Lipizzaner: A System That Scales Robust Generative Adversarial Network Training

Tom Schmiedlechner
CSAIL, MIT, USA
schmied@mit.edu

Ignavier Ng Zhi Yong
CSAIL, MIT, USA
ignavier@mit.edu

Abdullah Al-Dujaili
CSAIL, MIT, USA
aldujail@mit.edu

Erik Hemberg
CSAIL, MIT, USA
hembergerik@csail.mit.edu

Una-May O'Reilly
CSAIL, MIT, USA
unamay@csail.mit.edu

Abstract

GANs are difficult to train due to convergence pathologies such as mode and discriminator collapse. We introduce `Lipizzaner`, an open source software system that allows machine learning engineers to train GANs in a distributed and robust way. `Lipizzaner` distributes a competitive coevolutionary algorithm which, by virtue of dual, adapting, generator and discriminator populations, is robust to collapses. The algorithm is well suited to efficient distribution because it uses a spatial grid abstraction. Training is local to each cell and strong intermediate training results are exchanged among overlapping neighborhoods allowing high performing solutions to propagate and improve with more rounds of training. Experiments on common image datasets overcome critical collapses. Communication overhead scales linearly when increasing the number of compute instances and we observe that increasing scale leads to improved model performance.

1 Introduction

Despite their demonstrated success, it is well known that Generative Adversarial Networks (GANs) are difficult to train. The objective of training is to derive a generator that is able to completely thwart the discriminator in its ability to identify genuine samples from ones offered by the generator. GAN training can be formulated as a two-player minimax game: the (neural network) discriminator is trying to maximize its payoff (accuracy), and the (neural network) generator is trying to minimize the discriminator's payoff (accuracy). The two networks are differentiable, and therefore optimizing them is achieved by simultaneous gradient-based updates to their parameters. In practice, gradient-based GAN training often converges to payoffs that are sub-optimally stuck in oscillation or collapse. This is partly because gradient-based updates seek a stationary solution with zero gradient. This objective is a necessary condition for a single network to converge, but in the case of the GAN's coupled optimization, equilibrium is the corresponding necessary condition for convergence. Consequently, a variety of degenerate training behaviors has been observed—e.g., *mode collapse* [4], *discriminator collapse* [12], and *vanishing gradients* [3]. These unstable learning dynamics have been a key limiting factor in training GANs in a robust way, let alone tuning their hyperparameters or scaling training. Al-Dujaili et al. [1] offer a robust training solution that combines the training of multiple GANs with grid-based competitive coevolution. Succinctly, the training of each GAN pair is done with stochastic gradient descent (SGD) while the grid-based coevolutionary algorithm adaptively selects higher performing models for iterative training by referencing the GANs in cells in a local neighborhood. Overlapping neighborhoods and local communication allow efficient propagation of improving models. The impressive performance of this solution prompts us to distribute it, see

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Figure 1 (a), so that training is faster (by wall clock timing) and efficiently scalable. Our contribution is a scalable, parallelized and distributed GAN training system, implemented and licensed as open source[1], based on [1]'s solution. Additionally, the scaling allows us to experimentally determine that larger grids (i.e. more spatially distributed GAN training) yield better trained GANs.

2 Background

Improving GAN Training. Robust GAN training is still an open research topic [5]. Simple theoretical GAN models have been proposed to provide a better understanding of the problem [12]. For algorithmic implementations, several tips and tricks have been suggested to stabilize the training over the past years [6]. Some use hard-coded conditions to decrease the optimizers' learning rate after a given number of iterations [15], while others employ ensemble concepts [18]. Motivated by the similarity of degenerate behaviors in GAN training to the decade-old observed patterns in competitive coevolutionary algorithm dynamics (i.e., loss of gradient, focusing, and relativism), Al-Dujaili et al. [1] propose a spatial coevolution approach for GAN training. The authors conduct experiments on the theoretical GAN model of [12]. They show, using the theoretical model, that a basic coevolutionary algorithm with Gaussian-based mutations can escape behaviors such as mode and discriminator collapse. They also run a small-scale spatial coevolution with gradient-based mutations to update the neural net parameters and Gaussian-based mutations to update the hyperparameters on the MNIST and CelebA datasets. The bulk of attempts for improving GAN training have been designed to fit a single machine (or a single GPU). The advent of large-scale parallel computation infrastructure prompts our interest in scaling them. To do so, we select [1]'s solution because of its use of evolutionary computing.

