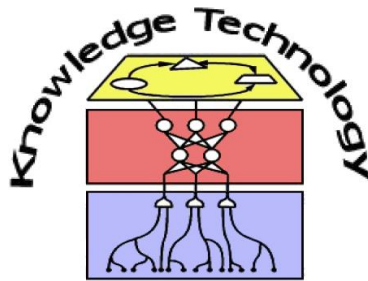


# Neural Networks

## Lecture 2: The Neuron and its Models



<http://www.informatik.uni-hamburg.de/WTM/>

# Overview

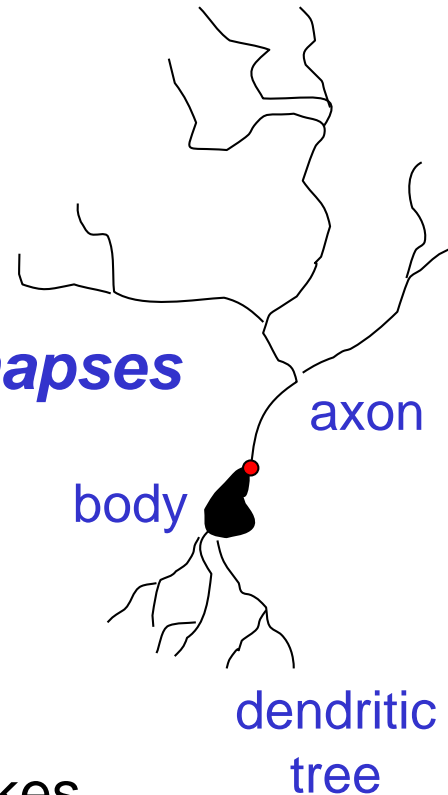
- Introduction, motivation, neural architectures
- General Background Sources
  - R. Rojas: Neural networks 1996 (book online available)  
<http://page.mi.fu-berlin.de/rojas/neural/neuron.pdf>
  - Marsland, S. Machine Learning: An Algorithmic Perspective. Chapman & Hall, 2009.
  - I. Goodfellow, Y. Bengio, A. Courville: Deep Learning 2016  
[www.deeplearningbook.org](http://www.deeplearningbook.org)
  - Introduction slides partly based on Jeff Hinton's lecture

# The goals of neural computation

- To understand how the brain actually works
- To understand a new style of 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. speech, vision, navigation)
    - Should be bad for things that brains are bad at (e.g. 23 x 71)
- To solve practical problems by developing novel learning algorithms

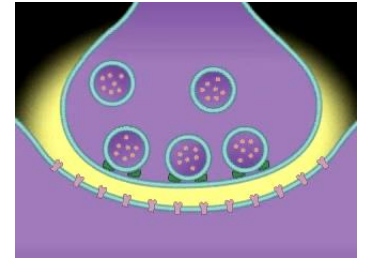
# A typical cortical neuron

- Gross physical structure:
  - There is one axon that branches
  - There is a **dendritic** tree that collects input from other neurons
- Axons typically contact dendritic trees at **synapses**
  - A spike of activity in the axon causes charge to be injected into the post-synaptic neuron
- Spike generation:
  - There is an **axon** that generates outgoing spikes whenever enough charge has flowed in at synapses to depolarize the cell membrane

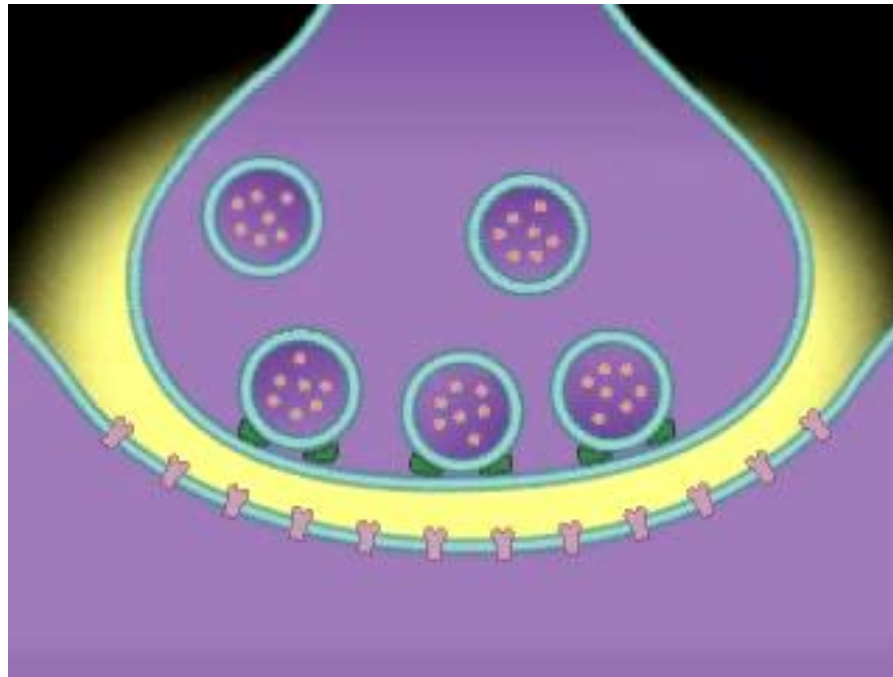


# Synapses

- When spike travels along an axon and arrives at synapse it causes vesicles of **transmitter** chemical to be released
- Transmitter molecules diffuse across **synaptic cleft** and bind to receptor molecules in membrane of the post-synaptic neuron
- The effectiveness of the synapse can be changed
  - vary the **number of vesicles** of transmitter
  - vary the **number of receptor** molecules
- Synapses are slow, but have advantages over RAM
  - adapt using locally available signals



# What biological synapses do...

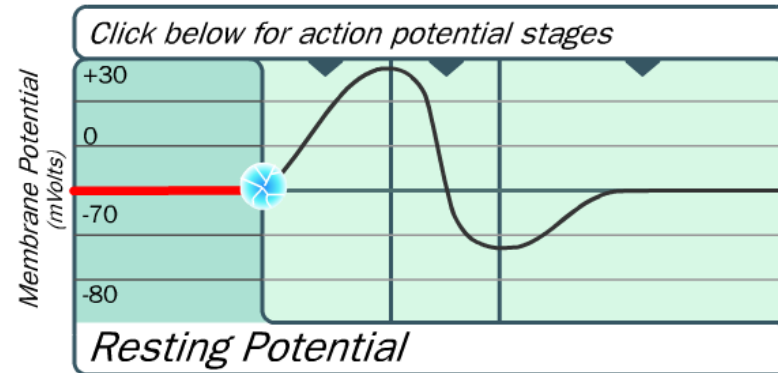


Transmitters are “universal”, e.g. they work in the human brain the same as transmitters in the mouse brain

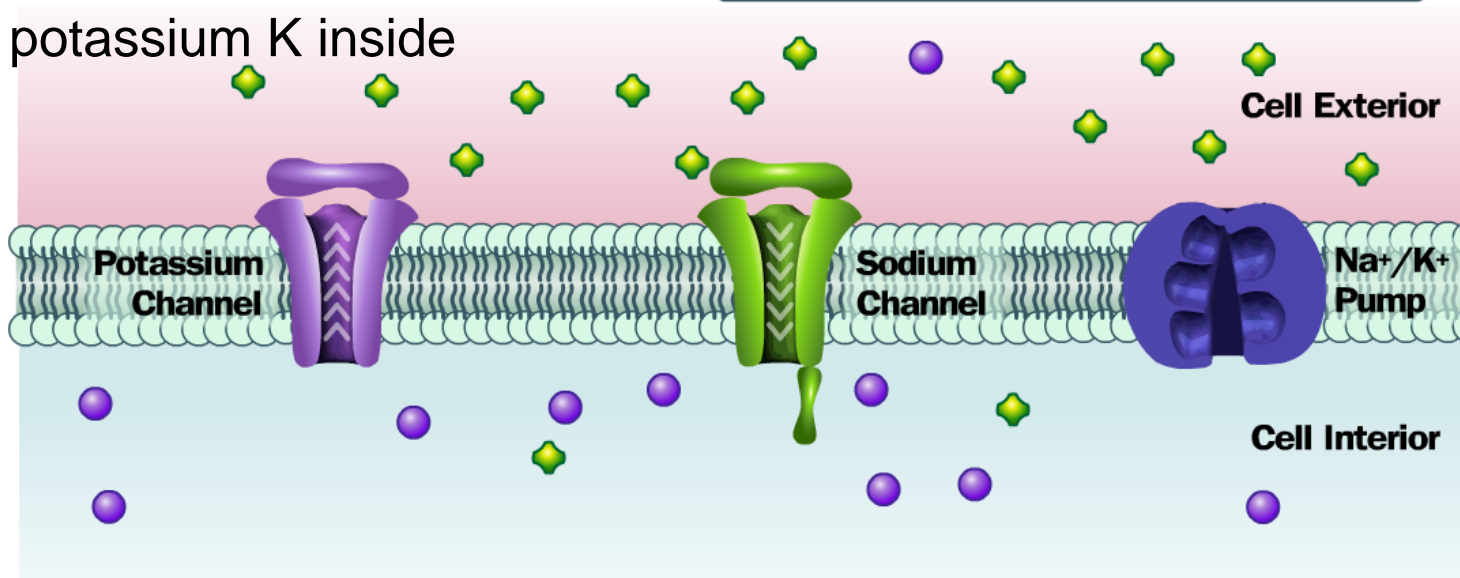
# Action Potential: Resting Potential

## Action Potential

*Introduction*  
*Resting Potential*  
*Depolarization*  
*Repolarization*  
*Return to Resting Potential*  
*Summary of Action Potential*  
*Zoom Out*



sodium Na outside,  
potassium K inside

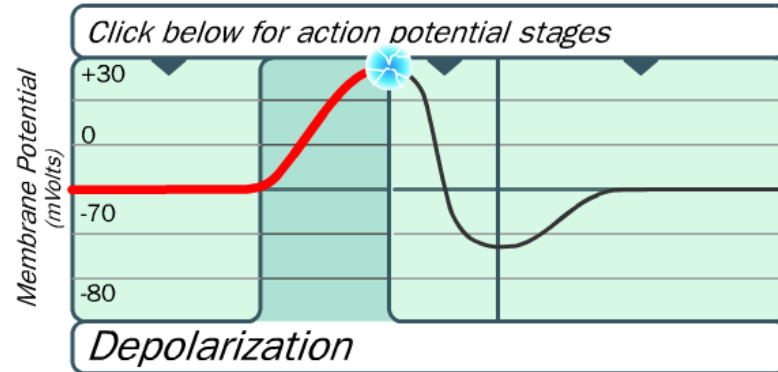


[[http://outreach.mcb.harvard.edu/animations/actionpotential\\_short.swf](http://outreach.mcb.harvard.edu/animations/actionpotential_short.swf)]

