

Pattern Analysis and Recognition

Lecture 10: Neural Networks

Resources

See the following sources for further information:

C. Bishop, *“Pattern Recognition and Machine Learning”*, Springer, 2006

Some related material available:

<http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm>

D. MacKay, *“Information Theory, Inference and Learning Algorithms”*, Cambridge University Press, 2003.

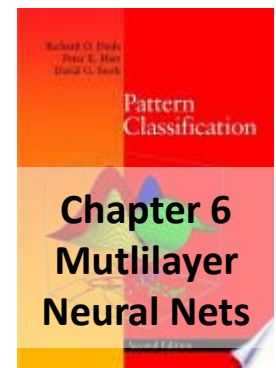
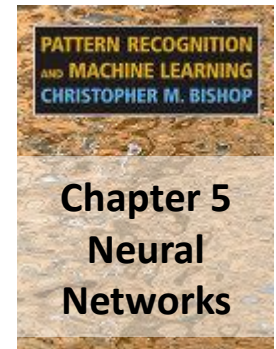
Book available online:

<http://www.inference.phy.cam.ac.uk/mackay/>

R.O. Duda, P.E. Hart, D.G. Stork, *“Pattern Classification”*, Wiley & Sons, 2000

Have a look inside at selected chapters:

http://books.google.es/books/about/Pattern_Classification.html?id=Br33IRC3PkQC&redir_esc=y



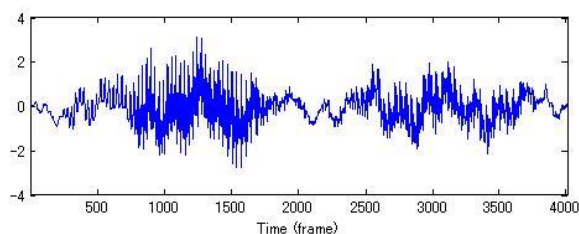
NEURAL NETWORKS INTRO

What have we learnt up to now?

$$y = \text{sgn}(\mathbf{w}^T \mathbf{x} + \mathbf{b})$$

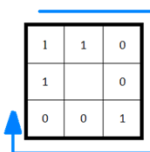
Machine learning solved: it's all about finding the right weights \mathbf{w}

With one handicap: this only works *IF* you have chosen correctly your features \mathbf{x}



9	1	4	2	6
7	8	9	2	7
6	6	5	3	3
8	1	4	7	1
4	6	2	1	3

LBP

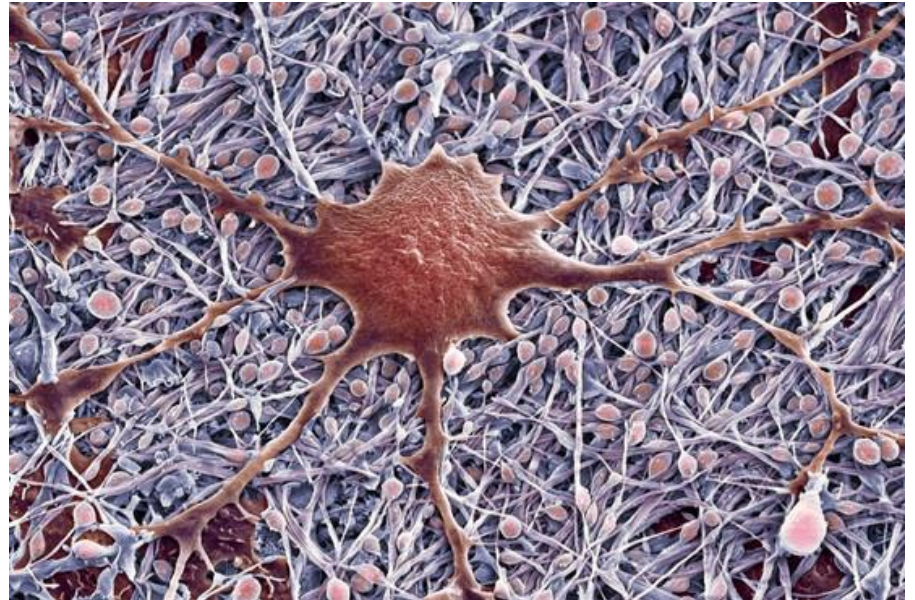
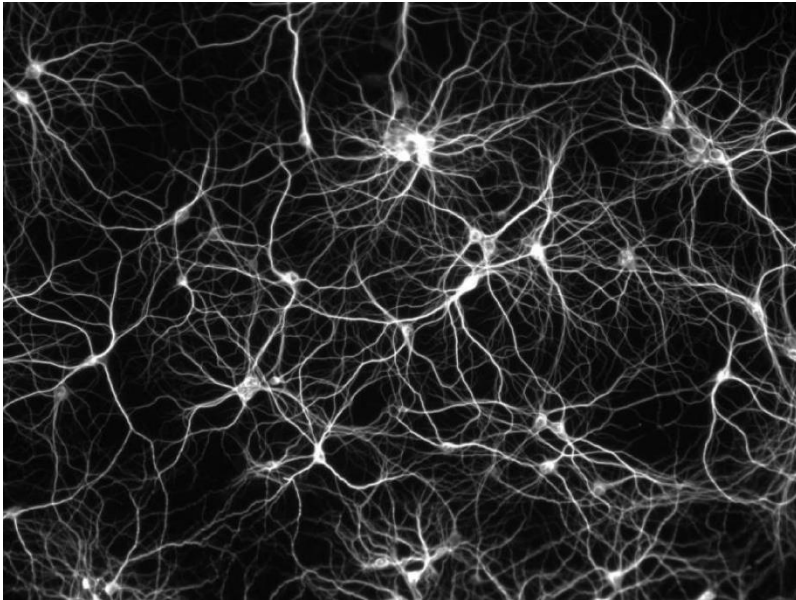


1	1	0
1	0	0
0	0	1



It would be nice to be able to “learn” what are the best features to use automatically: come up with a universal (not domain-dependent) algorithm that is able to represent any kind of the data in the best way for each problem.

Neurons



The human brain comprises about 10^{11} neurons each of which has 10^4 synapses

A hugely parallel machine that has an effective bandwidth to stored knowledge much better than a modern workstation

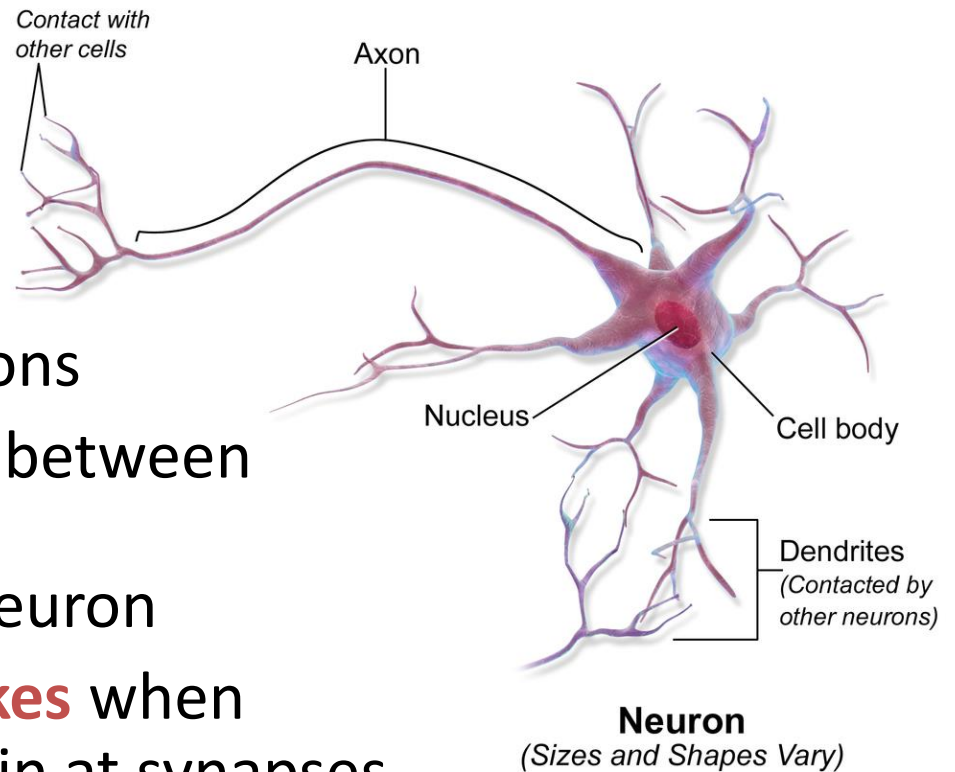
Neurons

Axon: branches and sends messages to other neurons

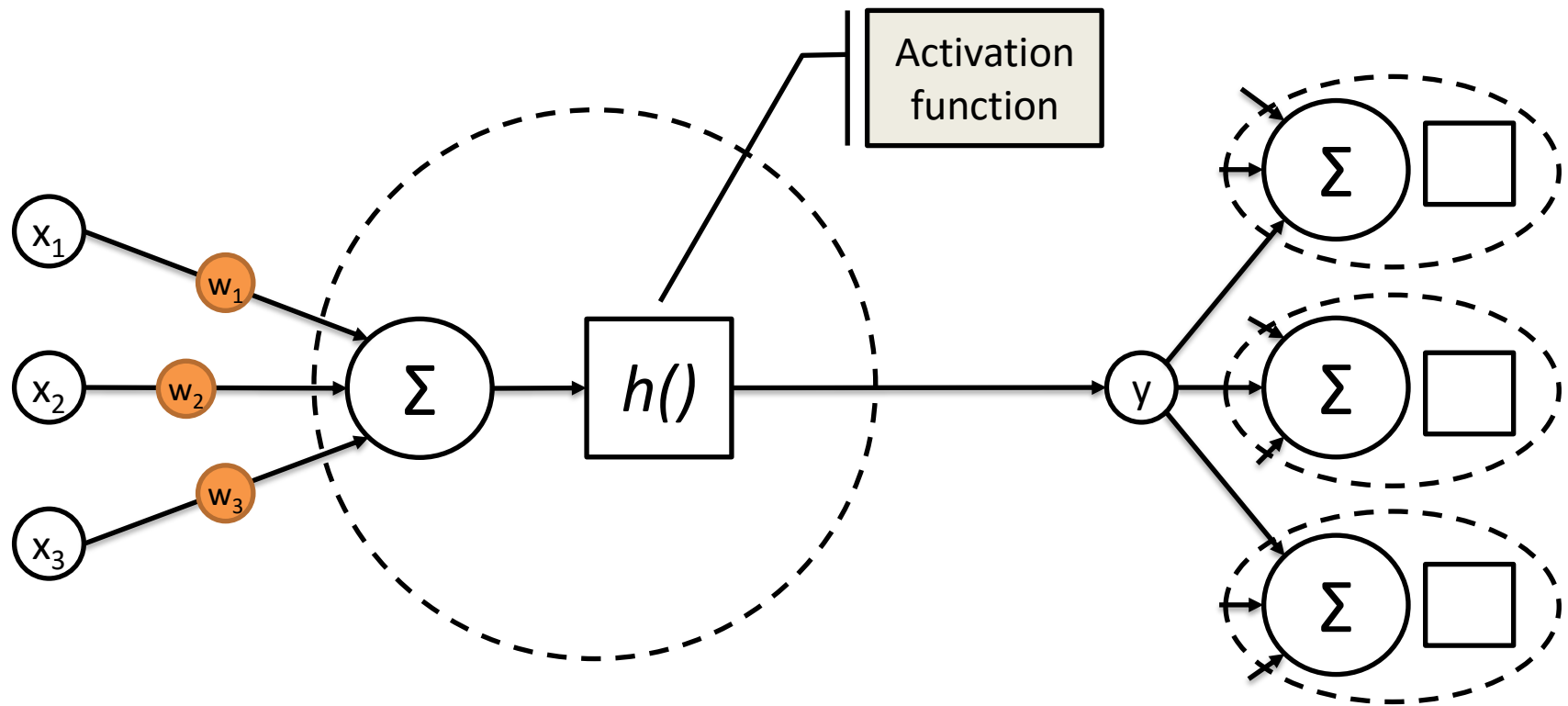
Dendritic tree: receives messages from other neurons

Synapses: are connections between an axon of a neuron and a dendritic tree of another neuron

A neuron can generate **spikes** when enough charge has flowed in at synapses (depolarise the “*axon hillock*”)

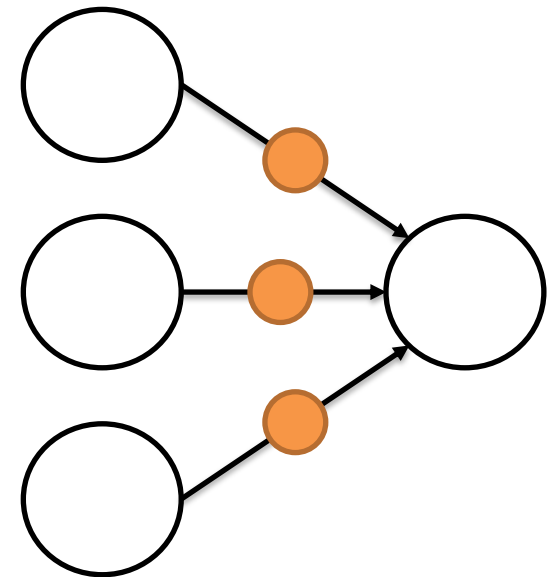


A computational model of an idealised neuron



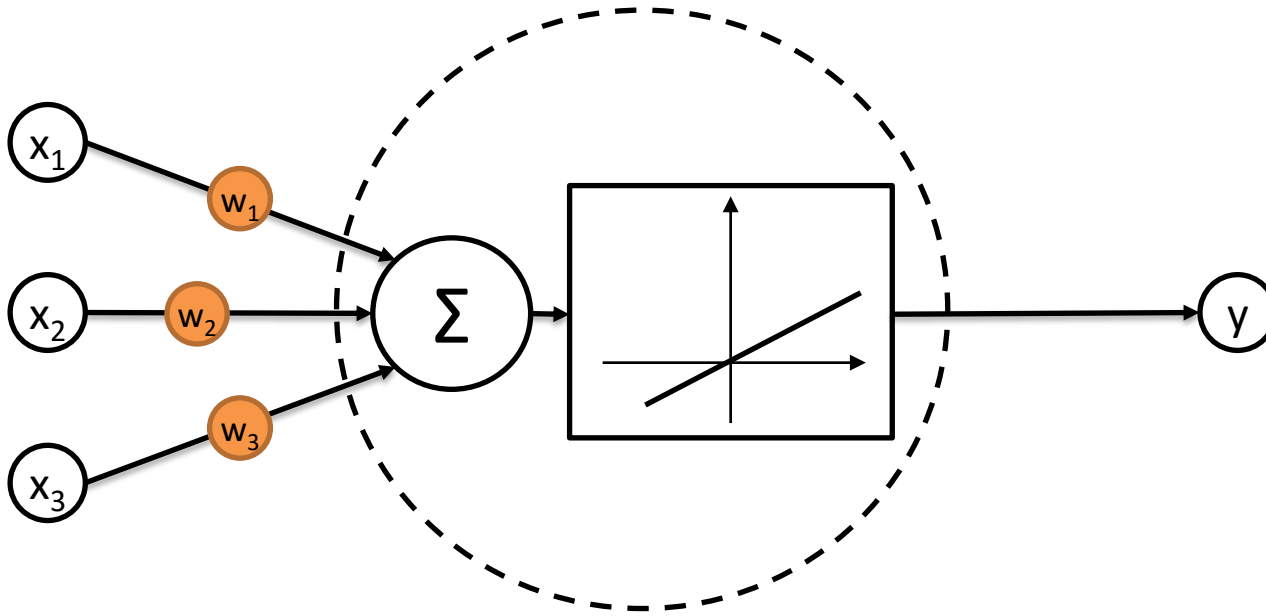
How the brain works on one slide

- Each neuron receives inputs from other neurons and might emit a signal to other neurons
 - A few neurons also connect to receptors
- 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
 - Varying the number of vesicles of transmitter
 - Varying the number of receptor molecules
- The human brain comprises about **10^{15} synapses** (weights)
 - A huge number of weights can affect the computation in a very short time. Much better bandwidth than a workstation.



