12 for each element of the `error_vector`. Then we join them together in the `sign_vector`. With respect to x_0, x_1 and x_2, we compute the derivatives of the energy function. For instance, when we differentiate the energy function according to x_0, every other x becomes a constant. Therefore, we take all the coefficients of x_0 from the first column of the matrix A and we multiply it with the `sign_vector`. After we define derivatives with respect to each x, we create the `gradient_vector` by combining them together.

- Neural Network Implemented With TensorFlow

```
def call(self, x, training=True):
    error = tf.matmul(self.A, self.solution) - self.b
    self.add_loss(tf.reduce_sum(tf.abs(error)))
    return self.solution
```

Code Listing 3.10: Least Absolute Value Loss Defined in call()

Code excerpt 3.10 is the implementation of the least absolute value problem in `call()` function. According to the energy function of the least absolute value problem defined in 2.14, we take the absolute value of the error with the help of `tf.abs()`. Then, with `reduce_sum()` we sum up the absolute values of the elements in the residual vector.

3.3.4 Chebyshev Problem

- Neural Network Implemented Without TensorFlow

```
for i in range(1000):

error_vector = A @ x - b

max_val = max(abs(error_vector))

energy_function = max_val
cheb_energies.append(energy_function)

sign_0 = np.sign(-b[0] + A[0] @ x)
sign_1 = np.sign(-b[1] + A[1] @ x)
sign_2 = np.sign(-b[2] + A[2] @ x)

if max_val == abs(error_vector[0]):
    gradient_vector = np.array([A[0][0]*sign_0, A[0][1]*sign_0, A[0][2]*
    sign_0]).T
elif max_val == abs(error_vector[1]):
    gradient_vector = np.array([A[1][0]*sign_1, A[1][1]*sign_1, A[1][2]*
    sign_1]).T
else:
    gradient_vector = np.array([A[2][0]*sign_2, A[2][1]*sign_2, A[2][2]*
    sign_2]).T

scaled_gradient = -m @ gradient_vector

x_new = x + scaled_gradient
x = x_new
```

Code Listing 3.11: Neural Network Implemented Without Tensorflow Using Chebyshev Loss

As opposed to the least absolute value problem, we take absolute values of the residuals and select the maximum value as defined in 2.15. To compute the gradient according to 2.17, we follow the same steps as the implementation of the least absolute value problem at start. After computing the signs for each element in the residual vector, starting from line 14 we search for the element that has the maximum absolute value. For instance, if it is the first element of the `error_vector`, then the gradient is composed of the derivatives from the first row of the matrix A with respect to x_0, x_1 and x_2. To be more concrete, $r_0 = a_{00} * x_0 + a_{01} * x_1 + a_{02} * x_2 - b_0$ is the first element of the `error_vector`. If r_0 has the highest absolute value, our `gradient_vector` will be:

$$\begin{bmatrix} a_{00} * sign_0 \\ a_{01} * sign_0 \\ a_{02} * sign_0 \end{bmatrix}.$$

- Neural Network Implemented With TensorFlow

```
1 def call(self, x, training=True):
2     error = tf.matmul(self.A, self.solution) - self.b
3     self.add_loss(tf.reduce_max(tf.abs(error)))
4     return self.solution
```

Code Listing 3.12: Chebyshev Loss Defined in call()

As seen in code excerpt 3.12, we first take the absolute value of the error. In order to obtain the maximum along the elements, we use the method `tf.reduce_max()`.

3.4 Evaluation of Loss

In this section we demonstrate for each problem the behaviour of the loss. We benefit from plots to give a deeper understanding. At the end of the section we compare each loss that we obtained from using four different energy functions.

3.4.1 Ordinary Least Squares Problem

- Neural Network Implemented Without Using TensorFlow

```
1 lstsq_loss_without_tf = lstsq_energies[999]
2 print("Loss without using tensorflow is ", lstsq_loss_without_tf)
```

Code Listing 3.13: Computation of Ordinary Least Squares Loss Without Using TensorFlow

After we trained the network with the implementation demonstrated in 3.2 over a 1000 times, the list `lstsq_energies` has collected an energy for each iteration. We compute the last element of the energy list in order to evaluate the loss of our model at the end. The loss of the network we built by adopting ordinary least squares problem is approximately 0.00015, which is 10^{-4}.

