KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Avoiding local minima in Deep Learning: a nonlinear optimal control approach

Jan Scheers

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
wiskundige ingenieurstechnieken

Promotor:
Prof. dr. ir. Panos Patrinos

Academiejaar 2020 – 2021

Preface

I would like to thank everybody who kept me busy the last year, especially my promoter and my assistants. I would also like to thank the jury for reading the text. My sincere gratitude also goes to my friends and my family.

Jan Scheers

Contents

Abstract

List of Abbreviations and Symbols

Abbreviations

ANN	Artificial Neural Network
MSE	Mean Squared Error
GD	Gradient Descent
BP	Backpropagation
OCP	Optimal Control Problem
MS	Multiple Shooting
NLP	Non-Linear Program
ALM	Augmented Lagrangian Method
DNN	Deep Neural Network
SGD	Stochastic Gradient Descent

Symbols

42	"The Answer to the Ultimate Question of and Everything" according to [**?**]
c	Speed of light
E	Energy
m	Mass
π	The number pi

Chapter 1

Introduction

1.1 Artificial Neural Networks

Artifical Neural Networks (ANNs) are a very popular machine learning model. They are known to be very expressive, leading to low statistical bias[REF]. With enough neurons, ANNs can approximate any function[REF]. They are especially useful for learning from very large data sets. But it is not entirely clear what the optimization of an ANN converges to, as the loss surface is highly non-convex[REF]. Nonetheless a number of results show that for wide enough networks, there are few "bad" local minima[REF].

ANNs are composed of 'neurons', which are in some ways analogous to biological neurons.Each neuron is a nonlinear function transforming the weighted sum of its inputs and a bias:

$$y = \sigma(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) \tag{1.1}$$

w_i are the weights, x_i are the inputs to the neuron, which come either from a previous neuron, or are fed into the network, σ is the activation function, and finally a bias b is also added to the sum. This is the McCulloch-Pitts neuron model[REF]. The most commonly used activation function σ is the Rectified Linear Unit (ReLU):

$$\sigma(x) = x^+ = \max(0, x) \tag{1.2}$$

Many other activation functions are possible, such as the sigmoid function ($\frac{1}{1+e^{-x}}$) or the hyperbolic tangent function ($\tanh(x)$).

A visual representation is shown in figure 1.1b. A full network is built by connecting layers of neurons as shown in figure 1.1a. An ANN can also be expressed as a combination of function composition and matrix multiplication, ignoring for a moment the bias vectors.

$$f(W, x) = W_L \sigma(W_{L-1} \sigma(...W_1 \sigma(W_0 x)...)) \tag{1.3}$$

where W_n are the matrixes of the connection weights and L is the depth of the network.

1.2 Neural Network Training

Training a neural network is an optimization problem as we will discuss in this section. Ruoyu Sun covers in [**?**] the current theory and algorithms for optimizing deep neural networks, upon which much of this section is based.

In a supervised learning problem a dataset of inputs and desired outputs is given: $x_i \in \mathbb{R}^{d_x}, y_i \in \mathbb{R}^{d_y}, i = 1, \dots, n$ with x_i the input vectors, y_i the desired output vectors and n the number of data points. We want the network to predict the output y_i based on the information in x_i, i.e. we want the network to learn the underlying mapping that connects the data. A standard fully connected network can be expressed as a combination of function composition and matrix multiplication as follows:

$$f_W(x) = W_L\sigma(W_{L-1}\sigma(...W_1\sigma(W_0x)...)) \tag{1.4}$$

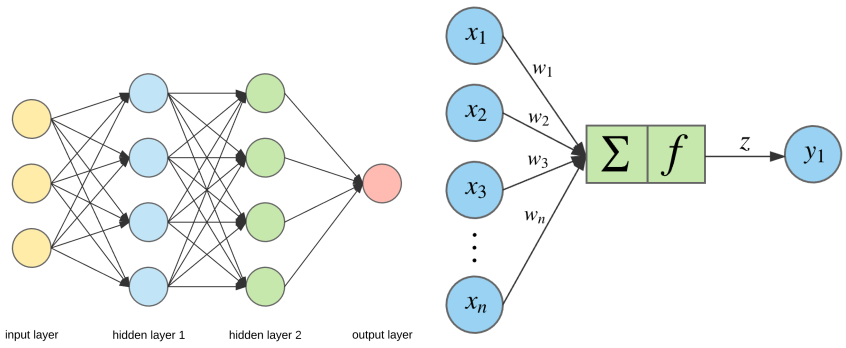
where L is the number of hidden layers in the network, W_j are matrixes of dimension $d_j \times d_{j-1}, j = 1 \dots L$ containing the connection weights and σ is the activation function. The bias vectors b_i have been omitted from this equation for clarity.

This function can also be defined recursively, which will be useful for later interpreting the network in an optimal control context.

$$\begin{aligned} z_0 &= x \\ z_{k+1} &= \sigma(W_k z_k + b_k), \quad k = 0, ..., L-1 \\ f_W(x) &= W_L z_L + b_L \end{aligned} \tag{1.5}$$

where b_k are the bias vectors, and the other variables are as defined before.

We want to pick the parameters of the neural network so that the predicted output $\hat{y}_i = f_W(x_i)$ is as close as possible to the true output y_i for a certain distance metric $l(\cdot, \cdot)$. Thus the optimization problem can be written as follows:



(A) Deep Neural Network (B) Single Neuron

Figure 1.1: Feedforward Deep Neural Network and Single Neuron - McCulloch-Pitts model. (Retrieved from https://towardsdatascience.com)

$$\underset{W}{\text{minimize}} \quad C(W) = \sum_{j=0}^{n} l(y_j, f_W(x_j)) \tag{1.6}$$

In this thesis only regression problems will be considered, where $l(x, y)$ is the quadratic loss function $l(x, y) = ||x^2 - y^2||$, a.k.a. mean square error (MSE). For classification problems cross entropy loss is the most common cost function: $l(x, y) = x\log(y) + (1-x)\log(y)$. Writing down equation 1.6 using the MSE gives the following optimization problem:

$$\underset{W}{\text{minimize}} \quad C(W) = \sum_{j=0}^{n} ||y_j - f_W(x_j)||_2^2 \tag{1.7}$$

Most methods for solving equation 1.6 are based on gradient descent (GD). This algorithm uses the gradient of the loss function to search for a local minimum:

$$W_{k+1} = W_k - \eta_k \nabla F(W_k) \tag{1.8}$$

where η_k is the step size (a.k.a. "learning rate") and $\nabla F(W_k)$ is the gradient of the loss function at the k-th iterate.

1.3 Neural network training as an optimal control problem

Neural network training can also be seen as an optimal control problem. A neural network is a multi-stage dynamical system, where every layer is a stage. The dynamics are ruled by the equations 1.5. In optimal control the following objective cost function is considered:

$$E(\theta) = \sum_{s=1}^{L-1} g^s(a^s, \theta^s) + h(a^L) \tag{1.9}$$

a^s and θ^s are the state and decision vectors at stage s, and g^s and h are the immediate cost at stage s and the terminal cost, respectively. In neural network training the immediate costs are usually dropped. If the terminal cost is taken to be the MSE, the same cost function as in equation 1.7 is found. The terminology and notation differences have been summarized in table 1.1. Using neural network terminology the optimal control problem to be solved is the following:

$$\begin{aligned} \underset{W}{\text{minimize}} \quad & \sum_{j=0}^{n} ||W_L z_{L,j} - y_j||_2^2 \\ \text{subject to} \quad & z_{0,j} = x_j \\ & z_{k+1,j} = \sigma(W_k z_{k,j} + b_k, 0), \quad k = 0, \ldots, L-1, j = 1, \ldots, n \end{aligned} \tag{1.10}$$

Notation	Optimal Control	Neural Network	Notation
θ	decision variables	weight parameters	W
a	state variables	(neuron) activation	z
	stage	layer	

Table 1.1: Comparison of notation between Control theory and Machine Learning

1.4 Backpropagation

The current standard algorithm for training a neural network is backpropagation (BP). It works by efficiently calculating the gradient for GD (equation 1.8). It was discovered and popularised in the context of neural networks by Rumelhart, Hinton & Williams (1986) [**?**]. But it has been shown that by viewing the problem in an optimal control context, the backpropagation algorithm is the same as the gradient formulas discovered by Kelley and Bryson in 1960 [**?**].

