10-601
Machine Learning

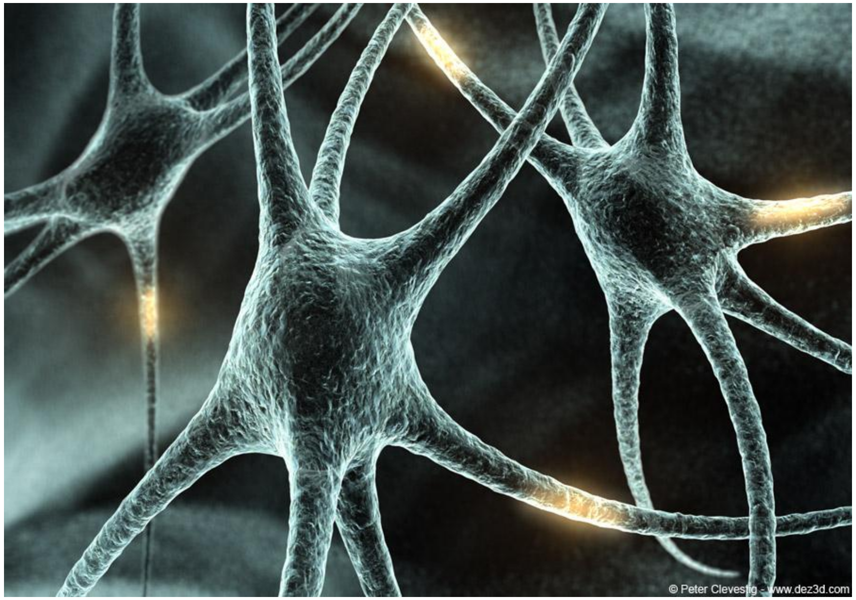
Neural Networks (NN)

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 little known), 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



What can we do with NN?

- Classification
- Regression

  Input: Real valued variables

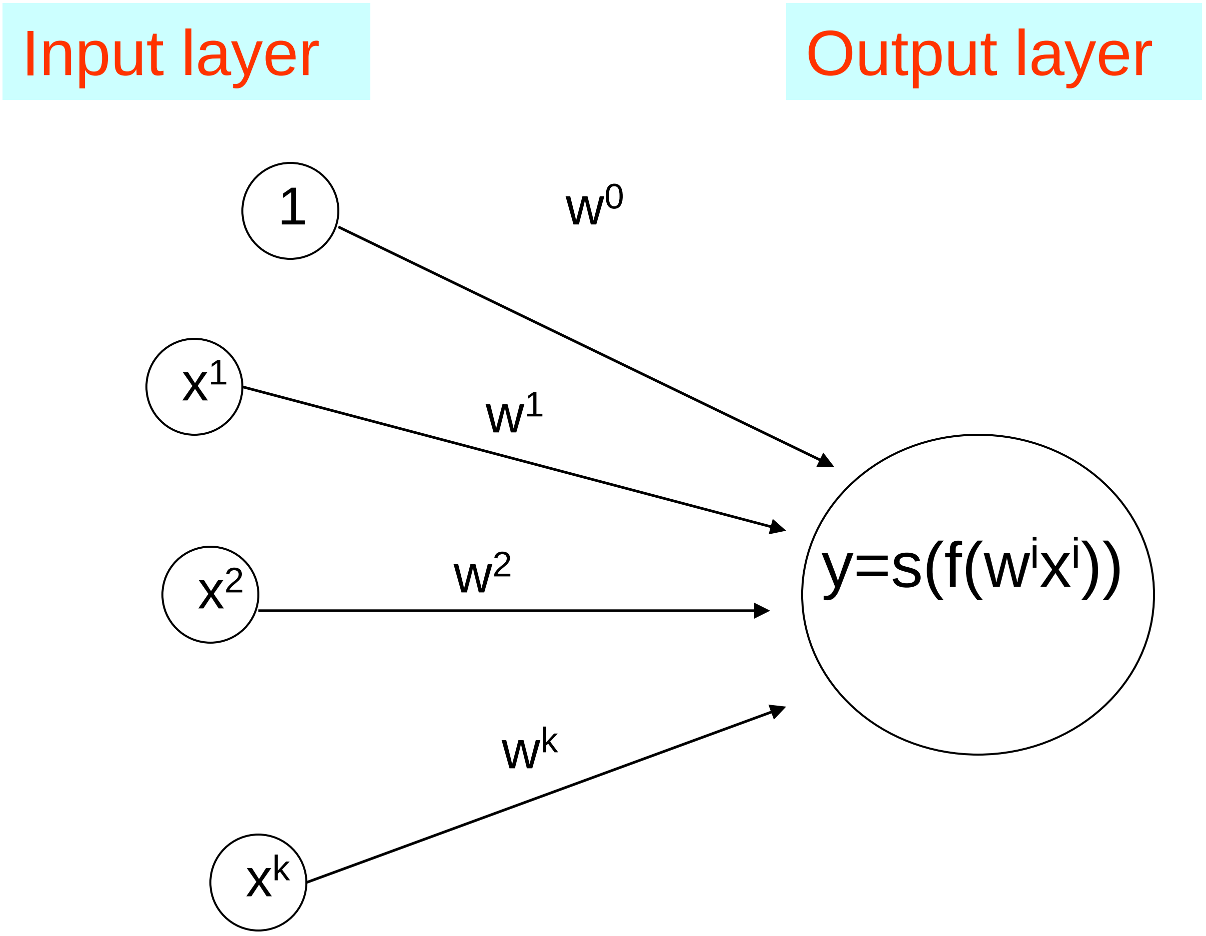
  Output: One or more real values
- Examples:

  - Predict the price of Google's stock from Microsoft's stock price

  - Predict distance to obstacle from various sensors

Back to NN: Preceptron

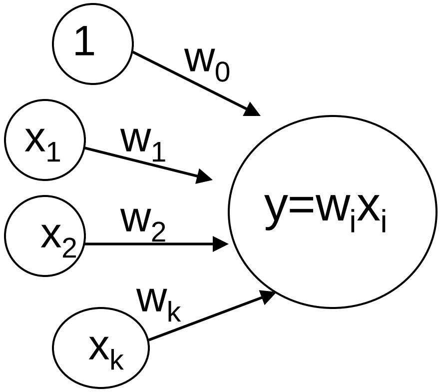
- The basic processing unit of a neural net



