# CPEN 400Q Project Report

Instructor:
Dr. Olivia Di Matteo

Submitted by:
Jason Bai (95612412)
Eva Wang (17665191)
Jane Shi (37998283)
Bobby Zou (65563090)

Thursday, April. 13, 2023

# 1 Introduction

Recurrent Neural Networks (RNN) have been a popular machine learning model for many different applications in recent years. RNNs have slowly become one of the most commonly used and studied algorithms to achieve deep learning goals. Applications of RNN include language translation, robotics control, time-series prediction, etc. While the classical model delivers prominent results and many resources have been put to optimize it, it is still relatively unknown what would be the next major advancement of such technology. With the recent breakthrough of quantum computing technologies, it is more apparent that Quantum Computing is no longer some arbitrary equation on the blackboards of theoretical physicists but a technology that is becoming accessible to all backgrounds. With millions of dollars poured into companies like Xanadu, IBM, and D-Wave, many novices are slowly getting the gist of it.

This paper examines the concept, method and implementation of a recent research paper published by a group of researchers from Japan on the topic of Quantum Recurrent Neural Network (QRNN), where it demonstrated an implementation of the classical RNN through a 6-qubit quantum computer, successfully mimicking the behavior of RNN quantumly. The paper served as an important stepping stone towards the overlap between classical neural network and quantum computing technologies and is a good starting point for anyone who wants to give the qubits a try.

# 2 Theory

## 2.1 Significance of Learning Temporal Data

Currently, some estimation shows that around 2.5 quintillion bytes of data [1] is generated by mankind everyday. Among this extremely large data set, many contain the dimension of time, and hence are called “time-series” data or “temporal” data. Some quick examples of such data series include weather data, stock prices, etc. The use of temporal datasets allow us to study and predict trends in the future by analyzing the information of the past. These predictions aid human activities and help us make better decisions and boost productivity significantly. With more and more data generated everyday, it is not hard to understand that feeding more and more data to a well developed model should allow it to make better and more accurate predictions as we go.

## 2.2 Using Quantum Recurrent Neural Network

The reason for spending time migrating the classical RNN into a quantum computer does not seem to be clear at first glance. The paper specifically mentioned that they have no data or evidence that the proposed implementation on an actual quantum computer would yield any speedup, effectively achieving “quantum advantage”. However, it is worth noting that a concept called “Quantum Reservoir Computing (QRC)”, which has a similar structure with the proposed QRNN discussed in this paper, was proven to show promising “quantum advantage” to its classical counterpart.

As mentioned above, though the concept of QRNN is probably still far away from commercial use, the study has become one of the fundamental cornerstones of bridging a well-developed neural network and the use of quantum computers. It opens possibilities that all current major concepts of machine learning algorithms can get translated into their quantum counterparts. Although this does not seem exciting, QRNNs may easily become the next area of research for many, since the advancement of existing models have become more heavily reliant on manipulating data size, rather than finding novel designs.

# 3 Derivation of Key Results in Software

The circuit for each iteration of the neural network is composed of two parts: the encoding step and the evolution step (Figure 3-1 below). The qubits are split into two groups. Group B will go through the encoding step and it will be reset to zero after each iteration. Then all qubits (Group A and Group B) will be fed into the evolution step.

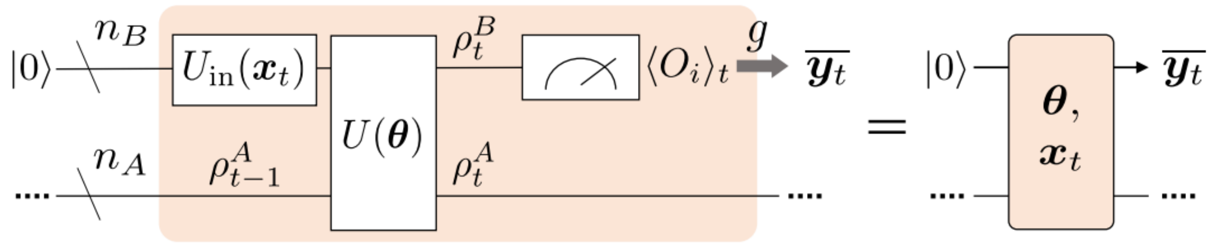

$|0\rangle$
$n_B$
$U_{\text{in}}(\boldsymbol{x}_t)$
$\rho_t^B$
$\langle O_i \rangle_t$
$g$
$\overline{\boldsymbol{y}_t}$
$n_A$
$\rho_{t-1}^A$
$U(\boldsymbol{\theta})$
$\rho_t^A$
$=$
$|0\rangle$
$\boldsymbol{\theta}$, $\boldsymbol{x}_t$
$\overline{\boldsymbol{y}_t}$

