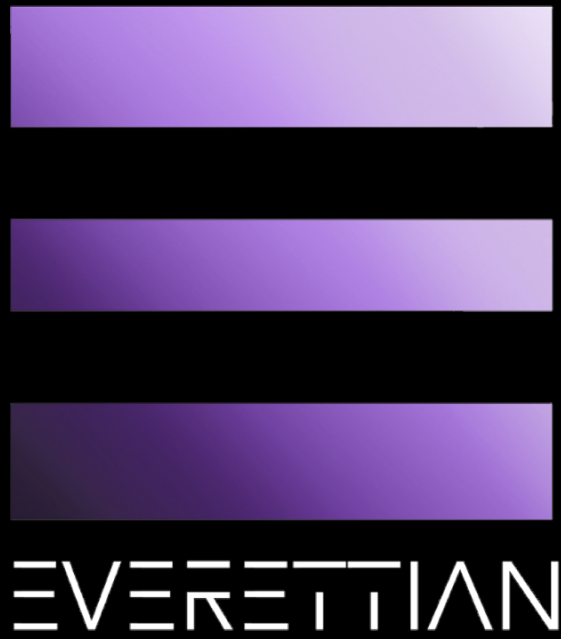


Training Neural Networks using Low-Depth Quantum Circuits

Guillaume Verdon

Co-Founder & Chief Scientific Officer

QC Meetup @YC

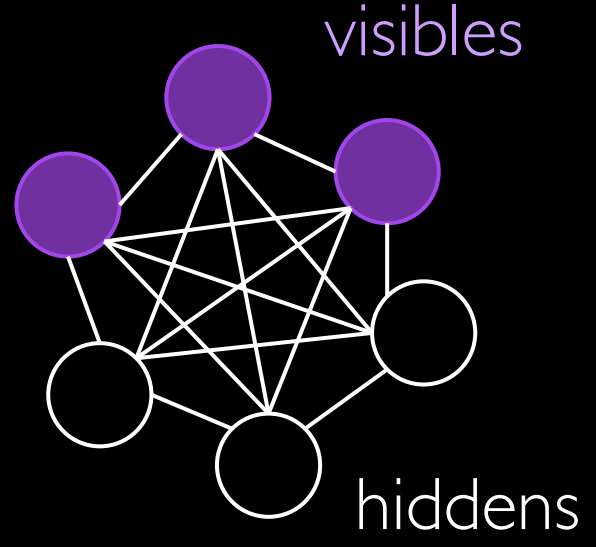
guillaume@everettian.com

Overview

Classical Boltzmann machines

- Energy-based model

$$E(z) = -\sum_{\langle jk \rangle} J_{jk} z_j z_k - \sum_j B_j z_j$$



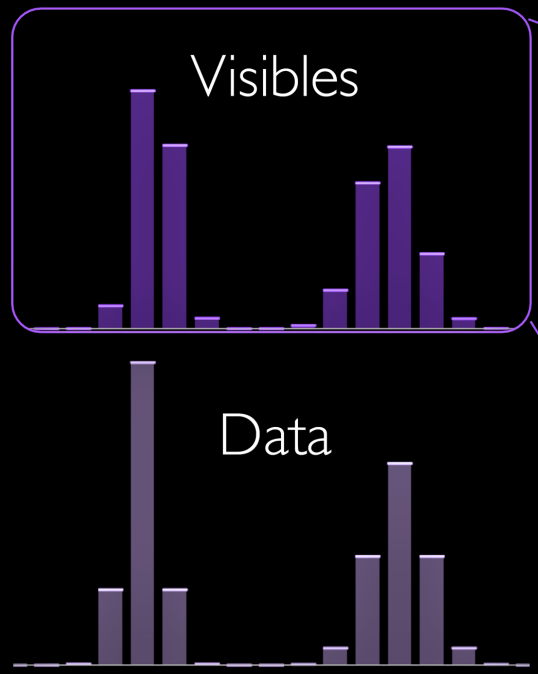
- Use thermal equilibrium statistics to mimic data

$$P(z) = \frac{e^{-\beta E(z)}}{\sum_z e^{-\beta E(z)}}$$

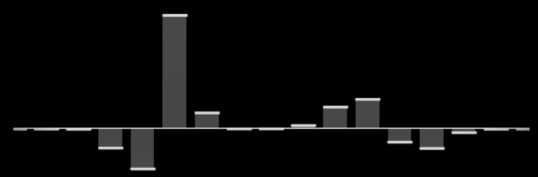
Classical Boltzmann machines

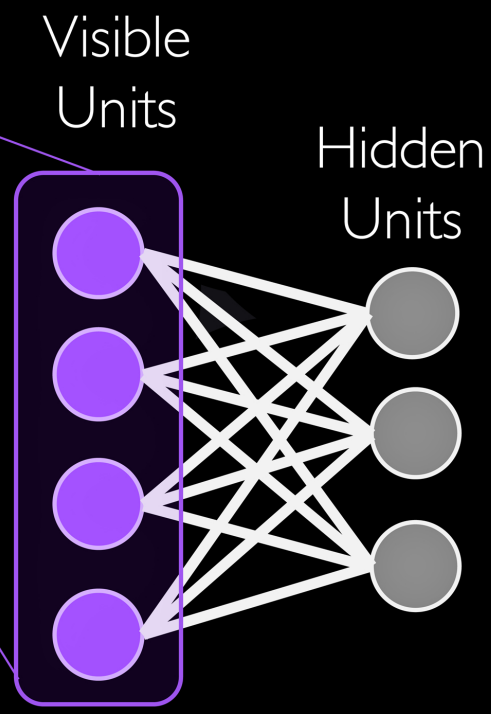
- Goal: reduce visibles' relative entropy to data

$$\underbrace{\sum_{x \in \text{data}} P_{\text{data}}(x) \log \left(\frac{P_{\text{vis}}(x)}{P_{\text{data}}(x)} \right)}_{D_{\text{KL}}(P_{\text{data}} || P_{\text{vis}})}$$



Relative Entropy





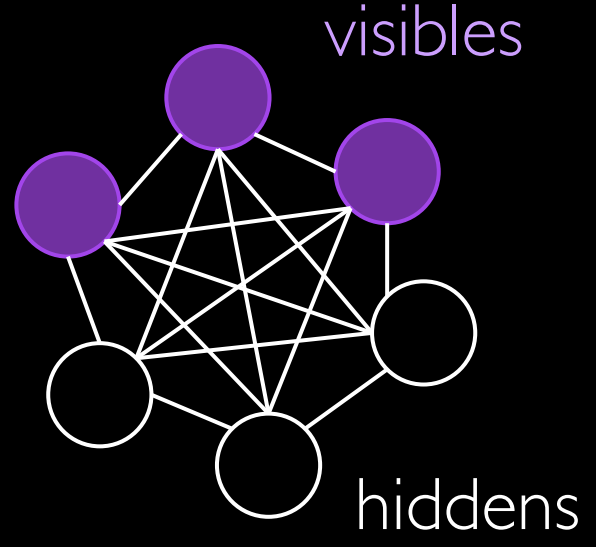
Restricted Boltzmann Machine (RBM)

- Need to sample from thermal (Gibbs) distribution for training
  - Use Monte Carlo techniques
  - Or... quantum computers!

