

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

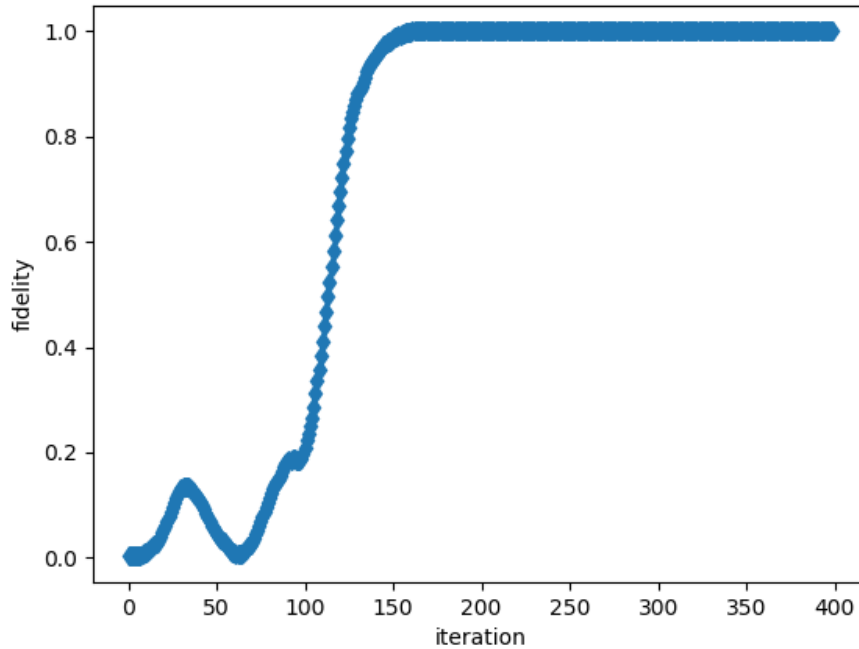


Figure 1: fidelity

```
[[ 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

```

1 from Quantum_Circuit import AdversarialQCircuit as QC
2 import numpy as np
3 from matplotlib import pyplot as plt
4
5 dec = 0.5
6 inc = 1.2
7 MaxStep = 6e-3 * np.pi
8 MinStep = 1e-6 * np.pi
9 InitStep = 1.5e-3 * np.pi
10
11
12 def gradient_descent(der1, der2, step, para):
13     for i in range(0, len(der1)):
14         if der1[i] * der2[i] > 0:
15             step[i] = min(inc*step[i], MaxStep)
16         elif der1[i] * der2[i] < 0:
17             step[i] = max(dec*step[i], MinStep)
18             der2[i] = 0
19         para[i] = para[i] - np.sign(der2[i]) * step[i]
20
21
22 def gradient_ascent(der1, der2, step, para):
23     for i in range(0, len(der1)):
24         if der1[i] * der2[i] > 0:
25             step[i] = min(inc*step[i], MaxStep)
26         elif der1[i] * der2[i] < 0:
27             step[i] = max(dec*step[i], MinStep)
28             der2[i] = 0
29         para[i] = para[i] + np.sign(der2[i]) * step[i]
30
31

```

