

---

# Generant Imatges amb un Ordinador Quàntic

---

**TREBALL DE RECERCA DE BATXILLERAT**

**IES MIQUEL TARRADELL**

**Autor:**

Tomàs Ockier Poblet  
2n Batxillerat  
[ockier1@gmail.com](mailto:ockier1@gmail.com)

**Tutor:**

Montserrat Parcet Bertran



Institut Miquel Tarradell

3 de gener de 2022  
Barcelona, Barcelona



# Índex

<b>I</b>	<b>Marc Teòric</b>	<b>8</b>
<b>1</b>	<b>Àlgebra lineal</b>	<b>9</b>
1.1	Vectors i espais vectorials . . . . .	9
1.2	Aplicacions lineals . . . . .	12
1.1	Tipus d'aplicacions lineals . . . . .	13
1.3	Producte interior i producte exterior . . . . .	15
1.1	Producte interior . . . . .	15
1.1.1	Propietats del producte interior . . . . .	16
1.2	Vectors ortonormals i ortogonals . . . . .	17
1.3	Producte exterior . . . . .	18
1.4	Producte tensorial . . . . .	19
1.1	Propietats del producte tensorial . . . . .	21
1.5	Traça . . . . .	22
<b>2</b>	<b>Computació quàntica</b>	<b>24</b>
2.1	Estats quàntics i superposicions . . . . .	24
2.2	Qubits i operacions quàntiques . . . . .	28
2.1	Representació geomètrica d'un qubit . . . . .	30
2.2	Operacions per a només un qubit . . . . .	31
2.3	Circuit quàntics . . . . .	34
2.4	Operacions per a múltiples qubits . . . . .	35
2.4.1	Entrellaçament quàntic . . . . .	36
2.4.2	Operacions controlades . . . . .	36
2.3	Mesurament quàntic . . . . .	38
2.4	Matriu de densitat . . . . .	40
2.1	Matriu de densitat reduïda . . . . .	41
2.1.1	Mesurament parcial . . . . .	43
2.5	Ordinadors quàntics . . . . .	44

<i>ÍNDEX</i>	2
<b>3 Intel·ligència artificial</b>	<b>45</b>
3.1 Xarxes neuronals . . . . .	46
3.2 Descens del gradient . . . . .	49
3.1 Backpropagation . . . . .	52
3.3 Generative adversarial networks . . . . .	54
<b>4 Generació d'imatges amb un ordinador quàntic</b>	<b>57</b>
4.1 Descens del gradient quàntic . . . . .	58
4.2 Circuits quàntics per xarxes neuronals . . . . .	58
 <b>II Part Experimental</b>	 <b>62</b>
<b>5 Plantejament de l'hipòtesi</b>	<b>63</b>
<b>6 Programació del model</b>	<b>64</b>
6.1 Discriminador . . . . .	65
6.2 Generador . . . . .	67
6.3 Creació del model . . . . .	70
6.4 Execució del model . . . . .	70
<b>7 Realització del experiment</b>	<b>74</b>
7.1 Anàlisi dels resultats . . . . .	75
 <b>III Conclusions</b>	 <b>78</b>
 <b>IV Apèndix</b>	 <b>83</b>
<b>Appendices</b>	<b>84</b>
<b>A Més àlgebra lineal</b>	<b>85</b>
A.1 Procediment de Gram–Schmidt . . . . .	85
A.2 Curs ràpid de la notació de Dirac . . . . .	87

<i>ÍNDEX</i>	3
<b>B Computació Quàntica vs Mecànica Quàntica</b>	<b>88</b>
B.1 Normalitzar . . . . .	89
<b>C Polarització d'un fotó</b>	<b>91</b>
<b>D Complexitat i algoritmes quàntics</b>	<b>93</b>
D.1 Algoritme de Grover . . . . .	94
D.2 Algoritme de Shor . . . . .	95
<b>E Codi</b>	<b>96</b>
E.1 Part I . . . . .	96
E.1 Capítol 3 . . . . .	96
E.1.0.1 Regressió lineal . . . . .	96
E.2 Part II . . . . .	97
E.1 Capítol 6 . . . . .	97
E.1.0.1 Codi original per la xarxa neuronal clàssica	97
E.1.0.2 Codi final per el discriminador . . . . .	100
E.1.0.3 Codi per el generador . . . . .	104
E.1.0.4 Definició del model . . . . .	110
E.1.0.5 Execució del model . . . . .	116
E.1.0.6 Multiprocessament . . . . .	117
E.1.0.7 Funcions extres . . . . .	120

## **Agraïments**

Volia agrair a la mfahfajlsdfhjads fladsm

## **Nota**

Abans de començar amb el treball vull dedicar un petit espai per poder parlar de que significa aquest treball per a mi, del tot l'esforç que he posat en aquest i finalment comentar tot lo que he après al llarg d'aquest llarg camí.



# **Introducció**



Desde hace más de un año, me he dedicado a estudiar computación cuántica durante mi tiempo libre. Buscaba investigar un campo relacionado con la mecánica cuántica, pero sin que sea muy complicado, que se pueda entender a un nivel teórico y que me entusiasme.

La Computación Cuántica encaja perfectamente con esos criterios. Es más sencilla que la mecánica cuántica debido a que no está basada en cálculo o ecuaciones diferenciales, se basa en la álgebra lineal, utilizando valores discretos, vectores y matrices. Además si se trabaja a un nivel teórico sencillo, no se tienen en consideración las interpretaciones físicas, lo cual simplifica mucho las cosas. Cuanto más me adentraba, más ganas tenía de seguir.

Mi parte favorita de este campo es el Quantum Machine Learning que consiste en diseñar y aplicar conceptos de Machine Learning a los ordenadores cuánticos, como por ejemplo implementar cuánticamente las famosas Redes Neuronales, que están detrás de la mayoría de inteligencias artificiales que vemos hoy en día [1].

QML es un campo de investigación joven y en crecimiento debido a que sus algoritmos son ideales para implementarlos con los ordenadores cuánticos actuales, los cuales no son muy potentes. Ejemplos de estas implementaciones serían [insertar aplicaciones aquí], etc.

De entre todos los tipos de algoritmos me he centrado en las Redes Neuronales Cuánticas, análogas cuánticas de las Redes Neuronales tan utilizadas hoy en día para hacer gran variedad de tareas. Me he interesado particularmente en ellas debido a que tenía experiencia en el pasado con las RNs clásicas y había visto que existen frameworks de software para trabajar con ellas como TensorFlow Quantum [2] que me podían ayudar.

Para adentrarme en el campo de QML, he tenido que adquirir conocimientos en álgebra lineal, cálculo y física. Dentro de QML en concreto me he dedicado a leer papers que me interesan y en un par de ocasiones intentar implementar los algoritmos detallados en esos papers. Puede parecer algo imposible en principio debido a que no tengo acceso directo a un ordenador cuántico, no obstante

estos no son necesarios debido a que las operaciones cuánticas pueden ser simuladas en un ordenador corriente de escritorio (con ciertas limitaciones). Pero puedo tener acceso a ordenadores cuánticos ya que IBM permite acceder a los suyos mediante IBM Quantum Experience [3], aunque nunca he dado uso de ello debido a que no lo veía necesario.

En este trabajo de investigación me he propuesto implementar mediante código uno de los algoritmos que he visto en un paper, una Red Adversaria Generativa Cuántica (GAN, en inglés) [4] que genera imágenes a partir de un circuito cuántico [5]. Como objetivo tengo verificar una sugerencia que hacen los autores del artículo: implementar una función no-lineal en una parte del algoritmo que podría mejorar el rendimiento de este. Mi hipótesis al igual que los autores (aunque ellos lo comentan muy brevemente) es que el algoritmo va a reducir ligeramente el número de interacciones que son necesarias para llegar a su punto óptimo. Es decir, el modelo con la función no-lineal va a necesitar menos operaciones que lo entren conseguir los mismos resultados que el modelo sin la función.

# **Part I**

**Marc Teòric**

# Capítol 1

## Àlgebra lineal

Quan vaig començar a buscar informació sobre computació quàntica, en vaig ràpidament adonar compte que necessitava molt més coneixement matemàtic, degut a que no entenia gairebé res dels llibres sobre computació quàntica. Però durant aquell temps, una serie de vídeos sobre àlgebra lineal en va captar l'atenció, que es justament la branca de les matemàtiques sobre la qual es basa la computació quàntica. Els vídeos son les lliçons que dona el Professor Gilbert Strang al Institut Tecnològic de Massachusetts (MIT en anglès) [6, 7]. Una vegada havia vist gairebé tots els vídeos, ja tenia bastants conceptes apresos.

Aquelles lliçons em van ajudar a entendre les matemàtiques de *Quantum Computation and Quantum Information* [8] i *Quantum Computing: A Gentle Introduction*. A poc a poc, vaig anar aprenent els fonaments matemàtics de la computació quàntica i mecànica quàntica.

En aquesta secció aniré explicant els conceptes bàsics de l'àlgebra lineal, per formar els coneixements en matemàtiques necessaris per poder comprendre aquest treball.

### 1.1 Vectors i espais vectorials

Els objectes fonamentals de l'àlgebra lineal són els espais vectorials. Un espai vectorial és el conjunt de tots els vectors amb les mateixes dimensions i amb

certes propietats en comú. Per exemple  $\mathbb{R}^3$  seria l'espai vectorial de tots els vectors de 3 dimensions els quals normalment s'utilitzen per representar punts en un espai tridimensional. En computació quàntica s'utilitzen en concret els espais vectorials anomenats espais de Hilbert, que són aquells en els que hi ha definit un producte interior [9]. En els espais de Hilbert es defineixen un conjunt d'operacions amb certes propietats, la quantitat que és necessària. S'ha de tenir en compte que els espais de Hilbert són molt més complicats que el que es representa en aquest treball, i que d'aquí en endavant, quan mencionï espai vectorial, em referiré a un espai de Hilbert.

Els espais vectorial estan definits per les seves bases, i.e. un conjunt de vectors independents  $B = \{|v_1\rangle, \dots, |v_n\rangle\}$ . Posat d'una altra manera: el conjunt  $B$  és una base pel l'espai  $V$ , si cada vector  $|v\rangle$  en l'espai es pot escriure com  $|v\rangle = \sum_i a_i |v_i\rangle$  per  $|v_i\rangle \in B$ .

La notació estàndard per l'àlgebra lienal en mecànica quàntica es la notació de Dirac, en la qual es representa un vector com  $|\psi\rangle$ . On  $\psi$  es la etiqueta del vector. Un vector  $|\psi\rangle$  amb  $n$  dimensions també pot ser representat com una matriu columna que te la forma:

$$|\psi\rangle = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-1} \\ z_n \end{bmatrix}$$

Pels seus elements  $\{z_1, z_2, \dots, z_{n-1}, z_n\} \in \mathbb{C}$ . Els vectors escrits com  $|\psi\rangle$  s'anomenen *ket*.

En els espais de Hilbert es defineix una adició de vectors<sup>1</sup>:

$$|\psi\rangle + |\varphi\rangle = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix} + \begin{bmatrix} \varphi_1 \\ \vdots \\ \varphi_n \end{bmatrix}$$

I una multiplicació escalar:

$$z|\psi\rangle = z \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix} = \begin{bmatrix} z\psi_1 \\ \vdots \\ z\psi_n \end{bmatrix}$$

On  $z$  és un escalar i  $|\psi\rangle$  un vector.

A més a més, hi ha definit un conjugat complex: Per  $z = a + bi$ , el seu conjugat  $z^*$  és igual a  $a - bi$ .

Aquesta noció es pot ampliar també per les matrius:

$$|\psi\rangle^* = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix}^* = \begin{bmatrix} \psi_1^* \\ \vdots \\ \psi_n^* \end{bmatrix}$$

$$A^* = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}^* = \begin{bmatrix} a_{11}^* & \cdots & a_{1n}^* \\ \vdots & \ddots & \vdots \\ a_{m1}^* & \cdots & a_{mn}^* \end{bmatrix}$$

Amb  $|\psi\rangle$  sent un vector de dimensions  $n$ , i  $A$  sent una matriu de dimensions  $m \times n$ .

Un altre concepte important es la transposada, representada per el superíndex  $T$  que 'rota' un vector o una matriu. Un vector columna amb una dimensió

---

<sup>1</sup> Els vectors d'aquesta definició tenen els seus elements representats per la seva etiqueta i un subíndex e.g. el vector  $|\psi\rangle$  te un element qualsevol  $\psi_1$  i el seu primer element es  $\psi_1$ . Aquesta notació es seguirà utilitzant al llarg del treball.

$n \times 1$  es transforma a un vector fila amb una dimensió  $1 \times n$ <sup>2</sup>:

$$|\psi\rangle^T = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix}^T = [\psi_1 \quad \dots \quad \psi_n]$$

El mateix és cert per les matrius, una matriu  $m \times n$  transposada es converteix en una matriu  $n \times m$ . Per exemple:

$$A^T = \begin{bmatrix} 2 & 3 \\ 6 & 4 \\ 2 & 5 \end{bmatrix}^T = \begin{bmatrix} 2 & 6 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

La composició d'un conjugat complex i la transposada s'anomena el conjugat Hermitià, la seva notació es una  $\dagger$  superindexada. Per un vector  $|\psi\rangle$  el seu conjugat Hermitià  $|\psi\rangle^\dagger$  és:

$$|\psi\rangle^\dagger = (|\psi\rangle^*)^T = [\psi_1^* \quad \dots \quad \psi_n^*] = \langle\psi|$$

El conjugat Hermitià compleix que  $|\psi\rangle^\dagger = \langle\psi|$  i  $\langle\psi|^\dagger = |\psi\rangle$ .

El conjugat Hermitià d'un vector columna  $|\psi\rangle$  s'anomena *bra* o vector dual. En la notació de Dirac un vector dual s'escriu com  $\langle\psi|$ .

## 1.2 Aplicacions lineals

Per poder operar amb vectors i fer operacions amb ells, s'utilitzen les matrius, que també son anomenades aplicacions lineal o transformacions lineals; noms que descriuen millor com funcionen aquests objectes. La definició formal d'una aplicació lineal pot ser bastant complicada, per aquesta raó, utilitzaré termes més informals en aquesta secció.

---

<sup>2</sup>En realitat els vectors columna son matrius amb dimensió  $n, 1$  però he estat ometent el 1. Quan hem refereixo a les dimensions de un vector qualsevol, només diré un numero, no obstant, especificaré si és un vector columna o un vector fila.

Bàsicament, una aplicació lineal transforma un vector en un altre vector, que poden o no ser d'espais diferents [10]. Més formalment, per un vector  $|v\rangle$  en un espai  $V$  i un vector  $|w\rangle$  en un espai  $W$ , una aplicació lineal  $A$  entre els vectors, fa l'acció:

$$A|v\rangle = |w\rangle$$

En altres paraules, l'aplicació transforma un element del espai vectorial  $V$  en un vector del espai vectorial  $W$ . Les aplicacions lineals han de complir les següents propietats:

1. Addició de vectors:

Donats els vectors  $|\psi\rangle$  i  $|\varphi\rangle$  en un mateix espai vectorial, i un operador lineal  $A$ :

$$A(|\psi\rangle + |\varphi\rangle) = A|\psi\rangle + A|\varphi\rangle$$

2. Producte escalar:

Donats els vectors  $|\psi\rangle$ , l'escalar  $z$  i la transformació lineal  $A$ , es certs que:

$$A(z|\psi\rangle) = zA|\psi\rangle$$

Aquestes afirmacions han de ser certes per tots els vectors i tots els escalars en els espais on les aplicacions actuen. Cal notar que una aplicació lineal no té perquè ser una matriu necessàriament, per exemple, les derivades i les integrals son aplicacions lineals, això es pot provar fàcilment al veure que compleixen els criteris especificats posteriorment. No obstant, les derivades i les integrals usualment no s'apliquen a vectors, sinó a les funcions, però es possible aplicar-les a vectors.

Les matrius serien la representació matricial de les aplicacions lineals [11].

## 1.1 Tipus d'aplicacions lineals

En la secció actual, exposaré els tipus bàsics d'aplicacions lineals que són indispensables en la teoria presentada en aquest capítol i la resta del treball.



## 1. Operador zero

Qualsevol espai vectorial té un vector zero expressat en notació de Dirac com a  $0$ , degut a que  $|0\rangle$  es un altre concepte totalment diferent en CQ i IQ<sup>3</sup>. Es aquell vector que per qualsevol vector  $|\psi\rangle$  i qualsevol escalar  $z$ , es compleix que: És l'element neutre:  $|\psi\rangle + 0 = |\psi\rangle$

I l'element nul pel producte escalar:  $z0 = 0$ .

El operador zero també s'escriu com a  $0$ .

## 2. Matriu inversa

Un matriu quadrada<sup>4</sup>  $A$  és invertible si existeix una matriu  $A^{-1}$  de manera que  $AA^{-1} = A^{-1}A = I$  és la matriu inversa de  $A$ . La manera més ràpida de saber si una matriu es invertible es veient si el seu determinant no és zero.

## 3. Matriu Identitat

Per a qualsevol espai vectorial  $V$  existeix un operador identitat  $I$  que es definit com  $I|\psi\rangle = |\psi\rangle$ , aquest operador no fa cap canvi al vectors als quals opera.

## 4. Matriu Unitària

Un operador unitari es qualsevol operador que no altera la norma<sup>5</sup> o longitud dels vectors al quals es aplicat, per tant, una matriu es unitària si  $AA^\dagger = I$  Per convertir qualsevol operador en unitari, es divideixen els seus component entre la longitud o norma del vector del operador.

## 5. Matrius Hermitianes

Una matriu Hermitiana compleix que:

$$\langle\psi|(A|\varphi\rangle) = (A^\dagger\langle\psi|)|\varphi\rangle$$

On  $|\psi\rangle$  i  $|\varphi\rangle$ , són dos vectors del espai en el qual actua  $A$ . Si apliquem el producte interior a aquesta equació tenim que:  $A = A^\dagger$ .

<sup>3</sup>Computació Quàntica i Informació Quàntica.

<sup>4</sup>Una matriu quadrada és una matriu amb dimensions  $n \times n$ , on  $n \in \mathbb{N}$ .

<sup>5</sup>El concepte generalitzat de la longitud d'un vector. Usualment parlaré de la norma  $\ell_2$ , esmentada com  $\|\cdot\|_2$ , i definida com  $\| |\psi\rangle \| = \sqrt{\psi_1^2 + \dots + \psi_n^2}$ .

## 1.3 Producte interior i producte exterior

### 1.1 Producte interior

Un vector dual  $\langle\psi|$  i un vector  $|\varphi\rangle$  combinats formen el producte interior  $\langle\psi|\varphi\rangle$ , el qual efectua una operació que agafa els dos vectors com a input i produeix un nombre complex com a output:

$$\langle a|b\rangle = a_1b_1 + a_2b_2 + \dots + a_{n-1}b_{n-1} + a_nb_n = z$$

Amb  $z, a_i, b_i \in \mathbb{C}$ .

Aquest producte per dos vectors en  $\mathbb{R}^2$  també es pot escriure com a:

$$\langle a|b\rangle = \|a\|_2 \cdot \|b\|_2 \cos \theta \quad (1.1)$$

Amb  $\|\cdot\|_2$  sent la norma  $\ell^2$  i  $\theta$  sent l'angle entre els vectors  $|a\rangle$  i  $|b\rangle$ . Com he dit l'equació (1.1) és equivalent al producte interior, no obstant, segons el que he vist, no es usada àmpliament ja que interpretar  $\theta$  com un angle entre vectors de dimensions altes no té molt de sentit. En contraposició, he vist aquest producte presentat en la seva interpretació geomètrica<sup>6</sup> com el producte entre un vector fila i un vector columna:

$$\langle a|b\rangle = \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Ja he definit la norma  $\ell_2$  com l'arrel quadrada de la suma del elements d'un vector al quadrat:

$$\|a\|_2 = \sqrt{\sum_i |a_i|^2}$$

---

<sup>6</sup>Els detalls exactes de l'interpretació geomètrica estan fora del domini d'aquest treball, malgrat que m'agradaria molt parlar sobre el tema.

No obstant, la definició més comú es basa en el producte interior. Com es pot veure el producte interior d'un vector per ell mateix es la suma dels seus components al quadrat:

$$\langle a|a \rangle = a_1 a_1 + \cdots + a_n a_n = a_1^2 + \cdots + a_n^2 = \sum_i |a_i|^2$$

Per tant, la norma pot ser definida com l'arrel quadrada del producte interior d'un vector:

$$\| |a\rangle \|_2 = \sqrt{\langle a|a \rangle} \quad (1.2)$$

Quan la norma es aplicada a un vector bidimensional, es pot veure que es lo mateix que la longitud Euclidiana d'aquell vector, això és perquè realment son el mateixos conceptes, tot i això, la norma és el concepte de longitud però generalitzat a vectors de dimensions altes.

Pel el que jo entenc algunes propietats de la longitud d'un vector bidimensional no es mantenen amb la norma d'un vector que te més de 2 dimensions. En altres paraules, la norma es comporta en certes maneres com la distancia des de d'origen (que es la definició de la longitud), per tant, no són exactament lo mateix. A més d'això, hi han diferents tipus de normes que s'utilitzen en diversos escenaris. Aquesta es la raó per la qual en refereixo a la norma, com la norma  $\ell^2$ . A aquesta norma també se li diu norma Euclidiana [12].

### 1.1.1 Propietats del producte interior

Les propietats bàsiques del producte interior són les següents:

1. És lineal en el segon argument  $(z_1 \langle a| + z_2 \langle c|) |b\rangle = z_1 \langle a|b\rangle + z_2 \langle c|b\rangle$
2. Té simetria en el conjugat  $\langle a|b\rangle = (\langle b|a\rangle)^*$
3.  $\langle a|a\rangle$  es no-negativa i real, excepte en el cas de  $\langle a|a\rangle = 0 \Leftrightarrow |a\rangle = 0$

## 1.2 Vectors ortonormals i ortogonals

A partir del concepte de norma sorgeixen els conceptes de un par de vectors ortonormals i un par de vectors ortogonals. Mirant l'equació (1.2) podem veure que si el producte interior del vector és 1, la norma del mateix vector també és 1. Un vector que té norma 1, és un vector unitari.

Dos vectors diferents de zero són ortogonals si el seu producte interior és zero. Si aquests vectors són bidimensionals, a partir de l'equació (1.1) podem veure que el cosinus de l'angle que fan entre ells és zero i per tant són perpendiculars entre ells:

Per  $|a\rangle$  i  $|b\rangle \neq 0$  :

$$\text{Si } \langle a|b\rangle = 0 \text{ tenim que: } \| |a\rangle \|_2 \cdot \| |b\rangle \|_2 \cos \theta = 0$$

Perquè  $|a\rangle$  i  $|b\rangle$  no son zero, les seves normes no són zero.

Per tant el terme que falta  $\cos \theta$  és igual a zero.

D'aquesta manera, l'angle  $\theta$  ha de ser  $\frac{\pi}{2}$ .

No obstant, pensar que la perpendicularitat i la ortogonalitat són els mateixos conceptes es un error, degut a que, el que siguin només es veritat per els vectors bidimensionals. Perquè com en el cas de la norma i la longitud, la ortogonalitat és el concepte de perpendicularitat generalitzat.

Quan barregem els conceptes de vector unitari i vectors ortogonals arribem a la ortonormalitat [9]. Un parell de vectors que no són zero, són ortogonals quan els dos són unitaris i també són ortogonals entre ells:

$$|a\rangle \text{ and } |b\rangle \text{ son ortonormals si: } \begin{cases} \langle a|b\rangle = 0 \\ \langle a|a\rangle = 1 \\ \langle b|b\rangle = 1 \end{cases}$$

Els vectors ortonormals són importants, àmpliament utilitzats tant en computació quàntica com en mecànica quàntica perquè serveixen per crear bases vectorials molt útils.

Una altra cosa que remarcar i no he dit es que al dir un par de vectors, aquest parell també pot ser un conjunt de vectors. Si un conjunt té tots els vectors unitaris i ortogonals entre tots ells, és un conjunt de vectors ortonormals. El conjunt de vectors  $B = \{|\beta_1\rangle, |\beta_2\rangle, \dots, |\beta_{n-1}\rangle, |\beta_n\rangle\}$  és ortonormal si  $\langle\beta_i|\beta_j\rangle = \delta_{ij} \forall i, j$  [9] on  $\delta_{ij}$  és el Kronecker delta, definit com:

$$\delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

### 1.3 Producte exterior

El producte exterior és una funció (expressada com  $|a\rangle\langle b|$ , amb  $|a\rangle$  i  $|b\rangle$  sent vectors) que agafa dos vectors i produeix un operador lineal com output. Al contrari que el producte interior, no hi ha un producte equiparable en les matemàtiques ensenyades al institut, i és una mira difícil d'entendre perquè agafa dos vectors que poden ser d'un espai diferent com input. És definit com:

Pels vectors  $|v\rangle$  i  $|v'\rangle$  amb dimensions  $m$  i el vector  $|w\rangle$  de dimensió  $n$ . El producte exterior és l'aplicació lineal  $A$  de dimensions  $m \times n$  en l'espai  $\text{Mat}_{m \times n}$ :

$$|v\rangle\langle w| = A \text{ with } A \in \text{Mat}_{m \times n}.$$

Amb la seva acció definida per:

$$(|v\rangle\langle w|)|v'\rangle \equiv |w\rangle\langle v|v'\rangle = \langle v|v'\rangle|w\rangle \quad (1.3)$$

A partir de l'equació (1.3) l'utilitat i significat del producte es bastant complicada d'entendre, per tant exposaré la manera de computar-lo per clarificar com funciona. Per dos vectors  $|a\rangle$  i  $|b\rangle$  de dimensions  $m$  i  $n$  respectivament, el seu

producte interior es computa multiplicant cada element de  $|a\rangle$  per cada element de  $|b\rangle$  formant una matriu amb mida  $m \times n$ :

$$|a\rangle \langle b| = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m b_1 & a_m b_2 & \cdots & a_m b_n \end{bmatrix}$$

L'utilitat d'aquest producte és mostrara més endavant.

## 1.4 Producte tensorial

L'últim producte que mencionaré és el tensorial, representat per el símbol  $\otimes$ . Aquest producte s'utilitza per crear espais vectorials més grans combinant espais vectorials més petits. La definició formal és bastant complicada, per tant en centraré en explicar la manera amb la qual es computa utilitzant la representació matricial d'aquest producte, anomenada el producte de Kronecker.

Per una matriu  $m \times n$ ,  $A$ , i una  $p \times q$  matriu  $B$ , el seu producte de Kronecker

[13] és la matriu de mida  $pm \times qn$ :

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix} \\
 &= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}
 \end{aligned}$$

Cal tenir en compte que  $a_{ij}B$  es una multiplicació escalar per una matriu, amb  $a_{ij}$  sent l'escalar i  $B$  sent la matriu.

Aquí hi ha un exemple més il·lustratiu amb dues matrius de mida  $2 \times 2$ , es pot veure que cada element de la primera matriu es multiplica per la segona matriu:

$$\begin{aligned}
 \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} &= \begin{bmatrix} 1 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix} \\
 &= \begin{bmatrix} 1 \times 0 & 1 \times 5 & 2 \times 0 & 2 \times 5 \\ 1 \times 6 & 1 \times 7 & 2 \times 6 & 2 \times 7 \\ 3 \times 0 & 3 \times 5 & 4 \times 0 & 4 \times 5 \\ 3 \times 6 & 3 \times 7 & 4 \times 6 & 4 \times 7 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}
 \end{aligned}$$

El producte de Kronecker també funciona amb vectors de la mateixa manera, però amb una multiplicació de escalar per vector.

