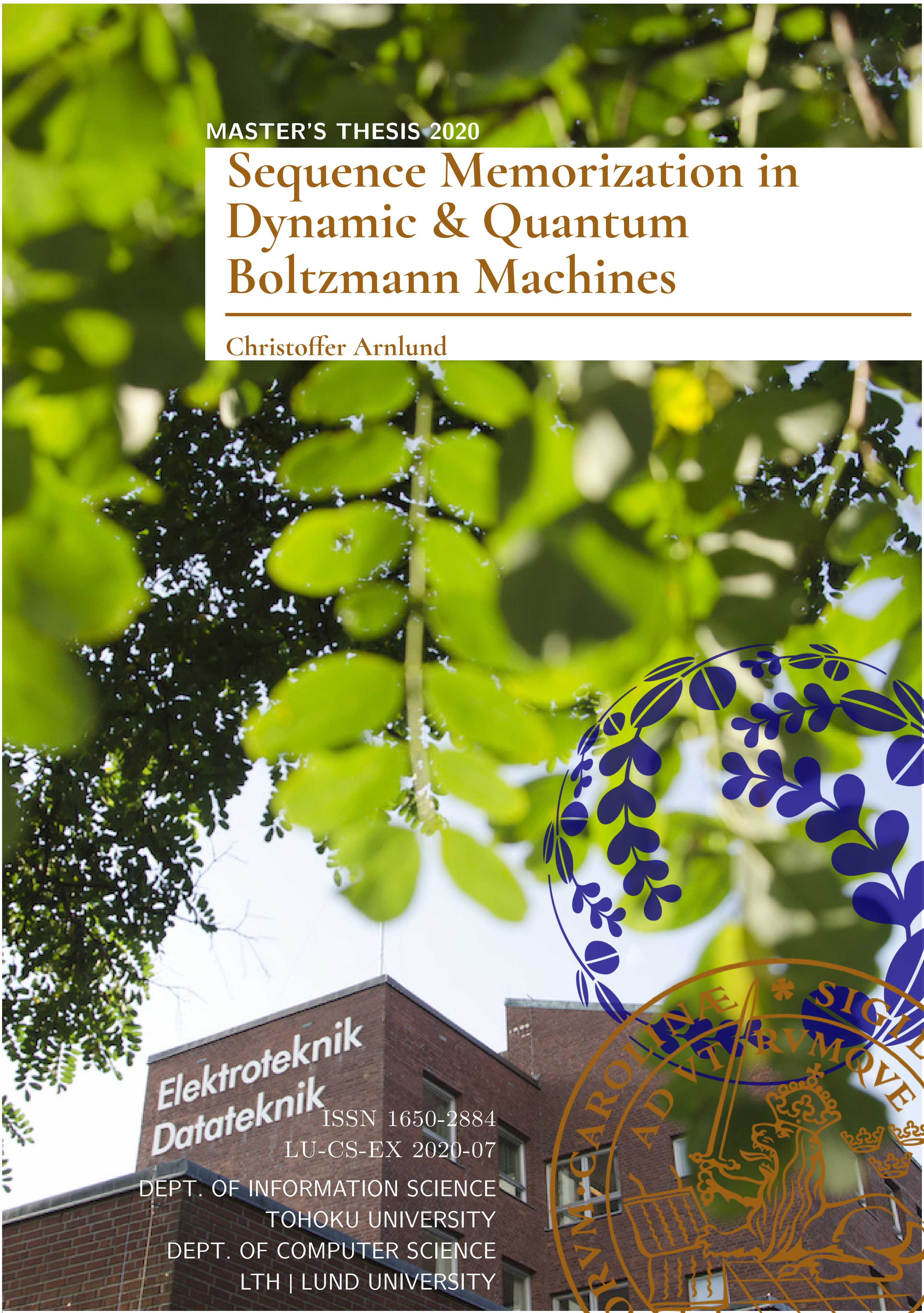


MASTER'S THESIS 2020

Sequence Memorization in Dynamic & Quantum Boltzmann Machines

Christoffer Arnlund

ISSN 1650-2884
LU-CS-EX 2020-07

DEPT. OF INFORMATION SCIENCE
TOHOKU UNIVERSITY
DEPT. OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE

Datavetenskap

LU-CS-EX: 2020-07

Sequence Memorization in Dynamic & Quantum Boltzmann Machines

Christoffer Arnlund

Sequence Memorization in Dynamic & Quantum Boltzmann Machines

Christoffer Arnlund
dat14car@student.lu.se

February 12, 2020

Master's work carried out at
the Department of Information Science, Tohoku University.

Supervisors:
Pierre Nugues, pierre.nugues@cs.lth.se
Kazuyuki Tanaka, kazu@tohoku.ac.jp
Masayuki Ohzeki, mohzeki@tohoku.ac.jp

Abstract

Memorizing sequences is useful for many applications such as *natural language processing* and *anomaly detection*, but can also be used to e.g. generate missing part of a photograph or music sheet.

In this work, we study the performance of memorizing sequences in *dynamic Boltzmann machines* and *quantum Boltzmann machines*. The hardware used for the quantum Boltzmann machine was a quantum annealing machine developed by D-Wave with 2000 qubits, the most powerful quantum annealing machine as to this day of writing.

The dynamic Boltzmann machines are trained with bitmap patterns of alphabetical images. We also train a quantum Boltzmann machine with the goal of learning classification of the same data set used for memorizing sequences.

The results show that dynamic Boltzmann machines perform well for sequence memorization for the data set we used. The quantum Boltzmann machines successfully classified the data set with an accuracy of $\sim 80\%$, but did not converge when solving sequence memorization.

This work was carried out at the Department of Computer Science, Lund University, Sweden and the Department of Information Science, Tohoku University, Japan.

Keywords: Quantum Boltzmann Machine, Dynamic Boltzmann Machine, Sequence Memorization, Sequence Model, D-Wave 2000Q, Quantum Annealing

Acknowledgements

We would like to thank Prof. Pierre Nugues for his support and advice, Prof. Volker Krueger for his role as examinator, the Colabs initiative for enabling this research in Japan and finally the Japan Student Services Organization (JASSO) for granting financial support for this research. We would also like to thank Prof. Kazuyuki Tanaka and Prof. Masayuki Ohzeki for allowing us to take part in their research and advice in the Tanaka Laboratory, Tohoku University.

Contents

Chapter 1

Introduction

An *artificial neural network* (ANN) can be described as a system that learns to perform a task by repeated exposures to examples of the task.

Implementations of ANNs commonly consist of sets of nodes (neurons) and connections between the nodes called edges. The edges have a weight that represents how strongly the neurons are bound. As the network gets examples presented to it, the weights are adjusted so that the output ultimately is as close as possible to the targets.

We can introduce a bias to an ANN: It is the shifting parameter $\boldsymbol{b}$ applied to the linear transformation of each layer. It is a constant which helps the model in a way that it can fit best for the given data.

There are many different types of architectures for an ANN such as feed-forward neural networks, recurrent neural networks, convolutional neural networks and energy-based models (EBM). EBM's are generative models and can be used to model the unknown distribution of some data such as images and text. A common implementation of an EBM is called a *Boltzmann network*.

We can use an ANN for many tasks, but in this report, we are focusing on training ANN's to memorize sequences. This is useful for many real-world applications today such as natural language processing, DNA sequencing, anomaly detection and it can be used to generate new information like music and art.

1.1 Boltzmann Machines

ANN's can be trained with the Hebb rule [1], which intuitively can be described as *cells that fire together, wire together*. Networks that are trained this way are called Hopfield networks. The neurons for a Hopfield network are deterministic; a network with neurons that are stochastic is called a Boltzmann machine (BM), see Figure 1.2 If we train a Boltzmann machine (BM) on sequences, they store static patterns and retrieve a particular pattern when an associated cue is presented to it. The architecture of a Boltzmann machine is shown in Figure 1.1.

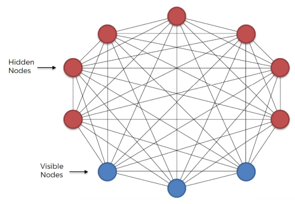


Figure 1.1: Boltzmann Machines use neural networks that are connected not only to other neurons in other layers, but also to neurons within the same layer.

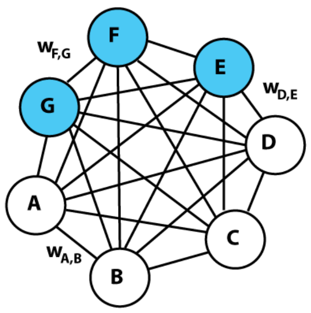


Figure 1.2: The Hopfield network has an "energy" defined for the overall network and produces binary results. But unlike Hopfield nets, Boltzmann machine neurons are stochastic.

One interesting property of these networks is that the training algorithm only needs local information and thus there is no need for backpropagation. BMs are in theory a general computational medium, e.g if trained on photographs the machine would model a distribution of photographs and could be used to e.g complete a partial photograph.

A shortcoming of BM's is that they stop being practical when the networks are scaled up. The need to compute all the weights between all the neurons creating a time complexity of 2^n where n is the number of neurons. This makes the networks grow exponentially in computation time as the network size increases. Another shortcoming is the so-called "variance trap", which means the neuron activations will saturate in the presence of noise. This is due to the fact that BMs are a stochastic system, the connection strengths are more plastic when the units being connected have activation probabilities intermediate between zero and one. The net effect is that noise causes the connection strengths to follow a random walk until the activities saturate.

The global energy E, in a Boltzmann machine, is given by [2] and shown in Equation 1.1.

$$E = -\left(\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i\right) \tag{1.1}$$

where:

- w_{ij} is the connection strength between neuron j and neuron i
- s_i is the state, $s_i \in \{0, 1\}$, of neuron i
- θ_i is the bias of neuron i in the global energy function. ($-\theta$ is the activation threshold for the neuron.)

