GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report for Scalable Computing Systems and Applications in AI, Big Data and HPC

Quantum Autoencoders - Quantum Neural Networks: Libraries and Applications

---

David Alexandre Silva

MatrNr: 24246893

Supervisor: Christian Boehme

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2024

Abstract

This report explores the field of quantum computing, with a particular focus on the development and use of quantum autoencoders (QAEs) in quantum neural networks. Quantum computing faces a major challenge in efficiently managing quantum resources such as qubits, particularly when dealing with complex data sets common in AI, Big Data, and High-Performance Computing (HPC) applications.

Classical autoencoders are proficient in data compression and feature extraction, but they struggle with high-dimensional data sets typical in these fields due to limitations in processing power and storage. Quantum autoencoders offer a novel solution by leveraging the principles of quantum mechanics to enhance data processing capabilities.

We present a new classification technique that utilises quantum autoencoders to compress and classify binary data, reducing the quantum resources required. This method addresses the challenge of resource constraints on Noisy Intermediate-Scale Quantum (NISQ) devices by optimising the quantum circuitry for autoencoding tasks, enabling the practical use of QAEs for data compression and feature learning. Note that we typically publish your report as PDF on our webpage. Let us know if you disagree.

The proposed solution's effectiveness was evaluated using a subset of the MNIST dataset, focusing on binary classification of handwritten digits. The variational quantum circuit was optimized using a combination of scikit-learn optimizers and a unique data shuffle method. This aimed to improve the learning process by exposing the model to a diverse set of images during training. The study shows that quantum autoencoders can be implemented cost-effectively with fewer qubits, making them suitable for processing high-dimensional data sets.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables iv

List of Figures iv

List of Listings iv

List of Abbreviations v

1 Introduction 1

1.1 Motivation and Objectives 2

2 Development 3

2.1 Loading and normalizing MNIST dataset 3

2.2 Building the autoencoder 4

2.3 Swap test 5

2.4 Concluding the autoencoder circuit 5

2.5 Optimisation 5

3 Results 6

4 Conclusion 7

References 8

A Code samples A1

List of Tables

List of Figures

1 Autoencoder layers representation 1
2 Quantum autoencoder layers representation 2
3 First element of the dataset before and after resizing 3
4 Swap test example with 2 qubits 5
5 MSE, PSNR and SSIM Results 7

List of Listings

1 Loading and re-scaling snippet 3
2 Resizing snippet 3
3 Function to create a quantum variational circuit 4
4 Final autoencoder circuit 6
5 Optimisation A2

List of Abbreviations

HPC High-Performance Computing

NISQ Noisy Intermediate-Scale Quantum

MNIST Modified National Institute of Standards and Technology

1 Introduction

An autoencoder or more precisely a classical autoencoder (CAE) is a special type of neural network that is used to compress and encode information from the input efficiently using representation learning. Autoencoders are represented usually by three layers: the encoder layer, the bottleneck layer and the decoder layer. As illustrated in Figure 1. The encoder compresses the input data into a lower-dimensional representation known as the bottleneck layer, and the decoder reconstructs the original input data from this encoding. The autoencoder learns to minimize the reconstruction error, encouraging the network to learn a compact and informative representation of the input data. Autoencoders are usually used for unsupervised learning, dimensionality reduction, and feature learning. As a result, CAEs are often used for real-world tasks such as image denoising, anomaly detection and face recognition. [Wik24]

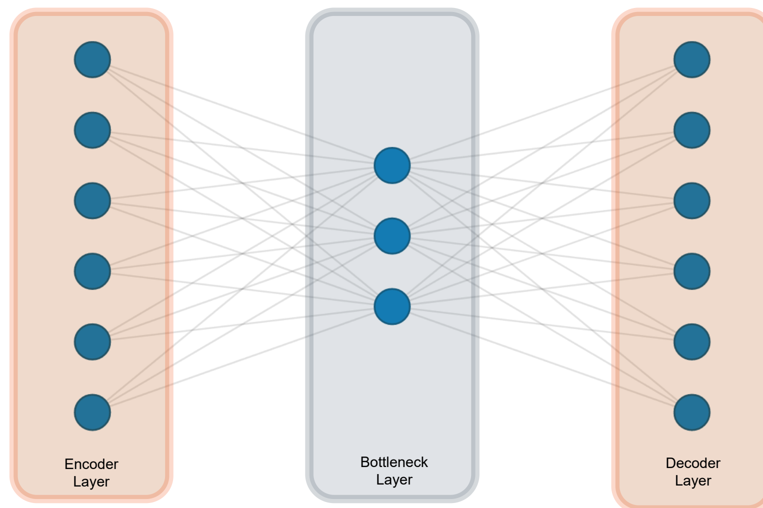


Figure 1: Autoencoder layers representation

Like its classical counterpart, the quantum version contains three layers. First, the input the state $|\psi >$ (containing n qubits) that will be compressed. This is our input layer (encoder). Then the parameterised circuit is applied to the input state, which acts as our encoder and 'compresses' our quantum state, reducing the dimensionality of our state to $n - k$ qubits.

