Introduction to Restricted Boltzmann Machines

by Edwin Chen on Mon 18 July 2011

Suppose you ask a bunch of users to rate a set of movies on a 0-100 scale. In classical factor analysis, you could then try to explain each movie and user in terms of a set of latent *factors*. For example, movies like Star Wars and Lord of the Rings might have strong associations with a latent science fiction and fantasy factor, and users who like Wall-E and Toy Story might have strong associations with a latent Pixar factor.

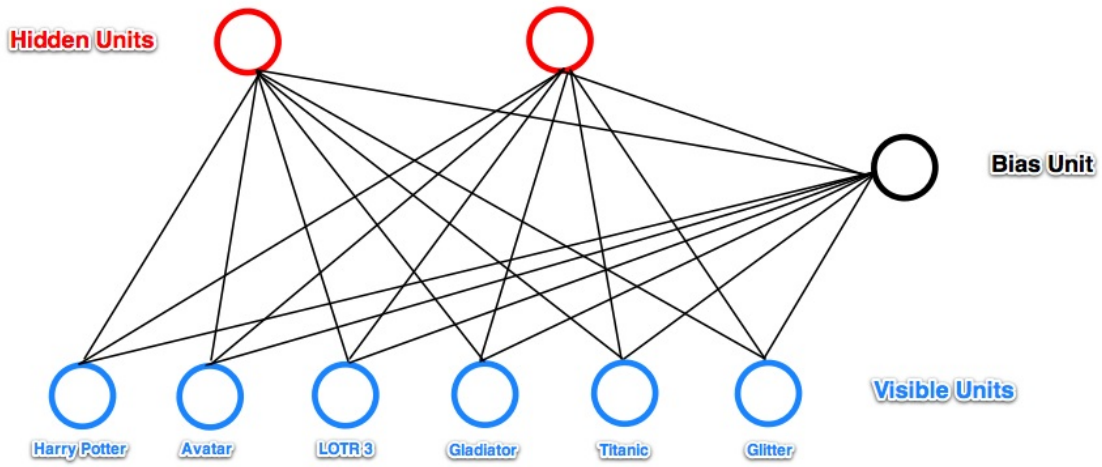
Restricted Boltzmann Machines essentially perform a *binary* version of factor analysis. (This is one way of thinking about RBMs; there are, of course, others, and lots of different ways to use RBMs, but I'll adopt this approach for this post.) Instead of users rating a set of movies on a continuous scale, they simply tell you whether they like a movie or not, and the RBM will try to discover latent factors that can explain the activation of these movie choices.

More technically, a Restricted Boltzmann Machine is a **stochastic neural network** (*neural network* meaning we have neuron-like units whose binary activations depend on the neighbors they're connected to; *stochastic* meaning these activations have a probabilistic element) consisting of:

- One layer of **visible units** (users' movie preferences whose states we know and set);
- One layer of **hidden units** (the latent factors we try to learn); and
- A bias unit (whose state is always on, and is a way of adjusting for the different inherent popularities of each movie).

Furthermore, each visible unit is connected to all the hidden units (this connection is undirected, so each hidden unit is also connected to all the visible units), and the bias unit is connected to all the visible units and all the hidden units. To make learning easier, we restrict the network so that no visible unit is connected to any other visible unit and no hidden unit is connected to any other hidden unit.

For example, suppose we have a set of six movies (Harry Potter, Avatar, LOTR 3, Gladiator, Titanic, and Glitter) and we ask users to tell us which ones they want to watch. If we want to learn two latent units underlying movie preferences – for example, two natural groups in our set of six movies appear to be SF/fantasy (containing Harry Potter, Avatar, and LOTR 3) and Oscar winners (containing LOTR 3, Gladiator, and Titanic), so we might hope that our latent units will correspond to these categories – then our RBM would look like the following:



(Note the resemblance to a factor analysis graphical model.)

State Activation

Restricted Boltzmann Machines, and neural networks in general, work by updating the states of some neurons given the states of others, so let's talk about how the states of individual units change. Assuming we know the connection weights in our RBM (we'll explain how to learn these below), to update the state of unit i:

- Compute the **activation energy** $a_i = \sum_j w_{ij} x_j$ of unit i, where the sum runs over all units j that unit i is connected to, w_{ij} is the weight of the connection between i and j, and x_j is the 0 or 1 state of unit j. In other words, all of unit i's neighbors send it a message, and we compute the sum of all these messages.

