Master's Degree in Artificial Intelligence

Exploring the Potential of Quantum Computing for Reinforcement Learning: A Case Study on the Cart Pole Environment

Macaluso Girolamo, Mistretta Marco

January 14, 2023

1 Introduction

Deep Reinforcement Learning (RL) has recently achieved impressive results on a variety of challenging tasks, including playing games, controlling robots, and optimizing complex systems. These successes have been made possible by advances in Machine Learning algorithms and hardware, which have enabled the training of Deep Neural Networks (DNNs) to model complex environments and make decisions based on sensory input.

However, the computational resources required for Deep RL can be huge, especially when the RL tasks involve high-dimensional observation spaces or long time horizons. In such cases, even the most powerful classical computers may be insufficient to achieve acceptable training times or sample efficiency.

To overcome these limitations, a promising approach is to use Quantum computers to accelerate the training of DNNs for RL. Quantum computers can potentially offer exponential speedups over Classical computers for certain computational tasks, due to their ability to represent and manipulate complex data in *superposition* and *entanglement*.

In this paper, we re-implement the quantum circuit introduced in "*Quantum agents in the Gym: a variational quantum algorithm for deep Q-learning (by Skolik et al)*" to learn the Cart Pole Environment. The algorithm combines Variational Quantum Algorithms (VQAs) and Deep Q-learning (DQL). Our Quantum agent is implemented in Qiskit, an open-source quantum computing framework.

Our main contribution is to understand if a Quantum agent can learn to solve RL more efficiently than a Classical agent, using only a small number of qubits and quantum gates.

We also analyze the impact of different hyper-parameters and parameters on the performance of our quantum agent, and discuss potential directions for future research.
Overall, our results provide evidence that quantum computers have the potential to significantly enhance the capabilities of RL algorithms, and open up new possibilities for solving complex optimization and decision-making problems in real-world applications.

2 Quantum Computing

The use of Quantum Machine Learning (QML) algorithms has the potential to significantly improve the performance of Machine Learning tasks, but it also comes with several technical challenges linked to the architecture of quantum computers that are now available, NISQ (Noisy Intermediate-Scale Quantum) devices, which are quantum computers with less than 50 qubits. Those are prone to noise and errors, which can significantly impact the accuracy of QML algorithms. These errors can arise from various sources such as *decoherence* and *gate imperfections*, and can be difficult to mitigate. The error scale with the depth of the quantum circuit so the complexity of suitable circuit is limited. Developing methods to reduce the impact of noise and errors on QML algorithms is an active area of research. A major challenge in QML is efficiently transferring data between classical and quantum systems. This requires algorithms and techniques to facilitate the exchange of information between these two types of systems, such as: base encoding, angle encoding, amplitude encoding and *arbitrary encoding* (the one we used in this work).

3 Reinforcement Learning

Reinforcement Learning is a type of Machine Learning in which an agent learns to interact with an environment in order to maximize a reward function $r(s, a)$. The goal of the agent is to learn a *policy*, which is a function that maps states of the environment to actions. The policy is learned through *trial and error*, as the agent receives a reward for each action it takes in the environment.
The important assumption that is usually made in Reinforcement Learning is that the transition model respect *Markov property*, so the future state of the environment is independent of the past states given the current state and the actions taken.

$$s_{t+1} = T(s_t, a_t) \tag{1}$$

Another important concept in Reinforcement Learning is the *total reward*, which is the sum of the rewards received by the agent over a given period of time. The total reward is used to evaluate the performance of the agent and to compare different policies.

$$R = \sum_{t=0}^{T} r(s_t, a_t) \tag{2}$$

A further one relevant quantity is the *expected reward*, which is the average reward that an agent can expect to receive over a long period of time, given a particular policy. The expected reward can be used to compare different policies and to identify the optimal policy, which is the policy that maximizes the expected reward.
The *value function* is a function that represents the expected return of an agent in a given state. It is denoted by $V(s)$ for a given state s, and represents the sum of the expected future rewards that the agent can expect to receive, starting from state s, following a particular policy π.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t | s_t = s] \tag{3}$$

The *state-action value function*, also known as the *Q-function*, is a function that represents the sum of the expected future rewards that the agent can expect to receive, starting from state s, taking action a, and following a particular policy, it is denoted by $Q(s, a)$. The Q-function is used to evaluate the performance of the agent and to guide the learning process.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a] \quad (4)$$

3.1 Deep Q-Learning

Deep *Q-learning* is a classical reinforcement learning algorithm, that uses deep neural networks to approximate the Q-function. The network takes as input the current state of the environment and outputs the Q-value for each possible action. During the learning process, the network is trained to minimize the difference between the predicted Q-values and the "target" Q-values, which are calculated using the Bellman equation. The *Bellman equation* is a recursive equation that defines the relationship between the Q-values of a state and the Q-values of the subsequent states.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_{t+1} + \gamma \max_{a} Q^{\pi}(s_{t+1}, a_{t+1})] \quad (5)$$

