Improving Population-Based Training for Neural Networks

Master's Thesis in Computer Science

Thomas Angeland

August 3, 2020
Halden, Norway

Abstract

In recent years, there has been a rise in complex and computationally expensive machine learning systems with many hyperparameters, such as deep convolutional neural networks. Historically, hyperparameters have provided humans a level of control over the learning process and the performance of the model, and it is well known that different problems require different hyperparameter configurations in order to obtain a good model. However, finding good configurations can be quite challenging, and complexity arises when more hyperparameters are introduced. In addition, research has shown that some tasks may benefit from certain hyperparameter schedules, suggesting that there exists an optimal hyperparameter configuration for every training step.

The issue has resulted in a great deal of research on automated hyperparameter optimization, but there is still a lack of purpose-built methods for generating hyperparameter schedules as they are much harder to estimate. Evolutionary approaches such as Population-Based Training (PBT) has shown great success in estimating good hyperparameter schedules, successfully demonstrating that it is capable of training the network and optimizing the hyperparameters at the same time. However, the method relies on simple selection and perturbation techniques when exploring new hyperparameters, and does not consider other, more advanced optimization methods.

In this thesis, we set out to improve upon the PBT method by incorporating heuristics from the powerful, versatile and evolutionary optimizer called Differential Evolution (DE). As a result, three procedures are proposed, named PBT-DE, PBT-SHADE and PBT-LSHADE. Of the proposed procedures, PBT-DE incorporates the initially proposed heuristics, while PBT-SHADE and PBT-LSHADE explores more recent and adaptive extensions based of SHADE and LSHADE. In addition, a purpose-built distributed queuing system is used for processing members per generation, and allows for asynchronous parallel training and adaption of members.

In order to assess the predictive performance, PBT and the proposed procedures were compared on the MNIST and Fashion-MNIST datasets using the MLP and LeNet-5 network architectures. The empirical results demonstrate that there is a statistical significant difference between PBT and the proposed procedures on the F1 score. Visual and statistical analysis suggest that each proposed procedure outperform PBT on all tested cases. The result data on the Fashion-MNIST dataset indicate a 0.748% and 0.384% gain on average accuracy with MLP and LeNet-5, respectively, when trained with PBT-LSHADE. On the MNIST dataset, result data shows that PBT-SHADE improved average accuracy with 0.21% using MLP, and PBT-DE improved average accuracy with 0.068% with LeNet-5. Furthermore, additional testing suggest that PBT-SHADE scales better with larger population sizes when compared to PBT.

Acknowledgments

I would like to offer my special thanks to my supervisor, Marius Geitle, for his commitment and professional guidance through each stage of this thesis. His willingness to give his time so generously has been very much appreciated.

I would also like to extend my thanks to the technicians of the Østfold University College IT department for their help in offering me the resources for running tests.

Finally, I wish to thank my family for their support and encouragement throughout my study. Especially Nathalie Hognås Hansen, for her love, dedication and patience.

Contents

List of Figures

List of Tables

List of Acronyms

BO-GP Bayesian optimization method with Gaussian Processes.

CCE Categorical Cross Entropy.

CMA-ES Covariance Matrix Adaptation Evolutionary Strategy.

CNN Convolutional Neural Network.

DE Differential Evolution.

L-SHADE SHADE with Linear Population Size Reduction.

MAE Mean Absolute Error.

MLP Multilayer Perceptron.

MSE Mean Square Error.

PBT Population-Based Training.

PBT-LSHADE Population-Based Training with LSHADE.

PBT-SHADE Population-Based Training with SHADE.

PBT-DE Population-Based Training with Differential Evolution.

PSO Particle Swarm Optimization.

ReLU Rectified Linear Unit.

RFA Random Fitness Approximation.

SGD Stochastic Gradient Descent.

SHADE Success-History Based Parameter Adaptation for Differential Evolution.

Chapter 1

Introduction

Hyperparameters are an essential part of almost every machine learning system, providing researchers and practitioners with different ways to *tune* the systems in order to obtain the optimal performance. A large variety of machine learning systems exist, ranging from artificial neural networks [19], kernel methods [165], to ensemble models [33, 21]. Most learners depend on an ambiguous, yet well-defined set of hyperparameters λ in order to operate appropriately. Generally, hyperparameters are represented by a tuple of mixed values in the form of non-fractional numbers such as integers, or fractional numbers such as decimals, to categorical values such as strings, boolean or fixed numbers. It is up to the user to define and assign the hyperparameters to a good state in order to get the most out of the selected learning method. While some hyperparameters may have less impact compared to other hyperparameters, they collectively change various aspects of the learning algorithm, which leads to a varying degree of influence on the resulting model and the performance. This is especially true for recent deep neural networks that depend on various hyperparameter choices about the neural network architecture, regularization, and optimization.

For complex systems such as neural networks, finding the optimal hyperparameter configuration can be quite difficult, as it most often requires multiple test-runs with different sets of hyperparameters. Traditionally, such searches are done manually by rule-of-thumb [72, 76], or by testing different combinations of hyperparameters with a predefined grid [146]). Generally, each hyperparameter combination is produced by a user with some level of tuning knowledge (and with some level of human intuition). Each set of hyperparameters are tested and scored on how well the system performed after training. If the last score is not satisfactory, the tester will produce a new set of hyperparameters, either based on the previous set or in an entirely new direction. This sequential tuning-process repeats until some criteria is met.

While tuning can be simple as described above, such manual methods and semi-automatic methods leave much to be desired. First of, they lack in terms of reproducibility, as they rely on the rule-of-thumb approach. Secondly, while these methods are practical with smaller sets of hyperparameters, they fall short for learning approaches that provide larger sets of hyperparameters [32] and/or training processes which benefit from hyperparameter schedules [82, 158, 5, 30, 23]. Thirdly, manual and semi-automatic approaches require human effort, which combined with tedious, repetitive and exhaustive tuning sessions (especially with computational expensive learning methods), can affect the final performance of the model. In addition, some hyperparameters are depended on each other, and it is possible that changing on hyperparameter might require changing other hyperparameters as well [158, 5]. Lastly, for complex systems, the training and evaluation of models will often require considerable amount of computational resources in order to complete within an appropriate time period. With a strict time-budget, the process may be cut short before the optimal hyperparameter configuration is found.

In order to address these issues, automated hyperparameter optimization receives increasing amounts of attention in machine learning [46]. Hyperparameter optimization has already been shown to outperform manual approaches performed by experts on multiple problems [18, 17]. When tailored to the problem at hand, hyperparameter optimization improves the performance of machine learning algorithms, which has led to new state-of-the-art performances for several machine learning benchmarks [131, 172]. Compared to manual search, hyperparameter optimization also improves the reproducibility and fairness of scientific studies. It provides fair comparisons between various learning approaches, as they all undergo the same tuning process [16, 166].

Research on the challenges of hyperparameter optimization have been conducted for some time [96, 101, 133, 159]. In the beginning, it was made clear that different datasets require different hyperparameters configurations [101]. Later it was found out that general-purpose pipelines can be adapted to specific application domains with hyperparameter optimization [49]. In recent times, it is clearly known that tuned hyperparameters yields better model performance compared to the default setting that is provided by many of the current machine learning libraries [129, 144, 162, 191]. While several approaches to hyperparameter optimization was introduced since its introduction [15, 140, 203], manual search and grid search still prevailed as the best method for hyperparameter tuning for many years for several reasons [141, 98, 149, 204]. Inevitably, this would all change later with the release of several powerful hyperparameter optimization algorithms [17, 172, 116, 125, 145, 123, 84, 61].

While some methods explore the hyperparameter search space in with a pre-determined grid or with random points [146, 17], others guided by exploiting the gradient directly [26, 109], or evolutionary approaches that uses natural selection theory to define a population of points in the search space that mutates over time [119, 130, 123, 125], they all return a single, fixed hyperparameter configuration that achieves the best performance. While this is the general goal of hyperparameter optimization, recent studies have looked into ways to adapt the hyperparameters while training in order to obtain the best model directly [84, 47].

1.1 Research Question and Method

In this study, we will attempt to improve upon Population-Based Training (PBT) [84], an evolutionary hyperparameter adaption procedure for neural networks. PBT has demonstrates competitive performance in several benchmarks when compared to random search. Because the method uses simple, random perturbation to explore new hyperparameters, we wonder whether there are ways to improve the exploration method with a well-established and well-researched metaheuristic called Differential Evolution (DE) [179]. Interestingly, DE has already shown some success for other evolutionary approaches [208] for training neural networks, reporting that hybridization of different evolutionary technologies may improve their search capability in network parameters learning. More recently, variants of DE have been shown to perform well for hyperparameter tuning as well [61].

1.1.1 Research Question

The research question of this study consists of one main question with two sub-questions; First, we need to find appropriate ways to apply DE heuristics to the PBT procedure:

RQ 1 *In what way can differential evolution heuristics be incorporated into population-based training for neural networks?*

Secondly, we need to compare the proposed PBT procedure(s) to the original PBT procedure:

RQ 1.1 *In what way will the differential evolution heuristics affect the performance of population-based training for neural networks?*

Lastly, the number of members in the population is one of the most important parameters for population-based training in regards to performance and algorithmic time complexity. Therefore, there is a compelling reason for testing different populations sizes:

RQ 1.2 *In what way will the number of members in the population affect the performance of population-based training with differential evolution?*

1.1.2 Method

In order to answer the research questions, we set out to obtain a broad understanding of the research field of hyperparameter optimization for machine learning. Therefore, important, novel approaches for hyperparameter optimization has been researched in order to establishing the current research field. Moreover, the original PBT [84] procedure is presented in detail in order to obtain a clear understanding of how the procedure operate, and which parts of the procedure that should be updated or replaced. In addition, some of the few recent PBT extensions will also be included. In a similar manner, the initial DE procedure will also be covered in detail, including how the algorithm works, how it performs and recent advancements made to the original algorithm. The literature review allows us to establish current trends and practices, as well as differences between methods in characteristics, strengths and weaknesses.

The literature review lays the foundation for developing new procedures based of PBT that incorporates heuristics from DE which answers RQ1. To obtain result data, developed procedures and the PBT procedure are tested on image classification with the MNIST [110] and Fashion-MNIST [207] datasets using the Multilayer Perceptron (MLP) and LeNet-5 network architectures in order to answer RQ1.1 and RQ1.2. The characteristics of each procedure relevant for analysis is the predictive performance, predictive performance over time, generalization error, time complexity and performance across different population sizes.

Performance is measured by three individual metrics, but all statistical analysis are based of result data by one determined metric: the F1 Score [29]. Because multiple procedures are proposed, statistical analysis is first and foremost conducted using the Welch ANOVA [205] test method followed up with a Pairwise Games-Howell [59] Post-Hoc analysis. This type of test is used in order to determine whether there are a statistical significant difference between the procedures on the result data, and if so, where the difference exist. When that is done, descriptive statistics and visual analysis is used in order to describe the difference between the numerical data obtained.

1.2 Outline

This section gives a short summary of the remaining chapters of this thesis.

Chapter 2 formulates the hyper-parameter optimization problem and explains some of the important findings known about the problem. In addition, a large part of this chapter consists of background information about a range of different model-free and model-based algorithms for hyperparameter optimization that precedes the PBT procedure. After that, general and mathematical descriptions is formulated for training and adapting neural networks with hyper-parameter optimization. Then, the chapter provides a description of PBT, including extensions that were available at the time. Lastly, a large portion of the chapter is reserved for giving a detailed description of DE, as well as significant advances proposed more recently.

Chapter 3 presents the proposed procedures and as well as information about testing setups for obtaining the experimental results. This include the implementation details regarding the PBT procedure, as well as the choice of procedure parameters and which key parts of the procedure that are included. Furthermore, three different PBT-based procedures with DE heuristics are proposed, where each procedure and their operations are explained in detail. Later, information about the experimental test setup is given, describing the various datasets and neural network architectures selected. This include the technique used for sampling the datasets into set divisions, as well as details about the implementation of the network architectures. Moreover, we define and describe the types of hyperparameters that will undergo optimization, and why certain hyperparameters were left out. At the end of the chapter, information about the technical aspects of running the procedures is given, including the overall flow of the system, programmatic approaches, as well as details about the computer environment, programming language and publicly available packages used.

Chapter 4 consists of the experimental results obtained from running the PBT procedure and the procedures proposed in this study. Here, the statistical significance is determined by running the Welch ANOVA [205] and Games-Howell [59] Post-Hoc analysis which are performed on the result data. In addition, multiple figures and tables are presented, describing the predictive performance of the algorithms, both final and over time. Later, some information is given regarding the time complexity of the procedures. Furthermore, the predictive performance results from running tests with different population sizes is presented and analyzed. Lastly, the chapter is ended with a visual analysis of the hyperparameter schedules generated from the procedures, focusing on the overall trends.

Chapter 5 includes a concise summary of the principal implications of the findings. First, the research questions are reiterated and answered based of the findings. After that, the results are put in context with the state-of-the-art results. To end the chapter, we acknowledge limitations and challenges and propose recommendations for future research.

Chapter 6 concludes this thesis by summarizing the main contributions of the work and ends on recommendations for future research.

Chapter 2

Background

This chapter consists of five sections. First of all, Section 2.1 give details about the general objective of hyperparameter optimization, including known challenges and the mathematical notations that will be used to describe the different methods mentioned throughout this chapter. Section 2.2 provides insight into several well-established methods of hyperparameter optimization, describing how they work and their characteristics. In Section 2.3, neural network training and hyperparameter adaption are explained, covering the major operations that is performed in order to train and optimize neural networks. In Section 2.4, PBT is described in more detail, including recent related work that sets out to improve upon the original procedure. Finally, Section 2.5 covers essential DE heuristics, as well as several recent state-of-the-art adaptive DE extensions.

2.1 Research Topic

In simple terms, the general goal of machine learning applications is to optimize the trainable parameters θ of a model $\mathcal{M}$ to minimize some predefined loss function $\mathcal{L}\left(\mathcal{X}^{(test)}; \mathcal{M}\right)$ on the given test data $\mathcal{X}^{(test)}$. The model $\mathcal{M}$ is constructed by a learning algorithm $\mathcal{A}$ using a training set $\mathcal{X}^{(train)}$, typically involving solving some convex optimization problem over the *parameter space* Θ. The goal of hyperparameter search is to find a set of $\mathcal{D}$ hyperparameters λ^* (not to be confused with θ) that yield an optimal model $\mathcal{M}^*$ which minimizes $\mathcal{L}\left(\mathcal{X}^{(test)}; \mathcal{M}\right)$ [31]. This is considered a non-differentiable, single-objective optimization problem for a constrained *hyperparameter configuration space*, or *search space*, $\Lambda = \Lambda_1 \times \Lambda_2 \times \ldots \Lambda_{\mathcal{D}}$ of mixed data types. The domain of the *n-th* hyperparameter is denoted by Λ_n. A hyperparameter configuration is denoted by the vector $\lambda \in \Lambda$, which becomes $\mathcal{A}_\lambda$ when the hyperparameters of $\mathcal{A}$ is instantiated to λ. The problem can be formalized as follows:

$$\begin{aligned} \lambda^* \approx \quad & \underset{\lambda \in \Lambda}{argmin} \mathcal{L}\left(\mathcal{A}_\lambda(\mathcal{X}^{(train)}; \mathcal{X}^{(test)})\right) \\ \approx \quad & \underset{\lambda \in \Lambda}{argmin} \mathcal{F}\left(\mathcal{A}_\lambda, \mathcal{L}, \mathcal{X}^{(train)}, \mathcal{X}^{(test)}\right) \end{aligned} \tag{2.1}$$

As shown in Equation 2.1, the objective function $\mathcal{F}$ is the general function to be optimized. It is dependent on a tuple of hyperparameters λ, as well as the machine learning algorithm $\mathcal{A}$, the chosen loss function $\mathcal{L}$, and the dataset $\mathcal{X}$. For supervised learning, $\mathcal{X}$ is commonly split into a training set $\mathcal{X}^{(train)}$ and a testing set $\mathcal{X}^{(test)}$ using hold-out or cross-validation methods [100, 45].

One of the major operations in the objective function $\mathcal{F}$ consists of training a model $\mathcal{M}$ through navigating Θ by computing the loss on $\mathcal{X}^{(train)}$ using $\mathcal{L}$. Simply put, the loss function $\mathcal{L}$ purpose is to guide the learning by applying a cost that encourage good predictions and discourage bad predictions (training is covered in more detail in Section 2.3). There have been proposed various methods for calculating the loss, like the Mean Square Error (MSE), Mean Absolute Error (MAE), cross-entropy [41], as well as problem-specific ones such as loss functions for class imbalance [120]. Many of the commonly used machine learning packages also include multiple loss functions as a categorical hyperparameter [146, 154, 1].

After training, $\mathcal{M}$ is validated on $\mathcal{X}^{(test)}$ by computing $\mathcal{L}$ or using an entirely different metric to describe the predictive performance. When performing hyperparameter optimization, the model $\mathcal{M}$ is typically validated using a portion of the training set to form a validation set $\mathcal{X}^{(valid)}$, and final predictive performance is reported using the test set $\mathcal{X}^{(test)}$. This is done to minimize the generalization error so that the hyperparameter optimization algorithm does not return λ^* that is *overfitted* on the test set.

The time it takes to perform hyperparameter optimization by navigating Λ in order to find the optimal set of hyperparameters λ^* depends heavily on the available computational resources, as well as the selected learning algorithm $\mathcal{A}$ and the size of the dataset $\mathcal{X}$. Certain types of hyperparameters may also have a great influence on the evaluation time; for example, the size of ensembles [21, 33], changes to the neural network architecture [19] or even regularization and kernel complexity for support vector machines [20]. With this in mind, hyperparameter evaluations suddenly become an computationally expensive issue that can take up to multiple days or even weeks [184, 42, 104].

Hyperparameters are most often defined as continuous values such as floating point values, often related to regularization. They may also be discrete values such as integers, commonly used to define the neural network architecture [19], parameterization of kernels in kernel methods [165], or size of ensembles [33, 17]. In addition, some learning algorithms also accept categorical values with no specific order, such as strings. These may be used to set different flags that enable or disable different operations in the learning algorithm [146]. This makes hyperparameter optimization a mixed-type problem, and special care must be taken for optimization algorithms that perform arithmetic operations directly with hyperparameters [91, 180, 17].

While some machine learning algorithms provide only a few hyperparameters [4, 35, 146] for optimization (effectively having a search space Λ of low dimensionality), other complex learners might supply hundreds of hyperparameters [16]. The number of hyperparameters may be extended even more if various methods of pre-processing or regularization (e.g. data augmentation, weight decay) are also subjected to optimization [77]. When that has been said, for many cases it has been empirically demonstrated that only a few hyperparameters are needed in order to impact model performance [17], but that requires the difficult task of determining which hyperparameters to prioritize.

2.2 Related Work

In machine learning, a wide variety of optimization methods have been proposed for selecting optimal hyperparameters, including model-free approaches [146, 17, 86, 116] and

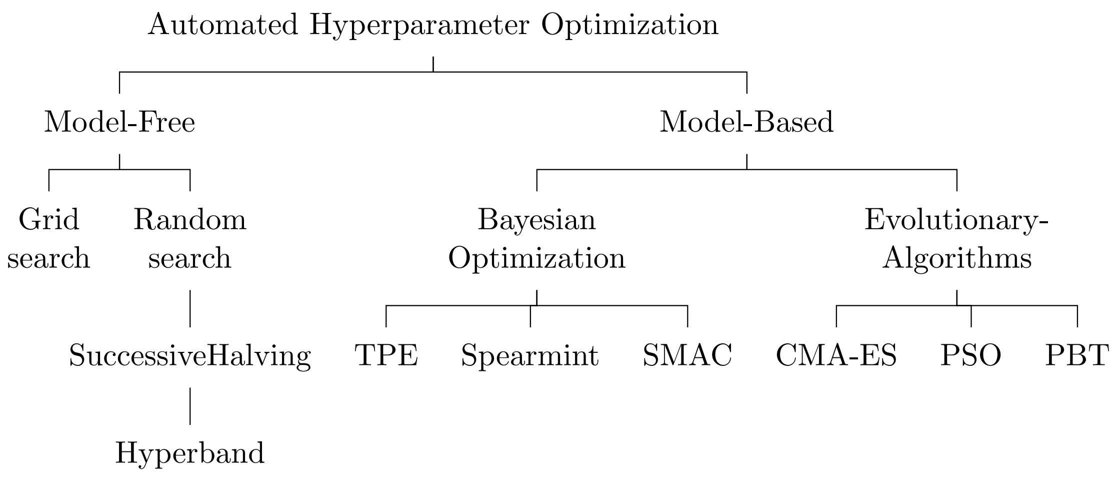


Figure 2.1: Approaches for automated hyperparameter optimization [123].

model-based approaches such as Bayesian optimization techniques [88, 10, 18, 46, 79, 172] and evolutionary methods [119, 130, 123, 125, 84]. This section will provide details about some of the fundamental hyperparameter optimization approaches for machine learning that have been suggested over the recent years. Figure 2.1 displays an overview of these approaches, split into *model-free* and *model-based* methods [123]. Through careful exploration and exploitation, model-based methods create a surrogate model of the search space Λ with the information obtained during optimization. Alternatively, model-free methods refers to algorithms that do not exploit knowledge about the search space Λ.

2.2.1 Grid Search

Grid search [146] is one of the simplest ways to perform hyperparameter optimization and is often used in combination with manual search [73, 108, 111]. The search space Λ is partitioned into a Cartesian grid, indexed by K configuration vectors λ_k, which contain a range of values for each particular hyperparameter. These values can be specified manually by a user, or they can be generated using approaches such as through logarithmic scale steps [17]. In order to perform grid search, every joint specification of hyperparameters λ_S are created by assembling every possible combination of values from each vector in Λ. The model $\mathcal{M}$ is then trained with the learning algorithm $\mathcal{A}$ for $S = \prod_{k=1}^{K} |L^{(k)}|$ trials (one for each set of combined hyperparameters λ) and evaluated using the objective function $\mathcal{F}$. While grid search works fine for smaller sets of hyperparameters, it evidently suffers from the *curse of dimensionality*; the number of trials grows exponentially with the number of hyperparameters at a rate of $O(nk)$ [14], assuming that n is the same for all hyperparameters. The time complexity issue can be mitigated with the use of parallel computing, but compared to more recent and efficient methods, grid search stands now as a slower, more brute-force way of approaching the problem of hyperparameter optimization.

2.2.2 Random Search

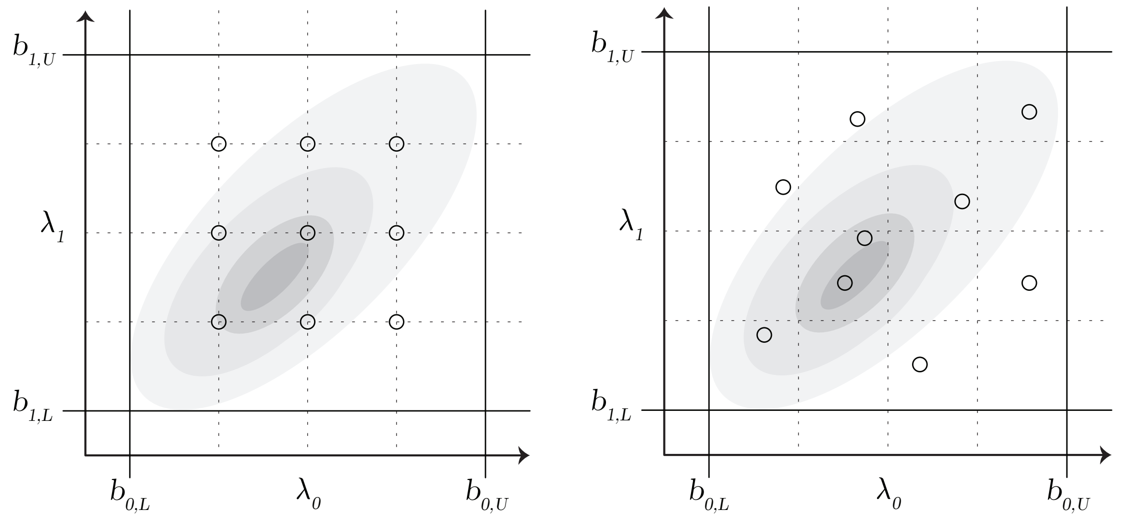
Random search [17] is another widely used hyperparameter optimization algorithm for machine learning, known by its easy implementation and public availability in machine learning packages [146]. The method shares similarities with grid search, but instead of discretizing the search space with a Cartesian grid, it randomly samples it, which improves localization of good regions for sensitive hyperparameters. At each iteration, λ_i is sampled from a uniform distribution $\mathcal{U}(\mathbb{u}_l, \mathbb{u}_u)$ bound by the lower and upper limits, $\mathbb{u}_l$ and $\mathbb{u}_u$, of the different hyperparameter types in the search space $\Lambda_{\mathcal{D}}$. These bounds are defined by the user as a vector of $\mathcal{D}$ dimensionality. If the sampled hyperparameter configuration λ_i receives a higher objective value from the objective function $f(\cdot)$, it is considered the new best evaluation λ^*. Random search is briefly summarized in Algorithm 1.

Algorithm 1 Random Search Optimization

```
1: function RandomSearch(max_iter, $\mathbb{u}_l, \mathbb{u}_u$))
2:     for $i \in \{0, 1, \ldots, i_{max} - 1\}$ do
3:         $\lambda_i \sim \mathcal{U}^{\mathcal{D}}(\mathbb{u}_l, \mathbb{u}_u)$
4:         if $f(\lambda_i) > f(\lambda^*)$ or $i = 0$ then
5:             $\lambda^* = \lambda_i$
6:         end if
7:     end for
8:     return $\lambda^*$
9: end function
```

As demonstrated in Figure 2.2, random search may be more precise in finding the optimum for each hyperparameter λ_k compared to grid search. The method has also shown to be very effective for non-cubic search spaces, i.e. if different ranges of variation are given to some of the hyperparameters. There are other reasons for why random search works well, e.g. its easy implementation or the fact that it can be computed asynchronously in parallel, but it still shares some of the drawbacks of manual search and grid search. Random search does not use the information gained by previous tries. which make every hyperparameter configuration a blind guess in the search space. As a consequence, finding the local minimum with a high degree of precision require more samples of the search space, which then require more resources and time to compute. While this may not be a problem for non-complex systems with small models, fast learning algorithms and few hyperparameters, the required time and resources drastically increase for complex systems with large models and many hyperparameters.

There have been proposed methods for improving random search, such as Successive-Halving [86]. Given the same set of hyperparameter configurations, SuccessiveHalving achieves more efficient allocation of resources by improving the division and selection of randomly generated hyperparameter configurations without increasing the amount of assumptions about the nature of the hyperparameter configurations space Λ. The algorithm consists of four steps: First, it uniformly allocate a budget to a set of hyperparameter configurations; secondly, it evaluates the predictive performance of all currently remaining configurations; then, it throws out the bottom half of the worst scoring configurations, and; repeat step two until one configuration remains. This is clarified in algorithm 2.



(a) grid search (b) random search

Figure 2.2: A comparison of search space coverage between grid search and random search with the same number of hyperparameter configuration samples.

Algorithm 2 SuccessiveHalving [86]

```
 1: require: $n$ sets of hyperparameter configurations $\lambda$, minimum score $r$, maximum
    resource $R$, reduction factor $\eta$, minimum early-stopping rate $s$
 2: function SuccessiveHalving($n, \lambda, r, R, \eta, s$)
 3:     $s_{max} = \lfloor \log_\eta (R/r) \rfloor$
 4:     assert $n \geq \eta^{s_{max}-s}$        ▷ ensure at least one configuration will be allocated $R$
 5:     $\lambda^* = \lambda$
 6:     for $i \in \{0, \ldots, s_{max} - s\}$ do
 7:         $n_i = \lfloor n\eta^{-i} \rfloor$
 8:         $r_i = r\eta^{i+s}$
 9:         $\mathbb{L} = \text{run\_then\_return\_loss\_values}\,(f(\lambda), r_i) : \lambda_i \in \lambda^*$
10:         $\lambda^* = \text{return\_top\_configurations}\,(\lambda^*, \mathbb{L}, n_i/\eta)$
11:     end for
12:     return $\lambda^*$
13: end function
```

While the algorithm allocates resources exponentially to well performing hyperparameter configurations, it unfortunately requires a number of hyperparameter configurations n as an input to the algorithm. For a given task with a finite time-budget B, B/n resources are distributed evenly to each configurations. However, for a fixed B, it is difficult to say whether we should prioritize many hyperparameter configurations (large n) with short training time (small B/n), or choose less hyperparameter configurations (small n) and longer average training times (large B/n). HyperBand [116] extends the SuccessiveHalving algorithm and addresses this n *versus* B/n problem by testing several distinct values of n for a fixed B. In other words, it essentially performs a grid search for possible values of n, and effectively model the problem of hyperparameter selection as a many-armed bandit. While parallelisation is natively supported, SuccessiveHalving and HyperBand seem impractical for single training optimization processes due to the large amount of computational resources that they require for the initial runs.

2.2.3 Bayesian Optimization

Bayesian optimization is a framework for sequential optimization of unknown objective functions, where it tries to model the conditional probability $p(y|\lambda)$ of the predictive performance (given by the validation metric, y) of a given hyperparameter configuration λ. Bayesian optimization method with Gaussian Processes (BO-GP) [172, 147] are common algorithms for Bayesian optimization. The goal is to describe the objective function $\mathcal{F}$ that is to be optimized by constructing a posterior distribution of functions (gaussian process). The posterior distribution gets better with more observations, and the algorithm get more certain of which regions in the hyperparameter search space that yield better results, while balancing the need for exploration and exploitation. For each iteration, the gaussian process learns from the points previously explored. The next point is determined based on the posterior distribution and an exploration strategy (e.g. Upper Confidence Bound (UCB) [174]). The method an be summarized in five steps: First, build a surrogate probability model of the objective function $\mathcal{F}$; secondly, determine which hyperparameter configurations that perform best on the surrogate; thirdly, apply these hyperparameters to the true objective function $\mathcal{F}$; fourthly, update the surrogate model with the new results; and lastly, repeat step 2–4 until B iterations or time is reached.

There have been proposed several variations of Bayesian optimization for neural networks. In addition to BO-GP [172, 147] and Gaussian Process Upper Confidence Bound (GP-UCB) [174], examples include Tree-structured Parzen Estimator (TPE) [18], which is a non-standard Bayesian optimization algorithm based on tree-structured Parzen density; Sequential Model-Based Optimization for General Algorithm Configuration (SMAC) [79], which uses random forests to model $p(y|\lambda)$ as a Gaussian distribution; and Spearmint [171, 185, 173], which uses Gaussian processes to model $p(y|\lambda)$ and slice samples the hyperparameters of the gaussian process. There is also a variant that mixes Hyperband and Bayesian optimization [51], as well as a variant that attempts to reduce inefficiency by training on a smaller dataset (while using the full validation dataset to evaluate its predictive performance) [99].

Compared to grid search and random search, Bayesian optimization has been shown [78, 191, 171] to obtain better results with less evaluations, because the algorithm can determine the quality of the distinct sets of hyperparameters before they are tested. However, the efficiency rapidly decreases for cases where the dimension of the search space Λ increases, to the point where it is not better than random search [116]. Another major drawback is that the algorithms are hard to parallelize due to their sequential nature; one trial needs to be completed before the next one can be started, because any trial needs the Gaussian process to be updated and the acquisition function to find its maximum. While some level of parallelization can be achieved by evaluating multiple acquisition function configurations simultaneously [62, 80, 28, 43], or by conditioning decisions on expectations over several hallucinated performance values for currently running trials [172], full parallelization appears to be more difficult to achieve as each step inherently depends on all the information gathered at that point. In addition, Gaussian processes are computationally expensive for large numbers of evaluations. The process requires the covariance matrix to be inverted, which requires n^2 operations, where n is the number of current evaluated configurations. Gaussian processes ultimately need $O(n^3)$ operations, which could lead to Bayesian optimization taking significant amount of time to complete for several thousand samples [157].

2.2.4 Covariance Matrix Adaptation Evolutionary Strategy

Evolutionary algorithms refer to a set of algorithms for population-based optimization inspired by natural selection. In natural selection, it is believed that from a generation of individuals, only the ones with the most beneficial traits gets to pass down their characteristics to the next generation. Evolution happens gradually when each generation becomes better adapted, or more *fit*, to the environment. This theory can be applied to hyperparameter optimization, where the population consists of different configurations of hyperparameters which undergo some level of mutation (e.g. perturbation) and tested on the objective function in order to determine which configurations achieve the best predictive performance.

One evolutionary approach for selecting hyperparameters is that of Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [66, 67, 125]. In order to determining the optimal λ^*, the algorithm iteratively samples a population of hyperparameter solutions from a parametric distribution over the search space Λ. These configurations are then tested and evaluated on the objective function. The evaluations are stored in tuples and forms a dataset which the algorithm uses to update the mean and covariance matrix of the search distribution. More precisely, the objective function $\mathcal{F} : \Re^n \to \Re$ is parametrized by the search space $\lambda \in \Lambda \in \Re^n$ (Equation 2.1).

The approach maintains a multivariate Gaussian distribution over the search space as $\lambda \sim \mathcal{N}(\lambda; \mathbb{m}, \boldsymbol{C})$, where $\mathbb{m}$ is a mean vector of n dimensionality and $\boldsymbol{C}$ is a $n \times n$ covariance matrix. In each generation g, the algorithm produces a population of N configurations from the distribution as $\lambda_i \sim \mathcal{N}(\lambda; \mathbb{m}_g, \boldsymbol{C}_g), i = 0, \ldots, N-1$, where $\mathbb{m}_g$ and $\boldsymbol{C}_g$ denote the mean vector ad covariance matrix of generation g. Then, each new candidate configuration λ is evaluated using $\mathcal{F}(\lambda_i)$ and sorted in an ascending order according to the achieved objective value (best first). From these, the first μ ($< N-1$) best configurations are selected for updating the $\mathbb{m}_g$ and $\boldsymbol{C}_g$. As a last parameter, the global step-size $\sigma \in \Re$, defined as the global standard deviation, is required by the user and controls the convergence rate of the covariance matrix update. The CMA-ES algorithm is summarized in Algorithm 3.

In the algorithm, step 1 and 2 initialize the parameters $\{\mathbb{m}, \boldsymbol{C}, \sigma\}$ which will be iteratively updated by step 3 to 17. Step 6 defines a multivariate normal distribution from which new hyperparameter configuration solutions are sampled in step 7. These new configurations are then evaluated by $\mathcal{F}(\cdot)$ in step 8. The sorted tuple of best candidates is denoted as $\mathbb{y}_{i,\ldots,N-1}$. The weighted (i.e. $w_i = 1/\mu$) sum of the best candidates $\bar{\mathbb{y}}$ is calculated in step 10, and used in step 11 to updated the mean vector $\mathbb{m}$. Through step 12 to 15, the covariance matrix update is calculated by three factors: the old information, the change of mean over time $\mathbb{p}_c$, and the rank-μ update which considers the *good* variations in the last generation. Based on the *conjugate evolution path*, $\mathbb{p}_\sigma$, step 13 updates the step-size control that limits the changes to be applied on the distribution in order to achieve faster convergence to the optimum while avoiding premature convergence. The other parameters consist of the variance effective selection mass μ_w, the learning rates c_1, c_c, c_σ, and the σ dampening factor d_σ.

In contrast to many traditional methods, CMA-ES makes fewer assumptions on the

Algorithm 3 Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [86]

```
 1: Initialize $\mathbb{m} \in \Re^n, \sigma \in \Re^+, N, \mu$
 2: Initialize $\boldsymbol{C} = I, \mathbb{p}_c = 0, \mathbb{p}_\sigma = 0$
 3: procedure CMA-ES($\mathbb{m}, \sigma, C, \mathbb{p}_c, \mathbb{p}_\sigma$)
 4:     while stopping criteria not met do
 5:         for $i \in \{0, 1, \ldots, N-1\}$ do
 6:             $\mathbb{y}_i \sim \mathcal{N}(0, \boldsymbol{C})$
 7:             $\lambda_i = \mathbb{m} + \sigma \times \mathbb{y}_i$                ▷ sample candidate solution
 8:             $f_i = \mathcal{F}(\lambda_i)$                ▷ measure performance (i.e. fitness)
 9:         end for
10:         $\bar{\mathbb{y}} = \sum_{n=0}^{\mu} w_i \mathbb{y}_{i:N-1}$
11:         $\mathbb{m} \leftarrow \mathbb{m} + \sigma\bar{\mathbb{y}}$                ▷ mean update
12:         $\mathbb{p}_\sigma \leftarrow (1 - c_\sigma)\mathbb{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_w}\boldsymbol{C}^{-\frac{1}{2}\bar{\mathbb{y}}}$
13:         $\sigma \leftarrow \sigma \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbb{p}_\sigma\|}{\|\mathcal{N}(0,I)\|}\right)\right)$                ▷ step-size control update
14:         $\mathbb{p}_c \leftarrow (1 - c_c)\mathbb{p}_c + \sqrt{c_c(2 - c_c)\mu_w}\bar{\mathbb{y}}$
15:         $\boldsymbol{C} \leftarrow (1 - c_1 - c_\mu)\boldsymbol{C} + c_1 \mathbb{p}_c \mathbb{p}_c^T + c_\mu \sum_{n=0}^{\mu} w_i \mathbb{y}_{i:N-1} \mathbb{y}_{i:N-1}^T$ ▷ Cov. matrix update
16:     end while
17: end procedure
```

nature of the underlying objective function. The derivative-free algorithm only exploits the ranking between the different hyperparameter configurations in order to learn the sample distribution. In 2013, [126] showed that CMA-ES outperformed more than 100 hyperparameter optimization methods on different black-box functions, concluding that it performs well for larger function evaluation budgets [125].

2.2.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was originally proposed in 1995 [91] for simulating social behavior [92], representing the movement of organisms such as a flock of birds or a school of fish. The philosophical aspects of PSO and swarm intelligence is described in detail in [93]. The algorithm was recently adapted to hyperparameter optimization for machine learning algorithms, such as support vector machines (SVMs) [119] and neural networks [130, 123], as another way of applying evolutionary principles for finding the optimal hyperparameter configuration λ^*. Similarly to CMA-ES, this approach also considers a population of N members, but the members are represented as $\mathcal{D}$-dimensional particles that evolves over each generation by moving towards the best individuals. Each *i-th* particle position represents a hyperparameter configuration vector $\lambda_i \in \Lambda \in \Re^{\mathcal{D}}$, and has a velocity $v_i \in \Re^{\mathcal{D}}$ which influences its movement. Let λ_i^* denote the best known position of the *i-th* particle, and λ^* denotes the best known position across all generations. A summary of the method is described in Algorithm 4.