BASIC NEURON MODELS

Linear Neurons

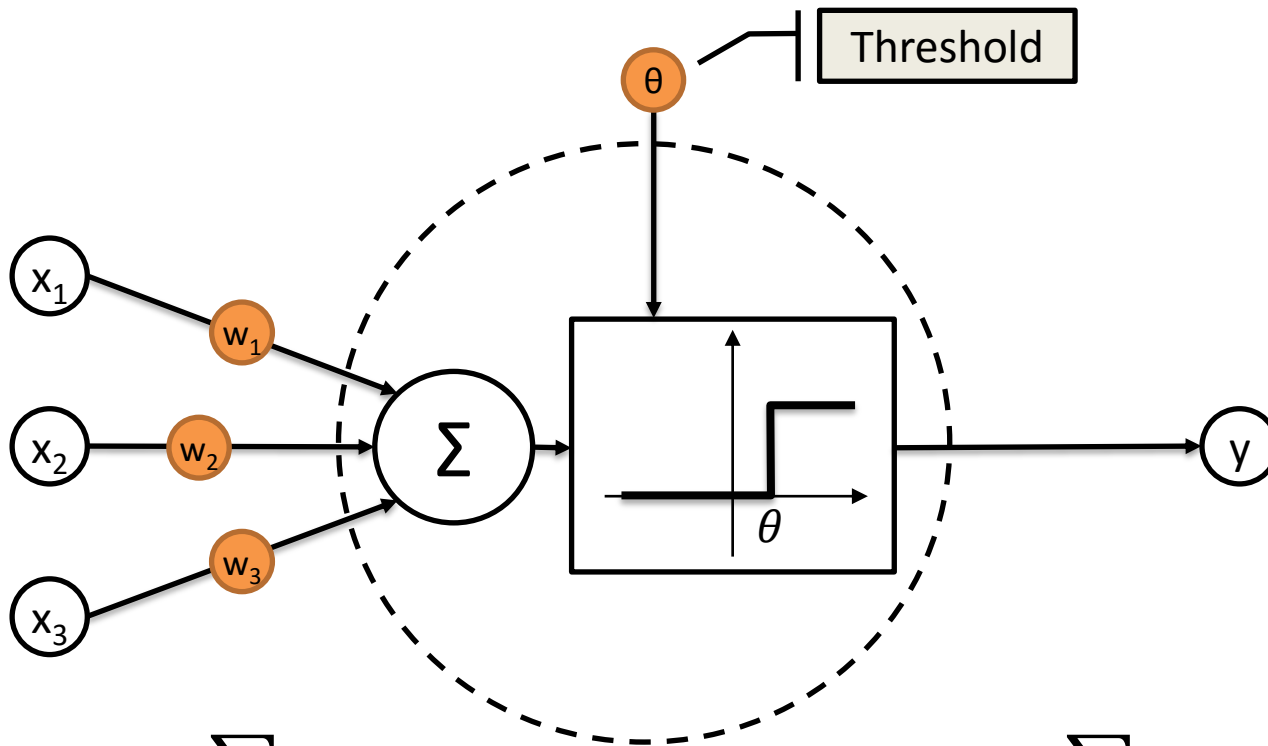


$$y = b + \sum_i w_i x_i$$

Diagram illustrating the equation $y = b + \sum_i w_i x_i$ with labels for its components:

- output**: points to y
- bias**: points to b
- i^{th} input**: points to x_i
- weight on i^{th} input**: points to w_i

Binary Threshold Neurons (McCulloch-Pitts – 1943)



$$z = \sum_i w_i x_i$$

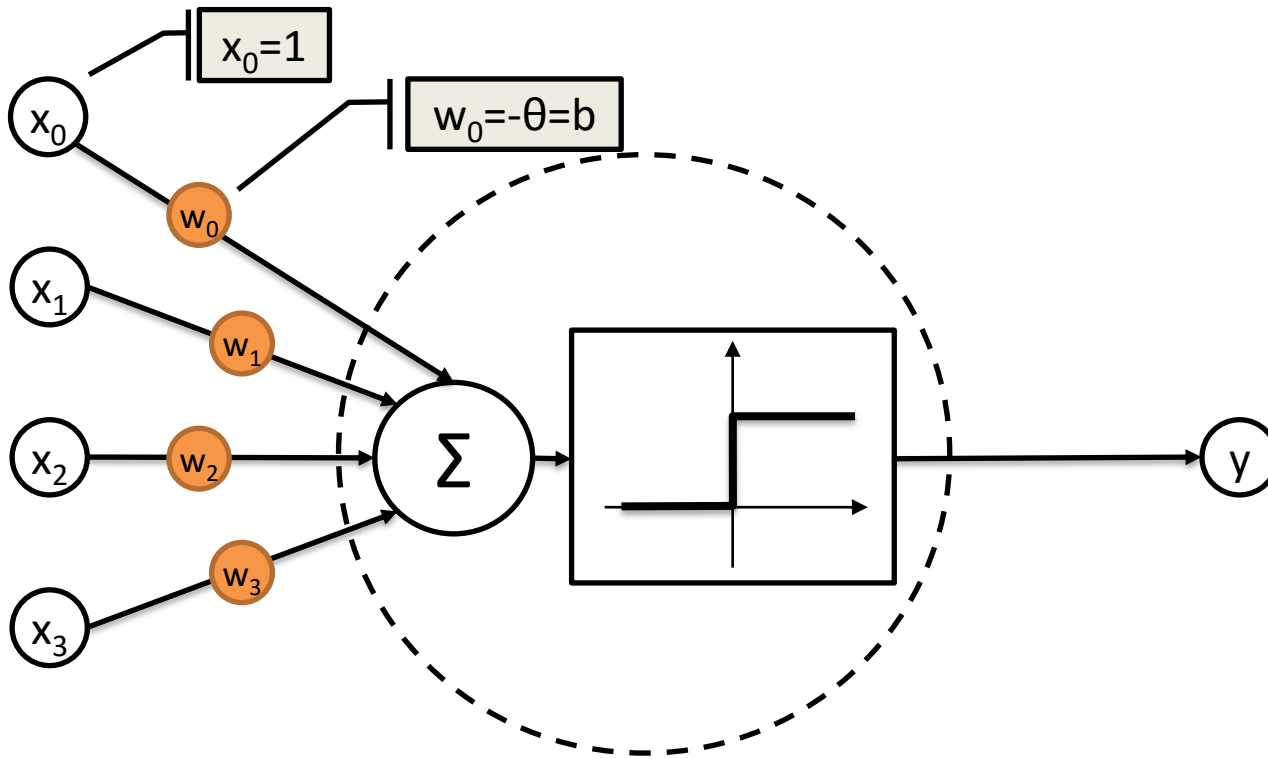
$$y = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$\theta = -b \quad \longrightarrow$$

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

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

Binary Threshold Neurons (McCulloch-Pitts – 1943)



$$w_0 = -\theta = b$$

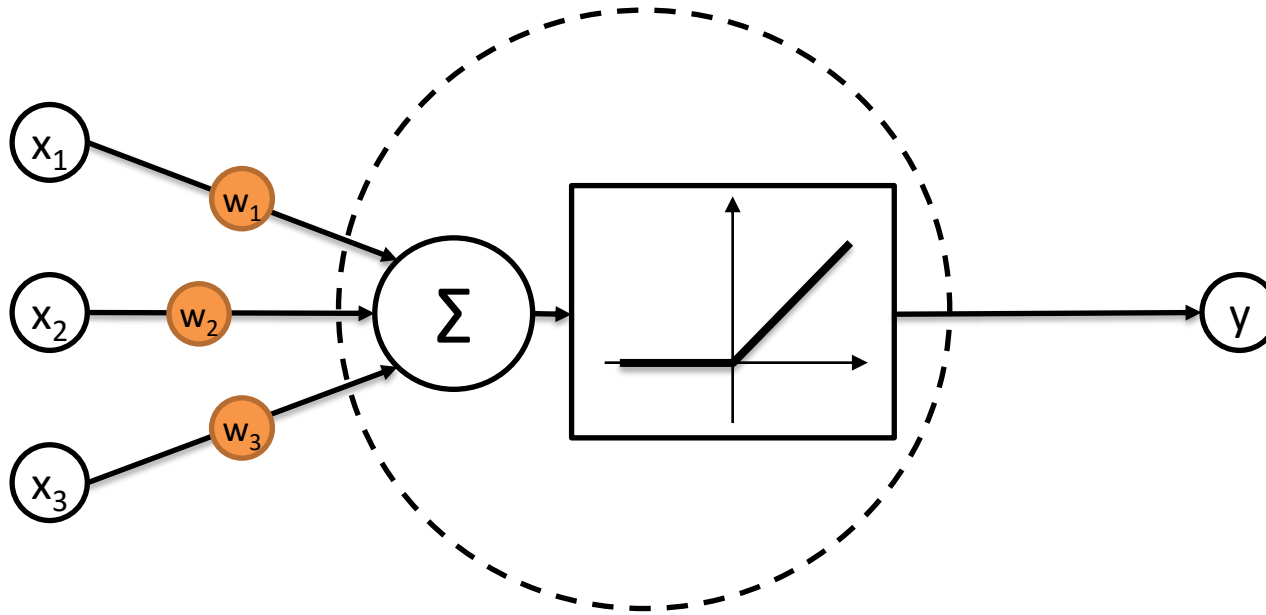
$$x_0 = 1$$

$$z = \sum_i w_i x_i$$

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

Rectified Linear Neurons

Output a non-linear function of the total input

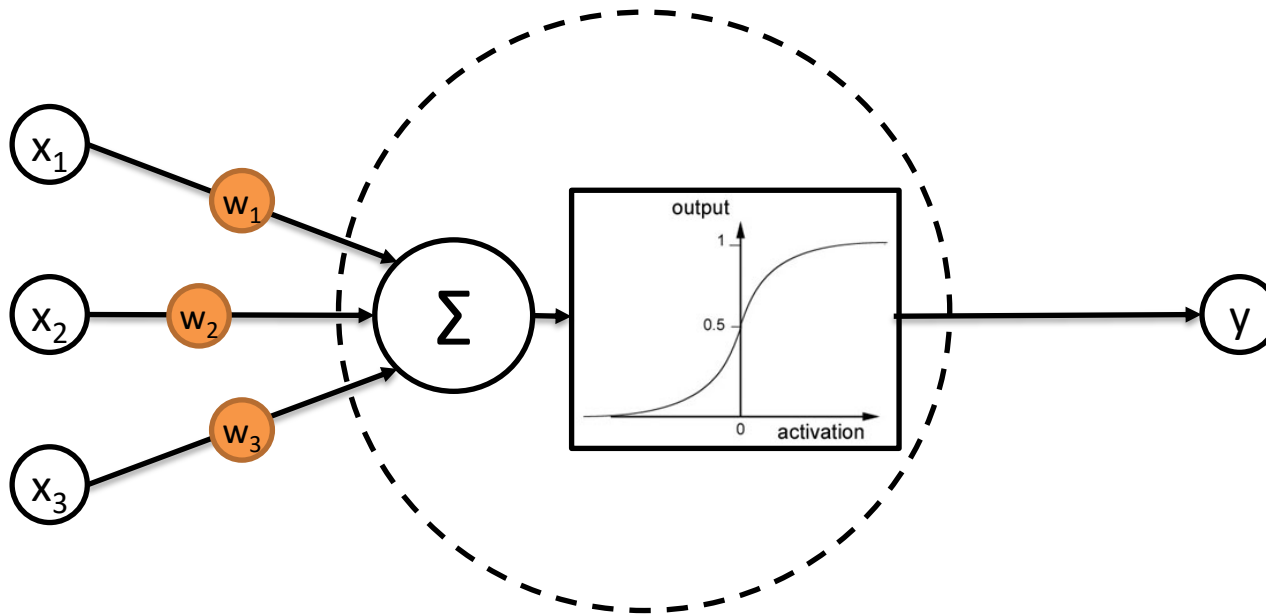


$$z = b + \sum_i w_i x_i$$

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

Logistic (Sigmoid) Neurons

Real-valued output – smooth and bounded between [0, 1]

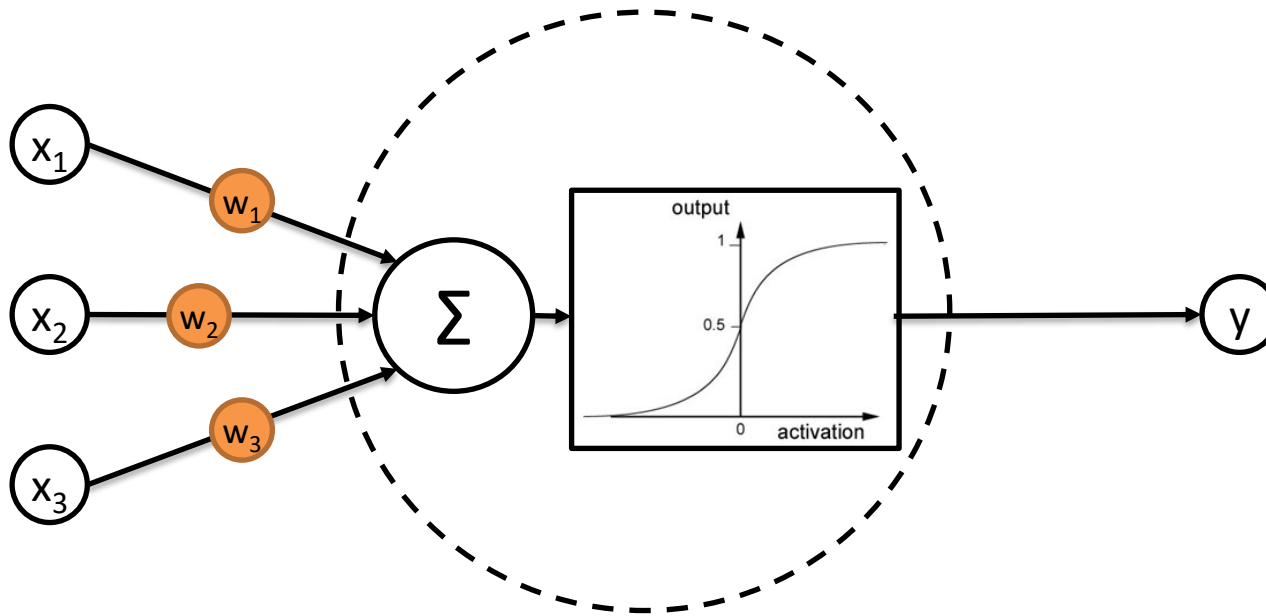


$$z = b + \sum_i w_i x_i$$

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

Stochastic Binary Neurons

Same equations as logistic units, but they treat the output as a probability of producing a spike



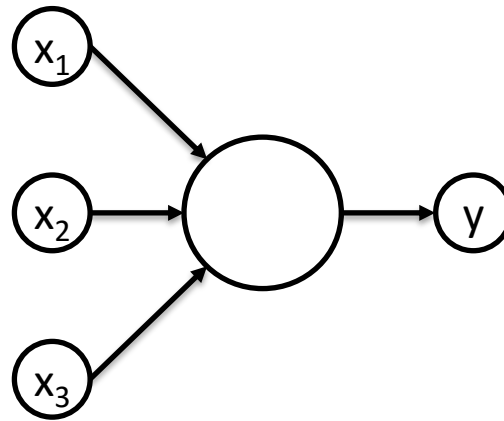
$$z = b + \sum_i w_i x_i$$

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

NEURAL NETWORK ARCHITECTURES

Single Neuron (Perceptron)

With a single neuron we can basically do linear / logistic regression (depending on how we define the activation function)



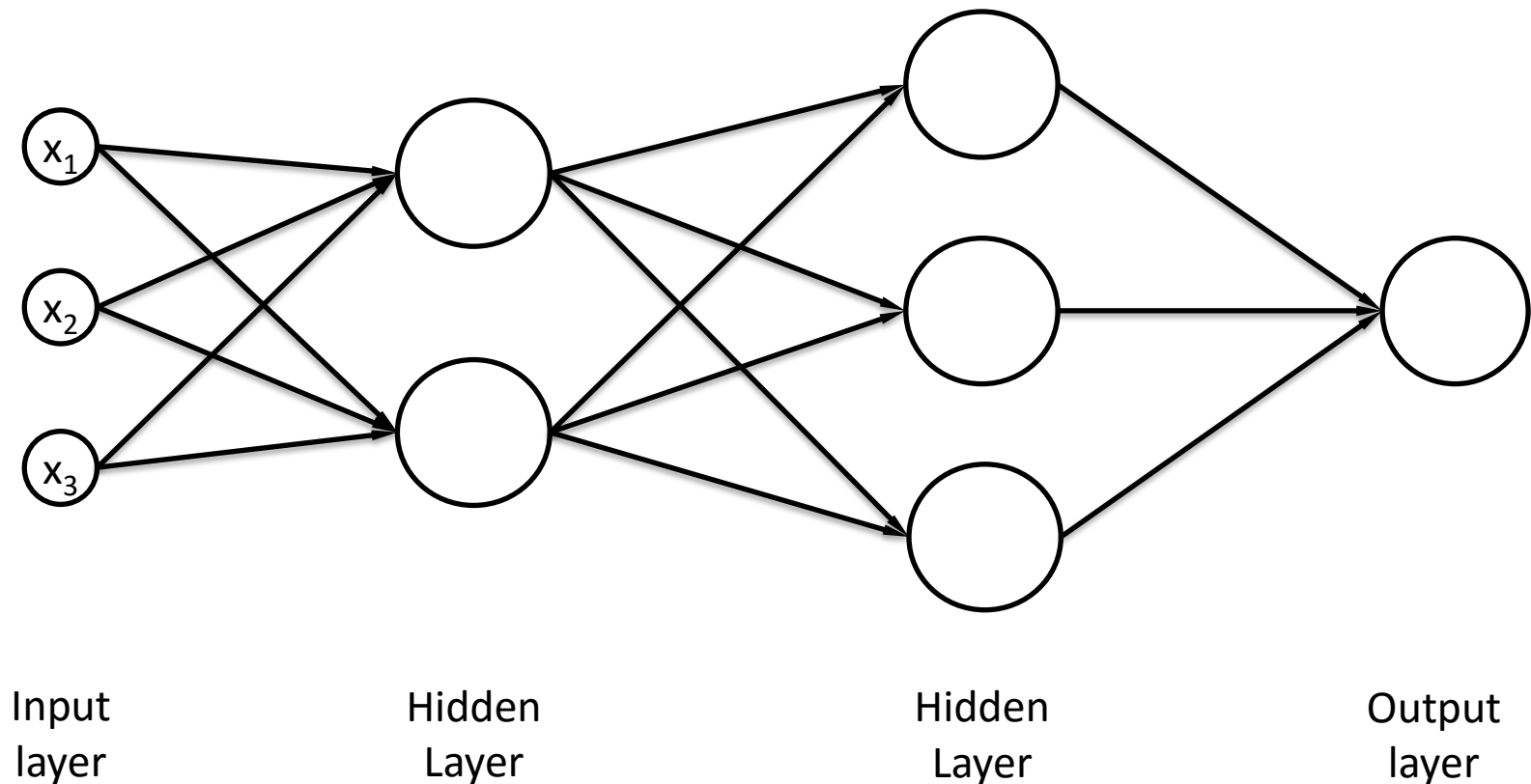
$$y = h\left(b + \sum_i w_i x_i\right)$$

Feed-Forward Neural Networks

No-cycles, all connections point to the same direction

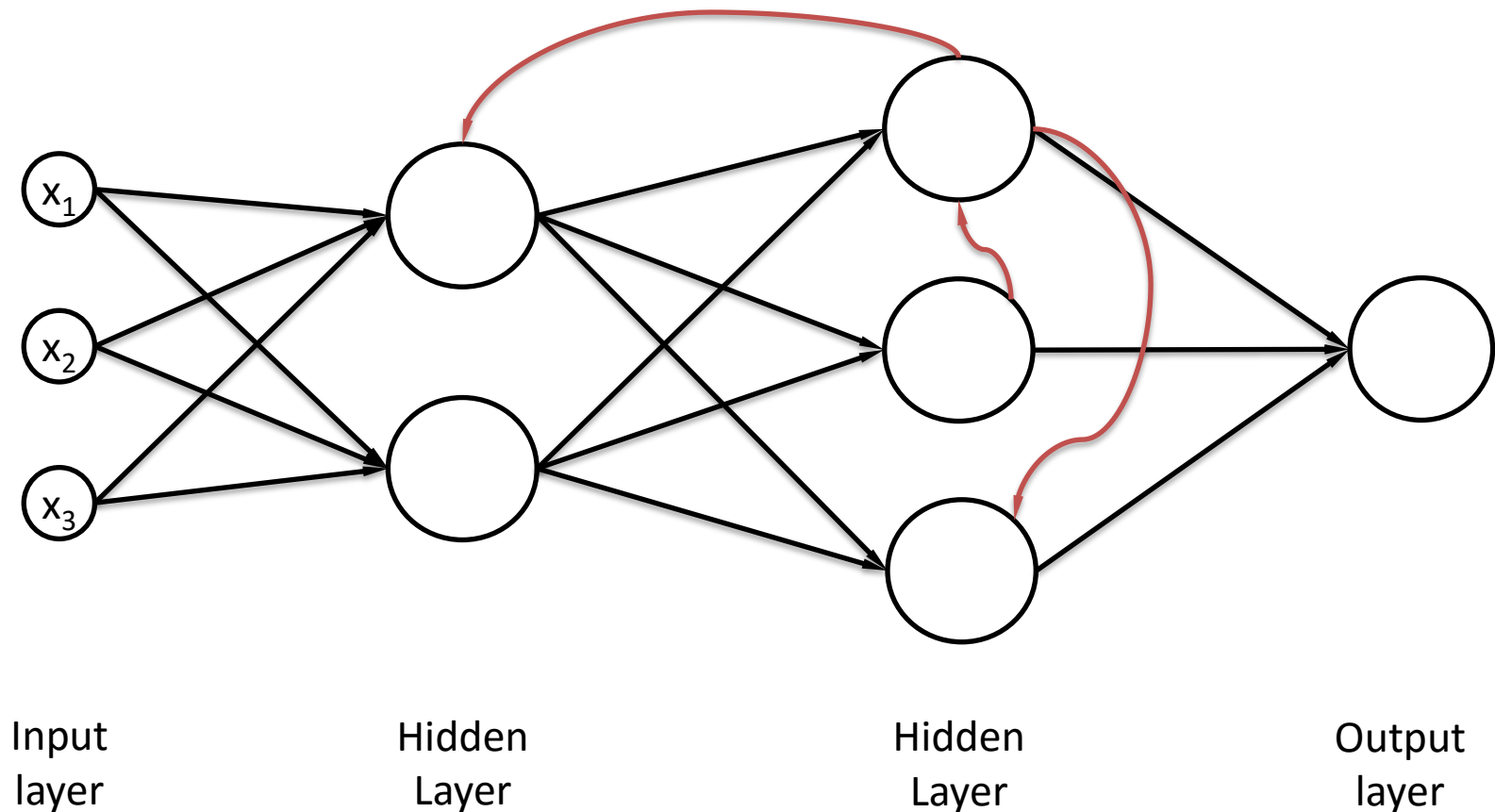
They perform a series of transformations of the input data (that change the similarities between cases) before the final output is calculated