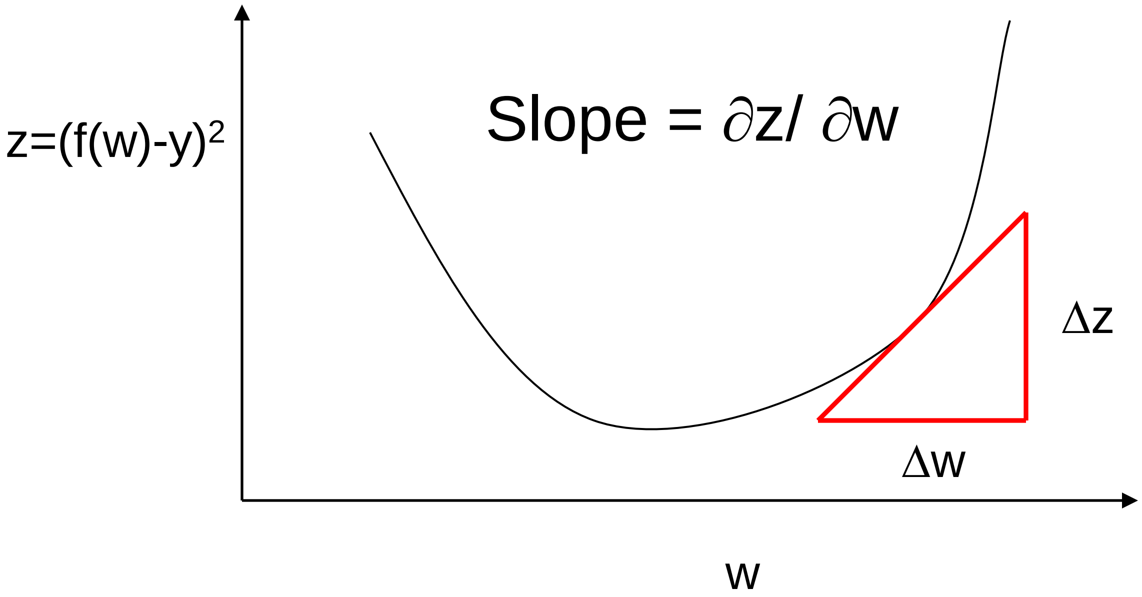
Linear regression

- 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?

  \- We will discuss this later, for now lets focus on the solution …



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

Gradient descent for linear regression

- Taking the derivative w.r.t. to each w^i for a sample X:

$$\frac{\partial}{\partial w^j}\left(y-\sum_k w^k x^k\right)^2 = -2x^j(y-\sum_k w^k x^k)$$

- And if we have n measurements then

$$\frac{\partial}{\partial w^j}\sum_{i=1}^{n}(y_i - \mathrm{w}^T \mathrm{X}_i)^2 = -2\sum_{i=1}^{n} x_i^j (y_i - \mathrm{w}^T \mathrm{X}_i)$$

where x^j_i is the j'th value of the i'th input vector

Gradient descent for linear regression

- If we have n measurements then

$$\frac{\partial}{\partial w^j}\sum_{i=1}^{n}(y_i - \mathrm{w}^T \mathrm{X}_i)^2 = -2\sum_{j=1}^{n} x_i^j (y_i - \mathrm{w}^T \mathrm{X}_i)$$

- Set $\delta_i = (y_i - \mathrm{w}^T \mathrm{X}_i)$

- Then our update rule can be written as

$$w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} x_i^j \delta_i$$

Gradient descent algorithm for linear regression

1. Chose λ
2. Start with a guess for **w**
3. Compute δ_i for all i
4. For all i set $$w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} x_i^j \delta_i$$
5. If no improvement for $$\sum_{i=1}^{n} (y_i - \mathrm{w}^T \mathrm{X}_i)^2$$

   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

We can also use the sigmoid function in NN

Instead of using the probabilistic maximum likelihood target function, we return to least squares when using the sigmoid in NN

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

$$\min \sum_i (y_i - g(w^T X_i))^2$$

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

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

$$\frac{\partial}{\partial w^j} \sum_i (y_i - g(w^T X_i))^2$$

$$= \sum_i 2(y_i - g(w^T X_i)) g(w^T X_i)(1-g(w^T X_i)) X_i^j$$

$$\stackrel{def}{=} \sum_i 2\delta_i g_i (1-g_i) X_i^j$$

$$g_i = g(w^T X_i)$$

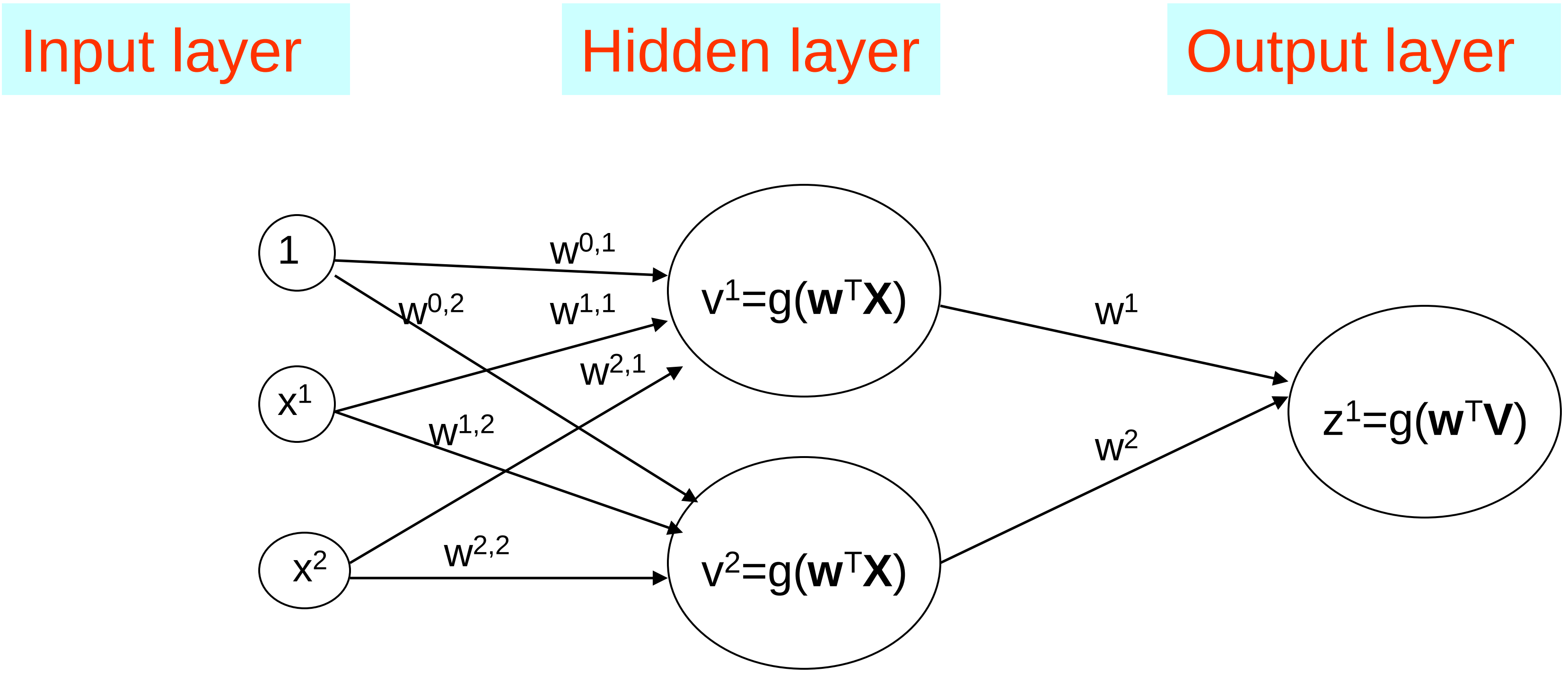
Revised algorithm for sigmoid regression

