MSc Mathematical Modeling (SEMFE)

Study of the Quantum Advantage in Quantum Machine Learning Applications for drug discovery

Karakasis Ioannis
09320015

Supervisor Professor:
Andreas Stafylopatis
(E.E. Department NTUA)

16/06/2022

Περιεχόμενα

1. Abstract 5

1 Drug Discovery methods **7**

1. RDKIT package 10

2 Calibration Data-set and Data Exploration **12**

3 Neural Network Models for Classical Computing **21**

1. Difficulties using GNNs 22
2. Types of Graph Neural Networks 22
3. GNN Algorithm with a focus on Application of Convolutional Networks on Graphs 23
4. Generative Adversarial Networks 25
5. MolGAN model 27
6. Quantum Neural Networks 28
7. Setting the base for the Quantum era 28
8. Data Encoding in QML 32
9. Angle Encoding 35

10. Variational Auto-encoders 37
11. Quantum VAEs 40
12. Quantum Graph Neural Networks 43
13. Quantum GANs 46

4 ML Models of Classical computing - Experimental Implementation 50

1. Convolutional Neural Network (CNN) 55
2. Graph Neural Network (GNN) 60
3. Generative Adversarial Network (GAN) 63

5 Quantum ML Models - Experimental Implementation 68

1. How to Run Quantum Algorithms 68
2. Comparison of Encoding Methods for Quantum Variational Classifiers 72
3. Quantum Convolutional Neural Networks (QCNN) 82
4. Quantum Generative Neural Networks (QGAN) 92

6 Experimental Results - Conclusion 100

7 Bibliography 103

8 Appendix 107

1. Dataset Pairplot before Encoding 108
2. QCNN - Quantum Circuit 109

The present thesis was implemented within the framework of the postgraduate program of the School of Applied Mathematics and Natural Sciences, Mathematical Modeling in Modern Technologies and Finance, at the National Technical University of Athens.

I would like to say a huge thank you to my supervising professors Georgios Siolas and Andreas Stafilopatis who for the last year have been helping and guiding me on this new journey of Machine Learning and Quantum Computers. From the beginning of our collaboration and throughout the writing of this paper, he was by my side and his presence was key to overcoming several difficulties in its development.

Finally, I would like to thank my parents Natalia and George and all the people close to me who supported me throughout this effort. This study is dedicated to a close friend and aims to raise awareness of the stressful conditions that affect nowadays society and especially young people.

1. Abstract

This thesis was authored under the postgraduate program, ”Mathematical Modeling in Cutting Edge Technologies Finance”, of the School of Applied Mathematics and Physic Sciences of the National Technical University of Athens. The study supporting the specific thesis is focused on proving how Classical Quantum Deep Learning Models can be used to solve a classification problem. The dataset chosen for this task was the QM9 dataset containing information on molecules and was limited to approximately 20 thousand molecules that can possibly be used to classify drugs that can be used as tricyclic antidepressant drugs or to design new molecules based on this benchmark dataset. However, the main purpose of this thesis was to try Quantum Machine Learning models on this dataset and prove that the quantum analogous of the models achieve the quantum supremacy. By the term quantum supremacy, we mean the quantum advantage provided on solving computational problems, when someone utilises a Quantum Computer. I tried to use approximately the same parameters in order to be able to compare the models. The results that we analysed showed that all quantum algorithms used performed better than their classical analogous or at least on the same level. The quantum models that didn’t outperform their classical analogous in accuracy, they outperformed them in run-time. We observed models that performed in 1-2% of the time that their classical analogous performed. These findings are very important as Quantum Computing and especially Quantum ML algorithms, is a field rapidly growing but still in its preamble stage and thus there is no abundance of experimental data. As described, it is very hard to test all the above in real quantum computers as they are limited, they are underdeveloped and are hard to access to run extended studies. Although, we ran all our algorithms on simulations of quantum computers with the help of Cirq package by Google, the pennylane package, which is focused in Quantum ML and the qiskit open source package provided by IBM, it is important to know the behavior of the quantum computers for when the time that we can build more complex architectures to be able to test their performance and be able to exploit them.

Introduction

The purpose of this thesis is to discover how the field of drug discovery can be extended with the use of Quantum Machine Learning algorithms. More specifically, I will study how some known techniques can be applied to medical data sets and try to define some conclusions on the performance of the predictive models, in order to understand which one is more suitable for specific tasks, such as drug discovery.

The ordinary methodologies used for drug discovery are via the experimental process and require many trials, consisting these methods time consuming and very costly. These drove us to use modern tools such as simulations and predictive models to exploit the abundance of data and map how the drug materials behave. One needs to start his study from the characteristics of atoms and molecules to entire healthcare databases on how to combine them to find treatment to the most common diseases. However, today's classical computers and even supercomputers can only simulate relatively simple molecules, limiting our capabilities to study more complex systems. Therefore, the focus of research is to achieve the so-called quantum supremacy. This can be achieved with the use of quantum computers, which exploit the laws of quantum mechanics to process information. Quantum computers use quantum bits, called qubits, which can encode zero and one simultaneously. This phenomenon is called the superposition of a quantum state.
Quantum Machine Learning uses quantum algorithms to perform complex machine learning tasks. Quantum ML allows us to translate ML algorithms into a quantum circuit that can take into account the architecture of the quantum computer and run more efficiently. Most of these algorithms are of course hybrid models using a combination of classical and quantum models. Companies and researchers have experimented more on these hybrid algorithms, because at the moment they are the more affordable choice. Hybrid algorithms combine the storage capacity of a classical computer, which is higher of that of a quantum computer, with the computational power of a quantum machine.

Κεφάλαιο 1

Drug Discovery methods

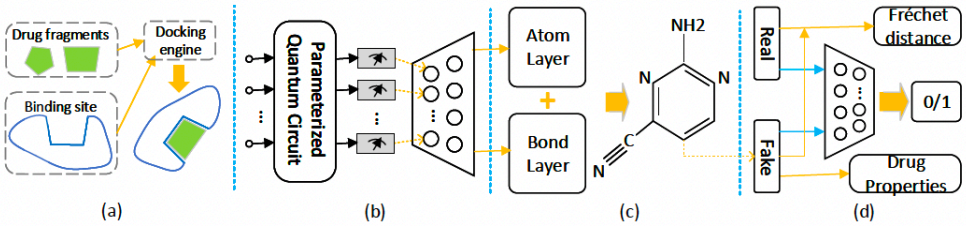
Target identification, molecular design, preclinical investigations, and clinical trials are all considered stages in the drug development pipeline. Searching for new pharmaceuticals can be thought of as navigating in chemical space, which is the collection of all organic molecules, and navigation in unexplored chemical space falls under the category of ”de novo” drug design. For the scope of this work, we chose to investigate how the most well-known machine learning techniques, such as Variational Autoencoders (VAEs), generative adversarial networks (GANs), and recurrent neural networks (RNNs), compare to each other and can be used to learn latent representations of molecules and generate large numbers of drug candidates for further high-throughput screening. To overcome the difficulty of dimensionality, we would need to use the quantum equivalent of all the aforementioned models to sample molecule compounds from sections of the chemical space that may be on the order of 10^{60}. GANs, for example, find therapeutic candidates by producing molecular structures that obey chemical and physical properties and have a high affinity for binding to a target disease's receptor. Instead, a QGAN, a qubit-efficient system with a hybrid generator, can be used to learn richer representations of molecules by searching exponentially huge chemical spaces with few qubits more effectively than a classical GAN. A hybrid quantum generator that supports varying numbers of qubits and quantum circuit layers and a conventional discriminator are usually used in these types of models.
Quantum GAN is one of the main applications of near-term quantum computers due to its strong expressive power in learning data distributions even with much less parameters compared to classical GANs. However, quantum neural network is still at its nascent stage due to qubit constraints on noisy quantum computers. Considering the specific task of drug discovery, we explore potential quantum advantages for both generative and predictive models due to the following reasons:

- Gate parameter exploration in Hilbert space is different from neural network parameter exploration
- Given a chemical region abundant of molecules, the inherent probabilistic nature of quantum systems helps generate more diverse and novel molecules surrounding that region.

Quantum ML involves parameter optimization of parameterized quantum circuits to obtain a desired input-output relationship.

Another popular application of QML in drug discovery is the use of Graph Neural Networks combined with libraries such as RDKIT or DeepChem to pre-process them. Some popular experiments are the prediction of molecules' toxicity and solubility. The first is a classification task, in contrary to the second, which is a regression task. Thus, we see that the prediction of each one of the molecule's properties may require different types of models that suit better each problem's conditions. The common theory behind most of the properties is the Density functional theory, which is used to calculate some molecular properties such as energies and enthalpies as targets.

The traditional method of drug discovery involves determining the characteristics of a receptor and the compounds that can facilitate its binding. Again, this is done using density functional theory. A fingerprint similarity method can also be used to identify potential targets of other molecules that are active against the receptor. The MolGAN model uses a generative adversarial network to learn and generate drug-like molecules based on the QM9 dataset. To improve the quality of generated compounds and stabilize the GAN training, a reward network based on chemical validity, RDKIT, is used. If we want to use Quantum ML, we can use a hybrid quantum-classical model, which is a variation of the MolGAN. This model us presented below:



Σχήμα 1.1: QGAN

The proposed model, from literature, has been proven to be qubit-efficient and

includes a hybrid generator and a classical discriminator. The model efficiently learns molecule distributions based on MolGAN. As we can see in 1.1:

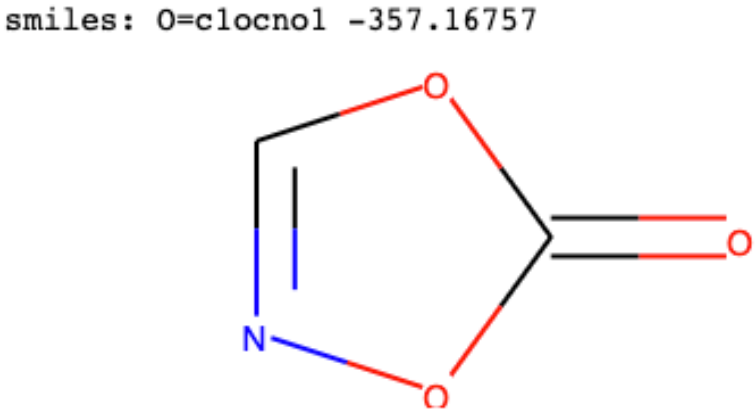
- (a) the validity of the generated molecules depend on if they have high affinity towards the receptor binding sites.
- At the second part (b), the quantum stage is displayed and
- In the the third section (c) an atom layer and a bond layer are applied to generate synthetic molecular graphs.
- Finally, in (d) section a batch of real molecules from the training dataset (QM9) and a batch of synthetic molecules that are generated from (c) stage are fed into the classical discriminator for real prediction and the score calculation.

In the specific paper, they used the Frechet distance as a criterion for scoring. After the above process, all results on the drug properties are evaluated using the RDKIT package.

If a probabilistic search is preferred, the receptor can be treated as a $N \times N \times N$ picture with 8 channels corresponding to the atom types. The ligand will be regarded as a $N \times N$ adjacency matrix with 6 bond type channels and a length N atom type vector with 7 atom type channels. At this point, it is important to mention that the maximal value is $N = 32$ atoms, which makes this approach feasible for generating large molecular graphs. To continue with, two variational autoencoders were used, one for the receptor pocket and one for the ligand. They will be utilized to match receptor pocket and ligand pairs. They are made up of three-dimensional convolutional layers that compress the original image into a 32-element latent representation before decompressing it back into the initial receptor pocket image. A two-way matching network connects the two VAEs, converting the receptor pocket's latent representation into the ligand's latent representation and vice versa. The activation function used in this study was a LeakyReLU combined with spectral normalization to prevent exploding/vanishing gradients. The Hybrid VAE is created by inserting quantum layers into the neural network at specific points. The pain point discovered during this experimental process was that the quantum embedding stage limited the model's quantum superiority. The quantum embedding used was angle encoding, followed by a data re-uploading method for measuring different techniques for quantum state preparation.

1. RDKIT package

RDKIT is a popular tool in cheminformatics that is written in C++ and Python and allows users to directly calculate molecule similarities. Graph embeddings could be used to perform a more thorough similarity search on molecules using classical algorithms. For many tasks, such as creating a molecule, this package employs SMILES(Simplified Molecular Input Line Entry System) strings. SMILES strings contain a letter that represents an atom as well as other symbols that contain information about the bonds between the atoms. Another task that RDKIT excels at is loading and visualizing molecules. When we visualize a molecule, we get an image that looks like below:



Σχήμα 1.2:

All of the substructures of the molecule contained in the specific smile can be seen in the above image. This is extremely useful information because the behavior of a molecule as a drug could be determined by certain substructures contained in this SMILE. We can conclude from this that molecules with similar key substructures are likely to behave similarly. When comparing different molecules, we can use RDKIT to match these parts. Tanitomo similarity is one criterion provided by RDKIT that can be used to compare them. This is a popular metric that uses substructure matching.

After loading and visualizing the molecules, one can use RDKIT to search similarities between them. This could be achieved by starting with a molecule that we know its properties and its behaviour and search for similarities with molecules in a large database, such as QM9 and that can be used as vanilla. If the information and properties provided by SMILES while working with RDKIT are insufficient, SMARTS can be used. SMARTS is a generalization of SMILES that can tell you

whether or not a molecule has rings. However, molecular fingerprints are utilized to encode the structure of a molecule in every case, as we discuss elsewhere in this thesis.These are mostly used to compare molecule substructures, characteristics, and the vast majority of information stored within a molecule. In a molecule, they hold each pattern of a specific size.

Κεφάλαιο 2

Calibration Data-set and Data Exploration

The main issue with this thesis was the lack of experimental data and the lack of variety to different types of implementations. Thus, one of the most important steps following in our procedure was to choose a well known data-set that could be used to calibrate our model. The wrong choice of data-set could lead to skewed results and false conclusions in our experiment.

For the scope of this thesis, we used a data-set generated from the Quantum Machines 9 (QM9) database. This data set contains over 134 thousand rows of information and provides us with quantum chemical parameters for a relevant, consistent, and extensive chemical space of small size organic compounds. This database may be used for bench-marking existing methods, developing new approaches such as hybrid quantum mechanics/machine learning, and systematic identification of structure-property connections, as stated in its documentation. The quantum chemistry properties mentioned above include geometric, energetic, electronic and thermodynamic properties. In more detail, this database can be used to report energy-minimal geometries, related harmonic frequencies, dipole moments, polarizabilities, as well as atomization energies, enthalpies, and free energies. In addition to structure information, QM9 incorporates SMILES from GDB9 and for relaxed geometry, as well as InChl for GDB9 and for relaxed geometry. The QM9 dataset is based on GDB9, the most general database of molecular quantum computations. However, the high computing cost prevents it from being used on a regular basis for extensive chemical space exploration.

Now, to further explain the above information. With the word SMILES we refer to the ”Simplified Molecular Input Line Entry System”, which is a chemical

representation that uses a sequence of characters to encode the molecule. SMILE mainly include symbols of atoms and their bonds, in addition to syntax rules that designate the validity of each entry of a string. The features derived from SMILES representation are called topological descriptors, since the feature extraction process relies on the molecule files. Finally, The advantages of using SMILES are

- an interpretation of the molecules that is more friendly to the user,
- they are simple to encode the molecular graph in or vice-versa,
- they are widely used and are a popular way for molecule representation and
- they consist a representation that is faster to compute.

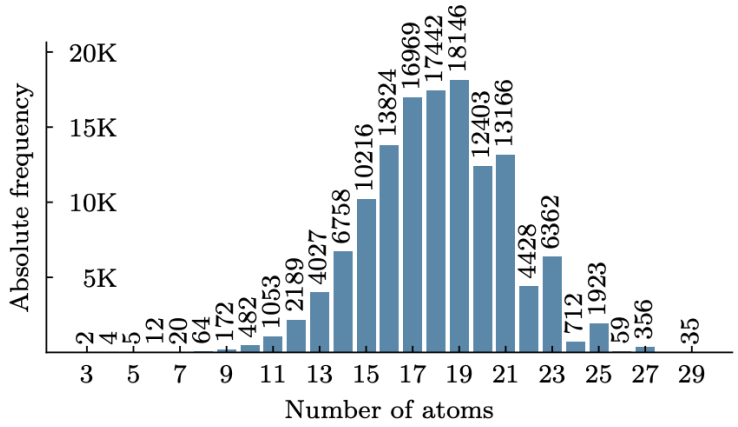
If one wants to study SMILES and data-sets like QM9 and approach these problems, firstly he has to follow the preprocessing methodology. Fortunately, for most of the cases this step is covered by libraries used in deep learning projects such as Deepchem or RDKIT, as mentioned in the previous chapter which is dedicated to the description of the RDKIT package. They offer useful features for molecular data, such as data loaders, splitters, featurizers, metrics, and even GNN models. These routines allow you to use SMILES as string representations of the molecule's 2D or 3D structure. It's important to remember that this step translates any molecule to a particular string that is almost certainly unique and can be translated back to the 2D structure. Additionally, various molecules may be assigned to the same SMILES string, lowering the model's performance. The binary vectors obtained from the SMILES are then used to represent whether a specific substructure of the molecule is present or not, and these vectors are referred to as fingerprints.

Now, with the use of RDKIT package and some basic data analysis, we can obtain some very important information for our data-set. Firstly, one can see the frequency in which the number of atoms appear in our dataset. Firstly, we can print the description of each label of the dataset that in reality each label depicts to one property.

I.	Property	Unit	Description
1	tag	-	"gdb9"; string constant to ease extraction via grep
2	index	-	Consecutive, 1-based integer identifier of molecule
3	A	GHz	Rotational constant A
4	B	GHz	Rotational constant B
5	C	GHz	Rotational constant C
6	mu	Debye	Dipole moment
7	alpha	Bohr^3	Isotropic polarizability
8	homo	Hartree	Energy of Highest occupied molecular orbital (HOMO)
9	lumo	Hartree	Energy of Lowest occupied molecular orbital (LUMO)
10	gap	Hartree	Gap, difference between LUMO and HOMO
11	r2	Bohr^2	Electronic spatial extent
12	zpve	Hartree	Zero point vibrational energy
13	U0	Hartree	Internal energy at 0 K
14	U	Hartree	Internal energy at 298.15 K
15	H	Hartree	Enthalpy at 298.15 K
16	G	Hartree	Free energy at 298.15 K
17	Cv	cal/(mol K)	Heat capacity at 298.15 K

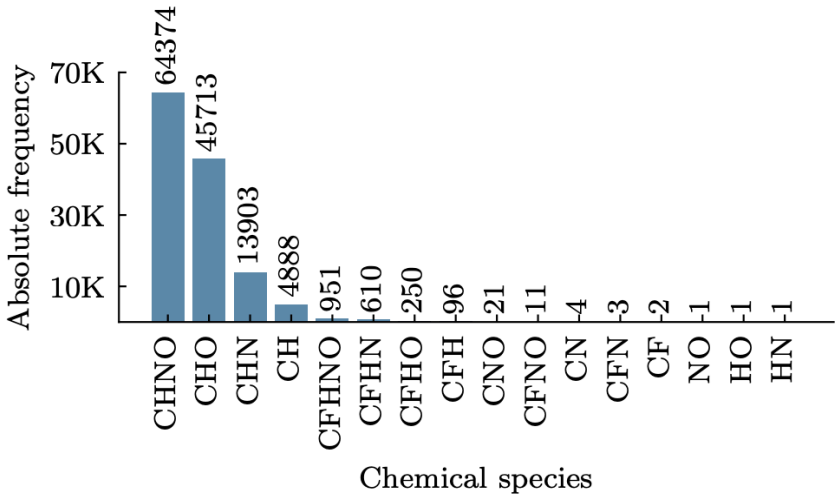
Σχήμα 2.1: Description of each property

Next we can plot the distribution based on the number of atoms in each molecule and based on the number of molecules by the chemical species.



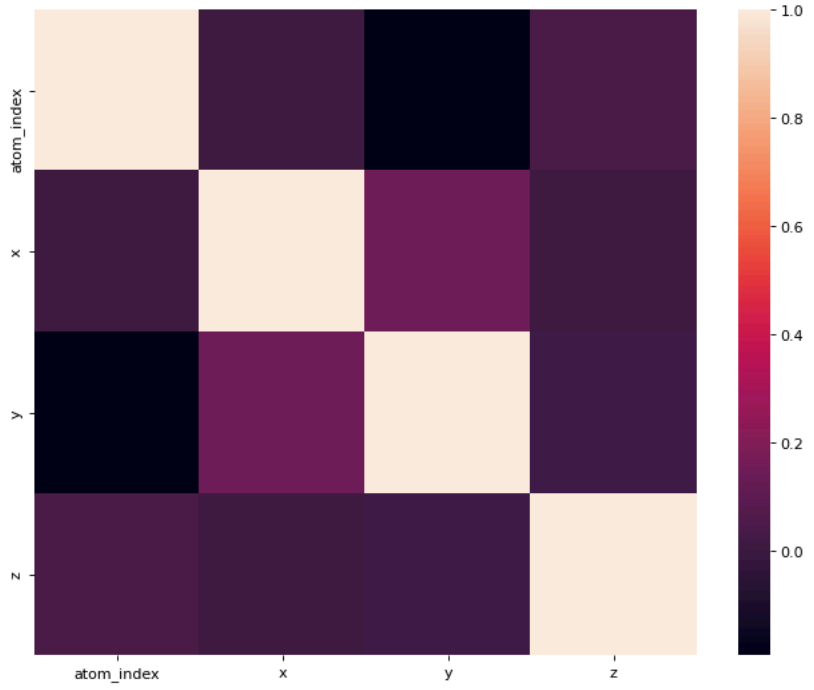
Σχήμα 2.2: Atoms Histogram

One can assume that the distribution that appears can be fitted be a Gaussian function with a center average of 19 atoms and that the limits are [3,29].



Σχήμα 2.3: Histogram based on the chemical species

Now, just to simplify our calculations we take a subset and do some data exploration to further understand our dataset. So if we want to see the correlation of our spatial data combined with the atom index we plot the heatmap representation below.

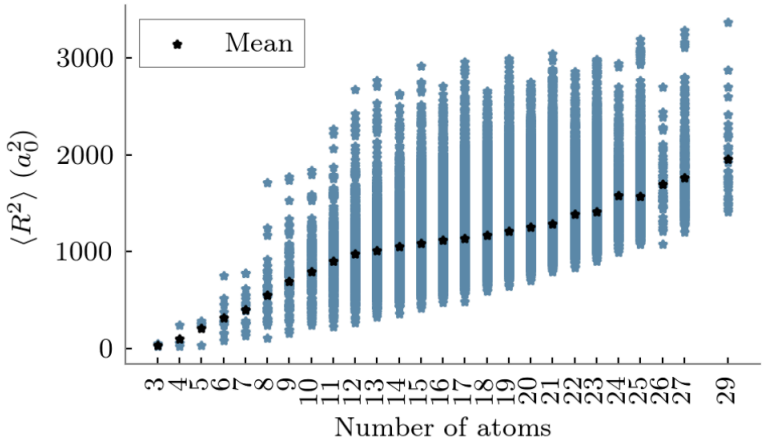


Σχήμα 2.4: Correlation of Spatial Data

and then in the table below we display some basic statistic analysis on this subset calculating the mean, standard deviation, etc.

	Atom Index	**x**	**y**	**z**
count	2.35887e+06	2.35887e+06	2.35887e+06	2.35887e+06
mean	9.757255e+00	9.495981e-02	-3.335625e-01	6.239050e-02
std	5.592444e+00	1.655403e+00	1.989328+00	1.445876e+00
min	0.0000000e+00	-9.234889e+00	-9.933938e+00	-9.134765e+00
25%	4.0000000e+00	-8.746228e-01	-1.826097e+00	-8.424758e-01
50%	9.0000000e+00	5.183962e-02	-4.034906e-01	1.092888e-02
75%	1.3000000e+00	1.116163e+00	1.373848e+00	9.393901e-01
max	2.8000000e+00	9.382240e+00	1.018196e+01	7.8894733e+00

Here I present a plot scatter plot that shows the electronic spatial extent $< R^2 >$ by the number of atoms.

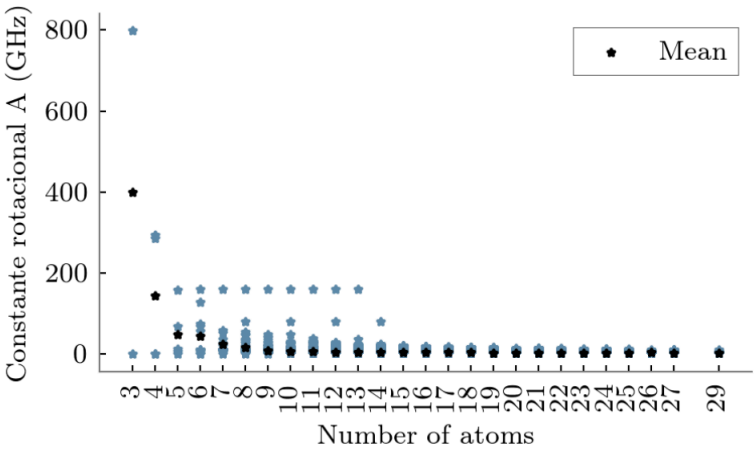


Σχήμα 2.5: Electronic spatial extent $< R^2 >$

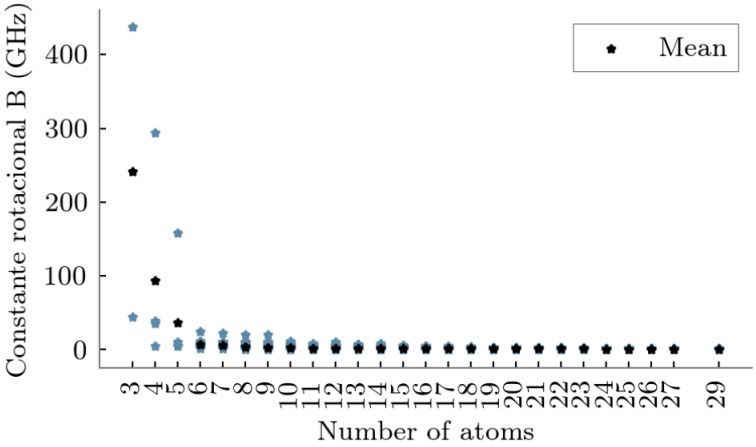
Next, we can easily calculate the abundance of the atoms in the chosen subset:

H	51.231456%
C	35.262954%
O	7.766499%
N	5.612082%
F	0.127010%

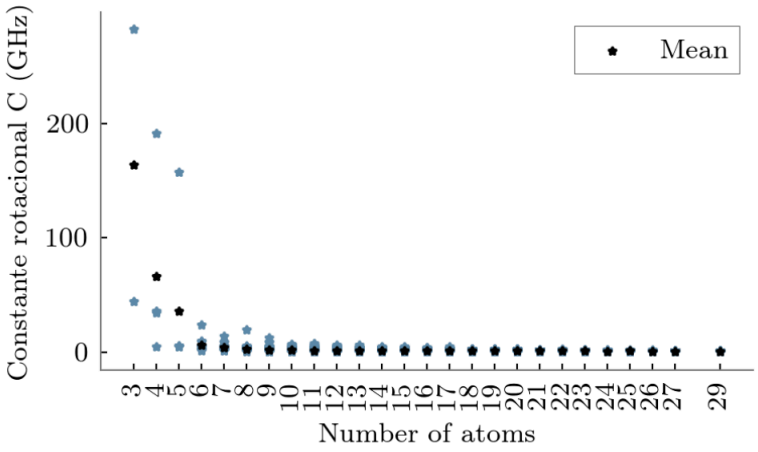
To continue with, we create some scatter plots for the rotational constants:



Σχήμα 2.6: Scatter plot for A rotational constant

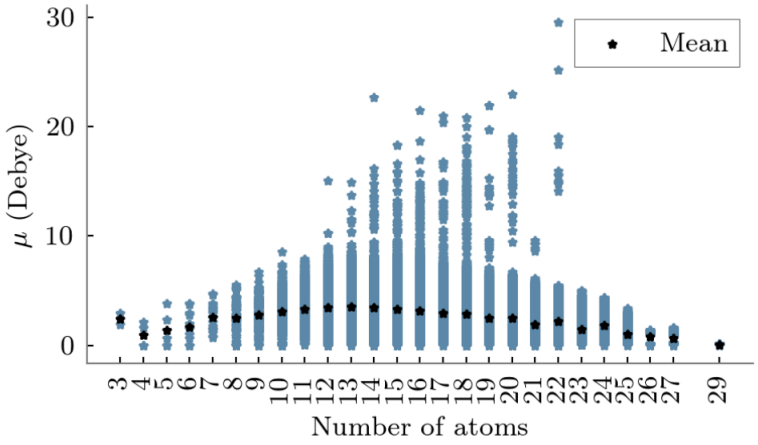


Σχήμα 2.7: Scatter plot for B rotational constant



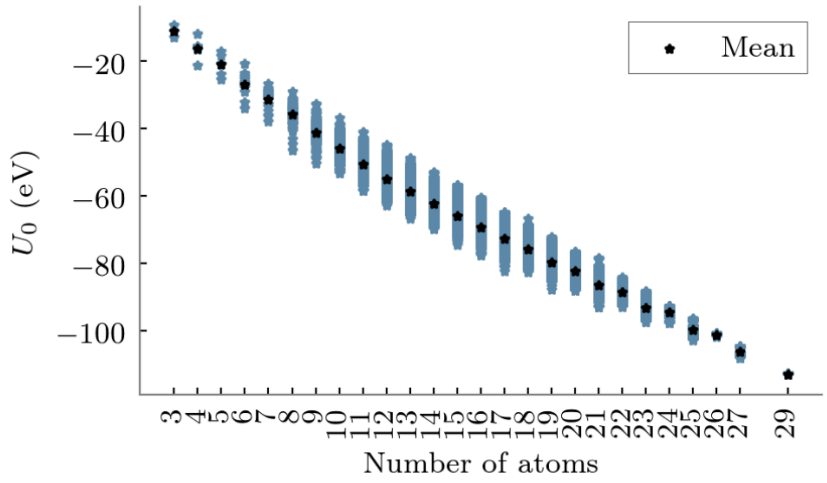
Σχήμα 2.8: Scatter plot for C rotational constant

Then I create a similar plot by substituting the dipole moment on the y-axis.

