UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Quantum Machine Learning | Project Report

Student name: *Konstantinos Nikoletos*
cs: 7115112200022

Course: *QML*
Semester: *Spring 2024*

1. The Parity Problem

The data used in this project consists of training and test sets designed for the binary classification parity problem with 3 inputs. The parity problem is a classical problem in computational learning theory and serves as a benchmark for evaluating the performance of machine learning models.

Problem definition

The parity problem involves determining whether the number of ones in a binary string is even or odd. For a given binary string of length n:

- If the number of ones is even, the output (label) is 0.
- If the number of ones is odd, the output (label) is 1.

In this project, the parity problem is framed with 3 inputs, which means each input is a binary string of length 3. The possible input combinations and their corresponding outputs are:

Input (Binary String)	Output (Parity)
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

Table 1: Inputs and Outputs for the Parity Problem with 3 Inputs

Data Description

The data for this project consists of noisy versions of the binary strings used in the parity problem. Noise is added to the edges of the binary strings to create a more challenging classification task. The provided datasets include:

- **Training Data**: A set of binary strings with corresponding parity labels for training the quantum classifier. The training data files are ‘classA_train.dat‘ and ‘classB_train.dat‘.

- **Test Data**: A set of binary strings with corresponding parity labels for evaluating the performance of the trained classifier. The test data files are 'classA_test.dat' and 'classB_test.dat'.

Each file contains samples with some added noise to simulate real-world scenarios where data is not perfectly clean. The training and test datasets are used to train the model and then validate its performance on unseen data.

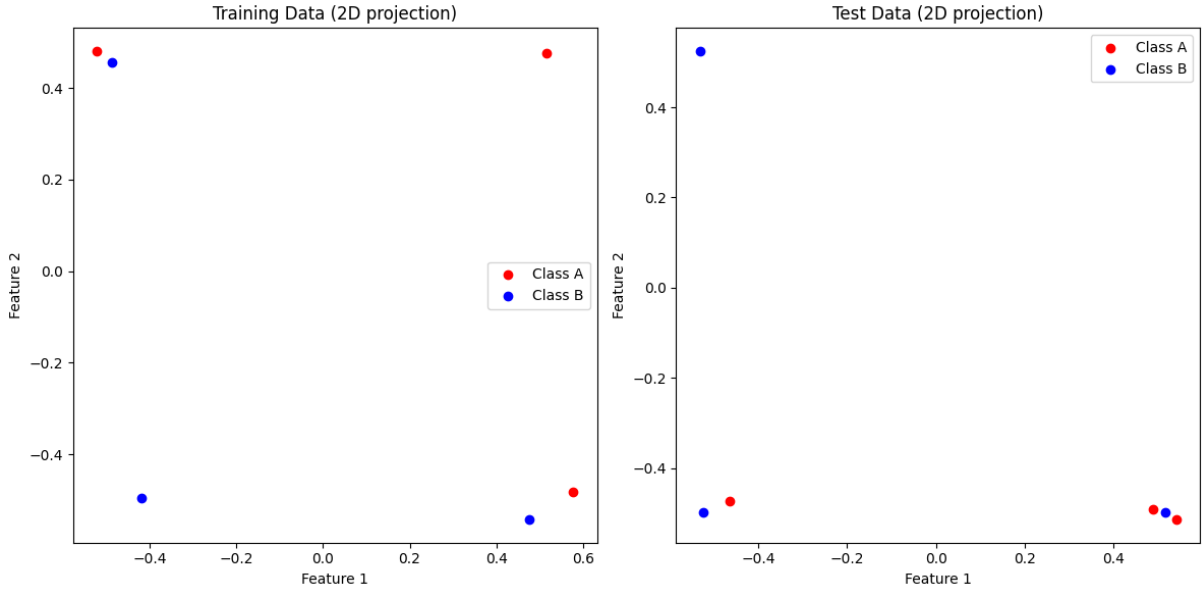

Figure 1: Parity in 3-bits (2D)