This section will explain how backpropagation works, it will largely follow and summarize [**?**]. Essentially BP evaluates the derivative of the cost function from the end to the front of the network, i.e. "backwards". It works on each input-output pair individually, with the full gradient given by the averaging over all pairs.

Given an input-output pair (x, y), the loss is:

$$C(y, W_L\sigma(W_{L-1}...\sigma(W_1\sigma(W_0x)))) \tag{1.11}$$

The loss is calculated forwards by evaluating the network, starting with the input x. Note the weighted input at each layer as $\nu_l = W_{l-1}z_{l-1} + b_{l-1}$. The activations $z_l = \sigma(\nu_l)$ and the derivatives $f_l' = \sigma'(\nu_l)$ at each layer l are stored for the backward pass.

The total derivative of the cost function C evaluated at the value of the network on the input x is given by the chain rule:

$$\begin{aligned}\frac{dC}{dx} &= \frac{dC}{dz_L}\cdot\frac{dz_L}{d\nu_L}\cdot\frac{d\nu_L}{dz_{L-1}}\cdot\frac{dz_{L-1}}{d\nu_{L-1}}\cdot\frac{d\nu_{L-1}}{dz_{L-2}}\cdots\frac{dz_0}{d\nu_0}\cdot\frac{d\nu_0}{dx}\\ &= \frac{dC}{dz_L}\cdot 1\cdot W_L\cdot f_{L-1}'\cdot W_{L-1}\ldots f_0'\cdot W_0\end{aligned} \tag{1.12}$$

The gradient ∇ is the transpose of the derivative.

$$\nabla_x C = W_0^T\cdot f_0'\ldots W_{L-1}^T\cdot f_{L-1}'\cdot W_L^T\cdot\nabla_{z_L}C \tag{1.13}$$

Backpropagation then essentially evaluates this expression from right to left. For this operation an auxiliary variable δ_l is introduced which is interpreted as the "error at layer l":

$$\delta_l = f_l'\cdot W_{l+1}^T\ldots W_{L-1}^T\cdot f_{L-1}'\cdot W_L^T\cdot\nabla_{z_L}C \tag{1.14}$$

The gradient of the weights in layer l is then:

$$\nabla_{W_l}C = \delta_l(z_l)^T \tag{1.15}$$

δ_l can easily be computed recursively:

$$\delta_{l-1} = f'_{l-1} \cdot W_l^T \cdot \delta_l \tag{1.16}$$

In this way the gradients of the weights are computed with only a few matrix operations per layer in a back to front fashion, this is backpropagation.

1.5 Direct Multiple Shooting Method

It has been shown that the BP algorithm detailed in the previous setting can be derived in an optimal control context in the spirit of dynamic programming [**?**]. But control theory has many other solution methods for OCPs. One of the more well known one is the direct multiple shooting (DMS) method [**?**].

In this method the state variables in the non-linear program (NLP), equation 1.10, are not eliminated using the dynamics. Instead the dynamics are kept as constraints to the NLP. This leads to a much larger NLP, but it will be more structured. This is in contrast to the direct single shooting method, where the dynamics are eliminated, leading to a small, but highly non-linear problem. The BP algorithm is analogous to a direct single shooting method.

The total number of variables that will be optimized for in a neural network is quite large. For a fully connected neural network of width W, depth L, there will be $\mathcal{O}(W^2L)$ weights to be optimized, a.k.a. decision variables in control theory. This is true for both BP and DMS. But for DMS, another $\mathcal{O}(WLN)$ state variables are added, with N the number of training samples. In control theory the tradeoff for making the problem larger in this way is that the problem becomes less non-linear. In practice this often makes the NLP easier to solve, which is why DMS is a common choice. For neural networks this could let the algorithm get stuck in a "bad" local minimum less often.

1.6 Goal of the Thesis

The main goal of this thesis is to implement the direct shooting method for training neural networks, and compare it to the industry standard backpropagation algorithm. First an initial exploration of the method will be conducted in MATLAB. Then an Augmented Lagrangian Method (ALM) will be developed to solve the DMS problem more efficiently, which will be coded in python.

The code will be compared to common gradient descent used in practice such as ADAM [**?**]. They will be compared in terms of speed, scalability, reliability and performance for a number of test problems.

In particular the objective will be to see if this algorithm can better handle known challenges for current training algorithms, such as the "vanishing/exploding gradient problem", or convergence to "bad local minima" (Goodfellow et al. [**?**], Sec. 8.2).

Chapter 2

Initial exploration

In this chapter an initial exploration of the problem is perfomed. The direct multiple shooting approach will be solved using a general nonlinear solver. For a couple small test problems the novel algorithm will be compared to the industry standard backpropagation algorithm

2.1 Direct Multiple Shooting Approach

The optimal control problem (OCP) of training a neural network in 1.10 that is at the core of this thesis is repeated here again:

$$\begin{aligned} \underset{W}{\text{minimize}} \quad & \sum_{j=0}^{n} ||W_L z_{L,j} - y_j||_2^2 \\ \text{subject to} \quad & z_{0,j} = x_j \\ & z_{k+1,j} = \sigma(W_k z_{k,j} + b_k, 0), \quad k = 0, \dots, L-1, j = 1, \dots, n \end{aligned} \tag{2.1}$$

This is a nonlinear program with nonlinear equality constraints. It's possible to eliminate the states z_k using the dynamics, giving rise to an unconstrained nonlinear program as follows:

$$\underset{W}{\text{minimize}} \quad \sum_{j=0}^{n} ||f_W(x_j) - y_j||_2^2 \tag{2.2}$$

Where f_W is the neural network as a function as in 1.3. Solving this problem would be the single shooting approach. One can find the gradient of this function using dynamic programming techniques, leading again to the backpropagation algorithm. A full derivation of the backpropagation algorithm using control theory can be found in Mizutani et al. [**?**] and Dreyfus et al. [**?**]

This thesis instead tries to solve equation directly, which corresponds to a multiple shooting approach.

2.2 Test setup

Before writing a custom solver, the problem is explored using a general nonlinear program(NLP) solver. For this task the `fmincon` method implemented in MATLAB is used together with the YALMIP optimization library. This method implements a interior point algorithm for solving constrained NLPs.

For comparison against the standard backpropagation the neural network toolbox of MATLAB is used. The default training algorithm in this toolbox is `trainlm` which implements a Levenberg-Marquardt backpropagation algorithm. However this method is too well suited for the simple curve fitting problem the next sections use for testing. As figure 2.1 shows, this algorithm finds a perfect fitting curve almost every time. For this reason `traingd` is used instead, which is a standard Gradient Descent(GD) backpropagation algorithm. GD based algorithms such as ADAM are some of the most used algorithms in practice because of their simplicity and low cost per epoch.[REF] In contrast to `trainlm` this method will often settle in a "bad" local minimum and will not reach the same performance.

