

日本語訳『Qiskit Textbook Machine Learning』勉強会

- ・教師なし学習
 - ・量子敵対的生成ネットワーク
-

Kifumi Numata (Qiskit Advocate)

Sep 21, 2022

本文(和): <https://ja.learn.qiskit.org/course/machine-learning/unsupervised-learning>
<https://ja.learn.qiskit.org/course/machine-learning/quantum-generative-adversarial-networks>

Qiskit テキストブック (beta)

量子機械学習編

和訳：<https://ja.learn.qiskit.org/course/machine-learning/introduction>

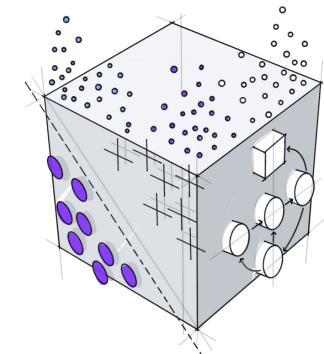
- はじめに(4月)
- パラメーター化量子回路(4月)
- データ符号化(5月)
- パラメーター化量子回路の学習(6月)
- 教師あり学習(7月)
- 変分分類(7月)
- 量子特徴量マップとカーネル(8月)
- 教師なし学習(9月)
- 量子敵対的生成ネットワーク(9月)
- プロジェクト

Quantum machine learning

This course contains around eight hours of content, and is aimed at self-learners who are comfortable with undergraduate-level mathematics and quantum computing fundamentals. This course will take you through key concepts in quantum machine learning, such as parameterized quantum circuits, training these circuits, and applying them to basic problems. By the end of the course, you'll understand the state of the field, and you'll be familiar with recent developments in both supervised and unsupervised learning such as quantum kernels and general adversarial networks. This course finishes with a project that you can use to showcase what you've learnt.

This course was created by IBM Quantum with the help of Qiskit Advocates through the Qiskit Advocate Mentoring Program.

Start learning



Qt

Qiskit テキストブック (beta)

量子機械学習編

和訳：<https://ja.learn.qiskit.org/course/machine-learning/introduction>

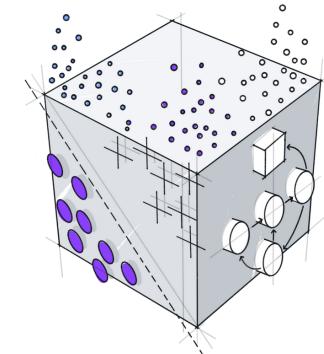
- はじめに(4月)
- パラメーター化量子回路(4月)
- データ符号化(5月)
- パラメーター化量子回路の学習(6月)
- 教師あり学習(7月)
- 変分分類(7月)
- 量子特徴量マップとカーネル(8月)
- **教師なし学習(9月)**
- **量子敵対的生成ネットワーク(9月)**
- プロジェクト

Quantum machine learning

This course contains around eight hours of content, and is aimed at self-learners who are comfortable with undergraduate-level mathematics and quantum computing fundamentals. This course will take you through key concepts in quantum machine learning, such as parameterized quantum circuits, training these circuits, and applying them to basic problems. By the end of the course, you'll understand the state of the field, and you'll be familiar with recent developments in both supervised and unsupervised learning such as quantum kernels and general adversarial networks. This course finishes with a project that you can use to showcase what you've learnt.

This course was created by IBM Quantum with the help of Qiskit Advocates through the Qiskit Advocate Mentoring Program.

Start learning

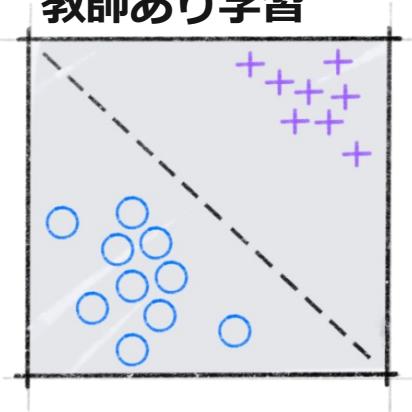


Qt

機械学習

機械学習は、以下の3つの分野に大別されます。

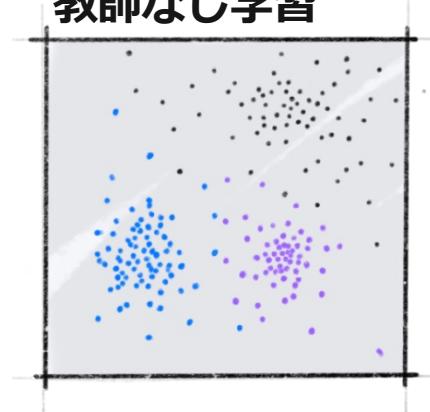
教師あり学習



ラベル付きデータ (x_i, y_i) :
マッピングする関数 $y = f(x)$ を学習。

例) 猫や犬の写真がラベル付けされた
集合から、新しい猫や犬の写真を識別する。

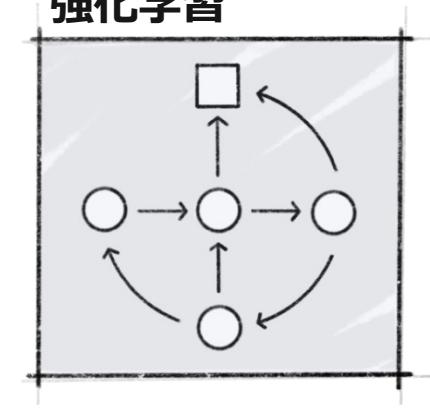
教師なし学習



ラベルなしデータ :
何らかの構造を学習。

例) 映画の視聴履歴に基づいて視聴者
をグループ分けし、新しい映画を推薦
する。

強化学習



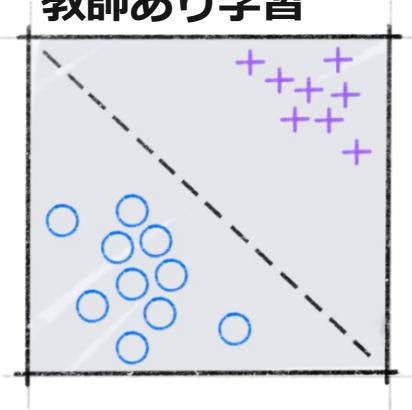
行動に応じて報酬が得られる環境
で、期待される報酬を最大化。

例) 「パックマン」のプレイ方法を
アルゴリズムで学習する。

機械学習

機械学習は、以下の3つの分野に大別されます。

教師あり学習

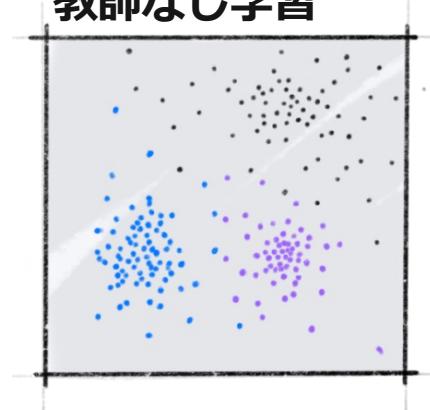


ラベル付きデータ (x_i, y_i) :
マッピングする関数 $y = f(x)$ を学習。

例) 猫や犬の写真がラベル付けされた
集合から、新しい猫や犬の写真を識別する。

変分分類 (量子NN)
量子特徴量マップとカーネル

教師なし学習

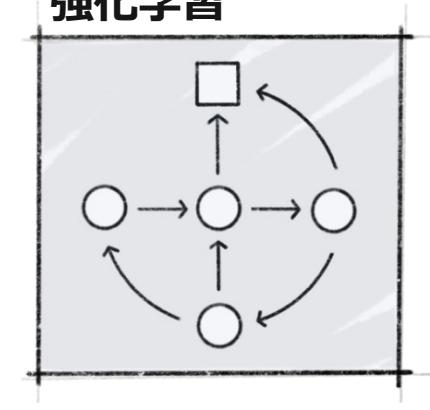


ラベルなしデータ :
何らかの構造を学習。

例) 映画の視聴履歴に基づいて視聴者
をグループ分けし、新しい映画を推薦
する。

量子敵対的生成ネットワーク

強化学習

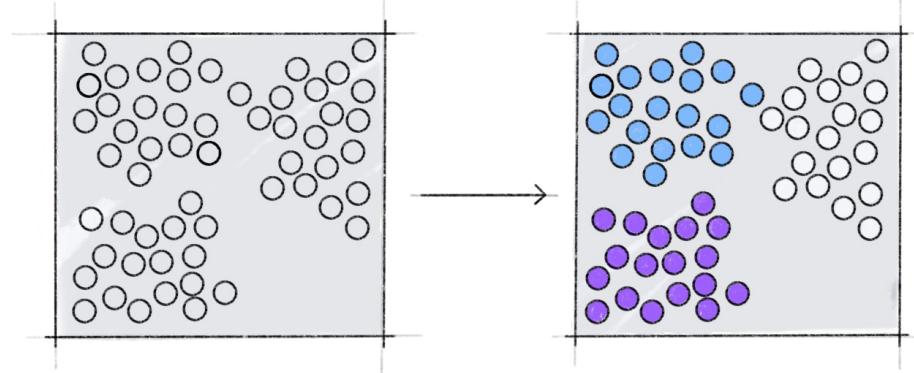


行動に応じて報酬が得られる環境
で、期待される報酬を最大化。

例) 「パックマン」のプレイ方法を
アルゴリズムで学習する。

教師なし学習

ラベルのないデータに基づいてデータのパターンと傾向を見つけて学習する機械学習タスク



古典の例)

- 主成分分析(principal component analysis, PCA)
- クラスタリング
- 変分オートエンコーダー(variational autoencoders, VAE)
- 敵対的生成ネットワーク(generative adversarial networks, GAN)
- 潜在拡散モデル(流行りの絵を描いてくれる人工知能)

敵対的生成ネットワーク

GAN (Generative Adversarial Networks)