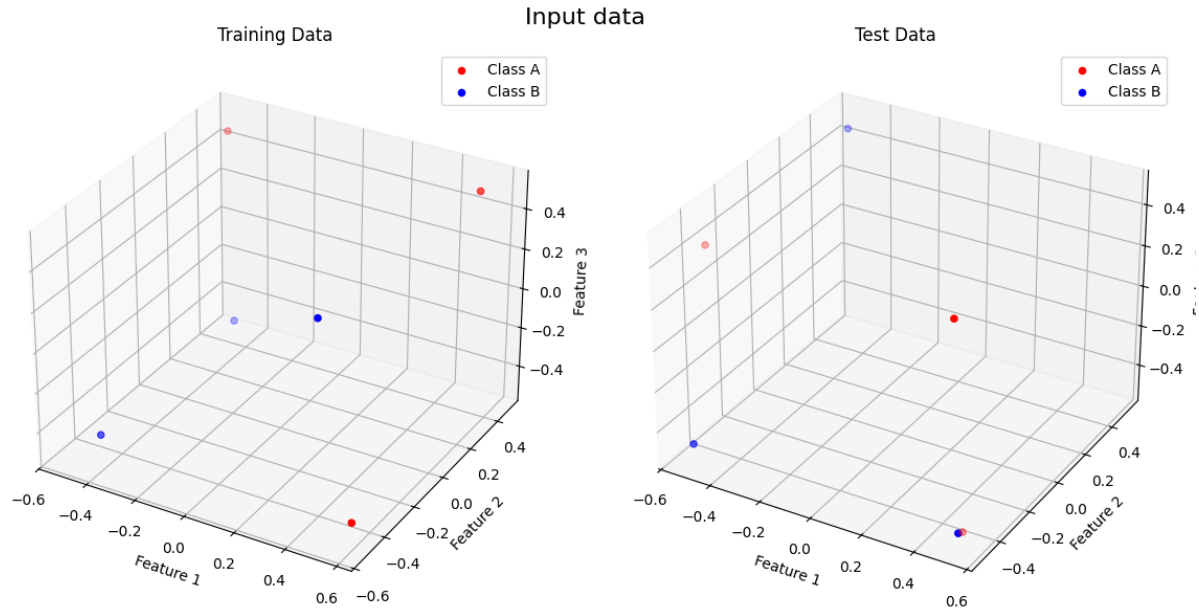

Figure 2: Parity in 3-bits (3D)

2. Building a simple Binary Classifier for the given data

2.1. Design of Circuit (a)

The quantum circuit designed for this task, consisting my first experiment, namely **Circuit (a)**, utilizes **3 qubits** and employs **angle embedding** and **basic entangler layers**. This cirquit is one of the first proposed from the oficial page of PennyLane[1]. The circuit design involves two main components:

- **Angle Embedding**: This step maps the classical input data into quantum states by encoding each input feature into the rotation angles of the qubits. This ensures that the input data is represented in the quantum domain.
- **Basic Entangler Layers**: These layers introduce entanglement between the qubits. The entangler layers consist of a series of controlled-NOT (CNOT) gates interspersed with single-qubit rotations.

The circuit is designed to measure the expectation value of the Pauli-Z operator on the first qubit after processing the input through the variational layers. This expectation value is used as the output of the classifier.

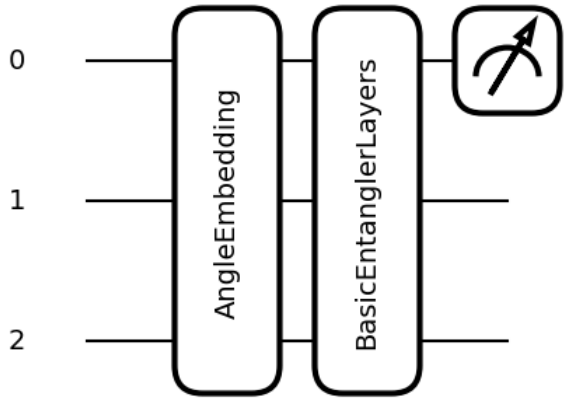

Figure 3: Circuit (a)

2.2. Cost Function

The cost function chosen for this task is the Mean Squared Error (MSE). The MSE is defined as the average of the squared differences between the predicted and true labels. Formally, it is given by:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1}$$

where N is the number of samples, y_i is the true label of the i-th sample, and $\hat{y}_i$ is the predicted label of the i-th sample. In the context of our quantum classifier, the predicted label is obtained by evaluating the expectation value of the quantum circuit output.

[1] https://pennylane.ai/qml/demos/tutorial_variational_classifier/

2.3. Optimization Method

The optimization method selected for training the quantum classifier is the **Nesterov Momentum Optimizer**. From my literature research, I found that this optimizer is known for its ability to accelerate the convergence of the training process by incorporating momentum into the updates of the parameters. The key steps in the optimization process are:

- **Parameter Initialization**: The weights of the variational circuit are initialized with small random values following a normal distribution. The bias is initialized to zero.
- **Batch Training**: The training is performed over **mini-batches** (of size 5) of the training data to efficiently update the parameters. In each iteration, a batch of data is randomly selected, and the parameters are updated to minimize the cost function.

The training is conducted for a specified number of iterations, during which the weights and bias are continuously adjusted to reduce the MSE.

2.4. Classification results

The performance of the classifier is evaluated based on its accuracy on both the training and test datasets. Accuracy is calculated as the proportion of correctly classified samples to the total number of samples. The classifier's predictions are compared against the true labels, and the accuracy is computed for both datasets.

The classifier's accuracy provides an indication of how well the model generalizes to unseen data. A high accuracy on the test data suggests that the model has effectively learned to distinguish between the classes in the parity problem.

Iteration	Train Cost	Train Accuracy	Test Accuracy
1	0.3408	0.5000	0.5000
2	0.2811	0.5000	0.5000
3	0.2867	0.6000	0.3500
4	0.7252	0.5000	0.5000
5	0.3184	0.5000	0.5000
6	0.2816	0.5000	0.5000
7	0.3689	0.5000	0.5000
8	0.2567	0.5000	0.5000
9	0.4822	0.5000	0.5000
10	0.5297	0.4000	0.6500

Table 2: Training and Test Accuracy over Iterations for Circuit (a)

Table 2 shows the results of this procedure. After 10 epochs we get an increased loss and the maximum for this circuit in Test Accuracy, scoring in 0.65. However from the Figure 4 we see an unstable training yielding poor results. This can also be verified from Figure 5 where we can see that this circuit predicts wrongly many points. Our next experiments will focus on chainging the circuit architecture, using alternative layers and techniques.

