# Quantum Computing for Machine Learning and Physics Simulation

Thesis by
Alexander Zlokapa

Thesis advisor:
Maria Spiropulu


CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2021

Alexander Zlokapa
ORCID: 0000-0002-4153-8646

# ACKNOWLEDGEMENTS

I am grateful to Professor Maria Spriopulu as my thesis advisor, academic advisor, and mentor for introducing me to quantum computing, high energy physics, and the excitement of research ever since my freshman year. My collaborators, especially Professor Daniel Jafferis, Professor Seth Lloyd, Dr. Murphy Niu, and Dr. Ryan Babbush have all provided remarkable patience and generosity throughout our explorations. Sam Davis helped as an equal foreigner to the physics of wormholes, and I'm very thankful for our discussions. Dr. Hartmut Neven, in both his capacities as a researcher and a mentor, provided more support than I could have asked for in my journey of learning quantum computing. I appreciate his generosity in letting me include in this thesis some work relevant to my internship at Google Quantum AI under his supervision.

In my broader adventure in physics, I'm especially appreciative of the Caltech mentors who pushed me towards a deeper understanding of everything around me: Professor Harvey Newman, for his illuminating lectures in freshman physics; Professor Rob Phillips, for his inspiring tenacity in "getting to the bottom of things"; and my peers, who never failed to support me both in academics and beyond. Finally and most importantly, I would like to thank my family for always encouraging me to pursue my curiosity.

# ABSTRACT

Quantum computing is widely thought to provide exponential speedups over classical algorithms for a variety of computational tasks. In classical computing, methods in artificial intelligence such as neural networks and adversarial learning have enabled drastic improvements in state-of-the-art performance for a variety of tasks. We consider the intersection of quantum computing with machine learning, including the quantum algorithms for deep learning on classical datasets, quantum adversarial learning for quantum states, and variational quantum machine learning for improved physics simulation.

We consider a standard deep neural network architecture and show that conditions amenable to trainability by gradient descent coincide with those necessary for an efficient quantum algorithm. Considering the neural network in the *infinite-width* limit using the neural tangent kernel formalism, we propose a quantum algorithm to train the neural network with vanishing error as the training dataset size increases. Under a sparse approximation of the neural tangent kernel, the training time scales logarithmically with the number of training examples, providing the first known exponential quantum speedup for feedforward neural networks. Related approximations to the neural tangent kernel are discussed, with numerical studies showing successful convergence beyond the proven regime. Our work suggests the applicability of the quantum computing to additional neural network architectures and common datasets such as MNIST, as well as kernel methods beyond the neural tangent kernel.

Generative adversarial networks (GANs) are one of the most widely adopted machine learning methods for data generation. We propose an *entangling* quantum GAN (EQ-GAN) that overcomes some limitations of previously proposed quantum GANs. EQ-GAN guarantees the convergence to a Nash equilibrium under minimax optimization of the discriminator and generator circuits by performing entangling operations between both the generator output and true quantum data. We show that EQ-GAN has additional robustness against coherent errors and demonstrate the effectiveness of EQ-GAN experimentally in a Google Sycamore superconducting quantum processor. By adversarially learning efficient representations of quantum states, we prepare an approximate quantum random access memory and demonstrate its use in applications including the training of near-term quantum neural networks.

With quantum computers providing a natural platform for physics simulation, we investigate the use of variational quantum circuits to simulate many-body systems with high fidelity in the near future. In particular, recent work shows that teleportation caused by introducing a weak coupling between two entangled SYK models is dual to a particle traversing an AdS-Schwarzschild wormhole, providing a mechanism to probe quantum gravity theories in the lab. To simulate such a system, we propose the process of *compressed Trotterization* to improve the fidelity of time evolution on noisy devices. The task of learning approximate time evolution circuits is shown to have a favorable training landscape, and numerical experiments demonstrate its relevance to simulating other many-body systems such as a Fermi-Hubbard model. For the SYK model in particular, we demonstrate the construction of a low-rank approximation that favors a shallower Trotterization. Finally, classical simulations of finite-$N$ SYK models suggest that teleportation via a traversable wormhole instead of random unitary scrambling is achievable with $O(20)$ qubits, providing further indication that such quantum gravity experiments may realizable with near-term quantum hardware.

# PUBLISHED CONTENT AND CONTRIBUTIONS

[1]M. Broughton, G. Verdon, T. McCourt, A. J. Martinez, J. H. Yoo, S. V. Isakov, P. Massey, M. Y. Niu, A. Zlokapa, R. Halavati, E. Peters, M. Leib, A. Skolik, M. Streif, D. V. Dollen, J. R. McClean, S. Boixo, D. Bacon, A. K. Ho, H. Neven, and M. Mohseni, *Tensorflow quantum: a software framework for quantum machine learning*, in preparation.
A. Zlokapa contributed discussion of quantum generative adversarial networks, as well as accompanying code and examples.

[2]A. Zlokapa and S. Lloyd, *A quantum algorithm for training wide and deep neural networks*, in preparation.
A. Zlokapa proposed and proved the main result for deep neural networks.

[3]A. Zlokapa, J. Lykken, S. Davis, D. Jafferis, and M. Spiropulu, *Near-term quantum simulation of wormhole teleportation*, in preparation.
A. Zlokapa prepared simulations of the Dirac SYK and proposed near-term quantum approaches to time evolution and state preparation.

[4]M. Y. Niu*, A. Zlokapa*, M. Broughton, S. Boixo, M. Mohseni, V. Smelyanskyi, and H. Neven, *Quantum generative adversarial networks with provable convergence*, 2021.
A. Zlokapa proposed the discriminator architecture, applications to noise suppression and quantum random access memory, and vanishing gradient example; also performed quantum hardware experiments and numerical simulation.

* indicates equal contribution.

# TABLE OF CONTENTS

*Chapter 1*

# A QUANTUM ALGORITHM FOR TRAINING DEEP NEURAL NETWORKS

[1]A. Zlokapa and S. Lloyd, *A quantum algorithm for training wide and deep neural networks*, in preparation.

## 1.1 Introduction

While deep neural networks have achieved state-of-the-art results in numerous relevant problems, the computational requirements of deep learning are expected to be increasingly costly as datasets and neural network architectures both grow in size. Founded in established complexity theory conjectures, *quantum computing* is widely believed to be computationally more powerful than classical computing. With experimental quantum computers currently solving certain computational tasks faster than modern supercomputers [1–4], quantum computers are expected to eventually achieve exponential and polynomial speedups over a wide variety of classical algorithms, including essential primitives in linear algebra and optimization [5–10]. Realizing an exponential speedup in relevant settings often requires stringent theoretical caveats to be satisfied such as sparsity and matrix conditioning [11], as well as advanced quantum hardware such as a quantum random access memory (QRAM). Although quantum machine learning algorithms have been proposed for common classical approaches to classification, clustering, regression, and other tasks in data analysis [12–16], the results vary in applicability due to these caveats. Despite the central importance of deep neural networks, proposals for quantum neural networks analogous to widely used classical deep learning architectures lack a rigorous demonstration of a quantum speedup for deep learning tasks on classical data [17–20].

Recent work on the dynamics of overparameterized neural networks has introduced the *neural tangent kernel* (NTK), representing large neural networks as linearized models applied to nonlinear features [21]. In particular, *deep* neural networks are empirically observed to achieve successful results [22–24], with theoretical justifications including batch normalization [25] and improved

conditioning [26]. This improved understanding of deep learning motivates a quantum algorithm for the NTK, which has properties that favor both optimization by gradient descent and by techniques from quantum computing.

**Contributions.** Within the NTK framework, we consider a standard fully-connected neural network architecture with an additional normalization condition on the activation function and reasonable dataset assumptions in a standard setting. Our main contributions can be summarized as follows:

- **Sparsified neural tangent kernels.** We study a *sparsified NTK* with a logarithmic number of nonzero off-diagonal elements. Deep neural networks are shown to naturally be dominated by a small neighborhood of nearby examples, causing the output of a sparsified NTK to converge to that of a dense NTK as the size of the dataset increases. Empirical examples are shown for both toy datasets and the MNIST dataset.

- **Exponential quantum speedup for sparsified neural tangent kernels.** We show that computing the output of a sparsified NTK can be performed exponentially more quickly with a quantum computer under widely held complexity theory conjectures. Once the dataset is stored in quantum memory, different neural networks can be trained with efficient memory input/output and computation of predictions. In particular, the precise well-conditioning properties of the NTK that allow efficient gradient descent are shown to be critical for a quantum speedup. Empirical experiments beyond the proven regime suggest that convolutional NTKs and chaotic kernel methods beyond the NTK may also benefit from the demonstrated quantum advantage.

- **Diagonal neural tangent kernels.** Due to the well-conditioning of a deep neural network, we show that the output of an NTK can be also be approximated by removing *all* off-diagonal elements of the NTK matrix. Once again, a quantum algorithm is shown to evaluate the approximate NTK's predictions in time logarithmic in the number of training examples. Empirical evidence shows that the diagonal NTK rapidly converges to the exact NTK, although it shown to have strictly greater error than the sparsified NTK approximation. While the sparsified NTK requires sparse matrix inversion and is thus likely robust to improvements in clas-

sical algorithms, it is an open question if there exist efficient quantum-inspired classical algorithms for the diagonal NTK, which may provide further insight into the effectiveness of deep learning using small data subsets.

**Neural network preliminaries**

We begin by describing the notation and initial assumptions for the neural network and data, adopting a framework similar to Agarwal et al. [26]. Consider a binary classification dataset $S$ of $n$ training examples $\{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}\}_{i=1}^n$. Throughout this work, we will require a notion of *separability* between any two given data examples.

**Definition 1.1.1** (Separability)**.** The separability of data points $\mathbf{x}_i, \mathbf{x}_j$ is given by $\delta_{ij} := 1 - |\mathbf{x}_i \cdot \mathbf{x}_j|$.

We make the following standard assumption about separability across the entire dataset [27–29] with an additional lower bound on the separability that is commonly satisfied (see Sec. A.5 of the Appendix A).

**Assumption 1.1.2.** Assume that $|\mathbf{x}_i \cdot \mathbf{x}_i| = 1$ for all $i$. For some $0 < \delta \leq 1$, let $|\mathbf{x}_i \cdot \mathbf{x}_j| \leq 1 - \delta$ for all $i, j \in [n]$ with $i \neq j$. Moreover, assume $\delta = \Omega(1/\operatorname{poly} n)$ for a dataset of size $n$.

To ensure the dataset is well-behaved, i.e. labels do not change at an arbitrarily small scale on the unit sphere, we require an additional assumption.

**Assumption 1.1.3.** Define the $\epsilon$-neighborhood around a given data point $\mathbf{x}_*$ sampled i.i.d. from the data distribution to be $N_\epsilon = \{i : \mathbf{x}_* \cdot \mathbf{x}_i \geq 1 - \epsilon\}$. There exists a constant $\epsilon$ such that with high probability $y_i = y_*$ for all $\{y_i : i \in N_\epsilon\}$. Moreover, the distribution of $\mathbf{x}_i$ within $N_\epsilon$ is approximately uniform.

We consider a feedforward neural network with $L$ hidden fully-connected layers of width $m$ and an activation function $\sigma : \mathbb{R} \to \mathbb{R}$ applied entry-wise. At initialization, its weights are drawn i.i.d. from $\mathcal{N}(0, 1)$. Defining each weight matrix $W_i$ at the $i$th hidden layer and output layer weights $v \in \mathbb{R}^m$, the neural network can be expressed as a function $f_{\mathrm{NN}} : \mathbb{R}^d \to \mathbb{R}$ that maps data $\mathbf{x}$ to real-valued output $y$:

$$f_{\text{NN}}(\mathbf{x}) := v \cdot \frac{1}{\sqrt{m}}\sigma\left(W_L \frac{1}{\sqrt{m}}\sigma\left(W_{L-1}\ldots\frac{1}{\sqrt{m}}\sigma\left(W_1\mathbf{x}\right)\ldots\right)\right). \tag{1.1}$$

Training the NTK with squared loss for a single-output regression in the wide limit yields at any timestep a Gaussian with mean $\mathbb{E}[f_*](t)$ and variance $\mathbb{V}[f_*(t)]$. In the limit of $t \to \infty$, training the neural network converges to an output distribution given by Lemma 1.1.1 [21, 30]:

**Lemma 1.1.1.** *Let $K_{\text{NTK}}$ denote the NTK of the neural network in Eq. 1.1 as $t \to \infty$, and let $f_0$ denote the output of the neural network at $t = 0$. Since $f_0$ produces Gaussian-distributed output, we can define the covariance between $\mathbf{x}_i$ and $\mathbf{x}_j$ in the infinite width limit:*

$$K_{\text{cov}}(\mathbf{x}_i, \mathbf{x}_j) = \lim_{m\to\infty} \mathbb{E}\left[f_0(\mathbf{x}_i) \cdot f_0(\mathbf{x}_j)\right].$$

*Consider a test data vector $\mathbf{x}_*$. Defining $(\mathbf{k}_{\text{NTK}})_*, (\mathbf{k}_{\text{cov}})_* \in \mathbb{R}^n$ as the vectors generated by applying the corresponding kernel to the vector $\mathbf{x}_*$ and the training set $\mathcal{S}$, the mean and variance of the Gaussian output $f_*$ of the converged NTK as $t \to \infty$ are given by:*

$$\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1}\mathbf{y} \tag{1.2a}$$

$$\begin{aligned}\mathbb{V}[f_*] = K_{\text{cov}}(\mathbf{x}_*, \mathbf{x}_*) + (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} K_{\text{cov}} K_{\text{NTK}}^{-1}(\mathbf{k}_{\text{NTK}})_* \\ - ((\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1}(\mathbf{k}_{\text{cov}})_* + h.c.),\end{aligned} \tag{1.2b}$$

*where h.c. denotes the Hermitian conjugate.*

We place additional conditions on the normalization of the activation function, which is equivalent to the application of batch normalization at each layer of a neural network [26].

**Assumption 1.1.4.** The activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is normalized such that

$$\mathop{\mathbb{E}}_{X\sim\mathcal{N}(0,1)}[\sigma(X)] = 0 \text{ and } \mathop{\mathbb{V}}_{X\sim\mathcal{N}(0,1)}[\sigma(X)] = \mathop{\mathbb{E}}_{X\sim\mathcal{N}(0,1)}[\sigma^2(X)] = 1. \tag{1.3}$$

Following Agarwal et al. [26], we define the nonlinearity of the activation function and note the effect of normalization on the resulting constant.

**Definition 1.1.5** (Coefficient of nonlinearity)**.** The coefficient of nonlinearity of the activation function $\sigma$ is defined to be $\mu := 1 - \left(\mathbb{E}_{X\sim\mathcal{N}(0,1)}[X\sigma(X)]\right)^2$.

**Corollary 1.1.2.** *Under Assumption 1.1.4, the nonlinearity of $\sigma$ is bounded to be $0 < \mu \leq 1$.*

We include an important result on convergence when training all the layers of the neural network (more formally stated in Sec. A.1 of the Appendix A).

**Theorem 1.1.3.** *Suppose $\sigma$ is smooth, bounded, and has bounded derivatives. If the width is a large enough constant (depending on $L, n, \delta$) and $L = \Omega\left(\frac{\log(n/\delta)}{\mu}\right)$, then gradient descent with high probability finds an $\epsilon$-suboptimal point in $O(\log(1/\epsilon))$ iterations.*

This motivates the definition of a threshold depth for which convergence by gradient descent is guaranteed:

$$L_{\text{conv}} := \frac{8\log(n/\delta)}{\mu}, \tag{1.4}$$

where a constant prefactor is included to ensure favorable properties of the NTK. Efficiently computing a matrix element of the NTK with $L = \Theta(L_{\text{conv}})$ requires a minimum data separability.

**Lemma 1.1.4.** *If $L = \Theta(L_{\text{conv}})$ and $\delta = \Omega(1/\text{poly}\, n)$, then an element of the NTK can be computed in $O(\text{polylog}\,(n)/\mu)$ time given the inner product between two data points.*

**Quantum preliminaries**

We assume some basic familiarity with quantum computing [31] but provide here the background on necessary quantum algorithms for the NTK. In particular, the quantum linear systems problem (QLSP) [32] provides the basis for a robust exponential speedup.

**Definition 1.1.6** (QLSP)**.** Let $A$ be an $n \times n$ Hermitian matrix with condition number $\kappa$, unit determinant, and at most $s$ nonzero entries in any row or column. Let $\mathbf{x}, \mathbf{b}$ be $n$-dimensional vectors such that $\mathbf{x} = A^{-1}\mathbf{b}$. We define the quantum states $|b\rangle, |x\rangle$ such that

$$|b\rangle := \frac{\sum_{i=1}^n b_i |i\rangle}{||\sum_{i=1}^n b_i |i\rangle||} \quad \text{and} \quad |x\rangle := \frac{\sum_{i=1}^n x_i |i\rangle}{||\sum_{i=1}^n x_i |i\rangle||}. \tag{1.5}$$

Given access to a procedure $\mathcal{P}_A$ that computes the locations and values of the nonzero entries in $A$, and a procedure $\mathcal{P}_B$ that prepares the state $|b\rangle$ in $O(\text{polylog}(n))$ time, output a state $|\tilde{x}\rangle$ such that $|| \, |\tilde{x}\rangle - |x\rangle \, || \leq \xi$, succeeding with probability larger than $1/2$ and providing a flag indicating success.

The HHL algorithm [5] solves QLSP for general $A$: the assumption that $A$ is Hermitian can be dropped without loss of generality by constructing a linear system defined by a Hermitian antidiagonal block matrix with components $A$ and $A^\dagger$. Further improvements of the QLSA beyond HHL obtain a runtime of $O(\log(n)\kappa s \, \text{polylog}(\kappa s/\xi))$ [6].

HHL can solve the problem of sparse matrix inversion, which is known to be BQP-complete. Thus, it is widely thought that no classical algorithm can fully replace HHL based on standard complexity conjectures. However, individual cases may be classically addressed: recent work on low-rank linear systems have yielded quantum-inspired classical algorithms [33]. Additionally, due to the dependence of the QLSA on matrix condition number and sparsity, there are several caveats that must be satisfied to achieve an exponential speedup in $n$ over known classical algorithms [11].

**Remark 1.1.5** (Caveats to QLSA exponential speedup.)**.** *The following conditions must be satisfied to achieve an exponential quantum speedup with a quantum linear systems algorithm (QLSA).*

1. *The matrix $A$ must be $s$-sparse with $s = O(\text{polylog}(n))$.*
2. *The matrix $A$ must be well-conditioned with $\kappa(A) = O(\text{polylog}(n))$.*
3. *The matrix $A$ must have rank at least $\Omega(\text{poly}(n))$.*
4. *The procedure $\mathcal{P}_A$ that provides the indices of nonzero elements in a given column must take $O(\text{polylog}(n))$ time.*
5. *The procedure $\mathcal{P}_B$ that loads the vector $\mathbf{b}$ into the quantum computer's memory must take $O(\text{polylog}(n))$ time.*
6. *The final state $|x\rangle$ must be efficiently read out, repeating the algorithm at most $O(\text{polylog}(n))$ times.*

As is typical in many quantum machine learning algorithms [12–16], we assume the existence of a quantum random access memory (QRAM) to store and access any necessary quantum states. The following binary tree QRAM subroutine was proposed by Kerenidis and Prakash [34] and is applied commonly in quantum machine learning [16, 35].

**Theorem 1.1.6** (QRAM)**.** *For $S \in \mathbb{R}^{n\times d}$, there exists a data structure that stores $S$ such that the time to insert, update or delete entry $S_{ij}$ is $O(\log^2(n))$. Moreover, a quantum algorithm with access to the data structure provides quantum access in time $O(\text{polylog}(nd))$.*

In the case of a quantum NTK algorithm, the training dataset must only be loaded into QRAM once; training different neural networks afterwards will only require logarithmic time in the training set size.

While sparse matrix inversion is BQP-complete and thus thought to be robust to the development of future classical algorithms, other techniques also provide quantum speedups. Using an approach similar to the q-means algorithm [16], we also achieve an exponential speedup over the standard classical approach to compute the NTK or its approximation. As with other quantum machine learning results, such speedups may ultimately give rise to quantum-inspired classical algorithms [33, 36–38]. Accordingly, we present two approximations to the neural tangent kernel with exponential quantum speedups: the first requires sparse matrix inversion, and the second has a slightly higher error without sparse matrix inversion. Both approximations asymptotically converge to the exact output of a deep neural network NTK.

## 1.2 Sparsified neural tangent kernel

We seek to approximate the NTK so as to converge to its expected output $\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1}\mathbf{y}$ on a test data vector $\mathbf{x}_*$. Classically, solving the linear equation $K_{\text{NTK}}\mathbf{v} = \mathbf{y}$ with an arbitrary $n \times n$ matrix $K_{\text{NTK}}$ requires time $O(n^3)$ to perform a Cholesky decomposition. In the *sparsified NTK*, we replace $K_{\text{NTK}}$ with an $s$-sparse matrix $\tilde{K}_{\text{NTK}}$, which has at most $s = O(\log n)$ nonzero elements in any matrix or column. When solving an $s$-sparse, well-conditioned system of linear equations, classical algorithms can estimate the solution up to precision $\xi$ in $O(s\, n \log(1/\xi))$ time [39]. Using a QLSA, we will obtain polylogarithmic dependence on $n$.

**Properties of the sparsified NTK**

To address caveats of a quantum speedup discussed above, we must describe several properties of the sparsified NTK as well as its convergence to the exact NTK. (Proofs of these properties are found in Secs. A.1 and A.4 of the Appendix A.) We begin by defining a *sparsification procedure* of the exact NTK, which relies on the structure of an NTK for a deep neural network. The use of a *deep* neural network has been recently shown to speed up optimization via gradient descent due to the well-conditioning of the NTK [26]. In our framework, the condition number approaches unity for neural networks deep enough to provably converge by gradient descent.

**Theorem 1.2.1** (Well-conditioning of the NTK)**.** *If $L \geq L_{\text{conv}}$, then the condition number $1 \leq \kappa(\tilde{K}_{\text{NTK}}) \leq \frac{1+1/n}{1-1/n}$ converges to unity as the training set size increases.*

The well-conditioning occurs because off-diagonal elements of the NTK rapidly vanish with larger datasets while all diagonal elements are equal. Since the NTK for the neural network given in Eq. 1.1 only depends on the inner product $\mathbf{x}_i \cdot \mathbf{x}_j$ for the $(i,j)$th matrix element, we can place precise bounds on the matrix elements.

**Lemma 1.2.2** (Upper bounds on the NTK)**.** *The diagonal of the NTK is equal everywhere and given by constant $(K_{\text{NTK}})_{11}$. If $L \geq L_{\text{conv}}$, then we have the following bounds on $(K_{\text{NTK}})_{ij}$ for $i \neq j$. If $0 < \delta_{ij} < 1/2$, $\left|\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}}\right| \leq \left(\frac{\delta}{\delta_{ij} n}\right)^2$, while for $1/2 \leq \delta_{ij} \leq 1$, $\left|\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}}\right| \leq \left(\frac{\delta}{n}\right)^2$.*

As data vectors become more parallel, the value of the NTK increases until it reaches a maximum when they are fully parallel. Since $\delta = \Omega(1/\text{poly } n)$ by Assumption 1.1.2, Lemma 1.2.2 shows that the largest off-diagonal elements (where $\delta_{ij} = \delta$) fall off more slowly (like $1/n^2$) than distant off-diagonal elements (like $1/n^{5/2}$ if, for instance, $\delta \sim n^{-1/2}$). As the dataset gets larger and the neural network gets deeper, this effect focuses the NTK on the most similar examples. We use this feature to sparsify the NTK matrix and probabilistically select off-diagonal matrix elements as nonzero according to their magnitude; such sparsification naturally conforms to the structure of the exact NTK and thus the inverses converge in the asymptotic limit.

**Theorem 1.2.3** (Convergence of the sparsified NTK to the exact NTK)**.** *Let $M = \tilde{K}_{\mathrm{NTK}}$ be a sparsification of the exact NTK $K_{\mathrm{NTK}}$ with $s = O(\log n)$ off-diagonal elements. The error of the matrix inverse vanishes as $\frac{||\tilde{K}_{\mathrm{NTK}}^{-1} - K_{\mathrm{NTK}}^{-1}||}{||K_{\mathrm{NTK}}^{-1}||} = O(1/n)$.*

To efficiently prepare quantum states, we also require a normalization factor that sets a maximum for any off-diagonal matrix element of the NTK. This is later used to create the oracle $\mathcal{P}_A$ and to ensure efficient readout. Since the NTK depends only on the inner product, the NTK normalization threshold is equivalent to a separability threshold between two data vectors. The scale of this separability is set by Assumption 1.1.3, which requires data within the neighborhood $N_\epsilon = \{i : 1 - \mathbf{x}_* \cdot \mathbf{x}_i \leq \epsilon\}$ around vector $\mathbf{x}_*$ to have the same label with high probability. If the dataset has dimension $d = O(\log n)$, one can always choose constant $\epsilon'$ such that $\epsilon' < \epsilon$ such that clipping NTK values does not introduce more than $O(1/n)$ error (see Sec. A.1 of the Appendix A). The key property of the NTK that allows this is a lower bound on the NTK matrix elements.

**Lemma 1.2.4** (Lower bounds on the NTK)**.** *Consider any $\mathbf{x}_i, \mathbf{x}_j$ with separability $\delta_{ij}$ such that $\mathbf{x}_i \cdot \mathbf{x}_j > 0$. For an NTK of depth $L = L_{\mathrm{conv}}$, we have*

$$\frac{(K_{\mathrm{NTK}})_{ij}}{(K_{\mathrm{NTK}})_{11}} \geq O(1) \cdot \delta_{ij} \left(\frac{1-\delta}{n}\right)^{O(1)}, \tag{1.6}$$

*where the constants are given by the choice of activation function, $\epsilon'$, and $\epsilon$.*

**Quantum algorithm**

From the classical properties of the NTK, we have satisfied caveats 1 through 3 of Remark 1.1.5 with the sparsified NTK, which converges to the exact NTK up to $O(1/n)$ error. Using the QRAM to store the training set, we have also satisfied the efficient loading of quantum state $|y\rangle$ in the QLSP $\tilde{K}_{\mathrm{NTK}} |v\rangle = |y\rangle$. Two quantum subroutines are required to train the neural network: the efficient computation of the NTK over the dataset in superposition, and the efficient readout of the NTK prediction. To prepare the oracle $\mathcal{P}_A$ for matrix elements and the state $|k_*\rangle$ corresponding to the vector $(\mathbf{k}_{\mathrm{NTK}})_*$ in Eq. 1.2, we require the following result, which uses the fact that the NTK only depends on the inner product $\mathbf{x}_i \cdot \mathbf{x}_j$.

**Theorem 1.2.5** (Kernel estimation)**.** *Let $S \in \mathbb{R}^{n\times d}$ be the training dataset of $\{\mathbf{x}_i\}$ unit norm vectors stored in the QRAM described in Theorem A.3.1. For test data vector $\mathbf{x}_* \in \mathbb{R}^d$ in QRAM and a constant $\epsilon'$, there exists a quantum algorithm that maps*

$$\frac{1}{\sqrt{n}}\sum_{i=0}^{n-1}|i\rangle|0\rangle \mapsto \frac{1}{\sqrt{P}}\sum_{i=0}^{n-1}k_i|i\rangle\,. \tag{1.7}$$

*Here, $k_i = \hat{K}_{\mathrm{NTK}}(\rho_i)/\hat{K}_{\mathrm{NTK}}(1-\epsilon')$ is restricted to $-1 \le k_i \le 1$, i.e. clipping all $|\hat{K}_{\mathrm{NTK}}(\rho_i)| > \hat{K}_{\mathrm{NTK}}(1-\epsilon')$. The state is prepared with error $|\rho_i - \mathbf{x}_*\cdot\mathbf{x}_i| \le \xi$ with probability $1-2\Delta$ in time $\tilde{O}(\mathrm{polylog}(nd)\log(1/\Delta)/\xi)$.*

Finally, we show that the NTK classification result $\mathrm{sign}\left((\mathbf{k}_{\mathrm{NTK}})_*\tilde{K}^{-1}_{\mathrm{NTK}}\mathbf{y}\right)$ can be read out efficiently.

**Theorem 1.2.6** (Efficient readout)**.** *Given states $|k_*\rangle = \frac{1}{\sqrt{P}}\sum_{i=0}^{n-1}k_i|i\rangle$ and $\tilde{K}^{-1}_{\mathrm{NTK}}|y\rangle$ for $|y\rangle = \frac{1}{\sqrt{n}}\sum_{i=0}^{n-1}y_i|i\rangle$, the quantity $\mathrm{sign}\left((\mathbf{k}_{\mathrm{NTK}})_*\tilde{K}^{-1}_{\mathrm{NTK}}\mathbf{y}\right)$ can be estimated up to $O(1/n)$ error after a polylogarithmic number of measurements in $n$.*

The full quantum algorithm for the sparse NTK approximation is summarized as follows.

**Step 1: load data into QRAM.** Given training dataset $S$ of $n$ training examples $\{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{-1,1\}\}_{i=1}^n$, use the QRAM procedure of Kerenidis and Prakash [34] to create a binary tree data structure with efficient quantum access. Note that although creating the QRAM requires $O(n)$ time to iterate through the entire dataset, the cost only occurs once: changes to the data structure and the training of different neural networks will only require time polylogarithmic in $n$.

**Step 2: prepare the state $|p_*\rangle = \frac{1}{\sqrt{n}}\sum_{i=0}^{n-1}|i\rangle|\mathbf{x}_*\cdot\mathbf{x}_i\rangle$ representing a binary encoding of all inner products between test data example $\mathbf{x}_*$ and all training examples $\mathbf{x}_i$.** Following the first half of the proof of Theorem 1.2.5 (see Theorem A.3.5 in the Appendix A), we first access all $\mathbf{x}_i$ in superposition from the QRAM. Apply amplitude estimation [40] and median evaluation [41] to compute all inner products between the test data state $|x_*\rangle$ and each entry $|x_i\rangle$ of the training dataset.

**Step 3: compute the NTK $|k_*\rangle$ between the text data $\mathbf{x}_*$ and all training examples $\mathbf{x}_i$.** Since the NTK function only depends on inner products $\mathbf{x}_i \cdot \mathbf{x}_*$ and can be efficiently computed by Lemma 1.1.4, there exists a unitary that can take the state from Step 2 and compute $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle |K_{\text{NTK}}(\mathbf{x}_*, \mathbf{x}_i)\rangle$. Normalizing by the NTK factor $\epsilon'$, we perform controlled rotations on each bit of the representation then post-select to obtain state $|k_*\rangle$ with $K_{\text{NTK}}(\mathbf{x}_*, \mathbf{x}_i)$ encoded as an amplitude of $|i\rangle$. The post-selection requires time $O(1/\text{polylog } n)$.

**Step 4: solve QLSP $\tilde{K}_{\text{NTK}} |v\rangle = |y\rangle$ and output the result as a quantum state.** This is a straightforward application of HHL [5] or similar QLSA with access to oracle $\mathcal{P}_A$ that efficiently provides the indices of nonzero elements in a given column. We use Theorem 1.2.5 similarly as in Steps 2-3 to identify $O(\log n)$ nonzero elements in the NTK, choosing larger elements with higher probability; to ensure the nonzero elements are distinct, the QRAM is modified in $O(\text{polylog } n)$ time after each measurement. Applying the well-conditioned result of Theorem 1.2.1 and the imposed sparsity condition, the QLSA solves $\tilde{K}_{\text{NTK}}^{-1} |y\rangle$ with polylogarithmic cost in $n$.

**Step 5: estimate the output of the NTK approximation, i.e. measure $\text{sign}\left((\mathbf{k}_{\text{NTK}})_* \tilde{K}_{\text{NTK}}^{-1} \mathbf{y}\right)$.** Using Theorem 1.2.6, we prepare a state encoding the relative sign of $|k_*\rangle$ and $\tilde{K}_{\text{NTK}}^{-1} |y\rangle$ to estimate the sign of $\langle k_*| \tilde{K}_{\text{NTK}}^{-1} |y\rangle$ via an inner product estimation subroutine [42]. Because of Assumption 1.1.3 for sufficiently large training sets, the overlap between these states is large enough to ensure up to $O(1/n)$ error with a polylogarithmic number of measurements in $n$. Since the product $(\mathbf{k}_{\text{NTK}})_* \tilde{K}_{\text{NTK}}^{-1} \mathbf{y}$ is proportional to $\langle k_*| \tilde{K}_{\text{NTK}}^{-1} |y\rangle$ up to a positive normalization factor and Theorem 1.2.3 ensures convergence to the exact NTK output, the final classification can be performed up to $O(1/n)$ error of the wide and deep neural network.

### 1.3 Diagonal neural tangent kernel

As given by the upper bounds of Lemma 1.2.2, increasing neural network depth causes the off-diagonal elements to vanish, and the NTK approaches a matrix proportional to the identity matrix. While this behavior ultimately causes deep neural networks to be well-conditioned and trainable by gradient descent (Theorem 1.1.3), it also allows wide neural networks of depth $L \geq L_{\text{conv}}$ to be approximated directly by the inner product $(\mathbf{k}_{\text{NTK}})_*^T \mathbf{y}$ instead of $(\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} \mathbf{y}$.

**Theorem 1.3.1** (Convergence of the diagonal NTK to the exact NTK)**.** *Let* $M = (K_{\text{NTK}})_{11} \cdot I$ *be proportional to the* $n \times n$ *identity matrix. The error of the matrix inverse vanishes as* $\frac{||M - K_{\text{NTK}}^{-1}||}{||K_{\text{NTK}}^{-1}||} = O(1/n)$.

While the asymptotic error of $O(1/n)$ is the same as Theorem 1.2.3, the error caused by taking the sparse matrix inverse is *strictly less* than the error caused by the diagonal approximation due to the Gershgorin circle theorem applied to the matrix inverse inequality underlying Theorem 1.3.1.

**Theorem 1.3.2** (Sparse approximation vs. diagonal approximation)**.** *Given sparse matrix* $\tilde{K}_{\text{NTK}}$ *with at most* $O(\log n)$ *nonzero off-diagonal elements in every row and column, define* $\mathbb{E}[f_*^{\text{sparse}}] = (\mathbf{k}_{\text{NTK}})_*^T \tilde{K}_{\text{NTK}}^{-1} \mathbf{y}$. *Under the diagonal approximation, define* $\mathbb{E}[f_*^{\text{diag}}] = (\mathbf{k}_{\text{NTK}})_*^T \mathbf{y} / (K_{\text{NTK}})_{11}$. *Compared to the exact NTK* $\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} \mathbf{y}$ *in expectation over* $\mathbf{x}_*$*, we have* $|\mathbb{E}[f_*^{\text{sparse}}] - \mathbb{E}[f_*]| < |\mathbb{E}[f_*^{\text{diag}}] - \mathbb{E}[f_*]|$.

The same quantum algorithm as given above applies, except the solution $|v\rangle$ to the QLSP $\tilde{K}_{\text{NTK}} |v\rangle = |y\rangle$ is replaced simply by $|y\rangle$. Since the efficient state preparation and readout results hold, the diagonal approximation has asymptotically the same error as the sparsified NTK. However, the quantum speedup of the sparsified NTK is likely robust to improved classical algorithms, since the reliance on sparse matrix inversion ensures that it solves a BQP-complete problem. In contrast, it is possible that quantum-inspired approaches may approximate the diagonal NTK approximation through, for instance, Monte Carlo estimation of the inner product $(\mathbf{k}_{\text{NTK}})_*^T \mathbf{y}$.

## 1.4 Numerical experiments

We provide numerical experiments that support claims of the NTK's behavior, the favorable scaling for a quantum speedup, and the resulting performance of the NTK and its sparse and diagonal approximations on various toy and benchmark datasets. The output of the infinite-width neural network defined in Eq. 1.1 is evaluated with the `neural-tangents` package [43]. Additional analysis can be found in Sec. A.6 of the Appendix A.

To provide intuition for Assumption 1.1.3 and define a toy dataset on which empirical performance can be measured, we consider the dataset of $(\mathbf{x}_i, y_i)$ on a $d$-dimensional unit sphere. Noisy labels $y_i = f(\mathbf{x}_i) + \eta$ with $\eta \sim \mathcal{N}(0, 0.05)$

are assigned based on $f(\mathbf{x}_i) = \sum_{j=1}^{d} \sin \frac{3\pi(\mathbf{x}_i)_j}{2}$ for $\mathbf{x}_i = (x_1, \ldots, x_d)$. The $\epsilon$-neighborhood of Assumption 1.1.3 defines a minimum angular resolution on the sphere at which one can expect to find a single class; i.e., Assumption 1.1.3 sets the scale at which neighboring data examples can be assumed to belong to the same class with high probability (Fig. 1.1).

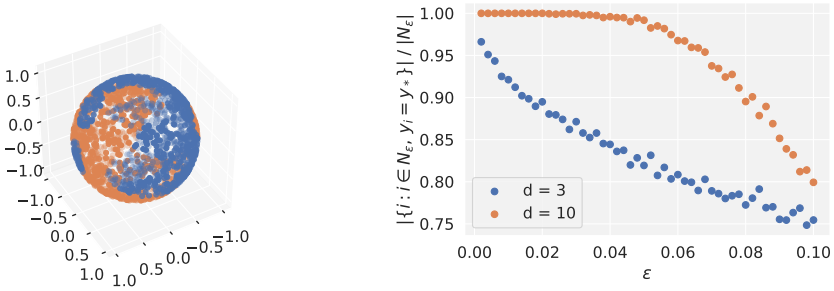



Figure 1.1: Left: true data distribution on a 3-dimensional sphere, where labels are assigned by sign $\sum_{j=1}^{d} \sin \frac{3\pi(\mathbf{x}_i)_j}{2}$. Right: fraction of examples in $N_\epsilon$ about random $y_*$ such that $y_i = y_*$; choosing $\epsilon = 0.01$ for the $d = 3$ dataset provides label homogeneity with probability $> 90\%$.

The uniform spherical dataset satisfies Assumption 1.1.2 that $\delta = \Omega(1/\text{poly } n)$; similarly, an empirical evaluation of the MNIST and CIFAR-10 datasets show that they also satisfy the assumption (see Sec. A.5 of the Appendix A). Hence, the NTK can be efficiently evaluated as a unitary applied to inner products.

To prepare the state $|k_*\rangle = \frac{1}{\sqrt{P}} \sum_{i=0}^{n-1} k_i |i\rangle$, post-selection requires $O(1/P)$ measurements as described in Theorem 1.2.5, where $P = \sum_{i=0}^{n-1} k_i^2$. To estimate the inner product ($\langle k_*| \tilde{K}_{\text{NTK}}^{-1} |y\rangle$ or $\langle k_*|y\rangle$) and compute the expected NTK output, the state overlap must scale at least like $\Omega(1/\text{polylog } n)$ to be measured efficiently. As seen in Fig. 1.2, both the state preparation and readout are efficient, consistent with the theoretical results.

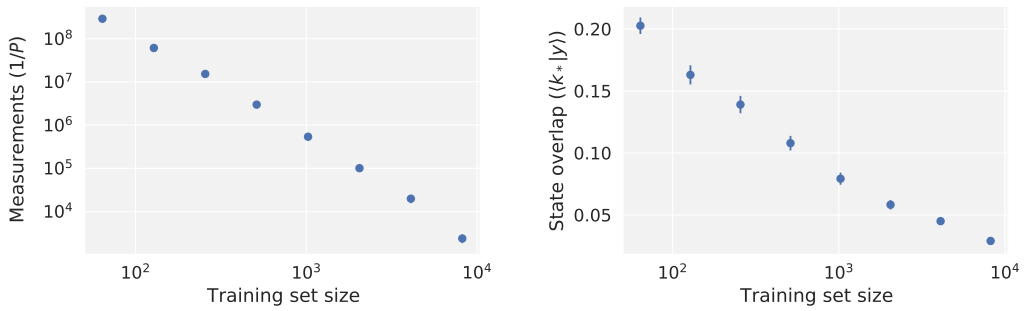



Figure 1.2: Scaling of the number of measurements required for state preparation (left) and of state overlap for efficient readout (right). As dataset size increases, the number of measurements required for state preparation *decreases* like $1/\text{poly}\ n$. Similarly, the state overlap $\langle k_* | y \rangle$ decreases like $\Omega(1/\text{polylog}\ n)$, ensuring that at most $O(\text{polylog}\ n)$ measurements are required for the final readout.

Hence, the quantum algorithm is confirmed to require at most a polylogarithmic number of measurements in training set size for state preparation and readout; this is sufficient for the diagonal NTK to achieve an exponential quantum speedup. In the case of the sparsified NTK approximation, the kernel is well-conditioned and constructed to be sparse, yielding a polylogarithmic cost of QLSA and also providing an exponential speedup.

Finally, we observe the performance of the exact and approximate NTKs on both the toy dataset and the MNIST dataset. While the toy dataset implements the neural network in Eq. 1.1, we also provide results on NTKs resembling more standard deep learning architectures to explore the generality of our result. For the MNIST dataset, we choose the non-residual convolutional Myrtle network [44] due to its straightforward architecture and use in previous benchmarks [45, 46]. The Myrtle NTK is also seen to be well-conditioned with rapidly vanishing off-diagonal elements (Sec. A.6 of the Appendix A), which suggests that it may provide the necessary properties for achieving a quantum speedup. As is expected, the performance of the NTK approximations approach the exact NTK rapidly, and the sparsified NTK has improved performance compared to the diagonal NTK approximation (Fig. 1.3).

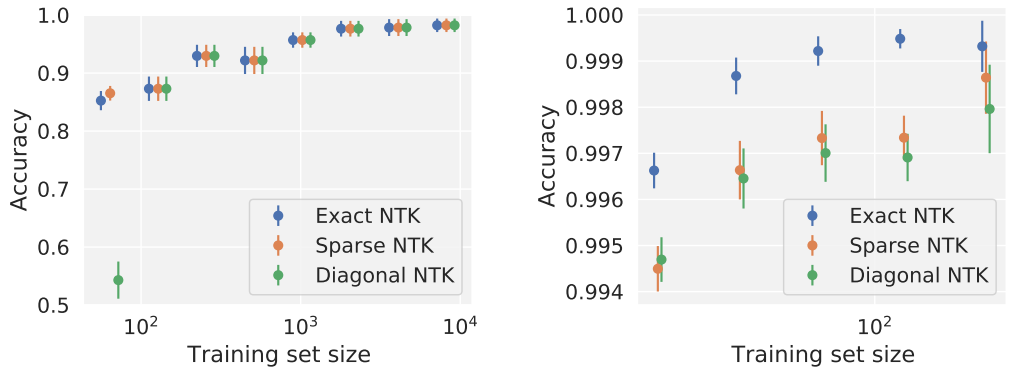



Figure 1.3: The exact NTK, sparsified NTK, and diagonal NTK on the 3-D spherical classification problem (left, Eq. 1.1) and MNIST 0/1 binary classification (right, Myrtle network [44]). The sphere NTK uses depth $L = L_{\text{conv}}/10$ while the MNIST NTK has a fixed depth with the Myrtle49 architecture. We expect the sparsified and diagonal approximations to converge to the exact NTK with error $O(1/n)$, and the sparsified NTK is expected to converge slightly faster. Error bars show two standard deviations.