- 古典の教師なし学習において最も広く使われている手法の一つ
- 写実的な画像生成や作曲が可能

<https://thiscatdoesnotexist.com/>



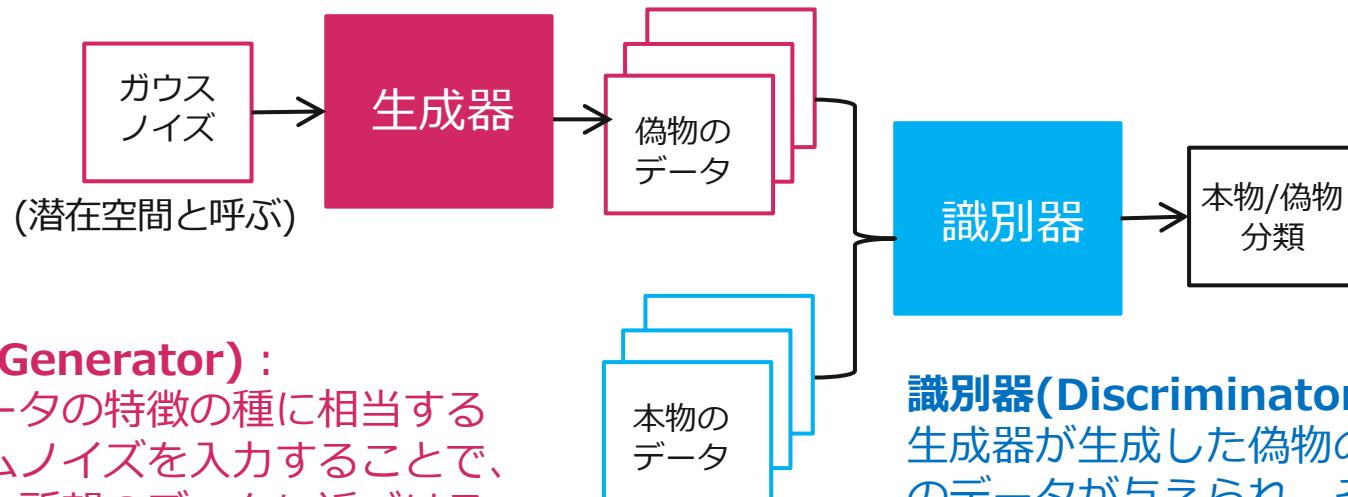
GANは「生成モデル」の一つ

生成モデルとは、ラベルなしのデータの分布を全体的にモデル化して学習して、新しいデータサンプルを**生成**できるようにする機械学習のタイプ。

GANで生成された猫(偽の猫)

敵対的生成ネットワーク

2つのニューラルネットワークで構成。



生成器(Generator) :

生成データの特徴の種に相当するランダムノイズを入力することで、ノイズを所望のデータに近づけるようにマッピング。

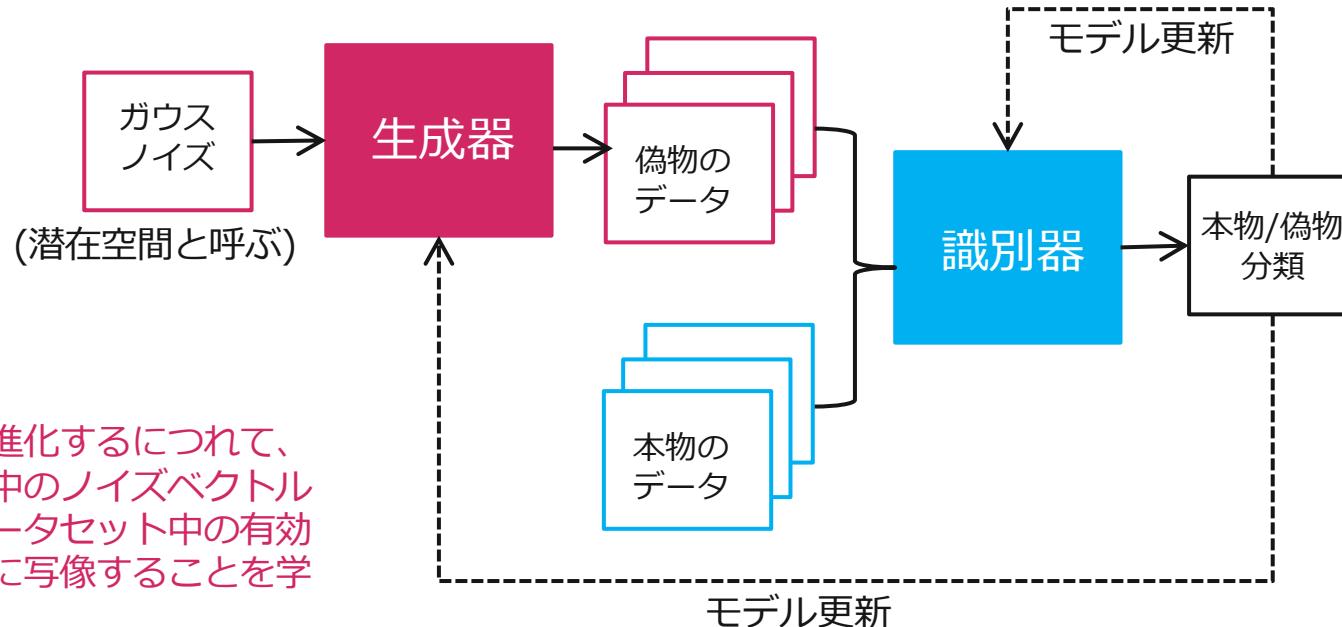
識別器(Discriminator) :

生成器が生成した偽物のデータと本物のデータが与えられ、その真偽を判定。

2つのネットワークを交互に競合させ、学習を進めることで、生成器は本物のデータに近い偽物データを生成できるようになる。

敵対的生成ネットワーク

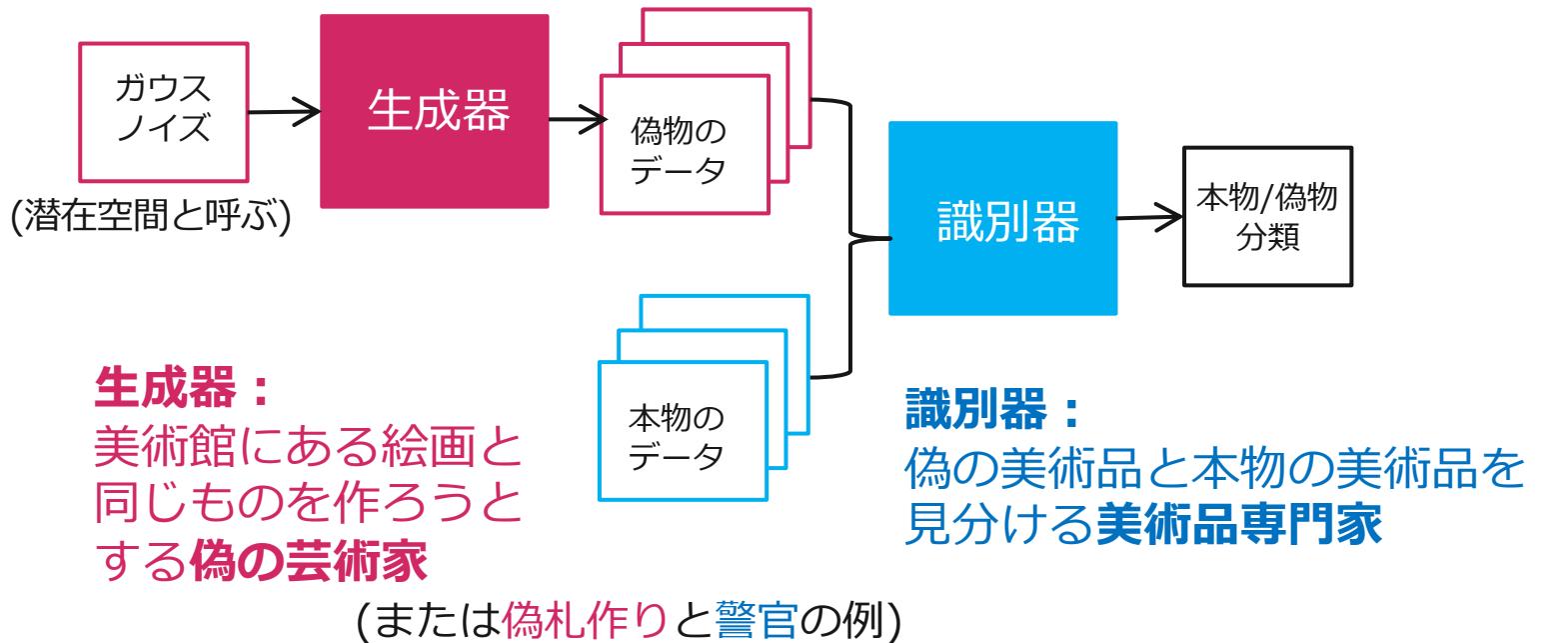
2つのニューラルネットワークで構成。



収束

生成器が実際のデータ分布と区別がつかないようなデータサンプルを生成し、識別器が真偽を判定できない状態になった時点で終了。

敵対的生成ネットワークの例え

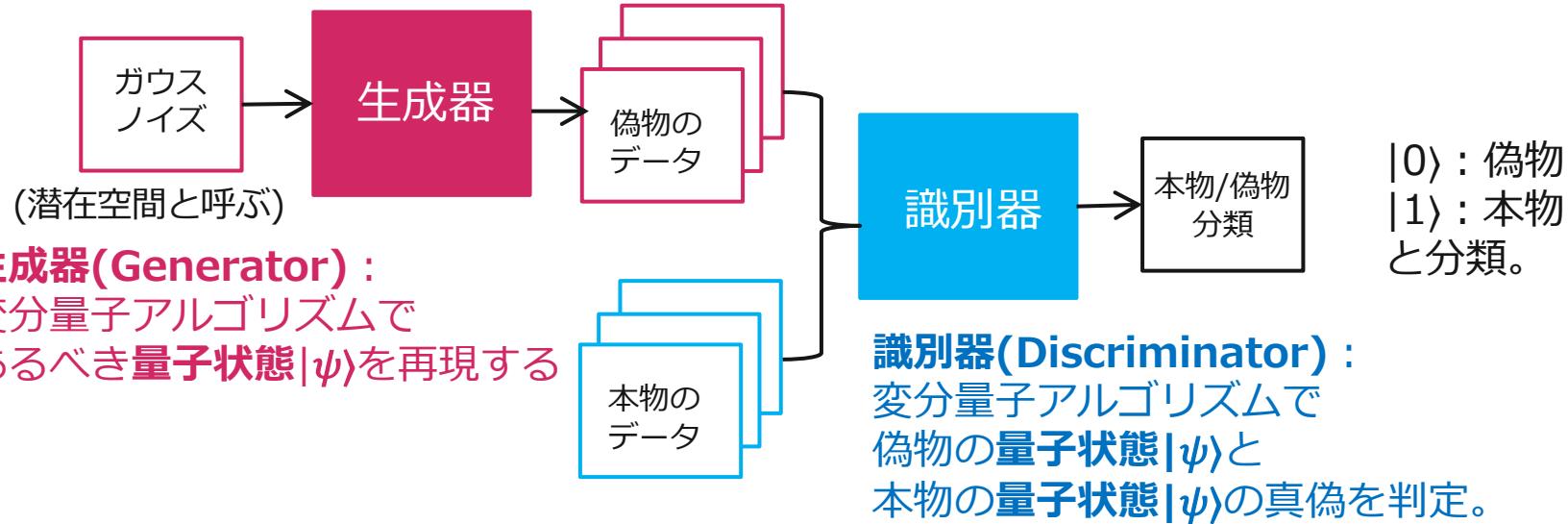


いたちごっこを繰り返して最終的には本物に近い偽物が生成されるようになる。

量子敵対的生成ネットワーク

ここでは、生成器、識別器とともに量子のものを説明。(片方が古典のものなどもある)

