UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA
DEPARTAMENTO DE MATEMÁTICA APLICADA

ALAN ALBERT PIOVESANA

An Introduction to Quantum Reinforcement Learning

Campinas
Jul/2022

ALAN ALBERT PIOVESANA

An Introduction to Quantum Reinforcement Learning

Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos para obtenção de créditos na disciplina Projeto Supervisionado, sob a orientação do(a) Prof. Dr. Marcelo Terra Cunha.

Resumo

Este texto é um estudo baseado em [1, 2], focado na aplicação de algoritmos quânticos variacionais (VQAs) em problemas de aprendizado por reforço (*reinforcement learning*). A estratégia utilizada foi de iniciar pelo estudo de redes neurais clássicas em um problema de classificação binária, passando, em seguida, para a versão quântica do mesmo, utilizando VQAs. Em seguida, estudou-se o problema de *reinforcement learning* clássico, e, por fim, a utilização de VQAs para aproximar a política ótima em um algoritmo de deep Q-learning, utilizado para resolver o problema canônico *Cart Pole* [3]. Os autores do artigo [1] criaram um algoritmo que usa cerca de um terço dos parâmetros da sua versão clássica para uma mesma pontuação acumulada e tempo de treino. Ao rodarmos o código disponível 10 vezes, extraindo o comportamento médio, a evolução no treino do agente também foi perceptível.

Abstract

This text is a study based on [1, 2], focused on applying variational quantum algorithms (VQAs) to reinforcement learning (RL) problems. The strategy used for this study was starting by understanding classical neural networks in a binary classification problem, moving, afterward, to the quantum version of the same problem, which uses VQAs instead. After that, we studied classical RL algorithms, and, finally, the application of VQAs to approximate the optimal policy on a deep Q-learning problem, whose solution was used to solve the canonical problem *Cart Pole* [3]. The authors of the article [1] created an algorithm that requires about one-third of the parameters of the classical version for a given score and training episode. By running the available code 10 times, and extracting the average behavior, the evolution of the agent's training was also clear to us.

Contents

1 Introduction

1.1 Quantum computing

Quantum computing has been a very active field in the past few decades. With some promising results of quantum processors achieving the so-called *quantum supremacy* in specific problems that classical computers would take much longer to solve, such as Google's Sycamore processor's solution to a benchmark problem [4], this field has been quickly evolving and gaining attention from the scientific community and industry.

A quantum computer is a machine that can leverage the power of collective properties of quantum mechanics, such as interference, superposition, and entanglement, to perform calculations [5]. The main difference from classical computers is that quantum computers do not use individual bits for encoding information. Instead, they can create superpositions of several states, and manipulate the probability distribution so that the readout - which is made by performing averages of the quantum states - yields larger probabilities for states that lead to the correct answers and lower ones for those leading to incorrect answers.

1.1.1 Variational quantum algorithms

Variational quantum algorithms (VQAs) are a class of quantum algorithms that uses a quantum circuit whose operations depend on parameters that can be classically tuned to achieve specific tasks. The gates on a quantum variational circuit are tunable utilizing a set of parameters Θ. VQAs have several applications, especially in artificial intelligence, in which the tunable circuits can be used to statistically learn the best configuration so that the circuit can solve tasks for itself. The VQA then acts similarly to a neural network, and one can use this configuration for the applications we explain in the next section.

1.2 Machine learning

Machine learning techniques are statistical-computational methods used to make predictions or fit data in a way that is not directly dependent on external influence.

Once the objective function to be optimized is defined, the algorithm itself learns the best solution to the problem.

This is a field that has received much attention in the last years since it has several proven useful applications both in science and industry.

Machine learning techniques fall into three categories: supervised learning, unsupervised learning, and reinforcement learning.

1.2.1 Supervised versus unsupervised learning

Supervised learning is a class of machine learning methods that aims at learning a function that solves specific problems, given a so-called *training set*, which is a set of data points $\{(x_i, y(x_i))\}$ with input x_i and expected output $y_i = y(x_i)$, providing a template for validating the method. Once the algorithm has achieved satisfactory performance in the training data, it can be used to create predictions for data that was unseen before, thus generalizing what it has learned.

Unsupervised learning, on the other hand, does not use a training set, and it is usually focused on recognizing patterns in unclassified data, for instance, solving grouping or classification tasks in which data points that are more similar to each other, according to some metric, fall in the same group, while points which are dissimilar fall into different ones. Unsupervised learning methods are trained for minimizing a cost function, such as the sum of the Euclidean distances of points in the same group, in the given example.

1.2.2 Reinforcement learning: one step forward

Along with the two other types, there is a third one, focused not only on approximating a function that fits the data well or recognizing unclassified patterns but also on making decisions based on a policy that states what action to take at each given step. Problems of this category are labeled *reinforcement learning* (RL). In this category, one is faced with the challenge of making an autonomous agent navigate through an unknown environment, and generate an optimal policy, which consists of a set of rules that shape the decision-making of the agent on how to react to the barriers and obstacles. This policy is what allows the agent to build its navigability strategy in the environment. It is a conditional probability distribution $\pi(a|s)$, which maps each state s to an action a

that the agent can take.

Apart from the addition of the decision-making process, one key difference between reinforcement learning problems and both supervised and unsupervised learning is that in the former, the data based on which the algorithm will learn is not readily available - instead, it is dynamically acquired while it moves through the environment.

Some reinforcement learning algorithms find the optimal policy by maximizing the expectation of return the agent has when choosing each state. This expectation is called *value function*, or *Q-function*[*], so that algorithms of this class are called *Q-learning*. When, furthermore, the policy learned is not the optimal one, but rather an approximation of it, relying on neural networks to make this approximation, the method is called *Deep Q-Learning*, and it is the one we will use to adapt to the quantum realm in this text.

It is important to notice that the goal of reinforcement learning problems is to not only build special-purpose learning algorithms, that can only learn how to master one environment - that is, an agent that can only play well a single game - but can also perform satisfactorily in several different environments.

When, apart from being agnostic to the choice of environment, the agent also does not need to be retrained when switching environments, we have what is called *transfer learning* [6], which are state-of-the-art reinforcement learning algorithms that use parts of the training focused on one task to solve other tasks.

1.3 Quantum reinforcement learning

When we bring together quantum computing and reinforcement learning, we get what is called *quantum reinforcement learning*. This is an interface field, which has only recently taken shape, with most papers being published in the last five years.

There are a few ways to turn classical reinforcement learning problems into quantum ones. Most of them result in a hybrid classical-quantum scheme, in which part of the processing is made on a quantum machine (or in a classical simulator of a quantum machine), and the remaining steps - usually the optimization parts - are done classically.

One example of a quantum reinforcement learning problem is the so-called *Quantum Maze* [7] - which uses a classical RL technique applied to a quantum problem:

[*]Be aware that "Q" here stands for "quality", not "quantum" - originally, this is a classical algorithm

a quantum particle in a maze that can change its pathways randomly. Another example, which is the one we will be focusing on in this paper, has both the agent and the environment behaving classically, but the neural network that is used to get an approximation for the optimal policy is replaced by a variational quantum circuit [1], defining what we call a *quantum deep Q-learning* algorithm.

1.4 Content organization

As mentioned, this text aims to study the theory and reproduce the implementation of a quantum deep Q-learning algorithm to solve a problem in which both the agent and environment are classical, but using a variational quantum circuit instead of a neural network. For this reason, we decided to use the following strategy:

- We will first briefly introduce classical neural networks, focusing on the feed-forward topology for classification tasks [8];
- We will then bring that same problem into the quantum realm, studying the mathematics of VQAs applied to quantum binary classifiers, following [2];
- In this same section, we will study a Python implementation of VQA solving the bit string parity problem, according to [9];
- Next, we will understand the basics of the classical reinforcement learning problem, focusing on the deep Q-learning method, following [10];
- Once having both a good grasp on VQAs and classical deep Q-networks, we will address the quantum deep Q-learning problem, using what was studied in the previous sections to build a VQA that learns a policy capable of solving a few simple problems on reinforcement learning, following the methodology described on [1].

The central idea of this text is therefore to introduce the student to quantum reinforcement learning through a literature review, starting from classical machine learning techniques, all the way up to the framework described in [1]. In this path, we will also reproduce a few computational implementations available online in open-source Python libraries, such as PennyLane [11] and Cirq [12].

2 Classical neural networks

An "artificial neural network" (ANN) - or, simply "neural network" (NN) - is a statistical method inspired by brain networks [13], which is used in machine learning to approximate a function f, which is unknown, but whose main properties can be inferred from the problem at hand.

This is usually done through supervised learning, with available training sets containing input vectors together with the expected output, and the goal of the NN is to learn a function that best fits the training data, and which can also be used in other, uncategorized data.

Below, there is a schematic of a classical artificial neural network, of a topology called *feed-forward*:

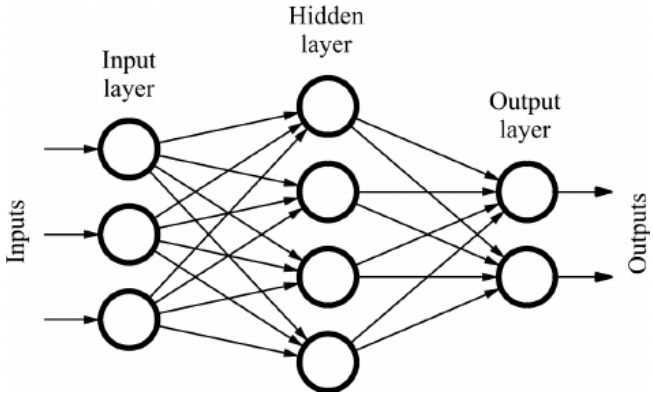


Figure 1: Schematic of a feed-forward classical neural network [8].

