Fachhochschul-Masterstudiengang
INFORMATION ENGINEERING & -MANAGEMENT
4232 Hagenberg, Austria

Industrial-Scale Evolutionary Machine Learning: Distributed Coevolutionary Learning in Generative Adversarial Networks

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Tom Schmiedlechner, BSc

Betreuerin: Dr. Una-May O'Reilly
Massachusetts Institute of Technology

Begutachter: Dr. Gabriel Kronberger
Fachhochschule Oberösterreich

Juni 2018

Eidesstattliche Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Tom Schmiedlechner

Linz, 21. 6. 2018

Ort, Datum

Name, Unterschrift

Contents

Acknowledgments

First and foremost, I would like to thank Una-May O'Reilly for the chance to write this thesis here at MIT. I am very grateful for the opportunity to participate in the research work of her and her group, and learned a lot during my time here. I would also like to thank Abdullah Al-Dujaili and Erik Hemberg very much for their constant support throughout this project, and the many hours they have invested in planning, discussions and reviews – because of their help, this work has become what it is now.

Special thanks also go to my supervisor, Gabriel Kronberger, who has made all this possible by making contact with the ALFA group and contributed lots of improvement suggestions.

Finally, I would like to thank my family and my girlfriend Magenta, who support me in all my decisions and made me who I am today.

Abstract

Generative Adversarial Networks (GANs) have received remarkable interest in recent research and generally show promising results in creating generative models with unsupervised learning methods. However, GANs exhibit different critical behaviors like mode and discriminator collapse due to their unstable adversarial nature – problems that have not been fully resolved yet.

In this thesis, we introduce the gradient-based coevolutionary GAN training framework *Lipizzaner*, which combines the advantages of coevolutionary algorithms with those of sophisticated gradient-based trainers for neural networks. It therefore profits from the strengths of both: on one hand, using coevolutionary algorithms leads to stability against collapsing systems, as weak models receive lower fitness values and are ultimately replaced by better performing individuals during the training process. On the other hand, Lipizzaner is able to compete with the training times of non-population based GAN implementations by using fast, gradient-based optimizers. In addition to this, Lipizzaner was designed to run not only on multiple GPUs, but on a distributed cluster of machines in a TCP/IP network. A spatial grid architecture, in which instances only communicate with local neighborhoods of limited size, is used to achieve linear scalability characteristics.

Experiments on commonly used datasets show that Lipizzaner is able to overcome or even prevent otherwise critical collapses, competes with other state-of-the-art GAN implementations in terms of training durations and accuracy, and scales very well in large-scale scenarios.

Chapter 1

Introduction

1.1 Motivation and Overview

There has been great progress in the field of machine learning in the past years, particularly as a result of the increasing usage of big data applications. The majority of current work goes into further research in the field of *deep learning* [8] – and while this positively affects progress in many sub-areas, training discriminative networks for classification or regression is still far more investigated than creating generative networks, which learn to create data from a specific target distribution.

Generative adversarial networks (GANs) aim to fill this gap by utilizing an opposing network, called the *discriminator*, in the process of training the *generator*. The discriminative network aims to distinguish real input data from samples that were created by the generative network, while the generator attempts to create samples the discriminator believes to be real. This leads to an oscillation between the two networks, and ultimately to optimized results for both of them. [19]

GANs are currently primarily used for vision-specific tasks like image or video generation, or in gaming applications – with the clear focus on the generative network. However, we see further possibilities in using the resulting discriminator as well, especially in security applications, or precisely the detection of malware. Training a discriminator against a malware-generating counterpart could result in better generalization to unseen examples, as a broader, varying search space is explored instead of a fixed training set [32, 53].

While GANs generally demonstrate promising results and receive great attention in current research, they still suffer from certain disadvantages, especially in regard of their stability. Mode collapse and discriminator collapse are typical examples of either the generator or the discriminator getting stuck in a local optimum, leading to no further progress as gradient-based optimizers are not able to step out of these situations [41]. As global optimization techniques, evolutionary algorithms are population based and able to select among collapsed and progressing GANs so that a population helps a GAN recover. Additionally, the minimax dynamics of competitive coevolutionary algorithms

are very similar to those of GANs [35] as well, making them an even better match.

However, using evolutionary algorithms to train neural networks is a complex task, additionally increased by the complicated, not yet fully understood dynamics of GANs. As utilizing gradient-based methods usually results in faster convergence, we aim to combine the advantages of both them and evolutionary methods, introducing a coevolutionary system that uses gradient-based optimizers to alter the neural network models instead of applying mutation and crossover operators [38]. In combination with the advantages of populations of adversarial networks, this results in a stable and fast training technique for GANs. To further improve the system, we utilize cooperative as well as competitive approaches to evolve optimizer and mixture parameters.

Finally, coevolutionary algorithms have been shown to be highly parallelizable, as individuals of the populations can reside either on multiple GPUs, or even on different physical machines [46]. This allows the usage of large population sizes even for deep neural networks that require large amounts of memory per model.

When summarized, the points above result in the following formal research question:

Research Question. *How can coevolutionary algorithms and gradient-based optimizers be combined to effectively train generative adversarial networks?*

1.2 Methods

This work focuses on the steps necessary to apply coevolutionary, gradient-based search to deep networks in GAN scenarios, and furthermore on the possibilities to distribute this process over a large computation cluster.

1. First, a coevolutionary framework for training generative adversarial frameworks is implemented and different options regarding the feasible evolutionary methods (e.g. genetic algorithms [18], natural evolution strategies [61], etc.) and ideal gradient-based optimizers (e.g. plain stochastic gradient descent or Adam [36]) are explored in a simplified problem domain.
2. In the second step, the previously introduced framework is extended to be distributable over multiple nodes in a cluster. This is in line with modern computing approaches (i.e. *horizontal* scaling), and allows the framework to both utilize larger population sizes, and compute solutions for more interesting and complex problems.

1.3 Related work

Coevolutionary concepts have never been applied to generative adversarial networks in preceding research. However, population-like concepts have been used with GANs in recent literature [5, 33], although they are not exploiting adaptive search on these populations. In contrast to the framework proposed in this thesis, these approaches are

also limited to a single GPU and therefore not suitable for large-scale scenarios that require distributed systems.

In terms of distributed evolutionary systems, there is interesting progress in utilizing genetic algorithms to train deep neural networks on highly distributed clusters, as demonstrated in [54] and [45]. These approaches however differ from Lipizzaner as they focus on using evolutionary algorithms to optimize a population of network weights and/or topologies, whereas we use them to optimize a population of networks.

1.4 Contributions

In this thesis, the following contributions are reported:

1. A detailed background overview about generative adversarial networks, coevolutionary algorithms, distribution architectures, and the technologies used during this project.
2. The detailed description of both design and implementation characteristics of *Lipizzaner*, a distributable, coevolutionary framework to simultaneously train multiple GANs with gradient-based optimizers.
3. Experiment results to evaluate the performance of Lipizzaner on relevant datasets that are also used in recent GAN literature.
4. Finally, we provide the Lipizzaner framework and experiment code for public usage.[1]

1.5 Thesis Outline

The rest of this thesis is organized as follows: first, a comprehensive theoretical overview about GANs, coevolutionary algorithms, and related distribution strategies is given in Chapter 2. Subsequently, the next chapter describes both the design and implementation details of Lipizzaner, the framework developed during the work on this thesis. Finally, the performance of Lipizzaner is shown in Chapter 4, followed by a discussion of the produced results and the conclusions in Chapter 5.

[1] https://github.com/ALFA-group/lipizzaner-gan

Chapter 2

Background

Lipizzaner combines multiple methods to make use of their combined advantages and abilities. Since these methods have different characteristics that have to be considered when using them, it is sensible to gather profound knowledge about them before discussing the design, architecture and implementation of this system.

This chapter therefore contains details about both the used methods and their underlying concepts. As artificial neural networks are a prerequisite of generative adversarial networks, they are described at the very beginning, followed by GANs themselves. Subsequently, coevolutionary algorithms and possible distribution topologies and technologies are described as well.

2.1 Artificial Neural Networks

Artificial neural networks (or often only *neural networks* or NN) are a class of artificial intelligence systems and loosely based on biological neural networks, i.e. the brains of humans and animals. They are currently the most researched model of artificial intelligence and neuroinformatics, and used for all kinds of machine learning tasks (such as computer vision, speech recognition, video gaming, etc.). While the concept of neural networks dates back to the 1940s, they experienced a revival in the last 10 years due to recent advantages in computational power and increasing availability of data. This was necessary, as training neural networks is an expensive task – especially in current fields of research like deep- and reinforcement learning. [23]

Most currently used neural networks consist of multiple layers of neurons, with each neuron having a weight and a bias value – which are used to parametrized the neurons' activation functions (e.g. *sigmoid* or *tanh*). The output value of neurons is then generated by applying these activation functions to their respective inputs. An example for a simple neural network is illustrated in Figure 2.1.

Neural networks are often trained with gradient-based methods, for example stochastic gradient descent (SGD), a large class of different learning algorithm implementations. They are based on the concept of learning from examples (or *supervised learning*) – i.e.

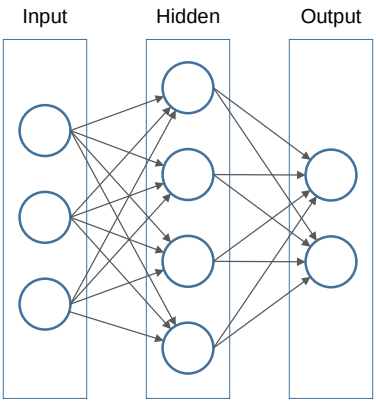


Figure 2.1: Simple example of a neural network, or more specific, a *multilayer perceptron*. It consists of three layers – three input and two output neurons, and one intermediate hidden layer. Topologies in which connections are not cyclic are called *feedforward* networks.

optimizing the network to generate a desired output for specific input values. Therefore, first labeled examples are propagated through the network. Then, the distance between the expected and the real output (i.e. the error) of all layers is calculated and propagated back through the network, adjusting weights and biases of the neurons. [23]

2.1.1 Neuroevolution

Gradient-based methods have become the most common approach to train neural networks during the last years – mostly because of the recent advances in terms of computational power, due to which the usage of deep neural networks became possible.

While SGD is known to outperform most other techniques in common scenarios [23], it still suffers from an important drawback: it cannot be distributed easily to multiple CPUs (due to the high number of small optimization steps) [12]. Even if this problem has been partially solved by other gradient-based algorithms, other conceptual problems – like getting stuck in local minima or saddle points – still persist. [57]

Neuroevolution differs from these gradient-based approaches by using concepts from evolutionary algorithms to evolve either parameters of the network (i.e. neuron weights and biases), or even complete network topologies. Neuroevolution has especially been shown to work well in scenarios that are close to natural evolution, like generating networks to control robots, or in game playing applications. [15, 58]

In recent research, neuroevolution was used to train deep neural networks, and has been shown to match the performance of state-of-the-art gradient-based techniques in the field of reinforcement learning [45, 54]. However, to perform well on this tasks, the proposed systems require enormous amounts of computational power; for example, to evolve the parameters of a neural network that plays Atari games with neuroevolutionary

reinforcement learning (and match the results of gradient-based training), more than 5000 state-of-the-art CPU cores were needed, and the training still took around one week [54]. This shows that distributed systems are an indispensable prerequisite for the success of neuroevolution in industrial-scale real world scenarios, where typically networks with many layers and millions of parameters are used.

2.2 Generative Adversarial Networks

Deep Learning has become one of the most used methods in machine learning, particularly because of the growing amount of available computational power and available data. It has been proven to show great results especially on *discriminative* tasks like speech and image recognition [8], as well as reinforcement learning [55]. However, creating well-performing *generative* networks that create samples from a specific target distribution is still an open research task. Generative networks are for instance useful to create images and 3D models, or objects in architecture, design, or video game applications.

Generative Adversarial Networks (GANs) are a class of unsupervised learning techniques [22] that aim to solve this problem, creating deep generative models by using an adversarial scenario in which a generator $\mathcal{G}$ is trained to create fake data that fools the discriminator (usually using an input vector from a latent space, i.e. random noise), and a discriminator $\mathcal{D}$ evaluates them against data from a real dataset, i.e. tries to distinguish real and fake data. $\mathcal{G}$ therefore learns the distribution of the input data, while $\mathcal{D}$ is concurrently trained to classify real and fake data as accurately as possible [19]. This concurrency is crucial, as it makes both models constantly adjust to each other, and therefore support the learning to occur incrementally in small steps. However, obtaining generative models with such an adversarial approach is challenging and often the approach does not yield a satisfactory model, as we show in our experiments in Chapter 4. The training process of GANs is also illustrated in Figure 2.2.

The ultimate goal of GANs is to create a generative model that performs so well that $\mathcal{D}$ is not able to distinguish real and fake data anymore, i.e. it resorts to merely guessing with a success rate of 50%. However, this may already be the case earlier during the training if both generator and discriminator perform equally badly. Measuring the success of generative learning techniques hence is still an open research question – one option is to compare $\mathcal{G}$ to an optimal discriminative model, if available.

2.2.1 Notation

Generative Adversarial Networks aim to learn two distributions: the generator $\mathcal{G}(z, \Theta_{\mathcal{G}})$ aims to learn a distribution $p_g(z)$ as close as possible to the real distribution $P(x)$ (from which we have some examples), with z defined as the latent input vector, and $\Theta_{\mathcal{G}}$ defined as the network parameters of $\mathcal{G}$ (i.e. the weights and biases). Respectively, the discriminator is defined as $\mathcal{D}(x, \Theta_{\mathcal{D}})$, with x as the input vector and $\Theta_{\mathcal{D}}$ as the network

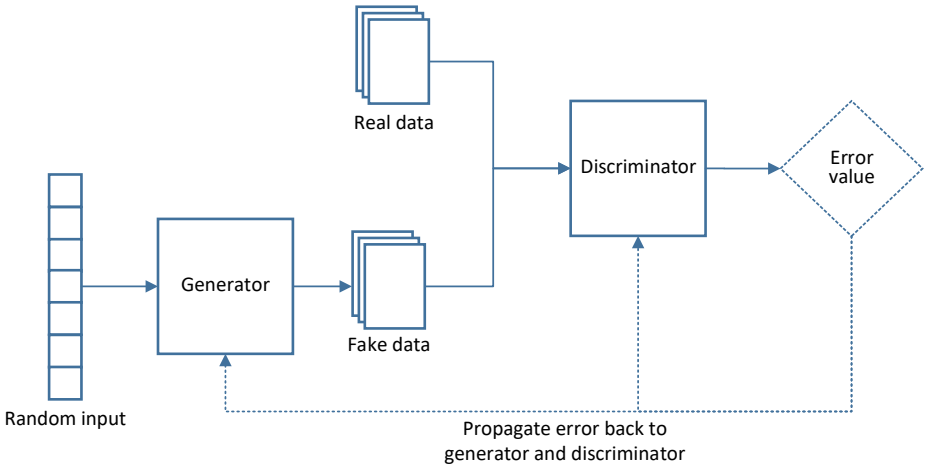


Figure 2.2: Simplified illustration of the behavior of generative adversarial networks (GANs). The discriminator $\mathcal{D}$ aims to maximize the distance between the recognized labels of real and fake data, while the generator $\mathcal{G}$ has the objective of minimizing it.

parameters, and outputs a single scalar that gives the probability x is real. $\mathcal{D}$ is then trained to maximize this probability, while $\mathcal{G}$ is coincidently trained to minimize it (i.e. has the objective to minimize $log(1 - \mathcal{D}(\mathbb{G}(z)))$).

GANs can therefore be seen as two-player minimax games with the goal of finding parameters u and v that optimize the value function

$$\min_{\mathcal{G}} \max_{\mathcal{D}} \mathcal{L}(\mathcal{D}, \mathcal{G}) = \mathbb{E}_{x \sim p_{data}(x)}[\phi(\mathcal{D}(x))] + \mathbb{E}_{x \sim p_z(z)}[\phi(1 - \mathcal{D}(\mathcal{G}(z)))] \tag{2.1}$$

with the concave objective function $\phi(x) = log(x)$. [19]

2.2.2 Advantages and Applications

GANs have several advantages over other techniques that can be used for generative modeling. First and foremost, they are able to generate samples of higher quality than most other established methods. Variational autoencoders, for example, tend to produce blurry images if applied to visual datasets. [37]

Also, unlike other generative techniques, GANs do not rely on Monte Carlo approximations and work better than other techniques such as Boltzmann machines [27] in higher dimensional scenarios. [52]

Finally, GANs are able to learn their respective target distribution in a completely unsupervised process, as no labeling of the data is needed and the generator is not trained on the source dataset, but only on the discriminator's error value (which it aims to minimize for the generated examples). [19]

Even if GANs are currently primarily used for image generation tasks, there's a wide

application space for generative learning techniques overall. Such applications include the creation of 3D models from images, computer games, and even more complex tasks as creating "creative" new versions from a space of already existing objects, which can be useful in architecture and design [19, 59, 64]. Although the concept of GANs is still relatively new (introduced in 2014 by Goodfellow et al. [19]), they are already used in real-world, large scale industrial scenarios, e.g. by Facebook to create predictive models that allow deeper insights into their users' actions [65].

While the usage of GANs has been primarily explored for these generative settings, the resulting discriminator models may also be of interest. In comparison to discriminative models learned from a fixed data distribution, models trained in adversarial scenarios receive different perturbations of the underlying distribution as well, and may therefore generalize better to unseen target distributions. This behavior is especially interesting in malware detection, where attackers may alter their code slightly to fool the discriminator. [29]

2.2.3 Disadvantages and Challenges

While often performing better than other generative modeling techniques, GANs exhibit typical weaknesses and vulnerabilities as well – often correlated to the nature of the gradient-based training methods they rely on [4, 5, 41]:

Oscillation and Divergence One of the main disadvantages of GANs is that they theoretically rely on finding a *Nash Equilibrium* between $\mathcal{G}$ and $\mathcal{D}$, i.e. a scenario where both players take the optimal steps to reach their respective goals in the minimax game described in 2.2.1, leading to a predictable optimal output. However, it is not guaranteed to find this equilibrium, and in fact recent literature suggests that it does not even exist [5]. This is presumably the primary reason for the often observed oscillating behavior where both $\mathcal{G}$ and $\mathcal{D}$ perform equally bad and are unable to step out of this situation. In a similar scenario, either $\mathcal{G}$ or $\mathcal{D}$ may converge to an optimal solution too fast, making its opponents task impossible. [5]

Mode and Discriminator Collapse One of the most-observed failures of GANs in real-world problems is *mode collapse*, which can occur when attempting to learn models from highly complex distributions, e.g. images of high visual quality [4]. In this scenario, the generator is unable to learn the full underlying distribution of the data, and attempts to fool the discriminator by producing samples from a small part of this distribution. Vice versa, the discriminator learns to distinguish real and fake values by focusing on another part of the distribution – which leads to the generator moving its focus to this area, and furthermore to oscillation in which neither $\mathcal{G}$ nor $\mathcal{D}$ is able to step out. [41]

Similarly, *discriminator collapse* is a phenomenon where $\mathcal{G}$ is able to fool $\mathcal{D}$ very well, leading to the latter being stuck in a local minimum [41]. Due to the nature of gradient-based approaches, methods like backpropagation are generally not able to escape these

local minima without further enhancements.

Convergence Criteria Finally, for all generative modeling techniques, evaluating the generated output quality of records sampled by $\mathcal{G}$ is still an open research task. As previously stated, even seemingly converged GANs may produce bad results, leading to the phenomena described above. Several measurements where introduced to quantify the final output quality of the generator [30], mostly focusing on image generation (like the *inception score* [52] or the *Fréchet inception distance* [25]). Another option is to use a pre-existing discriminator that is known to work well on the underlying distribution – however, such discriminators may not exist in unsupervised learning scenarios.

2.3 Coevolutionary Algorithms

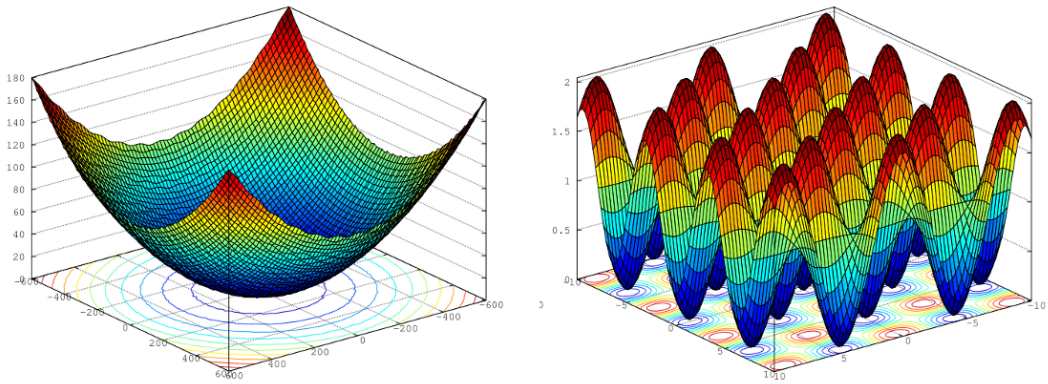
Evolutionary algorithms are population-based optimization techniques based on natural evolution. Generally, an evolutionary algorithm evolves a *population* of *individuals*, using different operators to create new generations from the best-rated individuals of the previous generation. To do so, each individual is evaluated against a *fitness function* – i.e. the function that should be optimized. Then a specified number of parent individuals is selected and used to create the next generation, usually using *crossover* and *mutation* operators to combine and perturb these parent individuals. [6]

Competitive coevolutionary algorithms are a subclass of evolutionary algorithms, with the difference that multiple populations (usually two) are simultaneously evolved instead of a single one [26]. These populations are usually referred to as *parasites and hosts* or *solutions and tests*, depending on the application scenario. Unlike the fitness function of classic evolutionary algorithms, which is usually defining the fitness if a single individual, coevolutionary algorithms use fitness functions that rate individuals according to their respective *opponent* population – similar to natural evolution processes, which also are strongly connected between different populations. Hence, parasites and hosts are also denoted as *predator and prey*, especially in competitive coevolution (see Section 2.3.1).

