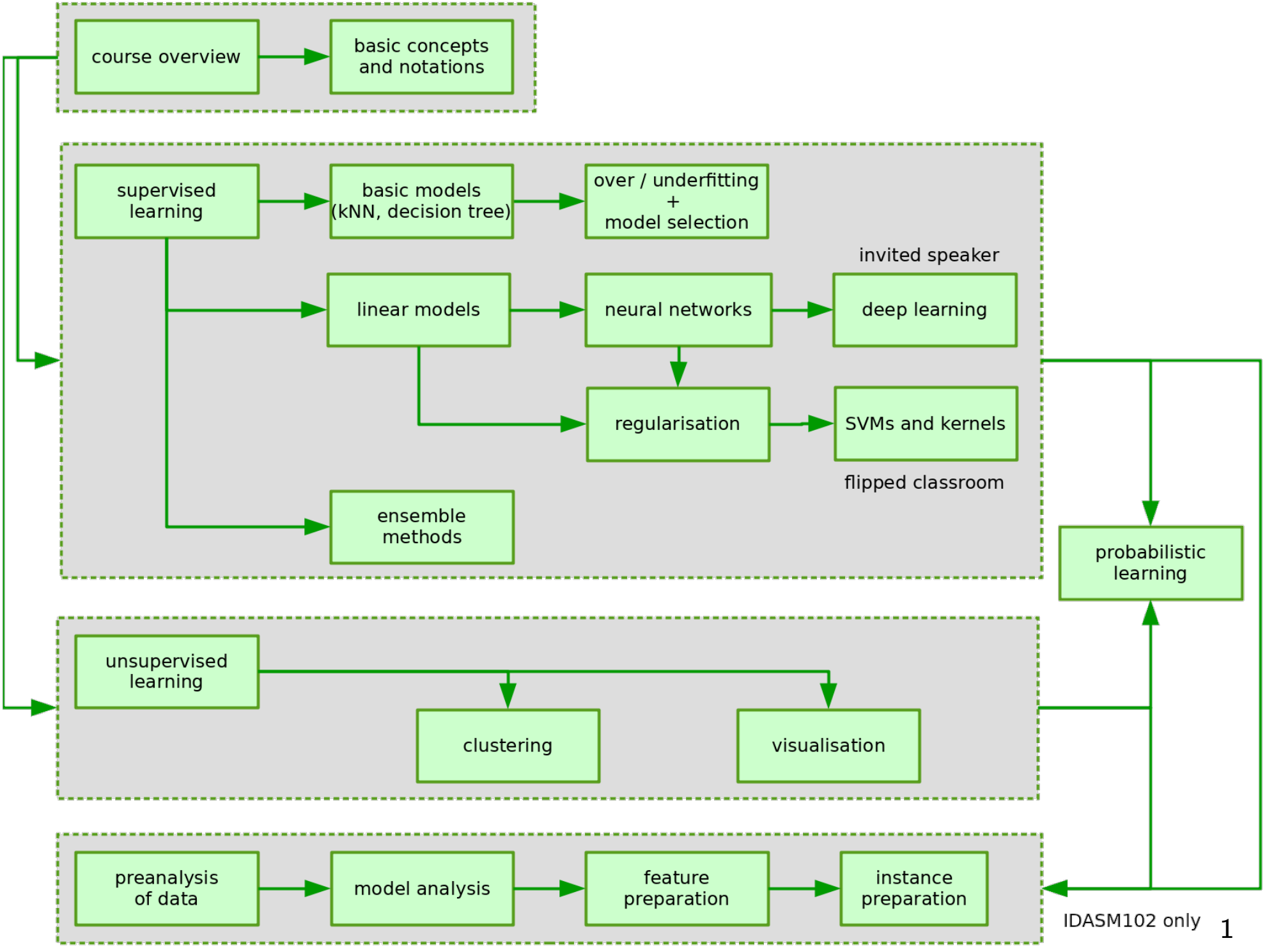
Machine Learning: Lesson 7

Single and Multilayer Artificial Neural Networks

Benoît Frénay - Faculty of Computer Science

UNIVERSITÉ DE NAMUR

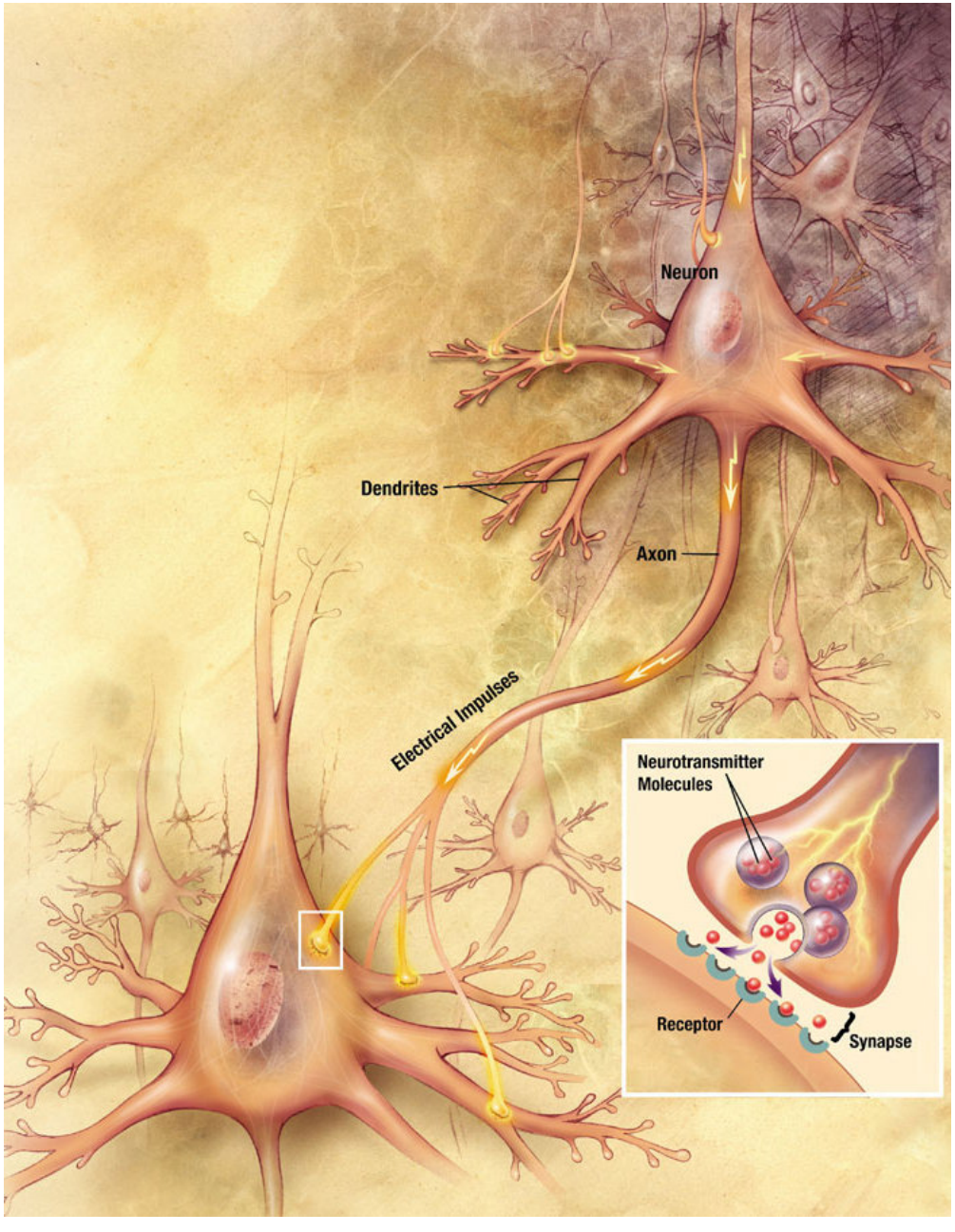

course overview
basic concepts and notations
supervised learning
basic models (kNN, decision tree)
over / underfitting + model selection
invited speaker
linear models
neural networks
deep learning
regularisation
SVMs and kernels
flipped classroom
ensemble methods
probabilistic learning
unsupervised learning
clustering
visualisation
preanalysis of data
model analysis
feature preparation
instance preparation
IDASM102 only

Outline of this Lesson

- a bit of history
- single-layer neural networks
  - simple adaptation rules
  - the first AI winter
- multi-layer neural networks
  - error back-propagation
  - classification with multi-layer perceptrons
- the future of neural networks

A Bit of History

Neurons and Neural Networks in the Brain



Programmable Computers vs. Biological Neural Networks

Digital Computer

- process one operation at a time (without threading and multicore)
- each operation is very fast ($\pm$0.3 ns for Intel Core i7-5960 3.5 GHhz)
- perform many basic operations (Tianhe-2 achieves 33.86 petaflops)
- quite large and energy intensive (720 m^2 and 24 MW for Tianhe-2)
- must be carefully programmed, good at tasks that require computing
- not resilient to physical damage, hardware failure and software faults

Human Brain

- highly parallel (10^{11} neurons with on average 10^4 connections)
- each operation is slow (fastest neurons switch in $\pm$1 ms)
- small (1.5 kg and 1200 cm^3) and energy efficient (12 W, 20% total)
- learn from experience and can recover from physical damage
- good at tasks related to vision, control, decision under uncertainty...

Programmable Computers vs. Biological Neural Networks

Digital Computer

- process one operation at a time (without threading and multicore)
- each operation is very fast (± 0.3 ns for Intel Core i7-5960 3.5 GHhz)
- perform many basic operations (Tianhe-2 achieves 33.86 petaflops)
- quite large and energy intensive (720 m^2 and 24 MW for Tianhe-2)
- must be carefully programmed, good at tasks that require computing
- not resilient to physical damage, hardware failure and software faults

Human Brain

- highly parallel (10^{11} neurons with on average 10^4 connections)
- each operation is slow (fastest neurons switch in ± 1 ms)
- small (1.5 kg and 1200 cm^3) and energy efficient (12 W, 20% total)
- learn from experience and can recover from physical damage
- good at tasks related to vision, control, decision under uncertainty...

Programmable Computers vs. Biological Neural Networks

Digital Computer

- process one operation at a time (without threading and multicore)
- each operation is very fast (± 0.3 ns for Intel Core i7-5960 3.5 GHhz)
- perform many basic operations (Tianhe-2 achieves 33.86 petaflops)
- quite large and energy intensive (720 m^2 and 24 MW for Tianhe-2)
- must be carefully programmed, good at tasks that require computing
- not resilient to physical damage, hardware failure and software faults

Human Brain

- highly parallel (10^{11} neurons with on average 10^4 connections)
- each operation is slow (fastest neurons switch in ± 1 ms)
- small (1.5 kg and 1200 cm^3) and energy efficient (12 W, 20% total)
- learn from experience and can recover from physical damage
- good at tasks related to vision, control, decision under uncertainty...

Programmable Computers vs. Biological Neural Networks

Digital Computer

- process one operation at a time (without threading and multicore)
- each operation is very fast ($\pm$0.3 ns for Intel Core i7-5960 3.5 GHhz)
- perform many basic operations (Tianhe-2 achieves 33.86 petaflops)
- quite large and energy intensive (720 m^2 and 24 MW for Tianhe-2)
- must be carefully programmed, good at tasks that require computing
- not resilient to physical damage, hardware failure and software faults

Human Brain

- highly parallel (10^{11} neurons with on average 10^4 connections)
- each operation is slow (fastest neurons switch in $\pm$1 ms)
- small (1.5 kg and 1200 cm^3) and energy efficient (12 W, 20% total)
- learn from experience and can recover from physical damage
- good at tasks related to vision, control, decision under uncertainty...

Programmable Computers vs. Biological Neural Networks

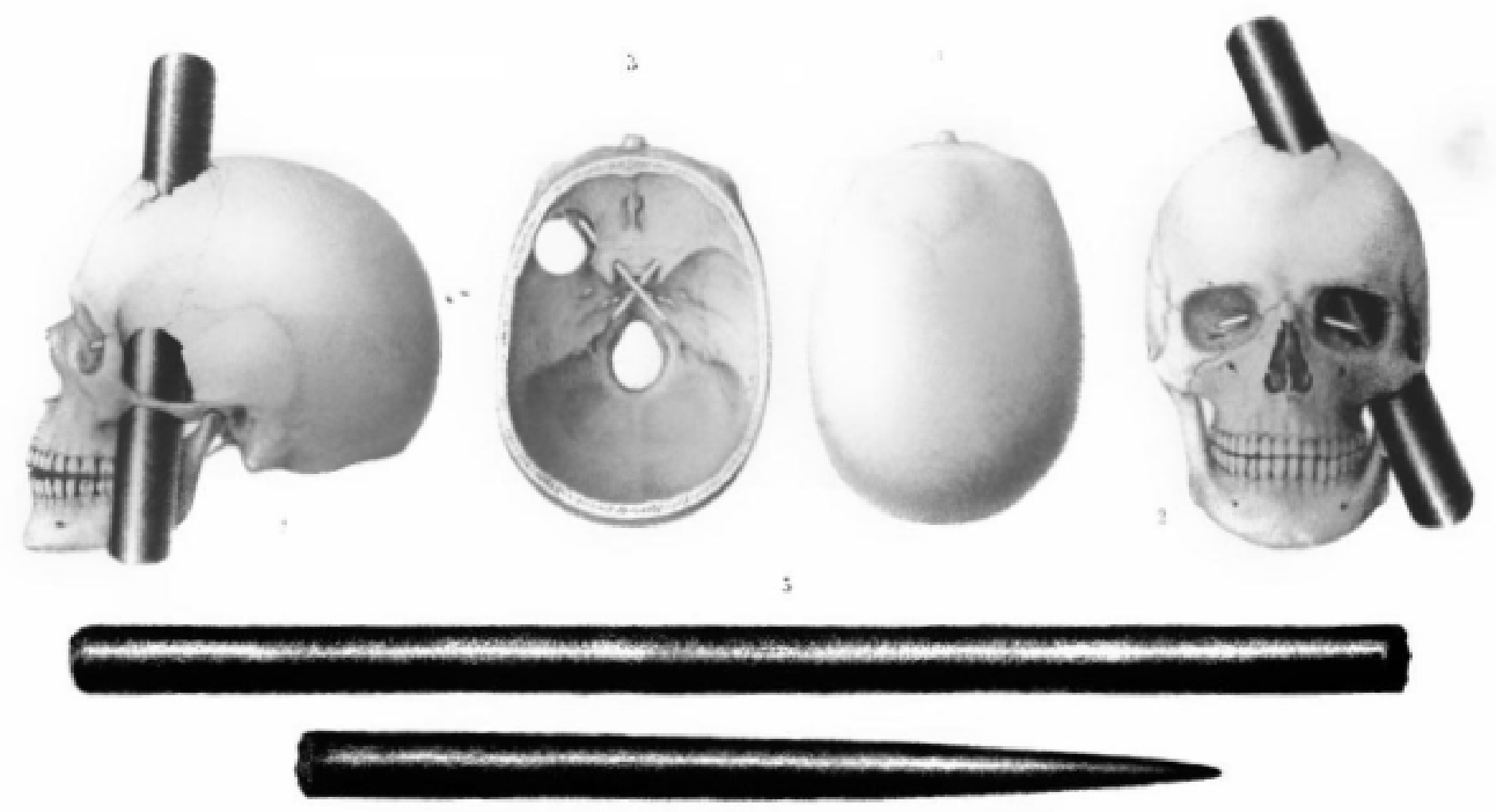
Digital Computer

- process one operation at a time (without threading and multicore)
- each operation is very fast ($\pm$0.3 ns for Intel Core i7-5960 3.5 GHhz)
- perform many basic operations (Tianhe-2 achieves 33.86 petaflops)
- quite large and energy intensive (720 m^2 and 24 MW for Tianhe-2)
- must be carefully programmed, good at tasks that require computing
- not resilient to physical damage, hardware failure and software faults

Human Brain

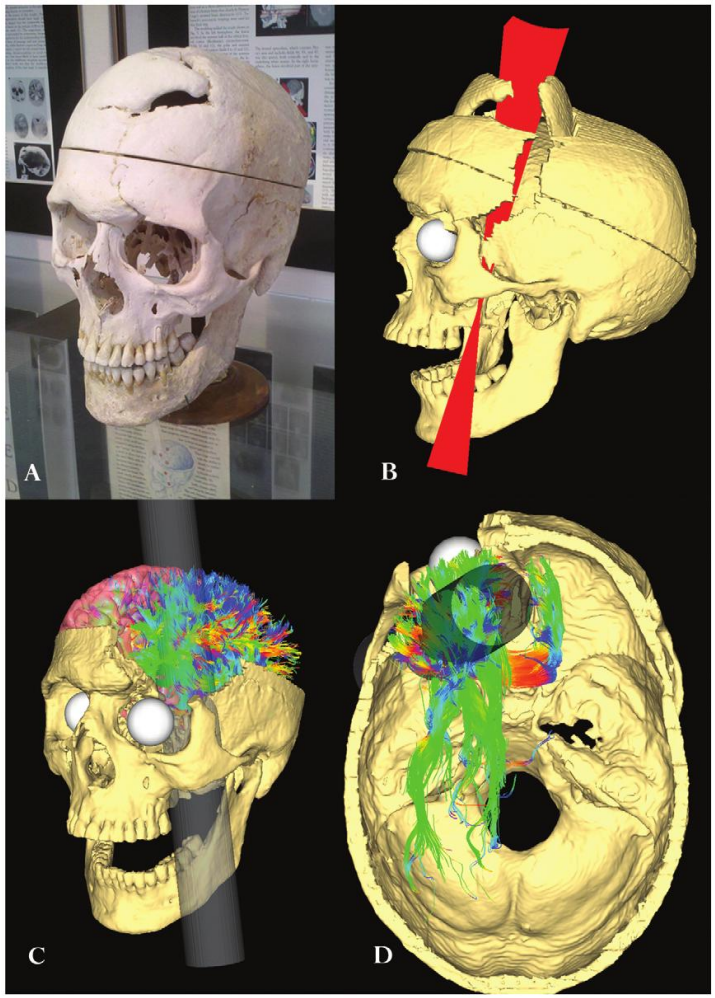
- highly parallel (10^{11} neurons with on average 10^4 connections)
- each operation is slow (fastest neurons switch in $\pm$1 ms)
- small (1.5 kg and 1200 cm^3) and energy efficient (12 W, 20% total)
- learn from experience and can recover from physical damage
- good at tasks related to vision, control, decision under uncertainty...

Recovery Ability of the Human Brain: Phineas P. Gage



Bigelow, Henry Jacob (July 1850). "Dr. Harlow's Case of Recovery from the Passage of an Iron Bar through the Head". American Journal of the Medical Sciences 20: 13-22. An early paper recounting the Phineas Gage case.

Recovery Ability of the Human Brain: Phineas P. Gage



"4% of the cortex was intersected by the rod's passage, 11% of total white matter was also damaged"

Van Horn JD, Irimia A, Torgerson CM, Chambers MC, Kikinis R, et al. (2012) Mapping Connectivity Damage in the Case of Phineas Gage. PLoS ONE 7(5):e37454. doi:10.1371/journal.pone.0037454

Artificial Neural Networks for Machine Learning

artificial neural networks = software of hardware architectures that (loosely) mimic the behaviour and abilities of real biological neural networks

Goals

- simulation: study and model biological neurons and neural networks
- engineering: build powerful computational tools to solve problems

here, focus on feedfoward multi-layer neural networks (no recurrence)

- can we solve classification/regression tasks with such ANNs ?
- how can we implement learning from experience in ANNs ?
- other structures include recurrent networks, RBF units, etc.

Artificial Neural Networks for Machine Learning

artificial neural networks = software of hardware architectures that (loosely) mimic the behaviour and abilities of real biological neural networks

Goals

- simulation: study and model biological neurons and neural networks
- engineering: build powerful computational tools to solve problems

here, focus on feedfoward multi-layer neural networks (no recurrence)

- can we solve classification/regression tasks with such ANNs ?
- how can we implement learning from experience in ANNs ?
- other structures include recurrent networks, RBF units, etc.