The swarm initialization is performed in step 3 to step 10. Each *i-th* particle λ_i in the swarm population is initialized by randomly sampling from a uniform distribution $\mathcal{U}(\mathbb{b}_l, \mathbb{b}_u)$ that is bound by lower limit $\mathbb{b}_l$ and upper limit $\mathbb{b}_u$ of the hyperparameters. The new, sampled particle becomes the current best particle λ_i^* in the generation, and has a chance to become the best known particle λ^* across all generations if $f(\lambda_i^*) > (\lambda^*)$. Similarly to the initial particles, the velocity v_i is also drawn from a uniform distribution,

Algorithm 4 Particle Swarm Optimization (PSO) [44, 123]

```
 1: procedure PSO($s, g_{max}, \mathbb{b}_l, \mathbb{b}_u$)
 2:     $f(\lambda^*) \leftarrow -\infty$
 3:     for $i \in \{0, 1, \dots, N-1\}$ do                         ▷ swarm initialization
 4:         $\lambda_i \sim \mathcal{U}(\mathbb{b}_l, \mathbb{b}_u)$
 5:         $\lambda_i^* \leftarrow \lambda$
 6:         if f($\lambda_i^*$) $> f(\lambda^*)$ then
 7:             $\lambda^* \leftarrow \lambda_i^*$
 8:         end if
 9:         $v_i \leftarrow \mathcal{U}(-|\mathbb{b}_l - \mathbb{b}_u|, |\mathbb{b}_l - \mathbb{b}_u|)$
10:     end for
11:     $g \leftarrow 1$
12:     while $g \leq g_{max}$ do                                      ▷ swarm evolution
13:         for $i \in \{0, 1, \dots, N-1\}$ do
14:             $\mathbb{r}_p, \mathbb{r}_g \sim \mathcal{U}(0, 1)$
15:             $v_i \leftarrow \omega v_i + \phi_p \mathbb{r}_p(\lambda_i^* - \lambda_i) + \phi_g \mathbb{r}_g(\lambda^* - \lambda_i)$
16:             $\lambda_i \leftarrow \lambda_i + v_i$
17:             if $f(\lambda_i^*) > f(\lambda^*)$ then
18:                 $\lambda^* \leftarrow \lambda_i^*$
19:                 if $\|\lambda^* - \lambda^*_{\text{prev}}\| < \delta$ then
20:                     return $\lambda^*$
21:                 end if
22:                 if $f(\lambda^*) - f(\lambda^*_{\text{prev}}) < \epsilon$ then
23:                     return $\lambda^*$
24:                 end if
25:             end if
26:             $g \leftarrow g + 1$
27:             $\lambda^*_{\text{prev}} \leftarrow \lambda^*$
28:         end for
29:     end while
30: end procedure
```

bounded by the upper and lower bounds of the hyperparameter limits in both positive and negative direction.

The swarm evolution is described in step 13 to step 28. Each swarm generation g, where g_{max} represents the maximum number of generations, the particles and the velocity values are updated by the following formula (step 15):

$$\upsilon_i \leftarrow \omega\upsilon_i + \phi_p \mathbb{r}_p(\lambda_i^* - \lambda_i) + \phi_g \mathbb{r}_g(\lambda^* - \lambda_i). \tag{2.2}$$

In order to increase search diversity, $\mathbb{r}_p$ and $\mathbb{r}_g$ are added as a stochastic component to the velocity updates, where each represents a random uniform number between 0 and 1, drawn from $\mathcal{U}(0, 1)$. To add resistance to velocity changes, ω is used as an inertia weight that scales the velocity. Velocity acceleration and deceleration is achieved with the acceleration coefficient ϕ_p and ϕ_g. These factors affect the level of influence λ_i^* and λ^* have on changes in velocity. In step 16, the new velocity is used to update particle λ_i. From here, the best particle position is modified for each particle (step 18), and the best swarm position is updated only if it is outperformed by one of the new particles. The evolution terminates when the best position λ^* is outperformed by less than the minimum step-size δ (step 19), or if the current best particle in the generation improves by less than the threshold ϵ (step 22), or if the maximum number of generations g_{max} has been met (step 12).

PSO has shown promising results [123] in terms of time-complexity, achieving the same accuracy performance as grid search and random search on the CIFAR-10 dataset [103], but considerably faster wall-clock time. The algorithm has also shown capable of optimizing a deep neural network model to a performance that outperforms the same model tuned by a human expert [124]. In terms of scalability, the algorithm scales linearly with the number of hyperparameters, i.e. the dimensionality $\mathcal{D}$, which makes it excellent for machine learning algorithms whose model performance depends heavily on a large number of hyperparameters. In addition, the algorithm requires only a few parameters itself and can be easily parallelized [124].

2.3 Automatic Hyperparameter Adaption for Neural Networks

So far, several methods of hyperparameter optimization have been proposed with difference in speed, efficiency, scalability and model dependency. Each method attempts to optimize the machine learning model $\mathcal{M}$ by finding the optimal hyperparameter configuration λ^* to be instantiated with the learning algorithm $\mathcal{A}$. However, current literature suggest that better predictive performance may be gained by finding a good hyper-parameter schedule $\{\lambda_t^*\}_{t=1}^T = \{\lambda_1^*, \lambda_2^*, \ldots, \lambda_T^*\}$ instead of a constant configuration [82, 158, 5, 30, 23]. In addition, simply training a machine learning model can take long time, depending on the algorithm $\mathcal{A}$, the hyperparameter configuration λ^*, as well as dataset $\mathcal{X}$. Therefore, recent studies have researched ways to perform hyper-parameter optimization and neural network training simultaneously, sometimes referred to as *automatic hyperparameter adaption*, allowing the hyper-parameter schedule to be obtained in a single training session. This section formulates automatic hyperparameter adaption for neural networks, which includes how neural networks are trained and optimized iteratively using a sequence of optimization steps.

2.3.1 Neural Networks

Neural networks are essential in order for evolutionary training of multiple networks to work, as algorithms such as PBT (described in Section 2.4) require the model $\mathcal{M}$ to be transferable to another instance of a learner, regardless of how many training iterations that have been performed. Unlike other learners where the model grows over time [57, 21, 27, 90, 154], neural networks retain their model shape and model size no matter how many training iterations are performed, making them excellent for parallel asynchronous training and model transfer in different training stages. Model transfer is made possible by copying the model data, i.e. the weights θ (commonly denoted $w \in W$), which connects the neurons $a \in A$ between each layer $l \in L$ in the network, to other neural network instances of the same model shape. The shape is decided based on the hyperparameters for network architecture, e.g. number of layers and neurons. In raw form, neural network models are usually represented by weight-matrices W and bias-matrices B, which leads to excellent computation speeds on graphical processing units (GPUs) because they are efficient with matrix multiplication [54].

When Neural Networks models are learning, they are essentially searching the solution space Θ for the best solution. Not to be mistaken by the hyperparameter search space Λ, the solution space can be viewed as the space of all functions that a neural network can approximate to any precision. It is commonly known that the size of the solution space heavily depends on the depth of the network and activation functions used in the network. With one or more hidden layers, the solution space can become very large, and it grows exponentially with the depth of the network [156].

Neural networks learn through trial and error, which is commonly referred to as training, an iterative process that extracts and re-applies knowledge about the solution space. The training process consists of many small *steps* which collectively optimizes the weights θ, e.g. by calculating the gradient of the objective function. One step includes one forward propagation and one back-propagation.

The mathematical steps required to perform forward and back-propagation is formulated in the following sections. For notations, weights are defined as w^l_{jk}, where the weight w is connected between the k-th neuron in the $(l-1)$-th layer and the j-th neuron in the l-th layer. A similar notation is used for biases and activations as well; the bias b^l_j is positioned at the j-th neuron in the l-th layer, and the activation x^l_j is positioned at the j-th neuron in the l-th layer.

Forward propagation

Forward propagation consist of computing the activations in each layer, where the activation x_{lj} at the j-th neuron in the l-th layer is defined as

$$x^l_j = \sigma\left(\sum_k w^l_{jk} x^{l-1}_k + b^l_j\right). \tag{2.3}$$

Here, the sum $\sum$ is of every activation in the previous $(l-1)$-th layer multiplied by the weights w^l_{jk} in the current layer and added to the bias b^l_j. After, the activation function σ is applied to the sum. Because the activation is related to the activations in the previous $(l-1)$-th layer, forward propagation must be computed sequentially from left to right, layer by layer.

Calculating each neuron individually is not the most efficient approach; As a matter of fact, Equation 2.3 can be rewritten in matrix form. In matrix form, each weight w that is connected to the neurons in the l-th layer makes up the components of the weight matrix W^l. For example, the weight w^l_{jk} is the same weight as the weight component in matrix W^l, at the j-th row and k-th column. Similarly, biases are defined similarly in matrix form as B^l for each l-th layer, where the bias components consists of b^l_j for all j's in the l-th layer. Lastly, the activations are defined as the matrix X^l, whose components are the activations b^l_j for all j's in the l-th layer. Now, Equation 2.3 can be re-defined in matrix form as

$$Z^l = W^l X^{l-1} + B^l, \tag{2.4}$$

$$X^l = \sigma\left(Z^l\right), \tag{2.5}$$

where the dot product of weight matrix W^l and activation matrix X^{l-1} is added to the bias matrix B^l in order to get Z^l. After that, the activation function σ is applied element-wise to the components in the matrix.

Back-Propagation

For backpropagation, the goal main objective is to calculate the partial derivatives $\delta C/\delta W$ and $\delta C/\delta B$ of the cost function C in the network. The cost function C is determined by the user. For example, MSE can be defined for weights w and biases b at single neurons as

$$\frac{\delta C}{\delta w_j} = \frac{1}{n}\sum_x x_j\left(\sigma\left(z\right) - y\right), \tag{2.6}$$

$$\frac{\delta C}{\delta b} = \frac{1}{n}\sum_x \left(\sigma\left(z\right) - y\right). \tag{2.7}$$

For all neurons in all layers, the cost function C can be generalized as

$$C = \frac{1}{N}\sum_{i=1}^{N}\left(f\left(x_i|\theta\right) - y_i\right)^2. \tag{2.8}$$

The error in the output layer, δ^L, is calculated using the formula

$$\delta^L = \bigtriangledown_X C \odot \sigma^{'}\left(Z^L\right). \tag{2.9}$$

In the formula, $\bigtriangledown_X C$ is a vector that contains the partial derivatives $\delta C/\delta x^L_j$, described as the rate of change of C with respect to the output activations. The error in the l-th layer, δ^l, is calculated using the error obtained in the next layer, δ^{l+1}, using the formula

$$\delta^l = \left(\left(W^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(Z^l\right), \tag{2.10}$$

where $(W^{l+1})^T$ is the transpose of the weight matrix W^{l+1} for the next layer. The transpose of the weight matrix is required in order to move backwards in the network and obtain the error at the l-th layer. Next, the dot product of the transposed matrix and the error in the next layer, δ^{l+1}, is calculated. Then, the result is multiplied element-wise

(Hadamard product, $\odot$) with $\sigma'(Z^l)$, where σ' is the derivative (inverse) of the activation function.

In the network, the rate of change of the cost with respect to any bias b is defined using

$$\frac{\delta C}{\delta b_j^l} = \delta_j^l, \tag{2.11}$$

which states that the rate of change $\delta C/\delta b_j^l$ is exactly equal to the error δ_j^l. Similarly, the rate of change of the cost with respect to any weight w is defined as

$$\frac{\delta C}{\delta w_{jk}^l} = x_k^{l-1}\delta_j^l, \tag{2.12}$$

and demonstrates using the quantities δ^l and x^{l-1} how to obtain the partial derivatives $\delta C/\delta w_{jk}^l$.

Summary

The formulas for forward propagation and back-propagation sets the foundation for how to train neural networks. Training consists of letting the network predict through forward propagation. The prediction is then used to calculate the gradient of the cost function which is used to update the weights and biases in hope that the update improves generalization towards an optimal solution. Algorithm 5 summarizes the operations required to compute the gradient of the cost function for networks represented by matrices.

Algorithm 5 Forward propagation and back-propagation in a neural network

```
1: function STEP($X, L, W, B, C, \sigma$)
2:     Set the corresponding activation $a^1$ for the input layer;
3:     for $l = 2, 3, \dots, L$ do
4:         $Z^l = W^l X^{l-1} + B^l$
5:         $X^l = \sigma\left(Z^l\right)$                              ▷ activations in the $l^{th}$ layer
6:     end for
7:     $\delta^L = \bigtriangledown_X C \odot \sigma'\left(Z^L\right)$                  ▷ error in the output layer
8:     for $l = L-1, L-2, \dots, 1$ do
9:         $\delta^l = \left(\left(W^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(Z^l\right)$        ▷ error in the $l^{th}$ layer
10:    end for
11:    return $\frac{\delta C}{\delta w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\delta C}{\delta b_j^l} = \delta_j^l$    ▷ The gradient of the cost function
12: end function
```

2.3.2 Formulating Automatic Hyperparameter Adaption

As demonstrated in the previous section, training neural networks occurs in small, consecutive steps which iteratively updates the network model by optimizing the objective function. These updates are often applied using an optimizer such as the Stochastic Gradient Descent (SGD), which has some suitable smoothness properties.

To simplify, each step can be generalized as

$$\theta_{t+1} \leftarrow \text{step}(\theta_t|\lambda_t), \tag{2.13}$$

which performs one forward propagation and back-propagation with the current model θ_t and current configuration of hyperparameters λ_t, returning the updated model θ_{t+1}. When training neural networks with a hyper-parameter schedule, steps are performed sequentially to converge towards some global or local minima in the neural network solution space using

$$\theta^* \approx \text{optimize}(\theta|(\lambda_t)_{t=1}^T) = \text{step}(\text{step}(\ldots\text{step}(\theta|\lambda_1)\ldots|\lambda_{T-1}|\lambda_T) \text{ [84]}. \tag{2.14}$$

In order to perform enough steps to approximate some optimal solution, one has to choose the number of steps T (i.e. iterations) to perform. Determining the appropriate T can be difficult, as the computational cost increases with how many steps that is required, and many steps does not always imply better predictive performance (as seen later in 4.3). The problem becomes more apparent for tasks that require a large dataset X, which further impedes the process. For such issues, SGD is a popular solution, as it lowers the computational resources needed and reduces the time it takes to train a network by updating the weights based on single samples $x \in X$ instead of the whole dataset $\mathcal{X}$. Still, the computational cost of steps can be very high, leading to the optimization of θ taking several days, or even several weeks. Given a poor selection of hyperparameters λ, there is also no guarantee for the neural network to be able to find a good solution or even converge towards any solution at all.

If not constant hyper-parameters are considered (e.g. same learning rate for all iterations), the number of possible hyperparameters grows exponentially for each step. In order to find the optimal weights θ^* and hyperparameter schedule $\{\lambda_t^*\}_{t=1}^T$, we must consider the following problem:

$$\theta^* \approx \text{optimize}(\theta|\lambda^*) \tag{2.15}$$

$$\{\lambda_t^*\}_{t=1}^T = \{\lambda_1^*, \lambda_2^*, \ldots, \lambda_T^*\} \approx \underset{\lambda\in\Lambda^T}{argmin}(\text{optimize}(\theta|\lambda)) \tag{2.16}$$

2.4 Population-Based Training

PBT [84, 85] is a novel, Lamarckian evolutionary approach in hyperparameter optimization for finding the optimal hyperparameter schedule $\{\lambda_t^*\}_{t=1}^T$ and model θ^* by training a series of neural network models in parallel. The method can be performed as quickly as other methods and has shown to outperform Random Search [17] in model performance on various benchmarks in deep reinforcement learning using A3Cstyle methods [135], as well as in supervised learning for machine translation [196] and Generative Adversarial Networks (GANs) [65]. While similar procedures have been explored independently [83, 47], PBT has gained increasing amount of attention since it was proposed. There has already been seen shown various use cases of PBT in AutoML, e.g. packages for hyperparameter optimization tuning [117, 75, 70] and frameworks [114, 115, 95]. PBT have also streamlined the experiment testing in different application-based domains with different machine learning approaches such as auto-encoders [94], reinforcement learners [84, 85, 60, 50, 211, 36, 71, 12, 138, 164, 58, 122, 200], neural networks [48, 84] and generative adversarial networks [84].

In order to approach $\{\lambda_1^*, \lambda_2^*, \ldots, \lambda_T^*\}$ and θ^* as described in Equation 2.15–2.16, PBT considers a population $\mathcal{P}$ consisting of N members $\{\theta_i\}_{i=1}^N$, initially formed with different hyperparameters $\{\lambda_i\}_{i=1}^N$ sampled from a uniform distribution. The goal is to determine the optimal model θ^* across the population, which PBT achieves by adapting the hyperparameters and copying weights based on some criteria. The approach defines two distinct methods, *exploit* and *explore*, that influence λ and θ. In short, the exploit method decides whether the member should continue exploring the current solution or simply abandon it. If exploration is considered, the explore method is used to provide a new set of hyperparameters λ. The initiative for exploitation and exploration depends on the individual performances of the entire population, where the worst performing members exploits and explores the best performing members characteristics, while the best performing members maintains their characteristics. Algorithm 6 summarizes the operations performed in the PBT procedure.

Algorithm 6 Population-Based Training (PBT) [84]

```
1: procedure TRAIN($\mathcal{P}$)                                    ▷ initial population $\mathcal{P}$
2:     for $((\theta, \lambda, p, t) \in \mathcal{P}$ (asynchronously in parallel) do
3:         while not end of training do
4:             $\theta \leftarrow$ step($\theta|\lambda$)        ▷ one step of optimization using hyperparameters $\lambda$
5:             $p \leftarrow$ eval($\theta$)                               ▷ current model evaluation
6:             if ready($p, t, \mathcal{P}$) then
7:                 $\lambda', \theta' \leftarrow$ exploit($\lambda, \theta, p, \mathcal{P}$)    ▷ use the rest of $\mathcal{P}$ to find better solution
8:                 if $\theta \neq \theta'$ then
9:                     $\lambda, \theta \leftarrow$ explore($\lambda', \theta', \mathcal{P}$)          ▷ produce new hyperparameters $\lambda$
10:                    $p \leftarrow$ eval($\theta$)                          ▷ new model evaluation
11:                end if
12:            end if
13:            update $\mathcal{P}$ with new $(\theta, \lambda, p, t+1)$                 ▷ update population $\mathcal{P}$
14:        end while
15:    end for
16: end procedure
```

All members in the population are trained and adapted individually and asynchronously in parallel until the end-criteria is met. In step 4, each member trains its model θ independently with the current hyperparameter configuration λ using the step-function. When the trained model is returned, it gets evaluated using the the eval-function, which obtains the current predictive performance p in step 5. In step 6, the member is evaluated whether is deemed ready, e.g. by having completed a minimum number of steps t, or reached some decided performance threshold, for exploitation and exploration. If ready, the member may exploit the rest of the population and explore new hyperparameters in step 7–10, or continue on without changing anything if exploitation does not yield a different model θ. At the end of the iteration in step 13, each individual member saves their current progress to the population $\mathcal{P}$ before they move on to their next $(t+1)$-th step.

The authors of PBT have suggested several different ways to perform exploitation and exploration, stating that the implementation depends entirely on the application. In the paper [84], the authors implemented exploit and explore differently for three of the learners that were tested: deep reinforcement learning using A3Cstyle methods [135], supervised learning for machine translation [196] and with GANs [65]. However, a common default approach seems to be initially outlined, and the full process is summarized in Algorithm 7.

Algorithm 7 Default Exploit and Explore [84]

```
1:  function ExploitAndExplore($\lambda, \theta, p, \mathcal{P}$)
2:      $F_{exploit} \leftarrow 0.2$                                  ▷ exploitation factor
3:      $m : (\lambda, \theta, p) \in \mathcal{P}$;                   ▷ $m$ is a particular member in the population
4:      $m_{elitist} \sim \mathcal{P}_{best}$,
        where $|\mathcal{P}_{best}| = \lfloor N \times F_{exploit} \rfloor$   ▷ sample a random top performing member
5:      if $m \neq m_{elitist}$ then
6:          $m \leftarrow m_{elitist}$
7:          for $k \in \{1, 2, 3, \ldots, D\}$ do
8:              $F_{explore} \sim \{0.8, 1.2\}$                    ▷ select random exploration factor
9:              $\lambda_k \leftarrow \lambda_k \times F_{explore}$   ▷ perform perturbation
10:         end for
11:     end if
12:     return $(\lambda, \theta, p) \in m$
13: end function
```

As shown in Algorithm 7, if a member is not performing well compared to the other members, it can have its weights and hyperparameters replaced by a random member $m_{elitist}$ from the top $\lfloor N \times F_{exploit} \rfloor$ performing members in the population, $\mathcal{P}_{best}$ (exploitation), where the inherited hyperparameter configuration is perturbed using random noise by multiplying with either 0.8 or 1.2 (exploration). If a member is one of the top performing members in $\mathcal{P}_{best}$, it is deemed an *elitist*, and can proceed with its current model and hyperparameter configuration without any modification. In order to apply this exploit and explore method, step 7–11 in Algorithm 6 are replaced by the operations described in Algorithm 7.

In addition to the exploit and explore implementation, PBT also requires a couple of additional functions and parameters. First of all, the user must define the end-criteria, which can be a function of the entire population $\mathcal{P}$ that decides how long the population will train for, or a function of individual members which decides when each members should stop. The user must also define the is-ready-function, which decides how frequent and under witch circumstance members should be adapted using exploit and explore. In addition, the user needs to assign the upper and lower bounds ($\mathbb{b}_L, \mathbb{b}_U$) of the hyperparameter search space Λ in order to generate the initial hyperparameter configuration λ_1 for every member in the population using a random uniform distribution.

PBT great time complexity assumes that the user has access to enough computational resources to parallelize the algorithm linear to how many members are in the population. The training is preferably conducted on the GPU, meaning that N GPUs are appropriate for a population of N members. However, obtaining enough processing devices can be quite expensive, especially for smaller research teams and laboratories with limited resources, and others have already proposed a distributed queuing system that allows for PBT to be carried out using fewer devices [218].

2.4.1 Advancements in Population-Based Training

Since the proposal of PBT, we have seen several suggestions for improvement of the standard algorithm, as well as variations that implement some of its heuristics [175], or other, entirely different algorithms that are simply inspired by it [69]. In this section, we will focus on the novel extensions to the original method that have been proposed so far.

Elfwing et al. [47] proposed Online meta-learning by parallel algorithm competition (OMPAC), which may arguably be the first method to implement hyperparameters optimization for deep reinforcement learning by the population-based approach. The difference between PBT and OMPAC lies in the selection method (i.e. exploitation method); all members in the population are evaluated synchronously after a decided number of episodes, where each member is approved for continuous learning based on stochastic universal sampling [11]. For experiments with Feudal Networks [197] on the Atari Learning Environment games [13], PBT used a exploitation strategy consisting of a truncation selection scheme that replaces the hyperparameters λ and weights θ of the bottom 20% performing members, using uniform sampling, by the top 20% performing members based on episodic reward. The experimental results of OMPAC suggests their selection scheme may be more effective in this example.

Cohen et al. [34] proposed Population-Based Training With Knowledge Sharing, which embraces the randomness between models instead of avoiding it by extending PBT with a novel distilling scheme [208] that enable knowledge sharing across generations. The authors argue that model diversity is important for the evolution of the population. The extended algorithm allows only the best individuals of the population to share some of their characteristics with the rest of the population, where the neural networks that achieve the best loss value are allowed to contribute to a shared teacher output for the training data. The contribution is reflected within the loss function of all networks in the population, and the teacher output affect the model training through the step-function. The extended algorithm was tested on the MNIST[1] [110], Fashion-MNIST[2] [207] and EMNIST [34] datasets, outperforming standard PBT on all tested cases.

Zhou et al. [217] proposed Two-stage population based training method for deep reinforcement learning (TS-PBT), which is an extension of PBT that proposes two stages: pre-training stage and hyperparameter adaptation training stage. The pre-training stage consists of training one member on fixed hyperparameters for a percentage of the total computational budget (in their case, 5% of the available resources). The goal of pre-training was to help the model obtain necessary knowledge about the hyperparameter search space Λ and its influence on the predictive performance of model $\mathcal{M}$ faster than the standard method. The second stage consists of the standard PBT algorithm with the exception that each member in the population $\mathcal{P}$ is initiated with with hyperparameter configuration obtained from the first stage, but perturbed with $0.8 \backsim 1.2$ (similar perturbing method that is used in PBT). Experiments are conducted on the Atari Learning Environment [13] and shows that TS-PBT outperforms PBT in all test environments; TS-PBT achieve faster convergence and 40% performance improvement over PBT in MsPacman, 310% in SpaceInvaders , 70% in SpaceInvaders, 2% in the Seaquest, 53% in BeamRider, 30% in Breakout and 38% in Qbert.

Zhou et al. [218] proposed an efficient online hyperparameter adaptation for deep reinforcement learning. Their method shares many similarities with PBT in the sense that a population of N members train in parallel and exploit their progresses, but there are some differences. First of, the top 20% performing members are marked as *elitists*, which lets them maintain their hyperparameter configuration across multiple steps. Secondly, the exploitation and exploration strategy is compressed into a recombination and mutation

[1]The MNIST dataset is available at `http://yann.lecun.com/exdb/mnist/`.

[2]The Fashion-MNIST dataset is available at `https://github.com/zalandoresearch/fashion-mnist`.

strategy. The recombination method is is performed between the remaining 80% bottom performing members, where it creates random pairs which randomly generate two intersections and exchange some hyperparameters. After recombination, the mutation method is performed similarly to the perturbing method used in PBT. Lastly, the algorithm runs using a distributed queue system instead of running asynchronously in parallel in separate processes, which is better for users with limited computational resources. For experiments, the method achieved 92% performance improvement over the PBT in MsPacmam, 70% in SpaceInvaders, 2% in the Seaquest, and 15% in BeamRider from the Atari Learning Environment [13].

2.4.2 Summary

Although PBT first appeared in 2017, various advances [34, 217, 218, 121] already show that the standard PBT algorithm can be greatly improved. In the original paper [84], PBT was also primarily tested on large datasets, and a recent study [195] shows that it may not perform better than random search on relatively smaller datasets, and that model transfer may even have a negative impact on the final predictive performance for smaller datasets. Interestingly, standard PBT, as well as many of the extensions, still rely heavily on the stochastic, random perturbation technique in order to carry out hyperparameter adaptation. In the next section, we take a look at an state-of-the-art evolutionary algorithm for global optimization, namely DE [179].

2.5 Differential Evolution

DE [179, 152] is a well-known population-based, direction-based, stochastic meta-heuristic method for global optimization of numeric hyperparameters of the continuous type [37, 3]. Unlike traditional evolutionary approaches, DE is derivative-free and uses the scaled differences between randomly selected members of the current population in order to generate the offspring's genome, which means that a separate probability distribution is not required in order to generate new hyperparameters [37]. The algorithm only requires three parameters, so no extensive tuning is needed in order to find appropriate values. DE has come a long way since it was first introduced by Ken Price and Rainer Storn in a series of papers that followed in quick succession in the late 1990s [179, 178, 176, 180], and later published in book-form in the late 2000s [152, 25]. As of today, the method and its derivatives still receives frequent extensions that improves the state-of-the-art results on a range of benchmarks and real-world applications [3], although with less significant advances. In addition, the initial implementation have recently shown promising results for hyperparameter optimization [163] when compared to SMAC [79], and further testing of more advanced methods [37, 3] was encouraged by the authors.

Similar to most evolutionary algorithms, DE creates a population $\mathcal{P}$ of N members, where each initial member is a randomly chosen $\mathcal{D}$-dimensional vector (point) over the parameter search space Λ, and represents a single hyperparameter configuration. The population is defined as

$$\begin{aligned} \mathcal{P}_x^g &= \left(\varkappa_i^g\right), i = 0, 1, \ldots, N-1, g = 0, 1, \ldots, g_{max}, \\ \varkappa_i^g &= \left(x_{j,i}^g\right), j = 0, 1, \ldots, \mathcal{D}-1, \end{aligned} \tag{2.17}$$

where N denotes the number of vectors in the population, $g = \in \{0, 1, \ldots, g_{max}\}$ is the generation counter, $i = \in \{0, 1, \ldots, N\}$ is the population index, and $j \in \{0, 1, \ldots, \mathcal{D}\}$ is the index of parameters within vectors. The dimensionality $\mathcal{D}$, which is the number of parameters to optimize. The members of the population is initialized via

$$x_{j,i}^{0} = \mathrm{rand}_j\,[0,1] \times (b_{j,U} - b_{j,L}) + b_{j,L}, \tag{2.18}$$

where the $\mathcal{D}$ dimensional initialization vectors, $b_{j,L} \in \mathbb{b}_L$ and $b_{j,U} \in \mathbb{b}_U$, indicate the lower and upper bounds of the parameter vector $x_{i,j} \in \mathbb{x}_i$. The hyperparameter search space Λ is restricted to these preset bounds. Every hyperparameter is initialized with the random number generator $\mathrm{rand}_j\,[0,1]$, which returns a uniformly distributed random number between 0 and 1, i.e. $0 \leq \mathrm{rand}_j\,[0,1] < 1$. The subscript j indicates that each hyperparameter gets assigned a random value.

One of the members in the population $\mathcal{P}$ could at some point represent a potential solution to the optimization problem. In order to generate new candidate solutions, DE first need to select three distinct members from the population to operate on, namely the *base vector* and two *difference vectors*. Each of these vectors are sampled from the pre-existing hyperparameter configurations in the population. When these are known, mutation is carried out by perturbing the base vector by some magnitude using the difference vectors, i.e.

$$\mathbb{v}_i^g = \mathbb{x}_{r0}^g + F \times \left(\mathbb{x}_{r1}^g - \mathbb{x}_{r2}^g\right), \tag{2.19}$$

where $\mathbb{v}_i^g$ is the generated mutation vector. The base vector $\mathbb{x}_{r0}^g$ and the difference vectors, $\mathbb{x}_{r1}^g$ and $\mathbb{x}_{r2}^g$, are selected based on the randomly generated indices $r0$, $r1$ and $r2$. These indices must be mutually exclusive. Over time, there have been proposed several novel ways to conduct mutation instead of using the approach described in Equation 2.19. For example, one approach considers selecting the current best vector as base vector, and it is also common to employ more than two difference vectors.

In DE, mutation is followed up with a crossover strategy for diversity enhancement, which generates a *trial vector* by selecting and mixing hyperparameters from the mutation vector $\mathbb{v}_{\mathbb{i}}^{\mathbb{g}}$ and the *target vector* $\mathbb{x}_{\mathbb{i}}^{\mathbb{g}}$. At least one component from the mutation vector $\mathbb{v}_i^g$ must be selected in order to satisfy $\mathbb{u}_i^g \neq \mathbb{x}_i^g$. This is ensured by randomly selecting the index of a gene $j_{rand} \in \{1, 2, \ldots, \mathcal{D}\}$. There are several variants of crossover [152] such as the commonly used *binomial crossover*, which is defined as

$$\mathbb{u}_i^g = \left(u_{j,0}^g, u_{j,1}^g, \ldots, u_{\mathcal{D},N}^g\right) = \begin{cases} v_{j,i}^g & \text{if}(\mathrm{rand}_j[0,1] \leq Cr) \text{ or } j = j_{rand} \\ x_{j,i}^g & \text{otherwise.} \end{cases} \tag{2.20}$$

The last step of the DE algorithm uses a greedy, one-to-one survivor selection scheme in order to determine whether the trial vector $\mathbb{u}_i^g$ or mutation vector $\mathbb{v}_i^g$ should survive to the next generation $g+1$. The survivor vector $\mathbb{x}_i^{g+1}$ becomes whichever vector that scores the lowest (or highest) objective function value. The selection for minimization problems is defined as

$$\mathbb{x}_i^{g+1=} = \begin{cases} \mathbb{u}_i^g & \text{if}(f(\mathbb{u}_i^g) \leq f(\mathbb{x}_i^g)) \\ \mathbb{x}_i^g & \text{otherwise.} \end{cases} \tag{2.21}$$

Equation 2.17 through Equation 2.20 describes the classic DE algorithm, and the full process is summarized in Algorithm 8. The approach can be classified as DE/rand/1/bin according the notation scheme proposed in [152]. The shorthand notation is a simple way to differentiate the various, initial implementations of DE. For example, DE/best/1/bin is the same as DE/rand/1/bin with the exception that it chooses the current best vector $\mathbb{x}^g_{best}$ instead of a random base vector $\mathbb{x}^g_{r0}$. Nowadays, the notation scheme DE/x/y/z seems to be depreciated [25] for more recent DE extensions, but it is still useful for describing the initial implementations of the algorithm [3].

Algorithm 8 Differential Evolution (DE) [179]

```
1: procedure DifferentialEvolution($N, F, Cr$)
2:     for $i \in \{0, 1, 2, \ldots, N-1\}$ do
3:         for $j \in \{0, 1, 2, \ldots, D-1\}$ do
4:             $x^0_{j,i} = \text{rand}_j\,[0,1] \times (b_{j,U} - b_{j,L}) + b_{j,L}$        ▷ set random starting points
5:         end for
6:     end for
7:     while $g \leq g_{max}$ do
8:         for $i \in \{0, 1, 2, \ldots, N-1\}$ do
9:             $r0 = \lfloor \text{rand}(0,1) \times N \rfloor$, where $r0 \neq i$        ▷ generate random indices
10:            $r1 = \lfloor \text{rand}(0,1) \times N \rfloor$, where $r1 \neq r0 \neq i$
11:            $r2 = \lfloor \text{rand}(0,1) \times N \rfloor$, where $r2 \neq r1 \neq r0 \neq i$
12:            $j_{rand} = \lfloor \text{rand}(0,1) \times \mathcal{D} \rfloor$        ▷ crossover dimension
13:            for $j \in \{0, 1, 2, \ldots, \mathcal{D}-1\}$ do
14:                if $\text{rand}(0,1) \leq Cr$ or $j = j_{rand}$ then
15:                    $u^g_{i,j} = v^g_{i,j} = x^g_{r0,j} + F \times \left(x^g_{r1,j} - x^g_{r2,j}\right)$        ▷ mutation
16:                else
17:                    $u^g_{i,j} = x^g_{i,j}$
18:                end if
19:            end for
20:            if $f\left(\mathbb{u}^g_i\right) \leq f\left(\mathbb{x}^g_i\right)$ then        ▷ selection
21:                $\mathbb{x}^{g+1}_i = \mathbb{u}^g_i$
22:            else
23:                $\mathbb{x}^{g+1}_i = \mathbb{x}^g_i$
24:            end if
25:        end for
26:    end while
27: end procedure
```

In general, DE considers three control parameters:

- the population size N, which is the total number of potential solutions in one generation;
- the mutation factor F, which is the amount of differentiation a perturbed solution can receive, also described as the exploration length; and
- the crossover rate Cr, which serves as the probability in which a offspring genome inherits the genes of a parent.

These parameters have great influence on the predictive performance of the algorithm, but leave much up to the tuning knowledge of individual users. In the next section, we will take a look at some of the recent advancements of DE which address these concerns.

2.5.1 Advancements in Differential Evolution

Over the years, there have been conducted extensive research on DE in general, and various extensions to the original algorithm have been proposed. Initially, the focus was primarily set on refining the DE algorithm and developing theories which explains its performance [152]. During this time, several preliminary recommendations where proposed on how to select appropriate parameter settings of DE [107, 153, 150, 151, 177]. From there, we have seen various improvements made on the first DE algorithm by proposing new selection rules for constraint handling [106, 105, 132], mutation schemes [52, 55, 56, 136, 137], crossover schemes [151, 210, 118], as well as approaches for parallel DE [202, 40]. Since the early 2000s, there is also an ongoing trend in developing *adaptive* DE [2, 22, 39, 53, 181, 190, 193, 194, 198, 199, 209, 215, 215], i.e. automatic adaption of DE parameters. In the following sections, we will describe some of these methods in more detail.

SHADE

Success-History Based Parameter Adaptation for Differential Evolution (SHADE) is an adaptive DE extension based of JADE [213] that adapts both the mutation factor F and crossover rate CR, effectively excluding these control parameters from user selection. When introduced, the method demonstrated to achieve significant advances in performance compared to the initial DE implementation for general optimization [187]. In order to adapt both F and CR, SHADE introduces new mutation, crossover and selection strategies. Because the algorithm changes so many aspects of the original implementation, it is covered in full in Algorithm 9. The following sections will provide more details about the specifics of the algorithm.

Control parameter assignment with historical memory. Unlike DE, SHADE maintains a historical memory that contains H entries of the DE parameters F and Cr, denoted M_F and M_{CR} respectively. The memory is represented with two separate lists of length H, and each entry is initialized with 0.5 as the starting value. Whenever a trial member is successful, the F_i and CR_i are recorded to the generation-assigned lists, S_F and S_{CR}, which are reset at the beginning of each generation.