Quantum Boltzmann machines

- Energy function → Hamiltonian operator

$$\hat{H} = -\sum_{\langle jk \rangle} J_{jk} \hat{Z}_j \hat{Z}_k - \sum_j B_j \hat{Z}_j$$



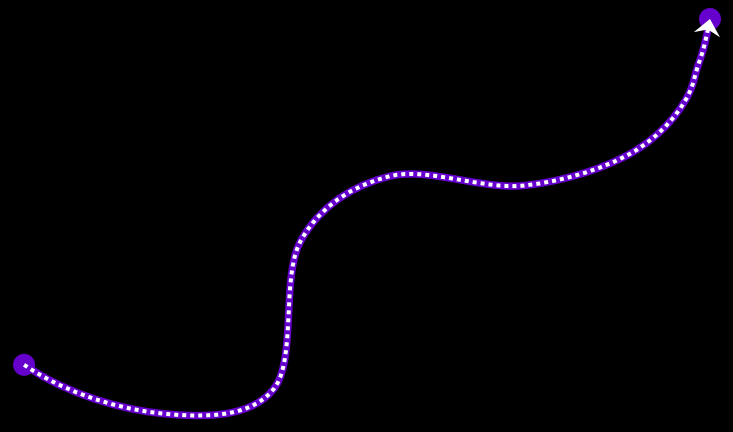
- For Gibbs sampling: need to simulate quantum thermalization
  - Quantum Analog
  - Quantum Digital
  - Quantum Approximate

$$\mathcal{H} = \otimes_j \mathcal{H}_j, \quad H_j = \mathbb{C}^2$$

EVERETTIAN

Quantum analog Gibbs sampling

- Quantum Annealers
  - Physical system enacting thermalization
  - Physics of chip are direct analogy of algorithm



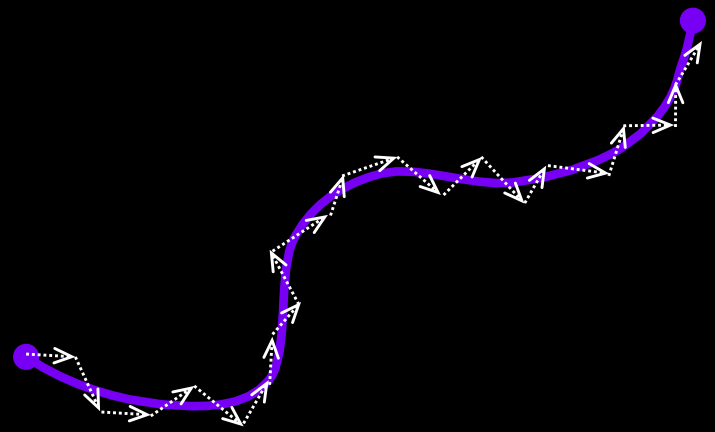
- Caveats:
  - Physical temperature
  - Connectivity issues
  - Low-quality (flux) qubits
  - Less flexible/embedding problems

Quantum Analog:

physical evolution governed by the physics of the chip

Quantum simulated thermalization with circuits

- Theoretically:
  - Quantum simulated to arbitrary accuracy, given enough gates
  - No connectivity problems (virtualized physics)
- Caveats:
  - Would need full fault-tolerance
  - $>10^9$ gates
- Near-term?
  - Could we do something similar but tailored to Noisy Intermediate Scale Quantum Devices?
  - Yes.



Quantum Simulation:

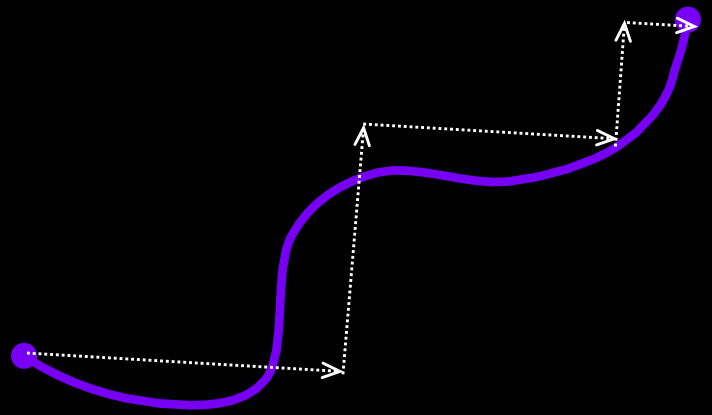
evolution decomposed into small pulses approximating evolution path

EVERETTIAN

Quantum-classical hybrid thermalization

- Hybrid Variational approach
  - Fixed number of quantum pulses
  - Classically optimize pulse lengths
  - CPU+QPU share optimization load

- Quantum Approximate Optimization Algorithm (QAOA)
  - inspired from Adiabatic QC
  - Proven quantum supremacy
  - Near-term implementable



Quantum-Classical Hybrid:

fixed number of pulses, classically variationally optimized to mimimize distance to target state

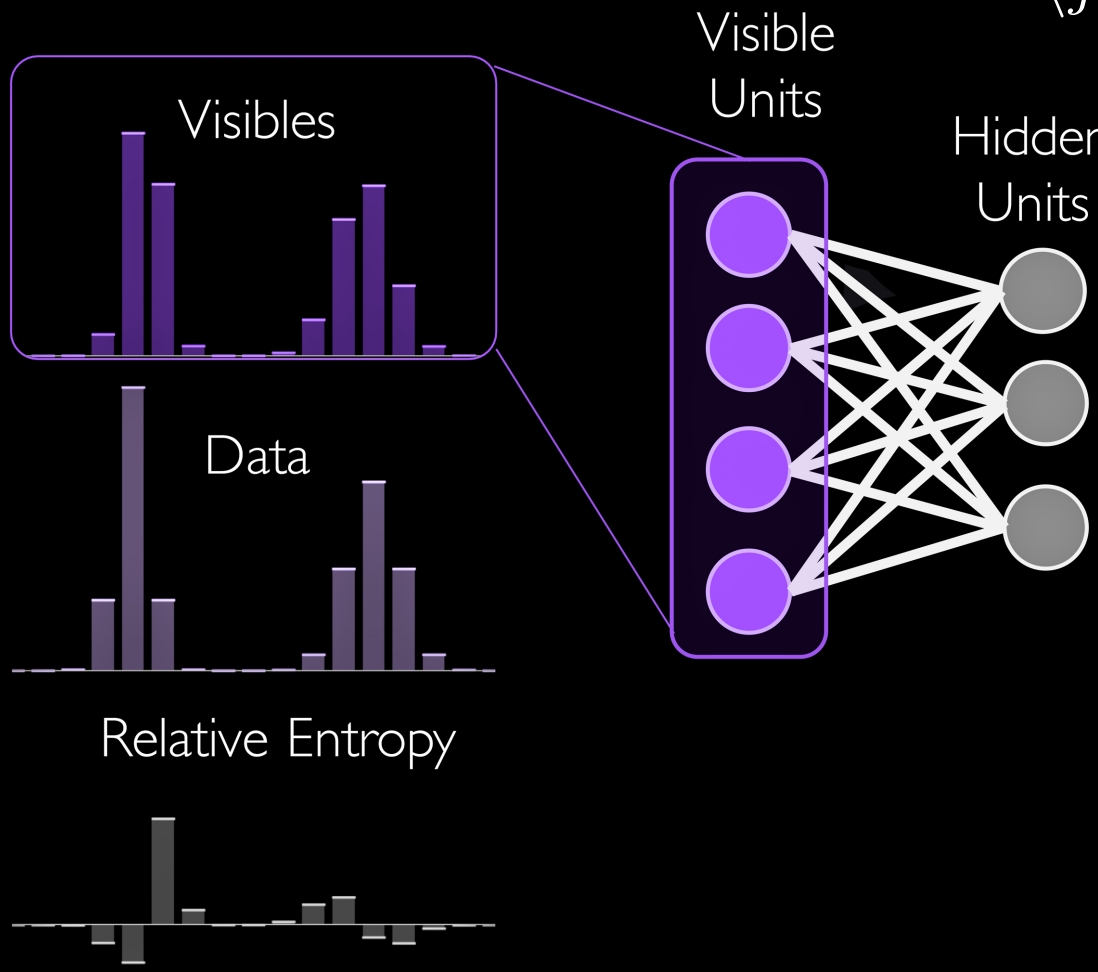
Technical Background

Training quantum Boltzmann machines

$$\rho \sim e^{-\beta H}$$

$$H = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k - \sum_j B_j Z_j$$

Network Hamiltonian



$$\boldsymbol{\theta} = \{\{J_{jk}\}, \{B_j\}\}_{jk}$$

Network parameters

Want to minimize

$$D(\rho_{\text{data}} \| \rho_{\text{vis}})$$

Gradient descent?

$$\delta\boldsymbol{\theta} = -\partial_{\boldsymbol{\theta}} D(\rho_{\text{data}} \| \rho_{\text{vis}})$$

