Zero or one?

Willie Aboumrad, and **Sang Kim**

Abstract

Image classification is an important task in the field of computer vision. Every year, researchers develop better and better models to obtain more accurate classifications on standardized benchmarks. Recently, quantum computing approaches have been proposed.

In this challenge you'll work like an IonQ Apps Team scientist to explore image classification using quantum machine learning (QML) methods.

We'll consider a binary classification problem and we challenge you to design a hybrid classical-quantum neural network with the highest possible classification accuracy.

We've granted you *exclusive* access to unreleased features of IonQ's next-generation SDK to help you handle the orchestration and all the classical parts of the hybrid pipeline, so you can focus on the most interesting, and purely quantum aspects:

- loading image data onto a quantum backend using encoding circuit,
- distinguishing encoded state vectors using a trainable ansatz, and
- extracting useful features for classification by measuring a collection of observables.

1. Your mission

In this challenge you'll implement a hybrid quantum-classical image classification pipeline using `ionqvision`, IonQ's next-generation platform for QML computations, and `PyTorch`. In particular, you'll train a quantum-classical neural network to distinguish between two types of different handwritten digits in the MNIST database.

Tip

If you haven't used `PyTorch` yet, don't sweat it!

This challenge is about the **quantum** in "Quantum Machine Learning". The accompanying Section 1.b will set up everything you need, so you can focus on building your best quantum circuits!

The figure below illustrates our hybrid pipeline. The operations inside gray box make up the *quantum layer*. That's where the magic happens.

Published Jun 15, 2024

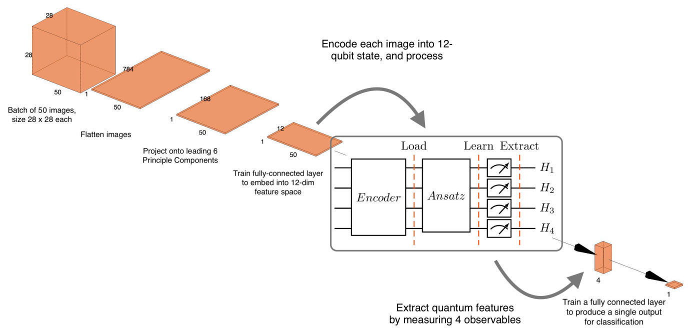

28
28
50
1
Batch of 50 images, size 28 x 28 each
784
50
Flatten images
168
1
50
Project onto leading 6 Principle Components
12
1
50
Train fully-connected layer to embed into 12-dim feature space
Encode each image into 12-qubit state, and process
Load
Learn
Extract
$Encoder$
$Ansatz$
H_1
H_2
H_3
H_4
4
1
Extract quantum features by measuring 4 observables
Train a fully connected layer to produce a single output for classification

Figure 1: A diagrammatic representation of our hybrid quantum-classical image classification pipeline

Your mission, should you choose to accept it, is to design the quantum layer that yields the **highest possible classification accuracy** (scored automatically on an unseen validation set). Simple, right?

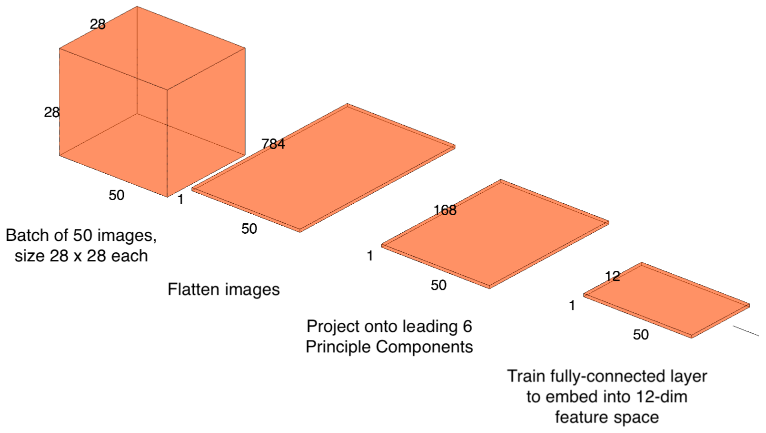
1.a. Classical, quantum, and then classical again

If you've made it thus far, congratulations!

You're about to embark on an exciting journey of quantum discovery. To begin, we'll take a closer look at the architecture of our hybrid neural network. It's important to keep in mind that going through our full *hybrid* pipeline requires **both** classical and quantum hardware, and software that facilitates communication between the two. We'll explain the classical and quantum parts of our pipeline in some detail below. Then we'll turn to the Section 1.b, which will get you up and running in no time!

