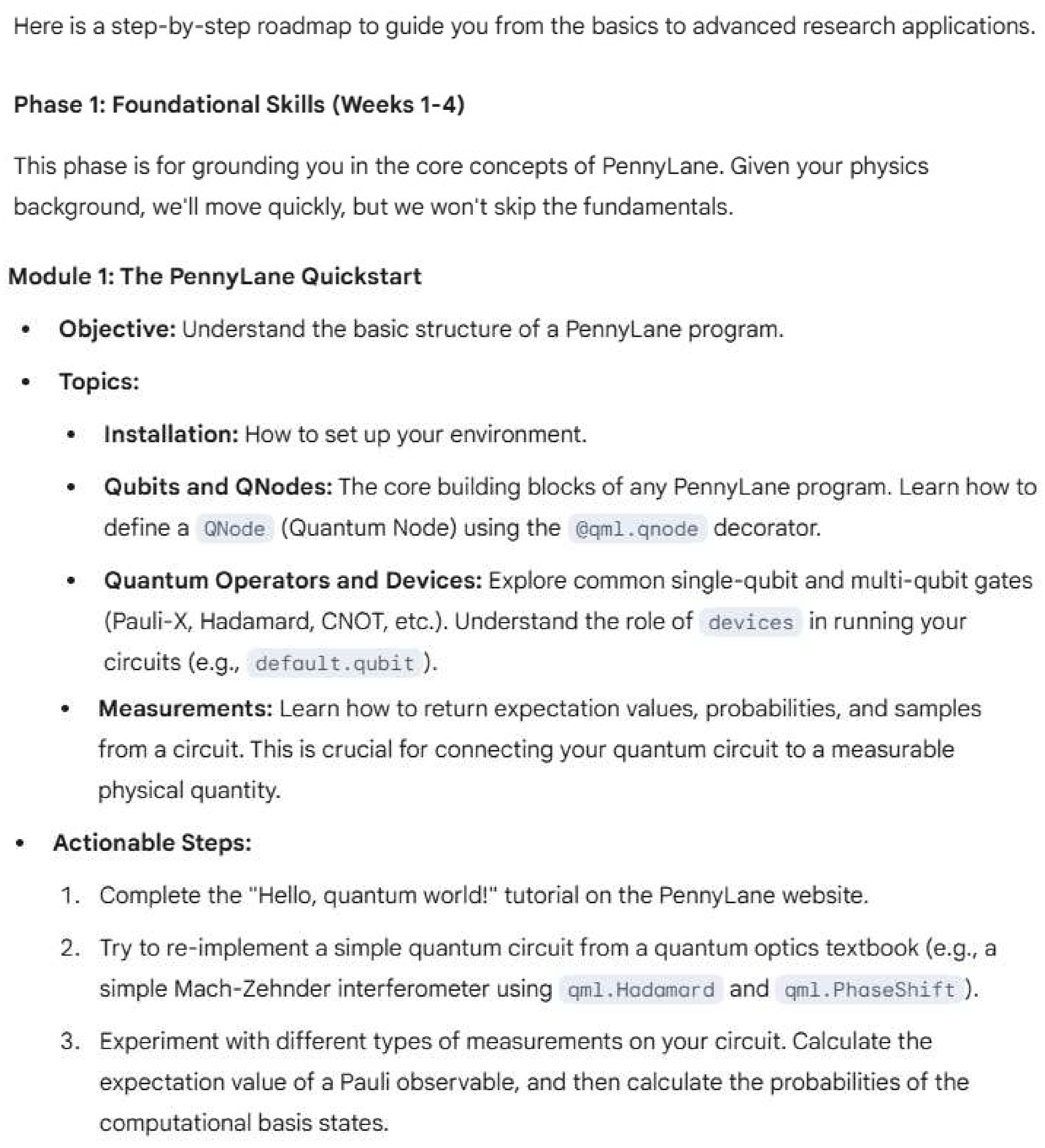
Road map

03 August 2025 10:15



Here is a step-by-step roadmap to guide you from the basics to advanced research applications.

Phase 1: Foundational Skills (Weeks 1-4)

This phase is for grounding you in the core concepts of PennyLane. Given your physics background, we'll move quickly, but we won't skip the fundamentals.

Module 1: The PennyLane Quickstart

- **Objective:** Understand the basic structure of a PennyLane program.
- **Topics:**
  - **Installation:** How to set up your environment.
  - **Qubits and QNodes:** The core building blocks of any PennyLane program. Learn how to define a `QNode` (Quantum Node) using the `@qml.qnode` decorator.
  - **Quantum Operators and Devices:** Explore common single-qubit and multi-qubit gates (Pauli-X, Hadamard, CNOT, etc.). Understand the role of `devices` in running your circuits (e.g., `default.qubit`).
  - **Measurements:** Learn how to return expectation values, probabilities, and samples from a circuit. This is crucial for connecting your quantum circuit to a measurable physical quantity.
- **Actionable Steps:**
  1. Complete the "Hello, quantum world!" tutorial on the PennyLane website.
  2. Try to re-implement a simple quantum circuit from a quantum optics textbook (e.g., a simple Mach-Zehnder interferometer using `qml.Hadamard` and `qml.PhaseShift`).
  3. Experiment with different types of measurements on your circuit. Calculate the expectation value of a Pauli observable, and then calculate the probabilities of the computational basis states.

Module 2: Differentiable Programming

- **Objective:** Master the concept of "quantum differentiable programming," the key feature of PennyLane.
- **Topics:**
  - **Variational Circuits:** Understand how to build circuits with trainable parameters. This is the cornerstone of variational quantum algorithms.
  - **Automatic Differentiation:** PennyLane's magic! Learn how to compute gradients of your circuit's output with respect to its parameters.
  - **Optimization:** Use classical optimizers from popular libraries like NumPy, JAX, or PyTorch to find the optimal parameters for your circuit.
- **Actionable Steps:**
  1. Work through the "Qubit rotation" demo on the PennyLane website. This is a classic example of optimizing a circuit to match a target output.
  2. Try to implement a simple "function fitting" problem. Define a target function (e.g., a sine wave) and use a simple parameterized quantum circuit to approximate it by minimizing a cost function.

Phase 2: Quantum Optics Focus (Weeks 5-8)

Now we'll move from generic quantum computing to your specific field of expertise. PennyLane has excellent support for Continuous Variable (CV) quantum computing, which is highly relevant to quantum optics.

Module 3: Continuous-Variable Quantum Computing

- **Objective:** Learn to model quantum optical systems using PennyLane's CV devices and gates.
- **Topics:**
  - **CV Devices:** Get acquainted with `default.gaussian` and `strawberryfields.fock`, which are simulators for bosonic systems.
  - **CV Gates:** Explore gates like `qml.Displacement`, `qml.Squeezing`, and `qml.Beamsplitter`. Relate these to the physical operations you know from quantum optics.
  - **Quadrature and Photon Number Measurements:** Understand how to measure observables like `qml.QuadX`, `qml.QuadP`, and `qml.NumberOperator`.

- **Actionable Steps:**
    1. Read the "Gaussian transformation" and "Function fitting with a photonic quantum neural network" demos.
    2. Implement a well-known quantum optics experiment, such as the generation of squeezed states or a two-mode squeezed vacuum, and simulate its properties.
    3. Use PennyLane's optimization capabilities to find the parameters of a circuit that maximizes the squeezing or entanglement in a state.

Module 4: Quantum Machine Learning for Quantum Optics

- **Objective:** Begin applying machine learning techniques to quantum optical problems.
- **Topics:**
    - **Quantum-Classical Hybrid Models:** Learn how to build models that combine classical neural networks with quantum circuits. This is a powerful paradigm for near-term quantum devices.
    - **Feature Encoding:** Understand how to map classical data into a quantum state.
    - **Solving Problems:** Think about how you can use QML to solve real-time problems in quantum optics, such as:
        - **State classification:** Can a QML model distinguish between different quantum states (e.g., coherent vs. squeezed states)?
        - **Quantum state tomography:** Can you use a variational circuit to reconstruct an unknown quantum state?
- **Actionable Steps:**
    1. Explore the PennyLane demos on quantum machine learning, such as "Quantum transfer learning."
    2. Design and implement a simple classification model to distinguish between a coherent state and a squeezed state based on quadrature measurements.
    3. Consider a specific problem in quantum optics that you find interesting and sketch out a QML approach to solving it.

Phase 3: Advanced Research and Job Preparation (Weeks 9-12+)

This is where we transition from guided learning to independent research and career-focused application.