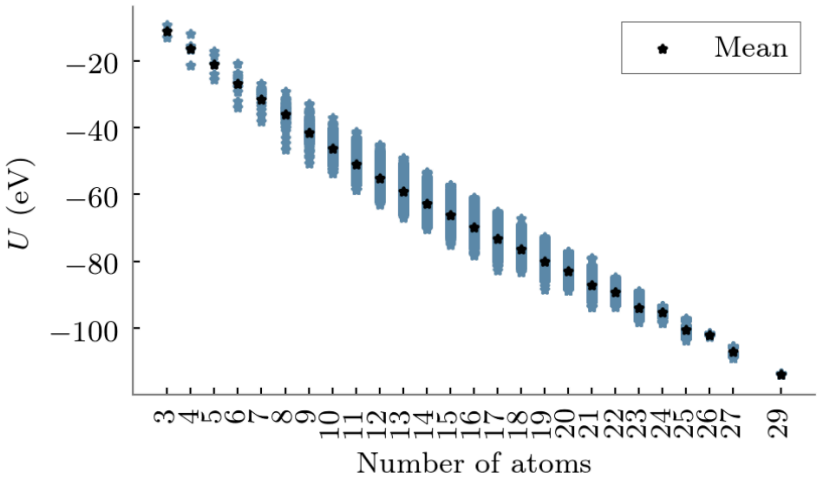


Σχήμα 2.9: Dipole moment vs Number of atoms

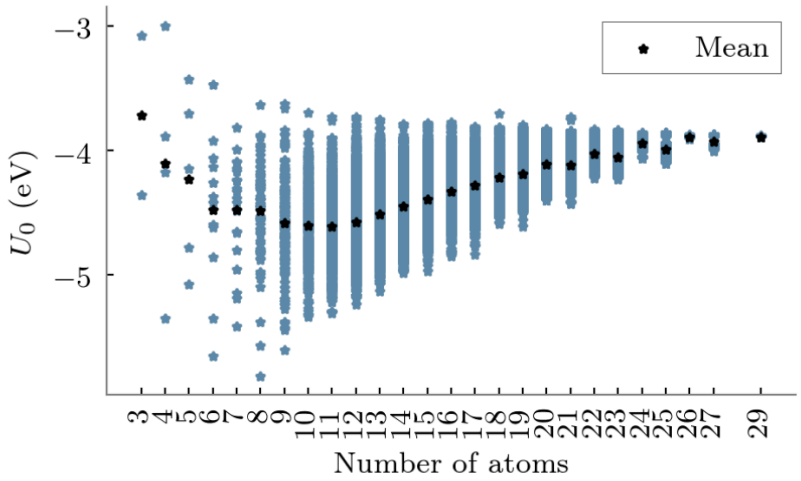
Finally, I display some energy enthalpy plots as we can extract many assumptions for the system of each molecule from these parameters.



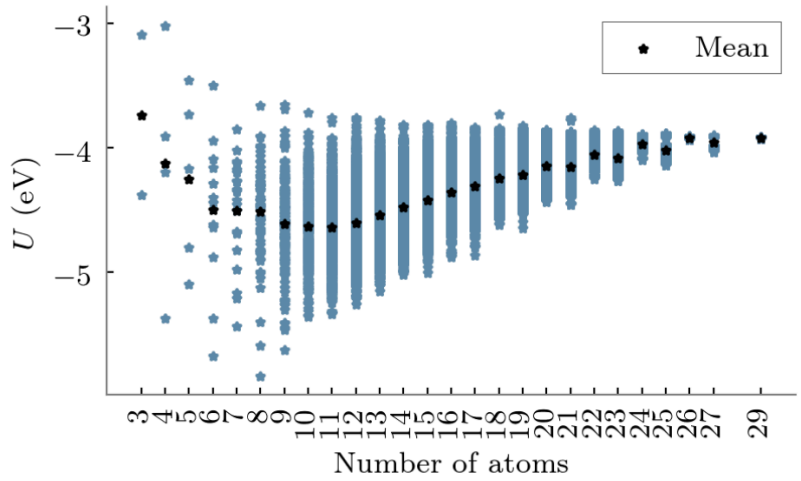
Σχήμα 2.10: Internal Energy at 0K by the number of atoms



Σχήμα 2.11: Internal Energy at 298.15K by the number of atoms



Σχήμα 2.12: Internal Energy at 0K per atom by the number of atoms



Σχήμα 2.13: Internal Energy at 298.15K per atom by the number of atoms

Κεφάλαιο 3

Neural Network Models for Classical Computing

For this particular experiment we decided to use several variations of Convolutional Neural Networks but the one type of Neural Network that made more sense to use and that we hoped that would fit the best to our use case, was the category of Graph Neural Networks. As mentioned in previous chapters, our data-set primarily contains the geometric properties of the molecules and their coordinates in the 3 dimensional space. Thus, one can resemble the combination of atoms to simple graphs. In our case, we have information about the different molecules consisting of their 3D coordinates and the atom type. So imagine the atoms as the nodes and the chemical bonds as edges. This is the simplest type of modeling one can use. Now, with the use of Graph Neural Networks we have the ability to predict various kinds of molecular properties. This can be achieved via the Density functional theory to calculate properties such as energies and enthalpies as our targets.

In order to genaralize the abilities of these models for all cases, GNN models is making four types of predictions. Beginning from local level, a GNN can predict the property of a node. Then, due to the neighborhoods in graphs this type of models can predict the link between two nodes and combining this with the previous prediction it can come to assumptions on the whole property of a graph. After that, we can scale this pattern and apply similar techniques to predict the similarity between two nodes or two graphs.

1. Difficulties using GNNs

As is well known, Graph Neural Networks are included in the deep learning chapter. The majority of neural networks are designed to cope with fixed-size or regularly arranged inputs. We cannot apply the same principle to GNN models in this case. The intricacy of the topological structure of networks or graphs is one of the most prevalent issues that arise. We cannot assume, as a general rule, that the input will be in the form of an image with a squared grid of pixels and follow specific rules on precise dimensions and density. In some cases, graph inputs lack spatial locality and uniformity. With this as a fact, we can see why the connections between the nodes are a result rather than a fact. In addition to this, there is lack of fixed node ordering or any reference point, adding difficulty to our study as we cannot assume the isomorphism of the problem. And if the spatial issues are not enough, another major aspect one should consider is that it includes different entities as nodes and different types of interactions between the nodes, the edges. Finally, what it makes it such a special case is that it has rich and heterogeneous features about entities and interactions.

2. Types of Graph Neural Networks

If we choose this type of network, we must also choose which type of GNN to employ on a second level. We must first choose between a Recurrent Graph Neural Network, a Spatial Convolutional Network, and a Spectral Convolutional Network. The Recurrent GNN's rationale is to define a parametrized function fw with which the final node state will be used to produce an output after k iterations to make a judgment regarding each node. The Banach Fixed-Point Theorem is used to construct this model. The difference between this and the Spatial Convolution Network is that the latter is more like CNN. This aggregates the features of neighboring nodes into the center node. The last two categories, in general, broaden the convolution process from grid data to graph data. The representation of a node v is created by combining its own features x_v and those of its neighbors x_u. Convolutional Graph Networks, unlike the first category, stack many graph convolutional layers to extract high-level node representations and aid in the development of more complicated GNN models. They also employ a set number of layers, each with a varied weight. By adding filters from the standpoint of graph convolutions by introducing filters from the perspective of graph signal processing, where the graph convolutional operation is regarded as removing noises from graph signals, spectral-based techniques construct graph convolutions. RecGNNs inspired spatial-based ways to define graph convolutions via information propagation. These outmatch the rest in efficiency,

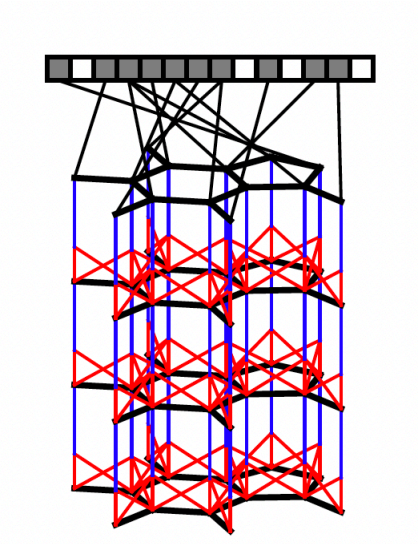
flexibility and generality.

This type of GNNs aims to learn node representations with recurrent neural architectures, starting with a more extensive investigation of Recurrent Graph Neural Networks. The underlying premise is that each node in a graph trades information with its neighbors continuously until a stable equilibrium is found. As a result, it works on one of the most fundamental principles of all physical systems: the quest for the lowest energy point. RecGNNs extract high-level node representations by repeatedly applying the same set of parameters over nodes in a graph.

In terms of efficiency, spatial models outperform spectral models in the final two categories. Spectral models must either compute eigenvectors or manage the entire graph at the same time. As you may imagine, both time and computing power are valuable commodities. Spatial models, on the other hand, are more scalable to huge graphs since they conduct convolutions directly in the graph domain via information propagation. In addition, rather than computing the entire graph, the computation can be done in batches of nodes. Spectral models, on the other hand, extend poorly to new graphs since they rely on a graph Fourier basis and presuppose a fixed graph. For all perturbations to the graph, this causes the eigenbasis to change. Spatial-based models, on the other hand, conduct graph convolutions locally on each node, allowing weights to be transferred easily across multiple locations and structures. Furthermore, spectral-based models can only work with undirected graphs. Because graph inputs such as edge inputs, directed graphs, signed graphs, and heterogeneous graphs may be easily incorporated into the aggregation function, spatial-based models are more versatile in handling multi-source graph inputs.

3. GNN Algorithm with a focus on Application of Convolutional Networks on Graphs

In this section of this thesis, we will shortly analyse a type of convolutional neural network that operates directly using as input graph structures and in most areas is better to apply on Molecular Fingerprints. The nature of the problem as shown below: To examine chemical fingerprints with a convolutional network, first one has to create a graph that matches the architecture of the molecule, with nodes representing atoms and edges representing chemical bonds. Fortunately, the SMILES in the QM-9 data set follow a similar rationale. Information travels between neighbors in the graph at each layer of the CNN. Finally, each network node activates one bit in the fixed-length fingerprint vector.



Σχήμα 3.1: GCN from literature

As previously discussed, one of the challenges with using any type of neural network is that they will almost certainly require fixed size inputs, as most machine learning pipelines cannot manage input of arbitrary size shape, which would be a major issue for researching molecules. This issue arises because molecules can be any size or form. Using a differential neural network with a graph as input, the molecular fingerprint vectors can be computed more quickly. In the [] publication, D. Duvenaud and colleagues developed a model that uses several layers, each of which applies the same local filter to each atom and its surroundings, with the final layers combining characteristics from all of the atoms in the molecule using a global pooling step. Each feature of a circular fingerprint vector can each only be activated by a single fragment of a single radius and in contrast to that, neural graph fingerprint features can be activated by variations of the same structure, making them more interpretable, and allowing shorter feature vectors.

This approach has superior predictive ability when compared to traditional fingerprints for solubility, pharmacological efficacy, and organic photovoltaic efficiency data. The approach can be streamlined to encode just relevant features by employing differentiable fingerprints, lowering downstream computing and regularization needs, and, as previously indicated, feature activation becomes much simpler and meaningful. However, computing the neural fingerprint of depth R, length L of a molecule with N atoms using a molecular convolutional net with F features at each layer costs $O(RNFL + RNF^2)$, which is a computationally expensive method. Furthermore, their implementations are still insensitive to stereo-isomer classification.

Graph NNs are better suited for applications like node classification, graph classification, node clustering, link prediction, and influence maximization. The major purpose of this thesis is to use it as a graph classification model to categorize different molecules, or even as a link prediction model to investigate how different types of atoms join to produce new molecules. We've discussed Convolutional Networks, and now we'd like to apply these convolutions to graphs. The goal of this exercise is to extend convolutions to any graph.

The basic idea behind graph convolutions is that it includes simple calculations, as each node aggregates info from surrounding nodes to update its embeddings. The feature vector from each neighboring node of the prior layer are aggregated with the top-right node itself. Next we can apply a non-linear transformation on the aggregated node vector to arrive at the updated embedding. We choose this to be a trainable weight matrix followed by an activation function (e.g. ReLU). Here is important to note that the application of the update function and the weight matrix is uniform across all nodes of that layer, meaning that the mechanism of embedding is the same which makes sense since even with CNNs the image filter weights are shared across the same layer. To extend this beyond a single-layer Neural Network, we can a deeper NN to construct the above transformation. Finally after the above steps, the subsequent layer's corresponding node is updated to the resulting feature embedding. The flow can be described as beginning from the input graph, followed by the GNN blocks. After these the transformed graph is implemented, in which a classification layer is added and finally we reach the prediction point.

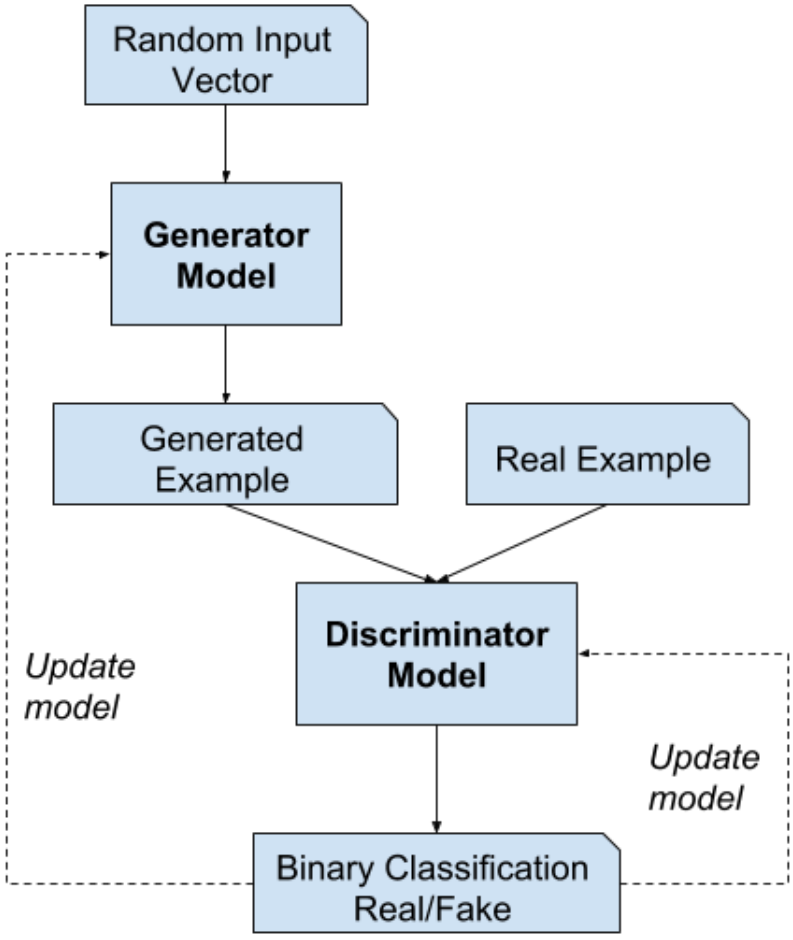
4. Generative Adversarial Networks

In classical machine learning, GANs are mostly used in data generation like high-resolution images. However, it has been observed that these kind of models are computationally expensive. Thus, the quantum analogous are here to overcome them. The purpose of using QGANs is a too in the race to thrive in the optimization process while implementing quantum circuits to produce a good estimation in any high-complexity problem.

To start with, in the classical world Generative models belong to unsupervised machine learning and in the exact case of Generative Adversarial Networks are being used as a training method for these models by framing the problem as a supervised learning problem with two types of models:

- The Generator Model which is trained to generate new data and
- The Discriminator Model with goal to classify these data and understand which are fake and real.

These two models are competitive with one another and as a whole, GANs, provide a method to domain-specific data augmentation and solutions to problems that require a generative solution. The model described by Ian Goodfellow in **??** and was the first proposed method that freed the user from Markov chains and approximate inference networks during training or sample generation.

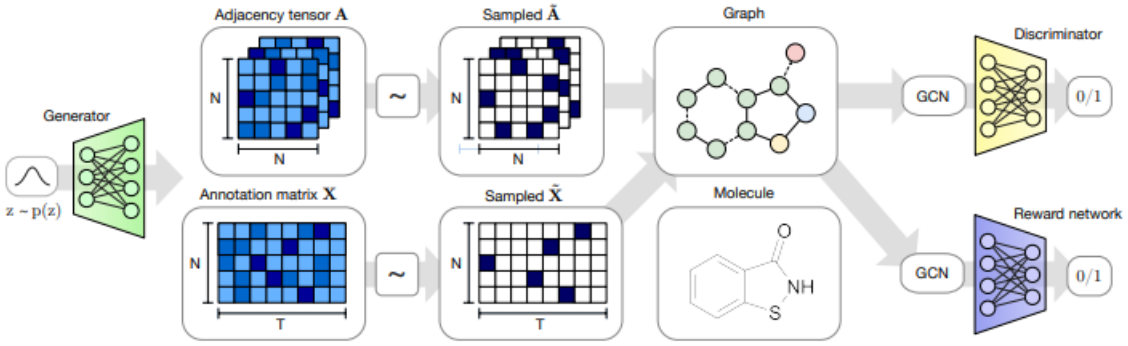


Σχήμα 3.2: GANs model

I provide more specific information for the use case of drug discovery by explaining MolGAN model in the next subsection.

5. MolGAN model

For the scope of this thesis, however, we focus in a specific type of GAN, the MolGAN. Thus model consists of three main components: a generator G_θ, a discriminator D_ϕ and a reward network $\hat{R}_\psi$.



Σχήμα 3.3: MolGAN flow

As shown in 3.3 the generator takes a sample from a prior distribution and generates a dense adjacency tensor A and an annotation matrix X. Subsequently, sparce and discrete A and X are obtained from A and X respectively via categorical sampling. The combination of A and X represents an annotated molecular graph which corresponds to a specific chemical compound. Finally, the graph is processed by both the discriminator and reward networks that are invariant to node order permutations and based on Relational-GCN layers. For the training of the discriminator the WGAN model's loss function is used, while the generator uses a linear combination of the WGAN's loss and RL loss:

$$L(\theta) = \lambda L_{WGAN} + (1-\lambda)L_{RL} \tag{3.1}$$

where $\lambda \in [0,1]$ is the hyperparameter that regulates the trade off between the two components and L_{WGAN} is given by:

$$L(x^{(i)}, G_\theta(z^(i); \phi) = -D_\phi(x^{(i)}) + D_\phi(G_\theta(z^{(i)})) + a(||\nabla_{\hat{x}^{(i)}} D_\phi(\hat{x}^{(i)})|| - 1)^2 \tag{3.2}$$

where the first part is the original WGAN loss and the second part is a gradient penalty, with a: a hyperparameter, $\hat{x}^{(i)}$ a sampled linear combination of $x^{(i)}$ and $G_\theta(z^{(i)})$. G_θ is the generative model, D_ϕ a discriminative model that learns to classify whether samples came from the data distribution rather than from G_θ.

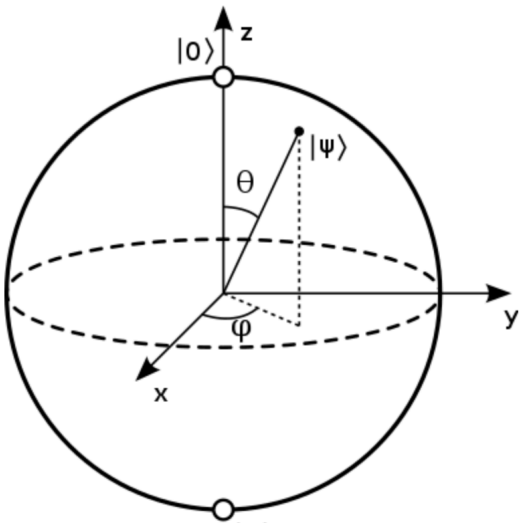
6. Quantum Neural Networks

7. Setting the base for the Quantum era

While entering the era of Quantum computers, we are entering the probabilistic era. We turn from the classical unit of information, the bit, to the quantum analogous, the qubit. The basic difference that we meet in quantum computing is the existence of superposition states. What do we mean by superposition? To answer that we first have to understand what a qubit is. In classical computing, the bit can receive two values, 0 and 1. In a qubit, we don't use 0 and 1 as numbers but we use $|0>$ and $|1>$ as two quantum states of a system, called quantum bit or qubit. Quantum computers are built to exploit the nature of qubits by obeying the laws of quantum mechanics. So according to the quantum theory, one particle enters a superposition of states, in which it behaves as if it were in both states simultaneously. Thus, now the number of computations that a quantum computer could undertake is 2^n, where n is the number of qubits used in this computer. The state of a qubit can be written as:

$$|q>=\alpha|0>+\beta|1>$$

where $|0>$ is a state described by the vector $(1,0)$, $|1>$ a state vector like $(0,1)$ and the constants α, β are two constants defining the % probability that the qubit will be in each state and by the probability theory $|\alpha|^2+|\beta|^2$. These are visualised in a unitary sphere, called the Bloch sphere. This sphere is a geometrical representation of the pure state space of a two level quantum mechanical system, a qubit.



Σχήμα 3.4: Bloch sphere

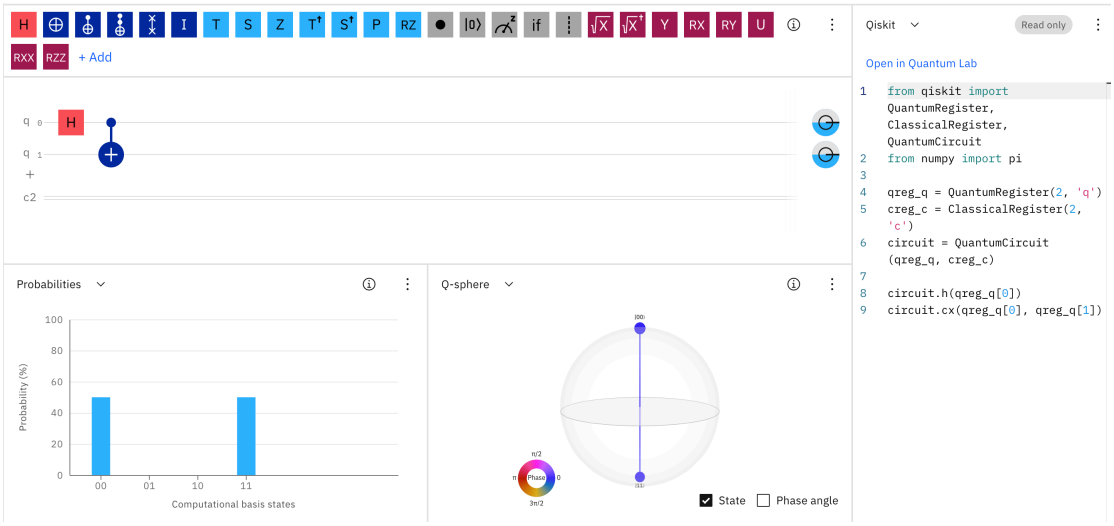
The two qubit states $|0>, |1>$ are represented by the z, -z axes. Any point $|\psi>$ on this sphere is represented by the equation $|q>$ I wrote above. The Bloch sphere is a generalization of a complex number z with $|z|^2 = 1$ as a point on the unit circle in the complex plane. Thus, we understand that the more generic form that we can write the equation of a qubit is the below:

$$|\psi>= cos(\frac{\theta}{2})|0> + e^{i\phi} sin(\frac{\theta}{2})|1>$$

where $\theta, \phi \in [0, \pi]$.

