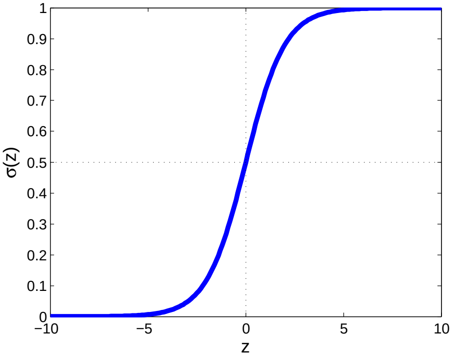
COMP 790-125: Goals for today

- Neural Nets
- (Restricted) Boltzman Machines
- Deep Learning

Sigmoid

Recall sigmoid function

$$\sigma(z) = \frac{1}{1 + \exp\{-z\}}$$



We used this function when we discussed logistic regression

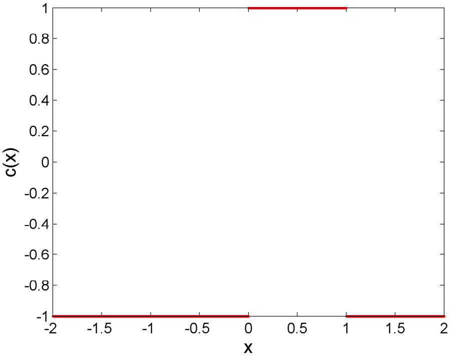
$$p(y = 1|\mathbf{x}) = \sigma(b + \mathbf{v}^T\mathbf{x})$$

Logistic regression is weak

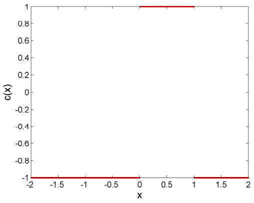
Logistic regression is a classifier with a linear decision boundary ($\mathbf{vx} + b > 0$) and hence a “weak” learner.

For example, assume that we are trying to learn the following concept

$$c(x) = \begin{cases} 1, & 0 \le x \le 1 \\ -1, & \text{otherwise.} \end{cases}$$



Logistic regression is weak



Regardless how many examples of (x, y) pairs we are given logistic regression will never be able to learn this concept.

However, using a derived set of features

$$
\begin{aligned}
x^1(x) &= \begin{cases} 1, & x > 1 \\ 0, & \text{otherwise} \end{cases} \\
x^2(x) &= \begin{cases} 1, & x < 0 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

the problem is trivial even for logistic regression.

Deeper architectures

Logistic regression is *shallow* – it builds separating hyperplane linearly from features.

Ability to induce more complex features requires deeper architecture (decision trees for example).

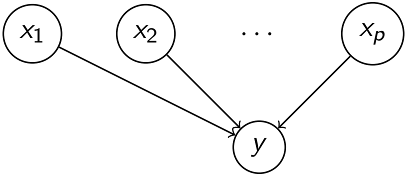
Neural networks and deep belief networks and deep boltzman machines are such alternatives.

Sigmoidal unit

A bit different story than with graphical models.

Nodes/units correspond to function evaluations – neurons responding to their inputs.

If inputs are strong enough, the neuron is activated, and it fires.



$$y = \sigma(b + \mathbf{v}^T \mathbf{x})$$

Sigmoidal unit – removing bias term

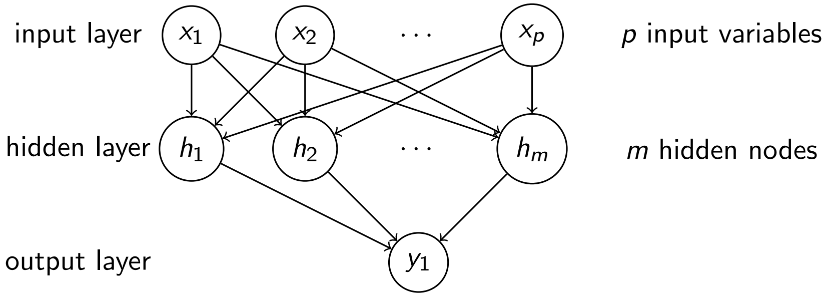
There is no need to treat bias term b separately.