The optimization target is the mean square error between the predicted and target q-value:

$$\frac{1}{N} \sum (r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2 \quad (6)$$

The training process is typically done using *experience replay*, which involves storing a dataset of past experiences and sampling from this dataset to compute the target Q-values. This allows the network to learn from a wide range of experiences and can help to stabilize the learning process. One challenge in deep Q-learning is balancing the **exploration and exploitation trade-off**. In order to learn the optimal Q-function, the agent must explore the environment and try different actions. However, if the agent spend too much time exploring, it may not make sufficient progress towards the ultimate goal. On the other hand, if the agent spends too much time exploiting its current knowledge, it may miss out on valuable opportunities. To address this issue, deep Q-learning algorithms often use exploration policies which determine the probability of selecting a random action rather than the action with the highest Q-value. After the Q-function has been trained, the policy can be derived from the Q-function by selecting the action with the maximum Q-value for each state. This is known as the greedy policy, as it always selects the action that appears to be the best choice based on the current knowledge of the Q-function. This policy leads to an optimal path if the q-function is optimal.

4 Quantum Deep Q-Learning

Variational quantum algorithms are considered the most promising approach to quantum machine learning. VQAs are a class of quantum algorithms that involve adjusting the parameters of a quantum circuit, or ansatz, to optimize a cost function. They are particularly well-suited for NISQ devices, as they can be implemented using shallow circuits.

In the context of reinforcement learning a Parameterized Quantum Circuit can be used as a powerful function approximator, like deep neural networks. One advantage of using PQCs for deep Q learning is that quantum states are more expressive, allowing the Q function to capture complex patterns and relationships in the data. This can be particularly useful for problems with large or continuous state spaces, where a classi-

cal neural network may struggle to capture the relationships, at the same time the limited number of qbits limit the uses of this techniques.

The main decisions that are needed to make in order to implement a Quantum Deep Q-learning algorithm are: the *ansatz*, the *encoding strategy*, the *readout strategy* and the *optimization*.

4.1 Ansatz

A good ansatz should be able to represent and manipulate quantum data in a flexible and expressive way. It should also be compact, with a small number of parameters and quantum gates, which can make it easier to optimize and to train and can reduce the resources required to implement and run the ansatz.

There is a trade-off between the number of gates, or in general the depth of the circuit, and the expressivity, because complex circuit are more expressive but at the same time more prone to error and hard to optimize.

The ansatz used in this work is the one in Figure 1, consist in two parameterized rotation on each qbit, first along Y then Z, and a daisy chain of CZ Gates to add entangling. This model is widely used and is even implemented as a base model in qiskit called *TwoLocal*. This circuit is repeated multiple times in order to increase the expressivity.

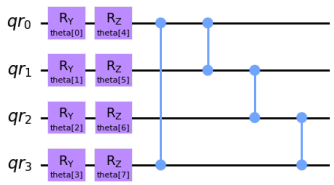


Figure 1: Ansatz

4.2 Encoding strategy

There are multiple possible encoding strategies to encode a classical state in a quantum circuit.

- Base encoding that encodes states as elements of the computational base, similarly to one-hot encoding, this is not suitable for continuous spaces as the one we consider in this work.
- Amplitude encoding represents classical data point as the amplitudes of a quantum state, requiring less qubits, however this is sensitive to noise and errors, as it relies on precise control of the amplitudes.
- Angle encoding encodes classical data into the rotation angles of qubits.
- **Arbitrary encoding** that encodes the N features of the classical state as N parameterized rotation gates, it's implemented with a costant depth quantum circuit, so it's efficient to be used with NISQ devices

The latter is the one used in this work, the input is encoded through R_x rotation. To represent continuous state as rotation angles the features have to be scaled in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$, this can be done through an arctan function. The scale of feature can be really different so passing it directly to arctan can make the train slower or even diverge, so the features are first scaled with **trainable weights** (w_d), thus the model can choose the best scaling for the input itself. The input rotation is:

$$x'_i = arctan(x_i \cdot w_{d_i}) \tag{7}$$

PQCs can be represented as a Fourier series, where the degree depends on the number of times the data encoding is repeated, so to further increase the model expressivity the data encoding can be repeated both in parallel, adding qbits, or serially, through data re-uploading. In the presented approach we will use **data re-uploading** at the beginning of each repetition of the ansatz, with the **same weights**.

4.3 Read-out Strategy

The output of the circuit represents the Q-value for each action in the input state, so the output must be continuous and the scale has depends on the environment considered. The latter is really crucial because the Q-values scale are high-dependent on the task and the actual performance of the agent, so **trainable output weight** are added to better scale the outputs and improve convergence speed. So the Q-value for state s and action a is:

$$Q(s,a) = \frac{\left\langle 0^{\otimes n} | U_\theta^\dagger O_a U_\theta | 0^{\otimes n} \right\rangle + 1}{2} \cdot w_{o_a} \quad (8)$$

Where the observable O_a is a Pauli-ZZ operator on the pair of q-bits corresponding to *action a*.

