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

Assessor:
Prof. dr. ir. Dirk Nuyens
Prof. dr. ir. Johan Suykens

Begeleider:
ir. Brecht Evens

Academiejaar 2020 – 2021

Preface

I would like to thank everybody who kept me busy the last year, especially my promoter and my assistant. 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

Huidige optimalisatiealgoritmes voor het trainen van diepe neurale netwerken zijn gebaseerd op gradiënt-afdalingsmethoden, die gebruik maken van het backpropagation algoritme om de gradiënt te berekenen. Deze algoritmes worden alom gebruikt en hebben een lange geschiedenis, maar hebben nog altijd enkele valkuilen zoals het probleem van "slechte" locale minima, waar het stationair punt gevonden door het algoritme slechter veel slechter presteert dan het globale minimum, of ook het "verdwijnende/ontploffende gradiënt" probleem. Sommige andere strategiën zijn geweest om de initialisatie te verbeteren, of om "vervroegd stoppen" aan te wenden om overfitting te vermijden.

Het trainen van een neuraal netwerk is een optimalisatieprobleem dat kan geherformuleerd worden in een context van optimale regeltechniek. Dit is een nieuw perspectief dat toelaat om backpropagation als een "single shooting" methode te zien, maar ook de gelegenheid geeft om een nieuwe oplossingsmethode voor te stellen op basis van "multiple shooting". In de theorie van regeltechniek wordt deze methode vaak aangewend voor het oplossen van zeer niet-lineaire problemen.

Het success van het backpropagation algoritme heeft er voor gezorgd dat weinig pogingen zijn gedaan om een alternatief te zoeken. In deze masterproef wordt de "multiple shooting" methode uitgewerkt. Er wordt een methode van de geaugmenteerde Lagrangiaan geimplementeerd om de haalbaarheid te onderzoeken van het trainen van neurale netwerken als een optimaal sturingsprobleem. Het nieuw algoritme wordt dan vergeleken met standaard backpropagation-optimalisatiealgoritmes die in de praktijk worden gebruikt.

Voor kleine problemen kan het nieuw algoritme concurreren zowel in termen van snelheid als qualiteit van oplossing. Voor een benaderingsprobleem dat traditioneel moeilijk trainbaar is met huidige gradiënt-afdalingsmethoden door het "afstervende ReLU" probleem, convergeert het nieuw algoritme vaker tot een goede oplossing. Het schaalt echter niet goed voor trainingsproblemen met grote datasets. Het is ook alleen toepasbaar op benaderingsproblemen met "mean squared error" als loss functie. In toekomstig onderzoek kunnen andere algoritmes onderzocht worden in plaats van de geaugmenteerde Langrangiaan methode. Ook kan het algoritme uitgebreid worden om andere loss functies te aanvaarden, wat het breder toepasbaar zou maken. Ten slotte kan een batch training modus het algoritme schaalbaarder maken voor grote datasets.

Abstract

Current optimization algorithms for training deep neural network models are based on gradient descent methods, which use the backpropagation algorithm calculate the gradient. These established algorithms have a long history and work well in practice, but still have some pitfalls such as "bad" local minima, where the stationary point found by the algorithm has worse performance than the global minimum, or also the "vanishing/exploding gradient" problem. Most improvements to these algorithms have been by refining the gradient descent algorithm such as by adding a momentum facor, or by Nesterov acceleration. Some other strategies have been to improve the initialization or to use "early stopping" to avoid overfitting.

The optimization problem of neural network training can be reformulated in an optimal control theory context. This is a perspective which allows one to view backpropagation as a "single shooting" method, but also gives the opportunity to propose new solution methods using "multiple shooting". In control theory this approach often works well for highly nonlinear problems.

The success of the backpropagation algorithm means few attempts have been made to propose an alternative. In this thesis the "multiple shooting" method is investigated. An augmented Lagrangian method is implemented to evaluate the feasibilty of solving neural network training as an optimal control problem using "multiple shooting". The novel algorithm is then compared to industry standard backpropagation algorithms.

For small problems the novel algorithm shows performance and speed comparable to current standard algorithms. A regression problem which is difficult to train for traditional gradient descent algorithms due to a "dying ReLU" problem, converges more often using the novel algorithm. However it scales poorly for training problems with large data sets. It also only works on regression problems using mean squared error loss. In future research different algorithmic frameworks can be explored instead of the Augmented Lagrangian method. ALso expanding the algorithm to handle different loss functions would make it more widely applicable, and implementing a batch training mode would improve scalability.

List of Abbreviations and Symbols

Abbreviations

AD	Algorithmic Differentiation
ADAM	ADaptive Moment Estimation
AL	Augmented Lagrangian
ALM	Augmented Lagrangian Method
ANN	Artificial Neural Network
BP	Backpropagation
DNN	Deep Neural Network
GD	Gradient Descent
LS	Least Squares
MS	Multiple Shooting
MSE	Mean Squared Error
NLP	Non-Linear Program
OCP	Optimal Control Problem
SGD	Stochastic Gradient Descent

Symbols

$\mathcal{L}_\beta$	β-Augmented Lagrangian
$\mathcal{N}(\mu, \sigma)$	Normal distribution with mean μ, std. dev. σ

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:

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

where w are the weights, x 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 ($\tanh(x)$). TODO: explain properties, show plot

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.

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: $x_i \in \mathbb{R}^{d_x}, y_j \in \mathbb{R}^{d_y}, i = 1, \ldots, n$ 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:

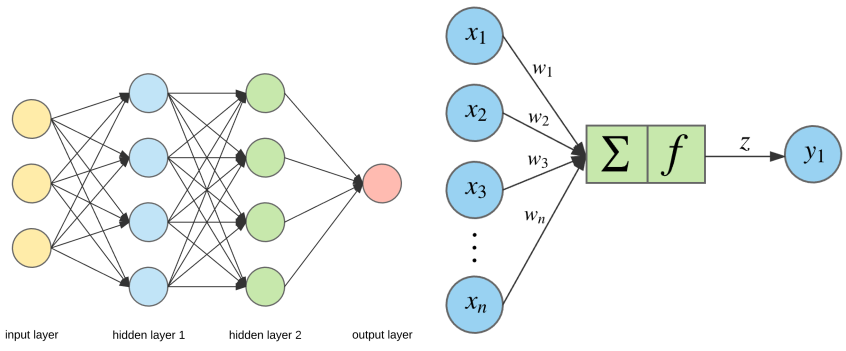
$$f_W(x) = \sigma_L(W_L\sigma(W_{L-1}\sigma_1(...W_1\sigma_0(W_0x)...))) \tag{1.3}$$

where L is the number of hidden layers in the network and σ_k is the activation function in each layer. In this equation, and also in the rest of the thesis, the bias vectors b_k have been included in the weight matrixes:

$$W_k = \begin{bmatrix} W_{k,orig} & b_k \end{bmatrix} \tag{1.4}$$

where W_k are the matrixes of dimension $d_k \times (d_{k-1} + 1), k = 1 \ldots L$ containing the connection weights extended with the bias vector b_k. Therefore the matrix multiplication in function 1.3 implicitly also contains the operation of extending the input vector with a constant 1 value. This simplifies the notation throughout the thesis

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



(A) Deep Neural Network (B) Single Neuron

Figure 1.1: Feedforward Deep Neural Network and Single Neuron - McCulloch-Pitts model [?].

$$\begin{aligned} z_0 &= x \\ z_{k+1} &= \sigma_k(W_k z_k), \quad k = 0, ..., L \\ f_W(x) &= z_{L+1} \end{aligned} \tag{1.5}$$

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:

$$\underset{W}{\text{minimize}} \quad C(W) = \sum_{i=0}^{n} l(y_i, f_W(x_i)) \tag{1.6}$$

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:

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

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.

TODO: explain Stochastic Gradient Descent, variants such as ADAM, rmsProp

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 stage cost at stage s and the terminal cost, respectively. In neural network training the stage 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:

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

$$\begin{aligned} \underset{W}{\text{minimize}} \quad & \sum_{j=0}^{n} ||\sigma_L(W_L z_L) - y_j||_2^2 \\ \text{subject to} \quad & z_{1,j} = \sigma_0(W_0, x_j), \qquad j = 1, \dots, n \\ & z_{k+1,j} = \sigma_k(W_k z_{k,j}), \qquad k = 1, \dots, L-1, j = 1, \dots, n \end{aligned} \tag{1.10}$$

This notation actually constructs j parallel dynamical systems with the same weights, but different inputs, outputs and states. The cost function then concatenates them all together [**?**].