Module 5: Advanced PennyLane and Research Tools

- **Objective:** Gain proficiency in advanced features and best practices for research.
- **Topics:**
  - **Hardware and Cloud Access:** Learn how to run your PennyLane programs on real quantum hardware and cloud platforms like Amazon Braket, IBM Quantum, or Xanadu's hardware.
  - **Customizing PennyLane:** Understand how to create custom gates, observables, and even devices to simulate new physics or hardware.
  - **Integrating with Other Libraries:** Use PennyLane with a powerful numerical backend like JAX for just-in-time compilation and high-performance computing, which can be crucial for your research.
- **Actionable Steps:**
  1. Learn about the `default.qubit` and `default.gaussian` devices' limitations and how to use plugins to access real hardware.
  2. Write a blog post or a small tutorial on a custom gate or observable you've created for a specific quantum optics problem.
  3. Rerun one of your previous circuits using the JAX backend for a performance boost.

Module 6: Research and Professional Development

- **Objective:** Position yourself as a strong candidate for a research scientist position.
- **Topics:**
  - **Identifying Research Problems:** Focus on a specific challenge in quantum optics that could benefit from a PennyLane-based approach. This could be in areas like quantum sensing, quantum communication, or novel quantum state generation.
  - **Building a Portfolio:** Develop a portfolio of projects, which could include:
    - A documented and well-commented GitHub repository of your code.
    - A detailed Jupyter notebook demonstrating a complete research idea.
    - A potential research paper draft.
  - **Networking and Job Search:**
    - Look for companies and research labs (e.g., Xanadu, PsiQuantum, IBM, Google) that are doing work at the intersection of quantum optics and quantum computing.
    - Connect with researchers on platforms like LinkedIn and participate in the PennyLane community forum.
- **Actionable Steps:**
  1. Choose a research problem. For example, "Using a variational quantum algorithm to optimize the design of a photonic circuit for a specific quantum state generation."
  2. Develop a comprehensive project on this topic, from a literature review to a complete PennyLane implementation.
  3. Start attending virtual talks and workshops from the PennyLane team and the broader quantum community.

Phase 1: module 1:Foundational Skills

03 August 2025 10:26

Phase 1, Module 1: The PennyLane Quickstart

Our first module focuses on getting your environment set up and understanding the fundamental building blocks: Qubits, QNodes, Quantum Operators, Devices, and Measurements.

Step 0: Setting up your PennyLane Environment

Before we write any code, you need to install PennyLane. You have two main options:

1. **Easiest: Google Colab**
   This is highly recommended for quick starts and experimenting without local setup complexities.
   - Open a new notebook in Google Colab.
   - In the first cell, run the following command (including the exclamation mark):

   ! pip install pennylane
   ```

   This will install PennyLane directly in your Colab environment.
2. **For Heavy Workloads: Local Installation with Conda (Recommended for long-term research)**
   If you plan on running larger simulations or want a dedicated local environment, using Conda is a robust approach.
   - **Install Miniconda:** Follow the instructions to install Miniconda on your system.
   - **Open your terminal (mac) or command line (Windows).**
   - **Create a new Conda environment:** Choose a descriptive name for your environment (e.g., `pennylane_env`).

     Bash
     ```
     conda create --name pennylane_env python=3.10
     ```

- **Activate the environment:** You'll need to do this every time you want to work on your PennyLane projects.

  Bash
  ```
  conda activate pennylane_env
  ```
  ```

- **Activate the environment:** You'll need to do this every time you want to work on your PennyLane projects.

  Bash

  ```
  conda activate pennylane_env
  ```

- **Install PennyLane:**

  Bash

  ```
  python -m pip install pennylane
  ```

- **Install other useful packages:** Jupyter (for notebooks) and Matplotlib (for plotting).

  Bash

  ```
  python -m pip install pennylane jupyter matplotlib
  ```

  - **Important Note:** Always use `python -m pip install` instead of `conda install` when adding new Python packages to this environment.
  - After installation, you can type `jupyter notebook` in your activated terminal to start a Jupyter server and create PennyLane programs.

Once your environment is ready, let's move to the core tasks.

Step 1: Complete the "Hello, quantum world!" tutorial

While there isn't a single universal "Hello, quantum world!" for quantum computing, a common introductory task in Quantum Machine Learning (QML) is the "Qubit Rotation" example. This involves optimizing a qubit's state from (|0\rangle) to (|1\rangle) using two rotation gates. This task is an excellent starting point as it introduces you to defining quantum circuits, using parameters, and performing measurements.

Here's how you can implement this in PennyLane:

Objective: Rotate a single qubit from the initial state (|0\rangle) to (|1\rangle) by optimizing two rotation angles.

1. **Import PennyLane and NumPy:**

Python

```python
import pennylane as qml
from pennylane import numpy as np # PennyLane's NumPy is differentiable
```

2. **Define a Device:**
A "device" in PennyLane is where your quantum circuit will run. For small-scale simulations and fast prototyping, `default.qubit` is ideal. It's included with PennyLane and is a CPU-based simulator.

Python

```python
dev = qml.device("default.qubit", wires=1) # We need 1 qubit (wire) for this ta
```

```
for this task
```

3. **Define your Quantum Function (Circuit):**
This is a standard Python function that contains PennyLane quantum operations. You'll use the `@qml.qnode` decorator to turn this function into a QNode, which is the central building block for quantum computations in PennyLane.
The circuit will apply an `RX` rotation (around the x-axis) and an `RY` rotation (around the y-axis) to the qubit. Finally, we measure the expectation value of the Pauli-Z operator, (\langle \sigma_z \rangle). Our goal is to make this expectation value approach -1, which corresponds to the (|1\rangle) state.

Python

```python
@qml.qnode(dev)
def circuit(params):
    qml.RX(params, wires=0) # Apply RX gate with parameter params on wire 0
    qml.RY(params, wires=0) # Apply RY gate with parameter params on wire 0
    return qml.expval(qml.PauliZ(0)) # Measure the expectation value of PauliZ
```

- `qml.RX` **and** `qml.RY` **:** These are single-qubit rotation gates.
- `qml.expval(qml.PauliZ(0))` **:** This measures the expectation value of the Pauli-Z observable on wire 0.

4. **Run the Circuit:**
   You can now call your `circuit` function with some initial parameters.

Python

```python
initial_params = np.array([0.54, 0.12], requires_grad=True) # Parameters for R
print(f"Initial expectation value: {circuit(initial_params)}")
```

```
for RX and RY
```

5. **Optimization (Preview for Module 2):**
   While optimization is the focus of Module 2, it's worth seeing how this "Hello world" example concludes. The goal is to find `params` that make `circuit(params)` close to -1.

Python

```python
# Define a cost function
def cost(params):
    return (circuit(params) - (-1))**2 # We want the expectation value to be -

# Initialize an optimizer
opt = qml.GradientDescentOptimizer(stepsize=0.4)

# Optimize the parameters
params = initial_params
for i in range(100):
    params = opt.step(cost, params)
    if (i + 1) % 10 == 0:
        print(f"Step {i+1}: Cost = {cost(params):.4f}, Expectation = {circuit(

print(f"\nOptimized expectation value: {circuit(params)}")
print(f"Optimized parameters: {params}")
to be -1



:ircuit(params):.4f}")
```

This optimization process uses PennyLane's automatic differentiation to calculate gradients of the cost function with respect to the circuit parameters, then updates the parameters using a classical optimizer.

Step 2: Re-implement a simple quantum circuit from a quantum optics textbook (e.g., a simple Mach-Zehnder interferometer)

Let's apply your physics knowledge to build a Mach-Zehnder interferometer. In quantum optics, this typically involves beam splitters and phase shifters. In qubit-based quantum computing, a Hadamard gate acts as a beam splitter, and a PhaseShift gate introduces a phase.

Objective: Simulate a Mach-Zehnder interferometer and observe its output.

1. **Import PennyLane and NumPy:**

   Python

   ```python
   import pennylane as qml
   from pennylane import numpy as np
   ```

2. **Define a Device:**
   We'll use `default.qubit` again, but this time with 2 wires, as a Mach-Zehnder interferometer typically involves two paths (which can be mapped to two qubits or two modes of a single photon). For simplicity, we'll use one qubit and consider the two paths as different states of that qubit.

   Python

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

3. **Define the Mach-Zehnder Circuit:**
   A basic Mach-Zehnder interferometer can be modeled with two Hadamard gates and a phase shift in between.
   - The first `qml.Hadamard` acts as the first beam splitter, putting the qubit into a superposition.
   - `qml.PhaseShift` applies a phase to one path (or state component).
   - The second `qml.Hadamard` acts as the second beam splitter, recombining the paths.
   - We'll measure the probability of finding the qubit in the (|0\rangle) state.

Python

```
@qml.qnode(dev)
def mach_zehnder_interferometer(phi):
    qml.Hadamard(wires=0)       # First beam splitter (Hadamard)
    qml.PhaseShift(phi, wires=0) # Phase shift on one path
    qml.Hadamard(wires=0)       # Second beam splitter (Hadamard)
    return qml.probs(wires=0)  # Measure probabilities of computational basis s
```

`states`

4. **Run and Observe:**
   Let's run the circuit for different phase shifts and see how the probabilities change.