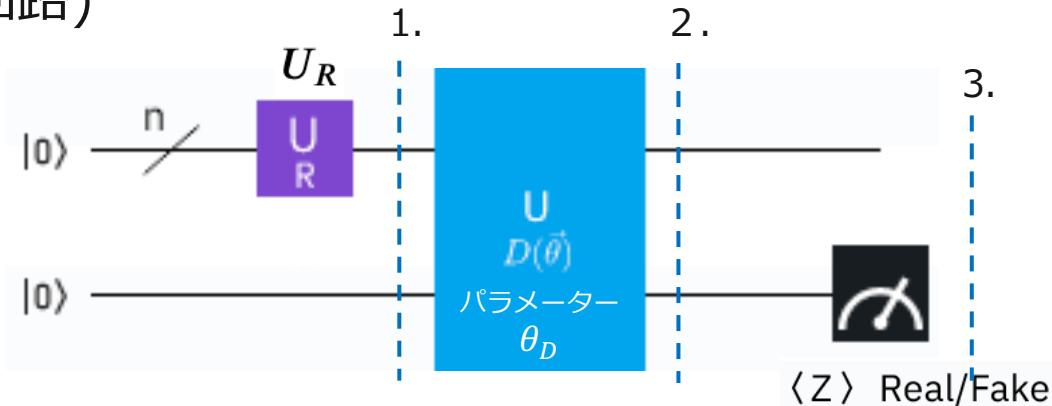
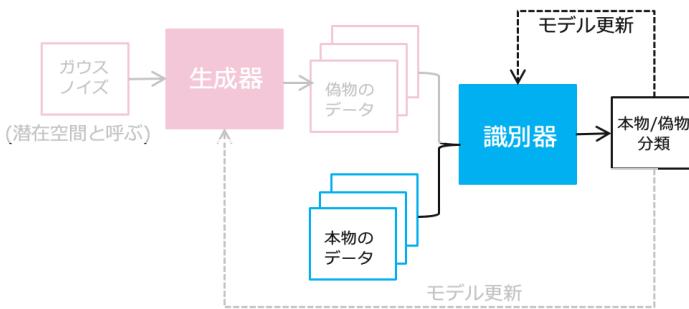
2つのパラメーター化回路の学習で、量子状態(確率分布)を再現する。



量子状態(確率分布)を再現できるとよいこと

- 量子積分や量子振幅増幅などの初期状態の準備として使える
- (愚直に作るとCNOT数が指数関数的に増えて現実的に実行できない)

識別器 (パラメーター θ_D の回路)



1. 本物のデータをユニタリー演算 U_R で量子状態にエンコード :

$$|\text{Data}_R\rangle = U_R |0\rangle^{\otimes n}$$

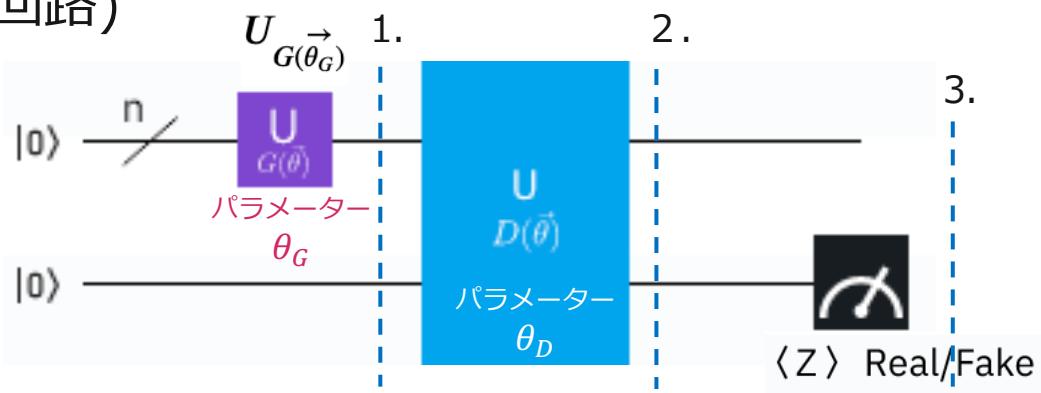
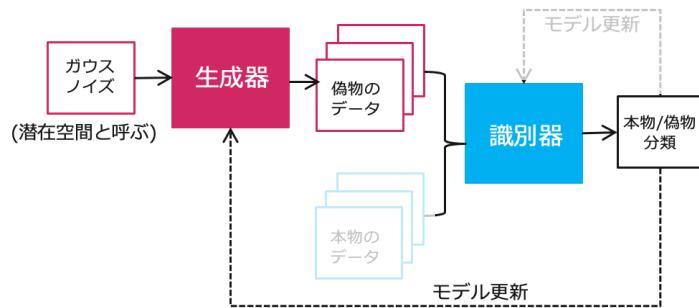
2. パラメーター化された識別器 $U_{D(\vec{\theta}_D)}$ を通す。

$$U_{D(\vec{\theta}_D)} (|\text{Data}_R\rangle \otimes |0\rangle)$$

3. 物理量 $I^{\otimes n} Z$ の期待値をとることで本物のデータに対する識別器のスコアを測定によって求める。
 $|0\rangle$ に対応する結果のときは入力データは偽物、 $|1\rangle$ に対応する結果のときは入力データは本物と分類する
 ように、 $\vec{\theta}_D$ を更新する。

$|0\rangle$: 偽物
 $|1\rangle$: 本物
 と分類。

生成器 (パラメーター θ_G の回路)



1. 偽の量子状態をパラメーター θ_G の $U_{G(\vec{\theta}_G)}$ で準備する。

$$|\text{Data}_G\rangle = U_{G(\vec{\theta}_G)} |0^{\otimes n}\rangle$$

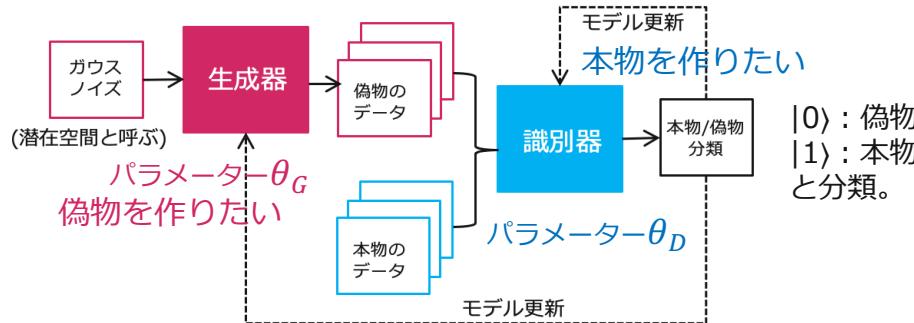
2. パラメーター化された識別器 $U_{D(\vec{\theta}_D)}$ に通す。

$$U_{D(\vec{\theta}_D)} (|\text{Data}_G\rangle \otimes |0\rangle)$$

3. 物理量 $I^{\otimes n} Z$ の期待値をとることで偽物のデータに対する識別器のスコアが得られる。
 $|0\rangle$ に対応する結果のときは入力データは偽物、 $|1\rangle$ に対応する結果のときは入力データは本物と分類する。

$|0\rangle$: 偽物
 $|1\rangle$: 本物
 と分類。

学習方法



ミニマックス決定法則で敵対するインセンティブを定式化

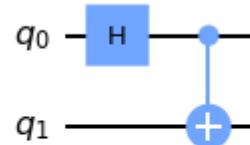
$$\min_{\vec{\theta}_G} \max_{\vec{\theta}_D} \left(\Pr_{\text{Discriminator}} \left(D(\vec{\theta}_D, R) = |\text{real}\rangle \right) + \Pr_{\text{Generator}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{fake}\rangle \right) \right)$$

識別器は、本物のデータを $|\text{real}\rangle$ と分類する確率を最大化(マックス)するように、 θ_D を最適化。

生成器は、逆で、識別器が偽物のデータを $|\text{fake}\rangle$ と分類する確率を最小化(ミニ)するように θ_G を最適化。
($|\text{real}\rangle$ と分類する確率を最大化(マックス)する)

