Visually Debugging Restricted Boltzmann Machine Training with a 3D Example

Jason Yosinski YOSINSKI@CS.CORNELL.EDU
Hod Lipson HOD.LIPSON@CORNELL.EDU
Department of Computer Science, Cornell University, Ithaca, NY, USA

Abstract

Restricted Boltzmann Machines (RBMs) are being applied to a growing number of problems with great success. In the process of training an RBM one must pick a number of parameters, but often these parameters are brittle and produce poor performance when slightly off. Here we describe several useful visualizations to assist in choosing appropriate values for these parameters. We also demonstrate a successful application of an RBM to a unique domain: learning a representation of synthetic 3D shapes.

1. Introduction

Restricted Boltzmann Machines (RBMs) have enjoyed recent success in learning features in a number of domains (Hinton & Salakhutdinov, 2006; Bengio et al., 2007; Lee et al., 2008; Ngiam et al., 2011). However, successfully training a Restricted Boltzmann Machines (RBM) is far from a straightforward proposition. There are many tuning parameters that must be carefully chosen. Results are sensitive to these parameters, and picking them correctly is often difficult.

With this in mind, we describe several visualizations that we have found helpful in tuning the requisite parameters. Many are adapted from the very useful paper "A practical guide to training restricted boltzmann machines" (Hinton, 2010).

This guide is aimed toward a novice trainer of RBMs who wishes to spend as little time in the trenches as possible. To this end we give concrete examples of how to implement the plots in the form of code snippets in both Python and Octave / Matlab.

For the remainder of the paper, we assume the RBM is trained using mini-batch based gradient descent using the Contrastive Divergence algorithm (Hinton, 2002).

2. Debugging RBMs

The four presented plots are arranged in roughly the order they should be used. Undesired behavior in earlier plots will produce further undesired behavior in later plots. Thus, debugging should be focused on the first plot showing unexpected behavior. Some of the code is loosely based on the Theano RBM tutorials (Bergstra et al., 2010).

2.1. Code setup

The following imports and initializations are assumed:

Python:

```
from numpy import tanh, fabs, mean, ones
from PIL import Image
from matplotlib.pyplot import hist, title, subplot
def sigmoid(xx):
    return .5 * (1 + tanh(xx / 2.))
```

Octave / Matlab:

```
sigmoid = inline('.5 * (1 + tanh(z / 2.))');
```

2.2. Probability of hidden activation

For a given input example, each hidden binary neuron has a probability of turning on. This probability is deterministic (involves no sampling noise) and is, for obvious reasons, always in [0,1]. Thus, in order to see how the hidden neurons are being used, how often they are likely to be on vs. off, and whether they are correlated, we plot this probability of activation for each hidden neuron for each example input within a mini-batch. These probabilities can be effectively visualized

Presented at the *Representation Learning Workshop, 29th International Conference on Machine Learning*, Edinburgh, Scotland, UK, 2012.

as grayscale values of an image where each row contains the hidden neuron activation probabilities for a single example, and each column contains the probabilities for a given neuron across examples. Figure 1 shows this plot.

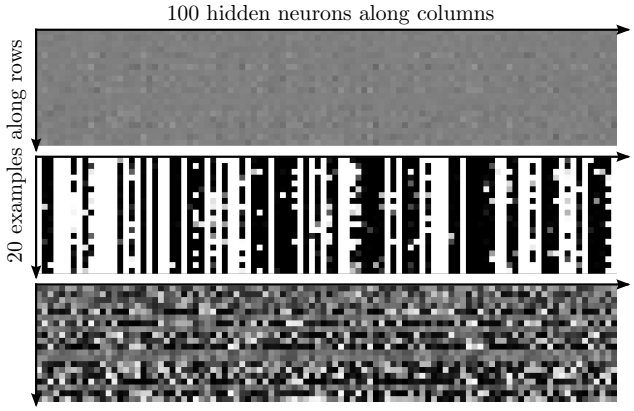

Figure 1. Hidden neuron activation probabilities for the first 100 neurons (of 1,000) and the first 20 example data points (of 50,000), where black represents $p = 0$ and white, $p = 1$. Each row shows different neurons' activations for a given input example, and each column shows a given neuron's activations across many examples. Top: correct dithered gray before training begins. Values are mostly in [.4, .6]. Middle: Values pegged to black or white after one mini-batch. Decrease initial W values or learning rate. Bottom: the learning has converged well after 45 epochs of training.

