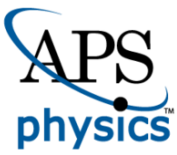

Machine learning for optimal parameter prediction in quantum key distribution

Wenyuan Wang and Hoi-Kwong Lo

Phys. Rev. A **100**, 062334 — Published 27 December 2019

DOI: 10.1103/PhysRevA.100.062334

Machine Learning for Optimal Parameter Prediction in Quantum Key Distribution

Wenyuan Wang[1, *] and Hoi-Kwong Lo[1, †]

[1] *Centre for Quantum Information and Quantum Control (CQIQC), Dept. of Electrical & Computer Engineering and Dept. of Physics, University of Toronto, Toronto, Ontario, M5S 3G4, Canada*

(Dated: November 7, 2019)

For a practical quantum key distribution (QKD) system, parameter optimization - the choice of intensities and probabilities of sending them - is a crucial step in gaining optimal performance, especially when one realistically considers finite communication time. With the increasing interest in the field to implement QKD over free-space on moving platforms, such as drones, handheld systems, and even satellites, one needs to perform parameter optimization with low latency and with very limited computing power. Moreover, with the advent of the Internet of Things (IoT), a highly attractive direction of QKD could be a quantum network with multiple devices and numerous connections, which provides a huge computational challenge for the controller that optimizes parameters for a large-scale network. Traditionally, such an optimization relies on brute-force search or local search algorithms, which are computationally intensive, and will be slow on low-power platforms (which increases latency in the system) or infeasible for even moderately large networks. In this work we present a new method that uses a neural network to directly predict the optimal parameters for QKD systems. We test our machine learning algorithm on hardware devices including a Raspberry Pi 3 single-board computer (similar devices are commonly used on drones) and a mobile phone, both of which have a power consumption of less than 5 watts, and we find a speedup of up to 2-4 orders of magnitude when compared to standard local search algorithms. The predicted parameters are highly accurate and can preserve e.g. over 95-99% of the optimal secure key rate for a given protocol. Moreover, our approach is highly general and can be applied to various kinds of common QKD protocols effectively.

I. BACKGROUND

A. Parameter Optimization in QKD

Quantum key distribution (QKD)[1–4] provides unconditional security in generating a pair of secure key between two parties, Alice and Bob. To address imperfections in realistic sources and detectors, decoy-state QKD [5–7] uses multiple intensities to estimate single-photon contributions, and allows the secure use of Weak Coherent Pulse (WCP) sources, while measurement-device-independent QKD (MDI-QKD) [8] addresses susceptibility of detectors to hacking by eliminating detector side channels and allowing Alice and Bob to send signals to an untrusted third party, Charles, who performs the measurement. A recently proposed protocol, Twin-Field (TF) QKD protocol [9], maintains a similar measurement-device-independence, but can significantly extend the maximum distance and overcome the maximum key rate versus distance tradeoff for repeaterless QKD [10, 11], and it has generated much interest in the community [12–16].

In reality, a QKD experiment always has a limited transmission time, therefore the total number of signals is finite. This means that, when estimating the single-photon contributions with decoy-state analysis, one would need to take into consideration the statistical fluctuations of the observables: the Gain and Quantum Bit Error Rate (QBER). This is called the finite-key analysis of QKD. When considering the above finite-size effects, the choice of intensities and probabilities of sending these intensities is crucial to getting the optimal rate. Therefore, we would need to perform an optimization for the search of parameters.

Note that in this paper, by "parameter optimization", we mainly discuss the optimization of the intensities of laser signals and the probabilities of choosing each intensity setting, specifically in the finite-size scenario (similar to the model outlined in Ref. [17] for MDI-QKD). There is also previous literature [18–20] discussing e.g. the optimization of the *number* of decoy states, but the subject of study is different here. Also, some of the above literature [18] discusses optimization of intensities in the asymptotic (infinite-data) limit, but here in the finite-size case that we study, the parameter space is much larger, making the problem much more computationally challenging. Additionally, in this paper we discuss a broader picture where our method can be applied to various kinds of QKD protocols (and potentially other optimization problems outside QKD or even in classical systems).

There have been various studies on BB84 [21, 22], MDI-QKD [17, 23, 24], and TF-QKD [25] under finite-size effects. Here in the paper, we will employ Ref.[22]'s method for BB84 under finite size effects, and use a standard error analysis for MDI-QKD [23, 26] and asymmetric TF-QKD [27]. Note that, our method is not really dependent on the security analysis model - in fact it is not really dependent on any specific protocol at all - and in principle Chernoff bound can be applied too, but for simplicity in this paper we only use a simple Gaussian assumption for the probability distribution of observables.

Traditionally, the optimization of parameters is implemented as either a brute-force global search for smaller number of parameters, or a local search for larger number of parameters. For instance, in several papers studying

* wenyuan.wang@mail.utoronto.ca
† hklo@comm.utoronto.ca

FIG. 1. Left: Raspberry Pi 3 single-board computer equipped with an Intel Movidius neural compute stick. Right: A smartphone (iPhone XR) running parameter prediction for a QKD protocol with an on-device neural network. In the same app one can also choose to run local search on the device (and compare its running time to that of neural networks).

MDI-QKD protocols with symmetric [17] and asymmetric channels [26], a local search method called coordinate descent algorithm is used to find the optimal set of intensity and probabilities.

However, optimization of parameters often requires significant computational power. This means that, a QKD system either has to to wait for an optimization offline (and suffer from delay), or use sub-optimal or even unoptimized parameters in real-time. Moreover, due to the amount of computing resource required, parameter optimization is usually limited to relatively powerful devices such as a desktop PC.

There is increasing interest in implementing QKD over free-space on mobile platforms, such as drones [28], handheld systems [29], and even satellites [30]. Such devices (e.g. single-board computers and mobile system-on-chips) are usually limited in computational power. As low-latency is important in such free-space applications, fast and accurate parameter optimization based on a changing environment in real time is a difficult task on such low-power platforms.

Moreover, with the advent of the internet of things (IoT), a highly attractive future direction of QKD is a quantum network that connects multiple devices, each of which could be portable and mobile, and numerous connections are present at the same time. This will present a great computational challenge for the controller of a quantum network with many pairs of users (where real-time optimization might simply be infeasible for even a moderate number of connections).

With the development of machine learning technologies based on neural networks in recent years, and with more and more low-power devices implementing on-board acceleration chips for neural networks, here we present a new method of using neural networks to help predict optimal parameters efficiently on low-power devices. We test our machine learning algorithm on real-life devices such as a single-board computer and a smart phone (see Fig. 1), and find that with our method they can easily perform parameter optimization in milliseconds, within a power consumption of less than 5 watts. We list some time benchmarking results in Table I. Such a method makes it possible to support *real-time* parameter optimization for free-space QKD systems, or large-scale QKD networks with thousands of connections.

