Using High-Performance Computing to Scale Generative Adversarial Networks

by
Diana J. Flores
B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2020

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2021

Signature of Author: _____
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by: _____
Erik Hemberg
Research Scientist
Thesis Supervisor

Certified by: _____
Jamal Toutouh
Postdoctoral Researcher
Thesis Supervisor

Certified by: _____
Una-May O'Reilly
Principal Investigator
Thesis Supervisor

Accepted by: _____
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Using High-Performance Computing to Scale Generative Adversarial Networks

by

Diana J. Flores

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in Partial Fulfillment of the Requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Generative adversarial networks(GANs) are methods that can be used for data augmentation, which helps in creating better detection models for rare or imbalanced datasets. They can be difficult to train due to issues such as mode collapse. We aim to improve the performance and accuracy of the Lipizzaner GAN framework by taking advantage of its distributed nature and running it at very large scales. Lipizzaner was implemented for robustness, but has not been tested at scale in high performance computing(HPC) systems. We believe that by utilizing HPC technologies, we can scale up Lipizzaner and observe performance enhancements. This thesis achieves this scale up, using Oak Ridge National Labs' Summit Supercomputer. We observed improvements in the performance of Lipizzaner, especially when run with poorer network architectures, which implies Lipizzaner is able to overcome network limitations through scale.

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Thesis Supervisor: Jamal Toutouh
Title: Postdoctoral Researcher

Thesis Supervisor: Una-May O'Reilly
Title: Principal Investigator

Acknowledgments

My years at MIT have truly been some of the best and most transformative of my life. They have allowed me to work with and learn from some of the brightest people I have ever met, while also giving me a community I would be lost without. Completing this thesis opened me up to new challenges and taught me to be more comfortable taking on responsibility in professional settings. Not only have I expanded my knowledge and understanding of computer science and engineering, but I have discovered and begun to grow into the person I want to be. It is impossible to imagine reaching this point in my life if not for all of the people who have supported me along the way, both academically and personally.

Firstly, I would like to thank my mentors and friends of the ALFA group at MIT CSAIL, specifically Erik Hemberg, Jamal Toutouh and Una-May O'Reilly for their guidance with this research and their trust in me to rise to the challenge. I would also like to recognize the Computational Data Analytics Group at the Oak Ridge National Laboratory, particularly Katie Schumann, who assisted with usage of the Summit Supercomputer for this research. Further, I would like to thank all of the advisors and professors who have supported me, as a student and a TA, for giving me the necessary background to succeed in this research.

Aside from academic support, I have also been blessed with the best personal support network I could have ever dreamed of. To all of the friends I have made and communities I have become a part of at MIT, I truly would not have made it through these last five years without the sense of home you gave me. Finally, and most importantly, I'd like to thank my family, especially my mom, dad, and older sisters. Thank you for shaping me into who I am and reminding me what is important. Thank you for always lifting me up and for believing I could thrive at MIT before I did. Thank you for loving me unconditionally.

Contents

1 Introduction **8**

1.1 Research Questions 9

1.2 Contributions 9

2 Background **11**

2.1 GANs 11

2.2 Lipizzaner Framework 12

2.3 Computing Environments 14

2.3.1 Cloud Computing Environment 14

2.3.2 HPC Environments 15

3 Methods **16**

3.1 Handling Failures 16

3.2 Improving Parallelization 17

3.3 Adhering to Resource Constraints 18

3.3.1 Time Constraints: Checkpointing 18

4 Experiments **21**

4.1 Experiment Set-up 21

4.2 Results 23

4.2.1 MNIST 23

4.2.2 COVID-19 26

5 Discussion **30**

5.1 GANs 30

5.2 HPC Environments 31

6 Conclusion and Future Work **33**

List of Tables

4.1 Experiment Settings 22
4.2 MNIST Neural Network Topologies 23
4.3 COVID-19 Neural Network Topologies 23
4.4 MNIST 4LP FID Scores and Runtimes(in minutes) 24
4.5 MNIST 4LP Rank-Sum P-Values 24
4.6 Covid-19 Inception Scores and Runtimes(in minutes) 27
4.7 Covid-19 Rank-Sum P-Values for Different Networks 27

List of Figures

2-1 GAN Architecture 11
2-2 Lipizzaner Grid Diagrams 13
2-3 Lipizzaner Flowchart[11] 13
2-4 Grid Cell Training Process[11] 14

4-1 Generated MNIST Images: 4 Layer Perceptron 25
4-2 Generated MNIST Images: CNN [5] 26
4-3 Generated COVID-19 Images: 4 Layer Perceptron 28
4-4 Generated COVID-19 Images: 28x28 CNN 28
4-5 Generated COVID-19 Images: 64x64 CNN 28
4-6 Generated Covid-19 Images: 128x128 CNN [5] 29

Chapter 1: Introduction

The advancement of machine learning has made it much easier to predict results and make decisions in many fields, such as finance, healthcare, and cyber security[17, 16]. The accuracy of these predictions, however, does not rely solely on the algorithms used, but also, to a large extent, on the data that is used for training. Problems may arise when datasets are limited by external factors. For example:

- Relevant events occurring rarely.
- Relevant events being difficult to classify or access(i.e. medical data is often not publicly available).

A generative adversarial network(GAN) is a framework that aims to learn a function of some unknown distribution via an adversarial process[9]. One application of GANs is image generation, which can help dataset limitations by providing augmentation. The Lipizzaner framework[11] seeks to do so in a distributed manner, running parallel co-evolutionary sub-populations of GANs atop a 2d grid. This design lends itself large scale high-performance computing(HPC); however, prior to this thesis, the Lipizzaner framework had not been run in HPC environments.

