ARTICLE OPEN

Check for updates

Hierarchical quantum circuit representations for neural architecture search

Matt Lourens[1✉], Ilya Sinayskiy[2,3], Daniel K. Park[4,5], Carsten Blank[6] and Francesco Petruccione[1,3,7]

Quantum circuit algorithms often require architectural design choices analogous to those made in constructing neural and tensor networks. These tend to be hierarchical, modular and exhibit repeating patterns. Neural Architecture Search (NAS) attempts to automate neural network design through learning network architecture and achieves state-of-the-art performance. We propose a framework for representing quantum circuit architectures using techniques from NAS, which enables search space design and architecture search. We use this framework to justify the importance of circuit architecture in quantum machine learning by generating a family of Quantum Convolutional Neural Networks (QCNNs) and evaluating them on a music genre classification dataset, GTZAN. Furthermore, we employ a genetic algorithm to perform Quantum Phase Recognition (QPR) as an example of architecture search with our representation. Finally, we implement the framework as an open-source Python package to enable dynamic circuit creation and facilitate circuit search space design for NAS.

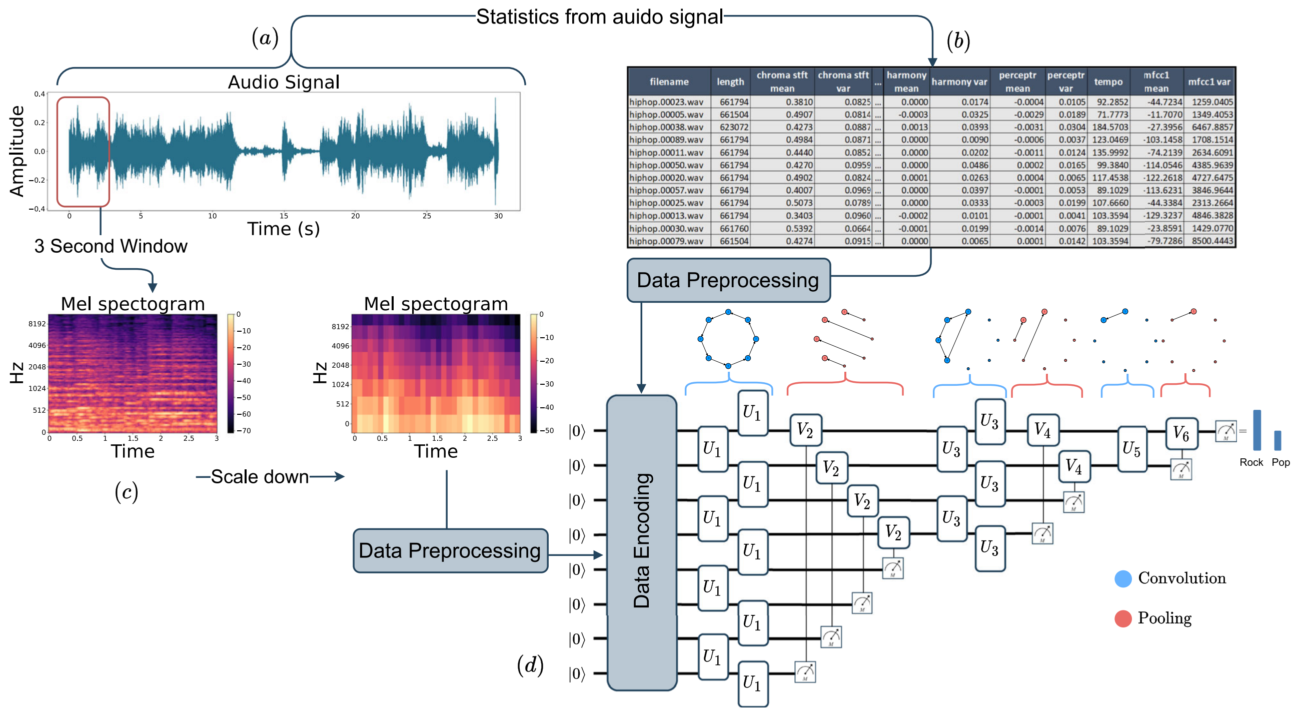
npj Quantum Information (2023)9:79 ; https://doi.org/10.1038/s41534-023-00747-z

INTRODUCTION

Machine learning using trainable quantum circuits provides promising applications for quantum computing[1–4]. Among various parameterised quantum circuit (PQC) models, the Quantum Convolutional Neural Network (QCNN) introduced in ref. [5] stands out for its shallow circuit depth, absence of barren plateaus[6], and good generalisation capabilities[7]. It has been implemented experimentally[8] and combines techniques from Quantum Error Correction (QEC), Tensor Networks (TNs) and deep learning. Research at this intersection has been fruitful, yielding deep-learning solutions for quantum many-body problems[9–12], quantum-inspired insights for deep learning[13–15] and equivalences between them[16–18]. Deep learning has been widely successful in recent years with applications spanning from content filtering and product recommendations to aided medical diagnosis and scientific research. Its main characteristic, learning features from raw data, eliminates the need for manual feature design by experts[19]. AlexNet[20] demonstrated this and marked the shift in focus from feature design to architecture design[21]. Naturally, the next step is learning network architecture, which Neural Architecture Search (NAS) aims to achieve[22]. NAS has already produced state-of-the-art deep-learning models with automatically designed architectures[21,23–25]. NAS consist of three main categories: search space, search strategy and performance estimation strategy[22]. The search space defines the set of possible architectures that a search algorithm can consider, and carefully designed search spaces help improve search efficiency and reduce computational complexity[26]. Search space design often involves encoding architectures using a cell-based representation. Usually, a set of primitive operations, such as convolutions or pooling, are combined into a cell to capture some design motif (compute graph). Different cells are then stacked to form a complete architecture. Cell-based representations are popular because they can capture repeated motifs and modular design patterns, which are often seen in successful hand-crafted architectures. Similar patterns also appear in quantum circuit designs[5,27–31]. For example, Grant et al.[27] use hierarchical architectures based on tensor networks to classify classical and quantum data. Similarly, Cong et al.[5] use the multiscale entanglement renormalisation ansatz (MERA) as an instance of their proposed QCNN and discuss generalisations for quantum analogues of convolution and pooling operations. In this work, we formalise these design patterns by providing a hierarchical representation for quantum circuits. We use the QCNN as a basis and illustrate search space design and architecture search with NAS for PQCs.

The QCNN belongs to the class of hybrid quantum-classical algorithms, in which a quantum computer executes the circuit, and a classical computer optimises its parameters. Two key factors must be considered when using PQCs for machine learning: the method of data encoding (feature map)[32,33] and the choice of a quantum circuit[34–36]. Both the challenge and objective are to find a suitable quantum circuit for a given feature map that is expressive and trainable[33]. The typical approach to finding a circuit is to keep the architecture (gates layout) fixed and to optimise continuous parameters such as rotation angles. Optimising architecture is referred to as variable structure ansatz in literature and is generally not the focus because of its computational complexity[2]. However, the architecture of a circuit can improve its expressive power and the effectiveness of initialisation techniques[28]. Also, the QCNN's defining characteristic is its architecture, which we found to impact model performance significantly. Therefore, we look towards NAS to optimise architecture in a quantum circuit setting. This approach, sometimes referred to as quantum architecture search (QAS)[37,38], has shown promising results for the variational quantum eigensolver (VQE)[39–42], the quantum approximate optimisation algorithm (QAOA)[43,44] and general architecture search[37,38,45–47]. However, these approaches are often task-specific or impose additional

[1]Physics Department, Stellenbosch University, Stellenbosch, South Africa. [2]School of Chemistry and Physics, University of KwaZulu-Natal, Durban, South Africa. [3]National Institute for Theoretical and Computational Sciences (NITheCS), Stellenbosch, South Africa. [4]Department of Applied Statistics, Yonsei University, Seoul, Korea. [5]Department of Statistics and Data Science, Yonsei University, Seoul, Korea. [6]Data Cybernetics, Landsberg, Germany. [7]School of Data Science and Computational Thinking, Stellenbosch University, Stellenbosch, South Africa. ✉email: lourensmattj@gmail.com

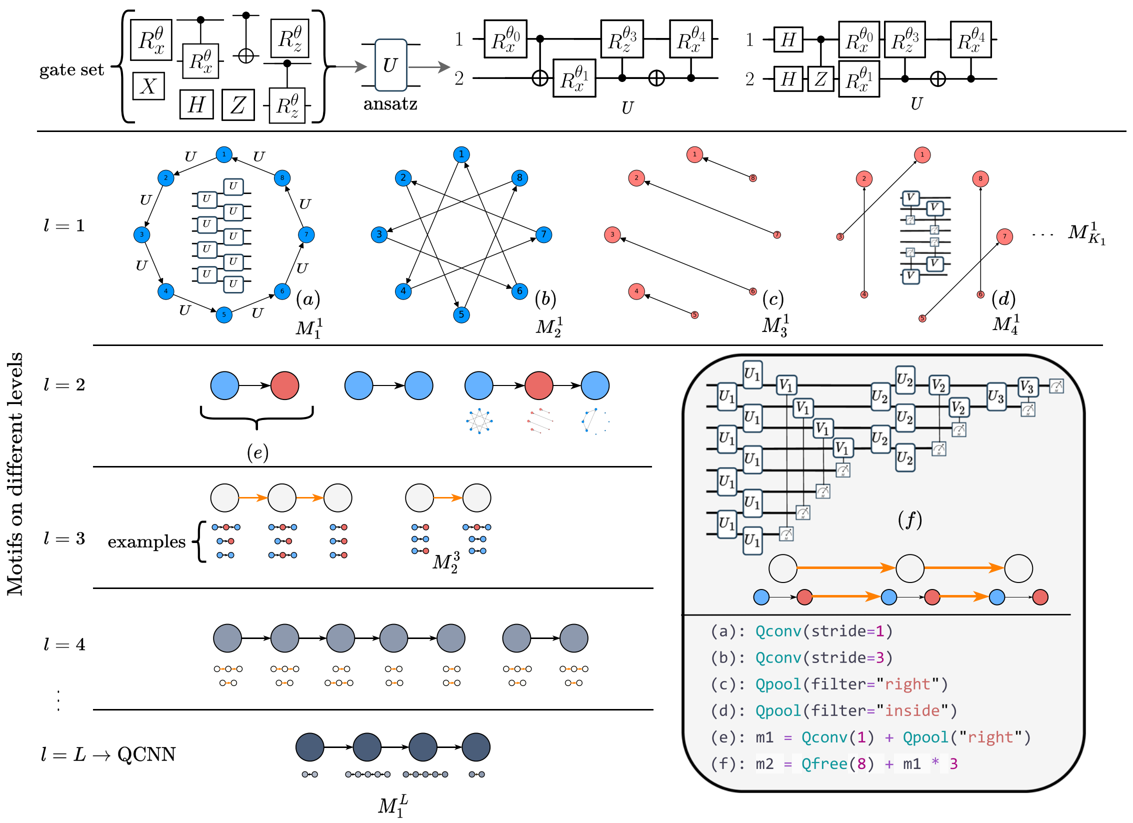


filename	length	chroma stft mean	chroma stft var	...	harmony mean	harmony var	perceptr mean	perceptr var	tempo	mfcc1 mean	mfcc1 var
hiphop.00023.wav	661794	0.3810	0.0825	...	0.0000	0.0174	-0.0004	0.0105	92.2852	-44.7234	1259.0405
hiphop.00005.wav	661504	0.4907	0.0814	...	-0.0003	0.0325	-0.0029	0.0189	71.7773	-11.7070	1349.4053
hiphop.00038.wav	623072	0.4273	0.0887	...	0.0013	0.0393	-0.0031	0.0304	184.5703	-27.3956	6467.8857
hiphop.00089.wav	661794	0.4984	0.0871	...	0.0000	0.0090	-0.0006	0.0037	123.0469	-103.1458	1708.1514
hiphop.00011.wav	661794	0.4440	0.0852	...	0.0000	0.0202	-0.0011	0.0124	135.9992	-74.2139	2634.6091
hiphop.00050.wav	661794	0.4270	0.0959	...	0.0000	0.0486	0.0002	0.0165	99.3840	-114.0546	4385.9639
hiphop.00020.wav	661794	0.4902	0.0824	...	0.0001	0.0263	0.0004	0.0065	117.4538	-122.2618	4727.6475
hiphop.00057.wav	661794	0.4007	0.0969	...	0.0000	0.0397	-0.0001	0.0053	89.1029	-113.6231	3846.9644
hiphop.00025.wav	661794	0.5073	0.0789	...	0.0000	0.0333	-0.0003	0.0199	107.6660	-44.3384	2313.2664
hiphop.00013.wav	661794	0.3403	0.0960	...	-0.0002	0.0101	-0.0001	0.0041	103.3594	-129.3237	4846.3828
hiphop.00030.wav	661760	0.5392	0.0664	...	-0.0001	0.0199	-0.0014	0.0076	89.1029	-23.8591	1429.0770
hiphop.00079.wav	661504	0.4274	0.0915	...	0.0000	0.0065	0.0001	0.0142	103.3594	-79.7286	8500.4443

Fig. 1 Machine-learning pipeline for music genre classification. The machine-learning pipeline we implemented for music genre classification. Given an audio signal of a song (**a**), we generate two forms of data: tabular (**b**) and image (**c**). Each form has data preprocessing applied before being encoded into a quantum state (**d**). The QCNN circuit shown in (**d**) favours Principal Component Analysis (PCA) because qubits are pooled from bottom to top, and principal components are encoded from top to bottom. This architecture is an instance of the reverse binary tree family that we generated with our framework.

constraints, such as circuit topology or allowable gates, to make them computationally feasible. To the best of the author's knowledge, there is currently no framework that can generate hierarchical architectures such as the QCNN without imposing such constraints.

