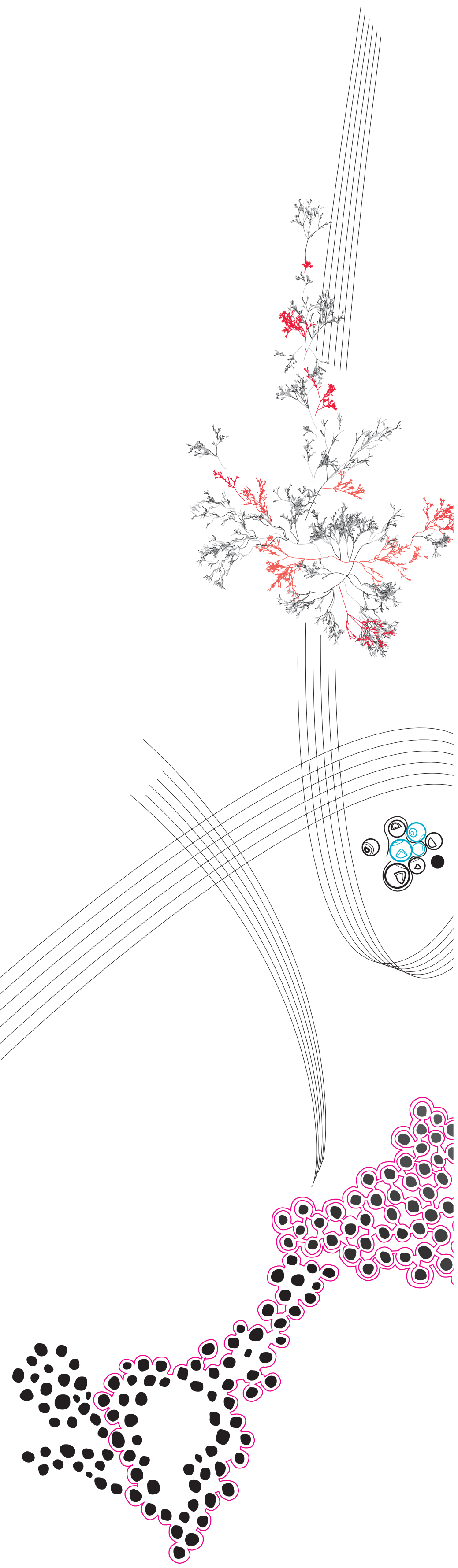
MSc Computer Science
Final Project



# Uncovering the Accuracy-Efficiency Trade-Off in Sparse Neural Network Training

Wouter Suidgeest

Supervisors:
Dr. Doina Bucur
Prof. Dr. Heike Trautmann
Dr. Luca Mariot
Jeroen Rook, MSc
Tim Smit, MSc (Capgemini)

November, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

## Abstract

Recently introduced sparse neural network training methods have been shown to match or even surpass the performance of comparable dense neural networks while increasing their computational efficiency. However, these experiments have been performed in controlled environments with fixed network architectures and hyperparameter configurations, potentially causing skewed results. We conducted experiments on six datasets, using multi-objective hyperparameter optimization in a configurable setting to approximate the accuracy-efficiency trade-off in neural networks with sparse neural network training. Afterwards, we performed a hyperparameter analysis to discover how hyperparameters influence this trade-off. Our results show that the efficiency of neural networks can be heavily improved for a slight decrease in accuracy and that sparse neural network training plays a vital but complex role in this trade-off.

*Keywords*: Multi-objective optimization, Hyperparameter optimization, Hyperparameter importance, Sparse neural networks, Sparse neural network training, Deep learning

# Contents

# Chapter 1

# Introduction

Over the past decade, deep neural networks have been the driving force of many research areas, producing state-of-the-art results for tasks such as image recognition and natural language processing [22, 15]. Unfortunately, what deep neural architectures win in performance, they lack in sustainability. Larger models with more parameters often outperform comparable smaller models, and these larger models come with substantial storage and energy costs. With the continued rise in popularity of machine learning and neural networks, these sustainability issues are becoming an increasingly large problem. Fortunately, there are ways to combat these issues. Pruning methods show that many models contain redundant parameters, which can be removed to decrease a model's size. These efficient new sparse neural network (SNN) structures created by pruning often come with a negligible loss in performance [11]. Pruning helps make trained neural networks more efficient, but training these networks remains an inefficient and computationally expensive task. Recent research introduced the concept of SNN training, where networks are also sparse during training, resulting in a more efficient way to create SNNs [82, 26]. Current research presents the effect of sparsity in SNN training in an accuracy-efficiency trade-off and shows that such networks require significantly fewer computations to train while having comparable or even improved accuracy compared to dense neural network (DNN) models [26]. However, these trade-offs are primarily explored in restricted environments with fixed network structures and hyperparameter configurations. We believe that research into the actual effects of sparsity is missing, as we might find a different accuracy-efficiency trade-off when applying sparse training to different network structures and hyperparameter configurations. Therefore, this research aims to uncover the true underlying accuracy-efficiency trade-off of SNN training. This goal directly leads us to our first research question:

> **RQ1:** What does the accuracy-efficiency trade-off in sparse neural network training look like in an environment with many configurable hyperparameters?

To generate comparable and measurable results from this question, we aim to answer the following sub-research questions simultaneously.

> **RQ1.1:** How does the accuracy-efficiency trade-off in sparse neural network training compare to the trade-offs found in previous studies without a configurable environment?

> **RQ1.2:** How is the accuracy-efficiency trade-off in sparse neural network training dependent on the network architecture type that is being optimized and the data type it is optimized on?

To find this trade-off, we use Hyperparameter Optimization (HPO). HPO is a sub-field of automated machine learning and automated algorithm configuration that focuses on finding the set of hyperparameters that optimizes the performance of a machine learning model on a given dataset [50, 9]. Machine learning performance can be measured in multiple ways with different objectives, but we focus on accuracy and efficiency in this study. Accuracy stands for the percentage of correctly predicted samples in a classification problem. Efficiency can be separated into two objectives: training efficiency and inference efficiency. We define training efficiency as the total number of floating point operations (FLOPs) needed to train a model and inference efficiency as the number of FLOPs needed to pass one data point through the network after training. Most HPO methods optimize only for a single objective, such as accuracy, but to determine the accuracy-efficiency trade-off, we use a Multi-Objective HPO (MO-HPO) method [57]. Such methods do not optimize a single objective but find different configurations that approximate the optimal underlying trade-off between these objectives. By combining this technology with a configurable SNN training environment and making accuracy, training efficiency and inference efficiency explicit objectives, we can approximate the accuracy-efficiency trade-off incurred by SNN training.

Knowing how accurate and efficient SNN training can be is our primary goal. However, these results become far more valuable if we gain further insights from them that can be used in future research. Therefore, we set the secondary goal to study the effects of the configured hyperparameters. This leads us to the second research question:

> **RQ2:** How do sparsity and other hyperparameters influence the accuracy-efficiency trade-off in sparse neural network training?

To study the influence of hyperparameters on an objective trade-off, we perform a multi-objective hyperparameter importance analysis [109]. In this analysis, the importance of each hyperparameter is quantified per objective, giving an idea of which hyperparameters play a role in improving which objectives. Furthermore, we will perform an analysis on the optimal values of each hyperparameter, showing how hyperparameters should be configured to find an optimal accuracy-efficiency trade-off.

To answer these research questions, we have chosen six benchmark datasets on which we performed three experiments. In these experiments, we approximate the accuracy-efficiency trade-off induced by SNN training similar to what was done in reproduction, using MO-HPO and a combination of these two methods. Furthermore, we analyse the resulting trade-offs and their corresponding hyperparameter configurations. Our results show that SNN training can drastically improve the efficiency of large neural network models for a small but unpredictable decrease in accuracy. The remainder of this report is structured as follows. Chapter 2 introduces SNN training literature and methods, better describing how this technique works and why its efficiency should be studied. Chapter 3 gives a detailed introduction to how MO-HPO can be used to approximate objective trade-offs and covers related work. Chapter 4 explains relevant methods and decisions and introduces a set of experiments with which we answer the first research question. Chapter 5 further specifies the details of these experiments and presents their results. Chapter 6 uses the results of the previous experiments for further analysis to answer the second research question. We conclude the report and discuss future work in Chapter 7.

# Chapter 2

# Sparse Neural Networks

This chapter provides a further theoretical background to Sparse Neural Networks. We first introduce core concepts of neural networks and show how sparsity could be a solution to their efficiency problems. Then, we cover the state-of-the-art of SNN training and show why more research into their efficiency is needed. To keep this chapter readable, these sections introduce only the key concepts related to this study, and we direct the reader to the referenced articles for a more detailed explanation.

## 2.1 Neural Networks

### 2.1.1 Machine Learning

Many people have heard of artificial intelligence (AI). While there does not seem to be a single definition for the term, it boils down to creating machines that have intelligence, a unique and complex trait of creatures such as humans. Intelligence is equally difficult to define, but one crucial aspect is that intelligent creatures are capable of learning: gaining a better understanding of something by analyzing examples. Allowing machines to mimic this behaviour is an essential step towards artificial intelligence, and it is what machine learning (ML) focuses on [10]. Machine learning often uses the fact that complex natural phenomena result from unknown probability distributions and, thus, can be mathematically modelled and estimated. This can be done in many ways, depending on the goals and available data. This report focuses on supervised learning, where a machine (a model) is exposed to many examples (a dataset) and finds relations between them. For example, we might want to use it to predict the type of flower given the size of its petals. There are many different methods which can be used to achieve this, such as linear regression, decision trees [93], support vector machines [17], random forest classifiers [41], restricted Boltzmann machines [104] and neural networks [96]. The process of exposing such a model to a dataset in order for it to learn is called fitting or training. After a model is trained, it can be deployed and used for the task it is trained for.

### 2.1.2 Supervised Training

To better understand how such methods work, we start by understanding how they are trained. To do so, we will explain essential aspects of supervised training for a classification task. The first step is to collect a dataset filled with input (e.g. petal size) and output (e.g. flower type) examples of the problem we wish to solve. Then, this dataset is split into three subsets: the training, validation and test dataset [36].

We use the training dataset to train a model on this data. Here, the first step is to choose a model type. The second step is choosing values for all hyperparameters[1]. Each model type has different hyperparameters, and the optimal values for these hyperparameters differ per problem. After choosing these values, we can train the model. Different models are trained in different ways. For example, some models systematically process the dataset once to learn parameters, while others iteratively loop through the data, where every complete pass-through of the training set is called an epoch. After training, we can evaluate how well a model has learned the data using the validation set. Evaluation is done by letting the trained model classify each sample in the validation set, after which we evaluate whether the result is correct. Different statistics can be calculated to measure this performance, of which accuracy (the percentage of correct predictions) is among the most popular.

This entire process is usually repeated multiple times with different hyperparameters. Eventually, the best model and hyperparameter configuration are chosen and evaluated once more on the test set. This gives an unbiased measure of the network's performance and presents the expected performance.

Since data is often scarce and more data generally better covers the data distribution, we want to use all available data to tweak our network. Therefore, a k-fold cross-validation approach is often used (see Figure 2.1). With this approach, we do not create a single training and validation dataset pair, but we create $k$ "folds" (i.e. train-validation pairs). For each fold, the network is trained on the training set and validated on the validation set. Eventual validation performance can be calculated by averaging the validation results. This may make the search for the best hyperparameters take longer, but it will result in better generalised results.

Formally, we define the ML problem using a formulation similar to the notation introduced in [57]. Our dataset $\mathcal{D}$ consists of input feature vectors $\mathbf{x}$ with output labels $y$. This is resampled into $\mathcal{J}$, a collection of $\mathcal{K}$ training and validation sets $\mathcal{D}^k_{train}$ and $\mathcal{D}^k_{val}$. An ML model $\mathcal{I}$ with hyperparameters $\lambda \in \Lambda$ can be learned to map $\mathbf{x}$ to $y$ using $\mathcal{D}^k_{train}$. This learned model is denoted as $\mathcal{I}_\lambda(\mathcal{D}^k_{train})$. We use the loss $L$, a difference measure between $y$ and the predicted output, to calculate how well the model $\mathcal{I}$ learned this mapping. When choosing our model and hyperparameters, we wish to minimize the expected generalization error $\hat{GE}$:

$$\hat{GE}(\mathcal{I}, \mathcal{J}, L, \lambda) := \frac{1}{K} \sum_{k=1}^{K} \frac{1}{|\mathcal{D}^k_{test}|} \sum_{(\mathbf{x},y) \in \mathcal{D}^k_{test}} L(y, \mathcal{I}_\lambda(\mathcal{D}^k_{train})(\mathbf{x})). \tag{2.1}$$

### 2.1.3 Introduction to Neural Networks

One of the first machines capable of learning was the perceptron, introduced by Frank Rosenblatt in 1958 [96]. The perceptron was designed for image recognition and could recognize simple shapes in $20 \times 20$ input images. It realised this by connecting 3 layers of so-called cells to each other: the projection, association and response layers. There are some limitations to the original Perceptron, but it inspired what would become one of the most popular ML models: the multi-layer perceptron.

Multi-layer perceptions (MLPs) are the most basic type of neural networks. They consist of several layers of so-called neurons. Usually, there are three or more layers: an

[1]The parameters that define an ML model are learnable. The parameters that define the model's learning process are hence referred to as hyperparameters.

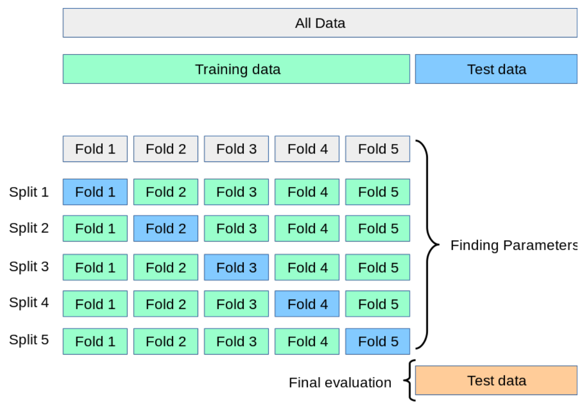


FIGURE 2.1: Experimental setup for a supervised learning task using k-fold cross-validation with k=5. The data is split into training and testing data, after which the training data is split into several folds with training and validation data. Image taken from[2].

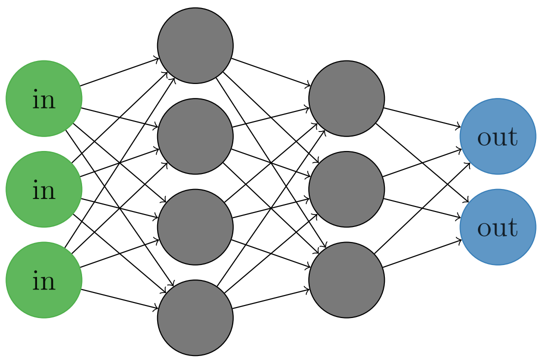


FIGURE 2.2: An example standard multi-layer perceptron with 3 inputs, 2 outputs and 2 hidden layers of sizes 4 and 3.

input layer, an output layer, and a number of hidden layers in between, as visualized in Figure 2.2. Each neuron is a non-linear function of its input, the outputs of the previous layer of neurons multiplied by so-called weights. Together, this makes the entire network a large, complex function of its input. By scaling the number of layers and their sizes and properly setting the weights, neural networks can describe any continuous function to arbitrary precision [45]. This 'universal approximation' property is one of the reasons for the popularity of neural networks. Machine learning often aims to model a distribution function, and we know that neural networks can model them.

In later years, backpropagation and stochastic gradient descent (SGD) [94, 115, 13] were introduced. SGD takes samples of the training dataset and passes them through the neural network. This output prediction is combined with the actual output to calculate the loss. Backpropagation uses this loss to slightly update all weights and biases of the network so that the loss will be lower on the next passthrough of this sample. By doing this iteratively for each sample in the dataset, the network slowly learns to map the given input data to the given output data in the training set. This technique allowed for the efficient and scalable training of neural networks, which led to an increase in the possibilities of neural networks.

Nowadays, a large part of neural network research is focused on deep learning: advanced

[2]See `https://scikit-learn.org/stable/modules/cross_validation`. Last visited on 17/10/2024.

neural network architectures consisting of multiple layers [65]. These architectures add multiple levels of abstraction, allowing models to better model underlying structures in data. Examples of deep learning architectures are convolutional neural networks [64], residual networks [38], and transformers [112]. Such architectures are better suited for different types of data and make neural networks capable of reaching state-of-the-art results in several research areas [15, 22, 54, 28].

### 2.1.4 Efficiency of Neural Networks

With an increase in network architecture complexity and hardware capabilities, neural networks have grown exponentially in size, with some networks needing over 10 billion times as many calculations as the Rosenblatt perceptron [97], and this trend does not seem to flatten yet. This increase in neural network size not only increases the networks' performance but also decreases the networks' efficiency. These larger networks require more storage and substantially more energy to be trained and deployed. Next to the decreased efficiency of training a single network, many more networks are trained. The large variability in network architectures and corresponding hyperparameter options makes it hard to find the best configuration, which causes developers to train many large neural networks. This large increase in neural network energy use is not only undesirable due to the global energy and sustainability crisis but also the immense financial costs required to achieve state-of-the-art results [3] and the impracticality of running neural networks on low-resource devices. While the true energy consumption and related costs of the largest AI models are often unknown or overestimated [107, 90], it is certain that the energy used by ML is rising and that ML practitioners and researchers should pay attention to the sustainability of their models [90, 111]. Schwartz et al. [100] warn for "Red AI": the trend of focusing AI research on small accuracy improvements in exchange for decreased efficiency. Instead, they recommend giving efficiency an equal priority as accuracy, which they dub "Green AI". Furthermore, research should continue to develop more efficient ML techniques.

#### Measuring Neural Network Efficiency

To prioritise efficiency, we first need to know how we can measure efficiency. First, we can separate efficiency into training efficiency and inference efficiency. Training efficiency relates to the computational efficiency of the entire training phase of the network. Inference efficiency relates to the computational efficiency of calculating a single result using the network. Calculating a single result is always substantially more efficient than the entire training process. Still, it is essential to note that the best statistic for optimal sustainability differs per use case. The more a network is used after training, the less relevant its training efficiency is relative to its inference efficiency. For example, Google spends about 1.5 times more energy on ML inference than ML training [90] while a network trained for research purposes generally spends much more energy on training than inference.

Since our primary goal of increasing efficiency is to decrease the energy consumption of neural networks, it makes sense to measure network efficiency based on energy consumption. While tools exist to measure ML energy consumption [3, 99], they are generally not used for this purpose. Energy consumption depends on the efficiency of your implementation, the technical capabilities of your (cloud) computer and the efficiency of energy production in the computer's area. When researching and developing new efficient neural

[3]See `https://lambdalabs.com/blog/demystifying-gpt-3`. Last visited on 04/03/2024.

network training methods, we need a unit of measure independent of these variables to optimize the method's efficiency. As such a measure, Schwartz et al. [100] advocate publishing and minimising the number of floating point operations (FLOPs[4]) used to train a network and the number of FLOPs needed for inference of the network. They define a FLOP as the addition, subtraction, multiplication or division of two floating point numbers. This way, the FLOPs represent the amount of work a network needs, independent of hardware or local energy efficiency differences. Furthermore, the FLOPs needed for a calculation can be computed relatively easily, making it a popular unit of measure for network efficiency.

However, using the FLOPs to measure efficiency also has problems if our goal is to improve sustainability. First, there is a big energy difference between different floating point operations. Luo and Sun [75] show that floating point multiplications require $\approx 3\times$ as much energy as floating point additions and vastly improve a model's energy efficiency by focussing on addition operations. Furthermore, Patterson et al. [90] argue that other operations, such as main memory accesses, also significantly influence an ML model's carbon emissions and should not be omitted from the calculation. For example, some neural networks can be made much more efficient by decreasing these other operations while using substantially more FLOPs [68]. Unfortunately, to our knowledge, no unit of measure correctly captures the efficiency by combining these operations. Instead, Patterson et al. [90] recommend measuring latency or carbon emissions directly, but these measures have the problem of incomparability mentioned before.

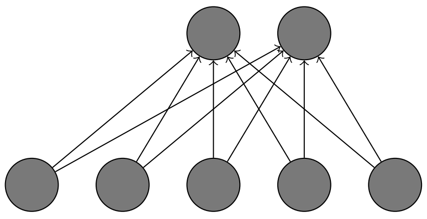
### Efficient Neural Networks

Now that we know why it is important to pay attention to the efficiency of neural networks and how we can measure this efficiency, we can look into how we can improve efficiency. First, ML practitioners can look at their choice of model. While we focus on neural networks, we must not forget about the abundance of other options. Different models yield the best results for different applications, and many models, such as decision trees, tend to be more efficient than neural networks [52]. If a neural network model is chosen, practitioners should pay attention to its relevant hyperparameters. Hyperparameters such as the size of a network and the number of training epochs influence a network's efficiency. However, next to these standard options, additional techniques have been developed to increase efficiency.

Many of the first techniques developed to increase the efficiency of neural networks focused on model compression: limiting a model's size after training. These techniques usually do not increase the training efficiency but do make a difference in inference efficiency and required storage. Some popular model compression techniques are knowledge distillation [40], quantization [31] and, most notably, pruning [86, 66, 11].

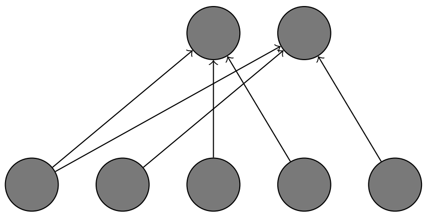
Pruning is one of the most popular model compression techniques, and its concept is fairly simple. After a neural network has been trained, we can analyze which parts are least important when generating results and remove them. For example, take the network layer in Figure 2.3a. After the network is trained, all weights have received values such that they map the relation between the desired input and output values. Looking at the values of the weights, we can see that some seem far less influential than others. In pruning, we determine the least important weights using a method such as magnitude-based selection [34] and delete them, leaving us with the layer in Figure 2.3b.

[4]We refer to the plural of FLOP as FLOPs. This is not to be confused with FLOPS, a popular measure for hardware performance denoting the number of FLOPs per second.



$$\Leftrightarrow \begin{pmatrix} 2.01 & -3.23 \\ \mathbf{0.01} & 1.58 \\ 1.70 & \mathbf{0.51} \\ -2.11 & \mathbf{-0.03} \\ \mathbf{-0.05} & 1.24 \end{pmatrix}$$

(A) A network layer after training



$$\Leftrightarrow \begin{pmatrix} 2.01 & -3.23 \\ \textbf{-} & 1.58 \\ 1.70 & \textbf{-} \\ -2.11 & \textbf{-} \\ \textbf{-} & 1.24 \end{pmatrix}$$

(B) A network layer after pruning

FIGURE 2.3: An example of a neural network layer and its corresponding weight matrix before and after pruning. A weight is deleted by freezing its value at 0.0.

## 2.2 Sparse Neural Networks

A network consisting only of fully connected layers, such as in Figure 2.3a, is called a dense neural network (DNN). Otherwise, if a network has sparse layers, such as the network in Figure 2.3b, the network is called a sparse neural network (SNN). This study focuses on SNN methods to improve neural network efficiency. We refer to the percentage of weights that have been pruned in an SNN as sparsity. So, if 90 of the 100 weights of a DNN have been pruned, we end up with an SNN with a sparsity of 90%. Generally, SNNs have sparsities ranging from 50 to 99%. In this section, we will elaborate on the advantages of SNNs and introduce other methods to create SNNs.

### 2.2.1 SNN Advantages

#### Efficiency

The most notable improvement of SNNs is their efficiency. By removing parameters from a network, we are reducing the number of calculations required to perform inference on a network. Most calculations in a neural network are multiplications between neuron outputs with weights of neurons in the next layer. SNNs reduce the number of weights in a network and, with that, directly limit the number of calculations needed to perform inference on the network. Quick estimates show us that a sparsity of 99% could remove $\approx$ 99% of computations from the popular AlexNet network [62], causing a 100$\times$ efficiency improvement [106]. However, such optimistic estimations are far from accurate. Since dense matrix populations are so frequent in computer calculations, GPUs and libraries are specially made to compute dense matrix calculations efficiently. Therefore, using default machine learning libraries and standard GPUs for sparse matrix calculations would not increase efficiency at all. Fortunately, specialized computers for sparse linear algebra exist[5], and research to optimize these calculations is ongoing [120, 79, 30], making it reasonable to assume that this efficiency limitation will cease to exist in the foreseeable future.

#### Storage

Logically, decreasing the number of weights that need to be stored decreases the total storage required for a neural network. However, the reduction in storage requirements does not directly scale with the sparsity. When storing a weight, we need its value and its location, specifying the two nodes it is connected to. This location is implicitly defined in dense matrices by the value's row and column but needs to be explicitly defined in sparse matrix formats. Therefore, storage requirements only decrease starting from a certain sparsity threshold. This threshold depends on the data type and storage format. For example, storing integers using the coordinate list format[6] stores three integers per matrix value: the value, and its row and column position. Therefore, this format decreases storage requirements starting from a sparsity of $\approx 66\%$.

#### Generalization

Many machine learning networks are overparameterized, and pruning aims to identify and remove redundant parameters. Often, these redundant parameters do not change the network's result or only do so in very specific cases. In the latter case, the network probably has learned noise: small, unforeseen perturbations in the training data that should not affect the result. By removing these weights, we force the network to 'focus' on the more critical aspects of the data, increasing its ability to model data it was not trained on, i.e. its generalization [42].

### 2.2.2 SNN Training

If a DNN can be decreased in size without losing much accuracy after training, it would make sense that we should be able to train an already pruned network from scratch. Such a training procedure could be more efficient and allow for larger SNNs, as the largest SNN that we can train is limited by the size of the largest DNN that we can train. While training a pruned network from scratch seemed to reach lower accuracies than dense networks, it was already shown in 2015 by Han et al. [34] that it is possible to retrain a pruned network from scratch if we use the same initial weights as were used in training the dense network (i.e. the weights are not re-initialized), which was backed up in 2019 by Frankle and Carbin with the Lottery Ticket Hypothesis [29]. This hypothesis claims that "A randomly-initialized, dense neural network contains a subnetwork [i.e. a winning ticket] that is initialized such that —when trained in isolation— it can match the test accuracy of the original network after training for at most the same number of iterations". Essentially, this makes sparse neural network training a subset selection problem for graphs [21, 69]. Furthermore, Frankle and Carbin showed that training an SNN from scratch with re-initialized weights is often possible when the sparsity stays under 80%. Following the studies showing that it is possible to train an SNN 'from scratch', more methods have been developed over the years to try to train an SNN from scratch, a research area which we call SNN training. The research on SNN training can be categorized into several types as

[5]See https://cerebras.ai/press-release/cerebras-announces-third-generation-wafer-scale-engine. Last visited on 04/08/2024.

[6]See https://docs.scipy.org/doc/scipy/reference/sparse.html. Last visited on 04/08/2024.