1. Chose λ
2. Start with a guess for **w**
3. Compute δ_i for all i
4. For all i set $w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} \delta_i g_i (1 - g_i) x_i^j$
5. If no improvement for $\sum_{i=1}^{n} (y_i - g(\mathrm{w}^T \mathrm{X}_i))^2$

   stop. Otherwise go to step 3

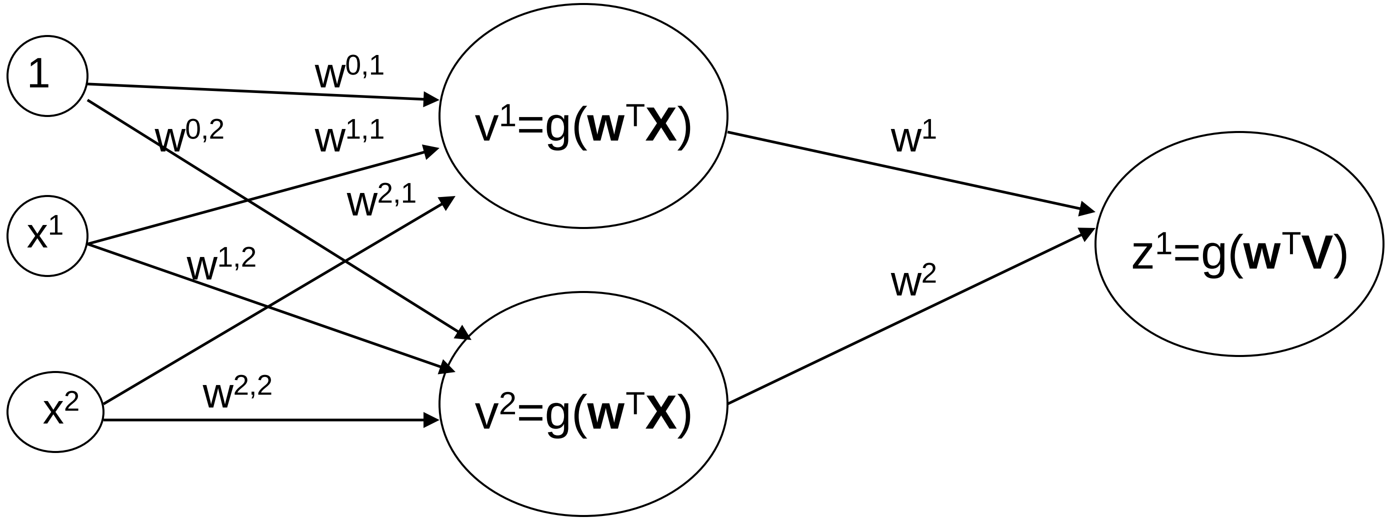
Multilayer neural networks

- 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



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



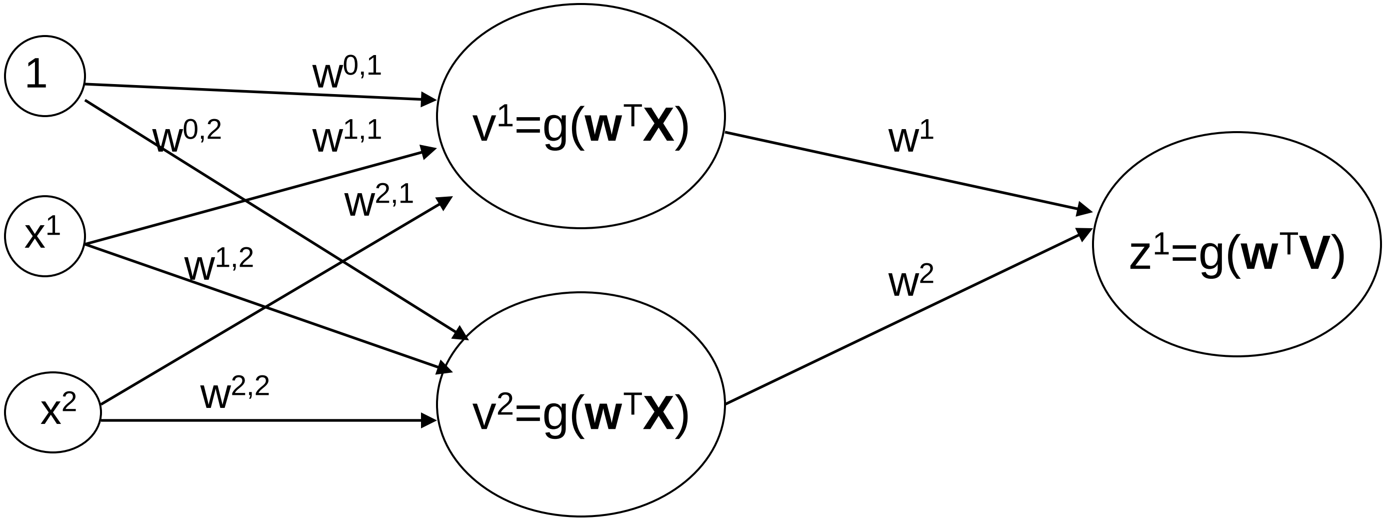
Learning the parameters for multilayer networks

- Its easy to compute the update rule for the output weights w_1 and w_2:

$$w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} \delta_i g_i (1 - g_i) v_i^j$$

V^j for the i'th input

where $\delta_i = y_i - g(\mathrm{w}^T \mathrm{v}_i)$

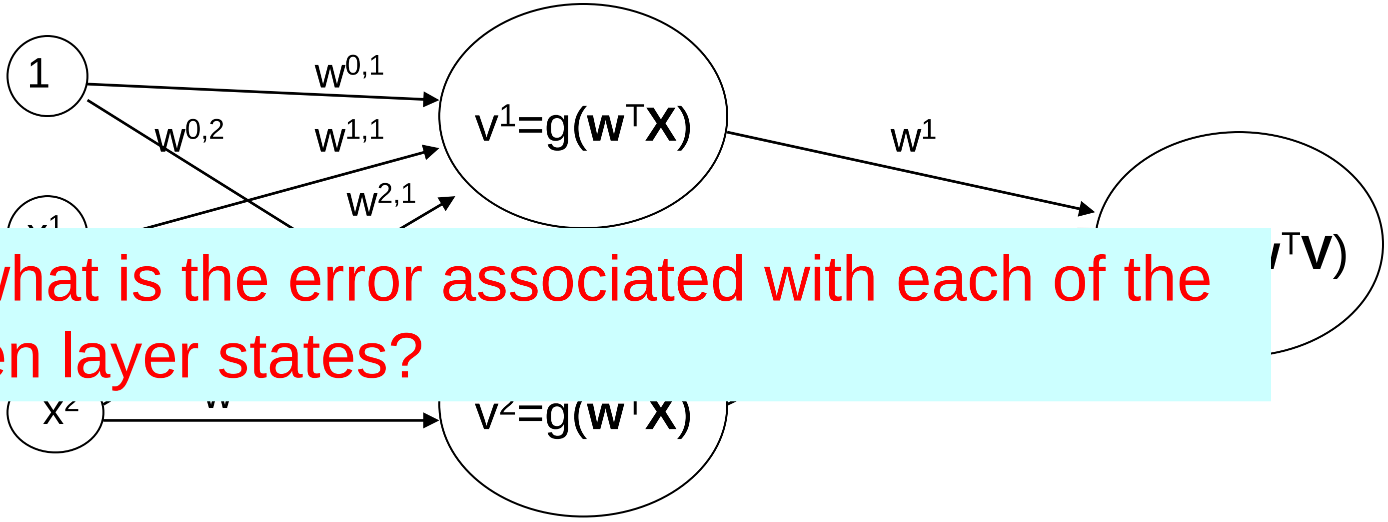


Learning the parameters for multilayer networks

- Its easy to compute the update rule for the output weights w_1 and w_2:

