The University Of Iowa Computer Science Ph.D. program Qualifing Exam Fall 2021

Jamil Gafur

September 12, 2021

1 Introduction

In this report, we will cover two different Heuristic Swarm optimizers the Particle Swarm Optimizer (PSO) [1], and the Gravitational Swarm Algorithm (GSA) [2]. We then see how the PSO can be used to train a Generative Adversarial Neural Networks (GAN) [3].

We first cover the background needed to understand the original GAN. This is done by introducing AutoEncoders. AutoEncoders have the ability to compress data into a Latent Feature Space. This ability to learn a Latent Feature Space is important because the GANs architecture takes advantage of this to generate output similar to its input. The GANs architecture is made up of two individual Networks. One Network determines if an input is real or generated, while the other generates data that is similar to the input dataset. This architecture has been shown to have issues with the vanishing/exploding gradient problem as well as mode collapse.

To mitigate these issues, three modifications have been proposed. The Evolutionary GAN (EGAN) [4] changes the training process. The Least Squared GAN (LSGAN) [5] changes the objective function. Finally, the Deep Convolutional GAN (DCGAN) [6] which updates the architecture.

We then introduce two different Heuristic Swarm Optimizers. These types of optimizers initialize random candidates in an optimization plane. They then update the candidates based on converging nature-inspired events. Some common events include Ants searching for food [7], Gravity [8], Evolution [9], and Bees to a flower [10].

Two of our focus papers cover Heuristic Swarm Algorithms. The first is the PSO [1]. This optimizer takes inspiration from a flock of birds searching for food. It was one of the first swarm algorithms to be developed and laid a foundation for future work. However, this algorithm did not fully utilize every candidate in the Swarm. This leads us to look for a more comprehensive Swarm algorithm.

The next focus paper is based on the GSA [2]. This optimizer utilizes the laws of physics to converge to an optimal value. Unlike the PSO, this algorithm takes into account each candidate by utilizing the laws of gravity. In the paper [2], the authors do a comparative study of the PSO, GSA, and a variation of the GSA.

We then transition to the third focus paper [3]. This paper presents a GAN that is an accumulation of the three previously discussed modifications. However, instead of using the evolutionary process to train the network, it uses the PSO algorithm. These adaptions show that while using the same data, we can obtain better results, and reduce the computational cost.

While Neural Networks as a whole are described later, the training process is broken into a few steps. First, the Neural Networks error is determined based on an error metric. Then the Network minimizes its error by updating its weights and biases. Once the model has a low enough error, it is then tested on an independent dataset to establish accuracy. The Network then can be used to make predictions, classifications, or other decisions.

2 Neural Networks

Inspired by the human brain's Neural System, the artificial Neural Network has become a tool for data scientists to use in a wide range of domains. Neural Networks are complex interconnected networks with a myriad of design setups and parameters that can affect their performance. A Neural network is an interconnected grouping of simple processing units (commonly referred to as neurons) that form a larger grouping (known as layers). Each neuron has a connection strength (or weight), and bias; while each layer has an associated activation function. These weights and biases are updated through training on input data. The collection of these layers are known as a Neural Networks architecture.

By changing the layers of a Neural Network and its activation functions, scientists can manipulate the Network to solve different problems. But, being adaptable is not enough to justify their frequent use; it is the fact that these Neural Networks can extract relevant knowledge from the data passed through it and apply this knowledge with great accuracy to new data.

This ability to learn from existing data and apply it in real-time to new data has become a driving force for their popularity. In 2020, scholars estimated that around 40 trillion gigabytes of data (40 zettabytes) have been produced and stored. Yet, with more data comes more responsibility; many machine learning applications turn into computationally heavy tasks. This can be due to the complexity of the data or the architecture can be so complex that they are replaceable with more simplistic models.

The term Neural Networks has taken the scientific community by storm. This is due to their ability to learn and extract underlying information from data. To better understand this concept, we can first look at the foundation on which they are based - the human brain.

It is estimated that there are 10^{11} *Neurons* in the human brain. These Neurons transfer information through electrical signals via the connecting synapses [11]. Figure 1 shows the passing of an electrical signal between two Neurons. Figure 2 shows a computational representation of two Neurons.

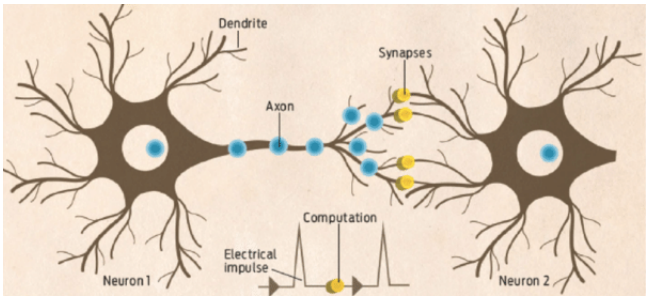

Figure 1: Two Neurons communicating to each other via electrical signals [11]

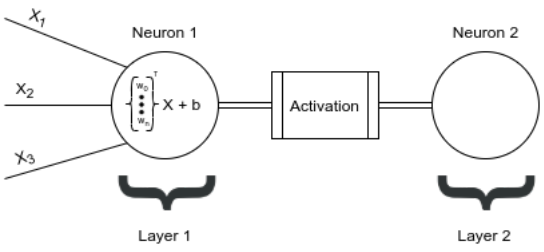

Figure 2: Two computational Neurons communicating to each other via an activation function

Each set of Neurons make up a layer in the Neural Network. The Network is a nested equation of weights, biases, and activation functions. Equation 4 represents the output of the whole Neural

Network.

$$\mathbf{W} = (W^1, W^2, \ldots, W^m) \tag{1}$$

$$\mathbf{b} = (b^1, b^2, \ldots, b^m) \tag{2}$$

$$\mathbf{a} = (a^1, a^2, \ldots, a^m) \tag{3}$$

$$y = f^m(\mathbf{x}; \mathbf{W}, \mathbf{b}) = a_m(W^m \cdots a_2(W^2 a_1(W^1 x + b^1) + b^2) \cdots) + b^m) \tag{4}$$

A single activation function transforms its input into a scalar value that is used as input for the next layer. These functions determine how activated their respective Neuron should be. A Neuron is considered active based on whether its input is relevant for the models' prediction.

