Quantum Generative Adversarial Network with Noise

Project Name: Quantum Generative Adversarial Network with Noise

Project member:

XXX
XXX
XXX

Dodument Type: Report

Project Start Time: 19/04/2020

Sourcecode Version: 0.0.1

Keywords: Variational Quantum Circuit, Machine Learning

Modify April 19, 2020

Submitted by:

YIXUAN ZHU

Contents

1 Experiment

Run the code then draw the picture
maxcut gradient
maxcut adam
maxcut adagrad
maxcut COBYLA
maxcut momentum

2 Next Plan

P: run the next code
eqv and GAN

3 Appendix

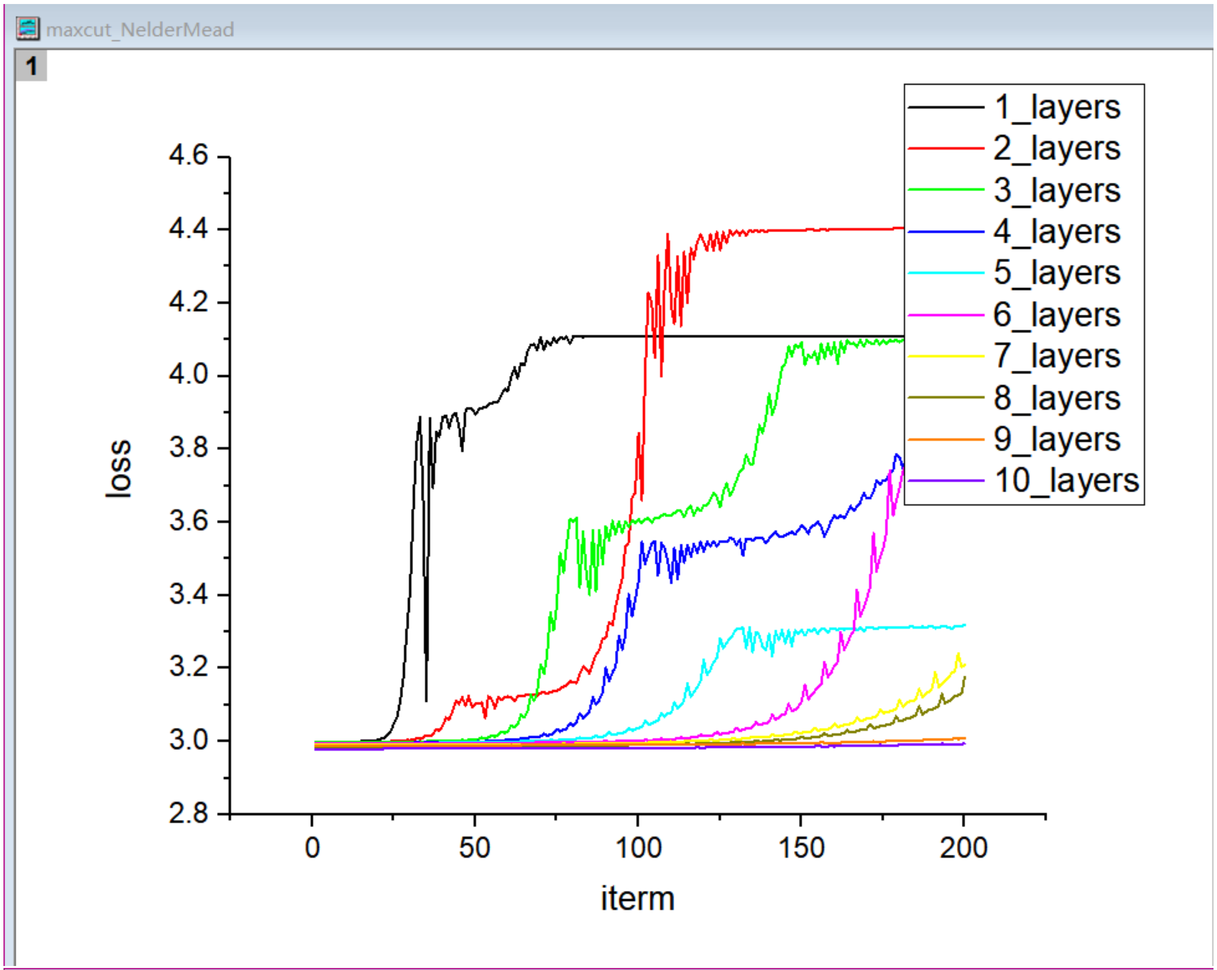
A Source Code

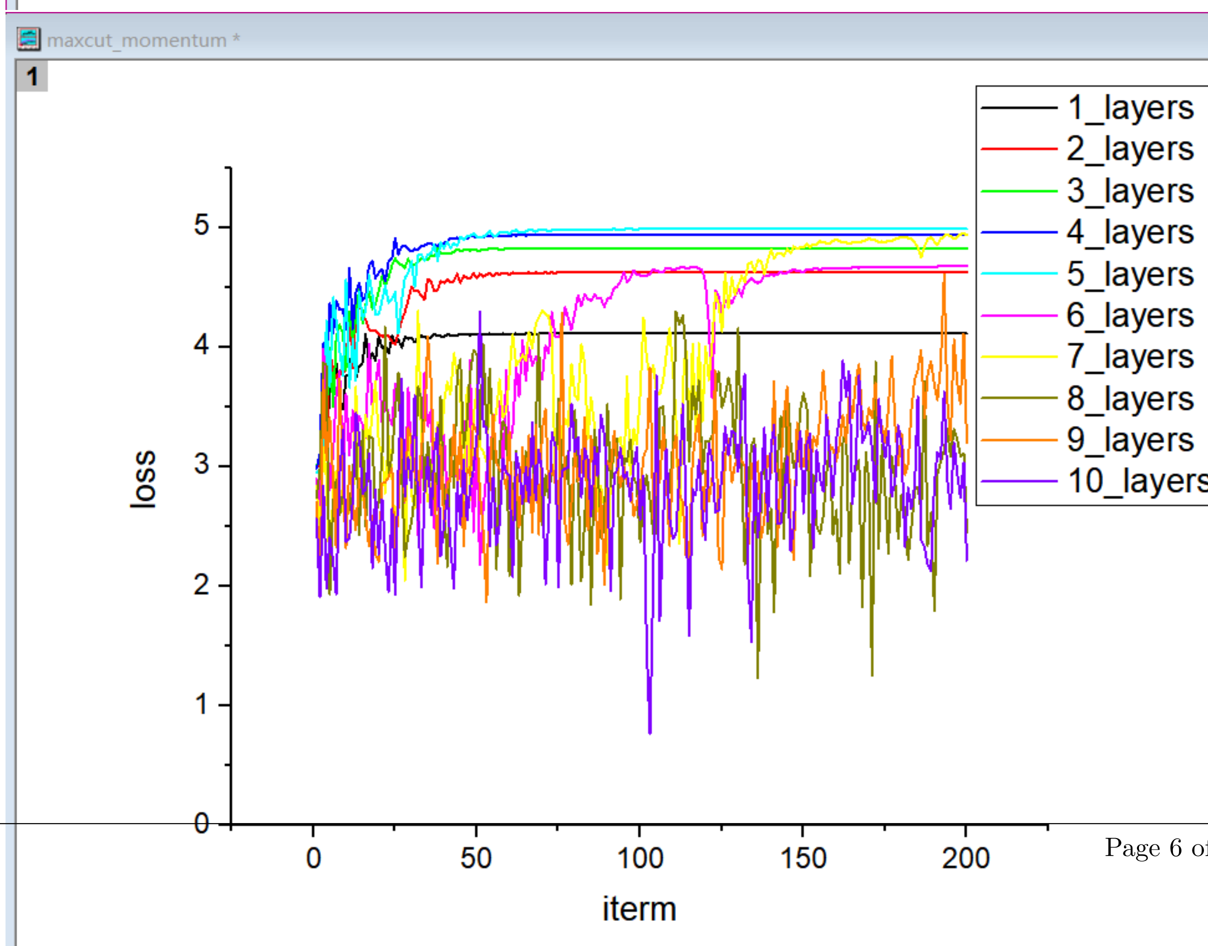
```

import pennylane as qml
from pennylane import numpy as np


##############################################################################
# Operators
# ~~~~~~~~~
# We specify the number of qubits (vertices) with ``n_wires`` and
# compose the unitary operators using the definitions
# above. :math:`U_B` operators act on individual wires, while :math:`U_C`
# operators act on wires whose corresponding vertices are joined by an edge in
# the graph. We also define the graph using
# the list ``graph``, which contains the tuples of vertices defining
# each edge in the graph.

n_wires = 5
graph = [(0, 1),(0, 3),(0,4),(2,3),(1,2),(3,4)]

# unitary operator U_B with parameter beta
def U_B(beta):
    for wire in range(n_wires):
        qml.RX(2 * beta, wires=wire)


# unitary operator U_C with parameter gamma
def U_C(gamma):
    for edge in graph:
        wire1 = edge[0]
        wire2 = edge[1]
```

```
        qml.CNOT(wires=[wire1, wire2])
        qml.RZ(gamma, wires=wire2)
        qml.CNOT(wires=[wire1, wire2])