Per els vectors  $|\psi\rangle$  i  $|\varphi\rangle$  de dimensions  $n$  i  $m$  respectivament:

$$|\psi\rangle \otimes |\varphi\rangle = \begin{bmatrix} \psi_1 |\varphi\rangle \\ \psi_2 |\varphi\rangle \\ \vdots \\ \psi_m |\varphi\rangle \end{bmatrix} = \begin{bmatrix} \psi_1 \varphi_1 \\ \psi_1 \varphi_2 \\ \vdots \\ \psi_1 \varphi_m \\ \vdots \\ \vdots \\ \psi_n \varphi_1 \\ \psi_n \varphi_2 \\ \vdots \\ \psi_n \varphi_m \end{bmatrix}$$

S'ha de tenir en compte que el producte de Kronecker també es pot fer entre un vector i una matriu, o viceversa, no obstant, no és molt comú fer-ho.

## 1.1 Propietats del producte tensorial

Les propietats bàsiques del producte tensorial són les següents [14, 15]:

### 1. Associativitat:

$$A \otimes (B + C) = A \otimes B + A \otimes C$$

$$(zA) \otimes B = A \otimes (zB) = z(A \otimes B)$$

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

$$A \otimes 0 = 0 \otimes A = 0$$

### 2. No-commutativitat <sup>7</sup>:

$$A \otimes B \neq B \otimes A$$

---

<sup>7</sup>Una cosa a notar és que  $A \otimes B$  i  $B \otimes A$  són permutativament equivalents:  
 $\exists P, Q \Rightarrow A \otimes B = P(B \otimes A)Q$  on  $P$  i  $Q$  són matrius permutatives.



## 1.5 Traça

La traça d'una matriu és tal sols la suma dels seus elements en la diagonal principal, la diagonal que va d'adalt a baix i d'esquerra a dreta.

Aquí hi ha una matriu  $A$  amb la seva diagonal principal marcada:

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

I la seva traça, representada per  $\text{Tr}[A]$  és:

$$\text{Tr}[A] = 1 + 1 + 1 = 3$$

Més formalment, la traça d'una matriu quadrada  $n$ -dimensional és:

$$\text{Tr}[A] = \sum_{i=1}^n a_{ii} = a_{11} + a_{22} + \cdots + a_{nn}$$

La traça d'una matriu té les propietats següents:

1. Operador lineal:

Degut a que la traça és un aplicació lienal, es compleix que:

$\text{Tr}[A + B] = \text{Tr}[A] + \text{Tr}[B]$  i  $\text{Tr}[zA] = z \text{Tr}[A]$ , per a totes les matrius quadrades  $A$  i  $B$ , i tots els escalars  $z$ .

2. Traça d'un producte tensorial:

$$\text{Tr}[A \otimes B] = \text{Tr}[A] \text{Tr}[B]$$

3. La transposada té la mateixa traça:

$$\text{Tr}[A] = \text{Tr}[A^T]$$

4. La traça d'un producte és cíclica:

Per una matriu  $A$  amb mida  $m \times n$  i una matriu  $B$  de la mateixa mida:

$$\text{Tr}[AB] = \text{Tr}[BA]$$

Una altre manera molt útil d'avaluar la traça d'un operador és a través del procediment de Gram-Schmidt<sup>8</sup> i el producte exterior. Utilitzant Gram-Schmidt per representar el vector unitat  $|\psi\rangle$  en una base ortonormal  $|i\rangle$  que inclou  $|\psi\rangle$  com el seu primer element, és compleix que:

$$\text{Tr}[A |\psi\rangle \langle\psi|] = \sum_i \langle i| A |\psi\rangle \langle\psi|i\rangle = \langle\psi| A |\psi\rangle$$

---

<sup>8</sup>Mirar [A.1](#) per una definició del procediment de Gram-Schmidt<sup>9</sup>.

# Capítol 2

## Computació quàntica

Després de bastant teoria matemàtica, ha arribat el moment de parlar sobre mecànica quàntica, en aquest capítol introduiré alguns conceptes bàsics sobre Informació Quàntica i Computació Quàntica (IQ i CQ).

La mecànica quàntica és un marc matemàtic o un conjunt de teories utilitzades per poder explicar les propietats físiques dels àtoms, molècules i partícules subatòmiques. És un marc que engloba tota la física quàntica, incloent la teoria d'informació quàntica. La manera correcta de definir la computació quàntica és a través de postulats formals de la mecànica quàntica, perquè amb ells les afirmacions que es fan en computació quàntica no semblen venir d'enlloc [16]. No obstant, per no complicar més aquesta secció, faré el millor possible per poder explicar els conceptes de la computació quàntica sense entrar en els conceptes més generals de la mecànica quàntica, a no ser que sigui estrictament necessari.

### 2.1 Estats quàntics i superposicions

Per descriure com evolucionen els sistemes físics a través del temps, es necessita representar els sistemes d'alguna manera. En computació quàntica és representen per estats quàntics, els quals són un tipus de distribucions de probabilitat que representen els possibles resultats d'una mesura on un sistema quàntic [17].

Imaginat que tens un boli, però que no saps de quin color és, no obstant, saps que pot ser vermell o blau. Per esbrinar de quin color és, pots provar a escriure amb el boli per veure el color de la tinta, o en altres paraules, fer una mesura del sistema. Saps que hi ha un 50% de probabilitat de que sigui vermell i un 50% de que sigui blau. En aquesta situació hipotètica tindries el teu sistema quàntic (el boli), una manera de mesurar-lo (escriure alguna cosa) i una llista amb els possibles resultats (50% vermell, 50% blau), només et falta una manera de representar-lo tot matemàticament, l'estat quàntic. Per tant, per què no intentem guardar la informació que sabem del boli en un vector?

Si posem cada probabilitat de treure un resultat en un vector tenim que:

$$\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

On la primera entrada és la possibilitat de que el boli sigui vermell i la segona entrada de que sigui blau, per fer-ho més senzill aquí està en colors:

$$\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

Cal remarcar que aquest vector està normalitzat amb la norma  $\ell_1$ , definida com la suma dels components d'un vector<sup>1</sup>, en altres paraules la norma  $\ell_1$  d'aquest vector és 1.

Llavors, hem d'escollir una operació matemàtica per poder extreure la informació del vector, com que el output ha de ser un número, podem provar a utilitzar un producte interior. Però primer s'ha de representar el vector com una combinació lineal de les seves bases:

$$0.5 |0\rangle + 0.5 |1\rangle = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

---

<sup>1</sup>Amb  $|a\rangle$  sent un vector, la norma  $\ell_1$ , denotada per  $\|\cdot\|_1$  és  $\| |a\rangle \|_1 = \sum_i a_i$ .

On  $|0\rangle$  és el vector  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ , i,  $|1\rangle$  és el vector  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . Per tant, per trobar la probabilitat, s'agafa el producte interior del vector amb la base corresponent a la probabilitat, com a continuació:

$$\langle 0|v\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = 0.5$$

$$\langle 1|v\rangle = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = 0.5 \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = 0.5$$

Aquest procediment és bastant senzill, com a un altre exemple, per representar un boli amb 6 colors possibles amb una possibilitat aleatòria d'escriure amb un color del 6 possibles, l'estat d'aquest boli és<sup>2</sup>:

$$|w\rangle = \begin{bmatrix} 0.25 \\ 0.3 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.05 \end{bmatrix}$$

Ara veure la probabilitat de per exemple treure el color verd, utilitzant la tercera base, la que correspon al color verd:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.25 \\ 0.3 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.05 \end{bmatrix} = 0.1$$

Deixant els bolis a una banda, ara els podem substituir per un sistema físic quàntic, com per exemple un fotó. Els fotons tenen certes propietats que poden ser

---

<sup>2</sup>Le posat colors a els elements per claredat.

mesurades, com la seva polarització<sup>3</sup>. Al mirar als fotons com ones en que oscil·len en el camp electromagnètic, la polarització és l'orientació geomètrica de l'ona. La polarització pot ser interpretada com un angle respecte a la direcció de propagació.

Al definir les bases del estat de polarització com vertical i horitzontal, denotades per els vectors  $|\rightarrow\rangle$  i  $|\uparrow\rangle$ , respectivament. Podem definir un estat de superposició entre les bases, representat com  $|\nearrow\rangle$  [18]:

$$|\nearrow\rangle = \alpha |\rightarrow\rangle + \beta |\uparrow\rangle \quad (2.1)$$

On  $\alpha$  i  $\beta$  són números complexos. Un estat en superposició es simplement un estat on l'angle de polarització no és 0 ni  $\frac{\pi}{2}$ . No ens hem de que preocupar de la descripció matemàtica exacte de la polarització dels fotons al parlar de computació quàntica perquè el que importa és l'informació que porten aquests estat, no la física dels sistemes que representen. Aquesta informació s'ha d'agafar mitjançant mesures, com amb els bolis. Aquestes mesures en el cas de la polarització dels fotons seria passar-los per diversos filtres de polarització, els quals deixen passar el fotó o l'absorbeixen, tot això d'una manera probabilística, es a dir, depenent de quin sigui l'estat tenen certa possibilitat de ser absorbits o no.

Per clarificar, el filtre lo que fa es col·lapsar el fotó en els dos possibles estats de polarització, l'estat en el quan el filtre es orientat o l'estat perpendicular a aquest. Si col·lapsa en l'estat del filtre el fotó passa pel el filtre, en cas de col·lapsar en l'altre estat, es absorbit. La manera en la que els fotons col·lapsen és probabilística, si agafen un filtre que està orientat horitzontalment respecte al fotons que li arriben, un fotó orientat horitzontalment té un 100% de possibilitats de poder passar, mentre que un fotó polaritzat verticalment té un 0% de probabilitat de poder passar. I si un fotó està polaritzat en un angle just entre vertical i horitzontal, es a dir a  $\frac{\pi}{4}$ , tindrà un 50% de possibilitats de passar i un 50% de no poder-hi.

---

<sup>3</sup>Concretament, són una propietat que les ones transversals tenen, el tipus d'ona de les ones electromagnètiques, que són els fotons en realitat.

Aquesta és la manera amb la qual els sistemes quàntics es comporten, a través de la probabilitat, on els estats que els representen tenen la informació sobre quines són aquestes possibilitats. No obstant, la polarització dels fotons són un sistema físic concret, quan es parla de computació quàntica és millor treballar amb conceptes més generals per poder expressar tants algoritmes com sigui possible i poder implementar aquests algoritmes en tants ordinadors quàntics com sigui possible. Per saqueta raó, en la branca de la informació quàntica i la computació quàntica es treballa amb qubits, en comptes de diversos sistemes físics.

## 2.2 Qubits i operacions quàntiques

Els ordinadors modern representen l'informació a través de cadenes de zeros i uns, anomenades bits. Tot; des de imatges a lletres o instruccions electròniques. Per exemple, la lletra t és representada per la cadena de bits 01110100, codificat a través de codi binari. Tot el que fas en un ordinador es codifica i representa en codi binari.

Degut a que estem molt acostumats a codi binari, en el camp de la computació quàntica també s'utilitza, no obstant, en comptes de bits s'utilitzen qubits. Un qubit es l'anàleg d'un bit, en altres paraules, és la unitat mínima d'informació utilitzada pels ordinadors quàntics. En els qubits podem aplicar propietats quàntiques com la superposició o l'entrellaçament<sup>4</sup>. Si un bit pot estar en l'estat 0 o en l'estat 1, un qubit pot estar en una combinació d'aquests estats, en un estat enmig del 1 i del 0. És una combinació lineal dels vectors que representen aquests estats,  $|0\rangle$  i  $|1\rangle$ :

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

On  $\alpha$  i  $\beta$  són nombres complexos i  $|\psi\rangle$  és un vector en espai de Hilbert bidimensional. Els vectors  $|0\rangle$  i  $|1\rangle$  són anomenats els vectors de la base computacionals,

---

<sup>4</sup>Ja he parlat de la primera amb la polarització del fotons, del entrellaçament parlaré més endavant.

representats com:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Per tant el vector  $|\psi\rangle$  és:

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

Aquest vector és un estat quàntic vàlid per representar un qubit, anomenat *statevector* o vector d'estat. No obstant, hi ha un important factor a tenir en compte. El vector ha d'estar normalitzat amb la norma  $\ell_2$ , per tant, els nombres  $\alpha$  i  $\beta$  no poden ser qualsevol nombre, necessiten ser els coeficients que formin un vector amb una norma de 1.

$$\| |\psi\rangle \| = 1$$

Llavors:

$$\begin{aligned} \|\psi\| &= \sqrt{|\alpha|^2 + |\beta|^2} = 1 \\ \Rightarrow |\alpha|^2 + |\beta|^2 &= 1 \end{aligned}$$

Definir un qubit com una 'combinació lineal dels estats fonamentals' no és de molta ajuda, per això, elaboraré més sobre aquesta definició: Una cadena de  $n$ -bit pot només representar una única combinació d'uns i zeros, mentre que una cadena de  $n$ -qubits representa una combinació de totes les possibles combinacions. En el cas d'un qubit, aquest és una combinació dels possibles estats  $|0\rangle$  i  $|1\rangle$ . Considera un qubit com una barreja dels estats possibles, amb cada coeficient de la combinació lineal sent el nombre que indica quant d'un estat forma part de la barreja.

Una cosa molt interessant passa quan s'augmenta el nombre qubits, la «quantitat d'informació» creix exponencialment. Per una cadena de  $n$  qubits la «quantitat d'informació» que té, en altres paraules la quantitat de nombres que representa és  $2^n$ , on aquests nombres són els coeficients de la combinació line-



al. Això es perquè quan afegeixes un qubit el nombre de combinacions possibles creix exponencialment, per tant es necessiten més coeficients per representar aquestes combinacions noves en la combinació lineal. Els qubits necessiten molta més informació per poder representar-los<sup>5</sup>, no com els bits que al ser només una combinació és necessita només saber quina combinació és. No passa res si això no s'entén perfectament, lo important és saber que es necessiten  $2^n$  números complexos<sup>6</sup> per representar  $n$ -qubits i que es necessiten  $n$  números binaris per representar  $n$ -bits.

Per il·lustrar tot això, 2 qubits es representen amb el *statevector*:

$$\begin{aligned}
 |\psi\rangle &= \alpha_1 |00\rangle + \alpha_2 |01\rangle + \alpha_3 |10\rangle + \alpha_4 |11\rangle \\
 &= \alpha_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \alpha_3 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \alpha_4 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix}
 \end{aligned}$$

Cal remarcar que els vectors de la base computacional serien les columnes d'una matriu identitat amb dimensions  $2^n \times 2^n$ , on  $n$  és el nombre de qubits.

Per poder representar informació amb qubits simplement es codifica aquesta informació en els qubits, això es pot fer per exemple mitjançant codi binari: Els números 0, 1, 2 i 3 són els bits 00, 01, 10 i 11 respectivament, per tant es poden representar amb dos qubits en els estats  $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$ ,  $|11\rangle$ , respectivament.

## 2.1 Representació geomètrica d'un qubit

Una aspecte que sempre m'agrada del qubits és la seva representació geomètrica, la esfera de Bloch 2.1. Si agafem una esfera unitària<sup>7</sup> que té els seus pol nord i sur definits per els vectors  $|0\rangle$  i  $|1\rangle$ , respectivament. Cada punt de la seva superfície és un estat quàntic vàlid on les seves bases computacionals són  $|0\rangle$  i  $|1\rangle$ .

<sup>5</sup>Informació que es manifesta amb els coeficients de la combinació lineal

<sup>6</sup>Són complexos degut a que els coeficients de la combinació lineal són números complexos.

<sup>7</sup>Una esfera que té radi 1 i que per tant qualsevol punt en la seva superfície correspon a un vector en  $\mathbb{R}^3$  unitari.

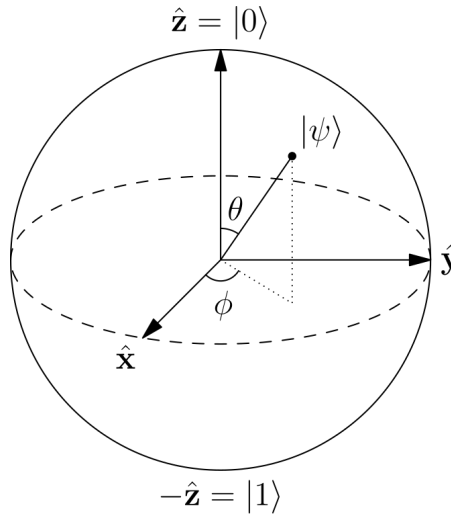


FIGURA 2.1: Esfera de Bloch, on es representa un estat arbitrari  $|\psi\rangle$  amb els vectors  $\hat{x}, \hat{y}, \hat{z}$  representat els eixos ortonormals de la esfera de radi 1.

## 2.2 Operacions per a només un qubit

Una vegada es té la informació representada, estaria bé poder operar amb aquella informació, aquest es justament lo que fa que els ordinadors siguin ordinadors, poder operar amb la informació. En computació quàntica els qubits al poder ser representats amb vectors que tenen els seus coeficients són operats per les anomenades portes lògiques quàntiques, que són matrius. Per exemple, si es vol passar de tindre un qubit en l'estat  $|0\rangle$  al estat  $|1\rangle$ , s'utilitza la porta lògica  $X$  representada a continuació:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Podem veure com fa l'acció al multiplicar la matriu per el vector:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Que en notació de Dirac s'expressaria com:

$$X |0\rangle = |1\rangle$$

D'una manera més general:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 \times \alpha + 1 \times \beta \\ 1 \times \alpha + 0 \times \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

Es pot veure que aquesta matriu lo que fa és donar la volta als coeficients d'un vector, per tant:

$$X |0\rangle = |1\rangle \text{ i } X |1\rangle = |0\rangle$$

Aquesta porta lògica forma part d'un grup important, les matrius de Pauli. Hi han 3 d'aquestes la X, la Y i la Z, usualment representades per  $X$ ,  $Y$ ,  $Z$  o per  $\sigma_x$ ,  $\sigma_y$ ,  $\sigma_z$ . Aquestes matrius són les següents:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & i \\ -i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Aquestes matrius són molt important en la mecànica quàntica<sup>8</sup> i són utilitzades àmpliament per descompondre i com a portes lògiques quàntiques.

A partir d'elles podem elaborar matrius que facin una rotació d'un angle  $\theta$  qualsevol en un del eixos de la representació geomètrica d'un qubit 2.1:

$$R_x(\theta) = e^{-i\theta X/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} X = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \quad (2.2)$$

$$R_y(\theta) = e^{-i\theta Y/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Y = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \quad (2.3)$$

$$R_z(\theta) = e^{-i\theta Z/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Z = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \quad (2.4)$$

Per exemple la matriu  $R_y(\cdot)$  (Eq.2.3) correspon a una rotació en el eix  $\hat{y}$  de la esfera de la figura 2.1.

<sup>8</sup>Al ser Hermitianes són observables, concretament ho són dels que corresponen al spin d'una partícula amb spin  $\frac{1}{2}$  bàsicament estan relacionades amb els operadors del moment angular.

Aquestes operacions poden resultar en superposicions si es fan rotacions amb certs angles. Però hi ha una porta lògica especial per poder fer una rotació que resulta en una superposició uniforme. Es a dir una superposició que tingui les mateixes probabilitats de resultar en  $|0\rangle$  o  $|1\rangle$ . Aquesta és la porta de Hadamard, denotada per  $H$ :

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.5)$$

Podem comprovar que és una superposició uniforme al aplicar-la al estat  $|0\rangle$ :

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

L'estat resultant és un estat especial que s'escriu com  $|+\rangle$ <sup>9</sup>. La probabilitat de que un estat col·lapsi en una determinada base és el coeficient de la seva base elevat al quadrat. Com que l'estat és:

$$|+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Al elevar al quadrat qualsevol dels coeficients es pot veure que dona  $\frac{1}{2}$ :

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}$$

Llavors tenim que la probabilitat per obtenir ambdós estats és la mateixa, es a dir que si mesurem l'estat  $|+\rangle$  hi ha la mateixa probabilitat de que surti  $|0\rangle$  o  $|1\rangle$ . La porta lògica de Hadamard és molt important ja que s'utilitza per crear distribucions uniformes, ja sigui en un qubit o en diversos<sup>10</sup>.

<sup>9</sup>Un altre estat similar és  $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ , quan la matriu  $H$  s'aplica al estat  $|1\rangle$ .

<sup>10</sup>S'aplica aquesta operació a cada qubit del sistema.

Altres operacions importants de només un qubit són les portes  $S$  i  $T$ :

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

## 2.3 Circuit quàntics

Aquestes operacions usualment es representen a través de circuits quàntics. Són representacions gràfiques que indiquen de quines operacions s'apliquen a quins qubits i en quin ordre<sup>11</sup>.

La forma de representar una porta  $H$  aplicada a un qubit és amb el circuit quàntic:

$$|0\rangle \text{ --- } \boxed{H} \text{ ---}$$

El qubit es representat per la línia que comença amb  $|0\rangle$ , amb  $|0\rangle$  sent el seu estat inicial. Aquests diagrames es llegeixen d'esquerra a dreta, la mateixa forma en la qual s'apliquen les operacions.

Un qubit en el qual se li aplica una porta  $H$  i després una porta  $X$ , es representat com:

$$|0\rangle \text{ --- } \boxed{H} \text{ --- } \boxed{X} \text{ ---}$$

Múltiples qubits son simplement més línies, aquí estan representat dos qubits amb portes  $X$  aplicades a cada un:

$$|0\rangle \text{ --- } \boxed{X} \text{ ---}$$

$$|0\rangle \text{ --- } \boxed{X} \text{ ---}$$

Amb múltiples qubits també hi han un ordre en el qual les portes s'han d'aplicar, un circuit en el qual es representa que s'aplica una porta Hadamard al primer, al principi, i una rotació<sup>12</sup> en l'eix  $x$  en el segon qubit a continuació, seria:

$$|0\rangle \text{ --- } \boxed{H} \text{ ---}$$

$$|0\rangle \text{ --- } \boxed{R_x(\theta)} \text{ ---}$$

<sup>11</sup>A mi em semblen semblats a les partitures musicals.

<sup>12</sup>D'un angle  $\theta$ .

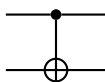
## 2.4 Operacions per a múltiples qubits

Lo realment interessant es quan s'apliquen portes a diversos qubits, perquè d'aquesta manera és pot arribar a tindre qubits entrellaçats. La porta més útil per entrellaçar qubits és la CNOT o *Controlled NOT*:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (2.6)$$

És bàsicament una porta  $X$ <sup>13</sup> que està controlada per un altre qubit, si l'altre qubit és  $|1\rangle$  s'aplica la porta  $X$  al altre qubit. Però si l'altre qubit està en superposició, per exemple en  $|+\rangle$ , aquesta superposició es passa també al qubit controlat, i al mesurar el qubit, la probabilitat de que s'apliqui la porta  $X$  és la probabilitat de mesurar l'estat  $|1\rangle$ . Es considera que aquests qubits estan entrellaçats, una mesura a un d'ells afecta a la mesura del altre. El qubit al qual se li aplica la porta  $X$  es diu *target* i el qubit sobre el qual depèn el *target* és el *control*.

Aquesta porta representada en un circuit s'escriu com:



On el qubit *control* és el primer i on el segon és el *target*, el qubit que té el símbol  $\oplus$ <sup>14</sup>.

<sup>13</sup>També anomenada porta NOT degut al paral·lisme que es fa amb la porta lògica del ordinadors clàssics NOT [19] que simplement inverteix els bits d'1 a 0 (i viceversa), igual que  $X$  que inverteix els qubits  $|1\rangle$  a  $|0\rangle$  i viceversa.

<sup>14</sup>A vegades escriure els circuits quàntics sense especificar l'estat inicial degut a que no és necessari.

### 2.4.1 Entrellaçament quàntic

L'exemple més senzill d'un entrellaçament quàntic en la computació quàntica són els parells de Bell, que es creen al aplicar a dos qubits una porta  $H$  al primer i després una porta CNOT als dos creant l'estat:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

Es pot veure que només hi han dos estats possibles  $|00\rangle$  i  $|11\rangle$  que tenen la mateixa probabilitat associada<sup>15</sup>. S'afecten l'un al altre en el sentit que quan es mesura només un dels qubits i dona per exemple  $|1\rangle$ , al mesurar l'altre també dona  $|1\rangle$ , d'aquesta manera acabant amb l'estat  $|11\rangle$ . En altres paraules, la mesura d'una part del sistema determina el resultat d'una mesura en una altra part del sistema.

Matemàticament un sistema quàntic, i.e. un conjunt de qubits, està entrellaçant quan aquest sistema no es pot descriure amb un producte tensorial de les parts. Per exemple estat  $|00\rangle$  es pot escriure com  $|0\rangle \otimes |0\rangle$ , mentre que l'estat  $\frac{|00\rangle + |11\rangle}{\sqrt{2}}$ , no. Per tant el primer no és sistema amb qubits entrellaçats i el segon si ho és.

A partir del entrellaçament i la superposició és com els ordinadors quàntics arribem a tenir avantatges en complexitat sobre els ordinadors clàssics, per més informació sobre els avantatges que presenten els algorismes quàntics en certes tasques veure l'apèndix D.

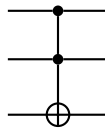
### 2.4.2 Operacions controlades

A part de la porta CNOT existeixen diverses portes quàntiques controlades. Realment és pot controlar qualsevol porta, en altres paraules, si l'estat del qubit *control* és  $|0\rangle$ , s'aplica qualsevol porta al qubit *target*. Fins i tot podem haver-hi diversos qubits *control* i *target*.

Per exemple existeix la porta Toffoli:

---

<sup>15</sup>Això es pot veure al elevar al quadrat els coeficients del dos, que donen  $\frac{1}{2}$ .



Que en la seva forma matricial és:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Aquesta porta aplica una porta  $X$  al últim qubit en cas de que els dos primers siguin  $|0\rangle$ .

Tornant a dos qubits, al veure la matriu per la porta CNOT, es pot apreciar que està composta per una matriu identitat i una porta  $X$ <sup>16</sup>:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

També al veure la porta  $Z$  controlada (CZ), es pot apreciar el mateix patró:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

---

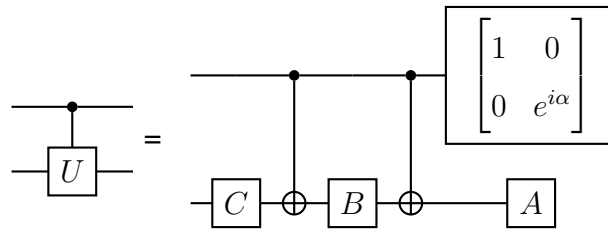
<sup>16</sup>La matriu identitat es troba a la cantonada superior esquerra, mentre que la porta  $X$  es troba al altre extrem.



