CS51 Final Project Writeup

Angela Fan, Andre Nguyen, Vincent Nguyen, George Zeng

April 2015

1 Instructions

Our code can be found in **iPython ONLINE NOTEBOOK**, which can be found in our git repository: `https://github.com/huihuifan/RBM-CS51.`

We annotate the purposes of each file in our GitHub ReadMe, but the code and output is in the **iPython ONLINE NOTEBOOK**. We have also included the code in our zip file, which we submitted with this report.

2 Video

URL: `https://www.youtube.com/watch?v=Jet46-33xWo&feature=youtu.be`.

3 Report

3.1 Implementation List

We implemented the following:

1. Restricted Boltzmann Machine
    - Binary with Sigmoid Activation Function
    - Continuous with Rectifier Linear Units
    - Training with Contrastive-Divergence 1
    - Training with Persistent Contrastive Divergence
2. Deep Belief Network stacking binary and continuous RBMs
3. Visualization Function to run, train, and plot the RBM and DBN
4. Single Perceptron
5. Eigenface transformation of Continuous Faces in the Wild dataset
6. Datasets:

- Toy dataset of movie users
- MNIST handwriting dataset, binary and continuous
- Faces in the Wild dataset, binary and eigenfaces

3.2 Notes

3.2.1 Faces in the Wild

As an extension, we explored running our continuous RBM on the Faces in the Wild dataset. First, we ran our continuous RBM on the raw Faces images, but we did not get many results besides noise. We then applied the same binarization to the Faces in the Wild dataset to the MNIST file, and we did get reasonable image outputs in the forms of faces. Although each unique face individually looks in a different direction, they average to look forward, so our output seems good. We theorize that the continuous RBM is unable to run on the raw images because they have too many distinct features, as our choosing of hidden units is random. Many papers in the literature restrict and localize their hidden units to represent specific features in the face, such as hair and sideburns. We were not able to implement this feature. Instead, we transformed the data using an eigenface decomposition by applying Principle Component Analysis using the Python library Sci-kit Learn, which we hoped would reduce the dimensionality and complexity of the features of the human face. We then attempted to re-run our continuous RBM on this simplified dataset.

3.2.2 Computational Constraints

Since the learning rate has to be very small for the continuous versions of RBM and DBN, the computation time for the continuous versions is approximately inversely proportional to the learning rate. Therefore, because we were limited in terms of our computational power i.e. we had no external server/computing cluster and we only had about a week to run our project, our results for the data sets, especially the "Faces in the Wild" data set, could definitely be improved. In general, machine learning researchers run their projects for a much longer time and with much more powerful computers. It would have been ideal if we also had a similar amount of resources to run our project.

3.3 How good was your original planning?

We were able to stick to our original schedule and construct the RBM and train it on both the toy dataset of movies as well as binarized version of the MNIST handwriting dataset according to the two-week task list written in our original specification. Additionally, we had time to explore a number of extensions to our original proposal.

3.4 How did your milestones go?

We found that our milestones were set reasonably; this is because we generously allocated time for each milestone's completion. In addition to concrete milestones regarding project modules and tasks, we found that setting aside dedicated days and times to work on the

project was extremely helpful in achieving the "hard" project deadlines. In summary, having a dual schedule system—one schedule for project details and features, and another schedule for when we would work on the project as a group and as individuals—aided us in completing the tasks set in both schedules.

3.5 What was your experience with design, interfaces, languages, systems, testing, etc.?

Since `Python` does not have especially strict manners of implementing private vs. public methods, we did not have too much trouble in the initial hiding and revealing of the methods of the classes that we used. Since our project also did not have many interfacing modules, it was fairly easy to verify which methods and values should be broadcast publicly.

As a language, `Python` provided us a friendly environment for rapid prototyping of the functions and methods used in our project. Moreover, the numerous packages created for Python for machine learning, statistical analysis, and other tasks related to scientific computating allowed us to transcend the task of developing our own data structures to represent matrices and work with matrix operations. We are forever indebted to `numpy` for its handling of array and array operations.

Additionally, `iPython Notebook` was extremely useful in testing the various components of our project. Instead of having to run an entire script, `iPython Notebook` allowed us to run individual snippets of code, effectively giving us an environment as friendly as `MATLAB` in debugging specific regions of code and verifying that the corresponding inputs and outputs were handled successfully.

