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

[1]

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

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

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

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

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 $\mathcal{V} = (x_1, \dots, x_N) \in (\mathbb{R}^2)^N$ with a convex hull that is equal to $\bar{\Omega}$ and $x_i \neq x_j$ for all $i \neq j$ in $\{1, \dots, N\}$. 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 our implementation, we discretize a rectangular domain by specifying the number of grid intervals first. Then, we divide the domain 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 divided 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 we have given a domain $\Omega = [a, a]$ with $a > 0$ and we define the number of grid intervals as 2, then our mesh would look like that:

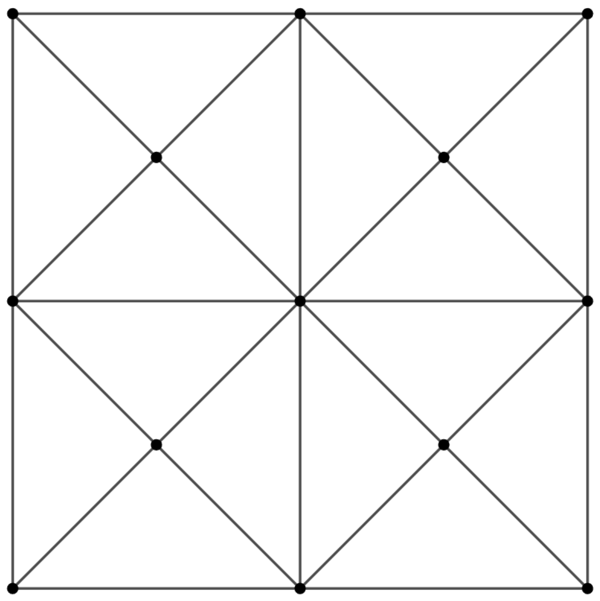

Figure 2.1: Example of a mesh with 2 grid intervals in a square shaped domain.

Now, let $\mathcal{P}_1(\hat{T}, \mathbb{R})$ be the space of polynomials up to order 1 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

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

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 $i = 1, \dots, N$.

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

$$u = \sum_{i=1}^{N} U_i \phi_i$$

with $U_i = u(x_i)$.

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

$L_n = \left(\int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}x \right)_{i,j=1,\ldots,N}$. Let

$$U = \begin{pmatrix} U_1 \\ \vdots \\ U_n \end{pmatrix} \text{ and } V = \begin{pmatrix} V_1 \\ \vdots \\ V_n \end{pmatrix}.$$

Then we have

$$\int_\Omega u \cdot v \, \mathrm{d}x = U^T M_n V \text{ and } \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x = U^T L_n V.$$

2.2.2 Discretization in time

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

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

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

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

where $\mathcal{P}_1(I_m, V_h)$ denotes the space of polynomials up to order 1, defined on I_m with values in V_h. Similarly, we define the time-discretized space of our control variables as

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

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

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

where $v_m(\cdot) = v(t_m, \cdot)$.

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. For $m = 0$, we set

$$U_0 = \begin{pmatrix} U_{0,1} \\ \vdots \\ U_{0,n} \end{pmatrix},$$

where $U_{0,i} = u_0(x_i)$ for $i = 1, \ldots, N$.

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

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

where u_m is a time discretization of u at the time step t_m, while $f_m = f(t_m, \cdot)$ and $q_m = q(t_m, \cdot)$. To solve the above equation, we define the matrix $\tilde{M}_n \in \mathbb{R}^{N\times N}$ as

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

and the matrix $\tilde{L}_n \in \mathbb{R}^{N\times N}$ as

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

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

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

In the pyMOR implementation, vectors F_m for $m = 0, \ldots, M$ are defined such that $V^T F_m \approx (f_m + q_m, v)$ for all $v \in V_h$ and $(F_m)_i = 0$ 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

$$\left(\tilde{M}_n^T + \frac{T}{2M}\tilde{L}_n^T\right) U_m = \tilde{M}_n^T U_{m-1} - \frac{T}{2M}\tilde{L}_n^T U_{m-1} + \frac{T}{2M} F_{m-1} + \frac{T}{2M} F_m, \tag{2.5}$$

which is solved after U_m with functions 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 (2.5). We calculate $j(q)$ now in the following way:

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

Integration by substitution yields

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

where $\hat{U}_m = (\hat{u}(t_m, x_i))_{i=1,\dots,N}$ and $Q_m = (q(t_m, x_i))_{i=1,\dots,N}$ for $m = 0, \dots, M$.

