

Neural Networks

(aka Deep Learning)

Ties together many ideas from the course

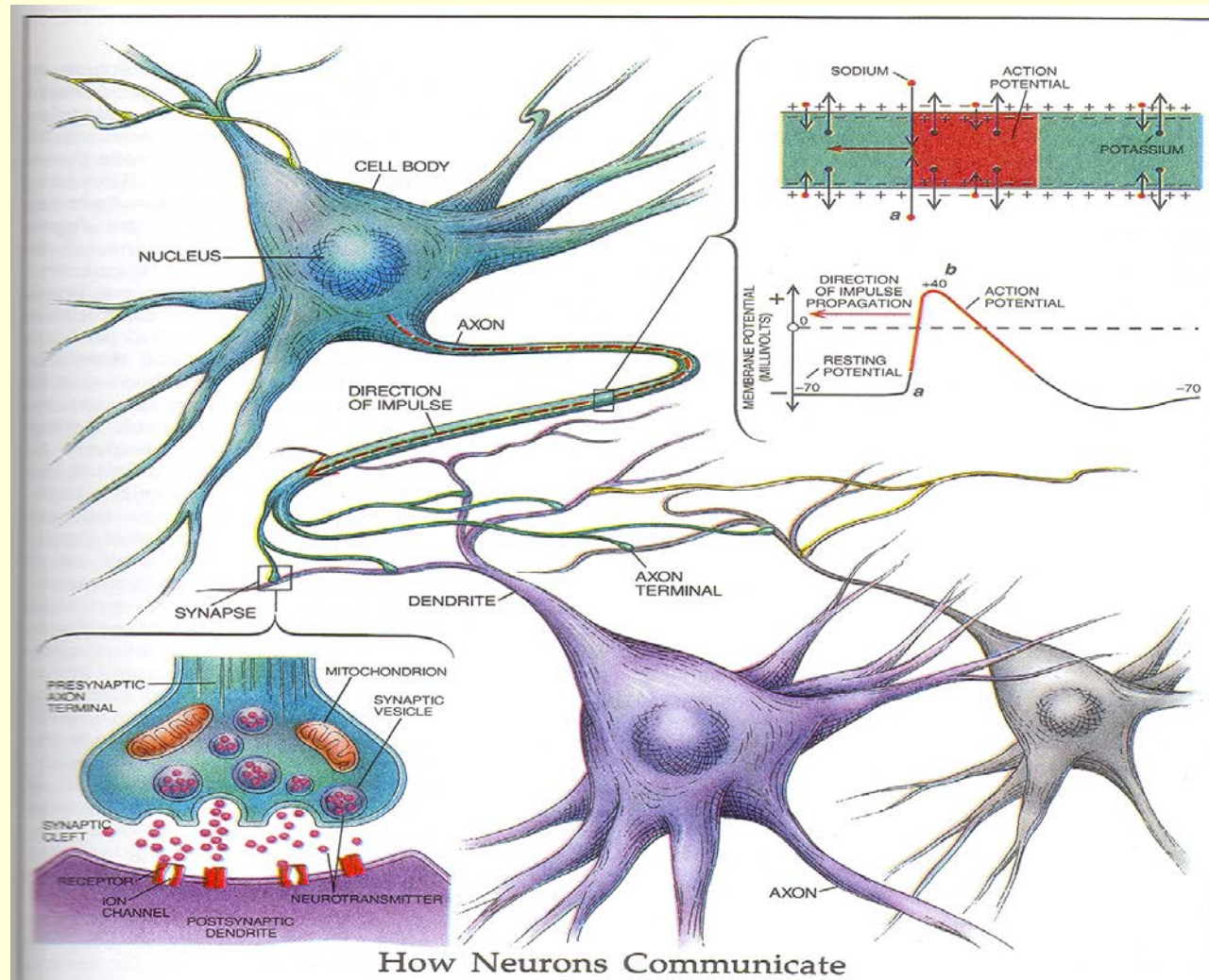
- logistic regression
- template matching/nearest neighbors
- decision forests / ensembles of weak learners
- stochastic gradient descent

What's new:

- feature learning (“end-to-end training”)
- Inspired by the brain

Reasons people study neural computation

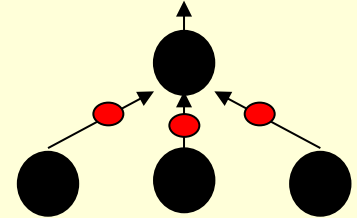
- To understand how the brain actually works.
 - Its very big and very complicated and made of stuff that dies when you poke it around. So we need to use computer simulations.
- To understand a style of parallel computation inspired by neurons and their adaptive connections.
 - Very different style from sequential computation.
 - should be good for things that brains are good at (e.g. vision)
 - Should be bad for things that brains are bad at (e.g. 23 x 71)
- To solve practical problems by using novel learning algorithms inspired by the brain
 - Learning algorithms can be very useful even if they are not how the brain actually works.



Beware of the
“Half-life of
Knowledge”!

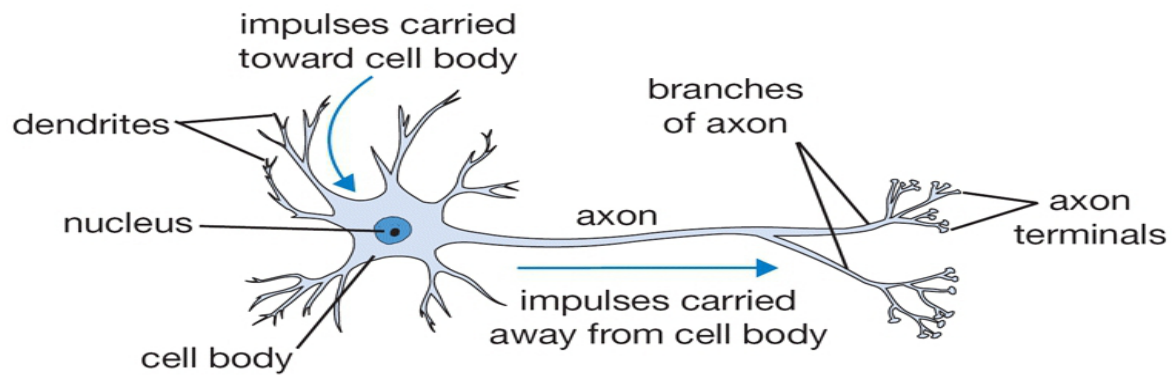
“How the brain works” on one slide!

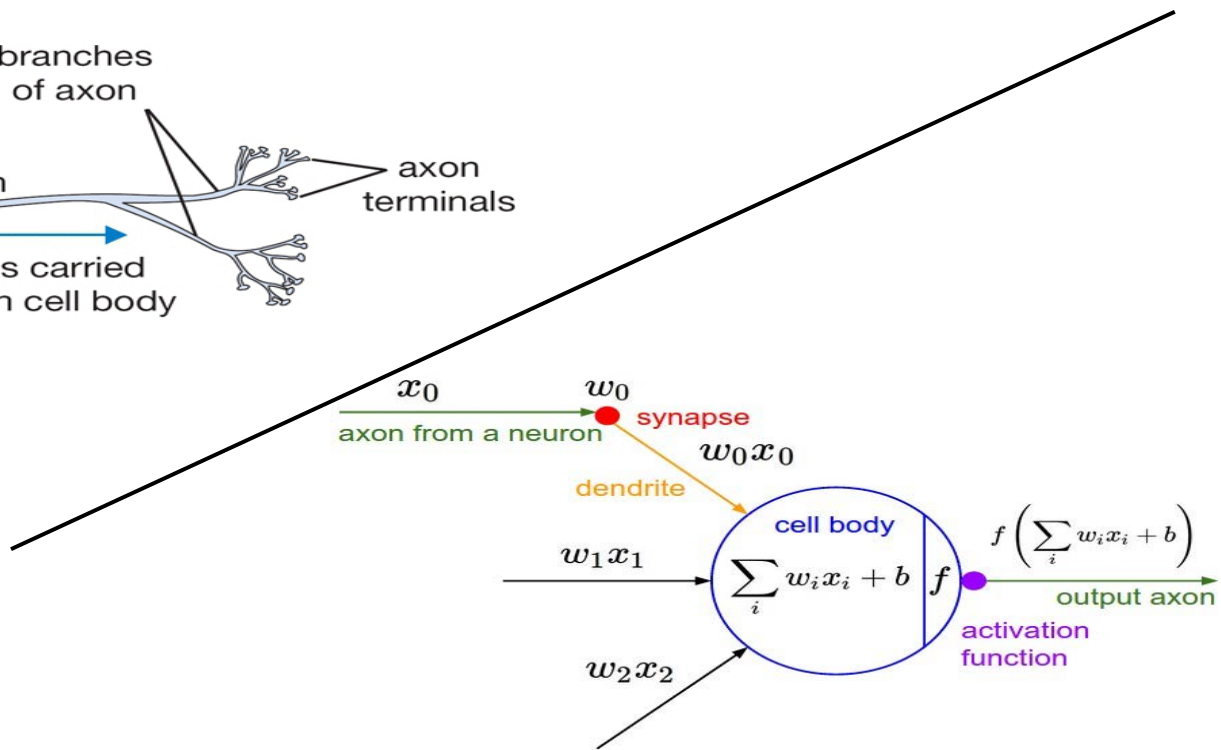
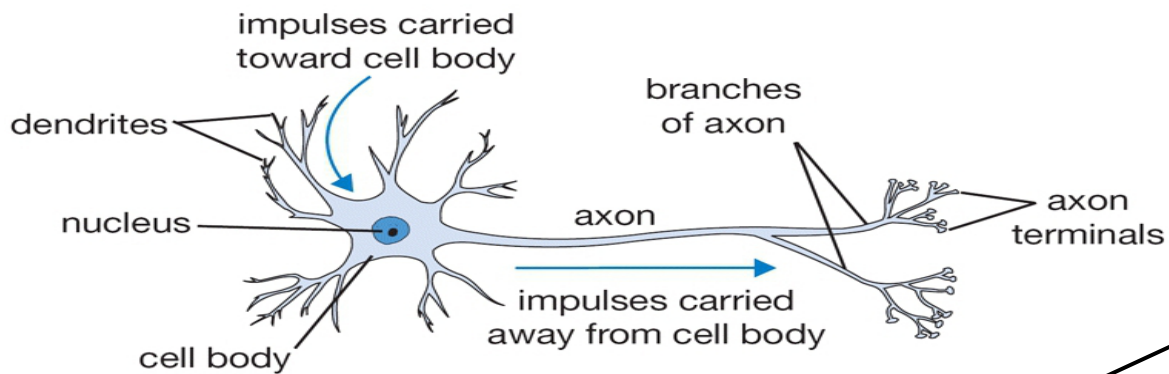
- Each neuron receives inputs from other neurons
 - A few neurons also connect to receptors.
 - Cortical neurons use spikes to communicate.
- The effect of each input line on the neuron is controlled by a synaptic weight
 - The weights can be positive or negative.
- The synaptic weights **adapt** so that the whole network learns to perform useful computations
 - Recognizing objects, understanding language, making plans, controlling the body.
- You have about 10^{11} neurons each with about 10^4 weights.
 - A huge number of weights can affect the computation in a very short time. Much better bandwidth than a computer.

