UNIVERSITÀ DEGLI STUDI FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali

Laurea Magistrale in Data Science, Calcolo Scientifico & Intelligenza Artificiale

Quantum Machine Learning project

HIERARCHICAL QUANTUM CIRCUIT REPRESENTATION FOR MUSIC GENRE CLASSIFICATION

LUDOVICO VENTURI

Supervisor: *prof. Filippo Caruso*

Academic Year 2023-2024

Table of contents

Abstract

In this project work I tried to replicate some of the results of the paper "Hierarchical quantum circuit representations for neural architecture search" [1].

In this paper the authors propose a framework named HierarQcal for representing Quantum Convolutional Neural Networks (QCNN) architectures using techniques from Neural Architectural Search (NAS). This framework enables search space design and architecture search, the former being the most challenging point in applying NAS to QCNN.

In this work I generate the family of QCNN architectures resembling reverse binary trees and I evaluate this family on the GTZAN music genre classification dataset showing that it is possible to improve model performance without increasing complexity.

1 Background

1.1 Supervised Learning for classification

The goal of classification is to use some data X alongside a function f_m (model) to accurately represent a discrete categorization y:

$$f_m(X, \theta) = \hat{y} \approx y$$

The data is used by iteratively changing the model parameters θ based on the disparity between the current representation $\hat{y}$ and the actual categorization y, measured with a cost function $C(y, \hat{y})$. It's a supervised problem because the label of each input is known from the start.

The cost function I use in this project is Binary Cross Entropy (BCE) from `torch.nn.BCELoss`. Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. In other words, if we consider a target probability distribution P and an approximation of the target distribution Q, then the cross-entropy of Q from P is the number of additional bits to represent an event using Q instead of P:

$$H(P, Q) = -\sum_{x \in X} P(x) \cdot ln(Q(x))$$

1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are successful because they don't need manual feature design and can learn high-level features from raw data. With CNN there is a focus shift: from feature design to architecture design.

....

1.3 Quantum Machine Learning

Using quantum computers it is possible to write hybrid quantum-classical algorithms already usable in the NISQ era, where the optimization of parameters is done classically and the function f_m is built as a Variational Quantum Circuit (VQC) that acts on a quantum state $|\psi\rangle$.

A VQC is a quantum circuit with trainable parameters in some gates, as a rotation angle $\theta, e.g. RY(\theta)$.

The point in using a VQC is that the state can move along all Hilbert space at every change of the parameters, so it is possible to sample from a classically intractable probability density function (pdf).