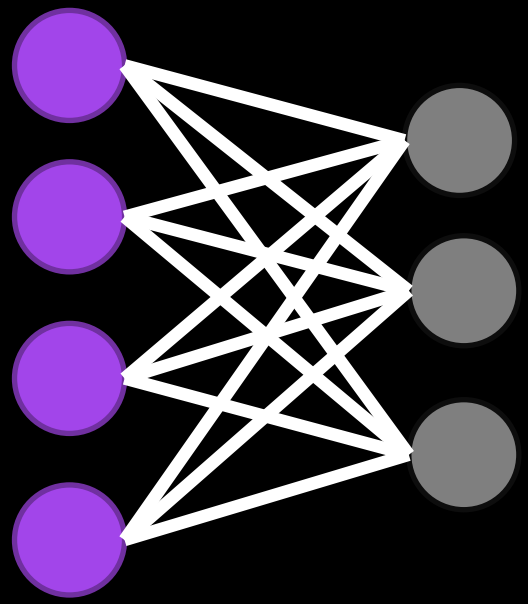
Too hard.
Minimize upper bound instead

EVERETTIAN

Bound-based update rule

$$H = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k - \sum_j B_j Z_j \qquad \boldsymbol{\theta} = \{\{J_{jk}\}, \{B_j\}\}_{jk}$$

~~Minimize KL~~ Minimize upper-bound[1] via update rule:



$$\delta\boldsymbol{\theta} = \frac{1}{|D|} \sum_{x \in D} \langle \partial_{\boldsymbol{\theta}} H \rangle_{H+V_x} - \langle \partial_{\boldsymbol{\theta}} H \rangle_H$$

Positive phase

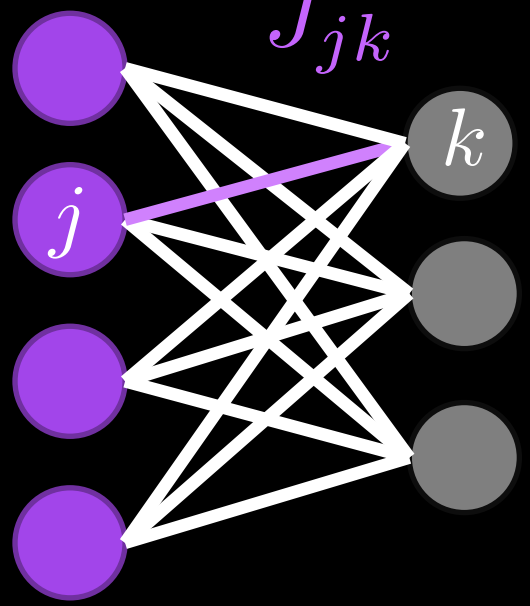
Negative phase

Gibbs sampling needed for each term

$$\langle \ldots \rangle_K \equiv \frac{\mathrm{tr}(e^{-\beta K} \ldots)}{\mathrm{tr}(e^{-\beta K})} \qquad V_x = -\log |x\rangle \langle x|_v$$

[1] Amin, M. H., Andriyash, E., Rolfe, J., Kulchytskyy, B., & Melko, R., *Quantum Boltzmann Machine*, **[arXiv:1601.02036]**

Example: updating weights



$$H = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k - \sum_j B_j Z_j$$

$$\partial_{J_{jk}} H = Z_j Z_k \qquad Z = |\uparrow\rangle\langle\uparrow| - |\downarrow\rangle\langle\downarrow|$$

Dataset

$$D = \{0010, 1100, \ldots\} \cong \{\uparrow\uparrow\downarrow\uparrow, \uparrow\uparrow\downarrow\downarrow, \ldots\}$$

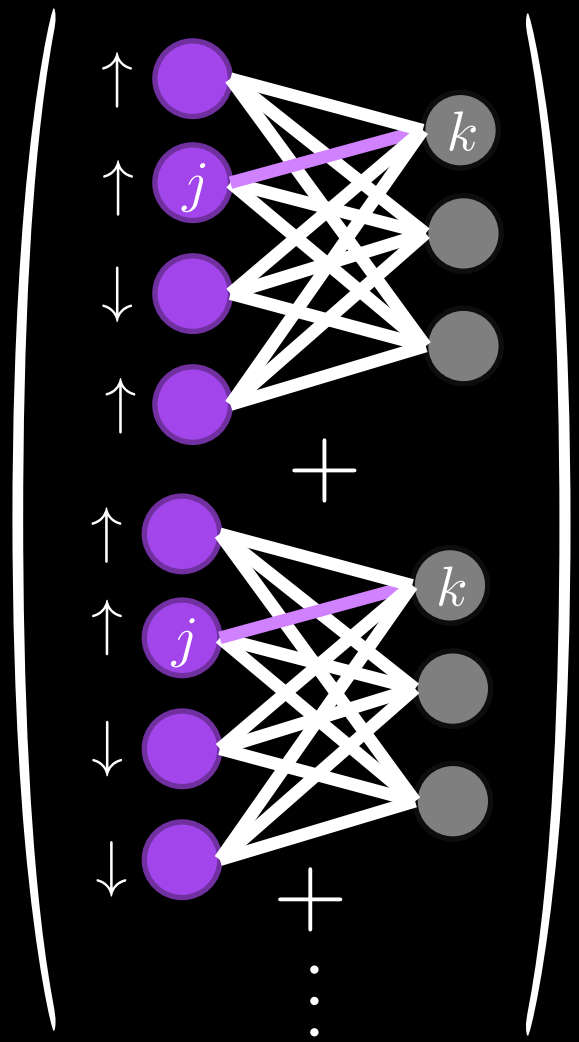
Measure clamped & unclamped thermal correlations

$$\delta J_{jk} = \frac{1}{|D|} \sum_{x \in D} \langle Z_j Z_k \rangle_{\text{clamped, x}} - \langle Z_j Z_k \rangle_{\text{unclamped}}$$

Example: updating weights J_{jk} $\partial_{J_{jk}} H = Z_j Z_k$

Dataset $D = \{0010, 1100, \ldots\} \cong \{\uparrow\uparrow\downarrow\uparrow, \uparrow\uparrow\downarrow\downarrow, \ldots\}$

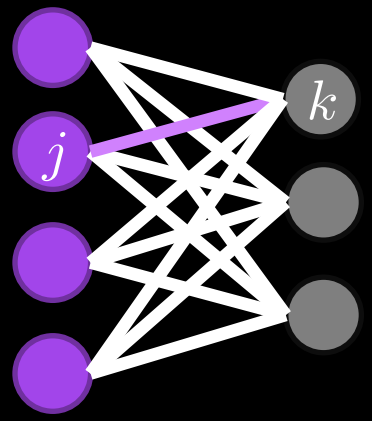
$$\delta J_{jk} = \frac{1}{|D|}$$



Pos. phase

Clamped Gibbs

$-$

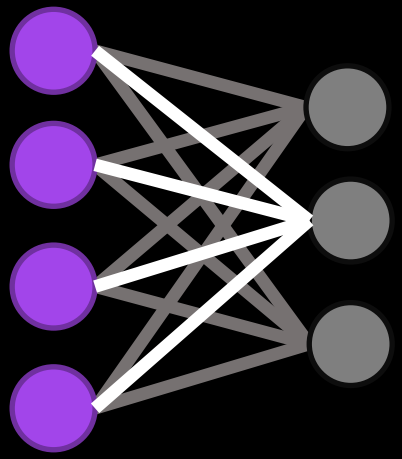


Neg. phase

Unclamped Gibbs

EVERETTIAN

Classical Alternatives to Clamped Sampling

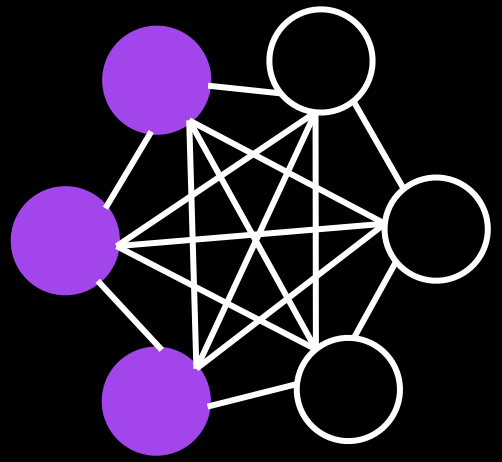


For RBM's:

- Feedforward data points

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

$$p(z_k = 1 | \vec{x}_{\text{vis}}) = \sigma(\sum_j J_{jk} x_j + B_j)$$



For more general BM's

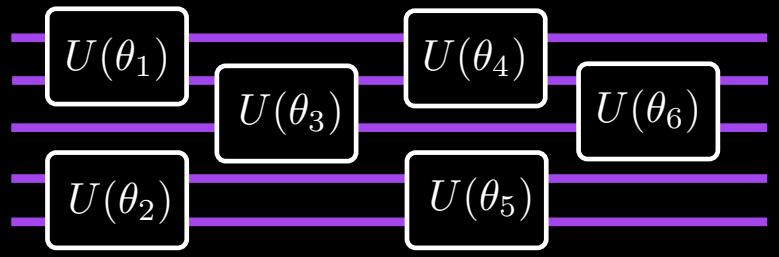
- Markov Chain Monte Carlo sampling
- Other sampling methods (QMC etc.)

