

Algorithms in Nature

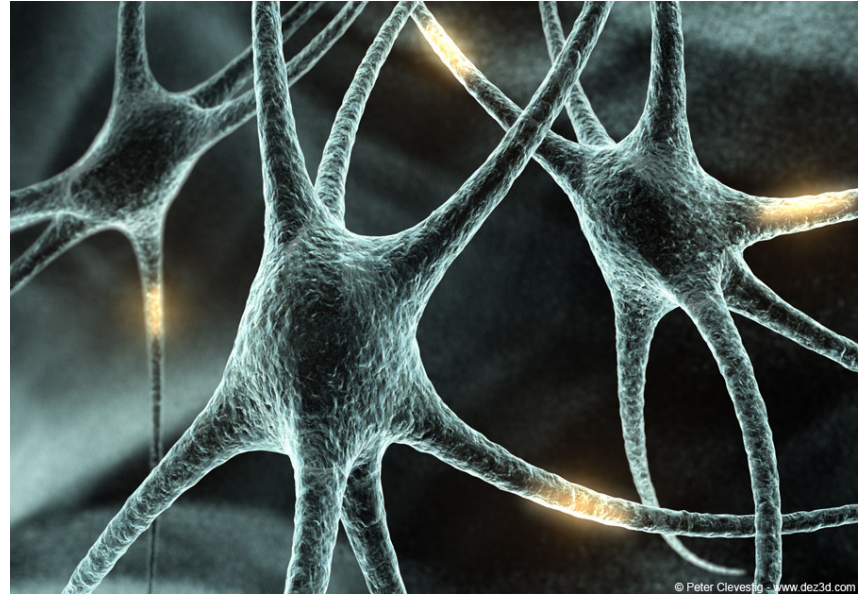
Neural Networks (NN)

Mimicking the brain

- In the early days of AI there was a lot of interest in developing models that can mimic human thinking.
- While no one knew exactly how the brain works (and, even though there was a lot of progress since, there is still a lot we do not know), some of the basic computational units were known
- A key component of these units is the neuron.

The Neuron

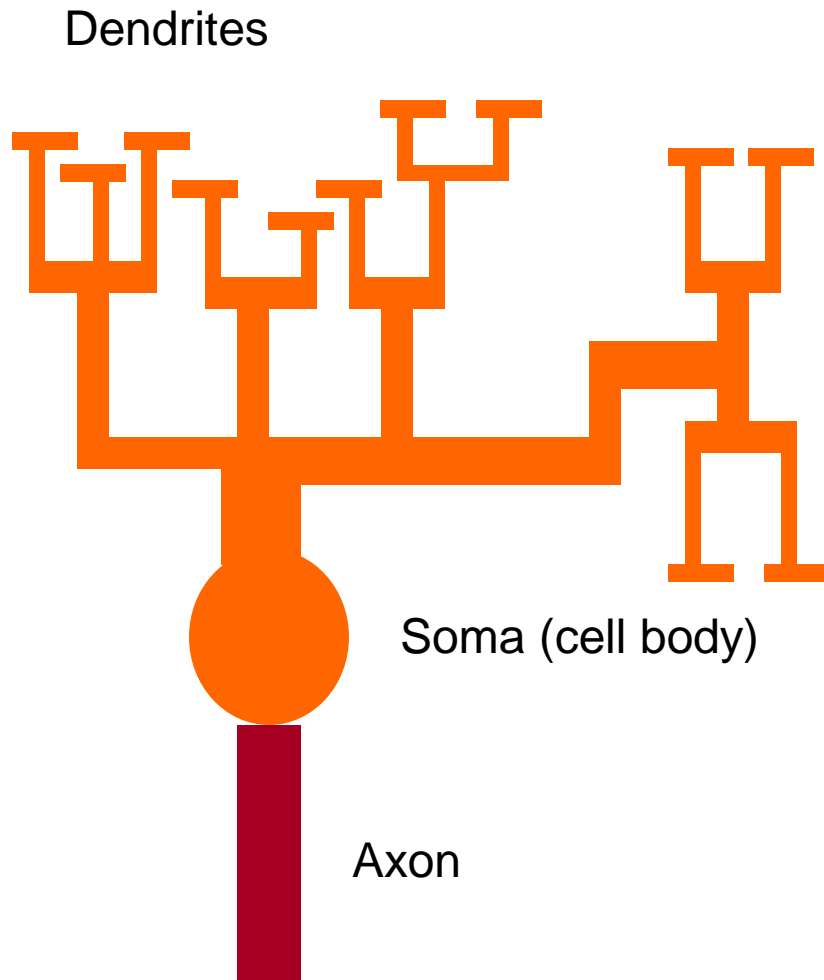
- A cell in the brain
- Highly connected to other neurons
- Thought to perform computations by integrating signals from other neurons
- Outputs of these computation may be transmitted to one or more neurons



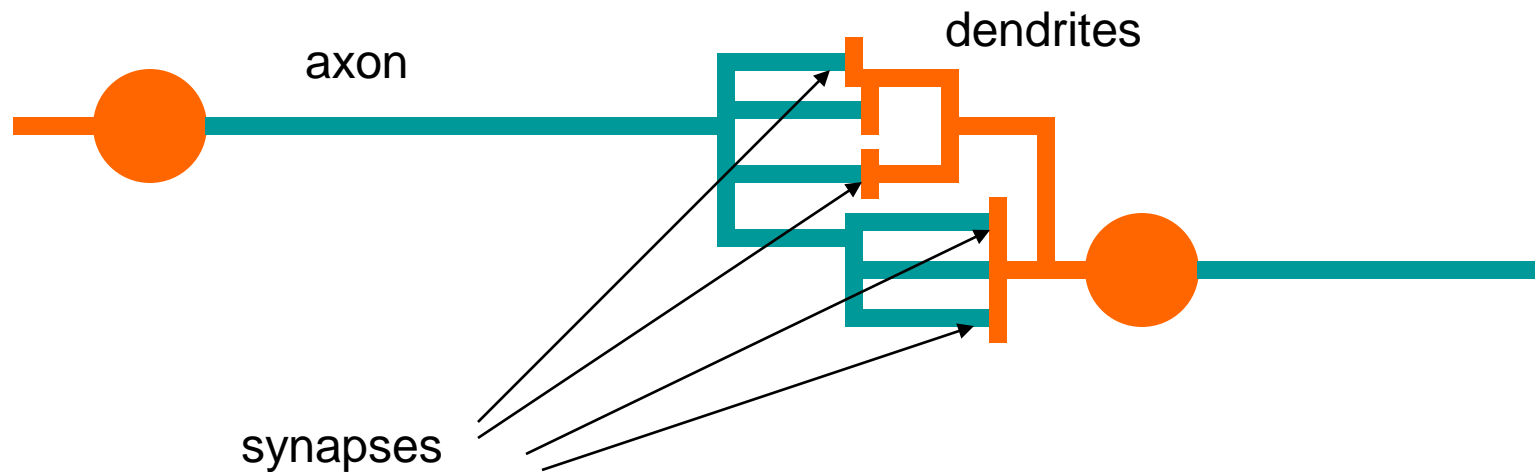
Biological inspiration

- The nervous system is built using relatively simple units, the neurons, so copying their behaviour and functionality could provide solutions to problems related to interpretation and optimization.

Biological inspiration



Biological inspiration



Synapses are the edges in this network, responsible for transmitting information between the neurons

Biological inspiration

- The spikes travelling along the axon of the pre-synaptic neuron trigger the release of neurotransmitter substances at the synapse.
- The neurotransmitters cause excitation or inhibition in the dendrite of the post-synaptic neuron.
- The integration of the excitatory and inhibitory signals may produce spikes in the post-synaptic neuron.
- The contribution of the signals depends on the strength of the synaptic connection.

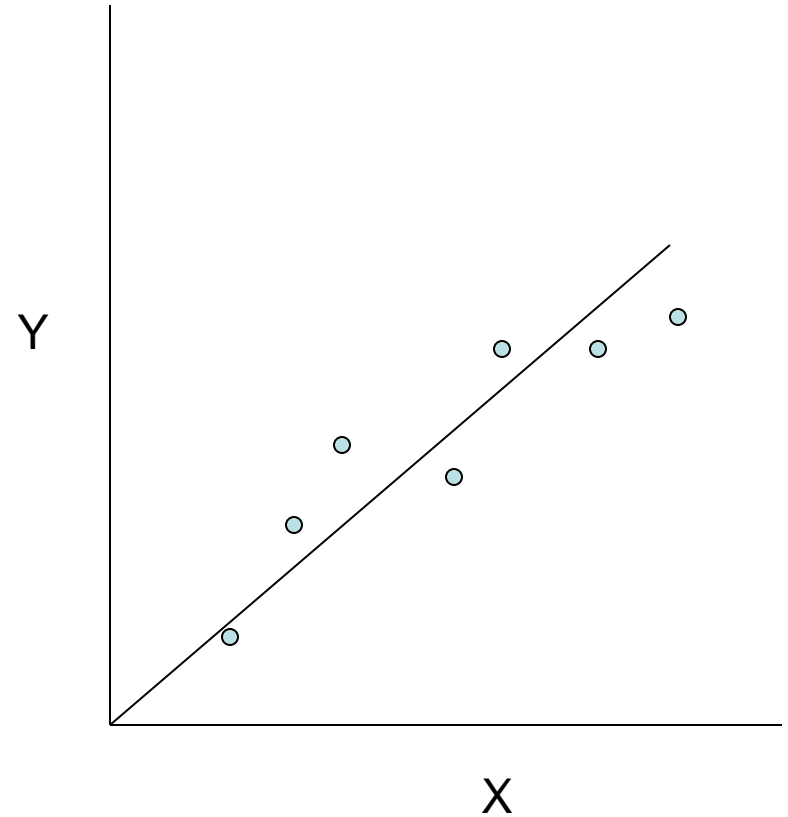
What can we do with NN?

