TECHNICAL UNIVERSITY OF CRETE

Introduction to Quantum Computing

Semester Project Report

July 1, 2025

Dimas Christos, 2021030183

Abstract

The integration of classical data into quantum systems is a critical challenge for the practical deployment of quantum algorithms.

This report provides a comprehensive literature review and critical analysis of the paper "Quantum Generative Adversarial Networks for learning and loading generic probability distributions" by Zoufal, Lucchi, and Woerner (2019), which addresses this significant challenge and presents a notable advancement in mitigating the data loading bottleneck in quantum computing, and this review aims to evaluate its contributions and the robustness of its approach.

The paper introduces a quantum Generative Adversarial Network (qGAN) methodology, employing a variational quantum circuit as a generator and a classical neural network as a discriminator, to learn and approximately load generic probability distributions with polynomial $O(poly(n))$ gate complexity.

This review examines the paper's core techniques, the architecture of the qGAN, its training process, and its demonstrated applications, including simulations and experiments on IBM Q quantum processors for tasks like modeling log-normal distributions relevant to quantum finance. Furthermore, this report will detail efforts to understand and potentially reproduce key findings, assessing the method's effectiveness and practical considerations.

Contents

1 Introduction

The realization of quantum advantage for many promising quantum algorithms is fundamentally linked to the ability to efficiently load classical data into quantum states. However, a significant hurdle exists: the exact representation of generic data or probability distributions within an n-qubit system typically requires a number of quantum gates scaling exponentially, $O(2^n)$. This exponential scaling can easily dominate the overall complexity of a quantum algorithm, thereby diminishing or even negating any potential quantum speedup . Consequently, developing methods for efficient data loading is paramount for unlocking the power of quantum computation.

2 Paper Summary

After reading the paper, it is clear that the authors address a significant challenge in quantum computing: the efficient loading of classical data into quantum states. The paper highlights the exponential complexity of exact data loading, which can hinder the performance of many promising quantum algorithms. But how can we observe this problem?

2.1 Problem Statement

First and foremost, it is important to understand why loading classical data into quantum states requires exponential gate complexity. While obvious, it is worth noting that with n qubits, we can represent 2^n different complex amplitudes:

$$|\psi\rangle = \sum_{i=0}^{2^n-1} c_i|i\rangle$$

so, loading classical data into a quantum state actually meens that we need to apply quite a few sequential gate operations, in order to turn a simple initial state, like $|0\rangle^{\otimes n}$, into an abritary target state, where any of the 2^n amplitudes c_i can be any desired value, fact that lead us up to $O(2^n)$ gates needed, even with the best known methods, to load an exact representation of a generic data structure into a n-qubit state.

The above mentioned fact is a fundamental limitation for many quantum algorithms. That happens because, data loading can easily dominate the overall complexity of the algorithms and diminish or even negate any potential quantum speedup, by setting its complexity up to $O(2^n)$.

To address this challenge, the authors propose a novel approach using a hybrid quantum-classical implementation of a Generative Adversarial Networks (GAN), to train a quantum channel such that it reflects a probability distribution implicitly given by data samples. Let's talk a bit about GANs though.

2.2 Generative Adversarial Networks

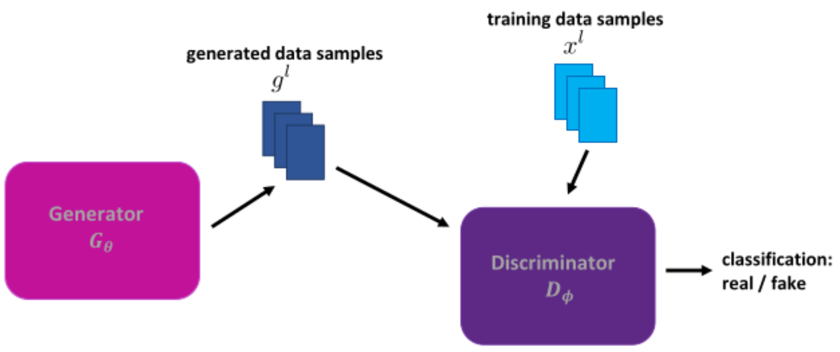
Generative Adversarial Networks (GANs) are a class of machine learning frameworks where two neural networks, a generator and a discriminator, are trained simultaneously. The generator creates data samples, while the discriminator evaluates them against real data, providing feedback to the generator. This machine learning framework has many advantages, but in our case the biggest is that this adversarial process leads to the generator producing increasingly realistic samples.

Considerining about the above model, we can imagine that the loss functions of generator and discriminator connect in a way. And this consideration is confirmed, with the following equations:

$$L_G(\phi, \theta) = -\frac{1}{m}\sum_{l=1}^{m}[\log(D_\phi(G_\theta(z^l)))]$$

$$L_D(D_\phi, G_\theta) = \frac{1}{m}\sum_{l=1}^{m}[\log D_\phi(x^l) + \log(1 - D_\phi(G_\theta(z^l)))]$$

for dataset $x^l \in \{x^0, ..., x^{n-1}\}$, $z^l \sim p_{prior}$ and m the size of the batches.



Now that we have a basic understanding of GANs, we can proceed to the proposed solution of the paper.

2.3 Proposed Solution

The authors introduce a hybrid quantum-classical Generative Adversarial Network (qGAN) methodology, using a quantum generator and a classical discriminator. Regarding the quantum generator, it is a parametrized quantum circuit that is trained to transform a given n-qubit input state $|\psi_{in}\rangle$ to an n-qubit output state.

$$G_\theta|\psi_{in}\rangle = |g_\theta\rangle$$

Parametrized quantum circuits (PQCs) are quantum circuits that consist of both fixed and parametrized gates. The fixed gates are typically multi-qubit entangling gates, such as the Controlled-NOT (CNOT) or Controlled-Z (CZ) gates, which are essential for creating complex correlations between qubits. The parametrized gates are usually single-qubit rotation gates, like rotations around the Y-axis (R_Y) or Z-axis (R_Z). The operations of these gates depend on a set of adjustable classical parameters, represented by θ, which typically define the rotation angles. PQC, are often used within a hybrid quantum-classical feedback loop, like Variational Quantum Circuits (VQAs). During this usage the process is iterative and shown below:

- Circuit is executed in a quantum processor with an inital state of parameters θ, to prepare the state.
- Circuit's output state is measured repeatedly.
- After measurement, a cost function that runs on a classical computer, evaluates the measurements.
- A classical optimization algorithm calculates an updated set of parameters θ.
- The new parameters are fed back into the quantum circuit, and repeat the process until whe reach our goal (minimize or maximize the function).

This usage is really similar to a Machine Learing model as you can see. But how the circuit is constructed in our case?