EVERETTIAN

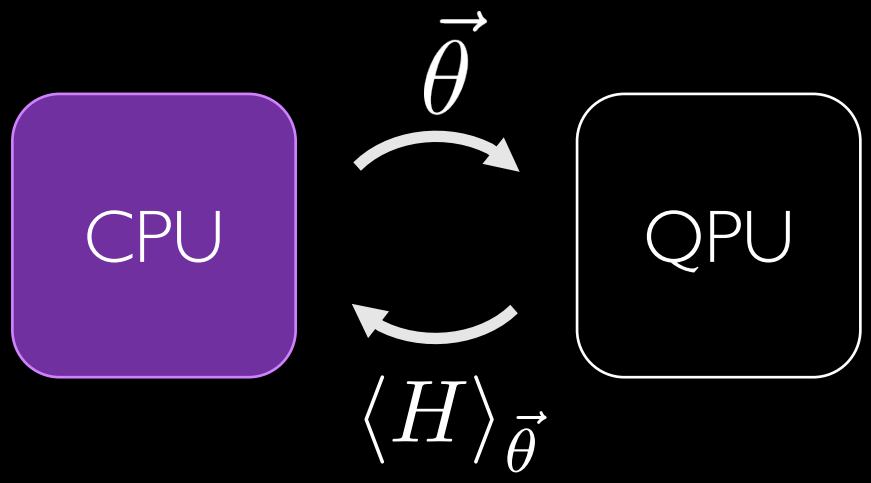
Quantum-Classical Variational Algorithms

- Goal: find $|\psi\rangle$ minimizing $\langle H \rangle$

Parameterize a family of ansatz circuits $U_{\vec{\theta}}$



Loop:



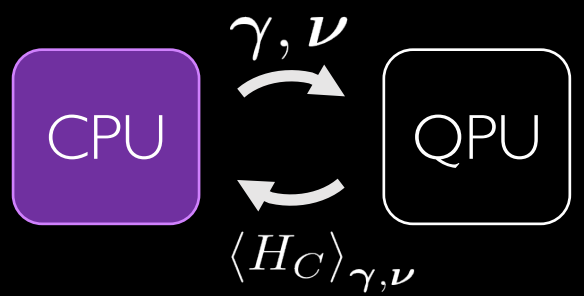
until min reached

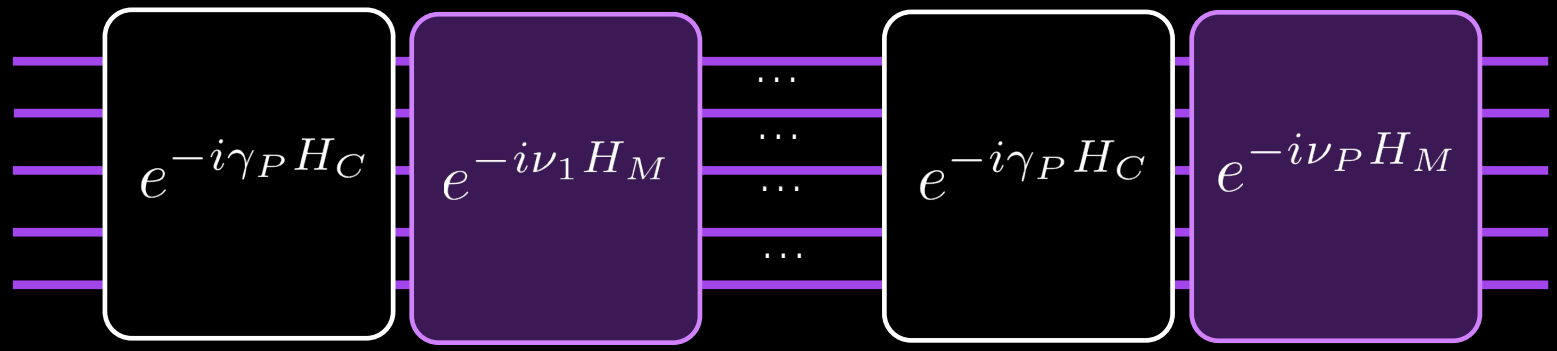
EVERETTIAN

Quantum Approximate Optimization Algorithm

- Hybrid variational algo inspired from Adiabatic QC

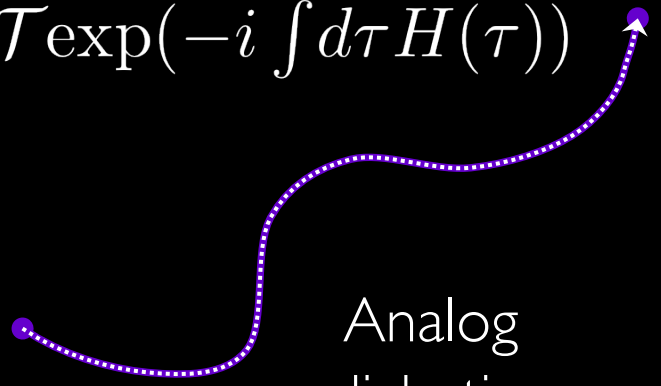
Mixer: $H_M = -\sum_j X_j$ Cost: $H_C = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k$





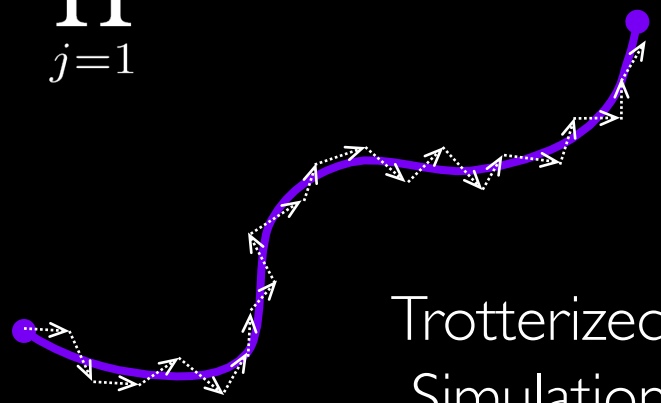
$$H(\tau) = (1-\tau)H_M + \tau H_C$$

$$\mathcal{T}\exp(-i\int d\tau H(\tau))$$



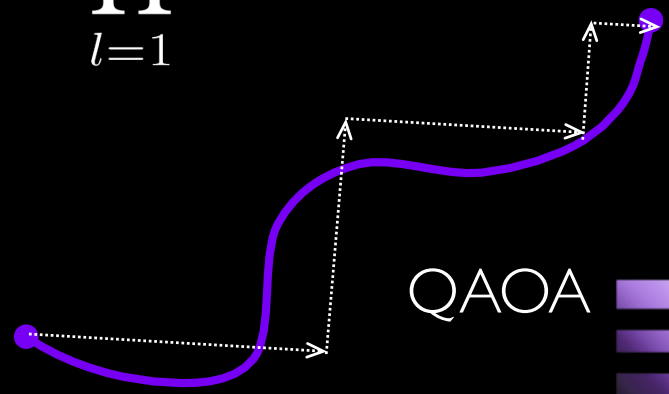
Analog adiabatic

$$\prod_{j=1}^{\lceil 1/\Delta\tau \rceil} e^{-i\tau\Delta\tau H_C} e^{-i(1-\tau)\Delta\tau H_M}$$