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:

$$C(y, W_L \sigma(W_{L-1} ... \sigma(W_1 \sigma(W_0 x)))) \tag{1.11}$$

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:

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

The gradient ∇ is the transpose of the derivative.

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

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":

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

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 $\mathcal{O}(W^2 L)$ 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 multiple 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 MS problem more efficiently, which will be coded in python.

The code will be compared to common Gradient Descent methods 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. [5], Sec. 8.2).

1.7 Structure of the Thesis

In chapter 2 the multiple shooting method will be investigated using MATLAB, and compared on some test problems against a gradient descent method.

In chapter 3 an Augmented Lagrangian (AL) Method will be developed. The algorithm will be explained and a nonlinear least squares(LS) problem at the heart of the algorithm will be examined. The Jacobian matrix associated with the LS problem will be analytically derived and presented.

In chapter 4 numerical tests will be applied to evaluate the performance of the AL method. It will be compared to the industry standard ADAM optimizer, which is a type of Gradient Descent method using backpropagation.

Finally in chapter 5 the conclusions of the thesis and possible future work will be presented.

Appendix A lists a collection of MATLAB code used in the thesis.

Appendix B lists a collection of python code used in the thesis.

The full source code can be downloaded from https://github.com/jan-scheers/thesis/.

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:

$$\begin{aligned} \underset{W}{\text{minimize}} \quad & \sum_{j=0}^{n} ||\sigma_L(W_L z_L) - y_j||_2^2 \\ \text{subject to} \quad & z_{1,j} = \sigma_0(W_0, x_j), \qquad j = 1, \dots, n \\ & z_{k+1,j} = \sigma_k(W_k z_{k,j}), \qquad k = 1, \dots, L-1, j = 1, \dots, n \end{aligned} \tag{2.1}$$

This is a nonlinear program with nonlinear equality constraints. It is 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 **??**. 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 KU Leuven []. [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.

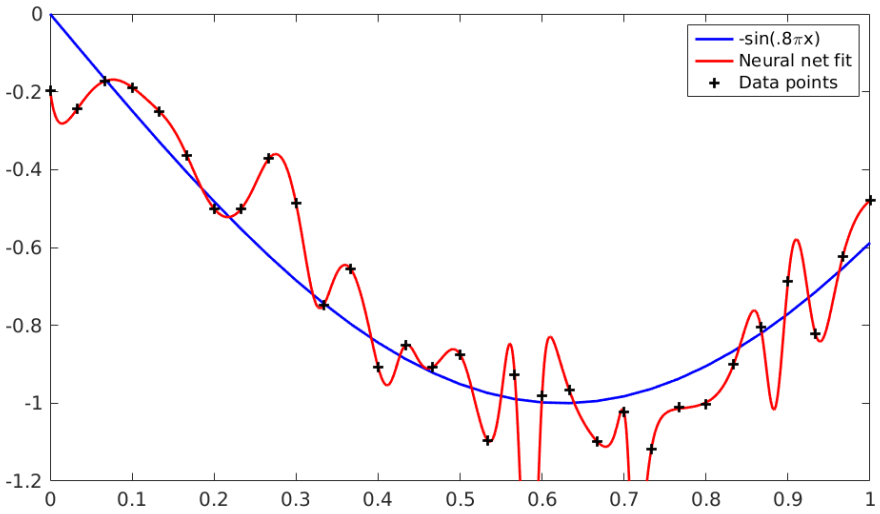
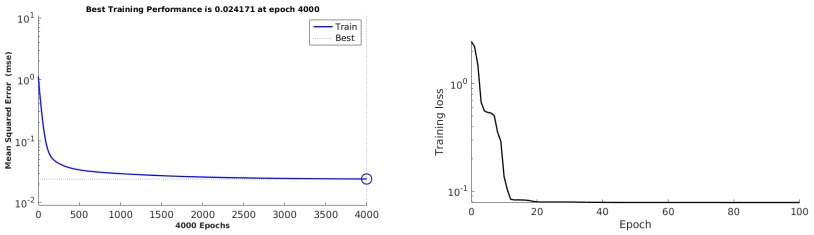
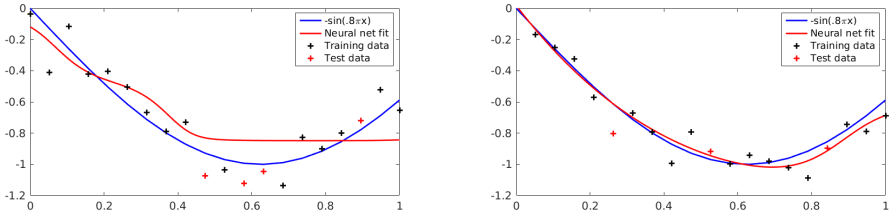


FIGURE 2.1: Test function approximated with `trainlm` training algorithm. The data has been overfitted.



(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. This is a common method for solving constrained Non Linear Programs (NLPs). Instead of using the classical method, an Augmented Lagrangian framework is adapted from a recent paper[11].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 [6],[9] and was known as the method of multipliers. The textbook examples of this method can be found in [4] and [3].

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

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

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:

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

where $\beta > 0$ is the penalty weight. This can be viewed as a penalty method which has been shifted using the term in λ[4]. When β or λ tend to infinity, $h(u)$ will be forced to zero, leading the Lagrangian to converge to the same solution as the original problem.

The algorithm proceeds as follows:

$$u_{k+1} = \underset{u}{\operatorname{argmin}}\ \mathcal{L}_\beta(u, \lambda_k)$$
$$\lambda_{k+1} = \lambda_k + \sigma_k h(u_{k+1})$$

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:

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

with τ_1, τ_2 the chosen tolerances.

3.2 Applied Augmented Lagrangian Method

The OCP equation 1.10 of training a neural net with MSE loss function is a constrained nonlinear least squares(LS) problem:

$$\begin{aligned}
&\underset{W}{\text{minimize}} && \sum_{j=0}^{n} ||\sigma_L(W_L z_L) - y_j||_2^2 \\
&\text{subject to} && z_{1,j} = \sigma_0(W_0, x_j), && j = 1, \dots, n \\
& && z_{k+1,j} = \sigma_k(W_k z_{k,j}), && k = 1, \dots, L-1, j = 1, \dots, n \\
& \Updownarrow \\
&\text{min} && \frac{1}{2}||F(u)||_2^2 \\
&\text{s. t.} && h(u) = 0
\end{aligned}$$

Where $u = \{W, z\}$ is the collection of both the weight and state variables into a single vector.

The subproblem that will be solved in each iteration is then:

$$\begin{aligned}
\underset{u}{\operatorname{argmin}} \quad \mathcal{L}_\beta(u, \lambda) &= \frac{1}{2}||F(u)||_2^2 + \langle \lambda, h(u) \rangle + \frac{\beta}{2}||h(u)||_2^2 \\
&= \frac{1}{2}||F(u)||_2^2 + \frac{\beta}{2}||h(u) + \lambda/\beta||_2^2 - \frac{1}{2\beta}||\lambda||_2^2 \\
&= \frac{\beta}{2}\left|\left|\begin{bmatrix} F(u)/\sqrt{\beta} \\ h(u) + \lambda/\beta \end{bmatrix}\right|\right|_2^2
\end{aligned} \tag{3.4}$$

Instead of using the textbook algorithm, an algorithmic framework from a more recent paper[11] is adapted to the problem, shown in Algorithm 1.