The weight $w_{i,j}$ are represented as a symmetric matrix $W = [w_{ij}]$, with zeros along the diagonal.

A unit i then turns on with the probability given by the logistic function:

$$prob(s_i = 1) = \frac{1}{1 + e^{-z_i}} \tag{1.2}$$

where the scalar T is temperature of the system and $z_i = b_i + \sum_j s_i w_i j$.

If the units are updated sequentially in any order that does not depend on their total inputs, the network will eventually reach a Boltzmann distribution, also called *equilibrium* or *stationary distribution*. This probability of a state vector $\mathbf{v}$ is determined solely by the energy of that state vector *relative* to the energies of all possible binary vectors:

$$P(\mathbf{v}) = \frac{e^{-E(v)}}{\sum_u e^{-E_u}} \tag{1.3}$$

where $E(v) = -\sum_i s_i^v b_i - \sum_{i>j} s_i^v s_j^v w_{ij}$ and s_i^v is the binary state assigned to unit i by state vector v.

Given a training set of state vectors (the data), learning consists of finding weights and biases (the learnable parameters) that make those state vectors good. More specifically, we try to find weights and biases that define a Boltzmann distribution in which the training vectors have high probability, i.e, small energy E.

Furthermore, the search can be improved with the help of simulated annealing [3], and as an extension of that, quantum annealing [4].

1.1.1 Restricted Boltzmann Machines

One big drawback of the Boltzmann machine is the running time complexity. A solution for this is to limit the network connectivity. RBM's are two-layered (one layer being visible and the other one being hidden) and all the nodes in one layer are connected to all the nodes in the other layer but no two nodes in the same layer are connected together. This is called Restricted Boltzmann Machines (RBM's) and is shown in Figure 1.3. An RBM is thus a quite different model from a feed-forward neural network. They have connections going both ways (forward and backward) that have a probabilistic interpretation.

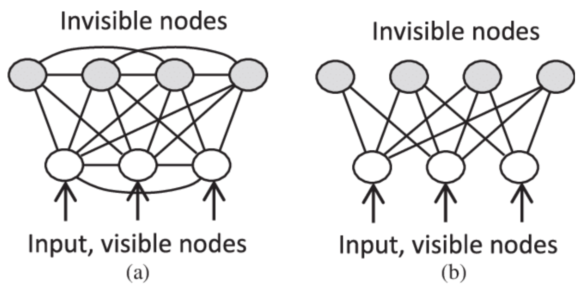


Figure 1.3: An example of a an Boltzmann machine architecture on the left and a Restricted Boltzmann architecture to the right (a) Boltzmann machine, (b) Restricted Boltzmann machine

1.1.2 Spiking Neural Networks

Spiking neural networks (SNNs) aim to increase the level of realism in neural networks. They incorporate the concept of time into their model by changing the idea that neurons fire directly, to instead fire only when a *membrane potential* (neurons electrical charge) reaches a specific value. When a neuron fires, it generates a signal which travels to other connected neurons that in their turn increase or decrease their potential according to this signal, a process called *synaptic time-dependent plasticity* [5].

A drawback Boltzmann machines have is their weak time dependency, which is an important variable when considering the memorization of sequences. To mitigate this a suggestion is to use synaptic time-dependent plasticity (STDP) [6].

STDP suggests that the co-activation of pre and postsynaptic neurons (The neuron where the signal is initiated is called the presynaptic neuron, while the neuron that receives the signal is called the postsynaptic neuron) sets a flag at the synapse, called an *eligibility trace*, that leads to a weight change only if an additional factor is present while the flag is set.

The weights of a SSNs are thus set based on two basic rules:

1. Any synapse that contributes to the firing of post-synaptic neuron should be made stronger;

2. Any synapse that does not contribute to a post-synaptic neuron should be made weaker.

The learning rules are also simple:

- If pre-connection neurons fire before a post-connection neuron the corresponding connection becomes stronger by a factor proportional to time (the lesser the time, the greater the change). This is called *long-term potentiation* (LTP) and can be seen in Figure 1.4.
- If the pre-connection fires after the post-connection the connection should become weaker by a factor proportional to time. This is called the Long Term Depression (LTD) and can be seen in Figure 1.4 .

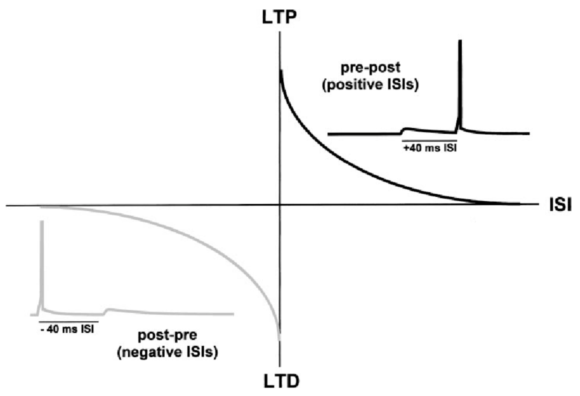


Figure 1.4: The figure shows the decay how the synapse eligibility trace is affected as a function of time. LTP is the long-term potentiation and LTD is the long-term depression.

1.1.3 Dynamic Boltzmann Machines

A previous study [7] postulates a new neural network that extends the Hebb rule with *spike-timing-dependent plasticity* (STDP).

A dynamic Boltzmann machine (DyBM) consists just like the Boltzmann machine of a set of neurons and synapses, but instead of a simple weight on the synapse, we now calculate the weights based on the neurons eligibility traces. Each neuron has a neural eligibility traces and the purpose of this is to aggregate the previous spikes in the neuron. The synapse also has a synaptic eligibility trace, and similarly to the neuron, this aggregates the previous spikes from the pre-synaptic neuron. The synaptic queue holds the values of the pre-synaptic neuron eligibility trace at a certain time.

Any neuron can be both a pre-synaptic and post-synaptic neuron and a neuron can be connected to itself via a synapse.

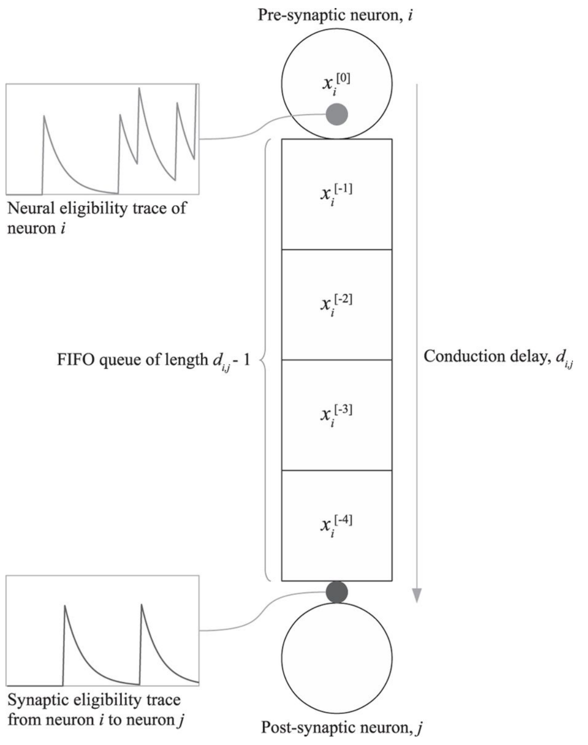

Pre-synaptic neuron, i
$x_i^{[0]}$
$x_i^{[-1]}$
$x_i^{[-2]}$
$x_i^{[-3]}$
$x_i^{[-4]}$
Neural eligibility trace of neuron i
FIFO queue of length $d_{i,j}$ - 1
Conduction delay, $d_{i,j}$
Synaptic eligibility trace from neuron i to neuron j
Post-synaptic neuron, j

Figure 1.5: A pre-synaptic neuron is connected to a post-synaptic neuron via a queue. The spike from a pre-synaptic neuron reaches the post-synaptic neuron after a delay. Each neuron stores an neural eligibility trace (an aggregation of previous spikes) and each synapse holds a synaptic eligibility trace (an aggregation of previous spike via the queue from the pre-synaptic neuron).

1.2 Quantum Computing

A classical computer uses electrons and gates to produce results that are in accordance with Boolean logic. For any two specific input states, we have one certain output state. The fundamental unit of classical computers are binary digits whose state is either 0 or 1. In a conventional semiconductor, this is represented as low and high voltage levels within a transistor.

Quantum computers are fundamentally different. The basic unit of registering a state is a so-called *qubit*, that also stores states of 0s and 1s but can be both at the same time. Instead of using transistors, the physical implementation of qubits can take shape in many forms such as using photons, electrons, and optical lattices [8].

A quantum system considers an electron's spin (the angular momentum) to be "up" or "down" and corresponds to the traditional computers 0 or 1.

Particles can exist in different states, for example they can be in different positions, have different energies or be moving at different speeds. In quantum mechanics, instead of thinking about a particle being in one state or changing between a variety of states, particles are thought of as existing across all the possible states at the same time. A good analogy can be thinking of lots of waves overlapping each other. This situation is known as a “superposition” of states.

When the electron is observed, the superposition collapses and the qubit obtains one state. It is impossible to witness an electron in a state of superposition because witnessing requires the very exchange of photons that causes such a superposition to collapse.

Each superposition state a qubit may take may be represented by a vector in a Bloch sphere, see Figure 1.6. The North and South poles of the Bloch sphere are typically chosen to correspond to the standard basis vectors 0 and 1 respectively, which in turn might correspond e.g. to the spin-up and spin-down states of an electron.