One problem with the cell-based representation for NAS is that the macro architecture, the sequence of cells, is fixed and must be chosen[22]. Recently, Liu et al.[26] proposed a hierarchical representation as a solution, where a cell sequence acts as the third level of a multi-level hierarchy. In this representation, lower-level motifs act as building blocks for higher-level ones, allowing both macro and micro architecture to be learned. In this work, we follow a similar approach and represent a QCNN architecture as a hierarchy of directed graphs. On the lowest level are primitive operations such as convolutions and pooling. The second level consists of sequences of these primitives, such as convolution-pooling or convolution–convolution units. Higher-level motifs then contain sequences of these lower-level motifs. For example, the third level could contain a sequence of three convolution-pooling units, as seen in Fig. 1d. For the primitives, we define hyperparameters such as strides and pooling filters that control their architectural effect. This way, the representation can capture design motifs on multiple levels, from the distribution of gates in a single layer to overall hierarchical patterns such as tensor tree networks. We demonstrate this by generating a family of QCNN architectures based on popular motifs in literature. We then benchmark this family of models and show that alternating architecture has a greater impact on model performance than other modelling components. By alternating architecture, we mean the following: given a quantum circuit that consist of n unitary gates, an altered architecture consists of the same n gates rearranged in a different way on the circuit. The types of rearrangements may be changing which qubits the gates act upon, altering the order of gate occurrences, or adjusting larger architectural motifs, such as pooling specific qubits (stop using them) while leaving others available for subsequent gates and so on. We create architectural families to show the impact of alternating architecture, any two instances of the family will have the exact same unitaries, just applied in a different order on different qubits. Consider the machine-learning pipeline for classifying musical genres from audio signals, seen in Fig. 1. We start with a 30-s recording of a song (Fig. 1a) and transform it in two ways. The first is tabular form (Fig. 1b), derived from standard digital signal processing statistics of the audio signal. The second is image form (Fig. 1c), constructed using a Mel-frequency spectrogram. Both datasets are benchmarked separately, with their own data preprocessing and encoding techniques applied. For the tabular data, we test Principal Component Analysis (PCA) and tree-based feature selection before encoding it in a quantum state using either qubit, IQP, or amplitude encoding. Once encoded, we choose two-qubit unitary ansatzes (Supplementary Fig. 1) U_m and V_m for the convolution and pooling primitives $m = 1, 2, \ldots, 6$, as shown in Fig. 1d. We test them across different instances of an architecture family. Of all the components in this pipeline, alternating architecture, that is changing how each U_m and each V_m are spread across the circuit, had the greatest impact on model performance. In addition to our theoretical framework, we implement it as an open-source Python package to enable dynamic QCNN creation and facilitate search space design for NAS. It allows users to experimentally determine suitable architectures for specific modelling setups, such as finding circuits that perform well under a specific noise or hardware configuration, which is particularly relevant in the Noisy Intermediate-Scale Quantum (NISQ)[48] era. In addition, as more qubits become available, the hierarchical nature of our framework provides a natural way to scale up the same model. In summary, our contributions are the architectural representation for QCNNs, a Python package for dynamic QCNN creation, and experimental results on the potential advantage of architecture search in a quantum setting.



```
(a): Qconv(stride=1)
(b): Qconv(stride=3)
(c): Qpool(filter="right")
(d): Qpool(filter="inside")
(e): m1 = Qconv(1) + Qpool("right")
(f): m2 = Qfree(8) + m1 * 3
```

Fig. 2 Hierarchical quantum circuit representation. An overview of our architectural representation for QCNNs. From a given set of gates, we build two-qubit unitary ansatzes. The representation then captures design motifs M_k^l on different levels l of the hierarchy. On the lowest level $l = 1$, we define primitives which act as building blocks for the architecture. For example, a convolution operation with stride one is encoded as the directed graph M_1^1. The directed graph M_3^1 is a pooling operation that measures the bottom half of the circuit. Combined, they form the level two motif (e): a convolution-pooling unit M_1^2. Higher-level motifs consist of combinations of lower-level motifs up until the final level $l = L$, which contains only one motif M_1^L, the complete QCNN architecture. M_1^L is a hierarchy of directed graphs fully specifying how to spread the unitary ansatzes across the circuit. The two lines of code (*e*) and (*f*) show the power of this representation as it is all that is required to create the entire QCNN circuit from Fig. 1d. The code comes from the Python package we implemented based on the work of this paper. It facilitates dynamic QCNN creation and search space design.

The remainder of this paper is structured as follows: we begin with our main results by summarising the architectural representation for QCNNs and then show the effect of alternating architecture, justifying its importance. We then provide an example of architecture search with our representation by employing an evolutionary algorithm to perform QPR. Following this, we give details of our framework by providing a mathematical formalism for the representation and describing its use. Next, with the formalism at hand, we show how it facilitates search space design by describing the space we created for the benchmark experiments. We then discuss generalisations of formalism and the applicability of our representation with search algorithms. After this, we elaborate on our experimental setup in "Methods". Finally, we discuss applications and future steps.

RESULTS

Architectural representation

Figure 2 shows our architectural representation for QCNNs. We define two-qubit unitary ansatzes from a given set of gates, and capture design motifs M_k^l on different levels l of the hierarchy. On the lowest level $l = 1$, we define primitives which act as building blocks for the architecture. For example, a convolution operation with stride one is encoded as the directed graph M_1^1, and with stride three as M_2^1. The directed graph M_3^1 is a pooling operation that measures the bottom half of the circuit, and M_4^1 measures from the inside outwards. Combined, they can form higher-level motifs such as convolution-pooling units M_1^2 (e), convolution–convolution units M_2^2, or convolution-pooling-convolution units M_3^2. The highest level $l = L$ contains only one motif M_1^L, the complete QCNN architecture. M_1^L is a hierarchy of directed graphs fully specifying how to spread the unitary ansatzes across the circuit. This hierarchical representation is based on the one from ref. [26] for deep neural networks (DNNs), and allows for the capture of modularised design patterns and repeated motifs. The two lines of code (*e*) and (*f*) show the power of this representation as it is all that is required to create the entire QCNN circuit from Fig. 1d. The code comes from the Python package we implemented based on the work of this paper. It facilitates dynamic QCNN creation and search space design.

Architectural impact

The details regarding specific notation and representation of the framework are given after this section, first we justify it with the following experimental results. We also give background on QCNNs in Supplementary Note 1 and on quantum machine learning in Supplementary Note 2 for more context. To illustrate the impact of architecture on model performance, we compare the fixed architecture from the experiments of ref. [29] to other architectures in the same family while keeping all other components the same. The only difference in each comparison is architecture (how the unitaries are spread across the circuit). The

Table 1. Architectural vs ansatz impact.

Architecture vs ansatz

Ansatz, # Params		Architecture: Reference	New alteration	Δ	Alteration (s_c, F^*, s_p)
a	6	65.37 ± 2.8	**75.14** ± 1.7	+ 9.77	(6, left, 2)
b	6	56.34 ± 3.2	**70.46** ± 1.0	+ 14.12	(1, odd, 3)
c	12	52.69 ± 3.8	**70.74** ± 1.3	+ 18.05	(1, odd, 0)
d	18	67.13 ± 1.5	**77.87** ± 2.4	+ 9.87	(1, outside, 2)
e	18	67.87 ± 2.5	**73.61** ± 1.8	+ 5.74	(6, left, 0)
f	18	69.21 ± 2.6	**74.80** ± 2.8	+ 5.59	(1, left, 3)
g	30	73.24 ± 2.9	**79.47** ± 2.2	+ 6.23	(2, left, 1)
h	30	69.35 ± 4.1	**71.71** ± 3.7	+ 2.36	(2, left, 1)

The average accuracy and standard deviation of the country vs rock genre pair on a held-out test set after 30 separate trained instances. The best performing architecture for each ansatz is highlighted in bold. All architectures come from the family of reverse binary trees, generated with algorithm 1. The "reference" architecture is the one used in the experiments of ref. [29] and the "alteration" was found through random search within the same family. The unitary ansatzes (a–h) are shown in Supplementary Fig. 1.

architecture in ref. [29] is represented within our framework as: $(s_c, F^*, s_p) = (1, \text{even}, 0) \mapsto \text{Qfree}(8) + (\text{Qconv}(1) + \text{Qpool}(0, F^{even}))$ 3, 3, see Algorithm 1. To evaluate their performance, we use the country vs rock genre pair, which proved to be one of the most difficult classification tasks from the 45 possible combinations. We compare eight unitary ansatzes with different levels of complexity, shown in Supplementary Fig. 1.

Table 1 shows the results of the comparisons, the reference architecture is as described above and the discovered alteration found via random search. We note the first important result, we improved the performance of every ansatz, in one case, by 18.05%, through the random search of the architecture space. Ansatz refers to the two-qubit unitary used for the convolution operation of a model. For example, the model in Fig. 1d is described by (1, right, 0) and ansatz (a) corresponds to U_1, U_2 and U_3 being the circuit in Supplementary Fig. 1a. Each value represents the average model accuracy and standard deviation from 30 separate trained instances on the same held-out test set.

The second important result is that alternating architecture can improve model performance without increasing complexity. For instance, the best-performing model for the reference architecture is with ansatz (g), which has an average accuracy of 73.24%. However, this ansatz causes the model to have $10 \times 3 = 30$ parameters. In contrast, by alternating the architecture with the simplest ansatz (a), the model outperformed the best reference model with an average accuracy of 75.14% while only having $3 \times 2 = 6$ parameters. The parameter counts come from each model having $N = 8$ qubits and the same number of unitaries, $3N - 2 \to 3(8) - 2 = 22$, of which 13 are for convolutions. See "Search space design" and Algorithm 1 for more details. A model has three convolutions, and each convolution shares weights between its two-qubit unitaries. This means that the two-qubit unitary ansatz primarily determines the number of parameters to optimise for a model. For example, a model with ansatz (a) have $2 \times 3 = 6$ parameters to optimise because the ansatz has two parameters.

Another interesting result is for ansatz (c), the reference architecture could only obtain an average accuracy of 52.69%, indicating its inability to find any kind of local minimum during training, leading one to think it might be a barren plateau. But, the altered architecture was able to find a local minima and improve the average accuracy by 18.05%.

We would like to note that our primary objective in these experiments is to demonstrate the potential for performance improvement. As such, we only conducted random search for ~2 h on an i7-1165G7 processor for each ansatz. Consequently, for higher parameter ansatzes, which correspond to longer training times, the search space was less explored. This is likely the reason behind the observed decrease in performance improvement for larger parameter ansatzes. Therefore the observed improvements are all lower bounds for the potential performance increase from alternating architecture. We anticipate that significantly better architectures may still exist within the space. Table 2 presents the performance of the family of reverse binary trees (as described in Algorithm 1) for ansatz (a). Due to its quick training time, ansatz (a)

Table 2. Search space of 1a.