Make sure to manually set the intensity limits to [0,1] rather than using any autoscale feature (e.g. do not use Matlab's `imagesc` with the default autoscaling behavior).

Given a mini-batch of training examples in X (dimension 20 x 1000), the following code produces the plots in Figure 1.

Python:

```
hMean = sigmoid(dot(X, rbm.W) + rbm.hBias)
image = Image.fromarray(hMean * 256).show()
```

Octave / Matlab:

```
hMean = sigmoid(X*W + repmat(hBias, 20, 1));
imagesc(hMean, [0, 1]);
colormap('gray'); axis('equal');
```

Before any training, the probability plot should be mostly a flat gray, perhaps with a little visible noise. That is, most hidden probabilities should be around .5, with some as low as .4 or as high as .6. If the plot is all black (near 0) or all white (near 1), the weights W or the hidden biases $hBias$ were initialized incorrectly. The weights W should initially be random and centered at 0, and $hBias$ should be 0, or at least centered at 0. If the probability plot contains both pixels pegged to black and pixels pegged to white, then the W has been initialized with values too large. Intuitively, the problem with this case is that all hidden neurons have already determined what features they are looking for before seeing any of the data.

So first initialize the W and $hBias$ such that this plot shows gray before training. It is useful to look at this plot each epoch for the same mini-batch, so one can see how the neuron activations evolve. If we view probability plots for a given mini-batch over time in quick succession (like a video), we can see the effect of training. Once training begins, generally the neurons' activations diverge from gray and converge toward their final shades over the course of only several epochs.

Surprisingly, while the activations converge nearly to their final values after only a few epochs, it is often an order of magnitude longer (say, 20 or 30 epochs) before the reconstruction error decreases. Apparently the neurons decide on their preferred stimulus easily, but then further fine tuning takes much longer.

Occasionally, if the learning rate is too high, the probabilities for a given mini-batch will not converge smoothly. The video view of the probability plot (or the filter plots below) shows this clearly as a flickering that persists for many epochs. The solution is to use a lower learning rate. If the rate is already so low that learning takes a long time, consider a smaller (simpler) input vector (e.g. for images use a smaller patch size).

2.3. Weight Histograms

In addition to the hidden probability plots above, which show the combined effects of W and $hBias$ to produce hidden probabilities, it is often useful to look at the values of $vBias$, W, and $hBias$ on aggregate. Figure 2 shows a set of useful plots: the top three show a histogram of values in $vBias$, W, and $hBias$, respectively, and the bottom three plots show histograms of the most recent (mini-batch) updates to the $vBias$, W, and $hBias$ values. For a quick sanity check without requiring the user to interpret the axis scales, we also show in the title the mean absolute magnitude of the values in each histogram.

Python:

```
def plotit(values):
    hist(values);
    title('mm = %g' % mean(fabs(values)))
subplot(231); plotit(rbm.vBias)
```

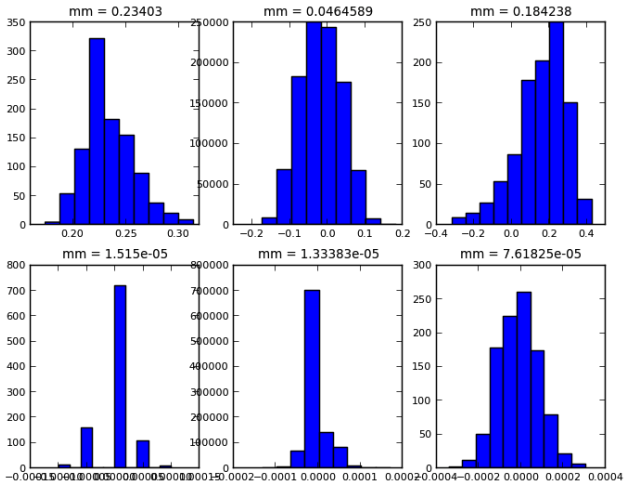

Figure 2. Histograms of *hBias*, W, *vBias* (top row) and the last batch updates to each (bottom row). The mean absolute magnitude of the values is shown above each plot.

```
subplot(232); plotit(rbm.W.flatten()
subplot(232); plotit(rbm.hBias)
subplot(232); plotit(rbm.dvBias)
subplot(232); plotit(rbm.dW.flatten())
subplot(232); plotit(rbm.dhBias)
```