##############################################################################
# We will need a way to sample
# a measurement of multiple qubits in the computational basis, so we define
# a Hermitian operator to do this. The eigenvalues of the operator are
# the qubit measurement values in integer form.


def comp_basis_measurement(wires):
    n_wires = len(wires)
    return qml.Hermitian(np.diag(range(2 ** n_wires)), wires=wires)


##############################################################################
# Circuit
# ~~~~~~~
# Next, we create a quantum device with 4 qubits.

dev = qml.device("default.qubit", wires=n_wires, analytic=True, shots=1)

##############################################################################
# We also require a quantum node which will apply the operators according to the
# angle parameters, and return the expectation value of the observable
# :math:`\sigma_z^{j}\sigma_z^{k}` to be used in each term of the objective function
    later on. The
# argument ``edge`` specifies the chosen edge term in the objective function, :math:`(
    j,k)`.
# Once optimized, the same quantum node can be used for sampling an approximately
    optimal bitstring
# if executed with the ``edge`` keyword set to ``None``. Additionally, we specify the
    number of layers
# (repeated applications of :math:`U_BU_C`) using the keyword ``n_layers``.

import xlrd

from xlutils.copy import copy as xl_copy

#V
rb = xlrd.open_workbook("./DATA/maxcut_Adam_DATA.xls",formatting_info=True)
workbook=xl_copy(rb)
print(workbook)
sheet = rb.sheets()[0]
col =sheet.ncols
sheet = workbook.get_sheet(0)


pauli_z = [[1, 0], [0, -1]]
pauli_z_2 = np.kron(pauli_z, pauli_z)