Trotterized Simulation

$$\prod_{l=1}^{P} e^{-i\nu_l H_M} e^{-i\gamma_l H_C}$$



QAOA

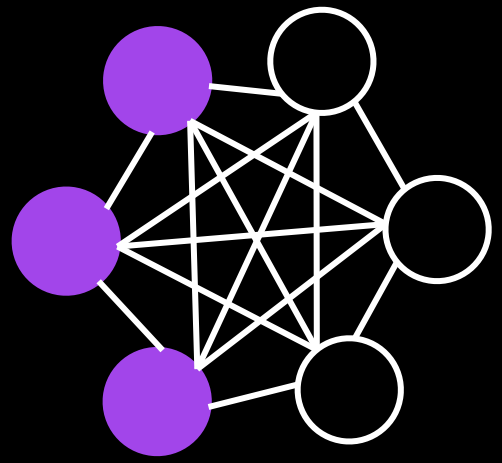
EVERETTIAN

Our algorithm

Quantum Approximate Thermalization

Say we want to sample from BM Gibbs state $\rho_\beta \sim e^{-\beta H_{\mathrm{bm}}}$

where $H_{\mathrm{bm}} = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k - \sum_j B_j Z_j$



- Initiate system in "easy" thermal state

$$\rho \sim e^{-\beta \sum_j X_j} \quad \text{or} \quad \rho \sim e^{-\beta \sum_j Z_j}$$

- Use QAOA to min $\langle H_C \rangle$ with

Cost: $H_C \equiv H_{\mathrm{bm}}$ Mixer: $H_M = -\sum_j X_j$

  - Brings our state closer to thermal (Gibbs) state

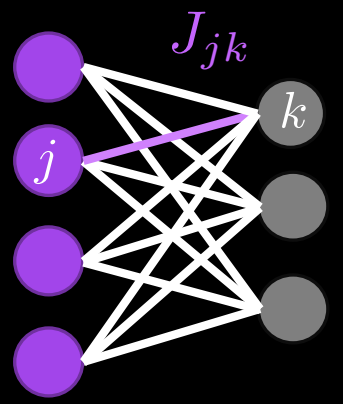
Free energy $F = \langle H_C \rangle - \frac{1}{\beta} S = D(\rho \| \rho_\beta)$ Relative entr.

Quantum Approximate Boltzmann Machines

QABoM

- Train (semi-)restricted BM using bound-based rule

$$\delta J_{jk} = \frac{1}{|D|} \sum_{x \in D} \langle Z_j Z_k \rangle_{\text{clamped, x}} - \langle Z_j Z_k \rangle_{\text{unclamped}}$$



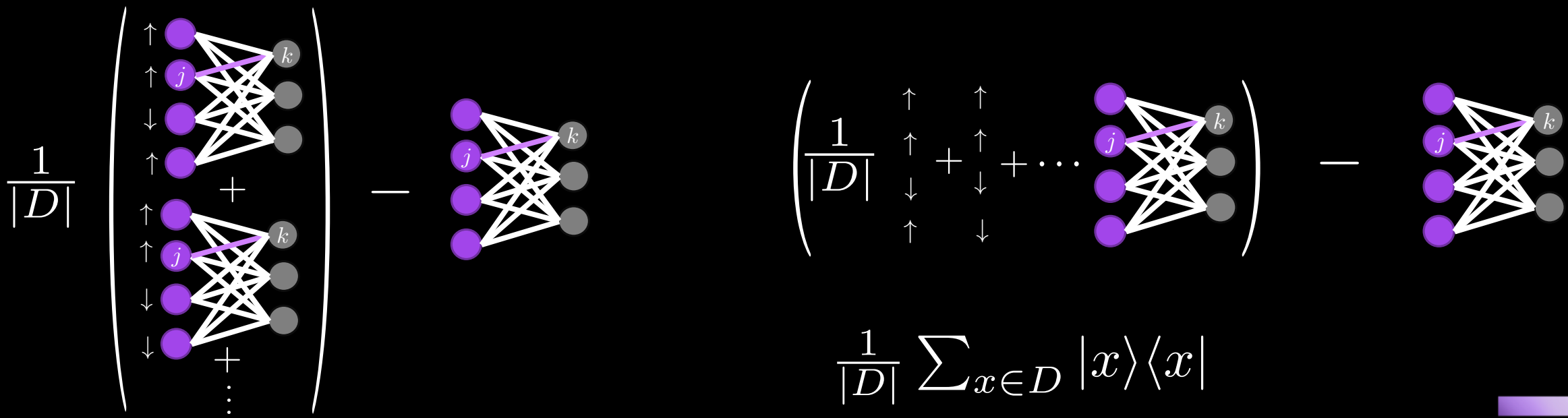
- Use Quantum Approximate Thermalization for
  - Unclamped + clamped Gibbs sampling
  - Clamping can be regular or quantum randomized

- For RBM case
  - Can do positive phase + inference classically

Quantum Randomized Clamping

QRC for QABoM $\quad \delta J_{jk} = \frac{1}{|D|} \sum_{x \in D} \langle Z_j Z_k \rangle_{\text{clamped, x}} - \langle Z_j Z_k \rangle_{\text{unclamped}}$

- Randomize clamping during QAOA
  - faster training AND better performance



$$\frac{1}{|D|} \sum_{x \in D} |x\rangle\langle x|$$

EVERETTIAN

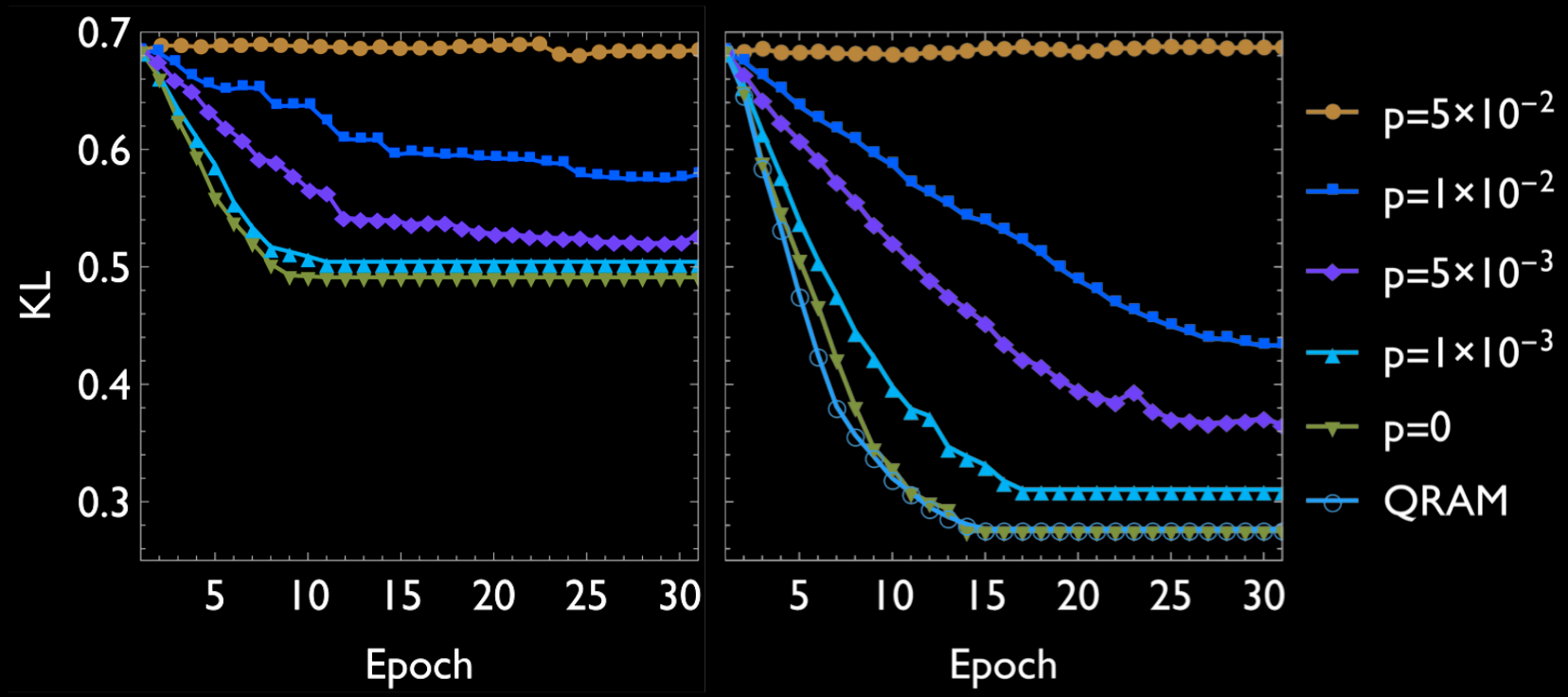
QABoM paper results

[arXiv:1712.05304]