Formally, competitive coevolutionary algorithms can be described as Minimax problems [13], and therefore share common characteristics with the concept of GANs.

A coevolutionary framework is not bound to a specific evolutionary method, but can host different ones according to the problem that should be solved; in Section 3.1.3, we illustrate how the proposed system is able to host implementations of evolution strategies as well as of genetic algorithms.

Advantages The major advantage of evolutionary algorithms is that the concept of local optima is not valid for them, as there is no implied neighborhood structure that is searched by a stepwise optimizer. As global optimizers that do not rely on gradients, they are naturally not affected by local optima – unlike gradient-based methods, where the user needs to take extra care of this case, it is assumed that beneficial properties of



(a) Global landscape **(b)** Local landscape

Figure 2.3: Fitness landscape of the Griewank function [20] in different scales.[1]

solutions can be combined by crossover. A typical example of a function that is globally concave, but has many local minima, is the Griewank function (as shown in Figure 2.3). With suitable parameters (i.e. a learning rate that allows the algorithm to step out of the local minima), evolutionary algorithms are able to perform well on this kind of functions. [6]

This is especially helpful when dealing with black-box problems, i.e. problems where no search gradient is available, or at least extremely hard to compute. An example for a problem like this in a coevolutionary setting could be the training of two adversarial encrypted neural networks, where both of them only provide an API to evaluate their fitness and alter their parameters, but not to read them. Such a setting can be used to hand over critical systems to clients without giving them the possibility to see inside them, e.g. in malware detection scenarios. Cybersecurity is another related application domain, in which one population contains attacks and the other one defensive configurations [17].

Another characteristic of evolutionary algorithms is that, while being computationally more expensive than typical gradient based methods, they can be easily parallelized, as the evolution of a single individual independent of all other individuals. When applied to large datasets that require the optimizer to be distributed to several nodes (e.g. in the cloud), evolutionary systems therefore profit from fewer synchronization steps, which leads to higher overall performance in this cases (for more details about distributed evolutionary systems, refer to Section 2.4).

Additionally, coevolutionary algorithms have been shown to perform very well in scenarios with a very broad or even changing search space, evolving solutions that can not be easily found using only a simple, non-evolving objective. Due to the simultaneous steps the algorithm makes, a broader solution space can be explored. [42]

[1] https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/Reference/Test Functions

Disadvantages While first drafted in the 1950s and 60s [7], evolutionary (and coevolutionary) algorithms were not of great research interest, because they are generally very computationally expensive. Due to their stochastic nature, evolutionary algorithms require multiple runs, each which expensive fitness evaluations, and revisit samples from the underlying distribution multiple times [6]. With growing research interest in the last years, several different methods to solve this issue were described, leading to a broad space of possibilities in the field of evolutionary algorithms that target different application scenarios (a fact that fits the *No free lunch* theorem [63]). Additionally, as stated above, parallelizing evolutionary optimization systems to different CPU cores or even nodes of a cluster has become an active field of research, also yielding different methods and topologies.

In coevolutionary scenarios, the computational costs become even higher than in classical evolutionary settings; due to the interdependent behavior of the individuals' fitness functions, each individual needs to be evaluated against each individual of the adversarial population, leading to a time complexity of $\mathcal{O}(n^2)$ when implemented naively. Possible solutions for this problem are illustrated in Section 2.4.

Evolutionary algorithms primarily rely on the diversity among their solution candidates, which it than mutates and combines with crossover. Losing this diversity therefore is a critical scenario that strongly resembles convergence to local optima of gradient-based optimizers, and has to be prevented by selecting large population sizes and elaborated algorithm designs. [6, 60]

2.3.1 Categories

Coevolutionary algorithms can be divided into two different categories, depending on the type of interaction between the populations:

- *Competitive* coevolutionary algorithms were the first described category. In competitive CAs, individuals of a population are awarded with higher fitness if they behave well against their respective opponent population, and vice versa, resembling a predator-prey behavior [26]. Hillis et al. introduced them to evolve sorting networks, were their individuals performed well if they were able to sort their opponents in the correct order – while those opponents reached higher fitness scores if they could confuse the sorting population's individuals well. Competitive coevolutionary algorithms are also popular in game theory, were each population represents a player of a two-player game [35].
- *Cooperative* coevolutionary algorithms follow a similar approach, but instead of treating the opponent population as opponents, individuals receive higher fitness values if they perform well in combination with their respective other population. The two populations therefore share a common objective, instead of competing with each other. They were shown to provide good results for complex optimization problems by dividing them into separate parts. [47]

While GANs can be seen as examples of a competitive scenario, we make use of both categories simultaneously in the proposed system, as described in Section 3.1.3.

2.3.2 Relation to GANs

As briefly mentioned above, both GANs and competitive coevolutionary algorithms use the minimax formulation (which is described in Section 2.2.1) [24]. They therefore share similar objectives and scenarios, in which each player (or population) aims to outperform its respective opponents by developing better solutions or tests that perform well against them.

However, unlike coevolutionary algorithms, GANs do not use populations, but use exactly one generator and one discriminator. This is one of the main reasons of many problems connected to GANs, recent research introduced several advanced GAN variants that use techniques similar to populations, making the concepts even more similar [28].

2.4 Distributed Coevolutionary Systems

As already mentioned in the previous sections, high computational costs are a problem for state-of-the-art implementations of coevolutionary algorithms. Despite the advantages of these algorithms, gradient-based methods tend to converge faster, and are therefore more commonly used in current research and industrial scenarios.

Distributing evolutionary systems does not solve this problem, but drastically reduces its impact by sharing the computational nodes over several CPU/GPU cores, or even nodes in a large computation cluster. With the recent advances in cloud computing and the resulting low prices of processing power, this has become an even more suitable option – but naturally, there arises the question of how to distribute these systems. As the underlying problem exists since the invention of evolutionary computing, many different distribution methods have been proposed; the following work therefore focuses on the distribution coevolutionary algorithms only. [46, 62]

2.4.1 Topology

In the most naive setting for distributed systems imaginable, each node (i.e. each machine) communicates with each other, leading to an all-to-all communication pattern (or n^2, where n is the number of nodes). This has several disadvantages:

- A high, exponentially growing *network load*, which would lead to lower communication speed in a network with limited capacity. Currently, private TCP/IP networks are often limited to transfer rates of one GBit/s, but high performance standards that can reach up to 400 GBit/s exist as well.
- Even when executed in a high performance network, the capability of the clients would further limit the expandability; with each added node, the load for all other would grow as well. In coevolutionary systems, this affects both the communication

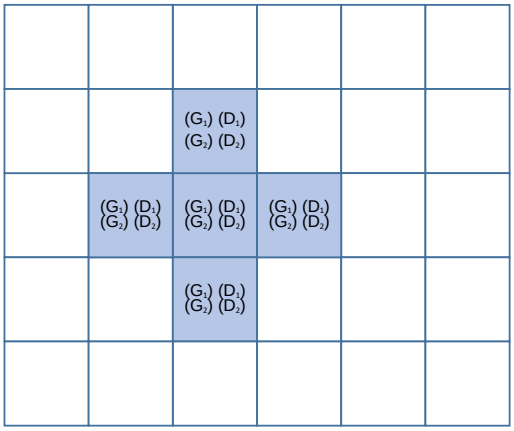


Figure 2.4: Two-dimensional figure of a spatial grid for distributed coevolution; each cell holds two individuals of each population and interacts with a local neighborhood of four other cells. [31]

component and the processing of the individuals (as each one would have to be evaluated against each other; again, the time complexity would be $\mathcal{O}(n^2)$).

Therefore, selecting the appropriate network topology for a system is crucial.

Spatial Grids *Spatial* or *toroidal grids* are a common distribution topology for coevolutionary systems that limit the required communication effort. Spatial/toroidal means that the grid has no edges, but is completely connected in a three-dimensional space. Therefore, each cell has the same number of neighbors.

In a spatial coevolutionary topology, both populations are split and distributed to separate cells of the grid; in different implementations, each cells could either hold individuals from both populations, or an additional dimension could be added to create a separate grid for each population. During the evolutionary process, each cell communicates only with its local neighbors – reducing the complexity of the communication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n * n^2_{neighborhood})$. A *neighborhood* is defined as a number of adjacent cells and specified by its size. In the simplest scenario, a neighborhood could therefore theoretically even consist of only one cell; however, five cells per neighborhood (one center and four adjacent cells) are a commonly used setting, as illustrated in Figure 2.4. [31]

The major advantage of this topology is that the complexity a neighborhood has to deal with does not grow with increased grid size; it is therefore theoretically possible to scale the system infinitely (while in practice, this is obviously limited by the amount of available nodes). While this applies to the necessary computational power, extremely large systems may need further steps to reduce the network load, e.g. by using subnets, if the network performance becomes a bottleneck.

Additionally to these computational advantages, using spatial grids also leads to

enhanced dynamics of the underlying coevolutionary framework; due to being logically separated, different neighborhoods on the grid may evolve into different directions of the fitness landscape, exploring a wider search space and therefore finding more solutions than a non-distributed coevolutionary algorithm. As neighborhoods are overlapping, these information can propagate through the grid as well. After the evolutionary process has finished, neighborhoods can be either combined, or the best one can be selected, depending on the problem scenario. This is usually done by an so-called *orchestrator*, which is responsible for starting experiments, synchronizing them, and collecting their results. [46, 62]

2.4.2 Communication Back-End

A crucial part of distributed systems is the underlying communication back-end. As the amounts of data that those systems must be able to process are still constantly growing, deploying all required instances of such a system on one machine is often not an option. Therefore, network protocols like TCP/IP have to be used instead of single-machine process synchronization, which could for instance be done via system calls. Since TCP/IP is a low-level protocol, several higher level protocols and API specifications were introduced to abstract it; those can generally be categorized in two types, as described in the following sections.

Regardless of the chosen solution, communication interfaces of a system should always be encapsulated into separate components, as future releases of a software may have requirements that differ from those at design time; this is especially relevant for prototypes, where communication speed is often not critical. However, this may change with future development progress of the system, and therefore require the replacement of the communication layer.

Web Services

Web services use a client-server architecture and utilize HTTP methods to exchange data between two machines. While these methods were initially not designed for machine-to-machine communication, their semantics match the CRUD principles (*Create*, *Read*, *Update* and *Delete*) very well, making them a very sophisticated and easy to use abstraction of TCP/IP communication. [1]

The main advantage of web services is their general availability – as the web is based on HTTP, a broad variety of client and server implementations for all relevant programming languages exist. Consequently, HTTP is supported by all commonly used systems and networks, and has been proven to work even under very high loads. Several extensions like *Representational state transfer* (REST) further enhance the underlying semantics of web services, making them easier to utilize in stable, reliable and comprehensible systems. [50]

Regardless these semantic options, resources in web services are generally uniquely defined by an URL, often called an *endpoint*. These endpoints can offer different methods

that can be either used to request, change or delete the specified resource.

Despite their advantages, web services however may not be applicable for high-speed parallel communication. As stated above, the client-server architecture of TCP limits communication to be between two machines exclusively – sharing resources over a whole cluster would therefore require one request from each client. Basic implementations of HTTP web services furthermore do not support bi-directional communication; however, this problem can be solved by using the *websockets* protocol. Websockets are HTTP compatible and establish long-running connections between multiple machines – even if they use TCP/IP underneath, so the issue of one-to-one communication persists. [14]

Message Brokers

In highly parallelized applications, where communication speed and throughput is a critical factor, web services may not be the best option; especially in distributed settings where one-to-all or all-to-all communication is required, HTTP is limited due to its point-to-point architecture.

For use cases in which a system's performance is limited by its communication speed, message brokers may therefore be a more suitable solution. While implementations and concepts vary strongly in this group of protocols and implementations, they usually share some common ideas [10]:

- High-level abstraction of many-to-many communication strategies, like distributing or gathering messages from or to different nodes of a cluster.
- Memory optimization, i.e. the possibility to distribute messages to other processes on the same node without copying them.
- Message queues to support non-blocking communication, which is needed by asynchronous computation models.

While these advantages are especially interesting for systems in which reliability and performance is crucial, message brokers have the disadvantage of being less lightweight than web services. Even if communication between nodes is mostly done via TCP, the additional runtimes often required by these communication frameworks may not be usable in all types of execution environments (or at least require special environment setups). Also, additional components may have to be included in the final software.

Chapter 3

Lipizzaner

In this chapter the main contribution of this thesis is presented: *Lipizzaner*[1], a distributed coevolutionary system to train GANs with gradient-based optimizers. It is introduced in two steps:

First, its general design principles and requirements are elaborated, with special focus on the architecture of the resulting system, and the reasons behind it.

Second, the concrete implementation steps and decisions are shown, including the used technologies, the implementation principles, and the functionality of the core components – the underlying trainers and their subcomponents, the distribution topology, and the analysis dashboard that was implemented to display the experiment results.

The full source code of the project, including additional user documentation and sample experiments, is published on GitHub[2].

3.1 System Design

Lipizzaner is a distributed, coevolutionary system with an underlying extensible framework that allows users to train mixtures of generative adversarial networks with gradient-based optimizers. It therefore combines the advantages of gradient-based GANs – like fast, efficient convergence – with those of coevolutionary algorithms.

It utilizes both competitive and cooperative coevolutionary principles. Besides the competitive nature of GANs themselves, the hyperparameters (e.g. the learning rates) can be evolved as well in a competitive manner. A cooperative approach is used to evolve the mixture parameters (or weights) that are later used to combine the results of the distribution mixture. This is elaborated in more detail in Section 3.1.3 and 3.1.3.

The primary reasons behind this concept are:

- Overcoming the limitations of gradient-based optimizers, like mode- and discriminator collapse, where either the generator or the discriminator get stuck in local

[1] Named after the famous horses of the Spanish riding school in Vienna.

[2] https://github.com/ALFA-group/lipizzaner-gan

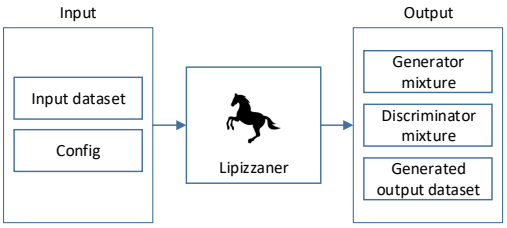


Figure 3.1: High-level overview about the data- and configuration flow through Lipizzaner.

optima. While coevolutionary approaches are theoretically not guaranteed to solve this issues, our experiments show that they perform well in GAN scenarios (as presented in Section 4.1).

- Both coevolutionary algorithms and GANs exhibit very complex dynamics and are therefore hard and computationally expensive to train. To generate results for industrial-scale scenarios, distributing a system therefore is a fundamental prerequisite; while approaches have been shown to run multiple instances of GANs on a single GPU [28], distributing them over multiple machines is still an open topic in research. Coevolutionary algorithms are a possible option to solve this, as they can be efficiently distributed on large scales [46].
- While GANs have been shown to generate promising results in many application scenarios, a general problem is their convergence to final, optimal results [48]. An approach to overcome this problem could be the evolution of the optimizers hyperparameters in a competitive manner, which should ideally lead to better convergence, and therefore avoid infinite oscillation between good, but suboptimal results.

3.1.1 Requirements

Lipizzaner is designed as a generic framework, with a special focus on extendability to allow its usage in different scenarios. In general, users should be able to train the system on an input dataset of their choice (specified by a respective configuration file that e.g. contains data and network specifications), and receive an optimal performing mixture of generators, discriminators and sample output data as a result (as shown in Figure 3.1).

In order to do so, one of the main requirements for the system was the support of multiple, exchangeable trainer components – besides the gradient-based optimizer, trainers for natural evolution strategies and genetic algorithms were implemented in the initial release. To support a broad range of scenarios, all other components (such as the data providing layer and the component for neural network creation) had to be exchangeable as well.

Easy usage and configuration was another central requirement of Lipizzaner, along

with the functionality to provide repeatable results that can be comfortably analyzed and compared.

Finally, the ability to be distributed over multiple machines was a crucial feature as well, as training populations of GANs with coevolutionary algorithms is a computationally expensive task.

The concrete design choices arising from these requirements will be discussed in the following sections.

Functionality and Interchangeability

To be prepared for future use cases, Lipizzaner was designed to run either as a framework or a standalone system; this means that all components should be usable with as few additional dependencies as possible, and the implementation generally follows the principles of *clean coding* [44].

Regardless of its execution type, Lipizzaner's core functionality is very similar to that of traditional coevolutionary systems: input- and configuration data is injected into the trainer component, which performs the evolutionary process on a population of models it maintains – which in this case are both generator and discriminator neural networks (more details about the supported functionality is described in Section 3.2.3).

Generally, great emphasis was placed on making Lipizzaner generic and configurable with minimal effort. All components are fully exchangeable and loosely coupled, most of them even at configuration file level (e.g. the used evolution methods can be changed without altering any source code). The components are connected by injecting their dependencies into the respective classes – a pattern often referred to as *Inversion of Control* (IoC) [16].

Configurability

The ability to be configured without any changes of the source code is a crucial functionality for a distributed system. Specifying central configuration files makes it possible to run experiments on large, distributed computing clusters without redeploying the software and therefore leads to significantly increased usability.

However, as parts of this project may also be used as a framework inside other systems, it is necessary to support two different kinds of configuration:

- Configuration files can be used to simplify the setup and execution of different experiments, especially on multiple nodes (as described above). The configuration parameters are stored in YAML files and passed to the application as a command line argument. The configuration files used for our experiments can be found in Appendix A, with their respective description in Chapter 4.
- Beside this, options can also be directly defined inside the source code, i.e. as parameters of the classes' constructors. When using this configuration method, no additional YAML files are needed (although they still can be used, as a mixture of both methods is possible).

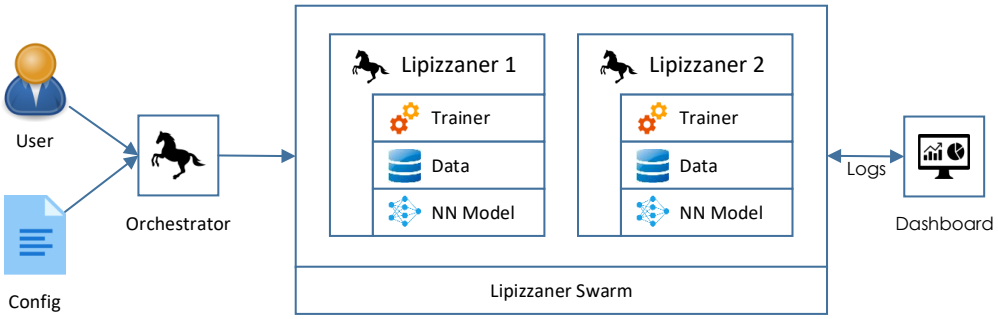


Figure 3.2: High-level view of Lipizzaner's distribution architecture.

Distribution

Distribution is a one of the most important requirements of most state-of-the-art machine learning systems, and even more in the domain of evolutionary computing. As evolutionary algorithms are highly computationally expensive, finding well-performing solutions for distributing them over multiple CPU/GPU cores, or even machines, is a topic of great research interest.

As GANs are relatively new [19], most effort is currently put into tweaking their results, and not into reducing the time they consume for training. To our knowledge, research on "distributed" GANs has currently only been published with the limitation of running them on a single GPU [5, 28]. However, we believe that – especially with sizes of state-of-the-art datasets – distribution is necessary for GANs as well.

As distributing a system usually leads to increased complexity, a central requirement of Lipizzaner was to provide a single point of configuration. Therefore, a *master node* is used to control multiple *clients* by starting experiments, orchestrating their work, and gathering results from them. A master session is unique and terminates after an experiment has finished. In contrast, the client nodes have been implemented to act as long-running services, permanently listening for incoming experiment commands. A high-level overview about the distribution architecture is shown in Figure 3.2.

Finally, an important objective was to reduce the amount of transmitted data. Two main points were considered regarding this: First, each client had to be implemented to maintain its own representation of the trained models, and hold a non-shared dataset[3]. Second, an intelligent communication pattern had to be selected, as all-to-all communication does not scale well in large scenarios and is therefore not feasible for industrial-scale distributed systems. Limiting the communication to local, overlapping neighborhoods that are only gathered at the end of the process drastically reduces the required

[3]While in current experiments each client uses the same dataset, the data layer had been designed to support shared or partitioned datasets as well, e.g. by adding a data loader that pulls samples from a network share or a data server.