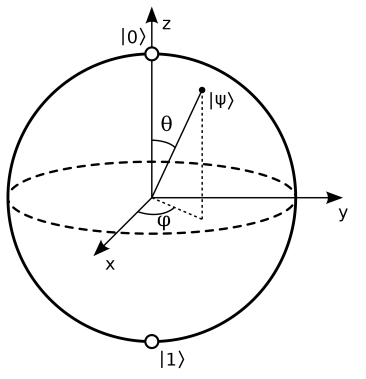


Figure 1.6: In quantum mechanics, the Bloch sphere is a geometrical representation of a qubit, named after the physicist Felix Bloch

In a system with n qubits, the number of possible superposition states for each qubit is 2^n. This would mean for a 4 qubit system, *each* qubit would have $2^4 = 16$ possible states and the whole system would have $16 \times 4 = 64$ possible state, this is comparing to a traditional computer where a 4 bit system only has 16 total states.

The advantages of quantum systems are clear here: If one can utilize the work done in the unobserved state, a quantum computer is capable of performing algorithmic functions on units problems that are today would take several thousands of years.

The challenge for quantum computers is to take full advantage of the ability to manipulate qubits during their superposition, prior to their decoherence (the reversion of qubits to their classical 0 or 1 states).

There are two types of quantum programs, which function very differently from one another: *quantum gate models* and *quantum annealing computers*. Quantum gate models create a quantum circuit and use gates to manipulate qubits in a similar way to classical computers and are very capable of solving generalized problems. A quantum annealing system is not a general-purpose computer. It can only solve certain problems that can be structured as energy minimization. Quantum annealing works best on problems, where there are a lot of potential solutions and finding “good enough” or “local minima” solutions.

1.3 Quantum Annealing

Quantum annealing uses the natural tendency of real-world quantum systems to find low-energy states e.g a cooling metal bar is the real-world scenario of the function "metalbar" finding the minimal energy "cool". If an optimization problem is analogous to a landscape of peaks and valleys, for instance, each coordinate represents a possible solution and its elevation represents its energy. The best solution is that with the lowest energy corresponding to the lowest point in the deepest valley in the landscape. There are a lot of optimization problems that quantum annealing machines can solve that have a big impact on today's society e.g. traveling salesman problems and workflow optimizations.

In more precise terms, a quantum annealing system translates a Hamiltonian function that describes the total energy of a quantum system into actual physical states [9] and this process is called quantum annealing.

A Hamiltonian is the sum of the kinetic energies of all the particles, plus the potential energy of the particles associated with the system. The expression of the Hamiltonian can take different forms and simplifications taken into account the concrete characteristics of the system under analysis: single or several particles in the system; interaction between particles; kind of potential energy; time varying potential or time independent one; etc [10]

The annealing process works as follows:

- First the system's qubits are placed in an absolute energy minimum.
- The hardware alters the configuration of the system so that the energy landscape reflects the problem that it needs to solve.
- If the configuration is successful, all the qubits end up with the lowest possible energy in the new landscape.
- This process ends up identifying the lowest energy state of that landscape.

Quantum annealing processors naturally return low-energy solutions. Applications can require the real minimum energy (optimization problems) or they might require low-energy samples (probabilistic sampling problem) [11].

Sampling from many low-energy states and characterizing the shape of the energy landscape is useful for machine-learning problems, where we want to build a probabilistic model of reality. The samples give us information about the model state for a set of parameters, which can then improve the model. See Figure 1.7.

Probabilistic models explicitly handle uncertainty by accounting for gaps in our knowledge and error in data sources. Probability distributions represent the unobserved quantities in a model. The distribution of the data is approximated based on a finite set of samples.

The models are inferred from the observed data, and learning occurs as it transforms the prior distribution, defined before observing the data, into a posterior distribution, defined afterwards. If the training process is successful, the learned distribution resembles the distribution of that generated data, allowing predictions to be made on unobserved data. For example, training on MINST dataset of handwritten digits, such a model can generate images resembling handwritten digits consistent with the training set.

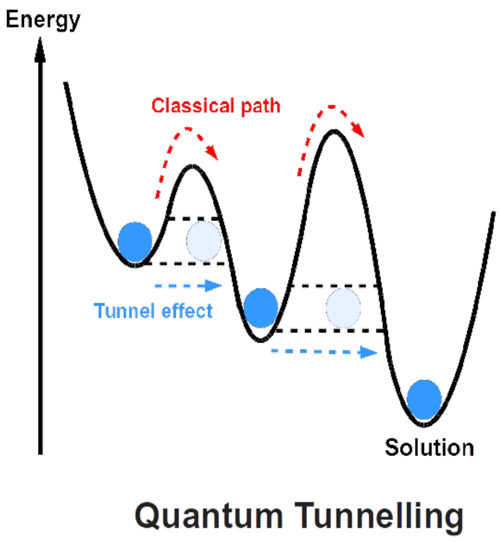


Figure 1.7: Leveraging the mechanics of quantum tunneling we can escape local minimums faster

1.4 Quantum Boltzmann Machine on the D-Wave 2000Q

The D-Wave 2000Q is a physical implementation of a quantum annealing machine with 2000 qubits and 5600 connections between the qubits called couplers.

The computer implements a Hamiltonian model with a signed state vector $s \in \{-1, 1\}$ [12, 13]. It also has the quadratic energy function (which is analogous with the energy function of a Boltzmann Machine):

$$E(s) = s^T J s + h^T s, \tag{1.4}$$

where J is analogous to the weights of a Boltzmann machine and h is analogous to its biases, see Equation 1.1 [14]. Utilizing this, we can now draw the conclusion that we can embed a Boltzmann network on the quantum annealing computer and by finding the global energy minimum we have effectively found the ideal weights for the network.

To do this we draw samples from a Boltzmann machine using the quadratic unconstrained binary optimization sampler. Samplers are processes that get the low energy states of a problem's objective function, which is a mathematical expression of the energy of a system. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by a Hamiltonian function and returns an iterator of samples, in order of increasing energy.

This machine-learning approach based on quantum Boltzmann distribution of a Hamiltonian is what we will call the quantum Boltzmann machine (QBM) [15]. The specifics of how to train a quantum Boltzmann machine is further explained in section 2.4.

Chapter 2

Methodology

This study involves two parts, the first one focuses on building a DyBM model in Python and running multiple experiments. The aim is to determine the ability of a DyBM model to memorize sequences but also to make a model that can be used as library code, that is easy to maintain, test and develop. Further, the package that is used to embed Boltzmann machines onto the quantum annealing computer is written in Python and eases a potential integration between these two libraries for future studies. The model is made available under a MIT licence [16].

The second part of this study focuses on building two QBM models. The first model is created to classify bitmap generated letters. The aim here is two test that the model is developed correctly and is able to classify letters. The second model is created to determine the ability of memorizing sequences using a QBM. The models are written in Python and also made available under a MIT licence [17].

2.1 Data sets

The data set being used are monochromatic bitmap images, created from text with the PIL package from Python. The reason this data set was chosen is three fold:

1. One reason is that it was used in the same study were the DyBM was postulated.
2. The second one is that it gives an easy way of visualising the network and how the training is progressing.

3. The last advantage is that the data set is can be quite small, something we have to take in consideration when using the D-wave 2000Q. The reason for this is that the quantum annealing computer only has 2000 qubits with 5600 couples. Since all neurons in a Boltzmann machine are fully connected this leads to one only being able to use around 25 neurons in a Boltzmann machine when embedding it into the quantum annealing computer.

An example of the training data can be seen in Figure 2.1.

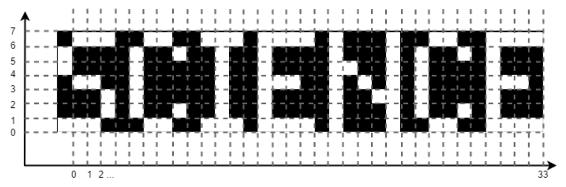


Figure 2.1: The bitmap is one of the data sets being used for developing, training and testing the qdybm. The x-axis represents the number of series a bitmap has and the y-axis represents the number of dimensions

Each bitmap is represented as a matrix of $s \cdot d$ values where s is the number of series and d is the dimensional of the data set. Each value v can have two states $\{0, 1\}$.

The values v are set in such a way that they form any shape or text. Figure 2.1 shows a bitmap with a dimension of 7 and 34 series.

2.2 Creating the DyBM Model

The first step in the study was to create a DyBM model using Python 2.7.

There are three main classes for the model are **Neuron**, **Axon**. and **Network**. To define the Axon queues, two classes are needed: Fifo and BinaryFifo. Below follows a quick description of the classes:

Neuron is the basis for the network, this hold a number of variables such as its *eligibility traces*, and *spike probability*. It also calculates the total energy of the Neuron based on the incoming and outgoing Axons.

Axon holds the connections between the Neurons and is implemented as a FIFO queue.

Network creates the architecture of the neurons and Axons. It also hold methods for training and free running the network.

Fifo, a First-in, First-out queue for holding the pre-synaptic neural eligibility traces.

BinaryFifo, a binary implementation that Fifo uses for more efficient calculation of the synaptic eligibility traces.

Implementation of DyBM

A previous study [7] describes the dynamic Boltzmann machine.

A DyBM consists of a set of neurons having memory units and first-in-first-out (FIFO) queues. Let N be the number of neurons. Each neuron takes a binary value of either 0 or 1 at each moment t, that is for $j \in [1, N]$, $x_j^{[t]}$ is the value of the j-th neuron at time t. A neuron, $i \in [1, N]$, can be connected to another neuron, $j \in [1, N]$ with a FIFO queue (synapse) of length $d_{ij} - 1$, where d_{ij} is the conduction delay from a pre-synaptic neuron i to post-synaptic neuron j. Any neuron can both be pre-synaptic or post-synaptic depending on what synapse is under consideration.