- Classification
- Regression
 - Input: Real valued variables
 - Output: One or more real values
- Examples:
 - Predict the price of Google's stock from Microsoft's stock price
 - Predict distance to obstacle from various sensors

Recall: Regression

- In linear regression we assume that y and x are related with the following equation:

$$y = wX + \varepsilon$$



Multivariate regression: Least squares

- We already presented a solution for determining the parameters of a general linear regression problem.

$$y = \mathbf{w}^T \phi(\mathbf{x}) + \varepsilon$$

Define:

$$\Phi = \begin{pmatrix} \phi_0(x^1) & \phi_1(x^1) & \cdots & \phi_m(x^1) \\ \phi_0(x^2) & \phi_1(x^2) & \cdots & \phi_m(x^2) \\ \vdots & \vdots & \cdots & \vdots \\ \phi_0(x^n) & \phi_1(x^n) & \cdots & \phi_m(x^n) \end{pmatrix}$$

Then deriving \mathbf{w} we get:

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$$

Multivariate regression: Least squares

- The solution turns out to be: $\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$

we need to invert a k by k matrix
($k-1$ is the number of features)

- This takes $O(k^3)$
- Depending on k this can be rather slow

Where we are

- Linear regression – solved!
- But
 - Solution may be slow
 - Does not address general regression problems of the form

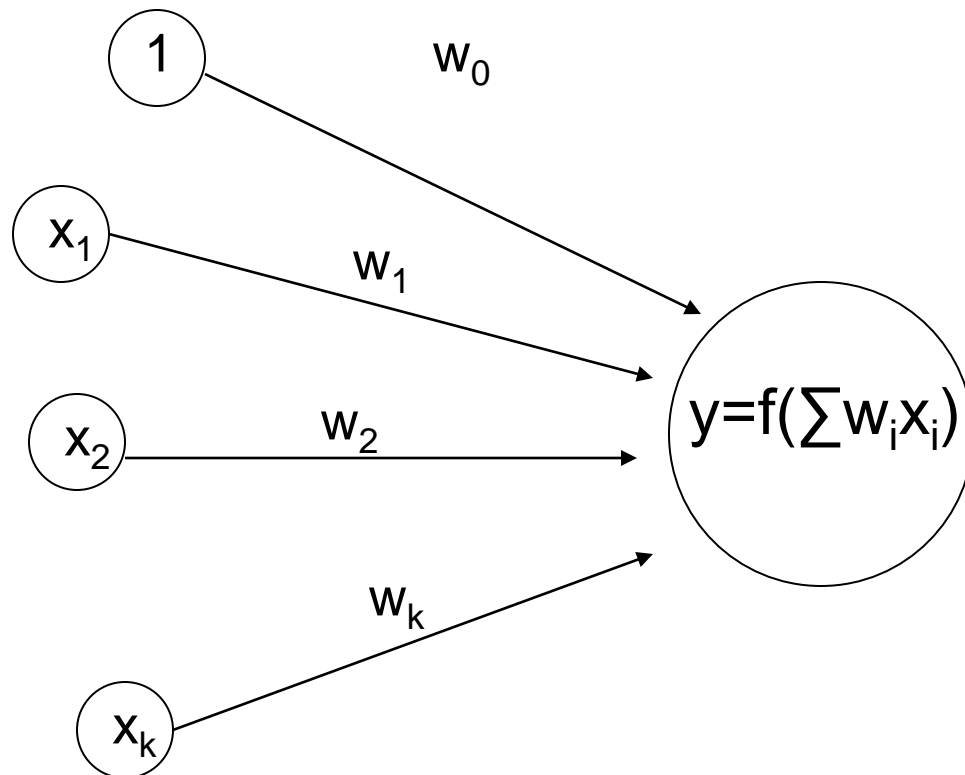
$$\mathbf{y} = f(\mathbf{w}^T \mathbf{x})$$

Back to NN: Perceptron

- The basic processing unit of a neural net

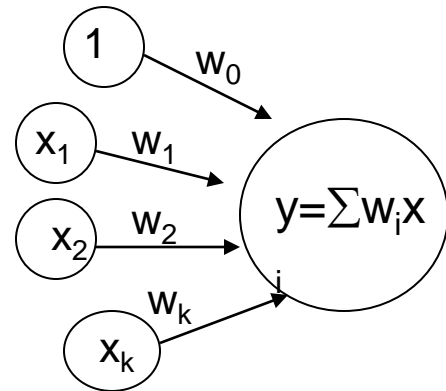
Input layer

Output layer

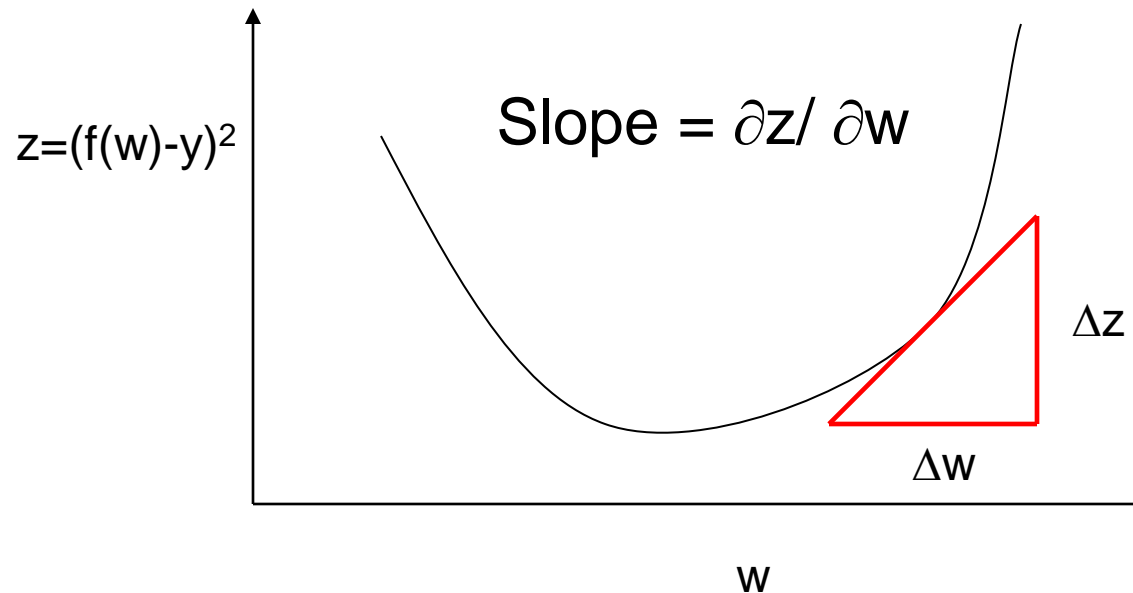


Linear regression

- Lets start by setting $f(\sum w_i x_i) = \sum w_i x_i$
- We are back to linear regression
- Unlike our original linear regression solution, for perceptrons we will use a different strategy
- Why?



Gradient descent



- Going in the *opposite* direction to the slope will lead to a smaller z
- But not too much, otherwise we would go beyond the optimal w

Gradient descent

- Going in the *opposite* direction to the slope will lead to a smaller z
- But not too much, otherwise we would go beyond the optimal w
- We thus update the weights by setting:

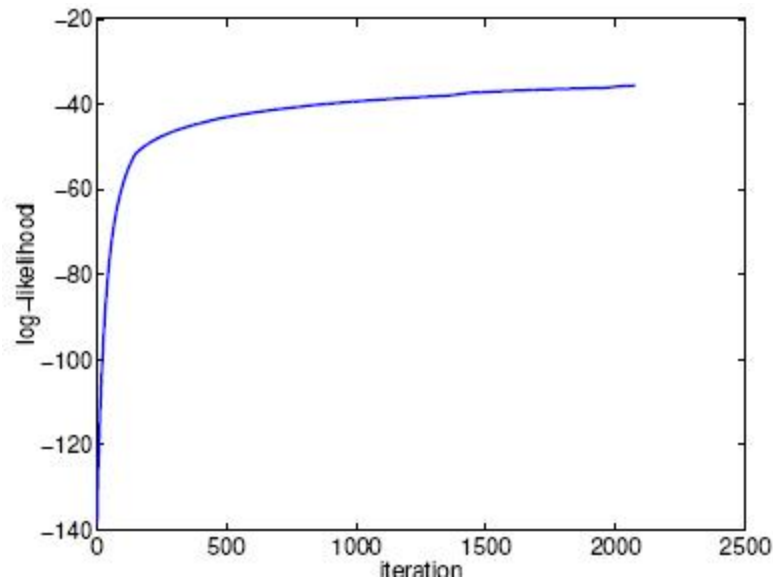
$$w \leftarrow w - \lambda \frac{\partial z}{\partial w}$$

where λ is small constant which is intended to prevent us from passing the optimal w

Example when choosing the ‘right’

λ

- We get a monotonically decreasing error as we perform more updates



Gradient descent for linear regression

- Taking the derivative w.r.t. to each w_i for a sample x :

$$\frac{\partial}{\partial w_i} \left(y - \sum_k w_k x_k \right)^2 = -2x_i (y - \sum_k w_k x_k)$$

- And if we have n measurements then

$$\frac{\partial}{\partial w_i} \sum_{j=1}^n (y_j - \mathbf{w}^T \mathbf{x}_j)^2 = -2 \sum_{j=1}^n x_{j,i} (y_j - \mathbf{w}^T \mathbf{x}_j)$$

where $x_{j,i}$ is the i 'th value of the j 'th input vector

Gradient descent for linear regression

- If we have n measurements then

$$\frac{\partial}{\partial w_i} \sum_{j=1}^n (y_j - \mathbf{w}^T \mathbf{x}_j)^2 = -2 \sum_{j=1}^n x_{j,i} (y_j - \mathbf{w}^T \mathbf{x}_j)$$

- Set $\delta_j = (y_j - \mathbf{w}^T \mathbf{x}_j)$
- Then our update rule can be written as

$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^n x_{j,i} \delta_j$$

Gradient descent algorithm for linear regression

1. Chose λ
2. Start with a guess for \mathbf{w}
3. Compute δ_j for all j
4. For all i set $w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^n x_{j,i} \delta_j$
5. If no improvement for $\sum_{j=1}^n (y_j - \mathbf{w}^T \mathbf{x}_j)^2$
stop. Otherwise go to step 3

Gradient descent vs. matrix inversion

- Advantages of matrix inversion
 - No iterations
 - No need to specify parameters
 - Closed form solution in a predictable time
- Advantages of gradient descent
 - Applicable regardless of the number of parameters
 - General, applies to other forms of regression