Python

```
# Test with different phase shifts
phases = np.linspace(0, 2 * np.pi, 20)
probabilities_0 =
probabilities_1 =

for phi in phases:
    probs = mach_zehnder_interferometer(phi)
    probabilities_0.append(probs)
    probabilities_1.append(probs)
    print(f"Phase = {phi:.2f} rad: P(|0>) = {probs:.4f}, P(|1>) = {probs:.4f}")
```

```
# You can also plot these results to see the interference pattern
import matplotlib.pyplot as plt
plt.plot(phases, probabilities_0, label='P(|0>)')
plt.plot(phases, probabilities_1, label='P(|1>)')
plt.xlabel("Phase Shift (phi)")
plt.ylabel("Probability")
plt.title("Mach-Zehnder Interferometer Output")
plt.legend()
plt.grid(True)
plt.show()
```

You should observe an interference pattern, where the probability of measuring (|0\rangle) (or (|1\rangle)) varies sinusoidally with the phase `phi`, just as you would expect from a physical Mach-Zehnder interferometer.

Step 3: Experiment with different types of measurements on your circuit

PennyLane offers several ways to extract information from your quantum circuits: expectation values, probabilities, and samples. Let's use the Mach-Zehnder circuit to explore these.

Objective: Understand and implement `qml.expval`, `qml.probs`, and `qml.sample`.

1. **Expectation Value of a Pauli Observable (`qml.expval`)**
   This returns the average value of an observable if you were to measure it many times. For a single qubit, `qml.PauliZ(0)` measures in the Z-basis.

Python

```python
@qml.qnode(dev)
def mz_expval(phi):
    qml.Hadamard(wires=0)
    qml.PhaseShift(phi, wires=0)
    qml.Hadamard(wires=0)
    return qml.expval(qml.PauliZ(0)) # Expectation value of PauliZ on wire 0

print(f"\n--- Expectation Value ---")
print(f"For phi = 0.0: <Z> = {mz_expval(0.0):.4f}")
print(f"For phi = {np.pi:.2f}: <Z> = {mz_expval(np.pi):.4f}")
```

- **Tensor Observables:** You can also measure the expectation value of a tensor product of observables using the `@` notation. For example, if you had two wires and wanted to measure (Z \otimes X): `qml.expval(qml.PauliZ(0) @ qml.PauliX(1))`.

2. **Probabilities of Computational Basis States (`qml.probs`)**
   This returns the probability of each computational basis state (e.g., (|0\rangle), (|1\rangle)).

```
Python
```

```python
@qml.qnode(dev)
def mz_probs(phi):
    qml.Hadamard(wires=0)
    qml.PhaseShift(phi, wires=0)
    qml.Hadamard(wires=0)
    return qml.probs(wires=0) # Probabilities of states on wire 0

print(f"\n--- Probabilities ---")
probs_at_zero = mz_probs(0.0)
print(f"For phi = 0.0: P(|0>) = {probs_at_zero:.4f}, P(|1>) = {probs_at_zero:.
probs_at_pi = mz_probs(np.pi)
print(f"For phi = {np.pi:.2f}: P(|0>) = {probs_at_pi:.4f}, P(|1>) = {probs_at_
```

```
{probs_at_zero:.4f}")

|1>) = {probs_at_pi:.4f}")
```

3. **Samples of a Single Measurement (`qml.sample`)**
   This returns raw measurement outcomes from a specified number of "shots" (repetitions of the circuit). This is particularly useful when simulating noisy hardware or when you need individual measurement results rather than averages.
   You need to configure the device to use a finite number of shots.

```
Python
```

```python
# Re-define device with shots
dev_shots = qml.device("default.qubit", wires=1, shots=100) # 100 repetitions

@qml.qnode(dev_shots)
def mz_samples(phi):
    qml.Hadamard(wires=0)
    qml.PhaseShift(phi, wires=0)
    qml.Hadamard(wires=0)
    return qml.sample(qml.PauliZ(0)) # Sample the PauliZ observable on wire 0

print(f"\n--- Samples (100 shots) ---")
samples_at_zero = mz_samples(0.0)
print(f"For phi = 0.0, first 10 samples: {samples_at_zero[:10]}")
print(f"Mean of samples at phi = 0.0: {np.mean(samples_at_zero):.4f}")
```

```python
samples_at_pi = mz_samples(np.pi)
print(f"For phi = {np.pi:.2f}, first 10 samples: {samples_at_pi[:10]}")
print(f"Mean of samples at phi = {np.pi:.2f}: {np.mean(samples_at_pi):.4f}")
```

- `qml.counts()`: Similar to `qml.sample()`, `qml.counts()` returns a dictionary of outcomes and their frequencies, which can be more convenient for large numbers of shots.

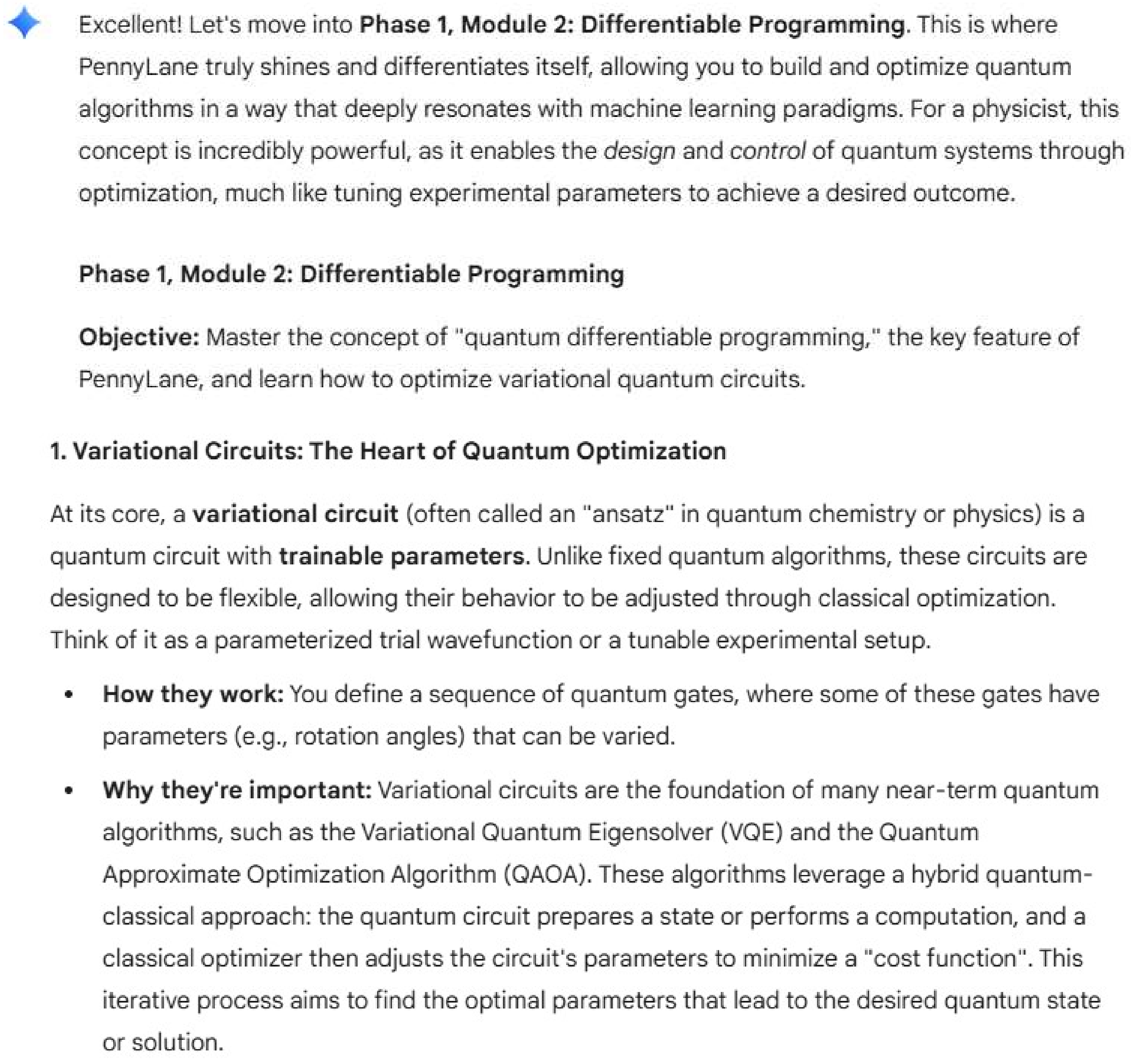
  Python

  ```python
  @qml.qnode(dev_shots)
  def mz_counts(phi):
      qml.Hadamard(wires=0)
      qml.PhaseShift(phi, wires=0)
      qml.Hadamard(wires=0)
      return qml.counts(wires=0) # Counts of computational basis states on wi

  print(f"\n--- Counts (100 shots) ---")
  counts_at_zero = mz_counts(0.0)
  print(f"For phi = 0.0: {counts_at_zero}")
  counts_at_pi = mz_counts(np.pi)
  print(f"For phi = {np.pi:.2f}: {counts_at_pi}")
  ```

Take your time to run these examples, modify parameters, and observe the outputs. Understanding these foundational elements is crucial before we move on to differentiable programming and more complex algorithms.

