Collaborative Filtering for Student Grade Analysis

by

Mufan Li

A thesis submitted in conformity with the requirements
for the degree of Masters of Science
Graduate Department of Statistical Sciences
University of Toronto

Abstract

Collaborative Filtering for Student Grade Analysis

Mufan Li
Masters of Science
Graduate Department of Statistical Sciences
University of Toronto
2016

This is my abstract.

Up to 250 words, single-spaced in block form in centre of separate page.

Acknowledgements

Put your Acknowledgements here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent developments in machine learning have made significant contributions to a wide range of fields that are not traditionally considered data science. In this research course, we intend to explore several of the machine learning techniques in applications to education.

Specifically, this research project aims to apply machine learning to analyze the student grade dataset from [1], which contains complete transcripts of undergraduate students from a major Canadian University. Similar to predicting user ratings, we are able to predict the grades for courses. From the predictions, this project intends to analyze the effect of choosing easier courses on student grades, specifically by comparing the predicted grades of courses students did not take against the courses taken within the same program. By analyzing the variation in course difficulty, these results could potentially improve curriculum design for educational institutions and admission procedure for graduate programs.

The project will focus on implementing three main methods of inference:

1. Matrix factorization (MF) [4] and if time permits probabilistic matrix factorization (PMF) [7, 10]
2. Restricted Boltzmann machines (RBM) [9]
3. Denoising auto-encoders (DAE) [13] and if time permits variational auto-encoders (VAE) [5]

Chapter 2

Supervised Learning

2.1 Feed-forward Neural Networks

We first consider a class of machine learning algorithms called supervised learning. In this case we have a dataset $\mathcal{D} = \{\mathbf{x}^{[n]}, \mathbf{y}^{[n]}\}, \mathbf{x}^{[n]} \in \mathbb{R}^{N_{in}}, \mathbf{y}^{[n]} \in \mathbb{R}^{N_{out}}, n \in \mathbb{N}$, with $\mathbf{x}$ as the input, and $\mathbf{y}$ as the label or output. We want to find a model $f(\mathbf{x}, \mathbf{w})$ such that it is the "closest" to $\mathbf{y}$, with $\mathbf{w}$ the parameters in the model. This section will introduce neural networks as the model f that predicts the labels $\mathbf{y}$.

In the simplest case, neural networks can be reduced to a generalized linear model (GLM), where the prediction is a linear combination of inputs but passed through a non-linear function:

$$f(\mathbf{x}, \mathbf{w}) = g\left(\sum_{i=1}^{N_{in}} w_i x_i + w_0\right)$$

Here $g(\cdot)$ is a non-linear function, with $\mathbf{x}$ is an N dimensional input vector, and $\mathbf{w}$ is the weight vector, which includes w_0 as the bias. Common choices of $g(\cdot)$, also known as activation functions, for neural networks include the logistic (sigmoid) function, the hyperbolic tangent function, and the rectified linear unit (ReLU):

$$\begin{aligned} g_{\text{logistic}}(z) &= \frac{1}{1+e^{-z}} \\ g_{\tanh}(z) &= \tanh(z) = \frac{1-e^{-2z}}{1+e^{-2z}} \\ g_{\text{ReLU}}(z) &= \max(z, 0) \end{aligned}$$

where $z = \left(\sum_{i=1}^{N} w_i x_i + w_0\right)$ denotes the linear combination for GLMs. Notice all of these functions have simple derivatives, and specifically logistic and hyperbolic tangent functions are monotonic and bounded by their horizontal asymptotes at infinities, which makes them great choices for binary classification problems.

Graphically, this can be represented by a series of input nodes $\{x_i\}$ connected to an output node f, with weights $\{w_i\}$ on the connections. Note bias is omitted from the graph but remains a parameter.

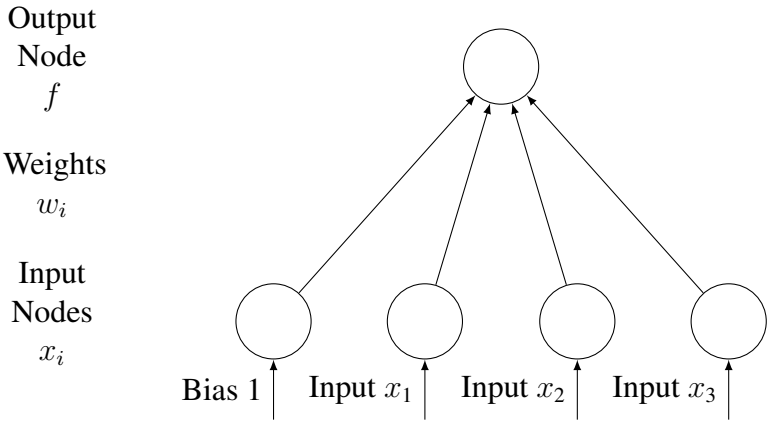


Figure 2.1: A generalized linear model represented in graphical form. In a neural network, this is also referred to as a single neuron.

