Regularisation in QCBM based generative models
Applied Quantum Algorithms Project Report

Yorgos Sotiropoulos

May 23, 2021

Contents

1 Introduction

The rise of quantum computing has given hopes in dealing with problems that classical computers cannot tackle in reasonable time-frames. One of the many applications that are being investigated lies within the regime of machine learning. Discriminative and generative models can both take advantage of the properties of quantum machines to achieve higher performance.

In this work we will be using Parameterised Quantum Circuits as generative models. In particular, we will be using Quantum Circuit Born Machines that will be trained in order to generate samples from a target distribution.

Dropout will be used as a regularisation technique and the regime of hyperparameters will be such that is not over-fitting friendly. We shall benchmark all of the models using Kullback-Leibler divergence and the relative gain when the model is regularised versus when it is not.

In section 2 we present the theoretical aspects of our study and in section 3 we shall apply them to two cases. In section 4 we draw our conclusions.

2 Theoretical Aspects

2.1 PQC and QCBMs

A parameterised quantum circuit (PQC) is a quantum circuit that consinsts of parametric single qubit gates and entangling gates. It is divided into layers that have the same structure and each layer is a fully parameterised unitary that can express a multitude of transformations. To realise this unitary, each layer is comprised of an arbitrary rotation for each of the qubits and entangling gates that entangle all qubits between them.

In this paper, one layer will be comprised of a an entangling layer as in figure 1 and the arbitrary single-qubit rotations will be generated by a sequence of two parameterised rotations over the z and x axes i.e. $R_z(\theta_i)$ and $R_x(\theta_i)$. It becomes immediately apparent that the dimension of the parameter space will be:

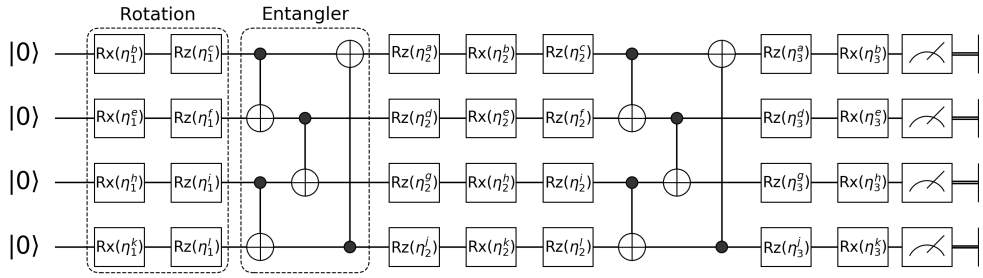


Figure 1: A Parameterised Quantum Circuit of size 4 and depth 2[8]

$$dim(\boldsymbol{\theta}) = 2 \cdot \#\,\text{qubits} \cdot \text{depth} \tag{1}$$

A Quantum Circuit Born Machine is a PQC whose output distribution is treated as a generative model. The parameters of course determine the output distribution which can be obtained by calculating the output statevector. Sampling from the distribution is done by just measuring the circuit.

In our code, the implementation of the PQC is noiseless and is being done by use of the Python library Cirq[3].

2.2 Loss function and Gradient

Using classical optimization methods, we can define a loss function and use gradient descent in order to train a QCBM. In this work, the loss function we shall be using is the the squared maximum mean discrepancy (MMD) which can be written as [5]:

$$L(p,q) = E_{x\sim p, y\sim p}\left[K(x,y)\right] - 2E_{x\sim p, y\sim q}\left[K(x,y)\right] + E_{x\sim q, y\sim q}\left[K(x,y)\right] \tag{2}$$

where K is the kernel which is selected as a mixture of Gaussians:

$$K(x,y) = \frac{1}{c}\sum_{i=1}^{c} \exp\left(-\frac{1}{2\sigma_i}|x-y|^2\right) \tag{3}$$

where σ_i are bandwidth parameters that control the width of the Gaussians. These are selected based on the representation but we shall select them as: $\sigma = \{0.25, 4\}$.

