Efficient encoding of a probability distribution

How do we overcome the probability loading problem in Quantum Monte Carlo?

Learning objectives of the challenge

The aim of this challenge is to study how to efficiently encode probability distributions. It will specifically focus in two methodologies: *Tensor Networks (TNs)* and *Quantum Generative Adversarial Networks (qGAN)*. The following are the learning objectives of the challenge:

1. Understand the complexity of uploading probability distributions into a quantum circuit
2. Why is this an important step? Which algorithms rely on this step to be done efficiently?
3. What would happen if we are not able to efficiently upload probability distributions into quantum circuits? What algorithms would not provide an advantage and what applications would not benefit from quantum computing?
4. What are the main bottlenecks to efficiently implement probability distributions into quantum circuits? What are the proposals in the literature to overcome them? Are there any fundamental blockers for it to be possible?
5. Understand how to encode a probability distribution and use Tensor Networks (TNs) and Quantum Generative Adversarial Networks (qGAN) to encode the given probability distribution.
6. Think about resource estimation with the most promising techniques, do they increase a lot the circuit depth?

The challenge

Introduction

In finance, a notable application involves using Quantum Amplitude Estimation to speed up Monte Carlo computations, commonly known as Quantum Monte Carlo (QMC). Monte Carlo (MC) methods are extensively utilized to solve problems involving uncertainty, random processes, or high-dimensional integrals. Due to their versatility, MC methods are crucial in both theoretical and applied research across various disciplines. Despite their power, classical MC methods face limitations in computational efficiency, especially with high-dimensional problems. QMC offers a quadratic speed-up over classical MC methods, potentially revolutionizing computational performance in financial modeling.

Although classical MC methods are powerful, they often encounter computational inefficiencies, particularly in high-dimensional problems. Quantum Monte Carlo promises a quadratic speed-up over classical methods, offering an opportunity to enhance computational performance. However, this theoretical speed-up has recently been questioned. The concern is that the speed-up is typically measured in terms of query complexity rather than overall computational complexity, and these are not necessarily equivalent. Querying a quantum computer involves significant overheads absent in classical computations, such as state preparation and error correction. Considering these additional operations, the actual computational advantage may be significantly reduced or even negated.

A significant bottleneck in QMC methods is state preparation, specifically the probability loading problem, which involves translating probability distributions into quantum states. This task is particularly challenging due to its poor scalability and the complexity of its computational steps. The Grover-Rudolph method, commonly used for this purpose, requires a series of computational steps that become increasingly complex as the precision of the state preparation increases. This preparation process is not only time-consuming but also prone to errors, often undermining the claimed advantages of QMC.

Various approaches can be found in the literature to address this problem.

Source: Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks

Section 1: Problem statement - probability distribution

The probability we want to encode is an n-dimensional multivariate normal distribution.

Consider an n-dimensional random vector $\mathbf{X} = (X_1, \dots, X_n)^T$.

- The multivariate normal distribution can be written as $\mathbf{X} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$, with μ the mean vector defined by $\mu = E[\mathbf{X}]$.
- $\mathbf{\Sigma}$ is the covariance matrix, giving the covariance between all pairs of the random vector $\mathbf{X}$, $\Sigma_{i,j} = E[(X_i - \mu_i)(X_j - \mu_j)]$ for all i, j.

If the covariance matrix Σ is positive definite, the distribution has the following **probability density function**:

$$f(\mathbf{x}, \mu, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right)$$

with $\mathbf{x}$ is an n-dimensional real vector. $|\mathbf{\Sigma}| \equiv \det\mathbf{\Sigma}$ the determinant of $\mathbf{\Sigma}$.

The data for the mean vector and the covariance matrix for the one, two and four dimensional cases are given in a different file.

Section 2: Tensor Networks (TNs)

Tensor Networks is widely used in quantum many-body physics. It is a mathematical framework and a numerical tool used to compress high-dimensional datasets by using the structure of their correlations.

