Final Report

Marco Russo

October 16, 2023

Contents

1 Introduction

Superconducting quantum computers are a particular technology that involve the realization of qubits by means of LC circuits with a Josephson junction. As shown in Fig. 1, this latter addition has the effect of introducing a nonlinearity (anharmonicity) that makes the frequency spacing between each energy level non-uniform. This is a desired effect for separating the two lowest levels, used for computation, from the higher ones, that are theoretically infinite and which are not desired to be used.

Qubits are controlled with electromagnetic pulses, i.e. microwaves, whose amplitude, phase and shape determine the angle and the axis of the rotation of the qubit state on the Bloch sphere, constituting therefore a quantum gate. It is easy to see that any 3D rotation can be achieved by means of a ZXZ rotation, so only R_z and R_x gates are necessary. Considering that R_z gates are virtually obtainable just by varying the phase of the pulses, it derives that we can just focus on the implementation of R_x gates. For any kind of relevant computation, this manipulation must be achieved with a high fidelity in order for the final result to be compatible with the theoretical one and to achieve the lowest possible error rate. This means that the pulse parameters must be optimal for every possible rotation. The parameters are chosen by the control system and the specific way that it chooses them determines the control algorithm. Standard feedback control is unfeasible, since measuring the states would imply the collapse of the wavefunction, and feedback itself requires some kind of measurement to happen. A choice can be feedforward control, where the algorithm operates according to some kind of optimization using a model of the system. In particular, Juqbox ([1]) can be used to simulate the hardware setting some physical parameters and then finding the optimal pulse parameters for a certain angle. Repeating this process for a set of angles, a set of parameters for them is obtained. This approach could be used online, which means that the control system could generate the pulse parameters on demand in real-time. However, this approach is unfeasible, since the control hardware has usually limited computational resources and the simulation and optimization stages take a lot of time, which would make the computation longer than desired (a gate should last in the order of nanoseconds). The opposite approach, that consists of building lookup tables offline, is unfeasible too because interpolation would provide a low accuracy and if we wanted to improve it we would have to build very large tables.

In this report another approach is explored, which consists of using a neural network. The idea is to train it on a set of angles and the corresponding parameters found by Juqbox, so that the network learns to find the parameters for all the other possible angles. A preliminary work ([2]) was already done in this sense using the Mean-Squared Error between the parameters found by the network and the ones found by Juqbox. However, the maximum reachable fidelity is limited by the fact that this loss function is not specific to the domain of the problem. Furthermore, it is also considered the fact that superconducting qubits present some characteristics, in particular the qubit anharmonicity (which describes the frequency spacing of the energy levels), that vary with temperature cycles: if a superconducting computer is heated up and then cooled down again, these characteristics change. Since the network was trained on some particular values of them, when they change then the accuracy inevitably decreases. This work also tries to address this problem pursuing the goal of achieving high fidelity for any possible scenario. The first step, as shown in the following sections, was to find a way to measure the fidelity of a set of gates that could suit this specific problem in an optimal way.

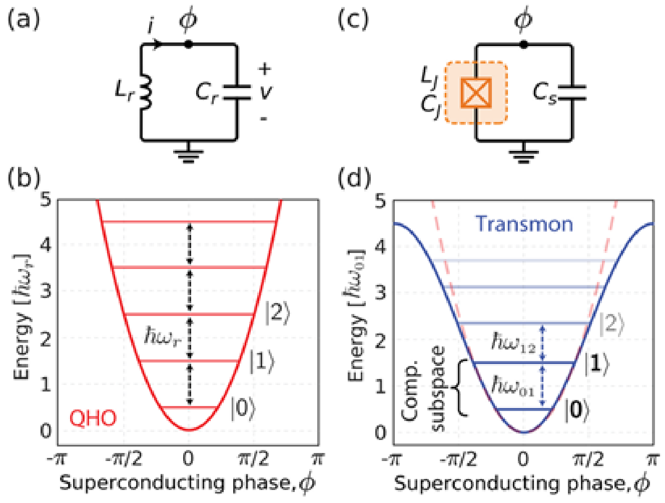

Figure 1: On the left, an LC circuit in superconductivity and the corresponding energy levels. On the right, the same LC circuit with the addition of a Josephson junction. The frequency spacing between each level is not uniform anymore.

