School of Computer Science
Faculty of Science and Engineering
University Of Nottingham
Malaysia

Review of Mr. Ping's Final Year Dissertation Report on:

- Hybrid Quantum Convolutional Neural Networks –

Student's Name : Hoo Kai Ping
Student Number : 20411482
Supervisor's Name : Tan Chye Cheah
Reviewer's Name : Mahmoud Yasser Sallam

REVIEW SUBMITTED IN PARTIAL FULFILLMENT OF THE INTERNSHIP REQUIREMENTS SET BY THE ASSIGNED SUPERVISOR THE UNIVERSITY OF NOTTINGHAM

Preamble

This review has been conducted with direct supervision and support from the Dissertation's supervisor, Dr. Tan Chye Cheah. The review is intended to verify the findings of the original report and detail how quantum image encoding functions, notably in a Hybrid Quantum Convolutional Neural Network (HQCNN) model.

I would like to thank Dr. Tan for being ever so patient and resourceful, providing me with all the relevant details I needed to accomplish this review and allowing me the time to do so. I would also like to thank Mr. Hoo Kai Ping, the author of the Dissertation Report for assisting me in completing this review by supplying me with the necessary files and verifying the modification done to the code while going through the different iterations, finally, I would like to extend my gratitude to Dr. Yasir Hafeez for lecturing us on COMP1032: Fundamentals of AI, which provided me with an essential understanding on CNNs, MLP heads & ANNs as a whole.

A reminder to the reader that this review is intended to be digestible by undergraduate Computer Science students and therefore various explanations, and definitions will be included to provide an enhanced reading experienced to those unfamiliar with the topic.

Abstract

This review will detail the theoretical aspects of the application of quantum image-encoding and the practical modifications made to the Hybrid Quantum Convolutional Neural Networks (HQCNNs) for each architecture using code snippets as reference points and examples.

This Review is made with the assumption that the original dissertation report has been read and therefore will limit the repetition of information, however, references will be made occasionally in order to support points or provide comparisons.

Furthermore, we will explore various points of interest in the original report with a focus on pointing out improvements and notable errors or if expected issues arose.

Contents

Chapter 1

Reflection

The review started on the 5th of June 2025 when Dr. Tan granted myself access to the original dissertation report, source code and relevant material regarding quantum computation and quantum information, between the 5th of June and the 17th of June, progress related to the glossary started which was also essential as it provided me with the time needed to comprehend topics related to the original report.

The first setback took place between the 19th of June until the 2nd week of July, this was caused by a medical reason that required hospitalization which hindered me from working on the internship, however once I received my medical discharge, progress rapidly picked back up with constant updates being sent to Dr. Tan related to replicating the training runs, resolving various dependency errors and updating the `README.md` file to become more comprehensive and assist users with resolving any issues that may arise in the future.

Initially, the plan was to compare Version 4's result from my runs with Mr. Ping's graphs in the report, this was later fleshed out to include all versions and to run the comparison directly in Tensorboard with the `.0` files to provide a more uniform method of discerning between runs.

Between the 3rd week of July until the 3rd week of August, progress remained steady with details being added to the `README.md` file which included troubleshooting methods relating to resolving dependency issues using `pip` and running TensorBoard to view all the runs correctly, meanwhile, the number of runs per version was set at three, this was done as a safety measure but later proved to be trivial as the runs were virtually indiscernible barring the time taken to process them which was attributed to other programs running at the time.

As the runs were underway, progress related to the documentation of the source code and version differences remained steady, where I got the idea of compartmentalizing the source code into various segments from a Cybersecurity seminar which was recommended by Dr. Tan.

When attempting to perform the comparison between the runs, it was uncovered that the reference runs for Version 1 and Version 2 were performed with a different `batch_size` and number of `steps`, this was found out quite late into the review so a decision was made use the information in Chapter 4.1.2, notably relating to Version 1's and Version 2 of the HQCNN as supporting notes when performing the result comparison.

Upon reflecting, there are various goals that I would have wanted to accomplish which include:

- Performing more runs while increasing the epoch to review the results of the HQCNN model under prolonged training sessions.
- Utilizing other datasets such as CIFAR-10, which was a goal in the original dissertation that was abandoned due to complications.
- Researching and implementing improvements onto the HQCNN model created by Mr. Ping.

These constitute some of the goals that crossed my mind once I reached the final stretch of the internship which meant that I had to focus on finalizing and refining the review on which I was working.

Chapter 2

Code Explanation

This chapter details the source code's functions and how each module operates in tandem to operate the HQCNN model, and for the purposes of this analysis the source code will be separated into five segments, these segments detail the precomputation and real-time quantum encoding process, the operations of the Quantum Branch/Pipeline and the classification of the feature map by the CNN, this is done in order to clearly differentiate between each module and allow for a better understanding of their inner workings.

It is of importance to note that the FYP provides further insight, notably in Chapters 3.1 and 3.2 which detail the operations of the CNN and HQCNN in greater detail.

2.1 Precomputation

The 'Precomputation' segment details the process which creates the pre-computed quantum patch states that allows for the training of the HQCNN model without the need to compute the quantum-encoded patch states in real-time, note that it doesn't substitute ENEQR_pennylane.py and in fact calls it to achieve its function.

- `precompute_eneqr.py`:

  This module downloads the MNIST dataset and from each image dissects it and creates a 14x14 grid which consists of 196 non-overlapping patches, it is of note that each patch is 2x2 pixels in size, this is set by the `patch_size` and `stride` variable in the `__init__()` call.

  The wires/qubit used per each patch is calculated by the equation $wires = 2 * ceil(log_2(patch\_size)) + q + 1$ where for Version 1, the values for

`patch_size` and `q` are 2 respectively, which after calculating it results in the number of wires/qubits to become 5, note that this is also done in `ENEQR_pennylane.py`.

Each 2x2 patch is then encoded using the ENEQR encoder (explored in Segment 3) after having its respective pixel value set between [0..1], after which the quantum expectation vectors are finally saved into a `.h5` file which has the training and testing datasets written each with the expectation vector and corresponding MNIST label.

Note that altering the 'gray-value' qubits `q` will necessitate the regeneration of the `.h5` file since if the HQCNN model is ran with a precomputed dataset intended for a different amount of wires various errors can pop up, which can be summed up to dimensional mismatch and run-time errors that take place when the SVQC attempts to apply rotation gates to an incorrect number of wires, this is shown in Figure 2.1.1.