Tensor Networks

Before starting with the challenge, let us first introduce Tensor Networks. Tensors can be thought as multi-dimensional arrays. We usually represent a tensor with a shape (we can choose the shape to be anything we want, for example a circle). Its indices are represented by lines. Lines that connect tensors imply summations over the index. One could see an example of a scalar, a vector, a matrix and a rank-3 tensor in the **picture** below:

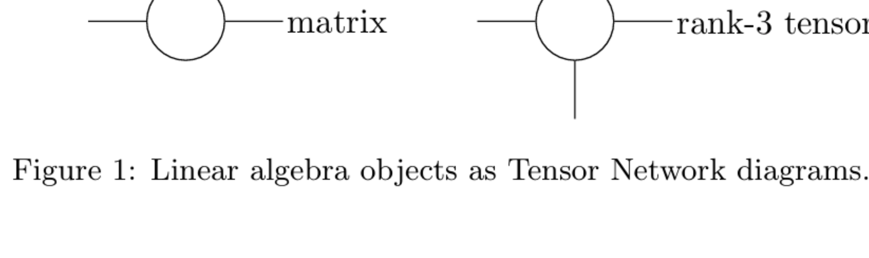


Figure 1: Linear algebra objects as Tensor Network diagrams.

Tensor-train cross approximation (TT-cross)

Tensor-train cross approximation for multidimensional arrays is explained in detail in this TT-cross paper. One does not need to know the technical details of the TT-cross algorithm for this challenge. By following that paper, as well as the paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks we summarize the key intakes.

The curse of dimensionality: A tensor with d indices (rank-d) has d dimensions (is a d-dimensional array). Directly numerically handling arrays with many dimensions (called **multidimensional arrays**) is not feasible due to the so-called 'curse of dimensionality'. This term refers to the exponential increase in memory needed to store an array with d indices, and the amount of operations needed to perform basic operations with such an array, as the dimensionality d increases.

The aim of TT-cross algorithm: This algorithm can accept **any** tensor. Its aim is to determine a suitable approximation of this tensor, approximating it to another (new) tensor with smaller number of parameters. Essentially, The TT-cross approximation algorithm is a technique used to approximate high-dimensional tensors in a low-dimensional format, known as Tensor Train (TT) format. The question then is, how do we accomplish this, and what tensor representation would be appropriate for the task?

The essence of the TT-cross algorithm: Consider the function f. The aim is to build a d-dimensional tensor in a Tensor Train format. For this, we sample whatever values we have within a chosen domain $\Omega \subset \mathbb{R}^d$ (but we cannot afford to sample the whole domain $\mathbb{R}^d$, since it grows exponentially with the number of dimensions d).

In more detail: Consider the tensor A, which is a d-dimensional array that represents the function f. It can be approximated as a TT-format as follows:

$$A(i_1, i_2, \dots, i_d) \approx \sum_{r_0, r_1, \dots, r_d} G_1(r_0, i_1, r_1) G_2(r_1, i_2, r_2) \cdots G_d(r_{d-1}, i_d, r_d)$$

Each tensor G_k is a 3-dimensional tensor that is called a **TT core**. There are d in number. r_k are the ranks (also known as the **virtual dimensions**). Note that $r_0 = r_d = 1$.

Consider the cores G_k, reshaped to be 2-dimensional tensors (matrices). The algorithm starts with an initial guess for the tensor cores G_k, and a guess for the index sets of the rows and the columns (here I_k and J_k correspondingly). These index sets, at each process of the iteration, target submatrices with near-maximal volume (volume = the absolute value of the determinant of a matrix). Then one computes the series of the tensor cores G_k over the selected entries. For this, one would need to identify the optimal low-rank approximation (with the use of singular value decomposition (SVD), for example). The final step is to assemble the tensor with the cores obtained in the previous step (which only have entries that correspond to I_k and J_k). The process continues until a stopping criterion is met, usually a predefined tolerance for the approximation error. The result is an approximation of the original tensor, one that can be manipulated much more efficiently. You could read more in page 5 of the paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks.

In this section we will closely follow the paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks to use Tensor Networks and the **tensor-train cross approximation (TT-cross)** algorithm to overcome the probability loading problem. The steps you will need to follow will be described in detail below.

For this challenge we recommend using jupyter notebook (see installation instructions on how to install). If you can not get this to work, another choice would be to use Google colab.

STEP 1 -- Construct the Tensor Network

For this step you will need to install tntorch, NumPy and PyTorch.

The aim of this step is to construct the *approximate* Tensor Network of the probability distribution defined in the **Problem Statement**. Before constructing the Tensor Network one would need to follow the **Quantization** process to map a continuous spectrum of values to a discrete one. After following this process, a continuous distribution $p(x)$ can be approximated by a discrete probability vector:

$$\mathbf{p} = [p(x_0), p(x_1), \dots, p(x_m)]$$

with m the number of discretized points. Here we assume that $m = 2^d - 1$. We also require the vector $\mathbf{p}$ to be normalized.

