Machine Learning Quantum Many-Body Systems Using Neural Autoregressive Distribution Estimation and GPU Acceleration

Authors: Caleb Sanders[*], Ignacio Varela,[*] Zhexuan Gong[†], and Alex Lidiak[‡]

13 May 2021

Abstract

Much research has been dedicated to developing accurate approximations of large many-body quantum mechanical systems and their ground states. As the number of qubits in a system increases, the number of possible spin states increases exponentially. Thus, it becomes infeasible to classically solve these systems for large N. In this report, we introduce a machine learning algorithm that solves for the ground state energy of an arbitrarily large many-body system by using a neural network as an ansatz to approximate the many-body wavefunction. We implement Neural Autoregressive Density Estimation (NADE) to draw from a distribution of 2^N possible states and perform accelerated optimization by running our ground state search on a graphical processing unit (GPU). Finally, we discuss testing results, possible improvements and potential future areas of research.

[*]Undergraduate Engineering Physics, Colorado School of Mines
[†]Professor, Colorado School of Mines
[‡]Graduate Physics, Colorado School of Mines

1 Introduction

Accurately modeling and solving entangled many-body systems is a very useful and challenging endeavor in quantum mechanics. Specifically, obtaining the ground state energy of a quantum many-body system has various applications in condensed matter, chemistry, nuclear, and other areas of physics [1]. However, a persistent problem is dealing with the exponentially increasing dimensionality of these many-body systems. The number of possible spin states in a system of N qubits increases as 2^N with its wavefunction given by

$$|\Psi\rangle = \sum_{s}^{2^N} \psi_s |s\rangle , \tag{1}$$

where $|\Psi\rangle$ is a linear superposition of basis states $|s\rangle = |s_1, s_2, ..., s_N\rangle$ weighted by ψ_s. As N grows, writing the many-body wavefunction becomes exponentially inefficient. Similarly, the dimensionality of both the Hamiltonian matrix and the computational cost required to compute it's eigenenergies also increases as 2^N, making these calculations very difficult for large N. Thus, with large systems, we introduce the need for an approximation of their wavefunction, also known as an "ansatz". Ideally, this compact representation of the wavefunction should have a number of parameters that scales polynomially rather than exponentially. In many cases, it is possible to obtain such an approximation because the the probability distribution of states in physical many-body systems only occupies a small region of the full Hilbert space.

Over the years, many ansatze have been developed (mean-field, Jastrow, matrix product state, etc.). While these ansatze approximate the many-body wavefunction with fewer parameters, they have limitations in their ability to simultaneously model high entanglement and variational freedom [1]. However, neural networks have recently shown great promise in representing entangled many-body systems and can be efficiently scaled to higher dimensions. In the neural network approach, a network is established with an input layer of N nodes, a hidden layer, and an output layer representing the wavefunction coefficient $\psi(s)$ for a given state s. If the hidden layer is constructed with a reasonable number of nodes (on the order of N), then the number of parameters (corresponding to the network weights and biases) scales polynomially. Furthermore, this approach introduces nonlinear mapping between inputs and outputs, thus allowing entanglement correlations to be encoded into the network.

Finally, the neural network ansatz can be integrated alongside a parallelized sample generation algorithm, Neural Autoregressive Density Estimation (NADE), to provide an accurate approximated snapshot of the full probability distribution of 2^N states. Then, modern machine learning tools such as backpropagation, gradient descent, and GPU acceleration introduce the potential for a fully integrated system that efficiently optimizes a network to the ground state energy of an arbitrarily large many-body system. In this report we investigate the use of a feed forward neural network (FFNN) as an ansatz and introduce a proof of concept optimization system.

2 Feed Forward Neural Networks

A FFNN is one of the most common networks used in machine learning. Figure 1 shows an example diagram of a simple FFNN with an input layer, a hidden layer, and an output layer (similar to the architecture used in our project).