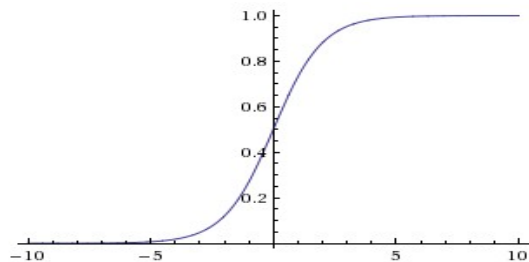
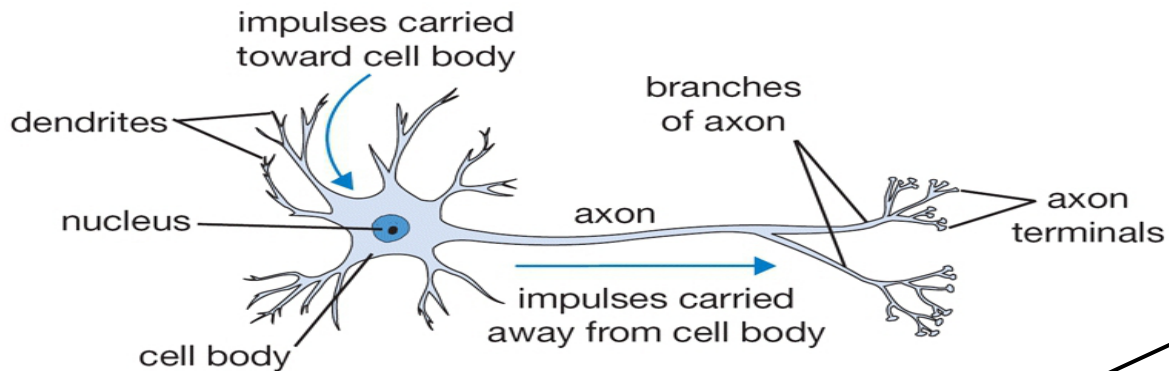


Modularity and the brain

- Different bits of the cortex do different things.
 - Local damage to the brain has specific effects.
 - Specific tasks increase the blood flow to specific regions.
- But cortex looks pretty much the same all over.
 - Early brain damage makes functions relocate.
- Cortex is made of general purpose stuff that has the ability to turn into special purpose hardware in response to experience.
 - This gives rapid parallel computation plus flexibility.
 - Conventional computers get flexibility by having stored sequential programs, but this requires very fast central processors to perform long sequential computations.