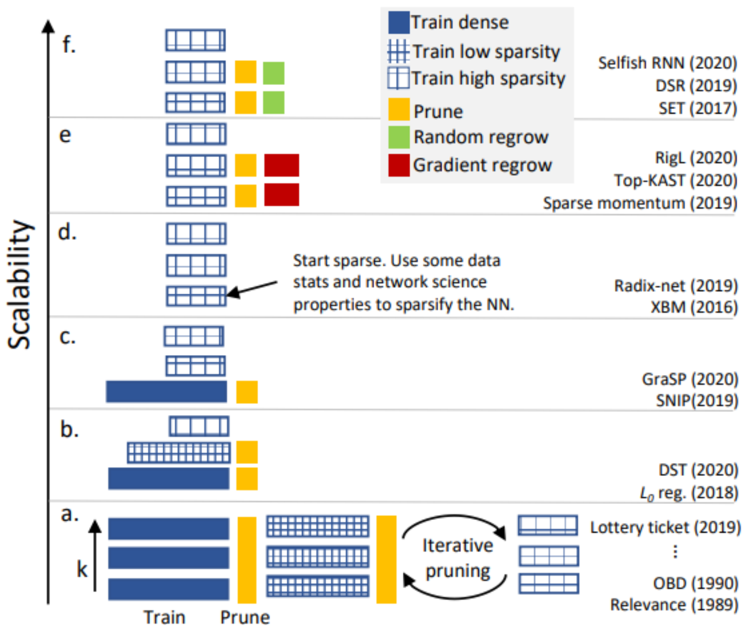


FIGURE 2.4: Schematic representation of various method types used to obtain sparse neural networks and a rough estimation of their scalability; a. Pruning, b. Simultaneously training and pruning, c. One-shot pruning, d. Static sparse training, e. Dynamic sparse training (gradient), f. Dynamic sparse training (random). Figure taken from [81].

in Figure 2.4. This overview only lists methods until 2020. Novel techniques have been developed since then (e.g. [73, 4, 63]), but the most important methods and categories are covered in Figure 2.4. In the remainder of this section, we will cover the static and dynamic sparse training categories and introduce relevant techniques in these categories.

### Static Sparse Training

In static sparse training, the topology and weights of an SNN are initialized using an initial DNN, sparsity level and dataset. While it has been shown that it is difficult to train a randomly initialized SNN with a high sparsity, these methods manage to define initial weights for a sparse network topology that can be trained using statistics from the training data and by applying network science properties. For example, Radix-Nets [58] create networks by combining topologies based on mixed-radix number systems. These networks exhibit path-connectedness (each output depends on all inputs) and symmetry (there is an equal length between each input and output pair). Furthermore, complex Boltzmann machines (XBMs) [80] create small-world (most node pairs are only a few steps away) and scale-free (most nodes have few connections, but some have many) topologies for Boltzmann machines while taking data distributions of the training data into account. These network science properties, which are well-studied and are known to exist in many biological (neural) networks, help these networks to be trained sparsely from scratch.

### Dynamic Sparse Training

In dynamic sparse training, an SNN's topology is not fixed after initialization but is constantly updated during training. We know from the Lottery Ticket Hypothesis that a trained DNN is primarily defined by a sparse subnetwork and that stochastic gradient descent searches for this subnetwork from all possible subnetworks during training. In dynamic sparse training, we can start with an imperfect sparse subnetwork and use the weight updates of stochastic gradient descent to update the topology towards the desired subnetwork. In practice, this often results in SNNs that achieve high accuracies and exhibit similar network properties as static sparse networks, such as small-worldness and scale-freeness [82].

Dynamic sparse training methods update their topology by periodically deleting and growing weights. They primarily differ in how they choose which weights to grow, how they choose which weights to delete and when topology updates take place. The most important distinction can be found in the first design choice, which we explain by highlighting two of the most relevant SNN training methods, SET [82] and RigL [26].

SET (Sparse Evolutionary Training) is one of the simplest dynamic sparse training methods and, to our knowledge, also the first. In its default setting, it initializes a network with a given sparsity following a random 'Erdös Rényi' topology, scaling the number of connections in a layer with the sizes of the input and output layers. In each training epoch, a fraction of the smallest positive and largest negative weights are removed from each layer, after which an equal number of weights is randomly regrown. After the last training epoch, the same fraction of weights is removed but none are regrown. With this method, the network sparsity is constant throughout the training process, and only relevant and trained weights will remain in the final network. Some details differ per implementation, such as the frequency of topology updates and the initial distribution of weights among layers. Despite being the first dynamic sparse training method, SET remains relevant among newer methods [16].

RigL (the Rigged Lottery) is a newer method, adding extra complexity to the sparse training process. RigL primarily differs from SET in its regrowth strategy. After removing a fraction of the smallest positive and largest negative weights from each layer, the gradients of all possible dense weights are computed. Afterwards, weights are grown not randomly but based on the dense gradients. This adds some computational overhead and limits the size of the largest possible sparse network by the largest possible dense network that fits in memory. In their work, Evci et al. [26] calculate that this overhead is often negligible and show that it allows RigL to achieve higher accuracies than other methods.

Several other SNN training methods exist that slightly change the behaviour of SET in different ways, such as [85] and [20]. For most methods, a user defines the dense starting topology of a network and the desired total sparsity level. These methods then choose how to distribute this sparsity among the different layers. Often, the first and last layers remain dense, while all other layers are sparsified.

### 2.2.3 SNN Training Results

Successful SNN training methods result in SNNs that match or surpass their dense counterparts in accuracy with significant efficiency improvements. For example, [26] report that using RigL to train WideResNet-22-2 on CIFAR-10 can surpass the test accuracy if a sparsity level of 50% is set and that the accuracy steadily decreases with higher sparsity levels. These results can be found in Figure 2.5. Furthermore, they note that RigL's accuracy consistently increases if trained longer. Using this knowledge, they show that RigL can be used to train a ResNet-50 on the Imagenet-2012 dataset with an accuracy increase of 0.3 percentage points using 0.42× the inference FLOPs but 2.09× the training FLOPs.

Knowing the trade-off between the accuracy and efficiency of an SNN training method, such as presented in Figure 2.5, allows neural network designers to choose the desired sparsity and training method for their networks. Such trade-offs are often presented in

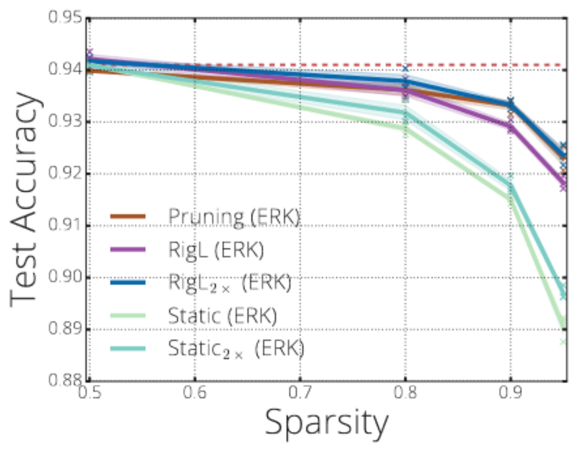


FIGURE 2.5: Test accuracies of sparse WideResNet-22-2's on CIFAR-10 task trained using different sparsity methods. Figure taken from [26].

papers that introduce new SNN training methods [82, 26, 118, 85]. However, we believe that there are three limitations to these results. First, none of them attempt to estimate the entire underlying trade-off. Instead, they choose a few levels of high sparsity, such as 80%, 90% and 95%. Arguably, these are the most interesting sparsity levels for SNNs due to their obvious efficiency improvements and their comparability with other papers. Nevertheless, the global trade-offs of these methods remain unknown and could provide interesting insights for future research.

Second, these studies research the capabilities of SNN training only in a few networks, such as ResNet-50 [37]. Different neural network training methods sometimes work better on different network architectures. To the best of our knowledge, no research has been conducted to discover whether SNN training results differ within different architectures.

Finally, all of these studies estimate the trade-offs by fixing all hyperparameters except for the sparsity, generating an accuracy-efficiency trade-off within this fixed hyperparameter environment. Sometimes, these fixed hyperparameters have been chosen using small hyperparameter optimization experiments, or their relations to sparsity are explored, but it might be perfectly possible that the best hyperparameter combinations remain unknown. We believe all hyperparameters need to be configurable to shift this local comparison to a global comparison and find a realistic trade-off between accuracy and efficiency.

To combat any of these limitations, a fully configurable SNN training environment must be combined with an automation tool. To the best of our knowledge, no SNN training method explores its method in such an environment.

# Chapter 3

# Hyperparameter Optimization

In this chapter, we explore the relevant theoretical background for hyperparameter optimization. Next to giving an introduction to the fields of hyperparameter optimization and importance, we list state-of-the-art methods and explore the use of hyperparameter optimization for sparse neural network training. Just as in Chapter 2, we only introduce the key concepts related to this study, and we direct the reader to the referenced articles for more detailed explanations.

## 3.1 Algorithm Configuration

When algorithm developers choose not to make premature design decisions themselves but expose those decisions as configuration parameters, algorithms can be more versatile, and development time can be saved. This approach, known as Programming by Optimization [43], has been embraced by developers in many fields, creating highly configurable algorithms with much potential. However, by increasing the possibilities of algorithms, it becomes increasingly difficult to find proper configurations for these algorithms, leading to the field of algorithm configuration (AC). Formally, we can define AC as a black-box optimisation problem using notations similar to [98]. AC aims to optimise a target algorithm $\mathcal{A}$ with its corresponding set of performance-affecting parameters $\Lambda$. Furthermore, there exists a set of problem instances $\mathcal{I}$ with probability distribution $\mathcal{P}$, for which we have a set of training examples $\mathcal{I}_{train} \subseteq \mathcal{I}$ that cover the problem space we wish to solve with $\mathcal{A}$. Finally, we need a cost measure $c$ that quantifies the cost of running the algorithm with configuration $\lambda \in \Lambda$ and instance $i \in \mathcal{I}$. After defining an objective function $m$ such as the total cost over $\mathcal{I}_{train}$ using $c$ and a configuration $\lambda$, AC can search for the configuration $\hat{\lambda} \in \Lambda$ that minimizes the cost over all problem instances as follows:

$$\hat{\lambda} \in arg \min_{\lambda \in \Lambda} m(c, \mathcal{I}_{train}, \lambda). \tag{3.1}$$

One can manually perform AC using, for example, trial and error. However, this has proven not to be a trivial task due to several problems. First, evaluating a configuration $\lambda$ on $\mathcal{I}_{train}$ is often a time-costly task that can easily take hours to execute. Second, the effects of and relationships between different configurable parameters are often complex. Therefore, we need expert knowledge to find good configurations, which we might not always have. Finally, the search space can become very complex due to these complex effects and relationships, making it difficult to find global optima even with expert knowledge. These difficulties make the automation of this process an attractive option and have resulted in

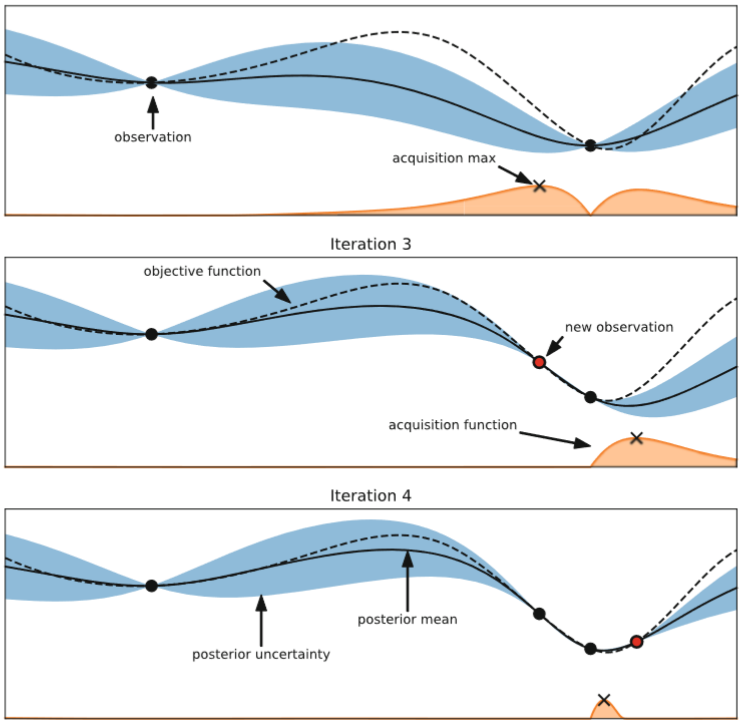


FIGURE 3.1: Illustration of Bayesian optimization. The goal is to minimize the dashed line using a surrogate model (black line with blue tube) by maximizing an acquisition function (orange curve). Figure taken from [50].

the research field of automated algorithm configuration (AAC), in which meta-algorithms automatically choose such configuration parameters.

### 3.1.1 AC Methods

Over the past two decades, many AAC methods have been developed to alleviate AC's problems. Indeed, AAC has been shown to find better configurations in multiple problem fields, such as SAT [55, 51] and MIP [55]. AAC methods can be subdivided into two groups: model-free and model-based methods. Model-based methods fit the results of evaluated configurations to a separate model. This model, the surrogate model, can be used to estimate the results of an unknown model. A popular model-based strategy is Bayesian optimization. Here, an acquisition function determines promising configurations by trading off exploration and exploitation. An illustration of how an acquisition function interacts with the surrogate model can be found in Figure 3.1. As the name suggests, model-free methods do not make such estimations.

Within both groups, ParamILS [49] and SMAC [48] are among the most popular. The model-free ParamILS is one of the first general-purpose AAC methods. It chooses a random parameter configuration, uses Iterated Local Search [74], changes one parameter in each iteration and avoids local optima by randomly resetting the parameter configuration while optimizing. On the other hand, SMAC (Sequential model-based optimization) is a general-purpose model-based method based on Bayesian optimization. SMAC uses a random forest classifier [41] as surrogate model, a classifier that is able to model regression and classification tasks, and the Expected Improvement (EI) [53] as acquisition function. This has been shown to successfully determine good configurations for many various tasks. Both methods have been further researched and developed over the years [71, 92].

## 3.2 Hyperparameter Optimization

A special case of AAC is hyperparameter optimization (HPO) [50], in which the hyperparameters of an ML model are configured. Just like in standard AAC, HPO focuses on optimizing a black-box function using only the input (hyperparameters) and output (performance metrics after training) of that function. HPO is characterized by a couple of aspects. Some of them also occur in AAC problems, but in HPO, they always occur together. First, evaluating a single instance for a configuration is very expensive since a model needs to be trained to evaluate it. Depending on the model and data size, this can take up to several days. Second, HPO does not have multiple problem instances since HPO aims to optimize a model for a single dataset instead of a problem class. Multiple instantiations of the dataset are generally created using k-fold cross-validation. However, this is limited, and the handful of created instances have a smaller problem class distribution than many problems in standard AAC. Finally, many ML models train iteratively, training on the same dataset for multiple epochs.

Formally, the problem is very comparable to AC. HPO is a black-box optimization method with the main goal of minimizing the expected generalization error as defined in Equation 2.1:

$$\hat{\lambda} \in arg \min_{\lambda \in \Lambda} \hat{GE}(\mathcal{I}, \mathcal{J}, L, \lambda). \tag{3.2}$$

### 3.2.1 HPO Methods

Due to these differences in HPO, different methods excel in HPO, and others have been developed for HPO. The most naive method is grid search, in which a user specifies a range of possible values for each hyperparameter. Grid search then evaluates the performance of all the combinations and returns the best hyperparameter configuration. While this simple method might seem promising at first, it suffers from several problems. It requires the user to specify potential values, it uses exponentially more function evaluations as the number of hyperparameters grows, it might evaluate many configurations that are even worse than the default configuration [114], and it might waste resources exploring unimportant hyperparameters.

As a response, another popular method for HPO is random search [6]. Random search samples configurations randomly from the configuration space until a stopping criterion is met and the best-found configuration is returned. This simple technique does not need user input, scales better with more hyperparameters, and better covers the hyperparameter space if unimportant hyperparameters are in play. This last advantage is shown in Figure 3.2, where it is shown how a random configuration can better find optima in an equal number of evaluations. This is only the case if a random search uniformly samples the hyperparameter space, which is often ensured with extra precautions such as Latin Hypercube Sampling [77]. Random search acts as a strong baseline for other HPO methods [9] and is a popular choice due to its simplicity.

A problem of grid and random search is that they do not learn from evaluated configurations. Instead, guided search methods could be used, which use previous evaluations to determine new configurations. Inspiration can be taken from the general AAC field, and indeed, we see that many guided search methods use Bayesian optimization [105], such as the HPO implementation of SMAC, SMAC4HPO [71]. In reality, most guided search methods can outperform random search [50].

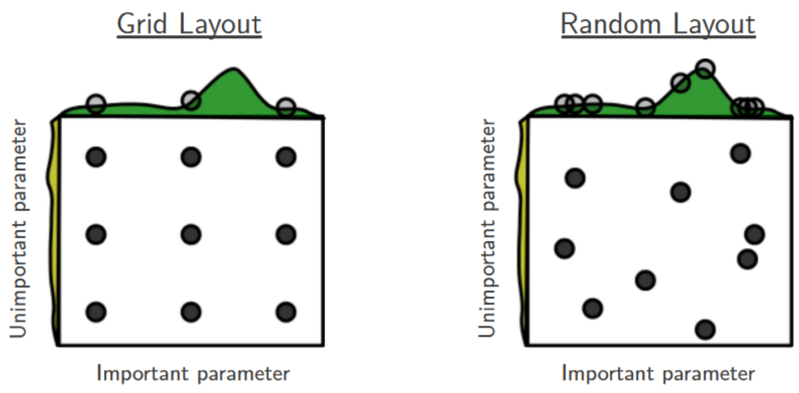


FIGURE 3.2: Grid and random search of nine trials for optimizing a function with two parameters, of which one has little importance. With grid search, only three values of the important variable are explored, while random search explores nine distinct values of the important variable. Figure taken from [6].

### 3.2.2 Multi-Objective Optimization

An unmentioned problem of HPO lies in its objectives. ML models are generally trained to optimise their accuracy on the prediction task, but this is often not the only goal. We might also be interested in minimizing the misclassification rate for medical applications or the inference efficiency for small and efficient models. This introduces the problem of multi-objective HPO (MO-HPO). Adding more objectives to the optimization algorithm is not trivial, as objectives are often conflicting. For example, it is impossible to reach the optimal efficiency of 0 FLOPs while reaching an accuracy of 100%. Therefore, a *decision maker* must determine a trade-off between the objectives. One can choose to determine this trade-off *a priori* by defining preferences and scaling the objectives. This turns the problem into a single objective optimization problem [78] and allows us to simply reuse existing HPO methods. However, defining the weight or importance of an objective without knowing the underlying trade-off is often very difficult.

#### Pareto Optimality

Due to the unknown underlying trade-off, MO-HPO methods often return not one but multiple solutions. With conflicting objectives and an unknown trade-off, multiple equally good solutions exist. To understand this, we explain the concept of Pareto dominance using the notation of [57]. Given two vectors $c$ and $c'$ in an objective space of size $m$, we say that $c$ dominates $c'$ (i.e. $c \prec c'$) if and only if

$$\begin{aligned}&\forall i \in \{1, \ldots, m\} : c_i \leq c'_i \land \\ &\exists j \in \{1, \ldots, m\} : c_j < c'_j.\end{aligned} \tag{3.3}$$

We use the same term to denote the relation between the hyperparameter configurations $\lambda$ and $\lambda'$ corresponding to the objective vectors $c$ and $c'$, i.e. $\lambda$ dominates $\lambda'$. In MO-HPO, we are looking for Pareto optimal configurations, i.e. those that are not dominated by any other configuration. When dealing with conflicting objectives, the following situation can cause $c$ and $c'$ to both be non-dominated solutions:

$$\exists i, j \in \{1, \ldots, m\} : c_i < c'_i \land c'_j < c_j. \tag{3.4}$$

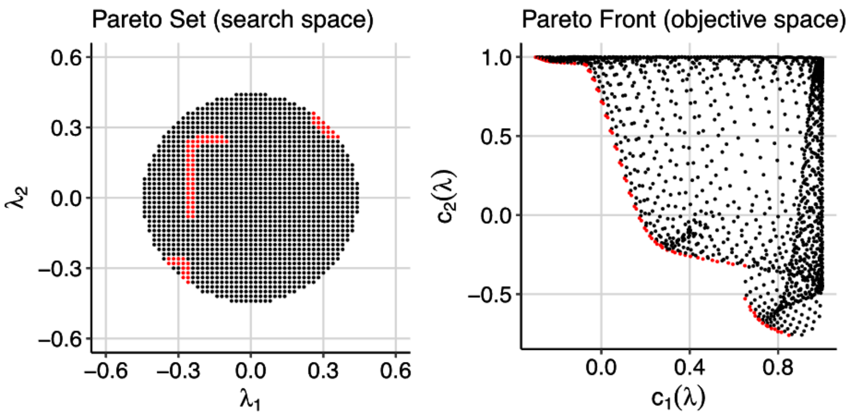


FIGURE 3.3: Illustration for a two-dimensional MOO problem with two objectives. The left plot shows the search space, and the right plot shows the objective space. Non-dominated configurations (the estimated Pareto set (left)) and their mapping to the co-domain (the estimated Pareto front (right)) are highlighted. Taken from [57].

The set of Pareto optimal configurations is called the Pareto set, and its corresponding set of objective scores is called the Pareto front. The Pareto front shows the underlying trade-off between the conflicting objectives, and to allow for *a posteriori* decision making, the goal of a MO-HPO method is to return a Pareto set that best approximates the Pareto front. Figure 3.3 shows an example Pareto set front for a 2-dimensional optimization problem. There are more advantages to having MO-HPO estimate the Pareto front next to informed decision-making. Unknown underlying trade-offs can be discovered, regions of the Pareto front can be found that scalarization methods cannot reach [23], and better results for individual objectives can be achieved, as MO-HPO is less likely to get stuck in local optima [60].

### MO-HPO methods

This subsection further introduces some MO-HPO problems and solutions, but we direct the reader to a recently published survey by Karl et al. [57] for a complete overview. MO-HPO can be approached in several ways. The basic grid search and random search can be easily adapted to solve MO-HPO problems by having them return all non-dominated solutions, and they can serve as the same baseline they do for single-objective HPO [57]. Alternatively, we can perform MO-HPO by scaling the objectives. While a prior scalarization does not result in a trade-off with multiple solutions, this can be achieved using online scalarization. The most popular scalarization method is ParEGO [59]. ParEGO is an extension of Bayesian optimization, in which the objectives are differently scaled in each iteration to ensure that the Pareto front is explored evenly. This allows us to use any single-objective HPO method based on Bayesian optimization to solve a MO-HPO problem with multiple solutions.

Another class of MO-HPO methods are MO evolutionary algorithms (MOEAs). These algorithms simultaneously evaluate a (often randomly initialized) set of configurations and generate new configurations by combining and perturbing the best configurations. EAs are not commonly used in the general HPO field as they require relatively many function evaluations [57]. However, due to their easy adaptability to multi-objective optimisation, they are very popular in the MO-HPO field [18, 24].

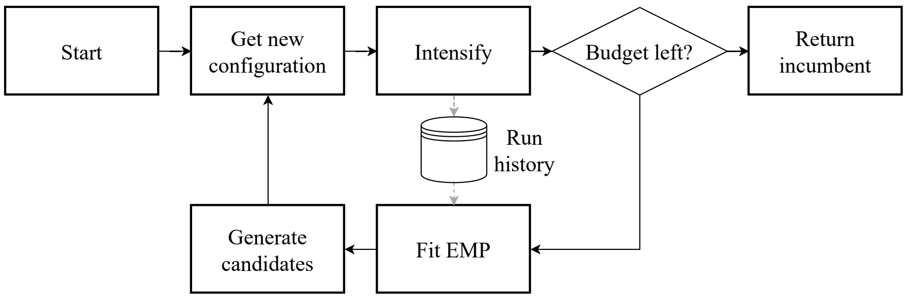


FIGURE 3.4: Overview of the process of (MO-)SMAC.

Furthermore, many MO-HPO methods have been created by extending popular single-objective HPO methods to the multi-objective setting, resulting in some of the best MO-HPO algorithms. Examples are MO-ParamILS [12], a multi-objective extension of ParamILS [49], and MO-SMAC [95], a multi-objective extension of SMAC [71]. To better understand how an HPO method can be extended to MO-HPO, we will give a broad explanation of the design of MO-SMAC. For a more in-depth explanation of all design choices, we direct the reader to [95].

### MO-SMAC

First, we further introduce SMAC. A global overview of the internal workings of SMAC can be found in Figure 3.4. SMAC keeps track of the best configuration (the incumbent), and a history of configurations and the objectives of the instances on which they have been evaluated. SMAC starts with a default configuration and evaluates this on an instance. From this point on, an iterative process starts. A model (the EMP) is fitted on the run history to predict the performance of unseen configurations. Then, new configurations are generated using a combination of random search and local search. These configurations are scored with an acquisition function that uses the EMP. A common choice is the expected improvement, balancing exploration and exploitation. The best configurations are evaluated on instances and compared against the incumbent in the intensification step. If a new configuration is better than the incumbent, it becomes the new incumbent. Configurations are not evaluated on multiple instances if they are not competitive with the incumbent. This process of fitting the EMP, generating configurations, scoring configurations and evaluating configurations is repeated while there is budget left, after which the incumbent is returned.

MO-SMAC changes a few steps of this process but globally consists of the same steps as in Figure 3.4. First, it keeps track of more than one incumbent configuration to approximate the Pareto front. Second, it uses the predicted hypervolume improvement as acquisition function to estimate how a new configuration can improve the approximated Pareto front. Third, it fits not one but multiple EMPs, one per objective. Finally, the intensification step is changed. Newly proposed configurations are compared against the incumbent configuration closest to them by Euclidean distance. If a proposed configuration is non-dominated after evaluation, it is added to the incumbent. If needed, other configurations are removed from the incumbent.

### 3.2.3 HPO and Efficiency

Next to optimising the performance of an ML model, HPO can be used to optimise its efficiency. In the remainder of this section, we explore studies that have used HPO for similar goals as ours, i.e. to optimize SNNs and to perform MO-HPO on similar objectives.

#### HPO on SNNs

To the best of our knowledge, no research has been conducted to apply HPO to static or dynamic SNN training. However, HPO has been used in other sparse methods, such as structured pruning. Structured pruning is a special type of pruning in which dense network topologies are pruned such that the resulting topology remains dense. For example, AutoCompress [72] is an AutoML structured pruning framework that optimizes network sparsity and accuracy. Their method can find parameter configurations that achieve an accuracy-efficiency trade-off that could not be reached before.

Huang et al. [46] perform MO-HPO on pruned architectures using two separate MOEAs. One optimizes the network architecture, while the other optimizes pruning hyperparameters. Combined, they find architectures that have improved accuracy and efficiency than other optimized methods.

Matthiesen [76] studied the relationship between sparsity, learning rate and weight decay in structured pruning using HPO. By making these last two hyperparameters configurable and performing hyperparameter dependence analyses, they found that the learning rate and weight decay behave similarly when a DNN is trained with structured pruning.