Phase 1: Module 2: Differentiable Programming!

03 August 2025 10:47



Excellent! Let's move into **Phase 1, Module 2: Differentiable Programming**. This is where PennyLane truly shines and differentiates itself, allowing you to build and optimize quantum algorithms in a way that deeply resonates with machine learning paradigms. For a physicist, this concept is incredibly powerful, as it enables the *design* and *control* of quantum systems through optimization, much like tuning experimental parameters to achieve a desired outcome.

Phase 1, Module 2: Differentiable Programming

Objective: Master the concept of "quantum differentiable programming," the key feature of PennyLane, and learn how to optimize variational quantum circuits.

1. Variational Circuits: The Heart of Quantum Optimization

At its core, a **variational circuit** (often called an "ansatz" in quantum chemistry or physics) is a quantum circuit with **trainable parameters.** Unlike fixed quantum algorithms, these circuits are designed to be flexible, allowing their behavior to be adjusted through classical optimization. Think of it as a parameterized trial wavefunction or a tunable experimental setup.

- **How they work:** You define a sequence of quantum gates, where some of these gates have parameters (e.g., rotation angles) that can be varied.
- **Why they're important:** Variational circuits are the foundation of many near-term quantum algorithms, such as the Variational Quantum Eigensolver (VQE) and the Quantum Approximate Optimization Algorithm (QAOA). These algorithms leverage a hybrid quantum-classical approach: the quantum circuit prepares a state or performs a computation, and a classical optimizer then adjusts the circuit's parameters to minimize a "cost function". This iterative process aims to find the optimal parameters that lead to the desired quantum state or solution.

2. Automatic Differentiation: PennyLane's Magic

Automatic differentiation (AD) is the ability of a software library to compute the exact derivatives of arbitrary numerical code. In PennyLane, this means you can automatically compute the gradients of your quantum computations with respect to their input parameters. This is what makes quantum algorithms "trainable".

- **Contrast with other methods:**
    - **Symbolic Differentiation:** Provides an exact mathematical expression for the derivative, but can be complex for intricate functions.
    - **Numerical Differentiation (Finite Differences):** Approximates the derivative by evaluating the function at two slightly different points. While always possible, its accuracy depends on the step size (
$epsilon$), and a very small
$epsilon$ can lead to instability and floating-point errors.
    - **Automatic Differentiation:** Unlike numerical differentiation, AD is an *exact* method. It works by breaking down a function into elementary operations (like addition, multiplication) for which derivatives are known, and then propagating these derivatives through a computational graph. The output is the numerical value of the derivative.
- **Parameter-Shift Rule:** Many quantum operations in PennyLane use "parameter-shift rules" for automatic differentiation. This method expresses the gradient of a function as a combination of that function evaluated at two different points, providing an exact derivative without the instability issues of finite differences.

This capability to compute quantum gradients means that quantum computations can be seamlessly integrated into automatically differentiable hybrid computation pipelines, just like classical neural networks.

3. Optimization: Guiding the Quantum Circuit

Once you can compute gradients, you can use **classical optimizers** to adjust the parameters of your variational circuit. PennyLane integrates with popular classical optimization libraries and provides its own built-in optimizers.

- **The Optimization Loop:**
  1. **Define a QNode:** Your quantum circuit with trainable parameters.
  2. **Define a Cost Function:** A classical function that quantifies how "good" the output of your QNode is (e.g., squared error from a target value, energy of a Hamiltonian).
  3. **Initialize Parameters:** Start with some initial values for your circuit's parameters.
  4. **Choose an Optimizer:** Select a classical optimizer (e.g., `qml.GradientDescentOptimizer`, `qml.AdamOptimizer`).
  5. **Iterate:** In each step of the optimization:
     - The optimizer calls the cost function.
     - PennyLane automatically computes the gradient of the cost function with respect to the QNode's parameters.
     - The optimizer uses these gradients to update the parameters in a direction that minimizes the cost.
  6. **Converge:** Repeat until the cost function is minimized or a certain number of steps are reached.

This hybrid quantum-classical approach allows the classical optimizer to explore the parameter space of the quantum circuit efficiently, finding the optimal settings for your quantum computation.

Actionable Steps:

Step 1: Revisit the "Qubit Rotation" Demo with a Focus on Optimization

In Module 1, we briefly touched upon the optimization part of the qubit rotation. Now, let's fully implement it, focusing on the differentiable aspect and the optimization loop. The goal is to rotate a single qubit from the initial state $|0$
$rangle$ to $|1$
$rangle$ by optimizing two rotation angles. This means we want the expectation value of `qml.PauliZ(0)` to be as close to -1 as possible.

1. **Setup:** Ensure your PennyLane environment is active.
2. **Code Implementation:**

2. **Code Implementation:**

Python

```python
import pennylane as qml
from pennylane import numpy as np # Use PennyLane's NumPy for automatic differe

# 1. Define a Device
dev = qml.device("default.qubit", wires=1)

# 2. Define your Quantum Function (QNode)
@qml.qnode(dev)
def circuit(params):
    # params for RX, params for RY
    qml.RX(params, wires=0)
    qml.RY(params, wires=0)
    return qml.expval(qml.PauliZ(0)) # We want this to be -1 for |1> state

# 3. Define the Cost Function
# Our target expectation value is -1 (for the |1> state)
def cost(params):
    return (circuit(params) - (-1))**2 # Squared difference from target
```

```
# 4. Initialize Parameters
# It's crucial to set requires_grad=True for parameters you want to optimize
initial_params = np.array([0.54, 0.12], requires_grad=True)
print(f"Initial parameters: {initial_params}")
print(f"Initial cost: {cost(initial_params):.4f}")
print(f"Initial expectation value: {circuit(initial_params):.4f}\n")

# 5. Choose an Optimizer
# We'll use the GradientDescentOptimizer
opt = qml.GradientDescentOptimizer(stepsize=0.4) # Learning rate

# 6. Optimize the Parameters
params = initial_params
num_steps = 100

print("Optimization Steps:")
for i in range(num_steps):
    # The opt.step() function automatically computes the gradient and updates p
    params = opt.step(cost, params)

    if (i + 1) % 10 == 0: # Print progress every 10 steps
        current_cost = cost(params)
        current_expval = circuit(params)
        print(f"Step {i+1}: Cost = {current_cost:.4f}, Expectation = {current_e

print(f"\nOptimized parameters: {params}")
print(f"Final cost: {cost(params):.4f}")
print(f"Final expectation value: {circuit(params):.4f}")

# Verify the state (optional, for understanding)
# You can inspect the state vector if using a state-vector simulator like defau
# Note: This is for understanding, not typically done in a real optimization lo
# print(f"Final state vector: {qml.state()(params)}")
```

Key Takeaway: Notice how `opt.step(cost, params)` handles the gradient computation implicitly. This is PennyLane's automatic differentiation at work. You define the circuit and the cost, and PennyLane figures out how to differentiate through the quantum operations.

```
{current_expval:.4f}")
```

Step 2: Implement a Simple "Function Fitting" Problem

Now, let's apply this concept to a slightly more complex task: using a quantum circuit to approximate a classical function. This is a fundamental concept in Quantum Machine Learning.

Objective: Use a simple parameterized quantum circuit to approximate a target classical function, for example, `f(x) = sin(x)`.

1. **Target Function:** We'll try to fit `f(x) = sin(x)` over a certain range.
2. **Quantum Circuit Design:** We need a circuit that takes a classical input `x` and produces a classical output (an expectation value) that approximates `sin(x)`. A common approach is to encode `x` into a rotation angle and then use a variational layer.

