Enhanced Quantum Autoencoders for Anomaly Detection

In this notebook, we review a quantum machine learning solution to anomaly detection using quantum autoencoders. We present a background on classical and quantum autoencoders firstly and briefly review the approaches to building a quantum autoencoder. Next, we detail our approach for how a quantum autoencoder can be used to perform anomaly detection and then test it on two different real-world problems breast cancer tests (BIO-QML challenge) and credit card fraud detection (Quantum Finance Challenge).

We train QAEs architectures solely on non-anomalous data. Then, given an anomaly datapoint coming from a different than learned distribution, QAE provides non-faithful reconstruction, indicating an anomaly appearance. We judge the reconstruction quality either using the fidelity test (suitable for simulations) and SWAP test (suitable for both simulations and QPU hardware). We extend previous work on anomaly detection using QAEs by employing for this task for the first time the enhanced autoencoder and the patch autoencoder. We obtain results showing in some cases up to 91% accuracy in identifying breast cancer and 88% in fraudulent transaction identification.

Also we detail a novel approach to building a quantum autoencoder that makes use of quantum entanglement as a resource to add an extra source of correlation between the compression and decompression process. We develop various conceptually different ideas: we let the encoder and decoder share Bell pairs, we entangle encoder and decoder qubits directly and we test what happens if we allow for both the encoder and decoder training contrary to the standard approach. So far, we have not encountered an architecture that would provide statistically significant advantage.

This notebook is only for presentation, the summary of our research during the hackathon, and not meant to be run. The notebooks and codes are all in the GitHub repository. One of the exploration notebooks that we used, for example, is this one.

1. Introduction

1.1 Quantum Autoencoder

An autoencoder is a neural network architecture that is trained in a way that allows it to discover structures within data, usually use as a way to reduce the dimensionality of the data. An autoencoder has two parts, the encoder and decoder. The data is first feed into the encoder, resulting in a new representation of that data. The decoder will then take this new representation of data and reproduce the reconstructed input.

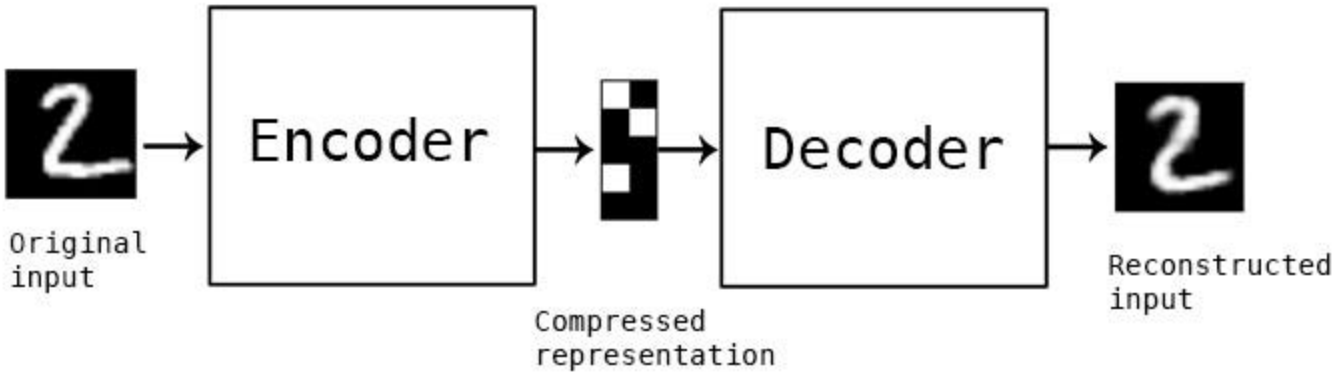


Figure 1. An ilustration of an autoencoder taking the MNIST dataset as input. Image source: here.

The aim for an ideal autoencoder is to train both the encoder and decoder so that the decoder is able to reproduce the original input dataset as close as possible. In the classical autoencoder, both the encoder and decoder usually are fully-connected neural networks.

A quantum autoencoder[1] takes its inspiration from the classical autoencoder. The difference is that instead of fully-connected neural networks, the encoder and decoder are now variational quantum circuits. Below is an example of circuit schematic for a quantum autoencoder.

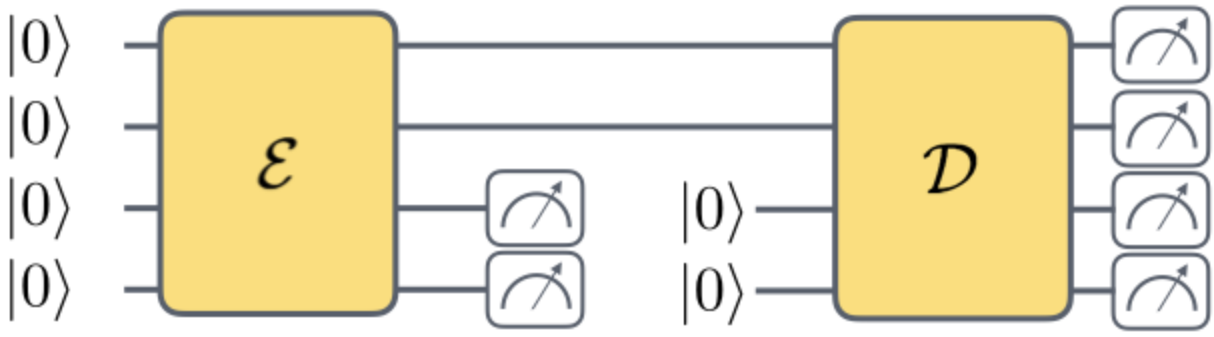


Figure 2. An example of a quantum autoencoder. Image source: here.