$$w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} \delta_i g_i (1 - g_i) v_i^j$$

where $$\delta_i = y_i - g(\mathrm{w}^T \mathrm{v}_i)$$

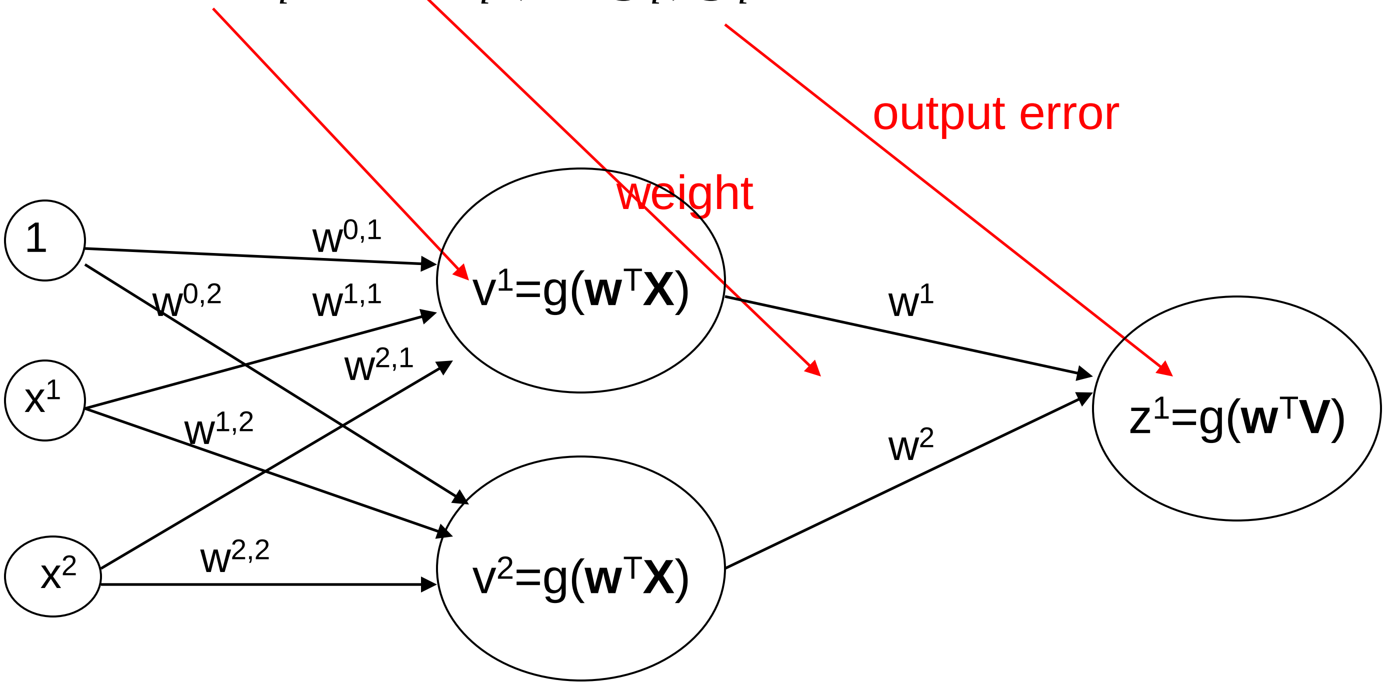


But what is the error associated with each of the hidden layer states?

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

$$\Delta_i^j = w^j \delta_i (1 - g_i) g_i$$



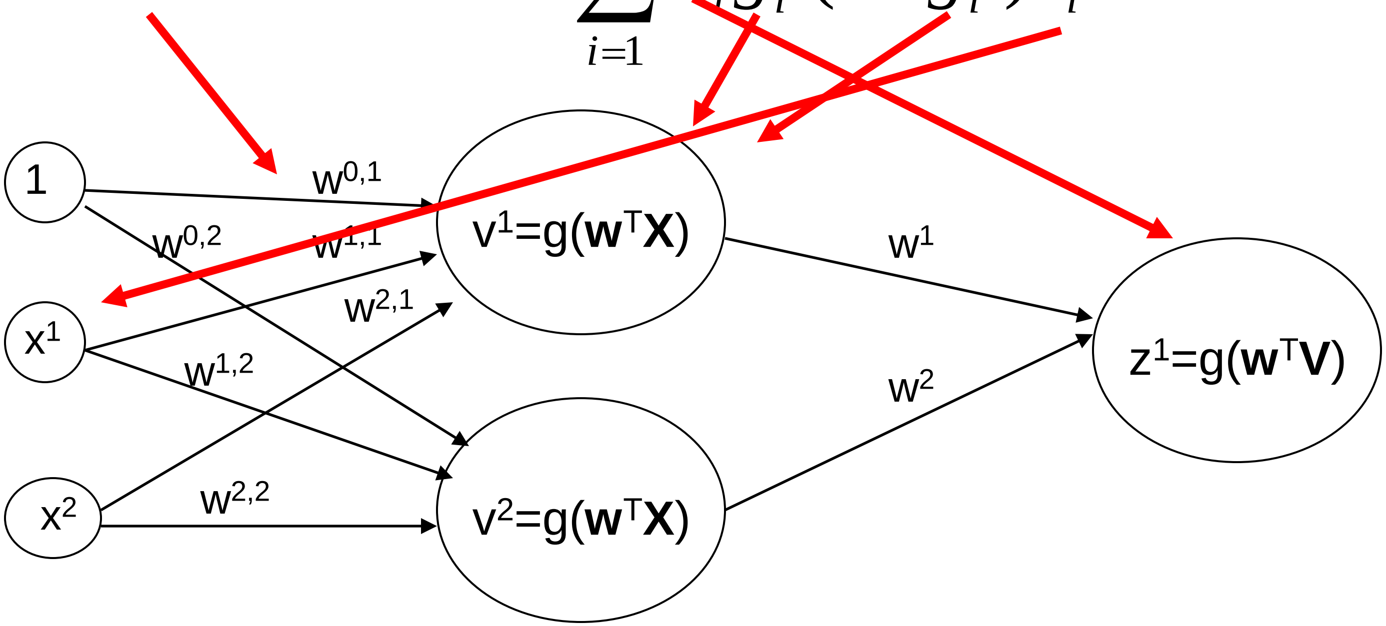
Backpropagation

- 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

$$\Delta_i^j = w^j \delta_i (1 - g_i) g_i$$

- Our update rule changes to:

$$w^{k,j} \leftarrow w^{k,j} + \lambda 2 \sum_{i=1}^{n} \Delta_i^j g_i^j (1 - g_i^j) x_i^k$$



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*:

$$Err_i = (y_i - g(\mathrm{w}^T g(\mathrm{w}^{\mathrm{j}^{\mathrm{T}}} \mathrm{x}))^2$$