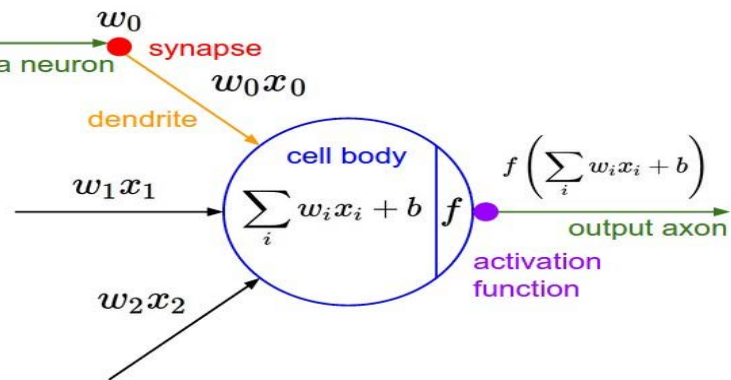




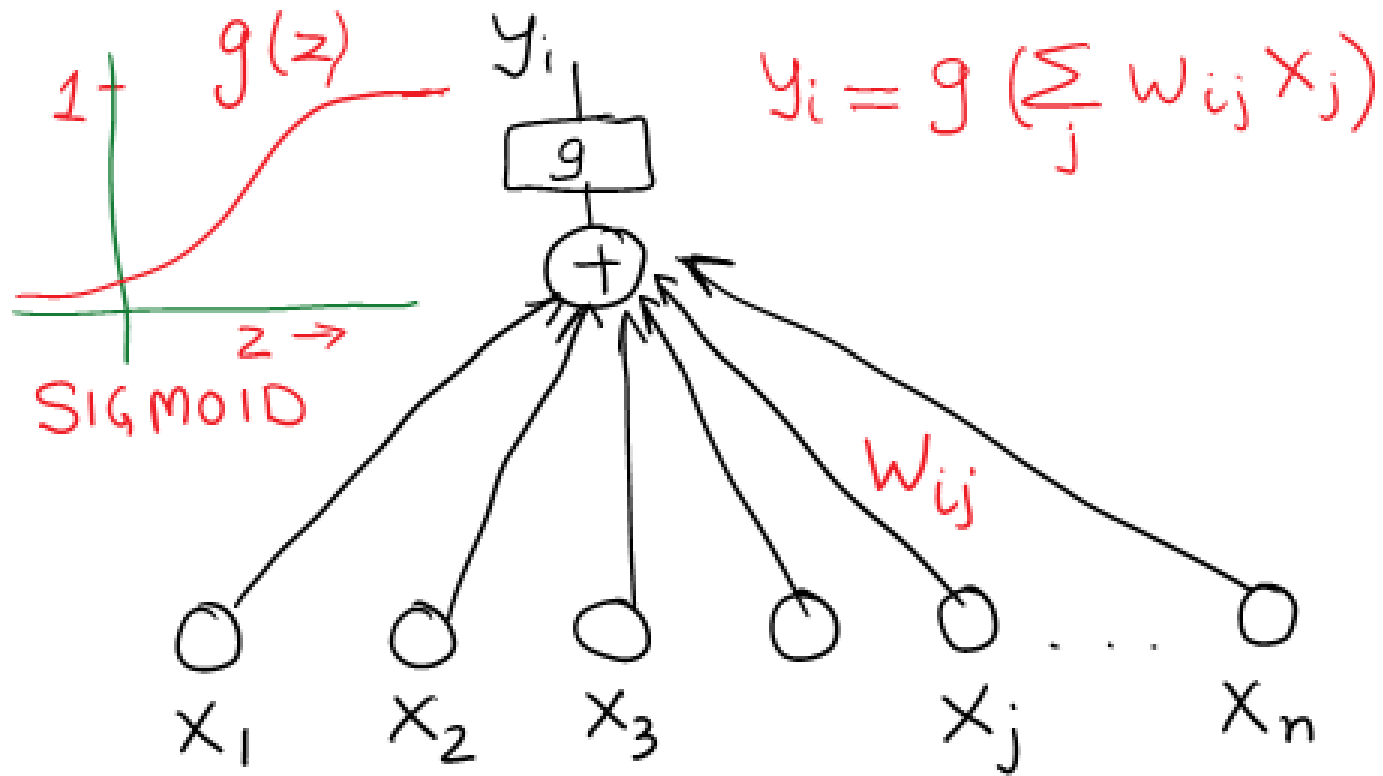


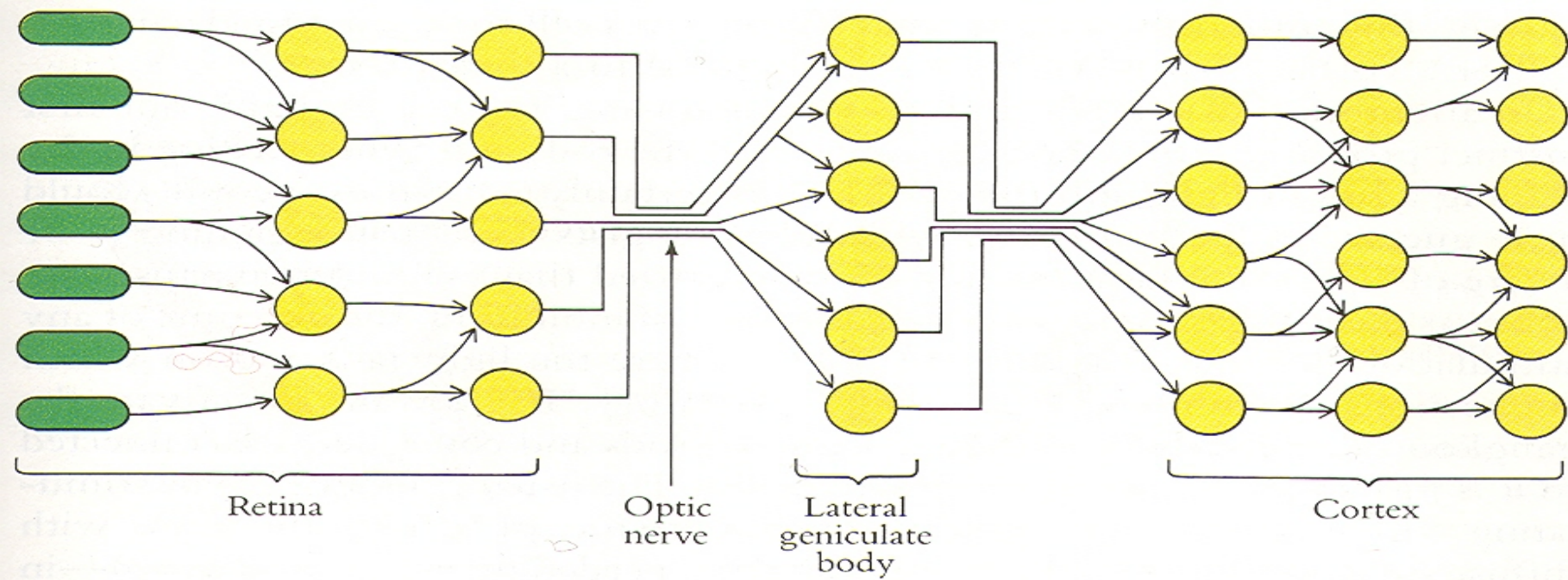
**sigmoid activation
function**

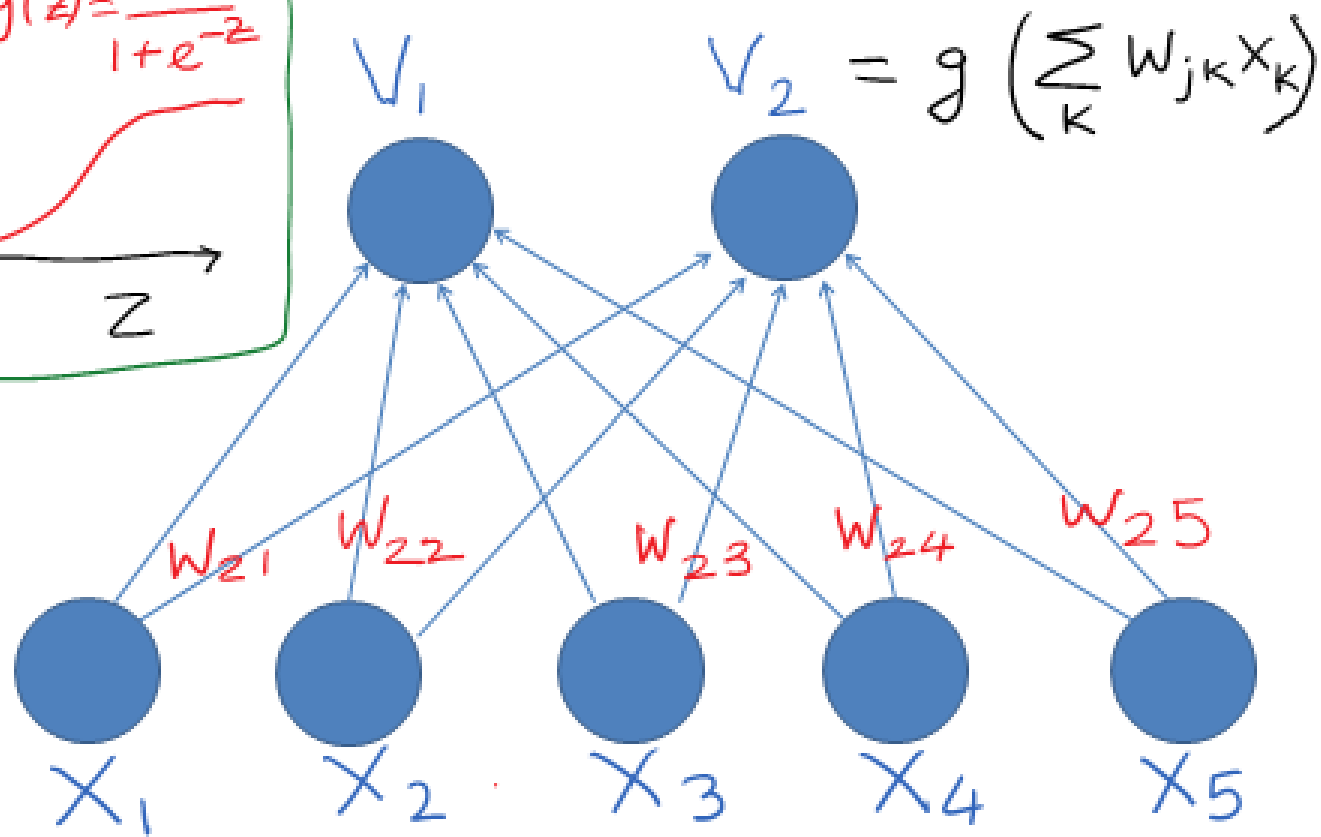
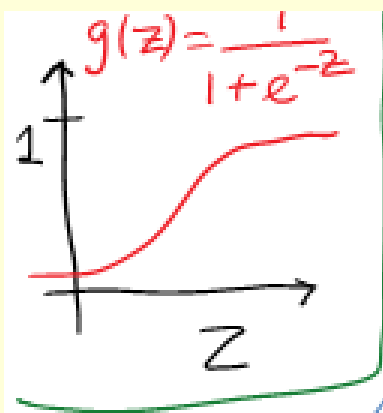
$$\frac{1}{1 + e^{-x}}$$



Mathematical Abstraction

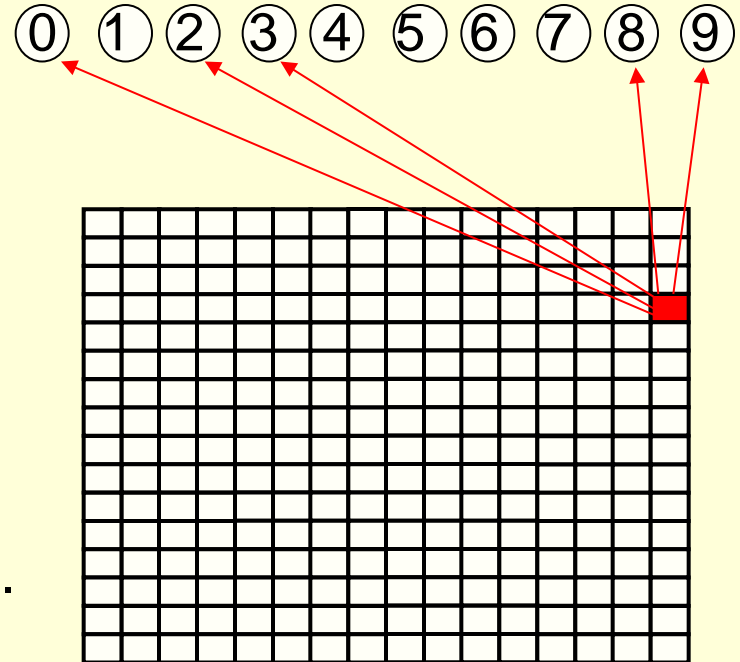




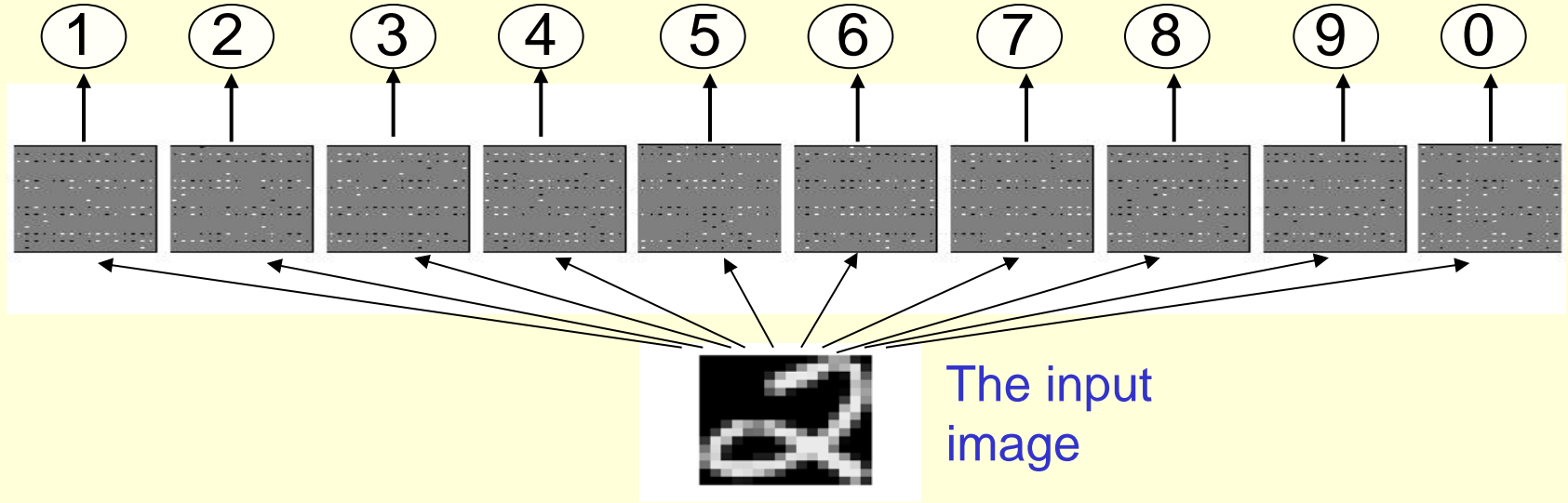


A very simple way to recognize handwritten shapes

- Consider a neural network with two layers of neurons.
 - neurons in the top layer represent known shapes.
 - neurons in the bottom layer represent pixel intensities.
- A pixel gets to vote if it has ink on it.
 - Each inked pixel can vote for several different shapes.
- The shape that gets the most votes wins.



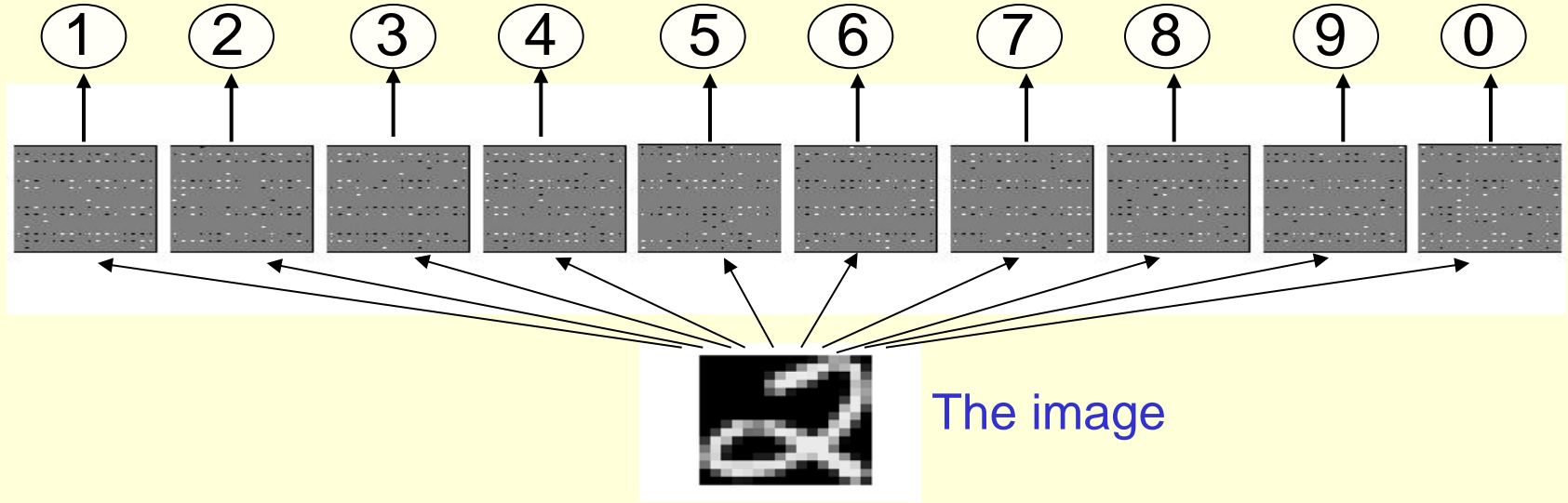
How to display the weights



Give each output unit its own “map” of the input image and display the weight coming from each pixel in the location of that pixel in the map.