Amb la matriu de la porta  $Z$  a la cantonada inferior dreta. Aquesta porta en un circuit quàntic es representa de la següent manera:



No obstant, una operació controlada de qualsevol matriu unitària  $U$  es forma a través del següent circuit:



On  $U, \alpha, A, B$  i  $C$  son tals que  $U = e^{i\alpha} A X B X C$  i  $ABC = I$ .

## 2.3 Mesurament quàntic

Com ja s'ha esmentat a la secció 2.2 al elevar al quadrat el coeficient d'un estat base que forma part d'un estat, s'obté la probabilitat d'obtenir aquest estat base quan es mesura.

Aquesta és la forma més simple de poder predir el mesurament d'un estat quàntic. però hi han altres mètodes que a vegades resulten més útils.

Els mesuraments quàntics també es ponen pensar com un conjunt d'operadors de mesura  $M_m$ . On la probabilitat d'obtindre l'estat  $m$  associat a un operador  $M_m$  al mesurar un  $|\psi\rangle$ :

$$\text{prob}(m) = \langle \psi | M_m^\dagger M_m | \psi \rangle \quad (2.7)$$

On l'estat després de la mesura, és:

$$\frac{M_m |\psi\rangle}{\sqrt{\langle \psi | M_m^\dagger M_m | \psi \rangle}}$$

Els operadors  $M_m$  han de complir que la suma de les seves probabilitats sigui 1:

$$1 = \sum_m \text{prob}(m) = \sum_m \langle \psi | M_m^\dagger M_m | \psi \rangle$$

La diferencia entre aquesta manera de fer les mesures i elevar els coeficients al quadrat, és que l'equació 2.7 és una forma més general, on en comptes de mesurar en la base computacional, és pot mesurar en qualsevol base. A més a més, utilitzant aquest mètode no es necessari saber la composició<sup>17</sup> del estat que vols mesurar.

Per mesurar en la base computacional un *statevector* s'utilitzen operadors de mesura derivats de la base computacional amb productes exteriors. Per crear l'aplicació lineal  $M_i$  associat a la base computacional  $|i\rangle$  s'agafa el producte exterior de la base:

$$M_i = |i\rangle \langle i|$$

Per tant la probabilitat que la mesura del estat  $|\psi\rangle$  resulti en  $|0\rangle$ , és:

$$\text{prob}(|0\rangle) = \langle \psi | 0 \rangle \langle 0 |^\dagger | 0 \rangle \langle 0 | \psi \rangle$$

Per  $|\psi\rangle = |0\rangle$  tenim que<sup>18</sup>:

$$\begin{aligned} \text{prob}(|0\rangle) &= \langle 0 | 0 \rangle \langle 0 |^\dagger | 0 \rangle \langle 0 | 0 \rangle \\ &= \langle 0 | 0 \rangle \langle 0 | 0 \rangle \langle 0 | 0 \rangle \\ &= \langle 0 | 0 \rangle \langle 0 | 0 \rangle \\ &= 1 \end{aligned}$$

El resultat té sentit degut a que si mesurem  $|0\rangle$  en la base  $|0\rangle$ , la probabilitat de provar l'estat hauria de ser 1.

<sup>17</sup>Els coeficients del estats base que descomponen el vector.

<sup>18</sup>Cal notar que  $|0\rangle \langle 0|^\dagger = |0\rangle \langle 0|$  i que  $|0\rangle$  és un vector unitari.

## 2.4 Matriu de densitat

Una serie de qubits es pot representar tant per un vector com per una matriu, anomenats *statevector* i *density matrix*, respectivament [8]. En aquesta secció aniré ràpidament sobre el concepte de la matriu de densitat i com les operacions que s'apliquen a un *statevector* poden ser aplicades a una *density matrix*. Després parlaré del mesuraments parcial d'un sistema, un concepte que és important per la part experimental del treball.

Una matriu densitat és la representació matemàtica d'un estat quàntic a partir de d'un matriu, es a dir, d'un operador. Aquesta representació serveix descriure sistemes quàntics que no són completament coneguts. Concretament aquests operadors són conjunts d'estats quàntics: Per un sistema que es descriu amb un conjunt d'estats  $\{|\psi_i\rangle\}$  amb cada element del conjunt tenint una probabilitat associada de  $p_i$ . La matriu densitat del sistema  $\rho$  és [20]:

$$\rho = \sum_i p_i |\psi_i\rangle \langle \psi_i|$$

La matriu densitat pot ser simplement  $|\psi\rangle \langle \psi|$  per un estat qualsevol  $|\psi\rangle$ , per exemple la matriu que representa  $|0\rangle$  és:

$$\rho = |0\rangle \langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

L'evolució d'un estat  $\rho$  que descriu un sistema quàntic, al igual que amb els vectors, s'efectua a partir d'operadors unitaris<sup>19</sup>, d'aquesta manera tenim que la evolució d'un operador densitat és:

$$\sum_i p_i U |\psi_i\rangle \langle \psi_i| U^\dagger = U \rho U^\dagger$$

Al igual que es fan mesures en *statevectors*, les podem fer amb les *density matrices*. Per operadors de mesura  $M_m$ , tenim que la probabilitat de tindre l'estat

<sup>19</sup>Recorda que han de preservar la norma del vector, i en el cas de les matrius la seva traça.

$m$  és:

$$\begin{aligned}\text{prob}(m) &= \langle \psi_i | M_m^\dagger M_m | \psi_i \rangle = \text{tr}(M_m^\dagger M_m | \psi_i \rangle \langle \psi_i |) \\ &= \text{tr}(M_m^\dagger M_m \rho)\end{aligned}$$

També tenim que l'estat  $|\psi_i\rangle$  després de la mesura  $m$  és [20]:

$$|\psi_i\rangle = \frac{M_m |\psi_i\rangle}{\sqrt{\langle \psi_i | M_m^\dagger M_m | \psi_i \rangle}} = \frac{M_m |\psi_i\rangle}{\sqrt{\text{tr}(M_m^\dagger M_m \rho)}} \quad (2.8)$$

La equació 2.8 es pot reescriure en termes d'una matriu de densitat després d'una mesura:

$$\begin{aligned}\rho_m &= \sum_i p_i \frac{M_m |\psi_i\rangle \langle \psi_i| M_m^\dagger}{\text{tr}(M_m^\dagger M_m \rho)} \\ &= \frac{M_m \rho M_m^\dagger}{\text{tr}(M_m^\dagger M_m \rho)}\end{aligned}$$

Perquè això sigui cert els operadors de mesura  $M_m$ , ha'n de satisfer:

$$\sum_m M_m^\dagger M_m = I$$

Per a tots els estats possibles  $m$ .

Per últim s'ha de recordar que les matrius densitat al igual que els vectors d'estat han de tenir certes característiques:

1. La traça de  $\rho$  ha d'equivaldre a 1.
2.  $\rho$  ha de ser un operador positiu<sup>20</sup>.

## 2.1 Matriu de densitat reduïda

Una aplicació important dels operadors de densitat és la descripció d'estats parcials amb l'operador de densitat reduït, i per tant la descripció dels mesuraments parcials.

<sup>20</sup>Un operador positiu  $A$  es aquell que  $\langle \psi | A | \psi \rangle \geq 0, \forall |\psi\rangle$ .

Un sistema físic compost per dos sistemes  $A$  i  $B$ , es descriu per una matriu de densitat  $\rho^{AB}$ , la matriu de densitat reduït del sistema  $A$  és:

$$\rho^A = \text{tr}_B(\rho^{AB}) \quad (2.9)$$

On  $\text{tr}_B$  és la traça parcial sobre el sistema  $B$ , que es defineix per l'equació següent [21]:

$$\text{tr}_B(|a_1\rangle\langle a_2| \otimes |b_1\rangle\langle b_2|) = |a_1\rangle\langle a_2| \text{tr}(|b_1\rangle\langle b_2|)$$

Amb  $|a_1\rangle$  i  $\langle a_2|$  sent estats vàlids pel sistema  $A$ , i  $|b_1\rangle$  i  $\langle b_2|$  sent-ho per  $B$ . El terme  $\text{tr}(|b_1\rangle\langle b_2|)$  s'omet quan els vectors del producte exterior son iguals i formen un operador de densitat vàlid, el qual té una traça de 1.

No obstant aquesta definició no es pot utilitzar quan no saps com representar la matriu de densitat  $\rho$  com a un producte tensorial, en el qual un dels termes és l'estat que es traça afora. En altres paraules, en la equació 2.9 si no es coneix  $\rho^A$  i  $\rho^B$  per  $\rho^{AB} = \rho^A \otimes \rho^B$ , aquesta equació no serveix de res.

Degut a això en el llibre de text *Quantum Computing: A Gentle Introduction* [22] els autors defineixen la traça parcial d'una altra manera més general, on només s'ha de saber les bases del sistemes  $A$  i  $B$  i un operador vàlid pel sistema  $AB$ . Per una matriu de densitat  $\rho^{AB}$  que representa el sistema  $A \otimes B$ , la traça parcial de  $\rho^{AB}$  sobre  $B$  és:

$$\text{tr}_B \rho^{AB} = \sum_i \langle \beta_i | \rho^{AB} | \beta_i \rangle$$

On el conjunt  $\{|\beta_i\rangle\}$  són les bases del sistema  $B$ . Les entrades de la matriu  $\text{tr}_B \rho^{AB}$  representades en termes de les bases  $|\alpha_i\rangle$  i  $|\beta_j\rangle$  del sistemes  $A$  i  $B$  respectivament, són:

$$(\text{tr} \rho^{AB})_{ij} = \sum_{k=0}^{M-1} \langle \alpha_i | \langle \beta_k | \rho^{AB} | \alpha_j \rangle | \beta_k \rangle$$

Amb la matriu sent:

$$\text{tr} \rho^{AB} = \sum_{i,j=0}^{N-1} \left( \sum_{k=0}^{M-1} \langle \alpha_i | \langle \beta_k | \rho^{AB} | \alpha_j \rangle | \beta_k \rangle \right) |\alpha_i\rangle \langle \alpha_j|$$

On  $N$  és la dimensió del sistema  $A$  i  $M$  és la dimensió del sistema  $B$ .

Es pot comprovar que aquestes definicions són correctes degut a que en els dos llibres utilitzen les seves definicions per tractar el mateix cas i obtenen el mateix resultat [23, 24].

### 2.1.1 Mesurament parcial

Es pot arribar a treure una mesura parcial<sup>21</sup> sobre un sistema de qubits amb la traça parcial i operadors de mesura. En el paper fet per Huang et.al. [5] es descriu un estat  $\rho$  després d'un mesurament parcial  $\Pi_A$  sobre un sistema  $A$  que perteneix a l'estat  $|\psi\rangle$  com:

$$\rho = \frac{\text{tr}_A(\Pi_A |\psi\rangle \langle\psi|)}{\text{tr}(\Pi_A \otimes I_{2^N-2^{N_A}} |\psi\rangle \langle\psi|)}$$

El sistema  $A$  està compost per  $N_A$  qubits per tant la resta del estat  $|\psi\rangle$  té  $N - N_A$ , on  $N$  és el nombre total de qubits del estat  $|\psi\rangle$ . D'aquesta manera  $\Pi_A \otimes I_{2^N-2^{N_A}}$  té  $2^N \times 2^N$  dimensions i pot ser multiplicat per la matriu  $|\psi\rangle \langle\psi|$  que té les mateixes dimensions. No obstant sorgeix un problema amb el numerador de l'equació perquè  $\Pi_A$  no té les mateixes dimensions que  $|\psi\rangle \langle\psi|$ , encara no he pogut utilitzar aquesta equació adequadament. No sé com computar-la. Aquest problema l'adreçaré en la part experimental del treball.

En el mateix article es planteja la mateixa equació però per l'estat post-mesura expressat en forma de vector d'estat, molt semblat a l'equació 2.8. L'única diferència es que en l'equació de l'article no s'expressa el operador de mesura en la forma  $M_m^\dagger M_m$ , en canvi el autors ho expressen tan sols com  $\Pi_A$ , més concretament  $I_{2^N-2^{N_A}} \otimes \Pi_A$ . Potser la forma plantejada per els autors té en compte el conjugat hermitià, però no estic segur. L'equació esmentada en l'article és la següent:

$$|\psi_m\rangle = \frac{I_{2^N-2^{N_A}} \otimes \Pi_A |\psi\rangle}{\sqrt{\text{tr}(I_{2^N-2^{N_A}} \otimes \Pi_A |\psi\rangle \langle\psi|)}}$$

Al igual que amb l'altra equació, no sé com computar aquest mesurament.

<sup>21</sup>Es a dir, una mesura a només una part dels qubits que conformen un sistema.

## 2.5 Ordinadors quàntics

Tota aquesta teoria és aplicada a través de qubits físics que s'ubiquen al ordinadors quàntics. Hi han diversos tipus d'ordinadors quàntics, degut a que els qubits poden ser diversos sistemes. Poden ser fotons, chips de silici superconductors o ions atrapats per imants. No elaboraré més sobre aquest tema degut a que no és el tema central d'aquest treball, m'he centrat molt més en la teoria.

Però si vull generar imatges amb un ordinador quàntic, no necessito un? No necessàriament, perquè puc simular l'evolució dels estats quàntics amb un ordinador, al cap i a la fi, és només àlgebra lineal, són operacions que es poden fer perfectament en un ordinador. No obstant, quan s'intenta simular un sistema quàntic de molt qubits<sup>22</sup> un ordinador de sobretaula tardaria molt de temps i realment no és viable.

---

<sup>22</sup>Més de 50 per exemple.

# Capítol 3

## Intel·ligència artificial

Segurament has sentit parlar de l'intel·ligència artificial o de les xarxes neuronals, són conceptes que semblen abstractes però jo penso que son bastant intuïtius, intentaré que tú et sentis de la mateixa manera al final d'aquest capítol.

Intel·ligència artificial és un mot una mica ambigu, que és refereix a qualsevol algoritme que entra dintre del camps del *machine learning* o aprenentatge automàtic<sup>1</sup>. Aquests algoritmes simplement s'alteren a ells mateixos per fer millor la tasca que s'ha lis ha designat, no importa quin és el objectiu o com ho aconsegueix, lo que importa és si aprèn automàticament. Cal notar que els canvis que s'efectuen sobre si mateixos no han de ser predeterminats, si l'algoritme té una llista de les instruccions que va executant segon la situació no seria una intel·ligència artificial o un algoritme de *machine learning*.

La manera que tenen aquests algoritmes d'alterar-se a si mateixos usualment es canviant els paràmetres de les operacions dels quals estan compostos. Per exemple, en una regressió lineal, s'actualitzen els paràmetres de la recta que representa la tendència de les dades, com es pot veure a la figura 3.1.

Hi han diversos mètodes per ajustar els paràmetres, el més comú es ajustar-los segons la derivada d'una funció anomenada funció de pèrdua o *loss function*, usualment representada per la lletra  $\mathcal{L}$ . Aquesta funció representa els objectius

---

<sup>1</sup>No obstant, col·loquialment s'utilitza per denominar a qualsevol algoritme o robot que és intel·ligent o sembla que és intel·ligent.



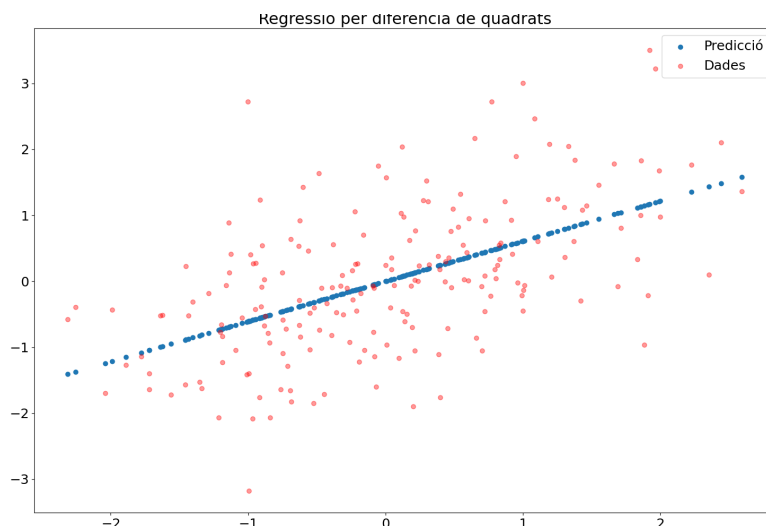


FIGURA 3.1: Exemple d'una regressió lineal de dades generades al atzar. Veure el codi a [E.1.0.1](#).

del programa i pot ser minimitzada o maximitzada, per exemple, en una regressió lineal es vol reduir la distancia entre els els punts de dades i la línia que prediu la tendència, com es pot veure en la figura 3.1.

Degut a que es poden realitzar molts tipus de funcions de pèrdua, ja sigui per la forma de la funció en si o per els paràmetres de la funció, el programes de *machine learning* són extremadament versàtils, la màxima expressió d'això es pot veure en les xarxes neuronals o *neural networks*. Aquests algoritmes són els més potents, complexos i polivalents, dintre del aprendizaje automàtic.. Precisa-ment utilitzo un d'aquests algoritmes d'aquests per generar les imatges. reconeixement d'imatges, la traducció i sintetització de textos, la conducció automàtica, els algoritmes de recomanació, i es clar, la generació d'imatges.

## 3.1 Xarxes neuronals

Aquests tipus d'algoritmes no tenen un nombre que fa recordar a les xarxes de neuronals dels nostres cervells per casualitat, estan directament inspirades en els nostres cervells. Són uns programes que consisteixen en la connexió de diverses operacions anomenades neurones, que conjuntament formen una xarxa, la qual s'organitza a partir de capes. Segons la variació del tipus de neurona i la estructura que aquestes formen podem tindre algoritmes destinats a fer diferents

tasques. Això juntament amb els diversos tipus de funció de pèrdua contribueix a la versatilitat de les xarxes neuronals. Aquests models d'intel·ligència artificial constitueixen el camp del *deep learning* o aprenentatge profund. S'anomenen d'aquesta forma per referenciar la gran profunditat d'aquests algorismes, es a dir el gran nombre de capes que tenen.

Una neurona consisteix simplement en una suma ponderada, a qual se li suma un altre número una funció no lineal que s'aplica al resultat. Les neurones tenen vectors com input i output. Per tant, una neurona es pot definir com:

$$\sigma \left( \sum_{i=1}^n w_i x_i + b \right) = \sigma (w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b)$$

Per  $\sigma$  sent una funció no lineal i  $n$  sent la mida del vector. Després estan el paràmetres,  $w_i$  i  $b$ , anomenats *weights* i *bias*.

Una neurona pot tindre diversos inputs que venen de diverses neurones, el mateix passa amb els outputs. Depenent de com es connectin entre si les neurones, aquestes passen a formar diversos tipus de capes. A partir dels tipus de capes i el nombre d'aquestes es com s'especifica l'arquitectura d'una xarxa neuronal.

Una vegada especificada la interconnectivitat de les neurones puc parlar de l'intuïció darrera dels paràmetres, *weight* especifica com de forta és la relació entre una neurona en una capa i una altre neurona en una capa veïna. I un *bias* especifica com d'important és una neurona, degut a que aquest número afecta al resultat a la suma, fent que aquesta sigui més alta.

Tornant a l'arquitectura, usualment aquesta es divideix en tres parts la capa d'input, les capes ocultes i la capa d'output. La quantitat de neurones que hi han a la capa d'inputs es la que defineix la mida del vector que es dona com input a la xarxa, degut a que cada element del vector es dona a cada neurona amb la capa. El mateix passa amb la capa d'outputs, cada output de cada neurona de la capa acaba sent un element en el vector que surt de la xarxa. Per tant el número de neurones que té cadascuna d'aquestes dues capes, especifica la mida del

vectors d'input i d'output de la xarxa respectivament. Per exemple si es vol donar com input a una xarxa una imatge de 16 per 16 píxels<sup>2</sup> calen 256 neurones en la capa d'inputs, una per cada píxel.

En canvi, les *hidden layers*, es a dir les capes ocultes, no tenen una mida determinada, el mateix passa amb el nombre d'aquestes que té la xarxa. Depenen de cada cas la quantitat de neurones que tenen aquestes capes i també el nombre d'aquestes, varia. Això, juntament amb el diversos tipus de capes és el que dona la versatilitat d'aquests algoritmes.

Entre els diferents tipus de capes que poden tindre les xarxes neuronals, la més comú i simple d'aquestes és la *fully connected layer*, o una capa completament connectada, veure la figura 3.2. Les neurones que formen aquesta capa estan connectades a totes les neurones de la capa anterior i així mateix, a totes les neurones de la capa següent. Aquestes capes l'única forma de variar que tenen és a partir de canviar el nombre de neurones, ja que no hi ha forma d'alterar el funcionament de les neurones.

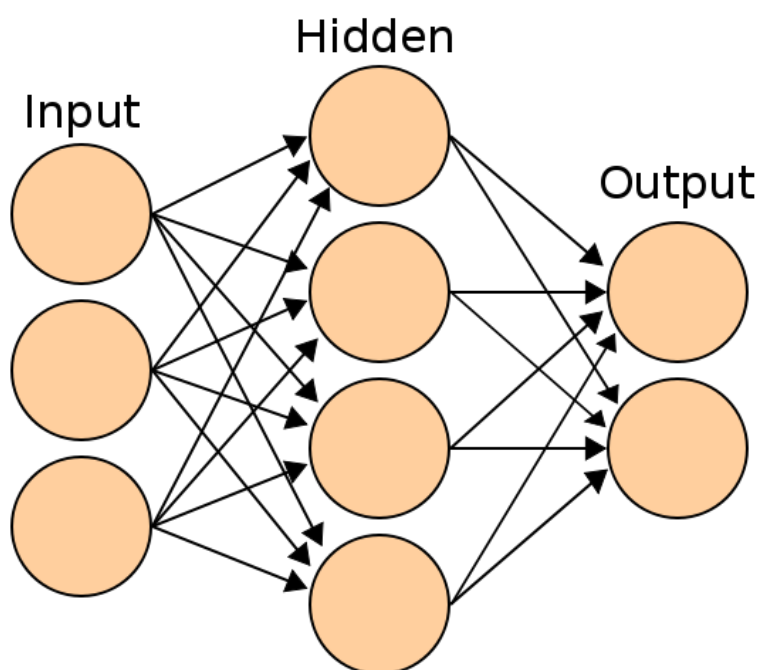


FIGURA 3.2: Usualment les xarxes neuronals es representen d'aquesta forma, amb fletxes i boles. Les boles representarien cada neurona i les fletxes mostren com estan connectades. La *hidden layer* d'aquesta representació es pot veure que és una *fully connected layer* degut a com està connectada a les altres capes, rebent cada neurona el output de cada neurona anterior.

<sup>2</sup>Una imatge en blanc i negre.

No obstant, es pot variar la funció d'activació que tenen, que ha de ser una funció no lineal. La més utilitzada és la sigmoide:

$$f(x) = \frac{1}{1 + e^{-x}}$$

## 3.2 Descens del gradient

Una vegada he parlat de les xarxes neuronals, he de comentar com aquestes evolucionen al llarg del temps, es a dir com es van actualitzen a si mateixes per complir la tasca que s'ha li's ha encomanat. Ja he comentat que existeix una funció anomenada *loss function*, la qual es deriva per actualitzar els paràmetres de la xarxa. El mecanisme per el qual s'actualitzen els paràmetres s'anomena el descens del gradient.

Aquest procés comença amb la funció de pèrdua, que esmenta els objectius de la xarxa. Els punts mínims d'aquesta funció representen els punts òptims de la xarxa, als quals es vol arribar.

Per explicar-lo utilitzaré un exemple pràctic, primer de tot parlaré sobre la funció de pèrdua que s'utilitza i a continuació de l'actualització dels paràmetres.

Parlaré de la classificació d'imatges, si es volen classificar imatges de gats i gossos, s'assignen dos etiquetes a aquestes, per exemple, 1 als gats i 0 als gossos. A continuació s'escull una funció de pèrdua<sup>3</sup> com *binary cross entropy* o *log loss*. La funció *binary cross entropy* és la següent:

$$\mathcal{L} = -t \log(y) - (1 - t) \log(1 - y) \quad (3.1)$$

Amb  $t_i$  sent l'etiqueta que ha de tenir l'imatge, anomenada etiqueta real, i  $y_i$  sent la etiqueta que el model assigna a l'imatge. Per tant si l'etiqueta real és 1, l'equació acaba sent:

$$\mathcal{L}_1 = -\log(y)$$

---

<sup>3</sup>Una que funcioni per la classificació d'imatges, es clar.

I el cas contrari per  $t = 0$  seria:

$$\mathcal{L}_0 = -\log(1 - y)$$

Si donen com input l'imatge d'un gat i el programa es dona com output 0.92 tenim que la pèrdua és de:

$$\mathcal{L} = -t \log(y) - (1 - t) \log(1 - y) = -\log(0.92) \simeq 0.0834$$

En canvi, si el programa es dona un output de 0.15 la pèrdua seria:

$$\mathcal{L} = -t \log(y) - (1 - t) \log(1 - y) = -\log(0.15) \simeq 1.897$$

Es pot veure que si l'etiqueta que posa el model s'allunya més de l'etiqueta real la pèrdua acaba és més gran. El mateix es pot veure per les imatges del gossos, es a dir, les imatges que haurien de tenir etiqueta zero:

$$\mathcal{L} = -0 \cdot \log(0.89) - (1 - 0) \cdot \log(1 - 0.89) = -\log(1 - 0.89) \simeq 2.207$$

$$\mathcal{L} = -0 \cdot \log(0.08) - (1 - 0) \cdot \log(1 - 0.08) = -\log(1 - 0.08) \simeq 0.0834$$

Per tant quan més pròximes estén les prediccions (els outputs del model) a les etiquetes ideals, menor serà la pèrdua. Llavors al minimitzar la funció es trobarà el punt òptim on totes les prediccions seran iguals a les etiquetes.

A aquests punts òptims s'ha arriben actualitzant els paràmetres a partir de la derivada de la funció, concretament a través del seu gradient. El gradient d'una funció és un vector on els seus elements són la derivada parcial respecte a cada paràmetre (per aclarir, una entrada per paràmetre). El gradient d'una funció  $f(\theta)$

es representa com  $\nabla f(\theta)$ , i es pot escriure en forma de vector com:

$$\nabla f(\theta) = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ \vdots \\ \frac{\partial f}{\partial \theta_{n-1}} \\ \frac{\partial f}{\partial \theta_n} \end{bmatrix}$$

No fa falta fixar-se en lo que és exactament un gradient, només s'ha de saber que cada paràmetre de la xarxa neuronal<sup>4</sup> s'ha d'actualitzar d'acord amb la derivada respecte al paràmetre. Això es pot entendre com canviar lleugerament un paràmetre acord amb la direcció de la derivada respecte a ell. L'objecte que s'encarrega d'actualitzar-los s'anomena optimitzador. L'optimitzador més senzill que hi ha és el següent:

$$\theta_i^t = \theta_i^{t-1} \pm \eta \frac{\partial \mathcal{L}(\theta)}{\partial \theta_i^{t-1}} \quad (3.2)$$

Es pot veure com s'actualitza el paràmetre  $\theta_i$ . En l'equació he posat el superíndex  $t$  per expressar aquesta actualització, passant d'un temps  $t - 1$  a  $t$ . A més a més, utilitzo el símbol  $\theta$  per expressar tots els paràmetres del model. En els optimitzadors s'afegeix un nombre  $\eta$  anomenat *learning rate*. Usualment és un nombre petit<sup>5</sup>, aquest nombre especifica com de ràpid canvien els paràmetres. El *learning rate* es pot anar ajustant depenen de la situació o del model en el qual s'implementi. Alteracions en aquest pot causar diversos fenòmens tan negatius com positius.