Artificial Neural Networks for Machine Learning

artificial neural networks = software of hardware architectures that (loosely) mimic the behaviour and abilities of real biological neural networks

Goals

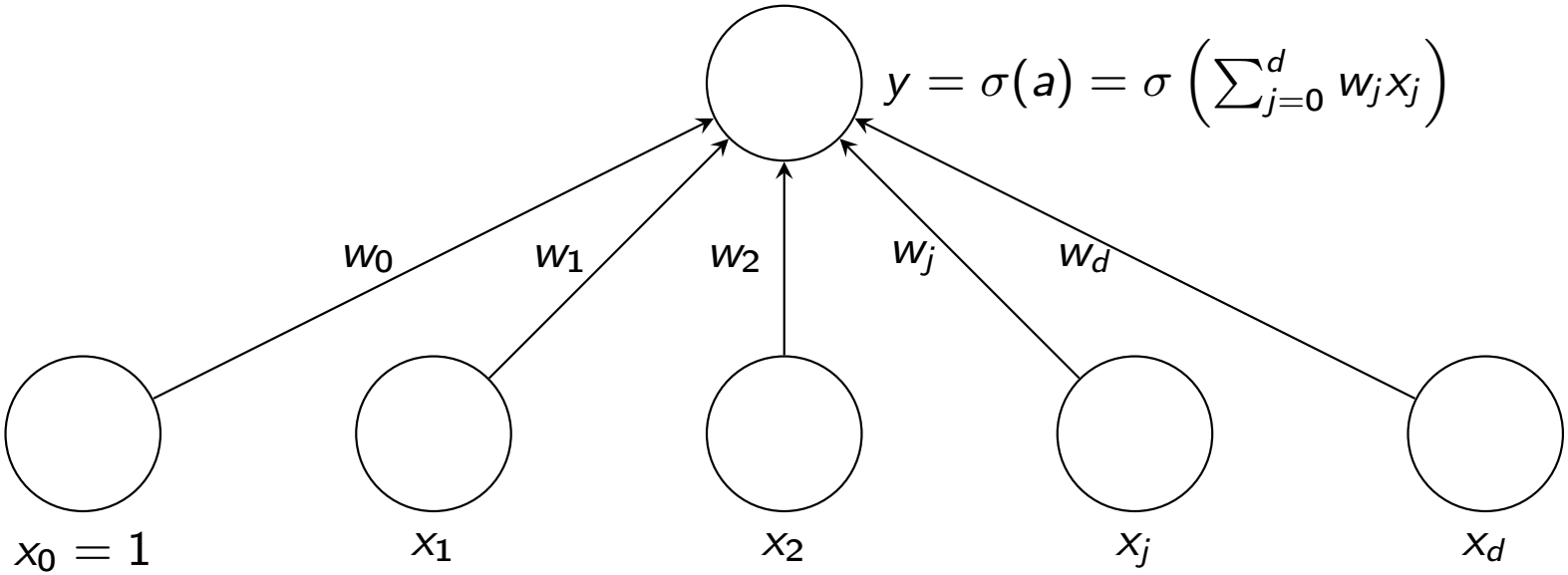
- simulation: study and model biological neurons and neural networks
- engineering: build powerful computational tools to solve problems

here, focus on feedfoward multi-layer neural networks (no recurrence)

- can we solve classification/regression tasks with such ANNs ?
- how can we implement learning from experience in ANNs ?
- other structures include recurrent networks, RBF units, etc.

Single-Layer Neural Networks

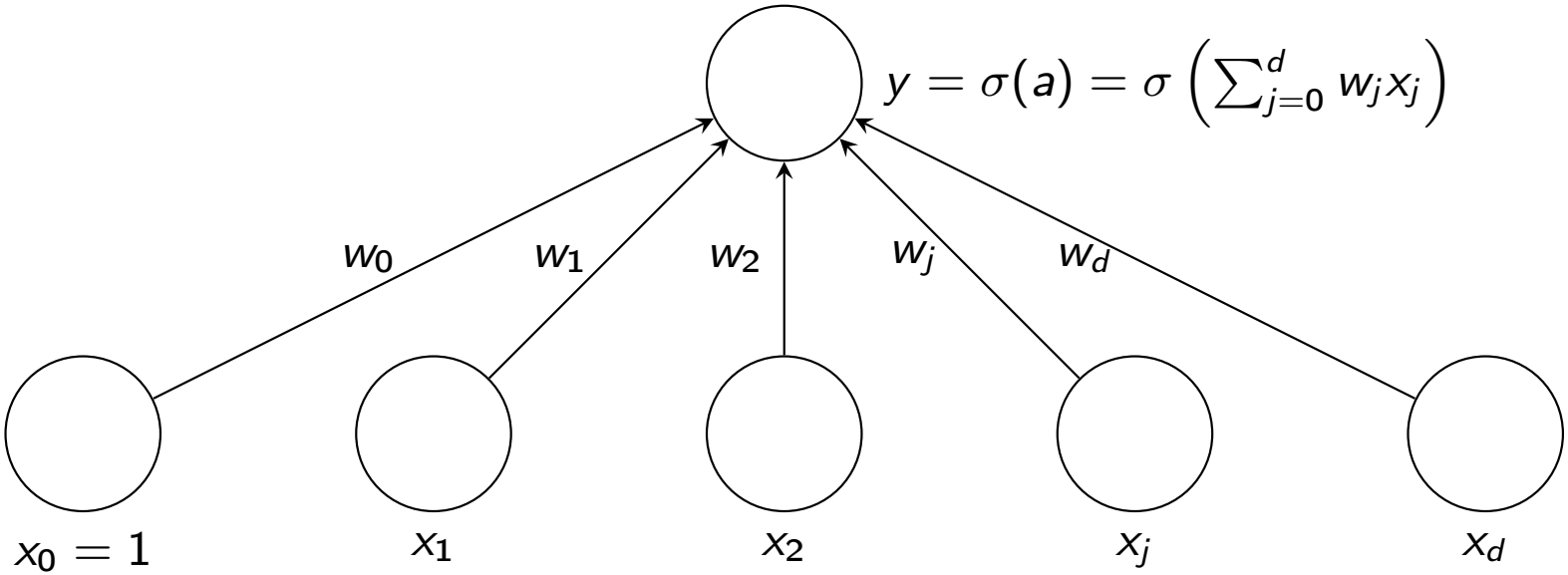
A Simple Model of Neuron



Analogy with real neurons

- input units x_j = sensory neurons (except x_0 to simplify notation)
- connection = synapse from dendrite to axon (except w_0 = bias)
- weight = synaptic weight/strength of influence (positive or negative)
- activation = total activation of the axon resulting from summation

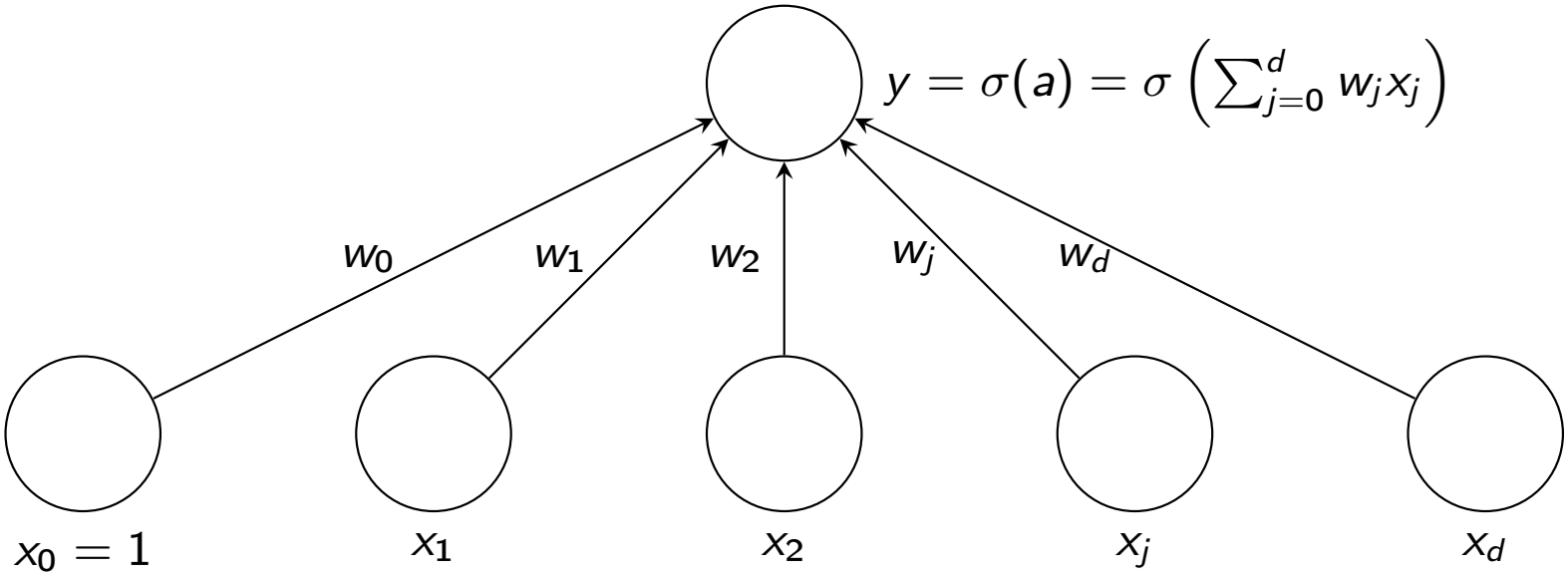
A Simple Model of Neuron



Analogy with real neurons

- input units x_j = sensory neurons (except x_0 to simplify notation)
- connection = synapse from dendrite to axon (except w_0 = bias)
- weight = synaptic weight/strength of influence (positive or negative)
- activation = total activation of the axon resulting from summation

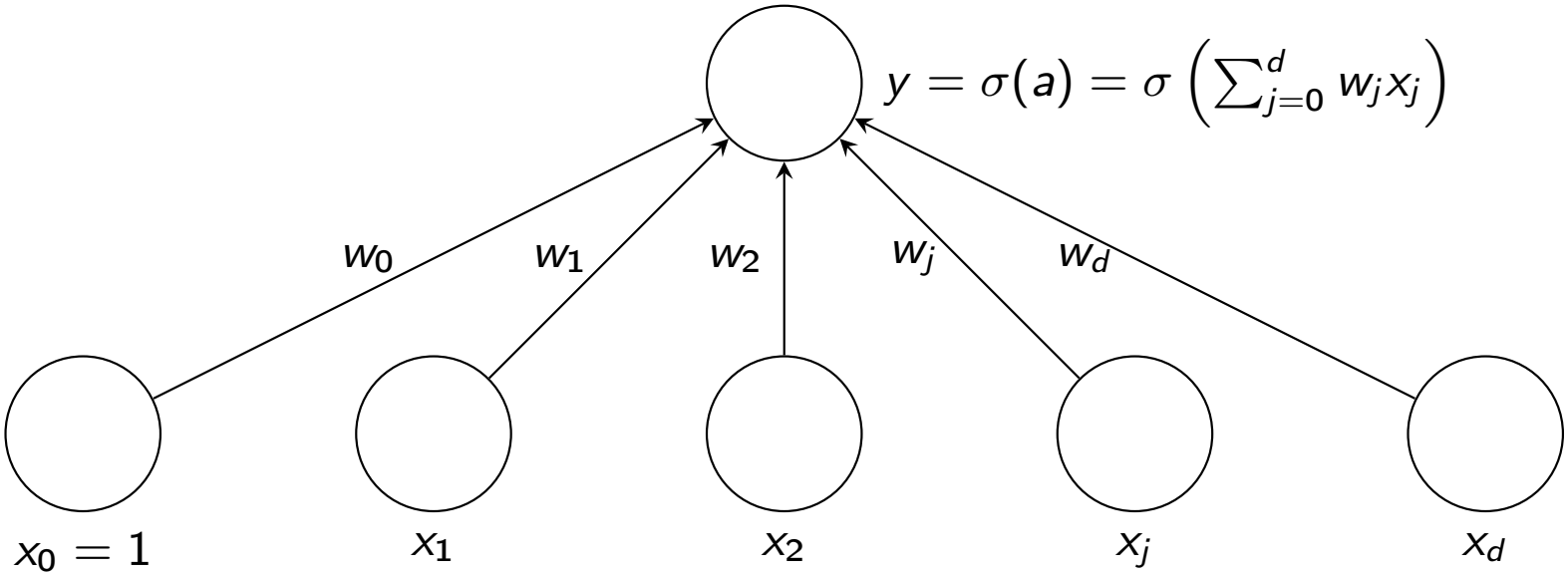
A Simple Model of Neuron



Analogy with real neurons

- input units x_j = sensory neurons (except x_0 to simplify notation)
- connection = synapse from dendrite to axon (except w_0 = bias)
- weight = synaptic weight/strength of influence (positive or negative)
- activation = total activation of the axon resulting from summation

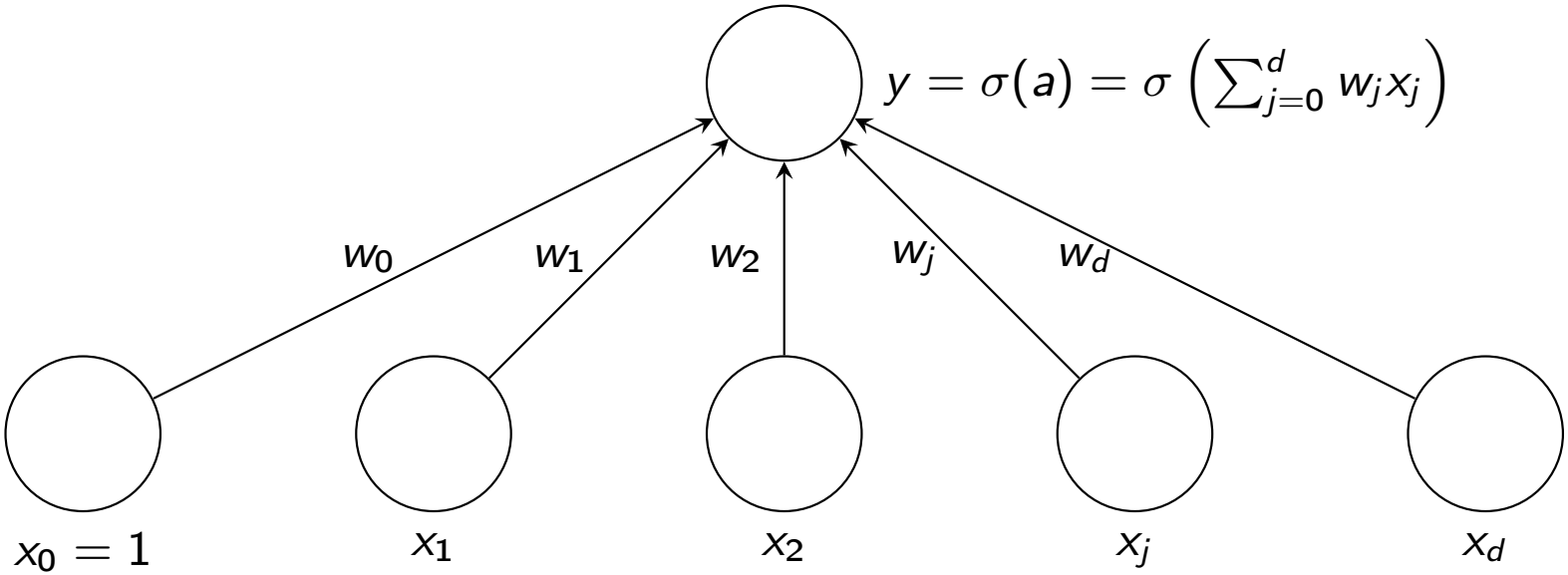
A Simple Model of Neuron



Analogy with real neurons

- input units x_j = sensory neurons (except x_0 to simplify notation)
- connection = synapse from dendrite to axon (except w_0 = bias)
- weight = synaptic weight/strength of influence (positive or negative)
- activation = total activation of the axon resulting from summation

A Simple Model of Neuron



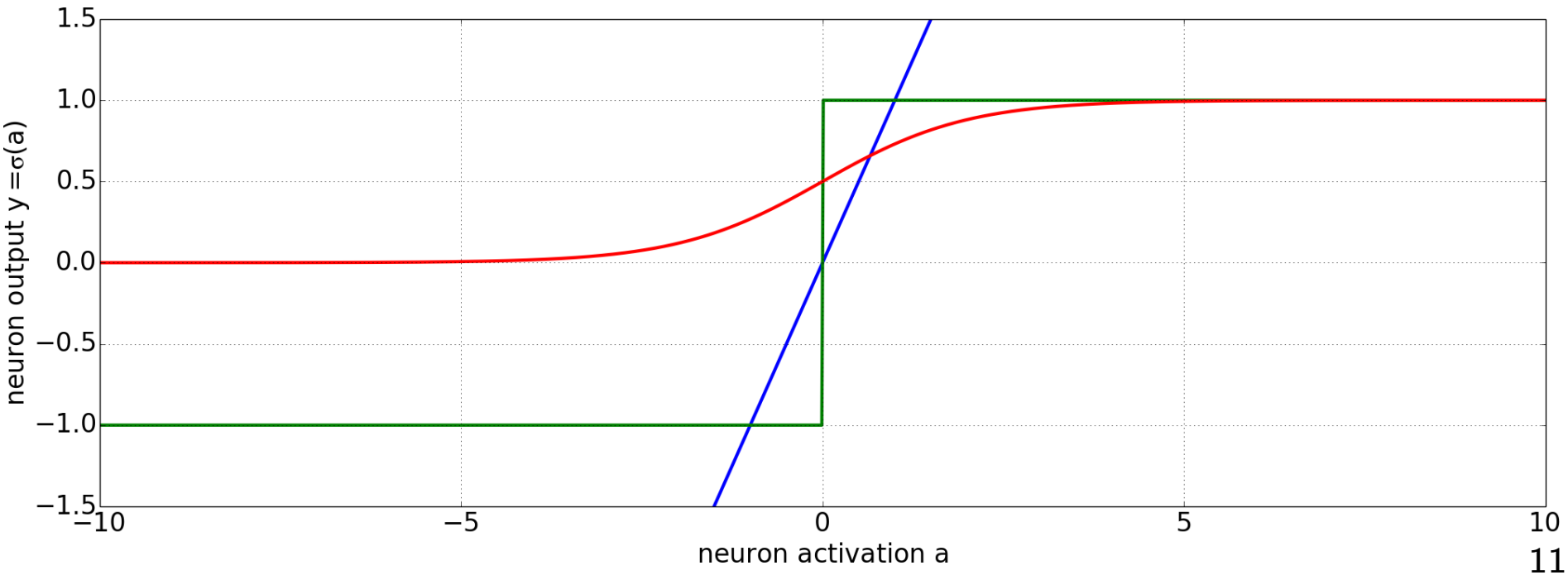
Analogy with real neurons

- input units x_j = sensory neurons (except x_0 to simplify notation)
- connection = synapse from dendrite to axon (except w_0 = bias)
- weight = synaptic weight/strength of influence (positive or negative)
- activation = total activation of the axon resulting from summation

A Simple Model of Neuron

Simple perceptron

- linear model with identity activation function $\sigma(a) = a$
- sigmoid unit with sigmoid activation function $\sigma(a) = \frac{1}{1+\exp(-a)}$
- linear discriminant if $\sigma(a) = \text{sign}(a) = \begin{cases} +1 & \text{if } a > 0 \\ -1 & \text{otherwise} \end{cases}$



Simple Adaptation Rules

Learning with a Single-Unit Perceptron

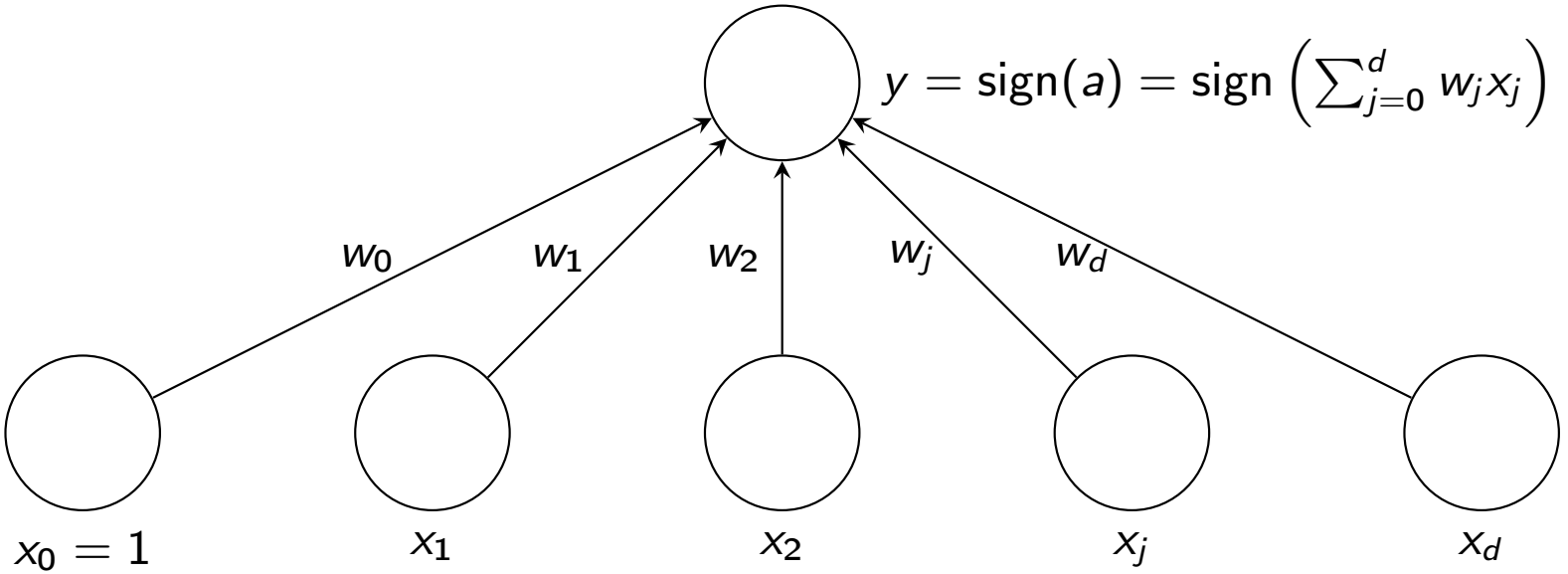
Classification tasks

percetron learning rule $\Rightarrow$ learning weights for thresholded units (sign)

Regression tasks

delta rule $\Rightarrow$ learning weights for non-linear continuous activation function