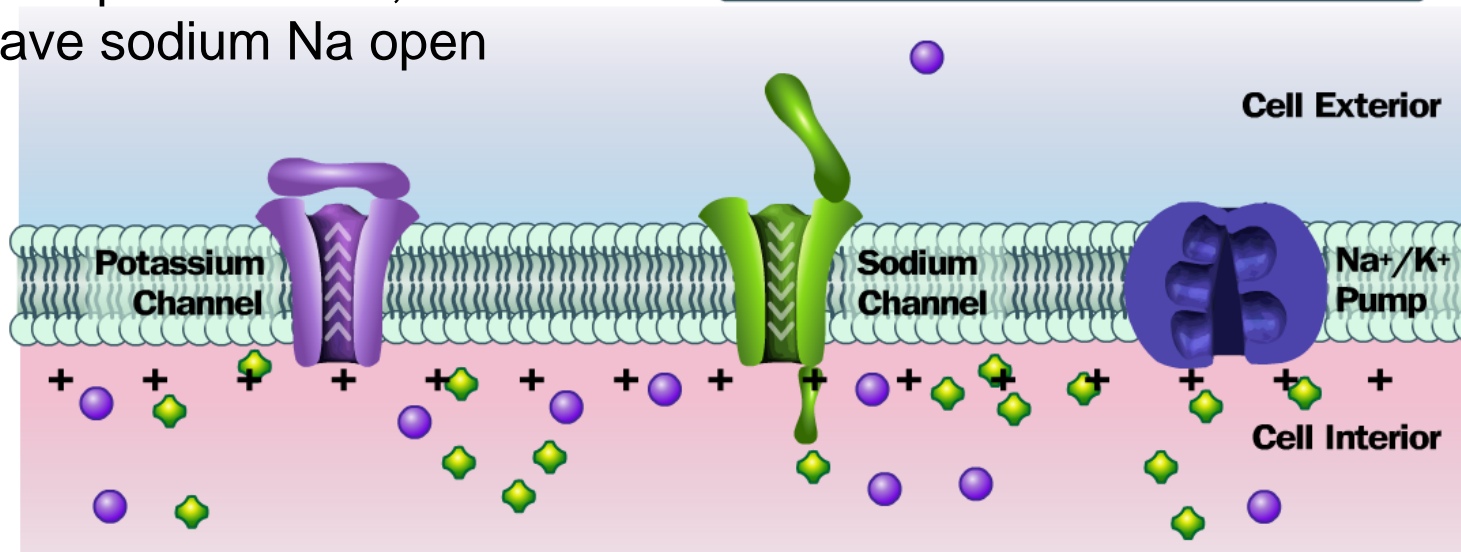
# Action Potential: Depolarization

## Action Potential

*Introduction*  
*Resting Potential*  
*Depolarization*  
*Repolarization*  
*Return to Resting Potential*  
*Summary of Action Potential*  
*Zoom Out*



close potassium K,  
leave sodium Na open



[[http://outreach.mcb.harvard.edu/animations/actionpotential\\_short.swf](http://outreach.mcb.harvard.edu/animations/actionpotential_short.swf)]

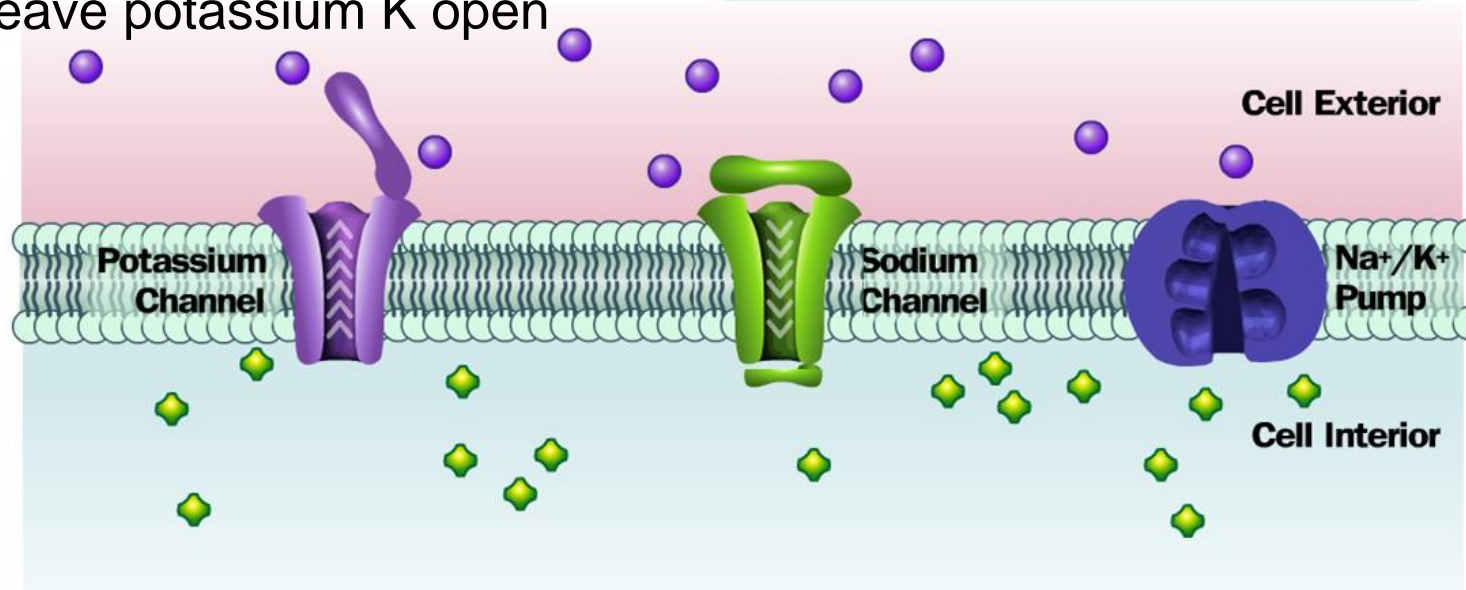
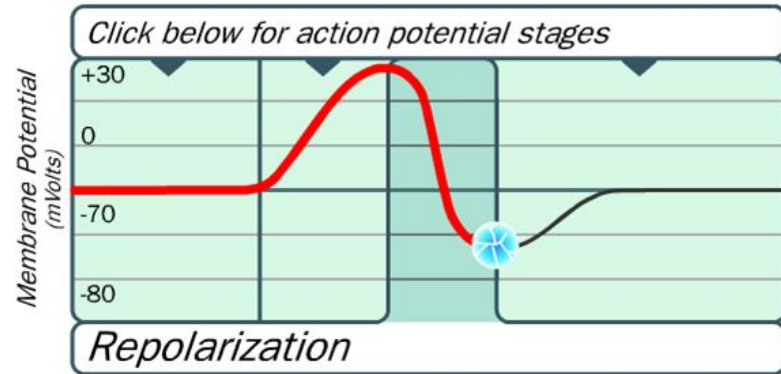


# Action Potential: Repolarization

## Action Potential

*Introduction*  
*Resting Potential*  
*Depolarization*  
*Repolarization*  
*Return to Resting Potential*  
*Summary of Action Potential*  
*Zoom Out*

close Na sodium  
leave potassium K open



[[http://outreach.mcb.harvard.edu/animations/actionpotential\\_short.swf](http://outreach.mcb.harvard.edu/animations/actionpotential_short.swf)]

# Modelling Neurons: Resting Potential

## Action Potential

*Introduction*

*Resting Potential*

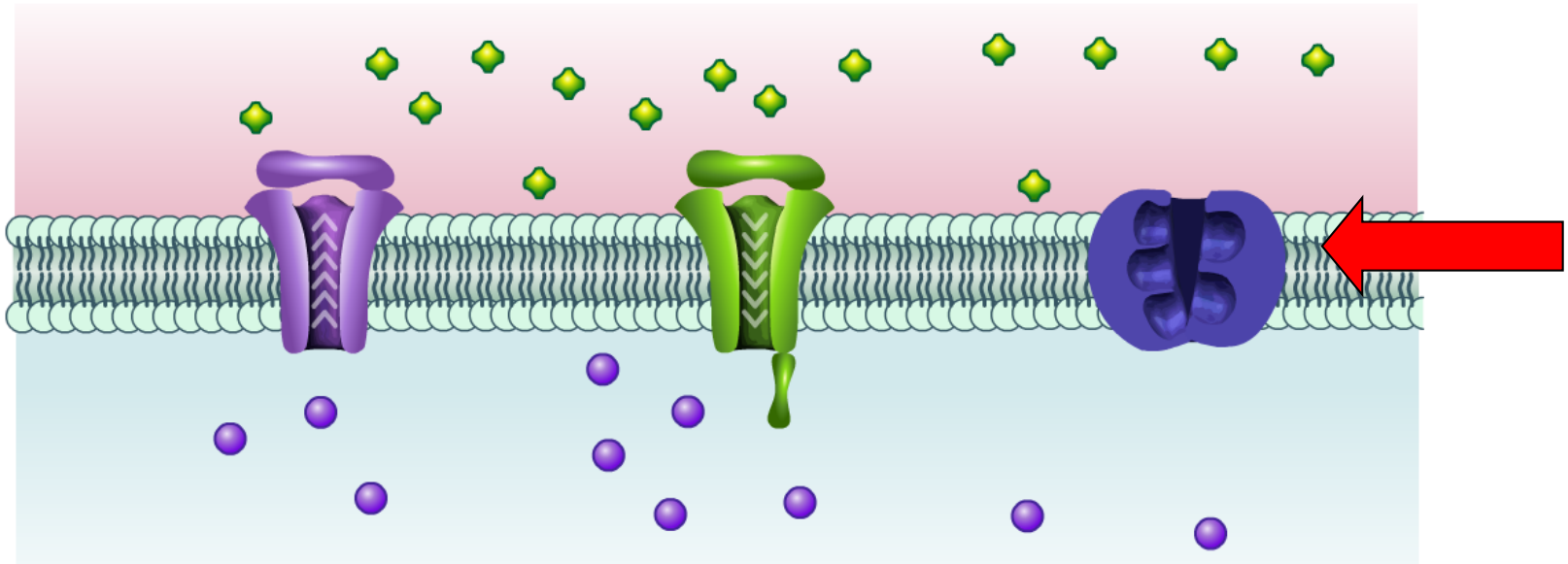
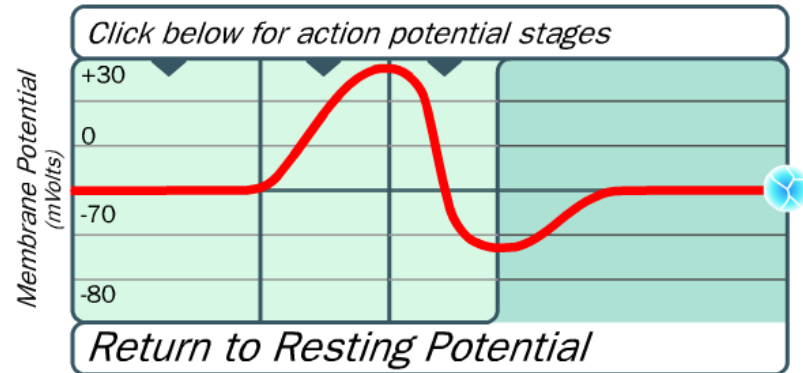
*Depolarization*

*Repolarization*

*Return to Resting Potential*

*Summary of Action Potential*

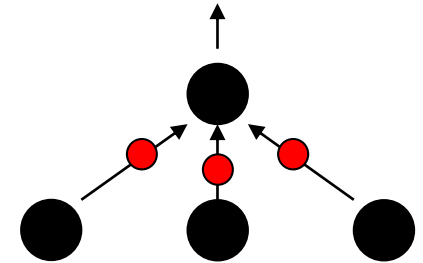
*Zoom Out*



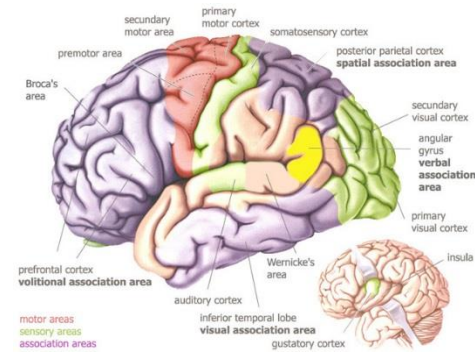
[[http://outreach.mcb.harvard.edu/animations/actionpotential\\_short.swf](http://outreach.mcb.harvard.edu/animations/actionpotential_short.swf)]

