Evolutionary Based Training of Generative Adversarial Networks

Evgeniia Martynova - s1038931
Lili Mészáros - s1015790
Nienke Wessel - s4598350

***Abstract*— In recent years, interest in both evolutionary algorithms and neural networks has been very high. Not surprisingly, there have been efforts in combining these two techniques to make even better neural networks. In this paper, we look into one recently published paper on using evolutionary algorithms to train Generative Adversarial Networks (GANs). We have implemented their algorithm and replicated most of their findings. This suggests that an evolutionary GAN indeed performs better, but is also significantly slower than other possible GANs.**

I. Introduction

In recent years, much research interest has been dedicated to both evolutionary algorithms and neural networks. While these two techniques might not have much overlap at face value, more recent research has shown that the two are very well combinable to make even stronger, better trainable neural networks. One of those recent endeavors is the paper of Wang *et al.* [1], where the authors propose applying an evolutionary training procedure to the training of a Generative Adversarial Network (GAN).

Our goal in this paper is to implement their algorithm and see whether we can replicate their results. We elaborated on their results by doing a more extensive analysis on a simulated dataset.

In this report, we will first review the necessary background knowledge on both GANs and evolutionary algorithms in the following section. Then we will discuss the proposed algorithm of Wang *et al.* [1] and related work. After that, we will go over our implementation choices and the reasoning behind them. Then we will discuss our results on several data sets with our implementation. We close off with a discussion and an outlook on the future.

II. Background materials

In this section, we will briefly discuss the theory behind GANs and Evolutionary Algorithms.

A. GANs

A Generative Adversarial Network, usually abbreviated to GAN, actually consists of two neural networks. They are used to generate fake data that should be as indistinguishable as possible from a training set of real data. Usually this data is a set of images, but one can also try to replicate other data.

The first neural network is called the *generator* and its task is to generate the fake data. The second neural network is called the *discriminator* and its task is to determine of a data point whether it is real (from the training set) or fake (generated by the generator). During training, the discriminator is trained to be as accurate as possible at distinguishing the real and fake data, while the generator uses the output of the discriminator (i.e. whether the discriminator managed to figure out a generated pictures was fake) to generate more plausible data. In this way, it becomes a sort of game between the generator and discriminator to fool the other. That also means we can provide a game theoretic interpretation of the training procedure. This interpretation will be explained later on in the section pertaining to the used algorithm.

Manisha & Gujar [2] provide an overview of the current state-of-the-art of GANs. Recent real life applications of GANs include restoring pictures [3], generating particle showers for the Large Hadron Collider of CERN [4] and improving the automatic classification of pearls [5].

B. Evolutionary algorithms

Evolutionary algorithms, which came about in the 80s of the previous century [6], have been used ever since for mostly optimization problems. The general idea of a genetic algorithm is that there is some gigantic generation of individuals, and we want to find the best individual. A common example is the collection of bit strings of a certain length, where we want to find the bit string of only 1s. Besides this collection, one also needs a fitness function to determine how 'good' each individual is. In our bit string example, a fitness function could be to count the number of ones and divide it by the length of the strings.

An iteration of the evolutionary algorithm would then be as follows. From the current individuals (which we call the *parents*), we create one or more new individuals by changing the parents genetically and randomly. It could be possible to combine several parents or use one parent and randomly change it a bit. In our bit example this would be flipping some bits applying some bitwise operation to two or multiple bit strings. We call these newly generated individuals the *offspring*. Next, we use our fitness function to compare the offspring and parents to see which ones are best. In our example case this gives us the bit strings that have the most ones. We select the best ones and those become the new parents. It is possible for an old parent to also be selected as the new parent.

Generally, this is repeated until a certain amount of iterations has passed or some convergence criterion is reached.

The difficulty in using evolutionary algorithms lies in finding the right fitness function. In general, it is assumed

that a function needs to be somewhat smooth in order for the algorithm to reach an optimum.

III. Algorithm proposed by Wang *et al.*

The idea proposed in Wang *et al.* [1] is that we can use an evolutionary algorithm to improve GANs training. A major issue in current GAN training is mode collapse. This means that the generator assigns most of its probability mass to a small space, while it should be distributed over more space. Wang *et al.* [1] propose that evolutionary learning with a suitable fitness function can prevent this issue from happening. In their algorithm, the collection of offspring is the generator networks, and the mutations are different loss functions. The fitness function evaluates the quality of a sample from each generator based on the discriminator's output.

A. Mutations

As mentioned in Wang *et al.* [1], objective functions (the loss function) most commonly used for GAN training does not perform equally well for when the discriminator performance changes during training. This of course tends to happen quite a lot during the training of a GAN. Therefore, it is desirable to use the loss function which deals with the current state of discriminator best at each GAN training step. The authors of the original paper proposed to use these objective functions as mutations in GAN evolution. The offspring consists of the generator networks that are obtained by training a parent generator with different objective functions. It then adjust its network weights according to this performance.

There are several possible objective functions, such as the minmax, heuristic or least squares functions. Their formulas can be found in VII-A.

We want to express different objective functions of generator through standard loss functions available in the libraries. Minmax and heuristic can be expressed through binary cross-entropy, and least squares through MSE. The detailed discussion of how objective functions used as E-GAN mutations should be expressed is given in the appendix VII-A.

B. Fitness function

We want the generator to produce high quality and diverse samples. High quality means that the sample successfully deceives discriminator, and diverse samples prevent the discriminator from taking easy countermeasures which might result in a mode collapse. The fitness function proposed by the paper is the following:

$$F = F_q + \gamma F_d$$

where

$$F_q = \mathbb{E}_z[D(G(z))]$$

is the quality score and

$$F_d = -log \parallel \bigtriangledown_D - \mathbb{E}_x[logD(x)] - \mathbb{E}_z[log(1 - D(G(z)))] \parallel$$

is the diversity score.

The detailed derivation of the formula used to calculate fitness function can be found in the appendix VII-B.

Although the purpose of γ hyper-parameter is clear in the original paper, it does not state which value was used and lacks any discussion about its choice. Moreover, we found that two available open-source implementations use very different values: $\gamma = 0.05$[1] and $\gamma = 1.0$[2]. Meanwhile, the proper γ choice seems important because $F_q \in [0, 1]$ by definition, whereas F_q can have substantially wider range of values. So, if we simply use $\gamma = 1$ the diversity score is likely to dominate the quality score and as a result the performance might be hampered. Indeed, while debugging the fitness function, we found that $|F_q|$ is generally a few times greater than 1. Therefore, we decided to tune this hyper-parameter on simulated datasets as described in the IV-A.1 sections.

C. Related work

Besides Wang *et al.* [1], there has not been other work on using evolutionary computing in training GANs. However, using evolutionary computing in training neural networks in general has become a popular research topic in the last twenty something years.

There have also been successful applications of genetic algorithm based neural networks to real life problem, such as colon cancer prediction [7], prediction of binding of peptites [8], and, more recently, oil price prediction [9]. An interesting overview of how evolutionary techniques can be used in making neural networks in general can be found in Stanley *et al.* [10].

IV. Experiments

In our experiments, we ran the E-GAN both on simulated datasets, out of which we extrapolated the best hyperparameter settings, and we ran it on two benchmark image datasets. The next section deals with the simulated dataset and the hyperparameter finding, while the second section is about the image datasets. The final subsection denotes some experimental setup details.

A. Simulated data

At first, we ran the experiments on two low-dimensional synthetic mixture of Gaussians datasets. The first one consists of eight 2D isotropic Gaussians with standard deviation $\sigma = 0.02$ and means evenly spaced in a circle of radius 2. The other one is composed of twenty five 2D isotropic Gaussians with standard deviation $\sigma = 0.05$ and means arranged on the grid $\mu \in \{-2, -1, 0, 1, 2\}$.

These datasets have become popular benchmarks in the literature (e.g. [11], [12], [13]) because, by design, their probability distribution exhibits quite a few modes with a high probability mass separated by large regions of very low probability mass. This property makes the mixture of Gaussians datasets challenging despite being 2D. Besides,

[1] https://github.com/WANG-Chaoyue/EvolutionaryGAN-pytorch

[2] https://github.com/WANG-Chaoyue/EvolutionaryGAN

since the true distribution and modes of synthetic datasets are known, mode collapse and GAN performance can be accurately measured.

1) Experimental setup: As in Wang *et al.* [1], the E-GAN is compared to other GANs using the different objective functions which are used as mutations in the evolutionary framework: GAN-Minmax, GAN-Heuristic and GAN-Least-Squares. We also compared it to the Wasserstain GAN (WGAN) [14], which was designed to improve the stability of learning and overcome the mode collapse issue, and thus provides an interesting point of comparison.