Algorithm 1: Inexact Augmented Lagrangian Method

Input: Initial weights vector W, penalty parameter β, stopping tolerance τ, input-target pairs $(x_i, y_i), i = 1, \ldots, n$

Initialization $u_0 = \{W, f_W(x)\}, \lambda_0 \in \mathcal{N}(0,1)$; *Initialize state variables by simulating network, initialize dual variables randomly*

for k = *0,1,...* **do**

$\eta_k = 1/\beta^k$; *Update tolerance*

find u_{k+1} such that

$||\nabla_{u_k} \mathcal{L}_{\beta^k}(u_k, \lambda_k)|| \leq \eta_k$; *Approx. primal solution*

$\sigma_{k+1} = \min(\frac{||h(u_0)|| \log^2 2}{||h(u_{k+1})|| k \log^2(k+1)}, 1)$; *Update dual step size*

$\lambda_{k+1} = \lambda_k + \sigma_{k+1} h(u_{k+1})$; *Update dual variables*

$||\nabla_{u_{k+1}} \mathcal{L}_{\beta^k}(u_{k+1}, \lambda_k)|| + ||h(u_{k+1})|| < \tau$; *Stopping Criterion*

end

The penalty parameter increases geometrically, $\beta_k = \beta_0^k$, and the tolerence decreases geometrically $\eta_k = 1/\beta_k$. It is called the inexact Augmented Lagrangian Method (iALM) because the optimizer u^* of subproblem 3.4 can only be found to an approximate solution. The choice of dual step size σ_k is to ensure the boundedness of the dual variables λ_k [11],[2].

Figure 3.1 shows the convergence behaviour of this algorithm. In this figure the algorithm was run for 10 epochs on a neural network training problem. The gradient of the β-Augmented Lagrangian is plotted in blue.

$$||\nabla_{u_k} \mathcal{L}_{\beta^k}(u_k, \lambda_k)|| = ||2(\nabla_{u_k} \begin{bmatrix} F(u)/\sqrt{\beta} \\ h(u) + \lambda/\beta \end{bmatrix}) \mathcal{L}_{\beta^k}(u_k, \lambda_k)|| \tag{3.5}$$

The gradient decreases geometrically as the tolerence is decreased in each step $\eta_{k+1} = \eta_k/\beta$. The MSE loss of the network is plotted in red. It is calculated by taking the current optimal weights at that epoch and simulating the network on the training data. In this example the MSE loss reaches a minimum after 6 iterations, which is a typical result.

The constraint violations, and the variables associated with the states are not relevant when evaluating the performance of the network. For this reason the stopping criterion in Algorithm 1 may not be the most practical choice. In deep learning many different stopping criteria are used. Often these are based on the training loss, or the loss on a validation set which is held apart from the training data. Usually a tradeoff will have to be made between training performance and overfitting the data (Goodfellow et al. [5], Sec. 8.1). This is a practical issue, in the next chapter the problem of choosing an appropriate stopping criterion is examined more fully.

3.3 Least Squares Solver

To solve the LS problem 3.4 in Algorithm 1, a Trust Region Reflective(`trf`) method is used, which is implemented in `scipy.optimize.least_squares`. The following

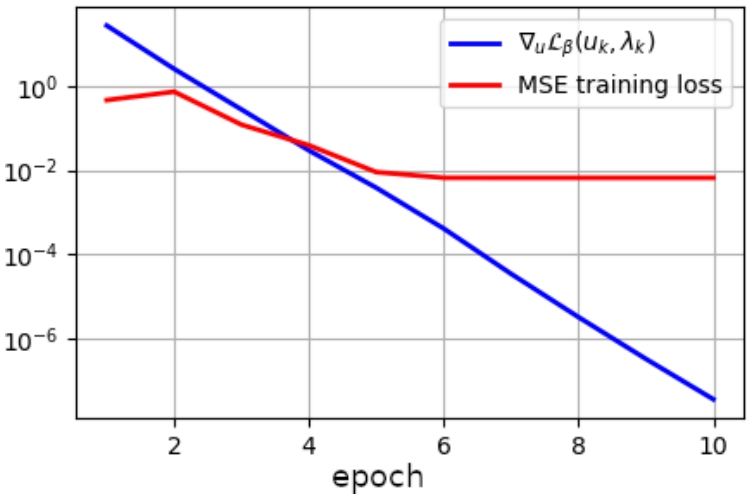


Figure 3.1: Typical convergence behaviour of Algorithm 1

description is given by `scipy`: "The algorithm iteratively solves trust-region subproblems augmented by a special diagonal quadratic term and with trust-region shape determined by the distance from the bounds and the direction of the gradient."[1]. The `trf` method is described as being robust for both bounded and unbounded problems, and well suited for sparse Jacobians.

The LS problem 3.4 is an unbounded problem for which the library recommends using a Levenberg-Marquardt method. However this method cannot handle cases where the Jacobian has more columns than rows, which can sometimes occur depending on the size of the network and the number of data points used for training. Therefore we cannot use the Levenberg-Marquardt algorithm.

To efficiently solve the least squares problem, the solver requires an analytical solution for the Jacobian, which will be explained in the next section.

3.4 Jacobian

To solve the least squares problem, the Jacobian matrix of $M_\beta(u, \lambda) = \begin{bmatrix} F(u)/\sqrt{\beta} \\ h(u) + \lambda/\beta \end{bmatrix}$ must be calculated. A Jacobian is the matrix of all partial derivatives of a vector valued function. In this case:

$$J_{M_\beta} = \begin{bmatrix} \frac{\partial M_\beta}{\partial u_1} & \frac{\partial M_\beta}{\partial u_2} & \dots & \frac{\partial M_\beta}{\partial u_n} \end{bmatrix} \tag{3.6}$$

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. In this section the partial derivative associated with each variable will be presented.

First the columns of J_{M_β} associated with the weight variables will be examined:

$$\frac{\partial M_\beta}{\partial W_k} = -z_{k,j}\sigma_k'(W_k z_{k,j}), k = 0, \dots, L, j = 1, \dots, n \tag{3.7}$$

W_k is a matrix of size $d_{k+1} \times (d_k + 1)$, and $z_{k,j}$ are vectors of size $d_k + 1$, therefore this evaluates as a 3D tensor. W_k must be vectorized first to allow this derivative to be used in the Jacobian. After vectorization the dimensions of this partial derivative as a block matrix are $d_{k+1}n \times d_{k+1}(d_k + 1)$. The last partial derivative $\frac{\partial M_\beta}{\partial W_L}$ is also multiplied by a factor $\frac{1}{\beta}$.

Next the columns of J_{M_β} associated with the states z_k are examined:

$$\frac{\partial M_\beta}{\partial z_k} = \begin{bmatrix} 1 \\ -W_k \sigma_k'(W_k z_k) \end{bmatrix}, k = 1, \ldots, L \tag{3.8}$$

W_k are as before and z_k are matrices of size $(d_k + 1) \times n$. The row of ones associated with the biases vector in W_k is not a variable so it is excluded from the jacobian. After vectorization of z_k the partial derivative has dimensions $(d_k + d_{k+1})n \times d_k n$.

For a fully connected neural network with identity output activation an example has been written out in Table 3.1. Figure 3.2a shows a visual representation of the matrix, where the nonzero elements have been colored black. An alternative representation, which is mathematically the same is to swap the rows corresponding to the loss function $F(u)$ to the bottom. This gives a matrix with a banded structure, which is is plotted in figure 3.2b. This is the representation used in the code.

Given a feedforward network with input dimension I, an output dimension O, and it 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 $M_\beta(u, \lambda)$ has an output dimension of $D \times W \times N + O \times N$. The dimension of the Jacobian for this network is therefore $(D \times W \times N + O \times N) \times (D \times W \times N + O + (D + I + O) \times W + (D-1) \times W^2)$. The Jacobian scales quadratically in size with the depth of the network and the number of datapoints. It scales cubically with the width of the network. The Jacobian will be taller than it is wide when $N \geq 1 + (D + I + O) \times W/O + (D-1) \times W^2/O$.

3.5 Numerical verification of Jacobian Matrix

In the previous section the Jacobian matrix was derived analytically. In this section will be explained how the Jacobian is verified algorithmically.