1.a.i. *Classical pre-processing:*

The first three layers in our pipeline are purely classical. They implement fairly basic operations commonly encountered in classical image classification pipelines: first flatten each (single-channel) input image, perform dimensionality reduction using PCA, and then embed the compressed images into an even lower-dimensional feature space using a trainable fully-connected layer.


28
28
50
1
Batch of 50 images, size 28 x 28 each
784
50
Flatten images
168
1
50
Project onto leading 6 Principle Components
12
1
50
Train fully-connected layer to embed into 12-dim feature space

Note

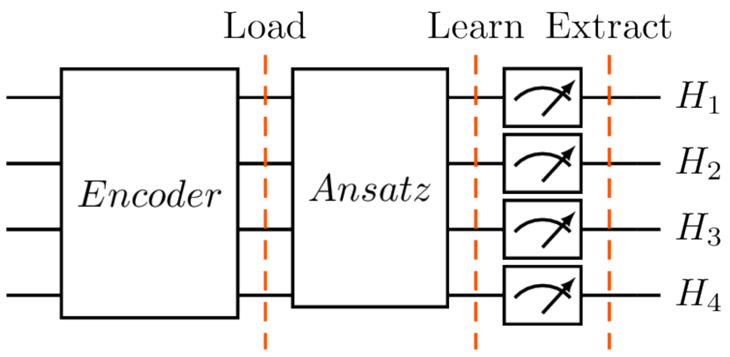
This part of the architecture is essentially **fixed**: the only thing you get to decide here is the dimension n of the resulting latent vectors. For instance, in Figure 1, $n = 12$.

1.a.ii. *Quantum Layer:*

Then comes the most interesting part: the "quantum layer." Our whole challenge revolves around this layer: the best design here will achieve the highest classification accuracy. And there's a **big prize** for that.

The quantum layer consists of three components:

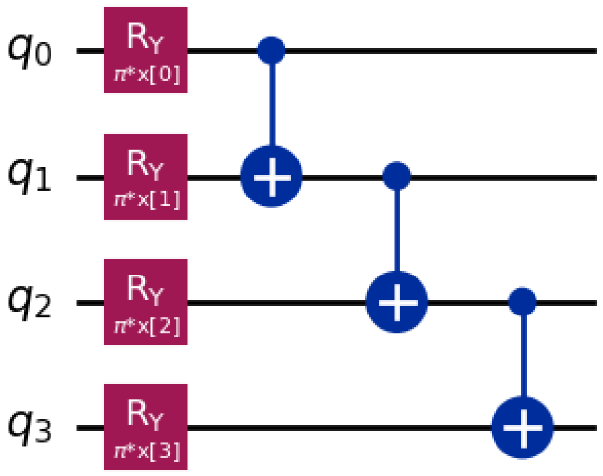
1. an encoding circuit used to load each n-dimensional latent vector into an n-dimensional qubit state,
2. a variational circuit on n qubits with trainable parameters used to learn how to separate encoded latent vectors in the high-dimensional multi-qubit state space, and
3. a list of m measureable quantities used to extract "quantum features" from the transformed n-qubit states for classification. (For instance, $m = 4$ in Figure 1.)



To get the ball rolling, look to the `ionqvision.ansatze.ansatz_library` module for inpiration. We've implemented some of the encoders and ansatze commonly encountered in the literature.

For instance, you could use the `AngleEncoder`, which has the following structure, for your first model.

```python
from ionqvision.ansatze.ansatz_library import AngleEncoder

encoder = AngleEncoder(num_qubits=4)
encoder.draw("mpl")
```