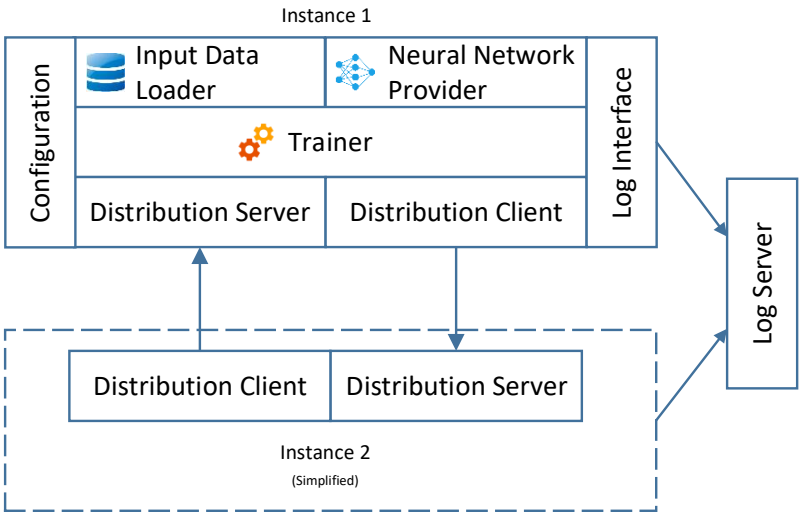


Figure 3.3: Detailed layer architecture of Lipizzaner. As the central component, the trainer accesses both input data and neural network models, and uses its client to connect to other instances' servers and access their respective state. Configuration and log interface are vertical layers, as they are used within all other components.

computation and network load, while also having positive effects on the coevolutionary behavior (see Section 2.4)

3.1.2 Architecture

The Lipizzaner system consists of two parts; a main application, containing all components of the gradient-based, coevolutionary GAN training framework, and a web application dashboard that can be used to analyze experiment results. Both follow layer-based implementation principles and are described in this section.

Lipizzaner Framework

This section describes the primary application presented in this thesis, the software Lipizzaner. Its architecture was designed with the sophisticated component-based layer approach, and fulfills the requirements specified above. The communication between the layers and components is illustrated in Figure 3.3. It should be noted that all described components in a distributed setup exist in each Lipizzaner instance (i.e. each application process).

Input data loader This component provides the data samples the system is trained on, i.e. the distribution the generator aims to reproduce, and provides an interface to access this data. It can be configured to either return complete datasets, or split them into

mini-batches for reduced memory consumption (which is required in most experiment scenarios). The data loader is independent of its underlying data source, which can be an image folder, an online source, or anything else the user injects into it. This also means that the system is not limited to images.

While the current implementations are limited to one dataset per Lipizzaner instance, the `DataLoader` interface is designed to support central (or partitioned) datasets as well.

In its current implementation, Lipizzaner contains multiple datasets by default: the image datasets MNIST [40], CIFAR10 [39] and CelebA [43], and more generic datasets that produce synthetic data points in a 2-dimensional space.

Neural network provider As Lipizzaner evolves the parameters of neural networks (i.e. their weights and biases), a component that handles these neural network models is needed. It exposes a factory class to create both generators and discriminators, as well as public interfaces to manipulate these networks, e.g. to read, adapt and export their parameters. It also hosts the functionality to compare neural networks based on their performance on a given batch of data.

By default, Lipizzaner includes multilayer perceptrons of different sizes, as well as the DCGAN architecture [48] that can be used for image datasets of higher dimensionality.

Training This layer hosts the central component of Lipizzaner and executes the training iterations of the evolutionary process itself. It accesses input data and the neural network models from their respective components and uses them to evolve solutions, with the settings provided by the configuration component. While the primary Lipizzaner algorithm uses gradient-based optimizers to update these solutions, the training component itself is independent from this behavior and can also be used with other, gradient-free algorithms as well (for example, evolution strategies and natural evolution strategies both are implemented as separate trainers in Lipizzaner).

Distribution server and client Consisting of two sub-components (client and server), the distribution layer is responsible for sending and receiving data via a TCP/IP interface. The server component offers a public web service API that can be used by the clients of other instances, and provides endpoints for accessing the current state of the respective instance – such as the parameters of the generator and discriminator individuals. Along with the fitness values of these individuals, most gradient based optimizers also maintain an internal state that has to be shared as well. While the state of some optimizers (like standard SGD) is very lightweight and consists of only a few numeric values (e.g. the learning momentum), more advanced optimizers like Adam require the transmission of complex state objects. Beside the results of the gradient-based training, choosing different optimizers therefore might also affect the communication speed of the system as well.

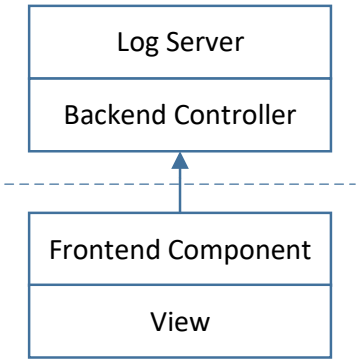


Figure 3.4: Layer architecture of the Lipizzaner dashboard. The components above the dashed line are located on the server, while the components below it are rendered by the client (i.e. the user's web browser).

Configuration In contrast to the horizontal layered components discussed above, the configuration component is aligned vertically over the whole system and therefore connected to all other modules. All relevant parameters can be specified in configuration files – this is crucial, as it allows users to run their experiments without having to manipulate the code, or to execute different experiments on multiple nodes without constantly having to redeploy the application itself to them.

Lipizzaner Dashboard

The Lipizzaner dashboard was designed to simplify the analysis of experiments results produced with the previously described system. Large scale distributed systems naturally create large amounts of log and result data that would be complicated to assess without a central data collection and presentation application. The complex behaviors we aim to monitor – for instance solutions propagating through the grid, oscillation between generators and discriminators, etc. – are also very hard to explore without assistance of a graphical user interface. The components of this dashboard application and their interactions are illustrated in Figure 3.4 and described in this section in detail.

Log database The log database is a system-wide component and accessed by both Lipizzaner – which writes data into it –, and the Lipizzaner dashboard – which reads and presents this data. It was designed to contain information in two hierarchical levels: first, details about the executed experiments, like the distribution topology, evolutionary and optimizer settings, and runtime information. The second, even more relevant hierarchy level stores detailed logs about each step of the experiment on each involved node – like the current state of hyperparameters, mixture weights and loss values, as well as the elapsed time and samples from the current generator distribution (e.g. generated

images).

Back-end controller The back-end controller is a server-side component that connects the front-end to the log database. It therefore has to contain all necessary preprocessing functionality and logic to convert the data into a format that is readable for this front-end before offering it via a web service with endpoints for both experiments and results.

Front-end component and view The front-end component contains all logic needed to access the experiment and result data from the back-end controller and to inject it into the view, which then presents it in a human-readable (and understandable) manner; it has to be possible to group data either by experiments only, or by both experiment and grid node. The view constructs charts and figures for all evolving result values, namely loss, hyperparameters, mixture weights and their respective quality. Additionally, it displays the generated result data (e.g. generated images) for each iteration, if available, and the consumed training time.

3.1.3 Coevolutionary Learning for GANs

As described in Section 2.1, neuroevolution has been shown to generate remarkable results in a broad area of deep learning applications. It therefore seems reasonable to apply these techniques to coevolutionary setups as well – specifically to train generative adversarial networks, which resemble the scenario of coevolution very closely. Working with GANs means to simultaneously train two ideally oscillating networks – generator and discriminator –, a task that is usually performed with classical gradient-based methods like stochastic gradient descent (i.e. backpropagation), with the ultimate goal of finding generators that create fake data which cannot be distinguished from real data.

This concept obviously shares some similarities to coevolution, where two populations are evolved in parallel as well, usually referred to as *solutions* and scenarios/tests, or *predators* and *prey* in competitive coevolution. Both concepts particularly target so-called *minimax* problems, as described in Section 2.3.

As we show in Section 4.1, gradient-free coevolutionary algorithms are able to perform well in low-dimensional synthetic GAN problems, and are able to avoid or escape scenarios that would be critical for gradient-based optimizers (like mode and discriminator collapse). However, our experiments also show that evolutionary algorithms are unfortunately not applicable to higher-dimensional coevolutionary GAN problems with reasonable resource consumption. In fact, global optimizers seem to be not suitable to optimize millions of parameters in an adversarial setting at all. Additionally, the computational costs – which are already very high in "normal", non-competitive neuroevolution – are even larger for GANs. In combination with the fact that most GAN application scenarios expose gradients, it is reasonable to utilize them as well.

To overcome the limitations of gradient-based approaches, Lipizzaner incorporates these trainers into a coevolutionary framework. Additionally to the natural competitive

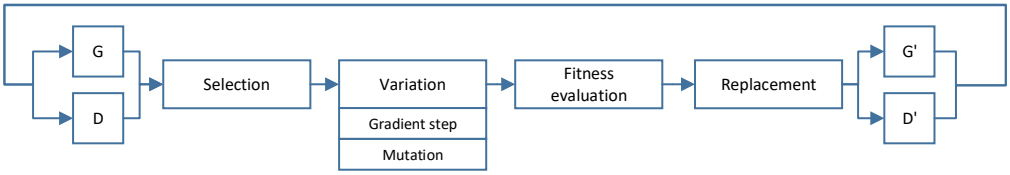


Figure 3.5: Evolutionary process of competitive coevolutionary learning for GANs, illustrating that both gradient and non-gradient variation is used in the process.

approach of GANs, Lipizzaner is able to both competitively evolve the optimizers' hyperparameters as well as cooperatively alter the weights of the resulting generator and discriminator mixtures. While this leads to several interesting effects, it hardly adds any additional computation costs to the system. Both competitive and cooperative approaches are described in the remaining part of this section.

Competitive Coevolution of Networks and Hyperparameters

The competitive coevolutionary process in Lipizzaner consists of two different tasks: aside from evolving the network parameters of the individuals in the generator and discriminator populations (i.e. the weights and biases of their neurons), the optimizers hyperparameters are evolved as well (see Figure 3.5).

Neural network parameters This primary part of the whole evolutionary process is responsible for updating biases and weights of the populations' neural network models. In contrast to classical coevolution, where global optimizers with evolutionary operators like mutation and recombination of the individuals would be used, Lipizzaner performs these updates with gradient-based optimizer steps (i.e. backpropagation). As stated above, this leads to considerably decreased training times, as no global search is performed.

That said, it has to be noted that Lipizzaner includes algorithms that perform gradient-free steps with evolution strategies or natural gradient-based steps with natural evolution strategies as well. As this first approach has been shown to perform not well enough on the high-dimensional problems we are interested in (see Chapter 4), these trainers were furthermore merely used for comparison purposes.

Optimizer hyperparameters Improving the final convergence of solutions (i.e. reach an equilibrium) is still an open topic in GAN research [5]. Some implementations reach better convergence by using hard-coded conditions to decrease the optimizer's learning rate after a given number of epochs [48] – but even if it may work for specific experiment scenarios, this approach requires manual elaboration of the best-fitting hyperparameters, and does not generalize to other problems.

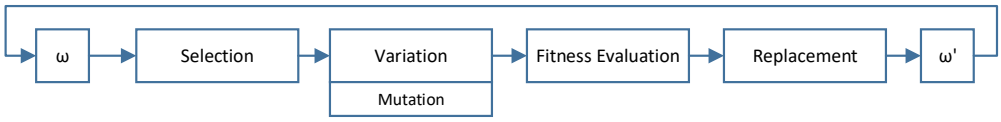


Figure 3.6: Cooperative coevolution for GAN mixture weights.

Additionally to the networks parameters, we therefore evolve these hyperparameters as well to create a self-optimizing system that may converge to lower learning rates during training. In Lipizzaner, specific learning rates are bound to each individual, effectively making them another parameter that is evolved in a separate step of the evolutionary process.

Cooperative Coevolution of Mixture Parameters

While simultaneously evolving pairs of generators and discriminators works well with the competitive approach described above, those separate individuals finally have to be combined in a way that performs as well as possible after the training process finishes. One way to combine or *fuse* neural networks is to gather them in a mixture (i.e. an ensemble), in which one of the elements is selected to process a request each time it is applied to an input. A probability vector for a neighborhood size of 3 could therefore be $[0.3, 0.1, 0.6]$ – meaning that the probability to draw a sample from the first individual would be three times as high as drawing one from the second one, etc. For future uses, Lipizzaner evolves discriminator mixtures as well, even if those are not used yet in our current experiments.

It would have been possible to achieve this by using a mixture with equal probabilities, resulting in an uniform random distribution when drawing samples from the mixture. However, as some of the neural networks in the mixture may be very similar in terms of their computed results (i.e. multiple generators may focus on the same mode of the target distribution), those should be given lower weights – as the ultimate goal of mixtures is to promote high diversity of the computed output. It is therefore reasonable to include the weights of these mixture parts into the evolutionary process as well [4].

In each iteration, Lipizzaner therefore runs a cooperative evolutionary step (i.e. evolves the weights in the mixture vectors) and computes a measurement quantifying the performance of the current neighborhoods mixture (i.e. its fitness). After the training process finishes, each neighborhood yields a mixture of a specific quality, allowing the user to select the best-performing one (see Figure 3.6).

3.2 Implementation

This section contains information about how the relevant parts of the system design described above are implemented in detail, including the used technologies and algorithms.

For easier understanding, the contents of this section build on each other. At first, basic knowledge about the used technologies is needed to understand their advantages and limitations. This is followed by details about the implementation of the distributed system, as this is a prerequisite of understanding how Lipizzaner works. Finally, the result analysis tools are described.

3.2.1 Technology Stacks

As described above, Lipizzaner consists of two separate applications; the main application that uses the underlying implemented framework to coevolutionary train GANs, and the analytics application that was designed to review and analyze the results of our distributed experiments.

While the training application was implemented in Python and using the PyTorch machine learning framework, the analytics dashboard is an ASP.NET Core application with an Angular frontend written in TypeScript. This polyglot approach was taken to make use of both technology stacks advantages and strengths in their respective field.

Distributed Deep Learning with Python

Selecting the best technology stack for a system is a complicated process, as the definition of *best* is very subjective and highly depends on the field of application, the approached objectives of the project, and especially the experience of the developers using it. However, in machine learning applications, Python is currently the by far most used programming language [68] due to several reasons [49]:

- Many features of Python were especially designed for mathematical and scientific applications – like the array indexing syntax, duck typing functionality, etc.
- Python is easy to learn and use, and therefore widely used in many educational institutions and universities. As many machine learning applications are initially developed in an academic or research context (or at least by people with a respective background), most developers are already used to it.
- Most relevant frameworks in the machine learning domain are primarily targeting Python, and a high number of packages for all kinds of related requirements exist (including well working, sophisticated package managers like *Pip* to obtain them).
- Python is platform-independent and runs on all kinds of environments, making it even easier to use for different application scenarios (as industrial fields of application may require different environments than those in research).

While Python therefore is well-suited for most machine learning applications, it also

comes with some drawbacks, especially when used to create large systems. Most constructs in Python, especially duck typing and lazy, unsafe evaluations, require careful, failsafe implementations, and complicate the system by the necessity of many runtime checks. Additionally, the Python threading system is hardly applicable in large, asynchronous applications (as described below). These points were considered and accepted as non-critical during the design of Lipizzaner, as it was planned to be a prototype application that will primarily be used in a research context, and not in productive industrial systems.

PyTorch Even if it is theoretically possible to manually implement all required functionality for training neural networks in plain Python, this would come with several major drawbacks – in particular much higher implementation time, additional code that has to be tested, and remarkably less features than a sophisticated external library or framework. It is therefore reasonable to use an externally developed package for these purposes, of which several exists – including PyTorch[4]. PyTorch is a relatively new machine learning framework for training neural networks, and developed as an *Open Source* project primarily by Facebook. While older, more well-known frameworks like TensorFlow[5] are more sophisticated than PyTorch, many of those are primarily designed to perform as well as possible in production scenarios, leading to several drawbacks, including a more complicated syntax and API, and more complex runtime requirements.

PyTorch was especially designed to be easy to use, and is therefore well-suited for academic projects and prototypes. While it allows to implement applications very quickly, PyTorch still suffers from typical "early open source project" problems, like API instabilities and more bugs than well-established frameworks like TensorFlow – for example, major version updates often lead to breaking changes of existing code. Also, parts of PyTorch are less sophisticated than others, as described in the next section.

In Lipizzaner, PyTorch is widely used for all tensor operations – i.e. defining and training neural networks, and generally working with data. As most operations on neural networks (like propagating data forward and errors backward through them) are matrix multiplications, Lipizzaner highly profits from PyTorch's GPU support for these calculations, leading to reduced training times for deep GANs from several hours per iteration to only minutes.

In contrast to the traditional concept of abstraction, some parts of PyTorch were not totally encapsulated due to two reasons: First, the level of central concepts such as the tensor implementation of PyTorch is integrated too deeply to be wrapped without having to re-implement (or at least also wrap) major parts of the remaining framework. Second, PyTorch is well-optimized for high performance applications, and encapsulating this functionality (e.g. the GPU acceleration) would render these optimizations useless. Therefore, PyTorch concepts are partly exposed to the user of the framework – however, special emphasis was put on interchangeability of these parts as well.

[4] https://pytorch.org
[5] https://www.tensorflow.org

Distributing PyTorch applications PyTorch natively supports distribution over multiple CPUs, GPUs and even machines in a network since 2017 – meaning that this is a relatively new functionality. The `torch.distributed` package offers a communication interface similar to MPI (*Message Passing Interface*, a messaging standard designed for parallel computing), with typical operations to send and receive data among multiple participants [66].

Using this functionality was the first approach while implementing Lipizzaner. However, as stated above, PyTorch is a relatively new framework, and the package for distributed computing is even more recent.As a consequence, we faced several issues and disadvantages during its implementation, which ultimately lead to the switch to web services (as described in the next section). Among these issues, the most critical were:

1. Due to the packages implementation design, using it requires a lot of additional code and complex transformations for certain use cases. As it is only supported to transmit single tensors at once, it is not possible to distribute objects or lists without some additional effort to package or concatenate them to tensors. While most of these operations are trivial, more complicated ones emerged as well. Additionally, even transformation operations that are relatively simple individually lead to a far more complex system when combined.
2. Distributing tensors stored on GPUs requires additional, third-party runtime environments that have to be installed on the client machines. Furthermore, enabling distributed GPU support requires PyTorch to be manually built on each node, a process that takes around one hour on state-of-the-art hardware.
3. Finally, errors are hard to find, as using the package prevents debugging in many cases (it cannot handle the thread blocking that occurs when a breakpoint is hit).

While some of these issues could be worked around or solved, especially the first one became critical for the development of Lipizzaner, as the system became highly error-prone, a situation additionally aggravated by the inability to debug it.

Asynchronous web services with Flask As PyTorch's distributed package has been shown to be infeasible, different alternatives remained, including a global cache database all nodes would access, or the usage of message buses. While those options would be feasible for production systems, they suffer from similar disadvantages as PyTorch's implementation; generally, the systems complexity would increase, as additional components or servers would be required. As the communication is limited to only a few requests per generation, web services were implemented as a lightweight alternative.

In its current implementation, each Lipizzaner application maintains a server and a client component (shown in Figure 3.3), and is therefore able to both request and transmit the current state of the instance's populations and optimizers.

While requesting data from a server can be done in one statement using either included Python functionality or the more versatile `requests` package, the available

solutions to implement the server differ. Lipizzaner uses Flask[6], a lightweight framework to implement HTTP web services in Python. It allows to specify multiple endpoints with different HTTP methods (like `GET`, `POST` and `DELETE`) that can then be accessed by HTTP clients. Flask is very easy to use and configure and requires nearly no additional boilerplate code; defining the endpoints and starting the application are usually the only required steps.

While this has been shown to work well in our scenario, it has a severe Python-related disadvantage that requires special attention: HTTP requests have to be handled asynchronously, especially when the main thread of the client is busy for very long times during heavy-weight, complex training processes. While this could be easily solved with threading in most modern programming languages, Python follows a different approach and does not provide true asynchronous threading (i.e. no usage of multiple cores or threads, that most state-of-the-art machines would provide). Due to the GIL (*Global Interpreter Lock*), only one thread can be executed in Python code at a time – which is necessary because the memory access of CPython (the default python interpreter) is not thread-safe. This leads to many computationally expensive context switches and drastically reduced performance, when multiple threads try to execute high workloads simultaneously.

There are multiple possible solutions or workarounds for this problem:

1. First, it would be possible to use processes instead of threads. As this issue is a general problem in Python, the multiprocessing system was added, which supports spawning or forking additional processes and provides methods to interact with them. However, sharing data between processes is complicated and slow, and locking resources on process level requires system-wide mutexes – which again significantly reduces performance [49, 67].
2. The most feasible workaround for us was to limit the workload of asynchronous operations that are not executed in the main worker thread to perform very lightweight operations only. Lipizzaner clients therefore constantly maintain two representations of the current population and optimizer states; the PyTorch tensors on the GPU that are used for the training process, and exchangeable, encoded data objects that can be queried by other nodes at any time without additional transformation effort. As this is only done once every time when an individual changes, the workload is even more reduced for typical scenarios where four neighbor nodes access each client. Additionally, allowing high timeouts or safely handling failed requests is also crucial, as even these lightweight operations may not be scheduled soon enough on very rare occasions.

