Deep Belief Networks and their application to Music

Introduction

In this project we investigate the new area of machine learning research called deep learning and explore some of its interesting applications. Deep learning has grabbed focus because of its ability to model highly varying functions associated with complex behaviours and human intelligence. Another major reason why they are so impactful is the fact that certain functions when compactly represented by deep architectures might require exponential number of computational elements in case we reduce depth.

Artificial Neural Networks

A classical perceptron is a linear classifier accepting a vector of real valued inputs and giving an output of 1 or -1 based on result being greater or lesser than threshold, or simply the difference from threshold for continuous output. An active neuron implies an input close to 1. A network of perceptron assuming a layered structure with each layer neurons serving as input to next layer is called ***feedforward*** network. Since cascading of such linear units would only result in piecewise linear functions we replace the discontinuous threshold with a sigmoid function. Mathematically the hypothesis class of perceptrons is given by:

$$H = \{h_{w,b} \mid h_{w,b}(x) = f(w^T x + b)\}$$

The linear perceptron is given by $h^L_{w,b}(x) = sign(w^T + b)$. A perceptron with sigmoid activation function is given by $h^S_{w,b}(x) = 1/(1+e^{-(w^T x+b)})$. In a feedforward network the set of nodes without parents form the input layer. The leaf nodes form the output layer. The nodes lying on path from input to output layer occurring in middle are called the hidden layer. For each edge in this network there is an associated weight. Let us represent the weight associated with connection between node *i* in layer *l* to node *j* in layer *l+1* as $W^{(l)}_{ij}$, and $b^{(l)}_i$ as the bias associated with unit *i* in layer *l+1*. Let $a^{(l)}_i$ denote the activation of neuron i in layer l. We can succinctly represent the computation activation values in feedforward phase by $a^{(l+1)} = f(W^{(l)}.a^{(l)}+b)$ where f is the activation function.

Sparse Autoencoder

A neural network which sets the target output values equal to the input values is called the autoencoder. This kind of network is used for unsupervised learning which basically refers to learning large datasets with few or no labels associated with them. A network is called sparse if we constrain the neurons to be inactive most of the time. Mathematically this can be achieved by setting the expected value of activation close to -1 (say -0.9).

We can define a cost function C for any neural network using error from output and regularization term as $C(W,b;x,y) = \frac{1}{2}(\left\|h_{W,b}(x) - y\right\|^2) - \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W^{(l)}_{ji})^2$. In an online learning scenario the gradients of cost function can be used to training weights by using steepest descent. Another strategy could be to minimize the expected value of cost function. The gradient of cost function can be efficiently described using the input, weight, biases, activation values and backpropagated error terms from expected output, and can be used to update weight and biases.

Backpropagation algorithm gives an efficient way to compute partial derivatives required to perform steepest descent. The algorithm involves first computation of all activations by feedforward computation over the neural network. Using the values computed in feed forward phase we train the weights for higher layers and backpropagate the error term to lower layers. For the output layer computed error is given by $s_i \delta_i^l = -(y_i - a_i^n) f'(W^{(l)T} x + b)$. Mathematically the propagated error is then given by $\delta_i^l = (\sum_{j=1}^{s_l} W_{ij}^l \delta_i^{l+1}) f'(W^{(l)T} x + b)$. The partial derivative of cost function and corresponding updates are given following equations.

$$\frac{\partial C}{\partial W_{ij}^l} = a_j^l \delta_i^{l+1} + \lambda W_{ij}^l \qquad \frac{\partial C}{\partial b_i^l} = \delta_i^{l+1}$$

$$W_{ij}^l(k+1) = W_{ij}^l(k) - \alpha(a_j^l \delta_i^{l+1} + \lambda W_{ij}^l(k)) \ , \ b_i^l(k+1) = b_i^l(k) - \alpha \delta_i^{l+1}$$

The sparsity on trained neural network can be enforced by keeping expected sparseness of neural network and reinforcing bias terms based on deviation of actual sparseness from expected sparseness.

