Quantum Generative Adversarial Networks

Smit Chaudhary[1]

[1]Delft University of Technology

May 26, 2021

Abstract

GANs are very powerful tool to train generative models in an adversarial game. In this work, we construct a quantum counterpart of the Generative Adversarial Network. We work where the data is quantum, thus, the generator and the discriminator are both quantum. We employ a QCBM for a generator. We prepare a circuit ansatz and we also show how to take gradient and how to construct the circuit for the gradient.

1 Introduction

Generative modeling is a learning task that involves discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original data set. Or in simpler words, a generative model describes how the data is generated[1].

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods. GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two components : the generator that we train to generate new examples, and the discriminator that tries to classify examples as either real (from the domain) or fake (generated)[2]. The two networks are trained together in a zero-sum game, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples. We discuss GANs in more detail in later sections. Though here we note that a number of different generative models can act as the generator in a GAN.

Leveraging this powerful idea, one can also form Quantum Generative Models or QGANs[3]. The motivation behind the idea is two fold: firstly, QGANs could potentially provide quantum speed up. That is, there could be distributions which could be efficiently generated by quantum GANs but not so with classical GANs. Secondly, the current quantum computers in the NISQ era allow us to use quantum generators as long as they are simple parametric circuits within the reach of current machines[4]. It should be noted that there are schemes to work with quantum data as well as for encoding classical data in qubits [5]

The rest of the paper is organised as follows: In section 2, reader is introduced to the basics and structure of a GAN as well as a QGAN. Section 3 establishes the problem statement and goes over the specific implementation. Results are presented in Section 4 followed by conclusions and future line of inquiries and improvements in section 5.

2 GANs

2.1 Classical GAN

As stated above, GANs consist of two models : A generator and a discriminator. Here, a more detailed description follows. Though, it must be noted that there is a huge variety in terms of the exact implementation of a GAN and thus the introduction here should be treated as simplified pedagogical version to establish the structure.

The generator tries to capture the distribution of true example for new data example generation. The discriminator is usually a binary classifier, discriminating generated examples from the true examples as accurately as possible. The optimization of GANs is a min-max optimization problem. The optimization terminates at a saddle point that is a minimum with respect to the generator and a maximum with respect to the discriminator. That is, the optimization goal is to reach Nash equilibrium. Then, the generator can be thought to have captured the real distribution of true examples. The most straightforward modelling is having both the discriminator and generator as potentially multi layer neural network. There can be other implementations as well.

Here, the discussion is kept as general as possible, the specifics of the generator, the discriminator, the loss function etc depend heavily on the use case. The reader will get a closer look at the quantities relevant to our system in later section once the problem has been established. The quintessential example to get intuition of the GAN is to think of the generator as an art counterfeiter and the discriminator as an art critic. The critic is trying to differentiate between the real and the fake art while the generator is trying to replicate the true art as good as possible.

2.1.1 The Discriminator

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying. The discriminator's training data comes from two sources:

1. **Real Data** : Instances of data from the actual real distribution. The discriminator uses these instances as positive examples during training.
2. **Fake or Generated Data** : Instances of data generated by the generator. The discriminator uses these instances as negative examples during training.

During the training phase of the discriminator, the discriminator does the following:

- Classifies the input data as real or fake.
- The discriminator loss penalises for any misclassification.
- Back propagate the signal from the loss to update the parameters of the discriminator.

Discriminator's performance is what teaches the discriminator to do better with each iteration. Additionally, discriminator's classification is what signals the generator as well.

2.1.2 The Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real. The generator feeds into the discriminator, and the discriminator produces the the classification that the generator is trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

Thus, just like for the discriminator, the signal has to be back-propagated to adjust the parameters (for classical neural networks, they are called weights) such that the output of the discriminator is as desired. Now, as the parameters of the generator change because of this optimization, it becomes better at counterfeiting. This in turn makes the generator change its parameters to be better able to classify the data. Thus, the generator and the discriminator are trained in turns.