Figure 4: Convergance of Circuit (a)

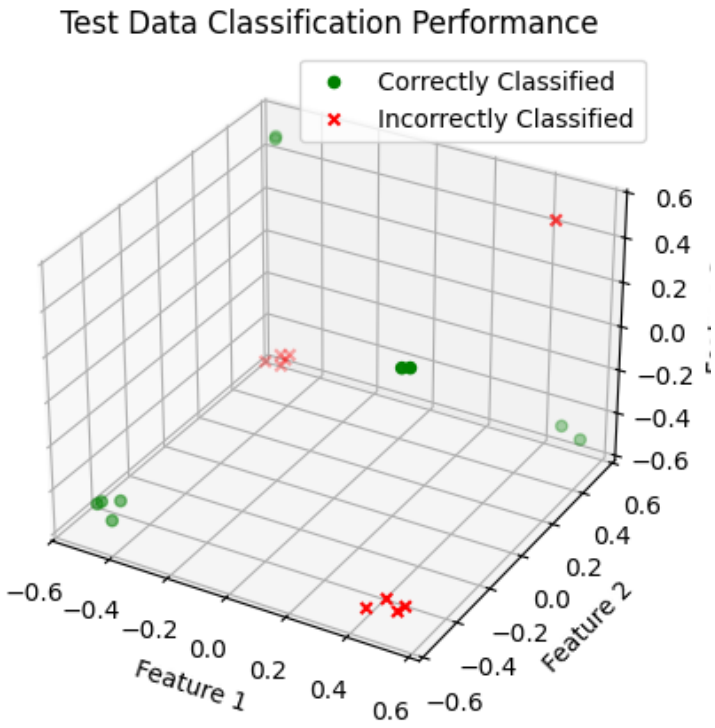

Figure 5: Predictions using Circuit (a)

3. Ansatz Circuit

To explore the performance of different quantum circuits, I experimented with an alternative ansatz circuit. Let's name this circuit as the **Circuit (b)**. The ansatz circuit involves similar components to the initial design but includes variations in the arrangement and types of gates used. The architecture of this circuit can be sheen in Figure 6.

1. **Input Embedding**: Similar to the initial circuit, each input feature is embedded using RZ rotation gates.
2. **Parameterized Rotations**: Each qubit undergoes a sequence of parameterized rotation gates RZ, RY, and RZ to introduce variational parameters.
3. **Entanglement**: The qubits are entangled using a different pattern of CNOT gates.

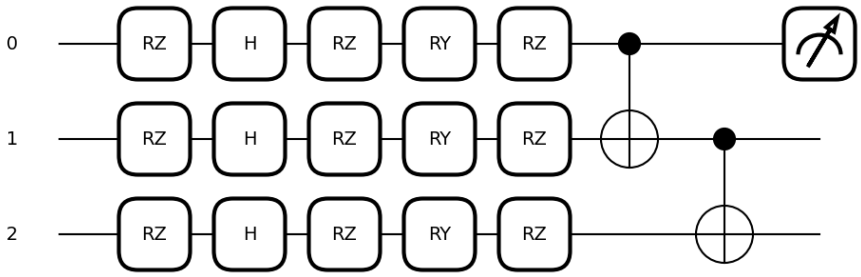

Figure 6: Circuit (b)

4. **Measurement**: The expectation value of the Pauli-Z operator is measured on the first qubit, which provides the output of the circuit.

The alternative ansatz aims to explore different configurations of variational parameters and entanglement patterns to determine their impact on the classifier's performance.

3.1. Training and Evaluation of the Ansatz Circuit

The training and evaluation procedure for the ansatz circuit follows the same steps as described for the initial circuit. The weights are initialized randomly, and the optimization is performed using the **Nesterov Momentum Optimizer**. The performance is evaluated based on the accuracy on both training and test datasets. The cost function used is again MSE.

Iteration	Train Cost	Train Accuracy	Test Accuracy
1	0.3429	0.5000	0.5000
2	0.2691	0.5000	0.5000
3	0.2514	0.5000	0.5000
4	0.2540	0.5000	0.5000
5	0.2658	0.5000	0.5000
...	...	...	...
14	0.2894	0.5000	0.5000
15	0.3029	0.5000	0.5000
16	0.3047	0.5000	0.5000
17	0.2958	0.5000	0.5000
18	0.2853	0.5000	0.5000

Table 3: Training and Test Accuracy over Iterations

As, it can been sheen also from the deliverable code, Table 3 and Figure 7, this circuit yielded only 0.5 accuracy. Meaning that it only predicted either the one category or the other and only. We can see that the Train Loss is decreasing however the Test Accuracy is not increasing. For this reason, this architecture was abandoned. This circuit was inspired from the B.Sc. Thesis, mentioned from the course supervisor[2].