We can simply introduce a node that is always 1 and let it serve as input to all other units.

$$\begin{tikzcd}
x_0 \arrow[drrr] & x_1 \arrow[drr] & x_2 \arrow[dr] & \cdots & x_p \arrow[dl] \\
 & & & y &
\end{tikzcd}$$

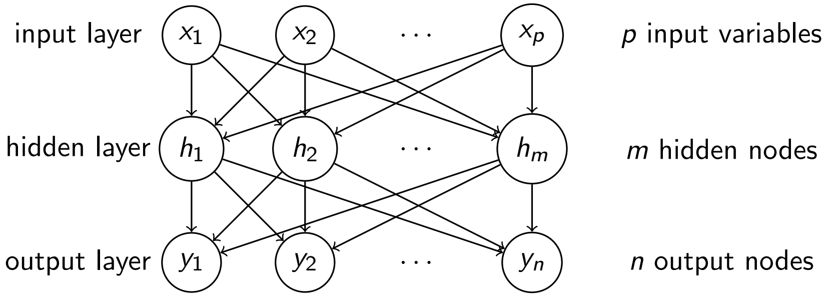
$$y = \sigma(\mathbf{v}^T \mathbf{x})$$

Neural network with a single hidden layer and a single output



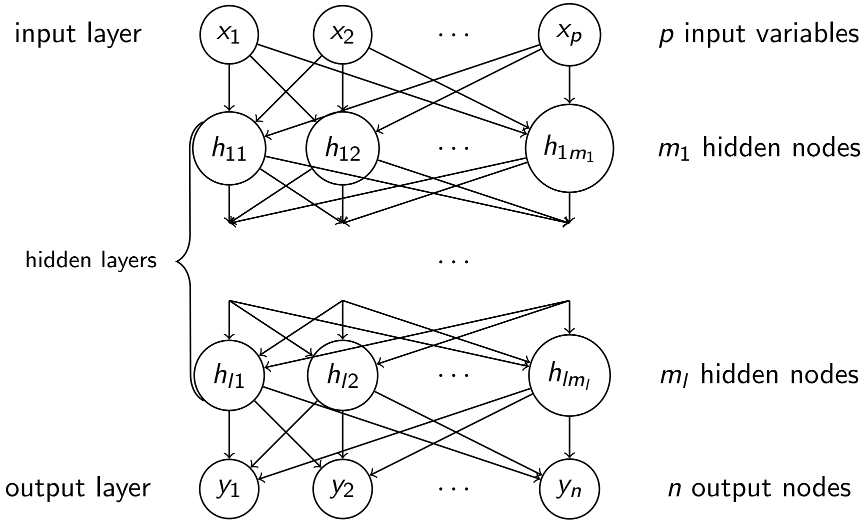
$$\begin{aligned} h_i &= \sigma(\mathbf{v}_{1i}^T \mathbf{x}) \\ y_1 &= \sigma(\mathbf{v}_{21}^T \mathbf{h}) \end{aligned}$$

Neural network with a single hidden layer and multiple outputs



$$
\begin{aligned}
h_i &= \sigma(\mathbf{v}_{1i}^T \mathbf{x}) \\
y_i &= \sigma(\mathbf{v}_{2i}^T \mathbf{h})
\end{aligned}
$$

Deeper architecture



Neural networks – computing a prediction

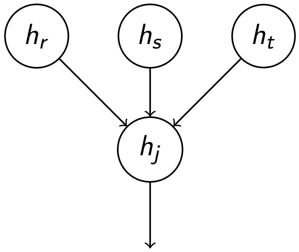
We can define a bit of notation to simplify discussion

- $a_{j,i}$, unit i's input to unit j
- $w_{j,i}$, weight of unit i's input to unit j
- $q_j = \sum_i w_{j,i} a_{j,i}$ net input to unit j
- o_j output of unit j

