Quantum Generative Adversarial Network with Noise

Project Name: Quantum Generative Adversarial Network with Noise

Project member:

Yiming Huang
Wanghao Ren
Zhili Chen

Dodument Type: Report

Project Start Time: 11/01/2017

Sourcecode Version: 0.0.1

Keywords: Variational Quantum Circuit, Machine Learning

Modify April 12, 2020

Submitted by:

XXX

Contents

1 Experiment **2**

2 Results **2**

3 Next Plan **2**

4 Appendix **2**

A Source Code 2

1 Experiment

The quantum countermeasure circuit used to approximate the pure state is an application in the context of noisy intermediate-scale quantum computer(NISQ). This week the basic structure of the circuit was reproduced.

I set the specified target state, and set the generator depth and the number of evaluator layers to 2 (this configuration is the optimal configuration mentioned in the original paper). Then I observed the relationship between the approximation degree and the iterations number by recording fidelity.

2 Results

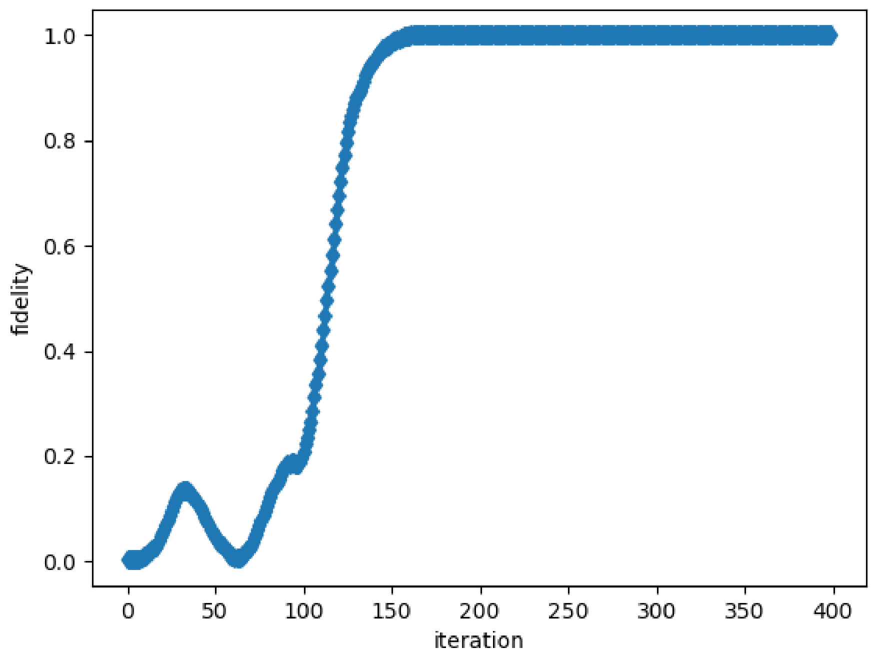

Figure 1: fidelity

```
[[ 1.00000000e+00-1.24709959e-05j]
 [ 1.30539553e-06-1.06168340e-11j]
 [-2.06771156e-06+1.37115189e-11j]
 [ 1.33851883e-06-5.46255329e-12j]
 [ 1.38835383e-06-1.88375846e-11j]
 [-2.60572807e-06+3.59907555e-11j]
 [-3.99434728e-07+3.41639907e-12j]
 [ 6.72481171e-12-4.30133528e-17j]]
```

Figure 2: generate state

After 150 iterations, the fidelity is close to 1. And converge to 1 in the following iterations. The generated state is similar to the set target state.
However, it has been found in many experiments that the fidelity tends to increase first and then decrease to 0 during the initial iteration.

3 Next Plan

1. Complete random generation of target states, conduct experiments and observe trends.
2. Introduce noise into the system, simulate the operation of this circuit in the context of NISQ and observe the effect of noise on the results.
3. Read the gradient descent literature and think about the problem of fidelity falling to zero in the early iterations.