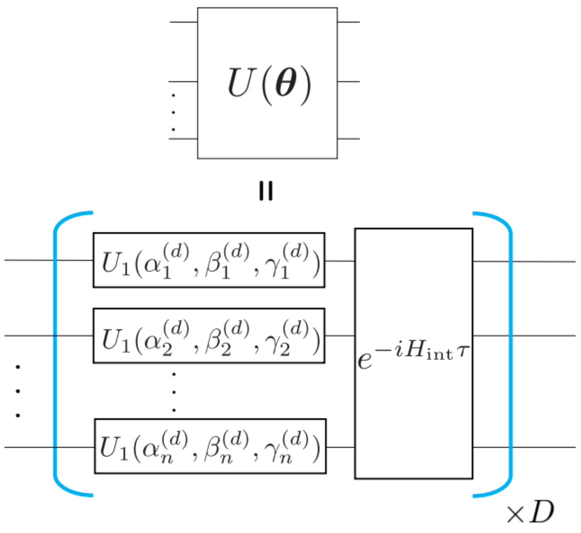
*Figure 3-1: Structure of each iteration.*

The encoding step $U_{in}$ is just an RY rotation of the arccosine of the iteration’s input($x_t$), applied to the Group B qubits (Figure 3-2 below).

$$U_{\text{in}}(x_t) = R_y(\arccos x_t)$$

*Figure 3-2: RY Rotation used in encoding step.*

The evolution step *U(θ)* consists of a process that is repeated *D* times, which is specified as having the value of *D=3* in the paper (Figure 3-3 below).



$U(\boldsymbol{\theta})$
$=$
$U_1(\alpha_1^{(d)}, \beta_1^{(d)}, \gamma_1^{(d)})$
$U_1(\alpha_2^{(d)}, \beta_2^{(d)}, \gamma_2^{(d)})$
$U_1(\alpha_n^{(d)}, \beta_n^{(d)}, \gamma_n^{(d)})$
$e^{-iH_{\text{int}}\tau}$
$\times D$

*Figure 3-3 Structure of the circuit used for each Evolution Step.*

Within this process, the training parameters α, β, γ are first used in $U_1$, which is a combination of RX and RZ rotations (Figure 3-4 below).

$$U_1(\alpha, \beta, \gamma) = R_x(\alpha)R_z(\beta)R_x(\gamma)$$

*Figure 3-4: Series of RX and RZ rotations that have training parameters passed into them (start of evolution step).*

Then, a Hamiltonian $H_{int}$ is constructed as a combination of PauliX's and PauliZ's (Figure 3-5 below). Coefficients $a_j$ and $J_{jk}$ are random values from a uniform distribution on [-1,1] that are fixed during training, and *n* is the total number of qubits($n_A$ + $n_B$ = *n*).

$$H_{\text{int}} = \sum_{j=1}^{n} a_j X_j + \sum_{j=1}^{n} \sum_{k=1}^{j-1} J_{jk} Z_j Z_k$$

*Figure 3-5: Equation for Hamiltonian used in evolution step.*

The Trotterized time-evolution operator for this Hamiltonian is then applied to all qubits (*qml.ApproxTimeEvolution()* abstracts away the calculation of the Hamiltonian being multiplied by negative *i* and evolution time tau(τ), and then raised to the exponential as shown in Figure 3-3).

Finally, the Z-expectation values of all Group B qubits are measured (Figure 3-6 below), which is then averaged and multiplied by training parameter *c* to get the final prediction value of the  current iteration.

As detailed in the paper, all training parameters are initialized to zero (except for *c*, which is set to 1) at the beginning. Also, tau is set to 0.2 because the paper suggests that only a specific range of tau provides an accurate prediction. A smaller value of tau will lead to less past information being passed into the next iteration, while a bigger value of tau will make the Hamiltonian dynamics too complex for rotations to extract information. The paper also did experiments on how the prediction's mean squared error is affected by the value of tau, and found that a value of 0.2 gives the smallest error.