- Neural Network Implemented With TensorFlow

```
lstsq_loss_with_tf = (lstsq_historian.history["loss"][999])
print("Loss using tensorflow is ", lstsq_loss_with_tf)
```

Code Listing 3.14: Computation of Ordinary Least Squares Loss Using TensorFlow

To compute the loss of the network built with the help of TensorFlow, we benefit from the `History` object as demonstrated in 3.14. The `network.fit()` method used in the code snippet 3.4 returns a `History` object. Furthermore, `History.history` attribute holds a recording of training loss values and metrics values at successive epochs [1].
As expected, the output is approximately 0.00015, which is the same output that we obtained from the implementation using only numerical operations. We used the same parameters for both neural networks to achieve the same results. Our main goal by adapting the iterative solution method to the TensorFlow environment is to benefit from its library. TensorFlow offers us some advantages, such as more practical implementation and faster convergence to the solution.

- Plotting the Loss

```
fig,ax = plt.subplots(1,1,figsize=(5,5))
ax.semilogy(lstsq_historian.history["loss"], "x",
                     label = "with tf", markevery= 5)
ax.semilogy(lstsq_energies, label = "without tf")
ax.set_title("Loss: Ordinary Least Squares Problem")
ax.set_xlabel("Iterations")
ax.set_ylabel("Loss")
ax.legend()
```

Code Listing 3.15: Plotting Ordinary Least Squares Loss

To plot the losses for both neural networks, we use code excerpt 3.15. Corresponding plot is demonstrated in Figure 3.1. Both energy functions are minimized in the same manner. For the neural network built with TensorFlow, there are several ways to improve the minimization process. For instance, training steps per epoch can be incremented. Hence, in one epoch x will be updated several times and we can converge faster to the solution than the neural network built without TensorFlow.

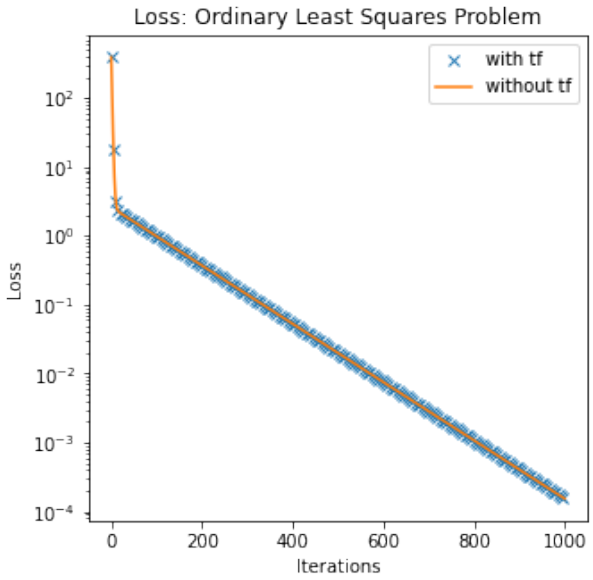


Figure 3.1: Ordinary Least Squares Loss With and Without Using TensorFlow

- Convergence to the Actual Solution
  Lastly, we evaluate the estimated x to see how well the networks have converged to the actual x.

```
print("network solution without using tf is", x)
print("network solution using tf is", network.solution.numpy().T)
```

Code Listing 3.16: Network Solutions Using Ordinary Least Squares Loss

The results of 3.16 are again approximately the same and the solution of the networks is:

$$\begin{bmatrix} 4.9769 \\ 3.0092 \\ -2.0030 \end{bmatrix}. \tag{3.3}$$

Both networks converged to the actual solution defined in 3.2.2 successfully with minor errors.

3.4.2 Iteratively Reweighted Least Squares Problem

The results are achieved in the same way for all the different energy functions. Therefore, from this point we provide only the outputs. For iteratively reweighted least squares, we selected $\alpha = 0.1$ and $\beta = 10$ so that the problem converges to the ordinary least squares problem. Thus, the loss we get is approximately the same as the ordinary least squares problem, being 0.00015. Again, we observe the same behaviour of the loss from both

