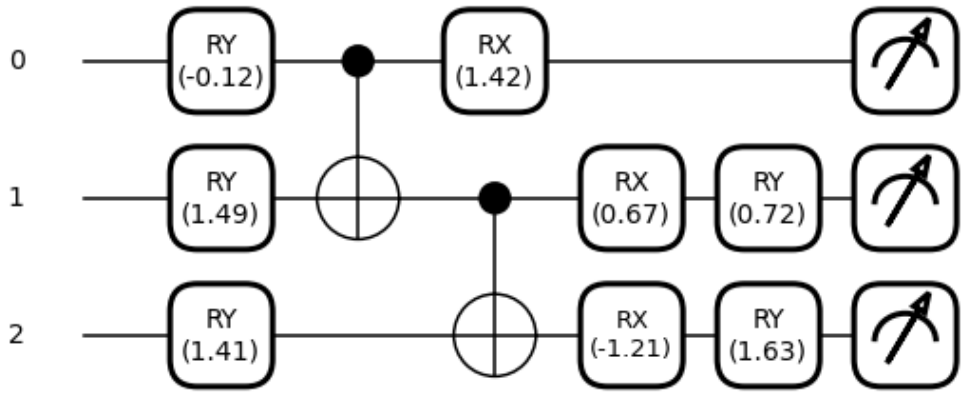
```
!pip install pennylane

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

dev_2qubits = qml.device("default.qubit", wires=2)
@qml.qnode(dev_2qubits)
def cost_fn_2qubits(params):
    # |psi_0>: state preparation
    qml.RY(np.pi / 4, wires=0)
    qml.RY(np.pi / 3, wires=1)
    # V0(theta0, theta1): Parametrized layer 0
    qml.RZ(params[0], wires=0)
    qml.RZ(params[1], wires=1)
    # W1: non-parametrized gates
    qml.CNOT(wires=[0, 1])
    # V_1(theta2, theta3): Parametrized layer 1
    qml.RY(params[2], wires=0)
    qml.RX(params[3], wires=1)
    # W2: non-parametrized gates
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliY(0))
params = np.array([0.432, -0.123, 0.543, 0.233])

dev_3qubits = qml.device("default.qubit", wires=3)
def circuit_3qubits(params, wires=0): # circuit for error, with
respect to b
    # |psi_0>: state preparation
    # V0(theta0, theta1): Parametrized layer 0
    qml.RY(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.RY(params[2], wires=2)
    # W1: non-parametrized gates
    qml.CNOT(wires=[0, 1])
    qml.CNOT(wires=[1, 2])
    # V_1(theta2, theta3): Parametrized layer 1
    qml.RX(params[3], wires=0)
    qml.RX(params[4], wires=1)
    qml.RX(params[5], wires=2)
    # W2: non-parametrized gates
    qml.RY(params[6], wires=1)
    qml.RY(params[7], wires=2)

coeffs_3qubits =[-0.124, 1.489, 1.409, 1.417, 0.671, -1.207, 0.717,
1.630 ];
init_params = np.array([0,0,0,0,0,0,0,0], requires_grad=True)
obs_3qubits = [qml.PauliZ(0),
qml.PauliZ(1),qml.PauliZ(2),qml.PauliZ(0),qml.PauliZ(1),
qml.PauliZ(2),qml.PauliX(0),qml.PauliX(1) ]
```

```
H_3qubits = qml.Hamiltonian(coeffs_3qubits, obs_3qubits)

@qml.qnode(dev_3qubits)
def cost_fn_3qubits(params):
    circuit_3qubits(params)
    return qml.expval(H_3qubits)


dev_3qubits = qml.device("default.qubit", wires=3)
@qml.qnode(dev_3qubits)
def circuit_3qubits(params, wires=3): # circuit for error, with
respect to b
    # |psi_0>: state preparation
    # V0(theta0, theta1): Parametrized layer 0
    qml.RY(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.RY(params[2], wires=2)
    # W1: non-parametrized gates
    qml.CNOT(wires=[0, 1])
    qml.CNOT(wires=[1, 2])
    # V_1(theta2, theta3): Parametrized layer 1
    qml.RX(params[3], wires=0)
    qml.RX(params[4], wires=1)
    qml.RX(params[5], wires=2)
    # W2: non-parametrized gates
    qml.RY(params[6], wires=1)
    qml.RY(params[7], wires=2)
    return qml.expval(H_3qubits)

params=[-0.124, 1.489, 1.409, 1.417, 0.671, -1.207, 0.717, 1.630 ];

fig, ax = qml.draw_mpl(circuit_3qubits, decimals=2)(params)
plt.show()
```