Perceptron Learning Rule



Perceptron for binary classification

simple linear discriminant with $\sigma(a) = \text{sign}(a)$ for binary classification

- to simplify notations, dummy feature $x_0 = 1$ is introduced
- y depends on $\sum_{j=0}^{d} w_j x_j = \mathbf{w} \cdot \mathbf{x} =$ alignment/similarity of $\mathbf{w}$ and $\mathbf{x}$
- you can think of $\mathbf{w}$ as "the" pattern that we are learning to detect

Perceptron Learning Rule

Input: dataset $\mathcal{D} = \{(\mathbf{x}_i, t_i)\}$
Output: weights that linearly separate both classes

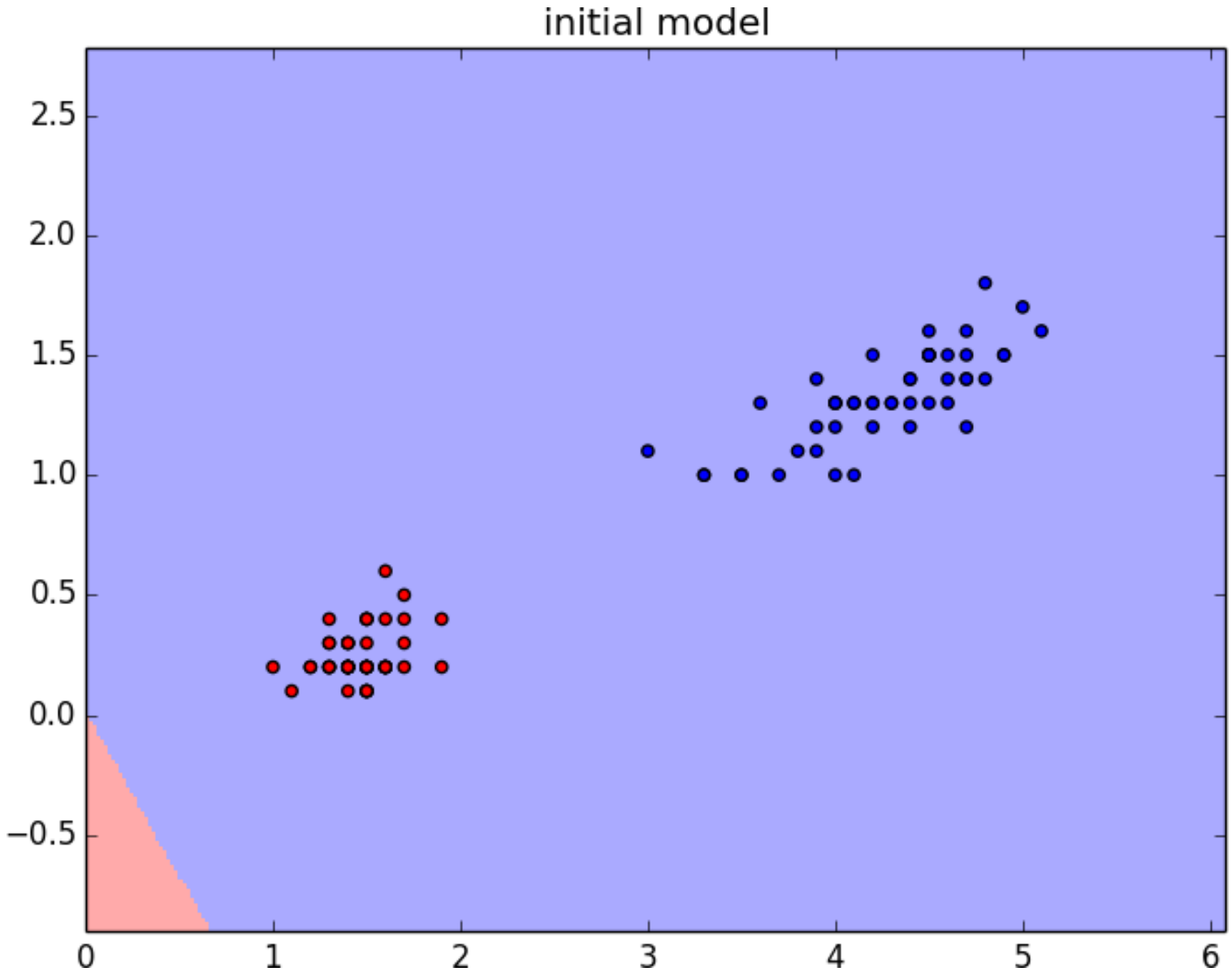
add $x_0 = 1$ in front of each vector $\mathbf{x}$ (trick to not care about bias)
initialise perceptron weights $\mathbf{w}_t = (w_0, w_1 \ldots w_d)$ randomly

while at least one instance is misclassified **do**
  for each instance $\mathbf{x}_i$ with target t_i in dataset **do**
    compute perceptron prediction $y_i = \text{sign}\, \mathbf{w}_t^T \mathbf{x}_i$

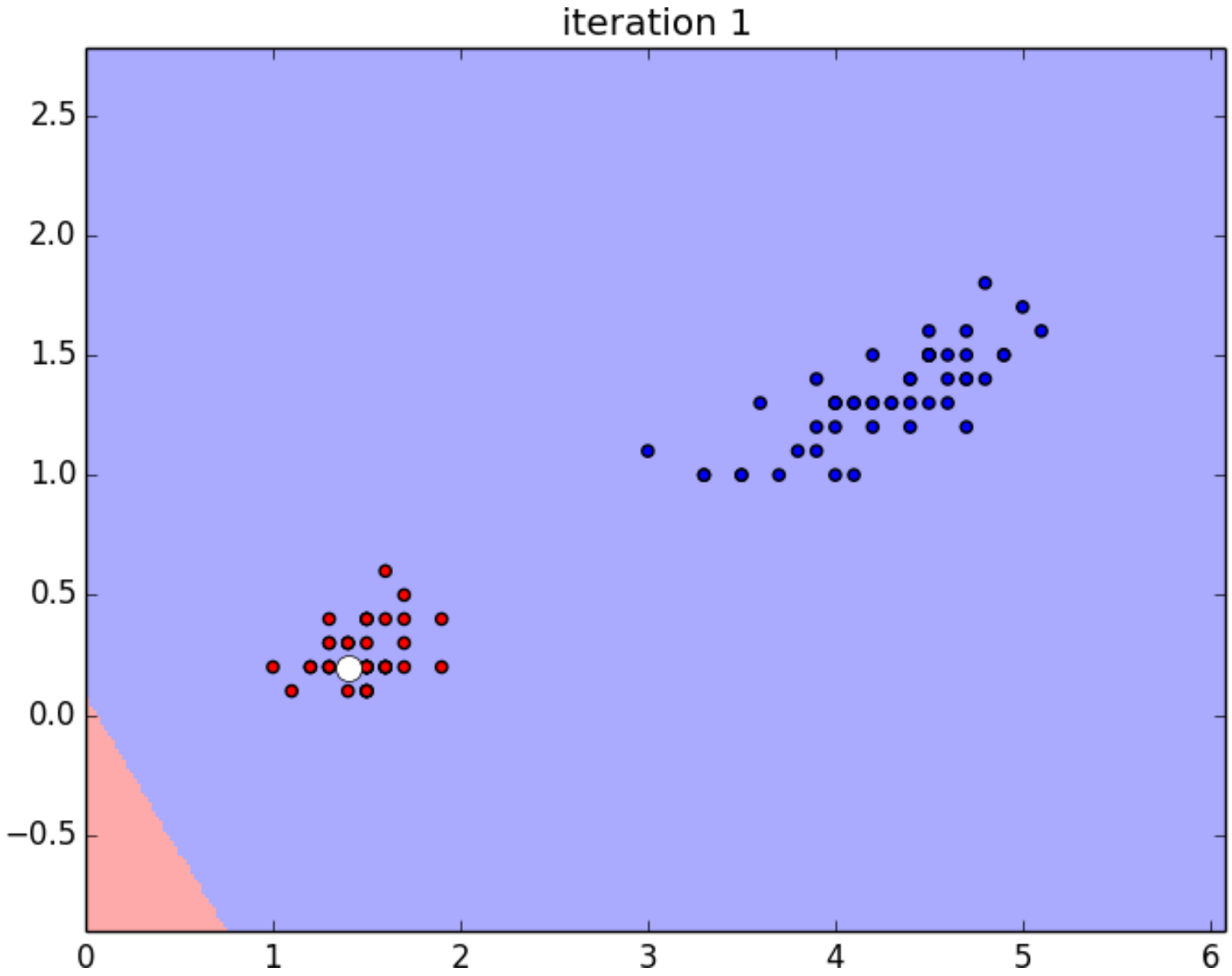
    if instance is misclassified by perceptron, i.e. $y_i \neq t_i$ **then**
      $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t (t_i - y_i) \mathbf{x}_i$
    end if
  end for
end while