2.4 Quantum Circuit

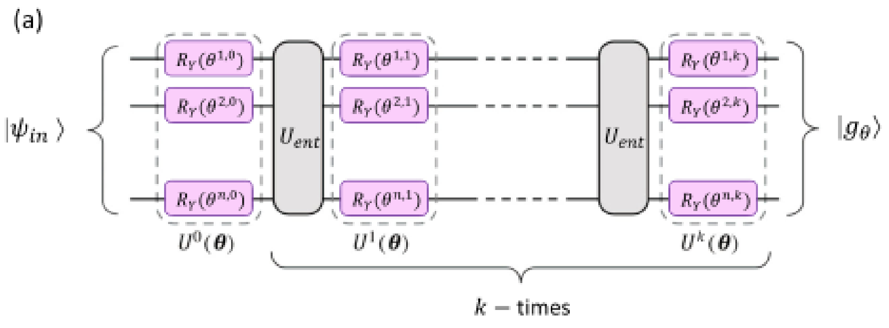
In order to create the quantum generator they consider a variational form of parametrized single-qubit rotations, here Pauli-Y-rotations and blocks of two-qubit gates, here CZ-gates, called entanglement blocks Uent. We first see a layer of the R_Y gates, and then k alternating layers of entanglement blocks and R_Y gates, where k is called the depth of the circuit. And the occured questions here about the generator are many:

1. How many qubits should we use?
2. How many layers should we use?
3. Why we use the particular gates?
4. What is the gate complexity of the circuit?

However, the answers to these questions are not left to the reader. The qubits that should be used are n, where 2^n is equal to the discrete size of the target distribution. The layer number should be such that discriminator and generator are almost equal but not too shallow, so that the generator can really learn, so we reach to its value after some experiments. The thought behind the choice of the gates, was to keep the state amplitudes as they are and only change their phases in order to not preturb the modeled probability distribution. Furthermore, the gate complexity of the circuit is:

$$\underbrace{O(n)}_{\substack{\text{first layer of}\\ R_Y\text{ rotations}}} \quad + \quad \underbrace{k \ \cdot O(n^p)}_{\substack{\text{entangler } U_{\text{ent}}\\ +\text{ subsequent } R_Y,\\ \text{repeated } k \text{ times}}} \quad = O\big(n+k\,n^p\big) \ = \ O\big(n+n^q\,n^p\big) \ = \ O\big(n^{p+q}\big) \ = \ O\big(\text{poly}(n)\big).$$

where p indicates how many gates used in one entangler/rotation block and q how many such blocks exist.



Applying the n-qubit generator to the input state gives:

$$|g_\theta\rangle = G_\theta|\psi_{in}\rangle = \prod_{p=1}^{k} \left(\otimes_{q=1}^{n}(R_Y(\theta^{q,p}))U_{ent}\right) \otimes_{q=1}^{n} (R_Y(\theta^{q,0}))|\psi_{in}\rangle$$

When $|g_\theta\rangle = G_\theta|\psi_{in}\rangle$ is computed, we can observe that the first column of G_θ is extracted regardless its content and we can find that for n qubits:

$$G_\theta|\psi_{in}\rangle = \otimes_{i=1}^{n} \begin{bmatrix} a & b \\ c & d \end{bmatrix} |0\rangle = \otimes_{i=1}^{n} \underbrace{\begin{bmatrix} a \\ c \end{bmatrix}}_{a|0\rangle + c|1\rangle}$$

That gives us the n-qubit state: $\sum_{j=0}^{2^n-1}[G_\theta]_{j,0}|j\rangle$, where the elements $[G_\theta]_{j,0}$ are complex functions of all the parameters θ in the circuit. So we conclude that:

$$|g_\theta\rangle = \sum_{j=0}^{2^n-1} \sqrt{p_\theta^j|j\rangle}$$

2.5 Discriminator

Discriminator is a classical neural network consists of a series of fully connected layers, which takes the output of the quantum generator and outputs a single value. More specifically:

- **Input layer:** Receives the flattened output from the quantum generator.
- **Hidden layers:** 2 fully connected layers with the nonlinear activation function LeakyReLU used, to capture complex patterns.
- **Output layer:** A single neuron using a sigmoid activation function to output a probability value, distinguishing between real and generated data.

2.6 Dataset and Preperation

The only thing left now is to talk about the real data that are used to train the discriminator. As the authors mention, the data are generated from a log-normal, a triangular and a bimodal distribution. Specifically:

- Log-Normal: $\mu = 1$ and $\sigma = 1$
- Triangular: $l = 0$, $u = 7$ and $m = 2$
- Bimodal: $\mu_1 = 0.5$, $\mu_2 = 3.5$, $\sigma_1 = 1$ *and* $\sigma_2 = 0.5$

This method is applied for 20,000 samples of the above distributions, rounded to integer values. All distributions are truncated to the range [0, 7], because we want to achieve a natural mapping between the sample space of the training data and the states that can be represented by the generator for simplicity. In case natural mapping does not exist, affine mapping can be applied classically after measuring the quantum state. However, when the resulting quantum channel is used within another quantum algorithm the mapping must be executed as part of the quantum circuit. Such affine mapping can be implemented in a gate-based quantum circuit with linearly many gates.

We also need to prepare the generator's input state $|\psi_{in}\rangle$, in order to ensure that it is trained effectively. The input state is initialized in 3 different ways either with a discretized uniform, or with a discritizer normal, or with a randomly chosen distribution.

Preparing a uniform distribution on 3 qubits requires the application of 3 Hadamard gates, i.e. one per qubit. Loading a normal distribution is achieved by fitting the parameters of a 3-qubit variational quantum circuit with depth 1 with a least squares loss function, more specifically the goal is to minimize the distance between the measurement probabilities p_j of the circuit output and the probability density function of a discretized normal distribution. For both cases, we sample generator parameters from a uniform distribution on [-0.1, 0.1]. Furthermore, regarding the random distribution $|\psi_{in}\rangle$ is set to $|0\rangle^{\otimes 3}$ and initialize the parameters of the variational form, in order to follow a uniform distribution on [-π, π]

2.7 Training

After that, the samples are fed, shuffled, into the discriminator in batches of 2000 samples per training step (epoch). There are totally 2000 training steps, which means that the discriminator is trained on the entire dataset multiple times, allowing it to learn the underlying patterns and characteristics of the data effectively. In the context of training a quantum representation of some training data's underlying random distribution, the Kolmogorov–Smirnov statistic as well as the relative entropy represent suitable measures to evaluate the training performance.

2.8 Application in finance

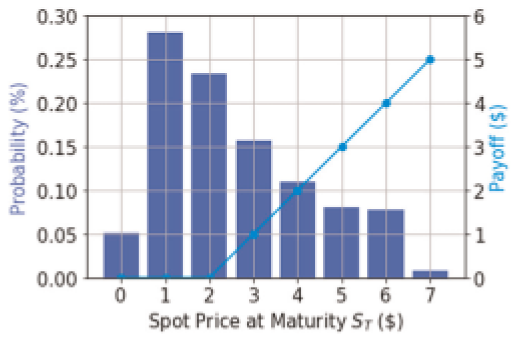
One final question arises though. In what case it can be used, in order to solve a real world problem? Authors propose a use case in the context of European call option pricing, where they employ qGANs to learn and load a model for the spot price of that asset.

The owner of a European call option is permitted, but not obliged, to buy an underlying asset for a given strike price K at a predefined future maturity date T, where the asset's spot price at maturity S_T is assumed to be uncertain. If $S_T \leq K$, i.e., the spot price is below the strike price, it is unreasonable to exercise the option and there is **no payoff**. However, if $S_T > K$, exercising the option to buy the asset for price K and immediately selling it again for S_T can realize a **payoff of** $S_T - K$. Thus, the payoff of the option is defined as $\max(S_T - K, 0)$. That brings us to the goal, that is to evaluate the expected payoff $\mathbb{E}[\max(S_T - K, 0)]$, whereby S_T is assumed to follow a particular random distribution. This corresponds to the fair option price before discounting.