Performance across architecture search space

F^*, s_p	Convolution stride, s_c: 1	2	3	4	5	6	7	Avg
Even	**67.01**	63.63	60.76	64.93	59.98	63.1	59.49	62.81
0	65.97	58.68	56.25	66.67	62.85	59.72	63.43	61.88
1	66.32	66.32	63.54	60.07	61.46	71.88	54.17	63.73
2	66.67	60.76	60.07	68.06	54.17	58.8	63.89	61.81
3	69.1	68.75	63.19	64.93	61.46	60.19	56.48	63.84
Inside	66.41	**71.96**	58.25	54.25	69.27	68.15	60.53	64.18
0	65.28	72.22	60.07	49.65	70.49	68.4	60.65	63.94
1	67.01	71.18	58.68	55.9	66.32	68.4	60.19	64.09
2	68.4	71.53	58.33	51.74	71.88	68.98	58.8	64.26
3	64.93	72.92	55.9	59.72	68.4	66.67	62.5	64.42
Left	62.85	61.63	59.38	59.03	51.56	**72.52**	72.45	62.22
0	66.67	67.01	56.94	61.46	52.08	71.18	73.61	63.79
1	59.03	62.15	52.78	57.99	52.08	71.18	73.61	60.8
2	63.19	63.19	63.19	60.76	51.74	**75.93**	71.76	63.51
3	62.5	54.17	64.58	55.9	50.35	72.69	70.83	60.79
Odd	61.11	**68.75**	63.37	62.76	64.67	60.52	57.99	62.96
0	60.76	71.88	63.19	58.33	63.54	59.38	57.87	62.29
1	63.54	67.36	64.58	63.54	64.24	62.5	59.26	63.73
2	60.42	70.14	64.58	65.97	69.1	58.8	56.94	64.16
3	59.72	65.62	61.11	63.19	61.81	61.11	57.87	61.65
Outside	60.68	65.8	65.54	57.12	62.15	59.83	**67.13**	62.51
0	67.36	59.72	71.88	54.17	67.01	60.07	70.37	64.15
1	53.47	69.79	62.15	56.25	61.11	58.33	70.83	61.49
2	57.99	70.83	60.07	61.11	59.03	59.26	66.67	62.07
3	63.89	62.85	68.06	56.94	61.46	61.57	60.65	62.29
Right	**70.05**	65.63	64.41	53.65	68.66	63.69	60.65	63.94
0	70.14	63.54	64.58	50	68.4	61.11	62.96	62.96
1	69.79	67.71	64.58	69.1	68.06	67.01	57.87	66.62
2	70.14	62.15	63.89	43.75	68.75	62.04	61.57	61.75
3	70.14	69.1	64.58	51.74	69.44	64.35	60.19	64.37
Avg	64.68	66.23	61.95	58.62	62.72	64.69	63.04	63.11

Country vs Rock average accuracy within the reverse binary tree search space, all with the ansatz from Supplementary Fig. 1a. The convolution stride s_c is shown on the horizontal axis and the combinations of pooling filter F^* and stride s_p on the vertical. The best pooling filter and convolution stride combinations are presented in bold along with the overall best architecture $(s_c, F^*, s_p) = (6, \text{left}, 2)$.

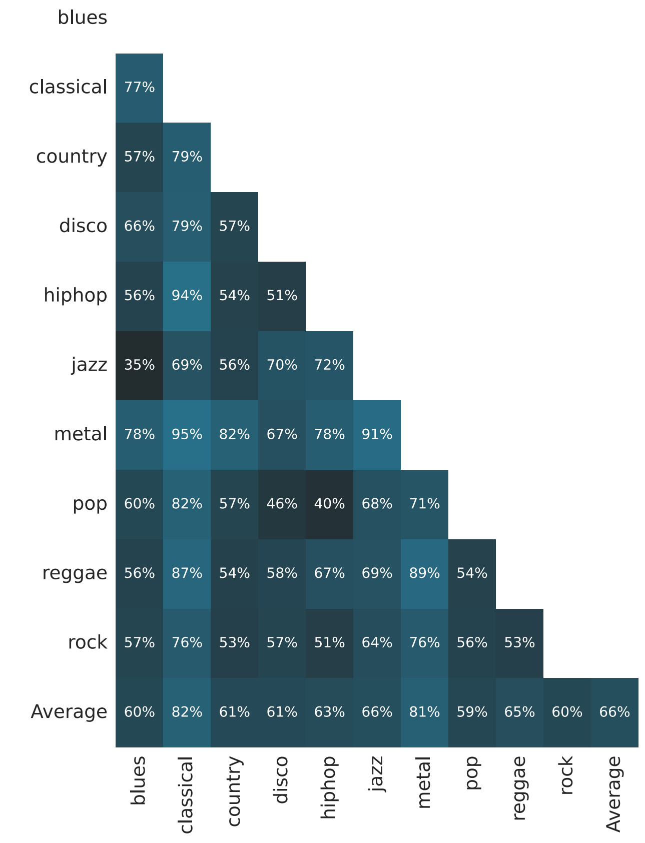


Fig. 3 $\boldsymbol{F_m^{\mathrm{right}}}$ **pooling filter results.** QCNN with the F_m^{right} pooling filter using low-resolution image data. The accuracies for all genre pairs are provided.

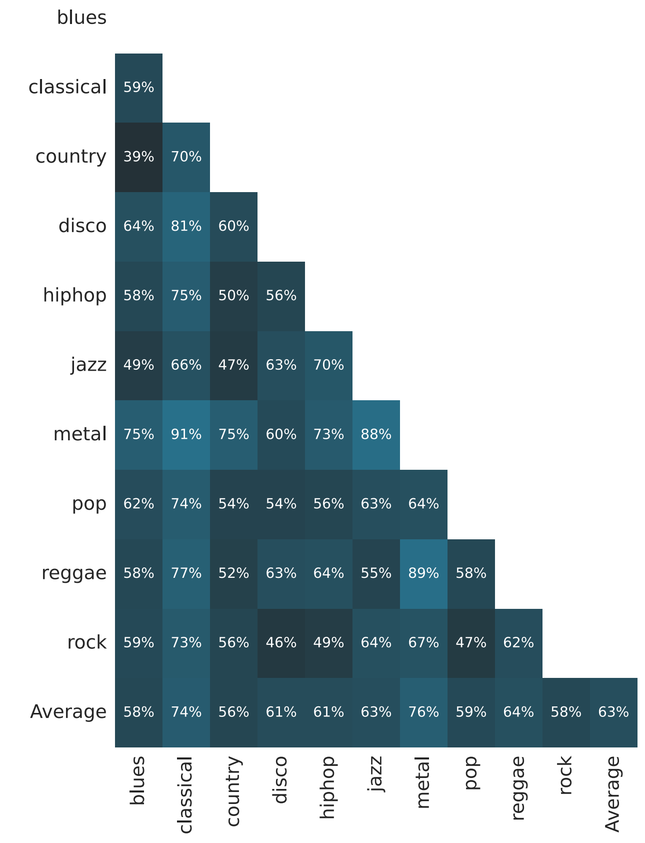


Fig. 4 $\boldsymbol{F_m^{\mathrm{even}}}$ **pooling filter results.** QCNN with the F_m^{even} pooling filter using low-resolution image data. The accuracies for all genre pairs are provided.

was the only case for which we managed to exhaust the search space (168 architectures). In "Search space design", we discuss how the size of the family can be easily increased or decreased. Each value represents the average accuracy of five trained instances on the country vs rock genre pair. The overall accuracy of the whole space is 63.11%, indicating that the reference architecture from Table 1 was close to the mean performance. The best-performing architecture in this space is $(s_c, F^*, s_p) = (6, \mathrm{left}, 2)$, with an average accuracy of 75.93%. This is the alteration from Table 1 discovered through random search within the family of reverse binary trees. It seems that the combination of F^{left} and $s_c = 6$ performs particularly well for this task, with an average accuracy of 72.52%. In general, it appears that the convolution stride s_c and pooling filter F^* have the most significant impact on performance. It is also worth noting that convolution strides of $s_c = 3, 4, 5$ performed poorly compared to the other values. The range of performance in this space goes from a minimum of 43.75% to a maximum of 75.93%, demonstrating the potential impact of architectural choices on model performance.

Finally, we compared the performance of two different architectures on the image data across all genres. This time, we used ansatz (g) to compare the F_m^{right} and F_m^{even} pooling filters, shown in Figs. 3 and 4. The image data is a low-resolution ($8 \times 32 = 256 = 2^8$ pixels) spectrogram of the audio signal. We did not expect high accuracy from this data, but were interested in the variation of performance for different architectures. Figures 3 and 4 show the difficulty of some genre pairs. Interestingly, the F_m^{right} pooling filter outperformed the F_m^{even} filter on almost all genres. If we focus on the genre pairs that the models were able to classify, we see that F_m^{right} had 14 models that achieved an accuracy above 75%, compared to the 5 of F_m^{even}. We also note that the image data had no PCA or tree-based feature selection applied to it, and the F_m^{right} filter was still favoured. A similar result was obtained with ansatz (a). This shows architecture impacts performance even on low-resolution data.

Architectural search

In this section, we present an example of applying our architectural representation in conjunction with evolutionary search to perform Quantum Phase Recognition (QPR). The specifics of the search algorithm can be found in "Generalisation and search" but we utilise an algorithm similar to the one employed in ref. [26]. Mutations involve replacing a primitive within a motif with a randomly generated one, while crossover consists of combining two motifs end-to-end, if possible, or interweaving them otherwise. To facilitate comparison, we consider the same task and setup from the original QCNN paper[5]. The objective is to recognise a $\mathbb{Z}_2 \times \mathbb{Z}_2$ symmetry-protected topological (SPT) phase for a ground state that belongs to a family of cluster-Ising Hamiltonians[49]:

$$H = -J\sum_{i=1}^{N-2} Z_i X_{i+1} Z_{i+2} - h_1 \sum_{i=1}^{N} X_i - h_2 \sum_{i=1}^{N-1} X_i X_{i+1}. \tag{1}$$

Here, X_i, Z_i are Pauli operators acting on the spin at site i and the SPT phase contains a $S = 1$ Haldane chain[50]. The ground state can belong to an SPT, paramagnetic or antiferromagnetic phase depending on the values of h_1, h_2, and J. Our goal is to identify a QCNN capable of distinguishing between SPT and other phases by measuring a single qubit. Following the approach in ref. [5], we consider a system of $N = 15$ spins and train a circuit on 40 equally spaced points along $h_2 = 0$, where the ground state is known to be in the SPT phase when $h_1 \leq 1$. We also evaluate the circuit with

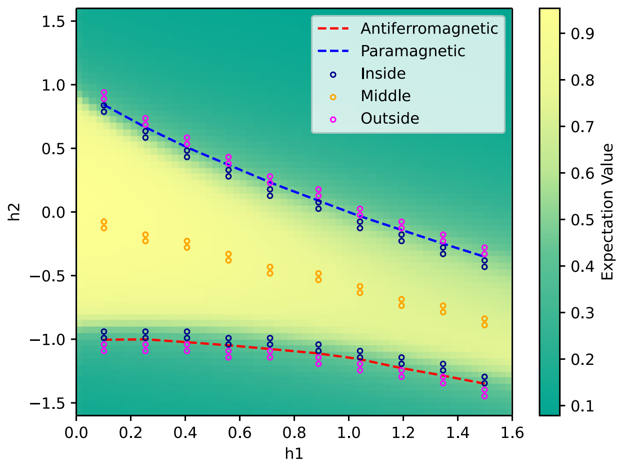


Fig. 5 Quantum phase recognition result. Expectation values for the circuit found via evolutionary search for a system of $N = 15$ spins. Points represent a test set of 64×64 ground states for various h_1 and h_2 values of the Hamiltonian, $J = 1$. The inside, middle and outside points were used to evaluate an architecture's fitness during search. The same colour scale as in[5] is used to facilitate comparison.

the same sample complexity[5]:

$$M_{\min} = \frac{1.96^2}{(\arcsin\sqrt{p} - \sqrt{\arcsin p_0})} \tag{2}$$

where P represents the probability of measuring a non-zero expectation value and $P_0 = 0.5$. Equation (2) calculates the minimum number of measurements required to be 95% confident that $P \neq 0.5$, with P being the expectation value of the circuit U encoded with the ground state $|\psi_g\rangle$ transformed into a probability: $p = (\langle\psi_g|U|\psi_g\rangle + 1)/2$. Therefore a well-performing QCNN will yield low values of $M_{\min}$ near the phase boundary for points within the SPT phase. We define the fitness of an architecture as a linear combination of the sample complexity values $M_{\text{in}}, M_{\text{middle}}$ for points in the SPT phase, and the mean squared error MSE_{out} for points outside the boundary. Figure 5 illustrates the points considered for M_{in}, M_{middle} and MSE_{out}. During search we assigned the majority of the weight to M_{in} as the goal is to develop a model that confidently identifies SPT phases near the boundary. To prevent a model from classifying all points as SPT, MSE_{out} is included, while M_{middle} ensures overall good performance. Finally, during search we added a regularisation term for the number of parameters, to find well-performing architectures with low computational complexity.