Perceptrons for classification

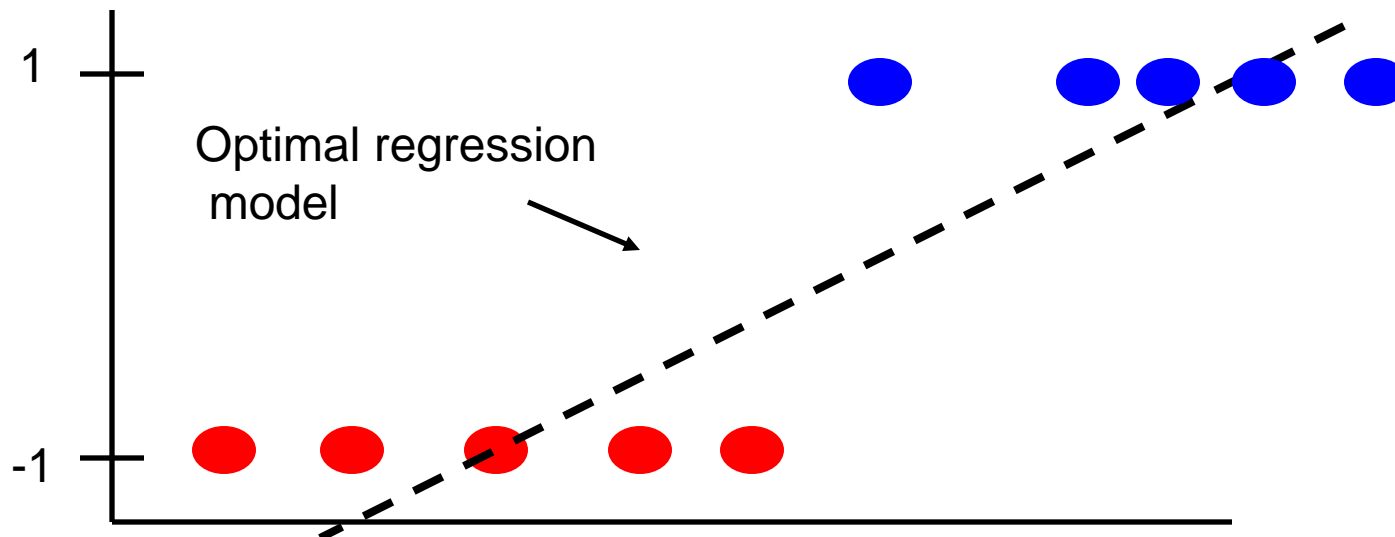
- So far we discussed regression
- However, perceptrons can also be used for classification
- For example, output 1 is $\mathbf{w}^T \mathbf{x} > 0$ and -1 otherwise
- Problem?

Regression for classification

- Assume we would like to use linear regression to learn the parameters for a classification problem
- Problems?

$$w^T x \geq 0 \Rightarrow \text{classify as } 1$$

$$w^T x < 0 \Rightarrow \text{classify as } -1$$



The sigmoid function

$$p(y | x; \theta)$$

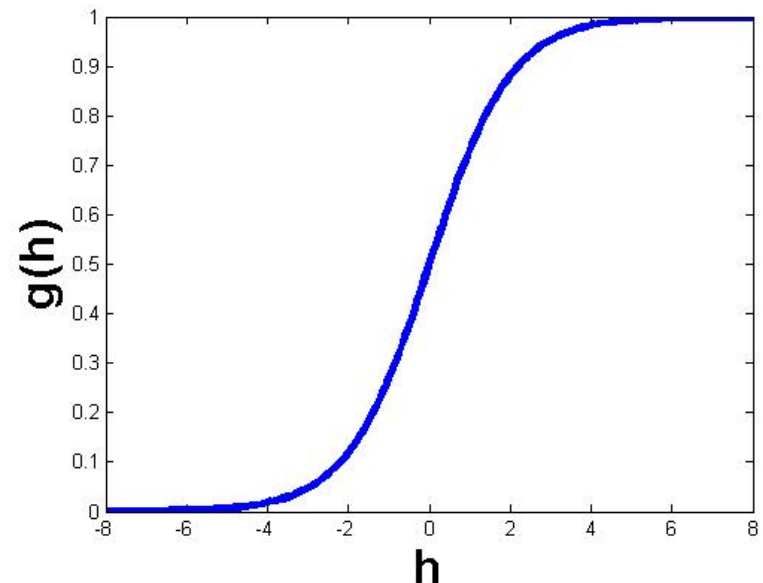
- To classify using regression models we replace the linear function with the sigmoid

Always between 0 and 1 $\longrightarrow g(h) = \frac{1}{1 + e^{-h}}$

$$p(y = 0 | x; \theta) = g(w^T x) = \frac{1}{1 + e^{w^T x}}$$

- Using the sigmoid we set (for binary classification problems)

$$p(y = 1 | x; \theta) = 1 - g(w^T x) = \frac{e^{w^T x}}{1 + e^{w^T x}}$$



The sigmoid function

$$p(y | x; \theta)$$

- To classify using regression models we replace the linear function with the sigmoid

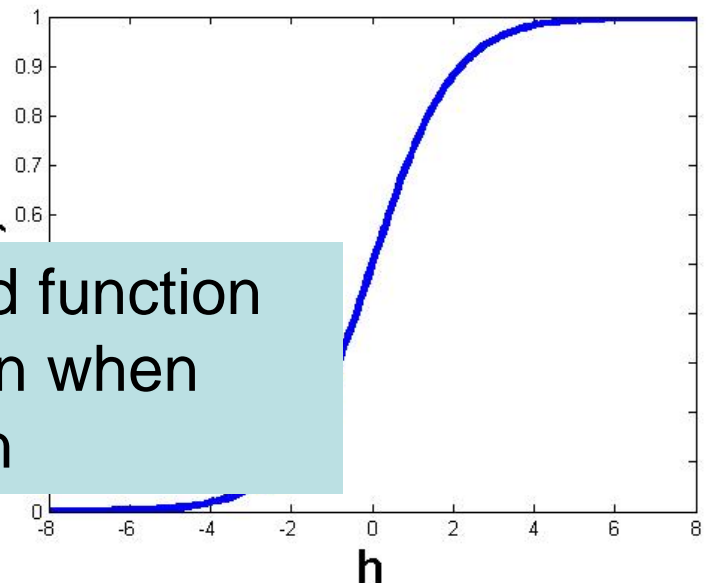
Always between 0 and 1 $\longrightarrow g(h) = \frac{1}{1 + e^{-h}}$

$p(y = 0 | x; \theta)$

We can use the sigmoid function as part of the perception when using it for classification

• Using the binary cla

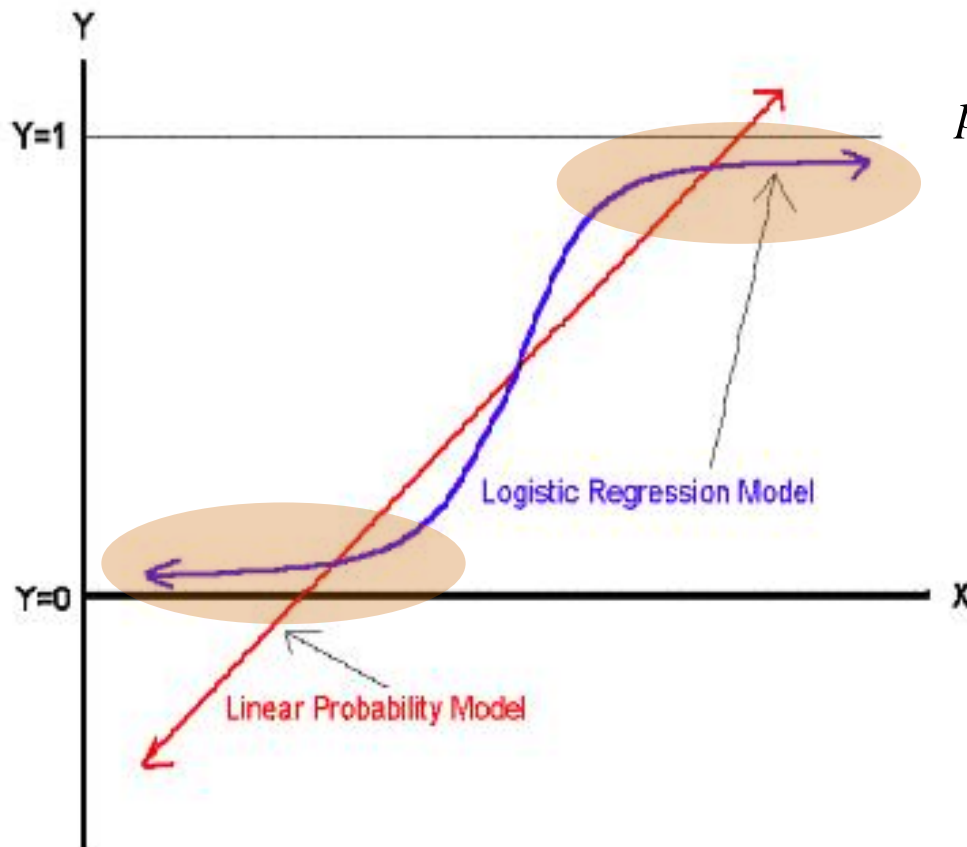
$$p(y = 1 | x; \theta) = 1 - g(w^T x) = \frac{e^{w^T x}}{1 + e^{w^T x}}$$



Logistic regression vs. Linear regression

$$p(y = 0 | x; \theta) = g(w^T x) = \frac{1}{1 + e^{w^T x}}$$

$$p(y = 1 | x; \theta) = 1 - g(w^T x) = \frac{e^{w^T x}}{1 + e^{w^T x}}$$



Non linear regression with NN


$$g(x) = \frac{1}{1 + e^{-x}}$$

- So how do we find the parameters?
- Least squares minimization when using a sigmoid function in a NN:

$$\min \sum_j (y_j - g(w^T x_j))^2$$

Taking the derivative w.r.t. w_i we get:

$$g'(x) = -g(x)(1 - g(x))$$

$$\begin{aligned} & \frac{\partial}{\partial w_i} \sum_j (y_j - g(w^T x_j))^2 \\ &= \sum_j -2(y_j - g(w^T x_j))g(w^T x_j)(1 - g(w^T x_j))x_{j,i} \end{aligned}$$


