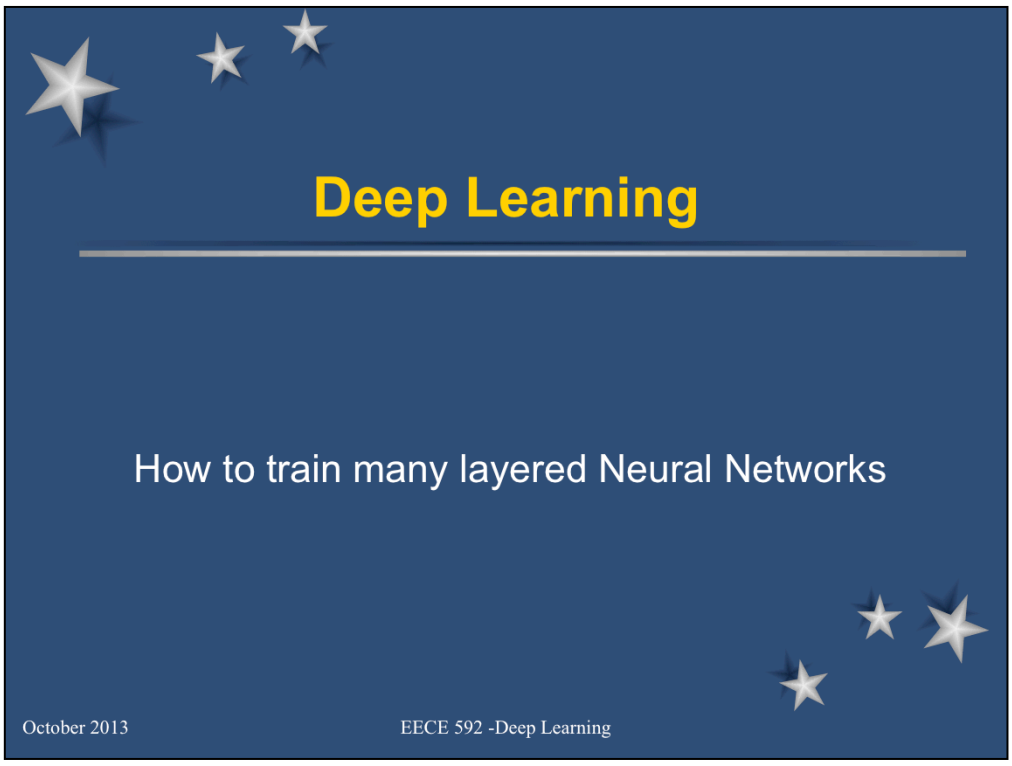
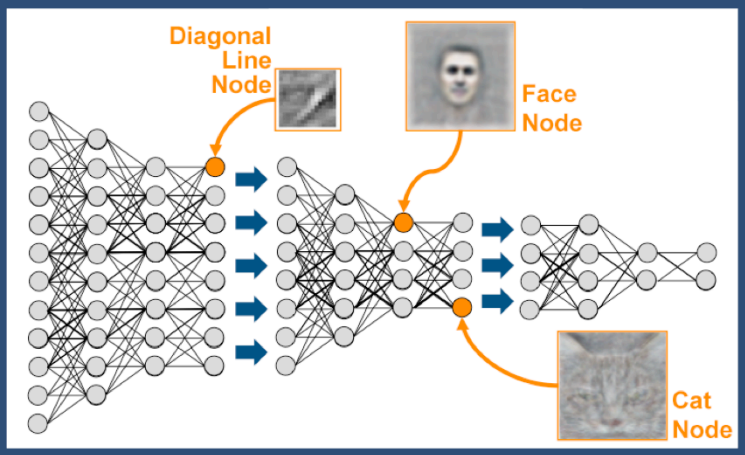

Deep Learning
How to train many layered Neural Networks
October 2013
EECE 592 -Deep Learning

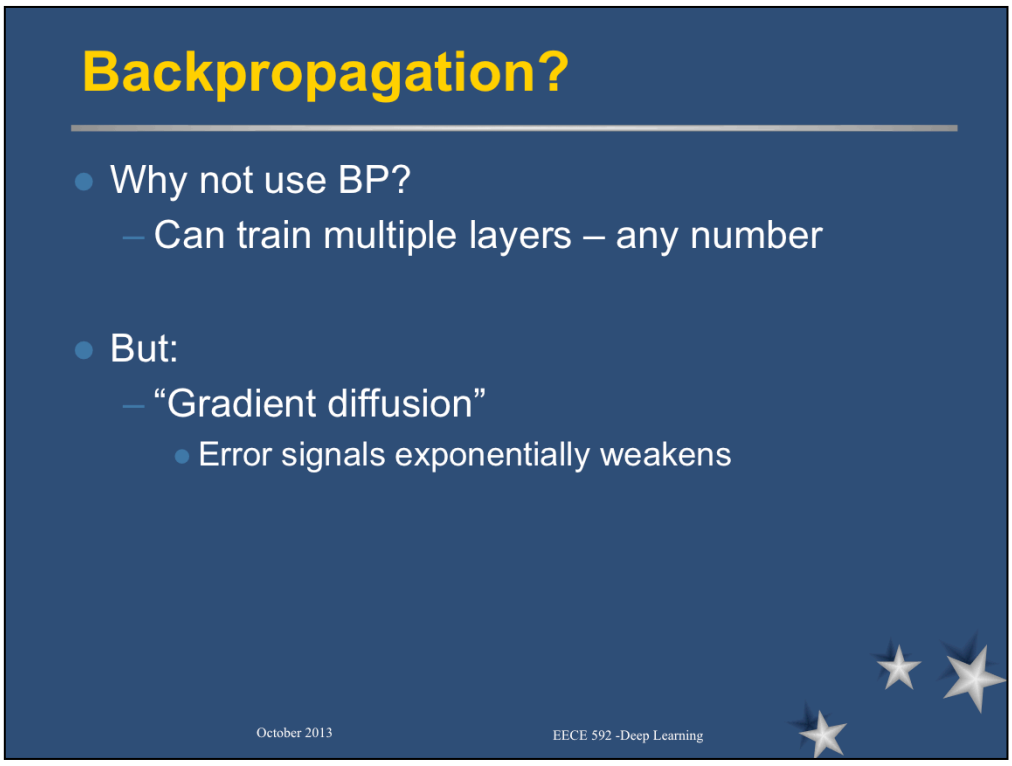
Deep Learning

- Why many layers?