B. Neural Network

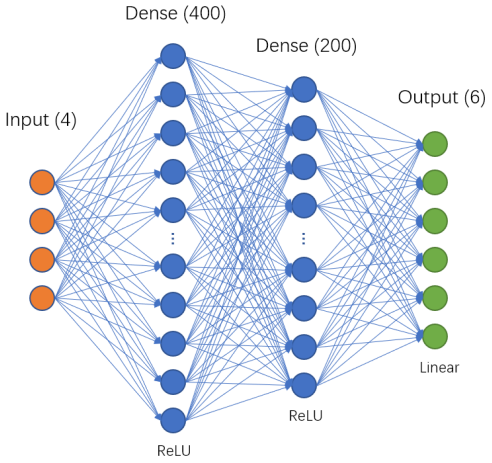

FIG. 2. An example of a neural network (in fact, here it is an illustration of the neural network used in our work). It has an input layer and an output layer of 4 and 6 neurons, respectively, and has two fully-connected "hidden" layers with 400 and 200 neurons with rectified linear unit (ReLU) function as activation. The cost function (not shown here) is mean squared error.

In this subsection we present a very brief introduction to machine learning with neural networks.

Neural networks are multiple-layered structures built from "neurons", which simulate the behavior of biological neurons in brains. Each neuron takes a linear combination of inputs x_i, with weight w_i and offset b, and calculates the activation. For instance:

$$\sigma(\sum w_i x_i + b) = \frac{1}{1+e^{-(\sum w_i x_i + b)}} \qquad (1)$$

where the example activation function is a commonly used sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, but it can have other forms, such as a rectified linear unit (ReLU) [33] function $max(0, \sum w_i x_i + b)$, a step function, or even a linear function $y = x$.

Each layer of the neural network consists of many neurons, and after accepting input from the previous layer and calculating the activation, it outputs the signals to the next layer. Overall, the effect of the neural network is to compute an output $\vec{y} = N(\vec{x})$ from the vector $\vec{x}$. A "cost function" (e.g. mean squared error) is defined on the output layer by comparing the network's calculated output $\{\vec{y^i}\} = \{N(\vec{x_0^i})\}$ on a set of input data $\{\vec{x_0^i}\}$, versus the set of desired output $\{\vec{y_0^i}\}$. It uses an algorithm

TABLE I. Time benchmarking between previous local search algorithm and our new algorithm using neural network (NN) for parameter optimization on various devices. Here as an example we consider two protocols: symmetric MDI-QKD [23] and asymmetric TF-QKD [27]. Devices include a desktop PC with an Intel i7-4790k quad-core CPU equipped with an Nvidia Titan Xp GPU, a modern mobile phone Apple iPhone XR with an on-board neural engine, and a low-power single-board computer Raspberry Pi 3 with quad-core CPU, equipped with an Intel Movidius neural compute stick [a]. As can be seen, neural network generally can provide over 2-4 orders of magnitude higher speed than local search depending on the protocol, enabling millisecond-level parameter optimization. Moreover, note that the smartphone and single-board computer provide similar performance with only less than 1/70 the power consumption of a PC, making them ideal for free-space QKD or a quantum internet-of-things with portable devices. More details on the benchmarking are provided in Section IV.

Protocol	Device	NN accelerator	Local search	NN	Power consumption
MDI-QKD	Desktop PC	Titan Xp GPU	0.1s	0.5-1.0ms	∼350w
TF-QKD	Desktop PC	Titan Xp GPU	2s	0.5-1.0ms	∼350w
MDI-QKD	iPhone XR	on-board neural engine	0.2s	∼1ms	<5w
TF-QKD	iPhone XR	on-board neural engine	N/A	∼1ms	<5w
MDI-QKD	Raspberry Pi 3	Intel neural compute stick	3-5s	2-3ms	<5w
TF-QKD	Raspberry Pi 3	Intel neural compute stick	15-16s	3ms	<5w

[a] The CPU on an iPhone XR has dual big cores + four small cores, but here for simplicity we use a single-threaded program for local search, since OpenMP multithreading library is not supported on Apple devices. OpenMP is supported on the PC and on Raspberry Pi 3, so multithreading is used for local search on these devices. Also, TF-QKD requires a linear solver, but all commercial linear solvers cannot be used on iPhone (while open-source solvers in principle can be compiled for iPhone, in practice the porting is highly non-trivial and very difficult). Therefore local search for TF-QKD cannot be performed on an iPhone - but note that the neural network can still predict the optimal parameters regardless of software library limitations, which is in fact an additional advantage of using a neural network. For TF-QKD, the PC and Raspberry Pi respectively use the commercial solver Gurobi [31] and the open-source solver Coin-OR [32] (as commercial solvers are not available for Raspberry Pi either).

called "backpropagation"[34] to quickly solve the partial derivatives of the cost function to the internal weights in the network, and adjusts the weights accordingly via an optimizer algorithm such as stochastic gradient descent (SGD) to minimize the cost function and let $\{\vec{y^i}\}$ approach $\{\vec{y_0^i}\}$ as much as possible. Over many iterations, the neural network will be able to learn the behavior of $\{\vec{x_0^i}\} \to \{\vec{y_0^i}\}$, so that people can use it to accept a new incoming data $\vec{x}$, and predict the corresponding $\vec{y}$. The *universal approximation theorem* of neural network [35] states that it is possible to infinitely approximate any given bounded, continuous function on a given defined domain with a neural network with even just a single hidden layer, which suggests that neural networks are highly flexible and robust structures that can be used in a wide range of scenarios where such mappings between two finite input/output vectors exist.

There is an increasing interest in the field in applying machine learning to improve the performance of quantum communication. For instance, there is recent literature that e.g. apply machine learning to continuous-variable (CV) QKD to improve the noise-filtering [36] and the prediction/compensation of intensity evolution of light over time [37], respectively.

In this work, we apply machine learning to predict the optimal intensity and probability parameters for QKD (based on given experimental parameters, such as channel loss, detector efficiency, misalignment, dark count rate, and data size), and show that with a simple fully-connected neural network with two layers, we can very accurately and efficiently predict parameters (that can achieve e.g. 95-99%, or even 99.9% the key rate depending on the protocol).

Our work demonstrates the feasibility of deploying neural networks on actual low-power devices, to make them perform fast QKD parameter optimization in real time, with up to 2-4 orders of magnitudes higher speed. This enables potential new applications in free-space or portable QKD devices, such as on a satellite[30], drone [28], or handheld [29] QKD system, where power consumption of devices is a crucial factor and computational power is severely limited, and traditional CPU-intensive optimization approaches based on local or global search are infeasible.

Additionally, we point out that with the higher optimization speed, we can also enable applications in a large-scale quantum internet-of-things (IoT) where many small devices can be interconnected (thus generating a large number of connections), and now with neural networks, even low-power devices such as a mobile phone will be able to optimize the parameters for hundreds of users in real-time in a quantum network.

