Department of Computer Science
University of Toronto
http://learning.cs.toronto.edu

6 King's College Rd, Toronto
M5S 3G4, Canada
fax: +1 416 978 1455

August 2, 2010

UTML TR 2010–003

A Practical Guide to Training Restricted Boltzmann Machines

Version 1

Geoffrey Hinton
Department of Computer Science, University of Toronto

A Practical Guide to Training Restricted Boltzmann Machines

Version 1

Geoffrey Hinton
Department of Computer Science, University of Toronto

Contents

[1]

[1] If you make use of this technical report to train an RBM, please cite it in any resulting publication.

1 Introduction

Restricted Boltzmann machines (RBMs) have been used as generative models of many different types of data including labeled or unlabeled images (Hinton et al., 2006a), windows of mel-cepstral coefficients that represent speech (Mohamed et al., 2009), bags of words that represent documents (Salakhutdinov and Hinton, 2009), and user ratings of movies (Salakhutdinov et al., 2007). In their conditional form they can be used to model high-dimensional temporal sequences such as video or motion capture data (Taylor et al., 2006) or speech (Mohamed and Hinton, 2010). Their most important use is as learning modules that are composed to form deep belief nets (Hinton et al., 2006a).

RBMs are usually trained using the contrastive divergence learning procedure (Hinton, 2002). This requires a certain amount of practical experience to decide how to set the values of numerical meta-parameters such as the learning rate, the momentum, the weight-cost, the sparsity target, the initial values of the weights, the number of hidden units and the size of each mini-batch. There are also decisions to be made about what types of units to use, whether to update their states stochastically or deterministically, how many times to update the states of the hidden units for each training case, and whether to start each sequence of state updates at a data-vector. In addition, it is useful to know how to monitor the progress of learning and when to terminate the training.

For any particular application, the code that was used gives a complete specification of all of these decisions, but it does not explain why the decisions were made or how minor changes will affect performance. More significantly, it does not provide a novice user with any guidance about how to make good decisions for a new application. This requires some sensible heuristics and the ability to relate failures of the learning to the decisions that caused those failures.

Over the last few years, the machine learning group at the University of Toronto has acquired considerable expertise at training RBMs and this guide is an attempt to share this expertise with other machine learning researchers. We are still on a fairly steep part of the learning curve, so the guide is a living document that will be updated from time to time and the version number should always be used when referring to it.

2 An overview of Restricted Boltzmann Machines and Contrastive Divergence

Skip this section if you already know about RBMs

Consider a training set of binary vectors which we will assume are binary images for the purposes of explanation. The training set can be modeled using a two-layer network called a "Restricted Boltzmann Machine" (Smolensky, 1986; Freund and Haussler, 1992; Hinton, 2002) in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections. The pixels correspond to "visible" units of the RBM because their states are observed; the feature detectors correspond to "hidden" units. A joint configuration, $(\mathbf{v}, \mathbf{h})$ of the visible and hidden units has an energy (Hopfield, 1982) given by:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{1}$$

where v_i, h_j are the binary states of visible unit i and hidden unit j, a_i, b_j are their biases and w_{ij} is the weight between them. The network assigns a probability to every possible pair of a visible and a

hidden vector via this energy function:

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z}\ e^{-E(\mathbf{v},\mathbf{h})} \tag{2}$$

where the "partition function", Z, is given by summing over all possible pairs of visible and hidden vectors:

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{3}$$

The probability that the network assigns to a visible vector, $\mathbf{v}$, is given by summing over all possible hidden vectors:

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{4}$$

The probability that the network assigns to a training image can be raised by adjusting the weights and biases to lower the energy of that image and to raise the energy of other images, especially those that have low energies and therefore make a big contribution to the partition function. The derivative of the log probability of a training vector with respect to a weight is surprisingly simple.

$$\frac{\partial \log p(\mathbf{v})}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{5}$$

where the angle brackets are used to denote expectations under the distribution specified by the subscript that follows. This leads to a very simple learning rule for performing stochastic steepest ascent in the log probability of the training data:

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{6}$$

where ϵ is a learning rate.

Because there are no direct connections between hidden units in an RBM, it is very easy to get an unbiased sample of $\langle v_i h_j \rangle_{data}$. Given a randomly selected training image, $\mathbf{v}$, the binary state, h_j, of each hidden unit, j, is set to 1 with probability

$$p(h_j = 1 \mid \mathbf{v}) = \sigma(b_j + \sum_i v_i w_{ij}) \tag{7}$$

where $\sigma(x)$ is the logistic sigmoid function $1/(1 + \exp(-x))$. $v_i h_j$ is then an unbiased sample.

Because there are no direct connections between visible units in an RBM, it is also very easy to get an unbiased sample of the state of a visible unit, *given a hidden vector*

$$p(v_i = 1 \mid \mathbf{h}) = \sigma(a_i + \sum_j h_j w_{ij}) \tag{8}$$

Getting an unbiased sample of $\langle v_i h_j \rangle_{model}$, however, is much more difficult. It can be done by starting at any random state of the visible units and performing alternating Gibbs sampling for a very long time. One iteration of alternating Gibbs sampling consists of updating all of the hidden units in parallel using equation 7 followed by updating all of the visible units in parallel using equation 8.

A much faster learning procedure was proposed in Hinton (2002). This starts by setting the states of the visible units to a training vector. Then the binary states of the hidden units are all computed in parallel using equation 7. Once binary states have been chosen for the hidden units,

a "reconstruction" is produced by setting each v_i to 1 with a probability given by equation 8. The change in a weight is then given by

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon}) \tag{9}$$

A simplified version of the same learning rule that uses the states of indivisdual units instead of pairwise products is used for the biases.

