Boltzmann's Cuisine

Gurvan L'Hostis*, Ronan Riochet†

13 January 2016

1 Introduction

We worked on Restricted Boltzmann machines as presented in Fischer and Igel's article [1]. Restricted Boltzmann machines (RBMs) are probabilistic graphical models which are a variation of Boltzmann Machines designed to allow efficient training. We applied this model to a recipe classification problem, using data from a recipe sharing website called Yummly [1]. In a first section we will present the theory about Restricted Boltzmann machines, introduce inference methods we use, along present dropout as a regularisation method. In the second part, we present the application of RBMs to our problem, the dataset and the results we obtained. Finally, we will show how we used the properties of Restricted Boltzmann machines to generate new recipes.

2 Restricted Boltzmann Machine

2.1 From Ising model to RBM

The Ising model is an energy-based model, where we associate an energy to a pair of activated nodes i and j, named $w_{i,j}$. The total energy of the model is the sum of energies associated to all pairs of variables:

$$E(v) = -\sum_{i<j<n} v_i w_{i,j} v_j \tag{1}$$

The probability of a of given state v is $P(v) = \frac{1}{Z}\exp(-E(v))$ with Z the partition function which is a normalisation constant.

Boltzmann machines as probabilistic neural networks were introduced by Hinton and Sejnowski in [2]. Koller and Friedman in [3] started considering them as probabilistic graphical models, using their capability to learn aspects of an unknown probability distributions based on samples from this distribution. The general structure of Boltzmann machines makes them difficult to train, and that is why Restricted Boltzmann machines where introduced.

In Restricted Boltzmann machines, only weights between visible and hidden variables have non-zeros values (see figure 1), which leads to the following energy:

$$E(v,h) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j} h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \tag{2}$$

The model is trained via gradient ascent, optimizing the likelihood of the model with respect to the weight $w_{i,j}$. The absence of connections between hidden variables makes hidden variables independent given the state of visible variables and vice versa:

*gurvan.lhostis@polytechnique.edu
†ronan.riochet@polytechnique.edu
[1] http://www.yummly.com/

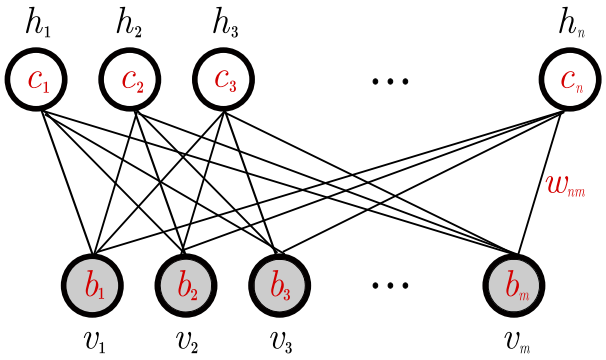

Figure 1: A restricted Boltzmann machine. Edges represent the weights of the model and biaises are associated with each variable.

$$p(h|v) = \prod_{i=1}^{n} p(h_i|v), \;\; p(v|h) = \prod_{j=1}^{m} p(v_i|h) \tag{3}$$

Summing on hidden variables:

$$\begin{aligned} p(v) &= \frac{1}{Z} \sum_h p(v,h) \\ &= \frac{1}{Z} \sum_h \exp(-E(v,h)) \\ &= \frac{1}{Z} \prod_{j=1}^{m} \exp(b_j v_j) \prod_{i=1}^{n} \left(1 + \exp\left(c_i + \sum_{j=1}^{m} w_{i,j} v_j\right)\right) \end{aligned}$$

2.2 Training

The learning is done by gradient ascent on (a proxy for) the log-likelihood. Since RBM are a form of Markov Random Fields with latent variables, we can directly take the associated expression for the derivative of the log-likelihood:

$$\frac{\partial \ln \mathcal{L}(\theta|v)}{\partial \theta} = -\sum_h p(h|v) \frac{\partial E(v,h)}{\partial \theta} + \sum_{v,h} p(v,h) \frac{\partial E(v,h)}{\partial \theta}$$

Summing over all possible states has an exponential complexity with that form and this is where the properties of RBM get interesting. We have

$$\begin{aligned} \frac{\partial \ln \mathcal{L}(\theta|v)}{\partial w_{ij}} &= -\sum_h p(h|v) \frac{\partial E(v,h)}{\partial w_{ij}} + \sum_{v,h} p(v,h) \frac{\partial E(v,h)}{\partial w_{ij}} \\ &= \sum_h p(h|v) h_i v_j - \sum_v p(v) \sum_h p(h|v) h_i v_j \\ &= p(H_i|v) v_j - \sum_v p(v) p(H_i = 1|v) v_j \end{aligned}$$

Thanks to the conditional independence properties, we obtain an expression that is tractable.