Our paper is organized as follows: In Section II we will describe how we can formulate parameter optimization as a function that can be approximated by a neural network. We then describe the structure of the neural network we use, and how we train it such that it learns to predict optimal parameters. In Section III we test our neural network approach with four example protocols, and show that neural networks can accurately predict parameters, which can be used to obtain near-optimal secure key rate for the protocols. In Section IV we describe two important use cases for our method and benchmark them: enabling real-time parameter optimization on low-power and low-latency portable devices, and paving the road for large-scale quantum networks. We conclude our paper in Section V.

II. METHODS

In this section we describe the process of training and validating a neural network for parameter optimization. As mentioned in Sec. I, the *universal approximation theorem* implies that the approach is not limited to any specific protocol. Here for simplicity, in this section when describing the methods we will first use a simple symmetric "4-intensity MDI-QKD protocol" [23] as an example protocol. Later in the next section when presenting the numerical results, we also include three other protocols, the asymmetric "7-intensity" MDI-QKD protocol[26], the BB84 protocol (under finite-size effects) [22], and the asymmetric TF-QKD protocol [27] to show that the method applies to them effectively too.

A. Optimal Parameters as a Function

Let us consider the symmetric-channel case for MDI-QKD. Alice and Bob have the same distance to Charles, hence they can choose the same parameters. When taking finite-size effects into consideration, the variables to be optimized will be a set of 6 parameters, $[s, \mu, \nu, P_s, P_\mu, P_\nu]$, where s, μ, ν are the signal and decoy intensities, and P_s, P_μ, P_ν are the probabilities of sending them. Here there are also two other parameters, the vacuum state ω and the vacuum state probability P_ω, but we assume the former is a constant, and the latter satisfies $P_\omega = 1 - P_s - P_\mu - P_\nu$, so neither are included as variables. Note that, since only the signal intensity s in the Z basis is used for key generation, and μ, ν in X basis are used for parameter estimation, P_s is also the basis choice probability. We will unite these 6 parameters into one parameter vector $\vec{p}$.

The calculation of the key rate depends not only on the intensities and the probabilities, but also on the experimental parameters, namely the distance L between Alice and Bob (or equivalently L_{BC}, the distance between the relay Charles and Bob), the detector efficiency η_d, the dark count probability Y_0, the basis misalignment e_d, the error-correction efficiency f_e, and the number of signals N sent by Alice. We will unite these parameters into one vector $\vec{e}$, which we call the "experimental parameters".

Therefore, we see that the QKD key rate can be expressed as

$$Rate = R(\vec{e}, \vec{p}) \tag{2}$$

which is a function of the experimental parameters $\vec{e}$, which cannot be controlled by the users, and the "user parameters" $\vec{p}$ (or just "parameters" for short, in the rest of the paper if not specifically mentioned), which can be adjusted by the users.

However, this only calculates the rate for a given fixed set of parameters and experimental parameters. To calculate the optimal rate, we need to calculate

$$R_{max}(\vec{e}) = max_{\vec{p}\in P} R(\vec{e}, \vec{p}) \tag{3}$$

which is the optimal rate for a given $\vec{e}$. By maximizing R, we also end up with a set of optimal parameters $\vec{p_{opt}}$. Note that $\vec{p_{opt}}$ is a function of $\vec{e}$ only, and the key objective in QKD optimization is to find the optimal set of $\vec{p_{opt}}$ based on the given $\vec{e}$:

$$\vec{p_{opt}}(\vec{e}) = argmax_{\vec{p}\in P} R(\vec{e}, \vec{p}) \tag{4}$$

Up so far, the optimal parameters are usually found by performing local or global searches [17, 26], which evaluate the function $R(\vec{e}, \vec{p})$ many times with different parameters to find the maximum. However, we make the key observation that the functions $R_{max}(\vec{e})$ and $\vec{p_{opt}}(\vec{e})$ are still *single-valued, deterministic functions* (despite that their mathematical forms are defined by *max* and *argmax* and not analytically attainable).

As mentioned in Section I, the universal approximation theorem of neural network states that it is possible to infinitely approximate any given bounded, continuous function on a given defined domain with a neural network (with a few or even a single hidden layer). Therefore, this suggests that it might be possible to use a neural network to fully described the behavior of the aforementioned optimal parameters function $\vec{p_{opt}}(\vec{e})$. Once such a neural network is trained, it can be used to directly find the optimal parameters and the key rate based on any input $\vec{e}$ by evaluating $\vec{p_{opt}}(\vec{e})$ and $R(e, \vec{p_{opt}})$ once each (rather than the traditional approach of evaluating the function $R(\vec{e}, \vec{p})$ many times), hence greatly accelerating the parameter optimization process. As we will later show, this method works for several common types of protocol as long as we can formulate a good analytical form of the key rate function. Nonetheless, this method also relies on the fact that the given protocol has a convex key rate versus parameters function (and has a bounded domain - which in practice is mostly just a "square" domain with acceptable constant upper/lower bound values for each dimension of $\vec{p}$), such that the optimization problem is a convex optimization and a local search works (or that the function is not too highly non-convex such that some simple global search techniques can address the non-convexity).

B. Design and Training of Network

Here we proceed to train a neural network to predict the optimal parameters. We first write a program that randomly samples the input data space to pick a random combination of $\vec{e}$ experimental parameters, and use local search algorithm [17] to calculate their corresponding optimal rate and parameters. The experimental parameter - optimal parameter data sets (for which we generate 10000 sets of data for 40 points from L_{BC} =0-200km, over the course of 6 hours) are then fed into the neural network trainer, to let it learn the characteristics of the function $\vec{p_{opt}}(\vec{e})$. The neural network structure is shown in Fig.2. With 4 input and 6 output elements, and two hidden layers with 200 and 400 ReLU neurons each. We use a mean squared error cost function.

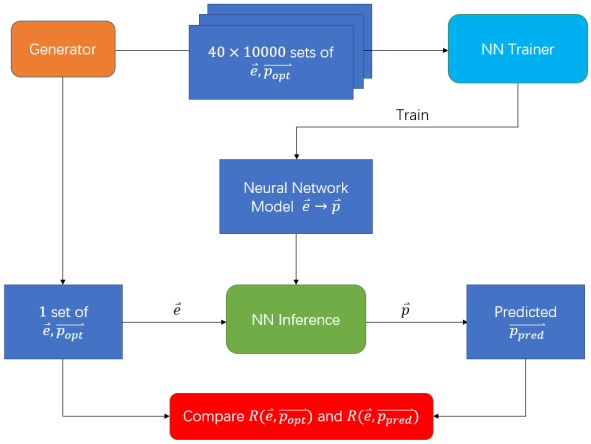

FIG. 3. Data flow of the training and testing of the neural network (NN). The rounded-corner boxes represent programs, and rectangular boxes represent data. The generator program generates many random sets of experimental parameters $\vec{e}$ and calculates the corresponding optimal parameters $\vec{p_{opt}}$. These data are used to train the neural network. After the training is complete, the network can be used to predict optimal parameters based on arbitrary new sets of random experimental data and generate $\vec{p_{pred}}$ (for instance, to plot the results of Fig. 4 and Fig. 5, for each protocol a single random set of data is used as input). Finally, another "validation" program calculates the key rate based on the actual optimal parameters $\vec{p_{opt}}$ found by local search and the predicted $\vec{p_{pred}}$ respectively, and compares their performances.