# Processing and Learning in the Brain

- Each **neuron receives inputs** from other neurons
  - Some neurons also connect to receptors
  - Cortical neurons use **spikes** to communicate
  - The timing of spikes is important
- Effect of each input line on neuron is controlled by **synaptic weight**
  - Weights can be positive or negative
- Synaptic **weights adapt** so that whole network learns to perform useful computations
  - Recognizing objects, understanding language, making plans, controlling the body
- About  $10^{12}$  neurons each with about  $10^3$  weights

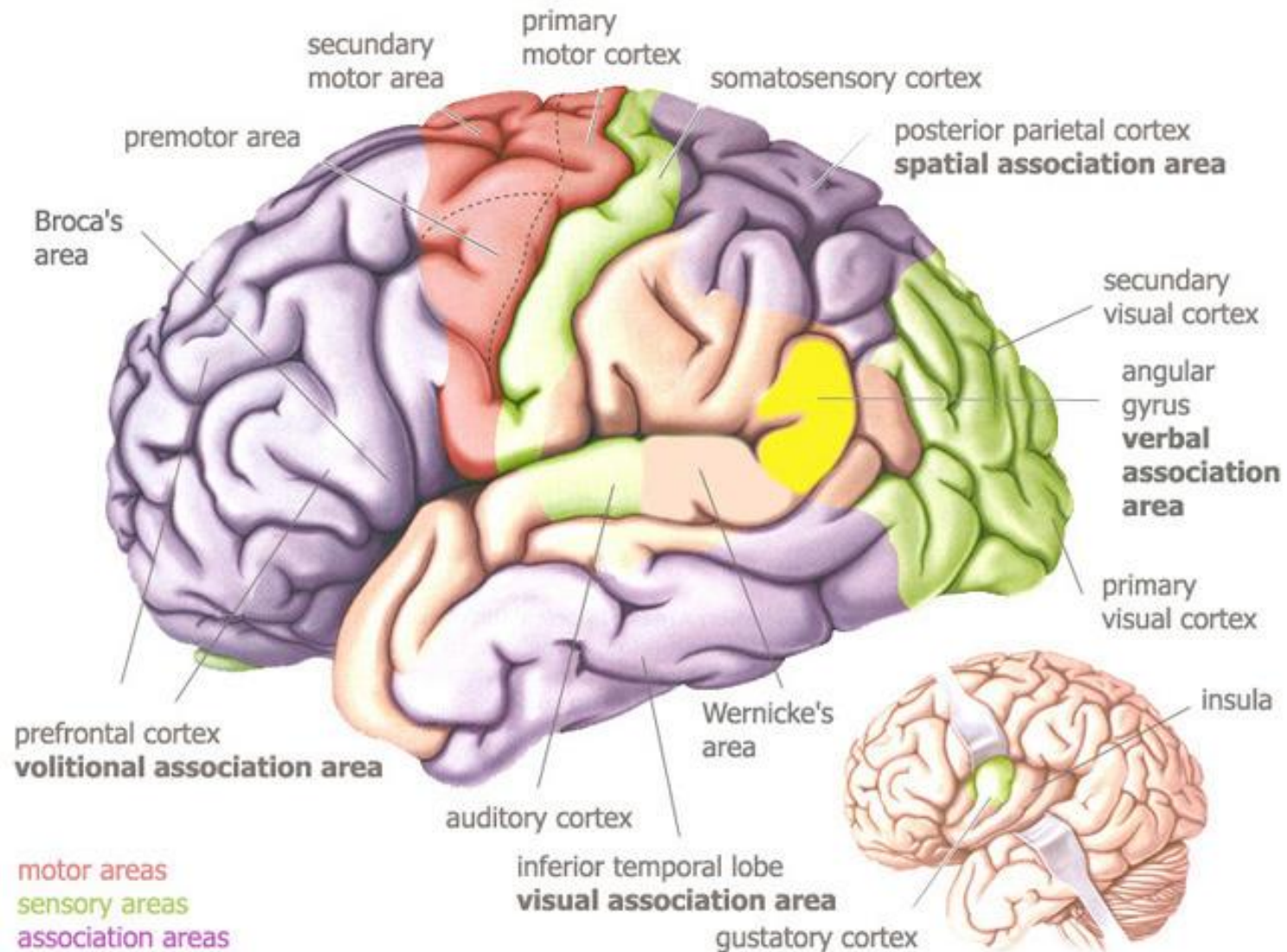


# Modularity and the brain



- Parts of cortex do *different jobs*
  - Local damage to brain has specific effects
  - Specific tasks increase the blood flow to specific regions.
- Cortex and neurons *about the same* everywhere
  - Early brain damage makes functions relocate
- **Cortex** is made of *general purpose hardware* that has ability to turn into special purpose hardware in response to experience
  - This gives rapid *parallel* computation
- **Conventional computers** get flexibility by having *stored programs*
  - Fast central processors that perform large computations *sequentially*

# Modularity and the brain



# Linear neurons

- These are simple but computationally limited
  - If we can make them learn we **may** get insight into more complicated neurons

Bias  
(threshold)

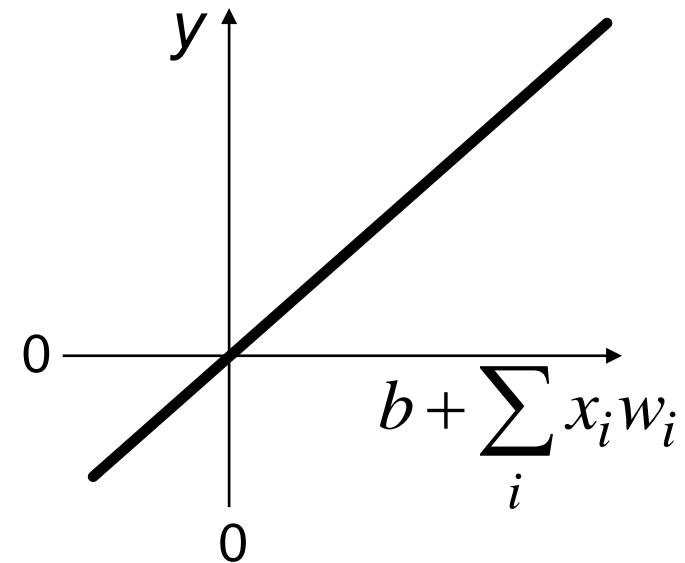
$y = b + \sum_i x_i w_i$

output

$i$   
index over  
input connections

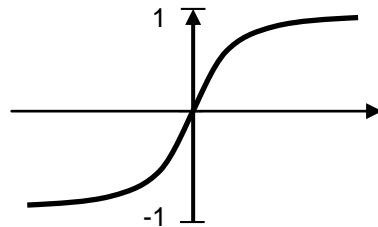
$i^{\text{th}}$  input

weight on  
 $i^{\text{th}}$  input



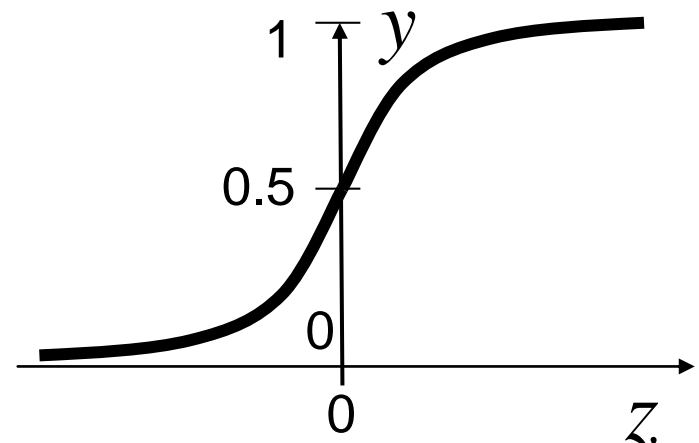
# Sigmoid neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
  - Typically they use the **logistic function**
- You will also often encounter the hyperbolic tangent  **$\tanh(x)$** .
  - It is symmetric at the origin:



$$z = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



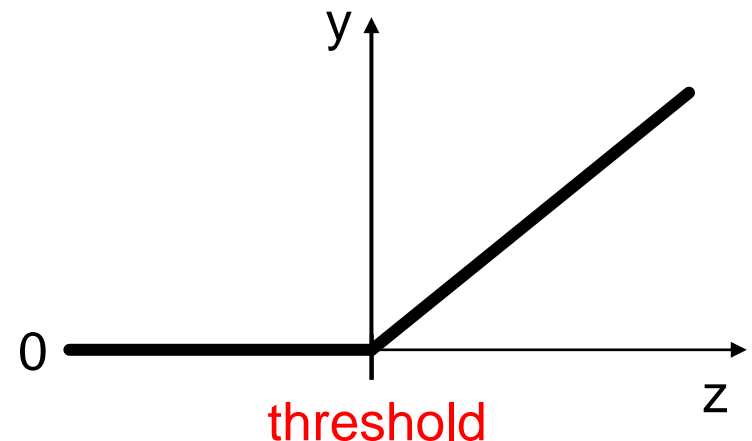


# Rectified Linear Units (ReLU)

- ReLUs are linearly activated but only in the positive range
- **Comparison** to sigmoid units:
  - No “negative”, symmetric activations
  - Faster computation (no exponential function needed)
  - Currently successfully used in Deep Neural Networks

$$z = b + \sum_i x_i w_i$$

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

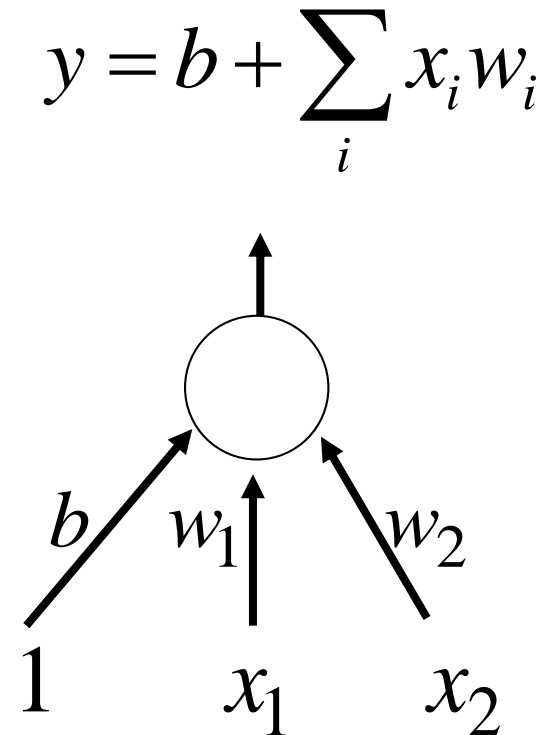




# Adding biases for thresholds

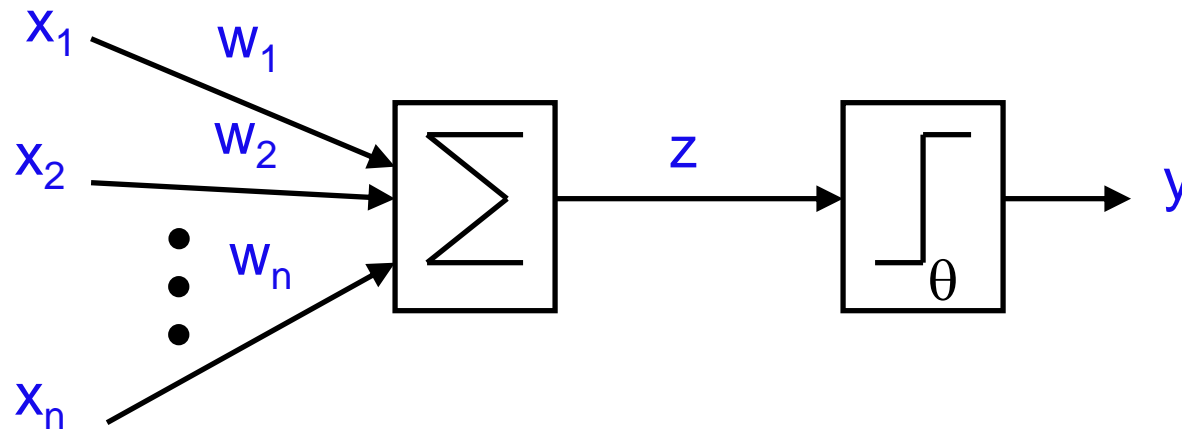
- A linear neuron is **more flexible** if we include a bias for thresholding
- We can avoid having to figure out a separate learning rule for the bias by using a **trick**:
  - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.

