Training General Quantum Neural Nets

Bryan Coutts

September 2, 2020

1 Introduction

This writeup documents some work I did in writing a tool to train quantum neural networks. The idea for this work was based on **this paper** by Farhi and Neven. This paper focused on taking a quantum circuit with some small (1-qubit and 2-qubit) gates, each one parameterized by a scalar θ, and analogously to a classical neural net, trained these weights to find optimal (or high-quality) values for the weights to minimize a loss function from training data.

Here, I consider a more general architecture, in which instead of each gate being parameterized by a scalar θ, each gate is allowed to be any quantum gate at all. This allows the model to better fit the data, while also reducing the amount of arbitrary choices one needs to make in designing the architecture. In this model, one simply decides where the quantum gates go, without needing to also specify a 1-dimensional parameterization for each gate (and decide which unitaries to cover with this parameterization).

Unfortunately, it turns out this approach is doomed to failure. **This paper** outlines a fundamental problem with quantum neural networks in this style. For sufficiently large quantum neural networks (which in fact, don't need to be very large at all), the gradient of our loss function will tend to be very close to 0. This causes gradient descent to make very little progress, and using this method to train a neural network fails to yield decent results even on easy training sets.

Still, I thought the technique I used to compute this efficiently was interesting, and wanted to document my work here before forgetting about it.

2 Quantum Circuit Architectures

First, we will rigorously define what we mean by an architecture. For our purposes, an architecture is an ordered list of quantum gates, where we only specify which qubits each gate applies to, but not what the gates themselves actually are. An architecture looks like a quantum circuit with none of the unitary gate boxes "filled in".

More formally, we define a *n-qubit Quantum Circuit Architecture* (n-qubit QCA for short) to be a tuple $\mathcal{A} = (A_1, \ldots, A_L)$ of subsets of $\{1, \ldots, n\}$; each A_i represents the set of qubits the corresponding quantum gate operates on. The order of this tuple is important; in the corresponding quantum circuit, the gate for A_1 will be applied first, then A_2, and so on.

Given a QCA, $\mathcal{A} = (A_1, \ldots, A_L)$, we define an *assignment* for $\mathcal{A}$ to be a choice of L unitary matrices $\mathcal{U} = (U_1, \ldots, U_L)$, such that each U_i acts on a space of size $2^{|A_i|}$ (i.e., is in $U(2^{|A_i|})$. An assignment transforms our QCA into a real quantum circuit in the obvious way: for each placeholder A_i, one fills it in with the $|A_i|$-qubit quantum gate defined by U_i, with the gates being applied in tuple order. Each U_i can be tensored with identity matrices on the qubits it does not operate on, in order to obtain a larger unitary $\widehat{U_i} \in U(2^n)$ describing its effect on the entire set of n qubits. Such a tensor product is cumbersome to write (after all, the qubits of U_i may not even be contiguous), so we will hide behind this notation, and simply use $\widehat{U_i}$ to refer to this larger unitary. Thus, the behaviour of a QCA $\mathcal{A}$ with an assigment $\mathcal{U}$ can be described by the unitary $U := \widehat{U_1} \cdots \widehat{U_L}$.

We will focus on the problem of binary classification: we have a set of n-bit strings, each labelled as true or false, and we wish to train a QCA that can classify them as accurately as possible. We will ignore the problem of coming up with a suitable QCA in the first place, and assume that we have both a QCA $\mathcal{A} = (A_1, \ldots, A_L)$, and that we also have decided on a measurement for taking the output from the QCA and deciding whether it's a true or false output. Any such measurement amounts to choosing orthogonal vectors $|\varphi_T\rangle$ and $|\varphi_F\rangle$, measuring the output state $|\varphi\rangle$ from our QCA, and outputting "True" with probability $\langle\varphi_T|\varphi\rangle$ and "False" with probability $\langle\varphi_F|\varphi\rangle$.

Thus, if we have an assignment $\mathcal{U}$ for our QCA, for a particular input $|\psi\rangle$ (say, with truth value "True"), our chance of being wrong is:

$$|\langle\varphi_F|U|\psi\rangle|^2 = |\langle\varphi_F|\widehat{U_1}\cdots\widehat{U_n}|\psi\rangle|^2$$

We wish to minimize this error chance (or perhaps, its mean-squared error taken over all or some of the training data). To this extent, we wish to use gradient descent, with the assignment unitaries $U_1, \ldots, U_L$ as the variable, to find an optimal (or at least high-quality) assignment. In order to do so, we wish to compute the gradient of our loss function, with respect to each unitary. Similarly to the case of training classical neural networks, we compute the loss gradient for one specific training sample; for this next part, we will fix an input $|\psi\rangle$, which without loss of generality we assume is a True case.