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

Revised algorithm for multilayered neural network

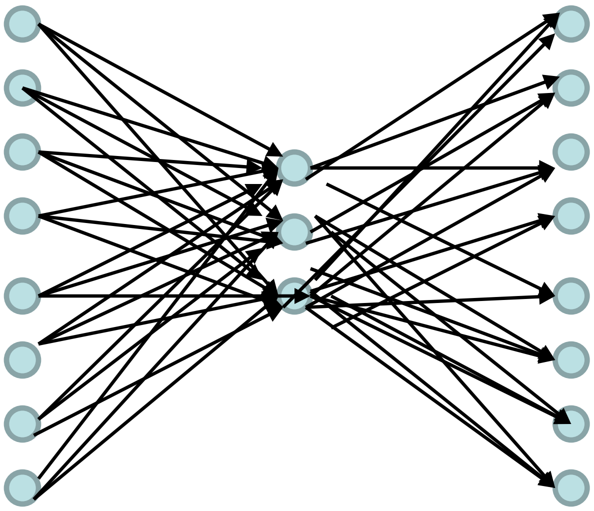
1. Chose λ
2. Start with a guess for **w, w^j**
3. Compute values v_i^j for all hidden layer states j and inputs i
4. Compute δ_i for all i: $\delta_i = y_i - g(\mathrm{w}^T \mathrm{v}_i)$
5. Compute $\Delta_i^j = w^j \delta_i (1 - g_i) g_i$ for all i and j
6. For all j set $$w^j \leftarrow w^j + \lambda 2 \sum_{i=1}^{n} \delta_i g_i (1 - g_i) v_i^j$$
7. For all k and j set $$w^{k,j} \leftarrow w^{k,j} + \lambda 2 \sum_{i=1}^{n} \Delta_i^j g_i^j (1 - g_i^j) x_i^k$$
8. If no improvement for $\sum_{i=1}^{n} \delta_i^2 + \sum_{i=1}^{s} (\Delta_i^j)^2$ stop. Otherwise go to step 3

Neural network encoding

- Assume we would like to learn the following (trivial?) output function:

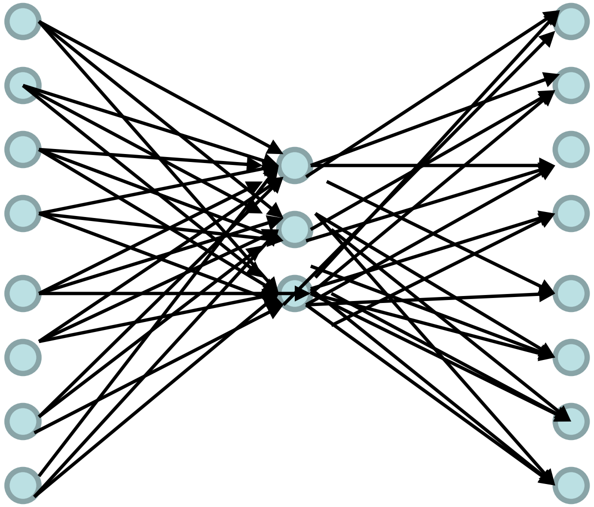
Input	Output
00000001	00000001
00000010	00000010
00000101	00000100
00001000	00001000
00010000	00010000
00100000	00100000
01000000	01000000
10000000	10000000

- Using the following network:



- Can this be done?

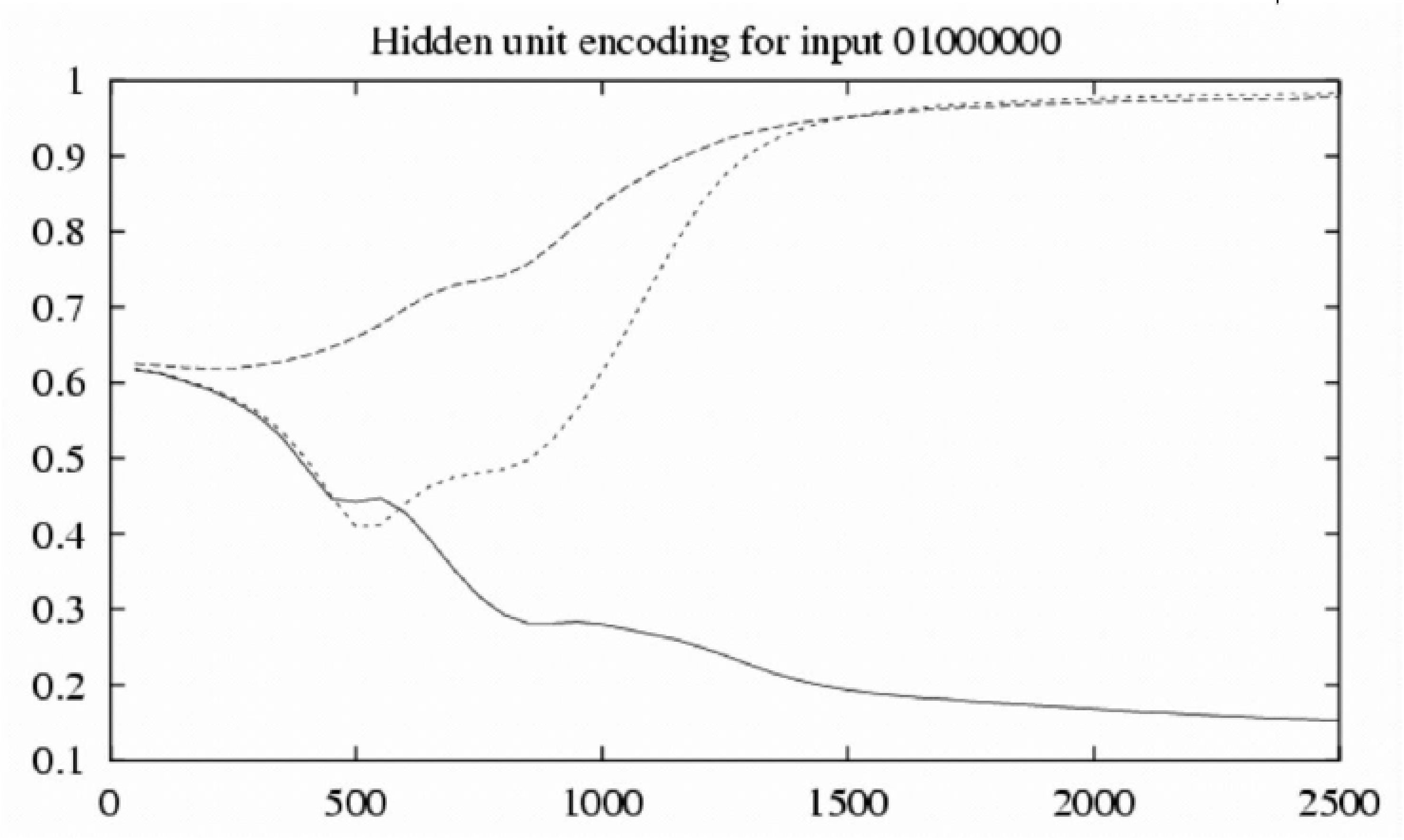
Learned parameters



Note that each value is assigned to the edge from the corresponding input

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Values for hidden layers



Examples

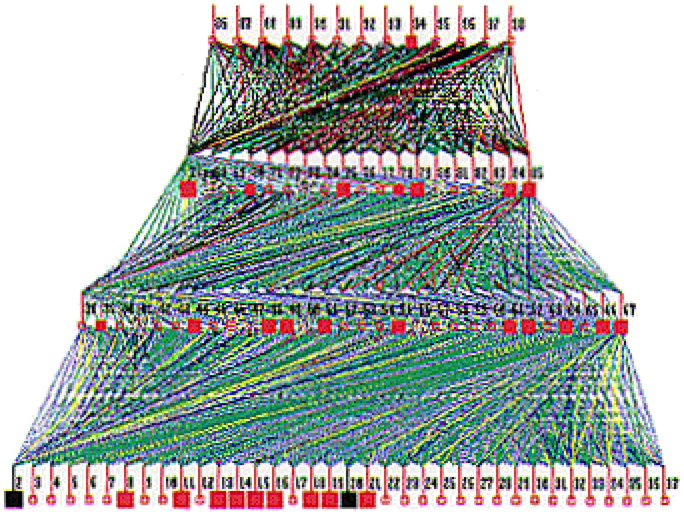


Figure 1: Feedforward ANN designed and tested for prediction of tactical air combat maneuvers.