Evolutionary Computing. Evolutionary algorithms are population-based optimization techniques. Competitive coevolutionary algorithms have adversarial populations (usually two) that simultaneously evolve [9] population solutions against each other. Unlike classic evolutionary algorithms, they employ fitness functions that rate solutions relative to their *opponent* population. Formally, these algorithms can be described with a minimax formulation [7, 2], and therefore share common characteristics with GANs.

Scaling Evolutionary Computing for ML. A team from OpenAI [16] applied a simplified version of Natural Evolution Strategies (NES) [19] with a novel communication strategy to a collection of reinforcement learning (RL) benchmark problems. Due to better parallelization over thousand cores, they achieved much faster training times (wall-clock time) than popular RL techniques. Likewise, a team from Uber AI [17] showed that deep convolutional networks with over 4 million parameters trained with genetic algorithms can also reach results competitive to those trained with OpenAI's NES and other RL algorithms. OpenAI ran their experiments on a computing cluster of 80 machines and 1440 CPU cores [16], whereas Uber AI employed a range of hundreds to thousands of CPU cores (depending on availability). Another effective mean to scale up evolutionary algorithm in a distributed setting is spatial (toroidal) coevolution, which controls the mixing of adversarial populations in coevolutionary algorithms. The members of populations are divided up on a grid of cells and each cell has a local neighborhood. A neighborhood is defined by adjacent cells and specified by its size, n_{cells}. This reduces the cost of overall communication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n_{cells}n)$, where n is the size of each population. Five cells per neighborhood (one center and four adjacent cells) are common [10]. With this notion of distributed evolution, each neighborhood can evolve in a different direction and more diverse points in the search space are explored [14, 20]. The next section presents `Lipizzaner`: a scalable, distributed system for coevolutionary GAN training.

3 The `Lipizzaner` System

We start with a brief description of how `Lipizzaner` trains. Second, we discuss the general design principles and requirements for a scalable architecture of the framework. Then, we describe the concrete implementation steps of the resulting system.

Coevolutionary GAN Training. The coevolutionary framework is executed in an asynchronous fashion as described in the following steps (depicted in Figure 1 (b)). *1)* Randomly initialize each cell in the grid with a generator and a discriminator of random weight and hyperparameters. The i-th

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

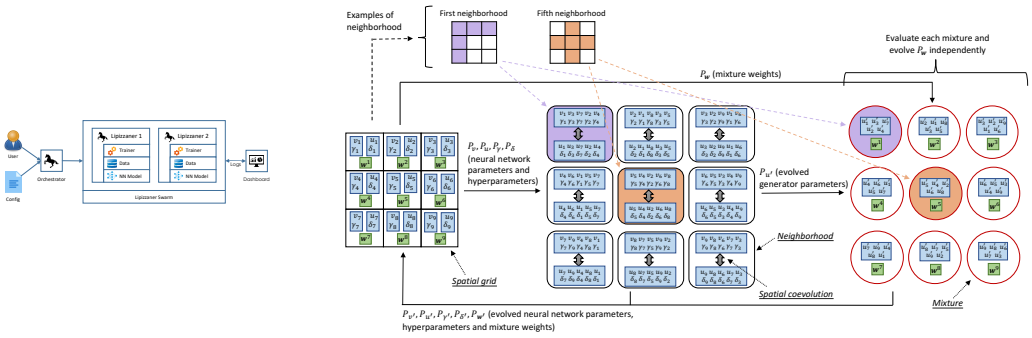


Figure 1: (a) High-level view of `Lipizzaner`'s architecture. User provides configuration file to the *orchestrator*. A distributed `Lipizzaner` swarm is controlled by the *orchestrator*. Each node in the swarm asynchronously trains the combination of the cell's GANs with its neighbors'. The dashboard shows progress and results. (b) `Lipizzaner` training on a 3×3 grid. $P_v = \{v_1, \dots v_9\}$ and $P_v = \{u_1, \dots u_9\}$ denote neural network parameters of discriminator and generator population respectively. $P_\gamma = \{\gamma_1, \dots, \gamma_9\}$ and $P_\delta = \{\delta_1, \dots, \delta_9\}$ denote the hyperparameters (e.g., learning rate) of discriminator and generator population, respectively. $P_\mathbf{w} = \{\mathbf{w}_1, \dots, \mathbf{w}_9\}$ denote the mixture weights. The $(\cdot)'$ notation denotes the value of $(\cdot)$ after one iteration of (co)evolution.