return $\mathbf{w}_t$

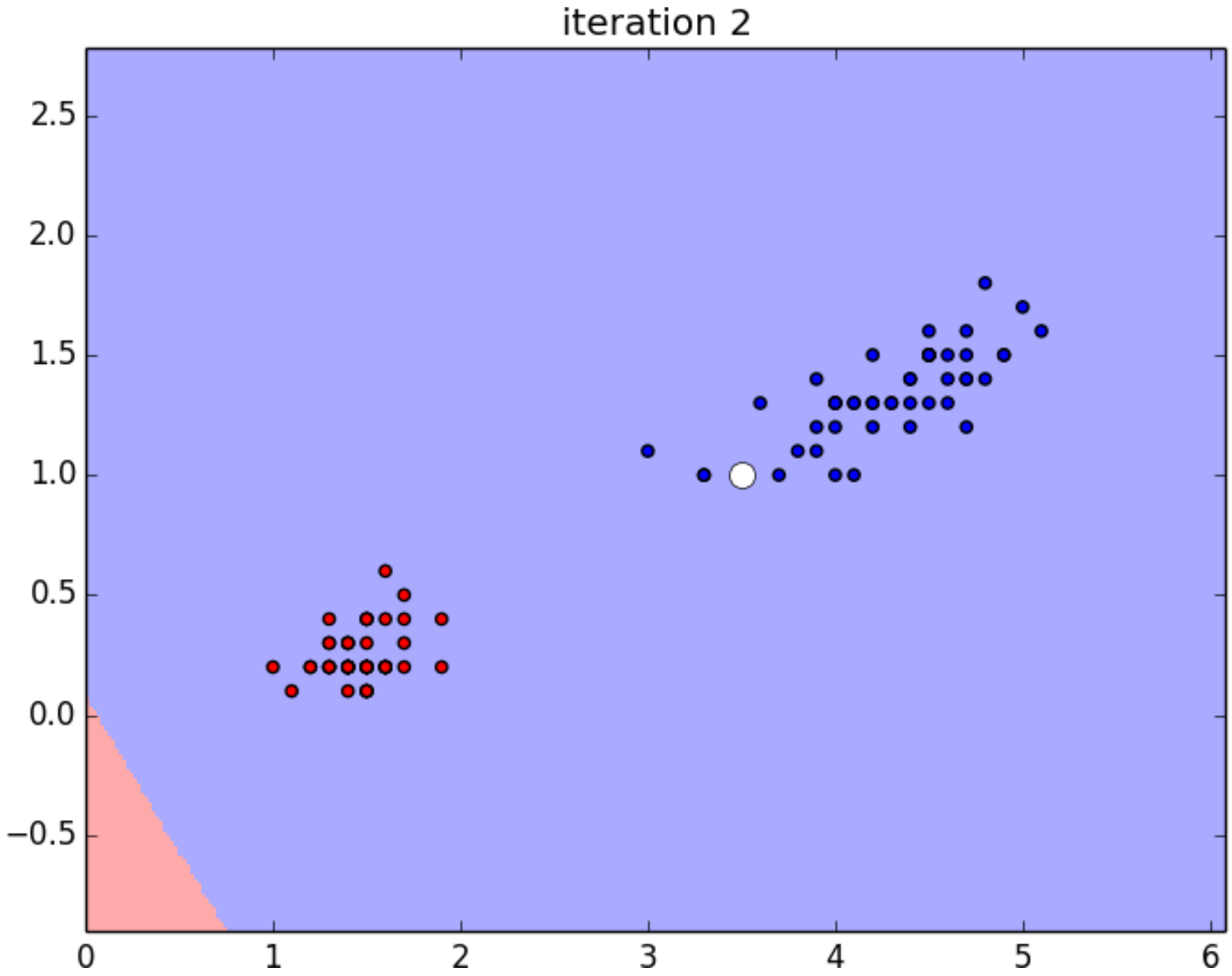
Perceptron Learning Rule



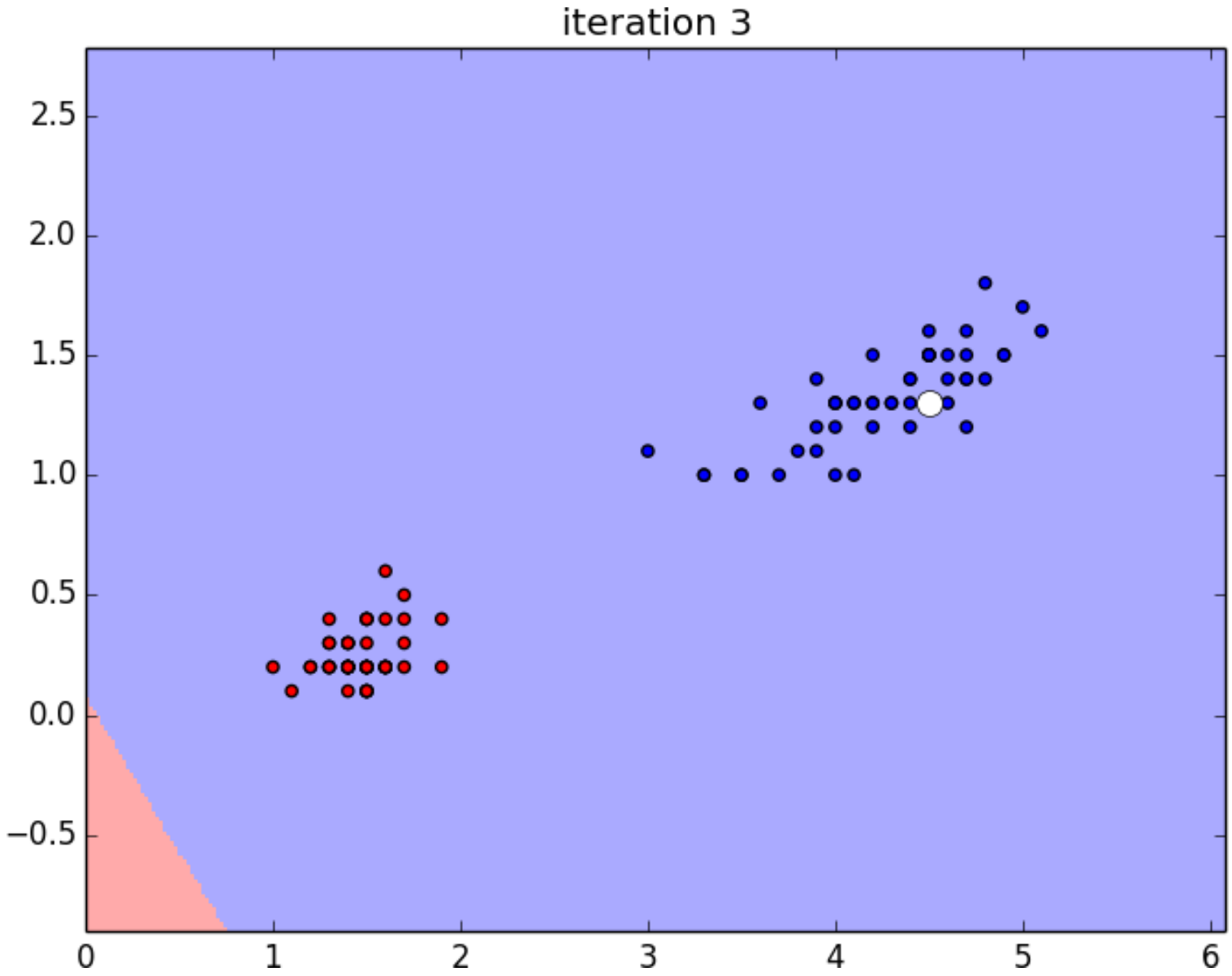
Perceptron Learning Rule



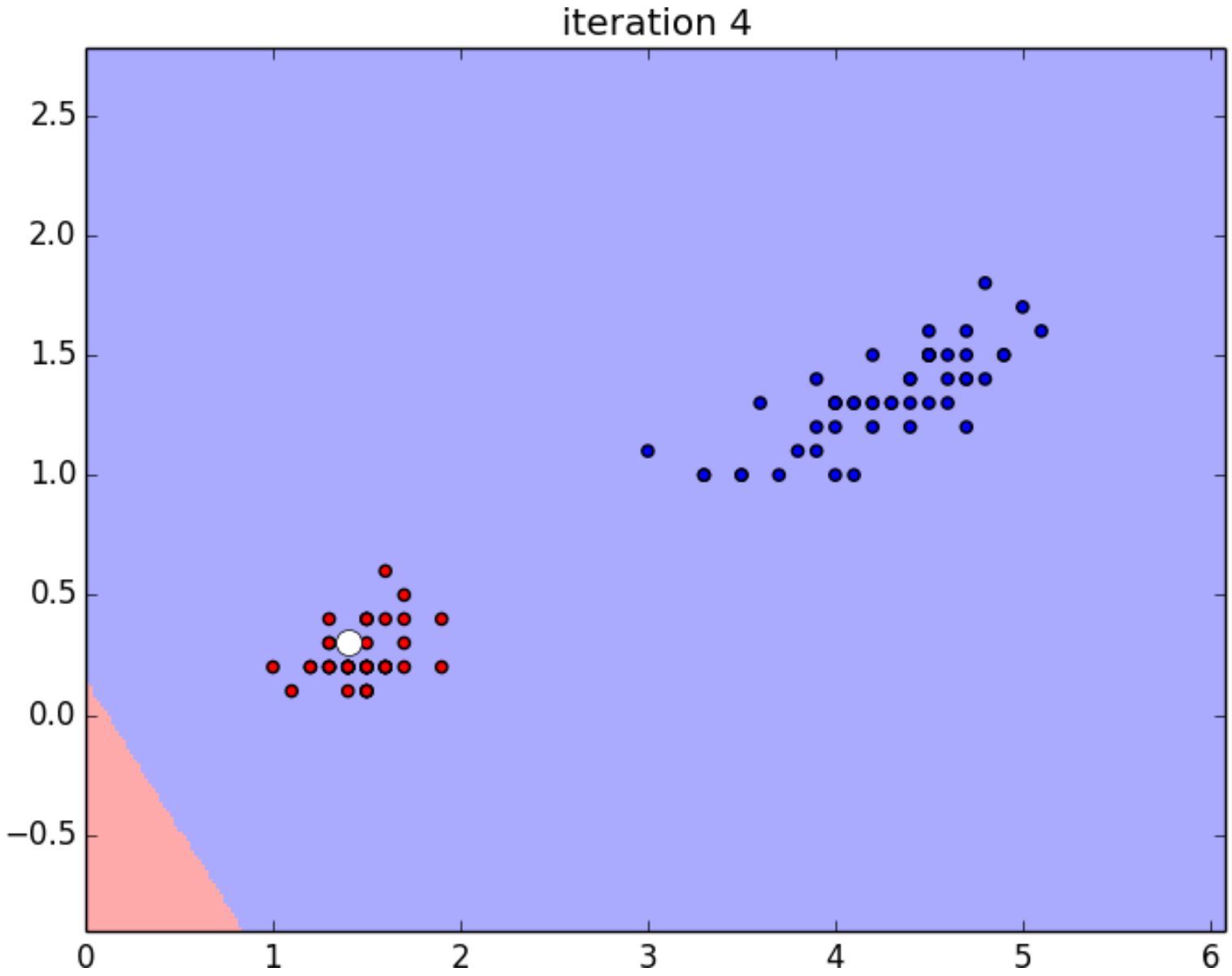
Perceptron Learning Rule



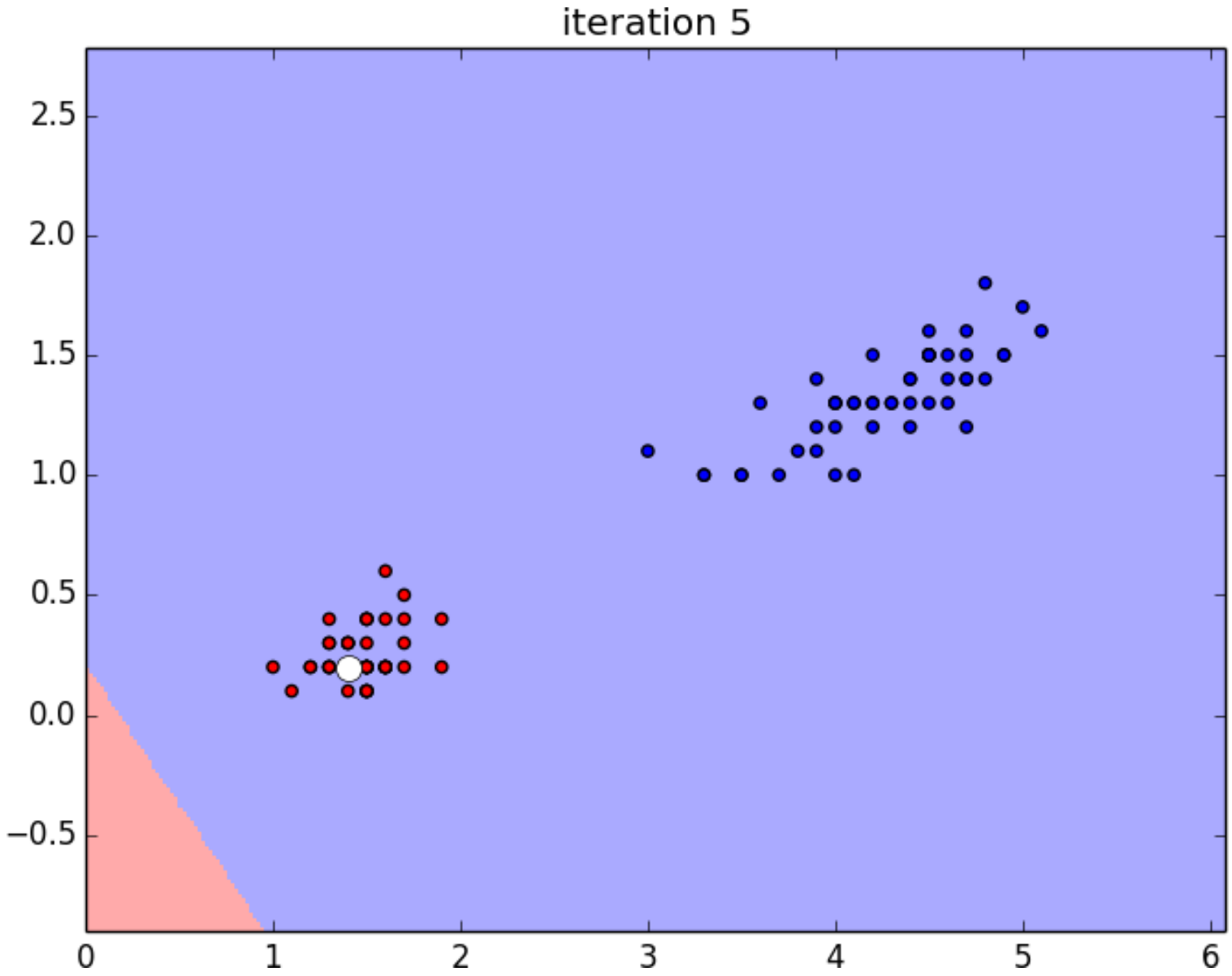
Perceptron Learning Rule



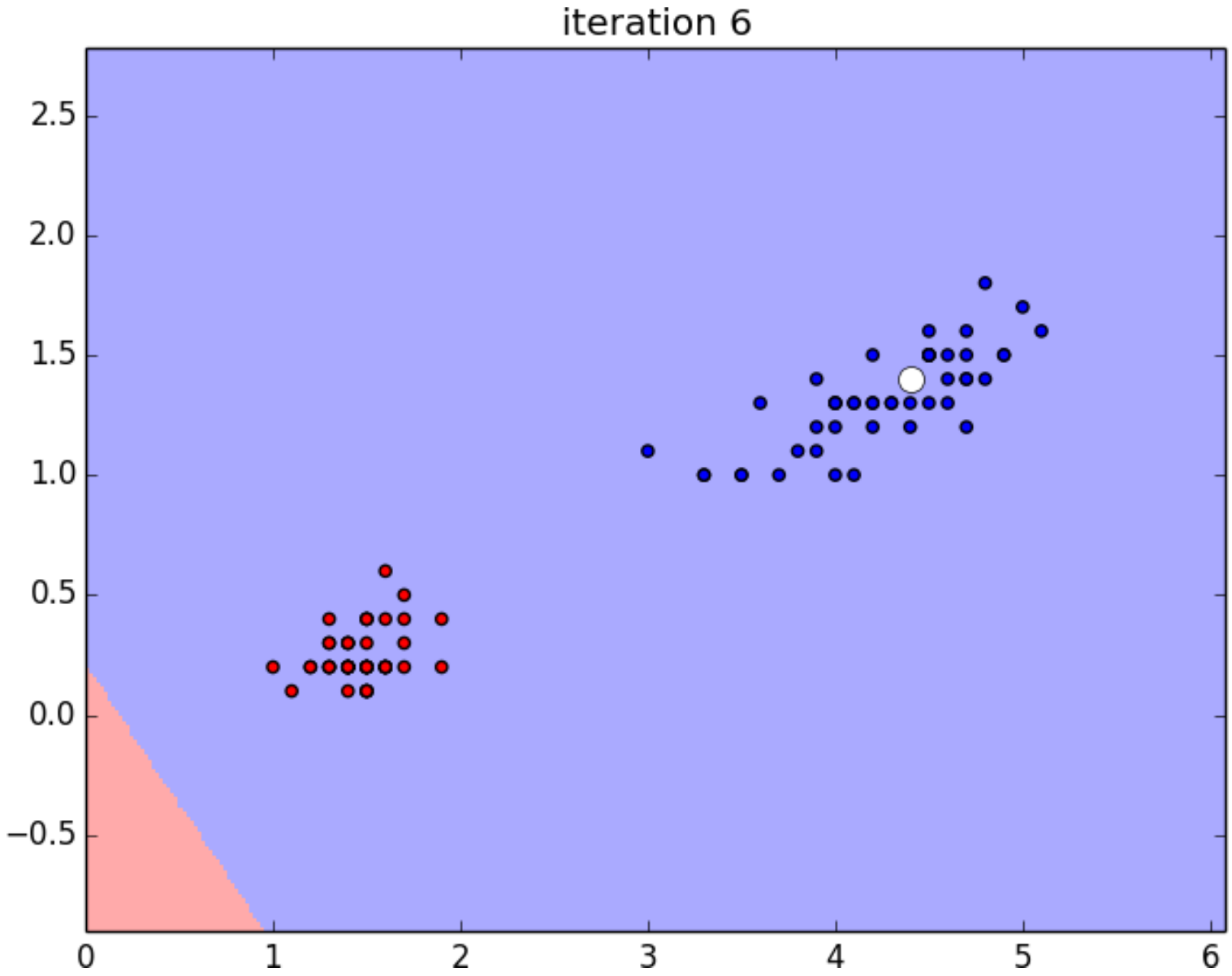
Perceptron Learning Rule



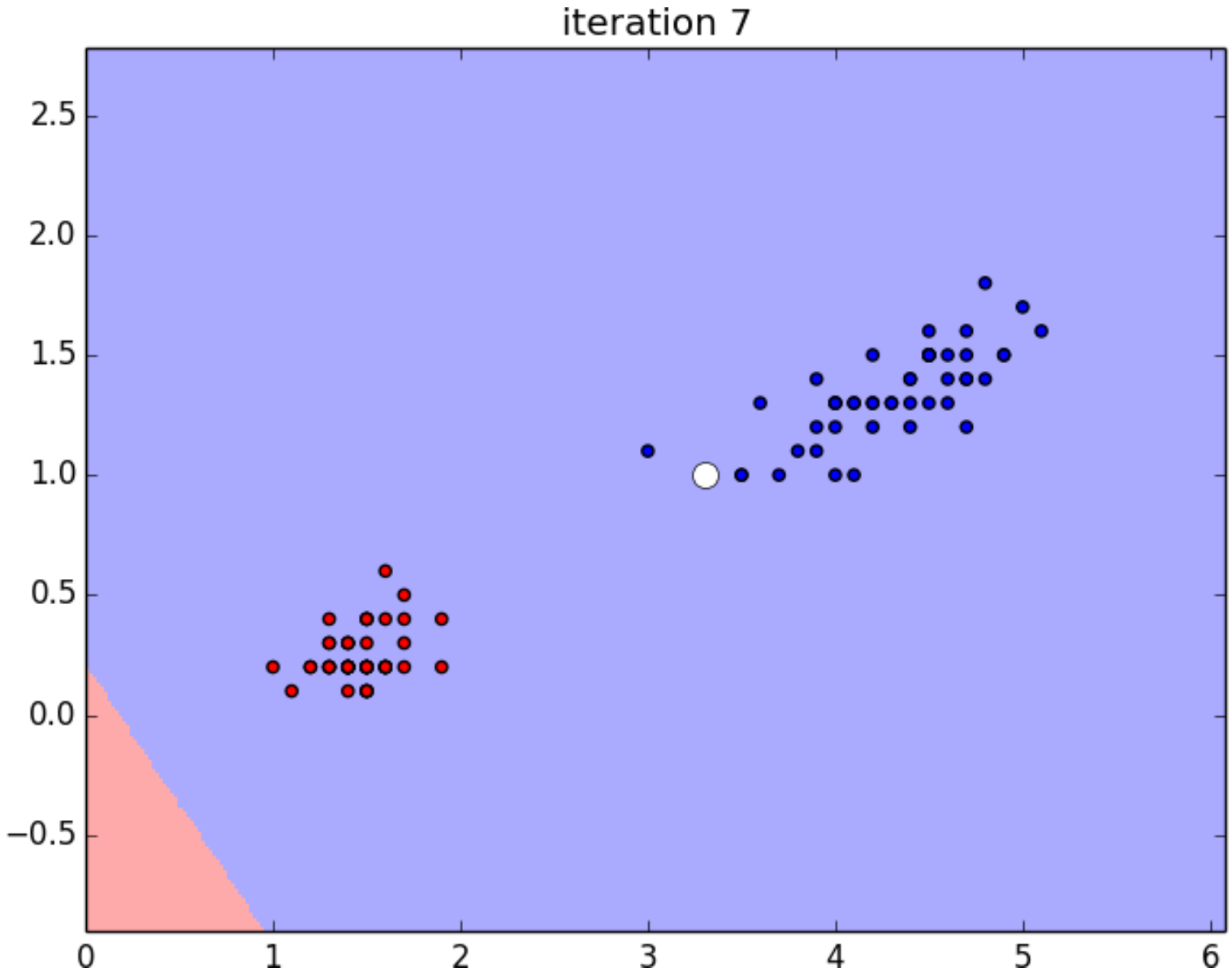
Perceptron Learning Rule



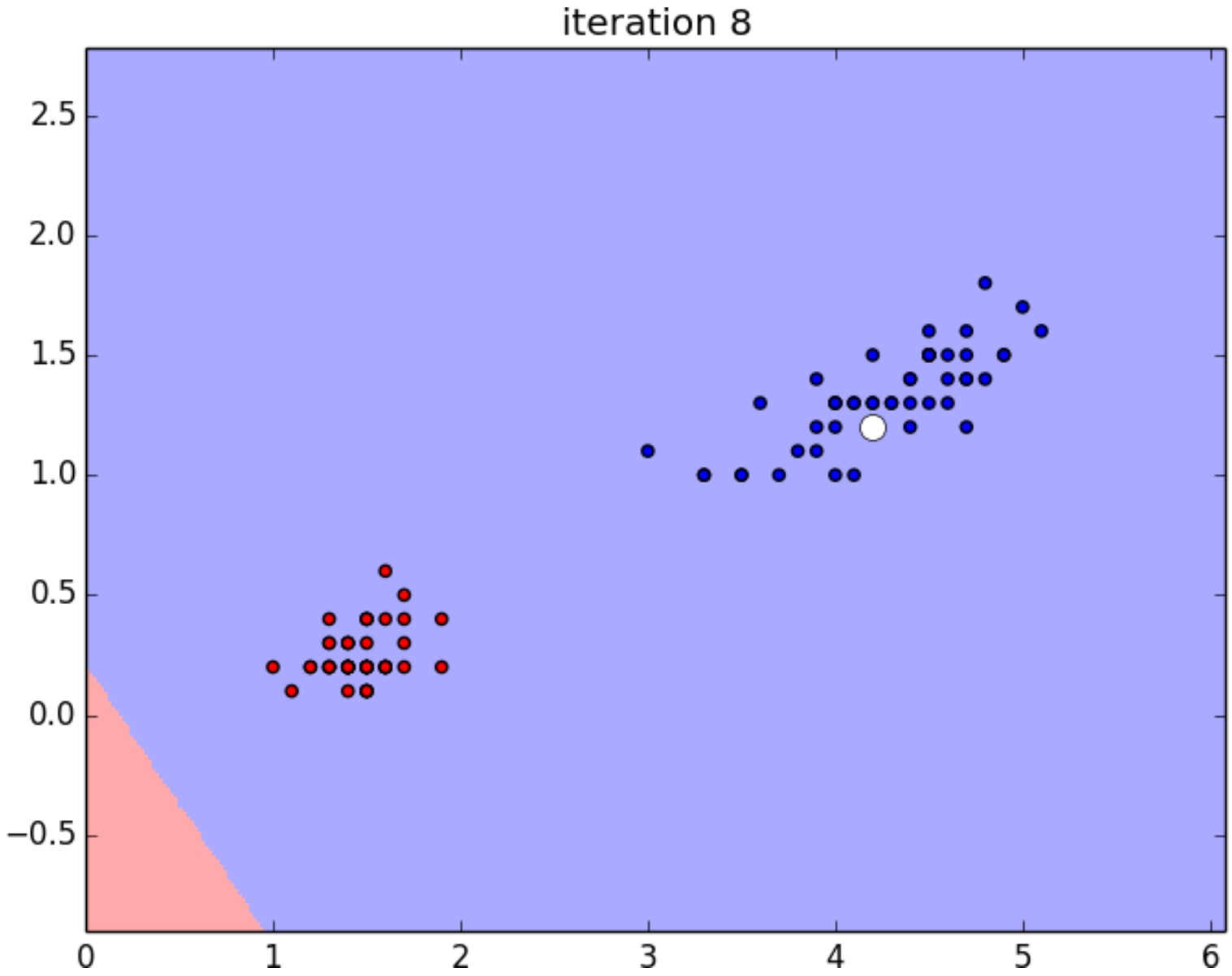
Perceptron Learning Rule



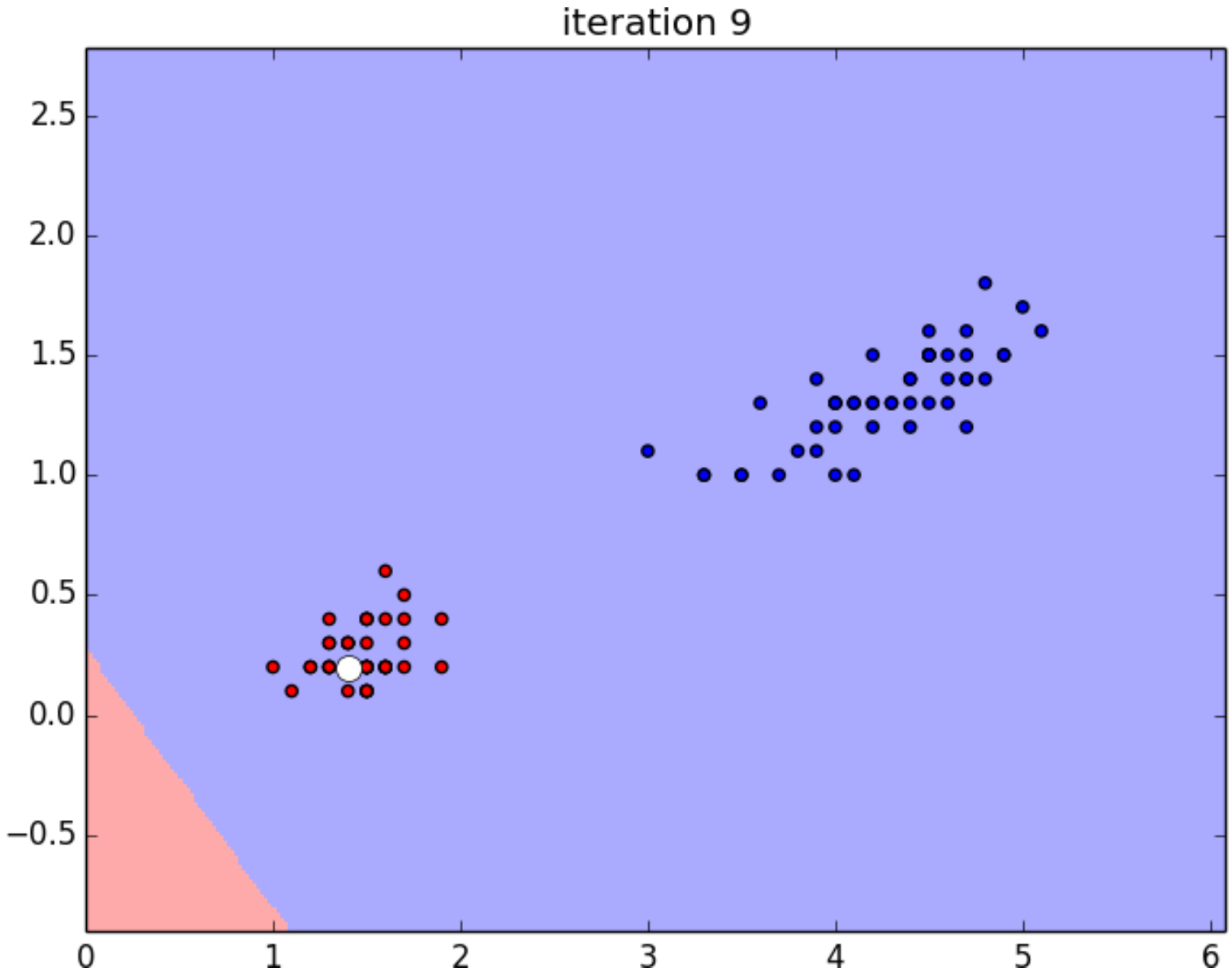
Perceptron Learning Rule



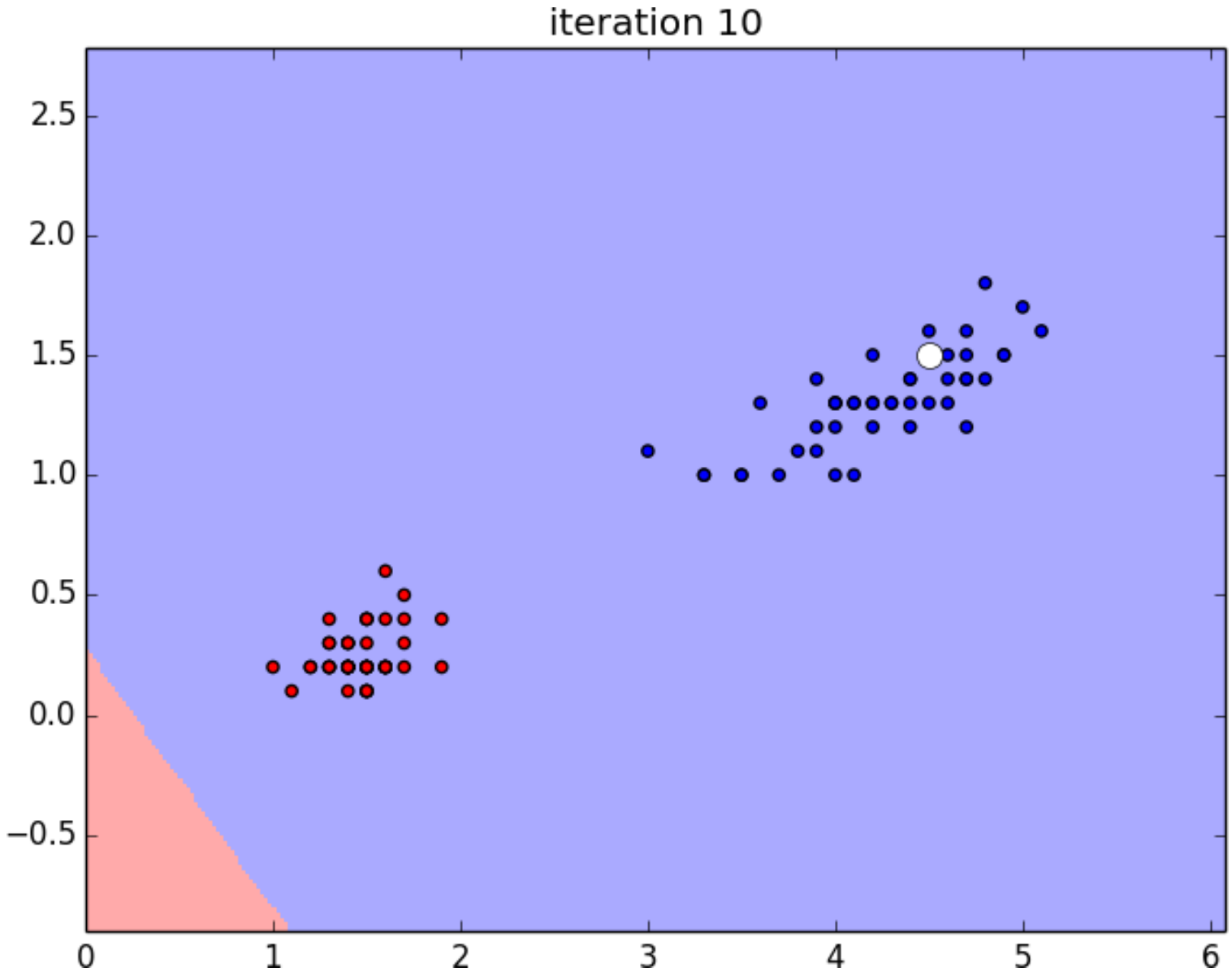
Perceptron Learning Rule



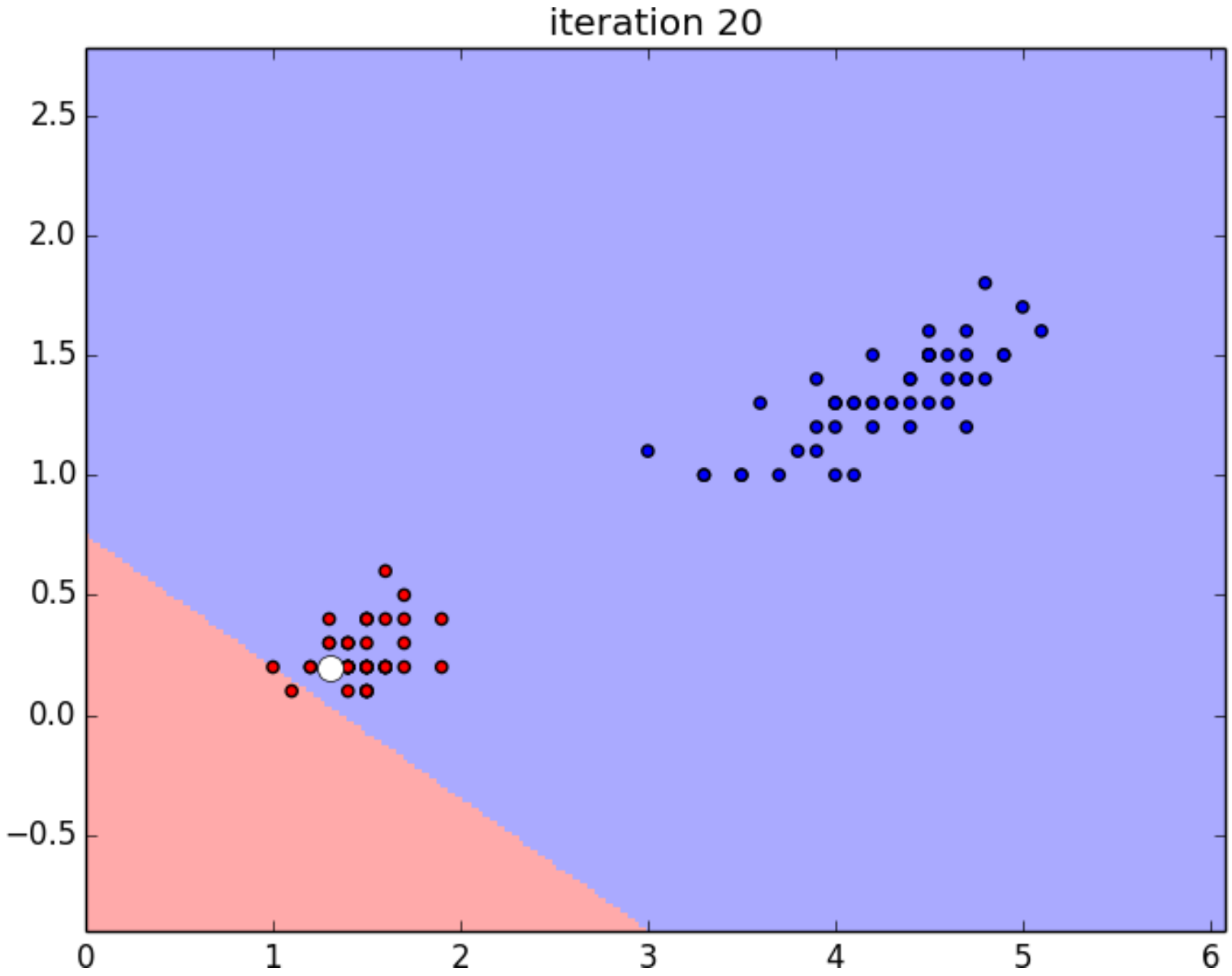
Perceptron Learning Rule



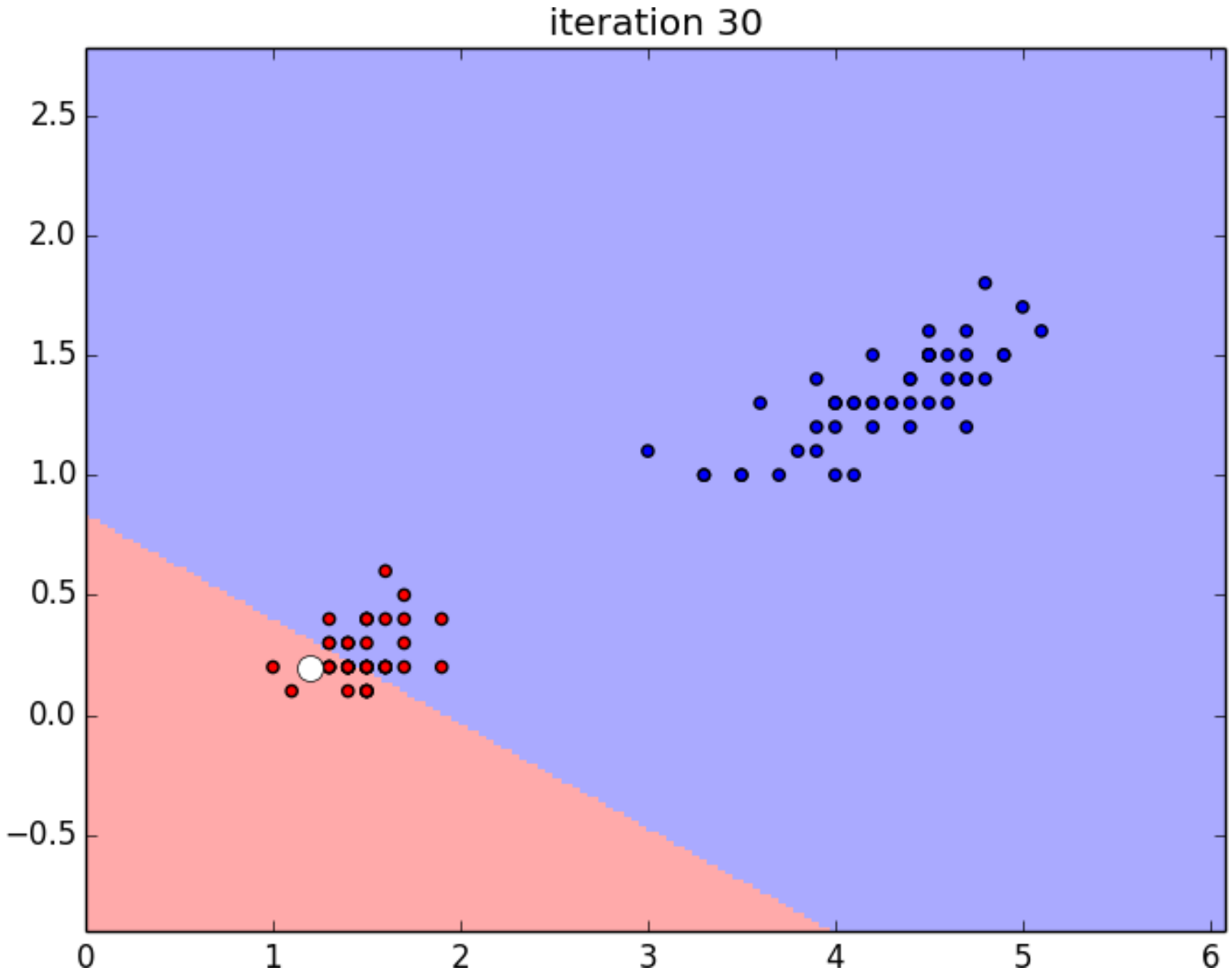
Perceptron Learning Rule



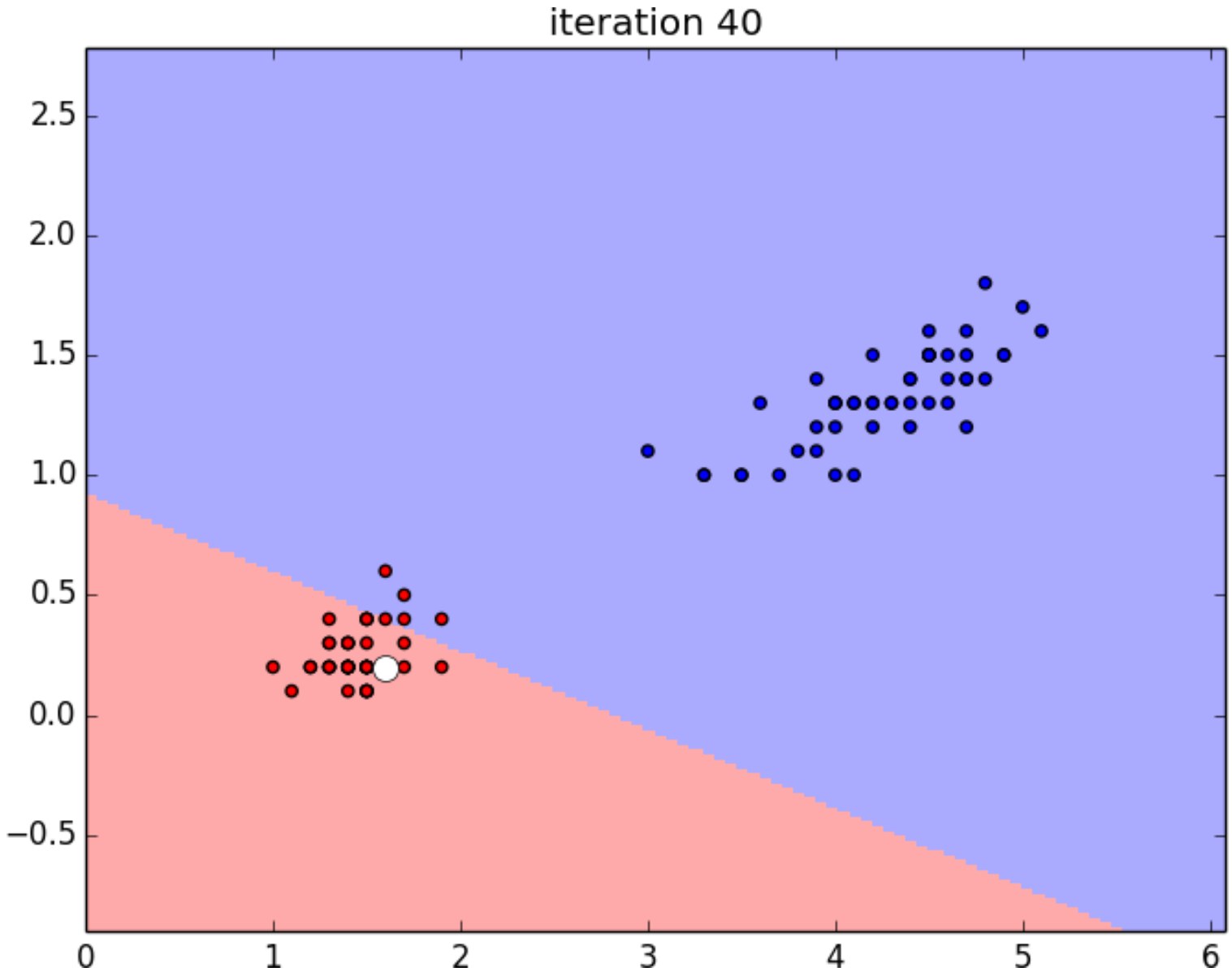
Perceptron Learning Rule



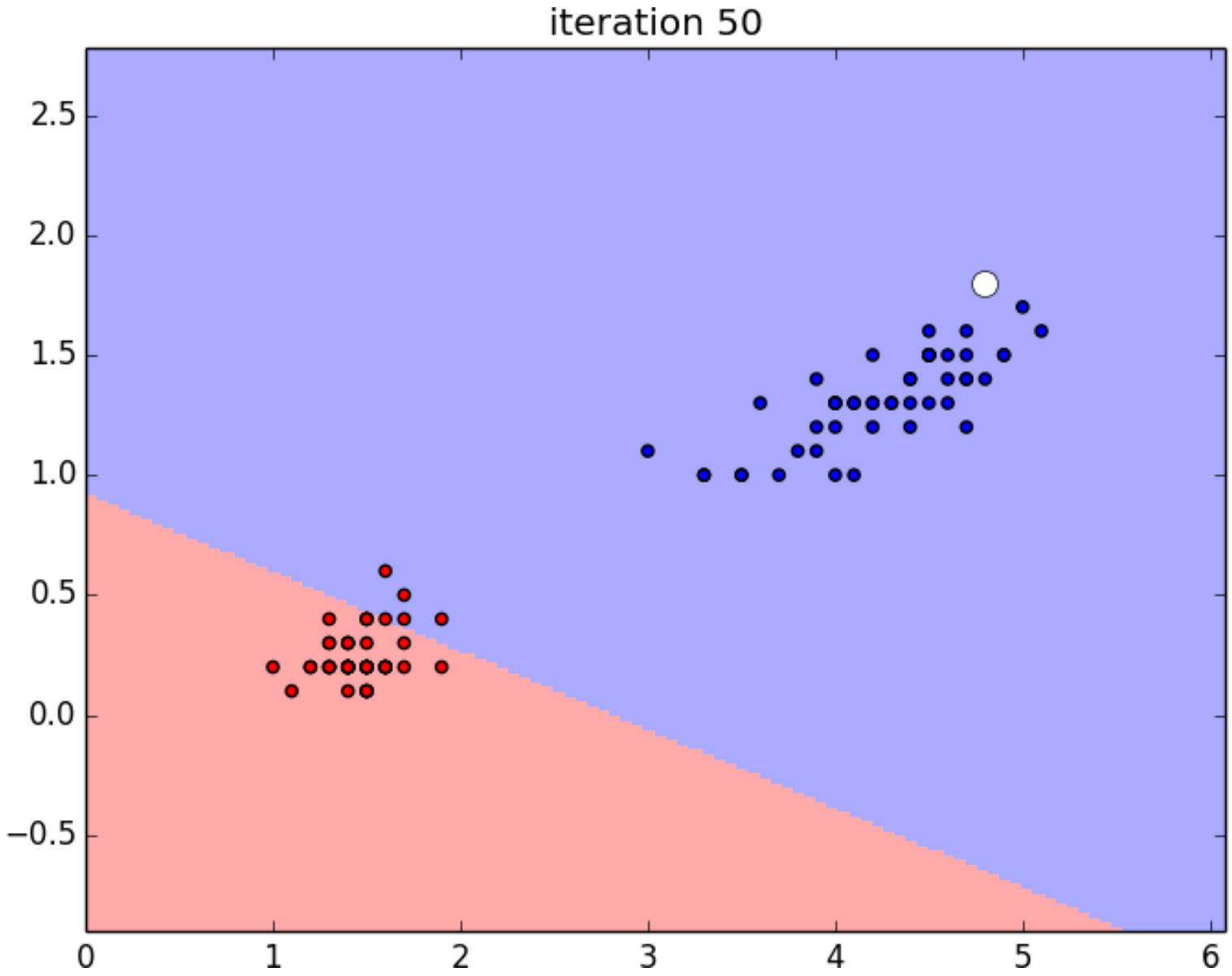
Perceptron Learning Rule



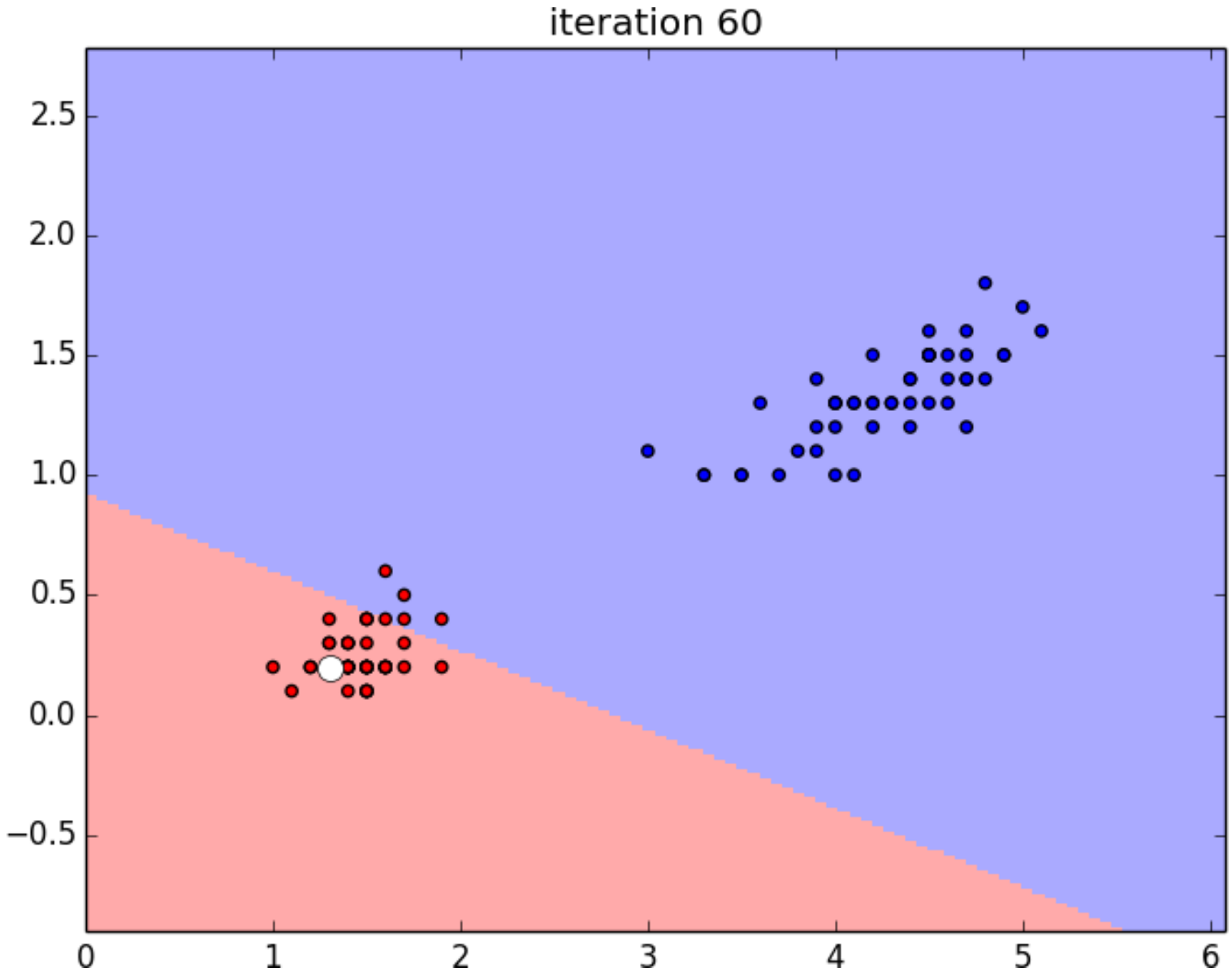
Perceptron Learning Rule



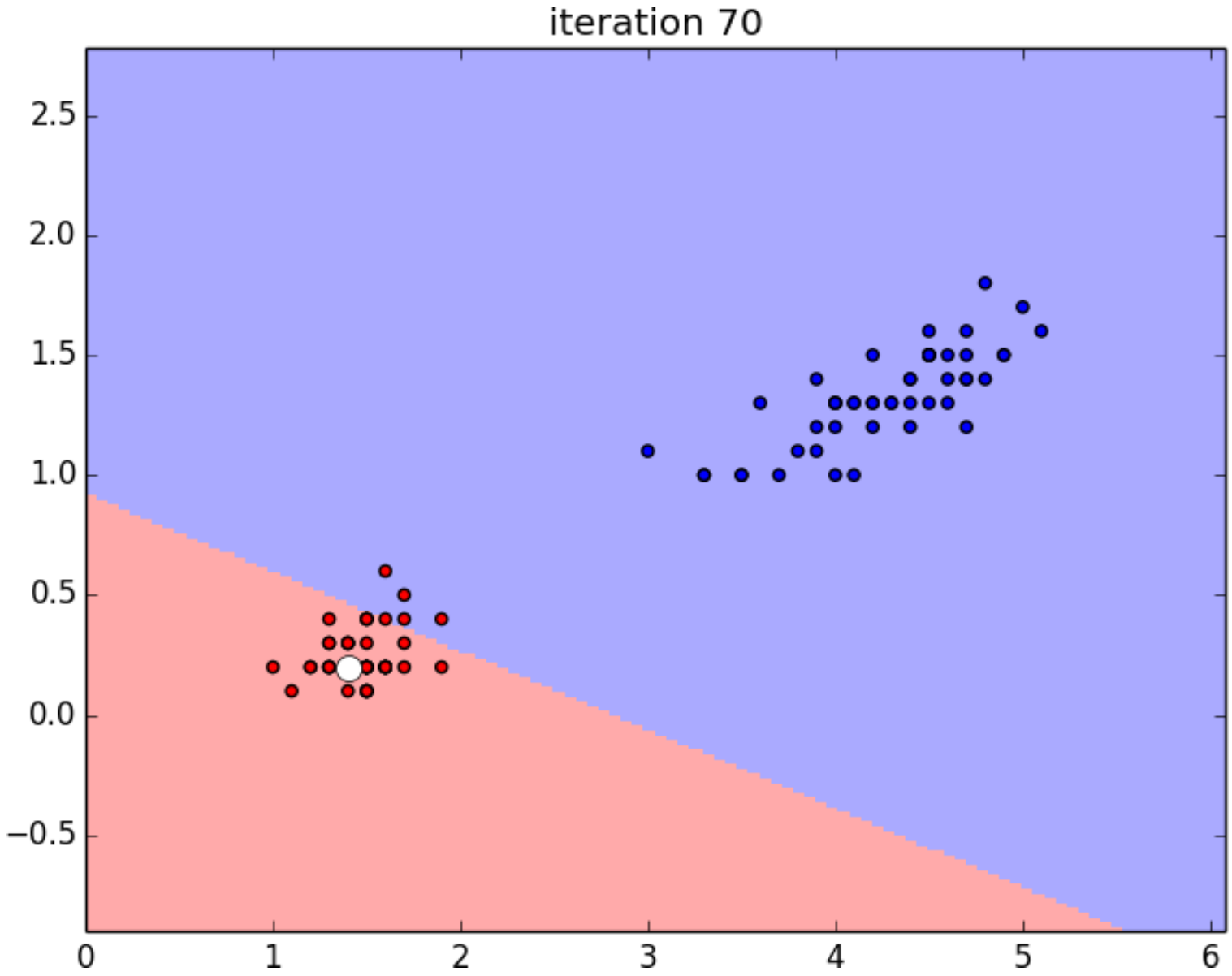
Perceptron Learning Rule



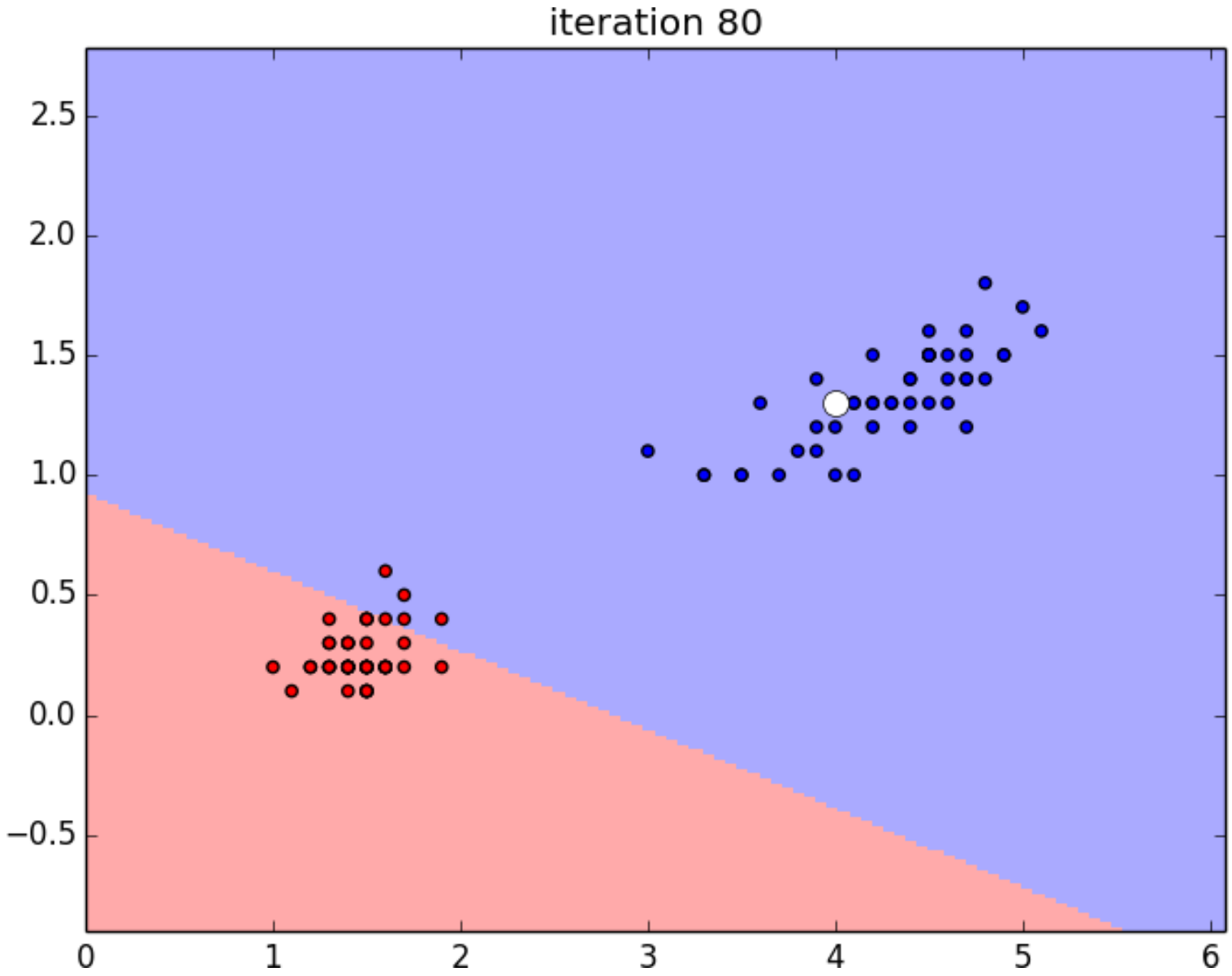
Perceptron Learning Rule



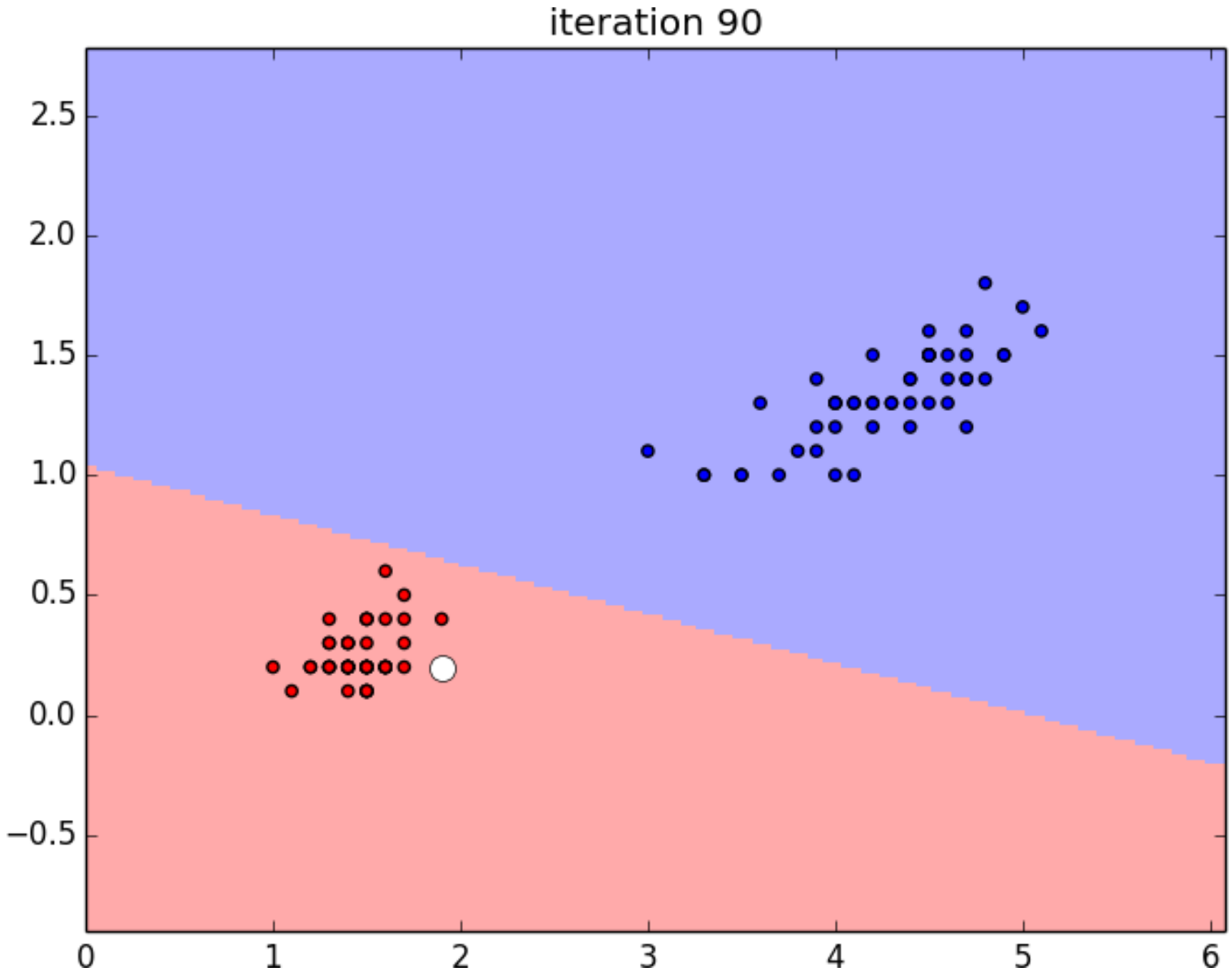
Perceptron Learning Rule



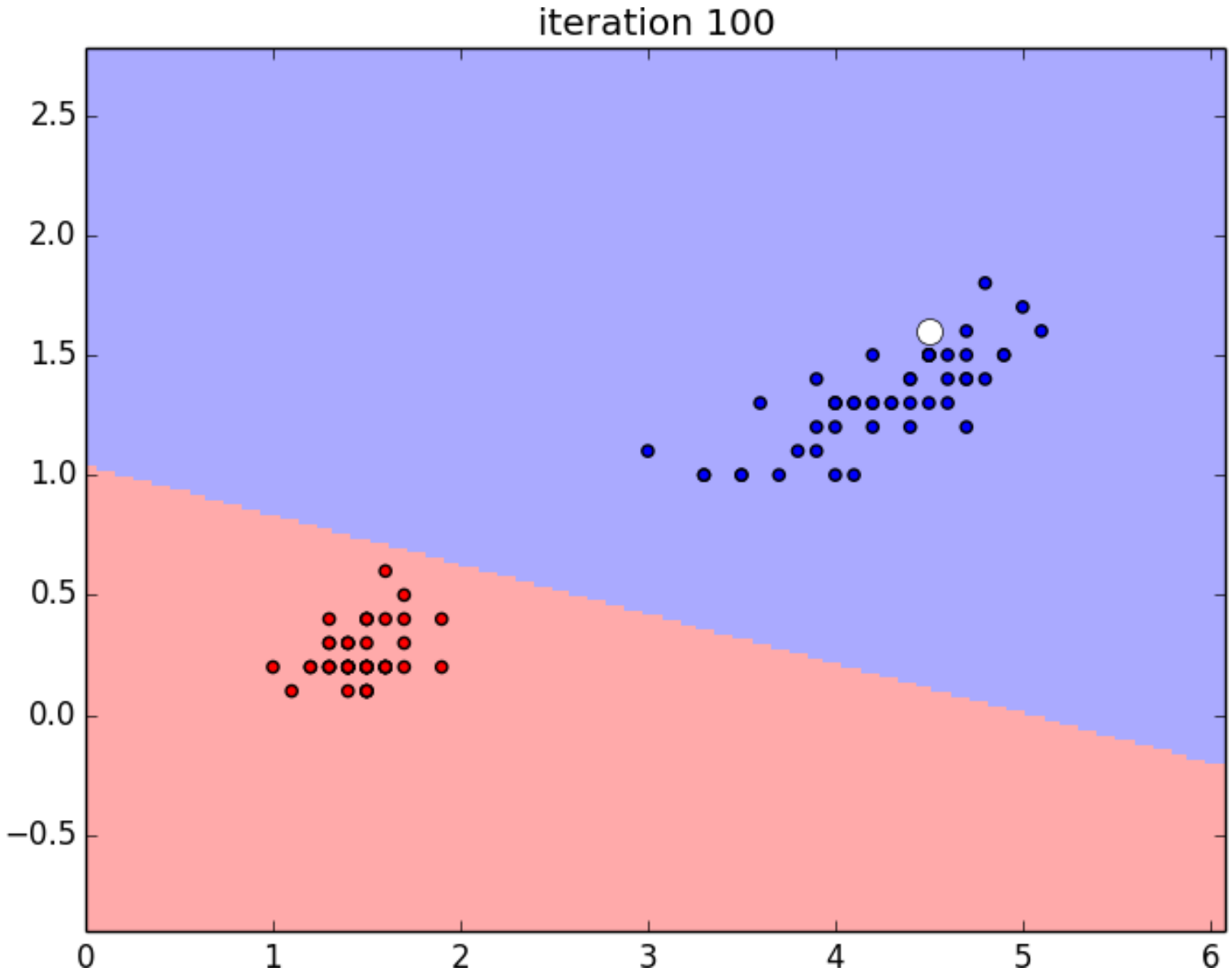
Perceptron Learning Rule



Perceptron Learning Rule



Perceptron Learning Rule



Perceptron Learning Rule

Perceptron training rule

online rule that iterates through training instances and each time does

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = +1$ is misclassified as $y = -1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + 2\alpha_t \|\mathbf{x}_i\|^2 \geq \mathbf{w}_t^T \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = -1$ is misclassified as $y = +1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i - 2\alpha_t \|\mathbf{x}_i\|^2 \leq \mathbf{w}_t^T \mathbf{x}_i$$

Convergence

if α is small enough, training converges for linearly separable problems

Perceptron Learning Rule

Perceptron training rule

online rule that iterates through training instances and each time does

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = +1$ is misclassified as $y = -1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + 2\alpha_t \|\mathbf{x}_i\|^2 \geq \mathbf{w}_t^T \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = -1$ is misclassified as $y = +1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i - 2\alpha_t \|\mathbf{x}_i\|^2 \leq \mathbf{w}_t^T \mathbf{x}_i$$

Convergence

if α is small enough, training converges for linearly separable problems