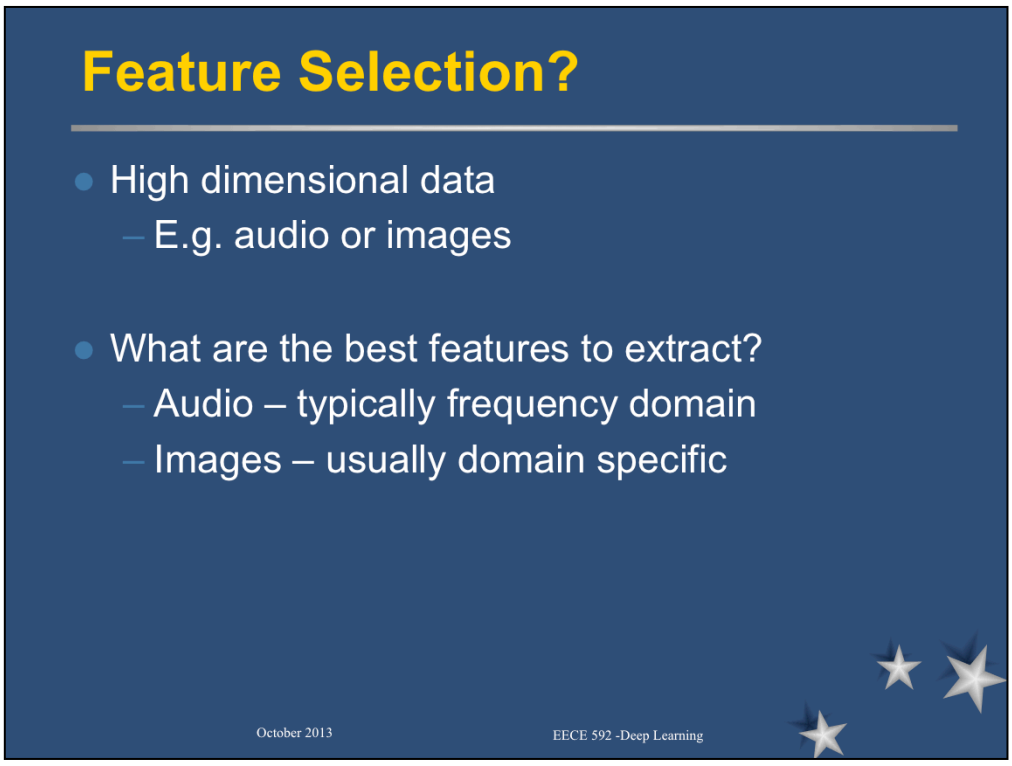


- http://theanalyticsstore.com/deep-learning/

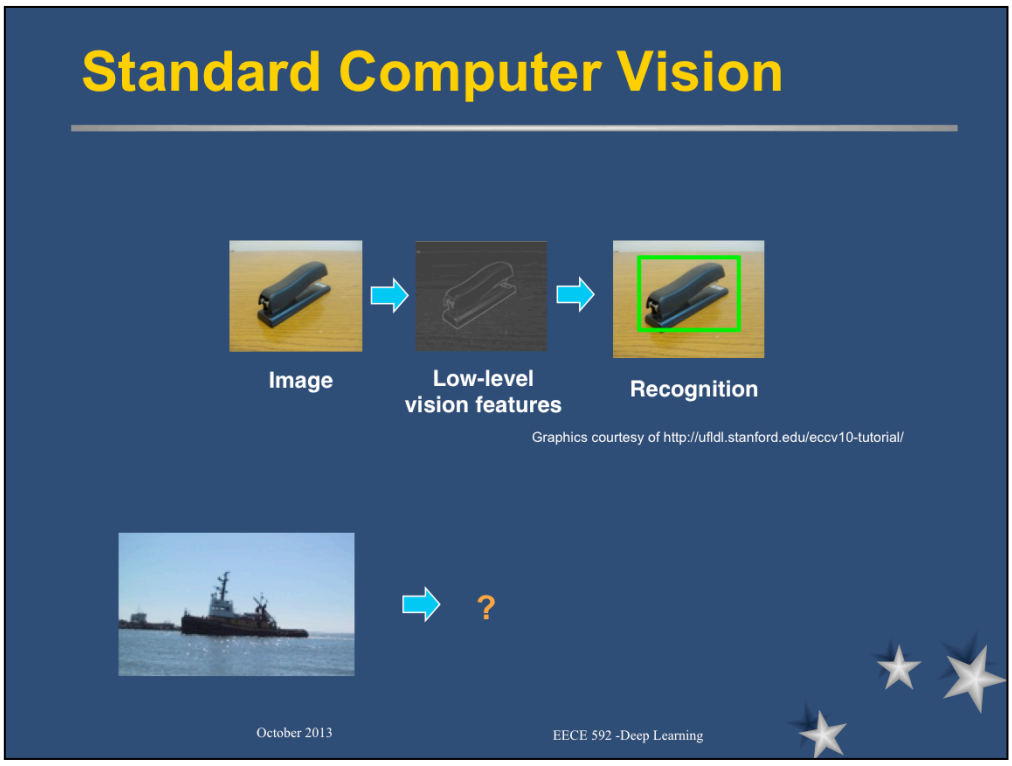
- A single layer in theory should be able to learn everything but may need very large numbers of neurons
- Humans organize knowledge hierarchically
- Even the brain is known to be multi-layered and hierarchical with different layers performing different levels of processing.
- So, in the neural network community, the benefit of multiple layers is well established. What has always been in doubt is how to train them.



- The BP learning algorithm can train any number of layers but it's slow. In practice, it is rare to see more than one hidden layer put into use. Even then, putting too many neurons in that layer can cause problems.
- The reason BP is slow at training many layers is "Gradient Diffusion". At each propagation, the magnitude of the error signal applied decreases exponentially. It rapidly diffuses.



For pattern recognition tasks in the audio or visual domain, the problem has traditionally involved the identification and specification of hand-engineered features in order to extract meaningful information from the signal. For example speech recognition typically use some form of Fourier analysis to extract features from the frequency domain. When dealing with images, the goal is to extract features that are in some way invariant to perspective, rotation, distortion, lighting etc. The quality of these features is a big factor in the overall success of the recognition process. However, hand-engineering features is not easy to get right.



- Traditionally, when faced with high dimensional data, we attempt to hand-engineer or find lower dimensional features that can help describe the content using less data. This is standard practice within the computer vision community.
- Take raw images for example. Even a small 256x256 image has an astronomical dimensionality. But what features should be used here for recognising pictures of staplers or boats, say? What if the images are instead high resolution x-ray images of bone fractures? Can we use the same features? Likely not. How do we find these features. D. Lowe's thesis proposed the use of pretty cool, perceptually significant features. But again, the allowed domain of images was fairly small.
- So feature engineering is hard, requires domain expertise and you don't really what the optimal features may be.

SIFT

- Scale Invariant Feature Transform
  - A popular and highly cited algorithm
  - invented by David Lowe, UBC
- But even Andrew Ng of Stanford says he doesn't understand it!
  - (http://www.youtube.com/watch?v=n1ViNeWhC24)

- Go to Google to learn more about SIFT.
- However, note that it is a complex algorithm.

Feature Learning ?

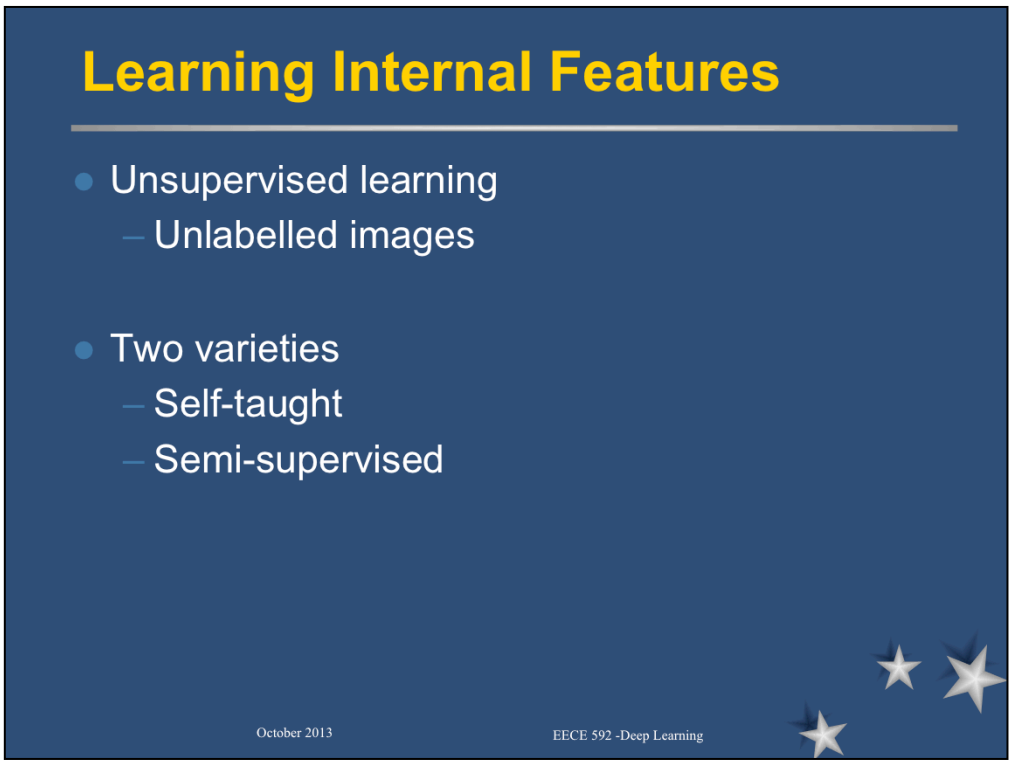
- Can features be learned?

?

- We use a deep neural net and train each layer one at a time, deepest ones first.
- Aim: to automatically develop feature representations

October 2013 EECE 592 -Deep Learning

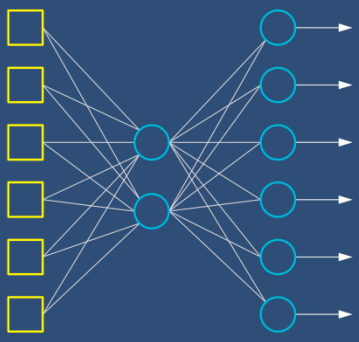
- Deep Learning is indeed all about feature learning!
- This was always the promise and the hope of neural networks. That they would be able to learn these feature representations.
- Attempting to train all layers at once proved difficult. It also required very large labelled data sets. Which until recently, were not available.