The gradient of this loss function[5] is:

$$\frac{\partial \mathcal{L}}{\partial \theta_l^\alpha} = \underset{x\sim p_{\theta+}, y\sim p_\theta}{E}[K(x,y)] - \underset{x\sim p_{\theta-}, y\sim p_\theta}{E}[K(x,y)] - \underset{x\sim p_{\theta+}, y\sim \pi}{E}[K(x,y)] + \underset{x\sim p_{\theta-}, y\sim \pi}{E}[K(x,y)] \tag{4}$$

where $p_{\theta\pm}$ correspond to output probabilities when every parameter θ_j is mapped to $\theta_j \pm \pi/2$.

As an optimization algorithm we will be using SciPy's[9] implementation of the Limited-memory BFGS method.

2.3 Regularisation as an over-fitting precaution

One of the most notorious pitfalls in machine learning goes by the name of "over-fitting". Simply put, this refers to a trained model that although the minimization of a loss function was achieved in a training sample, it does not generalise well. [1] Over-fitting occurs when our model does not account for potential noise in the training sample and thus performs poorly on the test data set. In the frame of generative models, this would mean that the model simply imitates the training sample distribution instead of unraveling the underlying patterns of the distribution.

Regularisation is a common way to tackle this problem. Regularisation aims to put a bias in the training process that punishes complexity and as a result will not allow the generative model to be too flexible and adapt too much to our training data. Here we shall present two ways this can be achieved:

Modifying the Loss Function: Adding a term that punishes complexity to the loss function is one way. This can be achieved by adding a term proportional to the sum of the absolute values (*L1 regularisation*) or squares (*L2 regularisation*) of the parameters[6]. L1 regularisation should be chosen when the parametric space is too large since it tends to shrink the least important parameters. We can transform the loss function as:

$$L \longrightarrow \begin{cases} L + \lambda \sum \theta_i^2 & \text{L2 regularisation} \\ L + \lambda \sum |\theta_i| & \text{L1 regularisation} \end{cases} \tag{5}$$

Note that λ should be optimized depending on the case. Also, if the gradient is calculated analytically, the regularisation term should also be accounted for.

Dropout: A regularisation technique often used in neural networks is *dropout*.[7] In the frame of neural networks this is equivalent to neutralising some nodes of the network. In our case, it will be setting some parameters equal to zero. This however has to be done with extra caution. Dropping out parameters as a post routine for example could lead to the model losing some parameters that were really vital to it. A procedure that allows for good re-adaptation of the model can be described as such:

1. Train the model for some time
2. Dropout a percentage of the smallest parameters
3. Train the model for some more time

Note that for a QCBMs the parameters are angles and should be cast on the interval $[-\pi, \pi]$ before determining whether its absolute value is small compared to other parameters.

The selection of the above time-frames and the dropout rate has to be optimized accordingly as well. Having a small dropout rate means that we barely affect the training process. Conversely, setting this ratio too high could mean that the model is not able to re-adapt to the data set and suffers thus from under-training.

In this work we will only be using dropout.

2.4 Benchmarking measures

To detect the effects of regularisation, comparing the generative model with and without regularisation is the main priority. To this end, we shall be using the *Kullback–Leibler divergence* also known as *relative entropy*.[1] This is a measure of proximity between two distributions and for discrete distributions it is defined as:

$$D_{\mathrm{KL}}(T \| P) = -\sum_{x \in \mathcal{X}} T(x) \log \left(\frac{P(x)}{T(x)} \right) \tag{6}$$

where in our case $T(x)$ is the test data set distribution and $P(x)$ is the trained generative model distribution. The lower the value of the relative entropy, the closer our generative model distribution is to the test data set and thus, the better the performance.

For the purposes of this work, we shall also introduce a comparative measure which we shall be referring to as *gain by regularisation* or simply *gain*. This will be defined as:

$$G := \frac{D_{\mathrm{KL}}^{N.R} - D_{\mathrm{KL}}^{R}}{D_{\mathrm{KL}}^{N.R}} \tag{7}$$

where $D_{\mathrm{KL}}^{(N.)R}$ is the (non) regularised model relative entropy.

This will encapsulate the effect of regularisation as a ratio. Positive values of this measure will mean that regularisation has positive effects for our model.

3 Applications

3.1 Hyperparameter selection

Before experimenting with the models, we should first discuss the peculiarities that stem from our resource-lacking implementations. In particular, because our model is based on a QCBM that is simulated by a classical computer, we need to make sure that every hyperparameter is on the right scale so as to avoid trivial cases or the need for too computationally expensive paradigms.