Table 3 and Fig. 5 show the performance of the best architecture found during search. The search algorithm identified a QCNN with only 11 parameters, in contrast to the 1308 parameters of the original reference architecture. For points in the SPT phase near the boundary, the sample complexity of the discovered architecture ($M_{\text{in}} = 36.079$) is lower than that of the reference (61.523), resulting in 25 fewer measurements required on average. Although the reference architecture exhibits slightly better sample complexity for points in the middle of the phase boundary ($M_{\text{middle}} = 10.992$) compared to the discovered architecture ($M_{\text{middle}} = 13.253$), and a marginally lower MSE for points outside the phase boundary (MSEout = 0.164 compared to MSEout = 0.167), the improvements in M_{in} and the number of parameters are substantial and more advantageous. The discovered architecture is shown in Supplementary Fig. 2, and the phase diagram it generates is shown in Fig. 5. The search was conducted on a system equipped with two Intel Xeon E5-2640 processors (2.0 GHz) and 128 GB of RAM, and it took ~2 h to discover the final architecture (over 831 generations). Although we anticipate that extending the search may yield even better architectures, the primary goal of this experiment was to demonstrate a representative example of the search process and showcase the ease of obtaining promising results. This emphasises the potential advantages of architecture search in quantum computing tasks, where the computational cost of a circuit can be reduced while maintaining or even improving performance. We attribute this success to a well-defined search space, with our representation aiming to simplify the process of creating such spaces. Moreover, our representation allows for the incorporation of hardware constraints, facilitating the search for architectures that perform well on specific quantum devices. We believe this to be a necessary step towards the development of efficient quantum algorithms for real-world applications. By employing a well-structured representation and search space, we can streamline the process of discovering optimised quantum circuit architectures that are better suited for specific tasks and hardware.

Table 3. Performance of architecture found with an evolutionary search.

Metric	Reference	Found
Number of parameters	1308	**11**
Sample complexity (inside)	61.523	**36.079**
Sample complexity (middle)	**10.992**	13.253
MSE (outside)	**0.164**	0.167

Different performance metrics (lower is better) for the 15-qubit QCNN from ref. [5] and the architecture found via evolutionary search. The best performing architecture for each metric is highlighted in bold. Sample complexity represents the expected number of measurements required to be 95% confident that the ground state is in the SPT phase (non-zero expectation value). Metrics are calculated on a set of points in the test set, where inside refers to SPT points near the phase boundary, outside to non-SPT points near the phase boundary and middle to points in between, as shown in Fig. 5.

Digraph formalism

We represent the QCNN architecture as a sequence of directed graphs, each acting as a primitive operation such as a convolution (Qconv) or pooling (Qpool). A primitive is the directed graph $G = (Q, E)$; its nodes Q represent available qubits, and oriented edges E the connectivity of the unitary applied between a pair of them. The direction of an edge indicates the order of interaction for the unitary. For example, a CNOT gate with qubit i as control and j as target is represented by the edge from qubit i to qubit j. We also introduce other primitives, such as Qfree, that free up pooled qubits for future operations. The effect of a primitive is based on its hyperparameters and the effect of its predecessor. This way, their individual and combined architectural effects are captured, enabling them to be dynamically stacked one after another to form the second level $l = 2$ motifs. Stacking these stacks in different ways constitutes higher-level motifs until a final level $l = L$, where one motif constitutes the entire QCNN architecture. In the case of pooling, controlled unitaries are used in place of measurement due to the deferred measurement principle[51]. We define a QCNN architecture in Definition 1.

Definition 1. The $k = 1, 2, \ldots K_l$ motif on level $l = 1, 2, \ldots, L$ is the tuple $M_k^l = (M_j^{l-1} | j \in \{1, 2, \ldots, K_{l-1}\})$. Motifs on the lowest level M_k^1 are primitive operations, which form the set $M^{(1)} = \{M_1^1, M_2^1, \ldots, M_{K_1}^1\}$. At the highest level $l = L$ there is only one motif M_1^L which is a hierarchy of tuples. M_1^L is flattened through an assemble operation: $M = \text{assemble}(M_1^L)$ which encodes each primitive into a directed graph $G_m = (Q_m, E_m)$, the nodes Q_m are available qubits and edges E_m the connectivity of

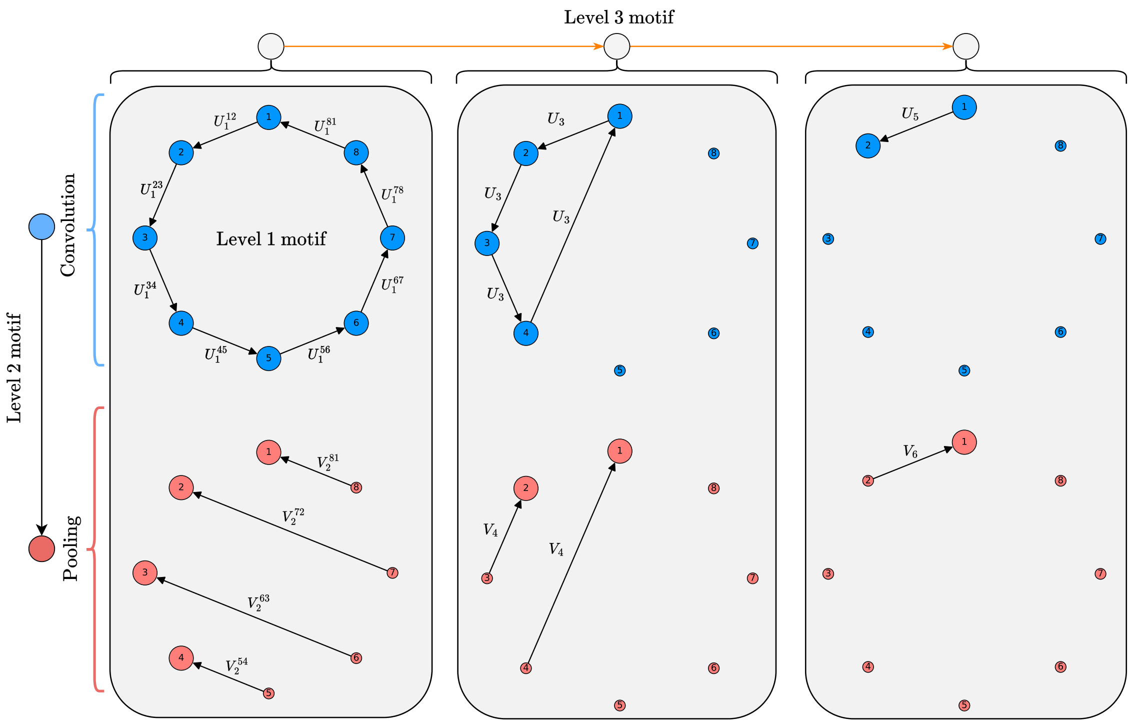


Fig. 6 Directed graph view of Fig. 1d. Graph view for the circuit architecture in Fig. 1d. The same two-qubit unitary is used in all layers for the convolution operation, i.e., $U_m^{ij} = U_m$. Similarly, in this example, we use the same two-qubit pooling unitaries $V_m^{ij} = V_m$. The top left graph is $G_1 = (Q_1^c, E_1^c)$ with all eight qubits Q_1^c available for the convolution operations $U_1^{ij}, (i,j) \in E_1^c$. Below G_1 is G_2 with half the qubits of Q_2^p measured, indicated by the ith indices of $V_m^{ij}, (i,j) \in E_2^p$. For example, qubit $8 \in Q_2^p$ is measured and V_2 applied to qubit $1 \in Q_2^p$ as indicated by $V_2^{81}, (8,1) \in E_2^p$. This pattern repeats until one qubit remains in G_6, which is measured and used to classify the music genre.

unitaries applied between them. M describes the entire QCNN architecture, $M = (G_1, G_2, \ldots, G_{|M|})$.

Figure 2 shows example motifs on different levels for a QCNN. Higher-level motifs are tuples, and the lowest-level ones directed graphs. The dependence between successive motifs is specified in Definition 2.

Definition 2. Let $x \in \{c, p, f\}$ indicate the primitive type for {Qconv, Qpool, Qfree} and M_1^L be the highest level motif for a QCNN. Then assemble(M_1^L) flattens depth-wise into $M = (G_1, G_2, \ldots, G_{|M|})$ where $G_m = (Q_m^x, E_m^x)$. G_1 is always a Qfree(N_q) primitive specifying the number of available qubits with N_q. For $m > 1$, G_m is defined as:

$$\text{If } G_m \text{ is a Qfree}(N_f) \text{ primitive then :}$$

$$\begin{aligned} Q_m^f &= \{1, 2, \ldots, N_f\}, \\ E_m^f &= \{\}. \end{aligned}$$

$$\text{If } G_m \text{ is a convolution primitive :}$$

$$\begin{aligned} Q_m^c &= \begin{cases} Q_{m-1}^x & \text{if } x \in \{c, f\}, \\ Q_{m-1}^x \setminus \{i \in (i,j) \in E_{m-1}^x\} & \text{if } x = p, \end{cases} \\ E_m^c &= \{(i,j) | (i,j) \in Q_m^c \times Q_m^c\}. \end{aligned}$$

$$\text{If } G_m \text{is a pooling primitive :}$$

$$\begin{aligned} Q_m^p &= \begin{cases} Q_{m-1}^x & \text{if} x \in \{c, f\}, \\ Q_{m-1}^x \setminus \{i \in (i,j) \in E_{m-1}^x\} & \text{if } x = p, \end{cases} \\ E_m^p &= \{(i,j) | (i,j) \in Q_m^p \times Q_m^p, i \neq j, \\ d^-(i) &= 0, d^+(i) = 1, d^-(j) \geq 1, d^+(j) = 0\}. \end{aligned}$$

with $d^-(i)$ and $d^+(i)$ referring to the indegree and outdegree of node i, respectively, and $\setminus$ to set difference.

We show this digraph perspective in Fig. 6, it is the data structure of the circuit in Fig. 1d. If the mth graph in M is a convolution, we denote its two-qubit unitary acting on qubit i and j as $U_m^{ij}(\theta)$. Similarly, for pooling, we notate the unitary as $V_m^{ij}(\theta)$. The action of $V_m^{ij}(\theta)$ is measuring qubit i (the control), which causes a unitary rotation V on qubit j (the target). With this figure and notational scheme in mind, Definition 2 reads as follows:

Q_m^x is the set of available qubits for the mth primitive in M, where $x \in \{c, p, f\}$ for convolution, pooling or Qfree respectively. The first primitive G_1 is Qfree(N_q) which specifies the number of available qubits N_q for future operations. Any proceeding $m > 1$ primitive G_m only has access to qubits not measured up to that point. This is the previous primitive's available qubits Q_{m-1}^x if its type $x \in \{c, f\}$ is a convolution or Qfree. Otherwise, for pooling, $x = p$, it's the set difference: $Q_{m-1}^x \setminus \{i \in (i,j) \in E_{m-1}^x\}$ since the i indices during pooling $(i,j) \in E_m^p$ indicates measured qubits. This is visualised as small red circles in Fig. 6. The only way to make those qubits available again is through Qfree(N_f), which can be used to free up N_f qubits. For the convolution primitive, E_m^c is the set of all pairs of qubits that have $U_m^{ij}(\theta)$ applied to them. Finally, for the pooling primitive, E_m^p is the set of pairs of qubits that have pooling unitaries $V_m^{ij}(\theta)$ applied to them. The restriction is that if qubit i is measured, it cannot have any other rotational unitary V applied to it within the same primitive G_m. This means the indegree d^- of node i is zero. Similarly, if qubit i is measured, it may only have one corresponding target, meaning that the outdegree d^+ of node i is one. In the same vein, no target qubit j can be the control for another, $d^+(j) = 0$. Every target qubit j have at least one corresponding control qubit i, $d^-(j) \geq 1$. It is possible for multiple measured qubits to have the same target qubit, giving E_m^p a surjective property.