The input to the quantum autoencoder is a quantum state, here represented as the $|0000\rangle$ state. A classsical data can be embedded into the circuit as a quantum state with several methods, but the ones we used in this project are the angle embedding and the amplitude embedding method.

The E and D are the quantum encoder and quantum decoder respectively, where $D = E^{\dagger}$. Some of the output qubits of the encoder are measured (the "trash qubits"), and we pass the rest ("the latent space qubits") to the decoder. The decoder will take those qubits together with some newly initialized qubits ($|00...00\rangle$) as inputs.

During training, we want to train both variational circuits so that the measured qubits from the encoder match the initialized qubits in the decoder. If the measured qubits are in the state of $|00...00\rangle$, the output qubits of the decoder must be in the state of $|0000\rangle$, the same as the state of the input state. This basically achieves what a classical autoencoder does, with the latent space qubits give the "compressed representation" of the input state, similar to how classical encoder will give the compressed representation of the input data.

1.2 Anomaly Detection with Autoencoder

One prominent application of an autoencoder is to detect anomaly from data. The idea is to train the autoencoder with only the true labeled data (the one considered as the non-anomaly data) until it can reproduce the input data to a certain similarity. After that, during the inference/detection, we use the sample data that we want to detect as inputs to the autoencoder, if the autoencoder is able to reproduce the data with an equal similarity with its performance during training (within a small tolerance), we can say that the data are also non-anomaly since they most likely come from the same distribution as the training data. But if the autoencoder is unable to reproduce the data within that similarity level, then we can hypothesize that the data might come from different distribution as the training data, and thus considered as "anomaly".

In quantum autoencoder, this is similar to check how similar the trash qubits' state and the $|00...00\rangle$ state are. There are multiple ways to check similarity level (fidelity) between two quantum states. On a quantum hardware we can employ the swap test, or we can directly calculate the fidelity if we run the simulation on classical hardware.

As an example, let's say we want to detect whether a certain set of physical symptoms in a patient indicates that the patient is infected with a disease with a quantum autoencoder. We can collect the negative data (set of physical symptoms) from healthy patients and train the autoencoder with that data. After that, if we want to detect a new patient, we can feed the physical symptoms of that patient to the autoencoder and check what is the fidelity between the trash qubits' state and $|00...00\rangle$ state. If the fidelity is low, we can hypothesize that the patient might be positive (unhealthy) and vice-versa.

2. Methods

2.1 Datasets

2.1.1 Breast Cancer Dataset

For the first dataset in our project, we used the Breast Cancer Wisconsin (Diagnostic) Dataset that was being produced and used first in the study of [3] and available at [4].

This dataset describes characteristics of the cell nuclei present in the digitized image of a fine needle aspirate (FNA) of a breast mass. In total, there are 30 features with 2 classes (malignant and benign).

We had 30 features, and since it is not a power of two, we decided to engineer two new futures that we call under and over average that count how many filters are at more than two standard deviations away. And this proved to be a good decision since they strongly correlate with the disease column. For some experiments, we encode all 32feutures on qubits, but to speed up things for some of them, we used only the top 8 more correlated features with the disease type.

2.1.2 Credit Card Fraud Detection Dataset

We used the dataset from https://www.kaggle.com/mlg-ulb/creditcardfraud which contains over 280,000 credit card transactions described by 30 features and 2 classes ('O' = non fraudulent transaction and '1' = fraudulent transaction).

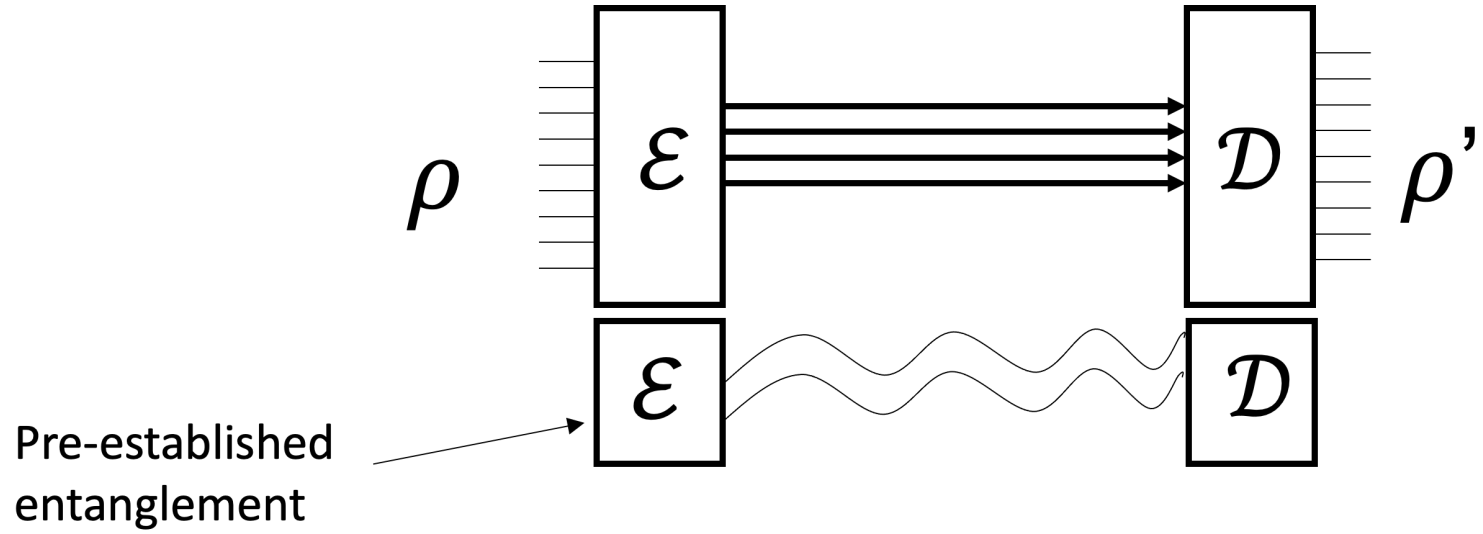
2.2 Our Approach for the Quantum Autoencoder