Optimizing a large number of parameters can quickly get out hand. Meanwhile, our QCBM should be relatively deep in order for it to be able to encapsulate the behavior of the target distribution. Although *deep* is vaguely defined, intuitively we can understand that the ratio of layers to qubits should be rather large. As a result, putting more qubits to the system means that we should also at least proportionally increase the layers. This not only increases the QCBM simulation expenses but most importantly increases quadratically the parameters to be optimized.

For this reason, we shall be working almost exclusively with 3 qubits and a depth of 12 layers. This brings the number of parameters to be optimized to 72 according to equation 1. This selection also allows us to narrow down our investigation and focus on other dependencies of our model.

Another very important consideration one has to make when approaching this problem has to do with the number of shots that one allocates to the PQC when this is queried. During training, the PQC is queried $2 * dim(\boldsymbol{\theta})$ for the gradient per iteration and when the training is completed, it has to be queried again in order to obtain the final generated distribution. Since we would like to eliminate fluctuations due to statistical error in the training process so as to estimate the average effect of regularisation , we shall be using the final statevector simulation rather than finite-shot sampling. Liu and Wang showcase in their paper [5] that this also means that this allows the loss function to become really small because of the gradient can be calculated exactly without statistical uncertainty.

Lastly, the optimization shall be done as such:

- Unregularised QCBM
  - 20 iterations of optimization
- Regularised QCBM
  - 5 iterations of optimization
  - dropout of a percentage of smallest parameters
  - 15 iterations of optimization

3.2 Benchmarking in the hyperparameter regime of choice

This simplistic approach of modelling that we have chosen does not come without pitfalls. In particular, the main issue arises from the really limited dimension of the distribution we finally obtain. Measuring a circuit of 3 qubits can give 2^3 different discrete values. Fluctuations due to random sampling of the training data set cannot be cancelled out on average due to this. As a result, by chance one can obtain misleading results and deem incorrectly that some regularisation (does not) help the model.

To this end, we shall be benchmarking the effect of regularisation with a measure which we shall be referring to as *average gain* and denote as $< G >$. This will be the average gain with the same selection of hyperparameters but over random seeds that are fed into NumPy[4]. To avoid bias, these seeds are taken iteratively from 0 with step 1. In each case study, we shall examine how this randomness affects the training and test data set.

The standard error of this average will be estimated by the standard error of the mean:

$$\sigma_{<G>_n} = \frac{\sigma_G}{\sqrt{n}} \tag{8}$$

where σ_G is the standard deviation of the G population and n is the number of different random seeds.

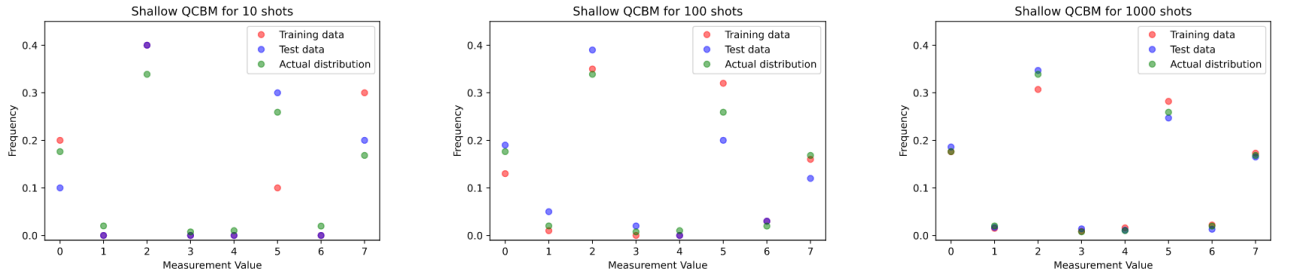
3.3 Shallow QCBM as target distribution

In this instance, the training and the test data set will be generated by a shallow QCBM. This QCBM will not be trained but will rather be initialized with random values as parameters. This means that the shallow QCBM will be a generator of a random distribution.

