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: December 20, 2018)

For quantum key distribution (QKD) with finite-size effects, parameter optimization - the choice of intensities and probabilities of sending them - is a crucial step to gain optimal performance. Traditionally, such an optimization relies on brute-force search, or local search algorithm such as Coordinate Descent. Here we present a new method of using a neural network to learn to predict the optimal point for the convex optimization of parameters for QKD with any given set of experiment devices and quantum channel conditions. Selecting one protocol (symmetric 4-intensity measurement-device-independent QKD) as an example, we show that with neural-network-predicted parameters we can achieve over 99.99% of the optimal key rate. Harnessing the parallel processing power of a graphical processing unit (GPU), the neural network allows us to efficiently pre-generate on a desktop PC a large "look-up table" of optimal parameters for all possible experimental parameter and channel conditions, up to a finite resolution, and deploy this table to low-power devices. This eliminates the need for any on-device computation, and allows us to deploy finite-size QKD with optimal parameter setting to low-power mobile QKD devices that have limited computing power, short communication time, and quickly changing channel losses and experimental parameters (e.g. misalignment and background noise), such as in satellite, drone, or handheld QKD systems. We show that, on a Raspberry Pi 3 single-board computer, using look-up table is up to 15-25 times faster than local search. In addition to the look-up table method, we also point out the potential to directly deploy neural networks to an increasing number of mobile devices equipped with neural processing units for real-time fast parameter optimization.

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 source 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.

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 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 optimizations for the search of parameters.

Traditionally, the optimization of parameters is implemented as either a brute-force global search for smaller number of parameters, or local search algorithms for larger number of parameters. For instance, in several literature studying MDI-QKD protocols in symmetric [9] and asymmetric channels [10], a local search method called coordinate descent algorithm is used to find the optimal set of intensity and probabilities.

However, optimization of parameters often require significant computational power, and is usually performed on a powerful desktop PC. With the advent of deep 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 study the possibility of using neural networks to help predict optimal parameters efficiently, and to allow real-time parameter optimization even on low-power devices that could be deployed to portable or free-space user scenarios of QKD.

B. Neural Network

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, an calculates the activation. For instance:

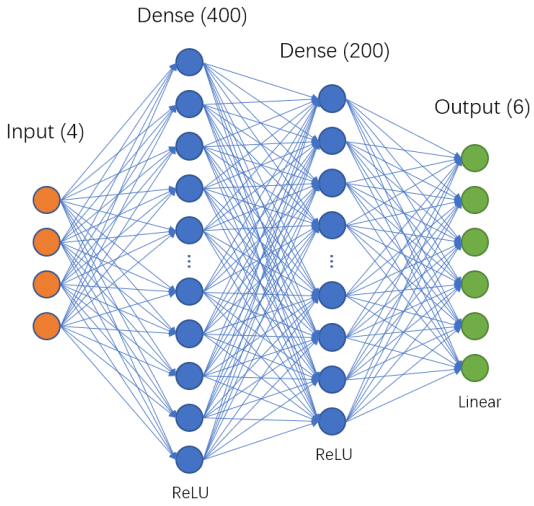

FIG. 1. An example of a neural network (in fact, here it is an illustration of the neural network used for this very project). It has an input layer and output layer of 4 and 6 neurons, respectively, and has two fully-connected hidden layers with 400 and 200 neurons with rectified linear unit (ReLU) function as activation. The cost function (not shown here) is mean squared error.

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

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) [11] 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 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 square 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 desired output $\{\vec{y_0^i}\}$. It uses an algorithm called "backpropagation"[12] 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 [13] 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 [14] and the prediction/compensation of intensity evolution of light over time [15], 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, misalignment, dark count, 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 over 99.99% the key rate.

We show that it's possible to use the neural network to pre-generate a "look-up table" of the optimal parameter to all possible inputs, which can be used as a static database that hardly requires any computational power from the device. This enables potential new applications in free-space or portable QKD devices, such as on a satellite[16], drone [17], or handheld [18] QKD system, where power consumption of devices is a crucial factor and computational power is severely limited, so that traditional CPU-intensive optimization approaches based on local or global search is infeasible. Additionally, we point out that the neural network could also potentially be directly deployed onto low-power real-time devices that are equipped with neural processing units (but have relatively weaker CPUs), to

facilitate parameter optimization in real time.