A general feed-forward neural network is defined by recursive GLMs with different weights. For example, a neural network with two hidden layers (three layers of recursion) is defined as:

$$h_j^{(1)} = g^{(1)}\left(\sum_{i=1}^{N^{(1)}} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right)$$
$$h_k^{(2)} = g^{(2)}\left(\sum_{j=1}^{N^{(2)}} w_{jk}^{(2)} h_j^{(1)} + w_{0k}^{(2)}\right)$$
$$f_l = g^{(3)}\left(\sum_{k=1}^{N^{(3)}} w_{kl}^{(3)} h_k^{(2)} + w_{0l}^{(3)}\right)$$

where $g(\cdot)^{(\alpha)}$ is some activation function, $h_j^{(\alpha)}$ denotes the j^{th} node of the α^{th} hidden layer, $w_{ij}^{(\alpha)}$ denotes the weight for the connection of the i^{th} node of the α^{th} layer to the j^{th} node of the $(\alpha+1)^{\text{th}}$ layer, and $N^{(\alpha)}$ denotes the number of nodes in the α^{th} layer. Additionally, let $N^{(4)}$ be the number of output nodes f_l, and $\mathbf{w} = \left[\mathbf{w}^{(1)}\mathbf{w}^{(2)}\mathbf{w}^{(3)}\right]$. Here we also note that $N^{(1)}$ is the

number of input nodes.

Graphically, this structure has a very clear representation:

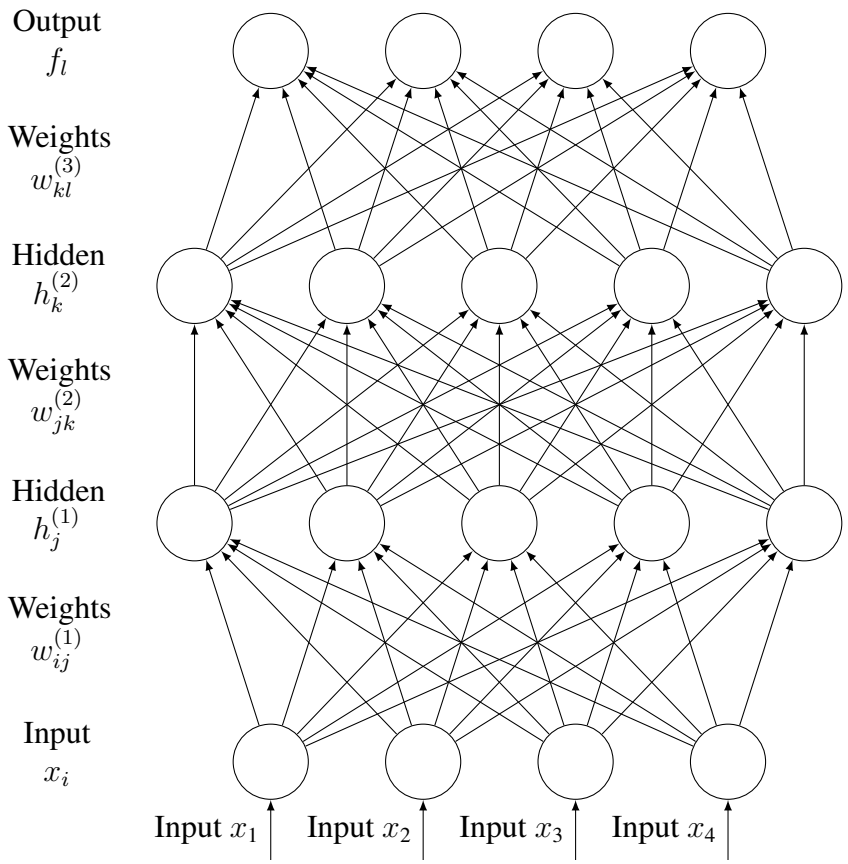


Figure 2.2: A generalized feed-forward neural network with two hidden layers. Bias parameters are not drawn for compactness, although they are present in all forward passing nodes.