The expression for when we learn from a training set S (of empirical distribution q) is

$$\sum_{v \in S} \frac{\partial \ln \mathcal{L}(\theta|v)}{\partial w_{ij}} \propto < v_i h_j >_{p(h|v)q(v)} - < v_i h_j >_{p(v,h)}$$
$$\propto < v_i h_j >_{\text{data}} - < v_i h_j >_{\text{model}}$$

There is still the issue of summing across the RBM distribution, this is approximated by sampling methods.

Contrastive divergence We approximate the model term through Gibbs sampling. Sampling from an RBM is cheap because of the conditional independence properties; the visible vector and the hidden vector of states can be sampled all at once.

Gibbs sampling usually requires a lot of iterations to produce good approximations, but it is experimentally proven that very few iterations suffice ($k = 1$ is often used).

The method we develop from this idea of sampling is called *contrastive divergence*. A step of CD for one training sample consists in sampling an h from that sample and a v from that h, repeated k times. The sampled visible vector is used as the model sample in the gradient ascent formula and the sum over hidden states is reduced to the hidden activation probabilities (see figure 2).

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \dots, V_m, H_1, \dots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \dots, n$, $j = 1, \dots, m$

init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \dots, n$, $j = 1, \dots, m$
forall the $\boldsymbol{v} \in S$ **do**
  $\boldsymbol{v}^{(0)} \leftarrow \boldsymbol{v}$
  for $t = 0, \dots, k-1$ **do**
    for $i = 1, \dots, n$ **do** sample $h_i^{(t)} \sim p(h_i \,|\, \boldsymbol{v}^{(t)})$
    for $j = 1, \dots, m$ **do** sample $v_j^{(t+1)} \sim p(v_j \,|\, \boldsymbol{h}^{(t)})$
  for $i = 1, \dots, n$, $j = 1, \dots, m$ **do**
    $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \,|\, \boldsymbol{v}^{(0)}) \cdot v_j^{(0)} - p(H_i = 1 \,|\, \boldsymbol{v}^{(k)}) \cdot v_j^{(k)}$
    $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$
    $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \,|\, \boldsymbol{v}^{(0)}) - p(H_i = 1 \,|\, \boldsymbol{v}^{(k)})$

Figure 2: Contrastive divergence

Persistent contrastive divergence A variation of contrastive divergence consists in starting the sampling chain from the end of the previous one instead of starting from a data sample every time.

The idea behind this is that we stay closer to the RBM distribution by not getting back to our empirical distribution; we used this method in our experiments.

2.3 Dropout

Dropout is an anti-overfitting technique for neural networks first described in 2006 that is now omnipresent in the field, improving the results on most reference problems.

The idea, applied to RBM, is to drop some hidden units (along with their connections) with probability p (see Figure 3). The fact that all hidden units are not trained at the same time prevents them from co-adaptating. It hence can be considered as a regularisation method for graphical models.

In their article [4], Hinton et al. make an analogy between dropout and the role of sexual reproduction in evolution. Indeed, sexual reproduction involves taking half of the

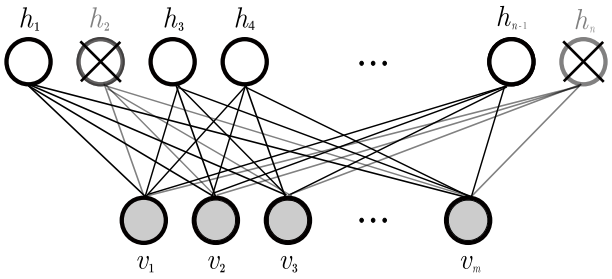

Figure 3: An instance of dropout on an RBM: hidden units are dropped with a probability p for every learning sample, for example h_2, ... and h_n. The learning algorithm is thus applied only to this thinned RBM for that given sample.

genes of one parent, half of the genes of the other parent and a part of random mutation to produce a new offspring. At first sight, an asexual reproduction where sequences of genes that works the best together are passed to the new offspring may be seen as optimal for evolution. However, sexual reproduction is likely to break up these co-adaptation between gene and make the new gene sequence more robust. Similarly, dropout should make the model more robust and create useful features on its own, without relying on other hidden units to correct its mistakes. The learnt features are less likely to rely on particular idiosyncrasies of the training data.