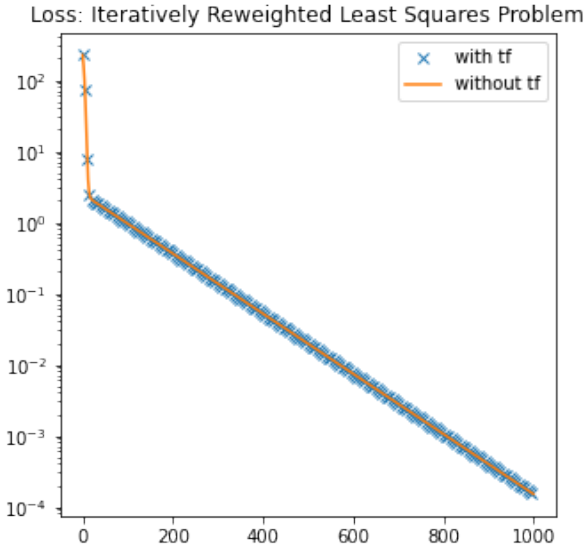


Figure 3.2: Iteratively Reweighted Least Squares Loss With and Without Using TensorFlow, $\alpha = 0.1$ and $\beta = 10$

neural networks using iteratively reweighted least squares problem as their loss function. The convergence to the actual x is as follows:

$$\begin{bmatrix} 4.9772 \\ 3.0091 \\ -2.0029 \end{bmatrix}.$$

We achieved nearly the same values for the solution x as we achieved in 3.3.

3.4.3 Least Absolute Value Problem

The loss we get from the neural network implemented without TensorFlow that adopts the least absolute value problem is 0.56000. From the neural network implemented with TensorFlow, we obtain a loss that is approximately 0.44000. As seen in Figure 3.3 there is chattering in the loss. Therefore, the obtained losses from the last iteration are different. For instance, if we examine the neural network implemented without using TensorFlow and compute the 998th energy, it is 0.42000. Nevertheless, the computed loss is higher than the least squares problems.
The network solutions without and with using TensorFlow in advance are as follows:

$$\begin{bmatrix} 4.84 \\ 3.02 \\ -2.02 \end{bmatrix}, \begin{bmatrix} 4.90 \\ 3.08 \\ -1.96 \end{bmatrix}.$$

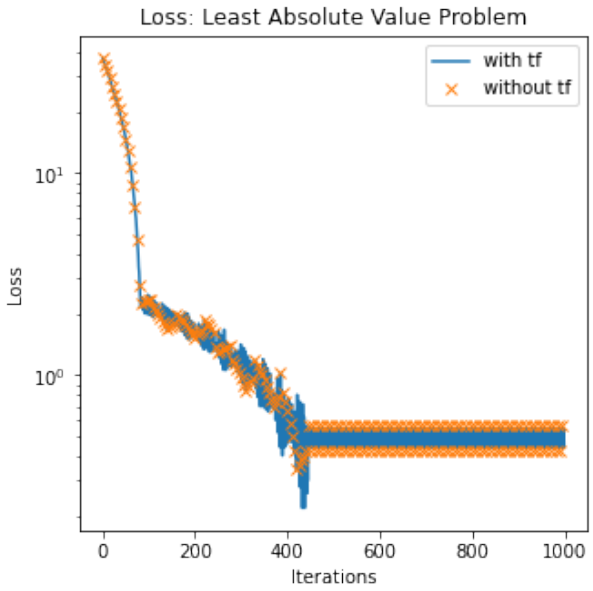


Figure 3.3: Least Absolute Value Loss With and Without Using TensorFlow

Hence, the network solutions have higher distance to the actual solution in contrast to least squares problems.

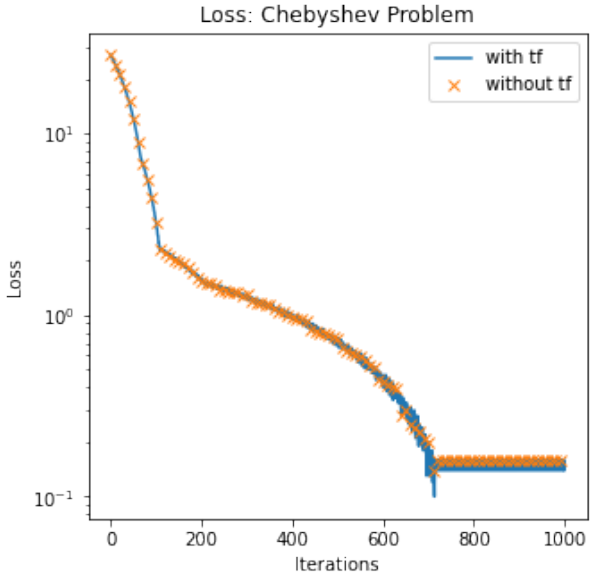


Figure 3.4: Chebyshev Loss With and Without Using TensorFlow

3.4.4 Chebyshev Problem

The loss we obtain using the Chebyshev problem, demonstrated in Figure 3.4, is 0.14000 for both neural networks. In comparison to the least absolute value problem, for this specific example we achieved a lower loss with Chebyshev. However, the loss is still higher than the least squares problems. The network solutions are the same for both implementations and it is:

$$\begin{bmatrix} 4.87 \\ 3.08 \\ -2.02 \end{bmatrix}.$$

Computed solution is near to the solution that we obtained from using the least absolute value problem. Nevertheless, the convergence is weaker than the solutions from using least squares problems.

3.4.5 Results

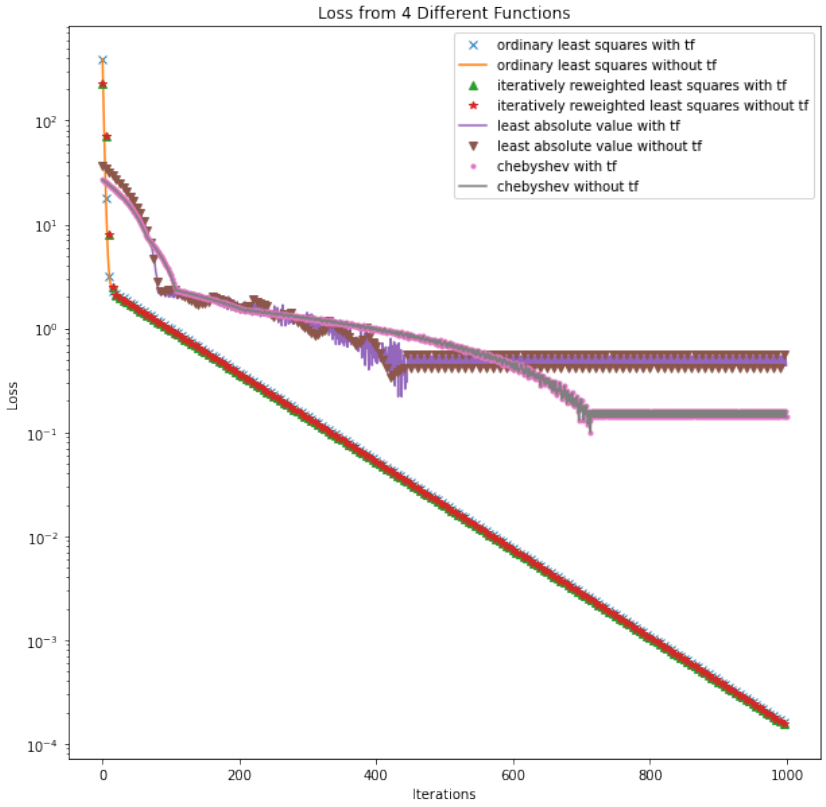


Figure 3.5: Comparison of 4 Different Energy Functions

We compare all the losses that we obtained from using different energy functions in Figure 3.5. Ordinary least squares and iteratively reweighted least squares functions behaved in the same manner. That is because we selected α and β as $\alpha = 1/\beta$. Additionally, it can be seen that the least squares problems achieved the lowest loss. Note that, for our small-scale example we used a measurement vector b that does not consist of any errors. There are no outliers in the data that may shift the least squares functions towards the error. Between the least absolute value and Chebyshev problems, we achieved a lower loss using the Chebyshev problem. As we described the mean absolute value error loss in 2.3.5, the energy function least absolute value can have convergence problems. Therefore, we have oscillations in Figure 3.5 when we compute the loss with the least absolute value problem.

3.5 Computation with Linear Solvers

We computed a solution x for our training sample A and b using the `solve()` method in 3.17.

```
1 x_standard = np.linalg.solve(A, b)
```

Code Listing 3.17: Computation of x With solve()

The reason we use the `solve()` method is that our A is square and has full rank. We could use the least squares method `lstsq()` as well. As shown in code excerpt 3.17, a single line is enough to solve a linear equation. For our small-scale problem, `solve()` method computes a solution `x_standard` same as the actual solution x defined in 3.2.2. Out of all the neural networks that adopted different loss functions, we obtained the best solution from the linear solver.
The simplicity of implementing a linear solver and the accuracy of the result compared to neural networks are noticeable. Both neural network and linear solver have the same goal, to solve $Ax = b$. Then, we must have some advantages over linear solvers when we solve linear systems with neural networks. The advantage of neural networks is that they can handle large-scale matrix operations. In contrast, linear solvers can not compute results for large-scale problems on our laptops because there is not enough memory to perform operations on such large matrices. Even though implementing a neural network is complicated, it offers a functionality that a direct linear solver is not capable of. To conclude, using linear solvers is feasible when we have a small-scale problem. In next Section 3.6, we will demonstrate an example, where using linear solvers does not work due to the size of the matrices.