The fair price of the option is estimated by sampling from the resulting distribution, as well as with a QAE (Quantum Amplitude Estimation) algorithm, *see Supplementary Methods 8.3 for explanation*, that uses the quantum generator trained with IBM Q Boeblingen (20 qubit chip) for data loading. In the refering case, a small example was implemented based on the analytically computable standard model for European option pricing, the Black-Scholes model.

Regarding the Black-Scholes model, it is worth mentioning that it often over- simplifies the real circumstances. In more realistic and complex cases, where the spot price follows a more generic stochastic process or where the payoff function has a more complicated structure, options are usually evaluated with Monte Carlo simulations. The catch in Monte Carlo simulation is that its estimation error is $\epsilon = O(1/\sqrt{N})$ that is higher than the $\epsilon = O(1/N)$ error estimation, of the QAE algorithm.

So after deciding to use Black-Scholes model as the reference point in order to compare the two above mentioned methods, they go on by doing the things below. According to the Black-Scholes model, the spot price at maturity S_T is assumed to follow a log-normal distribution like the ones created before, so the qGAN can be trained to learn this distribution successfully, after a carefull initialization though, with normal and uniform distributions to constitute the best options based on the previous experiments. Also, qGAN model was trained on real quantum hardware (IBM Q Boeblingen) with 20 qubits, using a batch size of 2000 samples and a circuit with the architecture mentioned above, Ansatz type, with 1 layer depth for 200 training steps (epochs).



The resulting quantum generator loads a random distribution that approximates the spot price at maturity S_T. More specifically, they integrated the distribution loading quantum channel into a quantum algorithm based on QAE, to evaluate the expected payoff $\mathbb{E}[\max\{S_T - K, 0\}]$, for $K = \$2$ approximate data loading. Finally, they compare the estimation of $\mathbb{E}[\max S_T - K, 0]$, done with three different methods:

Method	Distribution	Samples	CI
1. Analytic (Exact)	$X \sim$ Log-N(1, 1)	-	-
2. Monte Carlo (Quantum HW)	$\lvert g_\theta\rangle$	1024	± 0.0848
3. QAE (Simulated)	$\lvert g_\theta\rangle$	256	± 0.0710

Table 1: Comparisson of option price estimation methods, their samples and their confidence intervals.

1. An analytic evaluation with the exact (truncated and discretized) log-normal distribution p_{real},
2. A Monte Carlo simulation utilizing $|g_\theta\rangle$ trained and generated with the quantum hardware (IBM Q Boeblingen), i.e., 1024 random samples of S_T are drawn by measuring $|g_\theta\rangle$ and used to estimate the expected payoff, and
3. A classically simulated QAE-based evaluation using $m = 8$ evaluation qubits, i.e., $2^8 = 256$ quantum samples, where the probability distribution $|g_\theta\rangle$ is trained with IBM Q Boeblingen chip.

The resulting confidence intervals (CI) are shown for a confidence level of 95% for the Monte Carlo simulation as well as the QAE. The CIs are of comparable size, although, because of better scaling, QAE requires only a fourth of the samples.

Since the distribution is approximated, both CIs are close to the exact value but do not actually contain it. Note that the estimates and the CIs of the MC and the QAE evaluation are not subject to the same level of noise effects. This is because the QAE evaluation uses the generator parameters trained with IBM Q Boeblingen but is run with a quantum simulator, whereas the Monte Carlo simulation is solely run on actual quantum hardware. To be able to run QAE on a quantum computer, further improvements are required, e.g., longer coherence times and higher gate fidelities.

3 Result Reproduction

The next step, after the paper analysis, was to reproduce some results of it. For this task it was decided to reproduce the figure that illustrates the PDFs corresponding to $|g_\theta\rangle$ trained on samples from a log-normal distribution using a uniformly initialized quantum generator and its relative entropy (light blue plot).

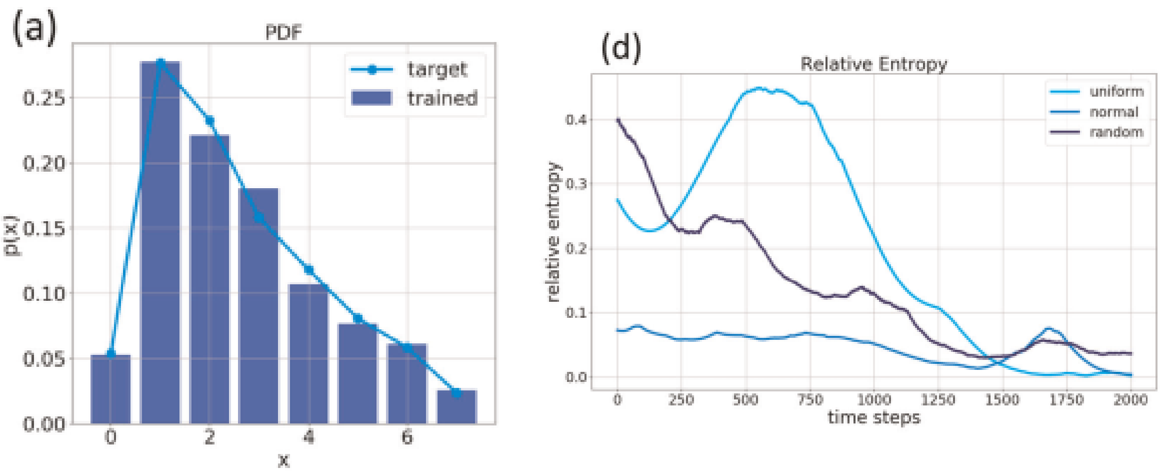


Figure 1: Fig.4 from the paper that we will try to recreate

In order to reproduce the results of the paper, we need to use the SDK Qiskit, a Python library for Quantum Computing developed by IBM software engineers, that helps in simulating quantum circuits on classical machines , and PyTorch, a machine learning library to represent the classical machine processes. The following steps outline the process of reproducing the results.

3.1 General configuration setup

After reading the paper, we can extract a lot of information about the way that authors implement this hybrid Q-GAN. First of all, we see that we need to use $\log_2(N)$ qubits, where N is the discrete values of the distribution (Truncated in $[0, 7]$). Also after reading the paper, we observed that with the given Discriminator architecture, a good depth for the quantum circuit is 3 layers and we decide to also use the given batch size (2000), distribution samples (20000) and training steps (2000). The two things that we decided to change were the learning rate of both optimizers that instead of 10^{-4}, was set to $2 \cdot 10^{-4}$ and the sample of generator parameters from the uniform distribution, that was now set to 10^{-2} instead of 10^{-1}, small changes but very helpful indeed in order to fine-tune the neural networks. Finally the option to run the specific model in GPU (cuda) was given.