We experimented with dropout on a reduced dataset, taking probability $p = 0.5$, which led to consistent increases in accuracies by approximately 2% which is in keeping with what is described in the literature. For this reason we kept using it in all our models and did not consider it a hyperparameter anymore.

2.4 Classification with an RBM

The problem we chose to tackle (see next section) is a classification problem. We therefore need to adapt the RBM which is made for unsupervised learning in its simple form.

Feature learning One way of doing so is to train the RBM in an unsupervised fashion, and then to then use it as a neural network. We use the hidden activation probabilities as learnt features which we input on a supplementary layer (see figure 4).

That layer we add is a simple softmax regression which we train over the features corresponding to the samples of the dataset.

We can then use this stack as a two-layer neural network to make predictions: the unlabelled sample is fed to the RBM, we retrieve the hidden activation probabilities, feed them to the softmax regression and retrieve the most probable label.

Softmax unit Another way is to directly encode the labels in the samples with a softmax unit (see figure 5). The only specificity of a softmax unit is that during sampling of v from h, one and only one visible unit is activated among the ones that are grouped under the softmax unit (with softmax sampling). Using RBMs as such makes them learn the joint distribution of cuisine labels and recipes directly.

Prediction is bit more subtle here: the predicted label is the one that yields the highest label+data joint probability. This computation is linear in the number of labels:

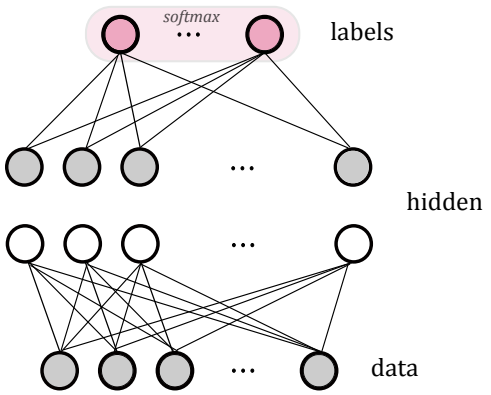

Figure 4: An RBM with a stacked softmax regression taking the hidden activation probabilities as inputs.

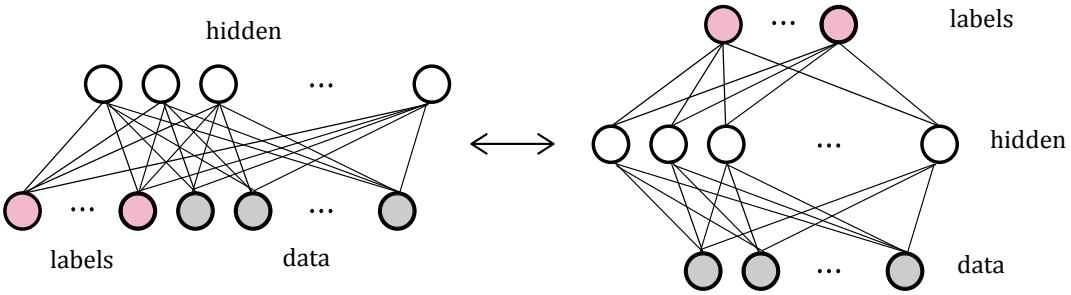

Figure 5: An RBM with a softmax unit to represent labels. This can be interpreted as a supplementary layer which is fed simultaneaously with the data layer during training.

$$
\begin{aligned}
P(v_{label}|v_{data}) &= \frac{p([v_{label}|v_{data}])}{p(v_{data})} \\
&= \frac{\sum_h p([v_{label}|v_{data}]), h)}{\sum_{v'_{label}} \sum_h p([v_{label'}|v_{data}]), h)} \\
P(v_{label}|v_{data}) &\propto \exp(a_{label}) * \prod_i (1 + \exp(b_i + (W.[v_{label}|v_{data}])_i))
\end{aligned}
$$

3 Experiments

3.1 Implementation insights

Theano Our dataset being quite large and high-dimensional, we focused on having an efficient implementation to allow fast training. *Theano* is a library for `python` that is designed for training neural networks in an efficient way because it provides formal computation tools and numerical stability tricks, which is why we chose to use it for our RBM implementation in the beginning.