Let us fix some $i \in \{1, \ldots, L\}$, and look at U_i. For simplicity, let's assume that U_i operates on the first d qubits, i.e., $A_i = \{1, \ldots, d\}$. Now, since we are optimizing over the manifold $U(2^d)$, some care is needed: if we were to compute the best direction to move U_i in and step in that direction, U_i would leave the manifold. Instead, we will parameterize $U(2^d)$ in the following way. From Lie theory, we know that the lie algebra of $U(2^d)$ is $u(2^d)$, the space of skew-hermitian matrices:

$$u(2^d) = \{X \in M_{2^d}(\mathbb{C}) : X = -X^*\}$$

And we know that $\exp : u(2^d) \to U(2^d)$ is a smooth, surjective mapping. Thus, given our current U_i, we can find a matrix $X_i \in u(2^d)$ such that $\exp(X_i) = U_i$; we can then optimize X_i itself instead. Since $u(2^d)$ is itself a vector space, this eliminates the above problem.

And so, we are thinking of our loss function f as a function that takes in X_i, and tells us what error it eventually causes us to have. Let $C_1 := \widehat{U_1}\cdots\widehat{U_{i-1}}$ and $C_2 := \widehat{U_{i+1}}\cdots\widehat{U_L}$ represent

the parts of the circuit before and after U_i, respectively. Then, we have

$$\begin{aligned} f(X_i) &= |\langle\varphi_F|\widehat{U_1}\cdots\widehat{U_n}|\psi\rangle|^2 \\ &= |\langle\varphi_F|C_1(U_i \otimes I)C_2|\psi\rangle|^2 \\ &= |\langle\varphi_F|C_1(\exp(X_i) \otimes I)C_2|\psi\rangle|^2 \end{aligned}$$

And so we need to compute $f'(X_i)$. It will be useful to split f into 3 parts $f_1 \circ f_2 \circ f_3$, where $f_3 = \exp$, $f_1 = |\cdot|^2$, and

$$\begin{aligned} f_2 : U(2^d) &\to \mathbb{C} \\ U &\mapsto \langle\varphi_F|C_1(U \otimes I)C_2|\psi\rangle \end{aligned}$$

Where I here is the 2^{n-d}-row identity matrix, acting on the qubits unused by U_i. Fortunately, f_2 is quite easy to differentiate, as it's a linear function. After some wrestling with notation, its derivative (at every point X in its domain) is:

$$f_2'(X) = \mathrm{Tr}_Y\left[C_2^*|\varphi_F\rangle\langle\psi|C_1^*\right]$$

Where $\mathrm{Tr}_Y = I \otimes \mathrm{Tr}$ is the partial trace operator, where I has 2^d rows. We note 3 computationally important things:

1. $\langle\psi|C_1^*$ is the conjugate transpose of the result of running the circuit on $|\psi\rangle$ until U_i.
2. $C_2^*|\varphi_F\rangle$ is the result of running the circuit in reverse on input φ_F, until U_i.
3. The partial trace of this matrix can be computed efficiently when d is small; in particular, $C_2^*|\varphi_F\rangle\langle\psi|C_1^*$ is a very large matrix, but we do not need to explicitly compute it.

Putting it all together via chain rule, our loss gradient is:

$$\begin{aligned} \frac{\partial f}{\partial X_i} &= f_3'(f_2(U_i)) \circ f_2'(U_i) \circ \exp'(X_i) \\ &= f_3'(f_2(U_i)) \circ \mathrm{Tr}_Y\left[C_2^*|\varphi_F\rangle\langle\psi|C_1^*\right] \circ \exp'(X_i) \end{aligned}$$

We can average this gradient over several training samples, and then move X_i in that direction, obtaining a new matrix X_i' and a new corresponding quantum gate U_i. We can do this for all gates in our assignment, and iterate the process in the usual way for gradient descent.

3 The caveat

Unfortunately, if we actually run this on even a remotely large QCA (say, one with a dozen or so qubits and gates), we'll find it doesn't work at all. It almost won't matter at all how we shape the architecture, nor what problem we're solving. When we actually compute the gradient, we'll get something extremely close to 0, and gradient descent will not converge in any reasonable amount of time. Even scaling up the gradient to a larger value, will just make iteration extremely noisy and it still won't converge.

As it turns out, this was already a well understood problem in quantum information theory. **This paper** proves a very strong result in this regard, essentially stating that the

probability of the gradient being non-negligible along any particular direction is exponentially small in the number of qubits in the circuit. This effectively rules out the entire approach I used in this paper.

Still, I feel that the technique here has some use. It does leverage some nice technical conveniences to compute the gradient efficiently. If nothing else, this helps illustrate the vanishing gradient problem for QCAs, as well as give some insight into the scale at which it really becomes a problem in practice. Additionally, there is plenty of room for the above to be augmented with various approaches to try to address these issues. For instance, the paper this was inspired by involves choosing 1-parameter families of quantum gates for each slot in the QCA; in the case of their example for solving the MNIST problem, this was enough for them to get at least some modest progress on that particular problem. This is something that this technqiue could potentially generlize; we could restrict the space $U(n)$ for one of our gates, to instead be some submanifold of $U(n)$, and treat that as our domain.