The training process for the generator looks like this:

1. Produce a fake sample.
2. Get the discriminator to classify it as real or fake.
3. Calculate loss from discriminator classification.
4. Back-propagate and update the parameters of the generator.

The generator and the discriminator are trained alternatively. The generator parameters are kept constant during the discriminator training phase and vice versa. As discriminator training tries to figure out how to distinguish real data from fake, it has to learn how to recognize the generator's flaws. Now, since the discriminator is basically learning where the generator is making mistakes, what it learns depends on what the actual mistakes are. Similarly, the discriminator parameters are kept constant during the generator training phase and the generator learns how to fool the discriminator.

2.1.3 The Loss functions

Since with GANs the aim is to produce a desired distribution, a measure of how close or different two distributions are is needed. There is not a single universally superior measure or loss function. Here I give an intuitive idea to the reader. First is presented the loss function as suggested in the original paper that introduced GAN and called *minimax* loss that uses cross entropy as the similarity measure:[6]

$$\mathcal{L} = \mathbf{E_r}(log(D(real)) + \mathbf{E_f}(log(1 - D(G))$$

Here,

- $D(real)$ is the discriminator's estimate that the real data is real.
- $\mathbf{E_r}$ and $\mathbf{E_f}$ are the expectation values over real and fake data.
- G is the generator's output, i.e., the fake data.
- $D(G)$ is the discriminator's estimate that the generated data is real.

Now, let me take this opportunity to introduce another loss function since it will be used later for the actual implementation. The spirit and the notation remains the same. This is called the Wasserstein loss[7]

What the discriminator wants to do is maximize the difference between its estimate of the state being real for real instances and fake instances. Thus, the loss function for the discriminator is

$$\mathcal{L} = D(real) - D(G)$$

The discriminator's goal is to maximize this function.

In a similar vein, and exactly as argued above for the minimax loss, the generator's loss function is

$$\mathcal{L} = D(G)$$

The generator tries to maximise this function.

Allow me to present an instructive example in line with the art critic-counterfeiter example stated earlier. Say the art is a 10×10 grey scale picture. The discriminator is a single layered Neural Network where it has 100 weights for the 100 pixel values and the weighted sum gives the estimate of the picture being real. The generator has its parameters such that starting from a set of random numbers, it produces 100 pixels and thus, the picture. At each training step, the weights of the discriminator are updated such that it maximizes its discriminatory behaviour and thus maximizes $D(real) - D(G)$. Note that $D(real)$ depends on 2 things : what the actual real picture is, and what the weights of the discriminator are. On the other hand, $D(G)$ depends on 2 things : What the picture generated by the generator is (which in turn depends on what the parameters of the generator are) and what the weights of the discriminator are.

I would like to emphasize a point here. The generator does not change the real image (nor does the discriminator, no one can), and it can not change the weights of the discriminator either. Thus, generator can not change $D(real)$ at all. It does influence it but does not actively change it. If this is not clear yet, it will be in a while. What the generator has the control over is its own weight and consequently the picture that is produced. It was pointed out earlier that $D(G)$ depends on two things : the weights of the generator and the weights of the discriminator. The generator tries to maximize $D(G)$ by changing its own weights. This in turn forces the discriminator to change its weights so that it classifies in the best possible way. Thus, the generator influences the weights of the discriminator through iterative learning.

Implementations of GANs are usually more complex and more involved than simple $100 \rightarrow 1$ Neural network as discriminator presented here, and they depend heavily on use cases.

2.2 Quantum GAN

Let us build on the ideas of a classical GAN and give details of a Quantum GAN. Structurally, both are the same. Quantum data is considered for quantum GANs since this is not only natural but this is where the quantum computers might do better than classical counterparts. The quantum data is some real state $|\psi_r\rangle$ Before jumping into that, it is worth taking a step back and make an important clarification. The Quantum GAN will have a generator just as in the classical one, and the model that is used is called Quantum Circuit Born Machine. We use a QCBM as an explicit generative model instead of an implicit generative model.