However, it was quite difficult to use and we returned to a plain python/numpy implementation when we experimented with dropout. This pure python implementation turned out to be five times faster than the theano one. After investigations, it appears that theano is useful if and only if training is to be done on a GPU, which we could not do.

As a benchmark for speed (and performance), we compared our RBM implementation to that of the library *Scikit-learn* which is also based on pure pyhthon/numpy on a reduced dataset. Ours was a little faster and provided better results, which can be explained by the fast that we implemented dropout and the library version did not.

Numerical stability As soon as we started using large RBMs (above 1000 hidden units), we began having numerical stability issues.

For example, the unnormalised energies used to make predictions reach infinity and we have to make computations in the log space. There is thus a $\log(1 + \exp(x))$ expression which is unstable, we used Numpy's built-in functions such as `logaddexp` to deal with this.

Our code thus contains a main class for RBMs as well as a subclass for RBM with a softmax unit for labels, a preprocessing script and a main script from which the training epochs are controlled.

3.2 Dataset

Our data is a set of recipes extracted from Yummly, a recipe sharing website, which was used for a now-closed Kaggle competition[2]. This dataset is made of 39774 recipes made of 6677 so-called ingredients taken from 20 types of cuisine (see figure 6).

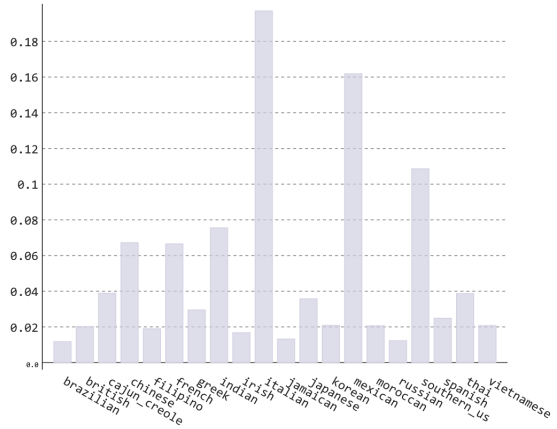

Figure 6: Distribution of recipes in the twenty types of cuisine.

The ingredients are in a string format taken directly from user inputs, they are not very clean: some ingredients appear several times because they are present two languages, because of brand prefixing or simply because of typos.

Inspired by what we read on the forum of the competition, we applied the following preprocessing pipeline:

- string normalisation (lowercase, remove special chars);
- removing quantities/units (grams, oz. etc);
- stemming with the package `nltk` (iced -> ice, peppers -> pepper);
- string splitting (slivered almond -> slivered, almond);
- one hot encoding of full ingredients + separate words.

This results in a 10k dimensional dataset, we did our first experiments on a reduced version containing only the 1000 most frequent features.

[2]https://www.kaggle.com/c/whats-cooking

3.3 Results

We split our original training data into `train_data` and `test_data`. We trained all our models on the `train_data` and used `test_data` to assess their performance. Given the time needed to train our models (several days), we could not use cross-validation on the full dataset.

We trained four different models, whose convergences are shown in Figure 7. The first model (in red) is a supervised RBM, as introduced above, trained with 2000 hidden units. The second (in green) is an unsupervised RBM trained with 2000 units, and combined with a logistic regression from scikit-learn. The third and fourth models are smaller models: in mauve a supervised RBM train with 200 hidden units and in yellow an unsupervised RBM trained with 2000 hidden units but on a smaller dataset (approx. 600 features).

