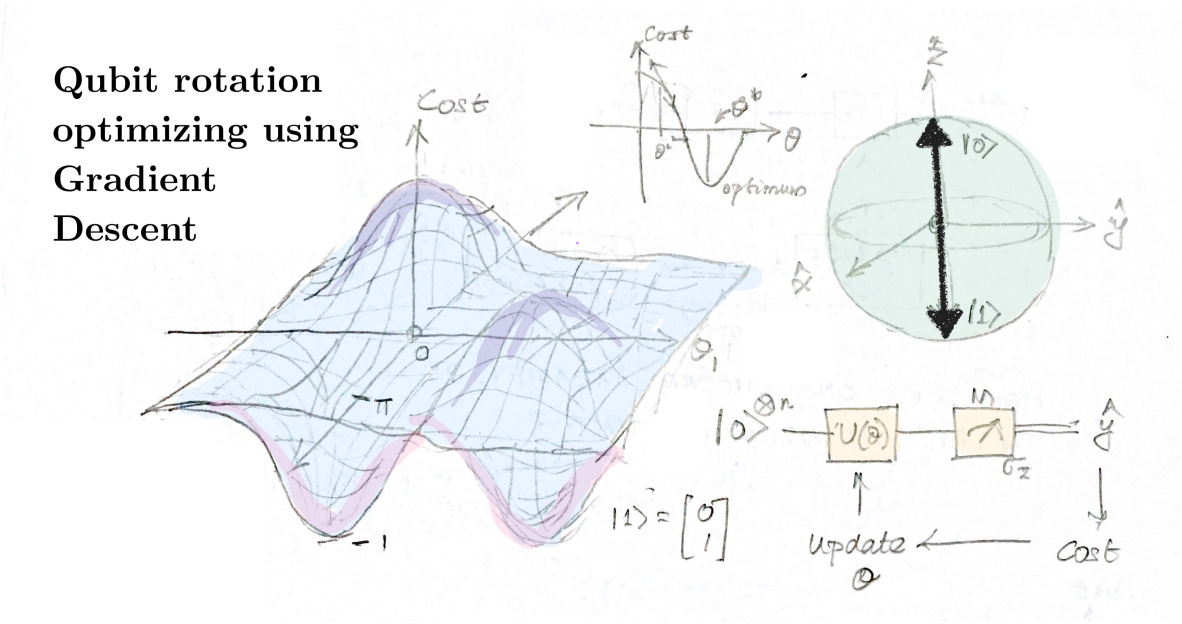
Qubit rotation optimization using gradient descent.

Rohith Krishna

16 September 2020

Quantum Machine Learning is a rapidly expanding and exciting field in quantum computing. This set of notes here are based on the lecture delivered by Amira Abbas [1] at the National Institute for Theoretical Physics as a part of their mini-school. Quantum Machine Learning, Qubit Rotation and Gradient Descent are first discussed. Then a simple circuit it constructed and the parameters of the machine learning model are optimized using Pennylane.



[1] Amira Abbas. NITheP Mini School Slides

Where can one apply quantum machine learning algorithms?

Let us first check out some really cool applications of quantum machine learning:

1. Logistics. In logistics, there are problems that involve the interaction of several variables - like transportations systems. Quantum computers and quantum algorithms are particularly useful in solving complex combinatorial problems.
2. Finance. Quantum Computing has already shown to have quadratic speed-ups in Monte Carlo methods that come up in pricing assets and derivatives, risk management, portfolio management and optimization etc., with much scope for innovation.
3. Material Science. The study of molecules is a very difficult problem with our current hardware and the hope is that quantum computers can handle these difficult computations and simulations in quantum chemistry - with applications in drug design, protein synthesis etc.
4. The Universe! - Richard Feynman famously quipped that nature is quantum and that any successful model of nature must incorporate quantum effects, which quantum computing essentially involves.