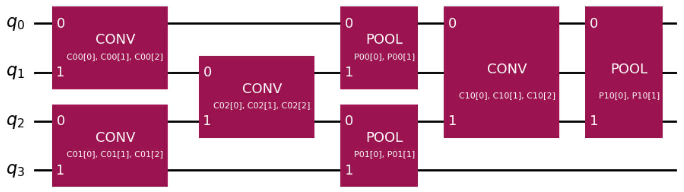


In addition, you can use its implementation as a **template** for your novel encoder designs!

```python
class AngleEncoder(VariationalAnsatz):
    """
    Implement a quantum circuit for higher-order sparse angle encoding.

    INPUT:

        - ``num_qubits`` -- number of qubits
        - ``entanglement_depth`` -- (optional) number layers of entangling CNOT
          gates: for each ``k`` in ``range(entanglement_depth)``, use gates
          ``CNOT(j, j + k + 1)``.
        - ``param_prefix`` -- (optional) string prefix for named circuit
          parameters

    EXAMPLES::

        >>> from ionqvision.ansatze.ansatz_library import AngleEncoder
        >>> ansatz = AngleEncoder(4, entanglement_depth=3, param_prefix="y")
        >>> ansatz.draw()
             ┌────────────┐
        q_0: ┤ Ry(π*y[0]) ├──■──────────────■─────────■──
             ├────────────┤┌─┴─┐            │         │
        q_1: ┤ Ry(π*y[1]) ├┤ X ├──■─────────┼────■────┼──
             ├────────────┤└───┘┌─┴─┐     ┌─┴─┐  │    │
        q_2: ┤ Ry(π*y[2]) ├─────┤ X ├──■──┤ X ├──┼────┼──
             ├────────────┤     └───┘┌─┴─┐└───┘┌─┴─┐┌─┴─┐
        q_3: ┤ Ry(π*y[3]) ├──────────┤ X ├─────┤ X ├┤ X ├
             └────────────┘          └───┘     └───┘└───┘
    """
    def __init__(self, num_qubits, entanglement_depth=1, param_prefix="x"):
        super().__init__(num_qubits)

        # Single-qubit angle encoders
        x = ParameterVector(param_prefix, num_qubits)
        [self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]

        # Entangling layers
        for k in range(entanglement_depth):
            top_qubit = 0
            while top_qubit < num_qubits - (k + 1):
                self.cx(top_qubit, top_qubit + k + 1)
                top_qubit += 1
```

Similarly, you can leverage the built-in `BrickworkLayoutAnsatz`, or the `QCNNAnsatz`, amongst others, for the trainable layer.

```
from ionqvision.ansatze.ansatz_library import QCNNAnsatz

ansatz = QCNNAnsatz(num_qubits=4)
ansatz.draw("mpl")
```



This ansatz is particularly interesting as it consists of a sequence of "convolution" filters interspersed with pooling operations that reduce the number of "active" qubits upon each layer. Note that our implemenation modifies the original design by replacing the mid-circuit measurements with controlled rotations.

If you'd like to try out this ansatz, be sure to implement the `QCNNAnsatz.ConvolutionBrickwork` and `QCNN.PoolingLayer` classes of your choosing!

```
class QCNNAnsatz(VariationalAnsatz):
    r"""
    Implement the Quantum Convolutional Network Ansatz (QCNN) as described in
    :cite:t:`2019:qcnn`.

    The quasi-local unitary $U_i$'s are entangling two-qubit gates with $6$
    variational parameters.
    They are laid out in a brickwork pattern with ``filter_depth`` layers.

    The pooling operations are implemented by two-qubit controlled rotations,
    with $2$ variational parameters.

    The circuit starts with ``num_qubits`` active qubits and then half the
    remaining qubits are discarded after each pooling operation until only a
    single active qubit remains. This final qubit is measured and the result is
    used for binary classification.
    """
```

```python
    class ConvolutionBrickwork(BrickworkLayoutAnsatz):
        """
        Implement the convolution filters for the :class:`.QCNNAnsatz`.
        """
        def __init__(self, num_qubits, num_layers, prefix=None, qubits=None,
initial_state=None):
            super().__init__(num_qubits, num_layers, blk_sz=3, prefix=prefix,
qubits=qubits, initial_state=initial_state)

        def two_qubit_block(self, theta, q1, q2):
            sub_circ = QuantumCircuit(2, name="CONV")
            sub_circ.ry(theta[0], 0)
            sub_circ.ry(theta[1], 1)
            sub_circ.rxx(theta[2], 0, 1)
            self.append(sub_circ.to_instruction(), [q1, q2])

    class PoolingLayer(BrickworkLayoutAnsatz):
        """
        Implement the pooling layer for the :class:`.QCNNAnsatz`.
        """
        def __init__(self, num_qubits, prefix=None, qubits=None):
            super().__init__(num_qubits, 1, blk_sz=2, prefix=prefix,
qubits=qubits)

        def two_qubit_block(self, theta, q1, q2):
            pool_op = QuantumCircuit(2, name="POOL")
            pool_op.crz(theta[0], 1, 0)
            pool_op.crx(theta[1], 1, 0, ctrl_state="0")
            self.append(pool_op.to_instruction(), [q1, q2])
```

