Unsupervised Discovery of Non-Linear Structure using Contrastive Backpropagation

G. E. Hinton, M. Welling, Y. W. Teh
Department of Computer Science
University of Toronto
Toronto, Canada M5S 3G4
and
S. Osindero
Gatsby Computational Neuroscience Unit
University College, London
London, UK WC1N 3AR

June 3, 2003

Abstract

We describe a way of modelling high-dimensional data-vectors by using an unsupervised, non-linear, multilayer neural network in which the activity of each neuron-like unit makes an additive contribution to a global energy score that indicates how surprised the network is by the data-vector. Units whose activities represent violations of learned constraints contribute positively to the global energy and units whose activities represent the presence of familiar features contribute negatively. The connection weights which determine how the activity of each unit depends on the activities in earlier layers are learned by

minimizing the energy assigned to data-vectors that are actually observed and maximizing the energy assigned to "confabulations". The confabulations are generated by perturbing an observed data-vector in a direction that decreases its energy under the current model, so confabulations are things that the model prefers to real data. Backpropagation of energy derivatives through the multilayer network is used both for computing the derivatives that are needed to adjust the weights and for computing how to perturb an observed data-vector to produce a confabulation that has lower energy.

The backpropagation algorithm (ref) trains the units in the intermediate layers of a feedforward neural net to represent features of the input vector that are useful for predicting the desired output. This is achieved by propagating information about the discrepancy between the actual output and the desired output backwards through the net to compute how to change the connection weights in a direction that reduces the discrepancy. In this paper we show how to use backpropagation to learn features and constraints when each input vector is not accompanied by a supervision signal that specifies the desired output.

When no desired output is specified, it is not immediately obvious what the goal of learning should be. We assume here that the aim is to characterize the observed data in terms of many different features and constraints that can be interpreted as hidden factors. These hidden factors could be used for subsequent decision making or they could be used to detect highly improbable data-vectors by using the global energy. We define the probabilty that the network assigns to a data-vector, $\mathbf{x}$, in terms of its global energy, $E(\mathbf{x})$:

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\sum_{\mathbf{v}} e^{-E(\mathbf{v})}} \tag{1}$$

The quality of the set of features and constraints discovered by the neural network can then be quantified by the summed log probability that gets assigned to the observed data-vectors. The contribution of a single data-vector to this sum is:

$$\log p(\mathbf{x}) = -E(\mathbf{x}) - \log \sum_{\mathbf{v}} e^{-E(\mathbf{v})} \tag{2}$$

The features and constraints can be improved by repeatedly adjusting the weights on the connections so as to maximize the log probability of the observed data. To perform gradient ascent in the log likelihood we would need to compute exact derivatives of the form:

$$\frac{\partial \log p(\mathbf{x})}{\partial w_{ij}} = -\frac{\partial E(\mathbf{x})}{\partial w_{ij}} + \sum_{\mathbf{v}} p(\mathbf{v}) \frac{\partial E(\mathbf{v})}{\partial w_{ij}} \tag{3}$$

where w_{ij} is the weight on the connection from unit i in one layer to unit j in the next layer.

The first term is easy to compute. We assume that each unit, j, sums the weighted activities coming from units, i, in the layer below to get its total input, $z_j = \sum_i y_i w_{ij}$, where an activity y_i in the layer below is equal to x_i if it is the input layer. A smooth non-linear function of z_j is then used to compute the unit's activity, y_j (see figure **??**). The energy contributed by the unit can be any smooth function of its activity.

After performing a forward pass through the network to compute the activities of all the units, we do a backward pass as described in **??**. The backward pass uses the chain rule to compute $\partial E(\mathbf{x})/\partial w_{ij}$ for every connection weight, and by backpropagating all the way to the inputs we can also compute $\partial E(\mathbf{x})/\partial x_i$ for each component, x_i, of the input vector.

