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

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

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

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

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 $\mathcal{V} = (x_1, \ldots, x_N) \in (\mathbb{R}^2)^N$ with a convex hull that is equal to $\bar{\Omega}$ and $x_i \neq x_j$ for all $i \neq j$ in $\{1, \ldots, N\}$. 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:

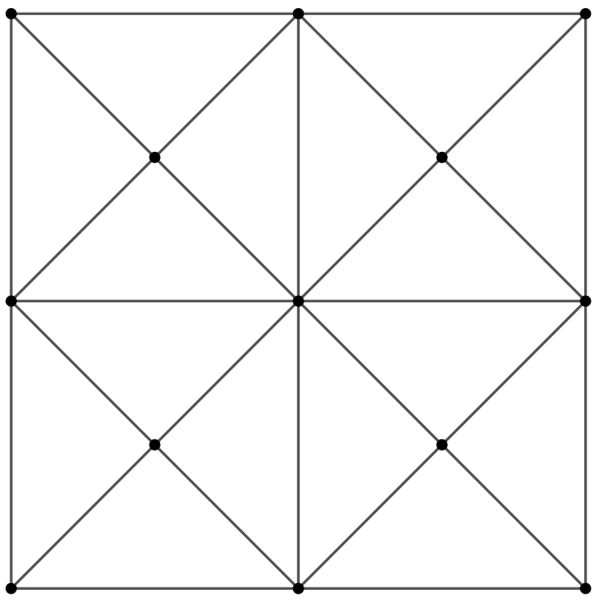

Figure 2.1: Example of a mesh with 2 grid intervals in a square shaped domain.

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 $\int_\Omega u \cdot v \, \mathrm{d}x$ 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 $M_n = \left(\int_\Omega \phi_i \cdot \phi_j \, \mathrm{d}x\right)_{i,j=1,\dots,N}$ and the stiffness matrix

$L_n = \left(\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}x\right)_{i,j=1,\dots,N}$. 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 $m = 0, \dots, M$ 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

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

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

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

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 $i = 1, \dots, N$.

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

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

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 $m = 0, \dots, M$, 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 $m = 0, \dots, M$ 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

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

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

$$q(t,x) = \sum_{i=1}^{N_b} \alpha_i(t)\phi_i(x) \tag{2.6}$$

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 (2.6).

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 $F : \mathbb{R}^{N_q} \to \mathbb{R}$ 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

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

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

where $k = 0, 1, 2, \dots$ 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

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

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 $\mathbf{C}_j := \left(\mathrm{Cov}(q^i_j, q^k_j)\right)_{i,k}$ with $j = 1, \dots, N_b$, 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 $m = 1, \dots, N$, 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

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

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

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

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

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 $F : \mathbb{R}^{N_q} \to \mathbb{R}$, our initial iterate $\mathbf{q}_0 \in \mathbb{R}^{N_q}$, the sample size $N \in \mathbb{N}$, the tolerance $\varepsilon > 0$, the maximum number of iterations $k^* \in \mathbb{N}$, the initial step size $\beta > 0$ for the computation of the next iterate, the initial step size $\tilde{\beta}$ 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 $\rho \in (-1,1)$ and a projection pr, where the default is the identity function $\mathrm{id} : x \to x$.

The implementation in pseudo code is shown here:

Algorithm 1 EnOpt algorithm

```
1: procedure EnOpt($F, \mathbf{q}_0, N, \varepsilon, k^*, \beta, \tilde{\beta}, r, \nu^*, \sigma^2, \rho, \mathrm{pr} = \mathrm{id}$)
2:     $F_0 \leftarrow F(\mathbf{q}_0)$
3:     $\mathbf{q}_1, T_1, \mathbf{C}^0_{\mathbf{q}_0}, F_1 \leftarrow \mathrm{OptStep}(F, \mathbf{q}_0, N, 0, \{\}, 0, F_0, \beta, \tilde{\beta}, r, \varepsilon, \nu^*, \sigma^2, \rho, \mathrm{pr})$
4:     $k \leftarrow 1$
5:     while $F_k > F_{k-1} + \varepsilon$ and $k < k^*$ do
6:         $\mathbf{q}_{k+1}, T_{k+1}, \mathbf{C}^k_{\mathbf{q}_k}, F_{k+1} \leftarrow \mathrm{OptStep}(F, \mathbf{q}_k, N, k, T_k, \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}, F_k, \beta, \tilde{\beta}, r, \varepsilon, \nu^*, \sigma^2, \rho, \mathrm{pr})$
7:         $k \leftarrow k + 1$
8:     return $\mathbf{q}^* \leftarrow \mathbf{q}_k$
```