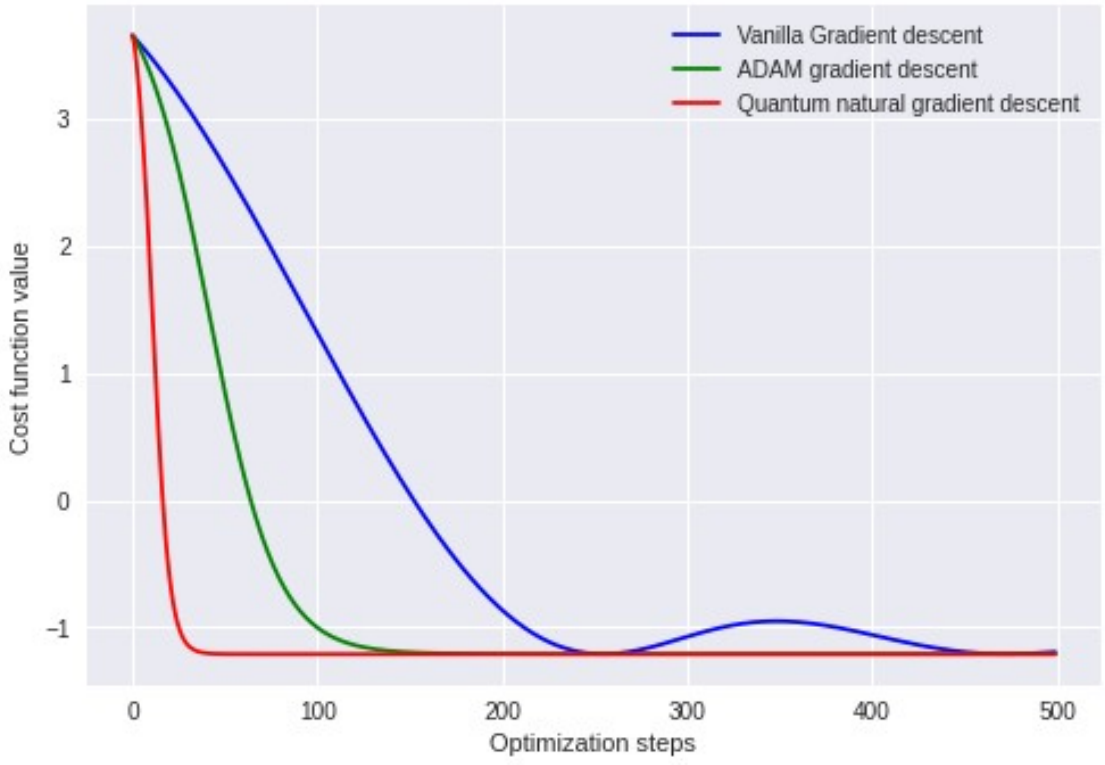
```
/usr/lib/python3.8/_collections_abc.py:832:
MatplotlibDeprecationWarning:
The datapath rcparam was deprecated in Matplotlib 3.2.1 and will be
removed two minor releases later.
  self[key] = other[key]
/usr/lib/python3.8/_collections_abc.py:832:
MatplotlibDeprecationWarning:
The savefig.frameon rcparam was deprecated in Matplotlib 3.1 and will
be removed in 3.3.
  self[key] = other[key]
/usr/lib/python3.8/_collections_abc.py:832:
MatplotlibDeprecationWarning:
The text.latex.unicode rcparam was deprecated in Matplotlib 3.0 and
will be removed in 3.2.
  self[key] = other[key]
/usr/lib/python3.8/_collections_abc.py:832:
```

```
MatplotlibDeprecationWarning:
The verbose.fileo rcparam was deprecated in Matplotlib 3.1 and will be
removed in 3.3.
  self[key] = other[key]
/usr/lib/python3.8/_collections_abc.py:832:
MatplotlibDeprecationWarning:
The verbose.level rcparam was deprecated in Matplotlib 3.1 and will be
removed in 3.3.
  self[key] = other[key]
```




```
def convergence(opt, cost_fn, init_params,  max_iterations,
step_size):
  params = init_params
  param_history = [params]
  cost_history = []
  if(opt == "VGD"):
    opt = qml.GradientDescentOptimizer(stepsize=step_size)
  elif(opt == "QNG"):
    opt = qml.QNGOptimizer(stepsize=step_size, approx="block-diag")
  elif(opt == "ADAM"):
    opt = qml.AdamOptimizer(stepsize=step_size, beta1=0.99,
beta2=0.99, eps=1e-08)

  for n in range(max_iterations):
      params, prev_energy = opt.step_and_cost(cost_fn, params)
      param_history.append(params)
      cost_history.append(prev_energy)
      energy = cost_fn(params)
  return cost_history, param_history
```

```
max_iterations = 500
step_size = 0.01
'''
QNGcost, QNGparams = convergence("QNG", cost_fn_2qubits, init_params,
max_iterations, step_size)
VGDcost, VGDparams = convergence("VGD", cost_fn_2qubits, init_params,
max_iterations, step_size)
ADAMcost, ADAMparams = convergence("ADAM", cost_fn_2qubits,
init_params,  max_iterations, step_size)
'''
QNGcost, QNGparams = convergence("QNG", cost_fn_3qubits, init_params,
max_iterations, step_size)
VGDcost, VGDparams = convergence("VGD", cost_fn_3qubits, init_params,
max_iterations, step_size)
ADAMcost, ADAMparams = convergence("ADAM", cost_fn_3qubits,
init_params,  max_iterations, step_size)

plt.style.use("seaborn")
plt.plot(VGDcost, "b", label="Vanilla Gradient descent")
plt.plot(ADAMcost, "g", label="ADAM gradient descent")
plt.plot(QNGcost, "r", label="Quantum natural gradient descent")

plt.ylabel("Cost function value")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



PART 2 : Binary Classification