The way a neural network learns the function that best fits the training data is by composing a set of linear operations and a nonlinear function, called the *activation function*. Each of these combined operations - applying the linear map and the nonlinear activation - defines what is called a *neuron*, which is usually a real number resulting from the operations. A set of neurons acting together is called a *layer*, and it represents one iteration of a chain of composite functions that approximate f. The function resulting from the composition of several layers is:

$$F(x) = f_N(f_{N-1}(\dots f_2(f_1(x))\dots)) \tag{1}$$

Here, each f_i represents the result of operation of layer i. Each neuron value is generated by a linear function, followed by a nonlinear activation. The application of a layer of neurons on an input then reads:

$$z_i = W_i\, x_{i-1} + b_i \tag{2}$$

$$x_i = \Sigma(z_i), \tag{3}$$

where z_i represents the linear output of layer i, using x_{i-1} as input (the result of the function in the previous layer), together with W_i (the weights matrix), and b_i (the bias vector) - both of which will be discussed next - while x_i is the final result of the operations of layer i, including the nonlinear activation function Σ. In this notation, Σ is the scalar field that applies σ to each of the neurons in the layer. To understand how an NN works, let us look at each of these terms in detail, applied to the simplest possible topology of neural networks: feed-forward NNs.

2.1 Feed-forward neural networks

This specific topology has a tree structure (no self-loops and no backward flow in the calculations) and is composed of three types of layers: one input, or feature layer, a set of one or more hidden layers, and one output layer.

This topology is used in a process that has two parts: forward and backward passes. In the former, the network receives as input one of the vectors from the training set, x_i, and outputs a value that depends on the network's weights and biases. The process then moves to the latter part, which consists in comparing the output value, $\hat{y}_i$, with the expected one, y_i, and adjusting the weights and biases to reduce the difference between them. This alternation between different passes is repeated up until a satisfactory result comes out of the network.

2.1.1 Forward pass

The forward pass, as described before, receives as input the training vectors, which are, layer-by-layer, processed by the NN to provide an output.

The first layer is used to encode data in a way that is readable by the network.

This can be done using several different methods, depending on the problem at hand. For example, in networks used to classify images, each pixel of the image can be stored as an entry for the input vector of the network, so that in an N by M pixels image, one would have an NM by 1 input vector [14].

The second is a set of one or more layers, which are responsible for the "learning" part. They take the encoded data from the previous layer and apply transformations to it until it generates an output. To move from one layer to the next one, vector calculations of the form (2) are carried out.

To decide which of the neurons of the previous layer have a larger impact on the current one, we define the weights matrix W_L, with entry w_{ij} giving the importance of neuron j in the $(L-1)$-th layer to the value of neuron i in the L-th layer. The bias term allows for more flexibility in the approximation process. On top of the weights and biases, which act as an affine transformation on x_{i-1}, we apply the nonlinear function σ.

The function σ, inspired by the activation of neurons in the human brain [15], is used to allow for nonlinear maps in the input x_i, so the neural network can go beyond a linear classifier technique. Function σ can be of several different shapes. In the early papers of ANNs, σ was taken to be a sigmoid function, which provides output values between 0 and 1 that can be taken as a proxy for probability. This function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

However, in more recent applications, this function is usually replaced by others, due to several problems associated with it, such as training performance and the vanishing gradient problem [16]. One of the most used nonlinear activation functions in current applications [15] is the so-called ReLU (an acronym for Rectified Linear Unit), which is defined as:

$$\sigma(x) = max(x, 0) \tag{5}$$

Apart from those two examples, several other activation functions are currently in use, and they all have the same goal: adding flexibility to the method to approximate nonlinear functions.

Finally, after the operations are carried out by the hidden layers, the output layer is responsible for retrieving the values produced by the last hidden layer and generating an answer to the problem. This answer can come in the form of a vector, but for simple problems such as the binary classification [17], the output is usually a number (0 vs 1, for example). This value that the network outputs can then be compared with the expected answer for that input, stored in the training set. If the output value is too distant from the expected one, the weights and biases need to be tuned so the network outputs a value that is closer to the expected. This mechanism is based on the backward pass of the method, which we discuss next.

2.1.2 Backward pass

This part moves in the opposite direction as the forward pass. After having run the network to output a value $\hat{y}_i$ given one of the inputs x_i in the training set, that value is compared to the corresponding target value in the training set, y_i. Sweeping over all training points and taking the average difference from output to target, we can define a cost to be minimized, usually called *loss function*. The loss function can take up several forms, one of which is the *Mean Squared Error* (MSE) [18]:

$$\mathcal{L}(\hat{y}_i, y_i) = \frac{1}{|\mathcal{T}|}\sqrt{\sum_i (\hat{y}_i - y_i)^2} \quad , \tag{6}$$

where the i index sweeps over all the data points in the training set $\mathcal{T}$, of cardinality $|\mathcal{T}|$, which we assume here to be finite.

In order to reach optimal values for $\mathcal{L}$, one varies the weights and biases of the network. But doing so in a random fashion would be computationally slow, so *gradient-descent* methods are employed. To do that, we first build a vector that contains all weights and biases of the network, layer-by-layer (each layer tagged by the super-index (L)):

$$v^T = \left[w_{11}^{(1)} \ \ldots \ w_{1n}^{(1)} \ w_{21}^{(1)} \ \ldots \ w_{2n}^{(1)} \ \ldots \ w_{mn}^{(1)} \ b_1^{(1)} \ \ldots \ b_n^{(1)} \ \ldots \ w_{11}^{(2)} \ \ldots \right] \tag{7}$$

Then, we apply the gradient-descent (GD) method to this vector. There are several variations of the classical GD method, both using momentum and stochastic

batches [19]. The classical update of the method is given by:

$$v_k = v_{k-1} - \gamma \nabla \mathcal{L}(v_{k-1}) \ , \tag{8}$$

where k indexes the iteration in which we are at.

One important feature is that, since we have an explicit rule to go from one layer to the next one, we can adjust weights in the initial layers by employing the derivative chain rule, to compute how the loss varies with respect to every term in v.

Once the weights and biases are adjusted to get closer to an optimal point, evaluating all input points of the training set, the forward pass is repeated, and a new loss value is computed. We repeat this process until we are ϵ-close to the minimum (that is, $\mathcal{L} < \epsilon$) or by limiting the number of times we evaluate the whole training set. Each of these full passes is called an *epoch*.

This way, a neural network takes as input a training dataset, and tunes its own inner parameters (weights and biases) to the best possible values, generating an output function that best solves the problem. In the case of a ReLU-activated network, that function is piece-wise linear.

This network can then be used to infer values for data that is not in the training set, and the accuracy with which it does so can vary depending on the size and variety of the training set, and also on how many epochs the algorithm runs through.

3 Variational quantum algorithms

Similarly to classical networks studied in the previous section, one can use a quantum circuit with tunable parameters to approximate a function that solves a particular problem. Let us now deep dive into how this is done.

3.1 Quantum circuits

To understand what a quantum circuit is, we first need to introduce the idea of a *qubit*. A qubit is the unit of information on quantum computing, and, unlike the classical bit, which can only be at state 0 *or* 1 at a given time, a qubit can be in a superposition of both states, so it is represented, using Dirac's notation [20], by:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad , \tag{9}$$

where $|0\rangle$ and $|1\rangle$ are the two states in the so-called *computational basis* and α and β are two complex numbers. These numbers have a probabilistic interpretation: the probability that the qubit's wavefunction collapses to state $|0\rangle$ after a measurement is $P(|0\rangle) = |\alpha|^2$. Similarly, the probability of it collapsing to state $|1\rangle$ is $P(|1\rangle) = |\beta|^2$ [20]. While no measurement is performed, the qubit is in neither of the basis states, but in a superposition of both.

It can be shown that, apart from a global phase factor, any qubit can be represented as a point on the surface of a sphere, called *Bloch sphere* [21, 22] as follows:

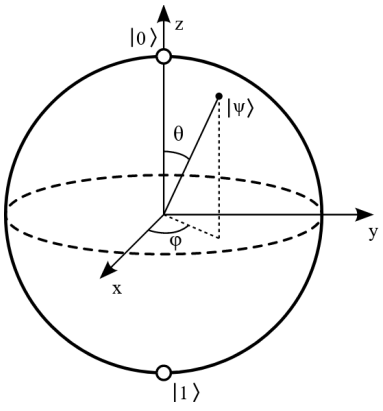


Figure 2: Representation of a qubit $|\psi\rangle$ as a vector on the Bloch sphere.