4 Appendix

A Source Code

Listing 1: simulate

```
from Quantum_Circuit import AdversarialQCircuit as QC
import numpy as np
from matplotlib import pyplot as plt

dec = 0.5
inc = 1.2
MaxStep = 6e-3 * np.pi
MinStep = 1e-6 * np.pi
InitStep = 1.5e-3 * np.pi


def gradient_descent(der1, der2, step, para):
    for i in range(0,len(der1)):
        if der1[i] * der2[i] > 0:
            step[i] = min(inc*step[i], MaxStep)
        elif der1[i] * der2[i] < 0:
            step[i] = max(dec*step[i], MinStep)
            der2[i] = 0
        para[i] = para[i] - np.sign(der2[i]) * step[i]


def gradient_ascent(der1, der2, step, para):
    for i in range(0,len(der1)):
        if der1[i] * der2[i] > 0:
            step[i] = min(inc*step[i], MaxStep)
        elif der1[i] * der2[i] < 0:
            step[i] = max(dec*step[i], MinStep)
            der2[i] = 0
        para[i] = para[i] + np.sign(der2[i]) * step[i]


```

```python
def fidelity(qcircuit):
    qcircuit.generate_state()
    gen_state = qcircuit.main_register.state
    qcircuit.register_initial()
    qcircuit.main_register.simulate(qcircuit.target)
    tar_state = qcircuit.main_register.state
    qcircuit.register_initial()
    a = np.dot(gen_state.T.conj(), tar_state)
    f = a * a.conj()
    return f[0][0].real


def irprop(qcircuit):
    y = []
    gen_para = qcircuit.get_generator_para()
    gen_der1 = [0] * len(gen_para)
    gen_der2 = qcircuit.get_generator_der_list()
    gen_step = [InitStep] * len(gen_para)
    dis_para = qcircuit.get_discriminator_para()
    dis_der1 = [0] * len(dis_para)
    dis_der2 = [0] * len(dis_para)
    dis_step = [InitStep] * len(dis_para)
    for i in range(0,400):
        gradient_descent(gen_der1, gen_der2, gen_step, gen_para)
        qcircuit.change_generator_para(gen_para)
        dis_der2 = qcircuit.get_discriminator_der_list()
        gradient_ascent(dis_der1, dis_der2, dis_step, dis_para)
        qcircuit.change_discriminator_para(dis_para)
        gen_der1 = gen_der2[:]
        gen_der2 = qcircuit.get_generator_para()
        dis_der1 = dis_der2[:]
        y.append(fidelity(qcircuit))
    x = np.arange(0, 400, 1)
    plt.xlabel("iteration")
    plt.ylabel("fidelity")
    plt.plot(x, y, 'd')
    plt.savefig('./test1.png')
    plt.show()


def simulate():
    list = [[1, 0], [1, 0], [1, 0]]
    cir = QC.QCircuitSimulator(2, 2, 3, list)
    irprop(cir)
    cir.generate_state()
    print(cir.main_register.state)

simulate()
```