2.2.1 Entanglement-Assisted Quantum Autoencoder

Quantum entanglement used as a resource adds an advantage that cannot be obtained with purely classical approaches. This phenomenon manifests itself in CHSH game and quantum superdense coding, where entanglement boosts the winning probabilities for the players and communication rate, respectively.

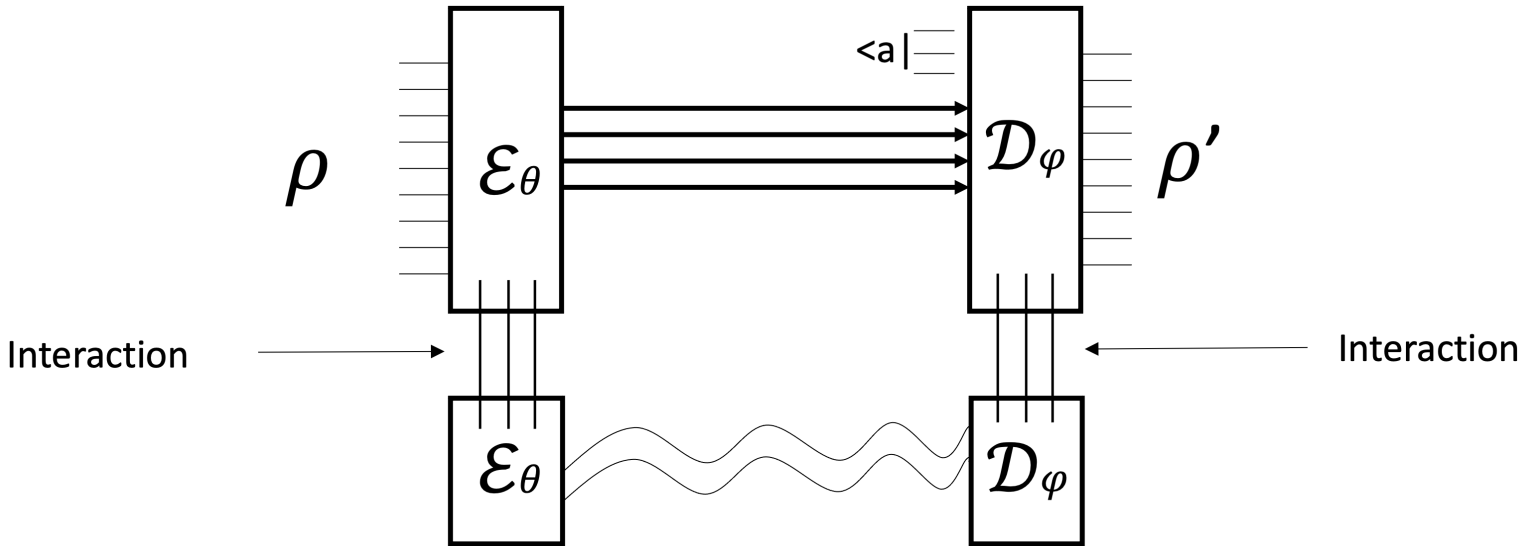
In our project, we investigate the use of entanglement in the task of quantum autoencoding. Quantum autoencoders[1] compress quantum information into smaller dimensional Hilbert spaces, and moreover, it is known that entanglement resources can aid in this compression[2]. We take state-of-the-art quantum autoencoder strategies, and add additional entanglement resources to test for a compression advantage. In future work, we aim to test the approach on datasets such that the known autoencoder schemes do not compress and decompress the data properly, but when we add the entanglement resources, the result improves. Indeed part of the work is to identify such datasets.

In the image below, we depict how the entanglement fits in the quantum autoencoding framework. The entanglement is established before the time of encoding, and is used to coordinate the sender and the receiver when encoding and decoding takes place. How the entanglement is used can vary and can produce varying results.



In our work, we tested two approaches: One in which the encoder interacts with with entangled qubits and the other in which the encoder does not interact with the entangled qubits. The two images below depict the two settings.

Interaction:



No Interaction: Image

The approach to interact the data qubits with the entangled qubits was the first approach. Here we simply extended the encoding ansatz circuit to contain the entangled bits, where the coding qubits were simply the first Bell state. We found that in training, this was too restrictive and performed worse than without entangelment. We then tried not interacting the qubits, but again this did not have much affect.

With further research, we learned that the entanglement can be best used as a perfect coordinator or with additional classical resources, it can be used for quantum teleportation, to reduce the requirement of the quantum transmission. Without classical information, we can perform measurements before hand and based on the measurement outcome can perform encoding and decoding. To implement this, we perform mid-circuit measure and perform classical control gates on the encoder and decoder.

These approaches are still under development and future work is to further develop these concepts.

3. Results

Note that in this section, we put the plots and prints of our results as images & texts since we ran the code in our temporary notebooks and moved all of the results to the final notebook (this notebook) after we finished.