   "**Nature** isn't classical, dammit, and if you want to make a simulation of **nature**, you'd better make it **quantum** mechanical, and by golly it's a wonderful problem, because it doesn't look so easy." - Richard Feynman

Quantum Machine Learning

Researchers often cast quantum and classical computing methods into the following four blocks, based on the classical/quantum nature of the data generating system and the data processing device. [2]

- **Classical Classical (CC)**. This would typically correspond to classical data used in classical processing devices and would fall under standard supervised and unsupervised machine learning model. Using quantum-theory inspired ML models also fall under this category.
- **Quantum Classical (QC)**. Here machine learning methods are used to derive insights on quantum measurement data, or in learning phase-transitions in many-body systems, etc. Several experimental and computational data generated by quantum processes can be studied using classical devices.
- **Classical Quantum (CQ)**. CQ and QQ are used synonymously with *quantum machine learning*. In CQ, one uses classical data such as time series variables or sales data or text or images and uses specially designed quantum algorithms to solve ML problems. These could either be supervised learning problems or even

[2]Source: Schuld, Maria. Supervised learning with quantum computers. Springer, 2018

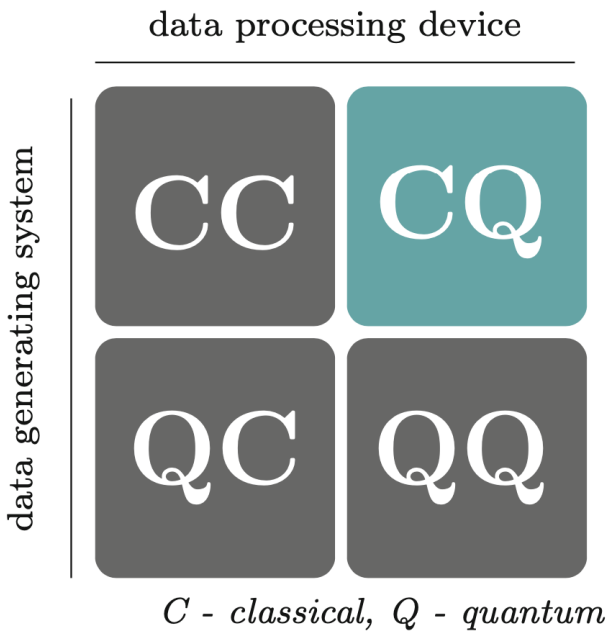


Figure 1: QML cast into 4 blocks

unsupervised learning methods that perform computationally complex tasks with ease.

- **Quantum Quantum (QQ)**. In physics and chemistry, one often wishes to simulate dynamics of a system that is inherently quantum mechanical in nature. When quantum computers are used in the generation/simulation of quantum mechanical system we refer to it as QQ.

Quantum Machine Learning Models

We know from *(classical)* Machine Learning that a parameterized model can be represented by $f(x;\theta)$. Here, x is some data that is input to the model and θ represents the model's parameters. The objective is to choose optimal parameters θ such that this model gives useful insights such as predictions of a response variable or predicted class of a target variable etc. Quantum Machine learning in some sense translates this process into a quantum system and the model output would be a quantum state with probabilities of occurence and expectation values, and one would have to make a measurement to obtain an output.

$$f(x;\theta) \mapsto |f(x;\theta)\rangle \tag{1}$$

The idea is that this Quantum Machine Learning (QML) model, which we are yet to define, has similar objectives as a CML model, but uses quantum mechanical algorithms

which inherently require a measurement process in some basis to obtain an output. It is pertinent now to ask whether 1. such a thing can be done? and if yes, 2. Is there an advantage in using QML models over standard ML models?

QML models are broadly of two kinds: *deterministic* and *variational* quantum models. The Deutsch-Jorza algorithms is an example for a **deterministic quantum model**. Here, the state $|\psi\rangle$ is transformed by circuits represented by $\boxed{U}$ to obtain the output y, with certainty.

$$|\psi\rangle \longrightarrow \boxed{U} \longrightarrow y \tag{2}$$

The other class that is popular in QML is the **variational quantum models**, which are applied in quantum chemistry amongst other fields. Examples for this model include: quantum variational eigensolver, quantum classifier, quantum support vector machines and quantum neural network. The basic premise of the variational model is that while it inputs a state $|\psi\rangle$, the output we obtain is a probability distribution over the possible outcomes, and one resorts to taking expectation values for the same $\langle y\rangle$. Also, note: the circuits $\boxed{U(\theta)}$ depend on model-specific parameters θ.

$$|\psi\rangle \longrightarrow \boxed{U(\theta)} \longrightarrow \langle y\rangle \tag{3}$$