4.4 Optimization

There are several optimization techniques that can be used to optimize the parameters of a parameterized quantum circuit. These techniques allow to adjust the parameters of the PQC to minimize a cost function. The most effective techniques are the one based on gradient descend, such as stochastic gradient descend, Adagrad or Adam. Other approach are based on genetic optimization, techniques that are inspired by the process of natural evolution. They involve creating a population of candidate solutions and using evolutionary operations, such as selection, crossover, and mutation, to evolve the population towards better solutions. This techniques are usually very slow to converge and require a lot of parameter tuning to work effectively. So in this work we stick on **gradient based optimization**.

To perform such optimization we need to compute the gradient of our system, for the classic parts such as the output weights we can use the standard automatic differentiation, for the PQC there are different strategies. One is to use the finite difference gradients, that are usually a bad idea in classical computation due to error propagation and its the same for PQC if we consider NISQ devices. Another approach, the one used in this project, is to use analytic gradients computed through **parameter shift rule**, that can be used when the circuit consist only of Pauli rotation.

$$\frac{\partial f}{\partial \theta_i} = \frac{f\left(\overrightarrow{\theta} + \frac{\pi}{2}\overrightarrow{e_i}\right) - f\left(\overrightarrow{\theta} - \frac{\pi}{2}\overrightarrow{e_i}\right)}{2} \quad (9)$$

5 The Cart Pole Environment

The cart pole problem is a well-known challenge in reinforcement learning, where an agent must learn how to balance a pole on top of a cart. The agent must decide between **two actions** at each step: depending on the actual state he can push the cart left or right. The state of the environment is determined by **four state-variables**: the *pole's angle* and *position*, as well as the *cart's position* and *velocity*. The gent must learn to anticipate the future **consequences of its actions**, as the pole becomes increasingly unstable over time. By learning to balance the pole on the cart, the agent can demonstrate its ability to learn about the environment, and the algorithm implemented can prove his stability and robustness. The relative low dimension of state space and the low discrete action space make this environment suitable for an analysis on NISQ devices, that's why the Cart Pole Environment has been chosen as case study for analysis and comparison with the classical approach.

5.1 Episode End

The episode length of this environment depends on the performance of the agent,

because this is a surviving task. The episode ends if any one of the following occurs:

- Failure 1: Pole Angle is greater than $\pm 12^{\circ}$
- Failure 2: Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
- **Success**: Episode length is greater than 200

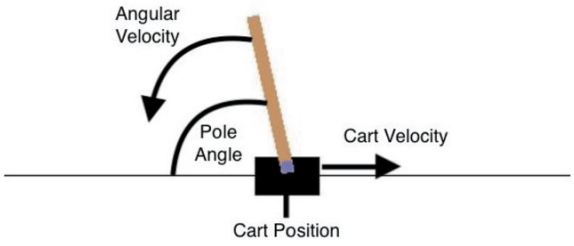


Figure 2: Cart Pole Environment

6 Implementation Details

The full implementation source code is available on our **GitHub**. For the implementation we need a bunch of classes and functions for implementing a quantum neural network (QNN) to solve the cart pole problem using Qiskit. The **EncodingLayer** class define a simple feed-forward neural network layer that encodes input data and applies the activation function to it 7. The **OutputLayer** class define another feed-forward neural network layer that scales the output of the QNN to the range of valid actions for the cart pole problem. The **CircuitBuilder** class use the **CircuitQNN** class from Qiskit to create a QNN circuit. The *buildQuantumCircuit* function define the structure of the QNN circuit, with the method **createCircuit** returns a **TorchConnector** that is a Pytorch module that encapsulates the quantum circuit, allowing us to easily perform the backward pass and the optimization. The **ReplayMemory** class define a cyclic buffer that stores past transitions experienced by the QNN as it interacts with the cart pole environment. Finally, the **Model** class define a reinforcement learning agent that uses the QNN to learn a policy for the cart pole problem. In the model we implemented the training loop for the QNN that is being used to solve the cart pole problem. The training loop is implemented in the *train* method, which takes as input the number of epochs to run and the batch size for each epoch. The train method use the Adam optimizer to optimize the weights of the QNN, encoding layer, and output layer. **Note that** here are used **three optimizers**, one for each part of the model. There was our fix to a library bug that makes impossible the optimization-step for the classical weights when you use only an optimizer. The loss function used is the *mean squared error* to compute the error between the predicted and target Q-values. The train method also use the *stretched exponential epsilon decay schedule* 8 for the exploration rate.

The *getQvalue* method perform the forward pass through the model, transforming the environment state into angles, through the encoding layer. Then binding it into the QNN, the measurement on the computational basis are summed to obtain $Pauli - ZZ_{01}$ and $Pauli - ZZ_{23}$, that are the unscaled q-values for left and right actions. Finally the output layer is applied to obtain the final q-values.