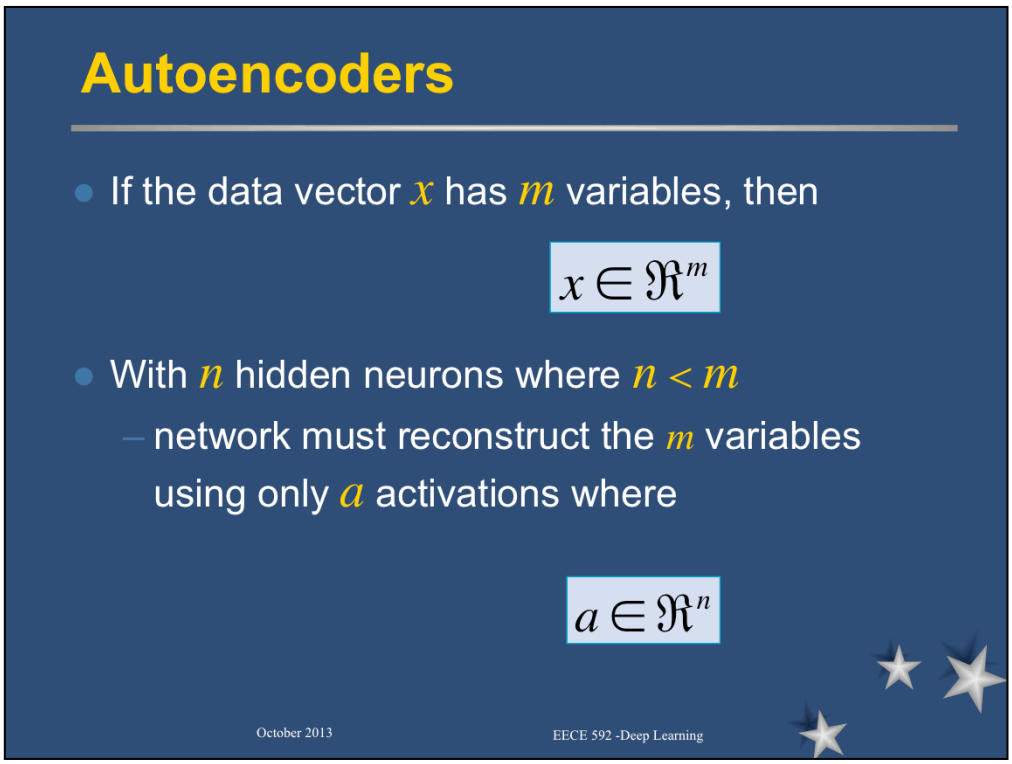
- As surprising as it seems, learning features in deep layers adopts an unsupervised learning paradigm. This can be done using a multi-layer perceptron configured as an autoencoder.
- Say that we want to learn to recognise images. And that we are interested in being able to detect pictures of boats and planes. There are two modes of unsupervised learning.
- Semi-supervised
    - In this case, you have a large collection of images. They are unlabelled but you know that they are either boats or planes. This kind of data is also not typically available. You would still have to perform some degree of categorisation to put each image into the correct 'bucket'.
- Self-taught
    - Your collection of images is still unlabelled, however, the distribution is different in that it contains pictures of boats, planes and perhaps many other items too. This kind of data is far more practical and readily available and the self-taught unsupervised approach is more applicable.

Autoencoders

- Inputs: vector x
- Outputs: the same vector x
- Hidden layer(s) somehow *encode* the vector x



An autoencoder is a neural net that takes an input vector $\boldsymbol{x}$ and generates on its output, the same vector $\boldsymbol{x}$. A single hidden layer is used, where the number of hidden neurons is small enough to force the network to find some efficient or compact representation of the data.

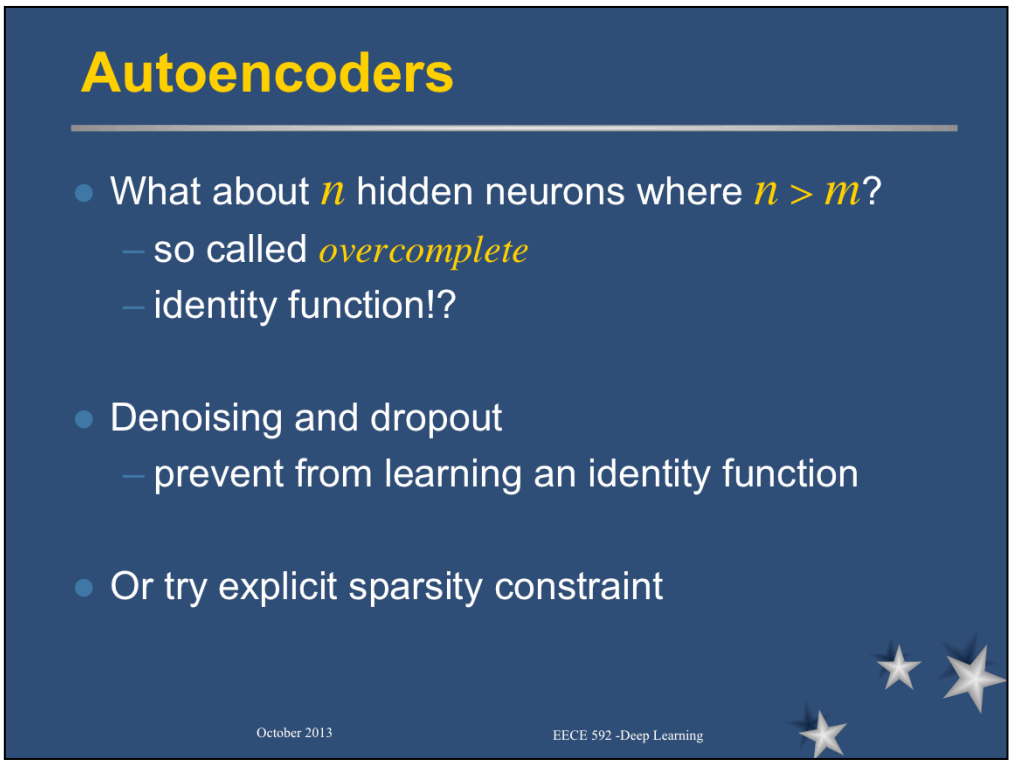


An autoencoder with an input/output vector of length m but only n hidden units where $n < m$, has to find a compressed representation that is able to reconstruct the m length vector using a much smaller state space.

State space of the data vector x: $x \in \Re^m$

State space of the hidden layer vector: $a \in \Re^n$

However this can only work if x is <u>not</u> a completely random distribution. If on the other hand there is structure inherent the input data, then such an approach should be able to discover those inherent correlations.



One potential problem with autoencoders is that you might expect the weights to simply learn the identity function. Which of course is useless. With $n<m$, this cannot happen. However with $n>m$ this is possible.

To force learning to avoid reaching an identity function, denoising or dropout can be used. These approaches work because they change the input being presented to the network by either corrupting, or actually removing parts of it. If this is done randomly, the autoencoder must learn a mapping based upon the underlying statistical dependencies in the input.

However, if any of that does not work, try to apply an explicit sparsity constraint.

Sparse autoencoders

- Why sparse representations?

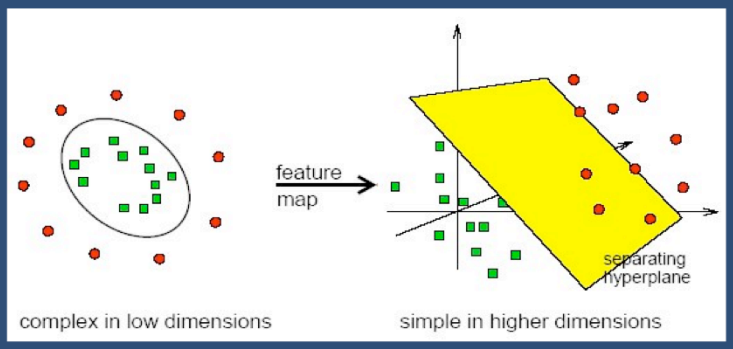


Figure from http://www.dtreg.com/svm.htm

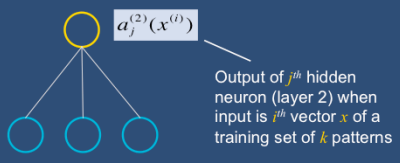
Typically, without sparse encoding, we want to limit the number of free parameters, i.e. hidden neurons in order to achieve good generalization. This is analogous to something like principal component analysis. But the brain is sparse!