2 Clifford groups

The standard in the literature is Randomized Benchmarking ([3]), which is based on testing the fidelity of Clifford gates. To understand what Clifford gates are, it is first useful to define the Pauli group. Given the unitaries $\sigma_0 = I, \sigma_1 = X, \sigma_2 = Y, \sigma_3 = Z$, the Pauli group for n qubits is defined as

$$\boldsymbol{P}_n := \{e^{\frac{ik\pi}{2}}\sigma_{j_1} \otimes \cdots \otimes \sigma_{j_n} | k = 0, \ldots, 3,\ j_k = 0, \ldots, 3\}. \tag{1}$$

For 1 qubit, this corresponds to

$$\boldsymbol{P}_1 := \{\pm\sigma_0, \pm i\sigma_0, \pm\sigma_1, \pm i\sigma_1, \pm\sigma_2, \pm i\sigma_2, \pm\sigma_3, \pm i\sigma_3\} = \langle\sigma_1, \sigma_2, \sigma_3\rangle. \tag{2}$$

At this point, the Clifford group is defined as

$$\boldsymbol{C}_n := \{V \in U_{2n} | V\boldsymbol{P}_n V^\dagger = \boldsymbol{P}_{\boldsymbol{n}}\}, \tag{3}$$

which, for 1 qubit, translates to

$$\boldsymbol{C}_{\boldsymbol{1}} = \langle H, S\rangle, \tag{4}$$

which corresponds to the group generated by the H and S gates. 1-qubit Clifford gates, then, are the gates whose unitaries belong to the 1-qubit Clifford group. It can be seen that the $R_x(\theta)$ gate is non-Clifford, except for some specific values of θ. The same goes for $R_z(\theta)$. For this reason, standard Randomized Benchmarking cannot be applied, and its statistical considerations are not valid anymore in this case.
The following section shows an adapted algorithm that can be used for non-Clifford gates, borrowing and adapting the idea of the original RB protocol, and showing a method for obtaining a confidence interval for the fidelity.

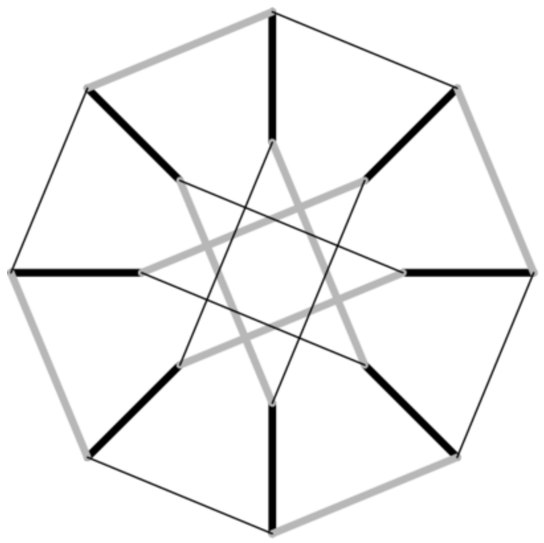

Figure 2: The Cayley graph of the Pauli group for 1 qubit.

3 Adapted Randomized Benchmarking (ARB)

Assuming that only $R_x(\theta)$ gates are to be tested, let $\mathcal{G}$ be the set of angles that we want to test and $|\mathcal{G}|$ its cardinality. For each angle θ_i, the corresponding imperfet gate is $\hat{R}_x(\theta_i)$. Follows the adapted Randomized Benchmarking algorithm for non-Clifford gates.

- Preliminary step 1: A set of all possible sequence lengths $M = \{m_1, \ldots, m_M\}$ is decided. Each sequence length defines the number of gates that are consecutively applied to the initial state (the last gate will be the inverse). Clearly, the denser M is the better the estimation will be; regarding the interval, a reasonable range would be from 2 to 100.
- Preliminary step 2: A total number K of random sequences is decided. Each $k-th$ sequence will be m gates long, depending on the current sequence length m. The first $m-1$ gates are uniformly sampled (possibly with repetition) among the ones that are to be tested. For each new sequence, a new sampling is done, so each sequence is different from all the others. A good K should be in the order of hundreds.
- For each $m \in M$:
  - For each sequence number $k \in \{1, \ldots, K\}$:
    * a random sequence of numbers $s_k = s_{1_k} s_{2_k} \ldots s_{m-1_k}$ is uniformly sampled ($s_{i_k} \in \{1, \ldots, |\mathcal{G}|\}$). Each s_{i_k} corresponds to the gate $\hat{R}_x(\theta_{s_{i_k}})$;
    * the initial state $|0 \ldots 0\rangle$ is prepared;
    * the sequence of gates $\hat{R}_x(\theta_{s_{1_k}}) \ldots \hat{R}_x(\theta_{s_{m-1_k}}) \hat{R}_x(-\sum_{i=1}^{m-1} \theta_{s_{i_k}})$ is applied;
    * N measurements are performed in the Pauli-Z basis; the probability of outcome 0 is estimated as $\hat{p}_{k,m} = \frac{\#zeros}{N}$, and its standard error is the one of the Binomial distribution, $SE_{\hat{p}_{k,m}} = \frac{\sqrt{\hat{p}_{k,m}(1-\hat{p}_{k,m})}}{N}$.
  - The average $avg_{\hat{p},m}$ of $\hat{p}_m$ is calculated over the K sequences, and its standard error will be $SE_{avg_{\hat{p},m}} = \frac{\sqrt{\sum SE^2_{\hat{p}_{k,m}}}}{K}$;
  - The current estimated fidelity with m gates is assigned to be $\mathcal{F}_m = avg_{\hat{p},m}$, and its standard error is $err_m = SE_{avg_{\hat{p},m}}$.