Since we have already selected our ansatz to be comprised of 3 qubits we shall use a shallow QCBM of 3 qubits so as to have a complete 1-to-1 mapping between the values that can be sampled between the two distributions. The depth of the shallow QCBM will be selected as 3. It is important that this depth is less than the depth of our ansatz since violating this could potentially mean that the target distribution cannot be expressed by our ansatz.

Another choice we have to make has to do with the sampling for the training. A large amount of shots would mean that the training and the test data set are really close to the actual distrbution and as such over-fitting will not be a major issue since the QCBM will be eventually mimicing almost the actual distribution. This can

also be seen in figure 2. Since we would like to showcase regularisation improving performance at some regime, we select a low number of shots that is representative of the sample but blindly over-fitting will not yield good results. Thus, the number of shots for the training data will be set to 100. The test data set will be the actual distribution since we have access to it.



(a) Shallow QCBM sampling for 10 shots

(b) Shallow QCBM sampling for 100 shots

(c) Shallow QCBM sampling for 1000 shots

Figure 2: Shallow QCBM sampling for different number of shots

As discussed earlier, the process shall be iterated for different random seeds. The random seed will in this case fully determine the output distribution of the QCBM as well as the random sampling from it for the training data. Although we could have sampled for the test data as well, we expect on average that the actual distribution will yield the same behavior.

The only variable that can now be swept is the dropout rate. To find how this affects performance, we iterate through 20 different random seeds for the interval $[0.1, 0.7]$. In figure 3 the results are plotted.

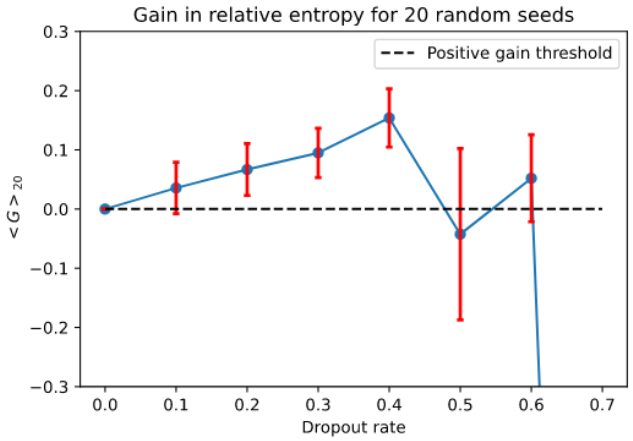


Figure 3: Average gain in relative entropy over 20 different random seeds for different dropout rates.

Although we have to be careful due to the low number of seeds taken, we can clearly see that increasing the dropout rate has a positive effect for our generative model up to some point. After 0.4 it is apparent that the gain undergoes big fluctuations as observed in the error-bar for dropout rate of 0.5. Quite notably, for a dropout rate of 0.7 the deep QCBM is no longer able to readapt and performs really poorly (-300%).

To evaluate the performance at this detected optimal value of 0.4 dropout rate, we ran it for 100 random seeds. The result that we obtained was.

$$< G^{0.4} >_{100} = (9.8 \pm 2.1)\% \tag{9}$$

We conclude thus that this value of dropout rate is indeed enhancing the performance of our generative model in this hyperparameter regime.

3.4 Wine generating QCBM

The reader should be advised that although the title of this sub-section is technically true, it is heavily misleading. Using a data set of white wines [2] we shall create a simplistic model of wines that will be casted

onto our ansatz distribution space. In particular, even though the data set contains 12 different attributes that each can attain a value within a range, we will only be working with three attributes that will in turn be divided into two bins each. The attributes we shall be working with are given in table 1 with their classification.

Attribute	Quality	Alcohol	pH
Classification #1	**B**ad	**L**ight	**A**cidic
Range #1	$[3, 6)$	$[8, 11)$	$[2.72, 3.3)$
Classification #2	**G**ood	**H**eavy	**M**ild
Range #2	$[6, 9]$	$[11, 14.2]$	$[3.3, 3.82]$

Table 1: Classification of the wine data set

For completeness, in figure 4 we present the distribution of the full 5000 wine data set.