GitHub was an interesting tool throughout the project development. Although it was initially challenging to navigate the world of pointers, merge conflicts, and pushing and pulling, it ultimately was a useful tool in distributing updated code quickly and accurately. In contrast to email or DropBox, GitHub gave us a quick and easy way to track changes made throughout the project timeline and, in the occasional instances when a notebook was corrupted or contained a syntax error that we could not find, we found the reversion under GitHub to be intuitive, minimizing miscommunication between group members.

3.6 What surprises, pleasant or otherwise, did you encounter on the way?

The most notable surprises and challenges that we encountered on the way were:

- Learning how to use GitHub—as mentioned above, the GitHub interface was initially very foreign and strange; however, once we had some experience with the basic functionality we didn't have too many more troubles with version control. Of course, we also had our fair share of niche GitHub issues i.e. partial commits, odd merge conflicts, etc. In the end, we found that the experience with GitHub was worth the time and effort

- Figuring out how numpy encapsulates arrays—in terms of mathematical operations and handling of matrices, numpy in Python is not as consistent as a purely technical

language such as MATLAB. In effect, we had to remember that array calls sometimes return matrices/arrays within another matrix/array.

- Matrix multiplication—writing out the math underlying the machine learning, then converting all of the math into matrix form, and then converting the matrices into numpy structures was surprisingly hard to keep track of

- Running times/efficiency of code—since neural networks are usually considered to be computationally intensive, we were surprised that we obtained fairly good results with running times (on ordinary computers) of reasonable orders of magnitude. We were able to run the MNIST data to obtain high quality results within an hour or two.

- Scientific computing—considerations regarding floating point arithmetic and overflow errors. Despite many of the extensions looking very simple on paper, they brought along a number of issues. For example, adding the continuous input to RBMs we thought would be a simple addition of an activation function, but the way the Rectifier Linear function is written caused float overflows.

- Neural Networks are somewhat hard to interpret from a theoretical standpoint- Training neural networks is complicated and requires great precision for them to be fine tuned. When we were moving from our binary RBM to our continuous RBM, we were debugging the weight matrix for a long period of time as all the weights were moving to zero, yet there didn't seem to be a bug in our code. Upon thinking about it more, we realized this was an accurate assessment- there was no code bug, but the continuous RBM simulated the real-life phenomenon of neuron "death" through trimming (lack of continuous stimulation. We then realized that because our learning rate was so high, many neurons were experiencing this death scenario, but when we adjusted to a very small learning rate, digits began to form. However, this small learning rate meant that the training takes an incredibly long number of epochs, which isn't possible computationally without a cluster.

3.7 What choices did you make that worked out well or badly?

As noted above, having a dual schedule system—a task list of project features and a timeline for dedicated working times—gave us the opportunity to constantly keep in mind how much time we would need to complete any remaining work for the project. Although we never found ourselves particularly pressured to modify the schedule and allocate more time than we had expected to work on the project, our dual schedule system gave us a back-up plan in the event that we needed to do so.

On top of that, we found that having group meetings and project sessions as soon as possible gave us a great amount of flexibility and reduced stress in the days leading up to the project deadline.

Another good choice that we made for the project was choosing to develop in Python as opposed to a lower-level language. Although we would love to write the underlying operations in an efficient manner at a low-level, we admittedly do not have enough experience as we would want in implementing code with greater efficiency than Python does with its

underlying C libraries. Python allowed us to write and check code quickly without having to worry about particularly aspects of memory management; it may be a good exercise in our future paths in computer science.