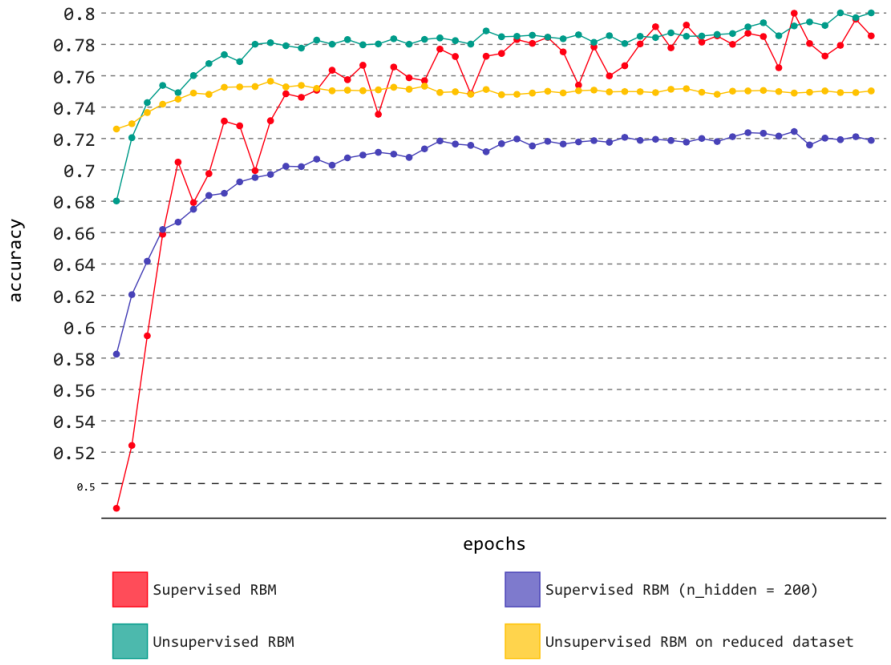

Figure 7: Accuracies on `test_data` for the 50 first epochs.

In the Kaggle competition, a test dataset of 10k samples for evaluation purposes is given, which we call `validation_data` to avoid confusion. We tested our two best models only once on this `validation_data`, along with our benchmarks `RandomForest` and `LogisticRegression()` from scikit-learn. The following table present these results:

Supervised RBM	0.77514
Unsupervised RBM	0.79334
RandomForest	0.75885
LogisticRegression	0.78530

We may first remark that the supervised RBM gave a lower accuracy than logistic regression, which is quite a shame after having trained for a full week. It should be remarked that it was trained only on our train date whereas other models where trained on train+test data.

Our unsupervised RBM, which we re-trained on train+test data, produced a better accuracy than the logistic regression but is not on par with the best performances in the competition leaderboard. Our submission would classify as 316th out of 1400 participations.

3.4 Recipe Generation

A interesting property of RBM is that they are generative models; they were for example used to sample images of digits after training on the MNIST dataset [5]. We therefore tried to sample recipes from our softmax RBM with a similar algorithm:

Initialise v to 0 everywhere;
REPEAT k times:
  $v_{1-20} = v_{label}$
  $h \sim p(h|v)$
  $v \sim p(v|h)$
RETURN ingredients of v.

The label part of the visible vector is erased and replaced by our desired label before every hidden sampling

Here are a few generated recipes (we verified that they did not exist in the dataset):

- Chinese: oyster sauce, sugar, dark soy sauce, light soy sauce, gai lan (Chinese broccoli);
- French: strawberries, Cointreau, sugar, creme fraiche;
- Mexican: salsa verde, crema mexican, tortilla chip, chorizo, monterey jack, chorizo sausage, cotija, poblano chile, queso fresco, corn tortilla.

We should remark that most of the time, our sampling returns empty recipes; we guess that this is caused by the sparsity of the inputs. We cannot judge of the aforementioned Chinese and Mexican recipes, but the French one seems quite interesting.

4 Conclusion

Restricted Boltzmann Machines probably are quite an overkill for this recipe classification problem. Indeed, the data is already very organised, which can explain why the hidden variable representation that we learn do not reveal much information, and the best performance in the competition did use simpler models.

However, our work brings at least one remarkable feature with recipe generation which is more interesting than generating images of digits. This reminds one of the power of such probabilistic graphical models and gives hope that RBM along with deep belief networks will someday come back to the forefront of deep learning.

References

[1] Asja Fischer and Christian Igel. "An Introduction to Restricted Boltzmann Machines." In: *CIARP*. Ed. by Luis Álvarez et al. Vol. 7441. Lecture Notes in Computer Science. Springer, 2012, pp. 14–36.

[2] H. Ackley, E. Hinton, and J. Sejnowski. "A learning algorithm for Boltzmann machines". In: *Cognitive Science* (1985), pp. 147–169.

[3] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[4] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958.

[5] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Comput.* 18.7 (July 2006), pp. 1527–1554.