3.6 Computation of a Large-Scale Linear System

For our large-scale problem, we create A and x randomly as normally distributed matrices. A has the shape (10000×10000) and x has the shape (10000×1). We compute b by

multiplying A and x. When we work with such large matrices, the sum of the elements in each row gets very large and numerical overflow can occur. To prevent any issues, we rescale A and b after sampling them. We accomplish rescaling by dividing both A and b by 10000. Since A and b are divided simultaneously with the same number:

$$(\frac{1}{10000} \times A)x = (\frac{1}{10000} \times b),$$

x stays the same. Figure 3.6 demonstrates the loss using ordinary least squares problem.

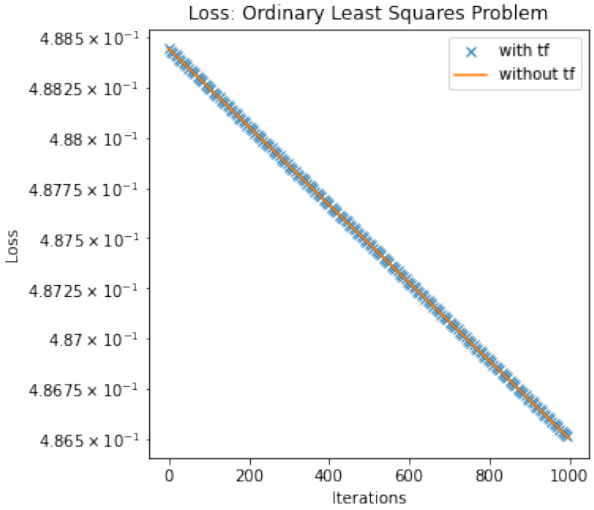


Figure 3.6: Ordinary Least Squares Loss With and Without Using TensorFlow

According to the loss, our network built without the help of TensorFlow performed the same as the network built with TensorFlow. The main difference between both implementations is the computation time. Using ordinary least squares loss, we achieved a solution in 141 seconds with TensorFlow, which is approximately two and a half minutes. The computation time of the implementation without TensorFlow is 357 seconds, which is approximately six minutes. As a result, with the help of TensorFlow we obtained the estimation of x more than twice as fast. Therefore, we continued to work with TensorFlow for this problem. Computation times for the other loss functions are between 141 and 143 seconds.
We compute the norm of the error as shown in 3.18 in order to check how well the network solutions are. These solutions belong to the networks that adopted ordinary least squares loss.

```
1 print(np.linalg.norm(A @ network.solution.numpy() - b))
2 print(np.linalg.norm(A @ x - b))
```

Code Listing 3.18: Computation of the Error According to the Ordinary Least Squares Problem

As a result we obtained 0.9864 for both implementations with and without using TensorFlow.