Algorithmic Differentation (AD) is a set of techniques which can be used to calculate the derivative of any computer code [10]. Because all code is composed of elementary operations, AD can use the chain rule alongside the operations to automatically compute derivatives of arbitrary order. By injecting code from an AD library into the calculation of the neural network, the Jacobian can be calculated numerically. For this the AlgoPy python library was used. The output of the AD 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 shown in Appendix B.1.1

Weight variables, each entry is a block matrix

	$\nabla^T_{W_0,b_0} M$	W_{0_1}	W_{0_2}	...	W_{0_W}	b_0
	dim	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

	$\nabla^T_{W_i,b_i} M$	W_{i_1}	W_{i_2}	...	W_{i_W}	b_i
	dim	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

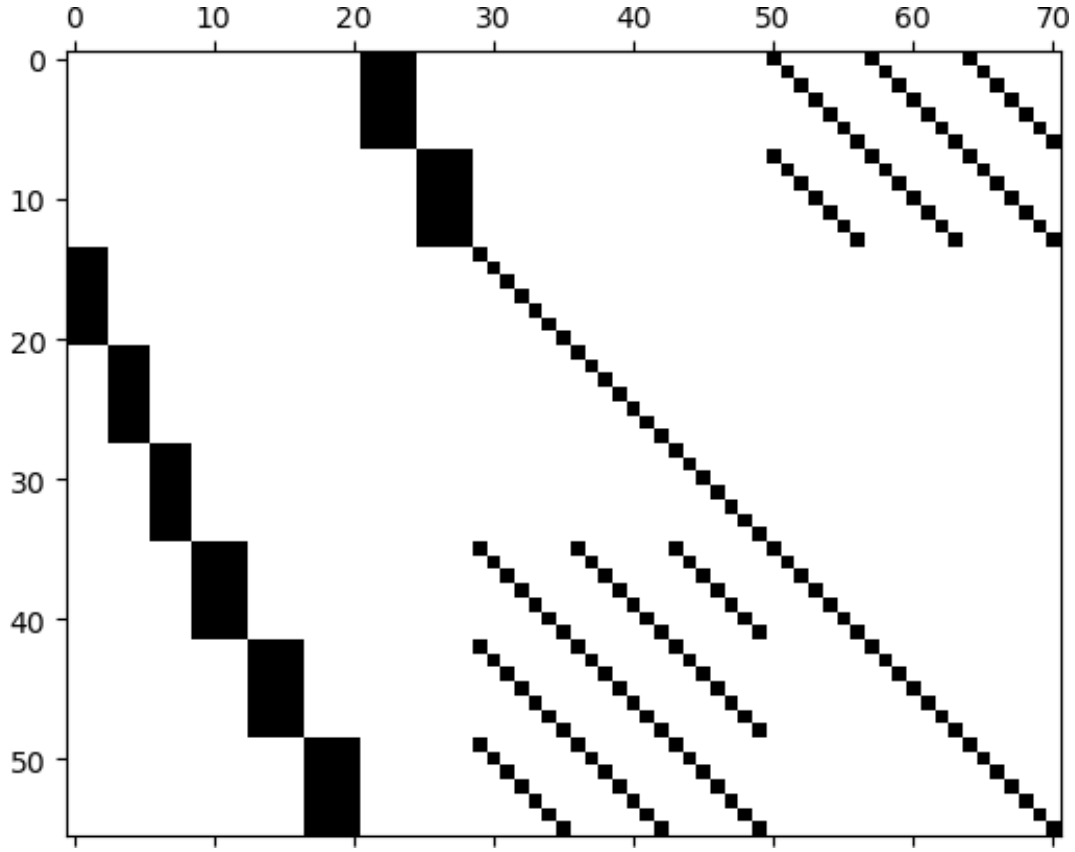
	$\nabla^T_{W_D,b_D} M$	W_{D_1}	W_{D_2}	...	W_{D_O}	b_D
	dim	W	W	...	W	O
F	N	$-\frac{z_D}{\sqrt{c}}\sigma'_O(W_{D_1}x + b_{D_1})$	0	...	0	$-\frac{1}{\sqrt{c}}\sigma'_O(W_{D_1}x + b_{D_1})$
	N	0	$-\frac{z_D}{\sqrt{c}}\sigma'_O(W_{D_2}x + b_{D_2})$	...	0	$-\frac{1}{\sqrt{c}}\sigma'_O(W_{D_2}x + b_{D_2})$
	...	...	...	...	...	...
	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

State variables, each entry is a diagonal matrix

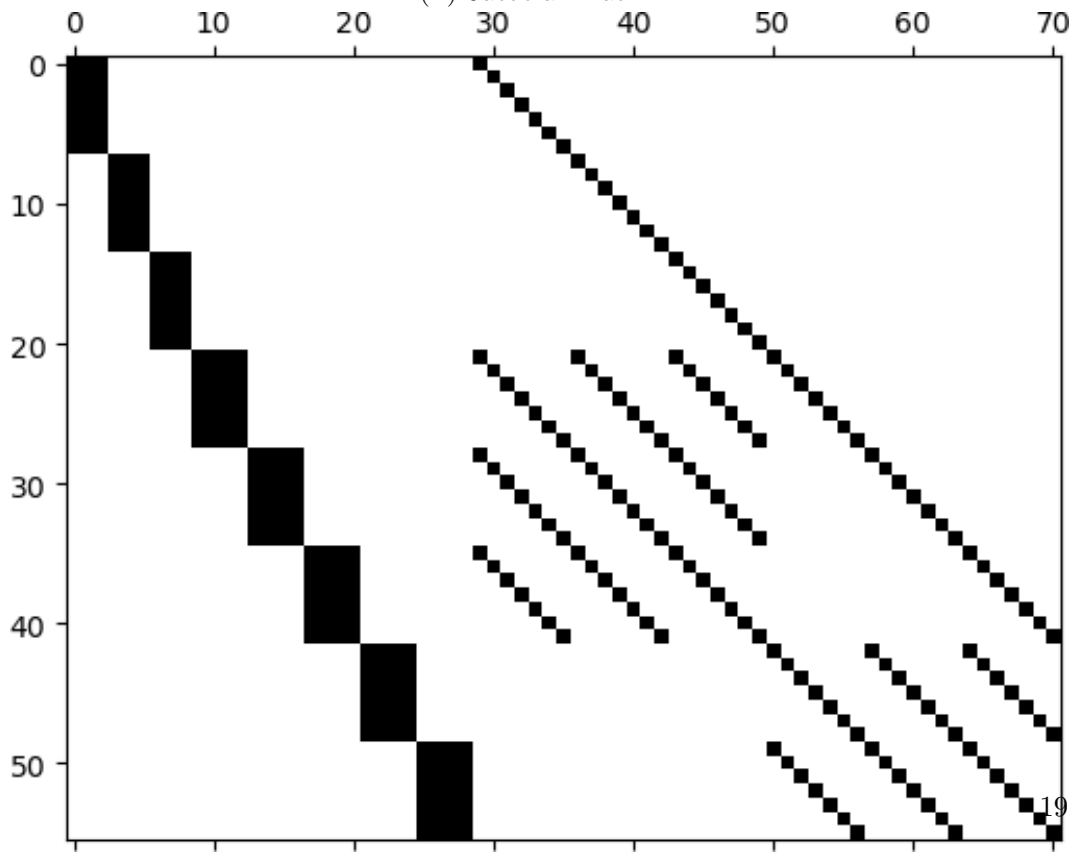
	$\nabla^T_{z_i} M$	z_{i_1}	z_{i_2}	...	z_{i_W}
	dim	N	N	...	N
F	O*N	0	0	...	0
h_1	W*N	0	0	...	0
...	...	...	...	...	...
h_i	N	1	0	...	0
	N	0	1	...	0
	...	...	...	...	...
	N	0	0	...	1
h_{i+1}	N	$-W_{i_{1,1}}\sigma'(W_{i_1}z_i + b_{i_1})$	$-W_{i_{1,2}}\sigma'(W_{i_1}z_i + b_{i_1})$	...	$-W_{i_{1,W}}\sigma'(W_{i_1}z_i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

	$\nabla^T_{z_D} M$	z_{D_1}	z_{D_2}	...	z_{D_W}
	dim	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. In this table the biases are not included in the weight matrices W_k. Each layer has a the same width W and same activation σ. There are D layers. The input dimension is I and the output dimension is O. There are N datapoints.



