"Integer Factorization Using Probability Computing"

Aleksandr Kariakin

Constructor University

28 July 2023

Motivation

There are many classes of problems that deterministic algorithms cannot efficiently address, including inference, invertible logic, sampling and optimization, leading to considerable interest in alternative computing schemes.

About this approach: binary stochastic neurons

Individual p-bits are stochastic building blocks with a normalized output m_i that takes on the values 0 and 1 with probabilities P_0 and P_1, respectively. These probabilities are controlled by their normalized inputs I_i; This is similar to the behaviour of a binary stochastic neuron, a well known concept in the field of stochastic neural networks and machine learning, which has an input–output relation

$$m_i = \theta[\sigma(I_i) – r],$$

where θ is the unit step function, σ is the sigmoidal function, r is a random number uniformly distributed between 0 and 1, and the input I_i is obtained from the synaptic function (described below).

About this approach: binary stochastic neurons

(NMOS) transistors to obtain a three-terminal p-bit (Fig. 2a). The output voltage for the ith p-bit, $V_{\mathrm{OUT},i}$, from this composite unit can be written in terms of the input voltage $V_{\mathrm{IN},i}$ in a form similar to the ideal binary stochastic neuron described above:

$$\frac{\overbrace{V_{\mathrm{OUT},i}}^{m_i}}{V_{\mathrm{DD}}} \approx \vartheta\left[\sigma\left\{\frac{\overbrace{V_{\mathrm{IN},i} - v_{0,i}}^{I_i}}{V_{0,i}}\right\} - r\right] \qquad (2)$$

About this approach

These p-bits can be used to perform useful functions by interconnecting them so that the ith p-bit is driven by a synaptic input I_i that is a function of all the other outputs $\{m_1, \ldots, m_N\}$. Boltzmann machines represent a subset of such networks for which I_i can be obtained from an energy function E using the relation $I_i = -\partial E(m_1, \ldots, m_N)/\partial m_i$.

Such networks will visit different configurations with probabilities given by the Boltzmann law $P(m_1, \ldots, m_N)$, which are proportional to $\exp[-E(m_1, \ldots, m_N)]$, so configurations with the lowest energy E occur with the highest probability. This property makes the networks naturally suited for solving optimization problems, similar to the way that AQC solves them, where the correct solution minimizes a cost function identified for E and is used to calculate the synaptic inputs I_i. Unlike in machine-learning schemes, these synaptic inputs are analytically deduced and not learned.

About this approach

The field of adiabatic quantum computing[9] (AQC) solves complex optimization problems by constructing networks of qubits in which the inter-qubit interactions are engineered to make the overall energy E reflect the cost function for the problem. One such algorithm[12] frames integer factorization of a given number F as an optimization problem by writing each of its factors X and Y in binary form and defining the cost function $E = (XY - F)^2$

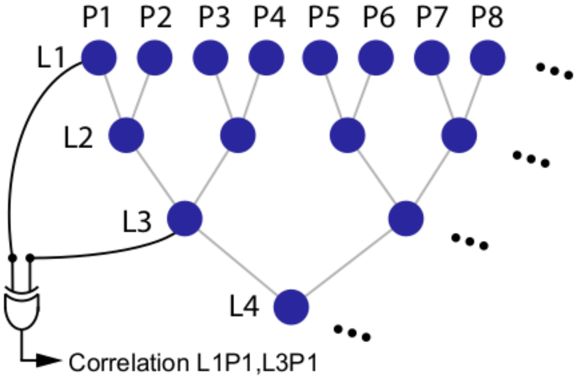
$$E\left(x_P, \ldots, x_1; y_Q, \ldots, y_1\right) = \left[\left(\sum_{p=0}^{P} 2^p x_p\right)\left(\sum_{q=0}^{Q} 2^q y_q\right) - F\right]^2 \tag{1}$$

with $x_0 = 1, y_0 = 1$ and P, Q denoting the number of bits needed to represent X and Y, respectively, so that the lowest energy state corresponds to the configuration of qubits $\{x_p, \ldots, x_1, y_q, \ldots, y_1\}$ that makes XY equal to F.

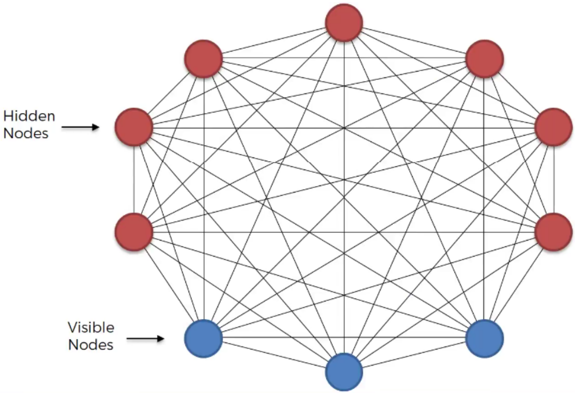
Why not Quantum Computing?

The increased number of qubits is a result of additional logical qubits in the Hamiltonian used to reduce the problem.