For input parameters, since η_d is physically no different from the transmittance (e.g. having half the η_d is equivalent to having 3dB more loss in the channel - note that here we will assume that loss from detector efficiency can be controlled by Eve and therefore can be merged into channel loss, and that all detectors have equal η_d), here as an example we fix it to 80% to simplify the network structure (so the input dimension is 4 instead of 5) - when using the network for inference, a different η_d can be simply multiplied onto the channel loss while keeping $\eta_d = 80\%$. We also normalize parameters by setting

$$\begin{aligned} e_1 &= L_{BC}/100 \\ e_2 &= -log_{10}(Y_0) \\ e_3 &= e_d \times 100 \\ e_4 &= log_{10}(N) \end{aligned} \tag{5}$$

to keep them at a similar order of magnitude of 1 (which the neural network is most comfortable with) - what we're doing is a simple scaling of inputs, and this pre-processing doesn't modify the actual data. The output parameters (intensities and probabilities) are within $(0, 1)$ to begin with (we don't consider intensities larger than 1 since these values usually provide poor or zero performance) so they don't need pre-processing. When generating random sets of experimental parameters for training, here as an example we use a range of common values for $e_1 \in [0, 2]$, $e_2 \in [5, 7]$, $e_3 \in [1, 3]$, $e_4 \in [11, 14]$ which correspond to $L_{BC} \in [0, 200]$, $Y_0 \in [10^{-7}, 10^{-5}]$, $e_d \in [0.01, 0.03]$, $N \in [10^{11}, 10^{14}]$. The normalized parameters are sampled uniformly from the range (i.e. some parameters Y_0, N are uniformly sampled in log scale). Also, note that, the range we set here (with commonly encountered values in experiment) is a testing example, but in practice one can train with a wider range for the input values to encompass more possible scenarios for experimental parameters (and reasonably with more sample training data).

We can also easily modify the setup to accommodate for other protocols by adjusting the number of input and output parameters. For the asymmetric MDI-QKD scenario, one can add an additional input parameter, the channel mismatch $x = \eta_A/\eta_B$, where η_A, η_B are the transmittances in Alice's and Bob's channels. We can normalize the mismatch too and make it an additional input variable:

$$e_5 = -log_{10}(x) \tag{6}$$

For the random training data, we sample $e_5 \in [0, 2]$, i.e. channel mismatch $x \in [0.01, 1]$. In this case the output parameter vector $\vec{p}$ would be $[s_A, \mu_A, \nu_A, P_{s_A}, P_{\mu_A}, P_{\nu_A}, s_B, \mu_B, \nu_B, P_{s_B}, P_{\mu_B}, P_{\nu_B}]$.

For BB84 under finite-size effects, the input vector is the same as in symmetric MDI-QKD, while the output parameter vector $\vec{p}$ would be $[\mu, \nu, P_\mu, P_\nu, P_X]$, where vacuum+weak decoy states are used (i.e. intensities are $[\mu, \nu, \omega]$, which correspond respectively to the signal, weak decoy, and vacuum states) and only one basis - for instance the X basis - is used for encoding. Here P_X is the probability of choosing the X basis. Since the parameter space of BB84 is slightly non-convex, when generating the training set, we have modified the local search to start from multiple random starting points (and choose the highest local maximum). This is a simple form of global search, and can mostly overcome the small non-convexity for the key rate versus parameters function for BB84.

For asymmetric TF-QKD, the input vector is the same as in asymmetric MDI-QKD, while the output parameter vector $\vec{p}$ is $[s_A, s_B, \mu, \nu, P_s, P_\mu, P_\nu]$, where, as shown in Ref. [27], to compensate for channel asymmetry, we employ asymmetric signal states s_A, s_B while using the identical vacuum+weak decoy states (and probabilities) for Alice and Bob, while leads to 2+5 output parameters. For the detector efficiency η_d, we assume it is part of the channel loss and merge it as part of L_{BC}, i.e. in the program η_d is set to 100% (and like for MDI-QKD, we assume the two detectors have equal detector efficiency). An additional note for TF-QKD is that it uses a linear program to calculate the key rate $R(\vec{e}, \vec{p})$, which makes it over an order of magnitude slower than e.g. MDI-QKD, which uses analytical functions to solve for the key rate. Due to the large amount of computation involved, we performed the data generation on the Niagara supercomputer (using about 4 nodes × 8 hours, or 1280 core hours as each node has 40 Intel Xeon cores, after which we chose 4500 sets of random data collected from multiple runs),

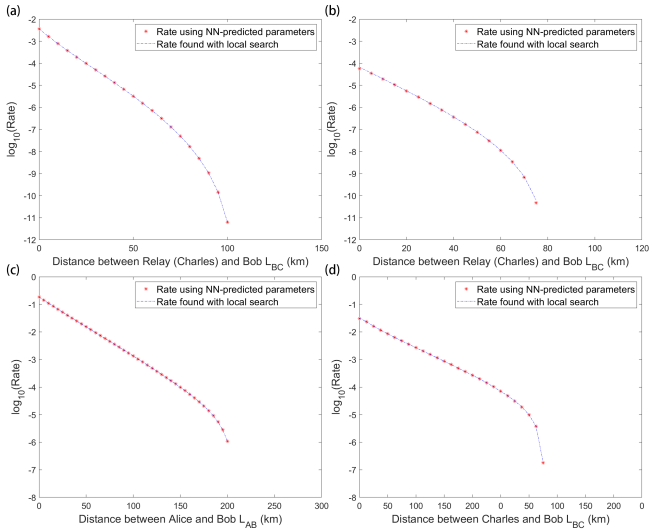

FIG. 4. Comparison of expected key rate using neural network (NN) predicted parameters vs using optimal parameters found by local search for various protocols, using experimental parameters from Table III, at different distances between Alice (or Charles) and Bob. We compare the key rate generated with either sets of parameters (dots with NN-predicted parameters, and lines with local search generated parameters). We tested four protocols: (a) symmetric MDI-QKD (4-intensity protocol) [23], (b) asymmetric MDI-QKD (7-intensity protocol) [26], (c) BB84 protocol [22], and (d) asymmetric TF-QKD protocol [27]. As can be seen, the key rate obtained using predicted parameters is very close to that obtained from using optimal parameters found with local search.