Cal notar que en l'optimitzador he utilitzar el signe  $\pm$  perquè si és positiu significa que s'està ascendint pel gradient, i si és negatiu s'està descendint. No obstant, usualment s'utilitza el signe negatiu per convenció, degut a que la majoria de funcions de pèrdua estan dissenyades per ser descendides.

Al fer la derivada de la funció de pèrdua es pot veure immediatament que s'ha d'aplicar la regla de la cadena, ja que l'estructura de la xarxa neuronal està

---

<sup>4</sup>Cada *weight* i cada *bias*.

<sup>5</sup>Entre 0.1 i 0.001 per exemple.

composta per una serie de funcions compostes entre si. En la secció següent parlaré del procediment que s'utilitza per avaluar el gradient, anomenat *backpropagation* o propagació inversa.

### 3.1 Backpropagation

Al aplicar la regla de la cadena 'de fora cap a dins' s'ha de començar a derivar per l'última capa i acabar per la primera, d'aquí ve el nom *backpropagation* perquè es propaga l'error en direcció contrària. Mentre que quan es dona un input a la xarxa, s'anomena *forward propagation* o *forward pass*.

No entraré en profunditat sobre l'intuïció de la *backpropagation* en aquesta secció, només em limitaré a esmentar la manera en la qual es calcula.

L'activació<sup>6</sup> d'una neurona  $j$  en l'última capa  $L$  de la xarxa es defineix com:

$$a_j^L = \sigma \left( \sum_{k=0}^{n_{L-1}} w_{jk}^L a_k^{L-1} + b_j^L \right)$$

On  $a_k^{L-1}$  és l'activació d'una neurona  $k$  en la capa anterior  $L - 1$ , i on el *weight*  $w_{jk}^L$  és el paràmetre que expressa en què mesura es connecta la neurona  $j$  a la neurona  $k$ . Com ja he dit, expressa com de forta és aquesta connexió. La suma es fa al llarg de  $n_{L-1}$  que és el nombre de neurones que té la capa  $L - 1$ . Utilitzant aquesta definició ja es poden efectuar les derivades.

No obstant, convé definir el terme  $z_j^L$  per procedir a fer les derivades. Simplement és l'equació d'una neurona però sense la funció d'activació:

$$z_j^L = \sum_{k=0}^{n_{L-1}} w_{jk}^L a_k^{L-1} + b_j^L \quad (3.3)$$

L'objectiu és obtenir la derivada de la *loss function* respecte a un *weight* qualsevol i un *bias* qualsevol. Per tant, s'han d'obtenir les següents derivades parcials:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^L} \text{ i } \frac{\partial \mathcal{L}}{\partial b_j^L}$$

---

<sup>6</sup>S'anomena així al resultat d'una neurona.

Començaré amb la derivada del *weight*, al aplicar la regla de cadena obtenim que:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^L} = \frac{\partial z_k^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial \mathcal{L}}{\partial a_j^L} \quad (3.4)$$

La primera derivada, és la derivada de  $z_j^L$  respecte al *weight*:

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1}$$

La següent és la derivada del resultat de la neurona  $a_j^L$  respecte a  $z_j^L$ , que resulta ser la derivada de la funció d'activació de la neurona:

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$$

Finalment està la derivada de la funció de pèrdua respecte al output de la xarxa, es a dir, l'activació d'una de les últimes neurones. La qual és simplement la derivada de la funció de pèrdua, per tant, varia en cada cas.

Ja sabent totes les derivades, es pot reescriure l'equació 3.4 es pot escriure com<sup>7</sup>:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^L} = \frac{\partial z_k^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial \mathcal{L}}{\partial a_j^L} = a_k^{L-1} \sigma'(z_j^L) \frac{\partial \mathcal{L}}{\partial a_j^L}$$

No obstant falta un detall, per efectuar la derivada de l'activació d'una neurona respecte a la funció de pèrdua s'ha d'afegir un sumatori:

$$\frac{\partial \mathcal{L}}{\partial a_j^{L-1}} = \sum_{j=0}^{n_{L-1}} \frac{\partial z_j^L}{\partial a_k^{L-1}} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial \mathcal{L}}{\partial a_j^L}$$

La suma representa que aquesta neurona té un output que es propaga cap a endavant afectant les neurones que la segueixen.

Finalment he d'esmentar la derivada de la funció de pèrdua respecte a un *bias* qualsevol  $b_k^L$ . Al aplicar la regla de la cadena es pot veure que aquesta derivada és:

$$\frac{\partial \mathcal{L}}{\partial b_k^L} = \frac{\partial z_k^L}{\partial b_k^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial \mathcal{L}}{\partial a_k^L}$$

---

<sup>7</sup>Deixo l'última derivada  $\frac{\partial \mathcal{L}}{\partial a_j^L}$  sense reescriure perquè aquest terme pot variar depenent de la funció de pèrdua que s'utilitza.



Al veure l'equació 3.3 es pot veure que la derivada  $\frac{\partial z_k^L}{\partial b_k^L}$  serà 1 degut a que  $b_k^L$  és una constant. Les altres derivades de l'equació ja les he esmentat anteriorment. Per tant finalment tenim que:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{jk}^L} &= a_k^{L-1} \sigma'(z_j^L) \frac{\partial \mathcal{L}}{\partial a_j^L} \\ \frac{\partial \mathcal{L}}{\partial b_k^L} &= \sigma'(z_j^L) \frac{\partial \mathcal{L}}{\partial a_j^L}\end{aligned}$$

Amb aquestes derivades ja es pot desenvolupar el vector gradient, que com ja he dit està compost per les derivades de cada paràmetre de la xarxa, no obstant, concretament està compost la mitjana d'aquestes, perquè es vol actualitzar els paràmetres per poder minimitzar la pèrdua en totes les dades disponibles. Per tant es calcula l'error d'unes quantes mostres del *dataset*, i s'efectua una mitjana d'aquests errors.

Tota aquesta teoria es veurà implementada en la part pràctica en forma de codi, degut a que m'he vist amb la necessitat de tenir una xarxa neuronal programada des de zero. No obstant, usualment s'utilitzen plataformes com *TensorFlow* [25] o *PyTorch* [26] al programar xarxes neuronals, degut a que aquests *frameworks* faciliten moltíssim el treball als programadors.

### 3.3 Generative adversial networks

Com ja he dit hi han molts tipus de xarxes neuronals, no obstant, en aquest treball només em centraré en un tipus en específic, les xarxes generatives adversatives o *generative adversial networks (GAN)* en anglès.

Aquestes xarxes, com el seu nom diu, s'utilitzen per generar dades, usualment s'apliquen a imatges. Es troben al darrera de projectes com *This person does not exist* [27, 28], una pàgina web que et genera una cara d'una persona que no existeix, ja que és una cara generada artificialment a partir d'aquest tipus de models.

Aquests tipus de models van ser introduïts per primera vegada al 2014 per Ian Goodfellow [4], des de llavors s'han convertit en un dels models de *deep learning* més sòlids i utilitzats.

Aquests algoritmes consisteixen en dos models (xarxes) diferents, un generador i un discriminador, amb objectius oposats. Per aquesta raó s'inclou paraula adversativa en el nom. El generador i discriminador es poden entendre com un falsificador de bitllets i un policia que el vol atrapar. On el generador és el falsificador de bitllets i el discriminador és el policia.

L'analogia és la següent: Una vegada el falsificador comença el seu entrenament, el policia contesta immediatament, i amb el temps es torna millor al seu treball, podent distingir entre els bitllets falsificats i els reals. El falsificador respon a això millorant les seves tècniques, per tant el policia han de millorar encara més. És un cicle en el qual aquestes forces antagonistes es fan millorar l'una al altra.

El mateix passa amb el generador i el discriminador. El discriminador aprèn a distingir entre les imatges reals i les imatges falses que fabrica el generador, mentre que el generador aprèn a enganyar al discriminador.

Si s'especifiquen bé els objectius de cada model, arriben a *zero sum game*<sup>8</sup>. La manera en la qual és soluciona aquest tipus de joc és arribant a un equilibri de Nash [5], on el discriminador no sap diferenciar entre les imatges reals i les falses<sup>9</sup>.

En l'article original [4] s'esmenta un pseudocodi per aquests models, que he representat en l'algoritme 1. En aquest es parla de *minibatch* que és simplement un grup d'imatges o de dades, i de soroll, que són dades generades aleatòriament i que es donen com input al generador perquè aquest no generi exactament les mateixes imatges cada vegada. El soroll afegeix variació als outputs del generador, però sempre s'intenta que sigui en una petita quantitat.

<sup>8</sup>Un *zero sum game* és simplement un joc entre dos jugadors en que per guanyar un l'altre ha de perdre e.g. joc d'estirar la corda entre dos equips.

<sup>9</sup>Que el discriminador no sàpiga diferenciar no implica arribar a un equilibri de Nash, aquest concepte és definit d'una altra forma que està fora de l'àmbit d'aquest treball [4].

---

**Algorithm 1** Pseudocodi per una xarxa generativa adversativa
 

---

**for** número de interaccions **do**  
     **for**  $k$  pasos **do**  
         Treure minibatch de  $m$  mostres de soroll  $\{z_i, \dots, z_m\}$  de la distribució de soroll  $p_g(z)$   
         Treure minibatch de  $m$  mostres d'exemples  $\{x_i, \dots, x_m\}$  de la distribució d'exemples  $p_{\text{data}}(x)$   
         Actualitzar el discriminador ascendint el seu gradient:

$$\nabla_{\theta} \frac{1}{m} \sum_{i=1}^m [\log D(x_i) + \log(1 - D(G(z_i)))]$$

**end for**  
     Treure minibatch de  $m$  mostres de soroll  $\{z_i, \dots, z_m\}$  de la distribució de soroll  $p_g(z)$   
     Actualitzar el generador descendent el seu gradient:

$$\nabla_{\theta} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z_i)))$$

**end for**

---

En el pseudocodi es pot veure que primer s'actualitza el discriminador  $k$  vegades i després el generador una sola vegada. Això és perquè interessa que el discriminador sàpiga distingir les imatges ràpidament, per poder indicar al generador com generar les imatges. Si el discriminador esmenta que unes imatges de gats són imatges de gossos, el generador fabricarà imatges de gossos pensant que ho són de gats, perquè aquest aprèn a partir del que l'indica el discriminador.

També es pot veure com és la funció de pèrdua, que en l'article el autors anomenen *Minimax loss function*, la qual en la pràctica és la mateixa que la *Binary Cross Entropy (BCE)*. Es pot veure que en la BCE, quan  $t = 0$ , que seria l'etiqueta per les imatges falses del generador, aquesta funció seria  $\log(1 - y)$ . En el cas contrari, per  $t = 1$ , la funció seria  $\log(y)$ .

## Capítol 4

# Generació d'imatges amb un ordinador quàntic

Investigadors en informació quàntica al veure el potencial que tenen els ordinadors quàntics i l'intel·ligència artificial, no es van poder resistir a crear un nou camp d'investigació, el *Quantum Machine Learning (QML)*, o aprenentatge automàtic quàntic. Al igual que les xarxes neuronals són les estrelles dintre del *machine learning*, les xarxes neuronals quàntiques també ho són dintre del *quantum machine learning*.

Des de que es van començar a investigar aquests algoritmes s'han arribat a implementar diversos tipus de xarxes neuronals. Principalment pel que fa a la generació i classificació d'imatges i dades.

No obstant aquests algoritmes no són completament quàntics, usualment consisteixen en actualitzar els paràmetres d'un circuit quàntic perquè aquest generi les dades. On les actualitzacions dels paràmetres es calculen amb un ordinador clàssic. Això es veurà molt clar quan expliqui la part pràctica del treball.

Abans de continuar amb el capítol he de dir que existeixen diversos algoritmes dintre del *quantum machine learning*, no tot en la vida són xarxes neuronals. Per exemple, es pot donar a terme classificació de dades mitjançant *support vector machines*<sup>1</sup> [29, 30] o amb un anàleg de la regressió lineal [31]. No obstant, en aquest capítol em centraré exclusivament en les xarxes neuronals quàntiques.

---

<sup>1</sup>Classificar dades de dimensions petites en espais vectorials molt grans.

## 4.1 Descens del gradient quàntic

De moment tindre en consideració que una xarxa neuronal quàntica és un circuit quàntic qualsevol però parametritzat, és a dir, que té portes quàntiques parametritzades. Això juntament amb una funció de pèrdua i un *dataset* és suficient per explicar com s'actualitzen els paràmetres.

L'objectiu principal és avaluar la derivada respecte a un paràmetre. Sorprenentment és dona a terme d'una manera més senzilla que amb una xarxa neuronal clàssica. Tan sols s'utilitza la definició de la derivada per poder avaluar-la: S'altera lleugerament un sol paràmetre i es treu la diferència entre dos outputs del circuit quàntic que tenen el paràmetre alterat. Aquest mètode per avaluar la derivada s'anomena *parameter shift* [2, 32].

Si un circuit quàntic té un vector de paràmetres  $\theta$ , per un paràmetre  $\theta_i$ , es defineix un vector de pertubació  $\Delta_i$  ple de zeros, de igual mida que  $\theta$ , però que en la posició de  $\theta_i$  té un 1. A partir del vector de pertubació es pot definir la derivada de la funció de pèrdua  $\mathcal{L}(\cdot)$  respecte al paràmetre  $\theta_i$ :

$$\frac{\partial}{\partial \theta_i} \mathcal{L}(\theta) = \mathcal{L}(\theta + \frac{\pi}{4} \Delta_i) - \mathcal{L}(\theta - \frac{\pi}{4} \Delta_i)$$

Es pot veure que el vector  $\theta \pm \frac{\pi}{4} \Delta_i$  és el vector  $\theta$  però amb una petita variació en la posició  $i$  que és la que correspon al paràmetre  $\theta_i$ . Amb aquest mètode ja es pot desenvolupar pràcticament qualsevol actualització de paràmetres, aquesta és la part senzilla de les xarxes neuronals quàntiques, la dificultat radica en la forma dels circuits quàntics que les componen.

## 4.2 Circuits quàntics per xarxes neuronals

L'única qualitat obligatòria que han de dintre aquests circuits és que han de estar parametritzats. Usualment estan compostos per una gran quantitat de portes rotacionals parametritzades, les quals ja he presentat en les equacions 2.2, 2.3 i 2.4.

El problema al qual s'enfrenten els investigadors dedicats a les xarxes neuronals quàntiques és la manera amb la qual implementar funcions no-lienals en els circuits quàntics. Aquests tipus de funcions són les responsables de la gran complexitat i profunditat de les xarxes neuronals clàssiques. A primera vista por semblar una tasca quasi impossible a causa de la naturalesa lineal de la computació quàntica. No obstant, durant els anys s'han desenvolupat diverses tècniques per donar a terme aquesta fita, les quals comentaré a continuació.

A partir d'una combinació de rotacions i portes que entrellacen qubits es pot arribar a implementar una funció semblant a la tangent hiperbòlica<sup>2</sup> en un circuit quàntic [33]. No obstant, aquest mètode té un gran desavantatge, consisteix en un circuit que s'ha d'anar mesurant i repetint per veure si funciona correctament, els autors parlen de *repeat until success* degut a que s'ha de mesurar un qubit i mirar si dona  $|0\rangle$  per assegurar que aquesta funció ha sigut aplicada correctament<sup>3</sup>.

En un article posterior<sup>4</sup>, en el qual els autors generen distribucions contínues a partir d'una xarxa generativa adversària quàntica. S'especifica que les no-linearitats presents en l'algoritme no formen part del circuit quàntic, es a dir, funcions que s'implementen clàssicament als resultats dels circuits quàntics o a les dades que s'introdueixen als circuits [34]. Aquesta és una mesura molt simple que possiblement implementaré durant la part experimental.

Al 2019 es va publicar un dels articles que més m'agraden<sup>5</sup> Cong et. al. (2019) [35], en ell els autors presenten una xarxa neuronal convolucional quàntica. No fa falta entrar en detall, però es diuen convolucionals perquè s'apliquen convolucions a les imatges que es volen classificar, es multiplica una part de l'imatge per una matriu que s'anomena filtre [36]. Simplement tenen un altre tipus de capes que no són les capes completament connectades. L'única afirmació dels autors que s'ha de recalcar, és que a partir de reduir els graus de llibertat

<sup>2</sup>La tangent hiperbòlica també s'utilitza com a funció no-lienal en el *deep learning*.

<sup>3</sup>En cas de que doni  $|1\rangle$ , s'ha de repetir el procés.

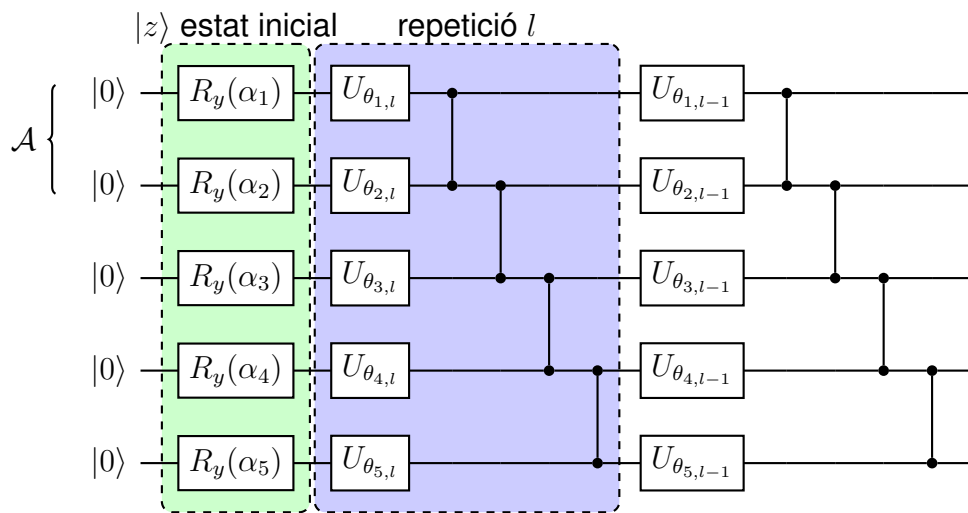
<sup>4</sup>Realitzat en part per un dels autors del mètode anterior.

<sup>5</sup>Utilitzen imatges de gats a les figures i és un dels primers articles que vaig llegir d'aquest camp fa més de dos anys. A més a més la xarxa neuronal esmentada en l'article està completament programada en TensorFlow Quantum [2], i això sempre s'agraeix.

(*degrees of freedom*) en els circuits quàntics, sorgeixen no-linearitats. Això es degut als mesuraments parcials que es donen a terme en un moment determinat del algoritme.

Un dels mètodes que m'ha semblat més interessant, és l'implementat en una altra xarxa convolucional quàntica, on al aplicar els filtres que s'utilitzen per la convolució, s'implementa una funció no-lienal. Aquest mètode no el puc arribar a comprendre, les equacions que utilitzen són molt complexes i no sembla que arriben a utilitzar un mesurament parcial en cap moment. Simplement els autors presenten una equació i diuen que no és lienal, i no puc arribar a comprendre perquè ho és.

Finalment he de parlar de l'article que he seguit per realitzar aquest treball, Huang et. al. (2021) [5], en aquest al igual que en l'article de Cong et. el. (2019) [35], s'utilitzen mesuraments quàntics per introduir no-linearitats al algoritme. Concretament els autors implementen aquests mesuraments tant en el generador quàntic d'imatges, com en el discriminador. Els circuits quàntics que utilitzen pel generador d'imatges tenen la següent forma:



On les portes  $R_y$  amb un paràmetre  $\alpha_i$ , marcades en verd, són utilitzades per introduir les dades al circuit. Aquestes dades són simplement soroll que crea varietat en els outputs dels circuits, al igual que el soroll que s'utilitza en les GAN clàssiques (algoritme 1).

El circuit que genera les imatges realment consisteix en les portes  $U_\theta$  i CZ, marcades en blau. Aquestes portes s'agrupen en les repeticions- $l$ , que s'utilitzen per afegir profunditat i complexitat als circuits. En aquest cas el circuit té dues repeticions- $l$ <sup>6</sup>. Al mesurar es fa un mesurament parcial però els autors especifiquen que es fa d'una forma concreta que explicaré a continuació.

Per un estat  $|\Psi_\alpha\rangle$  que surt del circuit especificat posteriorment, l'estat  $\rho(z)$  després d'una mesura parcial sobre el qubits  $\mathcal{A}$ , és:

$$\rho(z) = \frac{\text{tr}_{\mathcal{A}}(\Pi_{\mathcal{A}} |\Psi(z)\rangle \langle \Psi(z)|)}{\text{tr}(\Pi_{\mathcal{A}} \otimes I_{2^{N-N_{\mathcal{A}}}} |\Psi(z)\rangle \langle \Psi(z)|)}$$

Els autors afirmen que aquest estat  $\rho(\alpha)$ , és una funció no-lineal de l'estat  $|z\rangle$ . Això és degut a que tant el denominador com el numerador de l'equació són funcions de  $|z\rangle$ .

No obstant, en altres treballs com per exemple Zoufal et.al. (2019) [37], on es defineix una GAN quàntica que genera distribucions de probabilitat, els autors no mencionen la necessitat de tenir funcions no lineals en alguna part del algoritme.

---

<sup>6</sup>Es a dir  $l = 2$ .



## **Part II**

### **Part Experimental**

# Capítol 5

## Plantejament de l'hipòtesi

Podria haver anat per altres vies com la generació d'imatges amb color o l'implementació d'un d'una porta  $X$  en una part específica del circuit quàntic que genera les imatges, que al posar-la o no, el model generes dos tipus d'imatge a través del mateix circuit. Tanmateix, les dues propostes requerien de desenvolupar nous conceptes, ja que són idees meves pròpies, però al no tindre el coneixements necessaris vaig descartar-les.

Llavors, al ser l'implementació de les funcions no-lineals un assumpte lleugerament conflictiu entre els diversos models de xarxes neuronals quàntiques, com ja he comentat en la secció [4.2](#), havia decidit des d'un principi centrar-me en aquesta qüestió en concret.

La pregunta a investigar és la següent:

L'incorporació d'una funció no-lineal en el circuit quàntic del model, a partir d'una mesura parcial, causa que el model arribi més ràpidament al punt òptim?

En altres paraules, volia veure si la mesura parcial afectaria positivament en l'eficiència del model, fent que la generació de les imatges desitjades es dones a terme en un menor temps.

Sembla una qüestió senzilla, però la dificultat del experiment radica en crear la xarxa neuronal en si, amb totes les seves parts accessibles per poder fer els canvis que siguin necessaris. L'única manera de fer l'experiment seria programant el model, una tasca que tenia clar que faria des de el principi.

# Capítol 6

## Programació del model

Posteriorment a començar a programar el model, jo ja sabia que ho havia de realitzar en Python, ja havia creat algun algorisme abans de començar aquest treball i tenia experiència construint i executant circuits quàntics amb Cirq [38], una eina desenvolupada per Google. A més a més, sabia de l'existència de TensorFlow Quantum [2], una altra eina desenvolupada per Google destinada a la creació de xarxes neuronals quàntiques i algorismes de *quantum machine learning* en general. També tenia experiència en aquest *framework*. Per tant, TensorFlow Quantum va ser la meva primera opció, tenia pensat basar el meu codi en el tutorial de TensorFlow sobre una xarxa convolucional generativa adversarial. Havia de canviar el generador per un circuit quàntic que s'optimitza a partir d'un diferenciador<sup>1</sup> automàtic provinent de TensorFlow Quantum. Els canvis que corresponien al discriminador simplement havien de ser un canvi d'arquitectura, passar d'una xarxa més complexa a una de més simple que només consistiria a en unes poques capes totalment connectades.

El primer problema que em vaig trobar va ser la creació del *dataset* que alimenta a la xarxa discriminativa. A causa de la peculiaritat de les imatges que volia generar, havia de crear-lo manualment. Usualment les imatges que componen els *datasets* utilitzats en *deep machine learning* són extrems de bancs d'informació amb mides enormes. En el meu cas, havien de ser generades per

---

<sup>1</sup>Objecte que calcula el gradient d'un circuit quàntic.

mi, per tant havia de convertir matrius de Numpy en *datasets* de TensorFlow. Recordo que en va costar arribar a tindre la solució a aquest problema.

## 6.1 Discriminador

Una vegada ja tenia fet el *dataset* em vaig posar a fer el model. En el tutorial per una [DCGAN \(GAN convolucional\)](#) els dos models eren entrenats per la funció `train_step()`, que representa una iteració en el procés d'optimització. En aquesta es crida a la funció `tf.GradientTape` per guardar el diferenciador automàtic. El problema que vaig dintre amb aquesta funció es que directament no funcionava amb el discriminador, aquest no era optimitzat. Després d'intentar solucionar l'error pels meus propis medis, mirant la causa d'aquest i de buscar a forums la solució o causa, em vaig rendir. Ja havia estat uns quants mesos intentant desenvolupant el model amb TensorFlow i TensorFlow Quantum. Havia creat les capes del generador quàntic manualment, també ho havia fet amb el optimitzador<sup>2</sup>. Tenia el model gairebé acabat, però al no poder optimitzar el discriminador em vaig veure obligat a canviar d'estrategia.

Existeixen dos grans *frameworks* per crear i executar circuits quàntics: Cirq [38], desenvolupat per Google, i Qiskit [39], creat per IBM. Una vegada vaig decidir no continuar amb Cirq, havia de provar amb Qiskit. La veritat, havia d'haver començar amb Qiskit des de el principi, és més útil (té moltes més característiques), i el més important, té una major comunitat, per tant, es més fàcil trobar solucions als error que pots tenir i és més fàcil trobar a persones disposades a ajudar-te.

Al igual que Cirq té un *framework* per poder desenvolupar xarxes neuronals (TensorFlow Quantum); Qiskit també té el seu, anomenat PyTorch [26], no obstant no té una integració tan directa, ja que no estan desenvolupats pel mateix equip, ni la mateixa companyia.

---

<sup>2</sup>No sabia ni si funcionarien adequadament, degut que per provar-ho, tenia que tenir tot el model enllestit.

Per tant, al canviar de Cirq a Qiskit, també havia de canviar de TensorFlow a PyTorch, però no tenia res d'experiència amb PyTorch, sabia que la transició seria complicada, i tenia raó. No vaig ni aconseguir crear el *dataset* que contenia les imatges que alimentar al discriminador.

Després d'intentar-lo amb TensorFlow Quantum i amb PyTorch, vaig decidir prescindir de *frameworks* per crear models de *machine learning*. El discriminador, al ser una xarxa tan simple, la podia crear des de zero. Llavors vaig començar a buscar codi, volia una xarxa neuronal que feta amb Numpy, una llibreria de Python per fer càlculs amb vectors i matrius amb la qual tenia bastant experiència.

Després de provar dues opcions que més o menys m'agradaven<sup>3</sup>, va aparèixer un repositori<sup>4</sup> de Michael Nielsen, un dels autors de *Quantum Computation and Quantum Information* [8] que em va salvar.