2.2.1 QCBMs

Quantum circuit Born machines or QCBMs are generative models. A QCBM is basically a parametric circuit which produces quantum state. Now, here it is worth noting that QCBMs are usually used as implicit generative models. An implicit generative model does not produce the direct state but it actually performs projective measurements and hands us the corresponding probability distributions. Though, in our case, the need is to produce quantum data, i.e., quantum states and thus projective measurements are not performed on the states produced. Thus, in effect the QCBM is used as an explicit model as it outputs a quantum state. More details on the exact implementations follow in the next section.

2.2.2 The Generator

The general aim of training a GAN is to find a generator G which mimics the real data source. In the quantum case, G is defined to be a variational quantum circuit whose gates are parameterized by a set of parameters $\vec{\theta}_G$. The generator G produces some state $|\psi(\theta_G)\rangle$ which, at the end of the training, should be as close to the real state $|\psi_r\rangle$ as possible.

2.2.3 The Discriminator

As in the classical case, the training signal of the generator is provided by a discriminator D, parameterized by its own set of parameters $\vec{\theta}_D$. This could be implemented in a number of ways. The discriminator can be a separate QCBM as well. In this work, the discriminator performs quantum state tomography in a way by calculating expectation values of weighted Pauli strings. Note that given many samples of the state one can have a discriminator which properly does quantum state tomography, in the current implementation it does not seek to know what the state is, rather it directly calculates the weighted sum of expectation values of the Pauli strings and these weights are the parameters of the discriminator.

2.2.4 The Loss

Just as in the case for classical GANs, one can use a number of different similarity measures and losses. For example, if the data is in terms of the probability distributions produced by a quantum system, then all those measures from classical regime carry over. On the other hand, if the data is quantum, there are some natural candidates for similarity measure. For mixed states, that is, density matrices as samples, one can use trace distance, fidelity etc. For pure states, one can use fidelity between the two states as similarity measure.

$$F = |\langle\Psi_r|\psi(\theta_G)\rangle|^2$$

Similarly, there are a number of choices for loss function too. We use the Quantum Wasserstein Loss [4] that is similar to what was introduced for the classical GANs[7]. Thus, the loss that the discriminator maximizes is given by

$$\mathcal{L} = \langle\psi_r| D |\psi_r\rangle - \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

Maximizing this value means maximizing the difference between the estimates of the discriminator on the real and the generated data.

The Generator looks to maximize

$$\mathcal{L} = \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

To understand the details and working of the system, I would redirect the reader towards the discussion in 2.1.3 and the preceding subsection since the idea is the same.

3 Implementation

3.1 Problem Statement

Although the problem was alluded to in earlier sections, it is worth it to spend some time in stating the problem statement in full and on solid grounds. Firstly, we need to know that an instance of the data point is a quantum state. The state is of size n which is known. The real state can be any n qubit state in principle but here we choose to produce it with a randomized shallow QCBM. Though note that it is not necessary to use a QCBM to produce it.

The generator is a variational circuit. Here a QCBM is used as the generator. The circuit ansatz is discussed in the next subsection.

The similarity measure between the generated and the real state is given by the fidelity between two real state as noted in subsection 2.2.4 Additionally, the Loss is the Quantum Wasserstein Loss as defined in subsection 2.2.4

3.2 Circuit Ansatz

QCBM has been mentioned a number of times without actually defining what exactly it looks like. Here we take the opportunity and describe the circuit ansatz. The variational circuit is composed of the following components:

- A layer made of $R_z(\theta)$ on each qubit.
- A layer made of $R_x(\theta)$ on each qubit.
- A layer made of CNOT between each neighbouring pair
- Repeat all these layers d times where d is the *depth* of the circuit.

Note that the angles in the parametric rotations about Z and X are the parameters of the variational circuit. These angles collectively form the vector $\vec{\theta_G}$ and the state produced is $|\psi(\theta_g)\rangle$.