For each member x_i in each generation g, the mutation factor F_i is calculated by using the procedure

$$F_{i,new} \sim \mathcal{C}_i\left(M_{F,r_i}, 0.1\right) \tag{2.22}$$

$$F_i = \begin{cases} 1.0 & \text{if } F_{i,new} > 1.0 \\ repeat\ \ 2.22 - 2.23 & \text{if } F_{i,new} < 0.0 \\ F_{i,new} & \text{otherwise.} \end{cases} \tag{2.23}$$

where the r_i is a random index in the historical memory drawn from a uniform distribution $\mathcal{U}[0, H)$. The mutation factor F_i is sampled from a Cauchy distribution $\mathcal{C}_i(\mu, \sigma)$, where the mean μ is set to the random memory value M_{F,r_i} and the standard deviation σ set to 0.1. The formula for drawing a random sample from a Cauchy distribution is defined as

$$\mathcal{C}_i(\mu, \sigma) = \mu + \sigma \times \tanh\left(\pi \times (p - 0.5)\right), \tag{2.24}$$

Algorithm 9 Success-History Based Parameter Adaptation for Differential Evolution (SHADE) [187]

```
 1: procedure SHADE($N, G, H, p, r^{arc}$)
 2:     Population $\mathcal{P} = \emptyset$, Archive $A = \emptyset$, Memory Index $k = 0$;
 3:     $|A| \leftarrow round\left[N^{init} \times r^{arc}\right]$;                ▷ set extended archive size
 4:     Set all $H$ values in $M_{CR}$ and $M_F$ to 0.5;
 5:     for $i \in \{0, 1, 2, \ldots, N-1\}$ do
 6:         for $j \in \{0, 1, 2, \ldots, D-1\}$ do
 7:             $x^0_{j,i} = \text{rand}_j\,[0,1] \times (b_{j,U} - b_{j,L}) + b_{j,L}$;    ▷ set random starting points
 8:         end for
 9:     end for
10:     while $g \leq G$ do
11:         $S_{CR} = \emptyset, S_F = \emptyset$        ▷ Reset historical memory records from previous $G$
12:         for $i \in \{0, 1, 2, \ldots, N-1\}$ do
13:             $x^g_{r1} \sim \mathcal{P}$, where $r1 \neq i$;     ▷ sample a distinct member from population $\mathcal{P}$
14:             $x^g_{r2} \sim \mathcal{P} \cup A$, where $r2 \neq r1 \neq i$;   ▷ sample a distinct member from $\mathcal{P} \cup A$
15:             $x^g_{pbest} \sim \mathcal{P}$;                  ▷ sample one of the $100 \times p\%$ best members
16:             $j_{rand} = \lfloor \text{rand}[0.0, 1.0] \times \mathcal{D} \rfloor$;                 ▷ crossover dimension
17:             Generate $F_i$ and $Cr_i$ with $A$;                     ▷ Equation 2.22 and 2.25
18:             for $j \in \{0, 1, 2, \ldots, \mathcal{D}-1\}$ do
19:                 if rand$(0,1) \leq CR^g_i$ or $j = j_{rand}$ then
20:                     $v^g_{i,j} = x^g_{i,j} + F^g_i \times \left(x^g_{pbest,j} - x^g_{i,j}\right) + F^g_i \times \left(x^g_{r1,j} - x^g_{r2,j}\right)$  ▷ mutation
21:                     if $v^g_{i,j} < b_{j,L}$ then                  ▷ constrain if out-of-bounds
22:                         $u^g_{i,j} = \left(b_{j,L} + x^g_{i,j}\right)/2$;
23:                     else if $v^g_{i,j} > b_{j,L}$ then
24:                         $u^g_{i,j} = \left(b_{j,U} + x^g_{i,j}\right)/2$;
25:                     else
26:                         $u^g_{i,j} = v^g_{i,j}$;
27:                     end if
28:                 else
29:                     $u^g_{i,j} = x^g_{i,j}$;
30:                 end if
31:             end for
32:             if $f\left(\mathbb{u}^g_i\right) \leq f\left(\mathbb{x}^g_i\right)$ then                                ▷ selection
33:                 $\mathbb{x}^{g+1}_i = \mathbb{u}^g_i$;
34:             else
35:                 $\mathbb{x}^{g+1}_i = \mathbb{x}^g_i$;
36:             end if
37:             if $f\left(\mathbb{u}^g_i\right) < f\left(\mathbb{x}^g_i\right)$ then
38:                 $\mathbb{x}^g_i \rightarrow A$;                    ▷ add parent member to external archive
39:                 $CR^g_i \rightarrow S_{CR}, F^g_i \rightarrow S_F$;         ▷ add parameters to historical memory
40:             end if
41:         end for
42:         if $S_{CR} \neq \emptyset$ and $S_F \neq \emptyset$ then
43:             Update $M_{F,k}$ and $M_{CR,k}$ based on $S_F$, $S_{CR}$;             ▷ Algorithm 10
44:         end if
45:         $g \leftarrow g + 1$;
46:     end while
47: end procedure
```

where p is a random value drawn from a uniform distribution $\mathcal{U}(0.0, 1.0]$. If F_i is higher than 1.0, it will be truncated to 1, and if F_i becomes less than 0.0, the Equation 2.22–2.23 repeats until a valid value is generated.

Similarly, for each member $\mathbb{x}$ in each generation g, the crossover rate CR_i is generated using the formula

$$CR_i^{new} \sim \begin{cases} 0 & \text{if } M_{CR,r_i} = \perp \\ \mathcal{N}_i\left(M_{CR,r_i}, 0.1\right) & \text{otherwise.} \end{cases} \tag{2.25}$$

$$CR_i = \begin{cases} 1.0 & \text{if } CR_i^{new} > 1.0 \\ 0.0 & \text{if } CR_i^{new} < 1.0 \\ CR_i^{new} & \text{otherwise.} \end{cases} \tag{2.26}$$

CR_i is set to 0.0 if the memory value M_{CR,r_i} is equal to the termination value $\perp$. Otherwise, CR_i is drawn from a normal distribution $\mathcal{N}_i(\mu, \sigma)$, with the mean μ set to the random memory value M_{Cr,r_i} and the standard deviation σ set to 0.1. If CR_i becomes an invalid value outside the range $[0.0, 1.0]$, it will be clamped to the closest boundary.

Mutation. After the control parameters F_i and CR_i are generated, the mutant vector $\mathbb{v}_i^g$ is calculated by applying the DE/current-to-$p$best/1/bin mutation strategy, which was initially used in JADE [213], and is a variant of the DE/current-to-best/1/bin mutation strategy. The p variable is the decimal percentage of how many of the top-performing members in the population $\mathcal{P}$ that is considered when randomly selecting $\mathbb{x}_{pbest}^g$. A small p-value increases the greediness, and it is commonly defaulted to $p = 0.2$, meaning the 20% top performing members are considered when selecting $\mathbb{x}_{pbest}^g$. The mutation strategy is defined as

$$\mathbb{v}_i^g = \mathbb{x}_i^g + F_i^g \times \left(\mathbb{x}_{pbest}^g - \mathbb{x}_i^g\right) + F_i^g \times \left(\mathbb{x}_{r1}^g - \mathbb{x}_{r2}^g\right). \tag{2.27}$$

It is possible that the DE/current-to-$p$best/1/bin will generate hyperparameters outside the search space bounds (b_L, b_U). In order to ensure that valid hyperparameters are generated, the procedure uses a method [213] for constraining $\mathbb{v}_{i,j}^g$ for all dimensions $j \in [0, D)$:

$$v_{i,j}^g = \begin{cases} \left(b_{j,L} + x_{i,j}^g\right)/2 & \text{if } v_{i,j}^g < b_{j,L} \\ \left(b_{j,U} + x_{i,j}^g\right)/2 & \text{if } v_{i,j}^g > b_{j,L} \\ v_{i,j}^g & \text{otherwise.} \end{cases} \tag{2.28}$$

After applying the constraint, the mutant vector $\mathbb{v}_i^g$ is crossed with the base vector $\mathbb{x}_i^g$ similar to the binomial crossover strategy in DE (Equation 2.20), except the crossover rate is replaced with the generated CR_i value from Equation 2.26, and the final formula is defined as

$$\mathbb{u}_i^g = \begin{cases} v_{j,i}^g & \text{if } (\text{rand}_j[0,1] \leq CR_i) \text{ or } j = j_{rand} \\ x_{j,i}^g & \text{otherwise.} \end{cases} \tag{2.29}$$

When all trials of the vectors $\mathbb{u}_i^g, i \in [0, N)$ in the population $\mathcal{P}$ have been generated, SHADE employs the same selection process that is used in DE (Equation 2.21).

Diversity with external archive. Just like JADE, SHADE also includes an optional approach for maintaining diversity by using an external archive A that keeps records of the parent members $\mathbb{x}_i^g$ that were replaced by the trial members $\mathbb{u}_i^g$ in the selection process (Equation 2.21). Where DE disposes individuals and thereby excludes them for passing their characteristics down to future generations, SHADE uses the archive in union with the current population, i.e. $\mathcal{P} \cup A$, as an additional source for sampling $\mathbb{x}_{r2}^g$ used in Equation 2.27. Just like in JADE, the archive has a predefined size $|A|$ decided by a scalar parameter r^{arc} multiplied with the population size N and rounded to the nearest integer, i.e $\lfloor N \times r^{arc} \rceil$. In order to ensure that A never exceeds the size limit, a random selection from the archive is removed before appending the next parent member, if the size of the archive $|A|$ is equal to or exceeds the maximum size limit, $\lfloor N \times r^{arc} \rceil$.

Updating the historical memory. For each member in each generation, whenever the control parameters F_i and CR_i are successful in generating a trial member $\mathbb{u}_i^g$ that performs better than the parent member $\mathbb{x}_i^g$, the control parameters are appended to S_F and S_{CR}, each respectively. When the generation reaches its end, the historical memory is updated using Algorithm 10.

Algorithm 10 SHADE memory update [187]

```
 1: procedure UpdateMemory($S_F, S_{CR}$)
 2:     if $S_{CR} \neq \emptyset$ and $S_F \neq \emptyset$ then
 3:         if $M_{CR,k}^g = \perp$ or $\max(S_{CR}) = 0.0$ then
 4:             $M_{CR,k}^{g+1} = \perp$;            ▷ terminate memory for crossover rate, or
 5:         else
 6:             $M_{CR,k}^{g+1} = \text{mean}_{WL}(S_{CR})$;            ▷ update memory for crossover rate
 7:         end if
 8:         $M_{F,k}^{g+1} = \text{mean}_{WL}(S_{CF})$;            ▷ update memory for mutation factor
 9:         if $k < |H|$ then
10:             k = k + 1;            ▷ increment entry index, or
11:         else
12:             k = 0;            ▷ reset entry index
13:         end if
14:     else
15:         $M_{F,k}^{g+1} = M_{F,k}^g$;            ▷ maintain historical memory as it is
16:         $M_{CR,k}^{g+1} = M_{CR,k}^g$;
17:     end if
18: end procedure
```

As shown in Algorithm 10, the memory is only updated if at least one of the individuals in the generation g is successful in generating a trial member that performs better than the parent member. Moreover, only one position in the memory is updated each time it is called. That position is decided by index $k \in 0 \leq k < H$, which is always initialized to 0 at the beginning of the algorithm. In other words, in generation g, the k-th element in memory is updated for the next generation $g + 1$. After the memory is updated, the index k is updated; if the index is equal to $H - 1$ or higher, it is reset to 0, otherwise it is incremented by 1.

Both M_F and M_{CR} uses the weighted Lehmer mean defined in Equation 2.30 for updating memory records. As shown, the current k-th entry in M_F is updated directly with $\text{mean}_{WL}(S_F)$. The k-th entry in M_{CR}, on the other hand, is updated only if two conditions are met: (1) the current memory record $M^g_{CR,k}$ is not equal to the termination value $\perp$, and (2) the maximum value in S_{CR} is higher than 0.0. If the conditions are not satisfied, the new memory record $M^{g+1}_{CR,k}$ is assigned the termination value *perp*. Otherwise, the new record is generated using the weighted Lehmer mean, $\text{mean}_{WL}(S_{CR})$.

The weighted Lehmer mean is calculating using

$$\text{mean}_{WL}(S) = \frac{\sum_{k=0}^{|S|} w_k \times S_k^2}{\sum_{k=0}^{|S|} w_k \times S_k}, \tag{2.30}$$

$$w_k = \frac{\Delta f_k}{\sum_{l=0}^{|S|} \Delta f_l}, \tag{2.31}$$

$$\Delta f_k = \left| f\left(\mathbb{u}^g_k\right) - f\left(\mathbb{x}^g_k\right) \right|, \tag{2.32}$$

where the weight w^g_k used to calculate the weighted Lehmer mean for S^g_k is defined as the absolute score Δf^g_k between the parent member $\mathbb{x}^g_i$ and trial member $\mathbb{u}^g_i$, as formulated in Equation 2.32, divided by the sum of all absolute scores for each S_k as shown in Equation 2.31. Finally, the weighted Lehmer mean is calculated by dividing the sum of all weights w multiplied with the square of all memory records S, with the sum of all weights w multiplied with all memory records S, as shown in Equation 2.31.

2.5.2 LSHADE

Over time, it is not uncommon to see more and more members ending up exploring the same region in the search space (although with finer granularity). Such scenarios may waste computational budget on late exploration that is deemed too excessive for the task at hand, and a case can be made for reducing the complexity of the algorithm so that more budget is spent on fewer and fewer individuals towards the end of the generation span.

To address this concern, SHADE with Linear Population Size Reduction (L-SHADE) [188] extends SHADE by applying linear decay to the number of members in the population, periodically eliminating the least performing members. While L-SHADE employs a rather simple linear population size reduction (LPSR) technique, it has shown to provide significant improvements over SHADE with same budget [188], given the reduction in time complexity that it provides.

In implementation, L-SHADE extends SHADE with a simple population size N adjustment formula; at the end of each generation g, the population size N in the next generation $g + 1$ is defined as

$$N_{g+1} = round\left[\frac{N^{min} - N^{init}}{NFE^{max}} \times NFE_g + N^{init}\right], \tag{2.33}$$

where N^{init} is the initial population size and N^{min} is the final population size minimum. NFE_g is the number of fitness evaluations after generation g, and NFE^{max} is the maximum number of fitness evaluations to be processed, which is incremented for every selection procedure (Algorithm 9, line 32). The full implementation of L-SHADE is described in Algorithm 11.

Algorithm 11 SHADE with Linear Population Size Reduction (L-SHADE) [188]

```
1: procedure L-SHADE($N^{init}, N^{min}, G, p, H, r^{arc}, NFE^{max}$)
2:     perform initialization as in SHADE;
3:     $NFE = 0$, $N = N^{init}$;
4:     while $g \leq G$ do
5:         perform mutation, crossover and selection as in SHADE;
6:         update historical memory as in SHADE;
7:         $N_{g+1} = round\left[\frac{N^{min} - N^{init}}{NFE^{max}} \times NFE_g + N^{init}\right]$
8:         if $N_g < N_g + 1$ then
9:             $\Delta N_g = N_g - N_{g+1}$
10:            $\mathcal{P} \leftarrow \mathcal{P} \setminus \mathcal{P}_{\Delta N_g \text{worst}}$;    ▷ delete the $\Delta N_g$ worst members in $\mathcal{P}$
11:            $|A| \leftarrow round\,[N_{g+1} \times r^{arc}]$;    ▷ resize archive size according to $|\mathcal{P}|$
12:        end if
13:        $g \leftarrow g + 1$;
14:    end while
15: end procedure
```

2.5.3 Current state-of-the-art DE advancements

Recently, adaptive DE has emerged as one of the best techniques used to improve upon the initial implementation. In order to obtain better performance, adaptive DE automates the control parameter tuning process by sequentially adjusting one or several of the control parameters $\{N, F, Cr\}$ using novel adaptation strategies. In 2018, an extensive study [3] of different state-of-the-art adaptive DE schemes was published, describing the various approaches in depth with focus on their advantages and disadvantages.

The study estimates that the overall best performances are achieved by DE algorithms with advanced mutation strategies such as JADE with archive [213], MDE_pBX [81] and DEGL [38], because these could handle deficiencies in the standard DE algorithm such as greediness in DE/best/1/bin strategy as well as maintaining balance between exploration and exploitation. When combining advanced algorithm schemes such as DE/current-to-*p*best/1/bin with parameter adaptive schemes, algorithms such as SHADE [187, 189] obtained the best algorithmic design performance. In addition, great performances is also achieved with dynamic selection of multiple DE strategies during the evolution process, e.g.in EPSDE [127], HSPEADE [128] and CoDE [201], which impose different search step size at each step in the evolution in order to guide the search towards better direction. Of the 24 different algorithms listed in the study, only a few are deemed superior in terms of performance.

Awad et al. [7] proposed ensemble sinusoidal differential covariance matrix adaptation with Euclidean neighborhood (L-SHADE-*cn*EpSin), which adopts the DE/current-to-*p*best/1/bin [213] mutation strategy. The algorithm uses a crossover operator that is modified by a covariance matrix learning $C = BDB^T$, with Euclidean neighborhood between the best member and the rest of the members in the population. Here, B and B^T are orthogonal matrices and D is diagonal matrix with Eigen values. The number of members in the population, N, is adapted in each generation, using the same technique as [6] for linear population size reduction. The F value is also adapted during the evolution using adaptive sinusoidal increasing adjustment and non-adaptive sinusoidal decreasing adjustment. For each generation, the crossover rate Cr is adapted using normal distribution as in L-SHADE.

Awad et al. [9] proposed ensemble sinusoidal differential evolution with niching reduction (EsDE$_\mathrm{r}$-NR), which is an enhanced version of L-SHADE with ensemble parameter sinusoidal adaptation (L-SHADE-EpSin) [6]. During the evolution, the algorithm updates the value of F with multiple significant adaptation merits that mixes two sinusoidal methods with a Cauchy distribution. Additionally, the algorithm reduces the population size N using a novel niching-based reduction scheme. In order to improve quality of the solutions found, the algorithm also implements a restart method that is activated when the population size is reduced to 20 members, where half of the members are re-initialized using a modified Gaussian walk formula [9].

Lastly, Awad et al. [8] proposed differential crossover strategy based on covariance matrix learning with Euclidean neighborhood (L-*covn*SHADE). Like L-SHADE-*cn*EpSin [9], the algorithm uses the DE/current-to-*p*best/1/bin [213] mutation strategy. In addition, the algorithm modifies the crossover operator with a covariance matrix learning $C = BD^2B^T$, with Euclidean neighborhood between the best member and the rest of the members in the population. Here, B and B^T are orthogonal matrices and D^2 is diagonal matrix with Eigen values. The mutation factor F is adapted with a Cauchy distribution, and the crossover rate Cr is adapted with normal distribution with the same methodology as used in L-SHADE.

Chapter 3

Methodology

This chapter provides the implementation details necessary to answer the research questions of this thesis. First, the implementation of the original PBT procedure is described in Section 3.1. In the next two sections, three methods of hyperparameter adaptation based of the PBT procedure incorporated with DE heuristics are proposed. The Population-Based Training with Differential Evolution (PBT-DE) procedure is first proposed in Section 3.2, describing how it applies the initial DE heuristics to PBT in order to improve upon the hyperparameter exploration strategy. In Section 3.3, the PBT-DE procedure is expanded upon, and two additional procedures are proposed, called Population-Based Training with SHADE (PBT-SHADE) and Population-Based Training with LSHADE (PBT-LSHADE), based of the adaptive DE schemes, SHADE [187] and L-SHADE [188], respectively. In Section 3.4, four different experiments using the MNIST and Fashion-MNIST datasets and MLP and LeNet-5 neural network architectures are established, including the implementation details about the neural network architectures, learning algorithms, datasets, hyperparameters, loss function and evaluation metrics that will be used to measure the performances of each procedure. Lastly, given that training multiple neural networks in parallel is a complicated process, and because there is limited information and established practices on how to carry out such a process, technical details about the system and the flow of major operations are included in Section 3.5.

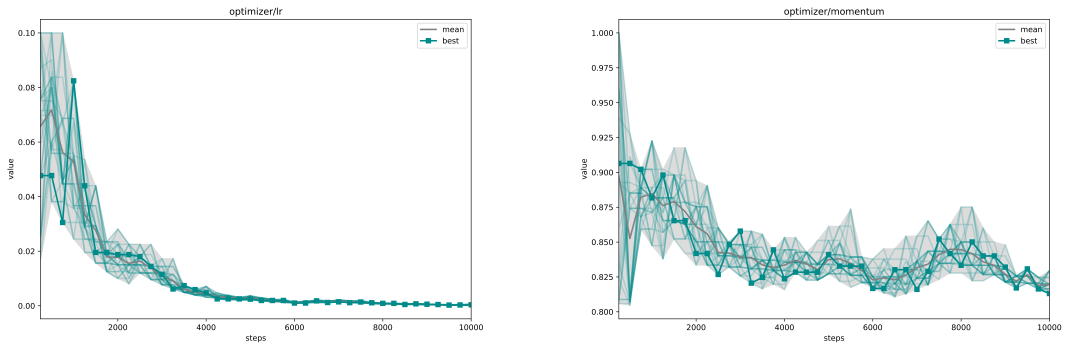
3.1 Implementing the PBT Baseline

Before DE heuristics could be incorporated, it was decided that a comparison measure was needed in order to know how the implementation performed. Therefore, PBT was implemented as described in Algorithm 6, using the assumed, default exploit and explore method as described in Algorithm 7, which exploits both θ and λ and explores just λ. As shown in the algorithm, the parameter for exploitation was defaulted to 0.8, which means 80% of the least performing members copies the θ and λ from the top 20% performing members. The perturbation parameters for exploration was defaulted to $\{0.80, 1.20\}$, which means that the hyperparameters are multiplied with either 0.80 or 1.20, chosen randomly based of a uniform distribution.

It was decided to implement the PBT procedure [84] using a distributed queuing system [47, 218] (see Section 3.5). This allows for members to be trained and adapted individually and asynchronously, and ensures that each member wont proceed to the next iteration before every individual in the population is finished. This effectively makes it possible to process members per generation rather than individually, and is essential for comparing the PBT procedure and the proposed procedures (see Section 3.2–3.3). The PBT procedure [84] was originally intended to be distributed over N processing devices (e.g. GPUs), which unfortunately allows for variance in computing speed across processors to affect individual member progression [218]. By ensuring that the training loop is synchronized between generations, member progression will not be affected by differences in processing devices.

Moreover, this thesis will not optimize hyperparameters that describe the neural network architecture, which ensure that each $\mathcal{M}$ is structured equally in terms of the number trainable network parameters. This ensures that the number of operations required to train, evaluate and adapt each member to be equal across generations, and it was confirmed early on in development that the observed wall-clock times for individual steps were almost equivalent on identical processing devices.

Current literature seem to indicated that hyperparameters are more likely to decay over time rather than increase over time when using the stochastic perturbation approach [84]. Early testing confirmed this notion, as it was noticed that that the implemented exploit and explore method produced similar schedules if enough time was given and assuming the chance of randomly selecting between 0.8 and 1.2 are equally weighted. Figure 3.1 demonstrates this behavior visually.



(a) learning rate (b) momentum

Figure 3.1: An example of two hyperparameter schedules generated by PBT.

3.2 Incorporating DE Heuristics Into PBT

Since its introduction, DE has received many improvements over the initial algorithm, proposing novel mutation, crossover and selection schemes that achieve state-of-the-art performances on the established benchmarks. Despite the promising work that has been done, it was not clear which procedure that would be a good candidate for improving PBT. Given the uncertainty, and the fact that this may be the first time PBT has been incorporated with DE for neural networks, it was decided to start out with the most initial approach: the DE/rand/1/bin strategy, described in Algorithm 8. The decision helped establish a firm baseline, which proved useful when incorporating adaptive DE heuristics later in Section 3.3.

This section describes the proposed PBT-DE procedure. The procedural structure was based of the common structure found in the original algorithm, as well as current extensions, described in Section 2.5. With this general knowledge, a list was constructed, containing each specific and distinct operation that must be considered (in the presented order) in addition to the operations required by the PBT procedure:

1. The operations preceding the next generation.
2. The method that generates the trial member with the mutation and crossover strategy and current population $\mathcal{P}$. This replaces the exploit and explore used in PBT.
3. The method that measures the fitness of the parent and trial in order to determine which is considered best, called $fitness$.
4. The method that selects between the parent and trial member based on their fitness score.
5. The operations after all members in the generation have selected their offspring.

Using these operations as basis for structuring the procedure, the first version of PBT incorporated with heuristics from DE was constructed, called PBT-DE, and is summarized in Algorithm 12.

PBT-DE uses most of the same algorithmic structure found in the PBT procedure, but it is merged with the necessary operations required by DE in order to incorporate DE heuristics, allowing members to be processed in generations. Like PBT, each member is trained, evaluated and adapted asynchronously in parallel, but the procedure wont proceed to the next generation before all members have completed their progression. Due to how different types of hyperparameters are implemented and represented numerically (see Section 3.5.4), all mathematical operations from DE were carried out without requiring any modifications of the DE/rand/1/bin strategy. In other words, the operations required for generating new hyperparameters were more or less identical to the original DE algorithm. This removed any potential bias in inducing unnecessary changes that might alter the heuristics of DE.

As shown in Algorithm 12, the PBT-DE procedure starts by initializing a population $\mathcal{P}$ of unordered members $m : \{\theta, \lambda, p, t\}$, where the initial hyperparameter configurations are sampled using a random, uniform distribution. Each individual performs t steps of training with θ and λ, followed up with a full evaluation of the trained θ before the hyperparameter search space is explored in order to generate the new θ'. If the *ready*-criterion is not met, the member skips the exploration of new hyperparameters and proceeds to step 20. However, if the the ready-criterion is met, a trial member is generated using the existing member, referred to as the *parent member*, and both the parent and trial member are measured and compared in order to decide which gets to pass on their genome. In the next sections, we will go into more detail about the algorithm and the specifics of each operation.

3.2.1 Parameters

Like most algorithms and procedures, PBT-DE requires a set of parameters that must be defined by the user. In addition to the population size N, the user must specify the

Algorithm 12 Population-Based Training with Differential Evolution (PBT-DE)

```
1:  procedure ADAPT($\mathcal{P}, N, t_s, t_e, \mathbb{b}_L, \mathbb{b}_U, F, CR$)
2:      initialize $\mathcal{P} \leftarrow \{m_1, m_2, \dots, m_N\}$                ▷ create $N$ members
3:      where $\{\theta_i, \lambda_i, p_i, t_i\} \in member$,
4:      $\lambda_{1,j} \sim \mathcal{U}[b_{j,L}, b_{j,U})$;        ▷ sample each $\lambda$ from a uniform distribution
5:      $t_1 \leftarrow 0$;
6:      while not end of training synchronously do
7:          for $m : \{\theta, \lambda, p, t\} \in \mathcal{P}$ asynchronously in parallel do
8:              $\theta, t \leftarrow \underset{b=t}{\overset{t_s}{step}}(\theta, \lambda)$;          ▷ $t_s$ steps optimization of $\theta$ with $\mathcal{X}^{(train)}$
9:              $p \leftarrow eval(\theta)$;                   ▷ full evaluation of $\theta$ with $\mathcal{X}^{(valid)}$
10:             if ready($\theta, \lambda, p, t, \mathcal{P}$) then
11:                 with $CR, F, \lambda_{r0}, \lambda_{r1}, \lambda_{r2}, j_{rand}$,            ▷ generate trial member
                    using DE/rand/1/bin, do
                    $\theta', \lambda', p', t' \leftarrow evolve(\theta, \lambda, p, \mathcal{P})$;
12:                 $\theta, p, t \leftarrow fitness(\theta, \lambda, p, t, t_e)$;     ▷ measure fitness with Algorithm 13
13:                 $\theta', p', t' \leftarrow fitness(\theta', \lambda', p', t', t_e)$;
14:                 if $p' \leq p$ then                                  ▷ selection
15:                     $m' \leftarrow \{\theta', \lambda', p', t'\}$;
16:                 else
17:                     $m' \leftarrow \{\theta, \lambda, p, t\}$;
18:                 end if
19:             end if
20:             $m' = m \rightarrow \mathcal{P}$;                                ▷ update population $\mathcal{P}$
21:         end for
22:     end while
23: end procedure
```

lower and upper boundaries, $\mathbb{b}_L$ and $\mathbb{b}_U$, of the hyperparameter search space Λ. Much like DE, the bounds are necessary for spawning the initial points in the search space as shown in step 4 in Algorithm 12. Moreover, the bounds are used for constraining the hyperparameters to a valid range as required by the DE/rand/1/bin strategy. Naturally, the user must also assign the mutation factor F and crossover rate CR that are required by initial DE algorithm.

There are also a couple of new parameters being introduced: the training steps t_s and fitness steps t_e. Of these, t_s describes how many steps of training to perform previous to new hyperparameter generation, and t_e describes how many steps to train and evaluate in order to measure fitness. Both t_s and t_e are decided by the user and will have an effect on both the performance and execution time of the algorithm. When the maximum number of generations G is know, the PBT-DE procedure has order of $O(G \times (n \times t_s + 2n \times t_e))$ complexity.

3.2.2 Members

Similar to PBT, PBT-DE represents each member in the population $\mathcal{P}$ with the tuple $\{\theta, \lambda, p, t\}$, where λ and p is the equivalent of the parent member $\mathbb{x}_i^g$ and the fitness score $f(\mathbb{x}_i^g)$ from DEs notation, respectively. As this thesis works primarily with neural networks,

θ represents the current weights and biases that represent the network model. Naturally, λ is the current hyperparameter configuration sampled from the search space Λ, and p is the current score that the member has obtained so far. Together, these values form the *member state*, which is given more detail in Section 3.4.3. Similarly, the trial member $\mathbb{u}_i^g$ is represented by the tuple $\{\theta', \lambda', p', t'\}$, which is the altered version of the parent member state. While not shown, t and t' remains the same for the parent and trial member.

3.2.3 Training and Evaluating

When comparing the PBT procedure with the proposed PBT-DE procedure, one of the operations that stand out is how the *step*-function is performed; particularly, the notation has changed to represent steps more clearly. To clarify, the phrase "t steps" is the shorthand notation for performing forward- and backward propagation t consecutive times with the training set $\mathcal{X}^{(train)}$ as defined in Equation 2.14. In this thesis, steps are performed using subsets of the training set, called *batches*, so t is also equivalent to the number of batched samples to use as inputs in the forward propagation. In Algorithm 12, where the total number of batches, $\mathcal{B}^{max}$, in the training set is the ceiling value of the number of samples in the training set $\mathcal{X}^{(train)}$ divided by the batch size $\mathcal{B}$, a training step is formulated as

$$\begin{aligned} \textbf{if} \quad & \mathcal{B}^{max} = \left\lceil |\mathcal{X}^{(train)}| / \mathcal{B} \right\rceil \\ \textbf{and} \quad & t' = (t + \Delta t) \bmod \mathcal{B}^{max}, \\ \textbf{then} \quad & \theta, t' \leftarrow \overset{\Delta t}{\underset{b=t}{step}}(\theta, \lambda). \end{aligned} \tag{3.1}$$

Underset $(b = t)$ marks the starting batch index in the training set $\mathcal{X}^{(train)}$, and the overset Δt is the number of batches, or steps, to perform from t. If the the number of batches $t + \Delta t$ exceeds the total amount of batches in $\mathcal{X}^{(train)}$, i.e. $t + \Delta t > B^{max}$, the out-of-bound indices will be replaced by the first $((t + \Delta t) \bmod \mathcal{B}^{max})$-th indices, effectively looping back to the start of $\mathcal{X}^{(train)}$.

The PBT-DE procedure includes the existing *eval*-function from PBT, defined as

$$p \leftarrow eval(\theta), \tag{3.2}$$

which measures the predictive performance of the network model θ with the entire validation set $\mathcal{X}^{(valid)}$.

The notation has also been extended to include evaluation of specific batch indices from $\mathcal{X}^{(valid)}$ using

$$p \leftarrow eval_K(\theta), \tag{3.3}$$

where K is the set of batch indices found in the batch-divided validation set $\mathcal{X}^{(valid)}$. When using this notation, evaluation will be performed exclusively with the specified batches pointed to by the indices K.

3.2.4 Generating New Trial Members

In step 11, the exploit and explore method from PBT has been replaced with the DE/rand/1/bin mutation strategy from DE, denoted by the *evolve*-function, which generates the new trial member $\{\theta', \lambda', p', t'\}$. Details about the specific procedure for generating the trial member ($\mathbb{u}$) is described in step 9–19 in Algorithm 8. λ' is the only components of the trial members that is different from the parent members, as the DE/rand/1/bin strategy was used exclusively to generate new hyperparameter configurations. This means that model transfer is not performed, as early testing indicated worse performance, and is discussed more in Chapter 5.

3.2.5 Measuring Fitness

One of the challenges with developing PBT-DE was implementing a method for evaluating different hyperparameter configurations on the same model θ. Unlike PBT, DE compares the pre-existing member (parent) and the generated member (trial) (Equation 2.21) in order to determine which one that gets to pass on their genome. The comparison is traditionally done by computing the fitness score, originally denoted $f(\cdot)$, of both the parent member and trial member and determining which score is considered the best. For non-complex fitness functions that would not take up much time compared to the remaining operations in the optimization algorithm, measuring fitness between members is a non-issue in terms of overall wall-clock time.

For neural networks, the only straight-forward way to test how good λ is for the current model *theta*, is to train *theta* with λ using training samples and evaluate performance using validation samples. In other words, the fitness function became essentially at least one step in a computational-heavy black-box optimization algorithm, i.e. deep neural network forward- and backward propagation with gradient descent. That made comparisons computationally more complex, as the operation would require a separate training step with $\mathcal{X}^{(train)}$, and a full evaluation of the updated θ' using $\mathcal{X}^{(valid)}$, for both the parent member as well as a the trial member.

Comparing members suddenly became a problem in terms of efficiency, and while it could be done as described above, the focus was set on finding another way to approximate the measure by reducing the amount of training and validation samples needed. In order to reduce complexity, a novel method for measuring the fitness of new hyperparameter configurations has been proposed, called Random Fitness Approximation (RFA), which acts as a surrogate for the actual fitness-function that is being optimized. In step 12 and step 13, RFA is used as the $fitness$-function in order to measure the performance of both the parent and trial member generated by the *evolve*-function. The RFA function is described in more detail in Algorithm 13.

Algorithm 13 Random Fitness Approximation (RFA)

```
1: function FITNESS($\theta, \lambda, p, t, t_e$)
2:     $\theta, t \leftarrow \overset{t_e}{\underset{b=t}{step}}(\theta, \lambda)$;    ▷ $t_e$ steps optimization of $\theta$ with $\mathcal{X}^{(train)}$
3:     $K \overset{rand}{\sim} \{r_0, \ldots, r_{(t_e-1)}\} \in \big[0, |\mathcal{X}^{(valid)}|\big)$;    ▷ sample $t_e$ random indices from $\mathcal{X}^{(valid)}$
4:     $p_r \leftarrow eval_K(\theta)$;    ▷ evaluation of $\theta$ with $|K|$ random samples from $\mathcal{X}^{(valid)}$
5:     $w \leftarrow \frac{t_e \times B}{|\mathcal{X}^{(valid)}|}$;    ▷ calculate the weight of the fitness measure
6:     $p \leftarrow p \times (1.0 - w) + p' \times w$;    ▷ calculate the weighted average of $p$ and $p_r$
7:     return $\theta, p, t$
8: end function
```

Of RFAs five arguments, θ, λ and p represents the member that is considered for fitness evaluation. Furthermore, t is the the current amount of steps that the member has performed, and t_e is the amount of steps (or batches) to use when training and evaluating the member. The algorithm starts by training the network model θ from the t-th step for t_e steps with the training set $\mathcal{X}^{(train)}$, which returns the updated θ and t. From there, the member is evaluated by selecting the indices of t_e random batches from the validation set $\mathcal{X}^{(valid)}$, which returns the unweighted score, p_r. Lastly, the member score p is updated to the weighted average between p and p_r in order to maintain balance between the approximated performance p_r and the previous, fully-evaluated performance p. This is done to ensure fair comparisons between individuals in the population $\mathcal{P}$. This also ensures that good characteristics that perform generally well across the entire evaluation set is prioritized.

3.2.6 Selecting Between Parent- and Trial Members

In step 14, after measuring the fitness score of both the parent and trial member with the $fitness$-function, the results are used to to compare the members. The selection determines whether the parent (θ, λ, p, t) or the trial $(\theta', \lambda', p', t')$ gets to pass on their characteristics; an essential step derived from DEs method of selection (Equation 2.21). Like the initial DEs implementation, the lowest (or highest) value wins the competition.

3.3 Incorporating Adaptive DE Heuristics

In the DE algorithm, the step size in the hyperparameter search space Λ is heavily dependent on the mutation factor F, which decides how much distance between the two randomly selected points in the search space sampled from the current generation that is used to generate new points. One may view F to be the equivalent of learning rate for neural network optimization. From literature, it is known that periodically adapting the learning rate, or other hyperparameters for that matter, improves neural network performance [82]. In PBT-DE, the control parameters F and CR are constant values that do not change after they have been assigned by the user. Similarly to how hyperparameter optimization has been successful for neural networks, it has been demonstrated that adaptive DE algorithms, which automatically and periodically adapt one or several control parameters, obtain far better results, and state-of-the-art DE extensions typically consists of one or more adaptive parameter schemes (see Section 2.5.1). In this section, two additional procedures are proposed, namely PBT-SHADE and PBT-LSHADE, and the following sub-sections introduces the implementation details.

3.3.1 Population-Based Training with SHADE

Given that many state-of-the-art DE extensions are based of SHADE [187], it was determined that this was a good adaptive DE scheme to incorporate, and the greedy DE/current-to-$p$best/1/bin mutation strategy was implemented in order to generate better trial members with adapted F and CR parameters. In order to generate F and CR, the historical memory of successful parameters was implemented. In order to obtain the $p$best member, the extended archive of parent members was implemented as well. The advantage of using SHADE is that it does not require any tuning of the mutation factor F, nor the crossover rate CR. Instead, it adds two new parameters that define the size of the historical memory

M and external archive A. All essential SHADE heuristics are covered in greater detail in the Section 2.5.1.

Following the same naming convention, the extended PBT-DE procedure is named PBT-SHADE. It implements all mandatory and optional operations from SHADE, and the procedure is described in Algorithm 14.