The above diagram shows data points belonging to two separate classes. On the left, the data is represented using two dimensions. In this case, we need a circular decision boundary in order to separate them. If somehow, we map the same data into 3 dimensions, it maybe that we can separate the data using a simple linear hyperplane. The concept is applied in SVM (Support Vector Machines) in which a kernel function is used to transform data into a higher dimensional space. See http://www.dtreg.com/svm.htm

Sparse Autoencoding with BP

- Explicit sparsity constraint ρ where :

$$\hat{\rho}_j = \frac{1}{k}\sum_{i=1}^{k}\left[a_j^{(2)}\left(x^{(i)}\right)\right]$$



- ..such that $\rho \approx \hat{\rho}$
  - where ρ is small e.g. ρ = 0.05

We define a sparsity constraint, *ρ-hat* as the activation of neuron *j* in the hidden layer averaged over all patterns in a training set:

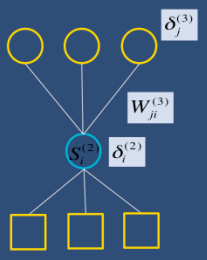
$$\hat{\rho}_j = \frac{1}{k}\sum_{i=1}^{k}\left[a_j^{(2)}\left(x^{(i)}\right)\right]$$

Where $a^{(2)}$ represents a neuron in the second layer. $x^{(i)}$ refers to the i^{th} pattern in a training set of *k* patterns. ρ is a sparsity parameter and is typically close to 0, e.g. $\rho = 0.05$. It is a way of effectively saying that we would like the average activation of each hidden neuron *j* to be close to 0.05. I.e. that we want only a few of the neurons in the hidden layer to be active at any one time. For a binary sigmoid threshold, this means that we want only a few to be close to 1 and most to close to 0.

Sparse Autoencoding with BP cont.

- Compute error signal as follows:

$$\delta_i^{(2)} = \left(\sum_{j=1} W_{ji}^{(3)} \delta_j^{(3)} \right) f'(s_i^{(2)})$$



- Where the additional penalty term is based upon the KL-divergence

Then, instead of the usual way of computing the error signal:

$$\delta_i^{(2)} = \left(\sum_{j=1} W_{ji}^{(3)} \delta_j^{(3)} \right) f'(s_i^{(2)})$$

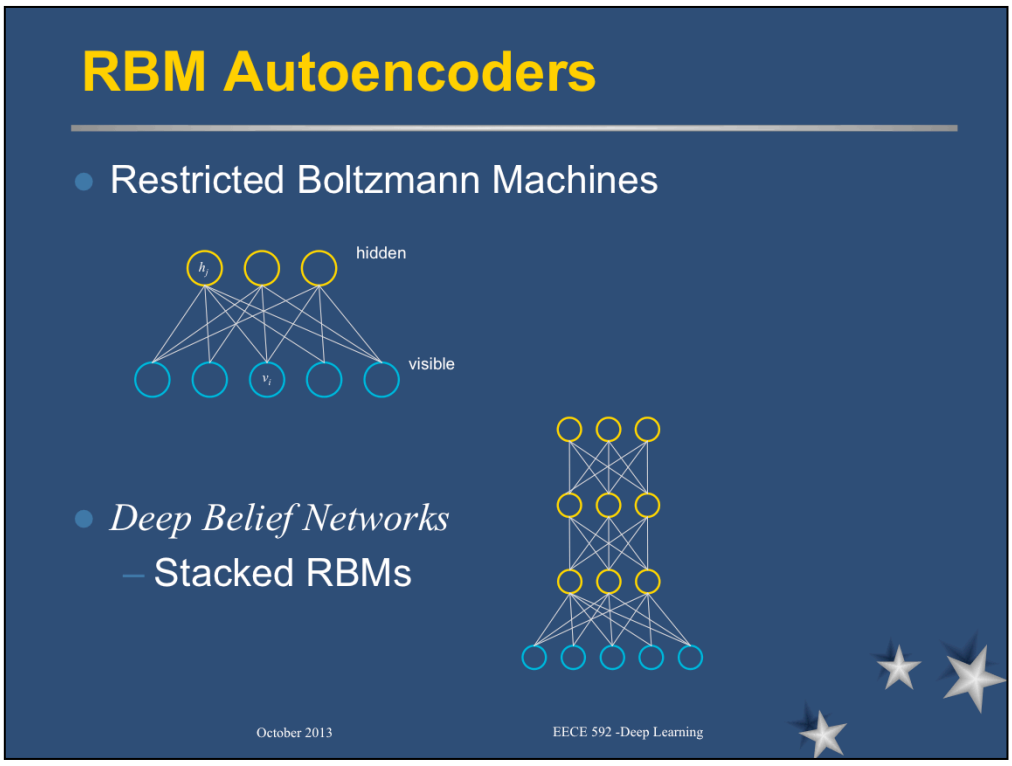
Do this instead:

$$\delta_i^{(2)} = \left(\sum_{j=1} W_{ji}^{(3)} \delta_j^{(3)} \right) f'(s_i^{(2)}) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right)$$

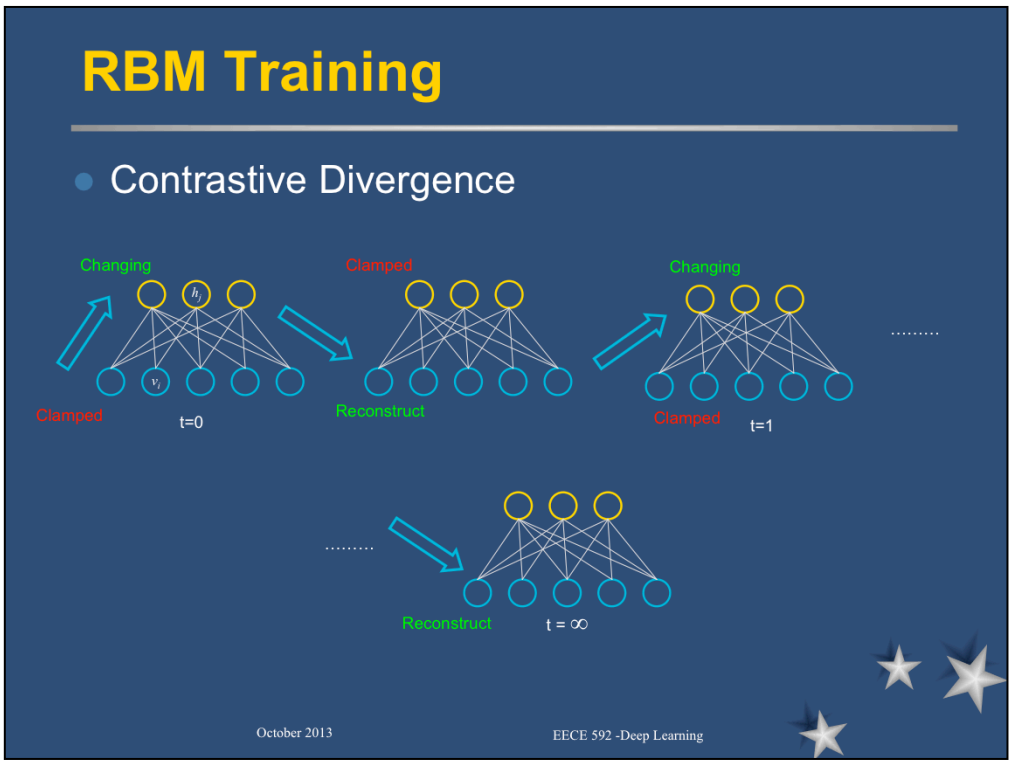
Where the penalty term is based upon the Kullback-Leiber divergence, KL $(\rho \,\|\, \hat{\rho}_j)$ This is a standard measure of the difference between two distributions. β is a coefficient used to control the fraction of the penalty term to be applied.