This parameterised circuit will depend on a set of parameters, which will be the nodes of our quantum auto-encoder. Throughout the training process, these parameters will be updated to optimise the loss function. For the rest of the circuit, the remaining k qubits should be ignored. This is the bottleneck layer and the input state is now compressed.

The final layer consists of adding k qubits (all in state $|0>$) and applying another parameterised circuit between the compressed state and the new qubits. This parameterised circuit acts as our decoder and reconstructs the input state from the compressed state using the new qubits. After the decoder, the original state is retained as the state travels to the output layer (decoder).[Fer+21] As illustrated in Figure 2.

Before building our quantum autoencoder, we need to note a few subtleties. Firstly, when implementing an autoencoder using Qiskit, we cannot introduce or ignore qubits in the middle of a quantum circuit. Hence, we must introduce our reference state as well as

our auxiliary qubits (whose role will be described in later sections) at the beginning of the circuit. Therefore, our input state will consist of our input state, reference state and an auxiliary qubit, as well as a classical register for performing measurements.

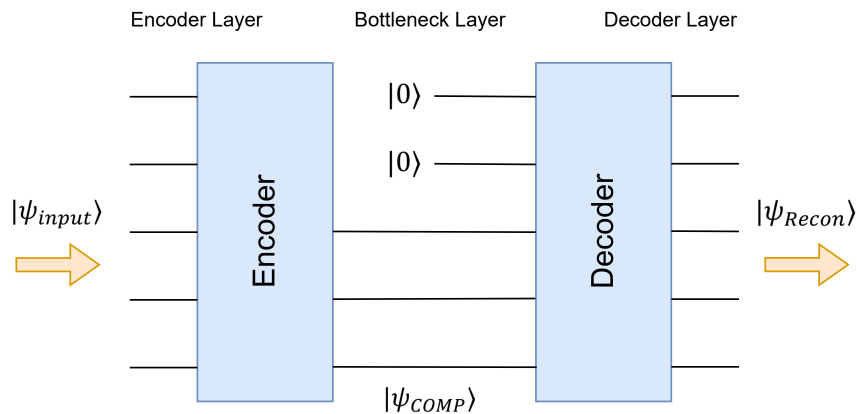


Figure 2: Quantum autoencoder layers representation

1.1 Motivation and Objectives

A quantum autoencoder employs the foundational principle of its classical counterpart, but it operates on statevectors instead of neurons. Put simply, a quantum state autoencoder is a circuit designed to take a statevector as its input and produce a compressed version of this statevector as output. To approximately retrieve the original statevector from its encoded form, one can use the reverse process, facilitated by a Quantum State Decoder. This approach is particularly advantageous because, as is well-known, resources in Noisy Intermediate-Scale Quantum (NISQ) machines are limited. Hence, utilizing a quantum autoencoder allows for the efficient use of these scarce resources.

Now that I understand what autoencoders are and why they are important, I will apply them to a variational circuit that classifies images to demonstrate the capabilities of a quantum autoencoder. This includes:

- Create the statevector that will be reduced.
- Build a quantum autoencoder.
- Present the decoder circuit.
- Optimize the circuits and evaluate the results of this optimization process.
- Present the quantum classifier and the results from a classification.

Given the reason that I had no previous experience in this subject I decided to apply this on a simple dataset. The Modified National Institute of Standards and Technology (MNIST) dataset is a large database of handwritten digits that is commonly used for training various image processing systems. The dataset is also widely used for training and testing in the field of machine learning. This dataset contains 60,000 training images and 10,000 testing images.[Ola14]

Although, I'll be using only images of the handwritten numbers 0 and 1, to simplify this practical example.

2 Development

There have been several versions of my code as I have tried to learn these new concepts and terms, while also trying to understand how each part of this area of quantum computing effectively works. Although this next section shows the final version, the earlier and less mature version is in the code attached to this report.

