Ludwig-Maximilian University of Munich

Master Thesis

# Configuration of Deep Neural Networks Using Model-Based Optimization

*A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Statistics at the*

Department of Statistics

Author: Benjamin Klepper

Supervisor 1: Prof. Dr. Bernd Bischl
Supervisor 2: Janek Thomas

Study program: Statistics with Focus on Economics and Social Science

January 3, 2018

# Declaration of Authorship

I, Benjamin Klepper, declare that this thesis titled "Configuration of Deep Neural Networks Using Model-Based Optimization" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Date:

Signed:

# Abstract

Deep neural networks have improved the performance results in image recognition, speech recognition and many other domains. Manually configuring the architecture and other hyperparameters of deep neural networks becomes unfeasible for large hyperparameter spaces. Automating this process requires the optimization of an expensive black-box function over a mixed and hierarchical search space. Model-based optimization is a state-of-the-art derivative-free technique for optimizing expensive black-box functions. This thesis extends the standard model-based optimization approach to take into account the high computational costs and search space complexity when configuring deep neural networks. The resulting algorithm's performance is compared to random search, the standard model-based optimization approach and a hand-designed model on CIFAR-10, Fashion-MNIST and MNIST rotated with background image. For this purpose, the R machine learning package mlr is extended to include functionalities of the deep learning framework MXNet. Of all algorithms, the standard model-based approach achieves the best average rank across all data sets. Nevertheless, all algorithms yield comparable results. On average, the hand-designed model outperforms the automatically configured models. The results indicate that the automatic configuration of kernel and stride sizes for convolutional layers of neural networks is inefficient when choosing the values independently. Several possibilities for improving the performance of the model-based optimization extension are proposed.

# Contents

# List of Figures

# List of Acronyms

| | |
|---|---|
| **AI** | artificial intelligence |
| **CIFAR** | Canadian Institute for Advanced Research |
| **DACE** | Design and Analysis of Computer Experiments |
| **EGO** | Efficient Global Optimization |
| **EI** | expected improvement |
| **GP** | Gaussian process |
| **MBO** | model-based optimization |
| **MBOINL** | model-based optimization with increasing number of layers |
| **PI** | probability of improvement |
| **ReLU** | rectified linear unit |
| **RF** | random forest |
| **SGD** | stochastic gradient descent |
| **SMBO** | sequential model-based optimization |

# Chapter 1

# Introduction

People speculated about programmable computers becoming intelligent when they were first conceived, which was over a hundred years before one was actually built [1]. Currently, artificial intelligence (AI) is a very active, rapidly growing area of research with a wide range of practical applications like speech recognition [2], image recognition [3], machine translation [4], automating repetitive tasks and making medical diagnoses [1]. In its early days, AI was used very successfully to solve problems that are intellectually difficult for humans and which can be formalized by a set of mathematical rules [1]. More challenging, however, are problems that humans solve intuitively but that are hard to formalize, like recognizing objects [1].

Several projects employing a knowledge-based approach to artificial intelligence (attempting to hard-code knowledge in a formal language) have not led to a major success [1]. This suggests that AI algorithms should be given the ability to extract their own knowledge from data [1].

Machine Learning models are models that can acquire knowledge by extracting patterns from raw data. These models can be used to identify objects in images, make product recommendations based on user preferences and select relevant results for web search queries. They are therefore present in many services and products, for example in e-commerce, social networks and also increasingly in cameras and smartphones [5].

While simple machine learning algorithms can help make decisions of a seemingly subjective nature like recommending cesarean delivery or detecting spam email, their performance depends on the representation of the provided data. Ideally, this data would consist of features that are highly correlated with the output. In contrast, providing pixels of an image for object detection would probably result in poor performance when using simple machine learning algorithms.

As the ability of simple machine learning models to process raw data is limited, profound domain knowledge and careful engineering is required to design and extract features from the original raw data and transform them into a suitable representation to serve as input for the machine learning model. Many learning tasks can be solved by designing and extracting meaningful features from data and applying a simple machine learning algorithm to them [1].

Prior to deep learning methods, many applications of machine learning combined hand engineered features with linear classifiers, which can only divide the input space into half-spaces separated by a hyperplane [5]. Complex problems like image recognition require the model to be insensitive to irrelevant variations of the input, like the position of an object, but sensitive to relevant variations like the difference between a white wolf and a white wolf-like dog [5]. On a pixel level, a linear classifier would be very sensitive to the former but rather insensitive to the latter and would therefore require pre-designed features.

Designing features may prove to be a difficult endeavour. In image recognition it can be difficult to describe a certain object in terms of pixel values as for example angle, position and brightness can vary for different images.

Representation learning refers to methods that can discover suitable representations from the data automatically [5]. In representation learning, machine learning is not only used to construct a mapping from the features to the output but also from the input data to the features (representation). Algorithms equipped with this capability can easily be applied to different kinds of tasks with little manual overhead while yielding very good performance, often better than with manually designed features [5].

Good features should separate the factors of variation in the data, which may be unobserved directly in the data but affect the observed data [1]. These factors can also be concepts or abstractions and can once again be illustrated with the angle, position and brightness of an object in a picture. Depending on the level of abstraction, some factors of variation may be difficult to extract directly from the data.

Deep learning solves this problem by employing a hierarchical approach to learning representations so that more abstract representations can be learned from previously learned, simpler representations. Deep learning models are therefore representation learning models with a hierarchy of representational levels and hence several levels of abstraction [5]. This hierarchical concept of representations that build upon each other and become increasingly complex allows the computer to learn more complicated concepts. The models are composed of multiple processing layers, each layer often a simple but non-linear module that transforms the representation to a more abstract one. Each layer has certain internal parameters to compute the representation of the data using the previous layer's representation [5]. An algorithm might first learn to detect different kinds of edges, then to detect shapes using combinations of edges and then to detect objects as combinations of shapes. A graph of the hierarchy of representations is multilayered and deep, hence the name deep learning.

An alternative interpretation of deep learning is that it is a sequential computer program with each layer being the state of the memory after executing a set of instructions in parallel. As an instruction in a sequence of instructions can refer to previous ones, the sequential approach offers greater flexibility. The difference in this kind of interpretation is that not all of the parameters in the model encode factors of variation but also state information to help execute the program that in turn produces a meaningful output from the input [1].

In recent years, deep learning has beaten other machine learning techniques not only at image recognition [6, 7] and speech recognition [8, 9] but also other tasks like reconstructing brain circuits [10], natural language understanding [11] and analyzing particle accelerator data [12].

The applications in practice thus increasingly make use of deep learning methods. When a sufficient number of transformations are composed, complex structures can be learned from data. In image classification, the first layer may represent edges (and whether they are present or absent) at certain locations. The second layer may detect specific arrangements of edges. The third layer may combine these arrangements to parts or objects. Therefore, the higher layers of representation are influenced less by unimportant variation in the data. The representation learned at each layer is extracted from the data without human supervision by using a general learning approach.

Supervised learning is the most common form of machine learning [5]. In supervised learning, the data used for training the model contains a so-called target, giving the correct output value for the feature values associated with it. Using an objective function that measures the error for the correct output and the output values computed by the model, the model adjusts its internal parameters to reduce the error. This objective function is also called performance measure.

In a deep learning model these internal parameters, also called weights, occur in large numbers. To correctly modify the weights, the learning algorithm typically computes the gradient vector which indicates how the error would change if the weights changed by a small amount. The weights are then adjusted in the opposite direction of the gradient (as the gradient gives the direction of the steepest ascent and the error should be minimized).

A very commonly used method is stochastic gradient descent (SGD) [5], which consists of repeatedly updating the weights with the average gradient of a random subsample of the training data. The gradient of each sample is a noisy estimate of the average gradient of the complete training data. After training, the performance of the model is measured on a different data set called test set. This provides a more reliable estimation of the model's ability to generalize (make meaningful predictions for unseen inputs).

Like the majority of machine learning algorithms, neural networks have so-called hyperparameters, parameters whose values are not learned during training and therefore have to be set in advance. The process of selecting a good (in some meaningful way) setting of hyperparameters is also referred to as tuning. In many cases, tuning the hyperparameters of neural networks proves to be especially difficult for several reasons.

First, neural networks may be sensitive to the hyperparameter setting, therefore it must be chosen carefully.

Second, the tuning process includes optimizing discrete and conditional hyperparameters, especially when configuring the architecture of a neural network. Additionally, the number of tunable hyperparameters can be very high compared to other machine learning models. The resulting hyperparameter space is a high-dimensional and non-trivial hierarchical search space.

Third, training a neural network can be computationally very costly. Designing neural networks by hand therefore requires expert knowledge and is in many cases not feasible [13].

Naturally, choosing good hyperparameters automatically with an efficient optimization algorithm is very desirable.

Optimization is a field of study that a great deal of research has been devoted to and that finds application in a wide range of areas, e. g., economics or mechanical design [14]. One specific problem setting is the optimiziation of a nonlinear function (the so-called *objective function*) over a compact set. Common assumptions include that this objective function has a closed analytical form which is known or is cheap to evaluate [15]. In practice, however, the objective function is often a black-box function [15]. These functions can be viewed as systems that produce numeric output given some input without much knowledge about their inner workings being available, also resulting in an unavailability of gradients [16]. Additionally, they may be expensive to evaluate as they often correspond to expensive processes like drug trials or financial investments [15]. Black-box functions also occur for many problems in machine learning, for example when optimizing hyperparameters of machine learning algorithms (see, e. g., [17, 18, 19]). In this scenario, the inputs of the black-box function could be the hyperparameters of the algorithm and the output one or more performance measures. More complex scenarios are also possible, e. g. optimizing a machine learning pipeline (see [16]). Optimizing a black-box function entails being restricted to obtaining observations by evaluating the function at sampled values and receiving a possibly noisy response. Since evaluations are expensive, it is desirable to optimize the function using a minimal number of evaluations. Due to the function's nature, optimization techniques that require differentiability or independent parameters are not directly applicable.

A lot of research has been devoted to finding good hyperparameters automatically using various approaches like random search [20], bandit-based approaches [21] and model-based optimization (MBO) (also called Bayesian optimization) [22, 23].
Other possible techniques are, for example, evolutionary algorithms [24, 25] and reinforcement learning [26, 27, 28]. However, both of these methods either have high demand for computational resources or are not competitive regarding their performance [13, 29].
MBO is a derivative-free approach for global optimization of black-box functions. Incorporating a sequential design strategy when applying MBO, also called sequential model-based optimization (SMBO) [30], has become an optimization strategy with state-of-the-art performance [16]. In MBO, the objective function (e. g. the machine learning model's performance) is approximated using a statistical model (e. g. a Gaussian process (GP)). This so-called surrogate model is then used to propose a new point that is likely to yield good performance according to some pre-defined criterion. Applying the standard MBO approach to tuning neural networks has revealed several areas with potential for improvement.
First, only numerical parameters can be tuned with a Gaussian process surrogate model when using a standard kernel. Hutter and Osborne [31] define a kernel for mixed continuous and discrete parameter spaces. Building on this, Swersky et al. [32] introduce kernels for Bayesian optimization of conditional parameter spaces. Wang et al. [33] introduce a new technique to eliminate the need for optimizing the acquisition function. Lévesque et al. [34] use model-based optimization for conditional hyperparameter spaces by injecting knowledge about conditions in the kernel. This is illustrated by simultaneously doing al-

gorithm selection and hyperparameter tuning.
Second, the scalability of MBO to higher dimensional parameter spaces can be improved. Wang et al. [35] use random embeddings to improve the scalability of Bayesian optimization to high dimensions for domains of continuous as well as discrete variables. Springenberg et al. [36] introduce an approach of using neural networks as surrogate models to improve the scaling of Bayesian optimization regarding the number of both hyperparameters and evaluations of the objective function.
Third, since training neural networks may become computationally costly, it is desirable to improve the resource allocation in the MBO process. Yogatama and Mann [37] propose a method of MBO that aims to reduce computational time by transferring information between data sets using a common response surface. Klein et al. [37] introduce a Bayesian optimization procedure which models loss and training time with respect to the size of the training set to extrapolate the validation error for a subsample of the training set to the whole training set. Feurer et al. [38] explore a way of transferring knowledge of hyperparameter configurations between data sets by using a metalearning procedure to propose the initialization of the MBO tuning process. Swersky et al. [39] provide a new technique for Bayesian optimization to decide when the training process of a model should be paused to start with a new one or continue training a previously paused one. This is done for models that are trained in an iterative fashion including neural networks. Illievski et al. [40] introduce a way to make Bayesian optimization for tuning deep neural networks more efficient by using a deterministic surrogate model. Smithson et al. [41] propose tuning neural networks by using Bayesian optimization with multi-objective exploration of the hyperparameter space using a neural network as the surrogate model, aiming to reduce the number of objective function evaluations. Domhan et al. [42] introduce a technique to extrapolate the learning curves of neural networks during training, aiming to determine if the training process should be stopped early to improve resource allocation. Hutter et al. [43] propose an MBO algorithm that can handle categorical parameters by using a random forest based surrogate model. Elsken et al. [13] use network morphisms (introduced as network transformations by Chen et al. [44]) to generate pre-trained models in optimizing the architecture of convolutional neural networks to reduce computational costs. Liu et al. [29] propose configuring the architecture of convolutional neural networks by training architectures with increasing complexity using a SMBO approach with a recurrent neural network surrogate and no acquisition function, matching state-of-the-art performance.
Besides these main points, other approaches to extend the MBO framework have been proposed. For example, Wu et al. [45] introduce a new Bayesian optimization algorithm that uses derivative information.
Welchowski and Schmid [46] tune kernel deep stacking networks using a combination of MBO and hill climbing.

This thesis presents an approach to extend MBO for tuning the hyperparameters (including the architecture) of feedforward and convolutional neural networks by combining MBO with a procedure inspired by hill climbing and multi-fidelity optimization. The new approach is compared to random search and the standard MBO method on CIFAR-10 [47],

Fashion-MNIST [48] and MNIST rotated with background images (used in [20], a variation of the well-known MNIST data set [49]). To carry out these benchmarks, a dedicated extension of the popular R [50] machine learning package mlr [51] is created to integrate functionalities of the MXNet [52] deep learning framework. All MBO procedures are done using the R package mlrMBO [16].
The remainder of the thesis is structured as follows: The concept of feedforward neural networks is introduced, focusing on fully connected and convolutional layers. Then, an overview of MBO is given and the model-based optimization with increasing number of layers (MBOINL) algorithm is introduced. Subsequently, the software, namely mlr, mlrMBO and MXNet, is described and illustrated by some examples. Finally, benchmark results are presented, followed by the summary and conclusion. All figures regarding the benchmark results have been created with R using the batchtools [53] package, the figures on neural networks and MBO have been created with Inkscape [54].

# Chapter 2

# Deep Neural Networks

## 2.1 History of Deep Learning

While the field of deep learning received its name fairly recently, its roots are in the 1940s. There have been three main waves of development.

In the 1940s-1960s the first wave was known as *cybernetics*, where simple linear models were used with a neuroscientific motivation (see, e. g., [55]). The limitations of linear models led to a decline of interest in neuroscientifically inspired learning, causing the first dip in popularity.

The second wave was from the 1980s to the mid-1990s and emerged from *connectionism*, which arose in the context of an interdisciplinary study of the mind called cognitive science, with the central idea that simple computational units achieve intelligence by being connected in a network [1]. A key concept from connectionism is *distributed representation* (see [1]): Each input in the representation should be represented by multiple features and each feature should be relevant for the representation of multiple inputs. Suppose the input is an image of either a cat or a dog, which is either brown or black. A possible representation of the input is to have a binary feature for each possible input, resulting in four features (features *black_cat*, *black_dog*, *brown_cat* and *brown_dog*, where exactly one has value 1 and the others have values 0). A distributed representation would be to have a feature for the type of animal (cat or dog) and for the color (brown or black).

Several groups independently discovered a simple way to train multilayer networks during the 1970s and 1980s using SGD [56] [57, 58, 59]: As long as the individual modules from the multilayer stack can be represented by sufficiently smooth functions of inputs and internal weights, the gradients can be obtained by using the so-called backpropagation procedure. The backpropagation procedure is basically an application of the chain rule for derivatives to compute the gradient of the objective function with respect to the internal weights of the multilayer network (by working backwards through the network from layer to layer).

Arguably, the work of Rumelhart et al. [59] had the most impact [1]. The second wave of neural networks ended because unfulfilled expectations driven by unrealistic claims of AI ventures coincided with other AI approaches achieving good results [1].

In the late 1990s neural networks were in large parts ignored by the communities of machine learning, computer vision and speech recognition since the approach was considered infeasible due to the risk of the gradient descent stopping in local minima [5]. Theoretical and empirical results suggest, however, that local minima are rarely a problem. Instead, the search space often contains saddle points where the gradient is zero and more importantly the values of the objective function are very similar. Hence, it does not make a big difference at which saddle point the algorithm is stuck [5].

In 2006 a research collaboration organized by the Canadian Institute for Advanced Research (CIFAR) resulted in creating unsupervised learning algorithms (algorithms that learn from data that consists of examples $\{x\}$ without any corresponding target value) that could create multilayered feature detectors from unlabelled data. A multilayered feature detector would be trained with unlabelled data (also called unsupervised pre-training) and afterwards an output layer would be added, creating a neural network with appropriate initial weight values that could be fine-tuned using backpropagation [5]. The third wave of AI research began in 2006 with a breakthrough in training a so-called deep belief network using a method called greedy layer-wise pretraining [60], which was quickly extended to other types of networks [61, 62].

At the time of writing, the third wave is ongoing and deep learning methods regularly outperform alternative AI methods [1]. However, the focus of the third wave has shifted from unsupvervised learning and generalizing from small datasets to leveraging large labeled datasets. Accordingly, the focus of this work lies on supervised learning.

Besides new techniques to train deep networks, two factors have played a crucial role in the increasing performance of deep learning methods. One is the increasing availability of appropriate and large datasets for machine learning applications. The other is the availability of computational resources, like faster CPUs, the advent of GPUs and better infrastructure for distributed computing, to run larger models. This has resulted in dramatically improved predictions and accuracy of recognition as well as an increased variety of successful applications.

## 2.2 Feedforward Neural Networks

Basic artifical neural networks implement a function $y(x; w)$. The central idea is that the algorithm can learn a relationship between inputs $x$ and target $t$ using provided examples. For a given $x$, a trained network should then output $y$ that is close to $t$ (in some appropriate way). Training a network consists of searching the parameter space of weights for values of $w$ that produces a function fitting the training data well (and generalizes well to unseen examples). Typically, this so-called performance of the model is measured using some objective function (error function) as a function of $w$. The training process is therefore a function optimization. Many optimization algorithms not only make use of the objective function but of its gradient (with respect to $w$) as well. For feedforward networks (see Subsec. 2.2.2), the backpropagation algorithm evaluates the gradient of output $y$ w.r.t. $w$, and therefore the gradient of the objective function w.r.t $w$ [63].

The following introduction can be found in [63].

### 2.2.1 Neuron

Neurons are the most basic building blocks for neural networks. We therefore seek to understand them before moving on to more complex networks. Indeed, a single neuron can be viewed as a very simple supervised neural network.
Assume we have a number $I \in \mathbb{N}$ of inputs $x_i$ and one output $y$. In a single neuron, each input is assigned a weight $w_i$. Additionally, there may exist an additional parameter $w_0$ called *bias*, which can be interpreted as the weight of input $x_0 = 1$. An *activity rule* is a local rule defining how the *activity* (output) of a neuron changes. The activity rule of a single neuron has two steps:

- An *activation* $a = w^\top x$, with either $w = (w_0, w_1, ..., w_I)$ and $x = (x_0, x_1, ..., x_I)$ or $w = (w_1, ..., w_I)$ and $x = (x_1, ..., x_I)$.