Now, let us also see how the discriminator is parameterized. The discriminator is parameterized by a set of parameters which we call $\vec{\alpha}$

α_{ij} is the weight of the j^{th} Pauli on i^{th} qubit. Thus, the full weighted Pauli string is given by

$$D = \otimes_i \sum_j \alpha_{ij} P_j$$

Thus, we are essentially taking tensor products over all qubits and the sum over all single qubit Paulis. $P_j \in (I, X, Y, Z)$

The loss function for the discriminator is then

$$\mathcal{L} = \langle\psi_r| D |\psi_r\rangle - \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

For the generator, the loss is given by:

$$\mathcal{L} = \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

3.3 Code Implementation

Note that instead of using libraries such as `cirq` or `qiskit`, we simply use `numpy` arrays and do the matrix multiplications corresponding to the quantum operations that are being done[8]. The repository for the code can be found at `this link here`

The primary components of the program are:

`Class Gate`: A class that represents a quantum gate. Attributes include what gate it is (single qubit rotation about what axis, CNOT etc), what qubit it acts on, and if applicable, what is the control qubit and what are the corresponding angle in case of rotations. One can also has access to the full matrix corresponding to the gate.

`Class Circuit`: A class that represents the circuit. Made up of all the gates that form the circuit. One can get the full matrix for the full circuit.

`Class Generator`: A class that represents the generator. Made up of all the circuit for the generator. One can modify the parameters of the circuit by calculating gradients etc.

`Class Generator`: A class that represents the discriminator. Made up of all the circuit for the discriminator. One can modify the parameters of the circuit by calculating gradients etc.

The general flow of the program is as follows:

1. Create the real state that the generator has to reproduce.
2. Initialise Generator and create a fake state.
3. Calculate gradients and update parameters of generator.
4. Calculate gradients and update parameters of discriminator.
5. Iteratively go over steps 2-4 until convergence.

3.4 Calculating Gradients

Now since the maximizing and the minimizing occurs using gradient ascent and descent respectively, it is important to see how exactly are the gradients are calculated.

First, let us have a look at the generator. As argued in section 3.2, the loss function that the generator tries to maximize is given by:

$$\mathcal{L} = \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

Now, $|\psi(\theta_G)\rangle = G(\theta_G)|0...0\rangle$ where G is the generator acting on the initial state.
Thus,

$$\mathcal{L} = \langle 0...0| G^\dagger(\theta_G) D G(\theta_G) |0...0\rangle$$

$$\frac{\partial\mathcal{L}}{\partial\theta_i} = \langle 0...0| \frac{\partial G^\dagger(\theta_G)}{\partial\theta_i} D G(\theta_G) |0...0\rangle + \langle 0...0| G^\dagger(\theta_G) D \frac{\partial G(\theta_G)}{\partial\theta_i} |0...0\rangle$$

Now, only if one could construct the circuit for the gradient (which is equivalent to constructing the matrix for the gradient since here we are dealing with matrix multiplications), one would be able to calculate the gradient of the loss with respect to the i^{th} parameters θ_i.

Firstly, it is to be noted that the changing the i^{th} parameters θ_i only affects the i^{th} parametric gate. Thus, we note that:

$$R_n(\theta) = e^{-\frac{i}{2}\theta P_n}$$
$$\frac{\partial R_n}{\partial\theta} = -\frac{i}{2} P_n \times e^{-\frac{i}{2}\theta P_n}$$

Thus, the circuit for the gradient is almost identical to the original circuit with just one change. That is, the parametric gate with respect to whose parameter we are taking the gradient is replaced by a product of the corresponding Pauli and the parameterised gate.

