10-708: Probabilistic Graphical Models 10-708, Spring 2014

26 : Deep Neural Networks and GMs

Lecturer: Matthew Gormley *Scribes: Hayden Luse*

1 Introduction

The goal of deep learning is to solve the problem of coming up with good features. This is valuable as one of the best ways to get state of the art results is good feature engineering. Not only can deep networks themselves produce state of the art results but graphical models can also use the features discovered by these networks. There is currently a huge amount of interest in these techniques and large amounts of corporate money are being funnelled into groups using them including a four hundred million dollar investment by Google in Deep Mind. This interest is based on the success these techniques have had and in many areas the current state of the art was achieved with deep networks that did not require hand tuned features. The basic ideas have been around since the 1960's however there have been marked improvements in recent years. Stochastic Gradient Descent for example was a huge improvement. Beyond that, using more hidden units, new nonlinear functions (ReLu's), better online optimization and simply having more powerful CPUs and GPUs have advanced the field significantly. This lecture focuses on the decision function and loss functions used in deep learning.

2 Background and Notation

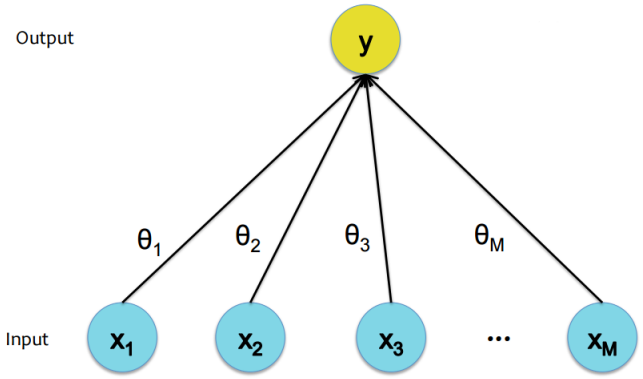
2.1 basic structure

1. Given training data $\{x_i, y_i\}_{i=1}^N$
2. Choose a decision function $\hat{y} = f_\theta(x_i)$
3. Choose a loss function $\ell(y, \hat{y}) \in R)$
4. Define a goal $\theta^* = \min_\theta \sum_{i=1}^N \ell(f_\theta(x_i), y_i)$
5. Train with SGD $\theta^{(t+1)} = \theta^t - \eta_t \nabla \ell(f_\theta(x_i), y_i)$

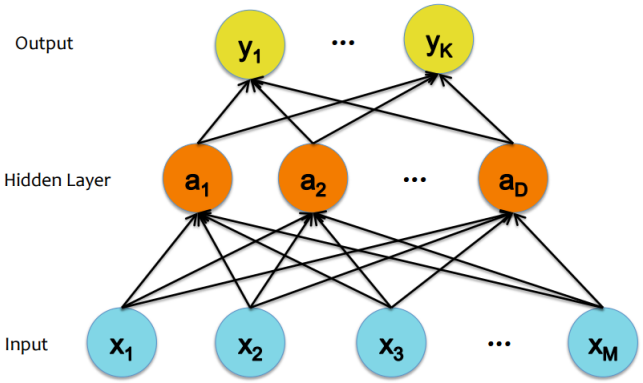
In order to calculate the gradients in SGD, backpropogation can be used which is a special case of a more general algorithm Reverse Mode Automatic Differentiation.

3 Deep Neural Networks (DNNs)

First we consider choosing $f_\theta(x_i) = h(\theta \cdot x_i)$ where $h(a) = a$ for the following network architecture.



In this case, the network is equivalent to linear regression. If instead we choose $h(a) = \frac{1}{1+\exp(a)}$ this network is equivalent to logistic regression. The next step in progressing towards deep networks is to begin adding hidden layers as in the following network.



This network also includes multiple outputs. A single layer network like this is already a universal function approximator and works well. However a network with further hidden layers can have fewer computational units for the same power and be representationally efficient. Deeper networks can generalize non-locally and may allow for hierarchy. Overall deeper networks have been shown to work better. The intuition for why deeper networks work better is that they allow for different levels of abstraction of features. As we do not know going in the "right" level of abstraction we can let the model figure it out.