Perceptron Learning Rule

Perceptron training rule

online rule that iterates through training instances and each time does

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = +1$ is misclassified as $y = -1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + 2\alpha_t \|\mathbf{x}_i\|^2 \geq \mathbf{w}_t^T \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = -1$ is misclassified as $y = +1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i - 2\alpha_t \|\mathbf{x}_i\|^2 \leq \mathbf{w}_t^T \mathbf{x}_i$$

Convergence

if α is small enough, training converges for linearly separable problems

Perceptron Learning Rule

Perceptron training rule

online rule that iterates through training instances and each time does

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i$$

if $\mathbf{x}_i$ with $t_i = +1$ is misclassified as $y = -1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + 2\alpha_t \|\mathbf{x}_i\|^2 \geq \mathbf{w}_t^T \mathbf{x}_i$$

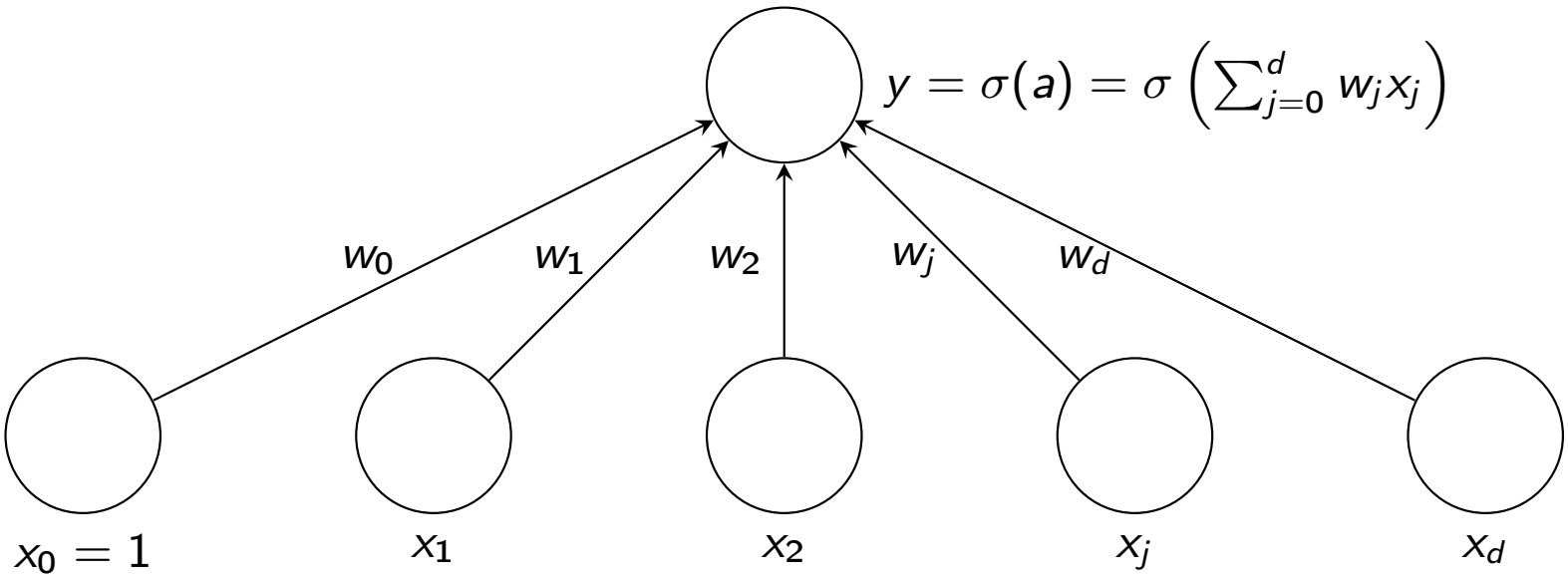
if $\mathbf{x}_i$ with $t_i = -1$ is misclassified as $y = +1$, weights are updated s.t.

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + \alpha_t \left(t_i - y_i\right) \mathbf{x}_i^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i - 2\alpha_t \|\mathbf{x}_i\|^2 \leq \mathbf{w}_t^T \mathbf{x}_i$$

Convergence

if α is small enough, training converges for linearly separable problems

Delta Rule



Perceptron for non-linear regression

non-linear regression with continuous differentiable activation function σ

Delta Rule (batch version)

Input: dataset $\mathcal{D} = \{(\mathbf{x}_i, t_i)\}$
Output: optimal weights for (non-)linear regression (w.r.t. MSE)

add $x_0 = 1$ in front of each vector $\mathbf{x}$ (trick to not care about bias)
initialise perceptron weights $\mathbf{w}_t = (w_0, w_1 \dots w_d)$ randomly

while MSE still changes significantly (or any convergence criterion) **do**
  if activation function is linear **then**
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - y_i)\, \mathbf{x}_i$
  else if activation function is sigmoid **then**
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - y_i)\, y_i\, (1 - y_i)\, \mathbf{x}_i$
  end if
end while

return $\mathbf{w}_t$

Delta Rule (batch version)

Impossible to exactly reproduce the target values

solution: minimise the mean square error (actually halved, for convenience)

$$E = \frac{1}{2\,n} \sum_{i=1}^{n} (t_i - y_i)^2 = \frac{1}{2\,n} \sum_{i=1}^{n} (t_i - \sigma(a_i))^2$$

optimality condition is $\nabla E = 0$, i.e. for each connection (= "synapse")