Deriving $g'(x)$

- Recall that $g(x)$ is the sigmoid function so

$$g(x) = \frac{1}{1 + e^{-x}}$$

- The derivation of $g'(x)$ is below

First, notice $g'(x) = g(x)(1 - g(x))$

Because: $g(x) = \frac{1}{1 + e^{-x}}$ so $g'(x) = \frac{-e^{-x}}{(1 + e^{-x})^2}$

$$= \frac{1 - 1 - e^{-x}}{(1 + e^{-x})^2} = \frac{1}{(1 + e^{-x})^2} - \frac{1}{1 + e^{-x}} = \frac{-1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}} \right) = -g(x)(1 - g(x))$$

New target function for NN

$$g(x) = \frac{1}{1 + e^{-x}}$$


- So how do we find the parameters?
- Least squares minimization when using a sigmoid function in a NN:

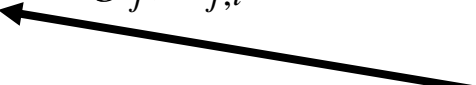
$$\min \sum_j (y_j - g(w^T x_j))^2$$

Taking the derivative w.r.t. w_i we get:

$$\begin{aligned} & \frac{\partial}{\partial w_i} \sum_j (y_j - g(w^T x_j))^2 \\ &= \sum_j 2(y_j - g(w^T x_j))g(w^T x_j)(1 - g(w^T x_j))x_{j,i} \end{aligned}$$

$$\stackrel{\text{def}}{=} \sum_j 2\delta_j g_j(1 - g_j)x_{j,i}$$

$$g'(x) = -g(x)(1 - g(x))$$


$$g_j = g(w^T x_j)$$


Revised algorithm for sigmoid regression

1. Chose λ
2. Start with a guess for \mathbf{w}
3. Compute δ_j for all j
4. For all i set $w_i \leftarrow w_i - \lambda 2 \sum_{j=1}^n \delta_j g_j (1 - g_j) x_{j,i}$
5. If no improvement for $\sum_{j=1}^n (y_j - g(\mathbf{w}^T \mathbf{x}_j))^2$
stop. Otherwise go to step 3

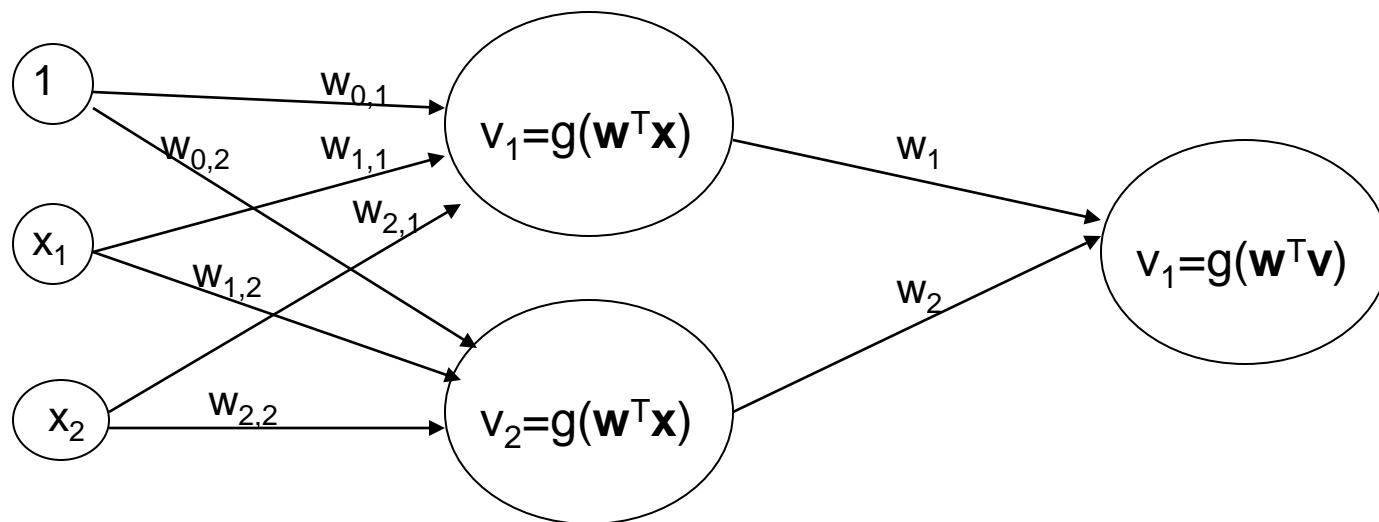
Multilayer neural networks

- So far we discussed networks with one layer.
- But these networks can be extended to combine several layers, increasing the set of functions that can be represented using a NN

Input layer

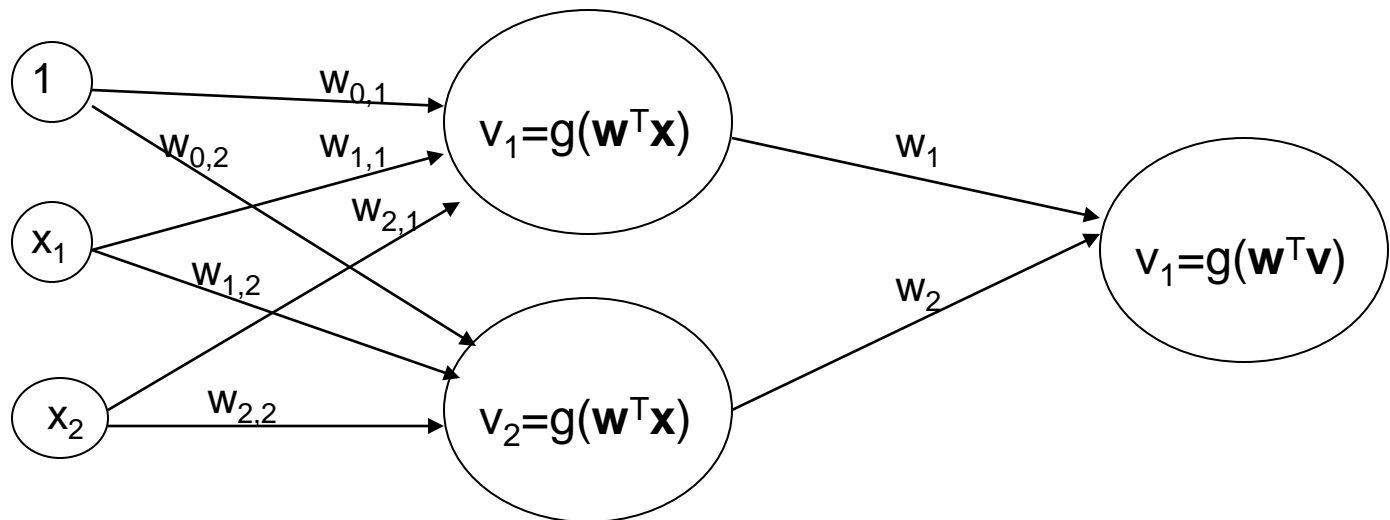
Hidden layer

Output layer



Learning the parameters for multilayer networks

- Gradient descent works by connecting the output to the inputs.
- But how do we use it for a multilayer network?
- We need to account for both, the output weights and the hidden layer weights



Learning the parameters for multilayer networks

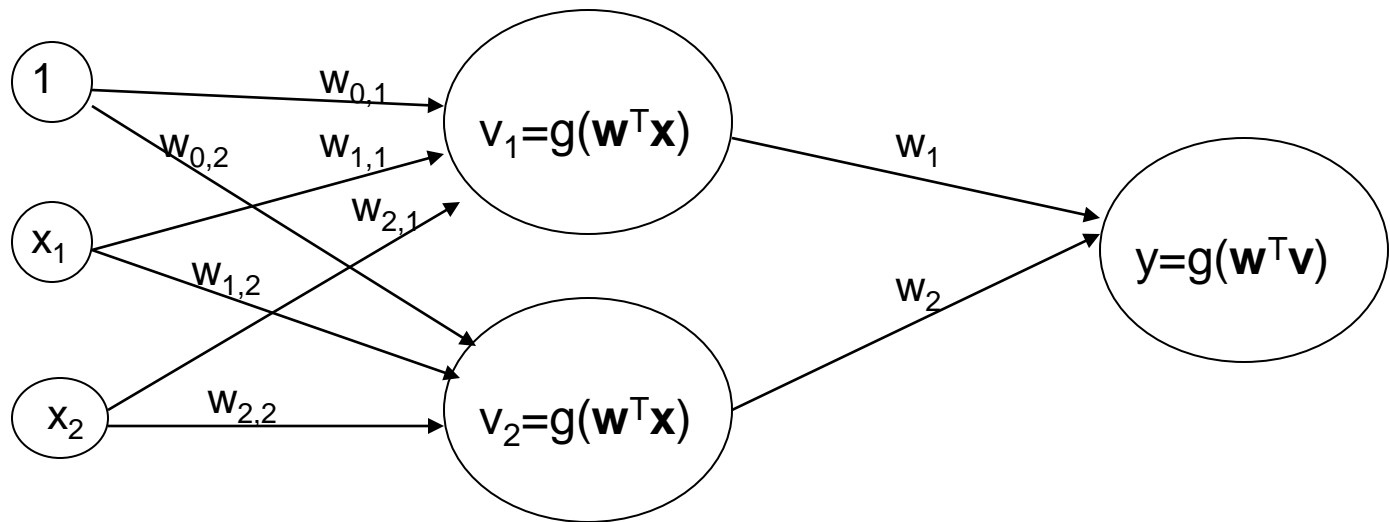
- If we know the values of the internal layer, it is easy to compute the update rule for the output weights w_1 and

w_2 :