- An *activation function* $f$.

A single neuron as a binary classifier could receive a number of input vectors $\{x^{(n)}\}_{n=1}^N$ with binary labels $\{t^{(n)}\}_{n=1}^N$ and produce an output $y(x; w)$ between 0 and 1 for each $x^{(n)}$. Note that $x^{(n)} = (x_1^{(n)}, ..., x_I^{(n)})$ or $x^{(n)} = (x_0^{(n)}, x_1^{(n)}, ..., x_I^{(n)})$. An objective function can be chosen as

$$G(w) = -\sum_{n=1}^{I} t^{(n)} \ln(y(x^{(n)}; w)) + (1 - t^{(n)}) \ln(1 - y(x^{(n)}; w)). \tag{2.1}$$

The activity is then computed as $y = f(a)$. Defining

$$g_j^{(n)} = -(t_j^{(n)} - y_j^{(n)}) x_j^{(n)}, \tag{2.2}$$

we can write

$$g_j = \frac{\partial G}{\partial w_j} = \sum_{n=1}^{N} g_j^{(n)}. \tag{2.3}$$

Therefore the gradient $\partial G / \partial w$ can be written as a sum of vectors $g^{(n)}$.
A simple on-line gradient descent training algorithm would be to sequentially put each observed input example through the network and update the weights $w$ in the opposite direction of $g^{(n)}$. More precisely, the weights are updated by

$$\Delta w_i = \eta (t - y) x_i^{(n)}. \tag{2.4}$$

Here, $\eta$ is the *learning rate*, the length of the step of each update, typically between 0 and 1. Choosing the right learning rate is not a trivial task. A learning rate that is too big might result in the updated weight “jumping” over an optimum. A learning rate that is very small however may result in very long computation time during the training phase as the small steps lead to a longer time until convergence.

An alternative to this on-line paradigm would be to use a batch of examples to update the weights. For $(x^{(n)}, t^{(n)})$ $(n = 1, ..., K), K \leq N$, the update would be $\Delta w_i = \sum_{n=1}^{K} g_i^{(n)}$. The number of updates times the number of examples in one batch can be larger than the actual number of observations in the training data. In this case, the updating process cycles through the data more than once. Therefore, observations are used multiple times for updates but not necessarily in the batch constellation. One update cycle through the training data is called epoch.

The phenomenon of *overfitting* occurs when the model fits the training data so well that the generalization ability of the model decreases. A high level interpretation would be that at a certain point during training, random noise in the data is interpreted as signal. As the overall aim is to obtain a model with good generalization, it is desirable to avoid overfitting. An intuitive approach called *early stopping* (see Subsec. 2.4.2) consists of monitoring a metric (usually some indicator of the ability to generalize) during training and utilizing a stopping criterion (e. g. when generalization worsens) to interrupt the training process. Another approach, called *regularization*, is to modify the objective function to incorporate a bias against unfavorable solutions $w$. A thorough overview of regularization for deep learning can be found in [1].

### 2.2.2 Multilayer Perceptron

A neural network is called a *feedforward* network if the neurons and their connections form a directed acyclic graph. They consist of a sequence of layers of neurons (where the neurons in each layer are not connected to each other). More precisely, they have an input layer, an output layer and a number of layers in between called hidden layers. They can be seen as a nonlinear parameterized transformation of inputs which can be used for regression as well as classification tasks. A fully connected layer is a layer of neurons where each neuron has all neurons from the previous layer as inputs and its output is in turn input to all the neurons from the subsequent layer. A multilayer perceptron is a feedforward network where all hidden layers are fully connected layers, see for example Fig. 2.1. A regression multilayer perceptron with one hidden layer could have $I$ inputs $x_i$, a hidden layer with

$$a_j^{(1)} = \sum_{l=1}^{I} w_{jl}^{(1)} x_l + \theta_j^{(1)},\ h_j = f^{(1)}(a_j^{(1)}),\ j = 1, ..., n_{\text{hidden}} \tag{2.5}$$

and an output layer with

$$a_i^{(2)} = \sum_{j=1}^{n_{hidden}} w_{ij}^{(2)} h_j + \theta_i^{(2)};\ y_i = f^{(2)}(a_i^{(2)}),\ i = 1, ..., n_{\text{out}}, \tag{2.6}$$

where $n_{\text{hidden}}$ is the number of neurons in the hidden layer, $n_{\text{out}}$ is the number of output neurons and $f^{(1)}$ and $f^{(2)}$ are the activation functions.

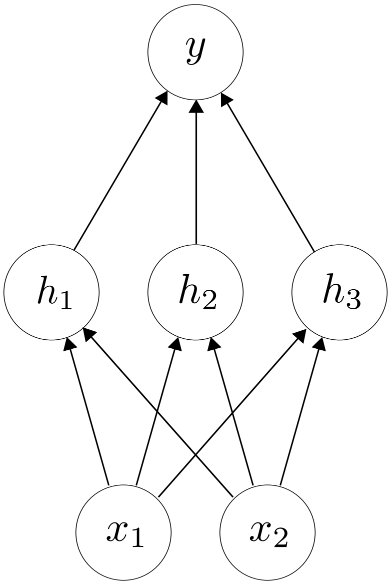


Figure 2.1: A multilayer perceptron with one hidden layer.

## 2.3 Convolutional Neural Networks

Convolutional neural networks are a special kind of network for data with grid-like topology. A very easy example would be 1-D time series data or 2-D image data containing the grey scale pixel values. A more complex example would be 3-D image data, such as actual 3-D images, for example medical CT scans, or 2-D color images with three so-called channels respectively containing the red, green and blue value of a pixel, which could also be viewed as a 2-D grid of vectors. As the name suggests, convolutional neural networks make use of a special linear mathematical operation called convolution. In essence, convolutional neural networks are feedforward neural networks that use convolution operations in one of the layers. A simple convolutional network could consist of an input layer, a hidden convolutional layer and an output layer that is fully connected to the outputs of the convolutional layer.

### 2.3.1 Convolution Operation

The convolution operation can be viewed as an aggregation of (function) values $x(t)$ with a weighting function $w(t)$

$$s(t) = \int x(a)w(t-a)\,\mathrm{d}a, \tag{2.7}$$

denoted by $s(t) = (x * w)$.
In the deep learning context, $x(t)$ is referred to as input, $w(t)$ as kernel and $s(t)$ as feature

map. If the domain of $x$ is discrete, the convolution takes the following form:

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{2.8}$$

In practice, the sum consists of finitely many parts. Convolutions also exist for multiple dimensions. A two-dimensional convolution with discrete domain space, input $I$ and kernel $K$ would be

$$S(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{2.9}$$

### 2.3.2 Motivation

Convolutions leverage three important ideas: Sparse interactions, parameter sharing and equivariant representations [1].
In contrast to a multilayer perceptron with fully connected layers, a convolutional network employs sparse interactions (also referred to as sparse connectivity or sparse weights) in the sense that the kernel does not take into account all inputs when computing a certain feature map value. For a time series, the kernel $w(t) = \frac{1}{3}\mathbb{1}_{\{-1,0,1\}}(t)$ would take the average of the value at time $t$ and its adjacent values. Therefore, the feature map would simplify to $s(t) = \frac{1}{3}(x(t-1)+x(t)+x(t+1))$. These kinds of sparse interactions can lead to reduction of required memory and improved statistical efficiency, as well as fewer computational operations [1].
In parameter sharing, parameters are used for more than one function in a model. Where every connection in a fully connected layer has an individual weight, the weights of the kernel stay the same for every location to which it is applied, leading to memory efficiency. This form of parameter sharing also leads to so-called equivariance of translation. A function $f$ is equivariant to function $g$ if $f(g(x)) = g(f(x))$. This means, for example, that for a time series at time $t$, it would make no difference whether we first apply the kernel and then shift all values one position to the left or first shift the values one position to the left and then apply the kernel.

### 2.3.3 Convolution and Pooling

A typical convolutional layer consists of three stages:

1. Apply several convolutions in parallel to obtain a set of activations.

2. Apply an activation function to the activations to obtain a set of activities.

3. Apply a pooling function that maps a location of the activities to a local summary statistic.

An example of pooling is the so-called max pooling where the maximum of a number of adjacent values is taken. Pooling often leads to approximate invariance of the representations with respect to small translations of input (if we swap to adjacent values, the maximum of a region is likely to stay the same). This invariance is useful if we care more about the existence of a certain feature rather than its exact location. Since pooling summarizes responses over neighborhoods of values, statistical efficiency can be improved if we compute the statistics only for regions spaced $k$ pixels apart instead of 1 pixel apart. This is also referred to as stride of size $k$. Assume we have values $x_1, ..., x_8$, then max pooling with kernel size 4 and stride 2 would give us $max(\{x_1, ..., x_4\})$, $max(\{x_3, ..., x_6\})$ and $max(\{x_5, ..., x_8\})$.
The convolutions used for neural networks in practice are usually different from the standard discrete convolution [1]. Usually, many convolutions are applied in parallel to extract different kinds of features, for example different kinds of edges in the same region of an image. These different convolutions are also referred to as filters or channels. The input itself is often not just a grid but a grid of vectors. To illustrate this on an example, assume we have 2-D data $V = (V_{j,k})$. We want to apply the 3-D kernel tensor $K = (K_{j,k,l})$ where

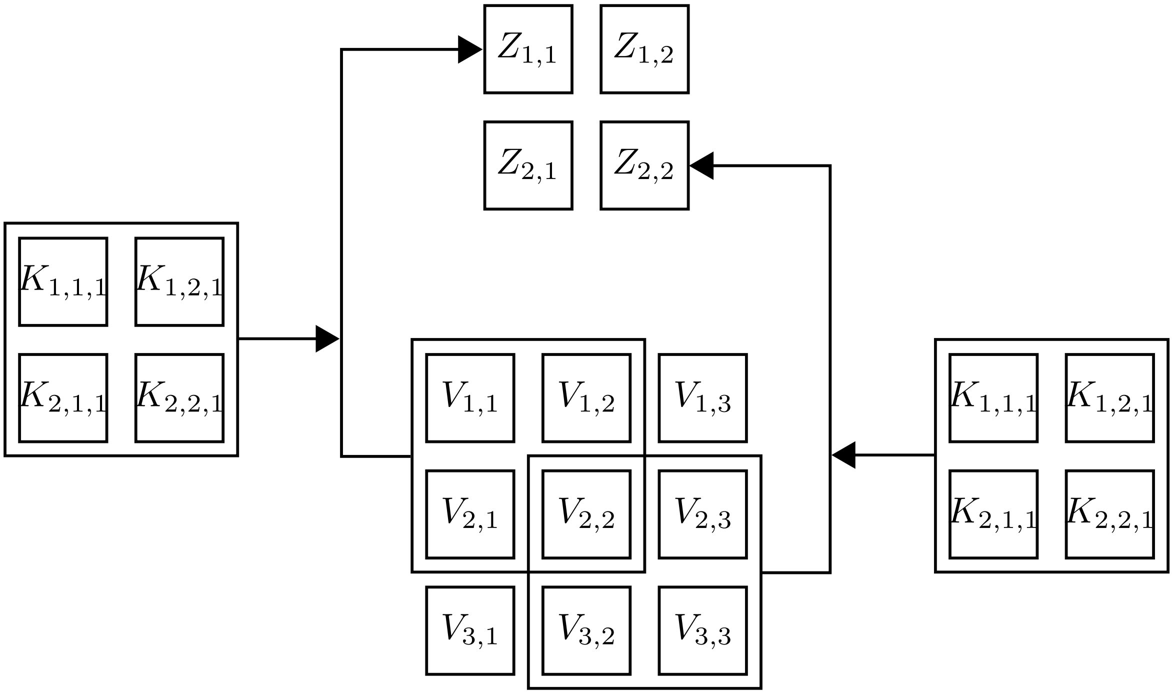


Figure 2.2: Example of a convolution. The $2 \times 2$ sized kernel $K$ of the first filter is applied to the 2-dimensional input $V \in \mathbb{R}^{3\times3}$ with stride 1 to produce the $2 \times 2$ sized output $Z$.

$K_{j,k,l}$ is the connection of an input unit with the output unit of the $j-$th filter with an offset of $k$ rows and $l$ columns between output and input unit (see Fig. 2.2 for an example).

Therefore output $Z = (Z_{j,k})$ can be written as

$$Z_{j,k} = \sum_{j,m,n} = V_{l,j-1+m,k-1+n} K_{l,m,n}. \tag{2.10}$$

For the convolution operation, we can also skip over some positions of the kernel in order to reduce computational costs. With stride $s$ (sample only every $s$ pixels) we have

$$Z_{j,k} = \sum_{j,m,n} = V_{l,(j-1)s+m,(k-1)s+n} K_{l,m,n}. \tag{2.11}$$

Another essential feature of convolution is zero padding the input $V$ (adding rows and columns of zeros at the margins). As the width of representations shrinks every layer when using convolutions, zero padding allows control over the kernel width and the size of the output independently without having to choose between fast shrinking of dimensions or small (enough) kernel sizes.

## 2.4 Initialization, Regularization and Optimization

### 2.4.1 Weight Initialization

Training deep learning models with current techniques is an iterative procedure and requires an initial set of values. These values can have a substantial impact on the speed of convergence, the quality of the found solution and the occurence of numerical difficulties [1]. It is not yet well defined which properties initial values should possess, therefore most initialization strategies are simple and heuristic [1]. One well-known property, however, is symmetry breaking. When initialized with the same weights, units with similar inputs and activation functions will, in essence, learn the same representation [1]. Initializing weights by sampling from a high entropy distribution is both cheap and unlikely to assign the same initial values. Popular choices are the uniform and the Gaussian distribution. Biases are typically set to values that are chosen heuristically [1].

### 2.4.2 Early Stopping

Early stopping is the most common form of regularization and requires almost no change in the training process [1]. When training models with a large effective capacity, overfitting can occur. This can be counteracted by early stopping. The idea is to monitor the generalization error during the training phase and stop training if the generalization error is increasing or has not decreased for a certain amount of time or update iterations. Monitoring the generalization error requires a validation set, which cannot be used for training. In cases where little data is available for training, this may be rather undesirable. However, there are strategies to still make use of the validation data during training (see [1] for details).

### 2.4.3 Dropout

Dropout [64] is a regularization method that is applicable for almost all models using distributed representation. It is computationally inexpensive as well as powerful [1]. In *bagging*, a number of different data sets are created by sampling with replacement from the original data set. Subsequently, the same number of models are trained, where one model is trained on exactly one data set. Dropout can be seen as an approximation of training (and evaluating) a bagged ensemble of exponentially many neural networks [1]. For each batch of data during training a kind of binary mask is randomly created. This binary mask determines for each input unit and hidden unit whether it is included or not. The binary values are sampled independently with a predefined probability (which is therefore a hyperparameter). The mask for each batch is independent from all the other masks created. In some sense, an ensemble of all sub-networks of the original network is trained. For a more in-depth introduction to dropout, please refer to [1]. Dropout has proven to be more effective than other standard computanionally inexpensive regularizers such as weight decay [64]. Although the actual cost of applying dropout during training is low, the cost for the complete system might end up being significant [1]. As dropout is a form of regularization, it reduces the effective capacity of the model and may therefore create the need to increase the model size, resulting in a potentially more expensive training and evaluation process.

### 2.4.4 Batch Normalization

Batch normalization [65] is an optimization technique that is actually not an algorithm but an adaptive reparameterization method. Deep models consist of a hierarchy of layers and their training involves updating certain weights. In some sense, the gradient update of the parameters is done under the implicit assumption that the weights of the other layers do not change when, in reality, the layers are actually updated simultaneously [1]. Batch normalization reduces the problem of coordinating these updates. It can be applied to individual input and hidden layers. The idea behind it is to reparameterize the activations of a layer for a given batch update by subtracting the mean and dividing the result by the standard deviation. Note that the output of each neuron is reparameterized separately and the mean and standard deviation are computed with respect to the different values coming from the batch data. An important point is that the back-propagation should also go through these operations. For more information on batch normalization please see [1].

### 2.4.5 Momentum

Training a neural network with SGD can be slow. Momentum is an optimization technique designed to improve the amount of required training time for high curvature (of the objective function w.r.t. the parameters), small and consistent gradients and noisy gradients. The idea of momentum can be explained informally by using the analogy of a ball rolling through the space that should be optimized. At a given moment, the negative gradient at the position of the ball gives the direction of steepest descent and therefore the direction in which the ball tends to roll. However, it does not always roll in the direction of the negative gradient since it may be in motion rolling towards a direction given by previous negative gradients pointing in other directions than the current one. Furthermore, if the descent is rather flat but always points in the same direction, the ball picks up more and more speed along the way. More formally, the step update is not only the negative gradient multiplied by the learning rate but also includes a exponentially decaying moving average of past gradients. Let $\alpha \in [0, 1)$, $\epsilon \in [0, 1)$ (learning rate) and $g_\theta$ denote the gradient at $\theta$. Then

$$v \leftarrow \alpha v - \epsilon g_\theta, \tag{2.12}$$

$$\theta \leftarrow \theta + v. \tag{2.13}$$

The hyperparameter $\alpha$ determines how quickly the contribution of $v$ decays. For a more thorough introduction to momentum, please refer to [1].

## 2.5 Hyperparameter Selection

A large part of deep learning algorithms have many hyperparameters that influence a majority of its behaviour concerning computational time, memory resources and the resulting model quality (in terms of the ability to generalize) [1]. Hyperparameters can be chosen either automatically or manually. Manual selection requires a thorough understanding of how the learning algorithm works and the hyperparameter's influence on this process while automatic selection may require less knowledge about the algorithm but more computational resources.

### 2.5.1 Manual Hyperparameter Tuning

To achieve good generalization of the model, the primary goal in manual hyperparameter tuning is to adjust the effective capacity of the model according to the complexity of the application. The effective capacity of the model describes the ability of the model to capture increasingly complex relationships between the representation and the target. It is constrained by the representational capacity of the model (what relationships could be captured), the learning algorithm's ability to minimize the cost function (if the model is trained in a suboptimal way it may not utilize its full capabilities) as well as regularization. The generalization error as a function of the hyperparameters typically follows a U-shaped curve [1]. On one end of the spectrum, the effective capacity is very low, which leads to underfitting and hence a high training error as well as a high generalization error. On the other end of the spectrum, the effective capacity is too high, resulting in overfitting. This shows in a low training error but a high generalization error. Some hyperparameters cannot explore the entire U-shape of the curve. Discrete hyperparameters, for example, can only capture a finite set of points. Other hyperparameters may have boundaries preventing them, such as regularization hyperparameters, which can usually only substract capacity and are therefore likely to be bounded in (at least) one direction. The learning rate is a hyperparameter of particular interest, as the *training* error is a U-shaped function of the learning rate. When too high, it can even increase the training error and values that are too small can slow training down significantly or lead to stagnation of the training error. It may therefore be the most important hyperparameter with respect to finding a suitable value [1]. Usually, a large, well regularized model is among the most performant ones [1]. When the training error is low, another approach to achieve good generalization is to use more training data. This approach is more of a brute-force nature and comes with an increase in computational expenses.