The points above illustrate that running web services in Python is not really optimal, but sufficient for the scenario we currently use them in. As the communication components are completely encapsulated, they might be exchanged by either message buses or a global caching system in future versions of Lipizzaner.

[6] http://flask.pocoo.org

Single Page Applications with ASP.NET Core

Even if Python is well-suited for machine learning applications, it has several disadvantages regarding the requirements to web applications – like the non-asynchronous threading described above. While it theoretically would have been possible to use it for the analytics dashboard, the required implementation effort would have been significantly higher than with a framework like ASP.NET Core, which was specifically designed for this use case.

Like many web applications, the dashboard consists of both a back- and a front-end component, with the first being responsible for accessing the experiment result data from the log database, and the latter for loading this data from the back-end and presenting it to the user with some additional control logic to select different experiments.

Developing such systems with the technology stack used here is highly simplified by ready-to-use templates for most relevant technology stacks Microsoft distributes with ASP.NET Core. Even if a Flask back-end could fulfill the same tasks as ASP.NET Core, assembling the parts of the resulting application is much easier in ASP.NET Core applications, as it is done completely automatically by a predefined environment setup tool.

ASP.NET Core The back-end of the Lipizzaner dashboard was realized with ASP.NET Core[7], a relatively new framework for developing web applications. However, even if of similar age as PyTorch, it is far more sophisticated as it was designed as a successor of the ASP.NET framework, and therefore highly profits from the experience gained from developing this.

While ASP.NET Core can technically run with the *full stack .NET framework* as well, it was primarily designed to use the *.NET Core* runtime – which comes with the advantage of being platform-independent, while the full stack framework is limited to Windows. As PyTorch works best on Unix-like environments such as Linux and Mac OS, supporting these operating systems is reasonable for the dashboard application as well (even if it would not have been required, as the two parts of the Lipizzaner systems are not interconnected by anything else than the database they both access).

When used with a Single Page Framework, ASP.NET Core fulfills two tasks:

1. Serving the static HTML, CSS and Javascript files the front end application consists of. This is a trivial task and does not require any additional steps than running a minimal web server.
2. Providing a HTTP web service that offers endpoints for the data the client uses – a task that is very similar to Flask's purpose inside the Lipizzaner framework. To do so, ASP.NET Core creates *routes* that define specific resources. Each time one of these resources is accessed, the respective controller method is called that loads data from the database, transforms it into the required output format, and sends it to the client inside the HTTP response message. If the route is not defined,

[7] https://docs.microsoft.com/en-us/aspnet/core

ASP.NET Core either attempts to map it to a static resource (e.g. a file or an image), or redirects to the main page and hands the task over to the client-side router if no static resource is available as well.

Given the fact that these two tasks are very simple, the resulting back-end application is expectably lightweight and consists merely of one controller and a few model classes that define the structure of the data. All of the remaining application is either defined inside the template, or – in most cases – implicitly assumed by the framework itself (as one of the design principles of ASP.NET Core is to require minimal configuration effort for common application scenarios).

Angular Angular[8] is a component based front-end framework to implement single-page applications (or SPAs) with TypeScript. In contrast to traditional web sites, single-page applications dynamically rewrite the content of the currently displayed page without completely reloading it from the server. Client-side logic is executed in the user's browser and usually only loads required parts of the requested data from the HTTP endpoints the back-end provide, and injects them into its views.

Angular abstracts this concept and takes care of data binding and synchronization, while providing functionality for most common use cases like requesting data from HTTP endpoints as well. An Angular application consists of multiple components, each containing both logic (in TypeScript, which is compiled to JavaScript during the build process), and view markup and style definitions (in HTML and CSS). These components can be grouped to modules to organize large applications – however, one module is usually enough for a relatively small application like the Lipizzaner dashboard. Like many modern frameworks, Angular has dependency injection deeply integrated into all parts of the framework – which makes it easy to encapsulate functionality into services and share them between multiple components of the application (as shown in Figure 3.7).

3.2.2 Distribution of Coevolutionary Systems

As described in Section 3.1.1, distribution and especially scalability was one of the main requirements of Lipizzaner. As the most trivial approach – an all-to-all communication pattern, where each node communicates with each other – does not fulfill the latter requirement, a more suitable solution was necessary. Beside the logical topology, the overlying, physical communication structure had to be defined as well. The implemented approaches to resolve these points are described in this section.

Spatial Grid Topology

Spatial (or *toroidal*) *grids* are a frequently used coevolutionary distribution architecture and have been shown to scale very well for large-scale applications [31, 46], as they

[8] https://angular.io
[9] https://angular.io/guide/architecture

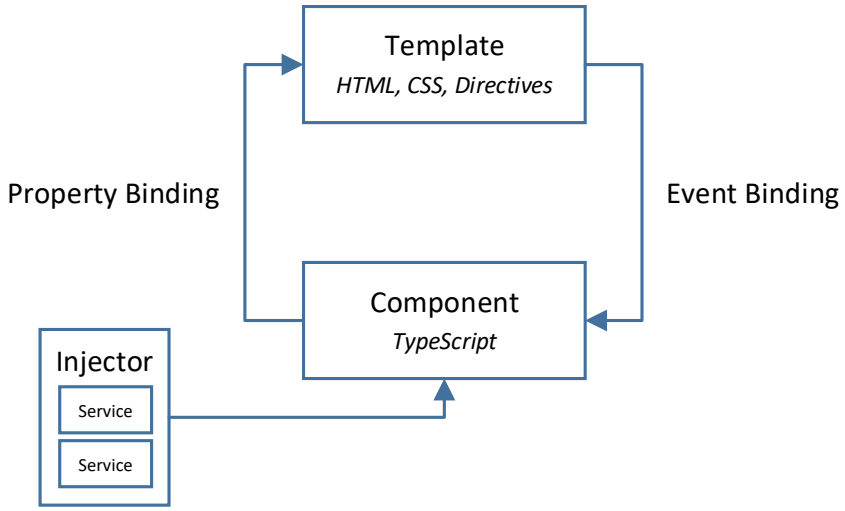


Figure 3.7: Overview of the correlations between the different parts of an Angular application. The component contains the background logic that is needed to populate the template, which accesses the components data by binding element values to its properties. The injector component provides services (like HTTP requesting) to multiple components.[9]

limit the communication between nodes (i.e. grid cells) to local neighborhoods. Adding additional nodes therefore does not increase the load on all other nodes, as it would in all-to-all communication scenarios.

In Lipizzaner, each logical grid cell represents one instance of the application, which can either run on the same or on multiple physical machines, as long as they are connected via a TCP/IP network. When using workstations with state-of-the-art hardware, our experiments have shown that running 4-6 cells per machine is possible, depending on the size and complexity of the training dataset (more details about the experiment configurations can be found in Chapter 4).

In addition to each cell's local generator and discriminator populations, it also holds the hyperparameters and mixture weights that are evolved as well. Figure 3.8 shows a two-dimensional representation of a spatial grid as it is used in Lipizzaner. In a non-two-dimensional perspective, each cell on the edge would be interconnected with the cells on the respective other side of the grid, as by definition a spatial grid has no edges.

As Lipizzaner therefore evolves populations of GANs distributed over the grid instead of a single generator and discriminator pair, each neighborhood returns one *mixture* (or an *ensemble*) for both generators and discriminator after the training process finishes. These mixtures consist of each best generator or discriminator of the respective cells in the neighborhood (i.e. five generators and five discriminators would be returned for a neighborhood size of five). To rate the quality of the resulting mixtures, Lipizzaner

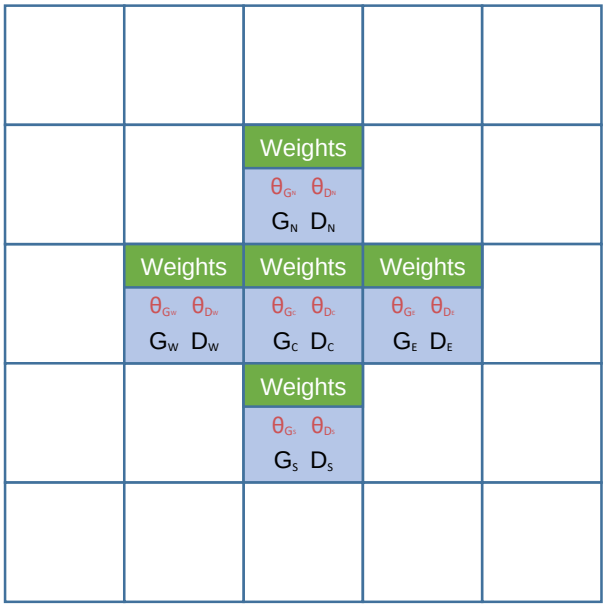


Figure 3.8: Two-dimensional representation of the spatial distribution grid used in Lipizzaner. Each cell holds one generator and discriminator pair, the individuals' respective hyperparameters, and the mixture weights that are evolved. While in this figure each cell contains exactly one generator and discriminator individual, it is possible to use higher cell population sizes as well. The colored cells show one of the 25 neighborhoods on this grid (at position $(2, 2)$).

uses either the *Inception Score* [52] or the *Fréchet Inception Distance* (or FID) [25], quality metrics that rate a model's output both according to its variety and how meaningful the generated images are. This is done by comparing the output to the *inception model*, which is pre-trained on the ImageNet dataset [56]. In addition, the FID also measures the distance to the original dataset.

Master/Client Architecture

While the spatial grid architecture leads to a fully scalable distributed system that can be expanded as needed, using a completely distributed architecture without any central master node would have made its usage very complicated, as it would have been necessary to start each experiment manually on each node – which is practically impossible in large scenarios. Using single-run client applications would furthermore drastically increase the complexity of deployments and experiments, as it would be necessary to manually copy configuration files and start the applications on each machine. Lipizzaner clients therefore run as background services and only have to be started once.

Since Lipizzaner was primarily designed to scale very well in large scenarios, implementing a master-client topology was therefore crucial to fulfill this requirement.

Long-running clients can be easily added to the ecosystem by just starting the application on more nodes, running pre-configured virtual machines, or with Docker (see Section 3.2.2). Using a single point of configuration furthermore simplifies the monitoring of the system, as well as gathering results from it after the process finishes (as it is only required to monitor the master node, and not the possibly high number of clients). In this section, both the functionality of client and master – including their underlying algorithms - is described.

Master As the Lipizzaner master (or *orchestrator*) is only meant to control a single experiment, its life cycle is limited to this (in contrast to the client, which is running as a background service and keeps listening for requests after an experiment has finished). The masters functionality can basically be grouped into three tasks, listed in chronological order:

1. At the very beginning, the master has to parse the configuration the user injects into it and determine the way to discover its clients. Their addresses can either be specified hard-coded in the configuration file, or an auto-discovery mode can be used to find all available clients in the network (this is done via a broadcast on a specific port all clients are listening on). As soon as the clients are discovered, the master furthermore checks if they are available, as they may currently be busy with other experiments. If the result complies with the requirements (i.e. enough clients are available, and the grid is a square), the master transmits the experiment to all of them. If the requirements are not fulfilled, the experiment is terminated.
2. As soon as the experiment is started on all clients, the master begins to periodically check their state, i.e. to verify if the experiment has finished, or if the client may not be reachable anymore. If the latter occurs, the master can currently be configured to show two different behaviors: first, the missing client can be ignored, as some missing nodes are often not critical for the result of an experiment, especially on large grids (this is the default behavior). Second, it is possible to quit the entire experiment on all nodes, which may be especially useful on smaller grids.

   In future releases of Lipizzaner, the client could try to replace lost nodes, e.g. by detecting new ones and transmitting the state of a neighboring node to the newly connected one. This is especially interesting in combination with Docker, as Docker Swarms (which are used in Lipizzaner) provide the functionality to restart containers if they terminate unexpectedly (see Section 3.2.2).
3. When the experiment finished on all (remaining) nodes, the master gathers their results – those include the generator and discriminator mixtures of each neighborhood and their respective inception or FID scores – and saves them to its hard disk in a restorable file format. The mixtures are finally ranked by their scores, and sample images are created and presented to the user (as described in Section 3.2.2).

The master application's procedure is described in Algorithm 3.1.

Algorithm 3.1: Main procedure of the Lipizzaner master application.

```
1:  LipizzanerMaster(path)                                   ▷ path to config file
    Runs the experiment on the specified clients, if available.
2:      config ← parseConfiguration(path)
3:      if config.autoDiscover then
4:          clients ←findAvailableClients()                          ▷ via broadcast
5:      else
6:          clients ← config.clients
7:      end if
8:      if allAvailable(clients) ≠ True then                         ▷ request client status
9:          return −1          ▷ end process if one or more clients are not available
10:     end if
11:     for all client in clients do                    ▷ start experiment on all clients
12:         startExperiment(client, config)
13:     end for
14:     repeat
15:         done ← True
16:         for all client in clients do
17:             state ← requestState(client)
18:             if state.isDead and config.cancelOnOutage then
19:                 cancelExperiments(clients.except(client))            ▷ cancel experiment
20:                 return −2
21:             else if state.done ≠ True then
22:                 done ← False
23:             end if
24:         end for
25:     until done = True
26:     results ← loadResults(clients)                  ▷ request results over HTTP
27:     saveResults(results)
28: end
```

All three tasks described above do not require much computation power, as the master application is meant to run directly on the user's machine. The minimal computation effort also makes it possible to orchestrate multiple experiments at once from a single point of configuration.

Client From a logical perspective, each client represents one cell in the spatial grid topology. While the master applications are bound to single experiments, clients can be compared to services or daemons, meaning that they are long-running background applications that are constantly listening for experiment requests. Currently, for the sake of simplicity, each instance of the client application can process only one experiment at a time. However, it is possible to run multiple clients per host machine at once, either as "normal" applications or inside multiple docker containers (see Section 3.2.2).

An experiment request contains all configuration options a user passes into the mas-

Program 3.1: Simplified excerpt of the class containing the HTTP endpoint definitions of each Lipizzaner instances server component.

```
class ClientAPI:
  app = Flask(__name__)

  @staticmethod
  @app.route('/experiments', methods=['POST'])
  def run_experiment():
    config = request.get_json()

    lipizzaner = Lipizzaner()
    lipizzaner.run(config)

    return Response()

  @staticmethod
  @app.route('/parameters/discriminators', methods=['GET'])
  def get_discriminators():
    discriminators = ConcurrentPopulations.instance().discriminators
    data = json.dumps(discriminators)
    return Response(response=data, status=200,
                    mimetype="application/json")

  @staticmethod
  @app.route('/parameters/generators', methods=['GET'])
  def get_generators():
    generators = ConcurrentPopulations.instance().generators
    data = json.dumps(generators)
    return Response(response=data, status=200,
                    mimetype="application/json")

  def listen(self, port):
    ConcurrentPopulations.instance().lock()
    self.app.run(threaded=True, port=port, host="0.0.0.0")
```

ter application, as it is basically forwarded to the client – with the benefit that the client applications do not have to be re-deployed for each changed experiment configuration. These configurations for example include algorithmic parameters (such as learning rate, population size, etc.), data loader specifications (the underlying dataset, batch size and count), and information about the logical topology of the grid. The latter is needed for the client to compute its position on the grid, and furthermore to determine its neighboring nodes.

The typical behavior of the client can again be divided into three steps:

1. If no experiment is active, the client runs as a background service and listens for experiment requests from a master on a specific port. As soon as an experiment

is requested, the client parses the received configuration file and executes the specified training algorithm (see 3.2.3). It furthermore requests data from the neighboring nodes each time the algorithm accesses the respective properties of the `ConcurrentPopulations` class, which means that the algorithm itself does not have to take care whether it is running in a distributed environment or not.

2. To make this possible, the Lipizzaner client provides HTTP endpoints via the web service mentioned in the previous step simultaneously to executing the training process. Other clients can access these endpoints and therefore request the current populations and optimizer parameters. As stated above, this can be critical in Python, as it supports no real multi-threading – when extremely costly experiments are running, accessing the interface may therefore be rather slow. This issue can be worked around by reducing the computational effort of each request, and preparing the required data packets in advance.

   Program 3.1 shows a slightly simplified version of the HTTP endpoints a Lipizzaner client offers.

3. It is not necessary to actively notify the master application when an experiment finishes, as the master has to monitor the clients activity by polling it anyhow to make sure they are still alive. The master therefore notices that an experiment has finished, and collects the results from the clients for further usage. After this, the client changes its state from `Busy` to `Idle` and waits for new experiment requests.

   The complete logs, resulting generator and discriminator models, and intermediate states and data are preserved on the client, as this information may be necessary to gain full insight into the results, especially if any error occurs during the training process.

Docker

Orchestrating large experiments in a failsafe way is a complex task when it comes to high numbers of clients, especially when working with non-persistent cloud platform systems – as the environment setup takes some time for each experiment. There are different ways this can be automated, either with scripts and vendor-specific command line interface tools, or by using Docker[10].

Docker is a software for container-based virtualization, an abstraction method where applications are put into precompiled container images that can later be distributed and executed on any host machine. In contrast to classical virtual machines, containers are more lightweight, as they make use of virtualization functionality of the host's underlying operating system, and do not emulate physical hardware as virtual machine hypervisors do. Container-based virtualization is therefore also called *application-level* virtualization.

[10] https://www.docker.com

Algorithm 3.2: Main procedure of the Lipizzaner client application.

```
1: LipizzanerMaster
   Listens for incoming requests to start experiments, retrieve the applications state, interchange populations, and query results, and hosts the underlying training algorithm.
2:     isBusy ← False
3:     isDone ← False
4:     while True do                                          ▷ run as background service
5:         isBusy ← waitForExperimentRequest()
6:         if request.type = startExperiment then             ▷ New experiment requested
7:             master, config ← parseRequest(request)
8:             if isBusy = True then
9:                 sendDeclineMessage(master) continue
10:            end if
11:            isBusy ← True
12:            isDone ← False
13:            results ← async startExperiment() ▷ asynchronously run the training algorithm
14:        else if request.type = stateRequest then           ▷ master requests current state
15:            respond(isBusy, isDone)
16:        else if request.type = populationsRequest then     ▷ other client requests population
17:            populations, optimizerParams ← loadCurrentState()
18:            respond(populations, optimizerParams)
19:        else                                               ▷ master requests experiment results
20:            if isDone = True then
21:                respond(results)
22:                isBusy ← False                             ▷ allow new experiment requests
23:            else
24:                respond(−1)                                ▷ respond with error
25:            end if
26:        end if
27:    end while
28: end
```

This behavior leads to drastically reduced overhead and makes Docker containers similarly fast as non-virtualized applications. Docker therefore comes with several advantages, which are especially useful in the context of distributed systems:

- As stated above, Docker's performance is comparable to that of non-virtualized applications – but in contrast to those, Docker containers can be configured to access only specific percentages of CPU cycles and memory, which simplifies running multiple applications with different requirements on one node.
- Using container-based virtualization highly increases deployment speed and quality, as Docker containers are defined by *Dockerfiles* – plain text files that can be put under version control. Besides their configuration, Docker containers have versions to guarantee reproducible results – which is an important feature for research applications as well.

- Docker containers are portable to any other host system that run the Docker daemon, making them independent from the host's operating system. This reduces the environment setup costs, as only docker has to be installed (a task that can be done with `docker-machine`, a command line tool that works with most cloud providers and creates ready-to-use Docker host virtual machines).
- Finally, Docker can be used in different ways to automate the orchestration of complex systems with multiple instances of different container applications by using *swarms* and the *compose* functionality.

Lipizzaner utilizes most of the above advantages, with especially the last one being of high benefit. Both local GPU clusters and Amazon's AWS[11] cloud platform have been used to perform the experiments described in Chapter 4. Orchestrating large numbers of nodes was a crucial part of the implementation itself. We used a combination of Docker swarms – which allow the one-point orchestration of replicated services on multiple connected *swarm nodes* – and `docker-machine` – a command line tool with drivers for AWS (and other cloud platforms) that offers commands for fast deployments and the setup of container hosts. Program 3.2 shows relevant parts of the deployment script used to create AWS instances and deploy multiple Lipizzaner containers onto them.

3.2.3 Trainers

The trainer is the core component of each Lipizzaner instance, as it defines and hosts the executed evolutionary process. The current implementation of Lipizzaner supports two different types of trainers; gradient-free trainers for evolution strategies and NES, and one trainer that updates the network parameters with gradient-based optimizers (i.e. the "main" Lipizzaner algorithm).

All trainers of both types are incorporated into the coevolutionary framework and were specifically implemented to train GANs, as they evolve populations of generators and discriminators. As initially stated, all components – including the trainers themselves – are fully exchangeable, a feature that is especially interesting for large experiments; additionally to networks and datasets, the type of the underlying GAN can be changed as well. We used this possibility in our experiments to first explore the system's advantages to the classic GAN by Goodfellow et al. [19], and further used the more stable Wasserstein GAN [3] for even better results as well. As most GAN types primarily differ by the way they update the models' weights and their fitness evaluation, this can easily be done for other types as well by simply injecting the necessary functionality into the respective trainer (as most training functionality is not class-specific, but based on interfaces passed to the constructor – a concept known as *Dependency Injection*, as described in Section 3.1.1).