Note: The bias is **often omitted** in equations for simplicity!



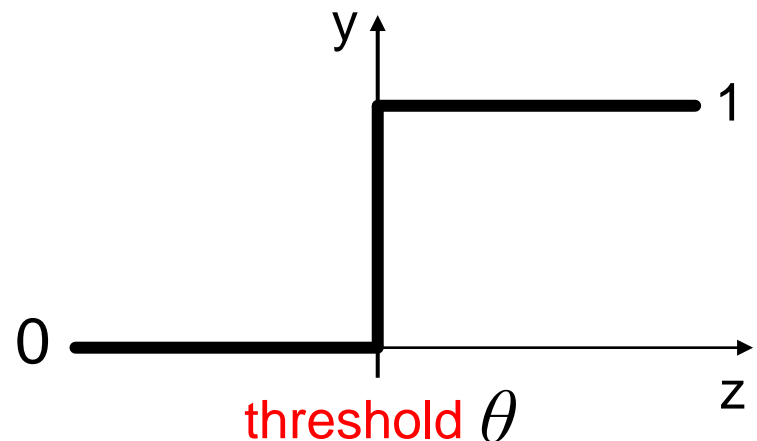
# Perceptron Neurons (McCulloch-Pitts)

- Binary threshold activation:



$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$



# Example for learning with linear neurons

- Each day you get lunch at the cafeteria.
  - Your diet consists of fish, chips, and beer.
  - You get several portions of each
- The cashier only tells you the total price of the meal
  - After several days, you should be able to figure out the price of each portion.
- Each meal price gives a linear constraint on the prices of the portions:

$$price = x_{fish}w_{fish} + x_{chips}w_{chips} + x_{beer}w_{beer}$$

- (→ “credit assignment” problems)

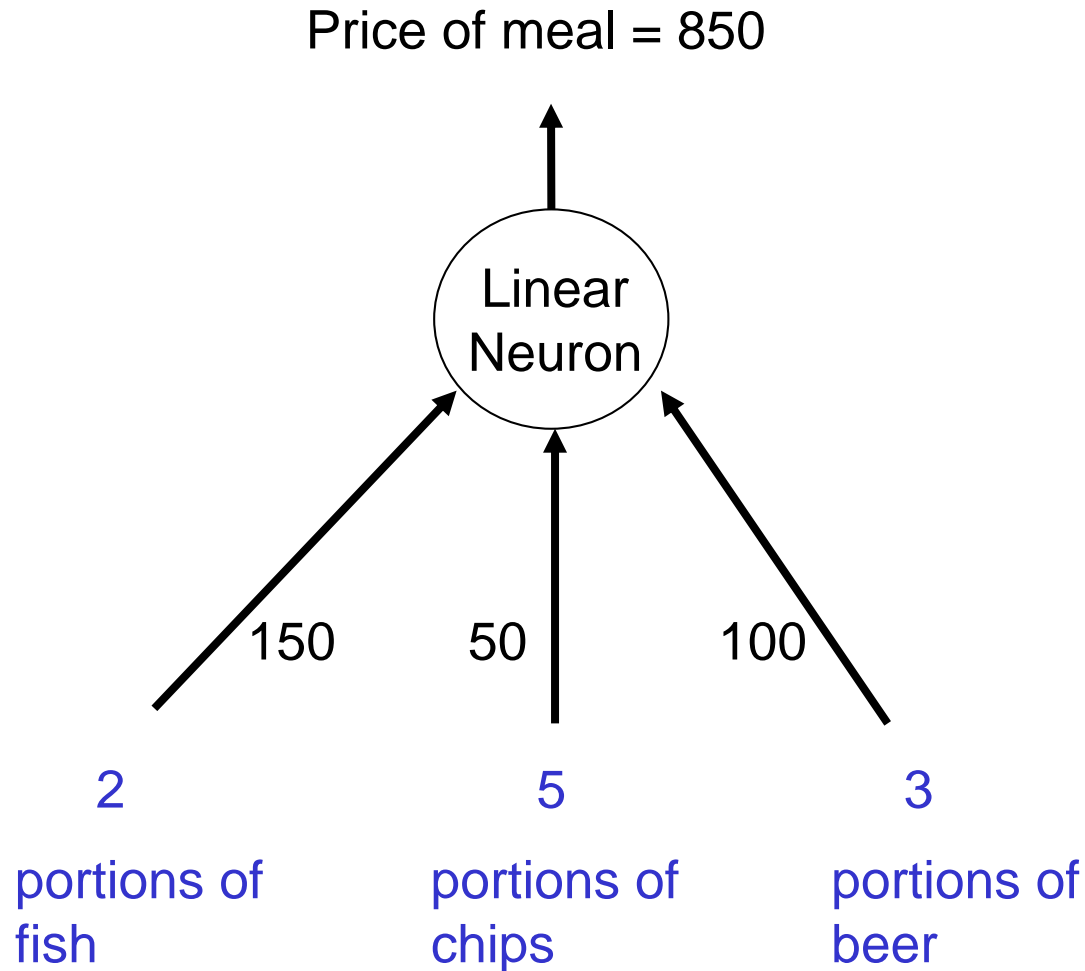
# Two ways to solve the equations

- The obvious approach is just to solve a set of **simultaneous linear equations**, one per meal.
- But we want a method that could be implemented in a **neural network**.
- The prices of the portions are like the weights in a linear neuron.

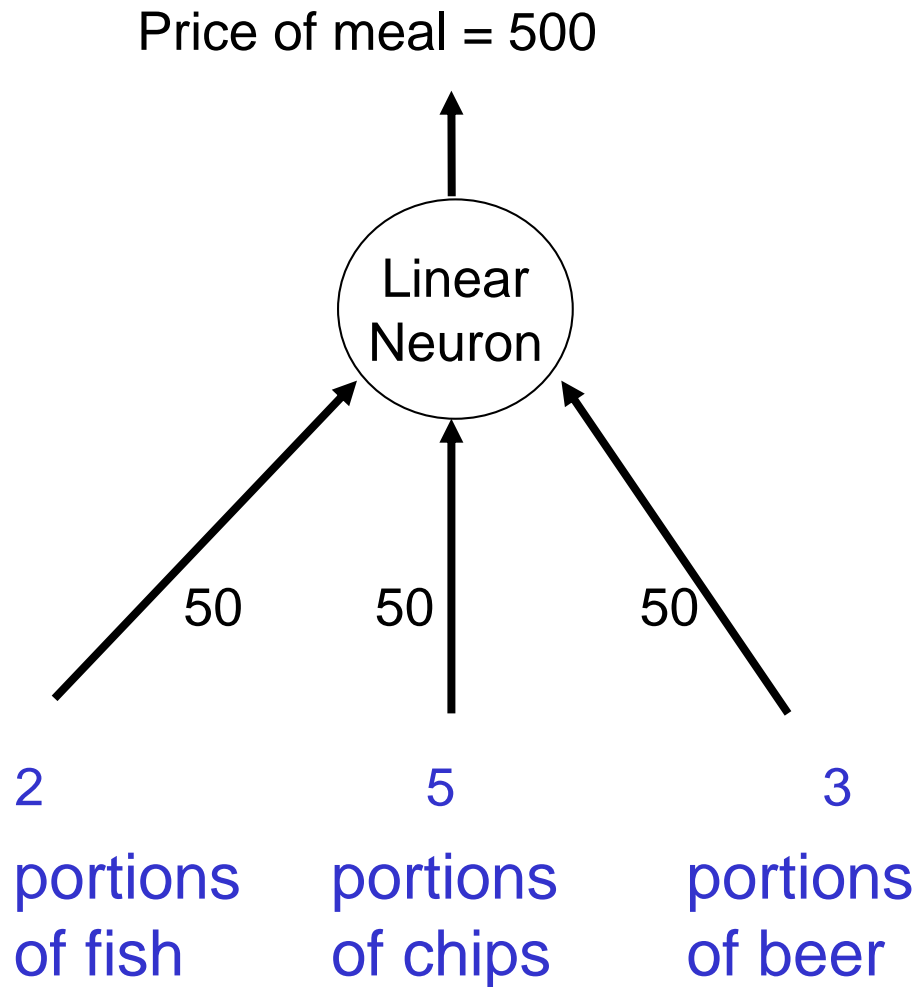
$$\mathbf{W} = (w_{fish}, w_{chips}, w_{beer})$$

- We will start with guesses for the weights and then **adjust the guesses to give a better fit** to the prices given by the cashier.

# The cashier's "brain" with correct weights



# A new initial model of the cashier's brain with arbitrary initial weights



- Residual error = 350
- The learning rule is:

$$\Delta w_i = \eta (t - y) x_i$$

- With a learning rate  $\eta$  of 1/35, the weight changes are +20, +50, +30
- This gives new weights of 70, 100, 80
- Computed price then 880 which is closer to the correct price

# Behaviour of the iterative learning procedure

- Do the updates to individual weights always make them get closer to their correct values?

No! - Notice that the weight for chips got worse!

- Does the online version of the learning procedure eventually get the right answer?

Yes, if the learning rate gradually decreases appropriately

- How quickly do the weights converge to their correct values?

Can be slow if two input dimensions are highly correlated (e.g. ketchup and chips).

- Can the iterative procedure be generalized to much more complicated, multi-layer, non-linear nets? YES! To come...

