Universität Münster
Fachbereich Mathematik und Informatik

Masterarbeit Mathematik

Machine learning based surrogate modeling to accelerate parabolic PDE constrained optimization

vorgelegt am: 9. Juli 2024
von: Andy Kevin Wert
Matrikelnummer: 461478
Erstgutachter: Prof. Dr. Mario Ohlberger
Zweitgutachter: Dr. Stephan Rave

Contents

1 Introduction

When extracting oil from an oil reservoir, there are many parameters that influence the cost and the resulting output. To achieve a greater profit, one wants to find parameters that optimize a net present value (NPV) function which evaluates the economic value of the chosen oil recovery strategy. A difficulty here is that the computation of this NPV function is usually expensive. Also, gradients are often not available. Therefore, one resorts to gradient free optimization methods. A popular example is the ensemble-based (EnOpt) optimization method which uses samples around each iterate to compute an approximation of a preconditioned gradient at that iterate. With this gradient, the next iterate is computed by a gradient ascent method. An extension of the EnOpt method is the so-called adaptive EnOpt algorithm. Here, the covariance matrix, that is used for the sampling around the iterates, is adjusted with respect to the result from the previous iteration.

In [1], a modification of the adaptive EnOpt method is proposed which is called the adaptive machine learning EnOpt (Adaptive-ML-EnOpt) algorithm. In this procedure, a neural network-based surrogate is trained in each iteration to reduce costly calls of the full order model (FOM) NPV function and thus speed up the computation. This method is applied to an enhanced oil recovery strategy where a polymer-water mixture is injected into the reservoir to bring the oil to the surface.

In this thesis, we apply the adaptive EnOpt and the Adaptive-ML-EnOpt algorithms to an optimal control problem that is constrained by parabolic equations. The source code can be found at [github link]. In order to run this code, the installation of the python packages ...[packages with links] is necessary. The installation of CUDA[link] is optional but recommended if available.

The constrained control problem, along with the solution strategy of the corresponding discretized parabolic equations, is presented in chapter 2. After that, chapter 3 presents the adaptive EnOpt algorithm. In chapter 4 the Adaptive-ML-EnOpt procedure from [1] is described. Finally, we test and compare these two algorithms in chapter 5.

2 Parabolic optimal control problems

2.1 Introduction to the problem

Our optimization problem is based on the problem that is presented in [2]. We consider a state variable u and a control variable q, defined on $(0,T)\times\Omega$ with $T\in\mathbb{R}$ and $\Omega\subset\mathbb{R}^n$.

The goal of this thesis is to minimize the function

$$J(q,u)=\frac{1}{2}\int_0^T\int_\Omega (u(t,x)-\hat{u}(t,x))^2\,\mathrm{d}x\,\mathrm{d}t+\frac{\alpha}{2}\int_0^T\int_\Omega q(t,x)^2\,\mathrm{d}x\,\mathrm{d}t, \tag{2.1a}$$

subject to the constraints

$$\begin{aligned}\partial_t u-\Delta u&=f+q \quad \text{in } (0,T)\times\Omega,\\ u(0)&=u_0 \qquad\qquad \text{in } \Omega,\end{aligned} \tag{2.1b}$$

with homogeneous Dirichlet boundary conditions on $(0,T)\times\partial\Omega$.

Let $V=H_0^1(\Omega)$, $H=L^2(\Omega)$ and $I=(0,T)$. We define our state space as

$$X:=\{v\mid v\in L^2(I,V) \text{ and } \partial_t v\in L^2(I,V^*)\}$$

and the control space as

$$Q:=L^2(I,L^2(\Omega)).$$

The notion of the inner products and norms on $L^2(\Omega)$ and $L^2(I,L^2(\Omega))$ is introduced as

$$\begin{aligned}(v,w)&:=(v,w)_{L^2(\Omega)}, & (v,w)_I&:=(v,w)_{L^2(I,L^2(\Omega))},\\ \|v\|&:=\|v\|_{L^2(\Omega)}, & \|v\|_I&:=\|v\|_{L^2(I,L^2(\Omega))}.\end{aligned}$$

By using the inner product, the weak form of the state equations (2.1b) for $q,f\in Q$ and $u_0\in V$ is given as

$$\begin{aligned}(\partial_t u,\phi)_I+(\nabla u,\nabla\phi)_I&=(f+q,\phi)_I \quad \forall\phi\in X,\\ u(0)&=u_0 \qquad\qquad \text{in } \Omega.\end{aligned} \tag{2.2}$$

With the weak state equations (2.2), we define the weak formulation of the optimal control problem (2.1) as

$$\text{Minimize } J(q,u):=\frac{1}{2}\|u-\hat{u}\|_I^2+\frac{\alpha}{2}\|q\|_I^2 \text{ subject to (2.2) and } (q,u)\in Q\times X. \tag{2.3}$$

Now we cite two results of the problems (2.2) and (2.3).

Proposition 2.1 ([2])**.** *For fixed $q,f\in Q$, and $u_0\in V$ there exists a unique solution $u\in X$ of problem* (2.2)*. Moreover, the solution exhibits the improved regularity*

$$u\in L^2(I,H^2(\Omega)\cap V)\cap H^1(I,L^2(\Omega))\hookrightarrow C(\bar{I},V).$$

It holds the stability estimate

$$\|\partial_t u\|_I+\|\nabla^2 u\|_I\le C\{\|f+q\|_I+\|\nabla u_0\|\}.$$

Proposition 2.2 ([2]). *For given $f, \hat{u} \in L^2(I,H)$, $u_0 \in V$, and $\alpha > 0$, the optimal control problem (2.3) admits a unique solution $(\bar{q}, \bar{u}) \in Q \times X$. The optimal control $\bar{q}$ posesses the regularity*

$$\bar{q} \in L^2(I, H^2(\Omega)) \cap H^1(I, L^2(\Omega)).$$

Due to the existence and uniqueness results from Proposition 2.1, we define $u(q)$ as the unique solution of (2.2) with respect to some $q \in Q$. This enables us to define a reduced cost functional $j : Q \to \mathbb{R}$ that is only dependent on the control q as

$$j(q) := J(q, u(q)).$$

From now on, the optimal control problem that we examine is:

$$\text{minimize } j(q) \text{ subject to } q \in Q. \tag{2.4}$$

2.2 Finite element discretization

In order to solve the optimization problem (2.4) numerically, the discretization of our model is now discussed. We begin with the presentation of the discretization in space with a n-D continuous Galerkin method. Then, we look at the discretization in time, which is done with a 1D continuous Galerkin method. From now on, we will also discuss some implementation details, so, in this chapter, how we handle the calculation of the objective function j. To solve the partial equations of (2.2), we use the Python package pyMOR.

2.2.1 Discretization in space

The discretization in space is shown on a 2-dimensional rectangular space $\Omega \subset \mathbb{R}^2$ with linear finite elements. We assume to have a vertex set $\mathcal{V} = (x_1, \ldots, x_{N_\mathcal{V}}) \in (\mathbb{R}^2)^{N_\mathcal{V}}$ with a convex hull that is equal to $\bar{\Omega}$ and $x_i \neq x_j$ for all $i \neq j$ in $\{1, \ldots, N_\mathcal{V}\}$. Let $\hat{T} = \{(x,y) \in [0,1]^2 \mid y \leq 1 - x\}$ be the reference triangle. Then,

$$\theta_l(\xi) = x_{l_1} + D\theta_l \begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix} \text{ with } D\theta_l = \begin{pmatrix} x_{l_2} - x_{l_1} & x_{l_3} - x_{l_1} \end{pmatrix}$$

is a transformation from the reference triangle $\hat{T}$ to some other triangle T_l with the corners $x_{l_1}, x_{l_2}, x_{l_3} \in \mathcal{V}$.

We define now a mesh $\mathcal{T} = \{T_l\}$ which consists of triangles $T_l = \theta_l(\hat{T})$, where $T_l \cap T_m$ for $T_l, T_m \in \mathcal{T}$ is either a common side, a common corner, or empty, and where $\bar{\Omega} = \cup_{T_l \in \mathcal{T}} T_l$. We also assume that every vertex in $\mathcal{V}$ is a corner of at least one triangle of $\mathcal{T}$.

In the pyMOR implementation, we discretize a rectangular domain by specifying the number of grid intervals first. Then, the domain is subdivided into smaller rectangles of the same size, so that the number of rectangles along the x- and the y-axis is equal to the predefined number of grid intervals. Each smaller rectangular unit is then partitioned into four equally sized triangles by adding a vertex into the center of the rectangle which is connected with the corners of the unit. The vertex set of the whole domain is now given by the union of the corners of all triangles. As an example, if a domain $\Omega = [a, a]$ with $a > 0$ is given and we set the number of grid intervals to two, then our mesh would look like that:

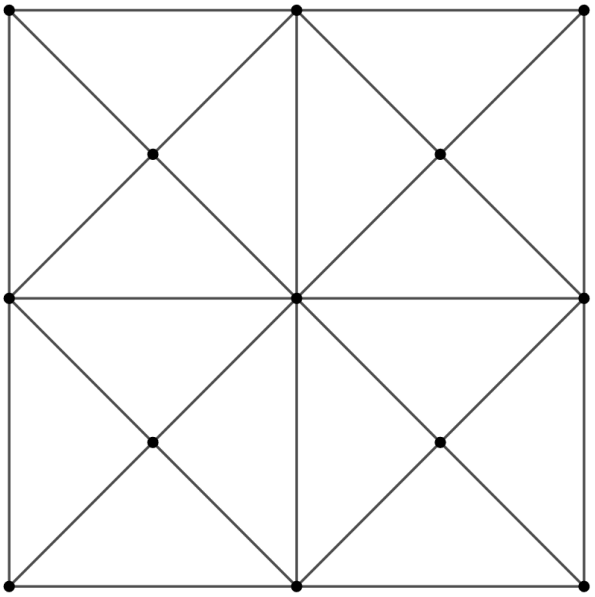

Figure 2.1: Example of a mesh with two grid intervals in a square-shaped domain. The dots denote elements of the vertex set $\mathcal{V}$.

Now, let $\mathcal{P}_1(\hat{T}, \mathbb{R})$ be the space of polynomials up to order one in $\hat{T}$. Then, $\{\psi_1, \psi_2, \psi_3\}$ with $\psi_1(\xi) = 1 - \xi_1 - \xi_2$, $\psi_2(\xi) = \xi_1$, $\psi_3(\xi) = \xi_2$ defines a basis of $\mathcal{P}_1(\hat{T}, \mathbb{R})$. Using this basis, we set

$$V_h = \operatorname{span}\{\phi_i, i = 0, \dots, N_\mathcal{V}\} \cap V$$

as the finite element space of our state variables with

$$\phi_i|_{T_l} = \begin{cases} 0 & \text{if } x_i \notin T_l \\ \psi_1 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}0\\0\end{pmatrix}\right) = x_i \\ \psi_2 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}1\\0\end{pmatrix}\right) = x_i \\ \psi_3 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}0\\1\end{pmatrix}\right) = x_i \end{cases}$$

for all $T_l \in \mathcal{T}$ and $i = 1, \dots, N_\mathcal{V}$.

By construction, every $u \in V_h$ is uniquely defined by

$$u = \sum_{i=1}^{N_\mathcal{V}} U_i \phi_i \tag{2.5}$$

with $U_i = u(x_i)$.

Now we want to calculate $\int_\Omega u \cdot v \,\mathrm{d}x$ and $\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x$ for all $u, v \in V_h$. In order to do that, we set the mass matrix $\mathbf{M}_n = \left(\int_\Omega \phi_i \cdot \phi_j \,\mathrm{d}x\right)_{i,j=1,\dots,N_\mathcal{V}}$ and the stiffness matrix

$\mathbf{L}_n = \left(\int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}x \right)_{i,j=1,\dots,N_\mathcal{V}}$. Let

$$\mathbf{U} = \begin{pmatrix} U_1 \\ \vdots \\ U_{N_\mathcal{V}} \end{pmatrix} \text{ and } \mathbf{V} = \begin{pmatrix} V_1 \\ \vdots \\ V_{N_\mathcal{V}} \end{pmatrix},$$

where $U_i = u(x_i)$, $V_i = v(x_i)$ for $i = 1, \dots, N_\mathcal{V}$ as in (2.5). Then we have, based on the representation of elements in V_h from equation (2.5), that

$$\int_\Omega u \cdot v \, \mathrm{d}x = \mathbf{U}^T \mathbf{M}_n \mathbf{V} \text{ and } \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x = \mathbf{U}^T \mathbf{L}_n \mathbf{V}.$$

2.2.2 Discretization in time

For the discretization in time, we first partition the time interval $\bar{I} = [0, T]$ as

$$\bar{I} = \{0\} \cup I_1 \cup I_2 \cup \cdots \cup I_{N_t}$$

with subintervals $I_m = (t_{m-1}, t_m]$, where $t_m = m\dfrac{T}{N_t}$ for $m = 0, \dots, N_t$ and $N_t \in \mathbb{N}$. We want that the discretizations of our functions are continuous in $\bar{I}$ and piecewise polynomial of order one in all subintervals I_m, so the discretization space of our state variables is

$$X_{k,h} := \{v \in C(\bar{I}, V_h) \mid v|_{I_m} \in \mathcal{P}_1(I_m, V_h), m = 1, 2, \dots, N_t\},$$

where $\mathcal{P}_1(I_m, V_h)$ denotes the space of polynomials up to order one, defined on I_m with values in V_h. Similarly, we define the time-discretized space of our control variables as

$$Q_d := \{v \in C(\bar{I}, H) \mid v|_{I_m} \in \mathcal{P}_1(I_m, H), m = 1, 2, \dots, N_t\} \supset X_{k,h}.$$

By using the Lagrange basis of $\mathcal{P}_1(I_m, \mathbb{R})$, we can write every function $v \in Q_d$ as

$$v(t, x) = \left(m - t\frac{N_t}{T} \right) v_{m-1}(x) + \left(t\frac{N_t}{T} - m + 1 \right) v_m(x) \text{ for } t \in I_m,$$

where $v_m(x) = v(t_m, x)$.

2.2.3 Crank-Nicolson scheme

Now, we solve the weak state equations (2.2) for the state $u \in X_{k,h}$, the control $q \in Q_d$, and $f \in Q$ numerically. Let $U_{m,i} := u_m(x_i)$ for $m = 0, \dots, N_t$ and $i = 1, \dots, N_\mathcal{V}$. We set

$$\mathbf{U}_m = \begin{pmatrix} U_{m,1} \\ \vdots \\ U_{m,N_\mathcal{V}} \end{pmatrix}$$

for $m = 0, \dots, N_t$. The initial discretized state vector $\mathbf{U}_0$ is already given by the values of u_0. Next, we want to solve the weak state equations for $\mathbf{U}_m$ at the other time steps $m = 1, \dots, N_t$, so that we can set $u_m = \sum_{i=1}^{N_\mathcal{V}} U_{m,i} \phi_i$ according to equation (2.5).