Lipizzaner was designed with scale in mind, meaning performance improvements are expected when increasing the number of sub-populations of GANs used. However, we anticipated several challenges with adapting it for HPC because, at larger scales, inefficiencies are made more clear and systems fail. To mitigate these issues, we believed Lipizzaner would require re-factoring to, incorporate error handling in the

case of client failure, improve parallelization, and adhere to resource constraints of computing environments.

1.1 Research Questions

At the onset of this thesis, we were interested in observing performance changes that come about when running Lipizzaner at scales only HPC systems can provide. In our efforts to learn this, we developed a series of research questions:

1. *Can Lipizzaner run efficiently on HPC environments?*
2. *What parts of Lipizzaner break when run at large scales?*
3. *What improves when running Lipizzaner at large scales?*
4. *Can larger scales improve performance of Lipizzaner on different datasets?*
5. *Can scale overcome a suboptimal choice in network architecture in the event of hardware limitations?*

1.2 Contributions

The main contributions of this thesis are:

- Successfully running Lipizzaner in the Summit[1] HPC environment
- Extending Lipizzaner to improve error handling in the case of client failures
- Performance enhancements through increased parallelization of the Lipizzaner algorithm
- Periodic checkpointing for pausing then resuming experiments
- Results that show increased scale can overcome suboptimal neural network architecture

[1]The work had support from Oak Ridge National Labs for project "Towards A Robust and Scalable Adversarial Learning Framework" (CSC387), and from MIT CSAIL Systems That Learn.

The rest of this thesis is organized as follows. Chapter 2 presents background for technologies used. Chapter 3 describes methods. Chapter 4 outlines experiment set up and presents results. Chapter 5 presents discussion. Chapter 6 presents our conclusions and future work.

Chapter 2: Background

This section describes technologies and architectures used. First, we present GAN training with notation. Second, we introduce the Lipizzaner framework. Finally, we discuss the computing environments used for implementation and experimentation.

2.1 GANs

GAN training is a method to learn some unknown distribution in an adversarial approach. This is accomplished by training two models: a generator and a discriminator. The generator is tasked with transforming random latent space into "fake" samples meant to resemble the input data. The discriminator takes in both generated data and "real" input, then must correctly label which is which. The competitive nature of this method is a *minimax* problem, in which the generator aims to minimize its error while the discriminator aims to maximize its performance.

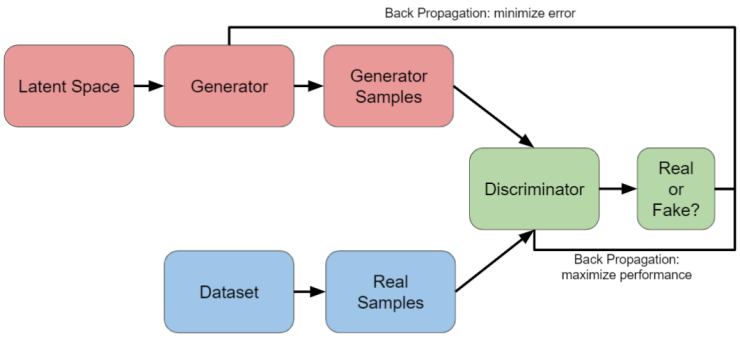


Figure 2-1: GAN Architecture

Formally we can define GANs using notation from [11]. Let $\mathcal{G} = \{G_g, g \in \mathcal{U}\}$, $\mathcal{U} \subset \mathcal{R}^p$ and $\mathcal{D} = \{D_d, d \in \mathcal{V}\}, \mathcal{V} \subset \mathcal{R}^p$ denote sets of generators and discriminators. G_g is a function $G_g : \mathcal{R}^l \to \mathcal{R}^v$ parameterized by g. D_d is a function $D_d : \mathcal{R}^v \to [0, 1]$ parameterized by d. The generator G_g defines a distribution $\mathcal{T}_{G_g}$, this is done by generating z from an l-dimensional Gaussian distribution and then applying G_g on z to generate a sample $x = G_g(z)$ of the distribution G_g. Finally, let $\mathcal{T}_*$ be the unknown target distribution to which we would like to fit our generative model G_g.

GAN training is done to find parameters that optimize the following min-max problem:

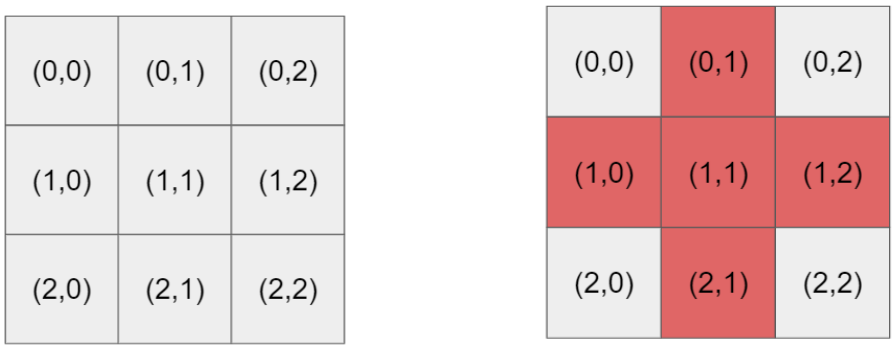
$$\min_{g \in \mathcal{U}} \max_{d \in \mathcal{V}} \mathcal{L}(g, d) \ , \ where \tag{2.1}$$

$$\mathcal{L}(g, d) = \mathbb{E}_{x \sim \mathcal{T}_*}[\phi(D_d(x))] + \mathbb{E}_{x \sim \mathcal{T}_{G_g}}[\phi(1 - D_d(x))] \ , \tag{2.2}$$

In achieving this, a successful generator can be used to augment rare datasets, which can improve the accuracy of prediction or decision-making models. However, training a GAN is no simple task. Some common problems when training are vanishing gradients, mode collapse, and discriminator collapse[6, 7, 13].