We assume $d_{i,j} \geq 1$. At each precise moment t, the tail of the queue holds the most recent value, $x_i^{[t-1]}$, and the head of the FIFO queues hold the oldest value given the length of the queue, $x_i^{[t-d_{i,j}+1]}$. For each time step, the value at the head of the FIFO queue is removed and the remaining values in the queue are pushed towards the head by one position and a new value is inserted at the tail of the FIFO queue. We allow a neuron to be connected to itself via a FIFO queue.

Neural Eligibility Traces

Each neuron stores a fixed number of neural eligibility traces, L. For $l \in [1, N]$ and $j \in [1, N]$, let

$$\gamma_{j,\ell}^{[t-1]} \equiv \sum_{s=-\infty}^{t-1} \mu_\ell^{t-s} x_j^{[s]} \tag{2.1}$$

where $\mu_\ell \in (0, 1)$ is the decay rate for the ℓ-th neural eligibility trace, which is the weighted sum of all the past value of that neuron, where the recent values have a greater weights than older ones.

Synaptic Eligibility Traces

Each neuron also stores synaptic eligibility traces, where the number of the synaptic eligibility traces depends on the number of the neurons that are connected to that neuron. Namely, for each of the pre-synaptic neurons that are connected to a post-synaptic neuron j, the neuron j stores a fixed number, K, of synaptic eligibility traces. For $k \in [1, K]$, let $a_{i,j,k}^{[t-1]}$ be the k-th synaptic eligibility traces of neuron j for pre-synaptic neuron i immediately before time t.

$$\alpha_{i,j,k}^{[t-1]} \equiv \sum_{s=-\infty}^{[t-d_{i,j}]} \lambda_k^{t-s-d_{i,j}} x_i^{[s]} \tag{2.2}$$

where $\lambda_k \in (0, 1)$is the decay rate for the k-th synaptic eligibility traces. That is, the synaptic eligibility trace is the *weighted* sum of the values that has reached neuron j, from a pre-synaptic neuron i, after the conduction delay $d_{i,j}$. Again, the recent values have greater weights than older ones.

Updating Eligibility Traces

The values of the eligibility traces stored at neuron j are updated locally at time t using the value of neuron j at time t and the values that have reached neuron j at time t from its pre-synaptic neurons. Specifically,

$$\gamma_{j,\ell}^{[t]} \leftarrow \mu_\ell \left(\gamma_{j,\ell}^{[t-1]} + x_j^{[t]}\right) \tag{2.3}$$

$$\alpha_{i,j,k}^{[t]} \leftarrow \lambda_\ell \left(\alpha_{i,j,k}^{[t-1]} + x_i^{[t-d_{i,j}]}\right) \tag{2.4}$$

for $\ell \in [1, L]$ and $k \in [1, K]$ and for each neuron i connected to the neurons j.

Learnable Parameters

The DyBM has three learnable parameters. The bias of the neurons, specifically each neuron j, is associated with a bias b_j. Each synapse (the FIFO queue between two neurons) has an associated *weight of long term potentiation* (LTP) and *weight of long term depression* (LTD).

The LTP from a pre-synaptic neuron, i, to a post-synaptic neuron j, is characterized with K parameters, $u_{i,j,k}$ for $k \in [1, K]$. The k-th LTP weight corresponds to the k-th synaptic eligibility trace.

The LTD weight from a (pre-synaptic) neuron, i, to a post-synaptic neuron j, is characterized with L parameters, $v_{i,j,\ell}$ for $\ell \in [1, L]$. The ℓ -th LTD weight corresponds to the ℓ-th neural eligibility trace. The learnable parameters are collectively denoted as θ.

Energy Definition

Similar to the conventional Boltzmann machine, the energy of the DyBM determines what pattern of values that the DyBM is more likely to generate. Contrary to the Boltzmann machine, the energy associated with a pattern at a moment depends on the patterns that the DyBM has previously generated.

Let $x^{[t]} = (x_j^{[t]})_{j\in[1,N]}$ be the vector of the values of the neurons at time t. Let $x^{[:t-1]} = (x^{[s]})_{s<t}$ be the sequence of the values of the DyBM before time t. The energy of the DyBM at

time t depends not only on $x^{[t]}$ but also on $x^{[:t-1]}$, which is stored as eligibility traces in the DyBM.

Let $E_\theta(x^{[t]}|x^{[:t-1]})$ be the energy of the DyBM at time t. The lower the energy of the DyBM with particular values $x^{[t]}$, the more likely the DyBM takes those values.

Decomposing the Energies

The energy can be decomposed into the energies of the individual neurons at time t. Specifically,

$$E_\theta(x^{[t]}|x^{[:t-1]}) = \sum_{j=1}^{N} E_\theta(x_j^{[t]}|x^{[:t-1]}) \tag{2.5}$$

The energy of neuron j, at time t depends on the value it takes as follows:

$$E_\theta(x^{[t]}|x^{[:t-1]}) = -b_j x_j^{[t]} - \sum_{i=1}^{N}\sum_{k=1}^{K} u_{i,j,k}\alpha_{i,j,k}^{[t-1]}x_j^{[t]} + \sum_{i=1}^{N}\sum_{\ell=1}^{L} v_{i,j,\ell}\beta_{i,j,\ell}^{[t-1]}x_j^{[t]} + \sum_{i=1}^{N}\sum_{\ell=1}^{L} v_{i,j,\ell}\gamma_{i,\ell}^{[t-1]}x_j^{[t]} \tag{2.6}$$

where

$$\beta_{i,j,\ell}^{[t-1]} = \sum_{s=t-d_{i,j}+1}^{t-1} \mu_\ell^{s-t} x_i^{[s]} \tag{2.7}$$

The first term of the right side of Equation 2.6 shows that a neuron having a large positive bias is likely to spike ($x_j^{[t]} = 1$) at any time t, because its energy tends to be low when it spikes. More precisely, the energy of the neuron is determined by the balance among the four terms on the right side of Equation 2.6.

The second term of the right side corresponds to LTP. Consider a pair of a pre-synaptic neuron, i, and a post-synaptic neuron, j, whose LTP weight, $u_{i,j,k}$ for $k \in [1, K]$, has a large positive value. Then j is likely to spike at time t, if the spikes from i have arrived shortly before time t, which makes $\alpha_{i,j,k}^{[t-1]}$ large for $k \in [1, K]$.

The third and fourth term can both be considered long-term depression (LTD). The third term only considers the spikes that are going to reach the post-synaptic neuron within the period of conduction delay and the last term takes into account the spikes that are arriving after the conduction delay [7].

Learning Rule

The probability distribution of the values that the DyBM generates depends on the values for the learning parameters θ.

When the DyBM is presented with the values, $x^{[t]}$ at time t, we update the learnable parameters in the direction of increasing log-likelihood.

This bias is updated as follows:

$$b_j \leftarrow b_j + \eta(x_j - \langle X_j \rangle_\theta) \tag{2.8}$$

where $\langle X_j \rangle_\theta$ denotes the expectation of the value that j generates at the time t given the values of the parameters and variables of the DyBM just before that time. Another way of saying this is: The bias is increased if the value present to the neuron is greater than what is expected, otherwise it is decreased.

The LTP weight is increased or decreased in the same way as the bias as follows:

$$u_{i,j,k} \leftarrow u_{i,j,k} + \eta(x_j - \langle X_j \rangle_\theta)\alpha_{i,j,k} \tag{2.9}$$

Now, the magnitude of the update is also proportional to the corresponding synaptic eligibility trace $\alpha_{i,j,k}$. The LTP weight is increased if the product of the presented value and the value of the synaptic eligibility trace is greater than what is expected. Otherwise, the LTP weight is decreased. Increasing $u_{i,j,k}$ results in increasing the probability that neuron j spikes particularly at the time s when $\alpha_{i,j,k}$ is high. This is what we expect with LTP.

The LTD weight is increased or decreased depending on two terms:

$$v_{i,j,l} \leftarrow v_{i,j,l} + \eta(\langle X_j \rangle_\theta - x_j)\beta_{i,j,l} + \eta(\langle X_i \rangle_\theta - x_i)\gamma_{i,j,l} \tag{2.10}$$

where β depends on the spikes traveling from neuron i to neuron j and γ represents the neural eligibility trace of neuron j. If the expected value of the first product is greater than the corresponding observed value, the LTD weight is increased. Increasing $v_{i,j,l}$ decreases the probability of that neuron j spikes at time time t when β is high. If the expected value of the second product is greater than the corresponding observed observed value, the LTD weight is also increased, which implies that the probability that neuron i spikes at t when γ is lower[7].

Parameters

The learning rate was initially set at $\eta = 1$ and then adjusted in the training using AdaGrad. During the experiments the N neurons were densely connected to each other and the conduction delay was set to 3. Each neuron was connected to itself. For all the values of the neurons, the neural eligibility traces and the synaptic eligibility traces were set to zero prior to training. The learning parameters where set independently from a normal distribution of 0.0 and standard deviation of 0.01. We implemented the algorithm for training in Python 2.7 and executed it on a Intel “Core i7” processor (4980HQ) with 16 GB of memory.

2.3 Sequence Memorization in DyBM

We did two experiments with the DyBM. The first experiment was with the data set “SCIENCE” and the second one was with the “COLABS” data set. The parameters were set as mentioned above. For training, we iteratively trained the network on sequence per time step, which meant we updated the biases and weights (LTP + LTD) and then we set the network to those sequence values by updating the neurons neural eligibility traces and propagating these values to the axons (updating the synaptic eligibility trace). This was repeated for all sequence (one epoch or period) and these epochs were shown to the network until the output of the network corresponded with training data set.

To get the output of the network it was run in a "free run" state. The weights and biases for the network were first saved. Then the first input sequence was set. The output of the network given the first input sequence was now retrieved and then the neuron eligibility traces were updated with the output sequence and the values were also propagated to the axons (synaptic eligibility trace). This is done for the entire epoch and then weights and biases are restored to continue training. Note that the saving and restoration of the weights and biases are only needed if we want to determine the progressing of the network after every epoch. We could also set the network to run a fixed number of iterations and then free run the network to determine the network state. This is however unpractical since we do not know when the network has converged but was done in the QBM.