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# 1. Define a Device
dev = qml.device("default.qubit", wires=1)

# 2. Define the Quantum Circuit for Function Fitting
@qml.qnode(dev)
def quantum_function_fitter(x, params):
    # Encode the input 'x' into a rotation
    qml.RX(x, wires=0)

    # Apply a variational layer with trainable parameters
    # This layer learns to transform the encoded input
    qml.RY(params, wires=0)
    qml.RZ(params, wires=0)
    qml.RX(params, wires=0)

    # Measure the expectation value of PauliZ as the output
    return qml.expval(qml.PauliZ(0))
```

```python
# 3. Define the Cost Function
# We want to minimize the squared error between the quantum circuit's output
# and the target classical function (sin(x))
def cost(params, x_data, y_data):
    predictions = [quantum_function_fitter(x, params) for x in x_data]
    return np.mean((np.array(predictions) - y_data)**2)

# 4. Generate Training Data
# Let's fit sin(x) over the range [0, 2*pi]
x_data = np.linspace(0, 2 * np.pi, 30, requires_grad=False) # Input data
y_data = np.sin(x_data) # Target output data

# 5. Initialize Parameters for the variational layer
# These are the parameters the optimizer will adjust
initial_params = np.random.rand(3) * 2 * np.pi # 3 parameters for RY, RZ, RX
initial_params.requires_grad = True # Make them trainable

print(f"Initial parameters: {initial_params}")
print(f"Initial cost: {cost(initial_params, x_data, y_data):.4f}\n")

# 6. Choose an Optimizer
opt = qml.AdamOptimizer(stepsize=0.1) # Adam is often good for ML tasks
```

```python
# 7. Optimize the Parameters
params = initial_params
num_steps = 200

print("Optimization Steps:")
for i in range(num_steps):
    # The opt.step() function takes the cost function, current parameters,
    # and any additional arguments for the cost function (x_data, y_data)
    params = opt.step(cost, params, x_data, y_data)

    if (i + 1) % 20 == 0:
        current_cost = cost(params, x_data, y_data)
        print(f"Step {i+1}: Cost = {current_cost:.4f}")

print(f"\nOptimized parameters: {params}")
print(f"Final cost: {cost(params, x_data, y_data):.4f}")

# 8. Visualize the Results
# Get predictions from the optimized circuit
predictions_optimized = [quantum_function_fitter(x, params) for x in x_data]
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_data, y_data, 'o', label='Target sin(x) Data')
plt.plot(x_data, predictions_optimized, '-', label='Quantum Circuit Approximati
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Quantum Circuit Function Fitting (sin(x))")
plt.legend()
plt.grid(True)
plt.show()
```

Experiment:

- Try different initial parameters.
- Change the number of optimization steps.
- Modify the `stepsize` of the optimizer.
- Try to fit a different simple function (e.g., `f(x) = x**2`). You might need to adjust the circuit's complexity (add more variational layers or different gates) for more complex functions.

This function fitting example clearly demonstrates the power of differentiable programming in PennyLane. You're using a quantum circuit as a "model" and training it using classical optimization techniques, which is the essence of Quantum Machine Learning.

Take your time to run these examples, understand each part, and experiment with modifications. This foundational understanding of variational circuits, automatic differentiation, and optimization is critical for everything that follows, especially when we move into quantum optics applications.

Let me know when you're ready, and we'll proceed to Phase 2: Quantum Optics Focus!

Phase 2, Module 3: Continuous-Variable Quantum Computing

03 August 2025 11:01

Excellent! Welcome to **Phase 2: Quantum Optics Focus**. This is where your deep understanding of physics will truly come alive within PennyLane. We'll shift from general quantum computing concepts to the specific tools and techniques for modeling and manipulating continuous-variable (CV) quantum systems, which are the natural language of quantum optics.

Phase 2, Module 3: Continuous-Variable Quantum Computing

Objective: Learn to model quantum optical systems using PennyLane's CV devices and gates, and understand how to measure relevant observables.

1. Introduction to Continuous-Variable (CV) Quantum Computing

In contrast to qubit-based quantum computing, which uses discrete two-level systems, **Continuous-Variable (CV) quantum computing** deals with systems that have an infinite-dimensional Hilbert space. The most common example, and the one most relevant to quantum optics, is the **bosonic harmonic oscillator**, often referred to as a "qumode" in this context.

Instead of Pauli operators, CV systems are described by continuous quantum operators like position (
$hatx$) and momentum (
$hatp$) quadratures, which satisfy canonical commutation relations. These quadratures are directly analogous to the electric and magnetic field components of light. By combining multiple qumodes and applying sequences of quantum gates, we can implement general CV quantum computations.

2. CV Devices in PennyLane

PennyLane provides specialized devices for simulating continuous-variable systems:

- `default.gaussian`: This is PennyLane's built-in Gaussian state simulator. It's ideal for simulating circuits composed entirely of Gaussian operations (e.g., displacement, squeezing, beamsplitters), as these operations preserve the Gaussian nature of quantum states. It supports parameter-shift differentiability and is available on all operating systems.

Python

```python
import pennylane as qml
from pennylane import numpy as np

# Initialize default.gaussian device with 1 qumode (wire)
dev_gaussian = qml.device("default.gaussian", wires=1)
print(f"Initialized Gaussian device: {dev_gaussian.name}")
```

- `strawberryfields.fock` : This device, part of the PennyLane-StrawberryFields plugin, is underpinned by Strawberry Fields' Fock basis simulator,. It represents quantum states in the Fock basis ($|0$
$rangle, |1$
$rangle,$
$dots, |D-1$
$rangle$), where `D` is the `cutoff_dim` parameter that limits the dimension of the Hilbert space.
    - **Key advantage:** It supports *any* continuous-variable operation, including both Gaussian and **non-Gaussian** operations (like the Kerr or CubicPhase gates), which are crucial for universal CV quantum computing,.
    - **Consideration:** Simulation time and memory grow much faster with the number of modes compared to qubit-based simulators due to the Fock basis truncation.

Python

```python
# You might need to install the Strawberry Fields plugin if you haven't already
#!pip install pennylane-sf

# Initialize strawberryfields.fock device with 2 qumodes and a cutoff dimensior
dev_fock = qml.device("strawberryfields.fock", wires=2, cutoff_dim=10) #
print(f"Initialized Fock device: {dev_fock.name} with cutoff_dim={dev_fock.cutc
```

```
naven't already:

utoff dimension
im=10) #
={dev_fock.cutoff_dim}")
```

3. CV Gates in PennyLane

PennyLane offers a rich set of CV gates that directly correspond to operations in quantum optics,:

- `qml.Displacement(alpha, phi, wires)` : Shifts the phase space by a complex number $alpha = re^{iphi}$. This is like applying a coherent state displacement operator. You can specify `r` and `phi` or a complex `alpha` .,
- `qml.Rotation(phi, wires)` : Rotates the phase space by an angle
$phi.$,
- `qml.Squeezing(r, phi, wires)` : Scales the phase space with a squeezing magnitude `r` and angle `phi` . This creates a squeezed vacuum state.,,
- `qml.Beamsplitter(theta, phi, wires)` : Represents a beam splitter interaction between

and angle `phi`. This creates a squeezed vacuum state.,,

- `qml.Beamsplitter(theta, phi, wires)`: Represents a beam splitter interaction between two qumodes. `theta` is the transmittivity angle (e.g., $pi/4$ for a 50-50 beam splitter), and `phi` is the phase angle.,,
- `qml.TwoModeSqueezing(r, phi, wires)`: Creates a two-mode squeezed vacuum state, which is a fundamental entangled state in quantum optics. It takes a squeezing magnitude `r` and phase `phi` and acts on two wires.,

- `qml.Kerr(kappa, wires)`: A non-Gaussian gate that introduces a Kerr nonlinearity, analogous to an optical Kerr medium.,
- `qml.CubicPhase(gamma, wires)`: Another non-Gaussian gate, introducing a cubic phase shift.,

Example: Applying CV Gates

Python

```python
@qml.qnode(dev_gaussian)
def cv_circuit_gaussian(r, phi_s, theta_bs, phi_bs):
    # Prepare a squeezed vacuum state on wire 0
    qml.Squeezing(r, phi_s, wires=0) #,

    # Apply a displacement to wire 0
    qml.Displacement(0.5, 0.0, wires=0) #,

    # Apply a beamsplitter between wire 0 and 1 (requires 2 wires)
    # Note: dev_gaussian was initialized with 1 wire, so let's re-initialize for 2
    # For this example, let's assume dev_gaussian is re-initialized with wires=2
    # qml.Beamsplitter(theta_bs, phi_bs, wires=) #,

    # For now, let's stick to 1 wire for simplicity to match dev_gaussian
    # If you want to use Beamsplitter, you'd need dev = qml.device("default.gaussi
    return qml.expval(qml.NumberOperator(0)) #,

# Example parameters
r_val = 0.5
phi_s_val = 0.0
theta_bs_val = np.pi / 4
phi_bs_val = np.pi / 2

# print(cv_circuit_gaussian(r_val, phi_s_val, theta_bs_val, phi_bs_val)) # This wo
```

```
 to match dev_gaussian
ev = qml.device("default.gaussian", wires=2)







