

# CPEN 400Q Project Report

## Experimental Quantum GANs

Yuyou Lai, Juntong Luo, Sam Schweigel, Bolong Tan

### Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Theory</b>	<b>2</b>
2.1	Background knowledge . . . . .	2
2.2	Convergence . . . . .	2
2.3	Training . . . . .	2
2.4	Gradients . . . . .	3
<b>3</b>	<b>Quantum GANs</b>	<b>4</b>
3.1	Multi-layer parameterized quantum circuits . . . . .	5
3.2	Post-selection . . . . .	5
3.3	Batch GANs . . . . .	5
<b>4</b>	<b>Results</b>	<b>6</b>
<b>5</b>	<b>Software</b>	<b>10</b>
<b>6</b>	<b>Reproducibility</b>	<b>12</b>
6.1	Experimenting with actual quantum hardware . . . . .	12
6.2	Simulating Noise of the Real Quantum Processor . . . . .	13

# 1 Introduction

In recent years, Generative Adversarial Networks (GANs) have revolutionized deep learning by enabling the generation of realistic images, videos, and audio. While classical GANs have been extensively studied, quantum GANs have emerged as a promising alternative due to their potential to achieve superior performance using fewer resources. The research paper titled "Experimental Quantum Generative Adversarial Networks for Image Generation" proposes two new methods for image generation using GANs trained on a quantum computer [4]. The paper delves into the theory behind GANs, discussing why they are expected to converge and how loss is computed. It also compares the benefits of quantum GANs to classical GANs and presents the results obtained using both proposed approaches: the patch and batch strategies. Additionally, the authors ran experiments on a real superconducting quantum processor, demonstrating the potential of quantum GANs to run on contemporary noisy intermediate-scale quantum (NISQ) machines. In this report, we present our implementation of the batch GAN strategy from the paper, aiming to reproduce and extend the key results presented while providing additional insights and contributions. We follow a similar structure to the original paper and discuss the theory and derivation of key results, implementation details, and experimental results.

## 2 Theory

### 2.1 Background knowledge

The paper proposes a new method for the task of image generation, which trains GANs using the quantum computer. GANs consist of two neural networks, a generator, and a discriminator. The job of the generator is to create synthetic data that is as close as possible to the real data, such as a text word or an image, whereas the discriminator tries to distinguish between the real data and generator-generated synthetic data.

### 2.2 Convergence

The training process can be viewed as a zero-sum game, where the generator tries to minimize the value of the objective and the discriminator tries to maximize it.

A GAN has *converged* when the generator can no longer improve, or the discriminator can no longer distinguish between real and synthetic data. At this point, both two players have optimal strategies, and the generator has a 50% chance of correctly distinguishing a generated image from a real one.

### 2.3 Training

If the training converges, the objectives for the generator and discriminator should find the parameters  $\theta$  for the generator and  $\gamma$  for the discriminator that satisfy:

$$\min_{\theta} \max_{\gamma} \mathcal{L}\{D_{\gamma}[G_{\theta}(z)], D_{\gamma}(x)\} := \mathbb{E}_{x \sim P_{data}(x)}[\log D_{\gamma}(x)] + \mathbb{E}_{z \sim P(z)}(\log\{1 - D_{\gamma}[G_{\theta}(z)]\})$$

Where  $D_{\gamma}(x)$  is the discriminator's estimate of the probability that training example  $x$  is real,  $\mathbb{E}_{x \sim P_{data}(x)}$  is the expected value over all training examples.  $G_{\theta}(z)$  is the generator's output when given noise  $z$ ,  $D_{\gamma}[G_{\theta}(z)]$  is the discriminator's estimate of the probability that a fake instance is real, and  $\mathbb{E}_{z \sim P(z)}$  is the expected value over all random inputs to the generator.

Specifically, for the discriminator, the loss is computed as the sum of the binary cross-entropy between the true labels (1 for real data and 0 for fake data) and the predicted probabilities. In other words, the discriminator aims to minimize the loss when correctly classifying real and fake examples. The lower the loss of the discriminator, the more effective the discriminator is in distinguishing between real and fake data. For the generator, the loss is computed as the binary cross-entropy between the true labels and the discriminator's prediction of the generated data. The generator aims to maximize this loss because it means that the discriminator is classifying the synthetic data as real. In other words, the generator aims to produce synthetic data that minimize the discriminator's ability to distinguish between real and fake examples.