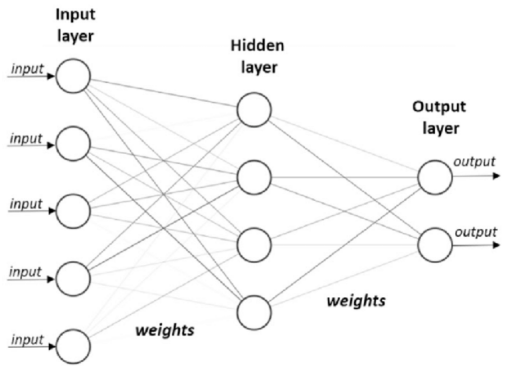


Figure 1: Example FFNN architecture

Connections are only present between adjacent layers, with no intra-layer connections. Layers in a FFNN are fully connected, which means the information passing through each node is fed through all nodes in the subsequent layer. Each node and connection is represented by a number value, the range of which can be tuned to a desired application. The numerical value of a given node, h_{ki}, can be expressed as a function of the nodes feeding into it and the connection values by which they are linked, known as weights. During a forward pass, nodes from previous layers are multiplied by their corresponding weights and summed, making each new node a linear combination of nodes from previous layers. A bias vector b of length equal to that of h_k is commonly added after this weighted sum. Finally, an activation function σ is included on each layer to introduce non-linearity to the system and control the range of possible values for each node (we used Tanh due to its range of (-1,1)). Equation 2 shows the output function for a given node i and layer k where W is the weight vector and s is the input vector.

$$h_{ki} = \sigma(W_k \bullet s + b_{ki}) \tag{2}$$

For our purposes, the input $|s\rangle$ is a simple 1D vector of binary values corresponding to the state of each bit in the system. The output vector is of length d, where d is the number of possible discrete quantum values for each qubit (1 and -1). In the FFNN representation, a system's full wavefunction is formally given by

$$\ln[\Psi(s)] = \sigma(W_k \sigma(W_{k-1}(...\sigma(W_1 s)))) \tag{3}$$

where k is the network depth (number of hidden layers) and s is the input state [1]. Note, in practice, the output of the network is represented by $\ln[\Psi(s)]$ rather than $\Psi(s)$ for a more efficient energy gradient calculation (see Section 4.2).

3 Neural Autoregressive Density Estimation

To optimize a network, the gradient of a cost function (energy) must be computed to shift the network's parameters towards a minimum. For large N, computing this energy gradient across all possible spin states is computationally impossible, thus requiring a stochastic sampling method to first draw from a distribution of 2^N states. NADE is an exact probability density estimator that naturally lends itself to parallelization, as sampled states can be generated independent of each other. In comparison to Markov-chain Monte Carlo (MCMC) sampling, NADE is much more computationally efficient and also produces broader spin configuration samples of the full Hilbert space [1]. When generating samples from a distribution, we first commit to representing the probability of sampling a given state $P(s)$ as a product of conditional probabilities [4],

$$P(s_1, ..., s_N) = \Pi_{i=1}^{N} p_i(s_i|s_{i-1}, ..., s_1). \tag{4}$$

The autoregressive nature of this sampling procedure allows us to decompose the probability of obtaining a bit at site i into the product of conditional probabilities for all bits prior to site i. Furthermore, the output of our network is denoted by

$$\hat{v}_i = (v_{i,s_1}, v_{i,s_2}, ..., v_{i,s_M}) \tag{5}$$

where $\hat{v}_i$ is an un-normalized probability distribution for s_i to take one of M discrete possible quantum values [1]. For our purposes, we are modeling a system of spin-$\frac{1}{2}$ particles, resulting in only two possible values for $\hat{v}_i$, $v_{i,pos}$ and $v_{i,neg}$. These values correspond to the probabilities of sampling positive and negative bits. While $\hat{v}_i$ can be expanded to include both real and complex values (as our collaborator Alex Lidiak has shown), we have chosen to only include real values for simplicity.