Listing 2: circuit

```python
import Quantum_Circuit.QGate as QG
import Quantum_Circuit.QRegister as QR
import numpy as np
```

```


class QCircuitSimulator:
    def __init__(self, GDepth, DDepth, n, pslist):
        self.qnum = n + 1
        self.d = 2 ** (n + 1)
        self.register = QR.QRegister(n + 1)
        self.main_register = QR.QRegister(n)
        self.anc_register = QR.QRegister(1)
        self.E0 = np.array([[1.0]])
        self.target = pslist
        self.ggate_list = []
        self.dgate_list = []
        for i in range(1, n+2):
            if i != n+1:
                self.E0 = np.kron(self.E0, QG.I)
            else:
                self.E0 = np.kron(self.E0, QG.P0)
        counter = 1
        while counter <= GDepth:
            self.ggate_list.extend(QG.Layer(n).gate_list)
            counter += 1
        counter = 1
        while counter <= DDepth:
            self.dgate_list.extend(QG.Layer(n+1).gate_list)
            counter += 1

    def register_initial(self):
        self.register.initial()
        self.main_register.initial()
        self.anc_register.initial()

    def generate_state(self):
        gcircuit = np.eye(2 ** (self.qnum-1))
        for i in self.ggate_list:
            if isinstance(i, QG.CNOT):
                gcircuit = np.dot(i.Cx, gcircuit)
            else:
                gcircuit = np.dot(i.PGate, gcircuit)
        self.main_register.state = np.dot(gcircuit, self.main_register.state)
        self.register.state = np.kron(self.main_register.state, self.anc_register.state)

    def discriminate_state(self):
        dcircuit = np.eye(self.d)
        for i in self.dgate_list:
            if isinstance(i, QG.CNOT):
                dcircuit = np.dot(i.Cx, dcircuit)
            else:
                dcircuit = np.dot(i.PGate, dcircuit)
        self.register.state = np.dot(dcircuit, self.register.state)

    def generator_derivative(self, i):
```

```
        p = self.ggate_list[i].para
        self.ggate_list[i].change_para(np.pi/2+p)
        self.generate_state()
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0,100):
            if(np.random.rand() < ancprob0):
                count += 1
        part1 = count / 100
        self.register_initial()
        self.ggate_list[i].change_para(np.pi / 2 - p)
        self.generate_state()
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0, 100):
            if (np.random.rand() < ancprob0):
                count += 1
        part2 = count / 100
        self.register_initial()
        self.ggate_list[i].change_para(p)
        der = -0.5 * 0.5 * (part1 - part2)
        return der

    def discriminator_derivative(self, i):
        p = self.dgate_list[i].para
        self.dgate_list[i].change_para(np.pi / 2 + p)
        self.generate_state()
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0, 100):
            if (np.random.rand() < ancprob0):
                count += 1
        part1 = count / 100
        self.register_initial()
        self.dgate_list[i].change_para(np.pi / 2 - p)
        self.generate_state()
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0, 100):
            if (np.random.rand() < ancprob0):
                count += 1
        part2 = count / 100
        self.register_initial()
        der1 = -0.5 * 0.5 * (part1 - part2)
        self.dgate_list[i].change_para(np.pi / 2 + p)
```

```
        self.main_register.simulate(self.target)
        self.register.state = np.kron(self.main_register.state, self.anc_register.state)
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0, 100):
            if (np.random.rand() < ancprob0):
                count += 1
        part1 = count / 100
        self.register_initial()
        self.dgate_list[i].change_para(np.pi / 2 - p)
        self.main_register.simulate(self.target)
        self.register.state = np.kron(self.main_register.state, self.anc_register.state)
        self.discriminate_state()
        ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
        ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
        count = 0
        for j in range(0, 100):
            if (np.random.rand() < ancprob0):
                count += 1
        part2 = count / 100
        self.register_initial()
        self.dgate_list[i].change_para(p)
        der2 = 0.5 * 0.5 * (part1 - part2)
        der = der1 + der2
        return der

    def get_discriminator_para(self):
        list = []
        for i in self.dgate_list:
            if isinstance(i, QG.ParameterizedGate):
                list.append(i.para)
        return list

    def get_generator_para(self):
        list = []
        for i in self.ggate_list:
            if isinstance(i, QG.ParameterizedGate):
                list.append(i.para)
        return list

    def get_discriminator_der_list(self):
        list = []
        for i in range(0, len(self.dgate_list)):
            if isinstance(self.dgate_list[i], QG.ParameterizedGate):
                list.append(self.discriminator_derivative(i))
        return list

    def get_generator_der_list(self):
        list = []
```

```
        for i in range(0, len(self.ggate_list)):
            if isinstance(self.ggate_list[i], QG.ParameterizedGate):
                list.append(self.generator_derivative(i))
        return list

    def change_generator_para(self, paralist):
        j = 0
        for i in self.ggate_list:
            if isinstance(i, QG.ParameterizedGate):
                i.change_para(paralist[j])
                j += 1

    def change_discriminator_para(self, paralist):
        j = 0
        for i in self.dgate_list:
            if isinstance(i, QG.ParameterizedGate):
                i.change_para(paralist[j])
                j += 1
```