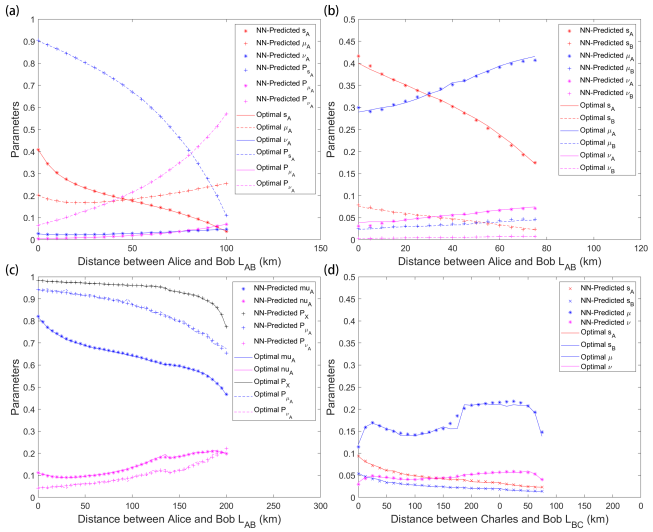

FIG. 5. Comparison of neural network (NN) predicted parameters vs optimal parameters found by local search for various protocols, using experimental parameters from Table III, at different distances between Alice (or Charles) and Bob. We compare neural network (NN) predicted parameters (dots) versus optimal parameters found by local search (lines), for four protocols: (a) symmetric MDI-QKD (4-intensity protocol), (b) asymmetric MDI-QKD (7-intensity protocol), (c) BB84 protocol, and (d) asymmetric TF-QKD protocol. Similar to Fig. 4, as can be seen, the NN-predicted parameters are very close to optimal values found with local search. Note that there are some noise present for the BB84 protocol. This is because the key rate versus parameters function shows some levels of non-convexity, and we combined local search with a randomized approach (similar to global search) that chooses results from multiple random starting points. Therefore there is some level of noise for the probability parameters (which are insensitive to small perturbations), while the neural network is shown to learn the overall shape of the global maximum of the parameters and return a smooth function. Similar applies for TF-QKD, which has small levels of non-convexity due to linear solvers being inherently non-convex, although due to the limitation in training time, we did not apply global search for TF-QKD.

but note that, these computations can be considered offline. That is, once one takes the time to generate the training set and obtain a neural network, end users can simply deploy the neural network to compute all future sets of data online in milliseconds.

We train the neural network using Adam [38] as the optimizer algorithm for 120 epochs (iterations), which takes roughly 40 minutes on an Nvidia Titan Xp GPU.

Note that, here to prevent overfitting, we have crudely employed early-stopping when training the model, by checking the validation set (20% of the data) and stopping the training when the validation set loss no longer decreases (despite that the training set still shows increasingly smaller error), which helps with preventing overfitting. We test in increments of 60 epochs at a time, and choose 120 epochs as the stopping point. We have also tested with adding e.g. Dropout layers, but the results change very little. Likely, since the data size itself (41 samples $\times$ 10000 data sets) is larger than the number of weights in the network (at the order of 200×400), the overfitting problem is not severe here.

III. NUMERICAL RESULTS

After the training is complete, we use the trained network for 4-intensity MDI-QKD protocol to take in three sets of random data, and record the results in Table II. As can be seen, the predicted parameters and the corresponding key rate are very close to the actual optimal values obtained by local search, with the NN-predicted parameters achieving up to 99.99% the optimal key rate.

Here we also fix one random set of experimental parameters as seen in Table III, and scan the neural network over L_{BC} =0-200km. The results are shown in Fig.4(a) and 5(a). As we can see, again the neural network works extremely well at predicting the optimal values for the parameters, and achieves very similar levels of key rate compared to using the traditional local search method.

We also use a similar approach to select a random set of input parameters and compare predicted key rate versus optimal key rate for each of 7-intensity (asymmetric MDI-QKD), BB84, and TF-QKD protocol. The results are included in Fig.4(b-d) and 5(b-d). As can be seen,

TABLE II. Optimal parameters found by local search vs neural network (NN) predicted parameters for symmetric MDI-QKD (4-intensity protocol) using three different random sets of experimental parameters (set b we include here is the same one used for Fig. 4(a) and Fig. 5(a), as listed in Table III), at the same distance L_{BC} of 50km between Charles and Bob. Y_0 is the dark count probability, e_d is the basis misalignment, and N is the number of signals sent by Alice. Here for simplicity, the detector efficiency is fixed at $\eta_d = 80\%$ (since it is equivalent to channel loss). Fibre loss per km is assumed to be $\alpha = 0.2dB/km$, the error-correction efficiency is $f_e = 1.16$, and finite-size security failure probability is $\epsilon = 10^{-7}$. As can be seen, the predicted parameters from our neural network are very close to the optimal parameters found by local search, within an 1% error. Moreover, the key rate is even closer, where the rate calculated with predicted parameters can achieve up to 99.99% (sometimes even higher than) the key rate found by local search for this protocol.

Set	Method	R	L_{BC}	Y_0	e_d	N	s	μ	ν	P_s	P_μ	P_ν
a	Local search	5.5390×10^{-5}	50km	7.50×10^{-6}	0.0115	3.14×10^{13}	0.431	0.170	0.0256	0.862	0.00736	0.0906
a	NN	5.5385×10^{-5}	50km	7.50×10^{-6}	0.0115	3.14×10^{13}	0.431	0.169	0.0257	0.861	0.00729	0.0901
b	Local search	3.2559×10^{-6}	50km	1.75×10^{-7}	0.0287	2.99×10^{12}	0.176	0.183	0.0290	0.670	0.0200	0.216
b	NN	3.2559×10^{-6}	50km	1.75×10^{-7}	0.0287	2.99×10^{12}	0.177	0.184	0.0289	0.670	0.0193	0.216
c	Local search	2.7738×10^{-6}	50km	4.29×10^{-7}	0.0196	2.11×10^{11}	0.209	0.241	0.0442	0.540	0.0339	0.298
c	NN	2.7739×10^{-6}	50km	4.29×10^{-7}	0.0196	2.11×10^{11}	0.210	0.242	0.0441	0.538	0.0338	0.298

TABLE III. Random experimental parameter sets we use for simulation of Fig. 4 and Fig. 5 for the four protocols (symmetric/asymmetric MDI-QKD, BB84, TF-QKD). Y_0 is the dark count probability, e_d is the basis misalignment, and N is the number of signals sent by Alice (and Bob, in MDI-QKD and TF-QKD). Here for simplicity, the detector efficiency is fixed at $\eta_d = 80\%$ for MDI-QKD and BB84, and 100% for TF-QKD (detector loss is included in the channel loss). The channel mismatch x for MDI-QKD and TF-QKD is the ratio of transmittances between Alice's and Bob's channels, η_A/η_B.

Protocol	x	e_d	Y_0	N	η_d
4-intensity MDI	1	0.029	1.7×10^{-7}	3.0×10^{12}	80%
7-intensity MDI	0.10	0.026	2.7×10^{-6}	2.6×10^{13}	80%
BB84	-	0.011	3.6×10^{-6}	3.2×10^{12}	80%
TF-QKD	0.54	0.024	1.8×10^{-6}	1.9×10^{13}	100%

the accuracy of neural network is very high in these cases too, with up to 95-99% the key rate for 7-intensity protocol, up to 99.99% for BB84, and ∼80-95% for asymmetric TF-QKD (the accuracy for TF-QKD is smaller, likely because of either the smaller training set, or the linear solvers used in TF-QKD bringing in inherent non-convexities, which make the data more noisy and more difficult to fit).