Restricted Boltzmann Machines

Boltzmann machines are the super class of deep learning architectures. A Boltzmann machine is a network of interconnected perceptron units with each unit being stochastic function. The probability of firing of each perceptron is given by sigmoid function. The global state of such distribution of perceptrons is governed by Maxwell Boltzmann statistics. We can theoretically define quantities like energy and temperature of such system as given below. It can be shown that the stochastic probability selection confirms with steady state energy distribution of these machines at a given temperature.

$$E = -\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

$$\Delta E_i = E_{i=off} - E_{i=on} = \sum_j w_{ij} s_j - \theta_i = -k_B T \ln(p_{i=off}) + k_B T \ln(p_{i=on})$$

$$\Rightarrow p_{i=on} = \frac{1}{1 + e^{(-\Delta E_i / T)}}$$

In a Maxwell-Boltzmann distribution the probability of a system state is governed by the energy of that state. Artificially inputting values inside the deep network is identically to fixing the temperature of particles of distribution to a certain value. To compute the correct distribution we need to ensure thermal equilibrium between particles. This distribution of machine states over training set is denoted by $P^0(v)$.

In generative mode the deep layer will emulate the Boltzmann distribution at its lowest possible energy. The expected correlation between units can again be computed by running the machine a number of times until it cools down and reaches a global minimum. The real distribution when left to itself can be denoted as $P^{\infty}(v)$. Our training requirement is to bring both these probabilities as close to one another as possible. This process of cooling down after setting temperature is called Simulated Annealing. The level of similarity between the states can be judged by Kullback-Leiber divergence(KL):

$$KL(P^0 \,||\, P_\theta^\infty) = \sum_v P^0(v) \ln\left(\frac{P^0(v)}{P_\theta^\infty(v)}\right)$$

Theoretically Boltzmann machine is extremely powerful computational medium. But this power comes at a cost. The training method of simulated annealing fails when a machine is scaled to anything larger than a trivial machine. However clever learning algorithms can be developed for a simplification of the Boltzmann machine called Restricted Boltzmann machine, which does not allow intralayer connections between hidden units. The structure of RBM makes sampling of steady state distribution easy because all hidden states can be updated in parallel independent of each other. These values of updated hidden states can be used to update the input units. It can be proved that we can use samples from n iterations to perform steepest descent to train this machine. Mathematically

$$\frac{\partial KL}{\partial w_{ij}} = -\frac{1}{R}\left[\, p_{ij}^\infty - p_{ij}^0 \right] = \frac{1}{R}\left(\left\langle v_i^0 h_j^0 \right\rangle - \left\langle v_i^\infty h_j^\infty \right\rangle\right)$$

It can be shown that auto encoder training approximates RBM training by contrastive divergence. The following figure (figure a) shows features learned by a RBM with 10 hidden units on an image databank of sofa images (generated by self). Compare these to edges (figure b) learnt by sparse auto encoder.

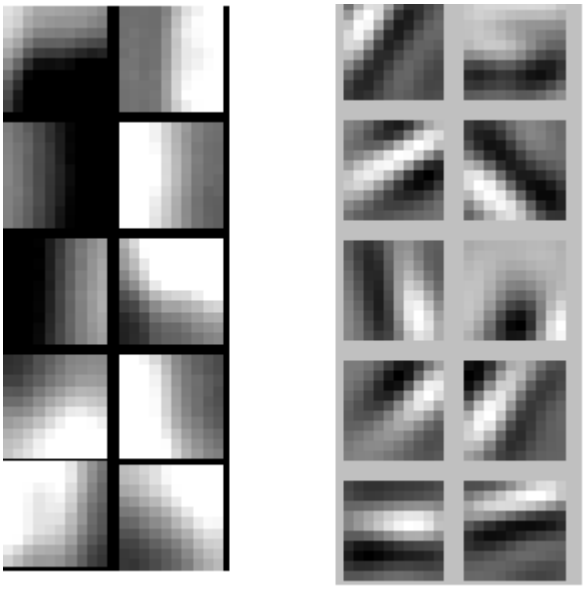

Figure 1.a (left) Weight Vector learnt from RBM Training. Figure 1.b (right) Weight vector learnt from Sparse Autoencoder.