Binder et al. [8] use MO optimization for HPO and feature selection. While they do not optimize SNNs, they do show that MO Bayesian optimization can be used to find efficient architectures.

#### MO-HPO for Efficiency

MO-HPO has been used on many occasions, and many implementations use it to optimize an objective related to efficiency. Bischl et al. [9] listed over thirty such applications, of which a handful use a Bayesian optimization optimizer. Here, we highlight the two works most comparable to our work.

Parsa et al. [89] present PABO. PABO performs Bayesian optimization to propose configurations for accuracy- and efficiency-related objectives and combines these into a single configuration using a supervisor agent. They compare their method against the multi-objective evolutionary algorithm NSGA-II [18] on three MO-HPO problems. In these problems, the dataset and architecture type (AlexNet [62] or VGG19 [101]) are defined, and the network's hyperparameters (including its shape and size) are optimized. All hyperparameters are coded as categorical parameters, in which a few options are possible to keep the configuration space small. Due to these smaller configuration spaces, they can compute the true Pareto front for one experiment and show that PABO finds a close approximation of the Pareto front. Furthermore, they show that PABO, a Bayesian optimization-based method, can approximate the Pareto front in substantially fewer configuration evaluations than NSGA-II. For example, in a search space of 6912 possibilities, PABO used 33 evaluations, while NSGA-II used 6000 evaluations.

Abdolshah et al. [2] present MOBO-PC, a method for multi-objective Bayesian optimization with preference-order constraints. Preference-order constraints can be given in the form of 'objective A is more important than objective B', causing MOBO-PC to estimate only a fraction of the Pareto Front. To show the capabilities of MOBO-PC, they perform a couple of experiments and compare the estimated Pareto fronts to those found by 'normal' Bayesian optimization methods, such as ParEGO [59], and another preference-based method, MOBO-RS [88]. In one of their experiments, they optimize a neural network's hyperparameters (including its shape and size) for accuracy and efficiency on the MNIST dataset with budgets of 100 and 200 evaluations per optimizer. They show that all 'normal' optimizers find very similar trade-offs with a budget of 200 evaluations. Furthermore, both preference-based optimizers find similar estimations of the restricted Pareto front. Interestingly, both preference-based optimizers find configurations with better accuracy and efficiency than the other optimizers. However, this study cannot use these optimizers as we wish to study the entire trade-off.

## 3.3 Hyperparameter Importance

HPO is often performed to find one of the best hyperparameter configurations for a given model and setting. However, sometimes, finding the best configuration is not our goal. Instead, we would like to gain a better understanding of the algorithm and its objective(s) to find directions for future research and development. Such results are generally achieved by evaluating hyperparameter importance (HPI). HPI research has two primary goals. First, pinpointing which hyperparameters are most important, i.e. which hyperparameters in the hyperparameter configuration space have the most significant impact on the objective space. The second goal is researching hyperparameter influence, i.e. discovering which values of these hyperparameters have the most potential of resulting in a desired area of the objective space. The results of HPI can show that certain hyperparameters have little effect and do not need to be optimized or that hyperparameters have more influence than previously thought. Furthermore, it can be used to find robust defaults for good overall performance or to discover complex relationships between different hyperparameters. These results can be extracted from evaluated configurations of HPO and are very useful for future research. However, reaching them is not trivial due to the complex and often unknown relationships between multiple hyperparameters. An optimizer might change hundreds of hyperparameters between two evaluations to find very different results, but just a handful of changed hyperparameters might cause these results [27]. Furthermore, many approaches carry the risk of finding biased results as HPO converges to and focuses on local optima [83].

### 3.3.1 Single-Objective HPI

Much research has been performed into the possibilities of HPI in the single-objective optimization field, starting in 2014 using the functional analysis of variance (fANOVA) framework [47], a method to efficiently derive sensitivity indices for random forest models. Random forest models are a popular choice of surrogate models in Bayesian optimization, where they model the relationship between hyperparameters and an objective. These sensitivity indices can be used for fANOVA, modelling the effects of hyperparameters such that they can quantify the relative importance of each hyperparameter. Another popular method to find the most important hyperparameters is ablation [27, 7]. Given two hyperparameter configurations (e.g. defaults vs optimized), ablation constructs an *ablation path* of configurations. These configurations represent small individual changes between the two given configurations and are evaluated to find the individual influence of each hyperparameter.

Knowing how important a hyperparameter is is the first step in HPI research. With this knowledge, researchers can choose not to optimize a certain hyperparameter and use a default value instead. Probst et al. [91] performed a large-scale HPI study on the hyperparameters of many popular ML models. By averaging the results of various datasets, they measure the 'tunability' of each hyperparameter. The less tunable a hyperparameter is, the more sense it makes to leave it at its default value. Furthermore, they use the same results to define new robust default values for some hyperparameters.

A final aspect of hyperparameter importance is knowing how an important hyperparameter influences the objective. Intuitively, an important hyperparameter is found to drastically improve the objective in some ranges and reduce it for others. Or, important hyperparameters yield good results in certain combinations while yielding much worse results in other combinations. An approach to show such results is with partial dependence plots. Moosbauer et al. [83] use the surrogate model of Bayesian optimization to create partial dependence plots of optimized hyperparameters reliably. This way, they visualize the influence of hyperparameters on their objective.

### 3.3.2 Multi-Objective HPI

Similar research can be performed using the insights gained from multi-objective optimization. Here, the results found in the objective space can be combined with the respective configurations in the decision space. From this, we can discover which variables influence the trade-off between our objectives and how they influence it. Little research has been put into multi-objective hyperparameter importance, but multi-objective adaptations of fANOVA and ablation have been introduced recently [109]. In these adaptations (MO-fANOVA and MO-Ablation), several scales for all objectives are calculated based on the trade-offs in the approximated Pareto front. Each scale is combined with the objectives to create a single objective on which single-objective fANOVA or ablation can be performed. Afterwards, these results can be combined to show the importance of hyperparameters as one changes the importance of the objectives.

Next to these methods, relevant research for multi-objective HPI can be found in the field of a posteriori multi-criterion decision-making (MCDM). MCDM methods are often separated into implicit and explicit methods. Explicit knowledge is presented as structured and comparable measures, such as descriptive data statistics (e.g. mean and standard deviation) [5]. On the other hand, implicit knowledge cannot be systematically compared and is generally presented using visualization methods such as [102].

Visualization techniques are the most popular techniques for the presentation of knowledge in MCDM. However, they often only look at the objective space [103]. To look at the importance of variables to the different solutions of a multi-objective problem, we must look at both the objective and the decision space. For such results, biclustering [110] combined with explicit knowledge statistics or Trend Mining 2.0 [103] could be used.

Unfortunately, many of these methods have been created for and studied in the general field of multi-objective optimization, not MO-HPO. Due to expensive evaluations of ML, MO-HPO results in much fewer evaluations than these methods expect. Therefore, more work is needed to better support multi-criterion decision making for hyperparameter importance research.

Chapter 4

# Methodology and Experimental Setup

This chapter works towards a set of experiments with which we uncover the accuracy-efficiency trade-off by listing and explaining specific methods used in these experiments. We start by defining our study's specific network architectures, resulting in the complete hyperparameter configuration space. Then, we concretely define our objectives and introduce an important statistic, the hypervolume indicator. Finally, we introduce the experiments and experimental setup.

## 4.1 Neural Network Architectures

In our configurable environment, we wish to work with several types of network architectures. Finding such well-working neural network architectures is a challenging and complex task in itself and can be tackled with a neural architecture search [25]. However, this would create too large of a search space for this study, and we wish to limit ourselves to a small number of hyperparameters that define the architecture. Therefore, we choose two popular neural architecture types that allow for variable shapes and sizes with a few hyperparameters. Similar to the networks studied by Mocanu et al. [82] and Evci et al. [26], we allow for two different neural network types, multi-layer perceptions (MLPs) and residual networks (ResNets) [37].

A technical introduction to (sparse) MLPs can be found in Subsection 2.1.3. In SNN training, all dense network layers can be made sparse by removing weights. We define the shape of an MLP by four hyperparameters: the number of layers, the size of the first layer, the size of the last layer and the size of the middlemost layer. The layers between these three layers are linearly scaled in size. If a shallow neural network of only one or two layers is chosen, the last and middlemost layers are not considered, respectively. With this setup, four hyperparameters can define many different MLPs in detail. An illustration of how this works can be found in Figure 4.1.

ResNets are a type of convolutional neural network and excel in deep learning for image recognition [37]. They can handle multi-dimensional input and learn patterns irrespective of their location in the input. Unlike MLPs, ResNets consist of several different types of layers: convolutional, pooling, and dense. Convolutional layers learn patterns in multi-dimensional input. Pooling layers change the dimensions of the input. Dense layers are the same as in MLPs. In SNN training, the parameters of convolutional and dense layers can be sparsified, and pooling layers are usually ignored. In a ResNet, the convolutional layers are placed in blocks of variable sizes, followed by pooling layers that result in fixed-size outputs.

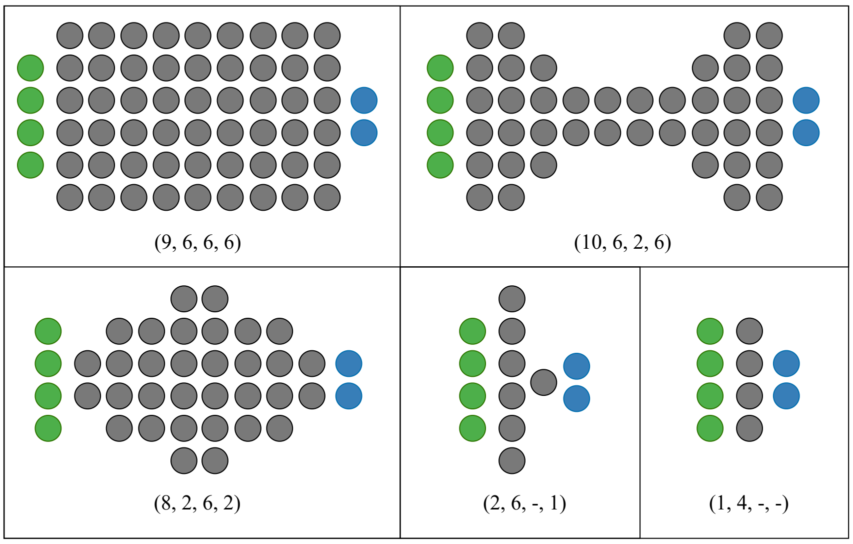


FIGURE 4.1: Illustration of the range of possible network shapes if we have 4 features and 2 classes. Shapes are determined as (Num MLP layers, Size first MLP layer, Size middle MLP layer, Size last MLP layer).

TABLE 4.1: Summary of the ResNet model architecture. The number of layers in all convolutional block are variable. Identity shortcut connections [37] are used within the convolutional blocks.

| layer name | output size | layer structure |
|---|---|---|
| conv0 | $32 \times 32$ | $3 \times 3$, 16, stride 1 |
| conv1_x | $32 \times 32$ | $[3 \times 3, 16] \times$ conv stage 1 |
| conv2_x | $16 \times 16$ | $[3 \times 3, 32] \times$ conv stage 2 |
| conv3_x | $8 \times 8$ | $[3 \times 3, 64] \times$ conv stage 3 |
| dense | $1 \times 1$ | average pool, dense layer size 10, softmax |

In our implementation, we allow for $32 \times 32 \times 3$ input images and use a ResNet architecture consisting of three blocks as detailed in Table 4.1. This is a similar architecture as used on the CIFAR-10 experiments in [37]. Three hyperparameters determine the number of convolutional layers per block.

### 4.1.1 Sparse Training Methods

We will use two dynamic SNN training algorithms, RigL [26] and SET [82], both further detailed in Figure 2.2.2. RigL is chosen as the state-of-the-art SNN training algorithm, and its introductory paper lists clear accuracy-efficiency trade-offs in multiple experiments and methods. Furthermore, we select SET for its simplicity, allowing us to trade off inference and training efficiency between different SNN training algorithms. As a baseline, we also include dense training. We use a set of four hyperparameters to initialize RigL and SET in various ways.

### 4.1.2 Hyperparameter Configuration Space

In addition to the shape and sparsity-related hyperparameters, we let neural networks be optimized using the learning rate, momentum, weight decay, dropout, and label smoothing hyperparameters. We set ranges to all hyperparameters to search for optimizations within this hyperparameter configuration space. These ranges have been chosen to capture the entire configuration range for sparsity hyperparameters, create models that fit in memory for shape hyperparameters and include typical values for training hyperparameters. Some hyperparameters are scaled logarithmically if it is known that they are better optimized that way. Dropout and label smoothing should not always be used and are en/disabled by auxiliary hyperparameters. In total, this results in a configuration search space of 23 hyperparameters. Their specifics can be found in Table 4.2.

## 4.2 Objectives

In all experiments, we train (sparse) neural networks and evaluate them on three objectives: accuracy, inference efficiency, and training efficiency. This section further elaborates on how we define and calculate these objectives.

### 4.2.1 Accuracy

In classification problems, a model's classification accuracy is defined as the percentage of samples correctly predicted by a trained model. In our experiments, we optimize for validation accuracy and report test accuracy. Both are calculated by evaluating a model on an unseen dataset after training the model for a fixed number of epochs on a training dataset. Since optimization is performed by minimizing the objectives, we often use the error rate ($1-$accuracy) instead of accuracy.

### 4.2.2 Efficiency

As stated in Chapter 2, a popular unit of measurement for network efficiency is the number of FLOPs needed for a computation. It allows comparison to previous studies and, to our knowledge, is the best hardware and implementation independent unit of measure relating to ML sustainability. Unfortunately, it does not always directly translate to the sustainability of a model. As a solution, we define efficiency by the number of floating point multiplication operations needed in a computation. Out of all floating point operations, these are the least energy-efficient [75], making them a better estimator of sustainability than FLOPs while still being hardware and implementation-independent. For the sake of simplicity, we still refer to the number of multiplications as FLOPs. We estimate the multiplications using Algorithm 1, for which we explain the individual components below.

#### Inference FLOPs

The inference FLOPs represent the number of FLOPs needed to pass a single input sample through the network, i.e. predict the output label $y$ of one input feature vector $\mathbf{x}$. Each layer needs a certain number of FLOPs to calculate an output from its input. When calculating the dense inference FLOPs, we can simply sum the FLOPs required for each layer. The sparse inference FLOPs only add a single term to this calculation. For all layer types, dense FLOPs are directly influenced by the number of parameters in that layer. Therefore, we can calculate the sparse FLOPs of a layer by multiplying the dense

Table 4.2: Hyperparameter configuration space of our implementation.

| Name | Type | Scaling | Range | Default |
|---|---|---|---|---|
| Sparsity hyperparameters | | | | |
| algorithm | categorical | - | [RigL, SET, Dense] | SET |
| *if algorithm $\neq$ Dense* | | | | |
| sparsity | real | linear | [0-1] | 0.5 |
| update frequency | integer | linear | [1-$10^6$] | 10 |
| update end | real | linear | [0.1-1] | 0.8 |
| sparsity distribution | categorical | - | [Uniform, ERK] | Uniform |
| Network shape hyperparameters | | | | |
| architecture | categorical | - | [MLP, ResNet] | MLP |
| *if architecture = MLP* | | | | |
| MLP layers | integer | linear | [1-10] | 3 |
| size first MLP layer | integer | linear | [1-1000] | 100 |
| *if MLP layers > 1* | | | | |
| size last MLP layer | integer | linear | [1-1000] | 100 |
| *if MLP layers > 2* | | | | |
| size middle MLP layer | integer | linear | [1-1000] | 100 |
| *if architecture = ResNet* | | | | |
| size conv block 1 | integer | linear | [1-4] | 2 |
| size conv block 2 | integer | linear | [1-4] | 2 |
| size conv block 3 | integer | linear | [1-4] | 2 |
| Training hyperparameters | | | | |
| initial learning rate | real | log | [$10^{-5}$, 1] | 0.01 |
| learning rate scheduler | categorical | - | [Constant, Cosine] | constant |
| momentum | real | log | [0-1] | 0.9 |
| weight decay | real | log | [$10^{-7}$, $10^{-2}$] | $10^{-3}$ |
| batch size | integer | log | [4-4096] | 256 |
| epochs | integer | log | [1-200] | 20 |
| use dropout | boolean | - | True/False | False |
| use label smoothing | boolean | - | True/False | False |
| *if use dropout* | | | | |
| dropout | real | linear | [0.1-0.5] | 0.3 |
| *if use label smoothing* | | | | |
| label smoothing | real | linear | [0.05-0.25] | 0.1 |

**Algorithm 1** Pseudocode of our efficiency calculation.

```
$flops_d \leftarrow 0$                                   ▷ Dense Inference FLOPs
$flops_s \leftarrow 0$                                   ▷ Sparse Inference FLOPs
for every layer $L$ do
    $L_{flops} \leftarrow 0$                             ▷ Layer FLOPs
    if $L$ is Dense layer then
        $L_{flops} \leftarrow$ input size $\times$ output size
    else if $L$ is Convolutional layer then
        $L_{flops} \leftarrow$ kernel width $\times$ kernel height $\times$ input filters
        $\times$output width $\times$ output height $\times$ output filters
    else if $L$ is Pooling layer then
        $L_{flops} \leftarrow 0$
    end if
    $flops_d \leftarrow flops_d + L_{flops}$
    $flops_s \leftarrow flops_s + L_{flops} \times L_{sparsity}$
end for
$flops \leftarrow 0$                                     ▷ Total FLOPs per train step
if algorithm = Dense then
    $flops \leftarrow 3 \times flops_d$
else if algorithm = SET then
    $flops \leftarrow 3 \times flops_s$
else if algorithm = RigL then
    $flops \leftarrow (3 \times flops_s \times \Delta T + 2 \times flops_s + flops_d) \div (\Delta T + 1)$
end if
return $flops_s$ and $flops \times$ epochs $\times$ train samples      ▷ Total Inference and Training FLOPs
```

FLOPs with the layer sparsity. Note that we cannot calculate the sparse inference FLOPs of the entire network from the dense inference FLOPs, as sparsity is not always uniformly distributed over all layers.

With a formula for the dense and sparse inference FLOPs, all that remains are formulas to calculate the dense FLOPs for each layer type. This study allows MLP and ResNet architectures consisting of dense, convolutional and pooling layers.

### Dense Layers

Dense layers consist of several neurons as introduced in Subsection 2.1.3. Each neuron output is calculated by multiplying all input values by corresponding weights, taking the sum of these values, and performing a non-linear operation on this result. This non-linear operation generally consists of a negligible number of multiplications and can be omitted from the FLOPs calculation. Therefore, the number of multiplications in a dense layer is equal to the number of weights in that layer and can be calculated by multiplying the input size with the number of neurons of a layer, i.e. its output size. This results in the following formula:

$$flops_{dense} = \text{input size} \times \text{output size}. \tag{4.1}$$

### Convolutional Layers

Convolutional layers have in- and output in three dimensions: width × height × filters. For each output point, a two-dimensional kernel is applied to an area of each input filter. Applying such a kernel to one area needs a number of multiplications equal to the number of parameters in that kernel, i.e. kernel width × kernel height. Since we apply this to each input filter, we multiply this by the number of input filters. Then, we multiply this by the number of output points, resulting in the following formula:

$$\begin{aligned} flops_{convolution} = \;& \text{kernel width} \times \text{kernel height} \times \text{input filters} \\ & \times \text{output width} \times \text{output height} \times \text{output filters}. \end{aligned} \tag{4.2}$$

### Pooling layers

A pooling layer calculates its output by aggregating its input over small areas. Generally, these calculations are simple, do not significantly add to the total FLOP count and are often omitted when calculating network efficiency. Furthermore, the number and sizes of pooling layers in our networks are fixed, and they are kept dense in our implementation, causing their efficiency to be a constant value independent of the hyperparameter configuration used. Therefore, we omit pooling layers when calculating inference FLOPs by setting their FLOPs to 0.

### Training Efficiency

Similar to the inference efficiency, we measure training efficiency by the number of multiplications needed to train a neural network. To compute these FLOPs, we use formulas introduced in [26]. These formulas use the sparse and dense inference FLOPs to estimate the total FLOPs needed to train a network. During training, each sample in the training dataset is used to update the network layers once per epoch. Such an update step consists of the forward pass and backward pass. In the forward pass, the sample is passed through the network once to calculate its loss, costing the inference FLOPs as previously computed. In the backward pass, this loss is used to calculate the gradients of all parameters and activations, costing approximately two times the inference FLOPs.

#### Dense Training

Dense neural networks add no overhead to these update steps. Therefore, its training FLOPs scale with:

$$flops_{training}^{dense} = 3 \times flops_{inference}^{dense}. \tag{4.3}$$

#### SET

When training a network with SET, random connections are grown every $\Delta T$ update steps. In their work, Evci et al. [26] argue that this operation can be done on chip efficiently, causing the training FLOPs to scale with:

$$flops_{training}^{set} = 3 \times flops_{inference}^{sparse}. \tag{4.4}$$

#### RigL

A network trained with RigL grows connections based on the dense gradients every $\Delta T$ update steps. This does add substantial computational overhead when $\Delta T$ is small and should, therefore, be included in our calculation. This overhead is estimated as $2 \times flops_{inference}^{sparse} + flops_{inference}^{dense}$ every $\Delta T$ update steps. This results in the following formula for the average FLOPs per update step:

$$flops_{training}^{rigl} = \frac{(3 \times \Delta T + 2 \times flops_{inference}^{sparse} + flops_{inference}^{dense})}{(\Delta T + 1)}. \tag{4.5}$$

### 4.2.3 Objective Scaling

With these definitions of inference and training efficiency, we can find variability in orders of magnitude in our objective space. This can become problematic when comparing the efficiency of different configurations. For example, suppose we have three different configurations with inference efficiencies of $10^7$, $10^5$ and $10^4$ FLOPs. In that case, we want to capture the vast efficiency improvement between the first two configurations, as well as the vast improvement between the latter two. To properly capture this, we often take the logarithm of the two efficiency objectives before visualization or computation of performance indicators that combine multiple objective values.

## 4.3 Hypervolume

Next to showing the accuracy-efficiency trade-offs themselves, we will often compare different trade-offs with each other. This can be done visually, but also by quantifying the trade-offs to a comparable measure. For this, we always use the hypervolume [33] of the approximated Pareto front. The hypervolume is the size of the space between all non-dominated objectives and a reference point outside of this objective space. It is often used to assess the quality of MO optimization as a better approximation of the Pareto front results in a larger hypervolume. Similarly, we can use it to compare the results of our experiments. First, we take the logarithm of both efficiency objectives, after which we normalize all objectives between 0 and 1 by the minimum and maximum objective in the observed trade-off to account for different objective scales. Afterwards, we can calculate the hypervolume of each resulting trade-off. We always use the reference point (1.1, 1.1, 1.1). This is an arbitrary point just outside the normalized objective space, so no point in the objective space has a hypervolume of 0. This results in a minimal hypervolume of $(1.1 - 1.0)^3 = 0.1^3 = 0.001$ and a maximal hypervolume of $1.1^3 \approx 1.331$.

We use the hypervolume to compare experiments in two ways. First, we can use it to compare the trade-offs between two datasets. Since all objectives are normalized before calculating the hypervolumes, they can be directly compared to each other to indicate the trade-off shape. A larger hypervolume implies a stabler accuracy when increasing the efficiency. Second, we can use it to compare the trade-offs found between two experiments on the same dataset. In this case, we normalize with the extremes of both experiments, and a larger hypervolume indicates a better approximation of the Pareto front.

## 4.4 Experimental Setup

### 4.4.1 Experiments

We define three different experiments. These experiments aim to uncover the underlying accuracy-efficiency trade-off in SNN training in a novel method, compare this method to the literature and verify the validity of our method. To start, we perform a reproduction experiment, resulting in a trade-off similar to the results presented in the literature. The results of these experiments serve as a baseline for the rest of the study. Second, we perform an optimization experiment on a hyperparameter configuration search space, allowing configurations similar to the fixed configurations in the reproduction experiments. By using MO-HPO in a broader environment, we can find a more general trade-off between our objectives according to the goal of this study. We compared these results to those of the reproduction experiments to study **RQ1.1**. As a final step to study the difference between using fixed configurations and MO-HPO, we perform an evaluation experiment on a subset of the optimization results. We compare these results with the optimization and reproduction trade-offs to conclude **RQ1**. Together, these experiments relate to each other as visualized in Figure 4.2.

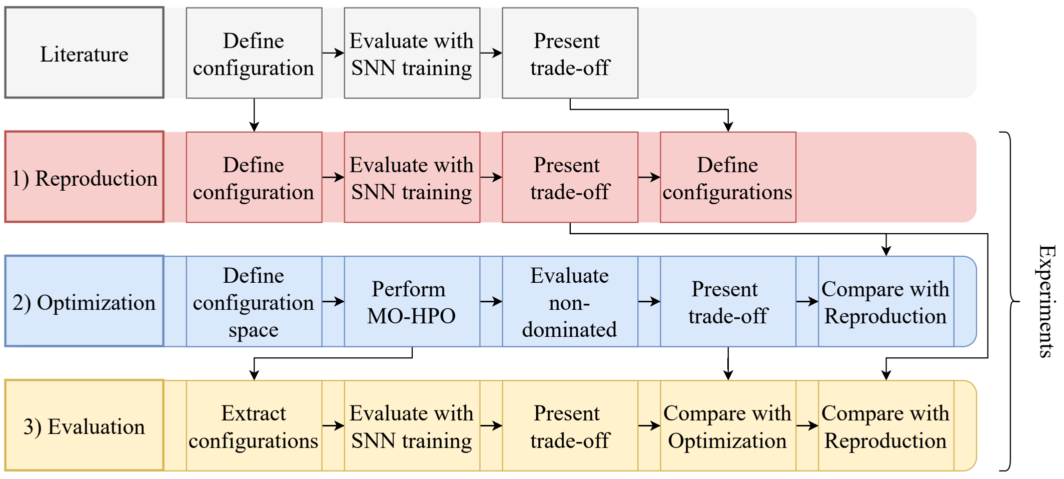


FIGURE 4.2: Overview of the different performed experiments and their relations.

Table 4.3: Chosen benchmark datasets.

| Dataset | Domain | Data Type | Features | Classes | Train Size | Test Size |
|---|---|---|---|---|---|---|
| MNIST [64] | Digits | Images | 784 | 10 | 60,000 | 10,000 |
| Fashion MNIST [117] | Products | Images | 784 | 10 | 60,000 | 10,000 |
| Higgs [116] | Particles | Tabular | 28 | 2 | 10,500,000 | 500,000 |
| Elec2 [35] | Power prices | Tabular | 8 | 2 | 26,932 | 11,542 |
| CIFAR-10 [61] | Objects | Images | 3,072 | 10 | 50,000 | 10,000 |
| SVHN [87] | Digits | Images | 3,072 | 10 | 73,257 | 26,032 |

### 4.4.2 Datasets

We use six benchmark datasets containing images and tabular data, allowing us to compare our results to previous SNN research and generalize to several problem cases (**RQ1.2**). Three of these have been used for experiments in SNN training literature: MNIST and Higgs in [82] and CIFAR10 in [26]. The MNIST dataset is a simple image classification problem with ten classes, the CIFAR10 dataset is a more complicated image classification problem with ten classes, and the Higgs dataset is a typical tabular classification problem with two classes for which we know that it is difficult to model with a neural network [32]. To generalize our results in these problem settings, we add three datasets comparable to these first three, resulting in three dataset pairs: Fashion MNIST for MNIST, Elec2 for Higgs and SVHN for CIFAR-10. Specifics of these six datasets are detailed in Table 4.3.