II. METHODS AND NUMERICAL RESULTS

A. Optimal Parameters as a Function

Here in this work we will use the symmetric "4-intensity MDI-QKD protocol" [19] as an example protocol to study parameter optimization. However, note that our method described in the following text can in principle also be applied to parameter optimization of other protocols too, such as finite-size BB84 [20], or asymmetric MDI-QKD [10], which are subject to future studies.

Here in the symmetric case, Alice and Bob have the same distance to Charles, hence they can choose the same parameters. The variables here will be a set of 6 parameters, $[s, \mu, \nu, P_s, P_\mu, P_\nu]$ for finite-size parameter optimization, where s, μ, ν are the signal and decoy intensities, and P_s, P_μ, P_ν are the probabilities of sending them. Since only 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, 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

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

However, this only calculates the rate for a given fixed set of intensities 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. Also, by maximizing R, we end up with a set of optimal parameters $\vec{p_{opt}}$. Note that $\vec{p_{opt}}$ is a function of $\vec{e}$ only, and 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 [9, 10], which evaluates 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 straightforwardly 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 parameter function $\vec{p_{opt}}(\vec{e})$. Once such a neural network is trained, it can be used to directly find the optimal parameter and 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) and greatly accelerate the parameter optimization process.

B. Numerical Results

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 [9] (as shown in Appendix A) 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.1. 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.

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), here as an example we fix it to 80% to simplify the network structure (so

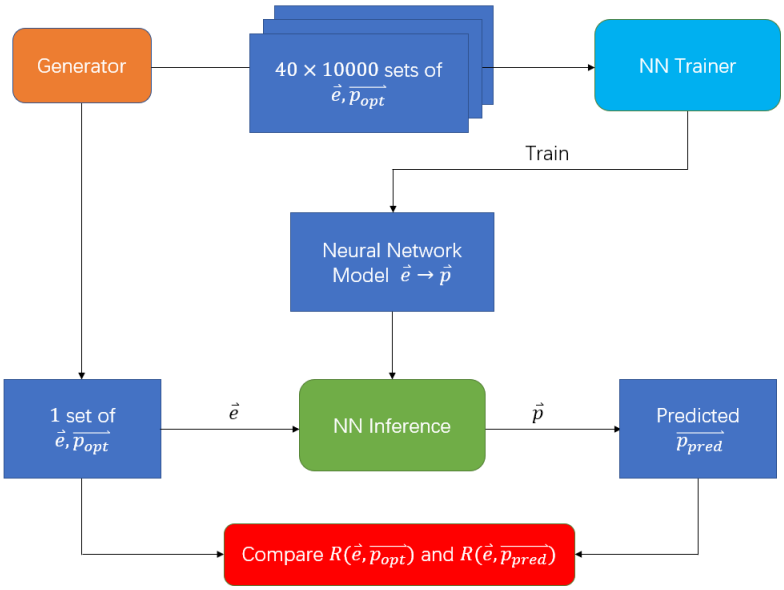

FIG. 2. Data flow of training and testing of the neural network (NN). The rounded boxes are programs, and squared boxes represent data. The generator program generates many random sets of experimental parameters $\vec{e}$ and calculates the corresponding optimal parameters $p_{\vec{opt}}$. These data are used to train the neural network. After the training is complete, the network can be used to predict on arbitrary new sets of random experimental data and generate $p_{\vec{pred}}$ (for instance, to plot the results of Fig. 3, a single random set of data is used as input). Finally, another program calculates the key rate based on the actual optimal parameters $p_{\vec{opt}}$ found by local search and the predicted $p_{\vec{pred}}$ respectively, and compare their performances.

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 amplitude 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.

We trained the network using Adam [21] as the optimizer algorithm for 120 epochs (iterations), which takes roughly 40 minutes on an Nvidia TITAN Xp GPU. After training is complete, we use the trained network to take in three sets of random data, and record the results in Table I. 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, and scan the neural network over L_{BC} =0-200km. The results are shown in Fig.3. 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 the traditional local search method.