Scientists See Promise in Deep-Learning Programs

Hao Zhang/The New York Times

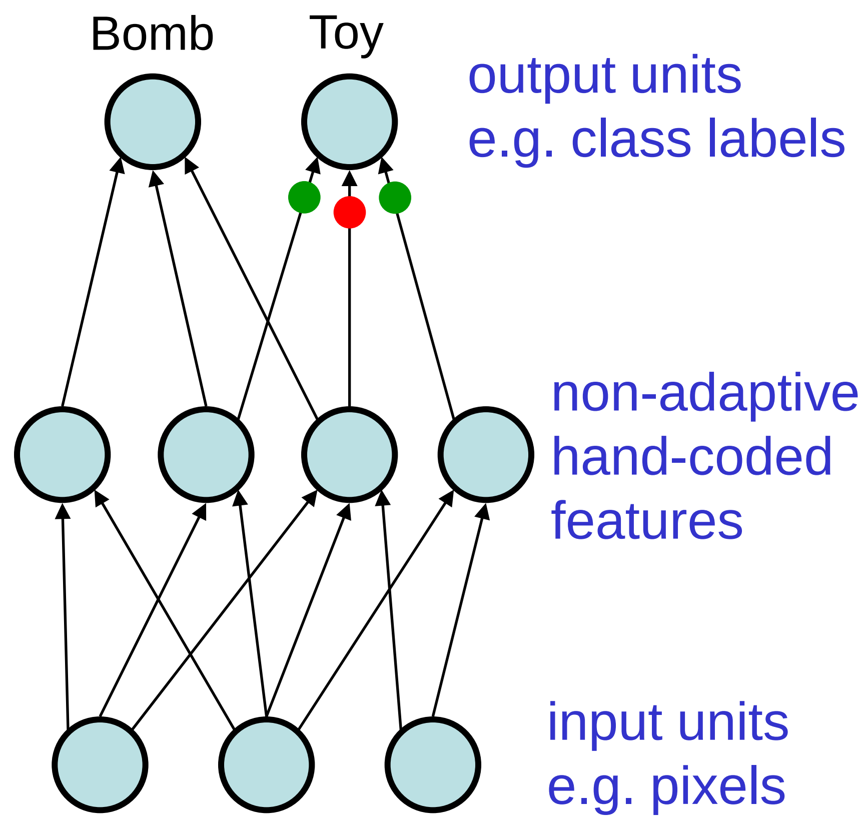
A voice recognition program translated a speech given by Richard F. Rashid, Microsoft's top scientist, into Mandarin Chinese.

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,

Historical background:
First generation neural networks

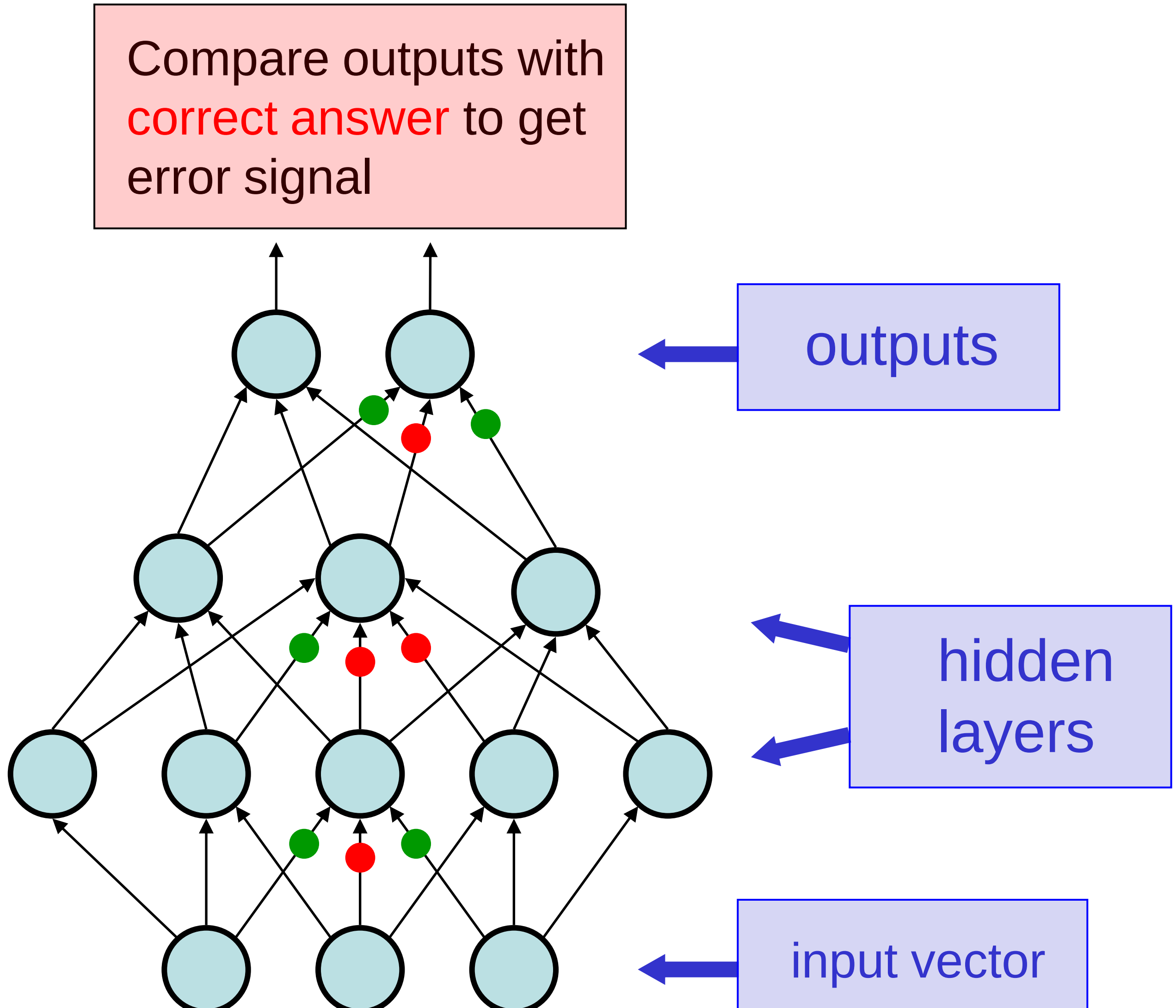
- Perceptrons (~1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
  – There was a neat learning algorithm for adjusting the weights.
  – But perceptrons are fundamentally limited in what they can learn to do.



