Training Variational GBS Distributions

Status | Done

Introduction

- Many quantum algorithms rely on the ability to train the parameters of quantum circuits.
- Training is often performed by evaluating gradients of a cost function with respect to circuit parameters, then employing gradient-based optimization methods.
- The probability of observing the output $S = (s_1, s_2, ..., s_m)$, $where\ s_i$ is the number of photons detected at the i-th mode, is given by:

$$Pr(S) = \frac{1}{\mathcal{N}} \frac{|Haf(A_s)|^2}{s_1!...s_m!}$$

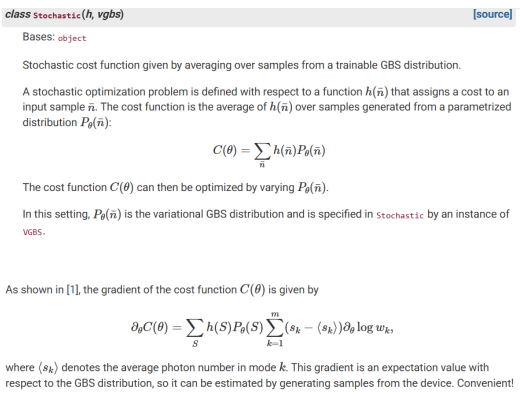
- A is an arbitrary symmetric matrix with eigenvalues bounded between -1 and 1. The matrix can also be rescaled by a constant factor, which is equivalent to fixing a total mean photon number in the distribution.
- We want to train this distribution to perform a specific task. Example: Reproduce the statistical properties of a dataset, or optimize a circuit to specific pattern with high probability.
- So, we want to find a parametrization (assignment of parameters) of the distribution (probability of GBS samples) and optimize using gradient based techniques. - Called Variational GBS (VGBS)
- But, gradients can be challenging to compute. So we do the following to A. (WAW parameterization) - $A \rightarrow A_W = WAW$ (W is a diagonal matrix)
  - This is easier because we have the following factorization.

    $Haf(A_W) = Haf(A)det(W)$
  - Also we can embed trainable parameter $\theta = (\theta_1, ..., \theta_d)$ into the diagonal elements of W. (Ex: an exponential embedding)

We look at the following 2 specific training tasks.

Stochastic Optimization

- Different from deterministic optimization, stochastic optimization algorithms employ processes with random factors to do so.
- Due to these processes with random factors, stochastic optimization does not guarantee finding the optimal result for a given problem. But, there is always a probability of finding the globally optimal result.
- Specifically, the probability of finding the globally optimal result in stochastic optimization relates to the available computing time. The probability of finding the globally optimal result increases as the execution time increases.



class `Stochastic`(*h*, *vgbs*) [source]

Bases: `object`

Stochastic cost function given by averaging over samples from a trainable GBS distribution.

A stochastic optimization problem is defined with respect to a function $h(\bar{n})$ that assigns a cost to an input sample $\bar{n}$. The cost function is the average of $h(\bar{n})$ over samples generated from a parametrized distribution $P_\theta(\bar{n})$:

$$C(\theta) = \sum_{\bar{n}} h(\bar{n}) P_\theta(\bar{n})$$

The cost function $C(\theta)$ can then be optimized by varying $P_\theta(\bar{n})$.

In this setting, $P_\theta(\bar{n})$ is the variational GBS distribution and is specified in `Stochastic` by an instance of `VGBS`.

As shown in [1], the gradient of the cost function $C(\theta)$ is given by

$$\partial_\theta C(\theta) = \sum_S h(S) P_\theta(S) \sum_{k=1}^{m} (s_k - \langle s_k \rangle) \partial_\theta \log w_k,$$

where $\langle s_k \rangle$ denotes the average photon number in mode k. This gradient is an expectation value with respect to the GBS distribution, so it can be estimated by generating samples from the device. Convenient!

The code

💡 **Goal:** We explore a basic example where the goal is to optimize the distribution to favour photons appearing in a specific subset of modes, while minimizing the number of photons in the remaining modes.

```
#Cost function
#The function is defined with respect to the subset of modes for which we want to observe many photons.
#This subset can be specified later on.
#For some sample s, we want to maximize the total number of photons in the subset.
def h(s):
    not_subset = [k for k in range(len(s)) if k not in subset]
    return sum(s[not_subset]) - sum(s[subset])

cost = train.Stochastic(h, vgbs)

#Defining the VGBS for a Lolipop graph
#The parameters are the WAW parametrization and the total mean photon number
#We just use A to be the adjacency of the graph
#GBS devices can operate either with photon number-resolving detectors or threshold detectors, so there is an opt
ion to specify which one we intend to use. We'll stick to detectors that can count photons.
#Note that VGBS is a class, so this is just declaring an object

n_mean = 6
vgbs = train.VGBS(A, n_mean, weights, threshold=False)
```