Figure 2.1.1: Screenshot demonstrating the errors raised by the mismatch of qubits.

- `hdf5_dataset.py`:

  This module reads the data from the `.h5` file which consists of the precomputed quantum states and labels and wraps it to enable compatibility with PyTorch. Note that it is purposely coded in a "lazily" fashion in order to mitigate file-handling and serialization issues.

2.2 Real-Time Quantum Encoding

This segment pertains to the process that is undertaken if the HQCNN is intentionally ran without utilizing a pre-computed dataset, note that this process is entirely optional and is not taken into account for the purposes of this review.

- `loaddata.py`:

  This module downloads the MNIST dataset and loads it into memory, furthermore it converts the images to tensors and then normalizes the pixel values between [0...1] in preparation for ENEQR encoding.

- `LRUCache.py`:

  The LRU "Least Recently Used" Cache acts as a temporary cache (albeit with a large capacity at 50,000 items) that stored quantum computed patch outputs in case an identical patch, in case it does, the cache pulls up the stored result and reuses it to cut down on computational workload and decrease the time taken to process the model.

2.3 Quantum Branch

The Quantum Branch is notably the most important as it pertains to the designation of the ENEQR quantum-image encoding technique and the SVQC which include the implementation of Hadamard, CNOT and CRY gates to achieve quantum entanglement patterns and super-positioning between the qubits.

- `ENEQR_pennylane.py`:

  This module contains the core implementation of the ENEQR module, which allows the conversion of a classical grayscale 2D image patch into a quantum state, this is done through the calculation of the total number of wires/qubits through the equation: $wires = 2n + q + 1$, where `n` is the position qubit pairs calculated using $n = ceil(log_2(patch\_size))$. Initialization of the PennyLane QNode is done

though `default.qubit` and the construction of the TorchLayer wraps the QNode in order to allow compatibility with PyTorch.

The QNode applies the Hadamard gates to all the position qubits in order to establish a state of equal superposition, this is done to allow the circuit to identify all the possible pixel locations in parallel, this is followed by an iterative call done through a nested for loop which goes through every pixel and encodes the grayscale intensity by normalizing the grayscale intensity value into an integer value where if it is |0⟩ (Black) it is skipped since it is already in the ground state |0⟩.

The next step involves acquiring the binary values of the pixel locations and the normalized 'gray-value' qubits followed by the conditional activation of the auxiliary qubit through the usage of an MCX Gate to target a designated spatial location, finally once the auxiliary qubit is activated, the CNOT gates are used to target the 'gray-value' qubits and designate the pixel's intensity value based on if the value of the i-th 'gray-value' qubit is 1. Finally, the auxiliary qubit is reset to its respective ground state |0⟩ to allow for the processing of the next pixel without carrying over any residual information and the Pauli-Z expectation value is returned for every patch which is equal to the length of the qubits.

- `quantum_circuit.py`:

Once the quantum state is acquired from the ENEQR module, it is ran through the SVQC which starts with generating the layout for the qubit adjacent pairs to allow for the future implementation of qubit entanglement, after which each qubit is rotated by RX & RZ gates and then CRY gates are utilized for the qubit adjacent pairs to establish bidirectional entanglement, note that this is done by using 2 CRY gates where each gate ensures entanglement in one direction.

1. Parameters 0-3 are used to apply an RX & RZ gate to qubit 1 and qubit 2.

2. Parameter 4 is used to apply a CRY gate that rotates qubit 2 based on the state of qubit 1 which allows for entanglement to occur where qubit 1 affects qubit 2.
3. Parameters 5-8 are used to apply another RX & RZ gate to qubit 1 and qubit 2.
4. Parameter 9 is used to apply a final CRY gate to rotate qubit 1 based on the state of qubit 2 which allows for entanglement to occur where qubit 2 affects qubit 1 which allows for backpropagation enabling bidirectional entanglement.

Finally, the SVQC measures and extracts the classical information in the form of a Pauli-Z expectation value of each qubit after all the pairs have been processed which fall in the range between [-1...1].

It is of note that the entire SVQC is wrapped in a PennyLane TorchLayer which allows for compatibility with PyTorch and ensures that the parameters are differentiable, enabling training with the HQCNN model.

- `quanvolutional.py`:

The quanvolutional layer acts as the hub connecting the quantum components together, enabling the execution of the ENEQR and SVQC modules, allowing for the reassembly of the features into a feature map which then undergoes classification by a CNN.

The process starts by acquiring the input images and encoding them into a quantum expectation vector if they haven't been already, once that's done, they are directed to the SVQC (`quantum_circuit.py`) where the patch's quantum vectors are loaded into qubits which are then entangled and undergo measurement of their Pauli-Z expectation value, this process produces the final quantum feature vector.

Upon processing all the patches, the results are collected into their respective feature map and are normalized based on a Min-Max Normalization layer `MinMaxNorm` to prepare for processing by the CNN.

2.4 Classical Branch

The Classical Branch handles all the classical components of the HQCNN and the source code as a whole, mainly possessing two roles; the first of which is to provide a performance baseline through the execution of a CNN that is purely classical with no quantum implementations in order to measure the effectiveness of the HQCNN and the second part involves the designation of the HQCNN model.

- `CNN.py`:

  The CNN model allows for a baseline performance metric which can be compared with the HQCNN model and serves as an independent tool by itself, possessing all the necessary components to function, going from the installation of the MNIST dataset and data visualization using Tensorboard to the logic which consists of the ReLU activations and convolutional layers.

- `model.py`:

  This is where the HQCNN is defined and wrapped, combining the quantum layer through the `quanvolutional` call which extracts the quantum features and rebuilds the feature map and the MLP head which consists of FC & ReLU Activation Layers that process the features and perform the final classification.

2.5 Merger & Execution

The final segment contains only one file and is considered to be the primary point of operations for this review as it enables the execution of the HQCNN model, this includes initializing the hybrid-quantum-classical model, defining the AdamW optimizer and the cross-entropy loss function in addition to managing all the high-level operations which will be deliberated below.