Training an RBM with noisy(depolarizing) circuits using Rigetti Forest QVM

$$\mathcal{N}_p(\rho) = (1-3p)\rho + pX\,\rho X + pY\,\rho Y + pZ\,\rho Z$$

Evolution of Error (lower is better)

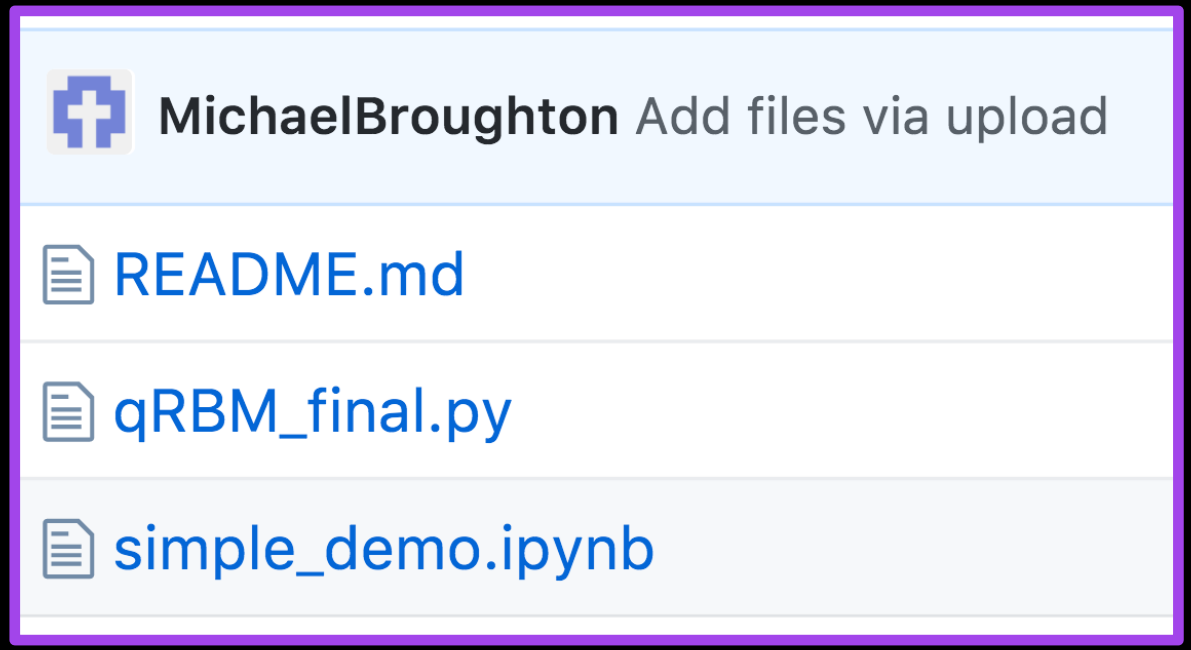


EVERETTIAN

It’s coding time!

Code walkthrough

GitHub.com/MichaelBroughton/QABoM



EVERETTIAN

Importing dependencies

```
import numpy as np
from scipy.optimize import minimize
from scipy.optimize import fmin_bfgs


import pyquil.quil as pq
import pyquil.api as api
from pyquil.paulis import *
from pyquil.gates import *


from grove.pyqaoa.qaoa import QAOA
from grove.pyvqe.vqe import VQE


import json
import copy
```

Setting up pyquil

Importing both QAOA and VQE

EVERETTIAN

Setting up the qRBM class

```
class qRBM:

    """
    Quantum Classical Hybrid RBM implementation.
    """


    def __init__(self, QM, num_visible, num_hidden, n_quantum_measurements=None, verbose=False):
        """
        create an RBM with the specified number of visible and hidden units

        Params
        ----------------------------------------------------------------

        QM:                     (rigetti QVM connection) QVM connection for which to use for quantum ci
        num_visible:            (int) Number of visible units in RBM
        num_hidden              (int) Number of hidden units in RBM
        n_quantum_measurements: (int) Number of measuremants to use for Quantum expectation estimation
        verbose:                (bool) Verbosity of qRBM

        ----------------------------------------------------------------
        """
        self.n_visible = num_visible
        self.n_hidden = num_hidden
        self.qvm = QM
        self.verbose = verbose
        self.n_quantum_measurements = n_quantum_measurements


        #tweak at your leisure.
        self.n_qaoa_steps = 1

        self.beta_temp = 2.0
```

Network setup

Temp and QAOA steps

Setting up the qRBM class

```
#don't tweak below here unless you know what you're doing.


# only want this for built in expectation calculations...
self.vqe_inst = VQE(minimizer=minimize,
                    minimizer_kwargs={'method': 'nelder-mead'})


self.state_prep_angle = np.arctan(np.e**(-self.beta_temp/2.0)) * 2.0
self.WEIGHTS = np.asarray(np.random.uniform(
                    low=-0.1 * np.sqrt(6. / (num_hidden + num_visible)),
                    high=0.1 * np.sqrt(6. / (num_hidden + num_visible)),
                    size=(num_visible, num_hidden)))

# IN THIS VERSION BIAS IS UNUSED!
self.BIAS = np.asarray(np.random.uniform(
                    low=-0.1 * np.sqrt(6. / (num_hidden + num_visible)),
                      high=0.1 * np.sqrt(6. / (num_hidden + num_visible)),
                      size=(num_hidden)))

#BIASES ARE ON HIDDENS.

# W[i][j] = ith visible to jth hidden
# Bias[j] = bias on jth hidden.
```

Use VQE for expect. values

Setting up random initial network parameters

EVERETTIAN

Setting up the unclamped QAOA sampling

```
def make_unclamped_QAOA(self):
    """
    Internal helper function for building QAOA circuit to get RBM expectation
    using Rigetti Quantum simulator

    Returns
    -------------------------------------------------
    nus:        (list) optimal parameters for cost hamiltonians in each layer of QAOA
    gammas:     (list) optimal parameters for mixer hamiltonians in each layer of QAOA
    para_prog:  (fxn closure) fxn to return QAOA circuit for any supplied nus and gammas
    -------------------------------------------------

    """

    visible_indices = [i for i in range(0, self.n_visible)]
    hidden_indices = [i + self.n_visible for i in range(0, self.n_hidden)]
```

Define hidden/visible indices

EVERETTIAN

Setting up the unclamped QAOA sampling

```
full_cost_operator = []
full_mixer_operator = []
for i in visible_indices:
    for j in hidden_indices:
        full_cost_operator.append(PauliSum([PauliTerm("Z", i, -1.0 * self.WEIGHTS[i][j - self.n_visible] * PauliTerm("Z", j, 1.0)]))

# UNCOMMENT THIS TO ADD BIAS IN *untested* in this version of code*
# for i in hidden_indices:
#   full_cost_operator.append(PauliSum([PauliTerm("Z", i, -1.0 * self.BIAS[i - self.n_visible])]))

for i in hidden_indices + visible_indices:
    full_mixer_operator.append(PauliSum([PauliTerm("X", i, 1.0)]))
```

Cost Hamiltonian

$$H_C = -\sum_{\langle jk \rangle} J_{jk} Z_j Z_k - \sum_j B_j Z_j$$

Mixer Hamiltonian

$$H_M = -\sum_j X_j$$

EVERETTIAN

Setting up the unclamped QAOA sampling

```
state_prep = pq.Program()
for i in visible_indices + hidden_indices:
    tmp = pq.Program()
    tmp.inst(RX(self.state_prep_angle, i + n_system), CNOT(i + n_system, i))
    state_prep += tmp
```

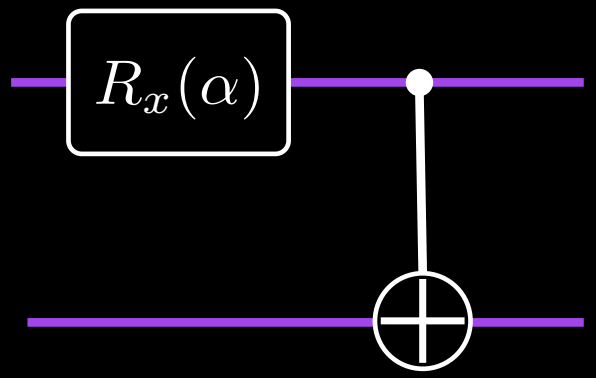
Prep thermal state using entanglement (for faster sim)

