Task4

March 25, 2025

1 Task 4: Quantum Generative Adversarial Network

We will be implementing quantum classifier using Cirq.

Since the data is in npz file and the cirq library supports python version 3.9 so, first we will save the file using a lower protocol by using python version 3.12 then we can use our standard 3.9 version which we are using for all other task.

[2]:
```
import numpy as np
import io

#Load the original container file
container = np.load('task4data.npz', allow_pickle=True)

#Extract the inner files
train_data = container['training_input.npy']
test_data  = container['test_input.npy']

#check if they are OD array it means it is a dictionary
if isinstance(train_data, np.ndarray) and train_data.ndim == 0:
    train_data = train_data.item()
if isinstance(test_data, np.ndarray) and test_data.ndim == 0:
    test_data = test_data.item()

X_train = train_data['0']
y_train = train_data['1']
X_test  = test_data['0']
y_test  = test_data['1']
# just check the shape of data
print("Original training data shapes:", X_train.shape, y_train.shape)
print("Original testing data shapes:", X_test.shape, y_test.shape)


# Save the files separatedly with protocol that support 3.9 version of python
np.save('training_input_converted.npy', train_data, allow_pickle=True)
np.save('test_input_converted.npy', test_data, allow_pickle=True)
```

```
# create the new container file as the dataset was before
converted_train = np.load('training_input_converted.npy', allow_pickle=True)
converted_test  = np.load('test_input_converted.npy', allow_pickle=True)

np.savez('task4data_converted.
 ↪npz',training_input=converted_train,test_input=converted_test)

print("Conversion complete.")
```

```
Original training data shapes: (50, 5) (50, 5)
Original testing data shapes: (50, 5) (50, 5)
Conversion complete.
```

Now we can switch to our original environment and of version 3.9

Import the required libraries

[3]:
```
import numpy as np
import cirq
import sympy
import copy
import math
from sklearn.metrics import roc_auc_score, accuracy_score
```

Load the converted Dataset

[4]:
```
container = np.load('task4data_converted.npz', allow_pickle=True)
train_data = container['training_input']
test_data  = container['test_input']

if isinstance(train_data, np.ndarray) and train_data.ndim == 0:
    train_data = train_data.item()
if isinstance(test_data, np.ndarray) and test_data.ndim == 0:
    test_data = test_data.item()

# Data keys: '0' for features and '1' for labels.
X_train = train_data['0']
y_train = train_data['1']
X_test  = test_data['0']
y_test  = test_data['1']

print("Training data shapes:", X_train.shape, y_train.shape)
print("Testing data shapes:", X_test.shape, y_test.shape)
```

```
Training data shapes: (50, 5) (50, 5)
Testing data shapes: (50, 5) (50, 5)
```

Normalize the data and convert label to scalars

[5]:
```
X_train = (X_train - np.mean(X_train, axis=0)) / np.std(X_train, axis=0)
X_test  = (X_test - np.mean(X_test, axis=0)) / np.std(X_test, axis=0)

y_train_scalar = np.array([int(y[0]) if isinstance(y, np.ndarray) and len(y) >␣
 ↪1 else int(y) for y in y_train])
y_test_scalar  = np.array([int(y[0]) if isinstance(y, np.ndarray) and len(y) >␣
 ↪1 else int(y) for y in y_test])
```

1.1 Quantum Encoding

We will be encoding each sample point using Rx rotation and then apply parameterized rotation

[7]:
```
def create_classifier_circuit(features, params):
    num_features = len(features)
    qubits = cirq.GridQubit.rect(1, num_features)
    circuit = cirq.Circuit()

    for i, feature in enumerate(features):
        circuit.append(cirq.rx(float(feature))(qubits[i]))

    for i in range(num_features):
        circuit.append(cirq.rx(float(params[i]))(qubits[i]))

    return circuit, qubits

simulator = cirq.Simulator()

def circuit_expectation(features, params):
    circuit, qubits = create_classifier_circuit(features, params)
    observable = cirq.Z(qubits[0])
    result = simulator.simulate_expectation_values(circuit,␣
 ↪observables=[observable])
    return result[0].real

# map probability in 0 to 1 range
def predict_probability(features, params):
    exp_val = circuit_expectation(features, params)
    return (1 + exp_val) / 2
```

Use binary cross entropy and compute gradient using parameter shift rule

1.1.1 Mathematical Formulation of Gradient Computation

1. Binary Cross-Entropy Loss

$$\mathcal{L} = -y \log(p) - (1 - y) \log(1 - p)$$

where (p) is the predicted probability.