The learning works well even though it is only crudely approximating the gradient of the log probability of the training data (Hinton, 2002). The learning rule is much more closely approximating the gradient of another objective function called the Contrastive Divergence (Hinton, 2002) which is the difference between two Kullback-Liebler divergences, but it ignores one tricky term in this objective function so it is not even following that gradient. Indeed, Sutskever and Tieleman have shown that it is not following the gradient of any function (Sutskever and Tieleman, 2010). Nevertheless, it works well enough to achieve success in many significant applications.

RBMs typically learn better models if more steps of alternating Gibbs sampling are used before collecting the statistics for the second term in the learning rule, which will be called the negative statistics. CD_n will be used to denote learning using n full steps of alternating Gibbs sampling.

3 How to collect statistics when using Contrastive Divergence

To begin with, we shall assume that all of the visible and hidden units are binary. Other types of units will be discussed in sections 13. We shall also assume that the purpose of the learning is to create a good generative model of the set of training vectors. When using RBMs to learn Deep Belief Nets (see the article on Deep Belief Networks at www.scholarpedia.org) that will subsequently be fine-tuned using backpropagation, the generative model is not the ultimate objective and it may be possible to save time by underfitting it, but we will ignore that here.

3.1 Updating the hidden states

Assuming that the hidden units are binary and that you are using CD_1, the hidden units should have stochastic binary states when they are being driven by a data-vector. The probability of turning on a hidden unit, j, is computed by applying the logistic function $\sigma(x) = 1/(1 + \exp(-x))$ to its "total input":

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij}) \tag{10}$$

and the hidden unit turns on if this probability is greater than a random number uniformly distributed between 0 and 1.

It is very important to make these hidden states binary, rather than using the probabilities themselves. If the probabilities are used, each hidden unit can communicate a real-value to the visible units during the reconstruction. This seriously violates the information bottleneck created by the fact that a hidden unit can convey at most one bit (on average). This information bottleneck acts as a strong regularizer.

For the last update of the hidden units, it is silly to use stochastic binary states because nothing depends on which state is chosen. So use the probability itself to avoid unnecessary sampling noise. When using CD_n, only the final update of the hidden units should use the probability.

3.2 Updating the visible states

Assuming that the visible units are binary, the correct way to update the visible states when generating a reconstruction is to stochastically pick a 1 or 0 with a probability determined by the total top-down input:

$$p_i = p(v_i = 1) = \sigma(a_i + \sum_j h_j w_{ij}) \tag{11}$$

However, it is common to use the probability, p_i, instead of sampling a binary value. This is not nearly as problematic as using probabilities for the data-driven hidden states and it reduces sampling noise thus allowing faster learning. There is some evidence that it leads to slightly worse density models (Tijmen Tieleman, personal communication, 2008). This probably does not matter when using an RBM to pretrain a layer of hidden features for use in a deep belief net.

3.3 Collecting the statistics needed for learning

Assuming that the visible units are using real-valued probabilities instead of stochastic binary values, there are two sensible ways to collect the positive statistics for the connection between visible unit i and hidden unit j:

$$\langle p_i h_j \rangle_{\text{data}} \quad \text{or} \quad \langle p_i p_j \rangle_{\text{data}}$$

where p_j is a probability and h_j is a binary state that takes value 1 with probability p_j. Using h_j is closer to the mathematical model of an RBM, but using p_j usually has less sampling noise which allows slightly faster learning[2].

3.4 A recipe for getting the learning signal for CD_1

When the hidden units are being driven by data, *always* use stochastic binary states. When they are being driven by reconstructions, always use probabilities without sampling.

Assuming the visible units use the logistic function, use real-valued probabilities for both the data and the reconstructions[3].

When collecting the pairwise statistics for learning weights or the individual statistics for learning biases, use the probabilities, not the binary states, and make sure the weights have random initial values to break symmetry.

4 The size of a mini-batch

It is possible to update the weights after estimating the gradient on a single training case, but it is often more efficient to divide the training set into small "mini-batches" of 10 to 100 cases[4]. This allows matrix-matrix multiplies to be used which is very advantageous on GPU boards or in Matlab.

[2]Using h_j always creates more noise in the positive statistics than using p_j but it can actually create less noise in the *difference* of the positive and negative statistics because the negative statistics depend on the binary decision for the state of j that is used for creating the reconstruction. The probability of j when driven by the reconstruction is highly correlated with the binary decision that was made for j when it was driven by the data.

[3]So there is nothing random about the generation of the reconstructions given the binary states of the hidden units.

[4]The word "batch" is confusing and will be avoided because when it is used to contrast with "on-line" it usually means the entire training set.

To avoid having to change the learning rate when the size of a mini-batch is changed, it is helpful to divide the total gradient computed on a mini-batch by the size of the mini-batch, so when talking about learning rates we will assume that they multiply the average, per-case gradient computed on a mini-batch, not the total gradient for the mini-batch.

It is a serious mistake to make the mini-batches too large when using stochastic gradient descent. Increasing the mini-batch size by a factor of N leads to a more reliable gradient estimate but it does not increase the maximum stable learning rate by a factor of N, so the net effect is that the weight updates are smaller *per gradient evaluation*[5].

4.1 A recipe for dividing the training set into mini-batches

For datasets that contain a small number of equiprobable classes, the ideal mini-batch size is often equal to the number of classes and each mini-batch should contain one example of each class to reduce the sampling error when estimating the gradient for the whole training set from a single mini-batch. For other datasets, first randomize the order of the training examples then use minibatches of size about 10.