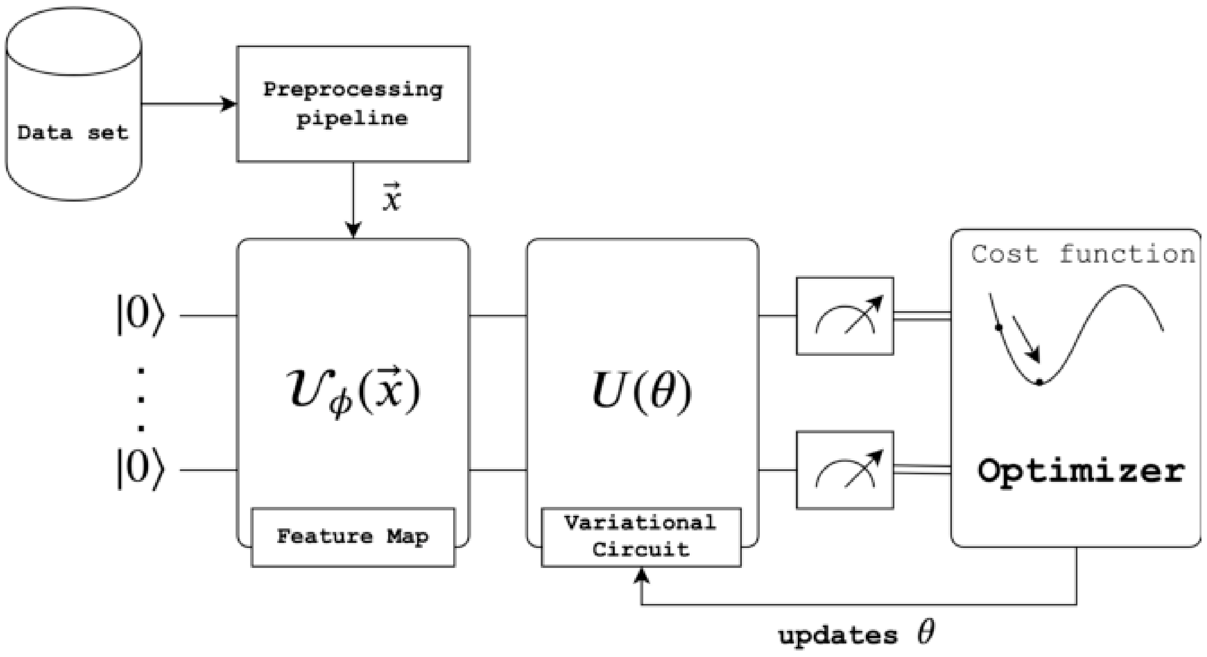


Figure 1.1: Variational Quantum Circuit [2]

1.3.1 Data encoding

The state $|\psi\rangle$ must be obtained through an embedding since we're in a Classical-Quantum (CQ) setting and the data we use for training is classic. This is done with a *feature map*, as can be seen in Figure 1.1.

A quantum embedding represents classical data as quantum states in a Hilbert space via a quantum feature map. It takes a classical data x and translates it into a set of gate parameters in a quantum circuit, creating a quantum state $|\psi_x\rangle$.

In this project I use *angle embedding* to encode classical data (audio statistics) to the circuit (see line 10 in Figure 2.6). With angle embedding, single-qubit rotation gates encode a classical $x_i \in \mathcal{R}$.

Each element of the input determines the angle of the rotation gate (e.g. an RY rotation gate). This approach requires n qubits to encode n input variables and can be defined as:

$$|\psi_x\rangle = \bigotimes_{i=1}^{n} cos(x_i)\,|0\rangle + sin(x_i)\,|1\rangle = \bigotimes_{i=1}^{n} R(x_i)\,|\psi_0\rangle$$

1.3.2 QCNNs

QCNN stands out among other parametrized quantum circuits (PQC) models for its shallow circuit depth, good generalization capabilities and absence of *barren plateaus*.

A *barren plateau* happens when the gradient of a cost function vanishes exponentially with system size, rendering the architecture untrainable for large problem sizes. For PQC, random circuits are often proposed as initial guesses for exploring the space of quantum states, due to exponential dimension of Hilbert space and the gradient estimation complexity on more than a few qubits.

It is important to note that for a wide class of PQC the probability that the gradient along any reasonable direction is non-zero to some fixed precision is exponentially small as a function of the number of qubits [3]. For QCNN in particular it is guaranteed that randomly initialized QCNN are trainable unlike many other PQC architectures, since the variance of the gradient vanishes no faster than polynomially [4] so QCNNs do not exhibit *barren plateaus*.

The next step is learning network architecture, which NAS aims to achieve [5]. NAS consists of 3 main components:

- search space
- search strategy
- performance estimation strategy

The *search space* defines the set of possible architectures that a search algorithm can consider, and a carefully designed search space is important for search efficiency. The main contribution of [1] is a framework that enables the dynamic generation of QCNN and the creation of QCNN search spaces: HierarQcal.

1.4 HierarQcal

HierarQcal is an open-source python package [6] that simplifies the process of creating general QCNN by enabling an hierarchical design process. It makes automatic generation of QCNN circuits easy and it facilitates QCNN search space design for NAS.

The package includes primitives such as *convolutions, pooling* and *dense layers* that can be stacked together hierarchically to form complex QCNN circuit architectures.

....

2 Methods

2.1 Dataset

The GTZAN dataset [7] is the most-used public dataset for evaluation in machine listening research for music genre recognition (MGR). The files were collected in 2000-2001 from a variety of sources including personal CDs, radio, microphone recordings, in order to represent a variety of recording conditions.