3.8 What would you like to do if there were more time?

With more time, we would like to explore the following:

- Other link functions for RBM, such as the hyperbolic tangent, softplus, extending our rectifier linear function to include Gaussian noise, and leaky rectifier linear units
- Train our RBM to take in non-image datasets for prediction purposes, such as testing on the Netflix movie database
- If we had a semester or so and some additional experience in lower-level programming, perhaps we could have our RBM and DBN run even faster and more efficiently. For example, we could have coded our RBM in C++ or even Cython. It would also afford us the opportunity to learn about computational complexity and the chance to explore some algorithms beyond the scope of this class.
- Learn how to package our project into a Python package that others could use. The scikit-learn package has a very basic and generic implementation of a neural network (BernoulliRBM). Although such packaging and dressing up of our project is not as directly relevant to the machine learning aspects of the project, it would be great for us in terms of "full-stack" development.
- (Noted in the previous "Notes" section as well.) We are very satisfied with our results for our project but we definitely could have obtained better results for our data sets, especially for the more complex data sets which would have benefited from a greater number of epochs/iterations through the RBM/DBN. If we had a greater level of computing resources as well as a generous time frame for the project, we would have used the increased computational power and time to obtain these better results.

3.9 How would you do things differently next time?

If we were starting over this project, we would:

- Think more about what the hidden layers mean. When we first started testing our RBM on the MNIST dataset, we couldn't understand why the reconstructions of the digits were so badly done, but the answer was simple- we weren't allocating enough hidden units for the RBM to have enough capability to distinguish between the different units.
- Avoid being intimidated by a large-scale software project: we initially approached the project with relatively low expectations in terms of how much we could complete and we were definitely unsure of whether we could even reach our first major goal of finishing the basic RBM but we found that our timelines were generous to complete the tasks along with the most of our other extensions.

- We would secure a computing cluster for our continuous RBM, as it probably needs to be trained for over 10,000 epochs to achieve good results without large-scale neuron death.
- Our ultimate goal for one of our extensions was to implement a Multialyer Perceptron to work alongside our RBM to provide a comparison between supervised (e.g. MLP) and unsupervised (e.g. RBM) neural network learning. However, we ended up only implementing a single perceptron

3.10 What was each group member's contribution to the project?

1. George Zeng-
    - Debugged the RBM—specifically ensuring that underlying code reflected correct math/matrix operations
    - Wrote original and final specification
    - Wrote final writeup (with Angela)
    - Condensed code, abstracted functions, wrote higher order functions for original RBM
    - Worked on perceptron implementation
2. Vincent Nguyen-
    - Binarized MNIST handwriting dataset, and trained RBM on it
    - Trained RBM on toy dataset of movie users
    - Downloaded and formatted MNIST and Faces in the Wild datasets
    - Trained RBM on Faces in the Wild
    - Wrote parameter optimization grid search code and produced cross-validation visualization code
    - Helped debug the matrix multiplication issues in the original RBM
    - Helped debug the Multilayer Perceptron (with George)
    - Narrated and edited the movie presentation
3. Andre Nguyen-
    - Coded the Deep Belief Network that stacked RBMs
    - Researched the theoretical aspects of RBMs, particularly the extensions, such as Rectifier Linear Units; additionally, researched DBNs
    - Wrote RBM sampling functions
    - Coded continuous extension to the project by writing a Rectifier Linear function for continuous RBMs and debugged continuous RBM

- Debugged Multilayer perceptron (with George)

4. Angela Fan-

   - Created movie presentation animations and wrote movie presentation script
   - Wrote Functionality Checkpoint and Final Writeup (with George)
   - Coded original RBM (with group help/debugging)
   - Wrote RBM visualization function of MNIST dataset and error per epoch
   - Combined, formatted, and commented code from various working files, and Github ReadMe
   - Added higher order functionality by factoring out common code between training algorithms
   - Reformatted objects to take boolean flags that accept user specifications
   - Wrote Persistent CD-k algorithm
   - Helped Andre with continuous RBM
   - Did eigenface decomposition using Sci-kit-learn

3.11 What is the most important thing you learned from the project?

When we originally started the project, we felt overwhelmed by the complexity of neural networks. We found papers with complicated RBM descriptions, and started coding following these papers, writing many functions. Eventually we realized that the core of the RBM was not, in fact, this complicated, and by using Numpy we could simplify many of the loops, and simplified our code until the RBM trained and produced quite good results on the MNIST handwriting dataset, allowing the entire RBM code to be only around 100 lines. We realized that algorithms seemed incredibly complicated, but if we took a step back and thought deeply about what we were coding and how all the moving parts came together, the neural network was not as complicated as anticipated.

4 Annotated Original Spec

Our annotations on the original specification will be in blue.