Unfortunately, the second term in Eq. 3 is much harder to deal with. It involves a weighted average of the derivatives from all conceivable data-vectors so it cannot be computed efficiently except in special cases. We usually expect, however, that the probability mass will be concentrated on a very small fraction of the conceivable data-vectors, so it seems reasonable to approximate the last term by averaging $\partial E(\mathbf{x})/\partial w_{ij}$ over a relatively small number of samples from the distribution $p(.)$. One way to generate samples from this distribution is to run a Markov chain that simulates a physical process with thermal noise. If we think of the dataspace as forming a horizontal plane and we represent the energy of each data-vector as height, the neural network defines a potential energy surface whose height and gradient are easy to compute. We imagine a particle on this surface that tends to move downhill but is also jittered by additional Gaussian noise. After enough steps, the particle will have lost all information about where it started and if we use small enough steps, its probability of being at any particular point in the dataspace will be given by the Boltzmann distribution in Eq 1. This is a painfully slow way of generating samples and even if the equilibrium distribution is reached, the variance created by sampling may mask the true learning signal.

Rather surprisingly, it is unnecessary to allow the simulated physical process to reach the equilibrium distribution. If we start the process at an observed data-vector and just run it for a few steps, we can generate a "confabulation" that works very well for adjusting the weights (Hinton, 2002). Intuitively, if the Markov chain starts to diverge from the data in a systematic way, we already have evidence that the model is imperfect and that it can be improved (in this local region of the dataspace) by reducing the energy of the initial data-vector and raising the energy of the confabulation. So the learning procedure cycles through the observed data-vectors adjusting each weight by:

$$\Delta w_{ij} = \eta \left(-\frac{\partial E(\mathbf{x})}{\partial w_{ij}} + \frac{\partial E(\hat{\mathbf{x}})}{\partial w_{ij}} \right) \tag{4}$$

where η is a learning rate and $\hat{\mathbf{x}}$ is a confabulation produced by starting at $\mathbf{x}$ and noisily following the gradient of the energy surface for a few steps (HMC note).

To illustrate the learning procedure, we applied it to the task of discovering the non-linear kinematic constraints in a simulated three dimensional "arm" that has five rigid links and six ball-joints. The first ball-joint attaches the arm to the origin, and each data-vector consists of the 15 cartesian coordinates of the remaining five ball-joints. This apparently 15-dimensional data really has only 10 degrees of freedom because of the 5 one-dimensional constraints imposed by the 5 rigid links. These constraints are of the form:

$$(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 - l^2 = 0 \tag{5}$$

where i and $i+1$ index neighboring joints. Because the constraints are highly non-linear, dimensionality-reduction methods like principal components analysis or factor analysis are of little help.

We used a neural net with 15 input units and two hidden layers. Each of the 15 units in the first hidden layer computes a weighted average of the inputs and then squares it. Each of the 5 units in the top layer computes a weighted average of the squares provided by the first hidden layer and adds a learned bias. It is clear that with the right weights and biases, each top-layer unit could implement one of the constraints represented by equation ?? by producing an output of exactly zero if and only if the constraint is satisfied. The question is whether the network can discover the appropriate weights and biases just by observing the data.

For this example, the units in the first hidden layer do not contribute to the energy and the units in the second hidden layer each contribute an energy of $\lambda_j \log(1 + y_j^2)$ where y_j is the activity of unit j in the second hidden layer and λ_j is a scale parameter that is also learned by contrastive backpropagation. This "heavy-tailed" energy function penalizes top-level units with non-zero outputs, but changing the output has little effect on the penalty if the output is already large.

Figure 1 shows the weights and top-level biases that were learned by contrastive backpropagation. For each pair of neighboring joints, there are three units in the first hidden layer that have learned to compute differences between the coordinates of the two joints. These differences are always computed in three orthogonal directions. Each unit in the second hidden layer has learned a linear combination of the 5 constraints, but it uses weights of exactly the same size for the three squared differences in each constraint so that it can exactly cancel the fixed sum of these three squared differences by using its bias.