The dataset consists of 1000 audio tracks each 30 seconds long. It contains 10 genres, each represented by 100 tracks. The tracks are all 22050Hz Mono 16-bit audio files in `.wav` format. The genres are: blues,classical, country, disco, hiphop, jazz, metal, pop, reggae, rock.

I used the GTZAN dataset from Kaggle [8], which already include statistics extracted from the audio sources. The information gathered from audio signals to produce the tabular dataset can be easily extracted with `librosa` [1] and contains: Chroma frequencies, Harmonic and percussive elements, Mel-frequency cepstral coefficients, Spectral bandwidth and others. For a description of these and other features see Appendix D of [1].

I did binary classification in this analysis, in particular I focused only on *rock vs country* classification, the most difficult task between all the $\binom{10}{2} = 45$ possible genre pairs.

2.2 Model implementation

2.2.1 Main workflow

Data is first preprocessed in two ways:

- feature scaling
- feature reduction

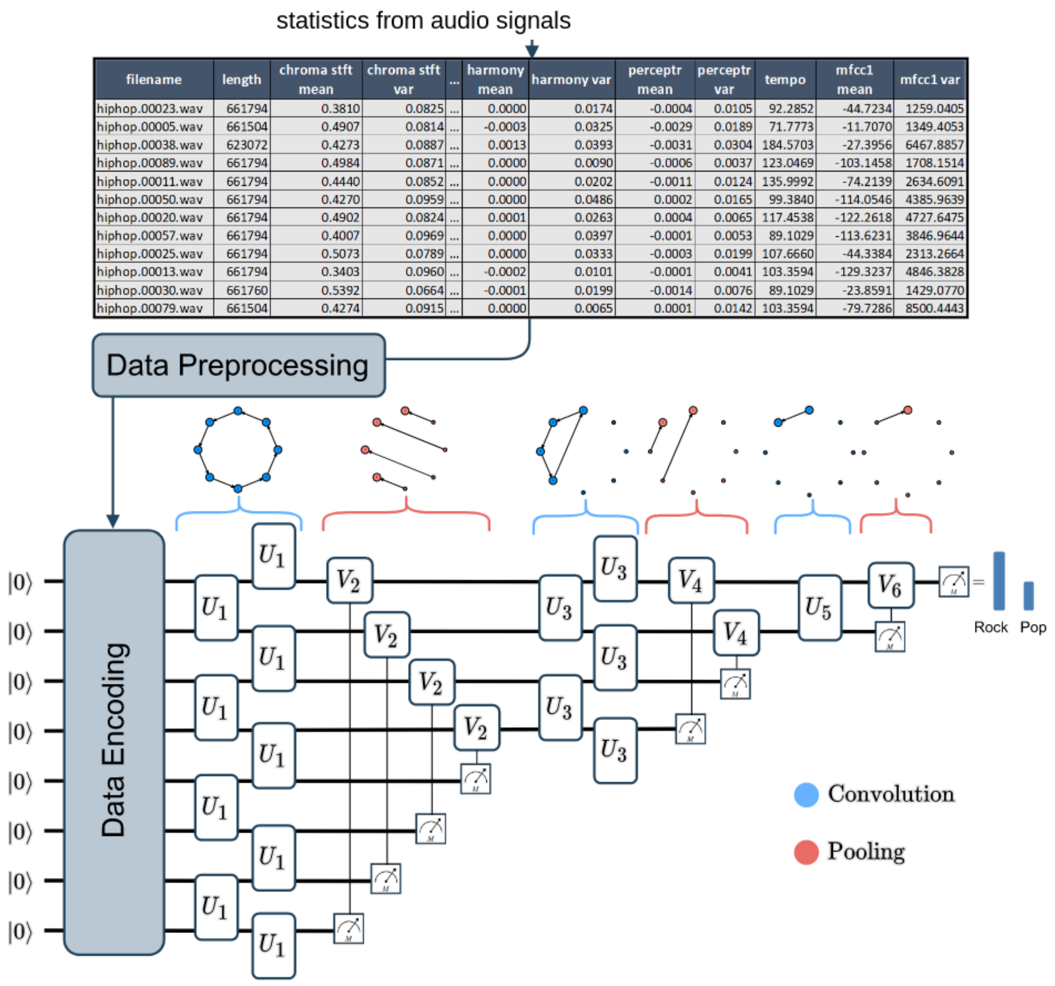
[1]a Python package for audio and music signal processing [9]

Features are scaled using `min-max` scaling in a range chosen for angle embedding: $[0, \pi/2]$. Then Principal Component Anaylisis with 8 components is used to perform the reduction.