```

32 def fidelity(qcircuit):
33     qcircuit.generate_state()
34     gen_state = qcircuit.main_register.state
35     qcircuit.register_initial()
36     qcircuit.main_register.simulate(qcircuit.target)
37     tar_state = qcircuit.main_register.state
38     qcircuit.register_initial()
39     a = np.dot(gen_state.T.conj(), tar_state)
40     f = a * a.conj()
41     return f[0][0].real
42
43
44 def irprop(qcircuit):
45     y = []
46     gen_para = qcircuit.get_generator_para()
47     gen_der1 = [0] * len(gen_para)
48     gen_der2 = qcircuit.get_generator_der_list()
49     gen_step = [InitStep] * len(gen_para)
50     dis_para = qcircuit.get_discriminator_para()
51     dis_der1 = [0] * len(dis_para)
52     dis_der2 = [0] * len(dis_para)
53     dis_step = [InitStep] * len(dis_para)
54     for i in range(0, 400):
55         gradient_descent(gen_der1, gen_der2, gen_step, gen_para)
56         qcircuit.change_generator_para(gen_para)
57         dis_der2 = qcircuit.get_discriminator_der_list()
58         gradient_ascent(dis_der1, dis_der2, dis_step, dis_para)
59         qcircuit.change_discriminator_para(dis_para)
60         gen_der1 = gen_der2[:]
61         gen_der2 = qcircuit.get_generator_para()
62         dis_der1 = dis_der2[:]
63         y.append(fidelity(qcircuit))
64     x = np.arange(0, 400, 1)
65     plt.xlabel("iteration")
66     plt.ylabel("fidelity")
67     plt.plot(x, y, 'd')
68     plt.savefig('./test1.png')
69     plt.show()
70
71
72 def simulate():
73     list = [[1, 0], [1, 0], [1, 0]]
74     cir = QC.QCircuitSimulator(2, 2, 3, list)
75     irprop(cir)
76     cir.generate_state()
77     print(cir.main_register.state)
78
79 simulate()

```

Listing 2: circuit

```

1 import Quantum_Circuit.QGate as QG
2 import Quantum_Circuit.QRegister as QR
3 import numpy as np

```

```

4
5
6 class QCircuitSimulator:
7     def __init__(self, GDepth, DDepth, n, pslist):
8         self.qnum = n + 1
9         self.d = 2 ** (n + 1)
10        self.register = QR.QRegister(n + 1)
11        self.main_register = QR.QRegister(n)
12        self.anc_register = QR.QRegister(1)
13        self.E0 = np.array([[1.0]])
14        self.target = pslist
15        self.ggate_list = []
16        self.dgate_list = []
17        for i in range(1, n+2):
18            if i != n+1:
19                self.E0 = np.kron(self.E0, QG.I)
20            else:
21                self.E0 = np.kron(self.E0, QG.P0)
22        counter = 1
23        while counter <= GDepth:
24            self.ggate_list.extend(QG.Layer(n).gate_list)
25            counter += 1
26        counter = 1
27        while counter <= DDepth:
28            self.dgate_list.extend(QG.Layer(n+1).gate_list)
29            counter += 1
30
31        def register_initial(self):
32            self.register.initial()
33            self.main_register.initial()
34            self.anc_register.initial()
35
36        def generate_state(self):
37            gcircuit = np.eye(2 ** (self.qnum-1))
38            for i in self.ggate_list:
39                if isinstance(i, QG.CNOT):
40                    gcircuit = np.dot(i.Cx, gcircuit)
41                else:
42                    gcircuit = np.dot(i.PGate, gcircuit)
43            self.main_register.state = np.dot(gcircuit, self.main_register.state)
44            self.register.state = np.kron(self.main_register.state, self.anc_register.state)
45
46        def discriminate_state(self):
47            dcircuit = np.eye(self.d)
48            for i in self.dgate_list:
49                if isinstance(i, QG.CNOT):
50                    dcircuit = np.dot(i.Cx, dcircuit)
51                else:
52                    dcircuit = np.dot(i.PGate, dcircuit)
53            self.register.state = np.dot(dcircuit, self.register.state)
54
55        def generator_derivative(self, i):

```