Another important definition is that of a quantum operator. Quantum *operators* are linear maps from a Hilbert space to itself [20]. Operators can be represented as matrices on the computational basis, so the action of an operator on a qubit becomes a matrix product. One important feature of quantum operators is that they can only be represented by unitary matrices, which means that, given a Hilbert space $\mathcal{H}$, an operator U is a linear map $\mathcal{H} \to \mathcal{H}$ with the following property:

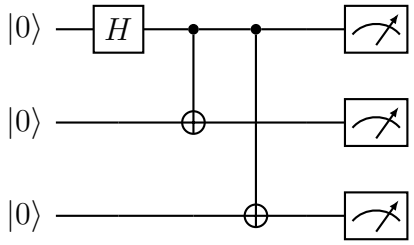
$$U\,U^\dagger = U^\dagger\,U = I \ ,$$

where $U^\dagger$ is the Hermitian conjugate of U (i.e. the complex conjugate of U transposed).

When we compose one or more operators, which we will call *gates*, acting on one or more qubits, we have a *quantum circuit*.

Therefore, a quantum circuit is a composition of quantum gates applied to a tensor product of input qubits. These gates are operators that can act on one, or more qubits at a time and the composition results in a linear transformation. Since quantum operators are unitary, the resulting composition is also unitary.

We represent quantum circuits by diagrams such as the following:



In this circuit example - which is called GHZ state, a 3-qubit implementation of a Bell-state [23] - one starts with the tensor product $|000\rangle$, represented by the 3 wires, each containing a $|0\rangle$ ket. The first ket is acted upon by a single-qubit operator called Hadamard gate, and the two other ones are both acted by a gate called Controlled-NOT (CNOT), which is said to be *controlled* by the first qubit. After the two controlled operations, all three qubits are measured, so it can be shown that the net effect of this circuit is the map:

$$|000\rangle \mapsto \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

The two gates mentioned here - Hadamard and CNOT - are two important quantum operators. The Hadamard gate is used in several quantum circuits, such as the single-qubit interference circuit [24]. This gate can be represented by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{10}$$

This gate operates on one qubit only, as follows:

$$|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$|1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

On the other hand, the CNOT gate acts on pairs of qubits, the first being called the *control* qubit, and the second, the *target* one. This gate flips the second qubit if the first one is on state $|1\rangle$, and applies the identity matrix otherwise. The matrix representation of the CNOT is, therefore:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{11}$$

The circuit discussed so far is an example illustration of the way we build quantum circuit diagrams: wires represent the different qubits, boxes with letters or symbols are the unitaries that operate on them, and meter boxes represent quantum measurements, carried out by taking the average of the output.

The circuit topology varies according to the problem, but it is noticeable the similarity between quantum circuits and neural networks, in the sense that both are a set of operations acting, in a network-distributed fashion, on inputs, and yielding measurable outputs. On the other hand, neural networks, as seen in the previous section, can approximate almost any function using the nonlinear activation over the neurons, while quantum circuits are restricted to operating linearly on inputs unless specific techniques

to introduce nonlinearities are used [25]. Another important difference between NNs and VQCs is that the latter leverage the properties of interference and superposition throughout the circuit, and the manipulation is made only to the probability amplitudes, not to the numbers (neurons) themselves, so one can only access the result of the calculations at the output measurement.

3.2 Variational quantum circuits

When one or more of the quantum gates in a quantum circuit depend on a set of parameters $\{\theta_i\}$, which can be adjusted to enable the circuit to solve certain problems, we have what is called a *Variational Quantum Circuit* (VQC), or, equivalently, a *Parametrized Quantum Circuit* (PQC). These circuits are particularly interesting because they share similarities with classical neural networks, as we will see in the example below. Algorithms that run on variational quantum circuits are called *Variational Quantum Algorithms* (VQAs).

3.2.1 Quantum binary classification

To show how to use a quantum circuit similarly to a neural network, we present a method for solving the benchmark *binary classification* problem, which is well-known in classical supervised learning [17], and is here adapted to run in quantum circuits.

This problem entails finding a function f that correctly yields 0 or 1 depending on the input, which is a binary string. To find f, one first needs to ensure that it is possible to build such a function for a generic binary input, which is a non-trivial question, given that quantum circuits can only apply linear operations.

3.2.2 Boolean functions in quantum circuits

To show that this problem is generally solvable in a quantum machine, we refer to the proof in [2] that a method that outputs any label function $l : \{-1, 1\}^N \to \{-1, 1\}$ can be built using the following unitary, representing the whole quantum circuit:

$$U_l = \exp\left[i\frac{\pi}{4}l(Z)X_{n+1}\right] \implies U_l\,|z, z_{n+1}\rangle = \exp\left[i\frac{\pi}{4}l(z)X_{n+1}\right]|z, z_{n+1}\rangle \tag{12}$$

Here z is the input bit string, which, in the quantum circuit, corresponds to a tensor product of inputs, and z_{n+1} is the auxiliary qubit where the measurement happens. In this notation, $l(Z)$ is the operator that is diagonal when acting on a tensor product of qubits associated with the computational basis (i.e. $l(Z)\left|z, z_{n+1}\right\rangle = l(z)\left|z, z_{n+1}\right\rangle$).

To illustrate the operation of $l(Z)$, take as an example a circuit with input bit strings of size 1 (so we need 2 qubits to produce $|z, z_{n+1}\rangle$). This means that, by varying both the input string and the output qubit, an arbitrary state $|\psi\rangle$ will be of the form:

$$|\psi\rangle = c_1 |00\rangle + c_2 |01\rangle + c_3 |10\rangle + c_4 |11\rangle \tag{13}$$

In this case, and considering that a $+1$ in the bit string maps to ket $|0\rangle$ and -1 maps to $|1\rangle$ (the other opposite mapping would work just as well), the representation of $l(Z)$ in the computational basis is:

$$\begin{aligned} l(Z)|00\rangle &= l(+1)|00\rangle \qquad & l(Z)|01\rangle &= l(+1)|01\rangle \\ l(Z)|10\rangle &= l(-1)|10\rangle \qquad & l(Z)|11\rangle &= l(-1)|11\rangle \end{aligned}$$

which, in matrix notation becomes:

$$l(Z)|\psi\rangle = \begin{bmatrix} l(+1) & 0 & 0 & 0 \\ 0 & l(+1) & 0 & 0 \\ 0 & 0 & l(-1) & 0 \\ 0 & 0 & 0 & l(-1) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} l(+1)c_1 \\ l(+1)c_2 \\ l(-1)c_3 \\ l(-1)c_4 \end{bmatrix} \tag{14}$$

If we restrict our analysis only to the basis states - which is the case in our problem, since qubits are input independently for each wire, so the entire state is simply a tensor product of them - the matrix representation of the operator $l(Z)$ in the computational basis can only be nonzero for one of the entries in the diagonal, and zero for the others.

Now, given that there exists such a unitary operator representing the whole circuit, it remains to understand how to decompose it into 2-qubit gates, to build a circuit topology that is feasible in real-world applications, since the larger the number of qubits

operated at the same time by a single gate, the more prone to errors the circuit is. This is also important because a bad choice of topology (or *ansatz*) can limit how well our circuit can learn [2].

To achieve that, we first switch the z variable, $z \in \{-1, 1\}$ to ordinary Boolean variables, with the transformation $b_i = \frac{1}{2}(1 - z_i)$, where each z_i is a bit in the i-th position on the binary string z. Now b can be decomposed using the Reed-Muller (RM) expansion, as proven in the appendix A.

With this change of variables, the unitary of the whole circuit, which depends on the label function l, can be re-written as:

$$U_l \left|z, z_{n+1}\right\rangle = \exp\left[i\frac{\pi}{4}(1 - 2B)X_{n+1}\right]\left|z, z_{n+1}\right\rangle = \exp\left[i\frac{\pi}{4}X_{n+1}\right]\exp\left[-i\frac{\pi}{2}BX_{n+1}\right]\left|z, z_{n+1}\right\rangle \tag{15}$$

Here, B is the matrix that is diagonal in the computational basis and that corresponds to b, similarly to the matrix representation of the operator $l(Z)$ we had before the change of variables.

Now, given that we have an explicit function that solves the problem using 2-qubit unitaries, we can proceed to solve the parity classification problem.

3.2.3 Subset parity classification

In our example, we intend to build a quantum circuit whose task is to learn a Boolean function $f : \{0,1\}^N \to \{0,1\}$, in order to classify the parity of a binary string z. Function f should be able to have the following property:

$$f(z) = \begin{cases} 0, & \text{if there is an even number of 1s in } z \\ 1, & \text{otherwise} \end{cases}$$

In the parity problem, we will be focusing only on the linear part of the RM expansion [2], because one can retrieve the parity of a subset $\mathcal{S}$ with the formula:

$$P_{\mathcal{S}} = \bigoplus_j a_j b_j \tag{16}$$

Recalling that the coefficients a_j correspond to Boolean differences, which are equal to one only if the function changes when changing the input, the a_js are all zero for bits j that are not contained in set $\mathcal{S}$. Also recall that the binary addition $\oplus$ is 0 when the operands are equal and 1 otherwise, so we can see that $P_{\mathcal{S}} = 0$ when the number of zeros and ones is even, and $P_{\mathcal{S}} = 1$ otherwise, as required in the problem statement.