Data augmentation in the form of random horizontal flips and random translations of at most 4 pixels in all directions are included in the training pipelines of CIFAR-10 and SVHN. All datasets are classification problems with balanced class distributions. We split all datasets by their recommended train and test split. If no test split was available, we randomly sampled 70% for training and 30% for testing.

### 4.4.3 MO-HPO

To approximate the accuracy-efficiency trade-off, we perform MO-HPO with MO-SMAC [95]. MO-SMAC requires relatively few function evaluations to approximate a Pareto front, which is necessary for our setting, where evaluating a single configuration can take a long time. MO-SMAC has many possible settings; we keep most of these at their default values except for the maximum number of incumbent configurations to keep track of. By default, this is set to ten, but this limit is often reached when working with more than two objectives, limiting MO-SMAC's ability to approximate the entire Pareto front. Therefore, we set MO-SMAC to track at most twenty incumbent configurations. Furthermore, we split our training datasets using 5-fold cross-validation and let MO-SMAC use these folds as problem instances to prevent overfitting. Each run starts with a default configuration, with default values defined in Table 4.2. MO-HPO optimization runs are usually run several times as MO-HPO is a stochastic process that can get stuck in local optima. We repeat each optimization run ten times with different seeds, after which we aggregate the results. A schematic overview of how MO-SMAC is used in our experiments can be found in Figure 4.3.

### 4.4.4 Implementation

We created a fully configurable sparse training environment in Python using Jax [14]. The environment loads data using Tensorflow Datasets [1], defines a network using Flax [39]

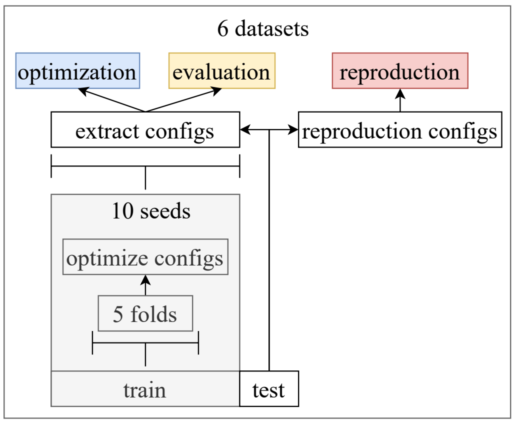


Figure 4.3: Overview of how the datasets reach results of the different experiments. Configurations are optimized on the folded train set using MO-SMAC. Selected configurations are evaluated on the complete train and test set with 5 seeds.

and properly trains this with Optax [19] and JaxPruner [67]. After training, the validation accuracy, inference efficiency and training efficiency are returned and logged. Our code and results are publicly available on GitHub [1].

The experiments were run on the EEMCS High-Performance Computing Cluster of the University of Twente. We used different compute nodes with Tesla P100, Tesla A100, Tesla A40, Tesla L40, Titan-X, GeForce gtx-1080ti, GeForce rtx-2080ti and Quadro rtx-6000 GPUs. These hardware differences cause differences in the training times but do not affect any of our objectives. In total, all experiments utilized approximately 7500 CPU hours and 5000 hours of GPU wall-clock time.

[1] https://github.com/zwouter/sparse-training-environment/tree/main.

# Chapter 5

# Uncovering the Trade-Off

This chapter focuses on our first research question: What does the accuracy-efficiency trade-off in sparse neural network training look like in a configurable environment? We further explain the specifics of the three experiments introduced in the previous chapter, after which we show and interpret each experiment's results in individual sections.

## 5.1 Reproduction Experiment

To start our reproduction experiment, we need a well-working hyperparameter configuration for each dataset. We choose configurations similar to the fixed hyperparameters used for MNIST, Higgs and CIFAR10 in [82] and [26]. The other three datasets are trained on the same configurations as their most comparable dataset. An overview of these configurations can be found in Appendix A. Within these configurations, we evaluate 15 different sparsities for SET and RigL. Furthermore, we add one configuration that does not use SNN training. This results in 31 configurations per dataset. This approach is similar to how Evci et al. [26] evaluated RigL against other SNN training methods and should yield somewhat comparable accuracy-efficiency trade-offs.

### 5.1.1 Results

Plots of all sparsity-accuracy trade-offs resulting from the reproduction experiments can be found in Figure 5.1. A more detailed overview of the results for the MNIST dataset is shown Table 5.1. All reproduction experiments show results as expected. Sparsely trained networks consistently yield comparable or slightly worse accuracies than their dense counterparts for a sparsity below 0.7. Starting from sparsities of 0.8 or higher, accuracy can substantially drop. In some cases of MLP networks, this substantial drop is more considerable when SET is used. For the Elec2 dataset, using SET with a sparsity of 0.99 even drops the accuracy to a random 50%, probably due to this dataset's small number of features. Furthermore, we observe less variance in the accuracies of the datasets used in previous literature. We used fixed hyperparameter configurations initially designed for the MNIST, Higgs and CIFAR10 datasets and reused these on the Fashion MNIST, Elec2 and SVHN datasets. This could mean that sparsity behaves better in an already optimized environment or that the datasets used in previous studies are better suited for sparse neural networks.

Due to configuration and implementation differences, we do not expect these results to align perfectly with the results of previous studies. This is also not a prerequisite, as the reproduction experiment only serves as a baseline for our later experiments, which

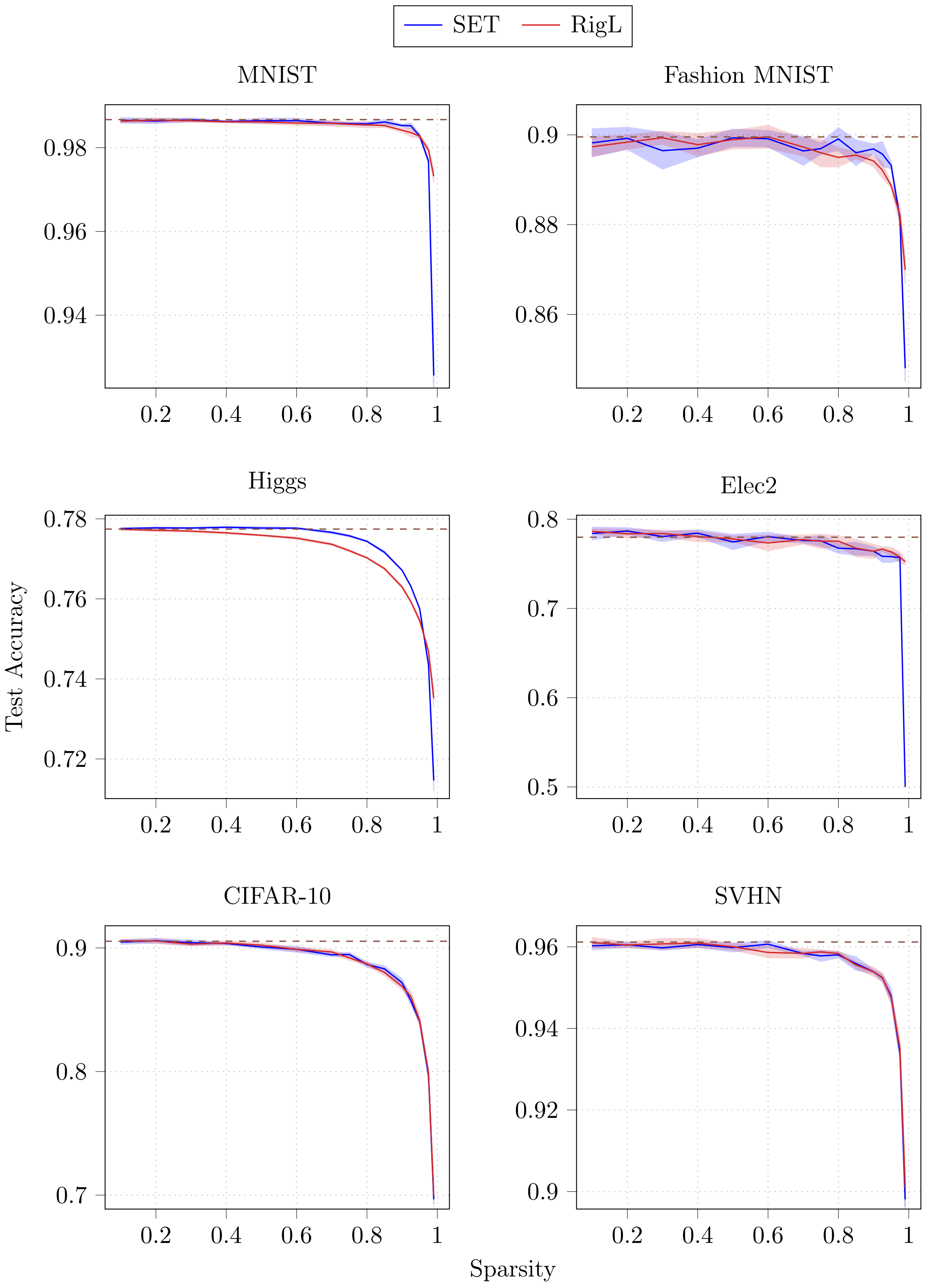


FIGURE 5.1: Results of all reproduction experiments. Average accuracies for 15 different sparsities over 5 runs are plotted for SET and RigL. Sparsities range from 0.1 to 0.99. The average accuracy on the densely trained model for each experiment is added as a dashed line.

TABLE 5.1: MNIST reproduction results with different sparsities and methods.

| Sparsity | Accuracy | Training FLOPs | Inference FLOPs | Accuracy | Training FLOPs | Inference FLOPs |
|---|---|---|---|---|---|---|
| Dense | $98.7 \pm 0.03$ | 1x (1.01e14) | 1x (2.79e6) | | | |
| | | SET | | | RigL | |
| 0.100 | $98.7 \pm 0.06$ | 0.90x | 0.90x | $98.6 \pm 0.04$ | 0.90x | 0.90x |
| 0.200 | $98.6 \pm 0.06$ | 0.80x | 0.80x | $98.7 \pm 0.05$ | 0.80x | 0.80x |
| 0.300 | $98.7 \pm 0.04$ | 0.70x | 0.70x | $98.6 \pm 0.04$ | 0.70x | 0.70x |
| 0.400 | $98.6 \pm 0.02$ | 0.60x | 0.60x | $98.6 \pm 0.02$ | 0.60x | 0.60x |
| 0.500 | $98.6 \pm 0.06$ | 0.50x | 0.50x | $98.6 \pm 0.05$ | 0.50x | 0.50x |
| 0.600 | $98.6 \pm 0.06$ | 0.40x | 0.40x | $98.6 \pm 0.05$ | 0.40x | 0.40x |
| 0.700 | $98.6 \pm 0.04$ | 0.30x | 0.30x | $98.6 \pm 0.07$ | 0.30x | 0.30x |
| 0.750 | $98.6 \pm 0.03$ | 0.25x | 0.25x | $98.6 \pm 0.06$ | 0.25x | 0.25x |
| 0.800 | $98.6 \pm 0.04$ | 0.20x | 0.20x | $98.5 \pm 0.07$ | 0.20x | 0.20x |
| 0.850 | $98.6 \pm 0.06$ | 0.15x | 0.15x | $98.5 \pm 0.04$ | 0.15x | 0.15x |
| 0.900 | $98.5 \pm 0.03$ | 0.10x | 0.10x | $98.4 \pm 0.05$ | 0.10x | 0.10x |
| 0.925 | $98.5 \pm 0.07$ | 0.07x | 0.07x | $98.4 \pm 0.10$ | 0.08x | 0.07x |
| 0.950 | $98.3 \pm 0.05$ | 0.05x | 0.05x | $98.3 \pm 0.05$ | 0.05x | 0.05x |
| 0.975 | $97.7 \pm 0.05$ | 0.02x | 0.03x | $97.9 \pm 0.09$ | 0.03x | 0.03x |
| 0.990 | $92.6 \pm 0.63$ | 0.01x | 0.01x | $97.3 \pm 0.15$ | 0.01x | 0.01x |

are executed in the same environment. However, the efficiency gains should be roughly comparable to previous research to ensure the validity of our implementation. Therefore, we compare our reproduction experiment results to the results of [26] and [82]. First, we look at the results of [26], where a sparsity-accuracy trade-off is plotted for CIFAR-10 when training a WideResNet-22-2 [119] with RigL and two other SNN methods (excluding SET). Overall, the performance of their WideResNet-22-2 is better than that of our ResNet with 20 layers. Furthermore, they evaluate only three sparsities ranging from 0.5 to $\approx$ 0.95, making the results hard to compare. However, for both experiments, the accuracy slowly drops as we increase the sparsity. These accuracy drops become more severe for sparsities above $\approx$ 0.8. Second, we compare our results against those of [82], showing the accuracy and efficiency differences between a dense MLP and an MLP trained with SET with a sparsity of $\approx$ 9.95 trained on MNIST and Higgs. Instead of the average accuracy, they present the best accuracy obtained over an unknown number of evaluations. For both datasets, the accuracy of our dense models is comparable to that of their dense models, but our sparse models with comparable sparsities yield slightly lower accuracies than theirs. This indicates that our implementation of SET is slightly worse than theirs. Still, since the results lie in a comparable range and otherwise behave as expected, we continue with our reproduction experiment results as a baseline for the remainder of this study.

As a final inspection of these results, we calculate the hypervolumes of the reproduction trade-offs and present them in Table 5.2. As introduced in Section 4.3, the hypervolume is a measure of the trade-off shape. If structurally applied sparsity consistently resulted in a comparable accuracy loss, we would see comparable hypervolumes across all datasets in this experiment. However, as could be expected, the observed hypervolumes are highly variable. From this, we conclude that SNN training introduces a different trade-off for each dataset, indicating the need to study SNN training in various environments.

TABLE 5.2: The highest found accuracy and trade-off hypervolumes for each dataset in the reproduction experiment.

| Dataset | Highest Accuracy | Hypervolume |
|---|---|---|
| MNIST | $98.7 \pm 0.03$ | 1.185 |
| Fashion MNIST | $90.0 \pm 0.25$ | 0.980 |
| Higgs | $77.8 \pm 0.02$ | 0.854 |
| Elec2 | $78.7 \pm 0.38$ | 1.203 |
| CIFAR-10 | $90.6 \pm 0.18$ | 0.670 |
| SVHN | $96.1 \pm 0.13$ | 0.738 |

TABLE 5.3: Hyperparameters with different constant values or ranges for each dataset.

| Dataset | Architecture | Batch Size | Max Epochs | Learning Rate Scheduler | Use Dropout | Use Label Smoothing |
|---|---|---|---|---|---|---|
| MNIST | MLP | 128 | 200 | Constant | True/False | False |
| Fashion MNIST | MLP | 128 | 200 | Constant | True/False | False |
| Elec2 | MLP | 128 | 200 | Constant | True/False | False |
| Higgs | MLP | 4096 | 200 | Constant | True/False | False |
| CIFAR-10 | ResNet | 256 | 150 | Cosine | False | True/False |
| SVHN | ResNet | 256 | 150 | Cosine | False | True/False |

## 5.2 Optimization Experiment

Having a baseline established, we can start approximating a new accuracy-efficiency trade-off for each dataset. The possible ranges for all hyperparameters are set in Table 4.2, but we limit some hyperparameters for all experiments. The network architecture type, batch size and learning rate scheduler are set to constant values to match the reproduction experiments. Furthermore, dropout and label smoothing are only used if enabled in the reproduction experiment. Finally, the maximum number of epochs is limited for larger datasets to speed up the optimization processes, resulting in the hyperparameter ranges in Table 5.3.

We use MO-SMAC with fixed budgets to approximate the Pareto front. To define our budgets, we follow a comparable *rule of thumb* as Horn et al. [44] when performing model-based MO-HPO, using a budget of $50d$, with $d$ defined as the total number of configurable real-valued hyperparameters and integer-valued hyperparameters with large ranges. Our experiments with MLP architectures have 11 such hyperparameters and experiments with ResNet architectures have 8, resulting in budgets of 550 and 400, respectively. We wish to include categorical and small-range integer-valued hyperparameters in this calculation due to the large number of such hyperparameters in our configuration space. Therefore, we extend the rule of thumb to $50d + 25c$, with $c$ being the number of categorical and small-range integer-valued hyperparameters. This results in budgets of 650 and 550 for the MLP and ResNet experiments, respectively. We confirmed these budgets to be reasonable with a small experiment where we optimized with a budget of 2500. Here, we saw the total hypervolume converge after 500 evaluations.

The results of different seeds are combined after optimization, and configuration objectives are averaged for each instance they were evaluated on. Not all configurations have been evaluated on the same instances, making comparing configurations based on these

Table 5.4: Spearman's correlation coefficient of each objective combination within the optimization results.

| Dataset | Accuracy Inference | Accuracy Training | Inference Training |
|---|---|---|---|
| MNIST | 0.98 | 0.98 | 0.95 |
| Fashion MNIST | 0.97 | 0.99 | 0.95 |
| Higgs | 0.95 | 0.95 | 0.94 |
| Elec2 | 0.95 | 0.98 | 0.91 |
| CIFAR-10 | 0.95 | 0.98 | 0.91 |
| SVHN | 0.96 | 0.95 | 0.88 |

results difficult. Therefore, we need to re-evaluate all non-dominated configurations on an equal set. We train them on the complete train set and evaluate them on the unseen test set. We also do this for the second non-dominated layer, as this might contain better configurations after re-evaluation due to the stochasticity of network training or overfitting on the validation set. Furthermore, this could strengthen an analysis of the trade-off later on by creating a larger dataset. Together, this results in our approximation of the accuracy-efficiency trade-off.

While we use MO-SMAC to approximate the entire trade-off, we are only interested in configurations resulting in a subset of the objective space. Namely, those that yield high accuracies. Ultimately, high efficiency is worthless if our model is useless. By design, the reproduction experiments already focus on this area of the trade-off. Therefore, we filter our optimization experiment on the configurations with at least the accuracy of the worst configuration in the reproduction experiment of that dataset. An exception to this rule is the experiment on the Elec2 dataset, where the worst reproduction configuration produces near-random results. Here, we take the second-worst-performing configuration as a cut-off point.

### 5.2.1 Results

These experiments result in six accuracy-efficiency trade-offs, one for each dataset. These trade-offs are three-dimensional and are challenging to inspect visually. Therefore, we visualize them by showing the two-dimensional planes of two objective pairs—one to show inference efficiency against accuracy and one to show training efficiency against accuracy. We do not show the inference efficiency against the training efficiency, as these add little extra information. We scale both efficiency objectives logarithmically in these plots to highlight the shape of the region of interest and the severity of the trade-offs. As an example, Appendix B shows how the trade-offs resulting from MO-SMAC map to the eventually shown plots for the MNIST dataset. All results can be found in Figure 5.2.

Similarly to the reproduction experiment results, these results show that the best-performing models, accuracy-wise, are often among the least efficient models. To better show this, we calculate Spearman's correlation coefficients between all objective combinations and present them in Table 5.4. The correlation coefficient scales between -1 and 1, with coefficients of -1 or 1 denoting a perfect correlation. As expected, there is a high correlation between the two efficiency objectives. More interesting is that they show a comparable or even higher correlation between the accuracy and either efficiency objective. Since we know that the efficiency objectives scale logarithmically, a linear decrease in accuracy results in an exponential increase in efficiency.

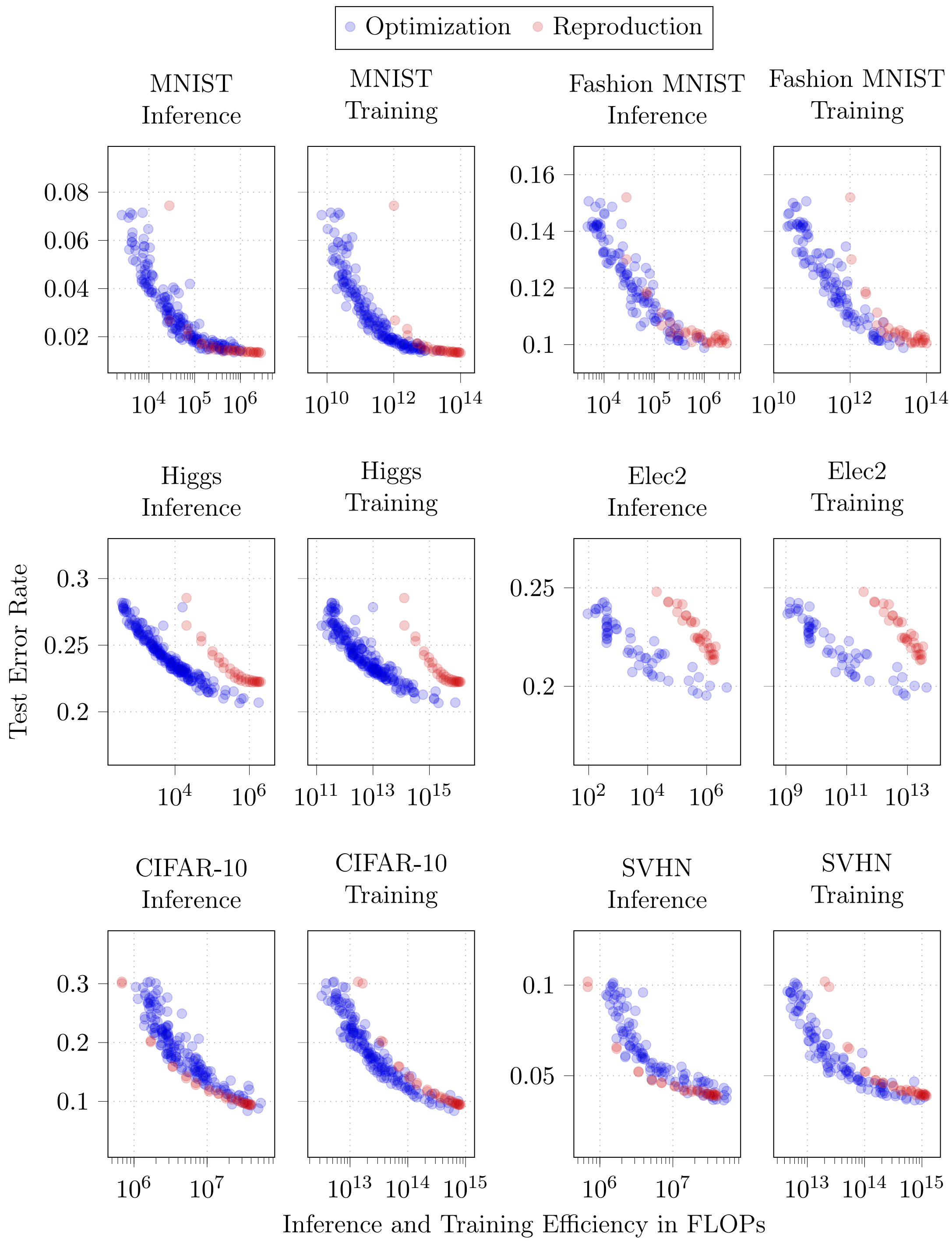


Figure 5.2: Results of all optimization experiments and their reproduction experiment results. All configurations are trained on the complete training dataset and evaluated on the test set. Optimization results are filtered on their error rate. Efficiency axes are logarithmically scaled.

### Comparison to Reproduction

To answer how the found trade-offs relate to the trade-offs found in previous research (**RQ1.1**), we also compared the optimization results to the reproduction results. As can already be seen in the plots of Figure 5.2, the reproduction experiments are focused on a small area of the objective space. While the reproduction experiments already show a large possible increase in efficiency, the optimization experiment shows that efficiency can often be improved even further with a smaller decrease in accuracy. Interestingly, the optimization results do not improve inference efficiency for the ResNet architectures (CIFAR-10 and SVHN). Apparently, finding well-performing architectures that are efficient at inference is more of a challenge in these models. Still, the training efficiency can be vastly improved.

To strengthen this visual analysis, we performed a hypervolume analysis. This analysis normalises the reproduction and optimization experiment to the same space. Relative hypervolumes of all reproduction and optimization experiments can be found in Table 5.5. Next to the hypervolume of each experiment, we show the highest accuracy found in each experiment and how both statistics change between the two experiments.

TABLE 5.5: Hypervolume comparison of optimization results with reproduction results. Objectives are normalized to the same space.

| Experiment | Reproduction | | Optimization | | Improvement | |
|---|---|---|---|---|---|---|
| Dataset | Highest Accuracy | Hyper-volume | Highest Accuracy | Hyper-volume | Highest Accuracy | Hyper-volume |
| MNIST | $98.7 \pm 0.03$ | 0.434 | $98.6 \pm 0.06$ | 0.955 | -0.1%pt | 2.20x |
| Fashion MNIST | $90.0 \pm 0.25$ | 0.427 | $90.1 \pm 0.19$ | 0.893 | 0.1%pt | 2.09x |
| Higgs | $77.8 \pm 0.02$ | 0.196 | $79.3 \pm 0.08$ | 0.865 | 1.5%pt | 4.41x |
| Elec2 | $78.7 \pm 0.38$ | 0.088 | $80.5 \pm 0.59$ | 0.800 | 1.8%pt | 9.09x |
| CIFAR-10 | $90.6 \pm 0.18$ | 0.508 | $91.6 \pm 0.29$ | 0.731 | 1.0%pt | 1.44x |
| SVHN | $96.1 \pm 0.13$ | 0.558 | $96.4 \pm 0.07$ | 0.809 | 0.3%pt | 1.45x |

TABLE 5.6: Statistics of the optimization trade-offs. We calculate the number of configurations within each trade-off, the highest accuracy, the average sparsity *if* RigL or SET was used, the distribution of SNN training algorithms, and the total hypervolume of the trade-off.

| Dataset | Config-urations | Highest Accuracy | Average Sparsity | Hyper-volume | Algorithm Percentages RigL | SET | Dense |
|---|---|---|---|---|---|---|---|
| MNIST | 176 | $98.6 \pm 0.06$ | 0.52 | 0.887 | 38.07 | 51.14 | 10.8 |
| Fashion MNIST | 111 | $90.1 \pm 0.19$ | 0.61 | 0.818 | 44.14 | 51.35 | 4.50 |
| Higgs | 206 | $79.3 \pm 0.08$ | 0.51 | 0.835 | 33.50 | 58.74 | 7.77 |
| Elec2 | 57 | $80.5 \pm 0.59$ | 0.41 | 0.797 | 33.33 | 47.37 | 19.30 |
| CIFAR-10 | 165 | $91.6 \pm 0.29$ | 0.79 | 0.799 | 52.12 | 43.03 | 4.85 |
| SVHN | 105 | $96.4 \pm 0.07$ | 0.70 | 0.912 | 50.48 | 40.00 | 9.52 |

From this table, we can see a few trends. First, we see that the hypervolume of all optimization experiments is substantially larger than that of the reproduction experiments. This means that our approximation of the Pareto front is closer to the actual Pareto front. This is as expected, as previous studies never meant to approximate the entire Pareto front and only show the possibilities of SNN training. Second, we see comparable hypervolume improvements in each dataset pair's two experiments. This indicates that even though the fixed hyperparameter configurations in the reproduction experiment were not designed for all datasets, they created trade-offs of comparable quality. Third, we see that the highest accuracy in the optimization experiments is often similar to that in the reproduction experiments. This means that the fixed configurations in the reproduction experiments were well-optimized configurations that could not be improved much, but we did find configurations that performed comparably. If we combine these observations with a visual inspection of Figure 5.2, we can say that our optimization trade-offs not only better cover the true Pareto front but actually dominate the reproduction trade-off. With this, we conclude that MO-HPO finds a better and more realistic accuracy-efficiency trade-off. Furthermore, the best-performing models are often comparable to the best multi-layer perceptron or ResNet in literature [113, 108, 70, 32, 84].