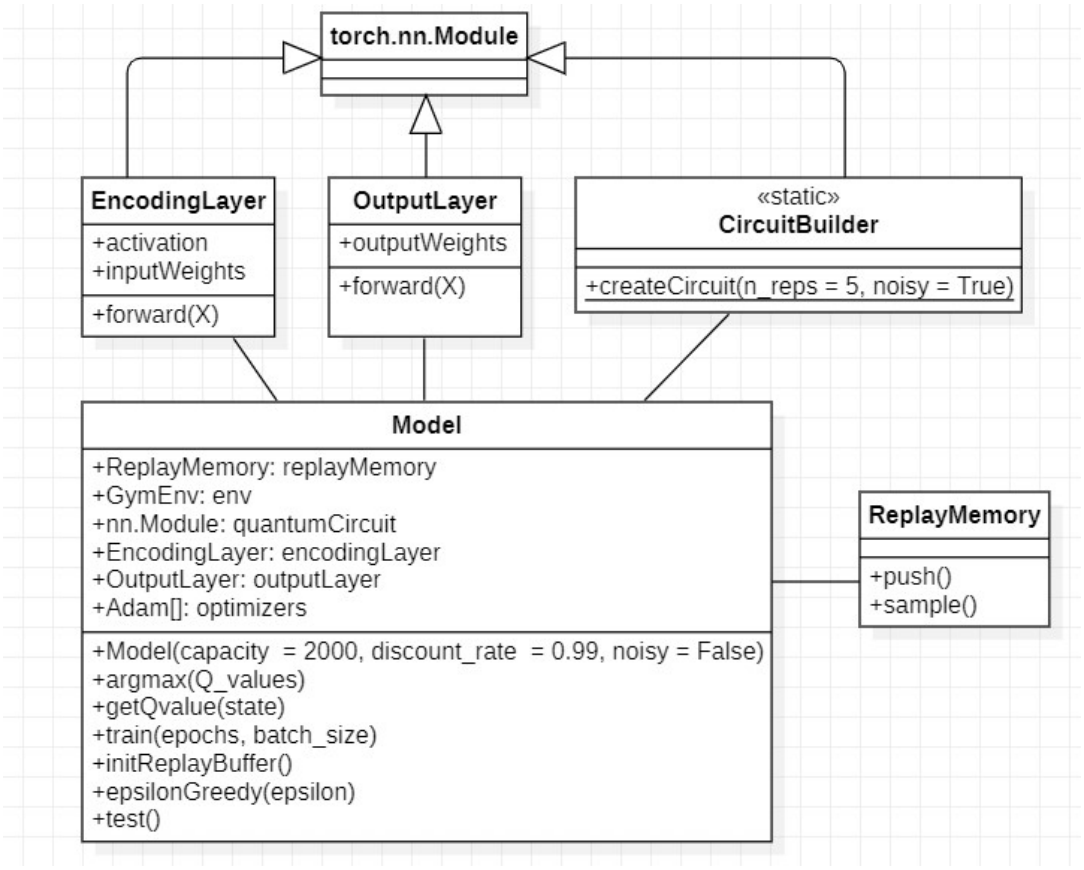


Figure 3: UML Diagram

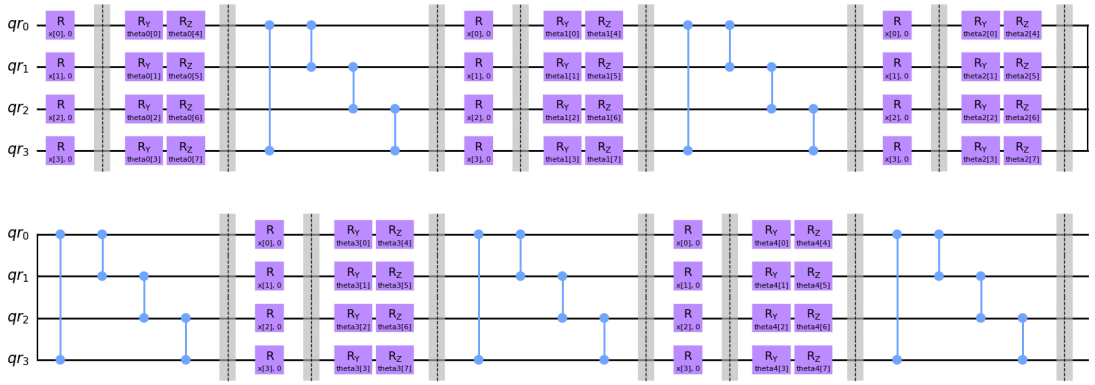


Figure 4: Circuit

7 Training

The quantistic part of the model was initially simulated only on **Aer Simulator**, due to the limitation of evaluating the circuit on a real machine, in terms of time because of the lines to execute jobs and in term of licences because of the enormous number of evaluation needed for forward and backward pass.

The model training phase turned out to be more difficult than expected. The model learned well in the exploration phase, but failed to fully resolve the environment (i.e., failed to achieve a stable average of at least 190 steps per 100 consecutive episodes). Even after doing a grid-search of the training hyperparameters, this didn't work. We later realized that the fundamental problem was that, *unlike the classical approach, our model was not robust with respect to the type of epsilon scheduling decay* used. We therefore tested various epsilon decays: linear, exponential...etc.
Our found is that the best one in this case of study was a **stretched exponential decay** with three optimization hyperparameters, implemented by *Subbiahh Natarajan* (for more details refer to the appendix 8). This effective scheduling allowed us a good exploration phase, and at the same time, a sudden transition to the exploitation phase, aimed at solving the environment in an absolute way. We were therefore able to train a model.