(A) Jacobian matrix



(B) Jacobian matrix, rearranged

FIGURE 3.2: Visual representation of Jacobian matrix for network with 2 inputs, 2 outputs, width 3, depth 2 and 7 datapoints. The non-zero elements have been colored black. The jacobian at the top is the same as the one on the bottom, but with swapped rows

3.6 Conclusion

In this chapter an Augmented Lagrangian framework was proposed to solve the Optimal Control Problem. A Least Squares solver was applied to the LS problem at the heart of the algorithm, and a Jacobian Matrix was analytically derived which is supplied to the solver. The Jacobian was also algorithmically verified. In the next chapter the algorithm will be investigated using numerical tests and compared against an industry standard optimizer.

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 training performance of ALM will be compared to the ADaptive Moment Estimation (ADAM) optimizer, which is a type of Stochastic Gradient Descent (SGD) method. This optimizer is widely used and is generally known to be robust (Goodfellow et al. [5], Sec. 8.5.3). Even though it is usually used in a mini-batch manner, the datasets in these problems are so small that the batch size is set to be as large as the dataset. This effectively makes it a true Gradient Descent method instead of an SGD. The learning rate is left at the default 0.001.

The weights of the network will be initialized using Xavier initialization for layers using the $\tanh(x)$ activation function and Kaiming initialization for layers using the ReLU activation function. Both training algorithms will start from the same initial point for each test. ADAM will halt when 10 epochs have passed without improvement, indicating a local minimum has been reached. For ALM the training will halt after a single epoch has passed without significant improvement:

$$(1+\epsilon)F(u_{k+1}) > F(u_k) \tag{4.2}$$

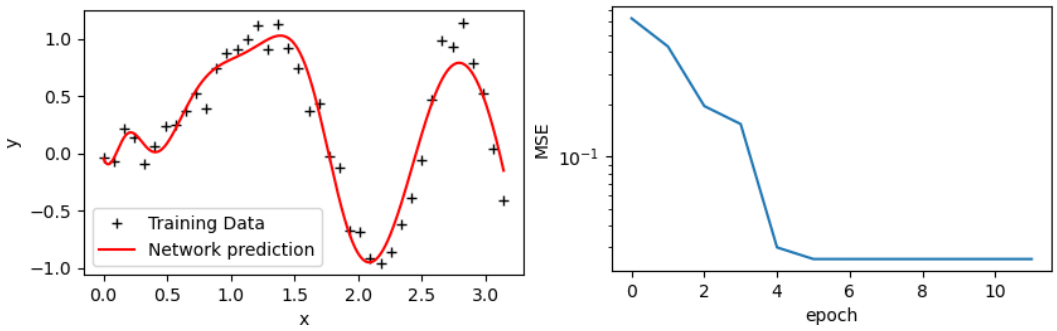
Where $F(u_k)$ is the loss at epoch k and ϵ is chosen to be $1e^{-2}$.
All tests are run on .... , GBS,

4.3 Fully connected feedforward network

For the first comparison a similar regression problem as in Chapter 2 will be considered. The sine function in equation 2.3 is quite simple, therefore a problem with a more depth is considered: a squared sine function. The function definition is as follows:

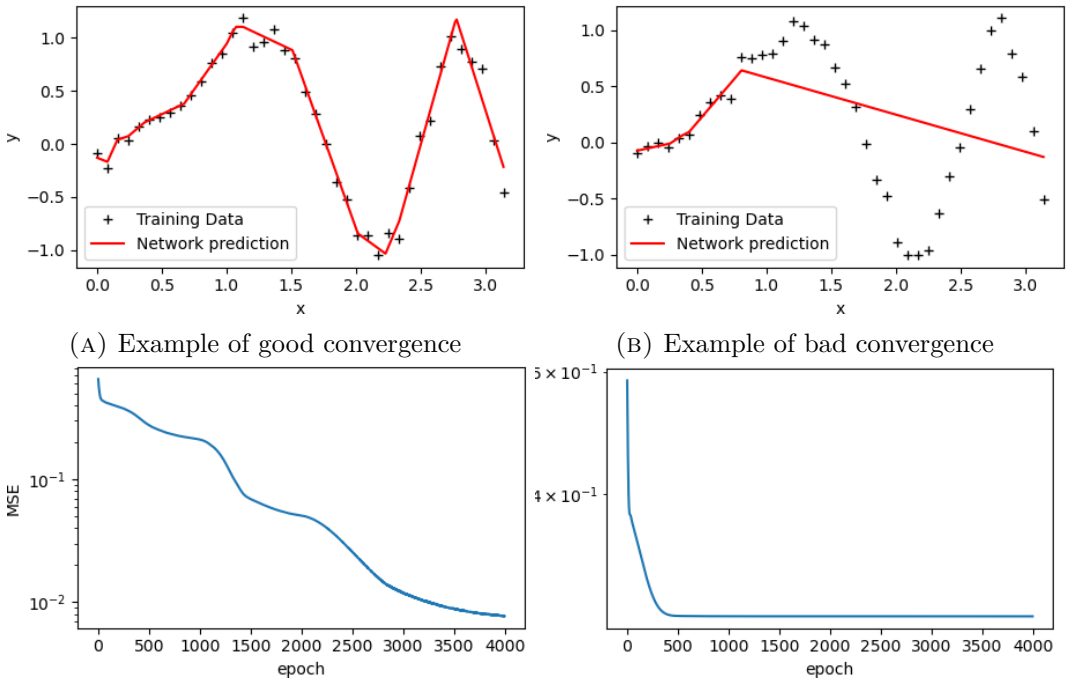
$$y = \sin^2(x) + \mathcal{N}(0,\delta), x \in [0,\pi] \tag{4.3}$$

This function oscillates progressively faster, and the training algorithm will not always find a good solution. Figure 4.2 shows two examples runs of the ADAM



(A) Example of good convergence (B) Training history of convergence

FIGURE 4.1: Example of convergence behaviour of ALM optimizer for network using *tanh* activation and 40 data points. The network has 2 hidden layers of 8 nodes. An example of bad convergence could not be found for the *tanh* activation function using ALM.



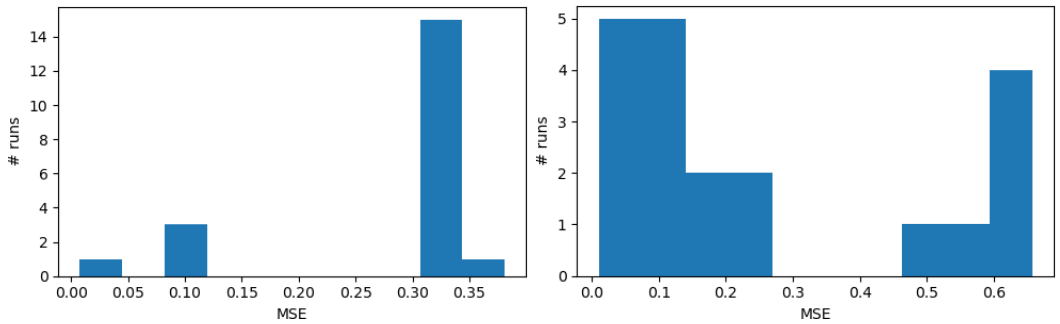
(A) Example of good convergence (B) Example of bad convergence

(C) Training history of good convergence (D) Training history of bad convergence

FIGURE 4.2: Examples of convergence behaviour of ADAM optimizer for network using ReLU activation and 40 data points. The network has 2 hidden layers of 16 nodes.

	$\sigma(\cdot) = \tanh(\cdot)$						$\sigma(\cdot) = \max(\cdot, 0)$					
	N	avg MSE	best MSE	time(s)	epochs	conv	N	avg MSE	best MSE	time(s)	epochs	conv
ADAM	10	$2.09e^{-2}$	$1.20e^{-11}$	18.7	3790	17	20	$1.22e^{-1}$	$3.15e^{-3}$	30.3	3500	5
ALM		$2.10e^{-1}$	$1.59e^{-1}$	1.85	6.38	16		$1.63e^{-1}$	$1.08e^{-2}$	4.41	5.47	15
ADAM	20	$1.25e^{-2}$	$9.84e^{-4}$	19.8	3960	17	40	$6.28e^{-2}$	$8.85e^{-3}$	22.4	2690	5
ALM		$5.87e^{-2}$	$7.69e^{-3}$	8.06	7.50	20		$9.89e^{-2}$	$1.37e^{-2}$	8.09	5.11	18
ADAM	40	$2.56e^{-2}$	$7.58e^{-3}$	19.2	3920	15	80	$7.61e^{-2}$	$7.64e^{-3}$	29.9	3150	4
ALM		$1.61e^{-2}$	$7.90e^{-3}$	11.6	6.90	20		$1.02e^{-1}$	$1.03e^{-2}$	25.3	4.21	14

Table 4.1: Results of tests using tanh and ReLU activation function.



