# 1 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.

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

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

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

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

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 $T_{\text{train}}$ 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 [2]. 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.

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 [3] 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 $i \in \{1, ..., L\}$. The biases $b_i$ are set to zero for $i \in \{1, ..., L\}$. The idea behind the random sampling is that the specified standard deviation prevents the exponential increase/ reduction of the input as shown in [3].

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.

**Algorithm 1** DNN construction

```
procedure 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) do
        normSample[$i$, :] ← sample[$i$][0]
        normVal[$i$] ← sample[$i$][1]
    minOut ← np.min(normVal)
    maxOut ← np.max(normVal)
    normSample ← normSample − minIn
    normVal ← normVal − minOut
    normSample ← normSample/(maxIn − minIn)
    normVal ← normVal/(maxOut − minOut)
    divide normSample and normVal into train/test splits $x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}}$
    DNN ← FullyConnectedNN($V_{\text{DNN}}$[0], activation_function = $V_{\text{DNN}}$[1])
    trainDNN(DNN, )
    evalDNN ← testDNN(DNN, )
    for $i = 1, \ldots,$ numberOfRestarts do
        DNN$_i$ ← FullyConnectedNN($V_{\text{DNN}}$[0], activation_function = $V_{\text{DNN}}$[1])
        trainDNN(DNN$_i$, )
        evalDNN$_i$ ← testDNN(DNN$_i$, )
        if evalDNN$_i$ < evalDNN then
            evalDNN ← evalDNN$_i$
            DNN ← DNN$_i$
    define the function $F_{\text{ML}}$
    return $\mathbf{q}_{k+1}, T_{k+1}, \mathbf{C}^k_{\mathbf{q}_k}, F_{k+1}$
```

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

# 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] 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`.

[3] 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`.