For $m = 1, \ldots, N_t$, we get with the Crank-Nicolson scheme that for all $v \in V_h$:

$$\begin{aligned}(u_m, v) + \frac{T}{2N_t}(\nabla u_m, \nabla v) &= (u_{m-1}, v) - \frac{T}{2N_t}(\nabla u_{m-1}, \nabla v) \\ &\quad + \frac{T}{2N_t}(f_{m-1} + q_{m-1}, v) + \frac{T}{2N_t}(f_m + q_m, v),\end{aligned}$$

where $u_m = u(t_m, \cdot)$, $f_m = f(t_m, \cdot)$ and $q_m = q(t_m, \cdot)$. To solve the above equation, while considering the homogeneous Dirichlet boundary conditions, we define the matrix $\tilde{\mathbf{M}}_n \in \mathbb{R}^{N_\mathcal{V} \times N_\mathcal{V}}$ as

$$\left(\tilde{\mathbf{M}}_n\right)_{i,j} = \begin{cases} 0 & \text{if } x_i \text{ or } x_j \text{ in } \partial\Omega \text{ and } i \neq j \\ 1 & \text{if } x_i \text{ or } x_j \text{ in } \partial\Omega \text{ and } i = j \\ (\mathbf{M}_n)_{i,j} & \text{else} \end{cases}$$

and the matrix $\tilde{\mathbf{L}}_n \in \mathbb{R}^{N_\mathcal{V} \times N_\mathcal{V}}$ as

$$\left(\tilde{\mathbf{L}}_n\right)_{i,j} = \begin{cases} 0 & \text{if } x_j \text{ in } \partial\Omega \\ (\mathbf{L}_n)_{i,j} & \text{else,} \end{cases}$$

so that $(u_m, v) = \mathbf{U}_m^T \tilde{\mathbf{M}}_n \mathbf{V}$ and $(\nabla u_m, \nabla v) = \mathbf{U}_m^T \tilde{\mathbf{L}}_n \mathbf{V}$ for all $m = 0, \ldots, N_t$, which is giving us

$$\begin{aligned}\mathbf{V}^T \tilde{\mathbf{M}}_n^T \mathbf{U}_m + \frac{T}{2N_t} \mathbf{V}^T \tilde{\mathbf{L}}_n^T \mathbf{U}_m &= \mathbf{V}^T \tilde{\mathbf{M}}_n^T \mathbf{U}_{m-1} - \frac{T}{2N_t} \mathbf{V}^T \tilde{\mathbf{L}}_n^T \mathbf{U}_{m-1} \\ &\quad + \frac{T}{2N_t}(f_{m-1} + q_{m-1}, v) + \frac{T}{2N_t}(f_m + q_m, v).\end{aligned}$$

In the pyMOR implementation, vectors $\mathbf{F}_m$ for $m = 0, \ldots, N_t$ are defined such that $\mathbf{V}^T \mathbf{F}_m \approx (f_m + q_m, v)$ for all $v \in \mathbf{V}_h$ and $(\mathbf{F}_m)_i = 0$ if the i-th entry in the vertex set $\mathcal{V}$ lies on the boundary of Ω. By using these vectors, we get the equation

$$\left(\tilde{\mathbf{M}}_n^T + \frac{T}{2N_t} \tilde{\mathbf{L}}_n^T\right) \mathbf{U}_m = \tilde{\mathbf{M}}_n^T \mathbf{U}_{m-1} - \frac{T}{2N_t} \tilde{\mathbf{L}}_n^T \mathbf{U}_{m-1} + \frac{T}{2N_t} \mathbf{F}_{m-1} + \frac{T}{2N_t} \mathbf{F}_m, \tag{2.6}$$

which is solved after $\mathbf{U}_m$ with algorithms from the Python package SciPy.

2.2.4 Calculation of the objective function value

For fixed $\hat{u}, f \in Q$, we define $u = u(q)$ for all $q \in Q_d$, so that it satisfies (2.6). We calculate $j(q)$ now in the following way:

$$\begin{aligned}j(q) \approx \quad & \frac{1}{2} \sum_{m=1}^{N_t} \int_{t_{m-1}}^{t_m} \quad \left(\left(m - t\frac{N_t}{T}\right)(u_{m-1} - \hat{u}_{m-1}) + \left(t\frac{N_t}{T} - m + 1\right)(u_m - \hat{u}_m), \right. \\ & \qquad \left. \left(m - t\frac{N_t}{T}\right)(u_{m-1} - \hat{u}_{m-1}) + \left(t\frac{N_t}{T} - m + 1\right)(u_m - \hat{u}_m) \right) \mathrm{d}t \\ & + \frac{\alpha}{2} \sum_{m=1}^{N_t} \int_{t_{m-1}}^{t_m} \quad \left(\left(m - t\frac{N_t}{T}\right) q_{m-1} + \left(t\frac{N_t}{T} - m + 1\right) q_m, \right. \\ & \qquad \left. \left(m - t\frac{N_t}{T}\right) q_{m-1} + \left(t\frac{N_t}{T} - m + 1\right) q_m \right) \mathrm{d}t.\end{aligned}$$

Integration by substitution yields

$$\begin{aligned} j(q) \approx \quad & \frac{T}{6N_t} \sum_{m=1}^{N_t} \quad (u_{m-1} - \hat{u}_{m-1}, u_{m-1} - \hat{u}_{m-1}) + (u_{m-1} - \hat{u}_{m-1}, u_m - \hat{u}_m) \\ & \qquad + (u_m - \hat{u}_m, u_m - \hat{u}_m) \\ & + \frac{\alpha T}{6N_t} \sum_{m=1}^{N_t} \quad (q_{m-1}, q_{m-1}) + (q_{m-1}, q_m) + (q_m, q_m) \\ \approx \quad & \frac{T}{6N_t} \sum_{m=1}^{N_t} \quad \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \\ & \qquad + \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ & \qquad + \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ & + \frac{\alpha T}{6N_t} \sum_{m=1}^{N_t} \quad \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_{m-1} + \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_m + \mathbf{Q}_m\mathbf{M}_n\mathbf{Q}_m, \end{aligned}$$

where $\hat{\mathbf{U}}_m = (\hat{u}(t_m, x_i))_{i=1,\dots,N_\mathcal{V}}$ and $\mathbf{Q}_m = (q(t_m, x_i))_{i=1,\dots,N_\mathcal{V}}$ for $m = 0, \dots, N_t$.

2.3 Optimization of the control variable

To optimize the control variable, we write every $q \in Q_d$, by using a fixed set of shape functions

$$\Phi = \{\phi_1, \dots, \phi_{N_b}\} \tag{2.7}$$

with $\phi_1, \dots, \phi_{N_b} \in H$ and scalars $q_1^0, q_1^1, \dots, q_1^{N_t}, \dots, q_{N_b}^0, q_{N_b}^1 \dots, q_{N_b}^{N_t} \in \mathbb{R}$, as

$$q(t, x) = \sum_{i=1}^{N_b} \alpha_i(t)\phi_i(x) \tag{2.8}$$

with

$$\alpha_i(t) = \begin{cases} q_i^{m-1}\left(m - t\frac{N_t}{T}\right) + q_i^m\left(t\frac{N_t}{T} - m + 1\right) & \text{if } t \in I_m \text{ with } m = 1, \dots, N_t \\ q_i^0 & \text{if } t = 0. \end{cases}$$

Each control variable that is written in this form can be represented as a vector

$$\mathbf{q} = \left[q_1^0, q_1^1, \dots, q_1^{N_t}, \dots, q_{N_b}^0, q_{N_b}^1 \dots, q_{N_b}^{N_t}\right]^T \in \mathcal{D} := \mathbb{R}^{N_\mathbf{q}},$$

with $N_\mathbf{q} = (N_t + 1) \cdot N_b$. Therefore, we write

$$j(\mathbf{q}) := j(q)$$

for each q that is defined like in (2.8).

In the next chapters, we present algorithms that minimize $j(\mathbf{q})$ with respect to its control vector $\mathbf{q}$.

3 Ensemble-based optimization algorithm

The adaptive ensemble-based optimization (EnOpt) algorithm is usually used to maximize the net present value of oil recovery methods with respect to a control vector. Examples are presented in [1], [3], [4]. In this chapter, we want to utilize the EnOpt algorithm to optimize the objective function j. Our implementation is similar to that in [1].

We begin by describing this algorithm for a general function $F : \mathbb{R}^{N_\mathbf{q}} \to \mathbb{R}$ to iteratively solve the optimization problem

$$\underset{\mathbf{q}\in\mathcal{D}}{\text{maximize}}\, F(\mathbf{q}).$$

We start at an initialization $\mathbf{q}_0$, which is updated iteratively with a preconditioned gradient ascent method that is given by

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \beta_k \mathbf{d}_k,$$

$$\mathbf{d}_k \approx \frac{\mathbf{C}^k_{\mathbf{q}_k}\mathbf{G}_k}{\|\mathbf{C}^k_{\mathbf{q}_k}\mathbf{G}_k\|_\infty},$$

where $k = 0, 1, 2, \ldots$ denotes the optimization iteration. β_k with $\beta_k > 0$ is computed by using a line search. Furthermore, $\mathbf{C}^k_{\mathbf{q}_k}$ denotes the user-defined covariance matrix of the control variables at the k-th iteration and $\mathbf{G}_k$ is the approximate gradient of F with respect to the control variables.

We define the initial covariance matrix $\mathbf{C}^0_{\mathbf{q}_0}$ so that the covariance between controls of different basis functions ϕ_i, ϕ_j is zero and

$$\mathrm{Cov}(q^i_j, q^{i+h}_j) = \sigma^2_j \rho^h \left(\frac{1}{1-\rho^2}\right), \text{ for all } h \in \{0, \ldots, M-i\}, \tag{3.1}$$

where $\sigma^2_j > 0$ is the variance for the basis function ϕ_j and $\rho \in (-1, 1)$ the correlation coefficient.

That means that for $\mathbf{C}_j := \left(\mathrm{Cov}(q^i_j, q^k_j)\right)_{i,k}$ with $j = 1, \ldots, N_b$, we set

$$\mathbf{C}^0_{\mathbf{q}_0} = \begin{pmatrix} \mathbf{C}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{C}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{C}_{N_b} \end{pmatrix}. \tag{3.2}$$

To compute the step direction $\mathbf{d}_k$ at iteration step k, we sample $\mathbf{q}_{k,m} \in \mathcal{D}$ for $m = 1, \ldots, N$, with $N \in \mathbb{N}$, from a multivariate Gaussian distribution with the mean $\mathbf{q}_k$ and the covariance $\mathbf{C}^k_{\mathbf{q}_k}$. Then we define

$$\mathbf{C}^k_{\mathbf{q}_k,F} := \frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m} - \mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)). \tag{3.3}$$

Now we set $\mathbf{d}_k = \dfrac{\mathbf{C}^k_{\mathbf{q}_k,F}}{\|\mathbf{C}^k_{\mathbf{q}_k,F}\|_\infty}$. This is valid since $\mathbf{C}^k_{\mathbf{q}_k,F}$ is an estimation of $\mathbf{C}^k_{\mathbf{q}_k}\mathbf{G}_k$, which can by shown like in [3]. Here, we begin with the Taylor expansion around $\mathbf{q}_k$ and get

$$\begin{aligned} & F(\mathbf{q}) = F(\mathbf{q}_k) + (\mathbf{q}-\mathbf{q}_k)^T\nabla F(\mathbf{q}_k) + O(\|\mathbf{q}-\mathbf{q}_k\|^2) \\ \Longrightarrow \quad & F(\mathbf{q}) - F(\mathbf{q}_k) = (\mathbf{q}-\mathbf{q}_k)^T\mathbf{G}_k + O(\|\mathbf{q}-\mathbf{q}_k\|^2). \end{aligned}$$

Multiplying both sides by $(\mathbf{q}-\mathbf{q}_k)$ and setting $\mathbf{q} = \mathbf{q}_{k,m}$ yields

$$\begin{aligned} & (\mathbf{q}_{k,m}-\mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)) \\ = \quad & (\mathbf{q}_{k,m}-\mathbf{q}_k)(\mathbf{q}_{k,m}-\mathbf{q}_k)^T\mathbf{G}_k + O(\|\mathbf{q}_{k,m}-\mathbf{q}_k\|^3), \end{aligned}$$

where $O(\|\mathbf{q}_{k,m}-\mathbf{q}_k\|^3)$ are the remaining terms containing order ≥ 3 of $(\mathbf{q}_{k,m}-\mathbf{q}_k)$. Neglecting $O(\|\mathbf{q}_{k,m}-\mathbf{q}_k\|^3)$ gives by summation over all samples and multiplication of both sides with $\dfrac{1}{N-1}$:

$$\begin{aligned} & \frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)) \\ \approx \quad & \left(\frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(\mathbf{q}_{k,m}-\mathbf{q}_k)^T\right)\mathbf{G}_k \\ \Longrightarrow \quad & \mathbf{C}^k_{\mathbf{q}_k,F} \approx \mathbf{C}^k_{\mathbf{q}_k}\mathbf{G}_k, \end{aligned}$$

since $\dfrac{1}{N-1}\displaystyle\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(\mathbf{q}_{k,m}-\mathbf{q}_k)^T$ is itself an approximation of $\mathbf{C}^k_{\mathbf{q}_k}$.

By using the samples $\{\mathbf{q}_{k-1,m}\}_{m=1}^N$ and the covariance matrix $\mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}$ from the last iteration, we update $\mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}$, like in [5], by setting

$$\mathbf{C}^k_{\mathbf{q}_k} = \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}} + \tilde{\beta}_k\tilde{\mathbf{d}}_k \text{ with} \tag{3.4}$$

$$\tilde{\mathbf{d}}_k = N^{-1}\sum_{m=1}^{N}(F(\mathbf{q}_{k-1,m}) - F(\mathbf{q}_k))((\mathbf{q}_{k-1,m}-\mathbf{q}_k)(\mathbf{q}_{k-1,m}-\mathbf{q}_k)^T - \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}), \tag{3.5}$$

where $\tilde{\beta}_k$ is a step size that is chosen so that no entries of the diagonal of $\mathbf{C}^k_{\mathbf{q}_k}$ are negative. How we set $\tilde{\beta}_k$ is shown below at the implementation of the entire EnOpt algorithm.

Now that we have described the optimization steps of this algorithm, we iterate until $F(\mathbf{q}_k) \leq F(\mathbf{q}_{k-1}) + \varepsilon$, where $\varepsilon > 0$.

In our implementation, the EnOpt algorithm takes the objective function $\mathrm{F} : \mathbb{R}^{N_\mathbf{q}} \to \mathbb{R}$, our initial iterate $\mathbf{q}_0 \in \mathbb{R}^{N_\mathbf{q}}$, the sample size $N \in \mathbb{N}$, the tolerance $\varepsilon > 0$, the maximum number of iterations $k^* \in \mathbb{N}$, the initial step size $\beta_1 > 0$ for the computation of the next iterate, the initial step size $\beta_2 > 0$ for the iteration of the covariance matrix, the step size contraction $r \in (0,1)$, the maximum number of step size trials $\nu^* \in \mathbb{N}$, the variance $\sigma^2 \in \mathbb{R}^{N_b}$ with positive elements, the correlation coefficient $\rho \in (-1,1)$, the number of time steps $N_t \in \mathbb{N}$, the number of basis functions $N_b \in \mathbb{N}$, a projection PR and an initial

covariance $\mathbf{C}_{\text{init}} \in \mathbb{R}^{N_\mathbf{q}}$. PR is set to the identity function **lambda** mu : mu and $\mathbf{C}_{\text{init}}$ is set to None by default.

PR is used to project the inputs onto a given set and $\mathbf{C}_{\text{init}}$ is an alternative definition for the initialization of the covariance. In this chapter we do not need to specify these inputs, however PR could be used if we had an optimization problem where the iterates were restricted to a certain spatial domain, which is here not the case.

The implementation in pseudo code is shown below:

Algorithm 1 EnOpt algorithm

1: **function** EnOpt(F, $\mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$, PR $\leftarrow$ **lambda** mu : mu, $\mathbf{C}_{\text{init}} \leftarrow$ None)
2: $F_k^{\text{prev}} \leftarrow \text{F}(\mathbf{q}_0)$
3: $\mathbf{q}_k, T_k, \mathbf{C}_k, F_k \leftarrow$ OptStep(F, $\mathbf{q}_0, N, 0, [\,], \mathbf{C}_{\text{init}}, F_k^{\text{prev}}, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b$, PR)
4: $k \leftarrow 1$
5: **while** $F_k > F_k^{\text{prev}} + \varepsilon$ and $k < k^*$ **do**
6: $F_k^{\text{prev}} \leftarrow F_k$
7: $\mathbf{q}_k, T_k, \mathbf{C}_k, F_k \leftarrow$ OptStep(F, $\mathbf{q}_k, N, k, T_k, \mathbf{C}_k, F_k, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b$, PR)
8: $k \leftarrow k + 1$
9: **return** $\mathbf{q}_k, k$

We begin by initializing F_k^{prev} as $\text{F}(\mathbf{q}_0)$. The next iterate $\mathbf{q}_k$, along with the sample T_k, the covariance $\mathbf{C}_k$ and the function value of $\mathbf{q}_k$, denoted by F_k, are computed by calling the initial optimization step OptStep, which is shown in algorithm 2.

After we initialized k as 1, we loop until the stop criterion is satisfied. In each optimization loop, F_k^{prev} is set to the function value of the last iterate and $\mathbf{q}_k, T_k, \mathbf{C}_k$ and F_k are updated by calling OptStep again.