Therefore, these m points, $\{x_i\}_{i=0}^m$ can be encoded **on** $d = log_2(m+1)$ **qubits**.

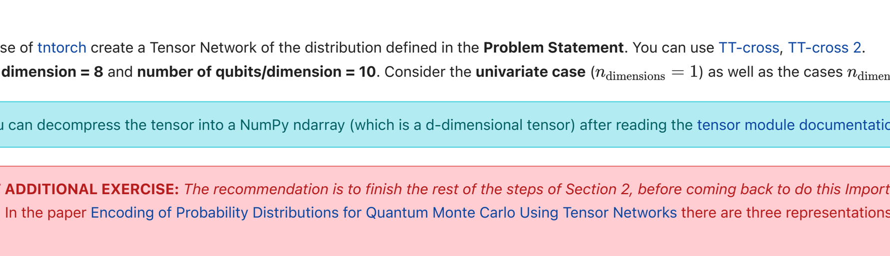
The corresponding state then would be:

$$|\psi\rangle = \sum_{i=0}^{2^d-1} \sqrt{p(x_i)} |(i)_2\rangle$$

with $(i)_2$ the index i in base 2:

$$(i)_2 = (i_1, \dots, i_d) = \sum_{k=1}^{d} i_k 2^{d-k}$$

The vector $\mathbf{p}$ of size 2^d can be written as a tensor $A \in \mathbb{R}^{2\times\cdots\times 2}$, by writting each $p(x_i)$ in a Tensor Network format, as shown by the figure:



Note that the *arrangement* of the tensors is particularly important when dealing with more than one dimensions (like in our case, where we have a multivariate distribution). This is because, for the 1D case, the only thing that matters is how many qubits one uses (that defines the granularity). However, with higher dimensions it is not trivial to see how to arrange the tensors to create a (one-dimensional) Tensor Train.

TASK:

- With the use of tntorch create a Tensor Network of the distribution defined in the **Problem Statement**. You can use TT-cross, TT-cross 2.
- Use **bond dimension = 8** and **number of qubits/dimension = 10**. Consider the **univariate case** ($n_{\text{dimensions}} = 1$) as well as the cases $n_{\text{dimensions}} = 2$ and $n_{\text{dimensions}} = 4$.

Note that you can decompress the tensor into a NumPy ndarray (which is a d-dimensional tensor) after reading the tensor module documentation of tntorch. You will be needing this for the exercises below.

IMPORTANT ADDITIONAL EXERCISE: *The recommendation is to finish the rest of the steps of Section 2, before coming back to do this Important Exercise.* As discussed above, the arrangement of the tensors is particularly important when dealing with multivariate distributions. In the paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks there are three representations presented: Sequential, Mirroring and Interleaving. Choose one (or more!) of them, justify your choice and create the Tensor Network.

Answer:

In [ ]:

STEP 2 -- Construct the circuit from the Tensor Network

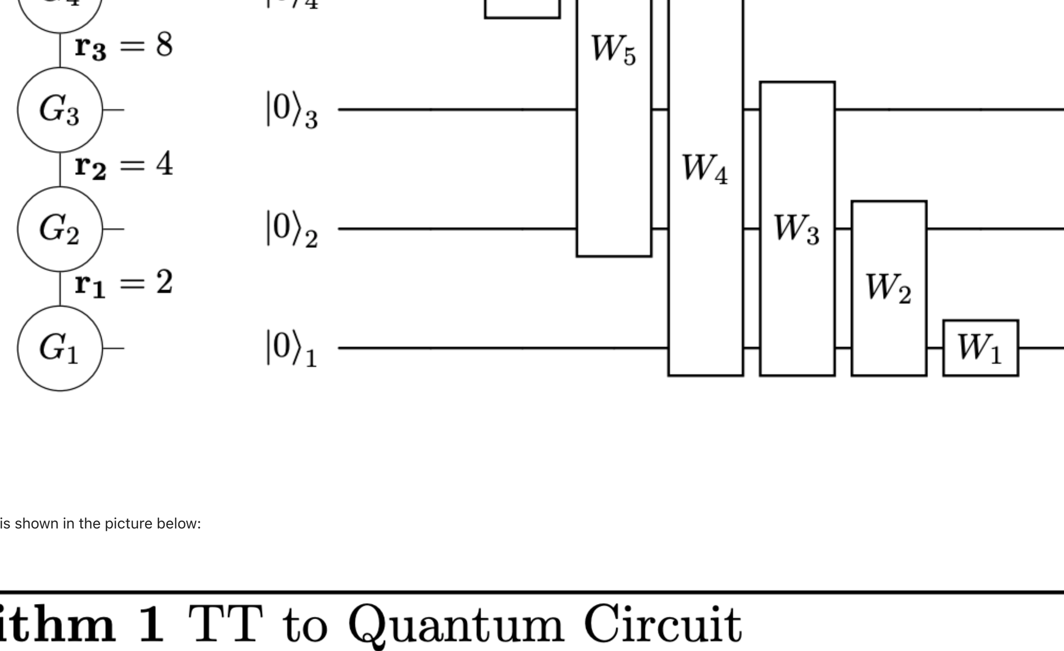
The aim of this step is to decompose the tensor $A(i_1, \dots, i_d)$, written in the following tensor network form:

$$A(i_1, \dots, i_d) = \sum_{r_0, \dots, r_d} G_1(r_0, i_1, r_1) G_2(r_1, i_2, r_2) \cdots G_d(r_{d-1}, i_d, r_d)$$

In order to construct a circuit, a set of unitary operators $\{W_i\}$ that represent quantum gates. Applying these gates into the initial state, we get the target state:

$$|\psi\rangle = W_1 W_2 \cdots W_d |0\rangle^{\otimes d}$$

The equivalence looks like this:



The algorithm you need to develop is shown in the picture below:

Algorithm 1 TT to Quantum Circuit

Input $G = [G_1, \dots, G_d]$ ▷ List of TT-cores
Output $W = [W_1, \dots, W_d]$ ▷ List unitary operators
$W \leftarrow []$
for $i \leftarrow 1$ to length(G) **do**
  $M \leftarrow$ reshape($G[i]$)
  $U, \Sigma, V^* \leftarrow$ **SVD**(M)
  $\Sigma \leftarrow$ **truncate**(Σ)
  $R \leftarrow (\Sigma V^*) \otimes I_2$
  $G[i+1] \leftarrow RG[i+1]$
  W.insert($G[i]$)
end for
return W

One would need to read the instructions in pages 8-10 of paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks to understand how to implement this algorithm. For completion, we write the steps below.

The aim is to find the unitary opeerators of the circuit above (operators W_k). We will do this sequentially, handling one tensor G_k at a time.

$\mathbf{k = 1}$:

- First reshape the tensor $M_1(i_1, r_1) = G_1(r_0 = 0, i_1, r_1)$ into a $\mathbb{C}^{2\times n_1}$ matrix. Note here that I already substitute the value of $r_0 = 0$.
- We factor the matrix M_1 with the use of the SVD theorem as following: $M_1 = U_1 \Sigma_1 V_1^*$, where $U_1 \in \mathbb{C}^{2\times 2}$ and $V_1^* \in \mathbb{C}^{n_1 \times n_1}$ are unitary matrices. $\Sigma_1 \in \mathbb{C}^{2\times n_1}$ is a diagonal matrix that contains the singular values of M_1.
- We substitute the above in $A(i_1, \dots, i_d)$:

$$\sum_{j=0,1} U_1(i_1, j) \Big[\sum_{r_2, \dots, r_d} \tilde{G}_2(j, i_2, r_2) \cdots G_d(r_{d-1}, i_d, r_d) \Big]$$

  with $\tilde{G}_2(j, i_2, r_2)$ **containing the rest matrices that come from the SVD**: $\tilde{G}_2(j, i_2, r_2) = \sum_{r_1} (\Sigma_1 V_1^*)(j, r_1) G_2(r_1, i_2, r_2)$. For now on we drop the tilde and write $\tilde{G}_2 = G_2$.
- We set W_1 to be **the unitary operator U_1 that comes from the SVD. This is our first unitary operator of the quantum circuit!** W_1 is acting only on the first qubit. It can be seen as the action of the unitary operation on $\mathbb{C}^2 \otimes \mathbb{C}^{2^{d-1}}$:

$$|0\rangle \otimes |v\rangle \to U_1(0,0)|0\rangle \otimes |v\rangle + U_1(1,0)|1\rangle \otimes |v\rangle$$
$$|1\rangle \otimes |v\rangle \to U_1(0,1)|0\rangle \otimes |v\rangle + U_1(1,1)|1\rangle \otimes |v\rangle$$

$\mathbf{k = 2}$:

- Now we turn our attention to the tensor in the brackets, which we name $A^{(2)}$:

$$A^{(2)}(j, i_2, \dots, i_d) = \sum_{r_2, \dots, r_d} G_2(j, i_2, r_2) \cdots G_d(r_{d-1}, i_d, r_d)$$

- The index j can be reduced by **truncating** the singular values of the matrix Σ: the rows of $\Sigma(j, :)$ are null if $j \geq \min(2, n_1)$. We use the symbol $a \wedge b$ for $\min(a, b)$ below. **Do not reduce the singular values here.**
- Assume that the bond dimensions are $n_2 = 2^{l_2}$.