For a proper comparison with the original paper's results, we aimed at an accurate replication of its experimental setup. However, we found that the generator and discriminator architectures described in Metz *et al.* [15], which the authors refer to, are different from the ones used in their own open-source implementation[3]. Eventually, we adopted the architectures from Turner *et al.* [13] because we found that they are close to this open-source implementation and their implementations makes it reasonably hard for the generator to learn the target distribution, thus giving a fairer comparison of GANs. The detailed description of toy generator and discriminator as well as used optimisers can be found in appendix VII-C.

Following the original paper, the GANs were trained for 50K iterations by alternating updates of the discriminator and the generator. For the convenience of the training progress monitoring, we split the iterations into 100 epochs, 500 steps in each. The batch size for each iteration is 64. The noise for both training and evaluation is sampled from a 2D uniform distribution over the $[-0.5, 0.5]$ interval.

To investigate the impact of γ and choose its value for the experiments on the image datasets, we trained the E-GAN on both simulated datasets with 21 different values of γ evenly spaced in the interval $[0, 1]$. The best γ was selected based on the evaluation metrics described in the next section.

2) Evaluation: The evaluation of the different GANs in the original paper is rather limited. The authors only made a visual comparison of the kernel density estimations (KDE). This makes it hard to reliably conclude which model is better when the differences are not so apparent from KDEs. Also, since we decided to tune γ and expected the result to be quite similar for adjacent values, we needed a more robust evaluation of generator performance.

The choice of good evaluation metrics for GANs is non-trivial. The most recent and comprehensive analysis of various proposed evaluation metrics for GANs that we found, is given in the paper by Borji [16]. The *qualitative measures* section describes *high quality samples rate* and *number of modes captured* metrics suitable for mixture of Gaussians datasets, which apparently were first used by Dumoulin *et al.* [11] and Srivastava *et al.* [12]. We employed an improved version of these metrics recently proposed in Turner *et al.* [13]:

- *High quality samples rate*. A sample is considered "high quality" and assigned to a mode if its L_2 distance to any mixture component is within four standard deviations of this component mean. Since for both simulated distributions the distance between components is such that their 4σ vicinities do not overlap, each sample can be assigned only to one mode.
- *Jensen-Shannon divergence (JSD)* between the sample mode distribution and uniform distribution. The motivation of this metric is that we want a perfect generator to produce samples uniformly distributed over all real modes. This value was proposed as a more stringent test of the spread of probability mass than the number of recovered modes (those to which at least one sample was assigned)
- *Standard deviation* of samples within each mode. We calculate it separately for x and y components.

The last metric we used to evaluate how well the generator had learned the distribution is the *average log-likelihood*. It measures the likelihood of the real data sample under the generated distribution approximated with KDE: $L = \frac{1}{N}\sum_{i=1}^{N} P_{gen}(x_i)$, where x_i belongs to real data sample, N is a real sample size, and P_{gen} is an estimate of the distribution learned by generator obtained from a generated sample. We decided to use it in spite of the known deficiencies mentioned in Borji [16]. The issue with KDE estimation for high-dimensional data is not applicable to 2d mixtures of Gaussians and it is an obvious way to compare two 2D distributions, which does not add a significant computational overhead.

For training progress and convergence monitoring, we sampled 10000 points from both the fixed noise and real distribution before training start. The described metrics were applied at the end of each epoch. A fixed noise sample was used to obtain a generator's fake sample and to evaluate it with selected qualitative measures as well as for KDE estimation for average log-likelihood calculation. A real sample is used to compute the average log-likelihood.

Additionally, to get more insight into the training progress, we saved a scatter plot of the generator sample obtained from the fixed noise at the end of each epoch. Finally, at the end of the GAN training, we saved the final generator's distribution KDE for a visual comparison.

To prevent differences in weights initialisation affect models performance, we use the same seed for each run.

3) Implementation of simulated distributions KDE: To obtain a KDE, we used the `KernelDensity` class from the `sklearn` library because it allows tuning the bandwidth parameter with cross-validation. A proper bandwidth value is essential for accurate density estimate. Interestingly, we found that for both datasets an optimal bandwidth returned by cross-validation equals half of the Gaussian components' standard deviation. Average log-likelihood is calculated based on the KDE obtained with optimal bandwidth. However, since the used standard deviations are very small, especially for the 8 Gaussians in a circle, for saving of KDE images we used bandwidths equal to 3 standard deviations to enlarge the learned modes and make visual comparison

[3] `https://github.com/WANG-Chaoyue/EvolutionaryGAN`

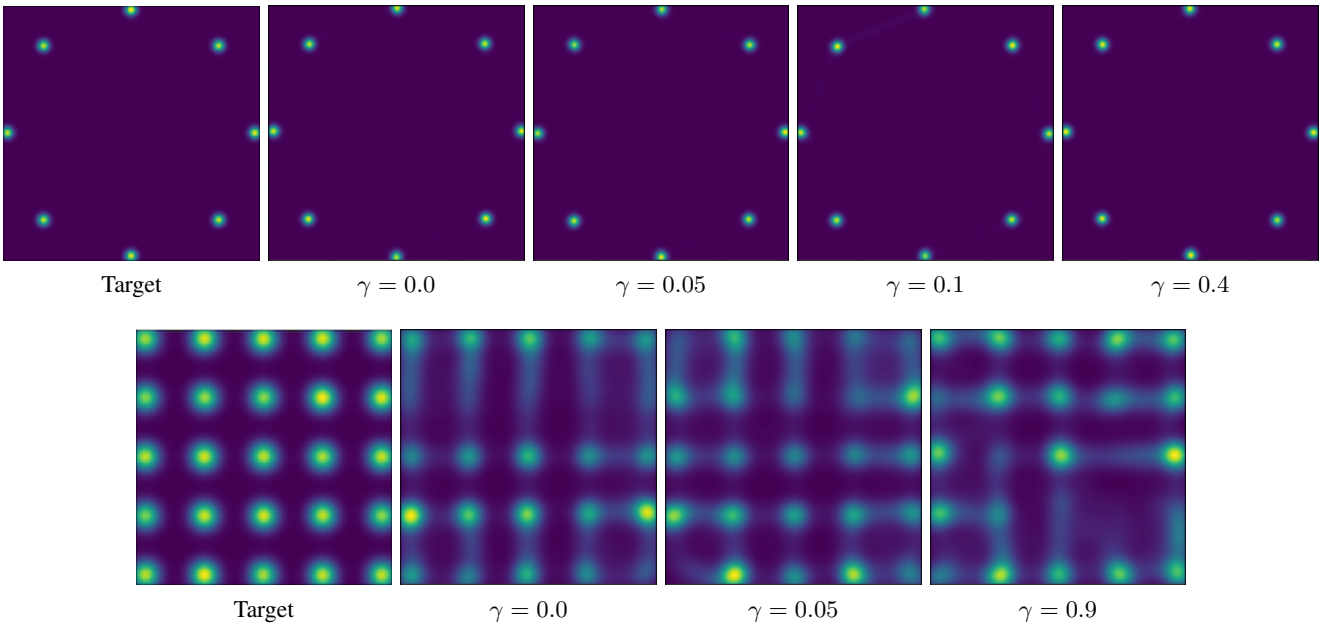


Fig. 1: The best KDEs obtained from sample of size 10000 by E-GANs with different γ for mixture of 8 Gaussians (top row) and 25 Gaussians (bottom row). Target distributions KDEs are given in the first column.

possible.

B. Image datasets

In order to compare performance, we used two image datasets; MNIST to have more simple data with less variety and grayscale images, and CelebA [17] exhibiting large variety and RGB images. PyTorch has these data sets integrated in its framework already, so no changes were made to them other than normalising the images.

The setup for the image datasets is largely the same as for the simulated dataset, except for that we compare the results only visually. The evaluation metrics commonly used for GANs trained on image data are more complex and time-consuming to integrate, and we did not manage to fit this into the given timeframe.

A major difference is that for the Celeb dataset, we only let the GANs run for 10 epochs, as running time was significantly higher for that dataset due to its size.

Another difference is that we did not look into different objective functions, but instead only used the minmax loss function for the E-GAN. Therefore, our comparison consists of the E-GAN, WGAN and GAN (where the latter uses the same network as the E-GAN, but does not use evolutionary updates).

The networks used for both datasets were linear layer networks with dropout and ReLU activation functions. The details of the networks can be found on our Github page [4]. Note that the original paper used DCGAN for their image datasets. We were not able to do so due to time and computational resource limitations.

[4]https://github.com/meszlili96/NaturalComputing/tree/master/Project

C. Experimental setup

Data Set	Epochs	Batch Size	Dataset length
Gaussians	100	64	32000
MNIST	100	64	60000
CelebA	10	64 or 128	202599

TABLE I: GANs training setup for different datasets.

We wrote all our code in Python, and used the Pytorch library for the implementation of the necessary neural networks. Our code can be found on our Github page[5]. In table I one can see more GANs training setup details. For the CelebA dataset, the GAN and WGAN had 128 batch size and the E-GAN 64 batch size. Due to time limitations, it was possible to run the E-GAN again with the 128 batch size.