generator is described by its neural net parameters u_i and hyperparameters δ_i. The j-th discriminator is similarly described with v_j and γ_j. The generators (discriminators) from all the cells form the generator (discriminator) population P_u, P_δ (P_v, P_γ). *2)* Each generator (discriminator) is evaluated against each of the discriminators (generators) in its neighborhood. The evaluation process computes $\mathcal{L}(u_i, v_j)$: the value of the GAN objective (loss function) $\mathcal{L}$ at the corresponding generator u_i and discriminator v_j. The values of a discriminator's (generator's) interactions are averaged (negative-averaged) to constitute its fitness. *3)* Generators and discriminators in each neighborhood are selected based on tournament selection. For the selected generator and discriminator, SGD training then performs gradient-based updates on their neural net parameters u_i and v_j, while Gaussian-based updates create new hyperparameters values δ_i' and γ_j'. *4)* `Lipizzaner` produces a mixture of generators. It assigns a mixture weight vector $\mathbf{w}$ for each neighborhood. These vectors are evolved using an ES-(1+1) algorithm which optimizes for the performance (e.g., Fréchet Inception Distance (FID) score [8]) of the neighborhood generators (weighted by $\mathbf{w}$). *5)* Go to step *2)*. We next describe `Lipizzaner`'s two system level modules.

System Modules. As shown in Figure 1 (a), `Lipizzaner` has a core and a dashboard module. *1)* The core module has a component-based design. All components exist on each distributed `Lipizzaner` instance. They are: *Input data loader* that manages the data samples for training, i.e. the distribution the generator tries to reproduce. *Neural network model* that generates or discriminates data. *Trainer* that executes the training iterations[2] of the evolutionary process itself. *Trainer* accesses input data and the models from their respective components and evolves them with the settings provided by the configuration. *Distribution server and client* that sends and receives data via a TCP/IP interface. The server component offers a public API and endpoints for accessing the state of an instance. The fitness values of the individuals and the internal state of the gradient optimizers are shared. The state of some optimizers is lightweight, while e.g. Adam requires transmission of complex state objects. *Configuration* that is vertically aligned over the system and connected to all components. All parameters are specified in configuration files. This reduces redeployment and code editing. *2)* `Lipizzaner`'s dashboard module is designed to simplify the analysis of experiments and make training transparent. The GUI-based monitoring shows phenomena like solutions propagating through the grid, oscillation between generators and discriminators, etc. GUI components and their interactions are illustrated in Figure 2. The *Log database* contains details about the executed experiments and instances. The *Back-end controller* is a server-side component connecting the front-end to the log database. Finally, *Front-end component and view* contains the logic to access the experiment and result data from the back-end controller and to inject it into the view.

[2]Iteration, generation and epoch are used interchangeably in our system

Implementation and Distribution. The core module is written in `Python3` and uses `pytorch`[3]. The dashboard is an `Angular.js` single page web application using `Type Script` and `ASP.NET Core`. The trainer component defines and hosts the executed evolutionary process. Currently `Lipizzaner` supports different types of trainers; gradient-free trainers for evolution strategies and Natural Evolution Strategies [19], and trainers that update the neural net parameters with gradient-based optimizers. Most GAN types primarily differ by the way they update their weights and fitness evaluation. `Lipizzaner` injects the necessary functionality into the respective trainer (as most training functionality is not class-specific, but based on interfaces passed to the constructor, i.e. *Dependency Injection*). The precise training steps differ slightly depending on the trainer and type of GAN, but all share a common coevolutionary baseline procedure.

To support distribution, `Lipizzaner` uses `Docker`[4]swarm and `docker-machine`. A master-client design pattern is used for communication. Clients are added by starting the application on more nodes, running pre-configured virtual machines, or with `Docker`.
The `Lipizzaner` master (or *orchestrator*) is meant to control a single experiment. Its tasks are: *1)* parse the configuration and connect to clients and transmits the experiment to all of them. *2)* periodically check client state, i.e. if the experiment has ended, or if the client may be not reachable anymore, unreachable clients can be ignored, or the entire experiment terminated. *3)* gather finished results, save them to its disk and rank the final mixtures by their scores. Create sample images. The task overhead requires only modest computation power. `Lipizzaner` instances communicate with 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.
From a logical perspective, each client represents one cell in the spatial grid topology. An experiment request contains all configuration options a user pass into the master application, as it is forwarded to the client. The typical behavior of the client has three steps: *1)* If no experiment is running, the client runs in the background and listens for experiment requests on a specific port. When an experiment is requested, the client parses the received configuration file and executes the specified training algorithm. It furthermore requests data from the neighboring cells each time the algorithm accesses the respective properties of the populations. *2)* It offers HTTP endpoints simultaneously to execute the training process. Other clients can also access these endpoints and request the current populations and optimizer parameters. *3)* The master actively monitors the clients and collects the results. After this, the client changes its state from `Busy` to `Idle` and waits for new experiment requests.