[2] https://pergamos.lib.uoa.gr/uoa/dl/object/3393917/file.pdf

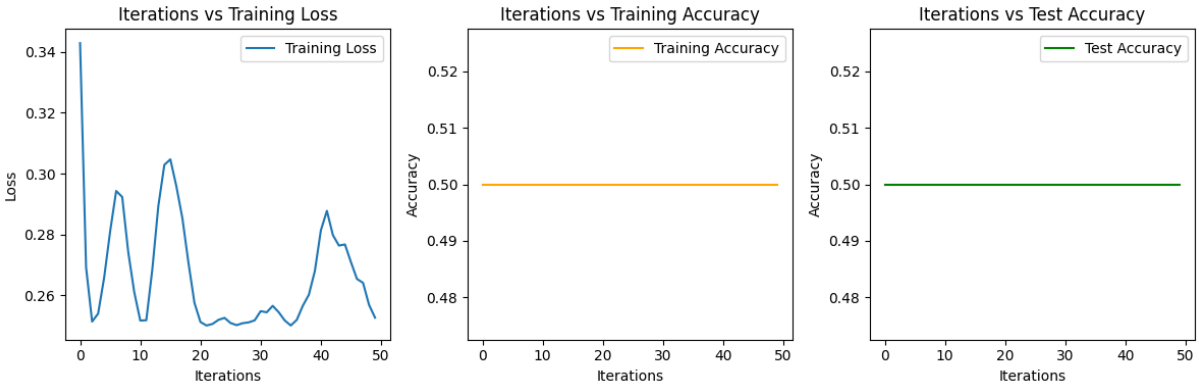

Figure 7: Circuit (b) convergence

4. Data reUploading Circuit

The ansatz circuit designed for this experiment utilizes n qubits, where n corresponds to the number of features in the training data. The circuit employs the data re-uploading technique, which involves embedding the classical input data multiple times throughout the quantum circuit. Let's name this circuit **Circuit (c)**. This approach enhances the expressive power of the quantum model. The key components of the circuit are:

1. **Data Re-Uploading Layers**: The input data is embedded into the quantum circuit using *RY* rotation gates. This embedding is repeated multiple times (specified by the parameter *num_reuploads*) throughout the circuit to increase the circuit's capacity to capture complex patterns in the data.

2. **Parameterized Rotations**: For each qubit, a sequence of parameterized rotation gates *RZ*, *RY*, and *RZ* is applied. These gates introduce variational parameters that are optimized during the training process.

3. **Entanglement Layers**: Controlled-NOT (CNOT) gates are applied between neighboring qubits to create entanglement. This step ensures that the qubits are correlated, allowing the circuit to capture intricate dependencies within the data.

4. **Measurement**: The expectation value of the Pauli-Z operator is measured on the first qubit. This measurement serves as the output of the quantum circuit and is used to make predictions.

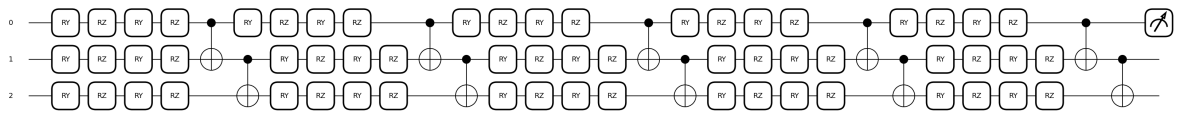

Figure 8: Circuit (c) - Using 5 re-uploads

4.1. Data reUploading Technique

The data re-uploading technique involves repeatedly embedding the classical input data into the quantum circuit at various stages. This method allows the quantum circuit to process the input data multiple times, thereby enhancing its ability to learn complex functions. The technique can be described as follows:

1. **Initial Embedding**: Each input feature is initially embedded using *RY* rotation gates.
2. **Repeated Embedding**: The same input data is re-embedded into the circuit after each layer of parameterized rotations and entanglement operations. This process is repeated for a specified number of layers (*num_uploads*).
3. **Enhanced Expressiveness**: By re-uploading the data multiple times, the circuit gains additional expressive power, enabling it to represent more complex decision boundaries.

4.2. Training and Evaluation Procedure

The training and evaluation procedure for the ansatz circuit follows a similar approach as described previously. The key steps include:

1. **Initialization**: The weights of the variational circuit are initialized with small random values. The bias term is initialized to zero.
2. **Optimization**: The **Adam Optimizer** is used to update the weights and bias iteratively. The training process uses mini-batches of the training data to efficiently update the parameters.
3. **Evaluation**: The classifier's performance is evaluated on both the training and test datasets by computing the accuracy, which is the proportion of correctly classified samples.
4. **Iterations**: The training process is conducted over a specified number of iterations (epochs), during which the parameters are continuously adjusted to minimize the MSE cost function.

4.3. Evaluation

After conducting multiple experiments with the types of circuits, I came across with the PennyLanne tutorial[3] showing the Data re-uploading architecture. So my experiment was to enhance a variational classifier circuit with data re-uploading. This is the best so far architecture, as it yields results up to 1.0 in the Test dataset and at the same time has smooth training curves (Figure 10). The "perfect" classification can also been seen from Figure 9 where all data points have been correctly predicted. Another major remark of this experiment is that Adam Optimizer works great alongside this circuit. Both Table 4 and Figure 10 show the decrease in the MSE loss and at the same time the increase in Test and Train Accuracy.