V. Results

A. γ tuning and analysis

We start by looking for the best value of γ to use in further experiments. To narrow down the options, we first compare the obtained KDEs visually. For the 8 Gaussians in circle dataset, γ values 0.0, 0.05, 0.1, 0.4 produced the best KDEs. They are given in the top row of figure 1. For the 25 Gaussians in a grid, γ values 0.0, 0.05, 0.9 gave the most convincing KDEs. They are depicted at the bottom row of figure 1. The results for all tried γ values can be found in Google Drive folder [6].

[5]https://github.com/meszlili96/NaturalComputing/tree/master/Project

[6]https://drive.google.com/drive/folders/1vC_HgUiDnGbGSMAQLpbppKlwZuioXlJP?usp=sharing

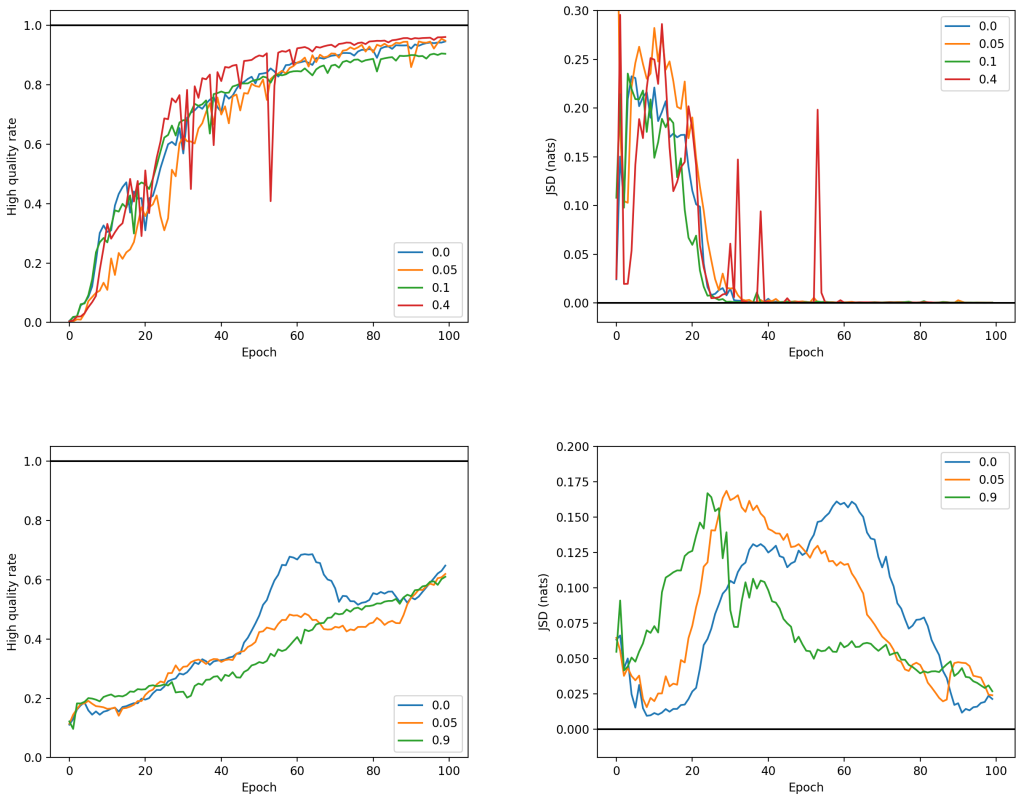


Fig. 2: High quality samples rate and JSD for E-GAN with different γ for mixture of 8 Gaussians (top) and mixture of 25 Gaussians (bottom). The target values are denoted by the black horizontal lines.

It can be seen that, visually, it is hardly possible to spot differences in KDEs for the 8 Gaussians dataset; the results of all γ values seem equally good. For the 25 Gaussians, the differences are more apparent. However, it is difficult to conclude which generator's distribution is the closest to the target.

Then, for both datasets, we compared the evaluation metrics to get more insight into GANs performances and convergences. We found that the Jensen-Shannon divergence and the rate of high quality samples were the most informative metrics for GAN comparison. The plots of these metrics are given in figure 2 for both 8 and 25 Gaussians datasets.

Standard deviation plots provide useful information as well, but the plots had too many oscillations for each γ, making it difficult to analyse when everything is given in one figure. Hence, we made a separate plot for each γ and put it in the appendix (figures 14 and 13). Consistently with [16], the average data log-likelihood metric did not help determine which generator performance is better. It quickly reaches similarly high value for all γ values and only rarely fluctuates. Nevertheless, we included its plot for illustration in the Appendix, figure 15.

In all the plots (especially for the 8 Gaussians dataset), we can see that effect of giving more weight to the diversity score matches its purpose. In the top row of figure 2, it can be noticed that the higher γ is, the more oscillations the high quality rate metric demonstrates during training. In terms of JSD, the convergence is good for all four values. However, $\gamma = 0.4$ produces three sharp spikes after the other γ values converged to 0, showing no similar spikes. With a further increase of γ, the number and amplitude of oscillations and spikes grow. It demonstrates that the generator tends to explore the target distribution space more and is more likely to produce samples that deviate from the learnt modes when in the fitness function more weight is given to the diversity score. From all the results, high γ values seem undesirable because training becomes unstable and performance decreases.

For the 25 Gaussians dataset (the bottom row of figure 2), the changes of metrics are much smoother and the performance is noticeably worse. It is interesting to have a closer look at the JSD plot. At the beginning of training, its value is already very low, then it starts to grow and decreases again by the end of the training loop. This happens because, in the Mixture of 25 Gaussians dataset, the modes are evenly spaced in a grid and in a first few epochs, the generator tends to produce samples evenly distributed in [-2, 2] interval. [7]

[7]Check figure 12 or the content of Google Drive folder linked above to see how the scatter plot of a sample generated from a fixed noise is changing by epoch.

Then, while it is learning the distribution, it discovers some modes faster than others, which causes a decrease in JSD. Finally, when a generator has learnt the distribution well, the JSD decreases to a value close to 0 again.

To simplify the comparison of γ even more, we looked at the final high quality rate value and the statistics of the x standard deviation in last 20 epochs, because we expect the best generator to reach steady convergence by the end of the training. For x standard deviation statistics the mean which is the closest to the true value and the smallest standard deviation are considered the best. These values are given in table II for the 8 Gaussians dataset and in table III for the 25 Gaussians dataset. Even though there is no single γ which clearly performs best for all the metrics, based on all the results it is sensible to choose $\gamma = 0.05$ for our experiments. It demonstrates good convergence in terms of high quality rate, JSD and standard deviation for both datasets. Overall, from the metrics, $\gamma = 0.0$ performs slightly better, but for a more challenging 25 Gaussians dataset, JSD convergence is better with $\gamma = 0.05$. Since image datasets are at least as complex, $\gamma = 0.05$ should work better in general.

γ	High quality rate	x stdev mean	x stdev stdev
0.0	0.9462	**0.02**	0.0056
0.05	0.9486	0.021	**0.0028**
0.1	0.9046	0.0206	0.0062
0.4	**0.9611**	0.0186	0.0037

TABLE II: High quality samples rate and x standard deviation's statistics in last 20 epochs for different γ for 8 Gaussians dataset.

γ	High quality rate	x stdev mean	x stdev stdev
0.0	**0.648**	**0.0633**	0.0083
0.05	0.6197	0.075	0.0052
0.9	0.6103	0.0789	**0.005**

TABLE III: High quality samples rate and x standard deviation's statistics in last 20 epochs for different γ for 25 Gaussians dataset.

B. Simulated Data

The final KDEs obtained with the different GANs for both datasets can be found in figure 3. As we can see, the difference in the performance is much clearer than it was for the E-GAN γ tuning. The evaluation metrics for the 8 Gaussians dataset are given in 4 and for 25 Gaussians in 5. The same statistics as for γ comparison can be found in tables IV and V.

Consistently with the paper's results, WGAN performance is the poorest (although in the paper it was only applied to the image datasets). Also, if we consider both datasets at once, E-GAN results are consistently better than the results of the GANs with a constant loss function and WGAN. However, the performance of GAN-Minmax, GAN-Least-Squares and GAN-Heuristic is different from what was observed in the original paper. There, GAN-Heuristic produced the worst results for both datasets, whereas for our run on the 8 Gaussians dataset the quality of the distribution learned by its generator was better than produced by E-GAN for all the metrics except x standard deviation (E-GAN's result is closer to the true value with lower standard deviation as it can be seen in table IV). Meanwhile, for GAN-Minmax the paper reports good performance on the 8 Gaussians dataset, but we got the second worst results with it.