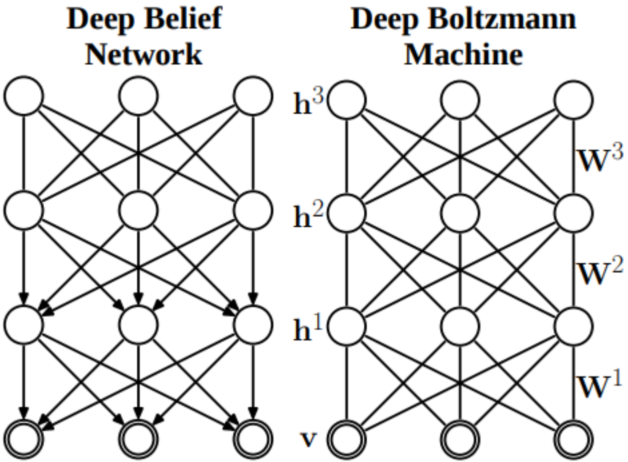
Why not Bayesian Network?



Why not Boltzmann Machine?



Why not Boltzmann Machine?



About this approach

The p-bits are electrically connected to form a functional asynchronous network, to which a modified adiabatic quantum computing algorithm that implements three- and four- body interactions is applied. An important aspect of this demonstration is the asynchronous operation of p-bits without any forced sequencing, unlike typical software implementations of Boltzmann machines, which require individual neurons or p-bits to be updated **sequentially**. This asynchronous feature allows the parallel operation of a large number of p-bits, leading to an unconventional computing paradigm.

Simulation from the article

In this section, we compare our experimental work with ideal simulations performed using software. The simulation updates all p-bits every Δt, flipping the ith p-bit with probability

$$P_i = 1 - exp(\frac{-\Delta t}{\tau_i}),$$

where the dwell time τ_i of the ith p-bit depends on the inputs I_i obtained from the synaptic function:

$$\tau_i = \tau_{0,i} exp(\pm I_i).$$

Here $\tau_{0,i}$ is the zero-bias dwell time, and I_i is positive if it is parallel to the state of the p-bit and negative if it is anti-parallel.

$$\tau_{0,i} = 10ps - 1ns?$$

Code Implementation

The following Python code transforms every bit using sigmoid and unit-step functions:

```python
def unit_step_with_rand(x):
 return torch.where(x >= torch.rand(1), torch.tensor(1.0), torch.tensor(0.0))

def transform(tensor: torch.Tensor):
 return unit_step_with_rand(torch.sigmoid(tensor))
```

Code Implementation

This function converts vector that is written like a tensor of binary values to number.

```
def number_from_tensor(tensor: torch.Tensor):
 powers_of_two = torch.vander(torch.Tensor([2]), N=tensor.size(dim=0)+1,
 increasing=True)
 scalar_mul = tensor * torch.reshape(powers_of_two, (-1,))[1:]
 return 1 + scalar_mul.sum()
```

Code Implementation

```
class MyIntegerFactorizationModel:

 def __init__(self, F: int, fitting_parameter: float):
     self.F = F
     length = F.bit_length()
     self.P = (length - 1) // 2
     self.Q = length - 2 - self.P
     # self.eternal_tensor = torch.ones(self.P + self.Q, requires_grad=True)
     self.eternal_tensor = torch.randint(0, 2, size=(self.P + self.Q,),
     dtype=torch.float32, requires_grad=True)
     self.tensor_collector = Counter()
     self.fitting_parameter = fitting_parameter
```

Code Implementation

These methods are calculating energy function and gradient for the task.

```
def energy_function(self, tensor: torch.Tensor) -> torch.Tensor:
    return self.fitting_parameter * (number_from_tensor(tensor[:self.P]) *
    number_from_tensor(tensor[self.P:]) - self.F) ** 2

def calculate_gradient(self):
    energy = self.energy_function(self.eternal_tensor)
    energy.backward(torch.ones(energy.shape))
    self.eternal_tensor.retain_grad()
    return self.eternal_tensor.grad
```

Code Implementation

These methods are evaluating a new tensor value and add current value to Counter.

```
def evaluate(self):
    self.manage_counting()
    gradient = self.calculate_gradient()
    index = torch.randint(self.P + self.Q, (1, ))
    current_grad = gradient[index]
    trans = transform(-current_grad)
    #trans = other_transform(-current_grad if self.eternal_tensor[index] == 0
    #else current_grad, 0.000005)
    with torch.no_grad():
        self.eternal_tensor.data[index] = trans

def manage_counting(self):
    first_number = number_from_tensor(self.eternal_tensor[:self.P])
    second_number = number_from_tensor(self.eternal_tensor[self.P:])
    self.tensor_collector[(first_number.item(), second_number.item())] += 1
```

Code Implementation

Other transform!

```
def dwell_function(tensor, tao):
    return tao * torch.exp(tensor)

def other_transform(tensor: torch.Tensor, delta):
    return unit_step_with_rand(1 -
    torch.exp(-delta/dwell_function(tensor, 10**(-11) - 10**(-9))))
```

Code Implementation

Calculate the answer

```
integer_factorization = MyIntegerFactorizationModel(35, 0.25)

for i in range(100):
    integer_factorization.evaluate()
print(integer_factorization.tensor_collector)
Output: Counter({(5.0, 7.0): 32, (7.0, 7.0): 25, (7.0, 3.0): 20, (1.0, 7.0): 11,
(3.0, 7.0): 6, (7.0, 5.0): 3, (7.0, 1.0): 1, (5.0, 3.0): 1,
(3.0, 3.0): 1})
```

Other approaches

- Ising Model
- cuda.synchronize()