3.1 Training

3.1.1 No Pre-Training

The first method we can use to train a deep network is to simply use the same strategy as for a shallow net and simply train the network with Backpropogation. Backpropogation is just repeated use of the chain rule. For a network with one hidden layer and a sigmoid activation function for the hidden units backpropogation is:

Forward Pass:

1. Loss function: $J = y^* \log q + (1 - y^*) \log(1 - q)$
2. Output (sigmoid) $y = \frac{1}{1+exp(-b)}$
3. Output (linear) $b = \sum_{j=0}^{D} \beta_j z_j$
4. Hidden (sigmoid) $z_j = \frac{1}{1+exp(-a_j)}$
5. Hidden (linear) $a_j = \sum_{i=0}^{M} a_{ji} x_i$

Backwards Pass:

1. $\frac{dJ}{dq} = \frac{y^*}{q} + \frac{(1-y^*)}{q-1}$
2. $\frac{dJ}{db} = \frac{dJ}{dq}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(b)}{(\exp(b)+1)^2}$
3. $\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$
4. $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
5. $\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(a_j)}{(\exp(a_j)+1)^2}$
6. $\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$
7. $\frac{dJ}{dx_i} = \frac{dJ}{d\alpha_{ji}}\frac{d\alpha_{ji}}{dx_i}, \frac{d\alpha_{ji}}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$

However this algorithm runs into two major problems. First, this is a convex optimization algorithm being used to optimize a non-convex function and as such gets stuck in local optima. Second, as this method has more and more multiplications as more hidden layers are added the gradients being propogated backwards through the network become vanishingly small.

3.1.2 Supervised Pre-Training

The idea of supervised pre-training for the network is to use labeled data and then greedily train the network layer by layer from the bottom up, fixing each layer's parameters after it has been trained. After this step, fine tune the network by training as usual with backpropogation. This approach performs better than with no pre-training but still fails to outperform shallow networks.

3.1.3 Unsupervised Pre-training

This works identically to supervised pre-training except for the output that the network is being trained to predict in each step. For unsupervised pretraining instead of predicting a label, the network predicts the input which is of greater dimension than the internal hidden layers. This setup is called an autoencoder and the goal is to minimize the reconstruction error. The intuition is that if the network can reduce the dimensionality of the input and reconstruct it accurately then it has learned useful features of the data. The pre-training is done with the same approach of greedy layerwise optimization and then fine tuning with backpropogation. This approach performs best and outperforms shallow networks.

In practice, using ReLu units instead of sigmoid helps and regularization helps significantly. Convolutional Neural Nets do not commonly use pretraining.

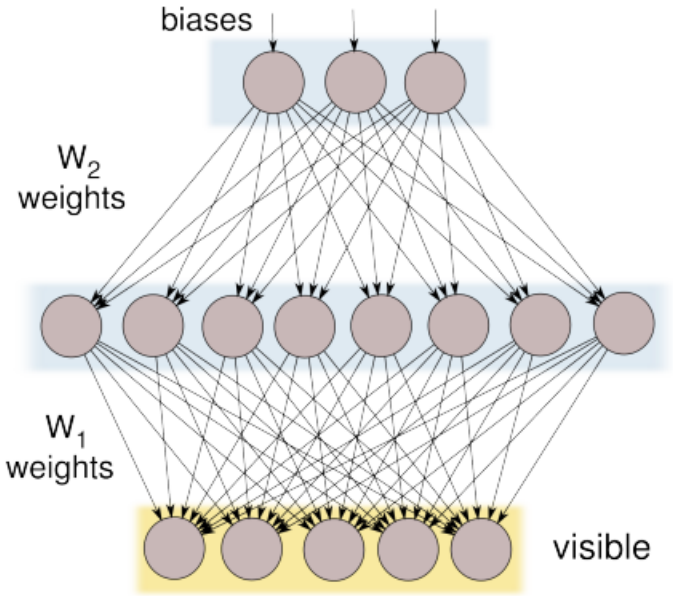
4 Deep Belief Networks

4.1 Background

The connection between graphical models and deep networks goes back to the very first deep learning papers in 2006 which were about innovations in training a particular flavor of Belief Network that also happened to be a neural net. Deep belief networks are generative models with the goal of learning unsupervised representations of probability distributions (such as for images) that can be sampled from.