On the 25 Gaussians dataset, similarly to paper's results, the quality of the distribution learnt by GAN-Least-Squares is competitive with E-GAN. From JSD and KDE plots we can see that E-GAN still performs slightly better and recovers more modes, even though high quality rate of generated samples is somewhat lower. Interestingly, GAN-Heuristic achieves the best high quality rate, but from its KDE and JSD plots, it is clear that it is due to mode collapse.

Finally, we looked into the statistic of the mutations selected in different stages of the E-GAN training process. For the 25 Gaussians dataset, the statistic is similar to the one presented in the paper in figure 4 (right) for a run on the CIFAR-10 dataset, although the difference in selected mutations share is less pronounced. We can see that the share of least squares mutation remains stable during training. At the beginning of the training, the heuristic mutation is preferred sightly more, but as the training continues the number of selected heuristic mutations is falling. Minmax mutation shows the opposite trend.

For the 8 Gaussians dataset, the picture is different and, for some reason, least squares mutation quickly starts to dominate the other mutations. Interestingly, heuristic mutation, which gave a better result than E-GAN when applied alone, is selected least often. We think that the statistics on the 8 Gaussians dataset is so different because this distribution is more straightforward to learn for a generator than the Mixture of 25 Gaussians or an image dataset. The capacity of the used generator is enough to quickly learn the distribution and produce plausible samples. Therefore, the advantages of different objective functions are less important when E-GAN is trained on this distribution.

To provide even more insight into the comparison of the considered GANs, we show scatter plots of the sample obtained from the fixed noise after 25th, 50th and 100th epoch in figures 11 and 12 in the Appendix.

GAN	High quality rate	x stdev mean	x stdev stdev
WGAN	0.1291	0.038	0.0044
GAN-Minmax	0.0492	0.0342	0.0062
GAN-Least-squares	0.0021	0.0339	0.0097
GAN-Heuristic	**0.9574**	0.0169	0.0038
E-GAN	0.9486	**0.021**	**0.0028**

TABLE IV: High quality samples rate and x stdev statistics in last 20 epochs for different GANs for 8 Gaussians.

C. Images Datasets

1) MNIST: The generated images for the MNIST dataset can be found in figure 7. From away, the images look similar.

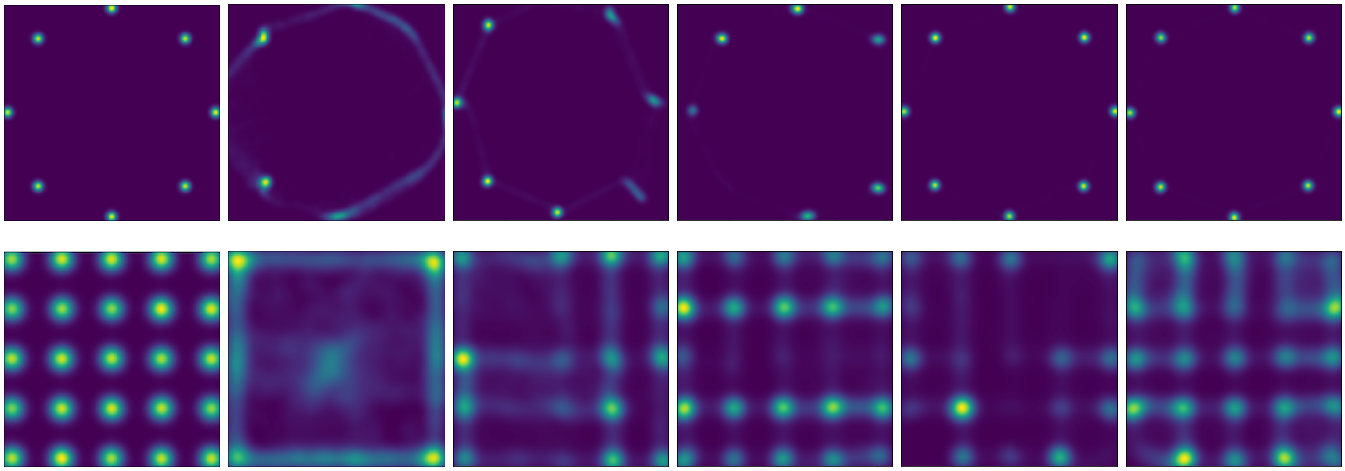


Fig. 3: KDE plots of the data generated by different GANs trained on mixtures of Gaussians. In the first row mixture of 8 Gaussians is shown, in the second - mixture of 25 Gaussians. Target distributions KDEs are given in the first column.

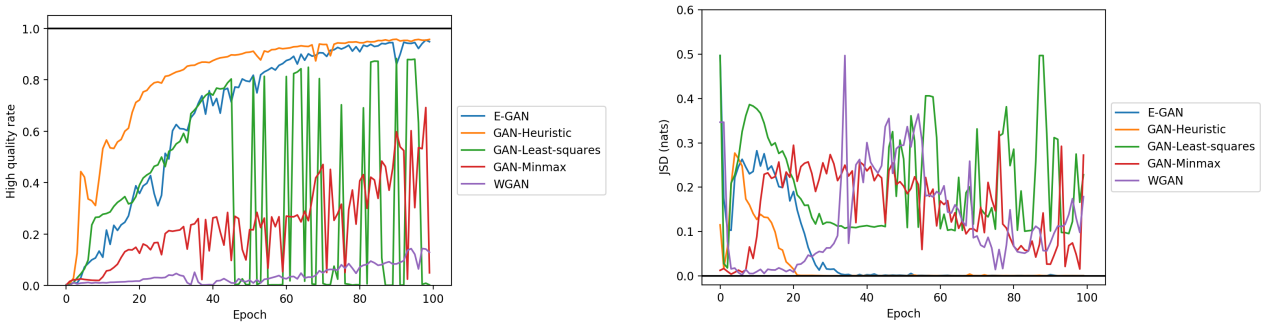


Fig. 4: Evaluation metrics for different GANs for mixture of 8 Gaussians.

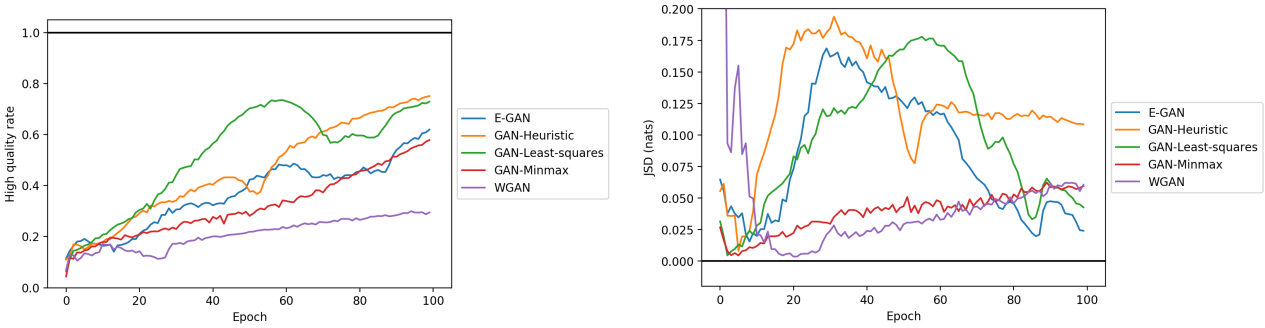
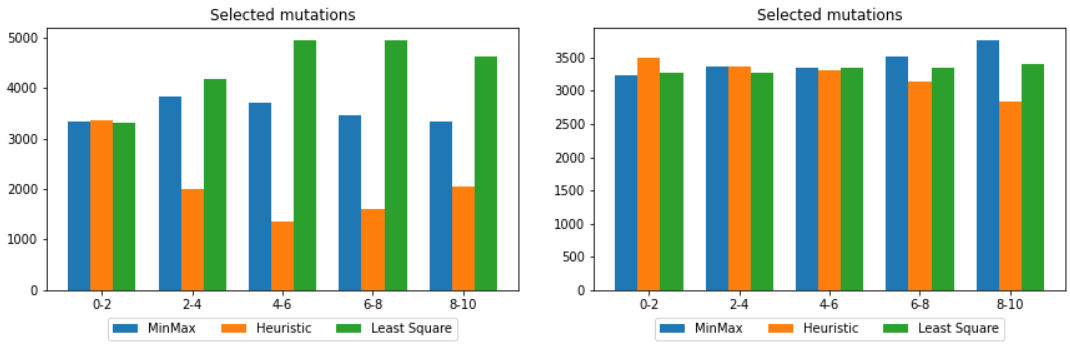


Fig. 5: Evaluation metrics for different GANs for mixture of 25 Gaussians.

Up closer, we see that the GAN results are grittier than the other results (i.e. have more random specs of white). The E-GAN has a bit of these random white specs, while the WGAN has almost none. The E-GAN has some very vague images. It looks like the WGAN outperformed the other GANs in this case, because we can identify more numbers in this set and there are less white specs or vagues ghost-like numbers.