4.1 Feature List

We believe that implementing an RBM will be a decent amount of work. We will implement the RBM as well as a few functions to help assess its performance. We will train and test our RBM on a toy dataset of 10 users and the movies they like. **We implemented the RBM successfully, and were able to train and test our RBM on a small dataset of movies.**

We will explore the following extensions, time permitting:

1. Persistent CD algorithm to approximate sampling from the energy function probability distribution, as an alternative to CD-k **We implemented Persistent CD-k as a separate public method inside the RBM class. We adjusted the RBM class to take an additional parameter that specified which training algorithm the user wanted to use.**

2. Train and test our RBM on more complicated datasets, such as MNIST handwriting image recognition and Faces in the Wild, a set of labeled images of faces **We trained and tested our RBM on the MNIST handwriting dataset, both a binarized version and the continuous version (using our Rectifier Linear Unit activation function). We also trained and tested our RBM on "Faces in the Wild."**

4.2 Technical Specification

A restricted Boltzmann machine is a simplified version of a Boltzmann machine in which nodes at hidden layers are restricted to non-cyclical graphs. The RBM will tune the parameters of an energy-based probabilistic function (linear in its parameters) so that desirable states have lower energy configurations. The energy function will be defined as

$$E(v, h) = -b'v - c'h - h'Wv$$

where W represents the weights that connect the hidden and visible nodes, and b and c are offsets of the visible and hidden node layers.

We will then sample from the probability distribution generated by the energy function using a Gibbs Sampler. In order to speed up the sampling, we will use the Contrastive Divergence algorithm with $k = 1$ (referred to as CD-k). The CD-k algorithm speeds up the sampling because it does not wait for the markov chain to converge, and allows us to initialize the markov chain with a known training example, avoiding the potential burn-in time. $\diamond\diamond\diamond$

We will implement this project in an entirely object oriented fashion. We will be coding a class RBM that takes three parameters:

- The number of hidden states
- The number of visible states
- A learning rate parameter

In the `init` function, the RBM will initialize a set of all weights that will be 0. Subsequently, we will use three methods:

- The first method will just train the neural network, which will allow the network to traing weights
- The second method will take the trained RBM, run it on the visible units, and will yield a generated sample of hidden units

- The third method will be a hidden method–it will take a trained RBM, run it on the hidden units, and then generate the visible units

We implemented an RBM class as described above, with private and public methods. We initialized weights randomly.

4.2.1 Project Responsibilities

Our project will naturally split into the following components; we note that not all of these tasks are equivalent in terms of time and effort so that ultimately we will divide the responsibilities into subtasks such that magnitudes of time and effort are distributed equally among the group members: **We combined some of these tasks, as some of the functions were higher level and encompassed others. The CD-k algorithm included contrastive divergence, and allows us to not have to write a complete Gibbs Sampler. CD-k also allowed us to train the RBM. To determine hyperparameters, we used cross-validation for parameter optimization. We implemented CD-k as a public method inside the RBM class.**

- Energy function
- Gibbs sampler
- CD-k algorithm
- Contrastive divergence
- Writing the method to train the neural network
- Specifications of hidden states
- Specifications of visible states
- Determining hyperparameters

In addition to these individual subtasks, we will also work collectively as a group to debug the finished program as well as integrate and synthesize the various different modules of the task. **We successfully worked as a group to debug.**

5 Next Steps

As an extension to our project, we intend to explore parameter optimization in the following ways: **We explored parameter optimization using grid-search to determine the learning rate and number of hidden states, by iterating over reasonable values of these two parameters. We did not need to use parameter optimization to determine the initial weights and initial biases, as the weights were initialized randomly and the biases were initialized to be 0. We monitored the fitting of the model by printing out the error after each epoch.**

- Adjusting the learning rate parameter
- Adjusting the initial weights
- Adjusting the initial biases
- Measuring and monitoring overfitting of the model

We will also explore the effects of using different types of units within the model. **We implemented a continuous activation function- called the Rectifier- that allowed us to use our RBM on a continuous version of the MNIST dataset. We implemented this by having the RBM take in an additional parameter that specified if the user wanted continuous or binary input, and then creating another private rectifier method.**