## 2.4 Gradients

We compute gradients for the discriminator's parameters in two configurations. In Figure 2, the loss is minimized when discriminator identifies the generator's image as fake, while in Figure 1 it is minimized when it guessed that the training example is real.

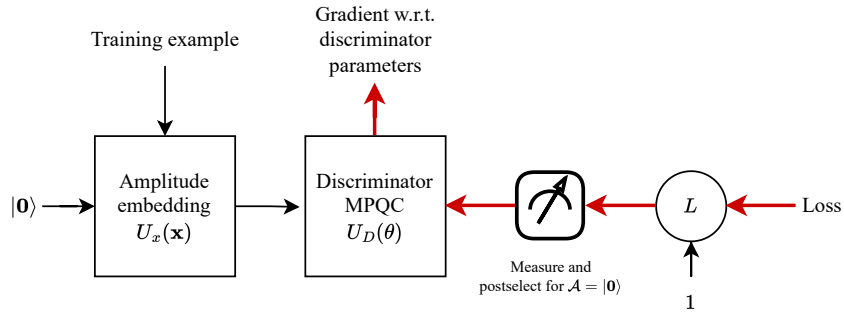


Figure 1: Gradients computation for the real example w.r.t. discriminator parameters.

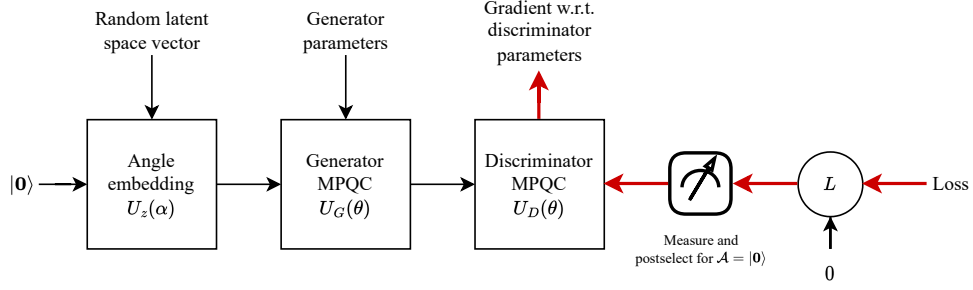


Figure 2: Gradients computation for the fake example w.r.t. discriminator parameters.

Gradients are computed for the generator in F Figure 3. The same configuration is similar Figure 2, but the target label provided to the loss function is reversed.

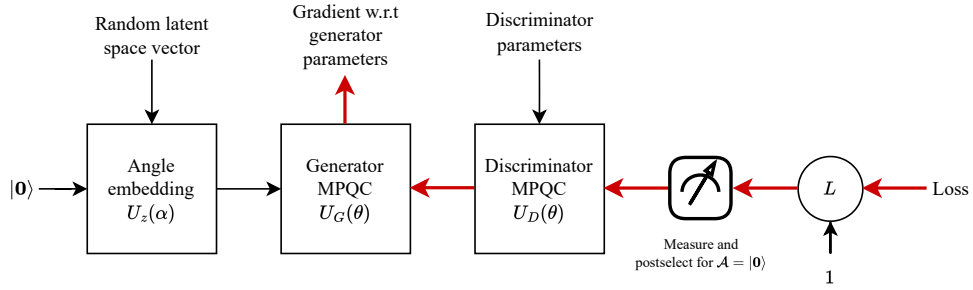


Figure 3: Gradients computation w.r.t. generator parameters.

### 3 Quantum GANs

Depending on the available quantum computing resources, the authors give two ways of getting some advantage when training a quantum circuit over a classical GAN:

- When there are many more feature dimensions (for example, the  $8 \times 8$  MNIST digits used as an example), we split up the feature vector and train many simple, independent quantum generators. Their output is stitched back together and fed to a single classical discriminator. During training, gradients from the discriminator are back-propagated through the quantum GANs to obtain gradients with respect to their trainable parameters. This is the “patch” strategy. An implementation is available on the PennyLane blog[2], so we focus on implementing batch GANs only.
- When there are enough qubits to embed an entire training example into the amplitudes of the feature register, we train both a quantum generator and discriminator circuit. With some additional qubits as an “index register”, we can embed a superposition of training examples for an entire basis over the index register. By training this circuit, we compute gradients for an entire batch training data with few additional gates. This is called the “batch strategy”.

### 3.1 Multi-layer parameterized quantum circuits

Both the generator and discriminator are multi-layer parameterized quantum circuits, which have been shown to be more expressive than neural networks on a parameter-to-parameter basis in many situations[11].