Now, using $P_{\mathcal{S}}$ as the RM expansion for the function we want our circuit to learn, the unitary in (15) becomes:

$$U_{P_{\mathcal{S}}} = \exp\left[i\frac{\pi}{4}X_{n+1}\right] \exp\left[-i\frac{\pi}{2}\bigoplus_j a_j B_j X_{n+1}\right] \tag{17}$$

With this formula at hand, the hardware implementation becomes, simply, a series of bit flips acting on pairs of qubits, one being the control, and the other the target.

3.2.4 Circuit ansatz

Since we derived the main properties our circuit needs to have to learn the parity of a bit string, which is equivalent to composing several controlled bit flips, we can use the following circuit topology, retrieved from [26]:

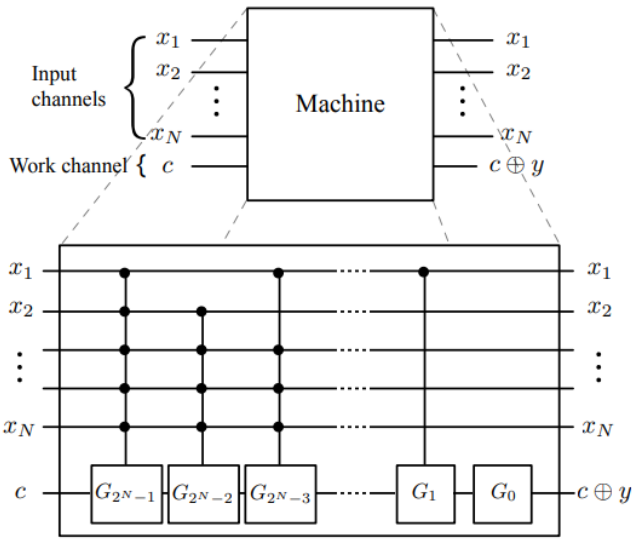


Figure 3: Schematic of one possible topology for this problem [26].

Note that there are 2^N gates to cover all possible input bit string combinations, and the G's are CNOT gates conditioned on zero, one or more of the input qubits. To see the correspondence between the gates and the RM coefficients that generate each possible Boolean function, consider the following table image, also taken from [26]:

Boolean function	a_0	a_1	G_0	G_1
$f_1 : x \mapsto 0$	0	0	Identity	Identity
$f_2 : x \mapsto 1$	1	0	NOT	Identity
$f_3 : x \mapsto x$	0	1	Identity	NOT
$f_4 : x \mapsto x \oplus 1$	1	1	NOT	NOT

Figure 4: Example for 2-sized input bit strings ($N = 2$), showing that all Reed-Muller coefficients can be represented by a unique configuration of controlled-NOT gates [26].

3.2.5 Learning

The algorithm described so far is hybrid (quantum and classical), in the sense that it is only the circuit, which replaces the neural network, that is quantum. All the optimization and updates are carried out classically.

After running the quantum circuit for all training inputs in the set $\{z\}$, we measure the output qubit, $|z_{n+1}\rangle$, estimating the average value of a Pauli-Y operator, obtaining:

$$\langle z, z_{n+1}| \, U^\dagger(\vec{\theta}) \, Y_{n+1} \, U(\vec{\theta}) \, |z, z_{n+1}\rangle \tag{18}$$

With this average output, for each fixed z, we can then carry on classically in the backward pass, by first defining a loss function:

$$loss(\vec{\theta}, z) = 1 - l(z) \, \langle z, z_{n+1}| \, U^\dagger(\vec{\theta}) \, Y_{n+1} \, U(\vec{\theta}) \, |z, z_{n+1}\rangle \tag{19}$$

This function is zero if the output of the circuit is equal to the expected label for that input, $l(z)$, and it is larger than zero in the case of opposite labels. Here we use the variable before the Boolean transformation (that is, $z \in \{-1, 1\}^N$).

3.2.6 Computational implementation

Having studied the theory behind one of the simplest quantum variational classifiers in the literature, in this section we follow the implementation suggested in [9], using a classical simulator for the quantum operations, and the Nesterov momentum algorithm for the gradient descent [19]. Furthermore, since the parity classification problem can be solved by using only the first order of the Reed-Muller expansion, the first N unitaries of the ansatz of figure 3 are sufficient, so the circuit only connects first-neighboring pairs of wires, in a ring topology.

By running the algorithm, which can be found in the GitHub repository created for this project [27], with a 4-qubit input circuit, we get the following charts:

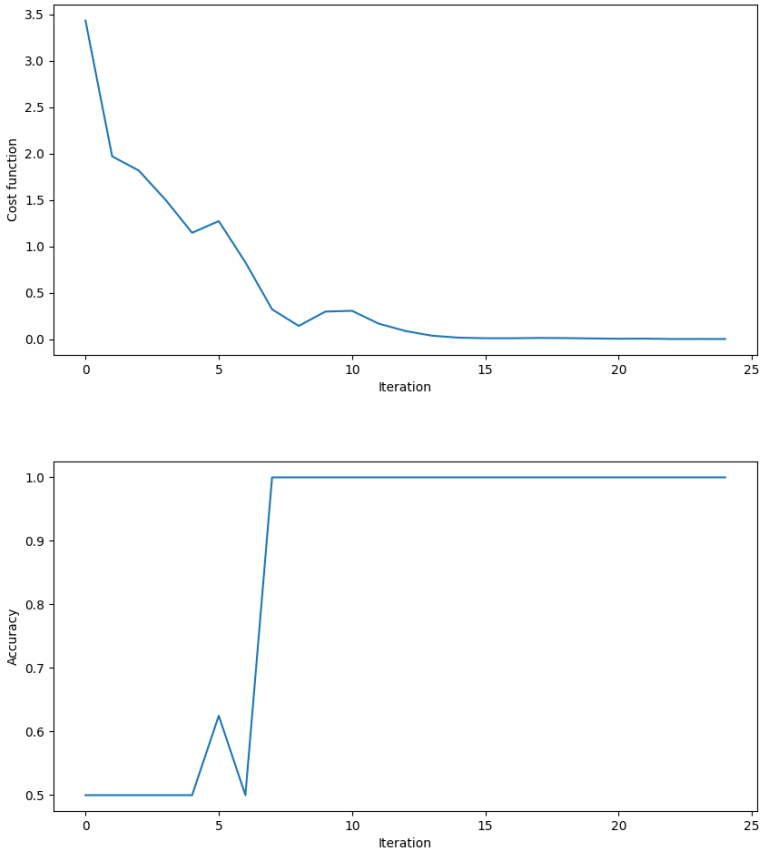


Figure 5: Graphs of the cost function and accuracy value by iteration.

As can be seen from the charts, in the case of short bit strings (4 qubits), the learning process is fast, reaching 100% accuracy after 8 steps, and the cost function also declines fast in a training set containing 16-bit strings. One of the reasons for choosing small bit strings is due to the limits imposed by the classical implementation of a quantum simulator, making the computing time scale exponentially with bit string length.

4 Classical reinforcement learning

4.1 Introduction

Reinforcement learning problems aim at finding the optimal policy for an agent to navigate through an environment. Assuming a discrete evolution, at each time step t, the agent is at a state $s_t \in \mathcal{S}$. The set $\mathcal{S}$ represents all possible configurations for the variables that define the relationship of the agent with the environment (e.g. position, velocity, etc), and the agent can take actions $a \in \mathcal{A}$, driving it from s_t to s_{t+1}. An *optimal policy* is a probability distribution $\pi_*(a|s_t)$ that maps state s_t to the action a that brings the best results by some metric. In this text, we will focus on Q-learning, which uses a value function to define this metric.

4.2 Q-learning

As mentioned, this type of reinforcement learning problem focuses on optimizing a so-called *value function* to get as close as possible to solving an optimality equation, called *Bellman equation*. For us to understand how this process works, we first need to make some assumptions about the nature of the system.

4.2.1 Environment returns

The first thing to notice is that, in RL problems, the agent's objective is not to maximize individual rewards coming from each possible action, but an overall return from all future actions. This return is a sum of all future individual rewards. If one is dealing with a finite process, in which there are *episodes*, after which the dynamics restart, the reward at time t can be defined as simply:

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T \ , \tag{20}$$

where T denotes the time step associated with the end of the episode. Alternatively, if there are no clear definition of episodes, and the environment interaction can continue

indefinitely, we have to use a *discounted* version of this definition, such as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad , \tag{21}$$

where γ, $0 \leq \gamma < 1$, is a discounting factor, which conveys the idea that future rewards impact present decisions less than current ones. Both of the previous definitions can be merged into one:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{22}$$

Here T can be taken as $T \to \infty$ to account for the continued interaction and γ can be set to 1 for the finite one, but not both conditions at the same time (otherwise the series may diverge depending on the rewards).

One point to notice is that, although most of the reinforcement learning process focuses on maximizing the return expectation, through a strategy called *exploitation*, it is sometimes useful to choose random actions in some of the steps, to make sure not to get stuck into local maxima. This is usually done through an *ϵ-greedy* algorithm, which defines a parameter ϵ, $0 \leq \epsilon \leq 1$, such that the agent takes the action that maximizes the expectation of returns with probability $(1 - \epsilon)$, but with probability ϵ, it picks a random action, in a strategy called *exploration* of the environment.