While `ionqvision` does not provide built-in qubit observables, you can set these up using `qiskit` as follows.

```python
from qiskit.quantum_info import SparsePauliOp

# Measure the expectation value of X_0, Y_0, Z_0
quantum_features = [
    SparsePauliOp(["IIIX"]),
    SparsePauliOp(["IIIY"]),
    SparsePauliOp(["IIIZ"])
]
```

It's important to keep in mind that the encoder, the ansatz, and the quantum feature vector are highly interrelated: the way you embed latent vectors into (multi-)qubit state space should dictate how you choose to transform the encoded state vectors, which should in turn inform what features you decide to measure.

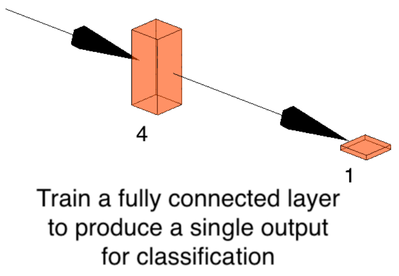
The best model will likely exploit synergies resulting from intentional co-design of the three components.

Important

This is where you should let your imagination run free! Get creative, and show us what you've got. Feel free to leverage the power of the internet and use every resource at your disposal.

1.a.iii. *Classical post-processing:*

After the quantum layer, feature vectors in the classification pipeline return to the classical device for some post-processing. In particular, we train a fully-connected layer with a scalar output to minimize the binary cross-entropy between the final value and the input image's true label.


4
1
Train a fully connected layer to produce a single output for classification

This final stage is intentionally light, to ensure the quantum layer is the star of the show.

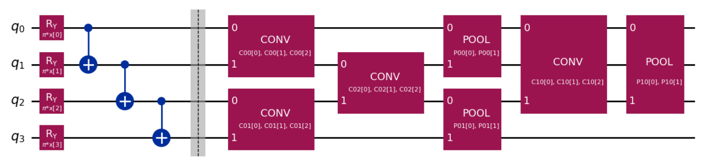
Note

This part of the architecture is totally **fixed**: the input dimension (m) is determined by the quantum feature vector, and the output is always a scalar.

1.b. *Training your model*

Once you've settled on your quantum layer, you can sit back, relax, and let `ionqvision` do the heavy lifting. First, set up your classifier and verify it's working as expected.

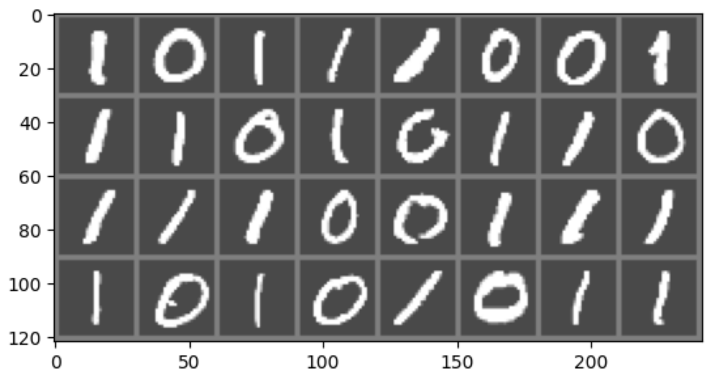
```python
from ionqvision.modules import BinaryMNISTClassifier

# Set up your classifier and check out your quantum layer
classifier = BinaryMNISTClassifier(encoder, ansatz, quantum_features)
classifier.quantum_layer.layer_qc.draw("mpl")
```



In case you're wondering, `BinaryMNISTClassifier` is a standard `torch.nn.Module`; at this point, all the parameters in your quantum layer have been registered with `torch` and the `autograd` will automatically compute the relevant gradients during the backward pass. Nothing else to worry about!

```python
# Verify the images loaded correctly
classifier.visualize_batch()
```