5 Monitoring the progress of learning

It is easy to compute the squared error between the data and the reconstructions, so this quantity is often printed out during learning. The reconstruction error on the entire training set should fall rapidly and consistently at the start of learning and then more slowly. Due to the noise in the gradient estimates, the reconstruction error on the individual mini-batches will fluctuate gently after the initial rapid descent. It may also oscillate gently with a period of a few mini-batches when using high momentum (see section 9).

Although it is convenient, the reconstruction error is actually a very poor measure of the progress of learning. It is not the function that CD_n learning is approximately optimizing, especially for $n >> 1$, and it systematically confounds two different quantities that are changing during the learning. The first is the difference between the empirical distribution of the training data and the equilibrium distribution of the RBM. The second is the mixing rate of the alternating Gibbs Markov chain. If the mixing rate is very low, the reconstruction error will be very small even when the distributions of the data and the model are very different. As the weights increase the mixing rate falls, so decreases in reconstruction error do not necessarily mean that the model is improving and, conversely, small increases do not necessarily mean the model is getting worse. Large increases, however, are a bad sign except when they are temporary and caused by changes in the learning rate, momentum, weight-cost or sparsity meta-parameters.

5.1 A recipe for using the reconstruction error

Use it but don't trust it. If you really want to know what is going on during the learning, use multiple histograms and graphic displays as described in section 15. Also consider using Annealed Importance

[5]The easy way to parallelize the learning on a cluster is to divide each mini-batch into sub-mini-batches and to use different cores to compute the gradients on each sub-mini-batch. The gradients computed by different cores must then be combined. To minimize the ratio of communication to computation, it is tempting to make the sub-mini-batches large. This usually makes the learning much less efficient, thus wiping out much of the gain achieved by using multiple cores (Vinod Nair, personal communication, 2007).

Sampling (Salakhutdinov and Murray, 2008) to estimate the density on held out data. If you are learning a joint density model of labelled data (see section 16), consider monitoring the discriminative performance on the training data and on a held out validation set.

6 Monitoring the overfitting

When learning a generative model, the obvious quantity to monitor is the probability that the current model assigns to a datapoint. When this probability starts to decrease for held out validation data, it is time to stop learning. Unfortunately, for large RBMs, it is very difficult to compute this probability because it requires knowledge of the partition function. Nevertheless, it is possible to directly monitor the overfitting by comparing the free energies of training data and held out validation data. In this comparison, the partition function cancels out. The free energy of a data vector can be computed in a time that is linear in the number of hidden units (see section 16.1). If the model is not overfitting at all, the average free energy should be about the same on training and validation data. As the model starts to overfit the average free energy of the validation data will rise relative to the average free energy of the training data and this gap represents the amount of overfitting[6].

6.1 A recipe for monitoring the overfitting

After every few epochs, compute the average free energy of a representative subset of the training data and compare it with the average free energy of a validation set. Always use the same subset of the training data. If the gap starts growing, the model is overfitting, though the probability of the training data may be growing even faster than the gap, so the probability of the validation data may still be improving. Make sure that the same weights are used when computing the two averages that you wish to compare.

7 The learning rate

If the learning rate is much too large, the reconstruction error usually increases dramatically and the weights may explode.

If the learning rate is reduced while the network is learning normally, the reconstruction error will usually fall significiantly. This is not necessarily a good thing. It is due, in part, to the smaller noise level in the stochastic weight updates and it is generally accompanied by slower learning in the long term. Towards the end of learning, however, it typically pays to decrease the learning rate. Averaging the weights across several updates is an alternative way to remove some of the noise from the final weights.

7.1 A recipe for setting the learning rates for weights and biases

A good rule of thumb for setting the learning rate (Max Welling, personal communication, 2002) is to look at a histogram of the weight updates and a histogram of the weights. The updates should be about 10^{-3} times the weights (to within about an order of magnitude). When a unit has a very

[6]The average free energies often change by large amounts during learning and this means very little because the log partition function also changes by large amounts. It is only *differences* in free energies that are easy to interpret without knowing the partition function.

large fan-in, the updates should be smaller since many small changes in the same direction can easily reverse the sign of the gradient. Conversely, for biases, the updates can be bigger.

8 The initial values of the weights and biases

The weights are typically initialized to small random values chosen from a zero-mean Gaussian with a standard deviation of about 0.01. Using larger random values can speed the initial learning, but it may lead to a slightly worse final model. Care should be taken to ensure that the initial weight values do not allow typical visible vectors to drive the hidden unit probabilities very close to 1 or 0 as this significantly slows the learning. If the statistics used for learning are stochastic, the initial weights can all be zero since the noise in the statistics will make the hidden units become different from one another even if they all have identical connectivities.

It is usually helpful to initialize the bias of visible unit i to $\log[p_i/(1-p_i)]$ where p_i is the proportion of training vectors in which unit i is on. If this is not done, the early stage of learning will use the hidden units to make i turn on with a probability of approximately p_i.

When using a sparsity target probability of t (see section 11), it makes sense to initialize the hidden biases to be $\log[t/(1-t)]$. Otherwise, initial hidden biases of 0 are usually fine. It is also possible to start the hidden units with quite large negative biases of about -4 as a crude way of encouraging sparsity.

8.1 A recipe for setting the initial values of the weights and biases

Use small random values for the weights chosen from a zero-mean Gaussian with a standard deviation of 0.01. Set the hidden biases to 0. Set the visible biases to $\log[p_i/(1-p_i)]$ where p_i is the proportion of training vectors in which unit i is on. Look at the activities of the hidden units occasionally to check that they are not always on or off.