The trick is computing ρ-*hat*. This involves averaging the activation of each hidden layer neuron over all patterns. This should be done during the forward propagation step and before any back-propagation.

Ref: Notes based upon CS294A lecture notes by Andrew Ng



Restricted Boltzmann Machines (RBM) have also been used for Deep Learning. They take the place of the autoencoder but effectively do the same job. RBMs have a single layer of hidden neurons and no intra-layer connections. Otherwise they are similar to regular Boltzmann Machines e.g. the neurons are stochastic. Deep Belief Networks (DBN) are stacked RBMs in which each successive layer of hidden neurons is trained separately and provides the input for the next, in the same way as for autoencoders.



You may recall that the training for the Boltzmann Machine is quite complex, involving reaching thermal equilibrium and then taking statistics.

The training of an RBM can be described in similar terms but there is a short cut. In the above diagram, we see consecutive phases in which either hidden neurons are being updated based upon clamped input units, or visible units being updated based upon clamped hidden units. In the literature the former phase is referred to as the *positive* phase and the latter as the *negative* phase.

At each step the outputs are updated according to the stochastic sigmoid function and after an infinite number of steps the system is expected to reach equilibrium. At which point we can update the weights as follows:

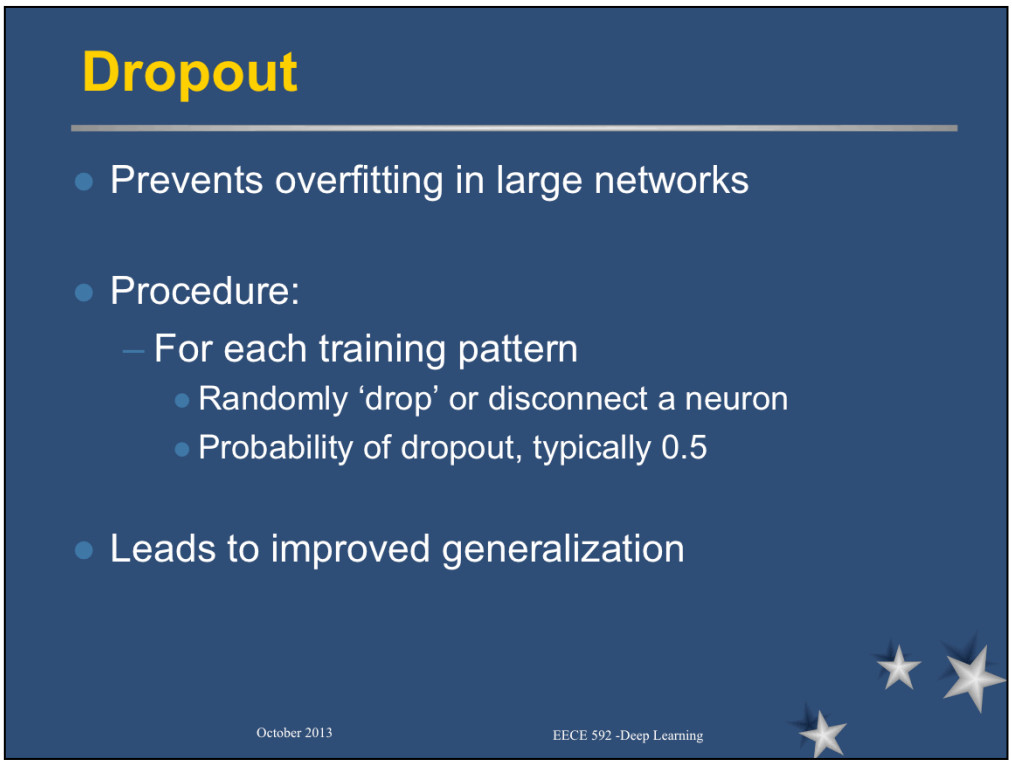
$$\Delta w_{ij} = \alpha(< v_i h_j >^0 - < v_i h_j >^\infty)$$

Where α is the step size, v_i a visible unit, h_j a hidden unit and $<v_i h_j>$ is the product of the outputs averaged over all input vectors. It turns out that a simpler, short cut works for RBMs. Instead of the above equation we do the following:

$$\Delta w_{ij} = \alpha(< v_i h_j >^0 - < v_i h_j >^1)$$

Meaning we just perform <u>one</u> step in the chain! This is much easier to compute and seems to work. The process is called contrastive divergence.

Ref: UTML TR 2010–003, *A Practical Guide to Training Restricted Boltzmann*

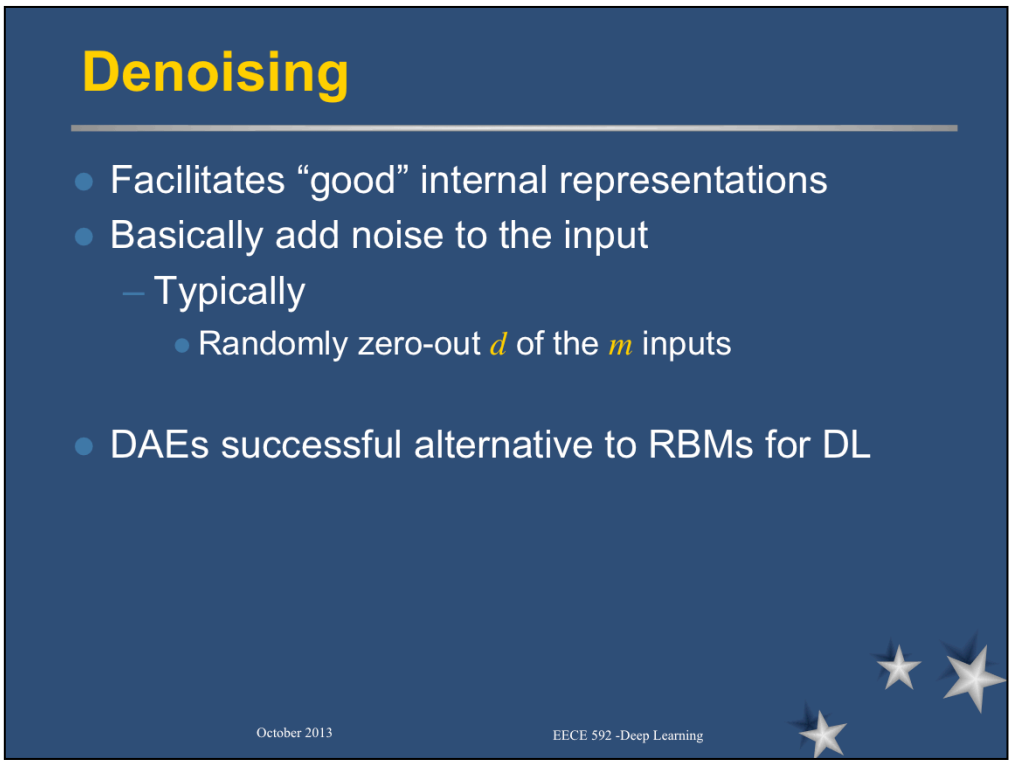


A neural net with a large number of hidden neurons, e.g. greater than the dimensionality of the input data will suffer from overfitting.

One way to address this is to perform "dropout". The term refers to the random removal of a hidden neuron during training. Typically, upon presentation of each training pattern, a hidden neuron is removed with 50% probability. At test/usage time, all neurons are used.

A way of looking at this is that we are actually training many neural nets, that largely share the same weights. E.g. if there are n neurons, then there are 2^n possible configurations. At test/usage times, the output of these different networks is essentially the averaged output from all of these. The concept is similar to that of the geometric averaging ensembles, e.g. random forests. Here many collections of different decision trees are separately trained on the same data and their outputs averaged together. With dropout, we can do the same with only one model.

Ref: *Improving neural networks by preventing co-adaptation of feature detectors* G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov. (http://arxiv.org/abs/1207.0580)



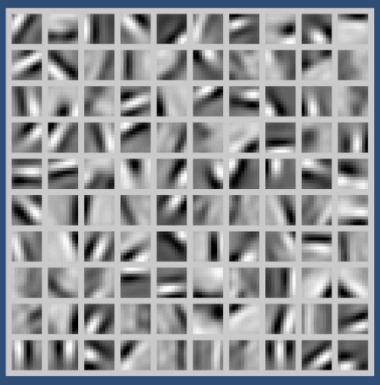
The term *denoising* refers to the process of reconstructing an input vector $\boldsymbol{x}$ from a noisy version of $\boldsymbol{x}$. It typically involves the use of corrupted inputs when training an autoencoder where one such procedure is the simple "zeroing" of a fixed number of randomly selected variables of the vector. Much like dropout, the motivation for denoising is to improve the networks ability to work with *larger* numbers of neurons in the hidden layer and hence arrive at good internal representations. I.e. representations that reflect the statistical variations in the training data. Here *large* refers to a number of hidden neurons greater than the dimensionality of the data.