$\mathbf{k \geq 2}$:

- We iterate by defining $A^{(k)}$:

$$A^{(k)}(j, i_k, \dots, i_d) = \sum_{r_k, \dots, r_d} G_k(j, i_k, r_k) \cdots G_d(r_{d-1}, i_d, r_d)$$

  with $j = 0, \dots, 2^{(k-1)\wedge l_{k-1}} - 1$.
- Reshape the tensor G_k into a matrix $M_k \in \mathbb{C}^{m_k \times n_k}$ with $m_k = 2(2^{(k-1)\wedge l_{k-1}}) = 2^{k \wedge (1 + l_{k-1})}$ and n_k the bond dimension.
- As before, we use SVD to decompose $M_k = U_k \Sigma_k V_k^*$.
- U_k is the kth unitary operator in the circuit, which we name W_k. Note that since the series of M_k are $m_k = 2^{k \wedge (1 + l_{k-1})}$ (which is the same as the series of W_k), the unitary operator W_k acts on the qubits $k, \dots, k - k \wedge l_{k-1}$.
- $\Sigma_k V_k^*$ is then passed to G_{k+1} and the process continues..

- The core tensors $\{G_i\}$ would need to be reshaped before the singular value decomposition (SVD). One could do that with numpy.reshape.
- If you want to permute the indices of tensors you can use numpy.transpose.
- One could read more on tensor networks here: The density-matrix renormalization group in the age of matrix product states . This could be needed specifically if you want to understand SVD. SVD is mentioned in page 15.

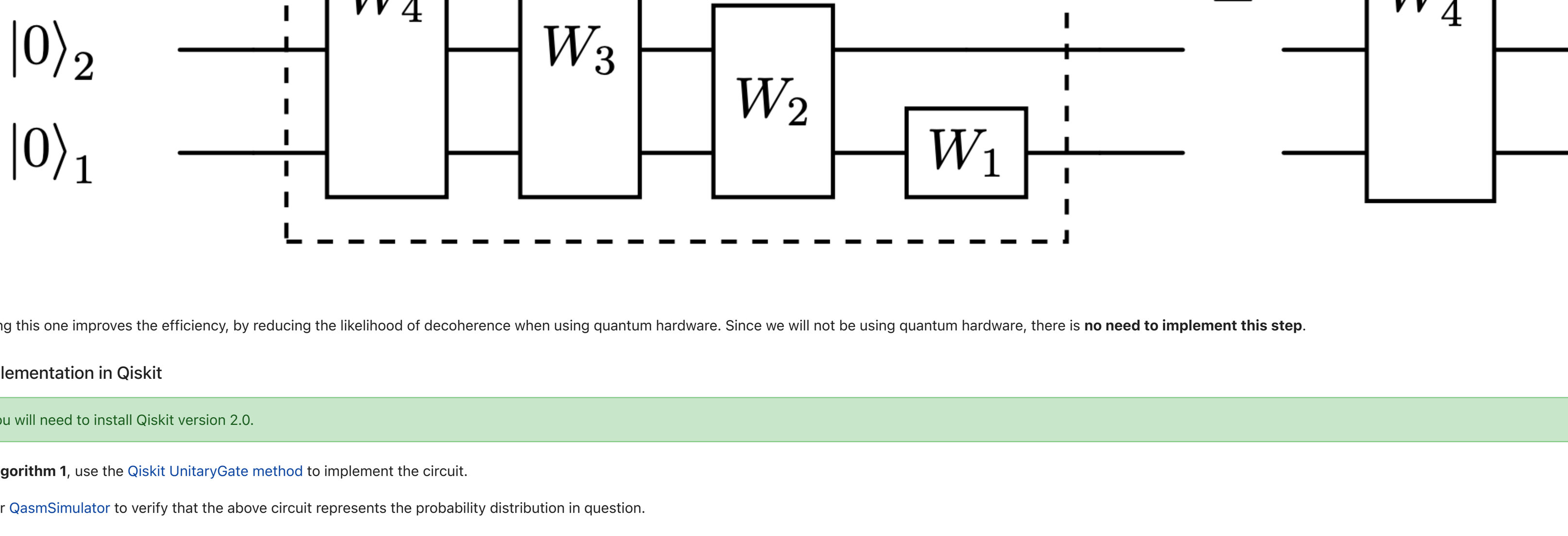
Answer:

In [ ]:

STEP 3 -- Reduce the depth of the circuit

This step is for your interest only - there is no need to implement this.