If a feed forward NN has more than one hidden layer, it is called “deep”



Recurrent Neural Networks

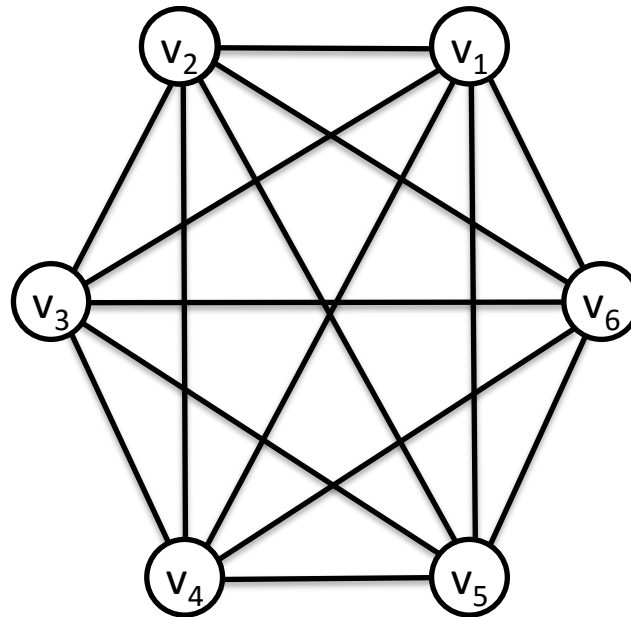
If we allow cycles, we get recurrent neural networks. RNNs can have “states”. They offer a natural way to model sequential data.



Symmetrically Connected Networks without hidden units (Hopfield Nets)

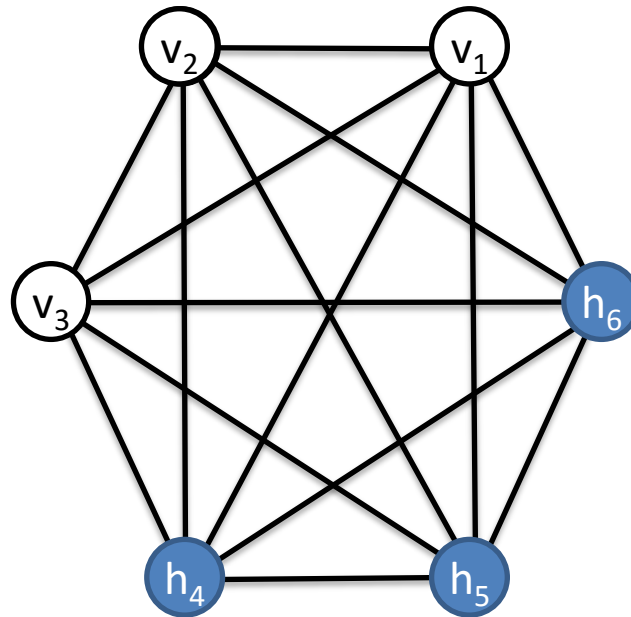
Similar to recurrent networks, but the connections are symmetrical (they have the same weight in both directions)

These are unsupervised networks



Symmetrically Connected Nets with hidden units (Boltzmann Machines)

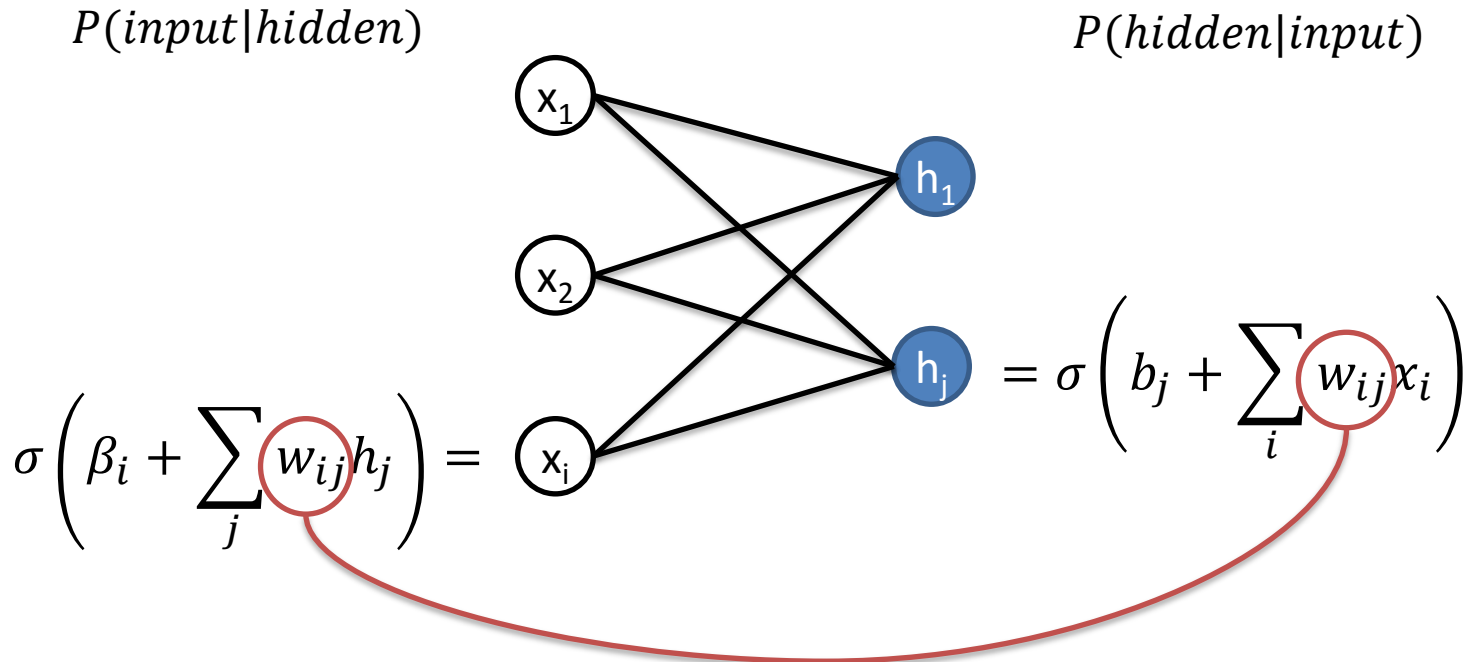
The stochastic counterpart of Hopfield nets. Much more powerful models with a simple learning algorithm.



Restricted Boltzmann Machines

Like Boltzmann machines, but their neurons must form a bipartite graph: there are no connections between nodes within a group.

Trained to maximise the likelihood of input data. It is the equivalent of calculating a mixture model.

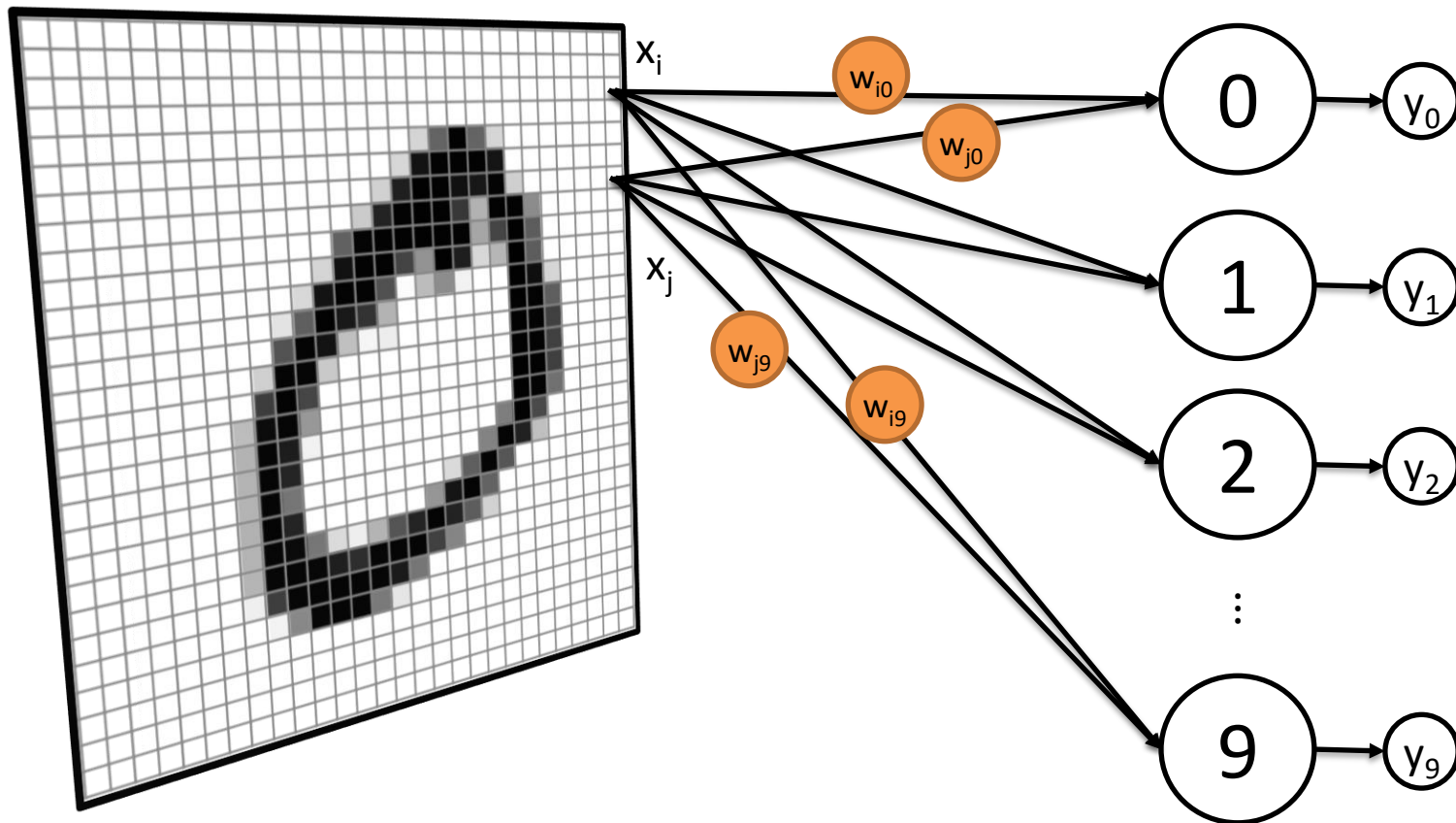


Slides reproduced from G. Hinton

LEARNING THE WEIGHTS OF A 2-LAYER NETWORK

A 2-layer architecture

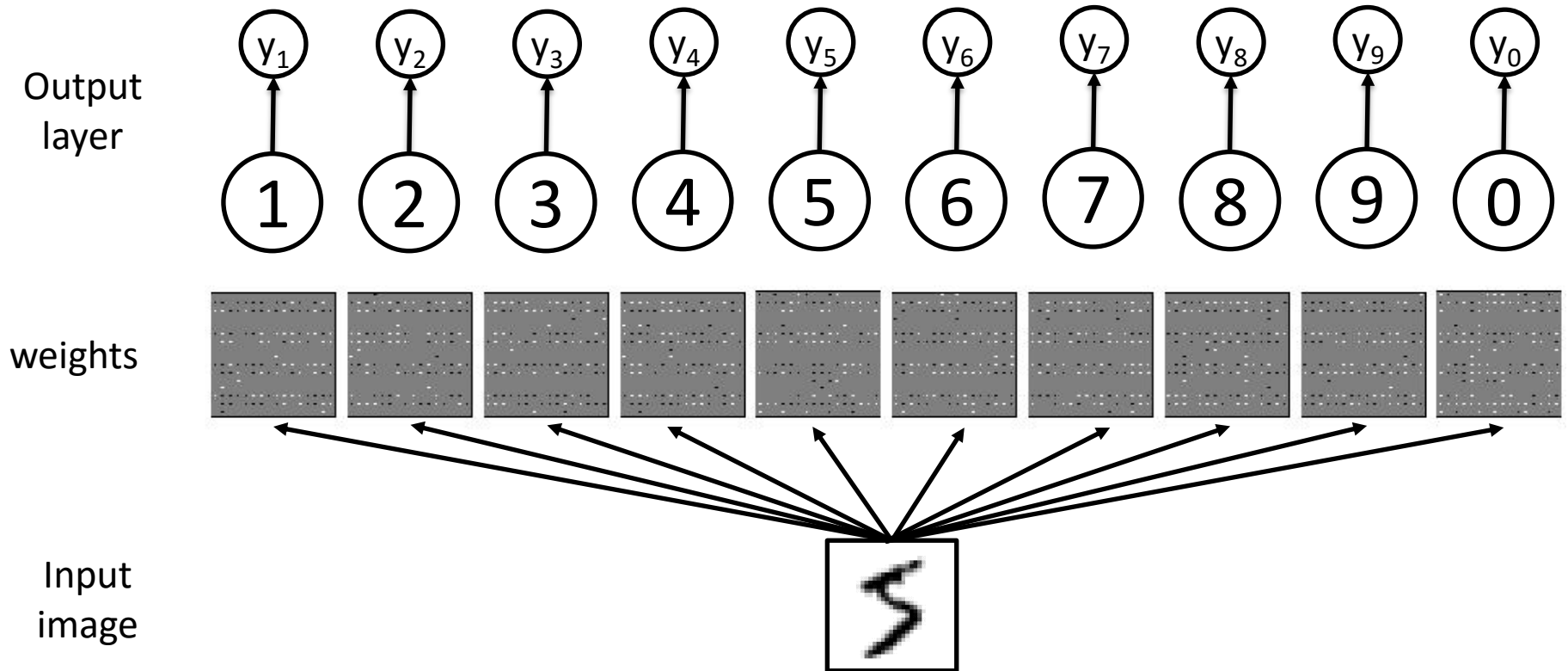
Each inked pixel can vote for several different shapes, and the shape with the most votes wins



Input: pixel values

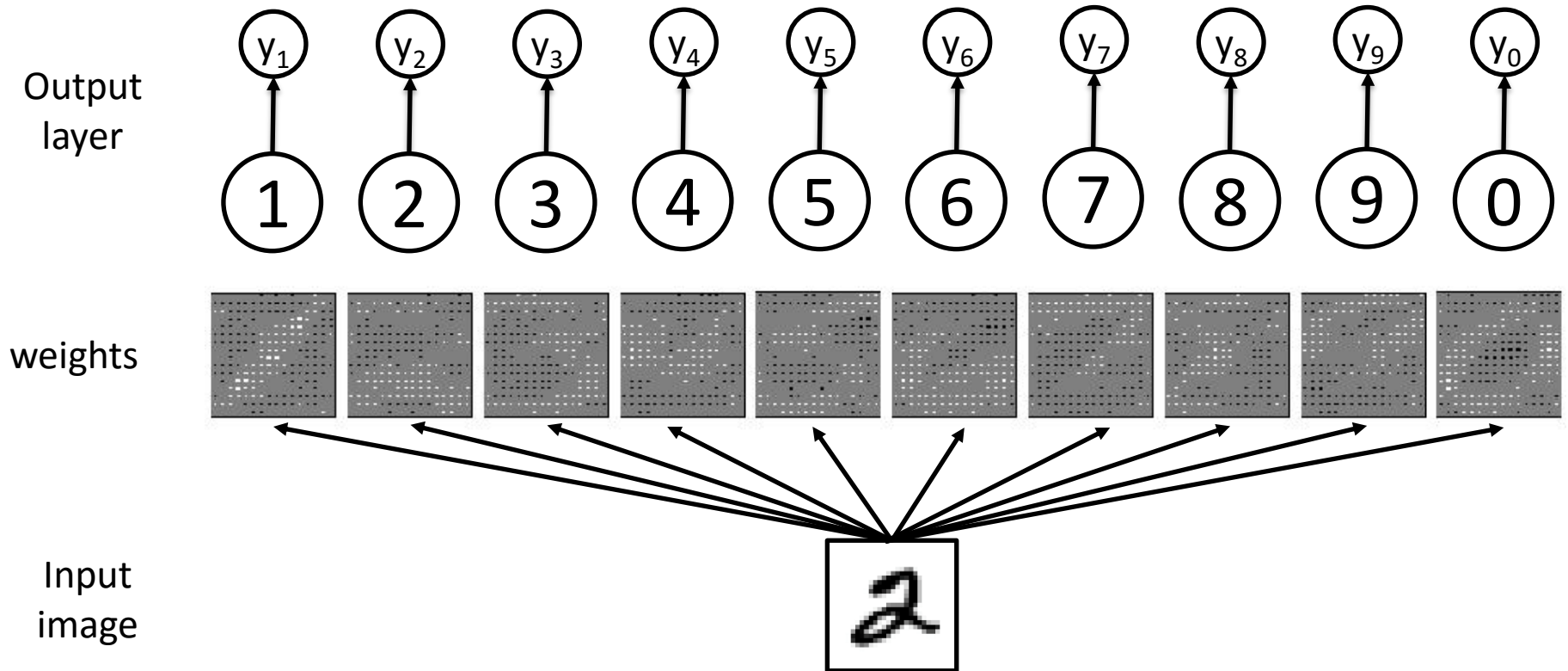
Output: 10 known digits (classes)