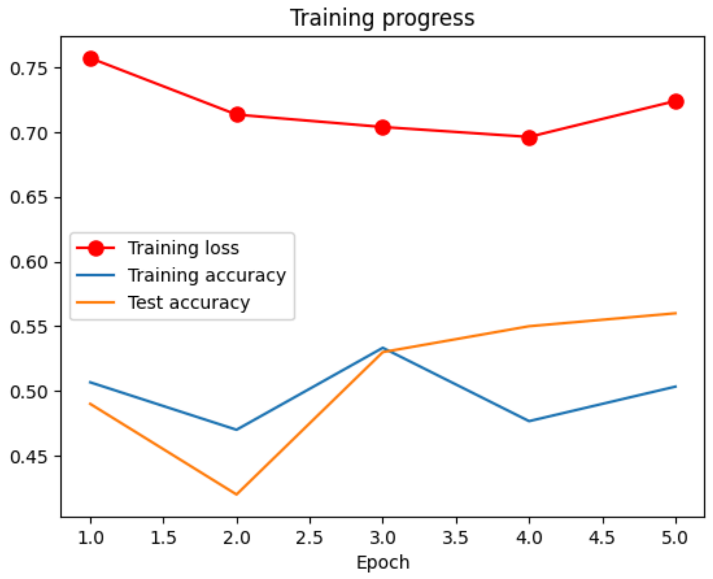
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range [0.28789353..1.9107434].
```



Now train your model. Use the `config` dictionary to control lower-level aspects of the training, like the number of `epochs`, the learning rate, the `betas` used by Adam, etc.

```python
# Get a (pre-processed) training and test set
train_set, test_set = classifier.get_train_test_set(train_size=300,
test_size=100)

# Configure model training hyper parameters
config = {
    "epochs": 5,
    "lr": 0.1,
    "batch_size": 50,
    "betas": (0.9, 0.99),
    "weight_decay": 1e-3,
    "clip_grad": True,
    "log_interval": 6,
}

# Train and plot the results
classifier.train_module(train_set, test_set, config)
classifier.plot_training_progress()
```

```
epoch:   1 | loss: 0.757
lr: 0.1000 | processed     6/    6 batches per epoch in 77.96s (0.32s forward /
12.19s backward)
Model achieved 50.667%  accuracy on TRAIN set.
Model achieved 49.000%  accuracy on TEST set.

epoch:   2 | loss: 0.714
lr: 0.1000 | processed     6/    6 batches per epoch in 78.93s (0.32s forward /
12.21s backward)
Model achieved 47.000%  accuracy on TRAIN set.
Model achieved 42.000%  accuracy on TEST set.

epoch:   3 | loss: 0.704
lr: 0.1000 | processed     6/    6 batches per epoch in 77.75s (0.30s forward /
12.09s backward)
Model achieved 53.333%  accuracy on TRAIN set.
Model achieved 53.000%  accuracy on TEST set.

epoch:   4 | loss: 0.696
lr: 0.1000 | processed     6/    6 batches per epoch in 78.70s (0.30s forward /
12.10s backward)
Model achieved 47.667%  accuracy on TRAIN set.
Model achieved 55.000%  accuracy on TEST set.

epoch:   5 | loss: 0.724
lr: 0.1000 | processed     6/    6 batches per epoch in 77.56s (0.29s forward / 
12.22s backward)
Model achieved 50.333%  accuracy on TRAIN set.
Model achieved 56.000%  accuracy on TEST set.
```



The model achieved '50.33%' on the **training** set and '56.00%' on the **test** set.

1.b.i. *Submitting your model for grading:*

Now comes the final thrill. Use your `classifier`'s `submit_model_for_grading` method to, well, submit your model for grading!

Your quantum layer and your model's trained weights will be serialized and then reconstructed in a sanitized environment. We'll run inference using the reconstructed model on an unseen validation set to compute your accuracy score.

Be sure to check our live leader board after a few minutes to see where your team stands!

```
classifier.submit_model_for_grading(path_to_repo="")
```

```
Success! Submitted trained_models/
model_feafb8bfa40f89677cfe3626f5c4b115ada161a2.zip for grading!
```

Note

The `submit_model_for_grading` method prepares an archive containing your trained model's weights, and everything that's needed to rebuild your quantum layer. The archive is stored in the `trained_models` directory in your copy of our repository.

Caution

Don't change the archive's name or its contents.

Good luck now! We hope you enjoy this exciting challenge.

2. Environment set up

To complete this challenge, you'll need a local installation of `ionqvision`. You can download the source files by `git clone`-ing the challenge repository and then installing using `pip`.

We recommend installing in a virtual environment, since `ionqvision` requires a number of dependencies, including minimum versions of `torch` and `qiskit`, and `qiskit_ionq`.

The following code snippet illustrates a minimal installation on Linux/macOS. It creates a virtual environment using `pip` and `venv`, activates it, and then installs `ionqvision`.