Want:

$$\rho \sim e^{-\beta \sum_j Z_j} \sim \bigotimes_j \sum_{z_j \in \{\pm 1\}} e^{-\beta z_j} |z_j\rangle\langle z_j|$$

Prep each qubit as:

$$\sqrt{2\cosh(\beta)} \sum_{z \in \{1,-1\}} e^{-\beta z} |z\rangle_S \otimes |z\rangle_E$$



EVERETTIAN

Setting up the unclamped QAOA sampling

```
full_QAOA = QAOA(self.qvm,
                 n_qubits=n_system,
                 steps=self.n_qaoa_steps,
                 ref_hamiltonian=full_mixer_operator,
                 cost_ham=full_cost_operator,
                 driver_ref=state_prep,
                 store_basis=True,
                 minimizer=fmin_bfgs,
                 minimizer_kwargs={'maxiter':50},
                 vqe_options={'samples': self.n_quantum_measurements},
                 rand_seed=1234)


nus, gammas = full_QAOA.get_angles()

if self.verbose:
    print 'Found following for nus and gammas from QAOA'
    print nus
    print gammas
    print '-'*80

program = full_QAOA.get_parameterized_program()
return nus, gammas, program, 0 #full_QAOA.result['fun']
```

Put QAOA ingredients together

Set optimal QAOA angles output

Use pyQuil parametric program

Setting up the network training

```
def train(self, DATA, learning_rate=0.1, n_epochs=100, quantum_percentage=1.0, classical_percentage=0.0):
    """
```

```
assert(quantum_percentage + classical_percentage == 1.0)

DATA = np.asarray(DATA)

for epoch in range(n_epochs):

    print 'Beginning epoch', epoch

    visible_indices = [i for i in range(0, self.n_visible)]
    hidden_indices = [i + self.n_visible for i in range(0, self.n_hidden)]

    new_weights = copy.deepcopy(self.WEIGHTS)
    new_bias = copy.deepcopy(self.BIAS)

    model_nus, model_gammas, model_para_prog, _ = self.make_unclamped_QAOA()
    model_sampling_prog = model_para_prog(np.hstack((model_nus, model_gammas)))

    print 'Found model expectation program....'
```

Option to mix quantum and classical RBM weight updates

Find optimal QAOA given current weights

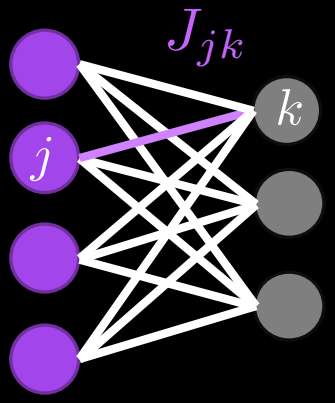
Setting up the network training

```
neg_phase_quantum = np.zeros_like(self.WEIGHTS)


# UNCOMMENT FOR BIAS
# neg_phase_quantum_bias = np.zeros_like(self.BIAS)


for a in range(self.n_visible):
    for b in range(self.n_hidden):
        model_expectation = self.vqe_inst.expectation(model_sampling_prog,
                                                      sZ(visible_indices[a]) * sZ(hidden_indices[b]),
                                                      self.n_quantum_measurements,
                                                      self.qvm)

        neg_phase_quantum[a][b] = model_expectation
```

Thermal correlation exp. value

We compute only unclamped (neg phase) quantumly for demo

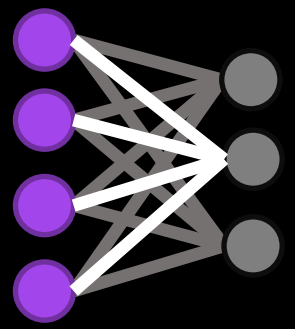
$$\delta J_{jk} = \frac{1}{|D|} \sum_{x \in D} \langle Z_j Z_k \rangle_{\text{clamped, x}} - \langle Z_j Z_k \rangle_{\text{unclamped}}$$



EVERETTIAN

Setting up the network training

```
#IF ADDING BIAS MODIFY THIS AS WELL!!!
#follow all standard conventions...
hidden_probs = self.sigmoid(np.dot(DATA, self.WEIGHTS))
pos_phase = np.dot(DATA.T, hidden_probs) * (1./float(len(DATA)))


pos_hidden_states = hidden_probs > np.random.rand(len(DATA), self.n_hidden)


neg_visible_activations = np.dot(pos_hidden_states, self.WEIGHTS.T)
neg_visible_probs = self.sigmoid(neg_visible_activations)

neg_hidden_activations = np.dot(neg_visible_probs, self.WEIGHTS)
neg_hidden_probs = self.sigmoid(neg_hidden_activations)


neg_phase_classical = np.dot(neg_visible_probs.T, neg_hidden_probs) * 1./len(DATA)

if self.verbose:
    print 'POSITIVE PHASE,'
    print pos_phase
    print 'NEGATIVE PHASE (QUANTUM)'
    print neg_phase_quantum
    print 'Negative PHASE(classical)'
    print neg_phase_classical
    print 'WEIGHTS'
    print self.WEIGHTS
    print '-'*80
```



$$p(z_k = 1|\vec{x}_{\text{vis}}) = \sigma(\sum_j J_{jk}x_j + B_j)$$

Setting up classical positive phase via feedforward

EVERETTIAN

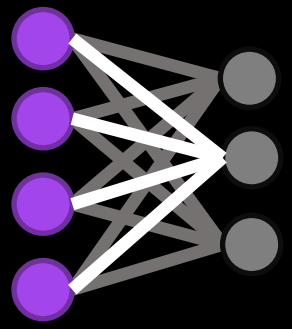
Updating weights

```
# can update weights with weighted avg of quantum and classical.
new_weights += learning_rate * (pos_phase - (classical_percentage*neg_phase_classical + quantum_percentage*neg_phase_quantum))

# UNCOMMENT HERE TO DO BIAS UPDATES
#can update bias with weighted avg of quantum and classical.
# new_bias += learning_rate * (pos_expect_bias - (0.0*neg_associations_bias + 1.0 * neg_phase_quantum_bias))
```

```
    self.WEIGHTS = copy.deepcopy(new_weights)
    self.BIAS = copy.deepcopy(new_bias)



    # fix from -1 to 0 for error calculation in probability terms.
    if self.verbose:
        error_measure_DATA = copy.deepcopy(DATA)
        error_measure_DATA[error_measure_DATA < 0] = 0
        print 'Error', np.sum((error_measure_DATA - neg_visible_probs) ** 2)


    with open("RBM_info.txt", "w") as myfile:
        myfile.write(json.dumps(list(self.WEIGHTS.tolist()))+'\n')
        myfile.write(json.dumps(list(self.BIAS.tolist()))+'\n')

    with open("RBM_history.txt", "a") as myfile:
        myfile.write(json.dumps(list(self.WEIGHTS.tolist()))+'\n')
        myfile.write(json.dumps(list(self.BIAS.tolist()))+'\n')
        myfile.write(str('-'*80) + '\n')

print 'Training Done!'
```