2.1 Loading and normalizing MNIST dataset

The MNIST dataset is loaded using TensorFlow, as it already has the dataset on it, and divided into a training set and a test set. The images are then normalized from the [0, 255] range to the [0.0, 1.0] range.

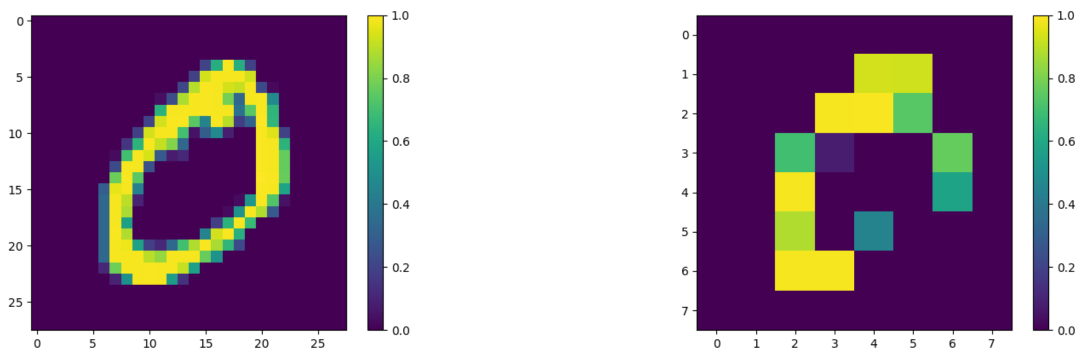
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train[..., np.newaxis]/255.0
x_test = x_test[..., np.newaxis]/255.0
```

Listing 1: Loading and re-scaling snippet

The dataset is then filtered by the numbers 0 and 1 as mentioned above. Due to my underpowered CPU, I decided to resize the images from the original size of 28x28 to 8x8 pixels. I used nearest neighbour interpolation as the resizing method. Since anti-alias has no effect when used with nearest neighbour interpolation.

```
x_train_small = tf.image.resize(
    x_train, (8,8), method='nearest', preserve_aspect_ratio=True).numpy()
x_test_small = tf.image.resize(
    x_test, (8,8), method='nearest', preserve_aspect_ratio=True).numpy()
```

Listing 2: Resizing snippet



(a) First element of the dataset

(b) First element of the dataset resized

Figure 3: First element of the dataset before and after resizing

Further optimisations have been made to the example for performance and non-valuable data reasons. These can be seen in the attached full code.

2.2 Building the autoencoder

For this segment, 6 qubits are considered because $64 = 2^6$. The amplitude method will be used with only a single layer. The circuit used for this application will be presented later.

It is confirmed that the input vector with index zero is normalised to a state vector. By inputting the state vector into the method initialize(vectorstate, qubits) a 6-qubit quantum circuit is created. It should be noted that the computational cost of this function may vary depending on the method generated, especially when 0 amplitude states dominate.

Placing the state vector into the initialize(vectorstate, qubits) method will also produce a 6-qubit quantum circuit. It is important to note that the computational complexity may vary depending on the generated method, especially in cases where 0 amplitude states are dominant.

