QRC-4-ESP

QUANTUM RESERVOIR COMPUTING
FOR EFFICIENT SIGNAL PROCESSING

Project Deliverable

D4.1 Software implementation of neural network for QRC

Lead beneficiary:	Justinmind
Author(s):	Enrique Cano Ayesa, Xavier Renom Portet
Contributor(s):	All consortium partners
Date of issue:	17/12/2024
Dissemination level:	Public (PU)

https://www.qrc-4-esp.eu

Funded by
the European Union

QRC-4-ESP

DOCUMENT HISTORY

Version and date	Changes
1.0 - 17/12/2024	Initial version

DISCLAIMER

Funded by the European Union. Views and opinions expressed are, however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

This document contains information which is proprietary to the QRC-4-ESP consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or parts, except with the prior written consent of the QRC-4-ESP coordinator or partner on behalf of the project consortium.

Table of Contents

1. Executive Summary

This deliverable, D4.1, represents the first milestone in Work Package 4 (WP4), showcasing the software implementation of a neural network designed for Quantum Reservoir Computing (QRC) benchmarking. The neural network integrates a Quantum Reservoir (QR) as its first, non-trained layer, offering a novel computational approach to signal processing and other tasks such as regression analysis and time series prediction. We have introduced multiple Neural Networks (NN) for different tasks to benchmark the QR+NN architecture along the WP4. The deliverable highlights the implementation process, software architecture of this hybrid QR and neural network.

The QR, which leverages quantum states for computation, significantly increases the dimensionality and nonlinearity of the input space without additional training overhead. The goal of WP4 is to work on the advantages of combining quantum computation with classical machine learning. The neural network has been tested using synthetic data and preliminary experimental outputs. Benchmarking studies will be done in future works to compare the performance of the QR-integrated network against a similar neural network without the QR layer, focusing on metrics such as mean squared error (MSE) and normalized MSE (NMSE).

This document outlines the network's design, implementation, and testing process. Additionally, it explores potential applications for QR integration to NN in the WP4.

2. Introduction

2.1. Objectives of the Deliverable and initial Demonstrator

The primary goal of D4.1 is to explain a software-implemented neural network that processes data using a Quantum Reservoir as its initial layer. This integration aims to enhance computational efficiency and predictive accuracy, highlighting the potential of QRC in real-world applications. Additionally, we are going to introduce various types of neural networks tailored for different tasks, such as classification, clustering, and other machine learning applications. These models will be used to demonstrate the versatility and adaptability of neural network architectures in leveraging Quantum Reservoir Computing for diverse computational challenges along the WP4. By integrating QRs into existing architectures, this work aims to reduce computational overhead while maintaining or improving predictive accuracy.

2.2. Overview of WP4 and Importance of Quantum Reservoir Computing (QRC) in Neural Networks

Work Package 4 focuses on integrating, demonstrating, and benchmarking QRC systems. By building on theoretical and experimental developments from previous work packages, WP4 targets the practical implementation of QRC systems in feed-forward neural networks and other architectures like LSTM.

As WP4 progresses, future milestones will aim to integrate QRs into more complex neural architectures, such as recurrent and convolutional models. Additionally, WP4 will explore real-world applications of QR-enhanced networks in quantum communication, advanced sensor networks, and other high-impact areas.

3. QRC Numerical model and Neural Network integration

A numerical model of a neural network designed for processing the output of a Quantum Reservoir has been developed. This model will bridge the gap between quantum-based data generation and traditional neural network processing, ensuring seamless integration. Corresponding software will also be developed, enabling the testing and validation. This dual approach ensures robustness and reliability, providing insights into the practical applications and limitations of the proposed solutions.

The numerical model developed for the neural network processing of Quantum Reservoir (QR) outputs builds on the Hamiltonian formulation described in D1.1. Specifically, the Hamiltonian governing the superconducting quantum reservoir incorporates the interaction between transmon qubits and a resonator, leveraging their coupled dynamics to encode data. This encoding process transforms input time-series data into high-dimensional quantum states, which are then utilized as features for the neural network.

In D1.1, the use of eigenfrequency calculations to model the quantum reservoir's response to input signals provides the foundation for optimizing the QR design. By defining the detunings between signal, qubit, and cavity frequencies, and accounting for coupling parameters, the QR model achieves a nonlinear transformation of input data. This feature is critical for the neural network's ability to handle complex regression and other tasks like classification or clustering. For more information we refer to the Deliverable at D1.1.

3.1. Design and Architecture

The neural network is structured with a QR simulator as its initial layer, followed by fully connected dense layers. The QR layer functions as an untrained computational substrate, encoding input data into high-dimensional quantum states. Subsequent layers, implemented in a classical neural network framework, process the QR's outputs to generate the predictions.

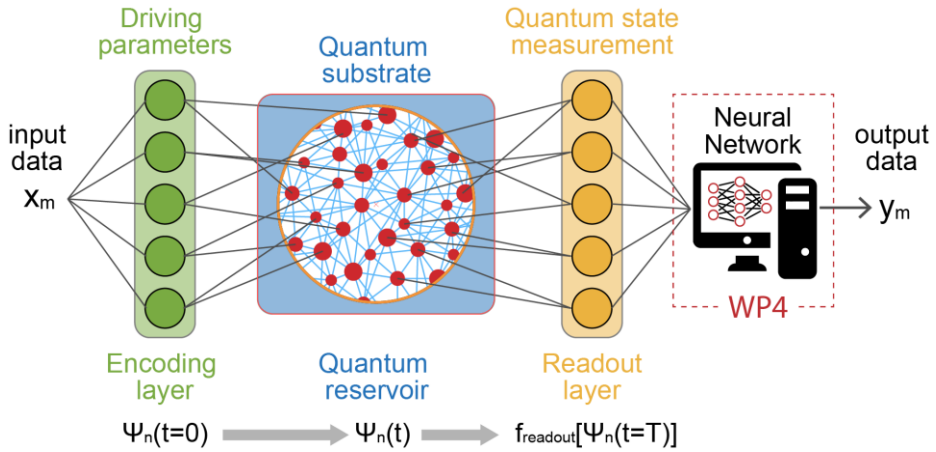

Figure 1: Schematic representation of the QRC protocol. WP4 focuses on developing the neural network part of the reservoir computing pipeline..

The developed neural network integrates the quantum reservoir as a foundational layer, inspired by the theoretical modeling in D1.1. Data encoding is achieved through modulation of the Hamiltonian parameters, ensuring a nonlinear transformation of input data. In D1.1, this encoding method was shown to enhance the separability of data in the feature space, a property exploited in the neural network's design.

The architecture is designed to be adaptable, allowing integration with task-specific modules such as convolutional layers for image recognition or recurrent layers for sequential data processing. This modular design ensures scalability and flexibility for diverse machine-learning applications.

3.2. Integration of the Quantum Reservoir Simulator

The QR is configured using a Python-based simulator that models qubits and their interactions. Input data is encoded into the quantum substrate by modulating system parameters such as detuning and Rabi frequency. Measurements from the QR layer, corresponding for example to the expectation value of the population operator, serve as input features for the neural network.

Further enhancements include refining data encoding processes by introducing dynamic parameter modulation to better adapt the QR's output to various data types, to achieve the goals of a better performance. This is the first version developed in the WP4 and will be updated along with the developments in this WP.

4. Implementation Details and Python Code

The network is implemented using Python and TensorFlow [1], leveraging QuTiP [2] for QR+NN simulations.

We have used TensorFlow for previous developments and internal experience, but we probably switch to PyTorch [3] in future developments (still to be decided), because the last developments for LLM and Generative AI have transitioned to PyTorch, even those like OpenAI that at the beginning preferred to use TensorFlow [4].

We use our library created to extend QR computation **qrc_qutip** developed in the project. The QR substrate operates with predefined physical parameters (e.g., number of qubits, coupling strengths) and evolves over time to produce dynamic states. The codebase includes functions for encoding data, configuring the QR, training the network on regression tasks, and finally, performing testing and and drawing the results.