An MPQC consists of several layers of a repeating pattern: trainable gates (generally rotations so that parameter shift derivatives can be taken), followed by a series of controlled gates to entangle the resulting state. Huang et. al use a structure like the one shown in figure 4, but MPQCs are flexible with respect to what gates are used, and in what order the entanglers are applied.

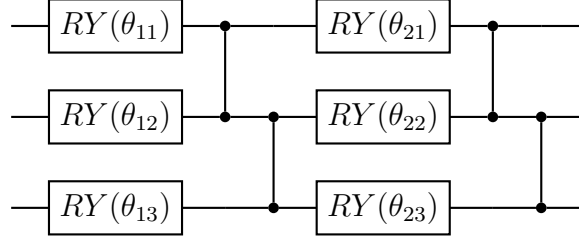


Figure 4: A two-layer MQPC using  $RY$  gates as trainable, and  $CZ$  gates for entangles, like the ones described in the paper.

### 3.2 Post-selection

It is well known that ordinary neural networks need nonlinearities to be universal function approximators[1]. MPQCs must also be able to learn nonlinearity if they are to be useful for image generation. This is accomplished by applying the MPQC to extra ancillary qubits and throwing away measurements where the ancillary subsystem is not in some fixed state. We choose  $|0\rangle$  arbitrarily.

### 3.3 Batch GANs

Figure 5 shows a high-level view of the entire batch GAN circuit.

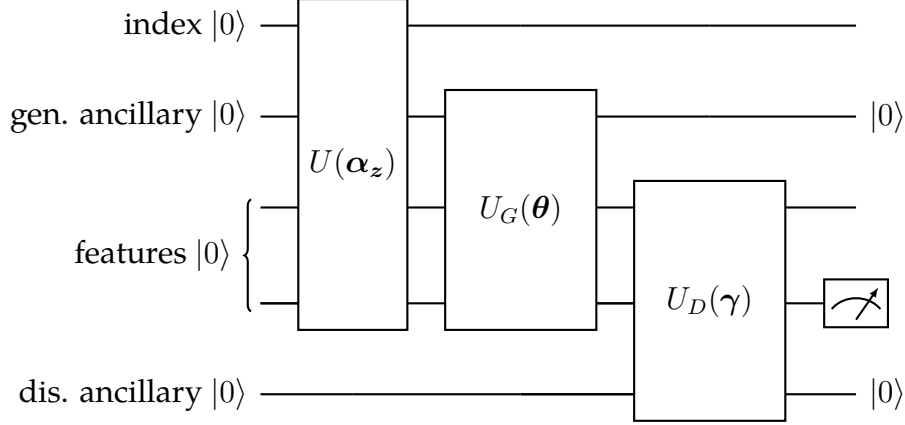


Figure 5: Schematic for batch GAN in the configuration used for training on generated examples.

The first unitary gate,  $U(\alpha_z)$ , is an angle embedding from the random latent space onto the feature register (as well as generator ancillary and index register).  $U_G(\theta)$  and  $U_D(\gamma)$  are the MPQCs for the generator and discriminator respectively. When training, we make a measurement, throw it away if the ancillary bits are not zero, and estimate the probability that a non-ancillary bit of the discriminator’s output (in the feature register) is one. This is the discriminator’s probability that the given example is real.

When training with real images, the generator circuit is missing. Instead, where  $x_i$  are training examples for  $i \in \{1, \dots, k\}$ , an amplitude embedding initializes the index register and feature registers with the state:

$$-2^k \sum_{1 \leq i \leq k} |i\rangle \otimes x_i$$

The discriminator output is read off as normal.

## 4 Results

Huang et al. demonstrate the patch architecture with an  $8 \times 8$  pixel, resized MNIST digit dataset. This is practical to run when divided into four patches of size  $8 \times 2$ , but a patch strategy does not apply for the discriminator (it must be large enough to “see” the generated image). To demonstrate batch GANs, they construct an artificial dataset small enough that an entire training example can be embedded into the feature register at once.

The dataset, referred to as the “grayscale bar dataset” in the paper, is defined by giving an explicit probability distribution for a random image  $x_{ij}$  (where  $i$  is the row and  $j$  is the column in the graphical representation). Some typographic errors from the paper have been corrected:

$$\mathbf{x}_{00} \sim \text{unif}(0.4, 0.6)$$

$$\mathbf{x}_{10} = 1 - \mathbf{x}_{00}$$

$$\mathbf{x}_{i1} = 0$$

The distribution is already normalized so that it sums to 1 (a prerequisite for the quantum generator).