9 Momentum

Momentum is a simple method for increasing the speed of learning when the objective function contains long, narrow and fairly straight ravines with a gentle but consistent gradient along the floor of the ravine and much steeper gradients up the sides of the ravine. The momentum method simulates a heavy ball rolling down a surface. The ball builds up velocity along the floor of the ravine, but not across the ravine because the opposing gradients on opposite sides of the ravine cancel each other out over time. Instead of using the estimated gradient times the learning rate to increment the *values* of the parameters, the momentum method uses this quantity to increment the *velocity*, $\mathbf{v}$, of the parameters and the current velocity is then used as the parameter increment.

The velocity of the ball is assumed to decay with time and the "momentum" meta-parmeter, α is the fraction of the previous velocity that remains after computing the gradient on a new mini-batch:

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \epsilon \frac{dE}{d\theta_i}(t) \tag{12}$$

If the gradient remains constant, the terminal velocity will exceed $\epsilon dE/d\theta_i$ by a factor of $1/(1-\alpha)$. This is a factor of 10 for a momentum of 0.9 which is a typical setting of this meta-parameter. The

temporal smoothing in the momentum method avoids the divergent oscillations across the ravine that would be caused by simply increasing the learning rate by a factor of $1/(1-\alpha)$.

The momentum method causes the parameters to move in a direction that is not the direction of steepest descent, so it bears some resemblance to methods like conjugate gradient, but the way it uses the previous gradients is much simpler. Unlike methods that use different learning rates for each parameter, momentum works just as well when the ravines are not aligned with the parameter axes.

An alternative way of viewing the momentum method (Tijmen Tieleman, personal communication, 2008) is as follows: It is equivalent to increasing the learning rate by a factor of $1/(1-\alpha)$ but delaying the full effect of each gradient estimate by dividing the full increment into a series of exponentially decaying installments. This gives the system time to respond to the early installments by moving to a region of parameter space that has opposing gradients before it feels the full effect of the increment. This, in turn, allows the learning rate to be larger without causing unstable oscillations.

At the start of learning, the random initial parameter values may create very large gradients and the system is unlikely to be in the floor of a ravine, so it is usually best to start with a low momentum of 0.5 for a number of parameter updates. This very conservative momentum typically makes the learning more stable than no momentum at all by damping oscillations across ravines (Hinton, 1978).

9.1 A recipe for using momentum

Start with a momentum of 0.5. Once the large initial progress in the reduction of the reconstruction error has settled down to gentle progress, increase the momentum to 0.9. This shock may cause a transient increase in the reconstruction error. If this causes a more lasting instability, keep reducing the learning rate by factors of 2 until the instability disappears.

10 Weight-decay

Weight-decay works by adding an extra term to the normal gradient. The extra term is the derivative of a function that penalizes large weights. The simplest penalty function, called "L2", is half of the sum of the squared weights times a coefficient which will be called the weight-cost.

It is important to multiply the derivative of the penalty term by the learning rate. Otherwise, changes in the learning rate change the function that is being optimized rather than just changing the optimization procedure.

There are four different reasons for using weight-decay in an RBM. The first is to improve generalization to new data by reducing overfitting to the training data[7]. The second is to make the receptive fields of the hidden units smoother and more interpretable by shrinking useless weights. The third is to "unstick" hidden units that have developed very large weights early in the training and are either always firmly on or always firmly off. A better way to allow such units to become useful again is to use a "sparsity" target as described in section 11.

The fourth reason is to improve the mixing rate of the alternating Gibbs Markov chain. With small weights, the Markov chain mixes more rapidly[8]. The CD learning procedure is based on ignoring

[7]Since the penalty is applied on every mini-batch, Bayesians really ought to divide the weight-cost by the size of the training set. They can then interpret weight-decay as the effect of a Gaussian weight prior whose variance is independent of the size of the training set. This division is typically not done. Instead, larger weight-costs are used for smaller training sets.

[8]With all zero weights, it reaches its rather boring stationary distribution in a single full step.

derivatives that come from later steps in the Markov chain (Hinton, Osindero and Teh, 2006), so it tends to approximate maximum likelihood learning better when the mixing is fast. The ignored derivatives are then small for the following reason: When a Markov chain is very close to its stationary distribution, the best parameters for modeling samples from the chain are very close to its current parameters.

A different form of weight-decay called "L1" is to use the derivative of the sum of the absolute values of the weights. This often causes many of the weights to become exactly zero whilst allowing a few of the weights to grow quite large. This can make it easier to interpret the weights. When learning features for images, for example, L1 weight-decay often leads to strongly localized receptive fields.

An alternative way to control the size of the weights is to impose a maximum allowed value on the sum of the squares or absolute values of the incoming weights for each unit. After each weight update, the weights are rescaled if they exceed this maximum value. This helps to avoid hidden units getting stuck with extremely small weights, but a sparsity target is probably a better way to avoid this problem.

10.1 A recipe for using weight-decay

For an RBM, sensible values for the weight-cost coefficient for L2 weight-decay typically range from 0.01 to 0.00001. Weight-cost is typically not applied to the hidden and visible biases because there are far fewer of these so they are less likely to cause overfitting. Also, the biases sometimes need to be quite large.

Try an initial weight-cost of 0.0001. If you are using Annealed Importance Sampling (Salakhutdinov and Murray, 2008) to estimate the density on a held-out validation set, try adjusting the weight-cost by factors of 2 to optimize density. Small differences in weight-cost are unlikely to cause big differences in performance. If you are training a joint density model that allows you to test discriminative performance on a validation set this can be used in place of the density for optimizing the weight-cost. However, in either case, remember that weight-decay does more than just preventing overfitting. It also increases the mixing rate which makes CD learning a better approximation to maximum likelihood. So even if overfitting is not a problem because the supply of training data is infinite, weight-decay can still be helpful.