Inference

```
def transform(self, DATA):
    """
    Transforms vectors from visible to hidden

    Params
    ------------------
    DATA: (list) matrix containing rows that are data samples
    ------------------

    Returns
    -----------------
    result: (list) the hidden layers invoked from the samples in data matrix
    -----------------
    """
    # MODIFY THIS IF INCLUDING BIAS
    return self.sigmoid(np.dot(DATA, self.WEIGHTS))
```



$$p(z_k = 1|\vec{x}_{\text{vis}}) = \sigma(\sum_j J_{jk} x_j + B_j)$$

EVERETTIAN

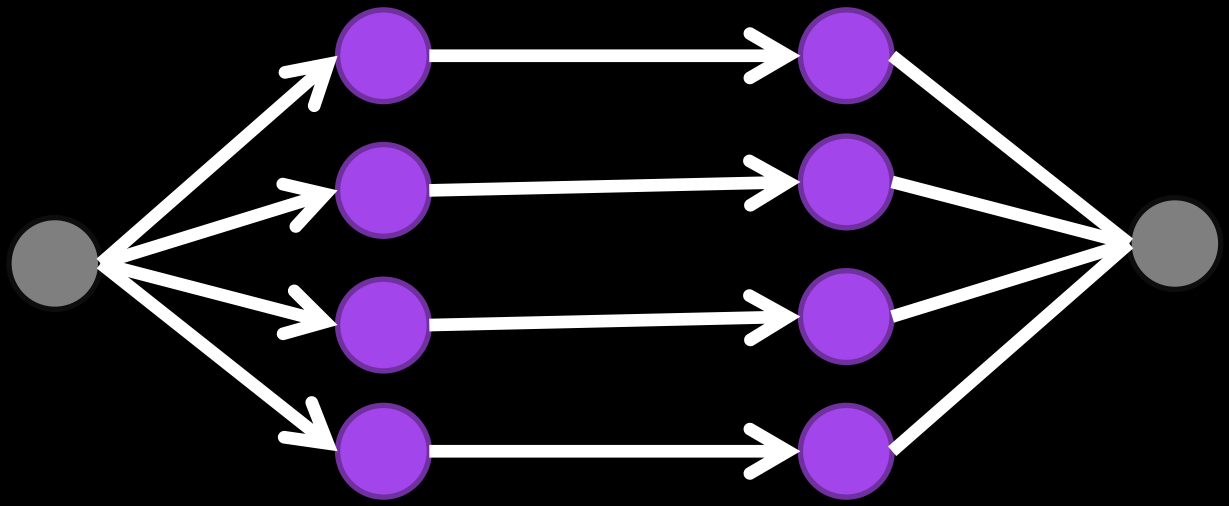
Simple example: Hidden bit subspace

- Encode a random bit redundantly into 4 bits
- Let our qBM find the needle in the haystack

Random coin flip

Encdoding

QABoM RBM decoding

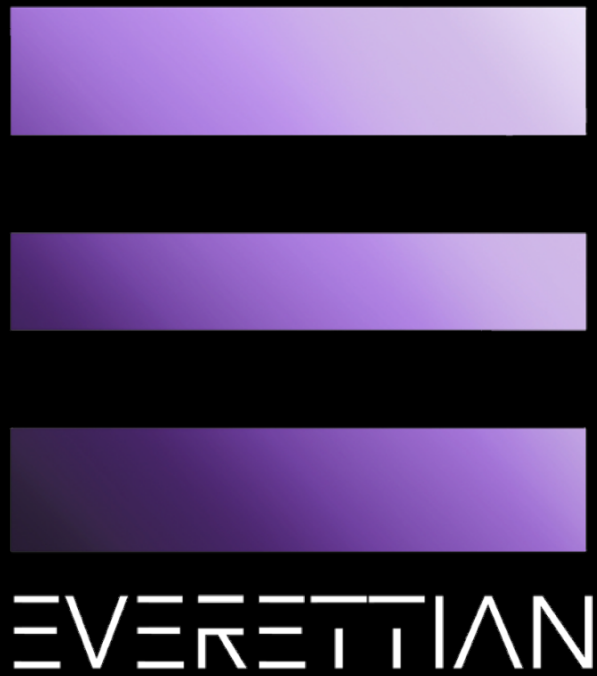


Can it identify this hidden variable?

To Jupyter and beyond!

A brief intermission

About Everettian



The Quantum Artificial Intelligence company.

Advancing and democratizing bleeding-edge
Quantum Machine Learning

The Quantum Valley

Waterloo: the perfect place to build the quantum future

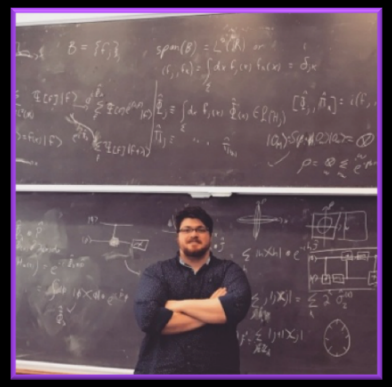
Entanglement with Toronto's classical AI ecosystem

EVERETTIAN

Join the team!

Tom Lubowe
CEO, Co-Founder

- Experience in FinTech Product Management
- Worked in Forex Execution, and Hedge Fund side of Finance Industry



Guillaume Verdon
CSO, Co-Founder

- PhD student in Quantum Machine Learning at IQC & Perimeter Institute
- MMath from IQC on Quantum Algorithms & Quantum Field Theory

Mike Broughton
CTO, Co-Founder

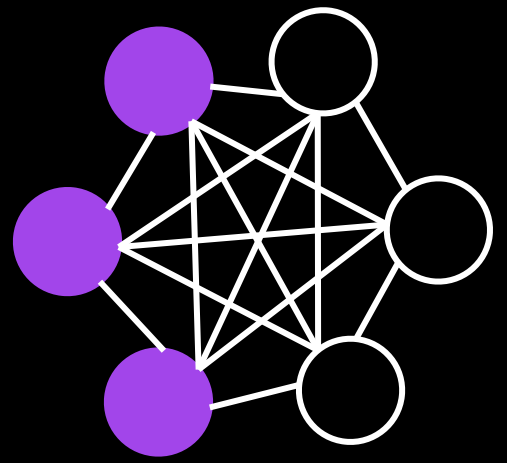
- Computer Science at UWaterloo and Institute for Quantum Computing (IQC)
- Early employee at Piinpoint, a Velocity & Y Combinator alumni company

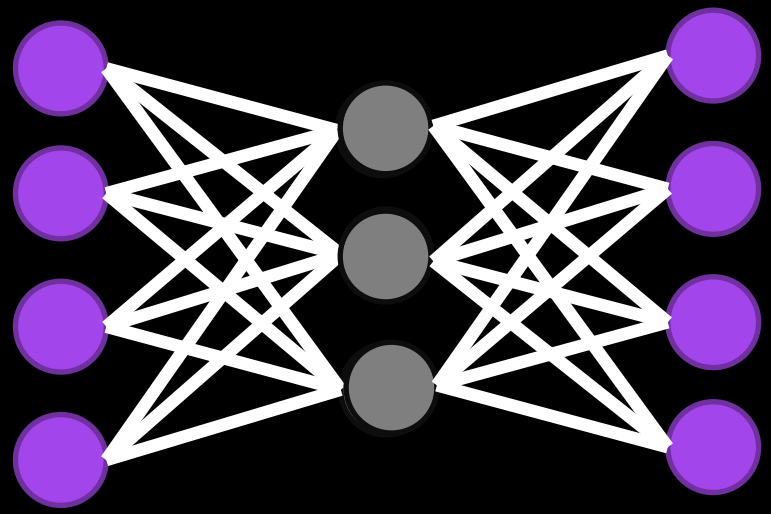
EVERETTIAN

Now back to the code…

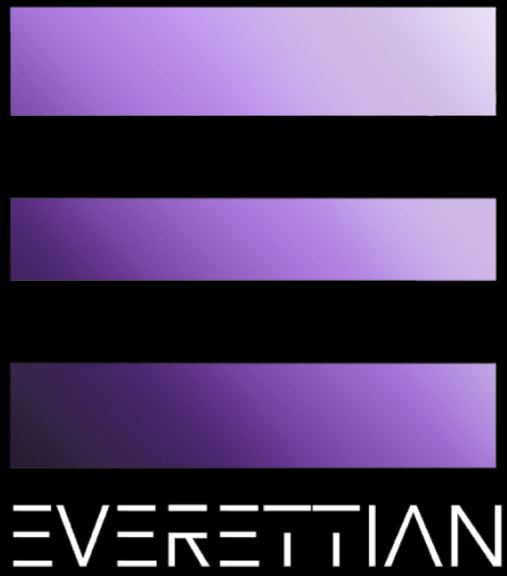
Outlook/Future work

- Hybrid quantum computers can train neural networks!
  - Time to implement on a chip
- Possible improvements
  - Variationally optimize temp/entropy
  - Further achieve lower Free energy
- Extensions
  - Apply to general BM
  - Apply to deep BM
  - Integrate into classical BM network





EVERETTIAN



Thanks!

Reach out: guillaume@everettian.com