3.1 Results on Breast Cancer Detection

In this project, we tried three different circuit ansatze for the quantum autoencoder to detect anomaly in the Breast Cancer Dataset. The first ansatz is our new proposed ansatz, the second and third ansatz are taken from previous related works as comparisons to ours.

3.1.1 First Ansatz

Here we were inspired by the scalable variational quantum encoder paper. The basic idea is that two different autoencoders encode the data. We used the autoencoders from the first quantum autoencoder paper that we named e2. However, we have done some tricks in the way in which we encode the data then are new: For the first encoder, we passed the data using amplitude embedding, and for the second encoder, we used angle embedding. The results for thet experiments can be found here.

However, below, we present the code for a circuit that provides us a better result. We made the two encoders interact by adding two CNOTs from the latent space of the one to the thrash qubit of another one.

```
shots = 2500
nr_trash=2
nr_latent=2
nr_ent=0

trash_qubits1=[i for i in range(nr_trash)]
latent_qubits1=[i for i in range(nr_trash,nr_trash+nr_latent)]
trash_qubits2=[i for i in range(nr_trash+nr_latent,2*nr_trash+nr_latent)]
latent_qubits2=[i for i in range(2*nr_trash+nr_latent,2*(nr_trash+nr_latent))]
aux_qubits=[i for i in range(2*(nr_trash+nr_latent),2*(nr_trash+nr_latent)+2*nr_trash)]
swap_qubit=[2*(nr_trash+nr_latent)+2*nr_trash]

qubits=[*trash_qubits1, *latent_qubits1, *trash_qubits2, *latent_qubits2, *aux_qubits, *swap_

print("Qubits:", qubits)

#set up the device
dev = qml.device("default.qubit", wires=qubits)
```

```
@qml.qnode(dev)
def training_circuit_example(init_params, encoder_params, reinit_state, x):
    # Initialization


    qml.templates.embeddings.AmplitudeEmbedding(
        init_params,
        wires=[*trash_qubits1, *latent_qubits1],
        normalize=True,
        pad_with=0.0j,
    )


    qml.templates.embeddings.AngleEmbedding(
        init_params[:4], wires=[*trash_qubits2, *latent_qubits2], rotation='X')
    qml.templates.embeddings.AngleEmbedding(
        init_params[:4], wires=[*trash_qubits2, *latent_qubits2], rotation='Z')

    qml.MottonenStatePreparation(reinit_state, wires=aux_qubits)

    #encoder
    e5_patch(*encoder_params[0],*encoder_params[1], [*trash_qubits1, *latent_qubits1], [*late
    qml.CNOT(wires=[latent_qubits1[0],trash_qubits2[0]])
    qml.CNOT(wires=[latent_qubits2[0],trash_qubits1[0]])

    #swap test
    trashes=[*trash_qubits1,*trash_qubits2]
    qml.Hadamard(wires=swap_qubit[0])
    for i in range(len(trashes)):
        qml.CSWAP(wires=[swap_qubit[0], aux_qubits[i], trashes[i]])
    qml.Hadamard(wires=swap_qubit[0])

    return [qml.probs(i) for i in swap_qubit]
```

```
epochs = 500
learning_rate = 0.0003
batch_size = 5
num_samples = 0.8 # proportion of the data used for training

beta1 = 0.9
beta2 = 0.999
opt = AdamOptimizer(learning_rate, beta1=beta1, beta2=beta2)
```

```
def fid_func(output):
    # Implemented as the Fidelity Loss
    # output[0] because we take the probability that the state after the
    # SWAP test is ket(0), like the reference state
    fidelity_loss = 1 / output[0]
    return fidelity_loss
```

```
def cost(encoder_params, X):
    reinit_state = [0 for i in range(2 ** len(aux_qubits))]
    reinit_state[0] = 1.0
    loss = 0.0
    for x in X:
        output = training_circuit_example(init_params=x[0], encoder_params=encoder_params, re
        f = fid_func(output)
        loss = loss + f
    return loss / len(X)
```

```
def fidelity(encoder_params, X):
    reinit_state = [0 for _ in range(2 ** len(aux_qubits))]
    reinit_state[0] = 1.0
    loss = 0.0
    for x in X:
        output =  training_circuit_example(init_params=x[0], encoder_params=encoder_params, r

        f = output[0]
        loss = loss + f
    return loss / len(X)
```

```
def iterate_batches(X, batch_size):

    random.shuffle(X)

    batch_list = []
    batch = []
    for x in X:
        if len(batch) < batch_size:
            batch.append(x)

        else:
            batch_list.append(batch)
            batch = []
    if len(batch) != 0:
        batch_list.append(batch)
    return batch_list
```

```
training_data = [ torch.tensor([input_data[i]]) for i in range(int(len(input_data)*num_sample
test_data = [torch.tensor([input_data[i]]) for i in range(int(len(input_data)*num_samples),le
```

```
batches=iterate_batches(training_data, batch_size)
```