# Supervised Learning

- Each **training case** consists of an input vector  $x$  and a desired output  $y$ .
  - **Regression**: Desired output is a real number
  - **Classification**: Desired output is a class label (1 or 0 is the simplest case), usually called target  $t$ .
- **Learning** usually means adjusting parameters to reduce the **difference** between **desired output** on each training case and the **actual output** produced by the model.
- Unsupervised Learning: No labels/classes/categories



# Perceptron Network

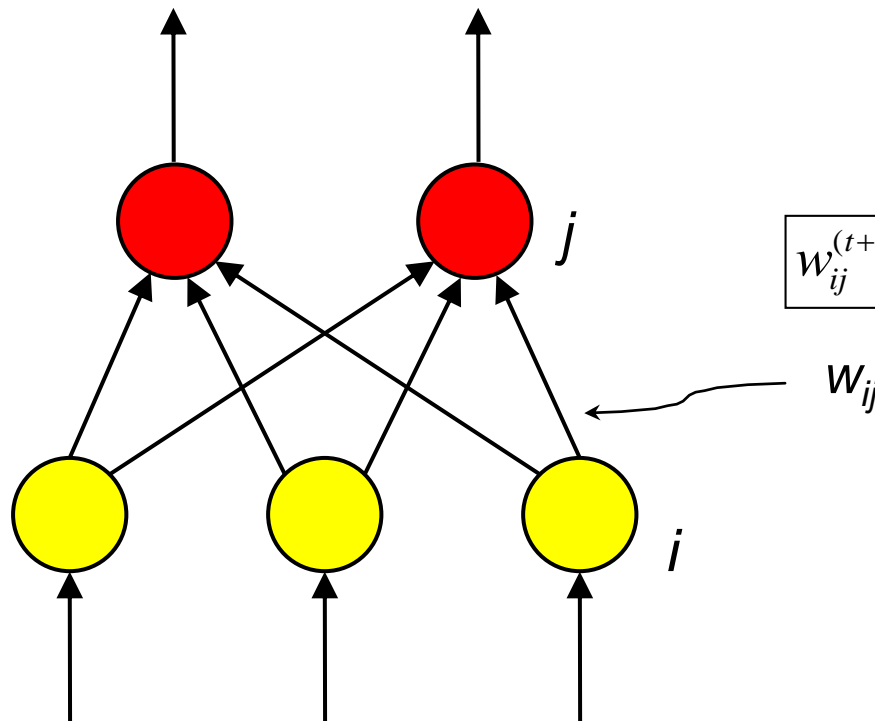
Using more than one neuron (unit):

Output vector:  $y$

Output layer

Input layer

Input vector:  $x$



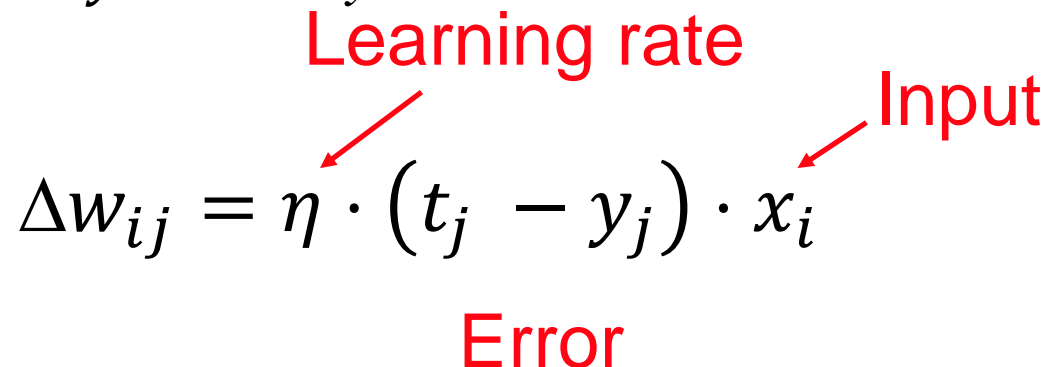
$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta(t_j - y_j)x_i$$

# Updating the Weights

- Delta rule: Gradient descent learning rule for single-layer perceptron

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

- We want to change the values of the weights
- Aim: minimize the **error**  $E$  at the output
- The error is the difference between the target  $t$  (label) and the actual output  $y$ :  $E = t - y$
- Use:



The diagram shows the weight update formula  $\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$ . Red arrows point from the text labels to the corresponding terms in the formula: 'Learning rate' points to  $\eta$ , 'Error' points to  $(t_j - y_j)$ , and 'Input' points to  $x_i$ .

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

# Perceptron Algorithm

- Initialisation: set all weights to small positive and negative random numbers
- For  $T$  iterations
  - Choose a new data point  $(\mathbf{x}, \mathbf{t}) = (\text{input}, \text{target})$
  - Compute the output activation of each neuron  $j$

$$z_j = \sum_{i=1}^n x_i w_{ij} \quad y_j = \begin{cases} 1 & z_j \geq \theta \\ 0 & z_j < \theta \end{cases}$$

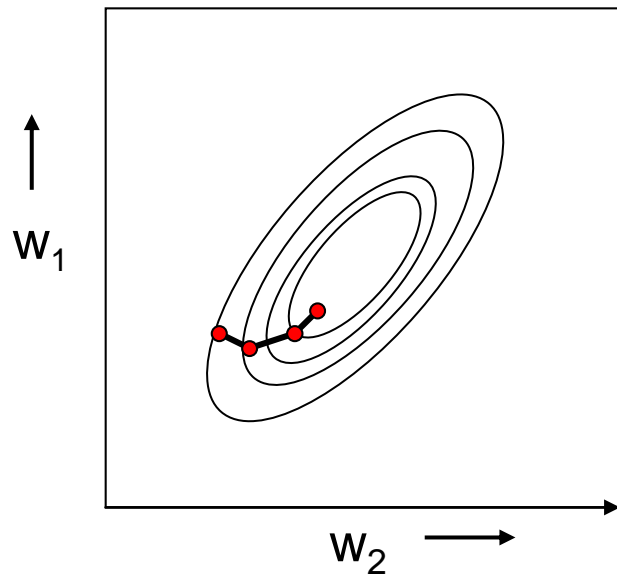
- Update each of the weights according to

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

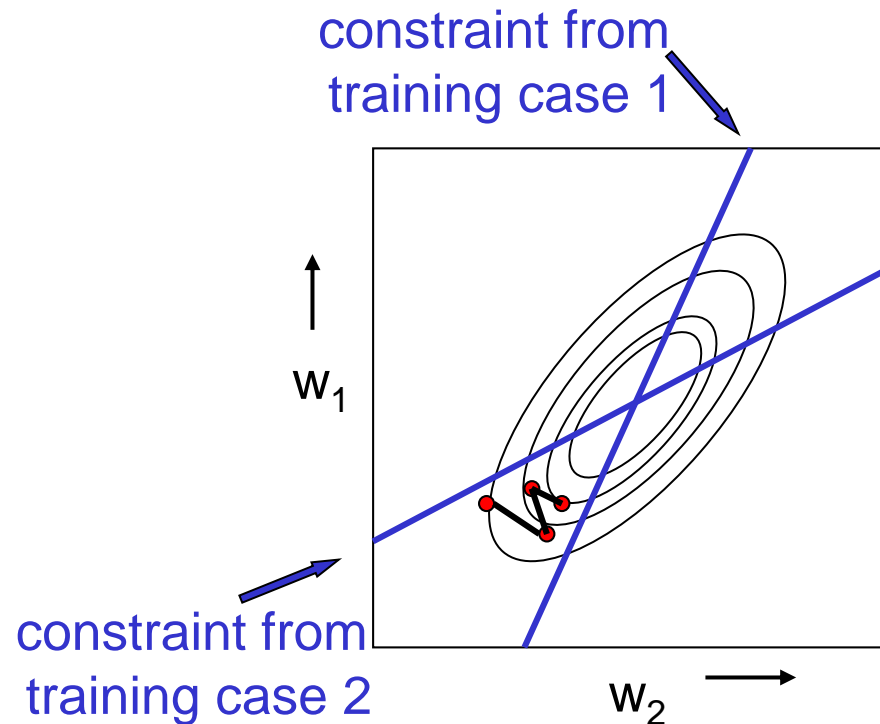
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

# Batch versus online (incremental) learning

- **Batch** learning does steepest descent on the error surface
- Update after whole epoch over all training cases

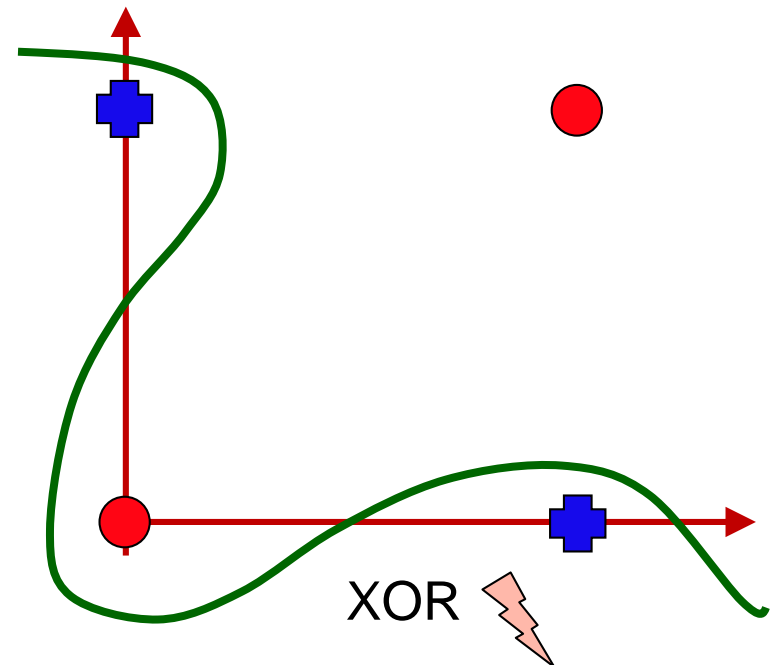
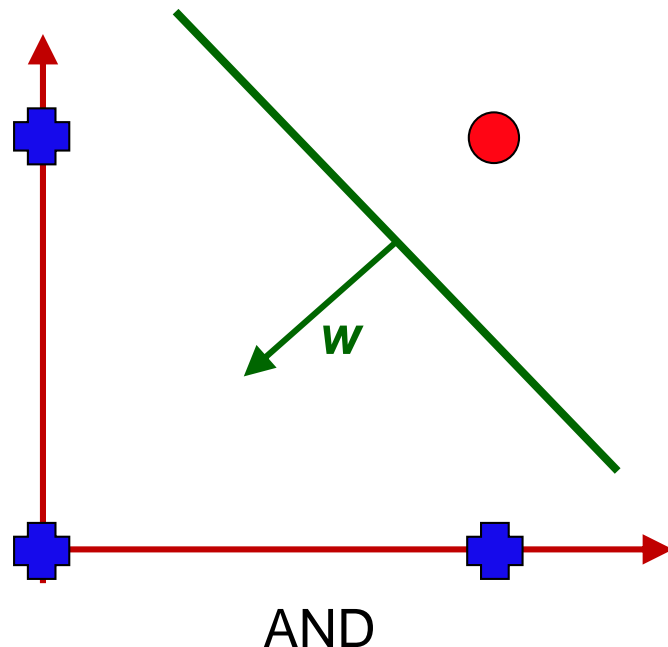