Following this definition, we can express a convolution or pooling operation for the mth graph in M as:

$$\widetilde{U}_m = \prod_{(i,j)\in E_m^c} U_m^{ij}(\theta), \tag{3}$$

$$\widetilde{V}_m = \prod_{(i,j)\in E_m^p} V_m^{ij}(\theta). \tag{4}$$

Let $\widetilde{W}_m = \widetilde{U}_m$ or $\widetilde{V}_m$ be the mth primitive in M based on whether it's a convolution or pooling and the identity I if it's a Qfree primitive. Then the state of the QCNN after one training run is:

$$|\psi\rangle = \widetilde{W}_{|M|} \cdots \widetilde{W}_4 \widetilde{W}_3 \widetilde{W}_2 \widetilde{W}_1 U_{\text{encoding}} |0\rangle. \tag{5}$$

We note that the choice of V is unrestricted, which means that within one layer each V can be a different rotation. Figure 1d shows a special case where the same V is used per layer, which is computationally favourable compared to using different ones. To enable weight sharing, the QCNN require convolution unitaries to be the same i.e., $U_m^{ij} = U_m^{kh}$ where $(i,j) \in, (k,h) \in E_m^c$. This formulation only regards one and two-qubit unitaries for convolutions, one-qubit unitaries being described with $E_m^c = (i,i), i \in Q_m^c$. In "Generalisation and search", we extend it to multiple qubit unitaries.

After training, $|\psi\rangle$ in Eq. (5) is measured based on the type of classification task, in this work, we focus on binary classification allowing us estimate $\hat{y}$ by measuring the remaining or specified qubit in the computational basis:

$$\hat{y} = P(y=1) = |\langle 1||\psi\rangle|^2. \tag{6}$$

We note that multi-class classification is also possible by measuring the other qubits and associating each with a different class outcome. Following this, we calculate the cost of a training run with $C(y,\hat{y})$, then using numerical optimisation the cost is reduced by updating the parameters from Eqs. (3) to (4) and repeat the whole process until some local minimum is reached. Resulting in a model alongside a set of parameters to be used for classifying unseen data.

Controlling the primitives

We define basic hyperparameters that control the individual architectural effect of a primitive. There are two broad classes of primitives, special and operational. A special primitive has no operational effect on the circuit, such as Qfree. Its purpose is to make qubits available for future operational primitives and therefore has one hyperparameter N_f for this specification. N_f is typically an integer or set of integers corresponding to qubit numberings:

$$Q_m^f = \{1, 2, \ldots, N_f\} \text{ if } N_f \text{ is an integer}, \tag{7}$$

$$Q_m^f = N_f \text{ if } N_f \text{ is a set of integers}. \tag{8}$$

Each operational primitive has its own stride parameter analogous to classical CNNs. For a given stride s, each qubit gets paired with the one s qubits away modulo the number of available qubits. For example, a stride of 1 pairs each qubit with its neighbour. This depends on the qubit numbering used which is based on the circuit topology. For illustration purposes, we use a circular topological ordering, but any layout is possible as long as some ordering is provided for Q_1^f. For the convolution primitive, we define its stride $s_c \in \{1,2,3,\ldots\}$ as:

$$E_m^c = \{(i, (i+s_c) \bmod |Q_m^c|) | i \in Q_m^c\} \text{ if } |Q_m^c| > 2, \tag{9}$$

$$E_m^c = \{(i,j) | i,j \in Q_m^c, i \neq j\} \text{ if } |Q_m^c| = 2, \tag{10}$$

$$E_m^c = \{(i,i) | i \in Q_m^c\} \text{ if } |Q_m^c| = 1. \tag{11}$$

Equation (10) captures the case where there are only two qubits available for a convolution and Eq. (11) when there is only one which implies the convolution unitaries only consist of single qubit gates. A stride of $s_c = 1$ is a typical design motif for PQCs and the graph formalism allow for a simple way to capture and generalise it. To achieve translational invariance for all strides the two constraints: $|E_m^c| = |Q_m^c|$ and $(i,j) \neq (k,h)$ where $(i,j) \in, (k,h) \in E_m^c$ are added. Another option for translational invariance is a Qdense primitive, which only differs from Qconv in that E_m^c generates all possible pairwise combinations of Q_m^c. This primitive is available in the python package but left out from the definition (because of its similarity).

Figure 7 shows different ways in which s_c generate E_m^c for $|Q_m^c| = 8$.

The pooling primitive has two hyperparameters, a stride s_p and filter F_m^*. The filter indicates which qubits to measure and the stride how to pair them with the qubits remaining. We define the filter as a binary string:

$$F_m^* = w_1 w_2 \cdots w_{|Q_m^p|} \begin{cases} w_i &= 1 \text{ if qubit } i \text{ is measured}, \\ w_i &= 0 \text{ otherwise}. \end{cases} \tag{12}$$

For $N=8$ qubits, the binary string $F_m^* = 00001111$ translates to measuring the rightmost qubits, i.e., $\{i | i \in Q_m^p, i \geq 5\}$. Figure 6 is an example where the pattern $F_2^* = 00001111 \to F_4^* = 0011 \to F_6^* = 01$ is used, visually the qubits are removed from bottom to top. Encoding filters as binary strings is useful since generating them becomes generating languages, enabling the use of computer scientific tools such as context free grammars and regular expressions to describe families of filters. Pooling primitives enable hierarchical architectures for QCNNs, and in "Search space design", we illustrate how they can be implemented to create a family resembling reverse binary trees. The action of the filter is expressed as: $F_m^* \star Q_m^p = Q_{m+1}^x$ where $\star$ slices Q_m^p corresponding to the 0 indices of F_m^*, i.e., $w_i = 0$(not measured). For example $010 \star \{4,7,2\} = \{4,2\}$. This example

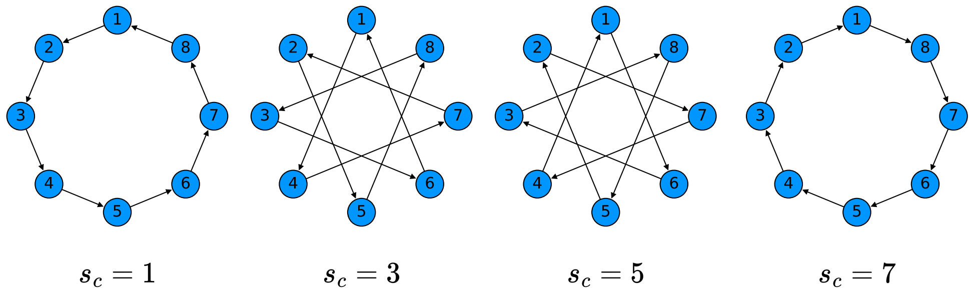


Fig. 7 Visualisation of different convolution strides. Diagram showing how changing the convolution stride s_c generates different configurations for E_m^c.

illustrates the case where an ordering was given to the set of available qubits to represent some specific topology of the circuit. Let $Q^x_{m+1} = F^*_m \star Q^p_m$ then the pooling primitive stride $s_p = \{1, 2, \dots\}$ is defined as:

$$E^p_m = \{(i, (j + s_p) \bmod |Q^x_{m+1}|) | i \in Q^p_m \setminus Q^x_{m+1}, j \in Q^x_{m+1}\}. \tag{13}$$

Search space design

We show how the digraph formalism facilitates QCNN generation and search space design. Grant et al.[27] exhibited the success of hierarchical designs that resemble reverse binary trees. To create a space of these architectures, we only need three levels of motifs. The idea is to reduce the system size in half until one qubit remains while alternating between convolution and pooling operations. Given N qubits, a convolution stride s_c, pooling stride s_p and a pooling filter F^* that reduce system size in half, a reverse binary tree QCNN is generated in Algorithm 1.

Algorithm 1. QCNN, reverse binary tree architecture.

```
Input: $N, s_c, s_p, F^*$
Output: QCNN$\rightarrow M = (G_1, G_2, \dots, G_{|M|})$

    ▷ Primitives:
    $M_1^1 \leftarrow$ Qconv(stride $= s_c$)
    $M_2^1 \leftarrow$ Qpool(stride $= s_p$, filter $= F^*$)

    ▷ Motif: alternate convolution and pooling
    $M_1^2 \leftarrow M_1^1 + M_2^1$

    ▷ Motif: repeat until one qubit remain
    $M_1^3 \leftarrow$ Qfree$(N) + M_1^2 \times \log_2 N$

    $M \leftarrow$ assemble$(M_1^3)$
```

Algorithm 1 shows how to create instances of this architecture family. First, two primitives are created on the first level of the hierarchy, a convolution operation M^1_1 and a pooling operation M^1_2. They are then sequentially combined on level two as $M^2_1 = (M^1_1, M^1_2)$ to form a convolution-pooling unit. The third-level motif M^3_1 repeats this second-level motif M^2_1 until the system only contains one qubit. This is $\log_2(N)$ repetitions for N qubits because we chose F^* to reduce the system size in half during each pooling operation. The addition and multiplication symbols act as append and extend for tuples. For example, $M^l_1 + M^l_2 = (M^l_1, M^l_2)$ and $M^l_k \times 3 = (M^l_k, M^l_k, M^l_k)$ which allow for an intuitive way to build motifs. It is easy to expand the algorithm for more intricate architectures, for instance, by increasing the number of motifs per level and the number of levels. A valid level four motif for Algorithm 1 would be $M^4_1 = (M^3_1 + M^3_2) \times 3$, where $M^3_2 = \text{Qfree}(4) + M^2_2 + M^2_1$ and $M^2_2 = M^1_1 \times 2$ which is the reverse binary tree architecture M^3_1 then two convolutions and one convolution-pooling unit on four qubits, all repeated three times. Motifs can also be randomly selected on each level to generate architectures. The python package we provide acts as a tool to facilitate architecture generation this way.

In more detail, we now analyse the family of architectures generated by Algorithm 1. First, we consider the possible pooling filters F^* that reduce system size in half. It is equivalent to generating strings for the language $A = \{w | w$ has an equal number of 0s and 1s, $|w| = |Q^p_m|\}$. Let $N_{m-1} = |Q^x_{m-1}|$ indicate the number of available qubits for the filter F^*_m. Then based on the $\binom{4}{2} = 6$ possible equal binary strings of length four, we construct the following pooling filters:

$$F^{\text{right}}_m = \{0^{\frac{n}{2}}1^{\frac{n}{2}} | n = N_{m-1}\}, \tag{14}$$

$$F^{\text{left}}_m = \{1^{\frac{n}{2}}0^{\frac{n}{2}} | n = N_{m-1}\}, \tag{15}$$

$$F^{\text{odd}}_m = \{(01)^{\frac{n}{2}} | n = N_{m-1}\}, \tag{16}$$

$$F^{\text{even}}_m = \{(10)^{\frac{n}{2}} | n = N_{m-1}\}, \tag{17}$$

$$F^{\text{inside}}_m = \begin{cases} \{0^{\frac{n}{4}}1^{\frac{n}{2}}0^{\frac{n}{4}} | n = N_{m-1}\} & \text{if } N_{m-1} > 2, \\ \{01\} & \text{if } N_{m-1} = 2, \end{cases} \tag{18}$$

$$F^{\text{outside}}_m = \begin{cases} \{1^{\frac{n}{4}}0^{\frac{n}{2}}1^{\frac{n}{4}} | n = N_{m-1}\} & \text{if}|, N_{m-1} > 2, \\ \{10\} & \text{if } N_{m-1} = 2. \end{cases} \tag{19}$$

where the exponent $a^3 \equiv \{a\} \circ \{a\} \circ \{a\} = aaa$ refers to the regular operation concatenation: $A \circ B = \{xy | x \in A, y \in B\}$. The pooling filter F_{inside} yields 0110. Visually this pattern pools qubits from the inside (the middle of the circuit). See Fig. 8c. Figure 8a shows the repeated usage of F^{right} for pooling. This particular pattern is useful for data preprocessing techniques such as principal component analysis (PCA) since PCA introduces an order of importance to the features used in the model. Typically, the first principal component (which explains the most variance) is encoded on the first qubit, the second principal component on the second qubit and so on. Therefore, it makes sense to pool the last qubits and leave the first qubits in the model for as long as possible.

If $N = 8$, $s_c = 1$, $s_p = 0$ and $F^* = F^{\text{right}}$ then Algorithm 1 generates the circuit in Fig. 1d, Fig. 2, Fig. 6f, and Fig. 8a. Specifically, Fig. 8 shows how different values for s_c, s_pandF^* generate different instances of the family using Algorithm 1. The possible combinations of N, s_c, s_p, F^* represent the search space/family size. Since F^* reduces system size in half, it's required that the number of available qubits N is a power of two. Using integer strides causes the $|E^c_m| = |Q^c_m|$ constraint (see "Controlling primitives"), which enable translational invariance. The complexity of the model(in terms of the number of unitaries used) then scales linearly with the number of qubits N available. Specifically, N qubits result in $3N - 2$ number of unitaries. This is because of the geometric series: $N(\frac{1}{2^0} + \frac{1}{2^1} + \dots + \frac{1}{2^{\log_2 N - 1}}) + N(\frac{1}{2^1} + \frac{1}{2^2} + \dots + \frac{1}{2^{\log_2 N - 1}})$. Where the first sum is for convolution unitaries and the second for pooling.

