Review of QAOA

Quantum Approximate Optimization for constrained optimization:

Find $z \in \{0, 1\}^n$ to minimize: $C(z) = \sum_i C_i(z)$ where

$C_i(z) = \begin{cases} 1 & \text{if } z \text{ satisfies constraint } i \\ 0 & \text{otherwise} \end{cases}$

Ex. MAXCUT is an instance of a constrained optimization problem.

Define a graph $G$ s.t. $i \in G$ is a vertex of the graph and $(i,j) \in E(G)$ is an edge from vertex $i$ to $j$ in the graph.

Let $C(z) = \langle z | \hat{Z} | z \rangle$ where $\hat{Z} = \frac{1}{2} (\mathbb{1} - Z_i Z_j)$

the "Cost Hamiltonian"

QAOA Algorithm

1. Pick a "driver Hamiltonian" whose ground state is easy to prepare on your quantum computer. E.g.

   $\hat{H}_D = \sum_{i \in G} H_i$ which has ground state $|\psi_D\rangle = \frac{1}{\sqrt{Z^{1/2}}} \sum_{z \in \{0, 1\}} |z\rangle$

2. Construct parameterized unitaries corresponding to the cost and driver Hamiltonians

   $U_D(\beta) = e^{-i\beta \hat{H}_D}$  \hspace{1cm}  $U_C(\gamma) = e^{-i\gamma \hat{Z}}$

3. Apply sequentially series of these unitaries to form the QAOA ansatz $U(\beta, \gamma)$ given by

   \begin{align*}
   &U_D^{(m)}(\beta_m) \quad U_C^{(m)}(\gamma_m) \quad \ldots \quad U_D^{(1)}(\beta_1) \quad U_C^{(1)}(\gamma_1) \quad \ldots \quad U_D^{(0)}(\beta_0) \quad U_C^{(0)}(\gamma_0) \\
   &\text{prepare ground state } |\psi_0\rangle \\
   &\text{one layer of QAOA} \quad \text{Repeat for p layers} \\
   &\text{measure all} 
   \end{align*}
4) Sample a set of bitstrings $\mathbf{Z}(\bar{B}, \bar{y})$ from running $U(\bar{B}, \bar{y})$. Calculate the score as the expected value

$$E(\bar{B}, \bar{y}) = \frac{1}{1^2(B, y)} \sum_{a \in 2(B, y)} \langle a | 2a \rangle$$

5) Run this in a minimization loop to choose a new $\bar{B'}, \bar{y'}$ to decrease $E(\bar{B}, \bar{y})$. Then repeat from #3.

* Note it is also a good idea to store the “best” bitstring $\mathbf{Z}$ that has been sampled so far.

6) Return the best bitstring after some number of iterations.

Implementing exponentiations of Paulis

In order to program this ansatz we need a way to turn the cost and driver Hamiltonians into quantum gates from our gate set. Both $\hat{H}_0$ and $\hat{Z}$ are given as sums of products of Pauli matrices.

If we can exponentiate sums of products of Paulis, then we can exponentiate any Hermitian matrix. We will show a technique to do this.

**Case (i)** $\hat{H} = \hat{Z}_j$ then $U = e^{-i\alpha \hat{Z}_j} = \prod_{i \neq j} e^{i \alpha \hat{Z}_i} = \prod_{i \neq j} e^{i \alpha \hat{Z}_i} = \prod_{i \neq j} e^{i \alpha \hat{Z}_i}$

The exponent of a diagonal matrix is easy.

**Case (ii)** $\hat{H} = \alpha \prod_i \hat{Z}_i$ some (all) products of $\hat{Z}_i$. We saw when studying error correction that this operator labels the parity of the qubits. Thus:

$$U = \begin{cases} \text{for } i \text{ in enumerate}(\text{MLI-13}): \\ \text{CNOT}(\text{MLI-13, MLI+1}) \\ \text{RZ}(\alpha) \text{ MLI-13} \end{cases} \text{ for } i \text{ in enumerate}[\text{reverse}(\text{MLI-13})]: \\ \text{CNOT (MLI-13, MLI+1})$$

Ex: $Z_0 Z_1 Z_2$

$$\text{RZ}(2\alpha)$$
Case (iii)

\[ H = \prod_{j \in M} P_j \text{ where } P_j \text{ is one of the Pauli matrices.} \]

Here we know that we can transform any Pauli into \( Z \) via unitary e.g.

\[ Z = H \times H \]

And vice versa any \( Z \) can be turned into another Pauli, e.g.

\[ HZH = X \]

Thus we reduce to case (ii) and apply pre and post rotations to sub j

\( \neq \) \( P_j \neq Z \)

Case (iv)

\[ H = H_1 + H_2 \]

This is where we can apply the Trotter formula from the previous lecture. When \( H_1 \) and \( H_2 \) commute this is trivial. We just do \( e^{iH_1} \) then \( e^{iH_2} \).

When they do not commute we use the Trotter-Suzuki formulas to approximate them.

\[
\left( e^{iH_1+H_2} \right) = \left( e^{iH_1/m} e^{iH_2/m} \right)^m
\]

These cases taken together suffice for any Hamiltonian \( H \) as it can be written as a sum of products of Pauli's.

Let's look at how this is implemented in Python.