Denoising Autoencoders (DAE) are a successful alternative to the use of RBMs for training deep layers.

Sparse Autoencoders: results

- Visualization of a hidden layer

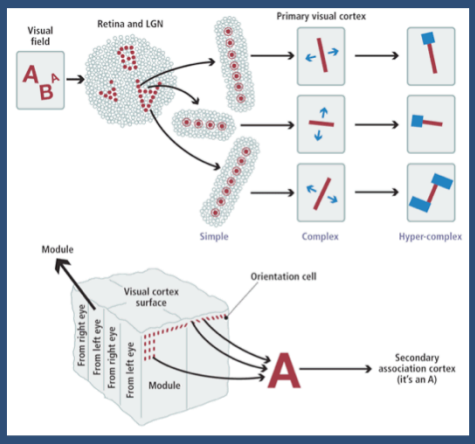
http://ufldl.stanford.edu/wiki/index.php/Visualizing_a_Trained_Autoencoder



- The image shows a visual representation of a hidden layer of n=100 neurons that was trained as a sparse autoencoder using 10x10 images.
- Each 'square' in the grid above represents the actual image that would generates a maximal response in each of the 100 hidden layer neurons.
- It appears that the hidden neurons have become edge detectors, with each one having a preference for a different orientation or position of edge.
- Note that this autoencoder was trained not using dropout, but an approach as described in http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity

Visual Cortex

- Hubel & Weisel (1969)
  - The same detectors in the brain!

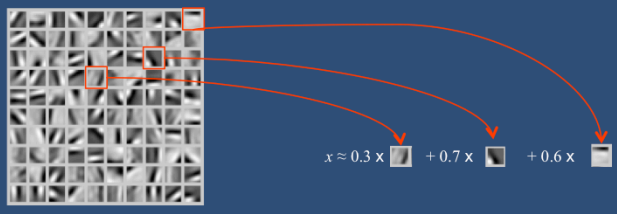


It turns out that the edge detectors that sparse autoencoders generate, have parallels in human biology. Hubel and Weisel back in 1969, demonstrated that there are many neurons in the visual cortex of the brain that detect edges. Further the physical organisation of these cells is quite regular, showing a gradual varying of preference to the orientation of an edge based upon physical location within the cortex.

How autoenoders work

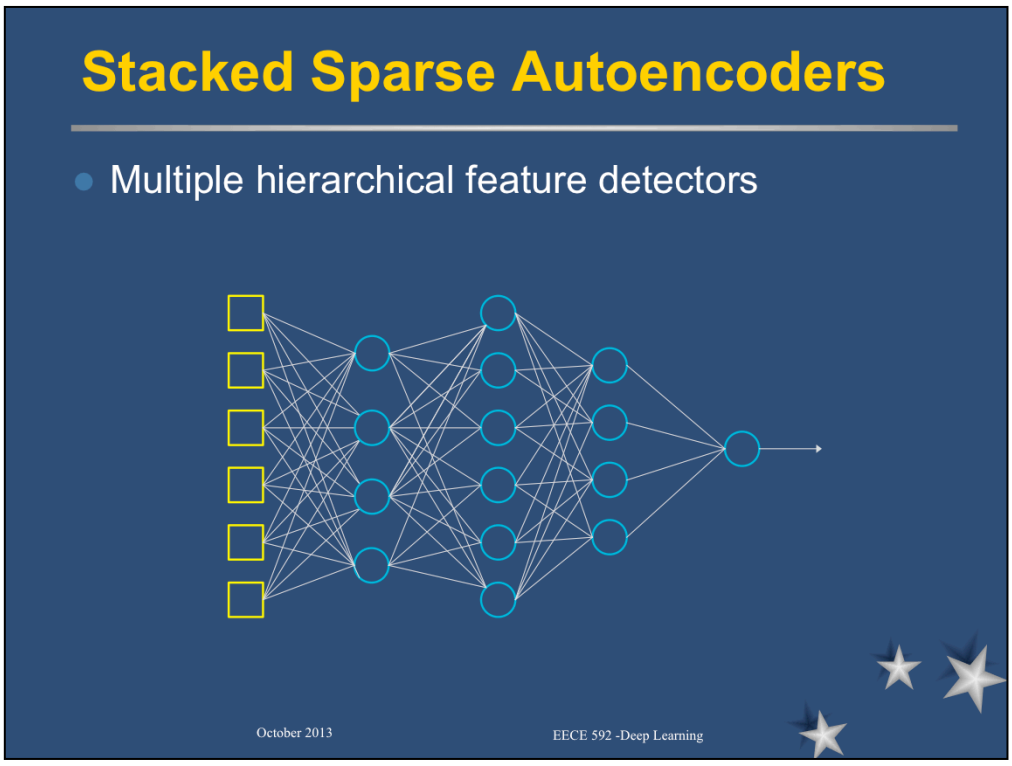
- Hidden layer vector for an input:
  - x =[0.0, 0.0, 0.0,…0.3,…0.0,…0.7,…0.6,….0.0]

Learned bases [ϕ_0, ϕ_1, ϕ_2, …… ϕ_{99}]

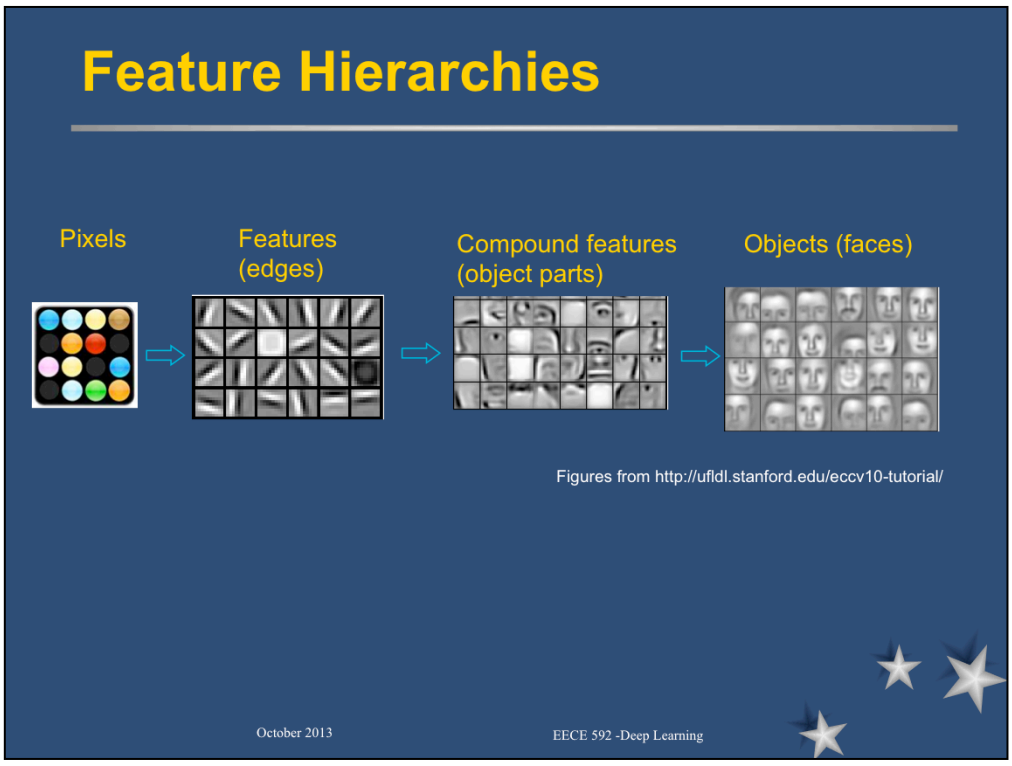


The hidden layer can be thought of as a set of bases, where each base is a learned feature.