Algorithm 14 Population-Based Training with SHADE (PBT-SHADE)

```
1: procedure ADAPT($\mathcal{P}, N, t_s, t_e, \mathbb{b}_L, \mathbb{b}_U, p_{best}, r^{arc}$)
2:     initialize $\mathcal{P} \leftarrow \{m_1, m_2, \dots, m_N\}$            ▷ create $N$ members
3:     where $m_i \leftarrow \{\theta_i, \lambda_i, p_i, t_i\}$,
4:     $\lambda_{i,j} \sim \mathcal{U}[b_{j,L}, b_{j,U})$,        ▷ sample each $\lambda$ from a uniform distribution
5:     $t_1 \leftarrow 0$;
6:     Archive $A = \emptyset$; $M_{CR} \leftarrow \{0.5, \dots\}$; $M_F \leftarrow \{0.5, \dots\}$;      ▷ initialize SHADE
7:     $|A| \leftarrow round\left[N^{init} \times r^{arc}\right]$;           ▷ set extended archive size
8:     while not end of training synchronously do
9:         $S_F = \emptyset$; $S_{CR} = \emptyset$; $S_w = \emptyset$;        ▷ initialize/reset temporary memory
10:        for $m : \{\theta, \lambda, p, t\} \in \mathcal{P}$ asynchronously in parallel do
11:            $\theta, t \leftarrow \overset{t_s}{\underset{b=t}{step}}(\theta, \lambda)$;        ▷ $t_s$ steps optimization of $\theta$ with $\mathcal{X}^{(train)}$
12:            $s \leftarrow eval(\theta)$;         ▷ full evaluation of $\theta$ with $\mathcal{X}^{(valid)}$
13:            if ready($\theta, \lambda, p, t, \mathcal{P}$) then
14:                with $CR_i, F_i, \lambda_{r1}, \lambda_{r2}, \lambda_{pbest}, j_{rand}$,      ▷ generate trial member
                   using DE/current-to-pbest/1/bin, do
                   $\theta', \lambda', p', t' \leftarrow evolve(\theta, \lambda, p, \mathcal{P})$;
15:                $\theta, p, t \leftarrow fitness(\theta, \lambda, p, t, t_e)$;     ▷ measure fitness with Algorithm 13
16:                $\theta', p', t' \leftarrow fitness(\theta', \lambda', p', t', t_e)$;
17:                if $p' \leq p$ then                                   ▷ selection
18:                    $m' \leftarrow \{\theta', \lambda', p', t'\}$;
19:                else
20:                    $m' \leftarrow \{\theta, \lambda, p, t\}$;
21:                end if
22:                if $p' < p$ then
23:                    $m \rightarrow A$;                  ▷ add parent to external archive
24:                    $w_i \leftarrow |p - p'|$;          ▷ calculate the absolute delta score
25:                    $CR_i \rightarrow S_{CR}$; $F_i \rightarrow S_F$; $w_i \rightarrow S_w$;     ▷ extend temporary memory
26:                end if
27:            end if
28:            $m' = m \rightarrow \mathcal{P}$;                         ▷ update population $\mathcal{P}$
29:        end for
30:        if $S_{CR} \neq \emptyset$ and $S_F \neq \emptyset$ and $S_w \neq \emptyset$ then
31:            update ($M_F$, $M_{CR}$) using $S_F$, $S_{CR}$ and $S_w$;   ▷ update historical memory
32:        end if
33:    end while
34: end procedure
```

In step 9, before each asynchronous generation call, PBT-SHADE resets the temporary historical memory sets S_F, S_{CR} and S_w. Please notice that S_w is not present in SHADE

(Algorithm 9), and was included in order to store the absolute delta score w between p and p'. In SHADE, w is normally defined in the weighted Lehmer mean function as one of the weights which is used to update the historical memory. The change does not affect the results in any way, but greatly decrease the time it take to update the historical memory because the fitness function does not need to be called more than once for each member.

As mentioned, PBT-SHADE replaces the initial DE/rand/1/bin mutation strategy with DE/current-to-$p$best/1/bin, which introduces a new parameter p_{best} that is decided by the user, and share close similarities with the exploitation factor in PBT. The p_{best} variable is the decimal percentage of the best members in the generation that is needed to sample λ_{pbest} (or $\varkappa_{pbest}$) which is used to generate trial members. Smaller p_{best}-values increase the greediness as fewer top-performing members are considered for sampling. In SHADE heuristics, p_{best} is referred to as p, but to avoid collision with the already-established PBTs parameter p, for score, the notation was changed.

From step 22 to 26, the external archive A and temporary memory (S_F, S_{CR}, S_w) is updated if the trial member outperforms the parent member. While the entire parent member state $\{\theta, \lambda, p, t\}$ is appended to the external memory as denoted by $m \leftarrow A$, only the hyperparameter configuration λ and fitness score p is used.

In step 31, after all members have completed the current generation, the next entries $M_{F,k}$ and $M_{CR,k}$ in the historical memory is updated by calculated the weighted Lehmer mean of both S_F, S_{CR} with the weights S_w. The entry index k is then incremented to the next entry or reset to zero if the current entry is positioned at the last index.

3.3.2 Population-Based Training with LSHADE

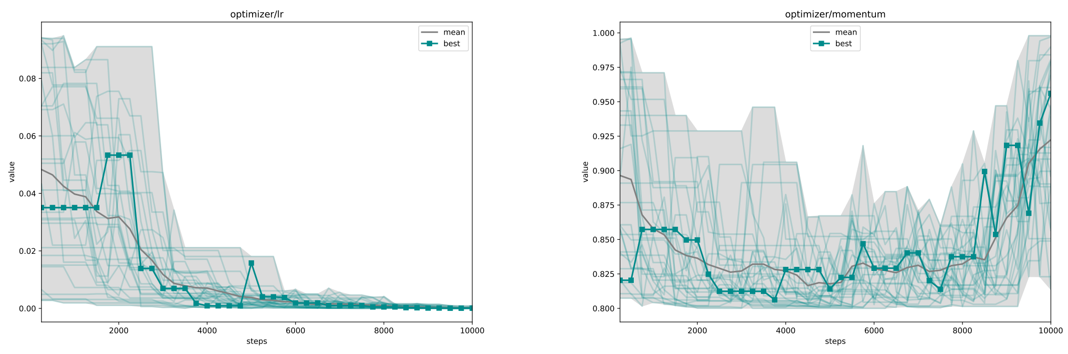
In order to fit more training steps within the time budget, it was decided that the next logical step would be look at how the budget is spent. Early testing indicated that the population of members would work towards a similar solution across the training span, regardless of how scattered they initially where. Figure 3.2 demonstrates this behavior. This decreased the average distance between points in Λ, and effectively promoted shorter steps within the search space region. Seeing that members would end up exploring a smaller and smaller region in the search space (although with finer granularity), there may be a possibility that computational resources are wasted in similar hyper-parameter configurations. These resources might be better spent training a smaller portion of the population for longer, which can be implemented by periodically eliminating bad members from the population, effectively shrinking the population size over time. For general optimization, there has already been proposed an novel extension to SHADE that implements this scheme.

L-SHADE [188], being an extension SHADE, was used to extend the PBT-SHADE procedure to include the additional operations that perform linear population size reduction over time, and the resulting procedure was named PBT-LSHADE. In practice, these operations shrink the population size from the initial size down to specified target size, which would free up resources over time and allow for two possibilities: (1) it could allow for more iterations within the same time budget, or (2) reduce the time complexity of the algorithm. The L-SHADE algorithm is described in more detail in Section 2.5.2. The PBT-LSHADE procedure is summarized in Algorithm 15.

In Algorithm 15, step 35–40 describes the population size reduction. First, the new population size N_{g+1} is calculated, which is decided by the initial population size N^{init} the target population size N^{min}, the target maximum number of fitness evaluations NFE^{max},

Algorithm 15 Population-Based Training with LSHADE (PBT-LSHADE)

```
1: procedure ADAPT($\mathcal{P}, N^{init}, N^{min}, NFE^{max}, t_s, t_e, \mathbb{b}_L, \mathbb{b}_U, p_{best}, r^{arc}$)
2:     initialize $\mathcal{P} \leftarrow \{m_1, m_2, \ldots, m_{(N^{init})}\}$    ▷ create $N$ members
3:     where $m_i \leftarrow \{\theta_i, \lambda_i, p_i, t_i\}$,
4:     $\lambda_{1,j} \sim \mathcal{U}[b_{j,L}, b_{j,U})$,    ▷ sample each $\lambda$ from a uniform distribution
5:     $t_1 \leftarrow 0$;
6:     Archive $A = \emptyset$; $M_{CR} \leftarrow \{0.5, \ldots\}$; $M_F \leftarrow \{0.5, \ldots\}$;    ▷ initialize SHADE
7:     $|A| \leftarrow round\left[N^{init} \times r^{arc}\right]$;    ▷ set extended archive size
8:     Number of fitness evaluations $NFE \leftarrow 0$;    ▷ initialize L-SHADE
9:     while not end of training synchronously do
10:        $S_F = \emptyset$; $S_{CR} = \emptyset$; $S_w = \emptyset$;    ▷ initialize/reset temporary memory
11:        for $m : \{\theta, \lambda, p, t\} \in \mathcal{P}$ asynchronously in parallel do
12:            $\theta, t \leftarrow \underset{b=t}{\overset{t_s}{step}}(\theta, \lambda)$;    ▷ $t_s$ steps optimization of $\theta$ with $\mathcal{X}^{(train)}$
13:            $s \leftarrow eval(\theta)$;    ▷ full evaluation of $\theta$ with $\mathcal{X}^{(valid)}$
14:            if ready$(\theta, \lambda, p, t, \mathcal{P})$ then
15:                with $CR_i, F_i, \lambda_{r1}, \lambda_{r2}, \lambda_{pbest}, j_{rand}$,    ▷ generate trial member
                   using DE/current-to-$p$best/1/bin, do
                   $\theta', \lambda', p', t' \leftarrow evolve(\theta, \lambda, p, \mathcal{P})$;
16:                $\theta, p, t \leftarrow fitness(\theta, \lambda, p, t, t_e)$;    ▷ measure fitness with Algorithm 13
17:                $\theta', p', t' \leftarrow fitness(\theta', \lambda', p', t', t_e)$;
18:                if $p' \leq p$ then    ▷ selection
19:                    $m' \leftarrow \{\theta', \lambda', p', t'\}$;
20:                else
21:                    $m' \leftarrow \{\theta, \lambda, p, t\}$;
22:                end if
23:                if $p' < p$ then
24:                    $m \rightarrow A$;    ▷ add parent to external archive
25:                    $w_i \leftarrow |p - p'|$;    ▷ calculate the absolute delta score
26:                    $CR_i \rightarrow S_{CR}$; $F_i \rightarrow S_F$; $w_i \rightarrow S_w$;    ▷ extend temporary memory
27:                end if
28:                $NFE \leftarrow NFE + 1$ ▷ Increment the number of fitness evaluations by 1.
29:            end if
30:            $m' = m \rightarrow \mathcal{P}$;    ▷ update population $\mathcal{P}$
31:        end for
32:        if $S_{CR} \neq \emptyset$ and $S_F \neq \emptyset$ and $S_w \neq \emptyset$ then
33:            update ($M_F$, $M_{CR}$) using $S_F$, $S_{CR}$ and $S_w$;    ▷ update historical memory
34:        end if
35:        $N_{g+1} = round\left[\frac{N^{min} - N^{init}}{NFE^{max}} \times NFE + N^{init}\right]$
36:        if $N_g < N_g + 1$ then
37:            $\Delta N_g = N_g - N_{g+1}$
38:            $\mathcal{P} \leftarrow \mathcal{P} \setminus \mathcal{P}_{\Delta N_g \text{worst}}$;    ▷ delete the $\Delta N_g$ worst members in $\mathcal{P}$
39:            $|A| \leftarrow round\left[N_{g+1} \times r^{arc}\right]$;    ▷ resize archive size according to $|\mathcal{P}|$
40:        end if
41:    end while
42: end procedure
```



(a) learning rate (b) momentum

Figure 3.2: An example of hyperparameter schedules generated by DE.

and the current number of fitness evaluations NFE registered so far. For each member, after each selection between the parent and trial, the number of fitness evaluations NFE is incremented by 1. If N_{g+1} is smaller than the current population size N_g, the worst ΔN_g members are deleted from the population $\mathcal{P}$.

3.4 Experiments

The following section covers the implementation details about each experiment performed in this thesis. For hyperparameter optimization on neural networks, an experiment usually defines a particular dataset and network combination that is commonly used for performance testing and analysis in a specific research field or for a certain application. To ensure that the results are repeatable, all used datasets and network architectures are well documented and publicly available. In order to carry out experiments, the MNIST [110] and Fashion-MNIST [207] datasets were used to evaluate model performances, as these are light-weight datasets for image classification that are still very popular within the research community. Complimenting the datasets, the MLP [134] and LeNet-5 [110] neural network architectures were used to define the network model. All experiments used the the SGD optimizer and the Categorical Cross Entropy (CCE) [41] loss function for updating the network model, and model performances are reported in the CCE, F1 [29] and accuracy score. The specific implementation details are reserved for the following sub-sections.

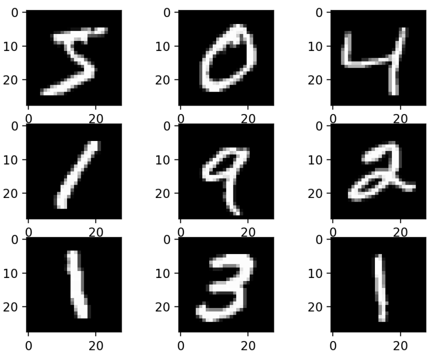
3.4.1 Datasets

The MNIST [110] and Fashion-MNIST [207] datasets are divided into three distinct sets for training, validation and testing as described in Table 3.1. A subest from both datasets are visually presented in Figure 3.3. These datasets were selected due to their maturity and popularity as benchmarks for hyperparameter optimization, among many other applications. In addition, the images contained within these datasets are black and white and of low resolution, which reduces the time and network architecture size required to process them. This is favorable, as training multiple neural networks in parallel is a computationally heavy task.

Table 3.1: Dataset Divisions

Set	MNIST	Fashion-MNIST
training	50 000	50 000
validation	10 000	10 000
testing	10 000	10 000
Total	70 000	70 000

Note. The numbers of samples, grouped by three distinct sets extracted from the MNIST and Fashion-MNIST dataset.


(a) MNIST

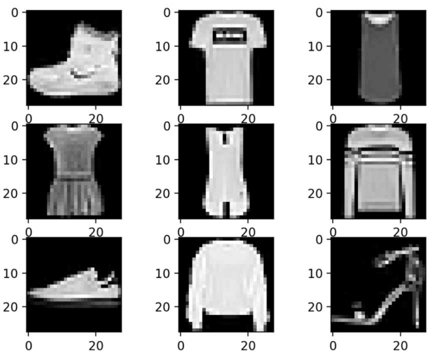

(b) Fashion-MNIST

Figure 3.3: Subsets from the MNIST and Fashion-MNIST datasets.

MNIST

MNIST is a well-established and well-researched dataset of real-world data for machine learning classification and pattern recognition. The dataset consists of exactly 70 000 images of handwritten digits with a resolution of 28x28 pixels in 1-channel (grayscale). The images are classified as single digits from 0 to 9. While the dataset could be considered *solved* in the sense that a near-perfect score has already been achieved [102], it is still used for benchmarking in current studies, including studies of PBT-extensions [143]. The dataset comes pre-divided into a training set of 60 000 images and a testing set of 10 000 images. In order to generate the validation set, 10 000 images from the training set was sampled using a stratified random sampling technique in order to maintain the same class distribution found in the training set. It is worth mentioning that the distribution was already highly balanced, so the training set might have consisted of enough samples that a simple random sampling would have provided a similarly balanced distribution.

Fashion-MNIST

Similarly to MNIST, Fashion-MNIST also contains 70 000 grayscale images with a resolution of 28x28 pixels that are associated with a label from 10 classes. The images are of different types of human clothing. While similar in form, Fashion-MNIST classification is considered a harder task for learners to perform, as MNIST digits are simply more distinguishable than images of clothing, resulting in a considerable gap in performance by the state-of-the-art [182, 206, 89, 87, 142, 74]. Similar to MNIST, Fashion-MNIST comes pre-divided into a training set and testing set, and the same stratified random sampling technique was used to generate the validation samples. The result was a training set of numprint50000 images, validation set of 10 000 images and testing set of 10 000 images.

Data Augmentation

Both MNIST and Fashion-MNIST were preprocessed using the same data augmentation techniques. All images were normalized with a mean of 0.1307 and standard deviation of 0.3081. All images were also increased to a resolution of 32x32 pixels using zero-padding, a technique which simply adds black pixels to all sides of the image in order to increase its resolution without increasing its quality or perform scale transformations. This was necessary in order to use the current network implementations of MLP and LeNet-5, as these expect an image input resolution of 32x32 pixels.

3.4.2 Models

As previously mentioned, the architecture of a neural network model can heavily influence the final model performance for any regression or classification task. In order to ensure repeatability for the experiments, it was decided to adopt two pre-defined architectures that vary in depth and complexity and are commonly used in literature: MLP [134, 143] and LeNet-5 [110]. The networks are in some aspects outdated compared to more complex, deeper network architectures, e.g. VGG16 [169], but they take considerable less time to train on limited hardware, given the number of free parameters each architecture consists of.

MLP

MLP is a feedforward neural network with multiple densely connected layers. While this model is very simple in design, especially compared to more recent architectures for pattern recognition, it could be described as the quintessential example of a deep learning model [64]. Just recently, MLP was used as one of the testing models in a PBT related paper [143]. As described in Table 3.2, the implementation of MLP consists of 3 densely connected layers followed up with a linear output layer. Each dense layer is followed up with a Rectified Linear Unit (ReLU) [139] activation layer.

Table 3.2: MLP Implementation

layer	neurons	parameters
dense	256	200 960
dense	128	32 896
dense	64	8 256
linear	10	650
total parameters		242 762

Note. Each dense layer is followed up with ReLU activation.

LeNet-5

The second model used in the experiments was LeNet-5 [110], a classical Convolutional Neural Network (CNN) used for handwritten and machine-printed character recognition. The LeNet-5 architecture consists of two pairs of convolutional and pooling layers, which are followed by a flattening convolutional layer connected to two fully-connected dense layers which is completed by a softmax classifier. Each pooling layer and dense layer is followed up with a ReLU activation layer. The implementation is described in Table 3.3.

Table 3.3: LeNet-5 Implementation

layer	neurons	kernel size	stride	parameters
convolutional	6	5×5	1×1	156
max pooling	6	2×2	2×2	0
convolutional	16	5×5	1×1	2 416
max pooling	16	2×2	2×2	0
flatten	-	-	-	0
dense	120	-	-	48 120
dense	84	-	-	10 164
linear	10	-	-	850
total parameters				61 706

Note. Each pooling layer and dense layer is followed up with ReLU activation.

3.4.3 Defining the Hyperparameter Search Space

In this thesis, hyperparameters are divided into two groups: The first is defined as the hyperparameters that influence the neural network model, and the second is defined as the hyperparameters that influence the optimization procedure. The same optimizer procedure is used for all experiments. A overview of the different types of hyperparameters that are optimized in the experiments are summarized in Table 3.4.

Table 3.4: Hyperparameter Search Space Configuration

name	group	type	lower bound	upper bound
learning rate	continuous	float	10^{-5}	10^{-1}
momentum	continuous	float	0.8	1.0
weight decay	continuous	float	0.0	10^{-3}

Note. A summary of each hyperparameter considered for optimization, including the lower- and upper boundaries.

As shown, Table 3.4 lists the lower- and upper boundaries vectors, $\mathbb{b}_L = \{10^{-5}, 0.8, 0.0\}$ and $\mathbb{b}_U = \{10^{-1}, 1.0, 10^{-2}\}$, respectively, and these are used to define the explorable region in the search space Λ. The values are chosen based on the general knowledge of the machine learning research community. Smaller and larger values than the ones specified are uncommon. Negative values are considered invalid.

Parameterization of the Network Model

For neural networks, the hyperparameter search space is first and foremost the parameters that define the solution space, i.e. the model architecture. It is important to establish that this thesis will not be including hyperparameters that change the number of neurons and hidden layers. As previously hinted, the reasoning behind that decision is that neural networks of unequal sizes cannot directly inherit each others weights and biases as the matrices that represent them are of unequal dimensions. While there exists frameworks that propose adaption of model architecture hyperparameters [115], this will not be tested in this thesis. However, that could be an excellent follow-up study, knowing that the network architecture have great influence on how well a network model is able to learn certain tasks.

Parameterization of the Network Optimizer

Except from model architecture that defines the solution space, most of the hyperparameters for neural networks are conventionally associated with the optimizer, and it was decided to use the SGD-optimizer for all experiments. SGD is an excellent and efficient optimizer with smoothing properties, and it was implemented in the original PBT paper [84]. Like most optimizers, common implementations of SGD[1] considers a couple of hyperparameters that can be optimized.

It is known that the number of training steps is essential to the final outcome of the model, where more training steps gives the learning optimizer more chances to update the network. Increasing the number of steps generally result in better performance, but only up to a certain point in time. The learning rate η is generally one of the most commonly tuned hyperparameters for neural network optimization, and influences the amount of update performed on the weights and biases by the optimizer in each training step, which decides how the neural network solution space is navigated. Typically, when traversing the solution space, the measured loss would usually end up in a region with either a local or global minima. If the learning rate is assigned too small, the amount of update applied will be smaller, resulting in slower learning speed, as well as higher chance that the network ends up in a local minima, but learning is also more stable. On the other hand, large learning rates allows for larger updates, which results in faster initial learning, and the learner is less likely to become stuck in a local minima, but learning might become unstable and more likely to miss good regions in the solution space. The learning rate is typically assigned to a value of 0.1 or lower, so it was decided to use the lower- and upper bounds of $[10^{-5}, 10^{-1}]$ for all experiments.

One of the most common hyperparameter associated with SGD is called the *momentum* [148, 155], which is a simple addition to the original algorithm. The hyperparameter is parameterized as a continuous numeric value commonly defaulted to 0.9, and helps accelerate gradients in the right direction. When momentum is applied, the optimizer updates the neural network model according to the exponentially weighted average of the gradient. It is known that when the network is properly initialized, SGD performs remarkably better with momentum if the hyperparameter is properly tuned [183]. It is also commonly known that good momentum values usually reside between $[0.8, 1.0]$, so these were used as the lower- and upper bounds in the search space for all experiments.

[1]SGD is implemented using the `optim`-module from PyTorch: `https://pytorch.org/docs/stable/optim.html`.

Current implementations of SGD also typically include regularization techniques such as *weight decay* [146, 1]. Weight decay, also known as *L2 penalty*, is parameterized with a continuous numeric value that controls the amount of regularization that is to be applied. Regularization is commonly used as a counter measure for overfitting, which is not uncommon for neural network solution spaces that are very rich. The weight decay hyperparameter is traditionally set very small (e.g. 0.01), and then manually balanced according to the model performance on unseen data in order to reduce generalization error. For all experiments, it was decided to constrain weight decay values the the lower- and upper bounds of $[0.0, 10^{-3}]$.

3.4.4 Metrics

In machine learning, metrics are specific types of measurements that have been developed to paint a picture of how well a model $\mathcal{M}$ performs in a certain area. So far, there is no universal way to measure network performance, although some metrics are used more than others. There exists a large variety of metrics for assessing models like neural networks, where some focuses on simply calculating how far the prediction is from the ground truth [170], to others who weights the measurement according to how well certain classes are represented [120]. Metrics are mainly created for three different types of tasks: regression, binary classification and multi-class classification. It is common to select at least two metrics for hyperparameter optimization; one metric for measuring loss $\mathcal{L}$ that is to be minimized by the optimizer that updates the neural network, and a validation metric for evaluating how well the model performs after training with the current hyperparameter configuration.

When measuring the performance of the hyperparameter algorithms relevant to this thesis, it was decided to use the same metrics for all experiments. Because each experiment considers a multi-class classification problem, the metric used for measuring loss, i.e. the loss function $\mathcal{L}$, is the CCE Score [41]. The evaluation metric used is called the F1 Score [29]. Lastly, accuracy was used as a third metric as it is very common in literature, used primarily for monitoring and analysis and has no effect on the algorithm.

Loss

CCE [41] is a class probability metric and loss function for both binary and multi-class classification. The metric is both easy to interpret and commonly used for various classification tasks. The equation for calculating the CCE[2] loss is defined as

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} 1_{y_i} \in C_c \log\left(P_{\text{model}}\left[y_i \in C_c\right]\right) \tag{3.4}$$

where the double sum is over the N observations, and over the C classes. $1_{y_i \in C_c}$ is a therm for the indicator function of observation i^{th} belonging to class c^{th}. The $p_{model}[y_i \in C_c]$ is the probability predicted by the classifier for the i-th observation to belong to the cth class. For multi-class classification (i.e. for more than two classes), the CCE outputs a vector C probabilities, where each component contains the probability of how likely the input belongs to a specific class, and the sum of these probabilities adds up to 1.0.

[2]CCE is calculated using the `torch.nn` module from PyTorch, at `https://pytorch.org/docs/master/generated/torch.nn.CrossEntropyLoss.html`.

Evaluation

The F1 score [29], also known as balanced F-score or F-measure, is another class-based metric. The metric can be described as the weighted, harmonic mean of the precision and recall (sensitivity). Precision is defined as

$$\begin{aligned} Precision &= \frac{True\ Positive}{True\ Positive + False\ Positive} \\ &= \frac{True\ Positive}{Total\ Predicted\ Positive}, \end{aligned} \tag{3.5}$$

and recall is defined as

$$\begin{aligned} Recall &= \frac{True\ Positive}{True\ Positive + False\ Negative} \\ &= \frac{True\ Positive}{Total\ Actual\ Positive}. \end{aligned} \tag{3.6}$$

With the precision and recall, the F1 score for binary classification is defined as

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \tag{3.7}$$

In order to obtain the F1 score for multi-class classification, i.e. for more than two classes, the F1 score was calculated for every class using the one vs. rest strategy, then averaged. The F1 score was calculated using the `f1_score` module[3] provided by the scikit-learn [146] package.

3.4.5 Algorithm Parameters

All procedures were performed with a initial population size of $N = 30$. The procedures ran until the end criterion, i.e. *not end of training*, was met, for which the cumulative number of iterations per generation was monitored and training was ended when it reached $N \times 40 = 1200$ times. That would ensure that the algorithm would run for at least 40 generations. For PBT, a step size of $t = 250$ was used between each exploitation and exploration. For PBT-DE, a training step size of $t = 242$ and fitness step size of $t_e = 8$ were used, which would add up to be equal to the number of steps performed by PBT for each member in each generation.

PBT was implemented using an exploitation factor of 0.2 and an exploration factors of $\sim [0.8, 1.2]$, as suggested by the initial paper [84].

PBT-DE was implemented using a mutation factor of $F = 0.2$ and a crossover rate of $CR = 0.8$, as suggested by the original authors [152].

[3]F1 score from scikit-learn at `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html`.

PBT-SHADE was implemented using the default parameters $H = 5$ and $r^{arc} = 2.0$, meaning that both M_F, M_{CR} in the historical memory was initialized to $\{0.5, 0.5, 0.5, 0.5, 0.5\}$, and the external archive size $|A|$ was equal to $r^{arc} \times N = 2.0 \times 30 = 60$. These parameters are described in more detail in Section 2.5.1. In addition, the procedure also requires a decimal percentage, p_{best}, of how many top performing members to consider when assigning $\mathbb{x}_{pbest}$. It was decided to use the default value of $p_{best} = 0.2$ in all experiments, meaning that only the 20% top performing members are considered for exploitation.

PBT-LSHADE was implemented using the same parameters as used for PBT-SHADE across all tests that were conducted. The L-SHADE-specific parameters used are derived from the default settings proposed by Tanabe and Fukunaga [188]. The targeted population size was set to $N^{min} = 4$, seeing that the DE/current-to-*p*best/1/bin mutation strategy requires a minimum of 4 members. The targeted number of fitness evaluations was set to $NFE^{max} = N \times 40 = 30 \times 40 = 1200$.

In order to obtain enough data to answer the research questions, each individual experiment was performed 50 consecutive times. Each test was performed in a Linux environment with two NVIDIA RTX 2080Ti GPUs with 10 989 GB of video memory each, where all member training and adaptation processes are distributed evenly among the GPU processors. All experiments used a batch size of $\mathcal{B} = 64$, meaning a single batch consisted of 64 images from the datasets, and each of these images were transferred from CPU memory to GPU memory in order to perform a single forward- and backward step in the neural network.

3.5 Implementation

When implementing the proposed procedures, it was discovered that there was a lack of purpose-built tools and technologies available that could facilitate the testing requirements. Therefore, we implemented a system for carrying out neural network training and hyperparameter adaption asynchronously in parallel on limited processing budget using a distributed queuing system. The system allowed for greater flexibility throughout the development, and ensured that each procedures were carried out using a shared framework employed by the system, inspired by a recently proposed PBT framework [115].

3.5.1 System Flow

Figure 3.4 is a flowchart of the main operations carried out by the system. The system flow process starts with defining each part that makes up the experiments: The hyperparameter search space, network model, network optimizer, evolution procedure (e.g. distinct, parameterized operations performed by PBT-LSHADE), the dataset and the sampled set divisions and an optional database connection (local or remote). With all required experiment parameters appropriately assigned, the controller process is started.

Workers

Before training and adaption can begin, the controller uses multiprocessing technologies to define a worker pool which spawns a specified number of processes, N_p (i.e. `n_jobs`),

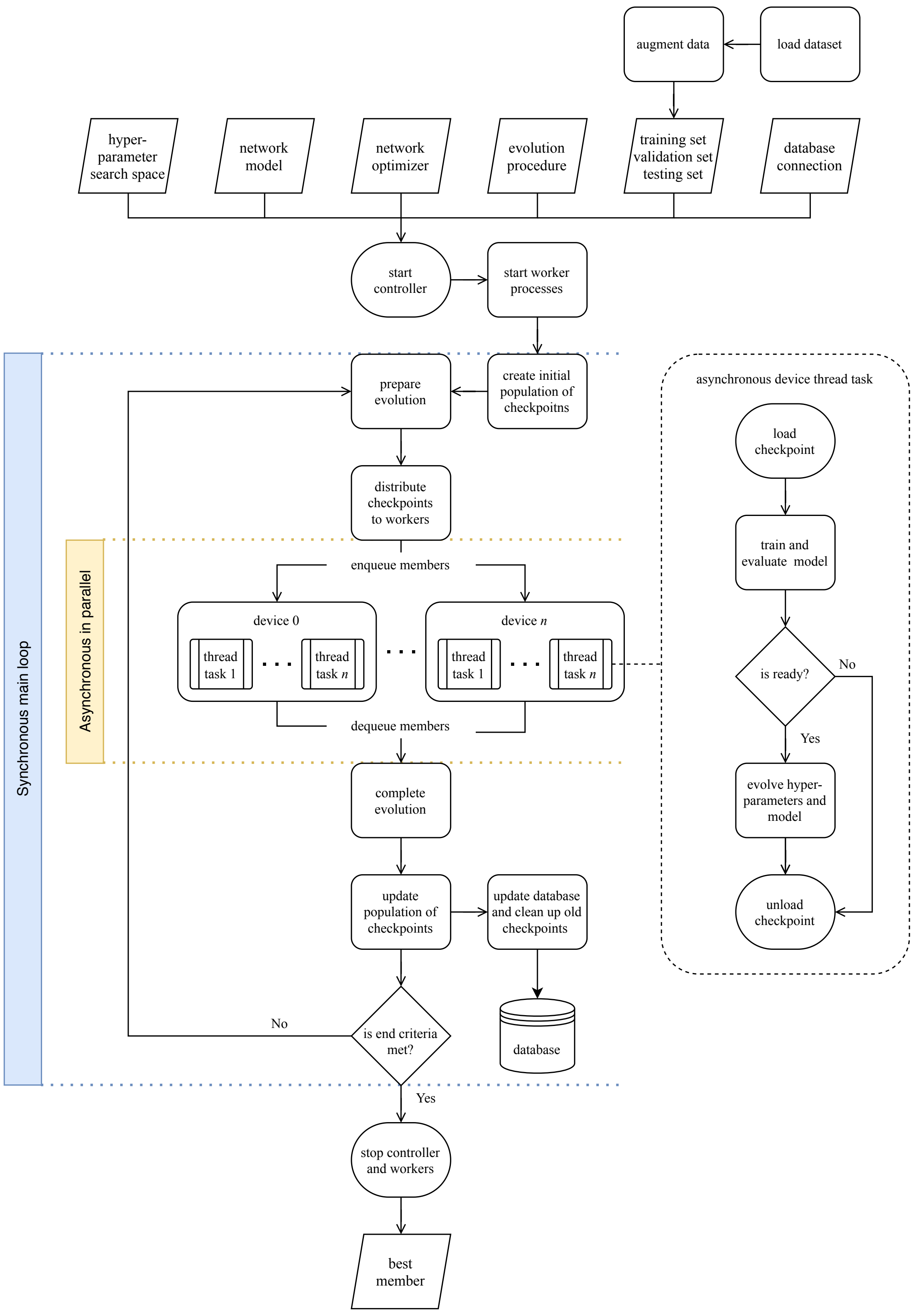


Figure 3.4: Flowchart of major system operations.

called *workers*. Each worker are assigned one and one only processing device (e.g. `gpu:0` for the 1th GPU, or `gpu:1` for the 2th GPU). The number of workers, N_p, should be set as large as possible for more distributed parallel processing, but is restricted to how much collective system memory or video memory that is available on the CPU or GPU processing units, respectively. When using GPU processing in this system, each worker must initialize the CUDA environment, which uses some of the available video memory.

In this thesis, 16 worker processes where employed, and the worker pool distributed the GPU devices evenly among these workers. A larger number of workers would have caused a CUDA out-of-memory error as only two GPUs where used in this thesis.

Population

In the next step of the system, all members in the population $\mathcal{P}$ of size N are initialized using the operations from each procedure that decides how hyperparameters are spawned in the search space. In this case, spawning process is equal for all processes. The population $\mathcal{P}$ is represented in system memory as a custom dictionary with a member identifier/key-value pair. Therefore, members are accessed and updated by unique identifier in order to ensure that the dictionary always consists of a distinct set of members.

At any step in the system, members are represented by a *checkpoint*, which is a data object that stores the current state of the member. The state is represented by several data components described in Table 3.5. These include the current state of the neural network model represented by the weight and bias matrices. The hyperparameters are represented by a special type of ordered dictionary that contains the current value, the lower and upper boundary, and the programmatic name of the hyperparameter. The optimizer parameters contains the current state of the network optimizer. In addition, the checkpoint state includes the number of steps and epochs in order to know where in the training process the member currently resides. The checkpoint also records the last score that was measured. Member checkpoints allow member-specific data to be constrained to single objects, which allows for better accessibility wherever members are referenced in the program.

Table 3.5: Data stored in a Checkpoint

Description	Variable Name	Data Type
Neural network weights/biases	`model_state`	Dictionary of Tensors
Optimizer parameters and state	`optimizer_state`	Dictionary of Tensors
Hyperparameters	`parameters`	Ordered Dictionary of Floats (*)
Steps	`steps`	Integer
Epochs	`epochs`	Integer
Loss	`loss`	Table of Floats

Note. (*) Hyperparameters are stored in a custom, ordered dictionary type that contains the upper and lower bounds.

After the members and workers are initialized, the main loop is entered. First, the system performs all the procedure operations that are required prior to generating the next generation of members. The PBT and PBT-DE procedures do not need this step as no prior operations are needed, but the PBT-SHADE and PBT-LSHADE procedures require it as they need to reset the historical memory H.

Workers and Asynchronous Processes

Next, the main loop enters the asynchronous part of the system. Here, the worker pool distributes the population $\mathcal{P}$ of N member checkpoints in chunks C of size $N_C = \lfloor N/N_p \rfloor$. These chunks are then queued across the worker processes along with the asynchronous process. If the number of chunks $|C|$ are not evenly divisible with the number of members N, one of the worker processes receives a chunk of extended size $N_C^* = n_c + 1$. As such, it is recommended to use a number of workers that ensures $N \mod N_p = 0$, if possible. Each worker processes the chunk C_i of member checkpoints with the asynchronous process in parallel using a pool of $|C_i|$ threads. When a thread task completes, the adapted member checkpoint gets immediately returned back to the worker pool via the return queue. In practice, this allows for members to be accessible the moment they are finished.

Post-Operations

After all members return from training and adaption and leaves the asynchronous portion of the system, any post operations required by the evolution procedure is performed at this point in the process. These are exclusively used by PBT-SHADE and PBT-LSHADE in order to update the historical memory. In addition, PBT-LSHADE performs linear population decay, which may shrink the population by one or more members.

Then, all remaining members are saved as the next $g + 1$-th generation, and each member checkpoint may also be saved to the database if a database provider is specified. Instead of relying on more complex database systems such as local SQL or cloud services, the database system we use creates and maintains a simple, accessible, local folder structure that can be easily navigated by the researcher during or after the training process has completed for analysis. In our case, the checkpoints of all members across all generations where saved, and these were used to construct data frames that would serve as the main source of data for the results, post-analysis and figures.

Depending on the size of the network model, checkpoint states could become quite large. In order to ensure that the system does not use up all accessible system memory, old member checkpoints are saved to a *database* and then removed from memory at the end of each generation. The system also optionally deletes old network model and optimizer states from old checkpoints in the database that is no longer of use in order to save space in database file system. Deletion is performed as the database can become quite large in size.

At last, the system checks to see whether the training adaptation process is finished by checking the end criteria. In our case, the end criteria is whether all members have performed t steps equal or above the specified max steps value, i.e. $t \geq N \times 40$. If true, the system exists the main loop, closes all workers processes and returns the best member. Otherwise, the system proceeds to generate the next $(g + 1)$-th generation of members.

3.5.2 Evolution Engines

The *evolution engine* is presented in the program as a framework for adapting members with evolutionary techniques. Several evolution engines where created for this system in order to carry out the testing, namely the exploit and explore method performed in the

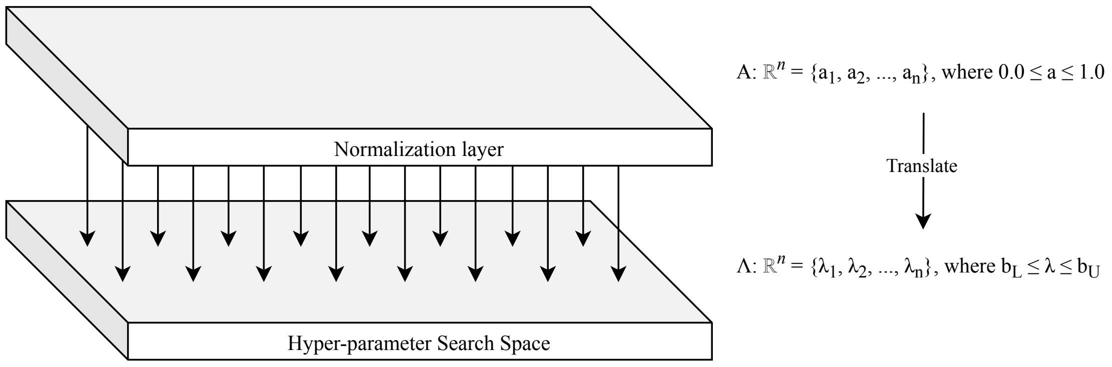

Normalization layer
Hyper-parameter Search Space
A: $\mathbb{R}^n = \{a_1, a_2, ..., a_n\}$, where $0.0 \leq a \leq 1.0$
Translate
Λ: $\mathbb{R}^n = \{\lambda_1, \lambda_2, ..., \lambda_n\}$, where $b_L \leq \lambda \leq b_U$

Figure 3.5: Hyperparameters mapped with a normalization layer.

original PBT procedure, as well as the mutation, crossover and selection strategies derived from DE, SHADE and L-SHADE. In order to get adaptive DE to work for multiprocessing, the external archive A and historical memory H were implemented as global registries that could be shared across all processes using a memory manager.

