**Quantum Machine Learning (QML) Documentation**

**Introduction to Quantum Machine Learning (QML) & Setup**

**Onboarding & Fundamentals**

**Introduction to Quantum Computing & Machine Learning**

Quantum computing leverages quantum mechanics principles to process information differently than classical computers. Quantum Machine Learning (QML) combines quantum computing with classical machine learning, aiming to enhance efficiency and solve complex problems that classical models struggle with.

**Setting Up PennyLane, Python, and Git**

PennyLane is a powerful open-source framework for quantum machine learning. To get started, install the necessary libraries:

pip install pennylane

pip install pennylane-qiskit

Additionally, configure Git for version control:

git config --global user.name "Your Name"

git config --global user.email "youremail@example.com"

**Classical vs. Quantum Computing**

| **Feature** | **Classical Computing** | **Quantum Computing** |
| --- | --- | --- |
| Data Representation | Bits (0 or 1) | Qubits (0, 1, or superposition) |
| Processing Power | Sequential/Parallel | Exponential via entanglement |
| Example Usage | Traditional ML, Web Apps | QML, Cryptography, Optimization |

**Basics of Quantum Circuits & QML and Mathematics**

**Basics of Linear Algebra, Probability Theory, and Complex Numbers**

* **Linear Algebra**: Essential for quantum computing; quantum states and operations are represented as vectors and matrices.
* **Probability Theory**: Quantum measurements are probabilistic, requiring an understanding of probability distributions.
* **Complex Numbers**: Quantum states use complex numbers, where coefficients (α and β) determine probabilities of measurement outcomes.

**Introduction to Quantum Encoding**

Quantum encoding maps classical data onto quantum states. The following example encodes a classical value into a quantum state using the RY gate:

import pennylane as qml

import numpy as np

def encode\_classical\_data(x):

dev = qml.device("default.qubit", wires=1)

@qml.qnode(dev)

def circuit():

qml.RY(x, wires=0)

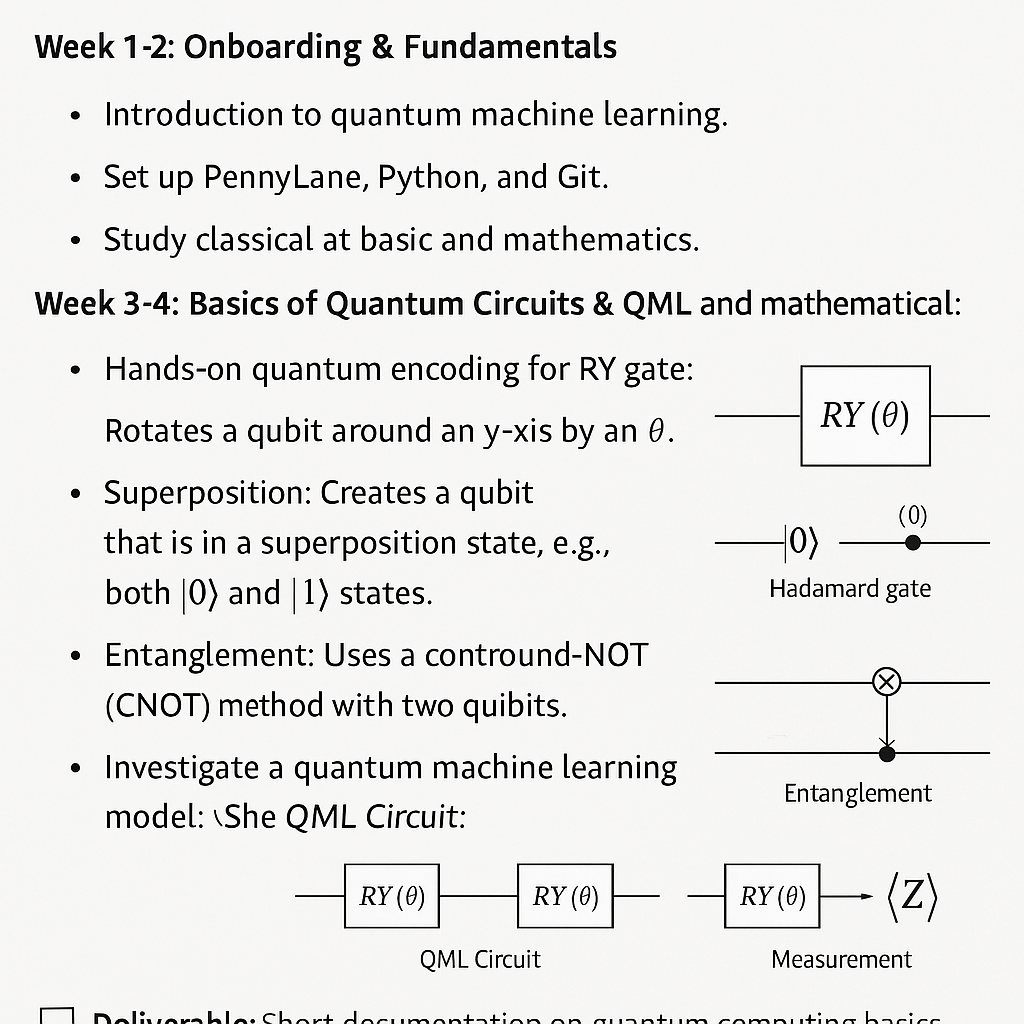
return qml.state()

