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

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. With enough neurons, ANNs can approximate any function. 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. Nonetheless a number of results show that for wide enough networks, there are few "bad" local minima.

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 tansig 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 [10] 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: $x_i \in \mathbb{R}^{d_x}, y_i \in \mathbb{R}^{d_y}, i = 1, \ldots, n$ 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:

$$f_W(x) = W_L \sigma(W_{L-1}\sigma(...W_1\sigma(W_0 x)...)) \tag{1.4}$$

where L is the number of hidden layers in the network, W_j are matrixes of dimension $d_j \times d_{j-1}, j = 1 \ldots L$ 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:

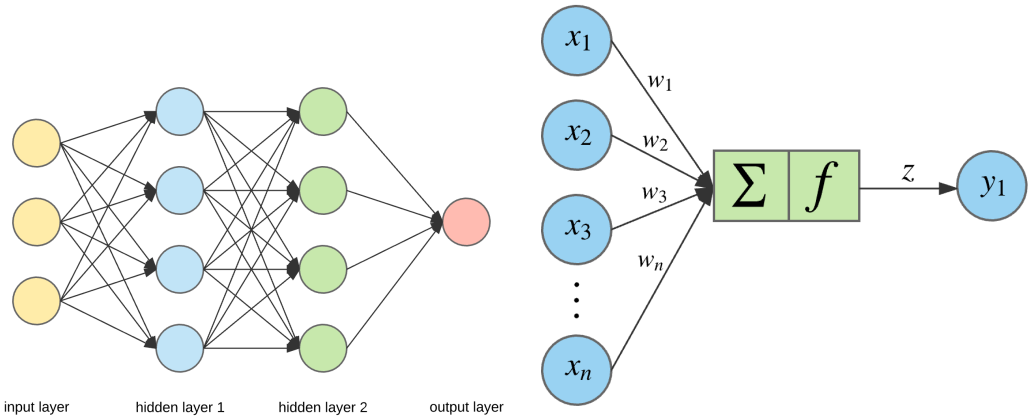


Figure 1.1: Feedforward Deep Neural Network and Single Neuron - McCulloch-Pitts model. (Retrieved from https://towardsdatascience.com)

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

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 **??**. 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	

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) [9]. 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 [3].

This section will explain how backpropagation works, it will largely follow and summarize [8]. 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 [7]. But control theory has many other solution methods for OCPs. One of the more well known one is the direct multiple shooting method (DMS) [2].

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 non-linear program (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 method is that the problem becomes less non-linear. In practice this often makes the NLP easier to solve, and that 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 the algorithm will be implemented in python using an Augmented Langrangian Method.

The code will be compared to common gradient descent and stochastic gradient descent algorithms used in practice such as ADAM [6]. They will be compared in terms of speed, scalability and reliability 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. [4], Sec. 8.2).

Chapter 2

Initial exploration

In this chapter an initial exploration of the problem is done. The backpropagation algorithm is compared relative to the simultaneous approach for a couple small regression problems.

For the backpropagation algorithm the experiments were done using the Neural Network Toolbox in MATLAB version 2018a. The training was done using the `trainlm` procedure which implements a Levenberg-Marquardt backpropagation algorithm.

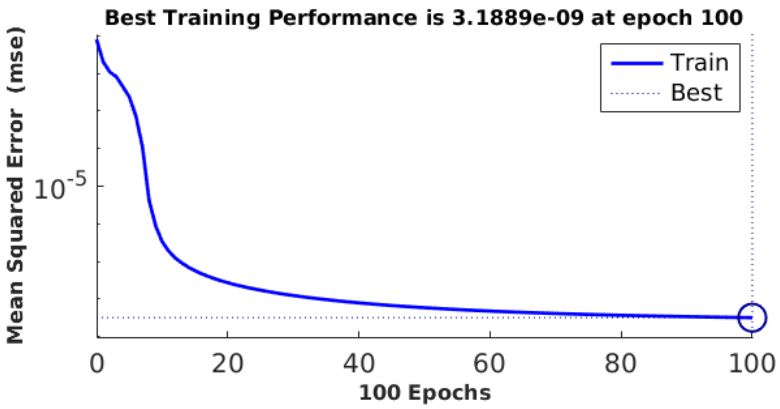
For the simultaneous approach the problems were set up using the YALMIP version R20190425 optimization library, implementing the OCP detailed in equation 1.10. This problem was then solved using MATLAB's `fmincon` procedure.