While most GLMs do not admit a closed-form solution, a satisfactory optimization can be achieved by the gradient descent method. In the neural network case, the optimization becomes more difficult as the number of parameters increase with the number of nodes and layers. However, we can still apply the gradient descent method and find a local optimum for the simpler neural networks. [2]

Once again we have a dataset $\mathcal{D} = \{\mathbf{x}^{[n]}, \mathbf{y}^{[n]}\}, n \in \mathbb{N}$, and we want to find a model $f(\mathbf{x}, \mathbf{w})$ such that it is the "closest" to $\mathbf{y}$. If the error function $E(f, \mathbf{y})$ and the model $f(\mathbf{x}, \mathbf{w})$ are differentiable with respect to $\mathbf{w}$, the model can be optimized by gradient descent. In other words, for any randomly initialized $\mathbf{w}^0$, an improvement $\mathbf{w}^{k+1}$ can be obtained by making a

small modification in the direction of the gradient with respect to $\mathbf{w}^k$:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \, \nabla_{\mathbf{w}^k} E \left(f(\mathbf{x}, \mathbf{w}^k), \mathbf{y} \right)$$

where $\eta > 0$ is hyper-parameter controlling the change of each optimization iteration, commonly called the learning rate. Note η is not part of the final model $f(\mathbf{x}, \mathbf{w})$, but it will significantly influence optimization.

In the two hidden layer neural network previously, a derivative with respect to any weight $w_{ij}^{(\alpha)}$ can be found by applying the chain rule to the derivatives. For example the derivative with respect to $w_{jk}^{(2)}$ where $j \neq 0$:

$$\begin{aligned}
\text{let } z_j^{(\alpha)} &= \sum_{i=1}^{N^{(\alpha)}} w_{ij}^{(\alpha)} h_i^{(\alpha-1)} + w_{0j}^{(\alpha)} \\
\text{then } \frac{\partial E}{\partial w_{jk}^{(2)}} &= \sum_{l=1}^{N^{(4)}} \frac{\partial E}{\partial f_l} \frac{\partial f_l}{\partial z_l^{(3)}} \frac{\partial z_l^{(3)}}{\partial h_k^{(2)}} \frac{\partial h_k^{(2)}}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial w_{jk}^{(2)}} \\
&= \sum_{l=1}^{N^{(4)}} \frac{\partial E}{\partial f_l} \frac{\partial g^{(3)}(z_l^{(3)})}{\partial z_l^{(3)}} w_{kl}^{(3)} \frac{\partial g^{(2)}(z_k^{(2)})}{\partial z_k^{(2)}} h_j^{(1)}.
\end{aligned}$$

Recall $g^{(\alpha)}(\cdot)$ is selected to have a simple derivative, making the complex appearing gradient term above easy to compute.

2.2 Common Techniques to Improve Training

While the setup described in the previous section remains valid and works for simple cases, several simple techniques can significantly improve the speed and quality of optimizing the parameters in the neural network. It is important to note these techniques are also applicable in unsupervised learning techniques in Section 3.

2.2.1 Mini-Batches

Since the computational complexity of the gradient $\nabla_{\mathbf{w}^k} E$ scales linearly with the data size, and the data tends to be highly similar, it is acceptable to approximate the gradient using a small portion of the data. The resulting algorithm is to first divide up the input data into smaller batches, and then perform a gradient update for every mini-batch at random order.

The algorithm is summarized as follows:

Initialize $\mathbf{w}^0, k = 0$;
Partition dataset $\mathcal{D}$ into N mini-batches $\{\mathcal{D}_n\}_{n=1}^N$;
repeat
  Randomize the order of mini-batches $\{\mathcal{D}_{n'}\}_{n'=1}^N$;
  for $n' = 1, \ldots, N$ **do**
    Use $\mathbf{x}_{n'}, \mathbf{y}_{n'} \in \mathcal{D}_{n'}$;
    $\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \nabla_{\mathbf{w}^k} E\left(f(\mathbf{x}_{n'}, \mathbf{w}^k), \mathbf{y}_{n'}\right)$;
    $k = k + 1$;
  end
until *convergence*;

Algorithm 1: The Mini-Batch Gradient Descent

Using mini-batches takes away the need to perform updates after iterating through the entire dataset, which significantly reduces the computational time.