The test case used in this chapter is a regression problem to fit a neural network to the following sine function:

$$y_j = -.8\sin(x_j) + \mathcal{N}(0, \delta), x \in [0, 1], j = 1..N \tag{2.3}$$

where $\delta = 0.1$ adds noise, and N is the number of datapoints. The input-output pairs (x_j, y_j) are split into a training set of size $\frac{4}{5}N$ and a test set of size $\frac{1}{5}N$. Figure 2.1 plots this function and shows the output of a network which has been fit using `trainlm`. This is also a clear example of overfitting.

This test problem, as well as the test problem used in chapter 3, **??**, come from the course on neural networks given at the KULeuven []. [version numbers, hardware specs]

2.3 Hyperbolic tangent activation function

In a first experiment a small fully connected feedforward neural network is constructed with 2 hidden layers with each layer containing 3 nodes with a hyperbolic tangent activation function: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This activation function is smooth and works well for small networks and for curve fitting.

This network is fitted to the data described in the previous section using both algorithms. For the convergence of the multiple shooting method it is important that the initial point is feasible, therefore the state variables z_k will be initiatialized by simulating the network once using random initialization for the weights.

Figure 2.2 shows the result of a good training run for each algorithm. Figure 2.2a and 2.2b show the training performance per epoch. The gradient descent algorithm trains for many more iterations, but each iterate is much cheaper. Both algorithms stop progressing after a while, indicating a local minimum has been reached. The prediction of the network after training is plotted in 2.2c and 2.2d.

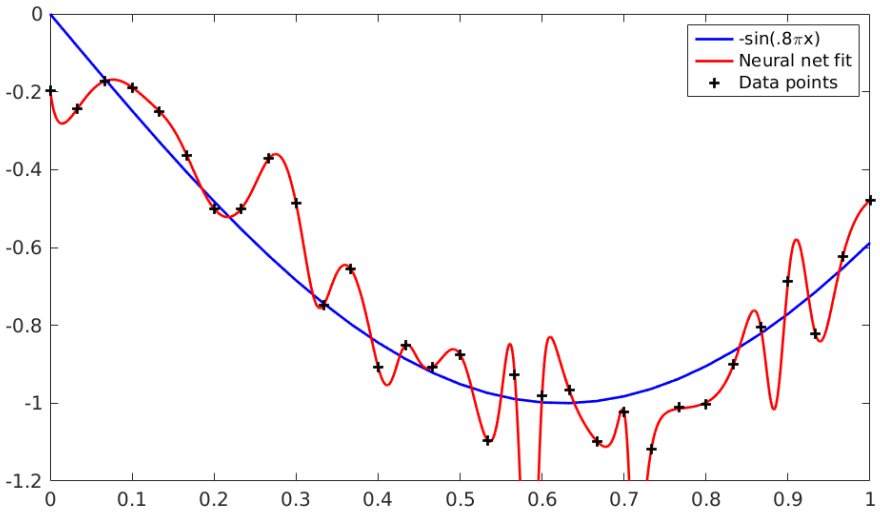
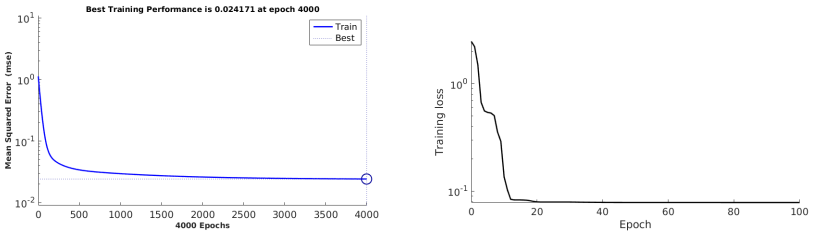
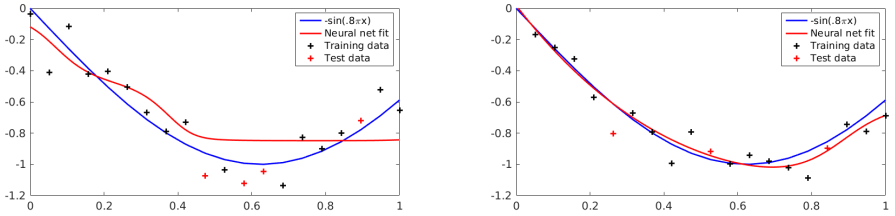


FIGURE 2.1: Test function approximated with `trainlm` training algorithm. The data has been overfitted.



(A) Training performance of Gradient Descent

(B) Training performance of Multiple Shooting



(C) Neural network fit with Gradient Descent

(D) Neural network fit with Multiple Shooting

FIGURE 2.2: Comparing algorithm performance for regression problem

Algorithm	best tr MSE	avg tr MSE	avg test MSE	avg run time
Gradient Descent	0.0108	0.0299	0.0605	1.577s
Multiple Shooting	0.0537	0.2350	0.2724	23.82s

Table 2.1: Result of 20 training runs for each algorithm for *tanh* activation function. Average MSE values are calculated over those runs which converged.

To quantify the difference in performance, each algorithm is run for 20 training runs. In each run both algorithms start with the same data and same initial weights. The weight initialization function is `initnw` which implements Nguyen-Widrow weight initialization. GD and MS are run for 2000 epochs and 40 epochs respectively, values which are chosen based on when each algorithm usually stops improving. The results are shown in table 2.1.

Using `fmincon`, MS only converges 6 out of 20 times, while GD always converges. GD also has much better average performance and runs much quicker. However, at its best, MS can at least show similar performance to GD. The GD algorithm runs on GPU while the MS algorithm runs on CPU, so faster runs should be expected.

2.4 Rectified linear unit

In this section the same experiment as last time is run, but with a different network architecture. Again a network of two layers is used, but with 8 neurons in each layer, and using a ReLU activation function.

For the multiple shooting approach the RELU activation function will have to be reformulated, in order to have smooth constraints. The ReLU function can be transformed as follows:

$$
\begin{gathered}
x_{k+1}^j = \max(W_k x_k^j, 0) \\
\Updownarrow \\
x_{k+1}^j = -\min(-W_k x_k^j, 0) \\
\Updownarrow \\
\min(x_{k+1}^j - W_k x_k^j) = 0 \\
\Updownarrow \\
(x_{k+1}^j - W_k x_k^j)^\top x_{k+1}^j = 0, \\
x_{k+1}^j \geq 0, x_{k+1}^j - W_k x_k^j \geq 0
\end{gathered}
$$

Table 2.2 shows the results after 20 training runs, under the same conditions as the previous section. In this case MS converged 16/20 times, while GD still always converges. GD still runs at the same speed as the previous test despite the larger nextwork, while the run time for MS has significantly increased.

Algorithm	best tr MSE	avg tr MSE	avg test MSE	avg run time
Gradient Descent	0.0041	0.0274	0.0489	1.480s
Multiple Shooting	0.0348	0.2111	0.2696	115.8s

TABLE 2.2: Result of 20 training runs for each algorithm for ReLU activation function. Average MSE values are calculated over those runs which converged.

2.5 Conclusion

The MS algorithm did not outperform the GD in any way in either of the tests. However this chapter has demonstrated that it is feasible to train a network in this manner. Good solutions are possible using this method. The main issue is that `fmincon` is a very general method, and not well suited to this specific problem. For this reason the next chapter will explore the Augmented Langrangian Method (ALM), a common algorithmic framework for solving constrained NLPs.