Now that one has the route to optimizing the generator, one needs to know how to calculate the gradients for the discriminator. Let us restate the loss of the discriminator:

$$\mathcal{L} = \langle\psi_r| D |\psi_r\rangle - \langle\psi(\theta_G)| D |\psi(\theta_G)\rangle$$

Now, say one takes the gradient with respect to α_{ij}.

$$\frac{\partial\mathcal{L}}{\partial\alpha_{ij}} = \langle\psi_r| \frac{\partial D}{\partial\alpha_{ij}} |\psi_r\rangle + \langle\psi_f| \frac{\partial D}{\partial\alpha_{ij}} |\psi_f\rangle$$

Note that the D is individually linear in all the weights α_{ij}'s.

$$D_r = \otimes_i \sum_j \alpha_{ij} P_j$$
$$\frac{\partial D_r}{\partial\alpha_{ij}} = \otimes_l \sum_k \gamma_{lk}^{i,j} P_k$$

$$\gamma_{lk}^{i,j} = \begin{cases} \alpha_{ij} & (i,j) \neq (k,l) \\ 1 & (i,j) \neq (k,l) \end{cases}$$

What these complicated statements above mean is that the gradient is the exact same matrix except that the weight of the Pauli (that weight is made 1) with respect to which we are taking the derivative since the D is linear in each α_{ij}

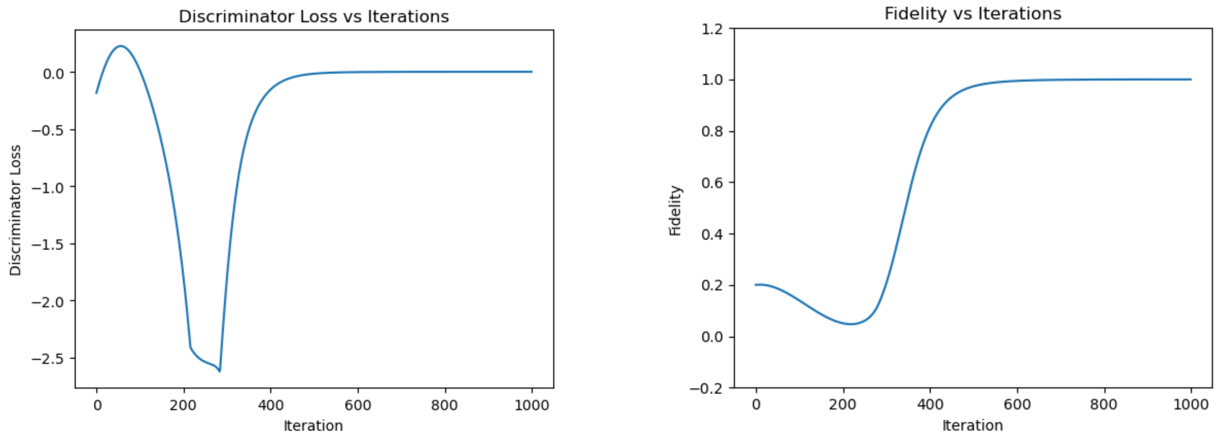
Now, being equipped with ways to calculate gradients, one can do gradient ascent for both the generator and the discriminator for their respective loss functions that they are trying to maximize.

4 Results

Before the results are presented, it is worthwhile to contemplate on what kind of behaviour to expect. What marks successful training run is the similarity between the real state and the state produced by the generator at the end of the training. Thus, the fidelity between the two states is expected to go up as the GAN is trained more.

Secondly, we also consider the Wasserstein loss. As the generator and the discriminator are trained more and more, both of them become better and better. The fake data generated by the generator is getting closer and closer to the real state. Thus, by the end of the training, discriminator randomly classifies the data as real or fake in both cases. Thus, its estimate of the state being real is $\frac{1}{2}$ in each case and the Wasserstein loss goes to zero.

Figure 1 shows the results for a system with 2 qubits, generator is a QCBM with depth 2. Thus, the number of parameters are $2 * 2 * 2 = 8$ since there are 2 parametric gates ($R_x(\theta)$ and $R_z(\theta)$) per qubit per each layer in depth.