- **Online** learning zig-zags around the direction of steepest descent

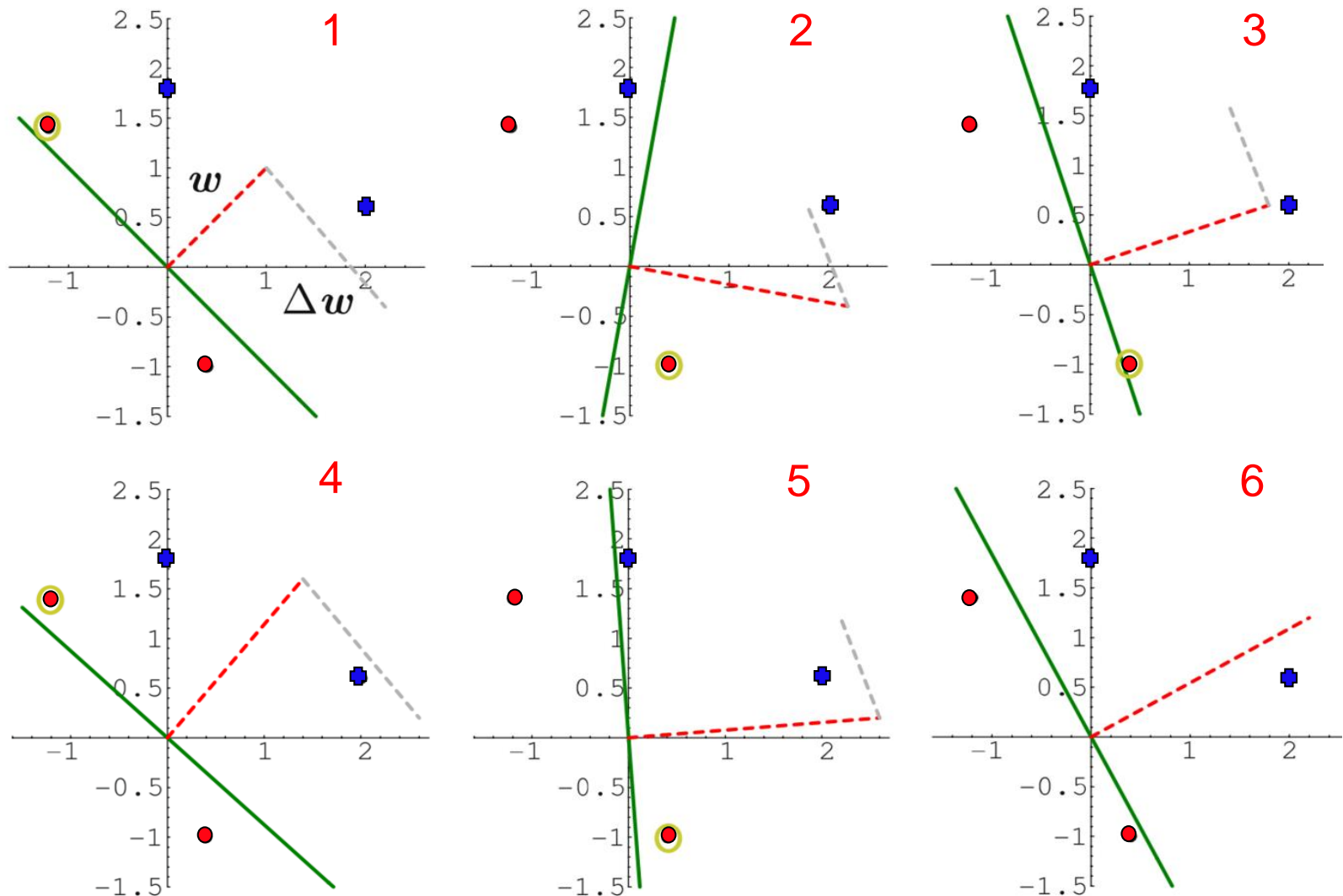


# The Perceptron as a linear classifier

- The update algorithm learns a *straight line* (decision boundary) that can separate the classes.
- It cannot learn classes if the data is not linearly separable:



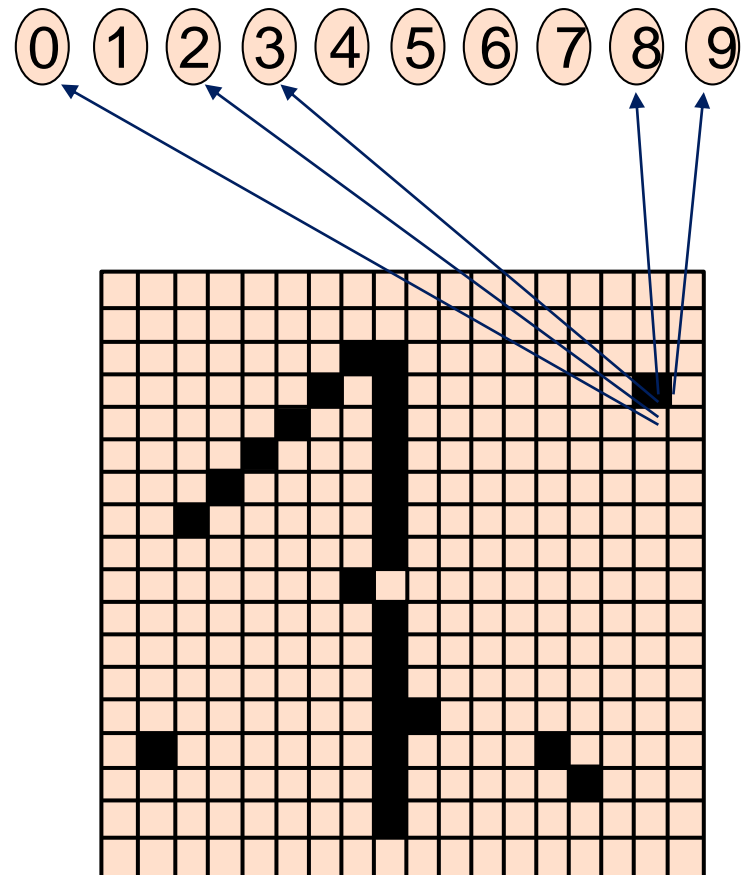
# Perceptron algorithm visualized for red/blue



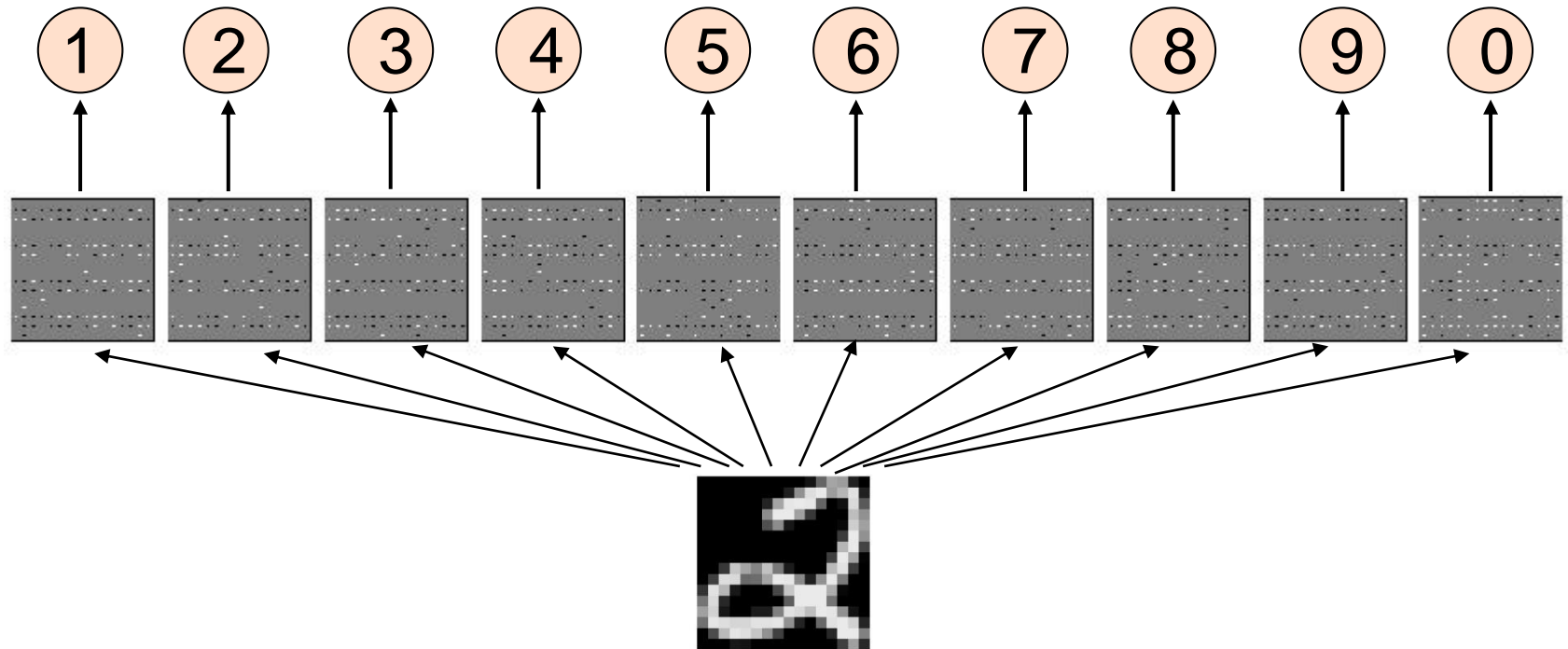
■ Visualisation based on Ertel 2009: Red and blue classes; yellow: wrongly classified

# A very simple and naive way to recognize handwritten shapes

- Consider neural network with two layers of neurons.
  - neurons in the **top** layer represent **known shapes**.
  - neurons in the **bottom** layer (input) 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



# Hinton Diagram: Displaying the weights



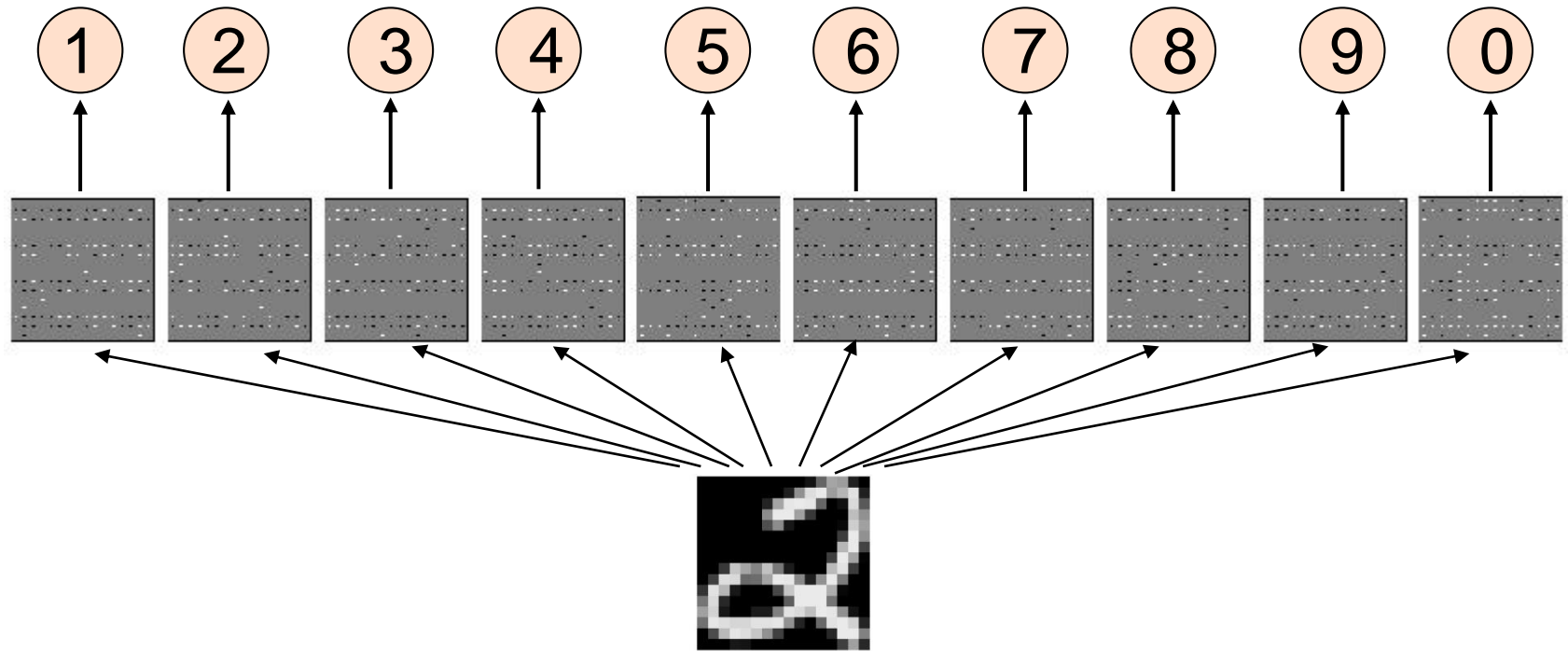
The image

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.