3.5.3 Controller

The component that orchestrate the entire evolution and training process is the *controller*. It decides when to call the evolver functions, and when to perform asynchronous training and evaluation of adapted members. It can also optionally do monitoring tasks like writing logs and creating real-time graphs using the TensorBoard [1] package[4]. More importantly, it makes sure to back up each generation to the database while it is running, and it performs garbage collection to remove any checkpoint states from the database that are no longer needed.

3.5.4 Hyperparameters

There are a few requirements for the implementation of Hyperparameters for neural networks. Fundamentally, each parameter is a value from either a continuous, discrete or categorical search space constrained to a predefined lower and upper boundary. While arithmetic operations come natural for both continuous and discrete numeric values, it poses a challenge for categorical parameters, e.g. strings. Therefore, categorical hyperparameters are not directly supported by DE, because it is a numeric optimizer where all hyperparameters must be able to be changed by arithmetic methods. While there could be implemented an exception for categorical values which uses methods like random selection to navigate the search space [115], this would alter the original heuristics from DE. Instead, it was decided to implement a layer on top of the underlying search space that maps numerical, continuous values to the underlying continuous, discrete or categorical values. This mapping layer is practically a normalized, continuous search space constrained from a lower limit of 0.0 to a upper limit of 1.0. The mapping of parameters are visualized in Figure 3.5.

For continuous and discrete hyperparameters, the mapping is a simple translation and

[4]Tensorboard is available at `https://www.tensorflow.org/tensorboard`.

defined as

$$x_{translated:Y} = y_{min} + (y_{max} - y_{min}) \times \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{3.8}$$

where x is the a value from the source space X to be mapped, x_{min} and x_{max} is the lower bound and upper bound of X, and y_{min} and y_{max} is the upper and lower bound of the target space Y. Because the normalized value must be a value between 0.0 and 1.0, the formula can be simplified as

$$\begin{aligned} \lambda_{translated:A} &= 0.0 + (1.0 - 0.0) \times \frac{\lambda - b_L}{b_U - b_L} \\ &= \frac{\lambda - \lambda_{min}}{b_U - b_L}, \end{aligned} \tag{3.9}$$

where b_L and b_U are the upper and lower bounds of λ.

On the other hand, categorical parameters are treated a bit different. The categorical search space is represented by a one-dimensional list with indices 0 to $l-1$, where l is the number of categorical parameters. The mapping layer cannot interface with the categorical parameters directly, but it can interface with the indices. Therefore, in a similar way to how discrete and continuous values are translated, the indices are translated too, and the categorical value can be retrieved.

For example, if the parameters are a discrete value from 0 to 100, a normalized value in the mapping layer of 0.50 would point to 50 in the underlying search space. Similarly, a normalized value of 0.734 would point to 73. For categorical values, where the underlying search space is represented by a list $\{string1, string2, string3\}$, a normalized value of 0.20 would point to index 0 which points to $string1$. Similarly, a normalized value of 0.67 would point to index 2 which points to $string3$.

The hyperparameter mapping layer is simply a view into the underlying search space, and allows us to limit arithmetic operations to the normalized value. This way has the advantage of enabling any type of hyperparameter to support any algorithm that generates new values based on the search space Λ, e.g. DE.

One downside of the mapping strategy is that categorical search spaces of just a few values does not reflect immediate change if the step size is too small. That is, if a step in the search space yields a normalized value that still points to the previous categorical value, no change has been made to the hyperparameter. This is especially noticeable for binary search spaces, e.g. True-False values where half of the search space points to one value.

3.5.5 Environment

The system was implemented in a Linux environment with Python[5] (version 3.7.6) – a popular high-level programming language that is widely used by both researchers and practitioners in the machine learning community. The processing is conducted mainly on multiple CUDA-enabled GPUs powered by the CUDA Toolkit[6] (version 10.2). Among many available machine learning packages in Python, we chose to use an open source machine learning framework called PyTorch[7] (version 1.5) to define and run deep learning

[5]Python is available at `https://www.python.org`.

[6]CUDA Toolkit is provided by NVIDIA at `https://developer.nvidia.com/cuda-downloads`.

[7]PyTorch is available at `https://pytorch.org`.

models. While Python is a high-level programming language, some of its components are implemented using low-level C-code (i.e. CPython), which makes it more efficient for certain tasks, despite its high-level abstraction.

Chapter 4

Results

This chapter presents the results and analysis obtained from running the PBT procedure (Section 3.1) against the PBT-DE, PBT-SHADE and PBT-LSHADE procedures (proposed in 3.2 and Section 3.3) on the aforementioned experiments established in Section 3.4. The result data are obtained by running 50 individual and consecutive tests with each procedure on the four different experiments that consists of the product of two different kinds of neural networks architectures, MLP and LeNet-5, as well as two different datasets for image classification, MNIST and Fashion-MNIST. The data presented is an accumulation of each best performing member in the last generation generated by the PBT, PBT-DE, PBT-SHADE and PBT-LSHADE procedures. Each of the best performing members are determined by the F1 score obtained from evaluating the member's model $\mathcal{M}$ on the validation set $\mathcal{X}^{(valid)}$ derived from the dataset that is currently used as measure. While multiple metric types are included in several figures and tables, the focus is primarily on the F1 score in order to determine how the procedures perform.

First, a comprehensive Welch ANOVA [205] and Post-Hoc analysis is presented in Section 4.2 in order to assess whether there is statistical significant difference between the tested procedures. Secondly, several figures and tables are presented in Section 4.3 on the train-, validation- and test scores obtained by calculating the CCE, F1 and accuracy across generations. Thirdly, some results are presented in regards to the procedures time complexity in Section 4.4. Next, the most common hyperparameter schedule trends are visualized in Section 4.5. Lastly, in order to understand how procedure parameters affect the predictive performance in relation to the time complexity, several tests are presented on different population sizes in Section 4.6.

4.1 Overview

Figure 4.1 shows the minimum, average and maximum F1 score obtained by the procedures on each experiment. Based on the results, it is clear that the PBT-SHADE and PBT-LSHADE procedures achieved the highest average on the MNIST and Fashion-MNIST datasets, each respectively. The highest recorded maximum F1 score is obtained primarily by the PBT-LSHADE procedure, except from the experiment that involved the MNIST dataset with the MLP neural network architecture. In this case, the maximum F1 score was obtained by the PBT-DE procedure. Overall, it was determined that the PBT-DE, PBT-SHADE and PBT-LSHADE procedures outperform the PBT procedure on all experiments. When that has been said, please refer to Section 4.3 for more in-depth

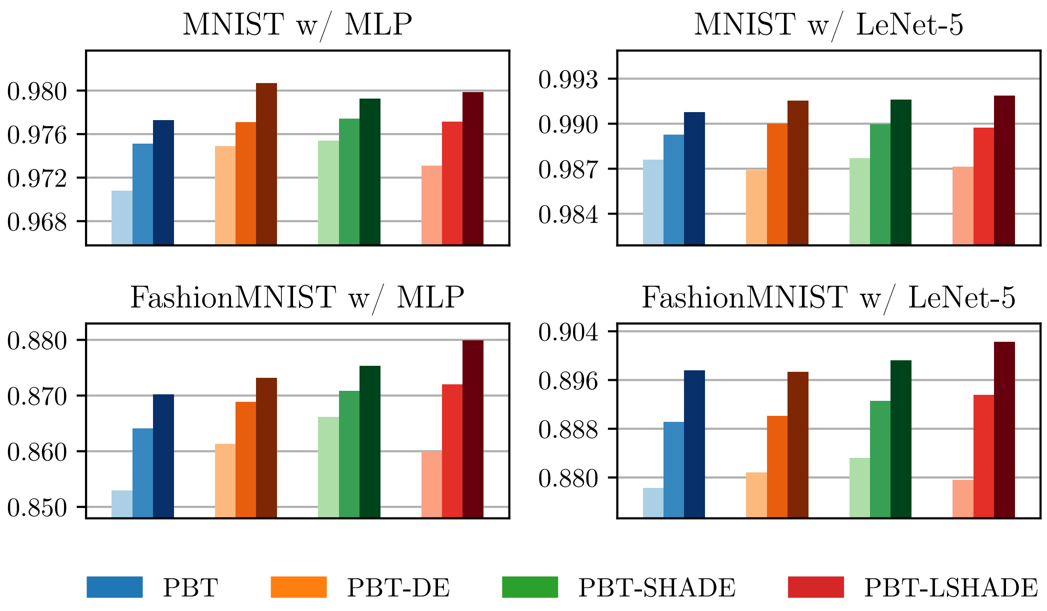


Figure 4.1: The F1 scores of each procedure on four different experiments. Bright color is minimum, medium color is average, and dark color is maximum.

analysis of the predictive performances of the individual procedures.

4.2 Welch ANOVA Analysis

Because multiple procedures (or groups) was measured, the Welch ANOVA (Analysis of Variance) [205] method was used in order to determine whether there is a statistical significant difference found in the procedures. For all tests, the dependent variable was the F1 score obtained from using the test dataset division $\mathcal{X}^{(test)}$ on the final model $\mathcal{M}$. A significance level of $\alpha = 0.01$ was used for all tests, and is the probability of rejecting the null hypothesis when it is true. The hypotheses tested by the Welch ANOVA method is defined as

$$\begin{aligned} H_0 &: \bar{x}_1 = \bar{x}_2 = \bar{x}_3 = \cdots = \bar{x}_k \\ H_1 &: \text{At least one } \bar{x}_i \text{ differ,} \end{aligned} \tag{4.1}$$

where the null hypothesis H_0 was accepted if all F1 means for each procedure was equal, otherwise the hypothesis H_1 was accepted if at least one of the means are different. The variable k is the number of procedures, which was $k = 4$ for this case.

4.2.1 Assumption Check

In order for the Welch ANOVA tests to be considered valid, it was ensured that all test data are mutually exclusive, and there are no repeated measures. Furthermore, the data was also tested for normality by using the Shapiro–Wilk [167] test. The null hypothesis

Table 4.1: The Shapiro-Wilk Test for Normality

Dataset	Model	Procedure	statistic	p-value
MNIST	MLP	PBT	0.954811	0.053948
		PBT-DE	0.975883	0.394113
		PBT-SHADE	0.976781	0.425588
		PBT-LSHADE	0.981376	0.61126
	LeNet-5	PBT	0.987843	0.883427
		PBT-DE	0.953751	0.0487421
		PBT-SHADE	0.961217	0.0999691
		PBT-LSHADE	0.977046	0.435194
Fashion MNIST	MLP	PBT	0.966956	0.173563
		PBT-DE	0.968978	0.210258
		PBT-SHADE	0.978189	0.478392
		PBT-LSHADE	0.964433	0.136303
	LeNet-5	PBT	0.980565	0.576026
		PBT-DE	0.970892	0.251519
		PBT-SHADE	0.96304	0.119192
		PBT-LSHADE	0.945705	0.0227907

Note. The test is used to assess whether the data is normally distributed. If the p-value is less than the chosen significance level $\alpha = 0.01$, the data is not normally distributed.

of a Shapiro-Wilk test is that the data is normally distributed. If the p-value generated with the Shapiro-Wilk test is less than the alpha level, $\alpha = 0.01$, i.e. $p < 0.01$, the null hypothesis is rejected. The results from the Shapiro-Wilk test are presented in 4.1 for each experiment, and it was clear that all p-values are greater than the alpha value. According to these findings, it can be determine with 99% certainty that the data was normalized. As an extra reassurance, a probability plot of the residuals for each individual group was included in Figure 4.2. It was clear that the ordered values follow along the diagonal line across the theoretical quantiles, indicating that the graphical testing supported the statistical findings that the data was normalized.

The reason for why the Welch ANOVA test was used instead of the traditional One-Way ANOVA test was because the method does not assume there is homogeneity of variances in the data. Using the Levene's method [113] for assessing equality of variances on the dependent variable for each experiment, it was observed that all experiments except for the MNIST dataset with the LeNet-5 architecture passed the test (see Table 4.2), meaning that any findings provided by the one-way ANOVA test could not be statistically trusted for the results obtained using the MNIST dataset with the LeNet-5 architecture.

4.2.2 Results

Table 4.3 presents the Welch ANOVA test results[1]. It is clear that, for all experiments, the p-values are very small, and ultimately less than the pre-established significance level, $\alpha = 0.01$, meaning it can be said with a 99% certainty that there was a statistically significant difference between at least two of the procedures for every experiment. The

[1]The Welch ANOVA test was carried out using a publicly available Python package, `https://pingouin-stats.org/generated/pingouin.welch_anova.html`

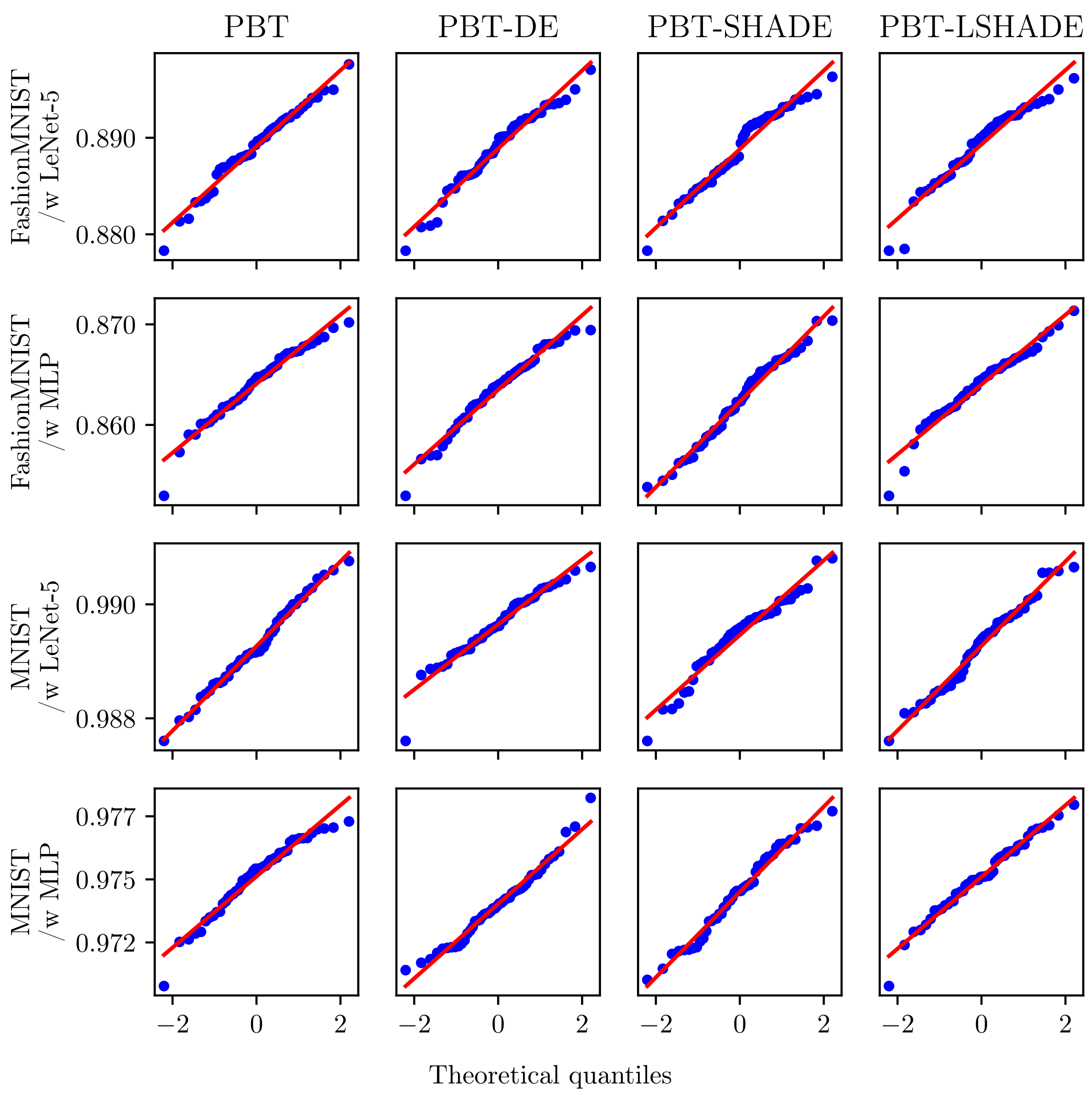


Figure 4.2: A probability plot of the residuals, where normality is indicated by the red diagonal.

Table 4.2: The Levene's Test for Homogeneity of Variance

Dataset	Model	statistic	p-value
MNIST	MLP	1.40579	0.242347
	LeNet-5	4.53437	0.00425292
Fashion MNIST	MLP	2.74165	0.0444369
	LeNet-5	1.8056	0.147454

Note. The test is used to assess whether the data is equal in variance. If the p-value is less than the chosen significance level $\alpha = 0.01$, it means there is no homogeneity of variance found in the data.

Table 4.3: The Welch ANOVA Test

Dataset	Model	d.f.N.	d.f.D.	F-value	p-value
MNIST	MLP	3	107.66085	32.17127	6.32×10^{-15}
	LeNet-5	3	107.85942	9.23710	1.72×10^{-5}
Fashion MNIST	MLP	3	107.18582	54.66338	1.62×10^{-21}
	LeNet-5	3	108.23603	12.51375	4.37×10^{-7}

Note. If the p-value is below the significance level $\alpha = 0.01$, it means there is a statistically significant difference between the procedures.

Welch ANOVA results alone does not specify which procedures that are statistically and significantly different, so the tests are naturally followed up with a Post-Hoc analysis.

Post-Hoc Analysis

The Pairwise Games-Howell [59] Post-Hoc test method was used for each experiment in order to assess which procedure pairs that were statistically and significantly different. While the Games-Howell test is similar to the Tukey's Honestly Significant Difference (Tukey's HSD) [192] Post-Hoc test, it is more robust to heterogeneity of variances, and so it is optimal after a Welch ANOVA test. Table 4.4 displays the results from the Games-Howell test[2].

From the data presented, it can be stated with a 99% certainty that there was a statistical significant difference between the PBT procedure compared with the PBT-DE, PBT-SHADE and PBT-LSHADE procedures from the test scores obtained from using the MNIST dataset and MLP neural network architecture. On the same experiment, it cannot be stated with 99% certainty that there was a statistical significant difference between the PBT-DE, PBT-SHADE and PBT-LSHADE procedure.

On the experiment with the MNIST dataset with the LeNet-5 architecture, it can be stated with 99% certainty that there was a statistical significant difference between the PBT and PBT-DE procedures, as well as the PBT and PBT-LSHADE procedure. For all other procedure pairs, the null hypothesis was rejected with a 99% certainty.

On the experiment with the Fashion-MNIST dataset with the MLP architecture, it can be stated with 99% certainty that there was a statistical significant difference between the PBT procedure and the PBT-DE, PBT and PBT-SHADE, PBT-LSHADE procedures. Moreover, there was also a statistical significant difference between the PBT-DE and PBT-SHADE procedures, and the PBT-DE and PBT-LSHADE procedures. The only procedure pair that failed the test, was the PBT-SHADE and PBT-LSHADE procedures.

Lastly, it can be said with 99% certainty that there was a statistical significant difference between the PBT and PBT-SHADE procedures, and the PBT and PBT-LSHADE procedures on the experiment with the Fashion-MNIST dataset and the LeNet-5 architecture. Furthermore, on the same experiment, it can be stated with 99% certainty that there was a statistical significant difference between the PBT-DE and PBT-SHADE procedure, as well as the PBT-DE and PBT-LSHADE procedures. For all other procedure pairs, it cannot be stated that there was a statistical significant difference.

[2]The Games-Howell test was carried out using a publicly available Python package, `https://pingouin-stats.org/generated/pingouin.pairwise_gameshowell.html`

Table 4.4: Pairwise Games-Howell Post-Hoc Test

Dataset	Model	Procedures	S.E.	T-value	d.f.	p-value
MNIST	MLP	PBT & PBT-DE	0.00019	-7.48914	96.44685	0.00100
		PBT & PBT-SHADE	0.00019	-7.42833	97.57683	0.00100
		PBT & PBT-LSHADE	0.00017	-9.54680	86.49154	0.00100
		PBT-DE & PBT-SHADE	0.00018	-0.15442	97.63001	0.90000
		PBT-DE & PBT-LSHADE	0.00016	-1.41861	92.26055	0.48795
		PBT-SHADE & PBT-LSHADE	0.00016	1.19252	89.59553	0.61521
	LeNet-5	PBT & PBT-DE	0.00011	-4.47966	95.15235	0.00100
		PBT & PBT-SHADE	0.00013	-2.36873	83.35057	0.08563
		PBT & PBT-LSHADE	0.00011	-4.48197	97.13978	0.00100
		PBT-DE & PBT-SHADE	0.00014	1.30806	91.54714	0.55055
		PBT-DE & PBT-LSHADE	0.00012	0.18933	97.37374	0.90000
		PBT-SHADE & PBT-LSHADE	0.00014	1.18470	87.92620	0.61958
Fashion MNIST	MLP	PBT & PBT-DE	0.00043	-7.87083	92.22133	0.00100
		PBT & PBT-SHADE	0.00051	-11.03481	97.14250	0.00100
		PBT & PBT-LSHADE	0.00041	-11.58976	86.27826	0.00100
		PBT-DE & PBT-SHADE	0.00046	-4.79091	88.03916	0.00100
		PBT-DE & PBT-LSHADE	0.00035	-3.93764	96.36414	0.00100
		PBT-SHADE & PBT-LSHADE	0.00044	-1.84570	81.67583	0.25400
	LeNet-5	PBT & PBT-DE	0.00052	-1.36620	97.04208	0.51802
		PBT & PBT-SHADE	0.00063	-5.03446	93.09950	0.00100
		PBT & PBT-LSHADE	0.00053	-4.62757	97.33800	0.00100
		PBT-DE & PBT-SHADE	0.00060	-4.03364	88.82178	0.00100
		PBT-DE & PBT-LSHADE	0.00050	-3.44474	97.97160	0.00350
		PBT-SHADE & PBT-LSHADE	0.00061	-1.16478	89.60407	0.63074

Note. If the p-value is below the significance level $\alpha = 0.01$, it means there is a statistically significant difference between the procedures.

4.3 Individual Performance Comparisons

This section presents the individual performances between the procedures on from all experiments. The following sub-sections is first divided on the different datasets, MNIST and Fashion-MNIST, and then on the different neural network architectures, MLP and LeNet-5. The result data obtained from each experiment is reported in every metric type for each dataset divisions. Of the divisions, the testing set $\mathcal{X}^{(test)}$ was considered the most important benchmark for assessing the predictive performance between the different procedures. Furthermore, the training- and validation sets are included because the difference between the scores obtained from each dataset division over time says something about how well the neural network model performs in terms of generalization.

In order to determine how well each optimization procedure performs across the entire generation span on unseen data, performance on the testing set $\mathcal{X}^{(test)}$ from the dataset was recorded at for each member at each step in the training. It is important to note that the results obtained from running these tests were not used by any of the procedures in any way, as this would have gone against good algorithmic testing practice where the unseen data is supposed to be unknown to the procedure. Otherwise, the final results would be skewed and biased. Furthermore, as an extra measure to reduce any potential researcher bias, these tests were not performed during development of the procedures; the result data obtained across generations was used purely for analytical purposes after the procedures had been tested.

Lastly, an overall numeric representation of the test data is included. The CCE score, F1 score and accuracy are presented in Table 4.5. Table 4.6 and Table 4.7, each respectively. Note that the F1 testing score for each dataset and network architecture was of most relevance when assessing the performance of each procedure. For convenience, the deemed best values are highlighted in bold.

4.3.1 Results from the MNIST Dataset

This section presents the results obtained from running the MLP and LeNet-5 architectures on the MNIST dataset. The results are first and foremost visualized through box plots, which is a type of plot used for representing groups of numerical data through their quartiles. Each figure consists of 9 box plots; one box plot for each product of the different evaluation metrics and dataset divisions. The first row shows the CCE, the second row shows the F1 score and third row shows the accuracy for all three dataset divisions.

Results with the MLP Architecture

Figure 4.3 depicts the box plots of the results from testing with the MLP architecture. It is clear that when viewing the test scores, the PBT-DE, PBT-SHADE and PBT-LSHADE procedures outperformed the PBT procedure in terms of highest (best) F1 mean obtained. Of the four, the PBT-LSHADE procedure achieved the highest recorded F1 score, but its variance was considerable higher than the rest, leading it to also achieving the second-to-lowest F1 score. Furthermore, the PBT-SHADE demonstrated the lowest amount of variance, meaning that the results it produced was more consistent compared to the rest of the procedures. In addition, PBT-SHADE also achieved the highest F1 mean of all the procedures at 0.9774, which lead us to determine that PBT-SHADE was the most consistently highest performing procedure on the MNIST dataset with the MLP architecture in

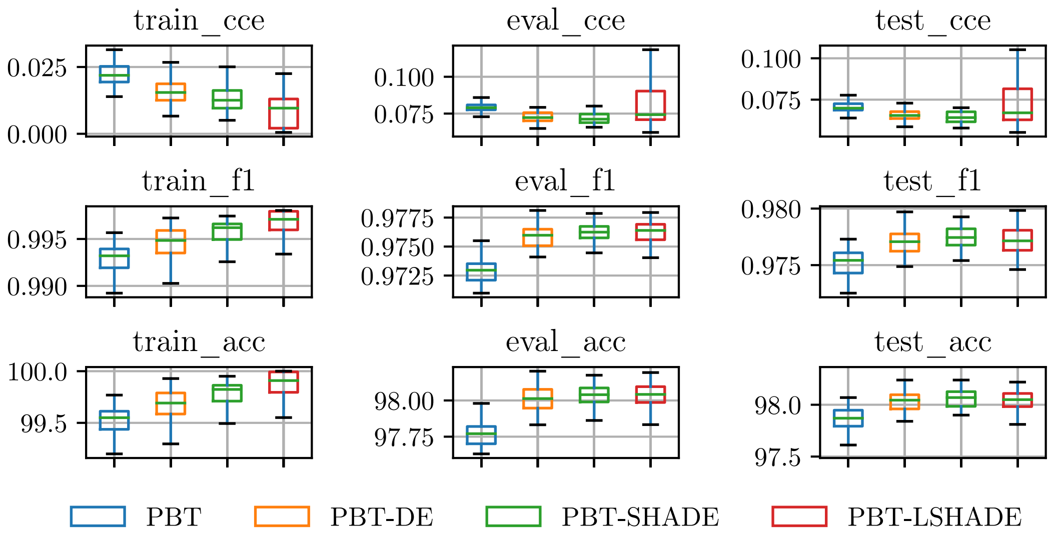


Figure 4.3: Box plot comparisons between the procedures from 50 consecutive tests performed with the MNIST dataset using MLP.

terms of F1 score.

In order to understand how each procedure performed over time, the metric scores obtained at every step across the duration of the procedure are displayed in Figure 4.4. The data points at each step represents the average score between the 50 tests that have been performed. The data is visualized over time (i.e. steps) with a line chart as a mean to view how well each procedure performed across generations. This is especially important as each procedure was tested with the same number of total accumulated steps, which allowed the PBT-LSHADE procedure to produce more than twice the amount of generations. In order to provide a good visualization of the data, the x-axis is compressed by a factor of 50% for the last generations generated by PBT-LSHADE. The plots are first divided by the metric types CCE score, f1 score and accuracy score in groups of three, where each row within the groups contain a different dataset division.

In Figure 4.4, it was observed that on all metrics, PBT performed better in the first 3 000 steps, but eventually stopped improving and was overtaken by the other procedures in the range between 3000 and 5000 steps. Out of the three DE-based procedures, PBT-DE displayed the lowest F1 score in the first 9000 steps, but finally performs similar to PBT-SHADE and PBT-LSHADE.

It is clear that when viewing how PBT-LSHADE performs over time, the procedure did not benefit too much from the extra steps it managed with the same budget. Moreover, PBT-LSHADE seemed to overfit the training set, which was reflected in the CCE score when comparing the training set with the evaluation and testing set. Despite the fact that the procedures was given control over weight decay as a mean to mitigate overfitting, the regularization technique seemed to not prevent overfitting due to excessive training. One explanation for why the procedures failed to reduce overfitting could be explained by the F1 validation score, which seemed to be not affected by the overfitting. Whether reducing overfitting or not would provide better results was not clear, but it could be interesting to see the outcome if a metric that is more capable of detecting overfitting was used for

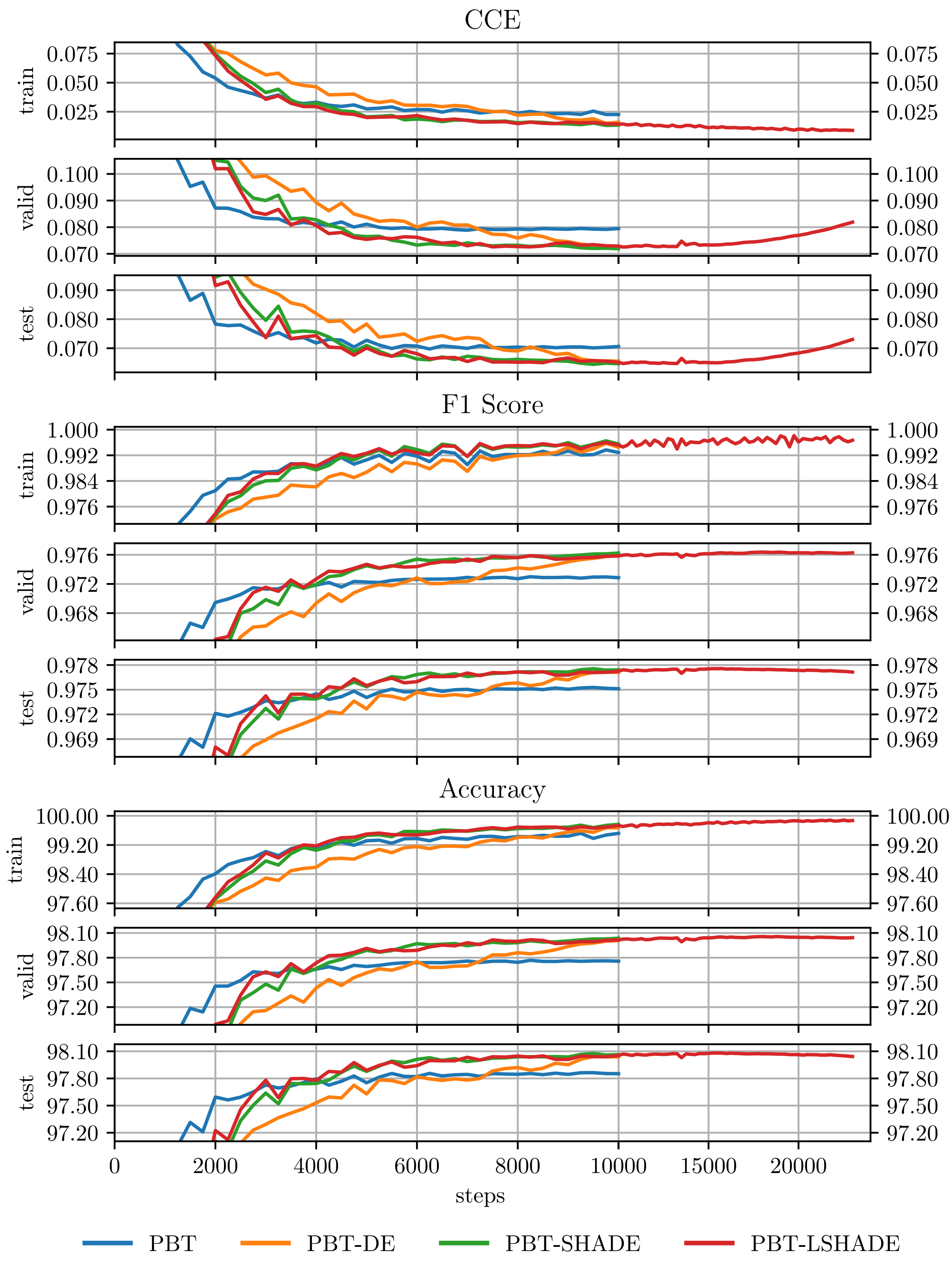


Figure 4.4: Line chart comparisons between the procedures from the average of 50 consecutive tests performed on the MNIST dataset using MLP.

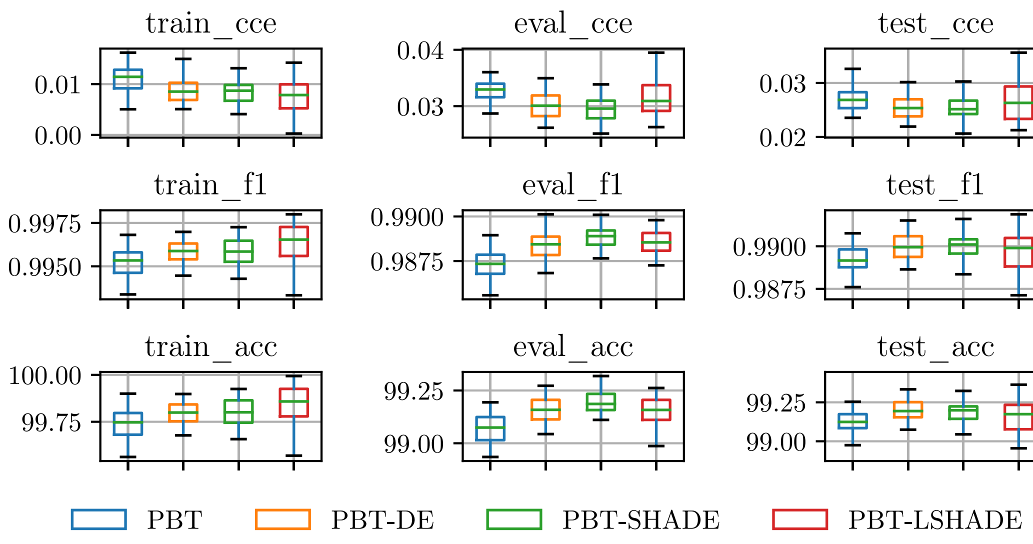


Figure 4.5: Box plot comparisons between the procedures from 50 consecutive tests performed with the MNIST dataset using LeNet-5.

validation, like the CCE. When that has been said, the F1 test score does show some degradation in the range between 15 000 and 23 500 steps, which means the score would be better if the training was ended sooner, around the 15 000 step mark.

It is also important to note that despite PBT-LSHADE using linear population decay, which reduces the number of members linearly over time, it did not seem to affect the predictive performance when compared to PBT-SHADE, which is the same procedure, but without linear population decay. That would suggest that a lower population size $N < 30$ could provide similar results, which would save the DE-procedures a generous amount of time if the number of GPUs or CPUs used are lower than the population size, which is true in our case.

Results with the LeNet-5 Architecture

The results from testing with the LeNet-5 architecture is shown in Figure 4.5. Similar to the results found from testing with the MLP architecture, PBT was outperformed once again by the PBT-DE, PBT-SHADE and PBT-LSHADE procedures on the averagely obtained F1 score. The L-SHADE obtained the highest F1 score this time as well, but it also had the highest amount of variance as well. In addition, PBT-DE demonstrated to achieve the lowest amount of variance, and while it did not achieve one of the highest F1 scores, it obtained the highest F1 mean at 0.9900. Considering what have been learned from the results, it was determined that PBT-DE displayed the best predictive performance on the MNIST dataset with the LeNet-5 architecture because of its consistent high F1 scores.

Figure 4.6 displays the obtained average score data from each procedure visualized over time. When inspecting the F1 test score, it was clear that the PBT procedure performed better than the other procedures for the first 4000 or so steps, but was eventually overtaken by the PBT-LSHADE procedure, and later by both the PBT-DE and PBT-SHADE procedures. In contrast with the results obtained from using the MLP architecture, PBT-DE performed considerably worse the first 8000 steps when compared to the rest, but eventually catch up before the 10 000 step mark.

As seen in the results obtained from testing with the MNIST dataset and MLP architecture, the extra steps performed by L-SHADE lead to overfitting the training set, which is reflected in the range between the training and validation CCE score. However, the amount of overfitting seemed to be lower than the other tests, and there was a noticeable degradation in both the F1 validation score and F1 testing score (as well as the accuracy equivalents). The reduction in overfitting could have happened because of the change in neural network architecture, but there is also a chance that the procedures were more successful in reducing some of the overfitting by optimizing the weight decay. Because of the overfitting, PBT-LSHADE procedure may have performed worse than both the PBT-DE and PBT-SHADE procedures. If the PBT-LSHADE procedure was stopped earlier, it would have achieved better results, and better time complexity as well.

4.3.2 Results from the Fashion-MNIST Dataset

This section presents the results obtained from running the MLP and LeNet-5 architectures on the Fashion-MNIST dataset for classifying human clothing from images. Figure 4.7 displays the box plots that show the results from testing with the MLP architecture. Each of the rows shows the CCE, F1 score and accuracy for all three dataset divisions, respectively.

In Figure 4.7, it is clear that the PBT-DE, PBT-SHADE and PBT-LSHADE procedures outperformed the PBT procedure on all tested metrics. Of the procedures, PBT-LSHADE displayed the highest amount of variance, which may have lead it to obtain both the best and worst recorded F1 score. While PBT-LSHADE displayed higher variance, it still obtained the highest F1 mean, and PBT-SHADE achieved the second-highest F1 mean.

Figure 4.8 displays the average metric scores obtained at every step, divided by each metric and dataset division. In the first steps, PBT performed better than the other procedures, but was eventually overtaken by all DE-based procedures in the range from 4 000 to 7 000 steps. For the first 10 000 steps, the PBT-SHADE procedure peformed better than the other procedures on all metrics.