Octave / Matlab:

```
function plotit(values)
    hist(values(:));
    title(sprintf('mm = %g', ...
          mean(mean(abs(values)))));
end
subplot(231); plotit(vBias);
subplot(232); plotit(W);
subplot(233); plotit(hBias);
subplot(234); plotit(dvBias);
subplot(235); plotit(dW);
subplot(236); plotit(dhBias);
```

Under normal, desired conditions in the middle of training, all histograms should look roughly Gaussian in shape, and the mean magnitudes of each of the lower three plots should be smaller than its corresponding upper plot by a factor of 10^2 to 10^4. If the change in weights is too small (i.e. a separation of more than 10^4), then the learning rate can probably be increased. If the change in weights is too large, the learning may explode and the weights diverge to infinity.

Note: occasionally the values of the weights may bifurcate into two separate, Gaussian shaped clusters in the middle of training. If one cluster starts to move off to infinity, the learning rate should be decreased to avoid divergence. However, we sometimes observed the weights to bifurcate into two clusters and then, 2-10 epochs later, to reconverge. We are not sure why this happens, but it seems to have no adverse effect.

Bifurcation notwithstanding, any time than any of the weights, even a small tail, drift off to infinity, the learning should be slowed down or stopped sooner.

2.4. Filters

Once the probability image and weight histograms are behaving satisfactorily, we plot the learned filter for each hidden neuron, one per column of W. Each filter is of the same dimension as the input data, and it is most useful to visualize the filters in the same way as the input data is visualized. In the cases of image patches, we show each filter as an image patch, or in this paper's example, we show the filters as 3D shapes Figure 5. Because readers are far more likely to train on images than 3D voxel data, in Figure 3 we show a 2D slice of our learned 3D filters and give code for the construction of this plot as if image data were used.

It is worth noting that filters may or may not be *sparse*. If filters are sparse, they will respond to very local features. Dense filters, on the other hand, respond to stimuli across the entire filter. Although the two types of filter are qualitatively different, we have observed cases in which both types are successful in learning the underlying density; that is, both types of filter are able to generate reasonable synthetic data using Gibbs sampling.

Python:

```
# Initialize background to dark gray
tiled = ones((11*10, 11*10), dtype='uint8') * 51

for row in xrange(nRows):
  for col in xrange(nCols):
    patch = X[row*nCols + col].reshape((10,10))
    normPatch = ((patch - patch.min()) /
                 (patch.max()-patch.min()+1e-6))
    tiled[row*11:row*11+10, col*11:col*11+10] = \
                              normPatch * 255
Image.fromarray(tiled).show()
```

Octave / Matlab:

```
tiled = ones(11*nRows, 11*nCols) * .2
# dark gray borders
for row = 1:nRows
  for col = 1:nCols
    patch = W(:,(row-1)*nCols+col);
    normPatch = (patch - min(patch)) / ...
                (max(patch)-min(patch)+1e-6);
    tiled((row-1)*11+1:(row-1)*11+10, ...
          (col-1)*11+1:(col-1)*11+10) = ...
```

```
            reshape(normPatch, 10, 10);
    end
end
imagesc(tiled, [0, 1]);
colormap('gray'); axis('equal');
```