## 1.5 Discussion

We propose the study of *approximate* neural tangent kernels (NTKs) — either sparsified or diagonal — corresponding to a wide fully-connected neural network architecture under standard data assumptions. In the regime of a neural network of sufficient depth to converge via gradient descent, we showed that these approximate NTKs converge to the exact output of a wide and deep neural network with $O(1/n)$ asymptotic error, where the sparsified NTK has strictly lower error than the diagonal NTK. Our main result is a quantum algorithm that provides the output of these approximate NTKs with an exponential speedup over the known classical algorithms, scaling logarithmically with the number of training examples. To the best of our knowledge, this represents the first quantum speedup for a classical feedforward neural network. Since sparse matrix multiplication is known to be BQP-complete and hence thought to be classically hard, we believe the result for the *sparsified NTK* is largely robust to further progress of classical algorithms. The quantum algorithm for the *diagonal NTK* also achieves an exponential speedup over current classical approaches, although quantum-inspired classical approaches may be possible; these may have implications for topics such as coresets [47, 48].

Although we only provide a theoretical treatment of a vanilla feedforward neural network, empirical results suggest successful applicability of the NTK

approximations to wide and deep convolutional deep learning architectures. Due to the versatility of the NTK framework across standard neural network architectures such as convolutional neural networks, graph neural networks, and transformers [45, 49–51], we anticipate future work studying quantum algorithms for additional architectures. Additionally, while our work focused on the neural tangent kernel, the approximation introduced by a sparsified or diagonal kernel may extend to any *chaotic kernel* [52, 53]. As the depth of a chaotic kernel increases, similar data entries become increasingly dissimilar due to random projections onto weight matrices; this may generally give rise to the vanishing off-diagonal elements that is key to kernel well-conditioning and successful approximation. Given the interest within the quantum machine learning community on kernel approaches due to the exponentially large Hilbert space offered by quantum computing [13, 54–58], this work may open new possibilities for improved machine learning methods amenable to quantum computing.

## References

[1]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor", Nature **574**, 505–510 (2019).

[2]A. Zlokapa, S. Boixo, and D. Lidar, *Boundaries of quantum supremacy via random circuit sampling*, 2020.

[3]J. Gray and S. Kourtis, *Hyper-optimized tensor network contraction*, 2020.

[4]C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, M. Szegedy, Y. Shi, and J. Chen, *Classical simulation of quantum supremacy circuits*, 2020.

[5]A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations", Phys. Rev. Lett. **103**, 150502 (2009).

[6]A. M. Childs, R. Kothari, and R. D. Somma, "Quantum algorithm for systems of linear equations with exponentially improved dependence on precision", SIAM Journal on Computing **46**, 1920–1950 (2017).

[7]S. Lloyd, M. Mohseni, and P. Rebentrost, "Quantum principal component analysis", Nature Physics **10**, 631–633 (2014).

[8]P. Rebentrost, A. Steffens, I. Marvian, and S. Lloyd, "Quantum singular-value decomposition of nonsparse low-rank matrices", Phys. Rev. A **97**, 012327 (2018).

[9]F. G. S. L. Brandao and K. M. Svore, "Quantum speed-ups for solving semidefinite programs", in 2017 ieee 58th annual symposium on foundations of computer science (focs) (2017), pp. 415–426.

[10]P. Rebentrost, M. Schuld, L. Wossnig, F. Petruccione, and S. Lloyd, "Quantum gradient descent and newton's method for constrained polynomial optimization", New Journal of Physics **21**, 073023 (2019).

[11]S. Aaronson, "Read the fine print", Nature Physics **11**, 291–293 (2015).

[12]S. Lloyd, M. Mohseni, and P. Rebentrost, *Quantum algorithms for supervised and unsupervised machine learning*, 2013.

[13]J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning", Nature **549**, 195–202 (2017).

[14]P. Rebentrost, M. Mohseni, and S. Lloyd, "Quantum support vector machine for big data classification", Phys. Rev. Lett. **113**, 130503 (2014).

[15]M. Schuld, I. Sinayskiy, and F. Petruccione, "Prediction by linear regression on a quantum computer", Phys. Rev. A **94**, 022342 (2016).

[16]I. Kerenidis, J. Landman, A. Luongo, and A. Prakash, "Q-means: a quantum algorithm for unsupervised machine learning", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 4134–4144.

[17]M. H. Amin, E. Andriyash, J. Rolfe, B. Kulchytskyy, and R. Melko, "Quantum boltzmann machine", Phys. Rev. X **8**, 021050 (2018).

[18]E. Farhi and H. Neven, *Classification with quantum neural networks on near term processors*, 2018.

[19]N. Killoran, T. R. Bromley, J. M. Arrazola, M. Schuld, N. Quesada, and S. Lloyd, "Continuous-variable quantum neural networks", Phys. Rev. Research **1**, 033063 (2019).

[20]S. Lloyd and C. Weedbrook, "Quantum generative adversarial learning", Phys. Rev. Lett. **121**, 040502 (2018).

[21]A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: convergence and generalization in neural networks", in Advances in neural information processing systems, Vol. 31, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (2018), pp. 8571–8580.

[22]K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015.

[23]C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", in Proceedings of the ieee conference on computer vision and pattern recognition (cvpr) (June 2015).

[24]K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in Proceedings of the ieee conference on computer vision and pattern recognition (cvpr) (June 2016).

[25]S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?", in Advances in neural information processing systems, Vol. 31, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (2018), pp. 2483–2493.

[26]N. Agarwal, P. Awasthi, and S. Kale, *A deep conditioning treatment of neural networks*, 2020.

[27]Y. Li and Y. Liang, "Learning overparameterized neural networks via stochastic gradient descent on structured data", in *Advances in neural information processing systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018), pp. 8157–8166.

[28]Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization", in Proceedings of the 36th international conference on machine learning, Vol. 97, edited by K. Chaudhuri and R. Salakhutdinov, Proceedings of Machine Learning Research (Sept. 2019), pp. 242–252.

[29]D. Zou and Q. Gu, "An improved analysis of training over-parameterized deep neural networks", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 2055–2064.

[30]J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 8572–8583.

[31]M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information: 10th anniversary edition*, 10th (Cambridge University Press, USA, 2011).

[32]D. Dervovic, M. Herbster, P. Mountney, S. Severini, N. Usher, and L. Wossnig, *Quantum linear systems algorithms: a primer*, 2018.

[33]E. Tang, "A quantum-inspired classical algorithm for recommendation systems", in Proceedings of the 51st annual acm sigact symposium on theory of computing, STOC 2019 (2019), pp. 217–228.

[34]I. Kerenidis and A. Prakash, "Quantum Recommendation Systems", in 8th innovations in theoretical computer science conference (itcs 2017), Vol. 67, edited by C. H. Papadimitriou, Leibniz International Proceedings in Informatics (LIPIcs) (2017), 49:1–49:21.

[35]I. Kerenidis, J. Landman, and A. Prakash, "Quantum algorithms for deep convolutional neural networks", in International conference on learning representations (2020).

[36]N.-H. Chia, A. Gilyén, T. Li, H.-H. Lin, E. Tang, and C. Wang, "Sampling-based sublinear low-rank matrix arithmetic framework for dequantizing quantum machine learning", in Proceedings of the 52nd annual acm sigact symposium on theory of computing, STOC 2020 (2020), pp. 387–400.

[37]E. Tang, *Quantum-inspired classical algorithms for principal component analysis and supervised clustering*, 2019.

[38]A. Gilyén, S. Lloyd, and E. Tang, *Quantum-inspired low-rank stochastic regression with logarithmic dependence on the dimension*, 2018.

[39]J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, tech. rep. (Carnegie Mellon University, USA, 1994).

[40]G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, "Quantum amplitude amplification and estimation", Contemporary Mathematics **305**, 53–74 (2002).

[41]N. Wiebe, A. Kapoor, and K. Svore, *Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning*, 2014.

[42]L. Zhao, Z. Zhao, P. Rebentrost, and J. Fitzsimons, *Compiling basic linear algebra subroutines for quantum computers*, 2019.

[43]R. Novak, L. Xiao, J. Hron, J. Lee, A. A. Alemi, J. Sohl-Dickstein, and S. S. Schoenholz, "Neural tangents: fast and easy infinite neural networks in python", in International conference on learning representations (2020).

[44]D. Page, *Myrtle.ai*, 2018.

[45]V. Shankar, A. Fang, W. Guo, S. Fridovich-Keil, J. Ragan-Kelley, L. Schmidt, and B. Recht, "Neural kernels without tangents", in Proceedings of the 37th international conference on machine learning, Vol. 119, edited by H. D. III and A. Singh, Proceedings of Machine Learning Research (13–18 Jul 2020), pp. 8614–8623.

[46]J. Lee, S. Schoenholz, J. Pennington, B. Adlam, L. Xiao, R. Novak, and J. Sohl-Dickstein, "Finite versus infinite neural networks: an empirical study", Advances in Neural Information Processing Systems **33** (2020).

[47]P. K. Agarwal, S. Har-Peled, K. R. Varadarajan, et al., "Geometric approximation via coresets", Combinatorial and computational geometry **52**, 1–30 (2005).

[48]A. W. Harrow, *Small quantum computers and large classical data sets*, 2020.

[49]J. Mairal, P. Koniusz, Z. Harchaoui, and C. Schmid, "Convolutional kernel networks", in Advances in neural information processing systems, Vol. 27, edited by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger (2014).

[50]S. S. Du, K. Hou, R. R. Salakhutdinov, B. Poczos, R. Wang, and K. Xu, "Graph neural tangent kernel: fusing graph neural networks with graph kernels", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019).

[51]G. Yang, *Tensor programs ii: neural tangent kernel for any architecture*, 2020.

[52]B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli, "Exponential expressivity in deep neural networks through transient chaos", in Advances in neural information processing systems, Vol. 29, edited by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (2016).

[53]J. Lee, J. Sohl-dickstein, J. Pennington, R. Novak, S. Schoenholz, and Y. Bahri, "Deep neural networks as gaussian processes", in International conference on learning representations (2018).

[54]M. Schuld and N. Killoran, "Quantum machine learning in feature hilbert spaces", Phys. Rev. Lett. **122**, 040504 (2019).

[55]V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta, "Supervised learning with quantum-enhanced feature spaces", Nature **567**, 209–212 (2019).

[56]C. Blank, D. K. Park, J.-K. K. Rhee, and F. Petruccione, "Quantum classifier with tailored quantum kernel", npj Quantum Information **6**, 41 (2020).

[57]S. Lloyd, M. Schuld, A. Ijaz, J. Izaac, and N. Killoran, *Quantum embeddings for machine learning*, 2020.

[58]M. Schuld, *Quantum machine learning models are kernel methods*, 2021.

*Chapter 2*

# QUANTUM GENERATIVE ADVERSARIAL NETWORKS

[1]M. Y. Niu*, A. Zlokapa*, M. Broughton, S. Boixo, M. Mohseni, V. Smelyanskyi, and H. Neven, *Quantum generative adversarial networks with provable convergence*, 2021.

[1]M. Broughton, G. Verdon, T. McCourt, A. J. Martinez, J. H. Yoo, S. V. Isakov, P. Massey, M. Y. Niu, A. Zlokapa, R. Halavati, E. Peters, M. Leib, A. Skolik, M. Streif, D. V. Dollen, J. R. McClean, S. Boixo, D. Bacon, A. K. Ho, H. Neven, and M. Mohseni, *Tensorflow quantum: a software framework for quantum machine learning*, in preparation.

(* indicates equal contribution.)

## 2.1 Overview

Generative adversarial networks (GANs) [1] are one of the most widely adopted *generative machine learning* methods, achieving state-of-the-art performance in a variety of high-dimensional and complex tasks including photorealistic image generation [2], super-resolution [3], and molecular synthesis [4]. Given access only to a training dataset $S = \{x_i\}$ sampled from an underlying data distribution $p_{\text{data}}(x)$, a GAN can generate realistic examples outside $S$. Certain probability distributions generated by quantum computers are thought to be classically hard to sample from under plausible conjectures [5–7], and learning to generate these samples using a classical GAN can also be formidably hard [8]. In this work, we focus on developing a fully quantum mechanical GAN, where the true data is given by a quantum state; the task is then to learn a *generator* circuit that can reproduce the same quantum state. Following the framework of a GAN, a *discriminator* circuit is presented either with the true data or with fake data from the generator. The generator and discriminator are then trained adversarially [9]: the generator attempts to fool the discriminator, while the discriminator attempts to correctly distinguish true and fake data. While we focus on quantum data, we present viable applications of the resulting machine learning architecture in the context of classical data.

Recent work on a quantum GAN (QuGAN) [10, 11] has proposed a direct analogy of the classical GAN architecture in designing the generator and discriminator circuits. We show that the proposed QuGAN does not always converge but rather in certain cases oscillates between a finite set of states due to mode collapse, and in general suffers from a non-unique Nash equilibrium. This motivates a new *entangling* quantum GAN (EQ-GAN) with a uniquely quantum twist: rather than providing the discriminator with *either* true *or* fake data, we allow the discriminator to entangle *both* true and fake data. We prove the convergence of the EQ-GAN to the global optimal Nash equilibrium. Numerical experiments confirm that the EQ-GAN converges on problem instances that the QuGAN failed on.

While the EQ-GAN has favorable convergence properties, the task of learning a quantum circuit to generate an unknown quantum state may also be solved in an entirely supervised approach. Rather than adversarially training the discriminator to distinguish between fake and real data, one may freeze the discriminator to perform an exact swap test, measuring the state fidelity between the true and fake data. While this would replicate the original state in the absence of noise, gate errors in the implementation of the discriminator will cause convergence to the incorrect optimum. We show that the adversarial approach of the EQ-GAN is more robust to such errors than the simpler supervised learning approach. Since training quantum machine learning models can require extensive time to compute gradients on current quantum hardware, resilience to gate errors drifting during the training process is especially valuable in the noisy intermediate-scale quantum (NISQ) era of quantum computing.

Other approaches to a quantum GAN may improve a quantum GAN's convergence properties — notably, recent work suggests that certain cost functions such as the Wasserstein metric may provide more robust convergence [12]. However, we find that the EQ-GAN's shallow discriminator is effective at suppressing device errors, making the EQ-GAN particularly relevant for near-term applications of quantum computing. Moreover, we demonstrate a proof-of-concept for learning a variational QRAM with the EQ-GAN, including an application in the broader context of quantum machine learning for classifying classical dataset.

## 2.2 Prior work

A GAN comprises of a parameterized generative network $G(\theta_g, z)$ and discriminator network $D(\theta_d, z)$. The generator maps a vector sampled from an input distribution $z \sim p_0(z)$ to a data example $G(\theta_g, z)$, thus transforming $p_0(z)$ to a new distribution $p_g(z)$ of fake data. The discriminator takes an input sample $x$ and gives the probability $D(\theta_d, x)$ that the sample is real (from the data) or fake (from the generative network). The training corresponds to a minimax optimization problem, where we alternate between improving the discriminator's ability to distinguish real/fake samples and improving the generator's ability to fool the discriminator. Specifically, we solve $\min_{\theta_g} \max_{\theta_d} V(\theta_g, \theta_d)$ for a cost function $V$:

$$\begin{aligned} V(\theta_g, \theta_d) = &\mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\log D(\theta_d, x)\right] \\ &+ \mathbb{E}_{z \sim p_0(z)} \left[\log \left(1 - D(\theta_d, G(\theta_g, z))\right)\right]. \end{aligned} \tag{2.1}$$

If $G$ and $D$ have enough capacity, i.e. approach the space of arbitrary functions, then it is proven in Ref. [1] that the global optimum of this minimax game exists and uniquely corresponds to $p_g(x) = p_{\text{data}}(x)$. While a multilayer perceptron can be used to parameterize $D$ and $G$, the dimensionality of the functional space can also be increased by replacing classical neural networks with quantum neural networks. In the most general case, the classical data can be represented by a density matrix $\sigma = \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right|$ where $p_i \in [0, 1]$ are positive bounded real numbers and $\left|\psi_i\right\rangle$ are orthogonal basis states. In the first proposal of a quantum GAN (QuGAN) [10, 11], the generative network is defined by a quantum circuit $U$ that outputs the quantum state $\rho = U(\theta_g)\rho_0 U^\dagger(\theta_g)$ from the initial state $\rho_0$. The discriminator takes either the real data $\sigma$ or the fake data $\rho$ and performs a positive operator valued measurement (POVM) defined by $T$ whose outcome determines the probability of data being true, or operator $F$ whose outcome determines the probability of data being fake, with $||T||_1, ||F||_1 \leq 1$. Hence, the discriminator predicts the probability that an unknown input state $\rho_{\text{in}}$ is true data by measuring the expectation value of $T$:

$$D(\theta_d, \rho_{\text{in}}) = \text{Tr}[T\rho_{\text{in}}]. \tag{2.2}$$

Following Ref. [11], the QuGAN solves the minimax game

$$\min_{\theta_g} \max_{T} \left(\text{Tr}[T\sigma] - \text{Tr}[T\rho(\theta_g)]\right) \ . \tag{2.3}$$

Unfortunately, minimax optimization might not converge to a good Nash equilibrium. When $\rho$ is close to $\sigma$, the optimal Hesltrom measurement operator $T = P^+(\sigma - \rho)$ is close to orthogonal to the true quantum data $\sigma$ and opposite to $\rho$. The next step of training will try align the generator state $\rho$ with $T$ to minimize the cost function in Eq. 2.3, perhaps overshooting $\sigma$. In the subsequent generator update, $T$ will again be opposite to $\rho$. This leads to the oscillation of the generator and discriminator, possibly preventing convergence; we show a case of infinite oscillation below.

### 2.3 Mode collapse example of QuGAN

We provide a concrete example of mode collapse in the original QuGAN architecture [10, 11]. Consider a true data state $\sigma$ and a generator initialized in state $\rho$, where each state is defined by

$$\sigma = \frac{1 + \cos(\pi/6)\sigma^x + \sin(\pi/6)\sigma^y}{2}, \tag{2.4}$$

$$\rho = \frac{1 + \cos(\pi/6)\sigma^x - \sin(\pi/6)\sigma^y}{2}. \tag{2.5}$$

Maximizing Eq. 2.3 with a Helstrom measurement by decomposing $\sigma - \rho = \frac{\sigma^y}{2}$, the discriminator will take $T = P^+(\sigma - \rho) = \frac{1+\sigma^y}{2}$. Optimizing over the space of density matrices, the generator will rotate $\rho$ to be parallel to $T$, also giving $\rho' = \frac{1+\sigma^y}{2}$. In the next iteration, the discriminator attempts to perform a new Helstrom measurement to distinguish $\sigma$ from $\rho'$, but this results in $T' = P^+(\sigma - \rho') = \rho$. As the generator realigns to match the new measurement operator, we find that $\rho'' = \rho$. It is now straightforward to see that if the QuGAN is trained to fully solve the minimax optimization problem each iteration, it will never converge; instead, it will always oscillate between states $\rho$ and $\rho'$, neither of which are the Nash equilibrium of the minimax game (Fig. 2.1) for the QuGAN performance under such mode-collapse.

More generally, we can consider the oscillation between a finite set of states. Let the function $T_\sigma(\rho) = P^+(\sigma - \rho)$ denote the optimal Helstrom measurement $P^+ = \sum_i \left|\phi_i^+\right\rangle \left\langle\phi_i^+\right|$ obtained from the positive part of the spectral decomposition of $\sigma - \rho$. If $T_\sigma^{(k)}$ is the $k$-fold composition of $T$ with itself, then the existence of some $k > 1$ such that $T^{(k)} = \rho$ is sufficient to ensure oscillation between $k$ states. For a system of $n$ qubits, we may achieve this by preparing the target and initial state separated by an angle of $\pi/3$ on the generalized Bloch sphere.

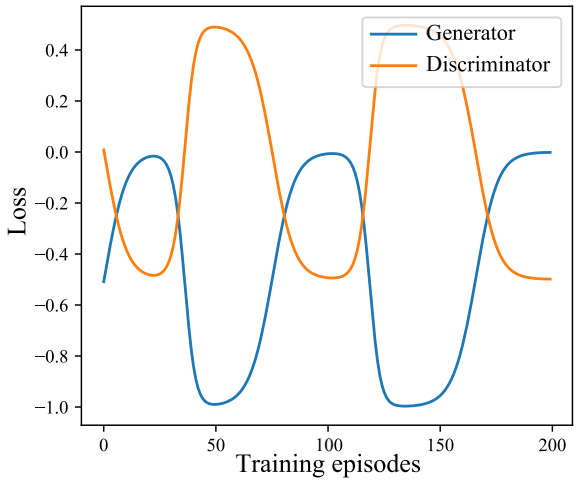



Figure 2.1: Performance of QuGAN [10, 11] learning the state defined in Eq. (2.4) with initialization given by Eq. 2.5. Mode collapse manifests as an oscillation in the generator and discriminator loss without converging to a global optimum. The implementation is based on the original architecture in Pennylane [13].

## 2.4 Convergence of EQ-GAN

To ensure convergence to a unique Nash equilibrium, we propose a new minimax optimization problem with a discriminator that is not directly analogous to the discriminator of a classical GAN. Rather than evaluating either fake or true data individually, the optimal discriminator is not only provided access to the true data $\sigma$ and an input state $\rho(\theta_g)$ prepared by the generator circuit parameterized by $\theta_g$, but also permited to perform a measurement on the joint system that in certain parameter value gives fidelity measurement between the two inputs:

$$D_\sigma^{\text{fid}}(\rho(\theta_g)) = \left(\text{Tr}\sqrt{\sigma^{1/2}\,\rho(\theta_g)\,\sigma^{1/2}}\right)^2. \tag{2.6}$$

Notice that in comparison Eq. 2.3 is a linear function of input states, which is not optimal in the state-certification problem [14] of evaluating quantum generative models. Let the discriminator $D_\sigma(\theta_d, \rho(\theta_g))$ represent the probability of measuring state $|0\rangle$ at the end of the discriminating circuit. If there exist parameters $\theta_d^{\text{opt}}$ that realize a perfect swap test, i.e. $D_\sigma(\theta_d^{\text{opt}}, \rho(\theta_g)) = \frac{1}{2} + \frac{1}{2} D_\sigma^{\text{fid}}(\rho(\theta_g))$, then $D_\sigma$ is sufficiently expressive to reach the optimal discriminator during optimization. Since a traditional swap test across two $n$-qubit

states requires two-qubit gates that span over $2n$ qubits, implementation on a quantum device with local connectivity incurs prohibitive overhead in circuit depth. Hence, we implement the discriminator with a parameterized destructive ancilla-free swap test [15]. The EQ-GAN architecture adversarially optimizes the generation of the state $\rho(\theta_g)$ and the learning of a fidelity measurement $D_\sigma$ (Fig. 2.2).

We define a minimax cost function closer to that of the classical GAN in Eq. 2.1:

$$\min_{\theta_g} \max_{\theta_d} V(\theta_g, \theta_d) = \min_{\theta_g} \max_{\theta_d} [1 - D_\sigma(\theta_d, \rho(\theta_g))], \tag{2.7}$$

where $D_\sigma(\theta_d, \rho(\theta_g))$ is the parameterization of the swap-test result. We now show that a *Nash equilibrium exists* at the desired location. Consider a swap test circuit ansatz for the discriminator $U(\theta_d) = \exp[-i\theta_d \text{CSWAP}]$, which is the matrix exponentiation of a perfect controlled swap gate with angle $\theta_d$. Under such ansatz, the input state $\rho_{\text{in}} = |\psi\rangle\langle\psi|$ and $\sigma = |\zeta\rangle\langle\zeta|$ will transform under the discriminator circuit into:

$$\begin{aligned} HU(\theta_d)H\,|0\rangle_{\text{a}}\,|\psi\rangle\,|\zeta\rangle = \frac{i\sin\theta_d}{2}\,|1\rangle_{\text{a}}\,[|\zeta\rangle\,|\psi\rangle - |\psi\rangle\,|\zeta\rangle] \\ + \frac{1}{2}\,|0\rangle_{\text{a}}\,[(e^{-i\theta_d} + \cos\theta_d)\,|\psi\rangle\,|\zeta\rangle - i\sin\theta_d\,|\zeta\rangle\,|\psi\rangle]. \end{aligned} \tag{2.8}$$

Given the circuit ansatz defined above with the predefined range for the swap angle $\theta$, the maximum value for distinguishing between two arbitrary states is uniquely achieved by perfect swap test angle $\theta = \pi/2$. More particularly, the probability of measuring state $|0\rangle$ at the end of the parameterized swap test depends on the swap angle $\theta$ according to

$$D_\sigma(\theta_d, \rho(\theta_g)) = \frac{1}{2}[1 + \cos^2\theta_d + \sin^2\theta_d D_\sigma^{\text{fid}}(\rho(\theta_g))]. \tag{2.9}$$

The discriminator aims to decrease the probablity of measuring $|0\rangle$, and thus minimize Eq. 2.9 by getting close to $\theta_d = \pi/2$ which corresponds to the perfect swap test given $D_\sigma^{\text{fid}}(\rho(\theta_g))) \leq 1$. The next step is for the generator to maximize $D_\sigma^{\text{fid}}(\rho(\theta_g)))$ by moving closer to the true data. Ultimately, the generator cannot improve when $\rho(\theta_g) = \sigma$.

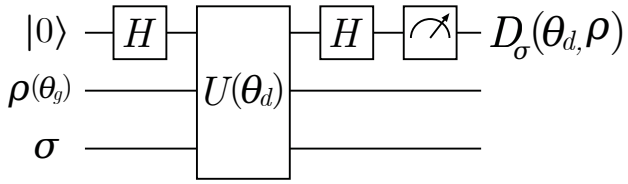



Figure 2.2: EQ-GAN architecture. The generator varies $\theta_g$ to fool the discriminator; the discriminator varies $\theta_d$ to distinguish the state. Since an optimal discriminator performs a swap test, the global optimum of the EQ-GAN occurs when $\rho(\theta_g) = \sigma$. While we include an ancilla qubit in the figure for clarity, we implement a destructive ancilla-free swap test [15].

The cost function defined in Eq. (2.7) does not assume that the input states $\sigma$ and $\rho(\theta_g)$ have to be pure states. For simplicity, the example we provided in Eq. (2.8) does assume pure state input. Below, we discuss a proposal for EQ-GAN to accommodate mixed state input by replacing the pure-state fidelity with a mixed state fidelity measurement. Other discriminator architectures may be chosen to ensure the existence of a Nash equilibrium. In the experiments presented below, we use a hardware-efficient ansatz designed to correct dominant coherent gate errors. Although a poorly chosen circuit parameterization may yield a non-convex loss function landscape and thus be difficult to optimize by gradient descent, this is an issue shared with the QuGAN due to the difficulty of expressing arbitrary unitaries as shallow quantum circuits as well as with classical GANs. However, the EQ-GAN architecture successfully converges on problem instances that are unreachable by a fully trained and properly parameterized a QuGAN.

We implement an ancilla-free swap test to perform state discrimination (Fig. 2.3). To evaluate the swap test on a Sycamore quantum device, we decompose each $CNOT$ gate into $(I \otimes H)CZ(I \otimes H)$ operations to use the native $CZ$ gate. As discussed in Sec. 2.5, the $CZ$ gate has unstable errors that can be effectively modeled with $Z$ rotations by an unknown angle on either qubit. The EQ-GAN formalism can overcome the single-qubit phase error by applying $RZ(\theta)$ gates directly after each $CZ$ operation. During adversarial training, the free angles $\theta$ are optimized with gradient descent to mitigate the two-qubit gate error. Due to the convergence properties provided by the generative adversarial framework, the discriminator provably converges to the best state discriminator possible. This motivates early stopping (as shown in Fig. 2.6) when the

discriminator loss indicates that the best state discriminator has been achieved.

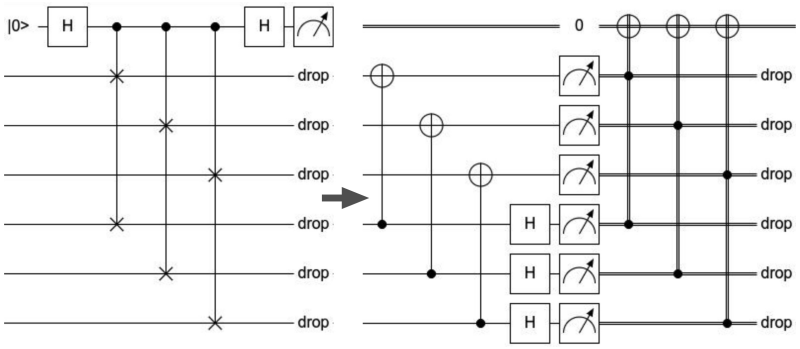



Figure 2.3: Ancilla-free swap test between two 3-qubit states. By rewriting the controlled-swap operation as CNOT and Toffoli gates and replacing computational basis operations with classical post processing, the swap test can be performed with an ancillary classical bit.

While classical GANs use a random latent vector to generate fake data, the quantum GAN proposed here and in the existing literature does not require any such random input. This comes with a price, especially when our goal is to learn quantum data in a mixed state. As shown in Fig. 2.4, a factor of two overhead in the number of qubits are needed for mixed-state learning based on Choi's theorem.

A closer look at the mathematical nature of a mixed state points us to a more efficient representation through a hybrid classical-quantum network. A mixed state represented in the most generic form $\rho = \sum_{i=1}^{2^n} P_i \left|\psi_i\right\rangle \left\langle\psi_i\right|$ is specified by a classical probability distribution $\{P_i\}$ over $2^n$ discrete variables corresponding to the set of quantum states $\{\left|\psi_i\right\rangle\}$ that diagonalize the density matrix. Naturally, one can efficiently represent the classical part of this representation, $\{P_i\}$, with a classical neural network, while a quantum circuit prepares the state $\left|\psi_i\right\rangle$ given parameter set $\theta_{g_i}$. In this way, we will be able to output a probabilistic mixture of the quantum state by sampling from $\{P_i\}$ and then prepare the associated state. This obviates the possible double exponential overhead in learning the full quantum channel that transform a fixed initial state to the desired mixed state, as illustrated in Fig. 2.4.

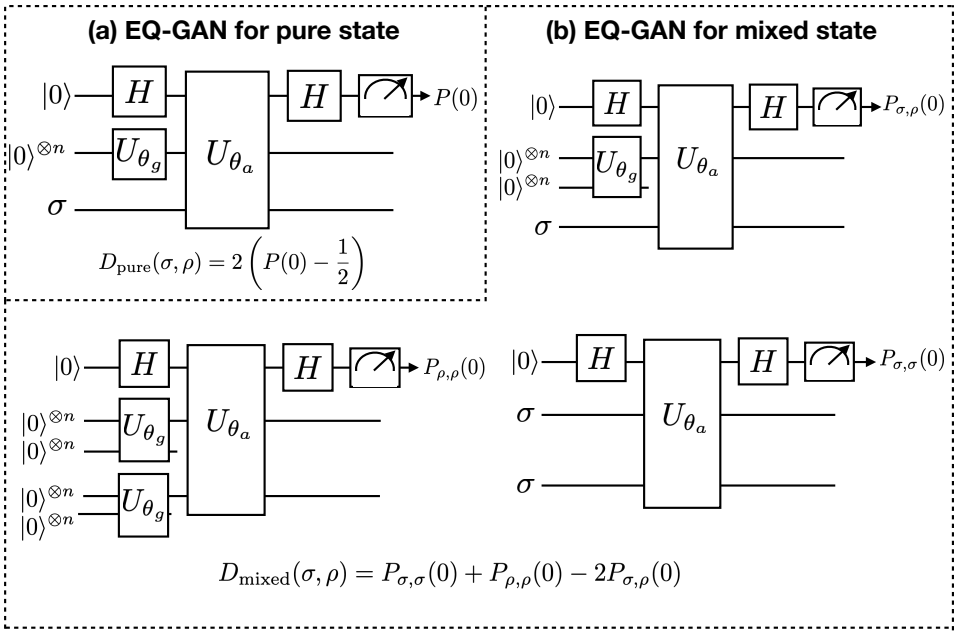



Figure 2.4: Diagram for EQ-GAN architecture based on quantum swap tests. (a) EQ-GAN for learning how to generate pure-state quantum data. (b) EQ-GAN for learning how to generate mixed-state quantum data. The true input data is represented by $\sigma$, and the fake input data $\rho$ is prepared by a unitary circuit $U_{\theta_g}$. The discriminator QNN realizes a unitary transformation represented by $U_{\vec{\theta}_a}$ jointly on the true data, fake data and the ancillary qubit. Measurement on the ancillary qubit is used for the cost function similarly to the EQ-GAN defined above.

## 2.5 Learning to suppress errors

We now show the improved robustness of an EQ-GAN against gate errors compared to a more straightforward supervised learning approach to learning an unknown quantum state. Rather than adversarially training the parameterized swap test used as a discriminator in EQ-GAN, a perfect swap test could be applied every iteration by a frozen discriminator. This may also cause the generator circuit to converge to the true data, since the swap test ensures a unique global optimum.

However, in the presence of gate errors in the swap test, this unique global optimum will be offset from the true data. Since EQ-GAN is agnostic to the precise parameterization of a perfect swap test, an appropriate ansatz can learn to correct coherent errors observed on near-term quantum hardware. In particular, the gate parameters such as conditional $Z$ phase, single qubit $Z$ phase and swap angles in two-qubit entangling gate can drift and oscillate

over the time scale of $O(10)$ minutes [16, 17]. Such unknown systematic and time-dependent coherent errors provides significant challenges for applications in quantum machine learning where gradient computation and update requires many measurements.

The large deviations in single-qubit and two-qubit $Z$ rotation angles can largely be mitigated by including additional single-qubit $Z$ phase compensations. The effectiveness and importance of such systematic error mitigation is recently demonstrated in the success of achieving the-state-of-art accuracy in energy estimation for fermionic molecules [18]. In learning the discriminator circuit that is closest to a true swap test, the adversarial learning of EQ-GAN provides a useful paradigm that may be broadly applicable to improving the fidelity of other near-term quantum algorithms.

Suppose the adversarial discriminator unitary is given by $U(\theta_d)$, where $U(\theta_d^{\text{opt}})$ corresponds to a perfect swap test in the absence of noise. Given a trace-preserving completely positive noisy channel $\mathcal{E}$, the discriminator is replaced by a new unitary operation $\tilde{U}(\theta_d)$. While a supervised approach would apply an approximate swap test given by $\tilde{U}(\theta_d^{\text{opt}})$, the adversarial swap test will generically perform better if there exist parameters $\theta_d^*$ such that $||\tilde{U}(\theta_d^*) - U(\theta_d^{\text{opt}})||_2 < ||\tilde{U}(\theta_d^{\text{opt}}) - U(\theta_d^{\text{opt}})||_2$. Because the discriminator defines the loss landscape optimized by the generator, the $\rho(\theta_g)$ produced by EQ-GAN may converge to a state closer to $\sigma$ than possible by a supervised approach if the parameterization of the noisy unitary $\tilde{U}$ is general enough to mitigate errors.

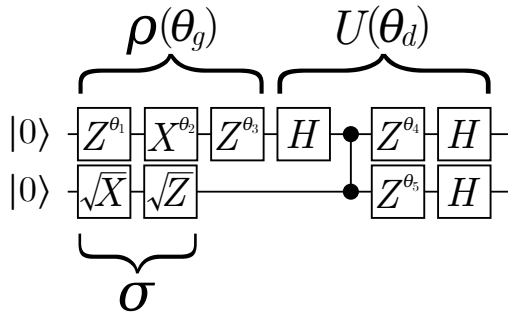



Figure 2.5: EQ-GAN experiment for learning a single-qubit state. The discriminator ($U(\theta_d)$ is constructed with free $Z$ rotation angles to suppress CZ gate errors, allowing the generator $\rho(\theta_g)$ to converge closer to the true data state $\sigma$ by varying $X$ and $Z$ rotation angles.

As an example, we consider the task of learning the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ on a quantum device with noise (Fig. 2.5). The discriminator is defined by a swap test with CZ gate providing the necessary two-qubit operation. To learn to correct gate errors, however, the discriminator adversarially learns the angles of single-qubit $Z$ rotations insert directly after the CZ gate. Hence, the EQ-GAN obtains a state overlap significantly better than that of the perfect swap test (Fig. 2.6). Although both methods do not stay at the optimal point, this is typical of noisy gradient measurements and minimax optimization:after convergence to the Nash equilibrium, discretization can induce perturbations while non-zero higher-order gradients lead the training to deviate from the global optimum [19].

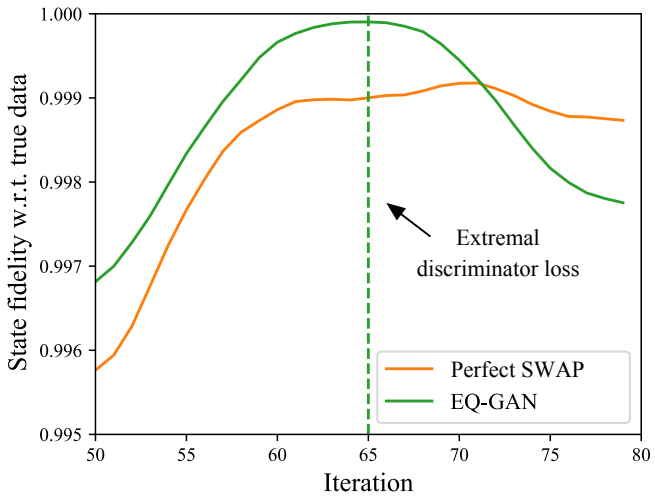



Figure 2.6: Comparison of EQ-GAN and a supervised learner (perfect swap test) on a physical quantum device. We experimentally confirm that the EQ-GAN converges to a higher state overlap by learning to correct such errors with additional single-qubit rotations. The "converged" EQ-GAN (dashed line) coincides with the iteration where the discriminator loss is minimized.

We report the average error after multiple runs of the EQ-GAN and supervised learner on an experimental device (Table 2.1).

| QML model | Minimum error in state fidelity |
|---|---|
| Supervised learner | $(2.4 \pm 0.5) \times 10^{-4}$ |
| **EQ-GAN** | $\mathbf{(0.6 \pm 0.2) \times 10^{-4}}$ |

Table 2.1: Comparison of EQ-GAN and a supervised learner (perfect swap test) on a Sycamore quantum device. The error of the EQ-GAN (i.e. $1 -$ state fidelity) is significantly lower than that of the supervised learner, demonstrating the successful adversarial training of an error-suppressed swap test. Uncertainties show two standard deviations.

## 2.6 Training EQ-GAN

While the original QuGAN architecture is shown to oscillate indefinitely for an example constructed in Fig. 2.1, we provide numerical experiments here to demonstrate the successful convergence of the proposed EQ-GAN architecture.

We illustrate a subtlety in the oscillatory analysis presented above. Within the GAN formalism, the generator and discriminator iteratively optimize a given loss function. When the optimization is allowed to converge to an extremum of the loss function in the QuGAN architecture specifically, the result is determined by a Helstrom measurement. It is for this case that indefinite oscillation is shown; in the case of learning the state $\sigma$ constructed above, oscillation between states $\rho$ and $\rho'$ result in a constant state overlap of 3/4.

However, the iterative optimization procedure to move towards the optimal Helstrom measurement may be only partly completed, i.e. the generator and discriminator are not allowed to extremize the loss function. With such a selection of hyperparameters, we observe that oscillation between states continues (Fig. 2.7), leading to unstable training for the QuGAN architecture. In comparison, the same hyperparameters perform well for the EQ-GAN architecture, which steadily approaches the true data state. Unstable training is difficult to overcome even in classical GAN architectures [20], and thus advances in understanding how to prevent such non-convergence are consequential for both quantum and classical machine learning.

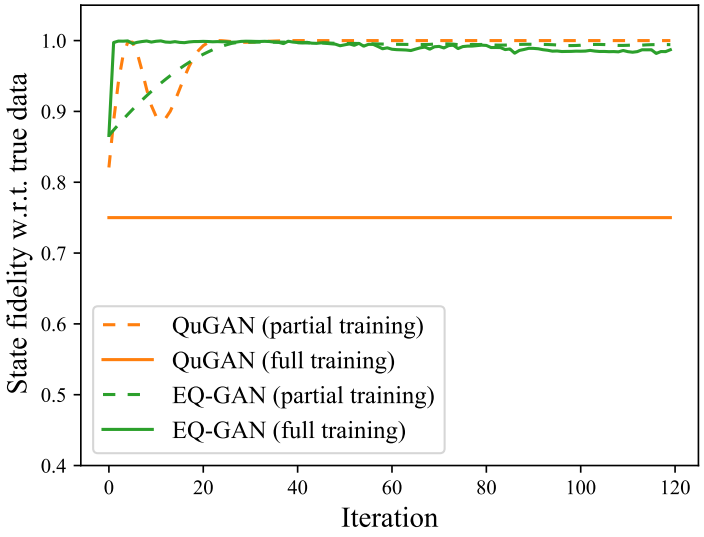



Figure 2.7: Comparison of QuGAN [10, 11] and EQ-GAN learning the state given by Eq. 2.4. *Full training* denotes training the generator for 50 epochs then the discriminator for 50 epochs each iteration; *partial training* denotes only 1 epoch per iteration. The QuGAN remains more unstable than EQ-GAN during training with either training configuration.

To help ensure stable training of the EQ-GAN architecture, we introduce a training procedure that capitalizes on the fact that the discriminator must converge to a swap test at the optimal Nash equilibrium. Rather than training both the generator and discriminator from the beginning, we pre-train the EQ-GAN in a supervised setting. In the first phase, the discriminator is frozen with the parameters of a perfect swap test, although the unitary $\tilde{U}(\theta_d^{\text{opt}})$ may be an imperfect swap test; the generator is trained until the loss converges. In the second phase of training, the discriminator is allowed to vary adversarially against the generator, seeking the parameters $\theta_d^*$. In the context of gate errors, this second phase may yield a unitary closer to a true swap test. The example shown in Fig. 2.6 on a physical quantum devices is replicated in Fig. 2.8 here, showing the two phases of training and the benefit of an adversarial swap test in the presence of noise.