4.2 Sigmoid Belief Networks

A sigmoid belief network is a directed graphical model (the following diagram is a graphical model not a neural net) of binary variables in fully connected layers where only the bottom layer is observed. The specific parameterization of the conditional probabilities is $p(x_i|\text{parents}(x_i)) = \frac{1}{1+\exp(-\sum_j w_i x_j)}$. The model has the following form.



4.3 Contrastive Divergence Training

Contrastive Divergence is a general tool for learning a generative distribution, where the derivative of the log partition function is intractable to compute.

$$\log L = \log P(D) = \sum_{v \in D} (\log(P^*(v)/Z) \propto \frac{1}{N} \sum_{v \in D} \log P^*(v) - \log Z$$

So

$$\frac{\partial}{\partial w} \log L \propto \frac{1}{N} \sum_{v \in D} \sum_{h} P(h|v) \frac{\partial}{\partial w}^{*}(x) - \sum_{v,h} P(v,h) \frac{\partial}{\partial w} \log P^{*}(x)$$

Contrastive divergence estimates the second term with a Monte Carlo estimate from a 1-step Gibbs sampler. The first term is called the clamped/wake phase and the second is called the unclamped/sleep/free phase. For a belief net the joint is automatically normalized: Z is a constant 1. The second term is zero. For the weight w_{ij} from j into i the gradient $\frac{\log L}{\partial w_{ij}} = (x_i - p_i)x_j$ so stochastic gradient ascent is $\Delta w_{ij} \propto (x_i - p_i)x_j$ which is called the delta rule. This is a stochastic version of the EM algorithm. In the E step we get samples from the posterior and in the M step we apply a learning rule that makes them more likely.

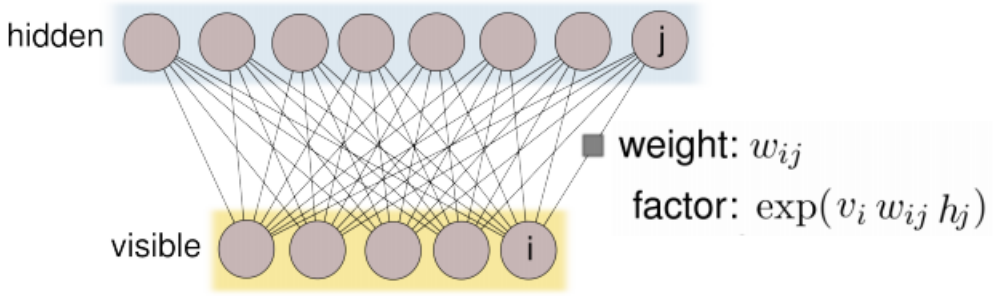
In practice applying Contrastive Divergence to a Deep Sigmoid Belief Net fails. Sampling from the posterior of many hidden layers does not approach equilibrium quickly enough.

4.4 Boltzman Machines

A Boltzman Machine is an undirected graphical model of binary variables with pairwise potentials. The parameterization of the potentials is $\psi_{ij}(x_i, x_j) = \exp(x_i W_{ij} x_j)$ so a higher value of W_{ij} leads to higher correlation between x_i and x_j.

4.5 Restricted Boltzman Machines

A restricted boltzman machine assumes all visible units are one layer and hidden units are another. Units within a layer are unconnected.



Since units within a layer are not connected we can do Gibbs sampling updating a whole layer at a time. So to learn an RBM, the program is to start with a training vector on visible units and then alternate between updating all hidden units in parallel and updating all visible units in parallel.