The contrastive backpropagation learning procedure is quite flexible. It puts no constraints other than smoothness on the activation functions or the functions for converting activations into energy contributions. The procedure can easily be modified to use recurrent neural networks that receive time-varying inputs such as video sequences. The energy of a whole sequence is simply defined to be some function of the time history of the activations of the hidden units. Backpropagation through time (ref) can then be used to obtain the derivatives of the energy *w.r.t.* the connection weights and also the energy gradients required for generating a whole confabulated sequence.

Notes

1. We used a simplified version of the Hybrid Monte Carlo procedure in which the particle is given a random initial momentum and its deterministic trajectory along the energy surface is then simulated for 10 time steps. If this simulation has no numerical errors the increase, ΔE, in the combined potential and kinetic energy will be zero. If ΔE is positive, the particle is returned to its initial position with a probability of $1 - exp(-\Delta E)$. The step size is slowly adapted so that only about 10% of the trajectories get rejected. Numerical errors up to second order are eliminated by using a "leapfrog" method (Neal, 19??) which uses the potential energy gradient at time t to compute the velocity increment between time $t - \frac{1}{2}$ and $t + \frac{1}{2}$ and uses the velocity at time $t + \frac{1}{2}$ to compute the position increment between time t and $t + 1$.

2. If the confabulations are generated by picking a value for one component of the data-vector from its posterior distribution given all the other components, our method is a version of pseudo-likelihood fitting (Besag, Heckerman) in which the use of a global energy function ensures that all of the conditional distributions are consistent.

Acknowledgments

We would like to thank David MacKay, Radford Neal, Sam Roweis, Zoubin Ghahramani, Chris Williams, Carl Rasmussen, Brian Sallans, Javier Movellan and Tim Marks for helpful discussions. This research was supported by the Gatsby Charitable foundation and NSERC.

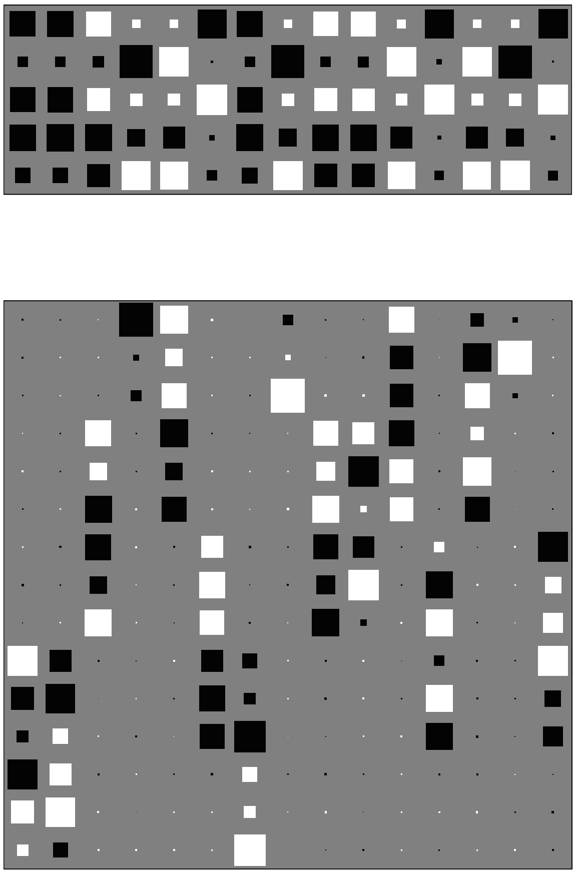

Figure 1: The areas of the small black and weight rectangles represent the magnitudes of the negative and positive weights learned by the network. Each column in the lower block represents the weights on the connections to a unit in the middle layer from the joint coordinates $x_1, y_1, z_1, x_2, y_2, z_2 ... x_5, y_5, z_5$. Each row in the higher block represents the weights on the connections to a top level unit from the middle layer units.