III. DISCUSSION

Here we have demonstrated that a neural network (NN) can be trained to very accurately simulate the optimal parameter function $p_{\vec{opt}}(\vec{e})$, and be used in effectively predicting 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 an NN-prediction program? Here we discuss the advantages and disadvantages of the NN program compared to the CD algorithm, and some proposed use cases for the neural network.

The advantages of the NN is that, first, it is several orders of magnitude faster than the CD algorithm. In fact, due

TABLE I. Optimal vs NN-predicted parameters for MDI-QKD using three different random data sets, at the same distance L_{BC} of 20km between Alice 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 $\eta_d = 80\%$. 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 are very close to the actual optimal parameters, within an 1% error. Moreover, the key rate is even closer, within 0.01% error (this might be because, at the maximum position, the partial derivative for each parameter is close to 0, so a slight perturbation on parameters will change the rate very little. This means that we don't have to be very accurate on the optimal parameters to get a good rate).

Method	L_{BC}	Y_0	e_d	N	s	μ	ν	P_s	P_μ	P_ν	R
Optimization	20km	1.28×10^{-7}	0.0123	1.29×10^{13}	0.501	0.166	0.0226	0.911	0.00417	0.0589	1.3335×10^{-3}
NN	20km	1.28×10^{-7}	0.0123	1.29×10^{13}	0.502	0.167	0.0229	0.912	0.00414	0.0579	1.3333×10^{-3}
Optimization	20km	3.87×10^{-6}	0.0101	6.44×10^{12}	0.541	0.179	0.0256	0.904	0.00480	0.0636	1.5195×10^{-3}
NN	20km	3.87×10^{-6}	0.0101	6.44×10^{12}	0.542	0.179	0.0257	0.903	0.0473	0.0633	1.5194×10^{-3}
Optimization	20km	7.62×10^{-7}	0.0190	3.94×10^{11}	0.346	0.212	0.0336	0.792	0.0123	0.136	3.7519×10^{-4}
NN	20km	7.62×10^{-7}	0.0190	3.94×10^{11}	0.346	0.212	0.336	0.793	0.0120	0.135	3.7517×10^{-4}