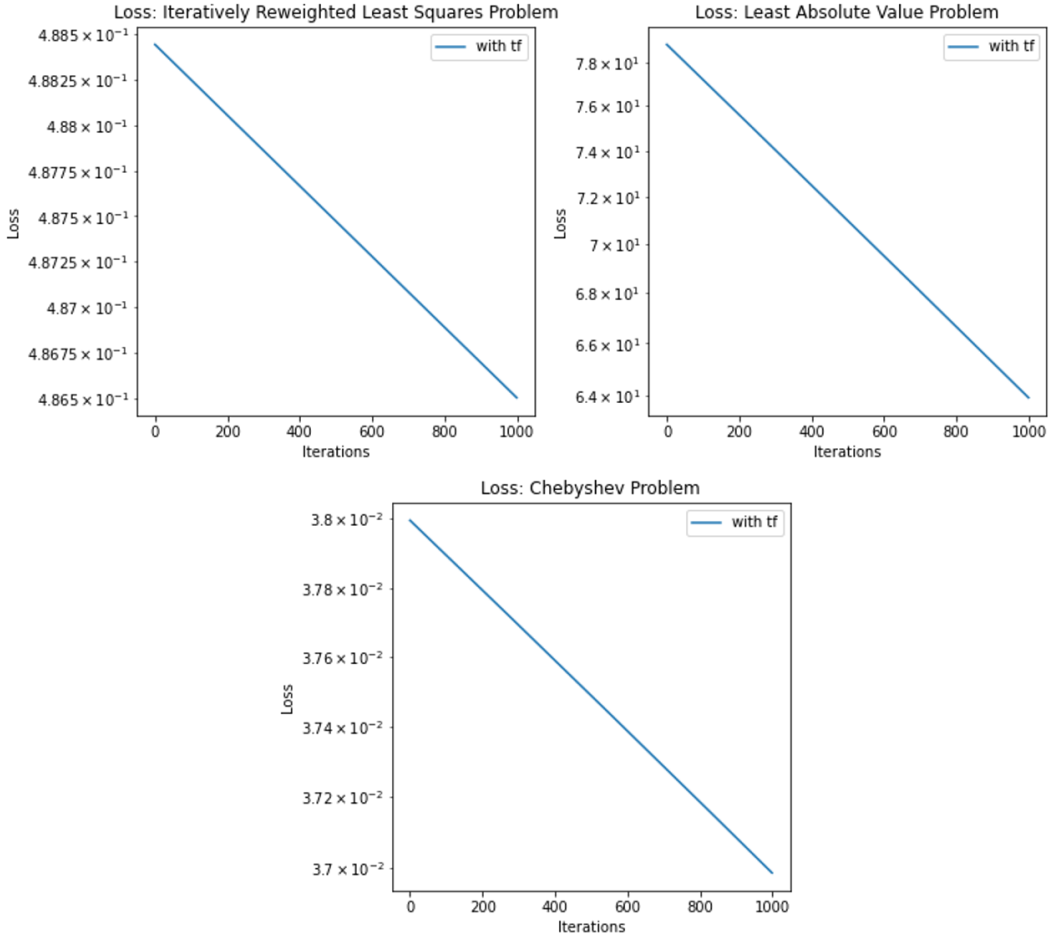


Figure 3.7: Different Losses Computed Using TensorFlow

In Figure 3.7 different losses computed with TensorFlow are demonstrated. Since the starting points for each loss are different, we compute the norm of the error with the same method we used in 3.18 in order to determine which loss function performed the best for our large-scale problem. The results are as follows:

- Iteratively Reweighted Least Squares Problem ($\alpha = 1$, $\beta = 10$) = 0.9864,
- Least Absolute Value Problem = 0.8413,
- Chebyshev Problem = 0.9883.

According to the results, we achieved the most accurate solution x with the least absolute value problem. As we discussed in 2.3.5, the least absolute value is more robust to the outliers than least squares problems. Hence, the errors (e.g. rounding errors) are handled better and the estimated x is closer to the actual solution. Least squares problems performed better than Chebyshev nevertheless worse than the least absolute value problem.

4 Conclusion

4.1 Summary

Neural networks have compatible features to work with matrix operations. In this thesis we proposed an iterative method to solve a linear system. We first examined the nonlinear neural networks in depth and then we adopted some of the characteristics of these neural networks to the models that we built. The reason why we implemented an iterative solution method instead of using pre-existing linear solvers is that we wanted to obtain a system that can work with large matrices. Linear solvers are easy to implement and they return more accurate results for small-scale problems $Ax = b$. To demonstrate how our network operates, we used a small-scale problem with a matrix $A \in \mathbb{R}^{3\times3}$ and a vector $b \in \mathbb{R}^3$ in 3.2.2. For the demonstrated problem, we obtained a solution x same as the actual solution with the `solve()` method. Hence, we achieved a better result from a linear solver than the networks that we built to solve a linear system. Nevertheless, in Section 3.6, we showed a large-scale problem, where we used an $A \in \mathbb{R}^{10000\times10000}$ and a $b \in \mathbb{R}^{10000}$, and we achieved an estimation of x that we could not achieve with linear solvers. The NumPy linear algebra functions `solve()` and `lstsq()` could not compute a result due to the large-scale numerical operations that had to be done to estimate a x.
We benefited from four different energy functions as we built the networks. These energy functions are used as loss functions and each of them has its advantages in different scenarios. For instance, ordinary least squares function performed the best on the small-scale problem because the measurement vector b did not contain any outliers in it. For the large-scale problem, the least absolute value function returned the most accurate result because it is more robust to the errors in the data. The errors might have occured in the sampling and rescaling process while rounding the floating-point numbers. With the iterative solution method we have the advantage of selecting a loss function to compute x. For instance, if it is thought that b might have some outliers in it, then we can opt for iteratively reweighted least squares or least absolute value problems. We also have the opportunity to define a more complex loss function to solve a linear system.
We computed the results for both networks implemented with and without using TensorFlow. Tensorflow has some features that ease the process of building a neural network. To define networks that adopt different loss functions without the help of TensorFlow, we need to start from the very beginning because implementing each loss requires different computations (e.g. computation of the gradients). In contrast, in the implementation with TensorFlow the only piece of code that needs to be modified is inside of the `call()` function, where we specify the loss functions. This feature of TensorFlow makes it practical to