In figure 9 we can see how the losses develop over the epochs. This tells us that all systems converge rather quickly. Only the E-GAN still shows a downwards trend. This might indicate that the E-GAN could still improve more with more iterations. However, for the other two GANs, more iterations would probably not be helpful, as is also corroborated by the



8 Gaussians 25 Gaussians

Fig. 6: The statistics of selected mutations in the E-GAN training process splitted by 10K number of steps.

GAN	High quality rate	x stdev mean	x stdev stdev
WGAN	0.2951	0.0917	**0.001**
GAN-Minmax	0.5786	0.0702	0.0033
GAN-Least-squares	0.729	**0.0627**	0.0043
GAN-Heuristic	**0.751**	0.0694	0.0038
E-GAN	0.6197	0.075	0.0052

TABLE V: High quality samples rate and x stdev statistics in last 20 epochs for different GAN for 25 Gaussians dataset.

small difference between the 25th and 100th epoch in figure 7.

Quite noticeable was the observed running time difference between the different GANs. The simple GAN was the fastest, and the WGAN needed between two and three times as much time. The E-GAN even needed almost seven times as much time as the simple GAN. While our testing setup for running time was rather limited (only one run, that did not use gpu), this does point to significant time differences, which should be taken into account when choosing for one GAN over another.

2) CelebA: The generated images for the CelebA dataset can be found in figure 8. The E-GAN results immediately catch the attention, as those images are really gritty and pixelated. They miss the smoothness that the other two GAN results exhibit. The WGAN clearly outperforms the other two GANs on this dataset.

In figure 10 we can see how the losses develop over the iterations. For the E-GAN we see the generator loss increasing over the epochs, while the discriminator loss stays almost the same. It is not quite clear why this happens, but this is probably related to the E-GAN's poor performance.

VI. Discussion

In this project, we succeeded in replicating most of the results of the paper which introduced E-GAN. We found that the evaluation of GAN performance is challenging and using only one evaluation method is unlikely to be enough for a reliable comparison of different GANs. Our main contribution for simulated data is an improvement of the evaluation metrics for the comparison of E-GAN with other GANs, and the analysis of varying γ E-GAN parameter, which was not done in the original paper. Therefore, we can now state that E-GAN performs better with a higher level of confidence.

The observed results for simulated data were generally consistent with the paper, although the performance of the GANs with the single objective functions deviated from the original significantly. This might be due to the differences in generator and discriminator architectures as well as other implementation details. Another possible reason could be specific weight initialisation, which affects GAN training. The time we had for the project was limited which did not allow us to perform and analyse multiple runs.

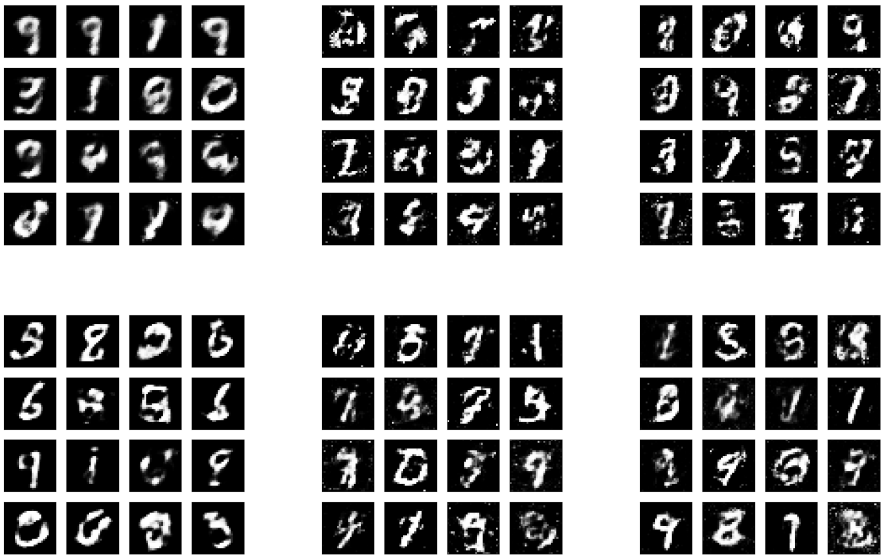
The results for the image datasets were surprising, as the authors implicate that the E-GAN should be superior, but we found WGAN to be superior. We should note that we used a simpler network architecture which might have influenced the results (i.e. the evolutionary part might be more useful in a more complicated network). The authors never clearly compare images generated by the different GANs in a normal situation, though, and only try to show robustness in artificial situations. This makes it hard to determine whether our results are in line with their results, i.e. that WGAN produces better images in a normal situation.

It was quite noticeable that E-GAN did take significantly more time to run than the other GANs. So while it does seem to perform better, this speed-performance trade-off is something that should be kept in mind when using the E-GAN.

A. Future work

One sensible future work direction we see is a further improvement of GAN comparison. Multiple training runs and a collection of statistics should give a more robust evaluation of a GAN's performance, and exclude the possibility of obtaining the best results by chance. A good GAN performance comparison should also pay more attention to running time differences. We only made a quick comparison for the MNIST dataset which seems to point to significant differences, but in order to determine the exact magnitude of these differences, more research is needed.

Another option would be to investigate how E-GAN efficiency can be improved. E-GAN implementation and running



WGAN GAN E-GAN

Fig. 7: Samples of generated images for MNIST at 25 epochs (top row) and 100 epochs (bottom row).

WGAN GAN E-GAN

Fig. 8: Samples of generated images for the CelebA dataset at 9 epochs (top row) and 10 epochs (bottom row).

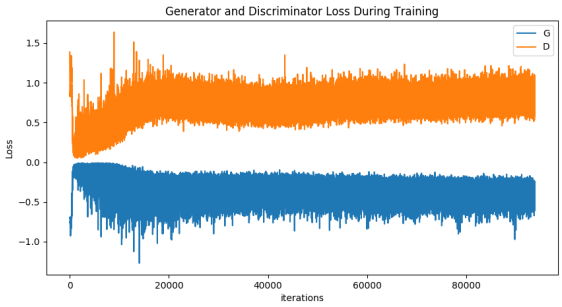
the experiments took a while and we did not have time to pursue this avenue. The authors of the paper say that E-GAN training is already rather efficient, since only one offspring survives after each evolutionary iteration. We thus expect it to be challenging to achieve better efficiency.

We intended to also score the generated images with the inception score [18], but due to time constraints we did not finish this. While the inception score has some serious drawbacks [19], we could not find a better suitable method to compare the quality of the different GANs, other than by just looking at the images.

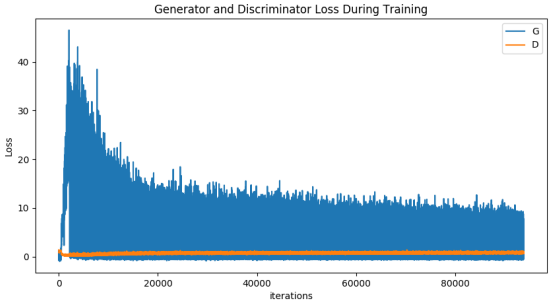
B. Conclusion

While it was not trivial to implement the E-GAN as proposed by Wang *et al.* [1], we managed to replicate most of their results. We made some contributions to simulated data evaluation and discussed its implications. Our image dataset exploration was rather limited, which remains a possible avenue for exploration.

WGAN



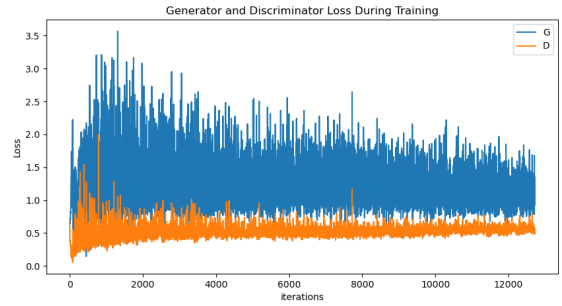
GAN



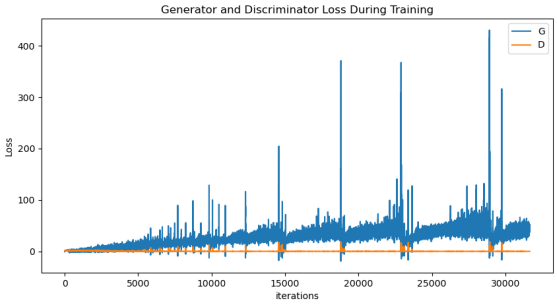
E-GAN

Fig. 9: The losses for the MNIST dataset. Note that the different GANs use different loss functions. The y-axes are therefore not comparable. Note that an iteration is a running one batch size. As the dataset consists of 60000 images and the batch size is 64, one epoch consists of 938 ($60000/64 = 937.5$) iterations. There were 100 epochs, making 93800 iterations.