In order to handle a qubit or multiple qubits, we use basic gates. such as in classical computing, which include the X-Gate, Y-Gate, CNOT Gate, NAND Gate, XOR Gate and the OR Gate, the Hadamard gate and some more complex operations. The Pauli X-Gate (NOT gate) inverts the value of a qubit from 0 to 1 or 1 to 0. This inversion gate is often referred as a bit-flip. The Pauli Y-Gate is equivalent to a bit value and phase flip in one operation. So, it flips the value $+1$ to -1 or vice versa. The CNOT gate operates as the classical AND gate and as one can imagine this gate doesn't operate to a single qubit but to a system of 2 or more. It is called CNOT, because it is a Controlled NOT gate. If either of the input qubits are 0, the resulting qubit will be 0. There is also the Toffoli gate (or else the CCNOT gate) for the 3 qubit circuits. It simply takes two input qubits and flips the value of a resulting qubit if the two inputs hold a value of 1. If we want to reach superposition, then we have to use the Hadamard gate. The Hadamard gate converts a 0 state to $\frac{1}{\sqrt{2}}(|0> + |1>)$ and a 1 state to $\frac{1}{\sqrt{2}}(|0> - |1>)$. In both cases, the resulting state displays an equal probability of each eventuality. The NAND Gate operates as a NOT AND gate and the exclusive OR (XOR) gate can be created with a CX-gate. This takes an input and an output qubit. The output qubit will be inverted only if the input qubit has a value of 1.

All the above can be used to compose a quantum circuit and a simple way is to use the IBM Quantum Composer for visualizing and creating these circuits. It is a drag and drop web-application that helps you incorporate quantum circuit controls and execute them in a simulator or on a real quantum computer at IBM.



Σχήμα 3.5: IBM Quantum Composer

As displayed in 3.5, firstly we have many gates and operations at our disposal and many of them are included in the gates previously discussed. Below the set of tools, one can find the drag-and-drop palette, where you can add the number of qubits you want for your circuit, as long as the classical register and after that perform operations on the qubits. By default, all qubits are initialized to the $|0>$ state. Here, I have used an example to construct one of the four Bell states, using a Hadamard and a CX-gate. At the bottom of the page, we can find the predicting probabilities, which in this case are 50% to end up in the state $|0>$ and 50% to end up to $|11>$. Next to this component we can find the Q-sphere. The Q-sphere provides a global view of multi-qubit quantum state in the computational basis. The size of the node is proportional to express the probability of the state and the color reflects the phase of each basis state. Finally, in the right side of the screen, we find the Quantum Lab section, where the IBM Quantum Composer converts our implemented quantum circuit to python code using the Qiskit library. This is very useful as in many projects and especially in Quantum Machine Learning, we use hybrid classical-quantum algorithms and is most desired to alternate between tasks using the same tool.

Now that we have explained some basic gates and the basic functions of the tool, it is a good time to highlight how to read the IBM Quantum Composer probability bar chart to understand the output from a quantum circuit. In this chart you can see which outcome that the qubit would result in after applying the series of quantum gates. As dictated by quantum mechanics laws, in order to know this result we have to measure our circuit. But when we continue with the measurement, the wave

function of the system collapses. What do we mean by that? All postulates in quantum mechanics are applicable on closed, isolated systems. The two basic postulates are the following:

- For an isolated quantum system, we can define a Hilbert space, aka a complex vector space with an inner product. This allows several operations like measurement of length and angles.
- The evolution of a closed system can be described with a unitary transformation U, such that $|\psi' >= U|\psi >$.

The main conclusion of the above is that closed systems described by unitary time evolution (Hamiltonian formalism) can be measured by projective measurements.

When a system contains more than one qubit and especially when the states are not well defined states of 0 and 1, things get more complicated. First of all, the wave function of the system is written in the form:

$$|\psi >= a_{00}|00 > +a_{01}|01 > +a_{10}|10 > +a_{11}|11 >$$

with $\sum_{ij} |a_{ij}|^2 = 1$.
The problem with this state is that although we have the four complex numbers, most of the system's information cannot be accessed by measurement. If we measure the entire system or perform a partial measurement of some qubits, we obtain a new state. The new superposition is obtained by crossing out all those terms of $|\psi >$ that are inconsistent with the outcome of the measurement. After normalization to obey the probability laws and the preservation laws, we get:

$$|\psi_{new} >= \frac{a_{00}|00 > +a_{01}|01 >}{\sqrt{|a_{00}|^2 + |a_{01}|^2}}$$

The big question after all is can we decompose our entire state to partial measurements of two qubits? Classically there would not be a problem with doing so and consider that each two qubits should be in a state of the form $\alpha|0 > +\beta|1 >$. However, we have seen in practice some states, such the Bell states, which are:

$$|\Phi^+ >= \frac{1}{\sqrt{2}}(|00 > +|11 >) \tag{3.3}$$

$$|\Phi^- >= \frac{1}{\sqrt{2}}(|00 > -|11 >) \tag{3.4}$$

$$|\Psi^+ >= \frac{1}{\sqrt{2}}(|01 > +|10 >) \tag{3.5}$$

$$|\Psi^- >= \frac{1}{\sqrt{2}}(|01 > -|10 >) \tag{3.6}$$

that cannot decomposed in two separate states of each qubit. This phenomenon is called entanglement. When two qubits are entangled, we cannot determine the state of each qubit separately because the state of each qubit has as much to do with the relationship of the two qubits as it does with their individual states. The above Bell states are considered as the states in which two qubits are maximally entangled.

If we measure the first state, then the outcome is 50% 0 and 50% 1. However if the outcome is 0, than a measurement of the second qubit will result in 0 for 100% probability no matter the spatial distance between the two qubits. In the same way as in a GNN, we take into account both the state of the node and the interaction it has with its neighbors, here the qubits that are in an entangled state, they combine. This is also the secret of Quantum Teleportation.

8. Data Encoding in QML

The algorithms employed in Quantum Machine Learning are hybrid classical-quantum algorithms, as discussed in earlier chapters. We must be able to adapt our input data to be usable by the various sorts of tools and libraries that will be used during the implementation due to the nature of these methods. For quantum computation, classical data encoding is crucial to the overall design and efficiency of the algorithm. By encoding, we mean loading classical data into the qubits' state. There are various options for doing so. Generally, in the qubit encoding process given a feature vector $\vec{x} = [x_1, \ldots, x_N]^T \in R^N$, the general qubit encoding maps $\vec{x} \to E(\vec{x})$ given by

$$|x >= \otimes_{i=1}^{[N/2]} f_i(x_{2i-1}, x_{2i})|0 > +g_i(x_{2i-1}, x_{2i})|1 > \tag{3.7}$$

where $f, g : R \times R \to C$ are such that $|f_i|^2 + |g_i|^2 = 1$ for every i.

Encoding can be categorized into two major categories:

- Digital encoding is the representation of data as qubit strings. This category is most preferable if data has to be processed by arithmetic computations.
- Analogue encoding represents data in the amplitudes of a state. This category is most preferable for machine learning algorithms, as mapping data into the large Hilbert space of the quantum device is needed.

To begin, one of the most common methods is basis encoding. When real numbers must be arithmetically manipulated in a quantum algorithm, this instance is more

applicable. sThe ideal goal is to represent real numbers in the binary system and then convert them to a quantum state on a computational basis. This method, however, is prohibitively expensive because to the large amount of qubits required. The Amplitude encoding method is another option for encoding our data. The data is encoded into the amplitudes of a quantum state, as the method's name suggests. When we need to take advantage of a quantum device's huge Hilbert space, this way is better. This encoding requires $log_2(n)$ qubits to represent an n-dimensional data point. Finally, one should include Angle encoding, or else Tensor product encoding, as one of the most well-known techniques. The n classical features are encoded into the rotation angle of the n qubit. It requires one rotation on each qubit. This method is mostly useful, when processing data in quantum neural networks. Before our data is a compatible input to our neural network, we apply a rotation gate over the x or y axis, RX or RY, respectively. Mathematically, these transformations can be written as the below matrices:

$$R_x(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix} \tag{3.8}$$

$$R_y(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ -sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix} \tag{3.9}$$

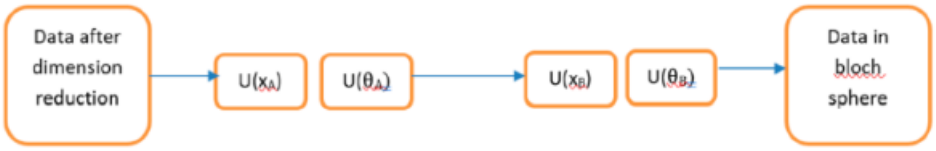
As it has been experimentally observed, this method is very efficient in operations. This occurs because, regardless of the quantity of data values to be encoded, only a fixed number of parallel processes are required. However, because each input vector component takes one qubit, this is also expensive in terms of qubits.
In addition to the above basic and most generally used techniques, other more specialized or combined techniques are being used. Firstly, there is the Qsample encoding method, which combines the basis and amplitude encoding methods. It associates a real amplitude vector with classical discrete probability distributions. The advantage is that we use mostly the amplitude logic, but all features are encoded in the qubit. Thus, we can assume that most of the above choices are suitable to be used to specific cases and the final choice for the encoding technique is very important as it influences the runtime of the loading process.

In an example case, researchers have tried a descriptor compression algorithm for Quantum Machine Learning of Sars-Cov-2 Data. They employed an MF to produce binary numbers with a default of 2048 bits in this algorithm. A large number of descriptors, as well as the use of qubits on the Quantum Computing network to represent them, can be a challenge. In most circumstances, decoherence noise introduced in the qubit system causes accuracy to fail. If there is a lack of linkage between some qubits, some additional noise may be injected to the system based on the quantum computer's architecture. The 2048 descriptor features were encoded using the following four methods:

- PCA,
- Common dimension reduction technique of linear discriminant analysis (LDA)
- Divide 2048 molecule fingerprint bits into ”x” groups, such that, each group has ”k” bits. The number of bits should divide 2048 completely. Then the ”k” bits are converted into base 10 or decimal value. Repeat until all the groups are converted to a decimal.
- Keep track of positions of 1 in the whole array.

Within the quantum computer, a quantum algorithm was used to solve the direct product for matrix operations and then calculate the M matrix. Then a quantum algorithm can be used to transform into waveforms to solve the complete Support Vector Machine on a quantum computer. While the SVM generally provides promising results, it takes considerable time to solve linear equations to solve for the kernel matrix using feature maps.

Finally, in the diagram below, one can see how the dimensionality reduction worked and the data were stored.



Σχήμα 3.6: Data in the Bloch sphere

Generally, using SVM models we reduce the 3-dimensional space to 2 dimensions. In this particular case, they managed to reduce to 3 to 10 dimensions to consider the qubit architecture. In the above diagram,you can find displayed a high-level abstraction of how data are stored in the Bloch sphere. The two layers, A and B with $U(x_A)$ denoting the unitary matrix applied on the input vector and $U(\theta_A)$ representing the unitary matrix applied to rotate the vector in the Bloch sphere.

As previously stated, dimensionality reduction is a critical task that may be accomplished using Neural Networks. To convert high-dimensional data to low-dimensional codes, a multi-layer neural network with a tiny central layer can be trained to reconstruct high-dimensional input vectors. Principal components analysis (PCA) is a simple and commonly used method in the classical approach, which discovers the directions of highest variance in the data set and represents each data

point by its coordinates along each of these directions. A multi-layer encoder network transforms high-dimensional data into a low-dimensional code, and a similar decoder network recovers the data from the code, is a generalized approach. For these networks, the required gradients are easily obtained by using the chain rule to backpropagate error derivatives first through the decoder network and then through the encoder network. After that, an ensemble of binary vectors can be modeled using a two-layer network, the restricted Boltzmann machine, in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections. To continue wit, the network assigns a probability to every possible image via the energy function. One possible joint configuration (v, h) of the visible and hidden units has an energy of:

$$E(v,h) = -\sum_{i \in pixels} b_i v_i - \sum_{j \in features} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{3.10}$$

where v_i, h_j are the binary states of pixel i and feature j, b_i, b_j are their biases and w_{ij} is the weight between them.

In the modeling that was used for this study, was a single layer of binary features, which was used for learning and the results were used as data for learning a second layer of features. This layer-by-layer learning process can be performed as many times as needed. Each feature layer captures high-order correlations between the actions of units in the layer below. The global fine-tuning stage then substitutes stochastic activity with deterministic, real-valued probability and employs back-propagation to fine-tune the weights for optimal reconstruction throughout the whole autoencoder. Classification and regression can both benefit from layer-by-layer pre-training. Pre-training was found to aid generalization by ensuring that the majority of the information in the weights originates from modeling the images or, in our instance, the graph structures.

9. Angle Encoding

Due to the nature of our problem, we will probably choose to use the Angle encoding method as the most suitable one to encode our data to our Quantum Graph Neural Network. Thus, we need to deep dive in this technique in order to understand how to implement it and to ensure that we will not have any data leakage at this step of the process. Angle encoding makes use of rotation gates to encode classical information x. The classical information determines angles of rotation gates, as shown below:

$$|x> = \otimes_i^n R(x_i)|0^n> \tag{3.11}$$

where R can be rotation over any axis, R_x, R_y, R_z. Usually, the number of qubits used for encoding is equal to the dimension of vector x. However, we should always keep in mind that each encoding is essentially a trade-off between three major aspects.

- the number of qubits should be minimal,
- the number of parallel operations should be minimal to minimize the width of the quantum circuit,
- the data must be represented appropriately for further calculations

Understanding this trade off, it may more suitable to some cases to try the Dense Angle Encoding technique. This subcategory exploits an additional property of qubits, the relative phase, to use the only $n/2$ qubits to encode n data points. This could be most useful in real life cases, while most quantum computers still have a limited number of qubits. In dense angle encoding, given a feature vector $\vec{x} = [x_1, \ldots, x_N]^T \in R^N$, the process maps $\vec{x} \to E(\vec{x})$ given by

$$|\vec{x}>= \otimes_{i=1}^{[N/2]} cos(\pi x_{2i-1})|0> + e^{2\pi i x 2i} sin(\pi x_{2i-1})|1> \tag{3.12}$$

And as mentioned above, we can refer to dense angle encoding as the process to encode two features per qubit by exploiting the relative phase degree of freedom, we can also try dense angle encoding for two-dimensional data $\vec{x} \in R^2$ with a single qubit given by

$$|\vec{x}>= cos(\pi x_1)|0> + e^{2\pi i x_2} sin(\pi x_1)|1> \tag{3.13}$$

with a density matrix

$$\rho_x = \begin{pmatrix} cos^2(\pi x_1) & e^{-2\pi i x 2} cos(\pi x_1) sin(\pi x_1) \\ e^{2\pi i x 2} cos(\pi x_1) sin(\pi x_1) & sin^2(\pi x_1) \end{pmatrix} \tag{3.14}$$

Lastly, the extreme version of this process is called Superdense Angle Encoding (SDAE). Let $\vec{x} = [x_1, \ldots, x_N]^T \in R^N$ be a feature vector and $\theta, \phi \in R^N$ be parameters. Then the superdense angle encoding maps $\vec{x} \to E(\vec{x})$ given by

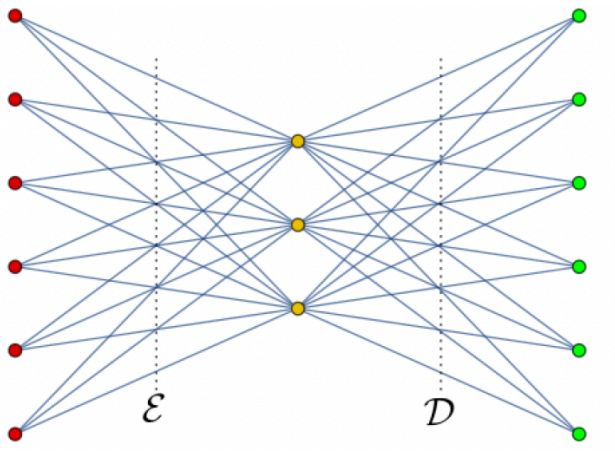
$$|x>= \otimes_{i=1}^{[N/2]} cos(\theta_i x_{2i-1} + \phi_i x_{2i})|0> + cos(\theta_i x_{2i-1} + \phi_i x_{2i})|1> \tag{3.15}$$

We see that this model includes two new hyper-parameters, θ and φ. This happens in order to optimize the process and increase robustness. Now as mentioned in

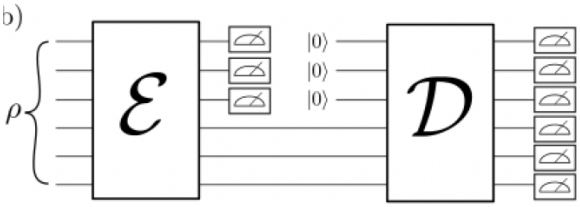
literature, dense angle encoding is seen to perform well on all data-sets and is capable of adapting well to noise, even outperforming the ideal case without noise and fixed encoding. From another point of view, superdense angle encoding does not perform well on any shown data-set since the generated decision boundary is highly non-linear and cannot correctly classify more than half the data-set.

10. Variational Auto-encoders

VAEs (Variational Autoencoders) are versatile generative models that may be used to a variety of media. A conventional autoencoder network can be described as a pair of two connected networks, the encoder and the decoder, if we wish to analyze it. The encoder network turns an input into a smaller, denser representation that the decoder network can utilize to convert it back to the original input. We examined Convolutional Graph Networks earlier in this chapter. As a result, we can conclude that the general operation of CNNs is also well understood. If we have an experiment with images as input, the CNN will take the large image and ”compress” it to a more compact and dense representation, which it will then use to categorize the image via a fully connected classifier network. The encoders operate in a similar manner. After that, the network produces a much smaller representation with enough information for the next component of the network to transform it into the required output format. Encoders, on the other hand, are trained in conjunction with other portions of the network and then optimized through backpropagation to provide encodings suitable for the task at hand. If we employ an autoencoder, the system will instruct the encoder to create encodings that are specifically beneficial for reconstructing its own input. As previously said, the complete network is usually trained together. The reconstruction loss is a penalty imposed on the network for producing outputs that differ from the input and is usually the mean-squared error or cross-entropy between the output and the input. Now that we’ve demonstrated that all encoders’ intuition is to decrease the input to a smaller representation of itself, the encoder must choose to delete information in this scenario. The encoder learns to preserve as much of the relevant information as possible in the limited encoding, and intelligently discard irrelevant parts. Next, the decode learns to take the encoding and properly reconstruct it into a full image. If we combine the above parts, we receive the system of an autoencoder.



Σχήμα 3.7: Variational Auto-Encoder



Σχήμα 3.8: VAE Circuit

In the first picture above, we present a graphical representation of six-bit autoencoder with a three-bit latent space. The map encodes a six-bit input, which are displayed with red points, into a three-bit intermediate state, which is symbolised with yellow points in the center, after which the decoder D attempts to reconstruct the input bits at the output, which is displayed as green points. At the second image, one can find the circuit implementation that corresponds to the above 6-3-6 quantum autoencoder. The issue with ordinary autoencoders, on the other hand, can be found in the generation stage. The issue is that the latent space into which the autoencoder translates its inputs and where its encoded vectors are stored may not be continuous or allow easy interpolation. This is a significant problem since we want to randomly sample from the latent space or produce variants on an input image from a continuous latent space while building generative models. If there are discontinuities in the space and we sample or generate a variation from there, the decoder will simply produce an implausible output because it has no concept how to deal with that part of the latent space.

At this point it is time to introduce Variational Autoencoders. This type of autoencoders has one unique property that separates them from the standard

autoencoders and that makes them most suitable for generative modeling. Their latent spaces are designed to be continuous, allowing easy random sampling and interpolation. This is achieved by forcing the encoder output a vector of means μ and a vector of standard deviations σ. We notice that the difference from the standard autoencoders is that the vectors produced are doubled but each vector has the same size as the one that the output vector from the standard encoder would have had. The generation is stochastic and by this design one can achieve that the encoding will vary one every single pass due to the sampling over the input. This sample encoding which is then passed to the decoder is achieved by exploiting the parameters of a vector that includes random variables of length the same of the initial vectors, using the μ, σ of the i-th random variable, from which the sampling happens. The mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the allowed variance from the mean. As encodings are randomly from anywhere inside the distribution, the decoder learns that not only is a single point in latent space referring to a sample of that class but all nearby points refer to the same as ell. This allows the decoder to not just decode single, specific encodings in the latent space but also small variances of itself, as the decoder is exposed to a range of variations of the encoding of the same input during training.

The ability to use interpolation between classes is a key characteristic of VAEs. Because the values of μ, σ have no limitations, the encoder can learn to create extremely different μ for distinct classes, clustering them apart and minimizing σ, ensuring that the encodings themselves do not differ significantly from the samples. This enables the decoder to reassemble the training data quickly. The Kullback-Leibler (KL) divergence must be introduced into the loss function to compel smooth interpolations and permit the creation of new samples. This metric is measured between two probability distributions and it simply measures how much they diverge from each other. If we want to optimize the probability distributions parameters, μ, σ to closely resemble that of the of the target distribution, we must try to minimize the KL divergence. For VAEs KL loss is equivalent to the sum of all the KL divergences between the component $X_i \approx N(\mu_i, \sigma_i^2)$ in X and the standard normal. It reaches its minimum for $\mu_i = 0$ and $\sigma_i = 1$. This measure is an excellent fit for autoencoders since it encourages the encoder to evenly distribute all encodings around the latent space's center. Pure KL loss causes encodings to be densely put randomly around the center of the latent space, with little respect for similarity between nearby encodings. It is difficult for the decoder to decipher anything significant from this region. As a result, this procedure need improvement. This produces a latent space that, on a local scale, retains the similarity of surrounding encodings by clustering that is tightly packed around the latent space origin. Following this strategy, we can eventually build separate clusters that the decoder can decode. The extra benefit is that when interpolating, there are no sudden gaps between clusters as we have a smooth mix of features a decoder can understand.

11. Quantum VAEs

One can utilize one of the approaches outlined above, or any basic quantum autoencoder that can incorporate some embedding procedures, to encode classical data to qubits. The particular options of quantum embedding and measurement are primarily determined by the dataset's dimensions and scales, and the optimal option is often a hybrid model, as pure quantum encoders only apply to normalized-scale data reconstruction. The reason for this is that the top bound of both expectation and probability is 1.

One can start with BQ-BVAE/AE which adopts amplitude embedding and expectation output for encoder and angle embedding and probability output for decoder. This autoencoder has already been used on QM9 data-set. Two variations of the same model were used, the first was fully baseline quantum VAEs (F-BQ-VAEs) and the second one was a hybrid quantum variant (H-BQ-VAEs). The main comparison of their performance was experimentally proven as shown:

TABLE I
COMPARISON OF NUMBER OF TRAINABLE PARAMETERS.

Parameter Type	VAE(AE)	F-BQ-VAE(AE)	H-BQ-VAE(AE)
Quantum	0 (0)	108 (108)	108 (108)
Classical	5694 (5610)	84 (0)	4286 (4202)
Total	5694 (5610)	192 (108)	4394 (4310)

Σχήμα 3.9: Autoencoders

It was shown that BQ-VAE/AE learned faster for normalized QM9 molecules as a function of the training epochs.
The other case that was studied is the category of scalable quantum autoencoders.

In this sub-chapter, we aim to focus in the quantum analogous of the previous chapter. The logic behind a quantum autoencoder is that such systems allow us to perform analogous machine learning tasks for quantum systems without exponentially costly classical memory, such as in the reduction of the data dimensionality. The model is not far from the classical reality, as it has been proven that for a specific model, if we choose a specific setting of parameters in the quantum network, it reduces to classical neural network exactly. Each node in a simple quantum auto-encoder system represents a qubit, with the output register represented by the first layer of the network. A unitary transformation from one layer to the next is represented by the edges linking adjacent layers. Finding unitaries that retain the quantum information

of the input through the smaller intermediate latent space is the main learning task for a quantum encoder. A successful encoding can be described as one in which $F(|\psi_i>, \rho_i^{out}) \approx 1$ for all input states, where F is the fidelity of the states and can be defined mathematically as

$$F(|\psi_i>, \rho_i^{out}) =< \psi_i ? \rho_i^{out} |\psi_i > \tag{3.16}$$

Fidelity is a measure of how much two states look-alike and is expressed as the probability that one state will pass a test to identify as the other. To return back to the topic of this chapter, as defined in recent studies, if we have an ensemble $\{p_i.|\psi_i>_{AB}\}$ of pure states on $n+k$ qubits, where subsystems A and B are comprised of n and k qubits, respectively, then we can consider a family of unitary operators $\{U^{\vec{p}}\}$. These operators act on $n+k$ qubits, where $\vec{p} = \{p_1, p_2, \dots\}$ is some set of parameters defining a unitary quantum circuit. Also let $|a>_{B'}$ be some fixed pure reference state of k qubits. Using classical machine learning methods, we wish to find the unitary $U^{\vec{p}}$, which maximizes the average fidelity, which we define to be the cost function

$$C_1(\vec{p}) = \sum_i p_i \cdot F(|\psi_i>, \rho_{i,\vec{p}}^{out}) \tag{3.17}$$

with,

$$\rho_{i,\vec{p}}^{out} = (U^{\vec{p}})_{AB}^{+} Tr_B[U_{AB}^{\vec{p}}[\psi_{i_{AB}} \otimes a_{B'}](U^{\vec{p}_{AB}})^{+}](U^{\vec{p}})_{AB'} \tag{3.18}$$

where we have used $|\psi_i>< \psi_i|_{AB} = \psi_{i_{AB}}$ and $|a>< a|_{B'} = a_{B'}$. The main goal is to find the best unitary $U^{\vec{p}}$, which on average best preserves the input state of the circuit. After some hypotheses and calculations there is also an alternative definition of the cost function in terms of the trash state fidelity,

$$C_2(\vec{p}) = \sum_i p_i \cdot F(Tr_A[U^{\vec{p}}|\psi_i>< \psi_i|_{AB}(U^{\vec{p}})^{+}], |a>_B) \tag{3.19}$$

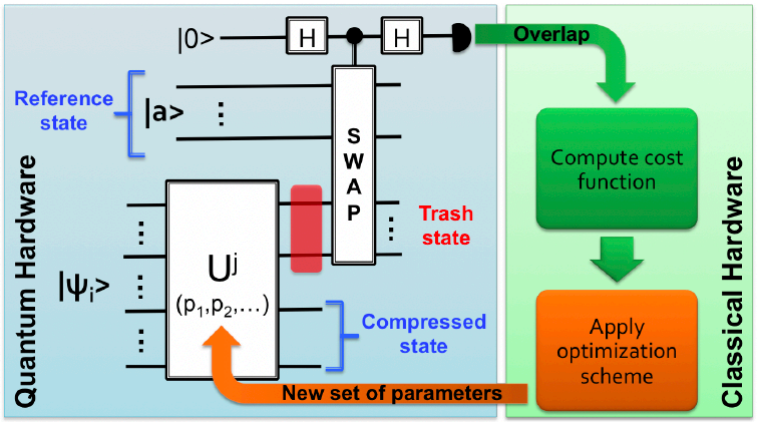
This equation differs from the previous one, as it is considered that $C_1 \leq C_2$. As trash state we consider the state with density matrix $\rho'_{B'} = Tr_A[|\psi'_i>< \psi'_i|_{AB'}]$ and it is preferable to look at this state instead of tracing over the AB system.

For the implementation of the quantum autoencoder to be efficient, the number of parameters and the number of gates in the circuit should scale polynomially with the number of input qubits. The other choice for generating the $U^{\vec{p}}$ is to employ a programmable quantum circuit that consists of a fixed network of gates, where a polynomial number of parameters associated to the gates constitute $\vec{p}$. The next step is to train the network bu maximizing the autoencoder cost function, C_2. The schematic representation of the training process can be depicted in 3.10. It consists of the following steps:

1. Efficiently prepare the input state, $|\psi_i>$, and the reference state.
2. Evolve under the encoding unitary, $U^{\vec{p}}$, where $\vec{p}$ is the set of parameters at a given optimization step.
3. Measure the fidelity between the trash state and the reference state via a SWAP test.