For the autoregressive property to be properly enforced, each v_{i,s_i} must only depend on spins $s_1, ..., s_{i-1}$. This condition can be easily met by applying a mask to the input before passing it through the network. To generate $\hat{v}_i$ for a bit at site i, values $s_1, ..., s_{i-1}$ of the mask are set to 1 while values $s_i, ..., s_N$ are set to 0. An important characteristic of this sampling method is that, to sample s_1, a fully masked state (state of all zeros) must be passed through the network. Thus, each sampled bit only depends on the network's parameters at the time it is sampled. Once $\hat{v}_i$ is generated for a given bit, each entry in the distribution is normalized according to the l_1 norm,

$$p_i(s_i|s_{i-1}, ..., s_1) = \frac{exp(v_{i,s_i})}{\sum_{s'} |exp(v_{i,s'})|}. \tag{6}$$

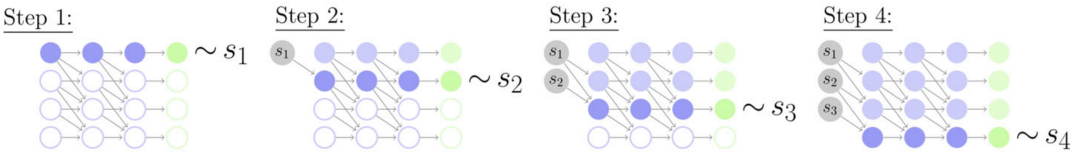


Figure 2: NADE algorithm generating a sampled state of 4 qubits using autoregressive masking. Empty nodes represent masked nodes, filled but faded nodes represent cached results from previous steps, fully filled nodes represent activated nodes to sample s_i. Generating a state of N sampled bits requires N forward passes through the algorithm. Diagram obtained from [1].

Similarly, NADE can be used to produce conditional wavefunctions in a quantum system, which we will refer to as QNADE. In the QNADE representation, a state's wavefunction coefficient $\psi(s)$ is given by

$$\psi(s) = \psi(s_1, ..., s_N) = \prod_{i=1}^{N} \psi_i(s_i|s_{i-1}, ..., s_1). \tag{7}$$

The only difference between conditional probabilities and conditional wavefunctions is, wavefunction coefficients are normalized according to the l_2 norm,

$$\psi_i(s_i|s_{i-1}, ..., s_1) = v_{i,s_i} \equiv \frac{exp(v_{i,s_i})}{\sqrt{\sum_{s'} |exp(v_{i,s'})|^2}}, \tag{8}$$

since the probability of obtaining a given state is defined by the Born rule, $P(s) = |\psi(s)|^2$. Once $\hat{v}_i$ has been obtained for a given state, v^2_{i,s_1} is denoted as the probability of sampling a positive bit while v^2_{i,s_2} is denoted as the probability of sampling a negative bit. The positive-bit probability can then be used to generate a sampled bit using the Bernoulli method. Figure 3 displays an example distribution generated using QNADE sampling for a system of 5 qubits. Note, the sampled distribution provides a more accurate representation of the actual distribution as the number of samples increases. Ultimately, our optimization procedure trains a neural network such that it learns to sample the ground state of a given Hamiltonian using the QNADE algorithm.

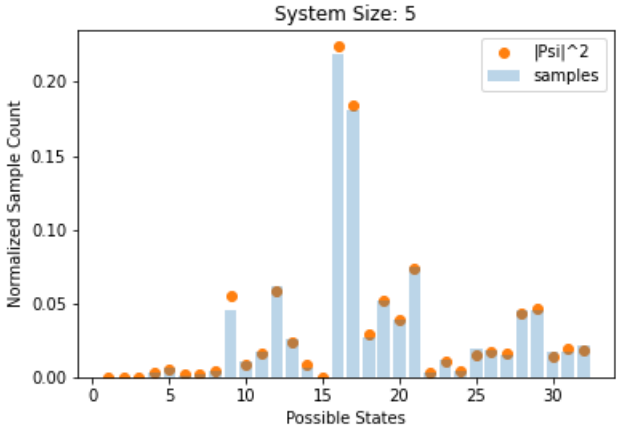


Figure 3: 2500 QNADE samples for a system of 5 qubits. Normalized sample count is plotted alongside theoretical state probability for 2^5 possible states. Network parameters initialized randomly.