_bs_val, phi_bs_val)) # This would run if dev_gaussian had 2 wires
```

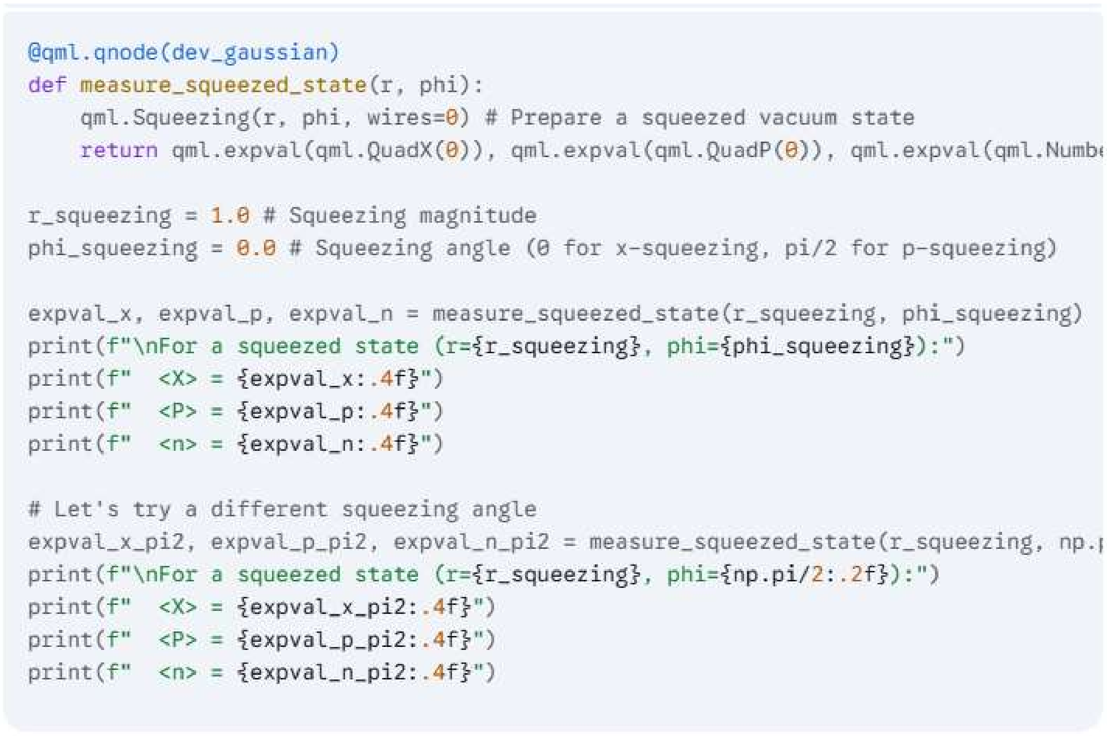
4. Quadrature and Photon Number Measurements

To extract information from CV quantum states, PennyLane provides specific observables:

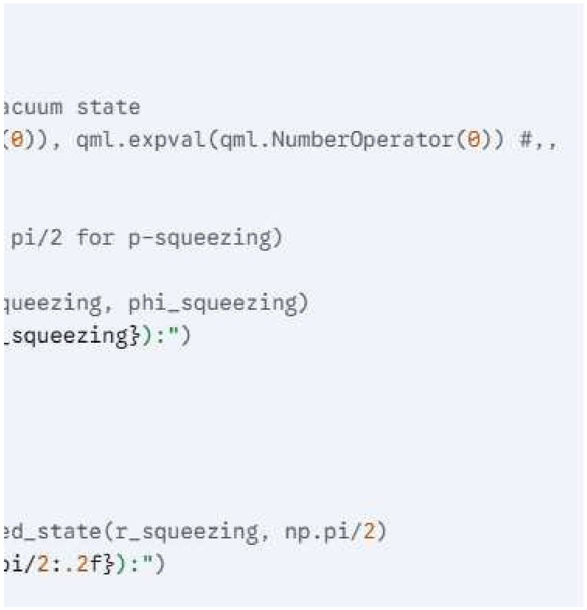
- `qml.QuadX(wires)` : Measures the position quadrature
  $hatx.,$
- `qml.QuadP(wires)` : Measures the momentum quadrature
  $hatp.,$
- `qml.QuadOperator(phi, wires)` : Measures the generalized quadrature
  $hatx\_$
  $phi =$
  $hatx$
  cos
  $phi +$
  $hatp$
  sin
  $phi.,$
- `qml.NumberOperator(wires)` : Measures the photon number operator
  $hatn.,$

Example: Measuring Quadratures and Photon Number

```
Python
```



```python
@qml.qnode(dev_gaussian)
def measure_squeezed_state(r, phi):
    qml.Squeezing(r, phi, wires=0) # Prepare a squeezed vacuum state
    return qml.expval(qml.QuadX(0)), qml.expval(qml.QuadP(0)), qml.expval(qml.Numbe

r_squeezing = 1.0 # Squeezing magnitude
phi_squeezing = 0.0 # Squeezing angle (0 for x-squeezing, pi/2 for p-squeezing)

expval_x, expval_p, expval_n = measure_squeezed_state(r_squeezing, phi_squeezing)
print(f"\nFor a squeezed state (r={r_squeezing}, phi={phi_squeezing}):")
print(f"  <X> = {expval_x:.4f}")
print(f"  <P> = {expval_p:.4f}")
print(f"  <n> = {expval_n:.4f}")

# Let's try a different squeezing angle
expval_x_pi2, expval_p_pi2, expval_n_pi2 = measure_squeezed_state(r_squeezing, np.p
print(f"\nFor a squeezed state (r={r_squeezing}, phi={np.pi/2:.2f}):")
print(f"  <X> = {expval_x_pi2:.4f}")
print(f"  <P> = {expval_p_pi2:.4f}")
print(f"  <n> = {expval_n_pi2:.4f}")
```



```
acuum state
(0)), qml.expval(qml.NumberOperator(0)) #,,

 pi/2 for p-squeezing)

queezing, phi_squeezing)
_squeezing}):")




ed_state(r_squeezing, np.pi/2)
pi/2:.2f}):")
```

Actionable Step 1: Demos (Summary)

You were asked to read the "Gaussian transformation" and "Function fitting with a photonic quantum neural network" demos. Here's a summary of what you would learn:

- **Gaussian Transformation Demos:** These typically illustrate how sequences of Gaussian gates (Displacement, Squeezing, Rotation, Beamsplitter) transform Gaussian states (like the vacuum or coherent states). You'd see how to represent these transformations and measure the resulting quadrature variances, which are key to understanding squeezing and entanglement.
- **Function Fitting with a Photonic Quantum Neural Network:** This demo would show how to use CV gates to build a quantum neural network (QNN) for classical function approximation, similar to the qubit-based example you did in Module 2. It would highlight how to encode classical data into CV states (e.g., using `qml.SqueezingEmbedding`,) and use CV measurements as the output. This demonstrates the hybrid quantum-classical approach in a CV context.

Actionable Step 2: Implement Squeezed States and Two-Mode Squeezed Vacuum (TMSV)

Let's implement the generation of these fundamental quantum optical states and characterize them.

A. Single-Mode Squeezed State Generation

A single-mode squeezed vacuum state is created by applying the `qml.Squeezing` operation to a vacuum state. We can then measure the variance of its quadratures to confirm squeezing. For a squeezed state, the variance of one quadrature is reduced below the vacuum noise level, while the other is increased.

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Device for simulating Gaussian states
dev_gaussian_squeezing = qml.device("default.gaussian", wires=1) #

@qml.qnode(dev_gaussian_squeezing)
def squeezed_state_circuit(r, phi):
    qml.Squeezing(r, phi, wires=0) #
    return qml.var(qml.QuadX(0)), qml.var(qml.QuadP(0)) # Measure variances of X ar

# Squeezing parameters
squeezing_magnitude = np.linspace(0, 2, 20) # Vary squeezing strength
squeezing_angle_x = 0.0 # Squeeze along X-quadrature
squeezing_angle_p = np.pi / 2 # Squeeze along P-quadrature

x_variances =
p_variances =
for r in squeezing_magnitude:
    var_x, var_p = squeezed_state_circuit(r, squeezing_angle_x)
    x_variances.append(var_x)
    p_variances.append(var_p)

plt.figure(figsize=(8, 5))
plt.plot(squeezing_magnitude, x_variances, label='Var(X) (phi=0)')
plt.plot(squeezing_magnitude, p_variances, label='Var(P) (phi=0)')
plt.axhline(y=0.5, color='r', linestyle='--', label='Vacuum Noise (0.5)') # Vacuum noise variance is 0.5 (hbar=2 convention)
plt.xlabel("Squeezing Magnitude (r)")
plt.ylabel("Quadrature Variance")
plt.title("Single-Mode Squeezing: Variance vs. Squeezing Magnitude")
plt.legend()
plt.grid(True)
plt.show()

# You should see Var(X) decrease below 0.5 and Var(P) increase above 0.5 for phi=0.
```

B. Two-Mode Squeezed Vacuum (TMSV) State Generation

The TMSV state is a maximally entangled Gaussian state, crucial for quantum communication and sensing. It's generated using `qml.TwoModeSqueezing`. We can characterize its entanglement by looking at the variance of the difference and sum of quadratures, or by photon number correlations.