- `main.py`:

  This script acts as the principal component which allows for the HQCNN model to train and undergo testing with an intuitive section for the modification of

hyperparameters, notably the q (Value Qubits) variable. It allows for the launching of the TensorBoard which is the main tool used to view the results achieved by running the HQCNN model in addition to possessing the main function used for the training loop which has been repeated over 10 epochs for each run performed. Another major component that is essential is the ability for it to alternate between utilizing the precomputed features acquired from 'Segment 1' or running off the raw images using 'Segment 2' which incorporates real-time quantum encoding.

Chapter 3

Version Iteration Details

This Chapter intends to detail the improvements implemented within each version and emphasize their significance and performance gains, it is of note that the referencing to the improvements within the source code was initially going to take place by stating the modified or implemented line number, that was found to be messy and convoluted and was switched out for mentioning the function within the module or directly placing a code snippet for ease of access.

3.1 Version 1 → 2

The upgrade from Version 1 to Version 2 consists of the introduction of wrap-around entanglement in `quantum_circuit.py`, notably in the `generate_pairs()` function with the line `wrap = (self.wires - 1, 0)`, it is an upgrade purely from the quantum-encoding branch.

This in effect adds an additional connection between the first and last qubits through the usage of a CNOT gate, effectively creating a circular closed loop (i.e. 0-1, 1-2, 2-3, 3-4, and notably 4-0) ensuring that each qubit is at most two (2) entangling steps away from any qubit in the ring, this allows the SVQC to capture more connections in the quantum-encoded image patches which slightly increases the architectural accuracy at the cost of increased processing time due to the additional CNOT gate per layer.

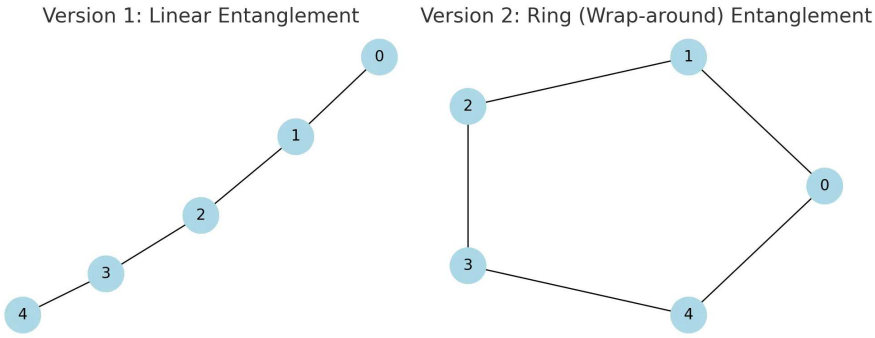

Figure 3.1.1: Illustration Comparison between V1 & V2's entanglement mechanism.

3.2 Version 2 → 3

The upgrade from Version 2 to Version 3 consists of the enhancements on the classical branch in the `model.py` module, with the additions consisting of; Two Batch Normalization (BN) layers, a second FC Layer, and a Dropout Layer.

Version 1/2 Code:

```
# FC layer input dimension: wires x (pooled_dim)^2
fc_input_dim = self.num_features * (pooled_dim ** 2)
# Deeper MLP head with BatchNorm
self.fc1    = nn.Linear(fc_input_dim, 128)
self.fc3    = nn.Linear(128, num_classes)
self.relu   = nn.ReLU()
```

```
# Deeper head forward
x = pooled.reshape(batch_size, -1).float()

x = self.fc1(x)                          # (B, 128)
x = self.relu(x)

out = self.fc3(x)                        # (B, num_classes)
return out
```

Figure 3.2.1: Code Snippet from Version 1-2's `model.py` module.

The additional layers provide various improvements across the board ranging from increasing the stability of the training process and accuracy of the training to ensuring that the neurons are not overfitted and do not rely on each other often.

Version 3 Code:

```
# FC layer input dimension: wires × (pooled_dim)^2
fc_input_dim = self.num_features * (pooled_dim ** 2)
# Deeper MLP head with BatchNorm
self.fc1     = nn.Linear(fc_input_dim, 256)
self.bn1     = nn.BatchNorm1d(256)          # Normalize 256 features :contentReference[oaicite:3
self.fc2     = nn.Linear(256, 128)
self.bn2     = nn.BatchNorm1d(128)          # Normalize 128 features :contentReference[oaicite:4
self.fc3     = nn.Linear(128, num_classes)
self.relu    = nn.ReLU()
self.dropout = nn.Dropout(p=0.3)            # Slightly lower dropout to retain capacity :content
```

```
# Deeper head forward
x = pooled.reshape(batch_size, -1).float()

x = self.fc1(x)                             # (B, 256)
x = self.bn1(x)                             # BatchNorm1d on 256 features :contentReference[oaici
x = self.relu(x)
x = self.dropout(x)

x = self.fc2(x)                             # (B, 128)
x = self.bn2(x)                             # BatchNorm1d on 128 features :contentReference[oaici
x = self.relu(x)
x = self.dropout(x)

out = self.fc3(x)                           # (B, num_classes)
return out
```

Figure 3.2.2: Code Snippet from Version 3's `model.py` module.

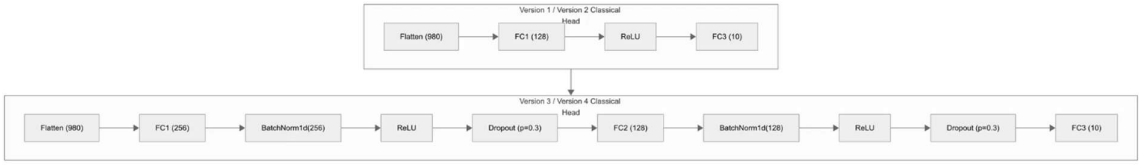

Figure 3.2.3: Diagram illustrating the improvements between Version 2 & 3.

3.3 Version 3 → 4

The upgrade from Version 3 to Version 4 consists of increasing the number of q-bit 'gray-value' qubits to 4 from 2 which increased the possible intensity levels to 16 as the intensity levels are calculated using 2^p, where p is the number of 'gray-value' qubits assigned, this is mainly accomplished in `main.py` with the modification of the hyperparameter q=2, and in `precompute_eneqr.py` with the modification of the `__init__` function to change the value of q to 4.

These changes in the quantum-encoding branch allow for the preservation of more image information, notably intensity details, however it necessitated the precomputation of the MNIST dataset again since upon changing the number of value qubits, we changed the circuit's structure which was intended to account for a total of 5 qubits instead of 7.