7.1 Comparison with Classical Approach

Then we started training a classic Deep Q Learning (DQN) vanilla algorithm found in the tutorials section of Torch. Our choice was to train the classical Model with the same hyperparameters used for the quantum approach, so as to be able to make a fair comparison (for example, it would not have been fair to use a larger batch size in the classical training, which we was not allowed to do in the quantum approach quantum cause of the computational time of the qiskit simulator). Against all odds, the vanilla model of Torch, famous for solving the cart pole environment quickly (5 minutes 5.143 seconds) and effectively, did not converge to the optimal solution. In a rather unusual way, when he got close to doing 100 consecutive episodes of solving the task, he then began to diverge and move further and further away from the optimal solution. This type of behavior is the same that we observed in the training phase of the quantum approach, in the ablation studies phase regarding the epsilon scheduling to be used. By changing the stretched exponential decay with a simple exponential decay, in fact, the trend is back to being more or less linear, as shown on the official Torch page.

7.2 Effect of the noise of NISQ devices

As a last experiment we decided to test the execution on a quantum machine (i.e. **IBM Oslo**). Unfortunately a single evaluation of the circuit (with 1024 runs) took no less than 10 minutes, given the waiting queue of the machine. For this reason we decided to test on a simulator provided once again by the *Aer Simulator* package, which simulated the noise produced by a real device based on its average recent noise, we used IBM Oslo noise model. The training on noisy simulator was not robust compared to our circuit and took way longer to compute a single iteration, so we were not able to obtain a satisfactory model in a short time. So we decided to test the performance of the noise-less model with the noisy simulator, the results were quite bad, the agent survived, on average, only for half of the step required to complete the task. This confirms that this approach is susceptible to noisy results, maybe a model trained with noise could better handle this task.