WGAN



GAN



E-GAN

Fig. 10: The losses for the CelebA dataset. Note that the different GANs use different loss functions. The y-axes are therefore not comparable. The calculation of the amount of iterations is the same as in figure 9. Note that since the batch size differed, the total amount of iterations also differs, but the amount of data used is the same.

REFERENCES

[1] 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.

[2] P. Manisha and S. Gujar, "Generative adversarial networks (gans): What it can generate and what it cannot?" *arXiv preprint arXiv:1804.00140*, 2019.

[3] B. Dolhansky and C. Canton Ferrer, "Eye in-painting with exemplar generative adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7902–7911.

[4] M. Paganini, L. de Oliveira, and B. Nachman, "Accelerating science with generative adversarial networks: an application to 3d particle showers in multilayer calorimeters," *Physical review letters*, vol. 120, no. 4, p. 042003, 2018.

[5] Q. Xuan, Z. Chen, Y. Liu, H. Huang, G. Bao, and D. Zhang, "Multiview generative adversarial network and its application in pearl classification," *IEEE Transactions on Industrial Electronics*, vol. 66, no. 10, pp. 8244–8252, 2018.

[6] J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms," in *Proceedings of the first international conference on genetic algorithms and their applications, 1985*. Lawrence Erlbaum Associates. Inc., Publishers, 1985.

[7] K.-J. Kim and S.-B. Cho, "Prediction of colon cancer using an evolutionary neural network," *Neurocomputing*, vol. 61, pp. 361–379, 2004.

[8] V. Brusic, G. Rudy, G. Honeyman, J. Hammer, and L. Harrison, "Prediction of mhc class ii-binding peptides using an evolutionary algorithm and artificial neural network." *Bioinformatics (Oxford, England)*, vol. 14, no. 2, pp. 121–130, 1998.

[9] H. Chiroma, S. Abdulkareem, and T. Herawan, "Evolutionary neural network model for west texas intermediate crude oil price prediction," *Applied Energy*, vol. 142, pp. 266–273, 2015.

[10] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019.

[11] V. Dumoulin, I. Belghazi, B. Poole, O. Mastropietro, A. Lamb, M. Arjovsky, and A. Courville, "Adversarially learned inference," *arXiv preprint arXiv:1606.00704*, 2016.

[12] A. Srivastava, L. Valkov, C. Russell, M. U. Gutmann, and C. Sutton, "Veegan: Reducing mode collapse in gans using implicit variational learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 3308–3318.

[13] R. Turner, J. Hung, E. Frank, Y. Saatchi, and J. Yosinski, "Metropolis-hastings generative adversarial networks," in *International Conference on Machine Learning*, 2019, pp. 6345–6353.

[14] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv preprint arXiv:1701.07875*, 2017.

[15] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *arXiv preprint arXiv:1611.02163*, 2016.

[16] A. Borji, "Pros and cons of gan evaluation measures," *Computer Vision and Image Understanding*, vol. 179, pp. 41–65, 2019.

[17] 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.

[18] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, 2016, pp. 2234–2242.

[19] S. Barratt and R. Sharma, "A note on the inception score," *arXiv preprint arXiv:1801.01973*, 2018.

VII. APPENDIX

A. The derivation of generator loss functions from objective functions

The objective functions described in section III-A have the following formulas:

1) Minmax:

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

2) Heuristic:

$$\mu_G^{heuristic} = -\frac{1}{2}\,\mathbb{E}_{z\sim p_z}[log(D(G(z)))]$$

3) Least-squares:

$$\mu_G^{least-squares} = \mathbb{E}_{z\sim p_z}[(1 - D(G(z)))^2]$$

To train a GAN with the different objective functions, we first need to derive the generator loss functions from them. Fortunately, all objective functions can be expressed through binary cross-entropy (BCE) or mean squared error (MSE).

BCE loss function:

$$H_p = -\frac{1}{N}\sum_i^N \{t_i \cdot log(p(y_i)) + (1 - t_i) log(1 - p(y_i))\}$$

MSE loss function:

$$MSE = -\frac{1}{N}\sum_i^N (t_i - p(y_i))^2$$

Where $t_i \in 0, 1$ is the true class label, y_i, a sample, and $p(y_i)$ the probability that the sample belongs to class 1. Then we express generator's objective functions through H_p and MSE. We assume that there are N i.i.d. samples from noise z_i, thus $p(z_i) = 1/N$.

Minmax

$$\begin{aligned}
\mu_G^{minmax} &= \frac{1}{2}\,\mathbb{E}_{z\sim p_z}[log(1 - D(G(z)))] \\
&= \frac{1}{2}\sum_{i=1}^N p(z_i) log(1 - D(G(z_i))) \\
&= \frac{1}{2}\cdot\frac{1}{N}\sum_{i=1}^N log(1 - D(G(z_i))) \\
&= |\text{Let labels } t_i = 0, \text{ then we can rewrite it as}| \\
&= \frac{1}{2}\cdot\frac{1}{N}\sum_{i=1}^N (1 - t_i) log(1 - D(G(z_i))) \\
&= -\frac{1}{2} H_p
\end{aligned} \tag{1}$$

Thus, minimizing μ_G^{minmax} equals maximizing the binary cross entropy with all the labels $t_i = 0$. This will make the generator create samples x such that $D(x) \to 1$.

Heuristic (logD trick)

$$\begin{aligned}\mu_G^{heuristic} &= -\frac{1}{2}\,\mathbb{E}_{z\sim p_z}[log(D(G(z)))] \\ &= -\frac{1}{2}\sum_{i=1}^{N} p(z_i) log(D(G(z_i))) \\ &= -\frac{1}{2}\cdot\frac{1}{N}\sum_{i=1}^{N} log(D(G(z_i))) \\ &= |\text{Let labels } t_i = 1, \text{ then we can rewrite it as}| \\ &= \frac{1}{2}\cdot\frac{1}{N}\sum_{i=1}^{N} t_i log(D(G(z_i))) \\ &= \frac{1}{2}H_p\end{aligned} \tag{2}$$

Therefore, minimizing $\mu_G^{heuristic}$ equals minimizing the binary cross entropy with the labels $t_i = 1$.

Least squares

$$\begin{aligned}\mu_G^{least-squares} &= \mathbb{E}_{z\sim p_z}[(1 - D(G(z)))^2] \\ &= \sum_{i=1}^{N} p(z_i)(1 - D(G(z_i)))^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}(1 - D(G(z_i)))^2 \\ &= MSE(D(G(z)), 1)\end{aligned} \tag{3}$$

As we can see, the $\mu_G^{least-squares}$ and MSE loss formulas are identical.

B. The derivation of the fitness score formula

This section demonstrate the derivation of the E-GAN fitness function constituents formulas used in our implementation.

The quality score is defined as:

$$F_q = \mathbb{E}_z[D(G(z))] = \frac{1}{N}\sum_{i=1}^{N} D(G(z_i))$$

And the diversity score as:

$$\begin{aligned}F_d &= -log \parallel \bigtriangledown_D - \mathbb{E}_x[logD(x)] - \mathbb{E}_z[log(1 - D(G(z)))] \parallel \\ &= |\text{Let } t_i = 1,\, t_j = 0| \\ &= -log \parallel \bigtriangledown_D - (\frac{1}{N_x}\sum_{i=1}^{N_x} t_i logD(x_i) \\ &+ \frac{1}{N_z}\sum_{j=1}^{N_z}(1 - t_j) log(1 - D(G(z_j))) \parallel = \\ &- log \parallel \bigtriangledown_D - (\frac{1}{N}\sum_{i=1}^{N}\{t_i logD(x_i) \\ &+ (1 - t_i) log(1 - D(G(z_i)))\} \parallel = -log \parallel \bigtriangledown_D H_p \parallel\end{aligned} \tag{4}$$

Where x_i denotes a real sample, N_x is the number of real samples, z_i denotes a sample from noise, and N_z is the number of generated samples. Thus, the diversity score is the logarithm of the norm of the gradient of the discriminator binary cross-entropy loss.

C. The details of the toy datasets experimental setup

The architectures of the discriminator and generators used for all GANs trained on the simulated data are given in tables VI and VII. Xavier-normal weights initialisation is applied to both networks. For all GANs, except WGAN, the discriminator and generator were trained with the Adam optimiser: parameters $\beta 1 = 0.5$, $\beta 2 = 0.999$ and learning rate $1e-4$. The WGAN was trained with RMSProp with a low learning rate set to 5e-5.

Layer type	Hidden units	x Batch norm	Activation
Fully connected	100	no	ReLU
Fully connected	100	yes	ReLU
Fully connected	100	no	ReLU
Fully connected	1	no	Sigmoid

TABLE VI: The discriminator architecture.