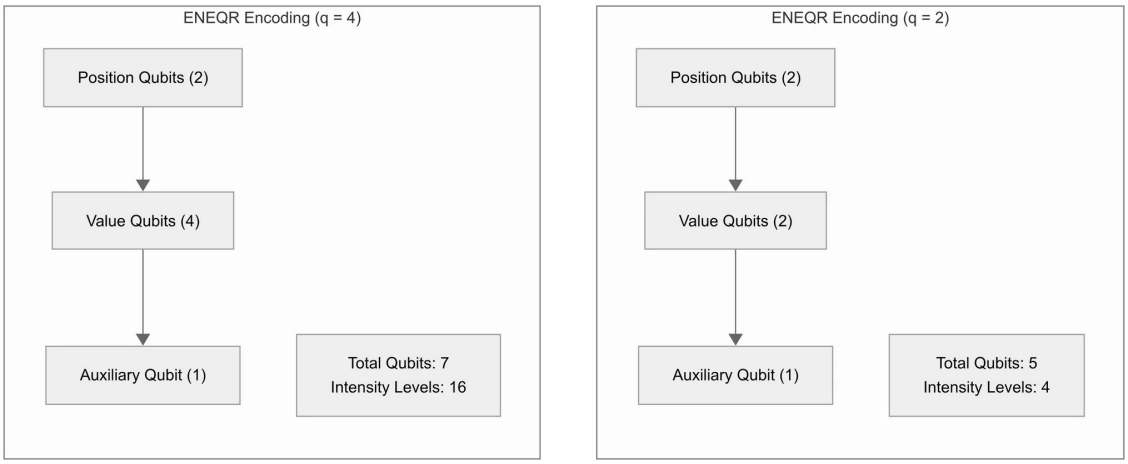

Figure 3.3.1: Diagram illustrating the qubits assigned per patch in between Versions 1-3 & Version 4.

Chapter 4

Points of Interest

This chapter pertains to the notable errors or discoveries uncovered during the course of this review and suggest corrections or improvements, they were found in collaboration with Mr. Ping, Dr. Tan and through cross-referencing the source code with the dissertation report and comparing them independently with my results, this chapter will not include grammatical, syntax or any form of lexical errors as that would be superfluous and arduous.

4.1 Incorrect Mention of CNOT Gates

In Chapter 3.2.1, Page 23, it is stated that:

> *We chose controlled-NOT (CNOT) gates for entanglement, as they create correlations between qubit pairs. The code reflects this structure by iterating through qubit indices and applying CNOTs (or another two-qubit gate) between q[i] and q[i+1] for i=0,1,2.*

This is in relation to the `quantum_circuit.py` module, however when we refer to the `generate_pairs()` and `variational_block()` functions, we uncover that the two-qubit gate used was a CRY Gate, this was confirmed by Mr. Ping when asked about it and can simply be changed to reflect the accurate gates used.

4.2 Incorrect Qubit Value in relation to Qubit Budget Per Patch

In Chapter 3.2.1, Page 23, it is stated that:

> *The code organizes this by applying the SVQC to every 2×2 patch across the image and collecting the four qubit outputs for each patch, constructing a feature map tensor.*

This statement is mostly correct, however the qubit budget per patch is set at five (5) when the value of the gray-value qubits '`q`' is set to 2, this is proven by referring to the equation used for calculating the number of wires/qubits assigned per patch:

$$\boldsymbol{self.wires = 2 * self.n + self.q + 1}$$

The value of the position qubits '`n`' is set to 2 and has not been modified with for the purposes of the report or this review which means that for Version 1 through 3, the qubit budget per patch is 2+2+1=5. This can be amended by modifying the value in the report directly.

4.3 Result Comparison Mismatch for V1 & V2

When attempting to compare the results between training runs against the reference runs, it was uncovered that there was a step mismatch within the reference runs for Versions 1 & 2, where they only reached a total of 4 steps, this was conveyed to Mr. Ping who responded stating that Versions 1 & 2 were ran using a `batch_size` of 32 instead of 64 as retaining the `batch_size` at 64 didn't provide significant performance or accuracy gains in his case, this is notably shown in the Epoch/Train_Accuracy and Epoch/Train_Loss graphs shown in Figure 4.3.1.

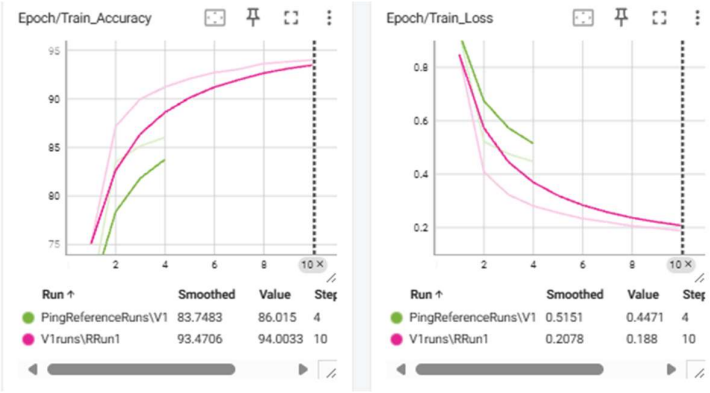

Figure 4.3.1: Diagram illustrating the improvements between Version 2 & 3.

In regard to reconciliation efforts, notes from Chapters 4.1.2 will be used as supporting evidence in order to account for any discrepancies.

4.4 Reference Run Fragmentation for Version's 3 & 4

When comparing the results between the training runs against the reference runs for Version's 3 & 4, it was stated that they were separated into two `.0` files, the reason stated was that the training could not be completed in one session. This would have made the results comparison arduous if not for the fact that placing both `.0` files in a folder allowed for them to be read as a singular file with a continuous training stream, this is shown in Figures 4.4.1 and Figures 4.4.2, where Part 1 of the file completely covers the first half of the merged runs and Part 2 covers the unsmoothed later half with an unmarked space in between stitched by their merger.

Figure 4.4.1: Screenshot from Tensorboard showing the colour codes for the reference run files for Version 3.