El repositori tenia codi per xarxes neuronals que estava estructurat d'una forma que m'agradava i encara més important, que entenia. Inclús tènien diverses versions d'una mateixa xarxa neuronal, amb un nivell de complexitat diferent. Llavors, a partir del model més simple que havia en el repositori<sup>5</sup>, vaig començar a desenvolupar el discriminador.

Com es pot veure al codi final per el discriminador al apèndix E.1.0.2, he fet bastants canvis, però no he canviat l'estructura o el funcionament teòric. La majoria dels canvis són per afegir més funcionalitat al model, com per exemple l'emmagatzematge de les dades per poder al final veure-les en un gràfic.

El canvi més important és que inclou les dues formes d'optimitzar el model, amb dues funcions de pèrdua que funcionen de manera diferents però que són la mateixa, la *Binary Cross Entropy* i la *MinMax*. Ja he parlat d'aquestes dues funcions en la part teòrica del treball. Això en el codi està materialitzat en dues

---

<sup>3</sup>Buscava codi estructurat d'una forma en concret, que estigui dissenyat amb la filosofia de *Object Oriented Programming*, una forma d'escriure codi en el qual tot s'implementa en un sol objecte.

<sup>4</sup>[link del repositori](#)

<sup>5</sup>Es pot veure el codi original al apèndix E.1.0.1.

funcions<sup>6</sup> diferents, `backprop_bce()` i `backprop_minimax()`. No fa falta entrar en detall sobre que va cadascuna de les funcions, degut a que, fan el mateix i no tenen ninguna diferencia en termes de eficàcia o rapidesa. Les vaig fer tan sols per comprovar que no hi hauria ninguna diferencia. El codi final del discriminadors es pot veure al apèndix [E.1.0.2](#). També en el repositori del treball hi ha un altre arxiu anomenat `discriminator.py`, que conté una altre versió en la qual intentava implementar diverses funcions d'activació en el model, però no ho vaig aconseguir, tanmateix, no em preocupa perquè no es una part vital del treball, no passa res per tindre el discriminador només amb la sigmoide.

## 6.2 Generador

El desenvolupament de l'altre part del model, el generador, va ser molt diferent. Després de provar a fer-ho amb TensorFlow, sense obtindre bons resultats, no tenia altra opció que fer-ho tot manualment i jo mateix<sup>7</sup>, no obstant, ja tenia experiència en fer pseudo-xarxes neuronals quàntiques i això em tranquil·litzava. Sabia perfectament el que tenia que fer, i com ho havia de fer: Implementar el mètode de *parameter shift* en els circuits quàntics dels quals vaig parlar en la secció [4.2](#).

Una vegada vaig crear els circuits quàntics amb una funció fins a un punt en el qual sentia que ja estava tot perfecte, em vaig posar amb la optimització, de la qual parlaré a continuació.

Primer de tot cal remarcar que el model s'optimitza a partir d'un *batch*, es a dir, un grup d'imatges, en aquest cas, un grup de dades. S'agafa la mitjana dels errors de cada *batch*, i s'actualitzen els paràmetres a partir d'aquesta. La motivació per treball en *batch* i no dades individuals es perquè s'ha de mirar l'error d'unes quantes dades a la vegada, d'aquesta manera l'optimització és més robusta.

---

<sup>6</sup>Funcions de Python.

<sup>7</sup>No em vaig ni plantejar buscar codi per internet perquè pensava que els autors dels paper que vaig llegir, les úniques persones que sabia que podien tindre el codi, no el penjarien.

Tornant al funcionament del procediment, per començar s'han d'agafar els paràmetres a optimitzar, que estaran en forma d'un vector  $\theta$ , escollir un d'ells que anomenaré  $\theta_i$ , que és el paràmetre que s'optimitzarà, i crear un vector de pertubació  $\Delta_i$ . Aquest vector consisteix en un vector amb la mateixa mida que  $\theta$ , però amb tot de zeros, menys a la posició del paràmetre que es vol optimitzar, en la qual tindrà un 1.

Lavors es creen dos vectors de paràmetres nous,  $\theta_i^+$  i  $\theta_i^-$ , multiplicant  $\pm \frac{\pi}{4}$  pel vector de pertubació, i sumant el resultat al vector de paràmetres original<sup>8</sup>. A continuació, es creen dues imatges, cadascuna corresponent a un dels vectors de paràmetres creats, una amb  $\theta_i^+$  i l'altra amb  $\theta_i^-$ . Per últim, s'avalua la funció de pèrdua per a cada imatge generada i es treu la diferència entre elles, d'aquesta manera calculant una derivada  $\frac{\partial \mathcal{L}}{\partial \theta_i}$  respecte al paràmetre  $\theta_i$ . Una vegada es té una derivada per a cada paràmetre de  $\theta$  es pot construir el vector gradient  $\nabla_\theta$ . He de dir aquest vector, ha de tenir la mateixa mida que el vector  $\theta$ , degut a que cada element en el gradient correspon a la derivada d'un paràmetre de  $\theta$ .

Al llarg de l'optimització es van sumant les derivades a  $\nabla_\theta$ , si això es fa per a tots els paràmetres de  $\theta$  i per a totes les dades del *batch*, finalment es pot treure la derivada mitjana dividint els elements de  $\nabla_\theta$  per la quantitat de dades que hi han en un *batch*.

Amb el gradient  $\nabla_\theta$  resultat es poden finalment optimitzar tots els paràmetres, d'acord amb les derivades mitjanes de les quals està compost.

El pseudocodi per aquest procediment es pot trobar en la figura 6.1. En aquesta es pot veure que es crida al discriminador perquè posi etiquetes a les dues imatges generades, això es per poder avaluar aquestes etiquetes amb la funció de pèrdua, per després poder agafar la diferència i d'aquesta manera calcular la derivada. El codi per el generador es pot trobar en l'apèndix, E.1.0.3.

---

<sup>8</sup>Cada un d'aquests vectors de paràmetres té una petita variació en un paràmetre, i cada vector té una variació en un sentit.

---

```

 $\nabla = 0$                                 ▷ Creació del vector  $\nabla$  que té la mateixa mida que  $\theta$ 
for soroll en batch do
  for paràmetre  $\theta_i$  en  $\theta$  do
     $\Delta_i = 0$                                 ▷ Creació del vector perturbació d'acord amb el vector  $\theta_i$ 
     $\theta_i^+ = \theta + \frac{\pi}{4} \Delta_i$ 
     $\theta_i^- = \theta - \frac{\pi}{4} \Delta_i$ 

     $\text{Imatge}_1 = \text{generador}(\text{soroll}, \theta_i^+)$ 
     $\text{Imatge}_2 = \text{generador}(\text{soroll}, \theta_i^-)$                                 ▷ Generació de les imatges

     $\text{Predicció}_1 = \text{discriminador}(\text{Imatge}_1)$ 
     $\text{Predicció}_2 = \text{discriminador}(\text{Imatge}_2)$                                 ▷ El discriminador posa una
    etiqueta a cada imatge

     $\text{Diferencia}_i = \mathcal{L}(\text{Predicció}_1) - \mathcal{L}(\text{Predicció}_2)$ 

     $\nabla_\theta = \nabla_i + \text{Diferencia}_i$                                 ▷ On  $\mathcal{L}$  és la funció de pèrdua del generador
  end for
end for

for  $\theta_i$  en  $\theta$  i  $\nabla_i$  en  $\nabla_\theta$  do                                ▷ Per a cada paràmetre i per a cada error
   $\theta_i^{t+1} = \theta_i + \frac{\eta}{M} \nabla_i$  ▷ Actualització del paràmetre amb la mitjana de l'error que
  correspon a aquest paràmetre
end for

```

---

FIGURA 6.1: **Pseudocodi per una xarxa generativa adversativa.** Em refereixo al input de la xarxa generacional com a soroll, ja que és més encertat d'aquesta manera. El vector  $\nabla$  té la mateixa mida que el vector de paràmetres  $\theta$ , cal notar que també té la mateixa mida els vectors  $\theta_i^\pm$ , degut a que aquests vectors són  $\theta$  però amb una alteració al paràmetre  $\theta_i$ . Les paraules *generador* i *discriminador* denoten els respectius models, per tant,  $\text{Imatge}_i$  i  $\text{Predicció}_i$  són els outputs dels models.



## 6.3 Creació del model

Quan ja es tenen les dues parts del model, aquestes s'han d'ajuntar d'alguna manera. El que jo he fet es posar-ho tot en una classe de Python anomenada `Quantum_GAN`, que conté les funcions per definir el model, per executar-lo, per guardar les seves dades i per crear els gràfics que serveixen per avaluar l'eficiència del model. No fa falta entrar en detall sobre aquesta part en particular del codi. Només cal mencionar que és el tros de codi que junta tot, tant el discriminador i el generador, i les altres funcions que són necessàries, com per exemple la sigmoide. Aquestes funcions que no estan tant en l'arxiu del generador com el discriminador es troben en `functions.py`.

Tanmateix s'han de tenir en compte algunes de les funcionalitats d'aquesta classe, com la creació de les gràfiques i com circulen les dades del generador al discriminadors. D'aquesta última qüestió parlaré a continuació.

A la funció de la classe `Quantum_GAN` que s'utilitza per entrenar, `Quantum_GAN.train()` se li dona, entre altres inputs, un dataset amb imatges reals i el soroll per poder entrenar el generador. Aquest dataset es divideix en grups d'imatges (els *batches*), cada *batch* conté tant imatges reals, com soroll en la mateixa quantitat.


En una iteració primer s'optimitza el generador, que substitueix el soroll del *batch* amb les imatges que genera. Llavors es passa el *batch* al discriminador que s'optimitza tant amb les imatges reals com amb les imatges falses. Aquest és el procés per el qual s'optimitzen els dos models. No obstant, aquesta funció dona a terme altres coses que són necessàries per la creació dels gràfics, per exemple, selecciona una imatge real aleatòria i una de falsa per avaluar la funció de pèrdua en l'iteració, també crea les etiquetes utilitzades en aquesta l'avaluació. Totes aquestes dades s'utilitzen per la creació de diversos gràfics que mostren l'evolució d'aquestes dades a través de tota l'optimització.

## 6.4 Execució del model

No obstant en l'arxiu `qgan.py`, no es troba l'execució del model, només està la definició de la classe `Quantum_GAN`. Això és perquè el codi realment s'executa

en l'arxiu `main.py`<sup>9</sup>. Aquest és l'únic arxiu que té codi que realment s'executa, els altres arxius només tenen definicions. Per projectes relativament grans, com aquest, convé tenir un arxiu que fa tot el treball, el qual crida a totes les funcions definides en altres arxius i les executa d'una manera ordenada.

Com es pot veure en l'apèndix, [E.1.0.5](#), aquesta arxiu té molt poc codi. En ell només es crea amb Numpy el dataset, es defineix el discriminador i el generador, amb els quals es crea el model amb `Quantum_GAN`. Finalment es crida a la funció `Quantum_GAN.train()` per optimitzar el model. Al acabar l'optimització es criden les funcions `Quantum_GAN.plot()`, `Quantum_GAN.create_gif()` i `Quantum_GAN.save()`, les quals donen a terme funcions complementàries com la creació de les gràfiques, la creació d'un GIF que mostra com les imatges van evolucionant al llarg de l'optimització, i l'emmagatzematge de les dades rellevants que s'han creat durant l'optimització.

Les imatges que vaig escollir per generar són les mateixes que van generar en l'article en el qual vaig basar el treball. No entraré molt en detall sobre les distribucions concretes que formen les imatges, però per tindre una imatge general sobre com són, es pot veure la figura. 

falta la figura

Si executen el model, amb un mida de batch de 10, tant el *learning rate* del discriminador com del generador a 0.1, i un total de iteracions de 400, hi ha una garantia d'arribar a la convergència desitjada, es dir que, que els dos models arriben al equilibri de Nash i no poden continuar l'optimització. En aquest punt és quan les imatges falses i les reals són iguals.

---

<sup>9</sup>Veure apèndix [E.1.0.5](#)

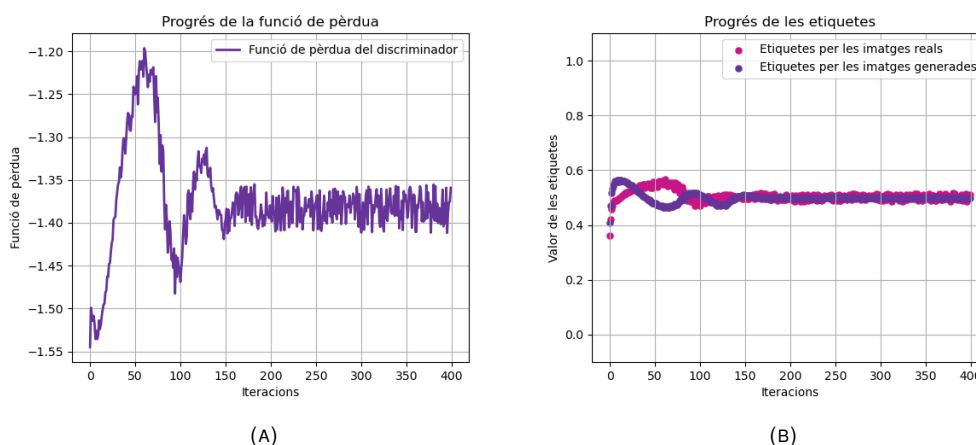


FIGURA 6.2: En pot veure com per l'iteració 250, el model ja s'ha estabilitzat, ja que les etiquetes i el valor de la funció de pèrdua convergeixen en un valor. **A)** Funció de pèrdua per cada iteració. A partir de l'iteració 175, es pot veure com el valor de la funció s'estabilitza en l'interval  $(-1.35, -1.4)$ , això concorda amb els valors de les etiquetes en les mateixes iteracions, ja que  $\log(\frac{1}{2}) + \log(1 - \frac{1}{2}) \simeq -1.38$ . **B)** Etiquetes per les imatges reals i generades per cada iteració. Es pot observar que els valors de les etiquetes per les imatges reals són més inestables que els valors de les generades. Això es perquè les imatges reals tenen una major variació en els valors dels píxels, mentre que en les generades aquest fet no es tan notable. Per tant el discriminador assigna etiquetes amb una major variació.

En la figura, 6.2b, es pot veure com les etiquetes pels dos tipus d'imatges van oscil·lant fins a estabilitzar-se. El mateix es pot dir per la funció de pèrdua, com es pot veure en la figura 6.2a.

Degut a que aquests gràfics són molt semblants als seus anàlegs de les GANs clàssiques i que les distribucions reals i generades són pràcticament iguals, com es pot veure a la figura 6.3, considero que el model funciona correctament.

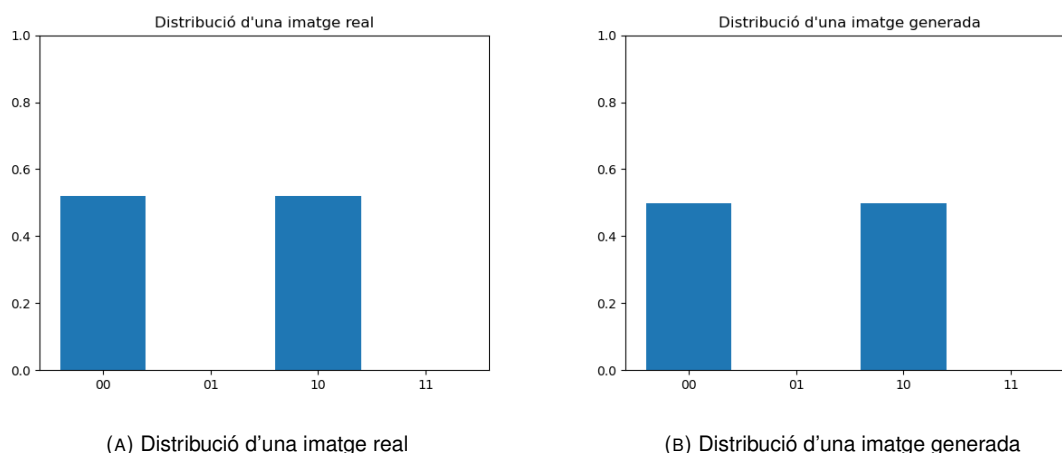


FIGURA 6.3: Comparació d'una imatge generada i una real, quan el model ha assolit la convergència. En el eix Y es pot veure el valor d'un píxel, mentre que en l'eix X estan els píxels.

Abans he especificat que al cap de 400 iteracions podem tenir la garantia d'arribar al punt d'equilibri, no obstant, es pot arribar a aquest punt amb menys iteracions, el que vull dir es que amb 400 de segur que s'arriba. Això és perquè, com a tots els model de *machine learning* hi ha una part de sort implicada, si els paràmetres inicials són més semblants als paràmetres desitjats, el model assolirà la convergència més ràpidament.

# Capítol 7

## Realització del experiment

Una vegada havia confirmat el correcte funcionament del model, vaig alterar el generador per poder acomodar els dos tipus de circuits quàntics que necessitava, uns amb un sistema ancilla, i uns altres sense.

Un sistema ancilla, són un grup de qubits sobre els quals es donen a terme operacions però que no es mesuren per treure el output del circuit. Aquests qubits es pot veure molt clar en la figura, .

poner figura

Primer de tot he de mencionar que havia de fer alguns canvis al generador per acomodar aquests qubits ancilla.

Segon, cal notar que no he seguit exactament els mateixos passos que en l'article original. En ell tenen aquesta equació [5]:

$$\rho = \frac{\text{tr}_A(\Pi_A |\psi\rangle \langle\psi|)}{\text{tr}(\Pi_A \otimes I_{2^N - 2^{N_A}} |\psi\rangle \langle\psi|)}$$

Com ja he comentat en la secció 2.1.1, no veig com aquesta equació pot tenir sentit, per tant la vaig ometre del meu experiment.

L'alternativa que he fet servir són els mesuraments de Qiskit, al definir el circuit específic quins són els qubits que no vull mesurar, que són els qubits que formen part del sistema ancilla. No obstant, no sé exactament que és lo que fa

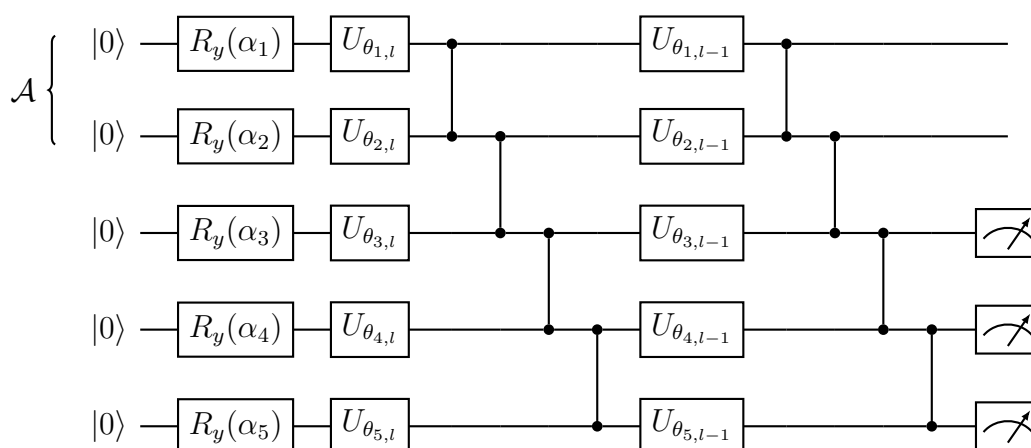


FIGURA 7.1: Aquí he marcat els qubits ancilla amb  $\mathcal{A}$ , són els dos primers. També al final del circuit he afegit mesures als qubits que s'han de mesurar.

Qiskit amb el mesurament. Però si empro aquest procediment en altres circuits, dels quals n'en se el resultat, aquest mètode fa el que m'espero<sup>1</sup>.

L'arxiu que utilitzo per fer els experiments és `experiment.py`. En ell es pot veure com defineixo dos discriminadors<sup>2</sup> i dos generadors, que es diferencien per els circuits que utilitzen, un amb qubits ancilla i l'altre sense. Aquests models s'agrupen en parelles per definir dues qGANs.

Cal notar que els generadors i els discriminadors<sup>3</sup> comencen amb els mateixos paràmetres, per tant en exactament les mateixes condicions, menys els circuits quàntics es clar. També s'utilitza el mateix dataset i altres hiperparàmetres.

## 7.1 Anàlisi dels resultats

Si comparem les gràfiques que mostren les etiquetes es pot veure una clara diferencia: Els models que tenen els circuits amb els qubits ancilla són més inestables que el que no els tenen. Això no vol dir necessàriament que siguin més eficients però l'estabilitat és un factor que es busca en les GANs.

<sup>1</sup>Parlo del parells de Bell, els circuits quàntics amb entrellaçament més simples que es poden fer.

<sup>2</sup>Que tenen les mateixes característiques.

<sup>3</sup>Al principi pensava no tenir els mateixos paràmetres inicials pels discriminadors, però al veure les gràfiques de les etiquetes, l'efecte que tenen aquests és molt notable. Es pot veure com a vegades les etiquetes comencen en uns valors de 1 i en altres de 0. [link per veure una imatge amb totes les gràfiques](#) (perdó per l'informalitat)

També es pot observar una clara diferència en les imatges generades en la primera iteració. En les primeres imatges d'un generador amb la funció no-lienal es pot veure un píxel amb un valor de 1 mentre que els altres estan al 0. Mentre que en l'altre tipus de generador es pot veure que els píxels tenen més o menys el mateix valor, d'aquesta manera formant una distribució uniforme. Aquest fet probablement es causat per l'estructura del circuit que té els qubits ancilla. No obstant, perquè exactament passa això està fora dels meus coneixements sobre la matèria<sup>4</sup>.

La part que realment m'intriga es que a partir de les primeres iteracions, en els circuits amb els qubits ancilla. El píxel que té el valor de 1, passa d'un a un altre. Aquest comportament succeeix a absolutament totes les vegades que he executat el codi. Al igual que he dit abans, arribar a comprendre això està fora del meu nivell de coneixement.

Estic bastant segur que aquesta fluctuació del píxels és el factor que causa l'inestabilitat que es pot observar en les etiquetes

Després de mirar les gràfiques que generaven els models, vaig posar-me a redactar les conclusions, no obstant, no estava satisfet amb la precisió de l'anàlisi de les dades. No podia saber amb certesa quin dels dos tipus de models era el més eficient. Necessitava una mètrica que en digues quina és la semblança entre les imatges generades i les imatges reals. Sabia que en l'article en el que he basat el treball els autors havien fet servir una mètrica anomenada *Férchet Score* o puntuació de *Férchet*, en la qual s'empleava la distància de *Férchet*. Aquesta distància serveix per comparar dues distribucions a partir de la seva mitjana i la seva covariància.

Llavors vaig decidir implantar aquesta mètrica, i veure com evoluciona al llarg de l'optimització.

En un article sobre l'avaluació de les imatges generades per les GAN [40], vaig trobar aquesta equació per calcular la distància de *Férchet*:

$$FD(r, g) = |\mu_r - \mu_g|^2 + \text{tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{\frac{1}{2}}\right)$$

---

<sup>4</sup>Saber com es comporten sistemes quàntics que tenen parts entrelaçades com aquest està fora del meu àmbit.

On  $\mu$  és la mitjana<sup>5</sup> i  $\Sigma$  és la covariància d'una distribució. Aquesta equació l'he implementat en Python mitjançant matrius, es a dir les imatges. Només em feia falta trobar una funció per poder calcular la covariància d'una matriu, afortunadament Numpy en té una, on cada fila representa una variable d'una distribució. No puc estar 100% segur de que he implementat aquesta mètrica correctament, degut a que no sé el funcionament exacte de la covariància, ni de la funció de Numpy. No obstant, sembla que funciona correctament.

Cal notar que si les dues imatges són exactament iguals, la distancia dona pràcticament zero<sup>6</sup>. Per considerar que dues imatges són «acceptablement semblants», han de tenir una distancia entre elles de  $10^{-3}$  aproximadament, i ha resultat molt útil.

---

<sup>5</sup>Jo he utilitzat la mitjana aritmètica.

<sup>6</sup>Python diu que és d'un ordre de magnitud de  $10^{-16}$  aproximadament.



## **Part III**

# **Conclusions**

	Presenta oscil·lació	No Presenta oscil·lació
Amb Funció No-Lineal	0	6
Sense Funció No-Lineal	5	1

TAULA 7.1: Les dades provenen d'un total de 6 model, 3 d'ells amb un total de 700 epoch i els altres 5 amb un total de 550. El nombre d'iteracions no hauria d'afectar de cada manera les dades. Degut si hi ha una oscil·lació, es pot veure clarament a partir de les 400 iteracions. Amb les dades es pot veure que és més probable que un model sense la funció lineal presenti una oscil·lació. Cal notar que cap model amb la funció ha tingut una oscil·lació. Les gràfiques que corresponen a cada model es poden veure en la figura [7.1](#).

Una vegada havia implementat la distància de Férchet per poder avaluar el rendiment dels models, vaig arribar a una clara conclusió:

Els models que tenen implementada la funció no-lienal són més eficients.

Arribo a aquesta conclusió degut a que els models sense la funció no arriben a estabilitzar-se, es a dir, no arriben al punt d'equilibri. En canvi, els models que si la presenten si ho fan. Arribar al punt d'equilibri ens garanteix que les imatges que generen siguin òptimes indefinidament. Per molt més que el model segueixi optimitzat-se, sempre donarà els mateixos resultats.

Els models que no presenten la funció, no assoleixen el punt d'equilibri, degut a que la fidelitat de les imatges oscil·la. Es a dir, poden arribar a generar imatges correctament, però no ho fan indefinidament, al cap d'unes interaccions la semblança de les imatges generades amb les real no es significativa.

Aquest problema no es causat pel discriminador, això es veritat perquè la variable independent dels experiment, és el circuit quàntic del generador. Per tant és el factor que té més possibilitat de ser el causant d'aquest comportament. No sé molt bé la causa exacte, però està clar que el mesurament parcial té un impacte. Tot i que hi han excepcions, a vegades els models sense la funció no experimenten aquesta oscil·lació. Però cal remarcar que les vegades que passa són notablement més altes que les que no passa. Com es pot veure amb les dades de la taula [7.1](#)

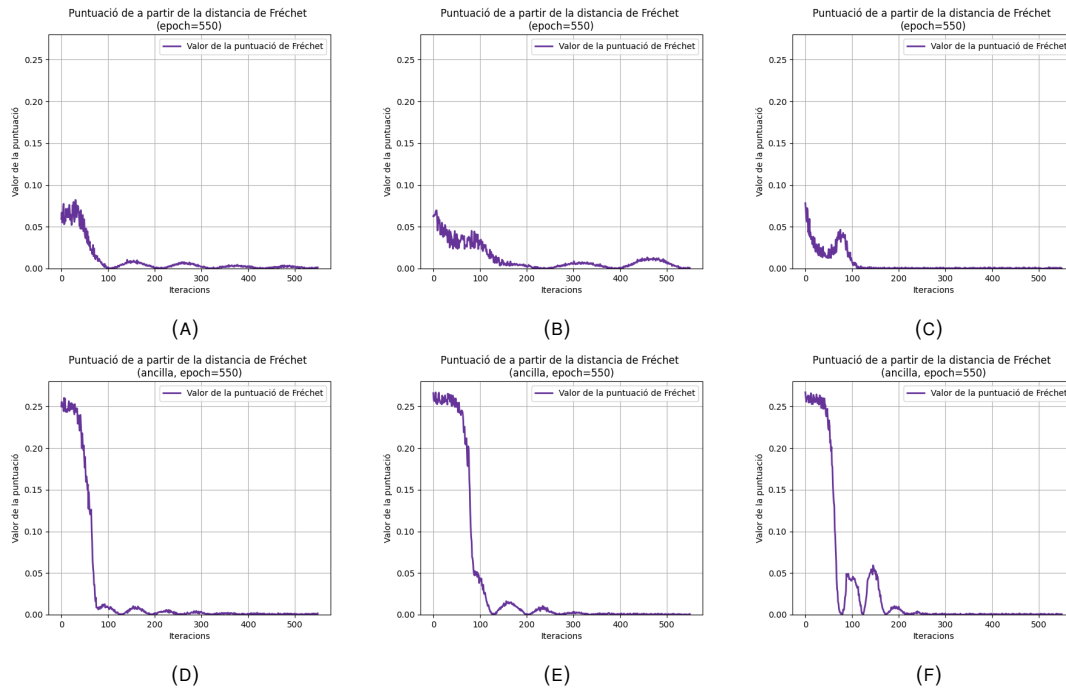


FIGURA 7.2: Totes les gràfiques corresponen a models que s'ha executat al llarg de 550 iteracions. Les figures **A**, **B** i **C**, corresponen a models sense la funció lineal. L'únic d'ells que no presenta una oscil·lació és el **C**. Les gràfiques sense la funció es poden comparar a les d'abaix, les quals representen models amb la funció implementada. Els models han estat creats per parelles, les quals estan organitzades verticalment. Es a dir, les gràfiques **A** i **D** representen models que tenen els mateixos paràmetre inicials. El mateix passa amb **B** i **E** i amb **C** i **F**.