```
from itertools import chain
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.preprocessing import minmax_scale
from sklearn.model_selection import train_test_split
import sklearn.metrics as metrics

import pennylane as qml
from pennylane import numpy as np
from pennylane.templates.embeddings import AngleEmbedding
from pennylane.templates.layers import StronglyEntanglingLayers
from pennylane.optimize import GradientDescentOptimizer
import seaborn
# load the dataset
iris = datasets.load_iris()
# shuffle the data
X, y = shuffle(iris.data, iris.target, random_state=0)
# select only 2 first classes from the data
X = X[y<=1]
y = y[y<=1]
# normalize data
X = minmax_scale(X, feature_range=(0, np.pi))
# split data into train+validation and test
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
test_size=0.2)
# split into train and validation
X_train, X_validation, y_train, y_validation =
train_test_split(X_train_val, y_train_val, test_size=0.20)
print(np.shape(X_train), y_train)
```

```
(64, 4) [0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1
1 0 0 1 1 1
 1 1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0]
```

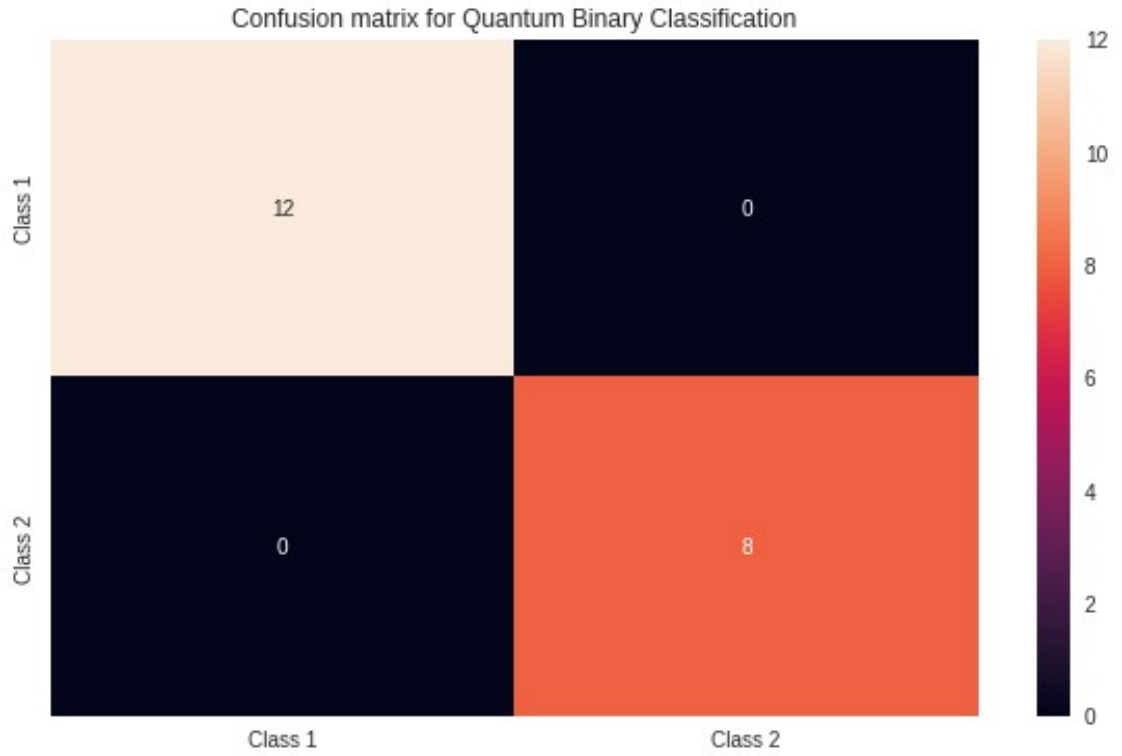
Quantum Binary classification