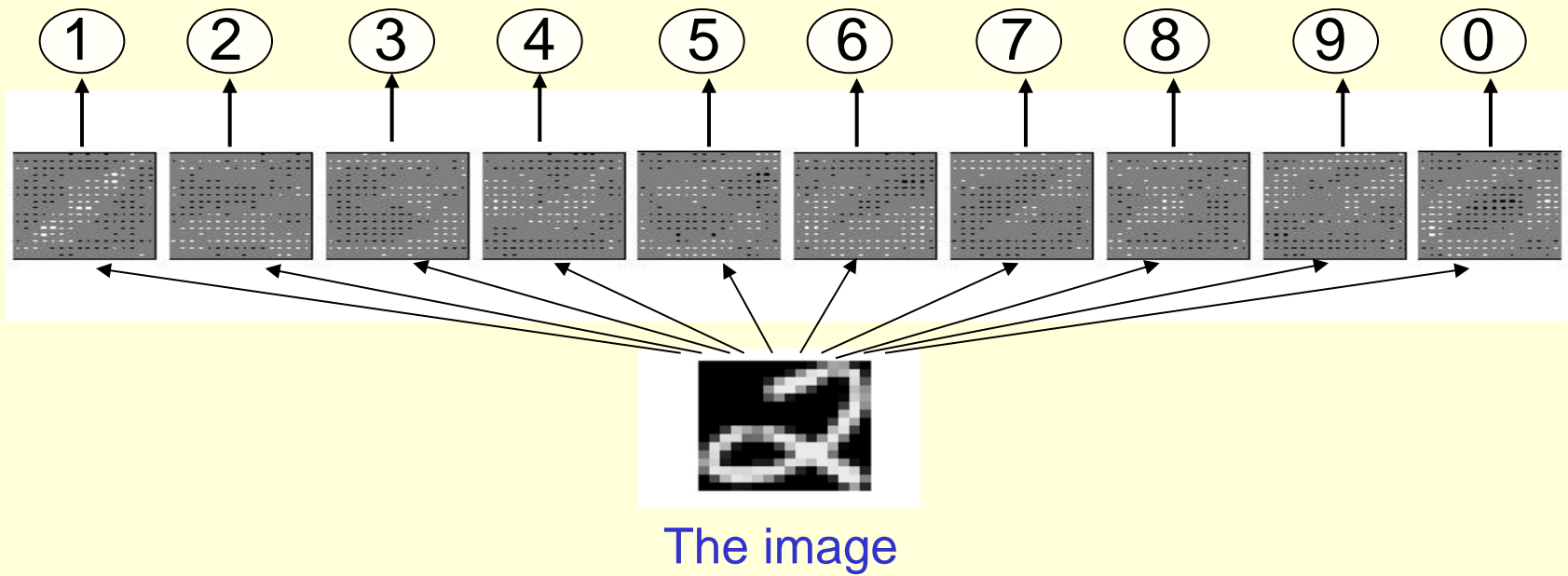
Use a black or white blob with the area representing the magnitude of the weight and the color representing the sign.