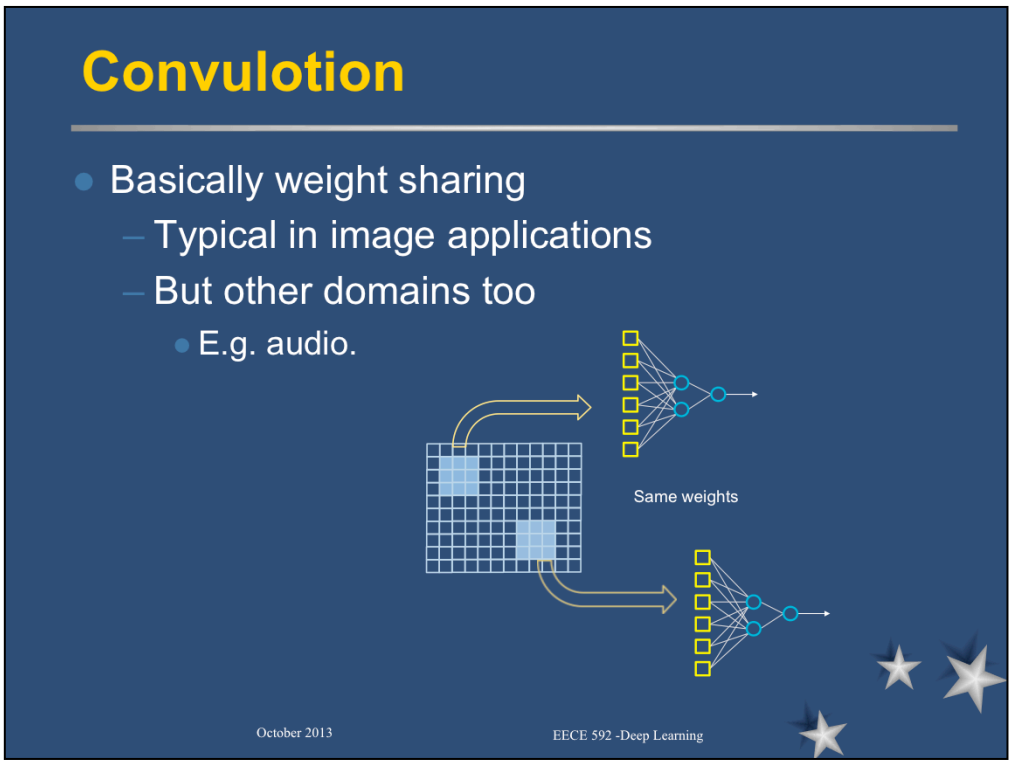
When an input image is applied to the net, the output at the hidden layer is sparse, i.e. only a small fraction of the bases will match. The implication for image recognition is that an image can be represented as a set of edges.



In deep learning, it is common to have more than one hidden layer. Deep networks of upto 9 hidden layers are not uncommon. Much like a Deep Belief Network which is built using layers of RBMs, to build a deep neural net, you essentially stack multiple sparse autoencoders, at each stage, training the next hidden layer using the output from the previous in the same unsupervised fashion. E.g. using backpropagation with dropout.



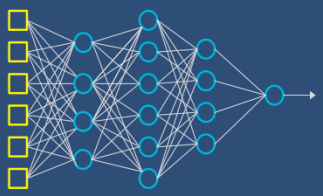
With a deep neural net, each layer develops a feature representation from the previous. With images as input, the first hidden layer develops edge detectors as we have seen. As you continue stacking more and more autoencoders, subsequent layers develop feature detectors based upon deeper features and as such a hierarchy of features is observed. For example, the second layer develops compound feature detectors which may be the parts of an object. (In this case faces).



Convolution is the term used to describe the act of replicating pre-trained neural networks and apply them to other regions of the input. In biological terms, the area of the input that forms the partial input is known as a *receptive field*. In the deep learning literature, the convolved neural nets or rather their outputs are known as kernel functions or *kernels*. These kernels are effectively feature detectors. Since convolution networks favour local connectivity, in terms of size, they are much smaller than fully connected feed forward networks and as therefore are easier to train.

Fine Tuning

- Last layer typically performs categorisation

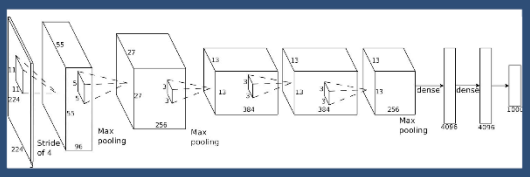


- Final step
  - Train whole network
  - Regular supervised BP

"Fine tuning" is the final stage of Deep Learning. Once all your autoencoder layers have been trained and stacked, the last stage is to finally perform supervised learning on the model. Since the autoencoders have already learned appropriate feature detectors any adjustments here really amount to fine tuning. On the other hand, the last layer of weights is brand new and will undergo the most training here to complete the categorisation task.

Image Classification

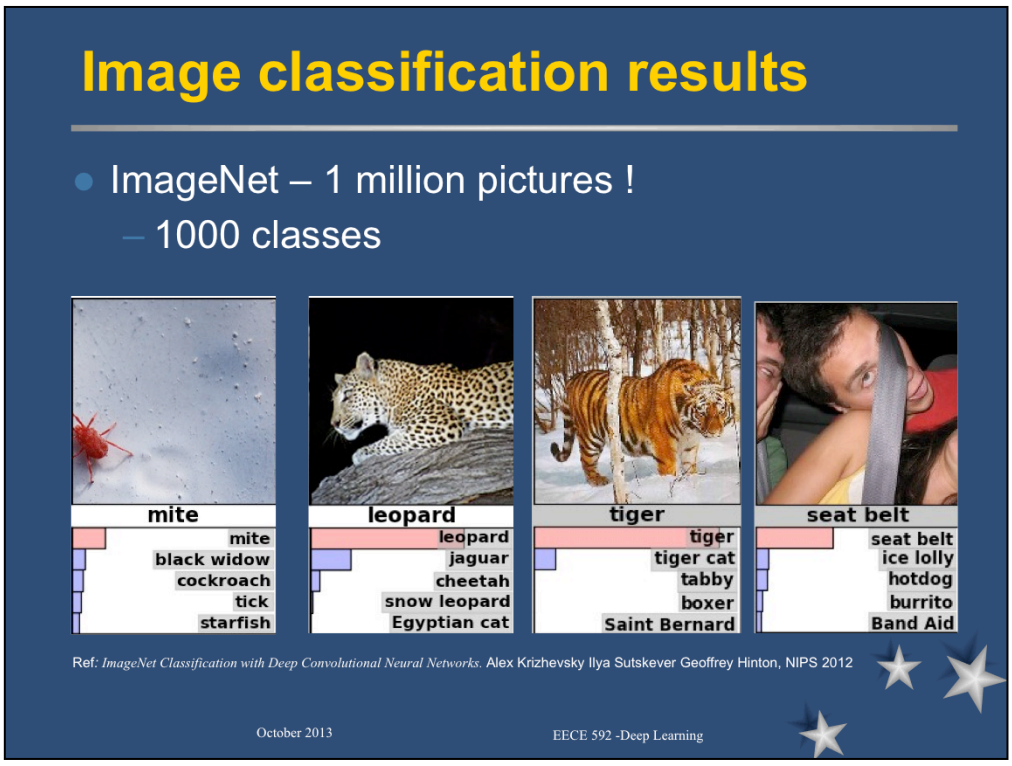
- Deep convolution network:
  - 650k neurons
  - 5 convolution layers
  - 1.2 million images
  - Final layer 1000 outputs
- Results:
  - 1000 classes
    - Top 1 error rate of 37.5%
    - Top 5 error rate of 17%



Ref: *ImageNet Classification with Deep Convolutional Neural Networks,* Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, NIPS 2012

The approach was used to enter a competition, which won with a best top-5 error rate of 15.3%. The 2nd place entry came in at 26.2%.

By the way, the training took about a week to complete on two GTX 580 3GB GPUs.



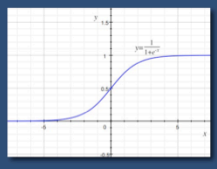
In these images, the title is the target classification. Below that are the results from the deep network in order of top five guesses. The red shaded result is of course the correct one.

Image Classification results

- Close guesses….