2.1 First experiment

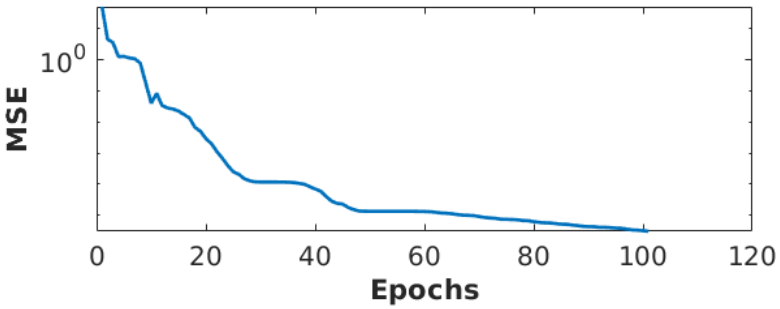
The first experiment conducted was to apply both algorithms to a test problem. A small neural network is constructed with 2 hidden layers with each layer containing 3 nodes with tansig activation function. The activation function was chosen because it works well for a small network. This network is then trained to approximate a piece of a sine function. Both algorithms are trained for 100 epochs. Figure **??** shows the training performance for a training run of each algorithm. Both perform well, but the simultaneous approach is much slower. This can be due to the fact that the new algorithm is not well optimized.

The backpropagation algorithm converges every time for this problem, taking anywhere between 10 and 1000 iterations. The stopping criteria is that the validation error increases for 6 consecutive iterations.

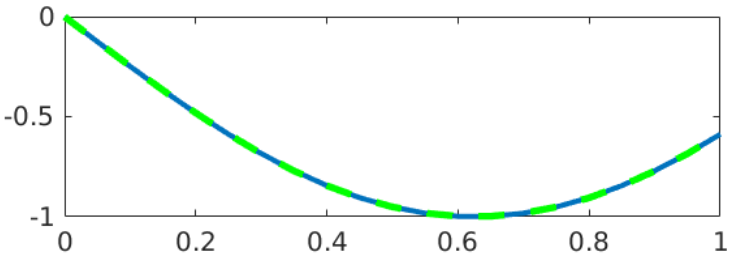
The simultaneous approach also converges if the intial conditions are set correctly. The weight variables are randomly initialized and the state variables are initialized by simulating the network once using the input vector, ensuring the initial point is feasible. The method converges every time, but it runs much slower. Its not known if this is due to the difference in optimization of the algorithms and the fact that the backpropagation algorithm is running on my GPU instead of the CPU.



(A) Training performance of backpropagation



(B) Training performance of simultaneous approach



(C) Sine function

Figure 2.1: Performance of algorithms for simple regression problem

2.2 Second experiment

For this experiment again the same regression problem of the previous section is considered but with a different network. Here a network of two layers is used, with 8 neurons in each layer, but using a ReLU activation function.

The backpropagation function will almost always converge for this problem.

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

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

But even using these smooth constraints, the algorithm will usually not converge. It might be that this is due to the use of the fmincon optimization method, which is very general. In the next chapter the simultaneous approach will be rewritten in python using a more specific algorithm.

2.3 Further exploration

There are many options that can be explored to further compare these algorithms.

- Activation function: ReLU is the most popular activation function in the field. Others are tansig, sigmoid, SoftPlus, leaky ReLU, etc.
- Network size: Networks can be up to thousands of neurons wide and hundreds of layers deep.
- Network architecture: There is a huge variety of possible network architectures. Convolutional neural networks for example are very popular for image recognition.
- Test problems: Neural networks have many applications. Some applications could benefit more from this training method than others.
- Optimization algorithm: fmincon is quite a general method, a more specific method might perform better

- Stopping criteria and initial conditions

  These will be explored in the next chapters

The option that will be explored in the next chapter, will be changing the optimization algorithm. The `fmincon` method will be replaced by a specifically written algorithm, exploiting the structure of the problem.

Chapter 3

Augmented Lagrangian Method

In this chapter the simultaneous approach is examined more closely and implemented into python. Instead of using `fmincon` which is a very general black box method, a more specific algorithm should both run faster and converge more often. For this problem the Augmented Lagrangian Method has been chosen.