The next algorithms use some functions from Python such as len for the length of a list or an array, as well as functions from the Python package NumPy, which are identified by starting with 'np.'.

Algorithm 2 OptStep algorithm

function OptStep(F, $\mathbf{q}_k, N, k, T_k, \mathbf{C}_k, F_k, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b$, PR $\leftarrow$ **lambda** mu : mu)
  $N_{\mathbf{q}} \leftarrow$ len($\mathbf{q}_k$)
  $\mathbf{C}_k^{\text{next}} \leftarrow$ np.zeros(($N_{\mathbf{q}}, N_{\mathbf{q}}$))
  if $k = 0$ **then**
    if $\mathbf{C}_k$ is None **then**
      $\mathbf{C}_k^{\text{next}} \leftarrow$ initCov(len($\mathbf{q}_k$), σ^2, ρ, N_t, N_b)
    else
      $\mathbf{C}_k^{\text{next}} \leftarrow \mathbf{C}_k$.copy( )
  else
    $\mathbf{C}_k^{\text{next}} \leftarrow$ updateCov($\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2$)
  sample $\leftarrow$ np.random.multivariate_normal($\mathbf{q}_k, \mathbf{C}_k^{\text{next}}$, size $\leftarrow N$)
  $T_k^{\text{next}} \leftarrow [\text{PR}(\text{sample}[j]), \text{F}(\text{PR}(\text{sample}[j]))]_{j=0}^{N-1}$
  $\mathbf{C}_F^k \leftarrow$ np.zeros($N_{\mathbf{q}}$)
  for $m = 0, \ldots, N-1$ **do**
    $\mathbf{C}_F^k \leftarrow \mathbf{C}_F^k + (T_k^{\text{next}}[m][0] - \mathbf{q}_k) \cdot (T_k^{\text{next}}[m][1] - F_k)$
  $\mathbf{C}_F^k \leftarrow 1/(N-1) \cdot \mathbf{C}_F^k$
  $\mathbf{d}_k \leftarrow$ np.zeros($N_{\mathbf{q}}$)
  $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}} \leftarrow$ np.copy($\mathbf{q}_k$), F_k
  if not np.all($\mathbf{C}_F^k == 0$) **then**
    $\mathbf{d}_k \leftarrow \mathbf{C}_F^k / \|\mathbf{C}_F^k\|_\infty$
    $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}} \leftarrow$ LineSearch(F, $\mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*$, PR)
  return $\mathbf{q}_k^{\text{next}}, T_k^{\text{next}}, \mathbf{C}_k^{\text{next}}, F_k^{\text{next}}$

We start the OptStep algorithm by updating the covariance matrix $\mathbf{C}_k$ or, if k is zero, initializing it. In the case that k is zero, we first check if there is a predefined covariance matrix $\mathbf{C}_k$. When there is one, $\mathbf{C}_k^{\text{next}}$ is set to that matrix. Otherwise, we define $\mathbf{C}_k^{\text{next}}$ by calling the function initCov, which sets the matrix like it is described in (3.2).

If k is not zero, we get the updated covariance matrix by calling updateCov, which is described in (3.4) and algorithm 3.

This covariance matrix is now used to get a Gaussian distributed sample with N elements around $\mathbf{q}_k$. The projected samples PR(sample[j]) and their respective function values F(PR(sample[j])) for $j = 0, \ldots, N-1$ are stored in the list T_k^{next}. The definition is here abbreviated as $[\text{PR}(\text{sample}[j]), \text{F}(\text{PR}(\text{sample}[j]))]_{j=0}^{N-1}$. In the code, this is done with an iteration through a for-loop.

After that, we set $\mathbf{C}_F^k$ like it is defined in (3.3). Then we check if $\mathbf{C}_F^k$ is equal to the zero vector. That can happen if the functional F is constant on the sample set. In that case, we let the next iterate $\mathbf{q}_k^{\text{next}}$ and the corresponding functional value F_k unchanged.

Otherwise, we set $\mathbf{d}_k$ to $\mathbf{C}_F^k$ divided by its sup norm. Here, $\|\mathbf{C}_F^k\|_\infty$ is an abbreviation of np.max(np.abs($\mathbf{C}_F^k$)). The next iterate $\mathbf{q}_k^{\text{next}}$ and its functional value F_k^{next} is now computed with the line search algorithm LineSearch, that is shown in algorithm 4.

We return $\mathbf{q}_k^{\text{next}}, T_k^{\text{next}}, \mathbf{C}_k^{\text{next}}$ and F_k^{next}.

Algorithm 3 Covariance matrix update

function UPDATECOV($\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2$)
  $N_{\mathbf{q}} \leftarrow$ LEN($\mathbf{q}_k$)
  $N \leftarrow$ LEN(T_k)
  $\mathbf{d}_k^{\text{cov}} \leftarrow$ NP.ZEROS($(N_{\mathbf{q}}, N_{\mathbf{q}})$)
  for $m = 0, \ldots, N-1$ **do**
    $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}} + (T_k[m][1] - F_k) \cdot ((T_k[m][0] - \mathbf{q}_k)$.RESHAPE$((N_{\mathbf{q}}, 1)) \cdot (T_k[m][0] - \mathbf{q}_k)$.RESHAPE$((1, N_{\mathbf{q}})) - \mathbf{C}_k)$
  $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}} / N$
  $\mathbf{C}_{\text{diag}}, \mathbf{d}_{\text{diag}} \leftarrow$ np.ZEROS$(N_{\mathbf{q}})$, np.ZEROS$(N_{\mathbf{q}})$
  for $i = 0, \ldots, N_{\mathbf{q}} - 1$ **do**
    $\mathbf{C}_{\text{diag}}[i] \leftarrow \mathbf{C}_k[i, i]$
    $\mathbf{d}_{\text{diag}}[i] \leftarrow \mathbf{d}_k^{\text{cov}}[i, i]$
  $\beta_2^{\text{iter}} \leftarrow \beta_2$
  while np.MIN$(\mathbf{C}_{\text{diag}} + \beta_2^{\text{iter}} \cdot \mathbf{d}_{\text{diag}}) \leq 0$ **do**
    $\beta_2^{\text{iter}} \leftarrow \beta_2^{\text{iter}} / 2$
  return $\mathbf{C}_k + \beta_2^{\text{iter}} * \mathbf{d}_k^{\text{cov}}$

For the update of the covariance matrix, we calculate the step direction $\mathbf{d}_k^{\text{cov}}$ like in 3.5 first. Then we want to make sure that the updated covariance matrix has only positive values on its diagonal.

For this purpose, $\mathbf{C}_{\text{diag}}$ and $\mathbf{d}_{\text{diag}}$ are defined as the vector of values on the diagonal of $\mathbf{C}_k$ and $\mathbf{d}_k^{\text{cov}}$ respectively. Then, the step size β_2^{iter}, initialized as β_2, is halfed until np.MIN$(\mathbf{C}_{\text{diag}} + \beta_2^{\text{iter}} * \mathbf{d}_{\text{diag}})$ is positive. Finally, the matrix $\mathbf{C}_k + \beta_2^{\text{iter}} * \mathbf{d}_k^{\text{cov}}$ is returned, like in (3.4).

Algorithm 4 Line search

function LINESEARCH(F, $\mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*$, PR)
  $\beta_1^{\text{iter}} \leftarrow \beta_1$
  $\mathbf{q}_k^{\text{next}} \leftarrow \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$
  $F_k^{\text{next}} \leftarrow \text{F}(\mathbf{q}_k^{\text{next}})$
  $\nu \leftarrow 0$
  while $F_k^{\text{next}} - F_k \leq \varepsilon$ and $\nu < \nu^*$ **do**
    $\beta_1^{\text{iter}} \leftarrow r \beta_1^{\text{iter}}$
    $\mathbf{q}_k^{\text{next}} \leftarrow \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$
    $F_k^{\text{next}} \leftarrow \text{F}(\mathbf{q}_k^{\text{next}})$
    $\nu \leftarrow \nu + 1$
  return $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}}$

At the start of the line search algorithm, we initialize the step size β_1^{iter} as β_1. Then we repeatedly calculate $\mathbf{q}_k^{\text{next}} = \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$ and $F_k^{\text{next}} = \text{F}(\mathbf{q}_k^{\text{next}})$ with simultaneous reduction of β_1^{iter} by multiplication with r until either $F_k^{\text{next}} - F_k > \varepsilon$ or $\nu \geq \nu^*$.

After the termination of the while-loop, $\mathbf{q}_k^{\text{next}}$ and F_k^{next} are returned. The reason for stopping the while loop also shows when $\mathbf{q}_k^{\text{next}}$ is the last iteration of the EnOpt algorithm, as the EnOpt algorithm stops when the function value of the next iteration is not greater than the function value of the last iteration plus ε, i.e. when $F_k^{\text{next}} > F_k + \varepsilon$.

Now we use this algorithm to optimize our objective function j. Since this is a maximization procedure and j should be minimized, we apply $-j$, denoted by $-\textsc{j}$, to the EnOpt algorithm, which gives us:

Algorithm 5 FOM-EnOpt algorithm

1: **function** FOM-EnOpt($\mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \mathbf{q}_{\text{base}}$)
2: **return** EnOpt($-\textsc{j}, \mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \textsc{len}(\mathbf{q}_{\text{base}})$)

$\mathbf{q}_{\text{base}}$ is here a list that consists of the basis functions, so Φ in (2.7). There are some more inputs that FOM-EnOpt requires, but we omit these as they are only needed for the calculation of j.

4 Adaptive-ML-EnOpt algorithm

In this chapter, we introduce the Adaptive-ML-EnOpt algorithm [1], which is a modified version of the EnOpt algorithm. This algorithm is supposed to reduce the number of FOM evaluations by using a machine learning-based surrogate functional, which improves the computation speed with respect to the EnOpt algorithm. Therefore, we introduce deep neural networks (DNNs) next. After that, the Adaptive-ML-EnOpt-algorithm is presented.

4.1 Deep neural networks

This description of deep neural networks is based on the definitions in [1].

DNNs are used here to approximate a function $f : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$ with $N_{\text{in}}, N_{\text{out}} \in \mathbb{N}$. We call $L \in \mathbb{N}$ the number of layers and $N_{\text{in}} = N_0, N_1, \ldots, N_{L-1}, N_L = N_{\text{out}}$ the number of neurons in each layer. We refer to the layers 1 to $L-1$ as the hidden layers. $W_i \in \mathbb{R}^{N_i \times N_{i-1}}$ denotes the weights in layer $i \in \{1, \ldots, L\}$ and $b_i \in \mathbb{R}^{N_i}$ the biases of the layer $i \in \{1, \ldots, L\}$. These are composed as $\mathbf{W} = ((W_1, b_1), \ldots, (W_L, b_L))$, which is a tuple of pairs of corresponding weights and biases.

$\rho : \mathbb{R} \to \mathbb{R}$ is the so-called activation function. A popular example is the rectified linear unit funtion $\rho(x) = \max(x, 0)$, however we will use the hyperbolic tangent funtion

$$\rho(x) = \tanh(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1}.$$

$\rho_n^* : \mathbb{R}^n \to \mathbb{R}^n$ is now defined as the component-wise application of ρ onto a vector of dimension n, so $\rho_n^*(x) = [\rho(x_1), \ldots, \rho(x_n)]^T$ for $x \in \mathbb{R}^n$.

To calculate the output $\Phi_{\mathbf{W}}(x) \in \mathbb{R}^{N_{\text{out}}}$ of a DNN for an input $x \in \mathbb{R}^{N_{\text{in}}}$, we apply the weights, biases, and activation function multiple times onto the input. It is calculated iteratively as shown here:

$$\begin{aligned} r_0(x) &:= x, \\ r_i(x) &:= \rho_{N_i}^*(W_i r_{i-1}(x) + b_i) \text{ for } i = 1, \ldots, L-1, \\ r_L(x) &:= W_L r_{L-1}(x) + b_L, \\ \Phi_{\mathbf{W}}(x) &:= r_L(x). \end{aligned}$$

Now we try to optimize the parameters in $\mathbf{W}$ such that $\Phi_{\mathbf{W}} \approx f$. To achieve this, we sample a set that consists of inputs $x_i \in X \subset \mathbb{R}^{N_{\text{in}}}$ and corresponding outputs $f(x_i) \in \mathbb{R}^{N_{\text{out}}}$ and assemble them in the training set

$$T_{\text{train}} = \{(x_1, f(x_1)), \ldots, (x_{N_{\text{train}}}, f(x_{N_{\text{train}}}))\} \subset X \times \mathbb{R}^{N_{\text{out}}}. \tag{4.1}$$

To evaluate the performance of our chosen $\mathbf{W}$, we use the mean squared error loss $\mathscr{L}(\Phi_{\mathbf{W}}, T_{\text{train}})$ to measure the distance between $\Phi_{\mathbf{W}}$ and f on a training set. The mean squared error loss

is defined as

$$\mathscr{L}(\Phi_{\mathbf{W}}, T_{\text{train}}) := \frac{1}{|T_{\text{train}}|} \sum_{(x,y)\in T_{\text{train}}} \|\Phi_{\mathbf{W}}(x) - y\|_2^2.$$

Since we want $\Phi_{\mathbf{W}}$ to be close to f, we minimize the loss function with respect to $\mathbf{W}$. For that, we use some gradient-based optimization method. By the structure of the DNN, we can use the chain rule multiple times to divide the gradient of $\mathscr{L}$ into much simpler gradient computations.

We want that $\Phi_{\mathbf{W}}$ is close to f on X but we train it only on a sample set of X, so we achieve that $\Phi_{\mathbf{W}}$ is only on T_{train} close to f. While we train, the mean squared error loss will eventually get better and better on the training set, but at some point the error on different samples will get worse [6]. This is called 'overfitting'.

To prevent overfitting, we use early stopping. For early stopping, we evaluate the loss function on a validation set $T_{\text{val}} \subset X \times \mathbb{R}^{N_{\text{out}}}$, where usually $T_{\text{val}} \cap T_{\text{train}} = \emptyset$. Our algorithm for early stopping looks like this:

- let $\mathbf{W}^{(k)}$ be the weights in epoch k
- compute $\mathscr{L}(\Phi_{\mathbf{W}^{(k)}}, T_{\text{val}})$ in each epoch
- save $\mathbf{W}^{(k^*)}$ at iteration k^* if it is the minimizer over all previous weights
- if $\mathscr{L}(\Phi_{\mathbf{W}^{(k^*+i)}}, T_{\text{val}}) \geq \mathscr{L}(\Phi_{\mathbf{W}^{(k^*)}}, T_{\text{val}})$ for all i from 0 to a prescribed number: abort the training and use $\mathbf{W}^{(k^*)}$

So we abort the training if the minimum loss is not decreasing over a prescribed number of consecutive epochs. Our reasoning behind this is that the loss on the validation set is not srictly decreasing and can even increase over some epochs, but that is fine for us as long as we can decrease the loss over time.

We present now the construction of a neural network that approximates a function $f : \mathbb{R}^n \to \mathbb{R}$ with $n \in \mathbb{N}$. The training of one neural network is shown in algorithm 6. It takes the initialization of the deep neural network (DNN), the inputs and outputs of the training set $(x_{\text{train}}, y_{\text{train}})$, the inputs and outputs of the validation set $(x_{\text{val}}, y_{\text{val}})$, the loss function (LOSS_FN), the optimizer (optimizer), the number of training epochs (epochs) and the number (earlyStop) that describes after how many iterations without improvement of the validation loss the training gets aborted.

In our algorithm, the loss function is chosen as the mean squared error loss and we use the L-BFGS optimizer with strong Wolfe line-search as our optimizer. The number of training epochs is only the maximum number of iterations since we apply early stopping to our training algorithm. Usually, the training terminates earlier because the loss over the validation set is not decreasing further.

The function TESTDNN in algorithm 6 returns the loss of the function LOSS_FN between the output of the DNN with the current parameters and the output of the objective function over the validation set which is denoted by y_{val}. So if LOSS_FN $= \mathscr{L}$, we have TESTDNN$(\text{DNN}, x_{\text{val}}, y_{\text{val}}, \text{LOSS_FN}) = \mathscr{L}(\text{DNN}, T_{\text{val}})$ with

$$T_{\text{val}} = \{((x_{\text{val}})_1, (y_{\text{val}})_1), \ldots, ((x_{\text{val}})_{N_{\text{val}}}, (y_{\text{val}})_{N_{\text{val}}})\}.$$

In addition, some funtions are imported from the Python package PyTorch. These functions are identified by the beginning 'torch.'.