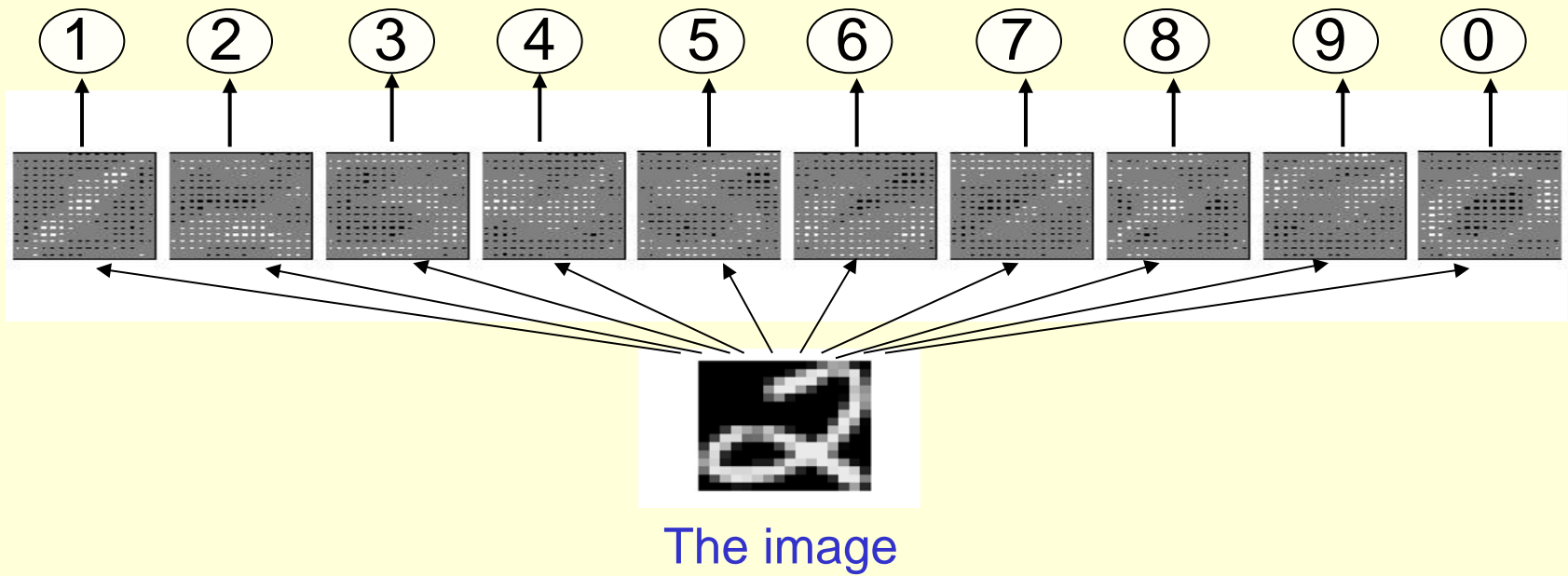
How to learn the weights

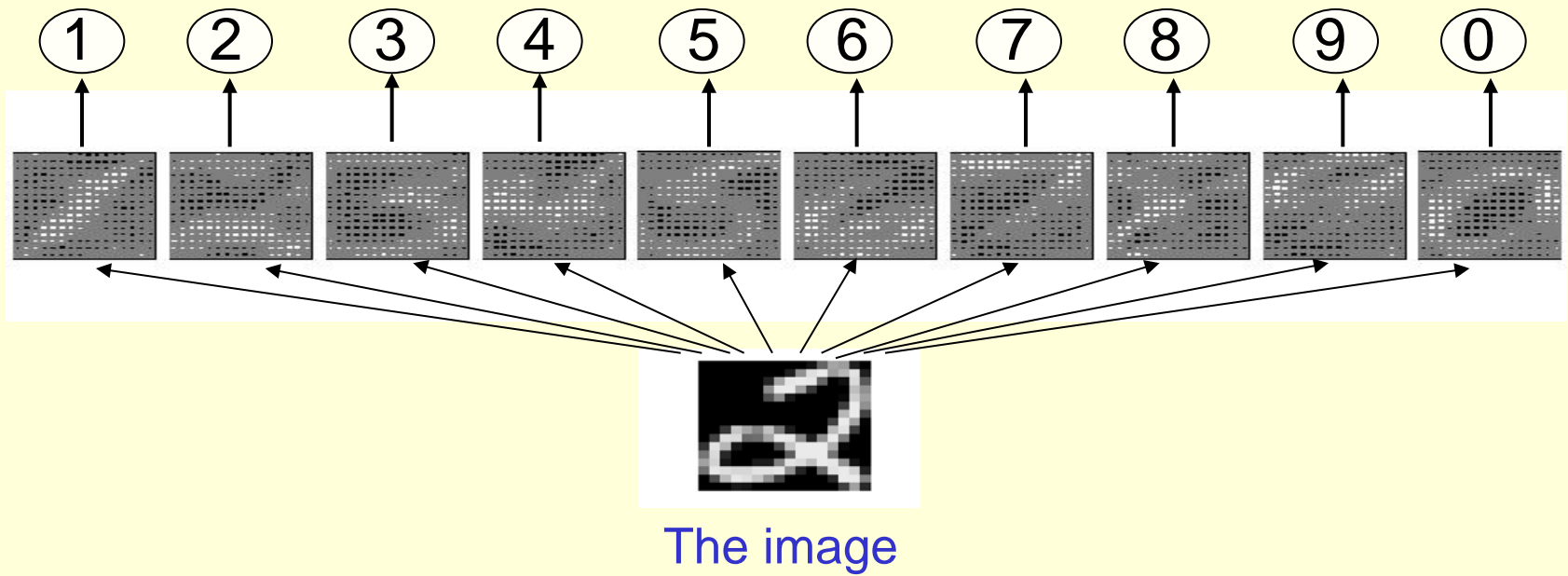


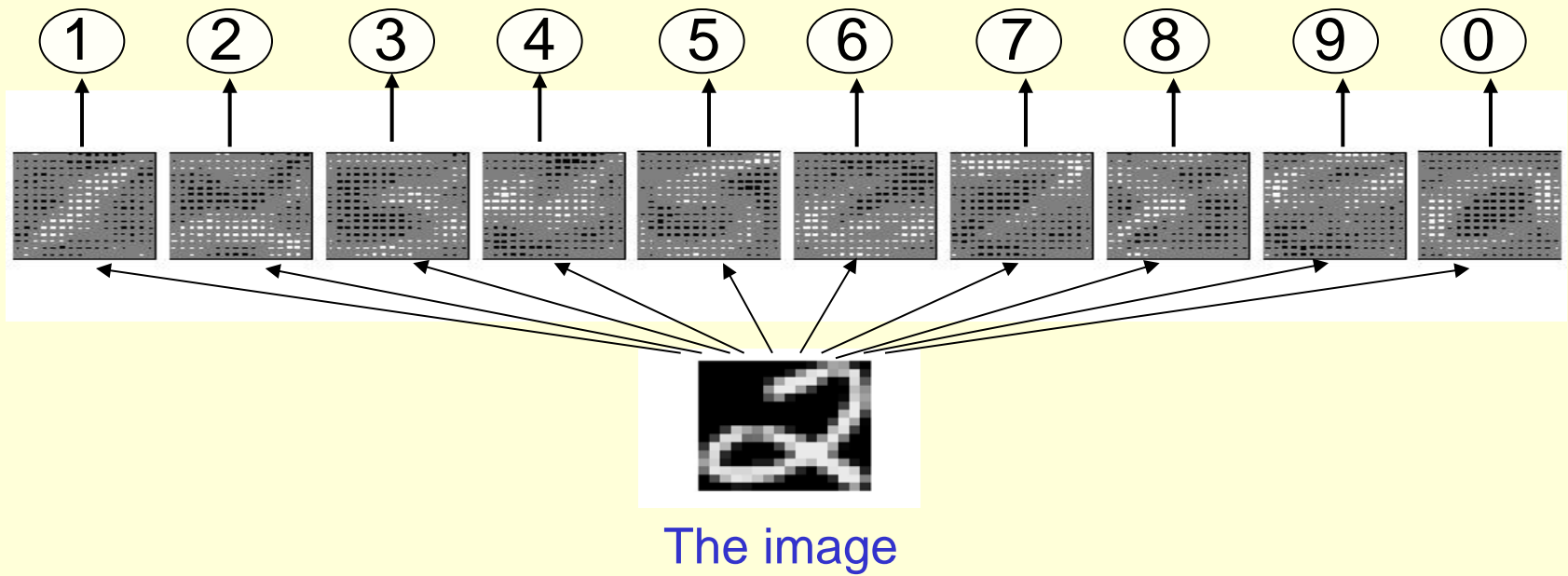
Show the network an image and **increment** the weights from active pixels to the correct class.

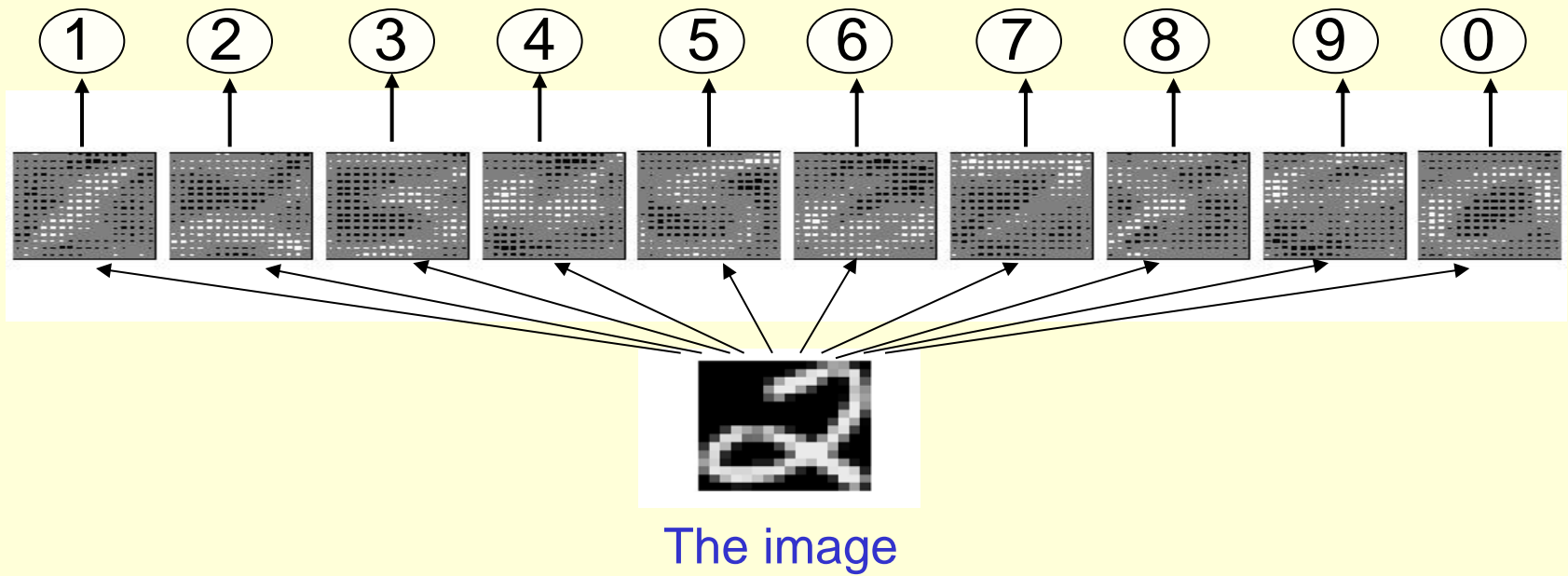
Then **decrement** the weights from active pixels to whatever class the network guesses.