### 2.5.2 Automatic Hyperparameter Tuning

Ideally, the deep learning algorithm receives data and computes a desired output without any manual tuning. Neural networks, however, often have a lot of hyperparameters and may benefit substantially from tuning them [1]. In most instances, tuning manually requires expertise and experience, both of which are not always present for every application. Tuning hyperparameters can be viewed as an optimization problem where the generalization error is optimized with respect to the hyperparameters. Algorithms for tuning hyperparameters can have hyperparameters themselves, e. g. search ranges, but these are usually easier to choose as similar choices work well for a wider range of tasks.

*Grid search* is a simple approach where the user selects a finite set of values for each hyperparameter. The model is then trained with every setting of the Cartesian product of these value sets to find the one with the best generalization. Note that as the set for each hyperparameter should be finite, the user needs to provide a discrete selection of sample points also for continuous hyperparameters. The result of using the grid search algorithm can be improved when it is applied repeatedly, where the next grid is placed around the current best setting. A major drawback of grid search are the high computational costs, as they grow exponentially with the number of hyperparameters. This can be lessened to some extent by using parallelization since there is almost no communication needed between different entities evaluating the search grid.

*Random Search* is comparably simple to implement, arguably more convenient to use and converges faster to favorable values [20]. Sample points are chosen according to a predefined probability distribution (e. g. a uniform distribution). Unlike grid search, continuous hyperparameters are not discretized which results in the exploration of a larger set of values without adding computational costs. Furthermore, less experiment runs are wasted [1]. When the influence of two different values for a hyperparameter only differs marginally, the setting is effectively run twice in grid search as the algorithm will evaluate both values with similar values of the remaining hyperparameters. This is unlikey to happen in a practical setting with random search if both values of the hyperparameter in question are evaluated as the remaining hyperparameters are unlikely to be the same.

In practice, the gradient is mostly unavailable due to high computational costs or nondifferentiability of certain components. A derivative-free optimization approach is *MBO*, where the generalization error (or value of some performance measure) is modeled w.r.t. the hyperparameters. A new proposal is then made based on optimizing within the model. A common technique for determining the relevance of a new proposal is to consider both the expected generalization error as well as the model's uncertainty around the expectation.

# Chapter 3

# Model-Based Optimization

## 3.1 History of Model-Based Optimization

The following overview of the origins of MBO is drawn primarily from [15].
An early work (and possibly the earliest according to [15] and [66]), incorporating an approach that resembles MBO is [67] using Wiener processes for searching a one-dimensional and unconstrained parameter space. The Wiener process interpolates the data points linearly in a piecewise manner with a quadratic variance model which is zero at each sample [66]. Kushner discussed the weakness of local optimizers (in particular gradient-based optimization algorithms) regarding converging to local optima and argued that it would be desirable to have the algorithm search the entire parameter space for each iteration. Similar to MBO, the approach of [67] proposes that the points are chosen sequentially according to the maximization of an improvement criterion, namely the probability of improvement (see Subsec. 3.5.1), and including a component that incorporates a trade-off resembling the exploration-exploitation trade-off (see Subsec. 3.2). This technique was later extended, e. g., in [68, 69]. A multidimensional MBO approach was published in [70], using linear combinations of Wiener fields and maximizing a critierion similar to expected improvement function (see Subsec. 3.5.2) [15]. A review of this work was published in [71]. Related work was published under the name *Kriging* (see Sec. 3.4.1). The technique, which was further advanced and primarily made popular by [72] (according to [15]), is named after the South African geologist (and student at the time of conducting his research) D. G. Krige who developed the technique [73]. Its goal is interpolating a random field using a linear predictor, the errors are modelled with a Gaussian process. Krige's work "formed the foundation for an entire field of study now known as geostatistics" [66] (see, e. g., [74, 75]). In the late 1980s, Kriging was applied to deterministic computer experiments [76, 77]. The resulting approach was named Design and Analysis of Computer Experiments (DACE). Specific techniques used in DACE deviated from the geostatistical approach in concentrating on a "small" part of the knowledge available [66]. MBO using Gaussian processes has been applied in the discipline of experimental design (see Sec. 3.6), for example by Jones et al. [30] who proposed a SMBO algorithm called Efficient Global Optimization (EGO) (see

Sec. 3.6). Several consistency proofs for variants of this algorithm exist ([78, 79]).

## 3.2 General Framework

MBO or Bayesian optimization is a powerful strategy for optimizing black-box functions that is very efficient in terms of required function evaluations (see [15]). Let $f : X \to \mathbb{R}$ denote a black-box function, where the input space is $X = X_1 \times X_2 \times ... \times X_d$. For $i \in \{1, ..., d\}$, $X_i$ can either represent the domain of a categorical variable with finitely many values $X_i = \{v_{i_1}, ..., v_{i_t}\}$, $t \in \mathbb{N}$, or the domain of a bounded numeric variable $X_i = [l_i, u_i]$, $l_i \in \mathbb{R}, u_i \in \mathbb{R}$ and $l_i < u_i$. Since $f$ should be optimized, we can assume without loss of generality that $f$ should be minimized, so essentially the goal is to find $x^*$ such that

$$x^* = \underset{x \in X}{\arg\min}\, f(x). \tag{3.1}$$

We assume that $f$ is expensive to evaluate, limiting the number of function evaluations by a budget.

Let $D_s = \{x_i, f(x_i) \,|\, i \in \{1, ..., s\}\}$ for $s \in \mathbb{N}$. In MBO, the knowledge of $D_s$ is incorporated using Bayes' theorem (which explains the term Bayesian optimization) to improve the assumed distribution over the space of objective functions:

$$\mathrm{P}(f|D_s) \propto \mathrm{P}(D_s|f)\mathrm{P}(f). \tag{3.2}$$

$P(f)$ is the *prior*, representing the belief about the space of objective functions. $\mathrm{P}(D_s|f)$ is the *likelihood*, representing how likely the present data is given the belief about the function space. $\mathrm{P}(f|D_s)$ is the *posterior*, essentially representing the updated belief about function space after observing $D_s$. Every update, prior knowledge and evidence is used to maximize the posterior, so each evaluation decreases the distance between the true optimum of $f$ and the expected optimum of $f$ using the posterior $\mathrm{P}(f|D_s)$ [15]. An informative prior can be used to express belief or knowledge about attributes of the objective function like the most likely locations of the maximum, even without actually knowing the function [15].

In practice, the generic SMBO algorithm looks as follows (see, e. g., [16]):
In the beginning, an initial design is generated by sampling points from $X$ and evaluating $f$ at these points. In each iteration, the objective function is (cheaply) estimated with a *surrogate model* (see Sec. 3.4). An *acquisition function* (see Sec. 3.5) is then used to efficiently determine the next sample location, as can be seen in Algorithm 1. The acquisition function is defined on $X$ and utilizes the surrogate model. It represents a trade-off between exploration (sample points in areas with high uncertainty about $f$) and exploitation (sample points at or near the expected optimum of $f$), for example by taking the mean and the variance of the surrogate model into account. The acquisition function is usually easy to optimize (see [15]) and its optimum gives the point of the next evaluation of the

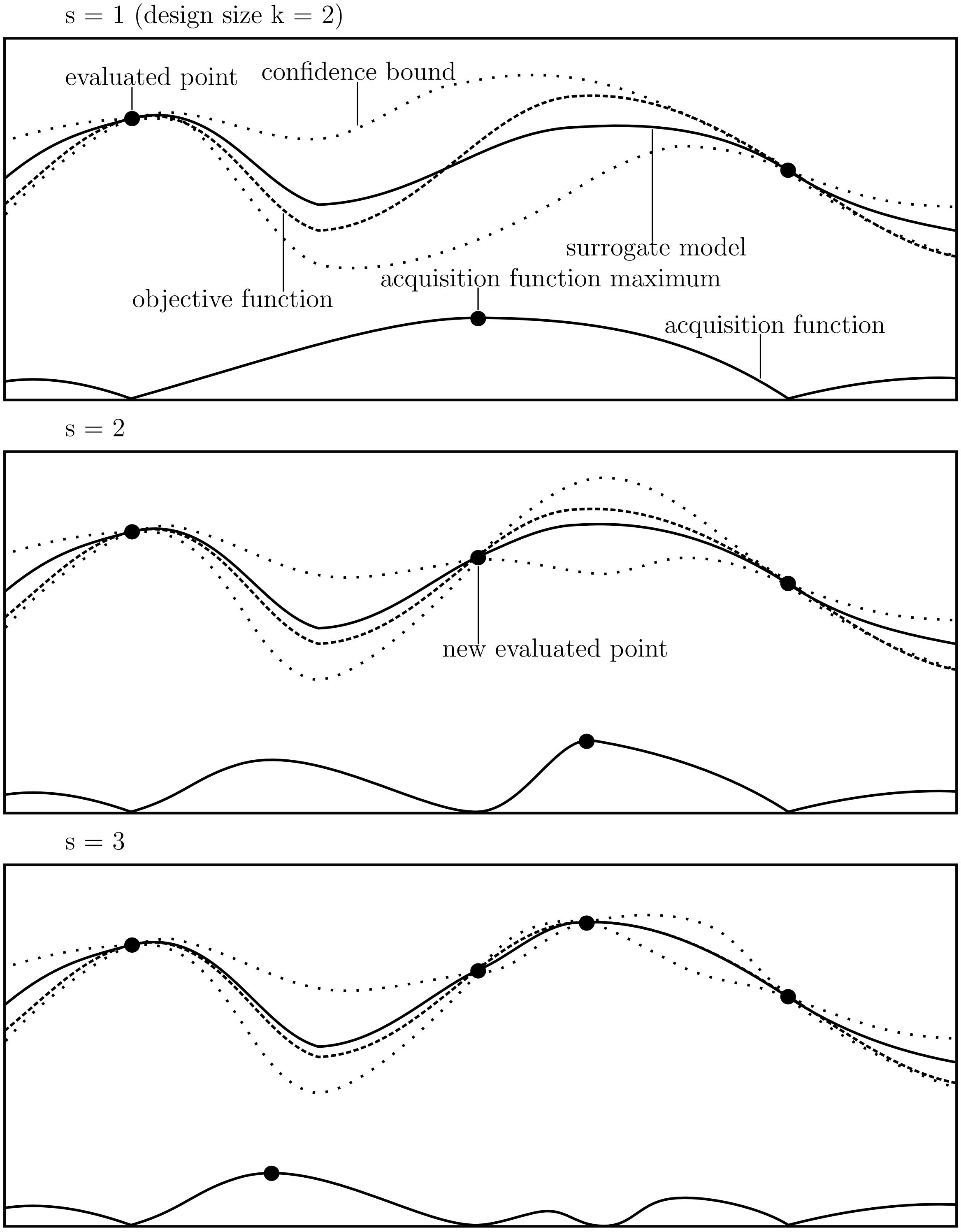


Figure 3.1: Example of SMBO iterations $s = 1, 2, 3$ with design size $k = 2$.

**Algorithm 1** Sequential Model-Based Optimization.

```
Generate initial design $D_0 = \{(x_i, f(x_i)) \,|\, i \in \{1, ..., k\}, x_i \in X\}$, $k \in \mathbb{N}$.
for $s = 1, 2, ...$ do
    Fit a regression model (the *surrogate model*, see Sec. 3.4) to $D_{s-1}$.
    Use the model to determine $x_s$ (often by optimizing an *acquisition function*, see Sec. 3.5).
    Evaluate $f$ at $x_s$ to obtain $f(x_s)$.
    Define $D_s = D_{s-1} \cup \{x_s\}$.
```

objective function. See Fig. 3.1 for an example. The SMBO algorithm terminates when a predefined termination criterion is fulfilled. This can, for example, be a total number of evaluations of $f$ or SMBO iterations, a time budget for evaluations of $f$, or reaching a certain objective value. SMBO is modular and the generic approach can be extended in various ways, e. g. optimizing an objective function $f$ with multi-dimensional output [80] [16] or proposing multiple points for one iteration of $s$ in Algorithm 1 [81, 82].

## 3.3 Initial Design

The surrogate models are fitted iteratively after new evaluations of $f$. To fit an initial surrogate model (for $s = 0$ in Algorithm 1), an initial design $\{(x_1, f(x_1)), ..., (x_k, f(x_k))\}$ consisting of sampled values of $X$ and the corresponding evaluated values of $f$ is created. Choosing the initial design may affect the optimization process, as for example a large initial design may result in an undesired amount of computing time or function evaluations for points that are not part of the actual optimization process (in the sense that they do not optimize the acquisition function at some stage). Conversely, a small design may result in a poor fit of the surrogate model and consequently suboptimal proposals for the next points to evaluate. This may also occur if the design does not cover $X$ appropriately. One way to generate an initial design is to use space-filling Latin Hypercube Designs [83], which ensures that all portions of the range are represented adequately for every input variable. Additionally, they are computationally cheap to generate and can handle many input variables [76]. Possible choices for the size of the design are $2 * d$ and $4 * d$, where $d$ denotes the number of tuned hyperparameters.

## 3.4 Surrogate Model

Surrogate models provide a functional relationship to the objective function while being computationally favorable (see [66]). A deciding factor for the choice of the surrogate model is the structure of the input space $X$. For input spaces with exclusively numeric variables ($X \subset \mathbb{R}$), Kriging (see Sec. 3.4.1 and Sec. 3.6) provides state-of-the-art performance and is hence recommended [16]. In practice, the parameter space rarely consists solely of numeric parameters, but often contains categorical parameters as well as hierarchical dependencies resulting in conditional parameters. For search spaces that include categorical variables, random forests [43] can be used to model $f$ without encoding categorical variables as (discrete) numeric variables. The Kriging approach can be extended for hierarchical input spaces by utilizing suitable kernel functions for both numeric and categorical variables [31], but this is still an ongoing process of research [32]. However, random forests are again a viable alternative. Hereafter, let $\hat{f}$ denote the surrogate model.

### 3.4.1 Kriging and the Gaussian Process

Originally developed by Krige [73] for analyzing mining data using statistics-based techniques, Kriging has provided a basis for and been used in geostatistics since the 1950s [66, 15]. Detailed introductions and examinations can be found in [84, 66, 85]. An alternative introduction to Kriging is given in [86]. While limited to a numeric-only input domain, Kriging is still arguably the most popular choice for surrogate models as it provides both flexibility and a local uncertainty estimator [16]. Assume we have observations $\{(x_i, y_i) \,|\, i \in \{1, ..., k\}\}$ consisting of input $x_i$ and corresponding output $y_i = y(x_i)$ and that their relationship is given by

$$y(x) = f(x) + \epsilon. \tag{3.3}$$

In contrast to many machine learning models, the usual assumption in Kriging is that the residuals are not independent and identically distributed $\epsilon \overset{iid}{\sim} \mathcal{N}(0, \sigma_\epsilon^2)$ but a function of $x$. They are assumed to be spatially correlated in the sense that the if error $\epsilon_i$ for sample point $x_i$ is high, the errors for sample points near $x_i$ are expected to be high as well. Therefore, in its essence, Kriging combines linear regression with a stochastic model fitted to the resulting residuals, namely $\epsilon(x) \sim \mathcal{N}(0, \sigma^2(x))$. The type of linear regression model is determined by the type of Kriging method used and can range from the zero function (*simple Kriging*) to a polynomial function (*universal Kriging*) and other models.

The GP is an extension of the multivariate Gaussian distribution. It is a stochastic process with infinite dimensions where any finite subselection of dimensions follows a multivariate Gaussian distribution [15]. A GP can be viewed as a distribution over functions, analogous to a Gaussian distribution being a distribution of a random variable over its possible values. Furthermore, just as a Gaussian distribution is specified by its mean and covariance, a GP is specified by its mean function and covariance function. Intuitively, it can be useful to think of a GP as a function, which returns the mean and variance of a Gaussian distribution over the possible values for the input value $x$ [15].

While Kriging and MBO are closely related, there are some key differences in practice regarding the way models are fitted (see, e. g., [15, 66]). However, the term Kriging often refers to MBO with a GP surrogate model. For a more in-depth introduction to Kriging see, e. g., [86].

### 3.4.2 Random Forest

A random forest (RF) for regression is a model which consists of a large collection of regression trees [87]. Regression trees are similar to decision trees but return a numeric value rather than a categorical value [43]. A prediction value is produced by aggregating the individual results in a suitable fashion. Regression trees can capture complex interactions in the data and have relatively low bias if grown sufficiently deep [87]. As the covariates are selected randomly in the tree-growing process, the correlation between the trees is reduced without increasing the variance too much. Regression trees can handle categorical input variables well. This property transfers to RFs, which, in addition, usually yield more accurate predictions than regression trees [43].
An algorithm of a RF for regression can be found in [87]:

1. For $b = 1$ to $B$:
   (a) Draw a bootstrap sample $Z^*$ of size $N$ from the training data.
   (b) Grow a random forest tree $T_b$ to the bootstrapped data by recursively repeating the following steps for each terminal node of the tree until the minimum node size $n_{min}$ is reached.
      i. Select $m$ covariates at random from the $p$ covariates.
      ii. Pick the best covariate split-point among the $m$.
      iii. Split the node into two child nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

A prediction at a new point $x$ is made via aggregation:

$$\hat{f}_{\mathrm{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x). \tag{3.4}$$

Note that in order to use certain acquisition functions (see for example Subsec. 3.5.2 and Subsec. 3.5.3), an uncertainty estimate needs to be computed. While this can be achieved in multiple ways, the jackknife estimator works reliably according to [16]. As the random forest is not a spatial model and does not come with a native uncertainty estimate out-of-the-box, the computed estimate is less intuitive in contrast to the Gaussian Process.

## 3.5 Acquisition Function

The purpose of the acquisition function $u$, also called infill criterion, guides the optimization process in the sense that for a given surrogate model, its optimum is proposed as the next evaluation point for the objective function $f$. Therefore, the desired point is either

$$\arg\max_{x \in X} u(x|D_s), \tag{3.5}$$

or

$$\arg\min_{x \in X} u(x|D_s), \tag{3.6}$$

depending on whether the acquisition function $u$ is maximized or minimized. Conditioning on $D_s$ refers to $\hat{f}(x|D_s)$, which is the surrogate model fitted to the available evaluations $D_s$ of $f$. The acquisition function typically also contains some trade-off between exploration, sampling points in regions with high uncertainty, and exploitation, sampling points in regions with favorable values of the surrogate model, by combining the posterior mean $\hat{\mu}(x|D_s)$ and the posterior deviation $\hat{s}(x|D_s)$ or the posterior variance $\hat{s}^2(x|D_s)$ in a suitable formula. Under the assumption that high values of $\hat{s}(x|D_s)$ suggest that few of the evaluated points lie in or close to the region of concern, points with high values of $\hat{s}(x|D_s)$ are of interest. Additionally, these points should have low $\hat{\mu}(x|D_s)$ when minimizing and high $\hat{\mu}(x|D_s)$ when maximizing the acquisition function (see [16]). As there are several acquisition functions, it can be unclear which one to use. Broch et al. [88] propose a method of utility selection where a portfolio of acquisition functions combined with a multi-armed bandit strategy is used instead of a single acquisition function.