Chapter 3

Augmented Lagrangian Method

In this chapter the direct multiple shooting approach is examined more closely and a more specific algorithm is designed to replace `fmincon`, which is a very general method. For this problem the Augmented Lagrangian Method has been chosen. It will be implemented in python using `numpy`,`scipy`, and `keras`

3.1 Classical Augmented Lagrangian Method

The Augmented Lagrangian Method (ALM) is a classical algorithmic framework for solving constrained NLPs. It was first discovered in 1969 [**?**],[**?**] and was known as the method of multipliers. The textbook examples of this method can be found in [**?**] and [**?**].

It is designed to minimize equality constrained optimization problems defined in the following way:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \end{aligned} \tag{3.1}$$

ALM solves this by minimizing a series of unconstrained problems in a similar manner as the penalty method. In each iteration a β-augmented Lagrangian $\mathcal{L}_\beta(x, \lambda)$ is minimized for x:

$$\min_{x} \ \max_{\lambda} \ \mathcal{L}_\beta(x, \lambda) = f(x) + < \lambda, h(x) > + \frac{\beta}{2} ||h(x)||_2^2 \tag{3.2}$$

where $\beta > 0$ is the penalty weight. This can be viewed as a penalty method which has been shifted using the term in λ[**?**]. When β or λ tend to infinity, $h(x)$ will be forced to zero, leading the Lagrangian to converge to the same solution as the original problem.

The algorithm proceeds as follows:

$$\begin{aligned} x_{k+1} &= \operatorname*{argmin}_{x} \ \mathcal{L}_\beta(x, \lambda_k) \\ \lambda_{k+1} &= \lambda_k + \sigma_k h(x_{k+1}) \end{aligned}$$

where σ_k is the step size at iteration k. Then in each step the penalty parameter β_k is increased or kept the same, depending on the size of the constraint violation. This continues until an acceptable solution has been found:

$$||h(x_k)|| \leq \tau_1 \text{ and } ||\nabla_x \mathcal{L}_{\beta_k}(x_k, \lambda_k)|| \leq \tau_2 \tag{3.3}$$

with τ_1, τ_2 the chosen tolerances.

3.2 Applied

The OCP equation 2.1 of training a neural net with MSE loss function is a nonlinear least squares problem:

$$\begin{aligned} \min \quad & \frac{1}{2}||F(x)||_2^2 \\ \text{s. t.} \quad & h(x) = 0 \end{aligned}$$

The minimization problem that will be solved in each iteration is then:

$$\begin{aligned} \min_x \quad \mathcal{L}_\beta(x, \lambda) &= \frac{1}{2}||F(x)||_2^2 + < \lambda, h(x) > + \frac{\beta}{2}||h(x)||_2^2 \\ &= \frac{1}{2}||F(x)||_2^2 + \frac{\beta}{2}||h(x) + \lambda/\beta||_2^2 - \frac{1}{2\beta}||\lambda||_2^2 \\ &= \frac{\beta}{2}\left|\left|\begin{bmatrix} F(x)/\sqrt{\beta} \\ h(x) + \lambda/\beta \end{bmatrix}\right|\right|_2^2 \\ &= \frac{\beta}{2}||M_\beta(x, \lambda)||_2^2 \end{aligned} \tag{3.4}$$

Instead of using the textbook algorithm, a more recent paper algorithmic framework is adapted to the problem, shown in Algorithm **??**

Algorithm 1: Inexact Augmented Lagrangian Method

Input: penalty parameter β, stopping tolerance τ
weight and state variables x_0, $\lambda_0 = \mathcal{N}(0, 1)$;
for k = *0,1,...* **do**
  $\eta_k = 1/\beta^k$;
  find x_{k+1} such that
  $||\nabla_x M_\beta(x, \lambda_k)|| \leq \eta_k$
  $\sigma_{k+1} = \min(\frac{||h(x_0)|| \log^2 2}{||h(x_k+1)|| k \log^2(k+1)}, 1)$
  $\lambda_{k+1} = \lambda_k + \sigma_{k+1} h(x_k + 1)$
  Stop if
  ...
end

TODO: cite textbook, 1906

Instead of using fmincon, now an Augmented Lagrangian Method (ALM) is used to solve the Optimal Control Problem (OCP). The OCP is described in equation

1.10 and is printed here again:

$$\begin{aligned}
\underset{W}{\text{minimize}} \quad & \frac{1}{2}\sum_{j=0}^{N} ||W_H z_H^j - y^j||^2 \\
\text{subject to} \quad & \\
& 0 = x_0 - \sigma(W_0 x_0^j + b_0) && j = 1, \ldots, N \\
& 0 = z_{k+1}^j - \sigma(W_k x_k^j + b_k), && k = 0, \ldots, H-1, j = 1, \ldots, N
\end{aligned}$$

This is a nonlinear least squares problem:

The ALM is designed to solve this sort of problem. It constructs an unconstrained problem by adding the constraints in both a penalty and a lagrangian term. The lagrangian cost function looks like this:

$$\mathcal{L}_c(x, \lambda) = \frac{1}{2}||F(x)||_2^2 + < \lambda, h(x) > + \frac{c}{2}||h(x)||_2^2 \tag{3.5}$$

where c is a penalty term, and λ are the lagrangian parameters. The algorithm consists of an inner and outer loop. In the inner loop of the algorithm $\mathcal{L}_c x, \lambda$ is minimized for x^k up to a certain tolerance.

$$\begin{aligned}
\min_x \quad \mathcal{L}_c(x, \lambda) &= \frac{1}{2}||F(x)||_2^2 + < \lambda, h(x) > + \frac{c}{2}||h(x)||_2^2 \\
&= \frac{1}{2}||F(x)||_2^2 + \frac{c}{2}||h(x) + \lambda/c||_2^2 - \frac{1}{2c}||\lambda||_2^2 \\
&= \frac{c}{2}\left|\left|\begin{bmatrix} F(x)/\sqrt{c} \\ h(x) + \lambda/c \end{bmatrix}\right|\right|_2^2 \\
&= \frac{c}{2}||G_c(x, \lambda)||_2^2
\end{aligned} \tag{3.6}$$

Where G_c is defined as the function within the square norm. This is a nonlinear least squares problem which is solved by the `least_squares` procedure implemented in the python package `scipy.optimize`. The function insid until the norm of the gradient of G_c meets a certain tolerance ϵ:

$$||\nabla_k G_{c_k}(x^k, \lambda^k)||_2 \leq \epsilon \tag{3.7}$$

where k is the current iterate and λ^k are the current lagrangian parameters. The Jacobian matrix $J = \nabla_k G_{c_k}^T$ is calculated analytically and analysed further in the next section. The Langrangian parameters λ^k are then updated in each iteration using this rule:

$$\lambda^{k+1} = \lambda^k + c_k h(x^k) \tag{3.8}$$

A new penalty parameter c^{k+1} are updated using the rule described in [], which depends on the norm of the gradient. If the constraints are sufficiently respected, the penalty parameter is not updated, otherwise the penalty parameter is increased by a