For input layer $o_j = x_j$, for the rest $o_j = \sigma(q_j)$. For output layers

- continuous y_j then $\hat{y}_j = q_j$
- binary y_j then $\hat{y}_j = \sigma(q_j)$

Neural networks - computing prediction



In this case

$$\begin{aligned} q_j &= \sum_i w_{j,i} a_{j,i} = w_{j,r} o_r + w_{j,s} o_s + w_{j,t} o_t \\ o_j &= \sigma(q_j) = \sigma(w_{j,r} o_r + w_{j,s} o_s + w_{j,t} o_t) \end{aligned}$$

Neural networks – forward propagation

The forward propagation computes output of neural network by iterating through units in topological order and computing:

$$\begin{aligned} q_j &= \sum_i w_{j,i} a_{j,i} \\ o_j &= \sigma(q_j) \end{aligned}$$

and for the output $\hat{y}_j = q_j$ or $\hat{y}_j = \sigma(q_j)$.

Neural networks – error

Given set of training pairs $(\mathbf{x}^t, y^t), t = 1, \dots T$, for each input $\mathbf{x}^t$ forward propagation computes $\hat{y}^t$.

To measure training error

- continuous outputs $E = \sum_t E_t = \sum_t (y_t - \hat{y}_t)^2$
- binary outputs $E = \sum_t E_t = \sum_t y_t \log \hat{y}_t$

A learning procedure finds weights that minimize this error.

Neural network training

The error functions used in neural network training are usually smooth.

By far the most popular way of training a neural network is gradient descent

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla E$$

where η is a reasonably small value, *learning rate*.

Computing gradient of neural network error

Back-propagation is an efficient algorithm for computing gradients of neural network errors.

It is a recursive application of chain rule.

$$\begin{aligned} E &= \sum_t E_t \\ \frac{\partial E}{\partial w_{j,i}} &= \sum_t \frac{\partial E_t}{\partial w_{j,i}} \end{aligned}$$

Since the error is additive across training examples so are the derivatives.

$$\frac{\partial E_t}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} \frac{\partial q_j}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i}$$

Derivatives of output unit's weights

$$\frac{\partial E_t}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i}$$

We will assume squared error $E_t = (\hat{y}_t - y_t)^2$.

If we assume that output of unit k, $o_k = \sigma(q_k)$ then

$$\frac{\partial E_t}{\partial q_k} = \frac{\partial E_t}{\partial o_k} \frac{\partial o_k}{\partial q_k} = \frac{\partial E_t}{\partial o_k} o_k(1 - o_k)$$

Under assumption of squared error

$$\frac{\partial E_t}{\partial o_k} = \frac{\partial}{\partial o_k}(o_k - y_t)^2 = 2(o_k - y^t)$$

Finally we can compute

$$\frac{\partial E_t}{\partial w_{k,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i} = \frac{\partial E_t}{\partial o_k} o_k(1 - o_k) a_{j,i} = 2(o_k - y^t) o_k(1 - o_k) a_{k,i}$$

Derivatives of output unit's weights

$$\frac{\partial E_t}{\partial w_{k,i}} = 2(o_k - y^t)o_k(1 - o_k)a_{k,i}$$

To compute these derivatives we need results of forward propagation starting with inputs $\mathbf{x}^t$, in particular o_k and $a_{k,i}$.

Derivatives of hidden unit's weights

Situation here is a little more complicated since outputs of hidden layer units serve as inputs to other units.

Let $\delta_j = \frac{\partial E_t}{\partial q_j}$

Assume that for each unit j we have a list of units F_j that take output of unit j as input – units j feeds into. For example,a node j in the last hidden layer feeds into an output node.

$$\begin{aligned}
\delta_j = \frac{\partial E_t}{\partial q_j} &= \sum_{i \in F_j} \frac{\partial E_t}{\partial q_i} \frac{\partial q_i}{\partial q_j} = \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial q_j} \\
&= \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial o_j} \frac{\partial o_j}{\partial q_j} \\
&= \frac{\partial o_j}{\partial q_j} \sum_{i \in F_j} \delta_i w_{i,j} \\
&= o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{i,j}
\end{aligned}$$

Backpropagation

We know how to compute derivative's of output unit's weights

$$\delta_k = \frac{\partial E_t}{\partial q_k} = 2(o_k - y^t)o_k(1 - o_k)$$

Now we can consider the layer l above the output layer. For each node j in that layer, F_j is subset of output nodes.

$$\delta_j = o_j(1 - o_j)\sum_{i \in F_j} \delta_i w_{i,j}$$

Hence we have all the δ's needed for layer l, and we can proceed in a backward fashion to the top layer.