When choosing an activation function, they must be computationally efficient and have easy to calculate derivatives. The derivative is used to calculate the gradient and update the layer's weights and biases. Two common problems found when updating the weights and biases are the vanishing gradient problem (Equation 5) and the exploding gradient problem (Equation 6).

Given a single activation function, a_i, over time,t, the vanishing gradient problem states that the gradient approaches zero. This means that while the Network is trying to update its weights and biases, the changes are minimal. When all the weights and biases in a layer have extremely small updates between training iteration, this is known as a "dead" layer. "Dead" layers are no longer useful in optimizing the Network. The exploding gradient is the opposite, where the gradient approaches infinity.

$$\lim_{t \to \pm\infty} \nabla a_i(t) = 0 \tag{5}$$

$$\lim_{t \to \pm\infty} \nabla a_i(t) = \infty \tag{6}$$

Some common activation functions are Sigmoid, Rectified Linear Unit (ReLU), and Leaky ReLU (LReLU). The Sigmoid function is defined in Equation 7. While this function is easily differentials, it can allow the Network to fall into the vanishing gradient problem. The Sigmoid activation function exists between zero and one, thus can be used for Networks that need to predict the probability of an input for a given classification.

$$S(x) = \frac{1}{(1 + e^{-x})} \tag{7}$$

$$g(x) = max(ax, x) \tag{8}$$

The ReLU and LReLU functions can be depicted in Equation 8; where ReLU sets $a = 0$ and $0 < a < 1$ for LReLU. This activation function will turn off Neurons and then allow the active ones to be kept for future training with the next layer's activation functions. However, this can lead to the Dying ReLU problem, where all the Neurons have a negative input and are turned off.

To solve this problem, the Leaky ReLU Function was developed. This variation of ReLU has a small positive slope on the left side of the graph, thus allowing for a negative gradient.

Another common complaint that comes with using Neural Networks is the need for large quantities of data, their "black box" design, and the output they produce. To train a Neural Network, a

large amount of *diverse* data is needed so that the weights and biases can be updated accordingly. If the data is skewed or small then the Network either learns the skew or overfits the data.

As previously stated the Neural Network is a grouping of the smaller set of Neurons; each set of Neurons is called a layer and building up different layers creates the Network. The first layer in the Network is called the input layer, the last layer is the output layer and any layer between them is considered a hidden layer.

The Neural Network is considered a black box because it can approximate any function, however, there is no way to understand its approximation. Since the Neural Network is a black box, sometimes it can produce non-realistic but "technically" correct solutions. This becomes an issue for debugging because the computations done in the hidden layers are difficult to represent. Finally, Neural Networks are considered very rigid in its design.

Once the input and output layer has been initialized, they can not be changed. For example, given a gray-scale image of 32 pixels by 32 a Neural Network can have an input layer of 1024 Neurons; each of which references an individual pixel grey-scale value. On the other end of the Network, the output layer can be used to generate a new image by having 1024 Neurons that correspond to the gray-scale value for each new pixel (known as a Generator), or it can have c Neurons where c represents a set number of classifications that the picture can be classified as (known as a Classifier). By changing the input layer, we change the number of Neurons the Network takes in, and by changing the output layer, we change what the Network is doing.

One Neural Network we will look at is an AutoEncoder. The AutoEncoder attempts to compress and then reconstruct its input with minimal error. This is done by decreasing and then increasing the number of neurons in each hidden layer. This method allows the Network to represent the data in a Latent Feature Space.

2.1 AutoEncoder

AutoEncoders are unsupervised Neural Networks that compress their input into smaller representations. They are considered unsupervised networks because its data does not need any labels. Rather, they attempt to recreate the data it is trained on. The input data passes through the Network into layers that have a gradually smaller number of Neurons. This section of the Network is the Encoder. From here the Network then has layers that gradually grow in size, this section is the Decoder. The point where the layers transition is the "bottleneck". An illustration of an AutoEncoder is shown in Figure 3.

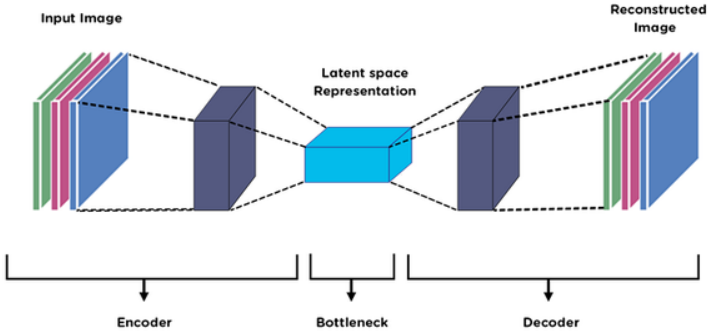

Figure 3: A visual representation of an AutoEncoder[12]

The purpose of an AutoEncoder is to compress its input data into a Latent Feature Space, then recreate it with minimal error. These Networks attempt to learn approximations of the identity function. While this seems particularly trivial to learn, by placing constraints on the Network, it can discover interesting features from the input data. These constraints are given from the Encoding and Decoding sections of the Network.

To offer an example, suppose the input set are images, where one image is a gray-scale value that is 10 by 10 in size, totaling 100 pixels. If we allow the input layer to have 100 Neurons, in the Encoding section of the AutoEncoder, let the second layer, or the first hidden layer, have 50 Neurons.

Since there are only 50 Neurons, the Network is forced to learn a compressed representation of the input. Then as we move towards the Decoding section, the AutoEncoder will have to take the saved features and rebuild them back into the original image. If the input were completely random, then the compression task would be very difficult. If the input features are correlated, then it attempts to discover correlations during compressing.

The ability of a Neural Network to learn a Latent Feature Space of some data is extremely useful. This ability is heavily used in both the Discriminator and the Generator of a GAN. The Generator is similar to the encoding section, where it learns the Latent Feature Space. While the Discriminator is similar to the decoding section, as it learns to recreate images with minimal error.

2.2 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a type of Neural Network that is comprised of two smaller Networks. The first Neural Network is called the Generator. The Generator takes in randomly generated input and attempts to converts it into an output with the same Latent Feature Space as the test dataset. The other Neural Network is the Discriminator, it learns the Latent Feature Space of the dataset and determines if its input is from the original dataset or a generated image. This architecture was shown to be well suited for image generation. An illustration of a GAN is given in Figure 4.