During training, we'll calculate gradients and evaluate the average of this cost function with respect to the GBS distribution. Both of these actions require estimating expectation values, so the number of samples in the estimation also needs to be specified. The parameters also need to be initialized. There is freedom in this choice, but here we'll set them all to zero. Finally, we'll aim to increase the number of photons in the "candy" part of the lollipop graph, which corresponds to the subset of modes `[0, 1, 2]`.

```
np.random.seed(1969)  # for reproducibility
d = nr_modes
params = np.zeros(d)
subset = [0, 1, 2]
nr_samples = 100

print('Initial mean photon numbers = ', vgbs.mean_photons_by_mode(params))

#We perform training over 200 steps of gradient descent with a learning rate of 0.01

nr_steps = 200
rate = 0.01

for i in range(nr_steps):
    params -= rate * cost.grad(params, nr_samples)
    if i % 50 == 0:
        print('Cost = {:.3f}'.format(cost.evaluate(params, nr_samples)))

print('Final mean photon numbers = ', vgbs.mean_photons_by_mode(params))

'''
Output of above code

Cost = -1.685
```

```
Cost = -3.892
Cost = -4.516
Cost = -4.677
Final mean photon numbers =  [3.1404 2.7654 1.1547 0.0931 0.0429]
'''
```

Note: The transformed WAW matrix needs to have eigenvalues bounded between -1 and 1, so continuing training indefinitely can lead to unphysical distributions when the weights become too large. It's important to monitor this behavior.

We can see that the parameters now work well.

```
Aw = vgbs.A(params)
samples = vgbs.generate_samples(Aw, n_samples=10)
print(samples)
```

Out:

```
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [1 1 0 1 1]
 [0 0 0 0 0]
 [3 1 2 0 0]
 [2 2 2 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [4 4 0 0 0]]
```

Unsupervised Learning

- Data are assumed to be sampled from an unknown distribution and a common goal is to learn that distribution.
- That is the goal is to use the data to train a model that can sample from a similar distribution, thus being able to generate new data.
- Training can be performed by minimizing the Kullback-Leibler (KL) divergence.

$$KL(\theta) = -\frac{1}{T}\sum_S \log[P_\theta(S)].$$

In this case S is an element of the data, $P(S)$ is the probability of observing that element when sampling from the GBS distribution, and T is the total number of elements in the data. For the GBS distribution in the WAW parametrization, the gradient of the KL divergence can be written as

$$\partial_\theta KL(\theta) = -\sum_{k=1}^{m} \frac{1}{w_k}(\langle s_k\rangle_{\text{data}} - \langle s_k\rangle_{\text{GBS}})\partial_\theta w_k.$$

- For unsupervised learning with VGBS (using KL divergence and WAW parametrization), we can efficiently calculate the *gradient of the cost function* using a classical computer. This makes the *training process* very fast and practical, even though the final goal is to leverage the quantum computer's unique ability to generate samples from a distribution that is otherwise classically intractable.

The code

```
#Pre-generated data

from strawberryfields.apps import data

data_pl = data.Planted()
data_samples = data_pl[:1000]  # we use only the first one thousand samples
A = data_pl.adj
nr_modes = len(A)

#We define the VGBS circuit in threshold mode, since this is how the data samples were generated.
#The cost function is the Kullback-Liebler divergence, which depends on the data samples and can be accessed u
sing KL
weights = train.Exp(nr_modes)
n_mean = 1
vgbs = train.VGBS(A, n_mean, weights, threshold=True)
```

```
cost = train.KL(data_samples, vgbs)

#We will look at the difference in mean photons cause the cost function is exp to calculate.

from numpy.linalg import norm

params = np.zeros(nr_modes)
steps = 1000
rate = 0.15

for i in range(steps):
    params -= rate * cost.grad(params)
    if i % 100 == 0:
        diff = cost.mean_n_data - vgbs.mean_clicks_by_mode(params)
        print('Norm of difference = {:.5f}'.format(norm(diff)))

'''
Output
Norm of difference = 0.90856
Norm of difference = 0.01818
Norm of difference = 0.00532
Norm of difference = 0.00177
Norm of difference = 0.00062
Norm of difference = 0.00022
Norm of difference = 0.00008
Norm of difference = 0.00003
Norm of difference = 0.00001
Norm of difference = 0.00000
'''
```

References

1. https://strawberryfields.ai/photonics/apps/run_tutorial_training.html#banchi2020training
2. https://strawberryfields.readthedocs.io/en/stable/code/api/api/strawberryfields.apps.train.VGBS.html#strawberryfie
   (There's more info in Methods)
3. https://strawberryfields.readthedocs.io/en/stable/code/api/api/strawberryfields.apps.data.Planted.html#strawberryf
4. https://pennylane.ai/qml/glossary/variational_circuit
5. https://www.baeldung.com/cs/deterministic-stochastic-optimization#:~:text=If%20the%20globally%20optimal%20result,solution%20in%20a%20given%20time