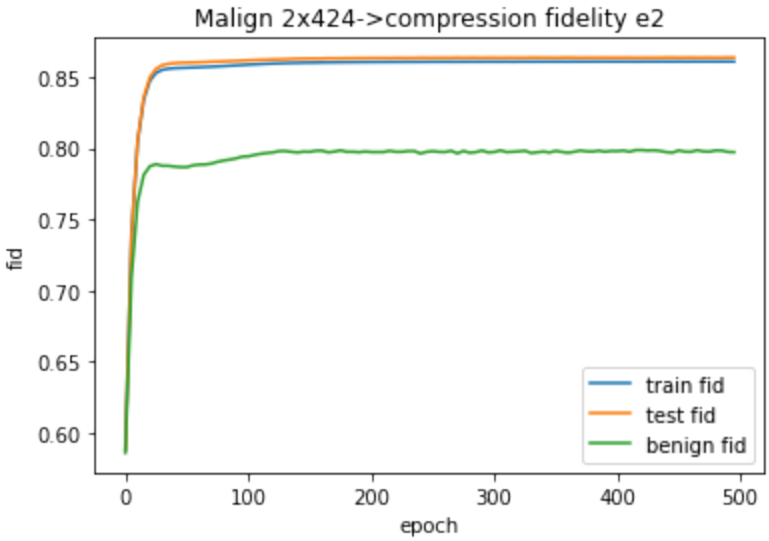
```
X_training = training_data
X_tes = test_data
```

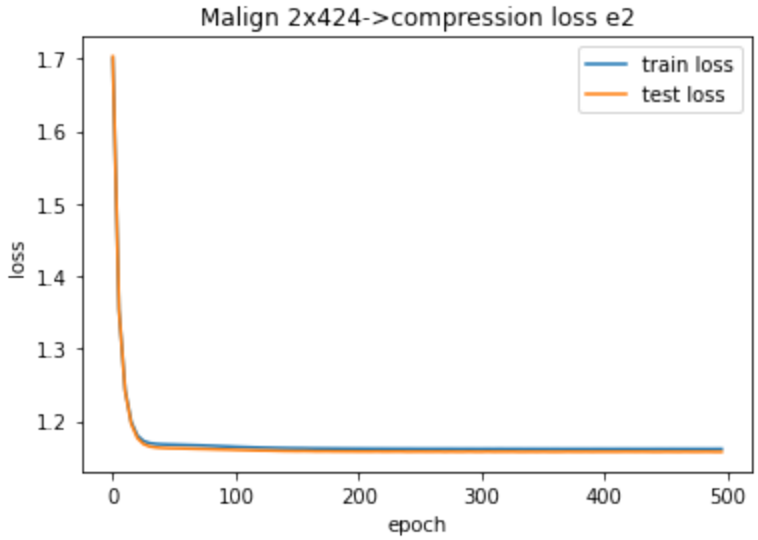
```
nr_encod_qubits = nr_trash + nr_latent
nr_par_encoder =  15 * int(nr_encod_qubits*(nr_encod_qubits-1)/2)
encoder_params = [np.random.uniform(size=(1, nr_par_encoder), requires_grad=True),np.random.u
```

```
np_benign = benign.to_numpy()
benign_data = [ torch.tensor([np_benign[i]]) for i in range(len(benign.to_numpy()))]
```

```
loss_hist=[]
fid_hist=[]

loss_hist_test=[]
fid_hist_test=[]

benign_fid=[]

for epoch in range(epochs):
    batches = iterate_batches(X=training_data, batch_size=batch_size)
    for xbatch in batches:
        encoder_params = opt.step(cost, encoder_params, X=xbatch)


    if epoch%5 == 0:

        loss_training = cost(encoder_params, X_training )
        fidel = fidelity(encoder_params, X_training )

        loss_hist.append(loss_training)
        fid_hist.append(fidel)
        print("Epoch:{} | Loss:{} | Fidelity:{}".format(epoch, loss_training, fidel))

        loss_test = cost(encoder_params, X_tes )
        fidel = fidelity(encoder_params, X_tes )
        loss_hist_test.append(loss_test)
        fid_hist_test.append(fidel)
        print("Test-Epoch:{} | Loss:{} | Fidelity:{}".format(epoch, loss_test, fidel))

        b_fidel = fidelity(encoder_params, benign_data )
        benign_fid.append(b_fidel)
        print("benign fid:{}".format(b_fidel))

        """
        experiment_parameters={"autoencoder":"e2","params":encoder_params}
        f=open("Cancer_encoder_e3-SelectedFeautures/params"+str(epoch)+".txt","w")
        f.write(str(experiment_parameters))
        f.close()
        """
```

Here we can see that the model starts to train. At the begging, it compresses the same maligned and benign data; however, after a few epochs, there is a gap between the quality of encoding the usual cel and the cells with anomalies appear.





Below, we have the histograms of the encoding fidelity. We can see a clear difference between these two categories. Unfortunately, they are still overlapping, but we can use them to spot dangerous cells with 0.9 accuracies. Simply by considering each cell that is compressed with a fidelity lower than the split value 0.845 benign and considered the cels that are well-compressed malign.

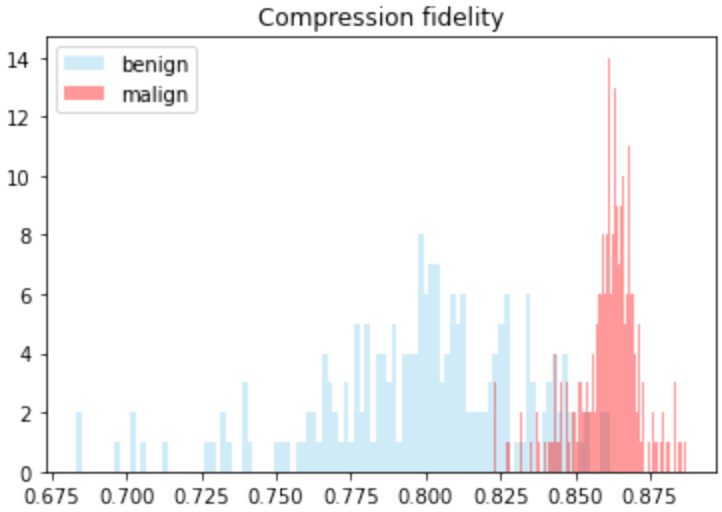
```
beningn_flist=[]
for b in benign_data:
    f=fidelity(encoder_params, [b])
    beningn_flist.append(f.item())

print(min(beningn_flist))
print(max(beningn_flist))
```