### 3.5.1 Probability of Improvement

Kushner [67] suggested using the probability of improvement (PI) as the acquisition function. Note that in this case, the acquisition function should be maximized. For a maximization of the objective function and a GP as a surrogate model the probability of improvement is

$$\begin{aligned} \mathrm{PI}(x) &= \mathrm{P}(f(x) \geq y_{\max}^{(s)}) \qquad (3.7) \\ &= \Phi\left(\frac{\hat{\mu}(x|D_s) - y_{\max}^{(s)}}{\hat{s}(x|D_s)}\right), \qquad (3.8) \end{aligned}$$

with $y_{\max}^{(s)} = \max\{f(x_i) \,|\, (x_i, f(x_i)) \in D_s\}$ the best evaluated value observed so far and $\Phi(\cdot)$ the cumulative distribution function of the Gaussian distribution. However, this formulation puts great emphasis on exploitation. Points that potentially result in a high improvement but with a high uncertainty are unfavored in comparison to points that offer little improvement with high certainty. To regulate exploration and exploitation, the trade-off parameter $\xi \geq 0$ can be added (see [15]):

$$\mathrm{PI}(x) = \mathrm{P}(f(x) \geq y_{\max}^{(s)} + \xi). \tag{3.9}$$

In this setting, greater values of $\xi$ put greater emphasize on exploration and vice versa. However, as Jones [86] points out, the emphasis on exploration or exploitation is highly sensitive to the choice of $\xi$, therefore choosing a suitable $\xi$ may be a difficult task. For a more detailed analysis of the probability of improvement, please refer to [86].

### 3.5.2 Expected Improvement

A popular (and arguably the most popular according to [16]) acquisition function is the expected improvement (EI). In contrast to the PI, the EI criterion takes the magnitude of the potential improvement into account. The expected improvement can be defined as

$$\mathrm{EI}(x) = \mathrm{E}(I(x)), \tag{3.10}$$

where the random variable $I(x)$ is the potential improvement at $x$ over the current optimum. In the case of minimizing the objective function this is

$$I(x) = \max\{y_{\min}^{(s)} - Y(x|D_s), 0\}, \tag{3.11}$$

where $y_{\min}^{(s)} = \min\{f(x_i) \,|\, (x_i, f(x_i)) \in D_s\}$ and $Y(x|D_s)$ is a random variable representing the posterior distribution at x. When using a Gaussian process, $Y(x|D_s)$ follows a Gaussian distribution, $Y(x|D_s) \sim N(\hat{\mu}(x|D_s), \hat{s}^2(x|D_s))$. In this case, let $\Phi$ and $\phi$ be the distribution and the density function of the Gaussian distribution, respectively. Then EI can be expressed as (see [30, 16])

$$\mathrm{EI}(x) = (y_{\min}^{(s)} - Y(x|D_s))\Phi\left(\frac{y_{\min}^{(s)} - \hat{\mu}(x|D_s)}{\hat{s}(x|D_s)}\right) + \hat{s}(x|D_s)\phi\left(\frac{y_{\min}^{(s)} - \hat{\mu}(x|D_s)}{\hat{s}(x|D_s)}\right). \tag{3.12}$$

EI has proven to be a well-balanced and highly effective acquisition function [30] and can even ensure global convergence [89].

### 3.5.3 Confidence Bound Criteria

A pragmatic approach to combine $\hat{\mu}(x|D_s)$ and $\hat{s}(x|D_s)$ is to use a confidence bound as the acquisition function. For minimizing the objective function, the lower confidence bound is

$$\mathrm{LCB}(x) = \hat{\mu}(x|D_s) - \lambda\hat{s}(x|D_s), \tag{3.13}$$

with $\lambda \geq 0$ balancing the trade-off between exploration and exploitation. Higher values of $\lambda$ put higher emphasis on exploration and vice versa. The special case $\lambda = 0$ illustrates the case in which all emphasis is put on exploitation. Analogously, the upper confidence bound for the case of maximizing the objective function is

$$\mathrm{UCB}(x) = \hat{\mu}(x|D_s) + \lambda\hat{s}(x|D_s). \tag{3.14}$$

### 3.5.4 Expected Improvement per Second

The main objective of applying MBO to tuning is to find a good setting of hyperparameters. Acquisition functions like EI are greedy in the sense that at every iteration the next proposal (ideally) yields the best progress in terms of the acquisition function. Snoek et al. [23] argue that most of the time in practice the concern is not to make progress in as few iterations as possible, but rather to make progress in as little time as possible. They therefore suggest to use the expected improvement per second as the acquisition function. Let $c(x)$ denote the duration function that gives the duration of evaluating the objective function $f$ for a set of hyperparameters $x$. Most likely, $c$ is a black-box function as well. Snoek et al. propose to model $\log(c)$ using a Gaussian process. Under the assumption that $c$ and $f$ are independent, one can compute the predicted expected inverse duration easily and hence the expected improvement per second.

### 3.5.5 Optimizing the Acquisition Function

To propose the next evaluation point for $f$, the acquisition function needs to be optimized. One characteristic of the acquisition function $u(\cdot)$ is that samples can be attained cheaply. While still a black-box optimization problem, this drastically simplifies the optimization in contrast to the objective function. Various approaches for optimization exist, like the branch and bound algorithm proposed by Jones et al. [30] or the multistart approach by Lizotte [90]. Another alternative is the *focus search* approach proposed by Bischl et al. [16].

## 3.6 DACE and EGO

DACE was introduced by Sacks et al. [76] and is a discipline focused on solving problems posed by the optimization task of a black-box function [16]. In this context, a computer experiment is defined as running a computer model or code a number of times with various inputs. The output of the code is assumed to be deterministic, i. e., rerunning the model with the same input parameters should yield the same output. For optimization, an efficient predictor should be fitted to the evaluated input values and the corresponding deterministic output to predict the output at untried inputs. To achieve this, the deterministic output is modelled as the realization of a stochastic process. This provides a statistical basis for choosing input parameters of the experiment as well as an estimate of uncertainty for the model's prediction. Indeed, DACE is Kriging (with a specific kind of kernel, see, e. g., [90, 30]) applied to experimental design [15].

The EGO algorithm introduced by Jones et al. [30] is a SMBO approach that combines the DACE model with the EI as acquisition function (see, e. g., [15, 90]). EGO is limited to optimizing noise-free functions with real-valued parameters [43]. Nonetheless, it has proven to be highly effective [16]. Since the publication of EGO, several extensions to the algorithm have been made (see, e. g., [66, 91]).

## 3.7 Extending the Sequential Model-Based Optimization Approach

Tuning the hyperparameters of neural networks is particularly difficult compared to other machine learning models, since the hyperparameter space is high dimensional and may contain both numeric and discrete parameters as well as hierarchies, especially when configuring the architecture. Additionally, training a neural network can have high computational cost. In particular, this may also prove problematic when applying the standard SMBO approach to tuning feedforward neural networks.

To combat these problems, a novel approach is proposed for applying SMBO to tuning a set of feedforward neural network hyperparameters (including architectural decisions like the number of layers). The proposed algorithm, called MBOINL, makes two key changes to the standard SMBO procedure:

First, it eliminates the design and starts the MBO process right away. (Note that in practice there may be a design with a very small number of points to evaluate due to technical reasons.) This aims to avoid possible high costs for exploring the hyperparameter space when using for example a Latin Hypercube design.

Second, the upper bound for the number of possible layers is first set to the lower bound and then iteratively increased until it reaches the actual lower bound, resembling a modified hill climbing approach. In each iteration, SMBO is applied to the new expanded hyperparameter space and the evaluations from previous iterations are provided as the design (yielding no additional computational cost). The aim of this is to learn about the influence of the hyperparameters in situations of comparably little computational costs and transfer this knowledge when exploring possibly more expensive regions of the hyperparameter space, inspired by the idea behind multi-fidelity optimization (see, e. g., [92]). MBOINL is described more formally in Algorithm 2.

**Algorithm 2** Model-Based Optimization with Increasing Number of Layers

```
Let $S$ denote the hyperparameter space to consider for tuning.
Define $l_{lower}$ and $l_{upper}$ as the minimum and maximum number of hidden layers, respectively. In particular, if the possible number of layers is contained in $S$ in dimension $j$, then $S_j = \{l_1, ..., l_k\} \subset \mathbb{N}$ with $k \in \mathbb{N}$ and $l_1 < ... < l_k$ and $l_1 = l_{lower},\ l_k = l_{upper}$.
Allocate a budget to each possible number of layers $\{b_l \,|\, l \in \{l_{lower}, ..., l_{upper}\}\}$.
Define $d_{l_{lower}-1} = \emptyset$.
for $l = l_{lower}, ..., l_{upper}$ do
    Define $\tilde{S} \subset S$ by $\tilde{S}_i = S_i \ \forall i \neq j$ and $\tilde{S}_j = \{l_{lower}, ..., l\}$.
    while $b_l$ not exhausted do
        Do SMBO over $\tilde{S}$ with $d_{l-1}$ as initial (and already evaluated) design.
        Define $d_l$ as the complete tuning run's result containing all evaluated hyperparameter combinations and their performance and including $d_{l-1}$.
Return $x^*$ as the best performing hyperparameter configuration from $d_{l_{upper}}$.
```

# Chapter 4

# Software

## 4.1 Tuning Hyperparameters with mlr and mlrMBO

This section gives an introduction on how to use mlrMBO for hyperparameter tuning in the context of machine learning using the mlr package. One possible approach is to use mlr to train a learner and evaluate its performance for a given hyperparameter configuration in the objective function. Alternatively, we can access mlrMBO's model-based optimization directly using mlr's tuning functionalities. This yields the benefit of integrating hyperparameter tuning with model-based optimization into machine learning experiments without any overhead.

### 4.1.1 mlr

mlr [51] is a R package that provides an interface to a large number of machine learning algorithms and techniques implemented in various R packages. While so far its focus is mainly on classification and regression techniques, there are extensions for survival analysis, clustering and general, example-specific cost sensitive learning. Using mlr yields multiple benefits. It eliminates the need for self-written wrappers when conducting complex experiments, reducing error-proneness and development time. Its modular design allows for high flexibility regarding both the range of tasks that mlr can be applied to as well as the method chosen to solve the respective task: Model resampling, hyperparameter optimization, feature selection, pre- and post-processing and model comparison can all be done with several implemented methods, chained together in a machine learning pipeline and can easily be parallelized. Additionally, the framework is easily extendable, allowing the user to implement self-written methods and plug them in at the desired point in the pipeline. A comprehensive and well structured tutorial on mlr can be found in [93].

### 4.1.2 mlrMBO

mlrMBO [16] is a R toolbox which implements a generic SMBO framework for optimizing expensive black-box functions using MBO. It is highly flexible due to its modular design, for example supporting custom surrogate models via the mlr toolbox. Its main focus in addition to the default SMBO procedure is mixed parameter space optimization, multi-point proposals and multi-objective optimization, offering even combinations of the three domains while achieving state-of-the-art performance in each domain [16]. In addition, it has been sucessfully applied to several use cases (see, e. g., [94, 95, 96]). It supports mixed spaces of integer, numerical, categorical and conditional parameters. Furthermore, it offers parallelization on various parallel backends and optimization of noisy functions. Its features also include easy-to-use visualization, logging and error-handling as well as simple interfaces to extend the package with custom functionality.

### 4.1.3 Custom Objective Function to Evaluate Performance

As an example, we tune the cost and the gamma parameter of a support vector machine (SVM) with an RBF kernel on the popular Iris data set that comes with mlr.
First, we load the required packages. Next, we configure mlr to suppress the learner output to improve output readability. Additionally, we define a global variable giving the number of tuning iterations. Note that this number is set (very) low to reduce runtime.

```r
library(mlrMBO)
library(mlr)

configureMlr(on.learner.warning = "quiet", show.learner.output = FALSE)

iters = 5
```

Now we define the parameter set. Note that the transformations added in the trafo argument mean that we tune the parameters on a logarithmic scale.

```r
par.set = makeParamSet(
  makeNumericParam("cost", -15, 15, trafo = function(x) 2^x),
  makeNumericParam("gamma", -15, 15, trafo = function(x) 2^x)
)
```

Next, we define the objective function. First, we define a learner and set its hyperparameters by using makeLearner. To evaluate its performance we use the resample function which automatically takes care of fitting the model and evaluating it on a test set. In this example, resampling is done using 3-fold cross-validation by passing the ResampleDesc object cv3, which comes predefined with mlr, as an argument to resample. The measure to be optimized can be specified (e.g by passing measures = ber for the *balanced error rate*), however mlr has a default for each task type. For classification the mmce (mean misclassification rate) is the default. We set the arguments minimize = TRUE and has.simple.signature = FALSE of the objective function to ensure correct execution. Note that the iris.task is provided automatically when loading mlr.

```
svm = makeSingleObjectiveFunction(name = "svm.tuning",
  fn = function(x) {
    lrn = makeLearner("classif.svm", par.vals = x)
    resample(lrn, iris.task, cv3, show.info = FALSE)$aggr
  },
  par.set = par.set,
  noisy = TRUE,
  has.simple.signature = FALSE,
  minimize = TRUE
)
```

Now we create a default MBOControl object and tune the RBF SVM. Note that since we do not explicitely pass a design, it will be created automatically.

```
ctrl = makeMBOControl()
ctrl = setMBOControlTermination(ctrl, iters = iters)

res = mbo(svm, control = ctrl, show.info = FALSE)
res$x

## $cost
## [1] 3.05909
##
## $gamma
## [1] -4.141986

res$y

## [1] 0.02
```

### 4.1.4 mlr Tuning Interface

Instead of defining an objective function where the learner's performance is evaluated, we can make use of model-based optimization directly from mlr. We just create a TuneControl object, passing the MBOControl object to it. Then we call tuneParams to tune the hyperparameters.

```
ctrl = makeMBOControl()
ctrl = setMBOControlTermination(ctrl, iters = iters)
tune.ctrl = makeTuneControlMBO(mbo.control = ctrl)
res = tuneParams(makeLearner("classif.svm"), iris.task, cv3,
  par.set = par.set, control = tune.ctrl, show.info = FALSE)
```

```
res$x

## $cost
## [1] 374.8397
##
## $gamma
## [1] 0.0002673261

res$y

## mmce.test.mean
##     0.02666667
```

### 4.1.5 Hierarchical Mixed Space Optimization

In many cases, the hyperparameter space is not just numerical but mixed and often even hierarchical. This can easily be done out-of-the-box and needs no adaption to our previous example as a suitable surrogate model is chosen automatically. To demonstrate this, we tune the cost and the kernel parameter of a SVM. When kernel takes the radial value, gamma needs to be specified. For a polynomial kernel, the degree needs to be specified.

```
par.set = makeParamSet(
  makeDiscreteParam("kernel",
    values = c("radial", "polynomial", "linear")),
  makeNumericParam("cost", -15, 15, trafo = function(x) 2^x),
  makeNumericParam("gamma", -15, 15, trafo = function(x) 2^x,
    requires = quote(kernel == "radial")),
  makeIntegerParam("degree", lower = 1, upper = 4,
    requires = quote(kernel == "polynomial"))
)
```

Now we can just repeat the setup from the previous example and tune the hyperparameters.

```
ctrl = makeMBOControl()
ctrl = setMBOControlTermination(ctrl, iters = iters)
tune.ctrl = makeTuneControlMBO(mbo.control = ctrl)
res = tuneParams(makeLearner("classif.svm"), iris.task, cv3,
  par.set = par.set, control = tune.ctrl, show.info = FALSE)
res$x

## $kernel
## [1] "radial"
##
## $cost
## [1] 3.242954
##
## $gamma
## [1] 0.1873382

res$y

## mmce.test.mean
##     0.02666667
```

### 4.1.6 Parallelization and Multi-Point Proposals

We can easily add multi-point proposals and parallelize the computation using the parallelMap package. In each iteration, we propose as many points as CPUs used for parallelization. We use EI as the infill criterion.

```
library(parallelMap)
ncpus = 2L

ctrl = makeMBOControl(propose.points = ncpus)
ctrl = setMBOControlTermination(ctrl, iters = iters)
ctrl = setMBOControlInfill(ctrl, crit = crit.ei)
ctrl = setMBOControlMultiPoint(ctrl, method = "cl", cl.lie = min)
tune.ctrl = makeTuneControlMBO(mbo.control = ctrl)
parallelStartMulticore(cpus = ncpus)

## Starting parallelization in mode=multicore with cpus=2.

res = tuneParams(makeLearner("classif.svm"), iris.task, cv3,
  par.set = par.set, control = tune.ctrl, show.info = FALSE)

## Mapping in parallel:  mode = multicore; cpus = 2; elements = 16.
## Mapping in parallel:  mode = multicore; cpus = 2; elements = 2.
## Mapping in parallel:  mode = multicore; cpus = 2; elements = 2.
## Mapping in parallel:  mode = multicore; cpus = 2; elements = 2.
## Mapping in parallel:  mode = multicore; cpus = 2; elements = 2.
## Mapping in parallel:  mode = multicore; cpus = 2; elements = 2.

parallelStop()
```

```
## Stopped parallelization.  All cleaned up.

res$x

## $kernel
## [1] "linear"
##
## $cost
## [1] 0.7035836

res$y

## mmce.test.mean
##     0.04666667
```

### 4.1.7 Machine Learning Pipeline Configuration

It is also possible to tune a whole machine learning pipeline, i.e. preprocessing and model configuration. The example pipeline is:

- Feature filtering based on an ANOVA test or covariance, such that between 50% and 100% of the features remain.
- Select either a SVM or a naive Bayes classifier.
- Tune parameters of the selected classifier.

First, we define the parameter space:

```
par.set = makeParamSet(
  makeDiscreteParam("fw.method", values = c("anova.test", "variance")),
  makeNumericParam("fw.perc", lower = 0.5, upper = 1),
  makeDiscreteParam("selected.learner",
    values = c("classif.svm", "classif.naiveBayes")),
  makeNumericParam("classif.svm.cost", -15, 15, trafo = function(x) 2^x,
    require = quote(selected.learner == "classif.svm")),
  makeNumericParam("classif.svm.gamma", -15, 15, trafo = function(x) 2^x,
    requires = quote(classif.svm.kernel == "radial" &
      selected.learner == "classif.svm")),
  makeIntegerParam("classif.svm.degree", lower = 1, upper = 4,
    requires = quote(classif.svm.kernel == "polynomial" &
      selected.learner == "classif.svm")),
  makeDiscreteParam("classif.svm.kernel",
    values = c("radial", "polynomial", "linear"),
    require = quote(selected.learner == "classif.svm"))
)
```

Next, we create the control objects and a suitable learner, combining makeFilterWrapper() with makeModelMultiplexer(). Afterwards, we can run tuneParams and check the results.

```
ctrl = makeMBOControl()
ctrl = setMBOControlTermination(ctrl, iters = iters)
lrn = makeFilterWrapper(makeModelMultiplexer(list("classif.svm",
  "classif.naiveBayes")))
tune.ctrl = makeTuneControlMBO(mbo.control = ctrl)

res = tuneParams(lrn, iris.task, cv3, par.set = par.set,
  control = tune.ctrl, show.info = FALSE)
res$x

## $fw.method
## [1] "variance"
##
## $fw.perc
## [1] 0.9798111
##
## $selected.learner
## [1] "classif.svm"
##
## $classif.svm.cost
## [1] 158.2035
##
## $classif.svm.kernel
## [1] "linear"

res$y

## mmce.test.mean
##           0.02
```

## 4.2 MXNet and mlr

Training deep neural networks and tuning hyperparameters using model-based optimization are tasks that not only need a sound theoretical foundation but also a reliable and efficient codebase. Ideally, one would use state-of-the-art libraries for both endeavours and combine them. The mlr learner classif.mxff integrates a part of the deep learning framework MXNet's functionality for feedforward neural networks, enabling the user to combine the easy-to-use and sophisticated functionalities of mlr with performant deep learning models. It was created because of this thesis' scope and can be regarded as one of its main results.

