Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Boltzmann Machine

Seminar Paper

Neural Networks

Michael Blesel, Oliver Pola
Matr.Nr. 6443269, 6769946
{3blesel, 5pola}@informatik.uni-hamburg.de

July 19, 2020

Abstract: The Boltzmann machine is a stochastic recurrent neural network with symmetrical connections. Its idea is based on models from physics, like the Ising model. It is similar to a Hopfield network, but with a hidden layer of neurons. Whereas most recent publications deal with the restricted Boltzmann machine, in our work the general case is discussed and implemented. We describe the concept and the learning algorithm in detail, arranged in multiple phases: setup, positive (using the inputs), free-running (let loose the inputs) and updating the weights. After the Boltzmann machine is trained, its application uses a recall phase, where it can produce output according to the distribution of inputs it has seen. These phases need a lot of computations and thus the training runs for a long period. One approach to improve the efficiency is to replace the asynchronous update, used in the concept, by a synchronous update that we do implement. With an experiment based on images of handwritten digits from the MNIST dataset, we show that our implementation is able to recall a certain pattern among multiple trained ones. Our results include an evaluation, how the number of epochs during training does influence the quality of the results. Another parameter, the generalized concept of temperature, is analyzed. The discussion of our results shows that it is difficult to tune the parameters to a growing problem size, increasing the number of images. In general the Boltzmann machine takes a long time to train and it is necessary but hard to tune the parameters to the specific problem. We conclude that the Boltzmann machine is a different but interesting approach to machine learning and further studies would be beneficial.

Contents

1 Introduction

The idea of a Boltzmann Machine (BM) is from Ackley, Hinton and Sejnowski, 1985 [1]. Similar to neural networks in general, the theoretical background is quite old, but could not have been used back then due to the lack of computational power. Now, that we have the necessary hardware, it is time to revive those ideas. But opposed to neural networks that are heavily used today, the BM did not gain that much attention.

The theoretical background of the BM is inspired by physics, especially statistical mechanics. The Ising model [6] describes ferromagnetism with an ensemble of atoms arranged in a lattice. These atoms have spins that are described having a state σ_i of either +1 or -1. A configuration σ assigns a value to each of the states. The total energy of the system can be described by its Hamiltonian $H(\sigma)$ that is based on the configuration and considers interactions between atoms and a possible external magnetic field. The interactions are usually modeled between neighbors on the lattice only. The probability $P(\sigma)$ that the system is in a certain configuration σ is described by the Boltzmann distribution

$$P(\sigma) \propto e^{-\frac{H(\sigma)}{k_B T}}$$

where T is the temperature and k_B is the Boltzmann constant. To achieve such a distribution by simulation on the lattice, the Metropolis algorithm [9] can be used. This randomly chooses lattice points and calculates the energy if the state was flipped. If the new energy is lower, the state is accepted. Otherwise the higher energy state is accepted with probability p based on the energy difference ΔE and the temperature.

$$p = e^{-\frac{\Delta E}{k_B T}}.$$

These states don't have to be atom spins and this principle could be applied to other topics than physics. Also the nodes could be arranged in some other network and not necessarily a lattice. And the energy could be generalized to a more abstract minimization problem. This is the basic idea that the BM tries to apply to neural networks.

2 Related Work

The BM is not the only neural network based on the physics model. There is also the earlier idea of a Hopfield network [5][10, Chapter 13][8, Chapter 43]. Whereas Hopfield networks do not have hidden units, this is the case for a BM. So a BM can also be considered to be a successor to a Hopfield network.

Being based on the theoretical background of the original BM paper from Ackley, Hinton and Sejnowski [1], we provide an implementation in Python. Our code is similar to and based on the code by Cartas [2], but we tried to optimize that code, make it more flexible by parameterization and add a new feature, the synchronous update.

In opposition to most recent work like Salakhutdinov, Mnih and Hinton [11], Hinton [4], Sutskever, Hinton and Taylor [13] or Courville, Bergstra and Bengio [3], we do not deal with the restricted BM (RBM), but with the general case.

3 Boltzmann Machine

In this chapter, we give a general explanation of the BM based on [1] and [8, Chapter 43]. We go into the structure of the network, how the learning and recall phases work and talk about the differences between the asynchronous and synchronous update procedures.

The BM is a stochastic recurrent neural network with symmetrical connections. Rojas [10, Chapter 14] provides an introduction to stochastic networks. A BM is used to extract key features from given training data by unsupervised learning.

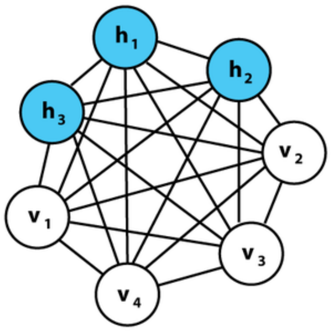

Figure 1: Representation of a small Boltzmann machine [12]

In Figure 1 we can see the structure of a small BM. It consists of two layers of units which are called the visible (v) and hidden (h) layer. All units in the BM are fully connected with symmetrical weights, which means that value of the weight from any units A to B is identical to the weight from B to A. All units in a BM have a binary state of either one or zero, indicating if they are currently activated or not. The weights can take any positive or negative real value.

The BM is used for unsupervised learning and has no strictly defined output layer. The visible layer is used during training to hold the input data and after learning the free running phase of the machine creates a binary vector of the states of the visible units which can be seen as the output. The general idea behind the BM is to feed it with binary input vectors from the training dataset during the learning phase in which the network learns the probability distribution of the set input units, which will enable it to reproduce patterns of visible units that will match the learned probability distribution in its recall phase. A detailed explanation of the learning and recall phases follows.

3.1 Learning Phase

Before we present the learning algorithm of the BM, we need to introduce the concept of 'clamped' and 'unclamped' units. As mentioned before, during the learning phase the visible units are set to the values of the binary input vectors from the training data. This procedure is called 'clamping' the visible units. This means, that during the positive phase of the learning algorithm, the states of these units will not change and they will influence the computed states of the hidden units.

In Figure 2 we present pseudo code for the basic learning algorithm. This is a simplified version of the algorithm from Cartas [2]. To best explain the algorithm, we are going to divide it into four logical parts: Setup Phase, Positive Phase, Free-Running Phase and Updating Weights.

1. **Setup Phase:**
   Before the learning algorithm can start, we need to set up some parameters. Firstly, the weight matrix is initialized with zeros. Intentionally we do not initialize with random values, as it is often done for other neural networks. This is necessary, since the weights need to be symmetric for all pairs of units. The next thing to do, is setting values for the temperature, learning and co-occurrence epochs. The learning and co-occurrence epochs parameters will be explained in the description of the following phases.
   After that the core algorithm starts. The first thing we can see in line 4 is that the whole learning procedure (the next three steps) is supposed to be executed until the values of the weight matrix have converged and no more changes are observed. From a theoretical point of view this happens when the equilibrium state has been reached. In practical implementations the while loop can be replaced by a for loop with a fixed number of iterations.

2. **Positive Phase:**
   The actual learning algorithm starts with the so called positive phase. In this phase the given input patterns are learned. First the units of the visible layer are set to the values of the current input vector and all visible units are clamped. All unclamped units (in this case the units of the hidden layer) are initialized with random binary values. Now the actual updating of the states takes place by calling the update function as many times as the learning epochs variable has been set to. This is another parameter that has to be set before the start of the learning algorithm and which has to be tuned to fit the problem.
   In the update function, the stochastic update of the states of the unclamped units takes place. The update calculations are done step-by-step for randomly chosen unclamped units. This version of the update procedure is called asynchronous and we will compare it to another possible version called the synchronous update, where the updates do not take place in order but are computed all at once in Section 3.3.

```
#Set up phase
SET learning_epochs, co-occurrence_epochs, temperature T
SET weight matrix W = 0
LEARNING():
  WHILE W continues to change DO:
    #Positive phase
    SET the sum co-occurrence matrix S = 0
    FOR EACH training pattern x:
      SET visible units to x and CLAMP them
      SET hidden units to random binary states
      CALL UPDATE-STATES(learning_epochs)
      CALL COLLECT-STATISTICS()
    SET $P_{ij}^{+}$ = average $S_{ij}$ over all training runs and patterns
    #Free-running phase (negative phase)
    SET states of all units to random binary values and UNCLAMP them
    CALL UPDATE-STATES(learning_epochs)
    SET the sum co-occurrence matrix S = 0
    CALL COLLECT-STATISTICS()
    SET $P_{ij}^{-}$ = average $S_{ij}$ over all training runs
    #Update weights $W_{ij}$
    Add $\Delta W_{ij} = k(P_{ij}^{+} - P_{ij}^{-})$, where k is a small constant

UPDATE-STATES(epochs):
  FOR 1 to number of epochs:
    FOR 1 to number of unclamped units:
      randomly select an unclamped unit i
      Calculate its energy difference: $\Delta E_i = \sum_j W_{ij} x_j$
      #Stochastically decide the resulting state
      IF random uniform $U[0,1] < \frac{1}{1+e^{-\Delta E_i/T}}$:
        SET $x_i = 1$
      ELSE:
        SET $x_i = 0$

COLLECT-STATISTICS():
  FOR 1 to number of co-occurrence_epochs:
    FOR 1 to number of unclamped units:
      randomly select an unclamped unit i
      Calculate its energy difference: $\Delta E_i = \sum_j W_{ij} x_j$
      #Stochastically decide the resulting state
      IF random uniform $U[0,1] < \frac{1}{1+e^{-\Delta E_i/T}}$:
        SET $x_i = 1$
      ELSE:
        SET $x_i = 0$
  FOR EACH pair of connected units:
    SET $S_{ij} = S_{ij} + x_i x_j$
```

Figure 2: The learning algorithm

For each chosen unit at first its energy difference is calculated with the formula

$$\Delta E_i = \sum_j W_{ij} x_j.$$

This is similar to other networks and lets the weights W_{ij} and state x_j of other units j influence the current unit i. Based on the energy difference ΔE_i then the probability p_i is calculated, that the state of unit i is 1 afterwards.

$$p_i = \frac{1}{1 + e^{-\Delta E_i / T}}.$$

The probability can then be evaluated, checking a random uniform number u in range $[0, 1]$ against p_i. If $u \leq p_i$, then set $x_i = 1$, otherwise set to 0.
Having a large weight between units i and j would cause a higher energy difference if unit j is already set to state 1. This would lead to a higher probability for unit i to become 1 as well. This should enable to memorize, that both units have often be seen together in state 1. A negative weight would cause the opposite effect and enables to express both units have rarely be seen together in state 1. Of course a state of 0 would also influence the sum and a missing term can increase or decrease the energy difference and thus the probability in a similar manner.
Also notice, that for a very high temperature T, the probability p_i becomes 0.5, so both states 0 and 1 will be equally likely. This means high temperature leads to randomly changing states. That is not good for a final state, but helpful to explore.

This whole procedure is repeated for all given input patterns. The last step of the positive phase is the collect statistics function which will be explained in the 'updating the weights' segment.

3. **Free-Running Phase:**
After the positive phase the negative or 'free-running' phase starts. This phase is different to the positive phase in the fact that now all units of both layers are treated as unclamped. This means that also the visible layers units states can be updated. Since no more input patterns are given to the visible layer all units are initialized with a random binary value and the network is run freely. In this phase, the previously learned information, which is stored in the weights, is analyzed by the network. Found correlations between for example two connected units often being active at the same time are strengthened. The process of finding a more optimal low energy state mainly occurs in this phase.
The free-running phase is also later used to produce the output patterns of the BM that a user will look at when applying the network on a real problem.

4. **Updating Weights:**
The updating of the weights during the learning phase is done by comparing statistics of so called co-occurring units in the positive and in the negative

phase. A co-occurrence is the case that two units i, j connected by a weight w_{ij} are both observed in the state 1. In the pseudo code of the algorithm these co-occurrences are summed up in the vector S_{ij} (line 45), where the field ij is increased by one every time the units i, j have both been observed in the state 1. Since we are dealing with a stochastic process here it is necessary to collect these statistics over multiple epochs and to average the results afterwards. This can be seen at the definitions of the P^+_{ij} and P^-_{ij} variables. P^+_{ij} holds the co-occurrence statistics for the positive phase and P^-_{ij} holds the statistics for the negative phase. It is important to recognize that the collection of the statistics in the algorithm only starts after a good amount of update steps have already been done for each phase. This is needed because we can only make meaningful assumptions about co-occurrences after each phase had a little time to already move towards converging on a low energy state. The 'collect-statistics' function then repeats a number of update steps during which all co-occurrences are summed up.
The actual changing of the weights happens after each time a pair of positive and negative phases has finished. Here the values of P^+_{ij} and P^-_{ij} are compared for every connected pair of units and their weight is increased by a set learning factor k if there where more co-occurrences observed in the positive phase over the negative phase or it is decreased if less co-occurrences where observed in the positive phase. This process reinforces the learned co-occurrence information from the input patterns. If during the positive phase where the visible layer is clamped to a specific pattern two units often showed a co-occurrence that means that this information should be learned and it will therefore be reinforced by changing their weight correspondingly. If however in the free-running phase two units are co-occurring more often than in the positive phase this does not represent the learned data properly and the probability of this co-occurrence has to be decreased by a corresponding change of weights.
This whole process of collecting the co-occurrence statistics and then updating the weights is done for every iteration of the learning algorithm. It leads to a network that in its free-running phase will produce visible layer patterns that represent the distribution of the learned input patterns as closely as possible.

Finally, the learning phase iterates in a loop over positive phase, the free-running phase and updating the weights. In theory this should be done until the weights converged. In practice the number of iterations will be fixed to guarantee termination within a reasonable time frame.

3.2 Recall Phase

What we call the recall phase in this paper, describes using the previously trained BM to create output patterns. As the BM does not have an output layer defined, the so called outputs of the machine are considered as the patterns in the visible layer which are created when a free-running phase is started on the trained network. There are multiple ways in which we can use this recall to produce meaningful results. The simplest one is to just let the machine run a few epochs in a free-running phase with all units being unclamped. In this configuration the basic idea behind the BM comes through and the different resulting patterns in the visible layer and their probability of occurrence should resemble the probability distribution of the learned input data. For example if the network is trained with a dataset that contains one specific pattern five times and another pattern only once the probability of the recall to produce the first pattern should be five times as high as the probability of producing the second pattern. Due to the stochastic nature of the BM it is of course more likely that the resulting patterns from the recall might not exactly match one of the two inputs but only closely resembles them. This behavior of course gets more common as the size of the input data increases. With this use case, we can already see how the BM could be used to extract key features from input datasets. The results might not reproduce one distinct input but they should with high probability create an output that strongly features commonalities between the different inputs.

A different approach of using the recall phase is not to start a completely free-running phase but to also add some incomplete input data to the visible layer. This can be achieved by setting the states of some visible units to fixed values and then clamping those units before executing the update procedure. A simple example for this is the reconstruction of images with broken or missing pixels. A BM is first trained with a set of images and in the recall phase is given a partial input image. Some of the visible units are initialized with the data from the partial image and clamped. The rest of the visible layer and the whole hidden layer are left unclamped and the update procedure is started. After running for a few epochs the resulting states of the visible layer are taken as the outputs and they should resemble a reconstructed version of the whole image with good probability.

A final use case for the recall phase is to create an artificial output layer for the network by dividing the visible layer into two distinct parts at the creation of the network. This has for example been done in the original paper on the BM [1]. Here the task was to solve a simple pattern matching problem called 'the encoder problem'. Without going into too much detail the idea was that there are input patterns of n bits that match to specific output patterns of also n bits. To learn this problem the general BM has to be slightly modified. At creation the visible layer is divided into two n unit layers which will not be interconnected but only have connections inside themselves and to the hidden layer. The input layer so to speak communicates with the output layer through the hidden layer. This approach could probably also be used to solve classification problems but it is questionable if the BM would be a good choice for this since there exist a lot of other networks and algorithms which solve these problems more efficiently.

After describing the different possible use cases for the recall procedure we take a look at the recall algorithm. The algorithm itself is not much different from the procedures we have already seen in the learning section. The basic idea is to just run the update function for a given number of epochs in a free-running phase and to then return the resulting states of the visible layer as outputs. Depending on which of the approaches described above is being used the only difference is to decide which visible units should be clamped and to initialize them. All unclamped units are initialized with random binary values and the update procedure is called and returns the whole visible layer as output at some point.

3.3 Asynchronous vs. Synchronous Updating

One of the disadvantages of the original version of the BM is its computational efficiency. Firstly the learning algorithm requires the production of a lot of independent random numbers which can be costly. The main concern however is that the original algorithm is not suited very well for parallelization. This is a very strong concern since modern hardware relies heavily on the use of parallel code execution to achieve good performance. This is true for today's multi-core CPUs as well as for GPU computing. If we take a closer look at the previously seen update procedure shown in Figure 3 we can see in line 4 that inside the loops each update is performed on a single unit in a randomly chosen order. This behavior is actually required to conform to the underlying theoretical proof for the BM's algorithm. This stems from the fact that the resulting state of one unit will influence the calculations to determine the state of the next chosen unit. This version of the update procedure is called the asynchronous version. This approach pretty much prohibits the parallelization of the update procedure and it impacts heavily on the performance of the BM and therefore limits the sizes of the layers if the algorithm is expected to finish in a reasonable time.

```
1 FOR 1 to number of epochs:
2   FOR 1 to number of unclamped units:
3     randomly select an unclamped unit i
4     Calculate its energy difference: $\Delta E_i = \sum_j W_{ij} x_j$
5     #Stochastically decide the resulting state
6     IF random uniform $U[0,1] < \frac{1}{1+e^{-\Delta E_i / T}}$:
7       SET $x_i = 1$
8     ELSE:
9       SET $x_i = 0$
```

Figure 3: The asynchronous update algorithm

To solve this performance issue a variation of the update algorithm exists which can be parallelized. This version is called the synchronous update procedure, and was already an option for Hopfield networks [8, Chapter 43]. Here the calculations for the state updates are all done in one step on all unclamped units. This at least in theory introduces some inaccuracies. It is however not clear whether these inaccuracies have a strong impact on the practical use of the BM and the use of the synchronous update procedure provides the big advantage of being parallelizable. In the implementation and results parts of this paper, we will compare both of these update procedures and apply qualitative metrics to measure the impact of using the synchronous update over the asynchronous one.

4 Implementation

Based on the Python implementation from Cartas [2], we generalize to allow some more flexibility by parameterization. Also we replace Python *for*-loops with equivalent expressions using NumPy functions, in order to speed up the execution. As a new feature the choice between synchronous and asynchronous update method is introduced.

The idea is to bring the BM to the current world of neural networks, that offers frameworks like TensorFlow[1] or PyTorch[2]. But the BM is a stochastic neural network, uses its own energy-based minimization process and does not use activation functions or backpropagation. The BM would not utilize any of the builtin tools these frameworks offer, and thus is hard to integrate.

Our final implementation of the Boltzmann machine can be found on GitHub[3]. We try to make it as generalized as possible so that our Boltzmann python class can be used quite easily on all kinds of problems that one might use a BM for. The final code has a low number of dependencies and should be easy to use. It is built with the use of NumPy, which is easy to install for any user. At the creation of a Boltzmann class instance a list of free parameters can be adjusted to fit the current use case:

- **Visible Layer Size**: This should simply be set to the size of the input patterns that will be learned
- **Hidden Layer Size**: A free parameter that has to be tuned to the problem. In our tests we do not find a good general recommendation for the size of the hidden layer in respect to the visible layer size. Intuitively the hidden layer should be as as large as possible to allow the BM to store as much information as possible. The drawback of this is however a strong increase in runtime. In practice the hidden layer size should probably be set close to the size of the visible layer to make good use of it and at the same time not cause too much of an increase in the runtime.
- **Output Layer Size**: This is an optional argument which can be set to zero, if not needed. The function of a so called output layer is to allow use cases as for example the encoder-problem which was described in Section 5.1, where the visible layer needs to be split into two parts that are not interconnected.
- **Annealing Schedule**: This argument expects a list of tuples of the form (<temperature>, <epochs>). These are the values for temperature and the number of epochs which will be used in the learning phase of the BM. In the

[1]`https://www.tensorflow.org`
[2]`https://pytorch.org`
[3]`https://github.com/oliver-pola/BoltzmannMachine`

general case a single tuple should suffice here but our implementation also supports the so called annealing method for learning where the learning phase is started with a high temperature which is then slowly decreased over time.

- **Co-occurrence Schedule**: This argument again takes a tuple of the form (<temperature>, <epochs>). This is the temperature and number of epoch which is spend in the COLLECT−STATISTICS part of the learning algorithm. The temperature chosen here should match the learning temperature in most cases but the number of epochs can differ since we only need enough epochs to get meaningful values for the update-weights part of the learning algorithm.
- **Synchronous Update**: Lastly it can be decided if the asynchronous or synchronous update procedure should be used.

The final code can as mentioned before be found in our github repository along with the code that was used to create the results in Section 5.

5 Results

To apply the BM on a simple but not too minimalistic problem, we use the MNIST dataset[4] [7] containing 60000 images of handwritten digits with labels. Each image is 28×28 pixels with grayscale values. Scaling the values to $[0, 1]$ would be a good example for using the pixel values as continuous state data. But so far the BM is made for binary states and we applied the midrange threshold to get $\{0, 1\}$ pixel states.

The experiment and its results are presented in two steps. First, we show a single use case of the BM in Section 5.1. And second, in Section 5.2, a search in the parameter space is performed. This shows different outcomes, after training the BM again and repeating the previous experiment.

5.1 Recalling Digit Pixels

From all the digits we pick only the digits 3, 4 and 5 and only a few examples in total. Then we train the BM on this subset, having one input for each pixel and no output. The size of the hidden layer is chosen to be equal to the input layer. The hidden layer is an important feature of the BM in opposition to a Hopfield network. The chosen size should provide the BM with some learning capacity, but due to time restrictions this parameter is not further evaluated.

The chosen total amount of epochs and the temperature are based on the evaluation in Section 5.2. The other meta-parameters are based on examples [2] and on small exploratory sample runs, but not evaluated in detail.

To test if the BM is able to recognize the shown patterns, we pick one of the shown digits, delete the bottom half of it and use the BM to restore the missing pixels. We try the recall multiple times and compute the average of the results to

[4] `http://yann.lecun.com/exdb/mnist/`

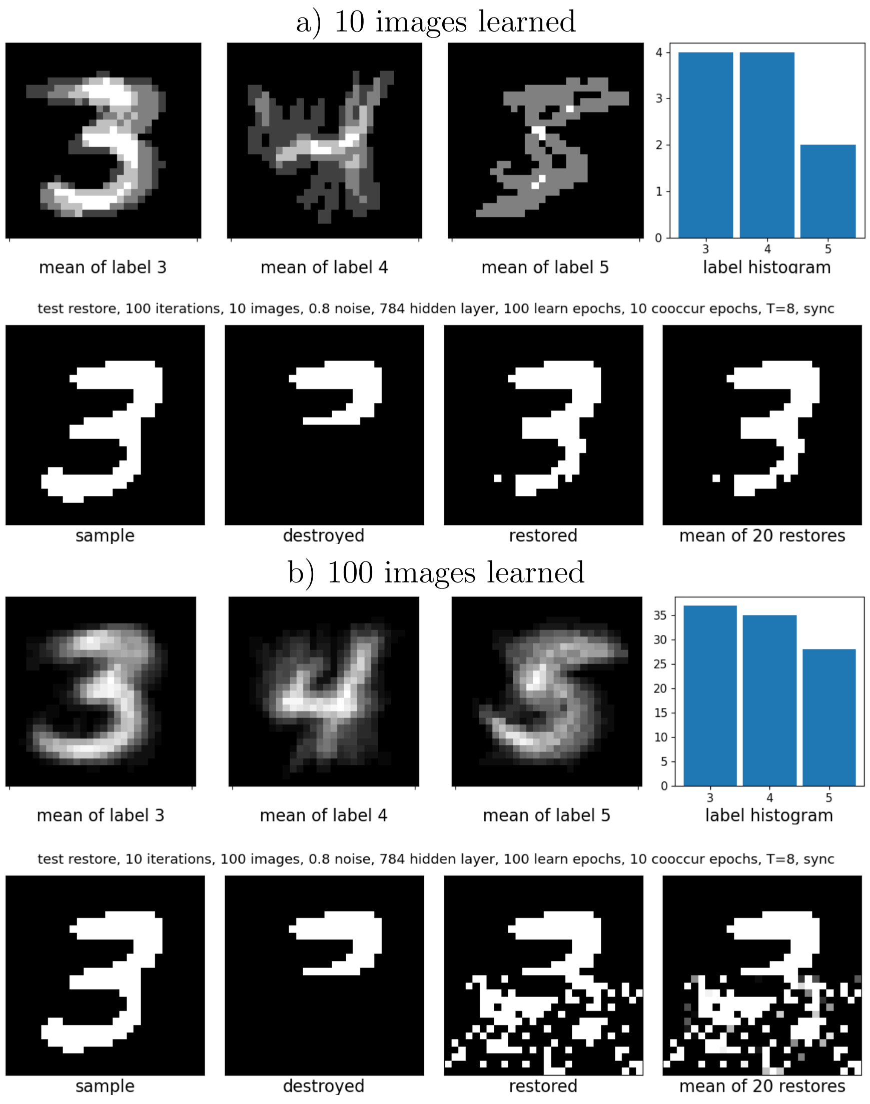

Figure 4: Restoring MNIST digits after a) 10, b) 100 images were learned
The BM learns the sample digit 3 to restore, but also learns other versions of the digit 3 and digits 4, 5 as well. From that sample the lower half is deleted and then restored by the BM. In a) we can see a successful restore of the pattern. The result of b) is not satisfactory.

get an idea about the distribution. We always use the same sample and delete the same pixels, so that the task does not vary in difficulty, knowing that the BM is unable to exploit that.

Figure 4 shows the distributions of the input images and the results of the recalled half pattern. If only 10 different images are used for training during 100 iterations, the restored digit can be identified very well, although it is not exactly the missing half. If instead of repeating the same image many times, we show some more variations of each digit, 100 images in total, the result is not satisfying. One reason might be if the variations differ too much, but the mean over the shown label 3 images shows, that there is not that much variation and the result should have been better.

We also can see that there is not much variation in calling recall more than once and almost the same output seems to be given each time. This means the restored pattern is a stable outcome, pretty well defined by the data distribution and the given input.

5.2 Meta-Parameter Evaluation

Having seen a good result from the experiment with 10 images in Section 5.1, that same set of meta-parameters is now used as a base for further analysis on the learning phase.

While all other meta-parameters are kept fixed, the same experiment is performed, but with varying total amount of epochs and temperature. To have a scalar to measure the quality q, a simple approach using the pixel values $x_{\text{sample}}(i)$ from the original and $x_{\text{restore}}(i)$ from the restored pattern at the same coordinate i is applied. The left term rewards pixels set in the restored pattern where a pixel was set in the original, and the right, negative, term punishes restored pixels where no original pixel was set:

$$q = \sum_{i:x_{\text{sample}}(i)=1} \frac{1}{20} \sum_{\text{20 recalls}} x_{\text{restore}}(i) - \sum_{i:x_{\text{sample}}(i)=0} \frac{1}{20} \sum_{\text{20 recalls}} x_{\text{restore}}(i)$$

The results in Figure 5 a) show that, at least for 10 images only, the results will not get better extending the number of epochs.

About temperature T is known that at $T = 0$ the BM is no longer stochastic, but deterministic. The higher T, the more unknown states are tested, instead of staying at the known minimum so far. A too high T though will just cause a lot of noise. From this, a quality maximum is expected for some mid-range T.

The course seen in Figure 5 b) is not expected and without explanation. To gain information out of this, an average over many runs of this experiment would be needed, but was not conducted due to time restrictions. Peaks at very low T are considered as lucky initial randomness in this instance, whereas higher T is considered to be of more random nature. The peak at $T = 8$ seemed to be most stable and was chosen to be the base value for our experiment.

As the synchronous update was implemented to benefit from easier parallelization, we focused to tune the meta-parameters for this version. Figure 5 shows that the behavior is different. For the chosen set of parameters the synchronous update is more stable and does not degrade when continuing learning for more epochs. Another set of parameters might be the sweet spot for the asynchronous update to reach this stability.

6 Discussion

As we have seen in this paper the BM is a very interesting approach for machine learning but it does not come without its drawbacks. The BM offers itself to do a

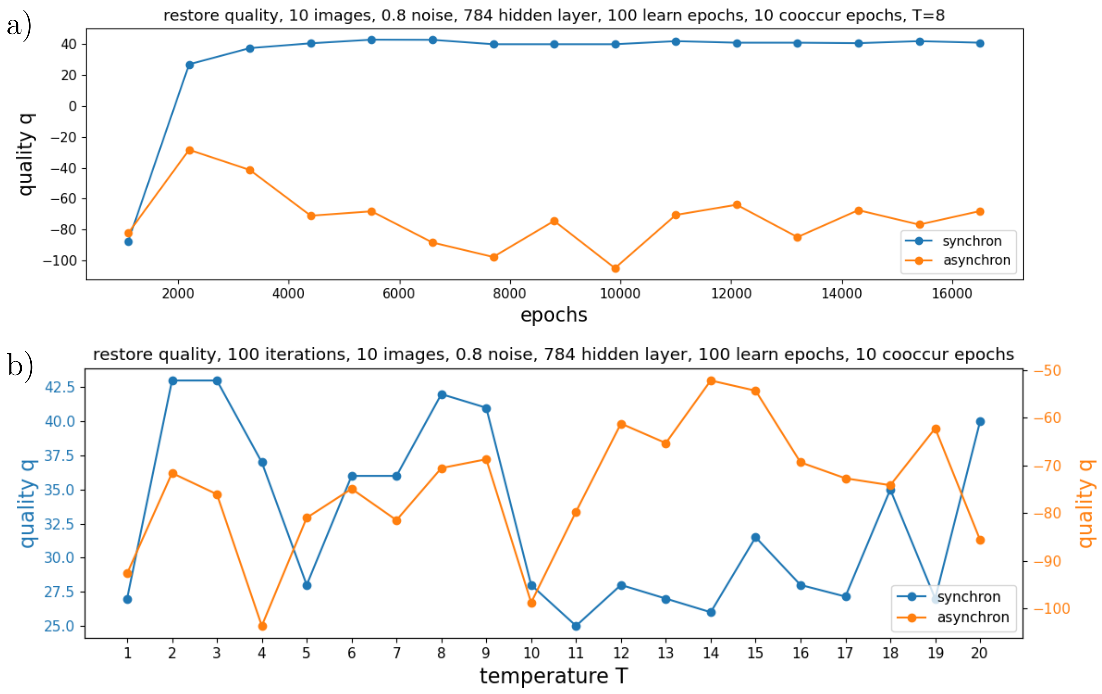

Figure 5: Evaluation of meta-parameters a) amount of epochs and b) temperature
On the same experiment from Section 5.1, the learning phase is repeated for every data point on the x-axis. After recalling the pattern, the quality is measured by q and displayed on the y-axis.

variety of different experiments that one might not be able to do with more classical neural networks and it especially sets itself apart with its stochastic nature. On the other end, however, the main concern with the BM is still the runtime. Even using the synchronous update method, our small experiments took a long time to run, even though the size of our data was by no means big. In its current form it would be infeasible for our implementation to run experiments on datasets of thousands of pictures or even pictures with a lot of pixels. The performance of our current implementation could hopefully be improved in future by enabling it to run fully parallelized on GPUs. As previously mentioned we tried to integrate our learning algorithm in a modern machine learning framework like Tensorflow but due to the Boltzmann machines very specific stochastic update process, which differs a lot from the normal types of layers that are implemented in these frameworks, this approach was unsuccessful for us.

The second thing we learned from experimenting with the BM is that it is a very complex task to tune all of the needed parameters correctly to a specific problem. As we have seen in Section 5 we could achieve quite good results for the restoration of destroyed input data for the synchronous update method, but for example using the same parameters for the asynchronous update already yielded strongly different results. Looking at the temperature plot in Section 5.1 we can see a lot of variation for the different temperatures and no clear trends can be made out. This leads us to the conclusion that the parameter choice is highly specific for each different experiment and it is hard to give general advice for the parameter choices.

We also observed a strong decrease in the quality of the results when running our experiments on a high number of images. This is most likely caused by firstly not finding the correct parameters for these kind of runs and secondly not being able to run them for enough epochs due to the previously mentioned long runtimes.

Overall we can say that the Boltzmann machine is a very interesting topic to work on but to be able to use it for actual real world problems it would require a lot more experiments and study that go beyond the scope of this seminar.

7 Future Work

The BM as it was defined and used in this paper is only applicable to static discrete data. As already mentioned when using the MNIST images, we therefor had to adjust the pixel values from range $[0, 255] \cap \mathbb{N}$ to be binary 0 or 1 via threshold. It would be nice to directly use values in continuous range $[0, 1]$ instead. Where we currently set the new state to 1 with probability p, this could be achieved by directly setting the new state to p. The other parts of the algorithm then need to be adapted as well.

As the BM is a recurrent neural network, a proper application seems to be time series data. This also seems to be where the Restricted BM is applied, that we have chosen not to cover in this paper. But it might to be possible to adapt the general BM to this task as well.

8 Conclusion

In this paper we introduced the BM. We have seen a very different approach to neural networks which is not exactly new but has not had the same prominence as the more commonly used networks in modern machine learning. The stochastic nature of the BM makes it very interesting for further studies and, if the performance problems can be mitigated a bit better than in our implementation, the BM definitely has its use cases in the field.

This seminar topic was mostly focused on the implementation and experiment aspects but with this paper we tried to bring the reader a good understanding of the core concept of the BM and how it could be used in practice.

We have also highlighted some of the problems we encountered when applying the BM on actual problems. As mentioned in Section 6 the long runtimes and difficult adjustments of all parameters proofed to be the most challenging aspects.

We also saw a lot of possible future improvements like extending the BM to work with continuous values or time series data which are beyond the scope of this seminar.

In conclusion we can say that the BM is very different from other neural networks, but an interesting approach to machine learning. Further study is still needed to unlock more of its potential.

References

[1] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A Learning Algorithm for Boltzmann Machines*. *Cognitive Science*, 9(1):147–169, 1985.

[2] Alejandro Cartas. Boltzmann Machines. `http://gorayni.blogspot.com/2014/06/boltzmann-machines.html`, 2014. (accessed 06/2020).

[3] Aaron Courville, James Bergstra, and Yoshua Bengio. A Spike and Slab Restricted Boltzmann Machine. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 233–241, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[4] Geoffrey Hinton. A Practical Guide to Training Restricted Boltzmann Machines (Version 1), 08 2010.

[5] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[6] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift fur Physik*, 31(1):253–258, February 1925.

[7] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[8] David JC MacKay. *Information theory, inference and learning algorithms.* Cambridge university press, 2003.

[9] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[10] Raúl Rojas. *Neural Networks: A Systematic Introduction.* Springer-Verlag, Berlin, Heidelberg, 1996.

[11] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 791–798, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Sunny vd. A graphical representation of an example Boltzmann machine. `https://commons.wikimedia.org/wiki/File:Boltzmannexamplev1.png`, 2012. (accessed 06/2020).

[13] Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. The recurrent temporal restricted boltzmann machine. In *Advances in Neural Information Processing Systems*, pages 1601–1608, 2009.