To conclude, the difference between the training and the free-running part is that in the training part we used the current training sequence to update the neural and synaptic eligibility traces but in the free-running part we get the sequence by sampling the network and then using this sequence to update the neural and synaptic eligibility traces.

The result of the experiments are shown in Figures 3.1, 3.2, 3.3, and 3.4.

2.4 Quantum Boltzmann Machine

Two models were created with the QBM. First, a model that we used to classify bitmap letters. Secondly, a model that was used to memorize bitmap image sequences..

Learning Rule

Given a training set of state vectors, learning consists of finding weights and biases (the parameters) that make those state vectors good. More specifically, the aim is to find weights and biases that define a Boltzmann distribution in which the training vectors have high probability. By differentiating Eq. 1.1 and using the fact that $\frac{\partial E(v)}{\partial w_{ij}} = -x_i^{[t]} x_j^{[t]}$, it can be shown that

$$\left\langle \frac{\partial logP(v)}{\partial w_{ij}} \right\rangle = \langle x_i x_j \rangle_{data} - \langle x_i x_j \rangle_{model} \tag{2.11}$$

where $\langle \cdot \rangle_{data}$ is an expected value in the data distribution and $\langle \cdot \rangle_{model}$ is an expected value when the Boltzmann machine is sampling state vectors from its equilibrium distribution at a temperature of 1. To perform gradient ascent in the log probability that the Boltzmann machine would generate the observed data when sampling from its equilibrium distribution, w_{ij} is incremented by a small learning rate times the RHS of Eq. 2.11. The learning rule for the bias b_i is the same as above but with x_j omitted.

If the observed data specifies a binary state for every unit in the Boltzmann machine, the learning problem is convex: There are no non-global optima in the parameter space. However, sampling from $\langle\cdot\rangle_{model}$ may involve overcoming energy barriers in the binary state space [18].

In addition to the basic update rule in Equation 2.11, we add L2 regularization to avoid over fitting and shrink the model and momentum to help accelerating the gradient vectors in the right direction and lead to faster convergence. The final learning expression we obtain is as follows:

$$Q_{i,j}^{[:t]} \leftarrow Q_{i,j}^{[:t]} - \eta * \left(\langle x_{i,j}\rangle - \langle X_{i,j}\rangle_\theta\right) + \eta\lambda * Q_{i,j}^{[t-1]} - \mu * \left(Q_{i,j}^{[:t]} - Q_{i,j}^{[:t-1]}\right) \tag{2.12}$$

where Q is an upper triangular matrix with size NxN where the diagonals $Q_{i,i}$ are the biases and $Q_{i,j>i}$ are the weights. The first part is the local update rule from Equation 2.11, η is as before the learning rate. The second part is the momentum, λ is the momentum coefficient and $Q_{i,j}^{[:t-1]}$ are the previous gradient vectors. The last part is the L2 loss and μ is the loss coefficient.

When we are trying to learn the parameters j and h we instead use the L2 loss function.

2.5 Classification in QBM

The first step was classifying the individual bitmap letters of the sentence “SCIENCE”.

The first experiment was with an 7x4 bitmap of the individual letters. We first trained the network on the first two letter “S” and C”. In the next experiment we moved on to classify all the letters “S”, “C”, “I”, “E” and “N”.

In order to classify the QBM, we first flattened the bitmap image letters, and added 7 more bits to determine the classification. Depending on which bit was activated it symbolized a certain letter. The encoding was:

- S : 1000000
- C : 0100000
- I : 0010000
- E : 0001000

- N : 0000100

The model was trained with Equation 2.12. The first step in constructing the matrix Q which is needed for the quantum annealing machine. The sampler embeds the networks and the samples a given number of times. To increase accuracy a high number(>100) of samples are required. Executing the sampling we get a new matrix Q. Using Equation 2.12 we can then update the matrix Q until we've are satisfied with the result.

We shuffled the images every time we trained and iterated through all the images for a set number of epochs.

The mean squared error (MSE) was calculated by the expected data value x_j and the sampled expected value $\langle X_j \rangle$. The accuracy was determined from the ratio of correctly predicted observations to the total observations. The results can be found in Figures 3.5, 3.6, 3.7 and 3.8.

2.6 Sequence Memorization in QBM

First, we created an experiment comprising a small network, namely a 7x6 matrix of the "SCIENCE" data set, but only using the first letter "S" and using only the basic learning rule without L2 and momentum.

After create the Q matrix and use the sampler to embed and get a new Q matrix back. We then used 2.11 to update the Q matrix. 200 sample reads were used for every sampled with an annealing time of 5 microseconds.

To do a similar "free run" simulation of the network as we did with DyBM proved difficult. Since we do not have direct access to the q-bits in the D-Wave machine this has to be done by adjusting the bias (the idea here is that by adjusting the bias we can increase the probability of the state we want). To activate a neuron we increased that the bias of that neuron and vice versa. The increment was relative to the existing bias by:

$$h_i \leftarrow h_i \pm \frac{1}{N} \tag{2.13}$$

where h_i is the bias for the neuron i and we increase or decrease the bias by a magnitude $\frac{1}{N}$ where N is the number of neurons.

However this is method is not ideal and there are other way to go about this. E.g. choosing a fraction of $\frac{1}{N}$ was completely arbitrary and both linear and exponential adjustment were tried, both with similar results.

This algorithm showed no convergence for any training sequences using this approach and no meaningful results could be found. This is believed to be to the fact that there is no way of simply "setting a state on a quantum Boltzmann machine. The crude way of changing the bias was simply not enough and more work is needed to be done in this field in order to fully understand how this could be possible.

Chapter 3

Results

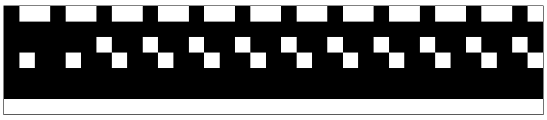
3.1 "SCIENCE" Data set DyBM

The results for the DyBM with the data set "SCIENCE" are shown in Figures 3.1 and 3.2. The execution time was 202 seconds.

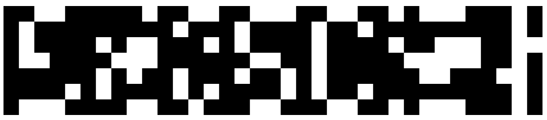
Optimization

Due to severe performance loss in contrast to the Java implementation (200x slower) several optimization where done to the code, most notably Numba was used to optimize matrix computations. Numba translates Python functions to optimized machine code at run time using the LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.[19]

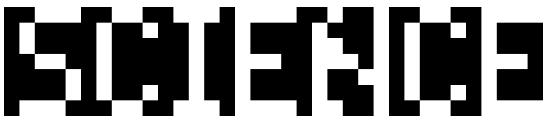
Even with an heavily optimized code the performance loss was still severe (100x slower).



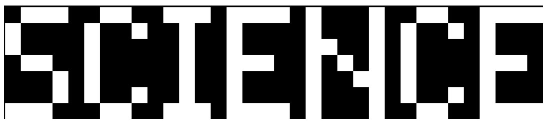
(a) Epoch 0



(b) Epoch 1700



(c) Epoch 2500



(d) Epoch 3080

Figure 3.1: The output of DyBM after *n* epochs of training with the data set "SCIENCE". The output starts of repeating and with no structure from the input data and progressively gets better. By Epoch 3080 the output has converged.

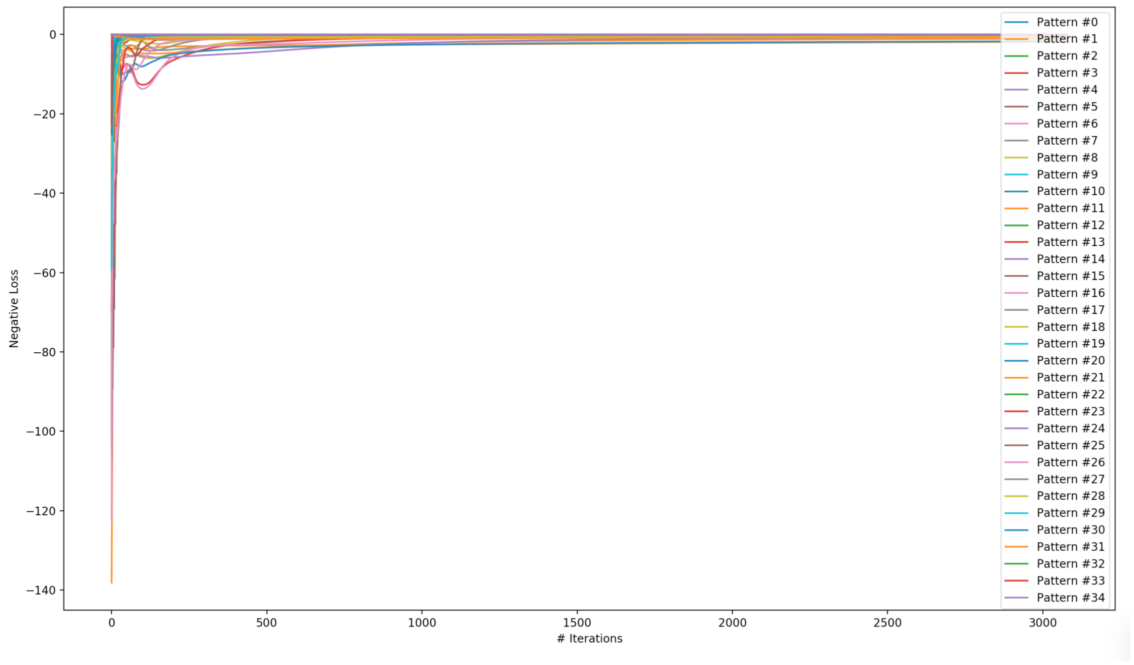
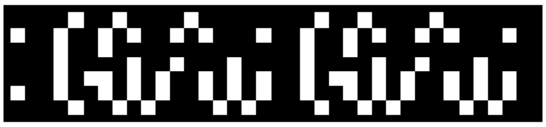


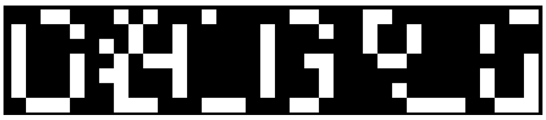
Figure 3.2: The plot show how the loss decreases in the network over training epochs for the data set "SCIENCE".