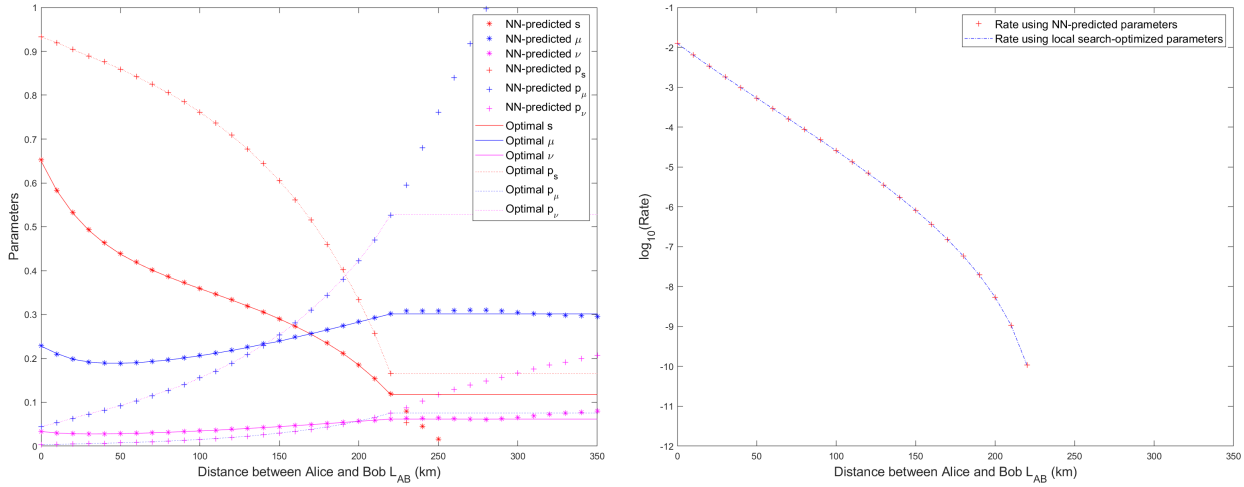

FIG. 3. Left: Comparison of neural network predicted parameters vs optimal parameters for a random experimental parameter set ($e_d = 0.0135, Y_0 = 6.16 \times 10^{-7}, N = 2.52 \times 10^{12}$) with fixed $\eta_d = 80\%$ and scanned through the distance between Alice and Bob of 0-400km (L_{BC} =0-200km). As can be seen, the predicted parameters are very close to the optimal parameters. Note that, since the NN does not have any training data on how to choose parameters after key rate is zero, it starts to output arbitrary results after the point where $R = 0$, but this does not affect the accuracy of the NN predictions since we are only interested in regions where $R > 0$. Right: Comparison of rate generated using predicted parameters, and optimal key rate. The blue line is the optimal rate found with local search algorithm, while the red data points are key rate calculated with NN-predicted parameters. As can be seen, with the predicted parameters we can get extremely close to the optimal key rate.

to the highly parallel architecture of the Graphics Processing Unit (GPU), the bottleneck of the computing time is only the communication from CPU to the GPU (which is roughly 200ms). There is no visible slowdown for data at the order of thousands. Notably, performing a prediction on all the 40×10000 training data takes only 2.8 seconds. Comparatively, the same set of data took over 23000s (approximately 6 hours) to generate with CD algorithm, which is over 8000 times slower.

Secondly, the NN program learns the entire input space of data, and can predict any combination of input without relying on initial search parameters. On the other hand, a CD algorithm requires that $R > 0$ at the start of the search. For instance, if required to calculate optimal parameters at $L = 40km$, the initial parameters might return a zero rate, therefore the CD algorithm has to start from $L = 0km$ and gradually evolute the distance parameter from there. If $R = 0$ even at $L = 0km$, it would have to resort to random scattershot strategy (i.e. test random combinations of parameters, until an $R > 0$ point that can start the CD algorithm is found). The NN program doesn't rely on starting position or neighboring data, and can accurately return the optimal parameter right away. This is also one of the advantages of the NN program.

The disadvantages, however, also comes from GPU. As the main bottleneck for GPU is the communication time with

TABLE II. Time benchmarking of using Coordinate Descent (CD) local search algorithm versus using look-up table pre-generated by neural network. The PC is a desktop computer with a quad-core i7-4790k@4GHz and 16GB of memory. The single-board computers are a quad-core 1.2GHz Raspberry Pi 3 (1GB memory) and a single-core 1GHz Raspberry Pi Zero W (500MB memory). As can be seen, on a powerful PC, the CPU-implemented local search method is fast enough, where the look-up table doesn't make much difference (the speed is likely mainly limited by disk access time). On the single-board computers, though, using a pre-generated look-up table is significantly faster than performing local search on CPU. Using the look-up table, we can gain 15-25 times faster speed on low-power devices, making the method very suitable for applications sensitive to both time and power-consumption, such as handheld QKD or satellite-to-ground QKD.

Device	local search with CPU	pre-generated look-up table
Desktop	0.1s	0.05s
Raspberry Pi 3	3-5s	0.2s
Raspberry Pi 0W	11-14s	0.5s

CPU, calculating a single point will be in fact slightly slower on the NN program (200ms) compared to 50-100ms for the CD algorithm (if the initial parameter is good and $R > 0$) on a quad-core desktop PC. Moreover, in applications that are not time-sensitive (say, an experimental setup with *fixed* device parameters and fixed channels, with a few weeks time allowed to set up and optimize parameters), it might be more straightforward to use CD, while the NN program is of interest to us for cases where faster prediction of optimal parameters is desired, such as a portable device for free-space QKD with parameters such as channel loss, misalignment, and background noise often changing in real time.

Here are some potentially interesting use cases for the neural network program:

1. **Look-up table for low-power QKD devices**. There are proposals to perform QKD on drones, via handheld devices, or on hot balloons, and satellites. On such devices, the power consumption is a major limiting factor, and the classical components (e.g. the on-board CPU) do not have significant processing power. Moreover, due to the inherently mobile characteristic of such systems, they often see varying channel transmittance and background noise level, and also require fast real-time feedback. In this case, due to the much weaker CPU, it might take significantly more time for the local search algorithm to run (especially if it meets a bad initial parameter set, and needs to scan multiple points or go for random trials, which would significantly increase the amount of computations), which may be non-ideal for real-time mobile platforms where processing time is crucial.