### 4.2.1 MXNet

MXNet [52] is a multi-language machine learning library focused on deep neural networks (in this regard it can also be referred to as a deep learning framework), designed to be flexible and efficient. It supports both declarative symbolic and imperative programming as well as mixtures of the two. A very simplistic way of describing the difference between those to programming paradigms is that in the former the user describes more abstractly what should be done (useful, e. g., for designing architectures) while in the latter the user specifies more how computations are performed [52]. Currently, it supports interfaces for Python, R, Scala, C++ and Julia. It is very scalable, from mobile devices over CPUs to GPUs offering parallelization with data parallelism or model parallelism. The following code example shows the difference between imperative and symbolic use.

```r
library(mxnet)
data(Sonar, package="mlbench")

Sonar[,61] = as.numeric(Sonar[,61])-1
train.ind = c(1:50, 100:150)
train.x = data.matrix(Sonar[train.ind, 1:60])
train.y = Sonar[train.ind, 61]
test.x = data.matrix(Sonar[-train.ind, 1:60])
test.y = Sonar[-train.ind, 61]

# imperative
mod.imp = mx.mlp(train.x, train.y, hidden_node = 5, out_node = 2,
  out_activation = "softmax", num.round = 3, array.batch.size = 15,
  learning.rate = 0.07, eval.metric = mx.metric.accuracy)

## Start training with 1 devices
## [1] Train-accuracy=0.455555555555556
## [2] Train-accuracy=0.514285714285714
## [3] Train-accuracy=0.514285714285714

# print the first 6 predicted probabilities
predict(mod.imp, test.x)[2, 1:6]

## [1] 0.4915301 0.4914192 0.4914140 0.4914446 0.4914594 0.4914471
```

```
# symbolic
sym = mx.symbol.Variable("data")
sym = mx.symbol.FullyConnected(sym, num_hidden=5)
sym = mx.symbol.Activation(sym, act_type = "relu")
sym = mx.symbol.FullyConnected(sym, num_hidden = 2)
sym = mx.symbol.SoftmaxOutput(sym, name = "sm")
mod.sym = mx.model.FeedForward.create(sym, X = train.x, y = train.y,
  num.round = 3, array.batch.size = 15, learning.rate = 0.07,
  eval.metric=mx.metric.accuracy)

## Start training with 1 devices
## [1] Train-accuracy=0.411111111111111
## [2] Train-accuracy=0.419047619047619
## [3] Train-accuracy=0.495238095238095

# print the first 6 predicted probabilities
predict(mod.sym, test.x)[2, 1:6]

## [1] 0.5109844 0.5109560 0.5109411 0.5109615 0.5109649 0.5109649
```

### 4.2.2 mlr Learner classif.mxff

Creating a custom learner for mlr is designed in a user-friendly fashion and well documented. Three functions have to be defined:

- A learner class which can by generated by mlr's makeLearner. This class contains, among other things, the name of the learner, its parameters and capabilities.
- A function implementing the training process of the learner which can be called by mlr's train function. This function receives data and returns a model.
- A function that takes data and returns the model's predictions, callable by mlr's predict function.

The learners are grouped into different categories: Classification, regression, survival analysis, cost-sensitive classification and multilabel classification. In each category (except for cost-sensitive classification, which constitutes a special case), the learners have the same prefix followed by a dot and the learner's name. The learner created for the purpose of this thesis is a classification learner and integrates the main functionalities of feedforward neural networks of MXNet, resulting in the name classif.mxff. At the time of writing, the learner is in the process of being integrated into the official mlr package. When using the learner, the user has two main options: One option is to use the learner by calling it like any other mlr learner. This does not provide as much flexibility in the architecture, but is more convenient and integrates nicely with all other mlr functionalities like tuning. Another option would be to construct a symbolic architecture with MXNet and pass it to the learner directly. In this case, the majority of hyperparameter settings of the learner is ignored and the symbol is evaluated directly. To illustrate this using an example, we create a feedforward neural network with:

- Two fully connected hidden layers with 5 units each and tanh as activation function.
- An output layer with 3 units and the softmax function as activation layer.

Using the mlr interface would yield:

```
lrn = makeLearner("classif.mxff", layers = 2, num.layer1 = 5, num.layer2 = 5,
  act1 = "tanh", act2 = "tanh", act.out = "softmax")
getHyperPars(lrn)

## $learning.rate
## [1] 0.1
##
## $array.layout
## [1] "rowmajor"
##
## $verbose
## [1] FALSE
##
## $layers
## [1] 2
##
## $num.layer1
## [1] 5
##
## $num.layer2
## [1] 5
##
## $act1
## [1] "tanh"
##
## $act2
## [1] "tanh"
##
## $act.out
## [1] "softmax"
```

Note that we do not have to specify the number of output units, as the learner detects them from the data automatically at the time of training. Considering that tanh and softmax are the default functions (also set at the time of training), the following would be sufficient:

```
lrn = makeLearner("classif.mxff", layers = 2, num.layer1 = 5, num.layer2 = 5)
getHyperPars(lrn)

## $learning.rate
## [1] 0.1
##
## $array.layout
## [1] "rowmajor"
##
## $verbose
## [1] FALSE
##
## $layers
## [1] 2
##
## $num.layer1
## [1] 5
##
## $num.layer2
## [1] 5
```

Passing the symbol would look like this:

```
sym = mxnet::mx.symbol.Variable("data")
sym = mxnet::mx.symbol.FullyConnected(sym, num_hidden = 5)
sym = mxnet::mx.symbol.Activation(sym, act_type = "tanh")
sym = mxnet::mx.symbol.FullyConnected(sym, num_hidden = 5)
sym = mxnet::mx.symbol.Activation(sym, act_type = "tanh")
sym = mxnet::mx.symbol.FullyConnected(sym, num_hidden = 3)
sym = mxnet::mx.symbol.SoftmaxOutput(sym)
lrn = makeLearner("classif.mxff", symbol = sym)
getHyperPars(lrn)

## $learning.rate
## [1] 0.1
##
## $array.layout
## [1] "rowmajor"
##
```

```
## $verbose
## [1] FALSE
##
## $symbol
## C++ object <0x7f88e36d2900> of class 'MXSymbol' <0x7f88e9a12380>
```

At the time of writing, the learner classif.mxff has the following properties:

- It can have up to 3 hidden layers (fully connected or convolutional).
- A fully connected layer cannot be followed by a convolutional layer.
- Each hidden layer can have a separate activation function.
- The activation for the output layer can be specified.
- For each convolutional layer, the kernel size and stride length for the filter and pooling can be chosen individually.
- For each convolutional layer, whether or not to use dilation [97] and, if used, its size can be chosen individually.
- For each convolutional layer, zero padding of individual size can be added after the filter as well as after the pooling.
- For each convolutional layer, the pooling can be chosen individually from maximum, average and sum.
- Dropout can be used. This is possible with a globally defined rate for all layers or choosing the layers to which to apply dropout individually with an individual rate.
- Batch normalization can be applied to all layers or to individual layers.
- Momentum can be used.
- An initializer for the weights can be specified.
- A ratio of the training data (validation ratio) can be chosen to monitor the generalization error during training.
- Alternatively, a new set of data to be used as validation data can be specified.
- The metric for measuring the generalization error can be specified.
- Early stopping can be used.
- A function to be called after every epoch can be specified.

- A function to be called after every batch can be specified.

- The device type, which will be passed to MXNet can be specified. This can, for example, be a single CPU or multiple GPUs.

- The number of epochs for training can be chosen.

- The optimizer for training can be specified.

- The batch size can be specified.

- The key-value store can be specified. (This is basically an object where data is shared when using multiple devices.)

- A symbol for a neural network created in MXNet to be used as the model can be specified.

- The learner can do binary classification as well as multiclass classification.

- The learner can predict either probabilities or one class.

- The learner can handle only numeric input features.

To demonstrate tuning hyperparameters of classif.mxff with mlr and mlrMBO, tuning a very simple set of neural networks on the Iris data will serve as a toy example.

```
lrn = makeLearner("classif.mxff", eval.metric = mx.metric.accuracy,
  num.round=20)
par.set = makeParamSet(
  makeIntegerParam(id = "layers", lower = 1L, upper = 2L),
  makeIntegerParam(id = "num.layer1", lower = 5L, upper = 10L),
  makeIntegerParam(id = "num.layer2", lower = 5L, upper = 10L,
    requires = quote(layers > 1)),
  makeDiscreteParam(id = "act1", c("tanh", "relu", "sigmoid")),
  makeDiscreteParam(id = "act2", c("tanh", "relu", "sigmoid"),
    requires = quote(layers > 1)),
  makeNumericParam(id = "learning.rate", lower = 0.05, upper = 0.3)
)
ctrl = makeMBOControl()
ctrl = setMBOControlTermination(ctrl, iters = 5)
tune.ctrl = makeTuneControlMBO(mbo.control = ctrl)
res = tuneParams(lrn, iris.task, cv3, par.set = par.set,
  control = tune.ctrl, show.info = FALSE)
res$x
```

```
## $layers
## [1] 1
##
## $num.layer1
## [1] 8
##
## $act1
## [1] "tanh"
##
## $learning.rate
## [1] 0.2472132
```

```
res$y
```

```
## mmce.test.mean
##      0.3333333
```

For practicality, the number of filters can be defined in two ways when using a convolutional layer. In some cases, it may be more convenient to use the num.layer parameters, which are also used to specify the number of neurons in a fully connected layer. However, when constructing a parameter set for tuning, it may be desirable to have a different parameter, as the upper limit of neurons and filters may differ. In this case, the num.filter parameters can be used. The following toy parameter set demonstrates this.

```
par.set = makeParamSet(
  # layers parameter only included for comprehensibility
  makeIntegerParam(id = "layers", lower = 1L, upper = 1L),
  makeLogicalParam(id = "conv.layer1"),
  makeIntegerParam(id = "num.layer1", lower = 50L, upper = 300L,
    requires = quote(conv.layer1 == FALSE)),
  makeIntegerParam(id = "num.filter1", lower = 10L, upper = 50L,
    requires = quote(conv.layer1 == TRUE))
)
```

## 4.3 Related Software

**DiceOptim** [98] is an R package that offers an EGO implementation. It was produced in the frame of the DICE (Deep Inside Computer Experiments) consortium which joined major french companies and public institutions with a high research interest in computer experiments with academic researchers from 2006 to 2009, in order to combine industrial problems with academic knowledge advancing research in the field of design and analysis of computer experiments. DiceOptim performs sequential and parallel Kriging-based optimization, using EI with single or multi-point proposals.

**rBayesianOptimization** [99] is also a R package offering an EGO implementation, written in pure R. Both DiceOptim and rBayesianOptimization offer different kernels for the GP, as well as acquisition functions but support only non-conditional variables.

**Spearmint** [23] is a Python package which offers a sophisticated EGO implementation focused on hyperparameter optimization of machine learning algorithms. It is designed in a modular fashion, allowing to choose between different acquisition functions as well as functionalities regarding distribution and management of the experiments. While Spearmint also offers different GP kernels and only supports non-conditional variables, it has been extended to support multi-criteria optimization [100].

**BayesOpt** [101] is an efficient C++ library that contains an extension of EGO to solve sequential experimental design problems. It supports mixed and conditional parameters and provides an interface for C, C++, Python, Matlab and Octave.

**SMAC** (sequential model-based algorithm configuration) [43] is a tool for optimizing algorithm parameters including hyperparameter optimization of machine learning algorithms. As it uses a random forest instead of a GP as a surrogate model, it allows for mixed parameter space optimization. Multi-objective optimization and parallelization are not supported.

**Hyperopt** [22] is a Python library for serial and parallel optimization that uses a tree-structured Parzen estimator instead of regression and supports numerical, categorical and conditional parameters. Although a general tool for black-box optimization, its main focus is machine learning algorithms.

**SPOT** [102] is a R implementation for model-based optimization and tuning of algorithms. It contains multi-objective optimization and offers different modeling techniques and algorithms including the optimization of noisy functions as well as functions with constraints.

# Chapter 5

# Benchmarks

To evaluate the performance of model-based optimization methods for tuning the hyperparameters of neural networks, random search, MBO and MBOINL are applied to tuning feedforward neural network hyperparameters (including the architecture) on CIFAR-10, MNIST rotated with background images and Fashion-MNIST. Additionally, a hand-designed learner is trained and evaluated on the same data sets. Initially, the exact benchmarks and the setup to conduct the benchmarks are explained. Afterwards, the performance of the different tuning algorithms and the hand-designed network is evaluated.

## 5.1 Benchmark Setup

Random search is implemented using mlr functionalities. MBO uses mlrMBO via the mlr interface and MBOINL uses some mlr functionalities and mlrMBO directly. The models trained are all using MXNet via the learner classif.mxff.

The benchmarks are run on a NVIDIA DGX-1, a specialized deep learning system featuring eight Tesla P100 GPU accelerators connected through NVLink, with 16 GB of High-Bandwidth-Memory and 170 TFLOPS performance. The kvstore hyperparameter (specifying the key-value store) is set to local, meaning gradient aggregation and weight updates are not run on the GPUs. This may result in a reduced computational performance. However, some critical memory issues may occur for certain hyperparameter configurations when not using a local key-value store. Every tuning method is run with and without early stopping, resulting in 6 different algorithms. Every tuning run (applying one algorithm to one data set for one time) is allocated a time budget of 4.5 hours. This time budget only includes proposing hyperparameter configurations and evaluating these proposed configurations.

Every algorithm is applied to every data set twice, resulting in a total 162 hours of tuning. All networks are trained using stochastic gradient descent. The benchmarks are run using the R package batchtools [53]. All designs are Latin Hypercube designs For regular MBO, the design size is $2 \cdot d = 64$ where $d$ is the size of the parameter set including the number of epochs. For comparability, MBO with and without early stopping use the same design

in every replication. Therefore, the design size also is 64 when early stopping is used.
For MBOINL a design of size 5 is computed in every stage for technical reasons. Indeed, this design contains at least one configuration with the maximum possible number of layers as convolutional layers. Furthermore, $\frac{1}{7}$ of the time budget is reserved to tuning neural networks with one hidden layer, $\frac{2}{7}$ are allocated to the tuning of networks of depth up to 2 and $\frac{4}{7}$ is for tuning networks of depth 1 to 3. The surrogate model for MBO and MBOINL is a random forest with nodesize 1. The acquisition function is the confidence bound criterion with $\lambda = 2$.

The data is split into training data, test data and validation data with a split of $(\frac{2}{3}, \frac{1}{6}, \frac{1}{6})$. The training data is used for training the model. After the training of a model, its performance is evaluated on the test set. When the tuning is finished, the configuration that had the best performance on the test set is trained on both training and test set and evaluated on the validation set. When using early stopping, the validation.ratio hyperparameter of classif.mxff is set to 0.15. In this case, only 85% of the training data is used for training and 15% is used to monitor the accuracy on a data set not used for training. For one tuning replication (evaluation of all six algorithms) all configurations are passed the same instance of training set, test set and validation set. The performance measure used is *accuracy*, which should be maximized.
In addition to the main benchmark, MBOINL is run twice on CIFAR-10 with the aim to minimize accuracy.
Without early stopping, the number of epochs is a hyperparameter with values between 10 and 50. With early stopping, the number of epochs is set to 50 and the early stopping mechanism causes the training to stop if the accuracy on the in-training validation set has not reached a new maximum for 10 epochs. Due to technical reasons, the model parameters of the moment the early stop occurs are taken to evaluate the model further. Taking the parameters from the epoch where the in-training validation error is the lowest might increase the performance.
Note that an increase of the upper limit of epochs might improve the overall best performance of every algorithm. However, to properly evaluate the quality of the configurations proposed by MBO, the design evaluation should finish early enough to allow sufficient time budget for the actual proposal procedure. This leads to a trade-off between the size and complexity of the architectures considered in the parameter set and the number of epochs. In this benchmark, the aim was to complete the design evaluation after a maximum of approximately two hours, resulting in the settings presented. Apart from the number of epochs, the tuned hyperparameters are:

- Learning rate in $[0.05, 0.3]$.
- Momentum in $[0.7, 0.99]$.
- Whether or not to use dropout (if so, global dropout rate in $[0.2, 0.8]$).
- Whether or not to use batch normalization (for all applicable layers or none).

- Batch size in $\{2^k | k \in \{2, ..., 7\}\}$ *per GPU* (so the actual batch size is in $\{8 \cdot 2^k | k \in \{2, ..., 7\}\}$).
- Number of hidden layers in $\{1, 2, 3\}$, either fully connected or convolutional (recall that for classif.mxff a fully connected hidden layer cannot be followed by a convolutional hidden layer).
- For fully connected layers, number of neurons in $\{50, ..., 250\}$ (separately for each layer).
- For convolutional layers, number of filters in $\{20, ..., 60\}$ (separately for each layer).
- Activation function of each hidden layer either tanh, relu or sigmoid (separately for each layer).

The activation function in the output layer is softmax.
For a convolutional layer, the following additional hyperparameters are present:

- Quadratic filter kernel size in $\{1, ..., 5\}$ (separately for each layer).
- Filter stride $(s_{f_1}, s_{f_2})$ with $s_{f_1}$, $s_{f_2}$ in $\{1, 2, 3\}$ (separately for each layer).
- Quadratic pooling kernel size in $\{1, ..., 5\}$ (separately for each layer).
- Pooling stride $(s_{p_1}, s_{p_2})$ with $s_{p_1}$, $s_{p_2}$ in $\{1, 2, 3\}$ (separately for each layer).

The pooling is always max pooling.

When training the network fails, an accuracy of 0 is imputed. Training may fail, for example, when the kernel and stride values of convolutional layers are chosen so large that at some point the kernel size exceeds the dimension of the data.
The hand-designed network is configured as follows:

- Trained with early stopping (maximum of 50 epochs, stop after no new accuracy maximum for 10 epochs).
- Three hidden layers.
- Hidden layer 1: Convolutional, activation relu, 60 filters, quadratic convolution and pooling kernels of size 3, stride sizes 1.
- Hidden layer 2: Convolutional, activation relu, 40 filters, quadratic convolution and pooling kernels of size 2, stride sizes 1.
- Hidden layer 3: Fully connected, activation relu, 250 neurons.
- Learning rate 0.1.
- Momentum 0.8.

- Batch normalization.
- No dropout.
- Batch size 1024 (128 per GPU).

## 5.2 Evaluation

Figure 5.1 shows all runs on CIFAR-10, including the number of evaluated configurations in each run. Algorithms using early stopping seem to do slightly more evaluations on average. However, the number of evaluated configurations is no indicator for the achieved performance.

Note how early in the tuning process of most random search replications the best performing configuration of the respective replication is found. While no method cleary outperforms the others over the entire tuning time, Figure 5.2 showing the average performance of the runs on CIFAR-10 shows that on average MBO (without early stopping) is better than the others over the whole tuning period. In the beginning of the run, MBO is on average worse than all other methods and seems to match the others around the time where the design evaluation is finished (see Fig. 5.1 for the time design evaluation is finished for each MBO replication). Until that point, MBO with early stopping follows the same trend but is a little ahead, likely due to stopping unpromising configurations earlier (see Fig. 5.1 and recall that MBO and MBO with early stopping receive the same design in every replication). However, it cannot match the performance after the design evaluation. In general, evaluating the design in MBO does not necessarily rule out finding a good configuration early. Random search, both with and without early stopping, is on average the best in the beginning but stagnates afterwards. MBOINL shows high variability concerning the test set performance, as its replications are the second best and the second worst of all runs. The variability of the MBOINL algorithm on CIFAR-10 is also visible in Fig. 5.3, which shows the number of layers in the evaluated configuration during tuning for the best MBO run, the best MBOINL run and both runs of MBOINL with early stopping. MBO converges clearly to three layers with one convolution, MBOINL narrows its search to the area of two to three layers with either one or two convolutions. One replication of MBOINL with early stopping mainly consists of configurations with one or two layers with one convolution and the other of configurations with two or three layers and no convolution. Also, in one of the runs configurations with three layers are barely explored, which happens at the time the small design due to technical reasons is evaluated. On Fashion-MNIST, all algorithms find a rather good performing configuration quickly and reach comparable results on the test set (see Fig. 5.4 and note the limits on the y-axis).