*Notebook*
Intro to Quantum Machine Learning

A binary quantum classifier (Havlicek 2018)

Define Classification Problem

Let training set \( T \) and test set \( S \) be sets of vectors \( \vec{x}, \vec{s} \in \mathbb{R}^d \)

Assume a labeling of both given by

\[
m : T \cup S \rightarrow \{+1, -1\}
\]

The training algorithm is only given the restriction of \( m \) to \( T \)

\[
m_T : T \rightarrow \{+1, -1\}
\]

\[
\vec{x} \rightarrow m_T(\vec{x})
\]

e.g. the label/ys of the training se\( T \).

The GOAL is to infer a map on the test set \( S \),

\[
m : S \rightarrow \{+1, -1\}
\]

That has the highest probability of agreement with \( m_T \), i.e. \( m \) on the test data.

For example this could be represented by an objective function

\[
\mathcal{O}(m) = \frac{1}{|S|} \sum_{\vec{s} \in S} \frac{1}{2} (1 - m(\vec{s}) m(\vec{s}))
\]

which is to be minimized.

Support Vector Machines

Are a classical approach to this problem. Find the separating "hyper-plane" that gives the best score. For example \( d = 2 \)

\[
\begin{array}{c}
\text{Simple example of an SVM} \\
\text{where we choose some optimal} \\
\text{line } \vec{w} \cdot \vec{x} + b = 0
\end{array}
\]
However not all data can be split into a nice linear separation. For example:

Has no linear separator. However if we do a "kernel" transformation and map our data into a higher dimensional space we can find a separating hyperplane.

The idea behind many Quantum SVM's is to use the large vector space on a GPU to act as the kernel in a large vector space. This means we can compute new non-linear classifications.

There are many algorithm variants to do this on a GPU. Today we will cover only one, Quantum Variational Classification.
Quantum Variational Classifier

Steps:
1. Encode classical vectors from $\mathbb{R}^d$ into a wavefunction on $N$ qubits e.g. a vector in $\mathbb{C}^n$.

2. Train QPU operations (unitary and measurements) to classify the wavefunction into labels $\in \mathbb{E} \times \mathbb{I}$.

$$U_\Phi(z) \xrightarrow{W(\theta)} f(z) \in \mathbb{E} \times \mathbb{I}$$

AND a choice of cost function and classical optimizer to find $\Theta^*$.

Feature encoding:
We want to do something that can show quantum advantage. Thus, we should choose feature encodings that are hard to do classically. We want $U_\Phi : T \xrightarrow{U_\Phi} C^n$ where $U_\Phi(x)$ is a diagonal unitary.
This is a good structure because calculating the inner product between wave functions generated by such circuits is \#P-hard \footnote{Goldberg & Cygan 2017}.

\[
k(\vec{x}, \vec{y}) = |\langle \Phi(\vec{x}) | \Phi(\vec{y}) \rangle|^2 \text{ is \#P-hard for this } U_{\Phi}(x)
\]

\textbf{Ex.} Let } U_{\Phi}(x) \text{ be some circuit:

\[
\begin{array}{l}
\text{for } q \text{ in qubits:} \\
\text{RZ}(\vec{\phi}_q^c) q
\end{array}
\]

\[
\begin{array}{l}
\text{for } q \text{ in qubits:} \\
\text{CPHASE}(\vec{\phi}_{q,t}) q, q_{\text{next}(t)}
\end{array}
\]

This moves us into a high dimensional space that is hard for classical computers to work in.

\textbf{Quantum Variational Classification}

Here we choose layers like in a VQE hardware ansatz:

\[
W(\vec{\theta}) = U_{\text{loc}}^{(d)}(\vec{\theta}_d) U_{\text{ent}} \cdots U_{\text{loc}}^{(2)}(\vec{\theta}_2) U_{\text{ent}} U_{\text{loc}}^{(1)}(\vec{\theta}_1)
\]

\[
U_{\text{loc}}^{(t)}(\vec{\theta}_t) = \text{ for } q \text{ in qubits:} \\
\text{RZ}(\vec{\phi}_{q,t}^z) q \\
\text{RY}(\vec{\phi}_{q,t}^x) q
\]

\[
U_{\text{ent}} = \text{ for } q \text{ in qubits:} \\
\text{CZ } q, q_{\text{next}(t)}
\]

\textit{OR} on hardware this can be the connectivity graph

With enough layers this is a universal quantum circuit.
For any \( x \in \text{Trus} \) we can now run this to obtain

\[
\mathcal{W}(\tilde{\Theta})|\Phi(x)\rangle
\]

Measure all qubits to obtain bitstrings \( z_r \in \{0,1\}^N \) for \( R \) repetitions. We then apply a labeling function \( \epsilon \).

\[
f(z_r) = \begin{cases} 
+1 & \text{if } \frac{\sum z_r}{R} > N/2 \\
-1 & \text{otherwise}
\end{cases}
\]

i.e. take the majority vote of the qubit outcomes.

For some \# of repetitions \( R \) let \( r_+ = \text{count of } +1 \text{ labels} \)

\[
R = R - r_+
\]

We then take the majority vote of the labels over the \( R \) repetitions to get the classification for \( x \).

Note a meta-parameter \( \text{bias} \) can be introduced here for the final classification e.g.

\[
x \mapsto \begin{cases} 
+1 & \text{if } r_+ > R - b \\
-1 & \text{otherwise}
\end{cases}
\]

Thus we have

\[
\text{LABEL}(\tilde{\Theta}, x) \rightarrow \mathbb{R} \cup \{1, -1\}
\]

Train \( \tilde{\Theta} \) by labeling over the training set.

Then test by running over the test set.

Show good kernels

\[
\Theta(e^{-Vx})
\]