4.1. Python Code Overview

Key sections of the code include:

- **QR Setup**: Configures the quantum substrate with adjustable physical parameters.
- **Data Encoding**: Maps input data to quantum states.
- **Readout**. Introduction the QR output as the input to the Neural Network.
- **Neural Network Training**: Uses TensorFlow to train the dense layers on QR outputs.
- **Evaluation (prediction) and Benchmarking**: Compares predictions against ground truth using MSE and NMSE metrics.

The code provided presents an implementation of a machine-learning architecture that combines a Quantum Reservoir (QR) with a Neural Network (NN) for time-series prediction tasks. The quantum reservoir acts as a non-trained feature extractor, transforming input data into a higher-dimensional space that the neural network can leverage for learning. The model predicts future values of a time series based on this processed information. This architecture uses the quantum system to enhance the representation of data, providing the neural network with a more expressive set of features.

4.2. QR Setup and Configuration

The quantum reservoir is initialized with a set of configurable physical parameters, including detuning, Rabi frequency, and coupling strengths between the qubits. These parameters define how the quantum system behaves and evolves over time. The **QR_Qutip** class in **qrc_qutip** library is responsible for simulating the interactions between qubits, with time-evolution governed by these parameters as introduced in D1.1. The reservoir's states evolve dynamically, reflecting the quantum system's response to the input signal. In this setup, random fluctuations are introduced to parameters like detuning and Rabi frequency, which adds variability to the system's dynamics, enabling it to handle complex and non-linear data patterns. The quantum reservoir produces a time series of reservoir measurements, stored in matrices (*ws_train* for training and *ws_test* for testing) that will later serve as inputs to the neural network. This mechanism enables the quantum system to encode temporal dependencies within the input data and maintain memory across time steps.

4.3. Data Encoding into Quantum States

The architecture maps the input data to the quantum states of the reservoir by encoding the data into the system's parameters. The input signal, which could be a sum of random sinusoids or a standard Mackey-Glass time-series (for the time-series simulation), interacts with the quantum system, affecting the evolution of the qubits' states. These interactions are used to modulate the quantum

system's evolution, resulting in a transformation of the input data into a set of features defined by the quantum states. The system's states at each time step are captured and stored, with the final reservoir states representing the processed features that the neural network will use. Feedback mechanisms ensure that the reservoir maintains temporal memory, continuously updating its states and incorporating past information into future predictions.

4.4. Readout of Quantum States

The readout process is a critical step in preparing the quantum reservoir outputs to be fed into the neural network. After the quantum reservoir has evolved in response to the input data, the final quantum state (or the state at each time step) needs to be extracted to form a usable feature vector. This is done by selecting specific components of the quantum state—such as the populations or amplitudes of the qubits—at the end of each time step.

In the code, the readout process occurs during the loop that evolves the quantum reservoir for both training and testing data. At each time step, the output state of the reservoir is recorded into the matrix *ws_train* (for training data) or *ws_test* (for testing data). Specifically, the quantum state at the last time step of the evolution is extracted, representing the reservoir's final state. This final state is what serves as the feature vector for the neural network. Feedback from the neural network to the quantum reservoir is also captured, ensuring that the reservoir's states evolve in a memory-driven manner, accounting for prior input information.

Once the quantum states are collected, they are organized into a feature matrix, which will then be used as input to the neural network. The readout procedure ensures that the neural network receives the processed, high-dimensional representations of the data, enabling it to learn complex mappings and temporal relationships.

4.5. Neural Network Training

Once the quantum reservoir's states are obtained, they are used as the input to a neural network for further processing. The first neural network developed is a multi-layer perceptron (MLP) for linear regression, and the subsequent NN are listed in the 'Multiple Neural Networks for QR-NN benchmarking' section where the output of the quantum reservoir forms the input to the first layer. The network consists of hidden layers, where the features from the quantum system are transformed to predict future values of the time series. In this architecture, the network learns the mapping between the quantum features and the target values, which represent the next value in the time series.

As already explained, the neural network is trained using TensorFlow, with the loss function set to mean squared error (MSE), and the Adam optimizer used for weight updates. During training, the model learns to minimize prediction error by adjusting the weights and biases through backpropagation. A validation split is used to monitor the model's performance on unseen data during training, helping prevent overfitting and ensuring that the model generalizes well to new, unseen data.

4.6. Evaluation (prediction) and Benchmarking

After training, the model's performance is evaluated on a separate test set. The mean squared error (MSE) and normalized mean squared error (NMSE) are calculated to assess how well the model's predictions match the actual values. The MSE measures the average squared difference between predicted and true values, while the NMSE provides a normalized performance metric by comparing the error to the variance of the target data. These metrics help quantify the model's accuracy and indicate how well the quantum reservoir is improving the prediction task. The evaluation results also help determine if further tuning of the model or adjustments to the quantum system could improve performance.

4.7. Extensions and Future Work

The Python implementation is designed to be modular, allowing easy extensions and modifications for future work and developed in Jupiter Notebooks [5]. The quantum reservoir can be enhanced by incorporating more complex qubit configurations or introducing adaptive learning rates to the neural network. Additionally, the system could be expanded to handle multi-dimensional time series data, integrate other types of quantum models (with different redouts), or improve training efficiency using techniques like dropout or batch normalization. These potential improvements would enable the architecture to address a broader range of complex, non-linear problems, opening up applications in various fields such as finance, physics, and machine learning itself. The architecture's flexibility makes it well-suited for future research and development in quantum-classical hybrid models.

5. Software Testing and Validation

The MLP (Multilayer Perceptron, the 1st feedforward NN we developed) and also the LSTM were tested on both synthetic and experimental datasets. Synthetic data included sinusoidal signals and random noise, while experimental data was derived from QR prototypes developed in other work packages. The validation process in D4.1 incorporates experimental insights from D1.1, where the reservoir's ability to process noisy and complex inputs was developed.

In D4.1, synthetic datasets, such as sinusoidal signals and random noise, are employed alongside experimental QR outputs to validate the network's predictions will be included along WP4. D1.1's results showed a strong correlation between theoretical models and experimental data, ensuring that the QR-enhanced network maintains high predictive accuracy across diverse scenarios, to ensure proper transition to the real physical reservoirs when they will be developed and available along the project.

The testing framework also incorporates the reservoir's response to parameter variations, as explored in D1.1. This approach ensures that the neural network is not only accurate but also adaptable to different operational environments.

6. Multiple Neural Networks for QR-NN benchmarking

We have introduced initially basic Neural Networks for **Linear Regression** and **Classification** as the initial developments because at the beginning the Quantum Reservoir software simulator has been tested with a Linear Regression testing as the output layer in D1.1. We are introducing several other Neural Networks for benchmarking the overall configuration in very different tasks.

6.1. Time-series example

This Quantum Reservoir Computing (QRC) architecture leverages the dynamics of a system of N=5 Rydberg atoms to perform time-series prediction. The quantum reservoir evolves over a fixed time interval. The input time-series is a sum of several sinusoidal functions with random frequencies and phases, split into training (60%) and testing (40%) datasets.