4 Energy and Gradient Calculation

4.1 Energy

Using the FFNN architecture described in Section 2, we parameterize the full wavefunction $\Psi(\Omega)$ by Ω, where Ω denotes the network weights and biases in a FFNN. Using standard machine learning gradient descent, these network parameters can be tuned such that the energy calculation is minimized during training and the ground state is approximated via QNADE. The energy of a system parameterized by Ω is given by

$$E(\Omega) = \frac{\Psi^\dagger(\Omega) H \Psi(\Omega)}{\Psi^\dagger(\Omega)\Psi(\Omega)}, \tag{9}$$

where H is an arbitrary Hamiltonian. To build compatibility with our ansatz, we introduce a localized energy quantity that is calculated per sample,

$$\epsilon(s) \equiv \frac{1}{\psi_s^*(\Omega)} \sum_{s'} \psi_{s'}^*(\Omega) H_{s',s}. \tag{10}$$

Using this localized energy, E can be expressed as the expectation value of $\epsilon(s)$,

$$E(\Omega) = \frac{\sum_s |\psi_s(\Omega)|^2 \epsilon(s)}{\sum_s |\psi_s(\Omega)|^2} \equiv \langle \epsilon(s) \rangle. \tag{11}$$

An H matrix corresponding to a physical Hamiltonian generally has an average sparsity between $O(N)$ and $O(N^2)$ [2], resulting in a relatively efficient calculation of $\epsilon(s)$ and $E(\Omega)$ across all samples.

Throughout this project, our focus has been on a 1D chain of spins in the presence of a transverse-field Ising model (TFIM), a model that describes the behavior of many

magnetic materials in nature. This model is particularly interesting because it parameterizes the paramagnetic-ferromagnetic quantum phase transition by the magnetic field coupling strength, g [3]. The ferromagnetic TFIM Hamiltonian is given by

$$\hat{H} = -\sum_{i=1}^{N} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z - g \sum_{i=1}^{N} \hat{\sigma}_i^x \tag{12}$$

where $\hat{\sigma}_i^z$ and $\hat{\sigma}_i^x$ are Pauli matrices acting on site i. This Hamiltonian is characterized by nearest-neighbour interactions in the z direction and an external magnetic field perpendicular to the z axis. Using the TFIM, we are able to calculate $\epsilon(s)$ for each sample as well as the total energy of the system, $E(\Omega)$. Note, this Hamiltonian was implemented with periodic boundary condition $\sigma_N^z = \sigma_1^z$.

4.2 Energy Gradient

To minimize energy, $E(\Omega)$ must be treated as the loss function during gradient descent. Thus, the multivariable optimization across all network parameters requires us to obtain the gradient of $E(\Omega)$ with respect to each network parameter Ω_k. The energy gradient across all network parameters can be expressed as

$$\frac{\partial E(\Omega)}{\partial \Omega_k} = \langle O_k(s)\epsilon(s)\rangle - \langle O_k(s)\rangle\langle \epsilon(s)\rangle = \langle O_k(s)\epsilon(s)\rangle - \langle O_k(s)\rangle E(\Omega) \tag{13}$$

where $O_k(s) \equiv \frac{1}{\psi_s(\Omega)} \frac{\partial \psi_s(\Omega)}{\partial \Omega_k}$. The introduction of $O_k(s)$ for each sample allows us to solve for the energy gradient analytically, thus significantly reducing the computational complexity of the energy gradient calculation.

Obtaining $O_k(s)$ for a given state requires calculating the gradient of its conditional wavefunction $\psi_s(\Omega)$ with respect to each network parameter Ω_k. Conveniently, this gradient can be accumulated autoregressively. Since the wavefunction of a sampled state is naturally a product of conditionals, its gradient with respect to some network parameter Ω_k can also be written as a product of conditionals,

$$\frac{\partial \psi(s)}{\partial \Omega_k} = \prod_{i=1}^{N} \frac{\partial \psi_i(s_i|s_{i-1}, ..., s_1)}{\partial \Omega_k}. \tag{14}$$

However, recalling the implementation of Eq.15 allows us to write the gradient as a sum of conditionals,

$$\frac{\partial \ln[\psi(s)]}{\partial \Omega_k} = \sum_{i=1}^{N} \frac{\partial \ln[\psi_i(s_i|s_{i-1}, ..., s_1)]}{\partial \Omega_k} = \sum_{i=1}^{N} \frac{1}{\psi_i} \frac{\partial \psi_i}{\partial \Omega_k}, \tag{15}$$

where ψ_i is the sampled v_{i,s_i} for a given masked state. This mathematical log trick allows the easy accumulation of $O_k(s)$ during the sample generation process. Matrix multiplication resulting from Eq.14 inherently requires more computation than element-wise addition, which makes a simple running sum a more efficient method to accumulate sample gradients. For information on parallel sample generation and gradient accumulation, see Section 5.1.