[3] https://pennylane.ai/qml/demos/tutorial_data_reuploading_classifier/

Iteration	Train Cost	Train Accuracy	Test Accuracy
1	0.4194	0.4500	0.5500
2	0.3766	0.4500	0.5000
3	0.3351	0.4750	0.5500
4	0.3307	0.5500	0.5500
...	...	...	...
8	0.2171	0.6500	0.6500
9	0.1923	0.8000	0.9000
...	...	...	...
21	0.1231	0.9250	0.8500
...	...	...	...
25	0.1426	0.7250	0.6500
26	0.1218	0.8250	0.8500
27	0.0946	0.9250	0.9000
28	0.0836	0.9750	0.9000
29	0.0732	1.0000	1.0000
30	0.0660	1.0000	1.0000

Table 4: Mini-batch Gradient Descent with Adam for Circuit (c)

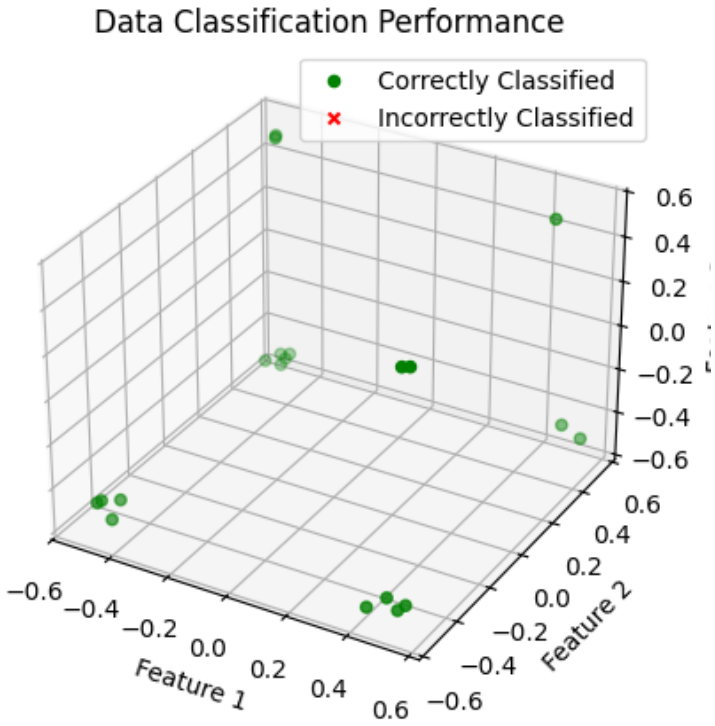

Figure 9: Predictions using Circuit (a)

5. A survey on Optimizers for Circuit (c)

In this section, we evaluate the performance of the best quantum classifier (**Circuit (c)**) using different optimizers. The optimizers considered in this study are Adam, Gradient Descent, Nesterov Momentum, and RMSProp. I experimented with three different learning rates for each optimizer: **0.5, 0.2, and 0.1**. The training was conducted over 50 epochs with a **batch size of 5**. The performance metrics include training loss, training accuracy, and test accuracy.

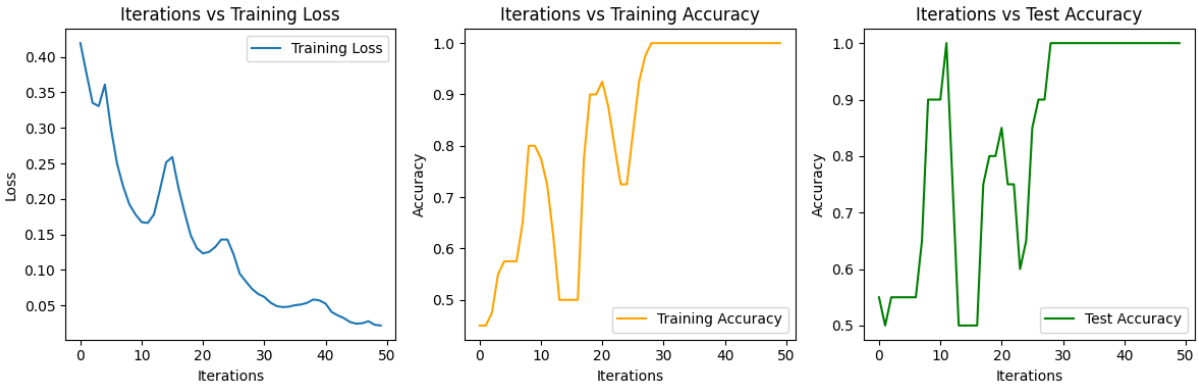

Figure 10: Convergence of Circuit (c)

5.1. Adam Optimizer

The Adam Optimizer combines the advantages of two other extensions of stochastic gradient descent. Specifically, it combines the Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. This optimizer is well-suited for problems that are large in terms of data/parameters or have noisy and/or sparse gradients.

5.2. Gradient Descent Optimizer

The Gradient Descent Optimizer is the standard optimization algorithm used in machine learning. It iteratively adjusts the parameters of the model to minimize the cost function. In each iteration, it updates the parameters by moving in the direction of the steepest descent as defined by the negative of the gradient.

5.3. Nesterov Momentum Optimizer

The Nesterov Momentum Optimizer is a variant of the gradient descent method that accelerates convergence by considering not only the gradient of the cost function but also the change in the gradient. It updates the parameters by taking into account the momentum of the previous updates, which helps in smoothing the optimization path and avoiding oscillations.

5.4. RMSProp Optimizer

The RMSProp Optimizer is another extension of the gradient descent algorithm designed to adapt the learning rate for each parameter individually. RMSProp divides the learning rate by an exponentially decaying average of squared gradients, which helps in handling the diminishing learning rates issue observed in AdaGrad.