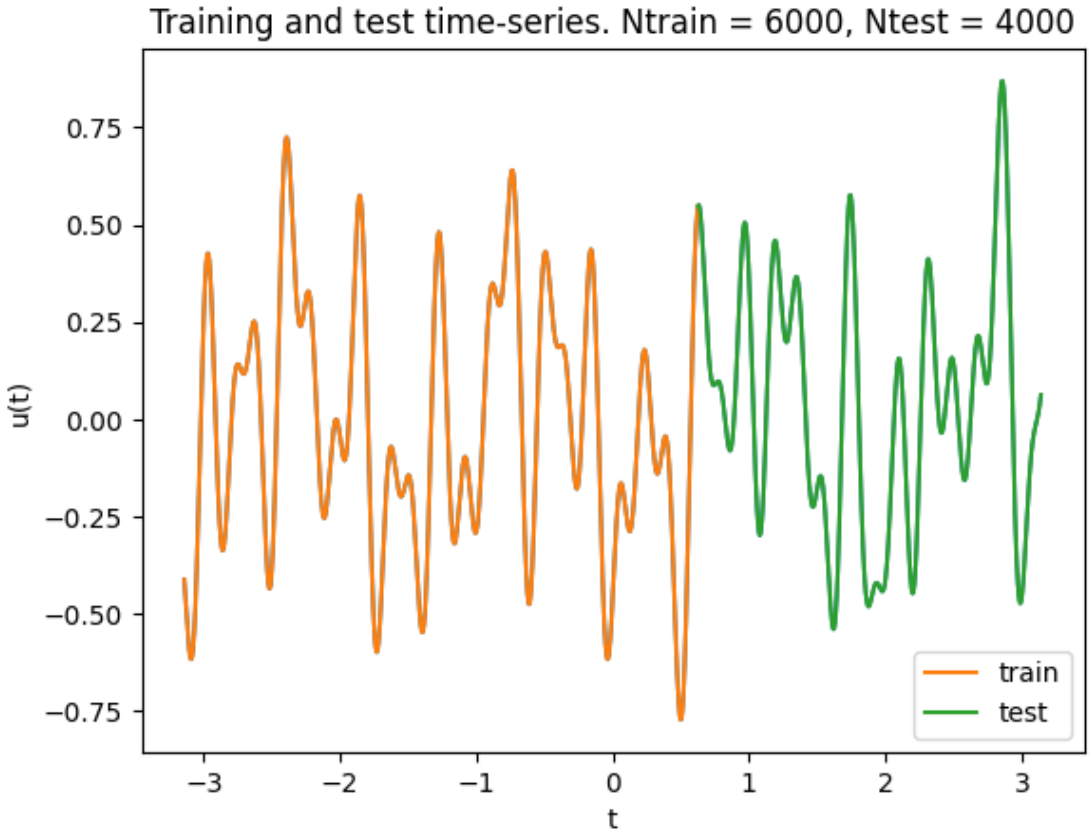

Figure 2: Benchmark dataset used in this report. The time-dependent signal is composed of a sum of 5 random sinusoidals. The time-series is split into training (orange line) and test (green line).

At each training time step, the input signal is mapped to the quantum reservoir by modulating the detuning and Rabi frequency. The system evolves, and the populations of the quantum states are recorded. These final states of the reservoir's evolution serve as the design matrix, while the target values represent the desired future states of the time-series.

6.2. Linear Regression (simple ridge regression)

Ridge regression with cross-validation (RidgeCV) learns a linear mapping from the reservoir states to the target outputs. The performance on the training set is evaluated using the Mean Squared Error (MSE) and the Normalized Mean Squared Error (NMSE), with plots showing predicted versus true values and the corresponding error distribution. In the test phase, the reservoir autonomously predicts

future states by feeding its previous output back as input. The test predictions are compared to the ground truth, and error metrics are calculated. Several plots visualize the system's dynamics, including the evolution of atomic states, the design matrix, and prediction performance. The reservoir's ability to transform the input data into a high-dimensional space allows the linear regression layer to capture complex patterns effectively, demonstrating the potential of quantum reservoirs for time-series prediction tasks.

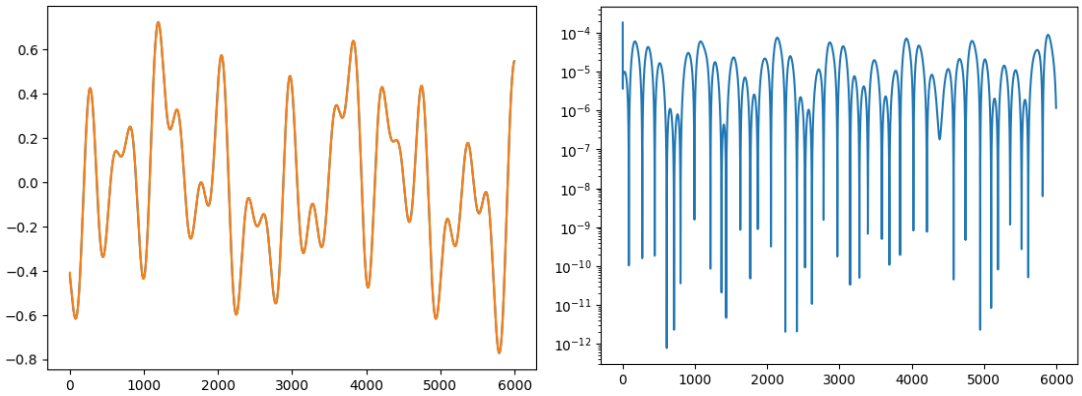

Figure 3: (left) Comparison of True (blue line) and Predicted (orange line) Values for Training Data. (right) Log-scale plot of the Mean Square Error (MSE) for the training data.

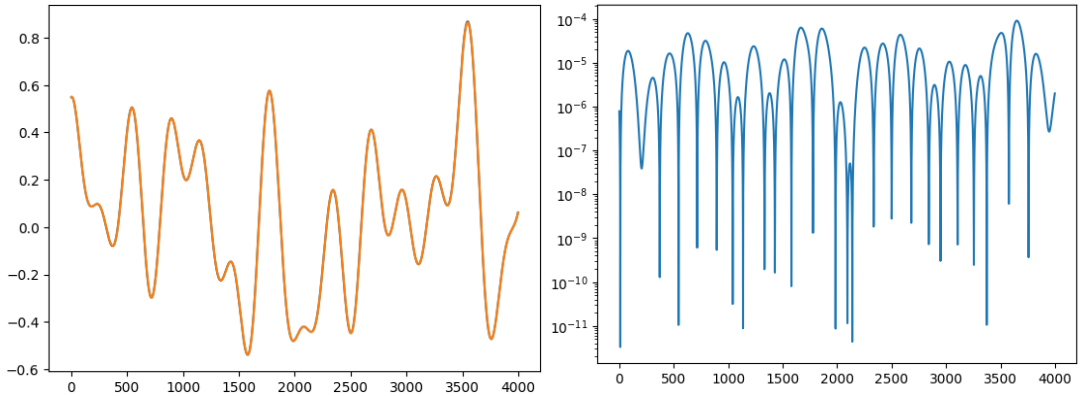

Figure 4: (left) Comparison of True (blue line) and Predicted (orange line) Values for Test Data. (right) Log-scale plot of the Mean Square Error (MSE) for the Test data.

6.3. Simple MLP (Multi-Layer Perceptron)

The Neural Network is composed of a sequential architecture with an input layer matching the features of the training data, two hidden layers with 64 and 32 units respectively, both using ReLU activation, and a single linear output layer for regression. It is compiled using the Adam optimizer, mean squared error (MSE) as the loss function, and mean absolute error (MAE) as a metric. The training data is split into 80% training and 20% validation sets to monitor performance. The network is trained for 50 epochs with a batch size of 32, leveraging the validation data to track generalization throughout training. Find the code in the appendix section.

Training and test time-series.