solve linear systems with the desired loss function. Another crucial advantage of Tensorflow is the faster computation time. With Tensorflow it takes approximately twice as fast to compute a solution than the implementation without using TensorFlow for the large-scale problem.
To conclude, we implemented a method to solve a linear system iteratively that has advantages over linear solvers. According to the results we also showed that using TensorFlow is beneficial to build the iterative solution method. While building the networks, we utilized different loss functions and we observed the behaviours of these loss functions in order to determine when to use them.

4.2 Future Work

In this thesis we provided an example of a linear system that did not contain any errors in the measurement vector b. For future work, with the use of different loss functions, linear systems that contain different types of errors can be computed with the iterative solution method. Thus, the losses after the computations can be observed in order to determine which loss function performs the best according to different types of errors. Further, new and more complex loss functions can be introduced to our model.
Lastly, as we discussed in Section 3.6 the time it takes to compute a solution with TensorFlow is approximately two and a half minutes for the large-scale problem. New methods can be introduced, to shorten the computation time of our model. Further, the iterative solution method can be improved in order to work with larger matrices than 10000×10000.

Bibliography

[1] Keras api reference. https://keras.io/api/. Accessed: 2021-04-08.

[2] Linear algebra (numpy.linalg). https://numpy.org/doc/stable/reference/routines.linalg.html. Accessed: 2021-04-06.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Beifang Chen. Systems of linear equations.

[5] A. Cichocki and R. Unbehauen. Neural networks for solving systems of linear equations and related problems. *IEEE Transactions and Circuits and Systems I: Fundemental Theory and Applications*, 1992.

[6] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[7] Bin Ding, Huimin Qian, and Jun Zhou. Activation functions and their characteristics in deep neural networks. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 1836–1841. IEEE, 2018.

[8] Randall J Erb. Introduction to backpropagation neural network computation. *Pharmaceutical research*, 10(2):165–170, 1993.

[9] 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*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[10] Anthony TC Goh. Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering*, 9(3):143–151, 1995.

[11] Gene H Golub. Some modified matrix eigenvalue problems. *Siam Review*, 15(2):318–334, 1973.

[12] Gene H Golub and Charles F Van Loan. An analysis of the total least squares problem. *SIAM journal on numerical analysis*, 17(6):883–893, 1980.

[13] Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. Noisy activation functions. In *International conference on machine learning*, pages 3059–3068. PMLR, 2016.

[14] Dan Hammerstrom. Working with neural networks. *IEEE spectrum*, 30(7):46–53, 1993.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[16] Yaoshiang Ho and Samuel Wookey. The real-world-weight cross-entropy loss function: Modeling the costs of mislabeling. *IEEE Access*, 8:4806–4813, 2019.

[17] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[18] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[19] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913*, 2017.

[20] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[21] Ivan Markovsky and Sabine Van Huffel. Overview of total least-squares methods. *Signal processing*, 87(10):2283–2302, 2007.

[22] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[23] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[24] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[26] Yousef Saad. Ilut: A dual threshold incomplete lu factorization. *Numerical linear algebra with applications*, 1(4):387–402, 1994.

[27] Sagar Sharma. Activation functions in neural networks. *towards data science*, 6.

[28] Zhou Wang and Alan C Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[29] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[30] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521, 2013.

[31] Zhilu Zhang and Mert R Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *arXiv preprint arXiv:1805.07836*, 2018.