The training approach uses a quantum-classical hybrid scheme, in which the quantum computer handles state preparation and measurement, while a classical computer handles optimization. The weighted average of fidelities between the trash state produced by compression and the reference state is defined as the cost function of the quantum autoencoder. The cost function is bounded by 1 and its optimization means to minimize the value of $log_{10}(1 - C_2)$. After the computation of C_2, we can feed it into a classical optimization routine that returns a new set of parameters for our compression circuit. All these steps are repeated until the optimization algorithm converges. Finally, as described in this sub-section the training could be implemented with states of a specific size, obtained by a given ansatz and then it can be used as a state preparation tool. Once the system has been trained to compress a specific set of states, the decompression unitary (U^+) can be used to generate states similar to those originally used for training. This can be achieved by preparing a state of the form $\Psi_I > \otimes|a >$ and evolving it under U^+, where $|Psi_I >$ has the size of the latent space and $|a >$ is the reference state used for training. However, one important consideration when using quantum autoencoders of this type is that the von-Neumann entropy of the density operator representing the ensemble $\{p_i, |\psi_i >_{AB}$ $\}$ limits the number of qubits to which it can be noiselessly compressed.

As shown in the image 3.9, autoencoders shrink the space between the first and the second layer. One important thing to keep in mind when working with an autoencoder network, the input nodes must be discarded after the initial encoding and then we initialise a new set of qubits based on a reference state, which will be used to implement the final decoding D that will be compared to the initial state.



Σχήμα 3.10: Training QVAEs

12. Quantum Graph Neural Networks

Quantum Graph Neural Networks is one emerging category of QML algorithms that is particularly suitable to be executed on distributed quantum systems over a quantum network. Now, if we want to understand how a hybrid quantum-classical model works, we may use it as an example to shift from the logic of a classical graph network to quantum computing. This GNN's goal is to forecast the probability that each edge in an event graph is a track segment. This is in the hopes of feeding new event graphs into the trained QGNN, which can then infer edge predictions that can be used to anticipate drug assembly over tiny molecules. An edge and node network, which share weights across layers and within the same layer, make up the specific model. Because they are applied repeatedly, edge and node networks exchange weights across layers. This calls attention to the fact that just one node network and one edge network are utilized, and they are applied repeatedly, therefore all weights of those networks are shared across all nodes and layers in each application.

Starting with the input network, we describe our input as a matrix of all the data points per node, which is then cast into a higher-dimensional embedding space using a single-layer neural network. At this point, we use N_D the number of hidden dimensions as a hyperparameter that dictates the size of all node embeddings and we define

$$InputNet(X) = \sigma(W^{(i)}X) \tag{3.20}$$

where X is the input feature matrix, $W^(i)$ the weight matrix for the fully connected layer and σ the activation function. Now, the shape of the input is defined from the number of nodes N_v and the number of coordinates, 3 and thus $dim(input) =$

$(N_v, 3)$. When the spatial coordinates of each data point are supplied, the InputNetwork converts them to an embedding space with N_D dimensions. The node embeddings are then fed through the edge network and then the node network for N_I iterations, concatenated with their perspective spatial coordinates. With each pass-through of the constituent networks, the node's latent variables (embeddings in N_D) and edge weights are alternately updated.

In contrast, the output shape is given by $dim(output) = (N_v, N_D)$ and the output represents the node embeddings for each data point.

In the next section we need to analyse is the Edge Network. This part of the model aims to predict the probability that a specific edge exists. The input of this network is a pair of node feature vectors. Here we have to note that each node feature vector is the concatenation of the 3 fixed spatial coordinates. The output is just a float representing the probability that the pair of nodes are connected and we get N_D trainable embedding values from this stage. If h is the feature vector at the input or output at the k-th layer and e_k is the probability at the k-th layer, we can define the below equation to mathematically model the purpose of the edge network.

$$e_k = EdgeNetwrok(h_o^{(k)} + h_i^{(k)}) \tag{3.21}$$

The Edge Network is a hybrid quantum-classical Neural Network and after taking a concatenated vector containing a pair of nodes, the data is fed through a quantum neural network sandwiched in between two trainable classical layers. For the input classification layer we know that the aim is to best condense the dimension form $2(3 + N_D)$ to N_Q, which is the number of qubits. Then, the N_Q outputs are rescaled to $[0, \pi]$, which then parametrizes the information encoding circuit (IEC). Unfortunately, the encoding scheme, independent to what it is, angle or amplitude encoding or some combination of encoding methods, it is not trainable. Continuing after the encoding step, the quantum state is evolved under the trainable parametrized circuit (PQC). Then all qubits are measured from the PQC to then be fed into a classical fully connected layer, which outputs a single edge probability value using an activation function.

Each node feature vector has an encapsulated embedding that has been trained to condense the most useful information for making this decision. As we know from the basic instance of graph convolutional NNs, each GNN layer, in this case the node network, is trained to provide the optimal embedding so that the two node embeddings may be used together to determine whether or not there is an edge between them. It's also worth noting that sandwiching the quantum neural network between two classical layers gives us more freedom with the aggregated network's input and output shapes. So having an input classical layer enables us to freely play

around with the hidden dimension size, while also retaining independent freedom over the number of qubits.
In terms of the node network, all edge network predictions are crucial for iteratively updating the node embeddings, which condense hidden graph properties. The quantum GNN's edge network is responsible for predicting track segments as a sub-component. As a result, it is critical for the node network. Furthermore, it has been demonstrated experimentally that information propagates to more nodes with more iterations. We capture broader graph attributes in this way, allowing it to update a node's local features with more understanding.

The input of the node network is a concatenated triplet of 3 node feature vectors:

- One weighted sum over all neighboring node features residing in the detector layer before the h_j layer,
- One $h'_{j,output}$ weighted sum over all neighboring node features that lie in a detector layer directly ahead of the h_j layer,
- the target node feature vector.

The node network's architecture consists of an input layer of $3(3 + N_D)$ dimension and an output layer of N_D dimension and is responsible for containing the node embedding. The last part of the algorithm that needs to be checked, is the ansatz that we will use when training the QGNN.

Thus, if we want to summarize the edge network serves as an attention mode that feeds information to the node network on which surrounding nodes are important to attend to. This helps the edge network identify the existence of a track segment since there's a higher correlation between two nodes that are likelier to form an edge.

When training the NN, we need to calculate the quantum gradients independently using inefficient analytic parameter-shift rules. In the training process the below steps are being executed:

1. Firstly, we feed-in a batch of individual graphs and conduct a forward pass to arrive at the predicted edge structure for each individual graph.
2. Then we calculate the binary cross-entropy loss between the predicted edge values and the labels.

3. After that we backpropagate the loss to find the gradients of each classical and quantum weight, by following the parameter-shift rule. The classical gradients are computed as usual leveraging auto-differentiation.

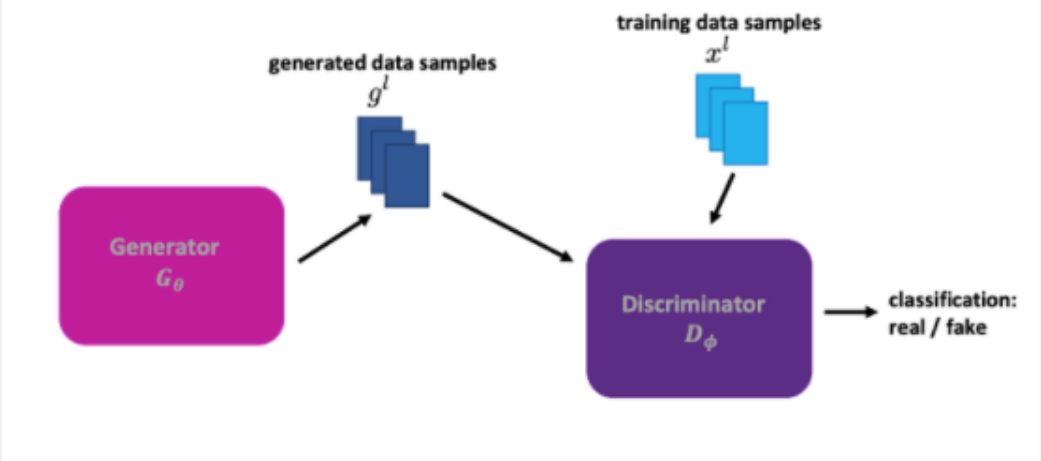
4. As the last step, we update each weight using calculated gradients with the optimizer of our choice.

5. Then we repeat the steps above for all batches in an epoch, over 10-100 epochs.

In the above steps, we mentioned cross-entropy as a loss function. When training the model we can feed the embeddings to any loss function and run stochastic gradient descent to train the weight parameters as dictated from the rest of the steps. The loss function should be chosen based on the node proximity in the graph and we should train the model for a supervised task, such as node classification.

13. Quantum GANs

The next level is for someone to try the quantum analogous of this model. QGANs obey the same rules as its classical analogous but it has many differences, starting from the architecture:



Σχήμα 3.11: QGANs

QGANs is a hybrid quantum-classical model, where at least one part of it is based on quantum algorithms, either being the discriminator or the generator.

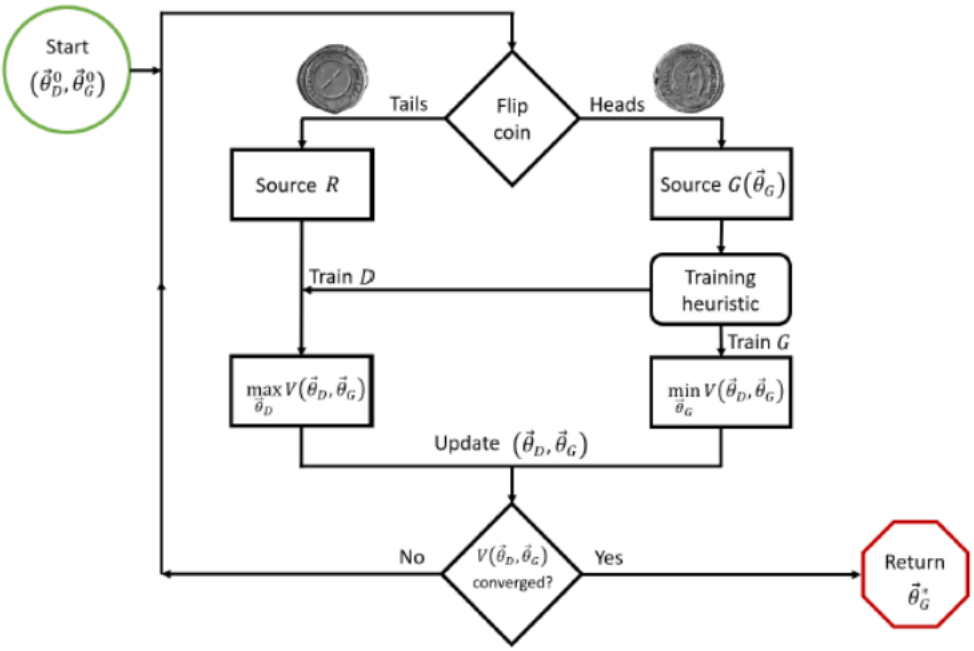
In this model, the algorithm uses the interplay of the Generator model and the Discriminator model to learn the probability distribution underlying given training data. The goal of this kind of network is for the quantum generator to learn the training data's underlying probability distribution and it loads a quantum state which is a model of the target distribution.

As one can conclude from the above statement, QGANs are very useful because they can load in polynomial time random probability distributions into quantum data states. The algorithm trains the quantum generator to create a quantum state, which represents the data's underlying probability distribution, and it achieves the so calling quantum advantage in combination with other algorithms such as Quantum Amplitude Estimation (QAE). From another perspective, this algorithm can be used for classical data for efficient QIP (Quantum Information Processing) and have applications in finance and banking. Of course, all the above matter because the Quantum algorithms have the potential to outperform their classical analogues and it is a more efficient method due to the computational cost and the time-cost of loading data into quantum states ($O(2^n)$).
In order to implement all the mentioned processes, in QGANs and in Quantum ML one should use Variational Quantum Algorithms. They are the key to achieve the desired quantum advantage. In general, VQAs have been developed for a great variety of applications. VQAs use tasks encoded in a cost function, which is evaluated by a quantum computer, and use a classical optimizer to train a parameterized quantum circuit and with their structure they have managed to emerge as a leading strategy to address the constraints of the current quantum devices. Some of these very binding constraints are the limited number of Qubits that are used and the noise processes that limit circuit depth. Thus, their adaptive nature help overcome those problems. To implement these quantum algorithms we choose from a variety of tools created by IBM to implement and visualize quantum circuits via the IBM-Quantum Experience and finally to measure and output our results.

When implementing QGANs, one can choose from a variety of generator-discriminator pairs depending on the execution environment either on quantum computers or simulations (a more detailed analysis on this is displayed in the "Experimental Method" chapter). In these models, real data has to engage the state preparation stage, usually through amplitude encoding, for encoding classical data in a quantum state, and this stage takes $Nlog(M)$ qubits where N is the training set size and M is feature dimension. In our case, were we study the QM9 data-set, more than 90 qubits are needed to discover real-like molecules. This total number of required qubits to reconstruct synthetic molecules is $\overset{9}{(2)}log5 + 9log5 > 90$, where 5 is the number of bond types and atom types contained in QM9. Thus, given the task complexity of learning molecule distribution, full quantum GAN can hardly encode all training data in a quantum way. This fact makes a full quantum model not feasible

in the near future and in researchers have proposed hybrid models to overcome this difficulty. These models include a hybrid generator and all variations exploit the strong expressive power of variational quantum circuits with exponential speedup up to $0(ploy(logM))$ time. Finally, in most models used, the discriminator imitates the MolGAN model and a reward networks are discarded since they observed that reward value is too minuscule to noticeably contribute to training the model. A single optimizer is used to update all quantum gate parameters and weights in neural network simultaneously and the discriminator is being updated alternatively.

The generalized process followed by a QGAN, which is described below, it can be summarized in the following diagram.



Σχήμα 3.12: QGAN flow

In general, the two quantum circuits (generator and discriminator) maximize and minimize the same optimization problem, as in the classical analogy. The generator learns the best parameters to send into its quantum gates after many training steps, and it separates out a quantum state that is relatively similar to the quantum state representing real data. To begin, one obtains expectation values from one discriminator instance. The expectation values are determined by the data supplied into the discriminator, and if both actual and created data were used, two expectation values were obtained, indicating the discriminator's performance on both types of data. Following this phase, the model can use the traditional method of a cost function to

optimize its parameters. The minimization of the cost function entails the maximization of the probability of correctly classifying fake data. In classical GANs, we use two cost functions, one for the discriminator and one for the generator but in a QGAN we can formalize them into one adversarial optimization problem, given by the expression:

$$min_{\vec{\theta}_G} max_{\vec{\theta}_D} (Pr(D(\vec{\theta}_D, R) = |real >) - Pr(D(\vec{\theta}_D, G(\vec{\theta}_G, z)) = |fake >) \tag{3.22}$$

where θ_D is the vector of parameters we plug into our discriminator ansatz, θ_G is the vector of parameters we plug into our generator ansatz. It is important to note that both parts depend on θ_D, but only the second part depends on θ_G. Mathematically, it is needed to condition the QGAN such that $|real >$ and $|fake >$ are orthogonal to retain the distinguishability of the two for the two for training Discriminator, resulting in a better Generator. We arbitrarily use as fact that $|real >= |0 >$ and $|fake >= |1 >$ with $+1, -1$ eigenvalues. For the discriminator we use a unitary operation of the form:

$$U_D = D(\vec{\theta}_D) \otimes I^{\otimes m} \tag{3.23}$$

with m: number of qubits in a register. The respective operators for the generator and source R are:

$$U_G = I^{\otimes(1+d)} \otimes G(\vec{\theta}_G, z) \tag{3.24}$$

$$U_R = I^{\otimes(1+d)} \otimes R \tag{3.25}$$

Now we can define the state when U_D is applied after U_G:

$$\rho^{DG}(\vec{\theta}_D, \vec{\theta}_G, z) = U_D(\vec{\theta}_D)\rho^G(\vec{\theta}_G, z)U_D^+(\vec{\theta}_D) \tag{3.26}$$

and the quantum state when U_D is applied after U_R:

$$\rho^{DR}(\vec{\theta}_D) = U_D(\vec{\theta}_D)\rho^R U_D^+(\vec{\theta}_D) \tag{3.27}$$

where $\rho^G(\vec{\theta}_G, z) = U_G(\vec{\theta}_G)\rho^0(z)U_G^+(\vec{\theta}_G)$. Thus, the cost function becomes:

$$V(\vec{\theta}_G, \vec{\theta}_D) = \frac{1}{2}(cos^2(\phi Tr(Z\rho^{DR}(\vec{\theta}_D)) - sin^2(\phi)Tr(Z\rho^{DG}(\vec{\theta}_D, \vec{\theta}_G, z))) \tag{3.28}$$

where ϕ is the bias of the source and Z is an observable. Note, that the expectation value of a density matrix with respect to an observable Z is the trace of that Z operator applied to the density matrix. Finally, the update rule used for the parameters $D(\theta_D^k)$ or $G(\theta_G^k)$, with learning rates X_D^k and X_G^k:

$$\vec{\theta}_D^{k+1} = \vec{\theta}_D^k + x_D^k \nabla_{\vec{\theta}_D} V(\vec{\theta}_D^k, \vec{\theta}_G^k) \tag{3.29}$$

$$\vec{\theta}_G^{k+1} = \vec{\theta}_G^k - x_G^k \nabla_{\vec{\theta}_G} V(\vec{\theta}_D^k, \vec{\theta}_G^k) \tag{3.30}$$

where k is a step.

If we consider that we use cross-entropy as loss function between the real and generated data $S(\rho^R||\rho^G) = Tr(\rho^R(log_2\rho^R - log_2\rho^G))$ and it converges at 0.

Κεφάλαιο 4

ML Models of Classical computing - Experimental Implementation

In this chapter, we will display and comment on the experimental process focused on the classical Machine Learning models. I tried to approach a classification problem. The starting idea was to search for similarities of molecules that are used in radiopharmateuticals compared to the molecules provided by the QM9 dataset. However, the similarity check that I implemented via the rdkit library, showed pretty low scores and decided to move on from this problem. For this similarity check I used radioactive isotopes including ^{18}F, ^{11}C and ^{3}H. The metric used was the Tanitomo similarity and the results received were of 10^{-2} order reaching a maximum at 0.3-0.4. This metric is considered as an appropriate choice for fingerprint-based similarity check. In this algorithm, we consider two sets A and B of fingerprint bits. AB is the set of common bits of fingerprints of both molecule A and B. The Tanitomo coefficient is given by the below equation:

$$T(A,B) = \frac{A^B}{A + B - A^B} \tag{4.1}$$

and it ranges from 0 when fingerprints have no bits in common, to 1 when the fingerprints are identical. A good value that can be used to consider two molecules as ”similar” is over 0.85.

One problem that caught my attention is the increasing numbers of young people with mental problems. Thus, I decided to focus my drug discovery study to molecules that could be used as tricyclic antidepressants. These drugs are used to ease depression by affecting neurotransmitters used to communicate between brain cells. Like most antidepressants, cyclic antidepressants work by ultimately effecting changes in brain chemistry and communication in brain nerve cell circuitry known to regulate mood,

to help relieve depression. Cyclic antidepressants block the re-absorption of the neurotransmitters serotonin and norepinephrine, increasing the levels of these two neurotransmitters in the brain. The Food and Drug Administration(FDA) has approved some antidepressants that are called doxepin, protriptyline, amoxapine and amitriptyline. Also, sometimes cyclic antidepressants are used to treat conditions other than depression, such as obsessive-compulsive disorder, anxiety disorder or nerve-related pain.

Thus, the purpose of this study is to study this classification problem of molecules used for designing antidepressant drugs. To achieve this, we used in our arsenal different types of Neural Networks including convolutional, graph and generative adversarial networks. This strategy is aimed to give as spherical conclusions on the QM9 dataset and how if it can reliably be used for antidepressant drug designing. But prior to implementing all the mentioned machine learning models, we have to prepare our dataset in order to be ready to use it as an input to our models.

To begin with, I loaded this dataset and all the prerequisite packages to my environment. The environment used was Google Colab's Jupyter Notebooks in need of a cloud solution. QM9 dataset was loaded automatically by using the chainer chemistry library and its original format is zipped file where each molecule's information is stored in each "xyz" file. Chainer Chemistry automatically merges these information in one csv file internally. Next in order to extract QM9 dataset the GGNNPreprocessor was used. We instantiate the preprocessor and follow with "get-qm9" methods to extract all labels. After this action, 15 types of physical properties were extracted and the basic encoding type of the information needed is through SMILES strings along with the dataset itself. The GGNNPreprocessor class receives as arguments:

1. The maximum number of atoms for each molecule and if the number of atoms is more than this value, this data is simply ignored.
2. The ouput size, which specifies the size of array returned by the "get-input-features" method. If the number of atoms in the molecule is less than this value, the returned arrays is padded to have fixed size.

After the completion of the above step we observe that QM9 dataset is a class of NumpyTupleDataset, where the i-th dataset features can be accessed by "dataset[i]". Additionally, due to using the GGNNPreprocessor class, each dataset consists of the following features:

1. atom feature: atomic number of a specific molecule,

2. adjacency matrix feature: GGNNPreprocessor extracts adjacency matrix of each bonding type,
3. label feature: chemical properties (labels) of a specific molecule.

The adjacency matrices of each type of bond between atoms in a specific molecule:
Single bond:

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

Double bond:

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

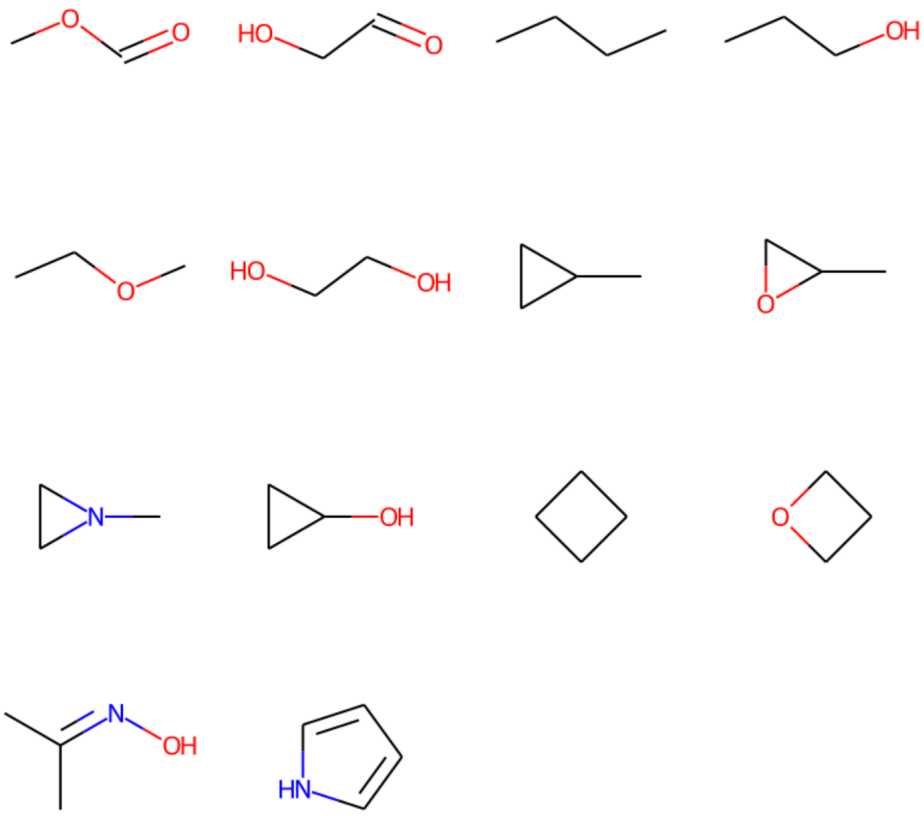
Triple bond:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Aromatic bond:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Next, I use some visualization modules of the RDKIT package to display some rows of our dataset to images, in order to understand it better4.1.

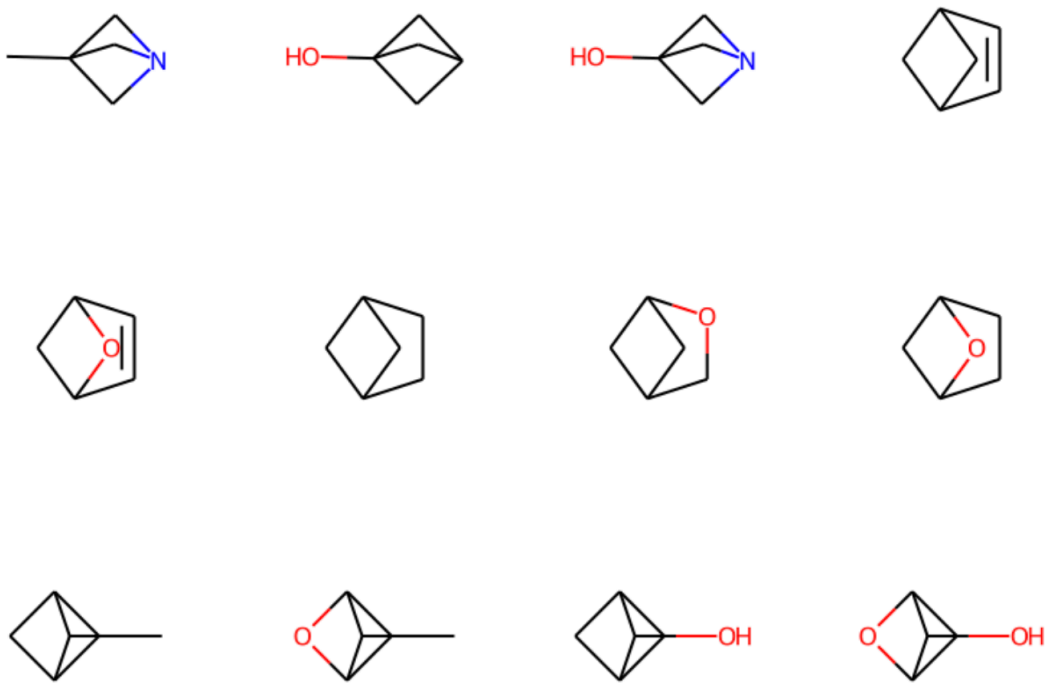


Σχήμα 4.1: Visualization of molecules in a grid

To continue with, I created an interactive visualization of the whole dataset for Jupyter Notebooks in order to be able to access each point in the dataset of 13 thousand rows. Finally, it is time to limit the dataset to our needs. Firstly, I contacted a similarity check on the molecules of Serotonin, Dopamine and Doxepin to find common characteristics that I will use as criteria to limit the dataset. As it is obvious by the name of the drugs, I filter the database to keep only molecules that have three ring substructures in their whole structure as molecules. With this criterion I limited my data to 19672 rows. I saved all molecule indexes to be able to isolate them from the whole dataset. Now that I have the rows I want to keep, the second criterion that I used to classify my dataset was the $logP$ coefficient. This is the octanol-water partition coefficient logP and is used in QSAR studies and rational drug design as a measure of molecular hydrophobicity. Hydrophobicity affects drug absorption, bioavailability, hydrophobic drug-receptor interactions, metabolism of molecules, as well as their toxicity. Other use of this coefficient is its key usage in studies of the environmental fate of chemicals. The value of known molecules used as ansatz in the present study can be seen in the below table [4]. However, as studying the chemistry and biological properties of all these drugs is not my

	SMILES	logP
Doxepin	CN(C)CCC=C1C2=CC=CC=C2COC3=CC=CC=C31	4.29
Serotonin	C1=CC2=C(C=C1O)C(=CN2)CCN	0.21
Dopamine	C1=CC(=C(C=C1CCN)O)O	-0.98
Amoxapine	C1CN(CCN1)C2=NC3=CC=CC=C3OC4=C2C=C(C=C4)Cl	3.4

area of expertise, in order to be more accurate we would need the assistance of an experienced researcher. All features and characteristics used to classify the data are based on personal observations, as we just want to test our models performances on this dataset and thus cannot be applied to real-life cases until provided official chemical criteria on these features by a professional in the specific area. Our final dataset include molecules of the below form:



Σχήμα 4.2: Molecule Visualization for final dataset

Depending on the feasibility of our models we use either a binary classification, either a multi-class classification splitting our data to 4 classes based on the values of logP coefficient. In most cases in the following chapters, we use data in the form of images/matrices combined with the logP feature for our classification problem and in few case we exploit other structural features of the model to show how we can optimize their encoding to classifiers. Finally, we aim to conclude on which level we can exploit the quantum supremacy at the moment, given the fact that quantum computers are in a preliminary stage and are constantly improving.