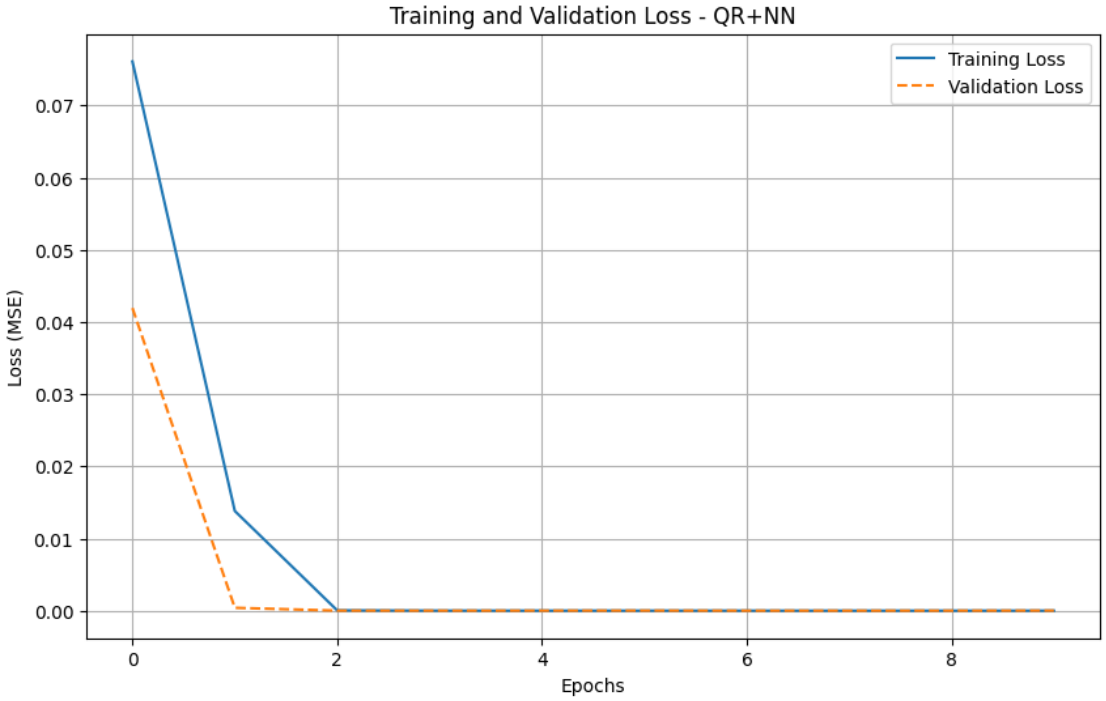

Figure 5: Plot Training and Validation Loss

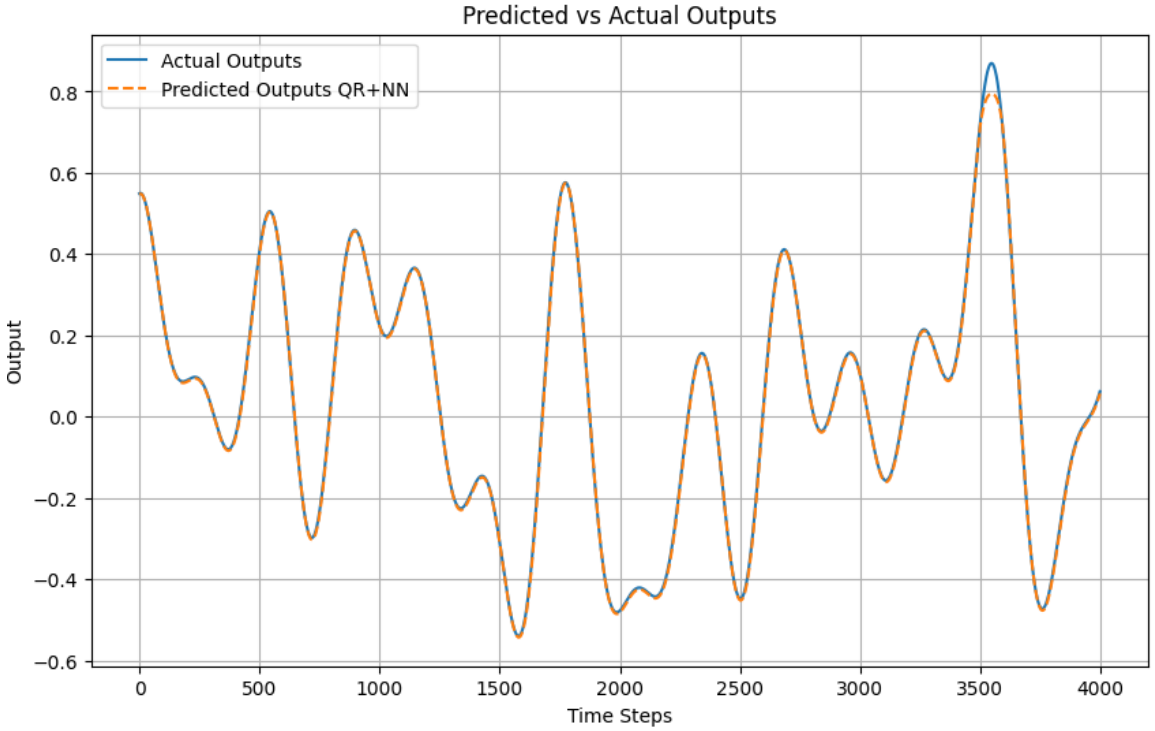

Figure 6: Plot Predicted vs Actual Outputs

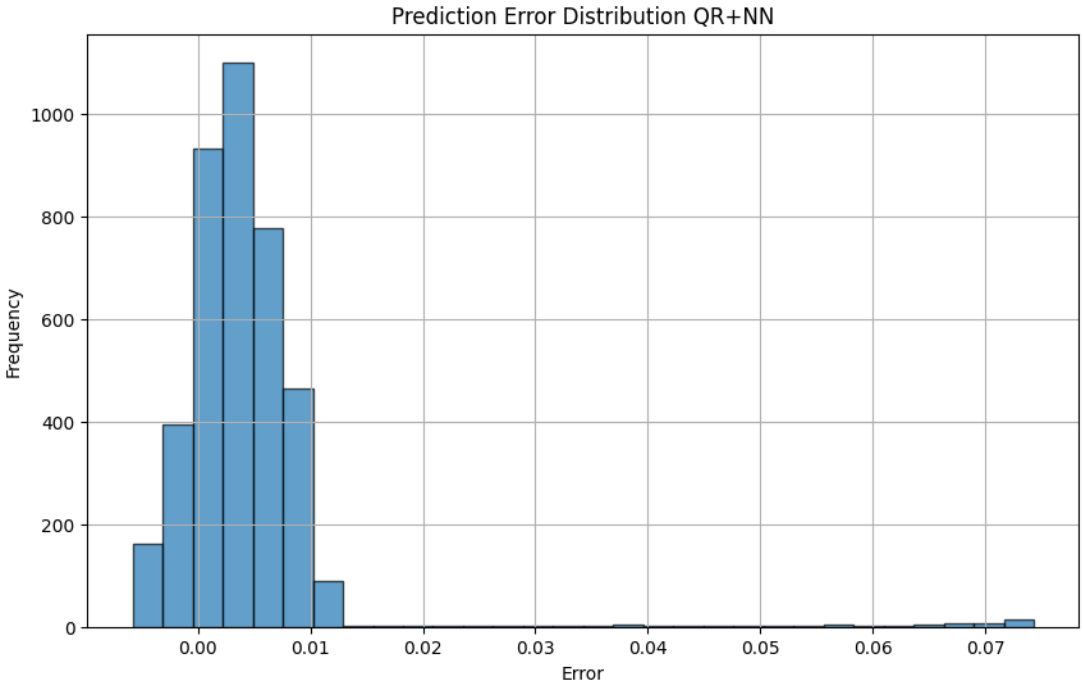

Figure 7: Plot Error Distribution

6.4. Time-Series Forecasting (LSTM)

The neural network in this case focuses on time-series forecasting using synthetic quantum reservoir outputs as input. It employs a Sequential architecture with an LSTM layer for capturing temporal dependencies, followed by dense layers for output predictions. The data in the example consists of sinusoidal signals with added noise to simulate time-series patterns, processed as single-feature inputs. The network is compiled with the Adam optimizer and a loss function like mean squared error, aiming to minimize forecasting errors during training. This setup enables effective modeling of sequential patterns in noisy time-series data. The main goal in this case is predicting future values based on historical data. For example for forecasting stock prices, weather or predicting energy consumption. We have choose the simplest option, a LSTM, other architecture could be the Transformer model, that is the base of LLM Generative AI. We use the same time series, in previous section.