```

56     p = self.ggate_list[i].para
57     self.ggate_list[i].change_para(np.pi/2+p)
58     self.generate_state()
59     self.discriminate_state()
60     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
61     ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
62     count = 0
63     for j in range(0,100):
64         if(np.random.rand() < ancprob0):
65             count += 1
66     part1 = count / 100
67     self.register_initial()
68     self.ggate_list[i].change_para(np.pi / 2 - p)
69     self.generate_state()
70     self.discriminate_state()
71     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
72     ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
73     count = 0
74     for j in range(0, 100):
75         if (np.random.rand() < ancprob0):
76             count += 1
77     part2 = count / 100
78     self.register_initial()
79     self.ggate_list[i].change_para(p)
80     der = -0.5 * 0.5 * (part1 - part2)
81     return der
82
83 def discriminator_derivative(self, i):
84     p = self.dgate_list[i].para
85     self.dgate_list[i].change_para(np.pi / 2 + p)
86     self.generate_state()
87     self.discriminate_state()
88     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
89     ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
90     count = 0
91     for j in range(0, 100):
92         if (np.random.rand() < ancprob0):
93             count += 1
94     part1 = count / 100
95     self.register_initial()
96     self.dgate_list[i].change_para(np.pi / 2 - p)
97     self.generate_state()
98     self.discriminate_state()
99     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
100    ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
101    count = 0
102    for j in range(0, 100):
103        if (np.random.rand() < ancprob0):
104            count += 1
105    part2 = count / 100
106    self.register_initial()
107    der1 = -0.5 * 0.5 * (part1 - part2)
108    self.dgate_list[i].change_para(np.pi / 2 + p)

```

```

109     self.main_register.simulate(self.target)
110     self.register.state = np.kron(self.main_register.state, self.anc_register.
        state)
111     self.discriminate_state()
112     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
113     ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
114     count = 0
115     for j in range(0, 100):
116         if (np.random.rand() < ancprob0):
117             count += 1
118     part1 = count / 100
119     self.register_initial()
120     self.dgate_list[i].change_para(np.pi / 2 - p)
121     self.main_register.simulate(self.target)
122     self.register.state = np.kron(self.main_register.state, self.anc_register.
        state)
123     self.discriminate_state()
124     ancqubitstate = np.dot(self.register.state, self.register.state.T.conj())
125     ancprob0 = np.trace(np.dot(self.E0, ancqubitstate))
126     count = 0
127     for j in range(0, 100):
128         if (np.random.rand() < ancprob0):
129             count += 1
130     part2 = count / 100
131     self.register_initial()
132     self.dgate_list[i].change_para(p)
133     der2 = 0.5 * 0.5 * (part1 - part2)
134     der = der1 + der2
135     return der
136
137 def get_discriminator_para(self):
138     list = []
139     for i in self.dgate_list:
140         if isinstance(i, QG.ParameterizedGate):
141             list.append(i.para)
142     return list
143
144 def get_generator_para(self):
145     list = []
146     for i in self.ggate_list:
147         if isinstance(i, QG.ParameterizedGate):
148             list.append(i.para)
149     return list
150
151 def get_discriminator_der_list(self):
152     list = []
153     for i in range(0, len(self.dgate_list)):
154         if isinstance(self.dgate_list[i], QG.ParameterizedGate):
155             list.append(self.discriminator_derivative(i))
156     return list
157
158 def get_generator_der_list(self):
159     list = []

```



```

160         for i in range(0, len(self.ggate_list)):
161             if isinstance(self.ggate_list[i], QG.ParameterizedGate):
162                 list.append(self.generator_derivative(i))
163         return list
164
165     def change_generator_para(self, paralist):
166         j = 0
167         for i in self.ggate_list:
168             if isinstance(i, QG.ParameterizedGate):
169                 i.change_para(paralist[j])
170                 j += 1
171
172     def change_discriminator_para(self, paralist):
173         j = 0
174         for i in self.dgate_list:
175             if isinstance(i, QG.ParameterizedGate):
176                 i.change_para(paralist[j])
177                 j += 1

```

Listing 3: gate

```

1 import numpy as np
2 from scipy import linalg as lin
3
4 Zero = np.array([[1.0],[0.0]])
5 One = np.array([[0.0],[1.0]])
6 X = np.array([[0,1],[1,0]])
7 Y = np.array([[0,-1j],[1j,0]])
8 Z = np.array([[1,0],[0,-1]])
9 I = np.eye(2)
10 P0 = np.dot(Zero, Zero.T)
11 P1 = np.dot(One, One.T)
12
13
14 class CNOT:
15     def __init__(self, i, j, n):
16         counter = 1
17         self.control_bit = i
18         self.act_bit = j
19         Cx1 = np.array([[1.0]])
20         Cx2 = np.array([[1.0]])
21         while counter <= n:
22             if counter == i:
23                 Cx1 = np.kron(Cx1, P0)
24                 Cx2 = np.kron(Cx2, P1)
25             elif counter == j:
26                 Cx1 = np.kron(Cx1, I)
27                 Cx2 = np.kron(Cx2, X)
28             else:
29                 Cx1 = np.kron(Cx1, I)
30                 Cx2 = np.kron(Cx2, I)
31             counter += 1
32         self.Cx = Cx1 + Cx2
33

```