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

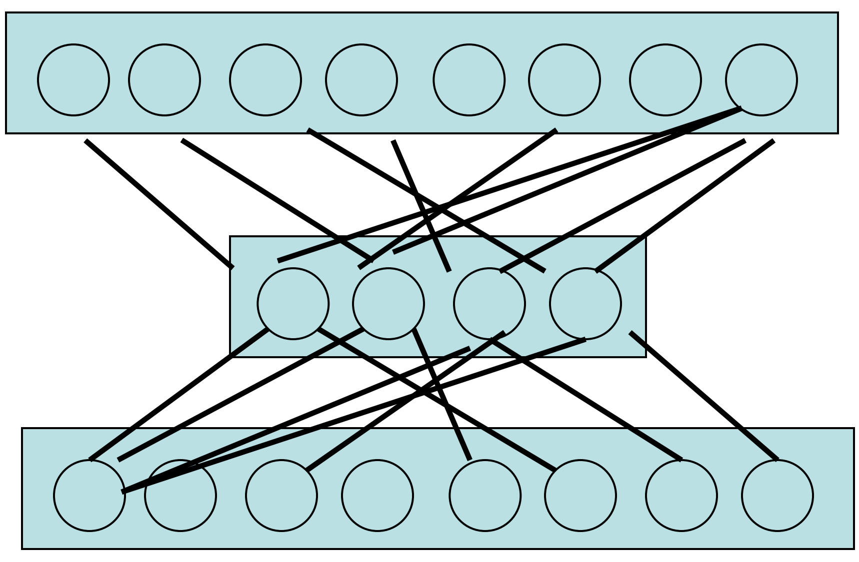
- Keep the efficiency and simplicity of using a gradient method for adjusting the weights, but use it for modeling the structure of the sensory input.
  - Iteratively learn the different layers.
  - Adjust the weights to maximize the probability that a generative model would have produced the sensory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