While the exact training steps may differ slightly depending on the trainer and used GAN type, all options share the same coevolutionary baseline procedure:

[11] https://aws.amazon.com

Program 3.2: Relevant parts of the script that deploys and runs a five by five Lipizzaner grid with 25 instances on AWS.

```
#!/bin/bash

node_count=5; instances_per_node=5; instance_type=p2.xlarge; image_id=ami-XXXXX

create_vm() {
  name=$1
  docker-machine create -d amazonec2 --amazonec2-instance-type ${instance_type} --
    amazonec2-ami ${image_id}  ${name}
}

start_containers() {
  docker pull alfa/lipizzaner:latest
  for j in $(seq 1 ${instances_per_node})
  do
    docker run -d --rm -e role=client --runtime=nvidia --network lpz-overlay
    tschmiedlechner/lipizzaner:latest
  done
}

# Create swarm manager
create_vm lpz-mgr
master_ip=$(docker-machine ip lpz-mgr)

eval $(docker-machine env lpz-mgr)
docker swarm init --advertise-addr ${master_ip}
docker network create -d overlay --attachable lpz-overlay
token=$(docker swarm join-token worker -q)
start_containers

# Create nodes and add them to swarm
for i in $(seq 1 $((node_count-1)))
do
  create_vm lpz-node-${i}
  eval $(docker-machine env lpz-node-${i})
  docker swarm join --token ${token} ${master_ip}:2377

  # Start containers on client node
  start_containers
done
```

1. **Selection** is used to create an intermediate population from individuals of the previous generation. The default selection method in Lipizzaner is tournament selection, where tournaments between multiple individuals determine if an individual is used as a seed for the generation.
2. **Variation** is used to alter individuals in the coevolutionary process. Depending

on the algorithm, this is either done by mutation, a natural gradient step, or a stochastic gradient step.

3. **Fitness evaluation** determines the quality of an individual. In GANs, the fitness depends on the respective opponent individual (e.g. a discriminator for a generator individual, and vice versa). This fitness evaluation is one of the main reasons for the complex and often unpredictable dynamics of GANs.
4. **Replacement** is the final step performed in each generation, where the algorithm decides if an individual is promoted to the next generation or discarded due to worse performance than the previous individuals showed.

   As GANs often exhibit a near-oscillating behavior, tuning the replacement step is crucial for coevolutionary GANs. Promoting only individuals with higher fitness than previous ones might ultimately lead to premature convergence because of rapid loss of diversity. Lipizzaner supports different methods that are partially elaborated in the experiments described in Chapter 4. However, our experiments showed that replacing large parts of the population leads to good results, as this behavior is most similar to typical GAN behavior. In spatial distributions, where we use a population of only one individual <per grid cell, we replace the individual regardless of its fitness value.

Additionally, all implemented trainers split the dataset into mini batches (usually around 100 data samples per batch), which is a commonly used approach to fit the training data into the memory of the used system.

Gradient-Free Trainers

Previous research has shown that large-scale neuroevolution for reinforcement learning is able to match the performance of state-of-the-art gradient-based algorithms [54]. Consequently, the initial idea behind Lipizzaner was to elaborate the possibilities of training GANs with gradient-free methods as well. This is primarily interesting due to several advantages of evolutionary methods, especially their ability to be parallelized and distributed easily.

While our experiments show that plain gradient-free approaches show partial success on low-dimensional problems, GAN dynamics seem to be too complex to train with global optimizers. Evolutionary searches for larger networks with approximately 3.5 million parameters per model did not show any progress even with long runtimes and high computational effort.

This section contains details about the two gradient-free approaches Lipizzaner supports: evolution strategies and natural evolution strategies (NES). In contrast to the Lipizzaner algorithm, which is distributed using a logical spatial grid, both algorithms described here are only locally parallelized, as most effort was put into the gradient-based trainer during implementation after the drawbacks of using plain gradient-free methods became apparent.

Evolution Strategies Evolution strategies (ES) are a subclass of evolutionary algorithms that are primarily used to evolve real-valued problem representations [9]. Evolution strategies have been shown to be able to perform well in neuroevolution in previous research, e.g. for reinforcement learning [51].

This trainer is based on (μ, λ) evolution strategies; this means that the trainer maintains a population of μ individuals, of which it creates λ child elements per generation by adding Gaussian noise to the parent that got selected (e.g. by tournament selection). These children are then ranked according to their fitness, and the μ best individuals are kept for the next generation, etc. Similarly, the Lipizzaner algorithm uses $(1 + 1)$ evolution strategies to evolve hyperparameters and mixture weights (as described in Section 3.2.3).

Two different versions of the ES trainer were implemented in Lipizzaner to explore their different behaviors [34]:

- The **sequential** version of this trainer takes turns in evolving generators and discriminators; each population therefore competes with the newest generation of its respective opponent. Even if both populations take one step after another, it is possible to perform larger steps for each one before the respective other is updated; a possibility we use in the Lipizzaner algorithm as well, as our experiments show that faster convergence can be reached by skipping generator steps (similar to the approach the Wasserstein GAN uses [3]).
- In contrast to this behavior, the parallel evolution strategies trainer evolves both populations concurrently in each step, with the advantage of simpler behavior and reduced implementation effort.

Program 3.3 shows the sequential implementation of the training algorithm as it is implemented in Lipizzaner. Details of the parallel version are accessible via the source code repository on GitHub[12].

As we elaborate in Chapters 4 and 5, ES exhibit interesting behavior on low-dimensional problems, such as estimating a model containing a small number of Gaussian distributions. However, for larger problems that require the usage of more complex neural networks, ES do not show signs of convergence during the training in our experiments.

Natural Evolution Strategies Natural evolution strategies (NES), as introduced by Wierstra et al. in 2008 [61], rely on the general concept of evolution strategies. In contrast to them, NES aim to estimate the *natural gradient* and updates individuals with respect to the fitness values of each offspring individual. Using the natural gradient instead of the plain gradient has the advantage of adjusting the convergence speed to the current slope of the fitness landscape [2]. This approach takes all offspring individuals into account, instead of only selecting the one with the currently optimal fitness, which also prevents early convergence to local optima.

[12] https://github.mit.edu/ALFA-CSec/lipizzaner_gan_distributed_tom

Algorithm 3.3: Sequential ES trainer, as implemented in Lipizzaner. **Input:**
X : Input dataset $\mathcal{L}$: Loss function γ : Population size
δ : Mutation rate β : Mutation probability τ : Tournament size

```
1:  SequentialES
2:      G_0, D_0 ← initializePopulations(γ)
3:      for t = 1; t < T; t++ do                          ▷ for each generation
4:          G_t ← G_{t−1}
5:          D_t ← D_{t−1}
6:          for batch in X do                             ▷ process mini batches
7:              G' ← select(G_t, τ)                       ▷ tournament select generators
8:              G' ← mutate(G', δ, β)                     ▷ Gaussian mutation
9:              sort(G')
10:             if L(G'_0) < L(G_{t,γ}) then
11:                 G_{t,γ} ← G'_0                        ▷ replace worst generator with best new one
12:             end if
13:
14:             D' ← select(D_t, τ)                       ▷ tournament select discriminators
15:             D' ← mutate(D', δ, β)                     ▷ Gaussian mutation
16:             sort(D')
17:             if L(D'_0) < L(D_{t,γ}) then
18:                 D_{t,γ} ← D'_0                        ▷ replace worst discriminator with best new one
19:             end if
20:             sort(G_t)
21:             sort(D_t)
22:         end for
23:     end for
24:
25:     return G_{t,0}, D_{t,0}                           ▷ return best GAN pair
26: end
```

In Lipizzaner, the initial implementation of NES [61] was altered in two different ways, enabling its usage in sequential and parallel coevolutionary setups [34]:

- The **sequential** implementation evolves one population after another, i.e. uses the resulting element of population g as an opponent during the fitness calculation for all individuals of the next generation of population d. The advantage of this approach is its lower number of function evaluations, as only one previous individual competes with the ones from the new generation. Its disadvantage lies in the comparatively smaller search space, as population pairs that would perform best together may not be found (as only one individual of each step is kept).
- The **parallel** coevolutionary implementation of NES is similar to the sequential one, with the difference that populations are evolved synchronously and each in-

dividual is compared to each one of the opposing population. This leads to higher chances of finding well-matching pairs, with the cost of a higher number of function evaluations ($\mathcal{O}(n^2)$ versus $\mathcal{O}(n)$).

As before, only the formalization of the slightly more complex sequential implementation of coevolutionary NES is shown in Algorithm 3.4. The implementation of the respective parallel version again can be found in the GitHub repository of Lipizzaner[13].

Algorithm 3.4: Sequential NES trainer, as implemented in Lipizzaner.
Input:
X : Input dataset $\mathcal{L}$: Loss function λ : Population size
δ : Mutation rate β : Mutation probability

```
1:  SequentialNES
2:      G_0, D_0 ← initializeGAN(δ)                    ▷ initialize a single GAN pair
3:      for t = 1; t <= T; t++ do                       ▷ for each generation
4:          for batch in X do                           ▷ process mini batches
5:              G' ← mutate(G_t, δ, β, λ)               ▷ create generators by Gaussian mutation
6:              γ_G ← L(G', D_t)                        ▷ calculate reward for each generator
7:              G_t ←summarize(G', γ_G)                 ▷ update with weighted permutations
8:
9:              D' ← mutate(D_t, δ, β, λ)               ▷ create discriminators by Gaussian mutation
10:             γ_D ← L(D', G_t)                        ▷ calculate reward for each discriminator
11:             D_t ←summarize(D', γ_D)                 ▷ update with weighted permutations
12:         end for
13:     end for
14:     return G_T, D_T                                 ▷ return resulting GAN pair
15: end
```

Lipizzaner Algorithm

Neuroevolution has been shown to achieve remarkable results even for complex tasks such as reinforcement learning in recent literature [51, 54]. While the initial objective of Lipizzaner therefore was to train GANs with completely gradient-free trainers, our first experiments showed that neuroevolution is not applicable for most mid- to large-scale scenarios. Solving the complex adversarial dynamics of GANs with more than 3.5 million parameters per neural network seems to be unachievable with unguided global optimization in reasonable time. This aligns with the results of previous research, in which extreme amounts of computational power and time were needed to compete with gradient-based optimizers even for non-adversarial use cases [54].

However, our experiments showed as well that coevolutionary algorithms are able to overcome critical behaviors that typically occur when GANs are trained with gradient-

[13] https://github.mit.edu/ALFA-CSec/lipizzaner_gan_distributed_tom

based algorithms: *mode collapse* – a scenario in which the generator focuses on certain modes of the target distribution, and the discriminator distinguishes real from fake data by the remaining modes, and *discriminator collapse* – where the discriminator gets stuck in local optima. The objective of Lipizzaner therefore ultimately was to combine the advantages of coevolutionary algorithms with gradient-based optimizers to benefit from their respective strengths:

- Gradient-based optimizers like Adam [36] have been shown to work very well in numerous applications of GANs [19, 28] in terms of training performance and time. As Lipizzaner is implemented to supported interchangeable trainers, using different optimizers is supported as well. This is required, as other GAN implementations like the Wasserstein GAN rely on those optimizers [3].
- Coevolutionary algorithms have two advantages we are primarily interested in. First, their population-based approach is able to prevent otherwise critical GAN behavior like mode and discriminator collapse. Second, coevolutionary systems have been shown to scale very well, especially in combination with a spatial grid distribution. As Lipizzaner is meant to be used in large-scale scenarios in which datasets might be too large for single nodes, this feature is crucial for the system as well.

The resulting algorithm combines these advantages and is loosely based on the previously introduced sequential evolution strategies algorithm, which was altered in several aspects:

- The previously unguided mutation operator was changed to increment the weights with the gradient-step delta computed by the optimizer instead of random Gaussian noise. This leads to drastically increased convergence speed and makes it possible to utilize recent advantages of gradient-based optimizers [36].
- The majority of the most commonly used optimizers rely on hyperparameters such as the learning rate, which is usually fixed during the whole training process. However, even if many momentum-based optimizers like Adam take steps to avoid overstepping optima in the fitness landscape, reducing the learning rate as the training progresses has been shown to improve the quality of the results by even better convergence [48]. In Lipizzaner, these individual-bound hyperparameters are therefore included into the evolutionary process as well and evolved mutually with the neural network parameters.
- Finally, to combine the resulting mixtures of generators, the algorithm evolves weights that determine the probability of drawing samples from the respective individual. In contrast to the previously described behavior, the mixture weights are not bound to individuals, but to respective nodes (i.e. each cell on the grid contains one vector of mixture weights, which are then evolved cooperatively).

This behavior is described in detail in Program 3.5.

Algorithm 3.5: `Lipizzaner`: Evolve distributed populations of generators $\mathcal{G}$ and discriminators $\mathcal{D}$ and the hyperparameters competitively, and the separate evolution of mixture weights ω.

Input:

T : Number of iterations E : Grid cells k : Neighborhood size

θ_{EA} : Parameters for `stepMixtureEA` θ_{COEV} : Parameters for `stepGANCoev`

1: **parfor** $c \in E$ **do** ▷ Asynchronous parallel execution of all cells in grid
2: $t \leftarrow 0$
3: $n, \omega \leftarrow$ initializeNeighborhoodAndMixtureWeights(c, k) ▷ Uniform initialization of settings
4: **for** $t \leq T$ **do** ▷ Iterations
5: $n \leftarrow$ `stepGANCoev`(n, θ_{COEV}) ▷ Coevolve GAN using Alg. 3.6
6: $\omega \leftarrow$ `stepMixtureEA`(ω, n, θ_{EA}) ▷ EA for mixture weights, Alg. 3.7
7: $t \leftarrow t + 1$
8: **end for**
9: **end parfor**
10: **return** $(n, \omega)^*$ ▷ Cell with best generator mixture

3.2.4 Analysis

In contrast to monitoring applications that run only a single instance of a program, analyzing distributed systems is often a difficult task that cannot be solved by manually interpreting log files and console outputs. Lipizzaner therefore includes a separate web application to display both current and previous experiment results and states, the *Lipizzaner dashboard*.

As mentioned above, a different technology stack than for the Lipizzaner framework itself was used to implement this application. Using ASP.NET Core in combination with predefined templates greatly simplified this task and allowed to focus most development effort on the framework, i.e. the main contribution of this thesis.

In addition to the description of the dashboard, this section also contains implementation details about the central log server (which connects the Lipizzaner framework and dashboard), and the used tooling – as this is often crucial in web development.

Log Server

When running multiple distributed experiments in different execution environments – local machines, public clouds, etc. – it soon becomes apparent that a more sophisticated logging system than using plain textual log files is required. Lipizzaner therefore uses a MongoDB[14] database server for these purposes due to several reasons:

- The primary reason to use MongoDB to store logs is the fact that it is schemaless, i.e. not bound to table definitions as for SQL databases. This is advantageous, as log messages are hard to generalize; additional fields may be necessary for specific

[14] https://www.mongodb.com

Algorithm 3.6: `stepGANCoev`: Evolve distributed populations of generators $\mathcal{G}$ and discriminators $\mathcal{D}$ and the hyperparameters α competitively.

Input:

τ : Tournament size γ : Replacement size X : Input dataset
β : Mutation probability n : Cell neighborhood

1: $n' \leftarrow \text{select}(n, \tau)$ ▷ Select based on fitness ($\mathcal{L}$)
2: $\mathbf{B} \leftarrow \text{getMiniBatches}(X)$ ▷ Load minibatchs
3: **for** $B \in \mathbf{B}$ **do** ▷ Loop over batches
4: $\alpha \leftarrow \text{mutateLearningRate}(\alpha, \beta)$ ▷ Update with with gaussian mutation
5: $D \leftarrow \text{getRandomOpponent}(n'_D)$ ▷ Get uniform random discriminator
6: **for** $G \in n'_G$ **do** ▷ Evalute generators
7: $\nabla_G \leftarrow \text{computeGradient}(G, D, \alpha_G)$ ▷ Compute gradient for neighborhood center
8: $G \leftarrow \text{updateNN}(G, \nabla_G, B)$ ▷ Update with gradient
9: **end for**
10: ▷ Now for discriminator
11: $G \leftarrow \text{getRandomOpponent}(n'_G)$ ▷ Get uniform random generator
12: **for** $D \in n'_D$ **do** ▷ Evalute discriminator
13: $\nabla_D \leftarrow \text{computeGradient}(D, G, \alpha_D)$ ▷ Compute gradient for neighborhood center
14: $D \leftarrow \text{updateGAN}(D, \nabla_D, B)$ ▷ Update with gradient
15: **end for**
16: **end for**
17: **for** $G, D \in n'_G \times n'_D$ **do** ▷ Evalute GANS
18: $\mathcal{L}_{D,G} \leftarrow \text{evaluate}(D, G, B)$ ▷ Evaluate GAN
19: **end for**
20: $\mathcal{L}_G \leftarrow \min(\mathcal{L}_{\cdot,G})$ ▷ Fitness is the worst loss ($\mathcal{L}$)
21: $\mathcal{L}_D \leftarrow \min(\mathcal{L}_{D,\cdot})$ ▷ Fitness is the worst loss ($\mathcal{L}$)
22: $n_G \leftarrow \text{replace}(n_G, n'_G, \gamma)$ ▷ Replace the worst generator
23: $n_D \leftarrow \text{replace}(n_D, n'_D, \gamma)$ ▷ Replace the worst discriminator
24: **return** n

Algorithm 3.7: `stepMixtureEA`: Evolve mixture weights ω.

Input:

μ : Mutation rate n : Cell neighborhood ω : Mixture weights

1: $\omega' \leftarrow \text{mutate}(\omega, \mu)$ ▷ Gaussian mutation of mixture weights
2: $\omega'_f \leftarrow \text{evaluateMixture}(\omega', n)$ ▷ Evaluate generator mixture inception score
3: $\omega \leftarrow \max(\omega', \omega)$ ▷ Replace if new mixture weights are better
4: **return** ω

events that are not used for other log entries. SQL databases can be tuned to support this use case as well, but this usually leads to drawbacks like empty fields or non-atomic field values.

- MongoDB is very lightweight and easy to install, especially when the available preconfigured Docker image is used. This drastically reduces the effort for setting up the database to a single command line call.

Program 3.3: Writing an experiment entry to MongoDB with its Python connector library.

```
def create_experiment(self, settings):
  master_ip = local_ip_address()
  name = settings['general']['distribution']['start_time']
  grid_size = len(settings['general']['distribution']['client_nodes'])

  experiment = {
    'name': name,
    'master': master_ip,
    'topology': {
        'type': 'grid',
        'grid_size': grid_size
    },
    'settings': settings
  }

  database = MongoClient(server_address, serverSelectionTimeoutMS=CONNECTION_TIMEOUT
    ).lipizzaner_db
  collection = database.experiments

  return str(collection.insert_one(experiment).inserted_id)
```

- Finally, like most NoSQL databases, MongoDB is generally very fast, as it does not have to enforce consistency or data type constraints. While this was not necessary for the small- to medium-scale experiments we describe in Chapter 4, it may be relevant for upcoming work that involves even higher numbers of clients and larger datasets.

Both the Lipizzaner framework and dashboard access the log database, with the main difference that the framework is only writing data, while the dashboard is primarily reading it – with the exception of marking experiments as deleted. This is done by adding an additional `deleted` flag into the respective entry, and not by actually deleting the record. This behavior guarantees that no relevant data is irretrievably lost in case of a faulty operation and is generally considered a good practice when working with databases [11].

As MongoDB offers native connector libraries for both Python and C#, implementing the database communication logic is a simple task in both environments. Programs 3.3 and 3.4 display the usage of both libraries in Lipizzaner.

Dashboard

Similar to most state-of-the-art web applications, the Lipizzaner dashboard is composed of two loosely coupled components: the back-end that serves static files and dynamic data, and the front-end, which is responsible for the presentation of data to the user.

Program 3.4: Reading a strongly typed list of experiments from MongoDB with its C# connector library from a Web API controller method.

```
[HttpGet]
public IEnumerable<Experiment> GetExperiments()
{
  // Exclude experiment results, as they are not needed here
  var builder = Builders<Experiment>.Projection;
  var fields = builder.Exclude(d => d.Results);

  // Connect to the 'experiments' collection
  var db = GetDatabase();
  var collection = db.GetCollection<Experiment>("experiments");

  // Return only non-deleted experiments
  return collection.Find(x => !x.IsDeleted).Project<Experiment>(fields).ToList();
}
```

Back-end The Lipizzaner dashboard uses an ASP.NET Core web application for its back-end tasks, with the underlying .NET Core framework. Using .NET Core is advantageous over building new web applications on the full .NET framework, as it is platform-independent and more lightweight to implement in general. Only if previous components would exist that have to be included into a web project, .NET Core might not be an option – however, this is not the case in Lipizzaner, as the dashboard was built without any pre-existing dependencies.

The logic contained in the back-end application is very simple and contains only one web API controller – i.e. one group of API endpoints that serves experiment and log entry data. ASP.NET Core maps relative paths like `api/experiments` to their respective methods, which then return the requested data as JSON. An example for a controller method is shown in Program 3.4.

The Lipizzaner back-end controller directly pulls the log data from MongoDB, which makes it possible to monitor the current state and health of experiments while they are running. As several experiments were executed on AWS, monitoring the state was crucial to detect errors and blocked experiments and therefore reduce the costs incurred.