```

```
@qml.qnode(dev)
def circuit(gammas, betas, edge=None, n_layers=1):
    # apply Hadamards to get the n qubit |+> state
    for wire in range(n_wires):
        qml.Hadamard(wires=wire)
    # p instances of unitary operators
    for i in range(n_layers):
        U_C(gammas[i])
        U_B(betas[i])
    if edge is None:
        # measurement phase
        return qml.sample(comp_basis_measurement(range(n_wires)))
    # during the optimization phase we are evaluating a term
    # in the objective using expval
    return qml.expval(qml.Hermitian(pauli_z_2, wires=edge))


import xlwt
import xlrd

from xlutils.copy import copy as xl_copy
'''
#rb = xlrd.open_workbook("./DATA/maxcut_Adagrad.xls",formatting_info=True)

workbook=xl_copy(rb)
print(workbook)
sheet = rb.sheets()[0]
col=sheet.ncols
sheet = workbook.get_sheet(0)
'''

def qaoa_maxcut(n_layers=1):
    print("\np={:d}".format(n_layers))

    # initialize the parameters near zero
    init_params = 0.01 * np.random.rand(2, n_layers)

    # minimize the negative of the objective function
    def objective(params):
        gammas = params[0]
        betas = params[1]
        neg_obj = 0
        for edge in graph:
            # objective for the MaxCut problem
            neg_obj -= 0.5 * (1 - circuit(gammas, betas, edge=edge, n_layers=n_layers)
                )
        return neg_obj

    # initialize optimizer: Adagrad works well empirically
    opt = qml.AdamOptimizer(stepsize=0.1)

    # optimize parameters in objective
```

```
    params = init_params
    steps = 200
    for i in range(steps):
        params = opt.step(objective, params)
        if (i + 1) % 1 == 0:
            print("Objective after step {:5d}: {: .7f}".format(i + 1, -objective(
                params)))
            #sheet.write(i, col, "{:0.7f}".format(-objective(params)))
            #sheet.write(i, col, "{:0.7f}".format(-objective(params)))

    #workbook.save('./DATA/maxcut_momentum.xls')

    # sample measured bitstrings 100 times
    bit_strings = []
    n_samples = 200
    for i in range(0, n_samples):
        bit_strings.append(int(circuit(params[0], params[1], edge=None, n_layers=
            n_layers)))

    # print optimal parameters and most frequently sampled bitstring
    counts = np.bincount(np.array(bit_strings))
    most_freq_bit_string = np.argmax(counts)
    print("Optimized (gamma, beta) vectors:\n{}".format(params[:, :n_layers]))
    print("Most frequently sampled bit string is: {:05b}".format(most_freq_bit_string)
        )
    workbook.save('./DATA/maxcut_Adam_DATA.xls')
    return -objective(params), bit_strings

n_layers = 10
bitstrings2 = qaoa_maxcut(n_layers=n_layers)[1]

import matplotlib.pyplot as plt

xticks = range(0, 2**n_wires)
#xtick_labels = list(map(lambda x: format(x, "04b"), xticks))
xtick_labels = list(map(lambda x: format(x, "0"+str(n_wires)+"b"), xticks))
bins = np.arange(0, 2**n_wires+1) - 0.5

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.title("n_layers=1")
plt.xlabel("bitstrings")
plt.ylabel("freq.")
plt.xticks(xticks, xtick_labels, rotation="vertical")
plt.hist(bitstrings2, bins=bins)
plt.subplot(1, 2, 2)
plt.title("n_layers={}".format(n_layers))
plt.xlabel("bitstrings")
plt.ylabel("freq.")
plt.xticks(xticks, xtick_labels, rotation="vertical")
plt.hist(bitstrings2, bins=bins)
plt.tight_layout()
plt.show()
```


maxcut_NelderMead
1
1_layers
2_layers
3_layers
4_layers
5_layers
6_layers
7_layers
8_layers
9_layers
10_layers
loss
4.6
4.4
4.2
4.0
3.8
3.6
3.4
3.2
3.0
2.8
0
50
100
150
200
iterm


maxcut_momentum *
1
1_layers
2_layers
3_layers
4_layers
5_layers
6_layers
7_layers
8_layers
9_layers
10_layers
loss
5
4
3
2
1
0
0
50
100
150
200
iterm