実装その1: 2量子ビットのベル状態を再現したい

本物の分布 : $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

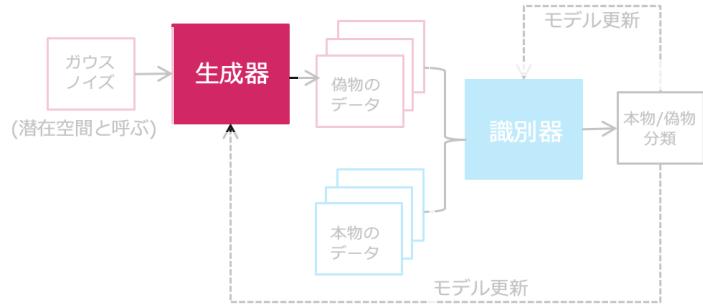


実装方針

生成器と識別器のパラメーター化回路と勾配計算まで、Qiskit。最適化はTensorFlowで古典計算。

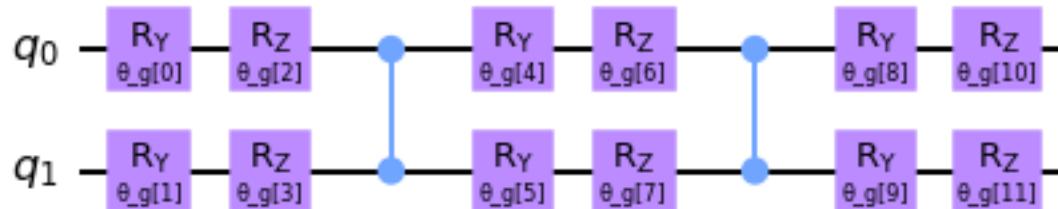
- 生成器と識別器のパラメーター化回路 : Qiskitの`circuit.library`の`TwoLocal`と`ParameterVector`クラス
- コスト関数の計算 : パラメーター化回路をQiskit Statevectorでシミュレーション
- 勾配計算 : Qiskit Machine Learningモジュールの`CircuitQNN`クラス
- 最適化 : TensorFlow(機械学習ライブラリー)のKeras(深層学習ライブラリー)のAdamオプティマイザー

変分量子生成器の実装

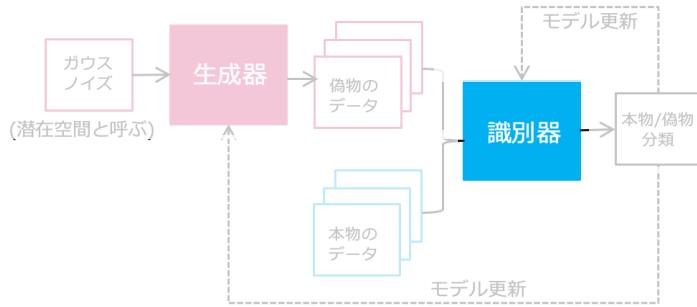


QGANに最適な生成器や識別器のAnsatz (仮の量子回路)はまだ見つかっていない。

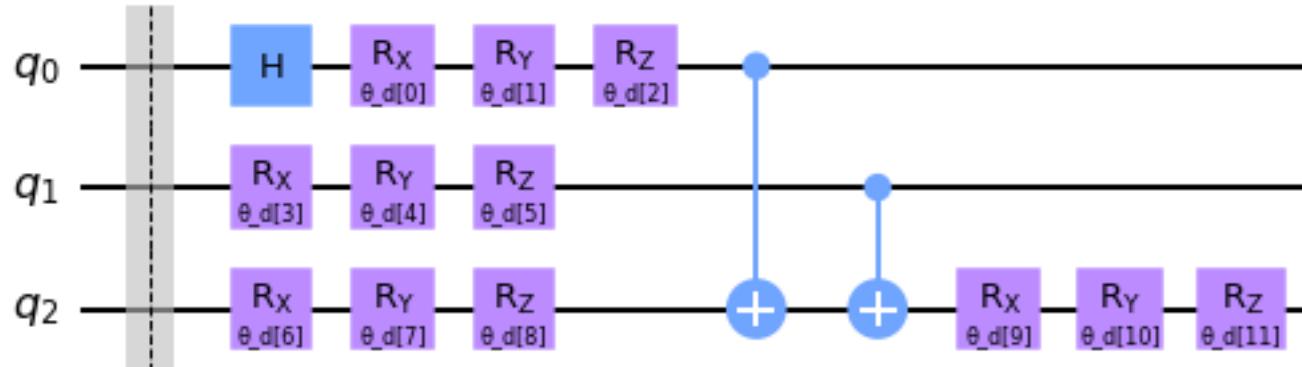
Ansatzを選ぶ際は、再現したい量子状態 $|\psi\rangle$ を完全に再現するのに十分な容量と表現力を持つ必要がある。今回は、少し恣意的に、TwoLocalクラス(R_y , R_z , CZ)を使う。



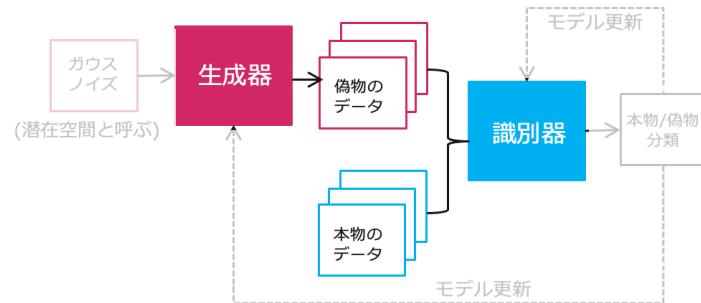
変分量子識別器の実装



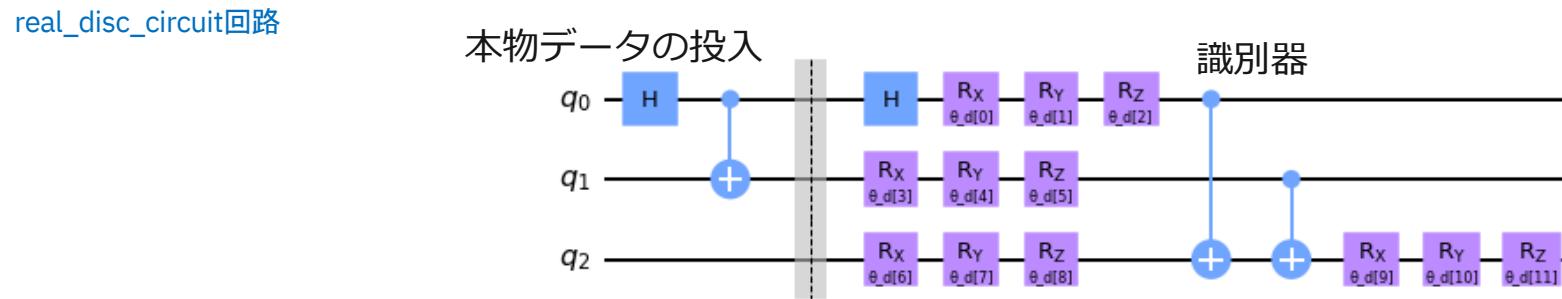
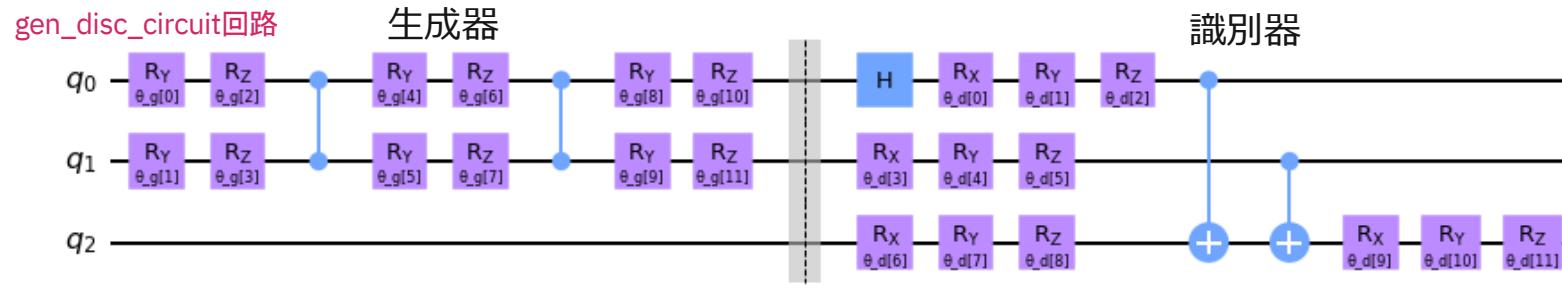
ParameterVectorクラスを使ってカスタムな変分量子回路を作成。(H, Rx, Ry, Rz, CX)



2つのパラメーター化回路



古典GANと違って、量子の場合は、全て”偽物の”波動関数を直接識別器に入力するため、ノイズの役割はそれほどないので、ここではシンプルにするために、変分量子生成器へのノイズの投入は省略。



コスト関数の構築と実装

ミニマックス決定則

$$\min_{\vec{\theta}_G} \max_{\vec{\theta}_D} \left(\Pr_{\substack{\text{Discriminator} \\ \text{Disc}}} \left(D(\vec{\theta}_D, R) = |\text{real}\rangle \right) + \Pr_{\substack{\text{Generator} \\ \text{Generator}}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{fake}\rangle \right) \right)$$

を $|\text{real}\rangle$ に揃えて

識別器(Discriminator)のコスト関数は、以下を最小化するように定義。

$$\text{Cost}_D = \Pr_{\substack{\text{Disc} \\ \text{Generator}}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{real}\rangle \right) - \Pr_{\substack{\text{Discriminator} \\ \text{Disc}}} \left(D(\vec{\theta}_D, R) = |\text{real}\rangle \right)$$

生成器が作った

偽データを本物だと思う

(誤分類する)確率を最小化

本物のデータ回路からの

真データを正しく分類する確率を最大化(逆符号をつける)

生成器(Generator)のコスト関数は、以下を最小化するように定義。

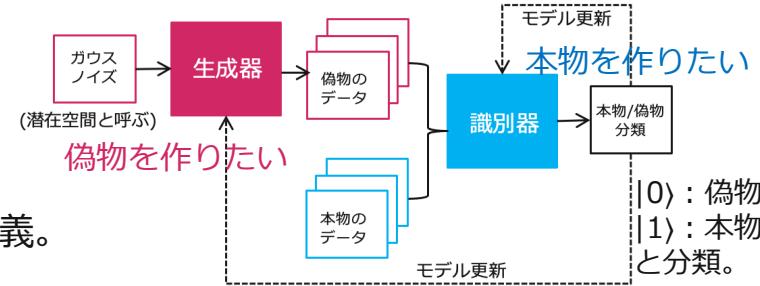
$$\text{Cost}_G = -\Pr_{\substack{\text{Disc} \\ \text{Generator}}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{real}\rangle \right)$$

生成器が作った偽データを識別器が本物だと思って欲しいので

偽データを誤分類する確率を最大化(逆符号をつける)