The Generator Network is similar to an AutoEncoder, in that it attempts to learn the Latent Feature Space of the dataset. The Generator "knows" that it is learning the correct Latent Feature Space when the Discriminator accuracy decreases.

The Discriminator is a classifier that is trained to learn a Latent Feature Space of the real dataset. It takes the output of the Generator and determines if the generated output is in the same Latent Feature Space as the data it learned. During training, the parameters of each model are independently updated.

The original GAN's objective function [5] is given in Equation 9.

$$\min_D \max_G V(G, D) \text{ where} \tag{9}$$

$$G = G(\mathbf{z}; \theta_g(t)) \tag{10}$$

$$D = D(\mathbf{x}; \theta_d(t)) \tag{11}$$

$$V(G, D) = E_{x \sim P_{data}(x)}[log(D(x))] + E_{z \sim P_z(z)}[log(1 - D(G(z)))] \tag{12}$$

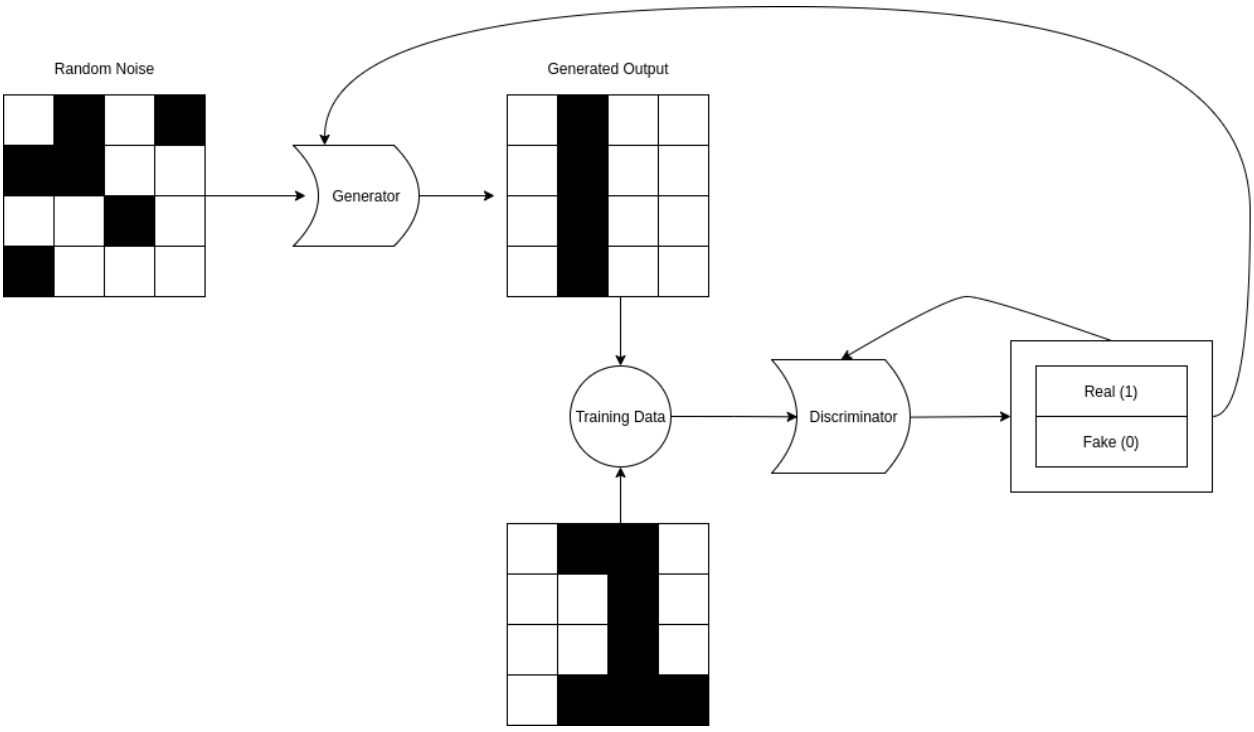

Figure 4: Example of the Generator and Discriminator in a GAN

To understand this equation, it can be broken into two separate parts;

$$E_{\mathbf{z}\sim P_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z})))] \tag{13}$$

$$E_{\mathbf{x}\sim P_{data}(\mathbf{x})}[logD(\mathbf{x})] \tag{14}$$

Equation 13 refers to the Generators ability to generate output in the same latent feature space as the dataset. Given some random noise $\mathbf{z}$, and the current parameters $\theta_g(t)$, the Generator creates an output, $G(\mathbf{z}; \theta_g(t))$. This output is then fed into the Discriminator which tries to classify it as a generated image or not. This function is trying to maximize the rate of false positives in the Network.

Equation 14 references the Discriminator's ability to determine if the predicted data (x) is real given the current parameters $\theta_d(t)$. Thus, a random input is passed to the Generator, which creates a fake output. This fake output is then sent to the Discriminator. This rate of false-positive is what the Equation is trying to minimize.

The total expected objective function value of the GAN is based on these two Networks working against each other. The Generator tries to cause the Discriminator to give false positives. The Discriminator trying to not give false positives.

This architecture has a few problems associated with it. First, if the accuracy of the Generator and the Discriminator are far apart, then the other Network would not be able to extract enough information from the other output. This in turn will force the Network to stop learning. Another issue is that the Generator can generate a single image; this is known as mode collapse.

To fix these problems three different GAN architectures have been proposed. These architecture are the Deep Convolutional GAN (DCGAN)[6], Least Squared GAN (LSGAN)[5] and the EGAN[11].

2.2.1 Deep Convolutional GAN (DCGAN)

The first derivation of the original GAN that we will look at is the Deep Convolutional GAN (DCGAN) [6]. This GAN changes the architecture of the Network to provide stable training for image generation.

The first change is to add convolutional layers to the Generators architecture. A convolution is a matrix filter (or kernel) that is multiplied by the input data. The product of this kernel is a smaller representation of the input data, commonly referred to as a "downsampled" representation of the data.

As shown in Figure 5 a kernel is applied to adjacent subsections of a single input and condenses these subsections into a smaller feature space. This also reduces the number of trainable weights and biases from the total number of inputs to the kernel size. When optimizing the convolutional layer, the convolutional kernels are updated.