```
1  def vqc(n, num_layers,params):
2      parameters = ParameterVector('\theta', 10*(num_layers))
3      len_p = len(parameters)
4      circuit = QuantumCircuit(n, 1) #create the quantum circuit with n qubits
5      for layer in range(num_layers):
6          for i in range(n):
7              circuit.ry(parameters[(layer)+i], i)
8          circuit.cx(2,0)
9          circuit.cx(3,1)
10         circuit.cx(5,4)
11         circuit.ry(parameters[6+(layer)],0)
12         circuit.ry(parameters[7+(layer)],1)
13         circuit.ry(parameters[8+(layer)],4)
14         circuit.cx(4,1)
15         circuit.ry(parameters[9+(layer)], 1)
16     params_dict = {}
17     i = 0
18     for p in parameters:
19         params_dict[p] = params[i]
20         i += 1
21     circuit = circuit.assign_parameters(parameters = params_dict)
22     return circuit
```

Listing 3: Function to create a quantum variational circuit

The process starts by specifying the number of layers and qubits. The parameter vectors are initialised with a string identifier and an integer for the vector length. The quantum circuit is created with n qubits. A circuit is created for each layer. A Ry gate is applied to each qubit. Based on the layer, the rotation of the Ry gate is defined in the parameter list. A barrier is created for clearer interpretation of the circuit sections. A

Cnot gate is applied between qubits 2 and 0, 3 and 1, and 5 and 4. In qubits 0, 1 and 4, *Ry* gates are applied with specified rotations from the parameter list. A *Cnot* gate is then applied between qubits 4 and 1. Finally, an *Ry* gate is applied to qubit 1 with the rotation specified in the parameter list.

A dictionary is then initialised with parameters assigned to it; the keys are composed of the string identifier and an integer specifying the vector length.

The structure of the network circuit includes 10 $Ry(\theta)$ q-gates and 4 *Cnot* gates. Since the cost is related to the number of *Cnot* gates, it is 4.

2.3 Swap test

The SWAP test is a commonly used procedure to compare two states by applying *Cnot* gates to each qubit.

Initiated by setting a new qubit value for the circuit design. The quantum circuit is then built using this qubit value. An H gate is applied to the first qubit, followed by a $CSwap$ gate involving the first qubit and the $i+1$ and $2*n-i$ qubits. Finally, another H gate is applied to the first qubit. Figure 4 shows an example of a swap test.

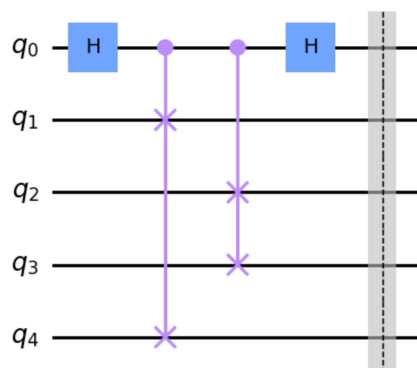


Figure 4: Swap test example with 2 qubits

2.4 Concluding the autoencoder circuit

Now I determine the circuit of the autoencoder to encode 6 qubits into 4 qubits

Start by specifying the number of qubits to be reduced. Build a quantum variational circuit, then build a swap test circuit. Initialise a new circuit that reflects the dimensions of the original circuit and the intended reduction in qubits. Integrate the original circuit, the quantum variational circuit and the swap test. Starting at qubit $sizereduce + 1$ for both the original circuit and the quantum variational circuit (QVC), and starting at qubit 0 for the swap test.

2.5 Optimisation

After learning how to construct a quantum autoencoder and its inverse, the next step is to improve the circuit. This will require demonstrating the optimization of the variational

```
size_reduce = 2
circuit_init = input_data(n,x_train[0])
circuit_vqc = vqc(n,num_layers,params)
circuit_swap_test = swap_test(size_reduce)

circuit_full = QuantumCircuit(n+size_reduce+1,1)

