"What optimization problem is deep learning solving?"

We compare unsupervised deep beleief nets (RBMs, DBNs, etc) and supervised deep nets (MLPs, CNNs, RNNs, etc)

Unsupervised DBNs have been described by Bengio (in his new book), as well as by Ng, Larochelle, etc. These nets are specififed using a statistical mechanics formalism, including specifying the Hamiltonian, Free Energy, and Partition funciton. Infernce is achieved by monte carlo sampling techniques of various forms.

Supervised Deep Nets are supported by modern frameworks like Google Tensorflow. They are specified using only an Energy function, and inference uses Stochastic Gradient Descent. Curiosuly, the Deep Net Energy functions do not resemble traditional Hopfield Nets or Ising Models, but, rather, resemble the form of the log conditional probabilities arising in Unsupervised DBMs like RBMs.

This begs the question...do Supervised Deep Nets, training with SGD, effectively (implicitly) run a monte carlo sampling ?

Consider a typical MLP, with 1 output vector (o), 2 hidden laters (h,k), and visbile units (v)

$$\mathbf{o} = \mathbf{W^{ok}k} + \mathbf{a^T k}$$

$$\mathbf{k} = ReLu(\mathbf{W^{kh}h} + \mathbf{b^T h})$$

$$\mathbf{h} = ReLu(\mathbf{W^{hv}v} + \mathbf{c^T v})$$

The outputs are typically converted to probabilities using a softmax transform, yielding a probabilistic prediciton for a class label

$$y_c^{pred} = Softmax(\mathbf{o})_\mathbf{c}$$

and the MLP minimizes the cross entropy between the true and predicted class labels

$$\min \sum_c y_c^{true} ln(y_c^{pred})$$

(although in TensorFlow, the crossentropy and softmax are usually combined into 1 single optimization function)

see https://github.com/aymericdamien/TensorFlow-Examples/blob/master/notebooks/3_NeuralNetworks

Now consider that the activation function, the Rectified Linear Unit (ReLu) is approximations the log sigmoid function

$$ReLu(x) \sim ln(1 + e^x)$$

and is applied pointwise to a vector $\mathbf{x}$

Note: when we take a sum of ReLus, we get a product of sigmoids

$$\sum_j ReLu(x_j) = \sum_j ln(1 + e^{x_j}) = ln(\prod_j (1 + e^{x_j}))$$

finally, WLOG, let the bias terms a, b,c = 0

We can now write

$$\sum_j \mathbf{k}_j = \sum_j Relu((\mathbf{W^{kh}h})_j) = \sum_j ln(1+exp((\mathbf{W^{kh}h})_j) = ln(\prod_j (1+exp((\mathbf{W^{kh}h})))$$

where j sums over the nodes in the Hidden layer k

Let us now assume that each node in the neural network can activated or not, and, therefore, exists in a superposition of on/off states. We therefore assign a state vector $\mathbf{c}_n$ to each node (n)

$$\mathbf{c}_n = [0, 1]$$

This gives something like

$$\mathbf{k} = ln(\prod_j (exp((\mathbf{cW^{kh}h})_j))$$

so we have now a way to get at a free energy for each layer...this is probably not correct but the intent is here...more to come

Because we are running SGD on a training set, we may write o,k,h as a function of x explicitly

$$\mathbf{k(x)} = ln(\prod_j (1 + exp((\mathbf{W^{kh}h(x)})_j))$$

using some of this

The SGD trained nets propagate errors using back-prop. This leads, seemingly, to walking an energy landscape. In DBNs, and in graphical methods generally, we propagate normalized probabilities. Lets take a look at the normalization in an SGD trained Deep Net (SGD-DN) , starting with the final softmax

On every batch $\mathcal{B}$ of training examples $\mathbf{x}$, we compute the normalization, sampled over the batch, along each output node o_c

$$Z(\mathbf{o}, \mathcal{B}) = \sum_c \sum_{\mathbf{x} \in \mathcal{B}} \exp[o_c(\mathbf{x})]$$

where

$$o_c(\mathbf{x}) = ReLu(\mathbf{W}_c^{ok}\mathbf{k(x)})$$

$$o_c(\mathbf{x}) = ln(1 + \exp(\mathbf{W}_c^{ok}\mathbf{k(x)}))$$

$$o_c(\mathbf{x}) = ln(1 + \exp(\sum_{j=1}^{N_k} W_{cj}^{ok} k_j(\mathbf{x})))$$

$$o_c(\mathbf{x}) = ln(1 + \prod_{j=1}^{N_k} \exp(W_{cj}^{ok} k_j(\mathbf{x})))$$

o_c has units of energy, and it would be nice if we could play some 'tricks' to get rid of the 1+product by some reasonable approximation

such as introducing the c=[0,1] vector

that is, if we could get an expression like

$$o_c(\mathbf{x}) \approx ln(\prod_{j=1}^{N_k} \exp(X_{cj} k_j(\mathbf{x})))$$

so that we can write each node as a sum of terms

$$o_c(\mathbf{x}) \approx \sum_{j=1}^{N_k} X_{cj} k_j(\mathbf{x})$$

or some reasonable form so we can continue to expand k and h in some linear way, at least as a first approximation

this may be too crude to work with ...

Free Energy per Layer

I have been trying to get an expression for the free energy per layer

That is, for k, h, and v, we want to define / construct

$$\mathcal{F}^k = -ln \sum_k e^{-\mathcal{H}(\mathbf{k}(\mathbf{x}))}$$

where $\mathcal{H}(\mathbf{k}(\mathbf{x}))$ is the renormalized Hamiltonian / Energy function for the K hidden layer

although in SGD I assume we are also sampling over the x in the Data set / batch, which I dont show here

likewise,

$$\mathcal{F}^h = -ln \sum_h e^{-\mathcal{H}(\mathbf{h}(\mathbf{x}))}$$

and the Free Energy of the visibile units is defined by an energy function $\mathcal{H}(\mathbf{v}(\mathbf{x}))$

$$\mathcal{F}^v = -ln \sum_v e^{-\mathcal{H}(\mathbf{v}(\mathbf{x}))}$$

During training, we want these Free Energies to be similar, so that

$$\Delta\mathcal{F} = \mathcal{F}^k - \mathcal{F}^h = \mathcal{F}^h - \mathcal{F}^v = 0$$

We note that the total partition function for , say , 1 Hidden layer, if we could define it, would be

$$Z(h,v) = \sum_h \sum_v e^{-E(h,v)}$$

as we do in say an RBM
so if we can write

$$E(h,v) = H(v) + V(v,h)$$

giving

$$Z = \sum_v e^{-\mathcal{H}(v)} = \sum_{h,v} e^{-\mathcal{H}(v)} e^{-\mathcal{H}(v,h)}$$

which gives the **intermediate normalization constraint**

$$\sum_h e^{-V(h,v)} = 1$$

which I am hoping can be used as a constraint to show how the SGD training 'effectively' propagates something like a normalized , perhaps conditional probability through the net