6 Annotated Final Spec

7 Signatures/Interfaces

See attached `Python` *code and comments for more detailed descriptions.*

Python is not strongly typed and we will probably not need to create too many objects outside of the arrays, lists, and other data structures provided in the `numpy` package for most of the interactions and implementations. **We used numpy for our matrices and matrix multiplication**

Python does not provide a strict method of defining public and private data types. However, within the overarching RBM class, we will have the following "private" methods: **We implemented the Logistic sigmoid function as a private method that took a matrix as a parameter and applied the Logistic sigmoid function to it. The others did not need to be implemented as separate methods, and were instead inside the CD-k method.**

- Hopfield Energy Function
- Logistic Sigmoid Function
- Partition Function
- Probability of Visible/Hidden Pair
- Probability of Visible

We will also have the following public methods: **We implemented all three of these methods. The learner training method we implemented was CD-k. We implemented CD-k as a method inside the learner. To predict hidden to visible and visible to hidden, we implemented two separate sampling functions.**

- Learner training method

- Method for predicting hidden to visible unit
- Method for predicting visible to hidden unit

Input formats will be pre-processed so that we can input a wide variety of types such as varying image sizes. **We created a binary version of the MNIST dataset for our binary RBM to run on, before extending our RBM to take continuous input. We binarized the MNIST dataset by looking at each pixel and setting an arbitrary threshold for intensity- if the pixel intensity was above this threshold, the pixel was black, and otherwise the pixel was white.**

8 Modules/Actual Code

See attached `Python` *code and comments for more detailed descriptions.* We will create an RBM class. Our extension versions will inherit from this class and override already existing methods. **We implemented an RBM class. Our extension of RBMs- Deep Belief Networks- used the RBM class by stacking the RBMs. The DBN we implemented takes in as a parameter a vector, where the length of the vector is the number of layers of RBMs that the user would like to stack, and each of the values in the vector corresponds to the number of hidden states.**

9 Timeline

9.1 Week of Monday, April 13th

We accomplished all of these tasks on time.

1. Finish final draft specification
2. Start creating the pseudocode for each function/class in the program
3. Within actual code, start skeleton code/signature for each function/class
4. Delegate concrete responsibilities to team members
5. Create fixed timeline for internal deadlines
6. Create first half of video that summarizes motivation and explanation of a neural network
7. Meet together on Sunday, April 19th for a day-long group coding session

9.2 Week of Monday, April 20th

We were ahead of schedule this week, and accomplished all of these tasks the previous week. This week, we instead focused on visualization with the MNIST dataset and optimization of our parameters. The optimization code took a substantial amount of time to run. We implemented our parameter optimization using a grid search, looping over reasonable ranges of the parameters. We created our visualization code using the Python library matplotlib, by initializing a grid of subplots and filling them in with images.

1. Implement basic version with binary hidden and visible units
2. Implement function definitions
3. Write test cases for methods defined inside the RBM class
4. Create and test our RBM on a toy dataset

9.3 Week of Monday, April 27th

We spent most of this week focusing on extensions, which we describe in annotations below.

1. *Overarching goals for this week:* finish implementation, start adjusting meta-parameters in application/implementation and optimize the RBM
2. Start testing the RBM on a simple data set such as the MNIST handwritten digit database
3. Analyze performance, make adjustments on RBM, choose optimal input data set (such that the computation is feasible given our time constraints)
4. Tidy up the code, add in any extra comments
5. Finish writing up the README
6. Finish video
7. *Extension:* Change the implementation such that hidden and visible units are not necessarily binary **We implemented the continuous RBM using a Rectifier Activation function by adding another method inside the RBM class. We added a parameter to the RBM class that specified if the user wanted a continuous or binary RBM.**
8. *Potential Extension:* Add graphical displays to check learning progress, overfitting, adjust learning rate **We added visualization functions of the error and sampling of the RBM, as well as graphs to help us adjust parameters**

We implemented some additional extensions- we trained and tested our RBM on "Faces in the Wild." We also implemented a better training algorithm, Persistent CD-k. We implemented Persistent CD-k as another method inside the RBM class.

10 Version Control

We used GitHub successfully. You can see our GitHub commit history for more information We have already created a GitHub repository for our final project. The clone URL is `https://github.com/huihuifan/CS51-FinalProject.git`. We will branch this repo as necessary.