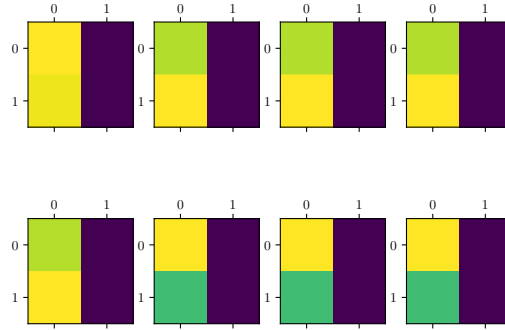


Figure 6: Samples from our generator for the bar dataset.

The authors then train a batch GAN with 12 trainable parameters, including a 3-layer quantum generator with 1 ancillary qubit, and a 4-layer quantum discriminator with 1 ancillary qubit.

Both the generator and discriminator use the same multi-layer parameterized circuit:  $RY$  trainable gates and  $CZ$  in the “staircase” configuration for the entanglers.

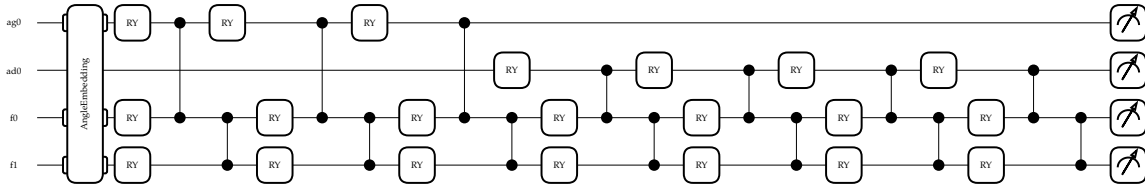


Figure 7: Our best guess at the exact quantum batch circuit used in the paper.

Figure 7 shows the circuit generated by our software (drawn by PennyLane). The generator ancillary qubits are labelled “agn”, discriminator ancillary “adn”, and the feature register is labelled “fn”. If there were index qubits, they would be labelled “in”.

After training the batch GAN for 350 iterations, its performance is compared with a classical GAN with a similar number of trainable parameters.

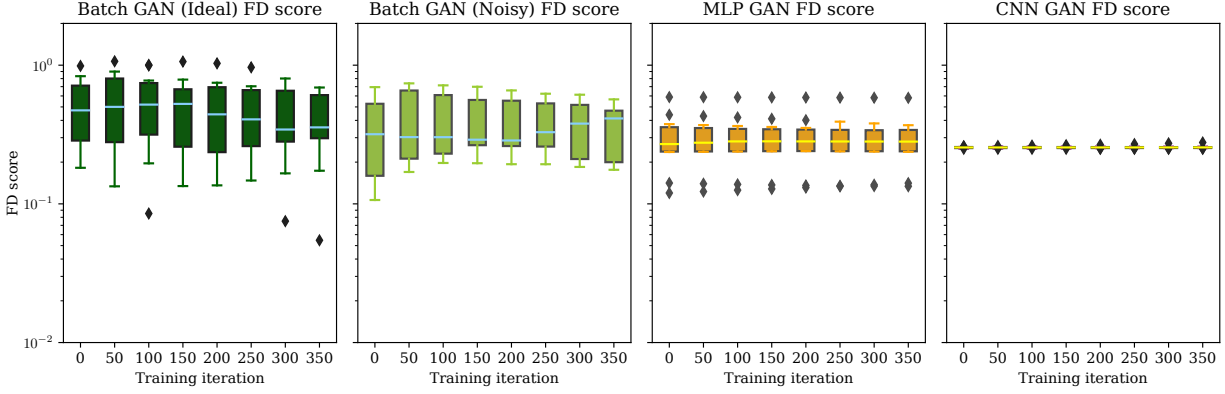


Figure 8: The performance of quantum batch GAN on idea device, quantum batch GAN on noisy device, classical GAN with MLP generator, and classical GAN with CNN generator.

In Figure 8, we compare the performance of our batch GAN, and the classical GANs described in the paper. All of them have similar number of trainable parameters ( $\approx 11$ ), but the best batch GANs show a much lower minimum FD score (more closely resembles the dataset). We found that, like the paper, quantum GANs are much more variable with respect to their initial parameters.