IV. APPLICATIONS AND BENCHMARKING

In the previous section we have demonstrated that a neural network (NN) can be trained to very accurately simulate the optimal parameter function $\vec{p_{opt}}(\vec{e})$ and be used to effectively predict the optimal parameters for QKD. The question is, since we already have an efficient coordinate descent (CD) algorithm, what is the potential use for such a NN-prediction approach? Here in this section, we will discuss two important use cases for the neural network.

1. Real-time optimization on low-power devices. While it takes considerable computing power to "train" a neural network (e.g. on a dedicated GPU), using it to predict (commonly called "inference") is computationally much cheaper, and will be much faster than performing a local search, even if the neural network is run on the same CPU. Moreover, in recent years, with the fast development and wide deployment of neural networks, many manufacturers have opted to develop dedicated chips that accelerate NN-inference on mobile low-power systems. Such chips can further improve inference speed with very little required power, and can also offload the computing tasks from the CPU (which is often reserved for more crucial tasks, such as camera signal processing or motor control on drones, or system operations and background applications on mobile phones).

Therefore, it would be more power-efficient (and much faster) to use a neural network running on inference chips, rather than using the computationally intensive local search algorithm with CPU on low-power devices. This can be especially important for free-space QKD scenarios such as drone-based, handheld, or satellite-ground QKD, which not only have very limited power budget, but also require low latency in real-time.

Note that, some protocols we use for demonstration here (BB84, and MDI-QKD with "independent bounds" finite key analysis as in [26]) are pretty fast to optimize to begin with, on the order of seconds even on a single-board computer. However, there are cases where optimization itself is much slower, for instance when global search and/or linear solver are needed. For instance, the asymmetric TF-QKD protocol [27] or the "9-intensity" MDI-QKD protocol (an asymmetric MDI-QKD protocol where Alice and Bob each use four instead of three decoy states, as described in Ref. [26], which requires using both linear solver and global search) would respectively take 2 seconds and 11 seconds to generate just one point even on a fast desktop PC. On single-board computers it is generally 10-30 times slower, meaning that an optimiza-

tion would likely take tens of seconds to even minutes, which is quite long since many free-space sessions might only have a window of minutes (e.g. satellite-ground or handheld QKD). Also, some alternative finite-key analysis, such as the "joint bounds" analysis (as opposed to using "independent bounds") proposed in [23], will introduce similar problems, as it involves linear solver and non-convexities in the key rate versus parameters function (which often necessitate global search).

Moreover, there are some practical reasons where software/hardware limitations might favor using neural network over performing local search on CPU on low-power platforms. For instance, as mentioned above, performing local search uses up all CPU resource, which would be non-ideal for drones and handheld systems that need the CPU for the control system, while a neural network running on a separate accelerator chip offloads the computational requirement. Also, software-wise, computing the key rate and performing optimization on CPU mean requiring the entire software stack to be set up on the mobile device - however, many software libraries, e.g. all commercial linear solver libraries, dont even work on mobile architecture such as ARM CPUs but a neural network pre-trained with data generated by linear solvers can still be run on these platforms. This can be shown in Table I where iPhones don't support linear solvers required for calculating the key rate, but a neural network can be used to directly output the parameters (without needing to calculate the key rate first).

Lastly, atmospheric turbulence causes the channel transmittance to quickly fluctuate at a time scale of 10-100 ms order [39] (and QKD over moving platforms can have average channel losses constantly changing with time e.g. due to changing distances). With a neural network it would be potentially feasible to quickly tune laser intensities based on sampled channel loss in real time. Nonetheless, a set of changing laser intensities would require modified finite-size analysis, so we only propose it as a possibility here and will leave the details for future discussions.

To benchmark the performance of neural networks on low-power devices, as examples, we choose the 4-intensity MDI-QKD protocol and the TF-QKD protocol, and test our neural network models on two popular mobile low-power platforms: a single-board computer, and a common mobile phone, as shown in Fig. 1. We implement both CPU-based local search algorithm and neural network prediction on the devices, and list the running time in Table I, where we compare neural networks to local search on the portable devices and on a powerful desktop PC. As shown in Table I, using neural networks, we can find the optimal parameters in milliseconds regardless of the protocol [1] (which is 2-4 orders of magnitude faster than local search on CPU), in a power footprint less than 1/70 that of a desktop PC.

In Table I we used the 4-intensity MDI-QKD protocol and the TF-QKD protocol as two examples, although note that for other protocols, e.g. 7-intensity MDI-QKD or BB84 protocol, the advantage of NN still holds, since the NN prediction time is little affected by the input/output size (for instance, in Fig. 2, there are 400×200 connections between the two middle hidden layers, and only 4×400 and 6×200 connections involving output or input neurons. This means that the numbers of input/output nodes have little impact on the overall complexity of the network), while local search time increases almost linearly with the number of output (searched) parameters. For instance, running 7-intensity MDI-QKD protocol, which has 12 output parameters, takes about 0.4s using local search on an iPhone XR - which is double the time of the 4-intensity MDI-QKD protocol, which has 6 output parameters - but with a NN it still takes about 1ms (making the advantage of using NN even greater in this case).

Additionally, note that even without neural network acceleration chips, many devices can still choose to (1) run the neural network on CPU (at the expense of some CPU resource), and this option is still much faster than local search (for instance, running neural network on iPhone XR with CPU takes between $1.3-2.0$ms, which is not that much slower than the dedicated neural accelerator chip). (2) generate a static "lookup table" for all possible inputs down to a given resolution. This is ideal for systems with extremely limited compute power or with software/hardware restrictions, such that neural networks cannot be run in real time. The lookup table can be generated using a GPU on a desktop computer first and stored on a mobile system to check when needed. This is slower than directly running a neural network, but it is still considerably faster than performing a local search. More details are discussed in Appendix A.

2. Quantum networks. In addition to free-space QKD applications which require low-power, low-latency devices, the neural network can also be very useful in a network setting, such as a quantum internet-of-things (IoT) where numerous small devices might be interconnected in networks as users or relays. For an untrusted relay network, MDI-QKD or TF-QKD protocol are desirable. However, the number of pairs of connections between users will increase quadratically with the number of users, which might quickly overload the compute resources of the relay/users.

With the neural network, any low-power device such as a single-board computer or a mobile phone can easily serve as a relay that connects to numerous users and optimizes e.g. ~5000 pairs of connections (100 users) in under 5 seconds for MDI-QKD or TF-QKD. This is a task previously unimaginable even for a powerful desktop PC, for which even supporting a network with 20 users would

[1] Note that, for neural networks it generally takes some time to load the model into the device when first used (about 0.2-0.3s on Titan Xp GPU and neural engine on the iPhone, and 3s on Raspberry Pi with the neural compute stick), but this only needs to be done once at boot time, and can be considered part of the startup time of the device - once the network is running, the predictions can be performed on many sets of data taking only milliseconds for each operation.

take from 20s (MDI-QKD) to 6 minutes (TF-QKD) to even 30 minutes ("9-intensity MDI-QKD"), if the protocol chosen is difficult to optimize. Therefore, our new method can greatly reduce the required compute power of devices and the latency of the systems when building a quantum Internet of Things.