Learning the weights



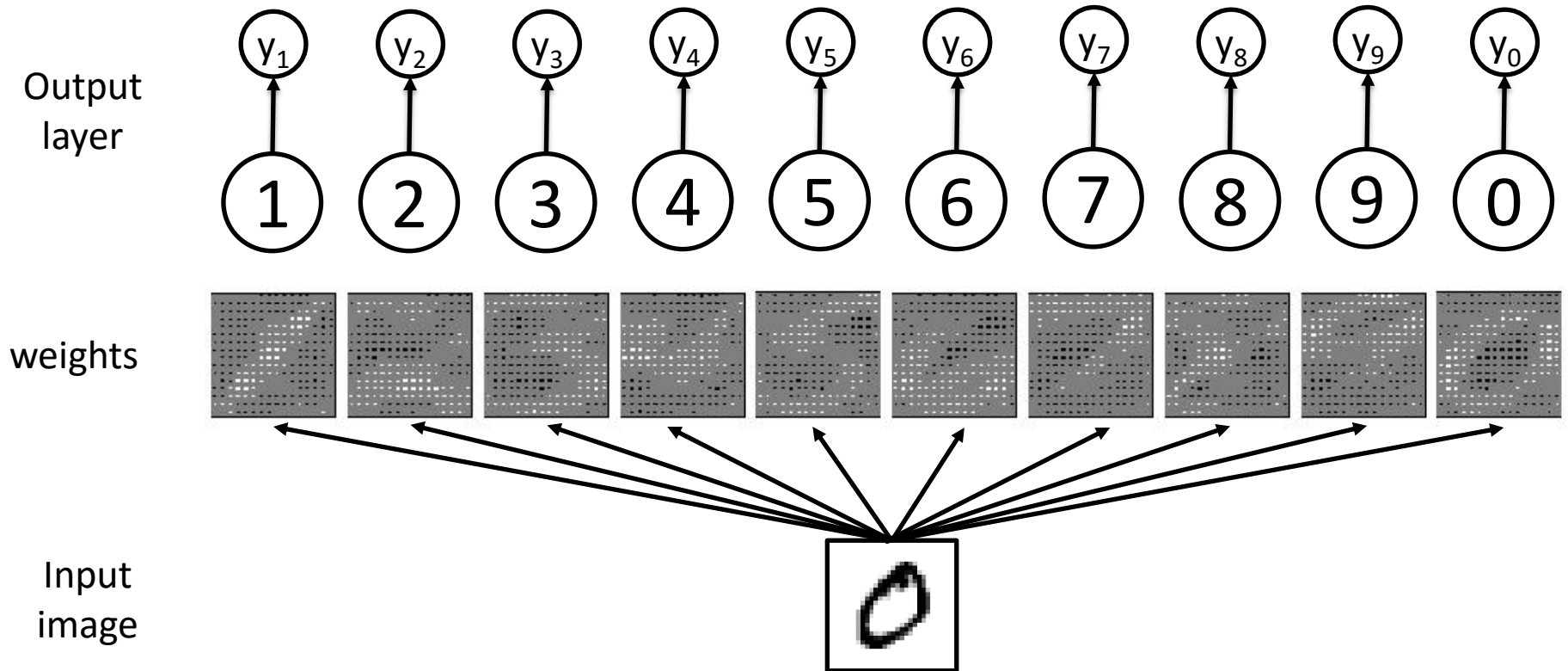
Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Learning the weights



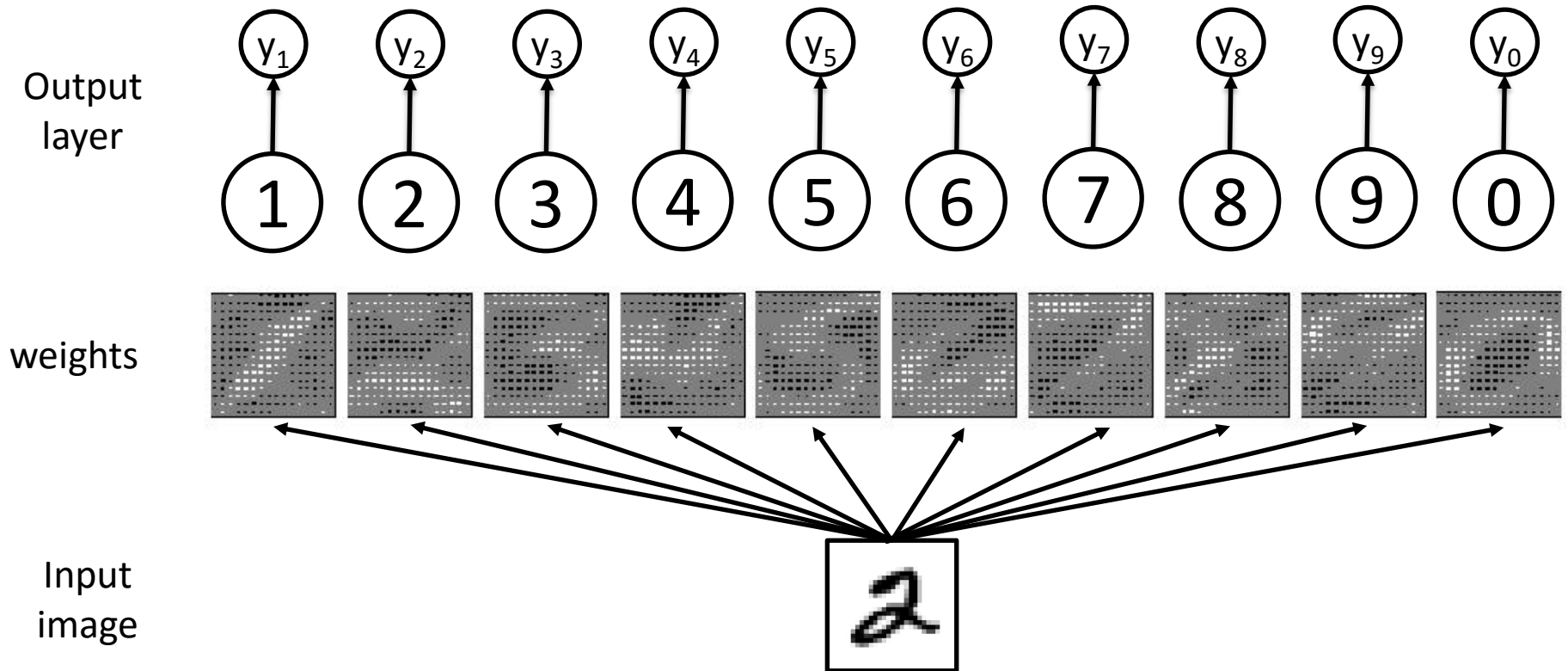
Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Learning the weights



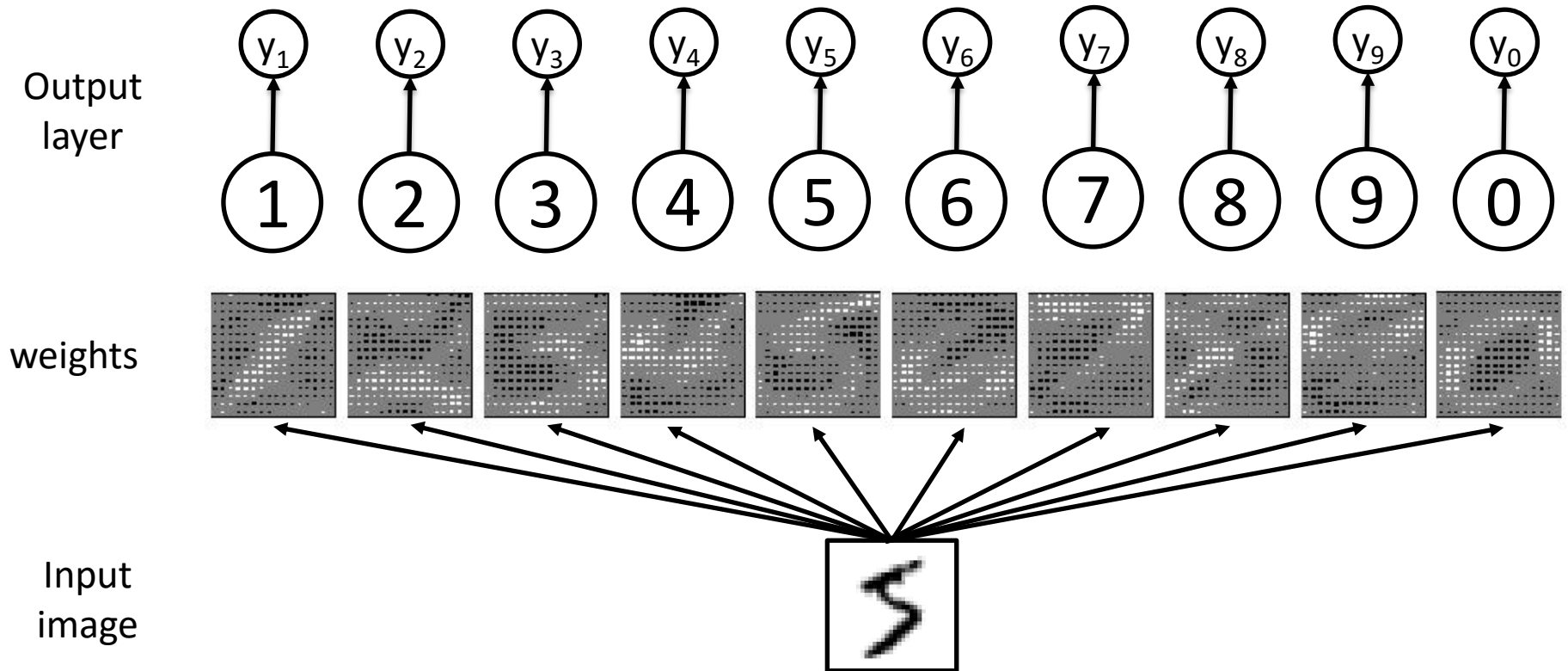
Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Learning the weights



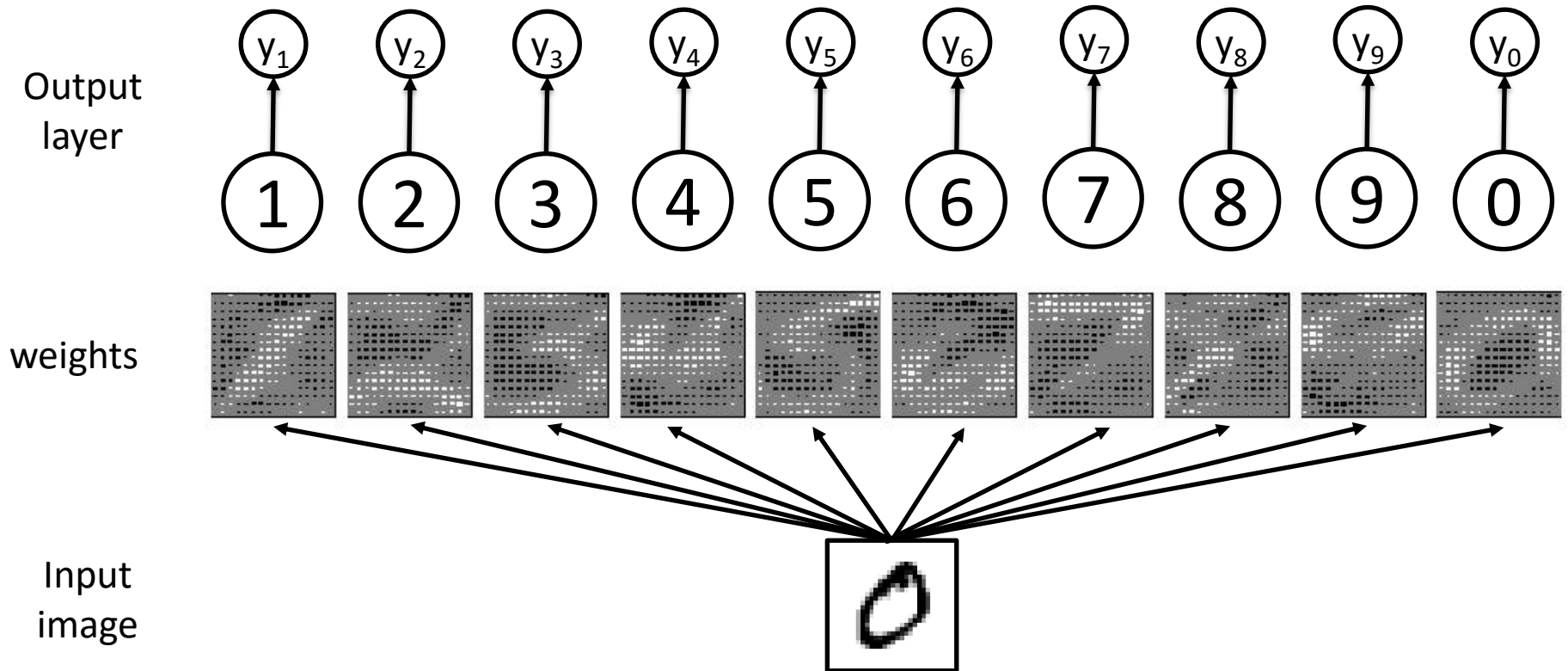
Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Learning the weights



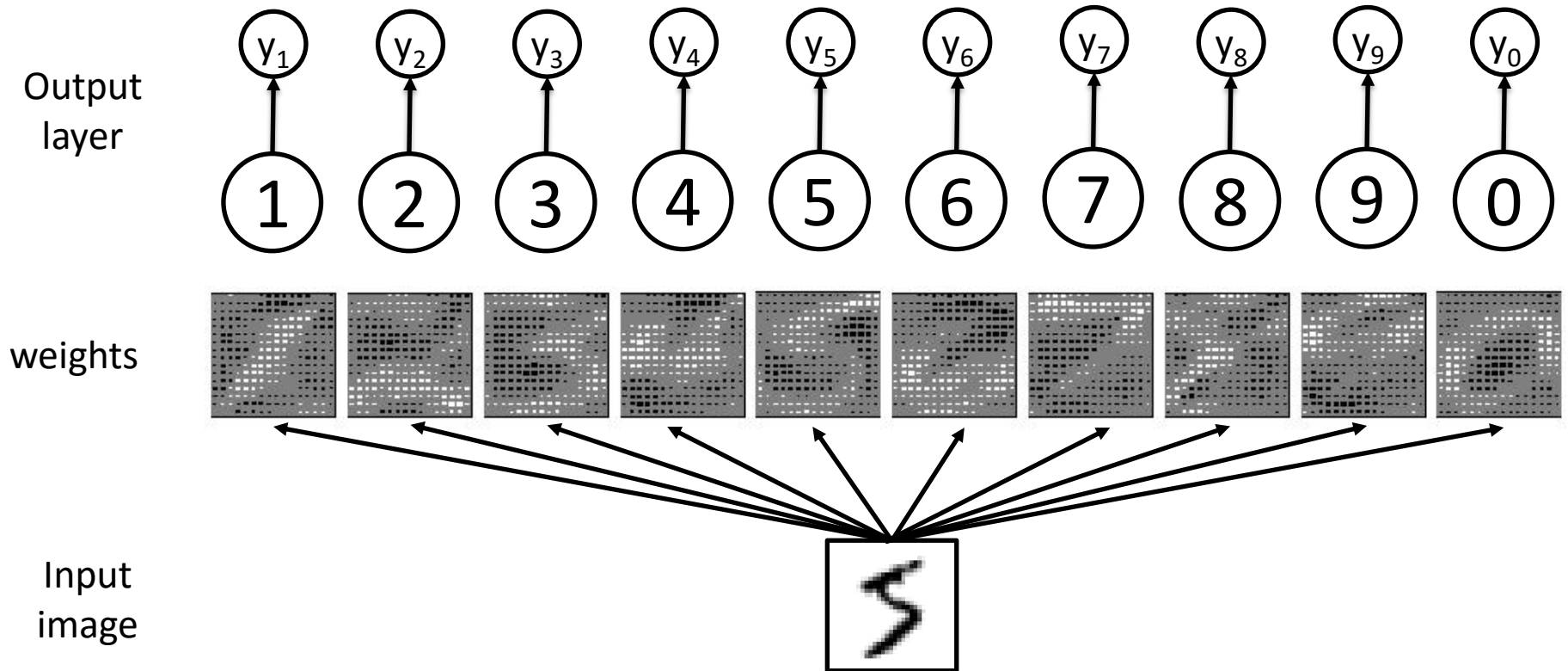
Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Learning the weights



Show the network an image. For each image increment the weights from inked pixels to the correct class, and decrement the weights from inked pixels to the wrong class

Limitations



A two-layer network with a single winner is equivalent to having a rigid template. Real-world variation cannot be captured by such simple template matching.