Once all the δ_is are computed we obtain

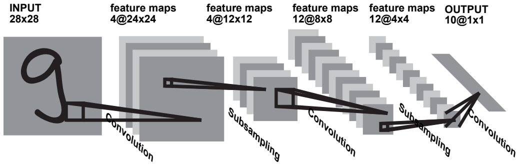
$$\frac{\partial E_t}{\partial w_{j,i}} = \delta_j a_{j,i}$$

Backprop algorithm

```
input  : $\mathbf{w}$, $\mathrm{Train} = \{(\mathbf{x}^t, y^t) : t = 1, \ldots T\}$
output: Error gradient $\frac{\partial E}{\partial \mathbf{w}}$
for $t = 1, 2, \ldots T$ do
    Forward propagate starting with $\mathbf{x}^t$
        Inputs: $o_i = x_i^t$
        Hiddens: $o_j = \sigma\left(\sum_i w_{ji} o_i\right)$
    Propagate deltas backward
        Outputs : $\delta_k = 2(o_k - y^t) o_k (1 - o_k)$
        Hiddens : $\delta_j = o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{i,j}$
    Compute gradients
        $\frac{\partial E_t}{\partial w_{ji}} = \delta_j a_{j,i}$
        $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial w_{ji}} + \frac{\partial E_t}{\partial w_{ji}}$
end
```

Convolutional networks

One of popular architectures, if not the most popular one



Neural network architecture for digit recognition [**?**]

Each plane is further constrained to use exactly the same weights.

Belief prop doesn't always work well

It is slow.

It gets stuck in local minima.

Requires plenty of labeled data.

But also gave some impressive results – digit recognition

Undirected neural network models

A function assigns energy to a configuration of variables $\mathbf{v}$

$$E(\mathbf{v}) = -\frac{1}{2}\mathbf{v}^T\Theta\mathbf{v}$$

This energy function induces a distribution over these variables, Boltzmann distribution

$$p(\mathbf{v}) = \frac{1}{Z}\exp\{-E(\mathbf{v})\}$$

where

$$Z = \sum_{\mathbf{v}}\exp\{-E(\mathbf{v})\}$$

A special type of Markov Random Fields, also referred to as Boltzmann Machines. Note that in a graphical representation if $\Theta_{i,j} = \Theta_{j,i} = 0$ then there is no connection between units i and j.

Learning Boltzmann Machines

Typically Boltzmann Machines observations span all the variables in the model. However, the gradients are still hard to compute.

$$
\begin{aligned}
\nabla_\theta \frac{1}{T} \sum_t \log p(\mathbf{v}^t; \theta) &= \nabla_\theta \frac{1}{T} \sum_t \log \frac{\exp\{-E(\mathbf{v}^t)\}}{\sum_{\mathbf{s}} \exp\{-E(\mathbf{s})\}} \\
&= \left(-\nabla_\theta \frac{1}{T} \sum_t E(\mathbf{v})\right) - \\
&\quad \underbrace{\nabla_\theta \log \sum_{\mathbf{s}} \exp\{-E(\mathbf{s})\}}_{\text{log partition function}}
\end{aligned}
$$