2.3 Optimization of the control variable

To optimize the control variable, we write every $q \in Q_d$, by using a fixed basis

$$\Phi = \{\phi_1, \dots, \phi_{N_b}\} \tag{2.6}$$

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

$$q(t,x) = \sum_{i=1}^{N_b} \alpha_i(t)\phi_i(x) \tag{2.7}$$

with

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

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

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

with $N_q = (M+1)\cdot N_b$. Therefore, we write

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

for each q that is defined like in (2.7).

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 algorithm (EnOpt) 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

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

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

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

$$\mathbf{C}^0_{\mathbf{q}_0} = \begin{pmatrix} \mathbf{C}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{C}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{C}_{N_b} \end{pmatrix}. \tag{3.1}$$

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 mean $\mathbf{q}_k$ and covariance $\mathbf{C}^k_{\mathbf{q}_k}$, and then we define

$$\mathbf{C}^k_{\mathbf{q}_k,F} := \frac{1}{N-1}\sum_{m=1}^{N} (\mathbf{q}_{k,m} - \mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)). \tag{3.2}$$

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

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

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 $\dfrac{1}{N-1}$:

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

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

$$\mathbf{C}^k_{\mathbf{q}_k} = \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}} + \tilde{\beta}_k \tilde{\mathbf{d}}_k \text{ with} \tag{3.3}$$

$$\tilde{\mathbf{d}}_k = N^{-1}\sum_{m=1}^{N}(F(\mathbf{q}_{k-1,m}) - F(\mathbf{q}_k))((\mathbf{q}_{k-1,m}-\mathbf{q}_k)(\mathbf{q}_{k-1,m}-\mathbf{q}_k)^T - \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}), \tag{3.4}$$

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 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 not the case.

The implementation in pseudo code is shown here:

Algorithm 1 EnOpt algorithm

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

We begin by initializing F_k^{prev} as $\mathrm{F}(\mathbf{q}_0)$. 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

```
1: function OptStep($F, \mathbf{q}_k, N, k, T_k, \mathbf{C}_k, F_k, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b, \textsc{pr} \leftarrow$ lambda mu : mu)
2:     $N_{\mathbf{q}} \leftarrow \textsc{len}(\mathbf{q}_k)$
3:     $\mathbf{C}_k^{\text{next}} \leftarrow \text{np.}\textsc{zeros}((N_{\mathbf{q}}, N_{\mathbf{q}}))$
4:     if $k = 0$ then
5:         if $\mathbf{C}_k$ is None then
6:             $\mathbf{C}_k^{\text{next}} \leftarrow \textsc{initCov}(\textsc{len}(\mathbf{q}_k), \sigma^2, \rho, N_t, N_b)$
7:         else
8:             $\mathbf{C}_k^{\text{next}} \leftarrow \mathbf{C}_k.\textsc{copy}(\,)$
9:     else
10:        $\mathbf{C}_k^{\text{next}} \leftarrow \textsc{updateCov}(\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2)$
11:    sample $\leftarrow$ np.random.$\textsc{multivariate\_normal}(\mathbf{q}_k, \mathbf{C}_k^{\text{next}}, \text{size} \leftarrow N)$
12:    $T_k^{\text{next}} \leftarrow [\textsc{pr}(\text{sample}[j]), \text{F}(\textsc{pr}(\text{sample}[j]))]_{j=0}^{N-1}$
13:    $\mathbf{C}_F^k \leftarrow \text{np.}\textsc{zeros}(N_{\mathbf{q}})$
14:    for $m = 0, \ldots, N-1$ do
15:        $\mathbf{C}_F^k \leftarrow \mathbf{C}_F^k + (T_k^{\text{next}}[m][0] - \mathbf{q}_k) \cdot (T_k^{\text{next}}[m][1] - F_k)$
16:    $\mathbf{C}_F^k \leftarrow 1/(N-1) \cdot \mathbf{C}_F^k$
17:    $\mathbf{d}_k \leftarrow \mathbf{C}_F^k / \|\mathbf{C}_F^k\|_\infty$
18:    $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}} \leftarrow \textsc{LineSearch}(\text{F}, \mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*, \textsc{pr})$
19:    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 (3.1).

If k is not zero, we get the updated covariance matrix by calling updateCov, which is described in (3.3) 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 $\textsc{pr}(\text{sample}[j])$ and their respective function values $\text{F}(\textsc{pr}(\text{sample}[j]))$ for $j = 0, \ldots, N-1$ are stored in the list T_k^{next}. The definition is here abbreviated as $[\textsc{pr}(\text{sample}[j]), \text{F}(\textsc{pr}(\text{sample}[j]))]_{j=0}^{N-1}$. 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 (3.2), and 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 $\text{np.}\textsc{max}(\text{np.}\textsc{abs}(\mathbf{C}_F^k))$.

The next iterate $\mathbf{q}_k^{\text{next}}$ and its function 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

```
1:  function UPDATECOV($\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2$)
2:      $N_\mathbf{q} \leftarrow$ LEN($\mathbf{q}_k$)
3:      $N \leftarrow$ LEN($T_k$)
4:      $\mathbf{d}_k^{\text{cov}} \leftarrow$ NP.ZEROS(($N_\mathbf{q}, N_\mathbf{q}$))
5:      for $m = 0, \dots, N-1$ do
6:          $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}} + (T_k[m][1] - F_k) \cdot ((T_k[m][0] - \mathbf{q}_k)$.RESHAPE(($N_\mathbf{q}, 1$)) $\cdot (T_k[m][0] -$
    $\mathbf{q}_k)$.RESHAPE(($1, N_\mathbf{q}$)) $- \mathbf{C}_k$)
7:      $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}}/N$
8:      $\mathbf{C}_{\text{diag}}, \mathbf{d}_{\text{diag}} \leftarrow$ np.ZEROS($N_\mathbf{q}$), np.ZEROS($N_\mathbf{q}$)
9:      for $i = 0, \dots, N_\mathbf{q} - 1$ do
10:         $\mathbf{C}_{\text{diag}}[i] \leftarrow \mathbf{C}_k[i,i]$
11:         $\mathbf{d}_{\text{diag}}[i] \leftarrow \mathbf{d}_k^{\text{cov}}[i,i]$
12:     $\beta_2^{\text{iter}} \leftarrow \beta_2$
13:     while np.MIN($\mathbf{C}_{\text{diag}} + \beta_2^{\text{iter}} \cdot \mathbf{d}_{\text{diag}}$) $\leq 0$ do
14:         $\beta_2^{\text{iter}} \leftarrow \beta_2^{\text{iter}}/2$
15:     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 3.4 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 (3.3).

