Quantum Code Generation with Conditional RNNs

Yoav Rabinovich
Minerva Schools at KGI

April 18, 2020

Quantum computing aims to harness the dynamics governing fundamental particles to break theoretical limits of computational efficiency, and great strides have been made recently in scaling up quantum processors and reducing noise in their operation. However, hardware is only half the battle. Coding quantum algorithms is difficult and non-intuitive, typically requiring expert knowledge. My project tries to address this problem by automating quantum coding using recurrent neural networks, whose performance in sequential prediction problems has also been advancing rapidly in recent years.

Figure 1: *Google's "Sycamore" Quantum Processor, used in their quantum supremacy experiment, contains only 54 quantum bits and needs to be kept in a dark room at 4 Kelvin (close to -450 degrees F) in order to minimize noise during computation.*

1 Quantum Computing

Computers are brilliant, but they're not all powerful. While modern computers can perform trillions of calculations per second, some problems are too complex to be solved efficiently using even our best algorithms. However, in the 1980s scientists envisioned using particles, not transistors, as basic units of computation. Isolated particles behave deviate from the behavior we expect from classical physics, and instead obey different dynamics dictated by the field of QUANTUM MECHANICS. The quantum information stored in particles, it turned out, can open paths to different – and perhaps better – forms of computation. "QUANTUM SUPREMACY" is the term describing the arrival of quantum computers that can implement algorithms that are faster than any classical equivalent. Just last year, a team at Google claimed to have achieved quantum supremacy on a narrow task, although the claim is disputed by IBM and others (Arute et al., 2019).

There exist quantum algorithms that trump any known classical alternative, such as Shor's algorithm for integer factorization (Shor, 2002), which is famous for its ability to overcome encryption protocols on which we rely globally, but even cutting-edge quantum processors today are too small and noisy to be applied to real-world problems. A theory of quantum error-correction and fault-tolerance provides great promise for the future of quantum computers (Gottesman, 2009), but quantum hardware is still at its infancy.

A different obstacle for quantum computing is algorithm design, which today requires an understanding of quantum information theory – which is rare among programmers. However, it turns out it's not that difficult once you take the physics out of it.

1.1 Generalized Probability

Quantum mechanics can be distilled to an extension of PROBABILITY THEORY. Classically, we can describe an object with two possible states, such as a coin, to be "Heads with 0.25 probability and tails with 0.75 probability" for instance, as long as these probabilities are:

1. Each between 0 and 1.
2. All add up to 1.

So we know that if the coin is heads with 0.25 probability and otherwise tails, then it's tails with 0.75 probability.

$$\begin{aligned}\text{Coin's state} &= 0.25H + 0.75T \\ P(H) &= 0.25 \\ P(T) &= 0.75\end{aligned}$$

However, quantum mechanics puts a slight twist on things: instead of a probabilities, states have an analogous property called **PROBABILITY AMPLITUDES**, which are a bit different:

1. Amplitudes have a size up to one: $|\alpha| <= 1$.
2. The sum of the squares of the amplitudes is what adds up to 1: $|\alpha|^2 + |\beta|^2 + \ldots = 1$. These squares therefore behave like the classical probabilities.

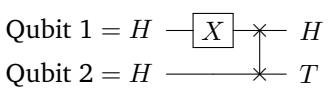
This seemingly small change makes all the difference, because unlike probabilities, amplitudes don't have to be positive. The state of a coin being heads can have amplitude -0.2, or even i! (Although we won't get into that, thank me later). For example, a quantum coin (which I'll name a "quoin") might be in a state like :

$$\begin{aligned}\text{Quoin's state} &= 0.5H + (-0.87)T \\ P(H) &= |0.5|^2 = 0.25 \\ P(T) &= |-0.87|^2 = 0.75\end{aligned}$$

These amplitudes, when added together, can then cancel each other out in complex ways, analogously to amplitudes of waves. This behavior is called **INTERFERENCE**, and it's the basis for unlocking the information processing potential of particles, which we dub **QUBITS**, for "quantum bits".

1.2 Quantum Bits and Circuits

Like a classical bit is an object that can occupy one of two states, a qubit is an object that can have amplitudes for two states. A qubit with amplitude 1 of being in state H behaves just like a classical bit with probability 1 of being in state H, but a qubit with an amplitude -0.5 of being in state H is distinct from a simple randomized bit because we can apply more diverse operations to it that we ever could to a classical bit. These operations are called **GATES** in quantum computing, and stringing together gates applied on a set of qubits, we can program a quantum algorithm, or **QUANTUM CIRCUIT**. For example, the following simple circuit on two qubits first uses a NOT gate (marked as X) to flip the first qubit from 1 to 0, and then uses a swap gate to swap the two, such that the output of the circuit on two H qubits is H, T.



Interference comes into play when we implement gates that behave differently for positive amplitudes than negative ones, enabling greater freedom than classical gates. For example, let's give names to two quoin states, which are both equally uncertain:

$$\begin{aligned}Q_1 &= 0.5H + 0.87T, & P(H) &= 0.25,\ P(T) = 0.75 \\ Q_2 &= 0.5H + (-0.87)T, & P(H) &= 0.25,\ P(T) = 0.75\end{aligned}$$

Then, we can find a single quantum gate that would distinguish between the two, despite their probabilities being indistinguishable!

While most practical circuits are more complex, these circuits can be seen as simply a string of gates, representing operations on one or more qubits. Still, wrangling qubits to do one's bidding is a tough task. However, there are exciting statistical models that can help automate the process by learning from examples.