4.2.2 Markov decision processes

In the previous section, we have defined the returns function at each time step, and now, to construct an objective function and for the RL problem to be reduced to solving a set of equations, we need an extra ingredient, called the *Markov hypothesis*. To understand it, consider an agent at time t taking an action that leads, at time $t+1$, to a response from the environment. This response, in the most general case, may depend on all past actions the agent has taken and states it has been at. For Markovian systems, on the other hand, the only variables that may influence how the environment will behave at $t+1$ are the *current* action and state. This means that, under this hypothesis, we have:

$$P(R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0; S_1, A_1; \ldots; S_t, A_t) = P(R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t) \tag{23}$$

This means that the conditional probability that the environment will behave as to provide r as a reward and settle at state s' at time $t+1$ given all its previous states and actions of the agent only depends on the current action and state. This is analogous to the *path independence* hypothesis, which is often used as an ideal approximation for solving problems. In the same way, in some cases, it is safe to assume the Markov property as an approximation for real-world problems that would be too complex to solve otherwise.

4.2.3 Value functions

Value functions are a way of quantifying the quality of a given action in a given state, provided we have an estimate of the future rewards coming from that choice.

A *policy* is a conditional probability distribution of the agent to take action a given that it is currently at state s. It defines a protocol for the agent on how to act at each state, by probabilistically sampling from this distribution, and it is denoted $\pi(a|s)$.

With these concepts, we can define the *state-value function* for state s under policy π as the expectation under π of the returns, provided the state is s:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[G_t \,|\, S_t = s] = \mathop{\mathbb{E}}_{\pi}\left[\sum_k \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{24}$$

If, instead of being at state s at time t and following the policy π thereafter, the agent takes action a, and only then follow π, we define another value function, the *action-state-value* function $q_\pi(s,a)$, which is similar to the state-value function, but also conditioning on action a being taken at time t:

$$q_\pi(s,a) = \mathop{\mathbb{E}}_{\pi}[G_t \,|\, S_t = s, A_t = a] = \mathop{\mathbb{E}}_{\pi}\left[\sum_k \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{25}$$

One advantage in dealing with value functions is that they satisfy recursive relationships, such as the one proven in appendix B:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{26}$$

This equation relates the value of state s under policy π with the values of all its possible

successor states, s'. This represents a sum over all possibilities of the three variables, a, s', and r, weighted by the policy. Equation (26) is called *Bellman equation*, and it represents a value transfer from all possible successor states back to state s, creating a tree-like structure and connecting the value of the full path to the next step. It is the basis of the class of RL algorithms called *Q-learning*.

4.2.4 Optimality relations

The goal of reinforcement learning algorithms is to maximize the total expected return, which is the sum of rewards over time. Having defined value functions, we can use them to define a partial ordering over the family of policy functions in the following way:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s),\ \forall s \in \mathcal{S}$$

Again, $\mathcal{S}$ is the set of all possible states the agent can take, assumed finite in our case.

With this finite partial ordering, there is always one policy that is better than or equal to all other ones, which is called the *optimal policy*, denoted as π_*. If there is more than one policy sharing the same optimal state-value, all of them are said to be optimal and denoted π_*. And their value functions are:

$$v_*(s) = \max_\pi v_\pi(s),\ \forall s \in \mathcal{S}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a),\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}(s)$$

Note that $\mathcal{A}(s)$ is the set of all possible actions to be taken at state $s \in \mathcal{S}$.

Under this (set of) policy(ies), it can be shown [10] that Bellman equation takes a special form, called *Bellman optimality equation*, which does not depend on the policy, since optimal policies are uniquely defined as having the largest state-values:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma\, v_*(s')] \tag{27}$$

Or, in terms of the action-state-value function:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \tag{28}$$

Note that, since there are N possible states, we have one Bellman optimality equation for each state - that is, a set of N nonlinear equations with N unknowns (each of the $v_*(s)$). Moreover, it can be proven that, when one is dealing with a finite Markov decision process, equation (28) has a unique solution [10].

Also note that, given $v_*(s)$, obtaining the optimal policy is simple: for each state s, there is one (or more) actions a leading to the maximum value $v_*(s)$. Since only those actions are relevant to us, we build a policy by simply assigning nonzero probability only to them, and zero to all others. We have a similar protocol for q_*, but with one less step, since action a is already chosen as a conditional of the value function.

Value functions are therefore important because they connect long-term behavior (maximization of the sum of all future rewards) to short-term. Once the optimal policy is determined through this process, the locally optimal solution is equivalent to the globally optimal, and one can then proceed greedily to find the most suitable path.

Despite this advantage, this method has two drawbacks: one, it requires storing the value function (either v_π or q_π) for each possible state, and the number of states becomes prohibitively large for complex environments. The second issue is that computing the Bellman set of nonlinear equations is computationally expensive. One workaround for both of these issues is to store only a few of the possible input states and to approximate q_π with a neural network, generalizing to unseen states a prediction for the value function coming from known states. This reduces the possibilities to only a few, also making the computation of each step faster. This can be done through several machine learning methods, but it is particularly well-suit for neural networks, given that this problem is about learning a function that fits known data and generalizes to unknown data, similarly to supervised learning on networks.

4.3 Deep Q-learning

So far, we have learned to find the optimal policy as an exact procedure that requires one to store the value functions for all possible states. This, as mentioned, only works for environments with only a few possible states. When this number of states grows, we need to find alternatives to the so-called *tabular methods*, avoiding the computation of all possible values and approximating them using a neural network.

4.4 Replay memory

In order to use a neural network in a supervised learning protocol, we need to gather data to be used as the training set. As mentioned before, a big difference between supervised and reinforcement learning is that the latter class of methods collects data from the environment dynamically as the agent moves around, while the former uses a static, *a-priori* training dataset. In the case of deep Q-networks (DQNs), this is done by building what is called a *replay memory*. For each time step t, the agent explores the environment, collecting data. The main variables collected are:

- The current state, s_t
- The action a_t taken while at state s_t
- The reward r_{t+1} at $t+1$ coming from that action
- The new state s_{t+1} after the action

Those variables can be stored as a tuple called the *experience* at time t:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \tag{29}$$

The collected experiences are then stored in a *replay memory* of the agent. To limit the size of this memory, we can use a threshold for the maximum number of experiences N. This memory is then shuffled to train the network, to avoid feeding it with training data points that are too correlated with each other, biasing the algorithm.

This memory thus provides the inputs x_i of the training set for the network. However, we still need the target outputs $y(x_i)$ for each of those inputs. To create them,

the algorithm processes the inputs using the network itself, as explained in the next section. For this reason, the learning part cannot be considered to be purely supervised [10], since both output and target are generated by the network.

4.4.1 Neural network architecture

The networks used to find the approximately optimal policy on a deep Q-learning setting take as input each state, with some method of data encoding (refer to section 2). It then processes this input state and outputs a few possibilities of the state-action-value function, each corresponding to one possible action.

This network has a simple feed-forward topology, and, for us to solve the Bellman equation approximately using it, we substitute each possible state-action-value in the RHS of equation (28). This can then be compared to the LHS by using the difference between the two as a loss function for the network: $\mathcal{L} = q_*(s, a) - q(s, a)$.

To compute the optimal state-action-value function $q_*(s, a)$ using equation (28), we need a term which is the maximum value of $q_*(s', a')$, where s' is the next state the agent can go to, and a' all possible actions to be taken, to find the best possible value. There are a few ways to do this, one of which is simply by re-running the network on a feed-forward basis, but now, instead of using state s as input, we use its successor, s'.

With q_* we have the "prediction" for the optimal value function, which can be compared to the network's output. The difference defines a loss function (or variations of it, such as the mean squared error loss discussed in section 2). We can then apply any gradient-descent optimization method for minimizing the loss and approximately solve the Bellman equation for state s. Note that the solution is restricted to that state, and its computation is done online, while the agent explores the environment. After computing the approximate q value for state s, the method proceeds to a new state at time $t + 1$ and starts again. To ensure that the same state is visited several times in order to iterate and get closer to optimality, some algorithms split the training into episodes, which improves the learning performance. This way, after a few episodes, the DQN method manages to find good approximations for the optimal policy without the computational cost of the tabular method.

5 Quantum reinforcement learning

5.1 Introduction

Finally, after having introduced classical neural networks, and variational quantum algorithms, understanding how to solve the binary classification for the bit string parity problem, and studying reinforcement learning using classical DQNs, we have gathered enough tools to tackle the core problem of this project: studying the adaptation of DQNs to using parameterized variational quantum circuits instead of classical NNs, and trying to replicate the application of this hybrid quantum-classical scheme to solve a canonical problem of reinforcement learning - namely, the *Cart Pole* problem [3]:

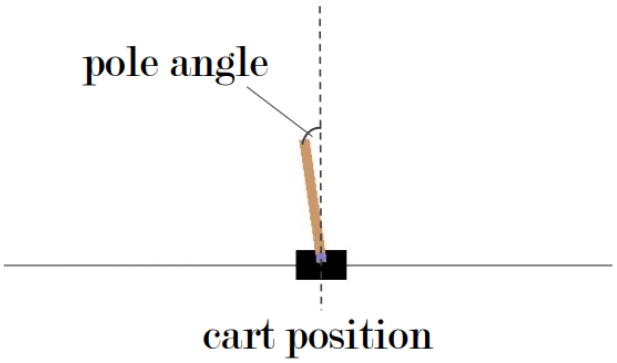


Figure 6: Schematic of the *Cart Pole* environment [1].