1. Convolutional Neural Network (CNN)

In this specific case study, I decided to focus on a multi-class classification analysis of the dataset. I decided to use as input the matrices which we have gained from the molecule images. The purpose of practically using images in this kind of network is because CNNs are immune to spatial variance and hence are able to detect features anywhere in the input images. Also CNNs are built in a way that can generate excellent predictions with minimal image preprocessing. For this case I split my data in 4 classes based on logP coefficient and split my data to 80% training dataset and 20% testing dataset.

I begun by choosing a simple MLP model with a few layers, which is displayed below [4.3].

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 64, 64, 4)]       0

 dense_9 (Dense)             (None, 64, 64, 5)         25

=================================================================
Total params: 25
Trainable params: 25
Non-trainable params: 0
_________________________________________________________________
```

Σχήμα 4.3: One Layer Initial Model

The results were pretty disappointing as one can observe [1.].

loss	accuracy	precision	val_loss	val_accuracy	val_precision
2.6527	0.5113	0.0306	25.0061	0.0490	0.0032
1.5340	0.6363	0.0704	29.3843	0.0490	0.0031
1.2716	0.6908	0.0924	27.2253	0.0490	0.0032
1.1985	0.6908	0.0988	28.5666	0.0490	0.0033

We had overfitting from the beginning as contradicting the good behavior of the training loss and accuracy, loss on the validation data kept increasing and the training stopped after only 4 epochs, due to it remained constant and was stopped by the Early Stopping. Also, both training and validation precision were kept under 1%. Thus, I made a few choices to change the architecture of the model and one can find the explanation in the final model.

The final choice for the architecture used for this model can be seen in [4.4]. One can see that we have used a Sequential model with its arguments including an Adam optimizer, 4 neurons and variable dropout. The hidden layers of the model include 3 two-dimensional convolutional filter of kernel size 3×3 to standardize the inputs to each layer for each mini-batch, combined with a ReLU activation function. As time is always a limitation, we decided to apply several batch normalization steps in order to reduce the number of epochs needed to train the network and to overcome the over-fitting phenomenon that appeared in previous models that I tried.

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, None, None, 100)   2800

 batch_normalization_6 (Batc  (None, None, None, 100)  400
 hNormalization)

 max_pooling2d_6 (MaxPooling  (None, None, None, 100)  0
 2D)

 dropout_8 (Dropout)         (None, None, None, 100)   0

 conv2d_7 (Conv2D)           (None, None, None, 80)    72080

 batch_normalization_7 (Batc  (None, None, None, 80)   320
 hNormalization)

 max_pooling2d_7 (MaxPooling  (None, None, None, 80)   0
 2D)

 dropout_9 (Dropout)         (None, None, None, 80)    0

 conv2d_8 (Conv2D)           (None, None, None, 32)    23072

 batch_normalization_8 (Batc  (None, None, None, 32)   128
 hNormalization)

 max_pooling2d_8 (MaxPooling  (None, None, None, 32)   0
 2D)

 dropout_10 (Dropout)        (None, None, None, 32)    0

 flatten_2 (Flatten)         (None, None)              0

 dense_4 (Dense)             (None, 8)                 73992

 dropout_11 (Dropout)        (None, 8)                 0

 dense_5 (Dense)             (None, 4)                 36

=================================================================
Total params: 172,828
Trainable params: 172,404
Non-trainable params: 424
_________________________________________________________________
```

Σχήμα 4.4: CNN architecture for classification problem

Also, as mentioned for the Batch Normalization layer, due to the overfitting problems that appeared in the first few attempts of the model analysed in this section, I decided to add a two-dimensional Max Pooling layer after each filter in order to provide to next layer only an abstracted form of the representation. By using this layer, I managed to reduce the over-fitting as I downsampled the input of the next layers and provided a basic translation invariance to the internal representation. After applying this triplet of layers multiple times, I used a dropout after each triplet to randomly ignore some neurons temporarily. Finally, to ensure that my input will be entered as one-dimensional to the final dense layers in combination to extra dropout layers. Before exiting the model, I used a final dense layer with a softmax activation function as it is used in multinomial cases to normalize the output of the model.

The metrics used in the present model were the categorical cross entropy function as a loss function, the Adam optimizer and accuracy and precision. In each run, I monitored validation accuracy and with the use of the Early Stopping callback I stopped the training after meeting its maximum value during 5 epochs, as I considered that the model would not improve further. The best results retrieved from this model with a learning rate $lr = 0.0001$ can be seen in the table below [4.5]:

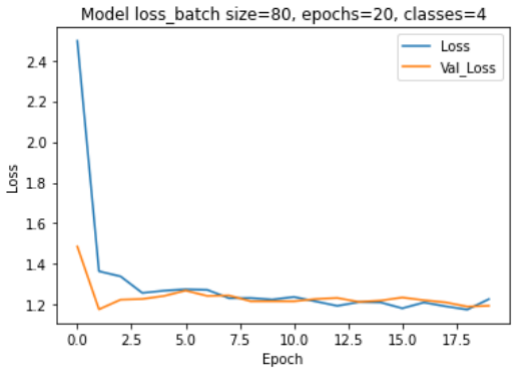
	loss	accuracy	precision_3	val_loss	val_accuracy	val_precision_3
0	2.496617	0.456500	0.439192	1.486472	0.200000	0.211982
1	1.364678	0.665000	0.646102	1.177607	0.710417	0.699797
2	1.339337	0.683000	0.681728	1.225186	0.718750	0.697395
3	1.258200	0.714932	0.701617	1.228504	0.691667	0.688391
4	1.269119	0.705500	0.703335	1.243383	0.679167	0.677618
5	1.276160	0.702500	0.701292	1.269683	0.685417	0.683128
6	1.273295	0.703500	0.703039	1.243559	0.731250	0.726337
7	1.232521	0.718500	0.712099	1.245967	0.691667	0.695122
8	1.233191	0.728500	0.725646	1.217297	0.704167	0.704167
9	1.225855	0.740500	0.733760	1.217422	0.729167	0.721992
10	1.238567	0.742500	0.740907	1.217267	0.712500	0.701461
11	1.216776	0.725500	0.723383	1.228390	0.704167	0.707113
12	1.194726	0.733000	0.732535	1.233482	0.735417	0.730612
13	1.213123	0.727000	0.721920	1.214403	0.708333	0.713389
14	1.211676	0.738000	0.738048	1.220848	0.654167	0.652807
15	1.182723	0.727500	0.725012	1.235777	0.687500	0.687243
16	1.211328	0.719500	0.718703	1.222572	0.733333	0.734864
17	1.192584	0.732500	0.732500	1.211921	0.722917	0.721649
18	1.176320	0.747500	0.693756	1.191294	0.718750	0.715164
19	1.227817	0.716500	0.620489	1.195041	0.712500	0.723493

Σχήμα 4.5: CNN statistics with lr=0.0001

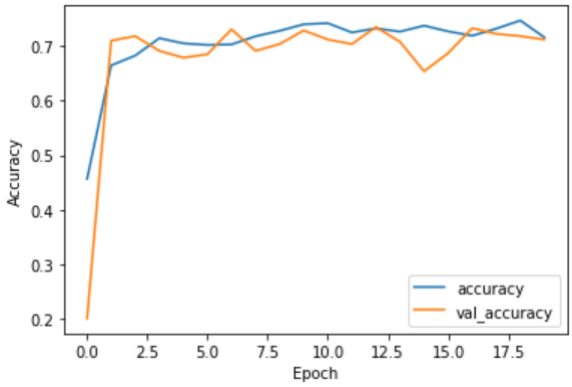
The results were obtained with parameters set as batch size = 80, epochs = 20, neurons=4. The model was executed on a GPU for 113.33 minutes with average epoch time equal to 7.25 minutes.

As one can observe, with this model we managed to reach the highest training accuracy of 74.25%, highest precision 74.091% in row 10, but the highest validation accuracy in row 12, which is equal to 73.54% and validation precision equal to 73.06%. Finally, both the training loss function and validation loss function follow

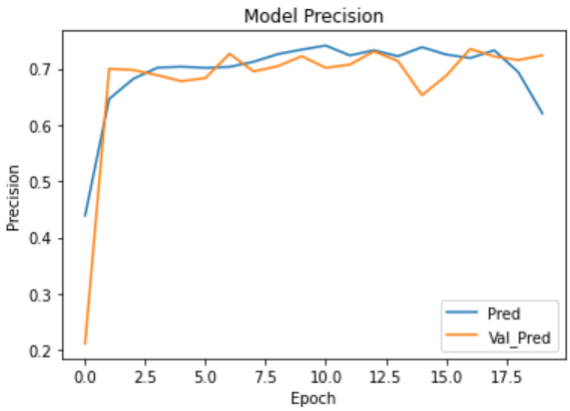
the descending x^2 behaviour that they are supposed to do. One can see the visualization of these numbers below:



Σχήμα 4.6: Model Loss (batch size=80, lr=0.0001, epochs=20)



Σχήμα 4.7: Model Accuracy (batch size=80, lr=0.0001, epochs=20)



Σχήμα 4.8: Model Precision (batch size=80, lr=0.0001, epochs=20)

In order to be sure that these results were the best we could achieve with this specific model architecture, I did a parameter analysis in order to find the optimized parameters.

	Avg Acc.	Avg Val.Acc.	Avg Prec.	Avg Val.Prec.
batch s=40, lr=0.01	0.720247	0.72542	0.703839	0.635477
batch s=80, lr=0.01	0.705395	0.68157	0.694795	0.679685
batch s=100, lr=0.01	0.726905	0.7267	0.71905	0.6789
batch s=100, lr=0.001	0.710743	0.69432	0.700125	0.693432
batch s=100, lr=0.0001	0.706395	0.68157	0.694745	0.679685

Thus, we observe from the above results that the optimum parameters are batch size=100 an learning rate=0.01 and we observe that the model's performance is worse better as we increase the batch size. This conclusion comes from the validation accuracy and validation precision, which are closer to the training value, when we increase the batch size. Especially, with such big difference between the training metrics and the validation metrics, we could assume that there is some kind of underfitting that is happening. Concerning, the dependence of the model's performance to the learning rate parameter, we notice that the optimum value is lr=0.01 as it achieves the highest performance and it doesn't cause overfitting. If we increased the learning rate even more, one could notice great overfitting.

2. Graph Neural Network (GNN)

For the purpose of this chapter we aim to utilise Graph Neural Networks to make predictions on our dataset, containing information about molecules from the QM9 database and finally clarify if some of them can be used as components of antidepressant drugs. To construct this model, I used TensorFlow and PyTorch. In this specific case, I decided to use a Graph Convolutional Network (GCN). However, the first task that need to be done is to prepare our input data. The graph data will be instantiated from the dataset's column containing the SMILES strings. In order to transform all SMILES to graph input, I create a "smiles2graph" function, which receives a SMILES string as the only argument. To manage this input vectors, I also utilize the graphs already contained in the QM9 dataset, the adjacency matrix for each respective bond that pairs atoms in each molecule.

To continue with, I also define a custom function that receives SMILES strings as the only input argument and outputs the atom features for an individual molecule

	nodes	edges
count	19672.0	19672.0
mean	8.9	21.2
std	0.4	1.2
min	5.0	12.0
25%	9.0	20.0
50%	9.0	22.0
75%	9.0	22.0
max	9.0	22.0

Σχήμα 4.9: Graph dataset statistics

in the form of a vector. The dataset used is the same used in the previous chapter for the CNN model. Now that I have the graphs with features, it's time to define the labels. I define the 4 different classes as done before, using the logP coefficient as a criterion:

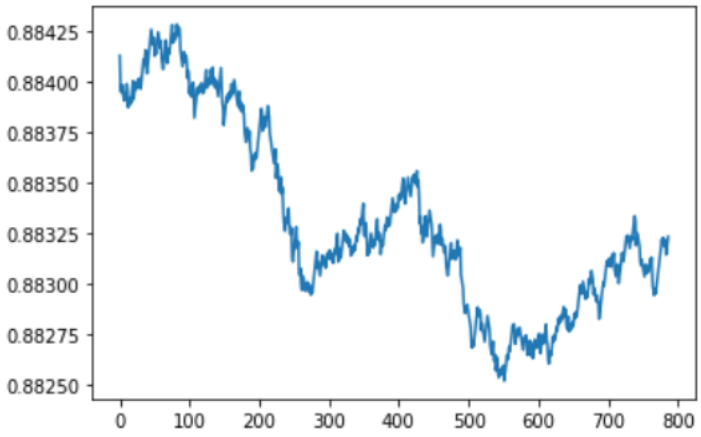
- Class 1: $logP < -2.84025$,
- Class 2: $logP \in [-2.84025, -0.7715)$,
- Class 3: $logP \in [-0.7715, 1.29725)$ and Class 4: $logP >= 1.29725$

Then I use the One-Hot Encoding method to embed my data. Then I construct a DGL graph dataset. By printing my dataset, I observe the form that the data have shape into. Each row of data contains the number of nodes, number of edges and the respective feature vector with its shape. Next, I define a class that receives as an argument the DGL dataset and creates the final synthetic dataset that i will use as input for my Graph Convolutional Network. At this point, I use a Graph data loader and a random sampler and I also define a class for the GCN model that contains the forward and backward passings.

As a final step of the data preprocessing stage, I extracted the below statistics concerning the nodes and edges of the generated graphs in the dataset. Now that I have gathered all the prerequisites, I define my model with the help of the GCN class and I use the length of the feature vector and the number of classes. I also utilise an Adam optimizer and PyTorch's cross entropy function as a loss function. For the

initialization of the model, I used an AtomEncoder combined with a dataloader, 3 convolutional filters and a linear layer on the hidden channels based on the number of classes in the dataset. For the forward passing, I first obtain the node embeddings, then I added a global mean pool as a readout layer and finally I use a final classifier.

After running the model, we receive an average loss function value equal to 0.114 and an average validation equal to 88.352%, which is a great result. The general behavior of the accuracy as function of the batch size can be seen below [4.10]:



Σχήμα 4.10: GCN Validation Accuracy vs Batch size

To achieve these results I used a total of 64 hidden layers, a learning rate equal to 0.01 and dropout equal to 0.5 on the final classifier of the model.

Thus, we can conclude that it is more effective to use a Graph Convolutional Network instead of a CNN to classify the molecules of this dataset. The great advantage that a GCN has and it makes it very powerful is that it can receive graphs with variable size as input, contradicting a Convolutional NN, which can only understand input with specific size. We would like to construct its quantum analogous to be able to compare the results and conclude on the quantum advantage, but as mentioned in literature, Quantum Graph NNs are very complex models and are still not completely feasible, but we have studied them in a theoretical level. As mentioned in a paper from CERN, where they implemented a QGNN to predict particle trajectories, it is has a very high computational cost as 1 epoch had a duration of 1 week.

3. Generative Adversarial Network (GAN)

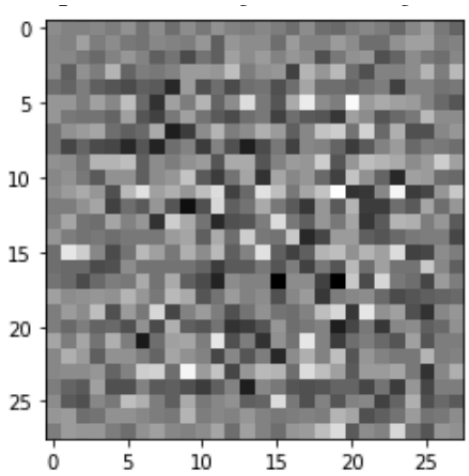
For this chapter, I implement a Generative Adversarial Network and I use the molecule images as input for my GAN model. The starting size that I import the images is 600×600 pixels. Then after slitting my data to train, test, validation datasets, I define the new dimensions for our images, which will be 28×28 pixels. I also use a batch size equal to 100, with a training sample of length equal to 800 and 20 training epochs. My dataset has been split to 11829 images for the train dataset, 3944 images for the validation dataset and 1945 images for the test dataset, with the use of Image Data Generators. Through the appliance of the generators, I also normalize images to fit [0, 1].

Subsequently, I set a buffer size equal to the size of the training sample and I shuffle the data. Then I define the generator model, which is a sequential model. As a first step I use a Dense layer with Batch Normalization and a Leaky ReLU activation function. Then I use three layers with a two-dimensional transpose convolutional filter combined again with Batch Normalization and a Leaky ReLU activation function. However, in the third layer I use a hyperbolic tangent, tanh, activation function instead of a Leaky ReLU. All the above can be summarized in the model diagram [4.11].

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 12544)             1254400

 batch_normalization (BatchN  (None, 12544)            50176
 ormalization)

 leaky_re_lu (LeakyReLU)     (None, 12544)             0

 reshape (Reshape)           (None, 7, 7, 256)         0

 conv2d_transpose (Conv2DTra  (None, 7, 7, 128)        819200
 nspose)

 batch_normalization_1 (Batc  (None, 7, 7, 128)        512
 hNormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 7, 7, 128)         0

 conv2d_transpose_1 (Conv2DT  (None, 14, 14, 64)       204800
 ranspose)

 batch_normalization_2 (Batc  (None, 14, 14, 64)       256
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 14, 14, 64)        0

 conv2d_transpose_2 (Conv2DT  (None, 28, 28, 1)        1600
 ranspose)

=================================================================
Total params: 2,330,944
Trainable params: 2,305,472
Non-trainable params: 25,472
_________________________________________________________________
```

Σχήμα 4.11: Generator Model Summary

Then I add random noise to an image and I feed it into the generator to see the results [4.12].



Σχήμα 4.12: Noisy Image

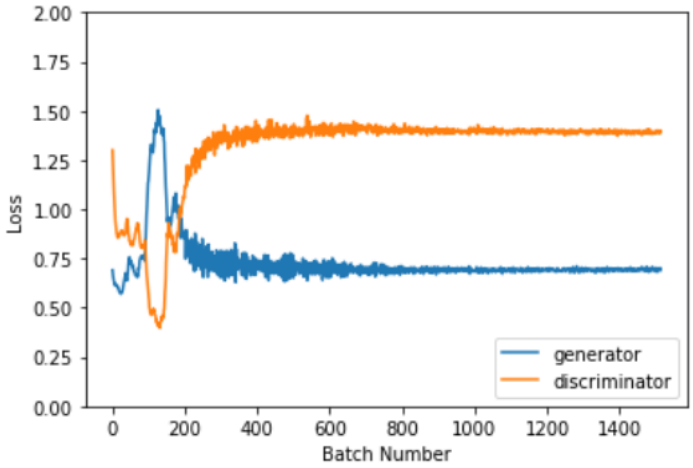
In the next step I define the discriminator model, which is also a sequential model. As a first step, I add two two-dimensional Convolutional filters combined with a Leaky ReLU activation function with a dropout=0.3. Lastly, I use a Flatten layer and a Dense layer to conclude this model.

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)            (None, 14, 14, 64)        1664

 leaky_re_lu_8 (LeakyReLU)    (None, 14, 14, 64)        0

 dropout_2 (Dropout)          (None, 14, 14, 64)        0

 conv2d_3 (Conv2D)            (None, 7, 7, 128)         204928

 leaky_re_lu_9 (LeakyReLU)    (None, 7, 7, 128)         0

 dropout_3 (Dropout)          (None, 7, 7, 128)         0

 flatten_1 (Flatten)          (None, 6272)              0

 dense_3 (Dense)              (None, 1)                 6273

=================================================================
Total params: 212,865
Trainable params: 212,865
Non-trainable params: 0
_________________________________________________________________
```

Σχήμα 4.13: Discriminator Model Summary

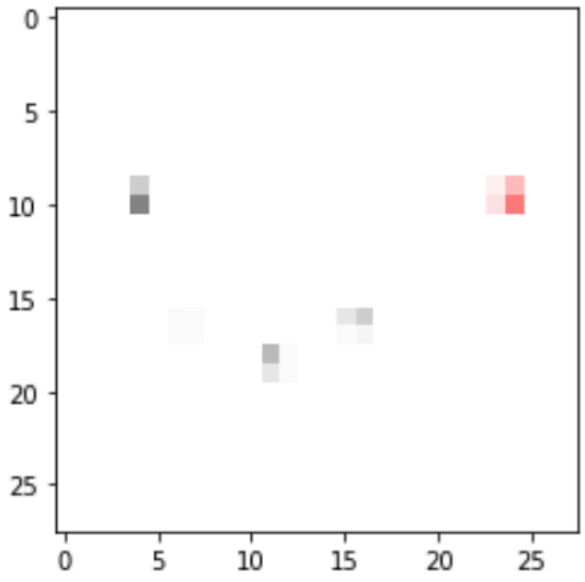
Then I use an Adam optimizer and custom loss function for each model. I define

the noise's dimension to 100, epochs to 15 and I train the whole model. Below, I display the loss function comparing the generator and discriminator [**??**].

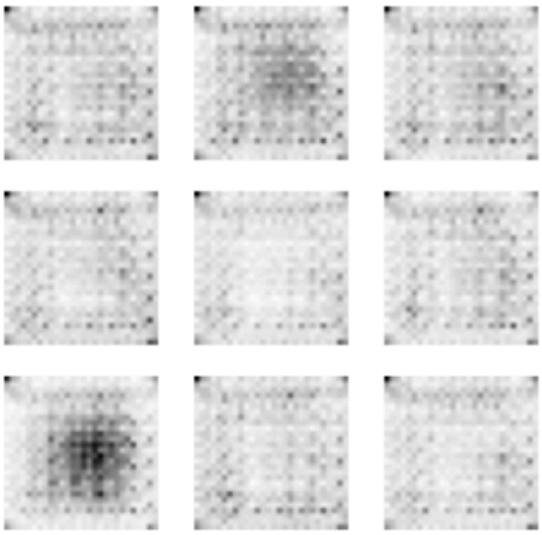


Σχήμα 4.14: Loss function of Generator vs the Discriminator as a function of batch size

The generated images from different running sessions are [] for a generated image in RGB and the [], where I tried to work with grayscale images.



Σχήμα 4.15: RGB Generated Image



```
Time for epoch 9 is 233.37536644935608 sec
```

Σχήμα 4.16: Grayscale Generated Image

Both images are very blurry due to the noise and we concluded to the fact that we ran the algorithm for a little number of epochs. If we continued to run the algorithm for a great number of epochs we would get a better image.

Κεφάλαιο 5

Quantum ML Models - Experimental Implementation

1. How to Run Quantum Algorithms

The most important question is in which device do we run our quantum algorithms and test our quantum circuits? If we want to describe the case of Qiskit library, one can use IBM backends. Most of the time quantum circuits are executed on Qiskit simulators. How do we compute a quantum circuit on a classical computer? We need to reconsider what a quantum state and operator really are. As used in previous chapter, in quantum mechanics, we use the Dirac notation that contains the $bra-ket$, $|>$. In a simple mathematical language, we can translate this state to a vector:

$$|0>=\begin{pmatrix}1\\0\end{pmatrix}$$

Quantum operators can again be translated to matrices. If for example we use the Hadamard operator, H, we use a matrix of the form:

$$H=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$$

Thus, one can easily understand that computing a quantum circuit is just a matrix multiplication. The resulting vector denotes the measurement amplitudes, whose squares are the measurement probabilities. Here is a point that one can more easily understand a part of the quantum supremacy. As all these operations can be thought as matrix multiplications, one can easily understand that if the matrix dimensions are too large, this operation would be very time-wasteful and power-consuming for

a classical computer. If we put it in a simple way, quantum computers can multiply many matrices in a single step.

One of the most popular providers for simulators is the one that provides for Basic Aer backends. This is a module of Python-based quantum simulators. There are several implementations. One can use the qasm simulator to retrieve the measurement counts empirically. This simulator takes into consideration only the classical bits we use when we measure our qubits. It uses a parameter called ”shots” to indicate to the computer how many times to run the circuit and to obtain the measured result. As we increase the number of shots, we increase our results accuracy. This simulator may not be the more accurate choice but it provides to the user the most realistic conditions, such as noise. If someone wants to use a simulator to calculate the exact state a qubit is in, he would probably have to choose the statevector simulator. However, one important note before using this simulator is to remove all measurements in our quantum circuit. As mentioned in a previous chapter of this thesis, if we measure the state of any qubit in our circuit, this will collapse the quantum superposition and inevitably result in a definite state the system could possibly be in. The statevector simulator backend calculates the state of the given quantum system. Finally, another popular simulator provided by the Basic Aer package is the unitary simulator. This simulator executes the circuit once and returns the final transformation matrix of the circuit itself. Running the circuit on an input state simply is multiplying the transformation matrix with the state vector. The same as before, when we use this simulator our circuit must not contain any measurements.