From Figure 4.8, it was noticed that the number of members in the population seemed to have an effect on the predictive performance of the network model in all cases, something that cannot be noticed in earlier results from the MNIST dataset in Figure 4.4. It was clear that the PBT-LSHADE procedure did not perform as well as both the PBT-DE and PBT-SHADE procedure for the first 10 000 steps. As the PBT-SHADE and PBT-LSHADE procedures are essentially the same procedures except for the linear population decay performed in PBT-LSHADE, the only explanation for the performance difference must be correlated with difference in population size over time.

When that has been said, the aforementioned difference was not reflected in the final results because PBT-LSHADE ran an additional 13 500 steps and managed to overtake on both the F1 and accuracy metric. Also, similar to other tests, the PBT-LSHADE procedure displayed signs of overfitting as seen in the CCE metric charts in Figure 4.8, but that is hardly noticeable in the other metrics.

In Figure 4.9, we display the box plots of the results from training and testing the Fashion-MNIST dataset with the LeNet-5 architecture. Similar to earlier examples, the PBT comes out the weakest of the four procedures on the average test F1 score. Out of the four procedures, the PBT-LSHADE procedure displays the highest amount of variance and

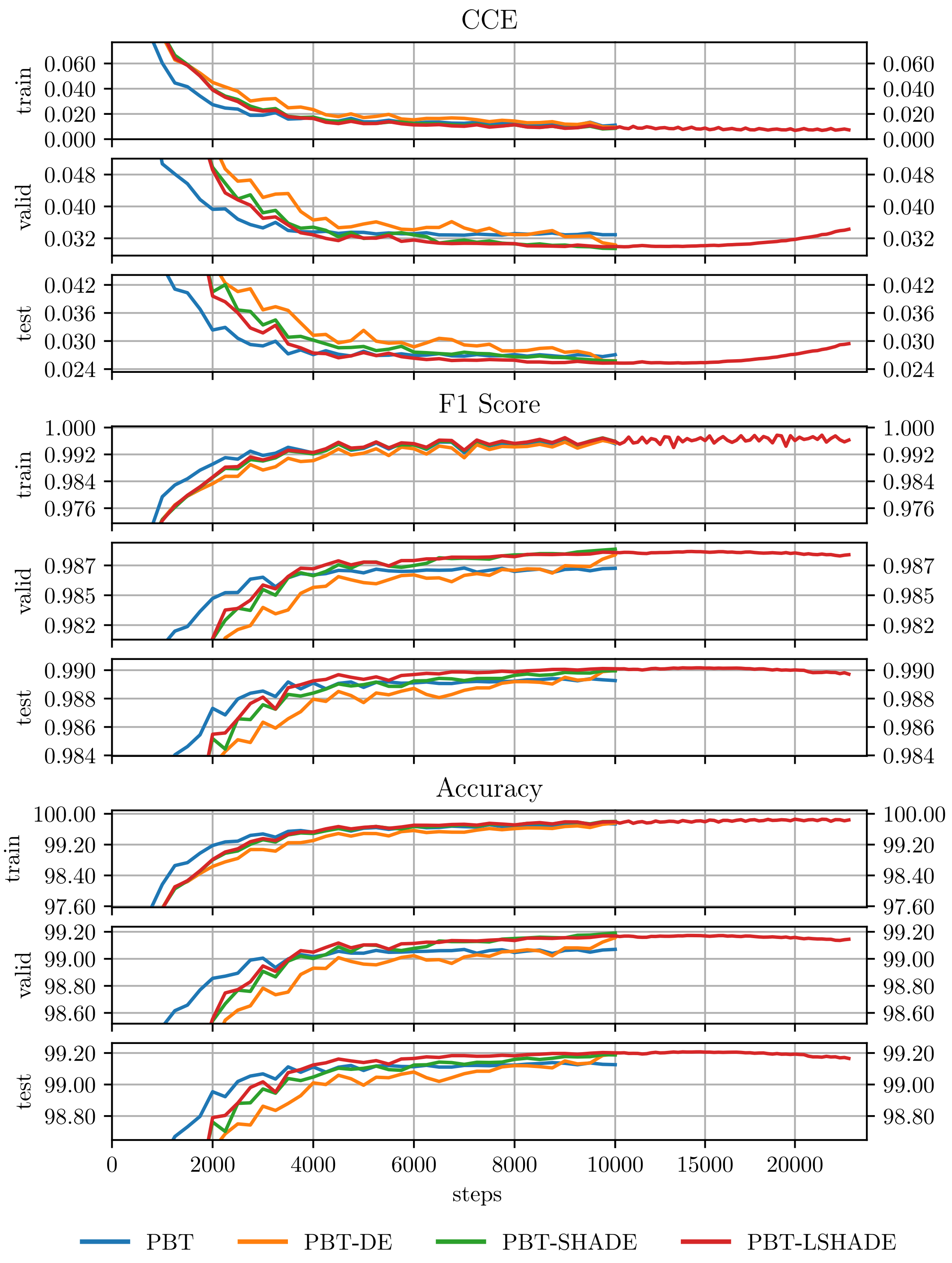


Figure 4.6: Line chart comparisons between the procedures from the average of 50 consecutive tests performed on the MNIST dataset using LeNet-5.

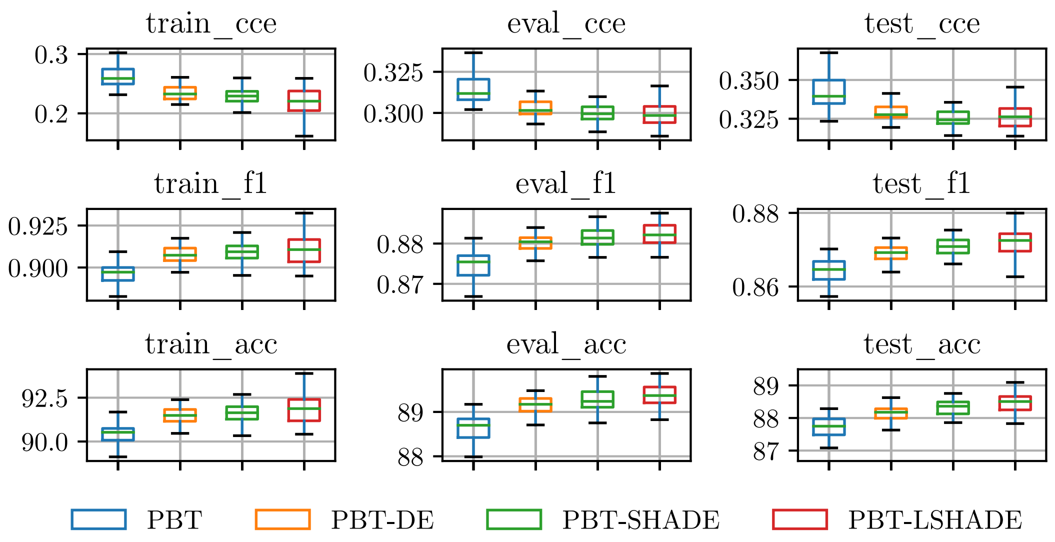


Figure 4.7: Box plot comparisons between the procedures from 50 consecutive tests performed with the Fashion-MNIST dataset using MLP.

obtains both the lowest and highest registered F1 score, while the PBT procedure displays the lowest amount of variance. Even so, the PBT-LSHADE procedure still obtains the highest average F1 test score, and PBT-LSHADE achieves the second highest average F1 test score.

When measuring predictive performance over time as shown in Figure 4.10, it is clear that the PBT-LSHADE procedure performed strongly the first 10 000 steps when compared to the other procedures, but because the PBT-LSHADE procedure was able to train for more than twice the amount of steps, it managed to overtake the other procedures and come out on top. The PBT procedure performed strong the first 6 000 steps, but was eventually overtaken by the other procedures.

Similarly to the results obtained from using the MLP neural architecture (Figure 4.8), it became evident that linear population decay over time hurt the predictive performance of the PBT-LSHADE procedure when compared to the PBT-SHADE procedure. Furthermore, it was clear that overfitting occurred with this architecture as well, as seen in the range from 17 000 to the end, both in the CCE validation curve and the final gap between the train CCE score and validation CCE score. However, overfitting did not seem to hurt predictive performance when inspecting the F1 and accuracy metric. When that has been said, it was noticed that the difference between the training scores and validation scores became greater after the 10 000 step mark.

4.4 Time complexity

For neural network hyperparameter adaption, time complexity describes the relation between the number of steps (or epochs), the population size and time. In order to say something about the time complexity of the PBT, PBT-DE, PBT-SHADE and PBT-LSHADE procedures, time was measured by registering the amount of time taken to perform three distinct tasks for each member in each generation. The tasks performed by each member

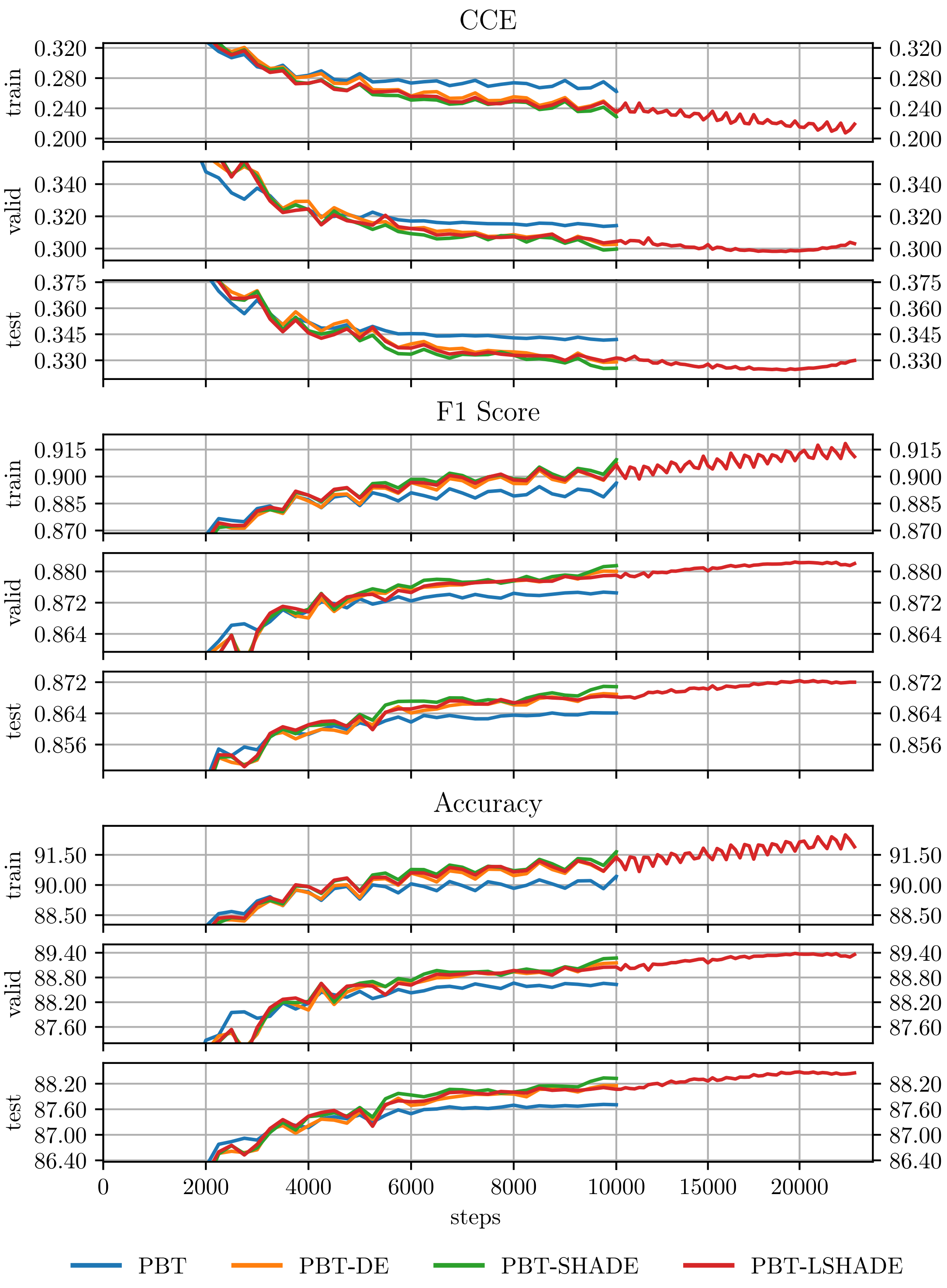


Figure 4.8: Line chart comparisons between the procedures from the average of 50 consecutive tests performed on the Fashion-MNIST dataset using MLP.

Table 4.5: Performance Statistics in CCE Score

Dataset	Model	Set	Algorithm	median	mean	std	min	max
MNIST	MLP	train	PBT	0.0219	0.0226	0.0052	0.0139	0.0416
			PBT-DE	0.0155	0.0160	0.0054	0.0066	0.0308
			PBT-SHADE	0.0126	0.0134	**0.0048**	0.0051	0.0262
			PBT-LSHADE	**0.0096**	**0.0089**	0.0065	**0.0005**	**0.0225**
		valid	PBT	0.0790	0.0795	**0.0031**	0.0729	0.0898
			PBT-DE	0.0723	0.0727	0.0036	0.0649	**0.0792**
			PBT-SHADE	**0.0712**	**0.0719**	0.0041	0.0657	0.0865
			PBT-LSHADE	0.0743	0.0819	0.0169	**0.0622**	0.1442
		test	PBT	0.0699	0.0706	0.0034	0.0639	0.0843
			PBT-DE	0.0654	0.0655	**0.0032**	0.0586	**0.0729**
			PBT-SHADE	**0.0641**	**0.0647**	0.0040	0.0578	0.0792
			PBT-LSHADE	0.0670	0.0730	0.0155	**0.0552**	0.1331
	LeNet-5	train	PBT	0.0114	0.0111	0.0031	0.0050	0.0233
			PBT-DE	0.0085	0.0095	0.0042	0.0051	0.0264
			PBT-SHADE	0.0087	0.0086	**0.0025**	0.0041	0.0165
			PBT-LSHADE	**0.0079**	**0.0073**	0.0039	**0.0003**	**0.0142**
		valid	PBT	0.0330	0.0329	**0.0020**	0.0287	0.0410
			PBT-DE	0.0301	0.0304	0.0026	0.0262	0.0408
			PBT-SHADE	**0.0296**	**0.0294**	0.0024	**0.0251**	**0.0358**
			PBT-LSHADE	0.0309	0.0343	0.0094	0.0263	0.0654
		test	PBT	0.0269	0.0271	**0.0023**	0.0235	0.0334
			PBT-DE	0.0253	0.0258	0.0030	0.0219	0.0401
			PBT-SHADE	**0.0251**	**0.0257**	0.0026	**0.0206**	**0.0331**
			PBT-LSHADE	0.0263	0.0294	0.0093	0.0213	0.0627
Fashion MNIST	MLP	train	PBT	0.2588	0.2623	0.0155	0.2314	0.3022
			PBT-DE	0.2326	0.2346	**0.0121**	0.2149	0.2607
			PBT-SHADE	0.2292	0.2288	0.0136	0.2014	0.2597
			PBT-LSHADE	**0.2205**	**0.2190**	0.0234	**0.1616**	**0.2590**
		valid	PBT	0.3118	0.3142	0.0079	0.3020	0.3366
			PBT-DE	0.3015	0.3026	**0.0049**	0.2933	0.3133
			PBT-SHADE	0.2995	**0.2997**	0.0053	0.2885	**0.3099**
			PBT-LSHADE	**0.2985**	0.3030	0.0183	**0.2859**	0.3954
		test	PBT	0.3395	0.3420	0.0094	0.3234	0.3675
			PBT-DE	0.3276	0.3291	0.0054	0.3194	0.3413
			PBT-SHADE	**0.3243**	**0.3255**	**0.0051**	0.3142	**0.3356**
			PBT-LSHADE	0.3263	0.3299	0.0188	**0.3138**	0.4100
	LeNet-5	train	PBT	0.1973	0.1968	**0.0117**	0.1711	0.2244
			PBT-DE	0.1928	0.1925	0.0171	0.1520	0.2364
			PBT-SHADE	0.1790	0.1775	0.0132	0.1388	**0.1973**
			PBT-LSHADE	**0.1659**	**0.1655**	0.0206	**0.1257**	0.2063
		valid	PBT	0.2661	0.2664	0.0068	0.2520	0.2854
			PBT-DE	0.2616	0.2633	0.0067	0.2517	0.2824
			PBT-SHADE	0.2611	**0.2608**	**0.0059**	**0.2470**	**0.2781**
			PBT-LSHADE	**0.2604**	0.2700	0.0231	0.2476	0.3540
		test	PBT	0.2766	0.2776	0.0082	0.2647	0.3038
			PBT-DE	0.2745	0.2752	0.0083	0.2601	0.2955
			PBT-SHADE	**0.2698**	**0.2703**	**0.0067**	0.2569	**0.2819**
			PBT-LSHADE	0.2710	0.2800	0.0257	**0.2534**	0.3712

Note. The best values are highlighted in bold.

Table 4.6: Performance Statistics in F1 Score

Dataset	Model	Set	Algorithm	median	mean	std	min	max
MNIST	MLP	train	PBT	0.9932	0.9929	0.0019	0.9854	0.9957
			PBT-DE	0.9948	0.9945	0.0019	0.9890	0.9972
			PBT-SHADE	0.9962	0.9955	0.0017	0.9907	0.9974
			PBT-LSHADE	**0.9971**	**0.9967**	**0.0015**	**0.9921**	**0.9980**
		valid	PBT	0.9730	0.9729	0.0012	0.9690	0.9755
			PBT-DE	0.9760	0.9759	0.0012	0.9714	0.9781
			PBT-SHADE	0.9762	0.9762	**0.0007**	**0.9745**	0.9779
			PBT-LSHADE	**0.9764**	**0.9763**	0.0010	0.9740	**0.9795**
		test	PBT	0.9754	0.9751	0.0014	0.9708	0.9773
			PBT-DE	0.9771	0.9771	0.0012	0.9749	**0.9807**
			PBT-SHADE	**0.9774**	**0.9774**	**0.0010**	**0.9754**	0.9793
			PBT-LSHADE	0.9771	0.9771	0.0013	0.9731	0.9798
	LeNet-5	train	PBT	0.9953	0.9952	0.0010	0.9917	0.9968
			PBT-DE	0.9959	0.9954	0.0016	0.9881	0.9970
			PBT-SHADE	0.9959	0.9959	**0.0009**	0.9930	0.9973
			PBT-LSHADE	**0.9965**	**0.9963**	0.0013	**0.9933**	**0.9980**
		valid	PBT	0.9873	0.9873	0.0008	0.9856	0.9889
			PBT-DE	0.9884	0.9884	0.0008	0.9862	**0.9901**
			PBT-SHADE	**0.9889**	**0.9889**	**0.0005**	**0.9876**	0.9901
			PBT-LSHADE	0.9885	0.9884	0.0009	0.9857	0.9898
		test	PBT	0.9892	0.9893	**0.0007**	0.9876	0.9908
			PBT-DE	0.9899	**0.9900**	0.0009	0.9869	0.9915
			PBT-SHADE	**0.9901**	0.9900	0.0008	**0.9877**	0.9916
			PBT-LSHADE	0.9899	0.9897	0.0011	0.9871	**0.9919**
Fashion MNIST	MLP	train	PBT	0.8972	0.8964	0.0056	0.8827	0.9094
			PBT-DE	0.9073	0.9073	**0.0046**	**0.8971**	0.9174
			PBT-SHADE	0.9094	0.9094	0.0055	0.8953	0.9208
			PBT-LSHADE	**0.9107**	**0.9111**	0.0097	0.8949	**0.9324**
		valid	PBT	0.8754	0.8745	0.0034	0.8647	0.8813
			PBT-DE	0.8804	0.8800	**0.0022**	0.8757	0.8840
			PBT-SHADE	0.8814	0.8815	0.0023	**0.8766**	0.8867
			PBT-LSHADE	**0.8821**	**0.8820**	0.0036	0.8691	**0.8876**
		test	PBT	0.8646	0.8641	0.0034	0.8529	0.8702
			PBT-DE	0.8692	0.8689	0.0026	0.8613	0.8732
			PBT-SHADE	0.8709	0.8708	**0.0023**	**0.8661**	0.8753
			PBT-LSHADE	**0.8725**	**0.8720**	0.0037	0.8600	**0.8799**
	LeNet-5	train	PBT	0.9195	0.9202	**0.0048**	0.9097	0.9314
			PBT-DE	0.9222	0.9226	0.0068	0.9069	0.9379
			PBT-SHADE	0.9284	0.9289	0.0057	**0.9186**	0.9458
			PBT-LSHADE	**0.9328**	**0.9327**	0.0082	0.9175	**0.9476**
		valid	PBT	0.8933	0.8933	0.0036	0.8848	0.9011
			PBT-DE	0.8955	0.8951	0.0025	0.8877	0.8995
			PBT-SHADE	0.8973	0.8976	**0.0020**	**0.8945**	0.9024
			PBT-LSHADE	**0.8981**	**0.8978**	0.0036	0.8873	**0.9028**
		test	PBT	0.8895	0.8891	0.0039	0.8783	0.8976
			PBT-DE	0.8908	0.8901	**0.0035**	0.8808	0.8973
			PBT-SHADE	0.8925	0.8926	0.0036	**0.8832**	0.8992
			PBT-LSHADE	**0.8945**	**0.8936**	0.0049	0.8796	**0.9023**

Note. The best values are highlighted in bold.

Table 4.7: Performance Statistics in Accuracy

Dataset	Model	Set	Algorithm	median	mean	std	min	max
MNIST	MLP	train	PBT	99.550	99.520	0.1713	98.825	99.769
			PBT-DE	99.692	99.670	0.1777	99.170	99.929
			PBT-SHADE	99.823	99.766	0.1577	99.315	99.951
			PBT-LSHADE	**99.909**	**99.871**	**0.1427**	**99.449**	**100.0**
		valid	PBT	97.771	97.758	0.1083	97.402	98.010
			PBT-DE	98.013	98.008	0.1025	97.672	98.202
			PBT-SHADE	98.040	98.037	**0.0700**	**97.862**	98.174
			PBT-LSHADE	**98.042**	**98.043**	0.0936	97.833	**98.363**
		test	PBT	97.870	97.853	0.1188	97.502	98.069
			PBT-DE	98.044	98.038	0.1058	97.840	**98.338**
			PBT-SHADE	**98.069**	**98.063**	**0.0826**	**97.900**	98.238
			PBT-LSHADE	98.049	98.042	0.1177	97.651	98.308
	LeNet-5	train	PBT	99.747	99.737	0.0956	99.425	99.900
			PBT-DE	99.799	99.760	0.1509	99.084	99.898
			PBT-SHADE	99.800	99.800	**0.0813**	99.543	99.925
			PBT-LSHADE	**99.858**	**99.840**	0.1164	**99.570**	**99.994**
		valid	PBT	99.074	99.070	0.0646	98.935	99.194
			PBT-DE	99.158	99.157	0.0679	98.931	99.272
			PBT-SHADE	**99.186**	**99.191**	**0.0520**	**99.111**	**99.318**
			PBT-LSHADE	99.158	99.145	0.0824	98.921	99.262
		test	PBT	99.124	99.126	**0.0635**	98.975	99.254
			PBT-DE	99.194	**99.194**	0.0727	98.915	99.333
			PBT-SHADE	**99.199**	99.188	0.0694	**99.005**	99.323
			PBT-LSHADE	99.174	99.165	0.1021	98.955	**99.363**
Fashion MNIST	MLP	train	PBT	90.525	90.430	0.5285	89.125	91.681
			PBT-DE	91.488	91.455	**0.4364**	**90.464**	92.386
			PBT-SHADE	91.641	91.655	0.5112	90.332	92.686
			PBT-LSHADE	**91.877**	**91.899**	0.9001	90.417	**93.883**
		valid	PBT	88.699	88.631	0.2940	87.759	89.172
			PBT-DE	89.172	89.153	**0.2092**	88.706	89.480
			PBT-SHADE	89.238	89.272	0.2328	**88.753**	89.801
			PBT-LSHADE	**89.369**	**89.358**	0.3305	88.210	**89.867**
		test	PBT	87.749	87.706	0.3116	86.644	88.286
			PBT-DE	88.177	88.153	0.2304	87.629	88.625
			PBT-SHADE	88.361	88.325	**0.2214**	**87.858**	88.754
			PBT-LSHADE	**88.505**	**88.454**	0.3381	87.520	**89.092**
	LeNet-5	train	PBT	92.622	92.681	**0.4450**	91.650	93.706
			PBT-DE	92.859	92.893	0.6430	91.398	94.360
			PBT-SHADE	93.406	93.483	0.5282	**92.624**	95.051
			PBT-LSHADE	**93.920**	**93.924**	0.7707	92.474	**95.384**
		valid	PBT	90.486	90.484	0.3307	89.749	91.252
			PBT-DE	90.654	90.640	0.2504	89.954	91.060
			PBT-SHADE	90.854	**90.892**	**0.2044**	**90.569**	91.335
			PBT-LSHADE	**90.923**	90.886	0.3501	89.916	**91.402**
		test	PBT	90.053	90.004	0.3459	89.072	90.695
			PBT-DE	90.147	90.087	0.3467	89.301	90.685
			PBT-SHADE	90.336	90.319	**0.3388**	**89.341**	90.904
			PBT-LSHADE	**90.496**	**90.388**	0.4364	89.112	**91.083**

Note. The best values are highlighted in bold.

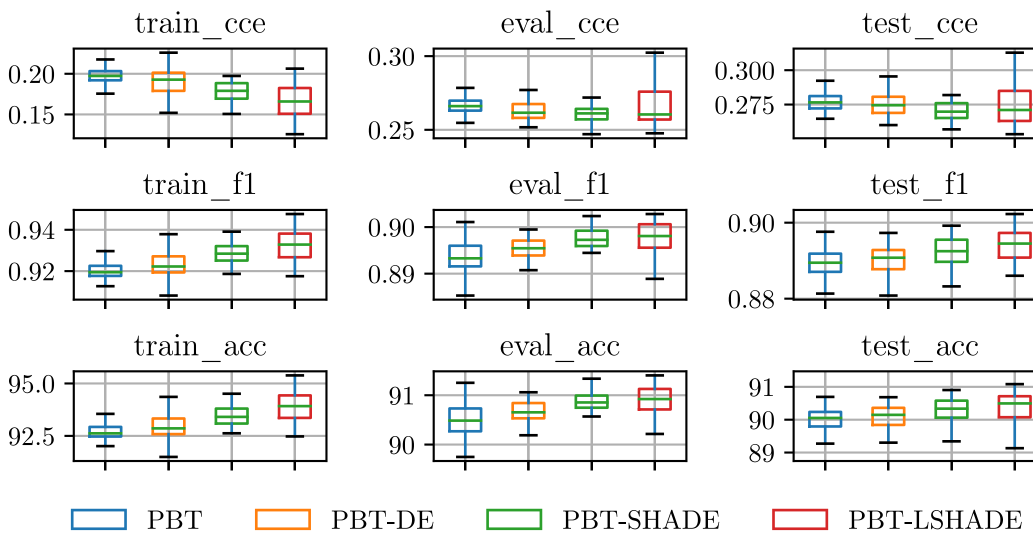


Figure 4.9: Box plot comparisons between the procedures from 50 consecutive tests performed with the Fashion-MNIST dataset using LeNet-5.

was categorized as *training*, *testing* and *evolving*.

The training-task is the procedure where the neural network is training with forward and backward propagation. For PBT, PBT-DE, PBT-SHADE and PBT-LSHADE, this included the *step*. and *eval*-function that preceded the generation of new hyperparameters. Next, the evolving-task consists of the operations conducted in order to generate new trial members. For the PBT procedure, the evolve task includes the exploit- and explore method. For the PBT-DE, PBT-SHADE and PBT-LSHADE procedures, the evolve task is defined by the operations required to generate new trial hyperparameters as well as compare the parent and trial with RFA. Lastly, the testing-task is whenever the test dataset $\mathcal{X}^{(test)}$ is used to measure how members are performing. This is an optional operation that has no effect on the outcome of the procedures, but it was included as it can be subtracted from the total time in order to obtain an approximate total if no testing was conducted.

The time for all three tasks at each step is visualized in Figure 4.11 for the Fashion-MNIST dataset with the LeNet-5 neural network architecture. Each data point represents the step-wise average of the 50 tests that were performed. For convenience, the total time is also included at the bottom of the figure.

In Figure 4.11, the PBT, PBT-DE and PBT-SHADE procedures display similar total time. Moreover, it is clear that the PBT-LSHADE procedure benefit from reducing the number of members over time. When considering the predictive performance measurements that PBT-LSHADE provided the first 10 000 steps as shown and discussed in earlier results, the procedure showed capabilities of providing competitive results with less operations when compared to the other procedures tested in this thesis.

Except from the PBT-LSHADE procedure that showed linear time decay across all tasks, there were some noticeable differences found in the evolve task. Especially the PBT procedure, which spent almost no time performing exploitation and exploration when compared to the DE-based procedures. Of the four procedures, PBT-SHADE spent the

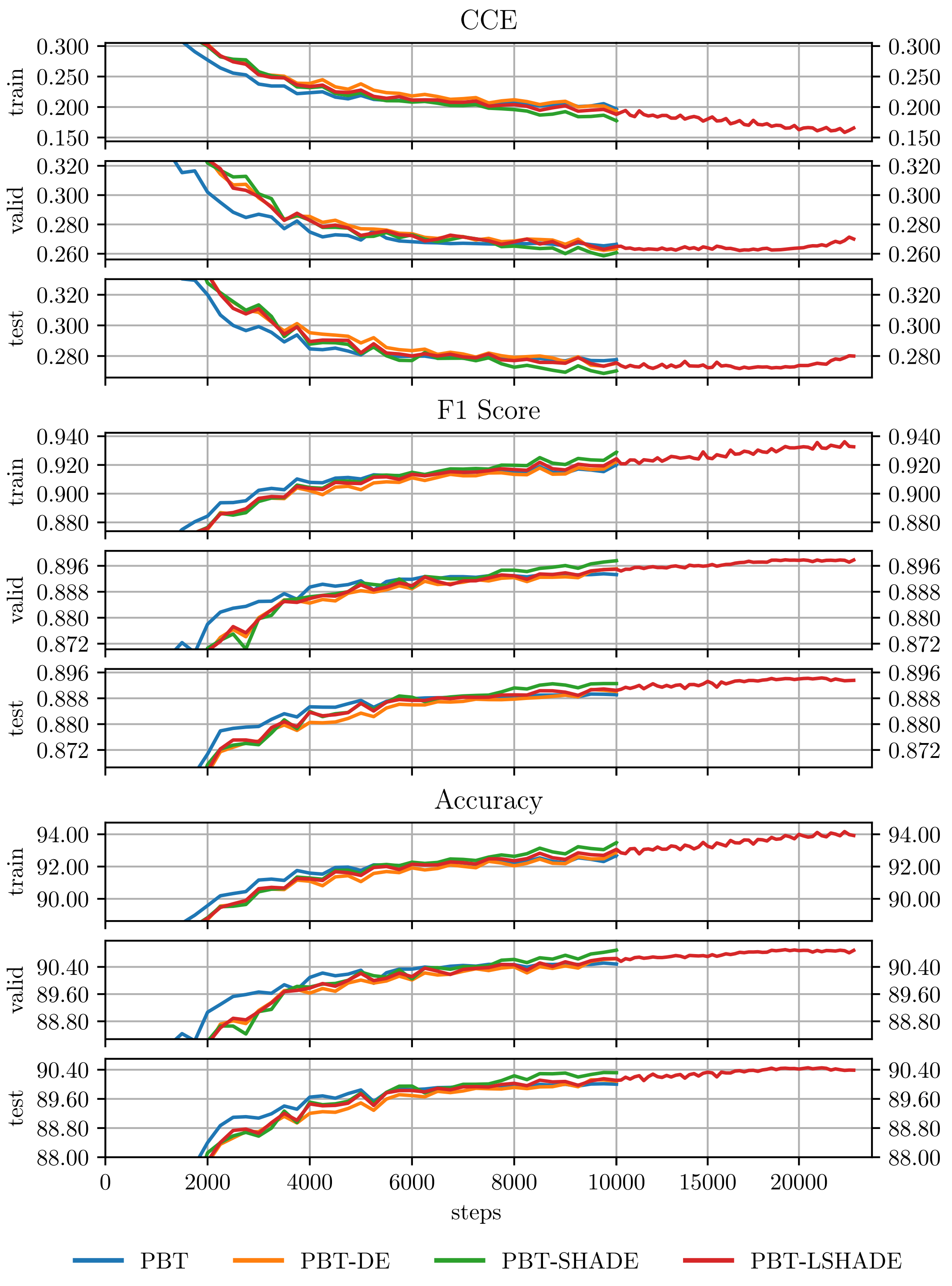


Figure 4.10: Line chart comparisons between the procedures from the average of 50 consecutive tests performed on the Fashion-MNIST dataset using LeNet-5.

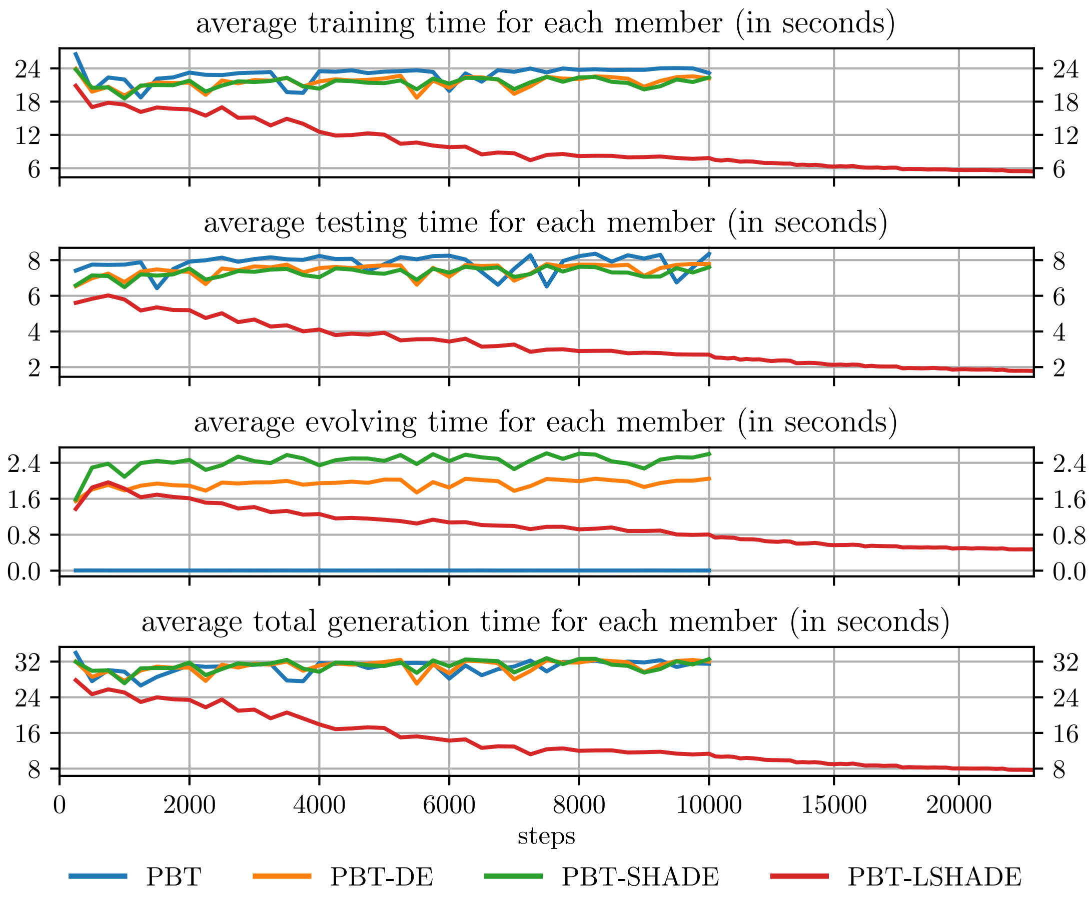


Figure 4.11: A comparison between the procedures on the member-wise average amount of time spent for each step on training, testing and evolving on the LeNet-5 neural network architecture.

most time when generating new trial members and performing the fitness comparisons with RFA.

Another notion is the fact that there was a slight difference between the PBT procedure and the PBT-DE and PBT-SHADE procedures in training and testing time. The training time can be explained by the difference step sizes, as the PBT procedure trained for 250 steps with the $\mathcal{X}^{(train)}$, and performed av full evaluation on $\mathcal{X}^{(valid)}$, while the PBT-DE, PBT-SHADE and PBT-LSHADE procedures performed 242 steps on $\mathcal{X}^{(train)}$ and a full evaluation on $\mathcal{X}^{(valid)}$. The remaining 8 training steps were performed in the RFA method, which would also explain larger evolution time.

4.5 Hyperparameter Schedules

From the perspective of the neural network optimization algorithm, and except from neural network state sharing conducted in the PBT procedure, the only difference between the PBT, PBT-DE, PBT-SHADE and PBT-LSHADE procedures is the final hyperparameter schedules that each procedure generates. In Figure 4.12, the hyperparameter schedules for the Fashion-MNIST dataset using the LeNet-5 neural network architecture, divided by each procedure, are presented. Because there was a lot of result data, the mean is displayed in strong, bold color, as well as the standard deviation which is displayed as dashed lines added and subtracted from the mean. Like earlier figures, data is compressed for the additional generations generated with PBT-LSHADE after 10 000 steps. It is also worth mentioning that we will not include hyperparameter schedules from the other experiments as they are very similar.

It is well known that reducing the learning rate over time typically lead to better results when training a neural network. In Figure 4.12, all four procedures generated learning rate schedules that are in line with that notion. While it is clear that each procedure generated similar-looking learning rate schedules, the PBT procedure generated learning rate schedules that deviated less from the mean when compared to the other procedures, meaning it typically found consistently similar schedules across multiple runs. In addition, the first learning rate values generated by the PBT procedure were approximately close to 0.08, while the learning rate generated by the PBT-DE, PBT-SHADE and PBT-LSHADE procedures typically started around 0.05.

In Figure 4.12, it is evident that the momentum schedules generated by each procedure varied more than the learning rate schedules, especially for the PBT-DE, PBT-SHADE and PBT-LSHADE procedures. There is also a difference in trend, where the PBT procedure generally and gradually reduced the momentum mean from 0.88 to 0.81, where most values oscillated between 0.80 and 0.85, providing the most stable schedules. On the other hand, PBT-DE procedure generated more varying momentum values that oscillate with greater power, where the mean was increasing slightly from 0.83 to 0.87. Lastly, both the PBT-SHADE and PBT-LSHADE procedures produced similar momentum schedules, where the momentum mean generally increased from 0.83-0.84 to 0.89-0.90 for the first 10 000 steps, and then slowly dipped down to 0.88 for the last steps generated by the PBT-LSHADE procedure. It is also clear that the momentum values generated by the PBT-DE procedure oscillated the most for the first steps.