3.2 Dataset generation

Dataset in our case, is a log-normal probability distribution created according to the paper's recommendations. More specifically, we generated 20000 samples from this distribution and truncate all of them in [0, 7], in order to have 8 discrete values. The continuous samples of the probabiliy distribution, $X \sim \text{Log-N}(1, 1)$ (because $\mu_x = 1$ and $\sigma_x^2 = 1$), are discretized and inserted into a tensor for easy processing and compatibility.

3.3 Quantum circuit, generator and discriminator creation

The quantum circuit is implemented using the Qiskit library, and it helps us implement a custom Variational Quantum Algorithm (VQA) for the generator. It is an easy circuit to implement in hardware level, as all circuits in VQA are, and it is based on the paper's description. More specifically, we use a circuit with 3 layers, each layer containing $R_Y(\theta)$ rotation gates and controlled-Z gates (CZ) to create entanglement between qubits. Before those 3 layers we also add a layer of Hadamard gates to create superposition and an initial $R_Y(\theta)$ gate layer, as the paper recommends. Using the above mentioned circuit, we can create the generator initialized in a uniform distribution.

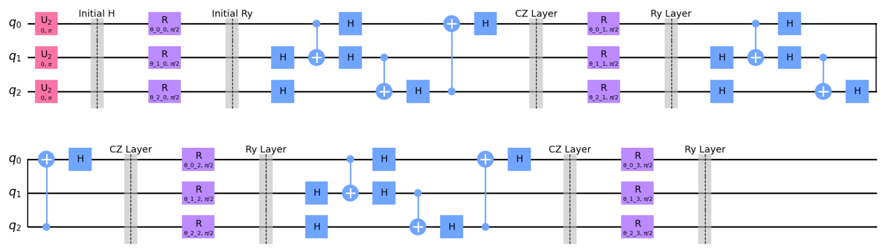


Figure 2: Quantum circuit used in the generator of the hybrid Q-GAN.

The discriminator is implemented using PyTorch, and it is a simple feed-forward neural network with 2 hidden layers. The first hidden layer has 50 neurons and uses the LeakyReLU activation function, while the second hidden layer has 20 neurons and also uses LeakyReLU. In the end, we have a single output neuron with a sigmoid activation function to produce a probability value between 0 and 1.

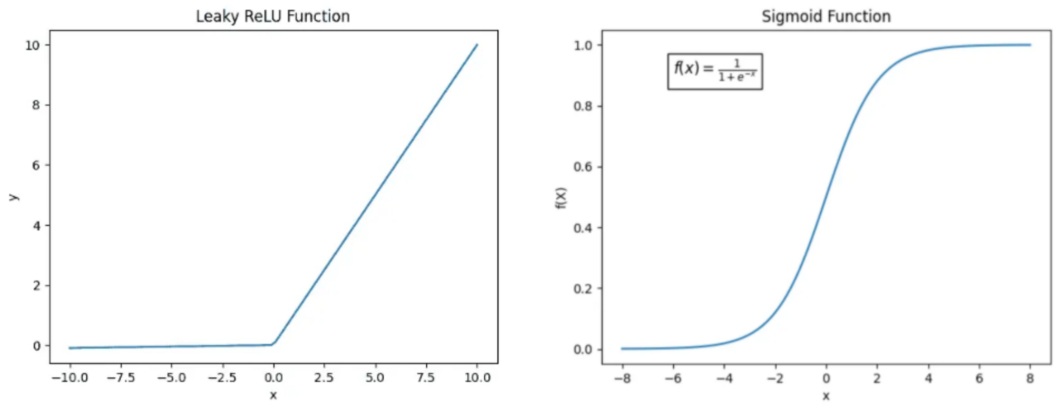


Figure 3: Plots of LeakyReLU and Sigmoid activation functions

3.4 Training loop

The training loop is implemented using PyTorch, and it consists the following steps:

1. Generate a one-hot batch[1]of samples from the truncated log-normal distribution and train discriminator.
 2. Generate a one-hot batch of samples from the generator's output distribution, also to train discriminator.
3. Train generator based on discriminator's output.
4. Calculate relative entropy, Kolmogorv-Smirnov statistic and store process.
5. Backpropagate the losses and update the weights of both the generator and discriminator using the AMSGRAD optimizer.
6. Repeat steps 1-5 for the specified number of training steps.

3.5 Results visulization

Finally, we can see the results and they are great. The loss functions of generator and discriminator, converge as expected in GANs and their metrics indicate that quantum generator is excellent at creating realistic data. More specifically, we recorded:

1. **Relative Entropy**: 0.0004
2. **Kolmogorv–Smirnov statistic**: 0.0081

Thing that means, the distribution is accepted by far, since the confidence level on paper was 95%. Also we can see that loss functions converge a bit late but this is probably due to the difference between the initialization of the circuit parameters.

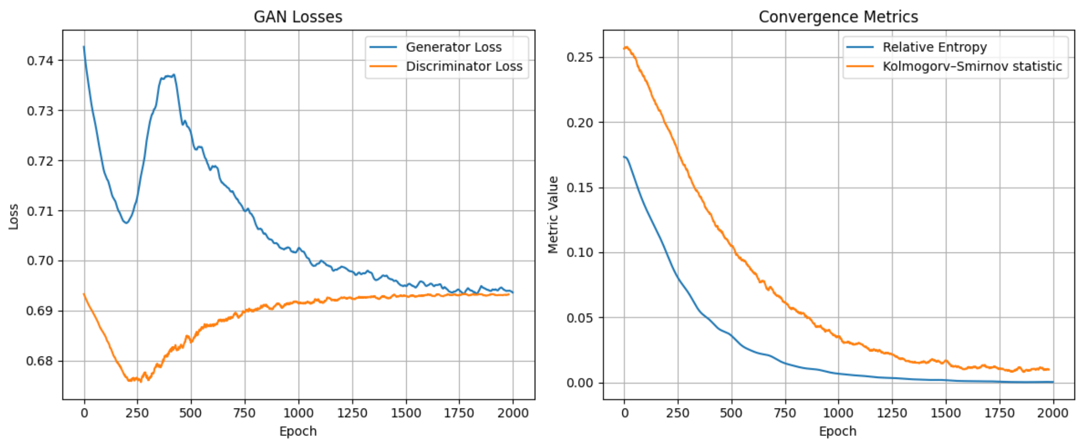


Figure 4: Loss functions, Relative entropy and KS statistic plots

Furthermore, below is presented the actual generated Log-Normal distribution in comparisson with the real one.

[1]A one-hot vector is a binary vector where only one element is 1 and all others are 0, often used to represent categorical data.

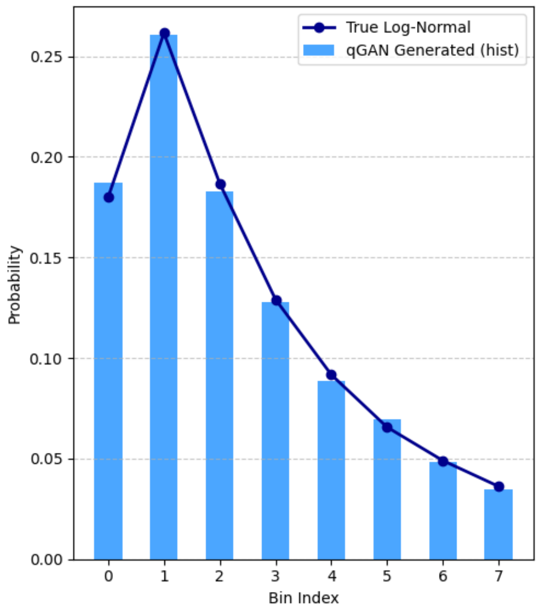


Figure 5: True Log-Normal versus qGAN generated

4 Comparissons

After the attempt to reproduct the results, with some considerable challenges, we concluded in quite similar results and slightly better in terms of the Log-Normal distribution generation. This can be shown by the side-by-side comparissons below, where we can see the generated distribution and the relative entropy in each case.