Front-end The dashboard's front-end – which is responsible for presenting the experiments to the user – is an Angular Single Page Application (SPA) and implemented in TypeScript. In contrast to the back-end logic, which is executed on a server machine primarily because of security reasons, as databases should never be directly accessed from client computers, the resulting JavaScript code the compiler generates is executed and rendered directly in the user's browser – with the advantage of reducing additional load on the server machine. A screenshot of the user interface is shown in Figure 3.9.

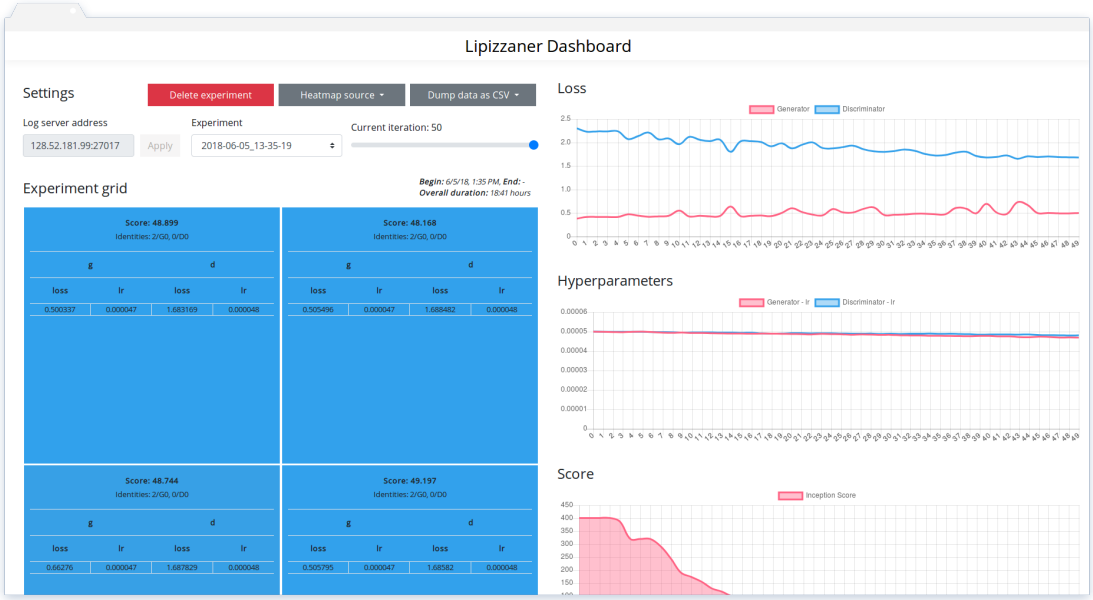


Figure 3.9: Screenshot of the Lipizzaner dashboard web application. The navigation component is rendered on the left, the details component on the right side of the screen.

The front-end application consists of one Angular module, which is furthermore grouped into two (visual) components:

- The **grid navigation** component is responsible for loading and displaying the experiment selection dialog elements, and furthermore – if an experiment was selected – shows details about its configuration, topology, and execution time frame. If an experiment has finished, samples from the resulting mixtures are shown in this component as well.

  Additionally, it is possible to scroll through training generations while displaying a live heat map of the grid. This was implemented to monitor if individuals propagate through the grid, a behavior we explored during the experiments described in Chapter 4.
- When a grid cell of a specific experiment is selected, the **details** component is loaded and displays drill-down information about the whole experiment history of the respective Lipizzaner application instance. This includes charts for loss, hyperparameters, mixture weights and score values. Intermediate generator output images for each generations are displayed as well, together with real data from the input dataset – which is helpful to visually rate the current experiment state.

As briefly mentioned above, a characteristic feature of SPAs is that the website is not reloaded during navigation. The Lipizzaner dashboard therefore uses the Angular routing package to dynamically update the website.

Tooling Because of the immensely growing number of frameworks, libraries and packages available to implement web applications, using specific tools to compose and handle those dependencies has become necessary to focus on the development process instead of manually organizing references. The Lipizzaner dashboard is based on the ASP.NET Core SPA template for Angular, which includes a full development environment for both back- and front-end applications:

- The **.NET Core CLI** is a command line tool that supports all development steps of .NET core applications, e.g. building, packaging and deploying them, and pulling and installing packages. The CLI is invoked with the `dotnet` command, and is also used to run the deployed application on the production environment by providing a minimal web server that can be executed on all relevant platforms (Windows, Linux, Mac OS, etc.).
- Similar to this, the **Angular CLI** – with the `ng` command – basically provides the same functionality for building, deploying and packaging Angular front-end SPAs.
- **NuGet** and **NPM** are package managers, with the former being responsible for back-end .NET packages, and NPM being used for front-end packages like the Bootstrap web style framework and Angular itself. While NPM has to be separately invoked with the `npm` command, NuGet is directly embedded into the .NET Core CLI.

The template furthermore combines all these functionalities into pre-defined `dotnet` build steps, and even invokes the Angular CLI to build and deploy the front-end. The whole application can therefore be built and run by invoking `dotnet run` from the root directory.

Chapter 4

Experiments

This chapter contains configuration details and results of the experiments that were conducted both in preparation for implementing Lipizzaner and to evaluate its final performance on state-of-the-art problems and datasets. It is organized as follows:

1. The first section of the chapter covers the steps taken to evaluate the applicability and advantages of coevolutionary algorithms on GAN problems.
2. The second section contains details about the results Lipizzaner yielded on image datasets like MNIST [40], CIFAR10 [39] and CelebA [43]. In the first part, the experiments conducted with gradient-free coevolutionary trainers are elaborated; the second part covers information about promising results we achieved by combining gradient-based optimizers with a coevolutionary framework.

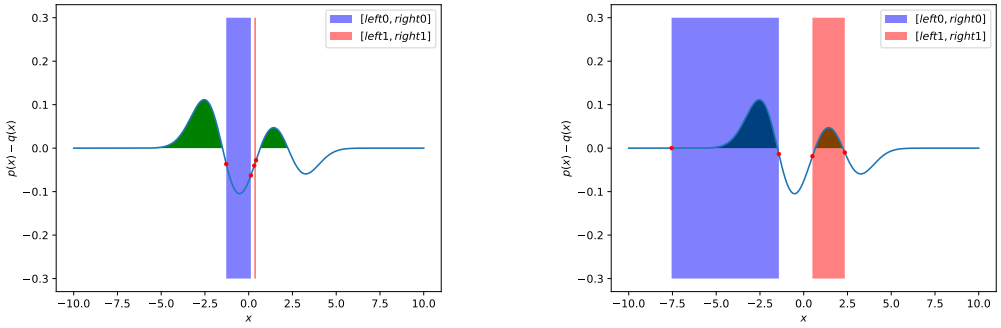
The respective experiment configuration files are listed in Appendix A.

Overall, the results show that Lipizzaner's combination of gradient-based optimizers and coevolutionary algorithms is able to perform very well on the conducted experiments. Lipizzaner is able to overcome or even prevent otherwise critical behavior and collapsed GANs even at very small grid sizes, and scales well to larger sizes without noticeably increasing the required runtime.

4.1 Synthetic Data

Preceding the implementation of Lipizzaner, experiments were conducted to elaborate the applicability of coevolutionary algorithms on GAN training. We thereto make use of a similar setup as described by Li et al. [41], where the generator has the objective of estimating two Gaussian distributions in a one-dimensional space, while the discriminator aims to distinguish real and generated fake data by changing its boundaries (the method is described in more detail in Section 4.1.2).

As those experiments yielded promising results in this small scale, they can be seen as the foundation of the work realized afterwards.



(a) Generation 1 **(b)** Generation 100

Figure 4.1: (a) Preliminary stage of discriminator collapse. Both boundaries are set to a local minimum of the fitness function, so training algorithms would have to initially decrease the fitness even more to step out of this scenario. (b) Coevolutionary algorithms are able to escape this situation without further optimizations.

4.1.1 Motivation

The main motivation behind the experiments described in this section was to elaborate if coevolutionary algorithms are able to stabilize of GANs regarding otherwise critical behaviors in a small, analyzable scale before the implementation of a larger system.

As the dynamics of GANs are still not completely understood [41], these behaviors are often hard to reproduce and even harder to fully understand and associate to their theoretical foundations. Using a simplified setup without underlying neural networks made it possible to eliminate incalculable factors and focus on the behavior of interest – namely *mode-* and *discriminator collapse*.

Mode collapse describes a phenomenon in which the generator focuses on specific modes of the target distribution – in the scenario used in these experiments, this is equal to focusing either only on μ_1 or μ_2. On the other hand, *discriminator collapse* occurs in scenarios where the generator is able to fool the discriminator very well, leading to the latter being stuck in a local minimum of the fitness landscape. Figure 4.1a shows the preliminary stage of this phenomenon; the discriminators objective is to maximize the positive area it covers in the given scenario, but to do so, it firstly would have to increase this area to overcome the local minimum.

4.1.2 Setup

To investigate coevolutionary dynamics for GAN training, we make use of the simple problem introduced in [41], in which no neural networks are used. Instead, the generator's objective is to estimate two Gaussian distributions $\mathcal{N}(\mu_1, 1)$ and $\mathcal{N}(\mu_2, 1)$.

Table 4.1: Setup of GAN dynamics experiments.

Generations	100
Mutation rate	1
Mutation probability	0.7
Population size	10
Tournament selection size	2
Number of replacements	5
Initialization range	[-0.2, 0.2]
Discriminator boundaries	[-10, 10]
μ_1^*	-2
μ_2^*	2

Formally, the set of possible generators is therefore defined as

$$\mathcal{G} = \left\{ \frac{1}{2}\mathcal{N}(\mu_1, 1) + \frac{1}{2}\mathcal{N}(\mu_2, 1) \mid \boldsymbol{\mu} \in \mathbb{R}^2 \right\} . \tag{4.1}$$

On the other hand, the discriminator set is expressed as follows.

$$\mathcal{D} = \{ \mathbb{I}_{[\ell_1, r_1]} + \mathbb{I}_{[\ell_2, r_2]} \mid \boldsymbol{\ell}, \boldsymbol{r} \in \mathbb{R}^2 \text{ s.t. } \ell_1 \leq r_1 \leq \ell_2 \leq r_2 \} . \tag{4.2}$$

Given a true distribution G_* with parameters $\boldsymbol{\mu}^*$, the GAN objective of this simple problem can be written as

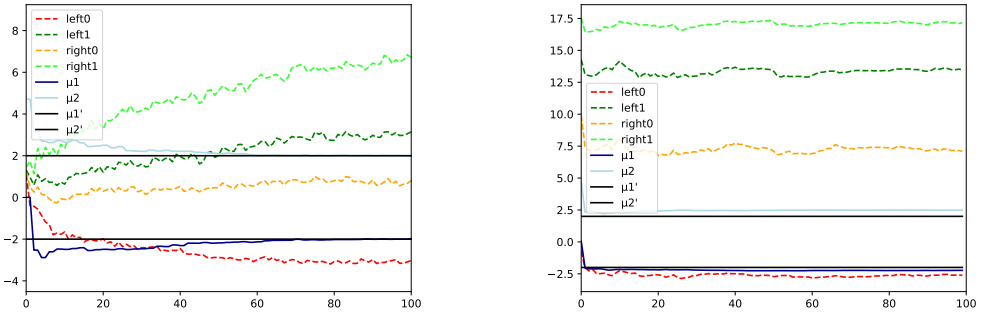
$$\min_{\boldsymbol{\mu}} \max_{\boldsymbol{\ell}, \boldsymbol{r}} \mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\ell}, \boldsymbol{r}) \text{ , where}$$

$$\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\ell}, \boldsymbol{r}) = \mathbb{E}_{x \sim G_*}[D_{\boldsymbol{\ell}, \boldsymbol{r}}(x)] + \mathbb{E}_{x \sim G_{\boldsymbol{\mu}}}[1 - D_{\boldsymbol{\ell}, \boldsymbol{r}}(x)] . \tag{4.3}$$

While being easy to understand and demonstrate, this simplified GAN variant exhibits the relevant dynamics we are elaborating. Unless stated otherwise, we ran each experiment of this group for 120 runs, each with 100 generations and a population size of 10. We also use Gaussian mutation with a learning rate of 1 as the only genetic operator, as no others were needed to succeed in the respective tasks. The full experiment setup is shown in Table 4.1.

4.1.3 Results

This sections shows the results gathered from the conducted experiments, and puts them into relation to comparable work done with pure gradient-based GANs on similar or equal tasks [5, 41]. Overall, the results illustrate that gradient-free coevolutionary algorithms are reliably able to escape both mode and discriminator collapse, including scenarios where gradient-based optimizers are guaranteed to fail.



(a) Alternating Coevolution **(b)** Parallel Coevolution

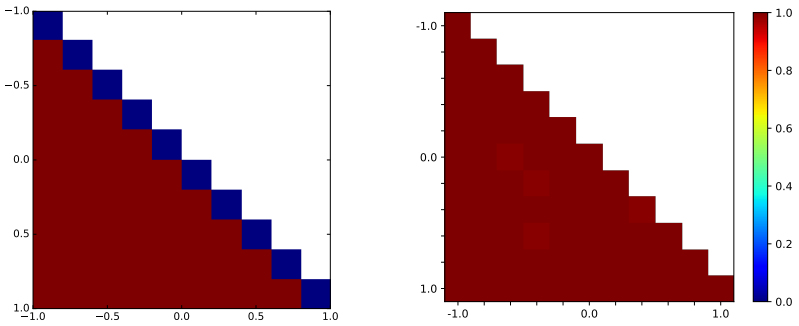
Figure 4.2: Mean results over 120 runs of (a) the alternating and (b) parallel implementation of coevolutionary algorithms in a simplified GAN setup with synthetic Gaussian data. While both perform well under the given circumstances, (a) yields slightly better results and converges more reliably.

Two different types of experiments were conducted to prove the applicability of coevolution on this task. First, we show the method is working for randomly set initial values (limited to a small initialization interval to make the problem more complex). Second, we elaborate the abilities of coevolutionary algorithms to resolve otherwise critical scenarios like mode and discriminator collapse.

General Results As expected, coevolutionary algorithms perform well in general scenarios in this experiment, and are able to match the results of gradient-based algorithms [41]. Both parallel and alternating implementations were evaluated (as described in Section 3.2.3).

The respective results are shown in Figure 4.2; while both algorithms are able to solve the given problem, the alternating implementation performs better and reaches convergence faster and more reliably than the parallel approach. Based on these experiments, we attribute the superiority of the alternating algorithm to its closer proximity to GANs, which strongly rely on the not yet fully understood adversarial interactions between generator and discriminator [19]. As Lipizzaner is also derived from the alternating implementation, the remaining results in this section are therefore focused on this approach.

Mode Collapse Figure 4.3 shows that coevolutionary GAN training is reliably able to step out of mode collapse scenarios, in contrast to the already optimized gradient-based *optimal training* introduced in [41], whose results are shown in the left part of the figure. The plot illustrates the average success rate over 120 runs with the respective initialization values for μ_1 and μ_2, which are plotted on the x and y axis. In this context, *success* is defined as the ability to reach a distance between the best generator of the last



(a) Optimal Training **(b)** Coevolutionary Training

Figure 4.3: Mean probability over 120 runs to escape a mode collapse scenario (where $\mu_1 = \mu_2$) of (a) optimal training as described by Li et al [41], and (b) coevolutionary training in a simplified GAN setup with synthetic Gaussian data. Initial values for μ_1 and μ_2 are plotted on the x and y axis. While even the computationally expensive optimal training (which relies on finding the optimal discriminator in each iteration) is not able to escape mode collapse, coevolutionary training does not exhibit any problems in these scenarios.

generation and optimality of < 0.1. While the *optimal training* method clearly fails on the diagonal representing collapsed modes (i.e. where $\mu_1 = \mu_2$), coevolutionary training performs equally well for all initialization values.

In addition to this, further experiments were executed to elaborate the influence of the *optimal training* approach mentioned above [41], where the discriminator does not perform real gradient-steps, but is set to the optimal values in each iteration. As expected, this even further decreases convergence time of coevolutionary training, as shown in Figure 4.4.

Discriminator Collapse The experiment results shown in Figure 4.5 show similar success rates for escaping scenarios in which the discriminator is initialized in a collapsed state. This is done by setting either one or both of the discriminator boundaries ($l1$ and $r1$, and $l2$ and $r2$) to a negative, convex position in the two-dimensional fitness landscape (for example, in Figure 4.1 both boundaries are initialized in a negative position – i.e. the worst case scenario). As overcoming these situations requires to initially further decrease the fitness, local optimizers are naturally not able to escape them, while global optimizers like the evolutionary algorithms used in this experiment explore viable solutions very quickly in this simplified setup.

4.2 Image Data

Motivated by the results shown in the previous section, the first implementation steps of Lipizzaner were focused on gradient-free neuroevolution. Evolution strategies (ES) and

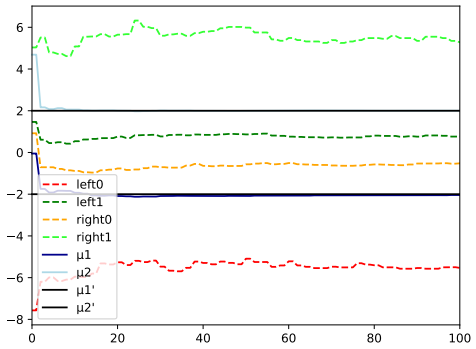


Figure 4.4: Mean results over 120 runs for alternating coevolution with fixed optimal discriminator training in a simplified GAN setup with synthetic Gaussian data. Similar to the method proposed in [41], the optimal discriminator bounds for the given point in time are calculated in each generation. As expected, this increases the convergence speed of coevolutionary algorithms in a similar way it does when using gradient-based optimizers.

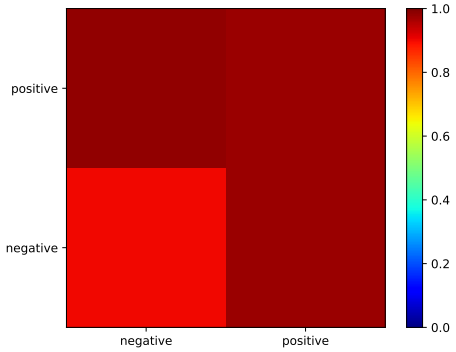


Figure 4.5: Mean probability over 120 runs of escaping discriminator collapse scenarios in a simplified GAN setup with synthetic Gaussian data. Each quadrant represents a different type of initialization, in which either none, one or both boundaries are placed in a local minimum of the fitness function.

natural evolution strategies (NES) were used to mutate the neural network parameters of the models (i.e. biases and weights). As described in Section 3.2.3, ES operate on a completely gradient-free basis, while NES uses the *natural gradient*.

Unfortunately, the conducted experiments show that, while yielding some results on lower-dimensional data, global optimizers are not performing well in the more complex GAN scenarios of interest that include the optimization of deep networks with millions of parameters. The second development iteration of Lipizzaner was therefore focused on gradient-based optimizers and their combination with coevolutionary algorithms, as

described in Chapter 3. A detailed analysis of the results – both of gradient-free and gradient-based experiments – can be found in Chapter 5.

4.2.1 Gradient-Free Coevolutionary Algorithms

Both evolution strategies and natural evolution strategies generally did not perform well on most experiments described in this section. As more complex image generation tasks did not yield any results at all, this section is focused on results for datasets that showed signs of convergence, i.e. a two-dimensional dataset for ES and the MNIST digit dataset for NES experiments.

Evolution Strategies

Multiple experiments with different hyperparameters were conducted to apply plain gradient-free coevolutionary algorithms to high-dimensional problems. Despite these efforts, neither the alternating nor the parallel implementation of evolution strategies showed any signs of convergence in this task. As very high numbers of parameters are necessary to generate e.g. visual content for these high-dimensional datasets (i.e. 3.5 million parameters per neural network [48]), we assume that global search algorithms are not able to perform these tasks in reasonable time.

However, small-scale experiments were conducted to prove the general applicability of the approach. In these experiments, the target distribution consisted of multiple points (or *modes*) in a two-dimensional space, and a simple multilayer perceptron (MLP) with 42 neurons was used. The full experiment setup is shown in Table 4.2.

Although the generator focused on different points in this distribution, it was not able to escape mode collapse and kept oscillating between different modes – presumable due to the relatively low diversity, which again was limited due to computational effort and time. Figure 4.6 shows different representative generations of the training process.

It has to be noted that the observed problems could probably be solved with the usage of more sophisticated GAN implementations [3] or by fine-tuning the used hyperparameters. However, as the method did clearly not scale to higher-dimensional problems, this approach was discarded as well.

Natural Evolution Strategies

As evolution strategies yielded no noticeable results on the MNIST datasets and only showed signs of convergence for lower-dimensional data distributions, it seemed reasonable to use a faster, guided evolutionary approach. Natural evolution strategies (NES) follow the *natural gradient* – an approach in which multiple offspring individuals are created, and the actual mutation of the single main population individual is computed with respect to their errors (see Section 3.2.3).

In contrast to the experiments described in the section above, a more complicated multilayer perceptron was used (with two hidden layers, and approximately 1150 neurons

Table 4.2: Setup for experiments conducted with evolution strategies, a two-dimensional target distribution, and a multilayer perceptron with one hidden layer.