Variational quantum models

One often finds several applications for the variational model, especially in the fields of chemistry and finance. This is because these variational models work really well. They run on near-term quantum hardware and are robust to noise in the system. Our simple circuit above can be extended when there are several input qubits, depicted below. Several qubits (n) are tranformed by the operations in $\boxed{U}$ and we obtain a function which is some kind of encoding of the data and parameters - $|f(x;\theta)\rangle$. The difference therefore, is that, since the outputs are probability distributions (or a superposition of output states), one has to perform a **measurement** in a certain basis to extract a real output, which in the ML parlance would be a predictor $\hat{y}$. The act of observing the state or measurement is represented by $\boxed{M}$. Note that circuits with classical input/output are represented by single lines ($\rightarrow$), while those which represent superposition of states are represented by double lines ($\Rightarrow$). Hence,

$$|\psi\rangle^{\bigotimes n} \longrightarrow \boxed{U(x;\theta)} \longrightarrow \boxed{M} \Longrightarrow \hat{y} \tag{4}$$

A note on measurement

Quantum mechanics introduces the method of calculating expectation values for states during measurement; this is discussed briefly in this section. Consider the pure state

$|0\rangle$. In matrix notation, this is the column vector, $|0\rangle = [1, 0]^T$. In the Bloch sphere, this would be along the z direction, and a measurement in σ_z would give a value of 1.

$$|0\rangle \longrightarrow \boxed{M}_{\sigma_z} \longrightarrow 1 \tag{5}$$

$$\langle 0| \sigma_z |0\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 \tag{6}$$

Likewise, one can apply the X gate to rotate $|0\rangle$ by 180 degrees in the Bloch sphere to get the vector $|1\rangle$ in the $-z$ direction. X is the state flip gate. Note however that these vectors are orthogonal in Hilbert space. A measurement of $|1\rangle$ in σ_z would result in -1. Therefore, the expectation value of output from the variational circuit $\langle \sigma_z \rangle$ ranges from +1 to -1.

$$|0\rangle \longrightarrow \boxed{X} \longrightarrow |1\rangle \longrightarrow \boxed{M}_{\sigma_z} \longrightarrow -1 \tag{7}$$

$$X \langle 0| = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \tag{8}$$

$$\langle 1| \sigma_z |1\rangle = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = -1 \tag{9}$$

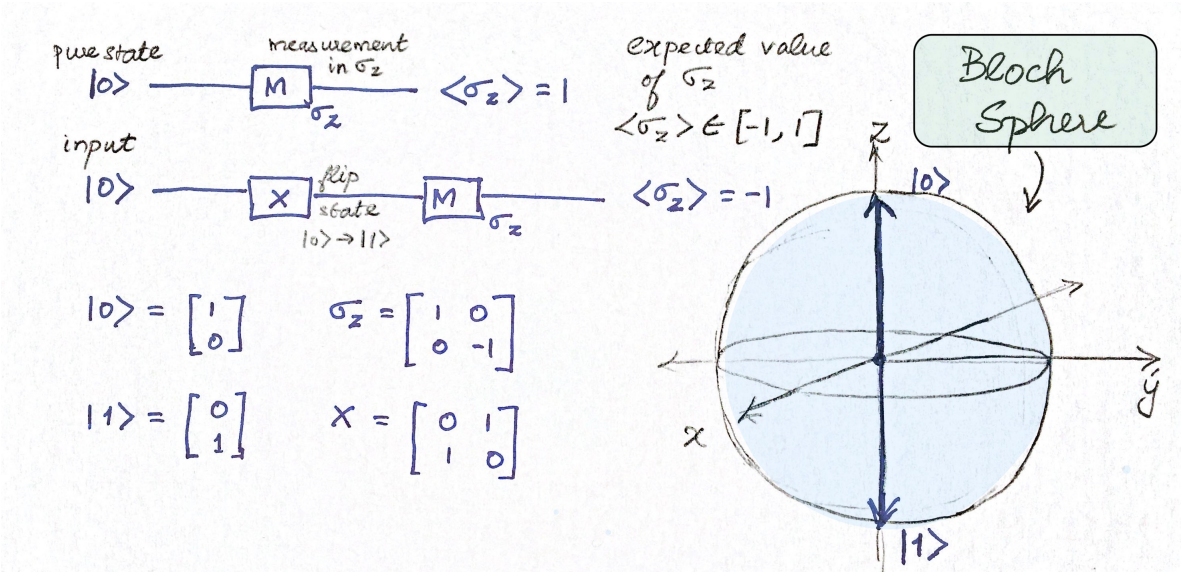


Figure 2: A visualization of the orthogonal $|0\rangle$ and $|1\rangle$ pure states in the Bloch sphere.