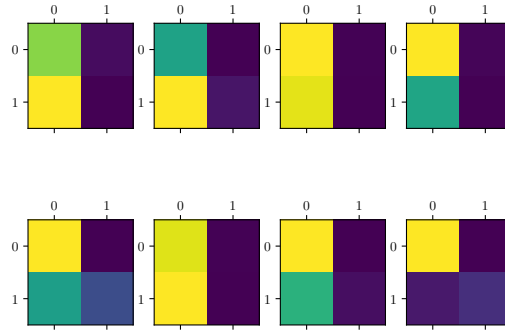


Figure 9: Some generated images from one particular trained batch GAN.

For example, in Figure 10, the same 12-parameter batch GAN is trained with differing initial parameters. The left GAN converges and generates good images in just under 1500 iterations, while the right has still not converged after 2000.



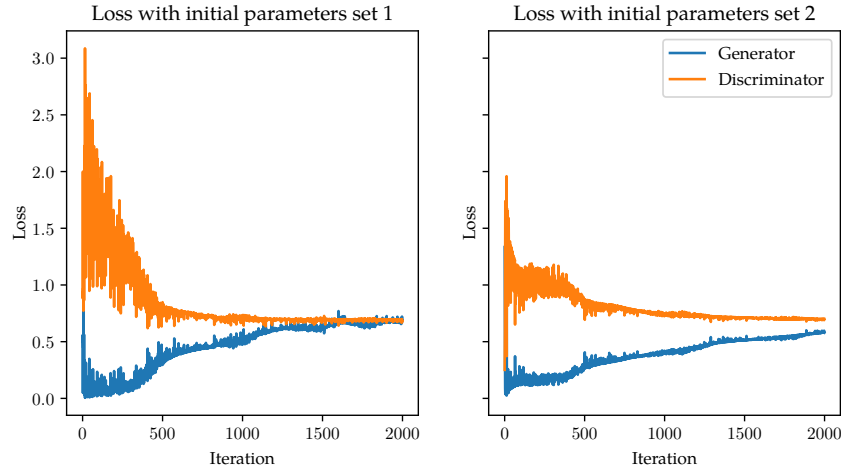


Figure 10: Comparison of the loss over time when training starting with two sets of random initial parameters.

Additionally, we compare quantum batch GANs with varying numbers of index qubits. The results are illustrated in Figure 11, which presents the loss over time with different number of index qubits. We observe that increasing the number of index qubits can indeed leverage parallelism and lower the number of iterations required for convergence. Note that the original paper does not include similar comparison; it does not present any evaluation with more than 0 index qubit.