Apart from all the simulators, a few quantum computers have been built and are available for public use, hosted by IBM-Quantum. All calls are executed via the IBMQ API over the Internet and one has to sign-up in order to obtain an API key to access his account and execute his code on the remote servers. The first step is to load your IBMQ account using API key. Once the account has been loaded, you can retrieve a backend provider to execute your code on a quantum computer. In these backends, you can find either simulators or real quantum computers. As input they receive a qobject, which is the Qiskit API serialization format and returns a BaseJob object. This object allows asynchronous running of jobs for retrieving results from a backend when the job is completed. While running your quantum circuit, you can monitor the run status of a job on a quantum computer by using several commands as shown below.

```python
from qiskit.tools import job_monitor

backend2 = qiskit.BasicAer.get_backend('qasm_simulator')
backend2.name() # qasm_simulator
backend2.version # 1
backend2.provider().backends() # [<QasmSimulatorPy('qasm_simulator')>,

job = qiskit.execute(qc, backend2)

job_monitor(job) # Problem status of the running job
# Job Status: job has successfully run

job.status()
# <JobStatus.DONE: 'job has successfully run'>

job.result().get_counts()
```

Σχήμα 5.1: Job Monitoring

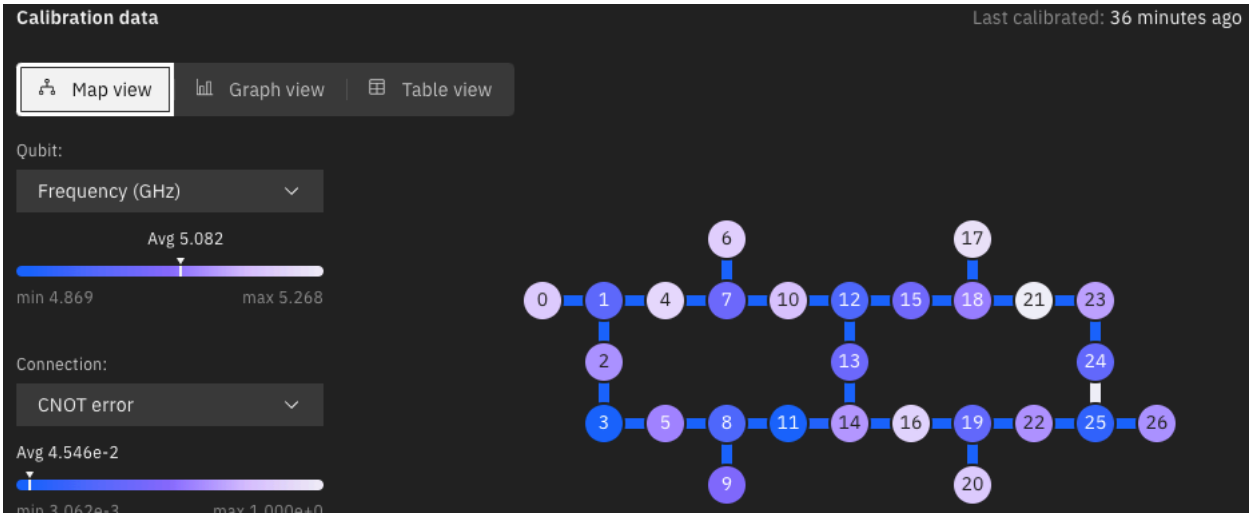
When choosing a real quantum computer one has to be cautious. It is very important to choose a quantum computer that can provide him the most suitable architecture for the specific use. One can see all available systems by visiting their personal accounts.

Name	Qubits	QV	CLOPS	Status	Total pending jobs	Processor type
ibm_washington Exploratory	127	64	850	Online - Queue paused	1784	Eagle r1
ibmq_brooklyn Exploratory	65	32	1.5K	Online	26	Hummingbird r2
ibmq_kolkata	27	128	2K	Online	91	Falcon r5.11
ibmq_montreal	27	128	2K	Online	280	Falcon r4
ibmq_mumbai Exploratory	27	128	1.8K	Online	658	Falcon r5.1
ibm_cairo	27	64	2.4K	Online - Queue paused	308	Falcon r5.11
ibm_auckland Exploratory	27	64	2.4K	Online	572	Falcon r5.11
ibm_hanoi	27	64	2.3K	Online	303	Falcon r5.11
ibmq_toronto	27	32	1.8K	Online	128	Falcon r4
ibm_peekskill Exploratory	27	-	-	Online	0	Falcon r8

Σχήμα 5.2: Real Quantum Computers

As displayed in the image above, all specifications are provided including if the quantum computer is available. Then some architectural information are provided,

such as the number of qubits and the Quantum Volume (QV) which is a metric that measures the capabilities and error rates of a quantum computer. The QV method quantifies the largest random circuit of equal width and depth that the computer can successfully implement. To continue with, for every system we have the CLOPS metric, which stands for Circuit Loop Operations per second. This is a metric correlated with how fast a quantum processor can execute circuits. More specifically, it measures the speed the processor can execute layers of a parametrized model circuit of the same sort used to measure QV. Finally, one can see the jobs that are pending to be done on the specific system and the type of the processor they run on. Additionally, on this page one can find more information on each system such as the qubits' frequencies in GHz and the error rates.



Σχήμα 5.3: ibmq_kolkata error map

In the above image, the information mentioned above are displayed for the IBM kolkata system along with its error map. As an error map we define the visualization of the node connections of a gate map, in addition to the error rate expected on the backend. By using the term gate map we refer to the visualization of the connections between the nodes on a physical quantum computing device.

Now, if someone uses the Pennylane package combined with Qiskit has to know that this quantum machine learning library is designed from the ground up to be hardware and device agnostic, allowing quantum functions to be easily dispatched to different devices. A single computation can even include multiple quantum devices from different vendors. Pennylane offers several built-in quantum devices, such as simple state-vector qubit simulator written in Python or using TensorFlow and

supporting classical backpropagation. Other choices include a fast state-vector qubit simulator written with C++ backend or a mixed-state qubit simulator written in Python. Several plugins can be installed separately, including integrations with Qiskit, Amazon Braket, Cirq, Strawberry Fields and more.

2. Comparison of Encoding Methods for Quantum Variational Classifiers

Before we move on to constructing different types of quantum circuits to transform the classical models to their quantum analogous. In this section, I aim to study different techniques on encoding the information provided by my dataset, which is a subset of the QM9 database. I design a variational quantum circuit for each encoding and I use the 4 out of the 5 columns of my dataset as the target to implement a basic binary classification of 0 and 1. The classifiers used are based on the column containing the logP coefficient data and the column containing the atomic number of each molecule. I considered to use a layer to design the ansatz that I will use for these models and I will study in which way the number of layer affects the performance of our quantum circuit. Thus, all processes shown below can be outlined as the follow steps:

1. Data pre-processing,
2. Quantum Embedding of the classical data and
3. Training of a layered variational quantum classifier

I begin the study of this chapter by shaping my dataset. I use the initialised dataset

Unnamed: 0	SMILES	logP
0	C1C2CC1C2	0.63610
1	C1C2CC1O2	0.16200
2	CC12CC(C1)C2	-0.82470
3	CC12CC(C1)O2	0.24940
4	CC12CN(C1)C2	0.13978
...	...	...
19667	FC(F)(F)C12CC(C1)O2	1.13150
19668	FC(F)(F)C12CN(C1)C2	0.07450
19669	FC(F)(F)C1C2CC1C2	-0.06800
19670	FC(F)(F)C1C2CC1O2	0.26280
19671	FC(F)(F)C1C2CN1C2	0.07450

Σχήμα 5.4: Initial Dataset

and by using this I obtain numerical features for each molecule that are vital to understand their properties and that define each molecule's structure. The features calculated are

- Number of atoms,
- Number of bonds,
- Atomic Number (Z),
- logP coefficient,
- Number of single bonds and
- Number of double bonds

Finally, the dataset that will be used for all the following techniques is the below:

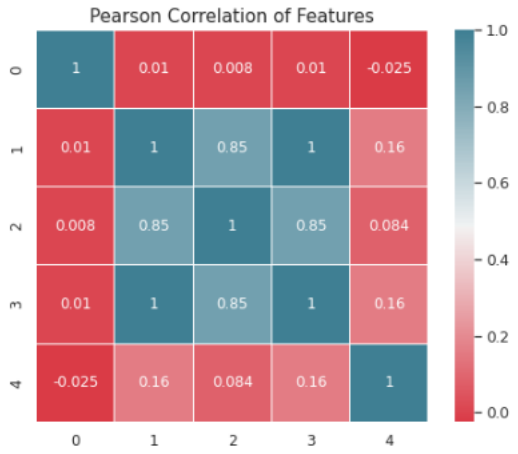
	logP	atomic_numbers	num_atoms	Z	double_bonds
0	0.63610	30	5	30	0
1	0.16200	32	5	32	0
2	-0.82470	36	6	36	0
3	0.24940	38	6	38	0
4	0.13978	37	6	37	0

Σχήμα 5.5: Dataset

and by using the pandas library I obtain the below statistics, which characterize the specific dataset:

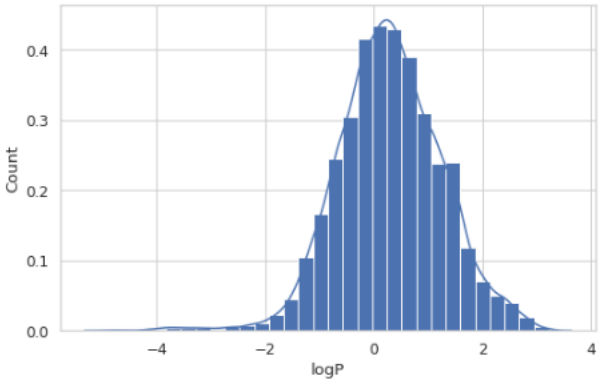
	logP	atomic_numbers	num_atoms	Z	double_bonds
count	19672.000000	19672.000000	19672.000000	19672.000000	19672.000000
mean	0.301025	56.366816	8.870730	56.366816	0.494866
std	0.963120	2.898807	0.380119	2.898807	0.604471
min	-4.909700	30.000000	5.000000	30.000000	0.000000
25%	-0.304680	56.000000	9.000000	56.000000	0.000000
50%	0.283300	57.000000	9.000000	57.000000	0.000000
75%	0.936770	58.000000	9.000000	58.000000	1.000000
max	3.366800	65.000000	9.000000	65.000000	3.000000

To better understand our dataset, I provide some additional statistics, such as a heatmap of the Pearson correlation of features



Σχήμα 5.6: Pearson Correlation

and the range of values of the first column



Σχήμα 5.7: Range of logP coefficient

Finally, if someone wants more information describing the dataset, he can look into the Appendix of this thesis, which contains a pairplot of the dataset.

After all the statistics displayed above, I continue with the preprocessing of the dataset's features. I start by normalizing the columns in train and test datasets to fit in the range [0,1]. After the rescaling, I split test dataframe into the respective feature set and ground truth labels and I freeze the input training and test data. Finally, I shift ground the truth labels from 0, 1 to -1, 1 to match the expectation values of the Pauli Z matrix. In addition, I split my data to train and test sub-sets and shuffle them and split into train and validation sets.

Amplitude Encoding Technique:

Now, as a first technique I try the Amplitude Encoding method. Since I have 4 features in my dataset, I decide to use 2 qubits and I use the "default.qubit" device provided by the pennylane package and I use the default shots=100, on how many times to repeat the circuits measurements, to provide the final results. First, I define a layer template as a sequence of trainable gates. This is similar to the layers in a neural network. This can be achieved by using a single qubit for rotations and a few controlled NOT gates (CNOT) as entanglers between the pairs of qubits. As a second step, I initialize the quantum device and prepare the quantum dataset by using Hadamard gates. After the initialization, I can finally embed my classical data as a quantum state in the Hilbert space and create my quantum model.

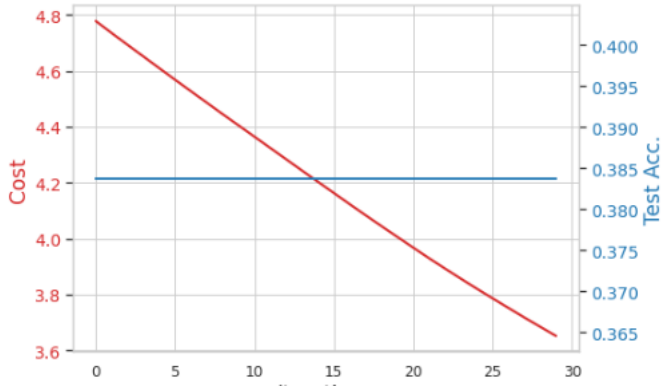
In order to be able to measure the techniques performance, I define a loss function and specifically a squared loss function and the accuracy for the binary classification model. On the other hand, I define the quantum Variational Classifier model and its respective loss function and I train the model. For training, I initialize

the parameters, the number of layers as 1, the initial weights and initial bias of the model.

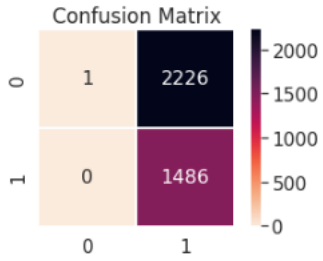
I also use batch-size=120, epochs=30, learning rate=0.01 for an Adam optimizer. The ouptut weights that I receive are equal to

$$\begin{pmatrix} 0.02082524 & 0.00774552 & -0.01176826 \\ -0.00760863 & -0.00517855 & 0.00466557 \end{pmatrix} \tag{5.1}$$

The results though, are displayed below.



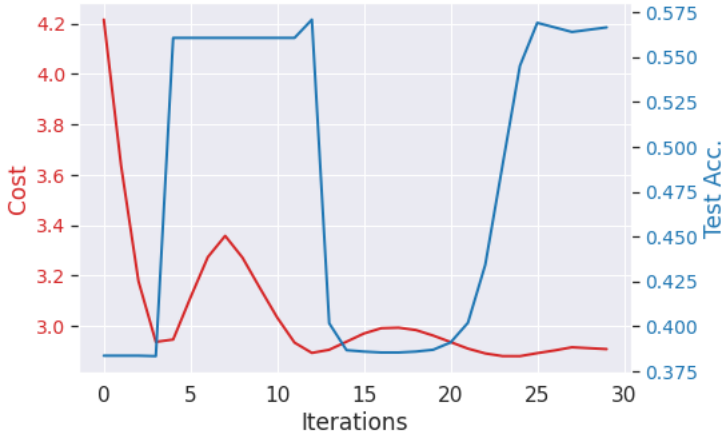
Σχήμα 5.8: Model Accuracy Cost for 1 layer



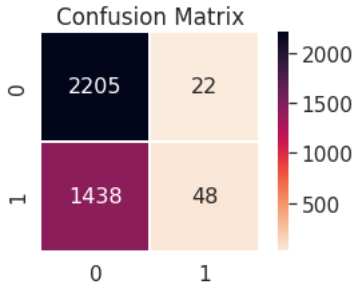
Σχήμα 5.9: Confusion Matrix of the single layer model

As it can be observed by the first plot, accuracy stays constant and it is equal to 38.5%, which is pretty bad and also the loss has a linear descending order instead of x^2 descending behaviour. Thus, we conclude the the model fails to classify the data.

To continue our analysis, I try using batch-size=170, epochs=30, learning rate=0.01 and 2 layers for the same model. This time I received the below results:



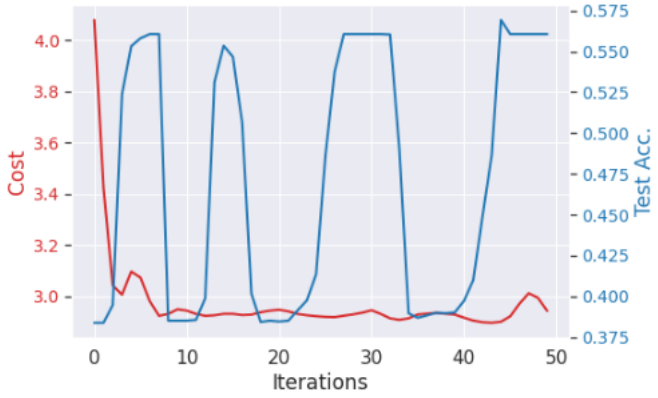
Σχήμα 5.10: Model Accuracy Cost for 2 layers



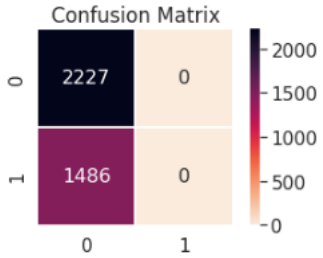
Σχήμα 5.11: Confusion Matrix of the two-layer model

Here we observe a slight improvement as the model accuracy increased approximately 25% and reached an average value of 57.26%. Also the cost function has better behavior as it is descending and non linear and it seems that at the last epochs, it slowly converges.

Lastly, I tried for the same parameters to increase the number of layers to 3. The results were similar to the previous case, concerning the accuracy, as it reached 56.595% again. However, the cost function becomes smoother and the convergence of the model is faster and smoother.



Σχήμα 5.12: Model Accuracy Cost for 3 layers



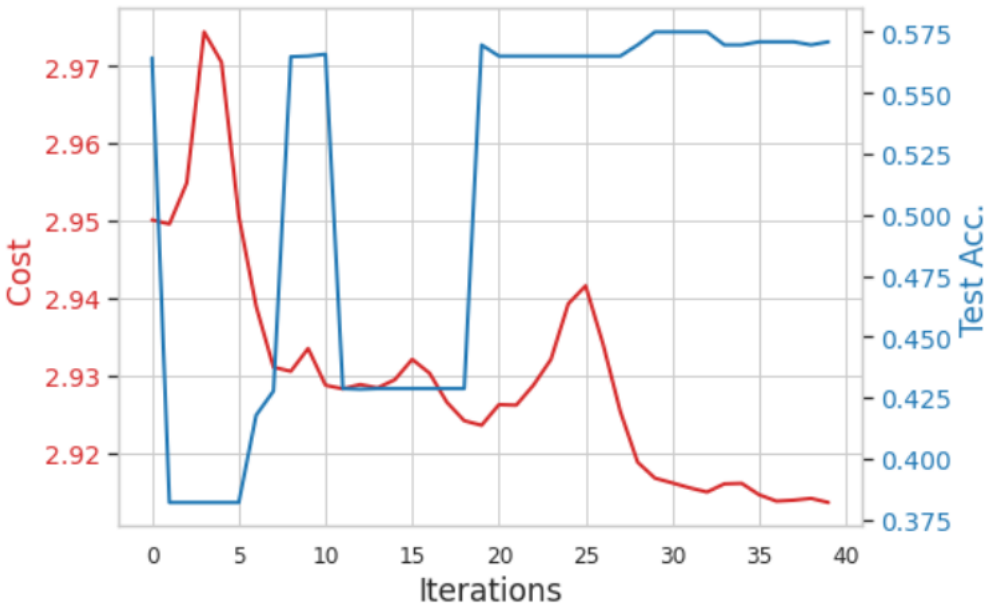
Σχήμα 5.13: Confusion Matrix of the three-layer model

Thus, the general conclusion for this technique is that we cannot reach very high test accuracy. However, if we used a more complex classifier, it may output better results. However, we noticed that by increasing the number of layers in the variational classifier, we manage to speed up the convergence of the model and it also becomes smoother. The performance of the model is increased. For the specific optimizer, Adam, with a learning rate of 0.1 we still achieve a faster initial convergence, however as one can see in the diagrams, the performance is unstable and saturates quickly. If we decrease the learning rate, the classifier has a slower but smoother convergence.

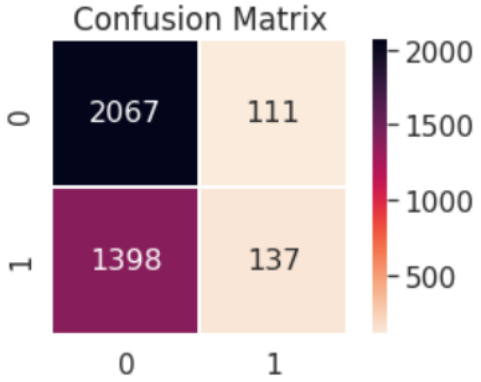
Angle Encoding Technique:

In this method I follow the exact same preprocessing flow for the pre-trained data. The great difference between this technique and the previous one is the number of qubits that are needed for the implementation. Since there are 4 features in our dataset, we require 4 qubits to perform the encoding. In Angle encoding, N features are encoded into the rotation angles of n qubits, where $N <= n$.

I begin with the single-layer model again. The parameters initialized are batch-size=150, total-iterations=40, learning-rate=0.1. The results that we receive are not very promising

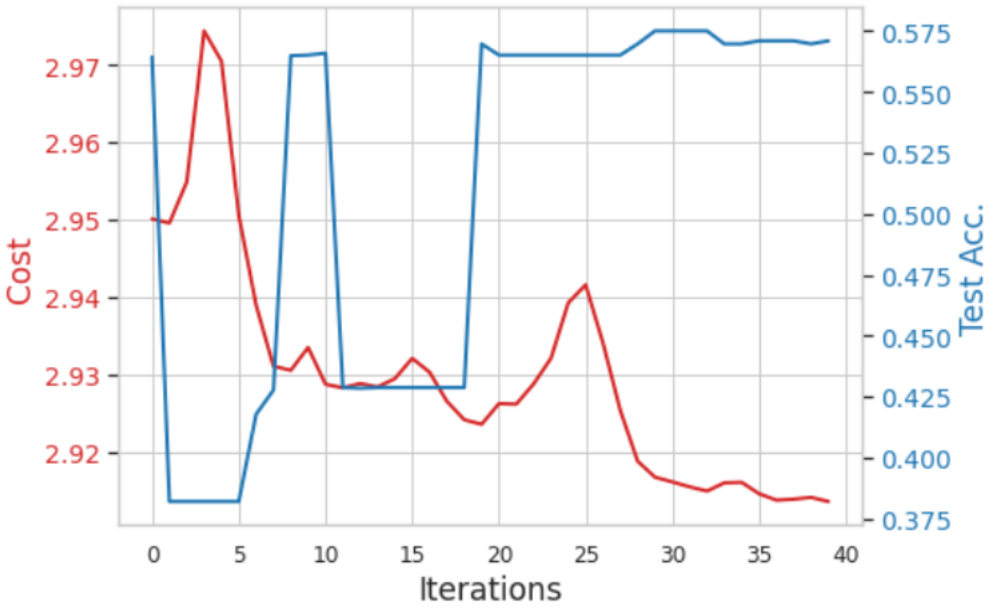


Σχήμα 5.14: Model Loss Accuracy stats - One layer



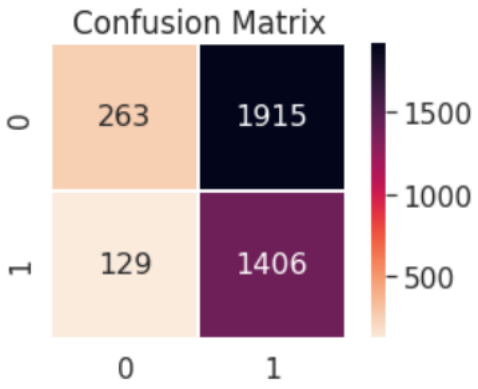
Σχήμα 5.15: Confusion Matrix of single-layer problem

Additionally, I repeat the same process with the same parameters as before, except the learning rate, which we decreased to lr=0.01. The results can be seen below:



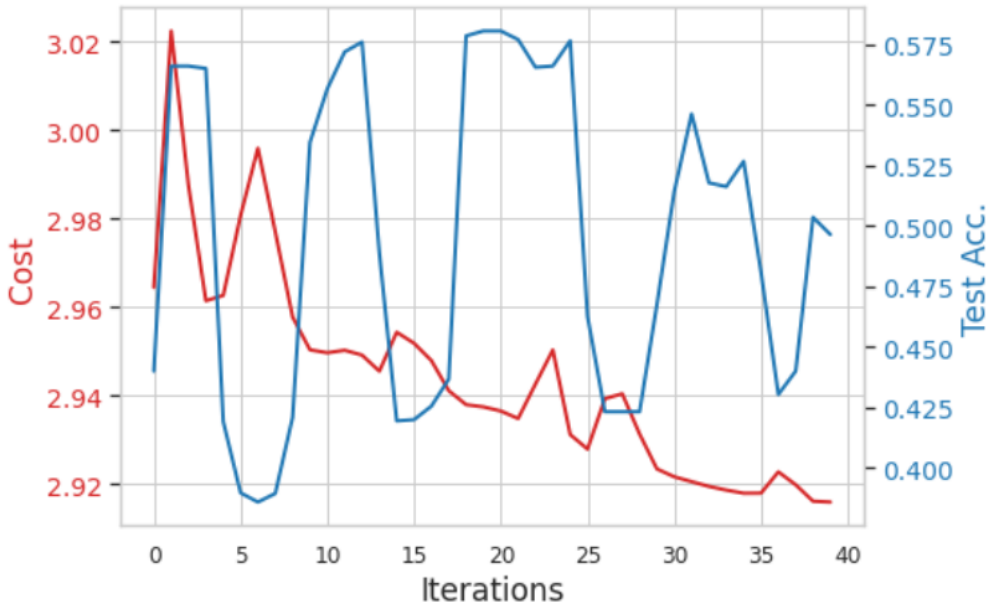
Σχήμα 5.16: Model Loss Accuracy - single-layer model

Again in this case we see that from the accuracy and the confusion model that our model fails to classify our data, with an average accuracy of 37%. Instead we see that loss function becomes smoother and converges, thus we can conclude that something is not wrong here. The bad performance can be explained as adding layers in our circuit the training time for each epoch and the model needs to be trained for more epochs. Thus, we believe that if we train our model for more epochs that the circuit will reach best accuracy and will outperform the previous circuits that have less layers. However, this has a great computational cost and we cannot satisfy this need in this process.

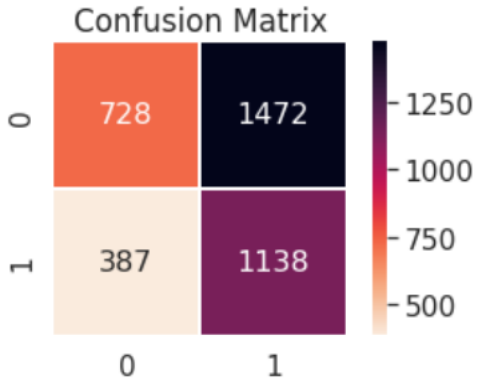


Σχήμα 5.17: Confusion Matrix of single-layer problem with lower lr

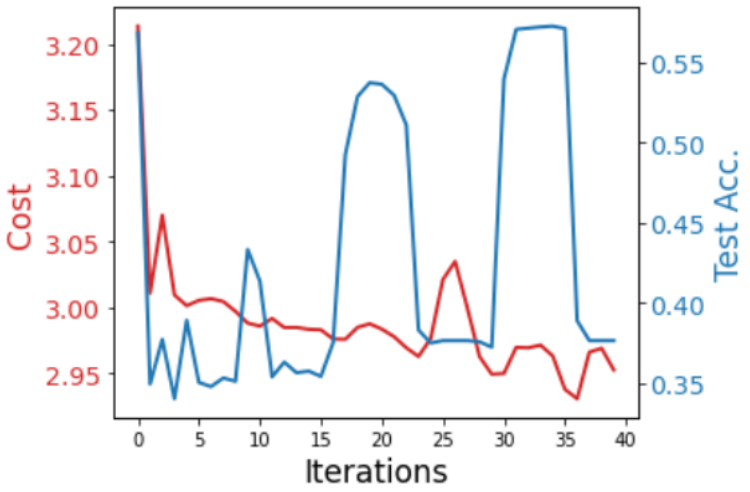
After that I repeated the first whole step, to obtain better insights and used 2 quantum layers for encoding. The respective results can are displayed in []:



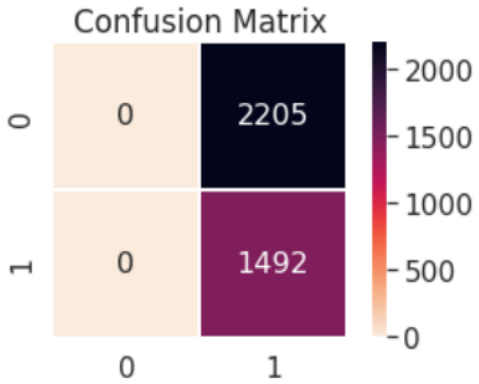
Σχήμα 5.18: Model Loss Accuracy - two-layer model



Σχήμα 5.19: Confusion Matrix of single-layer problem with lr=0.1



Σχήμα 5.20: Model Loss Accuracy - three-layer model



Σχήμα 5.21: Confusion Matrix of three-layer problem with lr=0.1

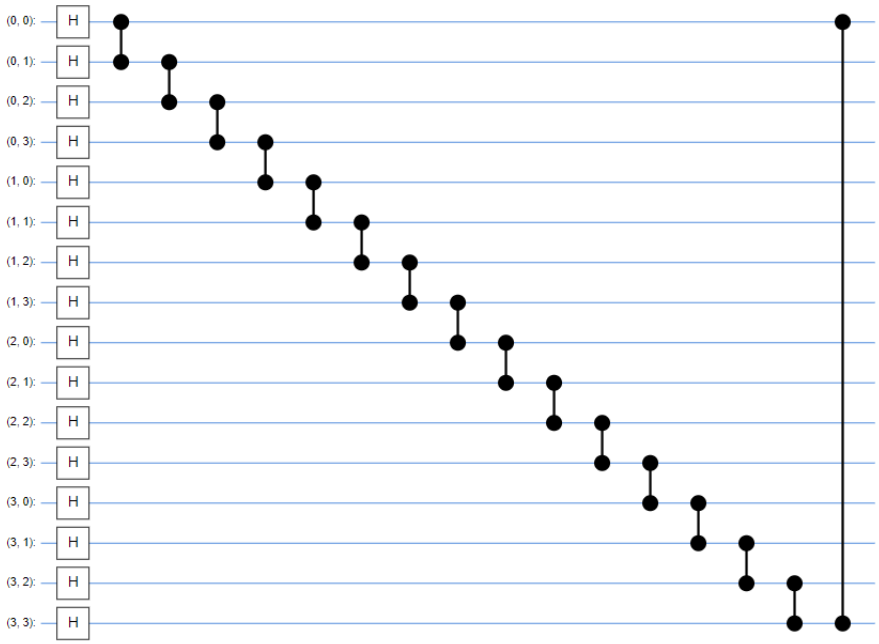
3. Quantum Convolutional Neural Networks (QCNN)

In this chapter, I tried to implement the quantum analogous of the Convolutional Neural Network that I created in a previous chapter. As we saw, in the classical model the optimum parameters were learning rate equal to lr=0.01 and batch size=100. Thus, I kept these parameters in order to be able to compare the two problems and get a conclusion on the question: can we achieve quantum supremacy in the area of CNN models?