Lastly, Figure 4.12 displays the trends derived from the weight decay hyperparameter schedules. When comparing the weight decay schedules to the other hyperparameters schedules, regardless of which procedure in question, there was not a clear indication

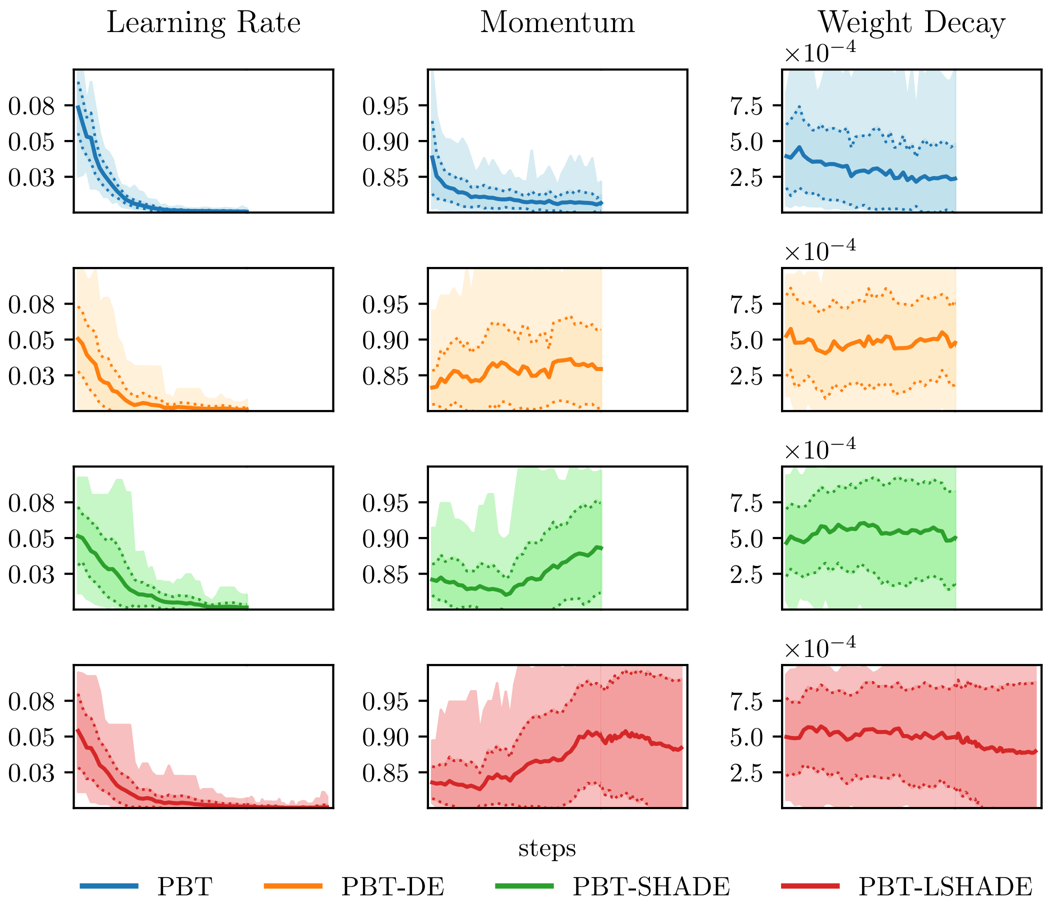


Figure 4.12: Trends derived from hyperparameter schedules generated for the Fashion-MNIST dataset using the LeNet-5 neural network architecture. The brightest color indicate minimum and maximum values, the middle color indicate the standard deviation subtracted from or added to the mean, and the darkest, strongest color indicate the mean.

to how this type of hyperparameter generally evolves in conjunction with the learning rate and momentum hyperparameters. There was some indication that the weight decay schedules generated by the PBT procedure are generally decreasing, and there was a similar indication found for the PBT-LSHADE procedure.

4.6 Impact of Population Size

When testing PBT, PBT-DE, PBT-SHADE and PBT-LSHADE, several parameters required by the procedures needed to be defined, and there were some uncertainty to which ranges of values that were appropriate for the tasks at hand. However, when examining current literature, some of these parameters were given more clarification. More precisely, the end-criterion, i.e. the number of cumulative steps a procedure should perform, as well as the step sizes used in the experimental testing, was inspired by the work conducted in another PBT-related study [143]. Furthermore, and as stated earlier, the parameters by the PBT and DE-based method were determined by their default values suggested by the original authors. What remains is the population size, N, which was decided to be $N = 30$ for all experiments.

However, it is common knowledge in evolutionary literature that population size N can play a large role in the final performance of the optimizer, both in terms of time complexity and predictive performance. Therefore, it was decided to perform additional tests with the PBT and PBT-SHADE procedures on the Fashion-MNIST dataset with the LeNet-5 neural network architecture on a range of population sizes. Figure 4.13 displays the results obtained from testing a population of 10, 20, 30, 40, 50, and 60 initial members. The experiments were conducted five times for each population size for 10 000 steps. The results shown is the average of the tests.

In Figure 4.13, it is clear that each metric follow a similar trend, and it is evident that PBT-SHADE achieved the highest predictive performance on all metrics for each population size. Based on these results, it seems that PBT did not obtain a significant benefit from higher population sizes, and peaks around $30 \leq N \leq 40$ on all metrics. On the other hand, the PBT-SHADE procedure indicated a gradual increase in predictive performance on all metrics with the number of members in the population, which lead to an increase in the performance gap between the average PBT and PBT-SHADE.

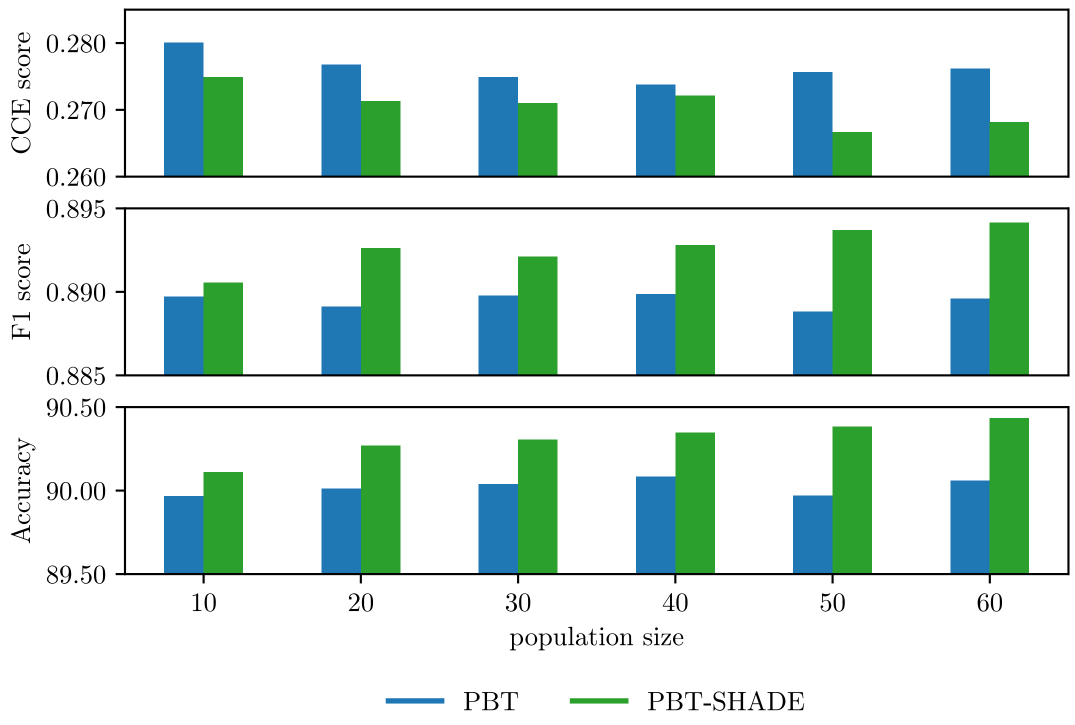


Figure 4.13: A bar plot of different population sizes and the acquired F1 test scores, obtained with the PBT and PBT-SHADE procedures on the Fashion-MNIST dataset with the LeNet-5 neural network architecture.

Chapter 5

Discussion

This chapter presents a concise summary of the principal implications of the obtained results and findings. In Section 5.1, the research questions are restated and provided an answer based on the methods proposed in Chapter 3 and the results presented in Chapter 4. Next, the results will be put in context with the state-of-the-art results achieved on the MNIST and Fashion-MNIST dataset in Section 5.2. Lastly, we will acknowledge the limitations and challenges of this thesis and propose recommendations for future research in Section 5.3.

5.1 Research Questions

This section restates and answers each of the research questions. The answers are based of the work that has been conducted by this thesis.

5.1.1 RQ1

The main research question of this thesis regards how differential evolution heuristics can be incorporated into population-based training for neural networks (RQ1). Based on the sparse studies on PBT (given its recency), as well as the comprehensive research available for DE, we have proposed three novel procedures based of the initial PBT procedure [84], namely PBT-DE, PBT-SHADE and PBT-LSHADE, that successfully incorporate heuristics from both the original DE procedure [179], as well the adaptive and more recent DE extensions, SHADE [187] and L-SHADE [188]. In order to incorporate the heuristics without major modifications, members were processed on a per generation basis while trained and adapted individually using a purpose-built distributed queuing system designed specifically for training multiple neural networks in parallel. The system ensured that the computing speed of different processing devices did not affect the individual member progression. For each proposed procedure, the inherited DE mutation, crossover and selection schemes have replaced the hyperparameter exploitation and exploration method used by PBT in order to generate better members. In order to selection which member that gets to pass on their genome, fitness was measured in the F1 metric using a novel function for assessing the predictive model performance on a subset of the training and validation samples, named RFA (Algorithm 13).

5.1.2 RQ1.1

The second research question, RQ1.1, main interest is the potential predictive performance difference between the PBT procedure and any procedures that incorporate DE heuristics into PBT. More specifically, in order to answer RQ1.1, four experiments were carried out using the MNIST and Fashion-MNIST datasets and the MLP and LeNet-5 neural network architectures. Each procedure were tested using an initial population of 30 members. The obtained results were tested using the Welch's ANOVA statistical significance test, followed up with a Games-Howel post-hoc analysis in order to determine if the results obtained from running the procedures are statistically and significantly differential. The test showed that for all cases, there was a statistical significant difference between the PBT procedure and at least one of the PBT-DE, PBT-SHADE or PBT-LSHADE procedures. Based of these findings, more accurate observations could be made from the predictive performance comparison analysis conducted in Section 4.3, and it was observed that for all tested cases, the PBT procedure was outperformed by the PBT-DE, PBT-SHADE and PBT-LSHADE procedures on the F1 score. Among the proposed procedures, PBT-SHADE and PBT-LSHADE obtained the best results on average, and PBT-SHADE showed to be most consistent. With the same budget, the PBT-LSHADE procedure managed to increase the number of epochs to more than double the number of epochs performed by the other procedures, given the reduction in time complexity by the linear population decay. Moreover, the PBT-LSHADE procedure showed comparable results to the other procedures with a smaller budget that allowed up to 10 000 steps. With more than 10 000 steps, the results obtained by the PBT-LSHADE procedure increased in F1 score and accuracy, yet showed signs of overfitting as the procedure processed more of the same samples in the training set. The findings suggest that the benefit of the PBT-LSHADE procedure is given by its time complexity, and provides good balance between time and predictive performance, especially for smaller budgets that consists of just a few processing devices. All in all, based on the results obtained, we can with 99% certainty state that DE heuristics have statistically and significantly improved upon the predictive performance of the PBT procedure.

5.1.3 RQ1.2

The third and last research question regards how the number of members in the initial population, N, affect the predictive performance of the PBT procedure with DE heuristics when compared to the original PBT procedure. In this case, the proposed PBT-SHADE procedure was tested against the PBT procedure with a set of population sizes, $N \sim \{10, 20, 30, 40, 50, 60\}$. The results shown in Section 4.6 suggested that the PBT-SHADE procedure benefit from higher population sizes, while the PBT procedure gave inconclusive results that failed to determine whether it benefit from more members. When comparing the two procedures, the results suggested that the PBT-SHADE procedure outperformed the PBT procedure on all population sizes. The results also indicated that the performance difference increased with larger population sizes. It is important to note that population sizes did not exceed $N = 60$, and there is a possibility that larger population sizes would have given more clarity or introduced different trends. However, larger population sizes drastically increase the time complexity, so there is a point to be made about the impracticability that comes with large population sizes. When the amount of processing devices is equal the population size, time becomes a smaller issue, as the most demanding computation can be distributed among the devices for both the PBT and PBT-SHADE procedure. Although, obtaining a large number of processing devices might not be viable for smaller research teams.

5.2 Comparing to Other Methods

In literature, performances on the MNIST and Fashion-MNIST datasets for classification are often reported in accuracy across different experimental setups. From Table 4.7, the findings show that the best score obtained on the MNIST and Fashion-MNIST datasets where 99.363% and 91.083% accuracy, respectively, on the LeNet-5 architecture with no dataset augmentations other than normalization and zero-padding. Keep in mind, the goal of this thesis was never to compete with the state of the art on image classification. No extensive data augmentation techniques such as like scaling, rotation or other transformations have been conducted, which have empirically demonstrated to improve results considerably [30, 89, 216]. Furthermore, deeper and more complex network architectures with more free parameters that require considerable more time to train [216] have not been explored either. It is also important to mention that 10 000 of the existing 60 000 training samples from both the MNIST and Fashion-MNIST dataset were reserved for validation, something that is not necessary when model validation is not required. Regardless, here are some of the recent results obtained on the MNIST and Fashion-MNIST datasets.

Lorenzo et al. [123] applied PSO to deep neural networks. PSO is already covered in some detail in Section 2.2.5. The hyperparameter optimization algorithm is applied on the SimpleNet-1 network architecture and obtains an accuracy of 98.92% using 16 particles (i.e. population size) and 10-fold cross validation on the MNIST dataset. In addition, the LeNet-4 network architecture was also tested, and the results show an accuracy of 99.34% on the MNIST dataset using 10-fold cross validation.

Wu [206] propose a general neural network framework which they call ProdSumNet, and demonstrates that good accuracy on the MNIST and Fashion-MNIST dataset can be achieved with a small number of trainable parameters. The results are obtained by applying the ProdSumNet framework to a custom LeNet-5 implementation which uses ReLU activation and dropout layers, and achieve a 99.34% accuracy on the MNIST dataset, and a 92.5% accuracy on the Fashion-MNIST dataset.

Sun et al. [182] propose a method for learning approximations to nonlinear dynamical systems using DNN (NeuPDE). The network architecture used approximates the forward integration of a first-order in time differential equation similar to ResNet and ODENet. When testing, no dataset augmentation was used other than normalization (similar to our approach). The NeuPDE method achieves an accuracy of 99.49% on the MNIST dataset, and an accuracy of 92.4% on the Fashion-MNIST dataset.

Ciregan et al. [30] propose a neural network architecture called Multi-column Deep Neural Networks for Image Classification (MCDNN). The network is optimized using Online Gradient Descent [24], and uses a learning rate decay schedule where the initial learning rate start at 0.001 and is multiplied by 0.993 every epoch. In addition, training samples are augmented with random translation by a maximum factor of 5% of the image height and width. The results show that MCDNN achieves 99.77% accuracy on the MNIST dataset.

Kabir et al. [89] propose SpinalNet – a neural network architecture that is heavily inspired by the human somatosensory system in order to receive large data efficiently and to achieve better performance. The network architecture consists of an input row, an intermediate row, and an output row. The intermediate row consists of multiple hidden layers, where each layer receives a part of the input. Each layer except the input layer receives the output from the preceding layer. Finally, the last layer, output layer, combines the weighted outputs of the hidden neurons in the intermediate row. When combined with VGG-5, SpinalNet achieves a accuracy of 99.72% on the MNIST dataset, and 94.68% accuracy on the Fashion-MNIST accuracy. Both datasets are augmented with random perspective and random rotation between 0 to 10 degrees.

Hirata and Takahashi [74] propose a neural network architecture which they call EnsNet, which is composed using a CNN and multiple fully connected sub-networks. In the model, last convolutional layer in the CNN distributes the feature maps it generates into the disjoint subsets. Each subnet is trained independently in order to predict the class label from the subset of assigned feature-maps. The output of the network is determines by majority vote of the CNN and the sub-networks. EnsNet obtains 99.84% accuracy on the MNIST dataset, and 95.30% accuracy on the Fashion-MNIST dataset.

Zhong et al. [216] propose a Random Erasing data augmentation method which during training, randomly selects rectangle regions in images and changes its pixels with random values. The data augmentation method is carried out in order to reduce overfitting as well as making the model more robust to occlusion. The method is tested with the ResNet, Wide Residual Networks (WRN) and ResNeXt neural network architectures, and obtains a maximum performance of 96.35% accuracy on the Fashion-MNIST dataset with the WRN-28-10 architecture configuration. Harris et al. [68] propose a different data augmentation method called FMix, which uses a binary mask obtained by using a threshold of low frequency images sampled from Fourier space. The FMix method used with the ResNet nerual network architecture achieves an accuracy of 96.36% on the Fashion-MNIST dataset. Lastly, Jayasundara et al. [87] propose a data augmentation technique which generates entirely new training samples from existing dataset samples. The new samples attempts to maintain the same variation found in human handwriting by adding random controlled noise. In addition, reconstruction is improved by combining several loss functions. The technique demonstrates an accuracy of 99.71% on the MNIST dataset and 93.71% on the Fashion-MNIST dataset, using the CapsNet [160] neural network architecture.

Byerly et al. [23] propose a convolutional neural network architecture that extracts features with different receptive fields and abstractions by allowing the network design to branch out after certain convolutions. Each branch transforms the output filters into two homogeneous vector capsules, referred to as filter capsules, which are merged later using a special merge strategy. The training samples are augmented using random rotation, random translation in both directions, random horizontal scaling and random erasure similar to the approach proposed by Zhong et al. [216]. The network is trained for 300 epochs using the Adam [97] optimizer with a initial learning rate of 0.001, which is decayed every epoch by a factor of 0.98. The results show an accuracy of 99.84% on the MNIST dataset – the highest accuracy on MNIST that we were able to observe in literature.

5.3 Challenges and Limitations

In this section, different challenges and limitations are pointed out and discussed.

5.3.1 Synchronous vs. Asynchronous

A lot of thought was given on how to carry out the comparative tests and analysis between the PBT procedure and the proposed procedures. The problem was that training and adaption in PBT is carried out individually by each member, while DE heuristics are carried out generation-wise. Eventually, a decision was made to incorporate heuristics with as few alterations as possible, seeing that other studies had faced similar issues [218]. This lead to PBT being implemented using the same distributed queuing system used by the proposed procedures, leaving only the method that trains and adapts members up for comparison. This allowed for fair and unbiased comparisons to be made on the procedures across generations. In other words, it was ensured that the only difference between the procedures was how new members were generated at each step. If PBT was implemented purely asynchronously, other factors such as variance in processor computing speed could have effected on the final outcome. For the sake of testing consistency and repeatability, the processing devices should only affect the time it takes to run the procedures, not the final result. Enforcing synchronization is not exclusive to this thesis; other PBT-related studies [47, 218] have proposed synchronized variations of PBT, using approaches such as a distributed queuing system similar to ours to efficiently carry out the operations on each member.

5.3.2 Knowledge About the Hyperparameter Search Space

There is also some things to be said about the assumptions of how much knowledge humans should have about the hyperparameter search space. Traditionally, humans have performed hyperparameter tuning manually be sequentially adjusting the hyperparameter configuration until when the result is determined to be good enough for the task at hand. The manual, rule-of-thumb method, as well as semi-automatic methods such as grid search, requires some form of pre-established knowledge about good hyperparameter ranges in the search space. Even fully automatic hyperparameter optimization methods [17, 172, 116, 125, 145, 123, 84, 61] required some knowledge about where in the search space that good hyperparameter values might reside. Methods that completely to remove the user from the optimization process is few and far between [63].

In case of the PBT procedure and the proposed procedures, initial points are determined by sampling values from a uniform distribution within the range of lower and upper boundaries which must be determined by the user. For all experiments, these boundaries were selected based on initial testing and current knowledge about the hyperparameter search space. When different ranges were tested, it was made clear how important it was to select appropriate bounds for each hyperparameter type, suggesting that poorly defined bounds will lead to worse predictive performance. Given the importance of these parameters, it was ensured to explicitly specify the upper and lower boundaries for each hyperparameter in this thesis so that each test is repeatable for future research.

5.3.3 Regulating Deep Networks

Although larger neural networks have demonstrated empirically to outperform human performance on difficult tasks [112, 186], overfitting, i.e. high generalization error, still remains as one of the fundamental issues of deep networks [161]. In order to prevent overfitting, the weight decay regularization technique was used when optimizing the network model in hope that the proposed procedures would be able to generate a good regularization schedule (see Section 3.4.3). Based on the results, it seems that weight decay might not have prevented overfitting entirely, something the was especially noticeable in the results obtained by PBT-LSHADE.

There have been suggestions for other regularization techniques that attempt to reduce the generalization error. A commonly used technique that is often included in machine learning packages [146, 1] is *early stopping*, which is a procedure that monitors the evaluation score and decides when it is time to stop the training process (preferably just before any overfitting occurs). This technique was not used in order to ensure each procedure was tested using up all the available budget.

Another common technique is to insert dropout layers in the network, which randomly deactivates a sample of neurons in the network. The deactivation has two effects: (1) from the point of a deactivated neuron, any activation of downstream neurons will be temporarily disabled in the forward pass, and (2) the deactivated neuron will not receive any weight update in the backward pass. However, dropout layers were not used in this thesis as it was decided to not alter the pre-established neural network architectures.

Another regularization technique is to use data augmentation to reduce overfitting [168] by increasing the number of training samples in order to accommodate for the increased number of free trainable parameters provided by deeper networks. The only data augmentation techniques used in this thesis were normalization and padding.

Although these techniques have shown to help reduce overfitting, Zhang et al. [212] demonstrated that explicit regularization is not a solution for reducing generalization error, and by observing the hyperparameter trend results in Section 4.5, it is clear that each tested procedure fails to find a good weight decay schedule. As of now, we might lack control over how neural networks generalizes training samples, and tools such as regularization techniques are still steps away from a perfect solutions that maintains perfect balance between memorization and generalization.

5.4 Future Work

In this section, some recommendations for future research are made on the topic.

5.4.1 Asynchronous Implementation

A synchronized generation loop may not be the best solution in terms of time complexity for future work that may investigate tuning of hyperparameters related to the neural network architecture (e.g. number of hidden layers). For such cases, an asynchronous optimization loop may be more appropriate, but this needs to be researched as there could be multiple ways to implement asynchronous operations for PBT with DE heuristics. This would also inherently impose modification on DE operations, which traditionally assume an iterative and synchronized procedure loop.

5.4.2 Network Sharing

In the PBT procedure, one of the key operations exploitation is the sharing of partially trained neural network models, i.e. model transfer among members in the population. So far, few network sharing strategies based of PBT have been proposed. Stepleton et al. [175] proposed an adapted PBT procedure which leaves the hyperparameters unchanged throughout the process. Instead, each set of network weights is allowed to "jump" between members while the hyperparameters remain in place, allowing hyperparameter schedules to adapt implicitly. On the other hand, Van Moere [195] showed that for smaller datasets like MNIST, weight copying may have a negative impact on the final accuracy, and concludes that PBT may not even perform better than random search [17] on datasets of similar size.

In this thesis, none of the proposed procedures implement network weight sharing. Early on when developing the procedures, a similar network sharing strategy to the one initially proposed was attempted by allowing the $p$best member to share its model $\mathcal{M}_{p\text{best}}$ with the other members. The results of the initial testing of PBT-DE and PBT-SHADE with network sharing indicated worse predictive performance when compared to the same procedures without network sharing. Based on these early observations, it was concluded that the implementation of network sharing did not benefit the predictive performance of the final procedures, and that bad performance may have been caused by less diversity in the member models, as more members became increasingly more similar. Less diversity may tip off the balance between exploitation and exploration, making the procedures cover less regions in the solution space Θ that could end up leading to a good model θ. However, it is not clear how network sharing should be implemented for PBT with DE heuristics, and we would like to encourage future studies to investigate potential network sharing strategies.

5.4.3 Comparative Study

The results and analysis from Chapter 4 serve as comparative measures between the original PBT procedure and the proposed procedures. Therefore, there is a need for a complete comparative study between the proposed procedures and the state-of-the-art on different application-based benchmarks in a controlled testing environment. The reason for why a controlled comparative study is recommended is based on the challenging and problematic aspects about making appropriate comparisons between different methods in machine learning. Based on observation, research articles may sometimes leave out critical implementation details on how the datasets are sampled, how the samples are augmented and how the learning algorithms are implemented. There is also great variety in research trends, techniques and challenges [214].

In order to compete against state-of-the-art methods, some recommendations will be suggested. Based on existing literature as the ones mentioned in Section 5.2, it is clear that combinations of different data augmentation techniques as well as using more complex learning algorithms are commonly used approaches when testing methods against the state-of-the-art [112, 186, 168]. Data augmentation seems to be especially important for deeper neural networks [168]. Future work should also test whether the procedures are able to successfully optimize network-specific parameters such as the number of hidden layers, or regularization techniques such as the probability in dropout layers.

Chapter 6

Conclusions

This thesis has investigated three research problems. First (RQ1), how to improve upon the initial PBT procedure by incorporating DE heuristics in order to improve exploration of new hyperparameters. Second (RQ1.1), determine the performance difference between the PBT procedure and the proposed procedures that implements DE heuristics, and third (RQ1.2), find out how PBT and the proposed procedures scale with different population sizes.

Three novel approaches for training and adapting multiple neural networks in parallel based of the PBT method and incorporated with DE heuristics, were proposed in order to answer RQ1. The first procedure proposed, named PBT-DE, is based of the initial DE algorithm. The other proposed procedures were named PBT-SHADE and PBT-LSHADE, and are inspired by adaptive DE extensions called SHADE and L-SHADE, respectively. In addition, a distributed queuing system was designed specifically for this task, and allowed for training and adapting each member in the population in parallel while keeping them synchronized between each generation.

Four experiments were carried out using the MNIST and Fashion-MNIST datasets and the MLP and the LeNet-5 neural network architectures in order to answer RQ1.1. Each experiment was carried out in the setting where all procedures are processed using a synchronized generation loop enclosing an asynchronous parallel training and adaption loop. The result data was reported in three metrics: the CCE score, F1 score and accuracy, where the F1 score was used primarily for assessing the performance. The findings demonstrated that the proposed procedures outperformed the PBT procedure in all experiments with an initial population of 30 members. Of the proposed procedures, the PBT-LSHADE procedure demonstrated the overall best scores, but the PBT-SHADE procedure displayed competitive scores with greater consistency. The highest registered scores were obtained by PBT-LSHADE, achieving an accuracy of 99.363% and 91.083% on the MNIST and Fashion-MNIST datasets, respectively, using the LeNet-5 architecture with no dataset augmentations other than normalization and zero-padding.

The PBT and PBT-SHADE procedure was tested using a range of additional population sizes, $(10, 20, 30, 40, 50, 60)$, on the Fashion-MNIST dataset using the LeNet-5 neural network architecture, in order to answer RQ1.2. The results demonstrate that the PBT-SHADE procedure scales better with larger population sizes compared to the PBT procedure, although additional testing of larger population sizes is suggested in order to obtain the full picture.

While the results in Chapter 4 are promising, the experiments were evidently carried out using a distributed queuing system that is not present in the original implementation of PBT. The decision has been thoroughly discussed in Chapter 5, and is based of the challenge of keeping fair comparison while promoting unbiased incorporation of DE heuristics without major modification. Therefore, it is still unknown how PBT with DE heuristics will perform in a truly asynchronous environment, and it is recommended that future work should investigate this further.

The promising results are achieved without using any model transfer between partially trained members. A similar network sharing strategy as the one suggested by PBT was initially attempted for both PBT-DE and PBT-SHADE, but both versions demonstrated poor predictive performance when compared to the same procedures without network sharing. There may still exist a better way to share network states between members, so future work in this area is highly recommended.

The successful improvements made to PBT means there could exist other modifications that may improve the procedure even further. Current findings are limited to image classification on the relatively old MNIST dataset. Therefore, future research is highly encouraged for more recent, real-life tasks that span different domain-based applications. Especially using larger and more complex neural networks, as well as more advanced data augmentations techniques. More advanced DE extensions should also be considered.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning, 2016. URL `https://www.tensorflow.org/`.

[2] H A Abbass. The self-adaptive Pareto differential evolution algorithm. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 1, pages 831–836, 2002. ISBN VO - 1. doi: 10.1109/CEC.2002.1007033.

[3] Rawaa Dawoud Al-Dabbagh, Ferrante Neri, Norisma Idris, and Mohd Sapiyan Baba. Algorithmic design issues in adaptive differential evolution schemes: Review and taxonomy. *Swarm and Evolutionary Computation*, 43:284–311, 12 2018. ISSN 22106502. doi: 10.1016/j.swevo.2018.03.008. URL `https://www.sciencedirect.com/science/article/pii/S2210650217305837https://linkinghub.elsevier.com/retrieve/pii/S2210650217305837`.

[4] N S Altman. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3):175–185, 8 1992. ISSN 0003-1305. doi: 10.1080/00031305.1992.10475879. URL `https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879`.

[5] Nii O Attoh-Okine. Analysis of learning rate and momentum term in backpropagation neural network algorithm trained to predict pavement performance. *Advances in Engineering Software*, 30(4):291–302, 1999. ISSN 0965-9978. doi: https://doi.org/10.1016/S0965-9978(98)00071-4. URL `http://www.sciencedirect.com/science/article/pii/S0965997898000714`.

[6] N H Awad, M Z Ali, P N Suganthan, and R G Reynolds. An ensemble sinusoidal parameter adaptation incorporated with L-SHADE for solving CEC2014 benchmark problems. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 2958–2965, 2016. ISBN VO -. doi: 10.1109/CEC.2016.7744163.

[7] N H Awad, M Z Ali, and P N Suganthan. Ensemble sinusoidal differential covariance matrix adaptation with Euclidean neighborhood for solving CEC2017 benchmark

problems. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 372–379, 2017. ISBN VO -. doi: 10.1109/CEC.2017.7969336.

[8] Noor H Awad, Mostafa Z Ali, Ponnuthurai N Suganthan, Robert G Reynolds, and Ali M Shatnawi. A novel differential crossover strategy based on covariance matrix learning with Euclidean neighborhood for solving real-world problems. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 380–386. IEEE, 6 2017. ISBN 978-1-5090-4601-0. doi: 10.1109/CEC.2017.7969337. URL `http://ieeexplore.ieee.org/document/7969337/`.

[9] Noor H Awad, Mostafa Z Ali, and Ponnuthurai N Suganthan. Ensemble of parameters in a sinusoidal differential evolution with niching-based population reduction. *Swarm and Evolutionary Computation*, 39:141–156, 4 2018. ISSN 22106502. doi: 10.1016/j.swevo.2017.09.009. URL `http://www.sciencedirect.com/science/article/pii/S2210650217305321https://linkinghub.elsevier.com/retrieve/pii/S2210650217305321`.

[10] François Bachoc. Cross Validation and Maximum Likelihood estimations of hyper-parameters of Gaussian processes with model misspecification. *Computational Statistics & Data Analysis*, 66:55–69, 2013. ISSN 0167-9473. doi: https://doi.org/10.1016/j.csda.2013.03.016. URL `http://www.sciencedirect.com/science/article/pii/S0167947313001187`.

[11] James E Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc. ISBN 0-8058-0158-8. URL `http://dl.acm.org/citation.cfm?id=42512.42515`.

[12] Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G Bellemare, and Michael Bowling. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2019.103216. URL `http://www.sciencedirect.com/science/article/pii/S0004370219300116`.

[13] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 7 2012. doi: 10.1613/jair.3912. URL `http://arxiv.org/abs/1207.4708http://dx.doi.org/10.1613/jair.3912`.

[14] R E Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961. ISBN 9781400874668. URL `https://books.google.no/books?id=iwbWCgAAQBAJ`.

[15] Yoshua Bengio. Gradient-Based Optimization of Hyperparameters. *Neural Computation*, 12:1889–1900, 2000.

[16] J Bergstra, D Yamins, and D D Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures.

In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3042832.

[17] James Bergstra and Yoshua Bengio. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2188385.2188395.

[18] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-parameter Optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 2546–2554, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3. URL http://dl.acm.org/citation.cfm?id=2986459.2986743.

[19] Christopher M Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., New York, NY, USA, 1995. ISBN 0198538642.

[20] Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston. Support Vector Machine Solvers. In *Large-Scale Kernel Machines*, page 1. MITP, 2007. URL http://ieeexplore.ieee.org/document/6279971.

[21] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[22] Janez Brest and Mirjam Sepesy Maučec. Self-adaptive differential evolution algorithm using population size reduction and three strategies. *Soft Computing*, 15 (11):2157–2174, 2011. ISSN 1433-7479. doi: 10.1007/s00500-010-0644-5. URL https://doi.org/10.1007/s00500-010-0644-5.

[23] Adam Byerly, Tatiana Kalganova, and Ian Dear. A Branching and Merging Convolutional Network with Homogeneous Filter Capsules. 2020. URL http://arxiv.org/abs/2001.09136.

[24] N Cesa-Bianchi, P M Long, and M K Warmuth. Worst-case quadratic loss bounds for prediction using linear functions and gradient descent. *IEEE Transactions on Neural Networks*, 7(3):604–619, 1996. ISSN 1941-0093 VO - 7. doi: 10.1109/72.501719.

[25] Uday K. Chakraborty, editor. *Advances in Differential Evolution*, volume 143 of *Studies in Computational Intelligence.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1 2008. ISBN 978-3-540-68827-3. doi: 10.1007/978-3-540-68830-3. URL http://link.springer.com/10.1007/978-3-540-68830-3.

[26] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing Multiple Parameters for Support Vector Machines. *Machine Learning*, 46(1): 131–159, 2002. ISSN 1573-0565. doi: 10.1023/A:1012450327387. URL https://doi.org/10.1023/A:1012450327387.

[27] Tianqi Chen and Carlos Guestrin. XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, New York, NY, USA, 8 2016. ACM. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://dl.acm.org/doi/10.1145/2939672.2939785.

[28] Clément Chevalier and David Ginsbourger. Fast Computation of the Multi-Points Expected Improvement with Applications in Batch Selection BT - Learning and Intelligent Optimization. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, pages 59–69, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-44973-4.

[29] Nancy Chinchor. MUC-4 Evaluation Metrics. In *Proceedings of the 4th Conference on Message Understanding*, MUC4 '92, page 22–29, USA, 1992. Association for Computational Linguistics. ISBN 1558602739. doi: 10.3115/1072064.1072067. URL `https://doi.org/10.3115/1072064.1072067`.

[30] D Ciregan, U Meier, and J Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012. ISBN 1063-6919 VO -. doi: 10.1109/CVPR.2012.6248110.

[31] Marc Claesen and Bart De Moor. Hyperparameter Search in Machine Learning. In *The 10th Metaheuristics International Conference*, Agadir, Morocco, 2 2015. URL `http://arxiv.org/abs/1502.02127`.

[32] Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. Easy Hyperparameter Search Using Optunity. In *International Workshop on Technical Computing for Machine Learning and Mathematical Engineering*, 12 2014. URL `https://arxiv.org/abs/1412.1114`.

[33] Marc Claesen, Frank De Smet, Johan A K Suykens, and Bart De Moor. EnsembleSVM: A Library for Ensemble Learning Using Support Vector Machines. *Journal of Machine Learning Research*, 15:141–145, 2014. URL `http://jmlr.org/papers/v15/claesen14a.html`.

[34] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre van Schaik. EMNIST: Extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE, 5 2017. ISBN 978-1-5090-6182-2. doi: 10.1109/IJCNN.2017.7966217. URL `http://ieeexplore.ieee.org/document/7966217/`.

[35] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995. ISSN 1573-0565. doi: 10.1023/A:1022627411411. URL `https://doi.org/10.1023/A:1022627411411`.

[36] Wojciech Marian Czarnecki, Siddhant M. Jayakumar, Max Jadcrbcrg, Leonard Hasenclever, Yec Whye Tch, Simon Osindero, Nicolas Heess, and Razvan Pascanu. Mix & match - Agent curricula for reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *35th International Conference on Machine Learning, ICML 2018*, volume 3, pages 1761–1773, 6 2018. ISBN 9781510867963.

[37] S Das and P N Suganthan. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011. ISSN VO - 15. doi: 10.1109/TEVC.2010.2059031.

[38] S Das, A Abraham, U K Chakraborty, and A Konar. Differential Evolution Using a Neighborhood-Based Mutation Operator. *IEEE Transactions on Evolutionary Computation*, 13(3):526–553, 2009. ISSN VO - 13. doi: 10.1109/TEVC.2008.2009457.

[39] S Das, A Mandal, and R Mukherjee. An Adaptive Differential Evolution Algorithm for Global Optimization in Dynamic Environments. *IEEE Transactions on Cybernetics*, 44(6):966–978, 2014. ISSN VO - 44. doi: 10.1109/TCYB.2013.2278188.

[40] Swagatam Das, Sankha Subhra Mullick, and P.N. Suganthan. Recent advances in differential evolution – An updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 4 2016. ISSN 22106502. doi: 10.1016/j.swevo.2016.01.004. URL `http://www.sciencedirect.com/science/article/pii/S2210650216000146https://linkinghub.elsevier.com/retrieve/pii/S2210650216000146`.

[41] Pieter-Tjerk de Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research*, 134(1):19–67, 2005. ISSN 1572-9338. doi: 10.1007/s10479-005-5724-z. URL `https://doi.org/10.1007/s10479-005-5724-z`.

[42] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y Ng. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc. URL `http://dl.acm.org/citation.cfm?id=2999134.2999271`.

[43] Thomas Desautels, Andreas Krause, and Joel W Burdick. Parallelizing Exploration-Exploitation Tradeoffs in Gaussian Process Bandit Optimization. *Journal of Machine Learning Research*, 15:4053–4103, 2014. URL `http://jmlr.org/papers/v15/desautels14a.html`.