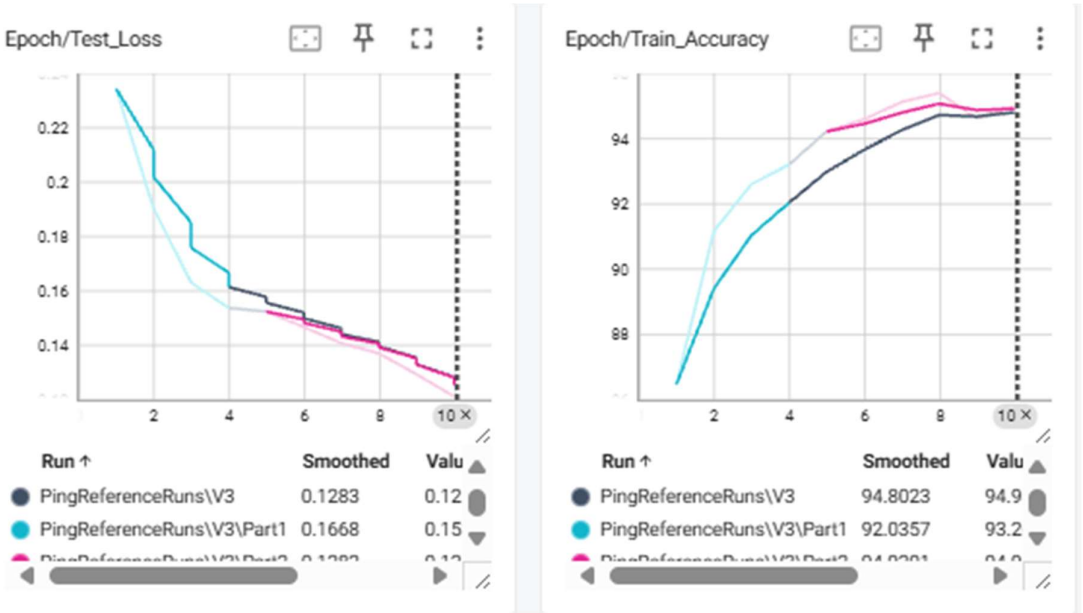

Figure 4.4.1: Reference Run Comparison for Version 3.

Chapter 5

Result Comparison

This chapter fully focuses on comparing the training runs performed locally against each other and against the reference runs provided by Mr. Ping, the locally trained runs were performed on a personal desktop with the following specifications:

CPU: I5-14400F; Baseline

GPU: RTX 4070; Overclocked by 200MHz

RAM: 32 GB of DDR4 at 2400MT/s

The specifications of the desktop performing the runs were stated as they will provide insight later on when comparing the locally trained files against the reference files, specifically in relation to time taken to finish training.

The first batch of comparisons will have the locally trained runs compared against each other followed by the comparison against the reference runs; this was decided as there was virtually no difference between the runs asides from the time taken to complete them which is attributed to application and background processes running during the training.

5.1 Version 1

Version 1's locally trained runs took on average 6 hours and 15 minutes to process and reached a peak of 94.0% accuracy which in contrast to the reference run's 86.4% is a significant jump and is puzzling as it necessitated rechecks to validate if the locally trained runs matched the architecture of Version 1's model through checking if the amount of wires were set to 5, ensuring that the wrap-around entanglement was not implemented

and more, however the validation confirmed that the locally trained runs were not ran on a different architecture, this is shown in Figures 5.1.1 and 5.1.2. A possible reason may be due to the different `batch_size` value used in the reference run (refer to Chapter 4.3), hindering it from reaching its full potential.

It is also of interest to mention that the locally trained runs possessed a steeper rise, and a lower loss rate compared to the reference runs.

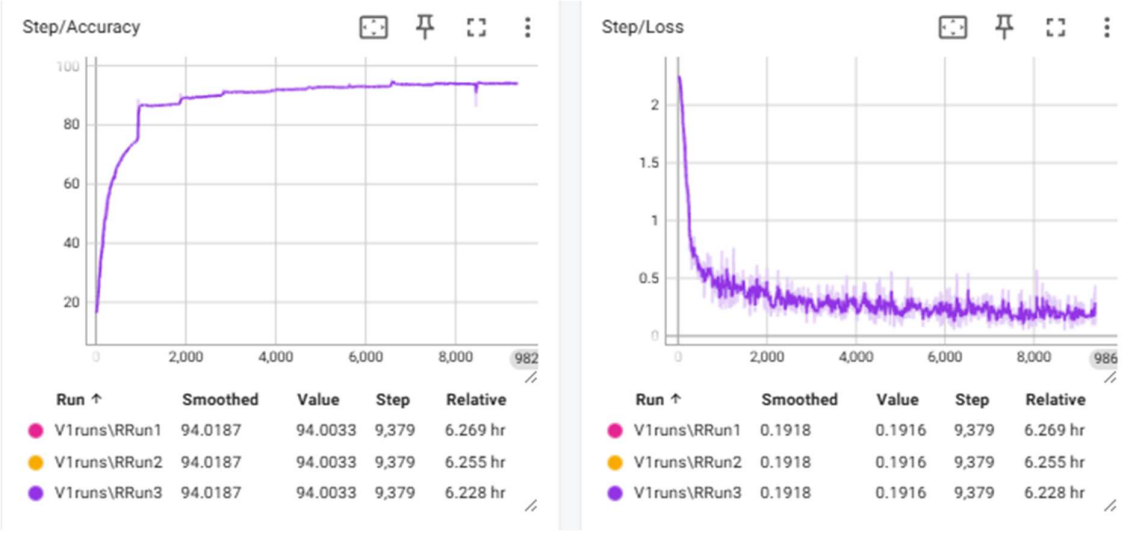

Figure 5.1.1: Locally ran Version 1 Model Training metrics.