I used `sklearn` for both operations. I also used `Pipeline` from `sklearn` to create a preprocessing pipeline that will be used for the search of hyperparameters (that will be discussed in Section 2.2.5).

```python
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline

pipeline = Pipeline(
    [
        ("scaler", MinMaxScaler((0, np.pi / 2))),
        ("pca", PCA(8)),
    ]
)
```

Each row of the data resulting from the preprocessing has 8 features and so it can be encoded through an angle embedding into a quantum circuit with 8 qubit. The quantum circuit used here is a QCNN with a hierarchical design: the idea is to reduce the system size in half until one qubit remains while alternating between convolution and pooling operations. Grant et al. [10] exhibited the success of hierarchical designs that resemble reverse binary trees. To create a space of these architectures only three levels of motifs are needed.



statistics from audio signals

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

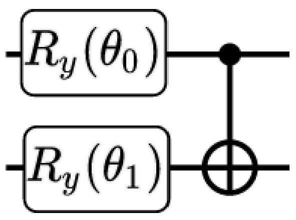
Figure 2.1: Main workflow of model implementation. Us are convolutional unitaries and Vs a re pooling unitaries. From [1]

I used $N = 8$ qubits with pennylane `default.qubit.torch` as simulation device (see line 2 in Figure 2.6). I tested each model based on different combinations of model architecture (that will be discussed in more detail later) and two-qubit unitary ansatzes.

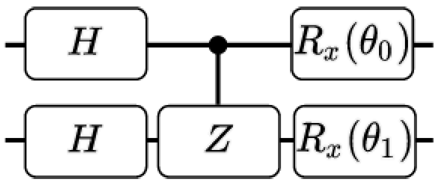
2.2.2 Ansatzes

I used 3 out of the 8 proposed ansatzes in Lourens et al. [1], in particular:

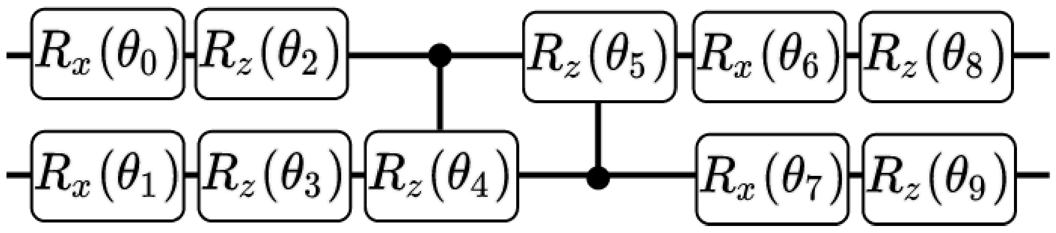
Below there is the correspondent code for each ansatz:



(a) Two parameter ansatz.



(b) Two parameter ansatz.



(g) Ten parameter ansatz.

Figure 2.2Convolution ansatzes used in this project, from [1]