2.2 Lipizzaner Framework

The Lipizzaner Framework[1] was developed to tackle common problems with GANs. Lipizzaner runs atop a 2D grid of cells, each of which sets up co-evolutionary subpopulations that get re-initialized periodically by communicating with neighboring cells[11]. Each cell initializes its own generator and discriminator network, which are configurable by the user. A cell's neighborhood is defined as itself, and the 4 cells directly above, below, to the right, and left of it. If at an edge, the neighborhood wraps around to the other side of the grid. Figure 2-2 displays the neighborhood the center grid space (1,1). Each cell's generator is updated by being compared to the discriminators in its neighborhood and vice-versa for each cell's discriminator. This allows each cell to be run asynchronously and in parallel.



(a) 3x3 Grid (b) Neighborhood Centered on (1,1)

Figure 2-2: Lipizzaner Grid Diagrams

To begin distributed training on Lipizzaner, a client node is brought up for each grid space desired and then a master node is brought up to oversee client training. At a high level, each client node loads a batch of the dataset and performs a configurable number of iterations of training before returning its results to the master node. The master then decides on the best grid space based on which generator sub-populations perform the best against their discriminators. Figure 2-3 depicts Lipizzaner's overall system diagram and Figure 2-4 depicts the training process at each grid cell. This process involves communication across client nodes as well as communication between client nodes and the master node. This communication is done via HTTP requests. Favorable results were seen when running Lipizzaner atop small grid spaces(i.e. 2x2 and 3x3).

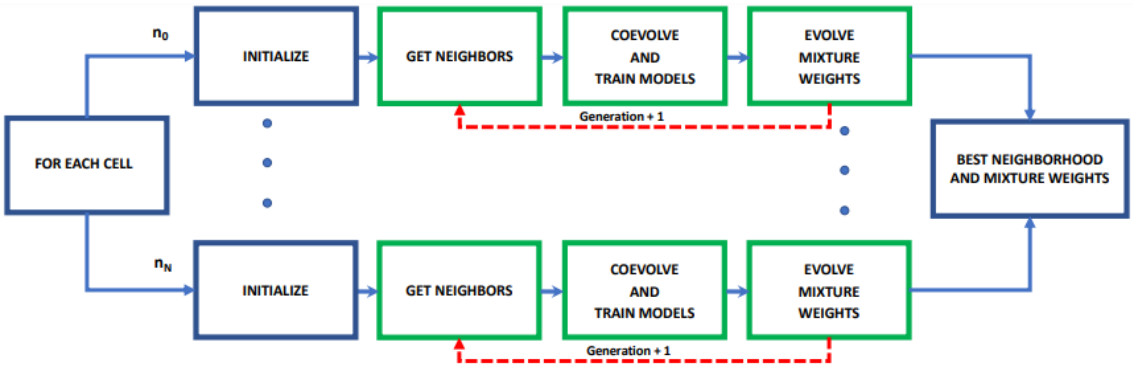


Figure 2-3: Lipizzaner Flowchart[11]

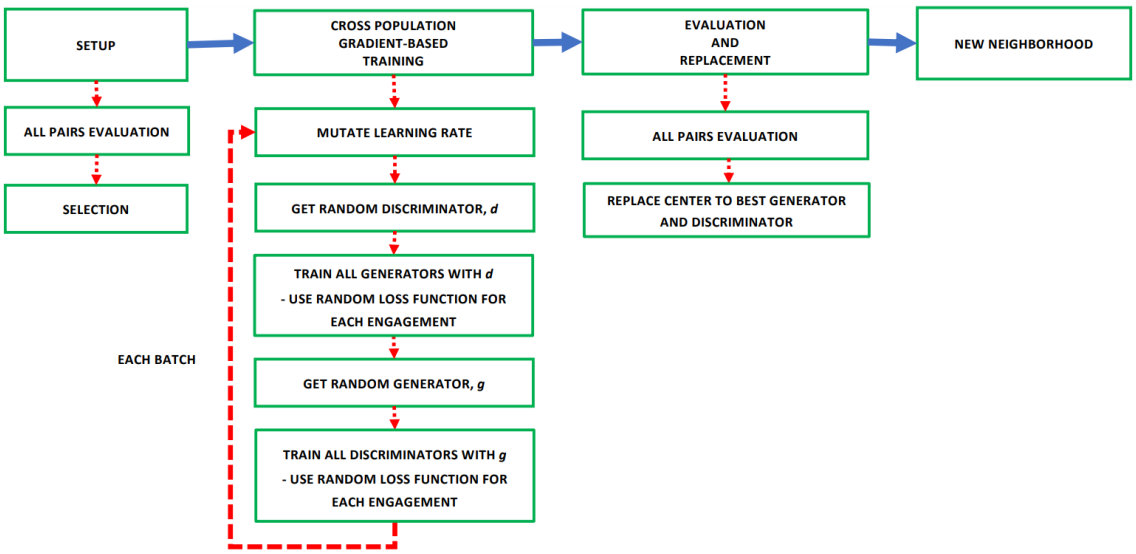


Figure 2-4: Grid Cell Training Process[11]

Previous work with Lipizzaner can be found in [11, 15, 5]. This thesis differs in that it is the first research looking to run Lipizzaner in High-Performance Computing(HPC) environments. This will allow us to run atop grid sizes that are not possible in the cloud environments that have been used in previous work. Further, it will allow us to find any shortcomings of Lipizzaner and implement it in a more robust way, so that it is still optimal at large scales.