```
#--------- Quantum Part
# number of qubits is equal to the number of features
n_qubits = X.shape[1]
# quantum device handle
dev = qml.device("default.qubit", wires=n_qubits)
# quantum circuit
@qml.qnode(dev)
def circuit(weights, x=None):
    AngleEmbedding(x, wires = range(n_qubits))
    StronglyEntanglingLayers(weights, wires = range(n_qubits))
    return qml.expval(qml.PauliZ(0))
# variational quantum classifier
def variational_classifier(theta, x=None):
```

```
    weights = theta[0]
    bias = theta[1]
    return circuit(weights, x=x) + bias
# cost function
def cost(theta, X, expectations):
    e_predicted = \
        np.array([variational_classifier(theta, x=x) for x in X])
    loss = np.mean((e_predicted - expectations)**2)
    return loss
# number of quantum layers
n_layers = 3
n_wires = n_qubits
# convert classes to expectations: 0 to -1, 1 to +1
e_train = np.empty_like(y_train)
e_train[y_train == 0] = -1
e_train[y_train == 1] = +1
# select learning batch size
batch_size = 5
# calculate numbe of batches
batches = len(X_train) // batch_size
# select number of epochs
n_epochs = 5
# draw random quantum node weights
param_shape = StronglyEntanglingLayers.shape(n_layers=n_layers,
n_wires=n_wires)
init_params = np.random.uniform(low=0, high=2*np.pi, size=param_shape,
requires_grad=True)
theta_weights = init_params
theta_bias = 0.0
theta_init = (theta_weights, theta_bias) # initial weights
# train the variational classifier
theta = theta_init
pennylane_opt = GradientDescentOptimizer() # build the optimizer
object

# split training data into batches
X_batches = np.array_split(np.arange(len(X_train)), batches)
cost_q = []
for it, batch_index in enumerate(chain(*(n_epochs * [X_batches]))):
    # Update the weights by one optimizer step
    batch_cost = \
        lambda theta: cost(theta, X_train[batch_index],
e_train[batch_index])
    theta = pennylane_opt.step(batch_cost, theta)
    cost_q.append(cost(theta, X_train[batch_index],
e_train[batch_index]))
    # use X_validation and y_validation to decide whether to stop
# end of learning loop

# convert expectations to classes
```

```
expectations = np.array([variational_classifier(theta, x=x) for x in
X_test])
prob_class_one = (expectations + 1.0) / 2.0
y_pred = (prob_class_one >= 0.5)

print(metrics.accuracy_score(y_test, y_pred))

cf_matrix = metrics.confusion_matrix(y_test, y_pred)
index = ["Class 1", "Class 2"]
columns = ["Class 1", "Class 2"]
import pandas as pd
cm_df = pd.DataFrame(cf_matrix ,columns,index)
plt.figure(figsize=(10,6))
plt.title("Confusion matrix for Quantum Binary Classification")
seaborn.heatmap(cm_df, annot=True)
```

```
/usr/local/lib/python3.7/dist-packages/pennylane/_grad.py:108:
UserWarning: Attempted to differentiate a function with no trainable
parameters. If this is unintended, please add trainable parameters via
the 'requires_grad' attribute or 'argnum' keyword.
  "Attempted to differentiate a function with no trainable parameters.
"