Algorithm 4 Line search

```
1:  function LINESEARCH(F, $\mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*$, PR)
2:      $\beta_1^{\text{iter}} \leftarrow \beta_1$
3:      $\mathbf{q}_k^{\text{next}} \leftarrow$ PR($\mathbf{q}_k + \beta_1^{\text{iter}}\mathbf{d}_k$)
4:      $F_k^{\text{next}} \leftarrow$ F($\mathbf{q}_k^{\text{next}}$)
5:      $\nu \leftarrow 0$
6:      while $F_k^{\text{next}} - F_k \leq \varepsilon$ and $\nu < \nu^*$ do
7:          $\beta_1^{\text{iter}} \leftarrow r\beta_1^{\text{iter}}$
8:          $\mathbf{q}_k^{\text{next}} \leftarrow$ PR($\mathbf{q}_k + \beta_1^{\text{iter}}\mathbf{d}_k$)
9:          $F_k^{\text{next}} \leftarrow$ F($\mathbf{q}_k^{\text{next}}$)
10:         $\nu \leftarrow \nu + 1$
        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 $\mathbf{q}_k^{\text{next}} =$ PR$(\mathbf{q}_k + \beta_1^{\text{iter}}\mathbf{d}_k)$ and $F_k^{\text{next}} = \mathrm{F}(\mathbf{q}_k^{\text{next}})$ 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 $-j$ 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($-\text{J}, \mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \text{LEN}(\mathbf{q}_{\text{base}})$)
```

$\mathbf{q}_{\text{base}}$ is here a list that consists of the basis functions, so Φ in (2.6). 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 function, 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. $W_i \in \mathbb{R}^{N_i \times N_{i-1}}$ 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 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 are saved as y_{train}. So if loss_fn $= \mathscr{L}$, we have $\text{testDNN}(\text{DNN}, x_{\text{val}}, y_{\text{val}}, \text{loss\_fn}) = \mathscr{L}(\text{DNN}, T_{\text{val}})$ 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, $x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
 2:     wait ← 0
 3:     minimalValidationLoss ← TESTDNN(DNN, $x_{\text{val}}, y_{\text{val}}$, 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, $x_{\text{val}}, y_{\text{val}}$, 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 $i \in \{1, \dots, L\}$. The biases b_i are set to zero for $i \in \{1, \dots, L\}$. 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

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

be a sample set of size N and

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

the sets that contain the inputs/ output 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

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

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

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

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

```
function CONSTRUCTDNN(sample, $V_{\text{DNN}}$, minIn, maxIn)
    normSample ← np.ZEROS(LEN(sample), LEN(sample[0][0]))
    normVal ← np.ZEROS(LEN(sample))
    for $i = 0, \ldots,$ LEN(sample) $- 1$ do
        normSample[$i$, :] ← sample[$i$][0]
        normVal[$i$] ← sample[$i$][1]
    minOut ← np.MIN(normVal)
    maxOut ← np.MAX(normVal)
    normSample ← (normSample − minIn)/(maxIn − minIn)
    normVal ← (normVal − minOut)/(maxOut − minOut)
    $x$ ← torch.FROM_NUMPY(normSample).TO(torch.float32)
    $y$ ← torch.FROM_NUMPY(normVal).TO(torch.float32)
    trainSplit ← INT(trainFrac ∗ LEN($x$))
    $x_{\text{train}}, y_{\text{train}}$ ← $x$[: trainSplit], $y$[: trainSplit]
    $x_{\text{val}}, y_{\text{val}}$ ← $x$[trainSplit :], $y$[trainSplit :]
    DNN ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
    LOSS_FN ← torch.nn.MSELOSS( )
    optimizer ← torch.optim.LBFGS(DNN.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
    TRAINDNN(DNN, $x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
    evalDNN ← TESTDNN(DNN, $x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
    for $i = 1, \ldots,$ restarts do
        DNN$_i$ ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
        optimizer ← torch.optim.LBFGS(DNN$_i$.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
        TRAINDNN(DNN$_i$, $x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
        evalDNN$_i$ ← TESTDNN(DNN$_i$, $x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
        if evalDNN$_i$ < evalDNN then
            evalDNN ← evalDNN$_i$
            DNN ← DNN$_i$
    function F$_{\text{ML}}$($x_{\text{inp}}$)
        scaledInput ← torch.FROM_NUMPY(($x_{\text{inp}}$ − minIn)/(maxIn − minIn)).TO(torch.float32)
        scaledOutput = DNN(scaledInput)
        return scaledOutput.NUMPY( )[0] · (maxOut − minOut) + minOut
    return F$_{\text{ML}}$
```

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 $[N_0, N_1, \dots, N_L]$, we would get a DNN with L layers and N_i neurons in layer $i = 0, 1, \dots, L$.