Once the energy gradient has been obtained, gradient descent can be performed based on a pre-determined optimization algorithm. For our project, we chose to implement Adam

optimization based on previous student's work and it's broad machine learning applicability. While Adam is more complex than the naive gradient descent given by Eq.16 (Adam is a combination of RMSprop and Stochastic Gradient Descent with momentum), the basic idea is to update each network parameter in the subsequent training loop with some slight adjustment to the parameter from the previous training loop,

$$\Omega_k^{l+1} = \Omega_k^l - \alpha \frac{\partial E}{\partial \Omega_k^l}. \tag{16}$$

Here, Ω_k^{l+1} denotes the network parameter to be updated, Ω_k^l is a network parameter in the current training loop, α is the learning rate, and $\frac{\partial E}{\partial \Omega_k^l}$ is the energy gradient for parameter k.

5 Optimization

Modern programming libraries have facilitated the application of machine learning to a variety of scientific problems. One of Python's machine learning libraries, Pytorch, is a great example of such a tool. With this library, developers can easily compute the gradients of a network's output with respect to it's parameters by using a built-in back-propagation function (which was used to calculate $O_k(s)$ in Eq.13). Pytorch also offers a variety of complex optimization algorithms that can be implemented with only a few lines of code. Finally, Pytorch has very simple GPU interfacing tools that allow developers to execute network training and large matrix computations on more suitable machines. All together, these tools provide a very streamlined optimization process that naturally lends itself to parallelization and efficiency. It is with these tools that we built our sampling algorithm and optimization procedure.

5.1 Fully Integrated Optimization Algorithm

Our full optimization loop combines QNADE sampling using a FFNN, the calculation of the energy gradient, and a network parameter update. One training iteration through the full algorithm can be broken down into four main steps:

1. Generate M samples, wavefunction coefficients, and gradients. For N iterations:
   (a) Mask all $s_i, ..., s_N$ bits. Execute a forward pass through the QNADE model to obtain $v_{i,pos}$ and $v_{i,neg}$ for M samples. Normalize.
   (b) Use Bernoulli sampling to generate M bits based on probability $v_{i,pos}^2$.
   (c) Flip sampled bit values of 0 to -1 (to differentiate from the mask).
   (d) Update M sample bit-strings with newly sampled bits.
   (e) Update M conditional wavefunction coefficients based on newly sampled bits. If $s_i == 1$, update $\psi_i(s) = \psi_i(s) * v_{i,pos}$. Else, update $\psi_i(s) = \psi_i(s) * v_{i,neg}$.
   (f) Calculate $O_k(s)$ for M samples using backpropagation. If $s_i == 1$, accumulate a backpropagation pass on $v_{i,pos}$. Else, accumulate a backpropagation pass on $v_{i,neg}$.
2. Calculate $\epsilon(s)$ for M samples. Calculate energy by averaging $\epsilon(s)$ (Eq.10, 11).

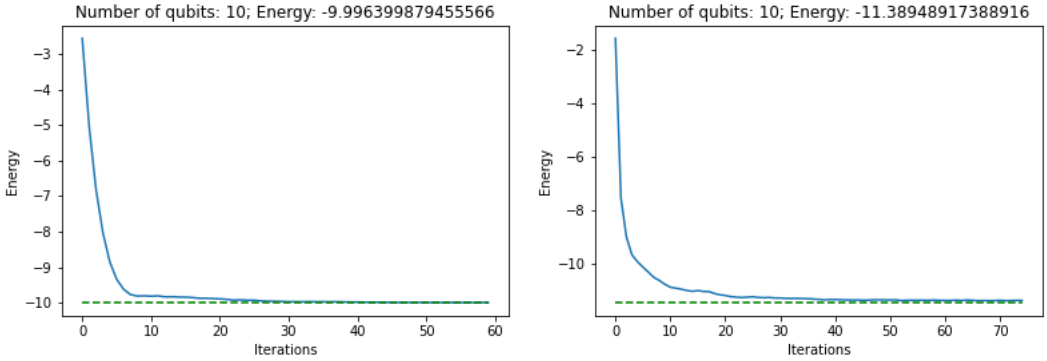
3. Calculate energy gradient by averaging all $O_k(s)$ weighted by $\epsilon(s)$ (Eq.13).