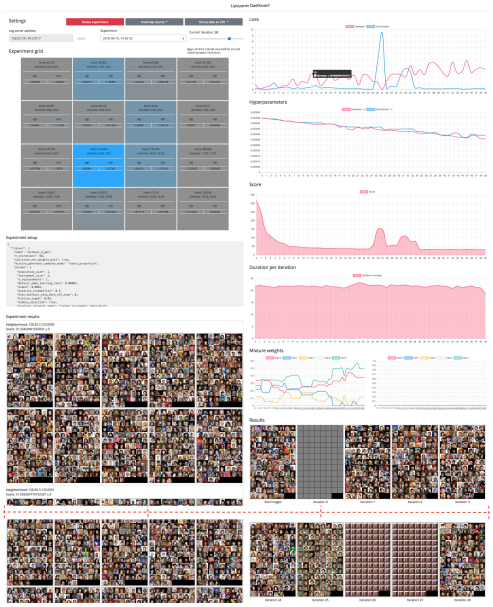


Figure 2: Screen shot of the `Lipizzaner` dashboard web application—a demo dashboard page can be found at `https://github.com/ALFA-group/lipizzaner-gan/blob/master/dashboard-demo/dashboard.html` for a better readibility. The orange dashed line indicates that images from iteration 4 to 23 are cropped. The navigation component is on the left, the details component is on the right side of the screen. The navigation loads and displays the experiment selection dialog elements. For a selected experiment, configuration, details, topology, and execution time are shown. For an experiment done, samples from the resulting mixtures are shown as well. It is possible to scroll through training iterations while displaying a live heat map of the grid. When a grid cell of a specific experiment is selected, the details component to the right displays drill-down information about the whole experiment history of the cell. This includes charts for loss, hyperparameters, mixture weights and score values. Intermediate generator output images for each iterations are displayed as well, together with real images from the input dataset.

[3] `https://pytorch.org/`
[4] `https://www.docker.com`

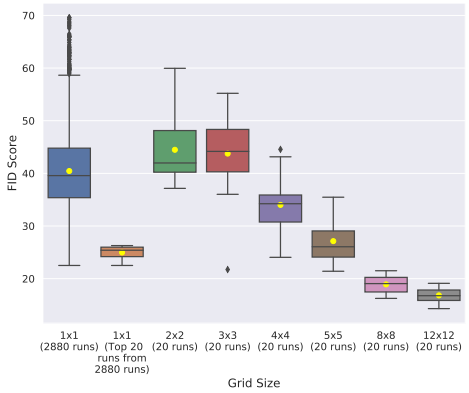


Figure 3: Box plot of the FID score for different grid sizes on MNIST. The x-axis shows box plots for different grid sizes and the y-axis shows the FID score. Yellow dots are the average FID, black diamonds are outliers. Larger grid sizes have lower FID scores. For the sake of legibility, only those experiments having FID score lower than 70 are included in the box plot. For 1×1 grid, there are 27, 7, 18 outliers which lie in the interval of (70, 100), (100, 140), (500, 1100) respectively.

Table 1: Setup for experiments conducted with the `Lipizzaner` system on MNIST and CelebA datasets.

Parameter	MNIST	CelebA
Coevolutionary settings		
Iterations	200	50
Population size per cell	1	1
Tournament size	2	2
Grid size	1×1 to 12×12	1×1 to 4×4
Mixture mutation scale	0.01	0.05
Hyperparameter mutation		
Optimizer	Adam	Adam
Initial learning rate	0.0002	0.00005
Mutation rate	0.0001	0.0001
Mutation probability	0.5	0.5
Network topology		
Network type	MLP	DCGAN
Input neurons	64	100
Number of hidden layers	2	4
Neurons per hidden layer	256	$16,384 - 131,072$
Output neurons	784	$64 \times 64 \times 3$
Activation function	tanh	tanh
Training settings		
Batch size	100	128
Skip N disc. steps	1	-

4 Experiments

This section provides empirical evaluation of `Lipizzaner` on two common image datasets, MNIST and CelebA. We assess the system in terms of its scaling properties and generative modeling performance. The settings used for the experiments with `Lipizzaner` are shown in Table 1.

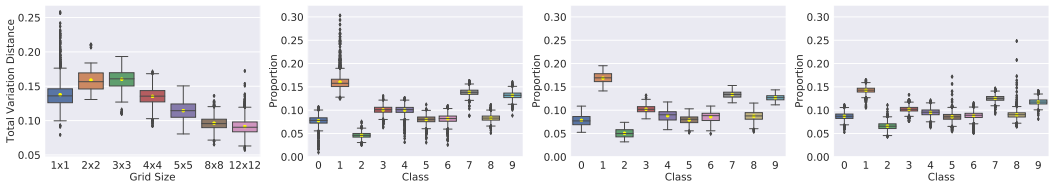
4.1 MNIST Dataset

Scalability and Performance. `Lipizzaner` improves the performance, convergence speed and stability of the generator for larger grid sizes when measuring the average FID score over multiple runs, see Figure 3. A rank sum test with Bonferroni correction shows significant differences for the grid sizes larger than 4×4 at 99% confidence level. One hypothesis for the behavior of `Lipizzaner` is that there is less overlap in the neighborhoods for these grid sizes. We execute the 1×1 grid for 2,880 runs in order to use similar compute effort as the 12×12 grid. Even then the minimum FID for the 1×1 grid (22.5) is higher than the maximum FID of the 8×8 grid (21.5). Multiple outliers and discriminator collapses are observed for 1×1 grid, whereas larger grid sizes not only improve the stability with smaller standard deviation and less outliers, but also manage to completely overcome discriminator collapses. These experiments were conducted on a GPU cluster node which consists of eight Nvidia Titan Xp with 12 GB RAM, 16 Intel Xeon cores with 2.2GHz each, and 125 GB RAM.

Generator Mixture Distribution. We study the distribution of the generator mixture. We follow [11] and report the total variation distance (TVD) for the different grid sizes, see Figure 4a. The larger grid sizes have lower TVD, which indicate that mixtures from larger grid sizes produce a more diverse set of images spanning across different classes. The distribution of each classes of generated images for 1×1 is in Figure 4b and is the least uniform. For the 4×4, Figure 4c, the distribution is more uniform. Finally, the distribution for 12×12, Figure 4d, is the most uniform.

4.2 Celebrity Faces Dataset

The CelebA dataset [13] contains 200,000 portraits of celebrities and is commonly used in GAN literature. `Lipizzaner` can overcome mode and discriminator collapses even when only the smallest possible grid size (2×2) is used. The increased diversity is sufficient to replace collapsed individuals in the next iteration, and even allows the system to prevent collapse in most runs. An example for a recovering system is shown from iteration 25 to 28 in Figure 2. The scaling performance for this



(a) TVD for different grid sizes (b) Classes distribution for 1×1 (c) Classes distribution for 4×4 (d) Classes distribution for 12×12

Figure 4: Generator mixture distribution for MNIST. The average TVD is shown in Figure 4a. The larger grid sizes have lower TVD, which indicate that mixtures from larger grid sizes produce a more diverse set of images spanning across different classes. This is further supported by visualizing the distribution of each classes of generated images for different grid sizes. The distribution of each classes of generated images for 1×1 is in Figure 4b. The 4×4, Figure 4c, show a more uniform distribution. The distribution for 12×12, Figure 4d, is the most uniform.

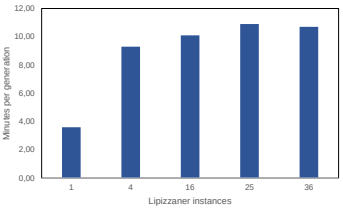


Figure 5: Near constant training times on AWS per iteration on the CelebA dataset, averaged over 30 iterations. X-axis shows the number of `Lipizzaner` instances and y-axis shows the duration in minutes per iteration.

data set is different, and has different computational requirements, so we are only able to measure the generative performance up to a 4×4 grid. The results show no statistical significant difference: 1×1 **(10 runs)** gives 31.89 ± 1.26, 2×2 **(10 runs)** gives 30.27 ± 0.50 and 4×4 **(10 runs)** gives 30.59 ± 1.03.

Scalability and Training Time. Scalability was one of the main requirements while designing `Lipizzaner`. The spatial grid distribution architecture, allows the computational effort to increase linearly instead of quadratically (up to 6×6 grid). This claim is supported by the chart shown in Figure 5, which illustrates a near linear training time per iteration for different numbers of connected instances. The initial relatively large step from one to four instances is caused by the fact that multiple instances were run per GPU for the distributed experiments; this increases the calculation effort per GPU, and therefore affects the training time as well. We also observed low communication durations in our experiments: exchanging data between two clients only takes 0.5 seconds on average in state-of-the-art Gigabit Ethernet networks and is only performed once per iteration. Additionally, the asynchronous communication pattern leads to the usage of different time slots and therefore reduces high network peak loads. The experiments were computed on AWS GPU cloud instances. Each instance had one Nvidia Tesla K80 GPU with 12 GB RAM, 4 Intel Xeon cores with 2.7 GHz each, and 60 GB RAM. The times shown are averaged over 30 iterations of training a DCGAN neural network pair on the CelebA dataset. The instances hosted Docker containers and connected through a virtual overlay network.

5 Conclusion

`Lipizzaner` yields promising results in the conducted experiments and is able to overcome otherwise critical scenarios like mode and discriminator collapse. The main advantage of incorporating GANs in coevolutionary algorithms is the usage of populations and therefore increased diversity among the possible solutions. Using a relatively small spatial grid is sufficient to overcome the common limitations of GANs, due to the spatial grid and asynchronous evaluation. The performance also improves with increased grid size. In addition, `Lipizzaner` scales well up to the grid sizes elaborated in the conducted experiments (i.e. a grid size of 12×12 for MNIST and 6×6 for CelebA). Future work includes extending the GAN trainers used (e.g., WGAN), investigating coevolutionary variants, and improving the dashboard for tracing solutions over time.

References

[1] Abdullah Al-Dujaili, Tom Schmiedlechner, , Erik Hemberg, and Una-May O'Reilly. Towards distributed coevolutionary gans. *AAAI 2018 Fall Symposium*, 2018.

[2] Abdullah Al-Dujaili, Shashank Srikant, Erik Hemberg, and Una-May O'Reilly. On the application of Danskin's theorem to derivative-free minimax optimization. *Int. Workshop on Global Optimization*, 2018.

[3] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

[4] Sanjeev Arora and Yi Zhang. Do gans actually learn the distribution? an empirical study. *arXiv preprint arXiv:1706.08224*, 2017.

[5] Sanjeev Arora, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. Generalization and equilibrium in generative adversarial nets (gans). *arXiv preprint arXiv:1703.00573*, 2017.

[6] Soumith Chintala, Emily Denton, Martin Arjovsky, and Michael Mathieu. How to train a gan? tips and tricks to make gans work. https://github.com/soumith/ganhacks, 2016.

[7] Jeffrey W Herrmann. A genetic algorithm for minimax optimization problems. In *CEC*, volume 2, pages 1099–1103. IEEE, 1999.

[8] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, and Bernhard Nessler. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *arXiv preprint arXiv:1706.08500*, 2017.

[9] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1):228 – 234, 1990. ISSN 0167-2789. doi: https://doi.org/10.1016/0167-2789(90)90076-2.

[10] Phil Husbands. Distributedcoevolutionary genetic algorithms for multi-criteria and multi-constraint optimisation. In *AISB Workshop on Evolutionary Computing*, pages 150–165. Springer, 1994.

[11] Chengtao Li, David Alvarez-Melis, Keyulu Xu, Stefanie Jegelka, and Suvrit Sra. Distributional adversarial networks. *arXiv preprint arXiv:1706.09549*, 2017.

[12] Jerry Li, Aleksander Madry, John Peebles, and Ludwig Schmidt. Towards understanding the dynamics of generative adversarial networks. *arXiv preprint arXiv:1706.09884*, 2017.

[13] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[14] Melanie Mitchell. Coevolutionary learning with spatially distributed populations. *Computational Intelligence: Principles and Practice*, 2006.

[15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[16] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[17] Kenneth O. Stanley and Jeff Clune. Welcoming the era of deep neuroevolution - uber engineering blog. https://eng.uber.com/deep-neuroevolution/, December 2017.

[18] Yaxing Wang, Lichao Zhang, and Joost van de Weijer. Ensembles of generative adversarial networks. *arXiv preprint arXiv:1612.00991*, 2016.

[19] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

[20] Nathan Williams and Melanie Mitchell. Investigating the success of spatial coevolution. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 523–530. ACM, 2005.