V. CONCLUSION AND DISCUSSIONS

In this work we have presented a simple way to train a neural network that accurately and efficiently predicts the optimal parameters for a given QKD protocol, based on the characterization of devices and channels. We show that the approach is general and not limited to any specific form of protocol, and demonstrate its effectiveness for four examples: symmetric/asymmetric MDI-QKD, BB84, and TF-QKD.

We show that an important use case for such an approach is to enable efficient parameter optimization on low-power devices. We can achieve 2-4 orders of magnitude faster optimization speed compared to local search, with a fraction of the power consumption. Our method can be implemented on either the increasingly popular neural network acceleration chips, or on common CPUs that have relatively weak performance. This can be highly useful not only for free-space QKD applications that require low latency and but have limited power budget, but also for a quantum internet-of-things (IoT) where even a small portable device connected to numerous users can easily optimize the parameters for all connections in real-time.

Here we have demonstrated that the technique of machine learning can indeed be used to optimize the performance of QKD protocols. The effectiveness of this simple demonstration suggests that it may be possible to apply similar methods to other optimization tasks, which are common in the designing and control of practical QKD systems, such as determining the optimal threshold for post-selection in free-space QKD, tuning the polarization controller motors for misalignment control, etc.. Such a method might even be applicable for optimization and control tasks in classical systems, to use a pre-trained neural network in accelerating well-defined but computationally intensive tasks. We hope that our work can further inspire future works in investigating how machine learning could help us in building better performing, more robust QKD systems.

Note added: After our posting of a first draft of this work on the preprint server [40], another work on a similar subject was subsequently posted on the preprint server [41] and later published at [42]. While both our work and the other work [41, 42] have similar approaches in parameter optimization with neural networks, and observe the huge speedup neural network has over CPU local search, a few important differences remain. Firstly, we show that the neural network method is a general approach not limited to any specific protocol (and show its versatile applications with four examples), while Ref. [41, 42] is limited to discussing asymmetric MDI-QKD only. Secondly, we point out that a key use case of this approach would be performing parameter optimization on low-power devices with neural networks. This was only briefly mentioned in passing in Ref. [41, 42]. In contrast, we perform testing and benchmarking on real hardware devices. Our work not only will allow more types of smaller portable devices to join a network setting, but also can be important for free-space QKD applications where low power consumption and low latency are crucial.

VI. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), U.S. Office of Naval Research (ONR). We sincerely thank Nvidia for generously providing a Titan Xp GPU through the GPU Grant Program. Some simulation results are generated on the Niagara supercomputer, and we thank SciNet [43] for providing the compute platform. We would also like to thank ZH Wang for kindly providing an Intel neural compute stick for testing.

All training data are generated from simulations based on models in Refs. [22, 23, 26, 27], using local/global search algorithm. The generated datasets, and the neural networks trained from them, are available upon reasonable request to the authors by email.

[1] C Bennett, G Brassard, " Quantum cryptography: Public key distribution and coin tossing." International Conference on Computer System and Signal Processing, IEEE (1984).
[2] AK Ekert, "Quantum cryptography based on Bells theorem." Physical review letters 67.6:661 (1991).
[3] P Shor, J Preskill, "Simple proof of security of the BB84 quantum key distribution protocol." Physical review letters 85.2 (2000): 441.
[4] N Gisin, G Ribordy, W Tittel, H Zbinden, "Quantum cryptography." Reviews of modern physics 74.1:145 (2002).
[5] WY Hwang, "Quantum key distribution with high loss: toward global secure communication." Physical Review Letters 91.5 (2003): 057901.
[6] HK Lo, XF Ma, and K Chen, "Decoy state quantum key distribution." Physical review letters 94.23 (2005): 230504.
[7] XB Wang, "Beating the photon-number-splitting attack in practical quantum cryptography." Physical review letters 94.23 (2005): 230503.
[8] HK Lo, M Curty, and B Qi, "Measurement-device-independent quantum key distribution." Physical review letters 108.13 (2012): 130503.
[9] M Lucamarini, ZL Yuan, JF Dynes, AJ Shields, Overcoming the ratedistance limit of quantum key distribution without quantum repeaters. Nature 557.7705:400 (2018).

[10] M Takeoka, S Guha, and M Wilde, Fundamental rate-loss tradeoff for optical quantum key distribution, Nature communications 5:5235 (2014).

[11] S Pirandola, R Laurenza, C Ottaviani, L Banchi, Fundamental limits of repeaterless quantum communications." Nature communications 8:15043 (2017).

[12] K Tamaki, HK Lo, W Wang, M Lucamarini, Information theoretic security of quantum key distribution overcoming the repeaterless secret key capacity bound. arXiv preprint arXiv:1805.05511 (2018).

[13] XF Ma, P Zeng, H Zhou, Phase-matching quantum key distribution. Physical Review X 8.3 (2018): 031043.

[14] XB Wang, ZW Yu, XL Hu, Twin-field quantum key distribution with large misalignment error. Physical Review A 98.6 (2018): 062323.

[15] J Lin, N Lütkenhaus. Simple security analysis of phase-matching measurement-device-independent quantum key distribution. Physical Review A 98.4 (2018): 042332.

[16] M Curty, K Azuma, HK Lo, Simple security proof of twin-field type quantum key distribution protocol. arXiv preprint arXiv:1807.07667 (2018).

[17] F Xu, H Xu, and HK Lo, "Protocol choice and parameter optimization in decoy-state measurement-device-independent quantum key distribution." Physical Review A 89.5 (2014): 052333.

[18] M Hayashi, "General theory for decoy-state quantum key distribution with an arbitrary number of intensities." New Journal of Physics 9.8 (2007): 284.

[19] T Tsurumaru, A Soujaeff, and S Takeuchi. "Exact minimum and maximum of yield with a finite number of decoy light intensities." Physical Review A 77.2 (2008): 022319.

[20] M Hayashi, "Optimal decoy intensity for decoy quantum key distribution." Journal of Physics A: Mathematical and Theoretical 49.16 (2016): 165301.

[21] M Hayashi, and R Nakayama. "Security analysis of the decoy method with the bennettbrassard 1984 protocol for finite key lengths." New Journal of Physics 16.6 (2014): 063009.

[22] CCW Lim, M Curty, N Walenta, F Xu and H Zbinden, "Concise security bounds for practical decoy-state quantum key distribution." Physical Review A 89.2 (2014): 022307.

[23] YH Zhou, ZW Yu, and XB Wang, "Making the decoy-state measurement-device-independent quantum key distribution practically useful." Physical Review A 93.4 (2016): 042324.