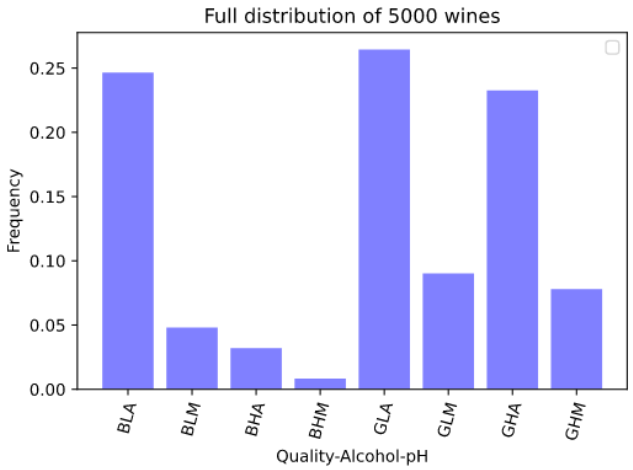


Figure 4: Total distribution of wine data set based on classification 1

The role of this generative model might seem quite absurd at first due to its aforementioned simplistic nature. The generated "wine" is a combination of three letters corresponding to an attribute each. Had we been able to include more attributes with better discretisation for each, the generated "wine" would be better defined. This is computationally impossible however for a classical computer simulating quantum circuits. In general, including n attributes in m bins each requires $\log_2(m^n)$ qubits. For example, if we included all 14 attributes with 8 bins each, we would need 42 qubits.

We can clearly see that the problem we are dealing with is the same as in the shallow QCBM example as the distribution has been casted on a subspace of dimension 2^3. The only selection that we need to do is the size of the training and test data set.

Again, if we were to take half of the full data set as a training sample and half as a test sample, these two will be quite close and as such over-fitting will not be a major issue. For this reason, we will only take a small part of the total data set (300 out of 5000) and divide into the training and data sets with a ratio of 3:2. Before slicing the data set for each random seed, our full data is randomly shuffled and thus we are able to generate different random samples from our total data set.

As far as the dropout rate is concerned, we shall go again with the value of 0.4 that we found earlier to be effective. Since the data set we are using here is of the same magnitude as in the shallow QCBM case, we do not expect our results to be that different. Indeed, running this setup for 100 different random seeds yielded:

$$< G^{0.4} >_{100} = (9.2 \pm 1.7)\% \tag{10}$$

which is quite close to the value found earlier

Thus, we see that by dropping out 0.4 of the smallest variables in this hyperparameter regime improves our generative model for the wine data set as well.

4 Conclusion

In this work, we presented the QCBM as a generative model that can be optimized via classical gradient descent methods. We gave an overview of two regularisation techniques used to prevent over-fitting: loss function modification and dropout. We used the latter to showcase that it can indeed improve the performance of the generative model in some hyperparameter regime by introducing the quantity that we called *gain*. This improvement of performance was consistent and present in two examples of target distributions: the output of a shallow randomly intialized QCBM and a real wine data set that was simplified, classified and casted on the subspace of the ansatz.

As a future outlook, this work can be expanded in multitude of ways. By relaxing the low sampling rates for the training sets that we imposed one could find the new optimal dropout rate. Experimenting with more qubits and deeper QCBMs with better classification of classical data would head towards a generalisation of the findings here. Another avenue to explore concerns the implications of using finite number of shots during training. Lastly, all of the above can be studied with different methods of regularisation, other than dropout.

References

[1] *Applied Quantum Algorithms Lecture notes.*

[2] P. Cortez et al. "Modeling wine preferences by data mining from physicochemical properties". In: *Decis. Support Syst.* 47 (2009), pp. 547–553.

[3] Cirq Developers. *Cirq*. Version v0.11.0. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/c May 2021. DOI: 10.5281/zenodo.4750446. URL: https://doi.org/10.5281/zenodo.4750446.

[4] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] Jin-Guo Liu and Lei Wang. "Differentiable learning of quantum circuit Born machines". In: *Physical Review A* 98.6 (Dec. 2018). ISSN: 2469-9934. DOI: 10.1103/physreva.98.062324. URL: http://dx.doi.org/10.1103/PhysRevA.98.062324.

[6] Andrew Y. Ng. "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 78. ISBN: 1581138385. DOI: 10.1145/1015330.1015435. URL: https://doi.org/10.1145/1015330.1015435.

[7] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[8] *Tutorial on QCBM*. URL: https://docs.yaoquantum.org/v0.1/tutorial/QCBM/.

[9] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.