- After calculating the quantities for each m, a fit is performed for $\bar{\mathcal{F}}_m = A + Bf^m$. Here, A, B, f are the coefficients to be found, m is the independent variable and $\bar{\mathcal{F}}_m$ the dependent variable. The err_ms are used as uncertainties to obtain the resulting uncertainties on A, B and f. The bounds should be $0 \leq A \leq 1$, $0 \leq B \leq 1$, $0 \leq f \leq 1$.

- As a result from the fit, an estimate $\hat{f}$ for the single-gate fidelity is obtained. Using scipy.optimize.curve_fit ([4]), also a covariance matrix is obtained for the three parameters, $\Sigma \in \mathbb{R}^{3,3}$. The third diagonal entry (if f was used as third parameter for the fitting) will contain $SE^2_{\hat{f}}$;

- Performing a 2-tailed Student's t-test, where the number of degrees of freedom is equal to $|M| - 3$ (3 is the number of parameters), and setting $\alpha = 0.05$, it is possible to obtain a 95% confidence interval for f, corresponding to $\left[\hat{f} - t_\alpha\sqrt{\Sigma(3,3)}, \hat{f} + t_\alpha\sqrt{\Sigma(3,3)}\right]$.

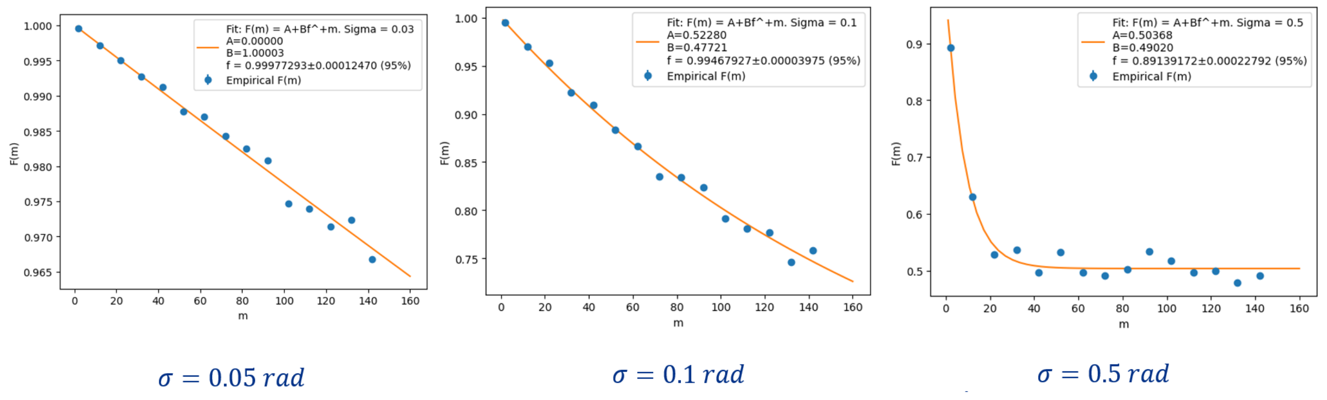

Figure 3: The first three experiments with artificially perturbed gates (K=500).