Box-size shows the magnitude. Black/white the sign.



# Learning the weights

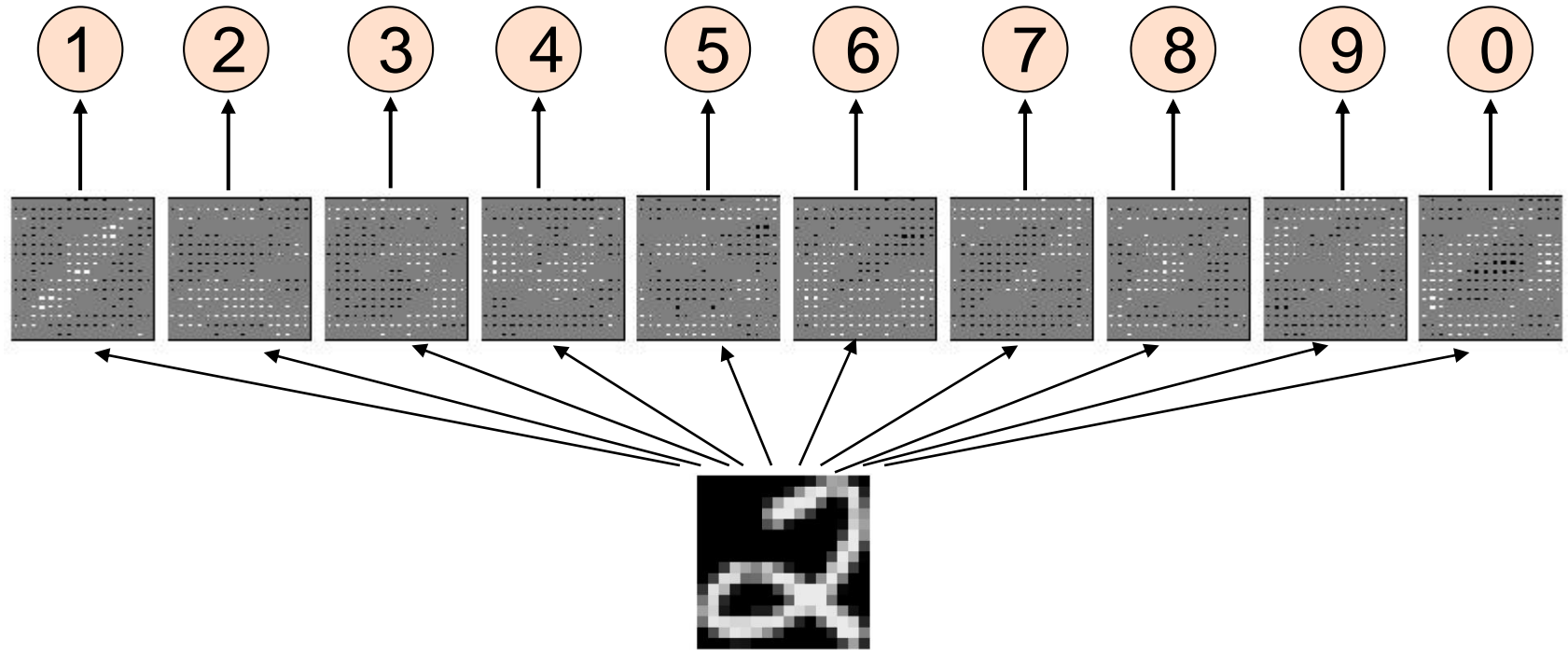


The image

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

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

# Learning the weights

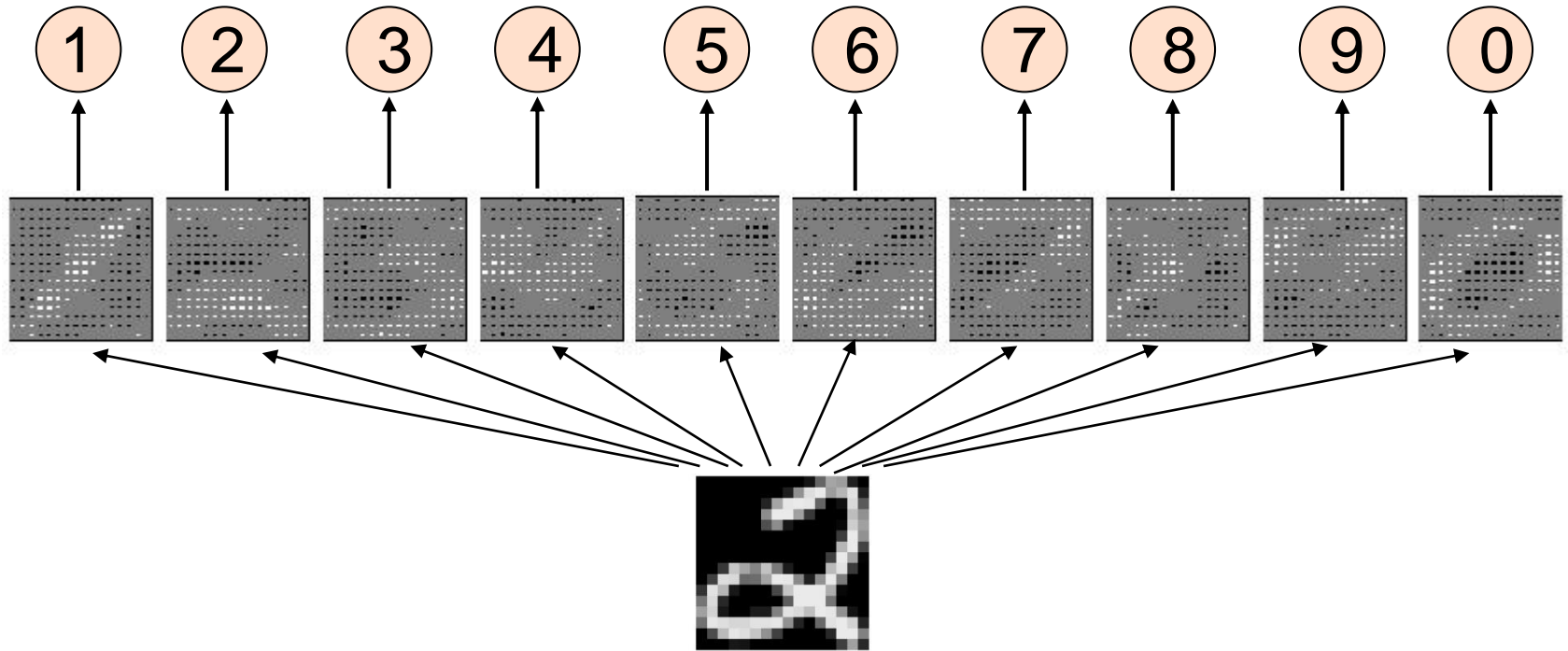


The image

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

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

# Learning the weights

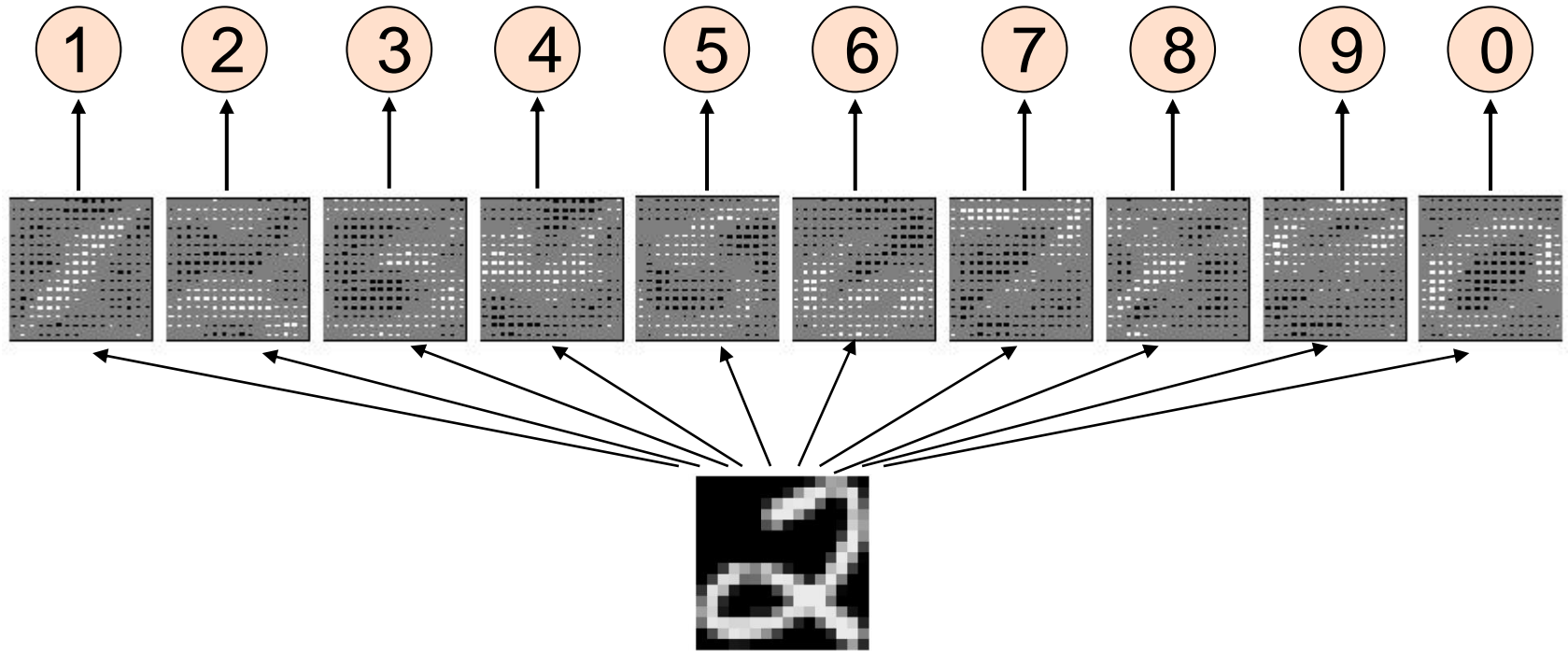


The image

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

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

# Learning the weights

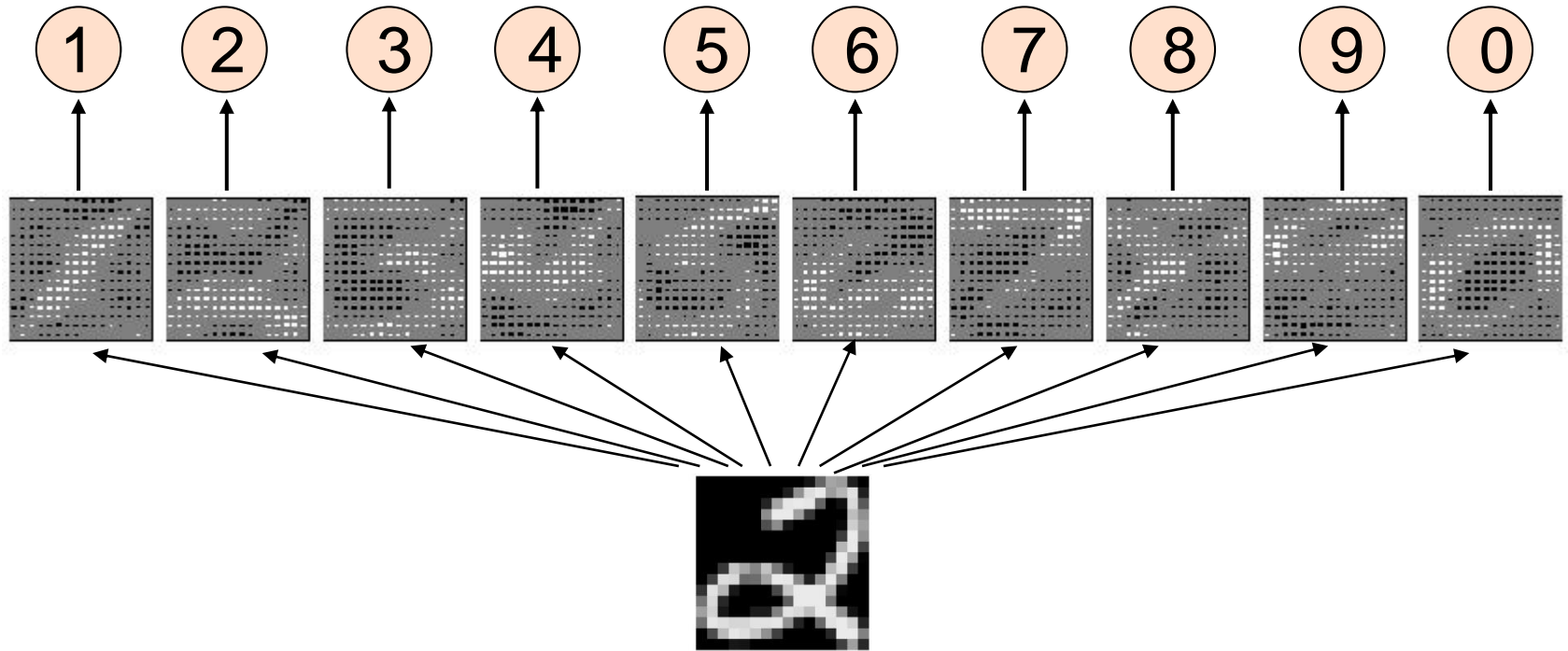


The image

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

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

# Learning the weights

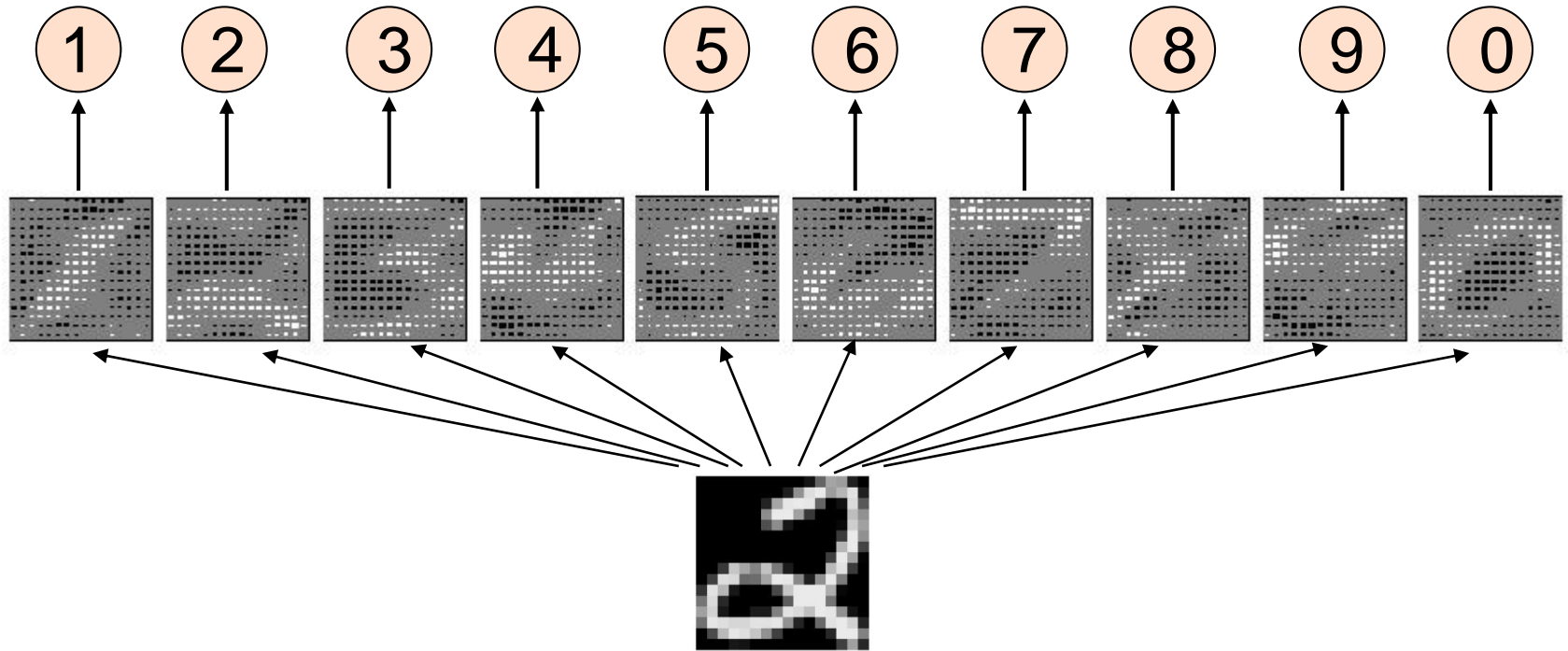


The image

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

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

# Learning the weights

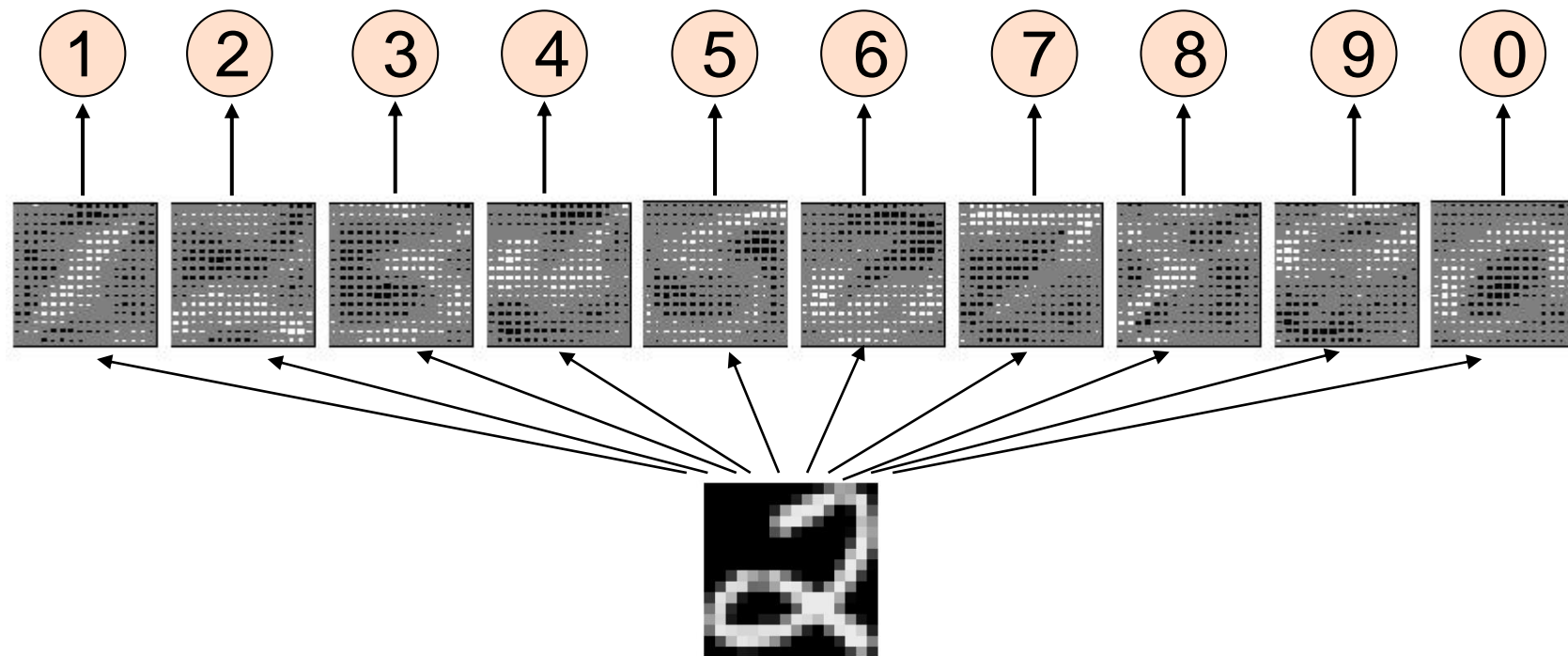


The image

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

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

# The learned weights



The image

# Why this simple system does not work

- 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 *shapes vary* are much too complicated to be captured by simple template matching on the raw data (noise!)
  - To robustly capture all possible shape variations we need to *learn and input features* instead of the raw data



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

0 0 0 1 1 1 1 1 1 2

2 2 2 2 2 2 2 3 3 3

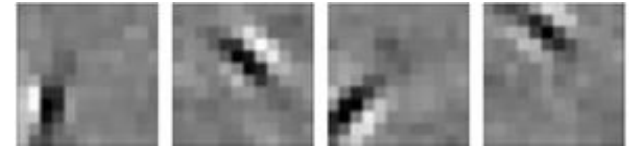
3 4 4 4 4 4 5 5 5 5

6 6 7 7 7 7 7 8 8 8

9 9 9 9 9 9 9 9 9