0.6833258225888227
0.8617865939720931

```
malign_flist=[]
for b in training_data:
    f=fidelity(encoder_params, [b])
    malign_flist.append(f.item())

print(min(malign_flist))
print(max(malign_flist))
```

0.8226940476819373

0.886700458890945



split: 0.845

benign classification accuracy: 0.9230769230769231

malignification accuracy: 0.9098360655737705

total accuracy: 0.9157175398633257

We also ran the inference on IBM QPU with credential as follow:

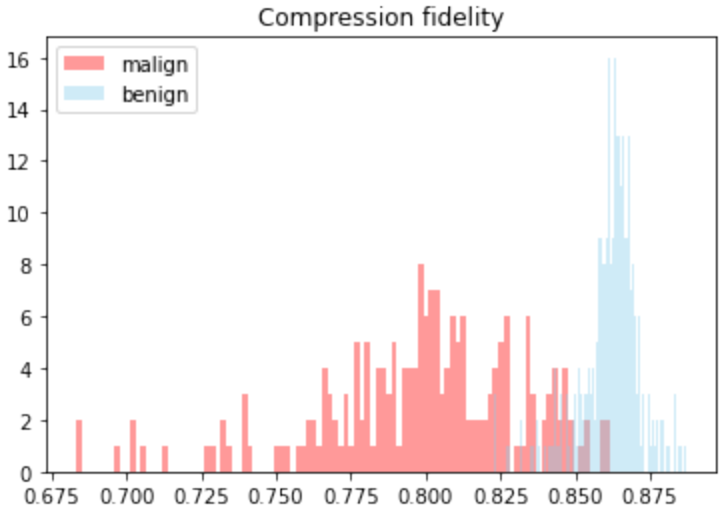
```
IBMQ.save_account(token,overwrite=True)  #save your creds
IBMQ.load_account()
provider=IBMQ.get_provider(hub='ibm-q-community', group='qhack-hackathon', project='16-qubit'
dev_qiskit = qml.device('qiskit.ibmq', wires=qubits, backend='ibmq_guadalupe',provider=provic
```

Below is the comparison of the inference compression fidelity between the one from simulation and the one from the IBM QPU.

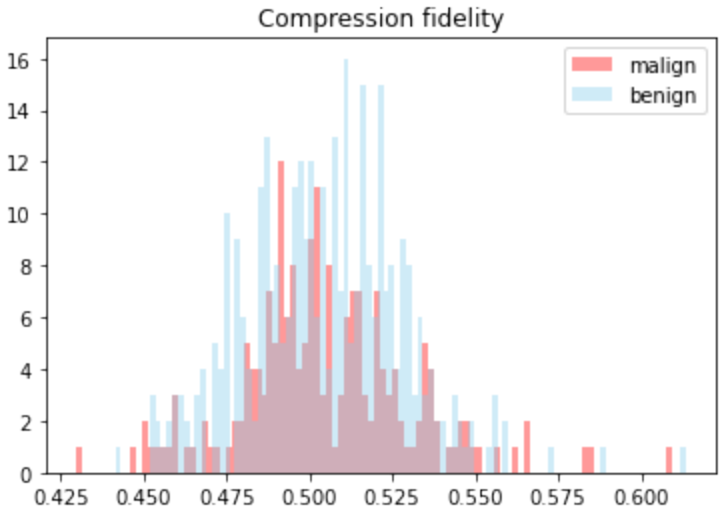
We also tried to run the trained parameters on IBM 16 qubits hardware, but it did not compress the data well because of the noise. With time and persistence, this problem can be mitigated.

https://github.com/VoicuTomut/Enhanced-Autoencoders-for-anomaly-detection/blob/main/Use-case_Cancer_detection/Cancer_encoder_e2.ipynb