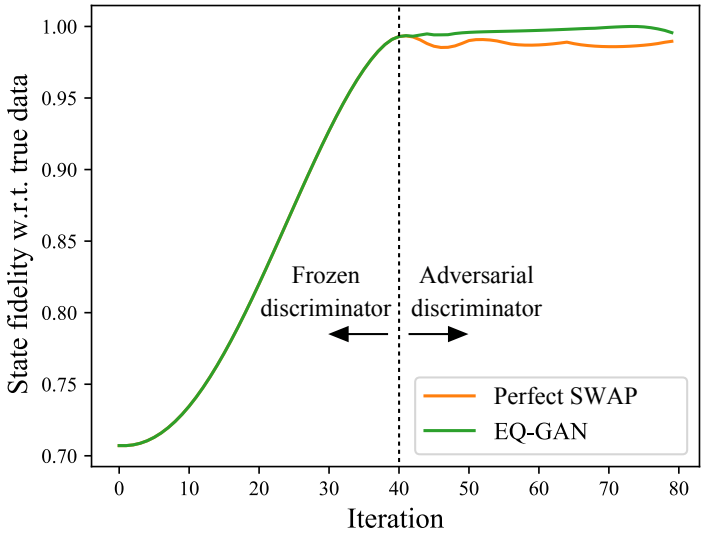



Figure 2.8: Comparison of EQ-GAN and a supervised learner for a simulated noise model. Normally distributed noise on single-qubit rotations are applied with a systematic bias away from zero, causing the discriminator of the supervised learner to force convergence to the incorrect state.

We provide additional motivation for using adversarial learning in the *noiseless* case. In particular, we construct an example for which supervised learning fails and adversarial learning successfully generates the true data state.

Given the generator ansatz shown in Fig. 2.9, define the data state to have angles $\alpha_0 = \beta_0 = \pi/2$ for corresponding rotations $R_x(\alpha_0), R_z(\beta_0)$. The generator then optimizes angles $\alpha, \beta$ towards achieving full state overlap. In general, the gradient of the state overlap is $\frac{\pi}{4}\sqrt{2 - 2\cos(2\pi\alpha)\cos(2\pi\beta)}$. By initializing the generator with $\alpha = \beta = 0$, the gradient and all higher derivatives of the overlap vanish. Since a noiseless supervised learning approach with a perfect swap test can only evaluate the gradient of a state overlap measurement, gradient descent will fail to converge to the correct values.

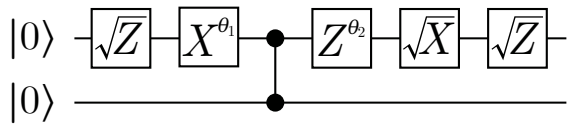



Figure 2.9: Generator and data circuit with a vanishing gradient given data defined by $X$ and $Z$ rotations of $\pi/2$ and a generator initialized with zero angles.

On the other hand, by allowing the discriminator to change, the issue of a vanishing gradient is circumvented and the generator learns the data state (Fig. 2.10). For simplicity, we use the same discriminator architecture as that used for suppressing errors. Parameters are optimized with vanilla stochastic gradient descent. The EQ-GAN learning rate schedule is manually tuned, and we verify that no selection of learning rate allows the supervised learner to converge.

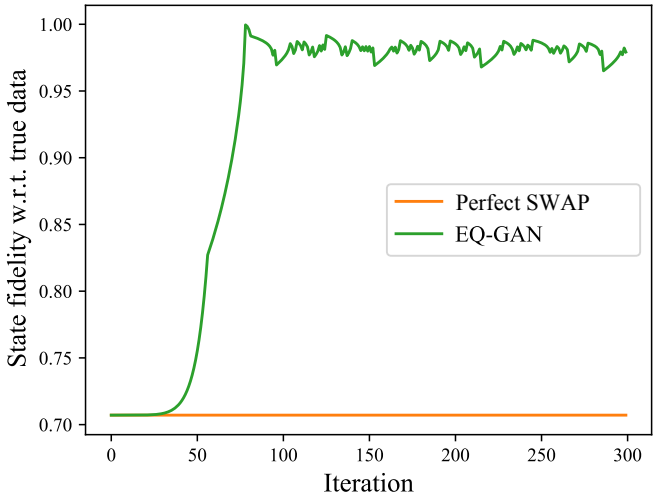
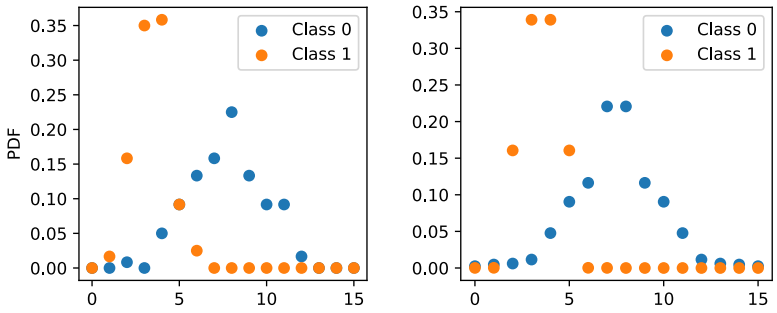



Figure 2.10: Demonstration of a vanishing gradient for a supervised learner and convergence for the EQ-GAN. While the supervised learner cannot be trained by gradient descent, the EQ-GAN achieves a state overlap of 0.97.

## 2.7 Application to QRAM

Many quantum machine learning applications require a quantum random access memory (QRAM) to load *classical* data in superposition [21]. More particularly, a set of classical data can be described by the empirical distribution $\{P_i\}$ over all possible input data $i$. Most quantum machine learning algorithms require the conversion from $\{P_i\}$ into a quantum state $\sum_i \sqrt{P_i}\,|\psi_i\rangle$, i.e. a superposition of orthogonal basis states $|\psi_i\rangle$ representing each single classical data entry with an amplitude proportional to the square root of the classical probability $P_i$. Preparing such a superposition of an arbitrary set of $n$ states takes $O(n)$ operations at best, which ruins the exponential speedup. Given a suitable ansatz, we may use an EQ-GAN to learn a state approximately equivalent to the superposition of data. To demonstrate a variational QRAM, we consider a dataset of two peaks sampled from different Gaussian distributions

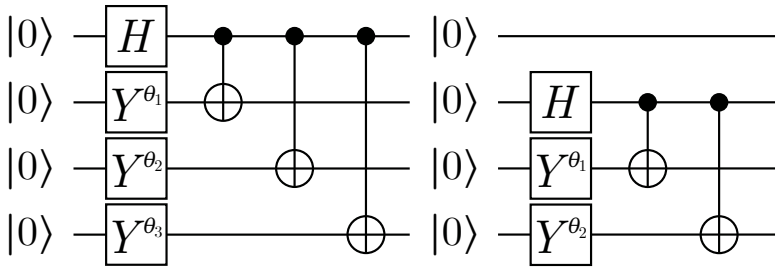
(Fig. 2.11).




(a) Empirical PDF (b) Variational QRAM

Figure 2.11: Two-peak total dataset (sampled from normal distributions, $N = 120$) and variational QRAM of the training dataset ($N = 60$). The variational QRAM is obtained by training an EQ-GAN to generate a state $\rho$ with the shallow peak ansatz to approximate an exact superposition of states $\sigma$. The training and test datasets (each $N = 60$) are both balanced between the two classes.

Exactly encoding the empirical probability density function requires a very deep circuit and multiple-control rotations; similarly, preparing a Gaussian distribution on a device with planar connectivity requires deep circuits. Hence, we select shallow circuit ansatzes (Fig. 2.12) that generate concatenated exponential functions to approximate a symmetric peak [22]. Once trained to approximate the empirical data distribution, the variational QRAM closely reproduces the original dataset.




(a) Class 0 ansatz (b) Class 1 ansatz

Figure 2.12: Variational QRAM ansatzes for generating peaks by learning $\theta_i$ parameters [22]. Class 0 corresponds to a centered peak, and Class 1 corresponds to an offset peak.

To use the native $CZ$ two-qubit gate, we implement a rank-4 entangling gate $G$ given by

$$G(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{-i\theta} & 0 & 0 \\ 0 & 0 & e^{-i\theta} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.10}$$

which is decomposed as shown in Fig. 2.13.

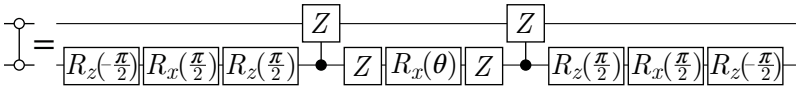



Figure 2.13: Decomposition of the two-qubit entangling gate $G(\theta)$ used in the QNN ansatz (Eq. 2.10).

Due to the planar connectivity of a Sycamore quantum device, we implement the QNN shown in Fig. 2.14 with a four-qubit data state. The QNN is trained in two ways: it is either trained via *sampling* (shown one training example each iteration, as in Ref. [23]) or via *superposition* (shown a superposition over an entire class each iteration), where the superposition methodology does not use an exact superposition of the training dataset. Instead, it uses a shallow approximation obtained by pre-training an EQ-GAN. We prepare a symmetric concatenation of exponential functions to approximate a peak with minimal circuit depth. In comparison, preparing a Gaussian distribution over $n$ qubits requires $(n-1)$-controlled rotations, which must be decomposed into $2^{n-1}$ $CZ$ gates to use the native gate basis (see Fig. 10 of [22]); additional swap operations are required to prepare the state on a planar architecture. Given the empirical dataset, we may also prepare an exact superposition of the data following a state preparation procedure such as that proposed in Ref. [24]. However, this also requires $n$-controlled rotations, leading to an exponential dependence in the number of qubits. All three versions of the QRAM are shown in Fig. 2.15.

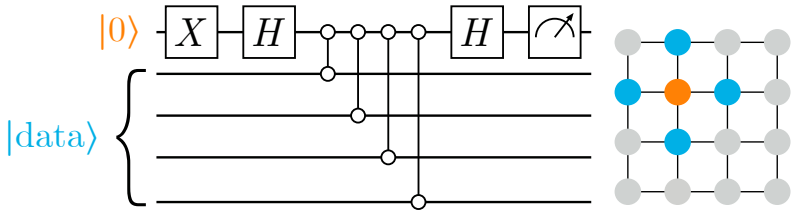



Figure 2.14: Quantum neural network architecture (left) and its corresponding layout on the Sycamore device (right). A four-qubit data state is constructed with the circuits shown in Fig. 2.12 and placed in the $|\text{data}\rangle$ state on the blue qubits. A readout qubit (orange) performs parameterized two-qubit interactions shown in Fig. 2.13.

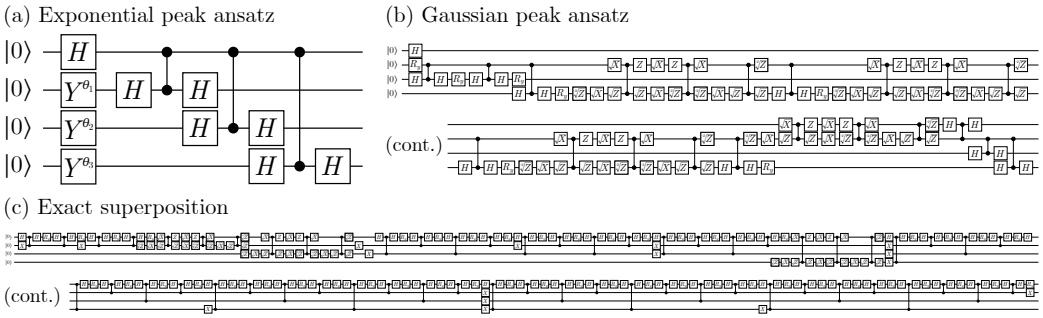



Figure 2.15: QRAM ansatzes for $n = 4$ qubits in planar connectivity with (a) exponential peaks (3 two-qubit gates), (b) Gaussian distribution (21 two-qubit gates), and (c) exact superposition (57 two-qubit gates). We adopt ansatz (a) because circuit depth scales polynomially for a QRAM with $n$ qubits, while (b) and (c) scale exponentially with $n$.

To ensure a fair comparison, we permit an equal number of queries to the quantum device. Consequently, for $N = 60$ examples with 30 examples per class, training via sampling is performed for 1 epoch with 60 corresponding to 60 iterations performed on the quantum device. However, training via superposition evaluates the superposition of each class 30 times (since there are two classes), also accessing the quantum device for 60 iterations.

Additionally, Bayesian optimization is used to tune different learning rates for the sampling and superposition methodologies. In simulation, we optimize over Adam learning rates from $10^{-4}$ to $10^{-1}$ with 10 random parameter tries and 40 evaluations of the Gaussian process estimator. For each parameter query, the output of the QNN is averaged over 10 trials to reduce any statistical fluctuations. QNNs using the final learning rates ($10^{-3.93}$ for sampling and

$10^{-1.83}$ for superposition) are then evaluated over 50 trials to obtain the final performance reported in Table 2.2 with computed standard deviations.

| Training data | Accuracy |
|---|---|
| Exact sampling | $53\% \pm 6\%$ |
| **Variational QRAM** | $\mathbf{69\% \pm 2\%}$ |

Table 2.2: Test accuracy ($N = 60$) of a quantum neural network (QNN) either trained on the all samples of the training dataset ($N = 60$) for a single epoch or trained on the variational QRAM for an equal number of circuit evaluations. Although the QNN trained on the variational QRAM did not have direct access to the original dataset, accuracy is evaluated on the raw dataset. Uncertainties show two standard deviations.

As a proof of principle for using such QRAM in a quantum machine learning context, we train a quantum neural network [23] and compute hinge loss either by considering each data entry individually (encoded as a quantum circuit) or by considering each class individually (encoded as a superposition in variational QRAM). Given the same number of circuit evaluations to compute gradients, the superposition converges to a better accuracy at the end of training despite using an approximate distribution (Table 2.2).

## 2.8 Discussion

Motivated by limitations of preexisting quantum GAN architectures in the literature, we propose the EQ-GAN architecture to overcome issues of non-convexity and mode collapse. We adopt a parameterization of Hilbert-Schmidt norm as the cost function as oppose to trace distance based on the optimality of Hilbert-Scmidt norm in state-certification problems. Similar advantages of Hilbert-Schmidt norm has been shown in quantum embedding design of quantum kernel learning [25]. Other approaches to a quantum GAN may improve a quantum GAN's convergence properties — notably, recent work suggests that certain cost functions such as the Wasserstein metric may provide more robust convergence [12]. However, we find that the EQ-GAN's shallow discriminator is effective at suppressing device errors, making the EQ-GAN particularly relevant for near-term applications of quantum computing. Moreover, we demonstrate a proof-of-concept for learning a variational QRAM with the EQ-GAN, including an application in the broader context of quantum machine learning for classifying classical dataset.

## References

[1]I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets", in Advances in neural information processing systems (2014), pp. 2672–2680.

[2]T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen, "Improved techniques for training gans", in Advances in neural information processing systems, Vol. 29, edited by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (2016), pp. 2234–2242.

[3]C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network", in Proceedings of the ieee conference on computer vision and pattern recognition (cvpr) (July 2017).

[4]E. Putin, A. Asadulaev, Y. Ivanenkov, V. Aladinskiy, B. Sanchez-Lengeling, A. Aspuru-Guzik, and A. Zhavoronkov, "Reinforced adversarial neural computer for de novo molecular design", Journal of Chemical Information and Modeling **58**, PMID: 29762023, 1194–1204 (2018).

[5]S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, J. M. Martinis, and H. Neven, "Characterizing quantum supremacy in near-term devices", arXiv preprint arXiv:1608.00263 (2016).

[6]S. Aaronson and A. Arkhipov, "The computational complexity of linear optics", in Proceedings of the forty-third annual acm symposium on theory of computing, STOC '11 (2011), pp. 333–342.

[7]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor", Nature **574**, 505–510 (2019).

[8]M. Y. Niu, A. M. Dai, L. Li, A. Odena, Z. Zhao, V. Smelyanskyi, H. Neven, and S. Boixo, "Learnability and complexity of quantum samples", arXiv preprint arXiv:2010.11983 (2020).

[9]C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks", arXiv preprint arXiv:1312.6199 (2013).

[10]P.-L. Dallaire-Demers and N. Killoran, "Quantum generative adversarial networks", Phys. Rev. A **98**, 012324 (2018).

[11]S. Lloyd and C. Weedbrook, "Quantum generative adversarial learning", Phys. Rev. Lett. **121**, 040502 (2018).

[12]B. T. Kiani, G. D. Palma, M. Marvian, Z.-W. Liu, and S. Lloyd, *Quantum earth mover's distance: a new approach to learning quantum data*, 2021.

[13]V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri, K. McKiernan, J. J. Meyer, Z. Niu, A. Száva, and N. Killoran, *Pennylane: automatic differentiation of hybrid quantum-classical computations*, 2020.

[14]C. Bădescu, R. O'Donnell, and J. Wright, "Quantum state certification", in Proceedings of the 51st annual acm sigact symposium on theory of computing (2019), pp. 503–514.

[15]J. C. Garcia-Escartin and P. Chamorro-Posada, "Swap test and hong-ou-mandel effect are equivalent", Phys. Rev. A **87**, 052330 (2013).

[16]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, et al., "Supplementary information for" quantum supremacy using a programmable superconducting processor"", arXiv:1910.11333 (2019).

[17]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, A. Bengtsson, S. Boixo, M. Broughton, B. B. Buckley, et al., "Observation of separated dynamics of charge and spin in the fermi-hubbard model", arXiv preprint arXiv:2010.07965 (2020).

[18]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, S. Boixo, M. Broughton, B. B. Buckley, D. A. Buell, B. Burkett, N. Bushnell, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, S. Demura, A. Dunsworth, E. Farhi, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, S. Habegger, M. P. Harrigan, A. Ho, S. Hong, T. Huang, W. J. Huggins, L. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, C. Jones, D. Kafri, K. Kechedzhi, J. Kelly, S. Kim, P. V. Klimov, A. Korotkov, F. Kostritsa, D. Landhuis, P. Laptev, M. Lindmark, E. Lucero, O. Martin, J. M. Martinis, J. R. McClean, M. McEwen, A. Megrant, X. Mi, M. Mohseni, W. Mruczkiewicz, J. Mutus, O. Naaman, M. Neeley, C. Neill, H. Neven, M. Y. Niu, T. E. O'Brien, E. Ostby, A. Petukhov, H. Putterman, C. Quintana, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, D. Strain, K. J. Sung, M. Szalay, T. Y. Takeshita, A. Vainsencher, T. White, N. Wiebe, Z. J. Yao, P. Yeh, and A. Zalcman, "Hartree-fock on a superconducting qubit quantum computer", Science **369**, 1084–1089 (2020).

[19]A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis", arXiv preprint arXiv:1809.11096 (2018).

[20]L. Mescheder, A. Geiger, and S. Nowozin, "Which training methods for GANs do actually converge?", in Proceedings of the 35th international conference on machine learning, Vol. 80, edited by J. Dy and A. Krause, Proceedings of Machine Learning Research (Oct. 2018), pp. 3481–3490.

[21]J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning", arXiv:1611.09347 (2016).

[22]N. Klco and M. J. Savage, "Minimally entangled state preparation of localized wave functions on quantum computers", Phys. Rev. A **102**, 012612 (2020).

[23]E. Farhi and H. Neven, *Classification with quantum neural networks on near term processors*, 2018.

[24]S. Aaronson, *The complexity of quantum states and transformations: from quantum money to black holes*, 2016.

[25]S. Lloyd, M. Schuld, A. Ijaz, J. Izaac, and N. Killoran, *Quantum embeddings for machine learning*, 2020.

*Chapter 3*

# NEAR-TERM QUANTUM SIMULATION OF WORMHOLE TELEPORTATION

[1]A. Zlokapa, J. Lykken, S. Davis, D. Jafferis, and M. Spiropulu, *Near-term quantum simulation of wormhole teleportation*, in preparation.

[2]A. Zlokapa, R. Babbush, and H. Neven, *Shallow variational circuits for time evolution*, in preparation.

## 3.1 Introduction

While quantum computers have been widely suggested as a tool to explore quantum systems through simulation beyond the classical regime [1, 2], it may be possible to also probe gravitational theories [3, 4]. In particular, *traversable wormholes* may arise from the holographic principle realized in the AdS/CFT correspondence [5] despite violating the notion from general relativity that a signal cannot be sent more quickly inside instead of outside a wormhole [6]. The *thermofield double state* is an entangled pure state between two copies of any quantum mechanical system such that each of the two copies is in the thermal density matrix with given temperature $1/\beta$ [7]. Considering gravity with AdS-like boundary conditions as the dual to a quantum system, the thermofield double state between two boundary CFTs is dual to an AdS-Schwarzschild wormhole [6]. By coupling the two quantum systems with an interaction, it is possible to send information from one system to the other [8]. This perturbative coupling between either side of the eternal AdS black hole allows the system to be probed behind the horizon without issues of bulk locality by examining the experience of the information that passes through the wormhole. Hence, the traversable wormhole provides a mechanism to understand the ER=EPR relation between entanglement and geometry [9]. Although the generality of the conjecture remains unclear, it has been suggested that quantum computers can aid exploration of the topic through simulations of quantum gravity [10].

We consider a recently proposed protocol [11] to realize wormhole teleportation with a Majorana SYK model [12, 13]. Two black holes, $L$ and $R$, are

prepared in the thermofield double state, and a qubit is teleported from $L$ to $R$ by traveling through the dual wormhole. Rather than simply measuring the teleportation fidelity, the system must be characterized in terms of the *causal propagator* to distinguish it from other teleportation mechanisms: the key feature of a traversable wormhole is the interaction bringing the left and right sides of the black hole into causal contact. Achieving wormhole teleportation is thus more difficult than using random unitary dynamics to achieve teleportation [14], which experiences time inversion of transmitted quantum information. The work of Gao and Jafferis [11] provides a concrete teleportation protocol amenable to implementation on a quantum computer using an $N$-qubit fermionic system, leading to perfect teleportation in the infinite $N$ limit. While understanding the protocol for finite $N$ is difficult via classical simulation, the large Hilbert space of a quantum computer makes quantum simulation a better tool to probe such wormhole teleportation.

In this work, we propose various procedures for realizing the wormhole teleportation protocol on a near-term quantum device. We study a *low-rank* SYK model built with Dirac fermions [15, 16] to improve the efficiency of Trotterization to perform time evolution. Additionally, variational methods for learning approximate quantum circuits are used to aid the simulatino. Besides preparing the thermofield double state as the ground state of a known Hamiltonian using a variational quantum eigensolver[17, 18], we propose a new approach to actively learn a shallow circuit that performs time evolution with high fidelity. The method of *compressed Trotterization* promises a smooth optimization landscape that avoids barren plateaus associated with variational quantum circuits [19] and is shown to be applicable across different Hamiltonian systems. Finally, we develop efficient classical simulations to provide estimates of quantum circuit sizes required to observe wormhole teleportation with finite $N$, suggesting that it may be achievable in the noisy intermediate-scale quantum (NISQ) era of quantum computing [20].

### 3.2 Wormhole teleportation

We describe the original wormhole teleportation protocol proposed by Gao and Jafferis [11] using the Majorana SYK model. Briefly, two SYK models of $N$ fermions each are entangled in a thermofield double state between two CFTs (i.e. a two-sided eternal black hole in the AdS picture). At time $t = -t_0$, a qubit is swapped into the left system: this is the message being teleported.

By $t = 0$, the chaotic system has fully scrambled the information across the system. At this point in time, we apply a coupling term between the left and right sides of the wormhole; if suitably chosen, this generates negative null energy in the bulk. When the message hits the shockwave, it receives a time *advance* instead of a time delay, causing the qubit to shift downwards in time instead of being irretrievably lost in the singularity. At time $t = t_1$, we can swap out the qubit from the right side of the wormhole, recovering the original message.

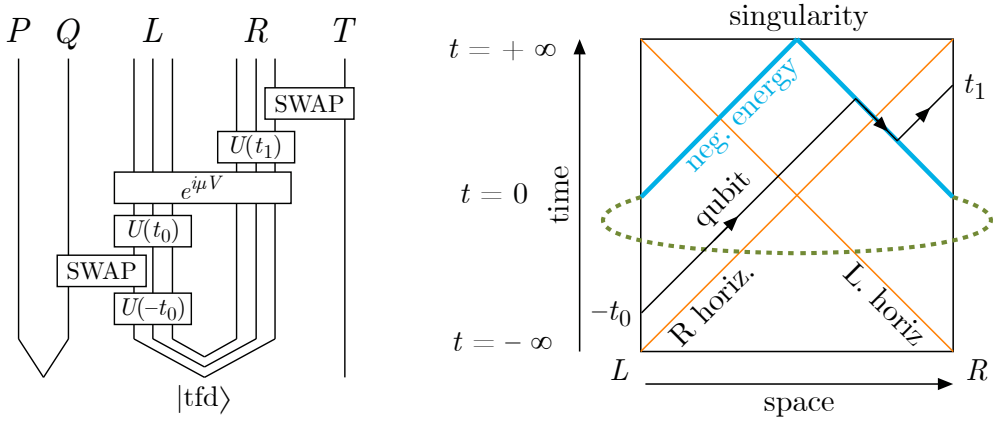



Figure 3.1: The wormhole teleportation protocol shown in the quantum information picture (left, quantum circuit) and gravity picture (right, Penrose diagram). The unitary $U(t)$ corresponds to time evolution under the left and right SYK models, i.e. $U(t) = e^{-i(H_L+H_R)t}$. Without the coupling $e^{i\mu V}$ between the left and right systems at $t = 0$ (green), no shockwave would occur and the message would continue to propagate in a straight line to enter the singularity.

As shown in Fig. 3.1, we can formally write down the wormhole teleportation protocol in the quantum information picture using quantum registers $P, Q, L, R$, and $T$. The teleportation shall send a message from $Q$ to $T$ across the thermofield double state between systems $L$ and $R$, while $P$ will be used as a reference to verify the teleportation: we shall seek to entangle $P$ and $T$ at the end of the protocol. Define the left and right Hamiltonians $H_L$ and $H_R$ with an even number $N$ of Majorana fermions $\psi$ on each side according to the Majorana SYK model with $q$ couplings, i.e.

$$H_{L,R} = i^{q/2} \sum_{1 \le j_1 < \cdots < j_q \le N} J^{L,R}_{j_1 \ldots j_q} \psi^{j_1}_{L,R} \ldots \psi^{j_q}_{L,R}, \tag{3.1}$$

where the couplings are chosen from a Gaussian distribution with mean zero and variance

$$\left\langle \left( J^{L,R}_{j_1 \ldots j_q} \right)^2 \right\rangle = \frac{J^2 (q-1)!}{N^{q-1}}. \tag{3.2}$$

To simulate the protocol on a quantum device, we perform the following, where times $t_0 \approx t_1$ are chosen to be roughly equal to the scrambling time.

1. Prepare the thermofield double state $|TFD\rangle = \frac{1}{\sqrt{Z}} \sum_n e^{-\beta E_n/2} |n\rangle_L \otimes |n\rangle_R$, where $|n\rangle_{L,R}$ are the eigenstates of the left and right systems.

2. Prepare a maximally entangled state $|\phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ between registers $P$ and $Q$.

3. At time $t = -t_0$, apply a SWAP operation between $Q$ and $L$ to insert the qubit in the wormhole.

4. At time $t = 0$, apply an interaction $e^{i\mu V}$ with $V = \frac{1}{N} \sum_i \psi^i_L \psi^i_R$ between the left and right systems. Note that to send a negative energy shockwave (Fig. 3.1), the sign of $\mu$ must be appropriately chosen.

5. At time $t = t_1$, apply a SWAP operation between $R$ and $T$ to extract the qubit from the wormhole.

At the end of the protocol, the register $T$ will be maximally entangled with the register $Q$.

### 3.3 Dirac SYK model

In the original protocol, the system of $N$ Majorana fermions in each SYK model are paired up into Dirac fermions and encoded within $N/2$ qubits. To be more conducive to implementation on quantum hardware and encourage use of a shallow Trotterization via a low-rank approximation (Sec. 3.4), we modify the Majorana SYK system described above and apply the teleportation protocol to a Dirac SYK system instead [15]. Hence, we replace Eq. 3.1 with the Hamiltonian given by

$$H = \frac{1}{(2N)^{3/2}} \sum_{i,j,k,l=1}^{N} J_{ij;kl} c^\dagger_i c^\dagger_j c_k c_l, \tag{3.3}$$

where the Dirac fermions obey

$$c_i c_j + c_j c_i = 0, \quad c_i c_j^\dagger + c_j^\dagger c_i = \delta_{ij}, \tag{3.4}$$

and complex Gaussian-distributed couplings are chosen with zero mean such that

$$J_{ij;kl} = -J_{ji;kl} = -J_{ij;lk} = J^*_{kl;ij}, \overline{|J_{ij;kl}|^2} = J^2. \tag{3.5}$$

In the large $N$ limit, the spectrum should be distributed as a Gaussian due to the selection of Gaussian couplings [21]. To achieve successful wormhole teleportation, we expect the spectrum to be approximately continuous between energy levels. This provides a threshold value on $N$, suggesting that at least $N \approx 10$ is required (Fig. 3.2).

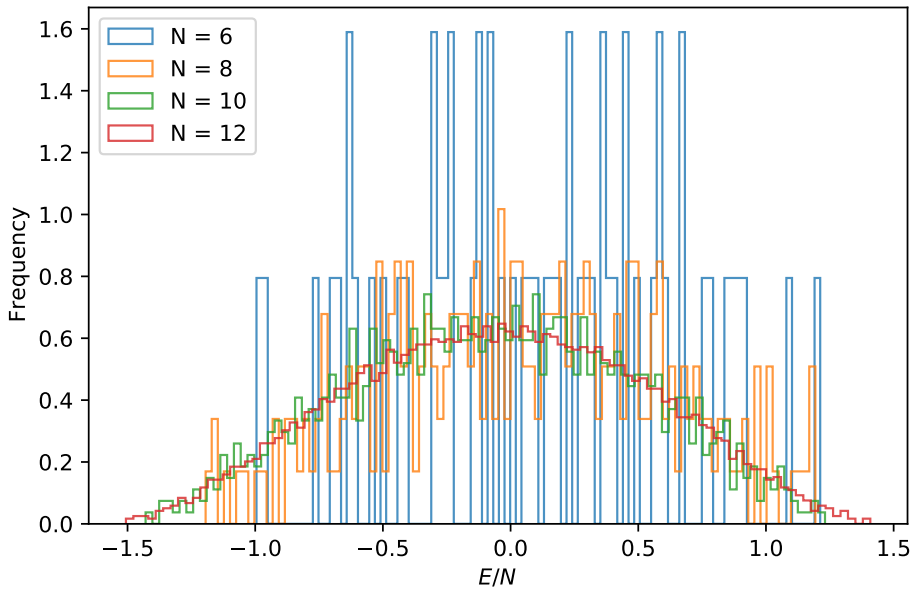



Figure 3.2: Spectrum of the Dirac SYK model for random instances of coefficients. A single SYK model must have an approximately continuous spectrum for wormhole teleportation to occur.

### 3.4 Low-rank SYK

To successfully implement wormhole teleportation on near-term quantum hardware, the key obstacle is fidelity: increasing the width (i.e. number of qubits) or depth of a quantum circuit causes errors to compound. Under a depolarization error model, the fidelity that decays like a power law with the addition

of gates and qubits. For a gate set $G$, gate errors $e_g$, qubit set $Q$, and qubit errors (measurement and state preparation) $e_q$, the fidelity of quantum hardware is well-fitted by $F = \prod_{g\in G}(1-e_g)\prod_{q\in Q}(1-e_q)$ [22]. Although preparing the thermofield double state may require a deep circuit, variational quantum eigensolvers have seen success with hardware-efficient ansatzes [17], suggesting that the thermofield double state can be prepared with a shallow circuit. Hence, the primary difficulty lies in performing the time evolution $e^{-i(H_L+H_R)t}$.

To address this issue, we propose using a low-rank SYK model. In particular, we may factorize an SYK Hamiltonian $H$ into the product of two-body terms. To satisfy the commutation relations of Eq. 3.5, we observe that $J$ is Hermitian to support commutation relations. Hence, we can diagonalize the tensor $J$ with indices $i, j, k, l$ into an $N^2 \times N^2$ supermatrix $W$ with composite indices $ij$ and $kl$ such that $W = PDP^\dagger$. Note that $W$ is not full rank, but rather has rank $L = O(N) < N/2$ with $\lim_{N\to\infty} L = N/2$. Letting $\lambda_k$ be the eigenvalues and $v_k \in \mathbb{C}^{N^2}$ be the eigenvectors, define coefficients $g_{pq,k}$ that correspond to the composite index $pq$ in the $k$th eigenvector. The SYK Hamiltonian can then be decomposed into

$$H = \sum_{k=1}^{L} \lambda_k f^\dagger(k) f(k), \quad f(k) = \sum_{p,q=1}^{N} g_{pq,k} c_p c_q. \tag{3.6}$$

Since the coefficients of the original SYK model are random, numerical experiments can instead randomly select the coefficients in this factorized Hamiltonian. Taking a single term $f^\dagger f$, we recover the original Dirac SYK model form to determine the appropriate distribution over $g_{pq,k}$. Letting $f = \sum_{i,j=1}^{N} g_{ij} c_i c_j$, we find that the simple case of $H = f^\dagger f$ gives

$$\begin{aligned} H &= \sum_{i,j,k,l=1}^{N} g^*_{ji} g_{kl} c_i^\dagger c_j^\dagger c_k c_l \\ &= \sum_{i,j,k,l=1} J_{ij;kl} c_i^\dagger c_j^\dagger c_k c_l. \end{aligned}$$

Since $J_{ij;kl}$ are normally distributed, the product of random variables $g^*_{ji} g_{kl}$ is normally distributed. Moreover, the condition $J_{ij;kl} = -J_{ji;kl} = -J_{ij;lk} = J^*_{kl;ij}$ is equivalent to

$$\begin{aligned} g^*_{ij} g_{kl} &= -g^*_{ji} g_{kl} \\ g^*_{ji} g_{lk} &= -g^*_{ji} g_{kl} \\ g^*_{lk} g_{ij} &= g^*_{ji} g_{kl}. \end{aligned}$$

The first two conditions imply that $g_{ij} = -g_{ji}$ for all $i, j$. Entering this into the final condition, we have $g^*_{lk} g_{ij} = g^*_{kl} g_{ji} = g^*_{ji} g_{kl}$. Since we require $g^*_{kl} g_{ji}$ to be real for arbitrary $i, j, k, l$, we use real coefficients.

To sample the coefficients $g_{ij}$ such that $g_{ij} g_{kl}$ is normally distributed, we sample i.i.d. from the distribution over random variables $ue^X$ where $u$ is a Rademacher random variable and $X$ is given by [23]

$$X = \frac{\log 2}{4} - G_{1/2,0} - \left[ \sum_{i=1}^{\infty} \frac{G_{1/2,i}}{2i+1} - \frac{1}{4} \log \left( 1 + \frac{1}{i} \right) \right] \tag{3.7}$$

where $G_{1/2,i} = \text{Gamma}(1/2, 1)$ are random variables labeled by $i$. This is numerically verified to produce a Gaussian distribution of coefficients when returned to the original SYK model form (Eq. 3.3).

The Dirac SYK model is now approximated in the form of an $L$-rank Hamiltonian given by Eq. 3.6, which may be more amenable to decompose into Givens rotations circuits $U_k$ of linear depth and linear connectivity [24]

$$H = \sum_{k=1}^{L} U_k \left( \sum_{p=1}^{N} h_{p,k} c^\dagger_p c_p \right)^\dagger \left( \sum_{p=1}^{N} h_{p,k} c^\dagger_p c_p \right) U^\dagger_k. \tag{3.8}$$

Since the Hamiltonian is now diagonalized, a single Trotter step has commuting terms that can help reduce circuit depth, possibly using a SWAP network [25].

Although using a low-rank Hamiltonian may improve circuit depth and require fewer gates, it may require a *wider* circuit due to a worsened spectrum. While Fig. 3.2 suggested that $N \approx 10$ Dirac fermions would be enough to generate a sufficiently dense spectrum for wormhole teleportation, the low-rank spectrum is less Gaussian and may require $N \approx 12$ (Fig. 3.3).

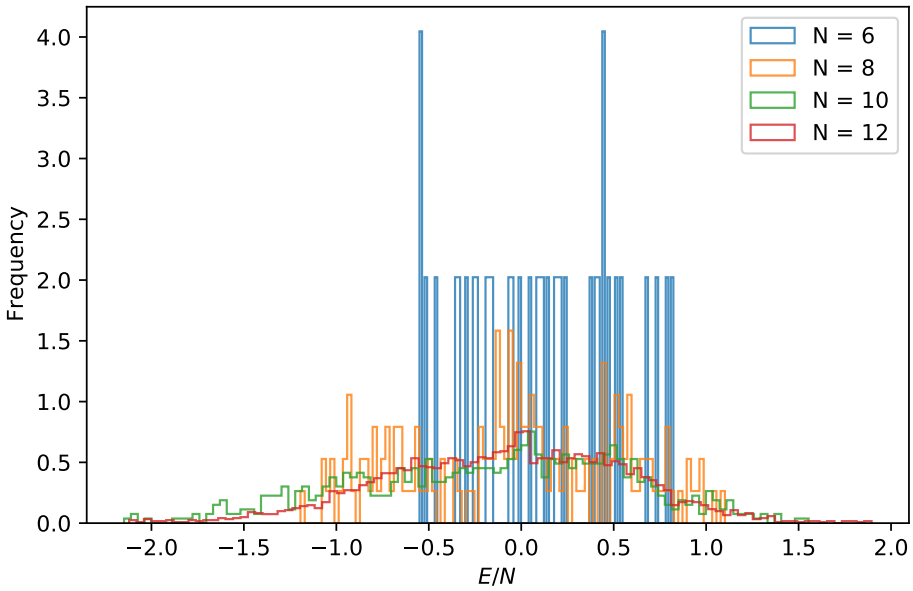



Figure 3.3: Spectrum of the Dirac SYK model with rank 2 for random instances of coefficients.

## 3.5 Shallow circuit time evolution

### Compressed Trotterization algorithm

To further reduce the circuit depth of the time evolution operation $e^{-iHt}$, we explore the possibility of using machine learning methods. In particular, learning a variational quantum circuit that time evolves *one particular state* to high fidelity may enable the use of shallower time evolution circuits. Although preparing a quantum circuit that performs wormhole teleportation on generic input messages may be more helpful to probe quantum gravity theories, a specialized shallow circuit may be more effective for near-term demonstrations of wormhole teleportation. In this section, we propose a method that applies generically to Hamiltonians beyond the SYK model, and we provide examples for a Fermi-Hubbard model and jellium model.

Using a parameterized ansatz $V(\theta)$, we may iteratively learn the full time evolution operator

$$U(T) \approx \prod_{j=1}^{N} \exp\left(-iH(jT/N)\frac{T}{N}\right) \tag{3.9}$$

by first learning shallow approximations to $U(T/N), U(2T/N), \ldots$ for some large $N$. This ensures that the optimization path is towards the true global

minimum, while only providing small corrections to the existing approximation. Hence, we find parameters $\theta_k$ where

$$\begin{aligned}
\theta_1 &= \underset{\theta}{\text{argmin}}\ ||e^{-iH(T/n)T/N} - V(\theta)|| \\
\theta_2 &= \underset{\theta}{\text{argmin}}\ ||e^{-iH(2T/N)T/N} V(\theta_1) - V(\theta)|| \\
&\vdots \\
\theta_N &= \underset{\theta}{\text{argmin}}\ ||e^{-iH(T)(T/N)} V(\theta_{N-1}) - V(\theta)||
\end{aligned}$$

and the notation $||A - B||$ denotes a suitable norm (such as trace norm or operator norm) of $A - B$. The final circuit $V(\theta_N)$ would then correspond to an approximation of $U(T)$.

Note that we can minimize the trace norm by measuring over a complete basis for an $n$-qubit system,

$$\theta_{k+1} = \underset{\theta}{\text{argmax}} \sum_{j=1}^{2^n} \langle j| \left(e^{-iH(kT/N)(T/N)} V(\theta_k)\right)^\dagger V(\theta_{k+1}) |j\rangle \tag{3.10}$$

where $\{|j\rangle\}$ are orthogonal. However, approximating a full unitary would likely require exponential time and result in a deeper circuit; hence, we proceed with preparing time evolution for a single state. We only minimize the inner product with the true time-evolved state, which can be evaluated with a swap test

$$\theta_{k+1} = \underset{\theta}{\text{argmax}}\ \langle x(0)| \left(e^{-iH(kT/N)(T/N)} V(\theta_k)\right)^\dagger V(\theta_{k+1}) |x(0)\rangle \tag{3.11}$$

as illustrated in Fig. 3.4.

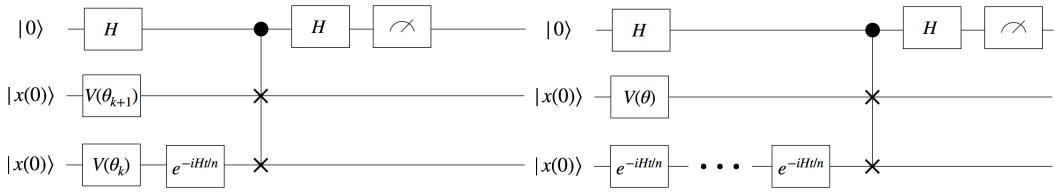



Figure 3.4: Schematic for learning a shallow approximation to a Trotterization, where the blue shaded region is optimized to maximize the value of the swap test. Left: proposed method, where approximations $V(\theta_k)$ are learned by iteratively adding the $k$th Trotter step $e^{-iHt/n}$. Right: naive method, where the approximation is learned with by swapping against the entire time evolution circuit, requiring a much deeper circuit (and hence more noise).

Hence, the optimization method ensures that only shallow circuits are run, accessing an approximation of a circuit with many Trotter steps while only

evaluating few Trotter steps in the physical realization. As an ansatz for the variational circuit $V(\theta)$, we use a single Trotter step but with each gate fully parameterized. By allowing increased expressibility within the Trotter step and removing constraints of equal rotations across different gates, we hope to extend the set of states reachable by the variational ansatz.

While optimization of many variational quantum circuits are subject to *barren plateaus* [19], the structure of the proposed time evolution learner ensures a smooth training landscape. Barren plateaus are characterized by poor directionality encoded in the expected gradient of random circuits (formalized by Levy's lemma); however, non-random selections such as those close to the identity are shown to produce a stronger gradient. This behavior is empirically seen in other variational optimization tasks, where the initialization of a circuit ansatz with identity blocks provides a strong gradient in the beginning of optimization that vanishes as the procedure moves away from the identity [26].

In the proposed approach, we append a single Trotter step over a small slice of time (i.e. $T/N$ is small for large $N$) to the previous fit of $V(\theta_k) \approx e^{-iH(kt/N)(T/N)}$. This encourages the successful initialization of parameters $\theta_{k+1}$ to ensure that $\left(e^{-iH(kT/N)(T/N)}V(\theta_k)\right)^\dagger V(\theta_{k+1})$ is close to the identity at the beginning of the next iteration's optimization. That is, sufficiently large $N$ suggests that each successive $\theta_{k+1}$ will only be a small perturbation away from $\theta_k$, and thus the entire optimization problem (step by step) will always remain near the identity.