The contrastive divergence technique fails to learn multilayer generative networks because Gibbs sampling cycles might no be sufficient for network to reach equilibrium. However a deep network can be simplified by considering training of one layer at a time. Training deeper layers then involves learning and fixing weights for previous layers to generate higher level data as input for that layer. If the RBM was perfectly learnt we can train the higher layers correctly.

Application of Deep Learning to Midi Music Data

A midi music file can be represented as a NX8 matrix with each column representing note number, velocity, start/end time etc. We can equivalently represent the data pictorially with each pixel assuming note velocity on note*time axis (Figure 2.a). For this experiment the data was sourced from online MIDI databank http://www.diskuspublishing.com/midi.html. We create feature vectors from MIDI music data from by taking patches from time*notes domain. This ensures that correlation between musical notes and their temporal location is not lost. Data generation involves selecting random files and taking random samples of size T*N in time*notes space. For sake of simplicity we keep the samples square i.e. T=N=10. The results from running machine on 50000 such samples are shown below (Figure 2.b). For microsecond level time interval we observe straight lines in features which represent single notes and chords.

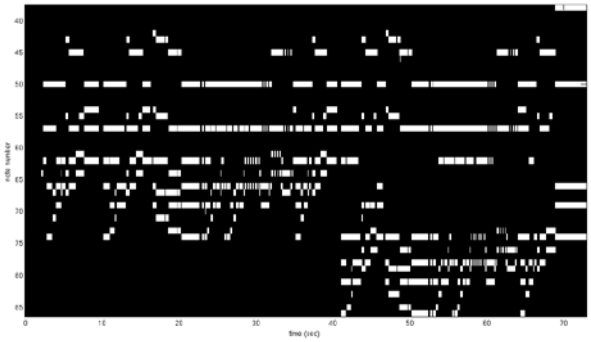

Figure 2.a Pictorial representation of midi data.

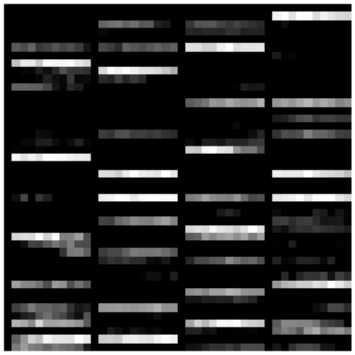

Figure 2.b Weight matrix for each RBM hidden neuron

When smaller windows in time are chosen network ends up learning chords. To learn better representation we rescale data to second and sub-second size intervals. We observe that RBM learns simple monotonic rise and falls as shown in figure 2.c. Note that these features are distinctly different from ones learnt on image data by RBM. The results below help us understand the importance of selecting a good feature vector.

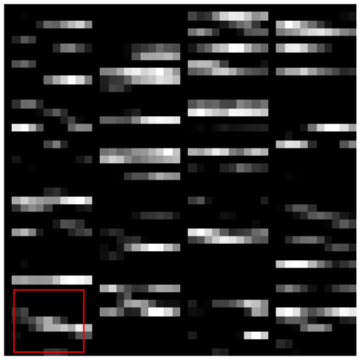

Figure 2.c Rise and Fall of notes learnt using RBM

Bibliography

1. http://www.stanford.edu/class/archive/cs/cs294a/cs294a.1104/
2. http://en.wikipedia.org/wiki/Boltzmann_machine
3. Bengio Y.; “Learning deep architectures for AI”
4. http://www.kenschutte.com/midi
5. http://en.wikipedia.org/wiki/Maxwell-Boltzmann_statistics
6. Hinton, G. E.; Osindero, S.; Teh, Y. "A fast learning algorithm for deep belief nets".
7. Ackley, D. H.; Hinton, G. E.; Sejnowski, T. J. "A Learning Algorithm for Boltzmann Machines"
8. Welling and Hinton.; “A New Learning Algorithm for Mean Field Boltzmann Machines”
9. http://www.cs.toronto.edu/~hinton/
10. Hinton, G. E. and Salakhutdinov, R. R.; Reducing the dimensionality of data with neural networks.
11. Please refer to http://web.mit.edu/~otkrist/www/machine_learning/project.tar for code and results.

Acknowledgement

Gabriel Zaccak for his feedback, help and support.