### Trade-Off Statistics

Before further diving into the influence of all hyperparameters in Chapter 6, we already start by looking at the effect of SNN training in the optimized trade-offs. We show several statistics of the optimisation results in Table 5.6. First, we look at the distribution of SNN training algorithms in the resulting trade-off. Two notable conclusions can be drawn from this. First, all trade-offs include only a few dense networks. This indicates that the best way to improve efficiency in our environment is by using SNN training algorithms instead of smaller, dense networks. Second, there seems to be no clear preference for SET or RigL if we use SNN training. Both algorithms are often equally distributed in all trade-offs, making them both viable options if we only consider our three objectives.

Given that most configurations use SNN training, we are interested in which values of sparsities were found. First, we look at the average sparsity in Table 5.6, where we can see that the average sparsity of datasets optimized with MLPs is considerably lower than those optimized with ResNets (CIFAR-10 and SVHN). To further inspect this, we colour map the accuracy-training efficiency trade-offs by the configuration's sparsities in Figure 5.3. This shows how most configurations in the trade-offs of MLP datasets use a sparsity between 0.3 and 0.8. In contrast, previous literature focused on sparsities of 0.8 or higher to highlight the potential of SNN training. While a higher sparsity increases the efficiency even further, we must not neglect the effect of sparsity on lower values.

To conclude, we look at the hypervolumes of all trade-offs. Just like in the reproduction experiments, these values can be used to compare the shapes of the trade-offs as they have been normalized to similar spaces. All hypervolumes lie close to each other, showing that the general shape of the accuracy-efficiency trade-off is similar in all datasets.

## 5.3 Evaluation Experiment

A final step in finding the trade-off is the evaluation experiment. Here, we evaluate configurations found by MO-SMAC with different SNN algorithms and sparsities as we did with the reproduction experiments. In doing so, we can discover whether the trade-off incurred by SNN training is similar in arbitrary well-performing networks. Furthermore, this step serves as a validation step of the quality of the approximated Pareto front. Since this is an expensive operation, we do this only on the 'most interesting' configurations in the Pareto set: those with the 20 highest accuracies. We defined 31 configurations with varying sparsities and SNN training algorithms in the reproduction experiment. To limit the number of configurations, we decrease that to 17 configurations for this experiment: one dense configuration and eight different sparsities for SET and RigL. Like in previous experiments, these configurations are trained and evaluated on the full train and test sets.

We compare the evaluation experiment against the optimization and the reproduction experiment. In comparing against the optimization experiment, we compare the objectives of the resulting 340 configurations with the optimization results. This shows the effect of a structured application of SNN training on optimized configurations. To compare the evaluation experiment to the reproduction experiment, we individually examine the 20 trade-offs resulting from the original 20 configurations. We are interested in two things. First, did our optimization experiment find hyperparameter configurations in which SNN training excels? Second, does this predefined set of SNN configurations result in similar trade-offs for arbitrary configurations? To answer this, we look at the highest accuracy in the trade-off, the sparsity of that configuration and the hypervolume of that trade-off after normalization.

### 5.3.1 Results

#### Configurations

First, we look at the initial sparsities of the configurations that we further investigate. This data can be found in Table 5.7. We can see that most high-accuracy configurations selected for the evaluation experiment are sparsely trained models.

#### Compared to Optimization

Figure 5.4 shows the trade-offs from the evaluation experiments against the optimization experiments, and Table 5.8 shows a hypervolume comparison between the two results. For the MLP datasets, the general shape of the trade-off remains similar to that of the optimization experiment. However, we see an improvement in the minimal error rate for Fashion MNIST, Higgs, and Elec2. These improvements are at the low-efficiency (high FLOPs) regions, which is where we sourced our configurations from, showing that our optimization trade-offs can be further improved by structurally applying sparsity. In the

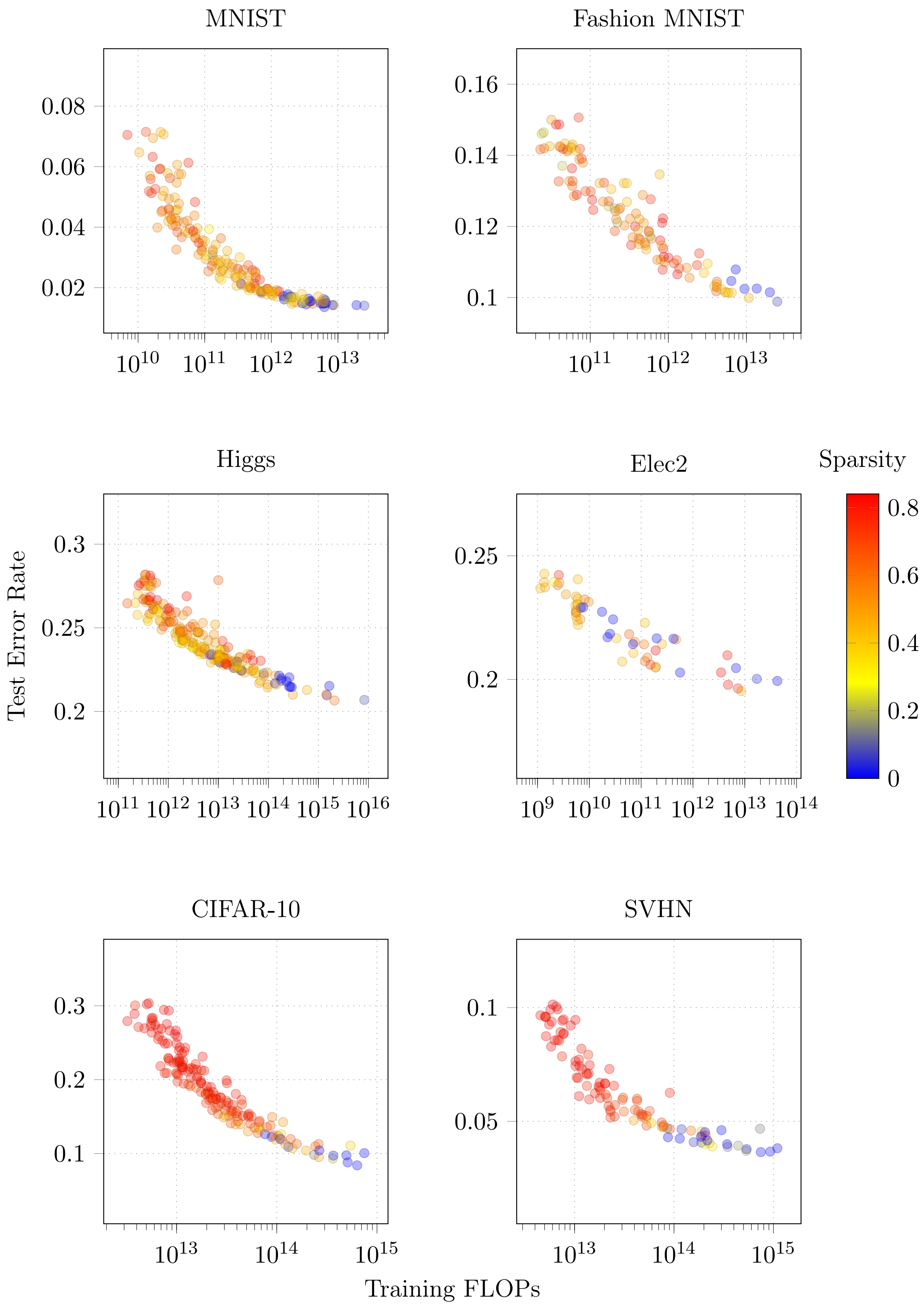


FIGURE 5.3: Test error rate vs training efficiency in the optimization results with sparsity colour mapped.

Table 5.7: Number of dense configurations and information on the sparsities within the 20 configurations selected for further analysis in the evaluation experiment.

| Dataset | Dense | Sparsity | | | |
|---|---|---|---|---|---|
| | Configs | Min | Max | Mean | Std Dev |
| MNIST | 5 | 0.37 | 0.95 | 0.54 | 0.16 |
| Fashion MNIST | 1 | 0.07 | 0.95 | 0.72 | 0.22 |
| Higgs | 2 | 0.09 | 0.68 | 0.51 | 0.13 |
| Elec2 | 3 | 0.01 | 0.79 | 0.46 | 0.15 |
| CIFAR10 | 4 | 0.15 | 0.83 | 0.61 | 0.19 |
| SVHN | 5 | 0.07 | 0.89 | 0.59 | 0.30 |

Table 5.8: Hypervolume comparison of evaluation results with optimization results. Objectives are normalized to the same space.

| Experiment | Optimization | | Evaluation | | Improvement | |
|---|---|---|---|---|---|---|
| Dataset | Highest Accuracy | Hyper-volume | Highest Accuracy | Hyper-volume | Highest Accuracy | Hyper-volume |
| MNIST | $98.6 \pm 0.06$ | 0.894 | $98.7 \pm 0.05$ | 0.835 | 0.1%pt | 0.93x |
| Fashion MNIST | $90.1 \pm 0.19$ | 0.818 | $90.5 \pm 0.00$ | 0.836 | 0.4%pt | 1.02x |
| Higgs | $79.3 \pm 0.08$ | 0.824 | $79.5 \pm 0.06$ | 0.680 | 0.2%pt | 0.83x |
| Elec2 | $80.5 \pm 0.59$ | 0.631 | $81.5 \pm 0.00$ | 0.761 | 1.0%pt | 1.21x |
| CIFAR-10 | $91.6 \pm 0.29$ | 0.758 | $91.7 \pm 0.22$ | 0.832 | 0.1%pt | 1.10x |
| SVHN | $96.4 \pm 0.07$ | 0.881 | $96.4 \pm 0.11$ | 0.996 | 0.0%pt | 1.13x |

ResNet experiments, we do not see this effect, most probably because the configurations with the best error rate were already dense. However, we do see that the inference efficiencies of the evaluation experiment surpass those of the optimization experiment. This shows that the configurations optimized for their error rate yield better results when trained with sparsity than those optimized for their efficiency.

In Section 3.2, we argued that automated HPO finds better configurations than manual optimization and that automated MO-HPO is the best way to approximate the Pareto front when dealing with conflicting objectives. Therefore, the fact that our optimized results can be improved with manual adaptations is surprising. This gives us enough reason to believe that the optimization experiment could have approximated a better trade-off if better initialized. We see several potential problems: first, the hyperparameter configuration space might be too complex or improperly defined to find the desired configurations. We are optimizing 14 or 15 hyperparameters simultaneously, which all have complex effects on each other. While we believe that we have chosen ranges and scales for these hyperparameters such that well-performing configurations could be easily found, it might be well possible that improvements can be made here. Second, the given budget might be too low. The hyperparameter configuration space is complex to grasp, and a higher budget can improve the surrogate models understanding of it. While we have evaluated the effect of a larger budget on a small scale and noticed little potential improvements, the determined budgets and $50d + 25c$ rule could still be too little. A side note to increasing the budget is its potential negative effects. Even though we used k-fold cross-validation to prevent overfitting on the train set, an optimizer might still overfit on the different folds if the budget is too large. Third, MO-SMAC might be initialized differently. Even

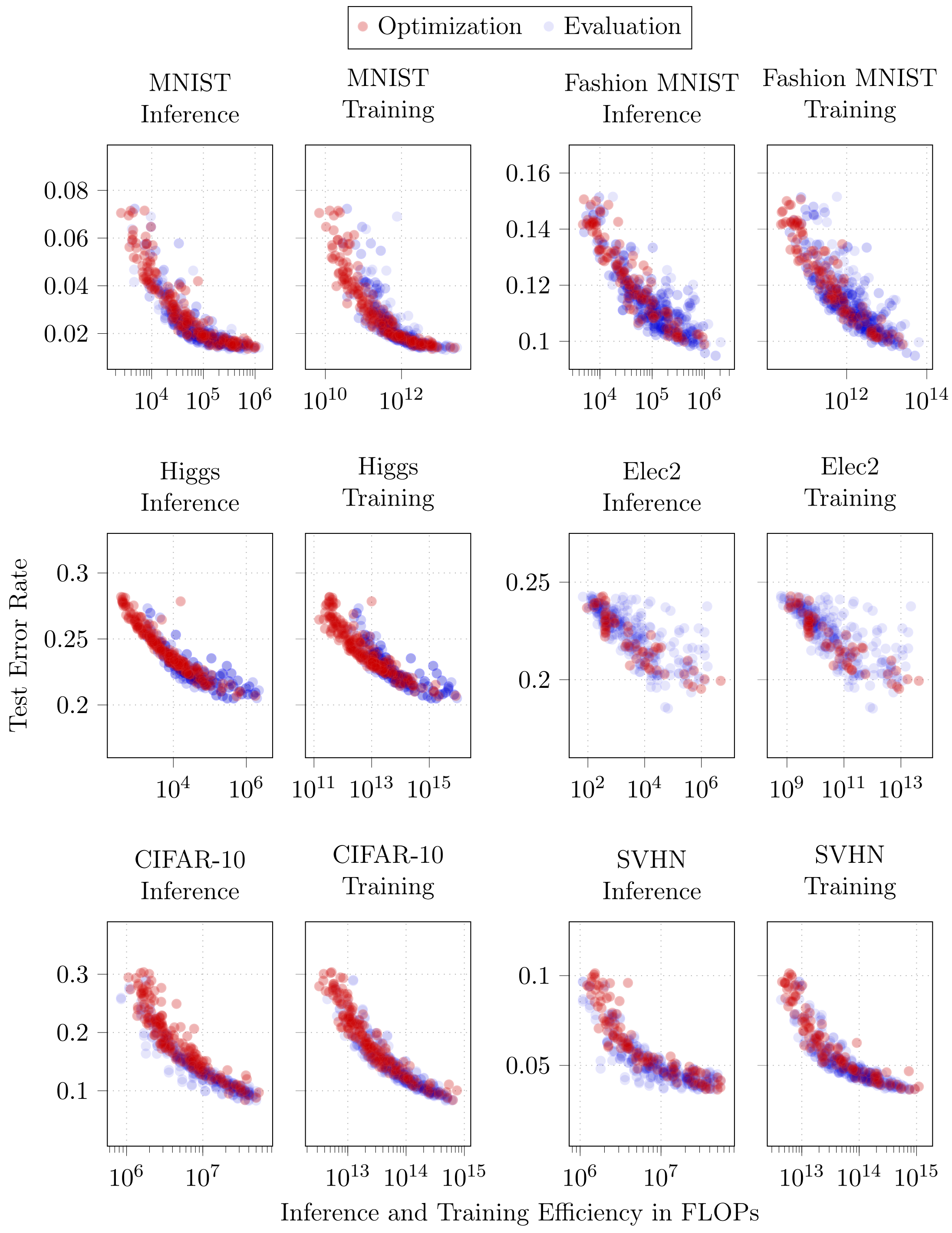


FIGURE 5.4: Results of the evaluation experiment compared to the optimization experiment. All configurations are trained on the complete training dataset and evaluated on the test set. Results are filtered on their error rate. Efficiency axes are logarithmically scaled.

though optimizers have the goal of reducing the manual optimization of hyperparameters, they add new hyperparameters themselves, too. Non-default settings such as evaluating configurations on different seeds, logarithmically scaled objectives, or modified objective surrogate models could further improve its ability to approximate the Pareto front.

### Compared to Reproduction

To compare the evaluation experiment against the reproduction experiment, we look at the 20 individual trade-offs resulting from the 20 configurations on which the evaluation experiment is based. Figure 5.5 plots the highest accuracy per trade-off against the hypervolume of each trade-off. First, we examine whether the configurations for one dataset result in comparable hypervolumes. This would mean that we can expect a specific accuracy-efficiency trade-off when applying sparsity to an optimized configuration and that the reproduction experiment is a valid method to test the effect of an SNN training method. Overall, we cannot say that this is the case. We see relatively comparable hypervolumes in the CIFAR-10 dataset, but this pattern does not return in the SVHN dataset. Therefore, we can conclude that the effect of SNN training cannot easily be predicted by or captured within the simple constraints of the reproduction experiment.

Furthermore, we look at the sparsities of the configuration with the highest accuracy of each trade-off. We primarily see dense or low-sparsity models reaching the highest accuracy in these trade-offs, especially in our datasets trained with ResNets. Even though most of these configurations were selected for SNN training in the optimization experiment, dense training often results in the best accuracy. There also seems to be little coherence in the trade-offs in which highly sparse configurations excel, further showing its unpredictability. For example, the two configurations with the sparsities resulting from the Higgs dataset are on opposing ends of the accuracy axis and come from trade-offs with substantially different hypervolumes.

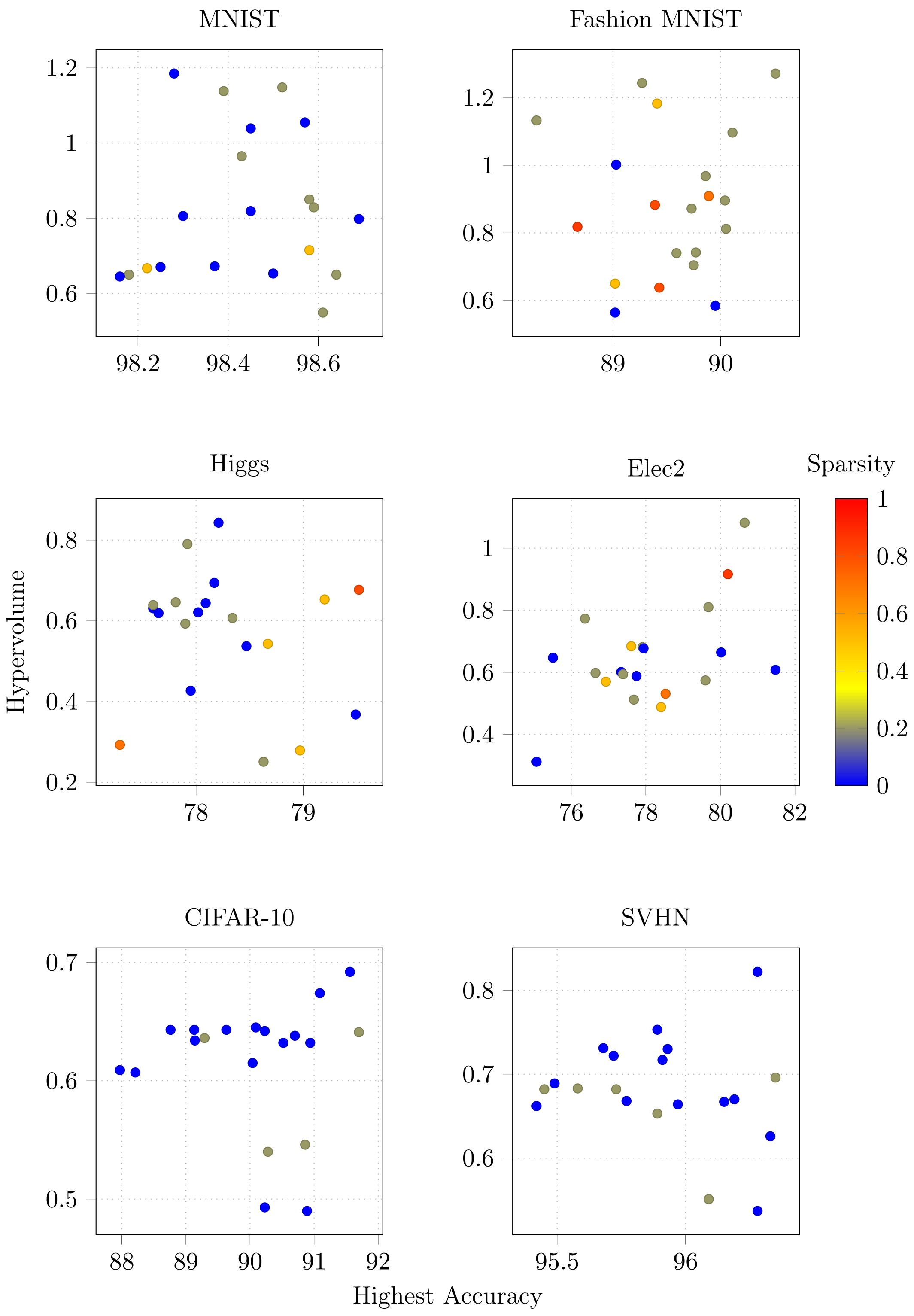


FIGURE 5.5: Evaluation experiment results. For each dataset, we selected the best 20 configurations and evaluated them with 17 different sparsity configurations. We plot the accuracy of the best of these 17 configurations against the hypervolume of those 17 configurations. The sparsity of the best configurations is colour-mapped.

# Chapter 6

# Hyperparameter Analysis

The previous chapter created accuracy-efficiency trade-offs with different experiments for every dataset. Until now, we have focused on studying the objective space resulting from these trade-offs and briefly analyzed the sparsities used to reach this. However, valuable insights can be learned by studying the hyperparameter configurations behind these trade-offs. In this chapter, we perform additional analyses on the optimization results to learn more about the nature of the accuracy-efficiency trade-off. First, we perform a hyperparameter importance analysis to determine influential hyperparameters. Then, we further dive into the optimal value ranges for all hyperparameters using hyperparameter influence methods.

## 6.1 Hyperparameter Importance

Before further looking into how specific optimized hyperparameters influence the trade-off, we wish to know which hyperparameters are most relevant to optimize. Some hyperparameters might not have much of an effect on any of our objectives, and some only on specific parts of our trade-off. We can use a hyperparameter importance (HPI) analysis to discover this. Such an analysis gives each hyperparameter in a hyperparameter configuration space a relative importance score. This importance score in itself carries little value but can be compared with the importance scores of other hyperparameters in the same configuration space. Instinctively, a hyperparameter with greater importance has a greater influence on the specified objective. In multi-objective optimization, we do not calculate this importance score for a single objective, but we calculate multiple importance scores for different parts of the trade-off.

We used MO-fANOVA [109] to quantify the importance of each hyperparameter in each trade-off section. MO-fANOVA (further introduced in Subsection 3.3.2) needs a set of hyperparameter configurations with corresponding objective scores, where a more extensive set results in better analysis. We have created such a dataset in our optimization experiments. However, this data can contain a bias towards some areas of the trade-off induced by the exploitation of MO-SMAC [83], which could cause MO-fANOVA to give biased results. Therefore, we evaluated 2000 configurations (similar to the number of unique configurations found in each optimization experiment) generated by a random search with Latin hypercube sampling for each dataset. Unlike the results of the optimization experiments, we did not filter on a specific subset of the objective space. We took the logarithm of the two efficiency objectives to capture the trade-off better, after which we performed data preparation as described in [109].

In the work of Theodorakopoulos et al. [109], MO-fANOVA is limited to two objectives.

This allows them to present the results in an interpretable two-dimensional plot directly. While the MO-fANOVA approach can be directly extended to three objectives by showing the results in a three-dimensional plot, we prefer two-dimensional plots since they are more readable. Therefore, we slightly modified their approach to plot three objectives in a single two-dimensional graph, which we dub the importance scale.

As a first step, we perform MO-fANOVA three times: once on the error rate and inference efficiency, once on the inference and training efficiency, and once on the training efficiency and error rate. This results in three plots, showing the relative importance of our hyperparameters when we scale between two objectives. For example, the first graph scales from the error rate to the inference efficiency. Here, the leftmost points show the relative importance if we set the error rate as our sole objective. Then, we see how the relative importance changes as we focus more and more on inference efficiency. Halfway through, this graph shows which hyperparameters are most essential to optimize when we are equally interested in error rate and inference efficiency. From this point on, we scale to the end of the graph, where the inference efficiency is our sole objective. The second graph scales from the inference to the training efficiency, with the first points showing the relative hyperparameter importance if inference efficiency is our sole objective. As these importances are equal to those at the end of the first graph, we can concatenate the first two graphs. Similarly, we can concatenate the last two graphs. In total, this results in a single graph, the importance scale, where we plot the relative hyperparameter importance as we scale from a sole focus on the error rate to the inference efficiency, to the training efficiency, and back to the error rate. The error rate can be found on both extremes of the scale such that all areas are included. Note that the plotted points in these scales are not evenly distributed throughout the x-axes and might be concentrated in small ranges. This happens when the objective space is not evenly covered, which is often the case in multi-objective optimization.

In addition to the plotted MO-fANOVA results, we present the single-objective fANOVA results for all three objectives. These show how the results would differ if we focused on only one objective and serve as a way to evaluate hyperparameters independently of the other objectives.

### 6.1.1 Results

The resulting MO-fANOVA importance scales are presented in Figure 6.1. The single-objective HPI values for the error rate in Table 6.1 and for the efficiency objectives in Appendix D. Plots showing the random search results from which we calculated these importances can be found in Appendix C. Hyperparameters with a relative importance near zero are not shown in the importance scales and HPI tables. These are all constant hyperparameters, dropout, label smoothing and the sparsity update frequency. What remains are 15 hyperparameters with varying importance. In the remainder of this section, we look at several interesting hyperparameters individually.

#### Algorithm

Surprisingly, the choice in the SNN training algorithm seems to be an unimportant hyperparameter throughout most of the importance scale. When zooming in, we can see a few non-zero values, such as at the end of the error-inference scale in MNIST and Fashion MNIST. There is no logical explanation as to why the algorithm would be more important at these spots and we account for these values as noise in the MO-fANOVA results.