A simple variational circuit

Let us now get back to our model and perform the following parametrized operations. Take the $|0\rangle$ state and rotate this vector through x axis by an angle of θ_1 and then the y axis by θ_2 using the rotation matrices $R_x(\theta_1)$ and $R_y(\theta_2)$, consecutively. Suppose the state of the system after performing these operations is $|\psi\rangle$. Now, perform a σ_z measurement to obtain the its expected value. We can easily compute this value in steps shown below:

$$|0\rangle \longrightarrow \boxed{R_x(\theta_1)} \longrightarrow \boxed{R_y(\theta_2)} \longrightarrow |\psi\rangle \longrightarrow \boxed{M}_{\sigma_z} \longrightarrow \langle\sigma_z\rangle \tag{10}$$

Note the definitions of the rotation matrices:

$$R_x(\theta_1) = \begin{bmatrix} \cos\frac{\theta_1}{2} & -i\sin\frac{\theta_1}{2} \\ -i\sin\frac{\theta_1}{2} & \cos\frac{\theta_1}{2} \end{bmatrix}, \qquad R_y(\theta_2) = \begin{bmatrix} \cos\frac{\theta_2}{2} & -\sin\frac{\theta_2}{2} \\ \sin\frac{\theta_2}{2} & \cos\frac{\theta_2}{2} \end{bmatrix} \tag{11}$$

The state of the system after rotations: $|\psi\rangle$ is measured along the z direction as:

$$\langle\psi|\,\sigma_z\,|\psi\rangle = \langle 0|\;\; R_x^\dagger(\theta_1)\; R_y^\dagger(\theta_2)\; \sigma_z\; R_y(\theta_2)\; R_x(\theta_1)\;\; |0\rangle = \cos\theta_1 \cos\theta_2 \tag{12}$$

Now that we have computed the expected value of this variational circuit to be $\langle\psi|\,\sigma_z\,|\psi\rangle = \cos\theta_1\cos\theta_2$, we can now think about choosing these parameters: θ_1 and θ_2.

Circuit Optimization

In classical Machine Learning, one defines a cost function for the chosen model, using techniques that minimize the cost function, one obtains the optimal values for the parameters. For example, a decision trees model has a variety of hyperparameters such as maximum depth of trees, number of child nodes at a particular parent node etc.; a random forest method involves optimizing the out-of-bag score, again using several hyperparameters. In neural networks, weights are used for the same purpose. The general scheme of optimization for the variational model would be:

However, one wonders now about the analogue of this in the quantum setting. Let us consider the expected value of the output state as the cost function itself for this variational model. Therefore, define: cost function $= \cos\theta_1\cos\theta_2$. We can immediately reckon that the lower bound for this cost function is -1. Also, recall from the note earlier that if $|0\rangle$ is the initial state (which is true in this case), then the minimum value of -1 corresponds the $|1\rangle$ state. In order to perform the cost function minimization we use the method of *gradient descent.*

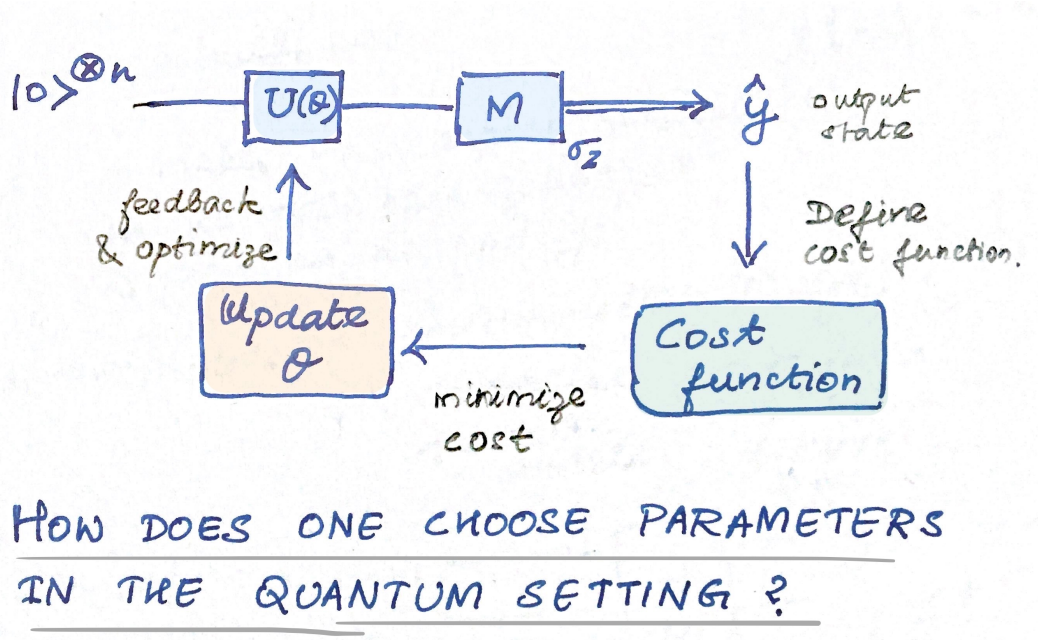


Figure 3: Cost function minimisation used in ML to find optimal model parameters.