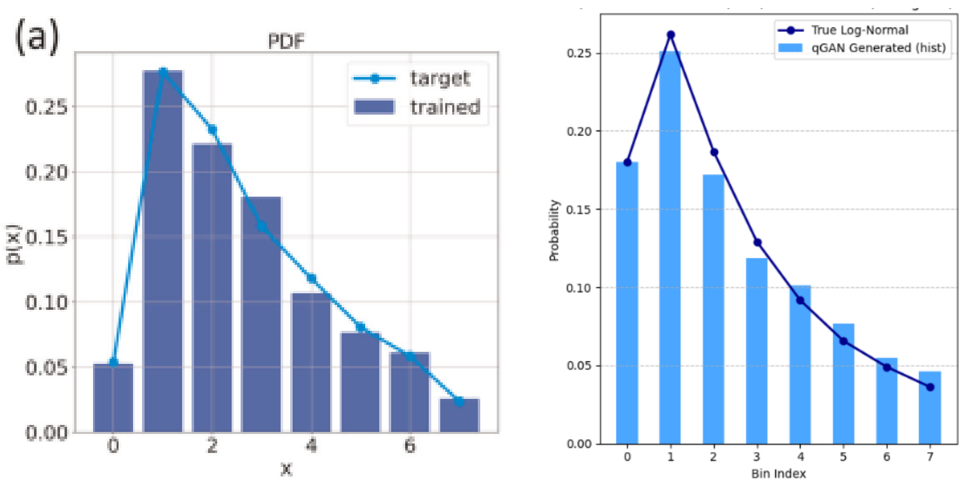


Figure 6: Comparisson of paper's figure (left) and reproduction's figure with **paper's setup** (right)

As we can see by the comparisson above the results given the same configuration are quite similar. While changing the setup to the one we refered below we achieve significantly better results.

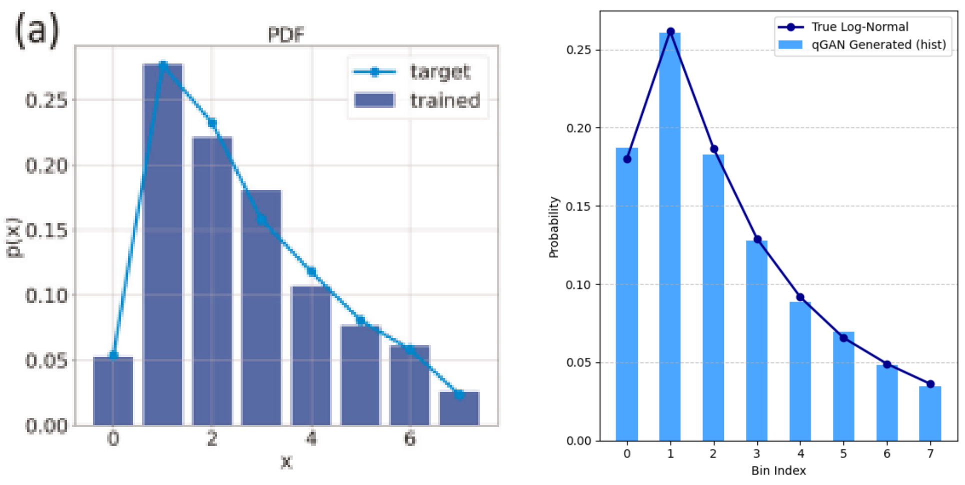


Figure 7: Comparisson of paper's figure (left) and reproduction's figure (right)

As we can observe, in the reproduction the relative entropy plot is smoother, still though they both converge to zero.

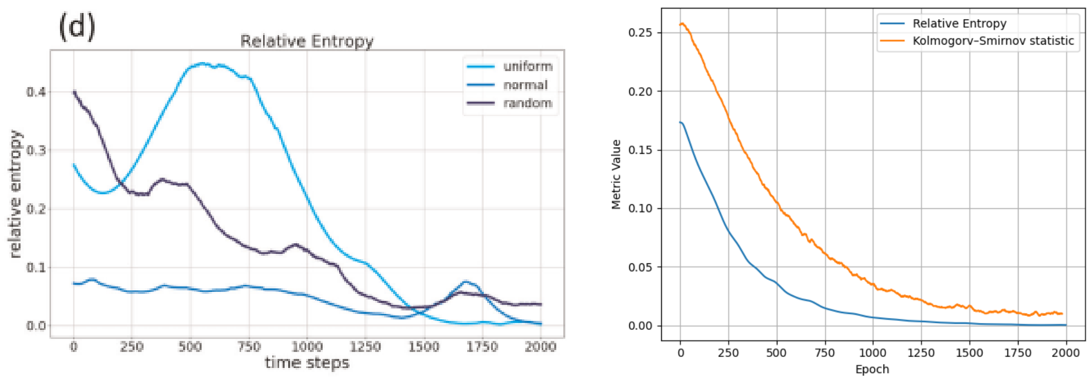


Figure 8: Comparisson of paper's relative entropy (left) and reproduction's relative entropy (right)

The challenges we encountered were mostly on how to process the dataset in order generator and discriminator to have a smooth workflow, how to give correct penalties in each model and on how to make the training faster and more efficient.

The above mentioned challenges were they main reason of two crucial additions / modifications:

1. To not work with simple data batches but with one-hot encoded batches
2. To use different loss functions to evaluate the training process of each network

One-hot encoding helps not only ensuring that the data can be processed correctly, but also effectively by reducing the computational load a lot. But someone might question, how can a method used to represent categorical data help us with a probability distribution?

That question, is the reason why the process of the one-hot encoding is shown below:

- We assign the indices of the probabilities into a new vector, called ***sampled indices*** and sized ***batch size***, where each index is the value of this vector, that occurs as frequent as the probabilities of the distribution indicate.
- The samples are then created by one-hot encoding indices, where for each ***sampled indices*** index, we create a vector with 1 set to the position that ***sampled indices*** value indicates.

An extensive example of how it works is placed in the appendix.

Furthermore, the reason why different loss functions where needed for discriminator and generator is that, discriminator, is evaluating batches of individual samples, while generator evaluates against a full probability distribution.

5 Discussion

The reproduction of the qGAN for loading a log-normal distribution provided several practical insights into the methodology's strengths, challenges, and broader implications. The process highlighted not only the theoretical advantages of the approach but also the nuances involved in a successful implementation.