$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^n \delta_j g_j (1 - g_j) v_{j,i}$$

$$\delta_j = y_j - g(\mathbf{w}^T \mathbf{v}_j)$$

where

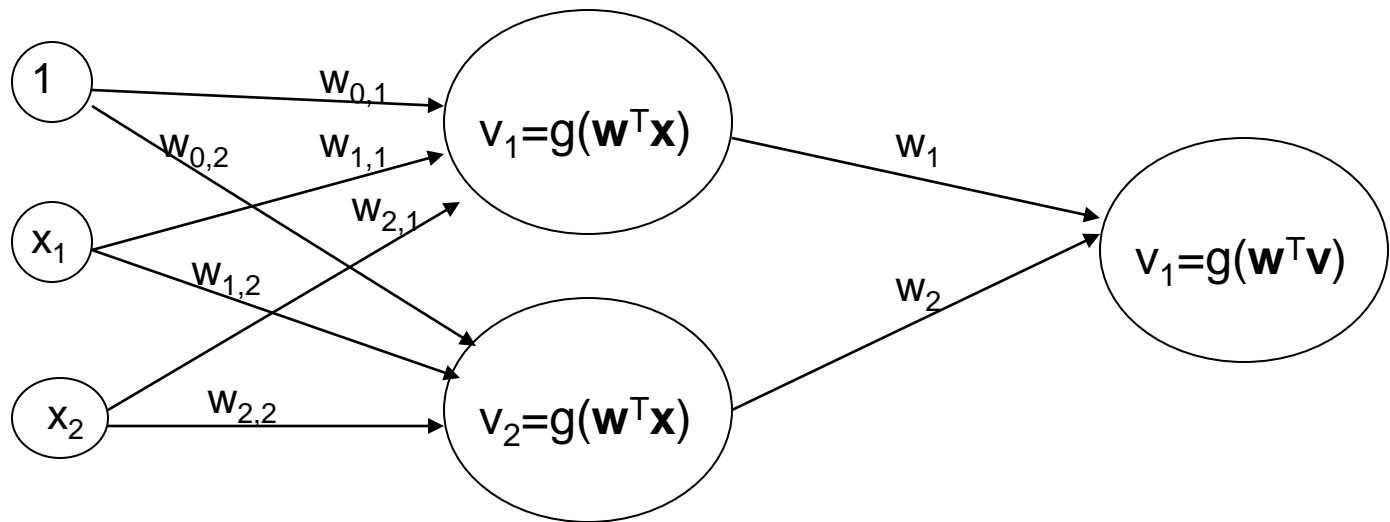


Learning the parameters for multilayer networks

- It's easy to compute the update rule for the output weights w_1 and w_2 :

$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^n \delta_j g_j (1 - g_j) v_{j,i}$$

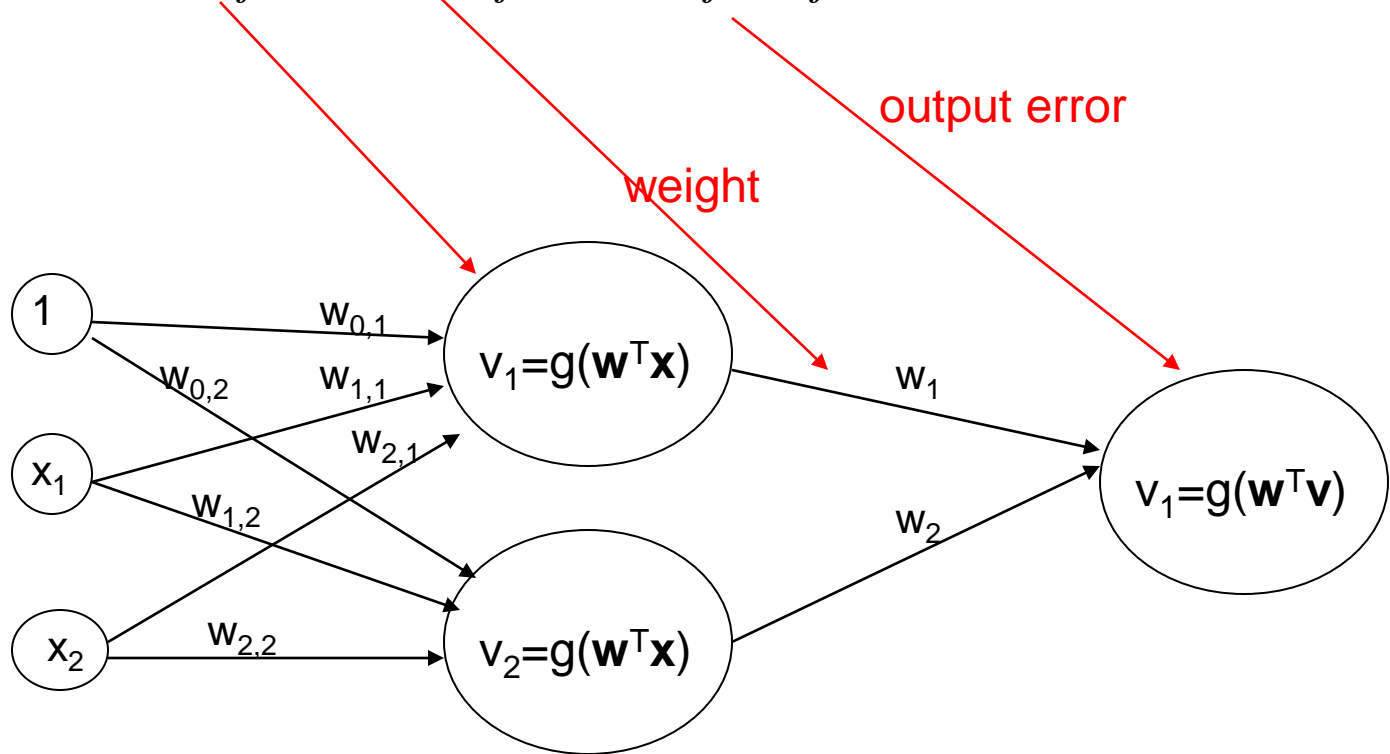
But what is the error associated with each of the hidden layer states?



Backpropagation

- A method for distributing the error among hidden layer states
- Using the error for each of these states we can employ gradient descent to update them
- Set

$$\Delta_{j,i} = w_i \delta_j (1 - g_j) g_j$$



Backpropagation

- A method for distributing the error among hidden layer states
- Using the error for each of these states we can employ gradient descent to update them
- Set

$$\Delta_{j,i} = w_i \delta_j (1 - g_j) g_j$$

- Our update rule changes to:

$$w_{k,i} \leftarrow w_{k,i} + \lambda 2 \sum_{j=1}^n \Delta_{j,i} g_{j,i} (1 - g_{j,i}) x_{j,k}$$

Backpropagation

The correct error term for each hidden state can be determined by taking the partial derivative for each of the weight parameters of the hidden layer w.r.t. the global error function*:

$$Err_j = (y_j - g(\mathbf{w}^T g(\mathbf{w}_i^T \mathbf{x})))^2$$

*See RN book for details (pages 746-747)

Revised algorithm for multilayered neural network

1. Chose λ
2. Start with a guess for \mathbf{w} , \mathbf{w}_i
3. Compute values $v_{i,j}$ for all hidden layer states i and inputs j
4. Compute δ_j for all j : $\delta_j = y_j - g(\mathbf{w}^T \mathbf{v}_j)$
5. Compute $\Delta_{j,i}$
6. For all i set
$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^n \delta_j g_j (1 - g_j) v_{j,i}$$
7. For all k and i set
$$w_{k,i} \leftarrow w_{k,i} + \lambda 2 \sum_{j=1}^n \Delta_{j,i} g_{j,i} (1 - g_{j,i}) x_{j,k}$$
8. If no improvement for $\sum_{j=1}^n \delta_j^2 + \sum_{i=1}^s \Delta_{j,i}^2$ stop. Otherwise go to step 3

Examples

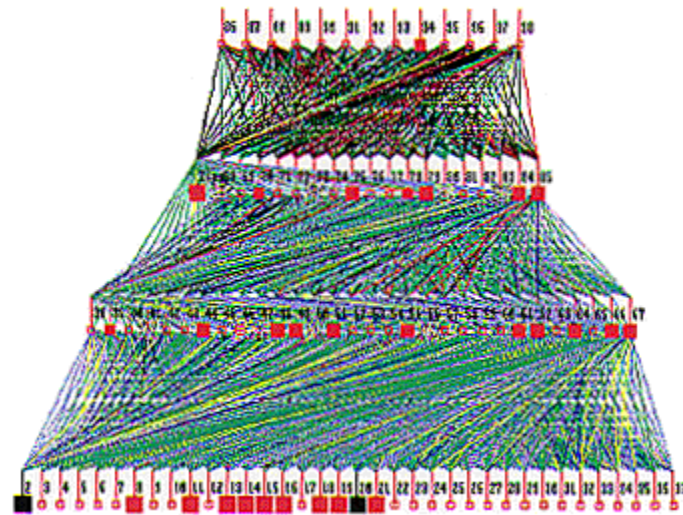


Figure 1: Feedforward ANN designed and **tested** for prediction of tactical air combat maneuvers.

Scientists See Promise in Deep-Learning Programs




Hao Zhang/The New York Times


A voice recognition program translated a speech given by Richard F. Rashid, Microsoft's top scientist, into Mandarin Chinese.