上記のコスト関数を実装するには、

最後の量子ビットが $|1\rangle$ として測定される確率を出すことで、与えられたサンプルが $|\text{real}\rangle$ (つまり $|1\rangle$)である確率を使って計算。今回は、Statevectorシミュレーターを使う。



コスト関数

生成器(Generator)のコスト関数

$$\text{Cost}_G = -\Pr_{\substack{\text{Disc} \\ \text{Generator}}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{real}\rangle \right)$$

生成器が作った偽データを識別器が本物だと思って欲しいので
偽データを誤分類する確率を最大化(逆符号をつける)

```
def generator_cost(gen_params):
```

"""オプティマイザが最小化するための生成器コスト関数。"""

.numpy()メソッドは、TensorFlowテンソルからnumpy配列を抽出します。

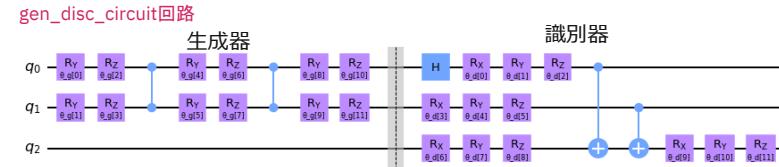
curr_params = np.append(disc_params.numpy(),
 gen_params.numpy()) ← パラメーター(θ_D と θ_G) をnp.ndarray配列に
 (このパラは後ほどTensorFlow形式で準備)

state_probs = Statevector(gen_disc_circuit
 .bind_parameters(curr_params)) ← パラメーターを生成器の回路に
 バインドして、状態ベクトルを計算

|1>をq2で測定する確率の総和を求める

prob_fake_true = np.sum(state_probs[0b100:]) ← q2が|1>= |real>である確率を求める
cost = -prob_fake_true ← (|1□□)の確率を足し合わせる

return cost ← Cost_Gはマイナスがつく



Statevectorの出力形式：

[000 001 010 011 100 101 110 111]

コスト関数

識別器(Discriminator)のコスト関数

$$\text{Cost}_D = \Pr_{\substack{\text{Disc} \\ \text{Generator}}} \left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{real}\rangle \right) - \Pr_{\substack{\text{Discriminator}}} \left(D(\vec{\theta}_D, R) = |\text{real}\rangle \right)$$

生成器が作った
 偽データを本物だと思う
 (誤分類する)確率を最小化
 本物のデータ回路からの
 真データを正しく分類する確率を最大化(逆符号をつける)

CircuitQNN で勾配計算

Qiskit Machine LearningモジュールのCircuitQNNクラスを使って、勾配計算を行う。

ここでCircuitQNNを設定しておくと、forward()メソッドで、回路の確率状態ベクトルが直接出力される。

```
from qiskit.utils import QuantumInstance
from qiskit_machine_learning.neural_networks import CircuitQNN

# Quantumインスタンスの定義 (statevectorとサンプルベース)
qi_sv = QuantumInstance(Aer.get_backend('aer_simulator_statevector'))

# 生成器の重みを更新するためにQNNを指定
gen_qnn = CircuitQNN(gen_disc_circuit, # パラメーター化回路
                      # 固定の入力パラ(識別器の重み)
                      gen_disc_circuit.parameters[:N_DPARAMS],
                      # 変更可能なパラ(生成器の重み)
                      gen_disc_circuit.parameters[N_DPARAMS:],
                      sparse=True, # 疎な確率ベクトルを返す
                      quantum_instance=qi_sv)

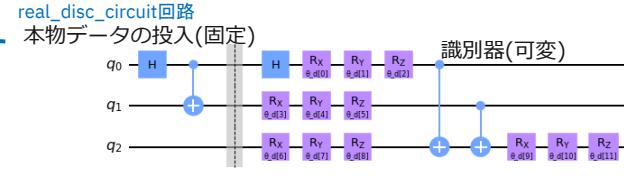
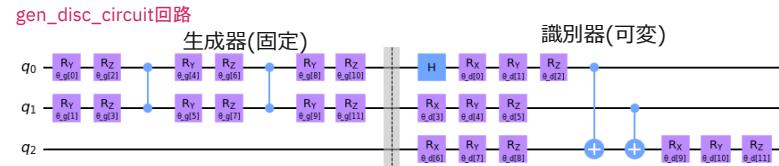
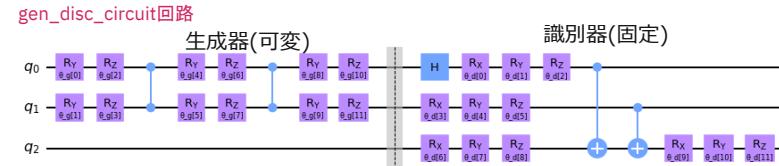
# 識別器の重みを更新するためにQNNを指定
disc_fake_qnn = CircuitQNN(gen_disc_circuit, # パラメーター化回路
                           # 固定の入力パラ(生成器の重み)
                           gen_disc_circuit.parameters[N_DPARAMS:],
                           # 変更可能なパラ(識別器の重み)
                           gen_disc_circuit.parameters[:N_DPARAMS],
                           sparse=True, # 疎な確率ベクトルを得る
                           quantum_instance=qi_sv)

disc_real_qnn = CircuitQNN(real_disc_circuit, # パラメーター化回路
                           # 入力パラメーターなし
                           # 変更可能なパラ(識別器の重み)
                           gen_disc_circuit.parameters[:N_DPARAMS],
                           sparse=True, # 疎な確率ベクトルを得る
                           quantum_instance=qi_sv)
```

コードの書き方 参照)

Qiskit Machine Learningモジュールのチュートリアル
「量子ニューラル・ネットワーク」：

https://qiskit.org/documentation/machine-learning/locale/ja_JP/tutorials/01_neural_networks.html



古典オプティマイザーで最適化し、パラメーター更新

機械学習ライブラリーTensorFlowの深層学習ライブラリーKerasのAdamオプティマイザーを使う。

Adamオプティマイザーは、Momentum法(運動量に基づく)の応用形で古典機械学習で広く使用されている。普通の勾配降下法(ランダムにパラメーターを更新)より優れていることが知られている。

```
# KerasからAdamオプティマイザーを初期化する
```

```
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.02)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.02)
```

初期パラメーターの設定もTensorFlow変数で

```
import tensorflow as tf
```

```
import pickle # 変数をシリアル化およびデシリアル化する
```

```
# パラメーターの初期化
```

```
init_gen_params = np.random.uniform(low=-np.pi, ← 亂数で初期パラメーターを設定
                                      high=np.pi,
                                      size=(N_GPARAMS,))
init_disc_params = np.random.uniform(low=-np.pi,
                                      high=np.pi,
                                      size=(N_DPARAMS,))
```

```
gen_params = tf.Variable(init_gen_params)
disc_params = tf.Variable(init_disc_params)
```

tf.Variable(変数)でTensorFlow形式に変換
.numpy()でnp.ndarrayに変換可能

← Kerasのオプティマイザーを使うために
TensorFlow変数に変換しておく

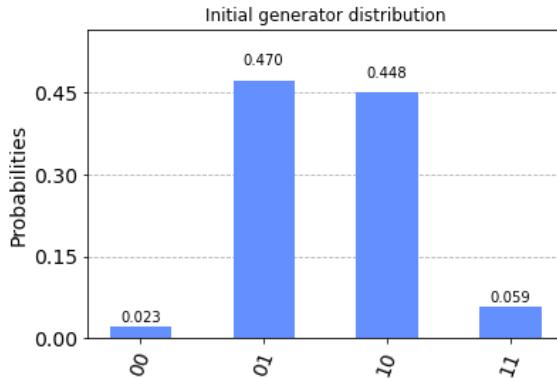
生成器の作成する初期状態分布

乱数で作成された生成器の初期出発点は、

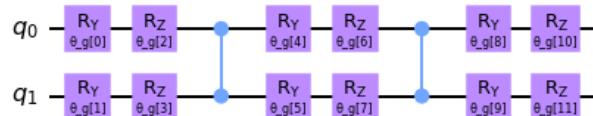
```
init_gen_circuit = generator.bind_parameters(init_gen_params)
init_prob_dict = Statevector(init_gen_circuit).probabilities_dict()

import matplotlib.pyplot as plt
fig, ax1 = plt.subplots(1, 1, sharey=True)
ax1.set_title("Initial generator distribution")
plot_histogram(init_prob_dict, ax=ax1)
```

(テキストブックとは別で) やってみた結果



generator回路



$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

を作りたいので

初期値としてはだいぶ異なる。

学習

ポイント

- 1回の生成器の更新に対して、識別器の重みは5回更新されるようとする。
古典GANでは、2つのネットワーク間の学習ステップ数が不均衡になることがあり、試行錯誤の結果、5:1の割合がベストプラクティスとされている。
- CircuitQNNのbackward()メソッドは、各バッチの各基底状態について、各重みに関する勾配を返す。
つまり、CircuitQNN.backward(...)[1].todense()から返される配列の型は
(バッチの数, 基底状態の数, パラメーターの数)
$$\text{Cost}_D = \Pr\left(D(\vec{\theta}_D, G(\vec{\theta}_G)) = |\text{real}\rangle\right) - \Pr\left(D(\vec{\theta}_D, R) = |\text{real}\rangle\right)$$

より、 Cost_D の勾配を求めるためには、 $|\text{1}\rangle = |\text{real}\rangle$ なので、まず $|\text{1}\square\square\rangle$ の基底について生成器側、識別器側のそれぞれにおいて、勾配を全て足し、次に関数 Cost_D の形にそって、その和を引く。
- GANの学習は不安定であるため、最適な生成器パラメーターを保存する。