Again, there is a comparably large variability in performance of MBOINL both with and without early stopping. MBO with and without early stopping as well as random search with early stopping reach the best average results on the test set, see Fig. 5.5. On MNIST rotated with background images MBOINL with and without early stopping achieve the best results as shown in Fig. 5.6. MBO with early stopping achieves a slightly worse result

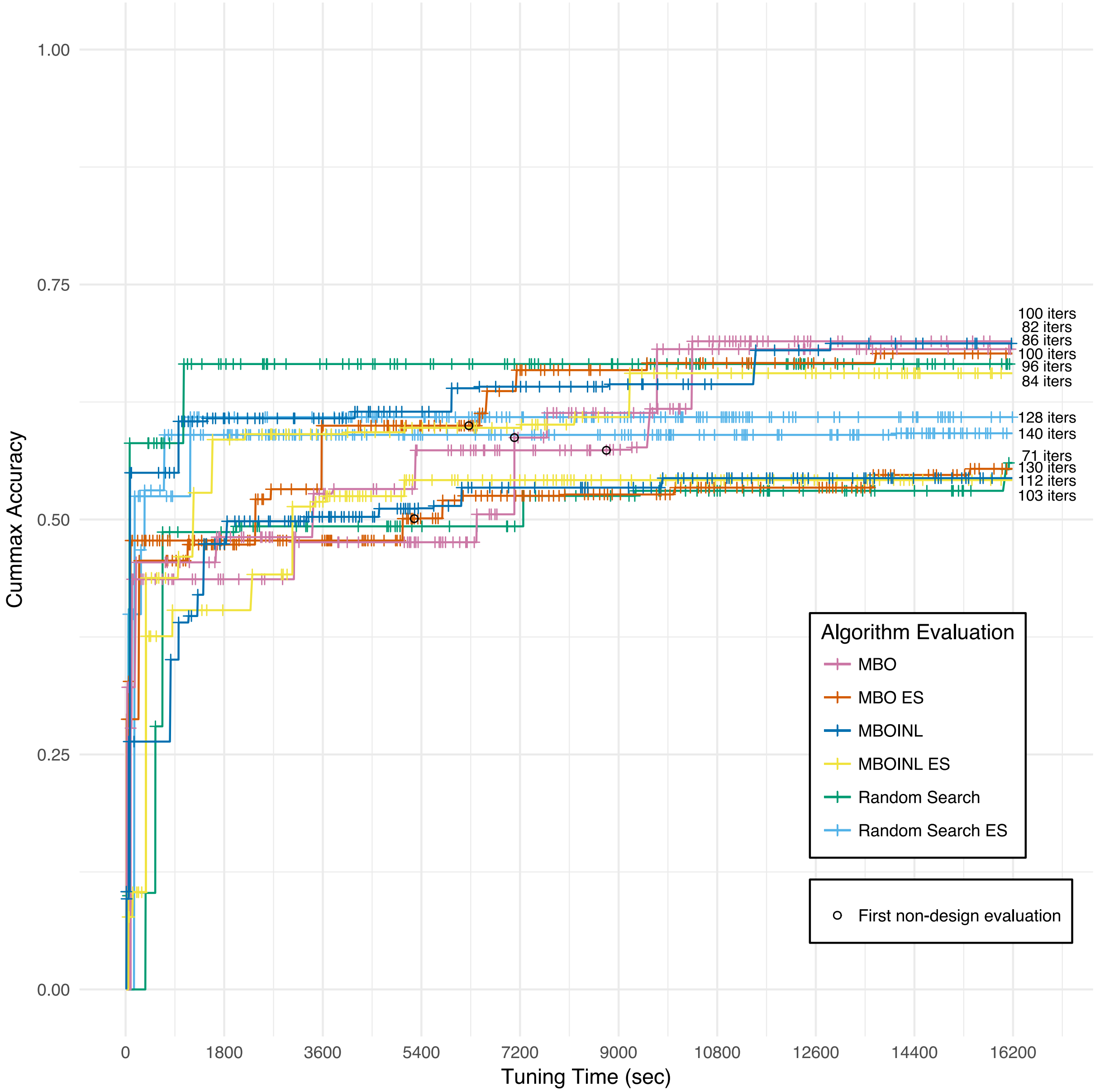


Figure 5.1: Tuning runs on CIFAR-10 showing the best achieved performance on the test set during tuning, the time points of every trained neural network (evaluation of the objective function by the algorithm), as well as the total number of trained neural networks.

but the performance in both replications has low variability. On average, MBOINL with early stopping achieved the best performance (see Fig. 5.7). There is a visible difference in average performance between random search and the other algorithms. Note the high difference between the number of trained networks of the different replications of MBO with early stopping.

Suprisingly, minimizing the accuracy during MBOINL yields good results regarding the

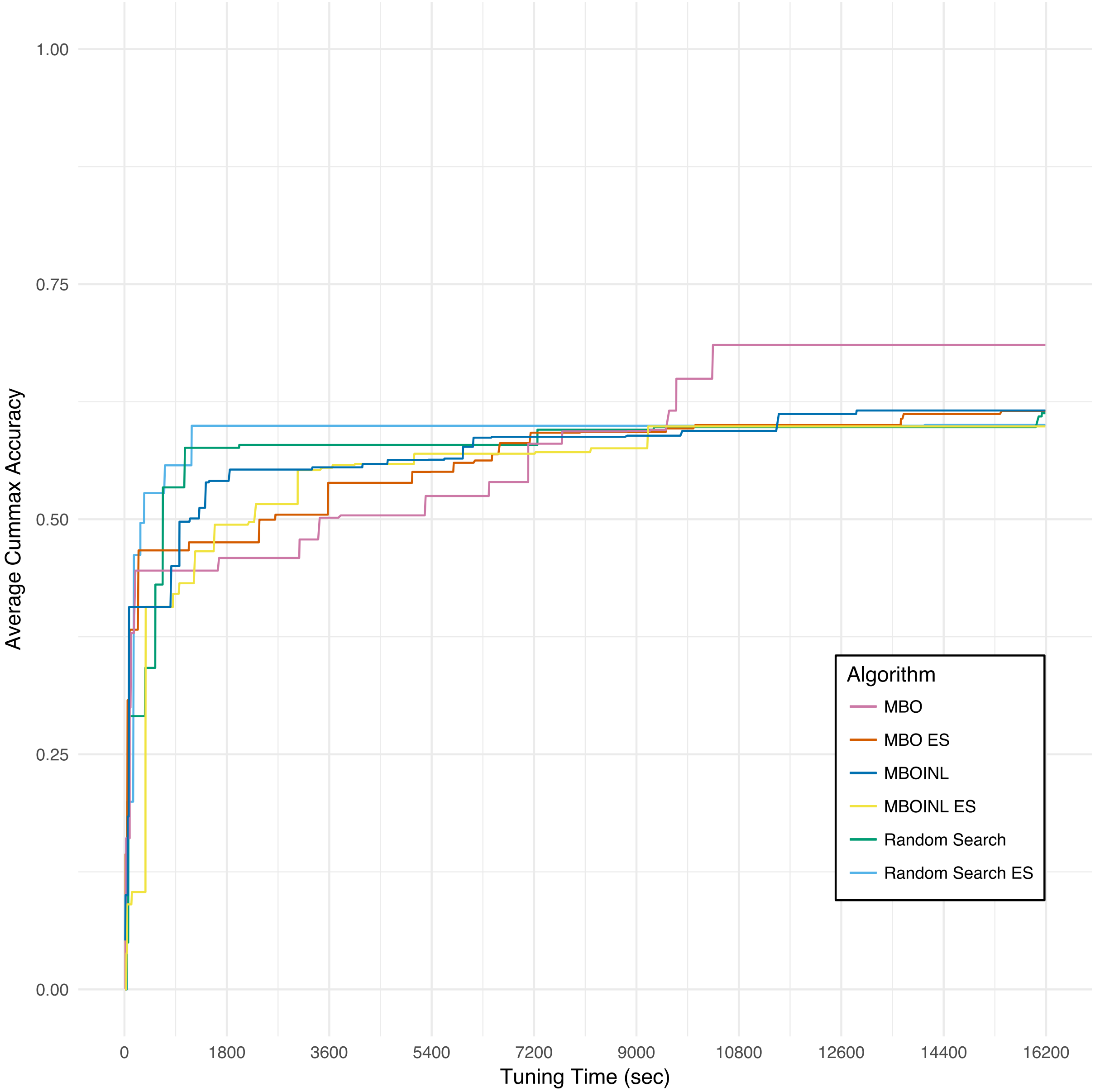


Figure 5.2: Average best achieved performance on the test set during tuning on CIFAR-10.

maximum accuracy performance by any evaluated network configuration. Minimizing the accuracy for MBOINL with early stopping reached an average (over both replications) accuracy of approximately 0.597 on the test set, where maximizing the accuracy reached an average accuracy of 0.599. Maximizing the accuracy for MBOINL with early stopping reached an average accuracy of approximately 0.652 on the test set, where maximizing the accuracy reached an average accuracy of 0.615. This is likely to be due to the fact that the region in the hyperparameter space with two or more convolutions contains a lot of configurations with poor combinations of kernel and stride sizes and hence imputed 0 accuracy.

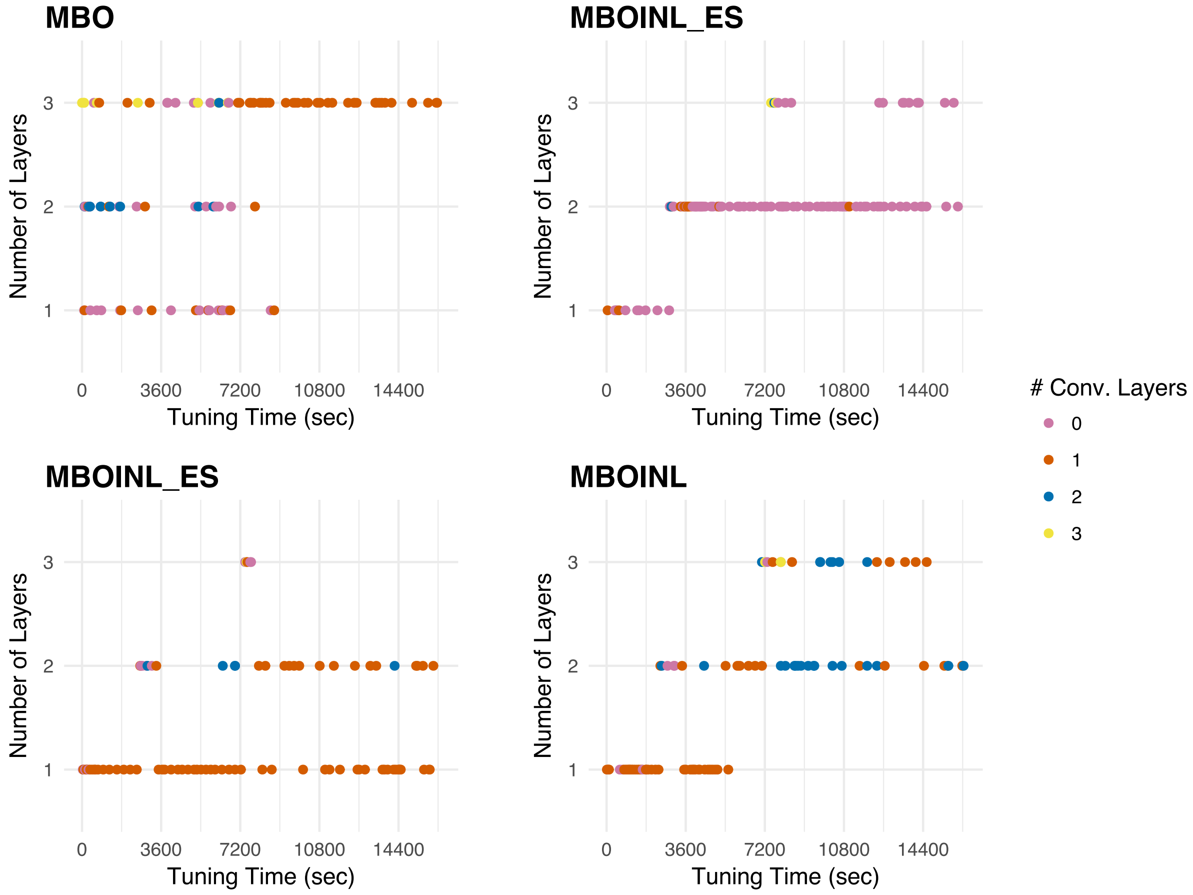


Figure 5.3: Number and kind of layers used as network configuration during training of the best MBO and best MBOINL run, as well as both runs of MBOINL with early stopping on CIFAR-10.

Therefore, MBOINL minimizing the accuracy proposes a large number of configurations located in this area of the hyperparameter space. However, this area also contains configurations performing rather well. Most likely, MBOINL, trying to minimize the accuracy, proposes these configurations accidentally while searching this particular area of the space. This phenomenon showcases the problem of tuning configurations with multiple convolutional layers for this particular hyperparameter set. The problem of exploring the region of the hyperparameter space with two or more convolutions is also visible in Fig. 5.8. In general, too many configurations attain an accuracy of 0 due to imputation because of ill-defined kernel and stride sizes. In consequence, the surrogate learner may not propose exploring this space any further. In most cases, only in a minority of replications, well-performing configurations with two or more convolutional layers are found. Figure 5.9 shows the ranks of the algorithms when comparing performance on the test set averaged over the replications on all data sets over time. A rank of 1 indicates that the algorithm was the best compared to the others. Very notably, both MBO algorithms have the lowest

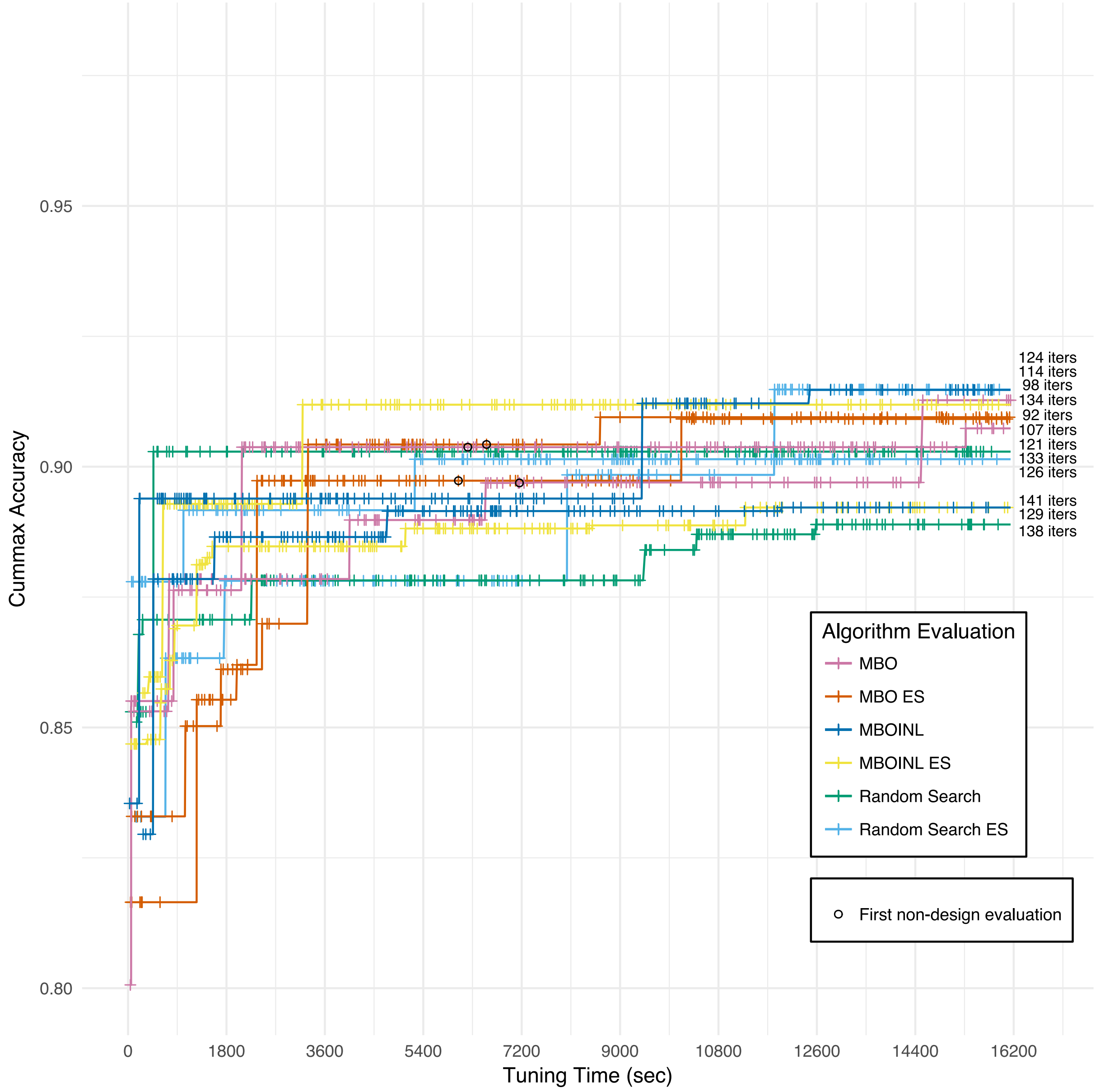


Figure 5.4: Tuning runs on Fashion-MNIST showing the best achieved performance on the test set during tuning, the time points of every trained neural network (evaluation of the objective function by the algorithm), as well as the total number of trained neural networks.

average ranks in the beginning with MBO without early stopping dropping below five at one point but both reach the top two stops at the end of the tuning process. Random search without early stopping reaches a rather high average rank in the beginning but has the lowest spot in the end. Random search has a rather low average rank in the beginning and stays roughly at the same level. Both MBOINL algorithms have rather good ranks in