Derivatives of log partition functions

$$-\frac{\partial}{\partial\theta}\log\sum_{\mathbf{z}}\exp\{-E(\mathbf{z})\} = \frac{\sum_{\mathbf{z}} - \exp\{-E(\mathbf{z})\}\frac{\partial}{\partial\theta}E(\mathbf{z})}{\sum_{\mathbf{z}^1}\exp\{-E(\mathbf{z}^1)\}}$$

$$= \sum_{\mathbf{z}}\frac{\exp\{-E(\mathbf{z})\}}{\sum_{\mathbf{z}^1}\exp\{-E(\mathbf{z}^1)\}}\frac{\partial E(\mathbf{z})}{\partial\theta}$$

Hence, this is an expectation. In Boltzmann Machine literature it is common to use physics notation for expectation

$$\sum_{\mathbf{z}}\frac{\exp\{-E(\mathbf{z})\}}{\sum_{\mathbf{z}^1}\exp\{-E(\mathbf{z}^1)\}}\frac{\partial E(\mathbf{z})}{\partial\theta} = \left\langle\frac{\partial E(\mathbf{z})}{\partial\theta}\right\rangle$$

Derivatives of Log-likelihood of Boltzmann Machines

$$\nabla_\theta \log p(\mathbf{v};\theta) = -\left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_0 + \left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_\infty$$

where the first expectation is with respect to empirical distribution in the sample

$$\left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_0 = \sum_t \frac{1}{T} \frac{\partial E(\mathbf{v}^t)}{\partial \theta}$$

and the second expectation is with respect to the model distribution

$$\left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_\infty = \sum_{\mathbf{v}} \frac{\exp\left\{-E(\mathbf{v})\right\}}{\sum_{\mathbf{s}} \exp\left\{-E(\mathbf{s})\right\}} \frac{\partial E(\mathbf{v})}{\partial \theta}$$

Contrastive divergence

Whereas $\left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_0$ is trivial to compute, $\left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_\infty$ is hard. It typically involves running Gibbs sampler for many iterations, hence the ∞ symbol.

The contrastive divergence replaces the exact gradient

$$\nabla_\theta \log p(\mathbf{v};\theta) = \left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_0 + \left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_\infty$$

with an inexact one

$$\nabla_\theta \log p(\mathbf{v};\theta) \approx \left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_0 + \left\langle \frac{\partial E(\mathbf{v})}{\partial \theta} \right\rangle_1$$

where both $\langle\cdot\rangle_1$ is a set of T samples produced by doing a single Gibbs sweep initialized from each observed datapoint.

Contrastive divergence algorithm for computing gradients

```
input  : $\theta$, $\mathrm{Train} = \{\mathbf{v}^t : t = 1, \dots T\}$
output: approximate gradient $\mathbf{g}$
for $t = 1, 2, \dots T$ do
    $\mathbf{g}_0 = \mathbf{g}_0 + \frac{1}{T}\frac{\partial E(\mathbf{v}^t)}{\partial \theta}$
    Initialize Gibbs sampler with $\mathbf{v}^t$ and do a full sweep to obtain $\mathbf{b}$
    $\mathbf{g}_1 = \mathbf{g}_1 + \frac{1}{T}\frac{\partial E(\mathbf{b})}{\partial \theta}$
    $\mathbf{g} = \mathbf{g} - \mathbf{g}_0 + \mathbf{g}_1$
end
```

Why does CD work?

At first glance this should not work out.

After all the gradient is wrong how can we hope to move in the right direction.

The key is that even a single sweep of Gibbs sampler tries to move the configuration to a higher probability under the current wrong parameters.

The parameters that drive this *corruption* away from the true data are those that need to be adjusted.

Restricted Boltzmann Machines

We assumed that the training data is composed of fully observed examples.

Now we will relax this assumption by letting some variables be unobserved.

The observed, input variables, and unobserved, hidden variables, will be connected. The restriction placed on RBMs is that the connections between input and hidden variables.

More formally RBMs energy can be written in the form

$$E(\mathbf{v}, \mathbf{h}) = \mathbf{a}^T \mathbf{v} + \mathbf{b}^T \mathbf{h} + \mathbf{v}^T \Theta \mathbf{h}$$

where $\mathbf{v}$ are observed and $\mathbf{h}$ are unobserved variables.