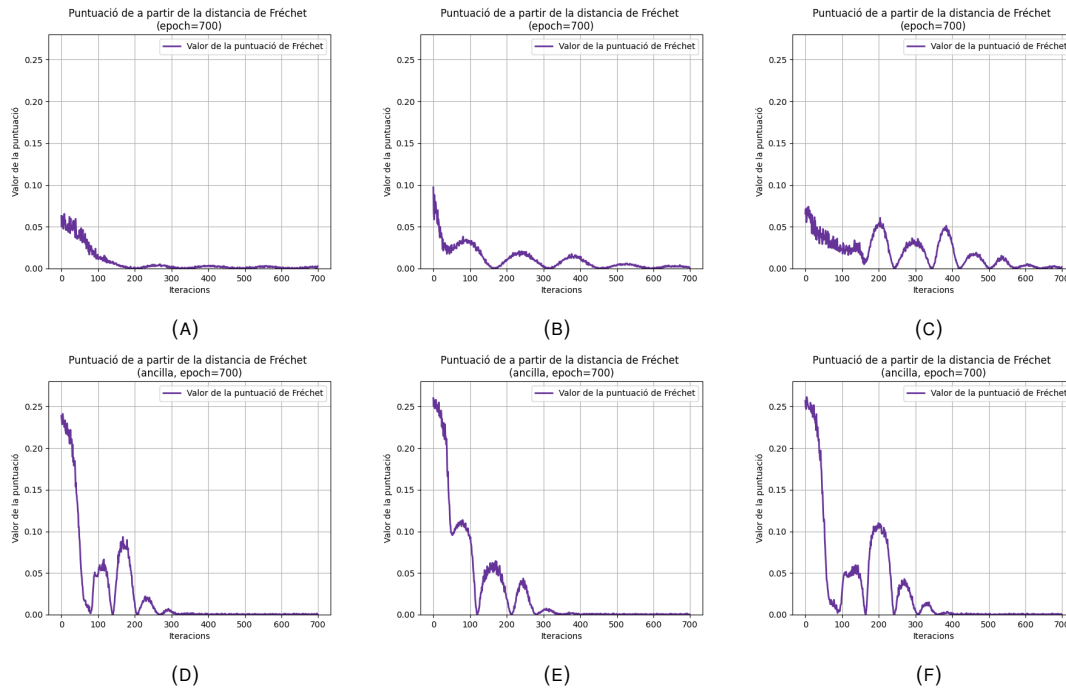


FIGURA 7.3: Aquestes gràfiques corresponen a models que s'han executat al llarg de 700 iteracions. Estan organitzades igual que les gràfiques de la figura 7.2. En aquests casos, com es pot observar tots els models sense la funció no-lineal presenten les oscil·lacions. No obstant en la gràfica **A**, aquesta es molt feble. Per veure com afecten les oscil·lacions es pot veure la figura, on estan representades les últimes imatges que han generat els models que corresponen les gràfiques d'aquesta figura.

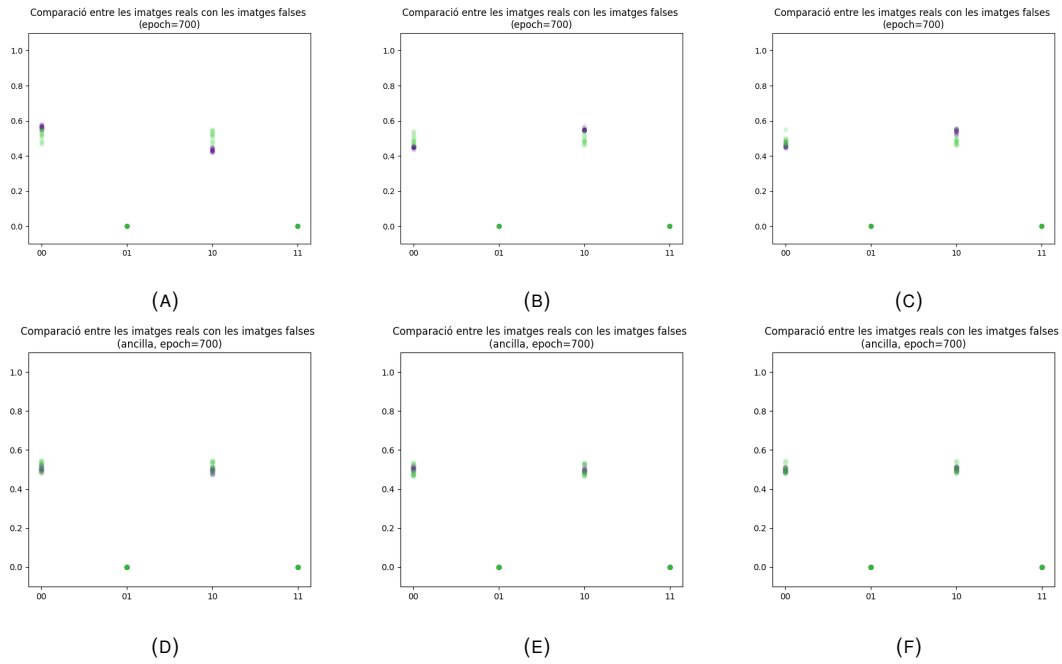


FIGURA 7.4: Aquestes gràfiques corresponen als mateixos models que els de la figura 7.3. Les posicions de les gràfiques són les mateixes, per tant si estan en la mateixa posició que les de l'altra figura, corresponen al mateix model. Les imatges generades pels models sense la funció no-lienal (**A**, **C**, **B**) no s'assemblen a les reals. Es pot veure com els punts de les imatges generades (color violeta) no estan als mateixos valors. Mentre que en les gràfiques **D**, **E** i **F**, sí que ho estan. Els punts violetes sembla que representen la mitjana dels punts verds. També es pot observar que les imatges generades tendeixen a ser més variades que les reals.

## **Part IV**

## **Apèndix**

# **Appendices**

# Apèndix A

## Més àlgebra lineal

Ja he escrit bastants pàgines sobre àlgebra lineal, però aparentment no eren les suficients perquè estic content amb el treball<sup>1</sup>, així que aquí hi ha més àlgebra lineal.

### A.1 Procediment de Gram–Schmidt

El procediment de Gram-Schmidt és un mètode utilitzat per produir bases per a espais vectorials[9]. Per un espai  $V$  amb producte interior de  $d$  dimensions amb el set de vectors  $|v_1\rangle, \dots, |v_d\rangle$ , podem definir una nova base de vectors ortonormals  $\{|u\rangle\}$ . El primer element d'aquest set és  $|u_1\rangle = |v_1\rangle / \||v_1\rangle\|$ , amb el següent element  $|v_{k+1}\rangle$  sent:

$$|u_{k+1}\rangle = \frac{|v_{k+1}\rangle - \sum_{i=1}^k \langle u_i | v_{k+1} \rangle |u_i\rangle}{\left\| |v_{k+1}\rangle - \sum_{i=1}^k \langle u_i | v_{k+1} \rangle |u_i\rangle \right\|}$$

Per  $k$  en el interval  $1 \leq k \leq d-1$ .

Si seguim per  $k$  en  $1 \leq k \leq d-1$ , obtenim el set de vectors  $|u_1\rangle, \dots, |u_d\rangle$  que es una base vàlida per l'espai ortonormal per l'estai  $V$ . Els vectors creats

---

<sup>1</sup>Hi han moltes coses guays i interessants que vull explicar.



han de tindre el mateix  $\text{span}^2$  que el dels vectors que originalment eren la base per  $V$ :

$$\text{span}(\{|v\rangle\}) = \text{span}(\{|v\rangle\}) = V$$

Cal notar que l'span de del set base és la definició del espai. En altres paraules, cada vector en  $V$  pot ser representat per una combinació del vectors base.

La prova de que és una base ortonormal és bastant simple: Podem veure immediatament que els elements de  $\{|u\rangle\}$  són vectors unitaris perquè estan normalitzats (els vectors  $|v_{k+1}\rangle - \sum_{i=1}^k \langle u_i | v_{k+1} \rangle |u_i\rangle$  estan dividits per la seva norma). També podem veure que són ortogonals els uns als altres mirant que el producte interior entre els doni 0:

Per  $k = 1$ :

$$|u_2\rangle = \frac{|v_2\rangle - \langle u_1 | v_2 \rangle |u_1\rangle}{\| |v_2\rangle - \langle u_1 | v_2 \rangle |u_1\rangle \|}$$

Per tant el producte interior amb  $|v_1\rangle$  és:

$$\begin{aligned} \langle u_1 | u_2 \rangle &= \langle u_1 | \left( \frac{|v_2\rangle - \langle u_1 | v_2 \rangle |u_1\rangle}{\| |v_2\rangle - \langle u_1 | v_2 \rangle |u_1\rangle \|} \right) \\ &= \frac{\langle u_1 | v_2 \rangle - \langle u_1 | v_2 \rangle \langle u_1 | u_1 \rangle}{\| |v_2\rangle - \langle u_1 | v_2 \rangle |u_1\rangle \|} \\ &= 0 \end{aligned}$$

Per inducció podem veure que per  $j \leq d$ , amb  $d$  sent la dimensió del espai vectorial:

$$\begin{aligned} \langle u_j | u_{n+1} \rangle &= \langle u_j | \left( \frac{|v_{n+1}\rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle |u_i\rangle}{\| |v_{n+1}\rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle |u_i\rangle \|} \right) \\ &= \frac{\langle u_j | v_{n+1} \rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle \langle u_j | u_i \rangle}{\| |v_{n+1}\rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle |u_i\rangle \|} \\ &= \frac{\langle u_j | v_{n+1} \rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle \delta_{ij}}{\| |v_{n+1}\rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle |u_i\rangle \|} \\ &= \frac{\langle u_j | v_{n+1} \rangle - \langle u_j | v_{n+1} \rangle}{\| |v_{n+1}\rangle - \sum_{i=1}^n \langle u_i | v_{n+1} \rangle |u_i\rangle \|} \\ &= 0 \end{aligned}$$

Tot això no és veu molt clar al principi però cal recordar que el producte interior de dos vectors ortonormals és zero, i que el producte interior entre el mateix vector

<sup>2</sup>L'span d'un set de vectors són totes les combinacions lineals possibles amb aquests vectors.

unitari és un.

## A.2 Curs ràpid de la notació de Dirac

A la taula següent hi ha un resum de conceptes matemàtics de l'àlgebra lineal importants expressats en la notació de Dirac<sup>3</sup> [41]

Notació	Descripció
$z$	Nombre complex
$z^*$	Conjugat complex d'un nombre complex $z$ . $(a + bi)^* = (a - bi)$
$ \psi\rangle$	Vector amb una etiqueta $\psi$ . Conegut com <i>ket</i>
$ \psi\rangle^T$	Transposada d'un vector $ \psi\rangle$
$ \psi\rangle^\dagger$	Conjugat Hermitià d'un vector. $ \psi\rangle^\dagger = ( \psi\rangle^T)^*$
$\langle\psi $	Vector dual a $ \psi\rangle$ . $ \psi\rangle = \langle\psi ^\dagger$ i $\langle\psi  =  \psi\rangle^\dagger$ . Conegut com <i>bra</i>
$\langle\varphi \psi\rangle$	Producte interior dels vectors $\langle\varphi $ i $ \psi\rangle$
$ \varphi\rangle\langle\psi $	Producte exterior del vectors $\langle\varphi $ i $ \psi\rangle$
$ \psi\rangle \otimes  \varphi\rangle$	Producte tensorial del vectors $ \varphi\rangle$ i $ \psi\rangle$
$0$	Vector zero i operador zero
$\mathbb{I}_n$	Matriu identitat de dimensions $n \times n$
$\mathbb{C}_n$	Espai vectorial complex de dimensió $n$
$\mathbb{C}_1$ o $\mathbb{C}$	Espai del nombres complexos

<sup>3</sup>La notació utilitzada per un espai vectorial complex i l'espai dels nombres complex no són de la notació de Dirac estàndard, però les poso per explicar el que signifiquen.

## Apèndix B

# Computació Quàntica vs Mecànica Quàntica

En la introducció havia mencionat que una de les raons per les quals havia començat a aprendre i recercar sobre computació quàntica era perquè era fàcil, en aquest apèndix explicaré exactament a que em refereixo per això amb un exemple pràctic. Dic que és fàcil, perquè si ho és, si es compara la computació quàntica amb la mecànica quàntica. Presentaré un exemple pràctic per il·lustrar-lo.

En mecànica quàntica la manera més usual de representar els estats quàntics, com els orbitals d'un àtom d'hidrogen, és a través de *wavefunctions* o funcions d'ona, no es solen representar mitjançant vectors d'estat. Aquestes funcions d'ona són molt útils i poden representar casos més generals que els vectors d'estat. No obstant, treballar amb elles és molt més complicat degut a que són funcions que depenen del temps, no són vectors. Això implica que s'han d'utilitzar altres operacions per normalitzar les funcions, determinar com evolucionen en el temps o per fer mesures. A continuació faré una comparació entre normalitzar una funció d'ona i un vector d'estat.

## B.1 Normalitzar

Degut a l'interpretació probabilística dels vectors d'estat i de les funcions d'ona, aquests objectes han de ser normalitzats perquè la suma de la probabilitats del possibles estats de mesura sigui 1.

Per una funció d'ona  $\Psi(x, t)$  que representa una partícula, la probabilitat de trobar aquesta partícula en un punt  $x$  és  $|\Psi(x, t)|^2$ . Per tant, la funció d'ona ha de ser normalitzada seguint la fórmula:

$$\int_{-\infty}^{+\infty} |\Psi(x, t)|^2 dx = 1 \quad (\text{B.1})$$

Degut a que la funció d'ona evoluciona a través del temps d'acord amb l'equació de Schrödinger, veure la figura B.1, qualsevol solució d'aquesta equació ha d'estar també normalitzada. En altres paraules, aquesta equació ha de preservar la normalització de les funcions d'ona [42].

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \frac{\partial^2 \Psi}{\partial x^2} + V\Psi$$

FIGURA B.1: **Equació de Schrödinger.** On  $\hbar$  és  $\hbar/2\pi$ , i  $V$  és una funció potencial d'energia.

Podem provar que aquesta equació preserva la fórmula B.1, començant per la igualtat trivial:

$$\frac{d}{dt} \int_{-\infty}^{+\infty} |\Psi(x, t)|^2 dx = \frac{\partial}{\partial t} \int_{-\infty}^{+\infty} |\Psi(x, t)|^2 dx$$

Per la regla del producte tenim que <sup>1</sup>:

$$\frac{\partial}{\partial t} |\Psi|^2 = \frac{\partial}{\partial t} (\Psi \Psi^*) = \Psi^* \frac{\partial \Psi}{\partial t} + \frac{\partial \Psi^*}{\partial t} \Psi$$

Ara l'equació de Schrödinger diu que

$$\frac{\partial \Psi}{\partial t} = \frac{i\hbar}{2m} \frac{\partial^2 \Psi}{\partial x^2} - \frac{i}{\hbar} V\Psi$$

<sup>1</sup>A partir d'ara escriuré  $\Psi(x, t)$  simplement com  $\Psi$  per no fer les equacions tan enrevessades.

després calculant el complex conjugat tenim que

$$\frac{\partial \Psi^*}{\partial t} = -\frac{i\hbar}{2m} \frac{\partial^2 \Psi^*}{\partial x^2} + \frac{i}{\hbar} V \Psi^*$$

per tant

$$\frac{\partial}{\partial t} |\Psi|^2 = \frac{i\hbar}{2m} \left( \Psi^* \frac{\partial^2 \Psi}{\partial x^2} - \frac{\partial^2 \Psi^*}{\partial x^2} \Psi \right) = \frac{\partial}{\partial x} \left[ \frac{i\hbar}{2m} \left( \Psi^* \frac{\partial \Psi}{\partial x} - \frac{\partial \Psi^*}{\partial x} \Psi \right) \right]$$

finalment podem avaluar l'integral del principi:

$$\frac{d}{dt} \int_{-\infty}^{+\infty} |\Psi(x, t)|^2 dx = \frac{i\hbar}{2m} \left( \Psi^* \frac{\partial \Psi}{\partial x} - \frac{\partial \Psi^*}{\partial x} \Psi \right) \Big|_{-\infty}^{+\infty}$$

Degut a que  $\Psi(x, t)$  ha de convergir a zero quan  $x$  va cap a infinit, es veritat que:

$$\frac{d}{dt} \int_{-\infty}^{+\infty} |\Psi(x, t)|^2 dx = 0$$

Es pot veure que l'integral es constant i per tant quan  $\Psi$  es normalitzada a  $t = 0$ , es queda d'aquesta manera per qualsevol  $t$  (positiu es clar).

# Apèndix C

## Polarització d'un fotó

En l'equació 2.1 he excluit el concepte de fase, que determina el tipus de polarització que té un fotó. Hi han 3 tipus:

1. **Linear:** Un fotó té polarització lineal quan els angles de la fase  $\alpha_x, \alpha_y$  en els estats base  $|x\rangle, |y\rangle$  són iguals:

$$\begin{aligned} |\nearrow\rangle &= \cos(\theta)e^{i\alpha_x} |x\rangle + \cos(\theta)e^{i\alpha_y} |y\rangle \\ &= [\cos(\theta) |x\rangle + \sin(\theta) |y\rangle]e^{i\alpha} \end{aligned}$$

On  $\alpha = \alpha_x = \alpha_y$ .

2. **Circular:** Quan els angles  $\alpha_x, \alpha_y$  son separats per exactament  $\frac{\pi}{2}$  i la amplitud per les dos bases és la mateixa:

$$\begin{aligned} |\nearrow\rangle &= \frac{1}{\sqrt{2}} \cos(\theta)e^{i\alpha_x} |x\rangle \pm i \frac{1}{\sqrt{2}} \sin(\theta)e^{i\alpha_y} |y\rangle \\ &= [\cos(\theta)e^{i\alpha_x} |x\rangle \pm i \sin(\theta)e^{i\alpha_y} |y\rangle] \frac{1}{\sqrt{2}} \end{aligned}$$

On el signe  $\pm$  indica la diferencia entre la diferencia entre la polarització circular cap a la dreta o la esquerra, amb  $+$  i  $-$ , respectivament.

3. **El·líptica:** On els angles  $\alpha_x, \alpha_y$  son diferents per una quantitat arbitraria<sup>1</sup>:

$$|\nearrow\rangle = \cos(\theta)e^{i\alpha_x} |x\rangle + \sin(\theta)e^{i\alpha_y} |y\rangle$$

Aquest és el cas més general.

---

<sup>1</sup>Però que no sigui la quantitat que dona a terme la polarització circular.

## Apèndix D

# Complexitat i algoritmes quàntics

En ciència de la computació existeix el concepte de *Big-O Notation*, una forma d'expressar lo eficients que són els algoritmes per fer certes tasques, en altres paraules la complexitat dels algoritmes. Bàsicament es una forma de classificar-los segon la rapidesa que tenen en ver la tasca que els correspon, aquesta rapidesa no és mesura en segons, degut a que aquesta mètrica pot variar d'ordinador a ordinador per les diferencies en hardware que aquest poden tindre. En canvi es mesure en nombre d'operacions o temps directament, però sense unitats.

La *Big-O Notation* consisteixes en definir el temps màxim que necessita un algoritme, es denota com  $O(\cdot)$  on l'argument usualment depèn de  $n$  que és la mida del input al algoritme, per exemple un algoritme de cerca ha de cercar a través de  $n$  coses. Com a un exemple més concret tenim que un l'algoritme de cerca de cadenes binaries corre en un temps  $O(\log_2 n)$ , on  $n$  és el nombre de cadenes entre les quals ha de cercar. Recorda que la notació  $O(\cdot)$  és el màxim, es a dir es *upper bounded*, això significa que  $\log_2 n$  és la quantitat de temps més gran en la que es troba la cadena, també es possible que es trobi-s'hi a la primera comprovació que es va<sup>1</sup>, llavors l'algoritme acabaria en un temps  $O(1)$ . Simplement és una manera de mirar lo eficients que són els algoritmes en relació a la mida del input que tenen.

---

<sup>1</sup>Que la primera cadena que es cerca, és la que s'ha de trobar.



Amb aquesta notació tenim una manera de comparar la eficiència que tenen els algoritmes quàntics amb la del clàssics que tenen la mateixa funció. Per il·lustrar aquesta avantatge en eficiència que presenten els algoritmes quàntics, presentaré a continuació els dos algoritmes quàntics més famosos, el de Grover, i el de Shor. Que serveixen per la cerca d'un element en una llista desordenada, i per la factorització en nombres primers respectivament.

## D.1 Algoritme de Grover

Al 1996, Lov Grover va presentar un algoritme quàntic per cercar en dades desordenades [43] (e.g. cercar el número de telèfon d'una persona en una llista desordenada). Per aquest problema un algoritme clàssic té una complexitat de  $O(N)$  cerques<sup>2</sup>, mentre que l'algoritme de Grover té una complexitat de  $O(\sqrt{N})$ , sent substancialment més eficient. En les paraules de Grover [43] (adaptades), un ordinador clàssic per tindre un probabilitat de  $\frac{1}{2}$  de trobar el número de telèfon d'una persona en una llista desordenada necessita mirar a un mínim de  $\frac{N}{2}$  números, mentre que amb el seu s'obté el número de telèfon en només  $O(\sqrt{N})$  passos<sup>3</sup>.

L'algoritme funciona de la següent manera: Agafar un sistema de  $n$  qubits en l'estat  $|0\rangle$  que resulten en una combinació de  $N = 2^n$  estats. Aplicar una distribució uniforme als qubits, es a dir, aplicar portes Hadamard a tots els qubits, tenint al final un estat resultant  $|s\rangle$ :

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} |x\rangle$$

Repetir  $\approx \frac{\pi}{4}\sqrt{N}$  vegades, el operador de Grover  $G$ , que consisteix en el següent conjunt d'instruccions:

1: Aplicar el *Oracle*  $U_\omega$

---

<sup>2</sup>Una cerca és quan es verifica si un element de la llista és l'element que es cerca.

<sup>3</sup>Per passos entenc que es refereix al nombre de vegades que es mira al oracle, es a dir, el nombre que de vegades que es verifica si s'ha trobat l'element que es cerca.

- 2: Aplicar portes Hadamard a tots els qubits
- 3: Aplicar el *Grover diffusion operator*  $U_s = 2 |s\rangle \langle s| - I$
- 4: Aplicar una altre vegada les portes Hadamard

El *oracle* és un tipus de funció que s'utilitza en els algorismes de cerca, té la finalitat de reconèixer si un element és l'element que s'està cercant. En els algorismes de cerca es construeixen al voltant de l'*oracle*, diversos algorismes tenen diverses formes de consultar al oracle. A més a més, la quantitat de consultes a l'*oracle* serveixen per quantificar la complexitat del algoritme, a més consultes, més ineficient és l'algoritme.

Al veure el procediment de l'algoritme es pot veure l'*oracle* es consulta aproximadament  $\frac{\pi}{4}\sqrt{N}$ , vegades, d'aquesta manera resultat en una complexitat aproximada de  $O(\sqrt{N})$ , on  $N = 2^n$  per  $n$  qubits, com ja he esmentat anteriorment.

No entraré més en profunditat sobre aquest algoritme, perquè sinó hauré d'introduir més conceptes de computació quàntica com ara la fase d'un estat, o començar a utilitzar notació més complicada que també hauré d'explicar. Em sap greu, perquè és un algoritme que funciona d'una manera bonica de veure. També em sembla el millor algoritme per poder explicar una de les millors característiques de la computació quàntica, tenir algorismes que et descarten automàticament els estats dolents dintre d'una combinació d'estats, d'aquesta manera lliurant un estat concret desitjat de entre la combinació. En el cas de l'algoritme de Grover aquest estat desitjat, és l'estat que està cercant l'*oracle*.

## D.2 Algoritme de Shor

No explicaré aquest algoritme en profunditat degut a que és molt complex<sup>4</sup>, no surt ni en els llibres de texts als quals tinc accés. Simplement en limitaré a explicar quina és la seva utilitat i el compararé amb les seves alternatives clàssiques.

L'algoritme de Shor per la factorització en nombres primers [44], és probablement l'algoritme quàntic més famós i més rellevant que hi ha.

---

<sup>4</sup>Hauria d'introduir nous conceptes com la fase d'un qubit i com efectuar la transformada de Fourier inversa en un circuit quàntic.

# Apèndix E

## Codi

En l'apèndix actual presentaré el codi que he utilitzat al llarg del treball. Està organitzat segons el moment en el qual he referenciat el codi en el text.

IMPORTANT: Si hi ha text en català dintre del codi (n'hi ha en forma de comentaris), no porta accents perquè els paquet que utilitzo per formateixar el codi d'aquesta manera dintre del document no els accepta. Però si es mira el codi en el [repositori del treball](#) es pot veure que si que té accents. No em deixa posar ni accents, ni la ce trencada, ni la ele geminada. L'error diu que no són caràcters que estan en Unicode UTF-8, quan si que estan allà, per fet que és un codi que suposadament està internacionalitzat per tots el llenguatges que s'utilitzen 'moderadament'.

### E.1 Part I

#### E.1 Capítol 3

**E.1.0.1 Regressió lienal** Codi per efectuar una regressió lineal a dades que es generen al atzar en el mateix arxiu, l'utilitzo per poder generar una gràfic per il·lustrar un exemple de regressió lienal. Aquest tros de codi l'he tret de GitHub<sup>1</sup>.