The resulting Group A qubits will be maintained and used together to train the neural network in the next iteration. In order to achieve that, we create an extra device with the same parameters and the same architecture to return the density matrix of the Group A qubits (instead of measuring expectation values). Therefore, we use this density matrix to initialize the Group A qubits in both devices using *qml.QubitDensityMatrix()* so that we can train the QRNN with past information.

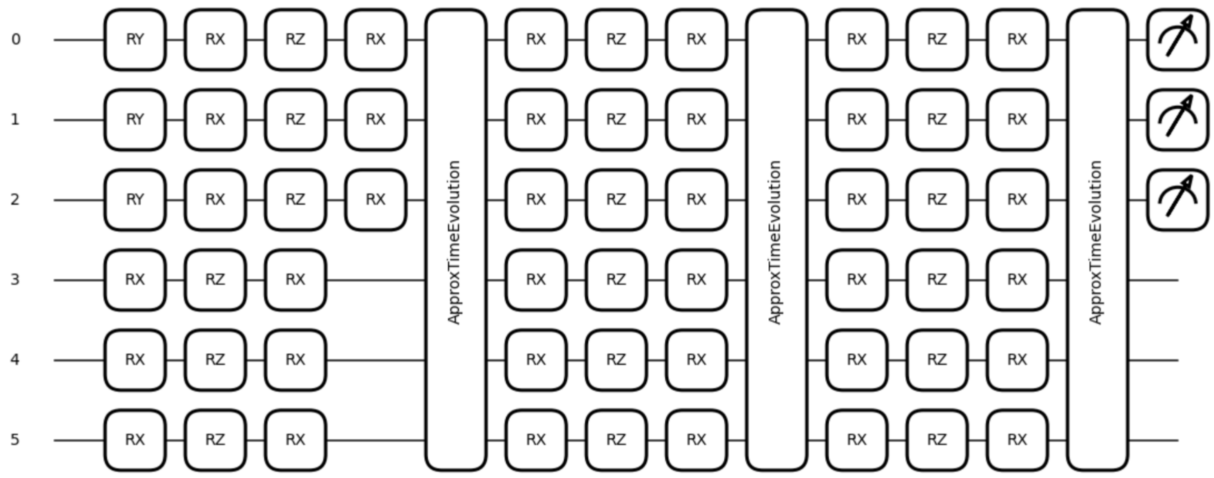
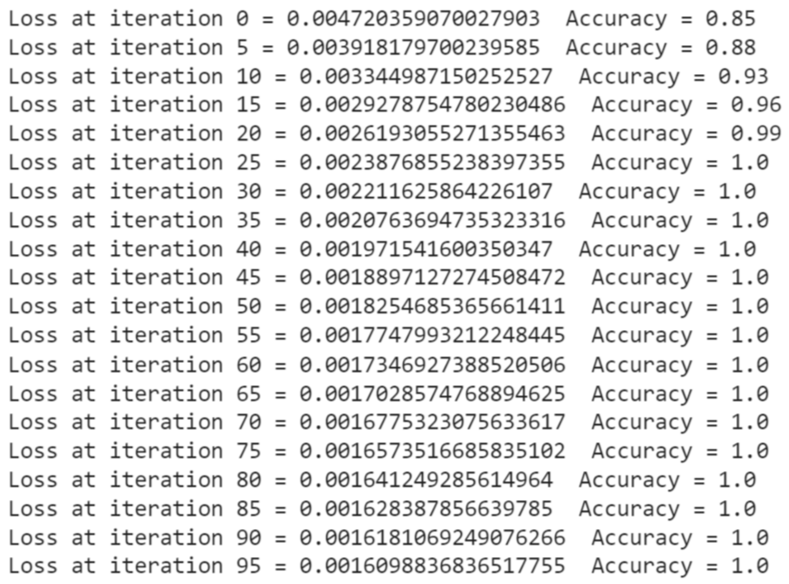


*Figure 3-6: Overall diagram of our software architecture (one iteration).*

## 4 Results of Software Implementation

We compute the accuracy by checking if the difference between the prediction and the true label is less than 0.1. An accuracy of 1.0 indicates that for all data points, the difference is less than 0.1.