Algorithm 6 DNN training

```
1:  function trainDNN(DNN, x_train, y_train, x_val, y_val, loss_fn, optimizer, epochs, earlyStop)
2:      wait ← 0
3:      minimalValidationLoss ← testDNN(DNN, x_val, y_val, loss_fn)
4:      torch.save(DNN.state_dict( ), 'checkpoint.pth')
5:      for epoch = 1, ..., epochs do
6:          DNN.train( )
7:          function closure( )
8:              y_pred ← DNN(x_train).reshape(len(y_train))
9:              loss ← loss_fn(y_pred, y_train)
10:             optimizer.zero_grad( )
11:             loss.backward( )
12:             return loss
13:         optimizer.step(closure)
14:         validationLoss ← testDNN(DNN, x_val, y_val, loss_fn)
15:         if validationLoss < minimalValidationLoss then
16:             wait ← 0
17:             minimalValidationLoss ← validationLoss
18:             torch.save(DNN.state_dict( ), 'checkpoint.pth')
19:         else
20:             wait ← wait + 1
21:         if wait ≥ earlyStop then
22:             DNN.load_state_dict(torch.load('checkpoint.pth'))
23:             return
```

We start the algorithm trainDNN by initializing the variable wait, which indicates the number of training epochs without a decrease of the loss on the evaluation set, as zero and the variable minimalValidationLoss, which shows the loss that was achieved 'wait' epochs ago, as the loss on the validation set for the DNN before the training begins. Then the weights and biases of the DNN are saved in the file 'checkpoint.pth'.

After that, the following operations are executed in every training epoch. The procedures that are performed between line 6 and line 13 can be described as doing one optimization step with the optimizer to decrease the loss on the training set by adjusting the weights and biases of the DNN. Then we check if the loss on the evaluation set is currently smaller than the minimal validation loss over all previous epochs.

If that is the case, the variable wait is set to zero, indicating that the current parameters of the DNN have the best performance over the validation set, and the variable minimalValidationLoss is updated to the current validation loss. Since the parameters of the DNN will be changed in the next epochs, the current weights and biases are saved again in the file 'checkpoint.pth'.

If the validation loss is not less than the minimal validation loss, the variable wait is increased by one.

To implement early stopping as described above, we check at the end of each training epoch whether the minimum loss has not decreased over so many consecutive epochs that

we terminate the algorithm prematurely. If wait $\geq$ earlyStop, the current parameters of the neural network are overwritten with the parameters that were saved when the minimum loss was reached and the algorithm is aborted.

Since we search for local minima of the loss function, the initial value $\mathbf{W}^{(0)}$ of our iteration effects the local optimum that we get and therefore the performance. We use Kaiming initialization [7] to set our initial value $\mathbf{W}^{(0)}$. With Kaiming initialization, the starting values are initialized randomly since the elements of the weights W_i are sampled from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{2/N_{i-1}}$ for $i \in \{1, \ldots, L\}$. The biases b_i are set to zero for $i \in \{1, \ldots, L\}$. The idea behind the random sampling is that the specified standard deviation prevents the exponential increase/ reduction of the input as shown in [7].

For the training of the DNN, we perform multiple restarts of the training algorithm with different initializations of $\mathbf{W}^{(0)}$ which minimizes the dependence of our neural network from the initial values. After we have trained enough DNNs, we select the neural network $\Phi_{\mathbf{W}^*}$ that has the smallest evaluation loss $\mathscr{L}(\Phi_{\mathbf{W}^*}, T_{\text{val}})$ over all restarts.

Before the whole algorithm for the construction of the DNN is presented, we look at the data that we use for the training. If we sample the inputs in a small area, it is likely that the corresponding outputs are also close to each other. Since we convert the values for the training of the neural network from 64-bit floating point numbers to 32-bit floating point numbers, it can even happen that the converted inputs or outputs are constant. In that case, the digits of these values that differ from each other get cut off at the conversion.

We want the values of the inputs and outputs to be distributed in such a way that significant differences are correctly represented. For that, the inputs $x \in \mathbb{R}^n$ and outputs $y \in \mathbb{R}$ are scaled to $\tilde{x} \in [0,1]^n$ and $\tilde{y} \in [0,1]$.

Let

$$T = \{(x_1, y_1), \ldots, (x_N, y_N)\} \tag{4.2}$$

be a sample set of size N and

$$T_x = \{x_1, \ldots, x_N\}, \qquad T_y = \{y_1, \ldots, y_N\}$$

the sets that contain the inputs and outputs of that sample.

We define $x^{\text{low}}, x^{\text{upp}} \in \mathbb{R}^n$ and $y^{\text{low}}, y^{\text{upp}} \in \mathbb{R}$ as

$$\begin{aligned} x_i^{\text{low}} &:= \min\{x_i \mid x \in T_x\} \text{ for } i = 1, \ldots, n, && (4.3)\\ x_i^{\text{upp}} &:= \max\{x_i \mid x \in T_x\} \text{ for } i = 1, \ldots, n, && (4.4)\\ y^{\text{low}} &:= \min T_y, && (4.5)\\ y^{\text{upp}} &:= \max T_y. && (4.6) \end{aligned}$$

Now, $\tilde{x}$ and $\tilde{y}$ are calculated as

$$\tilde{x}_i = \frac{x_i - x_i^{\text{low}}}{x_i^{\text{upp}} - x_i^{\text{low}}} \text{ for } i = 1, \ldots, n, \qquad \tilde{y} = \frac{y - y^{\text{low}}}{y^{\text{upp}} - y^{\text{low}}}. \tag{4.7}$$

After we have trained the neural network, the DNN outputs need to be rescaled so that we get a proper approximation of the function f. The output $\Phi(\tilde{x})$ of the DNN Φ is rescaled with the calculation $\Phi(\tilde{x}) \cdot (y^{\text{upp}} - y^{\text{low}}) + y^{\text{low}}$.

To summarize this, we present now the construction of a DNN as pseudo code in algorithm 7. Training parameters like the neural network structure "DNNStructure", the activation function "activFunc", the number of restarts of different DNN initializations "restarts", the number of training epochs "epochs", the number of epochs without decrease of the evaluation loss after which early stopping is applied "earlyStop", the fraction of the sample that is used for training "trainFrac" and the learning rate "learning_rate" are stored in V_{DNN}. We denote x^{low} as minIn, x^{upp} as maxIn, y^{low} as minOut and y^{upp} as maxOut. minIn and maxIn are calculated before the construction of the DNN and are taken as an input. The function FullyConnectedNN is imported from pymor.models.neural_network which is included in the Python package pyMor. It builds a neural network with Kaiming initialization where the number of neurons in each layer gets specified by the first and the activation function of the neural network by the second argument.

Algorithm 7 DNN construction

1: **function** CONSTRUCTDNN(sample, V_{DNN}, minIn, maxIn)
2: normSample ← np.ZEROS(LEN(sample), LEN(sample[0][0]))
3: normVal ← np.ZEROS(LEN(sample))
4: **for** $i = 0, \ldots,$ LEN(sample) $- 1$ **do**
5: normSample$[i, :]$ ← sample$[i][0]$
6: normVal$[i]$ ← sample$[i][1]$
7: minOut ← np.MIN(normVal)
8: maxOut ← np.MAX(normVal)
9: normSample ← (normSample − minIn)/(maxIn − minIn)
10: normVal ← (normVal − minOut)/(maxOut − minOut)
11: x ← torch.FROM_NUMPY(normSample).TO(torch.float32)
12: y ← torch.FROM_NUMPY(normVal).TO(torch.float32)
13: trainSplit ← INT(trainFrac ∗ LEN(x))
14: $x_{\text{train}}, y_{\text{train}} \leftarrow x[:\text{trainSplit}], y[:\text{trainSplit}]$
15: $x_{\text{val}}, y_{\text{val}} \leftarrow x[\text{trainSplit}:], y[\text{trainSplit}:]$
16: DNN ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
17: LOSS_FN ← torch.nn.MSELOSS( )
18: optimizer ← torch.optim.LBFGS(DNN.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
19: TRAINDNN(DNN, $x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
20: evalDNN ← TESTDNN(DNN, $x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
21: **for** $i = 1, \ldots,$ restarts **do**
22: DNN_i ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
23: optimizer ← torch.optim.LBFGS(DNN_i.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
24: TRAINDNN($\text{DNN}_i, x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
25: evalDNN_i ← TESTDNN($\text{DNN}_i, x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
26: **if** $\text{evalDNN}_i <$ evalDNN **then**
27: evalDNN ← evalDNN_i
28: DNN ← DNN_i
29: **function** $\text{F}_{\text{ML}}(x_{\text{inp}})$
30: scaledInput ← torch.FROM_NUMPY($(x_{\text{inp}}$ − minIn)/(maxIn − minIn)).TO(torch.float32)
31: scaledOutput ← DNN(scaledInput)
32: **return** scaledOutput.NUMPY( )$[0] \cdot$ (maxOut − minOut) + minOut
33: **return** F_{ML}

The algorithm CONSTRUCTDNN begins by setting the scaled samples for training and testing. For that, normSample is defined as a matrix where each row i is equal to the input sample sample$[i][0]$ and normVal is a vector where each element i is set to the output sample sample$[i][1]$. Then we define them like in (4.7).

After normSample and normVal are converted to a 32-bit floating point data type tensor from the torch package in lines 11 and 12, the training samples x_{train} and y_{train} are set to

a fraction of size trainFrac from the scaled and converted sample. The rest is used for the evaluation samples x_{val} and y_{val}.

Next, the neural network DNN with the structure DNNStructure and the activation function activFunc is initialized with Kaiming initialization. As an example, if DNNStructure would be equal to $[N_0, N_1, \ldots, N_L]$, we would get a DNN with L layers and N_i neurons in layer $i = 0, 1, \ldots, L$.

After the loss function is set to the MSE loss and the optimizer is set to the L-BFGS optimizer, we train the neural network DNN by calling trainDNN from the algorithm 6. evalDNN in line 20 is the loss of DNN on the scaled and converted evaluation set.

In the for-loop, multiple neural networks DNN_i are trained. If there is a neural network with a loss that is less than evalDNN, we overwrite DNN with the neural network DNN_i that has the smalles loss on the evaluation set.

After the for-loop, the function F_{ML} is defined. It takes an input x_{inp} and scales it like in line 9 while also converting it to a 32-bit float tensor like in line 11. Then the function calls DNN on the scaled and converted input, converts the DNN output back to a non-tensor float and rescales it in an inverted way compared to line 10.

Finally, F_{ML} is returned by constructDNN.

4.2 Modifying the EnOpt algorithm by using a neural network-based surrogate

For the next step, we use neural networks like in [1] to get an improved version of the EnOpt algorithm. Here, we want to replace calls of the FOM functional by a surrogate functional that is based on a neural network. The surrogate is supposed to be a local approximation of the FOM functional around the current iterate. Defining a surrogate functional with globally sufficient accuracy would be computationally too expensive.

Therefore we will introduce a trust-region (TR) approach [8]. Trust-region methods are applied in cases like this where we have a surrogate that can be trusted to be a good representation of the objective functional within a certain region around the current iterate. The idea is to do one optimization step so that the next iterate is in this region. After that, the quality of the new iterate is evaluated. If this iterate is acceptable, so if the corresponding objective functional value gives a better result, the trust-region might be enlarged for the next iteration step. In case the iterate yields not a sufficient improvement of the objective functional value, the trust-region is reduced and the last iteration step could be repeated in this new region.

We describe now the Adaptive-ML-EnOpt algorithm. This algorithm takes a functional F, the initial guess $\mathbf{q}_0 \in \mathbb{R}^{N_{\mathbf{q}}}$, the sample size $N \in \mathbb{N}$, the tolerances $\varepsilon_o, \varepsilon_i > 0$ for the outer and inner iterations, the maximum number of outer and inner iterations $k_o^*, k_i^* \in \mathbb{N}$, the DNN-specific variables V_{DNN}, the initial step size $\beta > 0$, the step size contraction $r \in (0, 1)$ and the maximum number of step size trials $\nu^* \in \mathbb{N}$.

Algorithm 8 Adaptive-ML-EnOpt algorithm

function AML-EnOpt($\mathrm{F}, \mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, V_{\mathrm{DNN}}, \delta_{\mathrm{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$)
  $N_{\mathbf{q}} \leftarrow \mathrm{LEN}(\mathbf{q}_0)$
  $F_k \leftarrow \mathrm{F}(\mathbf{q}_0)$
  $F_k^{\mathrm{next}} \leftarrow F_k$
  $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow \mathrm{OptStep}(\mathrm{F}, \mathbf{q}_0, N, 0, [\,], \mathrm{None}, F_k, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_n)$
  $k \leftarrow 1$
  $\mathbf{q}_k \leftarrow \mathbf{q}_0$
  $\mathbf{q}_k^{\mathrm{next}} \leftarrow \mathbf{q}_k.\mathrm{COPY}(\,)$
  $\delta \leftarrow \delta_{\mathrm{init}}$
  while $\tilde{F}_k > F_k + \varepsilon_o$ and $k < k_o^*$ **do**
    $T_k^x \leftarrow \mathrm{np.ZEROS}((N, N_{\mathbf{q}}))$
    for $i = 0, \ldots, N-1$ **do**
      $T_k^x[i, :] \leftarrow T_k[i][0]$
    $\mathrm{minIn} \leftarrow \mathrm{np.ZEROS}(N_{\mathbf{q}})$
    $\mathrm{maxIn} \leftarrow \mathrm{np.ZEROS}(N_{\mathbf{q}})$
    for $i = 0, \ldots, N_{\mathbf{q}} - 1$ **do**
      $\mathrm{minIn}[i] \leftarrow \mathrm{np.MIN}(T_k^x[:, i])$
      $\mathrm{maxIn}[i] \leftarrow \mathrm{np.MAX}(T_k^x[:, i])$
    $\mathbf{d}_k \leftarrow |\mathbf{q}_k - \tilde{\mathbf{q}}_k|$
    $\mathrm{tr} \leftarrow 1$
    while $F_k^{\mathrm{next}} \leq F_k + \varepsilon_o$ **do**
      assert $\mathrm{tr} \leq k_{\mathrm{TR}}^*$
      $\mathrm{F}_{\mathrm{ML}}^k \leftarrow \mathrm{constructDNN}(T_k, V_{\mathrm{DNN}}, \mathrm{minIn}, \mathrm{maxIn})$
      $F_k^{\mathrm{approx}} \leftarrow \mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k)$
      $\mathrm{flag}_{\mathrm{TR}} \leftarrow \mathrm{True}$
      while $\mathrm{flag}_{\mathrm{TR}}$ **do**
        $\mathbf{d}_k^{\mathrm{iter}} \leftarrow \delta \cdot \mathbf{d}_k$
        $\mathbf{q}_k^{\mathrm{next}} \leftarrow \mathrm{EnOpt}(\mathrm{F}_{\mathrm{ML}}^k, \mathbf{q}_k, N, \varepsilon_i, k_i^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b, \mathrm{PR} \leftarrow \mathbf{lambda}\ \mathrm{mu} : \mathrm{TR\text{-}Projection}(\mathrm{mu}, \mathbf{q}_k, \mathbf{d}_k^{\mathrm{iter}}), \mathbf{C}_{\mathrm{init}} \leftarrow \mathbf{C}_k)[0]$
        $F_k^{\mathrm{next}} \leftarrow \mathrm{F}(\mathbf{q}_k^{\mathrm{next}})$
        $\rho_k \leftarrow \dfrac{F_k^{\mathrm{next}} - F_k}{\mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k^{\mathrm{next}}) - F_k^{\mathrm{approx}}}$
        if $\rho_k < 0.25$ **then**
          $\delta \leftarrow 0.25 \cdot \delta$
        else
          if $\rho_k > 0.75$ and $\mathrm{np.ANY}(\mathrm{np.ABS}(\mathbf{q}_k - \mathbf{q}_k^{\mathrm{next}}) - \mathbf{d}_k^{\mathrm{iter}} = 0)$ **then**
            $\delta \leftarrow 2 \cdot \delta$
        if $\rho_k > 0$ **then**
          $\mathrm{flag}_{\mathrm{TR}} \leftarrow$ **False**
      $\mathrm{tr} \leftarrow \mathrm{tr} + 1$
    $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow \mathrm{OptStep}(\mathrm{F}, \mathbf{q}_k^{\mathrm{next}}, N, k, T_k, \mathbf{C}_k, F_k^{\mathrm{next}}, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_b)$
    $F_k \leftarrow F_k^{\mathrm{next}}$
    $\mathbf{q}_k \leftarrow \mathbf{q}_k^{\mathrm{next}}.\mathrm{COPY}(\,)$
    $k \leftarrow k + 1$
  return $\mathbf{q}^* \leftarrow \mathbf{q}_k$

We start the Adaptive-ML-EnOpt algorithm by initializing $N_{\mathbf{q}}$ as the length of $\mathbf{q}_0$ and F_k and F_k^{next} as the FOM objective functional value at $\mathbf{q}_0$. Similar to the EnOpt algorithm 1, we set $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k$ to the output of the function OptStep on the FOM objective functional F. Instead of repeating the optimization step algorithm on the FOM objective functional and using $\tilde{\mathbf{q}}_k$ as our iterate, we initialize the iterates $\mathbf{q}_k$ and $\mathbf{q}_k^{\text{next}}$ as $\mathbf{q}_0$. The $\tilde{F}_k$ from line 5 is then used in line 10 to check if the FOM objective functional value can be improved by more than ε_o. The while-loop is repeated until this is no longer the case.