---

<sup>1</sup>Gist realitzat per l'usuari *jimimvp*: [link](#)

```

1  import numpy as np
2  from matplotlib import pyplot as plt
3  import matplotlib
4
5  font = {'family' : 'Helvetica',
6         'size'    : 18}
7
8  matplotlib.rc('font', **font)
9
10 # generate the data
11 np.random.seed(222)
12 X = np.random.normal(0,1, (200, 1))
13 w_target = np.random.normal(0,1, (1,1))
14 # data + white noise
15 y = X@w_target + np.random.normal(0, 1, (200,1))
16
17 # least squares
18 w_estimate = np.linalg.inv(X.T@X)@X.T@y
19 y_estimate = X@w_estimate
20
21 # plot the data
22 plt.figure(figsize=(15,10))
23 plt.scatter(X.flat, y_estimate.flat, label="Predicció")
24 plt.scatter(X.flat, y.flat, color='red', alpha=0.4, label="Dades")
25 plt.tight_layout()
26 plt.title("Regressió per diferencia de quadrats")
27 plt.legend()
28 plt.savefig("least_squares.png")
29 plt.show()

```

LISTING E.1: Regressió lineal

## E.2 Part II

### E.1 Capítol 6

**E.1.0.1 Codi original per la xarxa neuronal clàssica** Està extret del [repositori de GitHub de Micheal Nielsen](#), concretament del arxiu `network.py`.

```

1  """
2  network.py
3  ~~~~~
4  A module to implement the stochastic gradient descent learning

```

```

5 algorithm for a feedforward neural network. Gradients are calculated
6 using backpropagation. Note that I have focused on making the code
7 simple, easily readable, and easily modifiable. It is not optimized,
8 and omits many desirable features.
9 """
10
11 ##### Libraries
12 # Standard library
13 import random
14
15 # Third-party libraries
16 import numpy as np
17
18 class Network(object):
19
20     def __init__(self, sizes):
21         """The list 'sizes' contains the number of neurons in the
22         respective layers of the network. For example, if the list
23         was [2, 3, 1] then it would be a three-layer network, with the
24         first layer containing 2 neurons, the second layer 3 neurons,
25         and the third layer 1 neuron. The biases and weights for the
26         network are initialized randomly, using a Gaussian
27         distribution with mean 0, and variance 1. Note that the first
28         layer is assumed to be an input layer, and by convention we
29         won't set any biases for those neurons, since biases are only
30         ever used in computing the outputs from later layers."""
31         self.num_layers = len(sizes)
32         self.sizes = sizes
33         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
34         self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes
35 [1:])]
36
37     def feedforward(self, a):
38         """Return the output of the network if 'a' is input."""
39         for b, w in zip(self.biases, self.weights):
40             a = sigmoid(np.dot(w, a)+b)
41         return a
42
43     def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
44         """Train the neural network using mini-batch stochastic
45         gradient descent. The 'training_data' is a list of tuples
46         '(x, y)' representing the training inputs and the desired
47         outputs. The other non-optional parameters are
48         self-explanatory. If 'test_data' is provided then the
49         network will be evaluated against the test data after each
50         epoch, and partial progress printed out. This is useful for
51         tracking progress, but slows things down substantially."""

```

```

51     if test_data: n_test = len(test_data)
52     n = len(training_data)
53     for j in xrange(epochs):
54         random.shuffle(training_data)
55         mini_batches = [training_data[k:k+mini_batch_size] for k in xrange(0, n
, mini_batch_size)]
56         for mini_batch in mini_batches:
57             self.update_mini_batch(mini_batch, eta)
58         if test_data:
59             print "Epoch {0}: {1} / {2}".format(j, self.evaluate(test_data),
n_test)
60         else:
61             print "Epoch {0} complete".format(j)
62
63     def update_mini_batch(self, mini_batch, eta):
64         """Update the network's weights and biases by applying
65         gradient descent using backpropagation to a single mini batch.
66         The 'mini_batch' is a list of tuples '(x, y)', and 'eta'
67         is the learning rate."""
68         nabla_b = [np.zeros(b.shape) for b in self.biases]
69         nabla_w = [np.zeros(w.shape) for w in self.weights]
70         for x, y in mini_batch:
71             delta_nabla_b, delta_nabla_w = self.backprop(x, y)
72             nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
73             nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
74             self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights,
nabla_w)]
75             self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases,
nabla_b)]
76
77     def backprop(self, x, y):
78         """Return a tuple '(nabla_b, nabla_w)' representing the
79         gradient for the cost function C_x. 'nabla_b' and
80         'nabla_w' are layer-by-layer lists of numpy arrays, similar
81         to 'self.biases' and 'self.weights'."""
82
83         nabla_b = [np.zeros(b.shape) for b in self.biases]
84         nabla_w = [np.zeros(w.shape) for w in self.weights]
85         # feedforward
86         activation = x
87         activations = [x] # list to store all the activations, layer by layer
88         zs = [] # list to store all the z vectors, layer by layer
89         for b, w in zip(self.biases, self.weights):
90             z = np.dot(w, activation)+b
91             zs.append(z)
92             activation = sigmoid(z)
93             activations.append(activation)

```

```

94
95     # backward pass
96     delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
97     nabla_b[-1] = delta
98     nabla_w[-1] = np.dot(delta, activations[-2].transpose())
99     # Note that the variable l in the loop below is used a little
100    # differently to the notation in Chapter 2 of the book. Here,
101    # l = 1 means the last layer of neurons, l = 2 is the
102    # second-last layer, and so on. It's a renumbering of the
103    # scheme in the book, used here to take advantage of the fact
104    # that Python can use negative indices in lists.
105    for l in xrange(2, self.num_layers):
106        z = zs[-l]
107        sp = sigmoid_prime(z)
108        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
109        nabla_b[-l] = delta
110        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
111    return (nabla_b, nabla_w)
112
113    def evaluate(self, test_data):
114        """Return the number of test inputs for which the neural
115        network outputs the correct result. Note that the neural
116        network's output is assumed to be the index of whichever
117        neuron in the final layer has the highest activation."""
118        test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data
119        ]
120
121        return sum(int(x == y) for (x, y) in test_results)
122
123    def cost_derivative(self, output_activations, y):
124        """Return the vector of partial derivatives \partial C_x /
125        \partial a for the output activations."""
126        return (output_activations - y)
127
128    ##### Miscellaneous functions
129    def sigmoid(z):
130        """The sigmoid function."""
131        return 1.0/(1.0+np.exp(-z))
132
133    def sigmoid_prime(z):
134        """Derivative of the sigmoid function."""
135        return sigmoid(z)*(1-sigmoid(z))

```

LISTING E.2: Codi original per la xarxa neuronal clàssica

**E.1.0.2 Codi final per el discriminador** Aquest codi es pot trobar al [repositori del treball](#) en l'arxiu `discriminator.py`.

```

1  """DISCRIMINATOR"""
2  import json
3  from typing import Dict, List
4
5  import numpy as np
6
7  from quantumGAN.functions import BCE_derivative, minimax_derivative_fake,
    minimax_derivative_real, sigmoid, sigmoid_prime
8
9
10 def load(filename):
11     f = open(filename, "r")
12     data = json.load(f)
13     f.close()
14     # cost = getattr(sys.modules[__name__], data["cost"])
15     net = ClassicalDiscriminator_that_works(data["sizes"], data["loss"])
16     net.weights = [np.array(w) for w in data["weights"]]
17     net.biases = [np.array(b) for b in data["biases"]]
18     return net
19
20
21 class ClassicalDiscriminator:
22
23     def __init__( self,
24                   sizes: List[int],
25                   type_loss: str) -> None:
26
27         self.num_layers = len(sizes)
28         self.sizes = sizes
29         self.type_loss = type_loss
30         self.data_loss = {"real": [], "fake": []}
31         self.ret: Dict[str, any] = {"loss": [],
32                                     "label real": [],
33                                     "label fake": [],
34                                     "label fake time": [],
35                                     "label real time": []}
36         self.biases = [np.random.randn(y, ) for y in sizes[1:]]
37         self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes
38                                                                [1:])]
39
40     def feedforward(self, a):
41         """Return the output of the network if 'a' is input."""
42         for b, w in zip(self.biases, self.weights):
43             a = sigmoid(np.dot(w, a) + b)
44         return a
45
46     def predict(self, x):

```



```

46     # feedforward
47     activation = x
48     zs = [] # list to store all the z vectors, layer by layer
49     for b, w in zip(self.biases, self.weights):
50         z = np.dot(w, activation) + b
51         zs.append(z)
52         activation = sigmoid(z)
53     return activation
54
55     def evaluate(self, test_data):
56         test_results = [(np.argmax(self.feedforward(x)), y)
57             for (x, y) in test_data]
58         return sum(int(x == y) for (x, y) in test_results)
59
60
61     def forwardprop(self, x: np.ndarray):
62         activation = x
63         activations = [x] # list to store all the activations, layer by layer
64         zs = [] # list to store all the z vectors, layer by layer
65         for b, w in zip(self.biases, self.weights):
66             z = np.dot(w, activation) + b
67             zs.append(z)
68             activation = sigmoid(z)
69             activations.append(activation)
70         return activation, activations, zs
71
72     def backprop_bce(self, image, label):
73         """Return a tuple '(nabla_b, nabla_w)' representing the
74         gradient for the cost function C_x. 'nabla_b' and
75         'nabla_w' are layer-by-layer lists of numpy arrays, similar
76         to 'self.biases' and 'self.weights'."""
77         nabla_b = [np.zeros(b.shape) for b in self.biases]
78         nabla_w = [np.zeros(w.shape) for w in self.weights]
79
80         # feedforward and back error calculation depending on type of image
81         activation, activations, zs = self.forwardprop(image)
82         delta = BCE_derivative(activations[-1], label) * sigmoid_prime(zs[-1])
83
84         # backward pass
85         nabla_b[-1] = delta
86         nabla_w[-1] = np.dot(delta, activations[-2].reshape(1, activations[-2].
87             shape[0]))
88
89         for l in range(2, self.num_layers):
90             z = zs[-l]
91             delta = np.dot(self.weights[-l + 1].transpose(), delta) * sigmoid_prime
92             (z)

```

```

91         nabla_b[-1] = delta
92         nabla_w[-1] = np.dot(delta.reshape(delta.shape[0], 1), activations[-1 -
1].reshape(1, activations[-1 - 1].shape[0]))
93     return nabla_b, nabla_w, activations[-1]
94
95 def backprop_minimax(self, real_image, fake_image, is_real):
96     """Return a tuple '(nabla_b, nabla_w)' representing the
97     gradient for the cost function C_x. 'nabla_b' and
98     'nabla_w' are layer-by-layer lists of numpy arrays, similar
99     to 'self.biases' and 'self.weights'."""
100     nabla_b = [np.zeros(b.shape) for b in self.biases]
101     nabla_w = [np.zeros(w.shape) for w in self.weights]
102
103     # feedforward and back error calculation depending on type of image
104     activation_real, activations_real, zs_real = self.forwardprop(real_image)
105     activation_fake, activations_fake, zs_fake = self.forwardprop(fake_image)
106
107     if is_real:
108         delta = minimax_derivative_real(activations_real[-1]) * sigmoid_prime(
zs_real[-1])
109         activations, zs = activations_real, zs_real
110     else:
111         delta = minimax_derivative_fake(activations_fake[-1]) * sigmoid_prime(
zs_fake[-1])
112         activations, zs = activations_fake, zs_fake
113
114     # backward pass
115     nabla_b[-1] = delta
116     nabla_w[-1] = np.dot(delta, activations[-2].reshape(1, activations[-2].
shape[0]))
117
118     for l in range(2, self.num_layers):
119         z = zs[-1]
120         delta = np.dot(self.weights[-l + 1].transpose(), delta) * sigmoid_prime
(z)
121
122         nabla_b[-1] = delta
123         nabla_w[-1] = np.dot(delta.reshape(delta.shape[0], 1),
activations[-1 - 1].reshape(1, activations[-1 - 1].shape[0]))
124     return nabla_b, nabla_w, activations[-1]
125
126 def train_mini_batch(self, mini_batch, learning_rate):
127     global label_real, label_fake
128     nabla_b = [np.zeros(b.shape) for b in self.biases]
129     nabla_w = [np.zeros(w.shape) for w in self.weights]
130
131     if self.type_loss == "binary cross entropy":
132         for real_image, fake_image in mini_batch:

```

```

133         delta_nabla_b, delta_nabla_w, label_real = self.backprop_bce(
134             real_image, np.array([1.]))
135         nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
136         nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
137
138         delta_nabla_b, delta_nabla_w, label_fake = self.backprop_bce(
139             fake_image, np.array([0.]))
140         nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
141         nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
142
143     elif self.type_loss == "minimax":
144         for real_image, fake_image in mini_batch:
145             delta_nabla_b, delta_nabla_w, label_real = self.backprop_minimax(
146                 real_image, fake_image, True)
147             nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
148             nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
149
150             delta_nabla_b, delta_nabla_w, label_fake = self.backprop_minimax(
151                 real_image, fake_image, False)
152             nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
153             nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
154
155         else:
156             raise Exception("type of loss function not valid")
157
158     # gradient descent
159     # nabla_w and nabla_b are multiplied by the learning rate
160     # and taken the mean of (dividing by the mini batch size)
161     self.weights = [w - (learning_rate / len(mini_batch)) * nw
162                     for w, nw in zip(self.weights, nabla_w)]
163     self.biases = [b - (learning_rate / len(mini_batch)) * nb
164                    for b, nb in zip(self.biases, nabla_b)]

```

LISTING E.3: Codi final pel discriminador

**E.1.0.3 Codi per el generador** Aquest codi es pot trobar al [repositori del treball](#) en l'arxiu `generador.py`.

```

1  """QUANTUM GENERATOR"""
2
3  from typing import Any, Dict, Optional, cast
4
5  import numpy as np
6  import qiskit
7  from qiskit import ClassicalRegister, QuantumRegister
8  from qiskit.circuit import QuantumCircuit
9  from qiskit.circuit.library import TwoLocal

```

```

10 from qiskit.providers.aer import AerSimulator
11
12 from quantumGAN.functions import create_entangler_map, create_real_keys,
    minimax_generator
13
14
15 class QuantumGenerator:
16
17     def __init__(
18         self,
19         shots: int,
20         num_qubits: int,
21         num_qubits_ancilla: int,
22         generator_circuit: Optional[QuantumCircuit] = None,
23         snapshot_dir: Optional[str] = None
24     ) -> None:
25
26         super().__init__()
27         # passar els arguments de la classe a metodes en de classes
28         # d'aquesta manera son accessibles per qualsevol funcio dintre de la classe
29         self.num_qubits_total = num_qubits
30         self.num_qubits_ancilla = num_qubits_ancilla
31         self.generator_circuit = generator_circuit
32         self.snapshot_dir = snapshot_dir
33         self.shots = shots
34         self.discriminator = None
35         self.ret: Dict[str, Any] = {"loss": []}
36         self.simulator = AerSimulator()
37
38     def init_parameters(self):
39         """Inicia els parametres inicial i crea el circuit al qual se li posen els
    parametres"""
40         # iniciacia dels parametres inicials i dels circuits al qual posar aquests
    parametres
41         self.generator_circuit = self.construct_circuit(latent_space_noise=None,
    to_measure=False)
42         # s'ha de crear primer el circuit porque d'aquesta manera es pot saber el
    nombre de parametres que es necessiten
43         self.parameter_values = np.random.normal(np.pi / 2, .1, self.
    generator_circuit.num_parameters)
44
45     def construct_circuit(self,
46                           latent_space_noise,
47                           to_measure: bool):
48         """Crea el circuit quantic des de zero a partir de diversos registres de
    qubits"""
49         if self.num_qubits_ancilla is 0:

```

```

50         qr = QuantumRegister(self.num_qubits_total, 'q')
51         cr = ClassicalRegister(self.num_qubits_total, 'c')
52         qc = QuantumCircuit(qr, cr)
53     else:
54         qr = QuantumRegister(self.num_qubits_total - self.num_qubits_ancilla, '
q')
55         anc = QuantumRegister(self.num_qubits_ancilla, 'ancilla')
56         cr = ClassicalRegister(self.num_qubits_total - self.num_qubits_ancilla,
'c')
57         qc = QuantumCircuit(anc, qr, cr)
58
59         # creacio de la part del circuit que conte la implantacio dels parametres d
'input. En cas que no es donin aquests parametres es creen automaticament
60         if latent_space_noise is None:
61             randoms = np.random.normal(-np.pi * .01, np.pi * .01, self.
num_qubits_total)
62             init_dist = qiskit.QuantumCircuit(self.num_qubits_total)
63
64             # es col·loca una porta RY en cada qubits i amb un parametre diferent
cadascuna
65             for index in range(self.num_qubits_total):
66                 init_dist.ry(randoms[index], index)
67         else:
68             init_dist = qiskit.QuantumCircuit(self.num_qubits_total)
69
70             for index in range(self.num_qubits_total):
71                 init_dist.ry(latent_space_noise[index], index)
72
73         # la funcio create_entangler_map crea les parelles de qubits a les qual col·
locar les portes CZ
74         # en funcio del nombre de qubits
75         if self.num_qubits_ancilla is 0:
76             entangler_map = create_entangler_map(self.num_qubits_total)
77         else:
78             entangler_map = create_entangler_map(self.num_qubits_total - self.
num_qubits_ancilla)
79
80         # creacio final dels circuits a partir una funcio integrada a Qiskit que va
repetint les operacions
81         # que se li especifiquen
82         ansatz = TwoLocal(int(self.num_qubits_total), 'ry', 'cz', entanglement=
entangler_map, reps=1, insert_barriers=True)
83
84         # aqui s'ajunten el circuit que funciona com a input amb el circuit que
consisteix en la repeticio
85         # de les portes RY i CZ
86         qc = qc.compose(init_dist, front=True)

```

```

87         qc = qc.compose(ansatz, front=False)
88
89         if to_measure:
90             qc.measure(qr, cr)
91
92         return qc
93
94     def set_discriminator(self, discriminator) -> None:
95         self.discriminator = discriminator
96
97     def get_output(
98         self,
99         latent_space_noise,
100         parameters: Optional[np.ndarray] = None
101     ):
102         """Retorna un output del generador quan se li dona un estat d'input i
103         opcionalment uns parametres en especific. Els pixels estan compostos per la
104         probabilitat que un qubit resulti en ket_0 en cada base. Per tant, els pixels
105         de l'imatge estan normalitzats amb la norma l-1."""
106
107         real_keys_set, real_keys_list = create_real_keys(self.num_qubits_total -
108         self.num_qubits_ancilla)
109
110         # en cas de que no es donin parametres com a input, es treuen els
111         parametres de la variable
112
113         # self.parameter_values. Es a dir els parametres que es creen
114         automaticament al principi i que es van
115
116         # actualitzant al mateix temps que el model s'optimitza
117         if parameters is None:
118             parameters = cast(np.ndarray, self.parameter_values)
119
120         qc = self.construct_circuit(latent_space_noise, True)
121
122         parameter_binds = {parameter_id: parameter_value for parameter_id,
123         parameter_value in zip(qc.parameters, parameters)}
124
125         # el metode bind_parametres del circuit quantic
126         qc = qc.bind_parameters(parameter_binds)
127
128         # Simulacio dels circuits mitjancant el simulador Aer de Qiskit. El nivell
129         d'optimitzacio es zero, porque al ser circuits petits i que simulen una vegada,
130         no es necessari. Al optimitzar el proces acaba sent mes lent.
131
132         result_ideal = qiskit.execute(experiments=qc,
133                                     backend=self.simulator,
134                                     shots=self.shots,
135                                     optimization_level=0).result()
136
137         counts = result_ideal.get_counts()
138

```

```

125     try:
126         # creacio de l'imatge resultant
127         pixels = np.array([counts[index] for index in list(real_keys_list)])
128
129     except KeyError:
130         # aquesta excepcio sorgeix quan en el diccionari dels resultats no
131         # estan totes les keys pel fet que qiskit, en cas de que no hi hagi un mesurament
132         # en una base, no inclou aquesta base en el diccionari
133         keys = counts.keys()
134         missing_keys = real_keys_set.difference(keys)
135         # s'utilitza un la resta entre dos sets per poder veure quina es la key
136         # que falta en el diccionari
137         for key_missing in missing_keys:
138             counts[key_missing] = 0
139
140         # una vegada es troba les keys que faltaven es crea l'imatge resultant
141         pixels = np.array([counts[index] for index in list(real_keys_list)])
142
143     pixels = pixels / self.shots
144     return pixels
145
146 def get_output_pixels(
147     self,
148     latent_space_noise,
149     params: Optional[np.ndarray] = None
150 ):
151     """Retorna un output del generador quan se li dona un estat d'input i
152     opcionalment uns parametres en especific. Cada pixel es la probabilitat de que
153     un qubits resulti en l'estat ket_0, per tant, els valors cada pixel (que son
154     independents entre si) es troba en l'interval (0, 1) """
155     qc = QuantumCircuit(self.num_qubits_total)
156
157     init_dist = qiskit.QuantumCircuit(self.num_qubits_total)
158     assert latent_space_noise.shape[0] == self.num_qubits_total
159
160     for num_qubit in range(self.num_qubits_total):
161         init_dist.ry(latent_space_noise[num_qubit], num_qubit)
162
163     if params is None:
164         params = cast(np.ndarray, self.parameter_values)
165
166     qc.assign_parameters(params)
167
168     # comptes de simular els valors que donara cada qubits, es simula l'estat
169     # final del circuit i d'aquest s'extreuen els valors que es mesuraran per a cada
170     # qubit
171     state_vector = qiskit.quantum_info.Statevector.from_instruction(qc)

```

```

164     pixels = []
165     for qubit in range(self.num_qubits_total):
166         # per treure la probabilitat s'utilitza una funcio implementada en
167         Qiskit
168         pixels.append(state_vector.proBABILITIES([qubit])[0])
169
170         # creacio de l'imatge resultada a partir de la llista que conte el valor
171         per a cada pixel
172         generated_samples = np.array(pixels)
173         generated_samples.flatten()
174
175     return generated_samples
176
177 def train_mini_batch(self, mini_batch, learning_rate):
178     """Optimitzacio del generador per una batch d'imatges. Retorna una batch de
179     les imatges generades amb unes imatges reals que poder donar com a input al
180     generador. """
181     nabla_theta = np.zeros_like(self.parameter_values.shape)
182     new_images = []
183
184     for _, noise in mini_batch:
185         for index, _ in enumerate(self.parameter_values):
186             perturbation_vector = np.zeros_like(self.parameter_values)
187             perturbation_vector[index] = 1
188
189             # Creacio dels parametres per generar les dues imatges
190             pos_params = self.parameter_values + (np.pi / 4) *
191             perturbation_vector
192             neg_params = self.parameter_values - (np.pi / 4) *
193             perturbation_vector
194
195             pos_result = self.get_output(noise, parameters=pos_params) #
196             Generacio imatges
197             neg_result = self.get_output(noise, parameters=neg_params)
198
199             pos_result = self.discriminator.predict(pos_result) # Assignacio
200             de les etiquetes
201             neg_result = self.discriminator.predict(neg_result)
202
203             # Diferencia entre les avaluacions de la funcio de perdua entre les
204             dues etiquetes
205             gradient = minimax_generator(pos_result) - minimax_generator(
206             neg_result)
207             nabla_theta[index] += gradient # Afegir derivada d'un parametre al
208             gradient
209             new_images.append(self.get_output(noise))

```



```

200         for index, _ in enumerate(self.parameter_values):
201             # Actualitzacio dels parametres a traves del gradient
202             self.parameter_values[index] += (learning_rate / len(mini_batch)) *
nabla_theta[index]
203
204         # Creacio de la batch d'imatges a retornar
205         mini_batch = [(datapoint[0], fake_image) for datapoint, fake_image in zip(
mini_batch, new_images)]
206         return mini_batch

```

LISTING E.4: Codi final pel generador

**E.1.0.4 Definició del model** Aquest codi es pot trobar al [repositori del treball](#) en l'arxiu `qgan.py`.

```

1  import glob
2  import os
3  import json
4  import random
5  from datetime import datetime
6  from typing import List
7
8  import imageio
9  import matplotlib.pyplot as plt
10 import numpy as np
11
12 from quantumGAN.discriminator import ClassicalDiscriminator
13 from quantumGAN.functions import fechet_distance, minimax, images_to_distribution,
    images_to_scatter
14 from quantumGAN.quantum_generator import QuantumGenerator
15
16
17 class Quantum_GAN:
18
19     def __init__(self,
20                 generator: QuantumGenerator,
21                 discriminator: ClassicalDiscriminator
22                 ):
23
24         self.last_batch = None
25         now = datetime.now()
26         init_time = now.strftime("%d_%m_%Y_%H_%M_%S")
27         self.path = "data/run{}".format(init_time)
28         self.path_images = self.path + "/images"
29         self.filename = "run.txt"
30

```

```

31     if not os.path.exists(self.path):
32         os.makedirs(self.path)
33
34     if not os.path.exists(self.path_images):
35         os.makedirs(self.path_images)
36
37     with open(os.path.join(self.path, self.filename), "w") as file:
38         file.write("RUN {} \n".format(init_time))
39         file.close()
40
41     self.generator = generator
42     self.discriminator = discriminator
43     self.loss_series, self.label_real_series, self.label_fake_series, self.
44     FD_score = [], [], [], []
45
46     self.generator.init_parameters()
47     self.example_g_circuit = self.generator.construct_circuit(
48         latent_space_noise=None,
49         to_measure=False)
50     self.generator.set_discriminator(self.discriminator)
51
52     def __repr__(self):
53         return "Discriminator Architecture: {} \n Generator Example Circuit: \n{}"
54         \
55         .format(self.discriminator.sizes, self.example_g_circuit)
56
57     def store_info(self, epoch, loss, real_label, fake_label):
58         file = open(os.path.join(self.path, self.filename), "a")
59         file.write("{} epoch LOSS {} Parameters {} REAL {} FAKE {} \n"
60             .format(epoch,
61                 loss,
62                 self.generator.parameter_values,
63                 real_label,
64                 fake_label))
65         file.close()
66
67     def plot(self):
68         # save data for plotting
69         fake_images, real_images = [], []
70         for image_batch in self.last_batch:
71             fake_images.append(image_batch[1])
72             real_images.append(image_batch[0])
73
74         keys, average_result = images_to_distribution(fake_images)
75         print(average_result)
76         if self.generator.num_qubits_ancilla != 0:

```

```

74         plt.title(f"Distribucio d'una imatge generada \n(ancilla, epoch={self.
num_epochs})")
75     else:
76         plt.title(f"Distribucio d'una imatge generada \n(epoch={self.num_epochs
})")
77     plt.ylim(0., 1.)
78     plt.bar(keys, average_result)
79     plt.savefig(self.path + "/fake_distribution.png")
80     plt.clf()
81
82     keys_real, average_result_real = images_to_distribution(real_images)
83     print(average_result_real)
84     if self.generator.num_qubits_ancilla != 0:
85         plt.title(f"Distribucio d'una imatge real \n(ancilla, epoch={self.
num_epochs})")
86     else:
87         plt.title(f"Distribucio d'una imatge real \n(epoch={self.num_epochs})")
88     plt.ylim(0., 1.)
89     plt.bar(keys_real, average_result_real)
90     plt.savefig(self.path + "/real_distribution.png")
91     plt.clf()
92
93     y_axis, x_axis = images_to_scatter(fake_images)
94     y_axis_real, x_axis_real = images_to_scatter(real_images)
95     if self.generator.num_qubits_ancilla != 0:
96         plt.title(f"Comparacio entre les imatges reals con les imatges falses \
n(ancilla, epoch={self.num_epochs})")
97     else:
98         plt.title(f"Comparacio entre les imatges reals con les imatges falses \
n(epoch={self.num_epochs})")
99     plt.ylim(0. - .1, 1. + .1)
100    plt.scatter(y_axis, x_axis, label='Valors per les imatges falses',
101                color='indigo',
102                linewidth=.1,
103                alpha=.2)
104    plt.scatter(y_axis_real, x_axis_real, label='Valors per les imatges reals',
105                color='limegreen',
106                linewidth=.1,
107                alpha=.2)
108    plt.savefig(self.path + "/scatter_plot.png")
109    plt.clf()
110
111    t_steps = np.arange(self.num_epochs)
112    plt.figure(figsize=(6, 5))
113    if self.generator.num_qubits_ancilla != 0:
114        plt.title(f"Progres de la funcio de perdua \n(ancilla, epoch={self.
num_epochs})")