3.1 Augmented Lagrangian Method

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:

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

This 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 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:

$$\mathcal{L}_c(x, \lambda) = \frac{1}{2}||F(x)||_2^2 + < \lambda, h(x) > + \frac{c}{2}||h(x)||_2^2 \tag{3.1}$$

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.

$$\begin{aligned}\min_x \quad \mathcal{L}_c(x,\lambda) &= \frac{1}{2}||F(x)||_2^2 + <\lambda, h(x)> + \frac{c}{2}||h(x)||_2^2 \\ &= \frac{1}{2}||F(x)||_2^2 + \frac{c}{2}||h(x)+\lambda/c||_2^2 - \frac{1}{2c}||\lambda||_2^2 \\ &= \frac{c}{2}\left|\left|\begin{bmatrix} F(x)/\sqrt{c} \\ h(x)+\lambda/c \end{bmatrix}\right|\right|_2^2 \\ &= \frac{c}{2}||G_c(x,\lambda)||_2^2\end{aligned} \tag{3.2}$$

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:

$$||\nabla_k G_{c_k}(x^k, \lambda^k)||_2 \leq \epsilon \tag{3.3}$$

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:

$$\lambda^{k+1} = \lambda^k + c_k h(x^k) \tag{3.4}$$

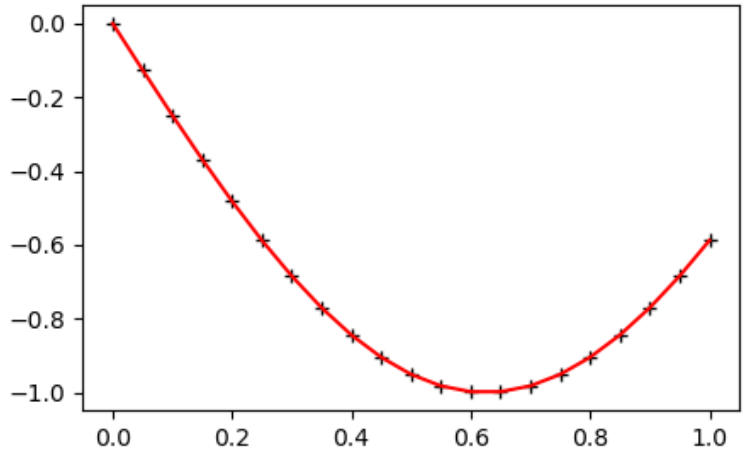
A new penalty parameter c^{k+1} are updated using the rule described in [], which depends on the norm of the gradient. If the constraints are sufficiently respected, the penalty parameter is not updated, otherwise the penalty parameter is increased by a factor 100. The algorithm continues in this manner until the norm of the Jacobian matrix reaches a tolerance ζ.

$$\begin{aligned}\mathcal{L}_c(x,\lambda) &= \frac{1}{2}||F(x)||_2^2 + <\lambda, h(x)> + \frac{c}{2}||h(x)||_2^2 \\ &= \frac{1}{2}||F(x)||_2^2 + \frac{c}{2}||h(x)+\lambda/c||_2^2 - \frac{1}{2c}||\lambda||_2^2 \\ &= \frac{c}{2}\left|\left|\begin{bmatrix} F(x)/\sqrt{c} \\ h(x)+\lambda/c \end{bmatrix}\right|\right|^2\end{aligned} \tag{3.5}$$

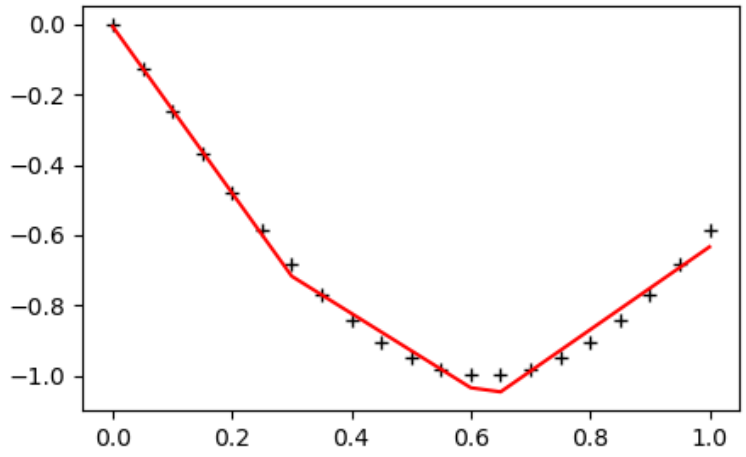
3.2 Jacobian

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.