Iterative learning of layers

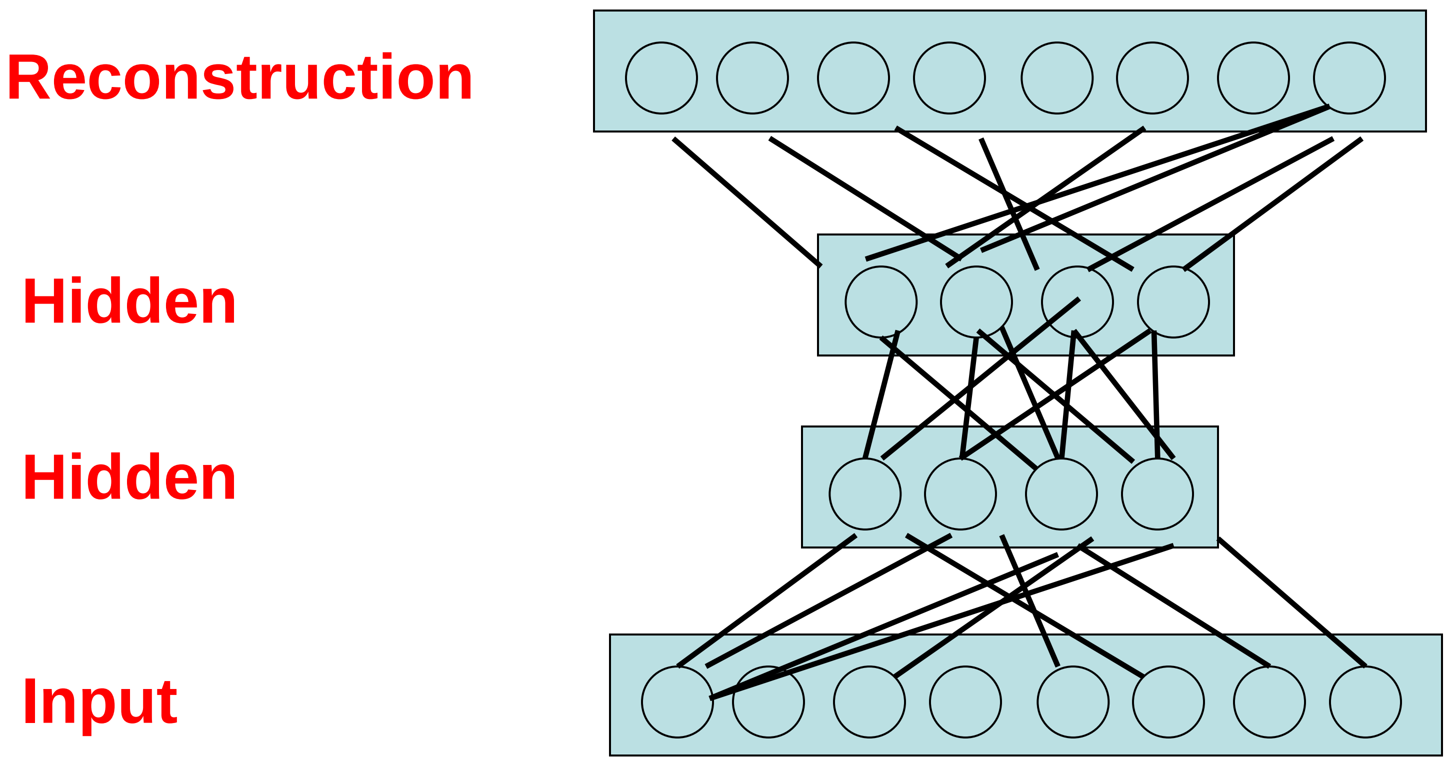
Reconstruction

Hidden

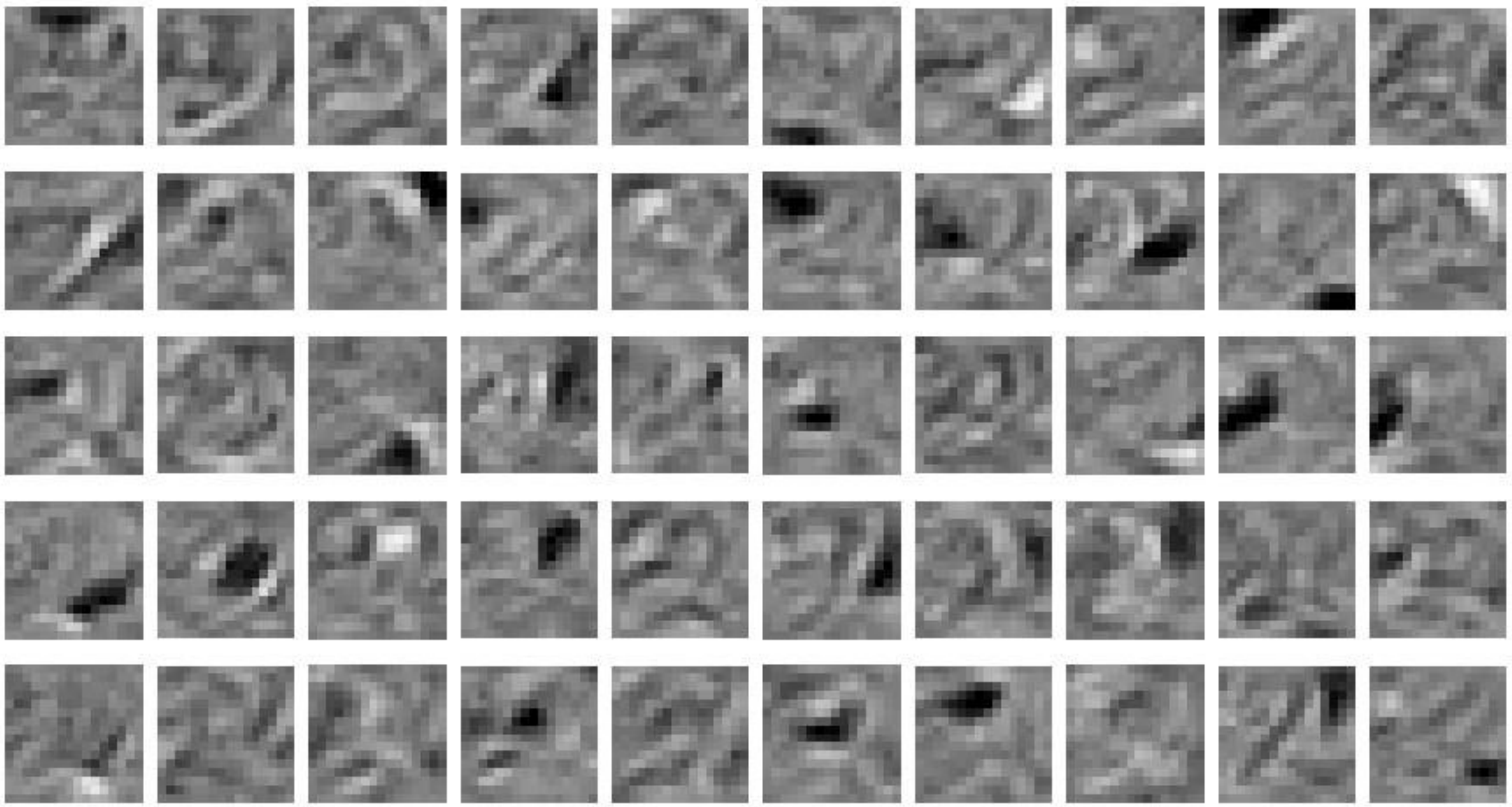
Input



Iterative learning of layers



The final 50 X 256 weights

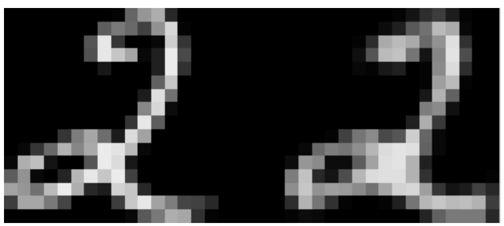


Each neuron grabs a different feature.

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

Data

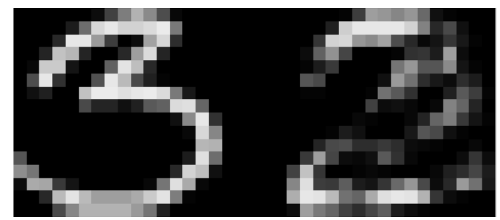
Reconstruction from activated binary features



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

Data

Reconstruction from activated binary features



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.

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: $g(x) = \dfrac{1}{1+e^{-x}}$ so $g'(x) = \dfrac{-e^{-x}}{\left(1+e^{-x}\right)^2}$

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

The Energy of a joint configuration

binary state of visible unit i

binary state of hidden unit j

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

weight between units i and j

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

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

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.

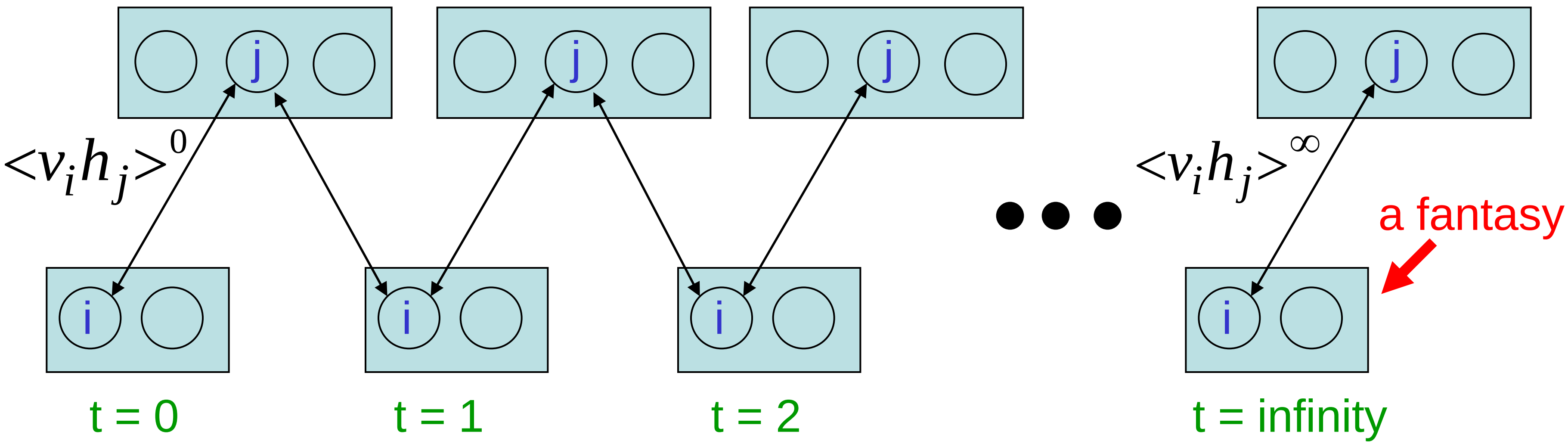
$$p(v,h) = \frac{e^{-E(v,h)}}{\sum_{u,g} e^{-E(u,g)}}$$

partition function

- The probability of a configuration of the visible units is the sum of the probabilities of all the joint configurations that contain it.

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

A picture of the maximum likelihood learning algorithm for an RBM

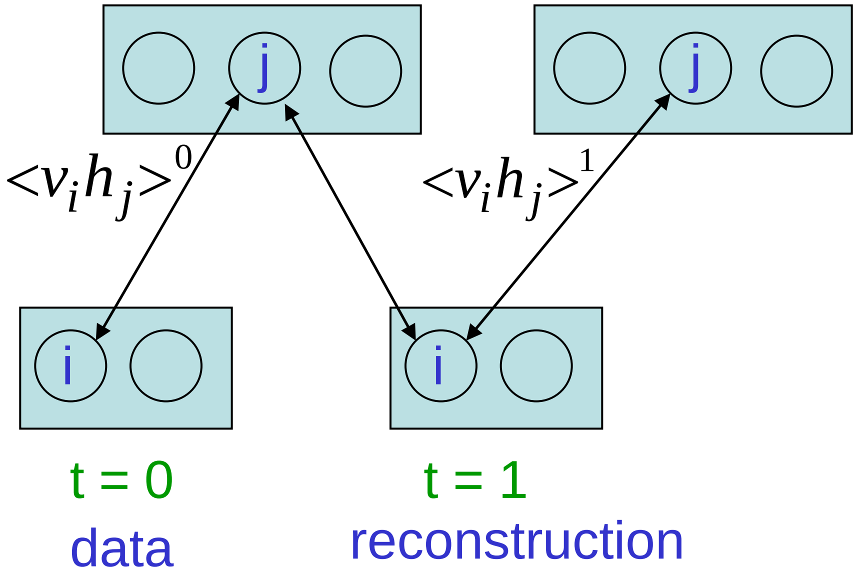


Start with a training vector on the visible units.

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

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

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.

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

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).