2.2.2 Momentum

Instead of letting the gradient dictate the change in $\mathbf{w}^k$, the idea of momentum is to let the gradient dictate the rate of change. If $\mathbf{w}^k$ is interpreted as a coordinate, and each optimization iteration as velocity, momentum can be seen as using the gradient as acceleration instead of velocity. This allows the optimization to accumulate speed in a consistent direction of the gradient, while making it harder to slow down and converge to a poor local minimum.

At the same time, momentum also improves the speed of convergence. It is well known that for a convex optimization problem, the gradient descent method will converge at a rate of $\mathcal{O}(k^{-1})$, where k is the number of iterations. The momentum technique is a special case of Nesterov Acceleration [8], where the rate of convergence is $\mathcal{O}(k^{-2})$ for a strongly convex objective function and a Lipschitz continuous gradient. Since the neural network objective function is highly non-convex, the momentum method tends to perform better in practice.

The formulation starts with a velocity vector $\mathbf{v}^0$ initialized to zero, and the rest in similar:

$$\mathbf{v}^{k+1} = \theta \mathbf{v}^k - \eta \nabla_{\mathbf{w}^k} E\left(f(\mathbf{x}, \mathbf{w}^k), \mathbf{y}\right)$$
$$\mathbf{w}^{k+1} = \mathbf{w}^k + \mathbf{v}^{k+1}$$

where $\theta \in [0, 1]$ is the hyper-parameter deciding the preservation of momentum. Here choosing

a larger θ would result in a stronger preservation of the velocity vector $\mathbf{v}$, which then retains more momentum.

2.2.3 Dropout

Another common issue for neural networks is over-fitting. Due to the large number of parameters, a neural network can tend to "store" the entire dataset into its parameters, hence over-fitting the training dataset. While regularization and early stopping are used for preventing this issue, we explore a simple yet highly effective technique called dropout [12].

The motivation for dropout comes from ensemble methods, where multiple model predictions are averaged, with each model weighted by its posterior probability given the data. Ensembles are highly successful when a large number of distinct models can be generated with relative low computation. A notable example is random forests [3] where each model is a simple decision tree. For neural networks, the simplest method to create distinct models is by considering different model architectures. However this is difficult to apply directly due to the computational cost of optimizing even one neural network.

Dropout is the method that attempts to incorporate random neural network architecture into the same training procedure. The technique specifically refers to "dropping out" some of the hidden units with some probability $p \in [0, 1]$, creating a random structure for each gradient descent iteration.

Specifically recall the feed-forward neural network, where we originally had

$$h_k^{(2)} = g^{(2)} \left(\sum_{j=1}^{N^{(2)}} w_{jk}^{(2)} h_j^{(1)} + w_{0k}^{(2)} \right)$$

Here for each gradient descent iteration, we introduce a list of Bernoulli random variables $r_j^{(1)} \sim$Bernoulli$(1-p)$, and modify the previous equation to

$$h_k^{(2)} = g^{(2)} \left(\sum_{j=1}^{N^{(2)}} w_{jk}^{(2)} h_j^{(1)} r_j^{(1)} + w_{0k}^{(2)} \right)$$

Observe that with probability p, each hidden node $h_j^{(1)}$ will be set to zero, hence creating a random structure. We also observe that since if the node $h_j^{(1)}$ is dropped, we have that the

parameter $w_{jk}^{(2)}$ stays unchanged for this iteration, since the hidden node $h_j^{(1)}$ does not affect the objective function for this iteration. Similarly, all the parameters $w_{ij}^{(1)}, \forall i$ intended for the nodes leading up to $h_j^{(1)}$ remain unchanged as well.

2.3 MNIST Hand-Written Digits Example

The Mixed National Institute of Standards and Technology (MNIST) dataset [6] is a collection of images of hand-written digits from various sources, with each image labeled the correct digit. The dataset contains 60,000 images for training (fitting), and 10,000 images for testing. The images are 28x28 in resolution, hence making $N^{(1)} = 784$ dimensions in input.

The data labels are changed to use the 1-of-K encoding scheme, where the label $\mathbf{y}$ is a binary vector of size K, with only one element taking a value of one. In this case, given 10 possible digits, we have a vector of size $N^{(4)} = 10$. For example, a possible scheme can label the digit "3" using the vector $[0, 0, 1, 0, \ldots]$ where only the 3^{rd} index is a "1".