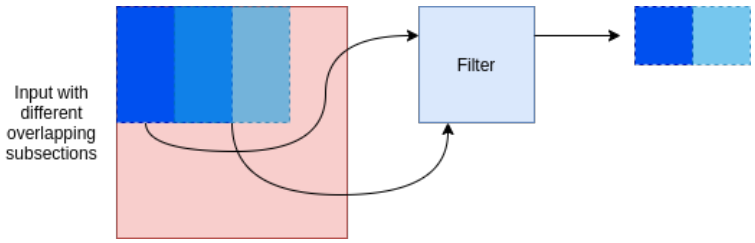

Figure 5: Example of a kernel filter applied to segments of an image to create a down sampled representation

After adding the convolutional layers to the Generator architecture, all fully connected layers following the convolutional layers were removed. These fully connected layers were replaced by connections of the highest convolutional features. These connections allowed the Generator to learn its downsampling. From here all of the activation functions were updated to ReLU, except for the output layer.

Finally, the paper[6] suggests applying batch normalization to the input dataset for training. This is done by chunking the whole input dataset into different subgroups, these subgroups are modified to have a zero mean and unit variance. By doing so they minimize the chance of exponentially growing computations.

The authors trained this GAN on three different datasets; the Large-scale Scene Understanding (LSUN), Imagenet-1k, and a face dataset. The face dataset consisted of scraped images of human faces from random web queries to use as training data. These faces were detected with the OpenCV face detector and contained three million images of ten thousand people.

The ImageNet-1k dataset contains 14M images, with 22,000 different classifications. The LSUN dataset contains around one million labeled images for 10 unique scene categories and 20 object categories. The LSUN dataset was created by collecting random samples of unlabeled images. From there they had human labeling of different categories and passed the newly labeled data into a classifier. This classifier went through the dataset and propagated the labeling onto different unclassified data which was then checked by a human for correctness.

2.2.2 Least Square GAN (LSGAN)

The next modification of the original GAN is the Least Squared GAN (LSGAN) [5]. This GAN was developed to reduce the vanishing/exploding gradient problem, as well as create a more real-

istic output from the Generator by changing the objective function. It was speculated that these gradient-based problems were due to the Binary Cross-Entropy loss function given in the original GAN (Equation 9). For the original GAN, at the beginning of training, the generated images would be labeled as correct, but not have an error associated with "how correct" the output was.

To remedy this, the authors developed the Least Square loss function, that would associate an error for correctly identified inputs. This new objective function is given in Equation 15 for the Discriminator and Equation 16 for the Generator. This loss function provides a smooth, non-saturating gradient in the Discriminator.

$$\min_D VD_{LSGAN} = \frac{1}{2}\mathbb{E}_{p_{data}}[(D(\mathbf{x}) - b)^2] + \frac{1}{2}\mathbb{E}_{\mathbf{z}\sim\ p_\mathbf{z}}[(D(G(\mathbf{z})) - a)^2] \tag{15}$$

$$\min_G VG_{LSGAN} = \frac{1}{2}\mathbb{E}_{\mathbf{z}\sim\ p_\mathbf{z}}[(D(G(\mathbf{z})) - c)^2] \tag{16}$$

where VD_{LSGAN} (Equation 15) is the expected value for the Discriminator (D). This Discriminator's expected value is based on the squared error of the Discriminators prediction with respect to the real dataset b and the squared error of its prediction to the fake dataset a. Since the Discriminator's job is to make correct predictions, we want to minimize these two errors. The Discriminator's objective function attracts data generated by the Generator towards the real Latent Feature Space.

On the other hand, VG_{LSGAN} (Equation 16) is the squared error of the Discriminators' output, of the Generators output, given some random noise ($\mathbf{z}$) with respect to c. Where c is the value the Generator wants the Discriminator to produce for its fake data. During training, the authors set $b = 1$ to represent real data, $a = 0$ to represent fake data, and $c = 1$ to try to fool the Discriminator as much as possible.

By implementing this new architecture, the Generator can apply some confidence error to correctly predicted images, based on how "correct" they are. While previously, the original GAN's Generator would only know if it was correct or not.

2.2.3 Evolutionary Generative Adversarial Networks (EGAN)

Finally, the Evolutionary Generative Adversarial Networks (EGAN) is the last GAN that we look at paper [4]. Unlike the LSGAN this GAN does not change its objective function, rather, it changes the training process. The EGAN utilizes an evolutionary process to optimize the objective function. They define the Discriminator as the "environment", and a population of Generators that "evolves" over time to optimize the Discriminator. This population of Generators evolves using different training objectives. These objectives produce new candidates for the next generation of Generators.

Thus, multiple Generators are tested against a single Discriminator. Based on each of the Generator's performance they update their respective weights and biases. For each epoch, the unique Generators produce offspring. These offspring simulate the evolutionary process and are susceptible to random mutations.

During each evolutionary epoch, the Generators consists of three stages variation, evaluation, and selection. The variation step is an asexual reproduction of a single Generator G. G produces N_m children with different training objectives. These new training objectives attempt to optimize the individual child by offering a "different perspective".

For the variation step, the authors used three mutations (or objective) functions that the generator could receive. The first is the min-max mutations (Equation 17), then the heuristic mutation (Equation 18) and the least squared mutation (Equation 19).

$$M_G^{minmax} = \frac{1}{2}\mathbb{E}_{\mathbf{z}\sim p_z}[log(1-(D(G(\mathbf{z}))))] \tag{17}$$

$$M_G^{heuristic} = -\frac{1}{2}\mathbb{E}_{\mathbf{z}\sim p_\mathbf{z}}[log(D(G(\mathbf{z})))] \tag{18}$$

$$M_G^{LSGAN} = \frac{1}{2}\mathbb{E}_{\mathbf{z}\sim\ p_\mathbf{z}}[(D(G(\mathbf{z}))-c)^2] \tag{19}$$

From there the Evaluation step evaluates how well a Generator is doing. This is calculated by Equation 20. Where equation 21 represents how well the GAN is performing and Equation 22 represents the diversity fitness score.

$$F = F_q + \lambda F_d \text{ where} \tag{20}$$

$$\lambda \geq 0$$

$$F_q = \mathbb{E}_\mathbf{z}[D(G(\mathbf{z}))] \tag{21}$$

$$F_d = -log(||\nabla_D - \mathbb{E}_x[log(D(\mathbf{x}))] - \mathbb{E}_\mathbf{z}[log(1-D(G(\mathbf{z})))]||) \tag{22}$$

Equation 22 was developed in [13] to stabilize the GAN optimization while reducing mode collapse. It proposes a gradient-based regularization term that stabilizes the GANs original objective function.