Limitations

To capture all the allowable variations of a digit we need to learn the **features** that it is composed of



PERCEPTRONS

Standard paradigm of classification

The standard perceptron architecture

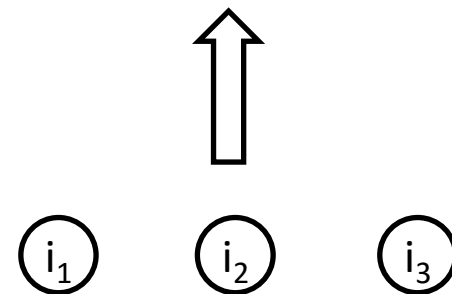
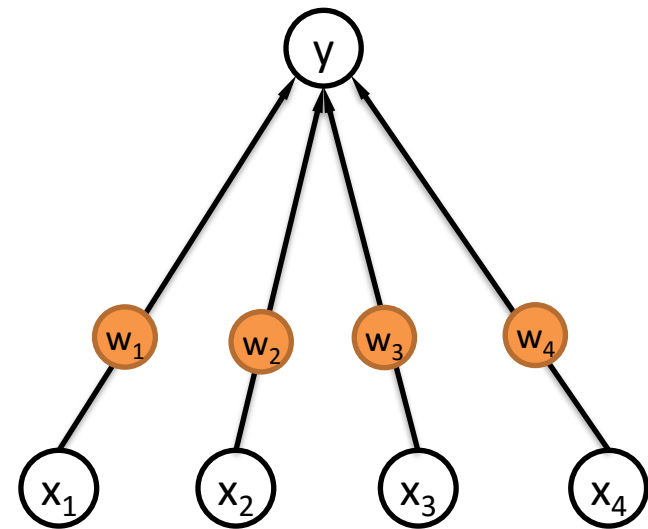
If this quantity is above some threshold,
decide that the input vector is a positive
example of the target class



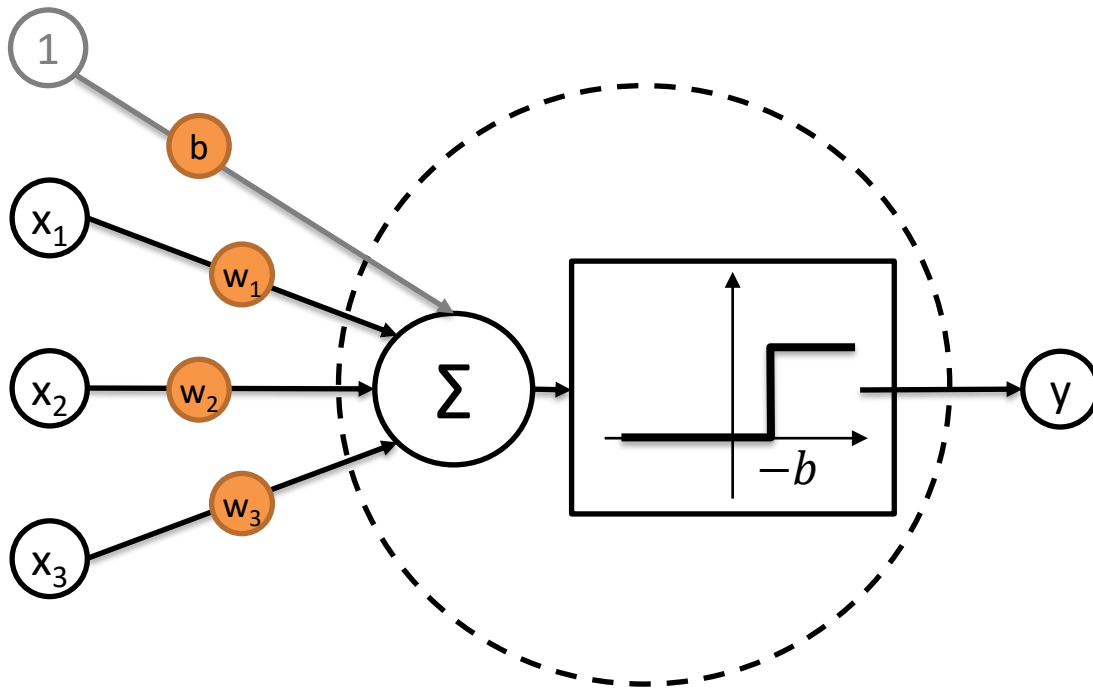
Learn how to weight each of the features
to get a single scalar quantity



Convert the raw input vector into a vector
of feature activations using hand-coded
weights or programs



Binary Threshold Neurons (McCulloch-Pitts – 1943)



$$z = b + \sum_i w_i x_i$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

We will be effectively trying to train binary output neurons as classifiers (decision units)

Perceptron Convergence Procedure

Add an extra component with value 1 to each input vector. The weight of this component is the “bias” weight, equal to minus the threshold

Pick training cases using any policy that ensures that every training sample will keep getting picked (can repeat cases, but uniformly)

For each training sample run the network

- If the output unit is correct, do not change weights
- If the output unit incorrectly outputs a zero (should be positive, but returns negative), add the input vector to the weight vector
- If the output unit incorrectly outputs a 1 (should be negative, but returns positive), subtract the input vector from the weight vector

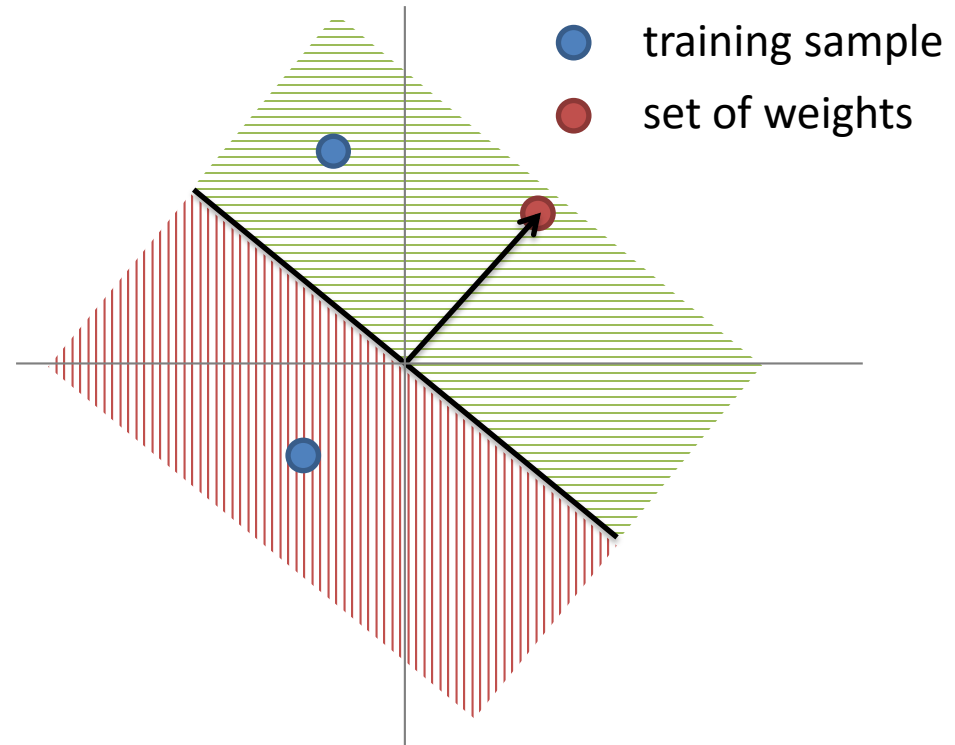
This is guaranteed to find a set of weights that gets the right answer for all the training cases **if any such set exists**

A Geometrical View

We will be switching between the data and the parameter space

In the normal (feature) space:

- Each training sample is a point
- Each set of weights defines a plane
- Training samples that this set of weights classifies as positive lie on one side, while training samples that this set of weights classify as negative lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

$$z = \sum_i w_i x_i$$

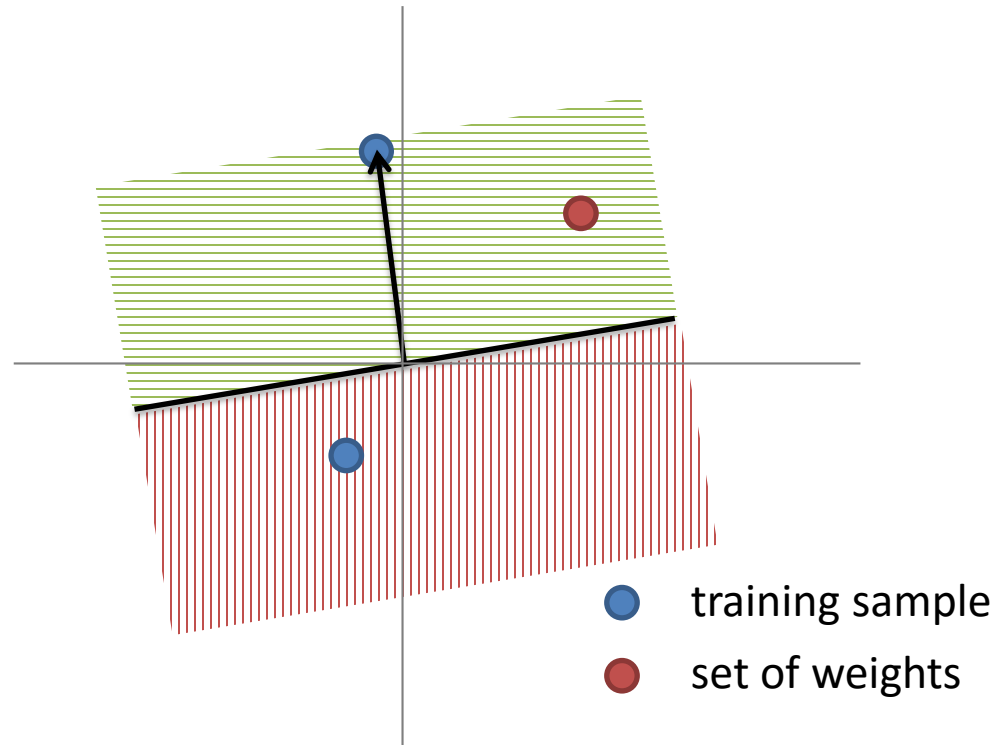
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

A Geometrical View

We will be switching between the data and the parameter space

In the parameter (weight) space:

- Each set of weights is a point
- Each training sample defines a plane
- Weights that classify it correctly lie on one side, while weights that do not lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

$$z = \sum_i w_i x_i$$

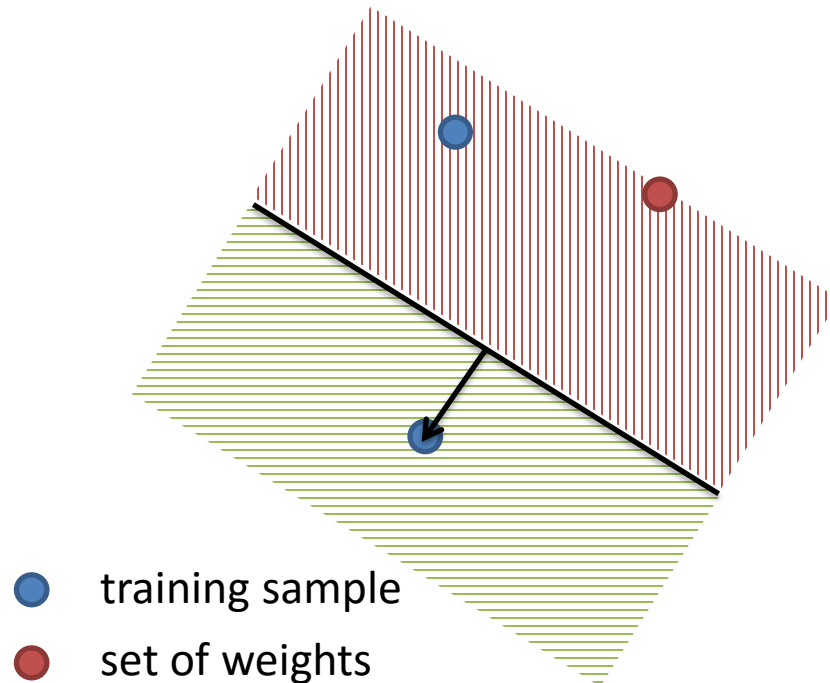
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

A Geometrical View

We will be switching between the data and the parameter space

In the parameter (weight) space:

- Each set of weights is a point
- Each training sample defines a plane
- Weights that classify it correctly lie on one side, while weights that do not lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

$$z = \sum_i w_i x_i$$

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

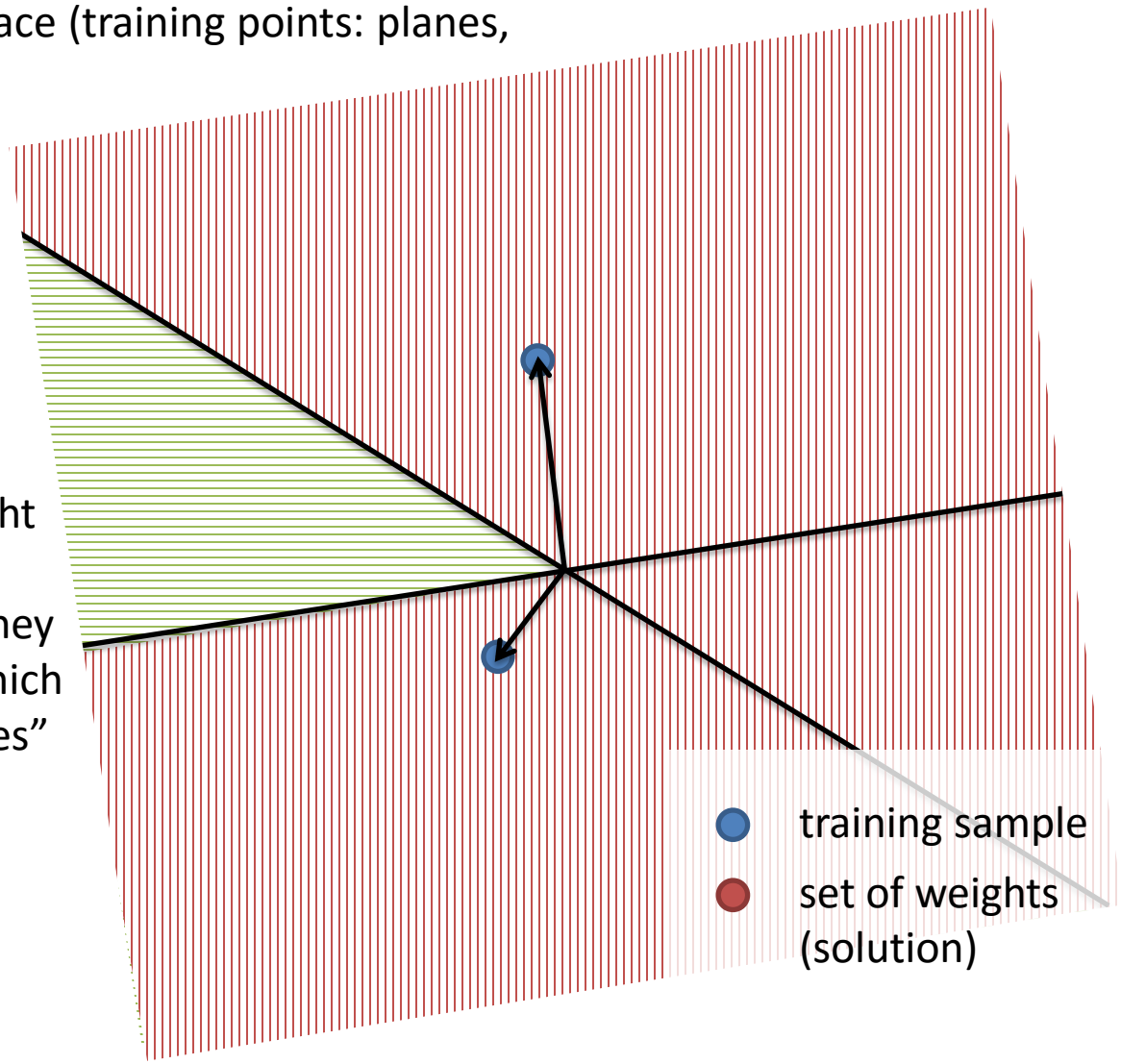
The cone of feasible solutions

We are in the parameter space (training points: planes, weights: points)

To classify all training samples correctly we need to find a set of weights on the correct side of all the planes

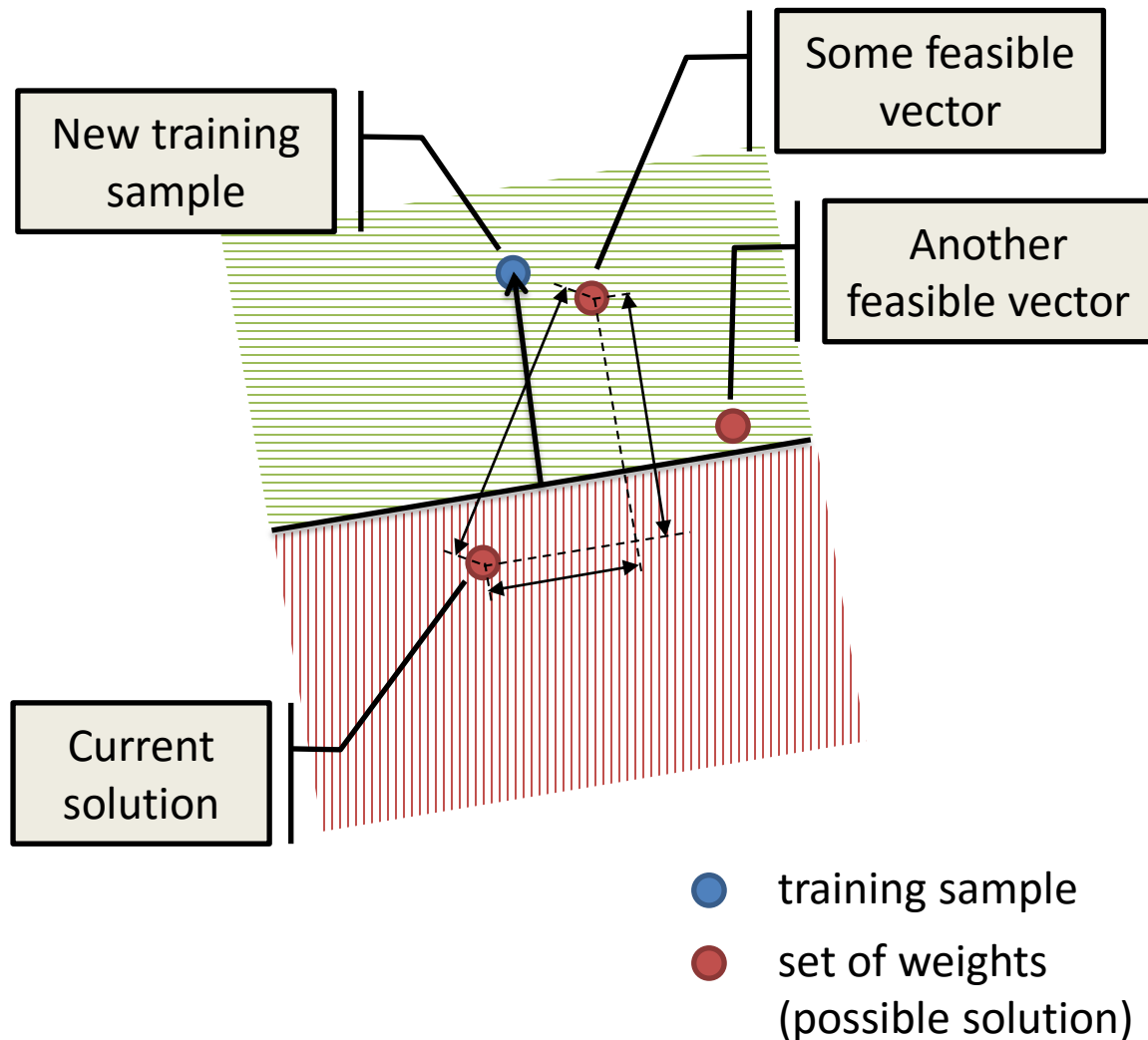
If there are any sets of weight that can give us the correct responses for all samples, they will lie in the hyper-cone which is the union of all “good sides”

The average of two good solutions is a good solution (the problem is convex)



Why does the learning procedure work?