- Let $p\_i = \sigma(a\_i)$, where $\sigma(x) = 1/(1 + exp(-x))$ is the logistic function. Note that p_i is close to 1 for large positive activation energies, and p_i is close to 0 for negative activation energies.
- We then turn unit i on with probability p_i, and turn it off with probability $1 - p\_i$.
- (In layman's terms, units that are positively connected to each other try to get each other to share the same state (i.e., be both on or off), while units that are negatively connected to each other are enemies that prefer to be in different states.)

For example, let's suppose our two hidden units really do correspond to SF/fantasy and Oscar winners.

- If Alice has told us her six binary preferences on our set of movies, we could then ask our RBM which of the hidden units her preferences activate (i.e., ask the RBM to explain her preferences in terms of latent factors). So the six movies send messages to the hidden units, telling them to update themselves. (Note that even if Alice has declared she wants to watch Harry Potter, Avatar, and LOTR 3, this doesn't guarantee that the SF/fantasy hidden unit will turn on, but only that it will turn on with high *probability*. This makes a bit of sense: in the real world, Alice wanting to watch all three of those movies makes us highly suspect she likes SF/fantasy in general, but there's a small chance she wants to watch them for other reasons. Thus, the RBM allows us to *generate* models of people in the messy, real world.)
- Conversely, if we know that one person likes SF/fantasy (so that the SF/fantasy unit is on), we can then ask the RBM which of the movie units that hidden unit turns on (i.e., ask the RBM to generate a set of movie recommendations). So the hidden units send messages to the movie units, telling them to update their states. (Again, note that the SF/fantasy unit being on doesn't guarantee that we'll always recommend all three of Harry Potter, Avatar, and LOTR 3 because, hey, not everyone who likes science fiction liked Avatar.)

Learning Weights

So how do we learn the connection weights in our network? Suppose we have a bunch of training examples, where each training example is a binary vector with six elements corresponding to a user's movie preferences. Then for each epoch, do the following:

- Take a training example (a set of six movie preferences). Set the states of the visible units to these preferences.
- Next, update the states of the hidden units using the logistic activation rule described above: for the jth hidden unit, compute its activation energy $a\_j = \sum\_i w\_{ij} x\_i$, and set x_j to 1 with probability $\sigma(a\_j)$ and to 0 with probability $1 - \sigma(a\_j)$. Then for each edge e_ij, compute $Positive(e\_ij) = x\_i * x\_j$ (i.e., for each pair of units, measure whether they're both on).
- Now **reconstruct** the visible units in a similar manner: for each visible unit, compute its activation energy a_i, and update its state. (Note that this *reconstruction* may not match the original preferences.) Then update the hidden units again, and compute $Negative(e\_ij) = x\_i * x\_j$ for each edge.
- Update the weight of each edge e_ij by setting $w\_ij = w\_ij + L * (Positive(e\_ij) - Negative(e\_ij))$, where L is a learning rate.
- Repeat over all training examples.

Continue until the network converges (i.e., the error between the training examples and their reconstructions falls below some threshold) or we reach some maximum number of epochs.

Why does this update rule make sense? Note that

- In the first phase, $Positive(e\_ij)$ measures the association between the ith and jth unit that we *want* the network to learn from our training examples;
- In the "reconstruction" phase, where the RBM generates the states of visible units based on its hypotheses about the hidden units alone, $Negative(e\_ij)$ measures the association that the network *itself* generates (or "daydreams" about) when no units are fixed to training data.

So by adding $Positive(e\_ij) - Negative(e\_ij)$ to each edge weight, we're helping the network's daydreams better match the reality of our training examples.

(You may hear this update rule called **contrastive divergence**, which is basically a funky term for "approximate gradient descent".)

Examples

I wrote a simple RBM implementation in Python (the code is heavily commented, so take a look if you're still a little fuzzy on how everything works), so let's use it to walk through some examples.

First, I trained the RBM using some fake data.

- Alice: (Harry Potter = 1, Avatar = 1, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0). Big SF/fantasy fan.
- Bob: (Harry Potter = 1, Avatar = 0, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0). SF/fantasy fan, but doesn't like Avatar.