Algorithm 2 OptStep algorithm

```
1: procedure OptStep($F, \mathbf{q}_k, N, k, T_k, \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}, F_k, \beta, \tilde{\beta}, r, \varepsilon, \nu^*, \sigma^2, \rho, \mathrm{pr}$)
2:     if $k = 0$ then
3:         Compute the initial covariance matrix $\mathbf{C}^0_{\mathbf{q}_0}$ like defined in (3.1) with $\mathbf{q}_0, \sigma^2, \rho$
4:     else
5:         $\mathbf{C}^k_{\mathbf{q}_k} \leftarrow \mathrm{updateCov}(\mathbf{q}_k, N, T_k, \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}, F_k, \tilde{\beta})$
6:     Sample $N$ control vectors $\{\mathbf{q}_{k,j}\}_{j=1}^N$ from a distribution $\mathcal{N}(\mathbf{q}_k, \mathbf{C}^k_{\mathbf{q}_k})$
7:     $T_{k+1} \leftarrow \{\mathbf{q}_{k,j}, F(\mathbf{q}_{k,j})\}_{j=1}^N$
8:     Compute the vector $\mathbf{C}^k_{\mathbf{q}_k,F}$ with $\mathbf{q}_k, F_k$ and the stored values of $T_{k+1}$ like in (3.2)
9:     Compute the search direction $\mathbf{d}_k = \mathbf{C}^k_{\mathbf{q}_k,F} / \|\mathbf{C}^k_{\mathbf{q}_k,F}\|_\infty$
10:    $\mathbf{q}_{k+1} \leftarrow \mathrm{LineSearch}(F, \mathbf{q}_k, \mathbf{d}_k, \beta, r, \varepsilon, \nu^*, \mathrm{pr})$
11:    $F_{k+1} \leftarrow F(\mathbf{q}_{k+1})$
12:    return $\mathbf{q}_{k+1}, T_{k+1}, \mathbf{C}^k_{\mathbf{q}_k}, F_{k+1}$
```

The next algorithm uses some functions from the Python package NumPy, which is imported here as np.

Algorithm 3 Covariance matrix update

```
1: procedure UPDATECOV($\mathbf{q}_k, N, T_k, \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}, F_k, \tilde{\beta}$)
2:     $N_q \leftarrow \text{len}(\mathbf{q}_k)$
3:     $\mathbf{d}_k \leftarrow \text{np.zeros}((N_q, N_q))$
4:     assert $\text{len}(T_k) == N$
5:     for $m = 0, \dots, N-1$ do
6:         $\mathbf{d}_k \leftarrow d_k + (T_k[m][1] - F_k) * ((T_k[m][0] - \mathbf{q}_k).\text{reshape}((N_q, 1)) * (T_k[m][0] - \mathbf{q}_k).\text{reshape}((1, N_q)) - \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}})$
7:     $\mathbf{d}_k \leftarrow \mathbf{d}_k / N$
8:     $\mathbf{C}_{\text{diag}} \leftarrow \text{np.zeros}(N_q)$
9:     $\mathbf{d}_{\text{diag}} \leftarrow \text{np.zeros}(N_q)$
10:    for $i = 0, \dots, N_q - 1$ do
11:        $\mathbf{C}_{\text{diag}}[i] \leftarrow \mathbf{C}^{k-1}_{\mathbf{q}_{k-1}}[i, i]$
12:        $\mathbf{d}_{\text{diag}}[i] \leftarrow \mathbf{d}_k[i, i]$
13:    $\tilde{\beta}_k \leftarrow \tilde{\beta}$
14:    while $\text{np.min}(\mathbf{C}_{\text{diag}} + \tilde{\beta}_k * \mathbf{d}_k) < 0$ do
15:        $\tilde{\beta}_k \leftarrow \tilde{\beta}_k / 2$
16:    return $\mathbf{C}^{k-1}_{\mathbf{q}_{k-1}} + \tilde{\beta}_k * \mathbf{d}_k$
```

Algorithm 4 Line search

```
1: procedure LINESEARCH($F, \mathbf{q}_k, \mathbf{d}_k, \beta, r, \varepsilon, \nu^*, \text{pr}$)
2:     $\beta_k \leftarrow \beta$
3:     $\mathbf{q}_{k+1} \leftarrow \text{pr}(\mathbf{q}_k + \beta_k \mathbf{d}_k)$
4:     $\nu \leftarrow 0$
5:     while $F(\mathbf{q}_{k+1}) - F(\mathbf{q}_k) \leq \varepsilon$ and $\nu < \nu^*$ do
6:         $\beta_k \leftarrow r\beta_k$
7:         $\mathbf{q}_{k+1} \leftarrow \text{pr}(\mathbf{q}_k + \beta_k \mathbf{d}_k)$
8:         $\nu \leftarrow \nu + 1$
       return $\mathbf{q}_{k+1}$
```