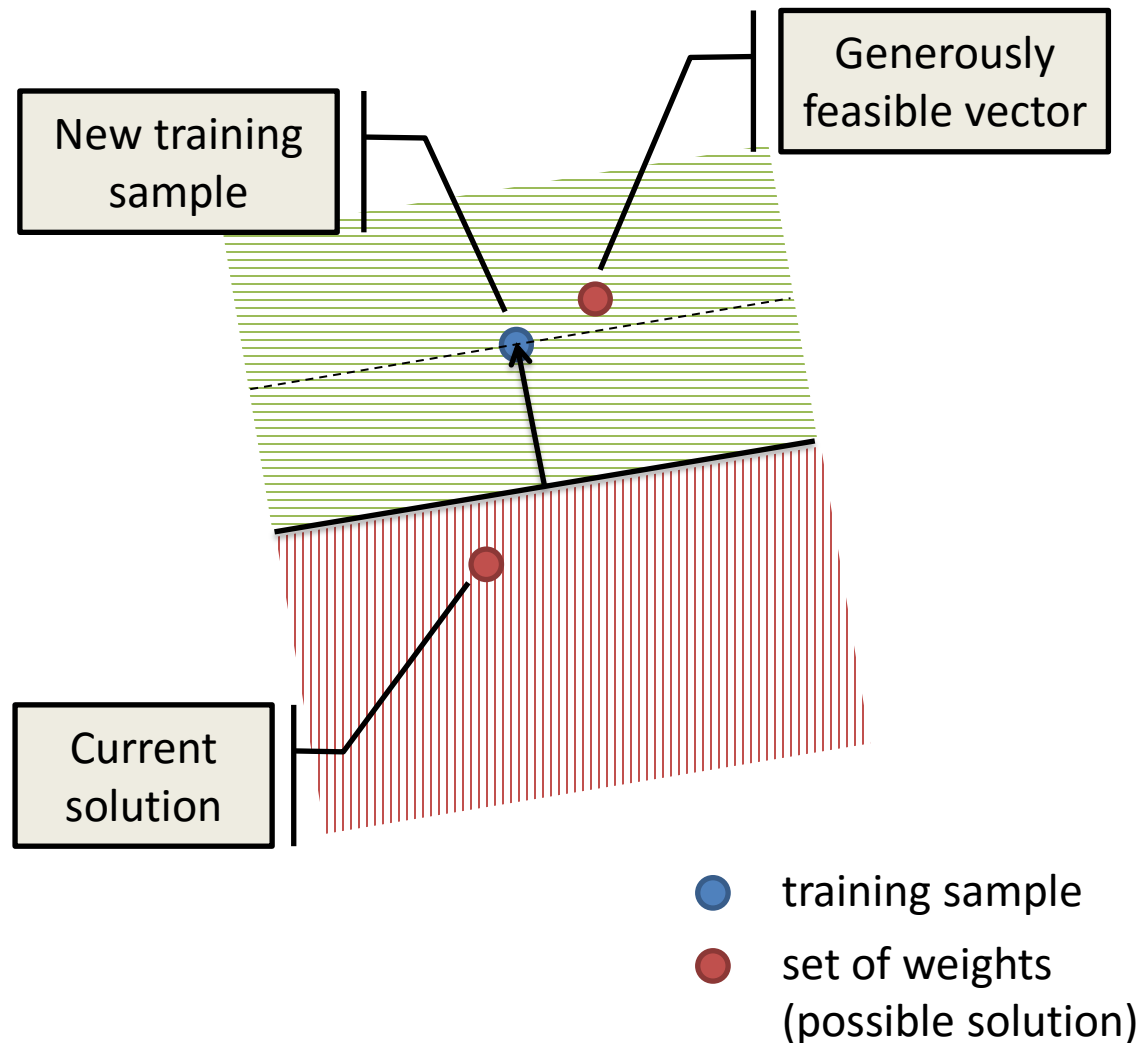
Claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors



Why does the learning procedure work?

Consider “generously feasible” weight vectors that lie within the feasible region by a margin at least as great as the length of the input vector that defines each constraint plane

Real claim: Every time the perceptron makes a mistake the squared distance to all of these generously feasible vectors is always reduced by at least the squared length of the update vector



Limitations of perceptrons

- If you use the right features you can do almost everything, if not, there are serious limitations
 - Suppose for example that you have binary inputs and you can have a separate feature unit for each binary code possible – any possible discrimination is doable
 - Too many features, does not generalise, the definition of overfitting
- It emphasises that the difficult bit of learning is to learn the right features
- This is the reason why perceptrons (and neural networks) fell out of favour in the late 1960s

Solving the XOR problem

Imagine the following training set:

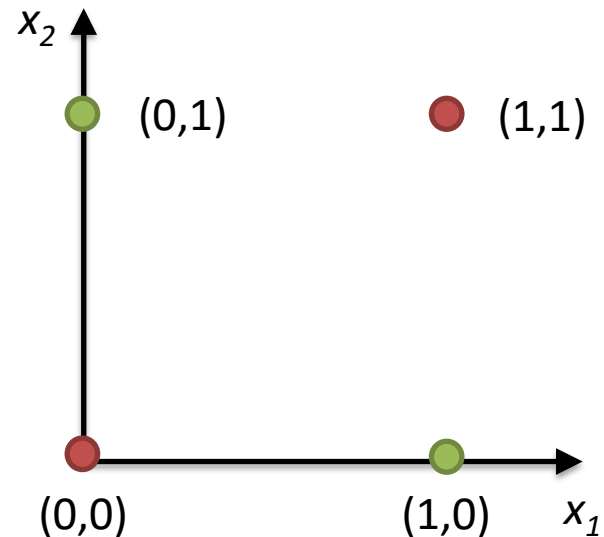
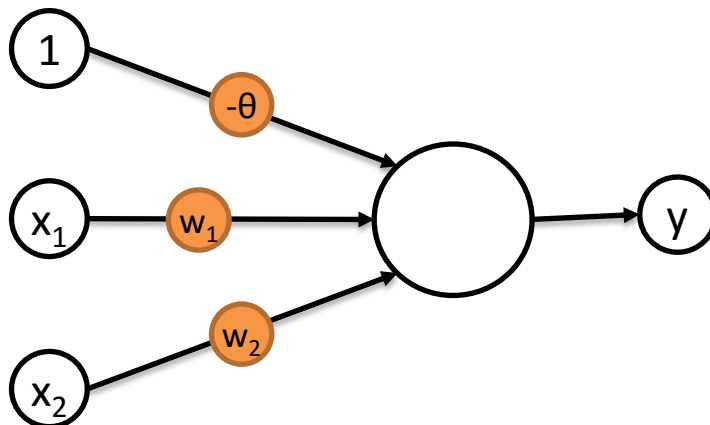
- Positive cases if x_1 is the same as x_2
- Negative cases if x_1 is different to x_2

$$w_2 < \theta$$

$$w_1 + w_2 \geq \theta$$

$$0 \geq \theta$$

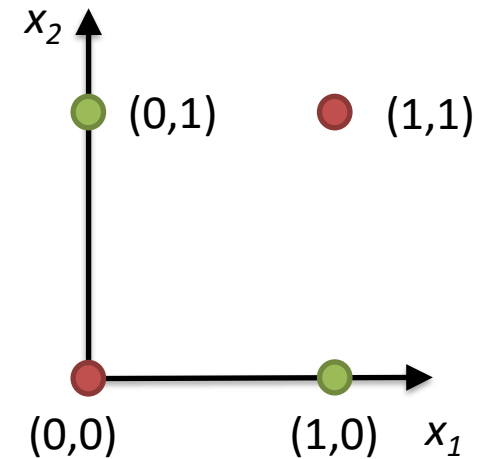
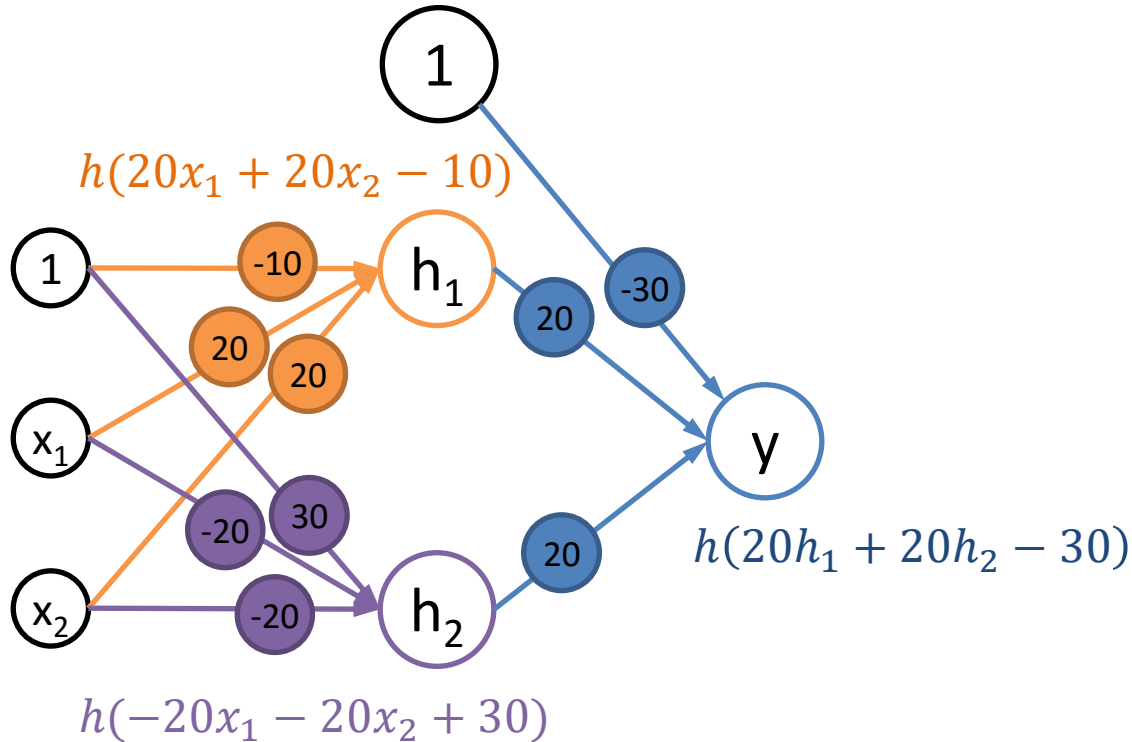
$$w_1 < \theta$$



$$y = \begin{cases} 1 & \sum_i w_i x_i \geq \theta \\ 0 & \sum_i w_i x_i < \theta \end{cases}$$

●
●

Solving the XOR problem - revisited



OR

$$\sigma(20 * 0 + 20 * 0 - 10) = 0$$

$$\sigma(20 * 1 + 20 * 1 - 10) = 1$$

$$\sigma(20 * 0 + 20 * 1 - 10) = 1$$

$$\sigma(20 * 1 + 20 * 0 - 10) = 1$$

NAND

$$\sigma(-20 * 0 - 20 * 0 + 30) = 1$$

$$\sigma(-20 * 1 - 20 * 1 + 30) = 0$$

$$\sigma(-20 * 0 - 20 * 1 + 30) = 1$$

$$\sigma(-20 * 1 - 20 * 0 + 30) = 1$$

AND

$$\sigma(20 * 0 + 20 * 1 - 30) = 0$$

$$\sigma(20 * 1 + 20 * 0 - 30) = 0$$

$$\sigma(20 * 1 + 20 * 1 - 30) = 1$$

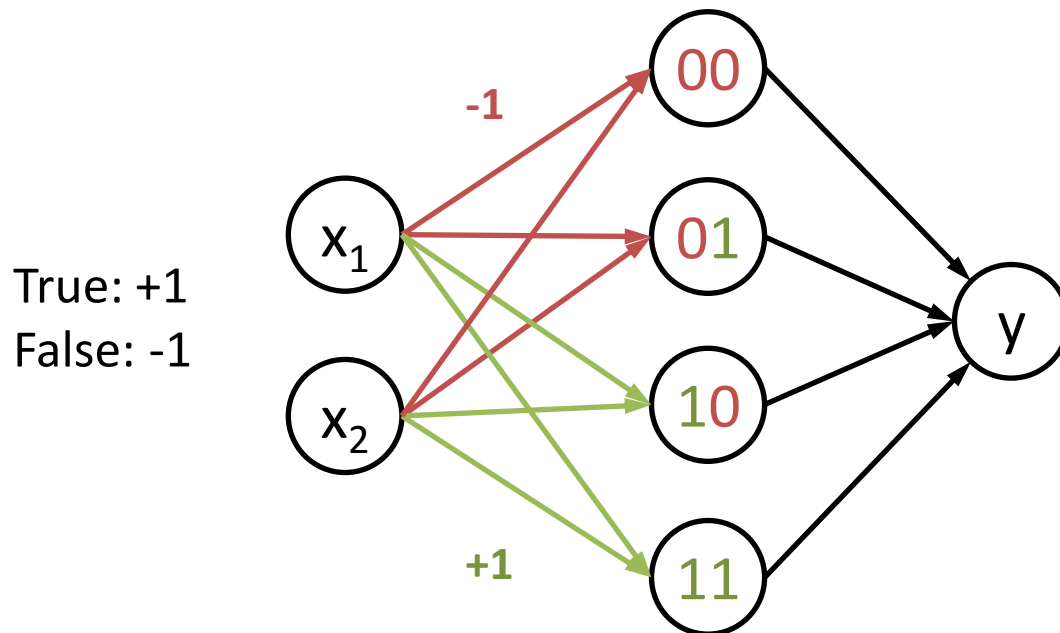
$$\sigma(20 * 0 + 20 * 0 - 30) = 0$$

Representation power of Neural Nets

When adding hidden layers, we are learning features

A Neural Network with a hidden layer can represent:

- Any bounded continuous function (to arbitrary ϵ)
 - Universal Approximation Theorem [Cybenko 1989]
- Any Boolean function (exactly)



LEARNING THE WEIGHTS OF A LINEAR NEURON

Learning the weights of a linear neuron

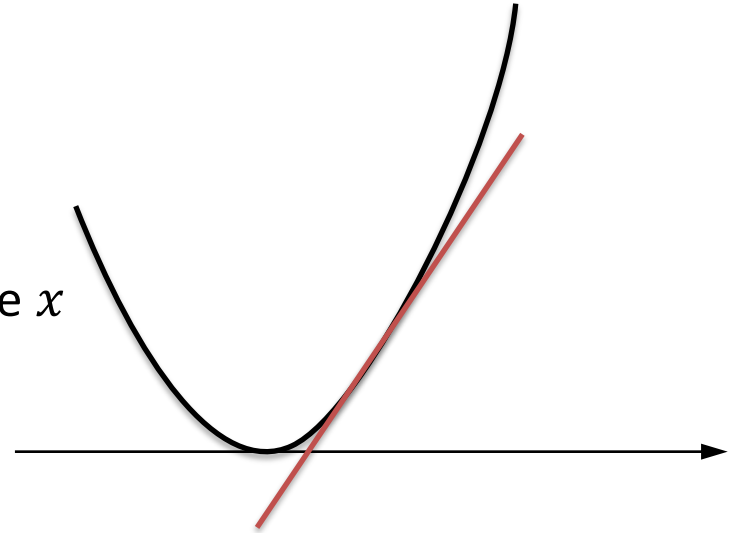
- Perceptron (binary neuron): **moving weights** closer to a good set of “generously feasible” weights
 - This type of guarantee cannot be extended to more complex networks in which the average of two good solutions may be a bad solution (non-convex)
- Linear neuron: We try instead to **minimise the distance** between predictions and true values
 - This can be true even for non-convex problems

Derivatives

The derivative of a function $f(x)$

can be interpreted as:

- How fast f changes around x
- Will f increase or decrease if we increase x
- Is x higher or lower than it should be (in respect to the optimum)



$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial}{\partial g} f(g(x)) \frac{\partial}{\partial x} g(x)$$

Chain Rule: if f grows twice as fast as g and g grows twice as fast as x , then f grows 2×2 times as fast as x

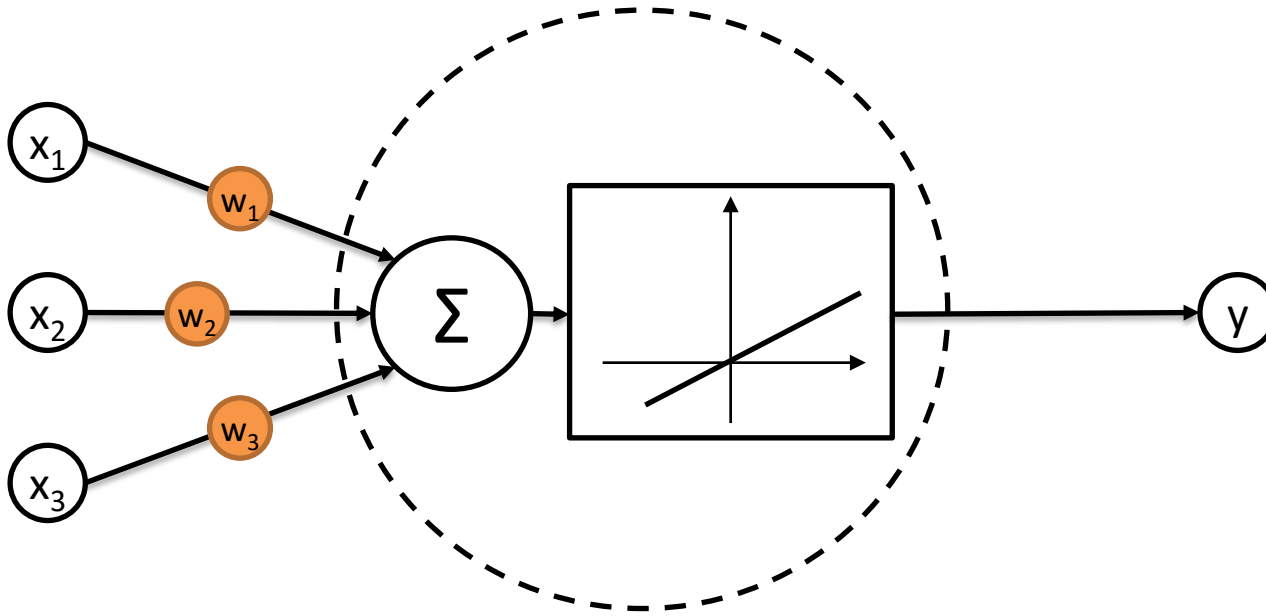
$$\frac{\partial}{\partial x} \sum_i f_i(x) = \sum_i \frac{\partial}{\partial x} f_i(x)$$

The derivative of the sum equals the sum of derivatives

$$\frac{\partial}{\partial x_k} \sum_i a_i f(x_i) = a_k \frac{\partial}{\partial x_k} f(x_k)$$