11 Encouraging sparse hidden activities

Hidden units that are only rarely active are typically easier to interpret than those that are active about half of the time. Also, discriminative performance is sometimes improved by using features that are only rarely active (Nair and Hinton, 2009).

Sparse activities of the binary hidden units can be achieved by specifying a "sparsity target" which is the desired probability of being active, $p << 1$. An additional penalty term is then used to encourage the actual probability of being active, q, to be close to p. q is estimated by using an exponentially decaying average of the mean probability that a unit is active in each mini-batch:

$$q_{new} = \lambda q_{old} + (1 - \lambda) q_{current} \tag{13}$$

where $q_{current}$ is the mean activation probability of the hidden unit on the current mini-batch.

The natural penalty measure to use is the cross entropy between the desired and actual distributions:

$$\text{Sparsity penalty} \propto -p \log q - (1-p) \log(1-q) \tag{14}$$

For logistic units this has a simple derivative of $q - p$ with respect to the total input to a unit. This derivative, scaled by a meta-parameter called "sparsity-cost", is used to adjust both the bias and the incoming weights of each hidden unit. It is important to apply the same derivative to both. If the derivative is only applied to the bias, for example, the bias will typically keep becoming more negative to ensure the hidden unit is rarely on, but the weights will keep becoming more positive to make the unit more useful.

11.1 A recipe for sparsity

Set the sparsity target to between 0.01 and 0.1[9]. Set the decay-rate, λ, of the estimated value of q to be between 0.9 and 0.99. Histogram the mean activities of the hidden units and set the sparsity-cost so that the hidden units have mean probabilities in the vicinity of the target. If the probabilities are tightly clustered around the target value, reduce the sparsity-cost so that it interferes less with the main objective of the learning.

12 The number of hidden units

Intuitions derived from discriminative machine learning are a bad guide for determining a sensible number of hidden units. In discriminative learning, the amount of constraint that a training case imposes on the parameters is equal to the number of bits that it takes to specify the label. Labels usually contain very few bits of information, so using more parameters than training cases will typically cause severe overfitting. When learning generative models of high-dimensional data, however, it is the number of bits that it takes to specify a data vector that determines how much constraint each training case imposes on the parameters of the model. This can be several orders of magnitude greater than number of bits required to specify a label. So it may be quite reasonable to fit a million parameters to 10,000 training images if each image contains 1,000 pixels. This would allow 1000 globally connected hidden units. If the hidden units are locally connected or if they use weight-sharing, many more can be used.

12.1 A recipe for choosing the number of hidden units

Assuming that the main issue is overfitting rather than the amount of computation at training or test time, estimate how many bits it would take to describe each data-vector if you were using a good model (*i.e.* estimate the typical negative $\log_2$ probability of a datavector under a good model). Then multiply that estimate by the number of training cases and use a number of parameters that is about an order of magnitude smaller. If you are using a sparsity target that is very small, you may be able to use more hidden units. If the training cases are highly redundant, as they typically will be for very big training sets, you need to use fewer parameters.

[9] If you are only using the sparsity target to revive hidden units that are never active and suppress hidden units that are always active, a target value of 0.5 makes sense (even though it makes nonsense of the name).

13 Different types of unit

RBM's were developed using binary visible and hidden units, but many other types of unit can also be used. A general treatment for units in the exponential family is given in (Welling et al., 2005). The main use of other types of unit is for dealing with data that is not well-modeled by binary (or logistic) visible units.

13.1 Softmax and multinomial units

For a binary unit, the probability of turning on is given by the logistic sigmoid function of its total input, x.

$$p = \sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x + e^0} \tag{15}$$

The energy contributed by the unit is $-x$ if it is on and 0 if it is off. Equation 15 makes it clear that the probability of each of the two possible states is proportional to the negative exponential of its energy. This can be generalized to K alternative states.

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \tag{16}$$

This is often called a "softmax" unit. It is the appropriate way to deal with a quantity that has K alternative values which are not ordered in any way. A softmax can be viewed as a set of binary units whose states are mutually constrained so that exactly one of the K states has value 1 and the rest have value 0. When viewed in this way, the learning rule for the binary units in a softmax is identical to the rule for standard binary units. The only difference is in the way the probabilities of the states are computed and the samples are taken.

A further generalization of the softmax unit is to sample N times (with replacement) from the probability distribution instead of just sampling once. The K different states can then have integer values bigger than 1, but the values must add to N. This is called a multinomial unit and, again, the learning rule is unchanged.

13.2 Gaussian visible units

For data such as patches of natural images or the Mel-Cepstrum coefficients used to represent speech, logistic units are a very poor representation. One solution is to replace the binary visible units by linear units with independent Gaussian noise. The energy function then becomes:

$$E(\mathbf{v},\mathbf{h}) = \sum_{i\in\text{vis}} \frac{(v_i - a_i)^2}{2\sigma_i^2} \quad - \sum_{j\in\text{hid}} b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{17}$$

where σ_i is the standard deviation of the Gaussian noise for visible unit i.

It is possible to learn the variance of the noise for each visible unit but this is difficult using CD_1. In many applications, it is much easier to first normalise each component of the data to have zero mean and unit variance and then to use noise free reconstructions, with the variance in equation 17 set to 1. The reconstructed value of a Gaussian visible unit is then equal to its top-down input from the binary hidden units plus its bias.

The learning rate needs to be about one or two orders of magnitude smaller than when using binary visible units and some of the failures reported in the literature are probably due to using a

learning rate that is much too big. A smaller learning rate is required because there is no upper bound to the size of a component in the reconstruction and if one component becomes very large, the weights emanating from it will get a very big learning signal. With binary hidden and visible units, the learning signal for each training case must lie between -1 and 1, so binary-binary nets are much more stable.