4. Update the network parameter gradient attributes with the calculated energy gradient. Update network parameters by calling Pytorch optimizer.

One important piece of this algorithm is an open-source library called Autograd Hacks. This library provides the functionality to compute $\frac{\partial \ln[\psi(s)]}{\partial \Omega_k}$ for M independent samples in parallel, thus enabling parallelized sample generation and gradient accumulation.

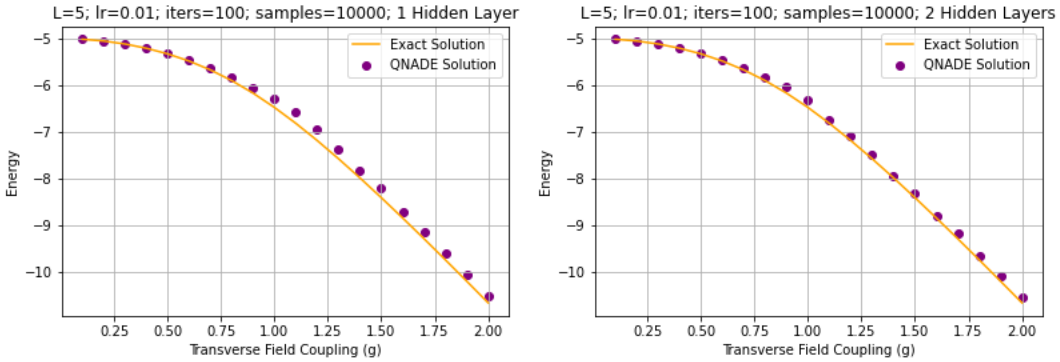
5.2 Convergence Results

We first obtained the ground state energy for a 10-qubit system in the absence of a magnetic field. As shown by Figure 4, good convergence to the ground state was obtained after approximately 25 training loops when $g = 0$ and after 50 training loops when $g = 0.75$. Optimization was also observed for multiple system sizes, g values and hidden layer structures (Figures 5 and 6).



(a) (b)

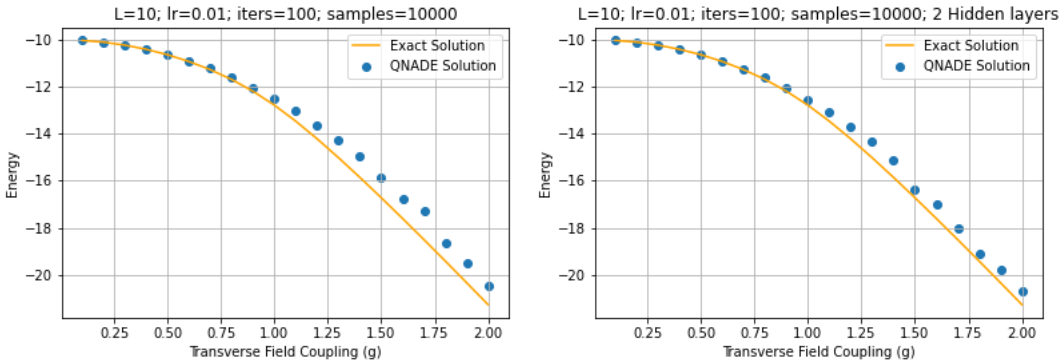
Figure 4: Convergence plots to the ground state energy (green dashed line) for (a) g=0 and (b) g=0.75. Adam optimization for 10 qubits, 10000 samples generated per training iteration. Randomized network initialization produces variational convergence plots for constant system size, but still results in consistent convergence. Learning rate was adjusted slightly per trial to improve convergence. Theoretical ground state energy was calculated exactly.