2.3 Computing Environments

In order to run the Lipizzaner framework, we utilize both cloud-computing and HPC environments. Each of these offer different functionalities that were helpful throughout the course of this thesis.

2.3.1 Cloud Computing Environment

MIT OpenStack[2] is cloud environment that provides Infrastructure as a Service to the MIT community. Throughout the course of this thesis, we relied on OpenStack mainly for debugging purposes. It does not have the compute power that other en-

vironments could provide, but did have the benefit of being able to run interactive experiments with less overhead, which was helpful when testing small cases or modifications.

2.3.2 HPC Environments

High-performance computing(HPC) is the usage of supercomputers for problems that cannot be performed on standard computers due to computational complexity or size. HPC systems are comprised of interconnected nodes, each of which having one or more graphics processing units(GPUs). Communication can occur across processing chips within one node, as well as across nodes, making HPC systems the ideal setting for parallel computing[10]. For this thesis, we had access to two HPC environments: the MIT Satori cluster[3] and the Summit Supercomputer[4] housed at Oak Ridge National Labs(ORNL).

MIT's Satori cluster is an IBM Power9 cluster and currently in a beta test phase. This means usage is limited to a small number of nodes. As a result, Satori could not yet be used to experiment with the scaling of Lipizzaner, but was a good resource as a testing HPC environment. ORNL's Summit Supercomputer, also an IBM system, is one of the fastest super computers in the world. It is comprised of 4,900 compute nodes and functioned as the primary experimentation environment for this thesis.

Chapter 3: Methods

This section presents methods for efficiently running Lipizzaner on HPC environments, as it was initially implemented for cloud-computing environments. Specifically, we discuss handling failures, improving parallelization, and adhering to resource constraints.

3.1 Handling Failures

One of the first roadblocks found from running large grid sizes with Lipizzaner was that the original code asserted that a square grid was used(Algorithm 1), which is difficult to guarantee when requesting a large number of clients. Since each Lipizzaner client opens an HTTP connection, there is a probability of failure when starting each client. On Summit, this probability was approximately 5%. When using small grid sizes, with less than 10 clients, it is not likely to observe a failure to connect; however, when using grids with over 100 clients, we can be almost certain to observe at least one failure. Therefore, the original Lipizzaner code required re-factoring to account for this issue.

Algorithm 1 Original Non-Square Grid Handling

```
1: ac ← successful clients
2: num_clients ← length(ac)
3: if num_clients == 0 or not is_square(num_clients) : then
4:     log critical error
5:     terminate all clients
6: end if
```

To solve this, we modified the Lipizzaner code to allow for rectangular grids by taking

the number of successfully connected clients, finding the largest rectangle size that can be made with those, and setting the grid to that size(Algorithm 2). We take the next largest rectangle size grid to provide symmetry and ensure each cell can locate its neighbors in a simple manner.

Algorithm 2 Modified for Rectangular Grids

```
ac ← successful clients
num_clients ← len(ac)
if not is_square(num_clients) : then
    # compute dimensions of new grid
    width ← int(round(sqrt(num_clients)))
    height ← num_clients//width
    # reduce set of successful clients
    ac ← ac[: width ∗ height]
    terminate all other clients
end if
if num_clients == 0 : then
    log critical error
end if
```

3.2 Improving Parallelization

Once we were successfully able to start and train Lipizzaner with large grid sizes, there were still aspects of the algorithm that were causing performance bottlenecks. For example, as mentioned in section 2.2, each client node reports to the master node once it completes its training. Initially, the master node would receive information from each client node and compute a final score for the performance of each node(Algorithm 3). When using 2x2 grids, this did not cause much overhead. However, upon scaling up, this computation was a clear bottleneck. By moving this computation into the clients(Algorithm 4), running in parallel, we were able to reduce the overall runtime of Lipizzaner.

Algorithm 3 Original

```
# inside master
for client in all_clients do
    Save generated images
    Compute FID Score
end for
choose best client
```

Algorithm 4 Modified

```
parfor client in all_clients do
    Save generated images
    Compute FID Score
    Push score to master
end parfor
# inside master :
for client in all_clients do
    receive score from client
end for
choose best client
```

3.3 Adhering to Resource Constraints

3.3.1 Time Constraints: Checkpointing

Summit has various wall-time restrictions for each experiment run. Experimenting with different datasets, Lipizzaner's training did not always fit within those limits. To mitigate this, it was necessary to improve checkpointing in Lipizzaner. This is the process of saving intermittent data in the event an experiment is killed or paused and needs to be resumed at a later time.

Saving a Checkpoint

When saving checkpoints for large grid sizes, the strictest limitation is memory. For smaller grids, saving a handful of files and network models does not use much disk space; however, when attempting to store information about over 100 network models, it is necessary to consider constraints. As a result, the goal for saving checkpoints was to ensure that no redundant or repeating information gets stored.

We update the code to checkpoint data so as to only store information about the

center grid in a neighborhood, as opposed the original method that saved information about all models in the neighborhood. Upon re-starting training, each cell contacts its neighborhood and pulls information from them, which means it is not necessary for one cell to store the information of its neighbors. Furthermore, file compression was implemented to reduce the disk space of checkpointing even further.

The frequency of checkpointing is a configurable setting in Lipizzaner, called the *checkpoint_period*. When training, a checkpoint gets saved every *checkpoint_period* iterations.