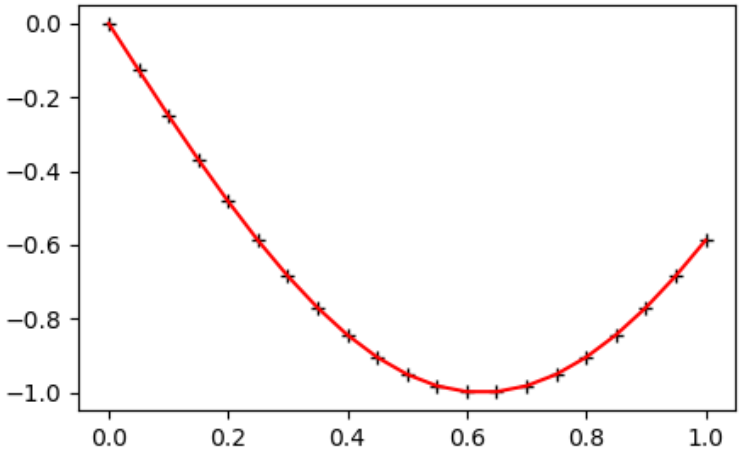
factor 100. The algorithm continues in this manner until the norm of the Jacobian matrix reaches a tolerance ζ.

$$\begin{aligned}\mathcal{L}_c(x,\lambda) &= \frac{1}{2}||F(x)||_2^2 + <\lambda, h(x)> + \frac{c}{2}||h(x)||_2^2 \\ &= \frac{1}{2}||F(x)||_2^2 + \frac{c}{2}||h(x)+\lambda/c||_2^2 - \frac{1}{2c}||\lambda||_2^2 \\ &= \frac{c}{2}\left|\left|\begin{bmatrix} F(x)/\sqrt{c} \\ h(x)+\lambda/c \end{bmatrix}\right|\right|^2\end{aligned} \tag{3.9}$$

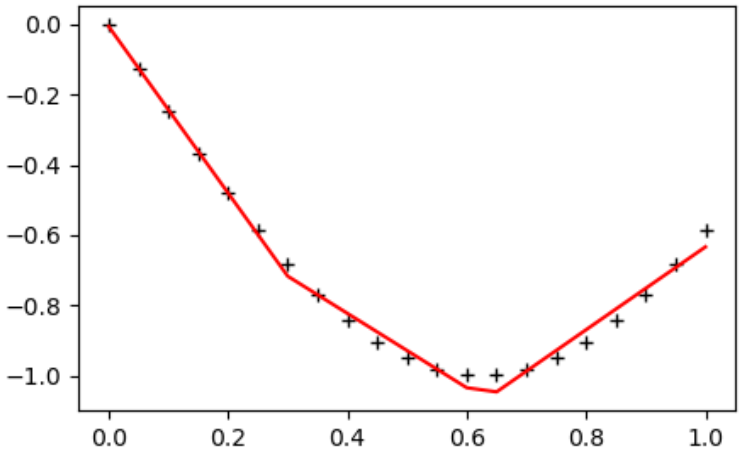
3.3 Jacobian

To solve the least squares problem, the Jacobian matrix must be calculated. It has a relatively sparse structure because there are no distant connections in the neural net, each layer is only connected to the next one and the previous one.

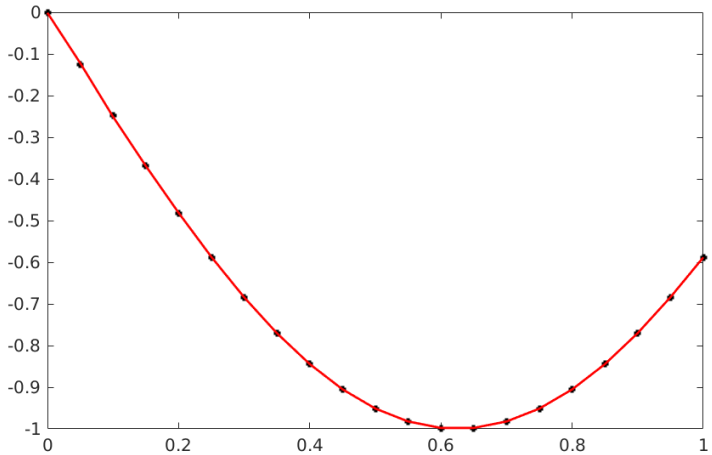
Neural networks obviously have many possible architectures, so a simple fully connected rectangular feedforward network is considered. It has an input dimension I, an output dimension O, and it has D hidden layers of width W. The weight matrixes have $I \times W + O \times W + (D-1) \times W \times W$ parameters, the bias vectors have $D \times W + O$ parameters and the state vectors have $D \times W \times N$ parameters. On the other hand $\mathcal{L}_c(x,\lambda)$ will have an output dimension of $D \times W \times N + O \times N$. The dimension of the Jacobian will be $(D \times W \times N + O \times N) \times (D \times W \times N + O + (D + I + O) \times W + (D-1) \times W^2)$. Therefore the Jacobian will be taller than it is wide when $N \geq 1 + (D+I+O) \times W/O + (D-1) \times W^2/O$. Written out completely the Jacobian will look like Table **??**



(A) Training performance of simultaneous approach, with tansig activation function



(B) Training performance of simultaneous approach, with ReLU activation function



(C) Training performance of backpropagation, with ReLU activation function

FIGURE 3.1: Performance of algorithms for simple regression problem

3.4 Algorithmic Verification of Jacobian

The Jacobian in the previous section was derived by hand. In this section will be explained how the Jacobian was verified algorithmically.

By Algorithmic Differentation the numerical value for the Jacobian can be deduced. For this the AlgoPy python module was used. By adding code from this packet to the calculation of the neural network, the Jacobian is calculated along with the network. This result was then compared to the analytical result for a number of different network configurations, confirming them to be equal within a small tolerance. The code is available at https://github.com/jan-scheers/thesis

3.5 Alternative Representation

In this section we explain an alternative representation which is mathematically the same. Figure **??** shows the Jacobian, but with the rows corresponding to the loss function put at the bottom. It shows a diagonal structure, because each layer in this feedforward net is only connected to the adjacent layers.

3.6 Testing

In this section the tests from the previous chapter are run again, this time using the ALM algorithm. The least squares problem in the inner loop is solved using the `least_squares` method in `numpy 1.20.1`, which is provided with the Jacobian calculated in the previous section. This time the stopping criterion is that the cost function described in equation **??** is less than a tolerance of $1e^{-6}$. Figure **??** and Figure **??** show the result of these two tests.

	$\nabla\mathcal{L}$	W_{0_1}	W_{0_2}	...	W_{0_W}	b_0
		I	I	...	I	W
F	O*N	0	0	...	0	0
h_1	N	$-x\sigma'(W_{0_1}x+b_{0_1})$	0	...	0	$-\sigma'(W_{0_1}x+b_{0_1})$
	N	0	$-x\sigma'(W_{0_2}x+b_{0_2})$	...	0	$-\sigma'(W_{0_2}x+b_{0_2})$
	...	...	...	...	...	...
	N	0	0	...	$-x\sigma'(W_{0_W}x+b_{0_W})$	$-\sigma'(W_{0_W}x+b_{0_W})$
h_2	W*N	0	0	...	0	0
...	...	...	...	...	...	...
h_D	W*N	0	0	...	0	0

	$\nabla\mathcal{L}$	W_{i_1}	W_{i_2}	...	W_{i_W}	b_1
		W	W	...	W	W
F	O*N	0	0	...	0	0
h_1	W*N	0	0	...	0	0
...	...	...	...	...	...	...
h_{i+1}	N	$-z_1\sigma'(W_{i_1}z+b_{i_1})$	0	...	0	$-\sigma'(W_{i_1}x+b_{i_1})$
	N	0	$-z_1\sigma'(W_{i_2}z+b_{i_2})$	...	0	$-\sigma'(W_{i_2}x+b_{i_2})$
	...	...	...	...	...	...
	N	0	0	...	$-z_1\sigma'(W_{i_W}z+b_{i_W})$	$-\sigma'(W_{i_W}x+b_{i_W})$
...	...	...	...	...	...	...
h_D	W*N	0	0	...	0	0