To best model this type of label vector, the softmax function is chosen for the output layer:

$$f_l = g^{(3)}(z_l^{(3)}) = \frac{\exp(z_l^{(3)})}{\sum_{k=1}^{N^{(4)}} \exp(z_k^{(3)})}$$

where $z_l^{(3)} = \sum_{k=1}^{N^{(3)}} w_{kl}^{(3)} h_k^{(2)} + w_{0l}^{(3)}$ is a linear combination of the final hidden layer. Since the denominator normalizes the sum, the f_l now adds up to one, and a "perfect" output is exactly the 1-of-K encoded label. If f_l is modeled as the probability of the image being digit l, suppose the correct digit is m, then the likelihood of making the correct prediction is:

$$L(\mathbf{f}, \mathbf{y}) = f_m = \prod_{l=1}^{N^{(4)}} {f_l}^{y_l}$$

since $y_m = 1$ is the only non-zero term in the label vector. We can then define the error function as negative log-likelihood:

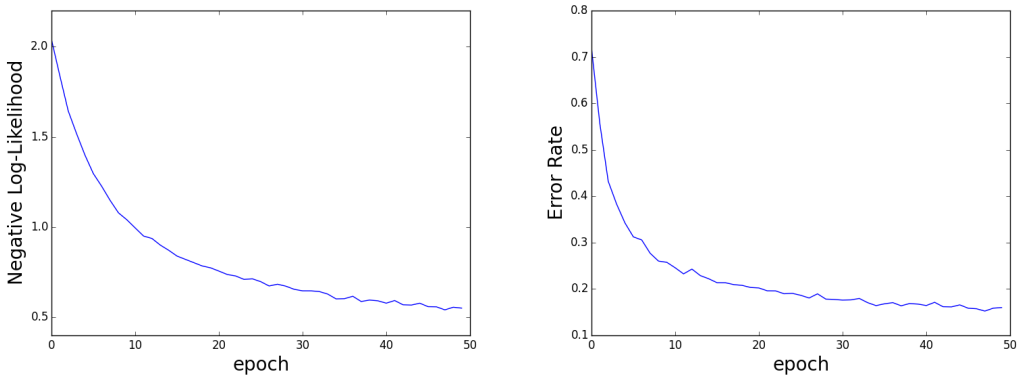
$$E(\mathbf{f}, \mathbf{y}) = -\log \prod_{l=1}^{N^{(4)}} {f_l}^{y_l} = -\sum_{l=1}^{N^{(4)}} y_l \log f_l$$

where $N^{(4)}$ is the number of output nodes, and minimizing E is equivalent to maximizing likelihood. Note taking the logarithm creates an error function with much simpler derivative,

hence simplifying the gradient descent method.

In the following experiment, a two hidden layer neural network is used to model the MNIST digits. We used $N^{(2)} = N^{(3)} = 1000$ nodes in the hidden layers, creating a structure of 784-1000-1000-10 $\left(N^{(1)} - N^{(2)} - N^{(3)} - N^{(4)}\right)$ nodes in each layer. We also chose $g^{(1)}(\cdot) = g^{(2)}(\cdot) = g_{\text{ReLU}}(\cdot)$ in the hidden layers, and softmax for the output layer. The hyper parameters were chosen as $\eta = 10^{-5}$ and $\theta = 0.9$. We also chose to update the weight vector $\mathbf{w}^k$ once for every 100 samples of digits, also known as a mini-batch.

After training (optimizing) for 50 epochs, with each epoch denoting one complete run through of the training dataset, we reach a test error rate of 16% (Figure 2.3b). By increasing



(a) The negative log-likelihood (b) The classification error rate

Figure 2.3: MNIST hand-written digits modeled using a two hidden layer neural network, with negative log-likelihood and classification error rate computed after each epoch.

the number of epochs and a few minor modifications, this model could potentially reach error rates as low as 2%. However, this will not be explored since it is not the main purpose of this document, and optimization can be very time consuming due to computation.

While the technique is more than a sufficient solution for recognizing hand-written digits, naively applying gradient descent to more complex neural networks tend to have poor results. Alternative methods will be discussed in the next section in order to address this problem.

Also note this type of neural networks is feed-forward, which mean it is limited to only supervised type problems where the data structure is consistent and a prediction target (label) is provided for each sample. For a collaborative filtering type problem, the inference is often made within the data structure itself, which makes an unsupervised learning problem. Feed-forward neural networks also fail to fully utilize the datasets that are partly labeled, known as semi-supervised problems. These problems would require other variations of neural networks

with different methods for inference.