3.2 “COLABS” Data set

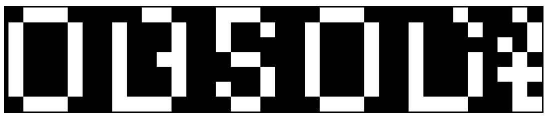
The results for the DyBM with the data set “COLABS” are shown in Figures 3.3 and 3.4. The execution time was 5 seconds.



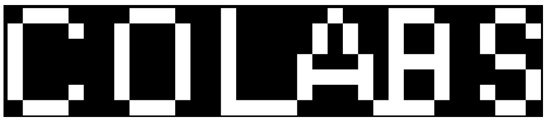
(a) Epoch 0



(b) Epoch 100



(c) Epoch 200



(d) Epoch 275

Figure 3.3: The output of DyBM after *n* epochs of training with the data set “COLABS”. The output starts of repeating and with no structure from the input data and progressively gets better. By Epoch 275 the output has converged.

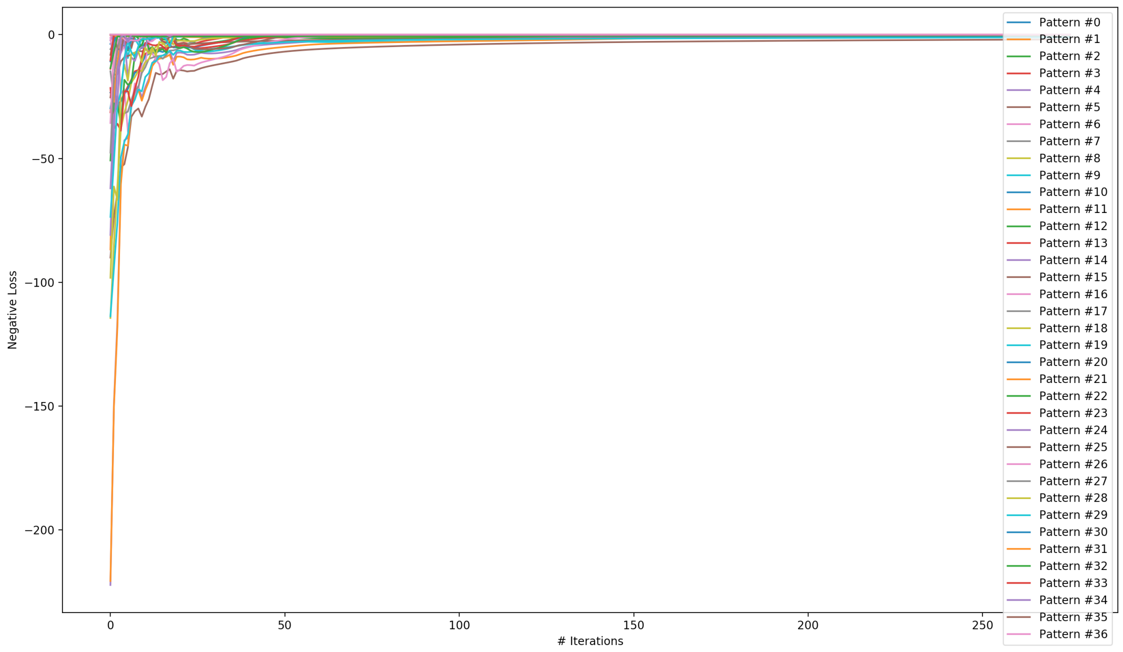
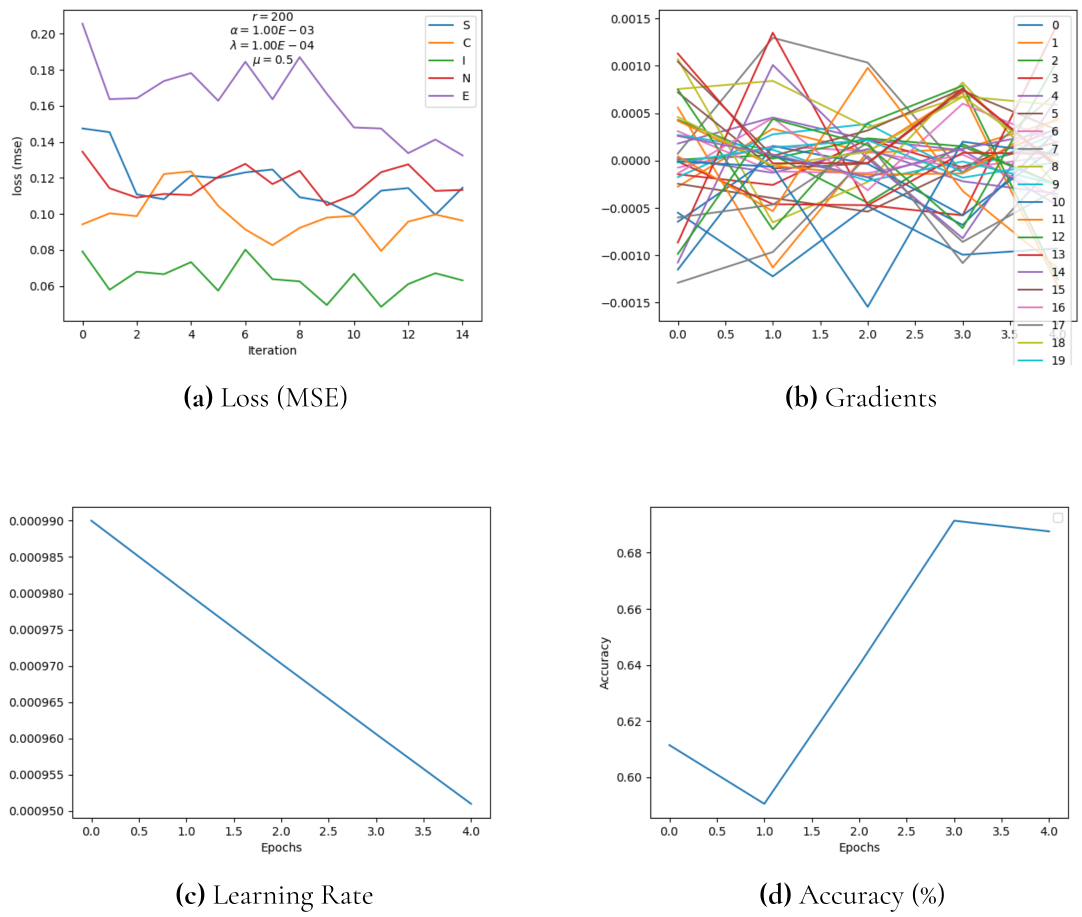


Figure 3.4: The plot show how the loss decreases in the network over training epochs for the data set “COLABS”.

3.3 QBM

3.3.1 Classification SCIENCE

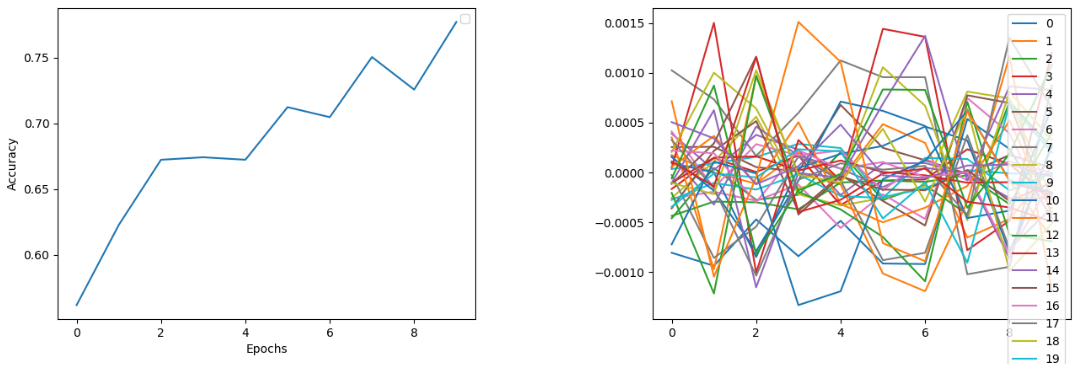
In this section we train the QBM to classify all the letters in SCIENCE, that is “S” “C” “I” “E” “N”. We did 4 experiments with increasing epochs (from 4 up to 25). The results shows that increasing the epochs also increase the accuracy (from 0.69 to 0.79) but changing any of the learning parameters could change the results and a lot of work went in to finding the chosen parameters. We would like the number of samples taken (r = 200) to be higher but due to cost and time restrictions of qpu we could not increase it. The Learning rule chosen was a simple linear rule, other rules were tried but did not seems to change the outcome significantly. Momentum was chosen to be $\mu = 0.5$ and proved to provide a good balance between speed of converging and oscillation.