	$\nabla\mathcal{L}$	W_{D_1}	W_{D_2}	...	W_{D_O}	b_D
		W	W	...	W	O
F	N	$-\frac{z_D}{\sqrt{c}}\sigma'_O(W_{D_1}x+b_{D_1})$	0	...	0	$-\frac{1}{\sqrt{c}}\sigma'_O(W_{D_1}x+b_{D_1})$
	N	0	$-\frac{z_D}{\sqrt{c}}\sigma'_O(W_{D_2}x+b_{D_2})$	...	0	$-\frac{1}{\sqrt{c}}\sigma'_O(W_{D_2}x+b_{D_2})$
	...	...	...	...	...	...
	N	0	0	...	$-\frac{z_D}{\sqrt{c}}\sigma'_O(W_{D_O}x+b_{D_O})$	$-\frac{1}{\sqrt{c}}\sigma'_O(W_{D_O}x+b_{D_O})$
h_1	W*N	0	0	...	0	0
...	...	...	...	...	...	...
h_D	W*N	0	0	...	0	0

Square Diagonal Matrices

	$\nabla\mathcal{L}$	z_{i_1}	z_{i_2}	...	z_{i_W}
		N	N	...	N
F	O*N	0	0	...	0
h_1	W*N	0	0	...	0
...	...	...	...	...	...
h_i	N	1	0	...	0
	N	0	1	...	0
	...	...	...	...	...
	N	0	0	...	1
h_{i+1}	N	$-W_{i_{1,1}}\sigma'(W_{i_1}z_i+b_{i_1})$	$-W_{i_{1,2}}\sigma'(W_{i_1}z_i+b_{i_1})$	...	$-W_{i_{1,W}}\sigma'(W_{i_1}z_i+b_{i_1})$
	N	$-W_{i_{2,1}}\sigma'(W_{i_2}z_i+b_{i_2})$	$-W_{i_{2,2}}\sigma'(W_{i_2}z_i+b_{i_2})$	...	$-W_{i_{2,W}}\sigma'(W_{i_2}z_i+b_{i_2})$
	...	...	...	...	...
	N	$-W_{i_{W,1}}\sigma'(W_{i_W}z_i+b_{i_W})$	$-W_{i_{W,2}}\sigma'(W_{i_W}z_i+b_{i_W})$	...	$-W_{i_{W,W}}\sigma'(W_{i_W}z_i+b_{i_W})$
...	...	...	...	...	...
h_D	W*N	0	0	...	0

	$\nabla\mathcal{L}$	z_{D_1}	z_{D_2}	...	z_{D_W}
		N	N	...	N
F	N	$-W_{D_{1,1}}\sigma'_O(W_{D_1}z_D+b_{D_1})$	$-W_{D_{1,2}}\sigma'_O(W_{D_1}z_D+b_{D_1})$	...	$-W_{D_{1,W}}\sigma'_O(W_{D_1}z_D+b_{D_1})$
	N	$-W_{D_{2,1}}\sigma'_O(W_{D_2}z_D+b_{D_2})$	$-W_{D_{2,2}}\sigma'_O(W_{D_2}z_D+b_{D_2})$	...	$-W_{D_{2,W}}\sigma'_O(W_{D_2}z_D+b_{D_2})$
	...	...	...	...	...
	N	$-W_{D_{O,1}}\sigma'_O(W_{D_O}z_D+b_{D_O})$	$-W_{D_{O,2}}\sigma'_O(W_{D_O}z_D+b_{D_O})$	...	$-W_{D_{O,W}}\sigma'_O(W_{D_O}z_D+b_{D_O})$
h_1	W*N	0	0	...	0
...	...	...	...	...	...
h_D	N	1	0	...	0
	N	0	1	...	0
	...	...	...	...	...
	N	0	0	...	1