5.5. Evaluation

The performance of the quantum classifier with different optimizers and learning rates is summarized in the provided results. The following figure shows the accuracy over epochs for each combination of optimizer and learning rate.

From the results Tables 5, 6,7, and 8 and Figure 11, we observe the following:

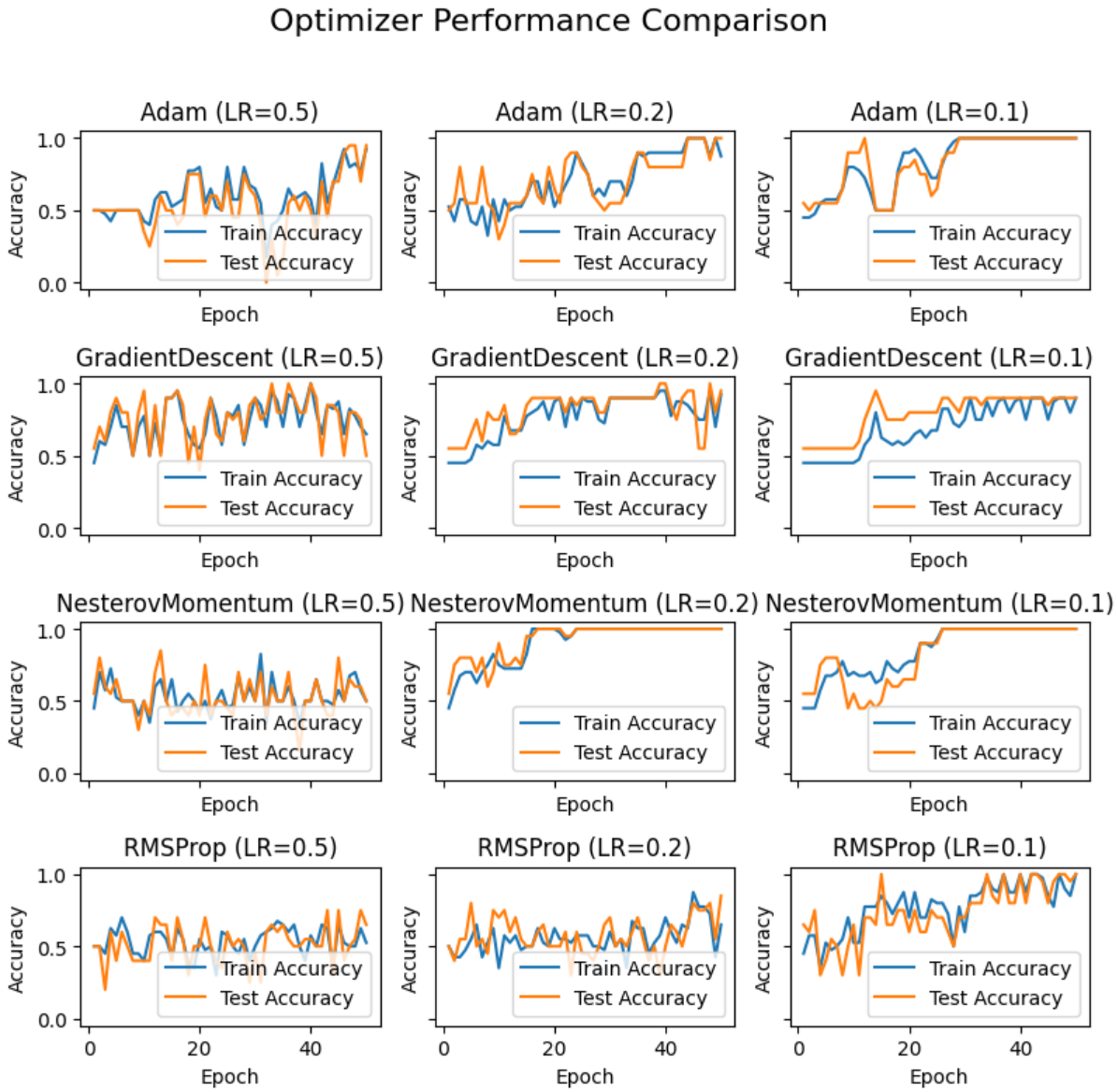

Figure 11: Optimizer Performance Comparison

- **Adam Optimizer**: The Adam optimizer demonstrated stable performance across different learning rates. With a higher learning rate of 0.5, the accuracy improved more rapidly, reaching perfect accuracy by the end of the training. However, lower learning rates also showed consistent improvements.
- **Gradient Descent Optimizer**: This optimizer showed slower convergence compared to Adam but still achieved reasonable accuracy. Lower learning rates (0.2 and 0.1) exhibited smoother accuracy curves, while a learning rate of 0.5 resulted in more fluctuations.
- **Nesterov Momentum Optimizer**: Nesterov Momentum provided a balance between stability and convergence speed. It achieved high accuracy with less fluctuation compared to standard gradient descent, particularly at learning rates of 0.2 and 0.1. For learning rate equal to 0.2 it's the fastest in converge algorithm.
- **RMSProp Optimizer**: RMSProp performed similarly to Adam, showing stable and consistent improvements across different learning rates. Higher learning rates facilitated

faster convergence. This optimizer reaches the lower accuracies in comparison with the others.