```
python3 -m venv .venv
source .venv/bin/activate
pip install -e /path/to/ionqvision
```

You could also use `conda` or any package manager to set up your development environment.

3. Leader board

Check out the latest challenge results here!

Tip

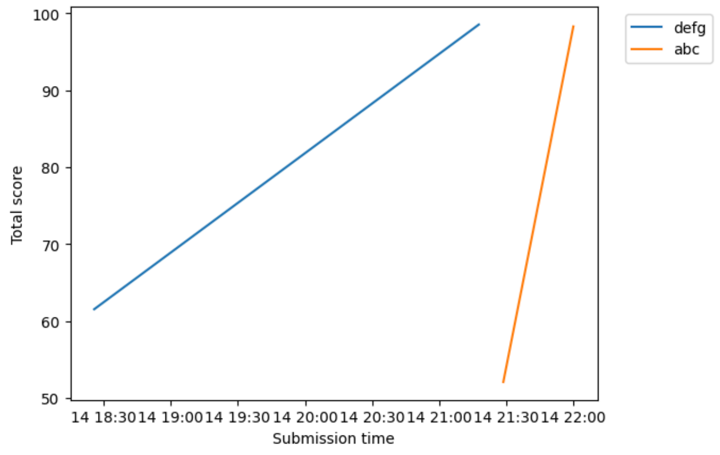
Be sure to refresh the page to get the latest results.

3.a. Top 10 teams

Check out the evolution of the leading teams's progress!

```
import matplotlib.pyplot as plt
import pandas as pd

# Rules of the game
MAX_GATES = 20
MAX_PARAMS = 55

# Leaderboard config
NUM_LEADERS = 2

# Helper function to compute total submission score
def compute_score(entry):
    # Accuracy
    score = entry.Accuracy

    # Gate penalty
    score += (MAX_GATES - entry.Gates) / MAX_GATES

    # Params penalty
    score += (MAX_PARAMS - entry.Parameters) / MAX_PARAMS
    return score

# Retrieve submissions and compute total score
results = pd.read_csv("results.csv", index_col=False)
results["Timestamp"] = pd.to_datetime(results.Timestamp, unit="s")
results.insert(1, "Score", results.apply(compute_score, axis=1))
```

```
# Get top teams, sorted by their highest scoring submission
gpby = results.groupby("Team")
top_teams = results.iloc[gpby.idxmax().Score].sort_values("Score",
ascending=False)
leaders = top_teams[:NUM_LEADERS].reset_index(drop=True)

# Extract leading teams' data
agg = gpby.agg(list)
[plt.plot(agg.loc[team].Timestamp, agg.loc[team].Score) for team in
leaders.Team]

# Make the plot pretty
plt.xlabel("Submission time")
plt.ylabel("Total score")
plt.legend(leaders.Team, bbox_to_anchor=(1.04, 1), loc="upper left");
```



```
leaders.rename(columns={"Score": "Top Score"})
```

	Team	Top Score	Accuracy	Gates	Parameters	Commit	Timestamp
0	defg	98.55	99.7	27	99	130923	2024-06-14 21:17:43
1	abc	98.30	98.3	20	55	128jkd	2024-06-14 21:59:59

3.b. Submission history

Please find all the challenge submissions below. Most recent submissions are at the top!

```
results.sort_values("Timestamp", ascending=False).reset_index(drop=True)
```

	Team	Score	Accuracy	Gates	Parameters	Commit	Timestamp
0	Willie Aboumrad	43.100000	42.0	10	22	35cd5c	2024-06-15 01:15:43
1	Willie Aboumrad	45.100000	44.0	10	22	10167a	2024-06-15 01:15:32
2	Willie Aboumrad	47.100000	46.0	10	22	34ae87	2024-06-15 01:15:17
3	Willie Aboumrad	42.100000	41.0	10	22	e6627d	2024-06-15 01:14:50

	Team	Score	Accuracy	Gates	Parameters	Commit	Timestamp
4	runner	43.100000	42.0	10	22	b65b43	2024-06-15 01:05:38
5	abc	98.300000	98.3	20	55	128jkd	2024-06-14 21:59:59
6	abc	52.100000	52.1	20	55	ai21j0	2024-06-14 21:28:41
7	eeee	70.154545	70.1	20	52	3k2l1l	2024-06-14 21:28:34
8	defg	98.550000	99.7	27	99	130923	2024-06-14 21:17:43
9	defg	61.554545	62.5	40	52	2479da	2024-06-14 18:25:48