TABLE 3.1: Jacobian of feedforward neural network

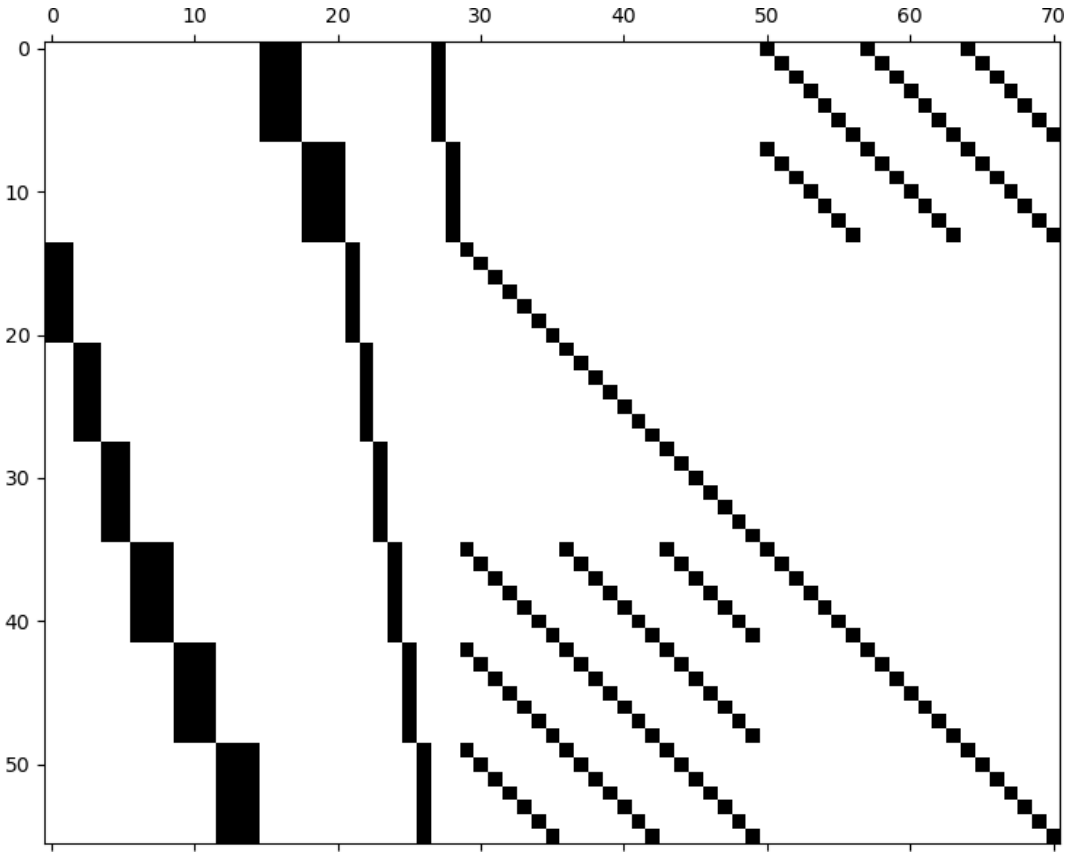


Figure 3.2: Nonzero elements of Jacobian, for network with I=2,O=2,W=3,D=2,N=7

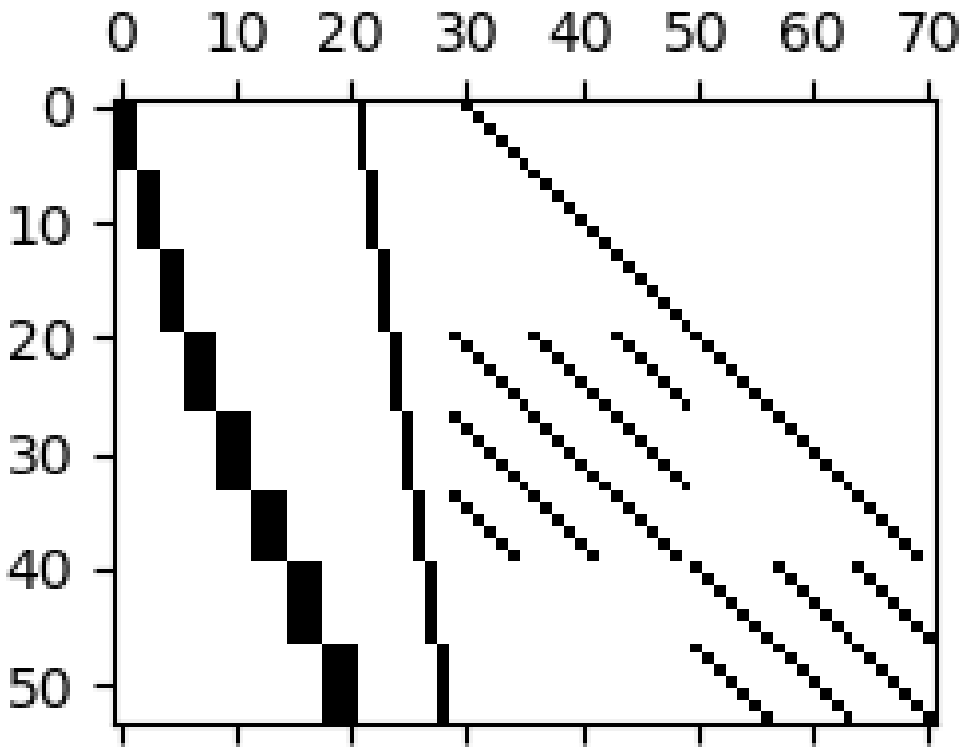


FIGURE 3.3: Nonzero elements of Jacobian, for network with I=2,O=2,W=3,D=2,N=7, rearranged

Chapter 4

Numerical Experiments

In this chapter we shall compare the augmented Lagrangian Method with several industry standard backpropagation algorithms. We considered the following algorithms: ADAM, ...

The first comparison will be run on a small example, for which we expect that all algorithms should easily converge to a good solution.

4.1 Training Algorithms and Stopping Criteria

This section will discuss the problem of comparing different training algorithms, which is not as straightforward as it might seem. In the literature many different methods of comparison are used, usually picking the one which fits their algorithm the best. The main issue is the choice of stopping criterion. One can choose a tolerance for the loss function, but this might not always be reached due to local minima. A second option is to stop after a set number of epochs, but an epoch in one algorithm is not necessarily equivalent to an epoch in another algorithm. A third option is to let each algorithm run for a specified length of time, and compare the loss on the training set after each run. This might be the most fair option, because average running time is the most important factor in practice. On the other hand it is hard for a new, experimental method to be as optimally coded as one that has been used in the field for many years already.

Further complicating the matter is that in practice many different early stopping criteria are used as well, to protect against overfitting. The most common early stopping criterion is to stop when a minimum in the validation dataset has been reached. Furthermore, because of the complexity of the loss surface of a DNN, and the random initialization of the weight matrices, each training run will follow a different trajectory and find a different local minimum.

Because the goal of this thesis is to compare training performance, overfitting is not a great concern. Therefore validation data will not be used in the training process. Instead the training will stop once the improvement in training loss stagnates, indicating a local minimum has been reached. ALM will stop when the following inequality holds true:

$$(1+\epsilon)C(W_{k+1}) > C(W_k) \tag{4.1}$$

Where $C(W_k)$ is the loss at epoch k and ϵ is a small tolerance value. Because ALM only takes only a few costly epochs to converge, it should make sense to not wait many epochs to confirm that the method has stagnated its progress. On the other hand the main algorithm against which the ALM method will be compared is the ADAM algorithm, which may take thousands of epochs to converge. To find the minimum in the training loss the `EarlyStopping` method implemented in `keras` will be used. This method has a patience value p, meaning that the training will stop once p epochs have passed without any improvement.

4.2 Test setup

Each test will average the results over many training runs so as to get a more accurate and fair picture of the performance that can be expected from each algorithm. Typically each test will use 20 training runs.

The weights of the network will be initialized using Xavier initialization for the layers using the $\tanh(x)$ activation function and Kaiming initialization for the layers using the ReLU activation function, as is standard in practice. Both algorithms will start with the same initalization in each comparison.

All tests are run on .... , GBS,

4.3 Fully connected feedforward network

For the first comparison a similar regression problem as in chapter one will be considered. In a first try the same sine function was used as in equation **??** in chapter 1. The issue with this function is that it is too simple of a training problem. Both algorithms converge quite quickly to a very good solution, one of which is shown in figure **??**, and it is difficult to differentiate the algorithms.

For this reason a problem with more depth is required. In this first test a squared sine function will be approximated instead. This function oscillates progressively faster, so that the training algorithm can always keep finding a better solution. Figure **??** shows how different training configurations let a network approximate a larger or smaller segment of the function. The function definition is as follows:

$$y = \sin^2(x) + \mathcal{N}(0, \delta), x \in [0, \pi] \tag{4.2}$$

The training performance of the ALM method will be compared to the Stochastic Gradient Descent (SGD) method implemented in `keras`. The weights of the network will be initialized using Xavier initialization for the layers using the $\tanh(x)$ activation function and Kaiming initialization for the layers using the ReLU activation function. Both training algorithms will start from the same initial point for each test. Training will stop when progress on the training loss stagnates, indicating a local minimum has been reached. For SGD the `EarlyStopping` class will be used to stop the training

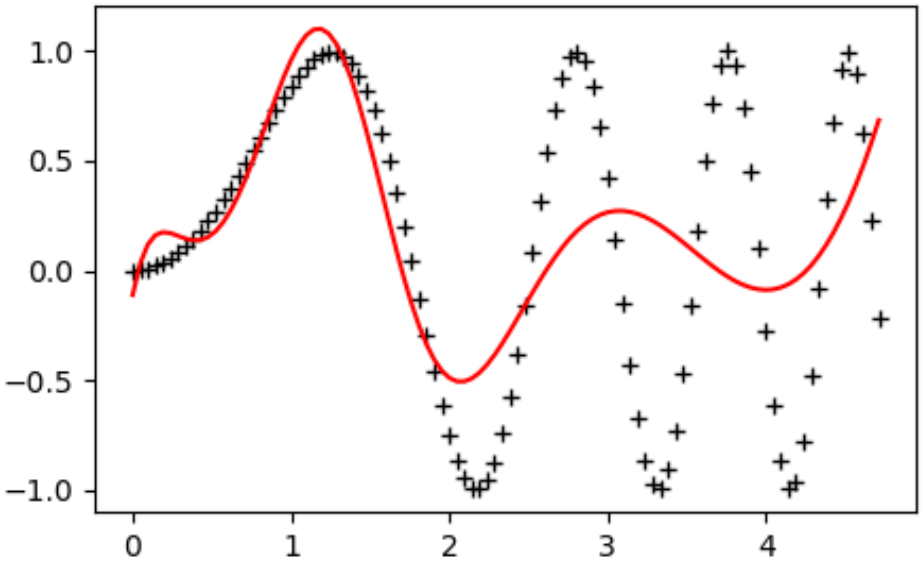


FIGURE 4.1: Feedforward network with 2 hidden layers of 20 relu units, trained on test function using ALM, using 80 training pairs

when 4 epochs have passed without improvement, while for ALM the following stopping criterion will be used:

$$(1+\epsilon)C(W_{k+1}) > C(W_k) \tag{4.3}$$

Where $C(W_k)$ is the loss at epoch k and ϵ is chosen to be $1e^{-2}$.