   Here, we present a rather audacious solution that hardly requires any processing power on low-power devices: using the neural network, we can *pre-generate* a static "look-up" table of optimal parameters on a desktop PC, and deploy this look-up table onto devices. For instance, we can set a 200 point resolution to each of Y_0, e_d, and N (which is well above the significant digits for measurements on the device parameters), and also 200 points within the distance range of $L_{BC} = 0 - 200km$. This will result in a total of 1.6×10^9 data points that need to be calculated. Such a task is only possible with the parallelizable nature of the neural network, and the immense processing power of the GPU. Using a Titan Xp GPU on desktop, predicting all the data points with NN-program would take an estimated time of 5.5 hours. On the other hand, running a local search algorithm on a quad-core i7-4790k desktop CPU would take as many as 24000 hours (which is roughly 2.7 years!). Intuitively, such generation of a complete "look-up table" is possible because we only take a small random sample (4×10^5) in the 4-dimensional input space as the training data, and use the neural network to learn the overall function shape with these data. Afterwards, once we have "learned" the function's behavior, we can predict all the 5×10^9 points over the entire input space with ease.

   In Table II we show a simple time benchmarking of the NN-inference and pre-generated database versus local search algorithm on different devices including a powerful workstation desktop computer and two models of low-power single board computers. We can see that on the low-power devices, using a pre-generated database will have a significant advantage over computing with local search algorithm on a CPU. Although such a database would take up more storage resource (generating, for instance, a 200-point resolution database would take up roughly 40GB of data, which can be divided into 400 smaller databases each taking up 100MB space), for such low-power devices, storage space is usually 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 here a simple solution for providing up to 15-25 times faster performance in parameter optimization, which can be very useful in applications where time and power are restricted. As Raspberry Pis use only roughly 5 watt energy, they can be easily deployed onto e.g. drones and handheld systems (for instance, a Raspberry Pi was in fact used as the on-board controller in the drone-based BB84 QKD experiment [17])

2. **Real-time neural networks on devices with neural processing units**.

TABLE III. Some representative low-power chips that includes on-board GPU or specialized neural processing units (NPU) for neural network inference acceleration, their performance in operations per second (OPS), and their power consumption. For comparison, an Intel 6-core CPU is included for comparison. Note that the performance and power consumption data included here are excerpted from online sources [22–30] (and also OPS is not consistently defined by different reports, and may be defined as operations on INT8, FP32, or (for CPU) FP64), so the listed data here is meant only for references to show the rough order-of-magnitudes for performance and power, and not as accurate representations for the specifications of these devices. We can see that these portable devices use much lower power consumption, yet are able to provide higher data throughput for running neural networks than even CPUs. However, these performances depend on highly specialized processing units and are only applicable to neural network inference tasks, and such devices will be much slower than CPU for non-neural-network-related tasks - such as local search algorithms. Therefore, using neural network and utilizing these GPU/NPU on-chip is potentially a viable option for efficient parameter optimization on low-power devices.

system on board/chip	manufacturer	OPS	power
Jetson TX2	Nvidia	2 trillion	7.5w
Jetson Xavier	Nvidia	30 trillion	<30w
A12 Bionic	Apple	5 trillion	<5w
Kirin 980	Huawei	2 trillion	<5w
Movidius Neural Compute Stick 2	Intel	1 trillion	1w
i7-5930k (CPU)	Intel	0.3 trillion	140w

While neural networks take immense computing power to "train" (e.g. on a dedicated GPU), using it to predict (commonly called "inference") is computationally much cheaper. Moreover, in recent years, with the fast development and wide application of neural networks, many manufacturers have opted to develop dedicated chips that accelerates NN-inference on mobile low-power systems. Some of the examples can be seen in Table III. Such chips can greatly 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 apps on cell phones). Therefore, it might be more efficient (and faster) to use an NN-program running on inference chips, rather than using the highly computationally intensive local search algorithm with CPU.