2 Sequential Prediction

To be able to predict elements in a sequence, we want to model a distribution of probabilities for elements to appear in certain orders. This is called a **LANGUAGE MODEL**, and language is indeed a good example for sequence prediction problems. We could learn to generate text, for example, by learning the probability for each word to appear in a sentence, given the previous words in that sentence. This conditional probability term can be written as:

$$P(\text{word} \mid \text{previous words}) = P(w_i | w_1, w_2, \ldots, w_{i-1})$$

How do we learn this model? We can use a corpus, which is a collection of sentences, and learn by example: Is the word "king" more likely to follow "Falafel" or "Tiger"? How much so? What if the word "delicious" appears before that? With enough examples and a simple correlation calculation, we can infer a language model. But these models can sometimes be too big to be tractable, or too slow to compute. Neural networks have exploded in popularity recently as versatile tools that can solve learning problems efficiently. For sequential prediction problems, the "king" is the **RECURRENT NEURAL NETWORK**.

2.1 Recurrent Neural Networks

RNNs are machine learning models that generate language models through a process of **OPTIMIZATION**. Neural networks learn not by an assumed or imposed correlation calculation, but by trial and error and course correction which arrives at a good procedure. This allows neural networks to reach a good model quickly and efficiently.

Neural networks are made of layers of interconnected nodes, each connection set to an adjustable weight. With every example, the model compares its prediction with the correct results, and the worse it has done, the harsher it will correct its weights to achieve a closer prediction. In this manner, neural networks train themselves through **GRADIENT DESCENT**, like a ball rolling down a hill: if the prediction is bad, we look around for the direction we can move the weights to lower the error, and go there. Eventually we'll reach an optimal solution. How do we find this direction? We take a **DERIVATIVE** of our error calculation. In calculus, derivatives are used to find slopes, which will point us in directions of descent.

RNNs are a neural network designed to predict sequences. It uses only one group of weights to predict any element in a sequence: these weights are applied "recursively" across the sequence to train and to predict. By doing that, they store a "memory" of past elements which can be used to generate the language model. Their structure is briefly explained in Figure 2.

But what if it's not just the word order we care about, but also what they're trying to say?

2.2 Introducing Conditions

In my project, I trained my network not just to remember in what order gates appear in circuits, but also to churn out circuits that hit the mark: given a target output of a circuit, it should generate a circuit that actually gets there. Networks like this are called conditional RNNs, since they introduce external context as **CONDITIONS**.

A good example of this is image captioning models, which train using pairs of pictures and captions describing them. Then, during prediction, the models are exposed to a target picture, and manage to describe it in words on their own. This is because the network has learn the conditional probability of each word given not only the previous words, but also the paired picture:

$$\begin{aligned} &P(\text{element} \mid \text{previous elements}, \text{conditions}) \\ &= P(w_i | w_1, w_2, \ldots, w_{i-1}, \theta_1, \theta_2, \ldots, \theta_N) \end{aligned}$$

With these tools under our belt, we can approach the problem at hand.

3 Project

I tested various ways to train conditional RNNs to generate quantum circuits, conditioned on a target output in various ways, and compared their effectiveness. The model diagram for my experiments is presented in Figure 3, but I won't discuss the details here. For those wondering, no, it didn't work. A discussion of causes and excuses can be found in my project, which is available through the following URL: `git.io/JvHGO`.

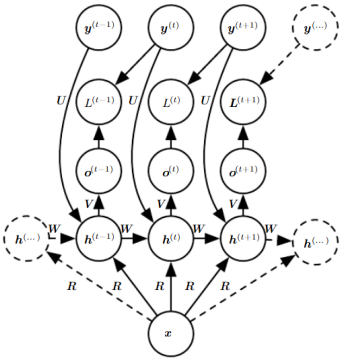


Figure 2: *A graph depicting a conditional RNN. The memory of the RNN (**h**) carries information through the sequence prediction process. At each time-step, it's fed static condition information (**x**), and produces a prediction (**o**). This prediction is then compared to the correct element (**y**) to calculate the error (**L**), and the correct prediction is fed into the next time-step to continue the sequence. From Goodfellow, Bengio, and Courville, 2016.*

References

Arute, Frank et al. (2019). "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779, pp. 505–510. ISSN: 14764687. DOI: 10.1038/s41586-019-1666-5. arXiv: 1911.00577.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

Gottesman, Daniel (2009). "An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation". In: pp. 13–58. arXiv: 0904.2557. URL: http://arxiv.org/abs/0904.2557.

Shor, P.W. (2002). "Algorithms for quantum computation: discrete logarithms and factoring". In: Institute of Electrical and Electronics Engineers (IEEE), pp. 124–134. DOI: 10.1109/sfcs.1994.365700.

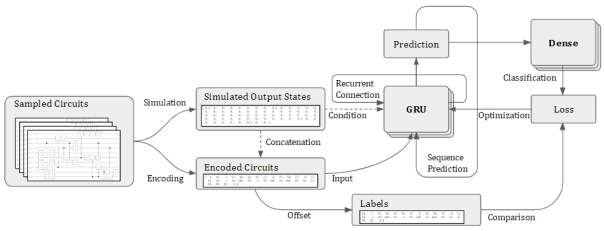


Figure 3: *Model architecture used in my project. In summary, a data-set of random circuits is drawn and encoded, and their output is simulated. The output is then fed as a condition, either directly to the memory unit of the RNN at the first time-step, or appended to the input of each iteration.*

#connotation: Targeting a layman audience, I anticipate anxiety approaching heavy technical topics. Therefore, I employ casual language, toy problems, simple examples, helpful emphasis markers and attractive visuals to keep the reader relaxed and engaged.