Simulation

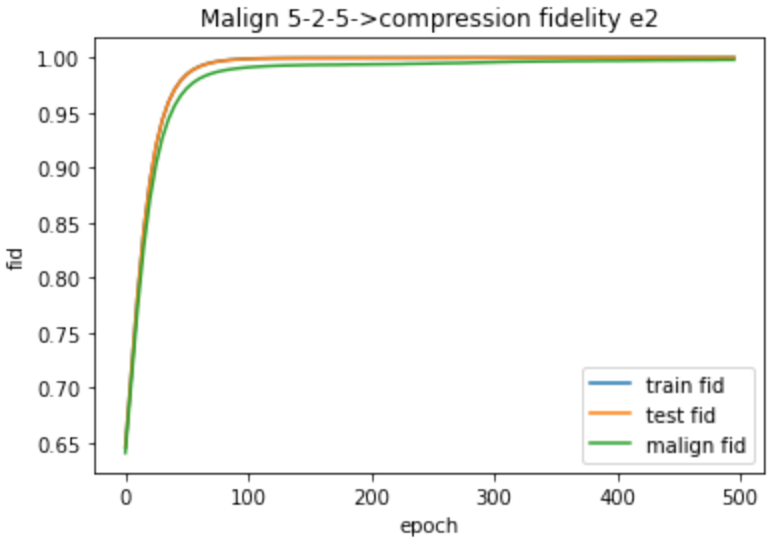


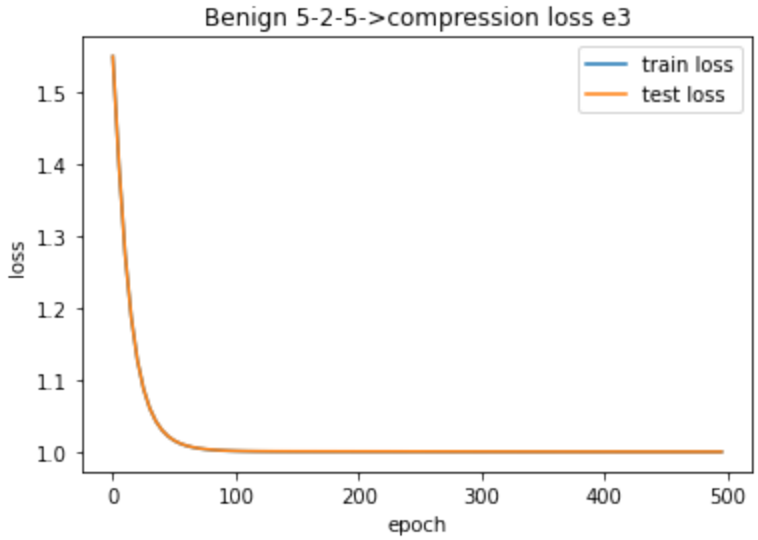
IBM QPU



3.1.2 Second Ansatz

For this experiment, we took circuit ansatz as proposed in [5].





split: 0.9989

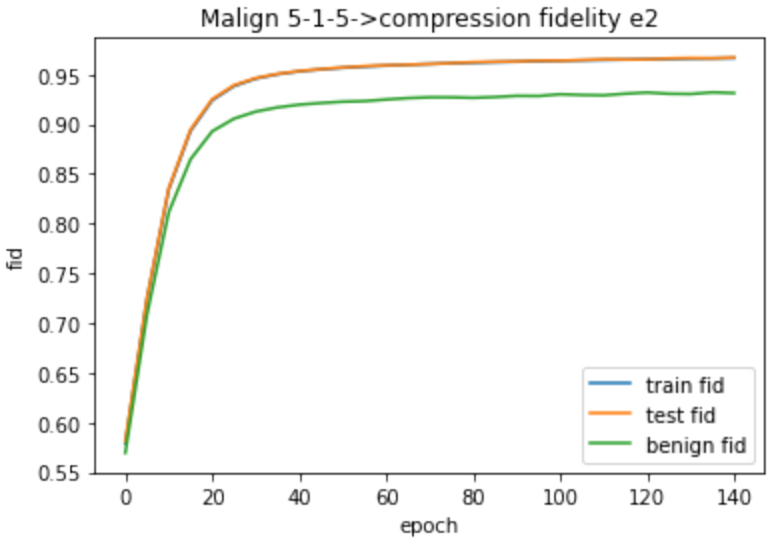
malign classification accuracy: 0.24102564102564103

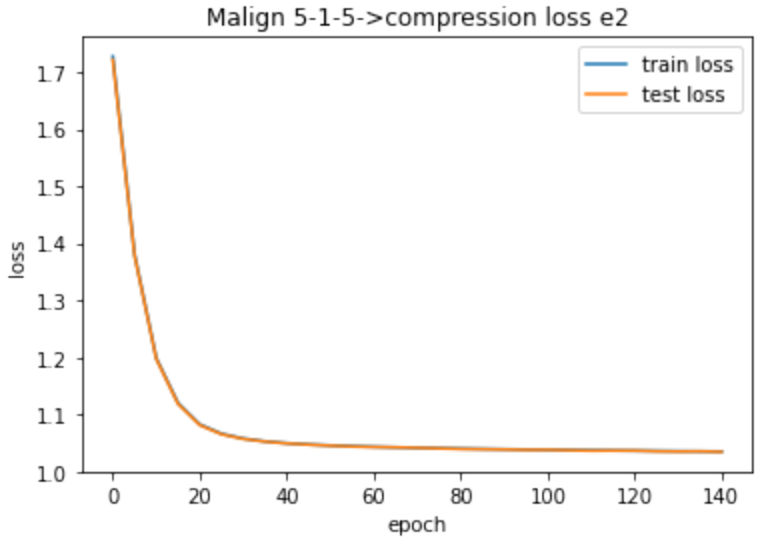
benign classification accuracy: 0.9631147540983607

total accuracy: 0.642369020501139

3.1.3 Third Ansatz

For this experiment, we took circuit ansatz as proposed in [1].



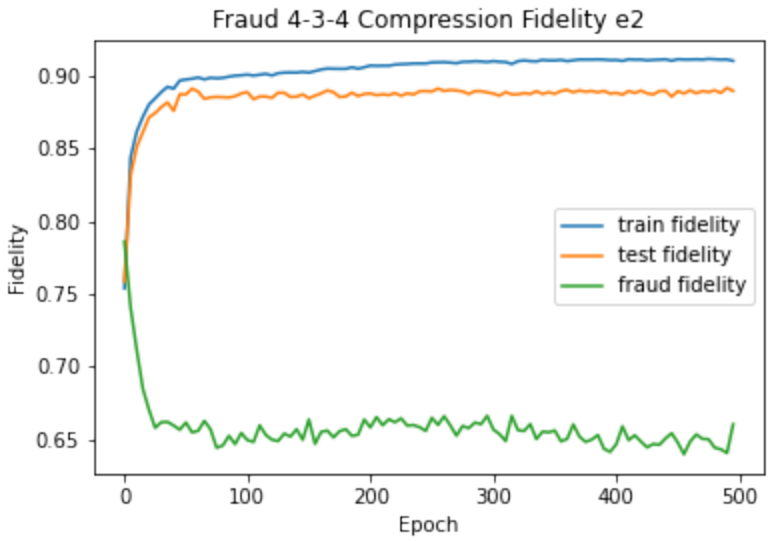


split: 0.96
benign classification accuracy: 0.9016393442622951
malign classification accuracy: 0.8076923076923077
total accuracy: 0.8698481561822126