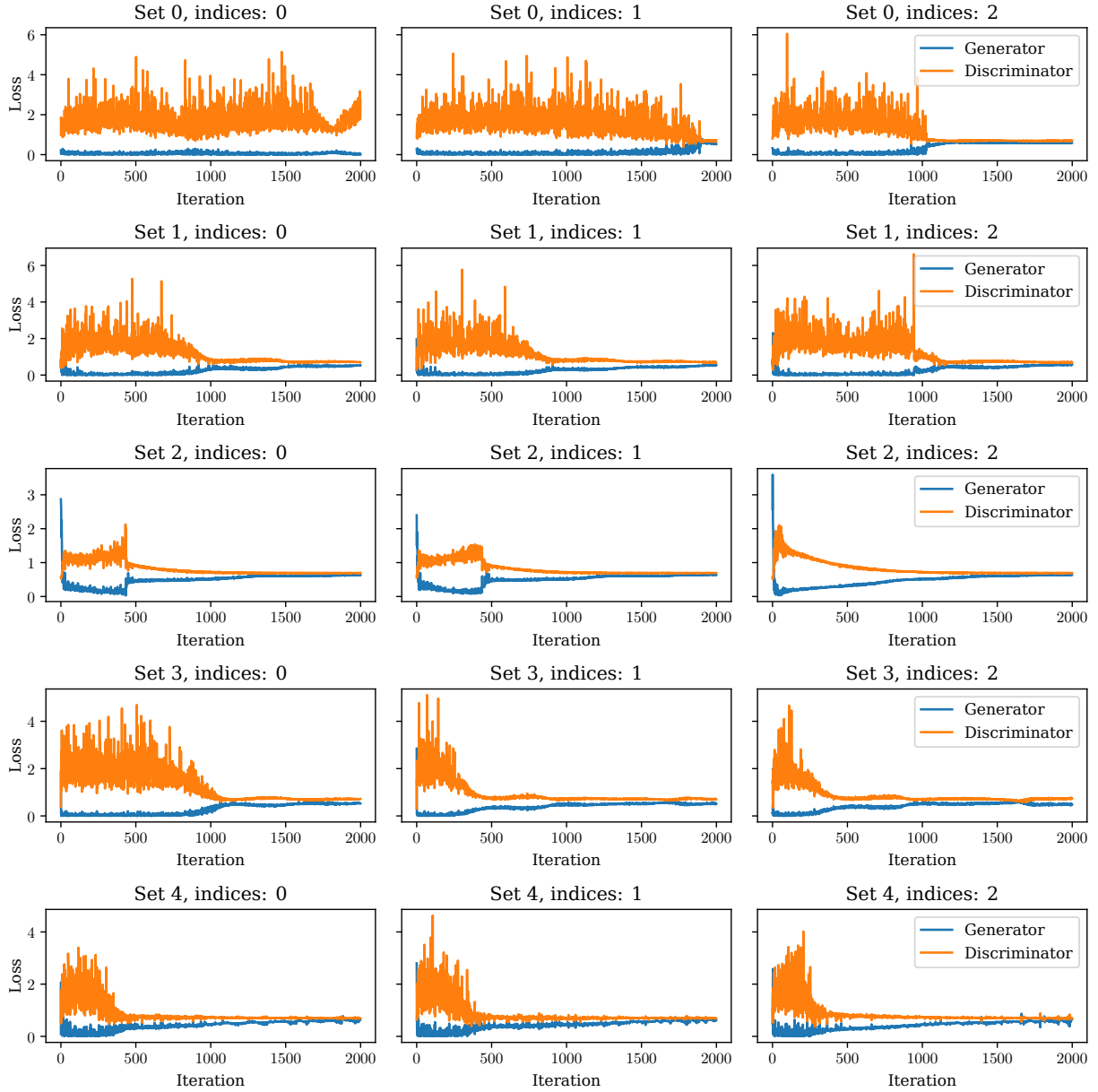


Figure 11: Comparison of the loss over time when training starting with different sets of initial parameters and numbers of index qubits.

## 5 Software

We choose to use PennyLane for a few reasons:

- PennyLane has excellent support for JAX [3], which investigated at the suggestion of Professor Olivia Di Matteo at our midterm check-in. JIT'ing the training step with

JAX is several orders of magnitude faster than using PyTorch (we now see >1000 iterations per second).

- PennyLane is designed to be “hardware and device agnostic” [7]. It offers many plugins that enable the same circuit written in PennyLane to be run on external quantum devices. Notably, we are able to run the quantum circuit on the JAX device (for low runtime), on the Qiskit Aer device (for simulating noise), and on IBMQ’s real quantum hardware without rewriting the quantum circuit.

JAX also helped us solve a serious issue with machine learning demonstrations. Since training is so sensitive to initial parameters, we want our data to be completely reproducible. Ordinarily, this can be solved by setting a seed for the global random number generator (for example, with Numpy). However, this is not ideal. Say we generate the training examples by invoking the RNG, and then generate the initial parameters by invoking it again. It is repeatable, but interferes with experimentation. If we were to increase the number of generated training examples, it would completely change the state of the RNG when it is used to generate the initial parameters, influencing the outcome of training.

We use JAX’s splittable RNG keys in a way that tries to minimize this unintended effect: to set up a GAN the user must pass a key, which is split into the subkeys required for each task. How the randomness is used after the key is split has no effect on the other keys.

We factored out the most reusable components into a library, the `quantumgan` module, in order to make experiments with quantum GANs as easy as possible. The code to train GANs (which is rather more involved than for many other machine learning models, since we need to interleave two optimizers and take multiple gradients with respect to some parameters but not all) is factored out into its own module, `quantumgan.train`, which will train any model that implements our GAN interface. Both our classical GAN and batch GAN implement this interface, and so are trained with the same code.

While one benefit of the batch GAN is to be able to train with many training examples at once, the paper never does any simulations with more than zero index qubits (making them completely sequential). We added support for this and tested it (simply setting the `batch_size` greater than zero in the arguments to `BatchGAN` will set up the index bits necessary).