Overall, the Adam and Nesterov optimizers performed the best in terms of both stability and convergence speed, making them suitable choices for this quantum classification task. Gradient Descent and RMSProp also provided reasonable performance, but with more sensitivity to the choice of learning rate.

Epoch	Optimizer	Learning Rate	Train Loss	Train Accuracy	Test Accuracy
44	Adam	0.2	0.0704	1.0000	1.0000
45	Adam	0.2	0.0791	1.0000	1.0000
...	...	...	...	...	...
49	Adam	0.2	0.0892	1.0000	1.0000
50	Adam	0.2	0.0861	0.8750	1.0000
12	Adam	0.1	0.1661	0.7250	1.0000
29	Adam	0.1	0.0732	1.0000	1.0000
...	...	...	...	...	...
49	Adam	0.1	0.0227	1.0000	1.0000
50	Adam	0.1	0.0216	1.0000	1.0000

Table 5: Adam Optimizer Results

Epoch	Optimizer	Learning Rate	Train Loss	Train Accuracy	Test Accuracy
33	GradientDescent	0.5	0.1060	0.9500	1.0000
36	GradientDescent	0.5	0.1049	0.9250	1.0000
40	GradientDescent	0.5	0.0897	1.0000	1.0000
39	GradientDescent	0.2	0.1136	0.9500	1.0000
40	GradientDescent	0.2	0.1109	0.9500	1.0000
48	GradientDescent	0.2	0.0983	0.9500	1.0000

Table 6: GradientDescent Optimizer Results

Epoch	Optimizer	Learning Rate	Train Loss	Train Accuracy	Test Accuracy
17	NesterovMomentum	0.2	0.0693	1.0000	1.0000
18	NesterovMomentum	0.2	0.0387	1.0000	1.0000
...	...	...	...	...	...
49	NesterovMomentum	0.2	0.0044	1.0000	1.0000
50	NesterovMomentum	0.2	0.0040	1.0000	1.0000
26	NesterovMomentum	0.1	0.0739	1.0000	1.0000
27	NesterovMomentum	0.1	0.0737	1.0000	1.0000
...	...	...	...	...	...
49	NesterovMomentum	0.1	0.0101	1.0000	1.0000
50	NesterovMomentum	0.1	0.0102	1.0000	1.0000

Table 7: NesterovMomentum Optimizer Results

Epoch	Optimizer	Learning Rate	Train Loss	Train Accuracy	Test Accuracy
15	RMSProp	0.1	0.1451	0.8500	1.0000
34	RMSProp	0.1	0.0852	0.9750	1.0000
...	...	...	...	...	...
48	RMSProp	0.1	0.0997	0.9000	1.0000
50	RMSProp	0.1	0.0958	1.0000	1.0000

Table 8: RMSProp Optimizer Results

6. The Parity Problem with 5 inputs

The parity problem with 5 inputs involves determining whether the number of ones in a binary string of length 5 is odd or even. The output is 1 if the number of ones is odd and 0 if it is even. A preview of the given test and train data can be seen in the bellow visualization (Figure 12).

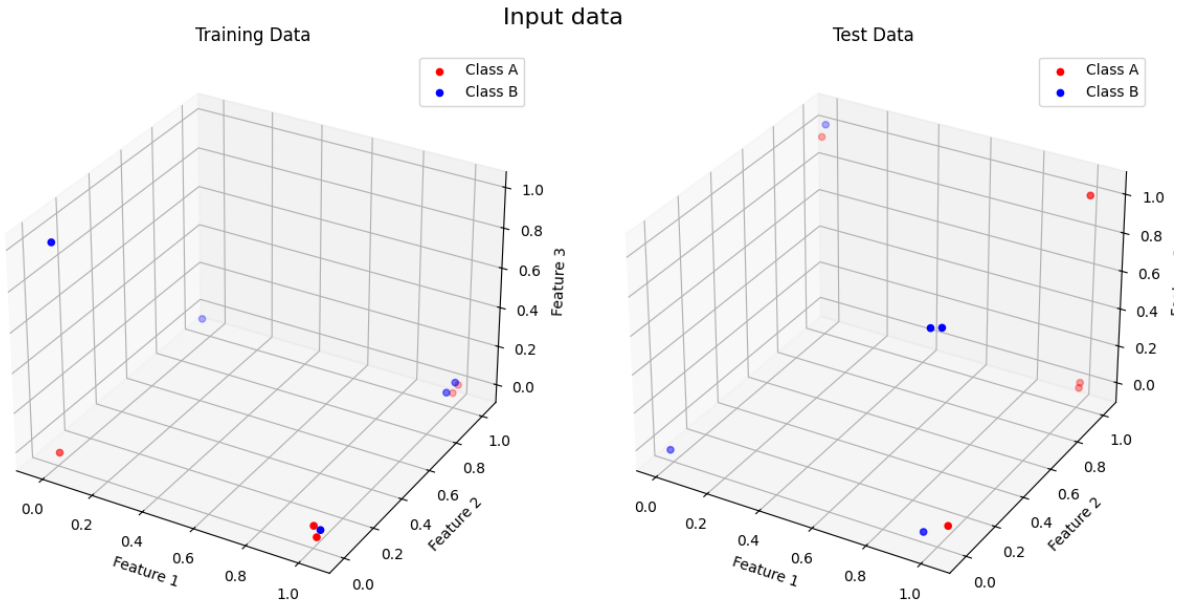

Figure 12: Parity-5 3D Visualization

6.1. Standard Scaling

After reading the data, I performed Standard scaling, also known as z-score normalization, that is a method used to standardize the features of a dataset. This technique ensures that each feature contributes equally to the learning process by transforming the features to have a mean of zero and a standard deviation of one. The formula for standard scaling is given by:

$$z = \frac{x - \mu}{\sigma} \tag{2}$$

where:

- x is the original value of the feature.
- μ is the mean of the feature.
- σ is the standard deviation of the feature.