Generalisation and search

The digraph formalism extends naturally to multi-qubit unitaries, enabling the representation of more intricate and larger-scale architectures. In general, a primitive with n-qubit unitaries is represented as a hypergraph $G = (Q, E)$, where the edges E consist of n-tuples. We introduce two additional hyperparameters, step and offset, which control the construction of E. For instance, Fig. 9 shows three primitives, each with 3-qubit unitaries. The first two have a stride of one, meaning that each 3-qubit unitary connects to its neighbours. In contrast, the last primitive has a stride of three, connecting every third qubit within the unitary. The offset parameter determines the starting point for counting; Fig. 9a begins with the first qubit, while Fig. 9b starts with the third. The step parameter controls the position of the next unitary; for example, Fig. 9a, b has a step of three, skipping two qubits before creating another edge starting on the third qubit. Consequently, the primitives with 2-qubit unitaries we've been considering thus far are all special cases with a step of one and an offset of zero. Another aspect to consider is the execution order of the unitaries, which by default is the sequence in which the edges were created for a primitive. Our package introduces an additional hyperparameter to control this order. For example to execute the third edge of Fig. 9a first followed by edge five, four, one and two, a value of

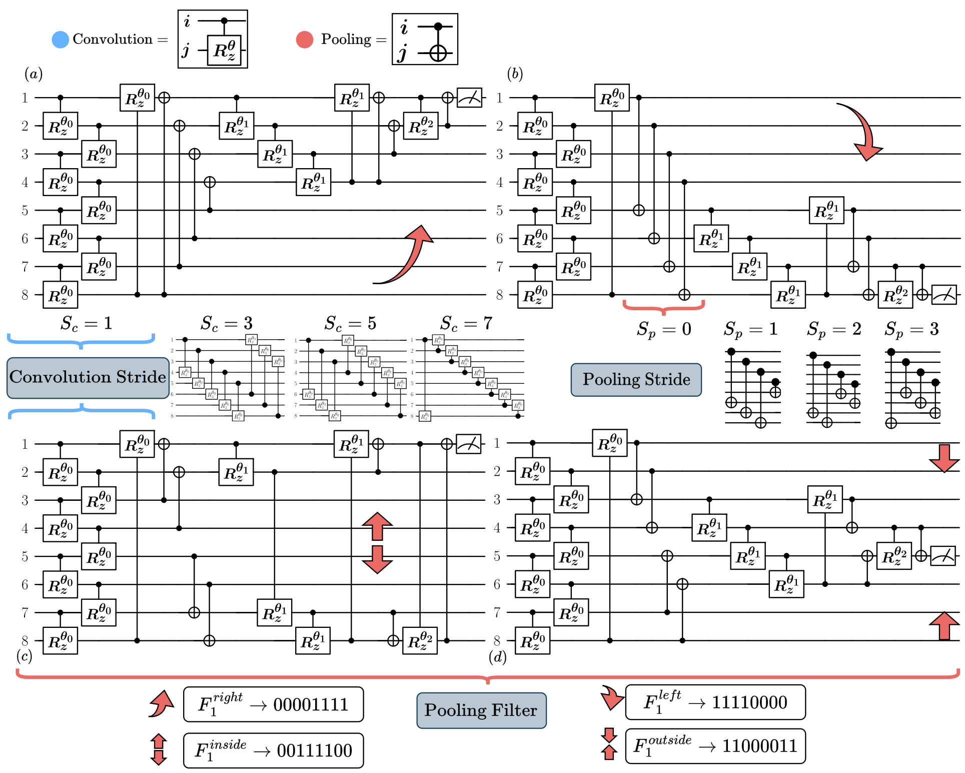


Fig. 8 Hyperparameters summary. An example of how the hyperparameters of the primitives effect the circuit architecture of the family generated by Algorithm 1. Three are shown, the convolution stride s_c, pooling stride s_p and pooling filter F^*. These are specified in "Controlling primitives". Controlled-R_z^θ gates are used for convolutions and CNOTs for pooling as an example. The convolution stride s_c determine how convolution unitaries are distributed across the circuit. Each convolution primitive typically consists of multiple unitaries and the QCNN requires them to be identical for weight sharing. The pooling stride s_p determine how pooling unitaries are distributed, for a given pooling primitive, a portion of available qubits gets pooled via controlled unitary operations and s_p dictates which controls match to which targets. The pooling filter F^* dictates which qubits to pool according to some recursive pattern/mask. For example, circuit d) always pools the outside qubits during pooling primitives, resulting in the middle qubit making it to the end of the circuit.

(3, 5, 4, 1, 2) can be passed to the edge order hyperparameter. Lastly, a boundary condition hyperparameter can also be specified, allowing for the definition of open or periodic boundaries for the qubits. This essentially determines whether edge creation is calculated in modulo with respect to the number of qubits or not, which in turn influences whether edge creation ceases when no further connections can be made based on the stride parameter.

Algorithm 2. Original QCNN from ref. [5].

```
Input: $n, N, F^*$
Output: QCNN→ $M = (G_1, G_2, \ldots, G_{|M|})$

▷ Primitives:
$M_1^1 \leftarrow \text{Qfree}(n+1) + \text{Qdense}()$
$M_2^1 \leftarrow \text{Qconv}(1, n, n-1, \text{mapping} = M_1^1)$
$M_3^1 \leftarrow \sum_{i=0}^{n-1} \text{Qconv}(1, n, i)$
$M_4^1 \leftarrow \text{Qpool}(1, n, 0, \text{filter} = F^*)$
$M_5^1 \leftarrow \text{Qconv}(1, n, 0)$

▷ Motif: Apply all primitives to N qubits
$M_1^2 \leftarrow \text{Qfree}(N) + \sum_{i=1}^{5} M_i^1$

▷ Motif: repeat d (depth) times
$M_1^3 \leftarrow M_1^2 * d$

$M \leftarrow \text{assemble}(M_1^3)$
```

The hyperparameters provided are sufficient to generate a diverse array of hierarchical architectures. For example, we demonstrate how to represent the original QCNN from ref. [5] within our framework in Algorithm 2. The arguments for each convolution and pooling primitive are stride, step, and offset. The Qdense primitive generates 2-qubit unitaries between all pairwise combinations of $n+1$ qubits. Subsequently, the second primitive M_2^1 takes M_1^1 as its mapping, which just means it treats M_1^1 as a single $n+1$-qubit unitary, and distributes it across the circuit with a stride of 1, step of n, and offset of $n-1$. This is followed by n convolutions of n-qubit unitaries, each having an offset incremented by one from the previous. For $n=3$ and $N=15$, the first and last convolution is illustrated in Fig. 9a, b. Next, a pooling layer with n-qubit unitaries is applied, measuring the outer $n-1$ qubits from each nth qubit, this corresponds to the filter $F^* = \{1^{\frac{n-1}{2}}01^{\frac{n-1}{2}}\}$. Finally, a convolution is performed on all remaining qubits. In practice, each of these primitives is given a mapping for their corresponding unitary. The mappings of the original QCNN are based on $2^v \times 2^v$ Gell–Mann matrices, where v indicates the number of qubits the unitary acts upon. For instance, the first unitary of the primitive M_2^1 operates on $v = n+1$ qubits, M_3^1 on $v = n$ qubits and pooling M_3^1 on n qubits where $v = n-1$ to leave a qubit for the control. For M_5^1, v equals the number of remaining qubits. It's easy to generate a family of architectures related to the original by providing the algorithm with different values of stride, step, offset, pooling filters, mappings and

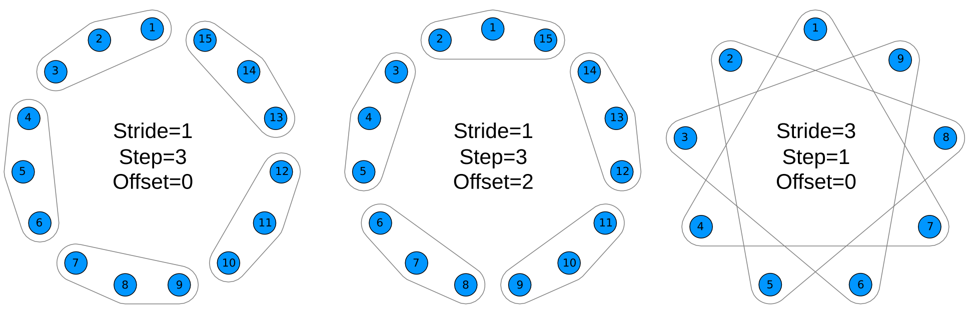
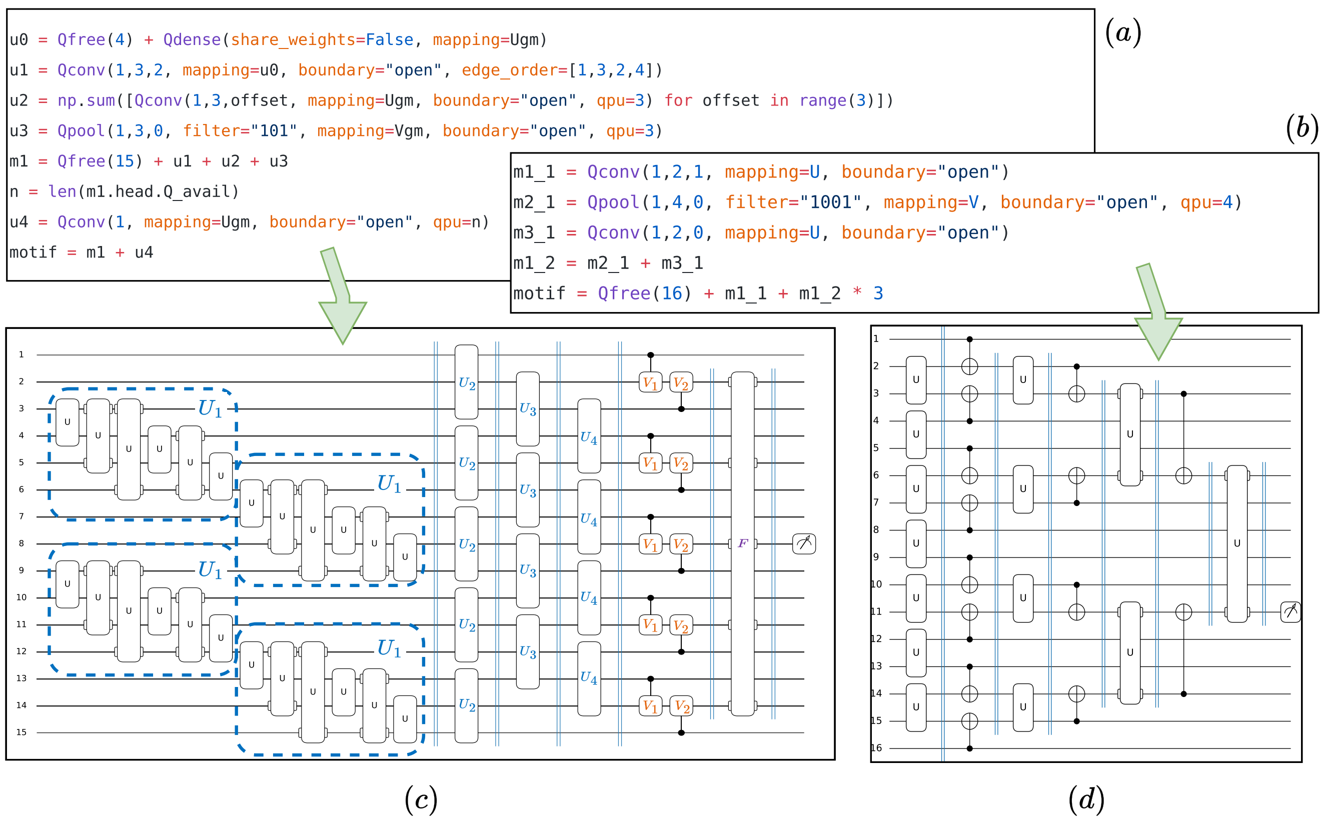