[44] R Eberhart and J Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995. ISBN VO -. doi: 10.1109/MHS.1995.494215.

[45] Bradley Efron and Gail Gong. A Leisurely Look at the Bootstrap, the Jackknife, and Cross-Validation. *The American Statistician*, 37(1):36–48, 1983. ISSN 00031305. doi: 10.2307/2685844. URL `http://www.jstor.org/stable/2685844`.

[46] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger H Hoos, and Kevin Leyton-brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *In NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.

[47] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Online meta-learning by parallel algorithm competition. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 426–433. ACM, 2018. ISBN 1450356184.

[48] O H Elibol, G Keskin, and A Thomas. Semi-supervised and Population Based Training for Voice Commands Recognition. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6371–6375, 2019. ISBN 2379-190X VO -. doi: 10.1109/ICASSP.2019.8683265.

[49] Hugo Jair Escalante, Manuel Montes, and Luis Enrique Sucar. Particle Swarm Model Selection. *J. Mach. Learn. Res.*, 10:405–440, 2009. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1577069.1577084.

[50] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Boron Yotam, Firoiu Vlad, Harley Tim, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *35th International Conference on Machine Learning, ICML 2018*, volume 4, pages 2263–2284, 2 2018. ISBN 9781510867963.

[51] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. *CoRR*, abs/1807.0, 2018. URL http://arxiv.org/abs/1807.01774.

[52] Hui-Yuan Fan and Jouni Lampinen. A Trigonometric Mutation Operation to Differential Evolution. *Journal of Global Optimization*, 27(1):105–129, 2003. ISSN 1573-2916. doi: 10.1023/A:1024653025686. URL https://doi.org/10.1023/A:1024653025686.

[53] Q Fan and X Yan. Self-Adaptive Differential Evolution Algorithm With Zoning Evolution of Control Parameters and Adaptive Mutation Strategies. *IEEE Transactions on Cybernetics*, 46(1):219–232, 2016. ISSN VO - 46. doi: 10.1109/TCYB.2015.2399478.

[54] K Fatahalian, J Sugerman, and P Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 133–137, New York, NY, USA, 2004. ACM. ISBN 3-905673-15-0. doi: 10.1145/1058129.1058148. URL http://doi.acm.org/10.1145/1058129.1058148.

[55] V Feoktistov and S Janaqi. Generalization of the strategies in differential evolution. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 165, 2004. ISBN VO -. doi: 10.1109/IPDPS.2004.1303160.

[56] Vitaliy Feoktistov and Stefan Janaqi. New Strategies in Differential Evolution. In I C Parmee, editor, *Adaptive Computing in Design and Manufacture VI*, pages 335–346. Springer London, London, 2004. ISBN 978-0-85729-338-1.

[57] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. URL http://www.sciencedirect.com/science/article/pii/S002200009791504X.

[58] Alexandre Galashov, Siddhant M. Jayakumar, Leonard Hasenclever, Dhruva Tirumala, Jonathan Schwarz, Guillaume Desjardins, Wojciech M. Czarnecki, Yee Whye Teh, Razvan Pascanu, and Nicolas Heess. Information asymmetry in KL-regularized RL. In *International Conference on Learning Representations*, 5 2019. URL http://arxiv.org/abs/1905.01240https://openreview.net/forum?id=S1lqMn05Ym.

[59] Paul A Games and John F Howell. Pairwise Multiple Comparison Procedures with Unequal N's and/or Variances: A Monte Carlo Study. *Journal of Educational Statistics*, 1(2):113–125, 7 1976. ISSN 03629791. doi: 10.2307/1164979. URL `http://www.jstor.org/stable/1164979`.

[60] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S M Ali Eslami, and Oriol Vinyals. Synthesizing Programs for Images using Reinforced Adversarial Learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1666–1675, Stockholmsmässan, Stockholm Sweden, 2018. PMLR. URL `http://proceedings.mlr.press/v80/ganin18a.html`.

[61] Marius Geitle and Roland Olsson. A New Baseline for Automated Hyper-Parameter Optimization. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 521–530, Cham, 2019. ISBN 978-3-030-37599-7. doi: 10.1007/978-3-030-37599-7{_}43. URL `http://link.springer.com/10.1007/978-3-030-37599-7_43`.

[62] David Ginsbourger, Rodolphe Le Riche, and Laurent Carraro. Kriging Is Well-Suited to Parallelize Optimization. In Yoel Tenne and Chi-Keong Goh, editors, *Adaptation Learning and Optimization*, pages 131–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2 edition, 2010. ISBN 978-3-642-10701-6. doi: 10.1007/978-3-642-10701-6{_}6. URL `https://doi.org/10.1007/978-3-642-10701-6_6http://link.springer.com/10.1007/978-3-642-10701-6_6`.

[63] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D Sculley, editors, *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495, New York, NY, USA, 8 2017. ACM. ISBN 9781450348874. doi: 10.1145/3097983.3098043. URL `https://dl.acm.org/doi/10.1145/3097983.3098043`.

[64] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN 0262035618. URL `http://www.deeplearningbook.org`.

[65] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press. URL `http://dl.acm.org/citation.cfm?id=2969033.2969125`.

[66] Nikolaus Hansen and Andreas Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evol. Comput.*, 9(2):159–195, 2001. ISSN 1063-6560. doi: 10.1162/106365601750190398. URL `http://dx.doi.org/10.1162/106365601750190398`.

[67] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18, 3 2003. ISSN

1063-6560. doi: 10.1162/106365603321828970. URL https://doi.org/10.1162/106365603321828970.

[68] Ethan Harris, Antonia Marcu, Matthew Painter, Mahesan Niranjan, Adam Prügel-Bennett, and Jonathon Hare. FMix: Enhancing Mixed Sample Data Augmentation. 2 2020. URL http://arxiv.org/abs/2002.12047.

[69] Greg Heinrich and Iuri Frosio. Metaoptimization on a Distributed System for Deep Reinforcement Learning. *arXiv preprint arXiv:1902.02725*, 2019.

[70] Lars Hertel, Julian Collado, Peter Sadowski, and Pierre Baldi. Sherpa: hyperparameter optimization for machine learning models. In *2018 Conference on Neural Information Processing Systems*, 2018.

[71] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-Task Deep Reinforcement Learning with PopArt. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01 SE - AAAI Technical Track: Machine Learning), 7 2019. doi: 10.1609/aaai.v33i01.33013796. URL https://www.aaai.org/ojs/index.php/AAAI/article/view/4266.

[72] Geoffrey E Hinton. A Practical Guide to Training Restricted Boltzmann Machines. In Grégoire Montavon, Geneviève B Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, pages 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8{_}32. URL https://doi.org/10.1007/978-3-642-35289-8_32http://link.springer.com/10.1007/978-3-642-35289-8_32.

[73] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.*, 18(7):1527–1554, 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. URL http://dx.doi.org/10.1162/neco.2006.18.7.1527.

[74] Daiki Hirata and Norikazu Takahashi. Ensemble learning in CNN augmented with fully connected subnetworks. 3 2020. URL http://arxiv.org/abs/2003.08562.

[75] Daniel Ho, Eric Liang, Ion Stoica, Pieter Abbeel, and Xi Chen. Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, pages 2731–2741, Long Beach, California, USA, 5 2019. PMLR. URL http://proceedings.mlr.press/v97/ho19b.html.

[76] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification. Technical report, Department of Computer Science, National Taiwan University, 2003. URL http://www.csie.ntu.edu.tw/~cjlin/papers.html.

[77] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Int. Res.*, 36(1): 267–306, 9 2009. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1734953.1734959.

[78] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In Carlos A Coello Coello, editor, *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-25566-3. doi: 10.1007/978-3-642-25566-3{_}40. URL `http://link.springer.com/10.1007/978-3-642-25566-3_40`.

[79] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-25565-6. doi: 10.1007/978-3-642-25566-3{_}40. URL `http://dx.doi.org/10.1007/978-3-642-25566-3_40http://link.springer.com/10.1007/978-3-642-25566-3_40`.

[80] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Parallel Algorithm Configuration. In Youssef Hamadi and Marc Schoenauer, editors, *International Conference on Learning and Intelligent Optimization*, pages 55–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34413-8. doi: 10.1007/978-3-642-34413-8{_}5. URL `http://link.springer.com/10.1007/978-3-642-34413-8_5`.

[81] S M Islam, S Das, S Ghosh, S Roy, and P N Suganthan. An Adaptive Differential Evolution Algorithm With Novel Mutation and Crossover Strategies for Global Numerical Optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):482–500, 2012. ISSN VO - 42. doi: 10.1109/TSMCB.2011.2167966.

[82] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(88)90003-2. URL `http://www.sciencedirect.com/science/article/pii/0893608088900032`.

[83] Sam Adé Jacobs, Nikoli Dryden, Roger Pearce, and Brian Van Essen. Towards Scalable Parallel Training of Deep Neural Networks. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, pages 5:1–5:9, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5137-9. doi: 10.1145/3146347.3146353. URL `http://doi.acm.org/10.1145/3146347.3146353`.

[84] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. 1:1–2, 11 2017. URL `http://arxiv.org/abs/1711.09846`.

[85] Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castañeda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 5 2019. ISSN 10959203. doi: 10.1126/

science.aau6249. URL http://dx.doi.org/10.1126/science.aau6249https://www.sciencemag.org/lookup/doi/10.1126/science.aau6249.

[86] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016*, pages 240–248, 2 2016. URL http://arxiv.org/abs/1502.07943.

[87] Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Jathushan Rajasegaran, Suranga Seneviratne, and Ranga Rodrigo. TextCaps: Handwritten Character Recognition With Very Small Datasets. *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 1 2019. doi: 10.1109/wacv.2019.00033. URL http://dx.doi.org/10.1109/WACV.2019.00033.

[88] 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, 1998. ISSN 1573-2916. doi: 10.1023/A:1008306431147. URL https://doi.org/10.1023/A:1008306431147.

[89] H M Dipu Kabir, Moloud Abdar, Seyed Mohammad Jafar Jalali, Abbas Khosravi, Amir F Atiya, Saeid Nahavandi, and Dipti Srinivasan. SpinalNet: Deep Neural Network with Gradual Input. In *Computer Vision and Pattern Recognition*, 2020.

[90] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

[91] J Kennedy and R Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995. ISBN VO - 4. doi: 10.1109/ICNN.1995.488968.

[92] James Kennedy. The particle swarm: social adaptation of knowledge. *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 303–308, 1997.

[93] James Kennedy, Russel C. Eberhart, and Youhui Shi. *Swarm Intelligence*. Elsevier, San Francisco, CA, USA, 1 edition, 2001. ISBN 978-1-55860-595-4. doi: 10.5555/370449. URL http://www.sciencedirect.com/science/article/pii/B9781558605954500000https://linkinghub.elsevier.com/retrieve/pii/B9781558605954500000.

[94] Mohammad Reza Keshtkaran and Chethan Pandarinath. Enabling hyperparameter optimization in sequential autoencoders for spiking neural data. In H. Wallach, H. Larochelle, A. Beygelzimer, F. Alche-Buc, E. Fox, and R. Garnett, editors, *33rd Conference on Neural Information Processing Systems (NIPS 2019)*, pages 15937–15947, Vancouver, Canada, 8 2019. Curran Associates, Inc. URL http://papers.nips.cc/paper/

`9722-enabling-hyperparameter-optimization-in-sequential-autoencoders-for-spiking-neura`
`pdf`.

[95] Jinwoong Kim, Minkyu Kim, Heungseok Park, Ernar Kusdavletov, Dongjun Lee, Adrian Kim, Ji-Hoon Kim, Jung-Woo Ha, and Nako Sung. CHOPT : Automated Hyperparameter Optimization Framework for Cloud-Based Machine Learning Platforms. 10 2018. URL `http://arxiv.org/abs/1810.03527`.

[96] R. D. King, C. Feng, and A. Sutherland. StatLog: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3):289–333, 5 1995. ISSN 0883-9514. doi: 10.1080/08839519508945477. URL `https://doi.org/10.1080/08839519508945477`.

[97] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, page 13, San Diego, 2015. Ithaca, NY: arXiv.org. URL `http://arxiv.org/abs/1412.6980`.

[98] S Kirkpatrick, C D Gelatt, and M P Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671 LP – 680, 5 1983. doi: 10.1126/science.220.4598.671. URL `http://science.sciencemag.org/content/220/4598/671.abstract`.

[99] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. *CoRR*, abs/1605.0, 2016. URL `http://arxiv.org/abs/1605.07079`.

[100] Ron Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8. URL `http://dl.acm.org/citation.cfm?id=1643031.1643047`.

[101] Ron Kohavi and George H John. Automatic Parameter Selection by Minimizing Estimated Error. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 304–312. Morgan Kaufmann, 1995.

[102] Kamran Kowsari, Mojtaba Heidarysafa, Donald E Brown, Kiana Jafari Meimandi, and Laura E Barnes. RMDL: Random Multimodel Deep Learning for Classification. In *Proceedings of the 2nd International Conference on Information System and Data Mining*, pages 19–28, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450363549. doi: 10.1145/3206098.3206111. URL `http://dx.doi.org/10.1145/3206098.3206111http://dl.acm.org/citation.cfm?doid=3206098.3206111`.

[103] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 5 2012.

[104] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL `http://doi.acm.org/10.1145/3065386`.

[105] J Lampinen. A constraint handling approach for the differential evolution algorithm. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 2, pages 1468–1473, 2002. ISBN VO - 2. doi: 10.1109/CEC.2002.1004459.

[106] Jouko Lampinen. Solving Problems Subject to Multiple Nonlinear Constraints by the Di erential Evolution. In *Proceedings of MENDEL'01-7th International Conference on Soft Computing, Brno, Czech Republic*, pages 50–57, 2001.

[107] Jouni Lampinen, Ivan Zelinka, and others. On stagnation of the differential evolution algorithm. In *Proceedings of MENDEL*, pages 76–83, 2000.

[108] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 473–480, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273556. URL `http://doi.acm.org/10.1145/1273496.1273556`.

[109] J Larsen, L K Hansen, C Svarer, and M Ohlsson. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pages 62–71, 1996. ISBN VO -. doi: 10.1109/NNSP.1996.548336.

[110] Y Lecun, L Bottou, Y Bengio, and P Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 1558-2256 VO - 86. doi: 10.1109/5.726791.

[111] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. URL `http://dl.acm.org/citation.cfm?id=645754.668382`.

[112] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015. ISSN 1476-4687. doi: 10.1038/nature14539. URL `https://doi.org/10.1038/nature14539`.

[113] Howard Levene. Robust tests for equality of variances. *Contributions to probability and statistics. Essays in honor of Harold Hotelling*, pages 279–292, 1960.

[114] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. Population Based Training as a Service. In *NIPS Systems for ML Workshop*, Montréal, Canada, 2018.

[115] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. A Generalized Framework for Population Based Training. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 1791–1799, New York, NY, USA, 7 2019. ACM. ISBN 9781450362016. doi: 10.1145/3292500.3330649. URL `http://doi.acm.org/10.1145/3292500.3330649https://dl.acm.org/doi/10.1145/3292500.3330649`.

[116] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18:1–52, 3 2018. ISSN 15337928. URL `http://arxiv.org/abs/1603.06560`.

[117] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. In *International Conference on Machine Learning AutoML workshop*, 7 2018. URL `http://arxiv.org/abs/1807.05118`.

[118] Chuan Lin, Anyong Qing, and Quanyuan Feng. A comparative study of crossover in differential evolution. *Journal of Heuristics*, 17(6):675–703, 2011. ISSN 1572-9397. doi: 10.1007/s10732-010-9151-1. URL `https://doi.org/10.1007/s10732-010-9151-1`.

[119] Shih-Wei Lin, Kuo-Ching Ying, Shih-Chieh Chen, and Zne-Jung Lee. Particle Swarm Optimization for Parameter Determination and Feature Selection of Support Vector Machines. *Expert Syst. Appl.*, 35(4):1817–1824, 11 2008. ISSN 0957-4174. doi: 10.1016/j.eswa.2007.08.088. URL `http://dx.doi.org/10.1016/j.eswa.2007.08.088`.

[120] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss for Dense Object Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(2):318–327, 2 2020. ISSN 0162-8828. doi: 10.1109/TPAMI.2018.2858826. URL `http://arxiv.org/abs/1708.02002https://ieeexplore.ieee.org/document/8417976/`.

[121] Siqi Liu, Guy Lever, Josh Merel, Saran Tunyasuvunakool, Nicolas Heess, and Thore Graepel. Emergent Coordination Through Competition. In *7th International Conference on Learning Representations, ICLR 2019*, 2 2019. URL `http://arxiv.org/abs/1902.07151`.

[122] Jiancheng Long, Hongming Zhang, Tianyang Yu, and Bo Xu. Iterative Update and Unified Representation for Multi-Agent Reinforcement Learning. 8 2019. URL `http://arxiv.org/abs/1908.06758`.

[123] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. Particle Swarm Optimization for Hyper-parameter Selection in Deep Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 481–488, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071208. URL `http://doi.acm.org/10.1145/3071178.3071208`.

[124] Pablo Ribalta Lorenzo, Jakub Nalepa, Luciano Sanchez Ramos, and José Ranilla Pastor. Hyper-parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1864–1871, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4939-0. doi: 10.1145/3067695.3084211. URL `http://doi.acm.org/10.1145/3067695.3084211`.

[125] Ilya Loshchilov and Frank Hutter. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. In *International Conference on Learning Representations*, 4 2016. URL http://arxiv.org/abs/1604.07269.

[126] Ilya Loshchilov, Marc Schoenauer, and Michèle Sebag. Bi-population CMA-ES agorithms with surrogate models and line searches. In *GECCO*, 2013.

[127] R Mallipeddi, P N Suganthan, Q K Pan, and M F Tasgetiren. Differential evolution algorithm with ensemble of parameters and mutation strategies. *Applied Soft Computing*, 11(2):1679–1696, 2011. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2010.04.024. URL http://www.sciencedirect.com/science/article/pii/S1568494610001043.

[128] Rammohan Mallipeddi. Harmony Search Based Parameter Ensemble Adaptation for Differential Evolution. *J. Applied Mathematics*, 2013:1–750819, 2013.

[129] Rafael Gomes Mantovani, Tomás Horváth, Ricardo Cerri, Joaquin Vanschoren, and André Carlos Ponce de Leon Ferreira de Carvalho. Hyper-Parameter Tuning of a Decision Tree Induction Algorithm. *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 37–42, 2016.

[130] Michael Meissner, Michael Schmuker, and Gisbert Schneider. Optimized Particle Swarm Optimization (OPSO) and its application to artificial neural network training. *BMC Bioinformatics*, 7(1):125, 2006. ISSN 1471-2105. doi: 10.1186/1471-2105-7-125. URL https://doi.org/10.1186/1471-2105-7-125.

[131] Gábor Melis, Chris Dyer, and Phil Blunsom. On the State of the Art of Evaluation in Neural Language Models. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 7 2018. URL https://arxiv.org/abs/1707.05589.

[132] Efrén Mezura-Montes, Carlos A Coello Coello, and Edy I Tun-Morales. Simple feasibility rules and differential evolution for constrained optimization. In *Mexican International Conference on Artificial Intelligence*, pages 707–716. Springer, 2004.

[133] Donald Michie, D J Spiegelhalter, C C Taylor, and John Campbell, editors. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, Upper Saddle River, NJ, USA, 1994. ISBN 0-13-106360-X.

[134] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 1969. ISBN 9780262130431.

[135] Volodymyr Mnih, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *33rd International Conference on Machine Learning, ICML 2016*, volume 4, pages 2850–2869, 2 2016. ISBN 9781510829008. URL http://arxiv.org/abs/1602.01783.

[136] Ali Wagdy Mohamed. An improved differential evolution algorithm with triangular mutation for global numerical optimization. *Computers & Industrial Engineering*, 85:359–375, 2015. ISSN 0360-8352. doi: https://doi.org/10.1016/j.

cie.2015.04.012. URL http://www.sciencedirect.com/science/article/pii/S0360835215001618.

[137] Ali Wagdy Mohamed and Ali Khater Mohamed. Adaptive guided differential evolution algorithm with novel mutation for numerical optimization. *International Journal of Machine Learning and Cybernetics*, 10(2):253–277, 2019. ISSN 1868-808X. doi: 10.1007/s13042-017-0711-7. URL https://doi.org/10.1007/s13042-017-0711-7.

[138] Alex Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo J. Rezende. Towards Interpretable Reinforcement Learning Using Attention Augmented Agents. In *33rd Conference on Neural Information Processing Systems*, Vancouver, Canada, 6 2019. URL http://arxiv.org/abs/1906.02500.

[139] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

[140] Alexander Nareyek. Choosing Search Heuristics by Non-Stationary Reinforcement Learning. In Mauricio G C Resende and Jorge Pinho de Sousa, editors, *Metaheuristics: Computer Decision-Making*, pages 523–544. Springer US, Boston, MA, 2003. ISBN 978-1-4757-4137-7. doi: 10.1007/978-1-4757-4137-7{_}25. URL https://doi.org/10.1007/978-1-4757-4137-7_25http://link.springer.com/10.1007/978-1-4757-4137-7_25.

[141] J A Nelder and R Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 1 1965. ISSN 0010-4620. doi: 10.1093/comjnl/7.4.308. URL https://doi.org/10.1093/comjnl/7.4.308.

[142] Arild Nøkland and Lars Hiller Eidnes. Training Neural Networks with Local Error Signals. In *International Conference on Machine Learning*, 2019.

[143] Stefan Oehmcke and Oliver Kramer. Knowledge Sharing for Population Based Neural Network Training. In Frank Trollmann and Anni-Yasmin Turhan, editors, *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 258–269. Springer International Publishing, Cham, 2018. ISBN 978-3-030-00111-7. doi: 10.1007/978-3-030-00111-7{_}22. URL http://link.springer.com/10.1007/978-3-030-00111-7_22.

[144] Randal S Olson, William La Cava, Zairah Mustahsan, Akshay Varik, and Jason H Moore. Data-driven advice for applying machine learning to bioinformatics problems. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 23: 192–203, 2018. ISSN 2335-6936. URL https://www.ncbi.nlm.nih.gov/pubmed/29218881https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5890912/.

[145] Yoshihiko Ozaki, Masaki Yano, and Masaki Onishi. Effective hyperparameter optimization using Nelder-Mead method in deep learning. *IPSJ Transactions on Computer Vision and Applications*, 9(1):20, 2017. ISSN 1882-6695. doi: 10.1186/s41074-017-0030-7. URL https://doi.org/10.1186/s41074-017-0030-7.

[146] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[147] Martin Pelikan, David E Goldberg, and Erick Cantú-Paz. BOA: The Bayesian Optimization Algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 525–532, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-611-4. URL `http://dl.acm.org/citation.cfm?id=2933923.2933973`.

[148] B T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL `http://www.sciencedirect.com/science/article/pii/0041555364901375`.

[149] M J D Powell. A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In Susana Gomez and Jean-Pierre Hennart, editors, *Advances in Optimization and Numerical Analysis*, pages 51–67. Springer Netherlands, Dordrecht, 1994. ISBN 978-94-015-8330-5. doi: 10.1007/978-94-015-8330-5{_}4. URL `https://doi.org/10.1007/978-94-015-8330-5_4http://link.springer.com/10.1007/978-94-015-8330-5_4`.

[150] K V Price. Differential evolution vs. the functions of the 2/sup nd/ ICEO. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 153–157, 1997. ISBN VO -. doi: 10.1109/ICEC.1997.592287.

[151] Kenneth Price. An Introduction to Differential Evolution. In *New Ideas in Optimization*, pages 79–108. McGraw-Hill Ltd., UK, London, UK, 1999. ISBN 0077095065.

[152] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential Evolution*. Natural Computing Series. Springer-Verlag, Berlin/Heidelberg, 2005. ISBN 3-540-20950-6. doi: 10.1007/3-540-31306-0. URL `http://link.springer.com/10.1007/3-540-31306-0`.

[153] Kenneth V Price and Rainer Storn. Differential Evolution-A simple evolution strategy for fast optimization. *Dr. Dobb's Journal of Software Tools 22*, 4:18–24, 1997.

[154] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 6638–6648, Montréal, Canada, 6 2018. URL `http://arxiv.org/abs/1706.09516`.

[155] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(98)00116-6. URL `http://www.sciencedirect.com/science/article/pii/S0893608098001166`.

[156] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the Expressive Power of Deep Neural Networks. In Doina Precup

and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2847–2854, International Convention Centre, Sydney, Australia, 2017. PMLR. URL `http://proceedings.mlr.press/v70/raghu17a.html`.

[157] Carl Edward Rasmussen and Christopher K I Williams. *Gaussian Processes for Machine Learning.* The MIT Press, 2006. ISBN 0-262-18253-X.

[158] Apostolos Nicholas Refenes, Achileas Zapranis, and Gavin Francis. Stock Performance Modeling Using Neural Networks: A Comparative Study with Regression Models. *Neural Netw.*, 7(2):375–388, 1994. ISSN 0893-6080. doi: 10.1016/0893-6080(94)90030-2. URL `https://doi.org/10.1016/0893-6080(94)90030-2`.

[159] Brian David Ripley. Statistical Aspects of Neural Networks. In O.E. Barndor-Nielsen, J.L. Jensen, and W.S. Kendall, editors, *Networks and Chaos - Statistical and Probabilistic Aspects*, pages 40–123. Chapman & Hall, 1993.

[160] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic Routing Between Capsules. In I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3856–3866. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf`.

[161] Shaeke Salman and Xiuwen Liu. Overfitting Mechanism and Avoidance in Deep Neural Networks. 1 2019. URL `http://arxiv.org/abs/1901.06566`.

[162] S Sanders and C Giraud-Carrier. Informing the Use of Hyperparameter Optimization Through Metalearning. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 1051–1056, 2017. ISBN VO -. doi: 10.1109/ICDM.2017.137.

[163] Mischa Schmidt, Shahd Safarani, Julia Gastinger, Tobias Jacobs, Sebastien Nicolas, and Anett Schulke. On the Performance of Differential Evolution for Hyperparameter Tuning. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2019-July, 4 2019. ISBN 9781728119854. doi: 10.1109/IJCNN.2019.8851978. URL `https://arxiv.org/abs/1904.06960`.

[164] Simon Schmitt, Jonathan J. Hudson, Augustin Zidek, Simon Osindero, Carl Doersch, Wojciech M. Czarnecki, Joel Z. Leibo, Heinrich Kuttler, Andrew Zisserman, Karen Simonyan, and S. M. Ali Eslami. Kickstarting Deep Reinforcement Learning. 3 2018. URL `http://arxiv.org/abs/1803.03835`.

[165] Bernhard Scholkopf and Alexander J Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, USA, 2001. ISBN 0262194759.

[166] D Sculley, Jasper Snoek, Alexander B Wiltschko, and Ali Rahimi. Winner's Curse? On Pace, Progress, and Empirical Rigor. In *ICLR*, 2018.

[167] S S Shapiro and M B Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 7 1965. ISSN 00063444. doi: 10.2307/2333709. URL `http://www.jstor.org/stable/2333709`.

[168] Connor Shorten and Taghi M Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1):60, 2019. ISSN 2196-1115. doi: 10.1186/s40537-019-0197-0. URL https://doi.org/10.1186/s40537-019-0197-0.

[169] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR 2015*, 2015.

[170] Sapna Singh, Daya Shankar Singh, and Shobhit Kumar. Modified Mean Square Error Algorithm with Reduced Cost of Training and Simulation Time for Character Recognition in Backpropagation Neural Network BT - Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applicat. pages 137–145, Cham, 2014. Springer International Publishing. ISBN 978-3-319-02931-3.

[171] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, volume 4, pages 2951–2959, 6 2012. ISBN 9781627480031. URL https://arxiv.org/abs/1206.2944.

[172] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999325.2999464.

[173] Jasper Snoek, Kevin Swersky, Richard Zemel, and Ryan P. Adams. Input Warping for Bayesian Optimization of Non-stationary Functions. In *31st International Conference on Machine Learning, ICML 2014*, volume 5, pages 3654–3662, 2 2014. ISBN 9781634393973. URL http://arxiv.org/abs/1402.0929.

[174] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias W. Seeger. Information-Theoretic Regret Bounds for Gaussian Process Optimization in the Bandit Setting. *IEEE Transactions on Information Theory*, 58(5):3250–3265, 12 2012. ISSN 00189448. doi: 10.1109/TIT.2011.2182033. URL https://arxiv.org/abs/0912.3995.

[175] Thomas Stepleton, Razvan Pascanu, Will Dabney, Siddhant M Jayakumar, Hubert Soyer, and Remi Munos. Low-pass Recurrent Neural Networks-A memory architecture for longer-term correlation discovery. *arXiv preprint arXiv:1805.04955*, 2018.

[176] R Storn. On the usage of differential evolution for function optimization. In *Proceedings of North American Fuzzy Information Processing*, pages 519–523, 1996. ISBN VO -. doi: 10.1109/NAFIPS.1996.534789.

[177] R Storn. On the usage of differential evolution for function optimization. In *Proceedings of North American Fuzzy Information Processing*, pages 519–523, 1996. ISBN VO -. doi: 10.1109/NAFIPS.1996.534789.

[178] R Storn and K Price. Minimizing the real functions of the ICEC'96 contest by differential evolution. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 842–844, 1996. ISBN VO -. doi: 10.1109/ICEC.1996.542711.

[179] Rainer Storn and Kenneth Price. Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute, 1947 Center Street, Berkeley, 1995.

[180] Rainer Storn and Kenneth Price. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997. ISSN 1573-2916. doi: 10.1023/A:1008202821328. URL `https://doi.org/10.1023/A:1008202821328`.

[181] P N Suganthan, Swagatam Das, Satrajit Mukherjee, and Sarthak Chatterjee. Adaptation methods in differential evolution: A review. In *20th International Conference on Soft Computing MENDEL*, volume 2014, pages 131–140, 2014.

[182] Yifan Sun, Linan Zhang, and Hayden Schaeffer. NeuPDE: Neural Network Based Ordinary and Partial Differential Equations for Modeling Time-Dependent Data, 2019.

[183] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton B T Proceedings of the 30th International Conference on Machine Learning. On the importance of initialization and momentum in deep learning, 2 2013. URL `http://proceedings.mlr.press/v28/sutskever13.pdfhttp://proceedings.mlr.press/v28/sutskever13.html`.

[184] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, volume 4, pages 3104–3112, 9 2014. URL `http://arxiv.org/abs/1409.3215`.

[185] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task Bayesian Optimization. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2004–2012, USA, 2013. Curran Associates Inc. URL `http://dl.acm.org/citation.cfm?id=2999792.2999836`.

[186] V Sze, Y Chen, T Yang, and J S Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. ISSN 1558-2256 VO - 105. doi: 10.1109/JPROC.2017.2761740.

[187] R Tanabe and A Fukunaga. Success-history based parameter adaptation for Differential Evolution. In *2013 IEEE Congress on Evolutionary Computation*, pages 71–78, 2013. ISBN 1941-0026 VO -. doi: 10.1109/CEC.2013.6557555.

[188] R Tanabe and A S Fukunaga. Improving the search performance of SHADE using linear population size reduction. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1658–1665, 2014. ISBN 1941-0026 VO -. doi: 10.1109/CEC.2014.6900380.

[189] Ryoji Tanabe and Alex Fukunaga. Evaluating the performance of SHADE on CEC 2013 benchmark problems. In *2013 IEEE Congress on evolutionary computation*, pages 1952–1959. IEEE, 2013.

[190] Jason Teo. Exploring dynamic self-adaptive populations in differential evolution. *Soft Computing*, 10(8):673–686, 2006. ISSN 1433-7479. doi: 10.1007/s00500-005-0537-1. URL `https://doi.org/10.1007/s00500-005-0537-1`.

[191] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, volume Part F1288, page 847, New York, New York, USA, 8 2013. ACM Press. ISBN 9781450321747. doi: 10.1145/2487575.2487629. URL `http://arxiv.org/abs/1208.3719http://dl.acm.org/citation.cfm?doid=2487575.2487629`.

[192] John W Tukey. Comparing Individual Means in the Analysis of Variance. *Biometrics*, 5(2):99–114, 7 1949. ISSN 0006341X, 15410420. doi: 10.2307/3001913. URL `http://www.jstor.org/stable/3001913`.

[193] Josef Tvrdík. Adaptation in differential evolution: A numerical comparison. *Applied Soft Computing*, 9(3):1149–1155, 2009. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2009.02.010. URL `http://www.sciencedirect.com/science/article/pii/S1568494609000301`.

[194] Onay Urfalioglu and Orhan Arikan. Self-adaptive randomized and rank-based differential evolution for multimodal problems. *Journal of Global Optimization*, 51(4):607–640, 2011. ISSN 1573-2916. doi: 10.1007/s10898-011-9646-9. URL `https://doi.org/10.1007/s10898-011-9646-9`.

[195] Anton Van Moere. *Exploring the edges of simplicity in machine learning with creative neural network models*. PhD thesis, Ghent University, 2017. URL `https://lib.ugent.be/fulltxt/RUG01/002/494/502/RUG01-002494502_2018_0001_AC.pdf`.

[196] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 5999–6009, 6 2017. URL `http://arxiv.org/abs/1706.03762`.

[197] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. FeUdal Networks for Hierarchical Reinforcement Learning. In *34th International Conference on Machine Learning, ICML 2017*, volume 7, pages 5409–5418, 3 2017. ISBN 9781510855144. URL `https://arxiv.org/abs/1703.01161`.

[198] Adam Viktorin, Roman Senkerik, Michal Pluhacek, and Tomas Kadavy. Archive analysis in SHADE. In *International Conference on Artificial Intelligence and Soft Computing*, pages 688–699. Springer, 2017.

[199] Hui Wang, Shahryar Rahnamayan, and Zhijian Wu. Parallel differential evolution with self-adapting control parameters and generalized opposition-based learning for solving high-dimensional optimization problems. *Journal of Parallel and Distributed Computing*, 73(1):62–73, 2013. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2012.02.019. URL `http://www.sciencedirect.com/science/article/pii/S0743731512000639`.

[200] Jane X Wang, Edward Hughes, Chrisantha Fernando, Wojciech M Czarnecki, Edgar A Duéñez-Guzmán, and Joel Z Leibo. Evolving intrinsic motivations for altruistic behavior. In *Proceedings of the 18th International Conference on Autonomous*

Agents and MultiAgent Systems, pages 683–692. International Foundation for Autonomous Agents and Multiagent Systems, 2019. ISBN 1450363091.

[201] Y Wang, Z Cai, and Q Zhang. Differential Evolution With Composite Trial Vector Generation Strategies and Control Parameters. *IEEE Transactions on Evolutionary Computation*, 15(1):55–66, 2011. ISSN VO - 15. doi: 10.1109/TEVC.2010.2087271.

[202] Matthieu Weber, Ferrante Neri, and Ville Tirronen. Shuffle or update parallel differential evolution for large-scale optimization. *Soft Computing*, 15(11):2089–2107, 2011. ISSN 1433-7479. doi: 10.1007/s00500-010-0640-9. URL `https://doi.org/10.1007/s00500-010-0640-9`.

[203] Claus Weihs, Karsten Luebke, and Irina Czogiel. Response Surface Methodology for Optimizing Hyper Parameters. Technical report, TU Dortmund University, 2006. URL `http://dx.doi.org/10.17877/DE290R-14252`.

[204] Thomas Weise. *Global Optimization Algorithms - Theory and Application*. Self-Published, second edition, 2009. URL `http://www.it-weise.de/`.

[205] B L Welch. On the Comparison of Several Mean Values: An Alternative Approach. *Biometrika*, 38(3/4):330–336, 7 1951. ISSN 00063444. doi: 10.2307/2332579. URL `http://www.jstor.org/stable/2332579`.

[206] Chai Wah Wu. ProdSumNet: reducing model parameters in deep neural networks via product-of-sums matrix decompositions, 2018.

[207] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. 8 2017. URL `https://arxiv.org/abs/1708.07747http://arxiv.org/abs/1708.07747`.

[208] Rui Xu, Ganesh K. Venayagamoorthy, and Donald C. Wunsch. Modeling of gene regulatory networks with hybrid differential evolution and particle swarm optimization. *Neural Networks*, 20(8):917–927, 10 2007. ISSN 0893-6080. doi: 10.1016/J.NEUNET.2007.07.002. URL `https://www.sciencedirect.com/science/article/pii/S0893608007000998#!`

[209] Daniela Zaharie. Parameter adaption in differential evolution by controlling the population diversity. *Analele Universităţii din Timişoara. Seria Matematică-Informatică*, 40, 1 2002.

[210] Daniela Zaharie. A comparative analysis of crossover variants in differential evolution. *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 171–181, 5 2006.

[211] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 6 2018. URL `https://arxiv.org/abs/1806.01830http://arxiv.org/abs/1806.01830`.

[212] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017.

[213] J Zhang and A C Sanderson. JADE: Adaptive Differential Evolution With Optional External Archive. *IEEE Transactions on Evolutionary Computation*, 13(5):945–958, 2009. ISSN VO - 13. doi: 10.1109/TEVC.2009.2014613.

[214] J M Zhang, M Harman, L Ma, and Y Liu. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering*, page 1, 2020. ISSN 1939-3520 VO -. doi: 10.1109/TSE.2019.2962027.

[215] Shuguang Zhao, Xu Wang, Liang Chen, and Wu Zhu. A Novel Self-adaptive Differential Evolution Algorithm with Population Size Adjustment Scheme. *Arabian Journal for Science and Engineering*, 39(8):6149–6174, 2014. ISSN 2191-4281. doi: 10.1007/s13369-014-1248-7. URL `https://doi.org/10.1007/s13369-014-1248-7`.

[216] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random Erasing Data Augmentation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):13001–13008, 2020. ISSN 2159-5399. doi: 10.1609/aaai.v34i07.7000.

[217] Yinda Zhou, Weiming Liu, and Bin Li. Two-stage population based training method for deep reinforcement learning. In *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*, pages 38–44. ACM, 2019. ISBN 1450366384.

[218] Yinda Zhou, Weiming Liu, and Bin Li. Efficient Online Hyperparameter Adaptation for Deep Reinforcement Learning BT - Applications of Evolutionary Computation. In Paul Kaufmann and Pedro A Castillo, editors, *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 141–155, Cham, 2019. Springer International Publishing. ISBN 978-3-030-16692-2.