Chapter 3

Unsupervised Learning

3.1 Restricted Boltzmann Machines

On the other hand, restricted Boltzmann machines (RBM) is a completely different approach to problems without labels. RBM is a type of unsupervised learning algorithm, for there are no labels to "supervise" the learning. The purpose of unsupervised algorithms are to find structural patterns within the data itself. In this case, we are interested in the relationships between the performance in difference courses, and how this helps us predict the grades.

A RBM is a Markov random field in the form of a bipartite graph, where the joint probability follows a Boltzmann type distribution. The bipartite graph structure creates two layers without internal connections. One layer, called the visible layer, contain the input data; in this case, the visible values are the grades of each student. These nodes are connected to the other layer, called the hidden layer, with symmetrical weighted connections.

Suppose the graph have N visible nodes and M hidden nodes, with each visible node denoted v_i, hidden nodes denoted h_j, weights between two nodes w_{ij}, b_i and a_j be bias parameters, and σ_i be the standard deviation of grades for each course. Here each visible node v_i represents the grade for course i, where a specific student is fixed. Let $\theta = \{w_{ij}, a_j, b_i, \sigma_i\}\ \forall i, j$, $\mathbf{v} = \{v_i\}\ \forall i$, and $\mathbf{h} = \{h_j\}\ \forall j$ denote the collections. Additionally, we let the hidden nodes only take on binary values, i.e. $v_i \in \mathbb{R}, h_j \in \{0, 1\}$. We can then define the energy function

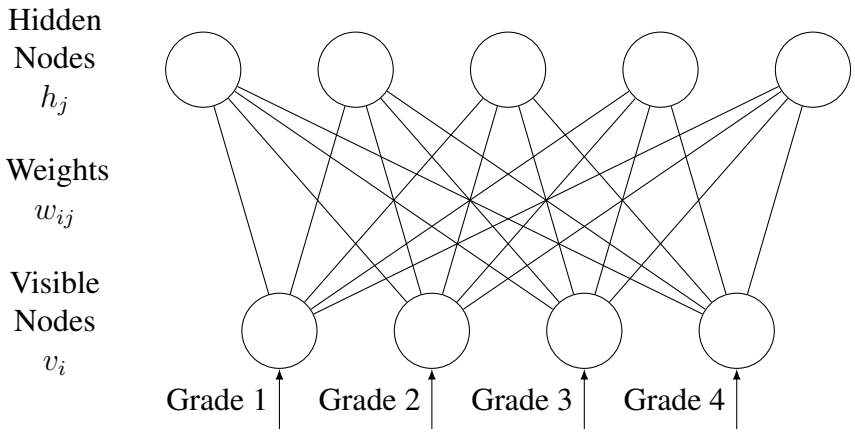


Figure 3.1: A restricted Boltzmann machine (RBM) with 4 courses and 5 hidden nodes for a specific student.

and the joint distribution for the graph:

$$E(\mathbf{v}, \mathbf{h}|\theta) = \sum_{i=1}^{N} \frac{(b_i - v_i)^2}{2\sigma_i} - \sum_{i=1}^{N}\sum_{j=1}^{M} w_{ij} h_j \frac{v_i}{\sigma_i} - \sum_{i=j}^{M} a_j h_j$$

$$P(\mathbf{v}, \mathbf{h}|\theta) = \frac{\exp\left[-E(\mathbf{v}, \mathbf{h}|\theta)\right]}{\mathcal{Z}}$$

where $\mathcal{Z}$ is the partition function normalizing the distribution. After marginalizing over the hidden nodes $\mathbf{h}$, we can find the gradient of the likelihood function with respect to the parameters θ to perform steepest descent optimization. Finding the gradient requires the use of Gibbs sampling, although [11] showed the approximate gradient after very few iterations of Gibbs sampling is sufficient for optimization.

$$\frac{\partial P(\mathbf{v}|\theta)}{\partial w_{ij}} = \mathbf{E}_{\text{data}}(v_i h_j) - \mathbf{E}_{\text{model}}(v_i h_j)$$

where $\mathbf{E}_{\text{data}}$ refers to expectation of observing the case within data, and $\mathbf{E}_{\text{model}}$ is the expectation of the current model with parameters θ. Instead of using Gibbs sampling until convergence to find $\mathbf{E}_{\text{model}}$, [11] uses k iterations for a very good approximation of the gradient. This method is referred to contrastive divergence (CD) by the authors in [11], where CD-k refers to k iterations used in Gibbs sampling. As a result, we have a very good algorithm optimize the RBM for likelihood.