After training for 100 iterations on a triangular wave (blue line in Figure 4-1), we get a loss of approximately 0.0016 and an accuracy of 1.0.


```
Loss at iteration 0 = 0.004720359070027903  Accuracy = 0.85
Loss at iteration 5 = 0.003918179700239585  Accuracy = 0.88
Loss at iteration 10 = 0.003344987150252527  Accuracy = 0.93
Loss at iteration 15 = 0.0029278754780230486  Accuracy = 0.96
Loss at iteration 20 = 0.0026193055271355463  Accuracy = 0.99
Loss at iteration 25 = 0.0023876855238397355  Accuracy = 1.0
Loss at iteration 30 = 0.002211625864226107  Accuracy = 1.0
Loss at iteration 35 = 0.0020763694735323316  Accuracy = 1.0
Loss at iteration 40 = 0.001971541600350347  Accuracy = 1.0
Loss at iteration 45 = 0.0018897127274508472  Accuracy = 1.0
Loss at iteration 50 = 0.0018254685365661411  Accuracy = 1.0
Loss at iteration 55 = 0.0017747993212248445  Accuracy = 1.0
Loss at iteration 60 = 0.0017346927388520506  Accuracy = 1.0
Loss at iteration 65 = 0.0017028574768894625  Accuracy = 1.0
Loss at iteration 70 = 0.0016775323075633617  Accuracy = 1.0
Loss at iteration 75 = 0.0016573516685835102  Accuracy = 1.0
Loss at iteration 80 = 0.001641249285614964  Accuracy = 1.0
Loss at iteration 85 = 0.001628387856639785  Accuracy = 1.0
Loss at iteration 90 = 0.0016181069249076266  Accuracy = 1.0
Loss at iteration 95 = 0.0016098836836517755  Accuracy = 1.0
```

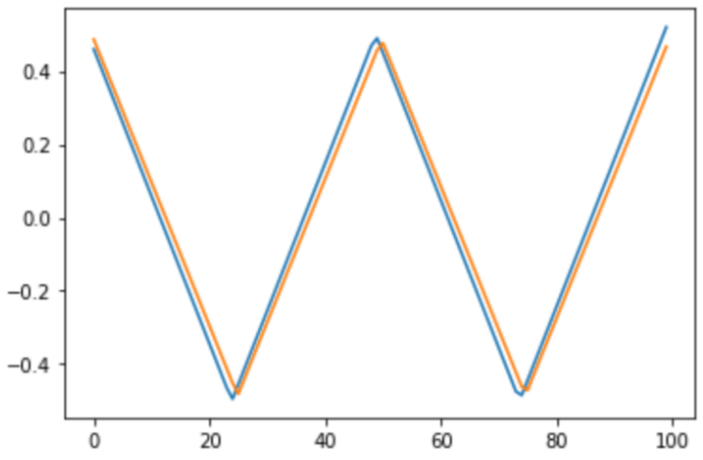
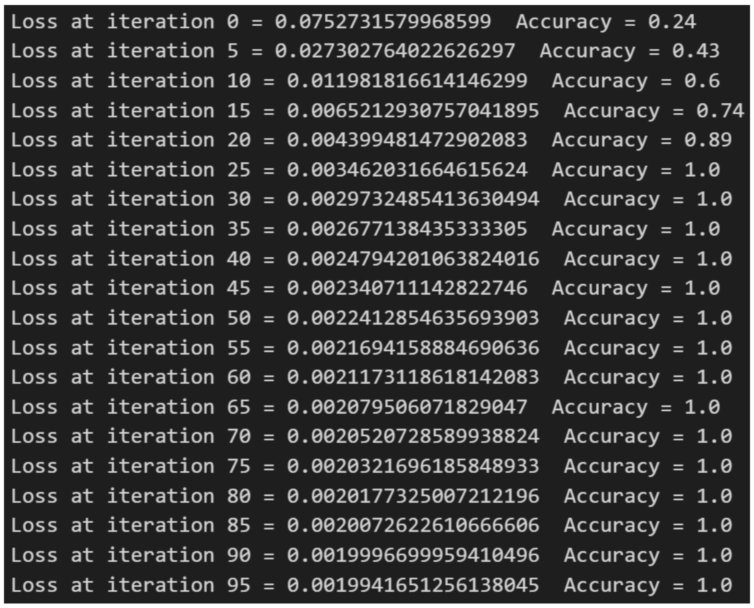