Fig. 9 Hypergraph example. Examples how 3-qubit unitaries are represented with the framework. For general *n*-qubit unitaries the graphs become hypergraphs with *n*-tuples as edges.



```
u0 = Qfree(4) + Qdense(share_weights=False, mapping=Ugm)
u1 = Qconv(1,3,2, mapping=u0, boundary="open", edge_order=[1,3,2,4])
u2 = np.sum([Qconv(1,3,offset, mapping=Ugm, boundary="open", qpu=3) for offset in range(3)])
u3 = Qpool(1,3,0, filter="101", mapping=Vgm, boundary="open", qpu=3)
m1 = Qfree(15) + u1 + u2 + u3
n = len(m1.head.Q_avail)
u4 = Qconv(1, mapping=Ugm, boundary="open", qpu=n)
motif = m1 + u4
```

```
m1_1 = Qconv(1,2,1, mapping=U, boundary="open")
m2_1 = Qpool(1,4,0, filter="1001", mapping=V, boundary="open", qpu=4)
m3_1 = Qconv(1,2,0, mapping=U, boundary="open")
m1_2 = m2_1 + m3_1
motif = Qfree(16) + m1_1 + m1_2 * 3
```

Fig. 10 Representation of architectures from literature. Example architectures from ref. [5] (**c**) and ref. [27] (**d**), generated using our Python package to demonstrate its expressibility, interpretability, and scalability. **a** The 15-qubit original QCNN is created with the first three parameters of each primitive being stride, step, and offset, respectively. The unitary U mappings employ generalised Gell–Mann matrices parametrised based on the number of qubits they act upon. Line 5, $\text{Qfree}(15) + m_1 + m_2 + m_3$, controls the system size; applying the same architecture to N qubits only requires changing it to Qfree(N). To introduce a depth d parameter to the circuit, the last line should be modified to $m_4 * d + m_5$. **b** A 16-qubit MERA circuit is generated. For an 8-qubit MERA circuit, the last line would be changed to $\text{Qfree}(8) + m_1^1 + m_2^1 * 2$, and in general, $\text{Qfree}(N) + m_1^1 + m_2^1(\log_2 N - 1)$ produces an N-qubit MERA circuit. These examples highlight the representation's strengths: the essence of an architecture is captured with a few lines of code in a modular and understandable manner, and scaling up to larger systems is accomplished with minimal adjustments.

relaxing the dependence on n based on how large we want the search space to be.

Next, we discuss the applicability of search algorithms with our representation. The framework's expressiveness is demonstrated in Fig. 2e, f, where only two lines of code are needed to specify a complete architecture, and in Fig. 10, which illustrates how to capture circuits from refs. [5,27]. This expressiveness allows search algorithms to explore an extensive range of architectures and numerous design choices. Moreover, the modularity of the framework enables search algorithms to identify robust building blocks to combine into motifs, serving as the foundation for architectural designs. This is especially advantageous in the context of genetic algorithms, as it facilitates the definition of crossover and mutation operations in various ways. For example, mutations can involve adjusting a single hyperparameter of a primitive or replacing an entire primitive within a motif. Crossovers may include combining motifs at the same or different levels or interweaving two motifs by alternating their final sequence of primitives. In the case of reinforcement learning, modularity allows an agent to make decisions at multiple levels of granularity, enabling it to explore and exploit different combinations of primitives and motifs. Hill climbing algorithms can also leverage this modularity in various ways. For instance, we can generate a random fixed high-level motif, such as a MERA circuit, and then iteratively optimise the hyperparameters of each primitive within the motif. In each step, we adjust a hyperparameter to neighbouring values, evaluate the resulting objective values, and select the best configuration. Once we have updated all the hyperparameters of all primitives, we obtain a final motif, which can be used as a starting point for the next iteration. This approach of adjusting individual hyperparameters within a multi-level motif allows for incremental changes to the architecture. Such fine-grained modifications can be beneficial in approaches like Bayesian optimisation, where smoothness in objective values is advantageous. In addition, the hierarchical nature of our representation allows it to capture scale-invariant circuit motifs, as seen in the MERA (Fig. 10) and TTN (Fig. 8) circuits. This inherent capability promotes scalability: it enables search algorithms to start with smaller subsystems, then scale up to larger ones using only the 'Qfree' primitive. As a result, there's no need to re-establish such motifs for larger systems, leading to reduced computational costs and broadened possibilities for architectural exploration (provided there are scale-invariant features to the problem at hand). Lastly, the intuitive nature of the representation facilitates interpretability, for instance, in one experiment, we observed a spike in performance for a convolution stride of five. Upon further investigation, we discovered a strong correlation between features one and six, which was previously unknown. This insight informed future experiments and design choices for the problem at hand. The specific experiment pertains to a pulsar classification task in radio astronomy, which is currently undergoing further experimentation.

Finally, we present the evolutionary algorithm used in our experiments, which is based on the approach described in ref. [26] and detailed in Algorithm 3. We refer to an architecture as a genotype, and its fitness is determined by the sample complexity for both inside and middle points, as well as the mean squared error (MSE) for outside points in the test set (see Fig. 5). Specifically, fitness $= c_1 \frac{M_{\text{in}}}{M_{\text{cap}}} + c_2 \frac{M_{\text{middle}}}{M_{\text{cap}}} + c_3 \text{MSE}_{\text{out}} + \lambda n_p$ where we cap M_{in} and M_{middle} by some large value M_{cap} and n_p is the number of parameters required for the architecture. The weights c_1, c_2, and c_3 sum to one, assigning importance to each term. Our experiments showed that setting $c_1 = 0.7$, $c_2 = 0.05$, and $c_3 = 0.25$ led to generally well-performing architectures, we also chose $M_{\text{cap}} = 500$ since fit genotypes exhibit sample complexity below 100. We initialise the population with a pool of 100 random primitives (Qconv, Qpool, Qdense), each having random hyperparameters. Upon initialisation, we perform mutation and crossover operations based on tournament selection with a 5% selection pressure. After the selection, we mutate the fittest genotype by choosing one of its primitives and replacing it with a randomly generated one. The crossover operator acts on the two fittest individuals, attempting to combine them tail-to-head. If this is not possible, they are interleaved up to the point where they can be combined. Just like the approach in[26], we do not remove any genotypes from the pool, leading to a more diverse population.

Algorithm 3. Evolutionary Search Algorithm

```
Input: Initial population $P$, memory table $M$ containing
fitness values
function CONTROLLER($P$, $M$)
    while True do
        $g_1, g_2 \leftarrow$ TOURNAMENT_SELECTION($M$)
        mutated $\leftarrow$ MUTATE($g_1$)
        combined $\leftarrow$ COMBINE($g_1, g_2$)
        Add mutated genotype to task queue
        Add combined genotype to task queue
        if idle worker available then
            Assign top task in task queue to the idle worker
        end if
    end while
end function
function WORKER(task, $M$)
    genotype $\leftarrow$ task.genotype
    fitness $\leftarrow$ EVALUATEFITNESS(genotype)
    Update memory table with the new fitness value
end function
```

DISCUSSION

This work has introduced a framework for representing hierarchical quantum circuits, enabling circuit generation and search space design. The framework is implemented as a readily available Python package that allows for the generation of complex quantum circuit architectures, which is particularly useful for Neural Architecture Search (NAS). Our numerical experiments illustrate the significance of alternating architectures for parameterised quantum circuits and present a pathway to enhance model performance without increasing its complexity.

Looking ahead, our primary goal is to investigate various search strategies leveraging this architectural representation, aiming to automate the discovery of quantum circuits tailored for a range of tasks. Although we've demonstrated the value of this representation for evolutionary algorithms, we're keen to explore alternative search algorithms such as reinforcement learning or Bayesian optimisation.

Another interesting consideration is the theoretical analysis of QCNN architectures that generalise well across multiple datasets. Recently, it has been shown how symmetry can be used to inform the inductive biases of a model[52,53], and we suspect that our numerical results stem from the search finding architectures that respect symmetries of the data. Symmetry is a natural starting point for creating primitives, the convolution primitive is already constrained by translational symmetry and additional primitives can be developed by considering other symmetries. This approach effectively narrows the search space, enabling a system to automatically discover general equivariant architectures that align well with the data. The framework also allows for the specification qubit orderings that correspond to physical hardware setups. Therefore, benchmarking the effect of noise on different architectures on NISQ devices would be a useful exploration.

METHODS

Overview

Figure 1 gives a broad view of the machine-learning pipeline we implement for the benchmarks. There are various factors influencing model performance during such a pipeline. Each step, from a raw audio signal to a classified musical genre, contains various possible configurations, the influence of which propagates throughout the pipeline. For this reason, it is difficult to isolate any configuration and evaluate its effect on the model. With our goal being to analyse QCNN architectures (Fig. 1d) on the audio data, we perform random search in the family created by Algorithm 1 with different choices of circuit ansatz and quantum data encoding. These are evaluated on two different datasets: Mel spectrogram data (Fig. 1b) and 2D statistical data (Fig. 1c), both being derived from the same audio signal (Fig. 1a). We preprocess the data based on requirements imposed by the model implementation before encoding it into a quantum state. These configurations are expanded on below:

Data

We aimed to use a practical and widely applicable dataset for the data component and chose the well-known[54] music genre dataset, GTZAN. It consists of 1000 audio tracks, each being a 30-s recording of some song. These recordings were obtained from radio, compact disks and compressed MP3 audio files[55]. Each is given a label of one of the following ten musical genres: blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, rock. Binary classification is used for the analysis of model performance across different architectures. Meaning there are $\binom{10}{2} = 45$ possible genre pairs to build models from. Each pair is equally balanced since there are 100 songs for each genre. The dataset enables the comparison of 45 models per configuration within the audio domain.

Model implementation

For all experiments, we evaluate instances of Algorithm 1 with $N = 8$ qubits, resulting in $3(8) - 2 = 22$ two-qubit unitaries. We test each model based on different combinations of model architecture, two-qubit unitary ansatz and quantum data encoding. The specific unitaries for U_m are chosen from a set of eight ansatzes that were used in ref. [29]. They are based on previous studies that explore the expressibility and entangling capability of parameterised circuits[56], hierarchical quantum classifiers[27] and extensions to the VQE[57]. These are shown in Supplementary Fig. 1, the ansatz for pooling also comes from ref. [29] and is shown in Supplementary Fig. 3. For quantum data encoding, we compare qubit encoding[36] with IQP encoding[58] on the tabular dataset. Amplitude encoding[59] is used for the image data.

Each model configuration considers all 45 genre pairs for classification, for example, rock vs reggae. Cross entropy is used as the cost function $C(y, \hat{y})$ during training, for rock vs reggae this would be:

$$C(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})). \tag{20}$$

where

$$y_i = \begin{cases} 1 & \text{if song } i \text{ is labelled rock,} \\ 0 & \text{if song } i \text{ labelled reggae.} \end{cases} \tag{21}$$

$\hat{y}_i$ is obtained from Eq. (6), i represents one observation, and both y, $\hat{y}$ are all the observations in vector form.

Data creation

We benchmark the model against two different forms of data, namely tabular and image. To construct the dataset in tabular form, we extract specific features from each audio signal using librosa[60] as shown in Fig. 1b. Each row represents a single audio track with its features as columns. The specific features extracted are those typically used by music information retrieval systems, namely: chroma frequencies, harmonic and percussive elements, Mel-frequency cepstral coefficients, root-mean-square, spectral bandwidth, spectral centroid, spectral roll-off, tempo and the zero crossing rate. See Supplementary Table 1 for a short description of these features. To construct the dataset in image form, we extract a Mel-frequency spectrogram (Fig. 1c) from each audio signal. The Mel scale is a non-linear transformation based on how humans perceive sound and is frequently used in speech recognition applications[61]. The spectrogram size depends on the number of qubits available for the QCNN. We can encode 2^N values with amplitude encoding into a quantum state, where N is the number of available qubits. Using $N = 8$ qubits, we scale the image to $8 \times 32 = 256 = 2^8$ pixels, normalising each pixel between 0 and 1. The downscaling is done by binning the Mel frequencies into eight groups and taking the first three seconds of each audio signal.

Data preprocessing

Two primary forms of preprocessing are applied to the data before it is sent to the model: feature scaling and feature selection. The features are scaled using min-max scaling, where the range is based on the type of quantum data encoding used. For amplitude encoding, the data is scaled to the range $[0, 1]$, qubit encoding to $[0, \pi/2]$ and IQP encoding to $[0, \pi]$. Feature selection is only applied to the tabular data. Using qubit encoding with $N = 8$ qubits results in selecting eight features. Principal Component Analysis (PCA) and decision trees are used to perform the selection. The tree-based selection is used to compare against PCA to verify whether PCA does not heavily bias the model's results.