学習：識別器

```
"""識別器パラメーターの更新"""
d_steps = 5 # N個の識別器の更新（生成器の更新ごと5倍更新） ← 1回の生成器の更新に対して、  
for disc_train_step in range(d_steps):  
    # 確率(fake/true)のθ_Dに対する偏微分をそれぞれ計算  
    d_fake = disc_fake_qnn.backward(gen_params, disc_params  
                                    )[1].todense()[0, 0b100:]  
    d_fake = np.sum(d_fake, axis=0) ← |1□□)の基底について生成器側、識別器側  
    d_real = disc_real_qnn.backward([], disc_params  
                                    )[1].todense()[0, 0b100:]  
    d_real = np.sum(d_real, axis=0) ← それれについて勾配を全て足す。  
  
    # Cost_Dの構造より  
    grad_dcst = [d_fake[i] - d_real[i] for i in range(N_DPARAMS)] ← 関数Cost_Dの形にそって、勾配の和を引く。  
    grad_dcst = tf.convert_to_tensor(grad_dcst)  
  
    # 識別器のパラメーターを勾配で更新する  
    discriminator_optimizer.apply_gradients(zip([grad_dcst],  
                                                [disc_params])) ← 勾配とパラメーターを指定して  
                                                Kerasオプティマイザーでパラメーター更新  
  
    # 識別器の損失を記録  
    if disc_train_step % d_steps == 0:  
        dloss.append(discriminator_cost(disc_params)) ← コストを記録
```

学習：生成器

"""生成器のパラメータ更新"""

```
for gen_train_step in range(1):
    #  $\theta_G$ に対する偏微分
    grads = gen_qnn.backward(disc_params, gen_params)
    grads = grads[1].todense()[0][0b100:]
    # Cost_Gの構造とその微分の線形性より
    grads = -np.sum(grads, axis=0)
    grads = tf.convert_to_tensor(grads)
```

|1□□)の基底について生成器の勾配を全て足して、
関数Cost_Gの形にする。

生成器のパラメーターを勾配で更新する

```
generator_optimizer.apply_gradients(zip([grads], [gen_params])) ← 勾配とパラメーターを指定して
gloss.append(generator_cost(gen_params)) ← Kerasオプティマイザーでパラメーター更新
```

コストを記録

カルバック・ライブラー(KL)情報量

P, Q を離散確率分布とするとき、 P の Q に対するカルバック・ライブラー (KL) 情報量は以下のように定義される。(wikipediaより)

$$D_{\text{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

KL情報量は、2つの分布間の距離を測定し、学習中の生成器の進捗を追跡するために使用される一般的なメトリックなので、

今回の生成器モデルの分布(Q)とターゲットとする分布(P)に対して、学習進捗を見るために使う。

KL情報量が低いほど、2つの分布が類似していることを示し、KL情報量が0であれば同等であることを意味する。

```
def calculate_KL(model_distribution: dict, target_distribution: dict):
    """Kullback Leibler Divergenceを用いたゲージモデルの性能評価"""
    KL = 0
    for bitstring, p_data in target_distribution.items():
        if np.isclose(p_data, 0, atol=1e-8):
            continue
        if bitstring in model_distribution.keys():
            KL += (p_data * np.log(p_data)
                   - p_data * np.log(model_distribution[bitstring]))
        else:
            KL += p_data * np.log(p_data) - p_data * np.log(1e-6)
    return KL
```

KL情報量で学習進度をモニター

```
# トレーニング中にメトリクスを追跡するための変数を初期化する
best_gen_params = tf.Variable(init_gen_params)
gloss = []
dloss = []
kl_div = []

"""KLを追跡して、最も性能の良い生成器の重みを保存する"""
# 生成器のパラメーターをupdateしてテスト回路を作る
gen_checkpoint_circuit = generator.bind_parameters(gen_params.numpy())
# 今の生成期の確率分布を取り出す
gen_prob_dict = Statevector(gen_checkpoint_circuit).probabilities_dict()
# 目的の回路の確率分布(定数)
real_prob_dict = Statevector(real_circuit).probabilities_dict()

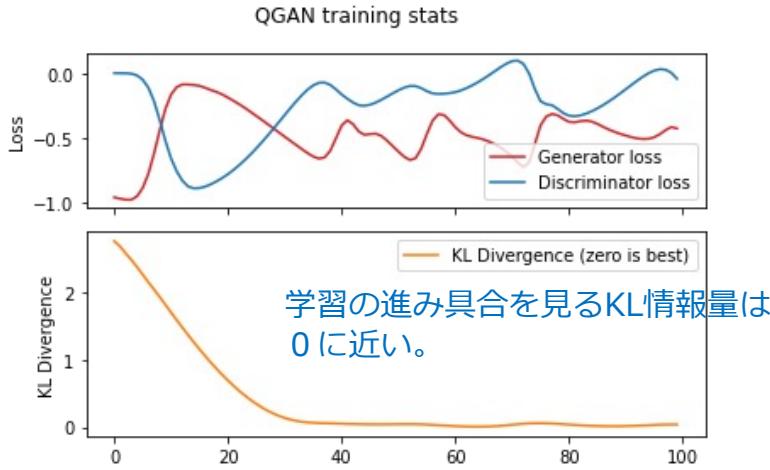
current_kl = calculate_KL(gen_prob_dict, real_prob_dict)
kl_div.append(current_kl)
if np.min(kl_div) == current_kl:
    # 新しいパラメーターの方が良い
    # serialize+deserializeでゼロになることを確認
    best_gen_params = pickle.loads(pickle.dumps(gen_params))
if epoch % 10 == 0:
    # 10エポックごとに表に表示
    for header, val in zip(table_headers.split('/'),
                           (epoch, gloss[-1], dloss[-1], kl_div[-1])):
        print(f'{val:.3g} {"rjust(len(header)+1)}', end="")
    print()
```

生成器のパラメーターを更新しつつ、
生成器モデルの分布(Q)と
本物の分布(P)(パラメーターなし)を使って
KL情報量を計算

GANの学習は不安定であるため、
KL情報量がよくなることを確認しながら、
生成器パラメーターを保存する。

Epoch / Generator cost / Discriminator cost / KL Div	0	0.000557	2.76
10	-0.163	-0.656	1.7
20	-0.187	-0.776	0.691
30	-0.489	-0.321	0.134
40	-0.403	-0.163	0.049
50	-0.603	-0.131	0.0419
60	-0.415	-0.146	0.0133
70	-0.643	0.0931	0.0331
80	-0.374	-0.327	0.0366
90	-0.48	-0.123	0.0142

学習結果

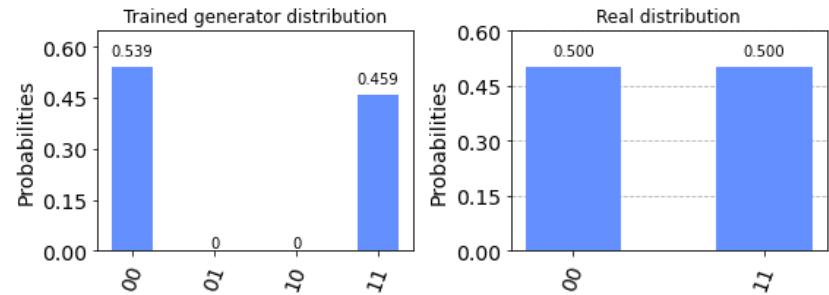


わずか100エポックで、生成器はベル状態 $|\psi\rangle$ をかなりよく近似。

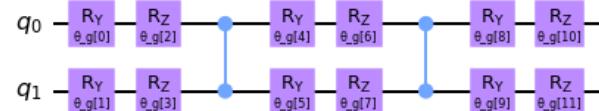
数回やってみないと結果が出ないことがある。と書いてあるが、確かに何回かやって見ると100エポックでうまく収束しない場合もあった。

GAN (QGANも含む) は、よく勾配が消失するため、識別器が優秀すぎることが原因であることが多い、とのこと。

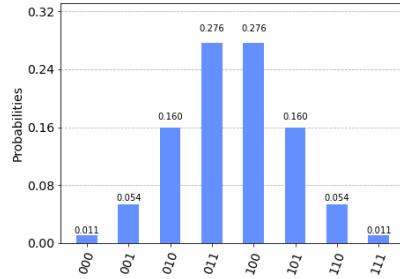
(テキストブックとは別で) やってみた結果



generator回路



実装その2: 3量子ビットの正規分布を再現



実装方針：先程と同じだが、勾配計算のみ異なる。

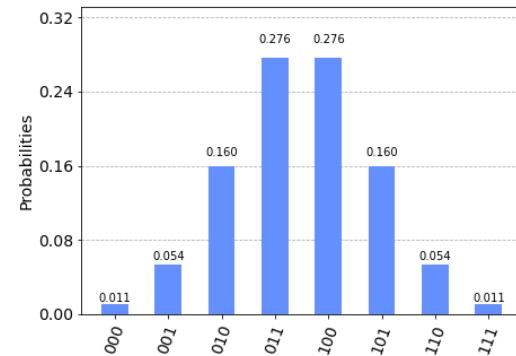
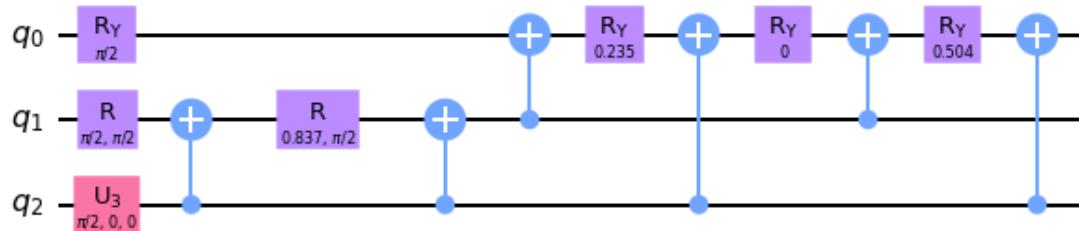
生成器と識別器のパラメーター化回路と勾配計算まで、Qiskit。最適化はTensorFlowで古典計算。

- 生成器と識別器のパラメーター化回路：Qiskitの`circuit.library`の`TwoLocal`と`ParameterVector`クラス
 - コスト関数の計算：パラメーター化回路をQiskit Statevectorでシミュレーション
 - 勾配計算：Qiskit Machine Learningモジュールの **OpflowQNN**
 - 最適化：TensorFlow(機械学習ライブラリー)のKeras(深層学習ライブラリー)のAdamオプティマイザー
- 参考：QiskitでQNNを実装する場合の方法は大きく2つ。
- CircuitQNN：測定による確率のサンプリング
 - OpflowQNN：オブザーバブルの期待値