*Figure 4-1: Training loss and results after running 100 iterations on the triangular wave.*

After training for 100 iterations on a cosine wave (blue line in Figure 4-2), we get a loss of approximately 0.002 and an accuracy of 1.0.



```
Loss at iteration 0 = 0.0752731579968599  Accuracy = 0.24
Loss at iteration 5 = 0.027302764022626297  Accuracy = 0.43
Loss at iteration 10 = 0.011981816614146299  Accuracy = 0.6
Loss at iteration 15 = 0.0065212930757041895  Accuracy = 0.74
Loss at iteration 20 = 0.004399481472902083  Accuracy = 0.89
Loss at iteration 25 = 0.003462031664615624  Accuracy = 1.0
Loss at iteration 30 = 0.0029732485413630494  Accuracy = 1.0
Loss at iteration 35 = 0.002677138435333305  Accuracy = 1.0
Loss at iteration 40 = 0.0024794201063824016  Accuracy = 1.0
Loss at iteration 45 = 0.002340711142822746  Accuracy = 1.0
Loss at iteration 50 = 0.0022412854635693903  Accuracy = 1.0
Loss at iteration 55 = 0.0021694158884690636  Accuracy = 1.0
Loss at iteration 60 = 0.0021173118618142083  Accuracy = 1.0
Loss at iteration 65 = 0.002079506071829047  Accuracy = 1.0
Loss at iteration 70 = 0.0020520728589938824  Accuracy = 1.0
Loss at iteration 75 = 0.0020321696185848933  Accuracy = 1.0
Loss at iteration 80 = 0.0020177325007212196  Accuracy = 1.0
Loss at iteration 85 = 0.0020072622610666606  Accuracy = 1.0
Loss at iteration 90 = 0.0019996699959410496  Accuracy = 1.0
Loss at iteration 95 = 0.0019941651256138045  Accuracy = 1.0
```

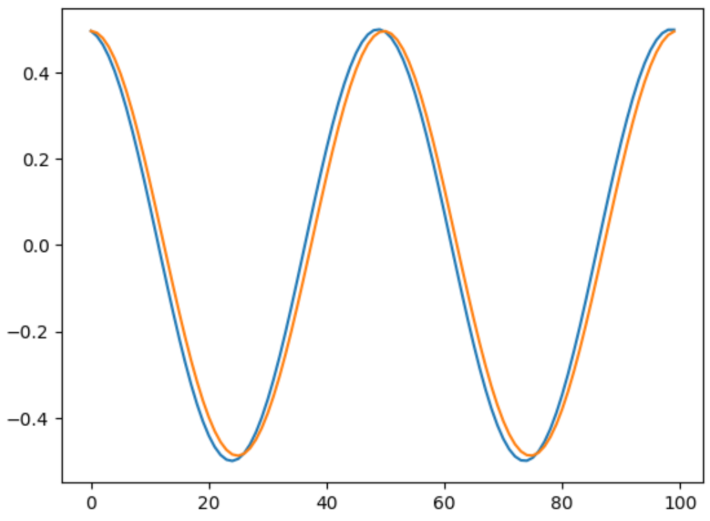


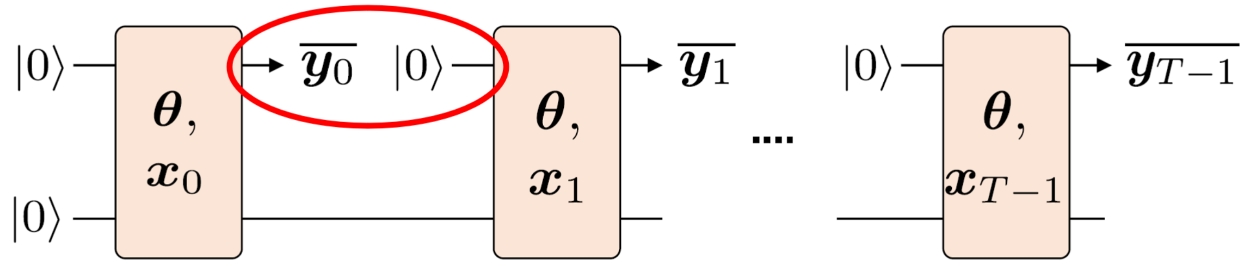
*Figure 4-2: Training loss and results after running 100 iterations on the cosine wave.*

## 5 Challenges affecting reproducibility

Ultimately, the results of the research paper were completely reproducible. During development, a few challenges affecting reproducibility appeared.