In the while loop, x^{low} from (4.3), denoted as minIn, and x^{upp} from (4.4), denoted as maxIn, are computed first. This happens between line 11 and line 18. $\mathbf{d}_k$ is the absolute value of the difference between our current iterate $\mathbf{q}_k$ and the $\tilde{\mathbf{q}}_k$ that resulted from the FOM optimization step in line 5. This is np.abs($\mathbf{q}_k - \tilde{\mathbf{q}}_k$) in the code and abbreviated as $|\mathbf{q}_k - \tilde{\mathbf{q}}_k|$ in line 19.

Now we want to compute the next iterate. We know from line 10 that it is possible to increase the FOM objective functional value of the current iterate by at least ε_o, for example by setting it to $\tilde{\mathbf{q}}_k$. Instead of using the $\tilde{\mathbf{q}}_k$ that we got from evaluations of the full order model F, the neural network-based surrogate functional F_{ML}^k is introduced by calling constructDNN in line 23. It should be noticed here that the sample T_k, that was obtained from the FOM optimization step in line 5 (and for the next outer iterations from line 39), is used for the training and testing of this DNN. This means that only samples around $\mathbf{q}_k$ are used for the training. Therefore we expect that the error between the FOM objective functional F and its surrogate F_{ML}^k is only sufficiently small for points that are close to $\mathbf{q}_k$. To take this into account, we use a trust-region method like in [8].

Here we repeat a while-loop until a certain condition is met. We allow this while-loop to only repeat k_{TR}^* times. If this number is exceeded, we stop the algorithm. This can happen if the parameters are chosen inappropriately for this problem. In that case we might want to run this algorithm with other parameters again.

The trust-region is characterized by $\mathbf{q}_k$, $\mathbf{d}_k$ and $\delta > 0$, which is initialized in line 9. It is defined by the algorithm TR-Projection which projects inputs into the trust-region as shown here:

Algorithm 9 Projection

```
1: function TR-Projection(x, q_k, d_k)
2:     upp ← q_k + d_k
3:     low ← q_k − d_k
4:     return np.maximum(np.minimum(x, upp), low)
```

Our trust-region is the area between $\mathbf{q}_k - \delta \cdot \mathbf{d}_k$ and $\mathbf{q}_k + \delta \cdot \mathbf{d}_k$. TR-Projection projects a point x into the trust-region by checking for each element $x[i]$ individually if it lies below $\mathbf{q}_k - \delta \cdot \mathbf{d}_k$ or above $\mathbf{q}_k + \delta \cdot \mathbf{d}_k$. If the former applies, $x[i]$ is set to $\mathbf{q}_k - \delta \cdot \mathbf{d}_k$. If the latter is true, $x[i]$ is set to $\mathbf{q}_k + \delta \cdot \mathbf{d}_k$. Otherwise, $x[i]$ is not changed.

The next iterate $\mathbf{q}_k^{\text{next}}$ is now computed by calling the EnOpt algorithm on the surrogate functional F_{ML}^k. For the inputs of the EnOpt algorithm, we set here the projection pr to **lambda** mu : TR-Projection(mu, $\mathbf{q}_k, \delta \cdot \mathbf{d}_k$), so that the operations are inside the trust-region, and the initial covariance matrix $\mathbf{C}_{\text{init}}$ to $\mathbf{C}_k$, so that the samples for the first iteration are distributed like the sample set that was used for the training of the surrogate functional.

To examine the quality of the iterate $\mathbf{q}_k^{\text{next}}$ and the surrogate F_{ML}^k, ρ_k is defined in line 30. If $\rho_k < 0.25$, the surrogate is not sufficiently accurate for us and we decrease the trust-region by multiplying δ with 0.25. If the condition in line 34 is true, the surrogate is a sufficiently good approximation of F and from $\text{np.ANY}(\text{np.ABS}(\mathbf{q}_k - \mathbf{q}_k^{\text{next}}) - \mathbf{d}_k^{\text{iter}} = 0)$ follows that $\mathbf{q}_k^{\text{next}}$ is at the border of the trust-region. In this case the trust-region is extended by multiplying δ with 2. If ρ_k is greater than zero, we leave the inner while-loop since F_k^{next} is greater than F_k. The denominator in line 30, $\mathrm{F}_{\text{ML}}^k(\mathbf{q}_k^{\text{next}}) - \mathrm{F}_{\text{ML}}^k(\mathbf{q}_k)$, should be positive because $\mathbf{q}_k^{\text{next}}$ results from the EnOpt algorithm on F_{ML}^k with the initialization $\mathbf{q}_k$.

If the condition in line 21 is still not satisfied, the same procedure is repeated with another surrogate. If it is satisfied, $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k$ is updated in line 39 similar to line 5. $\tilde{F}_k$ is used again in line 10 to check if an improvement of the FOM objective functional value by more than ε_o is possible until the condition in this line is no longer satisfied.

We minimize our objective funtion j now by applying $-j$ to the Adaptive-ML-EnOpt algorithm as it is shown in algorithm 10.

Algorithm 10 ROM-EnOpt algorithm

1: **function** ROM-EnOpt($\mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, V_{\text{DNN}}, \delta_{\text{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \mathbf{q}_{\text{base}}$)
2: $N_b \leftarrow \text{LEN}(\mathbf{q}_{\text{base}})$
3: **return** AML-EnOpt($-\text{J}, \mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, V_{\text{DNN}}, \delta_{\text{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$)

Like in the FOM-EnOpt algorithm 5, there are some more inputs that ROM-EnOpt requires, but we also omit these because they are only needed for the calculation of J. We require $\mathbf{q}_{\text{base}}$ instead of N_b as an input because $\mathbf{q}_{\text{base}}$ is used for the computation of J.

5 Numerical experiments

In this chapter we are going to measure the performance of our algorithms by applying them to an example of the weak problem (2.3). For this purpose, the solution of this problem is derived first. Afterwards, we compare the analytical solution with the solution from the FOM-EnOpt and the AML-EnOpt algorithm. Additionally, we examine how the optimization algorithms behave if parameters, such as the step size, are changed or if other neural network training parameters are used.

5.1 Example of an analytical problem

The problem (2.3) is solved in [2] with the adjoint state method [9], [10]. For this method, we use the Lagrangian $\mathcal{L} : Q \times X \times X \times V \to \mathbb{R}$ with

$$\mathcal{L}(q, u, z, \tilde{z}) = J(q, u) - (\partial_t u, z)_I - (\nabla u, \nabla z)_I + (f + q, z)_I + (u_0 - u(0), \tilde{z}) \tag{5.1}$$

We want to find now a stationary point $\bar{q}$ of j such that

$$j'(\bar{q})(\delta_q) = 0 \quad \forall \delta_q \in Q. \tag{5.2}$$

$j'(\bar{q})(\delta_q)$ is here the directional derivative, which is defined as

$$j'(\bar{q})(\delta_q) = \lim_{\tau \downarrow 0} \frac{j(q + \tau\delta_q) - j(q)}{\tau}.$$

If (5.2) holds, then $\bar{q}$ satisfies the first order optimality conditions of (2.4). In this case, $\bar{q}$ is even optimal due to the linear-quadratic structure of the optimal control problem [2].

If we choose u now as $u(q)$, so that it satisfies the weak state equations (2.3), we get

$$j(q) = J(q, u(q)) = \mathcal{L}(q, u(q), z, \tilde{z}),$$

since all terms after '$J(q, u)$' in (5.1) are equal to zero.

We get now the directional derivative of j by taking the directional derivative of $\mathcal{L}$ with respect to q. We note beforehand that

$$\begin{aligned} -(\partial_t u(q), \delta_z)_I - (\nabla u(q), \nabla \delta_z)_I + (f + q, \delta_z)_I &= 0 \quad \forall q \in Q, \delta_z \in X, \\ (u_0 - u(q)(0), \delta_{\tilde{z}}) &= 0 \quad \forall q \in Q, \delta_{\tilde{z}} \in X. \end{aligned}$$

By using this, we get the derivative

$$\begin{aligned} j'(q)(\delta_q) =& \mathcal{L}'(q, u(q), z, \tilde{z})(\delta_q) \\ =& \mathcal{L}'_q(q, u(q), z, \tilde{z})(\delta_q) + \mathcal{L}'_u(q, u(q), z, \tilde{z})(\delta_u). \end{aligned} \tag{5.3}$$

The first summand in the second line denotes the directional derivative of the Lagrangian with respect to q in direction $\delta_q \in Q$. Analogously, the second summand denotes the directional derivative of the Lagrangian with respect to u in direction $\delta_u \in X$. δ_u is here uniquely defined such that it satisfies

$$\begin{aligned}(\partial_t \delta_u, \phi)_I + (\nabla \delta_u, \nabla \phi)_I &= (\delta_q, \phi)_I \quad \forall \phi \in X,\\ \delta_u(0) &= 0 \qquad\qquad \text{in } \Omega.\end{aligned}$$

Then it holds, that for any $\tau \in \mathbb{R}$

$$\begin{aligned}(\partial_t(u(q) + \tau\delta_u), \phi)_I + (\nabla(u(q) + \tau\delta_u), \nabla\phi)_I &= (f + q + \tau\delta_q, \phi)_I \quad \forall \phi \in X,\\ (u(q) + \tau\delta_u)(0) &= u_0 \qquad\qquad \text{in } \Omega.\end{aligned}$$

Since $u(q+\tau\delta_q)$ is uniquely defined to satisfy the equations above, it follows that $u(q+\tau\delta_q) = u(q) + \tau\delta_u$.

Therefore we get for every functional g which has a directional derivative in δ_u-direction:

$$\begin{aligned}g'_q(u(q))(\delta_q) &= \lim_{\tau\downarrow 0} \frac{g(u(q+\tau\delta_q)) - g(u(q))}{\tau}\\ &= \lim_{\tau\downarrow 0} \frac{g(u(q)+\tau\delta_u) - g(u(q))}{\tau}\\ &= g'_u(u(q))(\delta_u).\end{aligned}$$

Because of this, we have the summand $\mathcal{L}'_u(q, u(q), z, \tilde{z})(\delta_u)$ in equation (5.3).

We calculate now $\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u)$. For this we compute $J'_u(q, u)(\delta_u)$ first:

$$\begin{aligned}J'_u(q,u)(\delta_u) &= \lim_{\tau\downarrow 0} \frac{1}{2\tau} \int_0^T \int_\Omega (u(t,x) + \tau\delta_u(t,x) - \hat{u}(t,x))^2 - (u(t,x) - \hat{u}(t,x))^2 \,\mathrm{d}x\,\mathrm{d}t\\ &= \lim_{\tau\downarrow 0} \int_0^T \int_\Omega \delta_u(t,x)(u(t,x) - \hat{u}(t,x)) \,\mathrm{d}x\,\mathrm{d}t + \frac{\tau}{2} \int_0^T \int_\Omega (\delta_u(t,x))^2 \,\mathrm{d}x\,\mathrm{d}t\\ &= \int_0^T \int_\Omega \delta_u(t,x)(u(t,x) - \hat{u}(t,x)) \,\mathrm{d}x\,\mathrm{d}t = (\delta_u, u - \hat{u})_I\end{aligned}$$

We get with similar calculations for the other summands in the Lagrangian:

$$\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) = (\delta_u, u - \hat{u})_I - (\partial_t \delta_u, z)_I - (\nabla\delta_u, \nabla z)_I - (\delta_u(0), \tilde{z}).$$

Since we can choose $z \in X$ and $\tilde{z} \in V$ freely, they are set such that $\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) = 0$. Integration by parts gives us

$$\begin{aligned}&\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) &&= 0\\ \iff\quad &(\delta_u(T), z(T)) - (\delta_u(0), z(0)) - (\delta_u, \partial_t z)_I + (\nabla\delta_u, \nabla z)_I + (\delta_u(0), \tilde{z}) &&= (\delta_u, u - \hat{u})_I.\end{aligned}$$

This holds if we set $\tilde{z} = z(0)$ and z such that

$$\begin{aligned}-(\phi, \partial_t z)_I + (\nabla\phi, \nabla z)_I &= (\phi, u - \hat{u})_I \quad \forall \phi \in X,\\ z(T) &= 0 \qquad\qquad \text{in } \Omega.\end{aligned} \tag{5.4}$$

We call this the adjoined state equation.

With $z(q)$ defined such as in (5.4) and $\tilde{z}(q) = z(q)(0)$, it holds $\mathcal{L}'_u(q, u(q), z(q), \tilde{z}(q))(\delta_u) = 0$ and therefore the expression in (5.3) is equal to

$$\begin{aligned} j'(q)(\delta_q) &= \mathcal{L}'_q(q, u(q), z(q), \tilde{z}(q))(\delta_q) \\ &= (\alpha q, \delta_q)_I + (z(q), \delta_q)_I \\ &= (\alpha q + z(q), \delta_q). \end{aligned}$$

For the testing of our algorithms, we consider the problem (2.3) on $\Omega \times I = (0,1)^2 \times (0, 0.1)$. The following example is originally described in [2].

To define the functions in (2.3), we use the eigenfunction

$$w_a(t, x_1, x_2) := \exp(a\pi^2 t) \sin(\pi x_1) \sin(\pi x_2) \text{ for } a \in \mathbb{R}.$$

Now we set

$$\begin{aligned} f(t, x_1, x_2) &:= -\pi^4 w_a(T, x_1, x_2), \\ \hat{u}(t, x_1, x_2) &:= \frac{a^2 - 5}{2 + a} \pi^2 w_a(t, x_1, x_2) + 2\pi^2 w_a(T, x_1, x_2), \\ u_0(x_1, x_2) &:= \frac{-1}{2 + a} \pi^2 w_a(0, x_1, x_2). \end{aligned}$$

If we set the regularization parameter α in the objective functional (2.1a) as π^{-4}, we get the optimal solution $(\bar{q}, \bar{u}, \bar{z})$, where

$$\begin{aligned} \bar{q}(t, x_1, x_2) &:= -\pi^4 \left(w_a(t, x_1, x_2) - w_a(T, x_1, x_2) \right), \\ \bar{u}(t, x_1, x_2) &:= \frac{-1}{2 + a} \pi^2 w_a(t, x_1, x_2), \\ \bar{z}(t, x_1, x_2) &:= w_a(t, x_1, x_2) - w_a(T, x_1, x_2). \end{aligned} \tag{5.5}$$

It holds that $\bar{q} \in Q, \bar{u} \in X, \bar{z} \in X$. We confirm now that $(\bar{q}, \bar{u}, \bar{z})$ is a minimizer by checking if $\bar{q}$ satisfies (5.2), $\bar{u}$ (2.2) and $\bar{z}$ (5.4).