Reading a Checkpoint

The default behavior when initializing a Lipizzaner client is not to read checkpoints, but to update random initial settings. However, if we want to re-start a paused experiment, the desired functionality is to resume training with the same network settings that were last saved. A first pass at reading checkpoints is outlined in Algorithm 5.

Algorithm 5 Reading a Checkpoint File

```
1: if checkpoint_files exists then
2:     checkpoint_path ←' checkpoints/[client_cell_number].yml'
3:     with uncompress_and_open(checkpoint_path) as f :
4:     self.generator, self.discriminator ← parse_checkpoint(f)
5: else
6:     self.generator, self.discriminator ← None, None
7: end if
```

With larger grid sizes, iterations of training could become out of sync, leaving a small number of clients far behind the rest. Therefore, these clients may not have completed saving checkpoints when the experiment was halted. To avoid errors reading any such files, exception handling was added to checkpoint parsing, with the default behavior being used if an error was caught. While this is not an ideal solution, errors parsing checkpoint files were rare(approx. 2% of clients), therefore there are enough other nodes to produce successful results.

Configurable Grid Sizes

Further, it is difficult to guarantee the exact number of clients that will successfully connect on Summit. As mentioned in section 3.1, Lipizzaner client connection has a small probability of failure that impacts experiments of large grid sizes. As a result guaranteeing that successive jobs will successfully connect the exact same number of clients is not always possible, which can cause inconsistency issues when an experiment requires multiple rounds of checkpointing to complete. To mitigate this, we implement configurable grid sizes.

We add the fields $max\_clients$ and $min\_clients$ to the configuration files that dictate training parameters and update the Lipizzaner master code according to Algorithm 6. This range can be adjusted to be as wide or narrow as the user desires, or disregarded completely. This, combined with checkpoint error handling, ensures that the majority of nodes in a large experiment successfully parse checkpoint data.

Algorithm 6 Grid Size Range

$ac \leftarrow successful\ clients$
$num\_clients \leftarrow length(ac)$
if $num\_clients > max\_clients$ **then**
    $ac \leftarrow successful\ clients[: max\_clients]$
end if
# $[Rectangular\ grid\ handling]$
if $num\_clients < min\_clients$ **then**
    $log\ critical\ error$
    $terminate\ all\ clients$
end if

Chapter 4: Experiments

This section outlines experiment setup and results for two datasets: MNIST and Covid-19 positive chest X-rays. With these results, we aimed to answer:

- *Can Lipizzaner run efficiently on HPC environments?*
- *What parts of Lipizzaner break when run at large scales?*
- *What improves when running Lipizzaner at large scales?*
- *Can larger scales improve performance of Lipizzaner on a dataset of Covid-19 positive chest X-ray images?*
- *Can scale overcome a lesser choice in network architecture in the event of hardware limitations?*

4.1 Experiment Set-up

Preliminary experiments were performed using MNIST, a database of images of handwritten digits from 0-9. It is commonly used in machine learning research and has previously been used in experiments with Lipizzaner. This made it a natural choice as a primary database for this thesis. Additionally, experiments were performed using a dataset of COVID-19 positive chest X-ray images. We sought to extend upon the research presented in [5] and determine if we could observe the same improvements seen in the MNIST database. The dataset contains 190 images and SMOTE augmentation[8] is used to enhance this.

Experiment settings for both datasets can be seen in Table 4.1. As we were interested in how scale compares with network complexity, we conducted experiments of various network architectures for both datasets. Network topologies for MNIST and COVID-19 experiments can be found in Table 4.2 and Table 4.3, respectively. [5] uses a CNN for 128x128 size images for experiments; however, the network for this was too large for the memory constraints on Summit. As a result, we tested with two smaller CNNs, one for 28x28 images and a second for 64x64 images. We refer to these as CNN28 and CNN64 respectively. Each summit node has 6 processors and, as such, it is simpler to request allocations in multiples of 6. Therefore, grid sizes vary at intervals of 6 across experiments.

Table 4.1: Experiment Settings

Parameter Name	**MNIST**	**COVID-19**
Coevolutionary Parameters		
Iterations	200	200
Population size per cell	1	1
Tournament Size	2	2
Grid Size	See Results Table	
Mixture Mutation Scale	0.01	0.01
Network Settings		
Activation function	*tanh*	*tanh*
Loss function	BCE loss	BCE loss
Hyperparameter Mutation Settings		
Optimizer	Adam	Adam
Initial Learning Rate	0.0002	0.0002
Mutation Rate	0.0001	0.0001
Mutation Probability	0.5	0.5
Training Settings		
Batch Size	100	70

Table 4.2: MNIST Neural Network Topologies

Network Type	**4LP**	**CNN**
Input Neurons	64	100
Number of hidden layers	2	3
Neurons per hidden layer	256	512, 256, 128
Total Parameters	512	896
Output neurons	784	784

Table 4.3: COVID-19 Neural Network Topologies

Network Type	**4LP**	**CNN28**	**CNN64**
Input Neurons	64	100	100
Number of hidden layers	2	3	4
Neurons per hidden layer	256	512, 256, 128	512, 256, 128, 64
Total Parameters	512	896	960
Output neurons	784	784	4096

4.2 Results

We report data collected at various grid sizes and network topologies, as well as sample generated images from each dataset used and statistical significance tests.

4.2.1 MNIST

MNIST experiments were evaluated using FID score[12], thus a lower score is desired. Table 4.4 shows scores for each grid size over a four layer perceptron network. The 3x3 results obtained here are similar to those obtained in [11]. Overall, larger grid sizes are seen to improve the outcome of training. Table 4.5 presents the p-values from computing the tie-corrected ranksum between each pair of grid size samples collected.