### 5.1 Challenge #1: Measure and Reset

After each iteration of the neural network, all the qubits in Group B (not the ones that save past information) must be reset to |0> (Figure 5-1 below).



$|0\rangle$
$|0\rangle$
$\boldsymbol{\theta}, \boldsymbol{x}_0$
$\overline{\boldsymbol{y}_0}$
$|0\rangle$
$\boldsymbol{\theta}, \boldsymbol{x}_1$
$\overline{\boldsymbol{y}_1}$
....
$|0\rangle$
$\boldsymbol{\theta}, \boldsymbol{x}_{T-1}$
$\overline{\boldsymbol{y}_{T-1}}$

*Figure 5-1: Measure and Reset of Group B qubits after each iteration.*

However, the rest of the qubits, which are part of Group A, must maintain their values and input them into the next iteration. The problem is that past information cannot be stored, since after each iteration, Pennylane will automatically reset all qubits to zeros.

Our initial idea was to create an extra set of 3 qubits for each iteration. Each of these sets would have their qubits just set to |0> before they are used in their corresponding iteration. This idea was discarded due to its heavy computational cost, because reproducing 100 iterations of training as detailed in the research paper would require (3)(100) = 300 extra qubits!

As recommended from Piazza discussions (@108), we looked into using the "default.mixed" device with the function *qml.density_matrix()* for returning the reduced density matrix that traces the Group A qubits only (and then using *qml.QubitDensityMatrix()* to re-initialize the state of the Group A qubits in the next iteration).

We settled on instantiating a total of 2 devices. Both devices are running the exact same circuit and functionality, but one device only measures the result of each iteration, whereas the other device only returns the density matrix of the Group A qubits, in order to re-initialize the next iteration correctly for both devices.

## 5.2 Challenge #2: Computing Accuracy

Similar to the demo from Lecture 8, we have a helper function that computes accuracy for using *qml.GradientDescentOptimizer()*. However, our predictions are numerical values (not categorical). This led to the realization that predictions should not be penalized heavily if they are close to true values, but do not exactly match.

Therefore, we modified our accuracy computation to instead use the absolute difference between predicted values and true values.

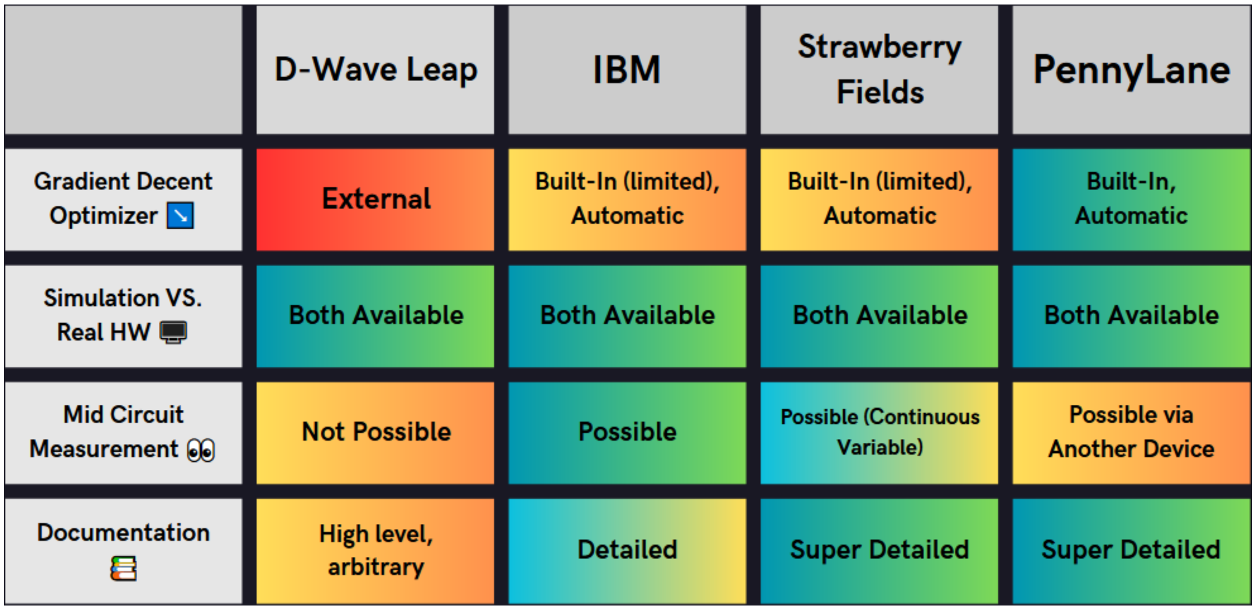
## 5.3 Challenge #3: Using a Hamiltonian