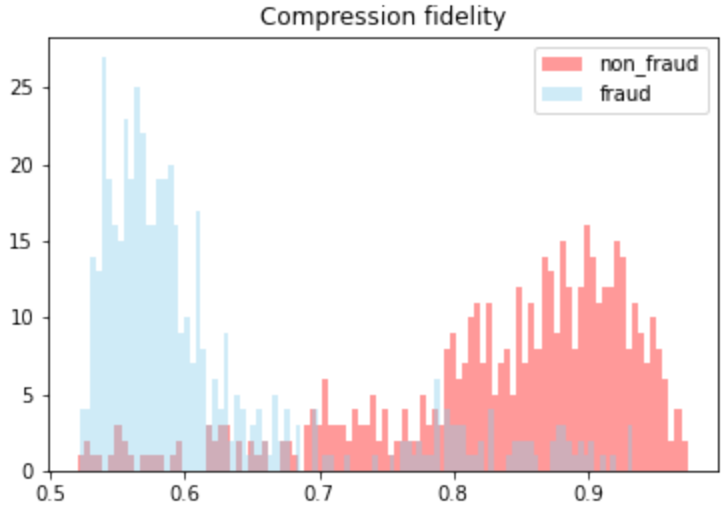
Above, you can see the results for anomaly detection that we get by implementing the ansatse from the paper [5] and [1] for comparison. However, we can do a lot of combinations between the way in which we encode the data and the features that we selected and the cels that we choose to train. We tried to test as many variations we can in the time that we have so please check this : https://github.com/VoicuTomut/Enhanced-Autoencoders-for-anomaly-detection/tree/main/Use-case_Cancer_detection

3.2 Results on Credit Card Fraud Detection

Training the e2 encoder on the credit card dataset ends up with a compression fidelity of non fraudulent transactions of .89 and a compression fidelity of fraudulent transactions of .66. We wanthe the compression fidelity of the non fraudlent transactions to be as high as possible and the compression fidelity of the fraudulent transactions to be as low as possible for the best results of distinguishing between non fraudulent transactions and fraudulent transactions.

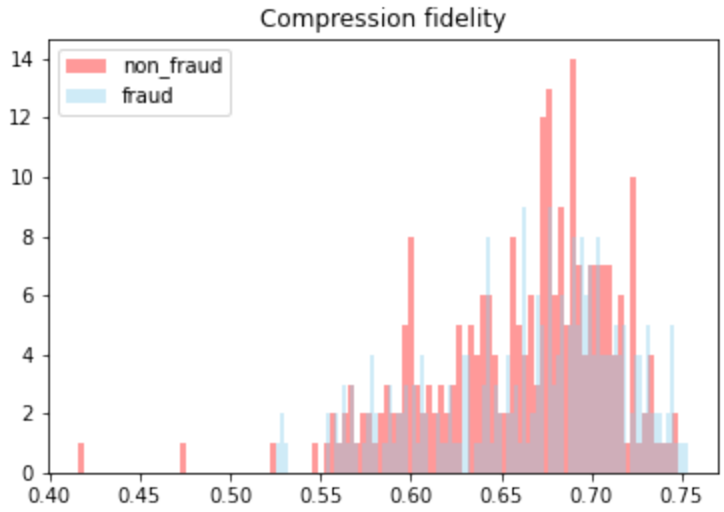


Here is the compression fidelity results which shows the separation between the compression fidelity of the fraudulent transactions and non fraudulent transactions. We want the histograms to have the least amount of overlap and the peaks to be as far as possible for the best results when classifying the transactions.



IBM 16-Qubit Guadalupe through Qiskit

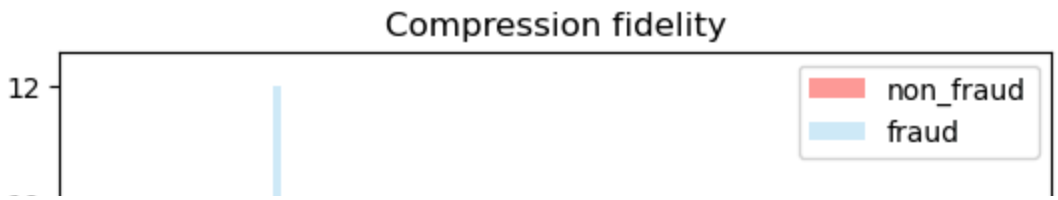
We tested our best parameters from training on the e2 encoder on IBM's 16 qubit quantum computer through Qiskit.



Ideally, we want there to be two peaks where non fraudulent transactions peak on the right side of the graph and the fraudulent transactions peak on the left side of the graph. The ovelap between the histograms means it will be hard to accurately classify between non fraudulent and fraudulent transaction. This lead to classification accuracy of non fraudulent transactions of .992 and a classification accuracy of fraudulent transactions of 0.0.

Rigetti Aspen-11 QPU through Amazon Braket

We used Amazon Braket to submit a job onto Rigetti's Aspen-11 QPU to test our best parameters came from training on the e2 encoder.



4. Closing

4.1 Conclusion