Finally, the Selection step selects the Generators that offer the best accuracy and stability based on Equation 20. These selected Generators are allowed to move to the next epoch. An example of the overall process can be shown in Figure 6.

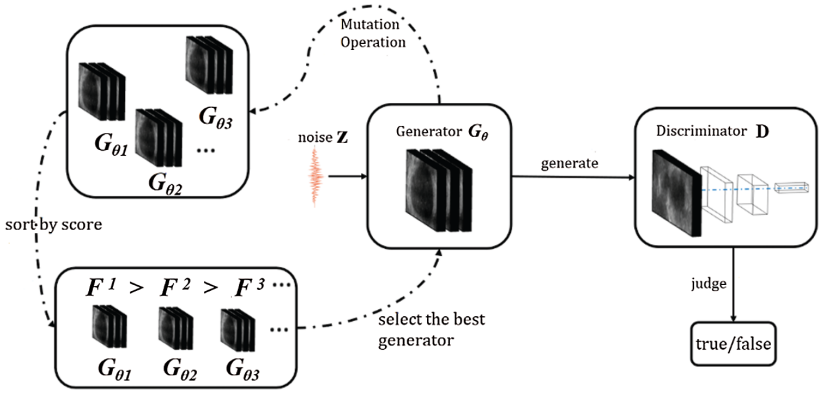

Figure 6: The structure of the Evolutionary GAN [14]

The data used for this paper is the CelebA dataset [15]. The goal of each generator was to generate high-quality 128x128 RGB human face images. The authors wanted to evaluate the embedding of the Latent Feature Space given some random vector $\mathbf{z}$. Therefore, given two Generated images G_1 and G_2 their respective latent vectors were recorded z_1 and z_2. The authors then performed a linear interpolation between z_1 and z_2 to generate new random vectors z_n. As these newly generated vectors were passed in, a transformation between G_1 and G_2 could be seen. An example is given in Figure 7.

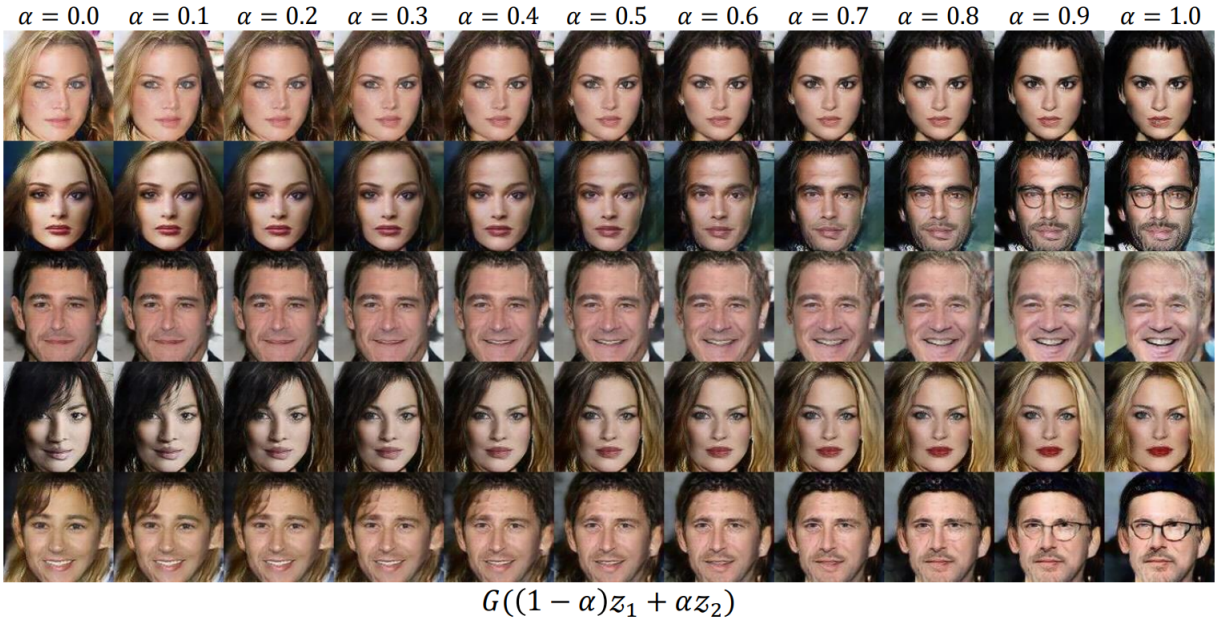

Figure 7: The images generated from a linear interpolation between two vectors z_1 and z_2 [14]

This GAN measures the generational performance of all offspring, and is based on the principle of survival of the fittest, the updated generators are selected as the parents for a new round of training. After each iteration, the Discriminator is updated to allow for a higher level of accuracy between the real data and the generated data by the evolved Generators. This Evolutionary training technique is an example of Heuristic Swarm optimization, which is covered in the next section.

3 Heuristic Swarm Optimization Techniques

Heuristic Swarm algorithms are used for finding an approximate solution for an objective function when general methods, like gradient descent, are too computationally expensive. These types of algorithms follow a similar format. First, a population S of random solutions p are scattered around the parameter space, at a given time t. From there each potential solution p_i is given a vector velocity V_i that updates its position for the next step at a given time. The difference between most algorithms is how this update vector is calculated.

These Swarm algorithms have been proven to be computationally inexpensive and create a balance between exploring the optimization space and exploiting the others particles in the Swarms.

$$\text{Given: } \min_x F(x) \text{ where} \tag{23}$$

$$x = \{x_0, \cdots, x_n\} \in \mathbb{R}^{n+1} \tag{24}$$

$$p_i(t) = x \tag{25}$$

$$Swarm(t) = S(t) = \{p_0, \cdots, p_m\} \in \mathbb{R}^{m+1} \tag{26}$$

$$p_{best}(t) = \underset{0 \leq s \leq t}{\operatorname{argmin}} F(p_i(s)) \tag{27}$$

$$g_{best}(t) = \underset{i}{\operatorname{argmin}} \underset{0 \leq s \leq t}{\operatorname{argmin}} F(p_i(s)) \tag{28}$$

$$g_{worse}(t) = \underset{i}{\operatorname{argmax}} \underset{0 \leq s \leq t}{\operatorname{argmin}} F(p_i(s)) \tag{29}$$

This balance is usually given by the "memory" of the particle. For minimization, this is given by communicating to the Swarm at each iteration its the best solution, Equation 27, for each particle p_i, as well as the global best, Equation 28, in the Swarm. We will cover two examples of Swarm optimization algorithms, the Particle Swarm Optimization (PSO) (Section 3.1) and the Gravitation Swarm Algorithm (Section 3.2).