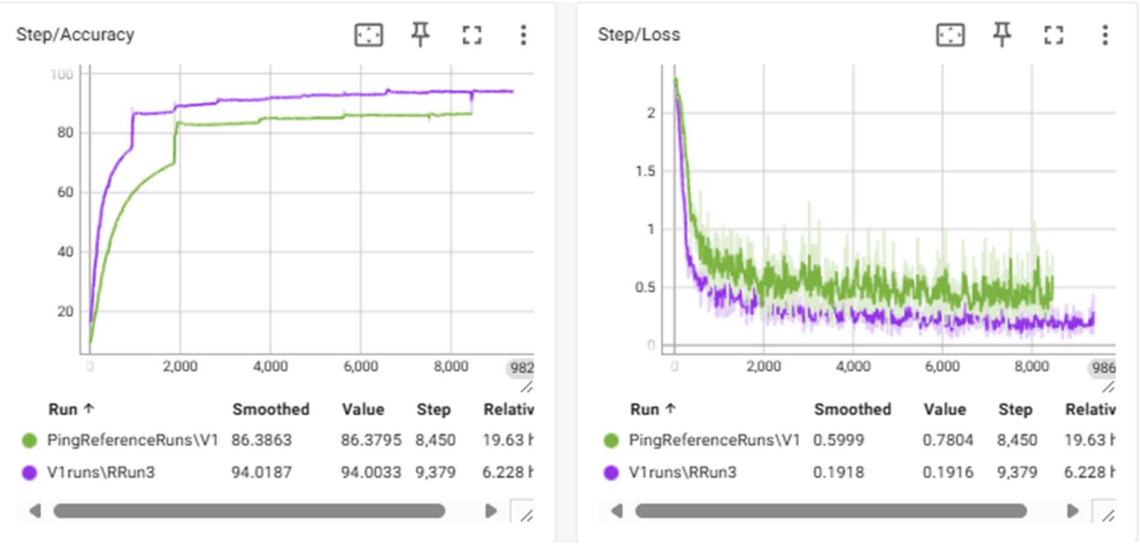

Figure 5.1.2: Version 1's Model Training metrics (Reference vs Locally ran)

5.2 Version 2

The unusual trend continues with Version 2's locally trained runs took on average 7 hours and reached a peak of 94.2% accuracy in comparison to the reference run's 88.5%, the performance gains are consistent with the original dissertation comparatively, with them being attributed to the introduction of wrap-around entanglement, this is demonstrated in Figure 5.2.1 and Figures 5.2.2.

Another persisting trend that stood out was that the training loss with the locally trained run possesed a lower loss than the reference run which stopped improving earlier.

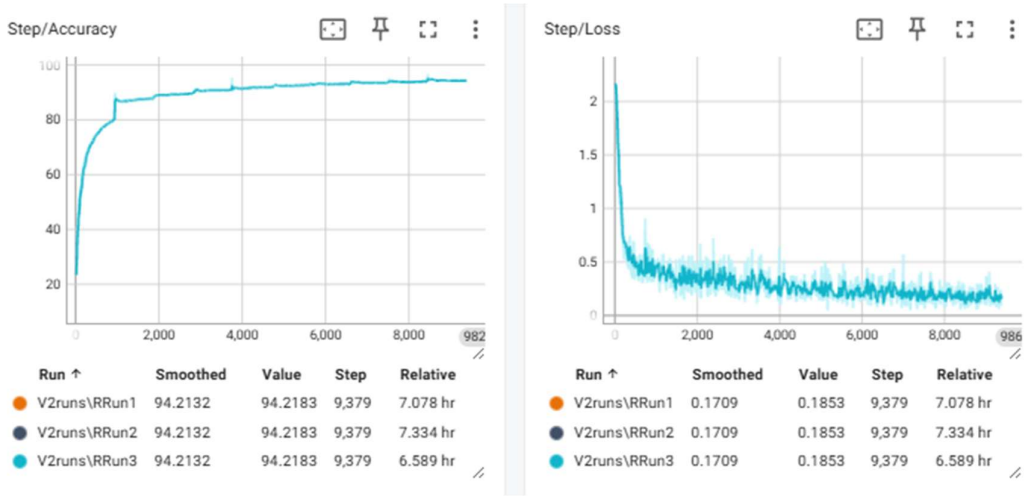

Figure 5.2.1: Locally ran Version 2 Model Training metrics.

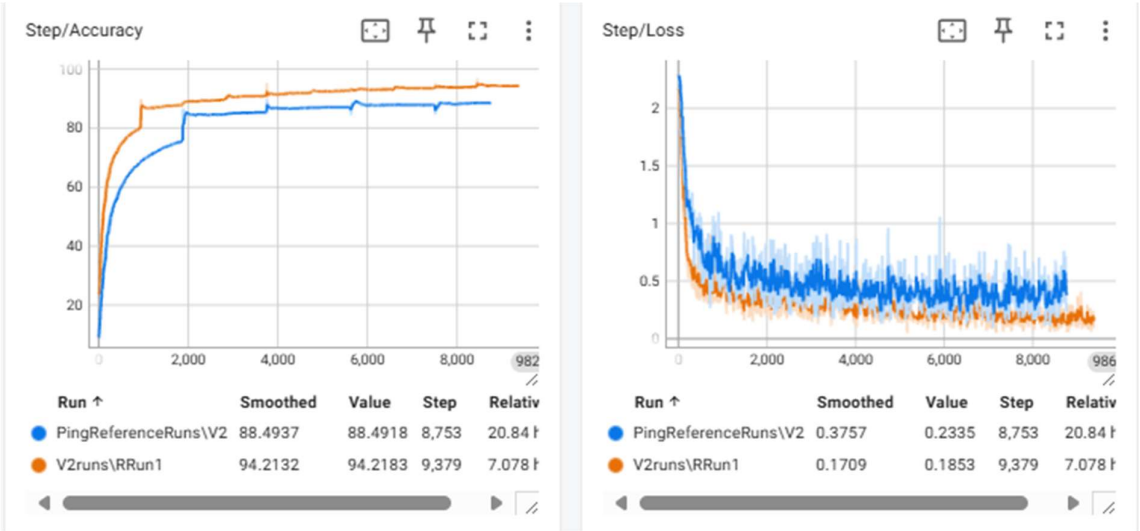

Figure 5.2.2: Version 2's Model Training metrics (Reference vs Locally ran)

Again, the files were verified to be consistent with Version 2's architecture, with the main change being the introduction of the wrap-around entanglement, and other modification such as the alterations to the MLP head and modification of the number of gray-value qubits remained the same.

5.3 Version 3

Once we move on to Version 3 and 4, we start seeing the results starting to closely resemble each other with Version 3's locally trained runs took on average 7 hours and 25 minutes and reached a peak of 95.2% accuracy while Mr. Ping's reference runs also reached a peak accuracy of 95.2%, albeit it took approximately 2 days 5 hours and 35 minutes, the improvement is attributed to the larger classical neural network with the introduction of the dropout layer and the additional BN and FC layers.