Figure 2.3

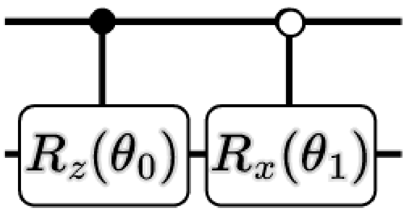
```
# Convolution ansatz (a)
def ansatz_conv_a(bits, symbols=None):
  qml.RY(symbols[0], wires=[bits[0]])
  qml.RY(symbols[1], wires=[bits[1]])
  qml.CNOT(wires=[bits[0], bits[1]])
U_ansatz_conv_a = Qunitary(ansatz_conv_a, n_symbols=2, arity=2)

# Convolution ansatz (b)
def ansatz_conv_b(bits, symbols=None):
  qml.Hadamard(wires=[bits[0]])
  qml.Hadamard(wires=[bits[1]])
  qml.CZ(wires=[bits[0], bits[1]])
  qml.RX(symbols[0], wires=[bits[0]])
  qml.RX(symbols[1], wires=[bits[1]])
U_ansatz_conv_b = Qunitary(ansatz_conv_b, n_symbols=2, arity=2)

# Convolution ansatz (g)
def ansatz_conv_g(bits, symbols):
    qml.RX(symbols[0], wires=bits[0])
    qml.RX(symbols[1], wires=bits[1])
    qml.RZ(symbols[2], wires=bits[0])
    qml.RZ(symbols[3], wires=bits[1])
    qml.CRZ(symbols[4], wires=[bits[1], bits[0]])
    qml.CRZ(symbols[5], wires=[bits[0], bits[1]])
    qml.RX(symbols[6], wires=bits[0])
    qml.RX(symbols[7], wires=bits[1])
    qml.RZ(symbols[8], wires=bits[0])
    qml.RZ(symbols[9], wires=bits[1])
U_ansatz_conv_g = Qunitary(ansatz_conv_g, n_symbols=10, arity=2)
```

For pooling ansatz I used:

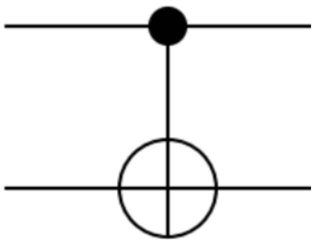
2.2.3 Circuit

The code below illustrates how a general motif that resembles reverse binary trees is built using HierarQcal. This motif is the base for every model I used in this project.

```
def qcnn_motif(ansatz_c, conv_stride, conv_step, conv_offset,
 ↪  share_weights, ansatz_p, pool_filter, pool_stride):
    qcnn = (
```



(a) (1) Pooling ansatz from [11]



(a) (2) Simpler pooling ansatz

Figure 2.5: Pooling ansatzes

```
        Qinit(8)
        + (
            Qcycle(
                stride=conv_stride,
                step=conv_step,
                offset=conv_offset,
                mapping=ansatz_c,
                share_weights=share_weights,
            )
            + Qmask(pool_filter, mapping=ansatz_p,
             ↪  strides=pool_stride)
        )
        * 3
    )

    return qcnn
```

The motif obtained from `qcnn_motif` is not ready to be executed. It needs to be translated to a pennylane circuit and also the input $x \in \mathcal{R}^8$ must be embedded into this circuit.

The real execution with input `x` and circuit parameters `symbols` is done with:

```
def net(motif, symbols, x):
    motif.set_symbols(symbols)
    circuit = get_circuit(motif, x)
    y_hat = circuit()

    return y_hat
```

```
def get_circuit(hierq, x=None):
    dev = qml.device("default.qubit.torch", wires=hierq.tail.Q,
    ↪ shots=None)

    @qml.qnode(dev, interface="torch", diff_method="backprop")
    def circuit():
        if isinstance(next(hierq.get_symbols(), False), sp.Symbol):
            # Pennylane doesn't support symbolic parameters, so if
            ↪ no symbols were set (i.e. they are still symbolic),
            ↪ we initialize them randomly
            hierq.set_symbols(np.random.uniform(0, 2 * np.pi,
    ↪ hierq.n_symbols))
        if x is not None:
            AngleEmbedding(x, wires=hierq.tail.Q, rotation="Y")
        hierq(backend="pennylane")  # This executes the compute
    ↪ graph in order
        return qml.probs(wires=hierq.head.Q[0])

    return circuit
```

Figure 2.6

2.2.4 Training

The dataset has been split in 70% training set and 30% test set. During training, 3-fold cross validation is used for each model. This step is automated during the search for hyperparameters.