7.3 Comparison Result

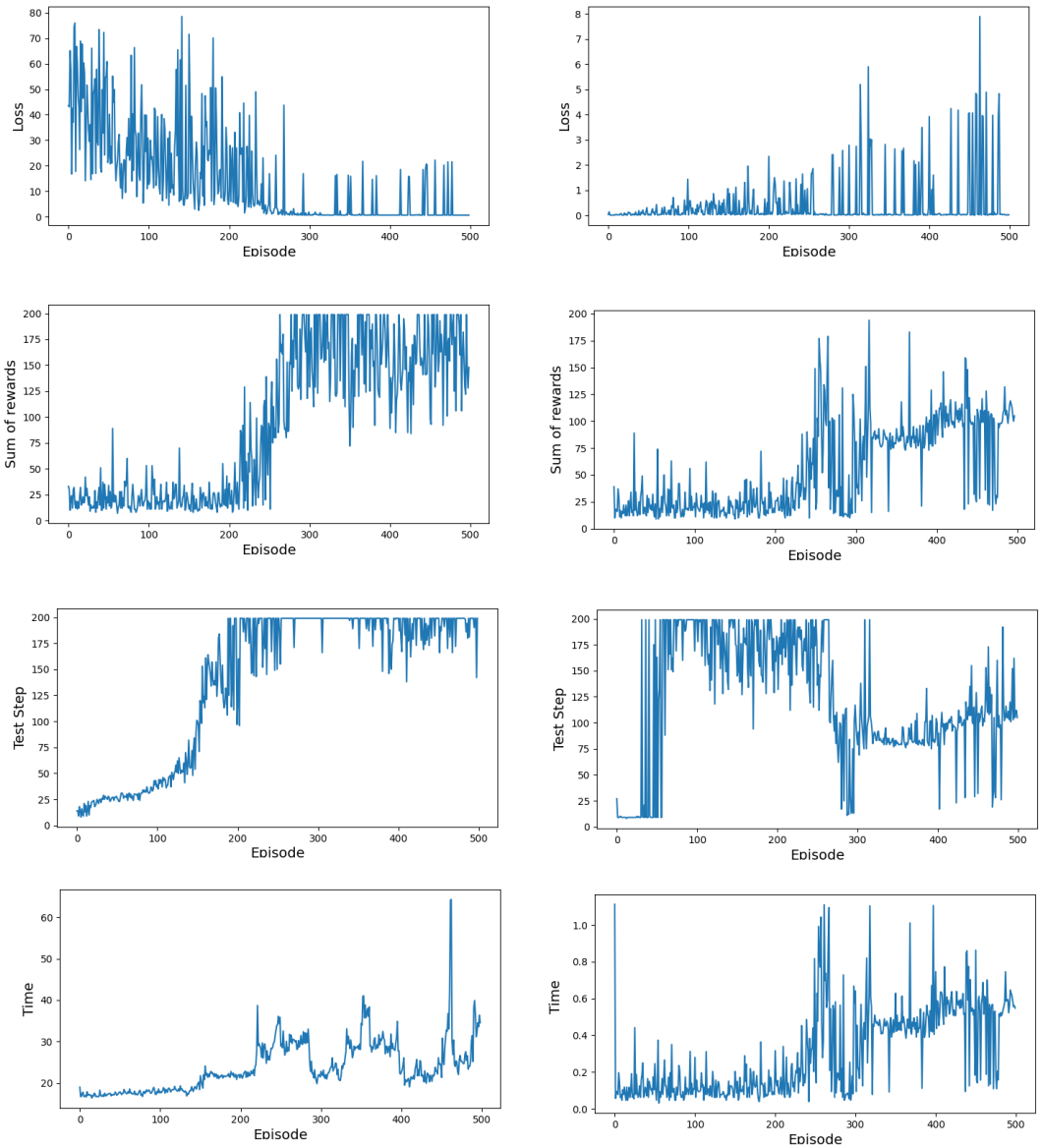


Figure 5: Comparison between Quantum (Left) and Classical (Right) Approach

From the graphs above 5, it's evident that quantum model training is globally more robust than its classical counterpart. In detail, as can be seen in the appendix 8, quantum training has made the transition from exploration to exploitation at the most appropriate time, i.e. when the model began to converge and the acquired experience began to be valid. This is the result of our targeted optimization of the model parameters and hyperparameters. However the classic model is undoubtedly faster than the quantum approach despite it has 17'000 trainable parameters, unlike the quantum approach which has only 46 trainable parameters.

8 Conclusion

In this work we have analyzed the use of quantum circuits in the field of reinforcement learning implementing a quantum version of deep q-learning, resolving Cart Pole, a simple continuous state environment. We have discovered how critical is the trade-off between exploration and exploitation for this approach, we have compared the performance with the classical version and analysed the impact of the noise of NISQ devices. This approach has great development potential and will be of course crucial in the near future with the availability of devices with more qbits and less noise, so it will be possible to scale this technique to more complex environments.

References

[1] Qiskit: An Open-source Framework for Quantum Computing. `https://qiskit.org/`.

[2] Peter Skolik, Alexander Melnikov, Kae Mitarai, Yudai Yamamoto, and Hartmut Neven. Quantum agents in the gym: a variational quantum algorithm for deep q-learning. *arXiv preprint arXiv:2103.08061*, 2021.

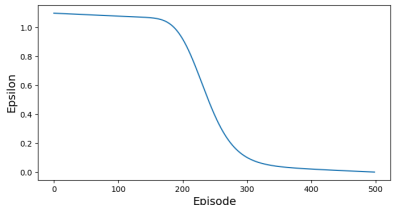
[3] Nicola Dalla Pozza, Lorenzo Buffoni, Stefano Martina, and Filippo Caruso. Quantum reinforcement learning: the maze problem, 2021.

[4] V. Saggio, B. E. Asenbeck, A. Hamann, T. Strömberg, P. Schiansky, V. Dunjko, N. Friis, N. C. Harris, M. Hochberg, D. Englund, S. Wölk, H. J. Briegel, and P. Walther. Experimental quantum speed-up in reinforcement learning agents. *Nature*, 591(7849):229–233, mar 2021.

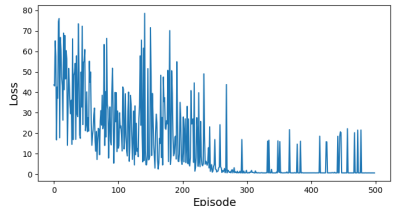
[5] Stretched Exponential Decay function for Epsilon Greedy Algorithm. `https://medium.com/analytics-vidhya/stretched-exponential-decay-\function-for-epsilon-greedy-\algorithm-98da6224c22f`.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

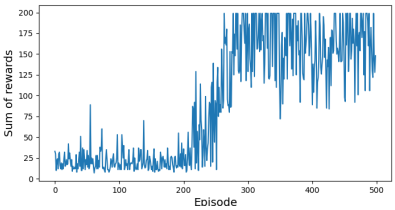
500-epochs QDQL



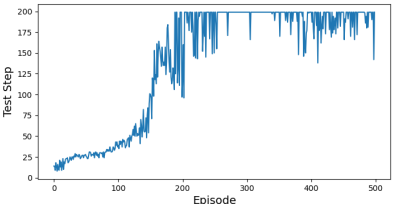
Epsilon Decay



Loss History



Sum of Rewards



Steps with Greedy-Policy

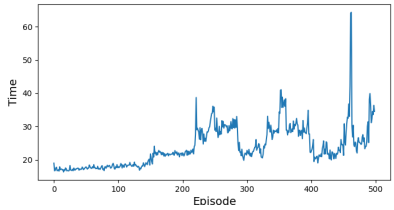
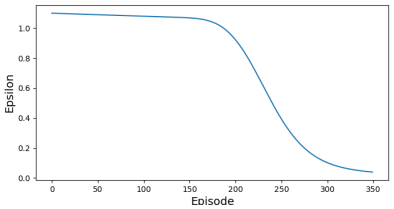
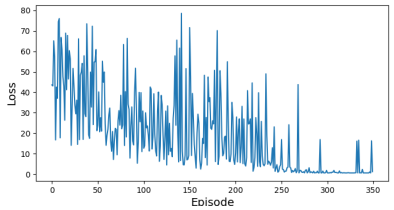