13.3 Gaussian visible and hidden units

If both the visible and the hidden units are Gaussian, the instability problems become much worse. The individual activities are held close to their means by quadratic "containment" terms with coefficients determined by the standard deviations of the assumed noise levels:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i \in \text{vis}} \frac{(v_i - a_i)^2}{2\sigma_i^2} \quad + \sum_{j \in \text{hid}} \frac{(h_j - b_j)^2}{2\sigma_j^2} - \sum_{i,j} \frac{v_i}{\sigma_i} \frac{h_j}{\sigma_j} w_{ij} \tag{18}$$

If any of the eigenvalues of the weight matrix become sufficiently large, the quadratic interaction terms can dominate the containment terms and there is then no lower bound to the energy that can be achieved by scaling up the activities in the direction of the corresponding eigenvector. With a sufficiently small learning rate, CD_1 can detect and correct these directions so it is possible to learn an undirected version of a factor analysis model (Marks and Movellan, 2001) using all Gaussian units, but this is harder than using EM (Ghahramani and Hinton, 1996) to learn a directed model.

13.4 Binomial units

A simple way to get a unit with noisy integer values in the range 0 to N is to make N separate copies of a binary unit and give them all the same weights and bias (Teh and Hinton, 2001). Since all copies receive the same total input, they all have the same probability, p, of turning on and this only has to be computed once. The expected number that are on is Np and the variance in this number is $Np(1-p)$. For small p, this acts like a Poisson unit, but as p approaches 1 the variance becomes small again which may not be desireable. Also, for small values of p the growth in p is exponential in the total input. This makes learning much less stable than for the rectified linear units described in section 13.5.

One nice thing about using weight-sharing to synthesize a new type of unit out of binary units is that the mathematics underlying binary-binary RBM's remains unchanged.

13.5 Rectified linear units

A small modification to binomial units makes them far more interesting as models of real neurons and also more useful for practical applications. All copies still have the same learned weight vector $\mathbf{w}$ and the same learned bias, b, but each copy has a different, fixed offset to the bias. If the offsets are $-0.5, -1.5, -2.5, ... -(N-0.5)$ the sum of the probabilities of the copies is extremely close to having a closed form:

$$\sum_{i=1}^{\infty} \sigma(x - i + 0.5) \approx \log(1 + e^x) \tag{19}$$

where $x = \mathbf{v}\mathbf{w}^T + b$. So the total activity of all of the copies behaves like a smoothed version of a rectified linear unit that saturates for sufficiently large input. Even though $\log(1 + e^x)$ is not in the exponential family, we can model it accurately using a set of binary units with shared weights and

fixed bias offsets. This set has no more parameters than an ordinary binary unit, but it provides a much more expressive variable. The variance is $\sigma(x)$ so units that are firmly off do not create noise and the noise does not become large when x is large.

A drawback of giving each copy a bias that differs by a fixed offset is that the logistic function needs to be used many times to get the probabilities required for sampling an integer value correctly. It is possible, however, to use a fast approximation in which the sampled value of the rectified linear unit is not constrained to be an integer. Instead it is approximated by $\max(0, x + N(0, 1)$ where $N(0, 1)$ is Gaussian noise with zero mean and unit variance. This type of rectified linear unit seems to work fine for either visible units or hidden units when training with CD_1 (Nair and Hinton, 2010).

If both visible and hidden units are rectified linear, a much smaller learning rate may be needed to avoid unstable dynamics in the activity or weight updates. If the weight between two rectified linear units is greater than 1 there is no lower bound to the energy that can be achieved by giving both units very high activities so there is no proper probability distribution. Nevertheless, contrastive divergence learning may still work provided the learning rate is low enough to give the learning time to detect and correct directions in which the Markov chain would blow up if allowed to run for many iterations. RBM's composed of rectified linear units are more stable than RBM's composed of Gaussian units because the rectification prevents biphasic oscillations of the weight dynamics in which units alternate between vary high positive activity for one mini-batch followed by very high negative activity for the next mini-batch.

14 Varieties of contrastive divergence

Although CD_1 is not a very good approximation to maximum likelihood learning, this does not seem to matter when an RBM is being learned in order to provide hidden features for training a higher-level RBM. CD_1 ensures that the hidden features retain most of the information in the data vector and it is not necessarily a good idea to use a form of CD that is a closer approximation to maximum likelihood but is worse at retaining the information in the data vector. If, however, the aim is to learn an RBM that is a good density or joint-density model, CD_1 is far from optimal.

At the beginning of learning, the weights are small and mixing is fast so CD_1 provides a good approximation to maximum likelihood. As the weights grow, the mixing gets worse and it makes sense to gradually increase the n in CD_n (Carreira-Perpignan and Hinton, 2005; Salakhutdinov et al., 2007). When n is increased, the difference of pairwise statistics that is used for learning will increase so it may be necessary to reduce the learning rate.

A more radical departure from CD_1 is called "persistent contrastive divergence" (Tieleman, 2008). Instead of initializing each alternating Gibbs Markov chain at a datavector, which is the essence of CD learning, we keep track of the states of a number of persistent chains or "fantasy particles". Each persisitent chain has its hidden and visible states updated one (or a few) times after each weight update. The learning signal is then the difference between the pairwise statistics measured on a mini-batch of data and the pairwise statistics measured on the persistent chains. Typically the number of persistent chains is the same as the size of a mini-batch, but there is no good reason for this. The persistent chains mix surprisingly fast because the weight-updates repel each chain from its current state by raising the energy of that state (Tieleman and Hinton, 2009).