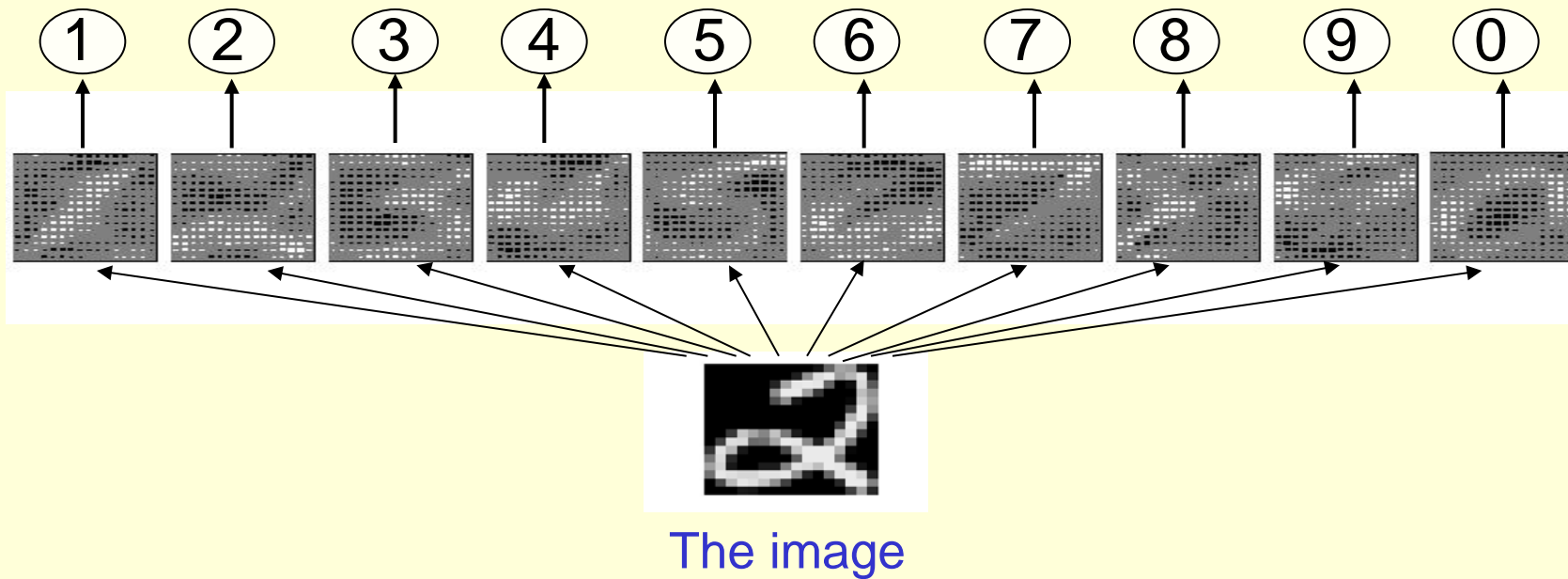








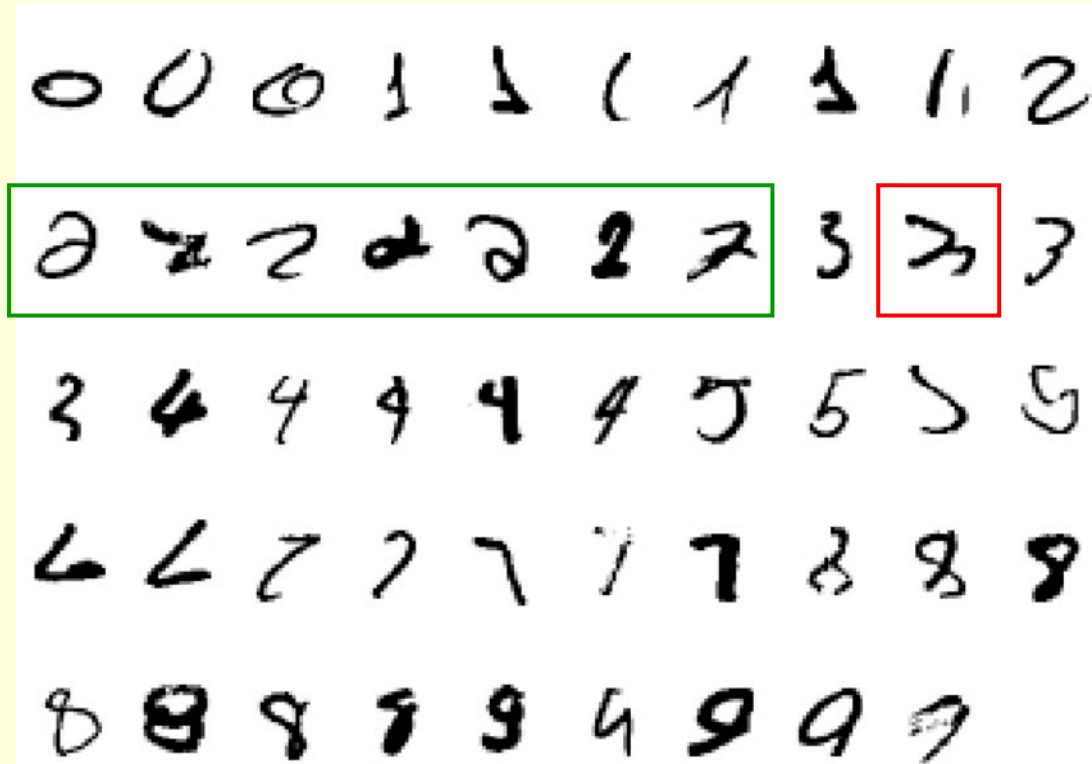
The learned weights



Why the simple learning algorithm is insufficient

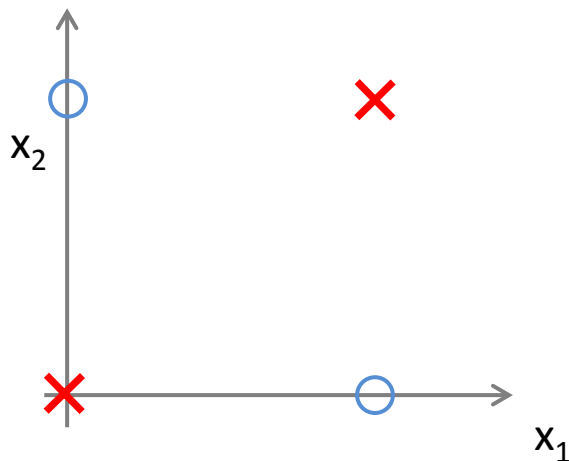
- A two layer network with a single winner in the top layer is equivalent to having a rigid template for each shape.
 - The winner is the template that has the biggest overlap with the ink.
- The ways in which hand-written digits vary are much too complicated to be captured by simple template matches of whole shapes.
 - To capture all the allowable variations of a digit we need to learn the features that it is composed of.

Examples of handwritten digits that can be recognized correctly the first time they are seen



Non-linear classification example: XOR/XNOR

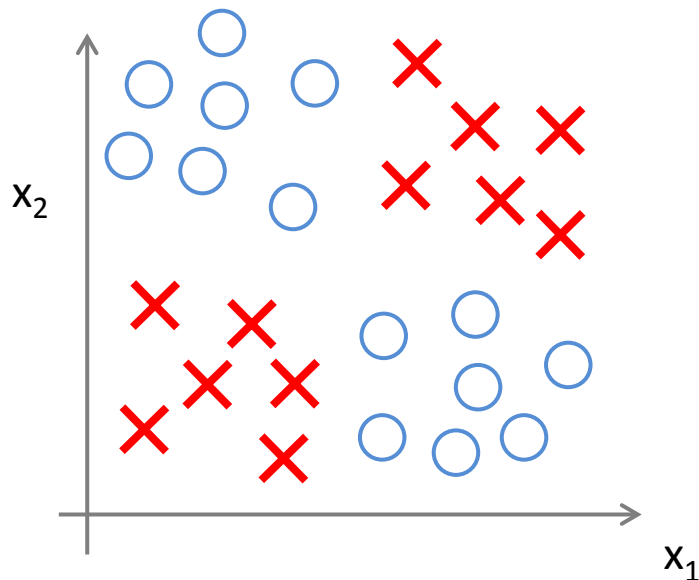
x_1, x_2 are binary (0 or 1).



$$y = x_1 \text{ XOR } x_2$$

$$x_1 \text{ XNOR } x_2$$

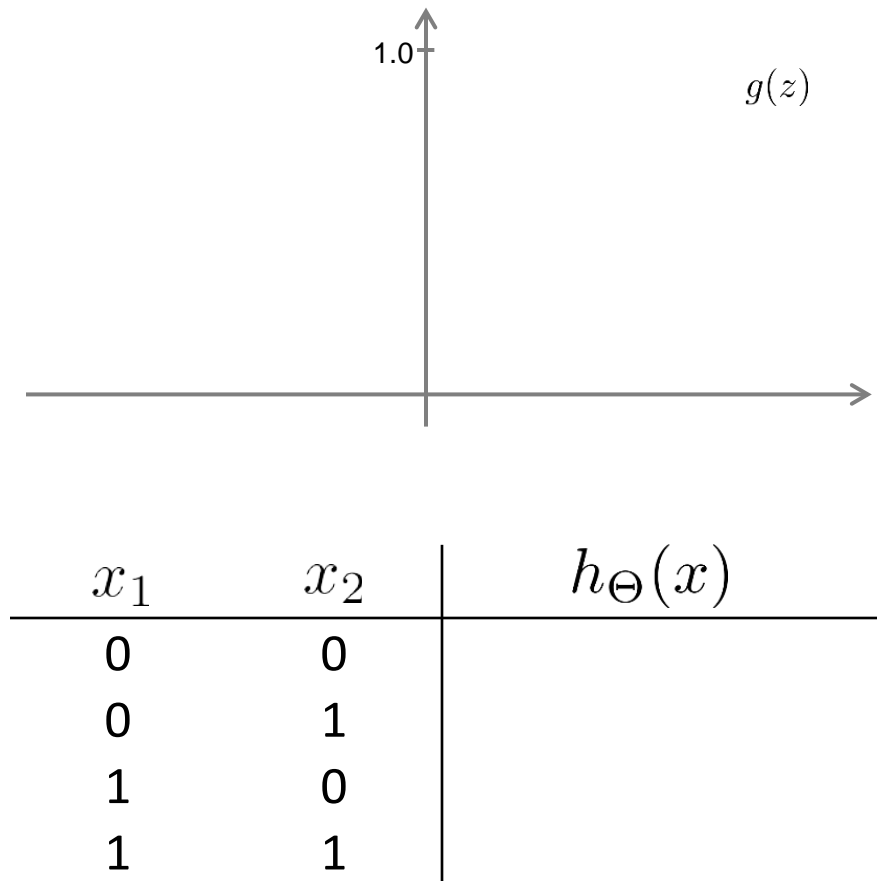
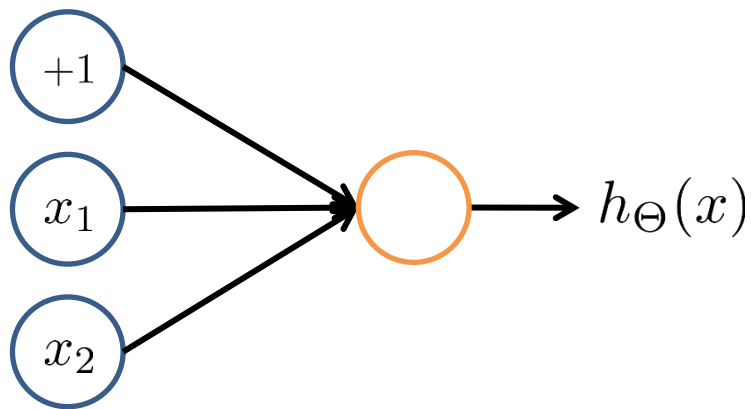
$$\text{NOT } (x_1 \text{ XOR } x_2)$$



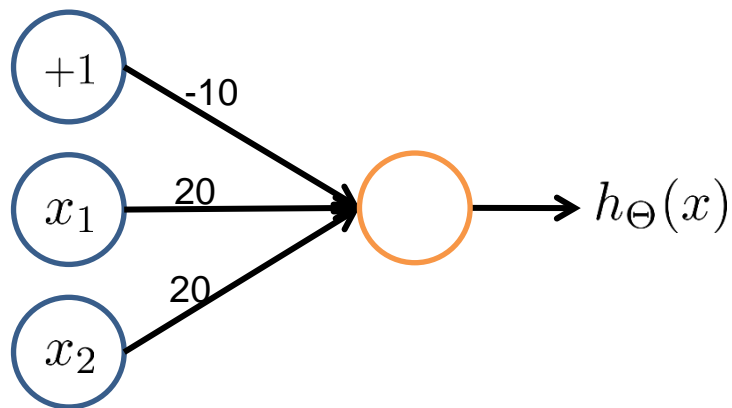
Simple example: AND

$$x_1, x_2 \in \{0, 1\}$$

$$y = x_1 \text{ AND } x_2$$

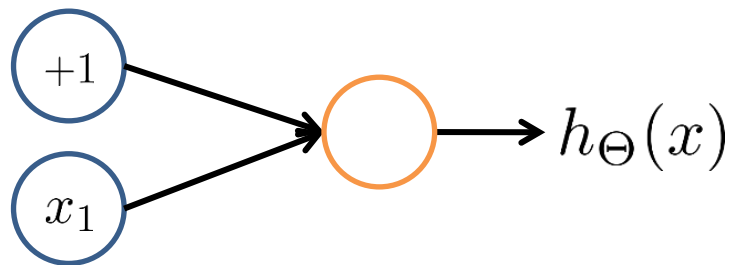


Example: OR function



x_1	x_2	$h_{\Theta}(x)$
0	0	
0	1	
1	0	
1	1	

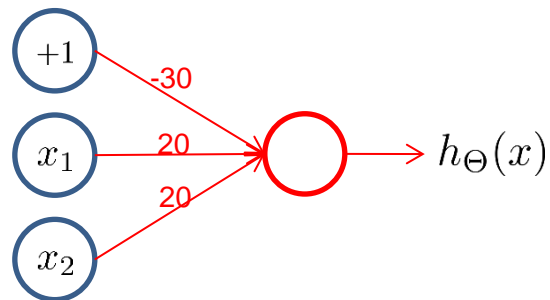
Negation:



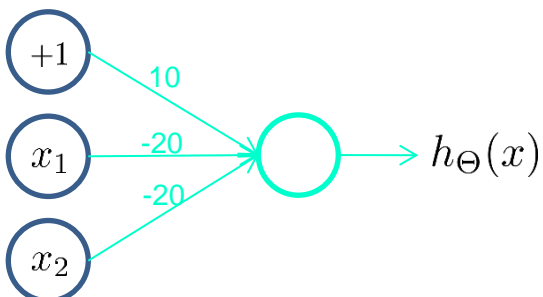
x_1	$h_{\Theta}(x)$
0	
1	

$$h_{\Theta}(x) = g(10 - 20x_1)$$

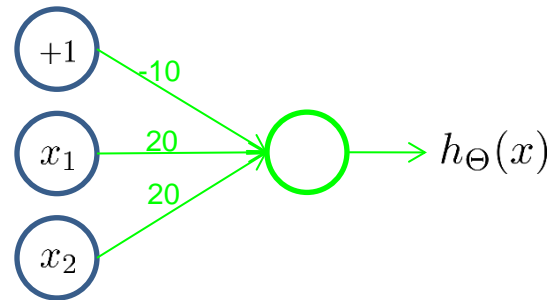
Putting it together: x_1 XNOR x_2



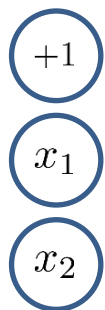
x_1 AND x_2



$(\text{NOT } x_1) \text{ AND } (\text{NOT } x_2)$

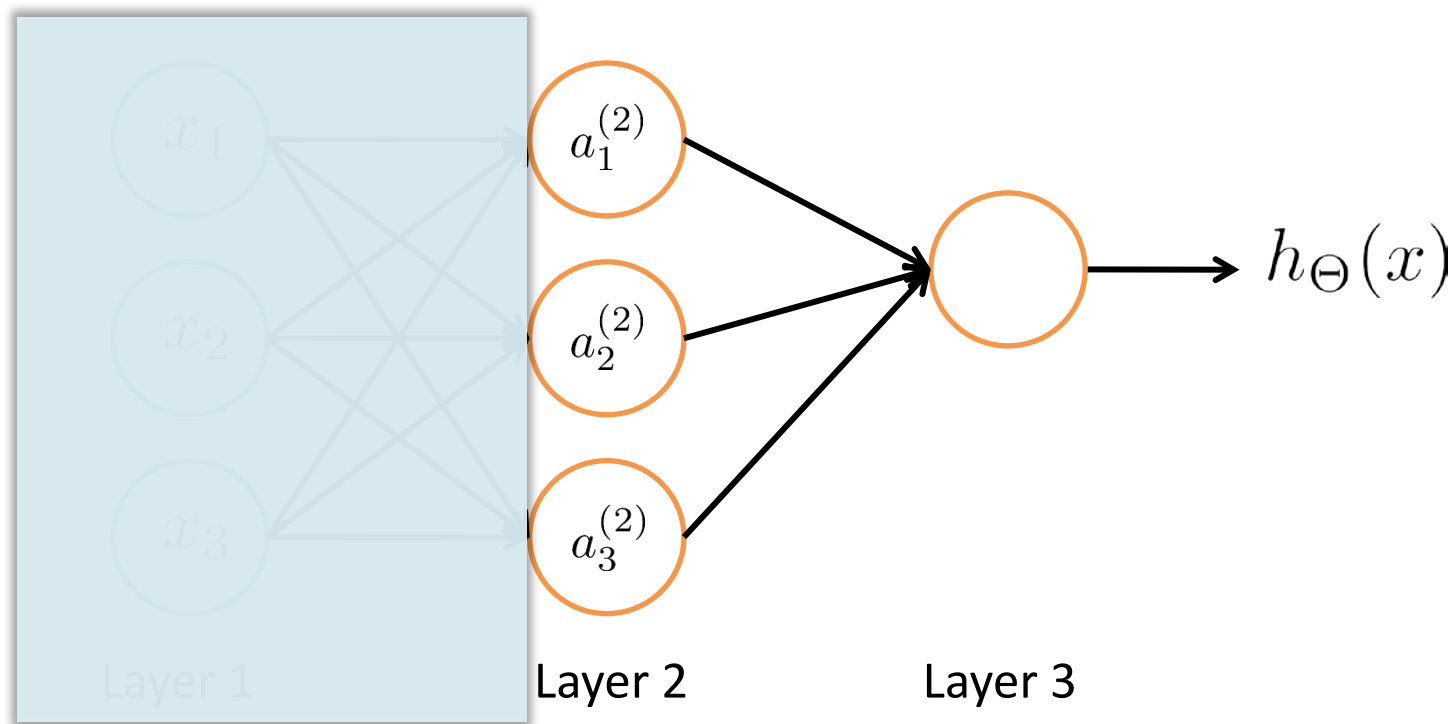


x_1 OR x_2

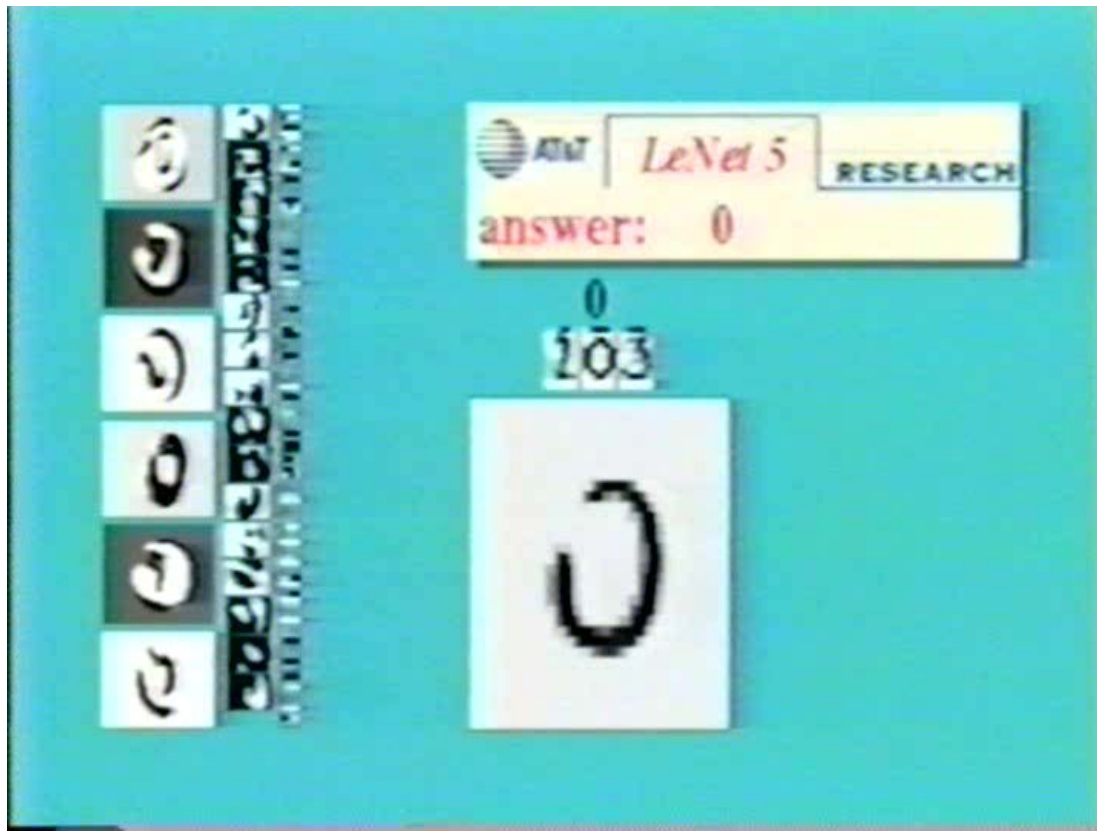


x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_{\Theta}(x)$
0	0			
0	1			
1	0			
1	1			

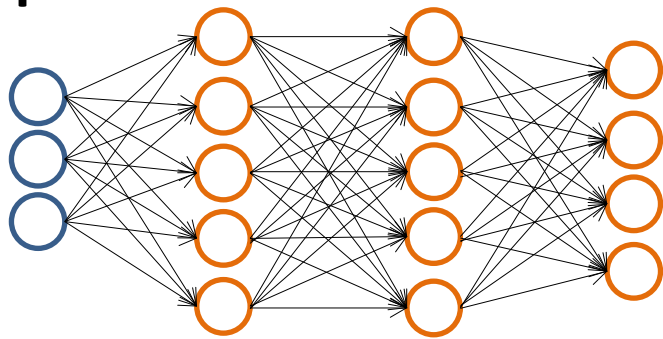
Neural Network learning its own features



Handwritten digit classification



Multiple output units: One-vs-all.