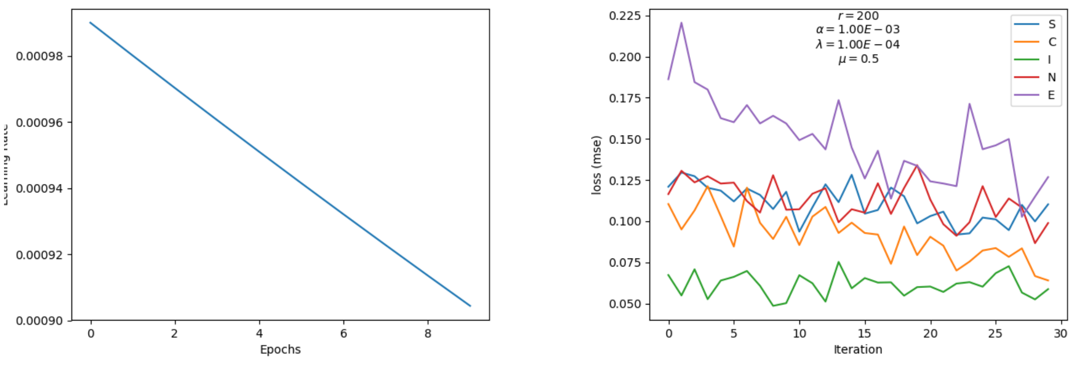
(a) Loss (MSE) **(b)** Gradients

(c) Learning Rate **(d)** Accuracy (%)

Figure 3.5: Total epochs trained: 4. Best Epoch was: 3. Total qpu access time: 3.0s. Best Accuracy: 0.69. Total iterations: 20

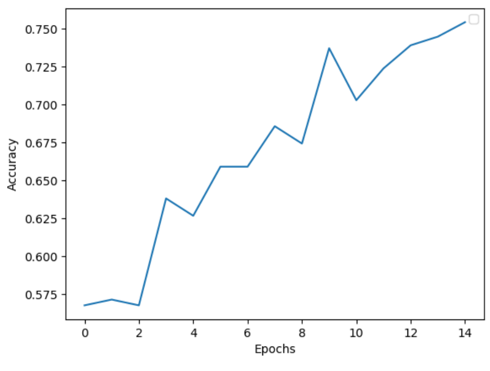


(a) Accuracy (b) Gradients

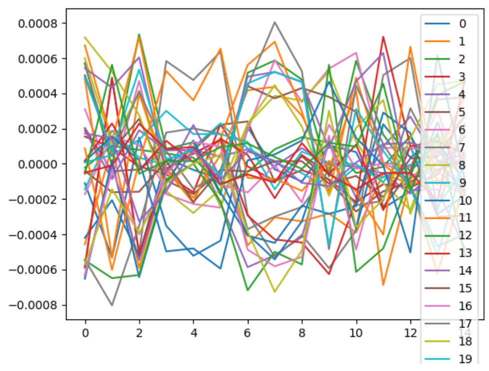


(c) Learning Rate (d) Mse

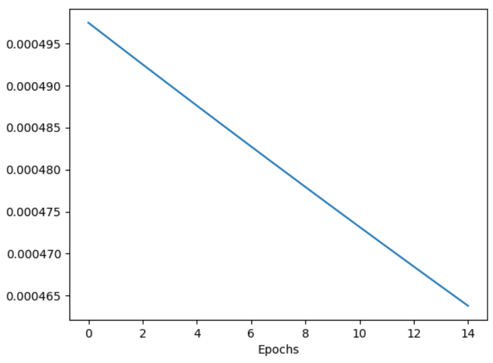
Figure 3.6: Total epochs trained: 10. Best Epoch was: 10. Total qpu access time: 6.1s. Best Accuracy: 0.79. Total iterations: 50.



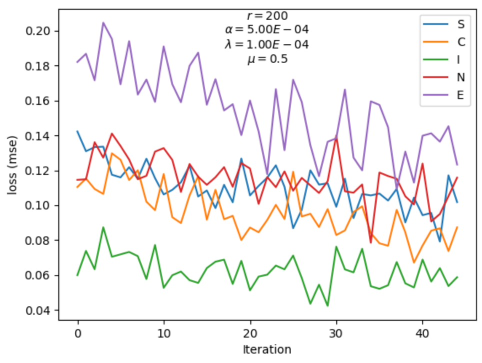
(a) Accuracy



(b) Gradients

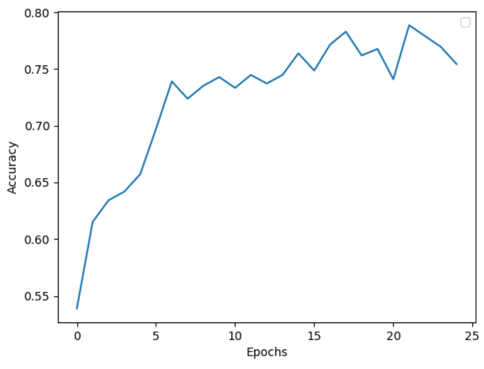


(c) Learning Rate

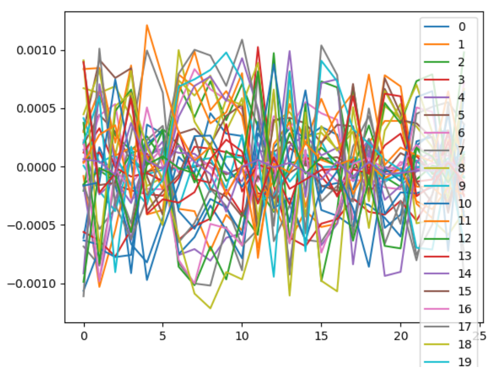


(d) Mse

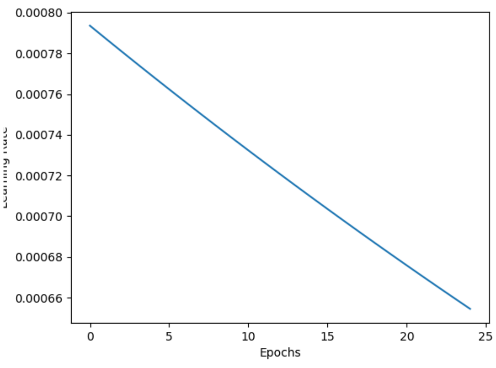
Figure 3.7: Total epochs trained: 15. Best Epoch was: 13. Total qpu access time: 9.1s, Best Accuracy: 0.76. Total iterations: 75



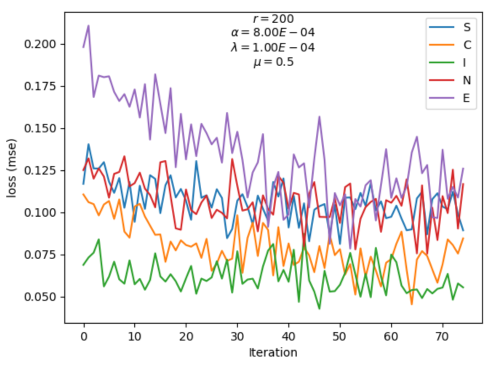
(a) Accuracy



(b) Gradients



(c) Learning Rate



(d) Mse

Figure 3.8: 25 epochs trained. Best Epoch was 16. Total qpu access time: 15.18s, Best Accuracy: 0.79. Total iterations: 125

Chapter 4

Discussion

QBM Sequence Memorization

There are many reasons to investigate why the QBM never converged. One reason relates to the setting of the neuron values between sequences. From previous experiments done outside of this project, it was shown that the QBM had the capability of classifying data, successfully. However, this was done by utilizing extra bits to represent the classification. To then obtain the classification we would set the biases of the bits to the same classification sequence as the classification we would like to obtain. However, in the memorization sequence algorithm, we want to use the same bits to obtain the next sequence and this is probably where the issues arise. Since we have to adjust the biases to set the bits on and off, but we still need them to keep the network we run the possibility of disturbing the network and not find a solution. One way of mitigating this would be to add a series of bits representing what order in the sequence the current sequence hold, and thus in the "free run" phase we could get the correct sequences in the correct order. However, this would then train the network in several classifications rather than a correlation function between the different sequences.

QBM Classification

The results of the classification show promising results with accuracy reaching about 80 %. One of the biggest issue with classifying the letter was the big changes in the accuracy due to the big variation of the quantum sampling. To mitigate this we tried different numbers of reads, but the limitations of training time (due to cost allocation) made it not possible to increase the number of sample reads more than around 500 per iteration. One of the key factor for training successfully was adding momentum to the learning equation. The optimal value for the momentum constant was found to be 0.5, the higher it was set the more the model had a tendency to oscillate and become unstable. If we had access to more training time we would have increased the number of samples for each iteration to around 1000ms as well as increasing the annealing time to 100ms.

DyBM Sequence Memorization

The DyBM implementation also has opportunities for improvement. As mentioned before there were many iterations of performance enhancement done to the algorithm to achieve somewhat reasonable levels of execution time, unfortunately even after using a lot of optimization techniques and converting a lot to C code with the help of Numba, speeds were still slow. This we assume is due to the innate nature of Python as a dynamic language and not pre-compiled. We also could have increased efficiency a lot if we would not have followed object-oriented programming since Numba, for the time being, does not support classes to be executed as "no-python" mode. The system in Python ended up not being very scalable because of its lack of performance, restructuring the code we could use Numba to its full extent the algorithm could become more scalable.

The conversion from Java to Python eventually ended up much longer than anticipated due to a large amount of small differences in error handling in Python and Java leading up to many bugs that were hard to track.

From the results, we can also see that the loss function of the DyBM abruptly diminishes but that then it's almost like the gradient vanishes and we get stuck in a local minimum for a long time. Solutions to this could be gradient clipping or possibly changing some parameters. To change all the parameters we could do a hyper-parameter search for different queue lengths, decay rates and so on.

With more experiments done we could set up more of a baseline, we could, for example, see that even though both data sets we used in DyBM had the same length, the training time greatly differed (11x slower for the "SCIENCE" data set). We would like to further study how similar sequences with the same length affect the loss and training time, how the length of the sequences affects training time and how different parameters affect the result.