Figure 5.3.1: Locally ran Version 3 Model Training metrics.

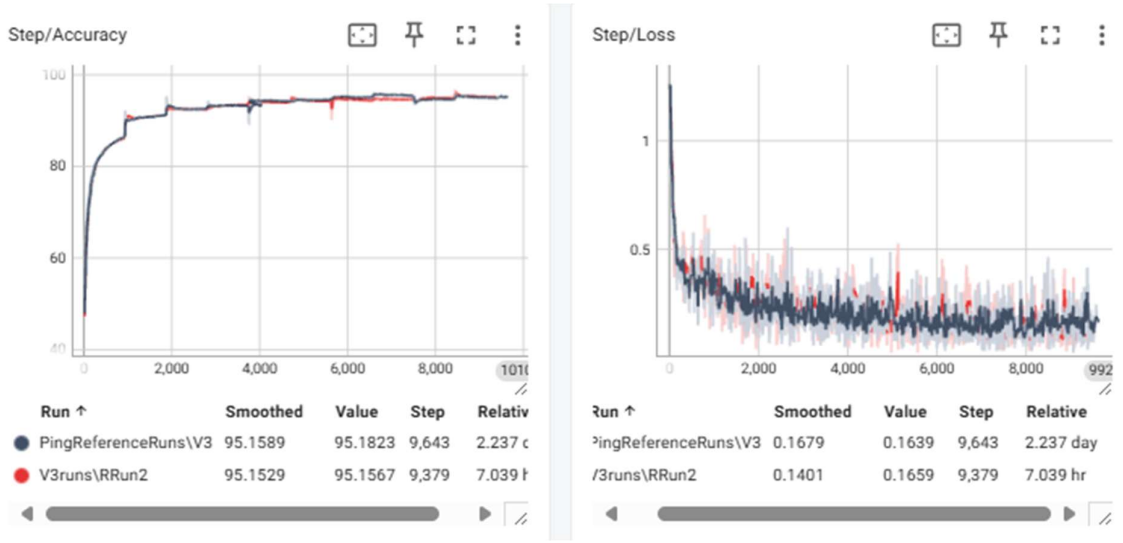

Figure 5.3.2: Version 3's Model Training metrics (Reference vs Locally ran)

Figures 5.3.1 and 5.3.2 demonstrate the metrics, and from there it can be observed that they overlap for the majority of the graph, with the locally trained run possessing slightly more variance.

5.4 Version 4

As for Version 4, it follows the same trend as Version 3, where the locally trained run creeps into the 96% accuracy rate, barely missing it by 0.0067% and taking on average 15 hours and 32 minutes in comparison to the reference run's 96.23% and taking approximately 1 day and 2 hours which is noticeably less time than Version 3's time taken.

This marks the largest increase in time taken to process a run and is attributed to increasing the number of gray-value qubits from 2 to 4 which in turns increased the total number of qubits from 5 to 7.

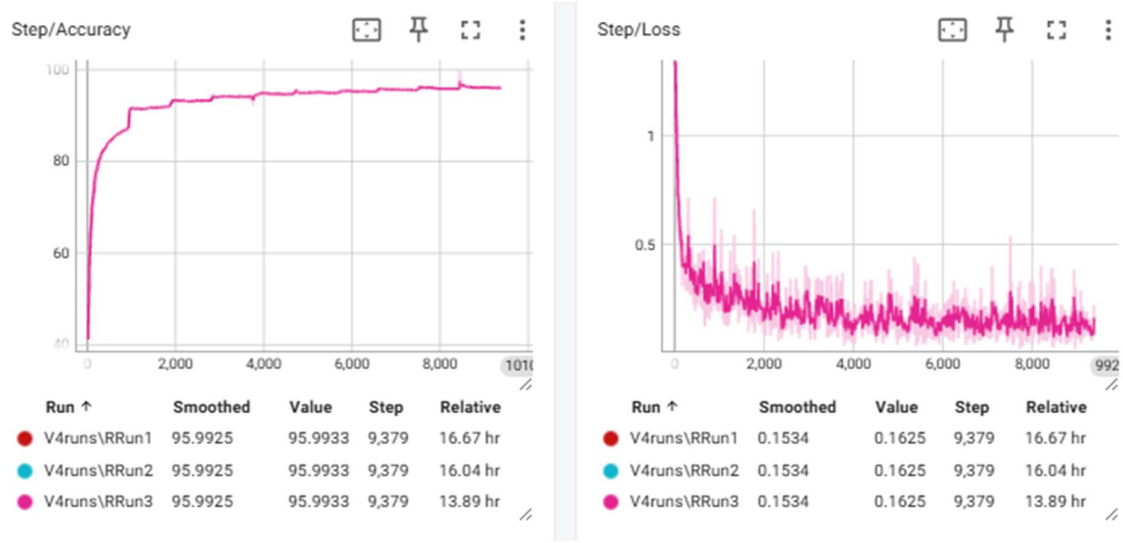

Figure 5.4.1: Locally ran Version 4 Model Training metrics.