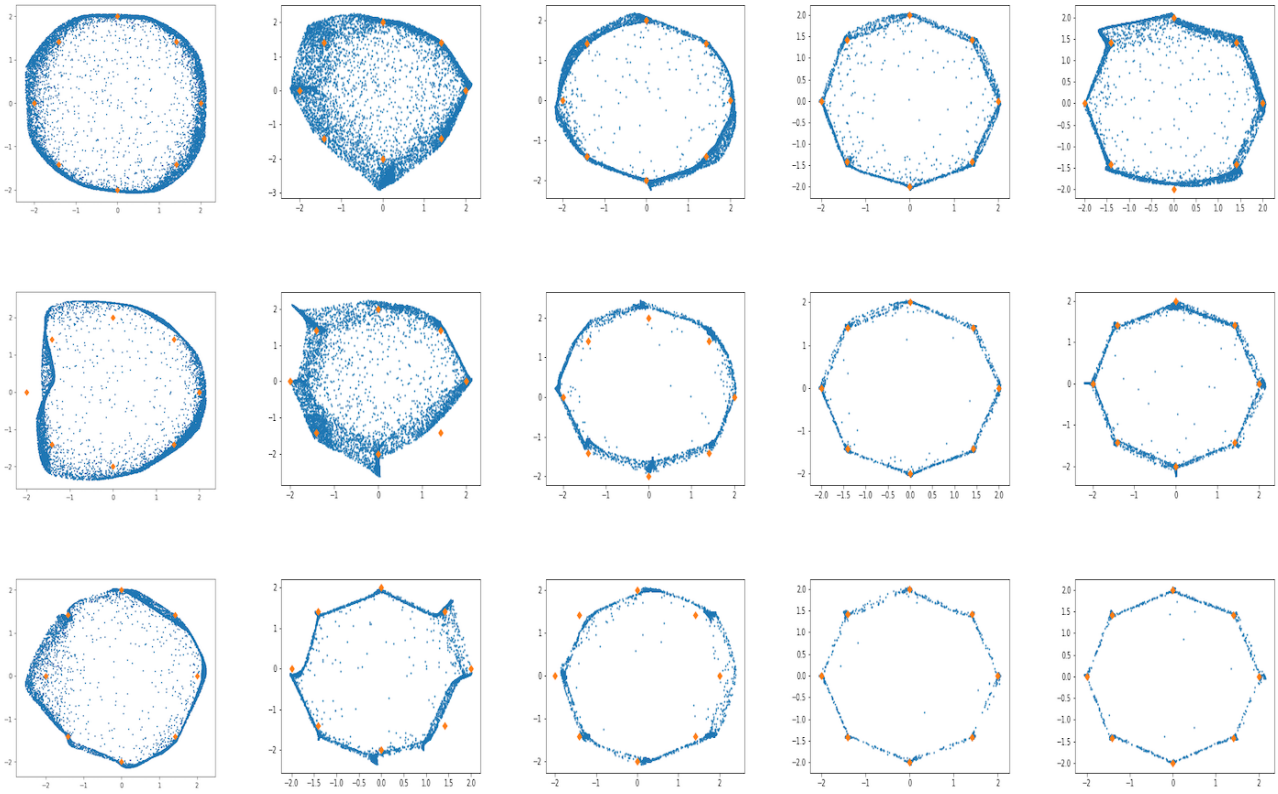
Layer type	Hidden units	x Batch norm	Activation
Fully connected	100	no	ReLU
Fully connected	100	no	ReLU
Fully connected	100	no	ReLU
Fully connected	2	no	Identity

TABLE VII: The generator architecture.

D. Elaborate results

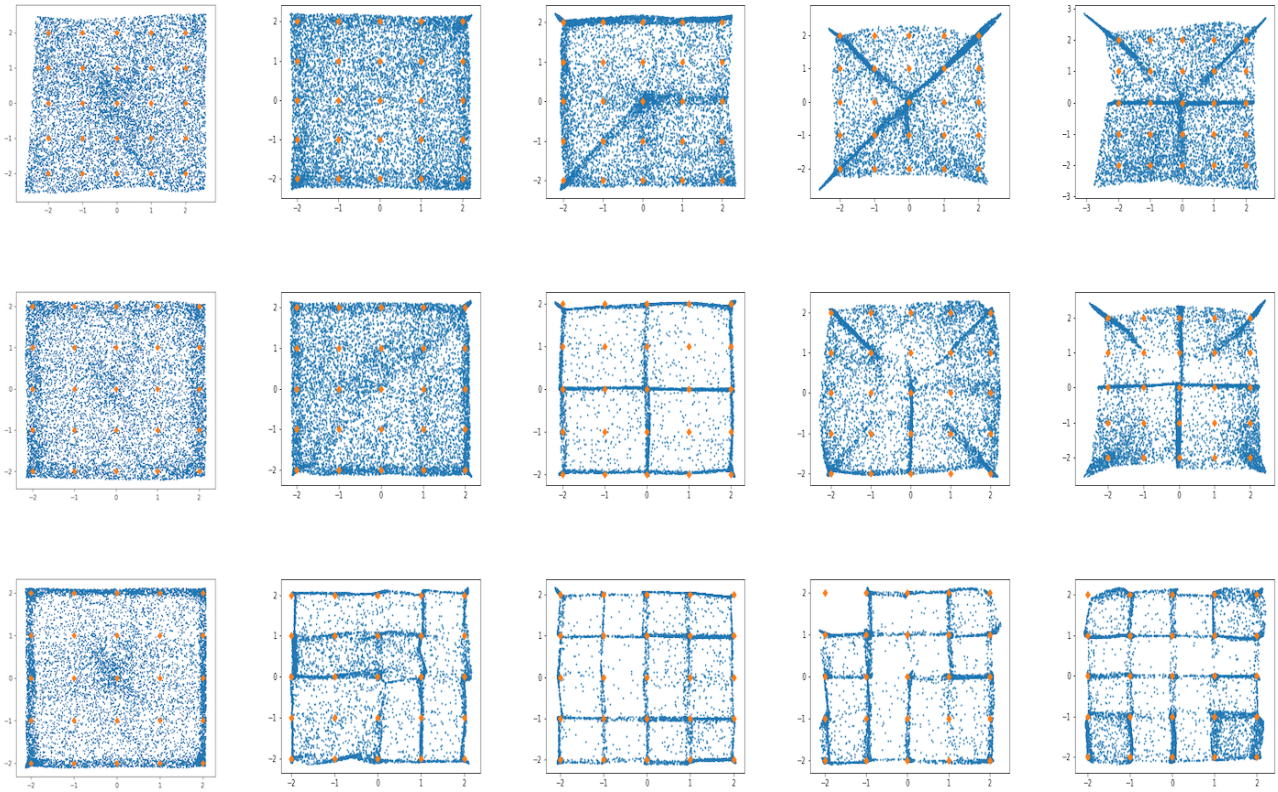
The rest of the appendix contains the more elaborate results of our experiments. We have the following figures:

- Figure 11 and 12 contain the scatter plots of samples produced from the different GANs at different training stages for the 8 and 25 Gaussians dataset respectively.
- Figure 13 and 14 show the different x standard deviations for different γ values on the 8 and 25 Gaussians dataset respectively.
- Figure 15 shows the average log-likelihood forthe E-GAN for different γ's for both Gaussians data sets.



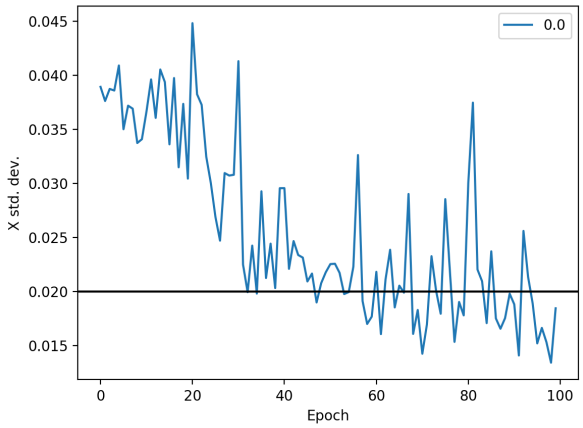
WGAN GAN-Minmax GAN-Leats-Squares GAN-Heuristic E-GAN

Fig. 11: Scatter plots of sample of size 10000 obtained from the fixed noise by generators produced by different GANs at the end of 25 (top), 50 (middle) and 100 (bottom) epochs for the 8 Gaussians dataset. The true modes are denoted with the orange diamonds.