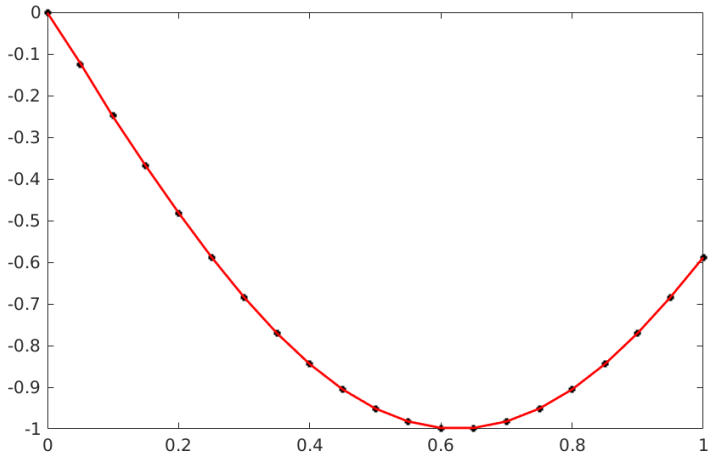
Neural networks obviously have many possible architectures, so a simple fully connected rectangular feedforward network is considered. It has an input dimension I, an output dimension O, and it has D hidden layers of width W. The weight matrixes have $I \times W + O \times W + (D-1) \times W \times W$ parameters, the bias vectors have $D \times W + O$



(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

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 3.1

3.3 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

3.4 Alternative Representation

In this section we explain an alternative representation which is mathematically the same. Figure 3.3 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.

3.5 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 3.5 is less than a tolerance of $1e^{-6}$. Figure 3.1a and Figure 3.1b 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