3.1 Particle Swarm Optimizer (PSO)

The PSO algorithm takes inspiration from a flock of birds searching for food[8, 16]. Unlike the Genetic algorithm or Evolutionary algorithm which was used in Section 2.2.3, the PSO is based on the "social" behavior.

The PSO does not need to cycle through the Variation, Evaluation, and Selection steps. It also is not based on the idea of survival of the fittest. Rather, it's a collaboration between particles.

Each particle has a position p_i and a velocity v_i . The position is updated based on the velocity. The velocity is updated based on the previous velocity, the particle's best solution p_{best}, and the Swarm's best solution, g_{best}.

Given an objective function F , the g_{best} is given in Equation 28, while p_{best} is defined in Equation 27. For a maximization, they would be its inverse. The PSO can find optimal solutions with minimal memory requirements and has paved the way for new Swarm-based optimizers.

Calculating the new velocity is based on the social and cognitive behavior of the Swarm. This movement is defined in Equation 30.

$$\mathbf{V_i(t)} = (w \cdot \mathbf{V_i(t-1)}) + \mathbf{C_i(p_{best}(t), t)} + \mathbf{S_i(g_{best}(t), t)} \text{ where} \tag{30}$$

$$\mathbf{S_i(p_{best}}, t) = C_1 * Rnd_i(t, 0, 1) * [\mathbf{p_{best}(t) - p_i(t)}] \tag{31}$$

$$\mathbf{C_i(g_{best}}, t) = C_2 * Rnd_j(t, 0, 1) * [\mathbf{g_{best}(t) - p_i(t)}] \tag{32}$$

Where w represents the inertia as a positive scalar such that $0.4 \leq w \leq .9$. The inertia is used to reduce the weighting of the previous velocity on the newly calculated velocity. When using this method it is good to define how to set this variable; as will be shown in Section 4.

Equation 31 is calculated by the difference between a particle's own best position and its current position $p_i(t)$. As the particle moves away from its optimal position, the difference must increase. As C_1 increases it attracts the particle back to its best position.

The products $Rnd_i(t, 0, 1)$ and $Rnd_j(t, 0, 1)$ were introduced as a stochastic variable called *craziness* [8]. This variable was used to introduce variations into the Swarm. These variations give the simulation an interesting and "lifelike" appearance.

Finally, the third term, Equation 32, is the cognitive aspect of the optimizer. It has a similar format to Equation 31, but instead utilizes g_{best}. This Function moves the new vector towards the $g_{best}(t)$ value, which, we hope will change over time.

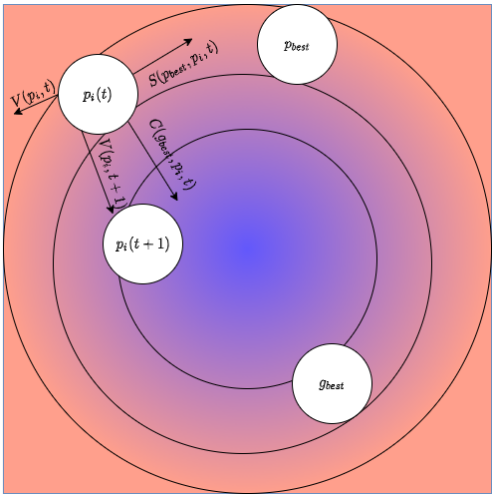

Figure 8: Example of a single particle velocity calculations for an iteration

For each iteration, the Swarm checks for a new g_{best} while each particle checks for a new p_{best}. The particles then update their position by adding their new velocity to their current position.

$$p_{i+1}(t) = p_i(t) + V_i(t) \tag{33}$$

The authors in [1] stated that there are a few disadvantages of this algorithm. First, the PSO suffers from a rise in search complexity as the dimensionality of the objective function increases. Then depending on the distribution, the Swarm can get stuck in a local minimum.

Despite these disadvantages, the PSO is designed to only be attracted to better solutions. Because of this the Swarms optimal value can never be worse than when it started. Furthermore, the calculations and implementation of this algorithm are very simple. This allows it to be implemented in a wide range of research and engineering applications.

Overall, this algorithm is extremely flexible and versatile for different use cases. However, since the velocity of each particle is only based on p_{best} and g_{best}, the Swarm as a whole is not fully utilized. In the next section, we will look at the Gravitational Swarm Algorithm. The GSA addresses this issue by simulating the law of Gravity and using every particle to update the position of a single particle.

3.2 Gravitational Search Algorithm (GSA)

The Gravitational Search Algorithm (GSA) [2] is a modification of the Particle Swarm Optimizer that is inspired by the foundation of Newtonian physic. Given two particles i and j with an assigned mass, they assert a gravitational pull towards each other. This Gravitational Force is proportional to the product of their masses and inversely proportional to the square of the distance between them. Such that the constant of proportionality is a universal constant.

To begin with, the authors define the universal gravity, and the force of gravity as given in Equation 34 and Equation 35.

$$G(t) = G_0(\frac{t_0}{t})^{-\beta} \tag{34}$$

$$\mathbf{GF}_i(t) = \sum_{j:j\neq i} G(t) \cdot Rnd_i(t,0,1) \cdot \frac{M_i(t) \cdot M_j(t)}{R_{ij}(t)^2} \cdot \frac{\mathbf{P}_i(t) - \mathbf{P}_j(t)}{R_{ij}(t)}, \qquad \text{where} \tag{35}$$

$$R_{ij}(t) = \|\mathbf{P}_i(t) - \mathbf{P}_j(t)\|_2 . \tag{36}$$

Where $G(t)$, is analogous to the universal gravitational constant. Given an initial gravity (G_0) it decays overtime at a rate of β where ($0 < \beta \leq 1$). For a given time, t, we can calculate a force of attraction of a single particle p_i concerning its position in the Swarm and its mass. This is then accelerated by the force of gravity and a uniform random number between zero and one at a given time ($Rnd_i(t,0,1)$).