(a) Loss vs iteration

(b) Fidelity vs iteration

Figure 1: Loss and Fidelity for QGAN with 2 qubits, randomly generated real state, depth 2

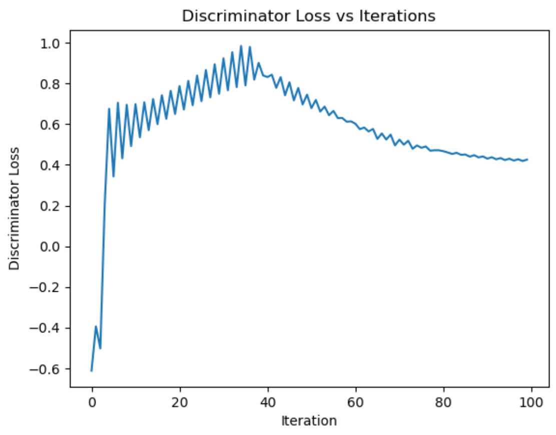
Firstly, it should be noted that the end results are as expected. The fidelity at the end of the training is 1 in addition to the loss going to 0. One can also comment on the training based on these curves. Initially, the discriminator and generator both have random parameters. The fall in fidelity until about ≈ 275 training steps is because the generator is learning the *wrong* things about the desirable direction to move in. I would like to remind the reader that the training of the generator depends on the signal from the discriminator. Since in early training steps, the discriminator has not learned enough yet, its signal is not reliable. One can see that instead of maximizing the loss, the loss is actually going down here. This is because although the discriminator attempts to maximize the loss, the generator is acting against it. After a few 100 training steps, the gradients have taken them towards their respective optima and they settle to the Nash equilibrium that is seen here.

The GAN is very sensitive to hyperparametrs. Before we exhibit this, let us first note that the parametric gates are periodic in the parameter since they are just rotations and what matters is the angle modulo 2π. One of the most important hyper parameter is the learning rate. If the learning rate of the generator is too high, then the angles shift by large values and in that case we *miss* the optima. Figure 2 shows this effect in action for a system with 1 qubit, 1 depth QCBM as a generator.

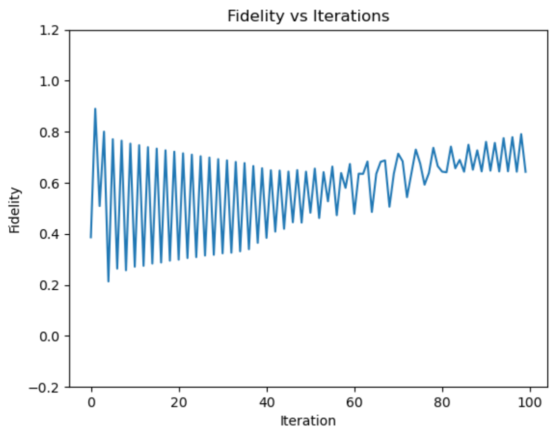
Since the training of the generator depends on the signal from the discriminator, it is important to train the discriminator right. Thus, it is common practice to update the parameters of the generator only once per several updates of the discriminator. Figure 3 shows the results for a system where the generator is updated for every 10 steps of the discriminator. The effect can be seen in the staggered nature of the plot as seen in figure 3

The generator is updated fewer times and thus, to make up for that, its learning rate is chosen to be higher, these facts collectively cause the staggered nature of the two plots every 10 steps as seen in Figure 3

An important note to be made here is that the GAN is reproducing the given real state and not the circuit that produced the said real state. As a result, the final parameters of the QCBM that acts as the generator