(a) Histogram of MSE of test batch using ADAM optimizer and 40 data points

(b) Histogram of MSE of test batch using ALM optimizer and 40 data points

Figure 4.3: Histograms of MSE batch using 40 data points and ReLU activation function. Both histograms show a bimodal distribution

optimizer for 4000 epochs. The only difference is the choice of initial weights and the random noise. The first example has converged well, and would probably improve even further if the number of epochs was increased. The second example has stagnated in a bad minimum. Figure 4.1 shows an example run of the ALM optimizer for 12 epochs using the *tanh* activation function. An example of bad convergence using this activation could not be found.

For the first experiment two similar networks are trained to approximate function 4.3. The first network has 2 hidden layers of 8 nodes with *tanh* activation function, the second has 2 hidden layers of 16 nodes with a ReLU activation function. Each is trained with progressively more data points. Every single configuration is run 20 times. The results are listed in table 4.1. The code from this test is found in appendix B

The last column of each subtable in table 4.1 shows the number of runs per batch which converged to a acceptable solution, which is defined as having an MSE of less than 0.25. For reference, the MSE of a straight line $y = 0$ has an MSE of approximately 0.55. This choice is based on experimental results: figure 4.3 shows the histograms of the MSE of both methods for the test batch using ReLU activation function and 40 data points. They show a clear bimodal distribution, differentiating those runs which converged, and those which did not.

The avg MSE, time and epochs has only been calculated over those runs which converged. In the best runs, ADAM typically beats ALM in performance, and could probably even improve further given more epochs. However on average the performance is similar, and ALM converges more often than ADAM.

The network using ReLU activation function is especially difficult to train, as ADAM only converges 14 times in 60 total runs. In comparison ALM converges 47 times. This may be expected as the main strength of ReLU is found in large networks. In this network the "dying ReLU" problem seems to appear often [8]. Because the derivative of ReLU is zero for all negative inputs, some nodes may get stuck and receive no gradient updates in a Gradient Descent method such as ADAM. On the other hand, the extra degrees of freedom in the ALM method may allow it to avoid this problem more often.

Another point is that the run time of ALM scales approximately quadratically with the amount of data points N, while the runtime of ADAM stays constant. This is because the size of the Jacobian in ALM scales approximately quadratically with N, see equation **??**. In backpropagation the only factor which scales with N is the computation of the loss in the forward pass, which must be executed for every sample. This test has a small dataset, for which the computation of the loss is small compared to the other parts of the ADAM algorithm. In Big Data problems this cost is mitigated by using mini-batches to compute an approximate gradient instead of computing over the whole dataset. This idea could be applied to the ALM method, but this is left for future work.

- [7] "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"
-

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 [5].

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 experiment in Chapter 1

A.1 First experiment source

```
R = zeros(20,6);
for k = 1:20

    N = 20;
    x = linspace(0,1,N);
    y = -sin(0.8*pi*x)+normrnd(0,0.1,size(x));
    [trainInd,valInd,testInd] = dividerand(N,0.8,0,0.2);


    net = fitnet([3 3]);
    net = configure(net,x,y);
    net.inputs{1}.processFcns = {};
    net.outputs{2}.processFcns = {};
    net.divideFcn = 'divideind';
    net.divideParam.trainInd = trainInd;
    net.divideParam.valInd = valInd;
    net.divideParam.testInd = testInd;


    N = 0.8*N;
    xin = x(trainInd);

    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(1,3);
    b3 = sdpvar(1,1);


    assign(w1,net.IW{1});
    assign(b1,net.b{1});
    assign(x1,tansig(value(w1*xin+repmat(b1,1,N))));
    assign(w2,net.LW{2,1});
    assign(b2,net.b{2});
    assign(x2,tansig(value(w2*x1 +repmat(b2,1,N))));
    assign(w3,net.LW{3,2});
    assign(b3,net.b{3});

    res = w3*x2 + b3 - y(trainInd);
    obj = res*res';

    net.trainFcn = 'traingd';
    net.trainParam.epochs = 2000;
    net.trainParam.max_fail = 200;
    [net, tr] = train(net,x,y);

    con = [x1 == tansig(w1*xin+repmat(b1,1,N));
           x2 == tansig(w2*x1 +repmat(b2,1,N))];
    ops = sdpsettings('usex0',1,'solver','fmincon');
    ops.fmincon.MaxFunEvals = 20000;
    ops.fmincon.MaxIter = 40;

    t0 = cputime;
    optimize(con,obj,ops);
    t1 = cputime-t0;

    x1s = tansig(value(w1)*x(testInd)+value(b1));
    x2s = tansig(value(w2)*x1s+value(b2));
    ysm = value(w3)*x2s+value(b3);

    R(k,:) = [tr.perf(end),tr.tperf(end),tr.time(end),value(obj),immse(ysm,y(testIn
end
%%
% figure(2);
% hold off
%
% plot(x,-sin(.8*pi*x),'b-','Linewidth',2)
% hold on;
%
```

```
% xsm = linspace(0,1,1000);
% x1s = tansig(value(w1)*xsm+value(b1));
% x2s = tansig(value(w2)*x1s+value(b2));
% ysm = value(w3)*x2s+value(b3);
%
% plot(xsm,ysm,'r-','Linewidth',2);
%
% plot(x(trainInd),y(trainInd),'k+','Linewidth',2,'Markersize',8);
% plot(x(testInd) ,y(testInd) ,'r+','Linewidth',2,'Markersize',8);
%
% ylim([-1.2,0])
%
% legend("-sin(.8\pix)","Neural net fit","Training data","Test data")
```

A.1.1 Second experiment source

```
R = zeros(20,6);
for k = 1:20

    N = 20;
    x = linspace(0,1,N);
    y = -sin(0.8*pi*x)+normrnd(0,0.1,size(x));
    [trainInd,valInd,testInd] = dividerand(N,0.8,0,0.2);


    net = fitnet([8 8]);
    net.inputs{1}.processFcns = {};
    net.outputs{2}.processFcns = {};
    net.layers{1}.transferFcn = 'poslin';
    net.layers{2}.transferFcn = 'poslin';
    net.divideFcn = 'divideind';
    net.divideParam.trainInd = trainInd;
    net.divideParam.valInd = valInd;
    net.divideParam.testInd = testInd;
    net = configure(net,x,y);


    N = 0.8*N;
    xin = x(trainInd);

    w1 = sdpvar(8,1);
    b1 = sdpvar(8,1);
    x1 = sdpvar(8,N);
    w2 = sdpvar(8,8,'full');
    b2 = sdpvar(8,1);
```

```
    x2 = sdpvar(8,N);
    w3 = sdpvar(1,8);
    b3 = sdpvar(1,1);

    assign(w1,net.IW{1});
    assign(b1,net.b{1});
    assign(x1,tansig(value(w1*xin+repmat(b1,1,N))));
    assign(w2,net.LW{2,1});
    assign(b2,net.b{2});
    assign(x2,tansig(value(w2*x1 +repmat(b2,1,N))));
    assign(w3,net.LW{3,2});
    assign(b3,net.b{3});

    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];

    res = w3*x2 + b3 - y(trainInd);
    obj = res*res';

    net.trainFcn = 'traingd';
    net.trainParam.epochs = 2000;
    net.trainParam.max_fail = 200;
    [net, tr] = train(net,x,y);

    ops = sdpsettings('usex0',1,'solver','fmincon');
    ops.fmincon.MaxFunEvals = 20000;
    ops.fmincon.MaxIter = 40;

    t0 = cputime;
    optimize(con,obj,ops);
    t1 = cputime-t0;

    x1s = poslin(value(w1)*x(testInd)+value(b1));
    x2s = poslin(value(w2)*x1s+value(b2));
    ysm = value(w3)*x2s+value(b3);

    R(k,:) = [tr.perf(end),tr.tperf(end),tr.time(end),value(obj),immse(ysm,y(testIn
end
%%
% figure(2);
% hold off
%
% plot(x,-sin(.8*pi*x),'b-','Linewidth',2)
```

```
% hold on;
%
% xsm = linspace(0,1,1000);
% x1s = tansig(value(w1)*xsm+value(b1));
% x2s = tansig(value(w2)*x1s+value(b2));
% ysm = value(w3)*x2s+value(b3);
%
% plot(xsm,ysm,'r-','Linewidth',2);
%
% plot(x(trainInd),y(trainInd),'k+','Linewidth',2,'Markersize',8);
% plot(x(testInd) ,y(testInd) ,'r+','Linewidth',2,'Markersize',8);
%
% ylim([-1.2,0])
%
% legend("-sin(.8\pix)","Neural net fit","Training data","Test data")
```