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Device for simulating Gaussian states with 2 wires
dev_gaussian_tmsv = qml.device("default.gaussian", wires=2) #

@qml.qnode(dev_gaussian_tmsv)
def tmsv_circuit(r, phi):
    qml.TwoModeSqueezing(r, phi, wires=) #
    # Measure the variance of the difference of position quadratures (x0 - x1)
    # and the sum of momentum quadratures (p0 + p1)
    return qml.var(qml.QuadX(0) - qml.QuadX(1)), qml.var(qml.QuadP(0) + qml.QuadP(:

# Squeezing parameters
squeezing_magnitude_tmsv = np.linspace(0, 2, 20)
squeezing_phase_tmsv = 0.0 # For simplicity, set phase to 0

diff_x_variances =
sum_p_variances =

for r in squeezing_magnitude_tmsv:
    var_diff_x, var_sum_p = tmsv_circuit(r, squeezing_phase_tmsv)
    diff_x_variances.append(var_diff_x)
    sum_p_variances.append(var_sum_p)

plt.figure(figsize=(8, 5))
plt.plot(squeezing_magnitude_tmsv, diff_x_variances, label='Var(X0 - X1)')
plt.plot(squeezing_magnitude_tmsv, sum_p_variances, label='Var(P0 + P1)')
plt.axhline(y=0.0, color='r', linestyle='--', label='Ideal Entanglement (0)') # Ide
plt.xlabel("Squeezing Magnitude (r)")
plt.ylabel("Quadrature Variance")
plt.title("Two-Mode Squeezing: Entanglement Indicators")
plt.legend()
plt.grid(True)
plt.show()

# For a TMSV state, both Var(X0 - X1) and Var(P0 + P1) should ideally approach 0 a:
# This indicates strong entanglement.
# Ideally, these variances go to 0

ı 0 as r increases.
```

Visualizing CV States (Wigner Function):
While PennyLane itself doesn't have a direct `qml.wigner_function` plotting utility, you can extract the state information (e.g., means and covariance matrix for Gaussian states) and use external libraries or custom plotting functions to visualize the Wigner function. The Wigner function is a quasi-probability distribution in phase space that provides a visual representation of the quantum state. For non-Gaussian states, plotting the Wigner function can be more complex and might require the `strawberryfields.fock` device and specific methods from Strawberry Fields.

Actionable Step 3: Optimize Squeezing or Entanglement

Let's use optimization to find the parameters that maximize the squeezing in a single-mode squeezed state. This means minimizing the variance of one quadrature.

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

dev_opt_squeezing = qml.device("default.gaussian", wires=1) #

@qml.qnode(dev_opt_squeezing)
def optimize_squeezing_circuit(params):
    # params = squeezing magnitude (r)
    # params = squeezing angle (phi)
    qml.Squeezing(params, params, wires=0) #
    return qml.var(qml.QuadX(0)) # We want to minimize the variance of the X quadra

# Define the cost function: simply the variance of QuadX
def cost(params):
    return optimize_squeezing_circuit(params)
```

```
# Initial parameters: (r, phi)
# Start with some small squeezing and an arbitrary angle
initial_params = np.array([0.1, 0.5], requires_grad=True)
print(f"Initial parameters: {initial_params}")
print(f"Initial cost (Var(X)): {cost(initial_params):.4f}\n")

# Optimizer
opt = qml.GradientDescentOptimizer(stepsize=0.1)

# Optimization loop
params = initial_params
num_steps = 100

costs =
r_values =
phi_values =

print("Optimization Steps:")
for i in range(num_steps):
    params = opt.step(cost, params)
    current_cost = cost(params)
    costs.append(current_cost)
    r_values.append(params)
    phi_values.append(params)

    if (i + 1) % 10 == 0:
        print(f"Step {i+1}: Cost (Var(X)) = {current_cost:.4f}, r = {params:.4f},

print(f"\nOptimized parameters: {params}")
print(f"Final cost (Var(X)): {cost(params):.4ff}")

# Plotting the optimization progress
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(range(num_steps), costs)
plt.xlabel("Optimization Step")
plt.ylabel("Cost (Var(X))")
plt.title("Cost Function (Var(X)) during Optimization")
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(range(num_steps), r_values, label='Squeezing Magnitude (r)')
plt.plot(range(num_steps), phi_values, label='Squeezing Angle (phi)')
plt.xlabel("Optimization Step")
plt.ylabel("Parameter Value")
plt.title("Parameter Evolution during Optimization")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# You should observe the cost (Var(X)) decreasing, and 'r' increasing while 'phi'
r = {params:.4f}, phi = {params:.4f}")
```

```
'phi' approaches 0 or pi.
```

This concludes Module 3. You've now been introduced to the core concepts of CV quantum computing in PennyLane, including devices, gates, measurements, and how to apply optimization to quantum optical problems.

Phase 2, Module 4: Quantum Machine Learning for Quantum Optics

03 August 2025 11:34

Phase 2, Module 4: Quantum Machine Learning for Quantum Optics

Objective: Begin applying machine learning techniques to quantum optical problems, focusing on hybrid models and state classification.

1. Quantum-Classical Hybrid Models

As you've seen, PennyLane excels at building **hybrid quantum-classical models**. These models combine the strengths of quantum circuits for processing quantum information with classical machine learning techniques for optimization and data handling. In quantum optics, this means you can use classical data (e.g., measurement results, desired state properties) to train quantum circuits that generate, manipulate, or classify quantum optical states.

2. Feature Encoding for CV Systems

To apply QML to classical data, you need a way to encode that data into a quantum state. For CV systems, PennyLane offers operations like `qml.SqueezingEmbedding`,. This template encodes features into the squeezing amplitudes or phases of modes.

Actionable Step 2: Design and Implement a Simple Classification Model (Coherent vs. Squeezed States)

Let's build a simple QML model to distinguish between two fundamental quantum optical states: a coherent state and a squeezed vacuum state.

Problem: Given a quantum state, determine if it's a coherent state or a squeezed vacuum state. We'll train a quantum circuit to classify them based on their quadrature expectation values.

1. **Generate Data:** We'll create a dataset of coherent and squeezed states with varying parameters.
2. **Quantum Circuit (Classifier Ansatz):** Design a parameterized CV circuit that can process the input state and produce an output that can be mapped to a classification label.
3. **Cost Function:** Define a cost function that measures the difference between the circuit's output and the true label.
4. **Optimization:** Train the circuit parameters using a classical optimizer.

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Device for CV simulations
dev_classifier = qml.device("default.gaussian", wires=1) #

# --- Data Generation ---
# We'll represent coherent states by their displacement (alpha)
# and squeezed states by their squeezing magnitude (r)
```

Python

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Device for CV simulations
dev_classifier = qml.device("default.gaussian", wires=1) #

# --- Data Generation ---
# We'll represent coherent states by their displacement (alpha)
# and squeezed states by their squeezing magnitude (r)
# For simplicity, we'll assume zero phase for both.

num_samples_per_class = 50

# Coherent states: vary displacement magnitude
coherent_magnitudes = np.random.uniform(0.1, 2.0, num_samples_per_class)
coherent_states_params = np.array([[mag, 0.0] for mag in coherent_magnitudes]) # [
coherent_labels = np.zeros(num_samples_per_class) # Label 0 for coherent
```

```python
# Squeezed states: vary squeezing magnitude
squeezed_magnitudes = np.random.uniform(0.1, 2.0, num_samples_per_class)
squeezed_states_params = np.array([[mag, 0.0] for mag in squeezed_magnitudes]) # [
squeezed_labels = np.ones(num_samples_per_class) # Label 1 for squeezed

# Combine data
all_params = np.vstack((coherent_states_params, squeezed_states_params))
all_labels = np.hstack((coherent_labels, squeezed_labels))

# Shuffle data
permutation = np.random.permutation(len(all_labels))
shuffled_params = all_params[permutation]
shuffled_labels = all_labels[permutation]

# --- Quantum Classifier Circuit ---
@qml.qnode(dev_classifier)
def quantum_classifier(input_state_type, input_param, classifier_weights):
    # input_state_type: 0 for coherent, 1 for squeezed
    # input_param: magnitude for coherent/squeezed state
```

```
# [magnitude, phase]
```

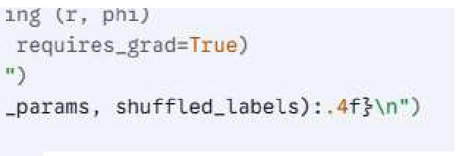
```
    if input_state_type == 0: # Coherent state
        qml.CoherentState(input_param, input_param, wires=0) #
    elif input_state_type == 1: # Squeezed state
        qml.SqueezedState(input_param, input_param, wires=0) #

    # Variational layer for classification
    # A simple layer of Displacement, Rotation, Squeezing
    qml.Displacement(classifier_weights, classifier_weights, wires=0) #,
    qml.Rotation(classifier_weights, wires=0) #,
    qml.Squeezing(classifier_weights, classifier_weights, wires=0) #,

    # Measure expectation value of NumberOperator for classification output
    # We'll map this to a probability for binary classification
    return qml.expval(qml.NumberOperator(0)) #,

# --- Cost Function ---
def square_loss(labels, predictions):
    return np.mean((labels - predictions)**2)
```

```
def cost(classifier_weights, input_params_data, input_labels_data):
    predictions =
    for i in range(len(input_params_data)):
        # The output of NumberOperator is non-negative, we can scale it to be betw
        # A simple sigmoid-like scaling or normalization can be applied if needed f
        # For simplicity, let's just use the raw expectation value and let the opt
        output_expval = quantum_classifier(input_labels_data[i], input_params_data
        predictions.append(output_expval)

    # Simple scaling of predictions to roughly  range for binary classification
    # This is a heuristic; a more robust approach might use a classical sigmoid la
    scaled_predictions = np.tanh(np.array(predictions)) # Using tanh to map to [-1
    scaled_predictions = (scaled_predictions + 1) / 2 # Map to

    return square_loss(input_labels_data, scaled_predictions)

# --- Optimization ---
# Initialize trainable weights for the classifier circuit
# 5 parameters for Displacement (r, phi), Rotation (phi), Squeezing (r, phi)
initial_classifier_weights = np.random.uniform(-np.pi, np.pi, 5, requires_grad=Tru
print(f"Initial classifier weights: {initial_classifier_weights}")
print(f"Initial cost: {cost(initial_classifier_weights, shuffled_params, shuffled_
```