circuit_full = circuit_full.compose(circuit_init,[i for i in range(size_reduce+1,n+si
circuit_full = circuit_full.compose(circuit_vqc,[i for i in range(size_reduce+1,n+siz
circuit_full = circuit_full.compose(circuit_swap_test,[i for i in range(2*size_reduce
circuit_full.draw(output="mpl")
```

Listing 4: Final autoencoder circuit

circuit and evaluating success using various metrics. Instead of using qiskit's optimizers, we chose to use those from scikit-learn due to integration difficulties with an amplitude map. In addition, we used the shuffle method to introduce a range of images in each optimization iteration, with the aim of improving the outcomes. As shown in Listing 5 on the appendix.

3 Results

To find out how well the optimisation of the variational circuit has worked, it's necessary to compare the images. The following metrics are used to compare the reconstructed images obtained by the autoencoder with the original images:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2,$$

Mean Square Error, where m is the image height I, n the width of the image K and i, j the positions x, y images pixels; the closer to 0 get the better the result.

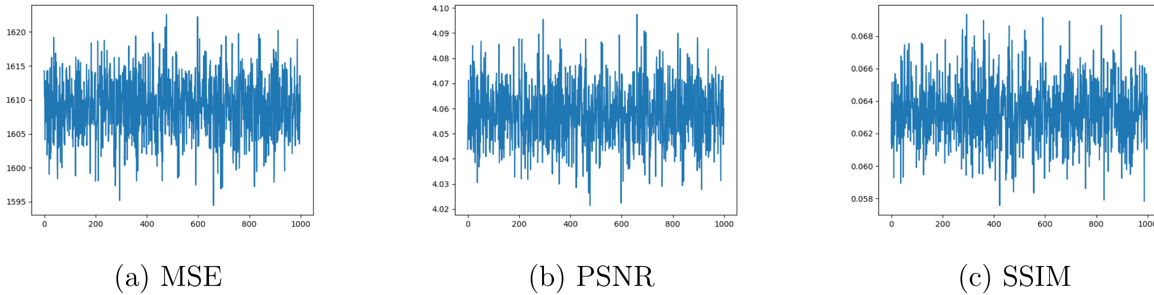
$$PSNR = 10 \times log_{10}(\frac{(mxn)^2}{MSE},$$

Peak Signal-to-noise Ratio, where m is the image height I, n the width of the image K and MSE the mean square error; the bigger the better.

Structural Similarity Index Measure, where μ is the mean, σ is the standard deviation and c is covariance ; The worst case is -1 and the best is 1.

I obtained these results from the optimisation, but I was not able to compare each sample plot simultaneously with different numbers of images. Nevertheless, the 1000 image sample was the most successful of the three metrics, achieving the most desired numbers for each metric.

$$\mathrm{SSIM}(x,y) = \frac{\left(2\mu_x\mu_y + c_1\right)\left(2\sigma_{xy} + c_2\right)}{\left(\mu_x^2 + \mu_y^2 + c_1\right)\left(\sigma_x^2 + \sigma_y^2 + c_2\right)},$$



(a) MSE (b) PSNR (c) SSIM

Figure 5: MSE, PSNR and SSIM Results

4 Conclusion

Current quantum models generally do not outperform classical models when working with classical data, especially with modern datasets that can contain over a million data points. However, the challenge posed by large data sets can be mitigated by specific data reduction techniques that reduce both the cost and the number of qubits required. Certain data sets are more amenable to quantum models than classical ones.

In this project, a classification approach for 0's and 1's using an autoencoder is presented. This strategy potentially reduces the number of qubits required for replication on a real quantum computer, resulting in a cost of 4.

It is evident that quantum computing has a bright future in several fields, such as cryptography, drug discovery, and complex system simulations. Recent studies have highlighted the significance of hybrid quantum-classical systems. In these systems, quantum devices perform tasks that take advantage of their quantum capabilities, while classical systems handle tasks that are better suited to traditional computing methods. This symbiosis between quantum and classical computing has the potential to solve previously intractable problems and achieve breakthroughs in science and technology.

References

[Fer+21] Martínez Vázquez María Fernanda et al. *Quantum Autoencoder with MNIST classification*. https://fullstackquantumcomputation.tech/blog/quantum-autoencoder/. 2021.

[Ola14] Christopher Olah. *Visualizing MNIST: An Exploration of Dimensionality Reduction*. https://colah.github.io/posts/2014-10-Visualizing-MNIST/. 2014.

[Wik24] Wikipedia contributors. *Quantum computing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Quantum_computing&oldid=1216052106. 2024.

A Code samples

```python
def objective_function(params):
    costo = 0
    shuffle(x_train) #reorganize the order of the train set items
    lenght= 5
    #For each item of the trainig set
    for i in range(lenght):

        circuit_init = input_data(n,x_train[i])
        circuit_vqc = vqc(n,num_layers,params)
        circuit_swap_test = swap_test(size_reduce)

        circuit_full = QuantumCircuit(n+size_reduce+1,1)

        circuit_full = circuit_full.compose(
        circuit_init,[i for i in range(size_reduce+1,n+size_reduce+1)])
        circuit_full = circuit_full.compose(
        circuit_vqc,[i for i in range(size_reduce+1,n+size_reduce+1)])
        circuit_full = circuit_full.compose(
        circuit_swap_test,[i for i in range(2*size_reduce+1)])
        circuit_full.measure(0, 0) #Measure the first qubit
        #qc.draw()
        shots= 8192 #Number of shots
        #Execute the circuit in the qasm_simulator
        job = execute( circuit_full, Aer.get_backend('qasm_simulator')
        ,shots=shots )
        counts = job.result().get_counts()
        probs = {}
        for output in ['0','1']:
            if output in counts:
                probs[output] = counts[output]/shots
            else:
                probs[output] = 0
        costo += (1 +probs['1'] -  probs['0'])

    return costo/lenght

for i in range(1):
    #Minimization of the objective_fucntion by a COBYLA method
    minimum = minimize(objective_function, params,
    method='COBYLA', tol=1e-6)
    params = minimum.x #Get the solution array
    #Show the cost of the solution array
    print(" cost: ",objective_function(params))
    print(params)
```

Listing 5: Optimisation