Firstly, the problem that we study in this case, is the same as its classical analogous.

We examine the classification of the same molecule dataset, which has been split to four classes, based on the logP coefficient. In this case, we used an image generator but we did not perform data augmentation. As observed in the classical case, it probably will cause overfitting and because we didn't use it in the classical model, we want to keep the same conditions for the quantum problem too. All images were normalized to $[0, 1]$ and we have split our dataset to twelve parts. We have split it to train, test and validation dataset and each contains the four classes. After the split, we have 11829 images in the training dataset, 3944 images to use in the validation dataset and 3945 images in the testing dataset.

The quantum circuit created for this case is displayed below



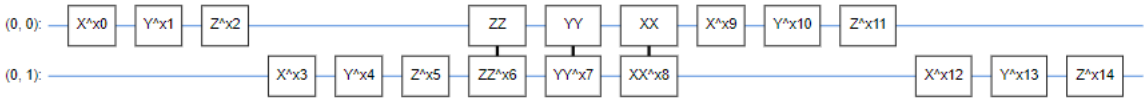
Σχήμα 5.22: Quantum circuit for Quantum CNN

As one can see in [5.22], we have applied a Hadamard gate to each qubit as a first layer of gates and then we have applied a controlled Z gate (CZ) between each qubit and its previous one. Finally, we apply a controlled Z gate between the first and the last qubit. The quantum circuit that is displayed has been designed with the use of the Cirq package, which belongs to Google.

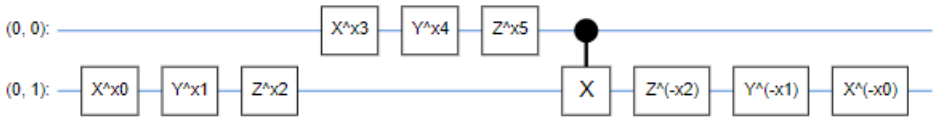
Then I create a circuit enacting a rotation of the Bloch sphere about the X, Y and Z axis and that depends on the values in "symbols". This Cirq circuit will act as a unitary operator to a single qubit [5.23]. To expand this, I also create a circuit that will act arbitrarily on two qubits [5.24] and finally I construct a third circuit to do

a parameterized 'pooling' operation, which attempts to reduce entanglement down from two qubits to just one [5.25].

Σχήμα 5.23: Single Qubit Unitary Operator



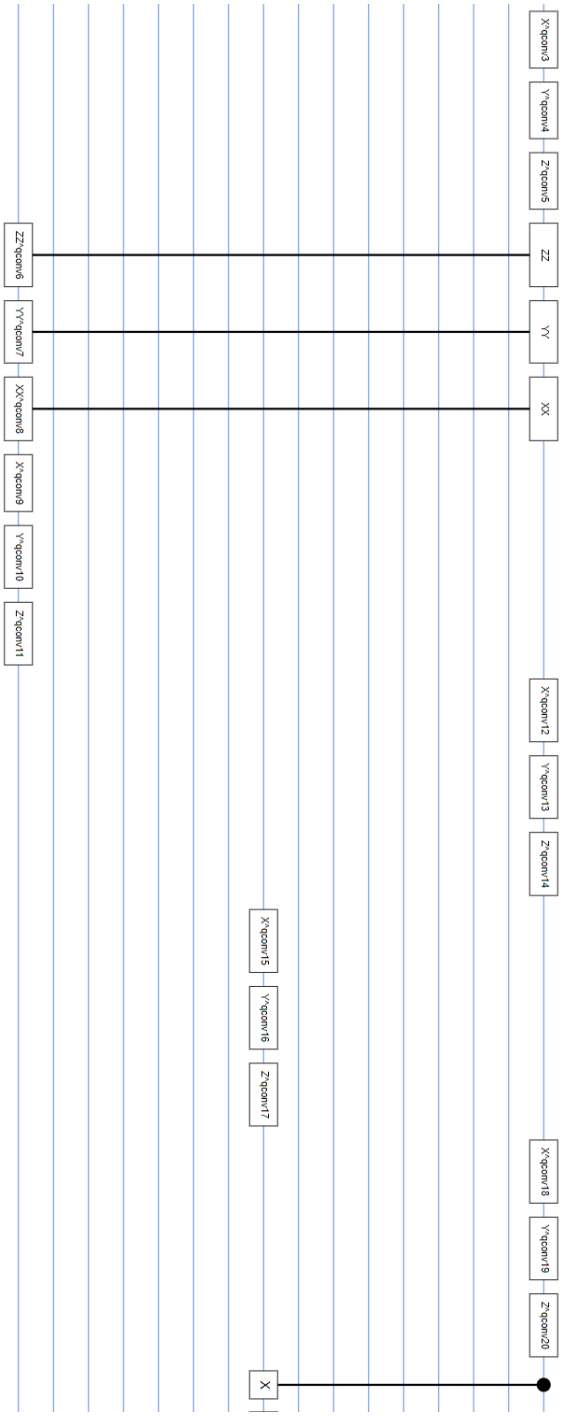
Σχήμα 5.24: Two Qubits Unitary Operator



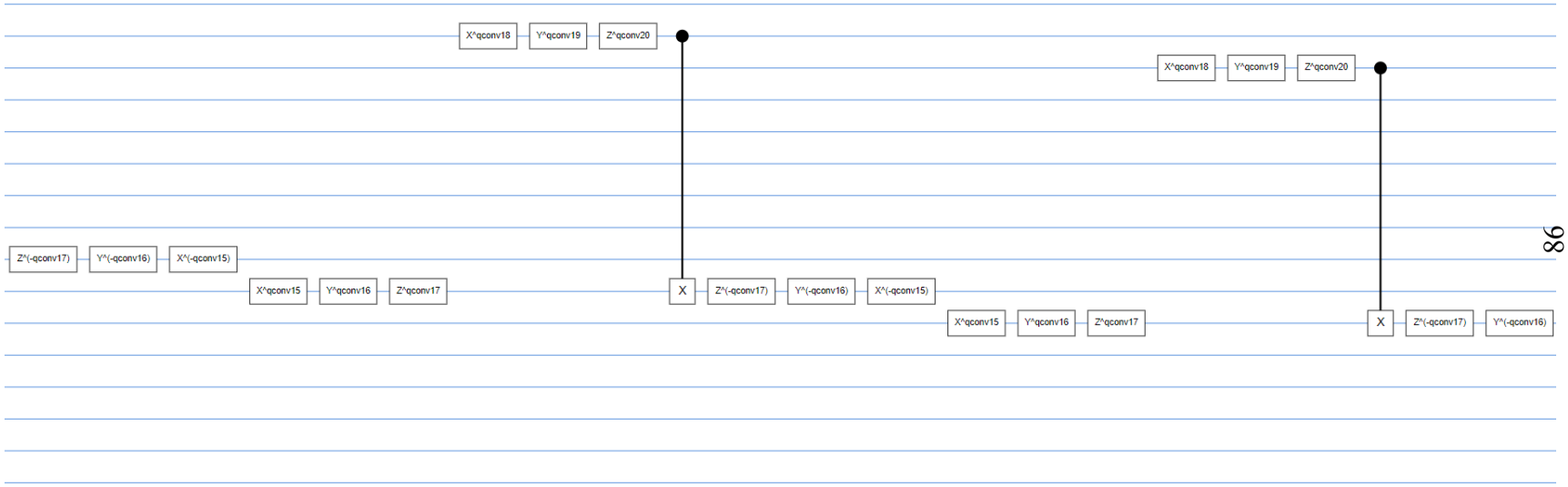
Σχήμα 5.25: Pooling on Two Qubits: Unitary Operator

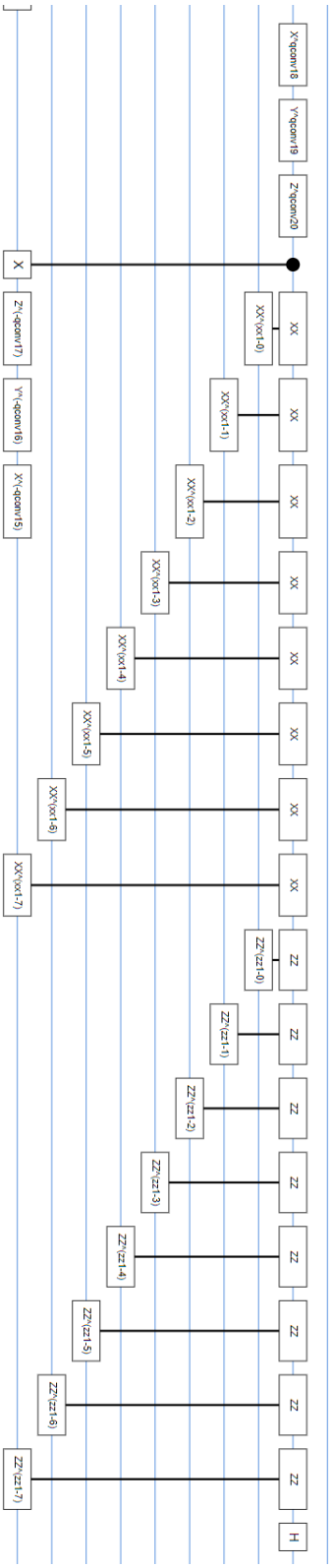
All circuits have been implemented on the basis that we use 1 qubit with variant circuit depth. The final circuit that is displayed below, uses 4 qubits in order to be able to exploit the capabilities of our operators, defined above on the specific dataset.

In order to be able to understand the fundamental case of classification in this chapter that is still totally unknown to us, I decided to present results for binary classification. This happens because all data rows though they belong to different classes, they are transformed to bit representation, in order to be able to transit to the qubit representation. Arriving to this input, we do a binary classification on the level of zeros and ones. The final circuit designed for this implementation includes two almost identical blocks of gate operations including convolutional layers. Next we continue with the following entangling layers, using controlled Z gates and Hadamard gates to cause superposition. After applying the first two blocks of quantum gates, the next layers can be seen below:



The below block is applied twice in different level of the quantum register.


X^qconv18
Y^qconv19
Z^qconv20
Z^(-qconv17)
Y^(-qconv16)
X^(-qconv15)
X^qconv15
Y^qconv16
Z^qconv17
X
Z^(-qconv17)
Y^(-qconv16)
X^(-qconv15)
X^qconv15
Y^qconv16
Z^qconv17
X^qconv18
Y^qconv19
Z^qconv20
X
Z^(-qconv17)
Y^(-qconv16)



Σχήμα 5.26: Final block of Gates

The model that is used has the below form:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None,)]                 0
_________________________________________________________________
add_circuit (AddCircuit)     (None,)                   0
_________________________________________________________________
pqc (PQC)                    (None, 1)                 37
=================================================================
Total params: 37
Trainable params: 37
Non-trainable params: 0
_________________________________________________________________
```

Σχήμα 5.27: QCNN Model Summary

The results that we receive from this hybrid quantum-classic model are displayed below:

	accuracy	loss	val_accuracy	val_loss
0	0.738281	0.720991	0.769309	0.471293
1	0.765625	0.462365	0.834350	0.429513
2	0.814453	0.455481	0.769309	0.461561
3	0.796875	0.477502	0.822154	0.446415
4	0.775391	0.483203	0.822154	0.438809

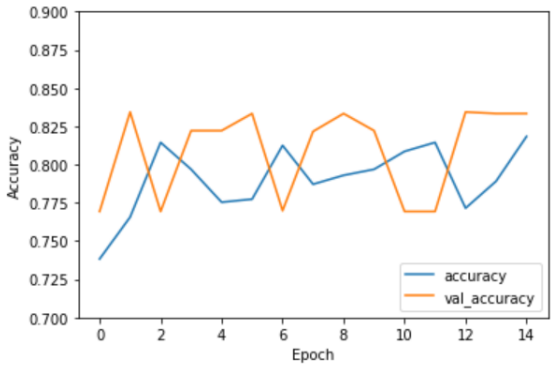
Σχήμα 5.28: Model Results: QCNN

As one can see from the statistics of the problem, we received a very high accuracy from the starting point of the process. We managed to reach maximum training accuracy of 81.445% and maximum validation accuracy of 83.435%. This hybrid quantum-classical model has been ran on a GPU with a run-time lasting 117.75 minutes with average epoch time being 7.85 minutes. For this case, we ran the model for 15 epochs, but with way less batch size than the classical case. We used a batch size equal to 16 with a learning rate, lr=0.01 on an Adam optimizer. We chose to train the dataset by splitting it to 80% training dataset and 20% test dataset and then by sampling a whole of 500 samples.

The first conclusion is that these results are much better than the classical case, so we reach immediate quantum advantage. This is due to the fact that we used a quantum circuit with large depth. By depth, we mean the number of layers used in the circuit with each layer in the quantum circuit substituting the classical multi-layers with simple quantum gates, which perform operations such as amplitude variance and rotation of the input vectors on the Bloch sphere. The reason I chose

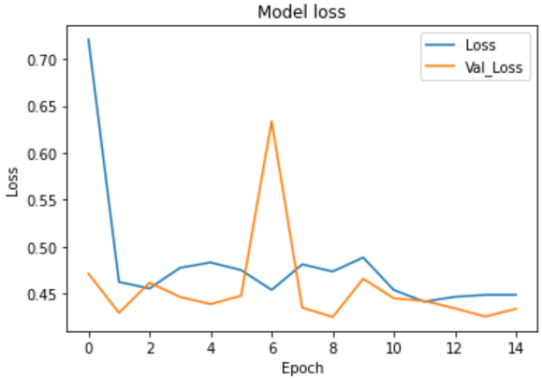
such a ”complex” circuit with great depth and various gates, is because as observed in the simple case of data encoding methods providing input to generic quantum variational classifiers, the performance of quantum models can be optimized by increasing the number of layers.

As one can observe in [5.29] the accuracy may not be increasing, but it is a good sign that the training and validation accuracies follow the same behavior.



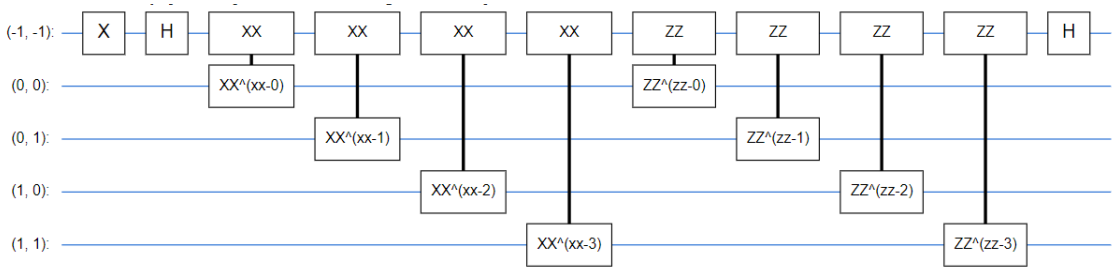
Σχήμα 5.29: Model Accuracy: QCNN

Secondly, concerning the loss of the hybrid quantum-classical model, we observe the expected behavior on both the training loss function and test loss function with only an outlier around 6 epochs.



Σχήμα 5.30: Model Loss: QCNN

Due to its performance, we wanted to try an additional implementation for the case of Quantum Convolutional Neural Networks, as it is an area with various usages and it is the basis to continue to more complex types of Neural Networks. In this



Σχήμα 5.31: Quantum Circuit with smaller depth

case, I choose the same strategy as in the previous quantum CNN model. I use the same dataset with 4 classes, but I binary encode my data, in order to display them in the bit representation. After achieving this transformation, we can study the binary classification problem that occurs. For this instance, we try a different quantum circuit, designed with different quantum gates and smaller depth. The current quantum circuit can be seen below: and the corresponding Neural Network, which is combined with the quantum circuit in order to train our model can be described as a simple PQC model that includes only 8 parameters [5.32].

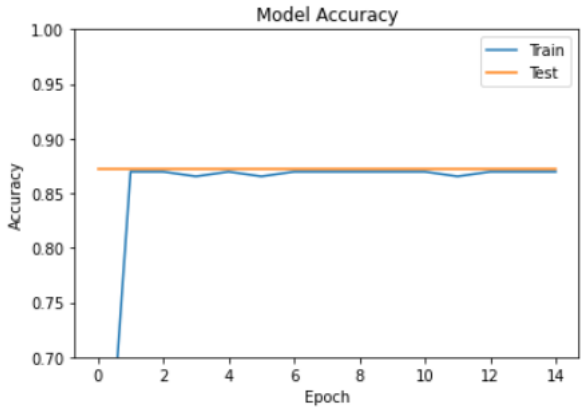
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
pqc (PQC)                    (None, 1)                 8
=================================================================
Total params: 8
Trainable params: 8
Non-trainable params: 0
_________________________________________________________________
```

Σχήμα 5.32: PQC model for QCNN

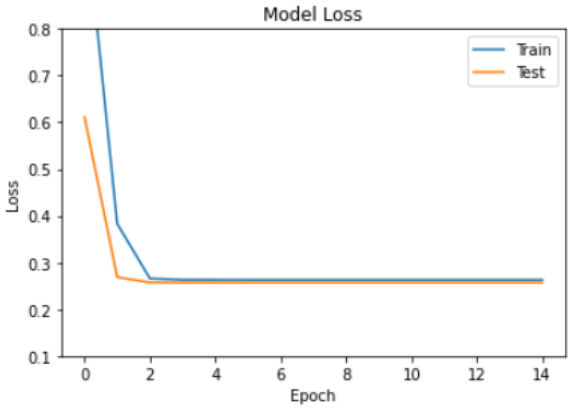
We manage to reach higher accuracy of 1-2% but our model is stationary and doesn’t continue to learn, as one can see.

```
Epoch 1/15
114/114 [==============================] - 8s 72ms/step - loss: 1.0676 - hinge_accuracy: 0.4423 - val_loss: 0.6116 - val_hinge_accuracy: 0.8716
Epoch 2/15
114/114 [==============================] - 9s 76ms/step - loss: 0.3829 - hinge_accuracy: 0.8697 - val_loss: 0.2685 - val_hinge_accuracy: 0.8716
Epoch 3/15
114/114 [==============================] - 8s 71ms/step - loss: 0.2658 - hinge_accuracy: 0.8697 - val_loss: 0.2572 - val_hinge_accuracy: 0.8716
Epoch 4/15
114/114 [==============================] - 9s 77ms/step - loss: 0.2630 - hinge_accuracy: 0.8654 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 5/15
114/114 [==============================] - 9s 77ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 6/15
114/114 [==============================] - 9s 75ms/step - loss: 0.2629 - hinge_accuracy: 0.8654 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 7/15
114/114 [==============================] - 8s 74ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 8/15
114/114 [==============================] - 9s 76ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 9/15
114/114 [==============================] - 9s 75ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 10/15
114/114 [==============================] - 8s 73ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 11/15
114/114 [==============================] - 8s 72ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 12/15
114/114 [==============================] - 8s 70ms/step - loss: 0.2629 - hinge_accuracy: 0.8654 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 13/15
114/114 [==============================] - 9s 82ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 14/15
114/114 [==============================] - 9s 78ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
Epoch 15/15
114/114 [==============================] - 9s 82ms/step - loss: 0.2629 - hinge_accuracy: 0.8697 - val_loss: 0.2569 - val_hinge_accuracy: 0.8716
```

Σχήμα 5.33: Smaller QCNN statistics



Σχήμα 5.34: Smaller QCNN - Model Accuracy



Σχήμα 5.35: Smaller QCNN - Model Loss

After evaluating our model we receive an average loss equal to 0.26355 and an average accuracy equal to 86.905%, which is a pretty good score. However, the high accuracy isn't the most unique aspect of this model. As one can observe in 5.33, our model ran for an average of 8 seconds per epoch and with 15 epochs, we got our results in just 2.125 minutes. Thus, in this case we may not have increased the model's accuracy compared to the previous implementation, however we achieved the quantum supremacy as our model ran in the 0.0185% of the previous model's run-time.

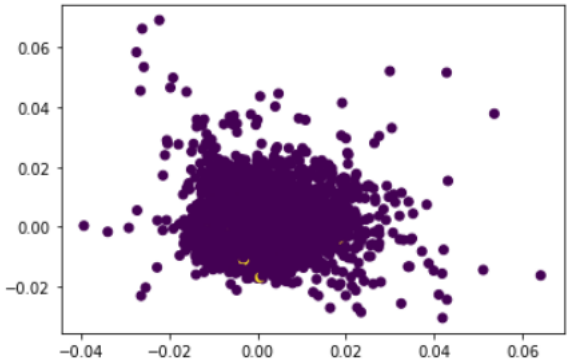
4. Quantum Generative Neural Networks (QGAN)

For this model I loaded the dataset of antidepressant drugs, which includes 19.5 thousand rows of SMILES strings. Then I exploit the RDKIT package, I convert the SMILES to molecule images that will be used as input for our quantum GAN. The images have been split to 4 classes, using the logP coefficient values as a criterion. I also use a Image Data Generator to enhance my input data and rescale all images, in order to be normalized to [0,1] space. The target size of images that are used has $(28, 28)$ dimensions and I use a batch size equal to 64 in the generator. Thus, I get a set of:

- train images of dimension: (10013, 28, 28, 4),
- train labels of dimension: (10013,4),
- test images of dimension: (3339, 28, 28, 4) and

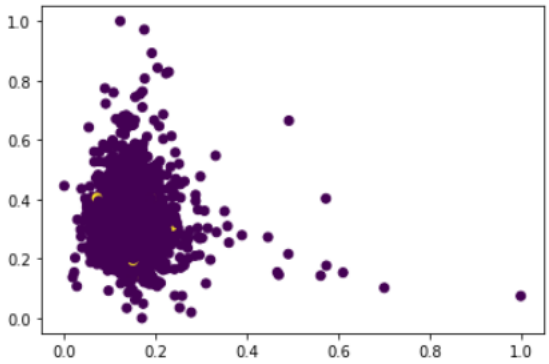
► test labels of dimension: (3339,4).

Then, I flatten the arrays and resize the images to (16,16) and I reshape my dataset to train images (10000, 256). To continue with, in order to reduce the problems dimensionality, I use the PCA algorithm with k=4. After this step, I receive as a result a vector which describes the Variance Ratio and is V.R. = [0.02082756, 0.01983622, 0.01914438, 0.01784568] and the respective Variance Ratio Cumulative = 0.07765384577214718. Then we obtain the scatter-plot in the vector space [5.36]:



Σχήμα 5.36: Resulting Vector Space, after PCA algorithm

and I normalize it to [0,1] range of values:



Σχήμα 5.37: Normalized Vector Space, to [0,1]

At this moment, I count the number of real data and is equal to 59 and then I extract the relevant data from the dataset.

Now, to continue with, I construct the quantum circuit using the ”default.qubit” device provided by the pennylane package. I construct 3 functions with the quantum gates, one to sample the generated data, the generator and the discriminator of the GAN model. Thus, in this case we choose to use a full-quantum model instead of a hybrid. Then I create a function of quantum gates to apply the discriminator on the real data and a function to apply the discriminator on the generated data. These are followed by a quantum function that calculates the probability of having real true results and the probability to receive fake true results. Finally, to conclude our implementation I construct a loss function for the generator and another one for the discriminator. In the first one I take as an argument the probability of getting a fake true result and in the second one I subtract the two probabilities:

$$disccost = Pr(fake - true) - Pr(real - true)$$

The equations of these probabilities can be expressed as:

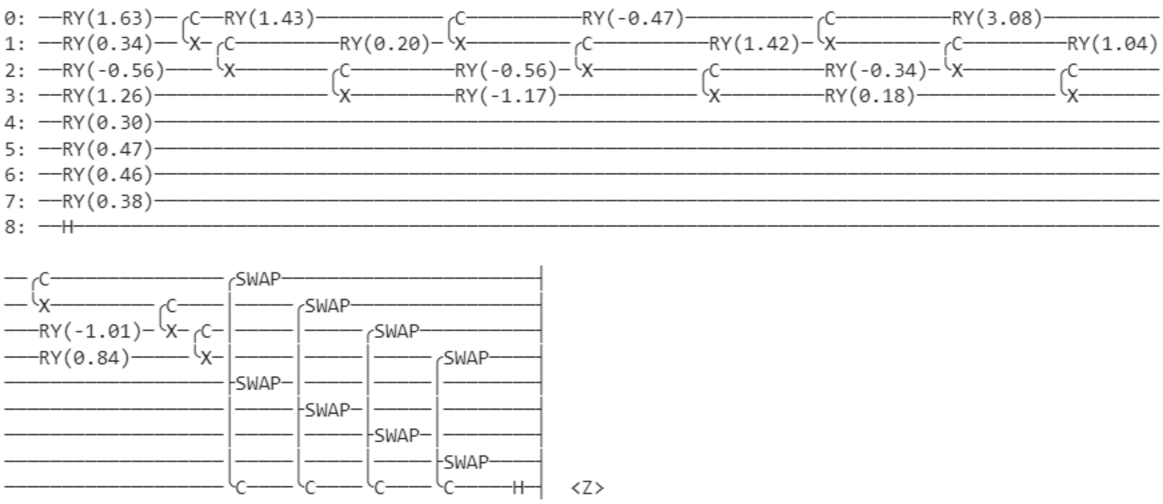
$$P_{fake} = abs(\frac{E(SWAPTEST(D, G)) - 0.5}{0.5} \tag{5.2}$$

and

$$P_{real} = abs(\frac{E(SWAPTEST(D, Real)) - 0.5}{0.5} \tag{5.3}$$

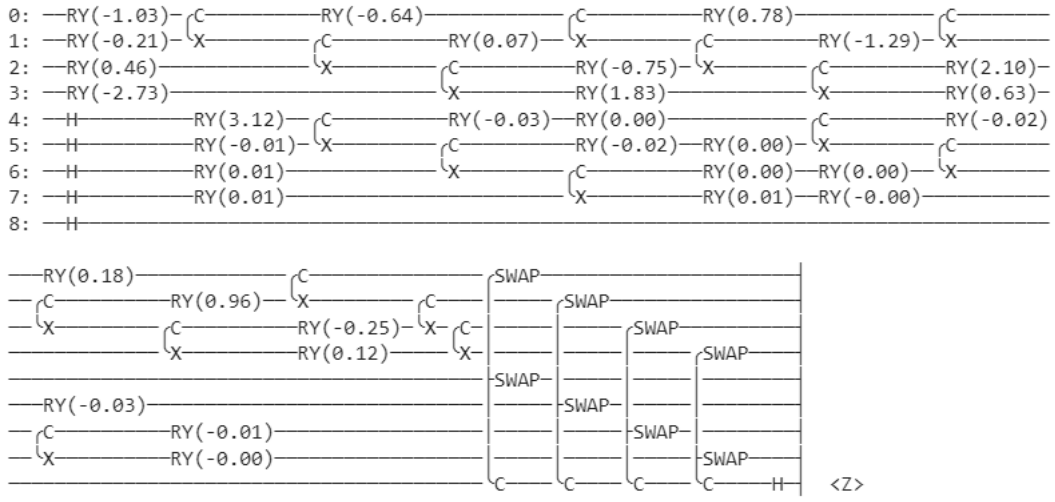
These can be used to measure the loss of the discriminator and the generator and can be used to update the defining parameters of the networks similarly to a classical GAN.