# Preprocessing the input vectors

Instead of trying to predict the answer directly from the raw inputs we can ...



- *extracting a layer of “features”*

- Sensible if we already know that certain combinations of input values useful (e.g. edges or corners in an image).

-1	0	+1
-1	0	+1
-1	0	+1

+1	+1	+1
0	0	0
-1	-1	-1

- *designing them by hand*

- Hand-coded features are equivalent to a layer of non-linear neurons that do not need to be learned

# Is preprocessing forbidden or effective?

- It is certainly ok if we **learn** the preprocessing
  - This makes learning more difficult, but more interesting...
- It is ok if we use a very big set of non-linear features that is **task-independent** (using e.g. unsupervised learning).
- Nevertheless it can be effective to introduce **domain knowledge into the network**
- We can also learn features with the neural network (Deep Learning)

# Recap of Today's Lecture

- Biological Action Potentials
- Activation Functions
- Perceptron
- Single-Layer Perceptron and Delta Rule
- Applied to simple problems
  - Quickly got to limitations in both complex (feature learning) and simple issues (learning XOR)
  - Next week: Making Perceptrons more powerful
  - Throughout lecture: Integrating techniques for feature learning, end-to-end training

# Reading

- Background
  - Books: Marsland (Chapter 2), Rojas (partly Chapter 2-3)
  - Linear Algebra background: Goodfellow (Part I, Chapter 2)
  - CommSy
- Knowledge Technology website  
<http://www.knowledge-technology.info/>
- *Voluntary exercise you should be able to solve (Marsland – Problem 2.1):*

Consider a neuron with 2 inputs, 1 output, and a threshold activation function. If the two weights are  $w_1 = 1$  and  $w_2 = 1$ , and the bias is  $b = -1.5$ , then what is the output for input (0,0)? What about for inputs (1,0), (0,1), and (1,1)?