By JOHN MARKOFF

Published: November 23, 2012

Using an artificial intelligence technique inspired by theories about how the brain recognizes patterns, technology companies are reporting startling gains in fields as diverse as computer vision,

 FACEBOOK

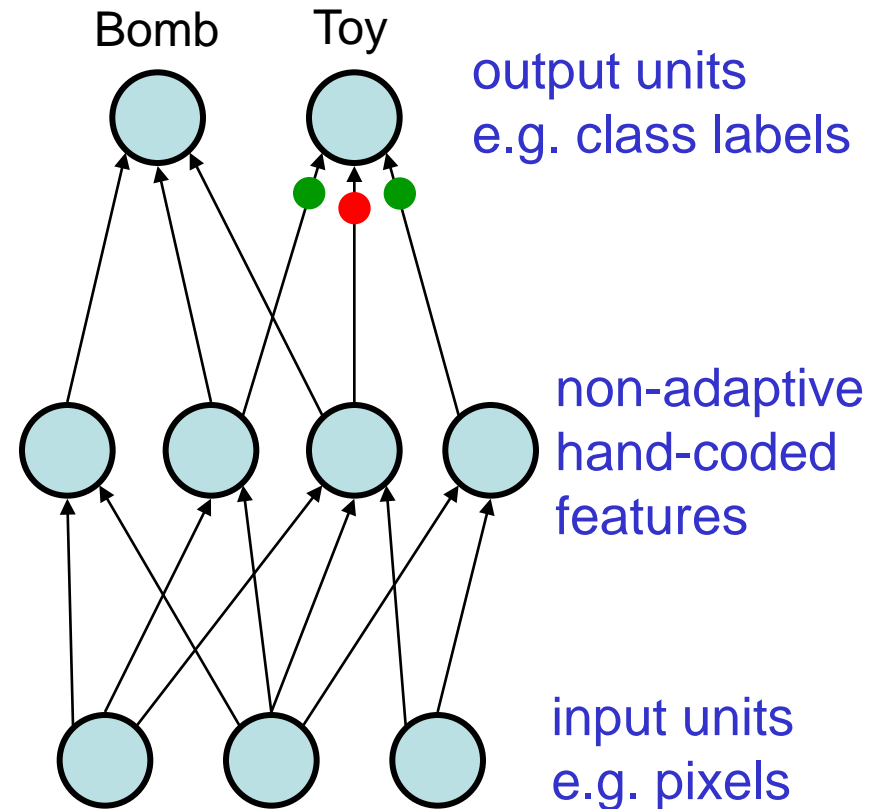
 TWITTER

 GOOGLE+

Historical background:

First generation neural networks

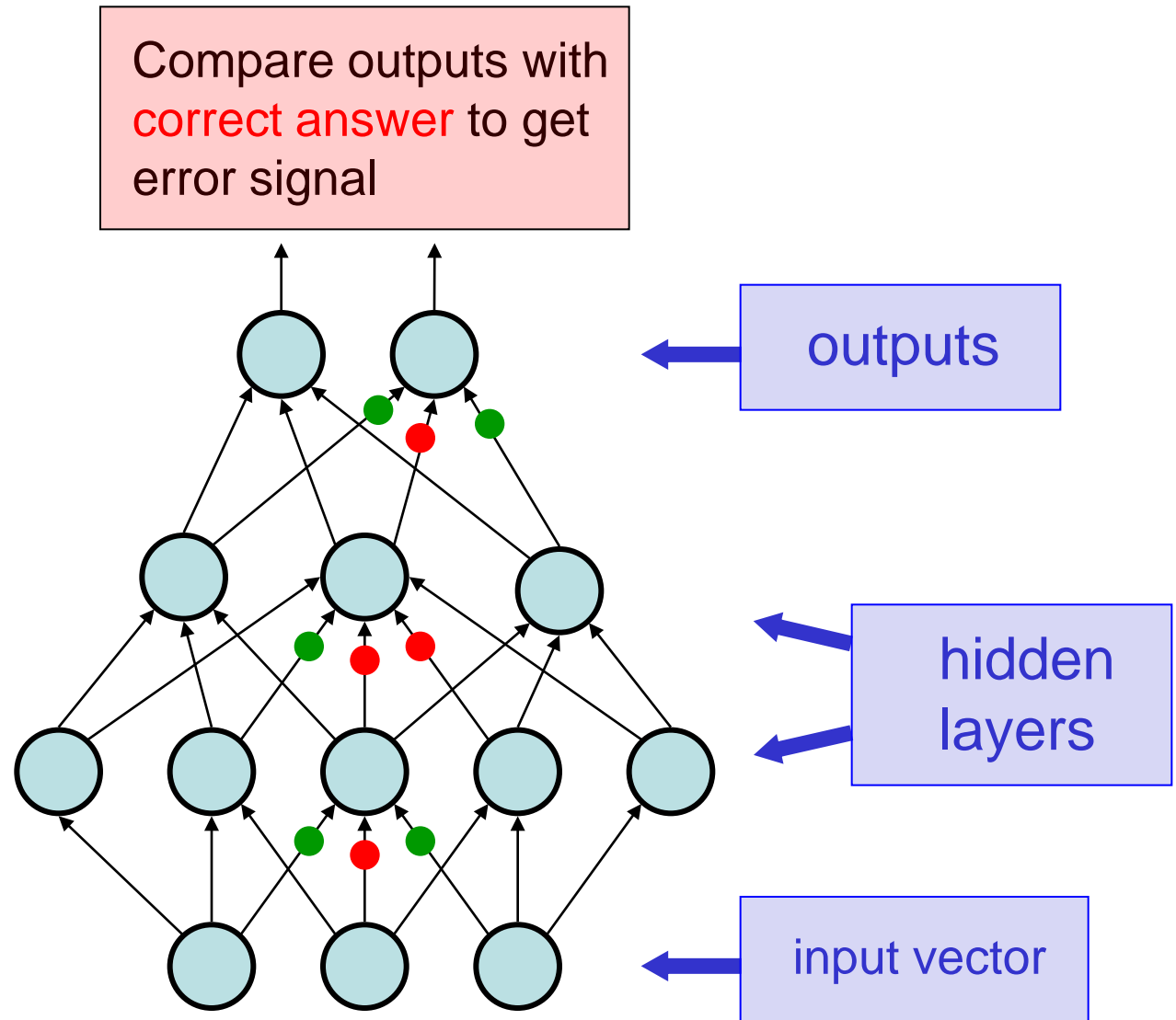
- Perceptrons (~1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
 - There was a neat learning algorithm for adjusting the weights.
 - But perceptrons are fundamentally limited in what they can learn to do.



Sketch of a typical perceptron from the 1960's

Second generation neural networks (~1985)

Back-propagate
error signal to
get derivatives
for learning



What is wrong with back-propagation?

- It requires labeled training data.
 - Almost all data is unlabeled.
- The learning time does not scale well
 - It is very slow in networks with multiple hidden layers.
- It can get stuck in poor local optima.

Overcoming the limitations of back-propagation

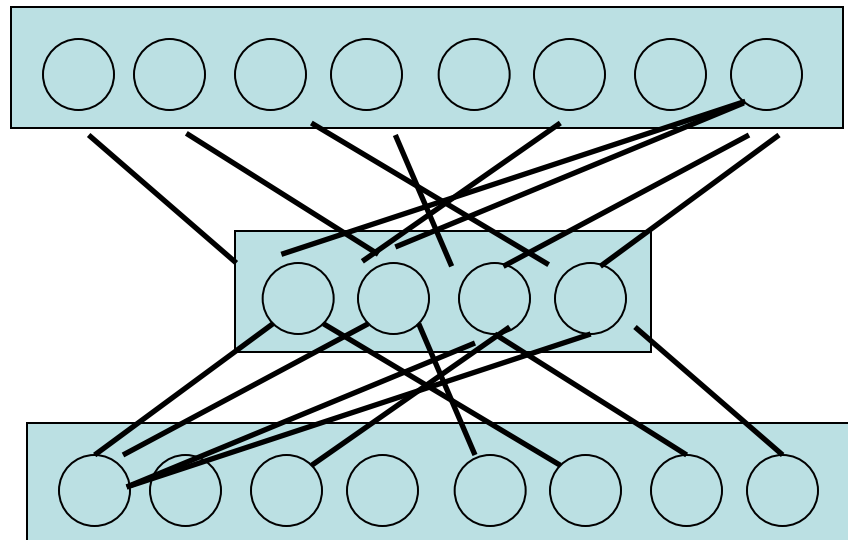
- Keep the efficiency and simplicity of using a gradient method for adjusting the weights, but use it for modeling the structure of the sensory input.
 - Iteratively learn the different layers.
 - Adjust the weights to maximize the probability that a generative model would have produced the sensory input.
 - Learn $p(\text{image})$ not $p(\text{label} \mid \text{image})$ for the lower layers.