After calculating $G(t)$, every particle calculates its new mass with respect to the objective function F (Equation 37). This mass is first scaled between g_{best} and g_{worse}. It is then normalized based on the total mass of the Swarm Equation 38.

$$m_i(t) = \frac{F(p_i(t)) - F(g_{worse}(t))}{F(g_{best}(t)) - F(g_{worse}(t))} \tag{37}$$

$$M_i(t) = \frac{m_i(t)}{\sum_j^{m+1} m_j(t)} \tag{38}$$

Given $\mathbf{GF_i(t)}$, the laws of physics describe the rate at which the particles move, or "acceleration" is defined as Equation 39.

$$\mathbf{A_i}(t) = \frac{\mathbf{GF_i(t)}}{M_i(t)} \tag{39}$$

The algorithm then defines the newest velocity as the sum of the product of the current velocity and a uniform random number between zero and one with the acceleration of the particle towards a new direction, Equation 40.

$$\mathbf{V_i(t+1)} = Rnd_j(t,0,1) \cdot \mathbf{V_i(t)} + \mathbf{A_i(t)} \tag{40}$$

This updated velocity is then added to the current position given in Equation 41.

$$\mathbf{p_i(t+1)} = \mathbf{p_i(t)} + \mathbf{V_i(t+1)} \tag{41}$$

With these laws, inspired by Newton's laws of motion and gravitation, the foundation of the GSA has been completed. This algorithm was then compared to two different heuristic algorithms, the PSO (Section 3.1) and the Central Force Optimizer (CFO). For the PSO algorithm, the observed differences between the two algorithms were as followed:

- The whole Swarm is utilized to update the velocity of a single particle in the GSA, while the PSO only used p_{best} and g_{best}.
- Since every particle is utilized, a single particle can "see" the optimization plane around themselves
- The PSO algorithm does not consider the distance between two particles.

On the other hand, the CFO algorithm is extremely similar to the GSA algorithm with a few differences. First, the CFO is inherently deterministic and does not use any random uniform scalar products in its computations. They then change the universal gravitational to be a scalar constant over time. Finally, the acceleration calculation is dependent proportionally on the fitness function. This however might cause the acceleration to be very high and the particles could go outside the search space.

Any particle that "flew" out of the domain was then moved to the midpoint between its past position and the minimum or maximum value of the coordinate lying outside the allowable range.

The authors [2] applied the GSA to different minimization functions and compared the results with a Real Genetic Algorithm (RGA) as well as the PSO. The population size of a Swarm was initialized to 50 ($m = 50$). Dimension was 30 ($n = 30$) and maximum iteration was $k = 1000$. The results given from the GSA, in most cases, provide better minimization solutions and in all cases are comparable with PSO, RGA, and CFO.

Heuristic Swarm optimization techniques are easy to implement and have shown to produce good results when applied to GANs [14]. Since the PSO and GSA algorithms have each particle working together rather than competing, it would be interesting to see how they would do for training GANs, in place of the Evolutionary algorithm. The final focus paper [3] does such. This paper combines the three previous GANS (EGAN, DCGAN, and LSGAN) and uses the PSO algorithm instead of the Evolutionary algorithm for training.

4 Application

As stated in Section 2.2, the original GAN architecture was well suited for image generation. Yet, it fell victim to the vanishing/exploding gradient problem and mode collapse. In [3], the authors discussed two ways developers could reduce these risks. They could update the Loss function, or change the training process. By using either of these methods, the network can stabilize and generate more realistic images.

As these GANs become more stable, companies are using them for different applications, such as hyper-realistic faces generation (Figure 9) and painting styles transfers (Figure 10). In [3], the authors implemented a DCGAN (Section 2.2.1) framework. They take this framework and update the objective function to the LSGAN (Section 2.2.2). They then use the PSO algorithm to update the weights of the Generator network, which is similar to the EGAN (Section 2.2.3).

Figure 9: A set of generated faces using the NVIDIA StyleGAN

Figure 10: Transferring different styles onto the Mona Lisa using Twin-GAN

The authors first performed a parameter study of the PSO algorithm. This was done by optimizing a multi-modal equation with a global minimum at the origin (Equation 42).

$$f(x,y) = \frac{\sin\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}} + \exp\left(\frac{\cos\left(2\pi x\right)+\sin\left(2\pi y\right)}{2}\right) - 2.71289 \tag{42}$$

The weighting functions updated the inertia at each iteration. Initially, the output of the weighting function was high and decreased over time. By doing so, the particles first search the region, and as the weight became smaller, they would then begin to converge.

After finding an optimal weight, Equation 43, they updated the evaluation function of a single particle. Similar to the EGAN, each particle represented an independent Generator. The particle's position represented the trainable weight of the convolution kernel. In paper [14], the authors defined a cost function for a particle that balanced its accuracy and its diversity (Equation 20).

$$w(k) = w_{start} - (w_{start} - w_{end})(\frac{k}{T_{max}})^2 \tag{43}$$

Over time, all the particles should theoretically converge. If this happened, they would all have similar output and be computationally expensive. To reduce this computational cost while maintaining a set of diverse particles the authors of [3] re-initialized half of the particles when they start to converge. The authors initialize eight particles with a normal distribution across the optimization plane with a mean of zero and variance of 0.2.

The authors then update the loss function. Rather than the original GANs objective function (Equation 9) which was susceptible to exploding/vanishing gradients, the authors used the Least Squared objective function (Equation 15) for the reasons stated in Section 2.2.2.

This newly designed network was trained twice on different datasets. First, it was trained on data from the Cifar 10 dataset [17], then the Celebrity faces dataset [15]. The Cifar 10 dataset is a set of 60000 images with each image having 32x32 pixels in RGB format. These images are separated into an equal distribution of 10 unique classes. The Celebrity face dataset consists of more than 200,000 celebrity images (shown in Figure 11).

Figure 11: Example training data from the celebrity face dataset

Once the training was completed, the output of the most "optimal" Generator (show in Figure 15) is compared to the generated images of the other networks using the Fréchet inception distance (FID) scoring metric [18].