Derivative with respect to one element of the sum

Linear Neurons

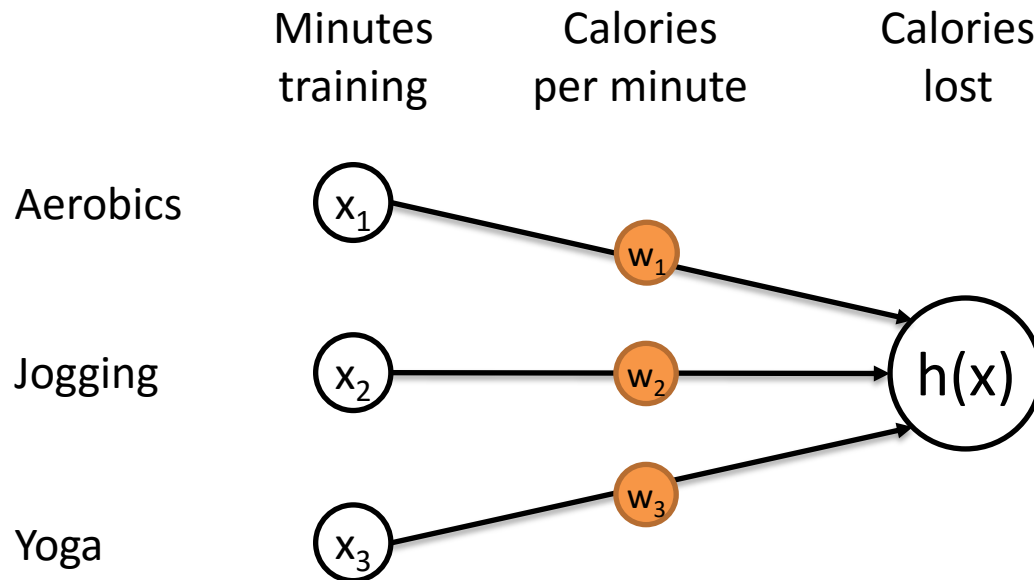


$$y = b + \sum_i w_i x_i$$

Diagram illustrating the equation $y = b + \sum_i w_i x_i$ with labels for its components:

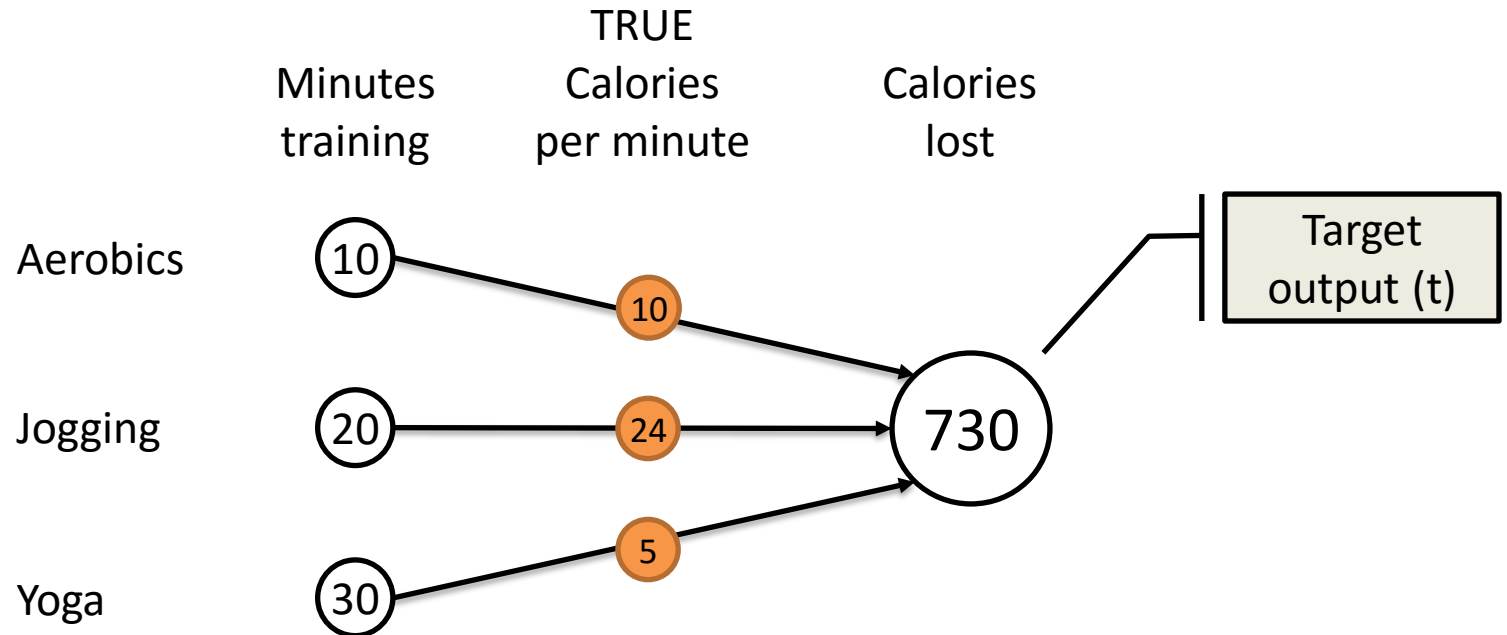
- output**: Points to y .
- bias**: Points to b .
- i^{th} input**: Points to x_i .
- weight on i^{th} input**: Points to w_i .

A day in the gym

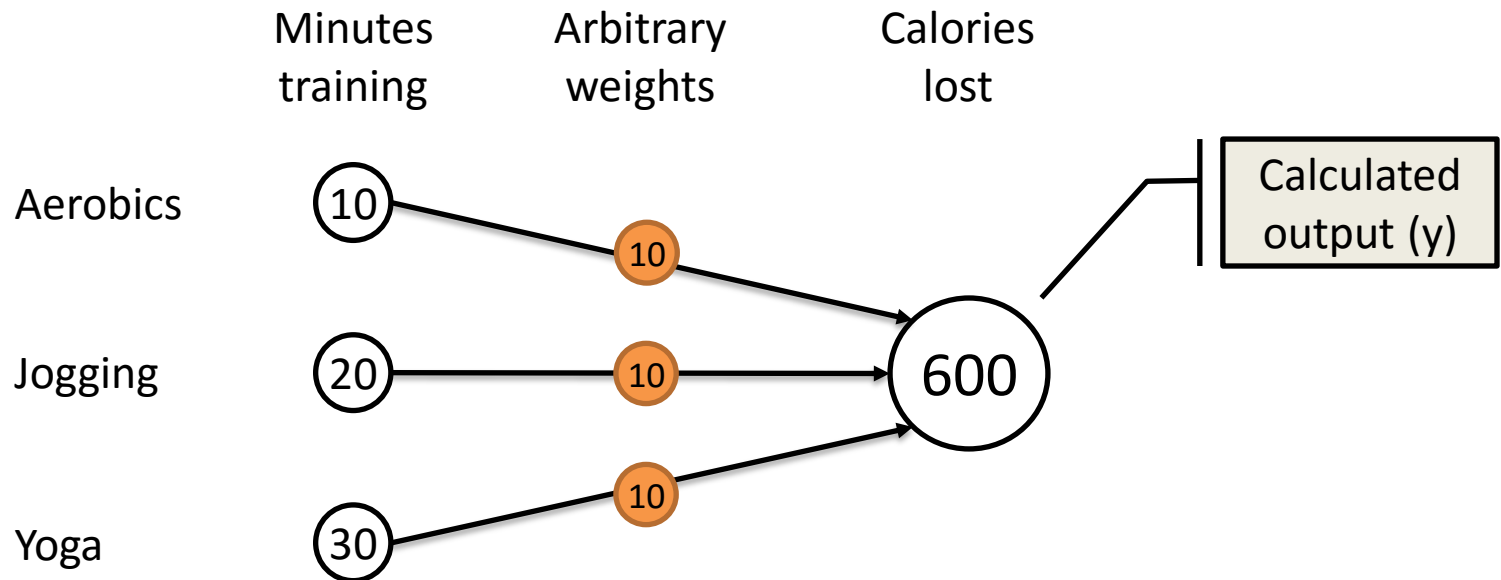


You can measure the total number of calories you have lost at the end of the session. You also know how much time you spent doing each exercise. If you repeat this many times, you should be able to figure out what is the contribution of each of the exercises to the final result.

A day in the gym



A day in the gym



Residual error $y - t = -130$

The rule for learning is $w_i \mapsto w_i - \alpha x_i (y - t)$

With a learning rate $\alpha = \frac{1}{260}$, the new weights would be: 15, 20, 25

The update rule

The derivatives of the logit with respects to inputs and weights

$$y = b + \sum_i w_i x_i \quad \frac{\partial y}{\partial w_i} = x_i \quad \frac{\partial y}{\partial x_i} = w_i$$

Error is defined as the squared residuals summed over all training samples

$$E = \frac{1}{2} \sum_{j=1}^m (y^{(j)} - t^{(j)})^2$$

Differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1}^m \frac{dE^{(j)}}{dy^{(j)}} \frac{\partial y^{(j)}}{\partial w_i} = \sum_{j=1}^m x_i^{(j)} (y^{(j)} - t^{(j)})$$

Batch update rule changes weights in proportion to their error derivatives summed over all training samples

$$w_i \leftarrow w_i - \alpha \sum_{j=1}^m x_i^{(j)} (y^{(j)} - t^{(j)})$$

This is the well known gradient descent rule

The on-line version, considering one sample at the time, is called the stochastic gradient descent algorithm

Similarity to the perceptron learning rule

Perceptron learning rule

We increment or decrement the weight vector by the **input vector**

We only change by the input vector only **when we make an error**

Stochastic gradient descent

We increment or decrement the weight vector by the input vector **scaled** by the residual error and the learning rate

We do so for **all input samples**

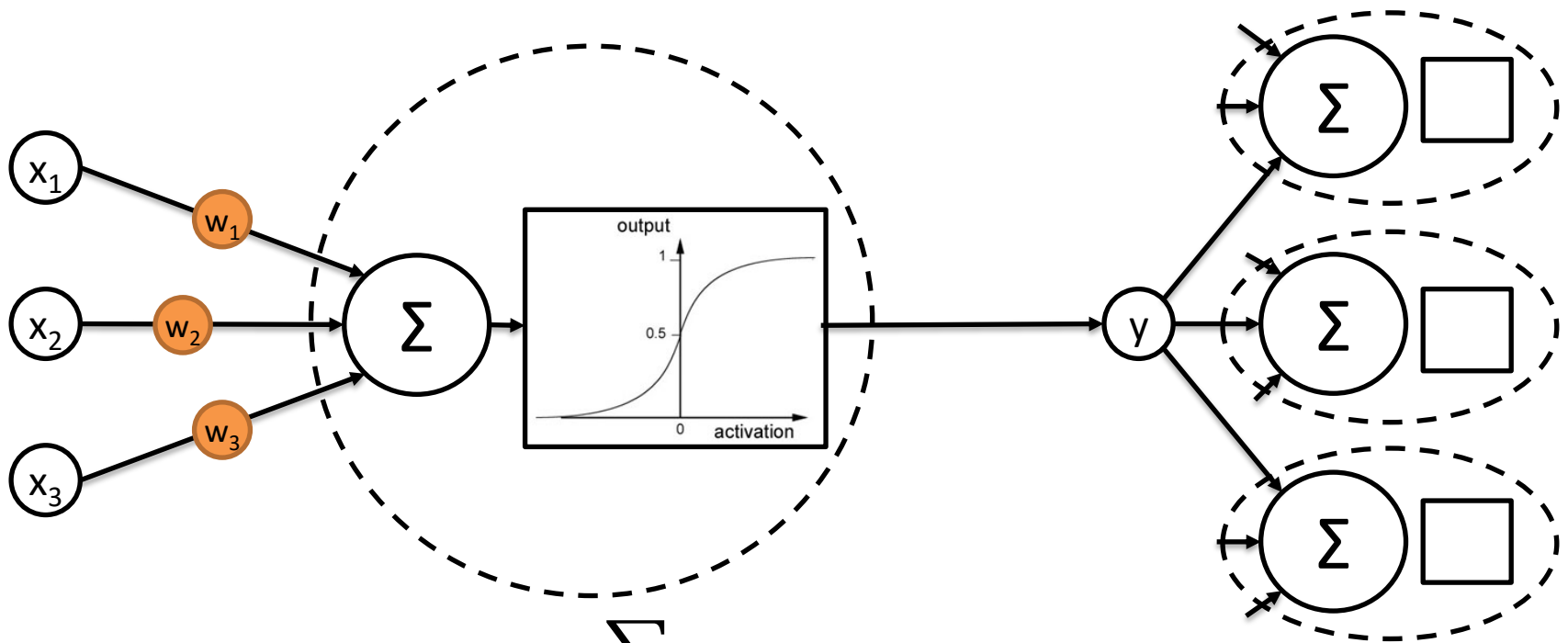
Inconvenience: we have to choose a learning rate

Do you spot any similarities to the SVM learning rule?

LEARNING THE WEIGHTS OF A LOGISTIC NEURON

Logistic Neurons

Real-valued output – smooth and bounded between [0, 1]



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

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

Derivatives of a sigmoid,

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

$$\sigma'(x) = \frac{\partial}{\partial x} (1 + e^{-x})^{-1} = -(1 + e^{-x})^{-2}(-e^{-x}) = \frac{1}{1 + e^{-x}} \frac{e^{-x}}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

$$\sigma'(f_x) = \sigma(f_x)(1 - \sigma(f_x)) \frac{\partial}{\partial x} f_x$$

The update rule

The activation function is the logistic function

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

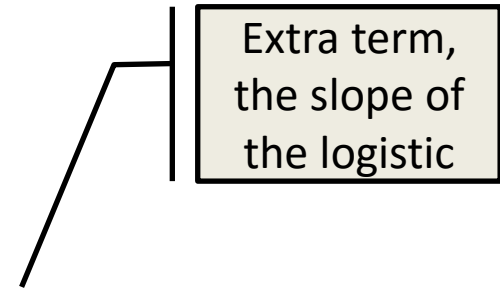
The derivatives of the logistic with respect to the weights is

$$\frac{\partial y}{\partial w_i} = \frac{dy}{dz} \frac{\partial z}{\partial w_i} = x_i y(1 - y)$$

Error is defined as the squared residuals summed over all training samples

$$E = \frac{1}{2} \sum_{j=1}^m (y^{(j)} - t^{(j)})^2$$

Differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1}^m \frac{dE^{(j)}}{dy^{(j)}} \frac{\partial y^{(j)}}{\partial w_i} = \sum_{j=1}^m \boxed{x_i^{(j)}} \boxed{y^{(j)}(1 - y^{(j)})} \boxed{(y^{(j)} - t^{(j)})}$$


Extra term, the slope of the logistic

Batch update rule

$$w_i \leftarrow w_i - \alpha \sum_{j=1}^m x_i^{(j)} y^{(j)} (1 - y^{(j)}) (y^{(j)} - t^{(j)})$$

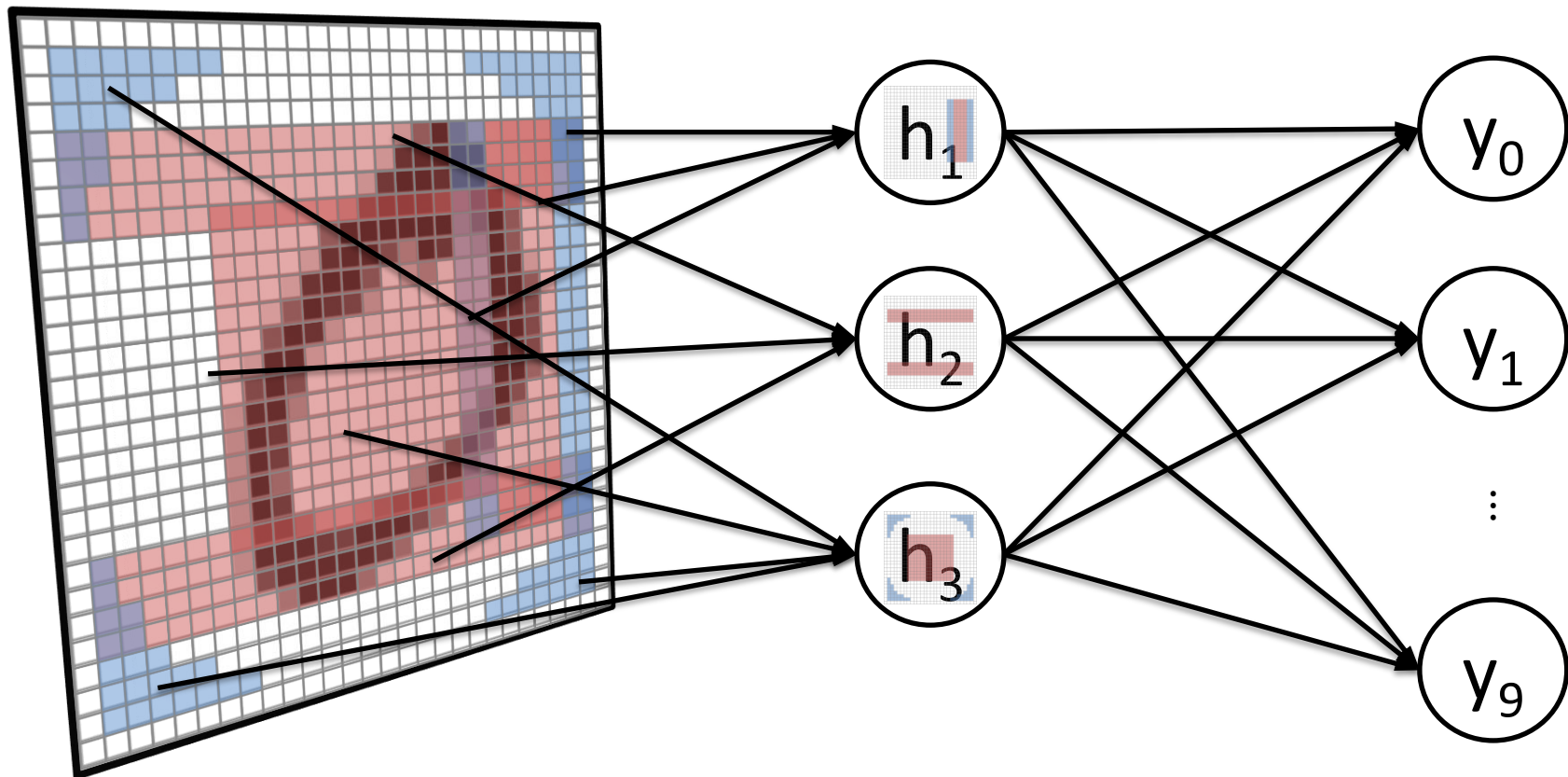
Multi-layer neural networks

LEARNING WITH HIDDEN UNITS

Insight

- Networks without hidden layers (mapping directly the input to an output) are very limited in what they can model
- If we hand-code the features beforehand, we have good algorithms (e.g. the perceptron) to perform learning, but it is difficult to design good features for each task
- It would be nice to learn the right features to use at the same time as learning the classifier
- Hidden units in neural networks are doing just that: encoding features