To perform inference on a missing grade value, one simply include an additional "visible"

node v_p, where the value is not known, but can be determined by the energy function:

$$\begin{aligned} P(v_p|\mathbf{v}) &\propto \sum_{\mathbf{h}} \exp[-E(v_p, \mathbf{v}, \mathbf{h})] \\ &= \prod_{j=1}^{M} \left(1 + \exp\left[\sum_{i=1}^{N} w_{ij} v_i\right]\right) \end{aligned}$$

3.2 Denoising Autoencoders

Another approach to unsupervised learning is using autoencoders (AEs), specifically in this case we will introduce the denoising autoencoders (DAEs) in [13]. The autoencoder is a compression model of input data, such that a high dimensional input can be encoded as a low dimensional representation, where the data can be reconstructed from the representation using a decoder.

For this problem we consider a dataset $\mathcal{D} = \{\mathbf{x}^{[n]}\}, \mathbf{x}^{[n]} \in \mathbb{R}^{N_{in}}, n \in \mathbb{N}$, with only $\mathbf{x}$ as the input. We also define a desired feature $\mathbf{h} \in \mathbb{R}^{N_{feat}}$ with $N_{feat} < N_{in}$, and an encoder-decoder pair $f(\mathbf{x}, \mathbf{w}^{(1)}), g(\mathbf{h}, \mathbf{w}^{(2)})$ such that the reconstruction $\hat{\mathbf{x}} = g \circ f(\mathbf{x}) \approx \mathbf{x}$. This results in a forced compression of input $\mathbf{x}$ into lower dimensional $\mathbf{h}$, and in the process, the parameterization $\mathbf{w}$ retains further information about the data structure.

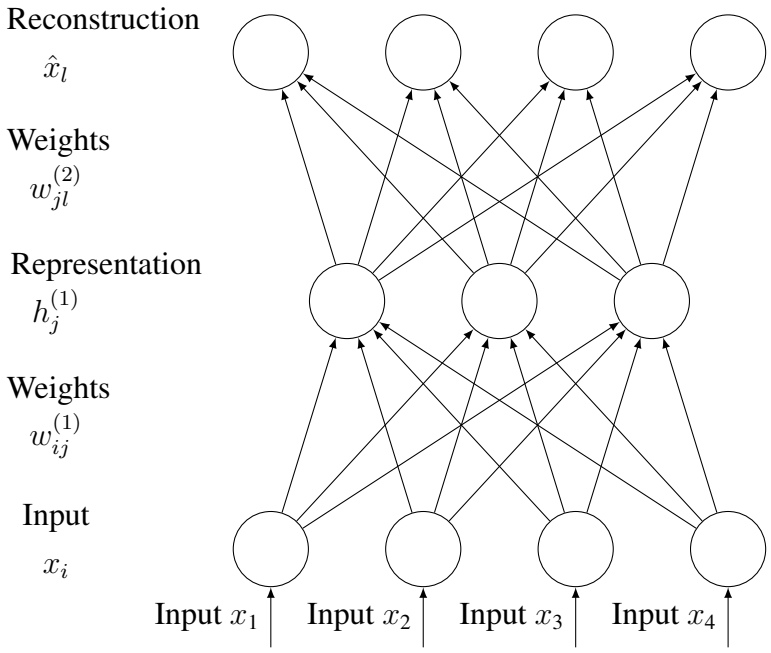


Figure 3.2: An one layer autoencoder with 4 input nodes and 3 representation nodes.

With this setup, we have a neural network described as in Figure 3.2. In this structure, we can optimize for the optimal parameters $\{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}\}$ using the gradient descent approach from feedforward neural networks 2.1. In practice, tied weights condition $\mathbf{w}^{(1)} = [\mathbf{w}^{(2)}]^\top$ is often enforced to start the optimization. Observe that when the weights are tied and the non-linear activation function is sigmoid, we have a striking similarity with the RBM: the representation $\mathbf{h}$ is exactly the probability of binary hidden layer sampled as 1.

However as [13] explained, pure compression retains insufficient information, especially when compared to RBMs; therefore the authors introduced a new optimization criterion: reconstruction from noisy inputs. Formally, we have a corruption function q that creates noisy inputs $\mathbf{v} = q(\mathbf{x})$. A popular choice of q is to randomly set a fraction of the input dimensions to zero.