Iterative learning of layers

Reconstruction

Hidden

Input



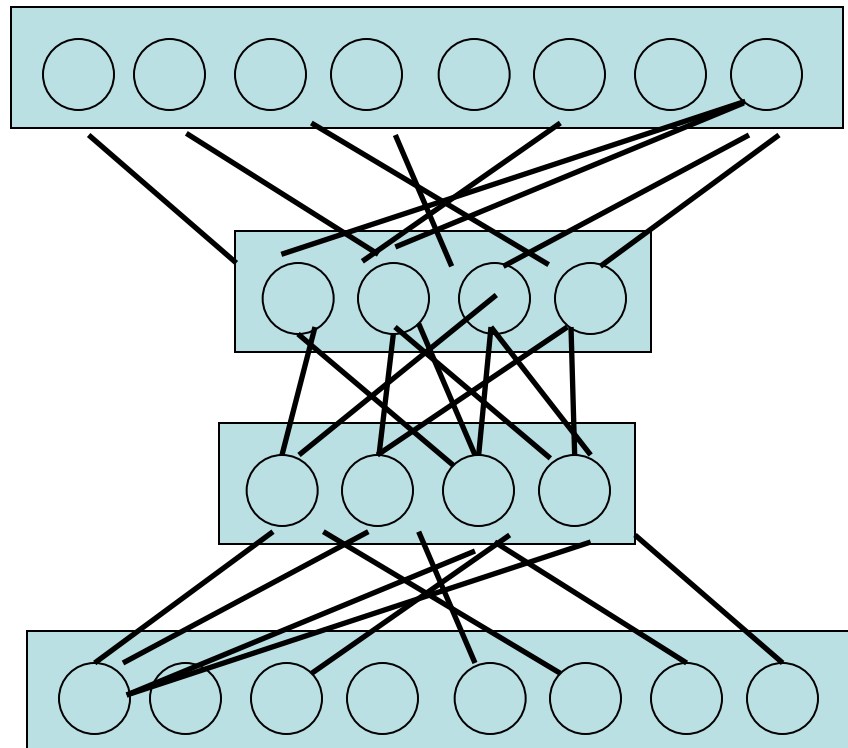
Iterative learning of layers

Reconstruction

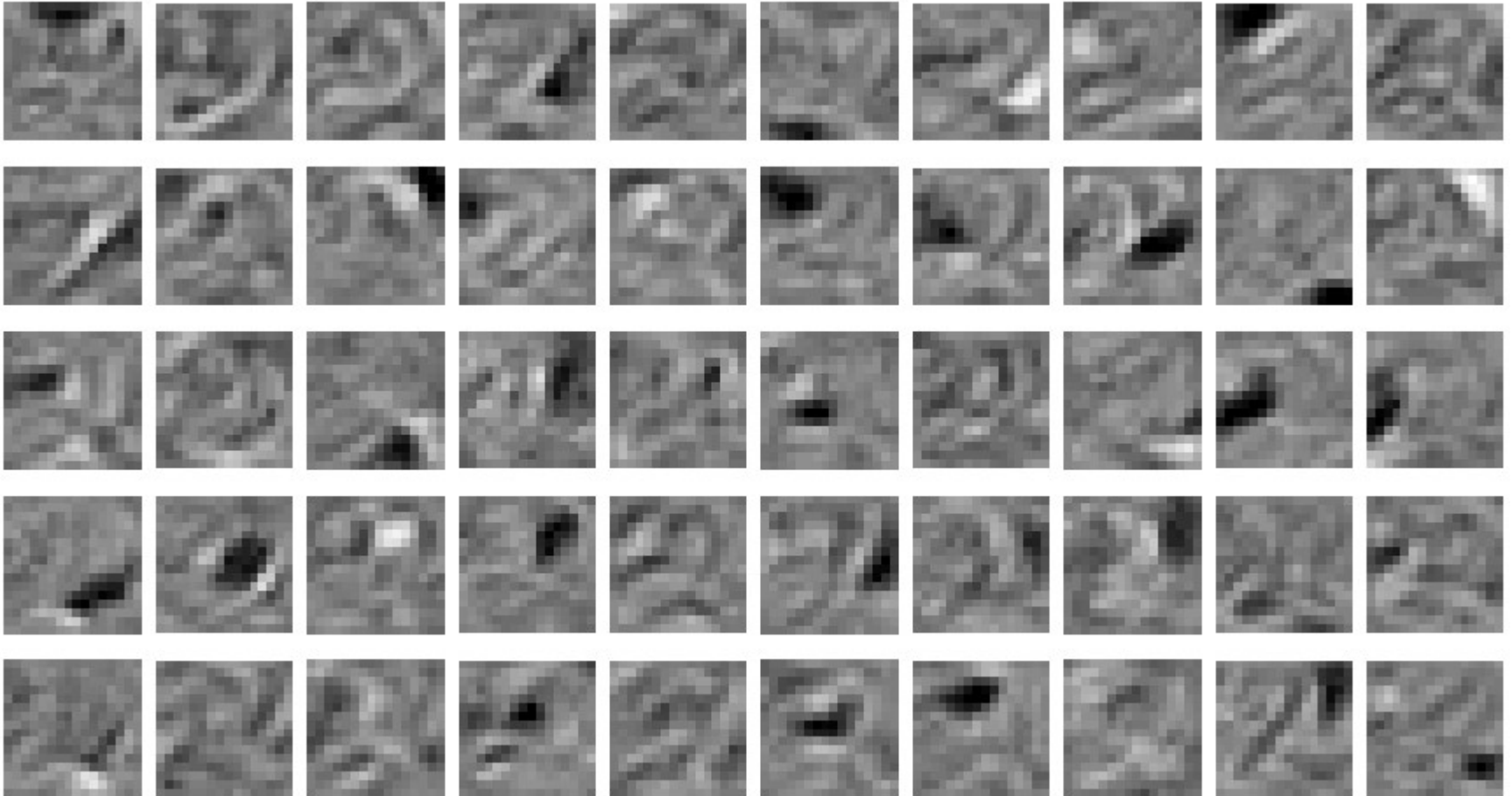
Hidden

Hidden

Input

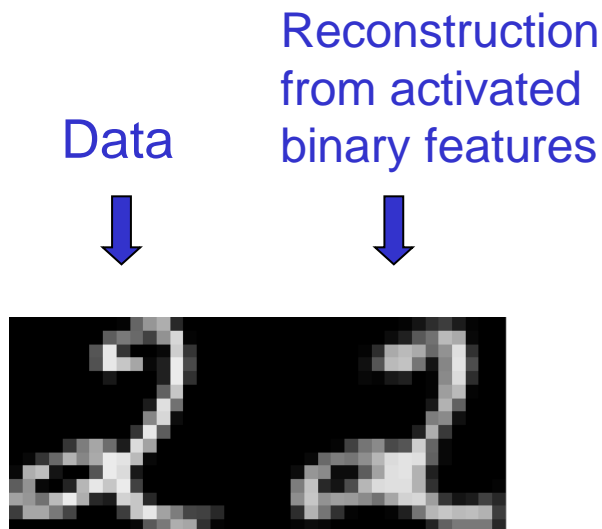


The final 50 \times 256 weights

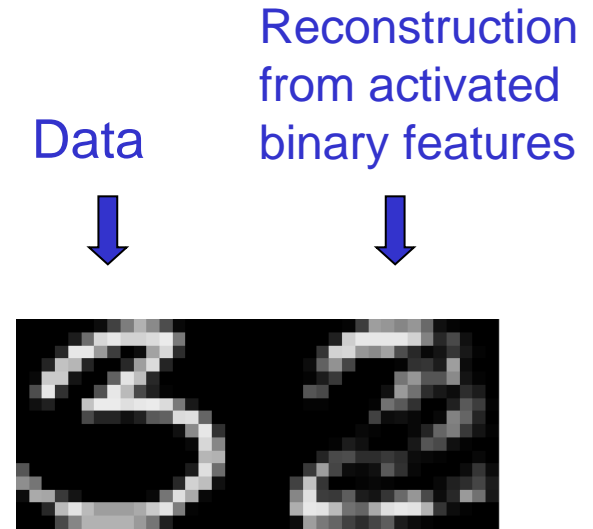


Each neuron grabs a different feature.

How well can we reconstruct the digit images from the binary feature activations?



New test images from the digit class that the model was trained on



Images from an unfamiliar digit class
(the network tries to see every image as a 2)

Training a deep network

(the main reason RBM's are interesting)

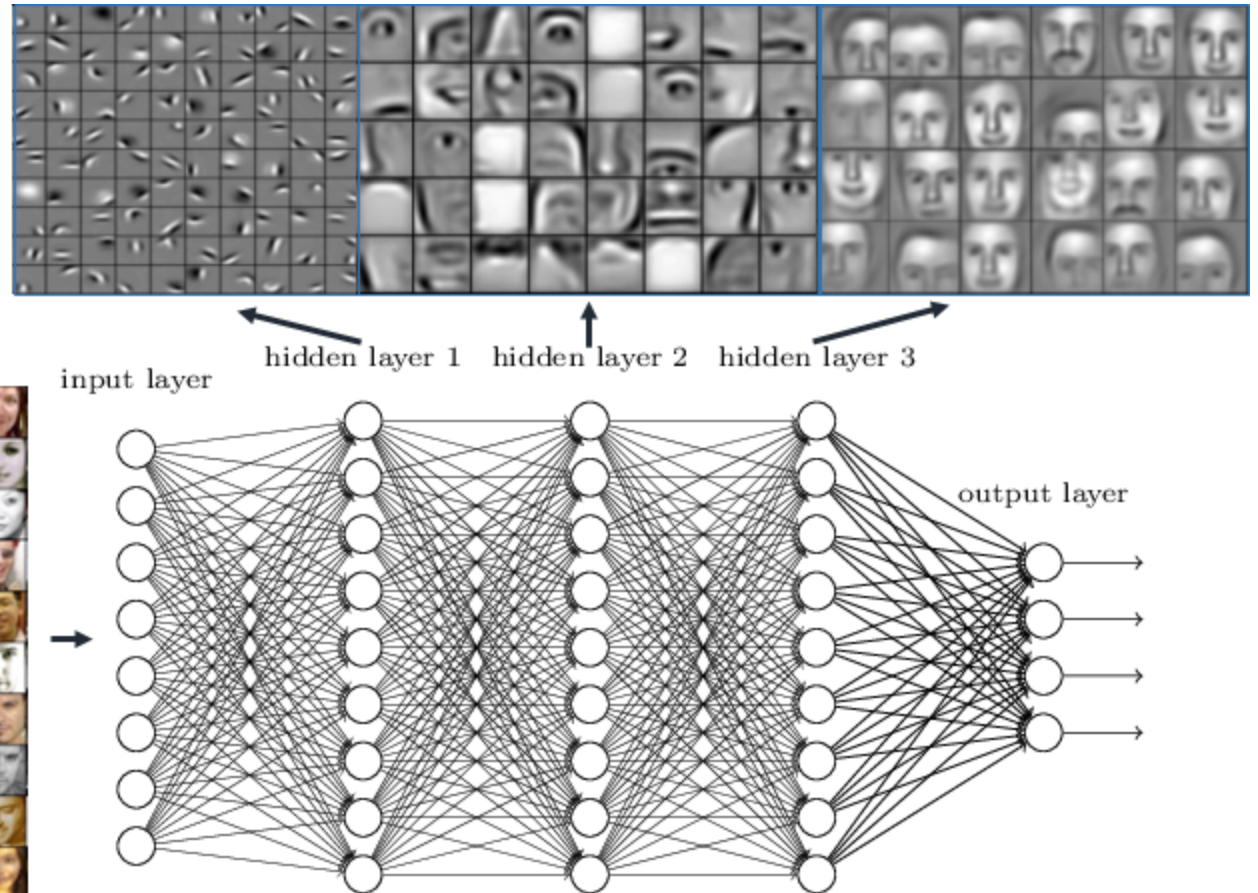
- First train a layer of features that receive input directly from the pixels.
- Then treat the activations of the trained features as if they were pixels and learn features of features in a second hidden layer.
- It can be proved that each time we add another layer of features we improve a variational lower bound on the log probability of the training data.
 - The proof is slightly complicated.
 - But it is based on a neat equivalence between an RBM and a deep directed model (described later)

Samples generated by letting the associative memory run with one label clamped. There are 1000 iterations of alternating Gibbs sampling between samples.