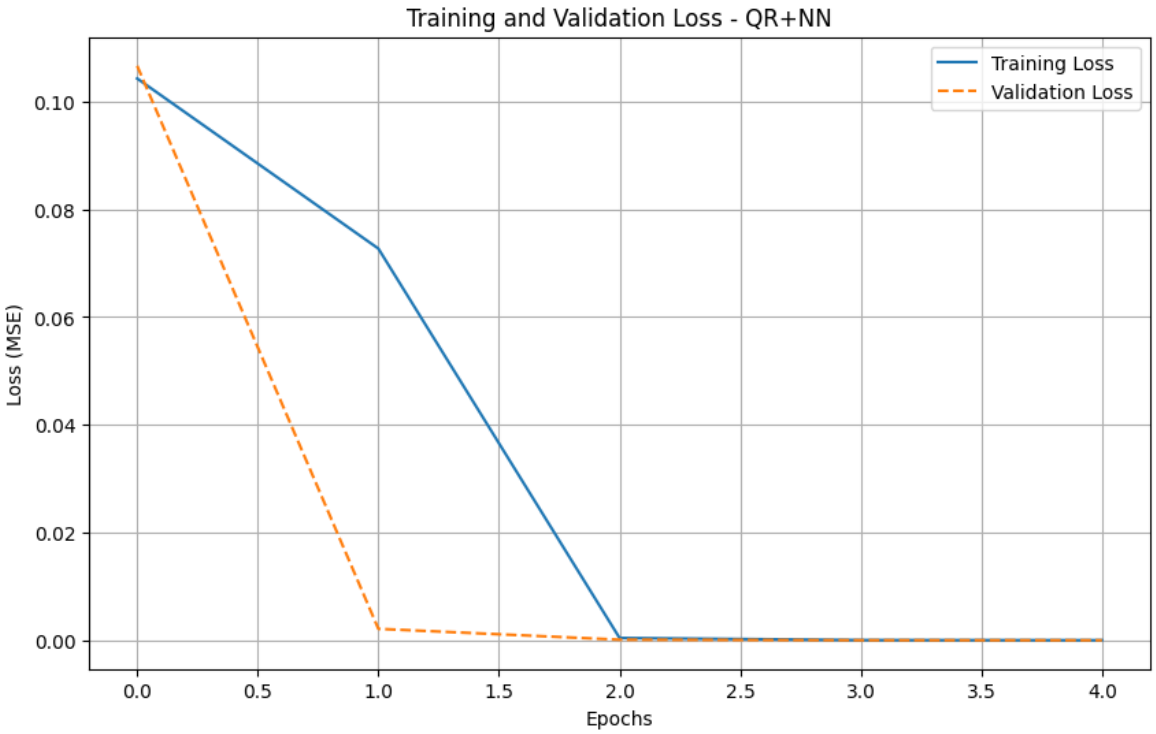

Figure 8: Plot Training and Validation Loss

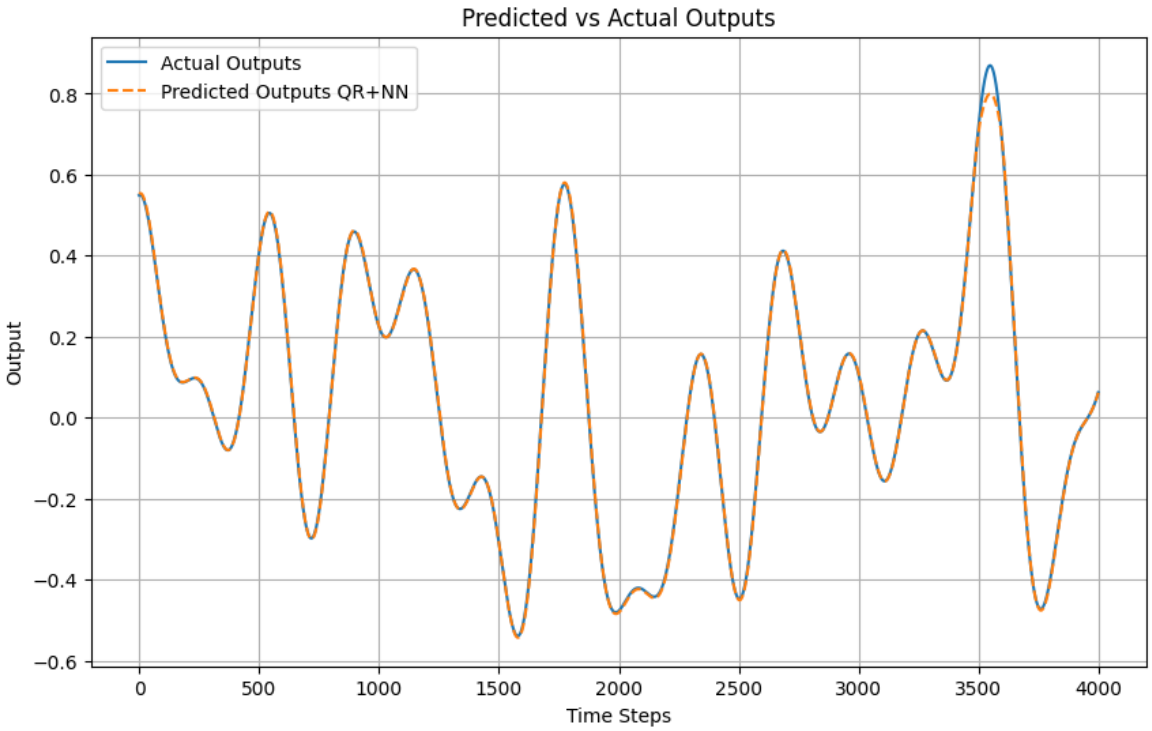

Figure 9: Plot Predicted vs Actual Outputs

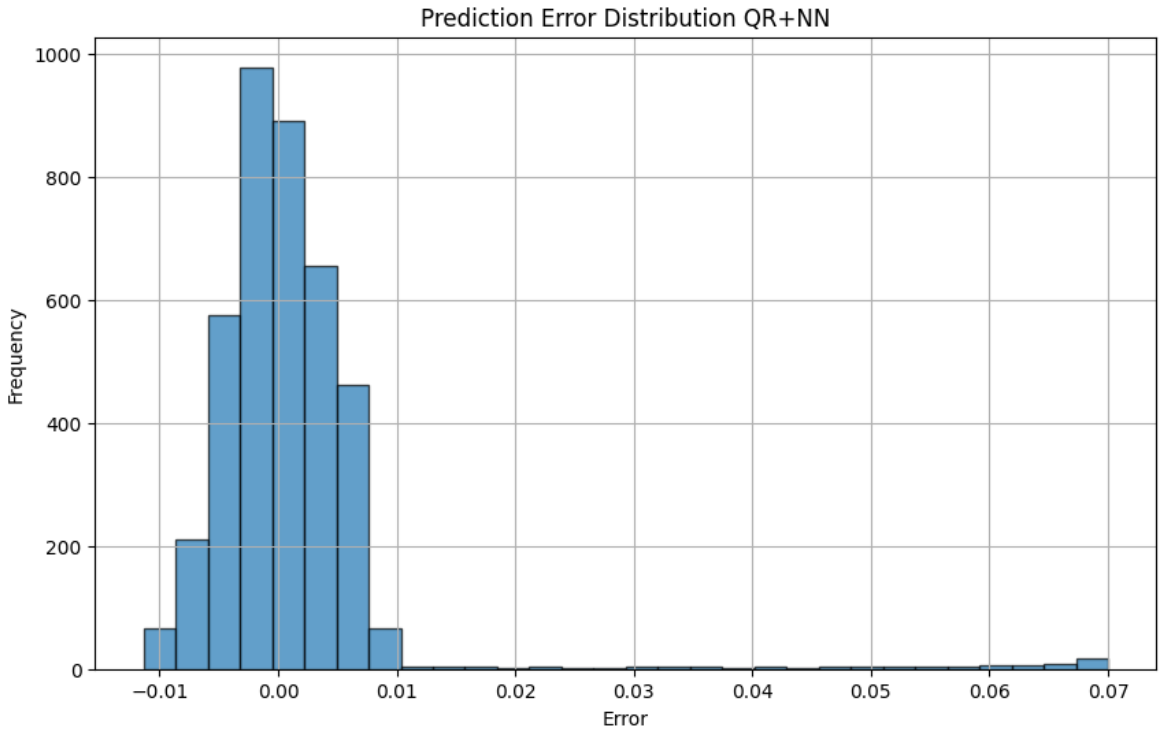

Figure 10: Plot Error Distribution

7. Other Neural Network architecture candidates to explore

Along the project we plan also to include simple NN on the following architectures to find tasks where the QR+NN configuration can be useful.