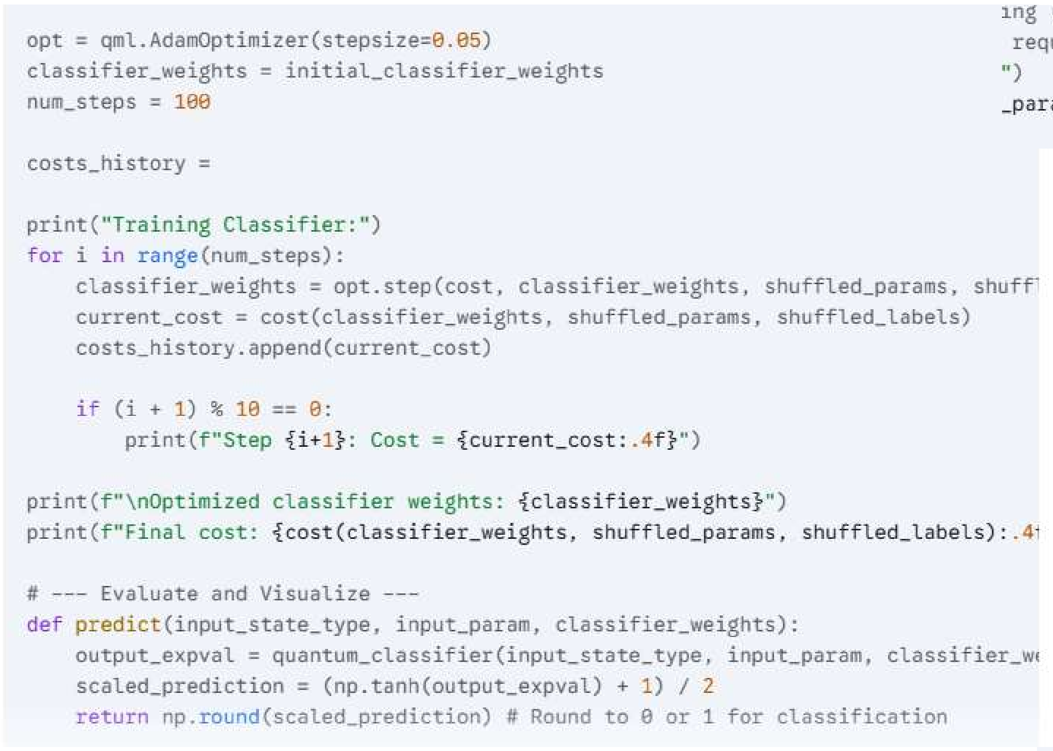
```
 to be between 0 and 1
 if needed for probability
let the optimizer learn to map it
params_data[i], classifier_weights)


fication
 sigmoid layer
 map to [-1, 1] then scale to
```

```
ing (r, phi)
 requires_grad=True)
")
_params, shuffled_labels):.4f}\n")
```

```
opt = qml.AdamOptimizer(stepsize=0.05)
classifier_weights = initial_classifier_weights
num_steps = 100

costs_history =

print("Training Classifier:")
```



```
ing (r, phi)
 requires_grad=True)
")
_params, shuffled_labels):.4f}\n")
```



```
opt = qml.AdamOptimizer(stepsize=0.05)
classifier_weights = initial_classifier_weights
num_steps = 100

costs_history =

print("Training Classifier:")
for i in range(num_steps):
    classifier_weights = opt.step(cost, classifier_weights, shuffled_params, shuff
    current_cost = cost(classifier_weights, shuffled_params, shuffled_labels)
    costs_history.append(current_cost)

    if (i + 1) % 10 == 0:
        print(f"Step {i+1}: Cost = {current_cost:.4f}")

print(f"\nOptimized classifier weights: {classifier_weights}")
print(f"Final cost: {cost(classifier_weights, shuffled_params, shuffled_labels):.4

# --- Evaluate and Visualize ---
def predict(input_state_type, input_param, classifier_weights):
    output_expval = quantum_classifier(input_state_type, input_param, classifier_we
    scaled_prediction = (np.tanh(output_expval) + 1) / 2
    return np.round(scaled_prediction) # Round to 0 or 1 for classification
```

```
shuffled_params, shuffled_labels)
 shuffled_labels)





")
s, shuffled_labels):.4f}")


s):
ut_param, classifier_weights)

 classification



arams[i], classifier_weights)
```

```
correct_predictions = 0
for i in range(len(shuffled_labels)):
    predicted_label = predict(shuffled_labels[i], shuffled_params[i], classifier_we
    if predicted_label == shuffled_labels[i]:
        correct_predictions += 1

accuracy = correct_predictions / len(shuffled_labels)
print(f"\nClassification Accuracy: {accuracy * 100:.2f}%")

plt.figure(figsize=(8, 5))
plt.plot(range(num_steps), costs_history)
plt.xlabel("Optimization Step")
plt.ylabel("Cost")
plt.title("Classifier Training Cost")
plt.grid(True)
plt.show()
```

This example demonstrates a basic QML classification task using CV operations. You can expand on this by:

- Using more complex variational layers.
- Exploring different feature encoding strategies (e.g., `qml.SqueezingEmbedding`, to encode classical data directly).
- Adding more classes or different types of states.

Actionable Step 3: Sketching a QML Approach for Quantum State Tomography (QST)

Quantum State Tomography (QST) is the process of reconstructing an unknown quantum state from a set of measurements. In quantum optics, this means determining the density matrix or Wigner function of a light field.

QML Approach Sketch:

1. **Unknown Target State:** Imagine you have an unknown quantum optical state (e.g., a noisy squeezed state, or a complex multi-mode entangled state) that you want to characterize. In a simulation, you would generate this state using a known circuit.
2. **Parameterized Reconstruction Circuit (Ansatz):**
   - Design a variational quantum circuit (your "ansatz") that can prepare a wide variety of quantum optical states. This circuit will have many trainable parameters.
   - For CV systems, this might involve layers of `qml.Displacement`, `qml.Squeezing`, `qml.Rotation`, `qml.Beamsplitter`, and potentially non-Gaussian gates if the target state is non-Gaussian.

3. **Measurement Strategy:**
   - To characterize a state, you need to perform measurements in different bases. For CV states, this means measuring various quadrature observables (`qml.QuadX` , `qml.QuadP` , `qml.QuadOperator` at different angles) and potentially photon number statistics (`qml.NumberOperator`).
   - In a QST protocol, you would run your unknown state through a set of measurement circuits, and then run your parameterized reconstruction circuit through the *same* set of measurement circuits.
4. **Cost Function (Fidelity Loss):**
   - The goal is to make the state prepared by your reconstruction circuit as close as possible to the unknown target state.
   - A common cost function for QST is based on **fidelity**. You would compare the measurement outcomes (e.g., expectation values of various observables) from the unknown state with those from your parameterized reconstruction circuit.
   - The cost function would quantify the "distance" between these two sets of measurement outcomes. For Gaussian states, you might compare their covariance matrices. For general states, you might use a squared error between measured expectation values of a complete set of observables.
5. **Optimization:**
   - Use PennyLane's differentiable programming capabilities and a classical optimizer (like Adam or GradientDescent) to minimize the cost function.
   - The optimizer will adjust the parameters of your reconstruction circuit until its output measurements closely match those of the unknown target state.
6. **Reconstruction:** Once optimized, the parameters of your reconstruction circuit define the reconstructed quantum state. You can then use PennyLane's `qml.state()` (if using a state-vector simulator like `default.gaussian` or `strawberryfields.fock` with `shots=None`) to get the state vector or density matrix, or derive the Wigner function from the optimized parameters.

This concludes Phase 2. You've now gained practical experience with CV quantum computing and started applying QML concepts to quantum optics problems. Take your time to experiment with these examples, modify parameters, and think about how these tools could be applied to specific research questions in your field.