When using persistent CD, the learning rate typically needs to be quite a lot smaller and the early phase of the learning is much slower in reducing the reconstruction error. In the early phase of learning the persistent chains often have very correlated states, but this goes away with time. The final reconstruction error is also typically larger than with CD_1 because persistent CD is, asymptotically,

performing maximum likelihood learning rather than trying to make the distribution of the one-step reconstructions resemble the distribution of the data. Persistent CD learns significantly better models than CD_1 or even CD_{10} (Tieleman, 2008) and is the recommended method if the aim is to build the best density model of the data.

Persistent CD can be improved by adding to the standard parameters an overlay of "fast weights" which learn very rapidly but also decay very rapidly (Tieleman and Hinton, 2009). These fast weights improve the mixing of the persistent chains. However, the use of fast weights introduces yet more meta-parameters and will not be discussed further here.

15 Displaying what is happening during learning

There are many ways in which learning can go wrong and most of the common problems are easy to diagnose with the right graphical displays. The three types of display described below give much more insight into what is happening than simply monitoring the reconstruction error.

Histograms of the weights, the visible biases and the hidden biases are very useful. In addition, it is useful to examine histograms of the increments to these parameters when they are updated, though it is wasteful to make these histograms after every update.

For domains in which the visible units have spatial or temporal structure (*e.g.* images or speech) it is very helpful to display, for each hidden unit, the weights connecting that hidden unit to the visible units. These "receptive" fields are a good way of visualizing what features the hidden units have learned. When displaying the receptive fields of many hidden units it can be very misleading to use different scales for different hidden units. Gray-scale displays of receptive fields are usually less pretty but much more informative than false colour displays.

For a single minibatch, it is very useful to see a two-dimensional, gray-scale display with a range of [0,1] that shows the probability of each binary hidden unit on each training case in a mini-batch[10]. This immediately allows you to see if some hidden units are never used or if some training cases activate an unusually large or small number of hidden units. It also shows how certain the hidden units are. When learning is working properly, this display should look thoroughly random without any obvious vertical or horizontal lines. Histograms can be used instead of this display, but it takes quite a few histograms to convey the same information.

16 Using RBM's for discrimination

There are three obvious ways of using RBMs for discrimination. The first is to use the hidden features learned by the RBM as the inputs for some standard discriminative method. This will not be discussed further here, though it is probably the most important way of using RBM's, especially when many layers of hidden features are learned unsupervised before starting on the discriminative training.

The second method is to train a separate RBM on each class. After training, the free energy of a test vector, $\mathbf{t}$, is computed (see subsection 16.1)for each class-specific RBM. The log probability that the RBM trained on class c assigns to the test vector is given by:

$$\log p(\mathbf{t}|c) = -F_c(\mathbf{t}) - \log Z_c \tag{20}$$

[10]If there are more than a few hundred hidden units, just use a subset of them.

where Z_c is the partition function of that RBM. Since each class-specific RBM will have a different, unknown partition function, the free energies cannot be used directly for discrimination. However, if the number of classes is small it is easy to deal with the unknown log partition functions by simply training a "softmax" model (on a separate training set) to predict the class from the free energies of all of the class specific RBMs:

$$\log p(class = c|\mathbf{t}) = \frac{e^{-F_c(\mathbf{t}) - \log \hat{Z}_c}}{\sum_d e^{-F_d(\mathbf{t}) - \log \hat{Z}_d}} \tag{21}$$

where the $\hat{Z}$ are parameters that are learned by maximum likleihood training of the softmax. Of course, equation 21 can also be used to learn the weights and biases of each RBM but this requires a lot of data to avoid overfitting. Combining the discriminative gradients for the weights and biases that come from equation 21 with the approximate gradients that come from contrastive divergence will often do better than either method alone. The approximate gradient produced by contrastive divergence acts as a strong regularizer to prevent overfitting and the discriminative gradient ensures that there is some pressure to use the weights and biases in a way that helps discrimination.

The third method is to train a joint density model using a single RBM that has two sets of visible units. In addition to the units that represent a data vector, there is a "softmax" label unit that represents the class. After training, each possible label is tried in turn with a test vector and the one that gives lowest free energy is chosen as the most likely class. The partition function is not a problem here, since it is the same for all classes. Again, it is possible to combine discriminiative and generative training of the joint RBM by using discriminative gradients that are the derivatives of the log probability of the correct class (Hinton, 2007):

$$\log p(class = c|\mathbf{t}) = \frac{e^{-F_c(\mathbf{t})}}{\sum_d e^{-F_d(\mathbf{t})}} \tag{22}$$

16.1 Computing the free energy of a visible vector

The free energy of visible vector $\mathbf{v}$ is the energy that a single configuration would need to have in order to have the same probability as all of the configurations that contain $\mathbf{v}$:

$$e^{-F(\mathbf{v})} = \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{23}$$

It is also given by the expected energy minus the entropy:

$$F(\mathbf{v}) = -\sum_i v_i a_i - \sum_j p_j x_j + \sum_j \left(p_j \log p_j + (1 - p_j) \log(1 - p_j)\right) \tag{24}$$

where $x_j = b_j + \sum_i v_i w_{ij}$ is the total input to hidden unit j and $p_j = \sigma(x_j)$ is the probability that $h_j = 1$ given $\mathbf{v}$. A good way to compute F($\mathbf{v}$) is to use yet another expression for the free energy:

$$F(\mathbf{v}) = -\sum_i v_i a_i - \sum_j \log(1 + e^{x_j}) \tag{25}$$

17 Dealing with missing values

In a directed belief net it is very easy to deal with missing values for visible variables. When performing inference, a missing value that is at the receiving end of a directed connection has no effect on the

units that send connections to it. This is not true for the undirected connections used in an RBM. To perform inference in the standard way, the missing value must first be filled in and there are at least two ways to do this.