In the first test the effect of the number of datapoints is analyzed, by taking a neural network

sine regression

- sine regression
- compare until validation performance under ...
- relu vs tanh
- depth of network
- width of network
- number of input-output pairs
- scalability
  - Optimization for deep learning: theory and algorithms

  - Goals of optimization
  - Problem formulation
  - Gradient descent:
  - Tips tricks
  - State of the art algorithms

- [?]
  - Optimization for deep learning: theory and algorithms
  - Goals of optimization
  - Problem formulation
  - Gradient descent:
  - Tips tricks
  - State of the art algorithms

- [?]

  "The Kelley-Bryson gradient formulas for such problems have been rediscovered by neural- network researchers and termed back propagation"

- [?]

  Practical Augmented lagrangian method

- [?] "Indeed the loss surface of neural networks optimization problems is highly non-convex: it has a high number of saddle points which may slow down the convergence (4). A number of results (3; 13; 14) suggest that for wide enough networks, there are very few "bad" local minima, i.e. local minima with much higher cost than the global minimum"

- [?] Original Backpropagation paper

4.4 Needed

- Basic Neural network intro
- Optimal control
- Simultaneous approach

Chapter 5

Conclusion

The main goal of this thesis was to present a novel algorithm for training deep neural networks, using methods commonly used in Optimal Control Theory. The optimization problem of neural network training was reformulated into an Optimal Control Problem. In a first step the direct multiple shooting method was applied to the OCP, which is commonly used in control theory for very nonlinear problems.

This method was implemented in MATLAB using a very general optimization function `fmincon`, proving that the problem was feasible. Then an Augmented Lagrangian Method was presented to solve the OCP. First a textbook ALM method was implemented, later this was improved by using an inexact ALM detailed in []. The Jacobian matrix at the center of the ALM was verified to be correct using automatic differentiation tools. This ALM method was then written into code using python with `numpy`,`scipy`, and `keras`

Finally the novel algorithm was tested against industry standard backpropagation methods, ADAM and SGD. It compares favorably for some smaller, harder training problems. (SPECIFY) However it scales poorly for larger datasets. As the title indicates, the original goal of this new algorithm was to better avoid "bad" local minima in training. But the new algorithm does not show much improvement compared to standard methods, and practice has shown that is not a common problem. In larger neural networks, experts suspect local minima usually have comparable performance to the global minimum [**?**].

TODO: Batch approach. It also cannot yet handle loss functions besides MSE, which could be the topic of further research.

Appendices

Appendix A

Source code

This appendix contains the source code of the experiments.

A.1 First experiment source

```
N = 21;
xin = linspace(0,1,21);
y = -sin(.8*pi*x);

w1 = sdpvar(3,1);
b1 = sdpvar(3,1);
x1 = sdpvar(3,N);
w2 = sdpvar(3,3,'full');
b2 = sdpvar(3,1);
x2 = sdpvar(3,N);
w3 = sdpvar(3,1);
b3 = sdpvar(1,1);


assign(w1,2*rand(3,1)-1);
assign(b1,2*rand(3,1)-1);
assign(x1,tansig(value(w1*xin+repmat(b1,1,N))));
assign(w2,2*rand(3,3)-1);
assign(b2,2*rand(3,1)-1);
assign(x2,tansig(value(w2*x1 +repmat(b2,1,N))));
assign(w3,2*rand(3,1)-1);
assign(b3,2*rand(1,1)-1);

res = w3'*x2 + b3 - y;
obj = res*res';
%
f1 = (x1-(w1*xin+repmat(b1,1,N)));
```

```
f2 = (x2-(w2*x1 +repmat(b2,1,N)));
% con = [f1 >= 0; x1 >= 0; f1.*x1 <= 0;
%        f2 >= 0; x2 >= 0; f2.*x2 <= 0];
% con = [x1 == max(w1*xin+repmat(b1,1,N),0);
%        x2 == max(w2*x1 +repmat(b2,1,N),0)];
con = [x1 == tansig(w1*xin+repmat(b1,1,N));
       x2 == tansig(w2*x1 +repmat(b2,1,N))];
ops = sdpsettings('usex0',1);
ops.fmincon.MaxFunEvals = 20000;
ops.fmincon.MaxIter = 200;

optimize(con,obj,ops);

%%
x1s = tansig(value(w1)*xin+value(b1));
x2s = tansig(value(w2)*x1s+value(b2));
ys = value(w3)'*x2s+value(b3);

hold off
plot(xin,y,'Linewidth',3);
hold on
plot(xin,ys,'g--','Linewidth',4);
```

A.1.1 Second experiment source

```
clear
N = 21;W = 10;
xin = linspace(0,1,21);
y = -sin(.8*pi*xin);

w1 = sdpvar(W,1);
b1 = sdpvar(W,1);
x1 = sdpvar(W,N);
w3 = sdpvar(W,1);
b3 = sdpvar(1,1);


assign(w1,2*rand(W,1)-1);
assign(b1,2*rand(W,1)-1);
assign(x1,poslin(value(w1*xin+repmat(b1,1,N))));
assign(w3,2*rand(W,1)-1);
assign(b3,2*rand(1,1)-1);

res = w3'*x1 + b3 - y;
obj = res*res';
```

```
%
f1 = (x1-(w1*xin+repmat(b1,1,N)));
%f2 = (x2-(w2*x1 +repmat(b2,1,N)));
con = [f1 >= 0; x1 >= 0; f1.*x1 <= 0;];
%        f2 >= 0; x2 >= 0; f2.*x2 <= 0];
% con = [x1 == max(w1*xin+repmat(b1,1,N),0);
%        x2 == max(w2*x1 +repmat(b2,1,N),0)];
% con = [x1 == tansig(w1*xin+repmat(b1,1,N));
%        x2 == tansig(w2*x1 +repmat(b2,1,N))];
ops = sdpsettings('usex0',1);
ops.fmincon.MaxFunEvals = 20000;
ops.fmincon.MaxIter = 200;

optimize(con,obj,ops);

%%
x1s = poslin(value(w1)*xin+value(b1));
%x2s = tansig(value(w2)*x1s+value(b2));
ys = value(w3)'*x1s+value(b3);

hold off
plot(xin,y,'Linewidth',3);
hold on
plot(xin,ys,'g--','Linewidth',4);
```

Bibliography

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Del Rey (reprint), 1995. ISBN-13: 978-0345391803.

[2] D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[3] E. G. Birgin and J. M. Martínez. *Practical augmented Lagrangian methodsPractical Augmented Lagrangian Methods*, pages 3013–3023. Springer US, Boston, MA, 2009.

[4] H. G. Bock and K.-J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608, 1984.

[5] S. E. Dreyfus. Artificial neural networks, back propagation, and the kelley-bryson gradient procedure. *Journal of Guidance, Control, and Dynamics*, 13(5):926–928, 1990.

[6] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] M. R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4(5):303–320, Nov 1969.

[8] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks, 2020.

[9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[10] E. Mizutani, S. E. Dreyfus, and K. Nishio. On derivation of mlp backpropagation from the kelley-bryson optimal-control gradient formula and its application. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 2, pages 167–172 vol.2, 2000.

[11] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com.

[12] M. J. D. POWELL. A method for nonlinear constraints in minimization problems. *Optimization*, pages 283–298, 1969.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[14] R. Sun. Optimization for deep learning: theory and algorithms, 2019.