Gradient Descent

From classical ML one is aware that gradient descent calculates the derivative of the cost function with respect to parameter at a particular point; then descends along the direction opposite to gradient vector, towards which the cost function becomes decreases and eventually reaches the minimum point. In our case consider a parameter vector $\theta = [\theta_1, \theta_2]$ and plot the cost as a function of θ. The plot of cost wrt. θ is shown. Then, start with a particular initial value for the parameter, say, $\theta = \theta^i$, and calculate the derivative of cost function at this point. Further, find the gradient direction and shift θ along the direction opposite to this direction. As one proceeds, we see that the θ^i's converge to a value corresponding to the minimum of the cost function.

In the quantum ML setting, a method of performing gradient descent is discussed in Schuld (2020).[3] The basic idea is that the parameter can be shifted by a small factor of s, above and below its initial value. The gradient then can be calculated from the difference in the measured outputs.

[3]Maria Schuld et al., Circuit-centric quantum classifiers., 2020.

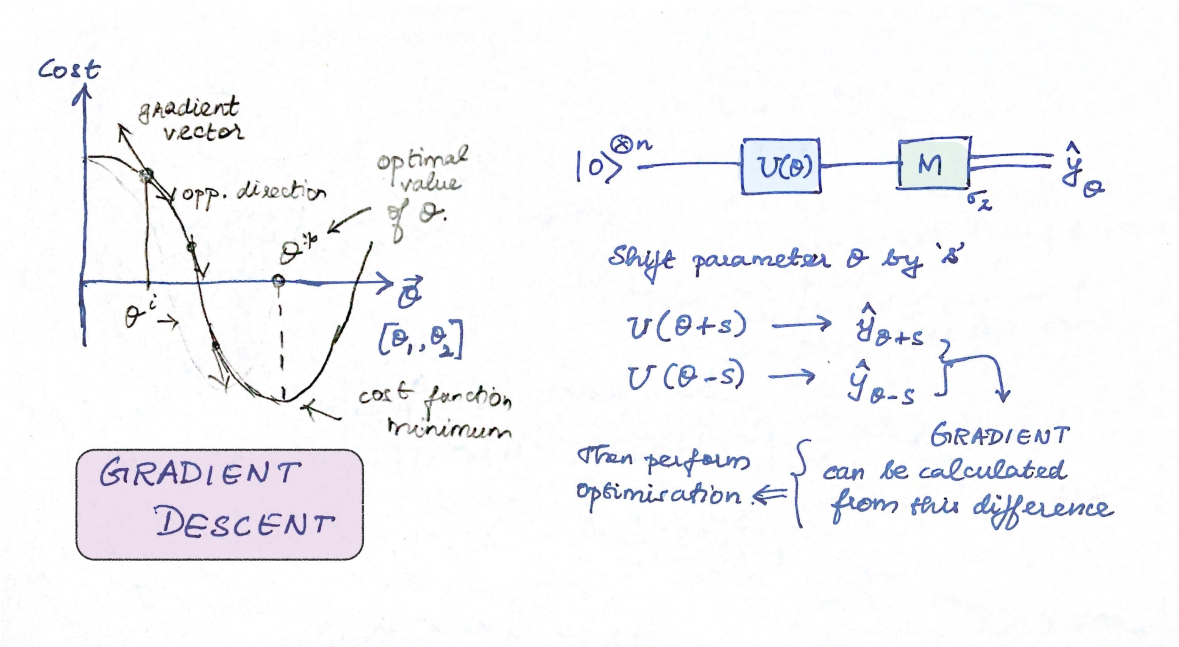


Figure 4: Gradient Descent method.

Coding Gradient Descent on Qubit Rotation using Pennylane

Importing packages

We use the python package called pennylane developed by Xanadu in performing this optimisation. [4]

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

Note. The numpy package is imported from pennylane and not directly.

Definition. A quantum device is a computational object that can apply operations featured in quantum mechanics and can return values resulting from measurement. Pennylane uses the function .device().[5]

Definition. QNodes are an abstract encapsulation of a quantum function, described by a quantum circuit. QNodes are bound to a particular quantum device, which is used to evaluate expectation and variance values of this circuit. Pennylane uses QNode class or .qnode() decorator.