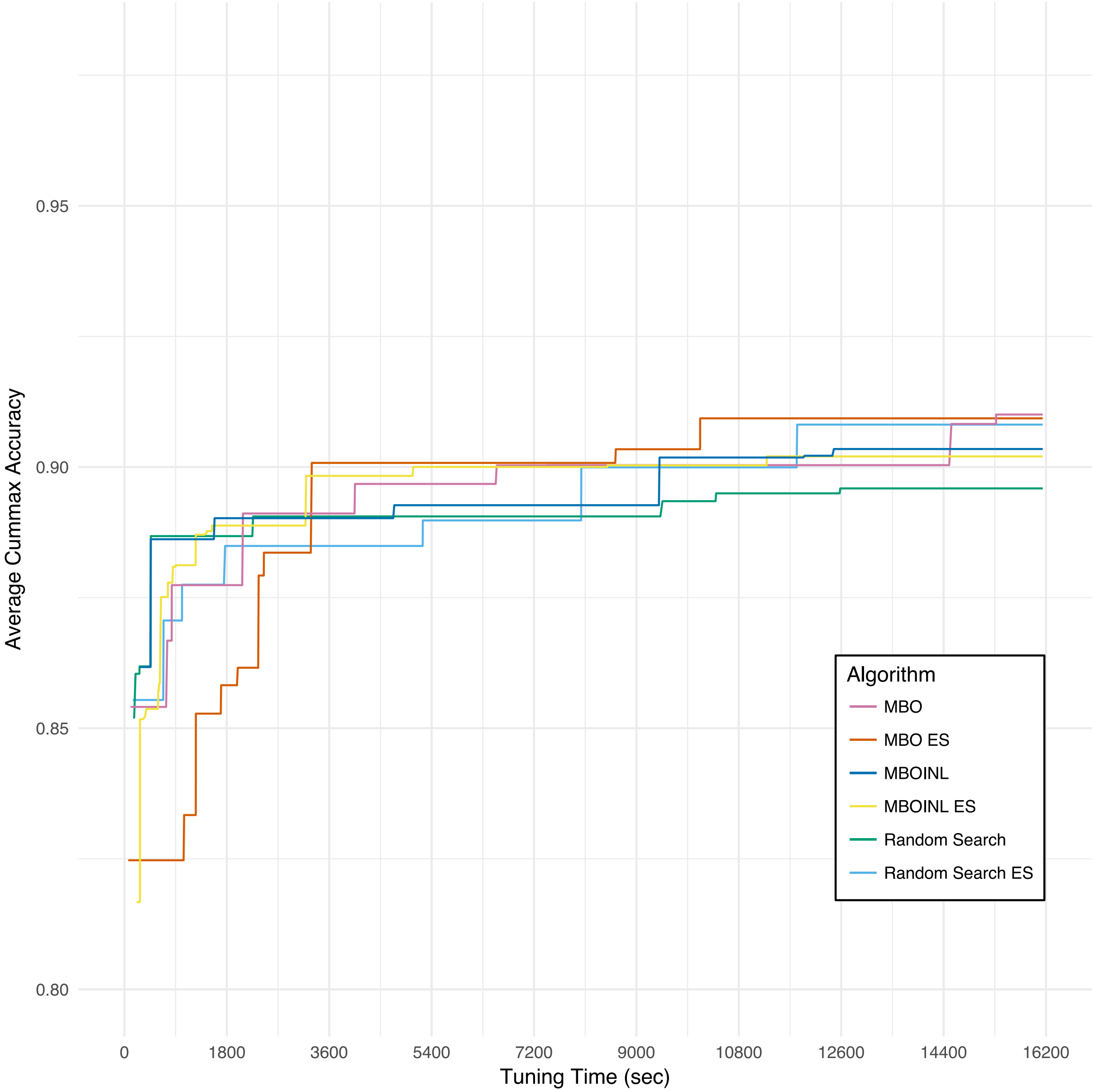


Figure 5.5: Average best achieved performance on the test set during tuning on Fashion-MNIST.

the beginning and reach the average ranks in the end. On the CIFAR-10 validation set, MBO had the best average performance of the algorithms as well as the least variability of performance concerning the different replications, as shown in Fig. 5.10. In contrast, random search without early stopping has the largest variability between replications and the worst average performance on CIFAR-10. On Fashion-MNIST, MBOINL without early stopping followed by MBO and random search both with early stopping have the best average performance of the algorithms. On MNIST rotated with background image, both MBO

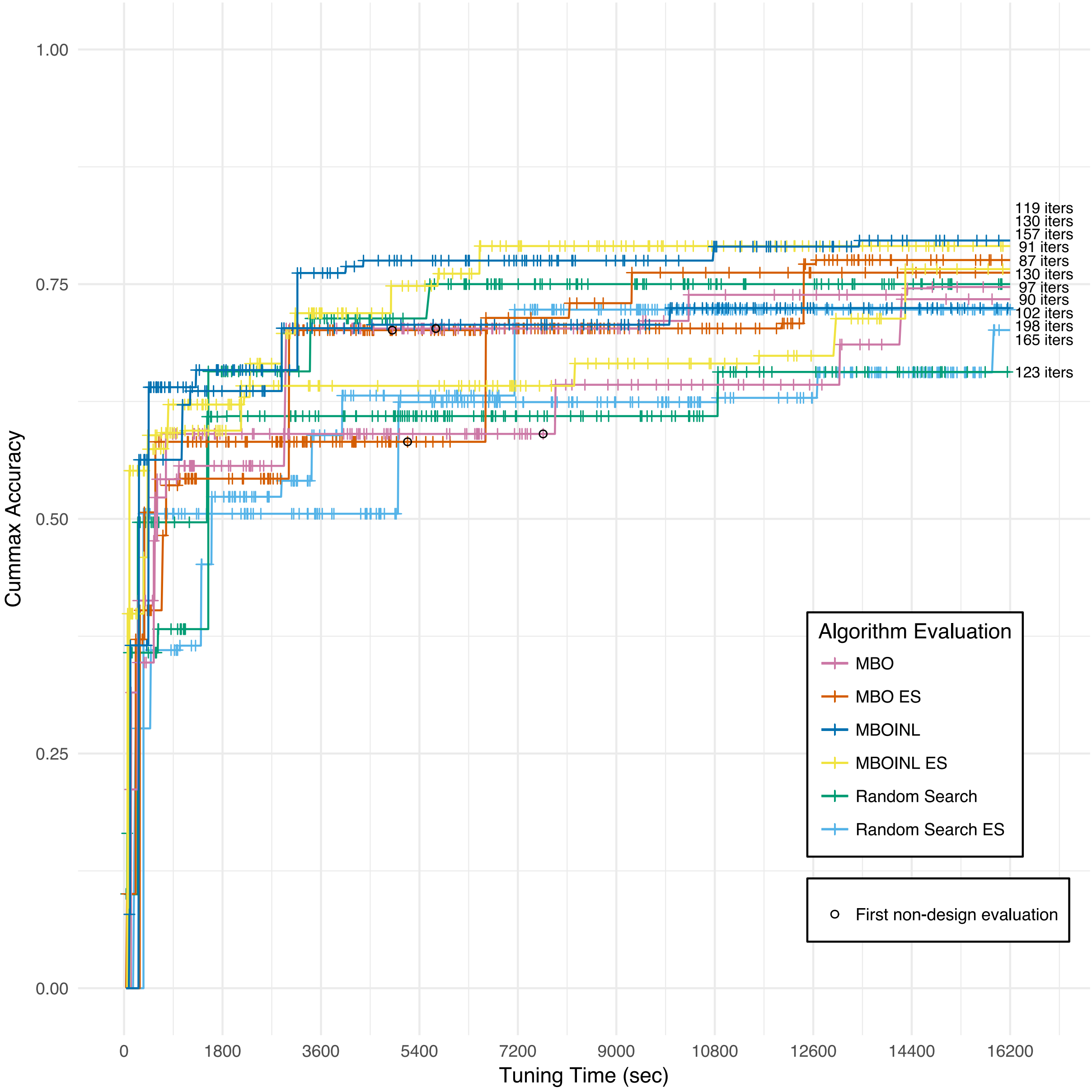


Figure 5.6: Tuning runs on MNIST rotated with background image showing the best achieved performance on the test set during tuning, the time points of every trained neural network (evaluation of the objective function by the algorithm), as well as the total number of trained neural networks.

algorithms have the best average performance. The figures and the table in Fig. 5.10 show that the hand-designed configuration consistently has a better performance than the algorithms average performances. Again, this highlights the problem of searching the region of the hyperparameter space with multiple convolutional layers (recall that the hand-designed configuration has three layers of which the first two are convolutional).

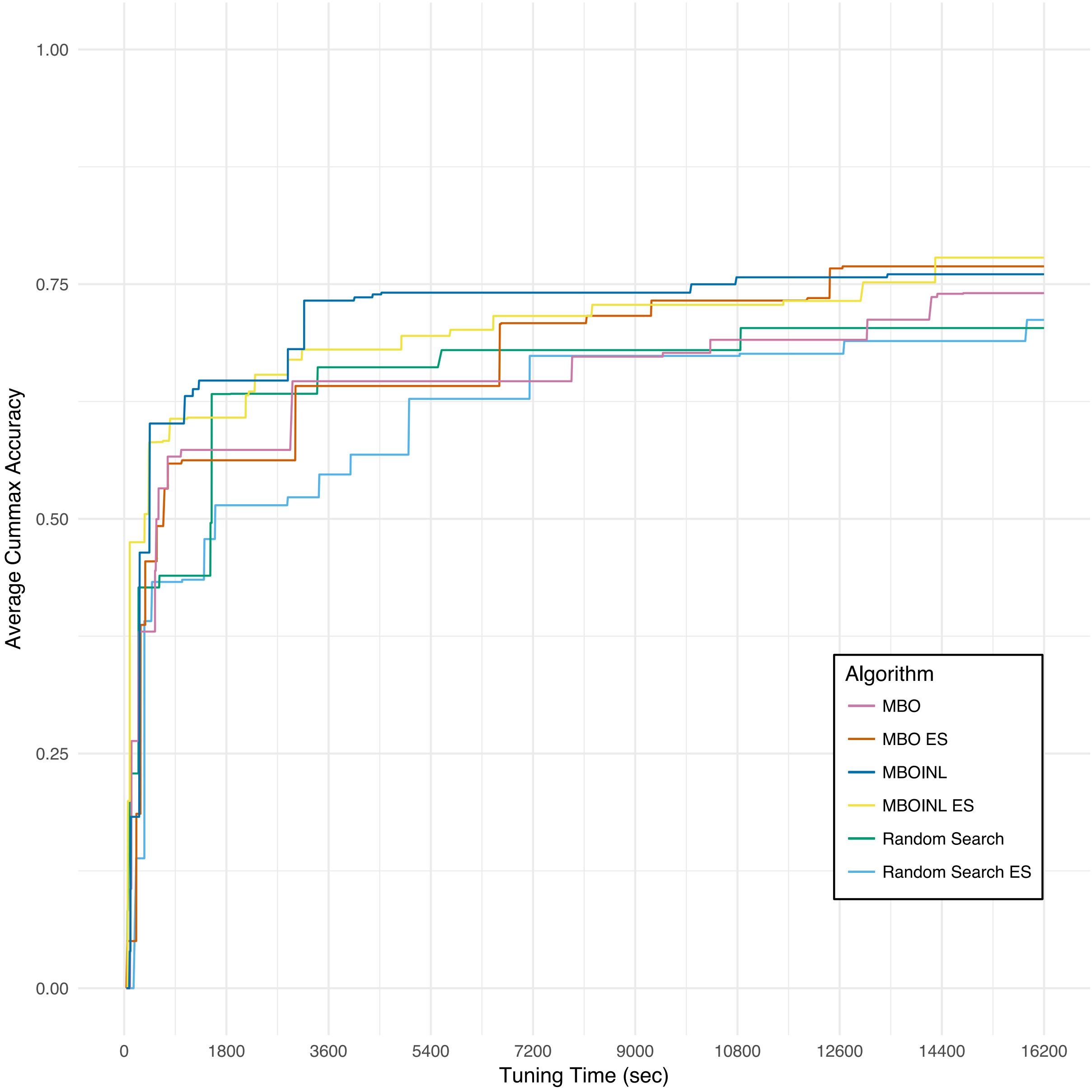


Figure 5.7: Average best achieved performance on the test set during tuning on MNIST rotated with background images.

Both MBO algorithms had the best rank averaged over all replications on all data sets, followed by MBOINL witout early stopping. Random search without early stopping reached the worst average rank.

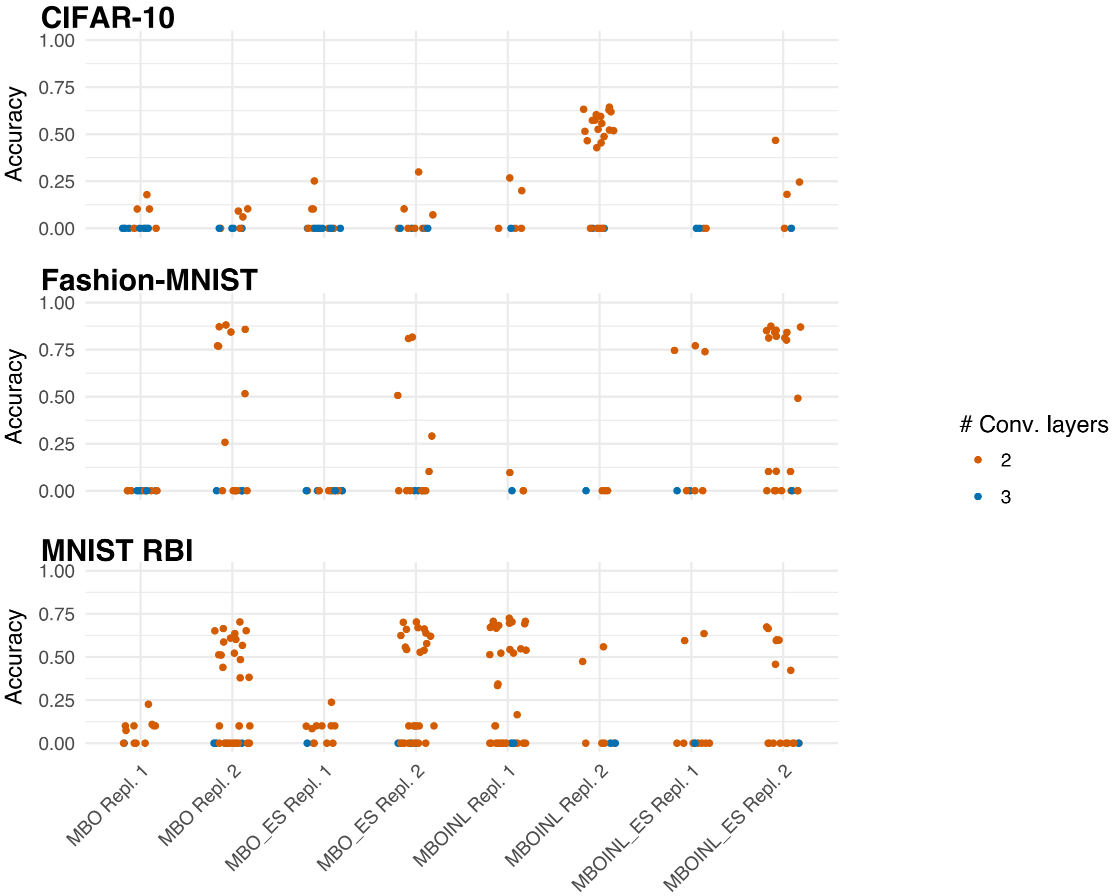


Figure 5.8: Performance of networks with two or three convolutional layers trained during the different replications of MBO and MBOINL both with and without early stopping.

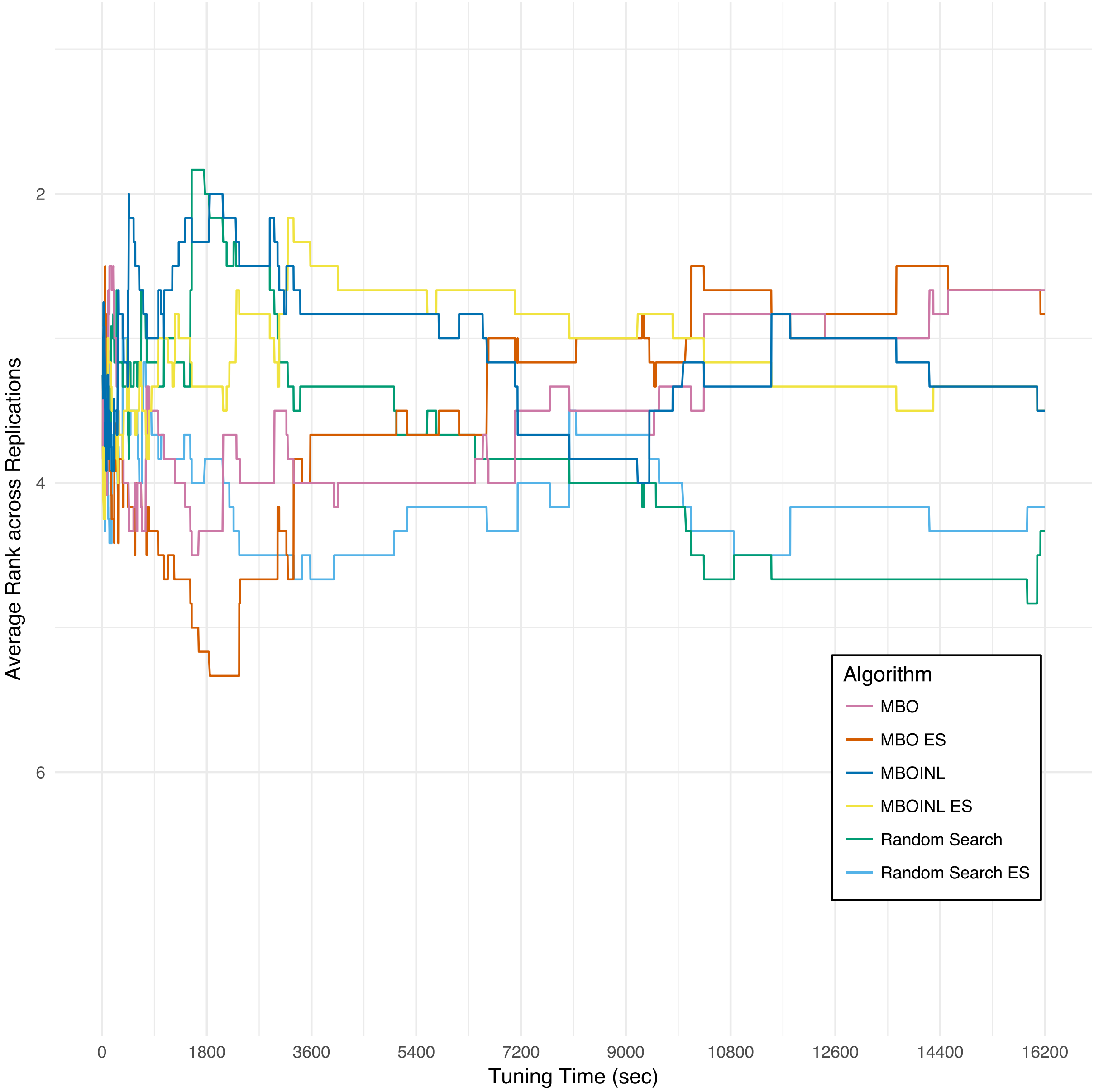
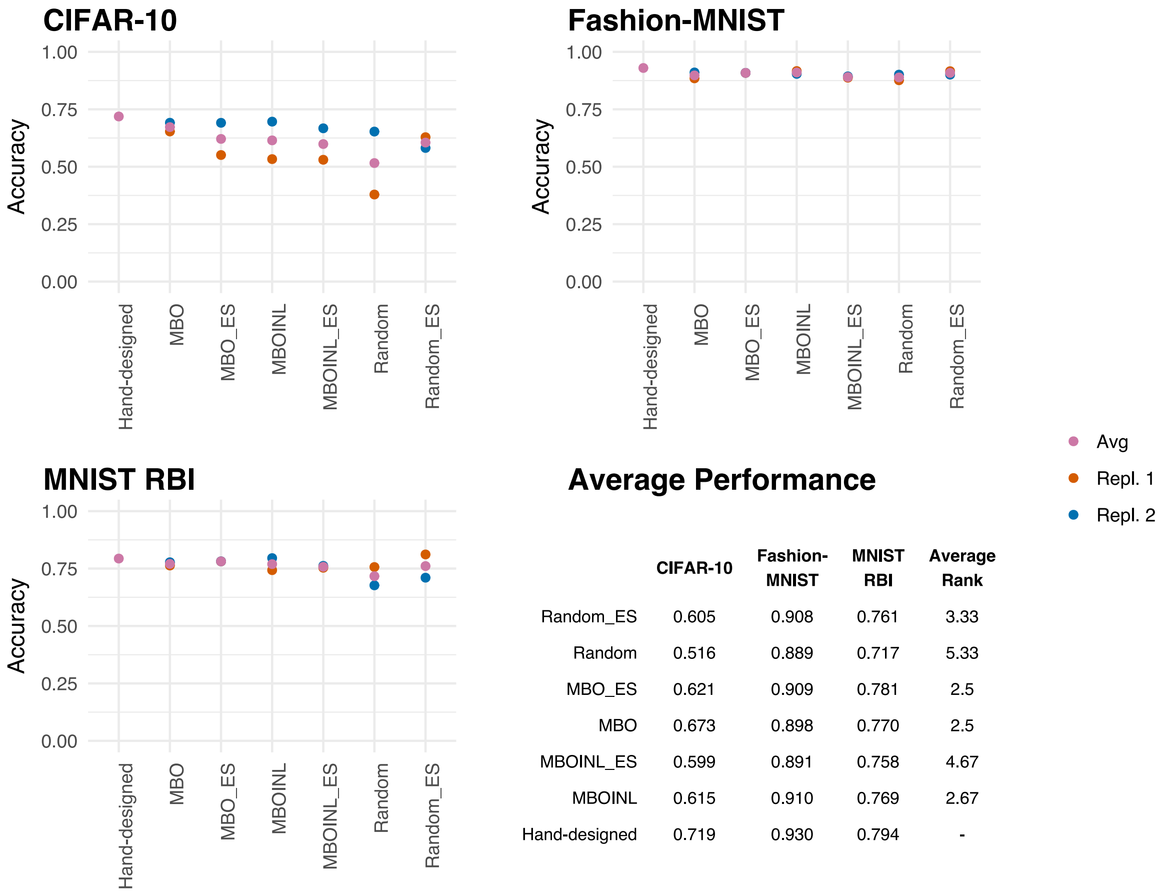


Figure 5.9: Rank of the performance on the test set (averaged over the replications on all data sets) over time.