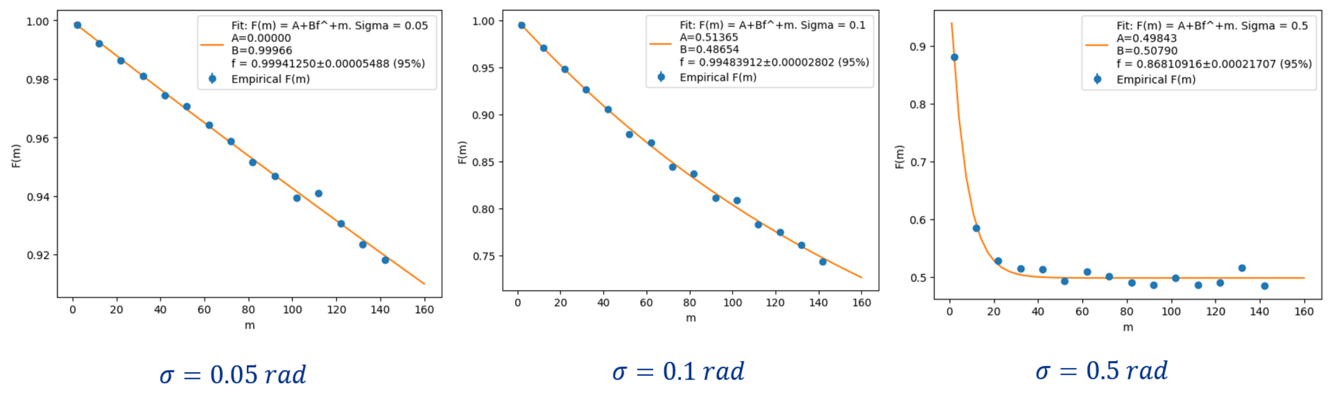

Figure 4: The last three experiments with artificially perturbed gates (K=1000).

4 Simulation of perturbed gates for testing ARB

To first have an idea of the results to be expected using ARB, an experiment is done generating a set of perturbed gates without simulating any physical system. In particular, the interval $[-\pi, \pi]$ is divided in 1000 angles and each angle θ_i is perturbed adding a gaussian noise, $\hat{\theta}_i = \theta_i + \mathcal{N}(0, \sigma)$. ARB is performed by doing a sequence of rotations using these perturbed angles, and the last rotation will be the inverse considering the sum of the exact angles instead. While the actual ARB shouldn't involve any exact gate even in the last unitary, here it was necessary since, if we used the sum of the perturbed angles to do the inverse, we would obtain exactly the initial state. An alternative could have been to use the sum of the same angles adding a further perturbation.
Many experiments are done, varying K and σ. In all of them $N = 1000$ and $M = range(2, 150, 10$. The first three have $K = 500$, and σ is equal first to $0.05rad$, then 0.1 rad and finally $0.5rad$. The results are shown in Fig. 3. The last three have $K = 1000$, and for σ the same variation is repeated. The results are shown in Fig. 4.

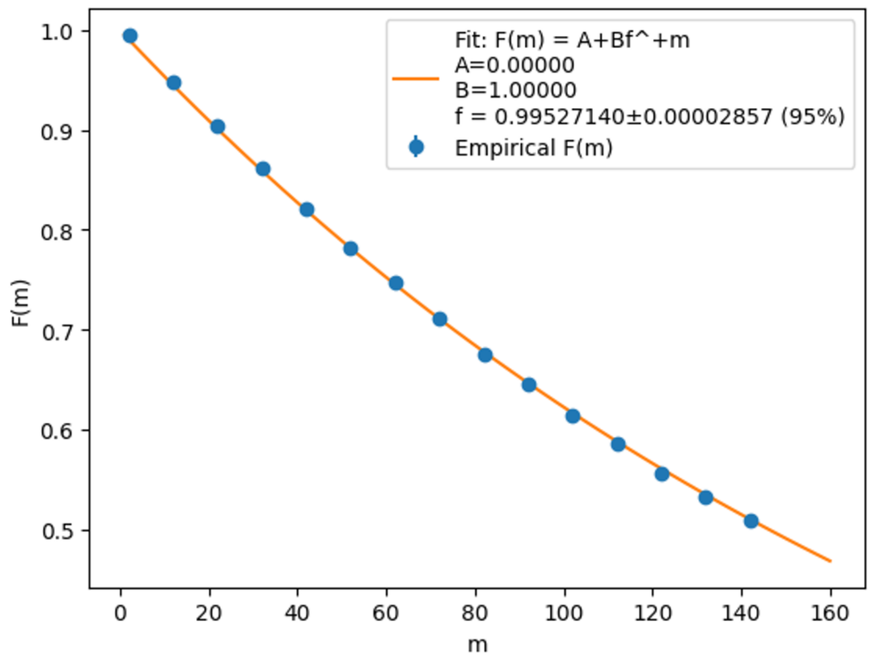

Figure 5: Fidelity estimated with ARB when the anharmonicity is unchanged.

5 Using ARB for testing a pre-trained Neural Network on different physical conditions

A neural network was previously trained considering a qubit with 0 guarded levels and an anharmonicity value equal to $200e-3$. The network was trained using as a loss function the Mean-Squared Error between the pulse parameters generated by Juqbox for the angles in the training set and the corresponding pulse parameters inferred by the NN. The actual physical system has infinite guarded levels, of which the probability to be populated gets lower the higher is the level. Plus, temperature cycles such as heating up the system and then cooling it down have the effect that some physical parameters, such as the qubit anharmonicity, become different. As a result, the performance of the NN would decrease, because it was trained on pulses that Juqbox generated considering different physical parameters. The idea is to use ARB to measure the fidelity of the pulses generated by the pre-trained neural network, first leaving the anharmonicity unchanged, then multiplying it by 10. In both cases, one guarded level is added. Before doing this, ARB was used to measure the fidelity with zero guarded levels and not varying the anharmonicity, which means that the scenario was exactly the one that the network was trained on, obtaining a fidelity equal to 0.99994.
As shown in Fig. 5, introducing one guarded level makes the fidelity decrease by two orders of magnitude. In Fig. 6, however, it is evident that multiplying the anharmonicity by 10 has the effect of compensating for this, since we are actually pulling the guarded level further apart from the two essential levels.

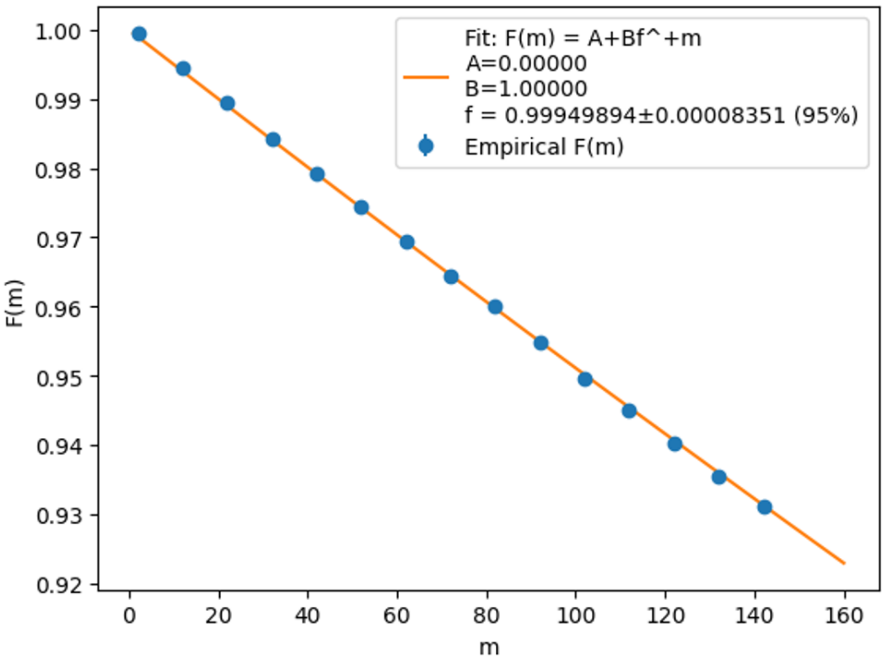

Figure 6: Fidelity estimated with ARB when the anharmonicity is x10 the one used during training.

6 Using ARB for fine-tuning

The idea is to take the pre-trained network and to fine-tune it to a different configuration, using ARB instead of MSE.
In particular, a scenario with G=1 guarded levels is considered, with a new anharmonicity equal to $200e-2$, which is 10 times the one that the network was trained on. Notice that this higher anharmonicity compensates for the new guarded level, as it provides a higher frequency separation of the two essential levels $|0\rangle, |1\rangle$ from the guarded level $|2\rangle$.

6.1 Code structure

The NN model was trained in Python using Tensorflow. As a result, it's natural to use Python to do the fine-tuning and the inference. From a high level point of view, the idea is, at each training epoch, to consider a set of angles from $-\pi$ to π and to infer a list of 20 pulse parameters for each of these angles. Then, for each angle, its corresponding list of parameters is used to make the system evolve according to the corresponding pulse and a unitary corresponding to that pulse is obtained. Doing this for all the angles, a set of unitaries is obtained corresponding to the gates that the network inferred for all the angles. At this point, ARB is run to measure the fidelity of these gates to the theoretical ones, and the weights of the network are updated accordingly performing a gradient descent.
A complication rises from the fact that Juqbox runs on Julia, so a way to transfer data from Python to Julia and viceversa must be found. One way is to read and write on files, but I/O can be notoriously slow. Another way, the one used here, is to use pipes, which are a type of Inter-Process Communication, exchanging data in JSON format.
Another thing to consider is that the unitaries that are obtained with Juqbox result from a numerical integration of differential equations, which means that due to limited precision they may not be, in fact, unitary. A renormalization is therefore done dividing them by their 2-norm. It could be useful to wonder whether this has a negative effect on training. It is also important to

note that Juqbox simulation is slow, especially when doing it for many angles, so a powerful server is needed.

6.2 Training

6.2.1 Optimization algorithm

The small version of the network has 8 dense layers (fully connected layers with ReLu activation function), resulting in 760 parameters (not to be confused with the pulse parameters; the network parameters are the weights and offsets of each layer). The fact that the loss function is calculated calling a script in another language makes it non-automatically differentiable for Tensorflow, so the gradient has to be computed by hand. The exact way would be to calculate the loss considering each parameter, first untouched, then perturbed by an ϵ, for us to estimate the derivative for that parameter; doing this for all the parameters we can estimate the gradient. This, however, results in the calculation of $760 * 2$ losses for each epoch, where each loss calculation involves a number of simulations equal to the number of angles used in training, which can be in the order of thousands. This massively slows down the process, and despite being accurate it is inconvenient.
Another method, proven to statistically converge to the same solution, is Simulatenous Perturbation Stochastic Approximation (SPSA). Here, all network parameters are perturbed at once, adding the same ϵ to all of them but with a sign that is random. Specifically, considering a function $f(\vec{\theta})$ of which we want to estimate the gradient, we perform

$$\nabla f(\vec{\theta}) = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \cdots \\ \frac{\partial f}{\partial \theta_n} \end{bmatrix} \approx \frac{f(\vec{\theta} + \epsilon\vec{\Delta}) - f(\vec{\theta})}{\epsilon} \vec{\Delta}^{-1} \tag{5}$$

where $\vec{\Delta} \in \{+1, -1\}^n$ and, in this case, $\vec{\Delta}^{-1} = \vec{\Delta}$ being its element their own reciprocal. The advantage of SPSA is that it only requires 2 evaluations for epoch, one with the perturbed parameters and one with the unvaried ones, independent of the network size.
During training, there are actually two hyperparameters that determine the "aggressiveness" of gradient descent. One is the ϵ by which the gradient is estimated, and the other is the learning rate α, where $\vec{\theta}_{k+1} = \vec{\theta}_k - \alpha\hat{\nabla} f(\vec{\theta}_k)$. Ideally for this problem they should be both around $10e-5$ with some tweaking if necessary. A smaller α makes the descent more stable, but it involves the risk of never exiting a local minimum. This is solved with a larger α, but then the training can become more unstable, never stopping at a minimum and keeping exiting the "valleys". Regarding ϵ, the smaller it is the better should the estimate of the gradient be, mimicking the $\vec{h} \to \vec{0}$ of the analytic gradient. On the other hand, a larger one can give a better idea of how the function changes further from the current point.

6.2.2 Training set construction

Also the way that the training set is built can vary. First of all, the network has to learn to generalize as much as possible, which is directly proportional to the number of angles used for training. However, the bigger the training set the slower the training. The idea is that, since the interval from $-\pi$ to π used for training is discrete, the network, while learning to minimize the loss on the angles in the training set, should also inherently be able to minimize it for all the angles in between as a result, without seeing them in training. This outcome gets naturally more likely as the interval gets denser. 1000 angles are a good starting point. A first try was done considering a training set whose angles are uniformly sampled for 500 times, and the same