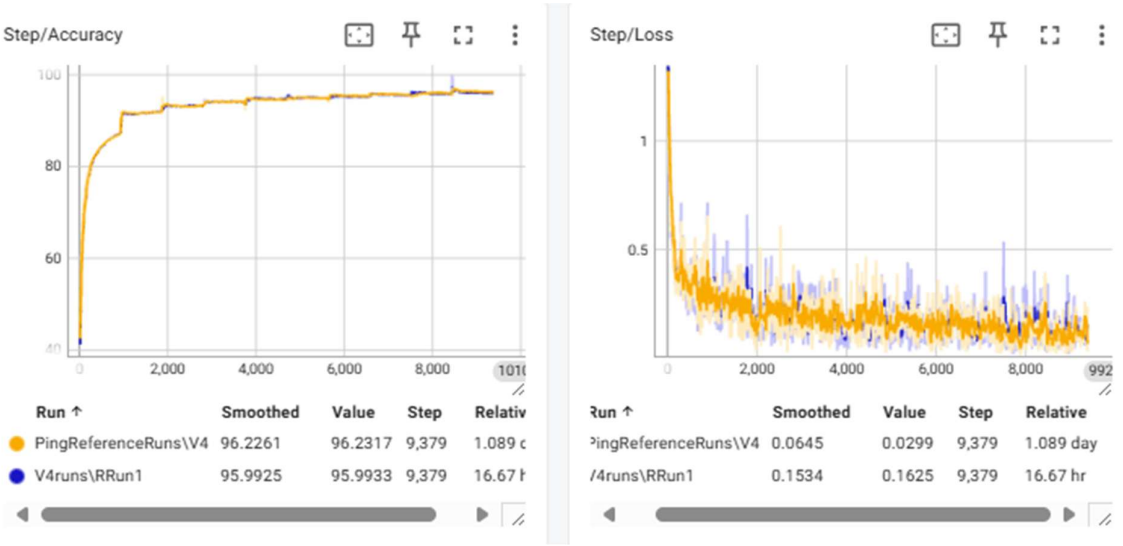

Figure 5.4.2: Version 4's Model Training metrics (Reference vs Locally ran)

From Figure 5.4.2, we can notice that the reference run is tightly fit in comparison to the locally trained run which possesses lightly more variance, also an important finding is that the reference run possesses the lowest loss, being ~5.43x lower than the locally trained run.

Glossary

The Glossary is intended to provide rudimentary knowledge of quantum encoding and quantum gates in order to assist the readers with understanding the FYP and the Review, it is of note that it will be separated into sections in order to compartmentalize all relevant terms to one part.

Quantum Encoding Techniques

- FRQI:

  FRQI is an abbreviation for 'Flexible Representation for Quantum Images' and it is an early quantum-image encoding technique that uses amplitudes to encode pixel intensities, this is achieved by utilizing one intensity qubit which is stored in a rotation angle alongside 2n position qubits, its main drawbacks constitute its need for repeated measurements to recreate the intensity and depth.

- NEQR:

  NEQR is an abbreviation for 'Novel Enhanced Quantum Representation' and it is an upgrade over FRQI as it stores exact pixel intensities/grayscale values in their basis state, producing the state: $\frac{1}{2^n}\sum_{x,y=0}^{2^{n-1}} I(x,y)\rangle|xy\rangle$. This is done using an additional set of 'q' intensity qubits, thus changing the equation from $(2n+1)$ to $(2n+q)$. The main drawback with this is the high qubit overhead and significant circuit depth which consists of the reliance of multiple CNOT gates.

- ENEQR:

  ENEQR is an abbreviation for 'Enhanced Novel Enhanced Quantum Representation' and it represents the merging of concepts between FRQI & NEQR by using Hadamard gates to put the 2n position qubits in a state of uniform

superposition in order to encode the spatial indices, which is supported by an auxiliary qubit that acts as the control bits for the MCX and CNOT gates in order to write the gray-value bit onto the 'q' intensity qubits, note that this increases the amount of qubits to $2n + q + 1$.

Variational Circuits

- VQC:

  VQC is an abbreviation for Variational Quantum Circuit, which utilize multiple, parameterized single-qubit rotation gates and enables entanglement using CNOT or CRY gates.

- SVQC:

  SVQC is an abbreviation for Structured Variational Quantum Circuit, its an improvement over RVQC (Random Variational Quantum Circuit) due to the organized nature of the quantum gates which allow for a more systematic approach, reducing randomness and allowing for improved readability.

Quantum Gates

- CNOT Gate:

  A Controlled Not Gate is a two-qubit quantum gate that inverts a qubits state if, and only if the control qubit is in the $|1\rangle$ state. It is used to achieve entanglement and is the equivalent of a reversible XOR gate.

- Hadamard Gate:

  A Hadamard Gate is a single-qubit quantum gate that maps a state into an equal, uniform superposition, where the qubit exists in a $|0\rangle$ and $|1\rangle$ state in parallel, this is done to allow quantum circuits to explore multiple paths in parallel.

- Pauli-X Gate:

  A Pauli-X Gate flips the state of a qubit from |0⟩ to |1⟩ and from |1⟩ to |0⟩, representing the quantum variant of a NOT gate.

- MCX Gate:

  A Multi-Controlled-X Gate is a multi-qubit gate that resembles a CNOT gate with the addition of more control qubits, where the Pauli-X operation is triggered once all the control qubits are in a |1⟩ state.

- RX, RY, RZ Gates:

  Rotation-X,Y,Z Gates are quantum gates that rotate a qubits state around the X,Y, or Z-axis by a given angle with the purpose of encoding classical data into quantum states.

- CRY Gate:

  A Controlled-RY Gate is a two-qubit RY Gate with a conditional activation based on if the control qubit is in a |1⟩ state, it is one of the other methods used to establish entanglement between qubits.

Values & Bits

- Classical Bit:

  A Classical Bit is the conventional unit of transmitting classical information, which only consists of two states; 0 and 1, which can be clearly observed and do not exhibit any quantum phenomena.

- Quantum Bit:

  A Quantum Bit, commonly referred to as a Qubit, is the conventional unit of transmitting quantum information, it consists of two states; |0⟩ and |1⟩ and can exhibit quantum phenomena such as entanglement and superposition where a qubit can exist in both states in parallel.

- Pauli-Z Expectation Value:

  The Pauli-Z Expectation Value is the measurement that dictates the degree that a qubit's state has along the Z-axis, ranging [-1,1] where -1 indicates that its perfectly in state $|1\rangle$, 1 indicates its in state $|\text{-}1\rangle$ and $|0\rangle$ indicates it is in equal superposition.

ANN [Artificial Neural Network] Terms:

- FC Layer:

  A Fully Connected Layer constitutes a layer where every neuron in a layer is connected to every neuron in the previous layer, this combines the extracted features and allows for the completion of tasks such as classification & regression.

- ReLU Activation Layer:

  A Rectified Linear Unit Layer is a non-linear activation layer that introduces non-linearity into neural networks and helps mitigate vanishing gradients, it is defined as $ReLU = max(0, z)$.

- BN Layer:

  A Batch Normalization Layer is a layer that normalizes scalar features to decrease the mean and unit variance to zero, this in turn stabilizes the training and improves the convergence of the model.

- MLP Head:

  A Multi-Layer Perceptron Head is a neural network which consists of FC layers and non-linear activation layers which gather data and extract features from within the dataset to achieve a task, such as classification.