Appendix B

Source code

This appendix contains the source code of the Augmented lagrangian method and the automatic verification of the Jacobian Matrix

B.1 Augmented lagrangian method

```
#alm.py
import numpy as np
import tensorflow as tf
from tensorflow import keras
from scipy import sparse
import scipy.optimize as op
import scipy.linalg as la

class Dense_d(keras.layers.Dense):
    def __init__(self,activation_,**kwargs):
        super().__init__(**kwargs)
        self.activation_ = activation_

class ALMModel:
    def __init__(self,model,x,y):
        self.model = model
        t = self.model(x)
        self.x = x
        self.y = y
        self.batch_size = x.shape[0]
        self.nz = sum([self.batch_size*model.layers[i].units for i in range(len(model.layers)-1)])

    def read(self,u):
        layers = self.model.layers
        w = []
        p = 0
        for i in range(0,len(layers)):
            s = layers[i].weights[0].shape
            m,n = s[1],s[0]+1
            w.append(u[p:p+m*n].reshape(m,n))
            p = p+m*n

        z = [self.x.transpose()]
```

```
        for i in range(1,len(layers)):
            m,n = w[i].shape[1]-1,self.batch_size
            zi = u[p:p+m*n].reshape(m,n)
            z.append(zi)
            p = p+m*n
        y = np.r_[self.y.transpose()]
        z.append(y)

        return w,z

    def write(self):
        u = np.empty(0)
        for layer in self.model.layers:
            w,b = layer.weights
            m,n = w.shape[1],w.shape[0]+1
            w = np.c_[w.numpy().transpose(),b.numpy().reshape(m,1)]
            u = np.append(u,w)
        z = [self.x]
        for i,layer in enumerate(self.model.layers):
            z.append(layer(z[i]))

        z = z[1:-1]
        z = [zi.numpy().transpose() for zi in z]
        u = np.append(u,np.concatenate(z))
        return u

    def set_weights(self,w):
        for i,layer in enumerate(self.model.layers):
            wi = w[i].transpose()
            layer.set_weights([wi[:-1,:],wi[-1:,].reshape((-1,))])

    def h(self,u):
        w,z = self.read(u)
        h = []
        for i,layer in enumerate(self.model.layers[:-1]):
            zi_e = np.r_[z[i],np.ones((1,self.batch_size))]
            hi = z[i+1] - np.array(layer.activation(w[i].dot(zi_e)))
            h.append(hi.reshape(-1))
        return np.concatenate(h)

    def L(self,u,beta,l):
        w,z = self.read(u)

        out_layer = self.model.layers[-1]
        z_e = np.r_[z[-2],np.ones((1,self.batch_size))]
        F = z[-1] - np.array(out_layer.activation(w[-1].dot(z_e)))
        F = F.reshape(-1)/np.sqrt(beta)

        h = self.h(u)+l/beta
        return np.concatenate([h,F])


    def JL(self,u,beta):
        layers = self.model.layers
        w,z = self.read(u)
```

```
    w_ = []
    z_ = []
    np.set_printoptions(linewidth=200)
    for i in range(len(layers)):
        # Add bias to weights
        zi_e = np.r_[z[i],np.ones((1,self.batch_size))]

        ai = layers[i].activation_(w[i].dot(zi_e))

        wi_ = -zi_e*ai[:,np.newaxis,:]
        wi_ = wi_*np.eye(w[i].shape[0])[:,:,np.newaxis,np.newaxis]
        wi_ = np.swapaxes(wi_,1,2).reshape(w[i].size,w[i].shape[0]*self.batch_size).transpose()

        w_.append(wi_)

        if i > 0:
            zi_ = -w[i][:,:-1].transpose()[:,:,np.newaxis]*ai
            zi_ = np.eye(self.batch_size)*zi_[:,:,:,np.newaxis]
            zi_ = np.swapaxes(zi_,1,2).reshape(z[i].size,z[i+1].size).transpose()
            z_.append(zi_)


    w_[-1] = w_[-1]/np.sqrt(beta)
    z_[-1] = z_[-1]/np.sqrt(beta)

    w_ = [sparse.csc_matrix(wi_) for wi_ in w_]
    z_ = [sparse.csc_matrix(zi_) for zi_ in z_]

    z_ = sparse.block_diag(z_)
    z_ = sparse.vstack((sparse.csr_matrix((w_[0].shape[0],z_.shape[1])),z_))
    z_ = z_+sparse.eye(z_.shape[0],z_.shape[1])
    w_ = sparse.block_diag(w_)
    J = sparse.hstack((w_,z_))
    return J


def fit_alm(self,val_data=None,beta=10,tau=1e-2):
    l = np.random.normal(0,1,self.nz)
    u = self.write()
    sigma_0,h_0 = 1,la.norm(self.h(u))
    hist = {"tol":np.empty(0),"njev":np.empty(0),"loss":np.empty(0),'val_loss':np.empty(0)}
    for k in range(10):
        beta_k = np.power(beta,k)
        eta_k = 1/beta_k
        fun = lambda u: self.L(u,beta_k,l)
        jac = lambda u: self.JL(u,beta_k)
        try:
            sol = op.least_squares(fun,u,jac,ftol=None,xtol=None,gtol=eta_k,tr_solver='lsmr')
        except:
            print("Divide by 0")
            break
        u = sol.x

        w,_ = self.read(u)
```

```
            self.set_weights(w)

            h = self.h(u)

            sigma = sigma_0*np.amin([h_0*np.power(np.log(2),2)/la.norm(h)/(k+1)/np.power(np.log(k+2),
              1])
            l = l + sigma*h

            # Convergence of Lagrangian function
            jac = self.JL(u,beta_k).toarray()
            tol = la.norm(2*self.JL(u,beta_k).transpose()*self.L(u,beta_k,l))+la.norm(h)
            hist['tol'] = np.append(hist['tol'],tol)
            hist['njev'] = np.append(hist['njev'],sol.njev)

            # Convergence of training loss
            y_pred = self.model(self.x)
            mse = keras.losses.MeanSquaredError()
            loss = mse(self.y,y_pred).numpy()
            hist['loss'] = np.append(hist['loss'],loss)
            if(val_data):
                # Convergence of val/test loss
                y_val_pred = self.model(val_data[0])
                val_loss = mse(y_val_pred,val_data[1]).numpy()
                print("epoch: ",k+1,"DL: ",tol,"njev: ",sol.njev,
                   "loss = ",loss,"val_loss = ",val_loss)
                hist['val_loss'] = np.append(hist['val_loss'],val_loss)

            if k>1 and ((1+tau)*hist['loss'][-1] > hist['loss'][-2]):
                break
        return hist
```