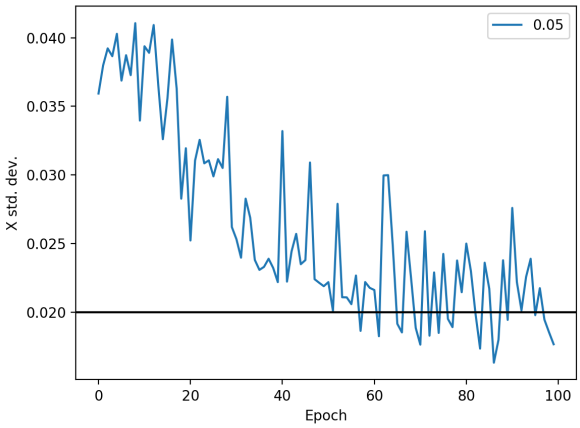


WGAN GAN-Minmax GAN-Leats-Squares GAN-Heuristic E-GAN

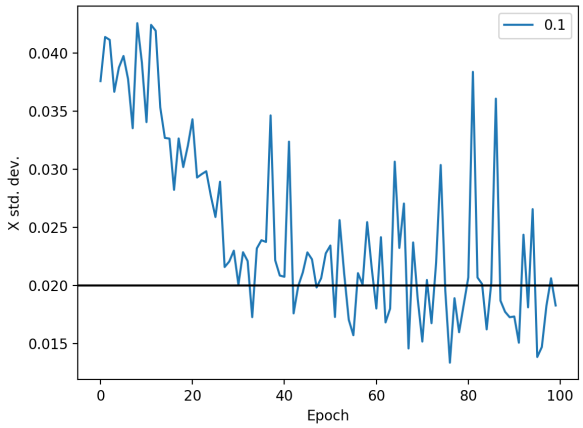
Fig. 12: Scatter plots of sample of size 10000 obtained from the fixed noise by generators produced by different GANs at the end of 25 (top), 50 (middle) and 100 (bottom) epochs for the 25 Gaussians dataset. The true modes are denoted with the orange diamonds.



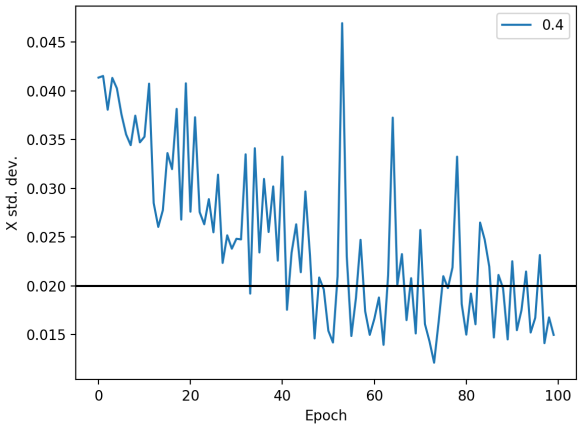
$\gamma = 0.0$



$\gamma = 0.05$

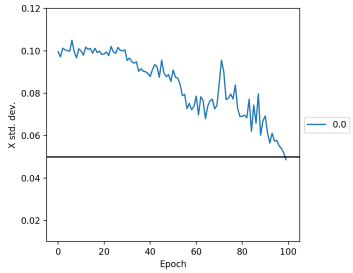


$\gamma = 0.1$

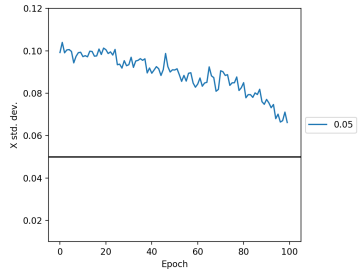


$\gamma = 0.4$

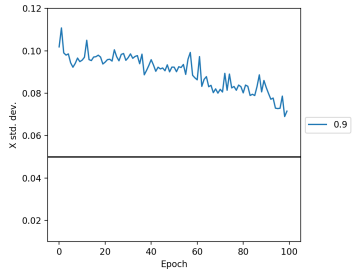
Fig. 13: x standard deviation for E-GANs with different γ for the mixture of 8 Gaussians. The target values are denoted by the black horizontal lines.



$\gamma = 0.0$

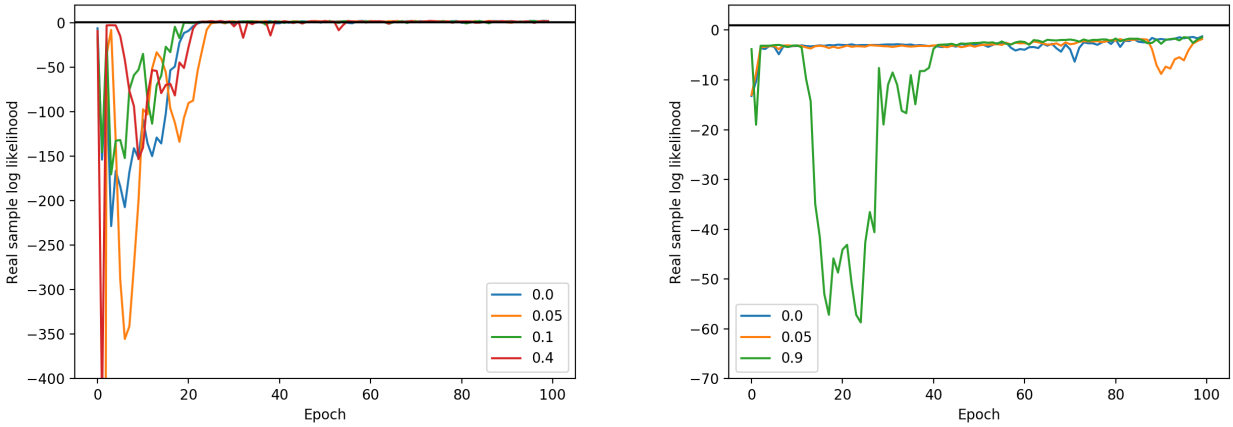


$\gamma = 0.05$



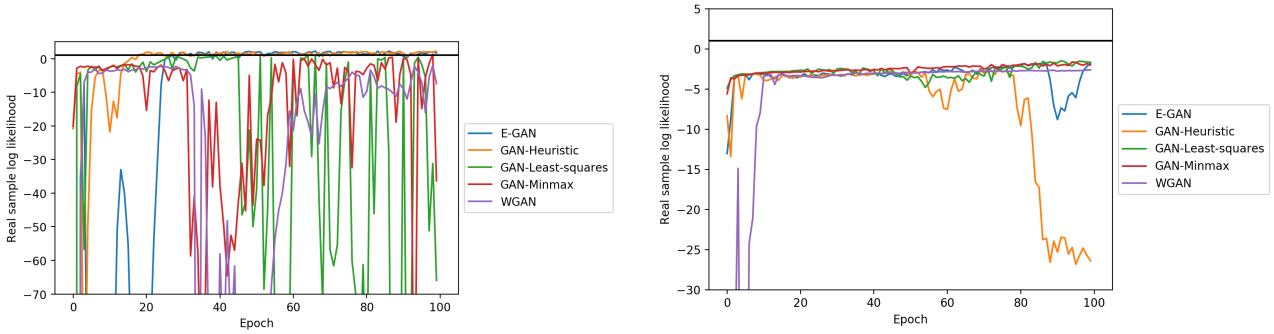
$\gamma = 0.9$

Fig. 14: x standard deviation for E-GANs with different γ for the mixture of 25 Gaussians. The target values are denoted by the black horizontal lines.



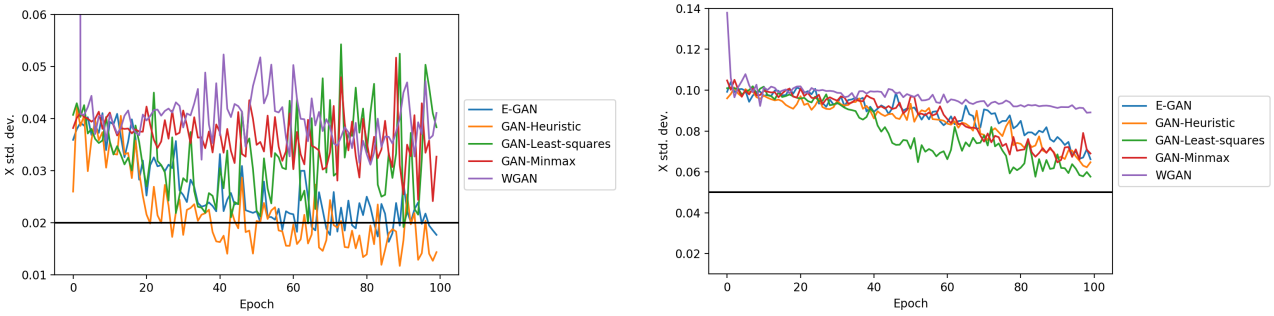
8 Gaussians 25 Gaussians

Fig. 15: Average log-likelihood for E-GANs with different γ values for the mixtures of Gaussians datasets.



8 Gaussians 25 Gaussians

Fig. 16: Average log-likelihood for different GANs for the mixtures of Gaussians datasets.



8 Gaussians 25 Gaussians

Fig. 17: x standard deviation for different GANs for the mixtures of Gaussians datasets.