[24] M Curty, F Xu, W Cui, CCW Lim, K Tamaki, HK Lo, "Finite-key analysis for measurement-device-independent quantum key distribution." Nature communications 5 (2014): 3732.

[25] K Maeda, T Sasaki, and M Koashi. "Repeaterless quantum key distribution with efficient finite-key analysis overcoming the rate-distance limit." Nature communications 10.1 (2019): 3140.

[26] W Wang, F Xu, and HK Lo. "Enabling a scalable high-rate measurement-device-independent quantum key distribution network." arXiv preprint arXiv:1807.03466 (2018).

[27] W Wang, HK Lo, Simple Method for Asymmetric Twin Field Quantum Key Distribution, arXiv preprint, arxiv: 1907.05291 (2019).

[28] AD Hill, J Chapman, K Herndon, C Chopp, DJ Gauthier, P Kwiat, "Drone-based Quantum Key Distribution", QCRYPT 2017 (2017).

[29] G Mlen, P Freiwang, J Luhn, T Vogl, M Rau, C Sonnleitner, W Rosenfeld, and H Weinfurter, "Handheld Quantum Key Distribution." Quantum Information and Measurement. Optical Society of America (2017).

[30] S-K Liao et al. "Satellite-to-ground quantum key distribution." Nature 549.7670 (2017): 43-47.

[31] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual" http://www.gurobi.com (2019).

[32] R Lougee-Heimer "The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community." IBM Journal of Research and Development 47.1:57-66 (2003).

[33] V Nair and GE Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th international conference on machine learning (ICML-10). 2010.

[34] R Hecht-Nielsen, "Theory of the backpropagation neural network." Neural networks for perception. 1992. 65-93.

[35] K Hornik, MStinchcombe, and H White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

[36] W Lu, C Huang, K Hou, L Shi, H Zhao, Z Li, J Qiu, "Recurrent neural network approach to quantum signal: coherent state restoration for continuous-variable quantum key distribution." Quantum Information Processing 17.5 (2018): 109.

[37] W Liu, P Huang, J Peng, J Fan, G Zeng, "Integrating machine learning to achieve an automatic parameter prediction for practical continuous-variable quantum key distribution." Physical Review A 97.2 (2018): 022316.

[38] DP Kingma and LJ Ba, "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[39] JP Bourgoin, "Experimental and Theoretical Demonstration of the Feasibility of Global Quantum Cryptography Using Satellites", PhD thesis at University of Waterloo (2014)

[40] W Wang, HK Lo, "Machine Learning for Optimal Parameter Prediction in Quantum Key Distribution." arXiv preprint arXiv:1812.07724 (2018).

[41] FY Lu, et al. "Parameter optimization and real-time calibration of measurement-device-independent quantum key distribution network based on back propagation artificial neural network." arXiv preprint arXiv:1812.08388 (2018).

[42] FY Lu, et al. "Parameter optimization and real-time calibration of a measurement-device-independent quantum key distribution network based on a back propagation artificial neural network." JOSA B 36.3: B92-B98 (2019).

[43] Chris Loken et al 2010 J. Phys.: Conf. Ser. 256 012026 doi: (10.1088/1742-6596/256/1/012026)

Appendix A: Lookup Table

As an alternative solution for devices with hardware limitations (very little CPU power and no GPU/AI-chip) or software limitations (libraries unsupported on the platform) that prevent them from directly running a neural network, it is still possible to get a speedup, by using a *pre-generated lookup table* of optimal parameters. For instance, for 4-intensity MDI-QKD, we can set a 100 point resolution to Y_0, e_d, and N, and 100 points from $L_{BC} = 0 - 200km$. This will result in a total of 1×10^8 data points that need to be calculated. Such a task is only possible with the parallelizable nature of the neural network, and the immense parallel processing power of the GPU. Predicting all the data points with a neural network on a desktop GPU would take an estimated time of 25 minutes. On the other hand, the local search algo-

TABLE IV. Time benchmarking of using local search algorithm versus using neural network (NN) inference and using pre-generated lookup table, for the 4-intensity MDI-QKD protocol. The desktop PC has an Intel i7-4790k quad-core CPU (with 16GB of RAM) and an Nvidia Titan Xp GPU. The single-board computers are a Raspberry Pi 3 with a quad-core CPU (with 1GB of RAM), and a Raspberry Pi Zero W with a single-core CPU (with 500MB of RAM). As can be seen, on the single-board computers, using a pre-generated lookup table is slower than directly using a neural network for inference, but it is still significantly faster than performing local search on CPU. By pre-generating a lookup table offline (e.g. on a desktop PC with GPU) and storing them on devices, we can still gain 15-25 times faster speed over local search on low-power devices, making the method suitable for devices where directly running neural networks is not feasible.

Device	Local search	NN	Lookup table
Desktop PC with GPU	0.1s	0.5-1.0ms	0.05s
Raspberry Pi 3	3-5s	2-3ms	0.2s
Raspberry Pi 0W	11-14s	N/A	0.5s

rithm takes 6 hours to generate the 4×10^5 training data alone, and would take as many as two months to sample all 1×10^8 input sets. The fast generation of such a lookup table is possible because we only take a small random sample (4×10^5) in the 4-dimensional input space, and use the neural network to learn the overall function shape with these data. Afterwards, once we have "learned" the function, we can predict (or, intuitively, interpolate) all the 1×10^8 points over the entire input parameter space with ease.

In Table IV we show a simple time benchmarking of the neural network inference and pre-generated lookup table versus local search algorithm on different devices including a powerful desktop PC and two models of low-power single board computers, for the 4-intensity MDI-QKD protocol. We can see that although using a lookup table is slower than directly running a neural network, it still has a significant advantage over local search on a CPU. This means that, the lookup table method is ideal for systems with extremely limited compute power or with software/hardware restrictions that prevent them from running a neural network (for instance, the Raspberry Pi 0W system has an older armv6 architecture, and neither Intel compute stick nor tensorflow are officially supported on the platform). The lookup table can be generated using a neural network running on a GPU on a desktop computer first, and stored on a mobile system to check when needed. Note that, this does not contradict a neural network's necessity, but rather is one of its application, since only with a neural network can we possibly generate a lookup table over such a large parameter space.

Nonetheless, such a database would take up more storage resource (generating, for instance, a 100-point resolution lookup table for 4-intensity MDI-QKD would take up roughly 2.4GB of space (assuming single-precision floating point of 4 bytes is used for each output parameter), which we can choose to divide into 10 smaller tables, each taking up 240MB space, to avoid loading the entire table in memory), for such low-power devices, storage space is a lot cheaper than the extremely-limited CPU and memory resource (for instance, Raspberry Pis can read SD cards, which can easily have 64-256GB of storage space), and using small but many databases, they can be quickly loaded in parts into Raspberry Pi's memory, too.

Therefore, here we show a simple solution to find optimal parameters using a lookup table pre-generated by a neural network offline, such that a speedup of up to 15-25 times can still be gained over running local search on a low-power device, even when directly running a neural network on the device is infeasible due to either hardware or software restrictions.