Model evaluation

The model is trained with 70% of the data while 30% is held out as a test set to evaluate performance. During training, fivefold cross-validation is used on each model. The average classification accuracy and standard deviation of 30 separate trained instances are calculated on the test set as performance metrics.

DATA AVAILABILITY

The music genre classification dataset analysed during the current study is available on TensorFlow Datasets, https://www.tensorflow.org/datasets/catalog/gtzan.

CODE AVAILABILITY

The theoretical framework discussed in this paper has been implemented as an open-source Python package, which is available on GitHub at https://github.com/matt-lourens/hierarqcal. This package was used to generate all the circuits in the paper.

Received: 13 December 2022; Accepted: 20 July 2023;
Published online: 05 August 2023

REFERENCES

1. Benedetti, M., Lloyd, E., Sack, S. & Fiorentini, M. Parameterized quantum circuits as machine learning models. *Quantum Sci. Technol.* **4**, 043001 (2019).
2. Cerezo, M. et al. Variational quantum algorithms. *Nat. Rev. Phys.* **3**, 625–644 (2021).
3. Mangini, S., Tacchino, F., Gerace, D., Bajoni, D. & Macchiavello, C. Quantum computing models for artificial neural networks. *Europhys. Lett.* **134**, 10002 (2021).
4. Bharti, K. et al. Noisy intermediate-scale quantum algorithms. *Rev. Mod. Phys.* **94**, 015004 (2022).
5. Cong, I., Choi, S. & Lukin, M. D. Quantum convolutional neural networks. *Nat. Phys.* **15**, 1273–1278 (2019).
6. Pesah, A. et al. Absence of barren plateaus in quantum convolutional neural networks. *Phys. Rev. X* **11**, 041011 (2021).
7. Banchi, L., Pereira, J. & Pirandola, S. Generalization in quantum machine learning: a quantum information standpoint. *PRX Quantum* **2**, 040321 (2021).
8. Herrmann, J. et al. Realizing quantum convolutional neural networks on a superconducting quantum processor to recognize quantum phases. *Nat. Commun.* **13**, 4144 (2022).
9. Carleo, G. & Troyer, M. Solving the quantum many-body problem with artificial neural networks. *Science* **355**, 602–606 (2017).

10. Carrasquilla, J. & Melko, R. G. Machine learning phases of matter. *Nat. Phys.* **13**, 431–434 (2017).
11. van Nieuwenburg, E. P. L., Liu, Y.-H. & Huber, S. D. Learning phase transitions by confusion. *Nat. Phys.* **13**, 435–439 (2017).
12. Deng, D.-L., Li, X. & Sarma, S. D. Machine learning topological states. *Phys. Rev. B* **96**, 195145 (2017).
13. Levine, Y., Sharir, O., Cohen, N. & Shashua, A. Quantum entanglement in deep learning architectures. *Phys. Rev. Lett.* **122**, 065301 (2019).
14. Stoudenmire, E. & Schwab, D. J. Supervised learning with tensor networks. *Adv. Neural Inf. Process. Syst.* **29** https://papers.nips.cc/paper/2016/hash/5314b9674c86e3f9d1ba25ef9bb32895-Abstract.html (2016).
15. Deng, D.-L., Li, X. & Sarma, S. D. Quantum entanglement in neural network states. *Phys. Rev. X* **7**, 021021 (2017).
16. Lin, H. W., Tegmark, M. & Rolnick, D. Why does deep and cheap learning work so well? *J. Stat. Phys.* **168**, 1223–1247 (2017).
17. Mehta, P. & Schwab, D. J. An exact mapping between the variational renormalization group and deep learning. Preprint at http://arxiv.org/abs/1410.3831 (2014).
18. Levine, Y., Yakira, D., Cohen, N. & Shashua, A. Deep learning and quantum entanglement: Fundamental connections with implications to network design. *Int. Conf. Learn. Represent*. https://openreview.net/forum?id=SywXXwJAb (2018).
19. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521**, 436–444 (2015).
20. Krizhevsky, A., Sutskever, I. & Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst* **25** https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (2012).
21. Zoph, B. & Le, Q. V. Neural architecture search with reinforcement learning. *Int. Conf. Learn. Represent*. https://openreview.net/forum?id=r1Ue8Hcxg (2017).
22. Elsken, T., Metzen, J. H. & Hutter, F. Neural architecture search: a survey. *J. Mach. Learn. Res.* **20**, 1–21 (2019).
23. Real, E., Aggarwal, A., Huang, Y. & Le, Q. V. Regularized evolution for image classifier architecture search. *Proc. AAAI Conf. Artif. Intell.* **33**, 4780–4789 (2019).
24. Zoph, B., Vasudevan, V., Shlens, J. & Le, Q. V. Learning transferable architectures for scalable image recognition. in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* 8697–8710 (IEEE, 2018).
25. Chen, L.-C. et al. Searching for efficient multi-scale architectures for dense image prediction. *Adv. Neural Inf. Process. Syst.* **31** https://papers.nips.cc/paper_files/paper/2018/hash/c90070e1f03e982448983975a0f52d57-Abstract.html (2018).
26. Liu, H., Simonyan, K., Vinyals, O., Fernando, C. & Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *Int. Conf. Learn. Represent*. https://openreview.net/forum?id=BJQRKzbA- (2018).
27. Grant, E. et al. Hierarchical quantum classifiers. *NPJ Quantum Inf.* **4**, 65 (2018).
28. Haug, T., Bharti, K. & Kim, M. Capacity and quantum geometry of parametrized quantum circuits. *PRX Quantum* **2**, 040309 (2022).
29. Hur, T., Kim, L. & Park, D. K. Quantum convolutional neural network for classical data classification. *Quantum Mach. Intell.* **4**, 3 (2022).
30. Oh, S., Choi, J. & Kim, J. A tutorial on quantum convolutional neural networks (QCNN). in *International Conference on Information and Communication Technology Convergence* 236–239 (IEEE, 2020).
31. Franken, L. & Georgiev, B. Explorations in quantum neural networks with intermediate measurements. *In Proc. ESSAN*, Vol. ES2020, 297–302 (2020).
32. McClean, J. R., Boixo, S., Smelyanskiy, V. N., Babbush, R. & Neven, H. Barren plateaus in quantum neural network training landscapes. *Nat. Commun.* **9**, 4812 (2018).
33. Holmes, Z., Sharma, K., Cerezo, M. & Coles, P. J. Connecting ansatz expressibility to gradient magnitudes and barren plateaus. *PRX Quantum* **3**, 010313 (2022).
34. Schuld, M., Sweke, R. & Meyer, J. J. Effect of data encoding on the expressive power of variational quantum-machine-learning models. *Phys. Rev. A* **103**, 032430 (2021).
35. Abbas, A. et al. The power of quantum neural networks. *Nat. Comput. Sci.* **1**, 403–409 (2021).
36. Schuld, M. Supervised quantum machine learning models are kernel methods. Preprint at https://arxiv.org/abs/2101.11020v2 (2021).
37. Zhang, S.-X., Hsieh, C.-Y., Zhang, S. & Yao, H. Differentiable quantum architecture search. *Quantum Sci. Technol.* **7**, 045023 (2022).
38. Zhang, S.-X., Hsieh, C.-Y., Zhang, S. & Yao, H. Neural predictor based quantum architecture search. *Mach. Learn. Sci. Technol.* **2**, 045027 (2021).
39. Grimsley, H. R., Economou, S. E., Barnes, E. & Mayhall, N. J. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nat. Commun.* **10**, 3007 (2019).
40. Tang, H. L. et al. Qubit-ADAPT-VQE: an adaptive algorithm for constructing hardware-efficient ansatze on a quantum processor. *PRX Quantum* **2**, 020310 (2021).
41. Yordanov, Y. S., Armaos, V., Barnes, C. H. W. & Arvidsson-Shukur, D. R. M. Qubit-excitation-based adaptive variational quantum eigensolver. *Commun. Phys.* **4**, 228 (2021).
42. Rattew, A. G., Hu, S., Pistoia, M., Chen, R. & Wood, S. A domain-agnostic, noise-resistant, hardware-efficient evolutionary variational quantum eigensolver. Preprint at https://arxiv.org/abs/1910.09694 (2020).
43. Zhu, L. et al. Adaptive quantum approximate optimization algorithm for solving combinatorial problems on a quantum computer. *Phys. Rev. Res.* **4**, 033029 (2022).
44. Li, L., Fan, M., Coram, M., Riley, P. & Leichenauer, S. Quantum optimization with a novel Gibbs objective function and ansatz architecture search. *Phys. Rev. Res.* **2**, 023074 (2020).
45. Ostaszewski, M., Grant, E. & Benedetti, M. Structure optimization for parameterized quantum circuits. *Quantum* **5**, 391 (2021).
46. Du, Y., Huang, T., You, S., Hsieh, M.-H. & Tao, D. Quantum circuit architecture search for variational quantum algorithms. *NPJ Quantum Inf.* **8**, 1–8 (2022).
47. Lu, Z., Shen, P.-X. & Deng, D.-L. Markovian quantum neuroevolution for machine learning. *Phys. Rev. Appl.* **16**, 044039 (2021).
48. Preskill, J. Quantum computing in the NISQ era and beyond. *Quantum* **2**, 79 (2018).
49. Smacchia, P. et al. Statistical mechanics of the cluster Ising model. *Phys. Rev. A* **84**, 022304 (2011).
50. Haldane, F. D. M. Nonlinear field theory of large-spin Heisenberg antiferromagnets: Semiclassically quantized solitons of the one-dimensional easy-axis néel state. *Phys. Rev. Lett.* **50**, 1153–1156 (1983).
51. Nielsen, M. A. & Chuang, I. L.*Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, 2011).
52. Larocca, M. et al. Group-invariant quantum machine learning. *PRX Quantum* **3**, 030341 (2022).
53. Meyer, J. J. et al. Exploiting symmetry in variational quantum machine learning. *PRX Quantum* **4**, 010328 (2023).
54. Sturm, B. L. A survey of evaluation in music genre recognition. in *Adaptive Multimedia Retrieval: Semantics, Context, and Adaptation*, vol. 8382 of *Lecture Notes in Computer Science* (eds Nürnberger, A. et al.) 29–66 (Springer International Publishing, 2014).
55. George, T., Georg, E. & Perry, C. Automatic musical genre classification of audio signals. *In Proc. ISMIR, Indiana*, Vol. 144. https://ismir2001.ismir.net/papers.html (2001).
56. Sim, S., Johnson, P. D. & Aspuru-Guzik, A. Expressibility and entangling capability of parameterized quantum circuits for hybrid quantum-classical algorithms. *Adv. Quantum Technol.* **2**, 1900070 (2019).
57. Parrish, R. M., Hohenstein, E. G., McMahon, P. L. & Martínez, T. J. Quantum computation of electronic transitions using a variational quantum eigensolver. *Phys. Rev. Lett.* **122**, 230401 (2019).
58. Havlícek, V. et al. Supervised learning with quantum-enhanced feature spaces. *Nature* **567**, 209–212 (2019).
59. Schuld, M. & Petruccione, F. *Machine Learning with Quantum Computers* (Springer International Publishing, 2021).
60. McFee, B. et al. librosa/librosa: 0.8.1rc2. https://doi.org/10.5281/zenodo.4792298 (2021).
61. Davis, S. & Mermelstein, P. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. Acoust. Speech Signal Process.* **28**, 357–366 (1980).

ACKNOWLEDGEMENTS
The development of the Python package was funded by Unitary Fund (https://unitary.fund/). Support from the Oppenheimer Memorial Trust, NICIS (National Integrated Cyber Infrastructure System) e-research grant QICSI7 and the South African Research Chair Initiative, Grant No. 64812 of the Department of Science and Innovation and the National Research Foundation of the Republic of South Africa is kindly acknowledged. This research was also supported by the Yonsei University Research Fund of 2022 (2022-22-0124), by the National Research Foundation of Korea (Grant No. 2022M3E4A1074591), and by the KIST Institutional Program (2E32241-23-010).

AUTHOR CONTRIBUTIONS
M.L. designed the theoretical framework, performed the numerical experiments, and implemented the software. I.S. and F.P. supervised the research. D.P. and C.B. conceived the experiment and provided insights on the reference architecture. All authors reviewed and discussed the analyses and results and contributed towards writing the manuscript.

COMPETING INTERESTS

C.B. is the Co-Founder of Data cybernetics, and Francesco Petruccione the Chair of Scientific Board and Co-Founder of QUNOVA computing. The remaining authors declare no competing interests.

ADDITIONAL INFORMATION

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41534-023-00747-z.

Correspondence and requests for materials should be addressed to Matt Lourens.