**Numerical experiments**

As an example to illustrate the applicability of this technique beyond the SYK model, we consider two systems: a Fermi-Hubbard approximation of 20 Trotter steps and a jellium system approximation of 16 Trotter steps, both compressed into a circuit of depth equal to a single Trotter step.

We prepare a $2 \times 2$ (8-qubit) Fermi-Hubbard Hamiltonian with arbitrarily chosen parameters (tunneling amplitude, Coulomb potential, chemical potential, magnetic field) given by $(1, 4, 1, 1)$. Optimization of the time evolution of an initial state $|x(0)\rangle = X_1X_2X_5X_6 |0\rangle$ is performed following the above approach (Fig. 3.5). While a single Trotter step yields a fidelity of 2% at $t = 1$, the approximation maintains a fidelity of 85% with the same-depth circuit due to approximating 20 Trotter steps.

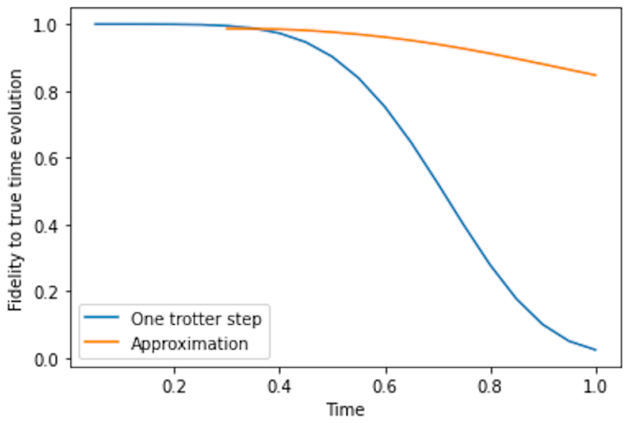



Figure 3.5: Fidelity of an eight-qubit Fermi-Hubbard model simulation over time. A single Trotter step is compared to learned time evolution, which has the same circuit as a single Trotter step but learns gate parameters to compress 20 Trotter steps.

Similarly, a smaller jellium system with two electrons on a $2 \times 2$ grid is benchmarked against true Trotterization. We include a noisy simulation with deploarizing noise to emphasize the additional advantages of having a shallow time evolution approximation (Fig. 3.6). While compressing 16 Trotter steps into a shallow circuit can already yield improvements over one and two Trotter steps, the advantage is further increased when noise is considered due to the lower fidelity of deeper Trotterizations. Although not shown in the figure, the fidelity of the approximate jellium time evolution remains above 90% through $t \approx 50$.

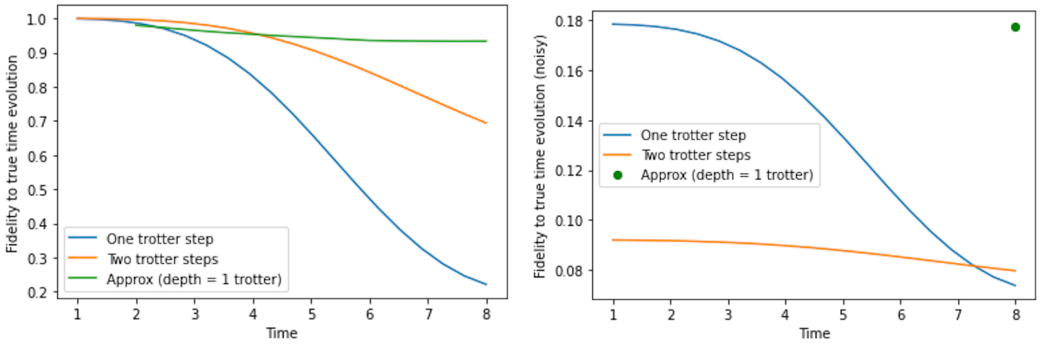



Figure 3.6: Fidelity of a four-qubit jellium Hamiltonian over time without noise (left) and with 0.5% depolarizing noise (right). Circuits with one and two Trotter steps are compared to learned time evolution, which compresses 16 Trotter steps into a circuit with the depth of one Trotter step.

### 3.6 Classical simulation of wormhole behavior

To view teleportation through the wormhole, we check two quantities: the mutual information between $P$ and $T$, and the left-right causal propagator $\mathcal{K}$. For both of these quantities, we expect an asymmetry in the sign of $\mu$ for the interaction $e^{i\mu V}$, since only one sign of $\mu$ will cause a negative energy shockwave.

The mutual information $I_{PT}$ is defined by

$$I_{PT} = S(P) + S(T) - S(PT), \tag{3.12}$$

where $S$ indicates von Neumann entropy. In perfect teleportation, $I_{PT} = 2\log 2$. Teleportation can occur *without* going through the wormhole: in the fully scrambled regime, the unitary $e^{iHt}$ is equivalent to a random unitary, causing signals to appear on the right side of system in reverse time order [11]. However, such scrambling is symmetric in $\mu$, allowing the wormhole teleportation to be identified by checking for the asymmetry in $\mu$.

In classical circuit simulations, memory (i.e. circuit width) provides the largest hardware constraint. Performing a Jordan-Wigner transformation to encode the Dirac fermions, we have creation operators of the form $\frac{1}{2}(X - iY)$ and annihilation operators of the form $\frac{1}{2}(X + iY)$. For the $n$th fermion, we prepend $Z^{\otimes(n-1)}$ to the above creation/annihilation operators. Hence, a system of $2N$ fermions ($N$ on the left and $N$ on the right) requires $2^{2N}$ qubits; including the registers $P, Q$, and $T$, we have a total of $2^{2N+3}$ qubits for the teleportation protocol. Storing a time evolution matrix of the form $e^{-iHt}$ where $t \approx t_{\text{scramble}}$ over the *entire* system, the matrix will be approximately a random unitary and thus dense. Hence, the memory required (taking a typical 128-bit encoding for complex numbers) reaches 64 GB at $n \approx 6$. Counting overhead on the diagonalization to perform the exponentiation (which uses a Cholesky decompomsition with running time $O(2^{6N})$), numerical simulations with reasonable performance reach around $N \approx 5$, which falls short of the estimated SYK models required to have an approximately continuous spectrum (Fig. 3.2). Nevertheless, even for a teleportation protocol evaluated at $N = 5$, some amount of asymmetry in $\mu$ is observed at $|\mu| \approx 5$ (Fig. 3.7).

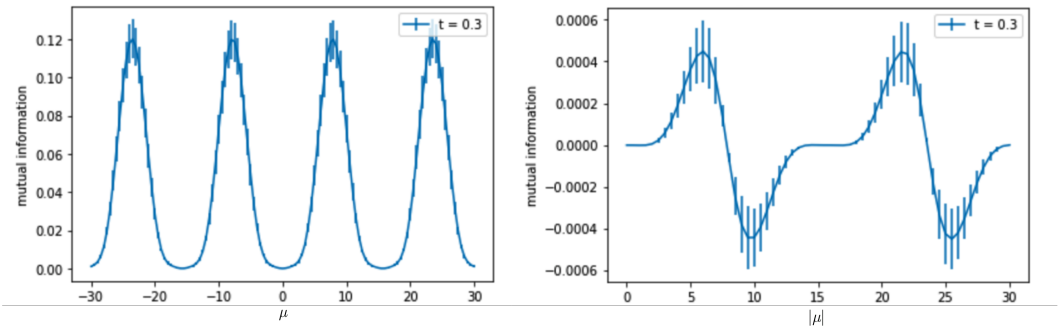



Figure 3.7: Mutual information of the teleportation protocol (left) and the asymmetry of mutual information between positive and negative $\mu$ (right). The time $t_0 = t_1 = 0.3$ is chosen to be approximately equal to the scrambling time; inverse temperature is $\beta = 10$. Error bars show one standard deviation across random instances of the Dirac SYK model.

The periodicity in $\mu$ is expected, as seen by the left-right causal propagator $\mathcal{K}$ from the Majorana SYK teleportation [11]. From considering the density matrix of the system under the teleportation protocol, the OTOC with Majorana fermions $\psi_{L,R}$ defined by

$$\mathcal{K} = \langle \{\psi_L(-t_0),\, e^{-i\mu V}\psi_R(t_1)e^{i\mu V}\}\rangle \tag{3.13}$$

$$= \mathrm{Im}\left(-\frac{e^{-i\mu/(Nq)}\omega^{2/q}}{J^{2/q}\left[\cosh\omega(t_0-t_1) - \frac{J(e^{-i\mu/N}-1)}{\omega}\sinh\omega(t_1 - i\beta/2)\sinh\omega t_0\right]^{2/q}}\right) \tag{3.14}$$

appears in the mutual information as

$$I_{PT} = \frac{1}{4}[(\mathcal{K}-1)^2\log(\mathcal{K}-1)^2 + (\mathcal{K}+1)^2\log(\mathcal{K}+1)^2 + 2(1-\mathcal{K}^2)\log\big(1-\mathcal{K}^2\big)], \tag{3.15}$$

which is close to maximal when $\mathcal{K}$ approaches $\pm 1$. Here, the interaction is given by $V = \frac{1}{N}\sum_j \psi_L^j\psi_R^j$ and $\omega$ is an integral constant. Since $\mathcal{K}$ has dependence like $e^{i\mu}$, we expect periodicity in $\mu$ for the mutual information. In wormhole teleportation, a single period of Fig. 3.7 on the left should have asymmetry in $\mu$; this asymmetry will flip sign as the next half-periods are added on the right and left, consistent with Fig. 3.7 on the right.

Note that although we numerically compute the Dirac SYK model, the behavior is largely expected to be the same, since Majorana fermions can be coupled into Dirac fermions with $c = \psi^1 + i\psi^2$ and thus the Dirac SYK corresponds

to a special selection of coefficients in the Majorana SYK model. In Eq. 3.13, wormhole teleportation appears as a peak in $\mathcal{K}$ as a function of $t = t_0 = t_1$, corresponding to the maximization of mutual information. Moreover, for fixed $\mu$, examining the denominator of Eq. 3.13 suggests that the peak should move to the right with larger $N$ (Fig. 3.8).

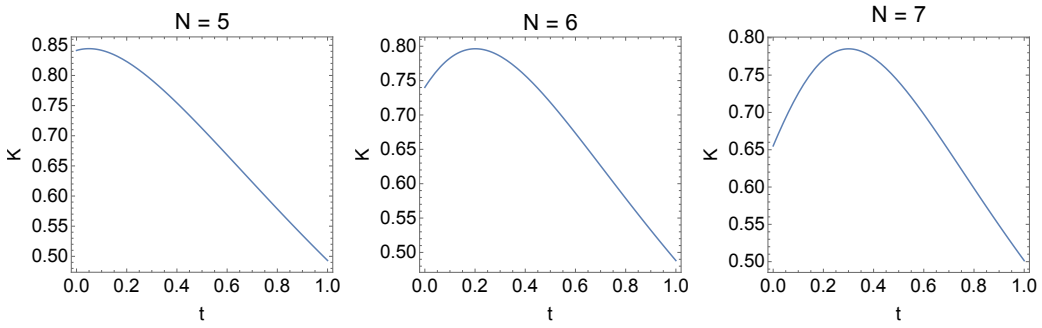



Figure 3.8: Causal propagator $\mathcal{K}(t_0, t_1)$ for $t_0 = t_1 \approx t_{\text{scramble}}$ on the Majorana SYK. At larger times (not shown), the propagator asymptotically reaches $\mathcal{K} \approx 0$.

To examine the causal propagator in the Dirac SYK model for larger $N$ with a continuous spectrum, we must modify our approach to coupling the left and right systems. As shown in Fig. 3.1, the left and right systems undergo independent time evolution; the only unitary across the entire system is $e^{i\mu V}$. In the case of the Dirac SYK, we take interaction $V = \frac{i}{N}\sum_j (c_L^j)^\dagger c_R^j + c_L^j (c_R^j)^\dagger$. Applying the Jordan-Wigner transformation described above, the unitary $e^{i\mu V}$ is an exponential of a sum of single-qubit Pauli operations. Expanding into a series for small $\mu$, this is equivalent to a Clifford circuit, which can be efficiently simulated classically in polynomial time and space by the Gottesman-Knill theorem [27]. Hence, only arbitrary unitaries $e^{-iH_L t}, e^{-iH_R t}$ over half of the qubits have exponential cost.

To take advantage of this, we prepare consecutive qubits corresponding to the left SYK model, and a similar register for the right SYK model. For the Dirac SYK, evaluating commutations relates the SYK model coefficients by

$$(J_{ij;kl} c_i^\dagger c_j^\dagger c_k c_l)_L \to (J_{ij;kl} c_l^\dagger c_k^\dagger c_j c_i)_R. \tag{3.16}$$

Each of the $e^{-iH_L t}, e^{-iH_R t}$ time evolution operators can be prepared individually, then a sparse matrix with a tensor product to the identity gives the full operator on the system.

The last component that must avoid $2^{2N}$ memory is the preparation of the thermofield double state. In the Majorana SYK case, the thermofield double state is given by

$$|TFD\rangle = \frac{1}{\sqrt{Z}} e^{-\beta(H_L+H_R)/4} |I\rangle \,, \tag{3.17}$$

where $|I\rangle$ is the maximally entangled state of the left and right systems, i.e. is the state annihilated by complex fermions

$$(\psi_L^j + i\psi_R^j) |I\rangle = 0 \tag{3.18}$$

for all $j$. Applying this condition, we find that $|I\rangle$ is the ground state of the interaction term $V$. This fact allows us to efficiently prepare the thermofield double state under the Dirac SYK interaction $V$.

Finally, we can evaluate the causal propagator $\mathcal{K}$ in the Dirac SYK teleportation protocol, where

$$\mathcal{K}(t_0, t_1) = \langle \mathrm{Re}\{a_L(-t_0), e^{-i\mu V} a_R^\dagger(t_1) e^{i\mu V}\}\rangle. \tag{3.19}$$

Expanding the anticommutator, the expectation over the thermofield double state is directly computed at our expected teleportation location of $|\mu| = 5$ (Fig. 3.9). Using the Clifford circuit expansion of $e^{i\mu V}$ and applying separate time evolution operators to the left and right systems, the memory-efficient simulation allows us to reach $N = 7$. Higher $N$ can be stored in memory, but the cost of matrix diagonalization becomes prohibitive.

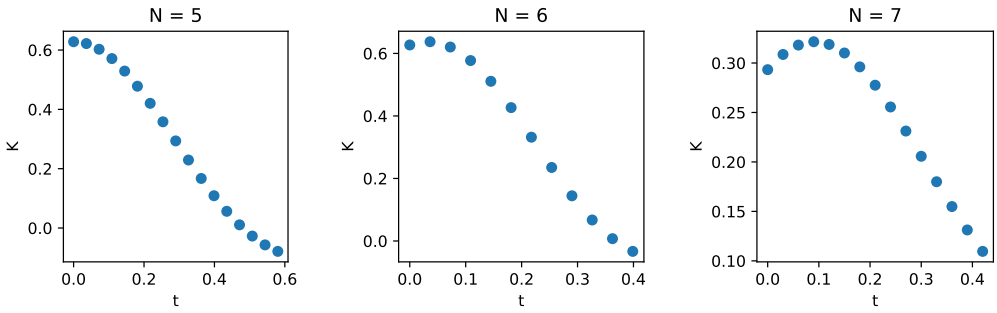



Figure 3.9: Causal propagator $\mathcal{K}(t_0, t_1)$ for $t_0 = t_1 \approx t_{\text{scramble}}$. At larger times (not shown), the propagator oscillates at $\mathcal{K} \approx 0$.

With increasing $N$, a peak corresponding to wormhole teleportation appears, and the peak moves to the right with larger $N$. Although we did not derive the precise form of $\mathcal{K}(t_0, t_1)$ for the Dirac SYK, we obtain the expected qualitative behavior from the Majorana causal propagator analysis (Fig. 3.8).

### 3.7 Near-term quantum simulation

From classical simulation, it appears that a Dirac SYK model with $N = O(10)$ may be sufficient to observe wormhole teleportation. With two SYK models and additional registers to swap in and out a qubit from the wormhole, this corresponds to a quantum devices with $O(25)$ qubits. We can now summarize the wormhole teleportation protocol in terms of the near-term approaches described above.

1. Consider the thermofield double state $|TFD\rangle = \frac{1}{\sqrt{Z}} \sum_n e^{-\beta E_n/2} |n\rangle_L \otimes |n\rangle_R$, where $|n\rangle_{L,R}$ are the eigenstates of two low-rank Dirac SYK Hamiltonians. To prepare the state, optimize a hardware-efficient variational quantum eigensolver [17] to the Hamiltonian $H = H_L + H_R + i\nu V$. The ground state is approximately the thermofield double state with inverse temperature $O(1/\nu)$ [18].

2. Prepare a maximally entangled state $|\phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ between registers $P$ and $Q$.

3. Time evolve to $t = -t_0$. Each Trotter step is a shallow circuit due to the low-rank SYK Hamiltonian. Compress successive Trotter steps into a shallow variational circuit by iteratively learning circuit parameters for each appended Trotter step.

4. Apply a SWAP operation between $Q$ and $L$ to insert the qubit in the wormhole.

5. Time evolve to $t = 0$; once again, this requires the compression of shallow Trotter steps into a shallow variational circuit.

6. At time $t = 0$, apply interaction $e^{i\mu V}$ between the left and right systems. Since $\mu V$ has size $O(1)$, the interaction can be decomposed into a series of single-qubit operators with constant scaling with $N$. Note that this is also an advantageous feature for near-term error-corrected quantum computing, since the resulting Clifford circuits have no $T$ gates, which are not transversal in many quantum error-correcting codes and are thus costly to correct [28].

7. Time evolve to $t = t_1$; once again, this requires the compression of shallow Trotter steps into a shallow variational circuit.

8. Apply a SWAP operation between $R$ and $T$ to extract the qubit from the wormhole.

While investigating quantum gravity through protocols such as wormhole teleportation may shed light on the ER=EPR conjecture, the proposed approaches may find wider applicability in many-body systems more generally. Notably, many-body teleportation without gravitational dynamics has been observed in chaotic spin chains and high-temperature SYK models [], providing fertile ground for further exploration [3, 4, 11]. Moreover, methods such as compressed Trotterization were shown above to be applicable to Hamiltonian simulation in general, promising relevance to a wider class of problems. For instance, Fermi-Hubbard dynamics with 16 qubits have achieved reasonable fidelity on experimental quantum hardware up to a depth of 55 Trotter steps [29]. If each Trotter step encoded $O(10)$ Trotter steps with a variational approximation (as shown in in Sec. 3.5 with an 8-qubit Fermi-Hubbard model), the simulation time could be significantly extended. Hence, while the investigation of quantum gravity on a quantum computer may ultimately provide insight into the effects of quantum and stringy corrections to a semi-classical gravity picture, our work is shown to be relevant for the broader advancement of quantum simulation.

## References

[1]S. Lloyd, "Universal quantum simulators", Science, 1073–1078 (1996).

[2]I. M. Georgescu, S. Ashhab, and F. Nori, "Quantum simulation", Reviews of Modern Physics **86**, 153 (2014).

[3]A. R. Brown, H. Gharibyan, S. Leichenauer, H. W. Lin, S. Nezami, G. Salton, L. Susskind, B. Swingle, and M. Walter, *Quantum gravity in the lab: teleportation by size and traversable wormholes*, 2021.

[4]S. Nezami, H. W. Lin, A. R. Brown, H. Gharibyan, S. Leichenauer, G. Salton, L. Susskind, B. Swingle, and M. Walter, *Quantum gravity in the lab: teleportation by size and traversable wormholes, part ii*, 2021.

[5]J. Maldacena, "The large-n limit of superconformal field theories and supergravity", International journal of theoretical physics **38**, 1113–1133 (1999).

[6]J. Maldacena, D. Stanford, and Z. Yang, "Diving into traversable wormholes", Fortschritte der Physik **65**, 1700034 (2017).

[7]W. Cottrell, B. Freivogel, D. M. Hofman, and S. F. Lokhande, "How to build the thermofield double state", Journal of High Energy Physics **2019**, 58 (2019).

[8]P. Gao, D. L. Jafferis, and A. C. Wall, "Traversable wormholes via a double trace deformation", Journal of High Energy Physics **2017**, 151 (2017).

[9]J. Maldacena and L. Susskind, "Cool horizons for entangled black holes", Fortschritte der Physik **61**, 781–811 (2013).

[10]L. Susskind, *Dear qubitzers, gr=qm*, 2017.

[11]P. Gao and D. L. Jafferis, *A traversable wormhole teleportation protocol in the syk model*, 2021.

[12]S. Sachdev and J. Ye, "Gapless spin-fluid ground state in a random quantum heisenberg magnet", Phys. Rev. Lett. **70**, 3339–3342 (1993).

[13]A. Kitaev, "A simple model of quantum holography", in Kitp strings seminar and entanglement, Vol. 12 (2015), p. 26.

[14]K. A. Landsman, C. Figgatt, T. Schuster, N. M. Linke, B. Yoshida, N. Y. Yao, and C. Monroe, "Verified quantum information scrambling", Nature **567**, 61–65 (2019).

[15]S. Sachdev, "Bekenstein-hawking entropy and strange metals", Physical Review X **5**, 041025 (2015).

[16]J. Kim, X. Cao, and E. Altman, "Low-rank sachdev-ye-kitaev models", Physical Review B **101**, 125112 (2020).

[17]A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets.", Nature **549**, 242 (2017).

[18]J. Maldacena and X.-L. Qi, *Eternal traversable wormhole*, 2018.

[19]J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, "Barren plateaus in quantum neural network training landscapes", Nature communications **9**, 1–6 (2018).

[20]J. Preskill, "Quantum computing in the nisq era and beyond", Quantum **2**, 79 (2018).

[21]A. M. Garcia-Garcia and J. J. Verbaarschot, "Spectral and thermodynamic properties of the sachdev-ye-kitaev model", Physical Review D **94**, 126010 (2016).

[22]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, et al., "Supplementary information for" quantum supremacy using a programmable superconducting processor"", arXiv:1910.11333 (2019).

[23]I. Pinelis, *The exp-normal distribution is infinitely divisible*, 2018.

[24]I. D. Kivlichan, J. McClean, N. Wiebe, C. Gidney, A. Aspuru-Guzik, G. K.-L. Chan, and R. Babbush, "Quantum simulation of electronic structure with linear depth and connectivity", Phys. Rev. Lett. **120**, 110501 (2018).

[25]M. P. Harrigan, K. J. Sung, M. Neeley, K. J. Satzinger, F. Arute, K. Arya, J. Atalaya, J. C. Bardin, R. Barends, S. Boixo, and et al., "Quantum approximate optimization of non-planar graph problems on a planar superconducting processor", Nature Physics **17**, 332–336 (2021).

[26]E. Grant, L. Wossnig, M. Ostaszewski, and M. Benedetti, "An initialization strategy for addressing barren plateaus in parametrized quantum circuits", Quantum **3**, 214 (2019).

[27]D. Gottesman, "The Heisenberg representation of quantum computers", in 22nd International Colloquium on Group Theoretical Methods in Physics (July 1998).

[28]X. Zhou, D. W. Leung, and I. L. Chuang, "Methodology for quantum logic gate construction", Physical Review A **62**, 052316 (2000).

[29]F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, A. Bengtsson, S. Boixo, M. Broughton, B. B. Buckley, et al., "Observation of separated dynamics of charge and spin in the fermi-hubbard model", arXiv preprint arXiv:2010.07965 (2020).

*A p p e n d i x A*

# THEORETICAL FRAMEWORK OF THE QUANTUM NEURAL TANGENT KERNEL

## A.1 Properties of the neural tangent kernel

Before providing proofs of the central results presented in the main text, we provide preliminary definitions and results related to the neural tangent kernel (NTK) that are used throughout the SM.

### Framework

To provide the necessary notation for the NTK, we repeat the definitions and data assumptions of the main text. Consider a binary classification dataset $S$ of $n$ training examples $\{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}\}_{i=1}^n$, where there are $O(n)$ examples in each class. To parameterize our results, we must define the separability between data examples.

**Definition A.1.1** (Separability)**.** The separability of data points $\mathbf{x}_i, \mathbf{x}_j$ is given by $\delta_{ij} := 1 - |\mathbf{x}_i \cdot \mathbf{x}_j|$.

We make the following standard assumption about separability across the entire dataset [1–3] with an additional lower bound on the separability that is commonly satisfied (see Sec. A.5 of the SM).

**Assumption A.1.2.** Assume that $|\mathbf{x}_i \cdot \mathbf{x}_i| = 1$ for all $i$. For some $0 < \delta \leq 1$, let $|\mathbf{x}_i \cdot \mathbf{x}_j| \leq 1 - \delta$ for all $i, j \in [n]$ with $i \neq j$. Moreover, assume $\delta = \Omega(1/\text{poly}\, n)$ for a dataset of size $n$.

We will be considering a classification problem on the dataset with $y_i = \pm 1$ associated with each $\mathbf{x}_i$. To ensure the dataset is well-behaved (i.e. does not change class at an infinitesimal scale), we require an additional assumption.

**Assumption A.1.3.** Define the $\epsilon$-neighborhood around a given data point $\mathbf{x}_*$ sampled i.i.d. from the data distribution to be $N_\epsilon = \{i : \mathbf{x}_* \cdot \mathbf{x}_i \geq 1 - \epsilon\}$. There exists a constant $\epsilon$ such that with high probability $y_i = y_*$ for all $\{y_i : i \in N_\epsilon\}$. Moreover, the distribution of $\mathbf{x}_i$ within $N_\epsilon$ is approximately uniform.

Finally, for the neural network with activation function $\sigma$, we require a normalization constraint equivalent to applying batchnorm after every layer of the neural network.

**Assumption A.1.4.** The activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is normalized such that

$$\mathop{\mathbb{E}}_{X\sim\mathcal{N}(0,1)}[\sigma(X)] = 0 \text{ and } \mathop{\mathbb{V}}_{X\sim\mathcal{N}(0,1)}[\sigma(X)] = \mathop{\mathbb{E}}_{X\sim\mathcal{N}(0,1)}[\sigma^2(X)] = 1. \tag{A.1}$$

Following Agarwal et al. [4], we define the nonlinearity of the activation function and note the effect of normalization on the resulting constant.

**Definition A.1.5** (Coefficient of nonlinearity)**.** The coefficient of nonlinearity of the activation function $\sigma$ is defined to be $\mu := 1 - \left(\mathbb{E}_{X\sim\mathcal{N}(0,1)}[X\sigma(X)]\right)^2$.

**Elements of the NTK**

To write the elements of the NTK, we define the dual activation function $\hat{\sigma}$ corresponding to the activation function $\sigma$ [5].

**Definition A.1.6.** Consider data $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^d$ such that $||\mathbf{x}_i|| = ||\mathbf{x}_j|| = 1$ and hence $\rho = \mathbf{x}_i \cdot \mathbf{x}_i \in [-1, 1]$. Define the conjugate activation function $\hat{\sigma} : [-1, 1] \to [-1, 1]$ as follows:

$$\hat{\sigma}(\mathbf{x}_i \cdot \mathbf{x}_j) := \mathbb{E}_{\mathbf{w}\sim\mathcal{N}(0,\, I_d)}[\sigma(\mathbf{w} \cdot \mathbf{x}_i)\sigma(\mathbf{w} \cdot \mathbf{x}_j)]. \tag{A.2}$$

From Arora et al. [6], the elements of the NTK are given by

$$(K_{\text{NTK}})_{ij} = \sum_{h=1}^{L+1} \hat{\sigma}^{(h-1)}(\rho_{ij}) \left( \prod_{h'=h}^{L} \dot{\hat{\sigma}}(\hat{\sigma}^{(h')}(\rho_{ij})) \right) \tag{A.3}$$

for $\rho_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j$. Throughout the text, we will use $(K_{\text{NTK}})_{ij}$ to denote the $ij$th matrix element $K_{\text{NTK}}(\mathbf{x}_i, \mathbf{x}_j)$. Because Eq. A.3 only requires the inner product $\mathbf{x}_i \cdot \mathbf{x}_j$, we will also define for convenience the function $\hat{K}_{\text{NTK}}(\mathbf{x}_i \cdot \mathbf{x}_j) := K_{\text{NTK}}(\mathbf{x}_i, \mathbf{x}_j)$.

For later use, we define a function $B(L, \delta, \mu)$: for any $\mu \in (0, 1]$, $\delta \in (0, 1)$ and a positive integer $L$, we let

$$B(L, \delta, \mu) := \frac{1}{2}\left(1 - \frac{\mu}{2}\right)^{L - L_0(\delta,\mu)}, \text{ where} \tag{A.4}$$

$$L_0(\delta, \mu) := \max\left\{ \left\lceil \frac{\log\left(\frac{1}{2\delta}\right)}{\log\left(1 + \frac{\mu}{2}\right)} \right\rceil, 0 \right\} = O\left(\frac{\log(1/\delta)}{\mu}\right). \tag{A.5}$$

The theorems in the main text are given in terms of $L_{\text{conv}}$, which arises naturally as the minimum depth for the neural network to converge by gradient descent (see Sec. A.1 below). This minimum depth is given by

$$L_{\text{conv}} := \frac{8\log(n/\delta)}{\mu}, \tag{A.6}$$

which is related to $L_0(\delta, \mu)$ as follows.

**Lemma A.1.1.** *For all $\mu, \delta \in (0, 1]$ and $n \geq 2$, we have that $L_{\text{conv}} \geq 2L_0(\delta, \mu)$.*

*Proof.* Since $L_{\text{conv}} > 0$, this is trivially satisfied if $\delta \geq 1/2$. For $\delta < 1/2$, we note that the derivative of $L_{\text{conv}}/L_0$ with respect to $\delta$ is given by

$$\frac{\partial}{\partial\delta}\frac{L_{\text{conv}}}{L_0} = \frac{4\log(2n)\log(1+\mu/2)}{\delta\mu\log^2(2\delta)} > 0, \tag{A.7}$$

and thus evaluating the limiting case of $\delta \to 0$ is sufficient to bound $L_{\text{conv}}/L_0$. Since $\lim_{\delta\to 0} L_{\text{conv}}/L_0 = \frac{4\log(1+\mu/2)}{\mu} > 1$, we conclude that $L_{\text{conv}} \geq 2L_0(\delta, \mu)$ for all allowed parameter values. □

Finally, we note some important properties of the conjugate activation function (Def. A.1.6) from Daniely et al. [5] and Agarwal et al. [4].

**Remark A.1.2.** *The following properties hold for an activation function normalized under Assumption A.1.4, where $h_0, h_1, \ldots$ denote the Hermite polynomials.*

1. *Let $a_i = \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\sigma(z)h_i(z)]$. Due to normalization of $\sigma$, $a_0 = 0$ and $\sum_{i=1}^{\infty} a_i^2 = 1$.*
2. *We have Hermite expansions $\sigma(u) = \sum_{i=1}^{\infty} a_i h_i(u)$ and $\hat{\sigma}(\rho) = \sum_{i=1}^{\infty} a_i^2 \rho^i$.*
3. *Due to normalization of $\sigma$, we have $0 < \mu \leq 1$ and in particular $\mu = 1 - a_1^2$.*
4. *If $\dot{\sigma}$ denotes the derivative of $\sigma$, then $\hat{\dot{\sigma}} = \dot{\hat{\sigma}}$.*

**NTK matrix element bounds**

We will require bounds on the matrix elements of the NTK. The proof of an upper bound may be found in Theorem 27 of Agarwal et al. [4], the result of which we state here.

**Theorem A.1.3** (NTK element upper bound)**.** *Consider an NTK corresponding to a neural network of depth $L$. The diagonal entries of $K_{\mathrm{NTK}}$ are all equal and given by $(K_{\mathrm{NTK}})_{ii} = \frac{\hat{\sigma}(1)^{L+1}-1}{\hat{\sigma}(1)-1}$. Furthermore, if $L \geq 2L_0(\delta, \mu)$, then*

$$\left|\frac{(K_{\mathrm{NTK}})_{ij}}{(K_{\mathrm{NTK}})_{11}}\right| \leq 2B(L/2, \delta_{ij}, \mu), \tag{A.8}$$

*where $|\mathbf{x}_i \cdot \mathbf{x}_j| = 1 - \delta_{ij}$ for $i \neq j$.*

As a result of Lemma A.1.1, we observe that Theorem A.1.3 on the matrix elements of the NTK is valid for all $L \geq L_{\mathrm{conv}}$. Moreover, we can simplify the bound further.

**Lemma A.1.4.** *If $L \geq L_{\mathrm{conv}}$, then we have the following bounds on $(K_{\mathrm{NTK}})_{ij}$ for $i \neq j$. If $0 < \delta_{ij} < 1/2$*

$$\left|\frac{(K_{\mathrm{NTK}})_{ij}}{(K_{\mathrm{NTK}})_{11}}\right| \leq \left(\frac{\delta}{\delta_{ij} n}\right)^2, \tag{A.9}$$

*while for $1/2 \leq \delta_{ij} \leq 1$,*

$$\left|\frac{(K_{\mathrm{NTK}})_{ij}}{(K_{\mathrm{NTK}})_{11}}\right| \leq \left(\frac{\delta}{n}\right)^2. \tag{A.10}$$

*Proof.* Since $\mu \in (0, 1]$, we find that for $\delta_{ij} < 1/2$

$$L_0(\delta_{ij}, \mu) = \max\left\{\left\lceil \frac{\log\left(\frac{1}{2\delta_{ij}}\right)}{\log\left(1+\frac{\mu}{2}\right)} \right\rceil, 0\right\} \leq \frac{5\log(1/\delta_{ij})}{2\mu}, \tag{A.11}$$

while for $\delta_{ij} \geq 1/2$ we have $L_0(\delta_{ij}, \mu) = 0$. Accordingly, we can weaken the bound on $2B(L/2, \delta, \mu)$ when the depth is set to $L = \alpha L_{\mathrm{conv}} \geq \frac{8\log(n/\delta)}{\mu}$, i.e. $\alpha \geq 1$. Taking the more nontrivial case of $\delta_{ij} < 1/2$, we have

$$\begin{aligned} 2B(L/2, \delta, \mu) &= \left(1 - \frac{\mu}{2}\right)^{\frac{8\alpha\log(n/\delta)}{2\mu} - \frac{5\log\left(1/\delta_{ij}\right)}{2\mu}} \\ &\leq \left(\frac{\delta}{n}\right)^{-8\alpha\log(1-\mu/2)/2\mu} \left(\frac{1}{\delta_{ij}}\right)^{-5\log(1-\mu/2)/2\mu}. \end{aligned} \tag{A.12}$$

Since $\mu \in (0, 1]$, we can take limiting cases of the exponents and observe that $\alpha = 1$ places the loosest bound. This gives for $\delta_{ij} < 1/2$,

$$2B(L/2, \delta, \mu) \leq \left(\frac{\delta}{n}\right)^2 \left(\frac{1}{\delta_{ij}}\right)^2. \tag{A.13}$$

Repeating the analysis with $L_0 = 0$ for $\delta_{ij} \geq 1/2$, we have $2B(L/2, \delta, \mu) \leq \left(\frac{\delta}{n}\right)^2$. Applying Theorem A.1.3 and noting that $L_0(\delta, \mu) \geq L_0(\delta_{ij}, \mu)$ when applying Lemma A.1.1, these results correspond to bounds on the NTK matrix element. □

Since $\delta = \min_{i,j} \delta_{ij}$, we have the following corollary.

**Corollary A.1.5** (Deep NTK upper bound)**.** *If $L \geq L_{\text{conv}}$, then $\left|\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}}\right| \leq \frac{1}{n^2}$ for all $i \neq j$.*

To show a lower bound, we must use some of the properties described in Remark A.1.2. We begin with a few simple properties of the dual activation function.

**Lemma A.1.6.** *For any $0 < \xi < 1$, the $h$-fold composition of the dual activation function with coefficient of nonlinearity $\mu$ satisfies $\hat{\sigma}^{(h)}(1-\xi) \geq (1-\mu)^h(1-\xi)$.*

*Proof.* Let $\rho = 1 - \xi > 0$. In the expansion $\hat{\sigma}(\rho) = \sum_{i=1}^{\infty} a_i^2 \rho^i$, each of the $a_i^2$ coefficients are nonnegative and $\mu = 1 - a_1^2$. Hence, $\hat{\sigma}(\rho) \geq a_1^2 \rho \geq (1-\mu)\rho$. To bound $\hat{\sigma}^{(h)}(\rho)$, we note that $\hat{\sigma}(\hat{\sigma}(\rho)) \geq \hat{\sigma}(z)$ for any $0 < z \leq \hat{\sigma}(\rho)$, and thus $\hat{\sigma}^{(h)}(\rho) \geq (1-\mu)^h \rho$. □

**Lemma A.1.7.** *For any $0 < \xi < 1$, the derivative of the dual activation function with coefficient of nonlinearity $\mu$ satisfies $\dot{\hat{\sigma}}(1-\xi) \geq 1 - \mu$.*

*Proof.* Let $\rho = 1 - \xi > 0$. Evaluating the derivative of the series given in Remark A.1.2, we find $\dot{\hat{\sigma}}(\rho) = \sum_{i=1}^{\infty} i a_i^2 \rho^{i-1} \geq a_1^2 = 1 - \mu$. Thus, $\dot{\hat{\sigma}}(1-\xi) \geq 1 - \mu$. □

We can now apply the definition of the NTK to compute a lower bound on a given matrix element.

**Theorem A.1.8** (Deep NTK lower bound)**.** *Consider any $\mathbf{x}_i, \mathbf{x}_j$ with separability $\delta_{ij}$ such that $\mathbf{x}_i \cdot \mathbf{x}_j > 0$. For an NTK of depth $L = L_{\text{conv}}$, we have*

$$\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}} \geq O(1) \cdot \delta_{ij} \left(\frac{1-\delta}{n}\right)^{O(1)}. \tag{A.14}$$

*Proof.* Since $\mathbf{x}_i \cdot \mathbf{x}_j > 0$ and the series expansion of Eq. A.3 has only positive coefficients per Remark A.1.2, the NTK element will be positive. From Theorem A.1.3, the diagonal of the NTK matrix is given by $(K_{\text{NTK}})_{ii} = \frac{\hat{\dot{\sigma}}(1)^{L+1}-1}{\hat{\dot{\sigma}}(1)-1}$ where $\hat{\dot{\sigma}}(1) > 1$. Simplifying notation by letting $\rho = \mathbf{x}_i \cdot \mathbf{x}_j$, we have

$$\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}} = \frac{\sum_{h=1}^{L+1} \hat{\sigma}^{(h-1)}(\rho) \left( \prod_{h'=h}^{L} \hat{\dot{\sigma}}(\hat{\sigma}^{(h')}(\rho)) \right)}{\frac{\hat{\dot{\sigma}}(1)^{L+1}-1}{\hat{\dot{\sigma}}(1)-1}} \tag{A.15}$$

$$\geq \frac{\hat{\dot{\sigma}}(1)-1}{\hat{\dot{\sigma}}(1)^{L+1}-1} \min_{h \in [L+1]} (1-\mu)^{h-1} \rho \left( \prod_{h'=h}^{L} (1-\mu) \right) \tag{A.16}$$

$$\geq \frac{\hat{\dot{\sigma}}(1)-1}{\hat{\dot{\sigma}}(1)^{L+1}-1} (1-\mu)^{L} \rho. \tag{A.17}$$

Taking $L = L_{\text{conv}} = \frac{8 \log(n/\delta)}{\mu}$ and using $0 < \mu \leq 1$, this gives the lower bound

$$\frac{(K_{\text{NTK}})_{ij}}{(K_{\text{NTK}})_{11}} \geq \delta_{ij} (\hat{\dot{\sigma}}(1)-1) \cdot \frac{(1-\mu)^{8 \log(n/\delta)/\mu}}{\hat{\dot{\sigma}}(1)^{8 \log(n/\delta)/\mu} \cdot \hat{\dot{\sigma}}(1) - 1} \tag{A.18}$$

$$\geq \delta_{ij} (\hat{\dot{\sigma}}(1)-1) \cdot \frac{(n/\delta)^{8 \log(1-\mu)/\mu}}{(n/\delta)^{8\hat{\dot{\sigma}}(1)/\mu} \cdot \hat{\dot{\sigma}}(1) - 1} \tag{A.19}$$

$$\geq O(1) \cdot \delta_{ij} \left( \frac{\delta}{n} \right)^{O(1)}, \tag{A.20}$$

since $\mu$ and $\hat{\dot{\sigma}}(1)$ are constants. □

**Gaussian-distributed output of the NTK**

We briefly comment on the output distribution of a trained NTK in the limit of $t \to \infty$ (Lemma 1.1.1 of the main text). Consider a test data example $\mathbf{x}_* \in \mathbb{R}^n$, and let the corresponding evaluations of the kernel between $\mathbf{x}_*$ and the training set $\mathcal{S}$ be denoted by $(\mathbf{k}_{\text{NTK}})_*, (\mathbf{k}_{\text{cov}})_* \in \mathbb{R}^n$. Since the neural network is initialized as a Gaussian distribution and the NTK describes an affine transform, a neural network with linearized dynamics (i.e. in the wide limit) will have Gaussian-distributed output. In particular, Corollary 1 of Lee et al. [7] gives the mean and variance of the Gaussian output $f_*$ of the converged NTK as $t \to \infty$:

$$\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} \mathbf{y} \tag{A.21}$$

$$\begin{aligned} \mathbb{V}[f_*] = K_{\text{cov}}(\mathbf{x}_*, \mathbf{x}_*) + (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} K_{\text{cov}} K_{\text{NTK}}^{-1} (\mathbf{k}_{\text{NTK}})_* \\ - ((\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} (\mathbf{k}_{\text{cov}})_* + h.c.), \end{aligned} \tag{A.22}$$

where *h.c.* denotes the Hermitian conjugate. Computing the output of a trained wide neural network thus consists of computing these two quantities.

**Neural network depth for the convergence of the NTK**

We now provide a more formal statement of Theorem 1.1.3 and, while the result is standard, we outline the proof due the widespread usage of $L_{\text{conv}}$ throughout our work. Since Agarwal et al. [4] do not comment on $L = \Omega\left(L_{\text{conv}}\right)$ specifically being a *lower* bound on the required neural network depth to converge via gradient descent, we wish to justify that particular argument in this section. As in our quantum algorithm, we assume a squared loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$ defines the empirical loss function over the neural network parameterized by weights $\vec{W}$:

$$\mathcal{L}(\vec{W}) := \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\vec{W}}(\mathbf{x}_i),\, y_i). \tag{A.23}$$

For smooth activation functions, we use a result of Lee et al. [7], reproduced here directly from Agarwal et al. [4]:

**Theorem A.1.9** (Convergence via gradient descent)**.** *Suppose that the activation $\sigma$ and its derivative $\sigma'$ further satisfy the properties that there exists a constant $c$, such that for all $\mathbf{x}_i, \mathbf{x}_j$*

$$|\sigma(\mathbf{x}_i)|, |\sigma'(\mathbf{x}_i)|, \frac{|\sigma'(\mathbf{x}_i) - \sigma'(\mathbf{x}_j)|}{|\mathbf{x}_i - \mathbf{x}_j|} \leq c.$$

*Then there exists a constant $N$ (depending on $L$, $n$, $\delta$) such that for width $m > N$ and setting the learning rate $\eta = 2(\lambda_{\min}(K_{\text{NTK}}) + \lambda_{\max}(K_{\text{NTK}}))^{-1}$, with high probability over the initialization the following is satisfied for gradient descent for all $t$,*

$$\mathcal{L}(\vec{W}(t)) \leq e^{-\Omega\left(\frac{t}{\kappa(K_{\text{NTK}})}\right)} \mathcal{L}(\vec{W}(0))$$

It thus suffices to show that the NTK is well-conditioned for $L = \Omega\left(\frac{\log(n/\delta)}{\mu}\right)$ in order to show that $\mathcal{L}(\vec{W}(t))$ to converge to $\mathcal{L}(\vec{W}(0))$ via gradient descent. This follows directly from the bounds placed on the maximum eigenvalue (Lemma A.2.1) and minimum eigenvalue (Lemma A.2.2). Since increasing $L$ only increases the minimum eigenvalue and decreases the maximum eigenvalue, we conclude that $L = \Omega\left(\frac{\log(n/\delta)}{\mu}\right) = \Omega(L_{\text{conv}})$ is a lower bound on the necessary neural network depth. Throughout this paper, we will apply this resulting depth to demonstrate the relevance of the regime for which a quantum speedup is obtained.

**NTK normalization**

We must determine the scaling of NTK matrix elements as the training set size increases. Two intermediate results essential to the quantum algorithm are provided, describing the behavior of ratios between NTK elements and a normalization factor that will be used for post-selection of quantum states.

**Lemma A.1.10.** *Given constants $\epsilon, \epsilon'$ such that $0 < \epsilon' < \epsilon < 1$, the ratio of neural tangent kernels $r = \hat{K}_{\text{NTK}}(1-\epsilon)/\hat{K}_{\text{NTK}}(1-\epsilon') = \Omega(1/\text{poly } L)$ for $L \geq L_{\text{conv}}$. In particular, if $L = L_{\text{conv}}$, then $r = \Omega(1/\text{polylog } n)$.*

*Proof.* When computing an NTK matrix element with Eq. A.3, the composition of dual activation functions (and its derivative) is positive since the coefficients of each series are positive by Remark A.1.2. Hence, $r$ is positive. Using induction, it can be shown that $\hat{\sigma}^{(L)}(1-\epsilon')/\hat{\sigma}^{(L)}(1-\epsilon) = O(\text{poly } L)$ and thus $\hat{\dot{\sigma}}\left(\hat{\sigma}^{(L)}(1-\epsilon')\right)/\hat{\dot{\sigma}}\left(\hat{\sigma}^{(L)}(1-\epsilon)\right) = O(\text{poly } L)$. By Eq. A.3, this implies a valid base case for the inductive assumption that $\hat{K}_{\text{NTK}}(1-\epsilon')/\hat{K}_{\text{NTK}}(1-\epsilon) = O(\text{poly } L)$. Checking $L+1$, we have

$$\frac{K_{L+1}(1-\epsilon')}{K_{L+1}(1-\epsilon)} = \frac{\hat{\dot{\sigma}}(\hat{\sigma}^{(L+1)}(1-\epsilon'))K_L(1-\epsilon') + \hat{\sigma}^{(L+1)}(1-\epsilon')}{\hat{\dot{\sigma}}(\hat{\sigma}^{(L+1)}(1-\epsilon))K_L(1-\epsilon) + \hat{\sigma}^{(L+1)}(1-\epsilon)} = O(\text{poly } L) \tag{A.24}$$

and thus $\hat{K}_{\text{NTK}}(1-\epsilon')/\hat{K}_{\text{NTK}}(1-\epsilon) = O(\text{poly } L)$. Taking the inverse for the ratio $r$ defined in the lemma statement, we have $r = \Omega(1/\text{poly } L)$. From our data assumptions, $\delta = \Omega(1/\text{poly } n)$, and thus taking $L = 8\log(n/\delta)/\mu$ ensures that $r = \Omega(1/\text{polylog } n)$. □

Lemma A.1.10 ensures proper normalization of the quantum states to prevent exponentially small state overlap. Since the result is critical to establishing the exponential speedup, we also illustrate an empirical example of the scaling from a dataset sampled uniformly on a 10-dimensional sphere (Fig. A.1). For ease of visualization, the figure shows $1/r = \hat{K}_{\text{NTK}}(1-\epsilon')/\hat{K}_{\text{NTK}}(1-\epsilon)$, which scales more slowly than $\log n$ for various values of $\epsilon, \epsilon'$.

Finally, we find that the normalization coefficient required to create valid quantum states.

**Corollary A.1.11.** *Define the NTK ratio $r(\rho) = \hat{K}_{\text{NTK}}(\rho)/\hat{K}_{\text{NTK}}(1-\epsilon')$ clipped to enforce $-1 \leq r \leq 1$. Define $P = \sum_{i=0}^{n-1} r^2(\rho_i)$, where $\rho_i = \mathbf{x}_i \cdot \mathbf{x}_*$ for*

*some* $\mathbf{x}_*$ *sampled i.i.d. from the same distribution as* $\mathbf{x}_i$*. We have the bounds* $P_0 \leq P \leq n$ *for some* $P_0 = \Omega(n/\text{polylog}(n))$*.*

*Proof.* The upper bound trivially follows from the clipping of $r$. By Lemma A.1.10, each term in $P$ contributes $\Omega(1/\text{polylog } n)$ to the sum; note that when applying Lemma A.1.10 to negative $\rho$, the same scaling with $L$ occurs. Summing over $n$ such terms, we find a lower bound of size $\Omega(n/\text{polylog } n)$. □

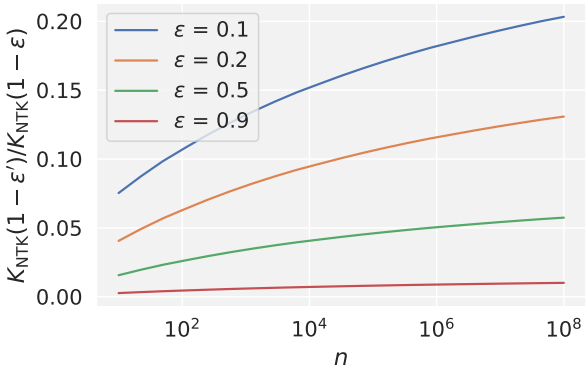



Figure A.1: Illustration of Lemma A.1.10 showing $1/r = \hat{K}_{\text{NTK}}(1-\epsilon')/\hat{K}_{\text{NTK}}(1-\epsilon)$ vs. $n$. The NTK is computed at neural network depth $L = L_{\text{conv}}$ (with $\mu = 0.5$) and the we fix $\epsilon' = 0.01$ while showing different values of $\epsilon$. The scaling is observed to be bounded by a logarithmic function (i.e. a straight line), consistent with the lemma.

## A.2 Computing the diagonal NTK approximation

To evaluate the expectation of the exact NTK, we must evaluate $\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1}\mathbf{y}$. To efficiently approximate such an infinitely wide neural network, we wish to show that $\mathbb{E}[f_*]$ is well-approximated by a value proportional to $(\mathbf{k}_{\text{NTK}})_*^T\mathbf{y}$, i.e. dropping the matrix inverse $K_{\text{NTK}}^{-1}$. It is this simpler inner product that we will evaluate using the quantum algorithm. As seen below, further reduction in error is also enabled by inverting a sparse matrix $\tilde{K}_{\text{NTK}}^{-1}$ rather than replacing it with the identity.

### Eigenvalue bounds of the NTK

We require bounds on the maximum and minimum eigenvalues of the NTK in order to compute error bounds on the NTK approximation evaluated by the quantum algorithm.

**Lemma A.2.1** (Maximum eigenvalue of NTK)**.** *If $L \geq L_{\text{conv}}$, then $\lambda_{\max}(K_{\text{NTK}}) \leq (K_{\text{NTK}})_{11}(1 + 1/n)$.*

*Proof.* As given by Theorem A.1.3, the diagonal elements of the NTK are equal and larger than the off-diagonal elements, since Lemma A.1.1 guarantees sufficient neural network depth $L_{\text{conv}} \geq 2L_0(\delta, \mu)$. By the Gershgorin circle theorem, $\lambda_{\max} \leq (K_{\text{NTK}})_{11}[1 + (n-1)(2B(L/2, \delta, \mu))]$. Applying Corollary A.1.5, this gives an upper bound of $\lambda_{\max} \leq (K_{\text{NTK}})_{11}[1 + (n-1)/n^2] \leq (K_{\text{NTK}})_{11}(1 + 1/n)$. □

**Lemma A.2.2** (Minimum eigenvalue of NTK)**.** *If $L \geq L_{\text{conv}}$, then $\lambda_{\min}(K_{\text{NTK}}) \geq (K_{\text{NTK}})_{11}(1 - 1/n)$.*

*Proof.* Similarly to above, the Gershgorin circle theorem with Corollary A.1.5 gives $\lambda_{\min}(K_{\text{NTK}}) \geq (K_{\text{NTK}})_{11}[1 - (n-1)/n^2] \geq (K_{\text{NTK}})_{11}(1 - 1/n)$. □

From these bounds, we conclude that the NTK is well-conditioned when representing a neural network deep enough to converge, consistent with the result of Agarwal et al. [4].

**Corollary A.2.3** (Conditioning of NTK)**.** *The condition number $1 \leq \kappa(K_{\text{NTK}}) \leq \frac{1+1/n}{1-1/n}$ converges to unity as $n \to \infty$.*

**Efficient computation of an NTK element**

In addition to the above properties of the NTK matrix, data separability ensures that a single element of the NTK.

**Lemma A.2.4** (Efficient NTK element computation)**.** *If $L = \Theta(L_{\text{conv}})$ and $\delta = \Omega(1/\text{poly}\, n)$, then an element of the NTK can be computed in $O(\text{polylog}\,(n)/\mu)$ time given the inner product between two data points.*

*Proof.* By Eq. A.3, a polynomial number of operations in $L$ are required to evaluate the NTK between two data points. Since $L_{\text{conv}} = O(\log(n/\delta)/\mu)$, choosing $\delta = O(1/\text{poly}\, n)$ ensures that $L = \Theta(L_{\text{conv}}) = \Theta(\text{polylog}(n)/\mu)$. Thus, an element of the NTK matrix can be computed in $O(\text{polylog}(n)/\mu)$ time given the inner product between data. An example of a dataset satisfying this condition is described in the main text and further discussed in Sec. A.5. □

**Convergence to exact NTK**

To bound the error caused by sparsifying the NTK, we require a result on matrix inverses (see Demmel [8] for a proof).

**Lemma A.2.5** (Perturbation of matrix inverses)**.** *Let $A$ be an $n \times n$ real matrix. A small perturbation $\epsilon X$ to $A$ causes a small perturbation of $A^{-1}$ bounded in spectral norm by*

$$\frac{||(A+\epsilon X)^{-1} - A^{-1}||}{||A^{-1}||} \leq \kappa(A) \cdot \frac{||\epsilon X||}{||A||} + O(||\epsilon X||^2). \tag{A.25}$$

**Theorem A.2.6** (Convergence to the exact NTK)**.** *Let $M = (K_{\text{NTK}})_{11} \cdot I$ be proportional to the $n \times n$ identity matrix. The error of the matrix inverse vanishes as $\frac{||M - K_{\text{NTK}}^{-1}||}{||K_{\text{NTK}}^{-1}||} = O(1/n)$.*

*Proof.* Define $n \times n$ matrix $A = K_{\text{NTK}}/(K_{\text{NTK}})_{11}$ and let $\epsilon X = I - A$. Since $A$ has a unit diagonal, $\epsilon X$ has a zero diagonal. By Corollary A.1.5, all elements of $\epsilon X$ are bounded in magnitude by $1/n^2$. By the Gershgorin circle theorem, the maximum eigenvalue of $X$ is thus $1/n$. Applying the results of Sec. A.2 and Lemma A.2.5, we find that

$$\frac{||(A+\epsilon X)^{-1} - A^{-1}||}{||A^{-1}||} \leq \frac{1+1/n}{1-1/n} \cdot \frac{1/n}{1+1/n} + O(1/n^2) = O(1/n). \tag{A.26}$$

Since $K_{\text{NTK}} = (K_{\text{NTK}})_{11} A$, this gives the required relation for the NTK itself. Hence, the error vanishes rapidly with a polynomial increase in dataset size. □

**Corollary A.2.7.** *For a training dataset of size $n$, the expectation of an infinite-width neural network $f$ of depth $L \geq L_{\text{conv}}$ on test data $\mathbf{x}_*$ can be estimated as $\mathbb{E}[f_*] \approx (\mathbf{k}_{\text{NTK}})_*^T \mathbf{y}$ up to $O(1/n)$ error.*

Finally, we will make the additional approximation caused by the clipping of NTK elements within a distance $1 - \mathbf{x}_i \cdot \mathbf{x}_* \leq \epsilon'$, where $\epsilon' < \epsilon$ is within the bound for which we expect $y_i = y_*$.

**Theorem A.2.8** (Approximate NTK)**.** *Let $\epsilon'$ be a constant such that $0 < \epsilon' < \epsilon < 1$, where $\epsilon$ denotes the neighborhood around $\mathbf{x}_*$ such that $y_i = y_*$ with high probability (see Assumption A.1.3). Define $(\tilde{\mathbf{k}}_{\text{NTK}})_*$ as the NTK evaluated between the single data point $\mathbf{x}_*$ and all examples $\mathbf{x}_i$ in the training set, with an additional clipping constraint to ensure all vector elements are*

*less than or equal to* $\hat{K}_{\mathrm{NTK}}(1-\epsilon')$ *in magnitude. There exists sufficiently large data dimension* $d$ *of order* $O(\log n)$ *such that the inner product* $(\tilde{\mathbf{k}}_{\mathrm{NTK}})_*^T\mathbf{y}$ *approximates* $(\mathbf{k}_{\mathrm{NTK}})_*^T\mathbf{y}$ *up to* $O(1/\mathrm{poly}\ n)$ *error.*

*Proof.* Per Assumption A.1.3, there exists constant $\epsilon$ such that for all $i$ in the $\epsilon$-neighborhood $N_\epsilon = \{i : 1 - \mathbf{x}_i \cdot \mathbf{x}_* \geq \epsilon\}$, we have $y_i = y_*$ with high probability. Moreover, since the data is approximately uniformly distributed within the $\epsilon$-neighborhood, $|N_\epsilon|$ can be approximated to be proportional to the area of a spherical cap on a $d$-dimensional sphere.

We now consider a similar argument for $\epsilon'$. Note that if $\epsilon'$ is sufficiently small that *no* examples in the training set satisfy $\mathbf{x}_* \cdot \mathbf{x}_i \geq 1 - \epsilon'$, then no truncation occurs and the theorem is trivially satisfied. Truncation occurs beyond magnitude $\hat{K}_{\mathrm{NTK}}(1-\epsilon')$. Since $0 < \epsilon' < \epsilon < 1$, the error $\xi$ introduced by truncation is given by the excess magnitude within the $\epsilon'$-neighborhood $N_{\epsilon'}$, i.e. $\xi \sim \sum_{i \in N_{\epsilon'}} \left( \frac{K_{\mathrm{NTK}}(\mathbf{x}_i, \mathbf{x}_*)}{\hat{K}_{\mathrm{NTK}}(1-\epsilon')} - 1 \right)$. Applying the matrix element bounds of Theorem A.1.8 and Corollary A.1.5, we have

$$\sum_{i \in N_{\epsilon'}} \frac{K_{\mathrm{NTK}}(\mathbf{x}_i, \mathbf{x}_*)}{\hat{K}_{\mathrm{NTK}}(1-\epsilon')} \leq |N_{\epsilon'}| \cdot \frac{\hat{K}_{\mathrm{NTK}}(1-\delta)}{\hat{K}_{\mathrm{NTK}}(1-\epsilon')} \leq O(1)|N_{\epsilon'}| \cdot \frac{1/n^2}{(1-\epsilon')(\delta/n)^{O(1)}}. \tag{A.27}$$

Since $\delta = \Omega(1/\mathrm{poly}\ n)$, the error is upper-bounded by a term scaling like $|N_{\epsilon'}| \cdot n^{O(1)}$. As argued for $N_\epsilon$, the size $|N_{\epsilon'}|$ is well-approximated by the area of a spherical cap. Writing such an area in terms of regularized beta functions, we find fractional error

$$\frac{|N_{\epsilon'}|}{|N_\epsilon|} \cdot n^{O(1)} \approx \frac{I_{1-(1-\epsilon')^2}((d-1)/2, 1/2)}{I_{1-(1-\epsilon)^2}((d-1)/2, 1/2)} \cdot n^{O(1)} \leq \left( \frac{\epsilon'}{2\epsilon} \right)^{(d-1)/2} \cdot n^{O(1)} \tag{A.28}$$

$$\leq \left( \frac{2}{d-1} \right)^{\log(2\epsilon/\epsilon')} \cdot n^{O(1)}. \tag{A.29}$$

Hence, for sufficiently large data dimension $d$ of size $O(\log n)$, the error introduced by clipping $(\mathbf{k}_{\mathrm{NTK}})_*$ will be suppressed by small $\epsilon'$. □

### A.3 Quantum algorithm

To evaluate the inner product $\mathbb{E}[f_*] \approx (\mathbf{k}_{\mathrm{NTK}})_*^T\mathbf{y}$, we require several standard quantum linear algebra routines. The results of the below theorems are empirically seen in Sec. A.6 to verify that the quantum algorithm is efficient and converges to the exact NTK.

**Quantum random access memory**

A key feature of attractive applications in quantum machine learning is achieving polylogarithmic dependence on training set size. However, the initial encoding of a training set trivially requires linear time, since each data example must be recorded once. To ensure that this linear overhead only occurs once, quantum random access memory (QRAM) can be used to prepare a classical data structure once and then efficiently read out data with quantum circuits in logarithmic time. We use the binary tree QRAM subroutine proposed by Kerenidis and Prakash [9] and applied commonly in quantum machine learning [10, 11]. The QRAM consists of a classical data structure that encodes a data matrix $S \in \mathbb{R}^{n \times d}$ with efficient *quantum access*.

**Definition A.3.1** (Quantum access)**.** Let $|S_i\rangle = \frac{1}{||S_i||} \sum_{j=0}^{d-1} S_{ij} |j\rangle$ denote the amplitude encoding of the $i$th row of data $S \in \mathbb{R}^{n \times d}$. Quantum access provides the mappings

- $|i\rangle |0\rangle \mapsto |i\rangle |S_i\rangle$
- $|0\rangle \mapsto \frac{1}{||S||_F} \sum_i ||S_i|| \, |i\rangle$

in time $T$ for $i \in [n]$.

The QRAM by Kerenidis and Prakash [9] provides quantum access in time $T$ that is polylogarithmic complexity with respect to both $n$ and $d$.

**Theorem A.3.1** (QRAM)**.** *For $S \in \mathbb{R}^{n \times d}$, there exists a data structure that stores $S$ such that the time to insert, update or delete entry $S_{ij}$ is $O(\log^2(n))$. Moreover, a quantum algorithm with access to the data structure provides quantum access in time $O(\text{polylog}(nd))$.*

Because the mapping $|i\rangle |0\rangle \mapsto |i\rangle |S_i\rangle$ is efficient, we can prepare a uniform superposition $\sum_{i=0}^{n-1} |i\rangle |0\rangle \mapsto \sum_{i=0}^{n-1} |i\rangle |S_i\rangle$ of the entire dataset. While preparing an arbitrary superposition is difficult, a uniform superposition is achieved with a constant-depth quantum circuit by applying Hadamard gates to all qubits. Hence, after a single $O(n)$ operation to prepare the data structure in QRAM, the dataset can be efficiently accessed by a quantum computer.

In our application of QRAM, we need to prepare states $|x\rangle = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |x_i\rangle$ and $|y\rangle = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} y_i |i\rangle$. For the state $|x\rangle$, Assumption A.1.4 ensures that

$||\mathbf{x}_i|| = 1$, allowing $|x\rangle$ to be directly prepared. For labels $y_i$, the classification problem ensures a known normalization factor $\sqrt{n}$.

**Preparation of kernel states**

To evaluate the neural network's prediction under the approximation $\mathbf{k}_*^T\mathbf{y}$, the NTK must be evaluated between a test data point $\mathbf{x}_*$ and the entire training set $\{\mathbf{x}_i\}$. In particular, we prepare the quantum state $|k_*\rangle = \sum_{i=0}^{n-1} |i\rangle\,|k_i\rangle$, where $k_i$ corresponds to an encoding of kernel elements $K_{\text{NTK}}(\mathbf{x}_*, \mathbf{x}_i)$ up to error $\xi$.

Since the NTK is only a function of the inner product $\rho_i = \mathbf{x}_\star \cdot \mathbf{x}_i$, we can use previous work on inner product estimation [10] to construct the kernel elements. By preparing this inner product in a quantum register, the NTK — which is efficient to compute classically on a single pair of data points by since $\delta = \Omega(1/\text{poly } n)$ — can be efficiently evaluated between the test data point and the entire training dataset. However, we first need the well-known subroutines of amplitude estimation [12] and median evaluation [13] as well as a basic translation from bitstring representations to amplitudes.

**Lemma A.3.2** (Amplitude estimation)**.** *Consider a quantum algorithm* $A : |0\rangle \mapsto \sqrt{p}\,|v,1\rangle + \sqrt{1-p}\,|g,0\rangle$ *for some garbage state* $|g\rangle$*. For any positive integer* $P$*, amplitude estimation outputs* $\tilde{p} \in [0,1]$ *such that*

$$|\tilde{p} - p| \leq 2\pi \frac{\sqrt{p(1-p)}P}{+} \left(\frac{\pi}{P}\right)^2 \tag{A.30}$$

*with probability at least* $8/\pi^2$ *using* $P$ *iterations of the algorithm* $A$*. If* $p = 0$*, then* $\tilde{p} = 0$ *with certainty, and similarly for* $p = 1$*.*

**Lemma A.3.3** (Median evaluation)**.** *Consider a unitary* $U : |0^{\otimes m}\rangle \mapsto \sqrt{\alpha}\,|v,1\rangle + \sqrt{1-\alpha}\,|g,0\rangle$ *for some* $1/2 < \alpha \leq 1$ *in time* $T$*. Then there exists a quantum algorithm that, for any* $\Delta > 0$ *and for any* $1/2 < \alpha_0 \leq \alpha$*, produces a state* $|\psi\rangle$ *such that* $||\,|\psi\rangle - \left|0^{\otimes mL}\right\rangle |x\rangle\,|| \leq \sqrt{2\Delta}$ *for some integer* $L$ *in time*

$$2T \left\lceil \frac{\log(1/\Delta)}{2(|\alpha_0| - 1/2)^2} \right\rceil. \tag{A.31}$$

**Lemma A.3.4** (Amplitude encoding)**.** *Given state* $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |k_i\rangle$ *with* $0 \leq k_i \leq 1$*, the state* $\frac{1}{\sqrt{P}} \sum_{i=0}^{n-1} k_i\,|i\rangle$ *may be prepared in time* $O(1/P)$ *with* $P = \sum_{i=0}^{n-1} k_i^2$*.*

*Proof.* We consider a single element $|k_i\rangle$ in the superposition $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |k_i\rangle$. Adding an ancilla to perform the map $|k_i\rangle\,|0\rangle \mapsto |k_i\rangle\,|\arccos k_i\rangle$, each bit of the

binary expansion $|k_i\rangle |\arccos k_i\rangle = |k_i\rangle |b_1\rangle \dots |b_m\rangle$ can be used as a rotation angle. Specifically, insert the ancilla $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and apply $m$ controlled rotations $\exp(ib_j\sigma^z/2^j)$ to obtain the state $|k_i\rangle |\arccos k_i\rangle (|k_i| |0\rangle + \sqrt{1-k_i^2} |1\rangle)$. By including an additional rotation controlled on the sign of $k_i$, the state $k_i |0\rangle + \sqrt{1-k_i^2} |1\rangle$ can be prepared. Applying the above in superposition, we have the state $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle \left(k_i |0\rangle + \sqrt{1-k_i^2} |1\rangle\right)$. Letting $P = \sum_{i=0}^{n-1} k_i^2$, post-selection on the final state gives $\frac{1}{\sqrt{P}} \sum_{i=0}^{n-1} k_i |i\rangle$ in time $O(1/P)$. □

Lemmas A.3.2 through A.3.4 provide the basic quantum computing background required. We may now prepare a quantum state corresponding to a superposition of $K_{\text{NTK}}(\mathbf{x}_*, \mathbf{x}_i)$ for all $i$ in the training set.

**Theorem A.3.5** (Kernel estimation)**.** *Let $S \in \mathbb{R}^{n\times d}$ be the training dataset of $\{\mathbf{x}_i\}$ unit norm vectors stored in the QRAM described in Theorem A.3.1. Consider the neural tangent kernel described in Eq. A.3 with coefficient of nonlinearity $\mu$. For test data vector $\mathbf{x}_* \in \mathbb{R}^d$ in QRAM and constant $\epsilon'$, there exists a quantum algorithm that maps*

$$\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle |0\rangle \mapsto \frac{1}{\sqrt{P}} \sum_{i=0}^{n-1} k_i |i\rangle . \tag{A.32}$$

*Here, $k_i = \hat{K}_{\text{NTK}}(\rho_i)/\hat{K}_{\text{NTK}}(1-\epsilon')$ is restricted to $-1 \le k_i \le 1$, i.e. clipping all $|\hat{K}_{\text{NTK}}(\rho_i)| > \hat{K}_{\text{NTK}}(1-\epsilon')$. The state is prepared with error $|\rho_i - \mathbf{x}_* \cdot \mathbf{x}_i| \le \xi$ with probability $1-2\Delta$ in time $\tilde{O}(\text{polylog}(nd)\log(1/\Delta)/\xi)$.*

*Proof.* Since the NTK is only a function of the inner product $\mathbf{x}_* \cdot \mathbf{x}_i$, we can compute the kernel elements after estimating the inner product between the test data and training data, following a similar approach to Kerenidis et al. [10]. Consider the initial state $|i\rangle \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) |0\rangle$. Using the QRAM as an oracle controlled on the second register, we can in $O(\text{polylog}(nd))$ time map $|i\rangle |0\rangle |0\rangle \mapsto |i\rangle |0\rangle |x_i\rangle$ and similarly $|i\rangle |1\rangle |0\rangle \mapsto |i\rangle |1\rangle |x_*\rangle$. (If $|x_*\rangle$ is not in QRAM, this operation only takes $O(d)$ time.) Applying a Hadamard gate on the second register, the state becomes

$$\frac{1}{2} |i\rangle (|0\rangle (|x_i\rangle + |x_*\rangle) + |1\rangle (|x_i\rangle - |x_*\rangle)) . \tag{A.33}$$

Measuring the second qubit in the computational basis, the probability of obtaining the $|1\rangle$ state is $p_i = \frac{1}{2}(1 - \langle x_i | x_\star \rangle)$ since the vectors are real-valued.

Writing the state $|1\rangle(|x_i\rangle - |x_*\rangle)$ as $|v_i, 1\rangle$, we have the mapping

$$A : |i\rangle|0\rangle \mapsto |i\rangle(\sqrt{p_i}\,|v_i, 1\rangle + \sqrt{1-p_i}\,|g_i, 0\rangle), \tag{A.34}$$

where $|g_i\rangle$ is a garbage state. The runtime of $A$ is $\tilde{O}(\text{polylog}(nd))$.

Applying amplitude estimation with $A$, we obtain a unitary $U$ that performs

$$U : |i\rangle|0\rangle \mapsto |i\rangle\left(\sqrt{\alpha}\,|\tilde{p}_i, g, 1\rangle + \sqrt{1-\alpha}\,|g', 0\rangle\right) \tag{A.35}$$

for garbage registers $g, g'$. By Lemma A.3.2, we have $|\tilde{p}_i - p_i| \leq \epsilon$ and $8/\pi^2 \leq \alpha \leq 1$ after $O(1/\epsilon)$ iterations. At this point, we now have runtime $\tilde{O}(\text{polylog}(nd)/\epsilon)$.

Applying median estimation, we finally obtain a state $|\psi_i\rangle$ such that $||\,|\psi_i\rangle - |0\rangle^{\otimes L}|\tilde{p}_i, g\rangle\,|| \leq \sqrt{2\Delta}$ in runtime $\tilde{O}(\text{polylog}(nd)\log(1/\Delta)/\epsilon)$. Performing this entire procedure but on the initial superposition $\sum_{i=0}^{n-1}|i\rangle\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle$, we now have the final state $\sum_{i=0}^{n-1}|\psi_i\rangle$.

Since $\left|\tilde{p}_i - \frac{1-\mathbf{x}_*\cdot\mathbf{x}_i}{2}\right| \leq \xi$, we can recover the inner product $\mathbf{x}_*\cdot\mathbf{x}_i$ as a quantum state. In general, there exists a unitary $V : \sum_x |x, 0\rangle \mapsto \sum_x |x, f(x)\rangle$ for any classical function $f$ with the same time complexity as $f$. Hence, we can choose $f$ that recovers $\mathbf{x}_*\cdot\mathbf{x}_i \approx 1 - 2\tilde{p}_i$ up to $O(\xi)$ error with probability $1 - 2\Delta$. Because $\delta = \Omega(1/\text{poly } n)$, evaluating the NTK between two data points takes time $O(\text{polylog}(n)/\mu)$ given their inner product. Again evaluating the classical function, we obtain the state $\frac{1}{\sqrt{n}}\sum_{i=0}^{n-1}|i\rangle|k_i\rangle$ where $k_i$ has $\leq O(\xi)$ error in time $\tilde{O}(\text{polylog}(nd)\log(1/\Delta)/\xi\mu)$.

Finally, we need to prepare the state $|k_*\rangle = \frac{1}{\sqrt{P}}\sum_{i=0}^{n-1}k_i|i\rangle$ for $P = \sum_i (k_i)^2$, where by Corollary A.1.11 we have $P = \Omega(n/\text{polylog } n)$. Applying Lemma A.3.4, preparing $|k_*\rangle$ requires $O(1/P)$ time, which is efficient in the training set size. □

**Efficient readout**

To estimate the sign of $(\mathbf{k}_{\text{NTK}})_*^T\mathbf{y}$ given our quantum states $|k_*\rangle$ and $|y\rangle$, pairwise measurements can be performed up to $O(1/n)$ error. To estimate the sign of $\langle k_*|y\rangle$, we encode the relative phase the states and perform an inner product estimation procedure such that $m$ measurements of the state gives $1/\sqrt{m}$ variance [14].

**Lemma A.3.6** (Inner product estimation)**.** *Given states $|k_*\rangle, |y\rangle \in \mathbb{R}^n$, estimating $\langle k_*|y\rangle$ with $m$ measurements has variance at most $1/\sqrt{m}$.*

*Proof.* Prepare initial state $\frac{1}{\sqrt{2}}(|0\rangle |k_*\rangle + |1\rangle |y\rangle)$. Applying a Hadamard gate to the first qubit, we obtain state $\frac{1}{2}(|0\rangle (|k_*\rangle + |y\rangle) + |1\rangle (|k_*\rangle - |y\rangle))$. Measuring the first qubit, the probability of obtaining $|0\rangle$ is $p = \frac{1}{2}(1 + \langle k_*|y\rangle)$. The binomial distribution over $m$ trials given this probability has variance $mp(1-p)$, and thus the variance of the estimate of $p$ is $p(1-p)/m$. Transforming to get the variance of the overlap estimate $\langle k_*|y\rangle$, we find variance of $\sqrt{\frac{1-(\langle k_*|y\rangle)^2}{m}}$. Since $(\langle k_*|y\rangle)^2 \leq 1$, this is upper-bounded by $1/\sqrt{m}$. □

To make sure the inner product has sufficient overlap for efficient readout, we provide the following result based on the data assumptions.

**Theorem A.3.7** (Efficient readout)**.** *Given states $|k_*\rangle$ and $|y\rangle$, a test example $\mathbf{x}_*$ can be classified up to $O(1/n)$ error with a logarithmic number of measurements in $n$.*

*Proof.* Under the identity matrix approximation of $K_{\text{NTK}}$, the prediction of $y_*$ is given by the sign of $\mathbf{k}_*^T\mathbf{y}$. As stated in Corollary A.2.7, this approximation is correct up to $O(1/n)$ error. By Theorem A.2.8, clipping the kernel evaluations of $\mathbf{k}_* \to \tilde{\mathbf{k}}_*$ with a $1-\epsilon'$ threshold only contributes $O(1/n)$ error for sufficiently high-dimensional data and sufficiently small $\epsilon'$. The inner product $\tilde{\mathbf{k}}_*^T\mathbf{y}$ is given by

$$\tilde{\mathbf{k}}_*^T\mathbf{y} = \langle k_*|y\rangle \cdot \sqrt{Pn}\frac{\hat{K}_{\text{NTK}}(1-\epsilon')}{(K_{\text{NTK}})_{11}}. \tag{A.36}$$

To show that $\langle k_*|y\rangle$ can be efficiently estimated by Lemma A.3.6, we must show that the overlap is at least $\Omega(1/\text{polylog } n)$. This is seen by breaking down states into positive and negative kernels and labels: $|k_*^+\rangle, |k_*^-\rangle, |y^+\rangle, |y_-\rangle$. The presence of an $\epsilon$-neighborhood implies that at least one of $|\langle k_*^+|y^+\rangle|^2$ or $|\langle k_*^+|y^-\rangle|^2$ is at least $\Omega(1/\text{polylog } n)$. By Corollary A.1.11, the normalization factor $P^+$ corresponding to $|k^+\rangle$ is upper-bounded by $n$, since we are summing over $O(n)$ terms of magnitude at least $\Omega(1/\text{polylog } n)$. By our data assumptions, $y_i = y_*$ with high probability in $N_\epsilon = \{i : \mathbf{x}_* \cdot \mathbf{x}_i \geq 1-\epsilon\}$. Since $\epsilon$ is a constant and examples are sampled i.i.d., $|N_\epsilon| = O(n)$. By Lemma A.1.10 we have $k_i^+ \geq k_0$ for some $k_0 = \Omega(1/\text{polylog } n)$. Hence, we have for one of $y_i^+$ or $y^-i$ (whichever corresponds to the majority label of neighborhood $N_\epsilon$) that

$$\frac{1}{\sqrt{P^+}\sqrt{n}}\left|\sum_{i\in N_\epsilon} k_i^+ y_i\right| = \frac{O(|N_\epsilon|)}{\sqrt{P^+}\sqrt{n}}\sum_{i\in N_\epsilon} k_i \geq O(1)k_0 \sim \Omega(1/\text{polylog } n). \tag{A.37}$$

Thus, we are guaranteed at least one efficiently measurable quantity between $|\langle k_*^+|y_+\rangle|^2$ and $|\langle k_*^+|y_-\rangle|^2$. We can expand $\langle k_*|y\rangle$ in terms of the positive and negative components

$$\langle k_*|y\rangle = \frac{\sqrt{P_+}}{\sqrt{(P_+ + P_-)(|Y_+| + |Y_-|)}}\Bigg(\sqrt{|Y_+|}\,|\langle k_*^+|y_+\rangle| - \sqrt{|Y_-|}\,|\langle k_*^+|y_-\rangle| - \sqrt{\frac{|Y_+|P_-}{P_+}}\,|\langle k_*^-|y_+\rangle| + \sqrt{\frac{|Y_-|P_-}{P_+}}\,|\langle k_*^-|y_-\rangle|\Bigg), \tag{A.38}$$

where $Y_+ = \{i : y_i = + = 1\}$ and similarly for $Y_-$. Since at least one of the terms has $\Omega(1/\text{polyylog } n)$ size and the probability of non-negligible terms fully cancelling is vanishingly small with $O(n)$ labels assigned to both $+1$ and $-1$, the product $\langle k_*|y\rangle$ will be of at least $\Omega(1/\text{polylog } n)$ size. Hence, we can apply Lemma A.3.6 to perform a polylogarithmic number of measurements and recover the sign of $\mathbf{k}_*^T\mathbf{y}$ up to $O(1/n)$ error by Eq. A.36.

□

## A.4 Computing the sparsified NTK approximation

We show that the neural tangent kernel satisfies the caveats required for QLSA to achieve an exponential speedup by inverting a sparsified NTK. Previously we considered the equation $\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1}\mathbf{y}$ by approximating $K_{\text{NTK}}^{-1}$ up to a matrix proportional to the identity, giving $O(1/n)$ error. Here, we approximate $K_{\text{NTK}}$ with a sparsification $\tilde{K}_{\text{NTK}}$ of the dense matrix. In particular, a logarithmic number of nonzero elements are allowed on each row and column, selecting the largest elements by an empirical threshold (see Sec. A.5). Although sparse matrix inversion is required, we use a quantum linear systems algorithm to perform it exponentially more quickly than is classically possible. In Sec. A.6, we verify numerically that the sparse NTK approximation converges more rapidly than the diagonal NTK approximation.

### Quantum linear systems algorithm

To *classically* solve the quantum linear systems problem (QLSP, Definition 1.1.6), a computational cost of at least $O(n)$ is required for a sparse, well-conditioned and positive definite $n \times n$ linear system. In particular, the conjugate gradient method [15] achieves $O(ns\sqrt{\kappa}\log(1/\xi))$ time to precision $\xi$ for a positive definite matrix. Proposed by Harrow, Hassidim and Lloyd, the HHL algo-

rithm [16] obtains an exponential speedup over this result. Informally, we summarize it as follows.

1. Prepare the data state $|b\rangle = \frac{1}{\sqrt{\mathbf{b}\cdot\mathbf{b}}} \sum_{i=0}^{n-1} b_i |i\rangle$. Writing $|b\rangle$ in the eigenbasis $\{\lambda_i, |\mu_i\rangle\}$ of $A$, define coefficients $\beta_i$ such that $|b\rangle = \sum_{i=0}^{n-1} \beta_i |\mu_i\rangle$.

2. Simulate the time evolution of $|b\rangle$ under the Hamiltonian defined by $A$ and apply a quantum Fourier Transform to obtain $|\psi_1\rangle = \sum_{i=0}^{n-1} \beta_i |\lambda_i\rangle |\mu_i\rangle$, where $|\lambda_i\rangle$ corresponds to a binary representation of $\lambda_i$ up to some precision.

3. Introduce an ancillary qubit in state $|0\rangle$ and perform a controlled rotation to obtain for some constant $C$

$$|\psi_2\rangle = \sum_{i=0}^{n-1} \beta_i |\lambda_i\rangle |\mu_i\rangle \left( \sqrt{1 - \frac{C^2}{\lambda_i^2}} |0\rangle + \frac{C}{\lambda_i} |1\rangle \right). \tag{A.39}$$

4. Reverse the phase estimation to uncompute $\lambda_i$ and measure the ancilla. If measurement yields the $|1\rangle$ state, then the final state of the system is $|\psi_3\rangle = |x\rangle = \sum_{i=0}^{n-1} \frac{\beta_i}{\lambda_i} |\mu_i\rangle$.

As described by Definition 1.1.6 in the main text, HHL solves the QLSP $A|x\rangle = |b\rangle$ corresponding to the linear equation $A\mathbf{x} = \mathbf{b}$. We write this formally in the following theorem; more details can be found in the summary provided by Dervovic et al. [17].

**Theorem A.4.1** (HHL algorithm)**.** *The quantum linear systems problem for $s$-sparse matrix $A \in \mathbb{R}^{n\times n}$ can be solved by a gate-efficient algorithm (i.e. with only logarithmic overhead in gate complexity) that makes $O(\kappa^2 s\mathrm{poly}(\log(s\kappa/\xi)/\xi))$ queries to the oracle $\mathcal{P}_A$ of the matrix $A$ and $O(\kappa s\mathrm{poly}(\log(s\kappa/\xi))/\xi)$ queries to the oracle to prepare the state corresponding to $\mathbf{b}$. Using a quantum random access memory for data access contributes a multiplicative factor of $O(\log n)$ to the runtime.*

As described by Childs et al. [18], the oracle $\mathcal{P}_A$ must perform the map

$$|j, \ell\rangle \mapsto |j, \nu(j, \ell)\rangle, \tag{A.40}$$

for any $j \in [n]$ and $\ell \in [s]$. The function $\nu : [n] \times [s] \to [n]$ computes the row index of the $\ell^{\text{th}}$ nonzero entry of the $j^{\text{th}}$ column. Note that in our case, the

NTK matrix is symmetric, so this is equivalent to compute a nonzero column index of a row.

We describe a suitable QRAM in the following subsection. Note that by replacing the phase estimation subroutine with algorithms based on Chebyshev polynomial decompositions, the dependence on precision can be improved. Similarly, improvements on the Hamiltonian simulation subroutine further improve the dependence on sparsity [19]. Based on these extension to HHL, QLSP can be solved in $O(\log(n)\kappa s \operatorname{polylog}(\kappa s/\xi))$ time [18].

As a result of the efficient kernel estimation, we can prepare the state $|k_*\rangle$ in polylogarithmic time in $n$. We now turn to the issue of constructing a sparse matrix $\tilde{K}_{\mathrm{NTK}}$ with a logarithmic number of nonzero elements in any row or column. To apply HHL, we need an efficient oracle $\mathcal{P}_A$ as required by Theorem A.4.1, which must report nonzero indices of any column in logarithmic time. In general, the NTK matrix is dense, and thus reporting any indices will suffice. However, we can efficiently choose *larger* nonzero indices: since the QRAM can be modified in logarithmic time, we can iteratively perform measurements that choose distinct data points closer to each other, causing a larger NTK matrix element.

**Lemma A.4.2** (NTK oracle)**.** *Let $\mathcal{P}_A$ denote an oracle that maps $|j,\ell\rangle \mapsto |j,\nu(j,\ell)\rangle$, i.e. computes the row index of the $\ell^{\text{th}}$ nonzero entry of the $j^{\text{th}}$ column in $A = K_{\mathrm{NTK}}$. Given a training set stored in QRAM, there exists a quantum circuit that implements the oracle in time polylogarithmic in the training set size.*

*Proof.* Assume we wish to find the nonzero elements of the $j$th column in the NTK matrix. By Theorem A.1.3, the diagonal element $(K_{\mathrm{NTK}})_{jj}$ is known to be nonzero (and equal to all other diagonal elements). To find up to $s$ other nonzero elements in the column, we temporarily remove $\mathbf{x}_j$ from the QRAM. By Theorem A.3.1, it requires $O(\log^2(n))$ time to remove a single element; to remove the entire vector $\mathbf{x}_j \in \mathbb{R}^d$ requires $O(d\log^2(n))$ time. We then prepare the state $\frac{1}{\sqrt{P}}\sum_i k_i |i\rangle$ by Theorem A.3.5, which has runtime polylogarithmic in $n$. Since $k_i \propto K_{\mathrm{NTK}}(\mathbf{x}_i, \mathbf{x}_j)$, measuring the state in the computational basis will cause it to collapse to $|i\rangle$ with probability proportional to $(K_{\mathrm{NTK}})^2_{jj}$. The index corresponding to the measured bitstring $i$ is taken to be nonzero. Although this simply biases the nonzero elements of our sparsified NTK to larger elements

of the exact NTK, we are still guaranteed that error will disappear by at most $O(1/n)$. Removing the measured index $i$ from the QRAM and repeating the process $O(\log n)$ times, the oracle $\mathcal{P}_A$ runs in $O(\log n)$ time. □

**Convergence of the sparsified NTK**

When returning $s = O(\log n)$ off-diagonal elements to the identity matrix approximation of the NTK, we require only minor modifications to the previously shown eigenvalue bounds.

**Lemma A.4.3** (Maximum eigenvalue of sparsified NTK)**.** *If $L \geq L_{\text{conv}}$, then $\lambda_{\max}(\tilde{K}_{\text{NTK}}) \leq (K_{\text{NTK}})_{11}(1 + 1/n)$.*

*Proof.* As given by Theorem A.1.3, the diagonal elements of the NTK are equal and larger than the off-diagonal elements, since Lemma A.1.1 guarantees sufficient neural network depth $L_{\text{conv}} \geq 2L_0(\delta, \mu)$. By the Gershgorin circle theorem, $\lambda_{\max} \leq (K_{\text{NTK}})_{11}[1 + s(2B(L/2, \delta, \mu))]]$. Applying Corollary A.1.5, this gives an upper bound of $\lambda_{\max} \leq (K_{\text{NTK}})_{11}[1 + s/n^2] \leq (K_{\text{NTK}})_{11}(1 + 1/n)$ since $s = O(\log n)$. □

**Lemma A.4.4** (Minimum eigenvalue of sparsified NTK)**.** *If $L \geq L_{\text{conv}}$, then $\lambda_{\min}(\tilde{K}_{\text{NTK}}) \geq (K_{\text{NTK}})_{11}(1 - 1/n)$.*

*Proof.* Similarly to above, the Gershgorin circle theorem with Corollary A.1.5 gives $\lambda_{\min}(K_{\text{NTK}}) \geq (K_{\text{NTK}})_{11}[1 - s/n^2] \geq (K_{\text{NTK}})_{11}(1 - 1/n)$. □

From these bounds, we conclude that the NTK is well-conditioned when representing a neural network deep enough to converge, consistent with the result of Agarwal et al. [4].

**Corollary A.4.5** (Conditioning of sparsified NTK)**.** *The condition number $1 \leq \kappa(\tilde{K}_{\text{NTK}}) \leq \frac{1+1/n}{1-1/n}$ converges to unity as $n \to \infty$.*

Hence, the sparsified NTK has a condition number that is well-suited to running HHL. Finally, we show that it converges to the exact NTK.

**Theorem A.4.6** (Convergence of the sparsified NTK to the exact NTK)**.** *Let $M = \tilde{K}_{\text{NTK}}$ be a sparsification of the exact NTK $\mathsf{K}_{\text{NTK}}$ with the complete diagonal and any subset of $s = O(\log n)$ off-diagonal elements. The error of the matrix inverse vanishes as $\frac{||\tilde{K}_{\text{NTK}}^{-1} - K_{\text{NTK}}^{-1}||}{||K_{\text{NTK}}^{-1}||} = O(1/n)$.*

*Proof.* Define matrices $A = K_{\text{NTK}}/(K_{\text{NTK}})_{11}$ and $\tilde{A} = \tilde{K}_{\text{NTK}}/(K_{\text{NTK}})_{11}$. Let $\epsilon X = \tilde{A} - A$. Since $A$ and $\tilde{A}$ both have unit diagonal, $\epsilon X$ has a zero diagonal. By Corollary A.1.5, all elements of $\epsilon X$ are bounded in magnitude by $1/n^2$. By the Gershgorin circle theorem, the maximum eigenvalue of $X$ is thus $1/n$. Applying the results of Sec. A.2 and Lemma A.2.5, we find that

$$\frac{||(A+\epsilon X)^{-1} - A^{-1}||}{||A^{-1}||} \leq \frac{1+1/n}{1-1/n} \cdot \frac{1/n}{1+1/n} + O(1/n^2) = O(1/n). \quad \text{(A.41)}$$

Since $K_{\text{NTK}} = (K_{\text{NTK}})_{11} A$, this gives the required relation for the NTK itself. Hence, the error vanishes rapidly with a polynomial increase in dataset size. □

Since we sparsify the NTK instead of replacing by a diagonal matrix, the Gerhsgorin circle theorem can be applied to the above results to find that the sparsified NTK is expected to converge slightly more quickly than the identity approximation; this is further assisted by the nonzero element oracle favoring larger off-diagonal elements.

**Corollary A.4.7** (Sparse approximation vs. diagonal approximation)**.** *Given sparse matrix $\tilde{K}_{\text{NTK}}$ with at most $O(\log n)$ nonzero off-diagonal elements in every row and column, define $\mathbb{E}[f_*^{\text{sparse}}] = (\mathbf{k}_{\text{NTK}})_*^T \tilde{K}_{\text{NTK}}^{-1} \mathbf{y}$. Under the diagonal approximation, define $\mathbb{E}[f_*^{\text{diag}}] = (\mathbf{k}_{\text{NTK}})_*^T \mathbf{y}/(K_{\text{NTK}})_{11}$. Compared to the exact NTK $\mathbb{E}[f_*] = (\mathbf{k}_{\text{NTK}})_*^T K_{\text{NTK}}^{-1} \mathbf{y}$ in expectation over $\mathbf{x}_*$, we have $|\mathbb{E}[f_*^{\text{sparse}}] - \mathbb{E}[f_*]| < |\mathbb{E}[f_*^{\text{diag}}] - \mathbb{E}[f_*]|$.*

Since the matrix error is the same as in Theorem A.2.6, the result of Theorem A.2.8 directly applies to the normalization of the initial state $|k_*\rangle$ and the consequent convergence of the NTK approximation. Moreover, since the modification only introduces $\log n$ number of elements bounded by $1/n$, the readout procedure (Theorem A.3.7) remains efficient with only polylogarithmic overhead.

## A.5 Datasets

To efficiently compute the NTK between data $\mathbf{x}_i, \mathbf{x}_j$ as is necessary to achieve an exponential speedup, we require $\delta = \Omega(1/\text{poly}\, n)$. Using a uniform distribution on a sphere, we motivate the power law $\delta(n) \approx a_1 n^{-a_2}$ with positive constants. We show such power laws to empirically hold on common datasets including MNIST and CIFAR-10, demonstrating that an exponential quantum

speedup can be achieved due to our dataset requirement of $\delta = \Omega(1/\text{poly } n)$ being satisfied.

**Uniform distribution on a sphere**

Define a dataset $\mathcal{S}$ of $n$ training examples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^d$ has fixed dimension and $y_i$ is bounded. Each $\mathbf{x}_i$ is sampled uniformly on the sphere $S^{d-1}$. We can define $\delta$ in terms of an $n \times n$ matrix $G$ defined similarly to the Gram matrix but with magnitudes of inner products, i.e. $G_{ij} = |\mathbf{x}_i \cdot \mathbf{x}_j|$. The minimum dataset separability is given by $1 - \rho_{\max}$, where $\rho_{\max}$ is the largest off-diagonal element of $G$.

Since the elements of $G$ are not drawn independently from a single distribution, we instead define a symmetric $n \times n$ matrix $A$ with elements drawn i.i.d. from the distribution of inner product magnitudes. We show that a power law $\delta(n) = a_1 n^{a_2}$ is satisfied for the matrix $A$.

**Lemma A.5.1.** *Let $A$ be a symmetric $n \times n$ matrix with elements $A_{ij}$ sampled from the distribution of inner products $|\mathbf{x}_i \cdot \mathbf{x}_j|$. Each matrix element is sampled i.i.d. with $\mathbf{x}_i, \mathbf{x}_j$ drawn uniformly at random from $S^{d-1}$ with $d \geq 2$. In the limit of large $n$, the separability $\delta = \min_{i,j}(1 - A_{ij})$ is lower-bounded by $\delta = \Omega(1/\text{poly}\, n)$. In particular, $\delta \approx \Omega(n^{4/(1-d)})$ to leading order in large $n$.*

*Proof.* We first determine the CDF of $|\mathbf{x}_i \cdot \mathbf{x}_j|$ for $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^d$ drawn uniformly at random from $S^{d-1}$. Without loss of generality, let $\mathbf{x}_i = (1, 0, \ldots, 0)$. Since the distribution is uniform on the surface of a sphere, symmetry under orthogonal matrix multiplication implies that we can let $\mathbf{x}_j = \frac{\mathbf{u}}{||\mathbf{u}||}$ for $\mathbf{u} \sim \mathcal{N}_d(0, 1)$ for $\mathbf{u} = (u_1, \ldots, u_d)$. Hence, the distribution of $\mathbf{x}_i \cdot \mathbf{x}_j$ is equivalent to the distribution of $\rho \sim u_1/\sqrt{u_1^2 + \cdots + u_d^2}$. Considering the random variable $\rho^2$, rearranging terms gives a ratio of $\chi^2$ variables and hence an $F$-distribution with 1 and $d - 1$ degrees of freedom. Evaluating the CDF for $\rho = |\mathbf{x}_i \cdot \mathbf{x}_j|$ in terms of the hypergeometric ${}_2F_1$ function gives

$$F(|\rho|) = \frac{2\Gamma\left(\frac{d}{2}\right)}{\sqrt{\pi}\Gamma\left(\frac{d-1}{2}\right)} {}_2F_1\left(\frac{1}{2}, \frac{3-d}{2}; \frac{3}{2}; |\rho|^2\right) |\rho|. \tag{A.42}$$

Suppose we sample from the distribution $m$ times, corresponding to the $m \approx n^2/2$ randomly chosen elements in the symmetric matrix $A$. To find the largest $|\rho|$ corresponding to the minimum separability, we seek the $(1 - 1/m)^{\text{th}}$ percentile of the $m$ elements. Following Mosteller's seminal work on order statis-

tics [20], the largest $|\rho|$ will be asymptotically normally distributed for large $m$, with a mean of $F^{-1}(1-1/m)$. Since we expect $|\rho|$ to converge to 1, we Taylor expand $|\rho| = 1-\delta$ around $\delta = 0$ to give

$$F(1-\delta) \approx 1 + \frac{2^{\frac{d-1}{2}}\delta^{\frac{d-1}{2}}\Gamma\left(\frac{d}{2}\right)}{\sqrt{\pi}\Gamma\left(\frac{d+1}{2}\right)}. \tag{A.43}$$

Solving for $\delta$ in $F(1-\delta) = 1-1/m$, we find that in expectation

$$\delta = \pi^{\frac{1}{d-1}}\left(\frac{2^{\frac{1}{2}-\frac{d}{2}}\Gamma\left(\frac{d+1}{2}\right)}{m\Gamma\left(\frac{d}{2}\right)}\right)^{\frac{2}{d-1}}. \tag{A.44}$$

Substituting back $m \approx n^2/2$ gives $\delta(n) = a_1 n^{a_2}$ with $a_2 = 4/(d-1)$. Taking a bounding case on $d$, we conclude that $\delta = \Omega(n^{-1}) = \Omega(1/\text{poly}\, n)$ for all $d \geq 3$. □

Although Lemma A.5.1 addresses an independently sampled matrix of inner products, we confirm that it empirically describes the spherical dataset (Fig. A.2).

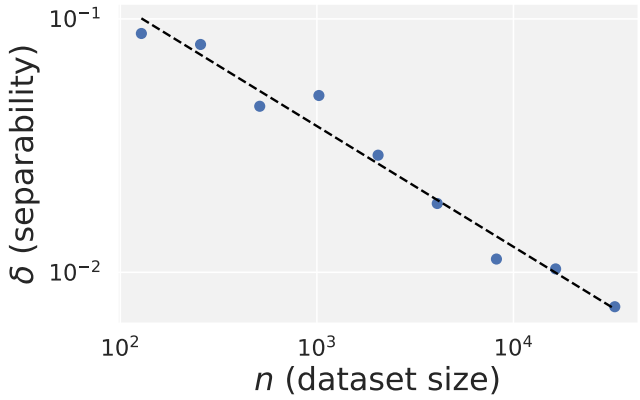



Figure A.2: Empirical fit of $\delta(n)$ for the sphere dataset with $d = 10$. We find $\delta(n) \approx 0.90n^{-0.46}$ ($R^2 = 0.96$), showing good agreement with the prediction of $\delta(n) \propto n^{-0.44}$ by Lemma A.5.1.

**Real-world datasets**

For MNIST, we consider the binary classification task between digits 0 and 1 then empirically determine $\delta(n)$ by subsampling the dataset. Similarly, for CIFAR-10, we consider the binary classification task between automobiles and

cats. As in the case of the uniform distribution on a sphere, both datasets cleary satisfy $\delta = \Omega(1/\text{poly } n)$ (Fig. A.3).

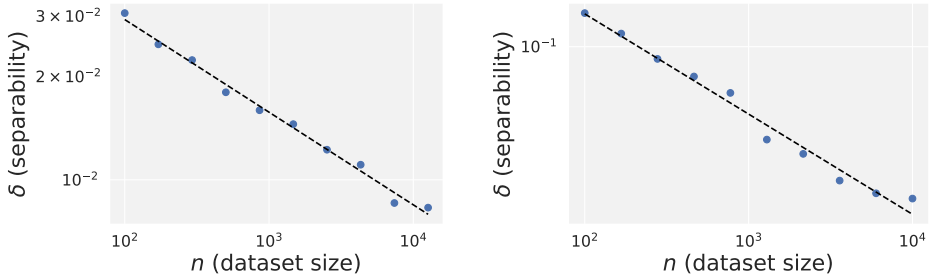


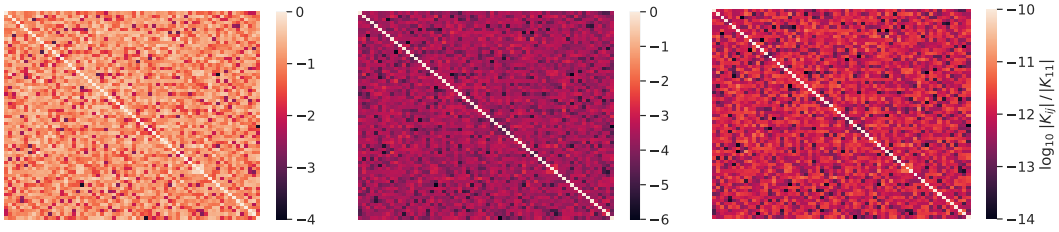

Figure A.3: Empirical fit of $\delta(n)$ for MNIST and CIFAR-10. We find $\delta(n) = 0.1n^{-0.3}$ ($R^2 = 0.991$) for MNIST and $\delta(n) = 3.0n^{-0.6}$ ($R^2 = 0.98$) for CIFAR-10.

## A.6 Numerical evaluation of the NTK

### Fully-connected neural tangent kernel (sphere dataset)

We consider a training dataset of examples $(\mathbf{x}_i, y_i)$ sampled from a uniform distribution on a sphere as in Sec. A.5, fixing the dimension $d = 3$. We take $y_i = f(\mathbf{x}_i) + \eta$ with $\eta \sim \mathcal{N}(0, 0.05)$ and $f(\mathbf{x}_i) = \sum_{j=1}^{d} \sin \frac{3\pi(\mathbf{x}_i)_j}{2}$ for $\mathbf{x}_i = (x_1, \ldots, x_d)$. For this dataset, we also consider Assumption A.1.3, which requires the label to vary smoothly at a given data resolution. We numerically determine $\epsilon$ such that the $\epsilon$-neighborhood $N_\epsilon = \{i : \mathbf{x}_* \cdot \mathbf{x}_i \geq 1 - \epsilon\}$ about some data point $\mathbf{x}_*$ satisfies $y_i = y_*$ with high probability. Choosing $\epsilon = 0.01$ on $d = 3$, for instance, would satisfy Assumption A.1.3 with over 90% probability. See Fig. 1.1 in the main text for a visualization of both the dataset and the choice of $\epsilon$.

The neural network is defined by Eq. 1.1 in the main text with an erf function used for activation ($\mu \approx 0.086$). We examine the performance of the sparsified NTK for shallower neural networks than $L_{\text{conv}}$ in order to probe larger datasets while remaining within reasonable time and memory constraints. Despite reducing the depth below the proven threshold, good convergence to the exact NTK is found. This convergence is likely due to the loose upper bound on the NTK matrix elements, which corresponds to an overestimate of the error introduced by sparsification. Consistent with the analysis of an increasingly strengthened diagonal, the NTK becomes well-conditioned for larger depths in Fig. A.4.




(a) Depth 5 (b) Depth 31 (c) Depth 100

Figure A.4: NTK matrix elements on the sphere dataset with the diagonal elements $K_{11} = K_{22} = \cdots = K_{nn}$ normalized to unity. As the neural network depth increases, the off-diagonal elements reduce in size compared to the diagonal, ensuring that the NTK is well-conditioned and that the prediction for a test example is based on its closest neighbors in the training set.

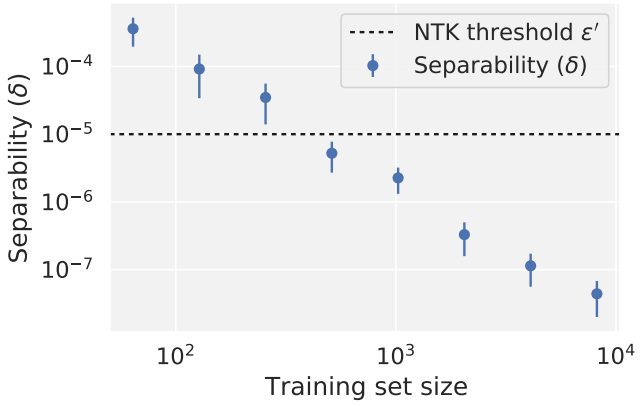



Figure A.5: Separability of the dataset for on a sphere with dimension $d = 3$, where the dashed line indicates $\epsilon'$. Kernel elements $\hat{K}_{\mathrm{NTK}}(\rho)$ are truncated to $\hat{K}_{\mathrm{NTK}}(1-\epsilon')$ where $\rho > 1-\epsilon'$, and thus datasets with size $n \geq 512$ onwards are subject to the normalization approximation of kernel vectors and matrix elements.

We may evaluate the impact of the approximations required by the quantum algorithm, as well as the theoretical results concerning the scaling of the number of measurements required to achieve bounded variance. To perform truncation with $\epsilon'$ for state normalization, we choose $\epsilon' = 10^{-5}$. This value is selected to be larger than the data separability for the larger datasets we examine (Fig. A.5). Thus, if truncating kernel elements by $\epsilon'$ were to reduce the NTK approximation's performance, it would be observed by a discrepancy

between the approximate NTK and the exact NTK when classifying larger datasets.

As discussed in the main text (see Fig. 1.2), preparing the state $|k_*\rangle = \frac{1}{\sqrt{P}} \sum_{i=0}^{n-1} k_i |i\rangle$ requires $O(1/P)$ measurements due to post-selection. For the toy dataset, this number of measurements is observed to decrease like $1/\text{poly } n$, as is consistent with Corollary A.1.11 albeit with a slightly different exponent due to a choice of $L \ll L_{\text{conv}}$ for the numerical simulation. Similarly, the state overlap $\langle k_*|y\rangle$ decreases like $\Omega(1/\text{polylog } n)$, ensuring that a logarithmic number of measurements are required.

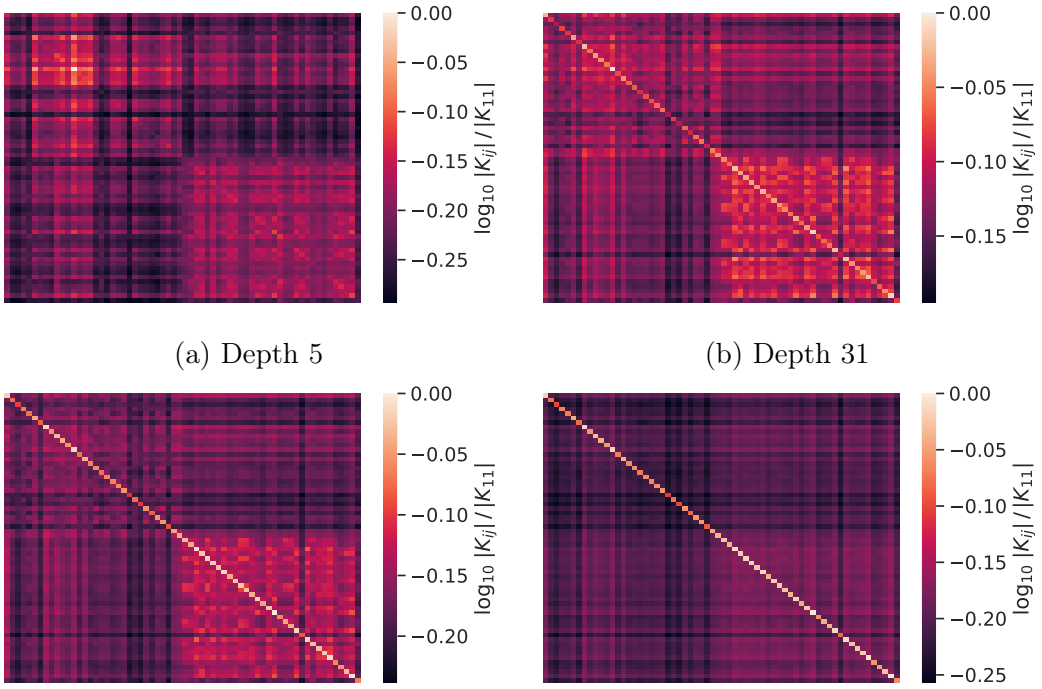
Finally, as expected, both the sparsified NTK (where $K_{\text{NTK}} \to \tilde{K}_{\text{NTK}}$ with a logarithmic number of nonzero elements each row/column) and the diagonal NTK (where $K_{\text{NTK}} \to (K_{\text{NTK}})_{11} I$ is diagonal) approximations are observed to converge to exactly the same output as the exact NTK (Fig. 1.3). At very small dataset sizes, the sparsified NTK performs better than the diagonal NTK, as previously discussed. For the sphere dataset, sparsification allows at most $\log n$ off-diagonal elements in the NTK.

**Convolutional neural tangent kernel (MNIST)**

To assess the generality of sparsification to more common deep learning architectures, we numerically evaluate the Myrtle [21] convolutional neural network on the MNIST dataset. Introduced as a standard benchmark architecture by Shankar et al. [22], the Myrtle architecture is a family of convolutional neural networks with ReLU activation functions, $3 \times 3$ convolutional filters, and $2 \times 2$ average pooling. Unlike in the vanilla fully-connected architecture, the Myrtle architecture does not satisfy the normalization condition of Assumption A.1.4 and does not yield and NTK with equal diagonal elements. For the task of binary classification, we one-hot encode two output neurons. Since sparsification can cause an effectively imbalanced dataset, a classification threshold on the difference between the two output neurons is used to decide the classification outcome. As described in the main text, a balanced training and test set are guaranteed, and thus the classification threshold is determined by the median of the output neuron values.

Similarly to the diminishing off-diagonal elements shown in Fig. A.4 for the vanilla fully-connected architecture, the convolutional neural network becomes well-conditioned and hence increasingly amenable to a quantum algorithm

with increasing depth (Fig. A.6).




(a) Depth 5

(b) Depth 31

(c) Depth 100

(d) Depth 1000

Figure A.6: NTK matrix elements on the MNIST binary classification dataset, with the maximum matrix element normalized to unity. As the neural network depth increases, the off-diagonal elements reduce in size compared to the diagonal, similarly to the simple neural network shown in Fig. A.4. The "0" class is placed in the top left corner and the "1" class is placed in the bottom right corner, illustrating the higher overlap within each class.

However, the Myrtle kernel does not experience as steep of a falloff as the vanilla fully-connected network; while off-diagonal elements shown in Fig. A.4 are at most a fraction $\sim 10^{-11}$ of the diagonal, deepening Myrtle from 5 to 1000 layers only causes the largest off-diagonal to reduce from 0.9 to 0.7 compared to the largest diagonal element. Hence, sparsification of $K_{\mathrm{NTK}}$ can cause a larger variation in the minimum eigenvalue of sparsified kernel $\tilde{K}_{\mathrm{NTK}}$, possibly even producing a zero or negative eigenvalue at the depths evaluated. Hence, we add diagonal regularization $\tilde{K}_{\mathrm{NTK}} \to \tilde{K}_{\mathrm{NTK}} + \sigma^2 I$ as is standard for improving matrix conditioning. While such conditioning could significantly impact performance, we find in Fig. 1.3 of the main text that the sparsified kernel remains competitive in performance to the exact kernel. For that NTK, we implement a Myrtle49 network although the required depth equivalent to

$L_{\text{conv}}$ is likely far deeper. Additionally, while $L_{\text{conv}}$ scales with $n$, we report experiments with fixed depth for MNIST. Although the diagonal is seen to dominate off-diagonal elements in Fig. A.6, the use of a Myrtle49 network can prevent the sparsified NTK (with $5 \log n$ nonzero off-diagonal elements) from being positive definite. Consequently, we amplify the diagonal by adding the identity matrix (scaled by the empirical NTK diagonal). This allows the sparsified Myrtle NTK to share the well-conditioned behavior of a deeper network.

## References

[1]Y. Li and Y. Liang, "Learning overparameterized neural networks via stochastic gradient descent on structured data", in *Advances in neural information processing systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018), pp. 8157–8166.

[2]Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization", in Proceedings of the 36th international conference on machine learning, Vol. 97, edited by K. Chaudhuri and R. Salakhutdinov, Proceedings of Machine Learning Research (Sept. 2019), pp. 242–252.

[3]D. Zou and Q. Gu, "An improved analysis of training over-parameterized deep neural networks", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 2055–2064.

[4]N. Agarwal, P. Awasthi, and S. Kale, *A deep conditioning treatment of neural networks*, 2020.

[5]A. Daniely, R. Frostig, and Y. Singer, "Toward deeper understanding of neural networks: the power of initialization and a dual view on expressivity", in Advances in neural information processing systems, Vol. 29, edited by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (2016), pp. 2253–2261.

[6]S. Arora, S. S. Du, W. Hu, Z. Li, R. R. Salakhutdinov, and R. Wang, "On exact computation with an infinitely wide neural net", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 8141–8150.

[7]J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 8572–8583.

[8]J. Demmel, "The componentwise distance to the nearest singular matrix", SIAM Journal on Matrix Analysis and Applications **13**, 10–19 (1992).

[9]I. Kerenidis and A. Prakash, "Quantum Recommendation Systems", in 8th innovations in theoretical computer science conference (itcs 2017), Vol. 67, edited by C. H. Papadimitriou, Leibniz International Proceedings in Informatics (LIPIcs) (2017), 49:1–49:21.

[10]I. Kerenidis, J. Landman, A. Luongo, and A. Prakash, "Q-means: a quantum algorithm for unsupervised machine learning", in Advances in neural information processing systems, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (2019), pp. 4134–4144.

[11]I. Kerenidis, J. Landman, and A. Prakash, "Quantum algorithms for deep convolutional neural networks", in International conference on learning representations (2020).

[12]G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, "Quantum amplitude amplification and estimation", Contemporary Mathematics **305**, 53–74 (2002).

[13]N. Wiebe, A. Kapoor, and K. Svore, *Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning*, 2014.

[14]L. Zhao, Z. Zhao, P. Rebentrost, and J. Fitzsimons, *Compiling basic linear algebra subroutines for quantum computers*, 2019.

[15]J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, tech. rep. (Carnegie Mellon University, USA, 1994).

[16]A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations", Phys. Rev. Lett. **103**, 150502 (2009).

[17]D. Dervovic, M. Herbster, P. Mountney, S. Severini, N. Usher, and L. Wossnig, *Quantum linear systems algorithms: a primer*, 2018.

[18]A. M. Childs, R. Kothari, and R. D. Somma, "Quantum algorithm for systems of linear equations with exponentially improved dependence on precision", SIAM Journal on Computing **46**, 1920–1950 (2017).

[19]D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters", in 2015 ieee 56th annual symposium on foundations of computer science (2015), pp. 792–809.

[20]F. Mosteller, "On Some Useful "Inefficient" Statistics", The Annals of Mathematical Statistics **17**, 377–408 (1946).

[21]D. Page, *Myrtle.ai*, 2018.

[22]V. Shankar, A. Fang, W. Guo, S. Fridovich-Keil, J. Ragan-Kelley, L. Schmidt, and B. Recht, "Neural kernels without tangents", in Proceedings of the 37th international conference on machine learning, Vol. 119, edited by H. D. III and A. Singh, Proceedings of Machine Learning Research (13–18 Jul 2020), pp. 8614–8623.

*Appendix B*

# CODE: QUANTUM NEURAL TANGENT KERNEL

In Chapter 1, we provided results of the neural tangent kernel (NTK) for a toy dataset on the sphere and the MNIST digit classification dataset. Here, we include code for creating the datasets and coputing the NTK, sparsified NTK, and diagonal NTK approximations.

```
from jax.config import config
config.update("jax_enable_x64", True)

import jax.numpy as np
from jax import jit
import functools
from jax import random
from jax.scipy.special import erf
import numpy as np2

import neural_tangents as nt
from neural_tangents import stax

import tensorflow_datasets as tfds
import itertools
from collections import namedtuple
from np.linalg import inv

seed = 12
key = random.PRNGKey(seed)

ds = tfds.as_numpy(
  tfds.load('mnist:3.*.*', batch_size=-1)
)

selection = (1, 0)
depth = 48
data_sizes = [8, 16, 32, 64, 128]
test_size = 64
batch_size = 4
trials = np.maximum(np.ceil(4096 / np.array(data_sizes)).astype(int),
                    256*np.ones(len(data_sizes)).astype(int))
```

```
33
34  # MNIST
35  def process_data(data_chunk, selection=[0, 1], class_size=None,
    ↪  shuffle=True):
36    # one-hot encode the labels and normalize the data.
37    global key
38    image, label = data_chunk['image'], data_chunk['label']
39    n_labels = 2
40
41    # pick two labels
42    indices = np.where((label == selection[0]) | (label == selection[1]))[0]
43
44    key, i_key = random.split(key, 2)
45    indices = random.permutation(i_key, indices).reshape(1, -1)
46
47    label = (label[tuple(indices)] == selection[0])
48
49    # balance if no class size is specified or class size too large
50    max_class_size = np.amin(np.unique(label, return_counts=True)[1])
51    if (class_size is None) or class_size > max_class_size:
52      class_size = max_class_size
53      print('class_size', class_size)
54
55    # select first class_size examples of each class
56    new_indices = []
57    for i in range(n_labels):
58      class_examples = np.where(label == i)[0]
59      new_indices += class_examples[:class_size].tolist()
60    key, j_key = random.split(key, 2)
61    if shuffle:
62      new_indices = random.permutation(j_key,
      ↪  np.array(new_indices)).reshape(1, -1)
63    else:
64      new_indices = np.array(new_indices).reshape(1, -1)
65
66    label = label[tuple(new_indices)].astype(np.int64)
67    label = np.eye(2)[label]
68
69    image = image[tuple(indices)][tuple(new_indices)]
70    image = (image - np.mean(image)) / np.std(image)
71    norm = np.sqrt(np.sum(image**2, axis=(1, 2, 3)))
72    image /= norm[:, np.newaxis, np.newaxis, np.newaxis]
73
```

```
74    return {'image': image, 'label': label}
75
76
77  # sphere dataset
78  noise_scale = 5e-2
79  key, x_key, y_key = random.split(key, 3)
80  def target_fn(x):
81    out = np.zeros(len(x))
82    for i in range(0, x.shape[1]):
83      out += np.sin((i+1)*np.pi*x[:, i]*0.75)
84    return np.reshape(out, (-1, 1))
85  def create_data(N, test_points=64, d=None, rand=False, rand_train=False):
86    global key
87    if d is None:
88      d = 3
89    if rand_train:
90      key, train_x_key, train_y_key = random.split(key, 3)
91    else:
92      train_x_key = x_key
93      train_y_key = y_key
94    train_xs = random.normal(train_x_key, (N, d))
95    norms = np.sqrt(np.sum(train_xs**2, axis=1))
96    train_xs = train_xs / np.repeat(norms[:, np.newaxis], d, axis=1)
97
98    train_ys = target_fn(train_xs)
99    train_ys += noise_scale * random.normal(train_y_key, (N, 1))
100   train = (train_xs, np.sign(train_ys))
101
102   if rand:
103     key, test_x_key, test_y_key = random.split(key, 3)
104     test_xs = random.normal(test_x_key, (test_points, d))
105     norms = np.sqrt(np.sum(test_xs**2, axis=1))
106     test_xs = test_xs / np.repeat(norms[:, np.newaxis], d, axis=1)
107   else:
108     # query points on a single path on the sphere surface
109     t = np.linspace(0, 2*np.pi, test_points)
110     test_x_0 = np.reshape(np.sin(t), (-1, 1))
111     test_x_1 = np.reshape(np.cos(t), (-1, 1))
112     test_xs = np.concatenate((test_x_0, test_x_1, np.zeros((test_points,
      ↪  d-2))),
113                 axis=1)
114
115   test_ys = target_fn(test_xs)
```

```
116   test = (test_xs, np.sign(test_ys))
117
118   return train, test
119
120
121 # MNIST neural network
122 def MyrtleNetworkMNIST(depth, W_std=np.sqrt(2.0), b_std=0.):
123   width = 1
124   activation_fn = stax.Relu()
125   layers = []
126   conv = functools.partial(stax.Conv, W_std=W_std, b_std=b_std,
      ↪  padding='SAME')
127   if depth == 4:
128     depths = [2, 1, 1]
129   else:
130     depths = [depth//3, depth//3, depth//3]
131
132   layers += [conv(width, (3, 3)), activation_fn] * depths[0]
133   layers += [stax.AvgPool((2, 2), strides=(2, 2))]
134   layers += [conv(width, (3, 3)), activation_fn] * depths[1]
135   layers += [stax.AvgPool((2, 2), strides=(2, 2))]
136   layers += [conv(width, (3, 3)), activation_fn] * depths[2]
137   layers += [stax.AvgPool((2, 2), strides=(2, 2))] * 2
138
139   layers += [stax.Flatten(), stax.Dense(2, W_std, b_std)]
140
141   return stax.serial(*layers)
142
143
144 # sphere neural network
145 def calc_var(key, s, samples=10000):
146   key, my_key = random.split(key)
147   x = random.normal(my_key, (samples,))
148   return np.mean(s(x)**2)
149 def calc_mu(key, s, samples=10000):
150   key, my_key = random.split(key)
151   x = random.normal(my_key, (samples,))
152   return 1 - np.mean(x*s(x))**2
153 scale = calc_var(key, erf, samples=10**8)
154 mu = calc_mu(key, lambda x: np.sqrt(1/scale)*erf(x), samples=10**8)
155 print('mu =', mu)
156 def create_network(key, L):
157   layers = []
```

```
158   for i in range(L):
159     layers.append(stax.Dense(512, W_std=np.sqrt(1/scale), b_std=0.0))
160     layers.append(stax.Erf())
161
162   init_fn, apply_fn, kernel_fn = stax.serial(
163     *layers,
164     stax.Dense(1, W_std=np.sqrt(1/scale), b_std=0.0)
165   )
166
167   apply_fn = jit(apply_fn)
168   kernel_fn = jit(kernel_fn, static_argnums=(2,))
169
170   return init_fn, apply_fn, kernel_fn
171
172 # if using the spherical dataset, k_thresh is computed from epsilon'
173 def compute_k_threshold(key, L, epsilon_norm, d):
174   col1 = 1 - epsilon_norm
175   col2 = np.sqrt(1 - col1*col1)
176   x0 = [1.0, 0] + [0]*(d-2)
177   x1 = [col1, col2] + [0]*(d-2)
178   x0 = np.array([x0])
179   data = np.array([x1])
180   init_fn, apply_fn, kernel_fn = create_network(key, L)
181   iterate_kernel = kernel_fn(data, x0, 'ntk')
182   norm = float(iterate_kernel.flatten()[0])
183   return np.abs(norm)
184
185
186 # here, we'll run the MNIST experiment
187 # select a k_thresh that corresponds to truncation occurring partway
    ↪  through our example
188 # set k_thresh to largest off-diagonal element (in magnitude)
189 k_thresh_n = 128
190 train = process_data(ds['train'], selection=selection,
    ↪  class_size=k_thresh_n)
191 pred_fn, _, kernel_fn = MyrtleNetworkMNIST(depth)
192 kernel_fn = nt.utils.batch.batch(kernel_fn, batch_size=batch_size)
193 kernel = kernel_fn(train['image'], train['image'], 'ntk')
194 k_thresh = np.amax(np.abs(kernel[np.triu_indices(k_thresh_n, 1)]))
195 print('k_thresh', k_thresh)
196
197 # k_thresh = 0.003114714274078037 4 # for depth = 30, selection = (1, 0),
    ↪  k_thresh_n = 128
```

```
198 # k_thresh = 0.0030612588922113663 # for depth = 30, selection = (3, 4),
    ↪  k_thresh_n = 128
199
200 def kernel_mag(row, k_thresh):
201   return normalize(row, k_thresh)**2
202
203 def normalize(m, k_thresh):
204   return np.clip(m/k_thresh, -1, 1)
205
206 # sparsify a matrix
207 # make nxn matrix have at most O(log(n)) nonzero elements per row/column
208 def sparsify(m, probability_function, k_thresh):
209   global key
210   target_sparsity = int(5*np.log(m.shape[1]))
211
212   out = np2.zeros(m.shape)
213   m2 = np2.array(m)
214
215   for i in range(len(m)):
216     # sample the other indices based on the probability function
217     probs = probability_function(m[i], k_thresh)
218     probs /= np.sum(probs)
219
220     key, p_key = random.split(key, 2)
221     nonzero_indices = random.choice(p_key, np.arange(len(m)),
      ↪  shape=(target_sparsity,),
222                                     replace=False, p=probs)
223     if i not in nonzero_indices:
224       nonzero_indices = np.concatenate((nonzero_indices, np.array([i])))
225     mask = np2.zeros(m.shape[1], dtype=bool)
226     mask[(tuple(nonzero_indices),)] = 1
227     row = m2[i] * mask
228     out[i] += row
229
230   return np.array(out)
231
232 # label given raw NTK output assuming a balanced test dataset
233 def classify(ntk_mean):
234   # find classification threshold given balanced output
235   ntk_mean = ntk_mean[:, 0] - ntk_mean[:, 1]
236   thresh = np.median(ntk_mean)
237   out = (np.sign(ntk_mean - thresh).flatten() + 1) / 2
238   return out
```

```
239
240 for i in range(len(data_sizes)):
241   class_size = data_sizes[i]
242   for t in range(trials[i]):
243     print('trial', t+1)
244     prefix = 'output/mnist_seed' + str(seed) + '_select' + str(selection) +
      ↪  '_depth'
245     prefix = prefix + str(depth) + '_data' + str(class_size) + '_trial' +
      ↪  str(t) + '_'
246
247     print('depth =', depth, 'class size =', class_size)
248     train = process_data(ds['train'], selection=selection,
      ↪  class_size=class_size)
249     test = process_data(ds['test'], selection=selection,
      ↪  class_size=test_size)
250     labels = test['label']
251     np.save(prefix + 'labels.npy', labels)
252
253     pred_fn, _, kernel_fn = MyrtleNetworkMNIST(depth)
254     kernel_fn = nt.utils.batch.batch(kernel_fn, batch_size=batch_size)
255
256     # the sparsified kernel will be asymmetric, so we can't just use the
      ↪  built-in cholesky
257     # hence, we evaluate k_*^T K^{-1} y manually
258     kernel_train = kernel_fn(train['image'], train['image'], 'ntk')
259     kernel_test = kernel_fn(test['image'], train['image'], 'ntk')
260
261     kernel_train_sparse = sparsify(kernel_train, kernel_mag, k_thresh)
262     kernel_train_identity =
      ↪  np.diag(kernel_train)*np.eye(kernel_train.shape[0])
263     conditioning =
      ↪  4*np.amax(np.diag(kernel_train_sparse))*np.eye(len(kernel_train_sparse))
264     kernel_train_sparse = kernel_train_sparse + conditioning
265     kernel_test_normalized = k_thresh*normalize(kernel_test, k_thresh)
266
267     mean = kernel_test @ inv(kernel_train) @ train['label']
268     mean_sparse = kernel_test_normalized @ inv(kernel_train_sparse) @
      ↪  train['label']
269     mean_identity = kernel_test_normalized @ inv(kernel_train_identity) @
      ↪  train['label']
270
271     np.save(prefix + 'exact.npy', mean)
272     np.save(prefix + 'sparse.npy', mean_sparse)
```

```
273     np.save(prefix + 'identity.npy', mean_identity)
274
275     acc = np.sum(classify(mean) == labels[:, 0])/len(labels)
276     acc_sparse = np.sum(classify(mean_sparse) == labels[:, 0])/len(labels)
277     acc_identity = np.sum(classify(mean_identity) == labels[:,
        ↪  0])/len(labels)
278     print('Exact classification accuracy:', acc)
279     print('Sparse classification accuracy:', acc_sparse)
280     print('Identity classification accuracy:', acc_identity)
```

*Appendix C*

# CODE: QUANTUM GENERATIVE ADVERSARIAL NETWORK

In Chapter 2, we proposed the entangling quantum generative adversarial network (EQ-GAN) architecture. Here, we provide an implementation that demonstrates learned noise suppression under an unknown noise model/

```
import tensorflow as tf
import tensorflow_quantum as tfq

import cirq
import sympy
import numpy as np

# visualization tools
import matplotlib.pyplot as plt
from cirq.contrib.svg import SVGCircuit

def generator_circuit(qubits, rotations):
  """Make a GHZ-like state with arbitrary phase using CZ gates.
  For the purposes of the noise experiment, we don't apply Z phase
  corrections, since the point is to match the generator and data
  gate parameters to know that there's high state overlap.

  Args:
    qubits: Python `lst` of `cirq.GridQubit`s
    rotations: Python `lst` indicating the X half rotations, Y half
      rotations and Z half rotations.
  """
  if len(rotations) != 3:
    raise ValueError("Number of needed rotations is 3.")

  u = [cirq.Z(qubits[0])**rotations[0],
    cirq.X(qubits[0])**rotations[1],
    cirq.Z(qubits[0])**rotations[2]]
  for q0, q1 in zip(qubits, qubits[1:]):
    u.extend([cirq.Y(q1)**0.5, cirq.X(q1), cirq.CZ(q0, q1),
          cirq.Y(q1)**0.5, cirq.X(q1)])
  return cirq.Circuit(u)
```

```
33
34  def discriminator_circuit(qubits_a, qubits_b, rotations):
35    """Make a variational swap test circuit with CZ as the two-qubit gate.
36
37    Args:
38      qubits_a: Python `lst` of `cirq.GridQubit`s indicating subsystem A's
39        qubits.
40      qubits_b: Python `lst` of `cirq.GridQubit`s indicating subsystem B's
41        qubits.
42      rotations: Python `lst` of shape [n_qubits, 2] containing Z rotation
43        parameters for the swap test.
44    """
45    if len(rotations) != len(qubits_a) or any(len(x) != 2 for x in
      ↪  rotations):
46      raise ValueError("rotations must be shape [len(qubits_a), 2]")
47
48    if len(qubits_a) != len(qubits_b):
49      raise ValueError("unequal system sizes.")
50
51    u = []
52    for i in range(len(qubits_a)):
53      q0 = qubits_a[i]
54      q1 = qubits_b[i]
55      u.extend([cirq.Y(q1)**0.5, cirq.X(q1), cirq.CZ(q0, q1),
        ↪  cirq.Z(q0)**rotations[i][0],
56                          cirq.Z(q1)**rotations[i][1], cirq.Y(q1)**0.5,
                            ↪  cirq.X(q1)])
57
58    # expanded Hadamard: H = X Y^(1/2)
59    for i, q in enumerate(qubits_a):
60      u.append(cirq.Y(q)**0.5)
61      u.append(cirq.X(q)**1.0)
62
63    return cirq.Circuit(u)
64
65  def swap_readout_op(qubits_a, qubits_b):
66    """Readout operation for variational swap test.
67
68    Computes the bitwise and of matched qubits from qubits_a and qubits_b.
69
70    When the states have perfect overlap the expectation of this op will be
    ↪  -1
71    when these states are orthogonal the expectation of this op will be 1.
```

```
72
73    Args:
74      qubits_a: Python `lst` of `cirq.GridQubit`s. The qubits system A act on
75      qubits_b: Python `lst` of `cirq.GridQubit`s. The qubits system B act on
76    """
77
78    def _countSetBits(n):
79      count = 0
80      while n:
81        count += n & 1
82        n >>= 1
83      return count
84
85    def _one_proj(a):
86      return 0.5 * (1 - cirq.Z(a))
87
88    if len(qubits_a) != len(qubits_b):
89      raise ValueError("unequal system sizes.")
90
91    ret_op = 0
92    for i in range(1 << len(qubits_a)):
93      if _countSetBits(i) % 2 == 0:
94        tmp_op = 1
95        for j, ch in enumerate(bin(i)[2:].zfill(len(qubits_a))):
96          intermediate = _one_proj(qubits_a[j]) * _one_proj(qubits_b[j])
97          if ch == '0':
98            intermediate = 1 - intermediate
99          tmp_op *= intermediate
100       ret_op += tmp_op
101
102   return 1.0 - (ret_op * 2 - 1)
103
104 # add controlled phase and Z phase errors after each CZ gate
105 # CZ phase error is fully random
106 # Z phase error is always the same for a given qubit index
107 class CZNoiseModel(cirq.NoiseModel):
108   def __init__(self, qubits, mean, stdev, seed=0):
109     self.mean = mean
110     self.stdev = stdev
111
112     np.random.seed(seed)
113     single_errors = {}
114     for q in qubits:
```

```
115         single_errors[q] = np.random.normal(self.mean[1], self.stdev[1])
116       self.single_errors = single_errors
117
118     def noisy_operation(self, op):
119       if isinstance(op.gate, cirq.ops.CZPowGate):
120         return [op,
          ↪  cirq.ops.CZPowGate(exponent=np.random.normal(self.mean[0],
          ↪  self.stdev[0]))(*op.qubits), cirq.ops.ZPowGate(exponent =
          ↪  self.single_errors[op.qubits[0]])(op.qubits[0]),
          ↪  cirq.ops.ZPowGate(exponent =
          ↪  self.single_errors[op.qubits[1]])(op.qubits[1])]
121
122       return op
123
124   def get_data_maker():
125     """Get appropriate dataset maker for a given circuit type."""
126     return generator_circuit
127
128   def get_circuit_maker():
129     """Get appropriate circuit maker for a given circuit type."""
130     return generator_circuit
131
132   def num_data_parameters(n_qubits):
133     """Get number of true data circuit parameters for a circuit type."""
134     return num_gen_parameters(n_qubits)
135
136   def num_gen_parameters(n_qubits):
137     """Get number of generator model parameters for a circuit type."""
138     return 3
139
140   def num_disc_parameters(n_qubits):
141     """Get number of discriminator model parameters for a circuit type."""
142     return 2*n_qubits
143
144   def get_rand_state(n_qubits, data_noise):
145     """Get number of data preparation circuit parameters for a circuit
      ↪  type."""
146     return np.random.uniform(-data_noise, data_noise,
147                  num_data_parameters(n_qubits))
148
149   def generate_data(data_qubits, generator_qubits, target_quantum_data,
150             data_noise, noise_model, n_points):
151     """Generate n_points data on data_qubits with generator_qubits linked for
```

```
152     later copying."""
153     data_maker = get_data_maker()
154
155     target_circuits = []
156     target_real_data_circuit = []
157
158     rand_states = []
159     for i in range(n_points):
160       rand_states.append(get_rand_state(len(data_qubits), data_noise))
161     for rand_state in rand_states:
162       rand_circuit = data_maker(data_qubits, rand_state +
        ↪  target_quantum_data)
163       rand_circuit_true_data_on_generator_qubit = data_maker(
164         generator_qubits, rand_state + target_quantum_data)
165
166       c_data = rand_circuit.with_noise(noise_model)
167       c_gen =
        ↪  rand_circuit_true_data_on_generator_qubit.with_noise(noise_model)
168
169       target_circuits.append(c_data)
170       target_real_data_circuit.append(c_gen)
171     target_circuits = tfq.convert_to_tensor(target_circuits)
172     target_real_data_circuit =
      ↪  tfq.convert_to_tensor(target_real_data_circuit)
173
174     return target_circuits, target_real_data_circuit
175
176   class SharedVar(tf.keras.layers.Layer):
177     """A custom tf.keras.layers.Layer used for sharing variables."""
178     def __init__(self, symbol_names, operators, init_vals, backend,
179           use_sampled):
180       """Custom keras layer used to share tf.Variables between several
181       tfq.layers.Expectation."""
182       super(SharedVar, self).__init__()
183       self.init_vals = init_vals
184       self.symbol_names = symbol_names
185       self.operators = operators
186       self.use_sampled = use_sampled
187       self.backend = backend
188
189     def build(self, input_shape):
190       # Build a tf.Variable that is the shape of the number of symbols.
191       self.w = self.add_weight(shape=(len(self.symbol_names),),
```

```
192                   initializer=tf.constant_initializer(
193                     self.init_vals))
194
195     def call(self, inputs):
196       # inputs[0] = circuit tensor
197       # inputs[1] = circuit tensor
198       # Their expectations are evaluated with shared variables between them
199       n_datapoints = tf.gather(tf.shape(inputs[0]), 0)
200       values = tf.tile(tf.expand_dims(self.w, 0), [n_datapoints, 1])
201       if not self.use_sampled:
202         return tfq.layers.Expectation(backend=self.backend)(
203           inputs[0],
204           symbol_names=self.symbol_names,
205           operators=self.operators,
206           symbol_values=values), tfq.layers.Expectation(
207             backend=self.backend)(inputs[1],
208                         symbol_names=self.symbol_names,
209                         operators=self.operators,
210                         symbol_values=values)
211       else:
212         return tfq.layers.SampledExpectation(backend=self.backend)(
213           inputs[0],
214           symbol_names=self.symbol_names,
215           operators=self.operators,
216           symbol_values=values,
217           repetitions=10000), tfq.layers.SampledExpectation(
218             backend=self.backend)(inputs[1],
219                         symbol_names=self.symbol_names,
220                         operators=self.operators,
221                         symbol_values=values,
222                         repetitions=10000)
223
224   def build_generator(generator_qubits,
225             data_qubits,
226             generator_symbols,
227             lr,
228             generator_initialization,
229             noise_model,
230             backend=None,
231             use_sampled=False,
232             regularization=0.000001,
233             optimizer=None):
234     """Build a generator tf.keras.Model using standard circuits.
```

```
235
236   Args:
237     generator_qubits: Python `lst` of `cirq.GridQubit`s indicating the
238       qubits that the generator should use.
239     data_qubits: Python `lst` of `cirq.GridQubit`s indicating the qubits
240       that the data will arrive on.
241     generator_symbols: Python `lst` of numbers or `sympy.Symbol`s
242       to use in the ansatze used for the generator.
243     lr: Python `float` the learning rate of the model.
244     backend: Python object for the backend type to use when running quantum
245       circuits.
246     generator_initialization: `np.ndarray` of initial values to place
247       inside of the generator symbols in the tensorflow managed
248       variables.
249     noise_model: `cirq.NoiseModel` to apply to circuits.
250     use_sampled: Python `bool` indicating whether or not to use analytical
251       expectation or sample based expectation calculation.
252     regularization: Python `float` added as margin to an orthogonal swap
    ↪  test.
253     optimizer: `tf.keras.optimizers` optimizer for training the circuit.
    ↪  Default
254       is tf.keras.optimizers.Adam.
255   """
256   if optimizer is None:
257     optimizer = tf.keras.optimizers.Adam
258
259   # Input for the circuits that generate the quantum data from the source.
260   signal_input = tf.keras.layers.Input(shape=(), dtype=tf.dtypes.string)
261
262   # Input for the swaptest circuits. These will have the variables from the
263   # discriminator resolved into them.
264   swap_test_input = tf.keras.layers.Input(shape=(), dtype=tf.dtypes.string)
265
266   data_and_generated = tfq.layers.AddCircuit()(signal_input,
267                           append=generator_circuit(
268                             generator_qubits,
269                             generator_symbols).
270                             with_noise(noise_model))
271
272   # Append the variational swap test on to the data on data_qubits
273   # and the "generated" data on generator_qubits.
274   full_swaptest = tfq.layers.AddCircuit()(data_and_generated,
275                         append=swap_test_input)
```

```
276
277   expectation_output = None
278   if not use_sampled:
279     expectation_output = tfq.layers.Expectation(backend=backend)(
280       full_swaptest,
281       symbol_names=generator_symbols,
282       operators=swap_readout_op(generator_qubits, data_qubits),
283       initializer=tf.constant_initializer(generator_initialization))
284
285   else:
286     expectation_output = tfq.layers.SampledExpectation(backend=backend)(
287       full_swaptest,
288       symbol_names=generator_symbols,
289       operators=swap_readout_op(generator_qubits, data_qubits),
290       initializer=tf.constant_initializer(generator_initialization),
291       repetitions=10000)
292
293   expectation_output = tf.add(expectation_output,
      ↪  tf.constant(regularization))
294   log_output = tf.math.log(expectation_output)
295
296   # Input is true data on data qubits, and swap_test_input for both qubits.
297   qgan_g_model = tf.keras.Model(inputs=[signal_input, swap_test_input],
298                  outputs=[expectation_output, log_output])
299
300   optimizerg = optimizer(learning_rate=lr)
301   lossg = lambda x, y: tf.reduce_mean(y)
302   qgan_g_model.compile(optimizer=optimizerg, loss=lossg,
      ↪  loss_weights=[0,1])
303
304   return qgan_g_model
305
306 def build_discriminator(generator_qubits,
307             data_qubits,
308             discriminator_symbols,
309             lr,
310             discriminator_initialization,
311             noise_model,
312             backend=None,
313             use_sampled=False,
314             regularization=0.000001,
315             optimizer=None):
316   """Build a discriminator model.
```

```
317
318   Args:
319     generator_qubits: Python `lst` of `cirq.GridQubit`s indicating the
320       qubits that the generator should use.
321     data_qubits: Python `lst` of `cirq.GridQubit`s indicating the qubits
322       that the data will arrive on.
323     discriminator_symbols: Python `lst` of numbers or `sympy.Symbol`s
324       to use in the ansatze used for the discriminator.
325     lr: Python `float` the learning rate of the model.
326     discriminator_initialization: `np.ndarray` of symbols to place
327       inside of the discriminator symbols in the tensorflow managed
328       variables.
329     backend: Python object for the backend type to use when running quantum
330       circuits.
331     use_sampled: Python `bool` indicating whether or not to use analytical
332       expectation or sample based expectation calculation.
333     regularization: Python `float` added as margin to an orthogonal swap
    ↪  test.
334     optimizer: `tf.keras.optimizers` optimizer for training the circuit.
    ↪  Default
335       is tf.keras.optimizers.Adam.
336   """
337   if optimizer is None:
338     optimizer = tf.keras.optimizers.Adam
339
340   # True data on data_qubits.
341   signal_input_d = tf.keras.layers.Input(shape=(), dtype=tf.dtypes.string)
342
343   # Generator data on generator_qubits.
344   load_generator_data_d = tf.keras.layers.Input(shape=(),
345                           dtype=tf.dtypes.string)
346
347   # True data on generator_qubits.
348   load_true_data_d = tf.keras.layers.Input(shape=(),
    ↪  dtype=tf.dtypes.string)
349
350   # Swap circuit with input.
351   swap_test_input_d = tfq.layers.AddCircuit()(
352     signal_input_d,
353     append=discriminator_circuit(data_qubits, generator_qubits,
354                 np.array(discriminator_symbols).reshape(-1, 2)).
355                 with_noise(noise_model))
356
```

```
357
358     # Swap test between the true data and generator.
359     swaptest_d = tfq.layers.AddCircuit()(load_generator_data_d,
360                         append=swap_test_input_d)
361
362     # Swap test between the true data and itself. Useful for how close to the
363     # "true" swap test we are over time as we train.
364     swapontruedata = tfq.layers.AddCircuit()(load_true_data_d,
365                           append=swap_test_input_d)
366
367     tmp = SharedVar(discriminator_symbols,
368             swap_readout_op(generator_qubits, data_qubits),
369             discriminator_initialization, backend, use_sampled)
370     expectation_output_d, expectation_output2 = tmp(
371       [swaptest_d, swapontruedata])
372
373     expectation_output_d = tf.add(expectation_output_d,
      ↪  tf.constant(regularization))
374     log_discrim_dist =
      ↪  tf.math.log(tf.keras.backend.flatten(expectation_output_d))
375     log_true_dist =
      ↪  tf.math.log(tf.keras.backend.flatten(expectation_output2))
376
377
378     final_output = -log_discrim_dist
379
380     qgan_d_model = tf.keras.Model(
381       inputs=[signal_input_d, load_generator_data_d, load_true_data_d],
382       outputs=[expectation_output_d, expectation_output2, final_output])
383
384     optimizerd = optimizer(learning_rate=lr)
385
386     # Difference between "generator vs true data" and "true vs true (given
387     # we many not be doing a perfect swap test yet)"
388     lossd = lambda x, y: -tf.reduce_mean(y)
389     qgan_d_model.compile(optimizer=optimizerd, loss=lossd,
      ↪  loss_weights=[0,0,1])
390
391     return qgan_d_model
392
393 def quantum_data_overlap(qubits, params_a, params_b):
394     """Compute overlap of quantum data circuits with params_a and
      ↪  params_b."""
```

```
395     sim = cirq.Simulator()
396     circuit_maker = get_circuit_maker()
397     data_maker = get_data_maker()
398     circuit_a = circuit_maker(qubits, params_a)
399     circuit_b = data_maker(qubits, params_b)
400     res_a = sim.simulate(circuit_a)
401     res_b = sim.simulate(circuit_b)
402     overlap = np.abs(np.vdot(res_a.final_state_vector,
        ↪  res_b.final_state_vector))
403     return overlap
404
405 def run_experiment(d_learn, g_learn, d_epoch, g_epoch, batchsize,
    ↪  n_episodes,
406             n_qubits, target_quantum_data, use_perfect_swap,
407             gate_error_mean, gate_error_stdev, n_data=1, data_noise=0,
408             use_sampled=False, log_interval=10, backend=None, seed=0):
409     """Run a QGAN experiment.
410
411     Args:
412       d_learn: Python `float` discriminator learning rate.
413       g_learn: Python `float` generator learning rate.
414       d_epoch: Python `int` number of discriminator iterations per episode.
415       g_epoch: Python `int` number of generator iterations per episode.
416       batchsize: Python `int` number of entries to use in a batch.
417       n_episodes: Python `int` number of total QGAN training episodes.
418       n_qubits: Python `int` number of qubits to use for each susbsystem.
419       target_quantum_data: Python object. True target state.
420       use_perfect_swap: `bool` whether or not to train discriminator.
421       gate_error_mean: mean angle error on 2-qubit gates (`None` if no
    ↪  noise).
422       gate_error_stdev: standard deviation of angle error on 2-qubit gates.
423       n_data: Python `int` number of total datapoints to generate.
424       data_noise: Python `float` bounds on noise in real data preparation.
425       use_sampled: Python `bool` whether or not analytical or sampled exp.
426       backend: None or `cirq.SimulatesFinalState` or `cirq.Sampler`.
427       log_interval: Python `int` log every log_interval episodes.
428       seed: seed of run for noise model and training.
429     """
430
431     circuit_maker = get_circuit_maker()
432     generator_initialization = np.zeros(num_gen_parameters(n_qubits))
433     discriminator_initialization = np.array([[0.0, 0.0]] * n_qubits)
434
```

```
435     # Create data and generator qubits
436     data_qubits = [cirq.GridQubit(1, k + 4) for k in range(n_qubits)]
437     generator_qubits = [cirq.GridQubit(2, k + 4) for k in range(n_qubits)]
438     ancilla = cirq.GridQubit(1, 5)  # potentially unused.
439     all_qubits = data_qubits + generator_qubits
440
441     # Noise on single-qubit gates
442     if (gate_error_mean is None) or (gate_error_stdev is None):
443       noise_model = None
444     else:
445       noise_model = CZNoiseModel(all_qubits, gate_error_mean,
        ↪  gate_error_stdev, seed=seed)
446
447     # Generator and Discriminator symbols
448     discriminator_parameters = []
449     generator_parameters = []
450     for j in range(num_disc_parameters(n_qubits)):
451       discriminator_parameters.append(sympy.Symbol('Discrimx{!r}'.format(j)))
452     for j in range(num_gen_parameters(n_qubits)):
453       generator_parameters.append(sympy.Symbol('Genx{!r}'.format(j)))
454     target_circuits, target_real_data_circuit = generate_data(data_qubits,
455       generator_qubits, target_quantum_data, data_noise, noise_model, n_data)
456
457     # Generator and Discriminator models
458     qgan_d_model = build_discriminator(
459       generator_qubits, data_qubits, discriminator_parameters, d_learn,
460       discriminator_initialization, noise_model, backend, use_sampled)
461     qgan_g_model = build_generator(
462       generator_qubits, data_qubits, generator_parameters, g_learn,
463       generator_initialization, noise_model, backend, use_sampled)
464
465     # Tracking info
466     d_loss = []
467     g_loss = []
468     overlap_record = []
469     param_history = []
470
471     repeats = 1
472     if not use_perfect_swap: # introduce adversarial second phase
473       repeats = 2
474       n_episodes = n_episodes // 2
475
476     for r in range(repeats):
```

```
477     if r == 0: # use perfect swap for first half
478       use_perfect_swap = True
479     elif r == 1: # use adversarial learning for second half
480       use_perfect_swap = False
481     # begin training
482     for k in range(1, n_episodes + 1):
483       if k != 0:
484         generator_initialization = qgan_g_model.trainable_variables[
485           0].numpy()
486
487       overlap_record.append(
488         quantum_data_overlap(data_qubits, generator_initialization,
489                     target_quantum_data))
490       param_history.append([qgan_g_model.trainable_variables[0].numpy(),
491                   qgan_d_model.trainable_variables[0].numpy()])
492
493       if not use_perfect_swap:
494         # prepare discriminator network input
495         gen_circuit = circuit_maker(generator_qubits,
          ↪  generator_initialization)
496         gen_circuit = gen_circuit.with_noise(noise_model)
497         load_generator_circuit = tf.tile(
498           tfq.convert_to_tensor(
499             [gen_circuit]),
500           tf.constant([n_data]))
501
502         historyd = qgan_d_model.fit(x=[
503           target_circuits, load_generator_circuit,
            ↪  target_real_data_circuit], y=[ tf.zeros_like(target_circuits,
            ↪  dtype=tf.float32), tf.zeros_like(target_circuits,
            ↪  dtype=tf.float32), tf.zeros_like(target_circuits,
            ↪  dtype=tf.float32)], epochs=d_epoch, batch_size=batchsize,
            ↪  verbose=0)
504
505         d_loss.append(historyd.history['loss'])
506
507         # prepare generator network input
508         discriminator_initialization = qgan_d_model.trainable_variables[
509           0].numpy().reshape((-1, 2))
510
511       # evaluate noisy swap test
512       swap_test_circuit = discriminator_circuit(
513         data_qubits, generator_qubits, discriminator_initialization)
```

```
514
515         swap_test_circuit = swap_test_circuit.with_noise(noise_model)
516         swap_test_circuit =
            ↪  tf.tile(tfq.convert_to_tensor([swap_test_circuit]),
517                    tf.constant([n_data]))
518
519         # record history
520         history = qgan_g_model.fit(x=[target_circuits, swap_test_circuit],
521                         y=[tf.zeros_like(target_circuits,
522
                                ↪  dtype=tf.float32),tf.zeros_like(target_circuits,
523                                dtype=tf.float32)],
524                         epochs=g_epoch,
525                         batch_size=batchsize,
526                         verbose=0)
527
528         g_loss.append(history.history['loss'])
529
530         if k % log_interval == 0:
531           print(f'Step = {k}. Overlap={overlap_record[-1]}')
532           print(f'Step = {k}. g_loss={g_loss[-1]}')
533           if not use_perfect_swap:
534             print(f'Step = {k}. d_loss={d_loss[-1]}')
535           print(f'Step = {k}.
              ↪  gen_params={qgan_g_model.trainable_variables[0].numpy()}')
536           print(f'Step = {k}.
              ↪  discrim_params={qgan_d_model.trainable_variables[0].numpy()}')
537
538           print('-'*50)
539
540     return np.array(g_loss), np.array(d_loss), np.array(overlap_record),
        ↪  np.array(param_history)
541
542   d_epoch = 1
543   g_epoch = 1
544   batchsize = 4
545
546   target_quantum_data = [0.0, 0.5, 0.5]
547
548   n_qubits = 1
549   d_learn = 0.01
550   g_learn = 0.01
551   n_episodes = 80
```

```
552
553 # format (radians): [controlled phase error, single-qubit Z phase error]
554 gate_error_mean = [0.0, 0.06]
555 gate_error_stdev = [0.005, 0.02]
556
557 # we run with a "perfect" swap test that is imperfect due to noise
558 use_perfect_swap = True
559 print('TRAINING PERFECT SWAP TEST')
560 g_loss_perf, d_loss_perf, overlap_perf, params_perf = run_experiment(
561             d_learn, g_learn, d_epoch, g_epoch, batchsize,
562             n_episodes, n_qubits, target_quantum_data,
563             use_perfect_swap, gate_error_mean, gate_error_stdev)
564 print()
565
566 # we run with adversarial training to see noise get suppressed
567 use_perfect_swap = False
568 print('TRAINING ADVERSARIAL SWAP TEST')
569 g_loss_adv, d_loss_adv, overlap_adv, params_adv = run_experiment(
570             d_learn, g_learn, d_epoch, g_epoch, batchsize,
571             n_episodes, n_qubits, target_quantum_data,
572             use_perfect_swap, gate_error_mean, gate_error_stdev)
573
574 def stopping_ind(d_loss, smoothing_period=5):
575   """Get overlap and parameters at minimum generator loss."""
576   # simple moving average
577   flattened_loss = np.array(d_loss).flatten()
578   if smoothing_period > 1:
579     smoothed = np.convolve(flattened_loss, np.ones(smoothing_period),
      ↪  'valid')
580     smoothed /= smoothing_period
581   else:
582     smoothed = flattened_loss
583
584   # find when the discriminator loss is lowest in the second half of
    ↪  training
585   # this corresponds to when the GAN is most fooled by the fake data
586   n_episodes = len(d_loss)*2
587   best_ind = n_episodes//2 + np.argmin(smoothed)
588   best_ind += smoothing_period // 2
589   if best_ind >= n_episodes:
590     best_ind = n_episodes - 1
591   return best_ind
592
```

```
fidelity_perfect_swap = overlap_perf**2
fidelity_adversarial = overlap_adv**2
adv_best_ind = stopping_ind(d_loss_adv, smoothing_period=5)

plt.figure(figsize=(5, 3.9))
plt.plot(fidelity_perfect_swap, 'C1', label='Perfect SWAP')
plt.plot(fidelity_adversarial, 'C2', label='EQ-GAN')
plt.axvline(x=adv_best_ind, c='C2', linestyle='--')
plt.legend(fontsize=12)
plt.xlabel('Iteration', fontsize=14)
plt.ylabel('Fidelity $|\\mathrm{data} | \\mathrm{generated} \\rangle|^2$',
↪  fontsize=14)
plt.tight_layout()
plt.show()
```

*Appendix D*

# CODE: VARIATIONAL QUANTUM RANDOM ACCESS MEMORY

In Chapter 2, we described the use of the entangling quantum generative adversarial network (EQ-GAN) to prepare a quantum random access memory (QRAM) representation of an approximation to the classical dataset in superposition. By loading the approximate dataset from the variational QRAM, a quantum neural network (QNN) is trained more quickly, showing a possible use for the EQ-GAN architecture in preparing shallow circuit approximations of deeper circuits. Here, we provide code to show that a QNN classifier converges more quickly when trained from a superposition over the dataset instead of individual data examples.

```
import tensorflow as tf
import tensorflow_quantum as tfq

import cirq
import sympy
import numpy as np

# visualization tools
import matplotlib.pyplot as plt
from cirq.contrib.svg import SVGCircuit

import collections
import itertools

from skopt import gp_minimize
from skopt.space.space import Real

hardware_backend = False

if hardware_backend:
  project_id = 'google.com:quantum-engine-trail-run'
  engine = cirq.google.Engine(project_id=project_id)
  testsamplerxmon_rainbow = engine.sampler(processor_id=['rainbow'],
  ↪  gate_set=cirq.google.XMON)
```

```
    backend = testsamplerxmon_rainbow
else:
    backend = None

# create 2-peak dataset
def create_data(seed, n, dataset_size=100):
    np.random.seed(seed)
    # sample data from Gaussian
    data0_raw = np.random.normal(2**(n-1), scale=2, size=dataset_size)
    bins = np.arange(2**n + 1).astype(np.float64)
    bins[-1] = np.inf
    counts0, _ = np.histogram(data0_raw, bins=bins)
    data0 = np.clip(np.floor(data0_raw), 0, 2**n - 1)

    data1_raw = np.random.normal(2**(n-2), scale=1, size=dataset_size)
    counts1, _ = np.histogram(data1_raw, bins=bins)
    data1 = np.clip(np.floor(data1_raw), 0, 2**n - 1)

    return data0, data1

# create circuits from dataset (for sampling)
size = 120
n = 4 # number of qubits
data0, data1 = create_data(0, n, dataset_size=size)

bins = np.arange(2**n + 1).astype(np.float64)
bins[-1] = np.inf
probs0, _ = np.histogram(data0, bins=bins)
probs1, _ = np.histogram(data1, bins=bins)

print('Classical dataset probabilities')
plt.figure(figsize=(3.2, 2.8))
plt.scatter(bins[:-1], probs0, label='Class 0')
plt.scatter(bins[:-1], probs1, label='Class 1')
plt.legend()
plt.ylabel('Count')
plt.tight_layout()
plt.savefig('classical_data.pdf')
plt.show()


# get qubits for a rainbow chip
def get_exp_qubits(n, class_type=-1):
```

```
67    # we hard-wire choice of qubits for n = 4 on quantum device
68    if class_type == 0:
69      return [cirq.GridQubit(2, 4), cirq.GridQubit(1, 4), cirq.GridQubit(2,
        ↪ 3),
70                  cirq.GridQubit(2, 5), cirq.GridQubit(3, 4)]
71    elif class_type == 1:
72      return [cirq.GridQubit(1, 4), cirq.GridQubit(2, 4), cirq.GridQubit(2,
        ↪ 3),
73                  cirq.GridQubit(2, 5), cirq.GridQubit(3, 4)]
74    else:
75      return [cirq.GridQubit(3, 4), cirq.GridQubit(1, 4), cirq.GridQubit(2,
        ↪ 3),
76                  cirq.GridQubit(2, 5), cirq.GridQubit(2, 4)]
77
78  # EQ-GAN generator for double exponential peaks
79  def build_qnn(qubits, model_type):
80    n = len(qubits)
81    u = []
82    angles = []
83    if model_type == 0:
84      center = 0
85      j = 0
86      for i in range(n):
87        if i == center:
88          u.extend([cirq.Y(qubits[i])**0.5, cirq.X(qubits[i])])
89        else:
90          theta = sympy.Symbol('t' + str(i))
91          angles.append(theta)
92          u.append(cirq.ry(2*theta).on(qubits[i]))
93          j += 1
94      for i in range(n):
95        if i != center:
96          u.extend([cirq.Y(qubits[i])**0.5, cirq.X(qubits[i]),
97                        cirq.CZ(qubits[center], qubits[i]),
98                        cirq.Y(qubits[i])**0.5, cirq.X(qubits[i])])
99      circuit = cirq.Circuit(u)
100   elif model_type == 1:
101     j = 0
102     center = 1
103     u.append(cirq.I.on(qubits[0]))
104     for i in range(1, n):
105       if i == center:
106         u.extend([cirq.Y(qubits[i])**0.5, cirq.X(qubits[i])])
```

```
107         else:
108           theta = sympy.Symbol('t' + str(i))
109           angles.append(theta)
110           u.append(cirq.ry(2*theta).on(qubits[i]))
111           j += 1
112       for i in range(1, n):
113         if i != center:
114           u.extend([cirq.Y(qubits[i])**0.5, cirq.X(qubits[i]),
115                     cirq.CZ(qubits[center], qubits[i]),
116                     cirq.Y(qubits[i])**0.5, cirq.X(qubits[i])])
117       circuit = cirq.Circuit(u)
118     return circuit, angles
119
120   # do a swap gate with CZ between q0 and q1
121   def compiled_swap(q0, q1):
122     u = []
123     u.extend([cirq.X(q0)**0.5])
124     u.extend([cirq.Z(q1)**-0.5, cirq.X(q1)**0.5, cirq.Z(q1)**0.5])
125     u.append(cirq.CZ(q0, q1))
126     u.extend([cirq.Z(q0)**-1, cirq.X(q0)**0.5, cirq.Z(q0)**1])
127     u.extend([cirq.Z(q1)**-1.5, cirq.X(q1)**0.5, cirq.Z(q1)**1.5])
128     u.append(cirq.CZ(q0, q1))
129     u.extend([cirq.X(q0)**0.5])
130     u.extend([cirq.Z(q1)**-0.5, cirq.X(q1)**0.5, cirq.Z(q1)**0.5])
131     u.append(cirq.CZ(q0, q1))
132     u.extend([cirq.Z(q0)**-0.5])
133     u.extend([cirq.Z(q1)**0.5])
134     return cirq.Circuit(u)
135
136   # get a learned circuit for a given dataset
137   def get_model(n, class_type):
138     # pre-trained weights from EQ-GAN on exactly the same training set
139     # QRAM is trained from 60 examples (half of the size = 120)
140     all_weights = [[1.3459893, 1.0012823, 0.94282967], [4.7395287,
      ↪  0.96802247]]
141
142     qubits = get_exp_qubits(n, class_type)
143     qnn, symbols = build_qnn(qubits[:-1], class_type)
144     resolver = {}
145     for i in range(len(symbols)):
146       resolver[symbols[i]] = all_weights[class_type][i]
147     resolved_qnn = cirq.resolve_parameters(qnn, resolver)
148
```

```
149   all_qubits = get_exp_qubits(n)
150   resolved_qnn += compiled_swap(all_qubits[0], all_qubits[-1])
151   return resolved_qnn
152
153 simulator = cirq.Simulator()
154 qubit_order_0 = [cirq.GridQubit(2, 4), cirq.GridQubit(3, 4),
    ↪  cirq.GridQubit(1, 4),
155                    cirq.GridQubit(2, 3), cirq.GridQubit(2, 5)]
156 result = simulator.simulate(get_model(n, 0),
    ↪  qubit_order=qubit_order_0).final_state_vector
157 probs_class_0 = np.abs(result)**2
158
159 qubit_order_1 = [cirq.GridQubit(2, 4), cirq.GridQubit(1, 4),
    ↪  cirq.GridQubit(3, 4),
160                    cirq.GridQubit(2, 3), cirq.GridQubit(2, 5)]
161 result = simulator.simulate(get_model(n, 1),
    ↪  qubit_order=qubit_order_1).final_state_vector
162 probs_class_1 = np.abs(result)**2
163
164 print('Variational QRAM')
165 plt.figure(figsize=(3.2, 2.8))
166 plt.scatter(np.arange(2**n), probs_class_0[:2**n], label='Class 0')
167 plt.scatter(np.arange(2**n), probs_class_1[:2**n], label='Class 1')
168 plt.ylabel('PDF')
169 plt.legend()
170 plt.tight_layout()
171 plt.savefig('quantum_data.pdf')
172 plt.show()
173
174 def convert_to_circuit(data, n):
175   values = np.ndarray.flatten(data)
176   qubits = get_exp_qubits(n)
177   circuit = cirq.Circuit()
178   for i, value in enumerate(values):
179     circuit.append(cirq.X(qubits[i])**value)
180   return circuit
181
182 # helper function to replace np.unpackbits with a custom bitstring length
183 def unpackbits(x, num_bits):
184   xshape = list(x.shape)
185   x = x.reshape([-1, 1])
186   mask = 2**np.arange(num_bits).reshape([1, num_bits])
```

```
187     return np.flip((x & mask).astype(bool).astype(int).reshape(xshape +
        ↪  [num_bits]), axis=1)
188
189   all_data = np.array([unpackbits(data0.astype(np.int64), n),
190                        unpackbits(data1.astype(np.int64), n)])
191
192   x_circ = [convert_to_circuit(x, n) for x in all_data[0]]
193   x_circ = x_circ + [convert_to_circuit(x, n) for x in all_data[1]]
194   y = np.array([0]*len(all_data[0]) + [1]*len(all_data[1]))
195
196   # define the QNN classifier
197   class ClassifierCircuitLayerBuilder():
198     def __init__(self, data_qubits, readouts):
199       self.data_qubits = data_qubits
200       self.readouts = readouts
201
202     def add_layer(self, circuit, prefix):
203       for j, readout in enumerate(self.readouts):
204         for i, qubit in enumerate(self.data_qubits):
205           symbol = sympy.Symbol(prefix + '-' + str(j) + '-' + str(i))
206           u = []
207           u.extend([cirq.Z(qubit)**-0.5, cirq.X(qubit)**0.5,
          ↪  cirq.Z(qubit)**0.5])
208           u.append(cirq.CZ(qubit, readout))
209           u.extend([cirq.Z(qubit)**-1, cirq.X(qubit)**symbol,
          ↪  cirq.Z(qubit)**1])
210           u.append(cirq.CZ(qubit, readout))
211           u.extend([cirq.Z(qubit)**0.5, cirq.X(qubit)**0.5,
          ↪  cirq.Z(qubit)**-0.5])
212           circuit += cirq.Circuit(u)
213
214   def build_quantum_classifier(n_readouts=1):
215     """Create a QNN model circuit and readout operation to go along with
        ↪  it."""
216     readouts = []
217     qubits = get_exp_qubits(n)
218     for i in range(n_readouts):
219       readouts.append(qubits[-1])
220     circuit = cirq.Circuit()
221     data_qubits = qubits[:-1]
222
223     # prepare the readout qubit
224     circuit.append(cirq.X.on_each(readouts))
```

```
225     circuit.append((cirq.Y**0.5).on_each(readouts))
226     circuit.append(cirq.X.on_each(readouts))
227 
228     builder = ClassifierCircuitLayerBuilder(data_qubits, readouts)
229 
230     # add layer(s)
231     builder.add_layer(circuit, "layer1")
232 
233     # prepare the readout qubit
234     circuit.append((cirq.Y**0.5).on_each(readouts))
235     circuit.append(cirq.X.on_each(readouts))
236 
237     total = cirq.Z(readouts[0])
238     for readout in readouts[1:]:
239       total += cirq.Z(readout)
240     return circuit, total/len(readouts)
241 
242 # create the QNN classifier
243 model_circuit, model_readout = build_quantum_classifier(1)
244 
245 def hinge_accuracy(y_true, y_pred):
246     y_true = tf.squeeze(y_true) > 0.0
247     y_pred = tf.squeeze(y_pred) > 0.0
248     result = tf.cast(y_true == y_pred, tf.float32)
249 
250     return tf.reduce_mean(result)
251 
252 x_train_tfcirc = tfq.convert_to_tensor(x_circ[:size//2])
253 x_test_tfcirc = tfq.convert_to_tensor(x_circ[size//2:])
254 y_train = y[:size//2]
255 y_test = y[size//2:]
256 
257 y_train_hinge = 2.0*y_train-1.0
258 y_test_hinge = 2.0*y_test-1.0
259 
260 epochs = 1
261 
262 # train non-superposition QNN classifier
263 def train_sample_qnn(n, averages=5, learning_rate=0.001):
264     sample_acc_data = []
265     for i in range(averages):
266       model = tf.keras.Sequential([
267           # The input is the data-circuit, encoded as a tf.string
```

```
268         tf.keras.layers.Input(shape=(), dtype=tf.string),
269         # The PQC layer returns the expected value of the readout gate,
            ↪  range [-1,1].
270         tfq.layers.PQC(model_circuit, model_readout, repetitions=10000,
            ↪  backend=backend),
271     ])
272     model.compile(
273         loss=tf.keras.losses.Hinge(),
274         optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
275         metrics=[hinge_accuracy])
276
277     qnn_history_sample = model.fit(
278           x_train_tfcirc, y_train_hinge,
279           batch_size=1,
280           epochs=epochs,
281           verbose=0,
282           validation_data=(x_test_tfcirc, y_test_hinge))
283
284     qnn_results_sample = model.evaluate(x_test_tfcirc, y_test)
285     sample_weights = model.get_weights()[0]
286     sample_acc_data.append(qnn_results_sample[1])
287     print('Trained model', i+1)
288
289   return sample_acc_data
290
291 # train superposition QNN classifier
292 def train_superpos_qnn(n, averages=5, learning_rate=0.001):
293   gen_circuit_class_0 = get_model(n, 0)
294   gen_circuit_class_1 = get_model(n, 1)
295   superposition_acc_data = []
296   for i in range(averages):
297     model = tf.keras.Sequential([
298         # The input is the data-circuit, encoded as a tf.string
299         tf.keras.layers.Input(shape=(), dtype=tf.string),
300         # The PQC layer returns the expected value of the readout gate,
            ↪  range [-1,1].
301         tfq.layers.PQC(model_circuit, model_readout, repetitions=10000,
            ↪  backend=backend),
302     ])
303     model.compile(
304         loss=tf.keras.losses.Hinge(),
305         optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
306         metrics=[hinge_accuracy])
```

```
307
308       x_superposition = tfq.convert_to_tensor([gen_circuit_class_0,
          ↪  gen_circuit_class_1])
309       y_superposition = np.array([-1, 1])
310
311       qnn_history_superposition = model.fit(
312             x_superposition, y_superposition,
313             batch_size=1,
314             epochs=epochs*len(x_train_tfcirc)//2,
315             verbose=0,
316             validation_data=(x_superposition, y_superposition))
317
318       qnn_results_superposition = model.evaluate(x_test_tfcirc, y_test)
319       superposition_weights = model.get_weights()[0]
320       superposition_acc_data.append(qnn_results_superposition[1])
321       print('Trained model', i+1)
322
323     return superposition_acc_data
324
325   tune = False # can tune hyperparameters with Bayesian optimization
326   if tune:
327     averages = 10
328     lr_range = [Real(-4, -1)]
329
330     def opt_helper_superpos(lr):
331       return -np.mean(train_superpos_qnn(averages=averages,
          ↪  learning_rate=10**lr[0]))
332     def opt_helper_sample(lr):
333       return -np.mean(train_sample_qnn(averages=averages,
          ↪  learning_rate=10**lr[0]))
334
335     res_sup = gp_minimize(opt_helper_superpos, lr_range, n_calls=50)
336     res_sam = gp_minimize(opt_helper_sample, lr_range, n_calls=50)
337     print("Superposition: x*=%.2f f(x*)=%.2f" % (res_sup.x[0], res_sup.fun))
338     print("Sample: x*=%.2f f(x*)=%.2f" % (res_sam.x[0], res_sam.fun))
339   else:
340     # pre-optimized parameter tunes
341     lr_tunes = {'superposition': 10**-1.83, 'sample': 10**-3.93}
342     averages = 20
343
344     superpos_qnn_data = train_superpos_qnn(n, averages=averages,
345                                learning_rate=lr_tunes['superposition'])
346     superpos_mean = np.mean(superpos_qnn_data)
```

```
347     superpos_std = np.std(superpos_qnn_data)/np.sqrt(averages)
348     print('QNN superposition accuracy (mean):', superpos_mean)
349     print('QNN superposition accuracy (stdev):', superpos_std)
350
351     sample_qnn_data = train_sample_qnn(n, averages=averages,
352                                learning_rate=lr_tunes['sample'])
353     sample_mean = np.mean(sample_qnn_data)
354     sample_std = np.std(sample_qnn_data)/np.sqrt(averages)
355     print('QNN sample accuracy (mean):', sample_mean)
356     print('QNN sample accuracy (stdev):', sample_std)
```

*Appendix E*

# CODE: SHALLOW CIRCUITS FOR TIME EVOLUTION

In Chapter 3, we described the method of *compressed Trotterization* to learn a shallow circuit representation of a Trotterized time evolution unitary $e^{-iHt}$. The code implementing the approach is provided below.

```
#
# approximate time evolution
#

import pickle

import numpy as np
import scipy
import sympy
import matplotlib.pyplot as plt

import cirq
import openfermion
import openfermioncirq
from openfermioncirq import trotter

import tensorflow as tf
import tensorflow_quantum as tfq

import cirq.contrib.noise_models
import sys

NOISY = False

def generate_LR_y(p, norm):
  n = len(p)//2;
  l = np.sum(p[:n]**2)
  r = np.sum(p[n:]**2)

  if r == 0:
    return 0
  t = 2*np.arcsin(np.sqrt(r / (l+r)))
  return t
```

```
34
35  def generate_LR_z(w, norm):
36    n = len(w)//2;
37    l = w[:n]
38    r = w[n:]
39    t = 0
40    for i in range(n):
41      t += r[i] - l[i]
42    return t/norm
43
44  def encode_state(coeffs, qubits):
45    p = np.abs(coeffs)
46    w = np.angle(coeffs)
47
48    n = len(qubits)
49    m = len(p)
50    norm = np.sum(p**2)
51
52    # CRy rotations
53    for i in range(n):
54      if i == 0:
55        t = generate_LR_y(p, norm)
56        yield cirq.ry(t)(qubits[0])
57      else:
58        divs = 2**i
59
60        # generate all binary strings of needed length
61        for j in range(divs):
62          start = m//divs * j
63          stop = m//divs * (j+1)
64          t = generate_LR_y(p[start:stop], norm)
65
66          if t == 0:
67            continue
68
69          # we need to control on zero, so use bit flips
70          # where mask is false, conjugate by X
71          mask = np.where(np.flip((((j & (1 << np.arange(i)))) >
            ↪  0).astype(int)) == 0)[0]
72          for k in mask:
73            yield cirq.X(qubits[k])
74          yield cirq.ry(t)(qubits[i]).controlled_by(*qubits[0:i])
75          for k in mask:
```

```
76            yield cirq.X(qubits[k])
77
78
79
80  # 1D Hubbard model
81  def get_hubbard_n(scale):
82    return 2*scale*scale
83
84  def hubbard_h(scale):
85    tunneling = 1.0
86    coulomb = 4.0
87    chemical_potential = 1.0
88    magnetic_field = 1.0
89    params = [tunneling, coulomb, chemical_potential, magnetic_field]
90
91    n_qubits = get_hubbard_n(scale)
92    x_dim = scale
93    y_dim = scale
94
95    hubbard_model = openfermion.fermi_hubbard(x_dim, y_dim, params[0],
      ↪  params[1], chemical_potential=params[2], magnetic_field=params[3])
96    diag = openfermion.get_diagonal_coulomb_hamiltonian(hubbard_model)
97    return diag
98
99  def hubbard(qubits, n_steps, scale=3, time=1, order=1, hamiltonian=None):
100   if hamiltonian is None:
101     diag = hubbard_h(scale)
102   else:
103     diag = hamiltonian
104
105   circuit = cirq.Circuit(
106       openfermioncirq.simulate_trotter(qubits, diag, time=time,
      ↪  n_steps=n_steps, order=order, omit_final_swaps=True)
107   )
108   print('length before optimization', len(list(circuit.all_operations())))
109   cirq.google.ConvertToXmonGates().optimize_circuit(circuit)
110   print('length after optimization', len(list(circuit.all_operations())))
111   return circuit, diag
112
113 # prepare random bitstring in computational basis
114 def get_state_preparation_circuit(qubits, selection=None):
115   if selection is None:
```

```
116       selection = np.random.choice(np.arange(len(qubits)),
          ↪  size=np.random.randint(len(qubits)), replace=False)
117     u = []
118     for s in selection:
119       u.append(cirq.X.on(qubits[s]))
120     for q in qubits:
121       u.append(cirq.I.on(q))
122     circuit = cirq.Circuit(u)
123     return circuit, selection
124
125
126   noise = cirq.contrib.noise_models.DepolarizingWithDampedReadoutNoiseModel(
127           depol_prob=0.005,
128           bitflip_prob=0.005,
129           decay_prob=0.005)
130   noisy_simulator = cirq.DensityMatrixSimulator(noise=noise)
131
132   def get_jellium_n(scale):
133     return scale*scale
134
135   def jellium_prep(scale, qubits):
136     # Set parameters of jellium model.
137     wigner_seitz_radius = 5. # Radius per electron in Bohr radii.
138     n_dimensions = 2 # Number of spatial dimensions.
139     grid_length = scale # Number of grid points in each dimension.
140     spinless = True # Whether to include spin degree of freedom or not.
141     n_electrons = 2 # Number of electrons.
142
143     # Figure out length scale based on Wigner-Seitz radius and construct a
        ↪  basis grid.
144     length_scale = openfermion.wigner_seitz_length_scale(
145         wigner_seitz_radius, n_electrons, n_dimensions)
146     grid = openfermion.Grid(n_dimensions, grid_length, length_scale)
147
148     # Initialize the model and print out.
149     fermion_hamiltonian = openfermion.jellium_model(grid, spinless=spinless,
        ↪  plane_wave=False)
150     # print(fermion_hamiltonian)
151
152     # Convert to DiagonalCoulombHamiltonian type.
153     hamiltonian =
        ↪  openfermion.get_diagonal_coulomb_hamiltonian(fermion_hamiltonian)
154
```

```
155     # Obtain the Bogoliubov transformation matrix.
156     quadratic_hamiltonian =
        ↪  openfermion.QuadraticHamiltonian(hamiltonian.one_body)
157     _, transformation_matrix, _ =
        ↪  quadratic_hamiltonian.diagonalizing_bogoliubov_transform()
158
159     # Create a circuit that prepares the mean-field state
160     occupied_orbitals = range(n_electrons)
161     n_qubits = len(qubits)
162     state_preparation_circuit = cirq.Circuit(
163         openfermioncirq.bogoliubov_transform(
164             qubits, transformation_matrix, initial_state=occupied_orbitals))
165
166     # Print circuit.
167     cirq.DropNegligible().optimize_circuit(state_preparation_circuit)
168
        ↪  cirq.google.ConvertToXmonGates().optimize_circuit(state_preparation_circuit)
169
170     initial_circuit = cirq.Circuit([cirq.X.on(qubits[0]), cirq.CX(qubits[0],
        ↪  qubits[1])])
171     # initial_state = sum(2 ** (n_qubits - 1 - i) for i in occupied_orbitals)
172     # px = np.zeros(2**n_qubits, dtype=np.complex64)
173     # px[initial_state] = 1
174     # initial_circuit = cirq.Circuit(encode_state(px, qubits))
175
176     return hamiltonian, initial_circuit + state_preparation_circuit
177
178   def jellium_trotter(qubits, hamiltonian, n_steps, scale=3, time=1,
      ↪  order=1):
179
180     # Construct circuit
181     circuit = cirq.Circuit(
182       openfermioncirq.simulate_trotter(
183           qubits, hamiltonian, time, n_steps, order,
184           algorithm=trotter.LINEAR_SWAP_NETWORK,
185           omit_final_swaps=True),
186       strategy=cirq.InsertStrategy.EARLIEST)
187
188     # Print circuit.
189     cirq.DropNegligible().optimize_circuit(circuit)
190     cirq.google.ConvertToXmonGates().optimize_circuit(circuit)
191
192     return circuit
```

```
193
194 def trotter_state(circuit, prep_circuit, noisy=False):
195   if noisy:
196     simulated_rho = cirq.DensityMatrixSimulator(noise=noise)
197     simulated_rho = simulated_rho.simulate(prep_circuit +
      ↪  circuit).final_density_matrix
198   else:
199     simulated_rho = cirq.DensityMatrixSimulator()
200     simulated_rho = simulated_rho.simulate(prep_circuit +
      ↪  circuit).final_density_matrix
201   return simulated_rho
202
203 def true_time_evolution_fidelity(H, circuit, prep_circuit, time,
    ↪  noisy=False):
204   hamiltonian_sparse = openfermion.get_sparse_operator(H)
205   initial_state = cirq.Simulator().simulate(prep_circuit).final_state
206   exact_state = scipy.sparse.linalg.expm_multiply(-1j * time *
    ↪  hamiltonian_sparse, initial_state)
207   exact_rho = np.outer(exact_state, exact_state.conj())
208   simulated_rho = trotter_state(circuit, prep_circuit, noisy=noisy)
209   return np.real(np.trace(np.matmul(simulated_rho, exact_rho)))
210
211
212 time_inc = 0.5
213 start_time = 2.0
214 end_time = 100.0
215 initial_s = int(np.round(start_time / time_inc)) + 1
216 final_s = initial_s + int(np.round((end_time - start_time) / time_inc))
217 scale = int(sys.argv[2])
218
219 n_qubits = get_jellium_n(scale)
220 qubits_fitting = cirq.GridQubit.rect(1, n_qubits)
221 qubits_fit = qubits_fitting
222 qubits_exact = qubits_fitting
223
224 H, prep_circuit_exact = jellium_prep(scale, qubits_exact)
225 H, prep_circuit_fit = jellium_prep(scale, qubits_fit)
226
227 circuit_inc = jellium_trotter(qubits_exact, H, 1, time=time_inc, order=0)
228 circuit_fit = jellium_trotter(qubits_fit, H, 1, time=start_time, order=0)
229
230 print('check fidelity', true_time_evolution_fidelity(H, circuit_fit,
    ↪  prep_circuit_fit, start_time, noisy=False))
```

```
231
232 initial_state =
    ↪  tf.convert_to_tensor(cirq.Simulator().simulate(prep_circuit_fit).final_state)
233 op = openfermion.get_sparse_operator(H).todense()
234 H_tf = tf.convert_to_tensor(op.astype(np.complex64))
235
236 with open('circuit_fit.pkl', 'wb') as f:
237   pickle.dump(circuit_fit, f)
238 with open('circuit_inc.pkl', 'wb') as f:
239   pickle.dump(circuit_inc, f)
240
241 from typing import overload, Any, Callable, List, Optional, Tuple, Union
242
243 from cirq.study import resolver
244 from cirq import protocols
245 from cirq.study.flatten_expressions import ExpressionMap
246
247 class ParamSymbolifier(resolver.ParamResolver):
248     """A `ParamResolver` that resolves all circuit parameters to unique
        ↪  symbols.
249
250     This is a mutable object that stores new expression to symbol mappings
251     when it is used to resolve parameters with `cirq.resolve_parameters` or
252     `_ParamFlattener.flatten_circuit`.  It is useful for replacing sympy
253     expressions from circuits with single symbols and transforming
    ↪  parameter
254     sweeps to match.
255     """
256
257     def __new__(cls, *args, **kwargs):
258         """Disables the behavior of `ParamResolver.__new__`."""
259         return super().__new__(cls)
260
261     def __init__(
262             self,
263             param_dict: Optional[resolver.ParamResolverOrSimilarType] =
                ↪  None,
264             *,  # Force keyword args
265             get_param_name: Callable[[
266                 sympy.Basic,
267             ], str] = None):
268         """Initializes a new _ParamFlattener.
269
```

```
270         Args:
271             param_dict: A default initial mapping from some parameter
    ↪ names,
272                 symbols, or expressions to other symbols or values.  Only
    ↪ sympy
273                 expressions and symbols not specified in `param_dict` will
    ↪ be
274                 flattened.
275             get_param_name: A callback function that returns a new
    ↪ parameter
276                 name for a given sympy expression or symbol.  If this
    ↪ function
277                 returns the same value for two different expressions,
    ↪ `'_#'` is
278                 appended to the name to avoid name collision where `#` is
    ↪ the
279                 number of previous collisions.  By default, returns the
280                 expression string surrounded by angle brackets e.g.
    ↪ `'<x+1>'`.
281         """
282         if hasattr(self, '_taken_symbols'):
283             # Has already been initialized
284             return
285         if isinstance(param_dict, resolver.ParamResolver):
286             params = param_dict.param_dict
287         else:
288             params = param_dict if param_dict else {}
289         symbol_params = {
290             _ensure_not_str(param): _ensure_not_str(val)
291             for param, val in params.items()
292         }
293         super().__init__(symbol_params)
294         if get_param_name is None:
295             get_param_name = self.default_get_param_name
296         self.get_param_name = get_param_name
297         self._taken_symbols = set(self.param_dict.values())
298         self.all_symbols = []
299         self.all_values = []
300 
301     @staticmethod
302     def default_get_param_name(val: sympy.Basic) -> str:
303         if isinstance(val, sympy.Symbol):
304             return val.name
```

```
305         return '<{!s}>'.format(val)
306
307     def _next_symbol(self, val: sympy.Basic) -> sympy.Symbol:
308         v = np.random.randint(1000000)
309         symbol = sympy.Symbol('param_' + str(v))
310         # name = self.get_param_name(val)
311         # symbol = sympy.Symbol(name)
312         # # Ensure the symbol hasn't already been used
313         collision = 0
314         while symbol in self.all_symbols:
315             collision += 1
316             symbol = sympy.Symbol('param_' + str(v + collision))
317         self.all_symbols.append(symbol)
318         return symbol
319
320     def value_of(self, value: Union[sympy.Basic, float, str]
321                  ) -> Union[sympy.Basic, float]:
322         """Resolves a symbol or expression to a new symbol unique to that
        ↪  value.
323
324         - If value is a float, returns it.
325         - If value is a str, treat it as a symbol with that name and
    ↪  continue.
326         - Otherwise return a symbol unique to the given value.  Return
327             `param_dict[value]` if it exists or create a new symbol and add
    ↪  it
328             to `param_dict`.
329
330         Args:
331             value: The sympy.Symbol, sympy expression, name, or float to
    ↪  resolve
332                 to a unique symbol or float.
333
334         Returns:
335             The unique symbol or value of the parameter as resolved by this
336             resolver.
337         """
338         self.all_values.append(value)
339         return self._next_symbol(sympy.Symbol('_'))
340
341         # if isinstance(value, (int, float)):
342         #     return value
343         # if isinstance(value, str):
```

```
344             #     value = sympy.Symbol(value)
345             # out = self.param_dict.get(value, None)
346             # if out is not None:
347             #     return out
348             # # Create a new symbol
349             # symbol = self._next_symbol(value)
350             # self.param_dict[value] = symbol
351             # self._taken_symbols.add(symbol)
352             # return symbol
353
354         # Default object truth, equality, and hash
355         __eq__ = object.__eq__
356         __ne__ = object.__ne__
357         __hash__ = object.__hash__
358
359         def __bool__(self) -> bool:
360             return True
361
362         def __repr__(self) -> str:
363             if self.get_param_name == self.default_get_param_name:
364                 return f'_ParamFlattener({self.param_dict!r})'
365             else:
366                 return (f'_ParamFlattener({self.param_dict!r}, '
367                         f'get_param_name={self.get_param_name!r})')
368
369         def flatten(self, val: Any) -> Any:
370             """Returns a copy of `val` with any symbols or expressions replaced
                ↪  with
371             new symbols.  `val` can be a `Circuit`, `Gate`, `Operation`, or
    ↪  other
372             type.
373
374             This method mutates the `_ParamFlattener` by storing any new
    ↪  mappings
375             from expression to symbol that is uses on val.
376
377             Args:
378                 val: The value to copy def symbolify(val: Any) -> Tuple[Any,
    ↪  'ExpressionMap']:
379         flattener = ParamSymbolifier()
380         val_flat = flattener.flatten(val)
381         # expr_map = ExpressionMap(flattener.param_dict)
```

```
382         return val_flat, flattener.all_symbols, flattener.all_valueswith
    ↪  substituted parameters.
383             """
384             return protocols.resolve_parameters(val, self)
385
386     def symbolify(val: Any) -> Tuple[Any, 'ExpressionMap']:
387         flattener = ParamSymbolifier()
388         val_flat = flattener.flatten(val)
389         # expr_map = ExpressionMap(flattener.param_dict)
390         return val_flat, flattener.all_symbols, flattener.all_values
391
392     def symbolify(val: Any) -> Tuple[Any, 'ExpressionMap']:
393         flattener = ParamSymbolifier()
394         val_flat = flattener.flatten(val)
395         # expr_map = ExpressionMap(flattener.param_dict)
396         return val_flat, flattener.all_symbols, flattener.all_values
397
398     parameterized_circuit, symbols, default_values = symbolify(circuit_fit)
399     print('DEFAULT VALUES', len(default_values))
400     parameterized_circuit_inc, symbols_inc, default_values_inc =
    ↪  symbolify(circuit_inc)
401
402     class TimeEvolver:
403       def __init__(self, qubits, prep_circuit, trotter_circuit, symbols=None,
      ↪  default_values=None):
404         self.qubits = qubits
405         self.circuit = prep_circuit + trotter_circuit
406         self.n = len(qubits)
407         self.symbols = symbols
408         self.default_values = default_values
409
410       def get_state(self, params=None):
411         if params is None:
412           return self.circuit
413
414         my_params = params.astype(np.float64)
415         resolver = {}
416         for t in range(len(self.symbols)):
417           resolver[self.symbols[t]] = my_params[t]
418         return cirq.resolve_parameters(self.circuit, resolver)
419
420     approx = TimeEvolver(qubits_fit, prep_circuit_fit, parameterized_circuit,
    ↪  symbols=symbols, default_values=default_values)
```

```
421
422 inc = TimeEvolver(qubits_exact, prep_circuit_exact,
    ↪  parameterized_circuit_inc, symbols=symbols_inc,
    ↪  default_values=default_values_inc)
423
424
425 from tensorflow_quantum.python.layers import Expectation
426 from tensorflow.keras import Input, Model
427 from tensorflow.keras.optimizers import Adam
428 import tensorflow.keras.backend as K
429
430
431 class haltCallback(tf.keras.callbacks.Callback):
432   def on_epoch_end(self, epoch, logs={}):
433     threshold = 1e-10
434     if(logs.get('loss') <= threshold):
435       print('Reached ' + str(threshold) + ' loss value so cancelling
        ↪  training!')
436       self.model.stop_training = True
437
438 trainingStopCallback = haltCallback()
439 earlyStoppingCallback = tf.keras.callbacks.EarlyStopping(monitor='loss',
    ↪  patience=50, restore_best_weights=True)
440
441
442 # radius goes from 0 to 1
443 def random_rotations(qubits, radius):
444   # do random single-qubit rotations on qubits
445   u = []
446   for q in qubits:
447     u.append(cirq.rx(2*(np.random.rand()-0.5)*radius*np.pi).on(q))
448     u.append(cirq.rz(2*(np.random.rand()-0.5)*radius*np.pi).on(q))
449   return cirq.Circuit(u)
450
451 # make training set with neighbors in prep circuit too
452 approx_neighbors = []
453 inc_neighbors = []
454 radius = float(sys.argv[1])
455 train_set_size = 20 # >= 1
456 for i in range(train_set_size - 1):
457   rand = random_rotations(qubits_fit, radius)
458   approx_neighbors.append(TimeEvolver(qubits_fit, rand + prep_circuit_fit,
```

```
459                 parameterized_circuit, symbols=symbols,
                    ↪  default_values=default_values))
460   inc_neighbors.append(TimeEvolver(qubits_exact, rand + prep_circuit_exact,
461     parameterized_circuit_inc, symbols=symbols_inc,
        ↪  default_values=default_values_inc))
462
463 op = []
464 for q in qubits_fit:
465  op.append(1 - cirq.Z(q))
466
467
468 circuit_input = Input(shape=(), dtype=tf.string)
469 output = Expectation()(
470           circuit_input,
471           symbol_names=approx.symbols,
472           operators=op)
473 # output = tf.math.reduce_sum(output, axis=-1, keepdims=True)
474 print(tf.shape(output))
475
476 # Model compile
477 model = Model(inputs=circuit_input, outputs=output)
478 adam = Adam(learning_rate=0.0005)
479 model.compile(optimizer=adam, loss='mse')
480 model.set_weights(np.array([approx.default_values]))
481
482 overlaps = []
483 times = []
484 simulator = cirq.Simulator()
485 old_weights = np.array(approx.default_values)
486 prefix = f'jellium{scale}_blurred_noswap_clean_radius{radius}'
487 if NOISY:
488   prefix = f'jellium{scale}_blurred_noswap_noisy_radius{radius}'
489 for s in range(initial_s, final_s):
490   print(s, 'out of', final_s)
491   target_circuit = inc.get_state(old_weights) + circuit_inc
492   total_circuit = approx.circuit + target_circuit**-1
493   all_circuits = [total_circuit]
494
495   for i in range(len(approx_neighbors)):
496     target_neighbor = inc_neighbors[i].get_state(old_weights) + circuit_inc
497     all_circuits.append(approx_neighbors[i].circuit + target_neighbor**-1)
498
499   model_input = tfq.convert_to_tensor(all_circuits)
```

```
500     print('**** checking prediction')
501     check_prediction = model.predict(model_input)
502     print(check_prediction)
503     print(tf.shape(check_prediction))
504     print('**** made prediction')
505     model_output = tf.convert_to_tensor([[0]*len(qubits_fit)]*train_set_size)
506     print(tf.shape(model_output))
507 
508     history = model.fit(x=model_input, y=model_output, batch_size=1,
        ↪  epochs=300, verbose=1,
509                                     callbacks=[trainingStopCallback,
                                        ↪  earlyStoppingCallback])
510 
511     old_weights = model.get_weights()[0].astype(np.float64)
512     base_circuit = approx.get_state(old_weights)
513     fit_state = simulator.simulate(base_circuit).final_state
514     time_evolver = tf.linalg.expm(-1j * s*time_inc * H_tf)
515     exact_state = tf.linalg.matvec(time_evolver, initial_state).numpy()
516 
517 
518     overlap = np.abs(fit_state.conj().dot(exact_state))**2
519     overlaps.append(overlap)
520     print(overlaps)
521     times.append(s*time_inc)
522     np.save(prefix + 'overlap.npy', [overlaps, times])
523     if s % 10 == 0:
524       model.save_weights(prefix + 'checkpoint')
525 
526 model.save_weights(prefix + 'checkpoint')
527 print(np.array([overlaps, times]))
```

*Appendix F*

# CODE: WORMHOLE CAUSAL PROPAGATOR

In Chapter 3, we analyzed the causal propagator $\mathcal{K}(t_0, t_1)$ of the Dirac SYK, finding a peak suggestive of teleportation through the wormhole. The code computing the causal propagator is provided below, including preparation of a low-rank SYK model.

```
#
# wormhole causal propagator
#

import numpy as np
import openfermion
import cirq
import sympy
from scipy.linalg import expm,eig
from scipy import sparse
from scipy.interpolate import interp1d
from openfermion.ops import MajoranaOperator, FermionOperator,
↪  QubitOperator
from openfermion.transforms import get_fermion_operator
import itertools
import matplotlib.pyplot as plt
import scipy

from openfermion.linalg.linear_qubit_operator import (
  LinearQubitOperator,
  LinearQubitOperatorOptions,
  ParallelLinearQubitOperator,
  apply_operator,
  generate_linear_qubit_operator,
)

def identity(n):
  return np.eye(2**n)
def s_identity(n):
  return sparse.identity(2**n)

def dirac_creation(index, coeff=1):
```

```
32    a_d = FermionOperator(term=(index, 1), coefficient=coeff)
33    return openfermion.jordan_wigner(a_d)
34
35  def dirac_annihilation(index, coeff=1):
36    a = FermionOperator(term=(index, 0), coefficient=coeff)
37    return openfermion.jordan_wigner(a)
38
39  def make_dirac_fermions(Nferm_tot, L_indices=None, R_indices=None):
40    a_left = []
41    a_d_left = []
42    a_right = []
43    a_d_right = []
44    if L_indices is None and R_indices is None:
45      L_indices, R_indices = make_wormhole_dirac_syk_indices(Nferm_tot)
46    for j in range(len(L_indices)):
47      a_d_left.append(dirac_creation(L_indices[j]))
48      a_left.append(dirac_annihilation(L_indices[j]))
49      a_d_right.append(dirac_creation(R_indices[j]))
50      a_right.append(dirac_annihilation(R_indices[j]))
51    return [[a_left, a_d_left], [a_right, a_d_right]]
52
53  def make_wormhole_dirac_syk_indices(Nferm_tot):
54    L_indices = list(range(0, Nferm_tot))
55    R_indices = list(range(Nferm_tot, 2*Nferm_tot))
56    return L_indices, R_indices
57
58  #make the coefficients
59  def make_coeffs(Nferm_tot, J, real=False):
60    variance = J**2/(2*Nferm_tot)**1.5
61    terms4 = []
62    coeffs4 = []
63    for i in range(Nferm_tot):
64      for j in range(Nferm_tot):
65        for k in range(Nferm_tot):
66          for l in range(Nferm_tot):
67            if real:
68              ijkl = np.random.normal(scale=variance) # real Dirac SYK
69            else:
70              ijkl = np.random.normal(scale=np.sqrt(variance/2),
                ↪  size=(2)).view(np.complex128)[0]
71            ind = [i, j, k, l]
72            if ind not in terms4:
73              if i == j:
```

```
74                terms4.append(ind)
75                coeffs4.append(0)
76                if k != l:
77                  terms4.append([i, j, l, k])
78                  coeffs4.append(0)
79              elif k == l:
80                terms4.append(ind)
81                coeffs4.append(0)
82                # i != j
83                terms4.append([j, i, k, l])
84                coeffs4.append(0)
85              elif i == k and j == l:
86                terms4.append(ind)
87                coeffs4.append(np.real(ijkl))
88                terms4.append([j, i, k, l])
89                coeffs4.append(-np.real(ijkl))
90                terms4.append([i, j, l, k])
91                coeffs4.append(-np.real(ijkl))
92                terms4.append([j, i, l, k])
93                coeffs4.append(np.real(ijkl))
94              elif i == l and j == k:
95                terms4.append(ind)
96                coeffs4.append(np.real(ijkl))
97                terms4.append([j, i, k, l])
98                coeffs4.append(-np.real(ijkl))
99                terms4.append([i, j, l, k])
100               coeffs4.append(-np.real(ijkl))
101               terms4.append([j, i, l, k])
102               coeffs4.append(np.real(ijkl))
103             else:
104               terms4.append(ind)
105               coeffs4.append(ijkl)
106               terms4.append([j, i, k, l])
107               coeffs4.append(-ijkl)
108               terms4.append([i, j, l, k])
109               coeffs4.append(-ijkl)
110               terms4.append([k, l, i, j])
111               coeffs4.append(np.conj(ijkl))
112               terms4.append([l, k, i, j])
113               coeffs4.append(-np.conj(ijkl))
114               terms4.append([l, k, j, i])
115               coeffs4.append(np.conj(ijkl))
116               terms4.append([k, l, j, i])
```

```
117                 coeffs4.append(-np.conj(ijkl))
118                 terms4.append([j, i, l, k])
119                 coeffs4.append(ijkl)
120     return terms4, coeffs4
121
122 def dirac_syk_hamiltonians(Nferm_tot, coeffs, q=4, L_indices=None,
    ↪  R_indices=None, low_rank=False):
123     diracs = make_dirac_fermions(Nferm_tot, L_indices=L_indices,
      ↪  R_indices=R_indices)
124     a_l = diracs[0][0]
125     a_d_l = diracs[0][1]
126     a_r = diracs[1][0]
127     a_d_r = diracs[1][1]
128     syks = [QubitOperator(), QubitOperator()]
129     terms = np.array(list(itertools.product(np.arange(Nferm_tot), repeat=q)))
130     for i in range(len(terms)): # for each possible combo of N C q fermions
131       #construct each term of the SYK.
132       syks[0] += coeffs[0][i] * a_d_l[terms[i][0]] * a_d_l[terms[i][1]] *
        ↪  a_l[terms[i][2]] * a_l[terms[i][3]]
133       syks[1] += coeffs[1][i] * a_r[terms[i][3]] * a_r[terms[i][2]] *
        ↪  a_d_r[terms[i][1]] * a_d_r[terms[i][0]]
134     return syks, diracs
135
136 # put the right SYK on the left indices (like with left fermions)
137 def small_right_syk(Nferm_tot, coeffs, q=4, L_indices=None,
    ↪  R_indices=None):
138     diracs = make_dirac_fermions(Nferm_tot, L_indices=L_indices,
      ↪  R_indices=R_indices)
139     a_r = diracs[0][0]
140     a_d_r = diracs[0][1]
141     terms = np.array(list(itertools.product(np.arange(Nferm_tot), repeat=q)))
142     syk = QubitOperator()
143     for i in range(len(terms)):
144       syk += coeffs[1][i] * a_r[terms[i][3]] * a_r[terms[i][2]] *
        ↪  a_d_r[terms[i][1]] * a_d_r[terms[i][0]]
145     return syk
146
147 # creates left and right syk hamiltonians.
148 # doesn't work with q != 4
149 def wormhole_dirac_syk_hamiltonians(Nferm_tot, q=4, L_indices=None,
    ↪  R_indices=None, J=1,
150                                     low_rank=-1, sparse=False, seed=None):
151     if seed is not None:
```

```
152       np.random.seed(seed)
153
154     terms = np.array(list(itertools.product(np.arange(Nferm_tot),
        ↪  repeat=q))).tolist()
155     coeffs = np.zeros((2, len(terms)), dtype=np.complex128)
156
157     if low_rank > -1:
158       coeffs[0] = make_coeffs_low_rank(Nferm_tot, J, rank=low_rank)
159       coeffs[1] = coeffs[0]
160     else:
161       terms4, coeffs4 = make_coeffs(Nferm_tot, J)
162       for i, t in enumerate(terms): # for each possible combo of N C q
          ↪  fermions
163         ind = terms4.index(t)
164         coeffs[0][i] = coeffs4[ind]
165         coeffs[1][i] = coeffs[0][i]
166
167       if sparse: # make half the coefficients 0
168         coeffs[0][np.random.choice(np.arange(len(coeffs[0])),
          ↪  size=len(coeffs[0])//2,
169                                    replace=False)] = 0
170         coeffs[1] = coeffs[0]
171     syks, diracs = dirac_syk_hamiltonians(Nferm_tot, coeffs, q=q,
        ↪  L_indices=L_indices, R_indices=R_indices)
172
173     return syks[0], syks[1], coeffs, diracs
174
175   from scipy.stats import normaltest
176
177   def sqrt_normal(size, series_terms=10):
178     # should work as series_terms -> infinity
179     s = 0
180     for i in range(1, series_terms+1):
181       s += np.random.gamma(0.5, size=size) / (2*i + 1) - np.log(1 + 1/i)/4
182     exponent = np.log(2) / 4 - np.random.gamma(0.5, size=size) - s
183     return np.random.choice([-1, 1], size=size) * np.exp(exponent)
184
185   # generate complex coefficients
186   def complex_coeff(size):
187     return sqrt_normal(size)/np.sqrt(2) + 1j*sqrt_normal(size)/np.sqrt(2)
188
189   # real Dirac SYK with given rank
190   def make_coeffs_low_rank(N, J, rank=1):
```

```
    variance = J**2/(2*N)**1.5
    terms = np.array(list(itertools.product(np.arange(N), repeat=4)))
    i1 = tuple(np.flip(terms[:, :2].transpose(), axis=0))
    i2 = tuple(terms[:, 2:].transpose())

    coeffs = np.zeros(len(terms))
    if rank % 2 == 0:
      lambdas = np.ones(rank) - 2*(np.arange(rank) % 2)
    else:
      lambdas = np.random.normal(size=rank)
    lambdas /= np.sqrt(np.sum(lambdas**2))

    for i in range(rank):
      g = sqrt_normal((N, N))

      # antisymmetrize
      g = np.tril(g) - np.tril(g, -1).T
      np.fill_diagonal(g, 0)

      coeffs += np.conj(g[i1]) * g[i2] * lambdas[i] * np.sqrt(variance)
    return coeffs

def make_exact_tfd(HL, HR, beta, time_reverse=True):
    HL_sparse = openfermion.get_sparse_operator(HL).toarray()
    HL_sparse = np.kron(HL_sparse,identity(1))
    HR_sparse = openfermion.get_sparse_operator(HR).toarray()
    H_sparse = np.add(HL_sparse, HR_sparse)
    N = int(np.log2(np.shape(H_sparse[0])))
    expH_diag = expm(-beta*H_sparse/4)
    if time_reverse:
      tfd = time_reversal(expH_diag@make_bell_pair(N))
    else:
      tfd = expH_diag@make_bell_pair(N)
    Z = np.sqrt(np.vdot(tfd,tfd))
    return tfd/Z

def make_bell_pair(N):
    #print("N tfd: ",N)
    zero = np.array([1,0])
    one = np.array([0,1])
    bell_pair = (np.kron(zero,zero)+np.kron(one,one))/np.sqrt(2)
    epr = bell_pair
    if N == 2:
```

```
234       return epr
235     for i in range(int(N/2)-1):
236       epr = np.kron(bell_pair,epr)
237     return(epr)
238
239   def time_reversal(psi,right=True):
240     N = int(np.log2(np.shape(psi)[0]))
241     m = time_reversal_op(N,right=right)
242     return m @ np.conjugate(psi)
243
244   def time_reversal_op(N,right=True):
245     Y = np.array([[0,-1j],[1j,0]])
246     mr = np.kron(identity(1),-1j*Y)
247     ml = np.kron(-1j*Y,identity(1))
248     if right:
249       m = mr
250     else:
251       m = ml
252     if N>2:
253       for q in range(int(N/2)-1):
254         if right:
255           m = np.kron(m,mr)
256         else:
257           m = np.kron(m,ml)
258     return m
259
260   def is_hermitian(a, rtol=1e-05, atol=1e-08):
261     return np.allclose(a, np.conj(a.T), rtol=rtol, atol=atol)
262
263   def dirac_interaction(diracs, partial_interaction=False):
264     a_l = diracs[0][0]
265     a_d_l = diracs[0][1]
266     a_r = diracs[1][0]
267     a_d_r = diracs[1][1]
268
269     N = len(a_l)
270     H_int = QubitOperator()
271     indices = range(N)
272     if partial_interaction:
273       indices = range(1, N)
274     for k in indices:
275       H_int += 1j*(a_d_l[k]*a_r[k] + a_l[k]*a_d_r[k])
276
```

```
277   return H_int / N
278
279 def generate_Z_string(begin, end):
280   """Returns a product of Z operators at qubit [begin, end]."""
281   operator = QubitOperator('')
282   for i in range(begin, end+1):
283     operator = QubitOperator(((i, 'Z'),), 1) * operator
284   return operator
285
286 # create the |I> state by taking the ground of the interaction
287 def find_I_dirac_ground(N):
288   diracs = make_dirac_fermions(N)
289   H_int = dirac_interaction(diracs)
290   V = openfermion.get_sparse_operator(H_int, 2*N).toarray()
291   eigs, vecs = np.linalg.eig(V)
292   vecs = np.transpose(vecs)
293   state = vecs[np.argmin(np.real(eigs))]
294   return state / np.sqrt(np.conj(state)@state)
295
296 # hard-code the ground state preparation when the first half of qubits are
    ↪  L and second half are R
297 def find_I_dirac(N):
298   n_qubits = 2*N
299   vec = np.zeros(2**n_qubits)
300   vec[0] = 1
301   for k in range(N):
302     Iop = generate_Z_string(0,k-1)*QubitOperator((k, 'X'), -1j)
303     Iop += generate_Z_string(0,N+k-1)*QubitOperator((N+k, 'X'), 1)
304     Iop = generate_linear_qubit_operator(Iop, n_qubits,
      ↪  options=LinearQubitOperatorOptions(2))
305     vec = Iop*vec
306   Istate = vec/np.sqrt(2**N)
307   return (-1)**(N // 4 + 1) * Istate
308
309 from scipy.linalg import expm
310 from scipy.sparse import csc_matrix
311
312 # Convert openfermion QubitOperator X to a 2**N by 2**N scipy sparse matrix
313 def make_sparse(N, X):
314   Xsparse = openfermion.get_sparse_operator(X)
315   dl = int(np.log2(Xsparse.shape[0]))
316   if dl < N:
317     Xsparse = sparse.kron(Xsparse, s_identity(N-dl))
```

```
318   return Xsparse
319
320 # convert openfermion QubitOperator X to a 2**N by 2**N scipy sparse matrix
    ↪  exp(cX)
321 def compute_expm(N, X, c=1):
322   return expm(c*csc_matrix(make_sparse(N, X)))
323
324 # make TFD with memory-efficient use of only H_L
325 def make_tfd(N, H_L, beta):
326   Istate = find_I_dirac(N)
327   upper_block = compute_expm(N, H_L, -beta/2)
328   tfd = sparse.kron(upper_block, s_identity(N))*Istate
329   Z = np.vdot(tfd, tfd)
330   tfd /= np.sqrt(Z)
331   return tfd
332
333 def compute_exact_exponential_mat(H, x):
334   return expm(x*openfermion.get_sparse_operator(H).toarray())
335
336 def compute_series_exponential_mat(N, H, x, return_list=False,
    ↪  max_power=10,
337                                    min_abs_coefficient=1e-11, tolerance =
                                       ↪  0.0001):
338   # Compute Exp(x*H) as a matrix power series
339   # N is the number of sites
340   # H is the Hamiltonian expressed as a QubitOperator
341   # x can be real e.g. x=-beta, or complex e.g. x=-i*t
342   # max_power is the maximum power allowed in the power series expansion
343   # min_abs_coefficient sets to zero any smaller QubitOperator coefficients
344   # tolerance determines when you have enough terms in the power series
345
346   # compute the highest power of 2 less than |x|
347   kmax = 0
348   for k in range(1, 500):
349     if 2**k > abs(x):
350       kmax = k - 1
351       break
352
353   # first we will compute Exp(x*H/2^kmax)
354   # then we will square the result kmax times to get Exp(x*H) =
    ↪  (Exp(x*H/2^kmax))^(2^kmax)
355   x = x/2**kmax
356
```

```
357   Hterms = H.terms
358   expH_list = []
359
360   # the power series expansion of the matrix exponential starts with the
      ↪  identity
361   ExpxH = QubitOperator((),1)
362   expH_list.append(QubitOperator((),1))
363
364   # add x*H
365   Hp = x*H
366   ExpxH += Hp
367   expH_list.append(Hp)
368   Hp_prev = Hp
369
370   terms = ExpxH.terms
371   coeffs = [c[1] for c in list(terms.items())]
372   for p in range(2, max_power+1):
373     Hp_new = Hp_prev*Hp/p
374     Hp_new.compress(min_abs_coefficient ) # Eliminates terms with small
      ↪  coefficients
375     operators = list(Hp_new.get_operators())
376     if operators == []:
377       continue
378     Hp_prev = Hp_new
379     ExpxH += Hp_new
380     expH_list.append(Hp_new)
381     Hp_prev = Hp_new
382     terms = ExpxH.terms
383     coeffs = [c[1] for c in list(terms.items())]
384     terms_new = Hp_new.terms
385     coeffs_new = [c[1] for c in list(terms_new.items())]
386     if p == 2 or len(coeffs) != len(coeffs_minusone):
387       coeffs_minusone = np.full(len(coeffs),10)
388     ratio = np.divide(coeffs,coeffs_minusone)
389     if np.abs(ratio).max() < 1 + tolerance and np.abs(ratio).min() > 1 -
      ↪  tolerance:
390       break
391     coeffs_minusone = coeffs
392
393   # square the result kmax times:
394   for k in range(kmax):
395     ExpxH = ExpxH*ExpxH
396
```

```
397     summed_op =    QubitOperator()
398     for qubit_op in expH_list:
399       summed_op += qubit_op
400
401     if return_list:
402       return expH_list
403     else:
404       return openfermion.get_sparse_operator(ExpxH, n_qubits=N).toarray()
405
406   def make_expV(N, diracs, mu, initial_state, partial_interaction=False,
    ↪  max_power=10, cores=20):
407     final_state = np.zeros(2**(2*N), dtype=np.complex64)
408     ops = compute_series_exponential_mat(2*N, -1 * mu *
      ↪  dirac_interaction(diracs,
409       partial_interaction=partial_interaction), -1j, max_power=max_power,
        ↪  return_list=True)
410     for op in ops:
411       state = generate_linear_qubit_operator(op, 2*N,
        ↪  options=LinearQubitOperatorOptions(cores))*initial_state
412       final_state += state
413     final_state /= np.sqrt(np.sum(np.abs(final_state)**2))
414     return final_state
415
416   def make_expV_inv(N, diracs, mu, initial_state, partial_interaction=False,
    ↪  max_power=10,
417                       cores=20):
418     return make_expV(N, diracs, -mu, initial_state,
      ↪  partial_interaction=partial_interaction,
419                       max_power=max_power, cores=cores)
420
421   def compute_K(N, mu, beta, t, low_rank=-1, max_power=10, cores=20,
    ↪  seed=None):
422     H_L, H_R, coeffs, diracs = wormhole_dirac_syk_hamiltonians(N,
      ↪  low_rank=low_rank, seed=seed)
423     H_R_small = small_right_syk(N, coeffs)
424
425     # make TFD
426     tfd = make_tfd(N, H_L, beta)
427
428     # make time evolution operators
429     timeL = sparse.kron(compute_expm(N, H_L, -1j*t), s_identity(N))
430     timeLd = timeL.conjugate().transpose()
431     timeR = sparse.kron(s_identity(N), compute_expm(N, H_R_small, 1j*t))
```

```
432     timeRd = timeR.conjugate().transpose()
433
434     # compute K
435     K = 0
436     final_state = np.zeros(2**(2*N), dtype=np.complex64)
437     for j in range(N):
438       print(j+1, 'out of', N)
439       ml = diracs[0][0][j] # a_L
440       ml = openfermion.get_sparse_operator(ml, n_qubits=2*N)
441       mleft = timeLd@ml@timeL
442       print('mleft')
443
444       mr = diracs[1][1][j] # a_R^\dagger
445       mr = openfermion.get_sparse_operator(mr, n_qubits=2*N)
446       mright = timeR@mr@timeRd
447       print('mright')
448
449       # first term in K: a_L U^\dagger a_R^\dagger U
450       state = make_expV(N, diracs, mu, tfd, max_power=max_power, cores=cores)
451       print('expV')
452       state = mright@state
453       state = make_expV_inv(N, diracs, mu, state, max_power=max_power,
          ↪  cores=cores)
454       print('expV inv')
455       state = mleft@state
456       Kterm = np.vdot(tfd, state)
457       K += Kterm
458
459       # second term in K: U^\dagger a_R^\dagger U a_L
460       state = mleft@tfd
461       state = make_expV(N, diracs, mu, state, max_power=max_power,
          ↪  cores=cores)
462       print('expV')
463       state = mright@state
464       state = make_expV_inv(N, diracs, mu, state, max_power=max_power,
          ↪  cores=cores)
465       print('expV inv')
466       Kterm = np.vdot(tfd, state)
467       K += Kterm
468     K /= N
469     return K
470
471   if __name__ == '__main__':
```

```
ks = []
ts = np.linspace(0, 0.48, num=17)
for t in ts:
  print('t =', t)
  k = np.real(compute_K(7, 4, 10, t, max_power=4, cores=32, seed=0))
  ks.append(k)
  np.save('ks.npy', ks)
```