One can merge the tensors W_i by summing over their virtual indices in order to create a quantum circuit with less depth. In the case of four qubits this is shown below:



Note that by doing this one improves the efficiency, by reducing the likelihood of decoherence when using quantum hardware. Since we will not be using quantum hardware, there is **no need to implement this step**.

STEP 4 -- Implementation in Qiskit

For this step you will need to install Qiskit version 2.0.

After applying **Algorithm 1**, use the Qiskit UnitaryGate method to implement the circuit.

Use the simulator QasmSimulator to verify that the above circuit represents the probability distribution in question.

Answer:

In [ ]:

STEP 5 -- Comparing with the target probability distribution

In this section you will calculate the metrics mentioned in Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks with the aim to evaluate the quality of the results. For this you will calculate a distance that measures the difference between cumulative distribution functions of probability distributions, the circuit depth as well as the time.

Use the Tensor Network create in **STEP 1** to compute the metrics below for the following cases:

- Univariate case ($n_{\text{dimensions}} = 1$)
- $n_{\text{dimensions}} = 2, 4$

Kolmogorov-Simrnov distance

The Kolmogorov-Simrnov distance is used to measure the maximum difference between two cumulative distribution functions of two probability distributions. It is defined as:

$$D = \sup_x |F(x) - G(x)|$$

with $F(x)$ and $G(x)$ two **cumulative distribution functions**, $\sup_x$ the supremum over all values of x. Note that the supremum is a concept used in real analysis and order theory to describe the smallest value that is greater than or equal to every element in a given set. $F(x)$ is the **exact distribution** and $G(x)$ the one approximated with Tensor Networks. As the **exact distribution**, use the numerical value of the probability distribution presented in **Section 1**.

Cumulative distribution: For a random variable (X), the cumulative distribution function $(F(x))$ is defined as:

$F(x) = P(X \leq x)$

This means $F(x)$ gives the probability that the random variable X is less than or equal to x.

Plot the **Kolmogorov-Simrnov distance = f(number of qubits)** for different bond dimensions of the Tensor Network between **the TT-cross approximation** and **the exact**, as indicated in the left part of figure 4 in paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks. What are your observations?

Additionally, one can also plot the **Kullback-Leibler (KL) Divergence = f(number of qubits)**. See page 15 of Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks.

Circuit Depth

Plot the **circuit depth = f(number of qubits)**. What are your observations?

Training time

Plot the **time = f(number of qubits)**. Here consider different times (the time to create the Tensor Network, the time to transform that to a circuit). What are your observations?

Answer:

In [ ]:

Note: the images of this section were taken from the paper Encoding of Probability Distributions for Quantum Monte Carlo Using Tensor Networks.

BONUS - Section 3: Quantum Generative Adversarial Networks (qGAN)

For this step, if you choose the Pennylane road, you will need to install Pennylane and PyTorch. You might need to install more libraries for this road. If you choose the Qiskit road you need to install qiskit-machine-learning=0.7.1, qiskit=0.45, qiskit_algorithms=0.2.2, torch, matplotlib, pylatexenc.

For this section, you can use one of the following:

- **Qiskit:** Follow the tutorial of Qiskit Qiskit qGAN tutorial which introduces qGAN to learn the probability distribution. The final step of the tutorial calculates the cumulative distribution functions of the generated distribution as well as the difference between the original and the generated distribution.
- **Pennylane:** Instead of implementing qGAN with Qiskit, use PennyLane. One could have a look at these tutorials: Tutorial 1, Tutorial 2 and Tutorial 3. As you can see, there is no tutorial with Pennylane that uses qGAN to learn probability distributions. Therefore if one chooses this path the data loader should be adjusted.

Answer:

In [ ]:

BONUS - Section 4: Calculating metrics and comparison: TNs VS qGAN

Decide what metrics to use to compare the methods **TT-cross approximation** and the **qGAN**. **STEP 5** from **Section 2** can give you some ideas. Why did you choose these metrics?

How do **TT-cross approximation** and the **qGAN** compare?

You can answer the last question even if you did not complete both TNs and qGAN sections (search in the literature)!

Answer:

In [ ]: