Quantum Gradient Descent via Jordan's Method

Jakub Filipek

June 2020

Abstract

Gradient Descent has been a key technique for optimizing many Machine Learning Algorithms. However, due to extremely large number of parameters of modern models (up to 10^{11} floating points) this problem has not been widely discussed in Quantum Computing Community. In this project, I will try (and fail) implementing algorithm presented in [4]. Additionally I will follow by discussing potential improvements of that paper, as discussed in [2]. Both of these are done on example of $f(x) = x^2$.

0 Project Report Organization

I will start with the introduction of classical gradient descent algorithm, which will be unchanged, and classical gradient calculation. In Section 2 I will explain Jordan's Algorithm, provide an explicit circuit, and do a much more explicit analysis of it than done originally in [4].

In Section 3 I will describe the experimental procedure, show results, as well as discuss possible reasons for the failure of this experiment.

Section 4 will discuss improvements to Jordan's Algorithm presented in [2] on a more abstract level. This is followed by similarly abstract description of improvements done in classical gradient calculation in Section 5.

The project report ends with conclusion in Section 6.

1 Introduction to Gradient Descent

1.1 Gradient Descent

Gradient Descent is trying to solve a problem of minimizing (or maximizing) a function:

$$\arg\min_x f(x) | x \in X \tag{1}$$

, where X is a continuos space of numbers (typically in $\mathcal{R}^d$).

Since often finding a global minimum is computationally impossible, a lot of problems reduce to finding local minimum from a given starting point, and then repeating an experiment for set number of random starting points. In particular, vanilla gradient descent algorithm looks as follows:

Algorithm 1: Gradient Descent Algorithm

Result: An approximation of local minima of f for starting point $\boldsymbol{w}$, and dataset X
Let f be parametrized by $\boldsymbol{w}$. ;
Let $\boldsymbol{w}_0 = \boldsymbol{w}$;
for *i = 0 to T* **do**
  Let $\nabla_{\boldsymbol{w}} f(X)$ be an average of gradients of f over dataset X, all with respect to $\boldsymbol{w}$;
  $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \nabla_{\boldsymbol{w}} f(X)$;
end
return $\boldsymbol{w}_T$

While this algorithm does not necessarily lead to the local minimum it performs exceptionally well in practice. It also is a basis for a family of *gradient descent* algorithms which have been a backbone of last decade's improvements in Machine Learning.

More importantly however, by the above algorithm we can see that we need to calculate the above gradient $O(Td)$ times for each initialization point. Additionally if we want to achieve ϵ-precise result we need to have $|X| \in O(\frac{1}{\epsilon^2})$, if sampled randomly.

Hence overall, this naive algorithm will take:

$$O(\frac{dTN}{\epsilon^2}) \tag{2}$$

gradient calculations, where N is number of random starting points of $\boldsymbol{w}$.

1.2 Important Technicalities of Gradient Calculation

Classically, gradient at point $\boldsymbol{w}$ is calculated using standard derivative calculation for each dimension:

$$\nabla_{w_i} = \frac{f(\boldsymbol{w} + \boldsymbol{h_i}) - f(\boldsymbol{w})}{h} \tag{3}$$

$$\boldsymbol{h_i} = \big(0, 0, \ldots, h, \ldots, 0\big) \qquad \text{where } h \text{ is at index } i \tag{4}$$

As we can see the $f(\boldsymbol{w})$ can be reused across all dimensions, and hence we require f to be called $d+1$ times classically.

Alternatively, we can switch Equation 3 to:

$$\nabla_{w_i} = \frac{f(\boldsymbol{w} + \frac{\boldsymbol{h_i}}{2}) - f(\boldsymbol{w} - \frac{\boldsymbol{h_i}}{2})}{h} \tag{5}$$

which the same definition of $\boldsymbol{h_i}$ as above. This requires f to be called $2d$ times, since nothing can be reused.

The Jordan algorithm calculates gradient using the second method, which generally leads to more accurate result.

Additionally for future use, we can rewrite Equation 3:

$$f(\boldsymbol{w} + \boldsymbol{h_i}) = f(\boldsymbol{w}) + h\nabla_{w_i} \tag{6}$$

2 Jordan's Algorithm

2.1 Prerequisites

In this section I will try to provide a much more explicit explanation of needed function oracle, and its interpretation than presented in the original paper.

Firstly, let us define a classical function $f : x \mapsto y$, where both the domain and the range are fixed point numbers (those can be though of as integers divided by some 2^p number). Let input be n-bit precise, and the output to be 2^{n_o}-bit precise. This means that, for example, if the domain and range are both $[0, 1]$, the distances between distinct inputs and outputs have to be 2^{-n} and 2^{-n_o}, respectively.

For ease of future notation let $2^n = N$ and $2^{n_o} = N_o$.

This allows us to create f' such that $f' : x' \mapsto y'$, where $x' \in [0, N]$ and $y' \in [0, N_o]$. f' on the other hand can be easily converted into a quantum oracle such that:

$$O_f : |x\rangle\,|a\rangle \mapsto |x\rangle\,|(a + f'(x)) \mod N_o\rangle \tag{7}$$

Let us combine this thinking with the gradient calculation from Equation 5. Let $x \in [-\frac{l}{2}, \frac{l}{2}]$. Then conversion $f \to f'$ would split l into N integers, and shift x by $\frac{l}{2}$.

Hence corresponding $f(x) = \frac{l}{N} f'(x' + \frac{N}{2})$, or going the other way:

$$f'(x') = \frac{N}{l} f(x - \frac{l}{2}) = \frac{N}{l} f(\frac{l}{N}(x' - \frac{N}{2})) \tag{8}$$

And similarly for the O_f corresponding to the f'.

This particular relation between f, f' and O_f is not explicit in [4], and only can be understood by transformations between equations, while the transformation from f to O_f is described as trivial. I believe that because of that this paper lacks a lot of clarity and makes it much harder to reproduce experimentally.

2.2 Algorithm Explanation

Jordan Algorithm is still based on a vanilla gradient descent as mentioned in Section 1.1, however gradient calculation itself it done on quantum device as described in [4], which is as follows.

An important note is that this gradient is calculated for $\boldsymbol{w} = \mathbf{0}$, but we can just shift any function to center it at 0 perform this calculation, and then shift it back.

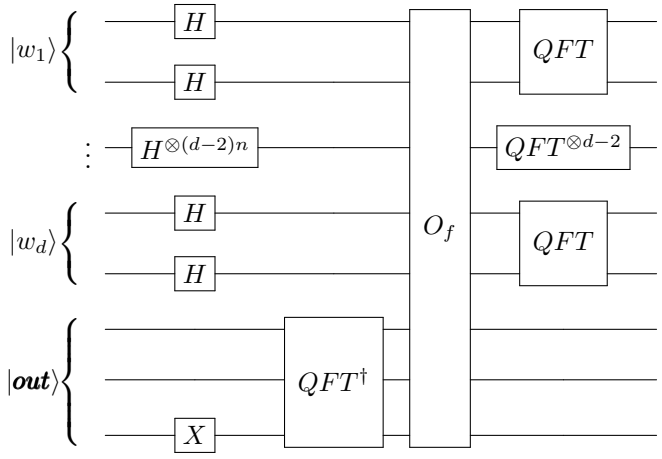

Figure 1: Circuit Visualization for Jordan's Algorithm

Then the above circuit (where inputs are all $|0\rangle$'s) will generate $|\frac{N}{m}\frac{\partial f}{\partial w_1}\rangle, |\frac{N}{m}\frac{\partial f}{\partial w_2}\rangle, \ldots$.

To show this let us consider the transformations of this circuit:

$$|0\rangle^{\otimes nd} |0\rangle^{n_o} \xrightarrow{H's, X} \tag{9}$$

$$\frac{1}{\sqrt{N^d}} \sum_w |\boldsymbol{w}\rangle |\mathbf{1}\rangle \xrightarrow{QFT^\dagger} \tag{10}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle \xrightarrow{O_f} \tag{11}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |(a + f'(\boldsymbol{w})) \mod 2_o^N\rangle = \tag{12}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a+f'(\mathbf{w})}{N_o}} |a\rangle = \tag{13}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{f'(\mathbf{w})}{N_o}} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle = \tag{14}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{N}{ml} f(\frac{l}{N}(w - \frac{N}{2}))} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle \qquad \text{from Equation 8} \tag{15}$$

Note the appearance of m in the last line. I can be thought of conversion of N_o (max magnitude of f') into magnitude of f, and is a maximal magnitude of ∇_f over the domain given domain. While I show derivation of relation between n_o and m, I will later explicitly use the formula derived in [4].

Continuing analysis of the circuit:

$$\frac{1}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{N}{ml} f(\frac{l}{N}(w-\frac{N}{2}))} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle = \tag{16}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{N}{ml}(f(0)+\frac{l}{N}(w-\frac{N}{2})\nabla_f)} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle = \quad \text{from Equation 6} \tag{17}$$

$$\frac{e^{i2\pi \frac{N}{ml}(f(0)-\frac{N}{2m})}}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{w}{m}\nabla_f} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle = \tag{18}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_w e^{i2\pi \frac{w}{m}\nabla_f} |\boldsymbol{w}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle \xrightarrow{QFT} \quad \text{Ignoring Global Phase} \tag{19}$$

$$\frac{1}{\sqrt{N^d N_o}} \sum_{i=1}^{d} |\frac{N}{m}\frac{\partial f}{\partial w_i}\rangle \sum_a e^{i2\pi \frac{a}{N_o}} |a\rangle \quad \text{Since } \nabla_f \text{ is just a vector of partial derivatives} \tag{20}$$

And hence we get a scaled gradient in the input registers.

3 Numerical Experiment

3.1 Problem Description

For the numerical I decided to calculate gradient of a rather simple function: $f(x) = x^2$. In this case input is a two dimensional vector, with an arbitrary domain (though I focused on region $[0, 2]$ for numerical reasons).

3.2 Implementation

For implementation I have used numpy, because initially I thought it would be the easiest way of implementing the code. However, now, I would probably use $Q\#$ and study less general problem due to failure of this implementation as explained later in Section 3.3.

As mentioned before I used:

- $d = 2$
- $n = 2$
- n_o was a hyperparameter I played around with and will discuss its impact in Section 3.3.
- $l = \frac{1}{8} = 0.125$

While I am aware of the fact that these are not values that are optimal or relate to each other in a consistent way, as described in Section 5.1 of [2], using them does not significantly change results and conclusions presented in Section 3.3. However, using them allowed for much faster dev time and testing different hypotheses about bugs.

Given a binary vector x, number of dimensions of input d, number of qubits for each input dimension n, number of qubits in the output register n_o and some d-dimensional point p for which to calculate f, the O_f

was implemented as follows:

Algorithm 2: My Implementation of Jordan's Algorithm

Result: Result of f at point p encoded into $nd + n_o$ dimensional binary vector
Let result = 0;
for $i = 0$ *to* d **do**
  Let x_i be the integer with binary representation x[ni:n(i + 1)];
  Let $x'_i = l\frac{x_i - \frac{N}{2}}{N} + p_i$;
  result = result + x'^2_i;
end
result = $\lfloor \frac{N}{l}$result$\rfloor$;
Let r be result encoded in n_o-binary vector (mod 2^{n_o} if needed);
Add r to the output register of x (mod 2^{n_o} if needed);

In more intuitive description Algorithm 2 performs following steps:

- Initialize sum to 0
- For each feature:
  - Convert feature from x' format to x (to match Equation 8)
  - Add result of $f(x)$ to sum
- Convert sum back to x' format (i.e. Integer)
- Add to output register

I had to also implement QFT, but since the circuit for that given in the class, I will not go into detail. All of the code (along with TeX and PDF version of this paper) is available in this repository.

3.3 Results

The first and initial test I have performed was to calculate the gradient at $p = \big(0, 0\big)$, since this exactly the case [4] described. For varying n_o I get following results:

n_o	3	4	5	6	7
$P(0)$	$1 - 9 \cdot 10^{19}$	$1 - 4 \cdot 10^{19}$	$1 - 3 \cdot 10^{19}$	$> 1 - 1 \cdot 10^{19}$	$> 1 - 1 \cdot 10^{19}$

We can see that the algorithm outputs the correct answer (i.e. gradient is equal to 0) with extremely high probability. This comes with no surprise since, in this case $O_f = \mathbb{1}$, and hence for input algorithms the whole circuit evaluates to $(\text{QFT})(\text{QFT}^\dagger) = \mathbb{1}$.

However, the implementation of the Jordan's Algorithm starts breaking when we move point p away from the function minima. Firstly for each point we will need to calculate m, which is the maximal derivative (i.e. $\max\{|\frac{\partial f(p+l)}{\partial x}|, |\frac{\partial f(p-l)}{\partial x}|\} = \max\{|2(p + l)|, |2(p - l)|\}$).

p	0.5	1	2
m	1.25	2.25	4.25
$\frac{N}{m}$	3.2	1.78	0.94
$\frac{N}{m}\frac{\partial f}{\partial x}$	3.2	3.55	3.76

Table 1: Approximate indexes of states that should have high probabilities after running Jordan's Algorithm for each p. A point corresponding to value p in this table is (p, p) (i.e. value is encoded in both dimensions).

We can see that $\frac{N}{m}$ is no longer 0 so we cannot expect $|\frac{N}{m}\frac{\partial f}{\partial x}\rangle$ to be exact integer. However, simultaneously we can see that we would expect the highest probabilities in $|3\rangle, |4\rangle$ and $|4\rangle$ respectively.

This is not reflected in the numerical results however, since the respective runs results in:

p	n_o	3	4	5	6	7
0.5	$P(3)$	0.031	0.037	0.15	0.0043	0.0011
1	$P(4)$	0.0	0.18	0.083	0.022	0.0052
2	$P(4)$	0.0	0.0	0.18	0.083	0.022

Table 2: Failed runs of the program. The moment we move away from the origin, the algorithm does no longer calculate the correct gradient.

We can see that the probabilities for correct answers are not close to the 1 at all. In fact all of them are below 0.2. This means that the algorithm is not implemented correctly, but there is still some debugging, and analysis to pin-point where the problem is in the implementation.

3.3.1 Possible Reason for Failure

Let us look at probabilities of $|0\rangle$ for the failed runs, which are shown in Table 2:

p	n_o	3	4	5	6	7
0.5	$P(0)$	0.18	0.67	0.91	0.98	0.99
1	$P(0)$	0.0	0.18	0.67	0.91	0.98
2	$P(0)$	0.0	0.0	0.18	0.67	0.91

Table 3: Probabilities of $P(0)$ state for the failed runs of the algorithm.

The interesting trend in Table 3 is a *drift* of $P(0)$ as we increase n_o, or number of qubits in the output register.

This suggests that there might be a significant problem with scaling that happens somewhere along in the algorithm. It most probably is due to my misunderstanding of some specific part of the algorithm, since I have tested both QFT and $f(x) = x^2$ for variety of inputs, and they have been providing expected outputs. I have not been able to exactly pinpoint what is causing the issue but to my best understanding it is within a function:

```
    result = 0
    relative_end = end - start
    half_size = 2**(size_of_dim - 1) * 1.0
    for disp_idx, idx in enumerate(range(0, relative_end - size_of_out, size_of_dim)):
        x = bitstring_to_num(inp[idx:idx + size_of_dim])
        x_arg = (x - half_size) / (2 * half_size) * l - displacements[disp_idx]
        result += x_arg ** 2

    result *= half_size * 2 / l

    clean_result = num_to_bitstring(int(result), size_of_out)
    inp[relative_end - size_of_out:relative_end] = \
        add_bit_strings(inp[relative_end - size_of_out:relative_end], clean_result)
```

which tries to implement the Algorithm 2.

This brings me to definition of m which is very vague in [4], with its description being "m is the size of the interval which bound the components of ∇f". My description of it above Table 1 matches it, but it is very possible that I have not implemented it properly.

Here, I just wanted to conclude my efforts on the practical side of the project. My goal was to work alone trying to reproduce results from [4], and expand on them by comparison to modern gradient calculation tools (autograd, JAX etc.). The main motivation behind that was that gradient calculation technique on classical computing also improved significantly over last decade.

However, I believe that what I have shown is that [4], while being of huge importance in for Quantum Gradient Calculation, is not explicit enough to be easily reproducible, and requires testing a lot of assumptions implicit within the text.

4 Improvements to Jordan's Algorithm

4.1 Different Way of Expressing Jordan's Algorithm

[2] presents a modified version of Jordan's Algorithm which uses phase-oracle rather than a bit-oracle. This allows for a more-intuitive representation of range by just real number. The precision is still bounded by domain, which is a bit-encoding of a state.

This results in a modified algorithm (Algorithm 2 in [2]). I have also tried implementing it, but failed similarly to original Jordan's algorithm. That being said I have not spent nearly enough time debugging to have any significant conclusion about implementation.

4.2 Amplitude Amplification

As briefly mentioned in Section 2.2 the algorithm does not output the correct gradient with 100% accuracy. This is because there can be higher degree (non-linear) terms in the gradient of the function, which cause uncertainty in the gradient calculation.

Assuming that we want to output the algorithm with ϵ accuracy, we would need to perform Jordan's algorithm $\frac{1}{\epsilon^2}$ times. However, this can be quadratically improved with Amplitude Amplification [1], as shown in class, to $\frac{1}{\epsilon}$.

This brings down the overall complexity of the algorithm (previously mentioned in Equation 2) down to:

$$O(\frac{dTN}{\epsilon}) \tag{21}$$

4.3 Grover Search

Since the algorithm using N data-points to calculate a gradient on, we can use a slightly modified version of Grover's Search [3]. Here, we need a subroutine that is finding a local minimum for a given point, (which is given by Algorithm 1), then we can use a continuos global optimization algorithm described in [5] (Algorithm 3).

This routine chooses up to $O(\sqrt{N})$ points at random at performs a local optimization algorithm from them. The slight modification in case of gradient descent is that M (a set of marked elements) is not really defined. However, we can define a threshold below which a minimum after gradient descent should be in order for that element to be marked. As we keep hitting lower and lower minima we can decrease this threshold decreasing size of M. This allows us to use almost the exact version of the algorithm described in [5], which results in overall complexity of the algorithm decreasing to:

$$O(\frac{dT\sqrt{N}}{\epsilon}) \tag{22}$$

4.4 $\sqrt{d}$ Improvements

I will give a very short and brief explanation of achieving a $\sqrt{d}$ complexity, as presented in [2]. The idea behind it is that a majority of functions that are being optimized are differentiable up to a very high order. This can lead to expressing the error of gradient to be less than the sum of these higher-order terms. This on the other hand nicely evaluates to be in $O(\sqrt{d})$.

A much more in-depth explanation of this method in presented in proof of Theorem 25 and Appendix A of [2]. When combined with both Grover Search and Amplitude Amplification the algorithm complexity of:

$$O(\frac{T\sqrt{dN}}{\epsilon}) \tag{23}$$

5 Devil's Advocate: Computational Improvements to Classical Gradient Descent

5.1 Modern Gradient Calculation Frameworks

Modern Frameworks such as PyTorch [7] or TensorFlow [6] are using a method called computation graphs to calculate gradients. These are effectively $O(1)$ computational time, since they do almost exactly the same number of operations as f. (Causing the calculation to be $O(TN)$ if parallelized)

For each of the operations such as activation functions, matrix multiplications, convolutions, loss functions etc. these frameworks have huge maps that map from an operation to its derivative calculation. In other words a gradient by linear approximation does not have to be calculated because it is known by the system already.

These frameworks essentially build a huge graph of computation in memory (along with weights creating a model), which when doing inference is traversed forward and when updating weights is traversed backwards. Due to vast popularity and size of these tools, the space of functions that can be realistically approximated is large.

However, there exist functions that will not be possible to realistically approximate at which point quantum computers will have a significant advantage.

Lastly, since quantum computers can provide speed-ups in solving linear equations there is still a role in gradient computation for them even in the computation graph method.

5.2 Active Sampling

Vast majority of quantum algorithms and test cases described in the quantum computing papers mentioned in this project report are rather simple functions. In classical optimization Active Learning algorithms tend to perform extremely well in such scenarios.

The simplest example for that would learning a threshold in a step function, there if we are sampling passively, we are not changing a domain from which we are sampling, and hence we would need $O(\frac{1}{\epsilon^2})$ to achieve error ϵ with regards to true threshold. However when using an active learning, we are constantly decreasing the domain, so that each new sample gives us new information. In such case a $O(\frac{1}{\epsilon})$ is needed.

The trade-off is pretty similar to Amplitude Amplification, and indeed these two ideas are very correlated, since they both are trying to maximize probability of sampling a meaningful information sample. However, it seems to me that Amplitude Amplification is stronger, since it does not require marked states to be close to each other.

There exist many different active learning algorithms some focused more on exploration of data (i.e. gain as much information with as few samples as possible), and some focused on exploitation (i.e. gain as much performance with as few samples as possible). Since these have implemented only on classical computers so far, they are causing certain trade-offs when comparing actually optimal classical and quantum algorithms.

Active Learning is still a rapidly growing field and it is unclear what gains (or sacrifices in equality) it might bring. Those techniques, in principle, should be implementable in quantum computers, thus further improving on gains of the quantum algorithms.

6 Conclusion

In conclusion this project have diverted from the initial plan significantly. I have experienced major issues with small tweaks that had to be done to get any reasonable output through O_f when implementing Jordan's algorithm, and still failed in the end. This forced me to read through [2] more carefully, but implementation of this algorithm was also not successful (though it was a last minute effort).

Overall, the project shifted that to more abstract discussion of the algorithm, along with more explicit explanation of the circuit, and how state looks like after each step. The more abstract discussion of improvements to Jordan's algorithm forced me to rethink whether there are any improvements on classical side, which resulted in Section 5.

While it is not a project I envisioned when submitting a proposal I believe that especially argument from Section 5.1, shows that it will be an extremely long time until quantum computers might be useful in quantum calculation, if ever. This is apart from faster matrix multiplication.

Lastly, I wanted to thank for a great course. I have learned a ton of stuff about quantum algorithms, which in turn helped me understand more sophisticated problems/algorithms such as ones mentioned during Quantum Seminars (at least on a basic level). While this quarter has extremely tough for a number of reasons, I really enjoyed a course, and even though it has been sometimes frustrating, I feel like it was a time well spent.

References

[1] G. Brassard and P. Hoyer. "An exact quantum polynomial-time algorithm for Simon's problem". In: *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems* (). DOI: 10.1109/istcs.1997.595153. URL: http://dx.doi.org/10.1109/ISTCS.1997.595153.

[2] András Gilyén, Srinivasan Arunachalam, and Nathan Wiebe. "Optimizing quantum optimization algorithms via faster quantum gradient computation". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms* (Jan. 2019), pp. 1425–1444. DOI: 10.1137/1.9781611975482.87. URL: http://dx.doi.org/10.1137/1.9781611975482.87.

[3] Lov K. Grover. "Quantum Computers Can Search Rapidly by Using Almost Any Transformation". In: *Physical Review Letters* 80.19 (May 1998), pp. 4329–4332. ISSN: 1079-7114. DOI: 10.1103/physrevlett.80.4329. URL: http://dx.doi.org/10.1103/PhysRevLett.80.4329.

[4] Stephen P. Jordan. "Fast Quantum Algorithm for Numerical Gradient Estimation". In: *Physical Review Letters* 95.5 (July 2005). ISSN: 1079-7114. DOI: 10.1103/physrevlett.95.050501. URL: http://dx.doi.org/10.1103/PhysRevLett.95.050501.

[5] Pedro Lara, Renato Portugal, and Carlile Lavor. *A New Hybrid Classical-Quantum Algorithm for Continuous Global Optimization Problems*. 2013. arXiv: 1301.4667 [math.OC].

[6] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[7] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.