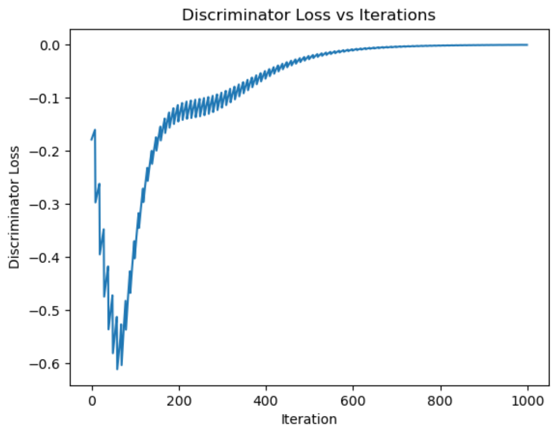


(a) Loss vs iteration

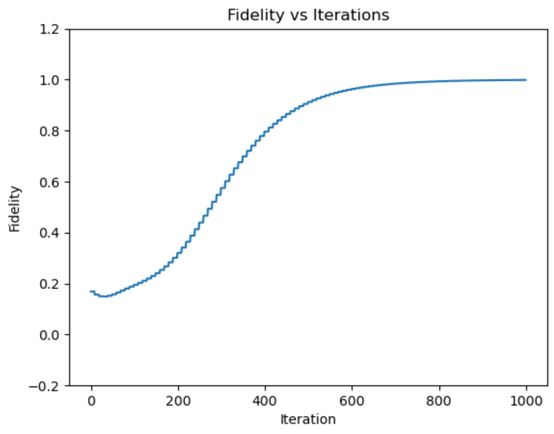


(b) Fidelity vs iteration

Figure 2: Loss and Fidelity for GAN with high learning rate for the generator



(a) Loss vs iteration for 1 qubit



(b) Fidelity vs iteration for 1 qubit

Figure 3: Loss and Fidelity for GAN when the generator is updated only once per 10 updates of the discriminator. System is 1 qubit, 1 depth QCBM as generator.

need not match those of the QCBM that produced the real state. This is to be expected since this would work even if the QCBM that produced the real state has different depth compared to the generator QCBM. As a matter of fact, the real state could have been any quantum state, not necessarily an output of a QCBM and the GAN presented here would still work.

The simulations done here are tractable. Note that the number of parameters of the discriminator grow linearly with the number of qubits n and the number of parameters in the generator grow linearly in both the number of qubits and the depth d of the QCBM.

$$||\alpha|| = \mathcal{O}(n)$$
$$||\theta_G|| = \mathcal{O}(nd)$$

On a mid level laptop with Intel core i5 processor (4 cores, 2.50 GHz), for a simulation of 3 qubit for depth 2, it takes less than 2 minutes time.

Comparison with a Classical discriminator

Before we present the comparison, I would like to emphasize a few important points. I would like to restate that the current system is a purely quantum system (though simulated classically, see sub-sections 3.3 and 3.4) because the samples that are generated are quantum states and the generator and the discriminator are equipped with Quantum Processing Units since both of them are able to take the quantum state and do operations on them. Now, if the discriminator is not quantum anymore, even if the generator is fully quantum and is producing quantum states, it can't pass it to the discriminator. This means, even if the data that we wish to produce is quantum, we must perform projective measurements on the states and then pass the classical probabilities to the discriminator.

The Discriminator we use is a simple feed forward neural network with 1 hidden layer. The input layer has dimensions 2 to take the id of the state and the probability of that state. It outputs its estimate of this set of state and probability resulting from measurements on the real state. The generator is still quantum and the same QCBM as discussed earlier. Though this time, the generator output is the probabilities that result from projective measurements to accommodate the classical nature of the discriminator.

The table below shows the comparison for representative results of the fidelity and the time taken for a fully quantum system and the one with classical discriminator. Note that the second row that says classical has only the discriminator which is classical as has been discussed above. This is for a 1 qubit system with the QCBM of length 1.