Listing 3: gate

```
import numpy as np
from scipy import linalg as lin

Zero = np.array([[1.0],[0.0]])
One = np.array([[0.0],[1.0]])
X = np.array([[0,1],[1,0]])
Y = np.array([[0,-1j],[1j,0]])
Z = np.array([[1,0],[0,-1]])
I = np.eye(2)
P0 = np.dot(Zero, Zero.T)
P1 = np.dot(One, One.T)


class CNOT:
    def __init__(self, i, j, n):
        counter = 1
        self.control_bit = i
        self.act_bit = j
        Cx1 = np.array([[1.0]])
        Cx2 = np.array([[1.0]])
        while counter <= n:
            if counter == i:
                Cx1 = np.kron(Cx1, P0)
                Cx2 = np.kron(Cx2, P1)
            elif counter == j:
                Cx1 = np.kron(Cx1, I)
                Cx2 = np.kron(Cx2, X)
            else:
                Cx1 = np.kron(Cx1, I)
                Cx2 = np.kron(Cx2, I)
            counter += 1
        self.Cx = Cx1 + Cx2

```

```

class ParameterizedGate:
    def __init__(self, i, g, n):
        self.para = np.random.uniform(-1, 1) * np.pi
        self.act_bit = i
        self.Gate = np.array([[1.0]])
        counter = 1
        while counter <= n:
            if counter == i:
                self.Gate = np.kron(self.Gate, g)
            else:
                self.Gate = np.kron(self.Gate, I)
            counter += 1
        self.PGate = lin.expm(-0.5j * self.para * self.Gate)

    def change_para(self, p):
        self.para = p
        self.PGate = lin.expm(-0.5j * self.para * self.Gate)


class TwoQbitGate:
    def __init__(self, i, j, n):
        self.tq_gate_list = []
        self.tq_gate_list.append(ParameterizedGate(i, Z, n))
        self.tq_gate_list.append(ParameterizedGate(i, Y, n))
        self.tq_gate_list.append(ParameterizedGate(i, Z, n))
        self.tq_gate_list.append(ParameterizedGate(j, Z, n))
        self.tq_gate_list.append(ParameterizedGate(j, Y, n))
        self.tq_gate_list.append(ParameterizedGate(j, Z, n))
        self.tq_gate_list.append(CNOT(j, i, n))
        self.tq_gate_list.append(ParameterizedGate(i, Z, n))
        self.tq_gate_list.append(ParameterizedGate(j, Y, n))
        self.tq_gate_list.append(CNOT(i, j, n))
        self.tq_gate_list.append(ParameterizedGate(j, Y, n))
        self.tq_gate_list.append(CNOT(j, i, n))
        self.tq_gate_list.append(ParameterizedGate(i, Z, n))
        self.tq_gate_list.append(ParameterizedGate(i, Y, n))
        self.tq_gate_list.append(ParameterizedGate(i, Z, n))
        self.tq_gate_list.append(ParameterizedGate(j, Z, n))
        self.tq_gate_list.append(ParameterizedGate(j, Y, n))
        self.tq_gate_list.append(ParameterizedGate(j, Z, n))


class Layer:
    def __init__(self, n):
        counter = 1
        self.gate_list = []
        while counter < n:
            self.gate_list.extend(TwoQbitGate(counter, counter+1, n).tq_gate_list)
            counter += 2
        counter = 2
        while counter < n:
            self.gate_list.extend(TwoQbitGate(counter, counter+1, n).tq_gate_list)
```

```
            counter += 2
```

Listing 4: register

```
import numpy as np


Zero = np.array([[1.0],[0.0]])
One = np.array([[0.0],[1.0]])


class QRegister:
    def __init__(self, n):
        self.length = n
        counter = 1
        self.state = np.array([[1.0]])
        while counter <= n:
            self.state = np.kron(self.state, Zero)
            counter += 1

    def initial(self):
        counter = 1
        self.state = np.array([[1.0]])
        while counter <= self.length:
            self.state = np.kron(self.state, Zero)
            counter += 1

    def simulate(self, pslist):
        counter = 0
        self.state = np.array([[1.0]])
        while counter < self.length:
            singlebitstate = pslist[counter][0] * Zero + pslist[counter][1] * One
            self.state = np.kron(self.state, singlebitstate)
            counter += 1
```