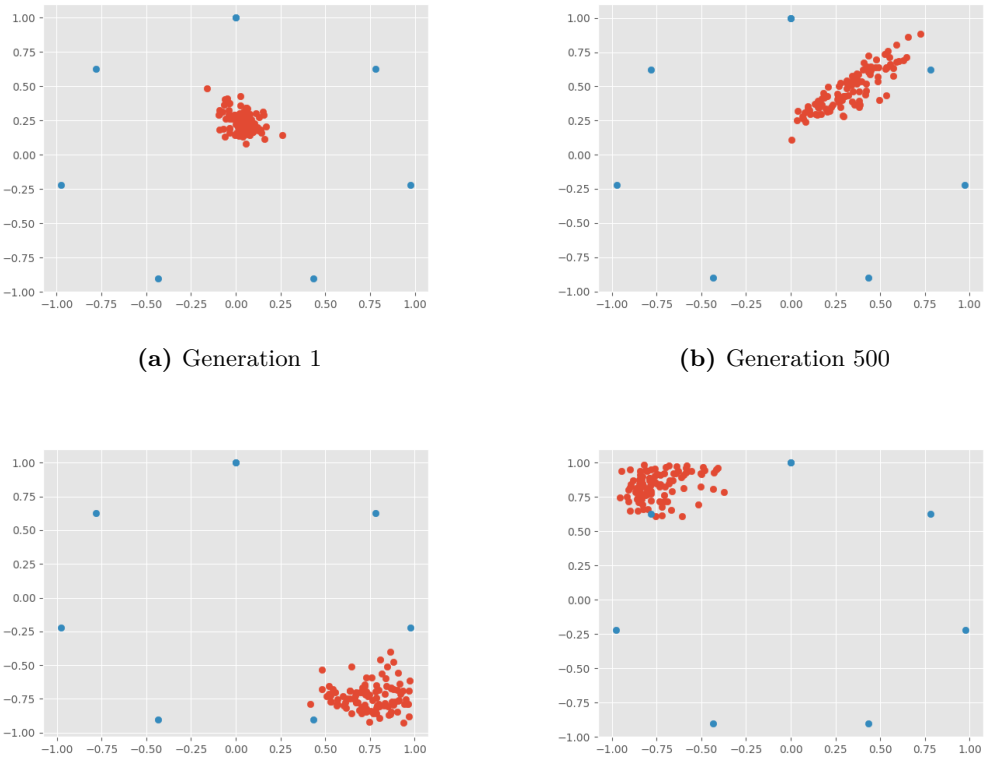
Coevolutionary settings	
Generations	1000
Mutation size	0.0001
Mutation probability	0.7
Population size	50
Number of replacements	10
Tournament selection size	2
Network topology	
Input neurons	20
Number of hidden layers	1
Neurons per hidden layer	20
Output neurons	2 (x and y)
Activation function	$tanh$

Table 4.3: Setup for experiments conducted with natural evolution strategies, the MNIST dataset [40], and a multilayer perceptron with two hidden layers.

Coevolutionary settings	
Generations	5000
Mutation size	0.0005
Population size	50
Network topology	
Input neurons	100
Number of hidden layers	2
Neurons per hidden layer	256
Output neurons	784 (gray-scale pixels)
Activation function	$tanh$

overall). As experiments with typical gradient-based learners showed, this deeper neural network is the minimum requirement to learn the MNIST dataset within a GAN. The full experiment setup is shown in Table 4.3.

As illustrated in Figure 4.7, this method shows initial signs of convergence after 2000-2500 generations. However, the quality of the generated results stops improving around this point in the evolutionary process, most likely due to the relatively small population size in combination with a rather high learning rate. These settings are nevertheless necessary to make progress in the evolutionary process in reasonable time. The training process shown in Figure 4.7 took more than 24 hours on a state-of-the-art machine, while a “normal” gradient-based GAN was able to produce near-perfect



(a) Generation 1 **(b)** Generation 500

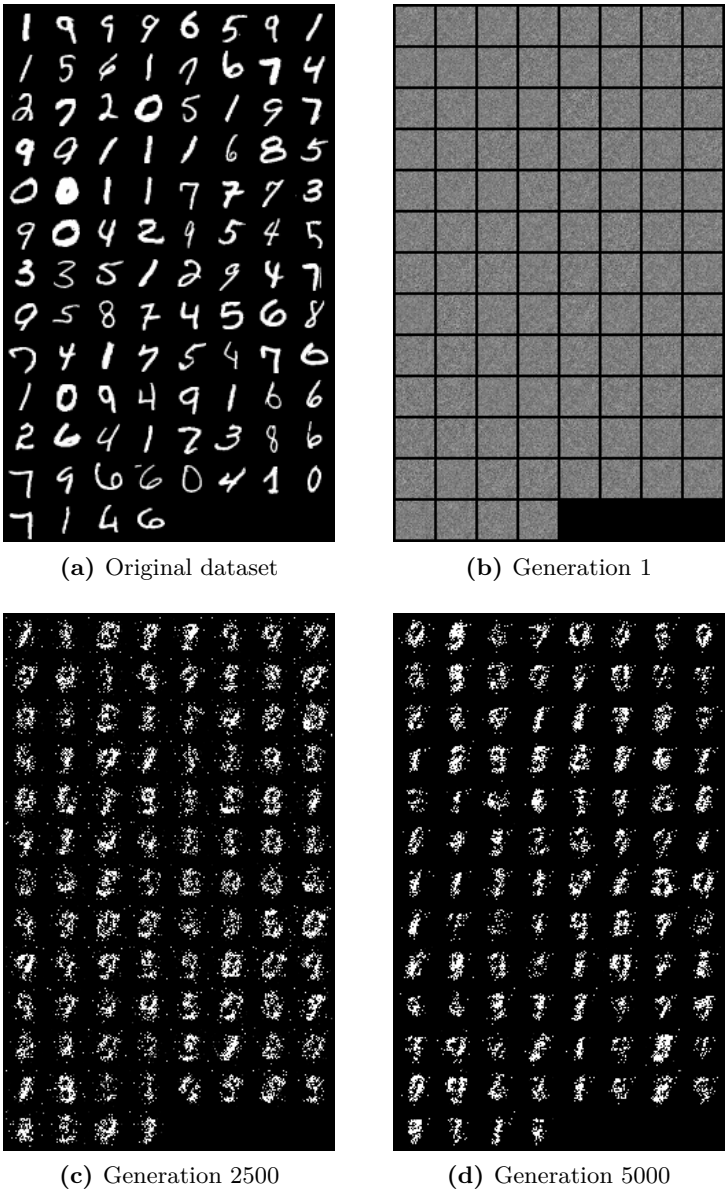
(c) Generation 750 **(d)** Generation 1000

Figure 4.6: Sample results of the experiments conducted with the non-distributed Lipizzaner evolution strategies trainer on a two-dimensional dataset. Even if the algorithm is able to converge to different modes of the target distribution, it is not able to escape this mode collapse scenario.

results in only minutes (even if it encountered the mode collapse scenario, as described in Section 4.2.2).

4.2.2 Gradient-Based Coevolutionary Algorithms

In contrast to the gradient-free approaches described in the section above, the Lipizzaner algorithm – which incorporates gradient-based optimizers into the coevolutionary process – shows both good performance in terms of speed and image quality, and stability against typical critical GAN pathologies like mode and discriminator collapse. To illustrate these behaviors, this section is split into two parts: first, the results achieved by a single Lipizzaner application instance with a population of one generator and discriminator is shown for comparison. Second, results from distributed runs with higher population sizes are presented.



(a) Original dataset **(b)** Generation 1

(c) Generation 2500 **(d)** Generation 5000

Figure 4.7: Typical results of the non-distributed Lipizzaner NES trainer on the MNIST dataset. Even if the evolutionary process shows signs of convergence, the quality of the produced images does not increase after a given point.

As previously stated, Lipizzaner is able to incorporate different GAN implementations. While quantitative results for the more sophisticated Wasserstein GAN (WGAN) [3] are presented in the following sections as well, the experiments described here were conducted with a "classic" GAN, as introduced by Goodfellow et al. [19]. This approach makes it easier to observe the critical pathologies and limitations of GANs, which are more complicated (but not impossible) to reproduce with other implementations.

Table 4.4: Setup for experiments conducted with the coevolutionary, gradient-based Lipizzaner algorithm, the MNIST dataset [40], and a multilayer perceptron with two hidden layers.

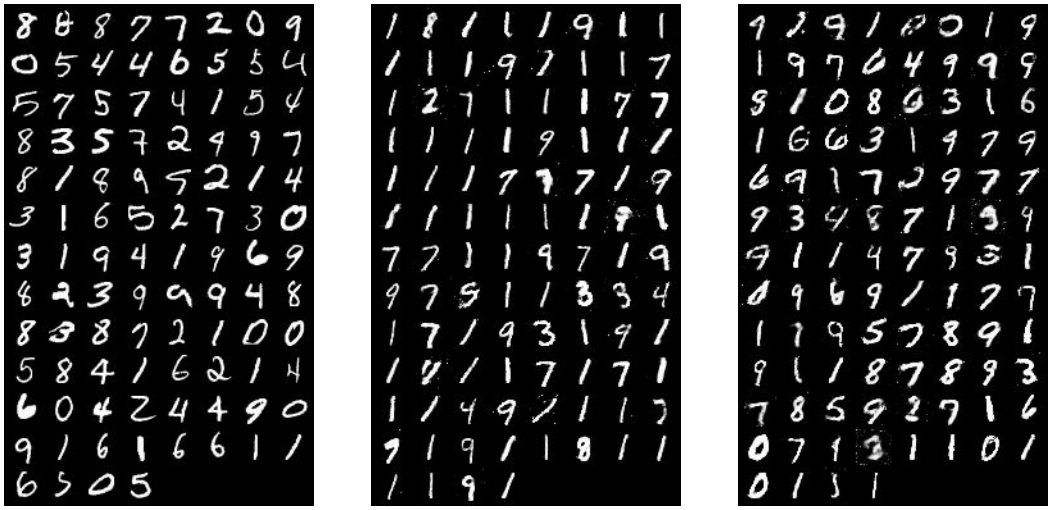
Coevolutionary settings		
Generations	400 (600 batches each)	30 (1600 batches each)
Population size per cell	1	1
Tournament size	2	2
Grid size	1x1, 2x2	1x1, 2x2
Mixture size	$1/n_{cells}$ (fixed)	0.01 (learning rate)
Hyperparameter mutation		
Optimizer	Adam	Adam
Initial learning rate	0.002	0.002
Mutation rate	0.0001	0.0001
Mutation probability	0.5	0.5
Network topology		
Network type	MLP	DCGAN
Input neurons	100	100
Number of hidden layers	2	4 (convolutional)
Neurons per hidden layer	256	16,384 − 131,072
Output neurons	784 (gray-scale pixels)	64x64x3 (RGB pixels)
Activation function	*tanh*	*tanh*
	MNIST	**CelebA**

MNIST

Training a generator on the MNIST dataset [40] – which contains 60,000 examples of handwritten digits – is a relatively trivial task, primarily because the target distribution consists of only monochrome pixels and is limited to ten modes (0−9). However, because of these characteristics, it is very well suited to demonstrate one of the critical GAN behaviors of interest, namely mode collapse.

As shown in Figure 4.8b, single instances of GANs reproducibly collapse to the digits one, seven and nine, presumably due to their similarity to the other digits. In contrast to this, Figure 4.8c shows that using a grid size of 2x2 (i.e. four Lipizzaner application instances) is enough to prevent this behavior in our experiments.

It should be noted that the pixel artifacts in the resulting image occur due to the relatively trivial multilayer perceptron used for these experiments. Also, a fixed mixture rate was used, as calculating the inception score to rate mixture qualities is not feasible for the MNIST dataset. All setup parameters are described in Table 4.4.



(a) Original dataset **(b)** Single instance **(c)** 2x2 grid

Figure 4.8: Samples from (a) the original dataset, (b) generated by a single GAN, and (c) generated by a mixture gathered from a 2x2 Lipizzaner grid after 400 generations. While the single instance results are mostly collapsed to the digits one, seven and nine, using multiple instances leads to more diverse results.

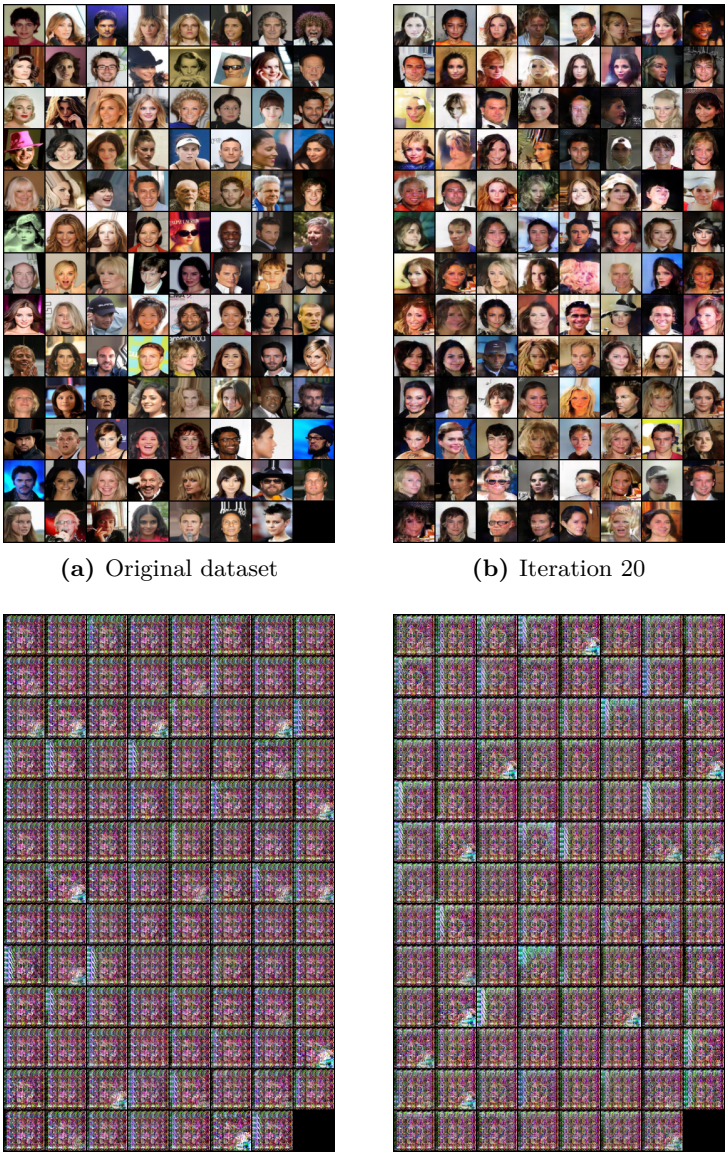
Celebrity Faces

The CelebA dataset [43] contains 200,000 portraits of celebrities and is commonly used to rate quantitative results in GAN literature. The reason for this is that generating images of a specific class (i.e. faces) is an easier task than using a broader target distribution, which is still an open problem even for state-of-the-art GANs.

This common usage makes it reasonable to evaluate the results of Lipizzaner on this dataset as well. Additionally, training a "classic" GAN on it often leads to a collapsed system that is not able to recover, which again illustrates instability as the primary problem GANs suffer from [19]. For comparison reasons with the results Lipizzaner yields, this behavior is shown in Figure 4.9.

Lipizzaner is able to overcome these otherwise critical behavior even when only the smallest possible grid size (2x2) is used. The increased diversity is enough to replace collapsed individuals in the next generation, and allows the system to even prevent the collapse in most conducted experiment runs. An example for a recovering system is shown in Figure 4.10.

As simple multilayer perceptrons are not sufficient for complex tasks like this, the more sophisticated DCGAN neural network architecture was used [48]. It consists of multiple convolutional layers and is able to generate RGB images of relatively high quality. Again the full experiment setup is presented in Table 4.4.



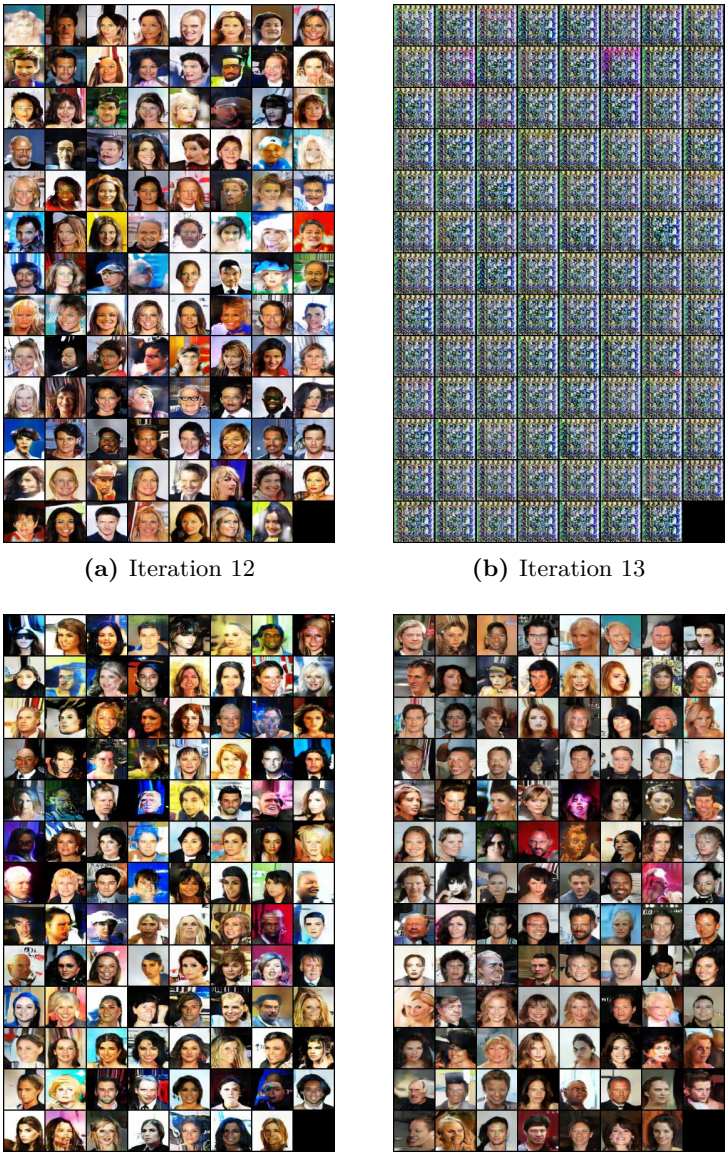
(a) Original dataset **(b)** Iteration 20

(c) Iteration 21 **(d)** Iteration 30

Figure 4.9: Generated output of a collapsed, single-instance GAN trained with the Lipizzaner gradient-based trainer: while the system works fine until iteration 20, it collapses in the following iteration and is unable to recover during the remaining training process.

Scalability and Training Time

Scalability was one of the main requirements of Lipizzaner, and particular importance was therefore placed on it during implementation. Due to the usage of a spatial grid distribution architecture, the required computational effort increases linearly with the number of cells instead of quadratically. In theory, Lipizzaner therefore allows unlim-



(a) Iteration 12 **(b)** Iteration 13

(c) Iteration 14 **(d)** Iteration 30

Figure 4.10: In contrast to the single-node results shown in Figure 4.9, a 2x2 gradient-based Lipizzaner grid is enough to overcome the otherwise critical system collapse observable in (c), and maintains a stable state until the end of the training process.

ited scaling – which is only limited by the amount of available hardware and network bandwidth in practice.

However, as Lipizzaner instances communicate via HTTP web services and exchange only relatively small amounts of data during the training process, it is possible to deploy multiple instances onto different machines and hence scale horizontally (in contrast to vertical scaling, where the hardware capacities of a single machine have to be increased).

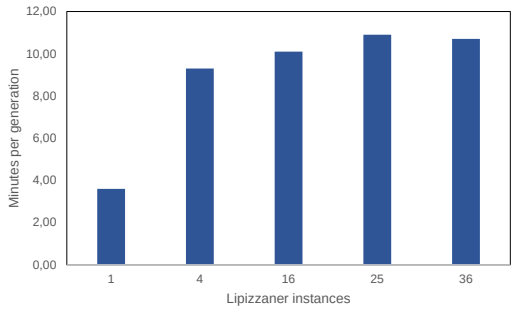


Figure 4.11: Training times on AWS per generation on the CelebA dataset, averaged over 30 generations. The X axis lists the number of Lipizzaner instances (i.e. processes or logical grid cells), while the Y axis shows the average training time per generation in minutes.

This proposition is supported by the chart shown in Figure 4.11, which illustrates a near linear training time per generation for different numbers of connected instances. The initial relatively large step from one to four cells is caused by the fact that four Lipizzaner processes were run per GPU in the distributed experiments; this increases the calculation effort per GPU, and therefore affects the training time as well.

We also observed promising low communication durations in our experiments: exchanging data between two clients only takes 0.5 seconds on average in state-of-the-art Ethernet networks and is only performed once per generation. Additionally, the asynchronous communication pattern between the clients leads to the usage of different time slots and therefore reduces network peak loads.