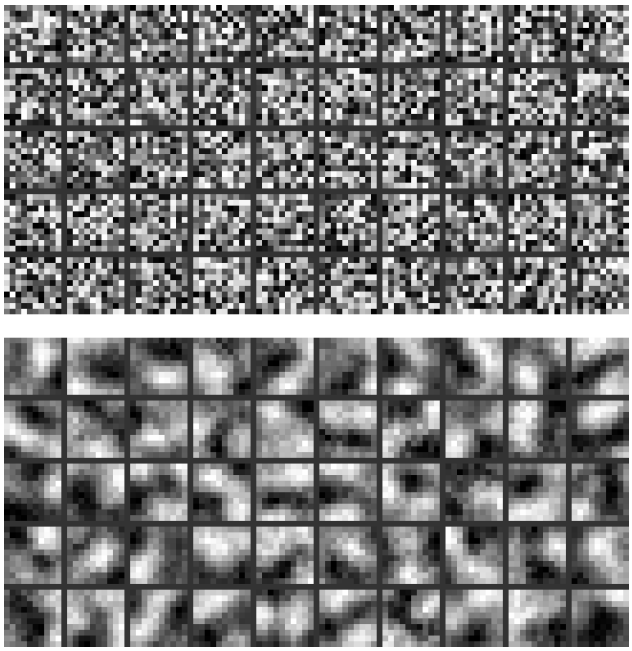

Figure 3. A 2D slice of the filters learned by an RBM for 3D shapes data. Each 10x10 tile shows a single layer (2D) slice of the preferred 3D stimulus for a single hidden neuron. The values within each tile have been normalized to be in $[0, 1]$ for ease of visualization. White regions are areas that the neuron prefers to be filled with voxels, and black areas are not preferred. Top: before learning, filters are random. Bottom: after 45 epochs of learning, filters are strongly locally correlated.

The above code works for the first layer in a Deep Belief Network (DBN) composed of several stacked layers of RBMs. Filters in layers beyond the first represent distributions over hidden neurons, not visible neurons, so visualization is more difficult. These higher level filters can still be visualized by plotting the visible pattern that would maximally activate the higher level neuron, but the connection is less direct.

2.5. Reconstruction error

Plotting the reconstruction error over time is also helpful. It should decrease, often after an initial plateau. If the training diverges, the reconstruction error will increase dramatically. Often we have obtained good results when the reconstruction error drops from its initial higher value to a lower plateau about halfway through training. This plot and code are fairly simple and are omitted for space.

2.6. Typical Training Timeline

In Table 1 we present an example timeline of events that occur during RBM training. This is meant to illustrate the important milestones in training and to provide a *single example* of the relative timing of notable events. It is not intended to provide a template which much be matched exactly, or even approximately.

Epoch	Event
0	Hidden probability plot gray, values .4 to .6
1	Hidden probability plot shows some pattern
2-4	Probability plot showing final pattern which may not significantly change for rest of training
0-45	Filters smoothly resolve over entire period
0-20	Reconstruction error decreases slowly
20-25	Reconstruction error decreases quickly
25-45	Reconstruction error decreases slowly

Table 1. A typical training timeline.

3. Example Problem and Results: 3D Spheres

We now consider an illustrative example of learning a feature representation for a class of synthetic 3D shapes. The 3D shape data was generated in the following manner. First, we define a 10 x 10 x 10 cube containing 1000 voxels. We then choose an x, y, and z location for the center of a sphere randomly from these 1000 voxels, and a radius for the sphere randomly between 1 voxel and 1/3 of the width (3.33 voxels). We then paint this sphere onto the 10 x 10 x 10 voxel canvas. Portions of the sphere that would fall outside the canvas are ignored, so partial spheres are common. Figure 4 shows example spheres from the dataset.

We then train an RBM on 50,000 example spheres from this dataset. The training parameters are shown in Table 2.

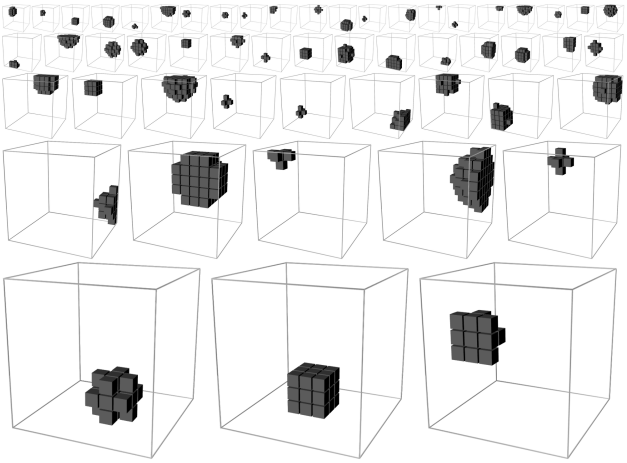

Figure 4. Exemplars from synthetic 3D dataset used for this study. Each sphere has a random (x, y, z) location and random width between 1 and 3.33 voxels.