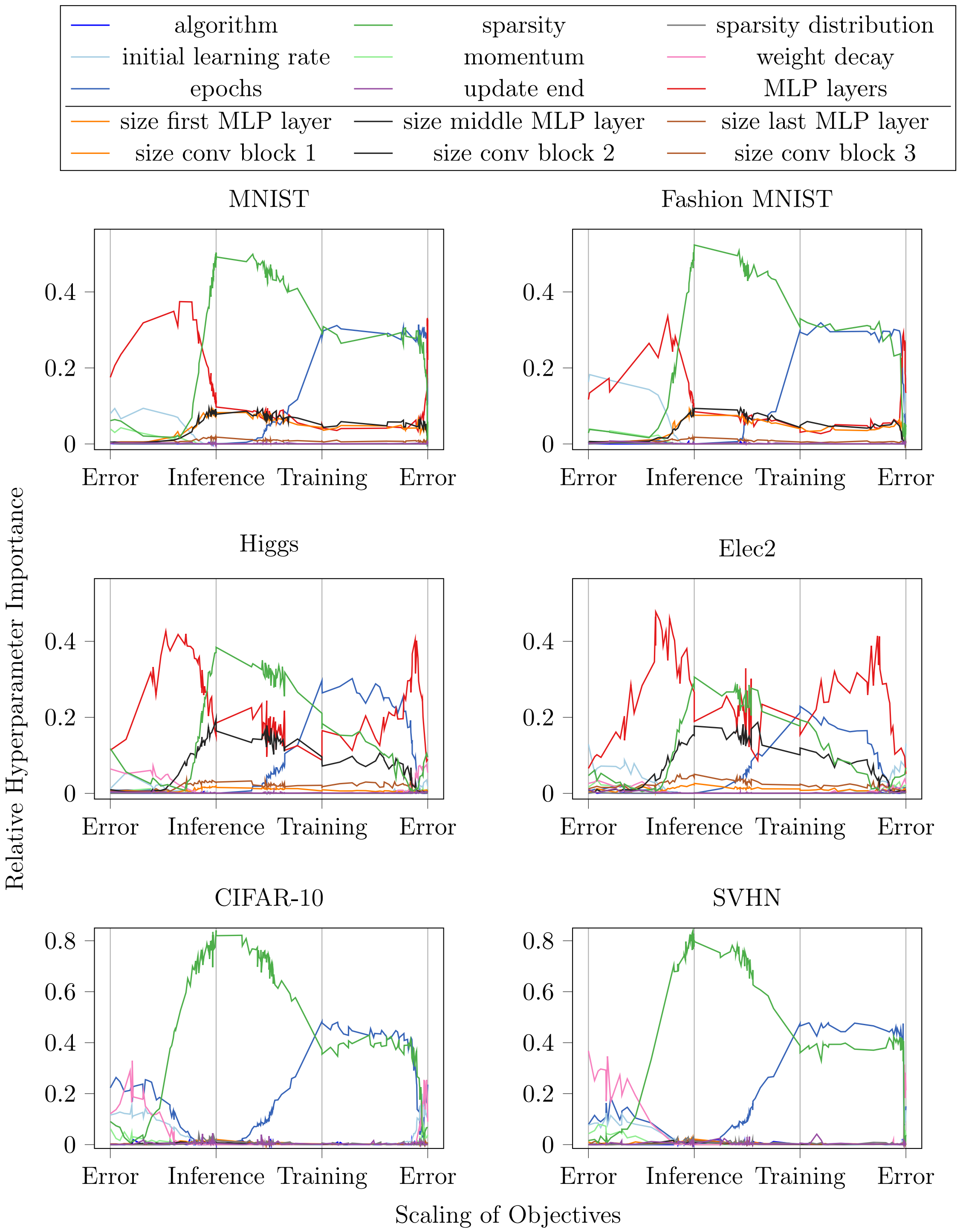


FIGURE 6.1: MO-fANOVA results. Each plot (i.e. objective scale) shows the relative importance of each hyperparameter in three different objective sections. In the first section of each plot, the importance of each hyperparameter on the error rate is slowly scaled towards the importance of the inference efficiency, showing how different hyperparameters are more important on different parts of that trade-off. The following two sections scale from inference to training efficiency and from training efficiency back to the error rate.

Table 6.1: Relative hyperparameter importance for the error rate for all configurable hyperparameters. The three greatest hyperparameter importances for each dataset are marked in **bold**.

| Hyperparameter | MNIST | Fashion MNIST | Higgs | Elec2 | CIFAR10 | SVHN |
|---|---|---|---|---|---|---|
| algorithm | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| sparsity | **0.066** | **0.049** | **0.097** | 0.029 | 0.076 | 0.020 |
| sparsity distribution | 0.002 | 0.000 | 0.000 | 0.001 | 0.013 | 0.000 |
| update end | 0.001 | 0.005 | 0.003 | 0.002 | 0.001 | 0.002 |
| initial learning rate | **0.083** | **0.136** | 0.032 | **0.128** | **0.114** | **0.111** |
| momentum | 0.034 | 0.022 | 0.011 | 0.016 | 0.023 | 0.053 |
| weight decay | 0.001 | 0.002 | **0.087** | **0.057** | **0.168** | **0.280** |
| epochs | 0.002 | 0.004 | 0.008 | 0.008 | **0.241** | **0.139** |
| MLP layers | **0.243** | **0.129** | **0.063** | **0.073** | - | - |
| size first MLP layer | 0.004 | 0.006 | 0.004 | 0.008 | - | - |
| size middle MLP layer | 0.005 | 0.005 | 0.003 | 0.006 | - | - |
| size last MLP layer | 0.002 | 0.006 | 0.007 | 0.009 | - | - |
| size conv block 1 | - | - | - | - | 0.000 | 0.000 |
| size conv block 2 | - | - | - | - | 0.000 | 0.000 |
| size conv block 3 | - | - | - | - | 0.001 | 0.000 |

### Sparsity

The fact that the SNN training algorithm is of little importance does not mean that sparsity is of little importance. On the contrary, sparsity plays a pivotal role throughout the entirety of all importance scales. Sparsity is the most important hyperparameter in determining inference and training efficiency. Furthermore, sparsity remains an important hyperparameter when we focus more on the error rate and is nearly as important as the learning rate when solely focusing on the error rate.

### Sparsity Distribution

We allow for two different sparsity distributions: uniform or ERK (Erdös Rényi Kernel [26]). Previous studies show that ERK distributions sometimes increase the accuracy at the cost of a worse efficiency [26, 67]. In our results, the sparsity distribution is of negligible importance to the error rate on all datasets and of little importance to the efficiency objectives when using ResNets.

### MLP Size

We examine the number of MLP layers and the three hyperparameters to determine layer sizes together. In all experiments, the number of MLP layers is the most important hyperparameter when optimizing for error rate and inference efficiency. This shows that the network's depth significantly impacts its accuracy and efficiency.

Furthermore, we see that the sizes of the layers show similar patterns over all MLP experiments. All are of little importance to the error rate, showing that most network sizes found by random search can result in well-performing models. Even though all three hyperparameters play a pivotal role in determining efficiency, not all are equally important. The size of the first MLP layer is always more important than the other two hyperparameters, which can be explained by the conditional nature of these variables, which are not considered in the fANOVA tests.

#### ResNet Size

The hyperparameters determining ResNet size (size conv block 1-3) play a small role for all objective scales. For the error rate, this means that the chosen classification problems were ‘simple’ enough for the smallest possible networks in our configuration space. For the efficiency objectives, this either means that our hyperparameter ranges did not allow for too much diversity in network size or that sparsity-related hyperparameters are just that much more important for efficiency. Either way, the best way to guide efficiency within these ResNets is by optimizing sparsity, not model size.

#### Epochs

All scales show similar results for the number of epochs. Although it has some effect on the error rate, it is most relevant in determining the training efficiency. This makes sense, as the training efficiency formula involves multiplying the inference efficiency by the number of training steps, which is directly related to the number of epochs. Its influence on the training efficiency is so large that it is the hyperparameter one should tweak if one wants to minimize training efficiency. All other hyperparameters are comparably important for inference and training efficiency.

#### Learning Rate

Traditionally, the learning rate is considered to be the most important hyperparameter when optimizing for neural network performance. We show that the network size is even more important for MLPs, and the weight decay and number of epochs are more important for ResNets. Other than that, we confirm this common assumption. Additionally, when we scale towards an increasing focus on efficiency, the importance of the learning rate remains stable for a long time and only becomes unimportant when we focus primarily on efficiency.

#### Weight Decay

The weight decay hyperparameter is used to update weight values in SGD and should affect the error rate. Its importance differs between the different datasets with MLPs but is often near zero, even when only focusing on the error rate. Interestingly, it plays a much more vital role in the datasets with ResNets, where it is one of the most important hyperparameters to optimize.

## 6.2 Hyperparameter Influence

The previous section investigated which hyperparameters are most influential in our configuration space. We concluded that sparsity is one of the most important hyperparameters for all objectives. Furthermore, most other sparsity-related hyperparameters were relatively unimportant to the trade-off. In this section, we will extend these insights by analyzing the hyperparameter values found in the optimization experiment results. We perform this analysis in two ways. First, we estimate the probability distributions of all optimized hyperparameters, i.e. the hyperparameters as found in the trade-offs resulting from the optimization experiment. This might reveal ideal default hyperparameter values or relevant configuration ranges. Second, we research the effect of sparsity-related hyperparameters on the trade-off using the trade-off hypervolume. This can indicate how different hyperparameter values are distributed over the trade-off and how we can optimize for either objective.

### 6.2.1 Hyperparameter Distributions

We use kernel density estimations with Scott's bandwidth estimation for real and large integer-valued hyperparameters to estimate the probability distributions of our optimized hyperparameters. These results are plotted in Figure 6.2. It is important to note that some trends (e.g. the dip in the peak of the CIFAR10 epochs plot, potentially) of these distributions might be artefacts of the KDE algorithm or MO-SMAC's optimization process. Therefore, we should be careful when interpreting these results.

Distributions of categorical and small integer-valued hyperparameters are estimated by calculating the percentage of configurations having each possible value. Resulting distributions for these optimized hyperparameters are plotted in Figure 6.3.

The number of samples for each dataset is stated in the first column of Table 5.6. We only consider the values of conditional hyperparameters if they are active. Therefore, the conditional hyperparameters (i.e. dropout, label smoothing, size middle/last MLP layer, sparsity distribution) have fewer samples.

#### General

Most hyperparameters cover most of their configuration space, and their distributions resemble beta distributions. This could be an effect of MO-SMAC, which samples its hyperparameters from a configuration space with beta distributions. Furthermore, it could be an effect of the KDE algorithm. In the remainder of this subsection, we will cover interesting insights from these results.

#### Initial Learning Rate

The lower bound for the initial learning rate was set at $10^{-5}$. These results show that this lower bound could have been set at least a factor of $10^2$ higher, as the learning rate is never set below $10^{-3}$. Furthermore, we can see that the initial learning rate for the ResNet experiments is generally larger than that of the other experiments. This is as expected, as the ResNet datasets use a cosine decay learning rate scheduler, whereas the MLP datasets have a constant learning rate.

#### Epochs

Most models are trained for less than 100 epochs. Apparently, accuracy has severely converged at this point, making further training unnecessary. The models trained for Elec2 are an exception to this. This could result from the small dataset size of Elec2, making the number of epochs less influential on efficiency, or models just need to be trained for longer on this dataset.

#### Dropout

The dropout hyperparameter was limited to a small region centred at 0.3. Unsurprisingly, we see that this hyperparameter's density is centred around 0.3 for the MNIST and Fashion MNIST datasets. More surprising is the dropout behaviour in our tabular datasets, Higgs and Elec2. In both these trade-offs, with 206 and 57 configurations each, only one

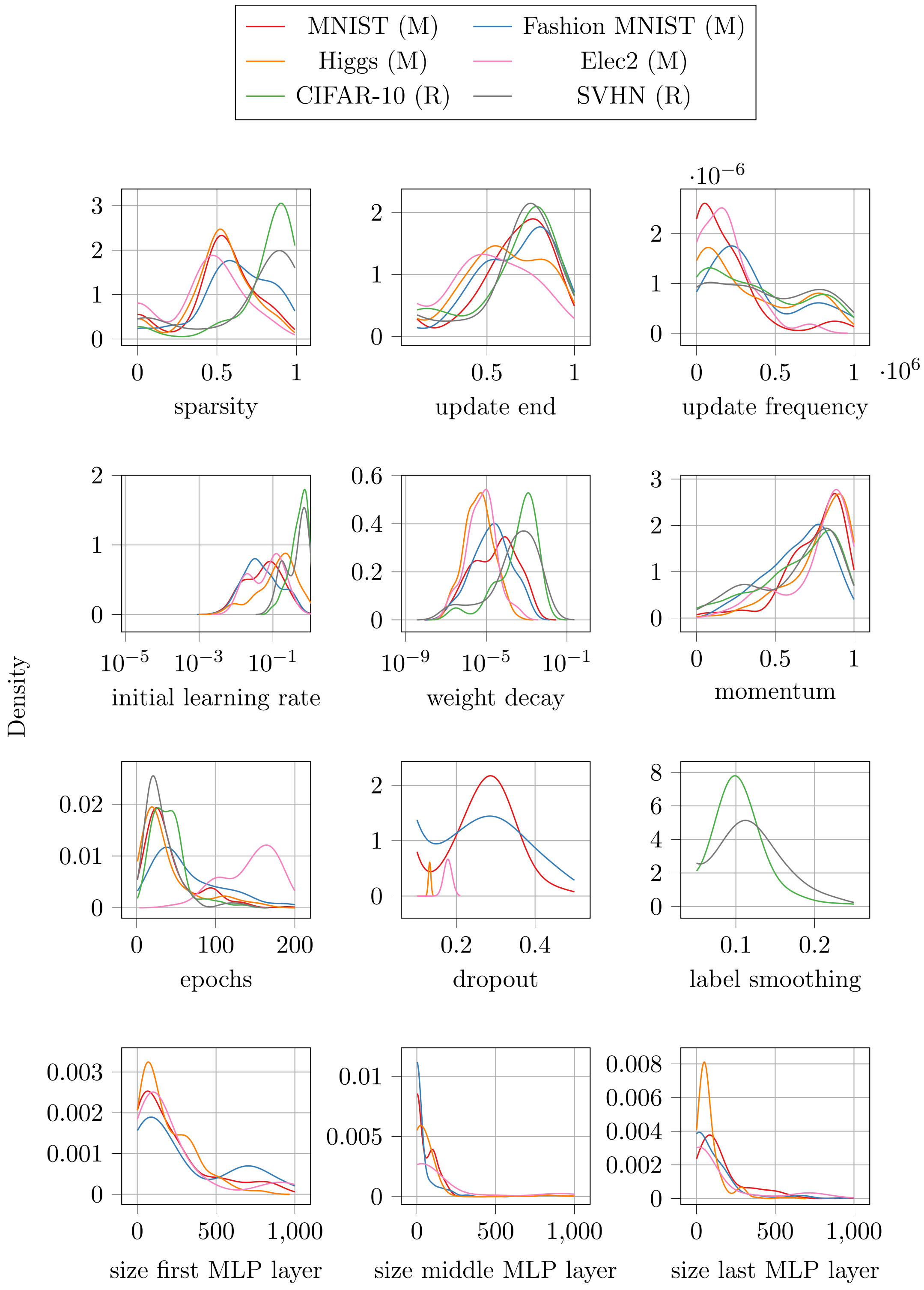


FIGURE 6.2: Kernel density estimations of 12 real and integer-valued hyperparameters in the optimization results. The number of samples for each dataset is stated in the first column of Table 5.6. Conditional hyperparameters (dropout, label smoothing, size middle/last MLP layer) have fewer samples.

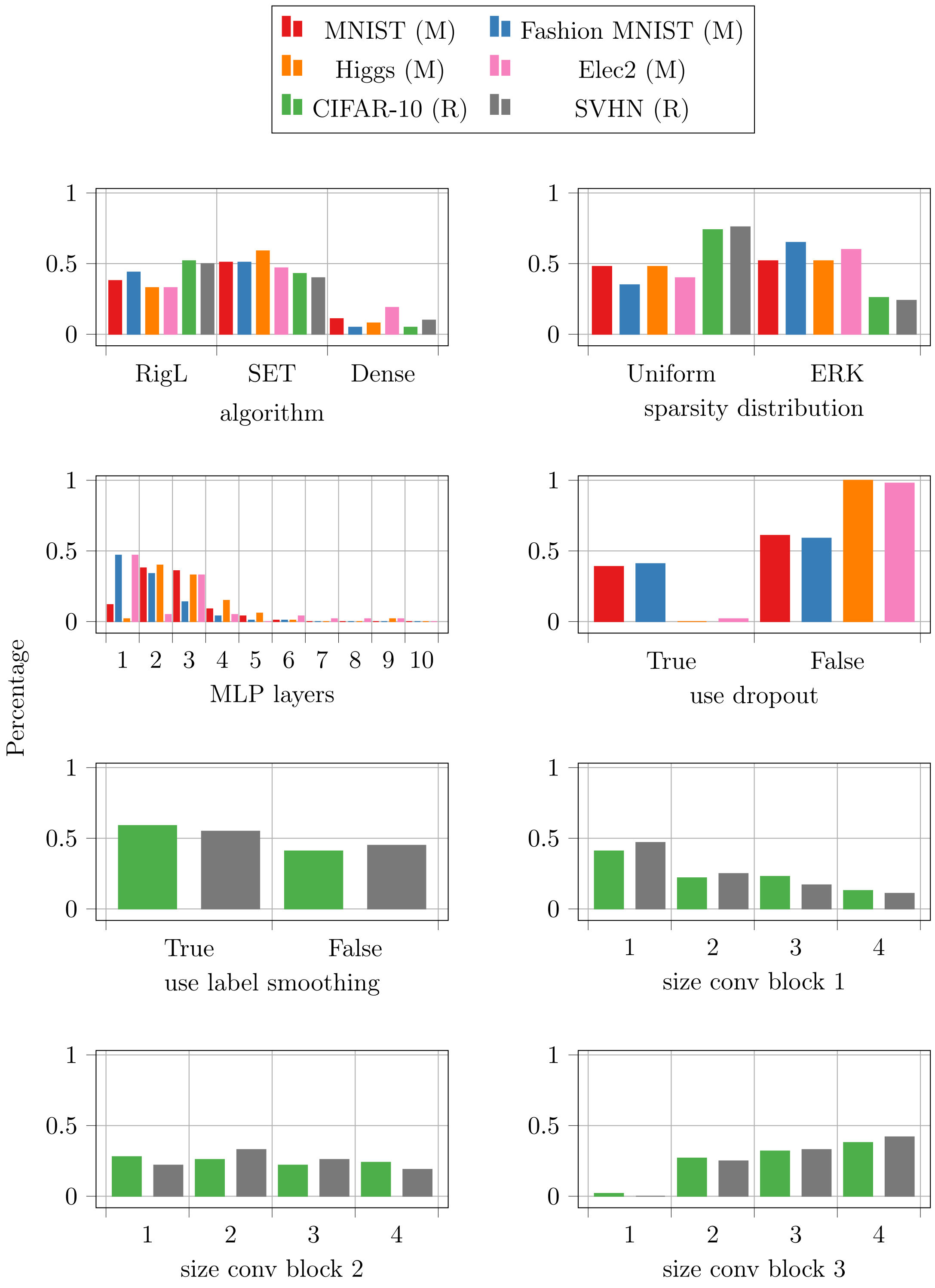


FIGURE 6.3: Value distributions of 6 categorical and small real-valued hyperparameters in the optimization results. The number of samples for each dataset is stated in the first column of Table 5.6. The sparsity distribution hyperparameter has fewer samples since it is conditional to SET and RigL.

configuration uses dropout. This is against our expectations, as a recent study has shown that regularization techniques such as dropout can substantially improve neural network performance on tabular datasets [56].

We do not know why dropout is optimized this way, but we believe it could indicate three things. First, our hyperparameter configuration space could somehow favour not using dropout in these problem settings. Second, these specific tabular datasets might behave differently from other tabular datasets. The Higgs dataset was already shown to be one of the datasets for which dropout could not improve accuracy in [56]. Third, dropout could have an unexpected relationship with SNN training methods in these datasets.

#### MLP Model Size

For all datasets modelled with MLPs, networks of only a few layers are sufficient. A small percentage of networks have over four layers, primarily in the tabular datasets. Furthermore, the sizes of these layers are kept at low values as well. Generally, the first layer is the largest, and the middle layer is the smallest. Our upper bound for each layer was set at 1000 neurons, equal to the sizes of all layers in [82]. However, this shows that much smaller networks are sufficient to model these datasets, with or without SNN training.

#### ResNet Model Size

The distributions of the ResNet model sizes are very comparable in both datasets. Here, we see two trends: the first convolutional block generally only has fewer layers than average, while the third block has more. This indicates that our optimized ResNets have increasing block sizes. This differs from how ResNets are initialized in [37], where the block sizes of small ResNets, such as ours, are kept equal.

### 6.2.2 Hypervolume Analysis - Method

We performed further analyses on the sparsity-related hyperparameters (SNN algorithm, sparsity, sparsity distribution, update frequency and update end) using the trade-off hypervolume. In these analyses, we filter on subsets of the configurations in the approximated Pareto fronts of the optimization experiment. We calculate the hypervolume for each subset and compare it to the hypervolume of all configurations. This shows which hyperparameter ranges cause the largest hypervolume improvements, i.e. give the best coverage of the trade-off.

When we analyze any hyperparameter, we can perform this analysis in four ways. First, we filter on all values and gradually increase the minimum value, causing a gradual decrease in hypervolume. Second, we can reverse this process, starting from only the lowest value and slowly increasing the maximum. Third, we can bin the hyperparameter space and calculate the hypervolume of each bin. In this analysis, the choice of cutoff points for the bins might have a major impact on the result, but an idea of the most relevant ranges can be extracted. Finally, we can calculate the hypervolume of all possible hyperparameter values, but this is only viable for categorical and small integer-valued hyperparameters.

### 6.2.3 Hypervolume Analysis - Results

#### Sparsity

First, we analyze sparsity. We conducted three analyses: increasing the minimal sparsity to demonstrate the extent of the trade-off that can be captured with sparsely trained models, increasing the maximal sparsity to emphasize the importance of high sparsity in covering the trade-off, and examining the distribution of sparsities over the trade-off in a binned hyperparameter space.

Results of these analyses can be found in Figure 6.4. By looking at Figure 6.4a where we filter by increasing the minimal sparsity, we see that near-maximal hypervolume is maintained up to a sparsity of 0.5 in all datasets. Thus, we can cover almost the entire trade-off with a sparsity of 0.5 or above. From this point on, the hypervolume starts to drop for most datasets at differing rates. An exception is the SVHN dataset, where near-maximal hypervolume is maintained until a sparsity of $\approx 0.8$. By looking at Figure 6.4b, we filter by increasing the maximal sparsity, we see most datasets converging slightly after a sparsity of 0.5 again, where SVHN needs sparsities of over 0.9 to near its complete hypervolume. We can also see how the datasets optimized with MLP architectures show a concave relationship between the maximal sparsity and hypervolume, whereas the ResNet datasets show a convex relationship. Finally, we see some repeating patterns with the previous graphs when looking at Figure 6.4c where we bin the sparsity range. The majority of the trade-off can be covered by using sparsities between 0.5 and 0.6 for most datasets. Exceptions to this are Elec2, for which we need slightly lower sparsities or (near) densely trained networks, and SVHN, where we capture the largest hypervolume between a sparsity of 0.8 and 0.9.

### Algorithm

Results of an analysis of the possible values for the SNN algorithm can be found in Figure 6.5a. The distribution of hypervolume over the different algorithms slightly differs from the distributions of the different algorithms themselves (Table 5.6). Where we previously saw a preference for SET or RigL in some datasets, here, the total hypervolume of SET and RigL is comparable in all datasets. This shows that every trade-off region that can be reached with one of the two methods can also be reached with the other. Furthermore, densely trained algorithms create a trade-off with a much smaller hypervolume comparable to their lesser prominence in Table 5.6.

### Sparsity Distribution

Results of an analysis of the possible values for the sparsity distribution can be found in Figure 6.5b. For each dataset, we see that one of the two distributions nearly covers the entire trade-off (ERK for MNIST, Fashion MNIST and Elec2, uniform for Higgs, CIFAR-10 and SVHN). This might indicate that uniform sparsity distributions are better suited for problems that are inherently less efficient; the optimized efficiency FLOPs for Higgs, CIFAR-10 and SVHN are an order of magnitude higher than those of the other datasets due to larger dataset sizes. However, the other sparsity distribution also always covers a substantial portion of the trade-off. Both distributions are relevant in all cases, making it hard to draw such a conclusion from these results.

Furthermore, these results contradict the findings of Evci et al. [26], showing that ERK distributions yield higher accuracies and lower efficiencies compared to uniform distributions. If this were consistently the case, ERK distributions should not cover the entire trade-off, as they would primarily be present in the highest accuracy - lowest efficiency region of the trade-off.

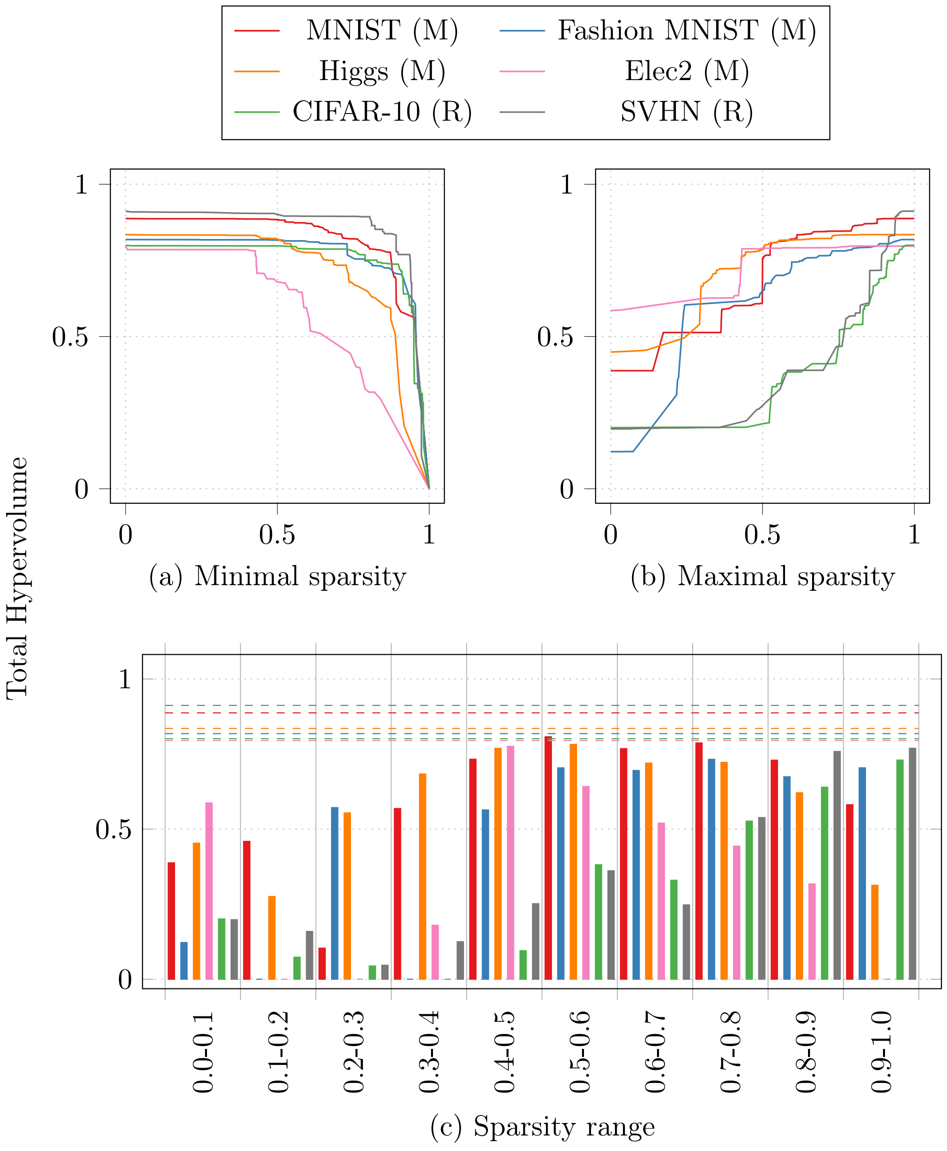


Figure 6.4: Total hypervolume of the optimization experiment results if we focus on specified sparsity values. **(a)** an increasing minimal sparsity. **(b)** an increasing maximal sparsity. **(c)** sparsity within fixed windows, with maximum hypervolumes added as dashed lines. The network architecture used for each dataset is specified as (M)LP and (R)esNet.