Conditional probabilities

Conditional probabilities for RBMs are straightforward

$$\begin{aligned} p(\mathbf{h}|\mathbf{v}) &= \prod_j p(h_j|\mathbf{v}) = \sigma(b_j + \sum_i v_i \theta_{i,j}) \\ p(\mathbf{v}|\mathbf{h}) &= \prod_i p(v_j|\mathbf{h}) = \sigma(a_i + \sum_j \theta_{i,j} h_j) \end{aligned}$$

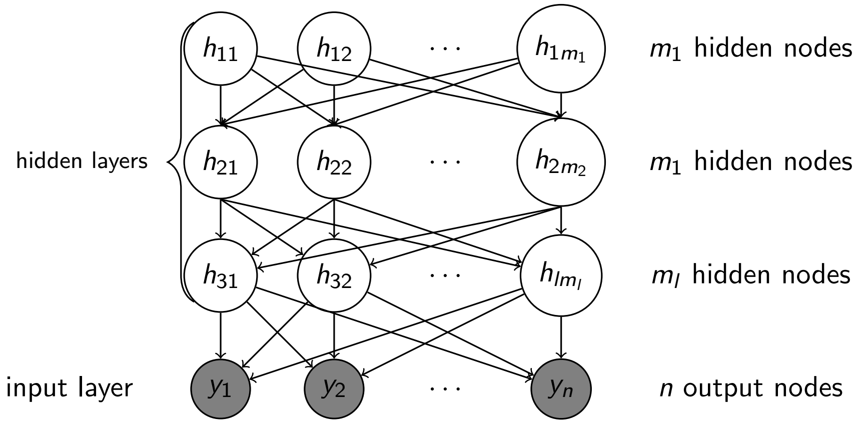
due to the conditional independence between members of the same layer given all of the members of the other layer.

This makes for a straightforward implementation of Gibbs sampler and Contrastive Divergence algorithm.

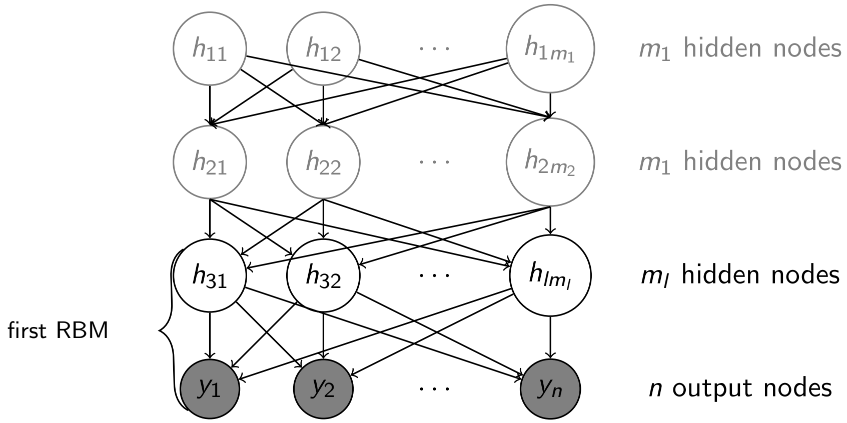
Contrastive Divergence for RBMs

```
input  : $\Theta$, $\mathrm{Train} = \{\mathbf{v}^t : t = 1, \dots T\}$
output: approximate gradient $\mathbf{g}$
for $t = 1, 2, \dots T$ do
    Sample $\mathbf{h}$ from $p(\mathbf{h}|\mathbf{v}^t)$
    $\mathbf{g}_0 = \mathbf{g}_0 + \frac{1}{T}\mathbf{v}^t\mathbf{h}^T$
    Sample $\mathbf{v}$ from $p(\mathbf{v}|\mathbf{h})$
    Sample $\mathbf{h}$ from $p(\mathbf{h}|\mathbf{v})$
    $\mathbf{g}_1 = \mathbf{g}_1 + \frac{1}{T}\mathbf{v}\mathbf{h}^T$
    $\mathbf{g} = \mathbf{g} - \mathbf{g}_0 + \mathbf{g}_1$
end
```