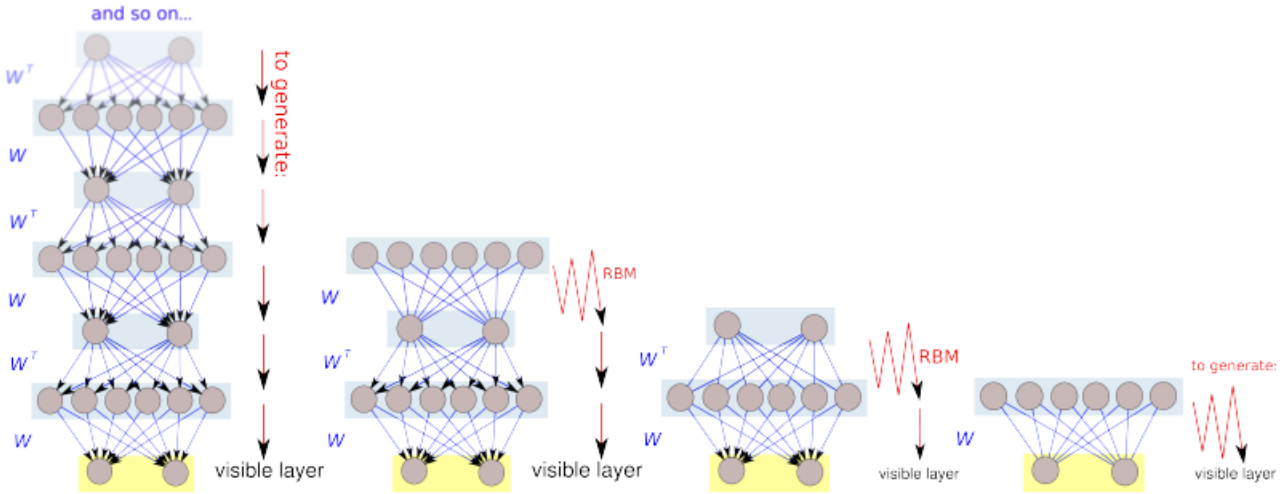
$$\Delta w_{ij} = \eta[< v_i h_j >^0 - < v_i h_j >^\infty]$$

By restricting connectivity we do not have to wait for equilibrium in the clamped phase. We can also curtail the Markov chain during learning. We can use

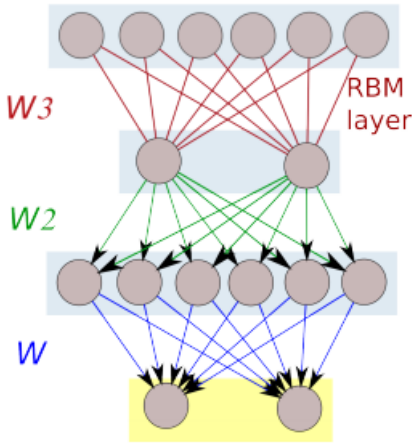
$$\Delta w_{ij} = \eta[< v_i h_j >^0 - < v_i h_j >^1]$$

which is not the correct gradient but works well in practice.

Another way to look at RBM's is that they are equivalent to infinitely deep belief networks. Sampling alternately from the visible units and the hidden units infinitely is the same as sampling from an infinitely deep belief network with the weights between each layer being tied. So training an RBM is actually training an infinitely deep sigmoid belief net.



Our goal then is to untie the weights of the layers of this net. We do this by first freezing the first RBM and then training another RBM on top of it. This unties the weights of layers 2+ in the net. We can then freeze the weights of the second layer to untie layers 3+ and so on. After untying the third layer, the structure is as follows:



Where the third RBM is as before equivalent to an infinitely deep belief network with tied weights. To train this network we can use the same wake sleep structure as before. The wake phase is doing a bottom up pass, starting with a pattern in the training set. Use the delta rule to make this more likely under the generative model. The sleep phase is to do a top-down pass starting from an equilibrium sample from the top RBM. Use the delta rule to make this more likely under the recognition model.