7.1. Classification task

The neural network for this classification task is a feedforward model with three layers: a 32-neuron hidden layer and a 16-neuron hidden layer, both using ReLU activation, and a final softmax output layer for three-class classification. It is compiled using the Adam optimizer with sparse categorical cross entropy as the loss function and accuracy as the performance metric. Training is conducted over 20 epochs with a batch size of 16, using 80% of the training data and 20% for validation. The model's performance is assessed using test data, with visualizations for accuracy and loss trends over epochs and a confusion matrix for predictions. The code in the appendix section.

7.2. Clustering

It applies K-Means clustering on scaled quantum reservoir outputs generated as synthetic data. StandardScaler is used to normalize the data, ensuring all features contribute equally to the clustering process. The clustering setup involves selecting the number of clusters and running the K-Means algorithm, with evaluation metrics such as silhouette scores to assess the quality of clusters. The code can be found in the appendix section.

7.3. Dimensionality Reduction

The application is to reduce the number of features in a dataset while retaining the essential information. This can be applied (Use Cases) for example in Compressing high-dimensional data for visualization or preprocessing or feature extraction from sensor data. The neural network for dimensionality reduction is an autoencoder designed to learn compressed representations of scaled quantum reservoir outputs. It consists of an input layer matching the feature dimensions, a bottleneck layer for compression, and a symmetric decoder for reconstruction. Compiled using the Adam optimizer and mean squared error as the loss function, the autoencoder is trained on scaled data to minimize reconstruction errors. This approach effectively retains essential information while reducing data dimensions, making it suitable for preprocessing high-dimensional datasets.

7.4. Anomaly Detection

This application is for Identifying data points that differ significantly from the majority of the data, this is anomaly detection. The classical example Use Case is Detecting fraud in financial transactions. Our selection is an Autoencoder. Trained to reconstruct normal data, anomalies are identified when reconstruction error exceeds a threshold. The goal is to achieve QR-enhanced networks that can improve anomaly detection in high-noise environments, such as identifying faults in industrial machinery or detecting outliers in medical data streams.

It uses an input layer matching the feature dimensions, a bottleneck layer for compressed representations, and a decoder for data reconstruction. The model is compiled with the Adam optimizer and mean squared error as the loss function, focusing on minimizing reconstruction loss during training. By comparing input and reconstructed data, the network detects anomalies as instances with high reconstruction errors, leveraging synthetic quantum reservoir outputs with injected anomalies for testing and evaluation. Part of the code can be found in the appendix section.

7.5. Simple Reinforcement Learning

The application in this case is to train a model to take actions in an environment to maximize a cumulative reward. An example Use Case can be Teaching an agent to play a game like chess or navigate a maze. As we already commented we try to choose the simpler option to fit the application. In this case the Neural Network Option is a Deep Q-Network (DQN) that combines Q-learning with deep learning to approximate action-value functions. The goal in this case is to enhance QRC with reinforcement learning by offering more nuanced state representations, enabling agents to navigate complex environments with greater efficiency. This is particularly relevant for applications in robotics and autonomous systems.

In the code section in the appendix find the development definition centered on the generator model and GAN training. We only provide the part of GAN training in this case. It utilizes TensorFlow's Sequential API to define the architecture, including dense layers with LeakyReLU activations. The network is designed to process synthetic quantum reservoir outputs as input, providing a foundational framework for tasks like data generation.

7.6. Generative Modeling

With this development we want to improve the process of generating new data that resembles the training data, using a Quantum Reservoir as the input or first layer. The common applications are of Generating synthetic images or text. We implement a Generative Adversarial Network (GAN) composed of a generator and a discriminator. The discriminator similarly uses dense layers to distinguish between real and generated data. Both models are compiled with Adam optimizers, and the GAN is trained in alternating steps: updating the generator to improve data generation and the discriminator to refine classification. Training involves minimizing loss functions for adversarial balance, leveraging synthetic quantum reservoir data for demonstration.

7.7. Signal Processing

The network architecture includes a 1D convolutional layer (1D-CNNs) for feature extraction, followed by max pooling, flattening, and dense layers to classify or analyze signals. It uses Dropout layers to prevent overfitting and is compiled with the Adam optimizer and loss functions tailored for regression or classification tasks. The model processes sinusoidal and noisy signal data to learn meaningful patterns, with training designed to optimize performance for signal-based applications. Common applications include processing EEG or ECG signals for medical diagnosis.

7.8. Sequence Modeling

We already introduced a simple LSTM but we will work deeper in future implementations of sequence modeling. Sequence modeling focuses on analyzing and predicting data that has a temporal or sequential order. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are popular architectures in this domain due to their ability to capture dependencies across time steps. These models excel in handling tasks like language modeling, speech recognition, and time-series forecasting, where the input data is sequential, and the model needs to understand how previous elements in the sequence influence future ones. RNNs and LSTMs use their internal states to process sequences and retain information over time, which is key to performing tasks such as predicting the next word in a sentence or forecasting stock prices.

7.9. Image Segmentation

Image segmentation refers to the process of dividing an image into meaningful components, typically to identify specific objects or regions. One common neural network used for this task is U-Net, which is particularly effective for biomedical image segmentation. U-Net is a convolutional neural network

(CNN) architecture that employs a symmetric encoder-decoder structure, where the encoder captures the context of an image, and the decoder refines the segmentation details. This makes it highly suitable for tasks like identifying tumors in medical images, where precise delineation of regions is crucial. U-Net's ability to work with small datasets and produce pixel-level classifications is a key strength.

7.10. Object Detection

Object detection involves locating and classifying objects within an image, a critical task in applications such as autonomous driving. Neural networks like YOLO (You Only Look Once) and Faster R-CNN are commonly used for this purpose. These architectures not only classify objects but also provide bounding boxes around detected objects in real-time, making them highly suitable for dynamic environments like self-driving cars, where identifying pedestrians, vehicles, and obstacles is essential. YOLO is known for its speed and efficiency, performing detection in a single pass, while Faster R-CNN provides high accuracy by using region proposal networks (RPNs) to identify potential object locations before classification.

7.11. Recommendation Systems

Recommendation systems predict the preferences of users based on their historical interactions and preferences. Neural networks are increasingly being used to enhance recommendation engines, with methods like collaborative filtering and deep learning-based models like DLRM (Deep Learning Recommendation Model). Collaborative filtering predicts a user's ratings for items based on the preferences of similar users, while DLRM uses embeddings for both users and items to model complex interactions. These systems are widely applied in platforms like Netflix or Amazon, where predicting the next movie to watch or product to buy requires understanding user behavior and preferences across vast amounts of data.

7.12. Multi-task Learning

Multi-task learning is an approach where a single model is trained to perform multiple tasks at once, improving generalization by leveraging shared representations. For example, a model can be simultaneously trained to perform sentiment analysis and topic classification on text data, benefiting from common features across these tasks. This is achieved by using a shared backbone network that processes input data, with task-specific heads that focus on different aspects of the problem. Multi-task learning is valuable in scenarios where tasks are related, as it enables the model to learn more robust features and improve performance across all tasks simultaneously, often with less data than training separate models.