```

```

115         else:
116             plt.title(f"Progres de la funcio de perdua \n(epoch={self.num_epochs})"
117         )
118             plt.plot(t_steps, self.loss_series, label='Funcio de perdua del
119         discriminador', color='rebeccapurple', linewidth=2)
120             plt.grid()
121             plt.legend(loc='best')
122             plt.xlabel('Iteracions')
123             plt.ylabel('Funcio de perdua')
124             plt.savefig(self.path + "/loss_plot.png")
125             plt.clf()
126
127             t_steps = np.arange(self.num_epochs)
128             plt.figure(figsize=(6, 5))
129             if self.generator.num_qubits_ancilla != 0:
130                 plt.title(f"Progres de les etiquetes \n(ancilla, epoch={self.num_epochs
131             })")
132             else:
133                 plt.title(f"Progres de les etiquetes \n(epoch={self.num_epochs})")
134             plt.scatter(t_steps, self.label_real_series, label='Etiquetes per les
135         imatges reals',
136             color='mediumvioletred',
137             linewidth=.1)
138             plt.scatter(t_steps, self.label_fake_series, label='Etiquetes per les
139         imatges generades',
140             color='rebeccapurple',
141             linewidth=.1)
142             plt.grid()
143             plt.ylim(0. - .1, 1. + .1)
144             plt.legend(loc='best')
145             plt.xlabel('Iteracions')
146             plt.ylabel('Valor de les etiquetes')
147             plt.savefig(self.path + "/labels_plot.png")
148             plt.clf()
149
150             plt.figure(figsize=(6, 5))
151             if self.generator.num_qubits_ancilla != 0:
152                 plt.title(f"Puntuacio de a partir de la distancia de Frechet \n(ancilla
153             , epoch={self.num_epochs})")
154             else:
155                 plt.title(f"Puntuacio de a partir de la distancia de Frechet \n(epoch={
156             self.num_epochs})")
157             plt.plot(t_steps, self.FD_score, label='Valor de la puntuacio de Frechet',
158             color='rebeccapurple', linewidth=2)
159             plt.grid()
160             plt.legend(loc='best')
161             plt.ylim(0., .28)

```

```

154     plt.xlabel('Iteracions')
155     plt.ylabel('Valor de la puntuacio')
156     plt.savefig(self.path + "/FD_score.png")
157     plt.clf()
158
159     def train(self,
160               num_epochs: int,
161               training_data: List,
162               batch_size: int,
163               generator_learning_rate: float,
164               discriminator_learning_rate: float,
165               is_save_images: bool):
166
167         self.num_epochs = num_epochs
168         self.training_data = training_data
169         self.batch_size = batch_size
170         self.batch_size = batch_size
171         self.generator_lr = generator_learning_rate
172         self.discriminator_lr = discriminator_learning_rate
173         self.is_save_images = is_save_images
174
175         noise = self.training_data[0][1]
176         time_init = datetime.now()
177         for o in range(self.num_epochs):
178             mini_batches = create_mini_batches(self.training_data, self.batch_size)
179             output_fake = self.generator.get_output(latent_space_noise=mini_batches
180 [0][0][1], parameters=None)
181
182             for mini_batch in mini_batches:
183                 self.last_batch = self.generator.train_mini_batch(mini_batch, self.
184 generator_lr)
185                 self.discriminator.train_mini_batch(self.last_batch, self.
186 discriminator_lr)
187
188             output_real = mini_batches[0][0][0]
189             if is_save_images:
190                 self.save_images(self.generator.get_output(latent_space_noise=noise
191 , parameters=None), o)
192
193             self.FD_score.append(fechet_distance(output_real, output_fake))
194             label_real, label_fake = self.discriminator.predict(output_real), self.
195 discriminator.predict(output_fake)
196             loss_final = 1 / 2 * (minimax(label_real, label_fake) + minimax(
197 label_real, label_fake))
198
199             self.loss_series.append(loss_final)
200             self.label_real_series.append(label_real)

```

```

195         self.label_fake_series.append(label_fake)
196
197         print("Epoch {}: Loss: {}".format(o, loss_final), output_real,
output_fake)
198         print(label_real[-1], label_fake[-1])
199         self.store_info(o, loss_final, label_real, label_fake)
200
201         time_now = datetime.now()
202         print((time_now - time_init).total_seconds(), "seconds")
203
204     def save_images(self, image, epoch):
205         image_shape = int(image.shape[0] / 2)
206         image = image.reshape(image_shape, image_shape)
207
208         plt.imshow(image, cmap='gray', vmax=1., vmin=0.)
209         plt.axis('off')
210         plt.savefig(self.path_images + '/image_at_epoch_{:04d}.png'.format(epoch))
211         plt.clf()
212
213     def create_gif(self):
214         anim_file = self.path + '/dcgan.gif'
215
216         with imageio.get_writer(anim_file, mode='I') as writer:
217             filenames = glob.glob(self.path_images + '/image*.png')
218             filenames = sorted(filenames)
219
220             for filename in filenames:
221                 image = imageio.imread(filename)
222                 writer.append_data(image)
223
224             image = imageio.imread(filename)
225             writer.append_data(image)
226
227     def save(self):
228         """Save the neural network to the file 'filename'."""
229         data = {"batch_size": self.batch_size,
230               "D_sizes": self.discriminator.sizes,
231               "D_weights": [w.tolist() for w in self.discriminator.weights],
232               "D_biases": [b.tolist() for b in self.discriminator.biases],
233               "D_loss": self.discriminator.type_loss,
234               "Q_parameters": [theta for theta in self.generator.parameter_values
],
235               "Q_shots": self.generator.shots,
236               "Q_num_qubits": self.generator.num_qubits_total,
237               "Q_num_qubits_ancilla": self.generator.num_qubits_ancilla,
238               "real_labels": self.label_real_series,
239               "fake_labels": self.label_fake_series,

```

```

240         "loss_series": self.loss_series
241     }
242     f = open(os.path.join(self.path, "data.txt"), "a")
243     json.dump(data, f)
244     f.close()
245
246
247     def create_mini_batches(training_data, mini_batch_size):
248         n = len(training_data)
249         random.shuffle(training_data)
250         mini_batches = [
251             training_data[k:k + mini_batch_size]
252             for k in range(0, n, mini_batch_size)]
253         return [mini_batches[0]]
254
255
256     def load_gan(filename):
257         f = open(filename, "r")
258         data = json.load(f)
259         f.close()
260         discriminator = ClassicalDiscriminator(data["D_sizes"], data["D_loss"])
261
262         generator = QuantumGenerator(num_qubits=data["Q_num_qubits"],
263                                     generator_circuit=None,
264                                     num_qubits_ancilla=data["Q_num_qubits_ancilla"]
265                                     ],
266                                     shots=data["Q_shots"])
267
268         quantum_gan = Quantum_GAN(generator, discriminator)
269
270         quantum_gan.discriminator.weights = [np.array(w) for w in data["D_weights"]
271         ]
272         quantum_gan.discriminator.biases = [np.array(b) for b in data["D_biases"]]
273         quantum_gan.generator.parameter_values = np.array(data["Q_parameters"])
274
275         return quantum_gan, data["batch_size"]

```

LISTING E.5: Codi per la definició del model

**E.1.0.5 Execució del model** Aquest codi es pot trobar al [repositori del treball](#) en l'arxiu `main.py`.

```

1 import numpy as np
2

```

```

3 from quantumGAN.discriminator import ClassicalDiscriminator
4 from quantumGAN.qgan import Quantum_GAN
5 from quantumGAN.quantum_generator import QuantumGenerator
6
7 num_qubits: int = 3
8
9 # Set number of training epochs
10 num_epochs = 150
11 # Batch size
12 batch_size = 10
13
14 train_data = []
15 for _ in range(800):
16     x2 = np.random.uniform(.55, .46, (2,))
17     fake_datapoint = np.random.uniform(-np.pi * .01, np.pi * .01, (num_qubits,))
18     real_datapoint = np.array([x2[0], 0, x2[0], 0])
19     train_data.append((real_datapoint, fake_datapoint))
20
21 discriminator = ClassicalDiscriminator(sizes=[4, 16, 8, 1],
22                                       type_loss="minimax")
23 generator = QuantumGenerator(num_qubits=num_qubits,
24                              generator_circuit=None,
25                              num_qubits_ancilla=1,
26                              shots=4096)
27
28 quantum_gan = Quantum_GAN(generator, discriminator)
29 print(quantum_gan)
30 print(num_epochs)
31 quantum_gan.generator.get_output(fake_datapoint[0], None)
32 quantum_gan.train(num_epochs, train_data, batch_size, .1, .1, True)
33
34 quantum_gan.plot()
35 quantum_gan.create_gif()
36 quantum_gan.save()

```

LISTING E.6: Codi final pel generador

**E.1.0.6 Multiprocessament** Per poder ser més eficient alhora de executar els models, vaig decidir utilitzar una característica molt útil de Python, el *multiprocessing*. Usualment una instància de Python, es a dir; un arxiu executant-se, només utilitza un nucli del processador a la vegada, però amb *multiprocessing* pots utilitzar múltiples nuclis amb un mateix arxiu, d'aquesta manera executant diverses línies de codi al mateix temps. Per poder executar diversos models a la mateixa



vegada i d'una forma simple, vaig utilitzar aquest modul de Python per executar els models dels quals he agafat les dades que he presentat en aquest treball.

Aquest codi es pot trobar al [repositori del treball](#) en l'arxiu `test_multi.py`.

```

1 import multiprocessing
2 import time
3 import numpy as np
4
5 from quantumGAN.discriminator import ClassicalDiscriminator
6 from quantumGAN.qgan import Quantum_GAN
7 from quantumGAN.quantum_generator import QuantumGenerator
8
9 batch_size = 10
10
11 train_data = []
12 # generacio de les dades
13 for _ in range(800):
14     x2 = np.random.uniform(.55, .46, (2,))
15     fake_datapoint = np.random.uniform(-np.pi * .01, np.pi * .01, (4,))
16     real_datapoint = np.array([x2[0], 0, x2[0], 0])
17     train_data.append((real_datapoint, fake_datapoint))
18 list_epochs = [550, 550, 550]
19
20 example_generator_ancilla = QuantumGenerator(num_qubits=4,
21                                             generator_circuit=None,
22                                             num_qubits_ancilla=2,
23                                             shots=4096)
24 example_generator = QuantumGenerator(num_qubits=2,
25                                     generator_circuit=None,
26                                     num_qubits_ancilla=0,
27                                     shots=4096)
28
29 # generem els parametres inicials d'acord amb el circuit amb un mayor nombre de
    parametres
30 circuit_ancilla = example_generator_ancilla.construct_circuit(None, False)
31 circuit_not_ancilla = example_generator_ancilla.construct_circuit(None, False)
32
33 init_parameters_ancilla = np.random.normal(np.pi / 2, .1, circuit_ancilla.
    num_parameters)
34
35 # canviem les dimensions dels parametres perquè coincideixin amb les dimensions
    necessaries pel circuit
36 # sense qubits ancilla
37 not_ancilla_list = init_parameters_ancilla.tolist()[
38 circuit_ancilla.num_parameters - circuit_not_ancilla.num_parameters:]
39 init_parameters_not_ancilla = np.array(not_ancilla_list)
40

```

```

41 example_discriminator = discriminator_ancilla = \
42 ClassicalDiscriminator(sizes=[4, 16, 8, 1], type_loss="minimax")
43
44 init_biases_discriminator, init_weights_discriminator = example_discriminator.
    init_parameters()
45
46
47 def to_train(num_epochs):
48     # definicio d'un discriminador pel model amb la funcio no-lineal i una altre
    pel model que no la te
49     discriminator_ancilla = \
50         ClassicalDiscriminator(sizes=[4, 16, 8, 1], type_loss="minimax")
51     discriminator_not_ancilla = \
52         ClassicalDiscriminator(sizes=[4, 16, 8, 1], type_loss="minimax")
53
54     # es defineix un generador que te la funcio no-lienal integrada en el circuit
55     generator_ancilla = \
56         QuantumGenerator(num_qubits=4,
57                           generator_circuit=None,
58                           num_qubits_ancilla=2, # IMPORTANT
59                           shots=4096)
60
61     # i un altre generador que no te la funcio no-lineal
62     generator_not_ancilla = \
63         QuantumGenerator(num_qubits=2,
64                           generator_circuit=None,
65                           num_qubits_ancilla=0, # IMPORTANT
66                           shots=4096)
67
68     quantum_gan_ancilla = \
69         Quantum_GAN(generator_ancilla, discriminator_ancilla)
70     time.sleep(1)
71     quantum_gan_not_ancilla = \
72         Quantum_GAN(generator_not_ancilla, discriminator_not_ancilla)
73     print(quantum_gan_not_ancilla.path)
74     print(quantum_gan_ancilla.path)
75
76     # abans de començar a entrenar el model s'ha de canviar els parametres als
    definits anteriorment
77     quantum_gan_ancilla.generator.parameter_values = init_parameters_ancilla
78     quantum_gan_ancilla.discriminator.weights = init_weights_discriminator
79     quantum_gan_ancilla.discriminator.biases = init_biases_discriminator
80     quantum_gan_ancilla.train(num_epochs, train_data, batch_size, .1, .1, False)
81
82     quantum_gan_not_ancilla.generator.parameter_values =
83     init_parameters_not_ancilla
84     quantum_gan_not_ancilla.discriminator.weights = init_weights_discriminator
85     quantum_gan_not_ancilla.discriminator.biases = init_biases_discriminator

```

```

84     quantum_gan_not_ancilla.train(num_epochs, train_data, batch_size, .1, .1, False
85     )
86
87     # una vegada ha acabat l'optimitzacio es creen els grafics per compara l'
88     eficiencia del models
89     quantum_gan_not_ancilla.plot()
90     quantum_gan_ancilla.plot()
91
92 def main():
93     # Crear una queue para compartir les dades entre els processos
94     # Crear i iniciar el proces de simulacio
95     jobs = []
96     for epoch in list_epochs:
97         simulate = multiprocessing.Process(None, to_train, args=(epoch,))
98         jobs.append(simulate)
99         time.sleep(2)
100         simulate.start()
101
102 if __name__ == '__main__':
103     main()

```

LISTING E.7: Executar els models amb multiprocessing

**E.1.0.7 Funcions extres** Hi han funcions en el codi que utilitzo en diversos arxius, aquestes les defineixo en l'arxiu `functions.py`, que es pot trobar al [repositori del treball](#). En aquest arxiu també hi ha una funció que dona a terme una traça parcial a una matriu donada, malgrat que no la he arribat a utilitzar.

```

1 import itertools
2 import math
3
4 import numpy as np
5 from matplotlib import pyplot as plt
6
7
8 # DATA PROCESSING
9 from scipy import linalg
10
11
12 def save_images_color(image, epoch):
13     plt.imshow(image.reshape(int(image.shape[0] / 3), 1, 3))
14     plt.axis('off')
15     plt.savefig('images/image_at_epoch_{:04d}.png'.format(epoch))

```

```

16
17
18 def create_real_keys(num_qubits):
19     lst = [[str(a) for a in i] for i in itertools.product([0, 1], repeat=num_qubits
20     )]
21     new_lst = []
22     for element in lst:
23         word = str()
24         for number in element:
25             word = word + number
26             new_lst.append(word)
27     return set(new_lst), new_lst
28
29 def create_entangler_map(num_qubits: int):
30     lst = [list(i) for i in itertools.combinations(range(num_qubits), 2)]
31     index = 0
32     entangler_map = []
33     for i in reversed(range(num_qubits)):
34         try:
35             entangler_map.append(lst[index])
36             index += i
37         except IndexError:
38             return entangler_map
39
40
41 def images_to_distribution(batch_image):
42     num_images = len(batch_image)
43     sum_result = 0
44     for image in batch_image:
45         sum_result += image
46     average_result = sum_result / num_images
47     keys = create_real_keys(int(math.sqrt(batch_image[0].shape[0]))) [1]
48     return keys, average_result
49
50
51 def images_to_scatter(batch_image):
52     keys = create_real_keys(int(math.sqrt(batch_image[0].shape[0]))) [1]
53     x_axis = []
54     y_axis = []
55
56     for image in batch_image:
57         pixel_count = 0
58         for pixel in image:
59             x_axis.append(pixel)
60             y_axis.append(keys[pixel_count])
61             pixel_count += 1

```

```

62
63     return y_axis, x_axis
64
65 def fechet_distance(image1: np.array,
66                     image2: np.array):
67     assert image1.shape == image2.shape
68     y = np.arange(0, image1.flatten().shape[0])
69
70     matrix_a_cov = np.cov(np.stack((y, image1.flatten())), axis=0)
71     matrix_b_cov = np.cov(np.stack((y, image2.flatten())), axis=0)
72
73     to_trace = matrix_a_cov + matrix_b_cov - 2*(linalg.fractional_matrix_power(np.
74     dot(matrix_a_cov, matrix_b_cov), .5))
75     return np.abs(image1.mean() - image2.mean())**2 + np.trace(to_trace)
76
77 # ACTIVATION FUNCTIONS
78
79 def sigmoid(z):
80     """The sigmoid function."""
81     return 1.0 / (1.0 + np.exp(-z))
82
83
84 def sigmoid_prime(z):
85     """Derivative of the sigmoid function."""
86     return sigmoid(z) * (1 - sigmoid(z))
87
88
89 def relu(z):
90     return np.maximum(0, z)
91
92
93 def relu_prime(z):
94     z[z <= 0] = 0
95     z[z > 0] = 1
96     return z
97
98
99 # LOSSES
100 def MSE_derivative(prediction, y):
101     return 2 * (y - prediction)
102
103
104 def MSE(prediction, y):
105     return (y - prediction) ** 2
106
107

```

```

108 def BCE_derivative(prediction, target):
109     # return prediction - target
110     return -target / prediction + (1 - target) / (1 - prediction)
111
112
113 def BCE(predictions: np.ndarray, targets: np.ndarray) -> np.ndarray:
114     return targets * np.log(predictions) + (1 - targets) * np.log(1 - predictions).
115     mean()
116
117 def minimax_derivative_real(real_prediction):
118     real_prediction = np.array(real_prediction)
119     return np.nan_to_num((-1) * (1 / real_prediction))
120
121
122 def minimax_derivative_fake(fake_prediction):
123     fake_prediction = np.array(fake_prediction)
124     return np.nan_to_num(1 / (1 - fake_prediction))
125
126
127 def minimax(real_prediction, fake_prediction):
128     return np.nan_to_num(np.log(real_prediction) + np.log(1 - fake_prediction))
129
130
131 def minimax_generator(prediction_fake):
132     return (-1) * np.log(1 - prediction_fake)
133
134
135 # QUANTUM FUNCTIONS
136
137 class Partial_Trace:
138     def __init__(self, state: np.array, qubits_out: int, side: str):
139
140         self.state = state
141         self.qubits_out = qubits_out
142         self.side = side
143
144         if self.state.ndim == 1:
145             self.state = np.outer(self.state, self.state)
146
147         self.total_dim = self.state.shape[0]
148
149         self.num_qubits = int(np.log2(self.total_dim))
150         self.a_dim = 2 ** (self.num_qubits - self.qubits_out)
151         self.b_dim = 2 ** self.qubits_out
152
153         # if self.side == "bot":

```

```

154     self.basis_b = [_ for _ in np.identity(int(self.b_dim))]
155     self.basis_a = [_ for _ in np.identity(int(self.a_dim))]
156
157     # elif self.side == "top":
158     #     self.basis_a = [_ for _ in np.identity(int(self.b_dim))]
159     #     self.basis_b = [_ for _ in np.identity(int(self.a_dim))]
160     #
161     # else:
162     #     raise NameError("invalid side argument, enter \"bot\" or \"top\"")
163     print(self.basis_a, self.basis_b)
164
165     def get_entry(self, index_i, index_j):
166         sigma = 0
167
168         if self.side == "bot":
169             for k in range(self.qubits_out + 1):
170                 ab_l = np.kron(self.basis_a[index_i],
171                               self.basis_b[k])
172                 ab_r = np.kron(self.basis_a[index_j],
173                               self.basis_b[k])
174
175                 print(ab_r, ab_l)
176
177                 right_side = np.dot(self.state, ab_r)
178                 sigma += np.inner(ab_l, right_side)
179
180         if self.side == "top":
181             for k in range(self.qubits_out + 1):
182                 ba_l = np.kron(self.basis_b[index_i],
183                               self.basis_a[k])
184                 ba_r = np.kron(self.basis_b[index_j],
185                               self.basis_a[k])
186
187                 print(ba_r, ba_l)
188
189                 right_side = np.dot(self.state, ba_r)
190                 sigma += np.inner(ba_l, right_side)
191
192         return sigma
193
194     def compute_matrix(self):
195         a = [_ for _ in range(self.a_dim)]
196         b = [_ for _ in range(self.a_dim)]
197
198         entries_pre = [(x, y) for x in a for y in b]
199         entries = []
200

```

```
201     for i_index, j_index in entries_pre:
202         entries.append(self.get_entry(i_index, j_index))
203
204     entries = np.array(entries)
205     return entries.reshape(self.a_dim, self.a_dim)
```

LISTING E.8: Functions varies



# Bibliografia

- [1] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. The quest for a quantum neural network. *Quantum Information Processing*, 13:2567–2586, 2014. [Online; accedit en 19/09/2021: [Link](#)].
- [2] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. TensorFlow Quantum: A software framework for quantum machine learning. 2021. [Online; accedit en 19/09/2021: [Link](#)].
- [3] IBM quantum, 2021. [Online; accedit en 19/09/2021: [Link](#)].
- [4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 2014. [Online; accedit en 19/09/2021: [Link](#)].
- [5] He-Liang Huang, Yuxuan Du, Ming Gong, Youwei Zhao, Yulin Wu, Chaoyue Wang, Shaowei Li, Futian Liang, Jin Lin, Yu Xu, and et al. Experimental quantum generative adversarial networks for image generation. *Physical Review Applied*, 16(2), 2021. [Online; accedit en 19/09/2021: [Link](#)].
- [6] Gilbert Strang. Linear Algebra MIT OCW, 2011. [Online; accedit en 19/09/2021: [Link](#)].

- [7] Gilbert Strang. Matrix methods in data analysis, signal processing, and machine learning mit ocw, 2018. [Online; accedit en 19/09/2021: [Link](#)].
- [8] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10 edition, 2010.
- [9] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 66. Cambridge University Press, 10 edition, 2010.
- [10] Sheldon Axler. *Linear Algebra Done Right*, chapter 3, pages 37–58. Springer, 2 edition, 1997.
- [11] Sheldon Axler. *Linear Algebra Done Right*, chapter 3, pages 48–52. Springer, 2 edition, 1997.
- [12] Wolfram MathWorld. L2-Norm, 2021. [Online; accedit en 19/09/2021: [Link](#)].
- [13] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 74. Cambridge University Press, 10 edition, 2010.
- [14] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 73. Cambridge University Press, 10 edition, 2010.
- [15] Wikipedia. Tensor product, 2021. [Online; accedit en 19/09/2021: [Link](#)].
- [16] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, pages 80–96. Cambridge University Press, 10 edition, 2010.
- [17] Asher Peres. *Quantum Theory: Concepts and Methods*, chapter 2, pages 24–29. Kluwer Academic Publishers, 2002.
- [18] Eleanor Rieffel and Wolfgang Polak. *Quantum Computing: A Gentle Introduction*, chapter 2, pages 12–13. MIT Press, 1 edition, 2011.

- [19] Wikipedia. Inverter (logic gate), 2021. [Online; accedit en 04/10/2021: [Link](#)].
- [20] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 99. Cambridge University Press, 10 edition, 2010.
- [21] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 105. Cambridge University Press, 10 edition, 2010.
- [22] Eleanor Rieffel and Wolfgang Polak. *Quantum Computing: A Gentle Introduction*. MIT Press, 1 edition, 2011.
- [23] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, pages 105–106. Cambridge University Press, 10 edition, 2010.
- [24] Eleanor Rieffel and Wolfgang Polak. *Quantum Computing: A Gentle Introduction*, chapter 10, pages 212–213. MIT Press, 1 edition, 2011.
- [25] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software disponible a [tensorflow.org](https://www.tensorflow.org).
- [26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary De-

- Vito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [27] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. 2020. [Online; accedit en 26/10/2021: [Link](#)].
- [28] lucidrains. This person does not exist, 2021. [Online; accedit en 26/10/2021: [Link](#)].
- [29] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, 2019.
- [30] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Physical Review Letters*, 122(4), 2019.
- [31] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. Prediction by linear regression on a quantum computer. *Physical Review A*, 94(2), 2016.
- [32] Aram W. Harrow and John C. Napp. Low-depth gradient measurements can improve convergence in variational hybrid quantum-classical algorithms. *Physical Review Letters*, 126(14), Apr 2021.
- [33] Yudong Cao, Gian Giacomo Guerreschi, and Alán Aspuru-Guzik. Quantum neuron: an elementary building block for machine learning on quantum computers. 2017. [Online; accedit en 6/11/2021: [Link](#)].
- [34] Jonathan Romero and Alan Aspuru-Guzik. Variational quantum generators: Generative adversarial quantum machine learning for continuous distributions. 2019. [Online; accedit en 6/11/2021: [Link](#)].
- [35] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. Quantum convolutional neural networks. *Nature Physics*, 15(12):1273–1278, Aug 2019.

- [36] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.
- [37] Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. Quantum generative adversarial networks for learning and loading random distributions. *npj Quantum Information*, 5(103), 2019. [Online; accedit en 01/11/2021: [Link](#)].
- [38] Cirq Developers. Cirq, August 2021. See full list of authors on Github: <https://github.com/quantumlib/Cirq/graphs/contributors>.
- [39] Qiskit Developers. Qiskit: An open-source framework for quantum computing, 2021.
- [40] Stanislau Semeniuta, Aliaksei Severyn, and Sylvain Gelly. On accurate evaluation of gans for language generation. 2019. [Online; accedit en 19/09/2021: [Link](#)].
- [41] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*, chapter 2, page 62. Cambridge University Press, 10 edition, 2010.
- [42] David J. Griffiths. *Introduction to Quantum Mechanics*, chapter 1, pages 11–12. Prentice Hall, 1995.
- [43] Lov K Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996. [Online; accedit en 05/10/2021: [Link](#)].
- [44] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997. [Online; accedit en 02/01/2022: [Link](#)].