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 function by a surrogate that is a neural network. The surrogate is supposed to be a local approximation of the FOM function around the current iterate. Doing that globally with a sufficient precision would be computationally too expensive.

Since the surrogate is only locally supposed to be a good approximation, we will introduce a trust region method. For that, we use an algorithm that projects inputs into the trust region.

Algorithm 8 Projection

```
1: function TR-PROJECTION($x, \mathbf{q}_k, \mathbf{d}_k$)
2:     upp ← $\mathbf{q}_k + \mathbf{d}_k$
3:     low ← $\mathbf{q}_k - \mathbf{d}_k$
4:     return NP.MAXIMUM(NP.MINIMUM($x$, upp), low)
```

We describe now the Adaptive-ML-EnOpt algorithm. This algorithm takes a function 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 9 Adaptive-ML-EnOpt algorithm

```
1: function AML-EnOpt($\mathrm{F}, \mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, V_{\mathrm{DNN}}, \delta_{\mathrm{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$)
2:     $N_\mathbf{q} \leftarrow$ LEN($\mathbf{q}_0$)
3:     $F_k \leftarrow \mathrm{F}(\mathbf{q}_0)$
4:     $F_k^{\mathrm{next}} \leftarrow F_k$
5:     $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow$ OptStep($\mathrm{F}, \mathbf{q}_0, N, 0, [\,], \mathrm{None}, F_k, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_n$)
6:     $k \leftarrow 1$
7:     $\mathbf{q}_k \leftarrow \mathbf{q}_0$
8:     $\mathbf{q}_k^{\mathrm{next}} \leftarrow \mathbf{q}_k$.COPY( )
9:     $\delta \leftarrow \delta_{\mathrm{init}}$
10:    while $\tilde{F}_k > F_k + \varepsilon_o$ and $k < k_o^*$ do
11:        $T_k^x \leftarrow$ np.ZEROS(($N, N_\mathbf{q}$))
12:        for $i = 0, \dots, N-1$ do
13:            $T_k^x[i, :] \leftarrow T_k[i][0]$
14:        minIn $\leftarrow$ np.ZEROS($N_\mathbf{q}$)
15:        maxIn $\leftarrow$ np.ZEROS($N_\mathbf{q}$)
16:        for $i = 0, \dots, N_\mathbf{q}-1$ do
17:            minIn[$i$] $\leftarrow$ np.MIN($T_k^x[:, i]$)
18:            maxIn[$i$] $\leftarrow$ np.MAX($T_k^x[:, i]$)
19:        $\mathbf{d}_k \leftarrow |\mathbf{q}_k - \tilde{\mathbf{q}}_k|$
20:        while $F_k^{\mathrm{next}} \leq F_k + \varepsilon_o$ do
21:            $F_{\mathrm{ML}}^k \leftarrow$ constructDNN($T_k, V_{\mathrm{DNN}}$, minIn, maxIn)
22:            $F_k^{\mathrm{approx}} \leftarrow \mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k)$
23:            flag$_{\mathrm{TR}}$ $\leftarrow$ True
24:            while flag$_{\mathrm{TR}}$ do
25:                $\mathbf{d}_k^{\mathrm{iter}} \leftarrow \delta \cdot \mathbf{d}_k$
26:                $\mathbf{q}_k^{\mathrm{next}} \leftarrow$ EnOpt($\mathrm{F}_{\mathrm{ML}}^k, \mathbf{q}_k, N, \varepsilon_i, k_i^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$, PR $\leftarrow$
    lambda mu : TR-Projection(mu, $\mathbf{q}_k, \mathbf{d}_k^{\mathrm{iter}}$), $\mathbf{C}_{\mathrm{init}} \leftarrow \mathbf{C}_k$)[0]
27:                $F_k^{\mathrm{next}} \leftarrow \mathrm{F}(\mathbf{q}_k^{\mathrm{next}})$
28:                $\rho_k \leftarrow \dfrac{F_k^{\mathrm{next}} - F_k}{\mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k^{\mathrm{next}}) - F_k^{\mathrm{approx}}}$
29:                if $\rho_k < 0.25$ then
30:                    $\delta \leftarrow 0.25 \cdot \delta$
31:                else
32:                    if $\rho_k > 0.75$ and np.ANY(np.ABS($\mathbf{q}_k - \mathbf{q}_k^{\mathrm{next}}$) $- \mathbf{d}_k^{\mathrm{iter}} = 0$) then
33:                        $\delta \leftarrow 2 \cdot \delta$