1.0

<matplotlib.axes._subplots.AxesSubplot at 0x7fd639419dd0>
```

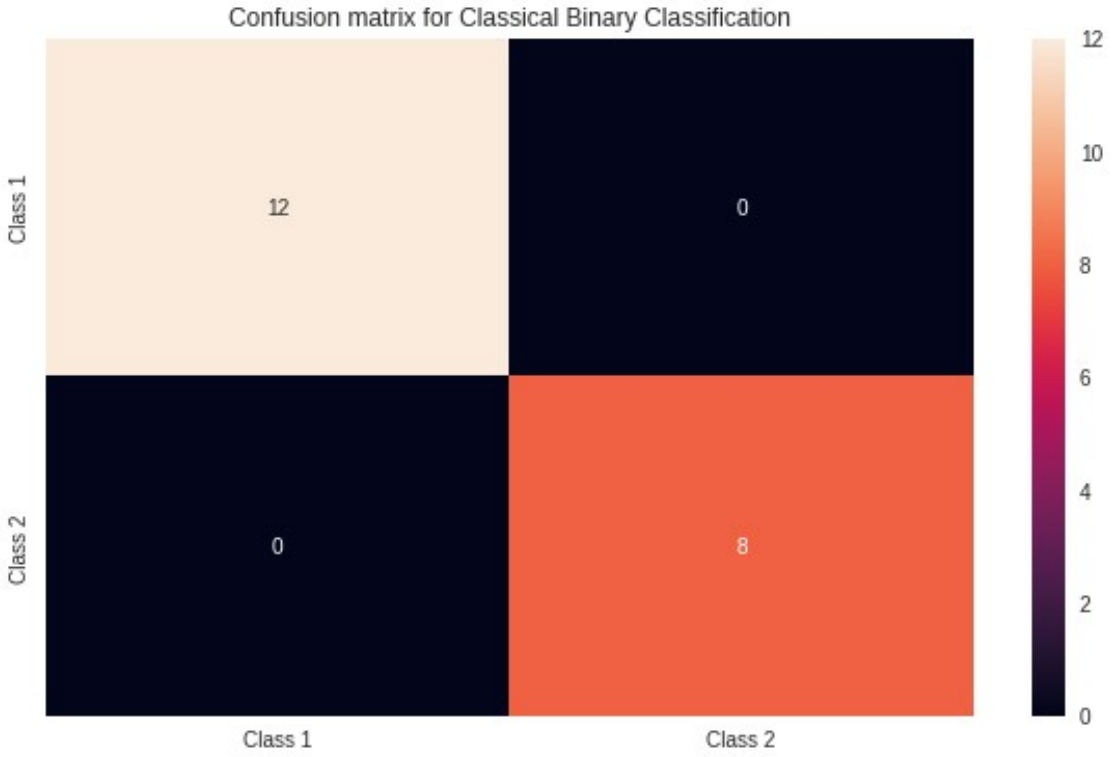


Classical binary classification

```
class NeuralNetwork():

  def __init__(self):
    np.random.seed(1)
    self.synaptic_weights = 2*np.random.random((4,1)) - 1

  def sigmoid(self, x):# The Sigmoid function, an S shaped curve,
normalizes the weighted sum of the inputs between 0 and 1.
    return 1 /(1+np.exp(-x))

  def sigmoid_derivative(self,x):  # The derivative of the Sigmoid
function.# The gradient of the Sigmoid curve
    return x*(1-x)

    # The training phase adjusts the weights each time to reduce the
error
  def train(self, training_inputs, training_outputs,
training_iterations):
    cost = []
    for iteration in range(training_iterations):
      output= self.think(training_inputs)
      # Calculate the error
      error = training_outputs - output
      cost.append(np.mean(error**2))
      # Adjustments refers to the backpropagation process
      adjustments = np.dot(training_inputs.T,
error*self.sigmoid_derivative(output))
      # Adjust the weights.
      self.synaptic_weights += adjustments
    return cost

  def think(self, inputs):
    # Pass inputs through our neural network (our single neuron).
    inputs = inputs.astype(float)
    output = self.sigmoid(np.dot(inputs, self.synaptic_weights))
    return output


neural_network = NeuralNetwork()

# The training set.each consisting of 4 input values and 1 output
value.
training_inputs = X_train
training_outputs = np.reshape(y_train, (len(y_train), 1))
cost = neural_network.train(training_inputs, training_outputs, 5)
expectations = neural_network.think(X_test)
```

```python
#If the score is higher than 0.5 then it's a 1 otherwise a 0
def getResult(score):
    if score < 0.5:
        return 0
    elif score >= 0.5:
        return 1

#Apply function on predicted dataframe
y_pred = []
for i in range(len(expectations)):
  y_pred.append( getResult(expectations[i]) )

#Evaluate model performance
print(metrics.accuracy_score(y_test, y_pred))

cf_matrix = metrics.confusion_matrix(y_test, y_pred)
index = ["Class 1", "Class 2"]
columns = ["Class 1", "Class 2"]
import pandas as pd
cm_df = pd.DataFrame(cf_matrix ,columns,index)
plt.figure(figsize=(10,6))
plt.title("Confusion matrix for Classical Binary Classification")
seaborn.heatmap(cm_df, annot=True)
```

```
1.0

<matplotlib.axes._subplots.AxesSubplot at 0x7fd63c869e90>
```




```
plt.plot(cost)
```

```
[<matplotlib.lines.Line2D at 0x7fd6391b4750>]
```