B.1.1 Numerical verification of Jacobian Matrix

```
# almtest.py
import net
from algopy import UTPM
import tensorflow as tf
import numpy as np
from tensorflow import keras
import alm
from matplotlib import pyplot

I,O,W,D,N, = 2,1,3,2,21
mu = 10

shape = (W,D,I,O)
sigma  = lambda x: np.tanh(x)
sigma_ = lambda x: 2/(np.cosh(2*x)+1)

tau  = lambda x: x
tau_ = lambda x: np.ones(x.shape)

x = np.random.normal(0,1,I*N).reshape((I,N))
y = np.random.normal(0,1,O*N).reshape((O,N))
```

```
u = np.random.random_sample((W*W*(D-1) + D*W + I*W + O*W + O + D*W*N,))
l = np.random.random_sample((D*W*N,))

nn = net.Net(shape,sigma,sigma_,tau,tau_,x,y)

_,z = nn.sim(u)
n_u = u.size
u[n_u-W*D*N:n_u] = z.ravel()

L = nn.eval_L(u,mu,l)
J = nn.eval_J_L(u,mu,l)

u_U = UTPM.init_jacobian(u)
L_U = net.eval_L_U(u_U,mu,l,nn)
J_U = UTPM.extract_jacobian(L_U)
J_U = J_U.reshape(J_U.shape[1:3])


model = keras.Sequential()
model.add(alm.Dense_d(activation_=sigma_,units=W,activation=tf.math.tanh,input_shape=(I,)))
model.add(alm.Dense_d(activation_=sigma_,units=W,activation=tf.math.tanh))
model.add(alm.Dense_d(activation_=tau_,units=O,activation=tau))
n2 = alm.ALMModel(model, x.transpose(), y.transpose())
model.summary()

g = np.arange(W)+1
a = g*(I+1)
b = a[2] + g*(W+1)
c = b[2] + W+1

g = np.r_[a,b,c]-1
h = np.arange(n_u-D*W*N,n_u)

mask = np.ones((n_u,),bool)

mask[g] = 0
mask[h] = 0

k = np.arange(n_u)
t = np.arange(n_u)
t[mask] = k[0:h[0]-D*W-O]
t[g] = k[h[0]-D*W-O:h[0]]
t[h] = k[h]

L2 = n2.L(u[t], mu, l)
J2 = n2.JL(u[t], mu).toarray()

Jnz = J2 != 0

J2 = np.c_[J2[:,mask],J2[:,g],J2[:,h]]

print(np.allclose(L,np.ravel(L_U.data[0,0])))
print(np.allclose(L,L2))
print(np.allclose(J,J_U))
print(np.allclose(J,J2))
```

```
np.set_printoptions(precision=3,linewidth=200)
pyplot.matshow(Jnz,cmap="binary")
pyplot.show()
```

B.2 batch tests

B.2.1 tanh test

```
import net
from algopy import UTPM
import tensorflow as tf
import numpy as np
from tensorflow import keras
import alm
from matplotlib import pyplot
import time

rng = np.random.default_rng()
delta = 0.1
W = 8
K = 20

sigma  = lambda x: tf.math.tanh(x)
sigma_ = lambda x: 2/(np.cosh(2*x)+1)

tau  = lambda x: x
tau_ = lambda x: np.ones(x.shape)


for N in [10,20,40]:
    x = np.linspace(0,np.pi,N).reshape((N,1))
    y = np.sin(x*x)+rng.normal(0,delta,x.shape)
    per = rng.permutation(N)

    te = np.empty((K,4))
    for k in range(K):
        madam = keras.Sequential()
        madam.add(keras.layers.Dense(activation="tanh",units=W,input_shape=(x.shape[1],)))
        madam.add(keras.layers.Dense(activation="tanh",units=W))
        madam.add(keras.layers.Dense(units=y.shape[1]))
        adam = keras.optimizers.Adam()
        es = keras.callbacks.EarlyStopping(monitor='loss',patience=10)
        madam.compile(optimizer=adam, loss='mean_squared_error')

        malm = keras.Sequential()
        malm.add(alm.Dense_d(activation=sigma,activation_=sigma_,units=W,input_shape=(x.shape[1],)))
        malm.add(alm.Dense_d(activation=sigma,activation_=sigma_,units=W))
        malm.add(alm.Dense_d(activation=tau,activation_=tau_,units=y.shape[1]))

        w = madam.get_weights()
        malm.set_weights(w)
```

```
        t0 = time.process_time()
        hist = madam.fit(x[per,:],y[per,:],batch_size=N,epochs=4000,callbacks=[es],verbose=0)
        t1 = time.process_time()-t0
        e1 = len(hist.history['loss'])
        print("t: ",t1,"k: ",e1,'loss: ', hist.history['loss'][-1])

        almnet = alm.ALMModel(malm, x[per,:], y[per,:])

        t0 = time.process_time()
        hist2 = almnet.fit_alm()
        t2 = time.process_time()-t0
        e2 = hist2['loss'].size
        print("t: ",t2,"k: ",e2, 'loss: ', hist2['loss'][-1])

        te[k,:] = [t1,t2,e1,e2]
        with open('testtanhn.npy','ab') as f:
            np.save(f,np.array(hist.history['loss']))
            np.save(f,np.concatenate([np.array(value) for value in
                                      hist2.values()]).reshape((3,-1)).transpose())
    with open('testtanhn.npy','ab') as f:
        np.save(f,te)
```

B.2.2 relu test

```
import net
from algopy import UTPM
import tensorflow as tf
import numpy as np
from tensorflow import keras
import alm
from matplotlib import pyplot
import time

rng = np.random.default_rng()
delta = 0.1
W = 16
K = 20

sigma  = lambda x: tf.nn.relu(x)
sigma_ = lambda x: np.greater(x,0,x)

tau  = lambda x: x
tau_ = lambda x: np.ones(x.shape)


for N in [20,40,80]:
    x = np.linspace(0,np.pi,N).reshape((N,1))
    y = np.sin(x*x)+rng.normal(0,delta,x.shape)
    per = rng.permutation(N)

    te = np.empty((K,4))
    for k in range(K):
        madam = keras.Sequential()
        madam.add(keras.layers.Dense(activation="relu",units=W,input_shape=(x.shape[1],)))
```

```
    madam.add(keras.layers.Dense(activation="relu",units=W))
    madam.add(keras.layers.Dense(units=y.shape[1]))
    adam = keras.optimizers.Adam()
    es = keras.callbacks.EarlyStopping(monitor='loss',patience=10)
    madam.compile(optimizer=adam, loss='mean_squared_error')

    malm = keras.Sequential()
    malm.add(alm.Dense_d(activation=sigma,activation_=sigma_,units=W,input_shape=(x.shape[1],)))
    malm.add(alm.Dense_d(activation=sigma,activation_=sigma_,units=W))
    malm.add(alm.Dense_d(activation=tau,activation_=tau_,units=y.shape[1]))

    w = madam.get_weights()
    malm.set_weights(w)

    t0 = time.process_time()
    hist = madam.fit(x[per,:],y[per,:],batch_size=N,epochs=4000,callbacks=[es],verbose=0)
    t1 = time.process_time()-t0
    e1 = len(hist.history['loss'])
    print("t: ",t1,"k: ",e1,'loss: ', hist.history['loss'][-1])

    almnet = alm.ALMModel(malm, x[per,:], y[per,:])

    t0 = time.process_time()
    hist2 = almnet.fit_alm()
    t2 = time.process_time()-t0
    e2 = hist2['loss'].size
    print("t: ",t2,"k: ",e2, 'loss: ', hist2['loss'][-1])

    te[k,:] = [t1,t2,e1,e2]
    with open('testrelun.npy','ab') as f:
        np.save(f,np.array(hist.history['loss']))
        np.save(f,np.concatenate([np.array(value) for value in
                                  hist2.values()]).reshape((3,-1)).transpose())
with open('testrelun.npy','ab') as f:
    np.save(f,te)
```

Bibliography

[1] scipy.optimize.least_squares. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html, retrieved 2021-05-31.

[2] D. P. Bertsekas. On penalty and multiplier methods for constrained minimization. *SIAM journal on control and optimization*, 14(2):216–235, 1976.

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

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

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

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

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

[8] L. Lu. Dying relu and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28(5):1671–1706, Jun 2020.

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

[10] L. B. Rall. *Automatic differentiation : techniques and applications*. Lecture notes in computer science 120. Springer, Berlin, 1981.

[11] M. F. Sahin, A. eftekhari, A. Alacaoglu, F. Latorre, and V. Cevher. An inexact augmented lagrangian framework for nonconvex optimization with nonlinear constraints. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.