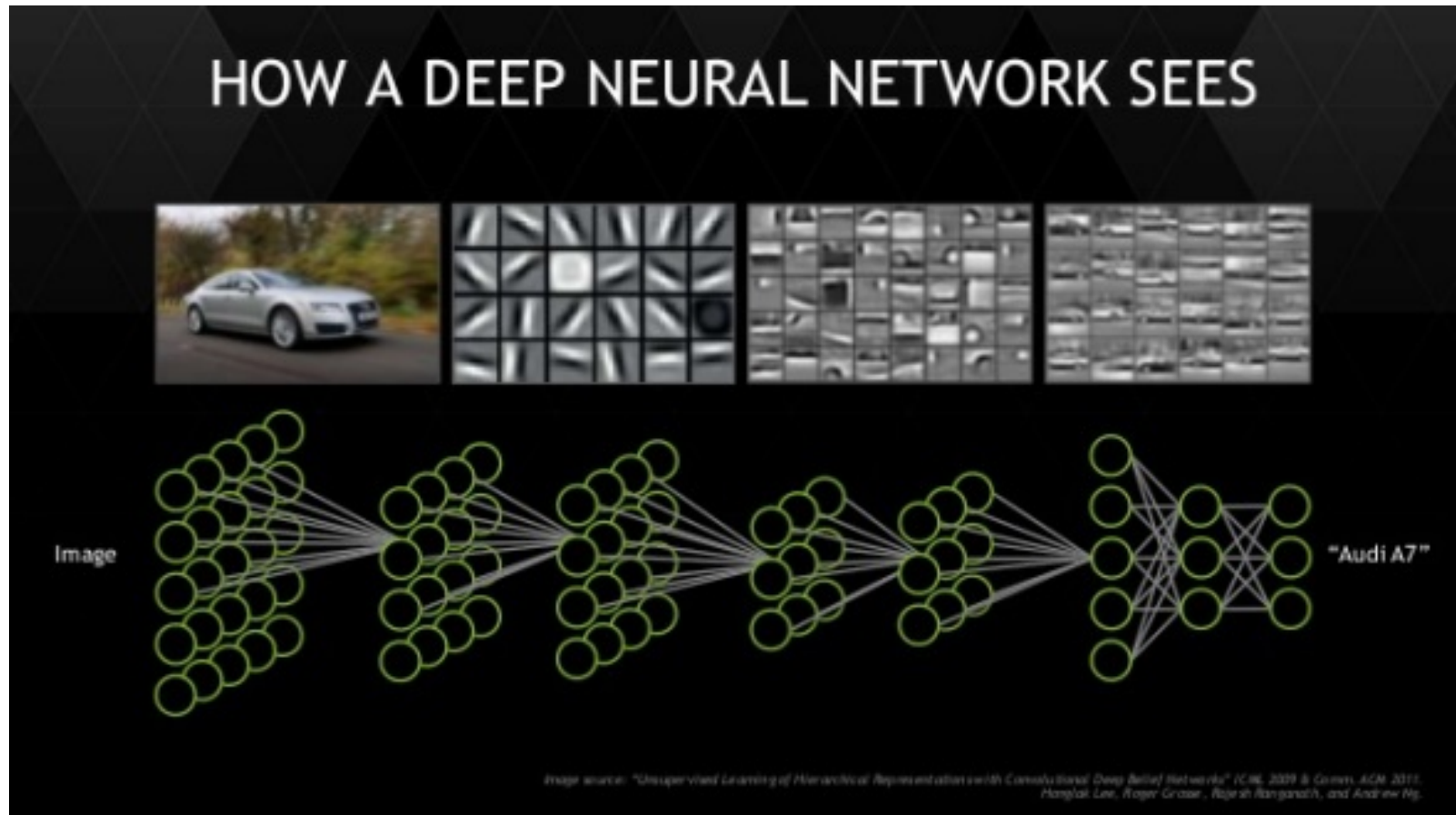


Features learned

Deep neural networks learn hierarchical feature representations



Features learned



What you should know

- Linear regression
 - Solving a linear regression problem
- Gradient descent
- Perceptrons
 - Sigmoid functions for classification
- Multilayered neural networks
 - Backpropagation

Deriving $g'(x)$

- Recall that $g(x)$ is the sigmoid function so

$$g(x) = \frac{1}{1 + e^{-x}}$$

- The derivation of $g'(x)$ is below

First, notice $g'(x) = g(x)(1 - g(x))$

Because: $g(x) = \frac{1}{1 + e^{-x}}$ so $g'(x) = \frac{-e^{-x}}{(1 + e^{-x})^2}$

$$= \frac{1 - 1 - e^{-x}}{(1 + e^{-x})^2} = \frac{1}{(1 + e^{-x})^2} - \frac{1}{1 + e^{-x}} = \frac{-1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}} \right) = -g(x)(1 - g(x))$$

The Energy of a joint configuration

binary state of
visible unit i

binary state of
hidden unit j

$$E(v, h) = - \sum_{i, j} v_i h_j w_{ij}$$

Energy with configuration
 v on the visible units and
 h on the hidden units

weight between
units i and j


$$-\frac{\partial E(v, h)}{\partial w_{ij}} = v_i h_j$$

Using energies to define probabilities

- The probability of a joint configuration over both visible and hidden units depends on the energy of that joint configuration compared with the energy of all other joint configurations.
- The probability of a configuration of the visible units is the sum of the probabilities of all the joint configurations that contain it.

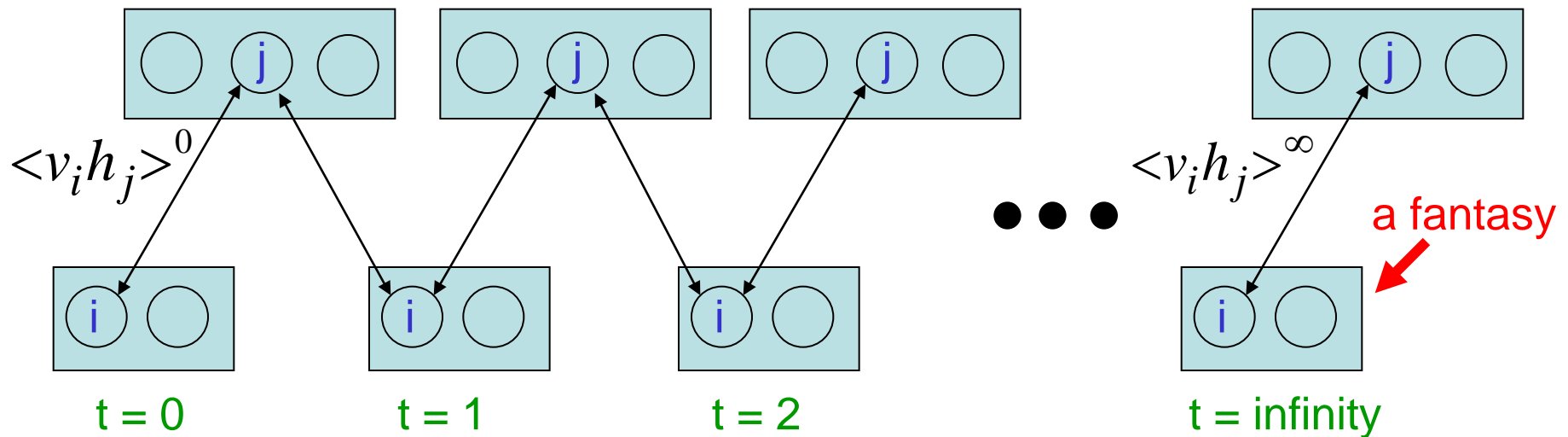
$$p(v, h) = \frac{e^{-E(v, h)}}{\sum_{u, g} e^{-E(u, g)}}$$

partition
function



$$p(v) = \frac{\sum_h e^{-E(v, h)}}{\sum_{u, g} e^{-E(u, g)}}$$

A picture of the maximum likelihood learning algorithm for an RBM

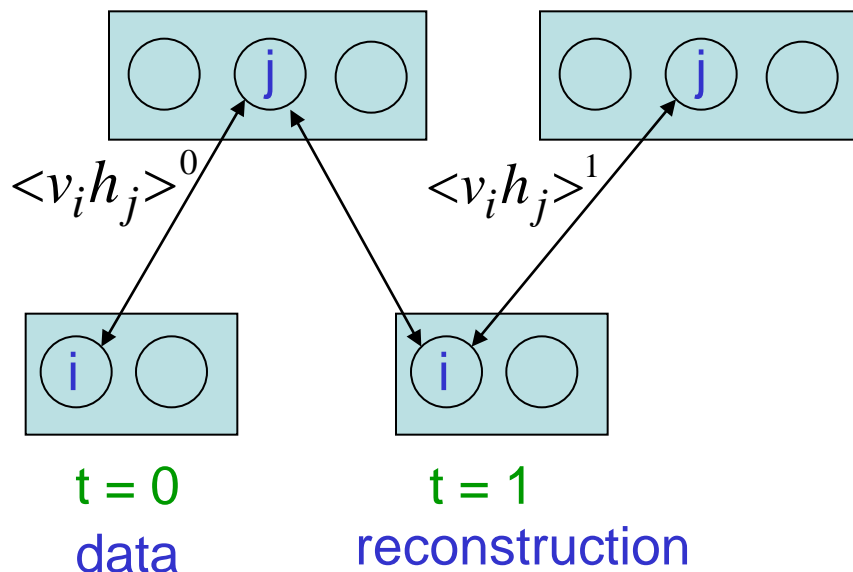


Start with a training vector on the visible units.

Then alternate between updating all the hidden units in parallel and updating all the visible units in parallel.

$$\frac{\partial \log p(v)}{\partial w_{ij}} = \langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^\infty$$

A quick way to learn an RBM



Start with a training vector on the visible units.

Update all the hidden units in parallel

Update the all the visible units in parallel to get a “reconstruction”.

Update the hidden units again.

$$\Delta w_{ij} = \varepsilon (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1)$$

This is not following the gradient of the log likelihood. But it works well. It is approximately following the gradient of another objective function (Carreira-Perpinan & Hinton, 2005).

What you should know

- Linear regression
 - Solving a linear regression problem
- Gradient descent
- Perceptrons
 - Sigmoid functions for classification
- Multilayered neural networks
 - Backpropagation