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: procedure FOM-ENOPT($\mathbf{q}_0, N, \varepsilon, k^*, \beta, \tilde{\beta}, r, \nu^*, \sigma^2, \rho$)
2:     return EnOpt($-j, \mathbf{q}_0, N, \varepsilon, k^*, \beta, \tilde{\beta}, r, \nu^*, \sigma^2, \rho$)
```

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 $N_{\text{in}} = N_0, N_1, \dots, N_{L-1}, N_L = N_{\text{out}}$ the number of neurons in each layer. $W_i \in \mathbb{R}^{N_i \times N_{i-1}}$ denotes the weights in layer $i \in \{1, \dots, L\}$ and $b_i \in \mathbb{R}^{N_i}$ the biases of the layer $i \in \{1, \dots, L\}$. These are composed as $\mathbf{W} = ((W_1, b_1), \dots, (W_L, b_L))$, 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 $\rho_n^*(x) = [\rho(x_1), \dots, \rho(x_n)]^T$ 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:

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

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

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

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]. We call that 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 that 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.

To summarize, 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 testing set $(x_{\text{test}}, y_{\text{test}})$, the loss function (loss_fn), the optimizer (optimizer), and the number of training epochs (epochs). In our case, 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 testing 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 testing set which are saved as y_{train}. So if loss_fn $= \mathscr{L}$, we have $\text{testDNN}(\text{DNN}, x_{\text{test}}, y_{\text{test}}, \text{loss\_fn}) = \mathscr{L}(\text{DNN}, T_{\text{test}})$ with

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

Algorithm 6 DNN training

```
1: procedure TRAINDNN(DNN, $x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}}$, loss_fn, optimizer, epochs)
2:     wait ← 0
3:     minimalTestLoss ← testDNN(DNN, $x_{\text{test}}, y_{\text{test}}$, loss_fn)
4:     save the current parameters of the DNN
5:     for epoch = 1, ..., epochs do
6:         do one training step with the optimizer
7:         testLoss ← testDNN(DNN, $x_{\text{test}}, y_{\text{test}}$, loss_fn)
8:         if testLoss < minimalTestLoss then
9:             wait ← 0
10:            minimalTestLoss ← testLoss
11:            save the current parameters of the DNN
12:        else
13:            wait ← wait + 1
14:        if wait ≥ earlyStop then
15:            overwrite the parameters of the DNN with the saved parameters
16:            return
```

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 combined loss $\mathscr{L}(\Phi_{\mathbf{W}^*}, T_{\text{train}}) + \mathscr{L}(\Phi_{\mathbf{W}^*}, T_{\text{val}})$ 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 double to float, 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

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

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

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

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

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

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

Algorithm 7 DNN construction

```
1:  procedure CONSTRUCTDNN(sample, $V_{\text{DNN}}$, minIn, maxIn)
2:      normSample ← np.zeros((len(sample), len(sample[0][0])))
3:      normVal ← np.zeros(len(sample))
4:      for $i = 0, \dots,$ 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:      scaleInput ← lambda mu : (mu − minIn)/(maxIn − minIn)
10:     scaleOutput ← lambda mu : (mu − minOut)/(maxOut − minOut)
11:     rescaleOutput ← lambda mu : mu · (maxOut − minOut) + minOut
12:     normSample ← scaleInput(normSample)
13:     normVal ← scaleOutput(normVal)
14:     divide normSample and normVal into train/test splits $x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}}$
15:     DNN ← FullyConnectedNN($V_{\text{DNN}}$[0], activation_function = $V_{\text{DNN}}$[1])
16:     trainDNN(DNN, $x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}}$, loss_fn, optimizer, epochs)
17:     evalDNN ← testDNN(DNN, $x_{\text{test}}, y_{\text{test}}$, loss_fn)
18:     for $i = 1, \dots,$ numberOfRestarts do
19:         DNN$_i$ ← FullyConnectedNN($V_{\text{DNN}}$[0], activation_function = $V_{\text{DNN}}$[1])
20:         trainDNN(DNN$_i$, $x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}}$, loss_fn, optimizer, epochs)
21:         evalDNN$_i$ ← testDNN(DNN$_i$, $x_{\text{test}}, y_{\text{test}}$, loss_fn)
22:         if evalDNN$_i$ < evalDNN then
23:             evalDNN ← evalDNN$_i$
24:             DNN ← DNN$_i$
25:     $F_{\text{ML}}$ ← lambda mu : rescaleOutput(DNN(scaleInput(mu)))
26:     return $F_{\text{ML}}$
```

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

5 Numerical experiments

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`.