The experiment results described in Figure 4.11 were computed on AWS GPU cloud instances. Each instance had one Nvidia Tesla K80 GPU with 12 GB memory, 4 Intel Xeon cores with 2.7 GHz each, and 60 GB system memory. The times shown are averaged over 30 generations of training a DCGAN neural network pair on the CelebA dataset. The instances were hosted in Docker containers and connected trough a virtual overlay network.

Chapter 5

Conclusions and Future Work

5.1 Results

As illustrated in Chapter 4, Lipizzaner yielded promising results in the conducted experiments and is able to overcome otherwise critical scenarios like mode and discriminator collapse. In this section, the reasons for these abilities and the general strengths and advantages of Lipizzaner are discussed.

5.1.1 Diversity

The main advantage of incorporating GANs in coevolutionary algorithms is the usage of populations and the therefore increased diversity among the possible solutions. While gradient-based optimizers have been shown to converge faster than other neural network optimization methods like neuroevolution, they also tend to get stuck in local optima. When used with GANs, this is even more critical due to their complex adversarial behavior, and often leads to instable and ultimately collapsed systems. However, the experiments described in Section 4.2.1 show that neuroevolution is currently no viable solution to optimize adversarial network of the required complexity.

The compromise between both methods is the usage of coevolutionary populations of generators and discriminators that are trained with gradient-based optimizers. As all individuals are randomly seeded and therefore evolve into different directions during the training process, weak individuals that are stuck in local optima are not replicated and ultimately removed from the population.

While large population sizes are often needed to achieve good performance in coevolutionary algorithms, our experiments show that even using a relatively small spatial grid is sufficient to overcome the common limitations of GANs, presumably due to two advantages of Lipizzaner's system design:

1. Using a spatial grid instead of non-distributed adversarial populations leads to slower exchange rates between the cells, and therefore may form local "hot spots" that perform extraordinary well in some regions of the fitness landscape. As these

individuals then propagate through the grid, hot spots presumably interact and fuse at some point during the training process.

2. As a result of the asynchronous training processes of the cells in the grid, different cells are often in different stages of the training process (i.e. compute different generations). Individuals from previous or upcoming generations may therefore be used during the training process, which further increases the diversity as well.

While our experiment results support these propositions, the exact propagation behaviors are hard to monitor in real world applications, especially in large grids. Although the Lipizzaner dashboard shows the origin of each individual on the grid, tracing the complete paths of individuals is currently still a difficult and time-consuming manual task.

5.1.2 Scalability

As shown in Section 4.2.2, Lipizzaner scales well up to the grid sizes elaborated in the conducted experiments (i.e. a grid size of six by six). Due to the exhibited linear time consumption, adding more nodes does not affect the system performance; although communication speed in the original system design was considered critical, it did not result in any measurable problems.

Experiments show that, depending on the processed dataset and used neural network architecture, five to eight Lipizzaner instances can be hosted on a single state-of-the-art GPU with 10-12 gigabytes of memory. While the training speed usually depends on the computational power of these GPUs, we observed that CPU power can be critical as well due to Python's multi-threading limitations. As each Lipizzaner process creates multiple threads, peaks of high CPU load may occur during the training process when computationally expensive steps are performed simultaneously. However, this problem was mostly resolved by minimizing the work of background threads as far as possible (as described in Section 3.2.1).

Using Docker drastically reduces the effort to deploy large grids to cloud providers such as AWS, especially when combined with the `docker-machine` command. As most relevant cloud providers such as AWS and Azure are included in these command line interfaces by default, adapting few parameters of a predefined deployment script is sufficient to change both the grid size and target platform. Lipizzaner's auto-discover functionality was particularly optimized to be used in Docker networks, and detects available clients independently from the underlying hardware or cloud provider.

5.1.3 Improved GAN Variants

Since the introduction of GANs in 2014 [19], many improvements to this concept were developed[1]. One of the most noticeable contributions is the *Wasserstein GAN* (WGAN),

[1] An extensive list can be found at https://github.com/hindupuravinash/the-gan-zoo.

which shows promising results in preventing critical GAN behaviors and is underpinned by a strong theoretical basis [3].

As using WGAN makes it significantly harder to provoke collapsed systems (even if it does not guarantee stability), a classic Goodfellow GAN was used initially in Lipizzaner to show that the system is reliably able to prevent or overcome these otherwise critical behaviors. However, to elaborate Lipizzaner's ability to incorporate different GAN implementations, the Wasserstein GAN was added as an alternative, and as expected yielded good results. Incorporating WGANs in Lipizzaner leads to slightly better generated samples and more stable training processes, as the loss curves of generators and discriminators are noticeable less oscillating. However, because of its underlying concept of training the generator in smaller steps than the discriminator, using WGANs significantly increases the training time.

5.2 Conclusions

In this thesis, the underlying theoretical background, the design principles, and the implementation details of Lipizzaner were described and evaluated. Coming back to the research question initially formulated in Chapter 1, Lipizzaner fulfills all included requirements; it is a framework that can be utilized to train generative adversarial networks with the combined advantages of gradient-based optimizers and coevolutionary algorithms, and distribute this training process over a well-scaling spatial grid architecture.

While our initial experiments illustrated that completely gradient-free optimizers are not feasible for training adversarial neural networks of the required complexity, replacing the Gaussian mutation step by gradient-based optimizers not only solves this problem, but also allows Lipizzaner to profit from the variety of GAN improvements published in current literature.

While most of these improved training techniques like WGAN enhance the system's stability, they often come with disadvantages (like the considerable increased training times of WGANs). In contrast to this, the experiments described in Chapter 4 illustrate that Lipizzaner is reliably able to overcome or even prevent otherwise critical behaviors like mode and discriminator collapse without slowing down the actual training process. In addition to these advantages, the usage of a spatial grid distribution architecture further allows linear scalability characteristics, making the system well prepared for future usage in large scale scenarios.

5.3 Future Work

From the very beginning, Lipizzaner was meant to be a foundation for future experiments and projects. Hence, great effort was put into creating a generic framework with interchangeable components to allows reuse and adaptation in this upcoming scenarios.

Generally, several technical improvements are possible to further enhance the system:

- Lipizzaner currently uses simple (1+1) evolution strategies to evolve the mixture weights of each neighborhood; it may be sensible to replace this by more sophisticated evolutionary algorithms like CMA-ES, or evolve a larger population of possible mixture weights [21].
- Observing the desired propagation behavior of individuals through the spatial grid is hard to monitor in complex adversarial scenarios. Adding the possibility to display individuals' complete paths over the whole evolutionary process in the Lipizzaner dashboard therefore would be helpful.
- Finally, adding data loaders that partition data from very large datasets may be required for future experiments. This could either be done by accessing parts of the data from a network share, or by using a standalone data server that handles requests from multiple clients.

Lipizzaner and the findings discovered during its implementation can be used in practice-relevant fields of application, like the domain of cybersecurity. GANs could be used to improve systems that detect malicious binaries (i.e. malware), either by additionally training the respective neural networks on data created by the generator to improve its generalization abilities, or even by using the discriminator for these tasks. Creating artificial, distributable samples from a sensitive dataset could also be a possible application.

Appendix A

Experiment Configuration

A.1 Gradient-Free Trainers

Evolution strategies configuration for two-dimensional dataset.

```
trainer:
  name: alternating_ea
  n_iterations: 1000
  params:
    sigma: 1
    alpha: 0.0001
    population_size: 50
    tournament_size: 2
    mutation_probability: 0.7
    n_replacements: 10
dataloader:
  dataset_name: circular
  use_batch: true
  batch_size: 100
  n_batches: 500
network:
  name: circular_problem_perceptron
  loss: bceloss
general: !include general.yml
```

Listing A.1: es-circular.yml

Natural evolution strategies configuration for MNIST dataset.

```
trainer:
  name: sequential_nes
  n_iterations: 5000
  params:
    sigma: 0.05
    alpha: 0.0005
    population_size: 50
```

```
dataloader:
  dataset_name: mnist
  use_batch: True
  batch_size: 100
  n_batches: 100
network:
  name: four_layer_perceptron
  loss: bceloss
general: !include general.yml
```

Listing A.2: nes-mnist.yml

A.2 Lipizzaner

General configuration file that is included in all experiments conducted with Lipizzaner.

```
logging:
  enabled: True
  log_level: INFO
  log_server: mongodb://user:password@mongodb.local:27017
  image_format: jpgV
output_dir: ./output
distribution:
  auto_discover: False
  master_node:
    address: 128.30.103.19
    port: 4999
    exit_clients_on_disconnect: True
  client_nodes:
    - address: 128.30.103.19
      # For single instance experiments
      port: 5000
      # For multi-instance experiments
      port: 5000-5003
```

Listing A.3: general.yml

Lipizzaner configuration file for MNIST experiments.

```
trainer:
  name: lipizzaner_gan
  n_iterations: 400
  params:
    population_size: 1
    tournament_size: 2
    n_replacements: 1
    default_adam_learning_rate: 0.0002
    # Hyperparameter mutation
    alpha: 0.0001
    mutation_probability: 0.5
    mixture:
      enabled: False
```

```
dataloader:
  dataset_name: mnist
  use_batch: True
  batch_size: 100
  n_batches: 0
  shuffle: True
network:
  name: four_layer_perceptron
  loss: bceloss
general: !include ../general.yml
```

Listing A.4: lpz-mnist.yml

Lipizzaner configuration file for CelebA experiments.

```
trainer:
  name: lipizzaner_gan
  n_iterations: 30
  params:
    population_size: 1
    tournament_size: 1
    n_replacements: 1
    default_adam_learning_rate: 0.0002
    # Hyperparameter mutation
    alpha: 0.0001
    mutation_probability: 0.5
    mixture:
      enabled: True
      sigma: 0.01
      inception_size: 100
      # Warning: Enabled CUDA will use enormous amounts of GPU memory
      cuda: False
dataloader:
  dataset_name: celeba
  use_batch: True
  batch_size: 128
  n_batches: 0
  shuffle: True
network:
  name: convolutional
  loss: bceloss
master:
  calculate_inception: True
  # Same amount of data as original CIFAR contains
  inception_size: 50000
  cuda: True
general: !include ../general.yml
```

Listing A.5: lpz-celeba.yml

References

Literature

[1] Gustavo Alonso et al. “Web services”. In: *Web Services*. Springer, 2004, pp. 123–149 (cit. on p. 14).

[2] Shun-Ichi Amari. “Natural Gradient Works Efficiently in Learning”. *Neural computation* 10.2 (1998), pp. 251–276 (cit. on p. 42).

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein GAN”. *arXiv preprint arXiv:1701.07875* (2017) (cit. on pp. 39, 42, 45, 58, 61, 69).

[4] Sanjeev Arora and Yi Zhang. “Do GANs actually learn the distribution? An empirical study”. *arXiv preprint arXiv:1706.08224* (2017) (cit. on pp. 8, 25).

[5] Sanjeev Arora et al. “Generalization and Equilibrium in Generative Adversarial Nets (GANs)”. *arXiv preprint arXiv:1703.00573* (2017) (cit. on pp. 2, 8, 19, 24, 54).

[6] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996 (cit. on pp. 9–11).

[7] Nils Aall Barricelli. “Numerical Testing of Evolution Theories”. *Acta Biotheoretica* 16.1-2 (1962), pp. 69–98 (cit. on p. 11).

[8] Yoshua Bengio et al. “Learning Deep Architectures for AI”. *Foundations and Trends in Machine Learning* 2.1 (2009), pp. 1–127 (cit. on pp. 1, 6).

[9] Hans-Georg Beyer and Hans-Paul Schwefel. “Evolution Strategies – A Comprehensive Introduction”. *Natural Computing* 1.1 (2002), pp. 3–52 (cit. on p. 42).

[10] Sean P Brydon and Inderjeet Singh. *Web services message broker architecture*. US Patent 7,702,724. Apr. 2010 (cit. on p. 15).

[11] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O’Reilly Media, Inc.", 2013 (cit. on p. 48).

[12] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems*. 2012, pp. 1223–1231 (cit. on p. 5).

[13] Abdullah Al-Dujaili et al. “RECKLESS: A Principled Approach to Black-Box Min-Max Optimization” (2018) (cit. on p. 9).

[14] Ian Fette. *The WebSocket Protocol*. RFC 6455. 2011. URL: https://tools.ietf.org/html/rfc6455 (cit. on p. 15).

[15] Dario Floreano, Peter Dürr, and Claudio Mattiussi. “Neuroevolution: From Architectures to Learning”. *Evolutionary Intelligence* 1.1 (2008), pp. 47–62 (cit. on p. 5).

[16] Martin Fowler. “Inversion of Control Containers and the Dependency Injection Pattern” (2004) (cit. on p. 18).

[17] Dennis Garcia et al. “Investigating Coevolutionary Archive Based Genetic Algorithms on Cyber Defense Networks”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM. 2017, pp. 1455–1462 (cit. on p. 10).

[18] David E Goldberg and John H Holland. “Genetic Algorithms and Machine Learning”. *Machine learning* 3.2 (1988), pp. 95–99 (cit. on p. 2).

[19] Ian Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680 (cit. on pp. 1, 6–8, 19, 39, 45, 55, 61, 63, 68).

[20] Andreas O. Griewank. “Generalized Descent for Global Optimization”. *Journal of Optimization Theory and Applications* 34.1 (1981), pp. 11–39 (cit. on p. 10).

[21] Nikolaus Hansen and Andreas Ostermeier. “Completely Derandomized Self-Adaptation in Evolution Strategies”. *Evolutionary computation* 9.2 (2001), pp. 159–195 (cit. on p. 70).

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. “Unsupervised Learning”. In: *The Elements of Statistical Learning*. Springer, 2009, pp. 485–585 (cit. on p. 6).

[23] Simon S. Haykin. *Neural networks and learning machines*. Vol. 3. Pearson, Upper Saddle River, NJ, USA, 2009 (cit. on pp. 4, 5).

[24] Jeffrey W Herrmann. “A genetic algorithm for minimax optimization problems”. In: *CEC*. Vol. 2. IEEE. 1999, pp. 1099–1103 (cit. on p. 12).

[25] Martin Heusel et al. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”. *arXiv preprint arXiv:1706.08500* (2017) (cit. on pp. 9, 33).

[26] W. Daniel Hillis. “Co-evolving parasites improve simulated evolution as an optimization procedure”. *Physica D: Nonlinear Phenomena* 42.1 (1990), pp. 228–234 (cit. on pp. 9, 11).

[27] Geoffrey E Hinton. “A Practical Guide to Training Restricted Boltzmann Machines”. In: *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 599–619 (cit. on p. 7).

[28] Quan Hoang et al. “Multi-Generator Generative Adversarial Nets”. *arXiv preprint arXiv:1708.02556* (2017) (cit. on pp. 12, 17, 19, 45).

[29] Alex Huang et al. “Adversarial Deep Learning for Robust Detection of Binary Encoded Malware”. *arXiv preprint arXiv:1801.02950* (2018) (cit. on p. 8).

[30] Gao Huang et al. *An empirical study on evaluation metrics of generative adversarial networks*. 2018. URL: https://openreview.net/forum?id=Sy1f0e-R- (cit. on p. 9).

[31] Phil Husbands. “Distributed Coevolutionary Genetic Algorithms for Multi-Criteria and Multi-Constraint Optimisation”. In: *AISB Workshop on Evolutionary Computing*. Springer. 1994, pp. 150–165 (cit. on pp. 13, 31).

[32] Andrew Ilyas et al. “The Robust Manifold Defense: Adversarial Training using Generative Models”. *arXiv preprint arXiv:1712.09196* (2017) (cit. on p. 1).

[33] Daniel Jiwoong Im et al. “Generative Adversarial Parallelization”. *arXiv preprint arXiv:1612.04021* (2016) (cit. on p. 2).

[34] Thomas Jansen and R. Paul Wiegand. “Sequential versus Parallel Cooperative Coevolutionary (1+1) EAs”. In: *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*. Vol. 1. IEEE. 2003, pp. 30–37 (cit. on pp. 42, 43).

[35] Mikkel T Jensen. “A new look at solving minimax problems with coevolutionary genetic algorithms”. In: *Metaheuristics: Computer Decision-Making*. Springer, 2003, pp. 369–384 (cit. on pp. 2, 11).

[36] Diederik P Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. *arXiv preprint arXiv:1412.6980* (2014) (cit. on pp. 2, 45).

[37] Diederik P Kingma and Max Welling. “Auto-Encoding Variational Bayes”. *arXiv preprint arXiv:1312.6114* (2013) (cit. on p. 7).

[38] Krzysztof Krawiec and Marcin Grzegorz Szubert. “Learning n-tuple networks for Othello by coevolutionary gradient search”. In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM. 2011, pp. 355–362 (cit. on p. 2).

[39] Alex Krizhevsky and Geoffrey Hinton. “Learning multiple layers of features from tiny images” (2009) (cit. on pp. 21, 52).

[40] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST Handwritten Digit Database”. *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010) (cit. on pp. 21, 52, 59, 62).

[41] Jerry Li et al. “Towards Understanding the Dynamics of Generative Adversarial Networks”. *arXiv preprint arXiv:1706.09884* (2017) (cit. on pp. 1, 8, 52–57).

[42] Jason Liang, Elliot Meyerson, and Risto Miikkulainen. “Evolutionary Architecture Search For Deep Multitask Networks”. *arXiv preprint arXiv:1803.03745* (2018) (cit. on p. 10).

[43] Ziwei Liu et al. “Deep Learning Face Attributes in the Wild”. In: *Proceedings of International Conference on Computer Vision (ICCV)*. Dec. 2015 (cit. on pp. 21, 52, 63).

[44] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009 (cit. on p. 18).

[45] Risto Miikkulainen et al. “Evolving Deep Neural Networks”. *arXiv preprint arXiv:1703.00548* (2017) (cit. on pp. 3, 5).

[46] Melanie Mitchell. “Coevolutionary Learning with Spatially Distributed Populations”. *Computational Intelligence: Principles and Practice* (2006) (cit. on pp. 2, 12, 14, 17, 31).

[47] Mitchell A Potter and Kenneth A De Jong. “A cooperative coevolutionary approach to function optimization”. In: *International Conference on Parallel Problem Solving from Nature*. Springer. 1994, pp. 249–257 (cit. on p. 11).

[48] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. *arXiv preprint arXiv:1511.06434* (2015) (cit. on pp. 17, 21, 24, 45, 58, 63).

[49] Sebastian Raschka. *Python Machine Learning*. Packt Publishing Ltd, 2015 (cit. on pp. 26, 29).

[50] Leonard Richardson and Sam Ruby. *RESTful web services*. O’Reilly Media, Inc., 2008 (cit. on p. 14).

[51] Tim Salimans et al. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. *arXiv preprint arXiv:1703.03864* (2017) (cit. on pp. 42, 44).

[52] Tim Salimans et al. “Improved Techniques for Training GANs”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 2234–2242 (cit. on pp. 7, 9, 33).

[53] Pouya Samangouei, Maya Kabkab, and Rama Chellappa. “Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models”. *arXiv preprint arXiv:1805.06605* (2018) (cit. on p. 1).

[54] Felipe Petroski Such et al. “Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning”. *arXiv preprint arXiv:1712.06567* (2017) (cit. on pp. 3, 5, 6, 41, 44).

[55] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998 (cit. on p. 6).

[56] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826 (cit. on p. 33).

[57] Gavin Taylor et al. “Training Neural Networks Without Gradients: A Scalable ADMM Approach”. In: *International Conference on Machine Learning*. 2016, pp. 2722–2731 (cit. on p. 5).

[58] Vinod K Valsalam et al. “Constructing controllers for physical multilegged robots using the ENSO neuroevolution approach”. *Evolutionary Intelligence* 5.1 (2012), pp. 45–56 (cit. on p. 5).

[59] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. “Generating Videos with Scene Dynamics”. In: *Advances In Neural Information Processing Systems*. 2016, pp. 613–621 (cit. on p. 8).

[60] Darrell Whitley. “An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls”. *Information and software technology* 43.14 (2001), pp. 817–831 (cit. on p. 11).

[61] Daan Wierstra et al. “Natural Evolution Strategies”. In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE. 2008, pp. 3381–3387 (cit. on pp. 2, 42, 43).

[62] Nathan Williams and Melanie Mitchell. “Investigating the success of spatial coevolution”. In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM. 2005, pp. 523–530 (cit. on pp. 12, 14).

[63] David H Wolpert and William G Macready. “No Free Lunch Theorems for Optimization”. *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82 (cit. on p. 11).

[64] Jiajun Wu et al. “Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 82–90 (cit. on p. 8).

Online sources

[65] Soumith Chintala and Yann LeCun. *A path to unsupervised learning through adversarial networks*. June 2016. URL: https://code.facebook.com/posts/1587249151575490 (visited on 05/04/2018) (cit. on p. 8).

[66] Torch Contributors. *Distributed communication package - torch.distributed*. May 2018. URL: https://pytorch.org/docs/master/distributed.html (visited on 05/04/2018) (cit. on p. 28).

[67] Guido van Rossu and Python Contributors. *multiprocessing – Process-based parallelism*. Python Software Foundation. June 2018. URL: https://docs.python.org/3/library/multiprocessing.html (visited on 06/21/2018) (cit. on p. 29).

[68] Christina Voskoglou. *What is the best programming language for Machine Learning?* May 2017. URL: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7 (visited on 05/04/2018) (cit. on p. 26).