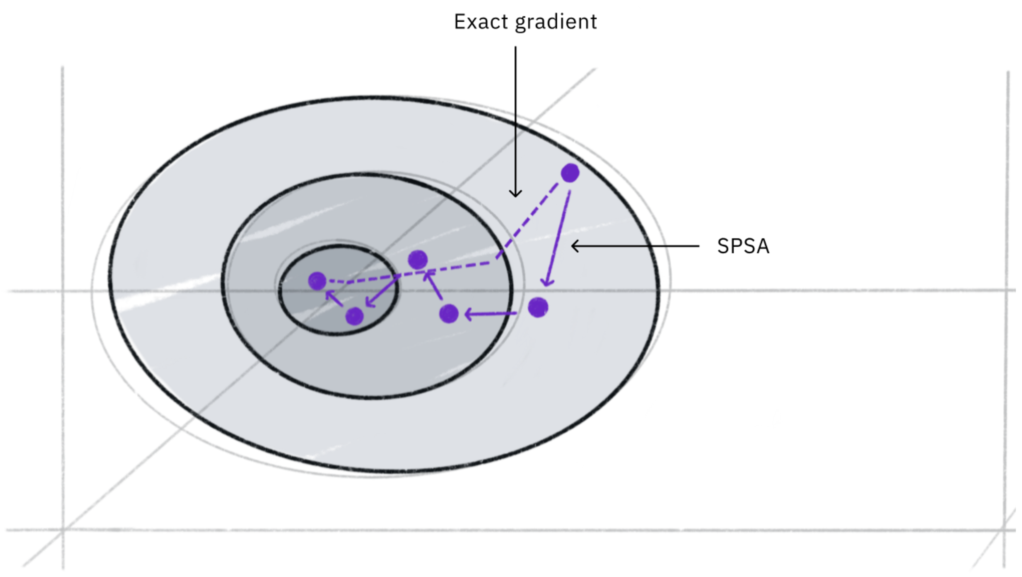

Figure 7: An illustration showing how SPSA works compared with standard gradient descent ([5])

was done for the validation set. The fact that they come from sampling means that the spacing between them is actually non-uniform. For providing a form of regularization, then, the training set was regenerated at each epoch, so that the network would have new angles to train on every time. Finally, $\alpha = 0.05$ was used.
A second try was done considering instead a fixed interval (not varying at each epoch) not sampled from a uniform distribution, but actually uniformly spaced. An $\alpha = 0.06$ was used. The training set is furthermore divided in 10 batches.

6.3 Results

The results of the first experiment are shown in Fig. 8. It is evident that the fact that the training set changes at each epoch makes the learning very hard and unstable, because we keep changing the examples that the network is being trained on. In Fig. 9, instead, the results of the second experiment indicate that fixing the training set and dividing it in batches is a better solution, achieving an improvement of the accuracy. However, since the training took place on a laptop, each epoch took approximately half an hour, so the process was overall very slow.

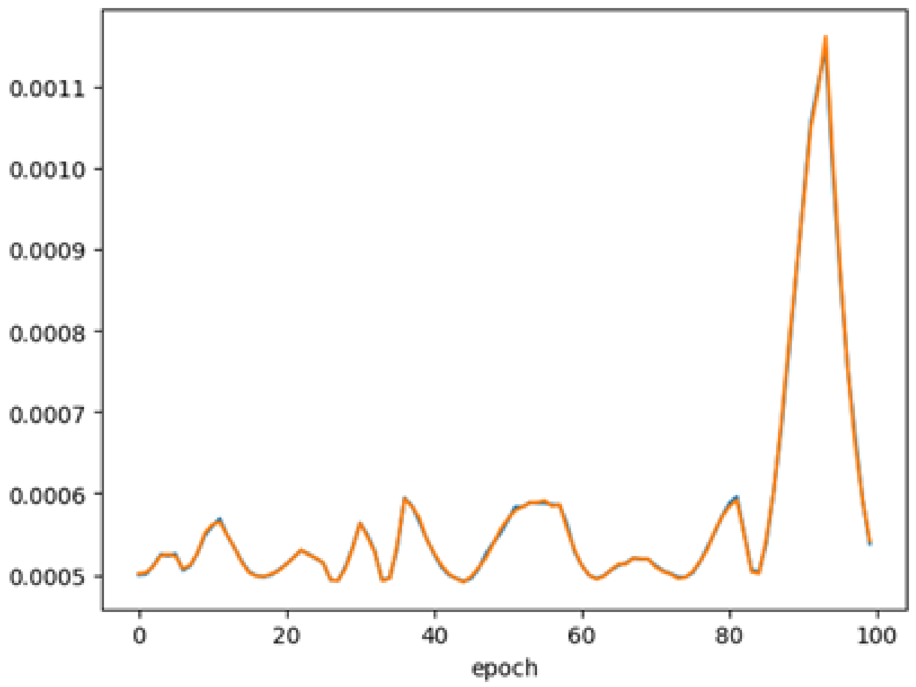

Figure 8: Loss function during training for the first experiment. Blue = training loss, red = validation loss