2. Probability Computation from Expectation Value

$$p = \frac{1 + \langle Z \rangle}{2}$$

3. Gradient of Loss w.r.t. (p)

$$\frac{\partial \mathcal{L}}{\partial p} = -\frac{y}{p} + \frac{(1-y)}{(1-p)}$$

4. Parameter-Shift Rule for Quantum Gradients Using the **parameter-shift rule**, the gradient of the expectation value is computed as:

$$\frac{\partial \langle Z \rangle}{\partial \theta_j} = \frac{\langle Z \rangle(\theta_j + \frac{\pi}{2}) - \langle Z \rangle(\theta_j - \frac{\pi}{2})}{2}$$

5. Applying the Chain Rule

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \frac{\partial \mathcal{L}}{\partial p} \times \frac{1}{2} \times \frac{\partial \langle Z \rangle}{\partial \theta_j}$$

6. Final Gradient Update

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \left(-\frac{y}{p} + \frac{(1-y)}{(1-p)} \right) \times \frac{1}{2} \times \frac{\langle Z \rangle(\theta_j + \frac{\pi}{2}) - \langle Z \rangle(\theta_j - \frac{\pi}{2})}{2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \left(-\frac{y}{p} + \frac{(1-y)}{(1-p)} \right) \times \frac{\langle Z \rangle(\theta_j + \frac{\pi}{2}) - \langle Z \rangle(\theta_j - \frac{\pi}{2})}{4}$$

[8]:
```python
def binary_cross_entropy_loss(y_true, p):
    eps = 1e-8
    return - (y_true * np.log(p + eps) + (1 - y_true) * np.log(1 - p + eps))

def loss_for_sample(features, y_true, params):
    p = predict_probability(features, params)
    return binary_cross_entropy_loss(y_true, p)

def compute_gradients(features, y_true, params):
    grad = np.zeros_like(params)
    p = predict_probability(features, params)
    dL_dp = - y_true / p + (1 - y_true) / (1 - p)

    shift = math.pi / 2
    for j in range(len(params)):
        params_plus = copy.deepcopy(params)
        params_minus = copy.deepcopy(params)
        params_plus[j] += shift
        params_minus[j] -= shift
```

```
        exp_plus = circuit_expectation(features, params_plus)
        exp_minus = circuit_expectation(features, params_minus)
        d_exp = (exp_plus - exp_minus) / 2  # parameter-shift derivative

        grad[j] = dL_dp * d_exp / 4  # combined chain rule factor
    return grad
```

1.2 Train

[ ]:
```
num_params = X_train.shape[1]  # One parameter per feature (qubit)
params = np.random.randn(num_params)  # Initialize parameters
learning_rate = 0.1
epochs = 20

print("\nTraining quantum classifier...")
for epoch in range(epochs):
    total_loss = 0.0
    total_grad = np.zeros_like(params)

    # Loop over each training sample.
    for i in range(len(X_train)):
        x_sample = X_train[i]
        y_sample = y_train_scalar[i]
        loss_val = loss_for_sample(x_sample, y_sample, params)
        grad_val = compute_gradients(x_sample, y_sample, params)
        total_loss += loss_val
        total_grad += grad_val

    avg_loss = total_loss / len(X_train)
    avg_grad = total_grad / len(X_train)

    # Update parameters using gradient descent.
    params -= learning_rate * avg_grad

    # training accuracy.
    train_preds = [1 if predict_probability(x, params) >= 0.5 else 0 for x in␣
 ↪X_train]
    train_acc = np.mean(train_preds == y_train_scalar)

    # test metrics.
    test_probs = np.array([predict_probability(x, params) for x in X_test])
    test_preds = (test_probs >= 0.5).astype(int)
    test_acc = accuracy_score(y_test_scalar, test_preds)

    print(f"Epoch {epoch+1}/{epochs} - Loss: {avg_loss:.4f} | Train Acc:␣
 ↪{train_acc:.4f} | Test Acc: {test_acc:.4f}")
```

```python
test_probs = np.array([predict_probability(x, params) for x in X_test])
test_preds = (test_probs >= 0.5).astype(int)
test_acc = accuracy_score(y_test_scalar, test_preds)

print("\nFinal Evaluation:")
print("Test Accuracy:", test_acc)
```

```
Training quantum classifier…
Epoch 1/20 - Loss: 4.9188 | Train Acc: 0.7800 | Test Acc: 0.4400
Epoch 2/20 - Loss: 0.9971 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 3/20 - Loss: 0.9380 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 4/20 - Loss: 0.8976 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 5/20 - Loss: 0.8683 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 6/20 - Loss: 0.8463 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 7/20 - Loss: 0.8297 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 8/20 - Loss: 0.8171 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 9/20 - Loss: 0.8075 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 10/20 - Loss: 0.8003 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 11/20 - Loss: 0.7949 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 12/20 - Loss: 0.7909 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 13/20 - Loss: 0.7880 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 14/20 - Loss: 0.7859 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 15/20 - Loss: 0.7844 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 16/20 - Loss: 0.7833 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 17/20 - Loss: 0.7826 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 18/20 - Loss: 0.7821 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 19/20 - Loss: 0.7817 | Train Acc: 0.7800 | Test Acc: 0.8200
Epoch 20/20 - Loss: 0.7815 | Train Acc: 0.7800 | Test Acc: 0.8200

Final Evaluation:
Test Accuracy: 0.82
```

The final accuracy is 82%

As we are initializing the parameters randomly the accuracy changes each time we run the code