Each model was trained without batch for 100 epochs employing the Adam optimizer with a learning rate of $1 \times 10 - 1$ that minimizes the Cross-Entropy Loss (see training code in Figure 2.7). All experiments were performed using Pytorch and the PennyLane Quantum library [12] on an AMD Ryzen 7 PRO 5850U with 16GB of RAM.

2.2.5 Hyperparameters search

To search in the architectures space I fixed the ansatzes (both for convolution and pooling) then I used `GrisSearchCV` from `sklearn` to perform an exhaustive search over the grid of hyperparameters.

In this way the preprocessing for each fold is managed internally and it is less error-prone. I needed to implement a subclass of `sklearn.base.BaseEstimator` and instantiate it as a last step of my pipeline in order to use `GridSearchCV`.

```python
def train(x, y, motif, N=100, lr=0.1, verbose=True):
    n_symbols = motif.n_symbols
    if n_symbols > 0:
        symbols = torch.rand(n_symbols, requires_grad=True)
        opt = torch.optim.Adam([symbols], lr=lr)
        for it in range(N):
            opt.zero_grad() # reset gradients
            y_hat = net(motif, symbols, x)
            loss = objective_function(y_hat, y)
            loss.backward()
            opt.step()

            if verbose:
                if it % 25 == 0:
                    print(f"Loss at step {it}: {loss}")
    else:
        symbols = None
        loss = objective_function(motif, [], x, y)
    return symbols, loss

def objective_function(y_hat, y):
    loss = nn.BCELoss()
    assert(len(y_hat) == len(y))
    # index 1 corresponds to predictions for being in class 1
    loss = loss(y_hat[:, 1], torch.tensor(y, dtype=torch.double))
    return loss
```

Figure 2.7

```python
from sklearn.model_selection import GridSearchCV

grid_params = {
                'model__stride_c':list(range(1,8)),
                'model__step_c':[1,2],
                'model__share_weights':[True, False],
                'model__filter_p':["!*","*!", "!*!", "*!*", "01",
                 ↪  "10"], #left, right, outside, inside, 01, 10#
                'model__stride_p':list(range(0,4)),
              }

grid = GridSearchCV(pipeline, grid_params, cv=3, n_jobs=8,
 ↪  verbose=True, refit=True)
```

Figure 2.8

The hyperparameters I decided to look for were:

- `share_weights`: True or False
- `convolution_stride`: between [1,7]
- `convolution_step`: between [1,2]
- `pooling filter`: between { left, right, outside, inside, odd, even }
- `pooling stride`: between [0,3]

These are not the total parameters possibile (for example it is also possible to set an `offset` for convolution). I decided to put as maximum limit of training runs ≈ 1000 (considering 3 runs for each configuration given the cross validation) which in practice was about 30 to 45 minutes on the test machine, leaving all the parameters searched also in [1].

`share_weights` is an interesting parameter because by disabling it is possible to gain some accuracy at the cost of adding a large number of parameters. For example for ansatz (a) we pass from 6 to 26 parameters to optimize.‘

3 Results

I ran multiple test with various ansatzes, keep in mind Figure 2.3 and Figure 2.5 for ansatzes naming.

3.1 Convolution ansatz (a)

For convolution ansatz (a) and pooling ansatz (1) I obtained the following results:

Table 3.1: a

	Share weights	Conv. stride	Conv. step	Pool. filter	Pool. stride	Mean validation scor
167	False	7	1	!*!	3	0.714770
217	False	6	1	*!*	1	0.707678
58	True	1	1	*!	2	0.707678
233	True	3	1	01	1	0.693494
123	True	3	1	!*!	3	0.693494
...	...	...	...	...	...	...
39	False	3	1	!*	3	0.542707
66	True	3	1	*!	2	0.542707
268	False	5	1	01	0	0.535615
188	True	6	1	*!*	0	0.535615
228	True	2	1	01	0	0.535615

The best configuration with `share_weights=True` selected was:

- `convolution_stride`: 7
- `convolution_step`: 1
- `pooling filter`: !*!
- `pooling stride`: 3

While the best configuration with `share_weights=False` selected was:

- `convolution_stride`: 1
- `convolution_step`: 1
- `pooling filter`: *!