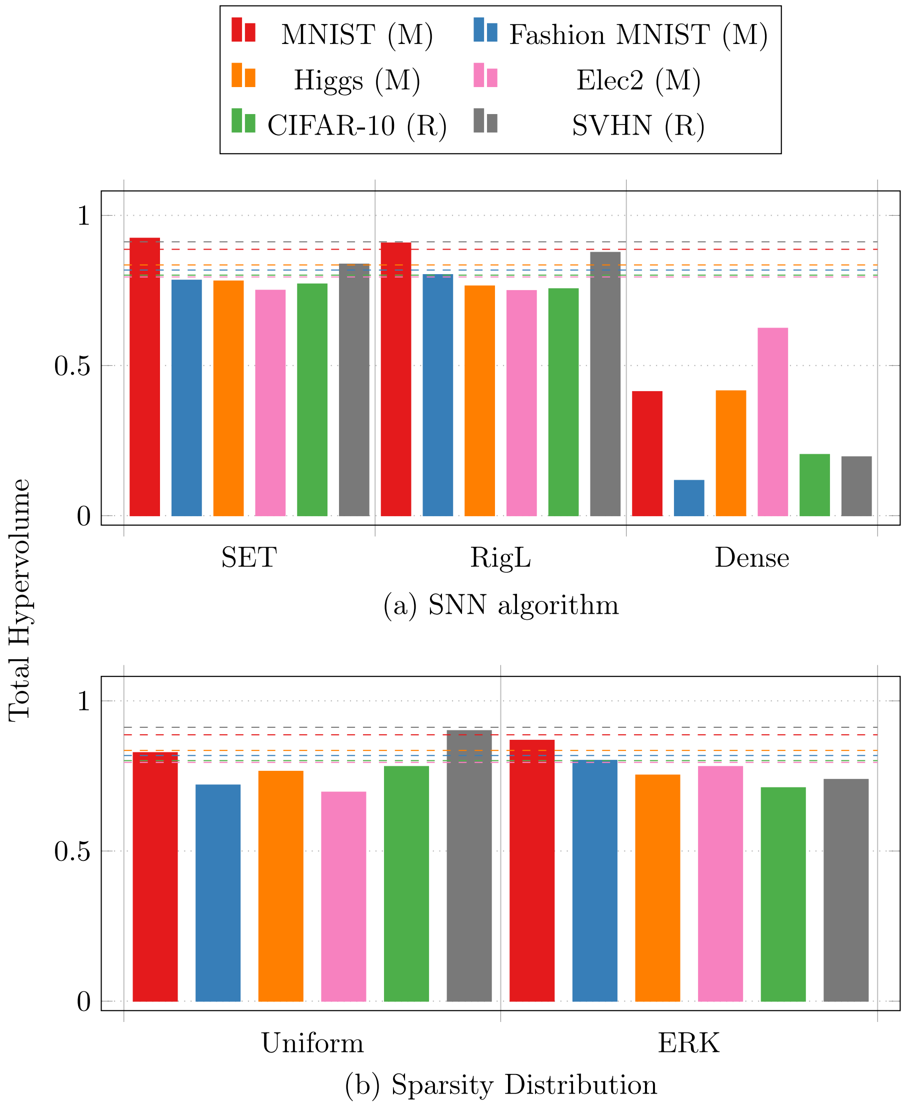


FIGURE 6.5: Total hypervolume of the optimization experiment results if filtered on specific SNN algorithms **(a)** or sparsity distributions **(b)**. Maximum hypervolumes of all datasets are added as dashed lines. The network architecture used for each dataset is specified as (M)LP and (R)esNet.

**Update Frequency & Update End**

Three types of hypervolume analyses on the update frequency and update end hyperparameters are shown in Figure 6.6. These plots seem to mirror the results of the density estimations in Figure 6.2, focusing on low update frequency and high update end values.

### 6.2.4 Subset Selection

To further study the optimal values of these hyperparameters, we have used a biclustering algorithm to perform subset selection on the configuration and objective space simultaneously. For this, we implemented[1] PAN [110], an algorithm specifically designed to find subsets in a multi-objective optimization trade-off. Unfortunately, this did not lead to clear subsets, indicating the need to perform hyperparameter optimization if we wish to find a configuration on the trade-off.

### 6.2.5 Summary

Together, these results lead us to answer our second research question: how do sparsity and other hyperparameters influence the accuracy-efficiency trade-off in sparse neural network training? Whereas multiple hyperparameters are essential in determining neural network performance, sparsity is the most important hyperparameter to trade off accuracy for efficiency. Dense neural networks should only be trained if efficiency is not one of our objectives. Otherwise, configurations optimized for both objectives can be defined with either SET or RigL, with preferred sparsities ranging from 0.4 to 0.8 for MLP architectures and 0.7 to 0.99 for ResNet architectures. The influence on the trade-off is more subtle for other hyperparameters. Optimal values for these hyperparameters differ greatly per use case, and we have shown that our ranges allow HPO to find these optimal values.

[1] Available at https://pypi.org/project/pan-biclustering/

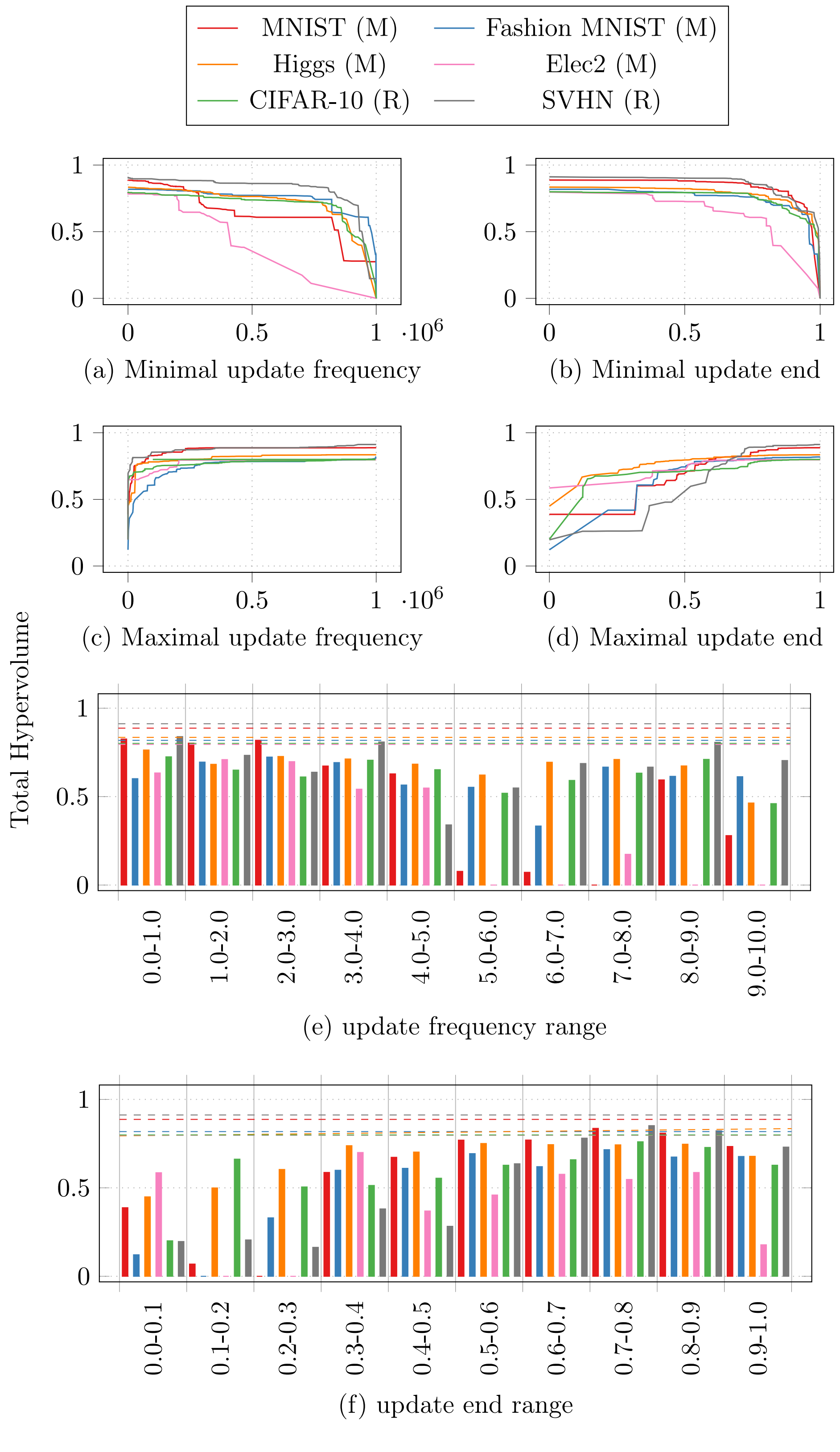


FIGURE 6.6: Total hypervolume of the optimization experiment results if we focus on specified update frequency or update end values. **(a)** and **(b)** increasing minimal values. **(d)** **(d)** increasing maximal values. **(e)** and **(f)** binned values, with maximum hypervolumes added as dashed lines. The network architecture used for each dataset is specified as (M)LP and (R)esNet.

# Chapter 7

# Conclusions and Discussion

This research aimed to improve our knowledge of the sustainability of deep neural networks. These networks require a lot of energy to be trained and used but can be made more efficient with SNN training. However, it was previously unknown how much more efficient these models could become and how this would affect their performance. Therefore, we set out to answer two research questions: what does the accuracy-efficiency trade-off in sparse neural network training look like in an environment with many configurable hyperparameters? And how do sparsity and other hyperparameters influence the accuracy-efficiency trade-off in sparse neural network training? To answer these questions, we have selected six benchmark classification datasets and performed three experiments on them. These experiments show a novel way to investigate multi-objective technologies using MO-HPO. We evaluated the results of current literature on SNN training, sought an accuracy-efficiency trade-off using MO-HPO and evaluated our method against that of previous research. Furthermore, we performed multiple analyses on the objective and hyperparameter configuration spaces of these results, gaining valuable insights into the possibilities of SNN training. In this chapter, we conclude on our findings and give directions for future research.

## 7.1 Conclusions

We have used MO-HPO in a hyperparameter configuration space with SNN training to present an approximated Pareto front for three objectives: accuracy, inference efficiency and training efficiency. We compared these results against the results when approximating this trade-off in a traditional way; by structurally applying SNN training to an already optimized configuration. Our results (Section 5.2) show that our method finds a much better approximation of the true accuracy-efficiency trade-off, giving us more insights into the possibilities when optimizing efficiency. However, we also see that these optimization results are not guaranteed to find configurations that yield higher accuracies than the configurations used in traditional research. Furthermore, we have seen that a combination of these two techniques (Section 5.3) often finds the best results, a comparable approximation of the trade-off, with better-performing configurations if we can afford a low efficiency. By analysis of the effect of SNN training on these results, we see that we cannot generalize the effects of increasing sparsity in arbitrary optimized configurations. Still, SNN training plays a vital role in all our approximated accuracy-efficiency trade-offs and allows for a larger increase in efficiency with a small decrease in accuracy than using smaller dense networks (Section 6.2).

Combined, we conclude that the accuracy-efficiency trade-off in an SNN training environment is of an exponential nature. The networks with the highest accuracy are often trained dense or with low sparsity. However, if we are willing to trade off a portion of this accuracy for a higher efficiency, a linear decrease in accuracy can cause an exponential increase in efficiency. The best way to find such configurations is by generating several well-performing configurations using HPO and by testing several desired sparsities on these configurations. Here, SET and RigL are both valid options. Furthermore, we see that inference and training efficiency are strongly correlated in these trade-offs, and the only way to reduce training efficiency compared to inference efficiency is by lowering the number of training epochs. These trade-offs behave similarly for all datasets and network architectures, but SNN training has a stronger influence on ResNets than MLPs.

## 7.2 Limitations and Future Work

In this section, we list several limitations in our research that should be taken into consideration when reading this work. Many of these limitations directly point toward potential directions for future research to improve our findings. Furthermore, we elaborate on potential other interesting future research directions for which this work can provide a basis.

#### Hardware Limitations of Sparse Matrices

Due to the practical implications of SNN training, our results might need to be taken with a grain of salt. We calculate efficiency using the required number of FLOPs. However, we know that sparse matrix calculations and sparse matrix storage are not as optimized as this estimation (Subsection 2.2.1). The efficiency gain of sparse matrix calculations depends heavily on implementation details and hardware specifics and is difficult to predict. Furthermore, sparse matrices can often be stored at a reduced size if their sparsity exceeds $\approx 0.66$ (Subsection 2.2.1). Our results find that useful sparsity values often lie below this threshold, causing little to no improvement in the storage of these networks, limiting the advantages of SNNs.

#### Measuring Neural Network Efficiency

While the FLOPs measurement makes our results most comparable to previous research, it does not fully capture efficiency in practice. Still, it is currently the best measurement for neural network efficiency. A more realistic efficiency measurement could be defined, incorporating, for example, the overhead of sparse matrix storage and multiplication and inefficiency of memory accesses. Using such a measurement in approximating the trade-off could favour certain configurations which are now estimated as inefficient or find that the trade-off is not as clear as our results make it seem.

#### Generalization of Problem Space

We have chosen benchmark datasets to make the results generalizable to future research. However, we still limited ourselves to balanced classification problems consisting of two data types. Many real-life applications of deep learning differ in these restrictions, and we do not know whether SNN training behaves differently in such environments. Future research could study different problem settings and how the effect of SNN training results differs. Options that might be considered are imbalanced datasets or regression problems. Furthermore, we have focused on the top-1 accuracy of classification problems while each neural network outputs a probability distribution for all classes. The effect of SNN training on this complete distribution, as opposed to merely the accuracy, remains unknown.

### Initializing MO-HPO

The optimization process can be altered in several ways to generate more realistic and generalizable results. First, AAC frameworks have many options and initialization settings that affect their performance. Examples of such options in MO-SMAC are the number of instances, maximum budget, default configuration and the number of incumbents to keep track of. Currently, it is hard to ensure that MO-SMAC finds the best configurations as little research has been put into how these settings should be chosen. Future work could further investigate the effects of MO-HPO settings and how they should be initialized. Second, we have approximated the entire trade-off, whereas we later only focused on a smaller region of the trade-off since we are not interested in models with unusable accuracies. For this reason, it could be worthwhile to focus our trade-off optimization on this trade-off region. This could be achieved with a preference-based MO-HPO optimizer.

### Extended Hyperparameter Configuration Space

The configuration space could be further altered to better represent the model architectures used in reality. Furthermore, such a study could further study the topological features of the network structures found in the accuracy-efficiency trade-off and their relation to SNN training. While this is difficult to incorporate in a fixed hyperparameter configuration space as defined in Appendix A, more complex architectures can be found using a complete NAS [25]. Other than implementing a NAS, other architectural types can also be added to the configuration space. We limited ourselves to MLPs and ResNets, but an extension to architectures such as recurrent neural networks could provide useful insights depending on the datasets studied.

Furthermore, a broader range of efficiency-improving methods could be studied. We focused on two dynamic sparse training algorithms compared to dense training in neural networks. We have shown that the SNN algorithms are comparable in many cases and that there is no clear preference for one or the other. Other methods might produce better results in certain instances or have a greater impact on the trade-off between the two efficiency objectives. Examples are static sparse training or post-training pruning, dense efficiency-improving techniques such as ensembling or knowledge distillation, or even alternative ML algorithms such as decision trees and SVMs.

### Hyperparameter Analysis

We have performed a global analysis on the optimal distributions and relative importance of all hyperparameters and performed a more thorough analysis focused on the sparsity-related hyperparameters. However, we believe that many more insights can be drawn from our results, such as the relationships between different hyperparameters. To aid such research, we have published all evaluated configurations and their objectives[1].

## 7.3 Final Thoughts

To conclude, we believe there is still much to be studied before we fully understand the complex effects of SNN training. However, we have made substantial progress by showing that sparse training is the best way to increase model efficiency and that dynamic sparse training almost always reduces model accuracy compared to dense training. Furthermore, we have explored a new way to evaluate complex algorithms influencing multiple objectives

[1] Available at `https://github.com/zwouter/sparse-training-environment/tree/main`

by using MO-AAC to approximate the underlying trade-off and use this in further analysis. This study opens the field of efficient neural network training to an abundance of potential future research and paves the way for structural analysis of complex algorithms.

# Bibliography

[1] TensorFlow Datasets, a collection of ready-to-use datasets. https://www.tensorflow.org/datasets.

[2] Majid Abdolshah, Alistair Shilton, Santu Rana, Sunil Gupta, and Svetha Venkatesh. Multi-objective Bayesian optimisation with preferences over objectives. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/a7b7e4b27722574c611fe91476a50238-Abstract.html.

[3] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, July 2020. URL: https://arxiv.org/abs/2007.03051.

[4] Zahra Atashgahi, Mykola Pechenizkiy, Raymond Veldhuis, and Decebal Constantin Mocanu. Adaptive Sparsity Level During Training for Efficient Time Series Forecasting with Transformers. In Albert Bifet, Jesse Davis, Tomas Krilavičius, Meelis Kull, Eirini Ntoutsi, and Indrė Žliobaitė, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track*, pages 3–20, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-70341-6_1.

[5] Sunith Bandaru, Amos H. C. Ng, and Kalyanmoy Deb. Data mining methods for knowledge discovery in multi-objective optimization: Part A - Survey. *Expert Systems with Applications*, 70:139–159, March 2017. URL: https://www.sciencedirect.com/science/article/pii/S0957417416305449, doi:10.1016/j.eswa.2016.10.015.

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, February 2012. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[7] André Biedenkapp, Marius Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, and Holger Hoos. Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. Issue: 1. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10657.

[8] Martin Binder, Julia Moosbauer, Janek Thomas, and Bernd Bischl. Multi-objective hyperparameter tuning and feature selection using filter ensembles. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 471–479, New York, NY, USA, June 2020. Association for Computing Machinery. 20 citations (Crossref) [2024-03-22]. URL: https://dl.acm.org/doi/10.1145/3377930.3389815, doi:10.1145/3377930.3389815.

[9] Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, et al. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(2):e1484, 2023.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[11] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the State of Neural Network Pruning? *Proceedings of Machine Learning and Systems*, 2:129–146, March 2020. URL: https://proceedings.mlsys.org/paper_files/paper/2020/hash/6c44dc73014d66ba49b28d483a8f8b0d-Abstract.html.

[12] Aymeric Blot, Holger H. Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. MO-ParamILS: A Multi-objective Automatic Algorithm Configuration Framework. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization*, pages 32–47, Cham, 2016. Springer International Publishing.

[13] Léon Bottou. On-line Learning and Stochastic Approximations. In David Saad, editor, *On-Line Learning in Neural Networks*, pages 9–42. Cambridge University Press, 1 edition, January 1999. URL: https://www.cambridge.org/core/product/identifier/CBO9780511569920A009/type/book_part, doi:10.1017/CBO9780511569920.003.

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL: https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[16] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–20, 2024. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence. URL: https://ieeexplore.ieee.org/abstract/document/10643325, doi:10.1109/TPAMI.2024.3447085.

[17] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. URL: http://link.springer.com/10.1007/BF00994018, doi:10.1007/BF00994018.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002. URL: https://ieeexplore.ieee.org/document/996017, doi:10.1109/4235.996017.

[19] DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL: http://github.com/google-deepmind.

[20] Tim Dettmers and Luke Zettlemoyer. Sparse Networks from Scratch: Faster Training without Losing Performance, August 2019. arXiv:1907.04840 [cs, stat]. URL: http://arxiv.org/abs/1907.04840, doi:10.48550/arXiv.1907.04840.

[21] Hyungrok Do, Myun-Seok Cheon, and Seoung Bum Kim. Graph Structured Sparse Subset Selection. *Information Sciences*, 518:71–94, May 2020. URL: https://www.sciencedirect.com/science/article/pii/S0020025519312125, doi:10.1016/j.ins.2019.12.086.

[22] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, June 2021. arXiv:2010.11929 [cs]. URL: http://arxiv.org/abs/2010.11929, doi:10.48550/arXiv.2010.11929.

[23] Matthias Ehrgott. *Multicriteria optimization: with 88 figures and 12 tables*. Springer, Berlin Heidelberg New York, second edition edition, 2005.

[24] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=ByME42AqK7.

[25] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL: http://jmlr.org/papers/v20/18-598.html.

[26] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the Lottery: Making All Tickets Winners. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2943–2952. PMLR, July 2020. URL: https://proceedings.mlr.press/v119/evci20a.html.

[27] Chris Fawcett and Holger H. Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, August

2016. URL: http://link.springer.com/10.1007/s10732-014-9275-9, doi:10.1007/s10732-014-9275-9.

[28] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022. Publisher: Nature Publishing Group. URL: https://www.nature.com/articles/s41586-022-05172-4, doi:10.1038/s41586-022-05172-4.

[29] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *7th International Conference on Learning Representations, ICLR*. OpenReview.net, May 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[30] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A Systematic Survey of General Sparse Matrix-matrix Multiplication. *ACM Computing Surveys*, 55(12):244:1–244:36, March 2023. URL: https://dl.acm.org/doi/10.1145/3571157, doi:10.1145/3571157.

[31] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A Survey of Quantization Methods for Efficient Neural Network Inference, June 2021. arXiv:2103.13630 [cs]. URL: http://arxiv.org/abs/2103.13630.

[32] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *NeurIPS 2022 Datasets and Benchmarks Track*, Advances in Neural Information Processing, New Orleans, United States, November 2022. URL: https://hal.science/hal-03723551.

[33] Andreia P. Guerreiro, Carlos M. Fonseca, and Luís Paquete. The Hypervolume Indicator: Computational Problems and Algorithms. *ACM Computing Surveys*, 54(6):1–42, July 2022. URL: https://dl.acm.org/doi/10.1145/3453474, doi:10.1145/3453474.

[34] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL: https://papers.nips.cc/paper_files/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html.

[35] M. Harries, University of New South Wales School of Computer Science, and Engineering. *Splice-2 Comparative Evaluation: Electricity Pricing*. PANDORA electronic collection. University of New South Wales, School of Computer Science and Engineering, 1999. URL: https://books.google.nl/books?id=1Zr1vQAACAAJ.

[36] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. *Math. Intell.*, 27:83–85, November 2004. doi:10.1007/BF02985802.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385 [cs]. URL: http://arxiv.org/abs/1512.03385, doi:10.48550/arXiv.1512.03385.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.

[39] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL: http://github.com/google/flax.

[40] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network, March 2015. arXiv:1503.02531 [cs, stat]. URL: http://arxiv.org/abs/1503.02531, doi:10.48550/arXiv.1503.02531.

[41] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, August 1995. URL: https://ieeexplore.ieee.org/document/598994, doi:10.1109/ICDAR.1995.598994.

[42] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(1), jan 2021.

[43] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, February 2012. URL: https://dl.acm.org/doi/10.1145/2076450.2076469, doi:10.1145/2076450.2076469.

[44] Daniel Horn and Bernd Bischl. Multi-objective parameter configuration of machine learning algorithms using model-based optimization. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, December 2016. URL: https://ieeexplore.ieee.org/document/7850221, doi:10.1109/SSCI.2016.7850221.

[45] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, January 1989. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208, doi:10.1016/0893-6080(89)90020-8.

[46] Junhao Huang, Weize Sun, and Lei Huang. Joint Structure and Parameter Optimization of Multiobjective Sparse Neural Network. *Neural Computation*, 33(4):1113–1143, 03 2021. URL: https://doi.org/10.1162/neco_a_01368, arXiv:https://direct.mit.edu/neco/article-pdf/33/4/1113/1902353/neco_a_01368.pdf, doi:10.1162/neco_a_01368.

[47] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An Efficient Approach for Assessing Hyperparameter Importance. In *Proceedings of the 31st International Conference on Machine Learning*, pages 754–762. PMLR, January 2014. URL: https://proceedings.mlr.press/v32/hutter14.html.

[48] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25566-3_40.

[49] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic Algorithm Configuration Based On Local Search. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1152–1157, Vancouver,

British Columbia, Canada, July 2007. AAAI Press. ParamILS. URL: https://dl.acm.org/doi/10.5555/1619797.1619831.

[50] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019. URL: http://link.springer.com/10.1007/978-3-030-05318-5.

[51] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. The Configurable SAT Solver Challenge (CSSC). *Artificial Intelligence*, 243:1–25, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0004370216301138, doi:10.1016/j.artint.2016.09.006.

[52] Yerlan Idelbayev, Arman Zharmagambetov, Magzhan Gabidolla, and Miguel A. Carreira-Perpinan. Faster Neural Net Inference via Forests of Sparse Oblique Decision Trees. October 2021. URL: https://openreview.net/forum?id=yulAchHedcT&referrer=%5Bthe%20profile%20of%20Arman%20Zharmagambetov%5D(%2Fprofile%3Fid%3D~Arman_Zharmagambetov1).

[53] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, 13(4):455–492, December 1998. doi:10.1023/A:1008306431147.

[54] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, August 2021. doi:10.1038/s41586-021-03819-2.

[55] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, page 751–756, NLD, 2010. IOS Press. doi:10.3233/978-1-60750-606-5-751.

[56] Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. *Regularization is all you Need: Simple Neural Nets can Excel on Tabular Data*. June 2021. doi:10.48550/arXiv.2106.11189.

[57] Florian Karl, Tobias Pielok, Julia Moosbauer, Florian Pfisterer, Stefan Coors, Martin Binder, Lennart Schneider, Janek Thomas, Jakob Richter, Michel Lang, Eduardo C. Garrido-Merchán, Juergen Branke, and Bernd Bischl. Multi-Objective Hyperparameter Optimization in Machine Learning—An Overview. *ACM Transactions on Evolutionary Learning and Optimization*, 3(4):16:1–16:50, December 2023. URL: https://dl.acm.org/doi/10.1145/3610536, doi:10.1145/3610536.

[58] Jeremy Kepner and Ryan Robinett. RadiX-Net: Structured Sparse Matrices for Deep Neural Networks. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 268–274, May 2019. doi:10.1109/IPDPSW.2019.00051.