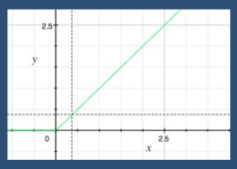
Rectified linear threshold

- Tips and tricks
- Threshold functions
  - Typically sigmoidal



  - In Deep Learning use rectilinear

$$y = \max(0, x)$$



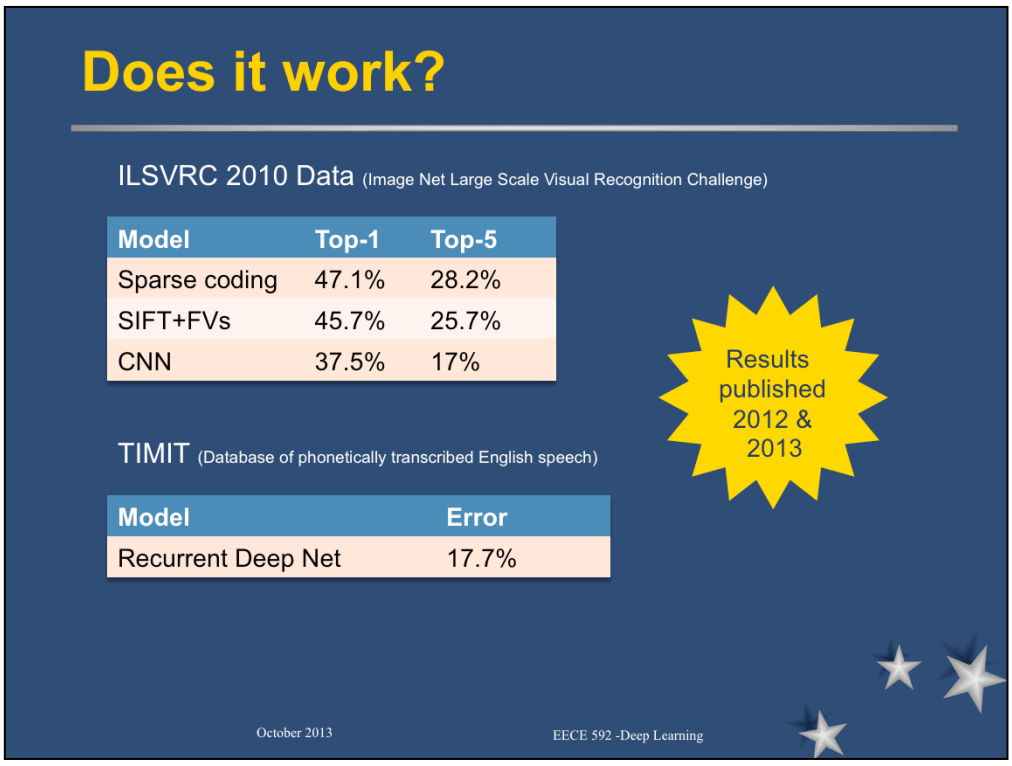
October 2013 EECE 592 -Deep Learning

When applying dropout, any threshold can be used. The exponential sigmoid is typically used with backpropagation. However, linear thresholds can also work. Various publications have reported that fastest training times are obtained with rectified linear thresholds e.g. $y = \max(0, x)$

See *Improving Deep Neural Networks for LVCSR Using Rectified Linear Units and Dropout.* George E. Dahl, Tara N. Sainath, Geoffrey E. Hinton (http://www.cs.toronto.edu/~hinton/absps/georgerectified.pdf)

How to Build a DNN: Recap

1. Obtain a large amount of unlabelled data
2. Use BP to train a feed-forward network configured as as autoencoder. Use "dropout", "denoising" or sparse encoding to avoid overfitting while using a reasonable number of neurons
    1. Add another layer
    2. Train that using the output of the previous layer
    3. Continue adding layers as necessary
3. Add a final output layer
4. "Fine tune" the network via supervised BP.



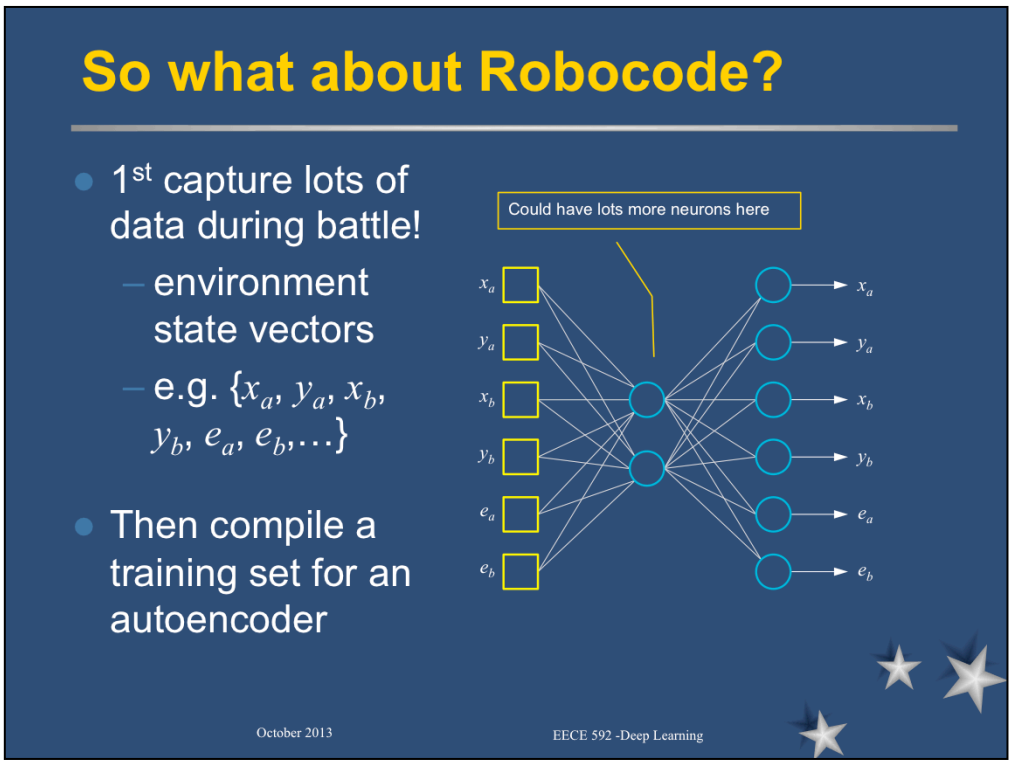
Convolution neural networks trained via deep learning have consistently been shown to better the state of the art in image recognition. [1] report top 1 and top 5 error rates significantly better than the previous best approach. In this case SIFT using hand engineered feature vectors. While an improvement of 8.2% in the top-1 error rate does not sound like much, consider that in the last decade, annual improvements in image recognition have been typically fractions of a per-cent!

Similar successes with audio data. E.g. a recurrent deep neural network has been shown to have an error rate of only 17.7% for speech recognition against the TIMIT data set. The authors claim this is the best recorded result to-date.

References

[1] Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012)
ImageNet Classification with Deep Convolutional Neural Networks
Advances in Neural Information Processing 25, MIT Press, Cambridge, MA

[2] Graves, A., Mohamed, A. and Hinton, G. E. (2013)
Speech Recognition with Deep Recurrent Neural Networks
In IEEE International Conference on Acoustic Speech and Signal Processing (ICASSP 2013) Vancouver, 2013



So if you want to apply Deep Learning to Robocode, here's how it could be done. The idea requires that first you capture lots of data! Unlabeled environmental state vectors are all you need. This data should be easy to obtain by using a tank whose sole job is to continuously write state vectors to file. This tank will likely need to stay alive for a while so perhaps use one of the stronger canned robots. Or perhaps collect data using a number of different pairs of robots. The goal here is to obtain sample states that represent a broad spectrum of expected situations. This data can then be used to train an autoencoder using the techniques presented in these slides. If the training was successful, you would obtain an internal representation where each hidden neuron is a detector for a particular kind of feature. You might want to try to visualize each hidden neuron to confirm this. (A neuron will have the highest activation when presented with an input vector that is identical to its weight vector). Theoretically, you can stack layers of hidden neurons in the autoencoder as deep as you like.

The final phase is "fine tuning". This involves adding an output layer to predict $Q(s,a)$. The network now behaves as a function approximator in your RL implementation. With a pre-learned set of internal neurons that are able to detect the significant features of the representation, the hope is the approximator should be more readily able to generalize the underlying Q-function. Good luck!