4.1 Outlook

- Converging QBM sequence Memorisation: A key factor that made the QBM not converge was the ability to accurately run the free run procedure on the QBM. To set a number of bits active this was done by increasing the bias of a particular q-bit, but the question still remains with how much the bias should be increased.

- Optimization: As mentioned above, the Python version of the DyBM is still much slower than the corresponding Java version and would need to become faster.
- Expand data sets: Using more data sets would enable us to study the performance of the sequence memorization in more depth. In particular we would like to study networks of different sizes. E.g with MIDI sequences one neuron for every note we would almost double the network size.
- Different Gradient Descent algorithms: Currently we use AdaGrad for the DyBM. It would be of interest to see how the training behaves with different algorithms such as RMSProp.
- Hybrid algorithm - Quantum Dynamic Boltzmann machine: Creating a hybrid algorithm that uses the neural and synaptic eligibility traces as well as LTD and LTP would be ultimate goal. The issue here lies in that the energy model for a QBM and a DyBm are different and that the sampling is done in a QBM. This leads to some problems such as how are all the eligibility traces and weights going to be updated. One viable solution would be to use the QBM as a sampler in the original DyBm.
- Elman networks: Elman Networks are recurrent and specifically designed to learn sequential or time-varying patterns. We would like to compare the results of this architecture to the one we have studied in this report.

Bibliography

[1] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. New York: Wiley, Jun. 1949.

[2] G. Hinton, "Boltzmann machines," *Encyclopedia of Machine Learning and Data Mining*, pp. 1–7, 2014.

[3] S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Jr. Vecchi, "Optimization by simulated annealing," *Science (New York, N.Y.)*, vol. 220, pp. 671–80, 06 1983.

[4] M. H. Amin, E. Andriyash, J. Rolfe, B. Kulchytskyy, and R. Melko, "Quantum boltzmann machine," *Phys. Rev. X*, vol. 8, p. 021050, May 2018. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.8.021050

[5] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in Neuroscience*, vol. 10, p. 508, 2016. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2016.00508

[6] D. E. Feldman, "The spike-timing dependence of plasticity," *Neuron*, vol. 75, no. 4, p. 556–571, 2012.

[7] T. Osogami and M. Otsuka, "Seven neurons memorizing sequences of alphabetical images via spike-timing dependent plasticity," *Scientific reports*, vol. 5, p. 14149, 09 2015.

[8] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2019.

[9] N. Chancellor, "Domain wall encoding of discrete variables for quantum annealing and qaoa," *Quantum Science and Technology*, vol. 4, no. 4, p. 045004, Jun 2019.

[10] D. McMahon, *Quantum Mechanics Demystified 2/E*. McGraw-Hill Professional Pub., 2013.

[11] "Introduction to quantum annealing." [Online]. Available: https://docs.dwavesys.com/docs/latest/c_gs_2.html

[12] "Getting started with the d-wave system." [Online]. Available: https://docs.dwavesys.com/docs/latest/doc_getting_started.html

[13] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse ising model," *Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics*, vol. 58, 04 1998.

[14] V. Dumoulin, I. Goodfellow, A. Courville, and Y. Bengio, "On the challenges of physical implementations of rbms," *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, 12 2013.

[15] M. H. Amin, E. Andriyash, J. Rolfe, B. Kulchytskyy, and R. Melko, "Quantum boltzmann machine," *Physical Review X*, vol. 8, no. 2, 2018.

[16] C. Arnlund, "pdybm." [Online]. Available: https://github.com/ChristofferArnlund/pydbm

[17] ——, "Python implementation of quantum boltzmann machine." [Online]. Available: https://github.com/ChristofferArnlund/qbm

[18] D. Ackley, G. Hinton, and T. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive Science*, vol. 9, no. 1, p. 147–169, 1985.

[19] "A high performance python compiler." [Online]. Available: https://numba.pydata.org/

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-11-08

MASTER'S THESIS Sequence Memorization in Dynamic & Quantum Boltzmann Machines

STUDENT Christoffer Arnlund
SUPERVISOR Pierre Nugues (LTH)
EXAMINER Volker Krueger (LTH)

Sequence Memorization in Dynamic & Quantum Boltzmann Machines

POPULAR SCIENCE SUMMARY **Christoffer Arnlund**

Memorizing sequences is useful for many applications such as *natural language processing* and *anomaly detection*, but can also be used generatively, e.g. to generate missing part of a photograph or music sheet. In this work, we study the performance of memorizing sequences in *dynamic Boltzmann machines* and *quantum Boltzmann machines*.

A *artificial neural network* (ANN) can be described as a system that learns to perform a task by repeated exposures to examples of the task.

There are many different types of architectures for an ANN such as feed-forward neural networks, recurrent neural networks, convolutional neural networks and energy-based models (EBM). EBM's are generative models and can be used to model the unknown distribution of some data such as images and text. A certain implementation type of an EBM is called a *Dynamic Boltzmann network*. This EBM can be either implemented in a classical computer or as a *Quantum Boltzmann Network*, meaning that the algorithms are used on a quantum computer. A quantum computer is a type of computer that uses quantum mechanics so that it can perform certain kinds of computation more efficiently than a regular computer can.

In this report, we are focusing on training Dynamic Boltzmann networks and Quantum Boltzmann Networks to memorize sequences. Memorizing sequences are useful for many real-world applications today such as natural language processing, DNA sequencing, anomaly detection and it can be used to generate new information like music and art.

We build a Dynamic Boltzmann Machine model in Python and run multiple experiments. The aim is to determine the ability of a DyBM model to memorize sequences but also to make a model that can be used as library code, that is easy to maintain, test and develop. The data set used is shown in Figure 1 and the results are shown in Figure 2.

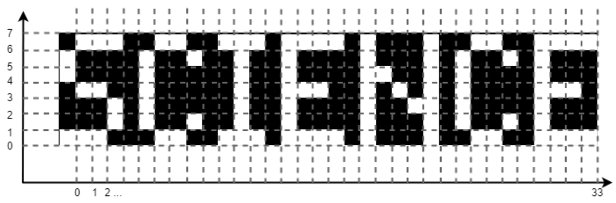


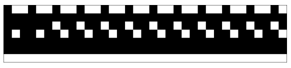
Figure 1: The bitmap is one of the data sets being used for developing, training and testing both models. The x-axis represents the number of series a bitmap has and the y-axis represents the number of dimensions.

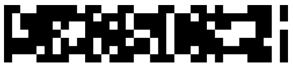
The second part of this study focuses on building two Quantum Boltzmann Machines. The first model is created to classify bitmap generated letters. The aim here is to test that the model is

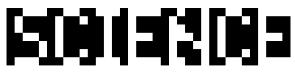
INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-11-08

MASTER'S THESIS Sequence Memorization in Dynamic & Quantum Boltzmann Machines

STUDENT Christoffer Arnlund
SUPERVISOR Pierre Nugues (LTH)
EXAMINER Volker Krueger (LTH)


(a) Epoch 0


(b) Epoch 1700


(c) Epoch 2500

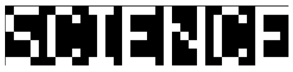
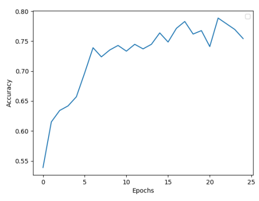
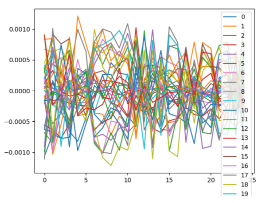

(d) Epoch 3080

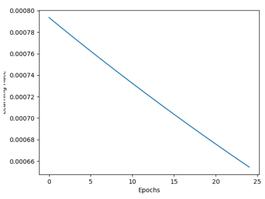
Figure 2: The output of Dynamic Boltzmann Machine after n epochs of training with the data set "SCIENCE". The output starts of repeating and with no structure from the input data and progressively gets better. By Epoch 3080 the output has converged.

developed correctly and is able to classify letters. The second model is created to determine the ability to memorize sequences using a Quantum Boltzmann Machine. The results for the classification experiment is shown in Figure 3.

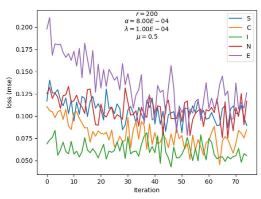


(a) Accuracy


(b) Gradients



(c) Learning Rate



(d) Mean square error

Figure 3: 25 epochs trained were trained. The best Epoch was 16. Total quantum processor unit access time: 15.18s, Best Accuracy: 79%. Total iterations: 125

From the results, we can see that the Dynamic Boltzmann Machines successfully manages to memorize sequences. However, due to the implementation being done in python several performance bottlenecks limited the size of the data set. Having chosen python for its accessibility, however, limited the networks performance. Would we have chosen a less accessible language such as `C++` we could potentially have had greater performance and is subject for another study.

For the Quantum Boltzmann Machines, we successfully trained the model to classify bitmap generated letters with accuracy reaching about 80 %. One of the biggest issues with classifying the letters was the big changes in the accuracy due to the big variation of quantum sampling.

When training the Quantum Boltzmann Machine to memorize sequences the Quantum Boltzmann Machine never converged. The reason for this is believed being the inability to set the biases for the different qubits of the quantum computer correctly. A limiting factor here was the access time of the quantum computer since it is very costly to run. With more time we believe that the biases could be set correctly.

Other topics that can be explored is increasing the number of data sets used, trying different optimization algorithms and optimize the Dynamic Boltzmann Machine code. Another topic that can be explored is Elman Networks. They are recurrent and specifically designed to learn sequential or time-varying patterns. Comparing the results of this architecture to the one we have studied in this report would be of interest.