7.13. Self-Supervised Learning

In this case the model learns to understand data without the need for explicit labels. This is done by designing pretext tasks where the model can generate its own labels from the data. Contrastive learning methods like SimCLR and BYOL are prominent in this field, where the goal is to learn useful representations of data by comparing similar and dissimilar samples. These models are pre-trained on unlabeled data and can later be fine-tuned for specific downstream tasks such as classification or detection. Self-supervised learning is increasingly used in computer vision and natural language processing, where labeled data is often scarce and expensive to obtain.

8. Potential Use Cases and Future Directions

8.1. Scalability and Optimization

Future iterations will explore scaling the QR layer by increasing the number of qubits and optimizing its physical parameters. Transitioning to quantum inputs and outputs will further enhance the system's capabilities, opening avenues for end-to-end QRC solutions.

8.2. Expanding Use Cases and long term vision

The current implementation focuses on regression and time series prediction, but the framework is already extended to tasks such as classification, clustering, and feature extraction. For instance, integrating the QR with convolutional layers could enable image recognition applications.

Building on D1.1's exploration of transmon-based reservoirs, the potential use cases for the QR-enhanced neural network include applications in quantum communication, sensor networks, and advanced signal processing. By leveraging the reservoir's ability to handle high-dimensional and nonlinear data, the network can be extended to tasks such as image recognition and reinforcement learning.

Future directions outlined in D1.1, such as incorporating memory effects into the reservoir and exploring higher-dimensional configurations, align with the scalability goals in D4.1. These enhancements will enable the development of end-to-end quantum-classical hybrid systems, paving the way for practical deployment in industrial applications.

The integration of Quantum Reservoirs (QR) into neural networks opens new possibilities for addressing computational challenges in domains requiring high-dimensional data transformation. By leveraging the QR's ability to encode nonlinear features without additional training complexity, this approach is particularly promising for areas like predictive maintenance, anomaly detection in sensor networks, and large-scale time-series forecasting. Future iterations will explore scaling the QR to handle more complex and diverse datasets.

The deliverable lays the foundation for integrating QRC systems into industrial applications, ranging from quantum communication to sensor networks. Future developments will focus on achieving practical deployment and scalability.

9. Conclusion and Next developments

Deliverable D4.1 successfully demonstrates the software implementation of a neural network leveraging a QR as its initial layer. The integration enhances computational efficiency and predictive accuracy, validating QRC's potential for advanced machine learning tasks. Benchmarking results highlight the QR's advantages over traditional approaches, paving the way for future developments in quantum-classical hybrid systems.

The outcomes of this deliverable establish a strong basis for subsequent milestones in WP4, contributing to the overall objectives of the QRC-4-ESP project. Future work will build on these achievements to expand the neural network's capabilities and explore its applications in diverse domains.

Future implementations will include automated debugging tools for QR configurations and performance tuning to ensure the model operates efficiently under diverse conditions. These enhancements will also make the framework more user-friendly for non-specialists in quantum computing.

Integration with our **qrc_qutip** library as the next development will further enhance the system's versatility and enable seamless integration with quantum computing frameworks. This integration will allow for more efficient simulation and analysis of quantum circuits within the neural network architecture. By leveraging the **qrc_qutip** library, the project will take advantage of state-of-the-art quantum algorithms and quantum resources, enhancing the overall performance of the model. Additionally, this step will streamline the process of adapting the framework for future applications, particularly in fields such as optimization, cryptography, and quantum-enhanced machine learning.

10. Appendix: Python code (neural network definitions)

10.1. Initial NN (Linear Regression)

```
# Define the training design matrix and target vector
istart = 2
iend = len(us_train) - 1
X_train = ws_train[istart:iend, :]                # Reservoir states as features
Y_train = us_train[istart+1:iend+1]               # Shifted target values

# Neural network setup
model = Sequential([
    Input(shape=(X_train.shape[1],), name="Quantum_Reservoir_Output"),
    Dense(64, activation="relu", name="Hidden_Layer_1"),
    Dense(32, activation="relu", name="Hidden_Layer_2"),
    Dense(1, activation="linear", name="Output_Layer")  # Single output for regression
])

# Compile the neural network
model.compile(optimizer="adam", loss="mse", metrics=["mae"])

# Train-validation split
X_train_split, X_val_split, Y_train_split, Y_val_split = train_test_split(X_train, Y_train, test_size=0.2, random_state=42)

# Train the neural network
history = model.fit(
    X_train_split, Y_train_split,
    validation_data=(X_val_split, Y_val_split),
    epochs=50,  # Number of epochs
    batch_size=32,  # Batch size
    verbose=1  # Show progress
)
```

10.2. Time-Series Forecasting (LSTM)

```
# --- Prepare Data for Time-Series Forecasting ---
def create_time_series_data(data, time_steps=10):
    """
    Create sequences of data for supervised learning.
    """
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps])
        y.append(data[i+time_steps])
    return np.array(X), np.array(y)

# Define time steps (sequence length) for LSTM
time_steps = 10
X, y = create_time_series_data(quantum_reservoir_data, time_steps)
print(f"Input Shape (X): {X.shape}, Target Shape (y): {y.shape}")

# Split data into training and testing sets
split_index = int(len(X) * 0.8)
X_train, y_train = X[:split_index], y[:split_index]
X_test, y_test = X[split_index:], y[split_index:]

# --- Build the LSTM Model ---
def build_lstm_model(input_shape):
    """
    Build an LSTM model for time-series forecasting.
    """
    model = Sequential(name="Time-Series_LSTM")
    model.add(LSTM(64, activation='relu', input_shape=input_shape))
    model.add(Dense(1))  # Single output for regression
    return model

# Initialize the model
input_shape = (X_train.shape[1], X_train.shape[2])  # (time_steps, features)
lstm_model = build_lstm_model(input_shape)

# Compile the model
lstm_model.compile(optimizer='adam', loss='mse', metrics=['mae'])
lstm_model.summary()

# --- Train the Model ---
history = lstm_model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, batch_size=32, verbose=1)
```

10.3. NN for classification

```
# Generate classification labels
targets = create_classification_targets(num_samples=qr_output.shape[0])

# --- Splitting and Scaling Data ---
# Split into training and testing datasets
X_train, X_test, y_train, y_test = train_test_split(qr_output, targets, test_size=0.2, random_state=42)

# Scale the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# --- Defining a Simple Feedforward Neural Network ---
model = Sequential([
    Dense(32, activation='relu', input_dim=X_train.shape[1]),
    Dense(16, activation='relu'),
    Dense(3, activation='softmax')  # Output layer for 3-class classification
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# --- Training the Model ---
history = model.fit(X_train, y_train, validation_split=0.2, epochs=20, batch_size=16, verbose=1)
```

10.4. NN for Clustering

```
# --- Defining and Running K-Means Clustering ---
def run_kmeans(data, n_clusters):
    """
    Apply KMeans clustering and return the cluster labels and the fitted model.
    """
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    cluster_labels = kmeans.fit_predict(data)
    return cluster_labels, kmeans

# Number of clusters
n_clusters = 3
cluster_labels, kmeans_model = run_kmeans(qr_output_scaled, n_clusters)

# --- Evaluating Clustering Performance ---
def evaluate_clustering(data, labels):
    """
    Evaluate the clustering using silhouette score.
    """
    score = silhouette_score(data, labels)
    print(f"Silhouette Score: {score:.4f}")
    return score

# Evaluate clustering performance
silhouette = evaluate_clustering(qr_output_scaled, cluster_labels)

# --- Visualizing Clustering Results ---
# Reduce dimensionality for visualization using PCA
from sklearn.decomposition import PCA
```