Parameter	Value
Visible neurons	1000 binary
Hidden neurons	400 binary
size of a mini batch	20
Epochs	45
Learning rate	.001
Initial $vBias$	0
Initial $hBias$	0
Initial W	uniform(-.022, .022)
Weight decay	none
Momentum	none
Sampling method	CD-1

Table 2. Training parameters used to learn a representation of 3D shapes in Section 3

The filters learned after 45 epochs of training are shown in Figure 5. Visualizing 3D shapes is difficult, but we can get a feel for the filters by plotting a solid voxel where the filter's response is high and increasingly transparent voxels when the response is smaller.

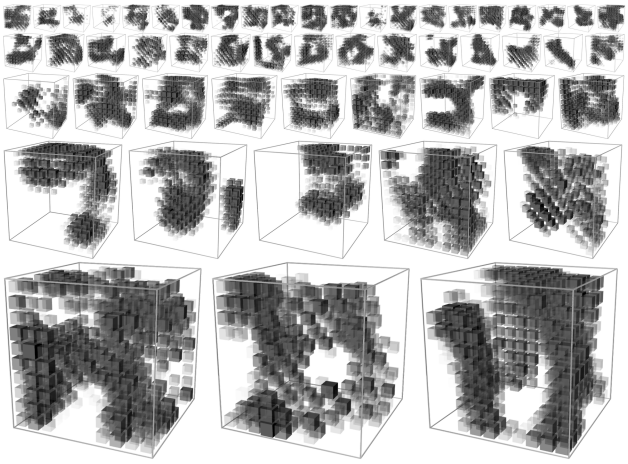

Figure 5. Filters learned by the RBM, i.e. columns of the W weight matrix. Voxels where the filter's response is high are opaque, and voxels where the response is lower become increasingly transparent.

Finally, in Figure 6 we show example Markov Chain Monte Carlo (MCMC) draws from the learned distribution using Gibbs sampling.

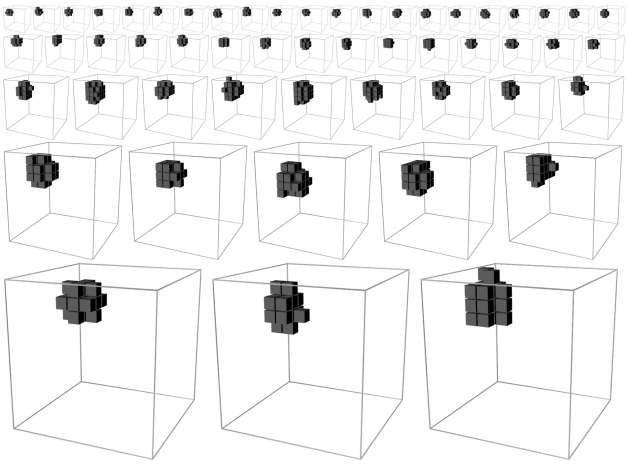

Figure 6. MCMC samples drawn from the learned distribution of shapes. The MCMC mixing rate is low, so the samples are highly correlated, but note that the generated shapes are all nearly perfectly spherical.

4. Conclusion

RBMs are difficult to train because many parameters must be set correctly. We have shown several visualizations that assist in picking these parameters and have provided code that can be used to generate each. We have also demonstrated a simple, though atypical, application of RBMs to learn a representation of synthetic 3D shapes, with good results.

5. Acknowledgements

This work was supported in part by NSF CDI Grant ECCS 0941561 and the DARPA Open Manufacturing program. The content of this paper is solely the responsibility of the authors and does not necessarily represent the official views of the sponsoring organizations.

References

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., and Montreal, U. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. URL http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf. Oral Presentation.

Hinton, G. A practical guide to training restricted boltzmann machines. *Momentum*, 9:1, 2010.

Hinton, G.E. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

Hinton, G.E. and Salakhutdinov, R.R. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504, 2006.

Lee, H., Ekanadham, C., and Ng, A. Sparse deep belief net model for visual area v2. *Advances in neural information processing systems*, 20:873–880, 2008.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A.Y. Multimodal deep learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning. 2010*, 2011.