$$\frac{\delta E}{\delta w_j} = -\frac{1}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i)) \frac{\delta \sigma(a_i)}{\delta w_j} = -\frac{1}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i))\, \sigma'(a_i) x_j = 0$$

issue: impossible to isolate $w_j \Rightarrow$ no "non-linear pseudo-inverse" method

Delta Rule (batch version)

Impossible to exactly reproduce the target values

solution: minimise the mean square error (actually halved, for convenience)

$$E = \frac{1}{2\,n} \sum_{i=1}^{n} (t_i - y_i)^2 = \frac{1}{2\,n} \sum_{i=1}^{n} (t_i - \sigma(a_i))^2$$

optimality condition is $\nabla E = 0$, i.e. for each connection (= "synapse")

$$\frac{\delta E}{\delta w_j} = -\frac{1}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i)) \frac{\delta \sigma(a_i)}{\delta w_j} = -\frac{1}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i))\, \sigma'(a_i) x_j = 0$$

issue: impossible to isolate $w_j \Rightarrow$ no "non-linear pseudo-inverse" method

Delta Rule (batch version)

Solution = gradient descent

idea: iteratively update synaptic weights in the direction of lower MSE

$$w_j \leftarrow w_j - \alpha \frac{\delta E}{\delta w_j} = w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i))\, \sigma'(a_i) x_j$$

for linear activation function $\sigma(a) = a$, the update rule becomes

$$w_j \leftarrow w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - y_i)\, x_j$$

limitation: all instances must be seen before weights are updated

Delta Rule (batch version)

Solution = gradient descent

idea: iteratively update synaptic weights in the direction of lower MSE

$$w_j \leftarrow w_j - \alpha \frac{\delta E}{\delta w_j} = w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i))\, \sigma'(a_i) x_j$$

for linear activation function $\sigma(a) = a$, the update rule becomes

$$w_j \leftarrow w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - y_i)\, x_j$$

limitation: all instances must be seen before weights are updated

Delta Rule (batch version)

Solution = gradient descent

idea: iteratively update synaptic weights in the direction of lower MSE

$$w_j \leftarrow w_j - \alpha \frac{\delta E}{\delta w_j} = w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - \sigma(a_i))\, \sigma'(a_i) x_j$$

for sigmoid activation function $\sigma(a) = 1/(1 + \exp(-a))$, the update rule is

$$w_j \leftarrow w_j + \frac{\alpha}{n} \sum_{i=1}^{n} (t_i - y_i)\, y_i\, (1 - y_i)\, x_j$$

limitation: all instances must be seen before weights are updated

Delta Rule (online version)