(a) (b)

Figure 5: 5-qubit convergence across a variety of g values for (a) 1 hidden layer and (b) 2 hidden layers

The largest discrepancy between the QNADE and exact solutions occurred around the quantum critical point, $g = 1$. This makes intuitive sense because a many-body system at the quantum phase transition is typically the hardest to solve. However, slightly adjusting the network architecture to include an additional hidden layer improved overall convergence accuracy. In the future, experimenting with network depth and the number of hidden nodes could be an interesting area of research. While it may improve convergence accuracy, a balance must be reached between network size and system size due to the additional time cost introduced by training larger networks. Similar data was obtained for a system of 10 qubits. However, the discrepancy between the QNADE and exact solutions was significantly larger for $g > 1$.



(a) (b)

Figure 6: 10-qubit convergence across a variety of g values for (a) 1 hidden layer and (b) 2 hidden layers

While our optimization system was often successful in converging to the ground state, especially for $g < 1$, this was not the case for every trial. Sometimes, optimization runs

would converge to a local minimum rather than the ground state energy. If this system is to be used in research, convergence success needs to be quantified. Thus, 100 tests were run for constant system sizes, providing more insight into convergence consistency and accuracy. The difference between expected energy and QNADE energy (Delta E) was calculated, binned and plotted (Figure 7).

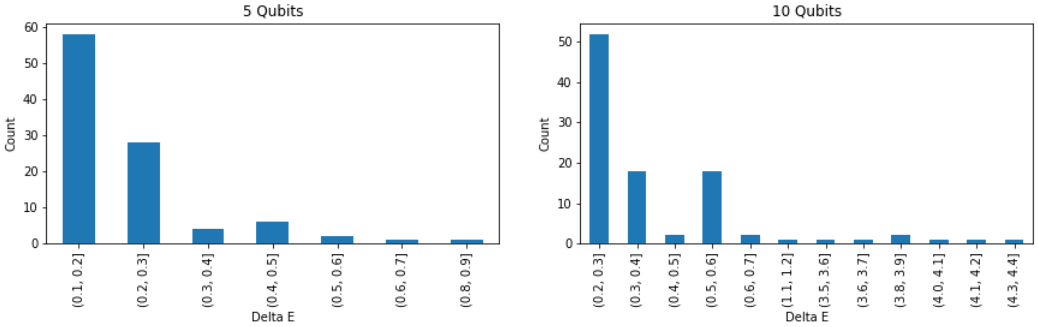


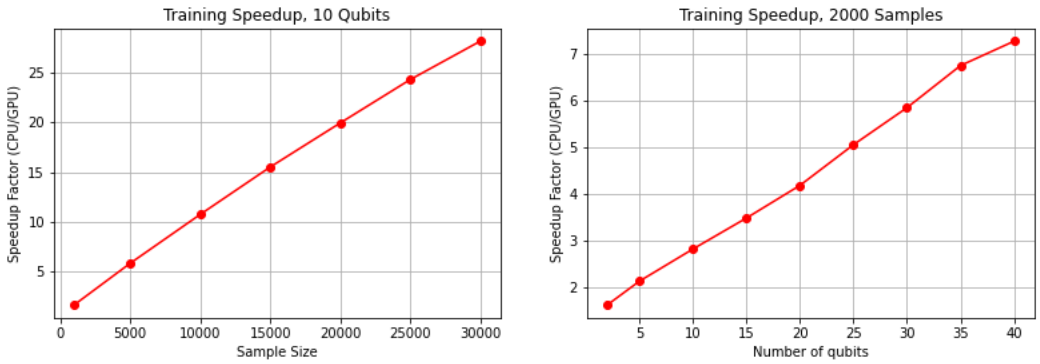
Figure 7: ΔE between exact and QNADE solutions.

These results echo previous tests with respect to convergence accuracy. For a smaller system size of 5 qubits, nearly 60% of QNADE solutions fell within within (0.1, 0.2] of the expected ground state energy. However, there was more discrepancy when testing 10 qubits, as none of the QNADE solutions fell in the range (0.1, 0.2] of the expected energy. In the 10-qubit system, the majority of QNADE solutions differed from the expected energy by (0.3, 0.4]. We're not sure why larger systems appear to have larger error. But, this behavior should be further investigated since convergence error that scales with system size could be problematic.