- Carol: (Harry Potter = 1, Avatar = 1, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0). Big SF/fantasy fan.
- David: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0). Big Oscar winners fan.
- Eric: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0). Oscar winners fan, except for Titanic.
- Fred: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0). Big Oscar winners fan.

The network learned the following weights:

	Bias Unit	Hidden 1	Hidden 2
Bias Unit	-0.08257658	-0.19041546	1.57007782
Harry Potter	-0.82602559	-7.08986885	4.96606654
Avatar	-1.84023877	-5.18354129	2.27197472
LOTR 3	3.92321075	2.51720193	4.11061383
Gladiator	0.10316995	6.74833901	-4.00505343
Titanic	-0.97646029	3.25474524	-5.59606865
Glitter	-4.44685751	-2.81563804	-2.91540988

Note that the first hidden unit seems to correspond to the Oscar winners, and the second hidden unit seems to correspond to the SF/fantasy movies, just as we were hoping.

What happens if we give the RBM a new user, George, who has (Harry Potter = 0, Avatar = 0, LOTR 3 = 0, Gladiator = 1, Titanic = 1, Glitter = 0) as his preferences? It turns the Oscar winners unit on (but not the SF/fantasy unit), correctly guessing that George probably likes movies that are Oscar winners.

What happens if we activate only the SF/fantasy unit, and run the RBM a bunch of different times? In my trials, it turned on Harry Potter, Avatar, and LOTR 3 three times; it turned on Avatar and LOTR 3, but not Harry Potter, once; and it turned on Harry Potter and LOTR 3, but not Avatar, twice. Note that, based on our training examples, these generated preferences do indeed match what we might expect real SF/fantasy fans want to watch.

Modifications

I tried to keep the connection-learning algorithm I described above pretty simple, so here are some modifications that often appear in practice:

- Above, $Negative(e\_ij)$ was determined by taking the product of the ith and jth units after reconstructing the visible units *once* and then updating the hidden units again. We could also take the product after some larger number of reconstructions (i.e., repeat updating the visible units, then the hidden units, then the visible units again, and so on); this is slower, but describes the network's daydreams more accurately.
- Instead of using $Positive(e\_ij) = x\_i*x\_j$, where x_i and x_j are binary 0 or 1 *states*, we could also let x_i and/or x_j be activation *probabilities*. Similarly for $Negative(e\_ij)$.
- We could penalize larger edge weights, in order to get a sparser or more regularized model.
- When updating edge weights, we could use a momentum factor: we would add to each edge a weighted sum of the current step as described above (i.e., $L*(Positive(e\_ij) - Negative(e\_ij))$ and the step previously taken.
- Instead of using only one training example in each epoch, we could use *batches* of examples in each epoch, and only update the network's weights after passing through all the examples in the batch. This can speed up the learning by taking advantage of fast matrix-multiplication algorithms.

Further

If you're interested in learning more about Restricted Boltzmann Machines, here are some good links.

- A Practical guide to training restricted Boltzmann machines, by Geoffrey Hinton.
- A talk by Andrew Ng on Unsupervised Feature Learning and Deep Learning.
- Restricted Boltzmann Machines for Collaborative Filtering. I found this paper hard to read, but it's an interesting application to the Netflix Prize.
- Geometry of the Restricted Boltzmann Machine. A very readable introduction to RBMs, "starting with the observation that its Zariski closure is a Hadamard power of the first secant variety of the Segre variety of projective lines". (I kid, I kid.)

Edwin Chen

Email
Twitter
Github
Google+
LinkedIn
Quora

Atom / RSS

Recent Posts

Moving Beyond CTR: Better Recommendations Through Human Evaluation

Propensity Modeling, Causal Inference, and Discovering Drivers of Growth

Improving Twitter Search with Real-Time Human Computation

Edge Prediction in a Social Graph: My Solution to Facebook's User Recommendation Contest on Kaggle

Soda vs. Pop with Twitter

Infinite Mixture Models with Nonparametric Bayes and the Dirichlet Process

Instant Interactive Visualization with d3 + ggplot2

Movie Recommendations and More via MapReduce and Scalding

Quick Introduction to ggplot2

Introduction to Conditional Random Fields