	Fidelity after 500 steps	Time to 500 steps (in s)	Fidelity after 1000 steps	Time to 1000 steps (in s)
QuGAN	0.991	2.5	9.996	5.1
Classical	0.445	7.2	0.604	14.2

One notes that the fidelity of the resultant state is lower in the second case in addition to it taking longer to train. While this does seem like a verification that quantum GANs are powerful and in some ways look even more powerful than quantum ones, one should be careful in terms of the conclusions one draws from this. Firstly, the classical GAN only promises to produce the same probability distribution and not the same state. This means the two are aiming towards two different goals. To make it more clear, the Quantum GAN if asked to reproduce the $|+\rangle$ state, would attempt to produce the $|+\rangle$ state. But the one with classical discriminator could very well produce the $|-\rangle$ state instead since it has the same distribution when measured in the standard basis. Also note that for quantum case, there is no latent space from which the generator takes data to produce distributions (or equivalently can say that the same point from the latent space is given) as in the works of Dallaire-Demers and Killoran [3]. So in a way, this is like comparing apples and oranges.

Additionally, the results I presented for the system with classical discriminator are far from the most optimal ones since I just implemented a simple neural network.

5 Conclusions and Future Works

In this work, I firstly introduced Classical GAN, its structure, the components and additionally, the Wasserstein loss function. Expanding on that, Quantum GANs were introduced and constructed. Then a QCBM is employed as a generator and the components of the Quantum GANs are defined. I also introduced the Quantum counterpart of the Wasserstein loss function. We then built a Quantum QGAN and present the method to construct circuit for the gradient of the loss functions. Next, the results are presented and hyper parameters are optimized to find appropriate learning rates. This results in the quantum GAN producing a state close to the desired state. We also replace the quantum discriminator with a classical neural network and compare the results for them.

The most important upgrade that can be made on the current implementation is to add regularisation. Adding regularisation would help in discouraging over fitting and would open avenues to accurately implement more complex GANs.

Similar to replacing the quantum discriminator with a classical one, one replace the quantum generator with a classical generator instead. Since the data produced would be classical, one needs to encode it into qubits for the quantum discriminator to classify.

Another straight-forward extension here would be to deal with mixed states instead of pure states. In that case, we will be dealing with density matrices. As a result, the similarity measure as well as the evolution and expectation value calculations would change only slightly, the rest of the structure remains the same.

Since the QCBMs that we are dealing with would be small for a small number of qubits, one can implement the circuit on an actual quantum hardware and see the effects of the noisy systems that we currently have.

Acknowledgements

Firstly, I would like to thank Vedran and Jordi for the chance to do a project that turned out to be an immense learning opportunity and for regular fun classes that equipped me with the strong fundamentals to undertake this. Additionally, I would also like to thank Casper, Simon, Stefano, Emile and all the TAs for the tutorials as well as for responding to my last minute mails. I would also like to acknowledge my friend Georgios Sotiropoulos for being a great sounding board and asking great questions. Explanation of concepts in the report is majorly based on our conversations.

The code can be found at `https://github.com/smitchaudhary/QGANs`

References

[1] *Google developer's platform.* `https://developers.google.com/machine-learning/gan/generative`. Accessed: 2021-05-22.

[2] Jie Gui et al. *A Review on Generative Adversarial Networks: Algorithms, Theory, and Applications.* 2020. arXiv: `2001.06937 [cs.LG]`.

[3] Pierre-Luc Dallaire-Demers and Nathan Killoran. "Quantum generative adversarial networks". In: *Physical Review A* 98.1 (July 2018). ISSN: 2469-9934. DOI: `10.1103/physreva.98.012324`. URL: `http://dx.doi.org/10.1103/PhysRevA.98.012324`.

[4] Shouvanik Chakrabarti et al. "Quantum wasserstein generative adversarial networks". In: *Advances in Neural Information Processing Systems.* 2019, pp. 6778–6789.

[5] Samuel A. Stein et al. *QuGAN: A Generative Adversarial Network Through Quantum States.* 2020. arXiv: `2010.09036 [quant-ph]`.

[6] Ian J. Goodfellow et al. *Generative Adversarial Networks.* 2014. arXiv: `1406.2661 [stat.ML]`.

[7] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN.* 2017. arXiv: `1701.07875 [stat.ML]`.

[8] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.