6 GPU Acceleration

While implementing a neural network as an ansatz greatly reduces the computational complexity required to solve the quantum many-body problem, this process still requires training a network over a specified number of iterations. Even after reducing the dimensionality of the system and parallelizing sample generation using QNADE, the training time of our optimization procedure still increased with the number of samples generated per training loop and the number of qubits in the system. However, this training time was greatly reduced by implementing the training procedure on a graphical processing unit (GPU).

Using Google Colab's free access to a 12GB NVIDIA Tesla K80 GPU, we independently tested the effect of system and sample size on CPU/GPU speedup factor (Figure 8). One of the major successes of the QNADE algorithm implemented on a GPU is a significant speedup factor for large sample batches. When tested with a batch size of 30000 samples per training iteration, GPU training is over 25 times faster than CPU training for 10 qubits. Since sample error is generally agreed to scale with $1/\sqrt{M}$ (where M is the number of samples), larger sample batches allow for more accurate density estimation and smoother convergence. While not quite as significant, there is also a clear GPU speedup at larger system sizes.



(a) (b)

Figure 8: CPU/GPU speedup factor for (a) constant system size and (b) constant sample size. Training involved 100 iterations through optimization loop. Convergence success was not evaluated for these optimization runs as convergence success has no impact on training time.

Since CPU/GPU speedup factor depends on the number of GPU cores, we expected to see the speedup factor saturate at a certain sample/system size. However, we ran into memory limitations on the GPU at about 55 qubits before this saturation could be observed. Speedup saturation also went unobserved during sample size testing because sample batches greater than 30000 were infeasible to test on the CPU. In the future, GPU memory limitations could be addressed by using a GPU with larger memory storage and/or adjusting our code to better optimize memory usage during training.

7 Conclusion and Future Work

During this project, we successfully built a machine learning algorithm that solves for the ground state energy of a quantum system by using a neural network with NADE sampling as an ansatz to approximate the many-body wavefunction. Furthermore, we implemented this optimization procedure on a GPU and achieved significant training speedup, especially with large sample batches. While we were successful in our proof of concept system, there are many elements that need further investigation before this system can be considered fully reliable. Our main area of concern is the accuracy discrepancy between the QNADE and exact solutions for larger systems. During testing, we realized the error between the QNADE and exact solutions appears to increase noticeably with a relatively small increase in system size from 5 to 10 qubits. Future research should investigate machine learning techniques that could improve convergence for these larger systems. For example, network parameter basis transformations, hidden layer adjustments, learning rate schedulers, model re-training, and hyper-parameter tuning are a few areas of interest.

In addition, this system should be tested with different Hamiltonians of varying entanglement levels to provide more insight into its efficacy. After brief testing with the

anti-ferromagnetic TFIM given by

$$\hat{H} = \sum_{i=1}^{N} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z + g \sum_{i=1}^{N} \hat{\sigma}_i^x, \tag{17}$$

convergence consistency appeared to be qualitatively worse than with the ferromagnetic TFIM. This model appeared to have more difficulty converging to the ground state energy and more often experienced jagged energy jumps or convergence to a local minimum during training. Unfortunately we were unable to obtain any quantitative data to compare these phases of the TFIM due to project time constraints. However, it makes intuitive sense that the anti-ferromagnetic TFIM ground state is more difficult to solve since it is Neel ordered and susceptible to spin frustration. Finally, further GPU testing should be done to observe the point at which CPU/GPU speedup factor saturates based on the number of GPU cores. Memory limitations also need to be address for large system sizes.

References

[1] O. Sharir, Y. Levine, N. Wies, G. Carleo, A. Shashua, Physical Review Letters 124, Deep Autoregressive Models for the Efficient Variational Simulation of Many-Body Quantum Systems (2020).

[2] Z. Gong, Learning quantum systems using neural networks (2020).

[3] B. Lloyd, M. Kuhnel, Z. Gong, Solving Quantum Many-body Problems with GPU Accelerated Machine Learning and Quantum Neural Networks (2019).

[4] B. Uriah, M. Cote, K. Gregor, I. Murray, H. Larochelle, Journal of Machine Learning Research 17, pp. 1-37, (2016).