At grid sizes larger than 18x18, training began to throw critical errors, suggesting that more error handling may be required to continue scaling up. For the purposes of this thesis, results obtained from 18x18 grids were sufficient. As grid sizes increase,

we see a decrease in the improvement in FID score. (e.g. The difference in average scores between 9 and 36 clients is about 15, but only about 5 between 36 and 144 clients and 2 between 144 and 324 clients) This suggests a limit to how well Lipizzaner can perform with the four layer perceptron model. See Figure 4-1 for a sample of generated images from each grid size.

Table 4.4: MNIST 4LP FID Scores and Runtimes(in minutes)

Grid Size	Clients	Trials	Mean	SD	Min	Max	Avg. Time
3x3	9	30	41.23	2.36	36.87	47.64	83
6x6	36	30	24.63	2.35	20.04	29.16	80
12x12	144	15	19.89	2.54	15.41	25.91	81
18x18	324	8	17.20	1.61	14.72	19.65	80

Table 4.5: MNIST 4LP Rank-Sum P-Values

	6x6	12x12	18x18
3x3	$3.02 * 10^{-11}$	$6.46 * 10^{-8}$	$1.88 * 10^{-5}$
6x6		$1.05 * 10^{-5}$	$1.88 * 10^{-5}$
12x12			$1.83 * 10^{-2}$

While the scale of Lipizzaner does provide significant improvements to the four layer perceptron model, we question whether this is the best model to use. By using the CNN network topology, we can begin to compare scale and complexity. We found that even on a 3x3 grid, the CNN achieves an average FID score of 2.54 over 2 experimental runs, which is less than the best score found for an 18x18 grid using four layer perceptron. See Figure 4-2 for generated sample images. Further, on 2 experimental runs of a 6x6 grid, we found an average score of 2.36. As we were not able to obtain a statistically significant number of trials in the interest of time, we cannot make conclusive statements about these findings; however, they do suggest that, given how well the CNN topology performs at a small scale, there are limits to what improvements scale can bring about.

(a) 3x3 Grid

(b) 6x6 Grid

(c) 12x12 Grid

(d) 18x18 Grid

Figure 4-1: Generated MNIST Images: 4 Layer Perceptron

Figure 4-2: Generated MNIST Images: CNN [5]

4.2.2 COVID-19

As GPU memory constraints prohibited use of a 128x128 CNN for experiments, we used the three alternative network topologies outlined above. Both CNN experiments required re-sizing the images, which was expected to worsen the overall quality of images and output. Table 4.6 is a table with a breakdown of results for each network at different grid sizes, with data collected over 6 runs of each grid size.

The reported scores in Table 4.6 are generated using inception score[14], therefore, larger scores are desired. We observed a decrease in performance at the 12x12 grid sizes on the four layer perceptron model. The reason for this remains unknown and requires further research. Overall, the less complex networks generated images that were not as effective as those generated in [5]. This implies that perhaps scale is not enough to completely overcome the loss of network complexity. However, in both CNN experiments, we do see improvements in both inception score and image quality as grid size increases, which implies that scale can mitigate the effects a poor network. Table 4.7 presents the p-values from computing the tie-corrected ranksum between each pair of grid size samples collected for each network.

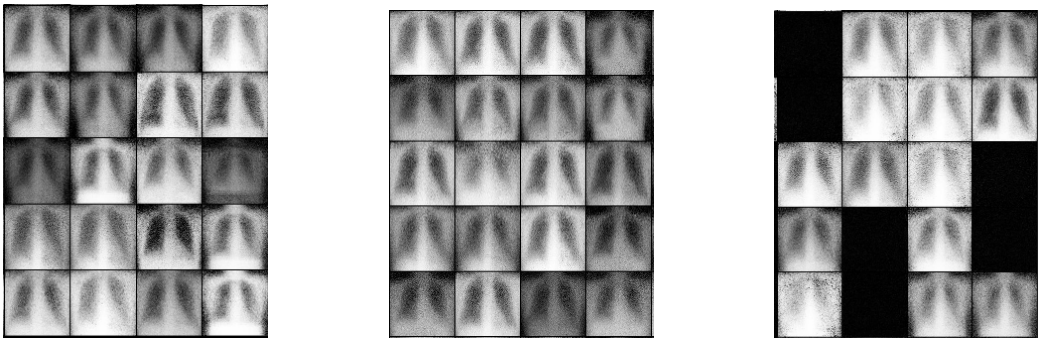
Table 4.6: Covid-19 Inception Scores and Runtimes(in minutes)

Network Type	Grid Size	Clients	Trials	Mean	SD	Min	Max	Avg. Time
4LP	3x3	9	6	1.26	0.046	1.18	1.32	138
	6x6	36	6	1.47	0.035	1.41	1.52	164
	12x12	144	6	1.21	0.016	1.19	1.24	166
CNN28	3x3	9	6	1.31	0.021	1.28	1.34	197
	6x6	36	6	1.46	0.019	1.44	1.50	203
	12x12	144	6	1.49	0.013	1.47	1.51	208
CNN64	3x3	9	6	1.37	0.011	1.36	1.39	215
	6x6	36	6	1.53	0.016	1.50	1.55	219
	12x12	144	6	1.61	0.011	1.59	1.62	226

Table 4.7: Covid-19 Rank-Sum P-Values for Different Networks

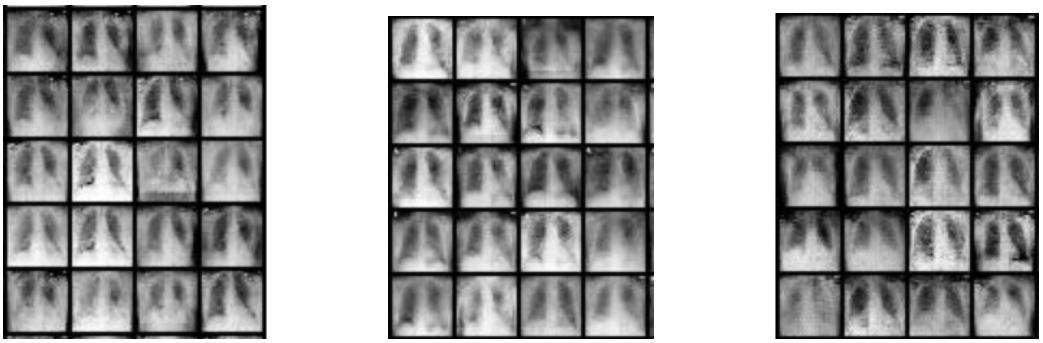
Network Type		6x6	12x12
4LP	3x3	$5.07 * 10^{-3}$	$9.16 * 10^{-2}$
	6x6		$4.92 * 10^{-3}$
	12x12		
CNN28	3x3	$4.99 * 10^{-3}$	$4.99 * 10^{-3}$
	6x6		$6.36 * 10^{-2}$
	12x12		
CNN64	3x3	$4.85 * 10^{-3}$	$4.77 * 10^{-3}$
	6x6		$4.85 * 10^{-3}$
	12x12		

As we can see in Figure 4-3, the four layer perceptron network model could not properly generate the details of the X-Ray images. We also see the impact of mode collapse on the 12x12 grid generated images. In both Figure 4-4 and Figure 4-5, we see improvement as grid sizes increase. However, when we compare these images with those presented in [5](See fig. 4-6), the overall quality is worse. Further, the average inception score for the CNN of size 128x128 run on a 3x3 grid was 1.83. This is significantly better than the values achieved through lesser models on Summit, regardless of grid size. This shows scale may not be able to completely outweigh network complexity, but does provide enhancements in the face of poor network conditions.



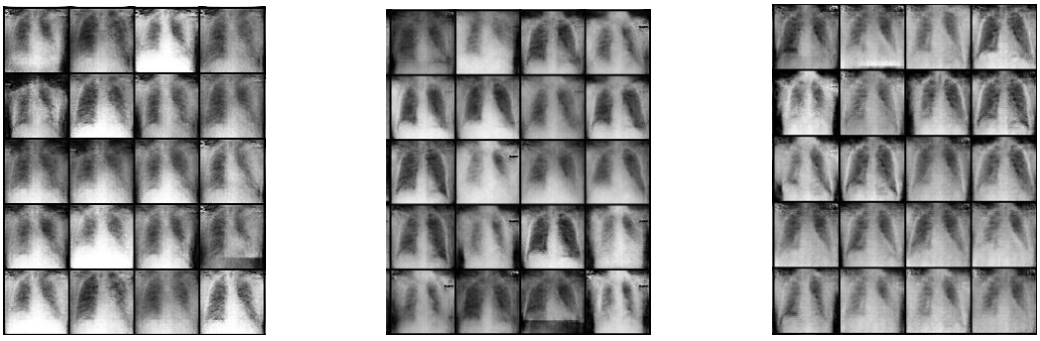
(a) 3x3 Grid (b) 6x6 Grid (c) 12x12 Grid

Figure 4-3: Generated COVID-19 Images: 4 Layer Perceptron



(a) 3x3 Grid (b) 6x6 Grid (c) 12x12 Grid

Figure 4-4: Generated COVID-19 Images: 28x28 CNN



(a) 3x3 Grid (b) 6x6 Grid (c) 12x12 Grid

Figure 4-5: Generated COVID-19 Images: 64x64 CNN

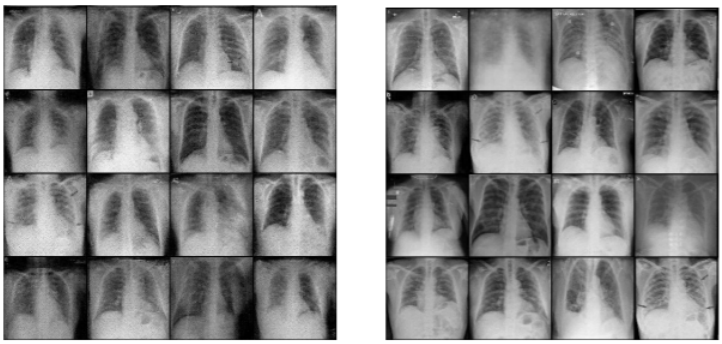


Figure 4-6: Generated Covid-19 Images: 128x128 CNN [5]

Chapter 5: Discussion

In this section, we discuss lessons learned about GANs and HPC computing environments over the course of this thesis.

5.1 GANs

Within the Lipizzaner framework, there are two methods to analyzing the success of an experiment. The first is by observing the images produced by the best grid space and the second is to check the FID or inception score of the best grid space. Without a deeper understanding of GANs, it is easy to take a good score as evidence that a GAN was successful. However, the score is a reflection of how well the generator was able to "fool" the discriminator, not how well the generator actually performed. Therefore, if the discriminator is so poorly trained that the generator fools it by default, then the generator will end with a good score, despite not being able to produce quality images.

Additionally, the course of this research showed that there is a point at which performance enhancements plateau with respect to grid size. Referring back to section 4.2, the difference in FID score for MNIST experiments becomes smaller with each grid size interval. While we may have observed improvements by continuing to increase grid size for MNIST experiments, we decided that we had answered research questions concerning scale with the 18x18 results. It was more important to pivot to new datsets and the research questions that come along with them.

5.2 HPC Environments

Over the course of this thesis, we have gotten the chance to work with very cutting edge HPC environments and utilize the resources they come with. However, there were quite a few difficulties that came with working in these environments and many lessons learned.

The first of these is being at the mercy of a queuing policy to run experiments. Depending on the availability and demand of the compute nodes, wait times for running an experiment could range anywhere between a few minutes to several hours. Additionally, there were also occasional days when the entire system would be down either for routine maintenance or poor network conditions. These issues caused some setbacks, particularly when still in a development phase of research. Testing and debugging can become increasingly difficult when one must wait an unspecified amount of time to see the results. Unfortunately, there is not much that can be done to resolve this issue.

Further, HPC systems, such as the one the Summit Supercomputer runs on, have various mechanisms in place for maintaining clean directories and only storing what appears necessary. One of these such mechanisms is garbage collection, in which all files that have not been touched in some threshold amount of time get wiped from directories. Given that the Lipizzaner framework is so large and my research only required modifications to a handful of files, there were times we would attempt to run experiments only to find that the majority of the files needed had been deleted. As this process did not occur often, we grew accustomed to committing and pushing modifications frequently to our GitHub repo, so that a pull could easily restore the environment.

Additionally, remote systems each have their own specifications and constraints, which can make the process of running the same experiments in two distinct systems very

different. This can lead to difficulties debugging, as one error may be impossible to replicate in a different system or one system may have enough resources to perform a task but another may not.

Chapter 6: Conclusion and Future Work

Upon completion of research, implementation, and testing for this thesis, we have learned that the spatial grid of Lipizzaner scales very well when the necessary computing resources are available. With the MNIST dataset, we observed significant improvements in FID scores and image quality at grid sizes of 6x6, 12x12 and 18x18, when compared with the baseline 3x3 grid. With the COVID-19 dataset, improvements were also seen in most cases, though perhaps not as significant as those seen with MNIST. This evidence shows that, in the face of network and/or hardware limitations, scale can improve Lipizzaner's results. However, these improvements do have limitations and thus are not enough to completely make up for reduced network complexity. These conclusions have opened up many more questions about Lipizzaner and GANs.

Firstly, we are interested in continuing work on Summit. We would like to increase grid size with the MNIST dataset to find what the true peak of performance is. Given that Summit has upwards of 4,000 compute nodes, we can run grid sizes up to 150x150; therefore, this thesis has only scratched the surface with what is possible. Further, there are many other meaningful datasets to scale up with Summit that would provide good extensions of this work.

Checkpointing continues to be an open area of research and development within Lipizzaner. The nature of checkpoint implementation contributed by this thesis is very specific to the use case of needing to pause an experiment and re-start it from that spot. As a result, it may not cover other cases in which checkpointing would be

helpful, such as single client failure. Further work can be done to generalize checkpointing to be able to handle cases like this. Additionally, there is room for more complex error handling when reading checkpoints and the possibility of storing multiple checkpoints for one grid space, instead of overwriting an old checkpoint at every new one.

Further, as there were memory constraints on Summit that prevented use of the ideal network parameters, a true comparison to baseline results was not able to be obtained. The Satori cluster at MIT has looser memory constraints and is able to run the full network model, but at the time of writing is in its beta testing phase and can only accommodate up to 4 clients(i.e. 2x2 grids). Once it is possible to utilize more resourced from the Satori cluster, it would be interesting to run experiments with the full network model on larger grid sizes.

Bibliography

[1] Lipizzaner open source. https://github.com/ALFA-group/lipizzaner-gan. Accessed: 2020-12-01.

[2] Lipizzaner open source. https://tig.csail.mit.edu/shared-computing/openstack/. Accessed: 2020-12-01.

[3] Lipizzaner open source. https://mit-satori.github.io/index.html. Accessed: 2020-12-01.

[4] Lipizzaner open source. https://www.olcf.ornl.gov/summit/. Accessed: 2020-12-01.

[5] J. Toutouh E. Hemberg AE. Perez, S. Nesmachnow and U. O'Reilly. Parallel/distributed implementation of cellular training for generative adversarial neural networks. *10th IEEE Workshop Parallel/Distributed Combinatorics and Optimization*, 28(3), 2020.

[6] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks, 2017.

[7] Sanjeev Arora, Andrej Risteski, and Yi Zhang. Do GANs learn the distribution? some theory and empirics. In *International Conference on Learning Representations*, 2018.

[8] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique, 2002.

[9] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[10] Jingoo Han, Luna Xu, M. Mustafa Rafique, Ali R. Butt, and Seung-Hwan Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

[11] Erik Hemberg, Jamal Toutouh, Abdullah Al-Dujaili, Tom Schmiedlechner, and Una-May O'Reilly. Spatial coevolution for the robust and scalable training of generative adversarial networks, Aug 2018.

[12] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

[13] Jerry Li, Aleksander Madry, John Peebles, and Ludwig Schmidt. Towards understanding the dynamics of generative adversarial networks. *CoRR*, abs/1706.09884, 2017.

[14] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[15] Jamal Toutouh, Erik Hemberg, and Una-May O'Reilly. Analyzing the components of distributed coevolutionary gan training, 2020.

[16] Yuan Xue, Tao Xu, Han Zhang, L. Rodney Long, and Xiaolei Huang. Segan: Adversarial network with multi-scale l1 loss for medical image segmentation. *Neuroinformatics*, 16(3-4):383–392, May 2018.

[17] Houssam Zenati, Chuan Sheng Foo, Bruno Lecouat, Gaurav Manek, and Vijay Ramaseshan Chandrasekhar. Efficient gan-based anomaly detection, 2019.