```

34
35 class ParameterizedGate:
36     def __init__(self, i, g, n):
37         self.para = np.random.uniform(-1, 1) * np.pi
38         self.act_bit = i
39         self.Gate = np.array([[1.0]])
40         counter = 1
41         while counter <= n:
42             if counter == i:
43                 self.Gate = np.kron(self.Gate, g)
44             else:
45                 self.Gate = np.kron(self.Gate, I)
46             counter += 1
47         self.PGate = lin.expm(-0.5j * self.para * self.Gate)
48
49     def change_para(self, p):
50         self.para = p
51         self.PGate = lin.expm(-0.5j * self.para * self.Gate)
52
53
54 class TwoQbitGate:
55     def __init__(self, i, j, n):
56         self.tq_gate_list = []
57         self.tq_gate_list.append(ParameterizedGate(i, Z, n))
58         self.tq_gate_list.append(ParameterizedGate(i, Y, n))
59         self.tq_gate_list.append(ParameterizedGate(i, Z, n))
60         self.tq_gate_list.append(ParameterizedGate(j, Z, n))
61         self.tq_gate_list.append(ParameterizedGate(j, Y, n))
62         self.tq_gate_list.append(ParameterizedGate(j, Z, n))
63         self.tq_gate_list.append(CNOT(j, i, n))
64         self.tq_gate_list.append(ParameterizedGate(i, Z, n))
65         self.tq_gate_list.append(ParameterizedGate(j, Y, n))
66         self.tq_gate_list.append(CNOT(i, j, n))
67         self.tq_gate_list.append(ParameterizedGate(j, Y, n))
68         self.tq_gate_list.append(CNOT(j, i, n))
69         self.tq_gate_list.append(ParameterizedGate(i, Z, n))
70         self.tq_gate_list.append(ParameterizedGate(i, Y, n))
71         self.tq_gate_list.append(ParameterizedGate(i, Z, n))
72         self.tq_gate_list.append(ParameterizedGate(j, Z, n))
73         self.tq_gate_list.append(ParameterizedGate(j, Y, n))
74         self.tq_gate_list.append(ParameterizedGate(j, Z, n))
75
76
77 class Layer:
78     def __init__(self, n):
79         counter = 1
80         self.gate_list = []
81         while counter < n:
82             self.gate_list.extend(TwoQbitGate(counter, counter+1, n).tq_gate_list)
83             counter += 2
84         counter = 2
85         while counter < n:
86             self.gate_list.extend(TwoQbitGate(counter, counter+1, n).tq_gate_list)

```

```
87         counter += 2
```

Listing 4: register

```
1  import numpy as np
2
3
4  Zero = np.array([[1.0],[0.0]])
5  One = np.array([[0.0],[1.0]])
6
7
8  class QRegister:
9      def __init__(self, n):
10         self.length = n
11         counter = 1
12         self.state = np.array([[1.0]])
13         while counter <= n:
14             self.state = np.kron(self.state, Zero)
15             counter += 1
16
17     def initial(self):
18         counter = 1
19         self.state = np.array([[1.0]])
20         while counter <= self.length:
21             self.state = np.kron(self.state, Zero)
22             counter += 1
23
24     def simulate(self, pslist):
25         counter = 0
26         self.state = np.array([[1.0]])
27         while counter < self.length:
28             singlebitstate = pslist[counter][0] * Zero + pslist[counter][1] * One
29             self.state = np.kron(self.state, singlebitstate)
30             counter += 1
```