[4]Pennylane Tutorials. Basic Gradient Descent on Qubit Rotation.
[5]Schuld, Maria. *Supervised learning with quantum computers.* Springer, 2018.

Initializing the quantum device

The pure state qubit simulator in pennylane is named `default.qubit`. We use this for our device `dev1`. The parameter `wires` corresponds to the number of qubits, 1 in our case.

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

Construct QNode

To tell pennylane that the circuit that is defined is supposed to be implemented on a quantum device, we use the `qnode()` decorator. Then we apply R_x and R_y rotations on the pure state $|0\rangle$. Also, we perform a measurement and calculate the expected value of σ_z.

```
# tell pennylane
@qml.qnode(dev)
def circuit(params):
    qml.RX(params[0], wires = 0)
# Rx is applied with the first parameter,on the first qubit
    qml.RY(params[1], wires = 0)
# Ry is applied with the second parameter,again on the same qubit.
    return qml.expval(qml.PauliZ(0))
    # return the expected value of    the Pauli Z operator on the qubit
```

Initialize the parameters

The parameters θ_1 and θ_2 are initialized. We also check if we obtain $\cos\theta_1 \cos\theta_2$ on applying the circuit.

```
params = [1.570796327,1.570796327]
circuit(params)
```

```
    0.0
```

```
params = [0,0]
circuit(params)
```

```
    1.0
```

```
params = [0,0.5]
circuit(params)
```

```
    0.8775825618903726
```

Calculate gradients

Using the same quantum device `dev` one can calculate the gradients of the function `circuit` that is encapsulated in the decorator `qnode`. The `.grad()` function is used to calculate the gradients (vector partial derivatives for the circuit). `argnum` is set to zero because the `circuit` takes one input `params`.

```
dcircuit = qml.grad(circuit, argnum=0)
print(dcircuit([0,0]))
```

```
[0.0, 0.0]
```

The above shows that for $\theta_1 = \theta_2 = 0$ the derivative on value of the expected value of the circuit (which is the function $\cos\theta_1 \cos\theta_2$) is 0. This is inline with theory.

Defining cost function and initial parameters

```
def cost(x):
    return circuit(x)

init_params = np.array([0.11,0.5])
cost(init_params)
```

```
0.8722785388513962
```

Optimising the cost function using gradient descent

We use an instance of the function GradientDescentOptimizer to perform gradient descent. The hyperparameters of this optimization are step size and initial parameters. This stepsize denotes how big I must move after every iteration.

```
opt = qml.GradientDescentOptimizer(stepsize=0.4)  # initialise the optimizer
steps = 100  # maximum the number of steps taken
params = init_params # set the initial parameter values
# running the optimization
for i in range(steps):
    params = opt.step(cost,params) # update parameter after every iteration.
    if (i + 1) % 5 == 0:
        print("Cost after step {:5d}: {: .7f}".format(i + 1, cost(params)))

print("Optimized rotation angles: {}".format(params))
```

```
Cost after step     5: -0.4151640
Cost after step    10: -0.9926820
Cost after step    15: -0.9999554
Cost after step    20: -0.9999997
Cost after step    25: -1.0000000
```

```
Cost after step    30: -1.0000000
Cost after step    35: -1.0000000
Cost after step    40: -1.0000000
Cost after step    45: -1.0000000
Cost after step    50: -1.0000000
Cost after step    55: -1.0000000
Cost after step    60: -1.0000000
Cost after step    65: -1.0000000
Cost after step    70: -1.0000000
Cost after step    75: -1.0000000
Cost after step    80: -1.0000000
Cost after step    85: -1.0000000
Cost after step    90: -1.0000000
Cost after step    95: -1.0000000
Cost after step   100: -1.0000000
Optimized rotation angles: [8.88540711e-17 3.14159265e+00]
```

We see that in 10 iteration, the cost has been reduced to -0.99, which is rather quick.
We also see that the optimized rotation angles are $\theta_1 = 0$ and $\theta_2 = \pi$.
This makes sense because $\cos 0 \cos \pi = -1$, which is the minimum value.

References

- Amira Abbas. NIThePP Mini School Slides.
- Pennylane Tutorials. Basic Gradient Descent on Qubit Rotation.
- Maria Schuld et al. Circuit-centric quantum classifiers.,
- Schuld, Maria. Supervised learning with quantum computers. Springer, 2018