By applying this transformation, I ensure that the features are on the same scale, which can improve the performance and convergence rate of many machine learning algorithms, including both classical and quantum classifiers.

6.2. Circuit Description

The quantum circuit I used for this experiment (Figure 13) consists of **5 qubits** and employs a data re-uploading technique with **10 re-uploading layers**. Each re-uploading layer embeds the input data using *RY* rotation gates followed by parameterized *RZ* and *RY* gates. Entanglement is introduced between neighboring qubits using CNOT gates. The circuit is optimized using a variational approach, similar to previous experiments.

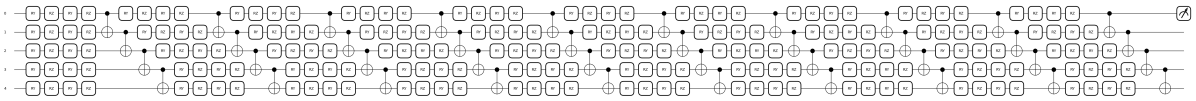

Figure 13: Circuit for Parity-5 with 10 re-uploads

6.3. Early Stopping Technique

Early stopping is a regularization technique used to prevent overfitting during training. It involves monitoring the training loss and stopping the training process when the loss stops improving for a specified number of consecutive epochs (patience). This helps in achieving a balance between underfitting and overfitting, ensuring that the model generalizes well to unseen data.

6.4. Performance Evaluation

The results for each training iteration are presented in the following table:

Iteration	Train Cost	Train Accuracy	Test Accuracy
1	0.4874	0.5312	0.5667
2	0.4196	0.5188	0.5500
3	0.3502	0.5188	0.5833
4	0.2935	0.5813	0.6000
5	0.2836	0.5375	0.5667
6	0.2747	0.4813	0.6000
7	0.2672	0.5250	0.6500
8	0.2680	0.6000	0.6667

Table 9: Training and Test Accuracy over Iterations with Early Stopping

The quantum classifier achieved a test accuracy of 0.6667, indicating its ability to learn the parity problem partially. However that was the perfect score I achieved from multiple different experiments.

6.5. Comparison with a Classical Neural Network

To benchmark the performance of the quantum classifier, I compared it with a classical neural

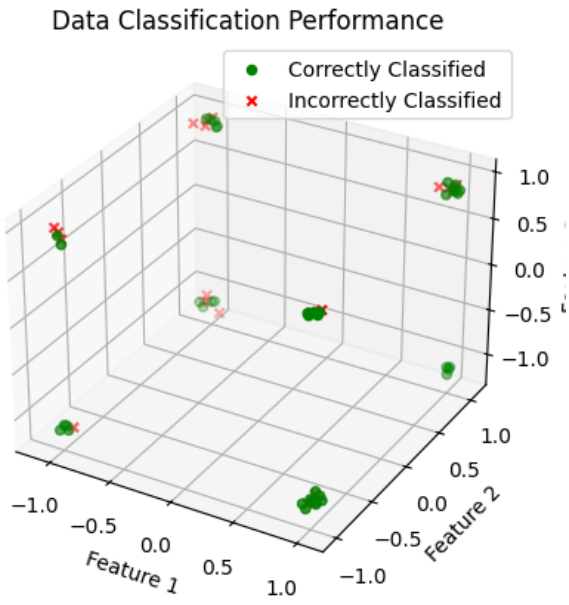

Figure 14: Predicted points

network (NN) using the same dataset. The classical NN was trained using the MLPClassifier from scikit-learn with a single hidden layer of 10 neurons. The training results are as follows:

```
Classical NN | Train Accuracy: 0.9125 | Test Accuracy: 0.9500
Number of parameters | Quantum NN: 151 | Classical NN: 71
```

The classical NN outperformed the quantum classifier, achieving higher training and test accuracies. The classical NN's test accuracy of **0.95** is significantly higher than the quantum classifier's test accuracy of 0.6667. Additionally, the classical NN required fewer parameters (71) compared to the quantum NN (151). The results indicate that while the quantum classifier demonstrates potential, the classical NN remains more effective for this particular task.

7. Deliverables

For this project I have implemented:

- **qml_classifier_a_model.ipynb**
  - Circuit (a) implementation and results (my first experiment)
- **qml_classifier_b_experiments_circuits.ipynb**
  - Circuit (a), (b), (c) implementation and results for all these
- **qml_classifier_b_best_model.ipynb**
  - Circuit (c) implementation and results
  - Optimizers survey code and results
- **qml_classifier_c.ipynb**
  - Circuit for Parity-5 implementation and results

8. Comments

I didn't follow the structure of the description and preferred to implement this project as a mini-survey. So the delivered tasks are a little bit mixed up, hope this won't be a problem.