- `pooling stride`: 2

Measuring them with the test set I obtain:

	share weights	accuracy	parameters
0	False	0.83	32
1	True	0.70	12

These are the results with the same convolution ansatz but with ansatz pooling (2):

	Share weights	Conv. stride	Conv. step	Pool. filter	Pool. stride	Mean validation scor
297	False	5	1	10	1	0.714770
311	False	1	2	10	3	0.714770
256	False	2	2	01	0	0.714770
98	False	4	2	*!	2	0.714770
101	False	5	2	*!	1	0.714770
245	False	6	1	01	1	0.707678

Since there are multiple configurations with the same validation score I try two of them on the test set:

	configuration	accuracy	parameters
0	1st	0.78	26
1	4th	0.80	14

Configuration number is referred to results above.

3.2 Convolution ansatz (b)

These are the results with pooling ansatz (1).

	Conv. stride	Conv. step	Pool. filter	Pool. stride	Mean validation score
71	4	1	*!	3	0.714770
276	7	2	01	0	0.714770
320	4	2	10	0	0.707678
224	1	1	01	0	0.707678
216	6	2	*!*	0	0.707678
300	6	1	10	0	0.707678

	configuration	accuracy	parameters
0	1st	0.683	32
1	2nd	0.750	20

3.3 Convolution ansatz (g)

For convolution ansatz (g) I decided to fix `share_weights=True` since the number of parameters would have grown from 30 to 130.

These are the results with pooling ansatz (1).

	Conv. stride	Conv. step	Pool. filter	Pool. stride	Mean validation score
230	2	1	01	2	0.714770
139	7	1	!*!	3	0.714770
296	5	1	10	0	0.714770
288	3	1	10	0	0.707678
254	1	2	01	2	0.707678
203	2	2	*!*	3	0.707678

I tried the first 3 configurations on the test set:

	configuration	accuracy	parameters
0	1st	0.750	36
1	2nd	0.867	36
2	3rd	0.750	36

After discovering the high accuracy of the 2nd configuration I also tried to set `share_weight=False` only for this particular configuration and this is the result I obtained on the test set:

	share weights	accuracy	parameters
0	False	0.883	136
1	True	0.867	36

I found these last two to be the most interesting architectures.

References

[1] Lourens, M., Sinayskiy, I., Park, D. K., Blank, C. and Petruccione, F. (2023). Hierarchical quantum circuit representations for neural architecture search. *npj Quantum Information* **9** 79.

[2] Anon. Building a Variational Quantum Classifier | Q-munity Tutorials.

[3] McClean, J. R., Boixo, S., Smelyanskiy, V. N., Babbush, R. and Neven, H. (2018). Barren plateaus in quantum neural network training landscapes. *Nat. Commun.* **9** 1–6.

[4] Pesah, A., Cerezo, M., Wang, S., Volkoff, T., Sornborger, A. T. and Coles, P. J. (2021). Absence of Barren Plateaus in Quantum Convolutional Neural Networks. *Phys. Rev. X* **11** 041011.

[5] Elsken, T., Metzen, J. H. and Hutter, F. (2019). Neural architecture search: A survey. *The Journal of Machine Learning Research* **20** 1997–2017.

[6] matt-lourens. (2024). hierarqcal. *GitHub.*

[7] Tzanetakis, G., Essl, G. and Cook, P. (2001). Automatic musical genre classification of audio signals.

[8] Anon. GTZAN Dataset - Music Genre Classification.

[9] McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E. and Nieto, O. (2015). Librosa: Audio and music signal analysis in python. In *SciPy* pp 18–24.

[10] Grant, E., Benedetti, M., Cao, S., Hallam, A., Lockhart, J., Stojevic, V., Green, A. G. and Severini, S. (2018). Hierarchical quantum classifiers. *npj Quantum Information* **4** 65.

[11] Hur, T., Kim, L. and Park, D. K. (2022). Quantum convolutional neural network for classical data classification. *Quantum Mach. Intell.* **4** 1–18.

[12] Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M. S., Alonso-Linaje, G., AkashNarayanan, B., Asadi, A., et al. (2018). Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968.*