Figure 6: Epochs Duration Time

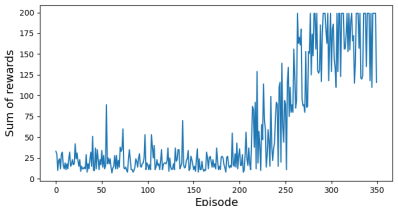
First 350-epochs QDQL



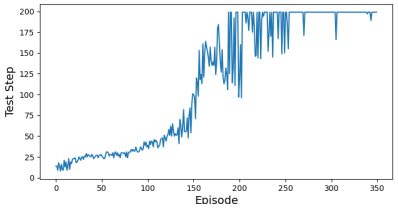
Epsilon Decay



Loss History



Sum of Rewards



Steps with Greedy-Policy

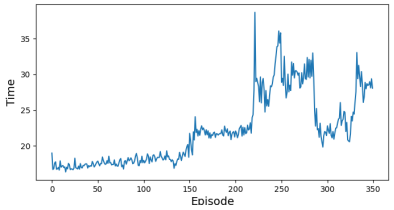


Figure 7: Epochs Duration Time

Appendix: Source Code

The full implementation source code is available on our **GitHub**

```
class EncodingLayer(torch.nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.in_dim = in_dim

        self.activation = torch.arctan
        self.inputWeights = torch.nn.Parameter(Tensor(np.ones(shape=in_dim)))
        torch.nn.init.uniform_(self.inputWeights, -1, 1)

    def forward(self, X):
        # a = X.repeat(1, 5)
        return self.activation(X * self.inputWeights)
```

Figure 8: Encoding Layer Class

```
class OutputLayer(torch.nn.Module):
    def __init__(self, outdim):
        super().__init__()

        self.outputWeights = torch.nn.Parameter(torch.Tensor([1, 1]))
        torch.nn.init.uniform_(self.outputWeights, 35, 40)

    def forward(self, X):
        return ((X + 1) / 2) * self.outputWeights
```

Figure 9: Output Layer Class

```
class CircuitBuilder:
    @staticmethod
    def createCircuit(n_qbits=4, n_reps=5, noisy=True):

        inputsParam, weightParam, quantumCircuitRaw = CircuitBuilder.buildQuantumCircuit(n_reps, n_qbits)

        if noisy:
            print("Noisy Backend, IBM Oslo")
            provider = qk.IBMQ.load_account()
            backend = provider.get_backend('ibm_oslo')

            backend_sim = AerSimulator.from_backend(backend)
            qi = QuantumInstance(backend_sim)
        else:
            print("NOT-Noisy Backend, Statevector Simulator")
            qi = QuantumInstance(qk.Aer.get_backend('statevector_simulator'))

        qnn = CircuitQNN(quantumCircuitRaw, input_params=inputsParam, weight_params=weightParam,
                         quantum_instance=qi)
        initial_weights = (2 * np.random.rand(qnn.num_weights) - 1)
        return TorchConnector(qnn, initial_weights)
```

Figure 10: Circuit Builder main-static-function

```
@staticmethod
def buildQuantumCircuit(n_reps, n_qbits):
    qr = QuantumRegister(n_qbits, 'qr')

    qc = QuantumCircuit(qr)

    parameters = qk.circuit.ParameterVector('theta', 2 * n_qbits * n_reps)
    inputParameters = qk.circuit.ParameterVector('x', n_qbits)
    for i in range(n_reps):

        qc.barrier()

        for j in range(n_qbits):
            qc.rx(inputParameters[j], j)
            qc.ry(parameters[j + (2 * i * n_qbits)], j)
            qc.rz(parameters[n_qbits + j + (2 * i * n_qbits)], j)

        qc.cz(qr[0], qr[3])
        qc.cz(qr[0], qr[1])
        qc.cz(qr[1], qr[2])
        qc.cz(qr[2], qr[3])

        qc.barrier()

    qc.draw(output='mpl', style={'backgroundcolor': '#EEEEEE'})
    plt.show()
    return inputParameters, parameters, qc
```

Figure 11: Circuit Builder subroutine-static-function

```python
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward', 'done'))


class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def getBatchSample(self, batch_size):
        transitions = random.sample(self.memory, batch_size)
        batch = Transition(*zip(*transitions))

        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)
        dones_batch = torch.cat(batch.done)
        next_states = torch.cat(batch.next_state)

        return state_batch, action_batch, next_states, reward_batch, dones_batch

    def __len__(self):
        return len(self.memory)
```