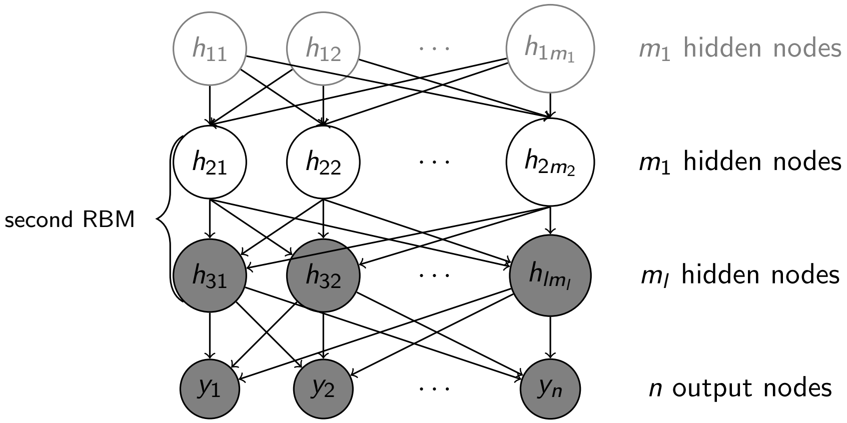
Stacking RBMs



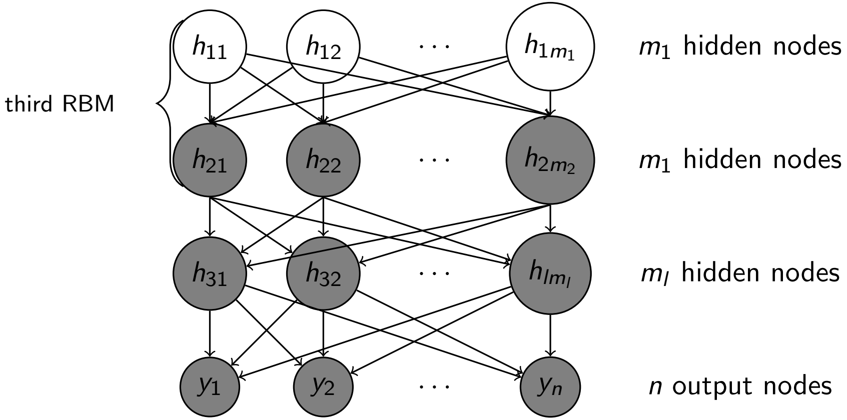
Stacking RBMs



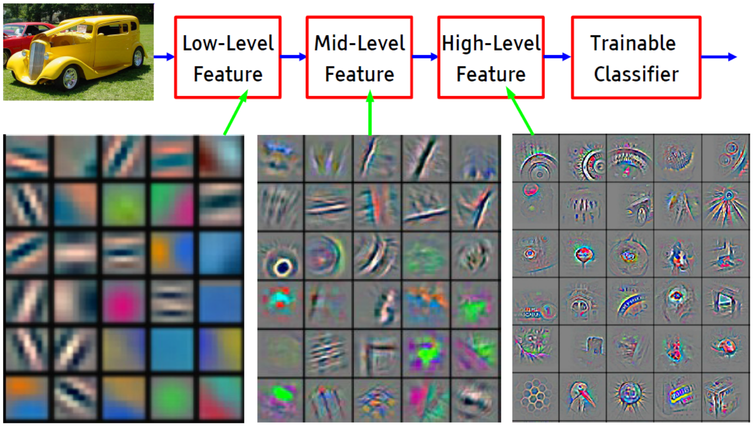
Stacking RBMs



Stacking RBMs



A view of convolutional network layers



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Deep Net training

Progressively train RBMs by considering pairs of layers.

Once an RBM is trained sample (or estimate from variational bound) assignment to the unobserved layer and use it is input layer for the next RBM.

Hints on how to train:
`http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf`

COMP 790-125: Goals for today

- Neural Nets
- (Restricted) Boltzman Machines
- Deep Learning