The Fréchet distance is a metric that is used to measure the minimum distance between distributions. The univariate Fréchet metric is given in Equation 44 where μ is the Mean, and σ is the standard deviation of two different datasets X, Y.

$$d(X,Y) = (\mu_X - \mu_Y)^2 + (\sigma_X - \sigma_Y)^2 \tag{44}$$

This multivariate Fréchet distance in Equation 45 has Σ representing the covariant matrix of its respective data. Also, Tr is the *trace* of the matrix or the sum of the diagonal elements in a matrix.

$$d(X, Y) = ||\mu_X - \mu_Y||^2 + Tr(\Sigma_X + \Sigma_Y - 2\sqrt{\Sigma_X \Sigma_Y}) \tag{45}$$

This multivariate FID score is used to evaluate the quality of images generated by GANs. We can generate the X and Y distributions based on the real and fake images in the training set. These distributions come from passing the real and fake images through the Inception v3 model. This pre-trained model has a 98 percent accuracy [19]. The lower scores are shown to correlate well with higher-quality images. These scores estimate the quality of a set of generated images based on the top-performing image classification model. The goal in developing the FID score was to evaluate generated images based on the statistics of a collection of generated images compared to the statistics of a collection of real images from the training domain.

The generated output for the EGAN (Figure 14) scored the best (36.1), then followed by the output of the newly developed PSO GAN (Figure 15) (36.7). This was then followed by the LSGAN (Figure 12) (42.3) and the DCGAN performing the worse at 58.4 (Figure 13).

Figure 12: Generated output from the LSGAN paper [5]

Figure 13: Generated output from the DCGAN paper [6]

Figure 14: Generated output from the EGAN paper [4]

Figure 15: Generated output from the PSO-GAN [3]

By comparing the generated output and FID scores of each of these GANs the authors showed that they can improve the stability of training and the quality of generated outputs. However, while the EGAN performed slightly better, its computational cost was higher.

5 Conclusion

In this report, we first introducing AutoEncoders and their ability to compress data into a lower Latent Feature Space. This ability came from the networks Encoding and Decoding layers. From there, we transitioned to the original GAN, and its ability to generate data in the same Latent Feature Space as its training data. This was done by the use of two different Neural Networks that comprised the GAN. While this original GAN was well suited for image generation, it was highly susceptible to the vanishing and exploding gradient problem. The original GAN was also susceptible to mode collapse due to the network's loss function.

To reduce these issues three different GANs were introduced. The LSGAN, the EGAN, and the DCGAN. The LSGAN changed the loss function of the GAN. This new loss function assigned an error to correctly labeled images. Thus, no matter how the Generator performed, it would be "pulled" towards the Latent Feature Space learned by the Discriminator.

Unlike the LSGAN, the DCGAN opted to change the architecture of the GAN, rather than its loss function. To reduce the number of trainable weights and offer a better association of features for an input, the DCGAN added convolutional layers into the architecture. These convolutional layers were then connected such that only the "strongest" features were propagated forward.

The EGAN updated the training process by using the Genetic Algorithm, which is a type of heuristic swarm optimizer. It first initialized multiple Generators and assigned each Generator a score based on its accuracy and diversity. The accuracy was used to get good results, while the diversity was used to minimize mode collapse. From there each Generator had "children" through asexual reproduction. The children were similar to their parents except they had a slightly different set of weights. Once these new Generators were assigned a score the most optimal ones move forward, while the others were "killed off".

Two different meta-heuristic algorithms of interest were the PSO and GSA. These two algorithms used nature-inspired algorithms that are used to optimize a given objective function. However, unlike first and second-order optimization techniques, these algorithms did not use a gradient or hessian. Rather they initialized random particles across the optimization plane. Based on how well each particle performed, they would update their location. While this reduced the computational cost of calculating derivatives, it did not give an exact solution.

We concluded this report with an application of a GAN called the PSO-GAN. This GAN was an accumulation of the three previously discussed GANs but uses the PSO algorithm to update its kernels. The author in [3] first performed a parameter study of the PSO using different weighting metrics and objective functions. From there they ran the PSO-GAN with eight particles. It was shown that this Neural Network performed similarly to the EGAN and better than both the LSGAN and the DCGAN.

This paper shows that by perturbing different aspects of a Neural Network, developers can increase the accuracy, and minimize its associated issues. However, with the previously discussed disadvantages of the PSO algorithm, I think it would be interesting to perform a similar study but with the GSA.

References

[1] M. Juneja and S. Nagar, "Particle swarm optimization algorithm and its parameters: A review," in *2016 International Conference on Control, Computing, Communication and Materials (ICCCCM)*, pp. 1–5, IEEE, 2016.

[2] E. Rashedi, H. Nezamabadi-Pour, and S. Saryazdi, "Gsa: a gravitational search algorithm," *Information sciences*, vol. 179, no. 13, pp. 2232–2248, 2009.

[3] L. Zhang and L. Zhao, "High-quality face image generation using particle swarm optimization-based generative adversarial networks," *Future Generation Computer Systems*, vol. 122, pp. 98–104, 2021.

[4] C. Wang, C. Xu, X. Yao, and D. Tao, "Evolutionary generative adversarial networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 6, pp. 921–934, 2019.

[5] X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. Paul Smolley, "Least squares generative adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 2794–2802, 2017.

[6] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[7] M. Dorigo and G. Di Caro, "Ant colony optimization: a new meta-heuristic," in *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, vol. 2, pp. 1470–1477, IEEE, 1999.

[8] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.

[9] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[10] X.-S. Yang, "Engineering optimizations via nature-inspired virtual bee algorithms," in *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pp. 317–323, Springer, 2005.

[11] L. Wang, S.-R. Lu, and J. Wen, "Recent advances on neuromorphic systems using phase-change materials," *Nanoscale research letters*, vol. 12, no. 1, pp. 1–22, 2017.

[12] D. Birla, "Autoencoders," Mar 2019.

[13] V. Nagarajan and J. Z. Kolter, "Gradient descent gan optimization is locally stable," *arXiv preprint arXiv:1706.04156*, 2017.

[14] Y. Fu, M. Gong, G. Yang, H. Wei, and J. Zhou, "Evolutionary gan-based data augmentation for cardiac magnetic resonance image," *CMC-COMPUTERS MATERIALS & CONTINUA*, vol. 68, no. 1, pp. 1359–1374, 2021.

[15] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[16] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, IEEE, 1998.

[17] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[18] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," *Advances in neural information processing systems*, vol. 30, 2017.

[19] A. E. Tio, "Face shape classification using inception v3," *arXiv preprint arXiv:1911.07916*, 2019.