Figure 12: Replay Memory Class

Appendix: Stretched Exponential Decay function for Epsilon Greedy Algorithm

We were looking for a decay function that can provide the RL agent following characteristics:

- More dwell time for exploration at initial part of episodes
- More exploitation with random exploration at end of episodes (quasi-deterministic policy)
- Smooth gradient while switching from exploration to exploitation

Searching online we found this **Stretched Exponential Decay Scheduling**, proposed by Subbiah Natarajan on **Medium**

```
EPISODES = 0

import math
import numpy as np

A = 0.5  # decides where we would like the agent to spend more time
B = 0.1  # decides the slope of transition Exploration to Exploitation
C = 0.1  # controls the steepness of left and right tail of the graph


def stretched_epsilon_decay(episod):
    standardized_time = (episod - A * EPISODES) / (B * EPISODES)
    cosh = np.cosh(math.exp(-standardized_time))
    epsilon = 1.1 - (1 / cosh + (episod * C / EPISODES))
    return epsilon
```

Figure 13: Stretched Exponential Decay

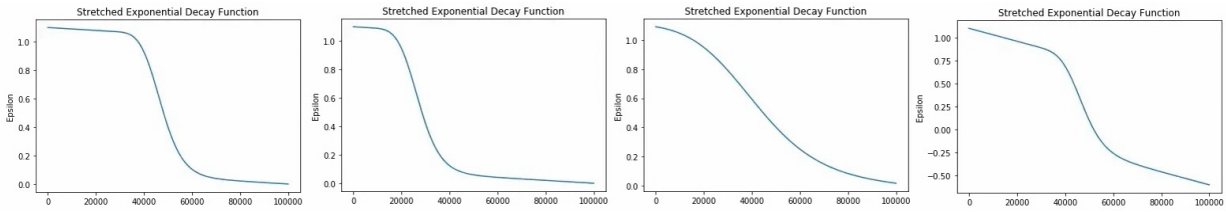


Figure 14: Hyperparameters A, B, C settings comparison

From left to right, in Figure 14:

1. A=0.5, B=0.1, C=0.1 (optimal choices)
2. A=0.3, the left tail is shortened
3. B=0.3, the transition portion has gradient = -45 degree
4. C=0.7, the left and right tail tends to get steeper

Appendix: Model hyperparameters

	CartPole Optimal
qubits	4
layers	5
discount	0.99
train w-in	yes
train w-out	yes
batch size	16
size replay memory	2000
lr	0.001
epsilon decay	Stretched-Exponential
	A = 0.5
	B = 0.1
	C = 0.1

Table 1: Hyperparameter settings of Optimal PQC

Appendix: Classic Approach's Results

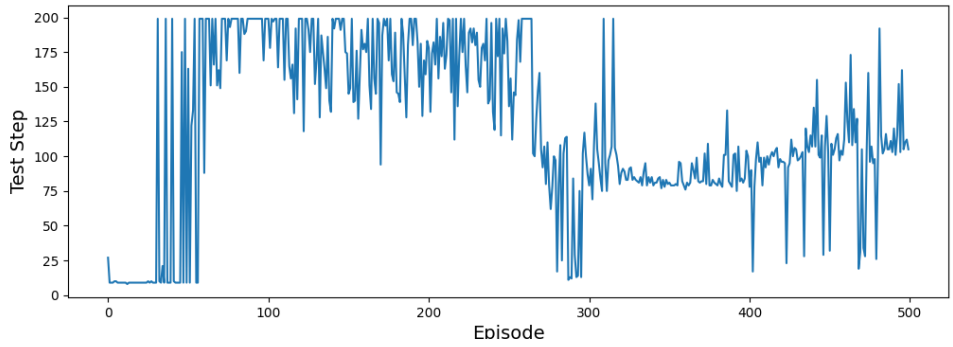


Figure 15: Classic Approach's Steps with Greedy-Policy

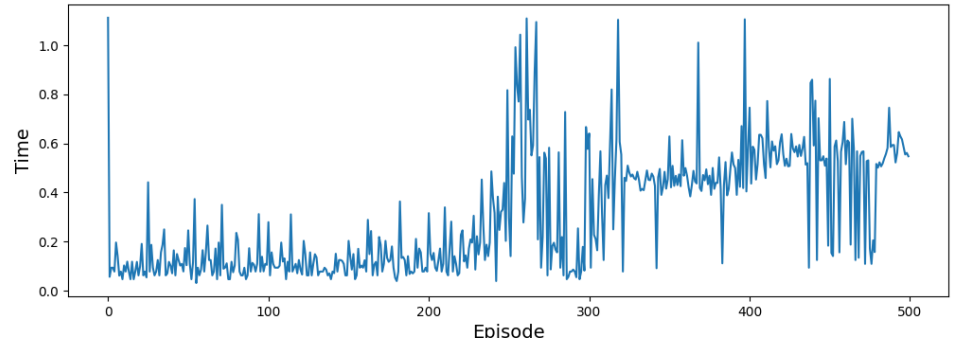


Figure 16: Classic Approach's Epochs Duration Time