5.2 Classical parts

By following the work done in [1], we use the same DQN RL setting explained in the previous section, but replace the NN with a variational quantum circuit.

In this algorithm, to compute the expected optimal value function, we replace the second forward pass in the network using the successor state in equation (28) by a second DQN, which receives the inputs one step delayed, to prevent the creation of a constructive feedback loop that could lead to oscillations - thus increasing the stability of the system [18]. The algorithm also follows an ϵ-greedy strategy, alternating between exploration and exploitation.

The loss function chosen in the reference article is the mean squared error (MSE). To build it, we first construct a vector whose entries are the Q-values for each possible action on state s. This vector, $\vec{q}$, can then be compared to another Q-vector, which is the output of the secondary DQN when passing the successor state s', called $\vec{q}_\delta$ (δ is the number of episodes that the secondary DQN is behind the main network). This comparison is done with an MSE sweeping over all possible actions and states in a batch of shuffled experiences $\mathcal{B}$, retrieved from the replay memory:

$$\mathcal{L}(q, q_\delta) = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \sqrt{\sum_{i=1}^{|\mathcal{A}|} (q_{b_i} - q_{\delta_{bi}})^2} \quad , \tag{30}$$

where $|\mathcal{B}|$ and $|\mathcal{A}|$ denote the cardinality of the batch set of experiences, and that of all possible actions, respectively. Note that both sets are assumed to be finite in our problem.

5.3 Quantum parts

5.3.1 Circuit topology

In order to replace classical neural networks, the article proposes the following ansatz, which is inspired by the analysis of hardware-efficient ansätze [28]. This is a layer that gets repeated several times. It was proven [29] that the expressivity of a quantum circuit is limited by how many times the data gets re-uploaded at the beginning of the circuit. Hence, it is often useful to repeat this pattern a few times.

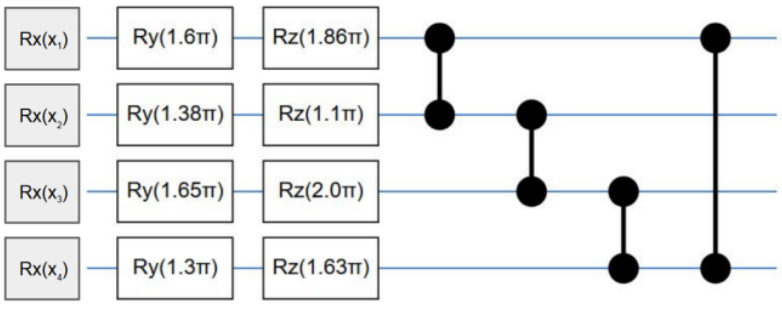


Figure 7: Schematic of the circuit ansatz used for this problem [1].

5.3.2 Encoding states

When one is dealing with a discrete state space, the encoding can be made in simple ways, such as mapping states to bit strings, as in the binary classifier implementation. But when the state space is continuous, as is the case in the *Cart Pole* problem, which has 4 continuous variables (position, speed, pole angle, and pole linear velocity at its tip), we need other approaches. In the strategy chosen by the authors, each input x gets re-scaled to $x' = \arctan(x)$, and these transformed inputs are used as angles in the X-rotations shown in figure 7. Another step used in the article to increase expressivity is to introduce trainable weights w_i in the argument of the arctan function, as:

$$x'_i = \arctan(x_i \cdot w_i) \tag{31}$$

5.3.3 Network output

The circuit takes each state as an input and outputs values of q for each possible action as a measurement of the following form [3]:

$$q_\theta(s, a) = \langle \mathcal{O}_a \rangle_{s,\theta} \quad , \tag{32}$$

where $\langle \mathcal{O}_a \rangle_{s,\theta}$ are the expected values of the observables $\mathcal{O}_a$, one per possible action.

One important point to recall is that VQCs are different from neural networks in the sense that they have fixed ranges of output, determined by the chosen observable to be measured. Because of this, the choice of observable should vary depending on the problem, to match the range of the Q-function.

5.4 Algorithmic implementation

The quantum DQN algorithm is implemented in the original paper [1], and one of the problems being solved is the one in the *Cart Pole v0* environment, in which an agent needs to balance a pole on a cart that is free to move horizontally in one direction on a frictionless track. As mentioned before, the state space is continuous, with four variables, and both cart position and pole angle have maximum and minimum values out of which the agent loses and the episode ends.

In the beginning, all four variables are randomly initialized, and the ϵ parameter is set to 1, so the agent moves freely through the environment, in a purely exploratory fashion, collecting experience points, e_t, that will be randomly batched to train the network for computing an approximation to the optimal value function.

After filling the replay memory up to its capacity with random moves, the algorithm starts decreasing the value of ϵ, along with the probability of random exploratory moves. In this phase, it starts computing approximations for the Q-value functions at each state, moving to a more exploitative phase.

Each step the agent moves, it collects positive points, which are summed along the training. Each episode ends when either the agent lets the pole fall or it reaches the maximum number of steps of $H = 200$.

The training is then repeated for a new episode, iteratively, re-vising some of the states it has visited previously. With each new episode, the value function at each state gets closer to the optimality condition, meaning the agent can collect more rewards than before. The environment is said to be *solved* when the average score of the last 100 episodes is at least 195.

Note that, since we have the $H = 200$ cap in the number of steps on a given episode, the value function, which is the expected value of the returns, becomes a finite sum, and we can estimate lower and upper bounds for it. With these bounds, we can re-scale the output range, from the one defined by the quantum operator to be measured to the range of values taken by the output $\vec{q}$.

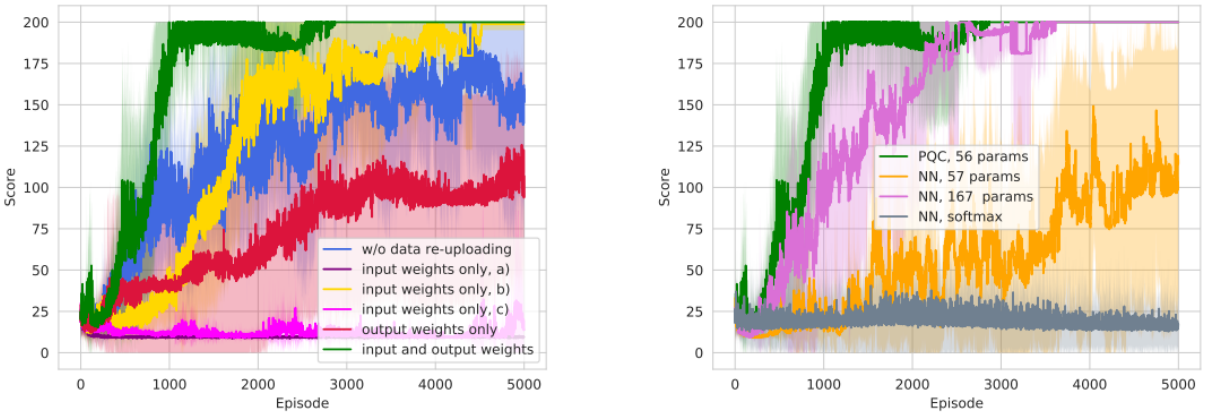
This re-scaling can be carried out by two different methods: either multiplying the output of the VQC by a constant factor, which depends on the problem or having flexible output weights that can be trained in the process alongside the other parameters of the circuit. In [1], the authors choose the second one, so the output of the circuit is given by:

$$q(s,a) = \frac{w_{\mathcal{O}_a}}{2}\left[\langle 0^{\otimes 4}| U_{\vec{\theta}}(s)^{\dagger}\mathcal{O}_a U_{\vec{\theta}}(s) |0^{\otimes 4}\rangle + 1\right] \tag{33}$$

Furthermore, the measured operator, $\mathcal{O}_a$, is chosen to be different when the actions are taken to the left ($\mathcal{O}_L = Z_1Z_2$) or to the right ($\mathcal{O}_R = Z_3Z_4$), both being Pauli-Z product operators, one acting on the qubit pair (1,2) and the other one on (3,4).

5.4.1 Results

By running the algorithm several times using the classical optimizer *Adam* [30], which is an extension of the stochastic gradient descent algorithm [19], and by making several specific hyperparameter adjustments [1], the authors reported the following charts:



(a) average scores with varying trainable weights

(b) average scores compared to NN

Figure 8: Graphs of average score by episode for several VQC model adjustments (a), and also compared to classical neural networks (b) [1].

As can be seen in the left-hand side chart, the configuration that achieved higher average scores in the least amount of episodes was the one in which the VQC had tunable parameters both in the input and output. Also, when there was no data re-uploading, the quality of the training oscillated more, as shown in the blue curve.

From the second chart, we see that, for the same number of parameters, the quantum version achieved better scores in fewer episodes than a classical neural network with 3 dense layers (having 4, 5, and 2 units, respectively, in each layer), and that it required almost three times fewer parameters to achieve the same result as the fully classical algorithm. Therefore, at least in this simple environment, the quantum version is competitive with the classical NN implementation, even showing some learning advantages over its classical counterpart.