34:                if $\rho_k > 0$ then
35:                    flag$_{\mathrm{TR}}$ $\leftarrow$ False
36:        $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow$ OptStep($\mathrm{F}, \mathbf{q}_k^{\mathrm{next}}, N, k, T_k, \mathbf{C}_k, F_k^{\mathrm{next}}, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_b$)
37:        $F_k \leftarrow F_k^{\mathrm{next}}$
38:        $\mathbf{q}_k \leftarrow \mathbf{q}_k^{\mathrm{next}}$.COPY( )
39:        $k \leftarrow k+1$
40:    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 function 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 function F. Instead of repeating the optimization step algorithm on the FOM function and using $\tilde{\mathbf{q}}_k$ as our iterate, we initialize the iterate $\mathbf{q}_k$ 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 function value can be improved by more than ε_o. The while-loop is entered 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 $\mathbf{q}_k$ 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 is is possible to increase the FOM function 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 function F_{ML}^k is introduced by calling constructDNN in line 21. Notice 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 36), 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 function 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 ....

Here we repeat a while-loop until a certain condition is met. 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 8, TR-Projection. 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 function F_{ML}^k. We also set the projection pr to 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.

To examine the quality of the iterate $\mathbf{q}_k^{\text{next}}$ and the surrogate F_{ML}^k, ρ_k is defined in line 28. 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 32 is true, the surrogate is a sufficiently good approximation of F and from np.any(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 28, $\mathrm{F}_{\text{ML}}^k(\mathbf{q}_k^{\text{next}}) - \mathrm{F}_{\text{ML}}^k(\mathbf{q}_k)$, 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 20 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 36 similar to line 5. $\tilde{F}_k$ is used again in line 10 to check if an improvement of the FOM function 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:     $N_b \leftarrow$ LEN($\mathbf{q}_{\text{base}}$)
3:     return AML-EnOpt($-$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

Here we consider the problem (2.3) on $\Omega \times I = (0,1)^2 \times (0, 0.1)$ with homogeneous Dirichlet boundary conditions. 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 $(\bar{q}, \bar{u})$, where

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

5.2 Numerical results

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	1
Initial trust region step size δ_{init}	50
Step size contraction r	0.5
Maximum step size trials ν^*	10
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^{-8}
Adaptive-ML-EnOpt outer iteration ε_o	10^{-8}

Method	Iteration	FOM value	Surrogate value
FOM – EnOpt	1		-
	2		-
	3		-
AML – EnOpt	1		
	2		
	3		

Method	Iteration	Outer iterations	Inner iterations
FOM – EnOpt	1		-
	2		-
	3		-
AML – EnOpt	1		
	2		
	3		

Method	Iteration	FOM evaluations	Surrogate evaluations
FOM – EnOpt	1		-
	2		-
	3		-
AML – EnOpt	1		
	2		
	3		

Method	Iteration	Training time (min)	Total run time (min)
FOM – EnOpt	1		-
	2		-
	3		-
AML – EnOpt	1		
	2		
	3		

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`.