The real discriminator function can be visualized with the below circuit:



Σχήμα 5.38: Visualization of the real discriminator function

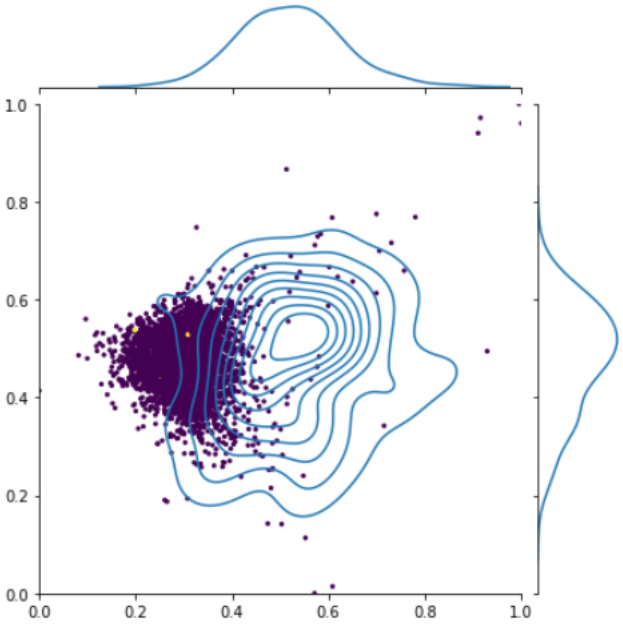
The discriminator for the generated data can be visualized in the below quantum circuit:



Σχήμα 5.39: Visualization of the discriminator function on generated images

This is one of the few models, in which I didn't use an Adam optimizer and instead I used a gradient descent optimizer with learning rate, lr=0.05. The result given in each epoch are of the below form:

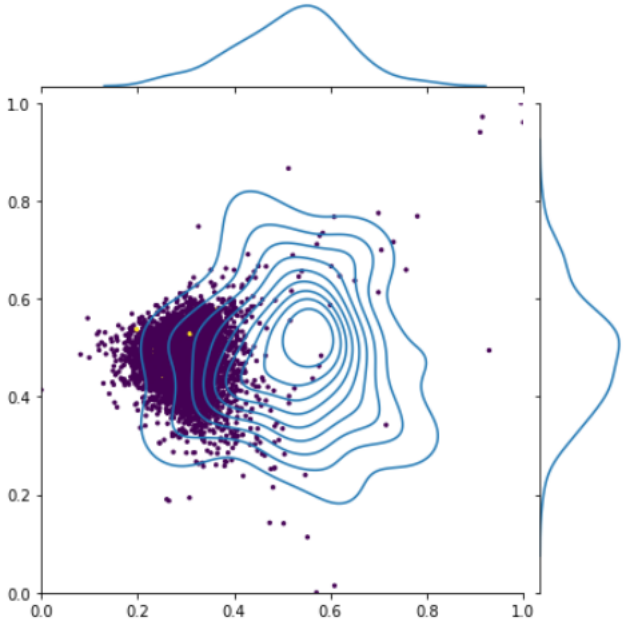
	Step #	**Cost**
Epoch 0	Step 1	-0.0500985666
	Step 16	-0.0501516775
	Step 31	-0.051600991791
	Step 46	-0.051554220037
Pr(Real = True) = 0.648361598057		
Pr(Gen = True) = 0.6032677431298		
Generator		
Epoch 0	Step 1	-0.6032677
	Step 16	-0.6032677
	Step 31	-0.6032677
	Step 46	-0.6032677
Discriminator cost = -0.05160991791		



Σχήμα 5.40: Vector space after classification

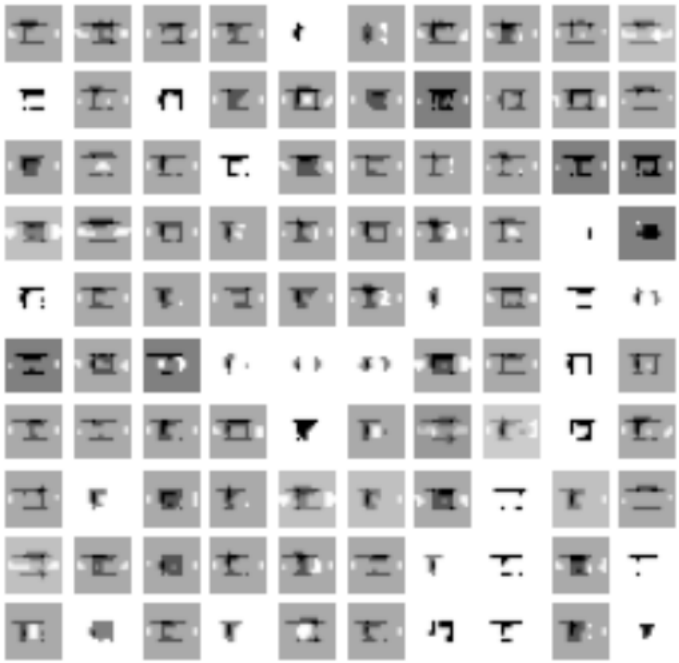
Thus, for this case we show a cost function for the discriminator and instead of the common model's metric, which is accuracy, we compare for each epoch the probabilities of the discriminator to predict a real true result on the real images and the probability of the discriminator to predict a fake true on a generated image. In the diagram [5.40], which is displayed above, we can see the normalized vector space and the data distribution after the trying to classify them in each epoch. The lines define the levels of confidentiality for calculating the probability that a point belongs to the wanted class. The model ran for 32 epochs on a GPU with a run-time of 4.582 minutes with an average of 8.591 seconds for each epoch. With this as a fact, we quickly understand why this model outperforms its classical analogous model. In the first part of the table, the discriminator cost function values are displayed and in the second section of the table, the generator function cost function is being calculated.

The final cumulative diagram of the vector space that the algorithm outcomes, is the below:



Σχήμα 5.41: Cumulative Vector space after training

And the average stage fidelity that we receive is $[0.51914769, 0.48085231]$. Below, I display in a grid form some of the generated images from our quantum model. They are blurry and have a lot of noise. However if we add many steps, they will become more clear. The problem of lack of time could not let us reach the stage, in which the images would have crystal-clear display.



Σχήμα 5.42: Grid of Generated Images from our Quantum GAN

In conclusion, I decided to use this type of architecture for my Quantum GAN running the discriminator and generator purely on quantum hardware and utilizing swap tests on qubits to calculate the value of loss functions. When using the same number of parameters, based on literature, this kind of model outperforms not only the classical GANs, but it also outperforms other quantum based GANs in the literature for up to 125% in terms of similarity between generated distributions and original data-sets. This architecture was based on the general QuGAN Design The architecture is fed our classical data, which is translated into quantum data. After the transformation, the parameters and quantum data are being used for the quantum discriminator and the quantum generator to learn from each other, as well as the quantum data.

SWAP tests are used in general to measure similarities in the quantum space. A SWAP test is a quantum algorithm that measures the difference between two quantum states. This algorithm requires the two qubits, whose similarity we will measure and a third qubit $|\Phi>$ to help in the measurement. In our model, the SWAP test is used to be able to accomplish communication between the two sub-circuits, the quantum Generator and the quantum Discriminator. The number of amplitudes that need to be computed are 2^n, (here n=4 qubits) and matrices describing the system or transformations reach sizes of $2^n \times 2^n$. Due to the last fact, the simulation of quantum circuits can quickly become infeasible and can become too costly to

simulate. This is also the reason, we cannot perform with very large number of epochs to produce better images.

Κεφάλαιο 6

Experimental Results - Conclusion

In this thesis, we tried to see in what level we can apply quantum algorithms for analyzing and predicting molecules from a large database, containing 193 thousand rows. However, we decided to focus our study to a subset of this dataset and see how we can classify molecules as compatible for use in the design of antidepressant drugs. Another case is classifying the antidepressant drugs to 4 classes based on the octanol-water partition coefficient, logP coefficient, of each molecule. We used a strategic preprocessing flow to use different inputs to suit better the needs of each model, in order to reach the best performance for each model. We used matrices, images, graphs and feature vectors as input spaces for our models. For the classical computing models we used the for-mentioned inputs to apply study both binary and multi-class classification in some cases. However, in most of the Quantum Machine Learning Models we were ”forced” to study the binary classification case. This happened due to the way we formed our input space. During our preprocessing flow, before training our models, we transformed our data to bit representation to be able consequently to get transferred to the qubit representation from there. Thus, after having encoded our data to these specific quantum states we tried to classify our data using this representation.

Before analysing the rest of the models it is important to mention as a disclaimer that the different case and the different types of model weren't approached in the same way, neither we presented the results in an identical representation. That is because we wanted to study each case individually first and then compare the results. It's case is a different type of problem and thus it need a different strategy to approach it. As we tried figuring out how to encode our input data to our quantum circuits, we compared two quantum embedding techniques. Both underperformed for our dataset and failed as they reached a maximum of 57% accuracy. We believe that

this problem could be surpassed with most powerful devices than devices and a method that could be studied more and that could potentially provide better results could be the Dense Angle encoding method for this case. This is due to the fact that we believed that the angle encoding method performed better than the amplitude encoding method and because it suits better the specific dataset. This is due to the fact that our data include geometrical parameters and have symmetries that could match the angle encoding method.

Using as a fact the above bit/qubit representation we see the results of each model and we concluded to a first level comparison of the models and defining the quantum advantage achieved in each case. First, I see the results of the Convolutional Neural Network with two Quantum Convolutional models [6].

	Accuracy	Runtime
CNN	74.25%	113.33 minutes
QCNN 1	83.435%	117.75 minutes
QCNN 2	86.905%	2.125 minutes

Thus, we can observe that for both cases there is a quantum advantage, either concerning the accuracy of the model or the runtime. We see that the difference isn't very large between the CNN and the first QCNN model, approximately 10%, but as we see with the second quantum model, we can construct a quantum CNN that can outperform the classical model both in accuracy and have the 0.1% of the its runtime duration.

Now concerning the Graph Convolutional model, we managed to study only the classical computing case, as its quantum analogous is still in a preliminary theoretical state and it is computationally and timely very costly to more applied cases. However, we observed that the Graph Convolutional Network outperformed the CNN model, by reaching an accuracy equal to 88.352% in the same time.

Finally, concerning the Generative Adversarial Networks, there are various ways to approach this study. However, I chose to approach the Quantum GAN with SWAP test gates and a fully-quantum model. We could use a hybrid model too, but we saw that if we encode the input data with the right way, a fully-quantum circuit performs better and if we have the computational power, it outperforms both the hybrid and classical computing method. We see that the quantum Gan runs for 32 epochs in a good runtime of 4.582 minutes. When using the same number of parameters, based on literature, this kind of model outperforms not only the

classical GANs, but it also outperforms other quantum based GANs in the literature for up to 125% in terms of similarity between generated distributions and original data-sets. This quantum advantage was observed in a smaller level in our case by achieving the probabilities Pr(Real = True) = 0.648361598057 and Pr(Gen = True) = 0.6032677431298.

Thus, we understand that quantum computers is an emerging field that has a lot to offer. However, the hardware is still underdeveloped and we can partially exploit the quantum advantage for now. All theoretical and computational studies, along with this own, show that if we manage to construct the suitable devices, we will unlock a great potential with Quantum Computing. So this is an encouraging thought to continue researching this field for a trip to reducing the complexity of our problems.

Κεφάλαιο 7

Bibliography

1. ”Convolutional Networks on Graphs for Learning Molecular Fingerprints”, D. Duvenaud, D. Maclaurin, J.A.I. Bombarelli, T. Hirzel, A.A Guzik, R.P. Adams, 2015,

2. H.L. Molgan. ’The generation of a unique machine description for chemical structure. Journal of Chemistry Documentation, 5(2):107-113, 1965,

3. RDKit: Open-Source cheminformatics. www.rdkit.org [11 April 2013],

4. D.Weininger, SMILES, a chemical language and information system. Journal of chemical information and computer sciences, 28(1):31-36,1988,

5. Thomas Unterthiner, Andreas Mayr, G ¨unter Klambauer, Marvin Steijaert, J¨org Wenger, Hugo Ceulemans, and Sepp Hochreiter. ”Deep learning as an opportunity in virtual screening”. Advances in Neural Information Processing Systems, 2014,

6. ”Semi-Supervised Classification with Graph Convolutional Networks.”, T.N. Kipf, M.Welling, published as a conference paper at ICLR 2017,

7. ”Quantum autoencoders for efficient compression of quantum data”, J.Romero, J.P.Olson, A.A. Guzik., (2017),

8. ”Quantum Variational Autoencoder”, A.Khoshaman, W. Vinci, B.Denis, E. Andriyash, H. Sadeghi, M.H. Amin,

9. ”Quanvolutional Neural Networks: Powring Image Recognition with Quantum Circuits, M. Henderson, S. Shakya, S.Pradhan, T.Cook, (2019),

10. Quantum Graph Neural Networks, G.Verdon, T. McCourt, (2019),

11. ”Generative chemistry: drug discovery with deep learning generative models”, Y.Bian, X. Xie,

12. ”Quantum Generative Models for Small Molecule Drug Discovery”, J. Li, R. Topaloglou, S. Ghosh, (2021),

13. ”Invited: Drug Discovery Approaches using Quantum Machine Learning”, J. Li, M. Alam, C. M. Sha, J. W. N. V. Dokholyan, S. Ghosh, (2021),

14. ”Robust data encodings for quantum classifiers”, R. LaRose, B. Coyle, (2020),

15. ”Quantum Machine Learning Algorithms for Drug Discovery Applications”, K. Batra, K.Z.Zorn, D.H. Foil, E. Minerali, V.O. Gawriljuk, T.R. Lane, S. Ekins, J. Chem.Inf.Model. 2021, 61, 2641-2647,

16. ”Convolution filter embedded quantum gate autoencoder”, K.Shiba, K. Sakatomo, K. Yamaguchi, D.B. Mall, T. Sogabe,

17. G.E. Hinton and R.R. Salakhutdinov,

18. ”Reducing the Dimensionality of Data with Neural Networks, G.E.Hinton, R.R. Salskhutdinov, Science 313 (5786), pp.504-507, 2006,

19. ”A Comprehensive Survey on Graph Neural Networks, Z. Wu, S.Pan, F. Chen, G. Long, C. Zhang, P.S. Yu,

20. ”Machine Learning Prediction of Nine Molecular Properties Based on the SMILES Representation of the QM9 Quantum-Chemistry Dataset”, G. A. Pinheiro, J. Mucelini, M.D. Soares, R.C. Prati, J. L. F. Da Silva, M. G. Quiles, J.Phys.Chem. A 2020, 124,47,9854-9866,

21. Yann LeCun et al., 1998, Gradient-Based Learning Applied to Document Recognition

22. Adit Deshpande, 2016, The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3)

23. C.-C. Jay Kuo, 2016, Understanding Convolutional Neural Networks with A Mathematical Model,

24. Qiskit documentation by IBM, https://qiskit.org/documentation/,

25. ”Quantum Graph Convolutional Neural Networks”, J. Zheng, Q. Gao, Q. Lv,

26. ”Encoding classical data into a quantum computer”, K.J.B. Ghosh, (2021),

27. ”Hybrid quantum classical graph neural networks for particle track reconstruction”, C.Tuysuz, C. Rieger, K. Novotny, B. Demirkoz, D. Dobos, K. Potamianos, S. Vallecorsa, J.-R. Vlimant, R. Forster, Quantum Machine Intelligence 3, 29 (2021),

28. ”A Gentle Introduction to Graph Neural Networks”, Sanchez-Lengeling, B. Reif, E. Pearce, A. Wiltcschlko,

29. ”Neural Message Passing for Quantum Chemisrty”, Gilmer J., Schoenholz S.S., 2017, Proceedings of the 34th International Conference on Machine Learning, vol.48, pp..1263-1272,

30. ”Learning Convolutional Neural Networks for Graphs”, Niepert M., Ahmed M., S.S. Riley, P.F. Vinalys, 2017, Proceedings of the 33rd International Coference on International Conference on Machine Learning, Vol.48, pp.2014-2023,

31. ”How Powerful are Graph Neural Networks?”, Xu K., Hu W., Leskovec J., 2017, Advances in Neural Information Processing Systems, Vol 30, pp.1024-1034, Curran Associates,

32. https://quantumalgorithms.org/chap-classical-data-quantum-computers.html

33. https://medium.com/mlearning-ai/quantum-data-and-its-embeddings-1-3b022b2f1245

34. ”Dataset’s chemical diversity limits the generalizability of machine learning predictions”, M.Glavatskikh, J. Leguy, G.Hunault, T. Cauchy, B. Da Mota, Journal of Cheminformatics 11, 69, 2019

35. QM9 documentation,

36. Supporting Information for ”Physically inspired deep learning of molecular excitations and photoemission spectra”, J. Westermayr, R.J. Maurer, The Royal Society of Chemistry, 2021,

37. ”Early phase drug discovery: Cheminformatics and computational techniques in identifying lead series”, C. Bryan, Z. Lei, H. Decornez, D.B. Kitchen, Bioorganic Medicinal Chemistry, vol.20, 18,2012, 5324-5342,

38. ”Machine Learning in cheminformatics and drug discovery”, Y.C. Lo, S.E. Rensi, R.B. Altman

39. https://pubchem.ncbi.nlm.nih.gov/,

40. ”Autoencoding Undirected Molecular Graphs with Neural Networks”, J.J. W. Olsen, P.E. Christensen, M.H. Hansen, A.R. Johansen, (2020),

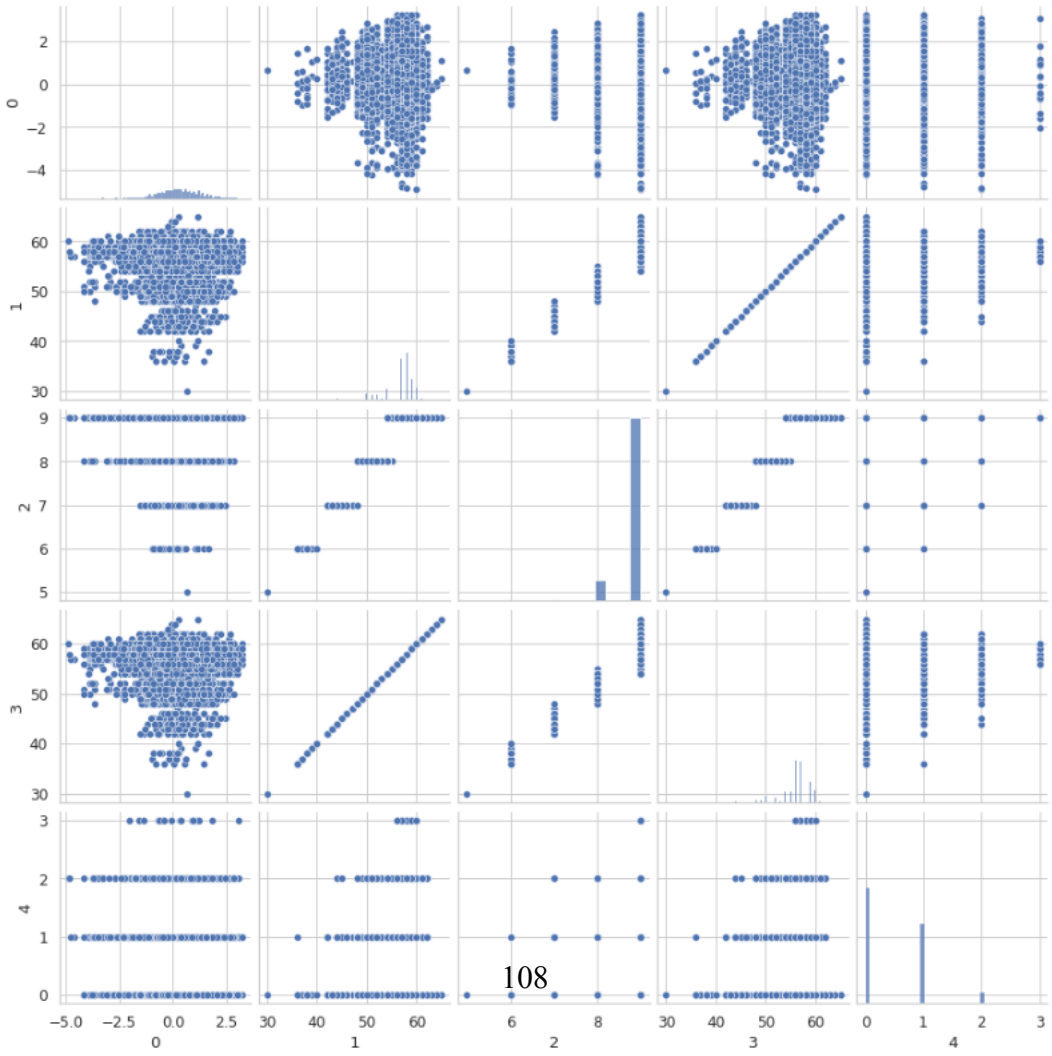
41. Circuit-centic quantum classifiers, M. Schuld, A. Bocharov, K. Svore, N. Wiebe,

42. ”Classification with Quantum Neural Networks on Near Term Processors”, E. Fahri, H. Neven.

Κεφάλαιο 8

Appendix

1. Dataset Pairplot before Encoding



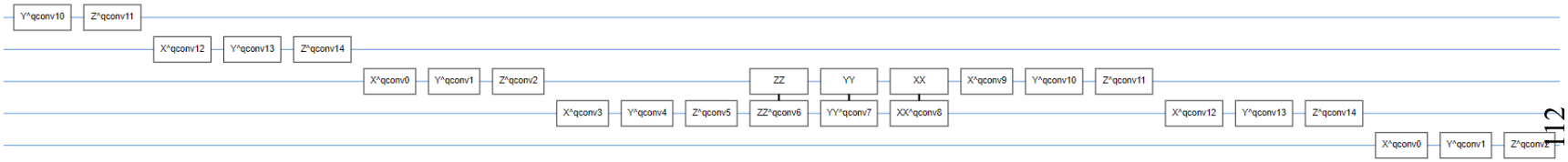
Σχήμα 8.1: Dataset from the chapter with the Encoding techniques

2. QCNN - Quantum Circuit

The block used twice in the quantum circuit that is described in the section of the Quantum Convolutional NN, can be find in the below 6 images.

(0, 0):
X^qconv0
Y^qconv1
Z^qconv2
(0, 1):
X^qconv3
Y^qconv4
Z^qconv5
ZZ^qconv6
YY^qconv7
XX^qconv8
X^qconv9
Y^qconv10
Z^qconv11
(0, 2):
X^qconv12
Y^qconv13
Z^qconv14
(0, 3):
X^qconv0
Y^qconv1
Z^qconv2
(1, 0):
X^qconv3
Y^qconv4
Z^qconv5
ZZ^qconv6
(1, 1):
ZZ
(1, 2):
YY
(1, 3):
XX
(2, 0):
(2, 1):
(2, 2):
(2, 3):
(3, 0):
(3, 1):
(3, 2):
(3, 3):

Y^qconv4
Z^qconv5
ZZ^qconv6
YY^qconv7
XX^qconv8
X^qconv9
Y^qconv10
Z^qconv11
X^qconv12
Y^qconv13
Z^qconv14
X^qconv0
Y^qconv1
Z^qconv2
X^qconv3
Y^qconv4
Z^qconv5
ZZ^qconv6
YY^qconv7
XX^qconv8
ZZ
YY
XX
X^qconv9
Y^qconv10


Y^qconv10
Z^qconv11
X^qconv12
Y^qconv13
Z^qconv14
X^qconv0
Y^qconv1
Z^qconv2
X^qconv3
Y^qconv4
Z^qconv5
ZZ
ZZ^qconv6
YY
YY^qconv7
XX
XX^qconv8
X^qconv9
Y^qconv10
Z^qconv11
X^qconv12
Y^qconv13
Z^qconv14
X^qconv0
Y^qconv1
Z^qconv2

X'qconv0
Y'qconv1
Z'qconv2
X'qconv3
Y'qconv4
Z'qconv5
ZZ'qconv6
ZZ
YY'qconv7
YY
XX'qconv8
XX
X'qconv9
Y'qconv10
Z'qconv11
X'qconv12
Y'qconv13
Z'qconv14
X'qconv0
Y'qconv1
Z'qconv2
X'qconv3
Y'qconv4
Z'qconv5
ZZ'qconv6
ZZ
YY'qconv7
YY

XX
X'qconv9
Y'qconv10
Z'qconv11
XX'qconv8
X'qconv12
Y'qconv13
Z'qconv14
X'qconv0
Y'qconv1
Z'qconv2
X'qconv3
Y'qconv4
Z'qconv5
ZZ'qconv6
ZZ
YY'qconv7
YY
XX'qconv8
XX
X'qconv9
Y'qconv10
Z'qconv11
X'qconv12
Y'qconv13
Z'qconv14

XX
X'qconv9
Y'qconv10
Z'qconv11
XX'qconv8
X'qconv12
Y'qconv13
Z'qconv14
X'qconv0
Y'qconv1
Z'qconv2
X'qconv3
Y'qconv4
Z'qconv5
ZZ'qconv6
ZZ
YY'qconv7
YY
XX'qconv8
XX
X'qconv9
Y'qconv10
Z'qconv11
X'qconv12
Y'qconv13
Z'qconv14