Solution = stochastic gradient descent

stochastic approximation to gradient descent: iteratively choose an instance x_i and update synaptic weights w.r.t. the error component $E_i = \frac{1}{2}(t_i - y_i)^2$

$$w_j \leftarrow w_j - \alpha \frac{\delta E_i}{\delta w_j} = w_j + \alpha (t_i - \sigma(a_i)) \sigma'(a_i) x_j$$

only $\mathbf{x}_i$ is used, but sequence of updates = approximation to batch update

for linear activation function $\sigma(a) = a$, the update rule becomes

$$w_j \leftarrow w_j + \alpha (t_i - y_i) x_j$$

remark: very close to the Perceptron learning rule, except that $\sigma \neq$ sign

Delta Rule (online version)

Solution = stochastic gradient descent

stochastic approximation to gradient descent: iteratively choose an instance x_i and update synaptic weights w.r.t. the error component $E_i = \frac{1}{2}(t_i - y_i)^2$

$$w_j \leftarrow w_j - \alpha \frac{\delta E_i}{\delta w_j} = w_j + \alpha (t_i - \sigma(a_i)) \sigma'(a_i) x_j$$

only $\mathbf{x}_i$ is used, but sequence of updates = approximation to batch update

for linear activation function $\sigma(a) = a$, the update rule becomes

$$w_j \leftarrow w_j + \alpha (t_i - y_i) x_j$$

remark: very close to the Perceptron learning rule, except that $\sigma \neq$ sign

Delta Rule (online version)

Solution = stochastic gradient descent

stochastic approximation to gradient descent: iteratively choose an instance x_i and update synaptic weights w.r.t. the error component $E_i = \frac{1}{2}(t_i - y_i)^2$

$$w_j \leftarrow w_j - \alpha \frac{\delta E_i}{\delta w_j} = w_j + \alpha (t_i - \sigma(a_i)) \sigma'(a_i) x_j$$

only $\mathbf{x}_i$ is used, but sequence of updates = approximation to batch update

for sigmoid activation function $\sigma(a) = 1/(1 + \exp(-a))$, the update rule is

$$w_j \leftarrow w_j + \alpha (t_i - y_i) y_i (1 - y_i) x_j$$

remark: very close to the Perceptron learning rule, except that $\sigma \neq$ sign

Delta Rule (online version)

Input: dataset $\mathcal{D} = \{(\mathbf{x}_i, t_i)\}$
Output: optimal weights for (non-)linear regression (w.r.t. MSE)

add $x_0 = 1$ in front of each vector $\mathbf{x}$ (trick to not care about bias)
initialise perceptron weights $\mathbf{w}_t = (w_0, w_1 \dots w_d)$ randomly

while MSE still changes significantly (or any convergence criterion) **do**
  for each instance $\mathbf{x}_i$ with target t_i in dataset **do**
    if activation function is linear **then**
      $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + (t_i - y_i)\,\mathbf{x}_i$
    else if activation function is sigmoid **then**
      $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + (t_i - y_i)\,y_i\,(1 - y_i)\,\mathbf{x}_i$
    end if
  end for
end while

return $\mathbf{w}_t$

Gradient Descent vs. Stochastic Gradient Descent

Gradient descent

- each iteration accurately estimates the direction for lower MSE
- the whole dataset must be seen to complete one iteration
- a few tens/hundred/thousands iterations are usually necessary

Stochastic gradient descent

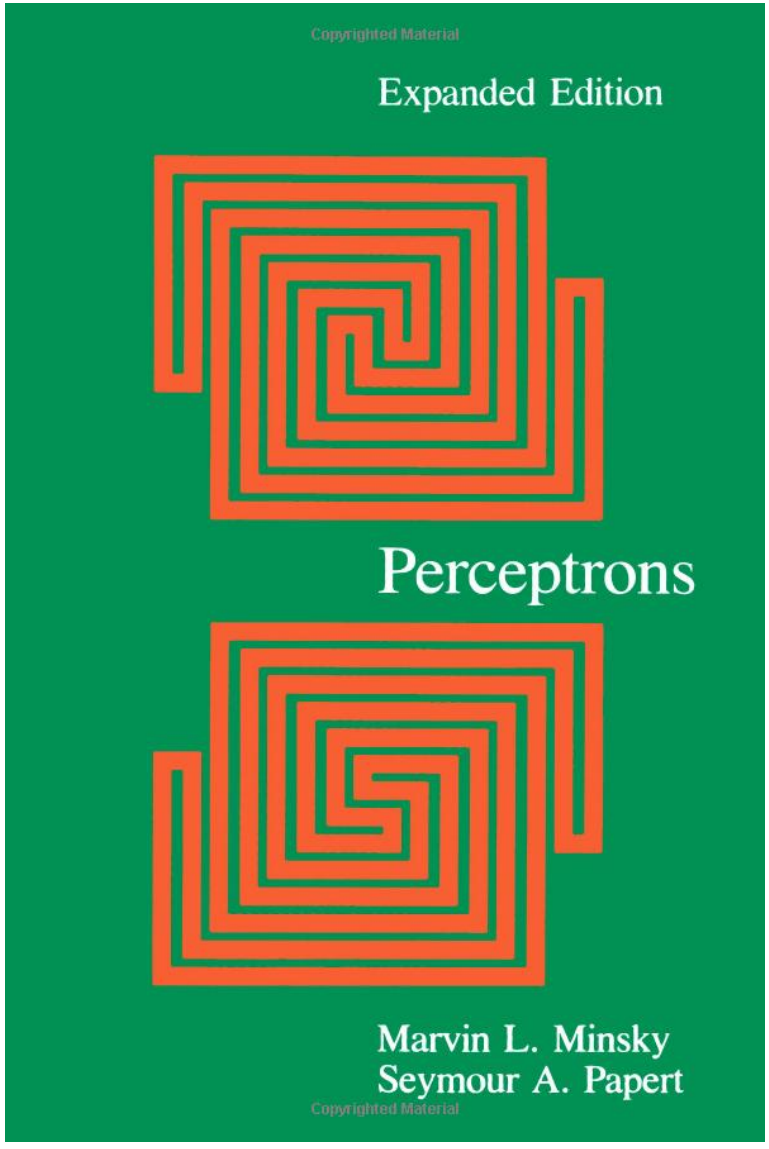
- each iteration goes in a different direction
- only one instance is necessary at each iteration
- the number of iterations can be very large, but does not depend on n

What algorithm should I use ?

SGD is often more suitable for large datasets ("big data") and data streams

The First AI Winter

Paper & Minsky Book on Perceptrons



Perceptrons
Expanded Edition
by Marvin L. Minsky and Seymour A. Papert

Perceptrons—the first systematic study of parallelism in computation—has remained a classical work on threshold automata networks for nearly two decades. It marked a historical turn in artificial intelligence and is required reading for anyone who wants to understand the connectionist revival that is going on today.

Artificial-intelligence research, which for a time concentrated on the programming of von Neumann computers, is swinging back to the idea that intelligence might emerge from the activity of networks of neuron-like entities. Minsky and Papert's book was the first example of a mathematical analysis carried far enough to show the exact limitations of a class of computing machines that could seriously be considered as models of the brain.

Now the new developments in mathematical tools, the recent interest of physicists in the theory of disordered matter, the new insights into and psychological models of how the brain works, and the evolution of fast computers that can simulate networks of automata have given *Perceptrons* new importance.

Minsky and Papert have added a chapter to their seminal study in which they discuss the current state of parallel computers, review developments since the appearance of the 1972 edition, and identify new research directions related to connectionism. The central theoretical challenge facing connectionism, they observe, is in reaching a deeper understanding of how "objects" or "agents" with individuality can emerge in a network. Progress in this area would link connectionism with what the authors have called "society theories of mind."

Marvin L. Minsky is Donner Professor of Science in MIT's Electrical Engineering and Computer Science Department, and Seymour A. Papert is Professor of Media Technology at MIT.

978-0-262-63111-2

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

Paper & Minsky Book on Perceptrons

Limitations of Single-Layer Perceptrons

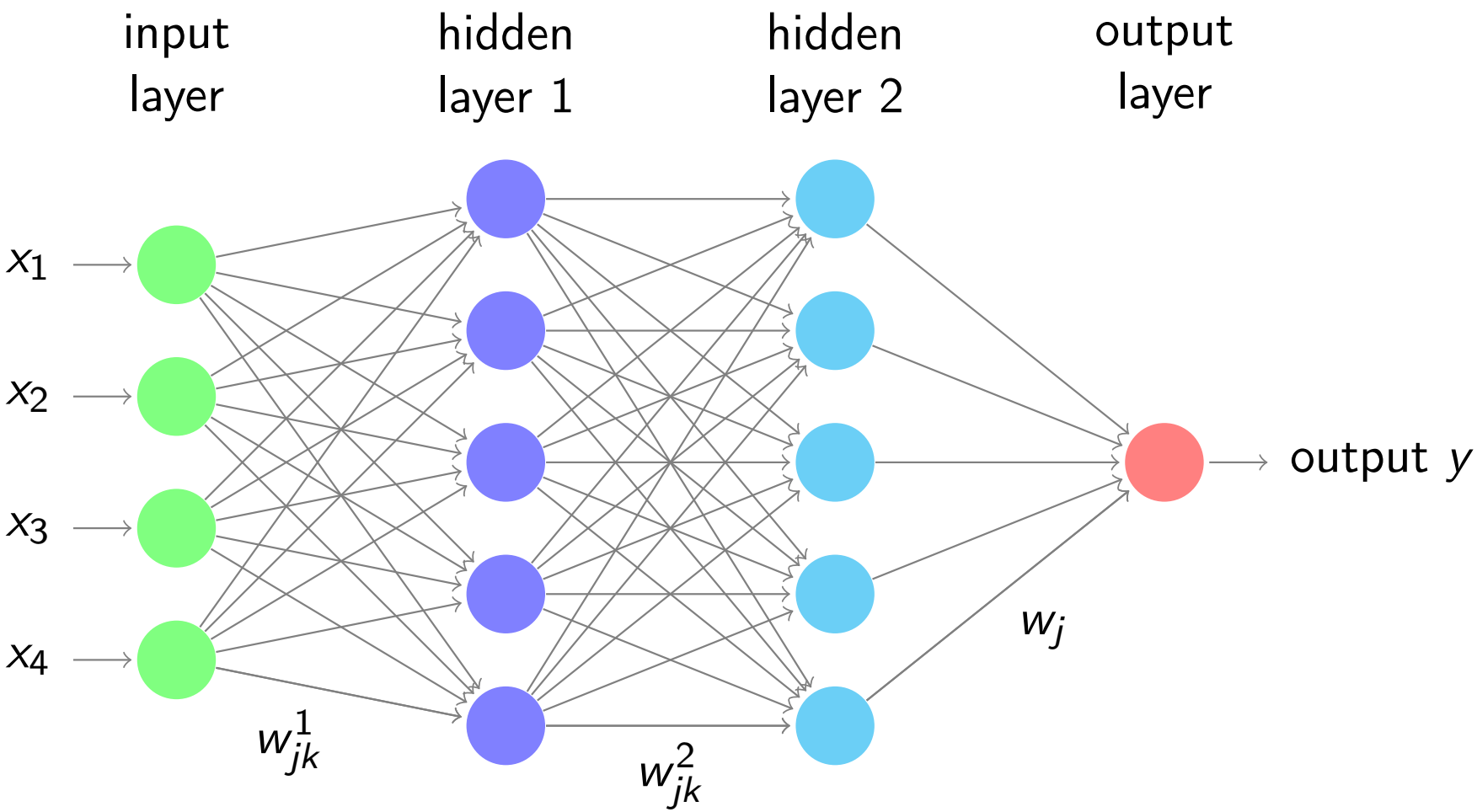
- single-neuron networks cannot solve XOR classification tasks
- some tasks require a hidden layer of fully connected neurons

The First AI Winter

Papert and Minsky triggered a lost of interest and funding for ANNs from 1969 to the middle '80s (similar issue in whole AI with the Lighthill report in 1973 and DARPA funding cuts), until the advent of back-propagation

Multi-Layer Neural Networks

Perceptrons with Hidden Layers



Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

Multi-Layer Perceptrons

Solution to the ANN crisis

hidden layers increase the expressiveness of perceptrons

- universal approximation = approximate arbitrarily complex functions
- efficient learning algorithms are available (back-propagation)
- the complexity of a MLP depends on the number of
  - hidden layers (each corresponds to a "level of representation")
  - hidden neurons in each hidden layer (complex task $\Rightarrow$ more neurons)

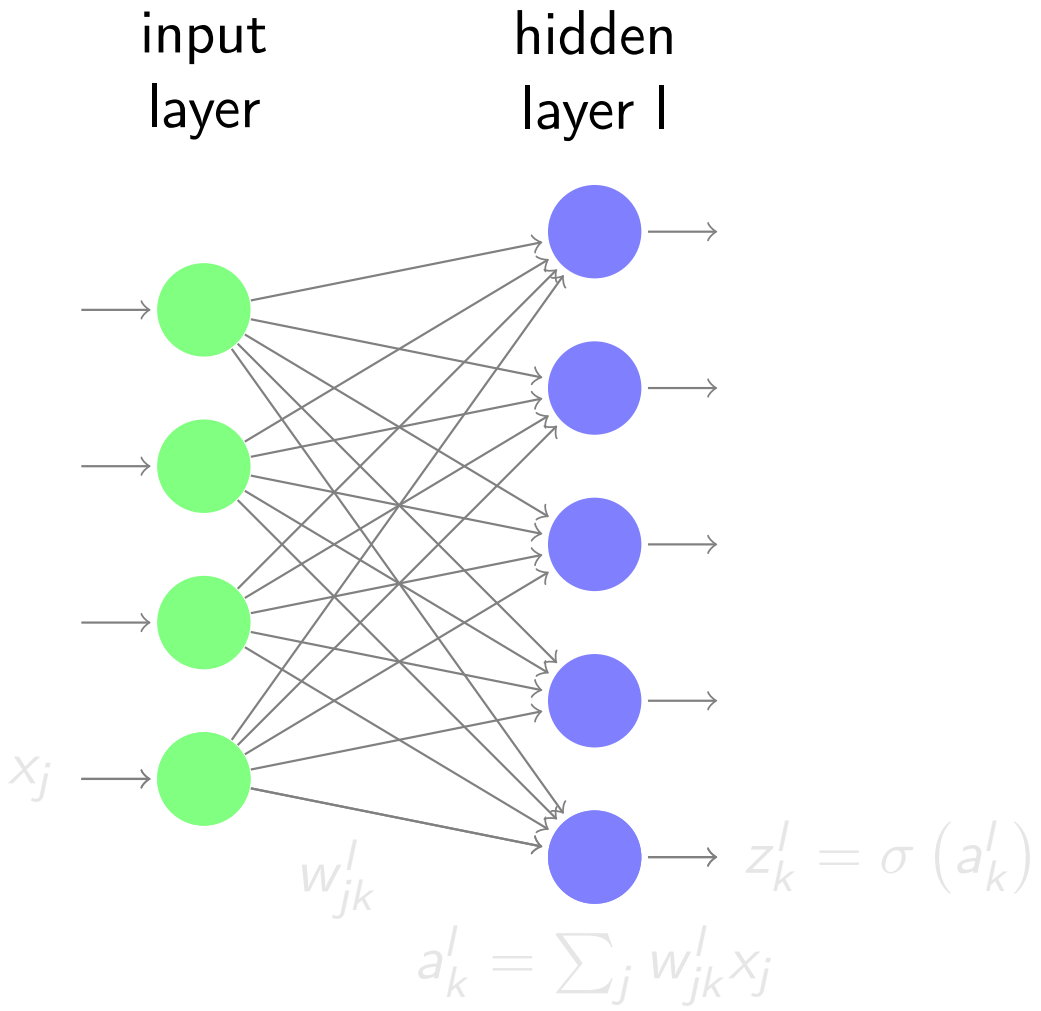
Universal approximation

a MLP with two-layer of neurons (i.e. one hidden layer) and a linear activation at the output is sufficient to approximate continuous functions

- in practice, using more than two layers may be necessary for images
- hot topic in machine learning $\Rightarrow$ deep learning (very deep MLPs)

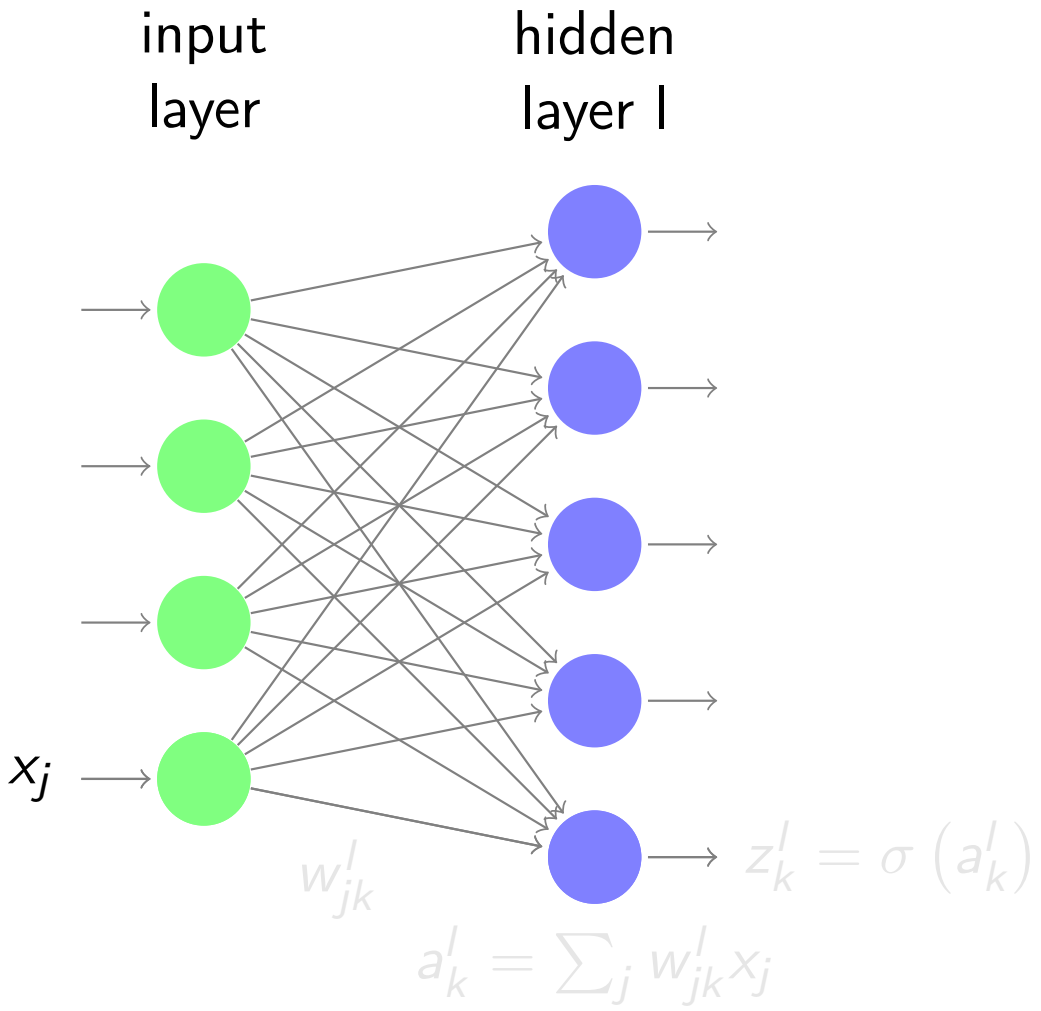
Error Back-Propagation

Activation Propagation for Prediction: Input Layer



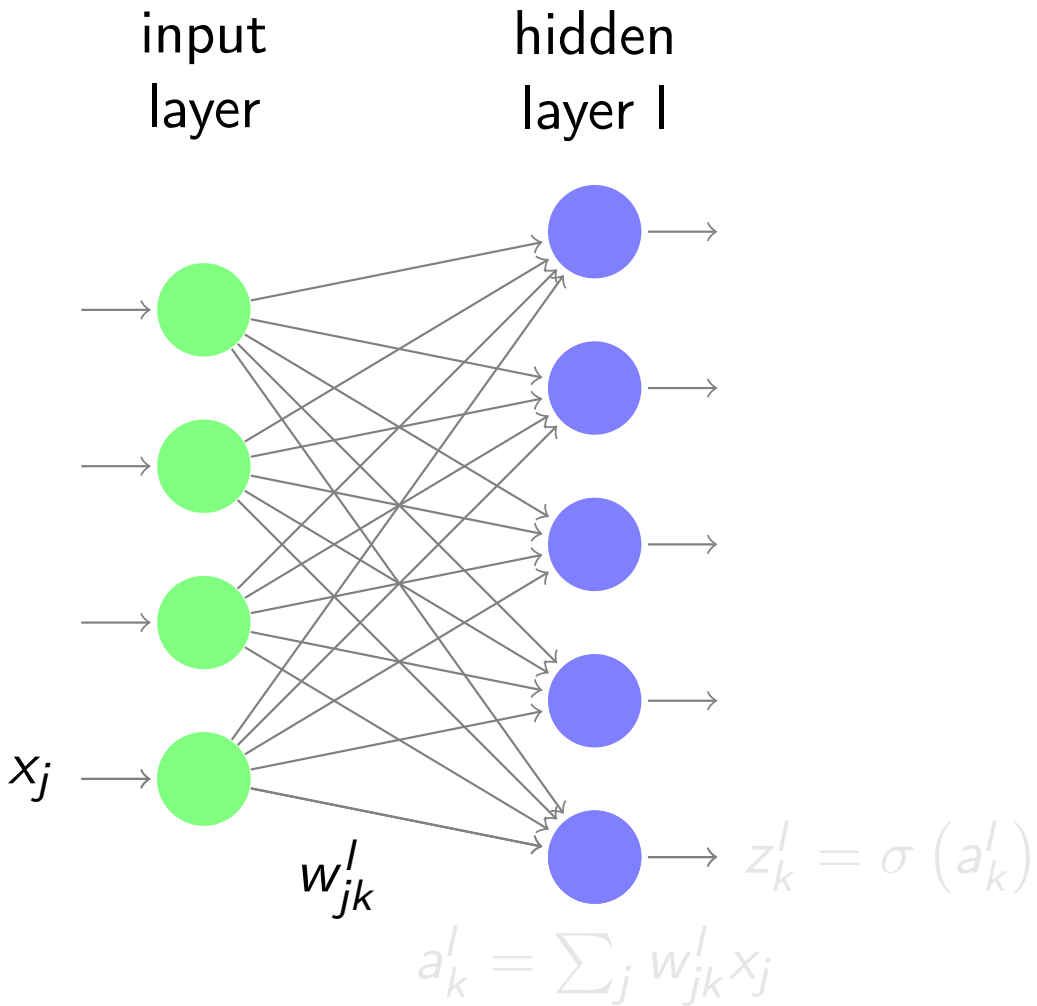
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Input Layer