Beginning with $\bar{z}$, we have $\bar{z}(T, x_1, x_2) = 0$ trivially for all $(x_1, x_2) \in \Omega$. Integration by parts gives for all $\phi \in X$

$$(\phi, \partial_t z)_I - (\nabla \phi, \nabla z)_I + (\phi, u - \hat{u})_I = (\phi, \partial_t z + \Delta z + u - \hat{u})_I.$$

Now we compute

$$\begin{aligned} \partial_t \bar{z}(t, x_1, x_2) &= \partial_t w_a(t, x_1, x_2) = a\pi^2 w_a(t, x_1, x_2) \\ \Delta \bar{z}(t, x_1, x_2) &= 2\pi^2 (w_a(T, x_1, x_2) - w_a(t, x_1, x_2)) \\ \bar{u}(t, x_1, x_2) - \hat{u}(t, x_1, x_2) &= \pi^2 ((2 - a) w_a(t, x_1, x_2) - 2 w_a(T, x_1, x_2)). \end{aligned}$$

From $\partial_t \bar{z}(t, x_1, x_2) + \Delta \bar{z}(t, x_1, x_2) + \bar{u}(t, x_1, x_2) - \hat{u}(t, x_1, x_2) = 0$ for all $(x_1, x_2) \in \Omega, t \in I$ follows that $\bar{z}$ satisfies the adjoined state equation (5.4).

By doing the same for $\bar{u}$, we see that it meets the initial condition $\bar{u}(0, x_1, x_2) = u_0(x_1, x_2)$ for all $(x_1, x_2) \in \Omega$ of (2.2). Again, integration by parts gives us for all $\phi \in X$

$$(\partial_t \bar{u}, \phi)_I + (\nabla \bar{u}, \nabla \phi)_I - (f + \bar{q}, \phi)_I = (\partial_t \bar{u} - \Delta \bar{u} - f - \bar{q}, \phi)_I.$$

It holds

$$\begin{aligned}
\partial_t \bar{u}(t,x_1,x_2) &= \frac{-a}{2+a}\pi^4 w_a(t,x_1,x_2) \\
\Delta \bar{u}(t,x_1,x_2) &= \frac{2}{2+a}\pi^4 w_a(t,x_1,x_2) \\
\partial_t \bar{u}(t,x_1,x_2) - \Delta \bar{u}(t,x_1,x_2) &= -\pi^4 w_a(t,x_1,x_2) \\
&= f(t,x_1,x_2) + \bar{q}(t,x_1,x_2).
\end{aligned}$$

Therefore $\partial_t \bar{u}(t,x_1,x_2) - \Delta \bar{u}(t,x_1,x_2) - f(t,x_1,x_2) - \bar{q}(t,x_1,x_2) = 0$ for all $(x_1,x_2) \in \Omega, t \in I$, so $\bar{u}$ fulfills the conditions in the weak state equation (2.2).

With $\alpha = \pi^{-4}$ we get $\alpha\bar{q}(t,x_1,x_2) + \bar{z}(t,x_1,x_2) = 0$ for all $(x_1,x_2) \in \Omega, t \in I$, which means that $\bar{q}$ satisfies the optimality condition (5.2). Now we conclude that $(\bar{q},\bar{u},\bar{z})$ is the optimal solution.

Since the solutions $q \in Q_d$ of our algorithms have the form that is described in (2.8), we need to define a control-shape functional before we search for an optimizer. $\bar{q}$ can be written as

$$\bar{q}(t,x_1,x_2) = -\pi^4(\exp(a\pi^2 t) - \exp(a\pi^2 T))\sin(\pi x_1)\sin(\pi x_2).$$

Now we have $\bar{q}$ as a product of the term $-\pi^4(\exp(a\pi^2 t) - \exp(a\pi^2 T))$, that is only dependent on t, and the shape functional $\sin(\pi x_1)\sin(\pi x_2)$, which depends only on x and is an element of H. Additional shape functionals for the control functional are not used so that we can compare the output of our optimization algorithms with the analytical solution $\bar{q}$. We expect that the elements q_1^i of the control vectors $\mathbf{q}$ that we get as outputs are close to

$$\bar{q}_1^i = -\pi^4(\exp(a\pi^2 t_i) - \exp(a\pi^2 T)) \text{ for } i = 0,\ldots,N_t. \tag{5.6}$$

For our tests, we choose a to be $-\sqrt{5}$, so that f and $\hat{u}$ do not depend on time.

5.2 Numerical results

Now we test the FOM- and AML-EnOpt algorithms on the example from the last section. We discretize the space $\Omega = (0,1)^2$ as described in subsection 2.2.1 with 50 grid intervals. The time interval $I = (0,T)$ with $T = 0.1$ that we observe is divided into 10 smaller intervals which are all of the same size. This is described in subsection 2.2.2 with $N_t = 10$. Since we have only one shape functional, the control vectors $\mathbf{q}$ that we search for have $N_{\mathbf{q}} = 11$ elements.

The initial control vector $\mathbf{q}_0$ is a vector with constant entries of the value -40. This value lies approximately between the minimum element in (5.6) which is $\bar{q}_1^0 \approx -87$ and the maximum $\bar{q}_1^{N_t} = 0$. The values of the constant initial control vector, along with other parameters for the algorithms that are presented in the chapters 3 and 4, are specified in the table 5.1. The notation in the table corresponds to the notation in these algorithms.

The neural network that is used for the surrogate functional consists of two hidden layers where each hidden layer has 25 neurons. Our activation function is the tanh function as mentioned in section 4.1. Early stopping is applied for the training of the DNN with a maximum of 1000 training epochs. The variable earlyStop in algorithm 6 is set to 20. We use the L-BFGS optimizer with strong Wolfe line-search for the minimization of the MSE

loss on the validation set. The validation set consists of 20% of the sample set, so the other 80% of the sample set are used for the training set. The number of training restarts is very small because a higher number would extend the training time of the DNN by so much that it makes the algorithm terminate slower. We restart the training two times, so we train three DNNs in total to construct the surrogate functional.

Before the results are presented, we want to remember that the FOM-EnOpt and the Adaptive-ML-EnOpt procedures minimize the objective functional j by maximizing the negative functional $-j$. The results that we show are converted back to the outputs that the objective functional j would give, although the true values during these algorithms are negative. As an example, the graphs in figure 5.1 would be mirrored on the y-axis if they had shown the values during the respective procedure.

For consistency, we will refer to $\mathbf{q}_k$ as the iterate at the beginning of the outer iteration k and $\mathbf{q}_k^{\text{next}}$ as the iterate at the end of the outer iteration k. This is based on the notation of the AML-EnOpt algorithm 8. If we talk about the FOM-EnOpt algorithm, we refer to $\mathbf{q}_k$ as the iterate before the call of the optimization step procedure in line 3, respectively line 7, in the k-th outer iteration of the algorithm 1 and thus we denote $\mathbf{q}_k^{\text{next}}$ as the iterate after this call in iteration k.

Table 5.1: Parameters used in the FOM-EnOpt and AML-EnOpt algorithms

Parameter	Value
Elements of the initial constant control vector $\mathbf{q}_0$	-40
Initial step size β_1	1
Initial covariance matrix adaption step size β_2	0.1
Initial trust-region step size δ_{init}	100
Step size contraction r	0.5
Maximum step size trials ν^*	10
Maximum (outer/ inner) iterations k^*, k_o^*, k_i^*	1000
Maximum trust-region iterations k_{TR}^*	5
Initial control-type variance σ_1^2	0.1
Constant correlation factor ρ	0.9
Perturbation size N	100
FOM-EnOpt ε	10^{-8}
Tolerances Adaptive-ML-EnOpt inner iteration ε_i	10^{-12}
Adaptive-ML-EnOpt outer iteration ε_o	10^{-8}

Figure 5.1 shows the development of the FOM objective functional value $\mathrm{J}(\mathbf{q}_k^{\text{next}})$ after each outer iteration during the FOM- and AML-EnOpt procedures, as well as the respective functional value $\mathrm{J}_{\text{ML}}^k(\mathbf{q}_k^{\text{next}})$ of the surrogate functional J_{ML}^k that is used in line 28 of the AML-EnOpt algorithm 8 to compute the iterate $\mathbf{q}_k^{\text{next}}$.

We can see here that the functional values of the FOM-EnOpt and the Adaptive-ML-EnOpt algorithms converge towards a minimum. The output of the FOM-EnOpt algorithm gives an output whose objective functional value is approximately 4.22982802 after 125 iterations. The FOM objective functional value of the output from the AML-EnOpt procedure is approximately 4.22981359 which is reached after only 25 outer iterations. So the Adaptive-EnOpt algorithm gives here not only an output that has a smaller objective functional value, but also requires far fewer outer iterations than the FOM-EnOpt algorithm to

terminate.

To compare the functional values of the objective functional and the surrogate functionals of the Adaptive-ML-EnOpt procedure, we examine figure 5.2. The plot at the top shows the same graphs as in figure 5.1, except that the objective functional values of the FOM-EnOpt algorithm are not included. The plot at the bottom shows only the last five outer iterations of the plot from above. We can see here that there is quite a big difference between the objective functional values and the surrogate functional values in the first outer iterations. The difference after the first iteration is approximately 0.02. However, this difference gets smaller during the runtime of the procedure and is in the order of 10^{-7} for the last iterations.

One reason for this phenomenon is the difference between $\mathbf{q}_k$ and $\mathbf{q}_k^{\text{next}}$ at different outer iterations. In the first iterations, we are relatively far away from an optimum and therefore the iterates change considerably. The surrogate functional is trained by a training and a validation set which are sampled around the iterate $\mathbf{q}_k$, so if the difference between $\mathbf{q}_k^{\text{next}}$ and $\mathbf{q}_k$ is large, the same tends to hold for the difference between $\mathbf{q}_k^{\text{next}}$ and the samples, so the surrogate functional is less precise at $\mathbf{q}_k^{\text{next}}$. As the algorithm progresses, the iterate converges towards an optimum and the differences between successive iterates are smaller, resulting in more accurate surrogate functional values.

This can be seen in figure 5.3. In the first iteration, the step from $\mathbf{q}_k$ to $\mathbf{q}_k^{\text{next}}$ is much larger than the step from $\mathbf{q}_k$ to $\tilde{\mathbf{q}}_k$. The same holds with regard to the difference between $\mathbf{q}_k$ and the samples in T_k. Therefore the DNN is inaccurate at $\mathbf{q}_k^{\text{next}}$. If this were the FOM-EnOpt algorithm, then $\tilde{\mathbf{q}}_k$ would be the next iterate, so we can see here a reason why the AML-EnOpt algorithm requires far fewer iterations than the FOM-EnOpt procedure which is that the machine learning-based algorithm takes larger steps at the beginning.

Although the step from $\mathbf{q}_k$ to $\mathbf{q}_k^{\text{next}}$ is still larger than the step from $\mathbf{q}_k$ to $\tilde{\mathbf{q}}_k$ at the last iteration, the step sizes differ not by a lot compared to the first iteration. $\mathbf{q}_k^{\text{next}}$ is even closer to $\mathbf{q}_k$ than all the samples are since the value of $\mathbf{q}_k^{\text{next}}$ at the bottom plot in figure 5.3 is smaller than the value of T_k min which is the sample in T_k with the minimum L^2-distance to $\mathbf{q}_k$.

Figure 5.4 shows the solutions that we get from the FOM-EnOpt and the Adaptive-ML-EnOpt algorithms, as well as the analytical solution and the initialization of the iterates at the top. At the bottom, the difference between the FOM-EnOpt, respectively AML-EnOpt, solution and the analytical solution is depicted. The plot at the bottom shows us that the two solutions strategies yield similar results. At most points, the solutions of both algorithms are relatively close to the analytical solution. However, the control values for the second, last and especially first time step are far from the analytically optimal values.

A reason for this might be inaccuracies of the state variable discretizations. The objective functional is calculated according to subsection 2.2.4 as

$$\begin{aligned}\frac{T}{6N_t}\sum_{m=1}^{N_t}&\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\mathbf{M}_n\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\\&+\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\mathbf{M}_n\left(\mathbf{U}_m-\hat{\mathbf{U}}_m\right)\\&+\left(\mathbf{U}_m-\hat{\mathbf{U}}_m\right)\mathbf{M}_n\left(\mathbf{U}_m-\hat{\mathbf{U}}_m\right)\\+\frac{\alpha T}{6N_t}\sum_{m=1}^{N_t}&\mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_{m-1}+\mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_m+\mathbf{Q}_m\mathbf{M}_n\mathbf{Q}_m,\end{aligned}\tag{5.7}$$

where the notation corresponds to the notation of subsection 2.2.4.

We compute the state vectors $\mathbf{U}_m$ with the Crank-Nicolson scheme (2.6):

$$\left(\tilde{\mathbf{M}}_n^T+\frac{T}{2N_t}\tilde{L}_n^T\right)\mathbf{U}_m=\tilde{\mathbf{M}}_n^T\mathbf{U}_{m-1}-\frac{T}{2N_t}\tilde{L}_n^T\mathbf{U}_{m-1}+\frac{T}{2N_t}\mathbf{F}_{m-1}+\frac{T}{2N_t}\mathbf{F}_m,$$

for $m=1,\dots,N_t$. The control q_1^m influences only the vector $\mathbf{F}_m$ directly for $m=0,\dots,N_t$. Hence, for $m=1,\dots,N_t-1$, each control q_1^m is used for the definitions of the states $\mathbf{U}_m$ and $\mathbf{U}_{m+1}$. Only q_1^0 and $q_1^{N_t}$ set just one state which is $\mathbf{U}_1$, respectively $\mathbf{U}_{N_t}$, while they are still weighted with the same value of $\frac{T}{2N_t}$ as every other control. Therefore changes of q_1^0 and $q_1^{N_t}$ tend to effect the first part of (5.7) less than the other control variables. Hence, the control variables may reduce the regularization term in (5.7) instead by going closer to zero. This can be seen for q_1^0. The first control of the analytical solution is approximately $\bar{q}_1^0\approx -87$ which is far from zero. Therefore, with the discretization, it might be beneficial to reduce the regularization term by increasing q_1^0. Since the difference q_1^0 and $\bar{q}_1^0$ is so large, it is not surprising that q_1^1 is also not a good approximation of $\bar{q}_1^1$. In contrast to q_1^0, $q_1^{N_t}$ is further away from zero than its respective control of the analytical solution. However, due to the quadratic nature of the regularization term, this has not such a strong effect on the objective functional. To reduce these inaccuracies, we could increase the number of time steps. This is examined further below.

To investigate the effects of different neural network structures, we test now the AML-EnOpt algorithm with different quantities of neurons in the hidden layer. The number of hidden layers is fixed to two. The progression of the FOM objective functional values for different DNN structures is shown in Figure 5.6. To distinguish the different results, the bottom plot shows the objective functional values of the last outer iterations. Since the procedure with 1000 neurons in the hidden layer has here many more outer iterations than the rest, we do not show this plot to make the differences between the other results clearer. The number of outer iterations, as well as the minimum, maximum, and average training and validation losses are shown in table 5.3. The values in the table 5.4, multiplied with 10^{-5} and added to 4.2298, are the FOM objective functional values that the outputs of the respective procedures yields.

The initial shape functional control variance σ_1^2 and the constant correlation factor ρ have a strong effect on the runtime and output of the FOM-EnOpt and AML-EnOpt algorithms. If the variance is too large, the FOM-EnOpt procedure tends to give a worse optimal

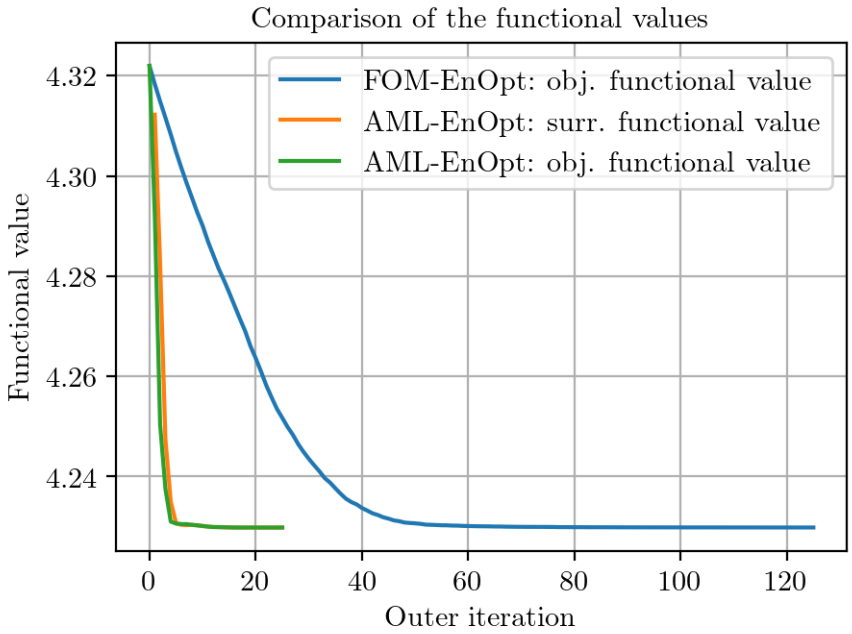

Figure 5.1: Comparison of the FOM objective functional values obtained during the outer iterations of the FOM-EnOpt and AML-EnOpt algorithms, as well as the surrogate functional values obtained during the outer iterations of the AML-EnOpt algorithm

solution. If the variance is too small, this algorithm needs more optimization steps than necessary to terminate.