3. **Assisting local search algorithm in finding initial points**: Another use of the NN program would be to assist the local search algorithms. For instance, the coordinate descent algorithm relies on the condition that key rate $R > 0$ at the start of the search. If $R = 0$ with the initial parameters, the algorithm would need to start from zero distance (which is more likely to provide a non-zero key rate) and gradually evolute towards the desired distance, or even resort to random "scatter-shot" selection of initial parameters. This would take significantly more time than the theoretical time of 50-100ms for one point (since it would need to calculate many more points to evolute from zero distance towards the desired position, or to randomly select initial points until a non-zero key rate is obtained).

   With the help of the NN program, we can either use the NN program's output directly (which is already very close to accurate rate by itself), or if we are being more picky and would like the exact optimal point, we can start the local search algorithm from the point predicted by NN, to have an "educated guess" at a good initial position.

In this work we have used the symmetric 4-intensity MDI-QKD protocol as an example, but we would like to point out that, since our neural network studies the general projection of a vector onto another vector, the results here hardly depends on the length of input/output vectors, and can potentially be applied in general to other protocols as long as they have parameter sets in the form of $\vec{p_{opt}}(\vec{e}) = argmax_{\vec{p}\in P} R(\vec{e}, \vec{p})$. For instance, it could potentially be applied to asymmetric MDI-QKD based on our recently proposed 7-intensity protocol (where Alice and Bob optimize a total of 12 parameters $[s_A, \mu_A, \nu_A, P_{s_A}, P_{\mu_A}, P_{\nu_A}, s_B, \mu_B, \nu_B, P_{s_B}, P_{\mu_B}, P_{\nu_B}]$), or finite-size BB84 (where Alice and Bob optimize a total of 5 parameters $[\mu, \nu, P_\mu, P_\nu, P_X]$ where P_X is the probability of selecting X basis), for which we only need to adjust the number of output neurons in the NN to 12 or 5 instead of 6. The application of the method described in this work to more protocols is a subject of our future interest.

Additionally, for low-power devices, we have mainly focused on the method of look-up tables. However, as we have mentioned, neural processing units could potentially greatly accelerate neural network tasks, which may allow running the neural network model in real time - rather than as a pre-generated look-up table, and will save more storage space on the devices. We are interested in testing out the performance of our neural network model on such NPU-enabled devices in the future too.

Perhaps more importantly, here we have demonstrated that the technique of machine learning can indeed be used to optimize the performance of a QKD protocol. The effectiveness of this first simple demonstration suggests that it

may be possible to apply similar methods to other optimization tasks, which are common in the designing of practical QKD systems, such as determining the optimal threshold for post-selection in free-space QKD, tuning the motors for misalignment control, etc.. 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.

IV. 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.

[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] 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.
[10] 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).
[11] V Nair and GE Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th international conference on machine learning (ICML-10). 2010.
[12] R Hecht-Nielsen, "Theory of the backpropagation neural network." Neural networks for perception. 1992. 65-93.
[13] K Hornik, MStinchcombe, and H White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.
[14] 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.
[15] 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.
[16] S-K Liao et al. "Satellite-to-ground quantum key distribution." Nature 549.7670 (2017): 43-47.
[17] AD Hill, J Chapman, K Herndon, C Chopp, DJ Gauthier, P Kwiat, "Drone-based Quantum Key Distribution", QCRYPT 2017 (2017).
[18] 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).
[19] 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.
[20] 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.
[21] DP Kingma and LJ Ba, "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).
[22] https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/
[23] https://www.electronicdesign.com/industrial/dnn-popularity-drives-nvidia-s-jetson-tx2
[24] https://blogs.nvidia.com/blog/2018/01/07/drive-xavier-processor/
[25] https://venturebeat.com/2018/11/14/intels-neural-compute-stick-2-is-8-times-faster-than-its-predecessor/
[26] https://ai.intel.com/myriad-x-evolving-low-power-vpus-deep-neural-networks/
[27] https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets/5
[28] https://medium.com/syncedreview/ai-chip-duel-apple-a12-bionic-vs-huawei-kirin-980-ec29cfe68632
[29] https://ark.intel.com/products/82931/Intel-Core-i7-5930K-Processor-15M-Cache-up-to-3-70-GHz-
[30] https://www.pugetsystems.com/labs/hpc/Linpack-performance-Haswell-E-Core-i7-5960X-and-5930K-594/