By following the authors' implementation [3, 27], we ran the algorithm for the *Cart Pole* environment on a personal computer, a Linux Ubuntu 20.04 bash on a Windows machine, with Intel i5 7th gen processor, and the obtained result was the following:

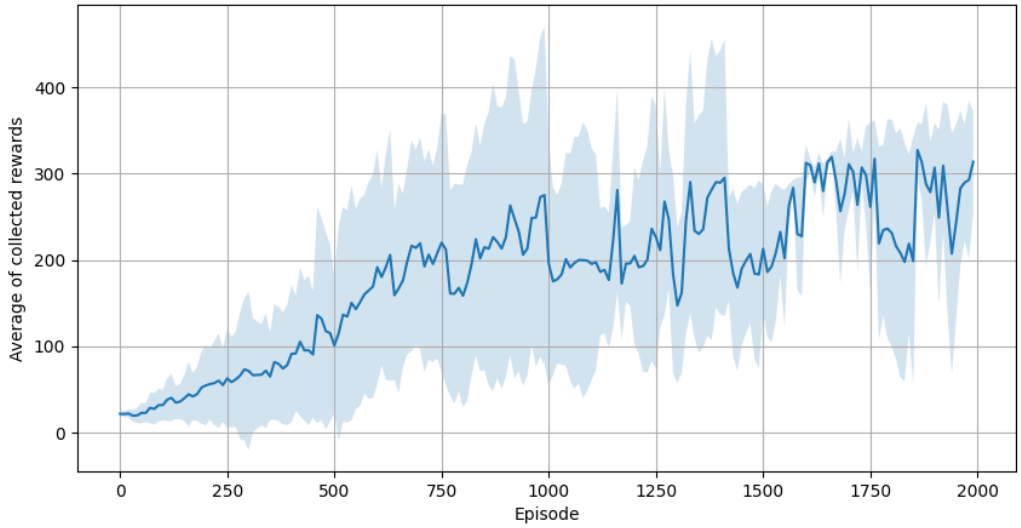


Figure 9: Collected rewards (10-episode moving average for 10 consecutive runs), with the average and standard deviation of all runs.

To generate this chart, we ran the code reported by the authors 10 times (resetting the training each time) and took the average and standard deviation of the runs, each of them considering a 10-episode moving average of the collected rewards.

Our results are noisier than the authors', most likely due to averaging over fewer runs. Even so, one can see an improvement in the learning process, gathering more rewards in the states as the algorithm proceeds. Note that, in the available code, the cutoff is achieved by either reaching 500 in collected rewards once or by hitting 2000 episodes - criteria that are slightly different from the one used in the original formulation of the game [1]. Also note that, on chart 9, after around 1000 episodes, the sample size shrinks, and only a few of the training runs had still not achieved the 500 points. Furthermore, as can be seen from the plot of all individual runs on appendix C, after around 1500 episodes, only two of the runs had still not finished, and the two curves intersect around episode 1600, explaining the sudden shrinkage on the standard deviation around that region.

6 Conclusions and perspectives

In this final section, we summarize what was covered in this text, as well as possible future paths this project could take.

Starting from minimal initial knowledge of machine learning techniques and quantum computing, we first studied how classical neural networks learn functions that fit data and can also extend the results from previously unseen data [8]. This was done under the supervised learning paradigm.

After that, we covered the idea of a quantum circuit and how we can use parameterized circuits to emulate the workings of neural networks. To get a better intuition in this topic, we focused on understanding and recreating the implementation of a quantum binary classifier using VQAs [2, 26, 9], which was used to learn the parity of an input bit string. This is one of the simplest use cases of VQAs for supervised learning, but it carries most of the main aspects of this class of methods, being a valuable introduction to the topic.

After that study, we deep-dived into classical reinforcement learning [10] - in particular, the deep Q-learning method, which optimizes the value function for each state and action to build an optimal policy. We have also seen that tabular approaches to this problem are usually not scalable, and studied the application of a neural network to reduce the number of calculations necessary [18], enabling the so-called Deep Q-Learning method to solve problems for more complex environments.

Finally, we brought all the ingredients together and studied one particular implementation of Deep Q-Learning replacing the classical neural network with a VQC using a hardware-efficient ansatz, and studied the authors' results [1] with a few variations in the method, also comparing it to classical neural networks, showing that, for the same number of parameters, the VQC-based model outperformed the classical one in terms of how fast it was able to train the agent. By running their implementation of the method, available at [3], we generated a curve that was noisier, likely due to fewer averages over different runs, but that clearly showed the evolution in the training process over the episodes.

As for possible ramifications of this work, there are a few points that could not be covered in depth, and which may be important knowledge to better understanding quantum machine learning algorithms. A few of them we can quote are:

- Better understanding how ansätze and data encoding techniques affect the expressivity and learning of quantum machine learning techniques based on VQAs [29].
- Studying a few articles on hyperparameter tuning, to understand how they affect the quality of learning. This includes both classical and quantum machine learning [31, 32].
- In the quantum DQN problem, better understanding the choice of operators being made to match the output range to the problem at hand (for instance, in the *Cart Pole* solution described, there were different operators for left and right movements of the agent [1])
- Understanding other quantum reinforcement learning techniques for finding optimal policies [33]
- Studying implementations of reinforcement learning methods in real-world applications in which, for example, the agent collects data from image sensors, and the networks used are Convolutional Neural Networks (CNNs) [34]

These are only a few possible paths to take from this point onwards in this rich and active field which is quantum machine learning (and, in particular, quantum reinforcement learning), with promising applications in the near future in several fields of science and engineering.

Appendix A Reed-Muller expansion

Consider a Boolean function $f : \{0,1\}^N \to \{0,1\}$, which takes as input a bit string $x_1 x_2 \dots x_N$. Since there are 2 options for each entry string, there are 2^N input possibilities. Furthermore, for each of those possibilities, there are 2 output options, 0 or 1. Thus there are 2^{2^N} such functions.

It is known [35] that all Boolean functions can be fully defined by writing down explicitly their truth tables, which is the same as listing all the individual correspondences between elements of domain and image. But there are ways to represent a Boolean function that do not rely on the truth table, and one of the simplest ones is the *Reed-Muller expansion*. To understand this expansion, first, let us define:

$$f_{x_i}(\vec{x}) = f(x_1, x_2, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) \tag{34}$$

$$f_{\bar{x}_i}(\vec{x}) = f(x_1, x_2, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) \tag{35}$$

which are called the *Shannon cofactors* of f with respect to x_i. These are the only values the function can take with respect to its i-th entry. Further, we define the Boolean difference:

$$df_{x_i} = \frac{\partial f}{\partial x_i} \coloneqq f_{x_i} \oplus f_{\bar{x}_i} \ , \tag{36}$$

which is equal to one if, by changing variable x_i, the function changes its value, and it is equal to zero otherwise.

With these definitions, we state the first-order Reed-Muller expansion for f:

$$f = f_{\bar{x}_i} \oplus x_i \frac{\partial f}{\partial x_i} \tag{37}$$

This equation can be verified by splitting it into two possible cases:

- If f does not change when varying x_i, then $f_{x_i} = f_{\bar{x}_i}$, and the XOR operation in the definition of the Boolean difference returns zero, so $f = f_{x_i}$
- Now, consider f flips its value when varying x_i. Then the Boolean difference is equal to 1, so that $f = f_{\bar{x}_i} \oplus x_i$, and then we have two possible cases: $x_i = 0 \implies f_{\bar{x}_i} = f$

or $x_i = 1 \implies f_{\bar{x}_i} = \bar{f} \implies f = f_{\bar{x}_i} \oplus 1$

The Reed-Muller formula can be further expanded to the other variables x_j, until reaching an expansion with constant coefficients. In this expansion, f can be written as the full form of the Reed-Muller expansion [35]:

$$f(\vec{x}) = a_0 \oplus (a_1 x_1 \oplus a_2 x_2 \oplus \ldots \oplus a_N x_N) \oplus (a_{12} x_1 x_2 \oplus a_{13} x_1 x_3 \oplus \ldots \oplus a_{NN} x_N x_{N-1}) \oplus \\ \oplus \ldots \oplus a_{123\ldots N} x_1 x_2 \ldots x_N$$

Appendix B Bellman equation

In this section we prove Bellman recursive equation following [36].
The definition of the state-value function under policy π is:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[G_t|S_t = s] \quad , \tag{38}$$

which, by the definition of the return G_t given in equation (22), together with the linearity of the expectation operator give:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[R_{t+1} + \gamma G_{t+1}|S_t = s] = \mathop{\mathbb{E}}_{\pi}[R_{t+1}|S_t = s] + \gamma \mathop{\mathbb{E}}_{\pi}[G_{t+1}|S_t = s] \tag{39}$$

The first term of the right-hand side of the equation can be written as:

$$\mathop{\mathbb{E}}_{\pi}[R_{t+1}|S_t = s] = \sum_{r\in\mathcal{R}} rp(r|s) \tag{40}$$

Here $p(r|s)$ is the probability of receiving reward r given that the agent is at state s at time t. This is equivalent to the union of all possible actions a being taken at time t, and all possible states s' at $t+1$ deriving from that action, that is:

$$p(r|s) = \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} p(s', a, r|s) \quad , \tag{41}$$

and, using the multiplication rule, considering the definition of policy π: $\pi(a|s) = p(a|s)$, we have:

$$p(r|s) = \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} \pi(a|s)p(s', r|a, s)$$

Therefore, by joining all pieces together, the first term at the RHS of equation (39) becomes:

$$\mathop{\mathbb{E}}_{\pi}[R_{t+1}|S_t = s] = \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} r\pi(a|s)p(s', r|a, s) \tag{42}$$

Now, let us also expand the second term at the RHS of (39):

$$\mathop{\mathbb{E}}_{\pi}[G_{t+1}|S_t = s] = \sum_{g\in\mathcal{G}} gp(g|s) \quad , \tag{43}$$

where it is assumed that the return G_{t+1} is a random variable that can take on a finite number of values $g \in \mathcal{G}$.