A particularly simple type of missing value occurs when learning a joint density for data in which each training case is composed of a vector $\mathbf{v}$ such as an image plus a single discrete label. If the label is missing from a subset of the cases, it can be Gibbs sampled from its exact conditional distribution. This is done by computing the free energy (see section 16.1) for each possible value of the label and then picking label l with probability proportional to $\exp(-F(l, \mathbf{v})$. After this, the training case is treated just like a complete training case.

For real-valued visible units, there is a different way to impute missing values that still works well even if several values are missing from the same training case (Hinton et al., 2006b). If the learning cycles through the training set many times, the missing values can be treated in just the same way as the other parameters. Starting with a sensible initial guess, a missing value is updated each time the weights are updated, but possibly using a different learning rate. The update for the missing value for visible unit i on training case c is:

$$\Delta v_i^c = \epsilon \left(\frac{\partial F}{\partial \hat{v}_i^c} - \frac{\partial F}{\partial v_i^c} \right) \tag{26}$$

where v_i^c is the imputed value and $\hat{v}_i^c$ is the reconstruction of the imputed value. Momentum can be used for imputed values in just the same way as it is used for the usual parameters.

There is a more radical way of dealing with missing values that can be used when the number of missing values is very large. This occurs, for example, with user preference data where most users do not express their preference for most objects (Salakhutdinov et al., 2007). Instead of trying to impute the missing values, we pretend they do not exist by using RBM's with different numbers of visible units for different training cases. The different RBMs form a family of different models with shared weights. Each RBM in the family can now do correct inference for its hidden states, but the tying of the weights means that they may not be ideal for any particular RBM. Adding a visible unit for a missing value and then performing correct inference that integrates out this missing value does not give the same distribution for the hidden units as simply ommitting the visible unit which is why this is a family of models rather than just one model. When using a family of models to deal with missing values, it can be very helpful to scale the hidden biases by the number of visible units in the RBM (Salakhutdinov and Hinton, 2009).

Acknowledgements

This research was supported by NSERC and the Canadian Institute for Advanced Research. Many of my past and present graduate students and postdocs have made valuable contributions to the body of practical knowledge described in this technical report. I have tried to acknowledge particularly valuable contributions in the report, but I cannot always recall who suggested what.

References

Carreira-Perpignan, M. A. and Hinton, G. E. (2005). On contrastive divergence learning. In *Artificial Intelligence and Statistics, 2005*.

Freund, Y. and Haussler, D. (1992). Unsupervised learning of distributions on binary vectors using two layer networks. In *Advances in Neural Information Processing Systems 4*, pages 912–919, San Mateo, CA. Morgan Kaufmann.

Ghahramani, Z. and Hinton, G. (1996). The EM algorithm for mixtures of factor analyzers. Technical Report CRG-TR-96-1, University of Toronto.

Hinton, G. E. (1978). Relaxation and its role in vision. In *PhD Thesis*.

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1711–1800.

Hinton, G. E. (2007). To recognize shapes, first learn to generate images. *Computational Neuroscience: Theoretical Insights into Brain Function.*

Hinton, G. E., Osindero, S., and Teh, Y. W. (2006a). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Hinton, G. E., Osindero, S., Welling, M., and Teh, Y. (2006b). Unsupervised discovery of non-linear structure using contrastive backpropagation. *Cognitive Science*, 30:725–731.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558.

Marks, T. K. and Movellan, J. R. (2001). Diffusion networks, product of experts, and factor analysis. In *Proc. Int. Conf. on Independent Component Analysis*, pages 481–485.

Mohamed, A. R., Dahl, G., and Hinton, G. E. (2009). Deep belief networks for phone recognition. In *NIPS 22 workshop on deep learning for speech recognition*.

Mohamed, A. R. and Hinton, G. E. (2010). Phone recognition using restricted boltzmann machines. In *ICASSP-2010*.

Nair, V. and Hinton, G. E. (2009). 3-d object recognition with deep belief nets. In *Advances in Neural Information Processing Systems*, volume 22, pages 1339–1347.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proc. 27th International Conference on Machine Learning*.

Salakhutdinov, R. R. and Hinton, G. E. (2009). Replicated softmax: An undirected topic model. In *Advances in Neural Information Processing Systems*, volume 22.

Salakhutdinov, R. R., Mnih, A., and Hinton, G. E. (2007). Restricted Boltzmann machines for collaborative filtering. In Ghahramani, Z., editor, *Proceedings of the International Conference on Machine Learning*, volume 24, pages 791–798. ACM.

Salakhutdinov, R. R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In *Proceedings of the International Conference on Machine Learning*, volume 25, pages 872 – 879.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing*, volume 1, chapter 6, pages 194–281. MIT Press, Cambridge.

Sutskever, I. and Tieleman (2010). On the convergence properties of contrastive divergence. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Sardinia, Italy.

Taylor, G., Hinton, G. E., and Roweis, S. T. (2006). Modeling human motion using binary latent variables. In *Advances in Neural Information Processing Systems*. MIT Press.

Teh, Y. and Hinton, G. E. (2001). Rate-coded restricted Boltzmann machines for face recognition. In *Advances in Neural Information Processing Systems*, volume 13, pages 908–914.

Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2008)*. ACM.

Tieleman, T. and Hinton, G. (2009). Using fast weights to improve persistent contrastive divergence. In *Proceedings of the 26th international conference on Machine learning*, pages 1033–1040. ACM New York, NY, USA.

Welling, M., Rosen-Zvi, M., and Hinton, G. E. (2005). Exponential family harmoniums with an application to information retrieval. In *Advances in Neural Information Processing Systems*, pages 1481–1488, Cambridge, MA. MIT Press.