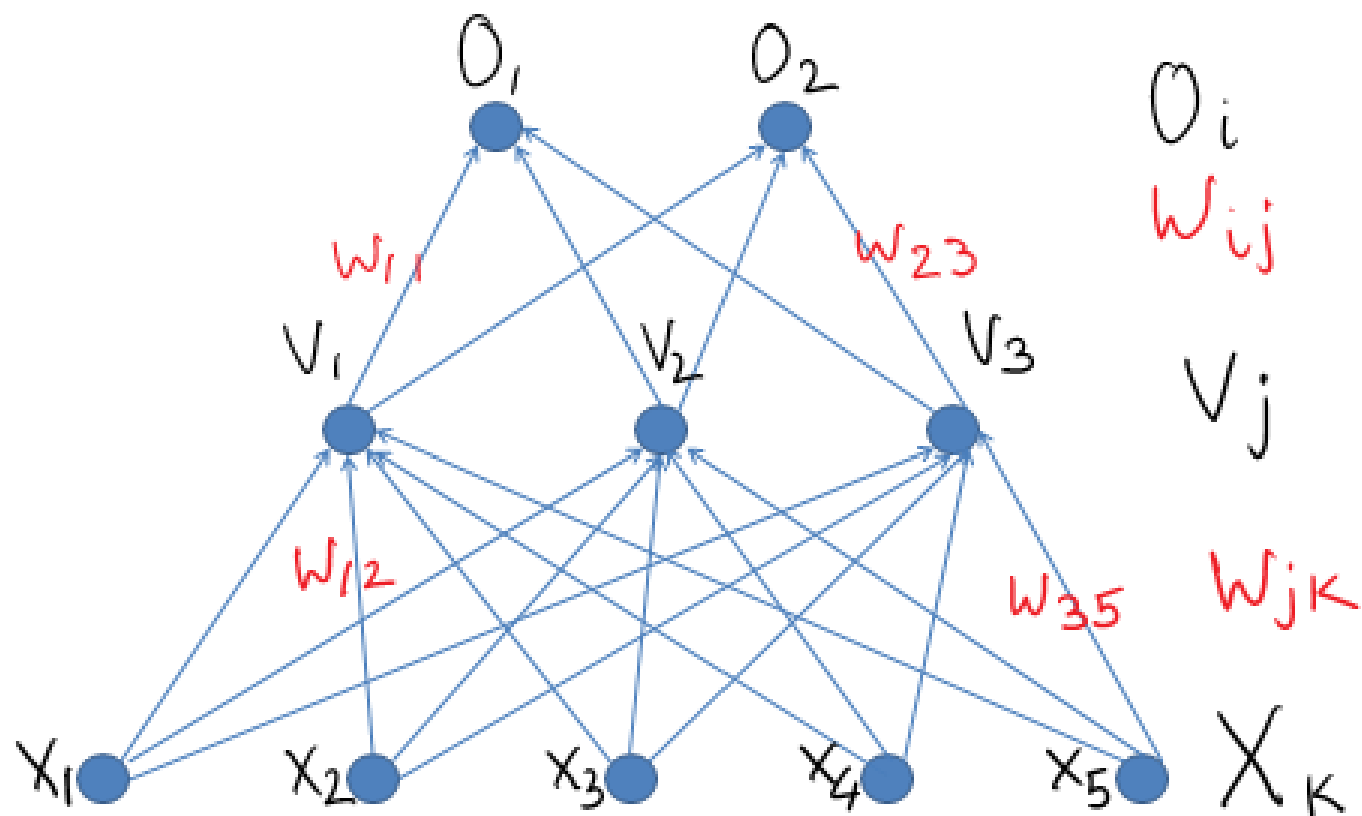


$$h_{\Theta}(x) \in \mathbb{R}^4$$

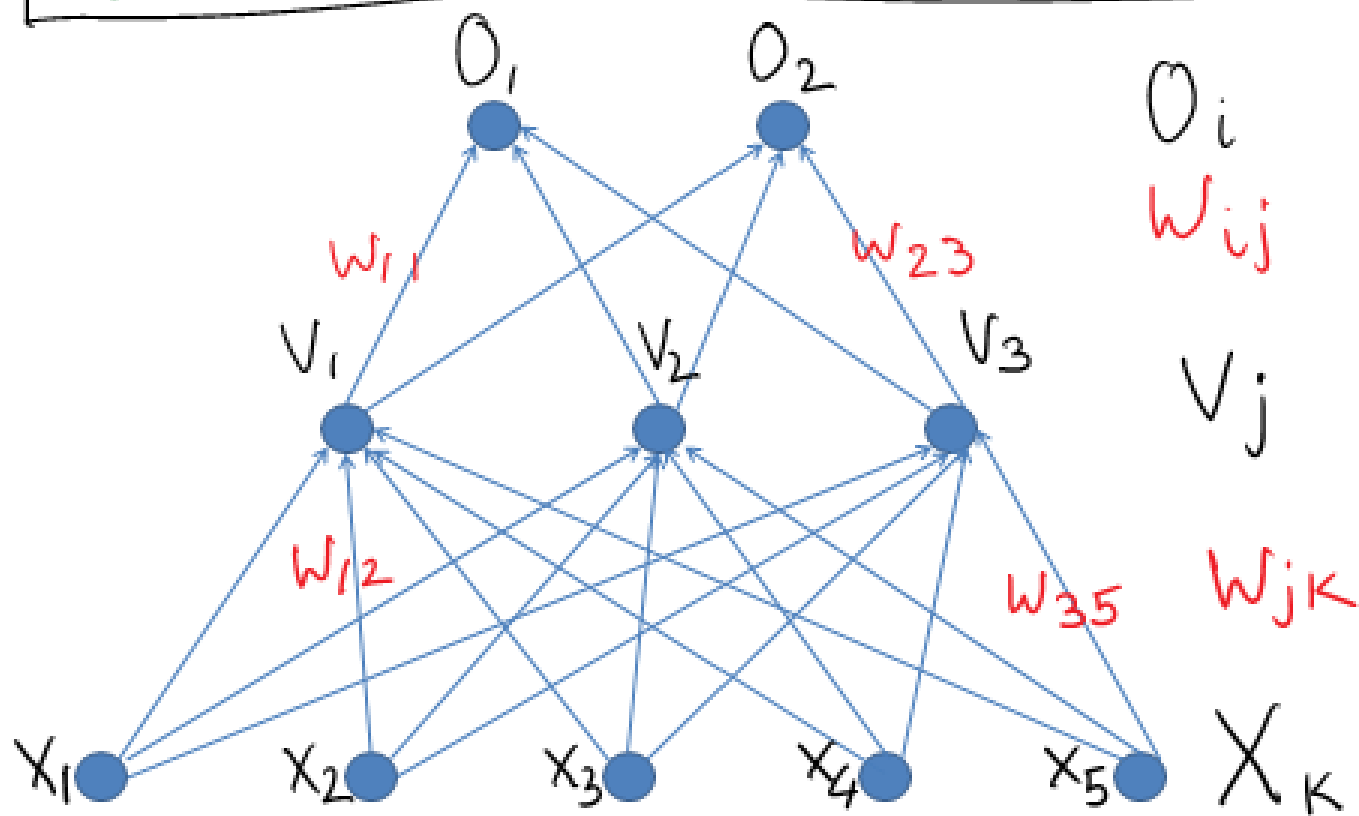
Want $h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.
when pedestrian when car when motorcycle

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

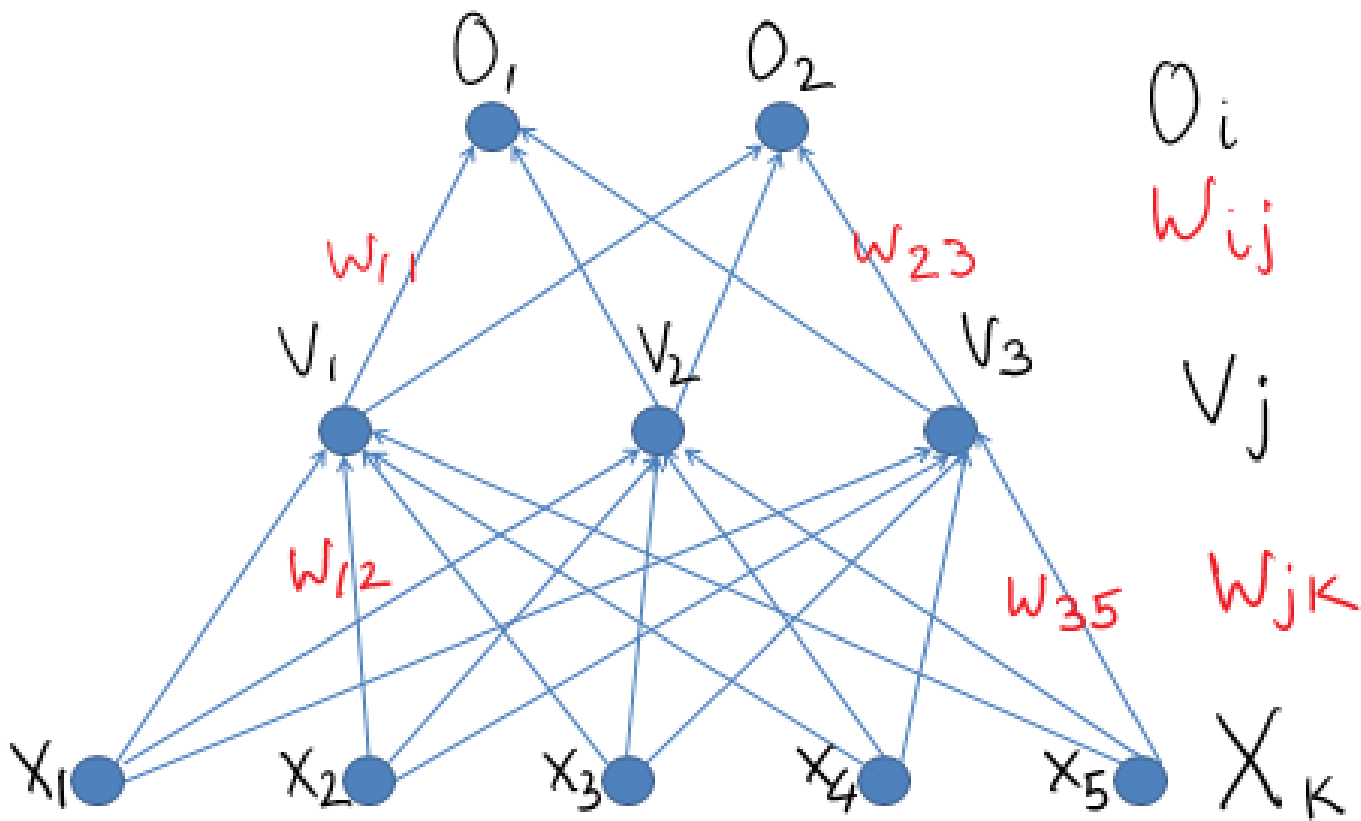
$y^{(i)}$ one of $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$
pedestrian car motorcycle truck



$$V_j = g\left(\sum_k w_{jk} x_k\right); \quad O_i = g\left(\sum_j w_{ij} V_j\right)$$



$$O_i = g\left(\sum_j W_{ij} g\left(\sum_k W_{jk} x_k\right)\right)$$

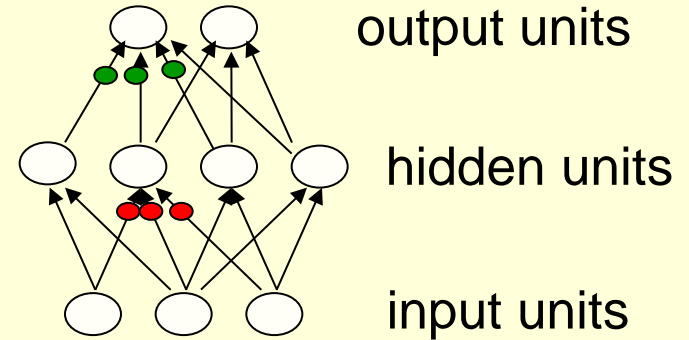


Training a neural network

Learning by perturbing weights

(this idea occurs to everyone who knows about evolution)

- Randomly perturb one weight and see if it improves performance. If so, save the change.
 - This is a form of reinforcement learning.
 - **Very inefficient.** We need to do multiple forward passes on a representative set of training cases just to change one weight.
 - Towards the end of learning, large weight perturbations will nearly always make things **worse**, because the weights need to have the right relative values.



Training a neural network

Gradient Descent

- **Numerical gradient:** easy to write ☺, slow ☹, approximate ☹
 - $O(N_w^2)$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- **(native) analytic gradient:** exact ☺, complicated ☹, slow ☹
 - $O(N_w^2)$
- **back-propagation (cached analytic gradient):** exact ☺, fast ☺, error-prone ☹
 - $O(N_w)$, similar to dynamic programming
 - glorified chain rule

In practice: Derive analytic gradient, check your implementation with numerical gradient