Similarly to equation (41), the probability distribution $p(g|s)$ sweeps over all possible states s' at time $t+1$, all possible actions a leading to those states, but now also sweeping over the possible returns r:

$$p(g|s) = \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} p(s', r, a, g|s) \ , \tag{44}$$

and, applying the multiplication (marginalization) rule twice, with the definition of the policy π:

$$\begin{aligned} p(g|s) &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} p(g|s', r, a, s) p(s', r, a|s) \\ &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} p(g|s', r, a, s) p(s', r|a, s) \pi(a|s) \end{aligned} \tag{45}$$

Now, let us assume that we are in a Markovian system, which means that only the action at time t and state at $t+1$ can influence variables at $t+1$, so the conditional probability term of equation (45) simplifies to:

$$p(g|s) = \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} p(g|s') p(s', r|a, s) \pi(a|s) \tag{46}$$

Joining the pieces together for the second term at the RHS of equation (39):

$$\begin{aligned} \mathop{\mathbb{E}}_{\pi} [G_{t+1}|S_t = s] &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} \left[\sum_{g\in\mathcal{G}} g p(g|s') \right] p(s', r|a, s) \pi(a|s) \\ &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} \mathop{\mathbb{E}}_{\pi} [G_{t+1}|S_{t+1} = s'] \, p(s', r|a, s) \pi(a|s) \ , \end{aligned} \tag{47}$$

and, finally, using the definition of v_π, we get:

$$\gamma \mathop{\mathbb{E}}_{\pi} [G_{t+1}|S_t = s] = \gamma \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}(s')} v_\pi(s') p(s', r|a, s) \pi(a|s) \tag{48}$$

Finally, joining both terms (eqs (42) and (48)) in the original equation ((39)),

we complete the proof of Bellman equation:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[R_{t+1}|S_t = s] = \sum_{r\in\mathcal{R}}\sum_{s'\in\mathcal{S}}\sum_{a\in\mathcal{A}(s')} r\pi(a|s)p(s',r|a,s) + \gamma\sum_{r\in\mathcal{R}}\sum_{s'\in\mathcal{S}}\sum_{a\in\mathcal{A}(s')} v_\pi(s')p(s',r|a,s)\pi(a|s)$$

$$\boxed{v_\pi(s) = \sum_{a\in\mathcal{A}(s')}\pi(a|s)\sum_{r\in\mathcal{R}}\sum_{s'\in\mathcal{S}} p(s',r|a,s)[r+\gamma v_\pi(s')]} \tag{49}$$

Appendix C Implementation results and tables

In this section, we provide tables and additional charts of sections 3 and 5.

C.1 Section 3: VQC implementation of the parity classifier

Table 1: Cost function, accuracy and iterations of the VQA of section 3.

Iteration	Cost	Accuracy
1	3.4355534	50%
2	1.9717733	50%
3	1.8182812	50%
4	1.5042404	50%
5	1.1477739	50%
6	1.273499	62.5%
7	0.8290628	50%
8	0.3226183	100%
9	0.1436206	100%
10	0.298281	100%
11	0.3064355	100%
12	0.1682335	100%
13	0.0892512	100%
14	0.0381562	100%
15	0.0170359	100%
16	0.0109353	100%
17	0.0108388	100%
18	0.0139196	100%
19	0.012398	100%
20	0.0085416	100%
21	0.0053549	100%
22	0.0065759	100%
23	0.0024883	100%
24	0.0029102	100%
25	0.0023471	100%

C.2 Section 5: Quantum DQN implementation

Next, we show the chart containing all individual runs whose collective behavior is displayed on image 9, a result that we generated by running 10 times the code reported by the authors [3, 27]:

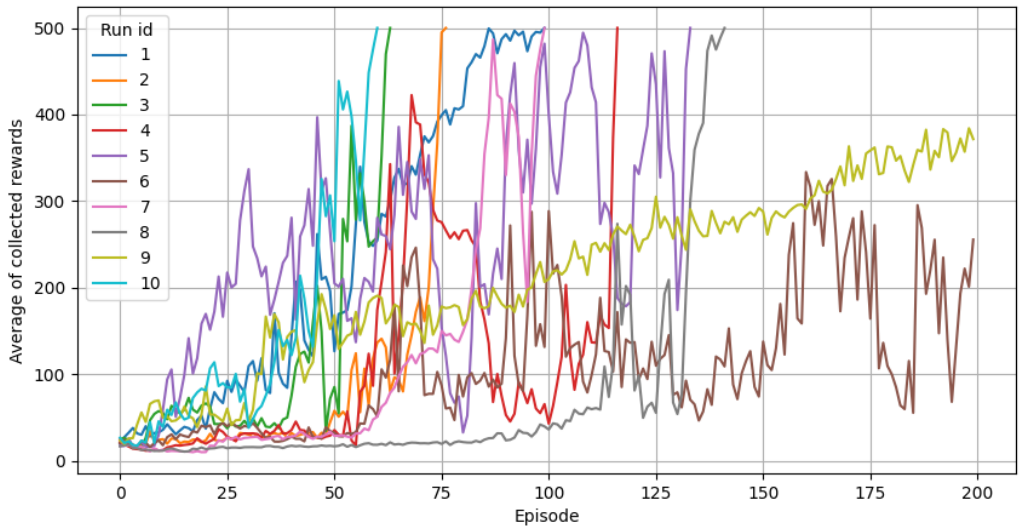


Figure 10: Collected rewards (10-episode moving average) for each of the 10 individual runs.

The table containing the values associated with these curves can be found in the "data" folder of the GitHub of this project [27].

References

[1] Andrea Skolik et al. Quantum agents in the gym: a variational quantum algorithm for deep q-learning. *Quantum*, 6:720, 2022.

[2] Edward Farhi and Hartmut Neven. Classification with quantum neural networks on near term processors. 2018.

[3] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[4] Frank Arute et al. Quantum supremacy using a programmable superconducting processor. 574, 2019.

[5] Wikipedia. Quantum computing wiki. https://en.wikipedia.org/wiki/Quantum_computing, 2022.

[6] Fuzhen Zhuang et al. A comprehensive survey on transfer learning, 2019.

[7] Nicola Dalla Pozza et al. Quantum reinforcement learning: the maze problem, 2021.

[8] Nguembang Fadja et al. Vision inspection with neural networks. 12 2018.

[9] Xanadu. Variational classifier. https://pennylane.ai/qml/demos/tutorial_variational_classifier.html, 2021.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2 edition, 2015.

[11] Ville Bergholm et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2018.

[12] Cirq Developers. Cirq, 2022. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[13] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. 5, 1943.

[14] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6), 2017.

[15] Bin Ding et al. Activation functions and their characteristics in deep neural networks. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 1836–1841, 2018.

[16] Sunitha Basodi et al. Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3, 2020.

[17] Yigit Alparslan et al. Towards searching efficient and accurate neural network architectures in binary classification problems. 2021.

[18] Volodymyr Mnih et al. Human-level control through deep reinforcement learning. 518, 2015.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[20] C. Cohen-Tannoudji et al. *Quantum Mechanics*. Wiley, New York, 2005.

[21] F. Bloch. Nuclear induction. *Phys. Rev.*, 70:460–474, 1946.

[22] Wikimedia Commons. Bloch sphere. https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg, 2009.

[23] Otto Menegasso Pires et al. Quantum circuit synthesis using projective simulation. *Inteligencia Artificial*, 24, 2021.

[24] Arthur Ekert and Tim Hosgood. *Introduction to Quantum Information Science*. 2022.

[25] Yudong Cao et al. Quantum neuron: an elementary building block for machine learning on quantum computers, 2017.

[26] Seokwon Yoo et al. A quantum speedup in machine learning: finding anin/i-bit boolean function for a classification. *New Journal of Physics*, 16, 2014.

[27] Alan Piovesana. Quantum reinforcement learning github repository. https://github.com/Alandroid/quantum-reinforcement-learning, 2022.

[28] Abhinav Kandala et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. 549, 2017.

[29] Maria Schuld et al. Effect of data encoding on the expressive power of variational quantum-machine-learning models. *Physical Review A*, 103(3), 2021.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[31] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications, 2020.

[32] Xavier Bonet-Monroig et al. Performance comparison of optimization methods on variational quantum algorithms, 2021.

[33] Sofiene Jerbi et al. Parametrized quantum policies for reinforcement learning, 2021.

[34] Tanmay Shankar et al. Reinforcement learning via recurrent convolutional neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2592–2597, 2016.

[35] *Reversible Computing*, chapter 1, pages 5–16. John Wiley Sons, Ltd.

[36] Stack Exchange. Proof of bellman equation. https://stats.stackexchange.com/questions/243384/deriving-bellmans-equation-in-reinforcement-learning, 2018.