7 Conclusion

From the results, it is evident that many epochs are needed to achieve a consistent improvement of accuracy, but it's likely that, if the training is uncontrolled, the loss will at some point stabilize or start increasing again. For this reason, an early stopping mechanism is necessary, and some kind of regularization could be useful to avoid overfitting (the first experiment was actually an extreme form of regularization that didn't succeed). The loss function is highly non-convex so it's very hard to achieve its global minimum, and a hyperparameter tuning process can help reaching the best possible local minima. However, a bigger neural network will probably be necessary to be able to achieve a lower loss, and it could provide the benefit of a faster learning.
An extreme alternative would be to perform full gradient descent, but this would require 76000% more computations.
Finally, it can be proven that any Bloch sphere rotation can actually be decomposed in a sequence of $R_z, \sqrt{X}, \sqrt{X}^{-1}$ gates, and, since R_z can actually be virtually achieved just by varying the phase of the microwaves, this means that we can just focus on only 2 angles for the R_x gate $(-\frac{\pi}{2}, +\frac{\pi}{2})$. This would simplify the training by orders of magnitude and would probably allow us to achieve an even higher accuracy, to the cost of course of longer gate sequences for achieving a single rotation (2 R_x gates instead of just 1).
The architecture of the network could however be changed so to include the qubit anharmonicity among the inputs, together with the rotation angle. This way, there would be no need to retrain the network each time the anharmonicity changes, but it could just be trained once considering a set of possible anharmonicities (the training would then involve the comparison between the final state obtained using the pulse parameters inferred by the network and the one obtained using the pulse parameters found by Juqbox with that particular anharmonicity, doing this for all possible anharmonicities).
In conclusion, adapted Randomized Benchmarking is evidently a good way of optimizing neural

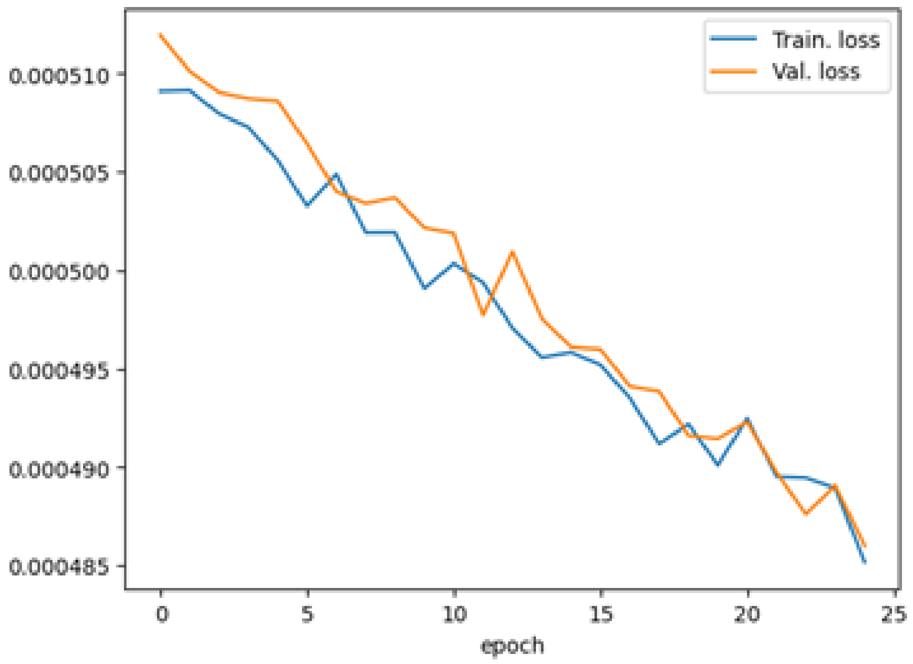

Figure 9: Loss function during training for the second experiment.

networks to work with varying physical conditions, better than simple MSE, making it possible to potentially achieve a high fidelity in any possible scenario.

References

[1] N. A. Petersson and F. Garcia, "Optimal control of closed quantum systems via b-splines with carrier waves," 2022.

[2] D. Xu, A. B. Ozguler, G. D. Guglielmo, N. Tran, G. N. Perdue, L. Carloni, and F. Fahim, "Neural network accelerator for quantum control," in *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*, IEEE, nov 2022.

[3] J. J. Wallman and S. T. Flammia, "Randomized benchmarking with confidence," *New Journal of Physics*, vol. 16, p. 103032, oct 2014.

[4] "scipy.optimize.curve_fit - scipy v1.11.3 manual," in *https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html*.

[5] "Training parametrized quantum circuits," in *https://learn.qiskit.org/course/machine-learning/training-quantum-circuits*.