10.5. Dimensionality Reduction

```
# Generate synthetic quantum reservoir outputs
qr_output = generate_qr_output()
print(f"Shape of Quantum Reservoir Outputs: {qr_output.shape}")

# --- Scaling the Data ---
# Scale the quantum reservoir outputs
scaler = StandardScaler()
qr_output_scaled = scaler.fit_transform(qr_output)
print("Data has been scaled.")

# --- Defining and Training the Autoencoder ---
input_dim = qr_output_scaled.shape[1]
encoding_dim = 2  # Dimensionality of the compressed representation

# Define the autoencoder model
autoencoder = Sequential([
    Dense(encoding_dim, activation='relu', input_dim=input_dim),
    Dense(input_dim, activation='sigmoid')
])

# Compile the model
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
history = autoencoder.fit(qr_output_scaled, qr_output_scaled,
                          epochs=50, batch_size=32, shuffle=True, validation_split=0.2, verbose=1)
```

10.6. Anomaly Detection

```
# Generate synthetic quantum reservoir outputs with anomalies
qr_output, labels = generate_qr_output()
print(f"Shape of Quantum Reservoir Outputs: {qr_output.shape}")
print(f"Number of Anomalies: {np.sum(labels)}")

# --- Splitting and Scaling the Data ---
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(qr_output, labels, test_size=0.2, random_state=42)

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
print("Data has been split and scaled.")

# --- Defining and Training the Autoencoder ---
input_dim = X_train_scaled.shape[1]

# Define the autoencoder model
autoencoder = Sequential([
    Dense(32, activation='relu', input_dim=input_dim),
    Dense(16, activation='relu'),
    Dense(8, activation='relu'),
    Dense(16, activation='relu'),
    Dense(32, activation='relu'),
    Dense(input_dim, activation='sigmoid')
])

# Compile the model
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder on normal data only (filter out anomalies)
normal_data = X_train_scaled[y_train == 0]
history = autoencoder.fit(normal_data, normal_data,
                          epochs=50, batch_size=32, shuffle=True, validation_split=0.2, verbose=1)
```

10.7. Simple Reinforcement Learning

```python
# --- Training the GAN ---
def train_gan(generator, discriminator, quantum_reservoir_data, epochs=5000, batch_size=32):
    half_batch = batch_size // 2
    d_losses, g_losses = [], []

    for epoch in range(epochs):
        # --- Train the Discriminator ---
        # Select a random half-batch of real data from the quantum reservoir
        idx = np.random.randint(0, quantum_reservoir_data.shape[0], half_batch)
        real_data = quantum_reservoir_data[idx]

        # Generate a half-batch of fake data using the generator
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_data = generator(noise, training=False)

        # Real and fake labels
        real_labels = np.ones((half_batch, 1), dtype=np.float32)
        fake_labels = np.zeros((half_batch, 1), dtype=np.float32)

        # Train discriminator
        with tf.GradientTape() as tape:
            real_loss = tf.keras.losses.binary_crossentropy(real_labels, discriminator(real_data, training=True))
            fake_loss = tf.keras.losses.binary_crossentropy(fake_labels, discriminator(fake_data, training=True))
            d_loss = tf.reduce_mean(real_loss) + tf.reduce_mean(fake_loss)
        grads = tape.gradient(d_loss, discriminator.trainable_variables)
        discriminator_optimizer.apply_gradients(zip(grads, discriminator.trainable_variables))
        d_losses.append(d_loss.numpy())

        # --- Train the Generator ---
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        misleading_labels = np.ones((batch_size, 1), dtype=np.float32)

        with tf.GradientTape() as tape:
            fake_data = generator(noise, training=True)
            g_loss = tf.reduce_mean(tf.keras.losses.binary_crossentropy(misleading_labels, discriminator(fake_data, training=False)))
        grads = tape.gradient(g_loss, generator.trainable_variables)
        generator_optimizer.apply_gradients(zip(grads, generator.trainable_variables))
        g_losses.append(g_loss.numpy())

        # Print progress
        if epoch % 1000 == 0:
            print(f"Epoch {epoch}/{epochs} | D Loss: {d_loss:.4f} | G Loss: {g_loss:.4f}")

    return d_losses, g_losses
```

10.8. Generative GAN (Generative Modeling)

```python
# --- Training the GAN ---
def train_gan(generator, discriminator, quantum_reservoir_data, epochs=5000, batch_size=32):
    """
    Train the GAN using a manual training loop.
    """
    half_batch = batch_size // 2
    d_losses, g_losses = [], []

    for epoch in range(epochs):
        # --- Train the Discriminator ---
        # Select a random half-batch of real data
        idx = np.random.randint(0, quantum_reservoir_data.shape[0], half_batch)
        real_data = quantum_reservoir_data[idx]

        # Generate a half-batch of fake data
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_data = generator(noise, training=False)

        # Real and fake labels
        real_labels = np.ones((half_batch, 1), dtype=np.float32)
        fake_labels = np.zeros((half_batch, 1), dtype=np.float32)

        with tf.GradientTape() as tape:
            real_loss = tf.keras.losses.binary_crossentropy(real_labels, discriminator(real_data, training=True))
            fake_loss = tf.keras.losses.binary_crossentropy(fake_labels, discriminator(fake_data, training=True))
            d_loss = tf.reduce_mean(real_loss) + tf.reduce_mean(fake_loss)
        grads = tape.gradient(d_loss, discriminator.trainable_variables)
        discriminator_optimizer.apply_gradients(zip(grads, discriminator.trainable_variables))
        d_losses.append(d_loss.numpy())

        # --- Train the Generator ---
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        misleading_labels = np.ones((batch_size, 1), dtype=np.float32)

        with tf.GradientTape() as tape:
            fake_data = generator(noise, training=True)
            g_loss = tf.reduce_mean(tf.keras.losses.binary_crossentropy(misleading_labels, discriminator(fake_data, training=False)))
        grads = tape.gradient(g_loss, generator.trainable_variables)
        generator_optimizer.apply_gradients(zip(grads, generator.trainable_variables))
        g_losses.append(g_loss.numpy())

        # Print progress
        if epoch % 1000 == 0:
            print(f"Epoch {epoch}/{epochs} | D Loss: {d_loss:.4f} | G Loss: {g_loss:.4f}")

    return d_losses, g_losses
```

10.9. Signal processing

```
# Define time steps for the 1D-CNN
time_steps = 50
X, y = create_signal_data(quantum_reservoir_signal, time_steps)
print(f"Input Shape (X): {X.shape}, Target Shape (y): {y.shape}")

# Split data into training and testing sets
split_index = int(len(X) * 0.8)
X_train, y_train = X[:split_index], y[:split_index]
X_test, y_test = X[split_index:], y[split_index:]

# --- Build the 1D-CNN Model ---
def build_1d_cnn(input_shape):
    """
    Build a 1D CNN for signal processing and classification.
    """
    model = Sequential(name="Signal_1D_CNN")
    model.add(Conv1D(32, kernel_size=3, activation='relu', input_shape=input_shape))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Conv1D(64, kernel_size=3, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification
    return model

# Initialize the model
input_shape = (X_train.shape[1], X_train.shape[2])  # (time_steps, features)
cnn_model = build_1d_cnn(input_shape)

# Compile the model
cnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
cnn_model.summary()

# --- Train the Model ---
history = cnn_model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=20, batch_size=32, verbose=1)
```

11. References

1. Google Brain, *TensorFlow: An Open-Source Machine Learning Framework.* Available at: https://www.tensorflow.org/
2. *J. R. Johansson, P. D. Nation, F. Nori, "QuTiP: An Open-Source Python Framework for the Dynamics of Open Quantum Systems." Available at: http://qutip.org/*
3. *Facebook AI Research, "PyTorch: An Open-Source Machine Learning Library for Python." Available at: https://pytorch.org/*
4. *SylphAI Inc., "LLM Engineer's Handbook: Master the Art of Engineering Large Language Models from Concept to Production." Available at: https://github.com/SylphAI-Inc/LLM-engineer-handbook/*
5. *Project Jupyter, "Jupyter Notebooks: An Open-Source Web Application for Interactive Computing." Available at: https://jupyter.org/*