return circuit()

print(encode\_classical\_data(np.pi/4))

**Output:** A quantum state representation of the input value.

**Visual Representation:** *(Diagram of a single qubit rotation using the RY gate)*



**Understanding the Visual Elements in the Image**

The following sections describe the key elements in the uploaded image:

1. **Quantum Encoding with RY Gate**
   * **Explanation**: The RY gate rotates a qubit around the Y-axis by a given angle, crucial for encoding classical data into quantum states.
   * **Circuit Representation**: A single-qubit rotation gate applied to a qubit to encode classical information.
   * **Expected Visual**: A qubit line with an RY(θ) gate.
2. **Quantum Superposition and Entanglement with Hadamard and CNOT Gates**
   * **Hadamard Gate (H)**
     + **Explanation**: Creates a superposition state (i.e., a qubit being in both |0⟩ and |1⟩ states).
     + **Expected Visual**: A circuit where a Hadamard gate is applied to a qubit, placing it in a superposition state.
   * **CNOT Gate**
     + **Explanation**: Entangles two qubits, meaning their states become dependent on each other.
     + **Expected Visual**: A controlled-NOT operation where the control qubit determines the state of the target qubit.

**Hands-on Python & PennyLane Setup for Quantum Computing**

The following example demonstrates the application of quantum gates:

@qml.qnode(dev)

def quantum\_circuit():

qml.Hadamard(wires=0)

qml.CNOT(wires=[0, 1])

return qml.state()

print(quantum\_circuit())

**Key Concepts:**

* **Hadamard Gate (H):** Creates a superposition state.
* **CNOT Gate:** Entangles two qubits, enabling quantum parallelism.

1. **Quantum Machine Learning Model (QML Circuit)**
   * **Explanation**: A multi-qubit circuit where rotations (RY) are applied, followed by a CNOT gate to establish entanglement, and then an expectation value is measured.
   * **Expected Visual**: A quantum circuit where two qubits undergo RY rotations followed by a CNOT gate and a measurement operation.

**Exploring PennyLane’s QML Framework**

PennyLane enables the implementation and simulation of quantum circuits. Below is a simple QML model:

import pennylane as qml

import tensorflow as tf

dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev, interface='tf')

def quantum\_model(inputs):

qml.RY(inputs[0], wires=0)

qml.RY(inputs[1], wires=1)

qml.CNOT(wires=[0, 1])

return qml.expval(qml.PauliZ(0))

x\_train = tf.constant([[0.1, 0.5], [0.6, 0.9]], dtype=tf.float32)

output = quantum\_model(x\_train[0])

print(output)

**Output:** Expectation value of the quantum system after transformation.

**Deliverable**

* **Refined and structured documentation** covering quantum computing fundamentals and QML basics.
* **Diagrams and images** illustrating quantum circuits and transformations (to be added as needed).