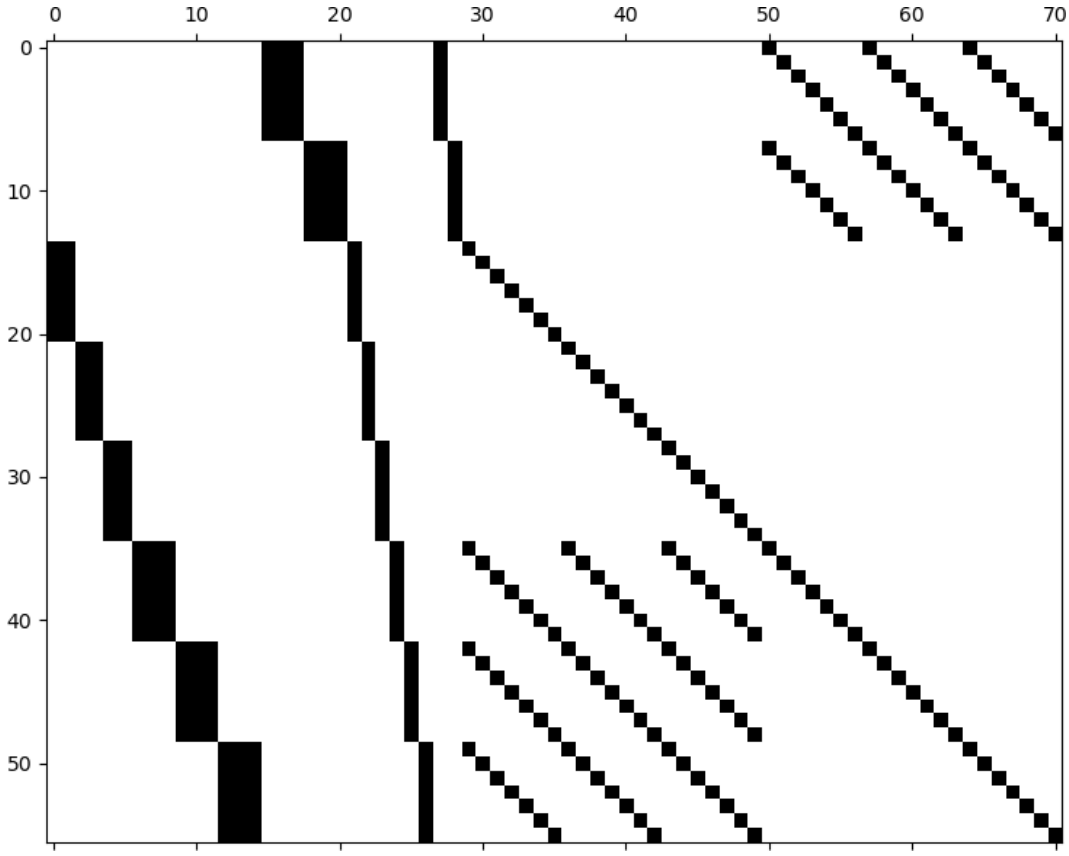


FIGURE 3.2: Nonzero elements of Jacobian, for network with I=2,O=2,W=3,D=2,N=7

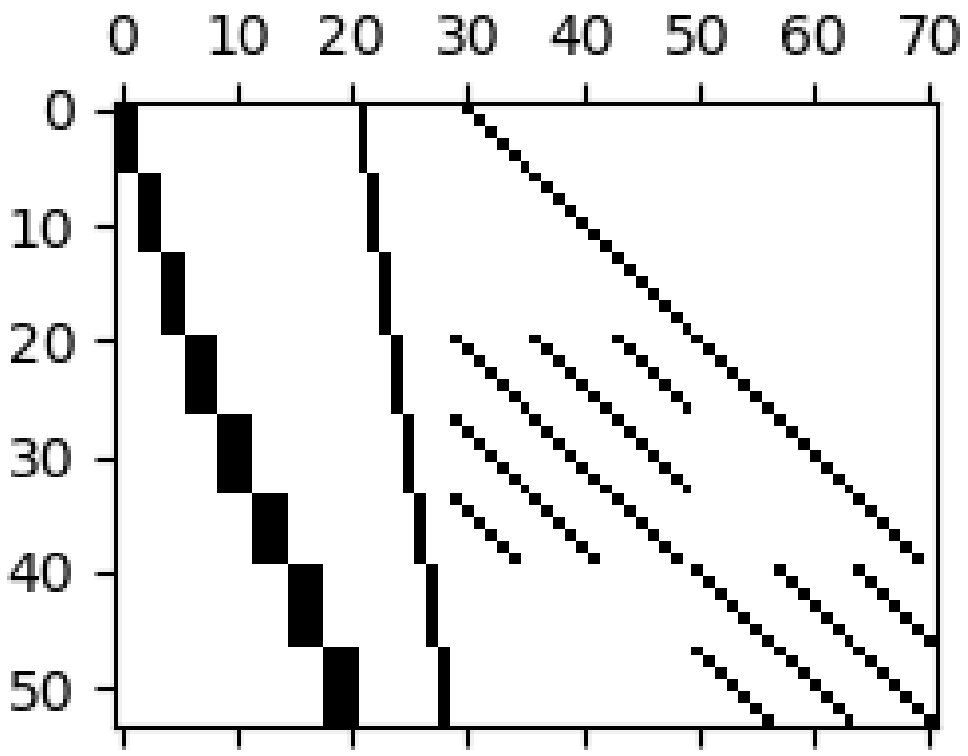


Figure 3.3: Nonzero elements of Jacobian, for network with I=2,O=2,W=3,D=2,N=7, rearranged

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

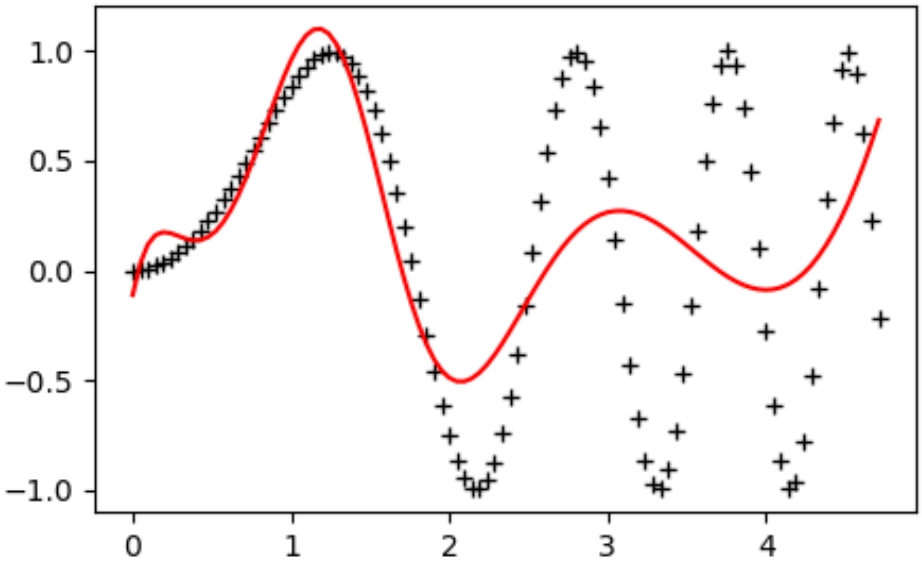


FIGURE 4.1: Feedforward network with 2 hidden layers of 20 relu units, trained on test function using ALM, using 80 training pairs

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

- [10]
  - Optimization for deep learning: theory and algorithms
  - Goals of optimization
  - Problem formulation
  - Gradient descent:
  - Tips tricks
  - State of the art algorithms

- [3]
  "The Kelley-Bryson gradient formulas for such problems have been rediscovered by neural- network researchers and termed back propagation"

- [1]
  Practical Augmented lagrangian method

- [5] "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"

- [9] Original Backpropagation paper

4.4 Needed

- Basic Neural network intro
- Optimal control
- Simultaneous approach

Chapter 5

Conclusion

The final chapter contains the overall conclusion. It also contains suggestions for future work and industrial applications.

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

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

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

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