本物の分布

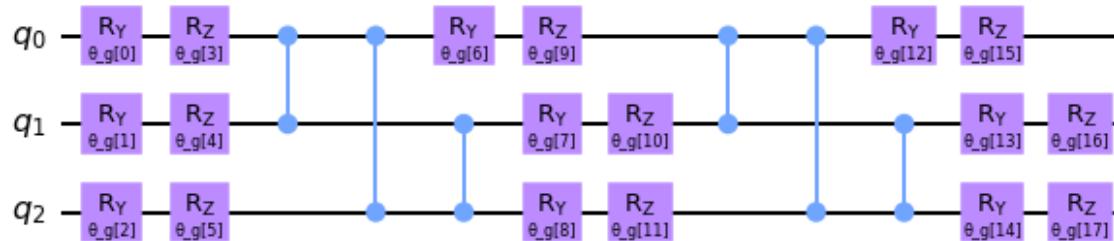
3量子ビットの正規分布をQiskit Financeモジュールで作る

```
from qiskit_finance.circuit.library import NormalDistribution  
  
REAL_DIST_NQUBITS = 3  
  
real_circuit = NormalDistribution(REAL_DIST_NQUBITS, mu=0, sigma=0.15)  
real_circuit = real_circuit.decompose().decompose().decompose()  
real_circuit.decompose().draw("mpl")
```

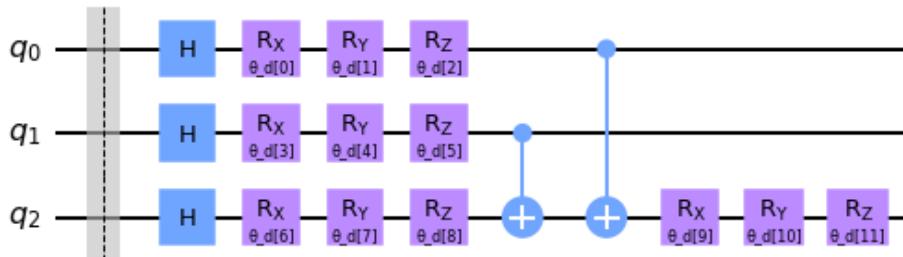


識別器と生成器の実装

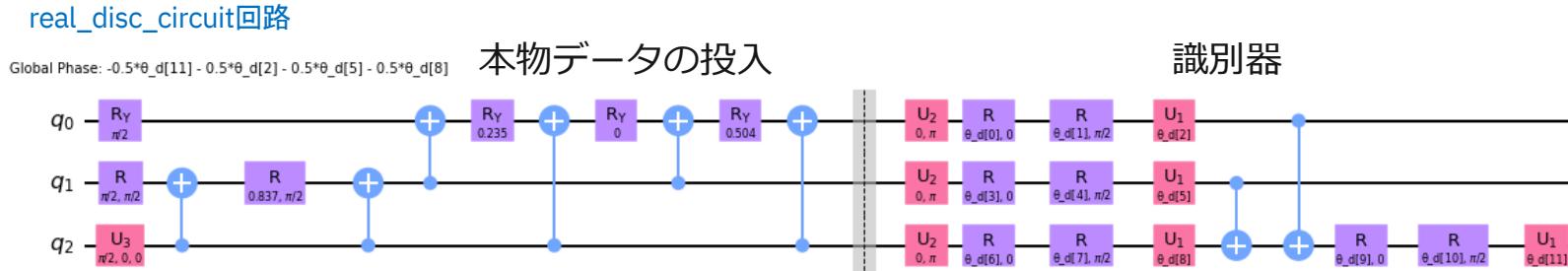
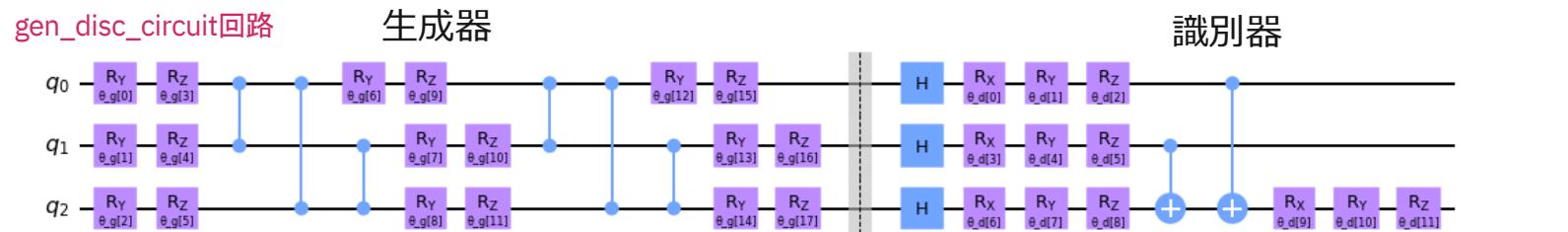
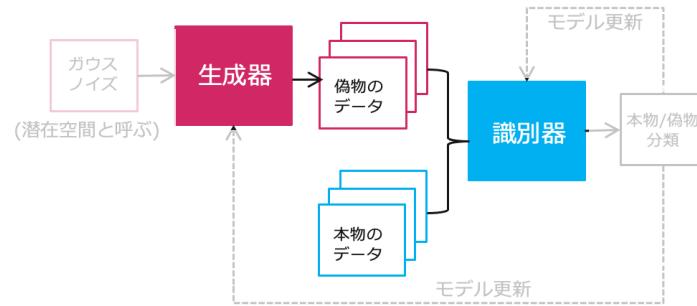
生成器：3量子ビットのTwoLocal回路（先程と同じ）



識別器：先程と同じParameterVectorでカスタムに作るが、今度は、3量子ビット回路で出力量子ビットを3量子ビット目にする。



2つのパラメーター化回路



OpflowQNNの引数の準備

Qiskit Machine Learningモジュールの OpflowQNNで、勾配計算を行う。
OpflowQNNは、パラメーター化回路を演算子にしたものを使って計算する。

```
from qiskit.providers.aer import QasmSimulator
from qiskit.opflow import (StateFn, PauliSumOp, ListOp,
                           Gradient, AerPauliExpectation)
from qiskit_machine_learning.neural_networks import OpflowQNN

# 期待値計算の方法を設定
expval = AerPauliExpectation()

# 勾配計算法を定義
gradient = Gradient()

# Quantum instanceをstatevectorで定義
qi_sv = QuantumInstance(Aer.get_backend('aer_simulator_statevector'))

# 回路の波動関数
gen_disc_sfn = StateFn(gen_disc_circuit)
real_disc_sfn = StateFn(real_disc_circuit)

# 最後の量子ビットのパウリZのexpvalを取り出すための演算子を構築
H1 = StateFn(PauliSumOp.from_list([('ZII', 1.0)]))

# 演算子と回路を合成し、目的の関数を得る
gendisc_op = ~H1 @ gen_disc_sfn #期待値 gendisc_op = <gen_disc_sfn|H1|gen_disc_sfn> を表す
realdisc_op = ~H1 @ real_disc_sfn #期待値 realdisc_op = <real_disc_sfn|H1|real_disc_sfn> を表す
```

コードの書き方 参照)

Qiskit Machine Learningモジュールのチュートリアル「量子ニューラル・ネットワーク」：
https://qiskit.org/documentation/machine-learning/locale/ja_JP/tutorials/01_neural_networks.html

OpflowQNNの設定

```
# OpflowQNNを期待値演算子、入力パラメーター、重みパラメーター、  
# 期待値計算の手法、勾配、量子インスタンスから構成する  
"""|fake> => |0> => 1 ; |real> => |1> => -1"""  
gen_opqnn = OpflowQNN(gendisc_op, # 期待値演算子  
                        gen_disc_circuit.parameters[:N_DPARAMS], # 固定の入力パラメーター(識別器の重み)  
                        gen_disc_circuit.parameters[N_DPARAMS:], # 変更可能な重み(生成器の重み)  
                        expval, # 期待値計算の手法  
                        gradient, # 勾配計算法  
                        qi_sv) # 生成器はこのexpvalを最小化したい  
  
disc_fake_opqnn = OpflowQNN(gendisc_op, # 期待値演算子  
                            gen_disc_circuit.parameters[N_DPARAMS:], # 固定の入力パラメーター(生成器の重み)  
                            gen_disc_circuit.parameters[:N_DPARAMS], # 変更可能な重み(識別器の重み)  
                            expval,  
                            gradient,  
                            qi_sv) # 識別器はこのexpvalを最大化したい  
  
disc_real_opqnn = OpflowQNN(realdisc_op, # 期待値演算子  
                           [], # 固定の入力パラメーターはない  
                           gen_disc_circuit.parameters[:N_DPARAMS], # 変更可能な重み(識別器の重み)  
                           expval,  
                           gradient,  
                           qi_sv) # 識別器はこのexpvalを最小化したい
```

生成器は自分のパラメーターを変えて
偽物の分布からの期待値が本物になるように
最小化する(後ほど設定)

識別器は自分のパラメーターを変えて
偽物の分布からの期待値は偽物なので
大きくなるようにする(後ほど設定)

識別器は自分のパラメーターを変えて
本物の分布からの期待値は本物なので
小さくなるようにする(後ほど設定)

コスト関数の再構築

OpflowQNNを使っているので、
量子回路を密度行列表現に直すと

識別器側：

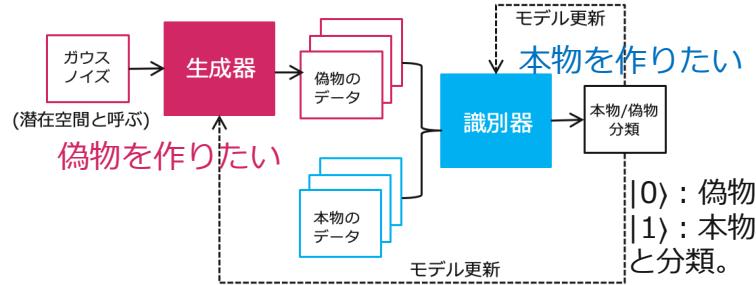
$$\rho^{DR} = U_{D(\vec{\theta}_D)} U_R |0\rangle^{\otimes n+1}$$

生成器側：

$$\rho^{DG} = U_{D(\vec{\theta}_D)} U_{G(\vec{\theta}_G)} |0\rangle^{\otimes n+1}$$