[59] Joshua Knowles. ParEGO: A Hybrid Algorithm with On-line Landscape Approximation for Expensive Multiobjective Optimization Problems. Technical Report TR-COMPSYSBIO-2004-01, University of Manchester, September 2004. doi:10.1109/TEVC.2005.851274.

[60] Joshua D. Knowles, Richard A. Watson, and David Corne. Reducing Local Optima in Single-Objective Problems by Multi-objectivization. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, EMO '01, pages 269–283, Berlin, Heidelberg, March 2001. Springer-Verlag. URL: https://dl.acm.org/doi/10.5555/647889.736521.

[61] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[63] Abhisek Kundu, Naveen Mellempudi, Dharma Vooturi, Bharat Kaul, and Pradeep Dubey. *AUTOSPARSE: Towards Automated Sparse Training of Deep Neural Networks*. April 2023. URL: http://arxiv.org/abs/2304.06941.

[64] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE. URL: https://ieeexplore.ieee.org/document/726791, doi:10.1109/5.726791.

[65] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. Publisher: Nature Publishing Group. URL: https://www.nature.com/articles/nature14539, doi:10.1038/nature14539.

[66] Yann LeCun, John Denker, and Sara Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper/1989/hash/6c9882bbac1c7093bd25041881277658-Abstract.html.

[67] Joo Hyung Lee, Wonpyo Park, Nicole Mitchell, Jonathan Pilault, Johan Obando-Ceron, Han-Byul Kim, Namhoon Lee, Elias Frantar, Yun Long, Amir Yazdanbakhsh, Shivani Agrawal, Suvinay Subramanian, Xin Wang, Sheng-Chun Kao, Xingyao Zhang, Trevor Gale, Aart Bik, Woohyun Han, Milen Ferev, Zhonglin Han, Hong-Seok Kim, Yann Dauphin, Gintare Karolina Dziugaite, Pablo Samuel Castro, and Utku Evci. JaxPruner: A concise library for sparsity research, December 2023. arXiv:2304.14082 [cs]. URL: http://arxiv.org/abs/2304.14082, doi:10.48550/arXiv.2304.14082.

[68] Sheng Li, Mingxing Tan, Ruoming Pang, Andrew Li, Liqun Cheng, Quoc V. Le, and Norman P. Jouppi. Searching for Fast Model Families on Datacenter Accelerators. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8081–8091, June 2021. ISSN: 2575-7075. URL: https://ieeexplore.ieee.org/document/9578371, doi:10.1109/CVPR46437.2021.00799.

[69] Zhengpin Li, Zheng Wei, Jian Wang, Yun Lin, and Byonghyo Shim. Fast Graph Subset Selection Based on G-optimal Design, December 2021. arXiv:2112.15403 [cs, eess, math]. URL: http://arxiv.org/abs/2112.15403, doi:10.48550/arXiv.2112.15403.

[70] Zhouhan Lin, Roland Memisevic, and Kishore Konda. How far can we go without convolution: Improving fully-connected networks, November 2015. arXiv:1511.02580 [cs]. URL: http://arxiv.org/abs/1511.02580.

[71] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL: http://jmlr.org/papers/v23/21-0888.html.

[72] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4876–4883, April 2020. 147 citations (Semantic Scholar/DOI) [2024-03-22] 96 citations (Crossref) [2024-03-22] Number: 04. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5924, doi:10.1609/aaai.v34i04.5924.

[73] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Zahra Atashgahi, Lu Yin, Huanyu Kou, Li Shen, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. Sparse Training via Boosting Pruning Plasticity with Neuroregeneration. In *Advances in Neural Information Processing Systems*, volume 34, pages 9908–9922. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/5227b6aaf294f5f027273aebf16015f2-Abstract.html.

[74] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated Local Search. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 320–353. Springer US, Boston, MA, 2003. doi:10.1007/0-306-48056-5_11.

[75] Hongyin Luo and Wei Sun. Addition is All You Need for Energy-efficient Language Models, October 2024. arXiv:2410.00907 version: 2. URL: http://arxiv.org/abs/2410.00907, doi:10.48550/arXiv.2410.00907.

[76] Jonna Matthiesen. The Impact of Deep Neural Network Pruning on the Hyperparameter Performance Space: An Empirical Study. October 2023. URL: https://gupea.ub.gu.se/handle/2077/78958.

[77] M. D. McKay, R. J. Beckman, and W. J. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21(2):239, May 1979. URL: https://www.jstor.org/stable/1268522?origin=crossref, doi:10.2307/1268522.

[78] Kaisa Miettinen. *Nonlinear Multiobjective Optimization*, volume 12 of *International Series in Operations Research & Management Science*. Springer US, Boston, MA, 1998. URL: http://link.springer.com/10.1007/978-1-4615-5563-6, doi:10.1007/978-1-4615-5563-6.

[79] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating Sparse Deep Neural

Networks, April 2021. arXiv:2104.08378 [cs]. URL: http://arxiv.org/abs/2104.08378, doi:10.48550/arXiv.2104.08378.

[80] Decebal Constantin Mocanu, Elena Mocanu, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. A topological insight into restricted Boltzmann machines. *Machine Learning*, 104(2):243–270, September 2016. doi:10.1007/s10994-016-5570-z.

[81] Decebal Constantin Mocanu, Elena Mocanu, Tiago Pinto, Selima Curci, Phuong H. Nguyen, Madeleine Gibescu, Damien Ernst, and Zita A. Vale. Sparse Training Theory for Scalable and Efficient Agents. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '21, page 34–38, Richland, SC, 2021. International Foundation for Autonomous Agents and Multiagent Systems. URL: https://dl.acm.org/doi/10.5555/3463952.3463960.

[82] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, June 2018. URL: https://www.nature.com/articles/s41467-018-04316-3, doi:10.1038/s41467-018-04316-3.

[83] Julia Moosbauer, Julia Herbinger, Giuseppe Casalicchio, Marius Lindauer, and Bernd Bischl. Explaining Hyperparameter Optimization via Partial Dependence Plots. In *Advances in Neural Information Processing Systems*, volume 34, pages 2280–2291. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/12ced2db6f0193dda91ba86224ea1cd8-Abstract.html.

[84] Thomas Moreau, Mathurin Massias, Alexandre Gramfort, Pierre Ablin, Pierre-Antoine Bannier, Benjamin Charlier, Mathieu Dagréou, Tom Dupre la Tour, Ghislain DURIF, Cassio F. Dantas, Quentin Klopfenstein, Johan Larsson, En Lai, Tanguy Lefort, Benoît Malézieux, Badr MOUFAD, Binh T. Nguyen, Alain Rakotomamonjy, Zaccharie Ramzi, Joseph Salmon, and Samuel Vaiter. Benchopt: Reproducible, efficient and collaborative optimization benchmarks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 25404–25421. Curran Associates, Inc., 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/a30769d9b62c9b94b72e21e0ca73f338-Paper-Conference.pdf.

[85] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *Proceedings of the 36th International Conference on Machine Learning*, pages 4646–4655. PMLR, May 2019. ISSN: 2640-3498. URL: https://proceedings.mlr.press/v97/mostafa19a.html.

[86] Michael C Mozer and Paul Smolensky. Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment. In *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988. URL: https://proceedings.neurips.cc/paper/1988/hash/07e1cd7dca89a1678042477183b7ac3f-Abstract.html.

[87] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011. URL: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[88] Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. A Flexible Framework for Multi-Objective Bayesian Optimization using Random Scalarizations. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 766–776. AUAI Press, 2019. URL: http://proceedings.mlr.press/v115/paria20a.html.

[89] Maryam Parsa, Aayush Ankit, Amirkoushyar Ziabari, and Kaushik Roy. PABO: Pseudo Agent-Based Multi-Objective Bayesian Hyperparameter Optimization for Efficient Neural Accelerator Design. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, November 2019. 24 citations (Semantic Scholar/DOI) [2024-05-10]. URL: https://ieeexplore.ieee.org/document/8942046, doi:10.1109/ICCAD45719.2019.8942046.

[90] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. *Computer*, 55(7):18–28, 2022. doi:10.1109/MC.2022.3148714.

[91] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: importance of hyperparameters of machine learning algorithms. *J. Mach. Learn. Res.*, 20(1):1934–1965, jan 2019. URL: https://dl.acm.org/doi/10.5555/3322706.3361994.

[92] Leslie Pérez Cáceres and Thomas Stützle. Exploring variable neighborhood search for automatic algorithm configuration. *Electronic Notes in Discrete Mathematics*, 58, April 2017. doi:10.1016/j.endm.2017.03.022.

[93] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, March 1986. doi:10.1007/BF00116251.

[94] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, September 1951. Publisher: Institute of Mathematical Statistics. URL: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-3/A-Stochastic-Approximation-Method/10.1214/aoms/1177729586.full, doi:10.1214/aoms/1177729586.

[95] Jeroen Rook, Carolin Benjamins, Jakob Bossek, Heike Trautmann, Holger Hoos, and Marius Lindauer. MO-SMAC: Multi-objective sequential model-based algorithm configuration. *Manuscript under review*, pages 1–23, 2024.

[96] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. Place: US Publisher: American Psychological Association. doi:10.1037/h0042519.

[97] Max Roser. The brief history of artificial intelligence: The world has changed fast – what might be next? *Our World in Data*, 2022. https://ourworldindata.org/brief-history-of-ai.

[98] Elias Schede, Jasmin Brandt, Alexander Tornede, Marcel Wever, Viktor Bengs, Eyke Hüllermeier, and Kevin Tierney. A Survey of Methods for Automated Algorithm Configuration. *J. Artif. Int. Res.*, 75, dec 2022. doi:10.1613/jair.1.13676.

[99] V. Schmidt, K. Goyal, A. Joshi, B. Feld, L. Conell, N. Laskaris, D. Blank, J. Wilson, S. Friedler, and S. Luccioni. CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing., 2021. doi:10.5281/zenodo.11097062.

[100] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Commun. ACM*, 63(12):54–63, November 2020. URL: https://dl.acm.org/doi/10.1145/3381831, doi:10.1145/3381831.

[101] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, April 2015. arXiv:1409.1556 [cs] version: 6. URL: http://arxiv.org/abs/1409.1556, doi:10.48550/arXiv.1409.1556.

[102] Henrik Smedberg and Sunith Bandaru. Interactive knowledge discovery and knowledge visualization for decision support in multi-objective optimization. *European Journal of Operational Research*, 306(3):1311–1329, May 2023. 16 citations (Semantic Scholar/DOI) [2024-03-25]. URL: https://www.sciencedirect.com/science/article/pii/S0377221722007202, doi:10.1016/j.ejor.2022.09.008.

[103] Henrik Smedberg, Sunith Bandaru, Amos H.C. Ng, and Kalyanmoy Deb. Trend Mining 2.0: Automating the Discovery of Variable Trends in the Objective Space. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, Glasgow, United Kingdom, July 2020. IEEE. 1 citations (Semantic Scholar/DOI) [2024-03-25]. URL: https://ieeexplore.ieee.org/document/9185892/, doi:10.1109/CEC48606.2020.9185892.

[104] Paul Smolensky et al. Information processing in dynamical systems: Foundations of harmony theory. 1986. URL: https://stanford.edu/~jlmcc/papers/PDP/Volume%201/Chap6_PDP86.pdf.

[105] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[106] E. H. Steerneman. Exploring the effect of merging techniques on the performance of merged sparse neural networks in a highly distributed setting, July 2022. Publisher: University of Twente. URL: https://essay.utwente.nl/91995/.

[107] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics. URL: https://aclanthology.org/P19-1355, doi:10.18653/v1/P19-1355.

[108] Luke Taylor, Andrew King, and Nicol Harper. Robust and accelerated single-spike spiking neural network training with applicability to challenging temporal tasks, October 2022. arXiv:2205.15286 [cs]. URL: http://arxiv.org/abs/2205.15286.

[109] Daphne Theodorakopoulos, Frederic Stahl, and Marius Lindauer. Hyperparameter Importance Analysis for Multi-Objective AutoML, May 2024. arXiv:2405.07640 [cs]. URL: http://arxiv.org/abs/2405.07640, doi:10.48550/arXiv.2405.07640.

[110] Tamara Ulrich. Pareto-Set Analysis: Biobjective Clustering in Decision and Objective Spaces. *Journal of Multi-Criteria Decision Analysis*, 20(5-6):217–234, 2013. 18 citations (Semantic Scholar/DOI) [2024-03-25]. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mcda.1477, doi:10.1002/mcda.1477.

[111] Aimee van Wynsberghe. Sustainable AI: AI for sustainability and the sustainability of AI. *AI and Ethics*, 1(3):213–218, August 2021. 194 citations (Semantic Scholar/DOI) [2024-03-25]. doi:10.1007/s43681-021-00043-6.

[112] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[113] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1058–1066. PMLR, May 2013. ISSN: 1938-7228. URL: https://proceedings.mlr.press/v28/wan13.html.

[114] Hilde J. P. Weerts, Andreas C. Mueller, and Joaquin Vanschoren. Importance of Tuning Hyperparameters of Machine Learning Algorithms. *Importance of Tuning Hyperparameters of Machine Learning Algorithms*, 20, 2020. URL: https://research.tue.nl/en/publications/importance-of-tuning-hyperparameters-of-machine-learning-algorith.

[115] Paul J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin, editors, *System Modeling and Optimization*, pages 762–770, Berlin, Heidelberg, 1982. Springer. doi:10.1007/BFb0006203.

[116] Daniel Whiteson. HIGGS. UCI Machine Learning Repository, 2014. doi:10.24432/C5V312.

[117] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, September 2017. arXiv:1708.07747 [cs, stat]. URL: http://arxiv.org/abs/1708.07747, doi:10.48550/arXiv.1708.07747.

[118] Lu Yin, Vlado Menkovski, Meng Fang, Tianjin Huang, Yulong Pei, Mykola Pechenizkiy, Decebal Constantin Mocanu, and Shiwei Liu. Superposing Many Tickets into One: A Performance Booster for Sparse Neural Network Training. May 2022. 17 pages, 5 figures, accepted by the 38th Conference on Uncertainty in Artificial Intelligence (UAI). URL: https://research.tue.nl/en/publications/superposing-many-tickets-into-one-a-performance-booster-for-spars.

[119] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks, June 2017. arXiv:1605.07146 [cs]. URL: http://arxiv.org/abs/1605.07146, doi:10.48550/arXiv.1605.07146.

[120] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274,

February 2020. ISSN: 2378-203X. URL: https://ieeexplore.ieee.org/abstract/document/9065428, doi:10.1109/HPCA47549.2020.00030.

# Chapter A

## Reproduction Configurations

Table A.1: Hyperparameter configurations used for each reproduction experiment. Update frequency and batch size values differ between Higgs and Elec2 due to the large difference in training samples. In these cases, values are given as '<Higgs> & <Elec2>'.

| Name | (Fashion) MNIST | Higgs & Elec2 | CIFAR10 & SVHN |
|---|---|---|---|
| Sparsity hyperparameters | | | |
| algorithm | - | - | - |
| sparsity | - | - | - |
| update frequency | 420 | 2563 & 300 | 100 |
| update end | 0.8 | 0.8 | 0.75 |
| sparsity distribution | uniform | uniform | erk |
| Network shape hyperparameters | | | |
| architecture | MLP | MLP | ResNet |
| MLP layers | 3 | 3 | - |
| size first MLP layer | 1000 | 1000 | - |
| size last MLP layer | 1000 | 1000 | - |
| size middle MLP layer | 1000 | 1000 | - |
| size conv block 1 | - | - | 3 |
| size conv block 2 | - | - | 3 |
| size conv block 3 | - | - | 3 |
| Training hyperparameters | | | |
| initial learning rate | 0.01 | 0.01 | 0.1 |
| learning rate scheduler | constant | constant | cosine |
| momentum | 0.9 | 0.9 | 0.9 |
| weight decay | 0.0002 | 0.0002 | 0.0001 |
| batch size | 128 | 4096 & 128 | 256 |
| epochs | 200 | 200 | 135 |
| use dropout | true | true | false |
| dropout | 0.3 | 0.3 | - |
| use label smoothing | false | false | true |
| label smoothing | - | - | 0.1 |

# Chapter B

## Optimization Results in Different Perspectives

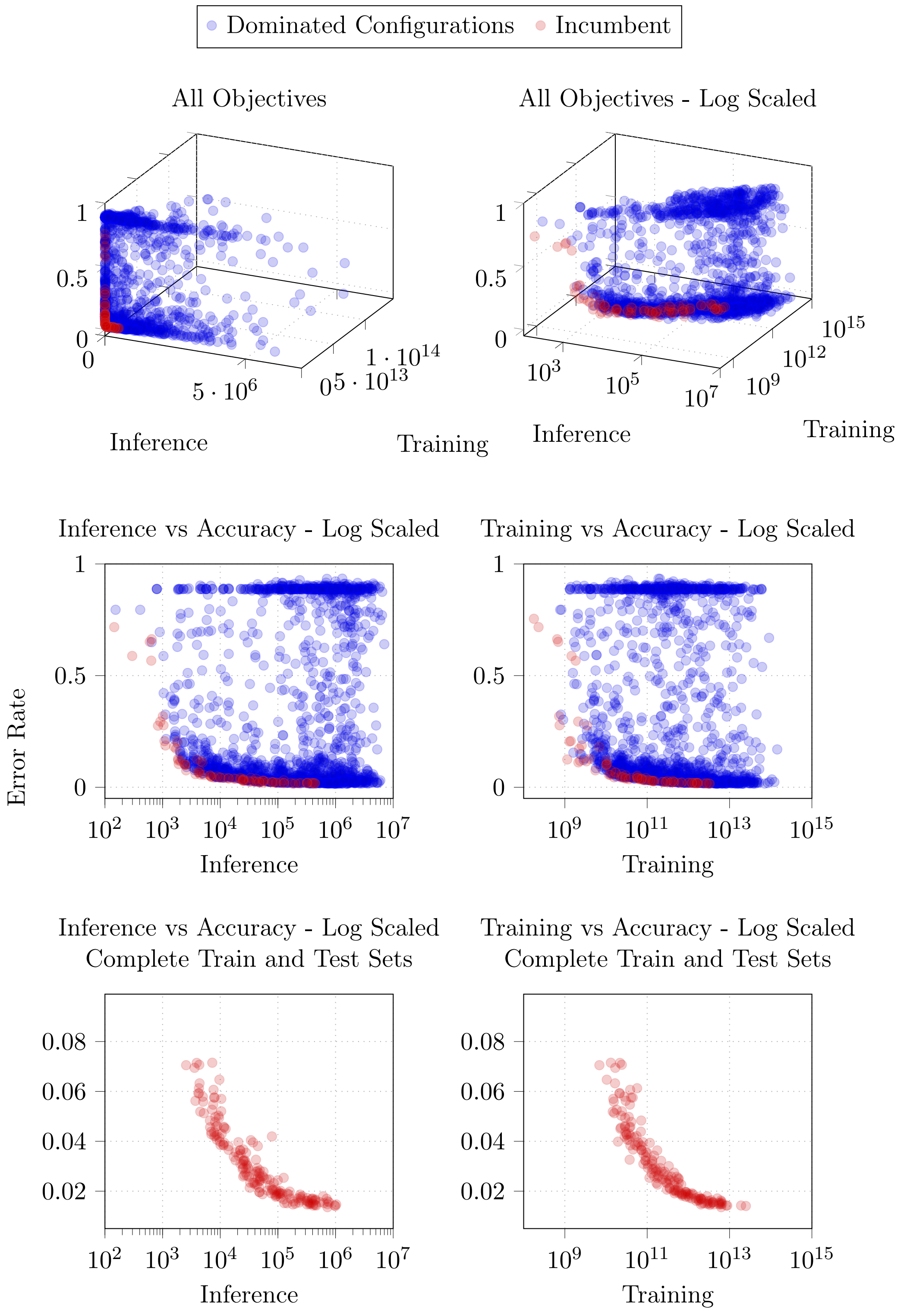


FIGURE B.1: Results of the optimization experiment on MNIST, plotted in several dimensions. The first four plots show the aggregated results of MO-SMAC, where accuracy is averaged over all validation folds on which a configuration was evaluated. The last two plots show the final trade-off after re-evaluating the first two non-dominated layers on the complete train and test set and filtering on the best performing configurations.

# Chapter C

## Hyperparameter Importance Random Search Experiment Plots

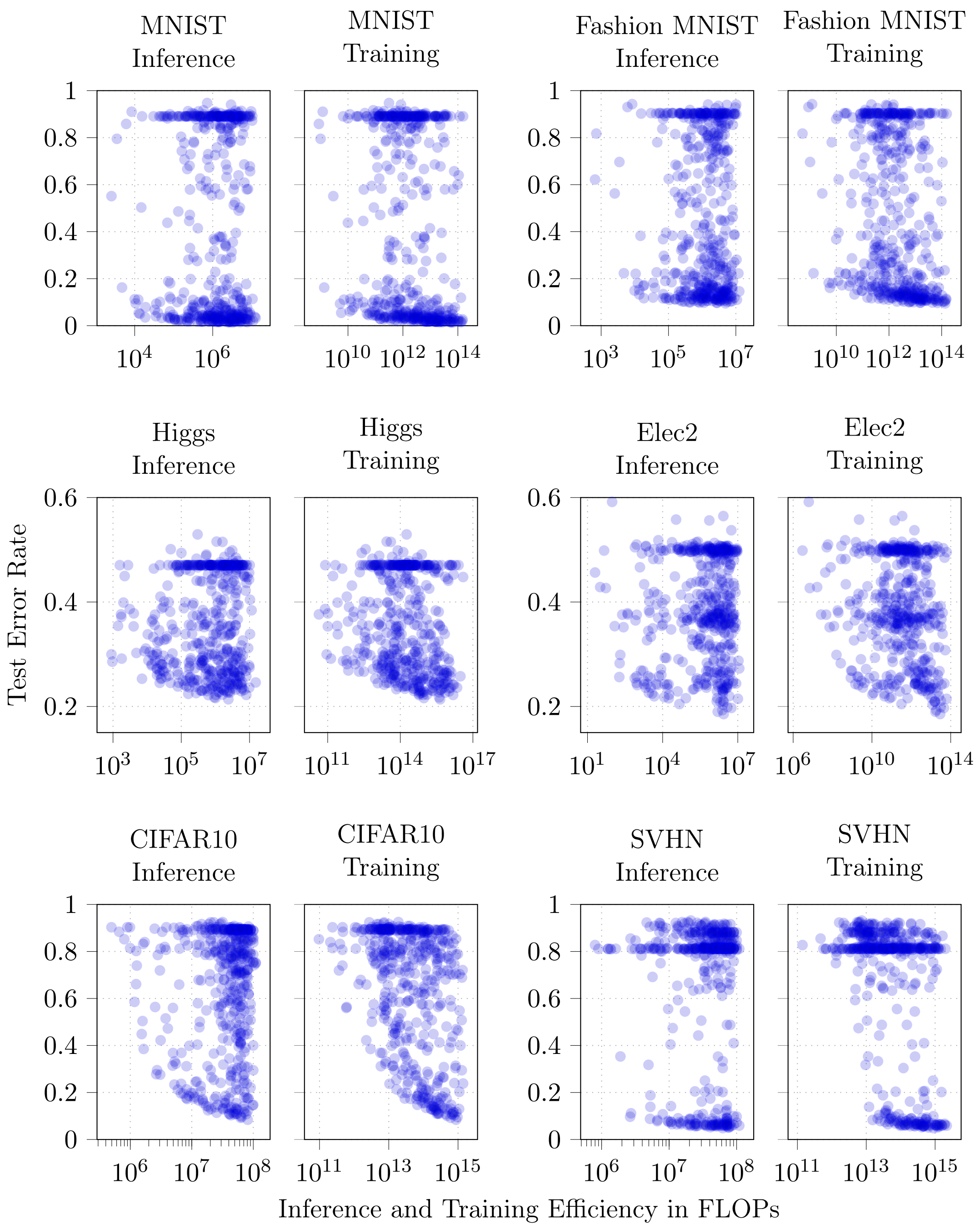


Figure C.1: Plots of all random search experiments created for the MO-fANOVA hyperparameter importance analysis. Results are not filtered on error rate. Efficiency axes are logarithmically scaled.

# Chapter D

## Single-Objective Hyperparameter Importance of Efficiency Objectives

Table D.1: Relative hyperparameter importance for the inference efficiency for all configurable hyperparameters. The three greatest hyperparameter importances for each dataset are marked in **bold**.

| Hyperparameter | MNIST | Fashion MNIST | Higgs | Elec2 | CIFAR10 | SVHN |
|---|---|---|---|---|---|---|
| algorithm | 0.000 | 0.007 | 0.000 | 0.003 | 0.000 | 0.000 |
| sparsity | **0.476** | **0.474** | **0.352** | **0.255** | **0.752** | **0.793** |
| sparsity distribution | 0.000 | 0.000 | 0.000 | 0.000 | **0.025** | **0.030** |
| update end | 0.001 | 0.000 | 0.000 | 0.003 | **0.021** | 0.006 |
| initial learning rate | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| momentum | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| weight decay | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| epochs | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| MLP layers | **0.097** | **0.101** | **0.231** | **0.279** | - | - |
| size first MLP layer | 0.085 | 0.081 | 0.017 | 0.021 | - | - |
| size middle MLP layer | **0.092** | **0.083** | **0.149** | **0.154** | - | - |
| size last MLP layer | 0.014 | 0.014 | 0.028 | 0.042 | - | - |
| size conv block 1 | - | - | - | - | **0.018** | **0.019** |
| size conv block 2 | - | - | - | - | 0.011 | 0.008 |
| size conv block 3 | - | - | - | - | 0.017 | 0.016 |

Table D.2: Relative hyperparameter importance for the training efficiency for all configurable hyperparameters. The three greatest hyperparameter importances for each dataset are marked in **bold**.

| Hyperparameter | MNIST | Fashion MNIST | Higgs | Elec2 | CIFAR10 | SVHN |
|---|---|---|---|---|---|---|
| algorithm | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| sparsity | **0.276** | **0.174** | **0.322** | **0.191** | **0.383** | **0.403** |
| sparsity distribution | 0.000 | 0.000 | 0.000 | 0.000 | 0.008 | 0.001 |
| update end | 0.000 | 0.001 | 0.000 | 0.000 | 0.001 | **0.003** |
| initial learning rate | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| momentum | 0.001 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| weight decay | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| epochs | **0.306** | **0.268** | **0.297** | **0.218** | **0.464** | **0.457** |
| MLP layers | **0.051** | **0.149** | 0.036 | **0.183** | - | - |
| size first MLP layer | 0.043 | 0.008 | 0.037 | 0.006 | - | - |
| size middle MLP layer | 0.038 | 0.110 | **0.049** | 0.109 | - | - |
| size last MLP layer | 0.005 | 0.015 | 0.003 | 0.017 | - | - |
| size conv block 1 | - | - | - | - | **0.018** | 0.002 |
| size conv block 2 | - | - | - | - | 0.011 | 0.000 |
| size conv block 3 | - | - | - | - | 0.017 | 0.002 |