The paper also mentions that MPQCs are trainable in many situations, even if the rotations and entanglers used are not the ones given. The layout of the entanglers can also be varied to better suit the available hardware. The paper, and our implementation, default to *CZ* and *RY*, but it is readily configured or extended by implementing the MPQC interface:

```
features_dim, batch_size = 4, 2
gen_params, dis_params = BatchGAN.init_params(
    params_key, features_dim,
    gen_layers=3, gen_ancillary=1,
    dis_layers=3, dis_ancillary=1,
)
gan = BatchGAN(
    features_dim, batch_size, gen_params, dis_params,
```

```

trainable=qml.RZ,
entangler=RandomEntangler(key, entangler=qml.CNOT),
)

```

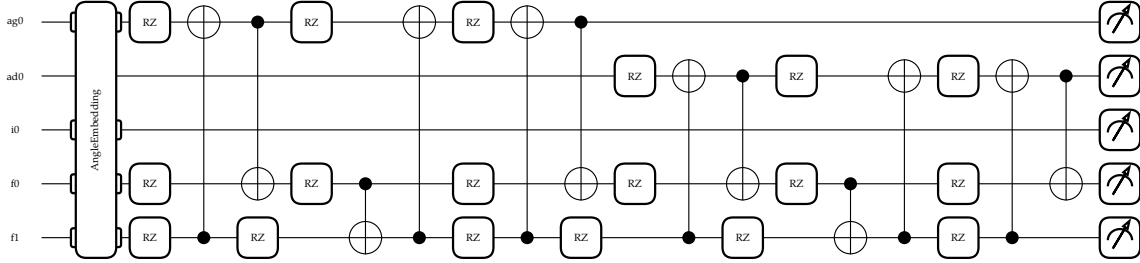


Figure 12: An exotic batch GAN produced by the above code, containing an index register and a pseudorandom entangler layout.

## 6 Reproducibility

Unfortunately, we encountered multiple major issues regarding reproducibility:

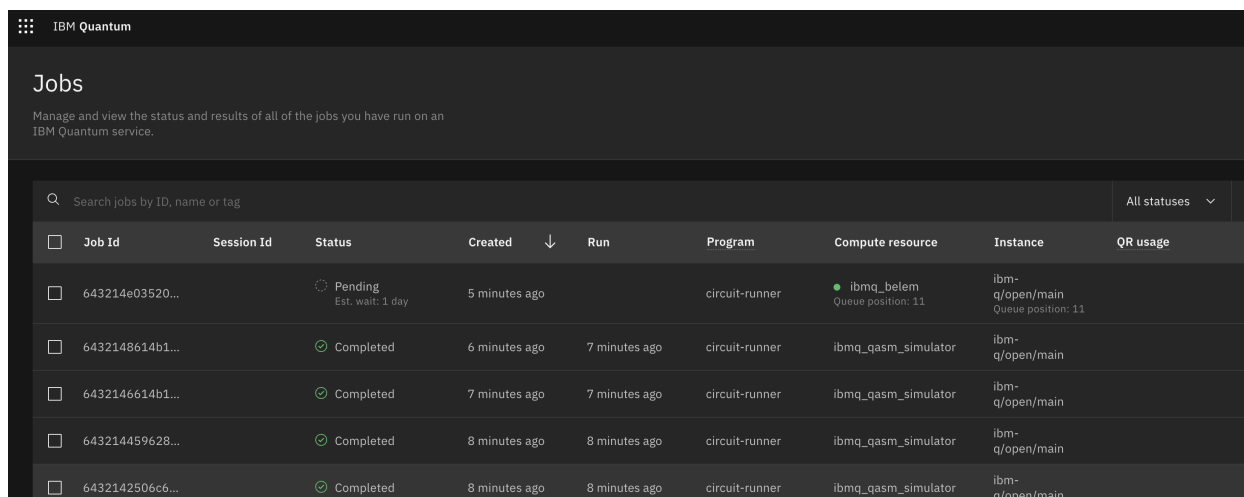
1. *Missing hyper-parameters.* The paper did not provide some important hyperparameters (e.g. number of neurons, learning rate) for their classical generators. Rather, the paper only provides the total number of trainable parameters, and we are unable to come up with a set of hyperparameters that provides the exact same numbers of trainable parameters.
2. *Fréchet distance (FD score).* There is no detailed discussion about how the trained GANs are scored. They reference two papers on the Fréchet/2-Wasserstein distance that are of relevance only to multivariate gaussian distributions, and their applicability to scoring the samples generated from the batch GAN against the known, non-gaussian distribution the training examples are drawn from. We were unable to exactly reproduce the Fréchet distance scores that appear in the paper's box plots. Despite the efforts made, the Fréchet distance scores we obtained are still far from the paper's results.
3. *Randomness.* We found the machine learning models used in the paper very sensitive to the random seed. We believe this might be an issue resulting from the small sizes of the models.