We choose the correlation factor to be close to one so that we get a smoother output. However, if this value is too large, the result is similar to a large σ_1^2 which we want to avoid. For example, if we set $\rho = 0.9$, we get from (3.1) that the variance of q_1^i is approximately $5.3 \cdot \sigma_1^2$ for $i = 0, \dots, N_t$. If ρ is equal to 0.99, the variance is with a value of approximately $50.3 \cdot \sigma_1^2$ almost ten times as high.

A poor choice of these values may have even more serious consequences for the FOM-EnOpt algorithm. If we set the variance too small, the neural network-based surrogate functional is not approximating the FOM objective functional sufficiently well in a large enough area and the AML-EnOpt algorithm fails. However, we still want to set the variance so small that the surrogate is a good approximation of the objective functional in a small area around the iterate when we are close to the optimal solution.

We conclude that σ_1^2 and ρ should be chosen carefully and dependent on each other. If we increase ρ to get smoother iterates, we might have to decrease σ_1^2 so that the area which contains our samples is not too large. Also, the covariance matrix adaption step size β_2 should be large enough that the samples deviate not too much when the iterates get close to the optimum but not too large since that might result in variances close to zero when the iterate is not close to the optimum. Hence, it might require some testing to find values that suit the available optimization problem.

Unlike the Adaptive-ML-EnOpt algorithm in [1], we need to employ a trust-region method.

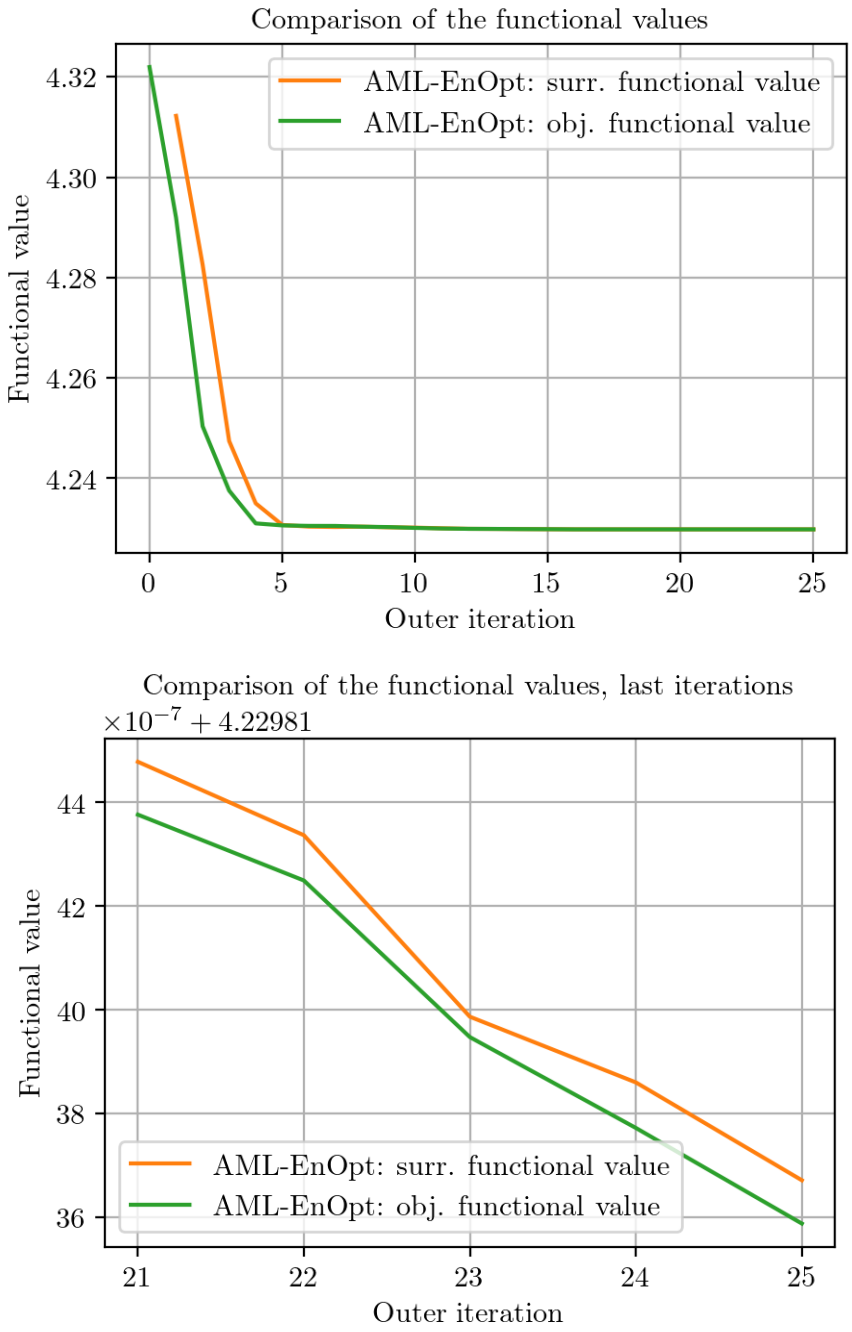

Figure 5.2: Comparison of the FOM objective functional values obtained during the outer iterations of the AML-EnOpt algorithm, as well as the respective surrogate functional values

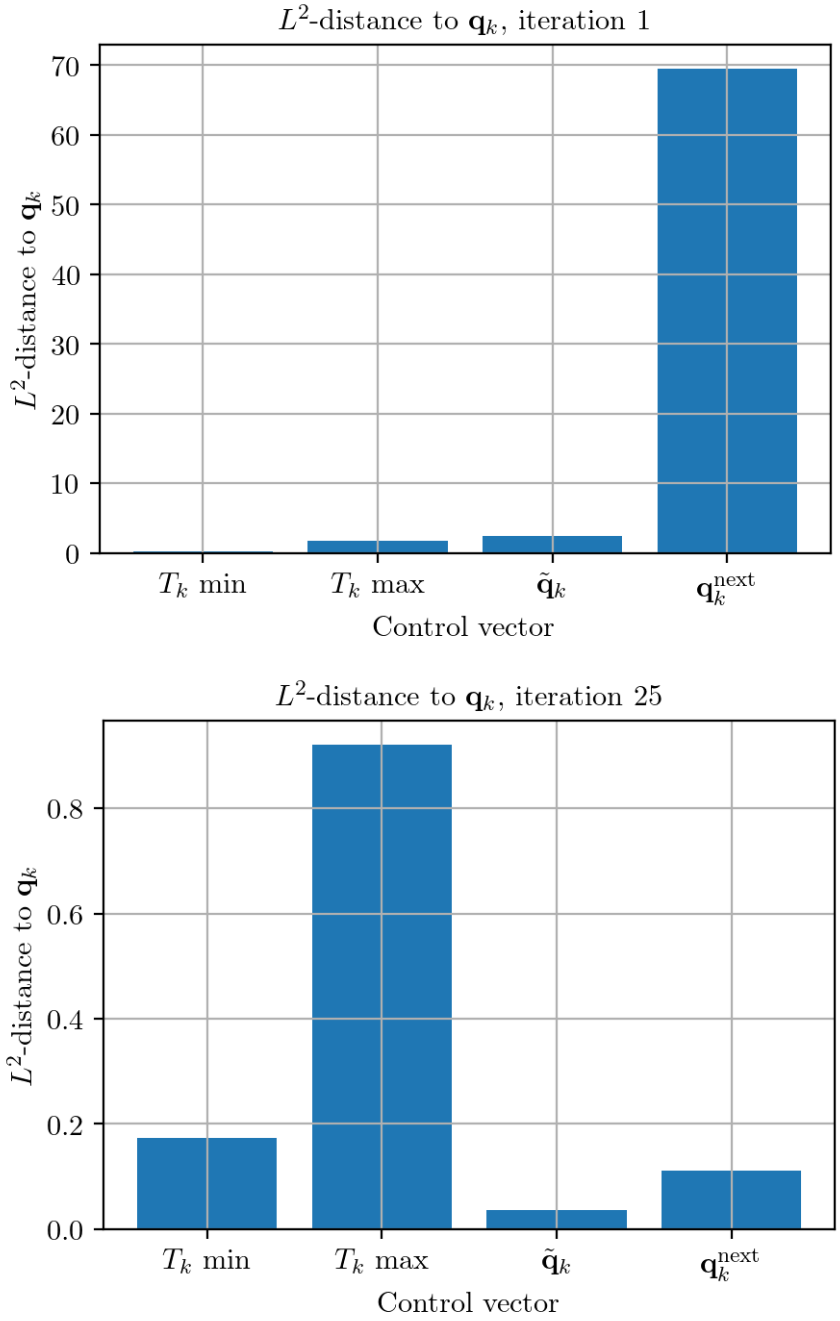

Figure 5.3: L^2-distance between $\mathbf{q}_k$ and T_k min, T_k max, $\tilde{\mathbf{q}}_k$ and $\mathbf{q}_k^{\text{next}}$ at the first (top) and last (bottom) outer iteration. $\mathbf{q}_k$ is here the iterate at the start of the respective iteration and $\mathbf{q}_k^{\text{next}}$ the iterate at the end of the iteration. T_k min is the sample in T_k with the minimum L^2-distance to $\mathbf{q}_k$ and T_k max the sample with the maximum L^2-distance to $\mathbf{q}_k$.

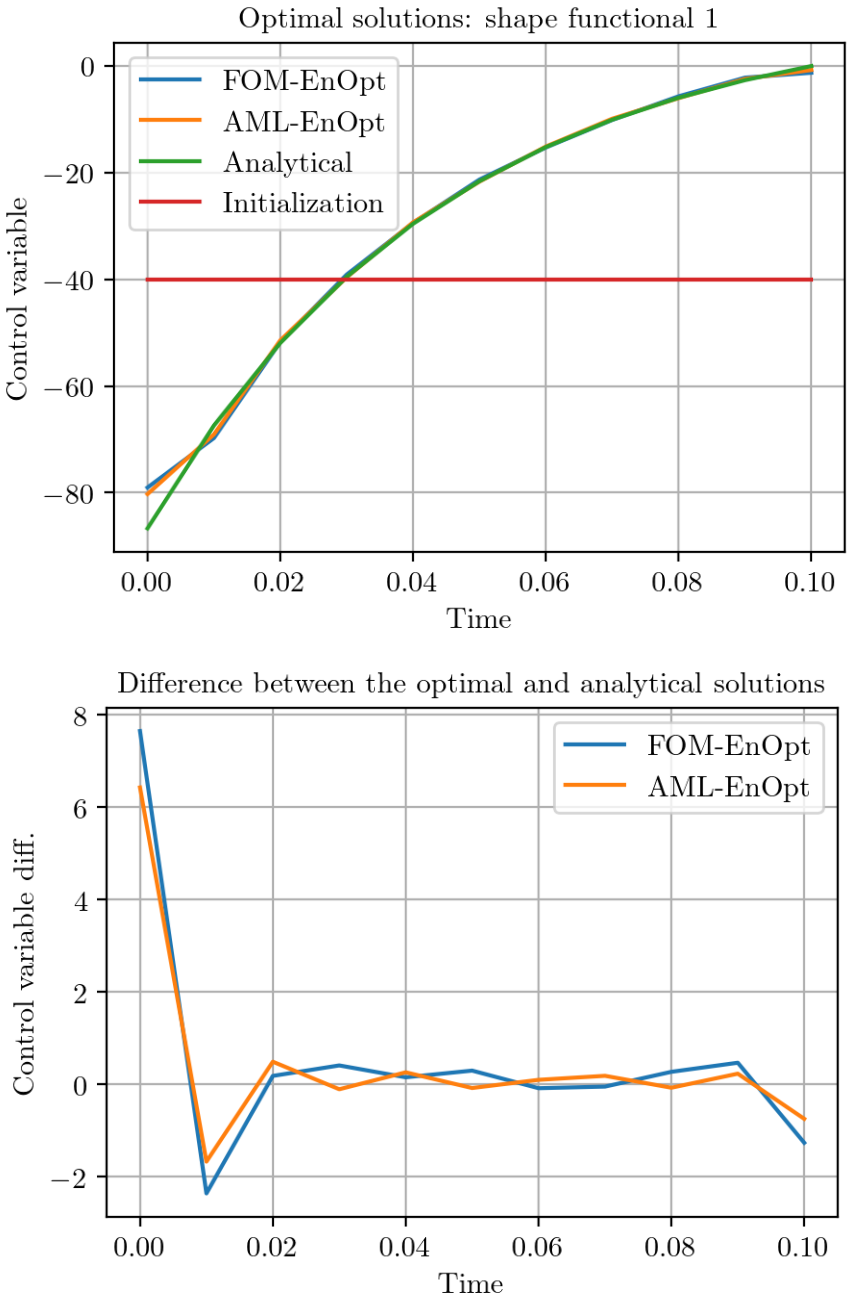

Figure 5.4: Comparison of the optimal solutions obtained from the FOM-EnOpt and the AML-EnOpt algorithms

If we proceed without it, we get the results of figure 5.7. The method seems to work up to the eighth iteration, but the machine learning-based surrogate suggests in the ninth outer iteration that there is an optimum near a point that is far from the actual optimum. This point is shown in the bottom plot of figure 5.7. Some entries of the resulting control vector exceed even the value of 600, although they should be below zero.

Before we increase the number of time steps, we compare how both algorithms get to their respective solution. During the FOM-EnOpt procedure, the iterates form a curve with a smooth shape early on. Going on, this curve changes only slightly after each iteration, mainly due to vertical stretching. With the Adaptive-ML-EnOpt method, the iterates are relatively close to the optimal point after a few iterations. Therefore, most iterations change the values of the iterates only slightly. To demonstrate this, the plots of the iterates after the fifth outer iteration are shown in figure 5.5. We see here that the curve of the FOM-EnOpt procedure has a much smoother shape than that of the Adaptive-ML-EnOpt method. However, the values of the FOM method iterate lie in a range between approximately -39 and -35 and are far away from the optimum while their counterparts take a minimum below -70 and a maximum of around 0.