5.1 Practicality of qGANs for Quantum Data Loading

The primary advantage of the qGAN methodology, as confirmed through this project, is its ability to learn and load a probability distribution with polynomial gate complexity, specifically $O(\text{poly}(n))$. This is a significant improvement over classical methods that require an exponential $O(2^n)$ number of gates for exact loading, which often negates any potential quantum speedup. However, the practical implementation revealed that achieving this theoretical advantage is non-trivial. The training process is highly sensitive to hyperparameter choices, including learning rates, optimizer settings (such as using AMSGrad), and the specific structure of the loss functions. The iterative tuning required to stabilize the training underscores that while qGANs are powerful, they are not a "plug-and-play" solution and demand careful calibration. The key would be the careful discretization of the data into a format suitable for the quantum generator's qubit register and ensuring the generator's ansatz has sufficient expressive power to model the target distribution's complexity

5.2 Insights Gained from Reproducing This Work

This reproduction effort offered several critical insights. Firstly, the importance of data representation became evident. The transition to using one-hot encoded batches was a crucial step for stabilizing the training process and ensuring the classical discriminator could process the discrete outputs of the quantum generator effectively. This highlights a key challenge in hybrid quantum-classical models: ensuring a smooth and compatible data pipeline between the two components. Secondly, the choice of loss function proved to be a critical factor. The initial instability was largely addressed by adopting a loss calculation for the generator that directly uses its output probability distribution, avoiding non-differentiable sampling steps and providing a more stable gradient. This contrasts with the discriminator's loss, which is more effectively calculated on batches of individual samples. This distinction underscores the unique challenges of training quantum generators.

5.3 Limitations

Despite the successful outcome, several limitations and challenges were encountered:

- **Training Instability:** As is common with GANs, the training process was prone to instability. Finding a stable equilibrium between the generator and discriminator required multiple iterations of tuning hyperparameters, loss functions, and regularization techniques.
- **Resource Constraints:** The entire process was conducted via simulation. Even for a small system of 3 qubits, the training was computationally intensive, requiring significant time and resources. Scaling this simulation to a larger number of qubits would become exponentially more challenging, emphasizing the need to eventually run such algorithms on actual quantum hardware.
- **Hyperparameter Sensitivity:** The model's performance was highly sensitive to the learning rates, the weight of the GP term, and the ratio of discriminator-to-generator updates. There is no one-size-fits-all configuration, and the optimal setup had to be discovered empirically.

5.4 Possible future investigations

Quantum Generative Adversarial Networks (qGANs) topic, has a really exciting future on its way and holds significant promise for advancing techniques. They are fast, efficient, and capable of making a lot of things. Think about having such a powerfull model that can produce generated data, so close to the real ones. It can be used in a way to "predict" the future, right? There are many things that we would love to predict, endless possibilities, and even if we do not manage to do so this study showed that qGANs can be integrated into other promising quantum algorithms. There is nothing left to do but to keep exploring and pushing the boundaries of what is possible.

6 Conclusion

This project successfully reviewed the principles of the "Quantum Generative Adversarial Networks for learning and loading random distributions" paper by Zoufal, Lucchi, Woerner and reproduced its core findings for loading distributions into quantum states. By implementing a hybrid qGAN with a custom variational quantum circuit as the generator and a classical neural network as the discriminator, we demonstrated that it is possible to train a quantum circuit to approximate a target probability distribution with high fidelity, as measured by low final values for relative entropy and the Kolmogorov-Smirnov statistic.

The final evaluation confirms the paper's significance in addressing the critical data-loading bottleneck in quantum computing. The qGAN approach offers a viable and resource-efficient alternative to exact loading methods, making a wider range of quantum algorithms more practical. The challenges encountered during reproduction, particularly concerning training stability and hyperparameter tuning, highlight the practical complexities of implementing hybrid quantum-classical machine learning models but also confirm the robustness of the underlying methodology once properly configured.

Future work based on this project could proceed in several exciting directions. Further research could explore the scalability of the approach by attempting to load more complex distributions requiring a larger number of qubits or some more qubit efficient algorithms and also by trying to fit qGANs in other cases. Finally, experimenting with different, more advanced variational ansaetze for the generator or alternative classical architectures for the discriminator could lead to even faster convergence and more accurate results.

7 Appendix

7.1 One-hot encoding example

Consider a random probability distribution: $X = \begin{bmatrix} \underbrace{0.1}_{0} & \underbrace{0.6}_{1} & \underbrace{0.2}_{2} & \underbrace{0.05}_{3} & \underbrace{0.05}_{4} \end{bmatrix}$ and batch size $= 10$.

As we can see we have 5 discrete values, so:

$$\textbf{\textit{sampled indices}}\ \text{(new vector)} = \begin{bmatrix} \underbrace{1}_{0} & \underbrace{1}_{1} & \underbrace{2}_{2} & \underbrace{0}_{3} & \underbrace{1}_{4} & \underbrace{3}_{5} & \underbrace{1}_{6} & \underbrace{2}_{7} & \underbrace{1}_{8} & \underbrace{4}_{9} \end{bmatrix}$$

This vector consists of the indexes of the X matrix, that occur as frequent as their probability tell us and it's length is equal to batch size. Then based on the index of ***sampled indices***:

$$\text{samples(index)} := \text{one-hot vector, where}$$

$$\text{one-hot vector(idx)} = \begin{cases} 1 & \text{,if sampled indices(index)} = idx \\ 0 & \text{,else} \end{cases}$$

while $idx \in \{0, 1, 2, 3, 4\}$.

Finally:

$$
\begin{aligned}
\text{samples}[0] &= \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix}\\
\text{samples}[1] &= \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix}\\
\text{samples}[2] &= \begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix}
\end{aligned}
$$

$$\vdots$$

$$\text{samples}[9] = \begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix}$$

7.2 Code snippets

```python
class Config:
    NUM_QUBITS = 3
    CIRCUIT_DEPTH = 3
    QC_WEIGHT_INIT_RANGE = 0.01 # Set after testing

    LOGNORMAL_MU = 1.0
    LOGNORMAL_SIGMA = 1.0
    DATA_RANGE_MIN = 0
    DATA_RANGE_MAX = 7

    BATCH_SIZE = 2000
    EPOCHS = 2000
    GEN_LR = 2e-4
    DISC_LR = 2e-4

    DISC_INPUT_SIZE = 2**NUM_QUBITS
    DISC_HIDDEN_1 = 50
    DISC_HIDDEN_2 = 20

    # Use GPU if available, otherwise use CPU
    DEVICE = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

config = Config()
print(f"Using device: {config.DEVICE}")
num_classes = 2**config.NUM_QUBITS
print(f"Number of classes/bins: {num_classes}")

```

Listing 1: Configuration setup.

```python
def initialize_state_custom(qubit_num):
    '''
    Initializes a quantum circuit with a uniform superposition
state.
    Args:
        qubit_num (int): Number of qubits in the quantum circuit.
    Returns:
        QuantumCircuit: A quantum circuit with all qubits in a
uniform superposition state.
    '''
    qc = QuantumCircuit(qubit_num)
    qc.h(qc.qubits)
    return qc

def create_variational_form(qubit_num, num_repetitions,
param_prefix="\theta"):
    '''
    Creates a custom variational form for a quantum circuit.
    Args:
        qubit_num (int): Number of qubits in the quantum circuit.
        num_repetitions (int): Number of repetitions for the entanglement and rotation layers.
        param_prefix (str): Prefix for the parameter names.
    Returns:
        tuple: A tuple containing the quantum circuit and a flat list of parameters.
    '''
    qc = initialize_state_custom(qubit_num)
    qc.barrier(label="Initial H")

    # Create parameters for the variational form
    theta_params = [[Parameter(f"{param_prefix}_{q}_{layer_idx}") for layer_idx in range(num_repetitions + 1)] for q in range(qubit_num)]

    for q_idx in range(qubit_num):
        qc.ry(theta_params[q_idx][0], q_idx)

    qc.barrier(label="Initial Ry")

    for rep_idx in range(num_repetitions):
        # Add entanglement layer with CZ gates
        for q_idx in range(qubit_num - 1):
            qc.cz(q_idx, q_idx + 1)

        # Add a CZ gate between the last qubit and the first qubit
        if qubit_num > qubit_num - 1:
            qc.cz(qubit_num - 1, 0)

        qc.barrier(label="CZ Layer")

        # Add rotation layer with Ry gates
        for q_idx in range(qubit_num):
            qc.ry(theta_params[q_idx][rep_idx+1], q_idx)

        qc.barrier(label="Ry Layer")

    flat_parameter_list = [p for sublist in theta_params for p in sublist]
    return qc, flat_parameter_list

```

Listing 2: Quantum circuit.

```python
def train(config, generator, discriminator, lognormal_distribution_tensor, history, criterion, optimizer_G, optimizer_D):
    '''
    Trains the qGAN model using the provided generator and discriminator.
    Args:
        config (Config): Configuration object containing hyperparameters and settings.
```

```
            generator (TorchConnector): The quantum generator model.
            discriminator (nn.Module): The classical discriminator
    model.
            lognormal_distribution_tensor (torch.Tensor): Tensor
    representing the target log-normal distribution.
            history (dict): Dictionary to store training history for
    logging and analysis.
            criterion (nn.Module): Loss function for training the
    discriminator.
            optimizer_G (torch.optim.Optimizer): Optimizer for the
    generator.
            optimizer_D (torch.optim.Optimizer): Optimizer for the
    discriminator.
        Returns:
            None
        This function performs the training loop, updating the
    generator and discriminator models.
        '''
        # Training loop
        for epoch in range(config.EPOCHS):

            # ====== STEP 1: Discriminator training with REAL data
    ======
            optimizer_D.zero_grad()
            real_data_samples_batch = create_one_hot_batch(
    lognormal_distribution_tensor.cpu().numpy(),
                                                                config.
    BATCH_SIZE, num_classes, config.DEVICE)
            labels_real_smoothed = torch.full((config.BATCH_SIZE, 1),
    1.0,device=config.DEVICE)
            pred_real = discriminator(real_data_samples_batch)
            loss_D_real = criterion(pred_real, labels_real_smoothed)

            # ====== STEP 2: Discriminator training with FAKE data
    ======
            with torch.no_grad():
                current_gen_probs_tensor_for_D = generator(torch.tensor
    ([]).to(config.DEVICE)).squeeze()

            # Generate fake data samples using the generator
            fake_data_samples_batch = create_one_hot_batch(
    current_gen_probs_tensor_for_D.cpu().numpy(),
                                                                config.
    BATCH_SIZE, num_classes, config.DEVICE)
            labels_fake_smoothed_for_D = torch.full((config.BATCH_SIZE,
     1), 0.0,device=config.DEVICE)
            pred_fake_detached = discriminator(fake_data_samples_batch)

            # No weights needed for the discriminator loss
            loss_D_fake = criterion(pred_fake_detached,
    labels_fake_smoothed_for_D)

            # Combine average value of loss and perform backpropagation
```

```python
        loss_D = (loss_D_real + loss_D_fake)/2
        loss_D.backward()
        optimizer_D.step()

        # ====== STEP 3: Generator training based on Discriminator's output ======
        optimizer_G.zero_grad()
        gen_probs_for_G_loss = generator(torch.tensor([]).to(config.DEVICE)).squeeze()
        d_eval_on_all_outcomes = discriminator(all_possible_outcomes_one_hot)
        loss_G = adversarial_loss(
            discriminator_output_on_all_samples=d_eval_on_all_outcomes,
            target_label_value=1,
            weights_distribution=gen_probs_for_G_loss
        )
        loss_G.backward()
        optimizer_G.step()

        # ====== STEP 4: Calculate Relative Entropy, Kolmogorv-Smirnov statistic and store process ======
        with torch.no_grad():
            current_gen_probs_tensor = generator(torch.tensor([]).to(config.DEVICE)).squeeze()
            current_gen_probs_np = current_gen_probs_tensor.cpu().numpy()

        # Calculate Relative Entropy and KS statistic
        kl_div, ks_stat = calculate_metrics_on_distributions(current_gen_probs_np, lognormal_distribution_tensor.cpu().numpy())

        # Store the losses and metrics in the history dictionary
        history['G_loss'].append(loss_G.item())
        history['D_loss'].append(loss_D.i   tem())
        history['KL_div'].append(kl_div)
        history['KS_stat'].append(ks_stat)

        # Print the progress every 100 epochs and in the last one
        if epoch % 100 == 0 or epoch == config.EPOCHS - 1:
            print(f"Epoch [{epoch}/{config.EPOCHS}] | Loss D: {loss_D.item():.4f} | Loss G: {loss_G.item():.4f} | Relative Entropy: {kl_div:.4f} | Kolmogorv-Smirnov statistic: {ks_stat:.4f}")

```

Listing 3: Training function.

7.3 Environment details

Component	**Version/Details**
Python	3.12.3
Qiskit	1.4.3
Qiskit Machine Learning	0.8.2
PyTorch	2.7.0
CUDA	12.6
Operating System	Ubuntu 24.04.2 LTS
Hardware	Intel Core i5-11th gen, iRISxe graphics (integrated), 32GB RAM
GPU support	Enabled (CUDA)

References

[1] Zoufal, C., Lucchi, A., & Woerner, S. (2019). *Quantum generative adversarial networks for learning and loading random distributions*. npj Quantum Information, 5(1), 103. Available: `https://www.nature.com/articles/s41534-019-0223-2`.

[2] David McMahon. (2007). *Quantum computing explained. John Wiley & Sons.*

[3] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). *Generative Adversarial Networks.* Advances in Neural Information Processing Systems, 27. Available: `https://arxiv.org/abs/1406.2661`

[4] Qiskit Development Team. (2023). *Qiskit: An Open-source Framework for Quantum Computing.* [Online]. Available: `https://qiskit.org`.

[5] Qiskit Development Team. *Qiskit: PyTorch qGAN Implementation.* Available: `https://qiskit.org/ecosystem/machine-learning/tutorials/04_torch_qgan.html`.

[6] D. Angelakis and team (2025). *Course's lecture notes*

[7] Medium (2024).

One Hot Encoding Explained. Available: `https://medium.com/@heyamit10/one-hot-encoding-explained-0b0130ccd78e`

[8] Quantum Amplitude Estimation paper (2000). *Quantum Amplitude Amplification and Estimation.* Available: `https://arxiv.org/pdf/quant-ph/0005055`

8 Supplementary Methods

8.1 Monte Carlo

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The validity of Monte Carlo methods is rooted in probability theory, specifically the Law of Large Numbers and the Central Limit Theorem.

- **Law of Large Numbers**: This theorem states that as the number of samples increases, the average of the results obtained will converge to the true expected value.
- **Central Limit Theorem and Error Estimation**: For a large number of samples, the distribution of the sample mean will be approximately normal. This allows us to quantify that Error estimation is $O(\frac{1}{\sqrt{N}})$

8.2 Black-Scholes model

Black-Scholes is a model used in finance to price option contracts. It gives a theoritical estimate of the price of European-style options only, because American could be exercised before expiration date. It makes certain assumptions:

- No dividends are paid out during the life of the option.
- Markets are random because market movements can't be predicted.
- There are no transaction costs in buying the option.
- The risk-free rate and volatility of the underlying asset are known and constant.
- The returns of the underlying asset are normally distributed.
- The option is European and can only be exercised at expiration.

and it uses the formula below to make its estimation:

$$C = SN(d_1) - K\exp^{-rt} N(d_2)$$

where $d_1 = \frac{\ln\frac{S}{K} + (r + \frac{\sigma_v^2}{2})t}{\sigma_s\sqrt{t}}$ and $d_2 = d_1 - \sigma_s\sqrt{t}$, also: C = Call option price, S = Current stock (or other underlying), K = Strike price, r = Risk-free interest rate, t = Time to maturity, N = A normal distribution.

8.3 Quantum Amplitude Estimation

Quantum Amplitude Estimation (QAE) is a fundamental quantum algorithm designed to estimate an unknown amplitude of a quantum state. This algorithm is built upon a more general subroutine known as Quantum Phase Estimation (QPE).

Consider a state prepared by a unitary operator A that acts on initial state $|\psi_0\rangle = |0\rangle^{\otimes(n+1)}$:

$$A|0\rangle^{\otimes(n+1)} = \sqrt{1-\alpha}|\psi_0\rangle^{\otimes n}|0\rangle + \sqrt{\alpha}|\psi_1\rangle^{\otimes n}|1\rangle$$

The goal is to estimate the value of the probability α. To do this, we use a unitary operator, Q, called Grover operator. The reason why it is used is because its eigenvalues are directly related to amplitude: $\lambda_{1,2} = \exp^{\pm 2i\theta_\alpha}$, where $\alpha = \sin^2(\theta_\alpha)$.

After that, we use Quantum Phase Estimation (QPE) to estimate the phase of Q operator. We can see that the estimation is going to be an approximation of $2\theta_\alpha$, for example $\phi = \frac{2\theta}{2\pi}$. In the end we can compute that $\theta_\alpha = \pi\phi$ and we find amplitude knowing that $\alpha = \sin^2(\theta_\alpha)$. The algorithm requires m additional evaluation qubits that control the applications of $Q = -AS_0A^\dagger S_{\psi_0}$, where $S_0 = I^{\otimes(n+1)} - 2|0\rangle\langle 0|^{\otimes(n+1)}$ and $S_{\psi_0} = I^{\otimes(n+1)} - 2|\psi_0\rangle\langle\psi_0| \otimes |0\rangle\langle 0|$

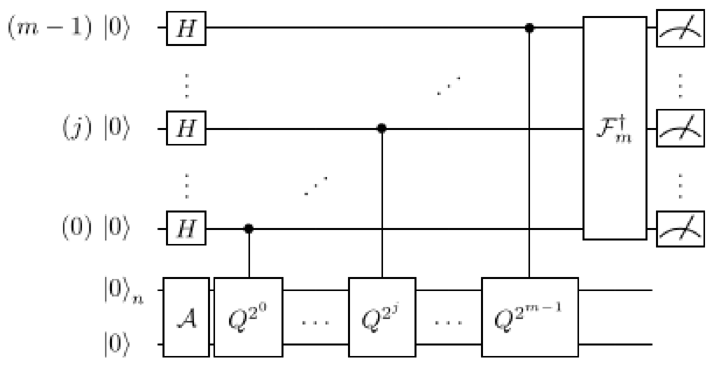


Figure 9: Quantum circuit for amplitude estimation with an inverse Quantum Fourier Transform denoted by $F_m^\dagger$

Inverse Quantum Fourier Transform (IQFT) is applied to the evaluation register, to transform the phase information into a computational basis state

The error in the outcome can be bounded by $\frac{\pi}{2^m}$. Considering that 2^m is the number of quantum samples used for the estimate evaluation, this error scaling is quadratically better than the classical Monte Carlo simulation.

QAE has various application fields (finance, chemistry, ML) and even the famous Grover algorithm can be seen as a specific instance of QAE.

8.4 QAE in the application

Instead of repeatedly measuring the state $|g_\theta\rangle$ on the quantum hardware (which would be a standard Monte Carlo approach), the researchers used a classical simulation of the Quantum Amplitude Estimation (QAE) algorithm to analyze it. The approach they choose contains m=8 evaluation qubits, so a $\frac{\pi}{256}$ error rate.

To use QAE for the pricing of European options, a suitable oracle A must be constructed. First, the uncertainty distribution that represents the spot price S_T of the underlying asset at the option's maturity T is loaded into a quantum state $\sum_j \sqrt{p_j}|j\rangle$. After that, the payoff function, $\max\{0, S_T - K\}$, must be encoded and implemented. To do it an ancilla qubit $|0\rangle$ is added and with the help of a comparator circuit it this ancilla is flipped to $|1\rangle$, only if the spot price j is greater than the strike price K.

So the state becomes:

$$\underbrace{\sum_{j=0}^{K} \sqrt{p_j}|j\rangle|0\rangle}_{\text{Payoff is } 0} + \underbrace{\sum_{j=K+1}^{2^n-1} \sqrt{p_j}|j\rangle|1\rangle}_{\text{Payoff is } > 0}$$

, since $|j\rangle|0\rangle \rightarrow \begin{cases} |j\rangle|0\rangle & \text{,if } i \leq K \\ |j\rangle|1\rangle & \text{,if } i > K \end{cases}$

After that another ancilla qubit is introduced to map the payoff to an amplitude.

$$A|0\rangle^{\otimes(n+1)} = \sum_{j=0}^{K} \sqrt{p_j}|j\rangle|0\rangle|0\rangle + \sum_{j=K+1}^{2^n-1} \sqrt{p_j}|j\rangle|1\rangle \left(\sqrt{1-f(i)}|0\rangle + \sqrt{f(i)|1\rangle}\right)$$

with $f(i) = \frac{j-K}{2^n-K-1}$.

Eventually the probability of the last ancilla to be $|1\rangle$ can be measured:

$$P(|1\rangle) = \sum_{j=K+1}^{2^n-1} \frac{p_j(j-K)}{2^n-K-1}$$

where $2^n - K - 1$ is the normalization constant for the probability to be inside $[0, 1]$, $\mathbb{E}[\max\{0, S_T - K\}] = \sum_{j=K+1}^{2^n-1} p_j(j-K)$ and $P|1\rangle = \alpha$.

Hence,

$$P(|1\rangle) = \frac{\mathbb{E}\max\{0, S_T - K\}}{2^n - K - 1} \Leftrightarrow \mathbb{E}[\max\{0, S_T - K\}] = a(2^n - K - 1)$$