To motivate denoising autoencoders, we consider an example with 2 inputs, i.e. $\mathbf{x} = \{x_1, x_2\}$, and let $x_1 \approx \phi(x_2)$ for some bijection ϕ. When x_1 is set to zero due to corruption, it remains possible to reconstruct x_1 by learning the relationship between x_1 and x_2, which gives us $\hat{x}_1 = \phi(x_2)$. Similarly, when x_2 is corrupted, ideally we can have $\hat{x}_2 = \phi^{-1}(x_1)$.

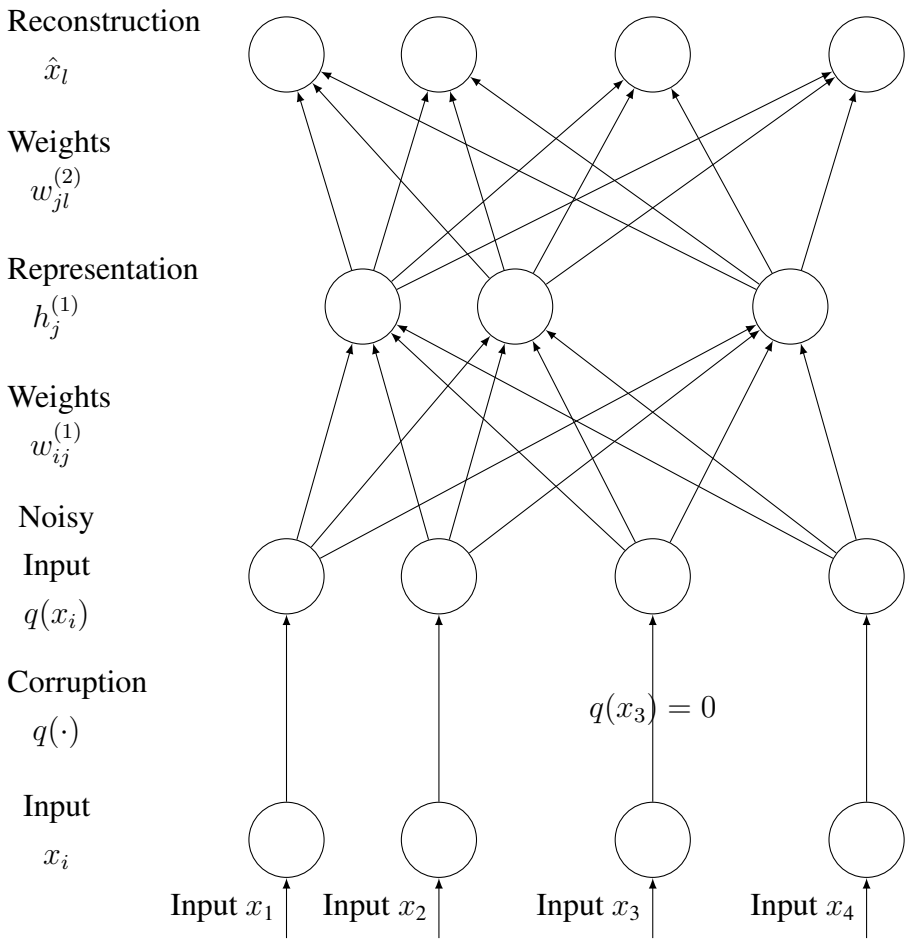


Figure 3.3: An one layer denoising autoencoder with 4 input nodes, 3 representation nodes, and symmetric weights. In this case, we have the third input corrupted by setting to zero.

Bibliography

[1] Michael A Bailey, Jeffrey S Rosenthal, and Albert H Yoon. Grades and incentives: assessing competing grade point average measures and postgraduate outcomes. *Studies in Higher Education*, (ahead-of-print):1–15, 2014.

[2] C Bishop. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn, 2007.

[3] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[4] Andrey Feuerverger, Yu He, Shashi Khatri, et al. Statistical significance of the netflix challenge. *Statistical Science*, 27(2):202–231, 2012.

[5] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[6] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[7] Andriy Mnih and Ruslan Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2007.

[8] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[9] Ruslan Salakhutdinov. *Learning deep generative models*. PhD thesis, University of Toronto, 2009.

[10] Ruslan Salakhutdinov and Andriy Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *Proceedings of the 25th international conference on Machine learning*, pages 880–887. ACM, 2008.

[11] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[13] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.