When the code was being developed, we had not yet learned about Hamiltonians in class. After discussions on Piazza (@80), we learned to use *qml.Hamiltonian*() to form a Hamiltonian and pass it to *qml.ApproxTimeEvolution()* to implement the evolution step of each iteration.

# 6 Choice of Programming Framework

Among all of the quantum programming frameworks that can potentially be used, we have chosen four mainstream frameworks to compare, these include: D-Wave Leap, IBM

Quantum, Xanadu Strawberry Fields, and Xanadu PennyLane. We have chosen to compare them via four main criterias, shown below:



| | D-Wave Leap | IBM | Strawberry Fields | PennyLane |
|---|---|---|---|---|
| Gradient Decent Optimizer ↘️ | External | Built-In (limited), Automatic | Built-In (limited), Automatic | Built-In, Automatic |
| Simulation VS. Real HW 🖥️ | Both Available | Both Available | Both Available | Both Available |
| Mid Circuit Measurement 👀 | Not Possible | Possible | Possible (Continuous Variable) | Possible via Another Device |
| Documentation 📚 | High level, arbitrary | Detailed | Super Detailed | Super Detailed |

*Figure 6-1: Quantum Programming Framework Comparison*

Though quantum frameworks such as IBM Quantum and Strawberry fields could be used, it seems to us that PennyLane makes the most sense as it contains a built-in fully automatic gradient descent optimizer, coupled with its detailed documentation. Furthurmore, our team members' knowledge was built mostly on PennyLane. Overall, the team has successfully implemented the measure and reset functionality (mid circuit measurement) within Pennylane, which was understood as a roadblock at first, without having to switch to another framework.

# 7 Conclusion

In this research, we have taken a deep dive into the newly proposed idea of a Quantum Recurrent Neural Network, effectively reproducing the results achieved by the paper via the use of PennyLane by Xanadu. We investigated the quality of design, and were able to successfully understand and realize both the excitement and limitations of the proposed technology. Below is our view on the limitations and open questions of QRNN and the future directions of exploration.

As mentioned above, QRNN has not shown any quantum advantage yet compared to its classical counterpart. Unfortunately, due to the lack of description of the classical RNN framework in the paper and time constraints, we were not able to properly collect enough data to determine any kinds of speedup. However, since the design of QRC, which has a very similar structure as the QRNN, has displayed promising quantum advantage, it is obvious that this is an open question that is yet to be answered and worth exploring.

Throughout our implementation, we have encountered and realized some architectural challenges of QRNN. With its requirement of mid circuit measurement and reset, this concept is very hard to operate alone when given to an annealer type quantum computer. Furthermore, the need to keep the state for 0.2 seconds (via the Hamiltonian) may be challenging for current quantum hardware with acceptable fidelity rate.

Throughout the implementation, we have noticed that initializing trainable parameters of alpha, beta, and gamma to all zeros have always yielded a better training loss and accuracy than starting with randomized numbers. It is still unknown why this would be the case and hence should be further explored. Lastly, the value of 0.2 of tau was empirically determined by using values between 0 and 10 to train the QRNN on the cosine wave, with a value of 0.2 yielding the lowest mean squared error. Hence, we suspect that for different kinds of temporal data, setting tau as a trainable parameter and letting the algorithm determine which tau value is the best is more appropriate. But in reality, how well can QRNNs do with these open questions answered? Only time will tell.

## 8 References

[1] “Infographic: How much data is produced every day?,” CloudTweaks, Dec. 13, 2021. https://cloudtweaks.com/2015/03/how-much-data-is-produced-every-day/

[2] Y. Takaki, K. Mitarai, M. Negoro, K. Fujii, and M. Kitagawa, “Learning temporal data with a variational quantum recurrent neural network,” Physical Review, vol. 103, no. 5, May 2021, doi: 10.1103/physreva.103.052414.