## Average Performance

| | CIFAR-10 | Fashion-MNIST | MNIST RBI | Average Rank |
|---|---|---|---|---|
| Random_ES | 0.605 | 0.908 | 0.761 | 3.33 |
| Random | 0.516 | 0.889 | 0.717 | 5.33 |
| MBO_ES | 0.621 | 0.909 | 0.781 | 2.5 |
| MBO | 0.673 | 0.898 | 0.770 | 2.5 |
| MBOINL_ES | 0.599 | 0.891 | 0.758 | 4.67 |
| MBOINL | 0.615 | 0.910 | 0.769 | 2.67 |
| Hand-designed | 0.719 | 0.930 | 0.794 | - |

Figure 5.10: Performance on the validation set and rank of the performance on the validation set averaged over the replications on all data sets.

# Chapter 6

# Summary and Conclusion

We have tuned hyperparameters of feedforward neural networks with one to three hidden layers on CIFAR-10, Fashion-MNIST and MNIST rotated with background images using random search, MBO and MBOINL, each with and without early stopping and compared it to the results of a hand-designed neural network.

To take advantage of state-of-the-art libraries and frameworks for machine learning, model-based optimization and deep learning, a dedicated learner was created as an extension of mlr, integrating MXNet's functionality for feedforward neural networks.

Overall, all algorithms have yielded comparable results, with MBO (either with or without early stopping) always among the best performing algorithms and often displaying less variability between the different replications than the other algorithms.

The hand-designed neural network with two convolutional layers and one fully connected hidden layer showed better results than the configurations found by the algorithms. The number of network configurations with multiple convolutional layers that yielded 0 accuracy (due to ill-defined kernel and stride sizes) proposed by the algorithms indicates that automatically configuring kernel and stride sizes of convolutional neural networks is non-trivial.

MBOINL did not show an improvement in performance compared to the common MBO approach. However, some changes to the MBOINL approach to increase performance could be a subject of future work.

Instead of providing no design in MBOINL (apart from the small design due to technical reasons), a design could be provided for the lowest level. This would lead to gaining more knowledge about the hyperparameter space in the cheapest way, which could be exploited at the higher levels. This approach is in the spirit of Liu et al. [29] where all configurations of the least complex models are trained and evaluated, but not for the more complex models.

Increasing the number of allowed layers could be changed to a more hill climbing oriented approach where the algorithm is forced to evaluate a minimum number of networks with the maximum number of allowed layers. This would help prevent cases where the algorithm evaluates an unfavorable configuration at the newly allowed maximum number of layers and resorts to networks with less layers without properly exploring the new region of the hyperparameter space.

MBOINL may also benefit from a refined way of allocating the time budget to the different levels.

Concerning the hyperparameter space, both MBO and MBOINL would benefit from more prior knowledge or more conditionality concerning the kernel and stride sizes in order to prevent avoiding two or more convolutional layers due to unfavorable combinations of values leading to dimensionality problems. For example, if the kernel size of the first convolutional layer was rather large, the kernel size of the second convolutional layer could be restricted to rather small values. A more pragmatic approach could be to just reduce the possible kernel sizes for higher convolutional layers in a fixed manner directly in the hyperparameter set. Note that with these changes, better-performing configurations with multiple convolutional layers would most likely be found by random search as well. The surrogate learner, however, could possibly benefit even more.

For MBOINL, a small design could be evaluated for every level, which contains a multi-convolutional architecture that does not fail due to kernel and stride sizes. Ensuring such configurations are in the design for MBO could also be beneficial for the algorithms performance.

Using early stopping seems to add no gain in performance. However, taking the parameters of the epoch with the best in-training validation performance could have an influence on this result. Additionally, a greater number of maximum epochs could have an impact as well. Both issues should be investigated further.

Regarding the performance of MBO approaches in comparison to random search, using the expected improvement per second as the acquisition function could add efficiency to the MBO search.

In the spirit of Elsken et al. [13], employing some kind of network morphisms at an appropriate stage in the search process could yield additional efficiency benefits.

# Appendix A

# Digital Appendix

The digital appendix can be found on the corresponding USB flash drive. README.txt contains a short description of the benchmark setup as well as a description of the file structure and workflow. The folders benchmark and benchmark_1 contain the experiment registries created with batchtools to carry out the benchmarks, their respective benchmark code as well as some preprocessed results. The folder plots contains the produced plots. They can directly be replicated by running plots.R using the file results_processed.RData which contains the benchmark results in a suitable form.

# Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning.* MIT press, 2016.

[2] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *arXiv:1409.4842*, 2014.

[8] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, November 2012.

[9] T. N. Sainath, A.-R. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for LVCSR.," in *ICASSP*, pp. 8614–8618, IEEE, 2013.

[10] M. Helmstaedter, K. L. Briggman, S. C. Turaga, V. Jain, H. S. Seung, and W. Denk, "Connectomic reconstruction of the inner plexiform layer in the mouse retina," *Nature*, vol. 500, pp. 168–174, 08 2013.

[11] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *CoRR*, vol. abs/1103.0398, 2011.

[12] T. Ciodaro, D. Deva, J. M. de Seixas, and D. Damazio, "Online particle detection with neural networks based on topological calorimetry information," *Journal of Physics: Conference Series*, vol. 368, no. 1, p. 012030, 2012.

[13] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[14] M. A. Osborne, R. Garnett, and S. J. Roberts, "Gaussian processes for global optimization," in *3rd international conference on learning and intelligent optimization (LION3)*, pp. 1–15, 2009.

[15] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[16] B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang, "mlrmbo: A modular framework for model-based optimization of expensive black-box functions," *arXiv preprint arXiv:1703.03373*, 2017.

[17] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms," *CoRR*, vol. abs/1208.3719, 2012.

[18] M. Lang, H. Kotthaus, P. Marwedel, C. Weihs, J. Rahnenführer, and B. Bischl, "Automatic model selection for high-dimensional survival analysis," *Journal of Statistical Computation and Simulation*, vol. 85, no. 1, pp. 62–76, 2015.

[19] D. Horn and B. Bischl, "Multi-objective parameter configuration of machine learning algorithms using model-based optimization," in *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pp. 1–8, IEEE, 2016.

[20] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[21] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *arXiv preprint arXiv:1603.06560*, 2016.

[22] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

[23] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, pp. 2951–2959, 2012.

[24] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[25] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," *arXiv preprint arXiv:1704.00764*, 2017.

[26] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[27] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Reinforcement learning for architecture search by network transformation," *arXiv preprint arXiv:1707.04873*, 2017.

[28] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[29] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," *arXiv preprint arXiv:1712.00559*, 2017.

[30] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[31] F. Hutter and M. A. Osborne, "A kernel for hierarchical parameter spaces," *arXiv preprint arXiv:1310.5738*, 2013.

[32] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, and M. A. Osborne, "Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces," *arXiv preprint arXiv:1409.4011*, 2014.

[33] Z. Wang, B. Shakibi, L. Jin, and N. Freitas, "Bayesian multi-scale optimistic optimization," in *Artificial Intelligence and Statistics*, pp. 1005–1014, 2014.

[34] J.-C. Lévesque, A. Durand, C. Gagné, and R. Sabourin, "Bayesian optimization for conditional hyperparameter spaces," in *Proceedings of the International Joint Conference on Neural Networks*, 2017.

[35] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, N. De Freitas, *et al.*, "Bayesian optimization in high dimensions via random embeddings.," in *IJCAI*, pp. 1778–1784, 2013.

[36] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust bayesian neural networks," in *Advances in Neural Information Processing Systems*, pp. 4134–4142, 2016.

[37] D. Yogatama and G. Mann, "Efficient transfer learning method for automatic hyperparameter tuning," in *Artificial Intelligence and Statistics*, pp. 1077–1085, 2014.

[38] M. Feurer, J. T. Springenberg, and F. Hutter, "Using meta-learning to initialize bayesian optimization of hyperparameters," in *Proceedings of the 2014 International Conference on Meta-learning and Algorithm Selection-Volume 1201*, pp. 3–10, CEUR-WS. org, 2014.

[39] K. Swersky, J. Snoek, and R. P. Adams, "Freeze-thaw bayesian optimization," *arXiv preprint arXiv:1406.3896*, 2014.

[40] I. Ilievski, T. Akhtar, J. Feng, and C. A. Shoemaker, "Hyperparameter optimization of deep neural networks using non-probabilistic RBF surrogate model," *arXiv preprint arXiv:1607.08316*, 2016.

[41] S. C. Smithson, G. Yang, W. J. Gross, and B. H. Meyer, "Neural networks designing neural networks: Multi-objective hyper-parameter optimization," in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pp. 1–8, IEEE, 2016.

[42] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves.," in *IJCAI*, pp. 3460–3468, 2015.

[43] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration.," *LION*, vol. 5, pp. 507–523, 2011.

[44] T. Chen, I. Goodfellow, and J. Shlens, "Net2net: Accelerating learning via knowledge transfer," *arXiv preprint arXiv:1511.05641*, 2015.

[45] J. Wu, M. Poloczek, A. Gordon Wilson, and P. I. Frazier, "Bayesian optimization with gradients," *arXiv e-prints*, 03 2017.

[46] T. Welchowski and M. Schmid, "A framework for parameter estimation and model selection in kernel deep stacking networks," *Artificial intelligence in medicine*, vol. 70, pp. 31–40, 2016.

[47] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., Department of Computer Science, University of Toronto, 2009.

[48] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms," 2017.

[49] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[50] R Core Team, *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2016.

[51] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones, "mlr: Machine learning in R," *Journal of Machine Learning Research*, vol. 17, no. 170, pp. 1–5, 2016.

[52] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[53] M. Lang, B. Bischl, and D. Surmann, "batchtools: Tools for R to work on batch systems," *The Journal of Open Source Software*, vol. 2, feb 2017.

[54] Inkscape Project, "Inkscape."

[55] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of mathematical biology*, vol. 52, no. 1, pp. 99–115, 1990.

[56] P. Werbos, *Beyond regression: New tools for predicting and analysis in the behavioral sciences.* PhD thesis, Harvard University, 10 1974.

[57] D. Parker, M. I. of Technology, and S. S. of Management, *Learning Logic: Casting the Cortex of the Human Brain in Silicon.* Technical report: Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science, 1985.

[58] Y. LeCun, "Une procédure d'apprentissage pour réseau à seuil asymétrique," *Proceedings of Cognitiva 85, Paris*, pp. 599–604, 1985.

[59] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 10 1986.

[60] G. E. Hinton, "To recognize shapes, first learn to generate images," *Progress in brain research*, vol. 165, pp. 535–547, 2007.

[61] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, (Cambridge, MA, USA), pp. 153–160, MIT Press, 2006.

[62] M. A. Ranzato, C. Poultney, S. Chopra, and Y. L. Cun, “Efficient learning of sparse representations with an energy-based model,” in *Advances in Neural Information Processing Systems 19* (P. B. Schölkopf, J. C. Platt, and T. Hoffman, eds.), pp. 1137–1144, MIT Press, 2007.

[63] D. J. MacKay, *Information theory, inference and learning algorithms.* Cambridge university press, 2003.

[64] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting.,” *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[65] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International Conference on Machine Learning*, pp. 448–456, 2015.

[66] M. J. Sasena, *Flexibility and efficiency enhancements for constrained global design optimization with kriging approximations.* PhD thesis, University of Michigan Ann Arbor, MI, 2002.

[67] H. J. Kushner, “A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise,” *Journal of Basic Engineering*, vol. 86, no. 1, pp. 97–106, 1964.

[68] J. F. Elder, “Global r/sup d/optimization when probes are expensive: the GROPE algorithm,” in *Systems, Man and Cybernetics, 1992., IEEE International Conference on*, pp. 577–582, IEEE, 1992.

[69] B. E. Stuckman, “A global search method for optimizing nonlinear systems,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 18, no. 6, pp. 965–977, 1988.

[70] J. Mockus, V. Tiesis, and A. Zilinskas, *The application of Bayesian methods for seeking the extremum*, vol. 2. North-Holand, 1978.

[71] J. Mockus, “Application of bayesian approach to numerical methods of global and stochastic optimization,” *Journal of Global Optimization*, vol. 4, no. 4, pp. 347–365, 1994.

[72] G. Metheron, “Theory of regionalized variables and its applications.,” *Cah. Centre Morrphol. Math.*, vol. 5, p. 211, 1971.

[73] D. G. Krige, “A statistical approach to some basic mine valuation problems on the witwatersrand,” *Journal of the Southern African Institute of Mining and Metallurgy*, vol. 52, no. 6, pp. 119–139, 1951.

[74] N. Cressie, “Spatial prediction and ordinary kriging,” *Mathematical geology*, vol. 20, no. 4, pp. 405–421, 1988.

[75] P. Goovaerts, *Geostatistics for natural resources evaluation.* Oxford University Press on Demand, 1997.

[76] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn, "Design and analysis of computer experiments," *Statistical science*, pp. 409–423, 1989.

[77] J. Sacks, S. B. Schiller, and W. J. Welch, "Designs for computer experiments," *Technometrics*, vol. 31, no. 1, pp. 41–47, 1989.

[78] M. Locatelli, "Bayesian algorithms for one-dimensional global optimization," *Journal of Global Optimization*, vol. 10, no. 1, pp. 57–76, 1997.

[79] A. Žilinskas and J. Žilinskas, "Global optimization based on a statistical model and simplicial partitioning," *Computers & Mathematics with Applications*, vol. 44, no. 7, pp. 957–967, 2002.

[80] D. Horn, T. Wagner, D. Biermann, C. Weihs, and B. Bischl, "Model-based multi-objective optimization: Taxonomy, multi-point proposal, toolbox and benchmark.," in *EMO (1)*, pp. 64–78, 2015.

[81] D. Ginsbourger, R. Le Riche, and L. Carraro, "Kriging is well-suited to parallelize optimization," *Computational Intelligence in Expensive Optimization Problems*, vol. 2, pp. 131–162, 2010.

[82] B. Bischl, S. Wessing, N. Bauer, K. Friedrichs, and C. Weihs, "MOI-MBO: multiobjective infill for parallel model-based optimization," in *International Conference on Learning and Intelligent Optimization*, pp. 173–186, Springer, 2014.

[83] M. D. McKay, R. J. Beckman, and W. J. Conover, "Comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[84] M. L. Stein, *Interpolation of spatial data: some theory for kriging.* Springer Science & Business Media, 2012.

[85] P. J. Diggle, J. Tawn, and R. Moyeed, "Model-based geostatistics," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 47, no. 3, pp. 299–350, 1998.

[86] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *Journal of global optimization*, vol. 21, no. 4, pp. 345–383, 2001.

[87] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning: Data mining, inference, and prediction," *Biometrics*, 2002.

[88] E. Brochu, M. D. Hoffman, and N. de Freitas, "Hedging strategies for bayesian optimization," in *CoRR*, vol. abs/1009.5419, 01 2010.

[89] E. Vazquez and J. Bect, “Convergence properties of the expected improvement algorithm with fixed mean and covariance functions,” *Journal of Statistical Planning and inference*, vol. 140, no. 11, pp. 3088–3095, 2010.

[90] D. J. Lizotte, *Practical bayesian optimization.* University of Alberta, 2008.

[91] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy, “An experimental investigation of model-based parameter optimisation: SPO and beyond,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 271–278, ACM, 2009.

[92] A. I. Forrester, A. Sóbester, and A. J. Keane, “Multi-fidelity optimization via surrogate modelling,” in *Proceedings of the royal society of london a: mathematical, physical and engineering sciences*, vol. 463, pp. 3251–3269, The Royal Society, 2007.

[93] J. Schiffner, B. Bischl, M. Lang, J. Richter, Z. M. Jones, P. Probst, F. Pfisterer, M. Gallo, D. Kirchhoff, T. Kühn, J. Thomas, and L. Kotthoff, “mlr tutorial,” *CoRR*, vol. abs/1609.06146, 2016.

[94] P. Koch, B. Bischl, O. Flasch, T. Bartz-Beielstein, C. Weihs, and W. Konen, “Tuning and evolution of support vector kernels,” *Evolutionary Intelligence*, vol. 5, no. 3, pp. 153–170, 2012.

[95] B. Bischl, J. Schiffner, and C. Weihs, “Benchmarking classification algorithms on high-performance computing clusters,” in *Data Analysis, Machine Learning and Knowledge Discovery*, pp. 23–31, Springer, 2014.

[96] S. Hess, T. Wagner, and B. Bischl, “Progress: progressive reinforcement-learning-based surrogate selection,” in *International Conference on Learning and Intelligent Optimization*, pp. 110–124, Springer, 2013.

[97] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” *CoRR*, vol. abs/1511.07122, 2015.

[98] O. Roustant, D. Ginsbourger, and Y. Deville, “Dicekriging, diceoptim: Two R packages for the analysis of computer experiments by kriging-based metamodelling and optimization,” *Journal of Statistical Software*, vol. 51, no. 1, p. 54p, 2012.

[99] Y. Yan, *rBayesianOptimization: Bayesian Optimization of Hyperparameters*, 2016. R package version 1.1.0.

[100] D. Hernández-Lobato, J. Hernandez-Lobato, A. Shah, and R. Adams, “Predictive entropy search for multi-objective bayesian optimization,” in *International Conference on Machine Learning*, pp. 1492–1501, 2016.

[101] R. Martinez-Cantin, “Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3735–3739, 2014.

[102] T. Bartz-Beielstein and M. Zaefferer, “A gentle introduction to sequential parameter optimization,” *Schriftenreihe CIplus TR*, vol. 1, p. 2012, 2012.