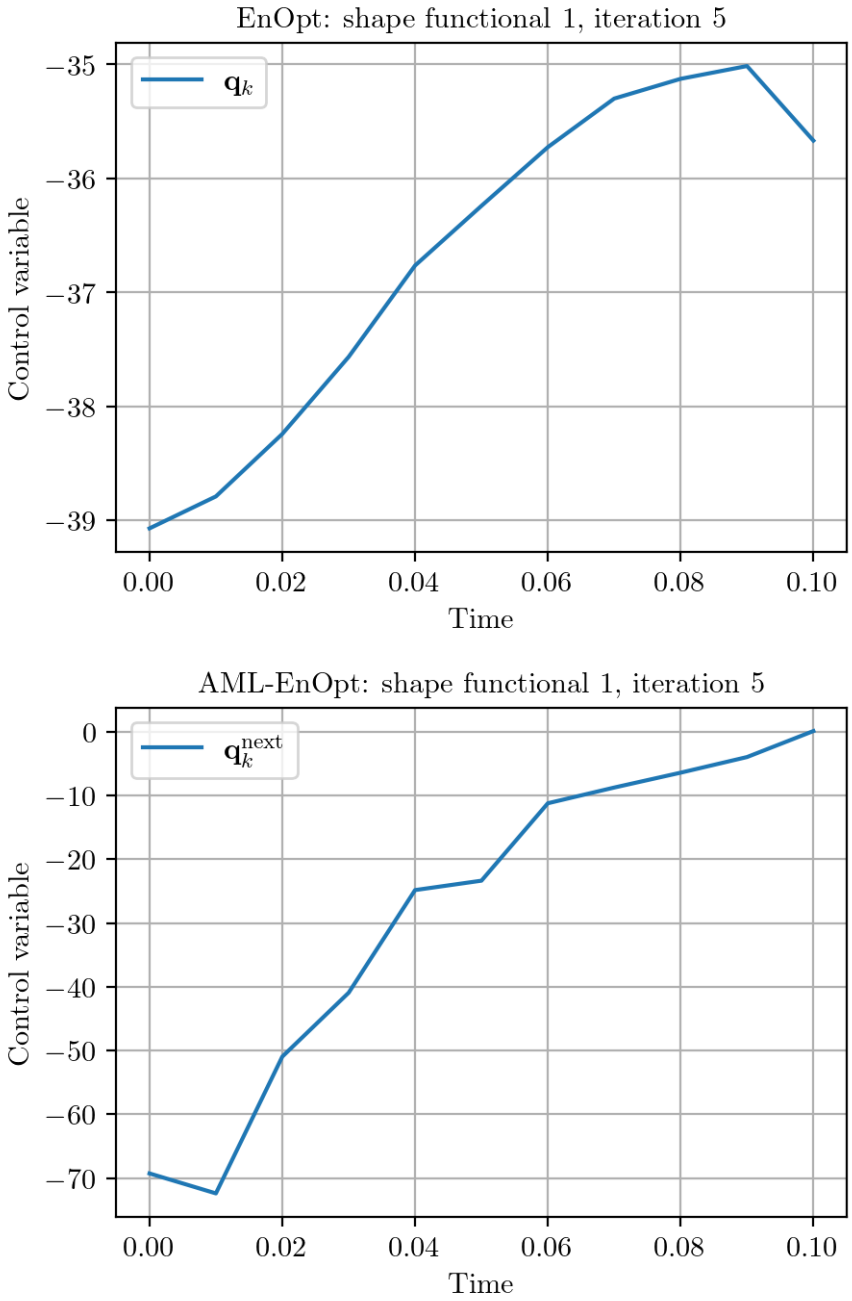

Figure 5.5: The iterates of the FOM-EnOpt (top) and the Adaptive-ML-EnOpt (bottom) algorithms after the fifth iteration

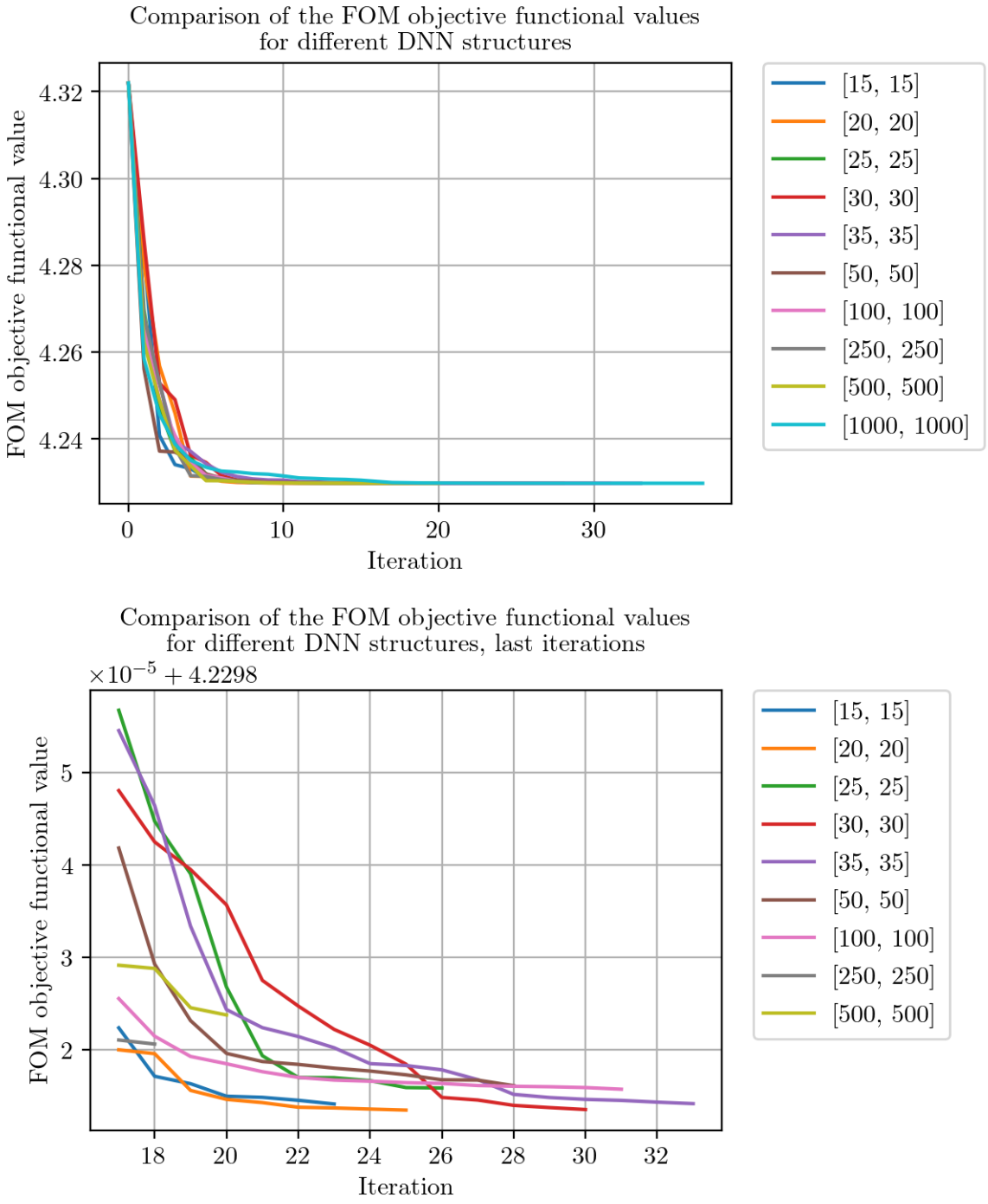

Figure 5.6: Comparison of the FOM objective functional values from the AML-EnOpt algorithm for different numbers of neurons in the hidden layers. The plot at the bottom shows the functional values of the last outer iterations without the result with 1000 neurons in the hidden layers.

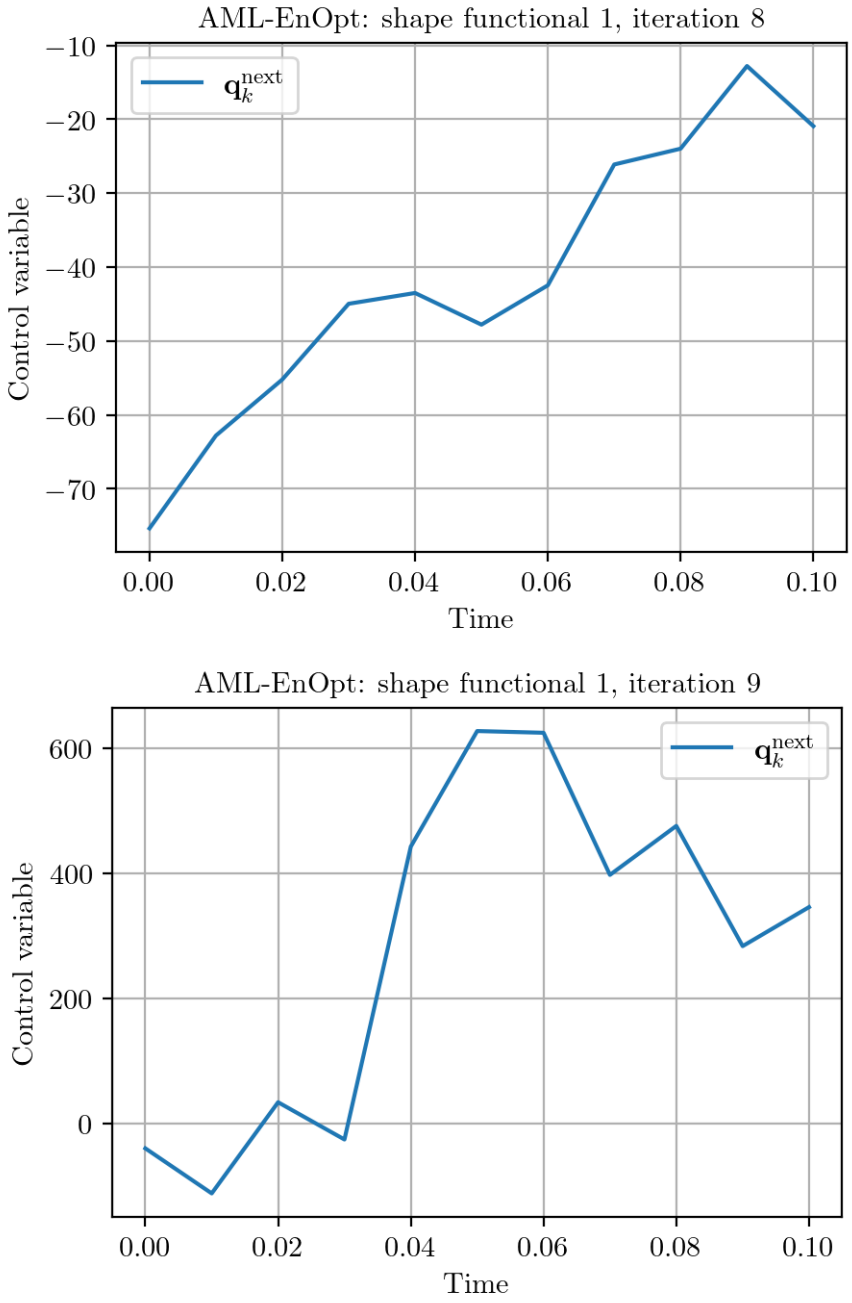

Figure 5.7: An example of iterates after the eighth and ninth outer iteration if the trust-region method is not applied

Table 5.2: Comparison of the results from the FOM-EnOpt and AML-EnOpt algorithms

Method	Iteration	FOM objective function value	Surrogate function value
FOM – EnOpt	1	4.229826194856661	-
	2	4.229829269880319	-
	3	4.2298361817288885	-
AML – EnOpt	1	4.229813297937858	4.229813485631982
	2	4.229813596812994	4.229813708587202
	3	4.229813126525124	4.22981328826163

Method	Iteration	Outer iterations	Inner iterations
FOM – EnOpt	1	119	-
	2	114	-
	3	124	-
AML – EnOpt	1	27	924
	2	27	877
	3	24	895

Method	Iteration	FOM evaluations	Surrogate evaluations
FOM – EnOpt	1	12107	-
	2	11629	-
	3	12643	-
AML – EnOpt	1	2912	95047
	2	2905	90241
	3	2589	91967

Method	Iteration	Total run time (min)	Training time (min)
FOM – EnOpt	1	37.67	-
	2	36.07	-
	3	39.22	-
AML – EnOpt	1	24.05	14.24
	2	23.87	14.17
	3	22.14	13.39

Table 5.3: Minimum, maximum, and average MSE loss on the training and validation set during the AML-EnOpt procedure with different numbers of neurons in the hidden layers of the neural network. The number of hidden layers is fixed to two.

Neurons	Outer	Training loss			Validation loss		
$N_1 = N_2$	iter.	Min.	Max.	Avg.	Min.	Max.	Avg.
15	23	$2.1 \cdot 10^{-7}$	$2.6 \cdot 10^{-4}$	$2.1 \cdot 10^{-5}$	$7.3 \cdot 10^{-7}$	$1.4 \cdot 10^{-3}$	$2.5 \cdot 10^{-4}$
20	25	$3.3 \cdot 10^{-7}$	$7.7 \cdot 10^{-4}$	$6.2 \cdot 10^{-5}$	$1.9 \cdot 10^{-6}$	$2.9 \cdot 10^{-3}$	$4.7 \cdot 10^{-4}$
25	26	$4.7 \cdot 10^{-7}$	$9.3 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$	$1.0 \cdot 10^{-6}$	$4.2 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$
30	30	$3.0 \cdot 10^{-7}$	$1.1 \cdot 10^{-4}$	$9.4 \cdot 10^{-6}$	$3.9 \cdot 10^{-7}$	$1.1 \cdot 10^{-3}$	$2.1 \cdot 10^{-4}$
35	33	$2.5 \cdot 10^{-7}$	$5.7 \cdot 10^{-4}$	$2.2 \cdot 10^{-5}$	$9.3 \cdot 10^{-7}$	$1.6 \cdot 10^{-3}$	$2.0 \cdot 10^{-4}$
50	28	$3.9 \cdot 10^{-7}$	$3.3 \cdot 10^{-4}$	$2.3 \cdot 10^{-5}$	$5.2 \cdot 10^{-7}$	$3.6 \cdot 10^{-4}$	$9.3 \cdot 10^{-5}$
100	31	$5.7 \cdot 10^{-7}$	$6.5 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$
250	18	$2.9 \cdot 10^{-7}$	$7.2 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$8.4 \cdot 10^{-7}$	$5.2 \cdot 10^{-4}$	$8.6 \cdot 10^{-5}$
500	20	$5.3 \cdot 10^{-7}$	$1.3 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$6.8 \cdot 10^{-7}$	$4.1 \cdot 10^{-4}$	$8.9 \cdot 10^{-5}$
1000	37	$4.0 \cdot 10^{-7}$	$9.0 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$	$1.4 \cdot 10^{-6}$	$4.2 \cdot 10^{-4}$	$9.0 \cdot 10^{-5}$

Table 5.4: FOM objective functional output values of the AML-EnOpt procedure with different numbers of neurons in the hidden layers of the neural network. The number of hidden layers is fixed to two.

Neurons ($N_1 = N_2$)	15	20	25	30	35	50	100	250	500	1000
FOM obj. func. val. ($\cdot 10^{-5} + 4.2298$)	1.41	1.35	1.59	1.35	1.42	1.61	1.57	2.06	2.38	1.63

Bibliography

[1] T. Keil, H. Kleikamp, R. J. Lorentzen, M. B. Oguntola, and M. Ohlberger, "Adaptive machine learning-based surrogate modeling to accelerate PDE-constrained optimization in enhanced oil recovery," *Advances in Computational Mathematics*, vol. 48, no. 6, p. 73, Nov. 2022.

[2] D. Meidner and B. Vexler, "A priori error estimates for space-time finite element discretization of parabolic optimal control problems part i: Problems without control constraints," *SIAM Journal on Control and Optimization*, vol. 47, no. 3, pp. 1150–1177, 2008. DOI: 10.1137/070694016. eprint: https://doi.org/10.1137/070694016. [Online]. Available: https://doi.org/10.1137/070694016.

[3] M. B. Oguntola and R. J. Lorentzen, "Ensemble-based constrained optimization using an exterior penalty method," *Journal of Petroleum Science and Engineering*, vol. 207, p. 109 165, 2021, ISSN: 0920-4105. DOI: https://doi.org/10.1016/j.petrol.2021.109165. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920410521008184.

[4] Y. Zhang, A. S. Stordal, and R. J. Lorentzen, "A natural hessian approximation for ensemble based optimization," en, *Comput. Geosci.*, vol. 27, no. 2, pp. 355–364, Apr. 2023.

[5] A. S. Stordal, S. P. Szklarz, and O. Leeuwenburgh, "A theoretical look at Ensemble-Based optimization in reservoir management," *Mathematical Geosciences*, vol. 48, no. 4, pp. 399–417, May 2016.

[6] L. Prechelt, "Early stopping — but when?" In *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 53–67, ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_5. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_5.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[8] J. Nocedal and S. Wright, *Numerical Optimization* (Springer Series in Operations Research and Financial Engineering), en, 2nd ed. New York, NY: Springer, Jul. 2006.

[9] R.-É. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, pp. 495–503, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:123458541.

[10] R. Becker, D. Meidner, and B. Vexler, "Efficient numerical solution of parabolic optimization problems by finite element methods," *Optimization Methods and Software*, vol. 22, Oct. 2007. DOI: 10.1080/10556780701228532.