### 6.1 Experimenting with actual quantum hardware

One of the key contributions of the paper is that the experiments were run on a real superconducting quantum processor. While we do not have access to the quantum device used by the authors, we learned that IBM Quantum is providing free access to their 7-qubit and

5-qubit QPUs through the Open Plan [5]. The PennyLane-Qiskit plugin also provides a convenient “qiskit.ibmq” device to run our circuit on the IBMQ’s hardware [8].

The sign-up process of IBMQ’s Open Plan is straightforward, and PennyLane’s “qiskit.ibmq” device works almost out-of-box, but the waiting time turns out to be outrageously long. As shown in Figure 13, despite being the 11th in the queue, the estimated wait time for the job we submitted was 1 day, making training quantum GANs on it unfeasible. This outrageously long queuing time is not uncommon on IBMQ machines [9].



Job Id	Session Id	Status	Created	Run	Program	Compute resource	Instance	QR usage
643214e03520...		Pending Est. wait: 1 day	5 minutes ago		circuit-runner	ibmq_belem Queue position: 11	ibmq-open/main Queue position: 11	
6432148614b1...		Completed	6 minutes ago	7 minutes ago	circuit-runner	ibmq_qasm_simulator	ibmq-open/main	
6432146614b1...		Completed	7 minutes ago	7 minutes ago	circuit-runner	ibmq_qasm_simulator	ibmq-open/main	
643214459628...		Completed	8 minutes ago	8 minutes ago	circuit-runner	ibmq_qasm_simulator	ibmq-open/main	
6432142506c6...		Completed	8 minutes ago	8 minutes ago	circuit-runner	ibmq_qasm_simulator	ibmq-open/main	

Figure 13: IBMQ Wait Time

As a result, we fall back to the approach of simulating noise in the superconducting quantum processor.

## 6.2 Simulating Noise of the Real Quantum Processor

We try to simulate the noise of the superconducting quantum processor using the Qiskit Aer device [10]. Regarding the noisiness of the real device, the paper presents the energy relaxation time, dephasing time, probabilities of correct readout, and gate fidelities of each qubit in the real quantum processor, but it does not provide the gate time, which is required by Aer for simulating energy relaxation. We assume it is 130 nanoseconds, a typical time for single-qubit gates on superconducting quantum computers [6]. Because of this missing information and the fact that we are unfamiliar with the underlying mechanics of physical quantum computers, the noise model we simulate could potentially deviate significantly from the actual quantum computer.

## References

- [1] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.
- [2] James Ellis. *Quantum GANs*. 2022. URL: [https://pennylane.ai/qml/demos/tutorial\\_quantum\\_gans.html](https://pennylane.ai/qml/demos/tutorial_quantum_gans.html) (visited on 04/14/2023).
- [3] Google. *JAX: Autograd and XLA*. 2021. URL: <https://github.com/google/jax> (visited on 04/15/2023).
- [4] He-Liang Huang et al. “Experimental Quantum Generative Adversarial Networks for Image Generation”. In: *Physical Review Applied* 16.2 (Aug. 2021). doi: 10.1103/physrevapplied.16.024051. URL: <https://doi.org/10.1103/physrevapplied.16.024051>.
- [5] *IBM Quantum Computing: Access Plans*. Oct. 2015. URL: <https://www.ibm.com/quantum/access-plans>.
- [6] Norbert M Linke et al. “Experimental comparison of two quantum computing architectures”. In: *Proceedings of the National Academy of Sciences* 114.13 (2017), pp. 3305–3310.
- [7] PennyLane. *How to Create Your Own Device in PennyLane*. Xanadu AI. Sept. 2022. URL: <https://pennylane.ai/blog/2022/09/how-to-create-your-own-device-in-pennylane/> (visited on 04/15/2023).
- [8] *Qiskit plugin*. URL: <https://docs.pennylane.ai/projects/qiskit>.
- [9] Gokul Subramanian Ravi et al. “Quantum Computing in the Cloud: Analyzing job and machine characteristics”. In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2021, pp. 39–50.
- [10] *The Aer device*. URL: <https://docs.pennylane.ai/projects/qiskit/en/latest/devices/aer.html>.
- [11] Pengyuan Zhai. “Are Quantum Circuits Better than Neural Networks at Learning Multi-dimensional Discrete Data? An Investigation into Practical Quantum Circuit Generative Models”. In: *arXiv preprint arXiv:2212.06380* (2022).