An architecture with hidden units



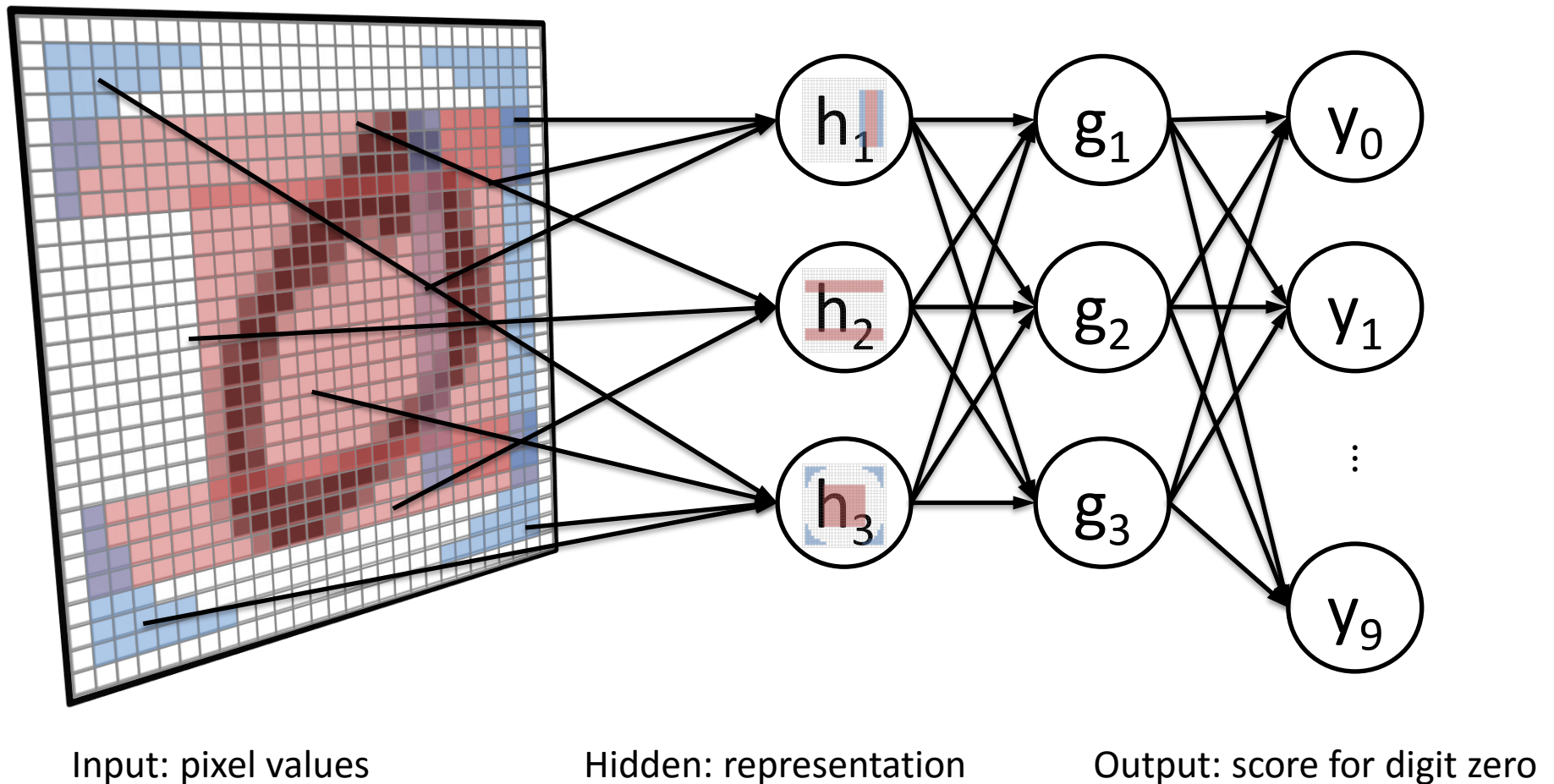
Input: pixel values

Hidden: representation

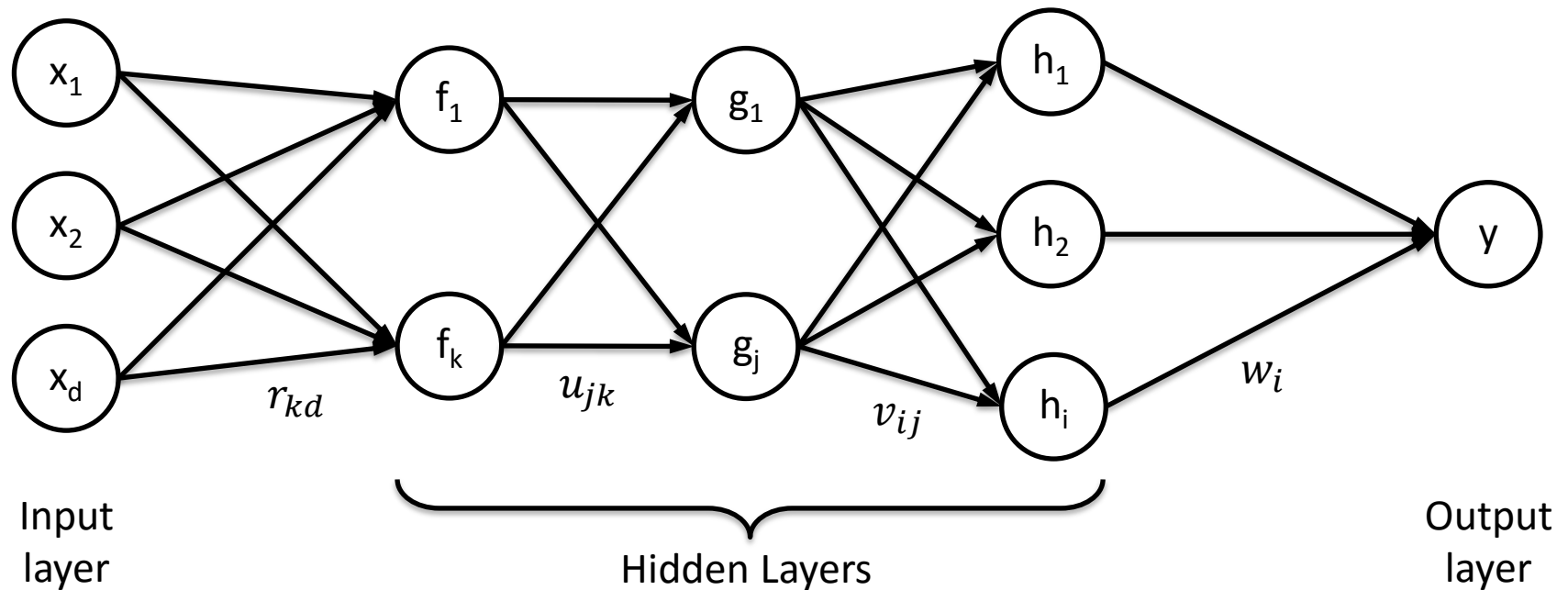
Output: score for digit zero

An architecture with hidden units

Initial hidden layers would give you low-level information, adding subsequent hidden layers the system can encode higher-level features



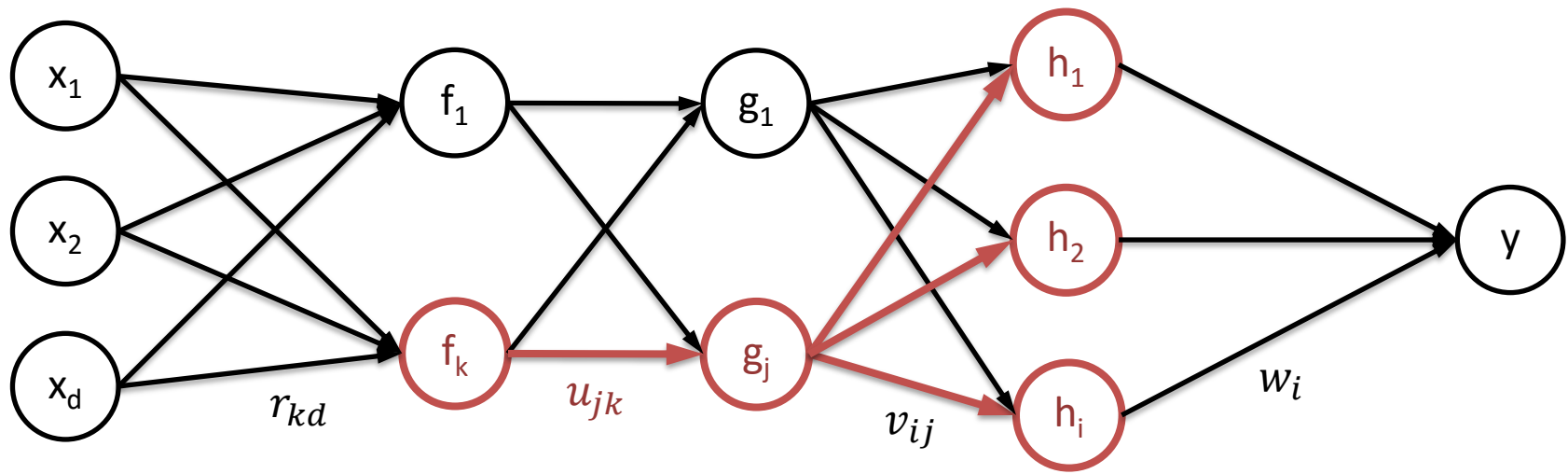
Backpropagation Algorithm



1. Receive new observation $\mathbf{x} = [x_1, x_2, \dots, x_d]$ and target output t
2. Feed-forward: for each unit g_j receiving input from units f_k of the previous layer: $g_j = \sigma(u_{j0} + \sum_k u_{jk} f_k)$
3. Get prediction y and calculate the error $E = \frac{1}{2}(y - t)^2$
4. **Back-propagate error**: for each unit g_j in each layer

Backpropagation Algorithm

How should I change u_{jk} ?



Compute error on g_j

For each u_{jk} that affects g_j

Compute error on u_{jk}

Update the weight

$$\underbrace{\frac{\partial E}{\partial g_j}} = \sum_i \underbrace{h_i(1 - h_i)v_{ij}} \underbrace{\frac{\partial E}{\partial h_i}}$$

Should g_j be higher or lower?

How h_i will change as g_j changes

Was h_i too high or too low?

$$\frac{\partial E}{\partial u_{jk}} = \underbrace{g_i(1 - g_i)f_k}_{\text{How } g_i \text{ will change as } u_{jk} \text{ changes}} \underbrace{\frac{\partial E}{\partial g_j}}_{\text{Should } g_j \text{ be higher or lower?}}$$

How g_i will change as u_{jk} changes

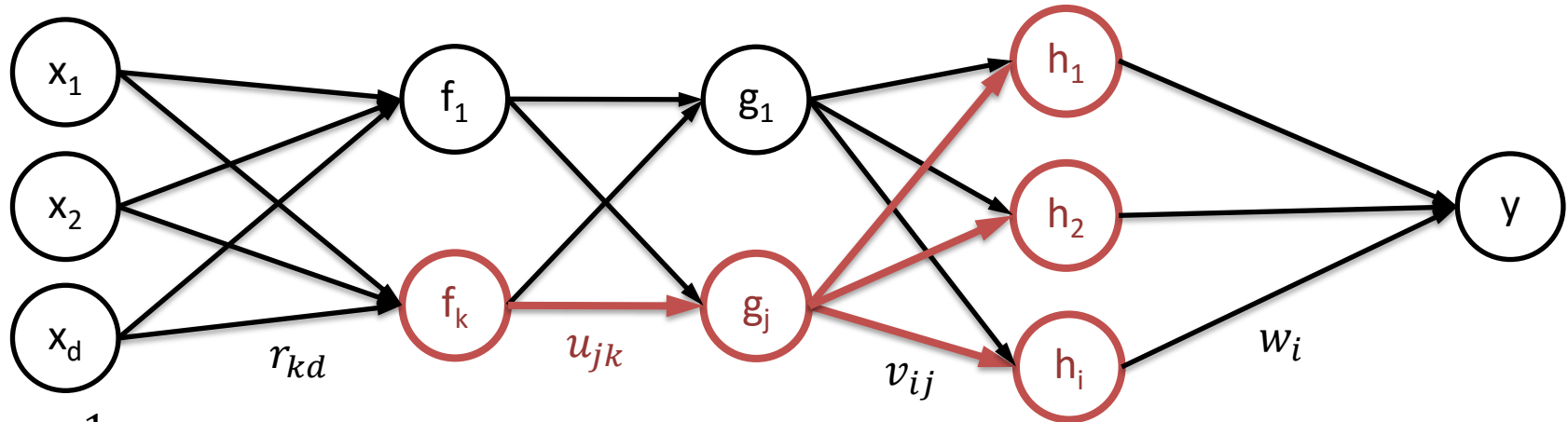
Should g_j be higher or lower?

$$u_{jk} \leftarrow u_{jk} - \underbrace{\alpha}_{\text{Using a learning rate } \alpha} \underbrace{\frac{\partial E}{\partial u_{jk}}}_{\text{According to the error calculated}}$$

Using a learning rate α According to the error calculated

Backpropagation Algorithm

$$\underbrace{f_k = \sigma\left(r_{k0} + \sum_d r_{kd}x_d\right)}_{\text{Input Layer}} \underbrace{g_j = \sigma\left(u_{j0} + \sum_k u_{jk}f_k\right)}_{\text{Hidden Layer}} \underbrace{h_i = \sigma\left(v_{i0} + \sum_j v_{ij}g_j\right)}_{\text{Output Layer}} \underbrace{y = \sigma\left(w_0 + \sum_i w_i h_i\right)}_{\text{Output Layer}}$$



$$E = \frac{1}{2}(y - t)^2$$

$$\frac{\partial E}{\partial h_i} = (y - t)y(1 - y)w_i$$

$$\frac{\partial E}{\partial g_j} = (y - t)y(1 - y) \sum_i w_i h_i (1 - h_i) v_{ij} = \sum_i h_i (1 - h_i) v_{ij} \frac{\partial E}{\partial h_i}$$

$$\frac{\partial E}{\partial u_{jk}} = (y - t)y(1 - y) \sum_i w_i h_i (1 - h_i) \sum_j v_{ij} g_j (1 - g_j) f_k = g_j (1 - g_j) f_k \frac{\partial E}{\partial g_j}$$

Still Not A Learning algorithm

- We know how to compute error derivatives for every weight on a single training point
- We need to see
 - How to use these individual error derivatives to discover a good set of weights
 - How do we make sure we do not overfit

Using all training points

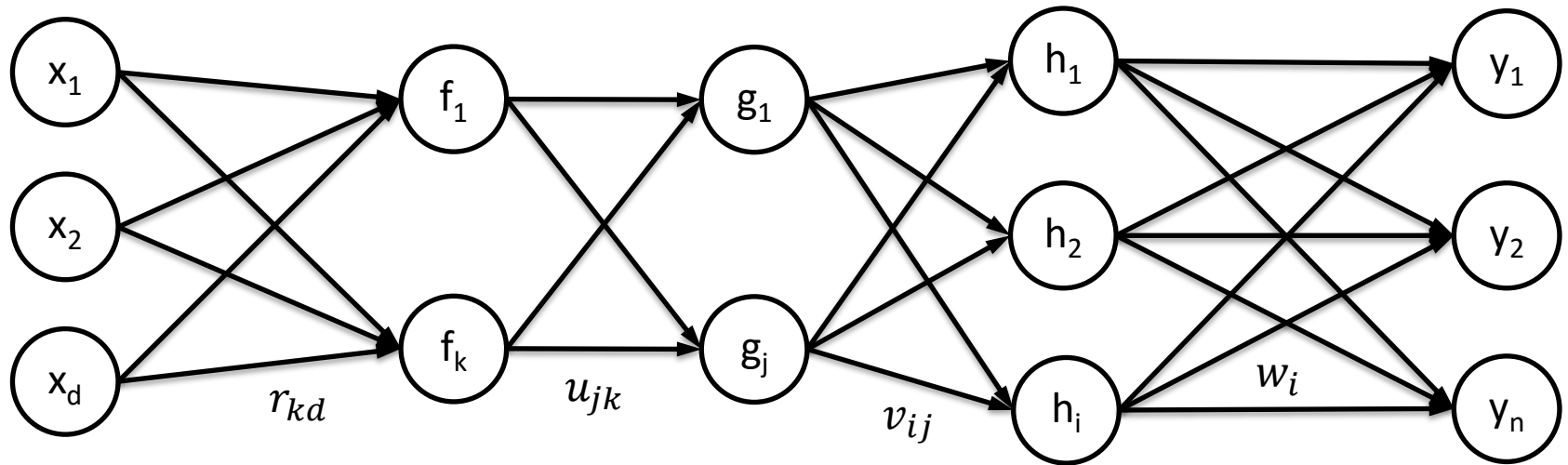
- How often to update the weights
 - **Online**: after each training case.
 - **Full batch**: after a full sweep through the training data.
 - **Mini-batch**: after a small sample of training cases.
- How much to update
 - Use a fixed learning rate?
 - Adapt the global learning rate?
 - Adapt the learning rate on each connection separately?
 - Don't use steepest descent?

Reducing Overfitting

A large number of different methods have been developed

- Weight-decay
- Weight-sharing
- Early stopping
- Model averaging
- Bayesian fitting of neural nets
- Dropout
- Generative pre-training

MULTICLASS NEURAL NETWORKS



1. To predict over multiple classes, add separate output y_c unit for each class c
2. During training, for every training sample $\{\mathbf{x}, c\}$, set $y_c = 1$ and $y_{i \neq c} = 0$
3. Predict class c with the highest y_c
4. If you want to turn the output into a probability, use the soft-max rule:

$$P(c|y) = \frac{e^{y_c}}{\sum_c e^{y_c}}$$

What's Next

		Mondays	Tuesdays				
		16:00 - 18:00	15:00 - 17:00				
Practical Sessions		M	T	W	T	F	Lectures
	Feb	8	9	10	11	12	Introduction and Linear Regression
P0. Introduction to Python, Linear Regression		15	16	17	18	19	Logistic Regression, Normalization
P1. Text non-text classification (Logistic Regression)		22	23	24	25	26	Regularization, Bias-variance decomposition
	Mar	29	1	2	3	4	Subspace methods, dimensionality reduction
		7	8	9	10	11	Probabilities, Bayesian inference
<i>Discussion of intermediate deliverables / project presentations</i>		14	15	16	17	18	Parameter Estimation, Bayesian Classification
		21	22	23	24	25	Easter Week
	Apr	28	29	30	31	1	Clustering, Gaussian Mixture Models, Expectation Maximisation
P2. Feature learning (k-means clustering, NN, bag of words)		4	5	6	7	8	Nearest Neighbour Classification
		11	12	13	14	15	
		18	19	20	21	22	Kernel methods, Support Vector Machines
<i>Discussion of intermediate deliverables / project presentations</i>		25	26	27	28	29	Neural Networks
P3. Text recognition (multi-class classification using SVMs)	May	2	3	4	5	6	Neural Networks
		9	10	11	12	13	Advanced Topics: Deep Nets
		16	17	18	19	20	Advanced Topics: Metric Learning, Preference Learning
<i>Final Project Presentations</i>		23	24	25	26	27	Advanced Topics: Structural Pattern Recognition
	Jun	30	31	1	2	3	Revision

LEGEND		
	Project Follow Up	
	Project presentations	
	Lectures	
	Project Deliverable due date	
	Vacation / No Class	