任意の密度行列 ρ に対する物理量 σ^P の期待値は $\text{tr}(\rho\sigma^P)$ 。

つまり、期待値計算は、量子回路の波動関数の密度行列と演算子をかけてTraceを取ることに相当する。



コスト関数の再構築

物理量Zの期待値は、 $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ からも分かるように

$\langle 0|Z|0\rangle = 1, \langle 1|Z|1\rangle = -1$ なので、

$|0\rangle$ が偽物、 $|1\rangle$ が本物とすると、

$\langle \text{fake}|Z|\text{fake}\rangle = 1, \langle \text{real}|Z|\text{real}\rangle = -1$

であることを考慮すると、

コスト関数は、最小値への最適化として、

生成器側：

$$\text{Cost}_G(\vec{\theta}_D, \vec{\theta}_G) = \text{tr}\left(Z \rho^{DG}(\vec{\theta}_D, \vec{\theta}_G)\right)$$

本物と同じ偽物を作りたいので

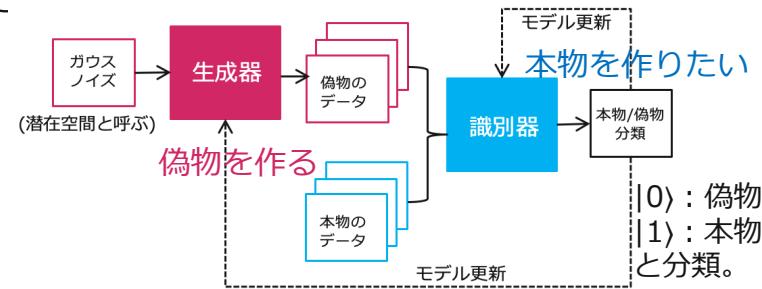
生成器側からの期待値は小さくしたい。

識別器側：

$$\text{Cost}_D(\vec{\theta}_D, \vec{\theta}_G) = \text{tr}\left(Z \rho^{DR}(\vec{\theta}_D)\right) - \text{tr}\left(Z \rho^{DG}(\vec{\theta}_D, \vec{\theta}_G)\right)$$

本物からの分布の
期待値は小さく。

生成器側からの期待値は偽物と判定したいので
期待値は大きくしたい。



初期値の設定とオプティマイザーの設定

先程と同じ設定

```
### START
init_gen_params = tf.Variable(np.random.uniform(low=-np.pi,
                                                high=np.pi,
                                                size=(N_GPARAMS)))
init_disc_params = tf.Variable(np.random.uniform(low=-np.pi,
                                                high=np.pi,
                                                size=(N_DPARAMS)))
gen_params = init_gen_params
disc_params = init_disc_params

generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.02)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.02)
```

TensorFlowのKerasのAdamオプティマイザー

学習

"""量子識別器のパラメーターの更新""""

```
for disc_train_step in range(d_steps):
    grad_dcost_fake = disc_fake_opqnn.backward(gen_params,
                                                disc_params)[1][0,0]
    grad_dcost_real = disc_real_opqnn.backward([], 
                                                disc_params)[1][0,0]
    grad_dcost = grad_dcost_real - grad_dcost_fake # as formulated above
    grad_dcost = tf.convert_to_tensor(grad_dcost)
```

update disc_params

```
discriminator_optimizer.apply_gradients(zip([grad_dcost],
                                            [disc_params]))
```

if disc_train_step % d_steps == 0:

```
dloss.append(discriminator_cost(disc_params))
```

勾配を計算

disc_params)[1][0,0]

disc_params)[1][0,0]

grad_dcost = grad_dcost_real - grad_dcost_fake # as formulated above

grad_dcost = tf.convert_to_tensor(grad_dcost)

Cost_Dの形より

$$\text{Cost}_D(\vec{\theta}_D, \vec{\theta}_G) = \text{tr}\left(Z\rho^{DR}(\vec{\theta}_D)\right) - \text{tr}\left(Z\rho^{DG}(\vec{\theta}_D, \vec{\theta}_G)\right)$$

Kerasオプティマイザーで
パラメーター更新

"""量子生成器のパラメーターの更新""""

```
for gen_train_step in range(1):
    # as formulated above
```

```
grad_gcost = gen_opqnn.backward(disc_params,
                                 gen_params)[1][0,0]
```

```
grad_gcost = tf.convert_to_tensor(grad_gcost)
```

update gen_params

```
generator_optimizer.apply_gradients(zip([grad_gcost],
                                         [gen_params]))
```

```
gloss.append(generator_cost(gen_params))
```

勾配を計算

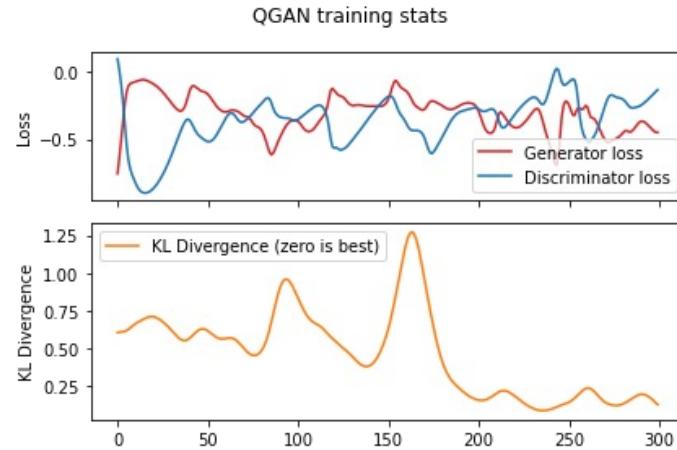
Cost_Gの形よりそのまま

$$\text{Cost}_G(\vec{\theta}_D, \vec{\theta}_G) = \text{tr}\left(Z\rho^{DG}(\vec{\theta}_D, \vec{\theta}_G)\right)$$

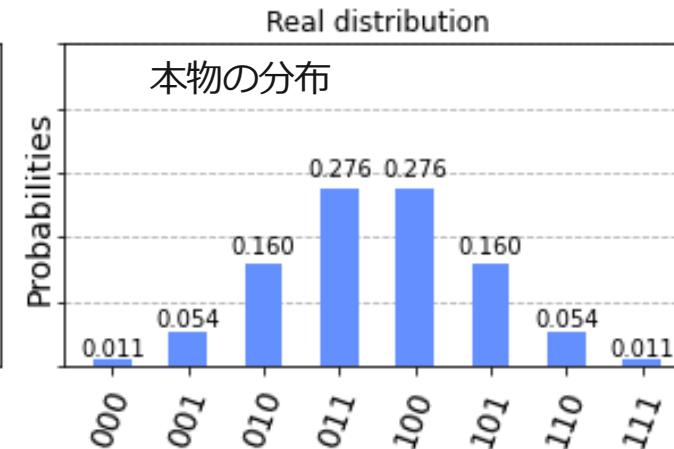
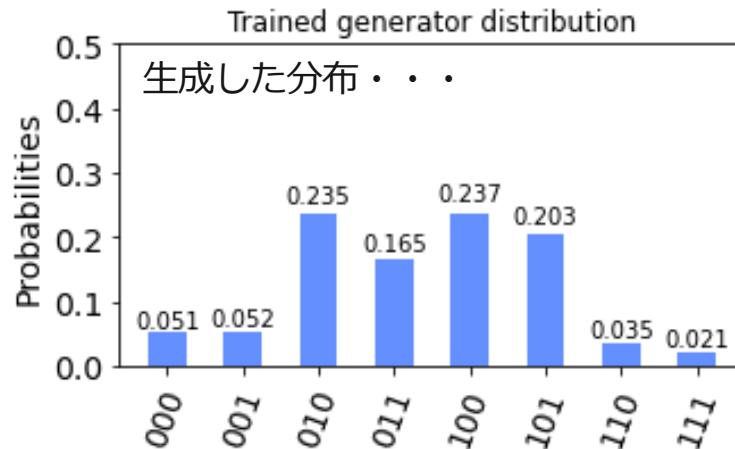
Kerasオプティマイザーでパラメーター更新

gloss.append(generator_cost(gen_params))

実行結果 (テキストブックとは別で) やってみた結果



学習の進み具合を見るKL情報量は
0に近づいてはいる。



QGANの今後の可能性

QGANの開発はまだ始まったばかりであり、その応用についてはまだまだ研究中。

古典的に作成が困難な確率分布のサンプリングや操作を可能にすることが期待。

例) 量子振幅推定におけるデータの準備

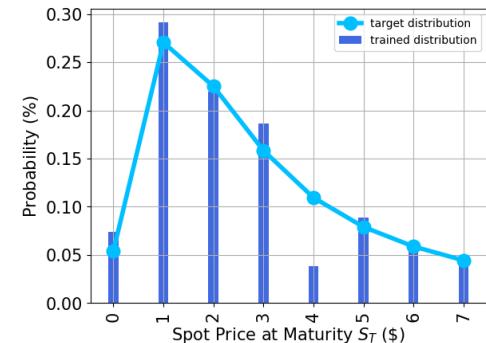
古典データを量子回路に準備するためのコストは高い（CNOTが指数関数的に増える！）ため、現状、アルゴリズムの量子優位性を生かすのは難しい。

QGANでは、確率分布の近似値を効率的に学習できるので、期待大。

- Qiskitチュートリアル：[qGANsによるオプション価格推定](#)
- Quantum Tokyo：[録画](#)、[資料](#)

例) 量子化学のシミュレーション

条件付きQGAN（条件付きラベルを生成器と識別器の両方に入力する）をVQEで使う方法が研究されている。



プロジェクト



ステップ1：コミュニティとつながる (Qiskit Slack [#textbook-projects](#)チャネル)

ステップ2：GitHubを使い始める

ステップ3：プロジェクトを決定する

- ブログ
- 実装したコードの公開
- 調査結果を公開
 - 各ansatz(ZZFeatureMap、 TwoLocal)をコストで比較。
 - 古典機械学習を分析し、量子回路を使用することでメリットが得られる可能性を見つけよう。
 - QGANの拡張。[\(論文リンク\)](#)
 - ansatzの表現能力や勾配の大きさ、不毛な台地について調べる。[\(論文リンク\)](#)
 - 他のデータ符号化の手法を調べる。[\(論文リンク\)](#)
 - 学んだ手法を量子データへ適用する。利点および、または制限は何ですか？[\(データリンクなど\)](#)

ステップ4：楽みましょう！ ([#textbook-projects](#)チャネルで助けを求める)

ステップ5：取り組みを共有しましょう