Appendix A: Coordinate Descent Based Algorithm for Parameter Optimization

This local search method called coordinate descent (CD) has been proposed for symmetric MDI-QKD in Ref.[9] and also applied to asymmetric MDI-QKD in Ref. [10]. This algorithm is a variant of the Gradient Descent method commonly used to find local maxima/minima. Take 4-intensity MDI-QKD as an example, among the 6 parameters, the algorithm only searches along one dimension (one axis direction) at a time, find the best value along this axis (and replace the searched dimension's variable with the new optimal variable), and moves on to search the next dimension. For instance, for iteration i, this might be:

$$R^i = R(s^i, \mu^i, \nu^i, P_s^i, P_\mu^i, P_\nu^i) \tag{A1}$$

Then, if we iterate along s direction, we find the next optimal point by

$$R^{i+1} = \max_{s\in[s_{min}, s_{max}]} R(s, \mu^i, \nu^i, P_s^i, P_\mu^i, P_\nu^i) = R(s^{i+1}, \mu^i, \nu^i, P_s^i, P_\mu^i, P_\nu^i) \tag{A2}$$

and then we can move on to iterate along μ direction. The search algorithm is stopped if all 6 (7) iterations yield almost the same result (meaning that one is already at a maxima/minima), or the max iteration count is reached. The algorithm is illustrated in Fig.4. This method guarantees a local maxima(minima), which, if given that the rate is a convex function of the parameters, will be the global maximum/minimum.

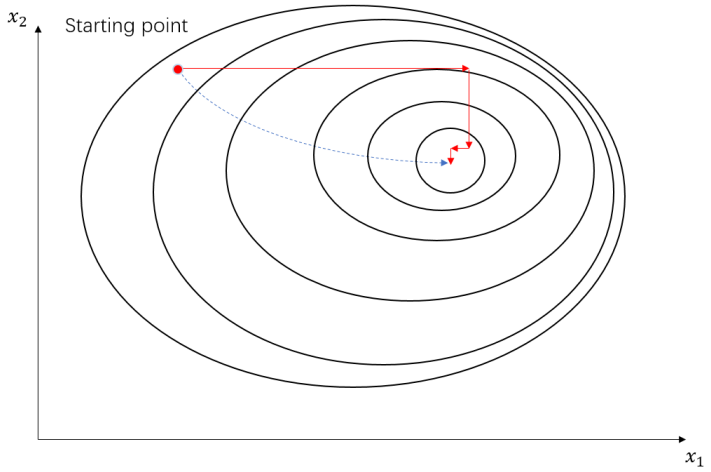

FIG. 4. Illustration of coordinate descent algorithm. Here we use two variables as an example. The algorithm descends along each axis at a time (and fix other axes) and iterates along each coordinate, until the optimal point is reached.

Additionally, to search along one specific axis, we can employ an additional method to further speed up the search: if we assume that there is only one maxima/minima along the axis, instead of starting out with a high search resolution, we can first coarsely search with large steps. Once we find the global maximum/minimum sample point, the actual minimum has to be within one step left or right of the optimal sample point, so we can start a finer search with one step left and right of the optimal point, with higher resolution. Such an iterative search can greatly increase the search speed along each axis, and can be efficiently parallelized.

Lastly, we note that the efficiency of the coordinate descent algorithm is starting-point dependent. Suppose the rate over parameter function is convex, we will always reach the same optimal point regardless of the starting point. However, if the starting point returns 0 rate (due to being too far away from optimal value), we will not be able to begin the search algorithm. To generate the optimal rate over a given set of continuous distance (say, a 1-D trace of rate over distance of 0-200km), it is efficient to use the optimal parameter set for the i_{th} distance as the starting point for the $(i+1)_{th}$ distance, under the assumption that the optimal point drifts slowly and continuously when the input distance (or transmittance) slightly changes. On the other hand, if we only want to get the optimal parameters at one distance (and that distance returns zero rate initially), we would have to start from zero distance, or (if it still has zero rate) use a "scattershot" approach to randomly generate parameters until one with a positive key rate can be found.