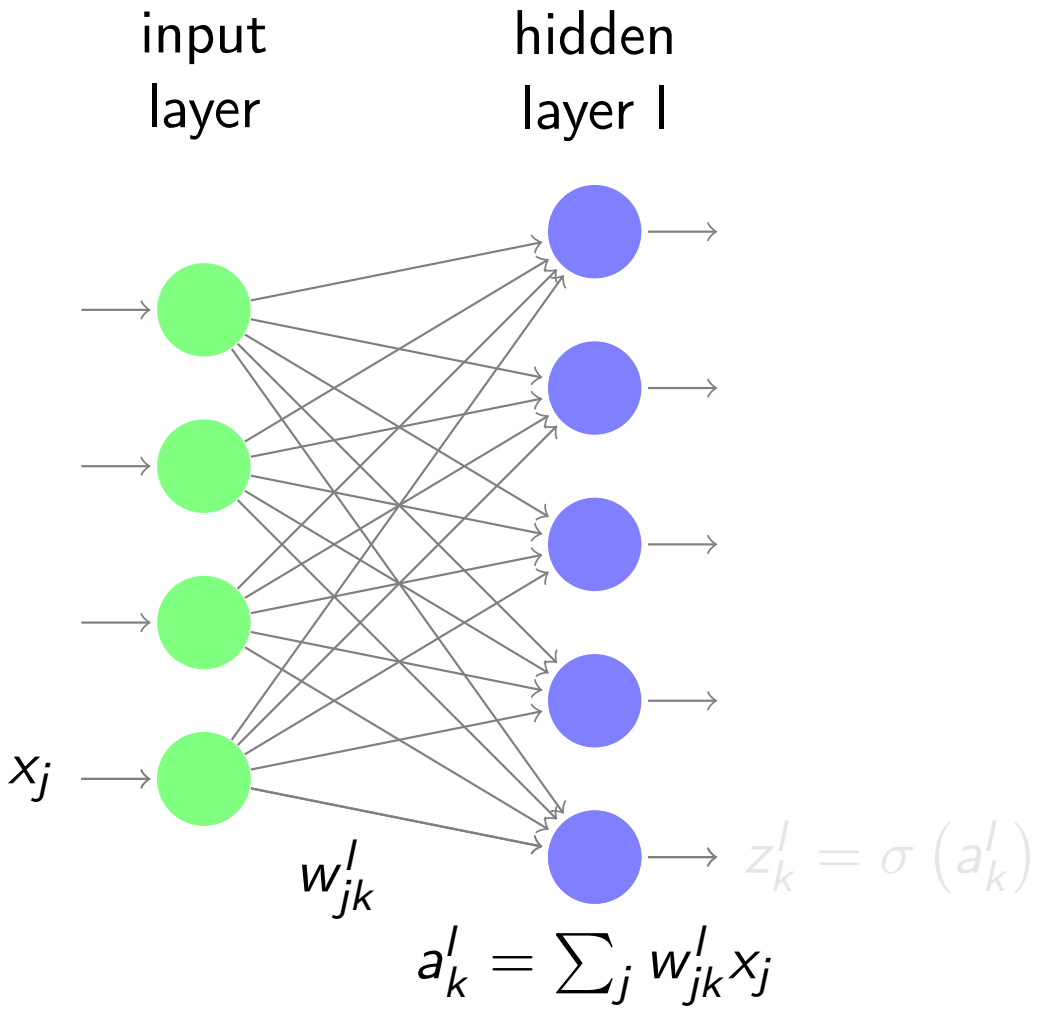
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Input Layer



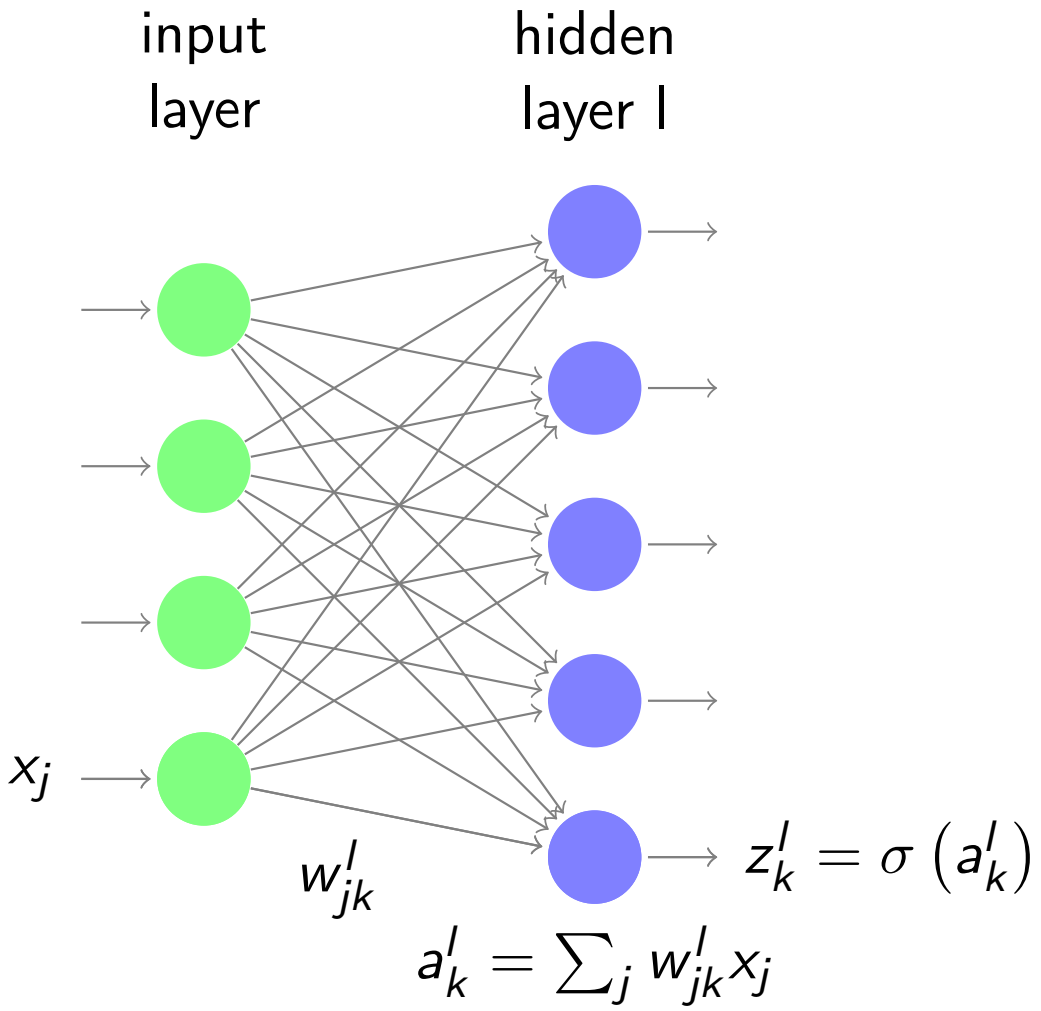
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Input Layer



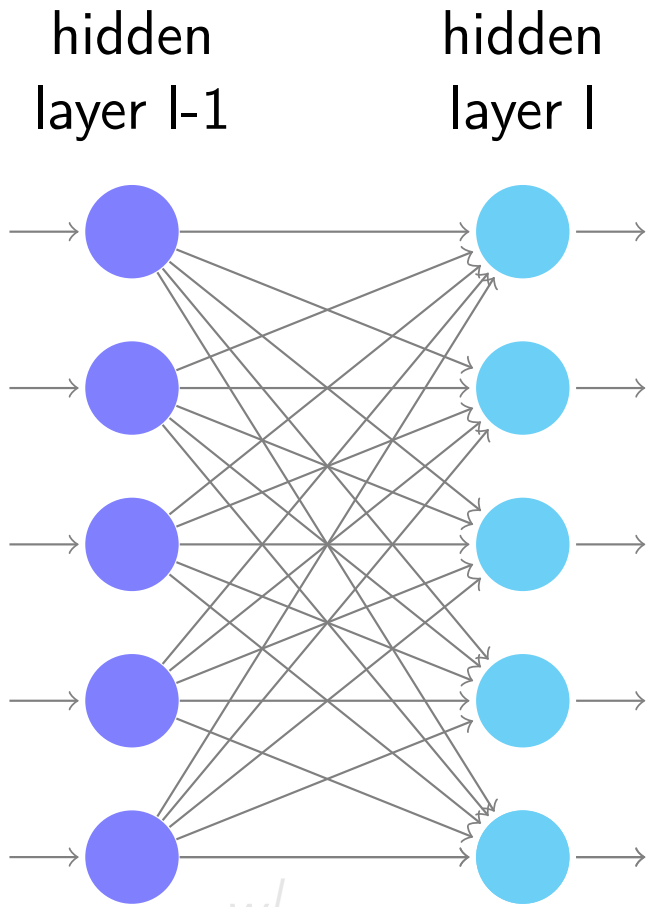
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Input Layer



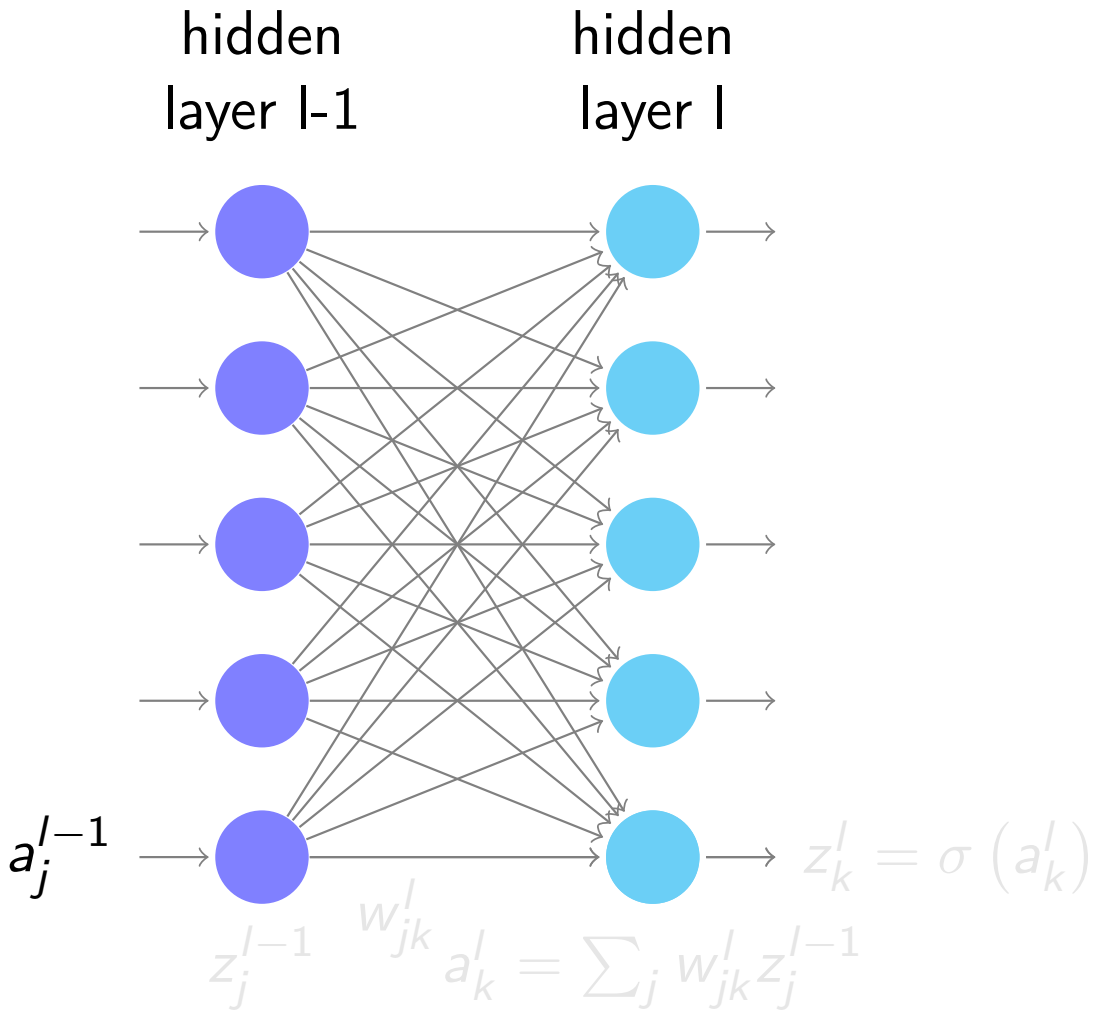
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



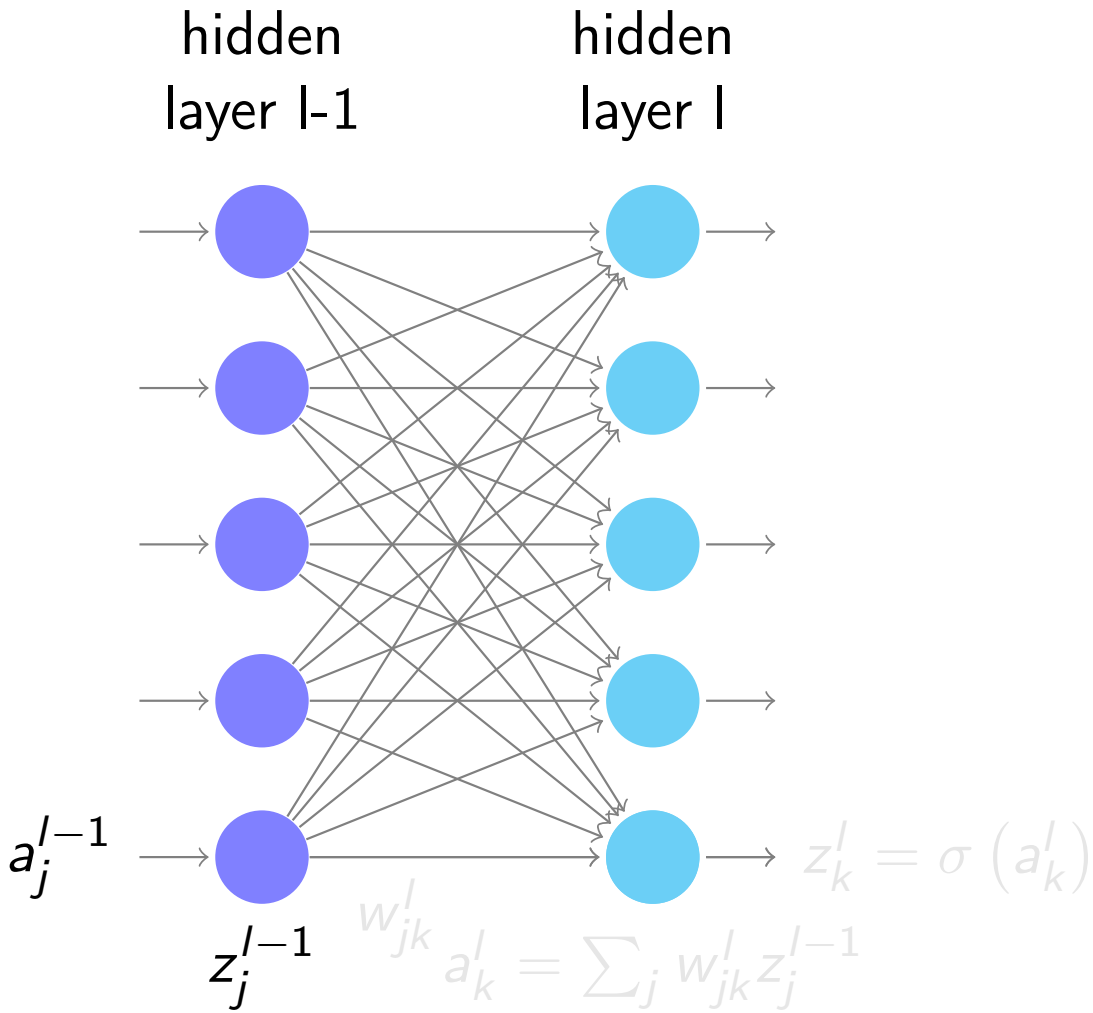
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



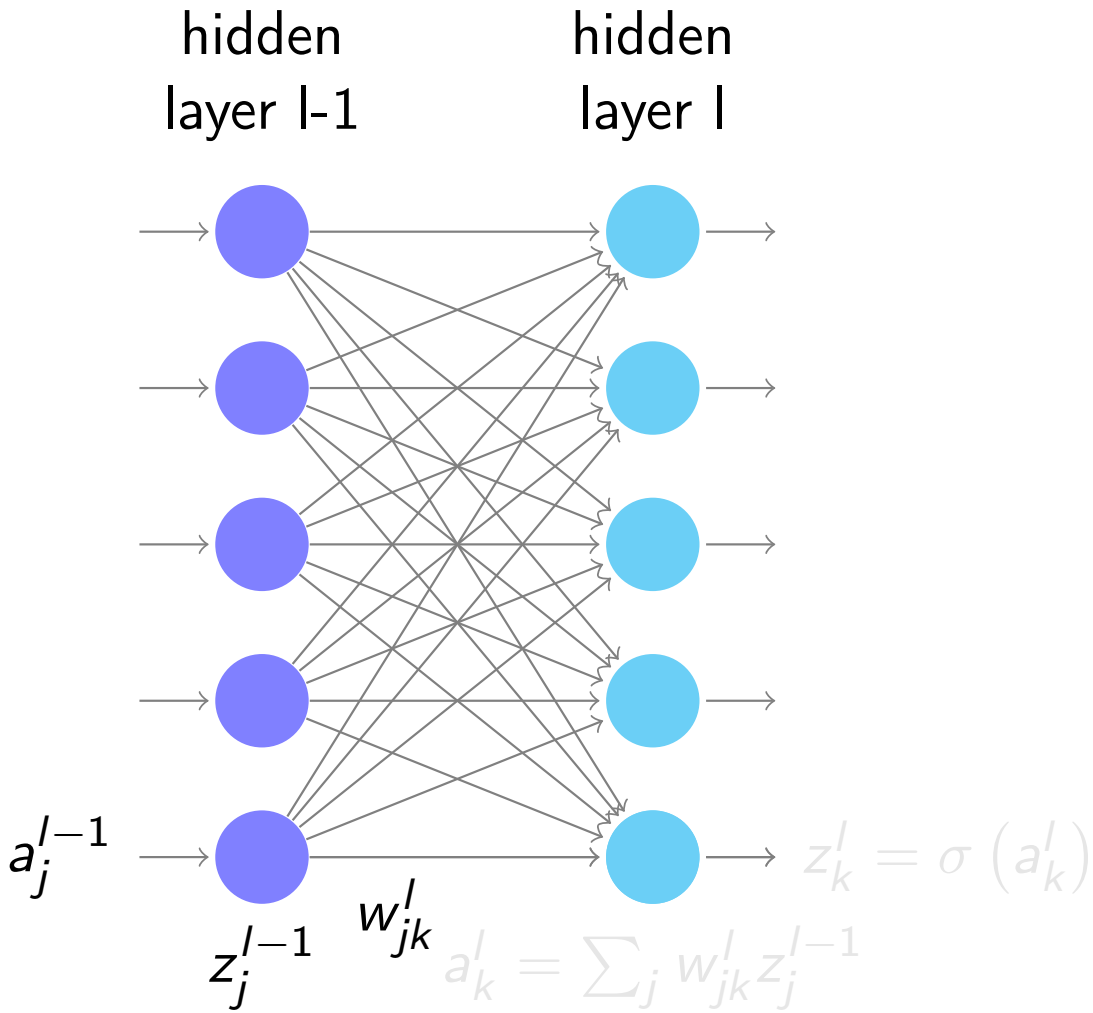
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



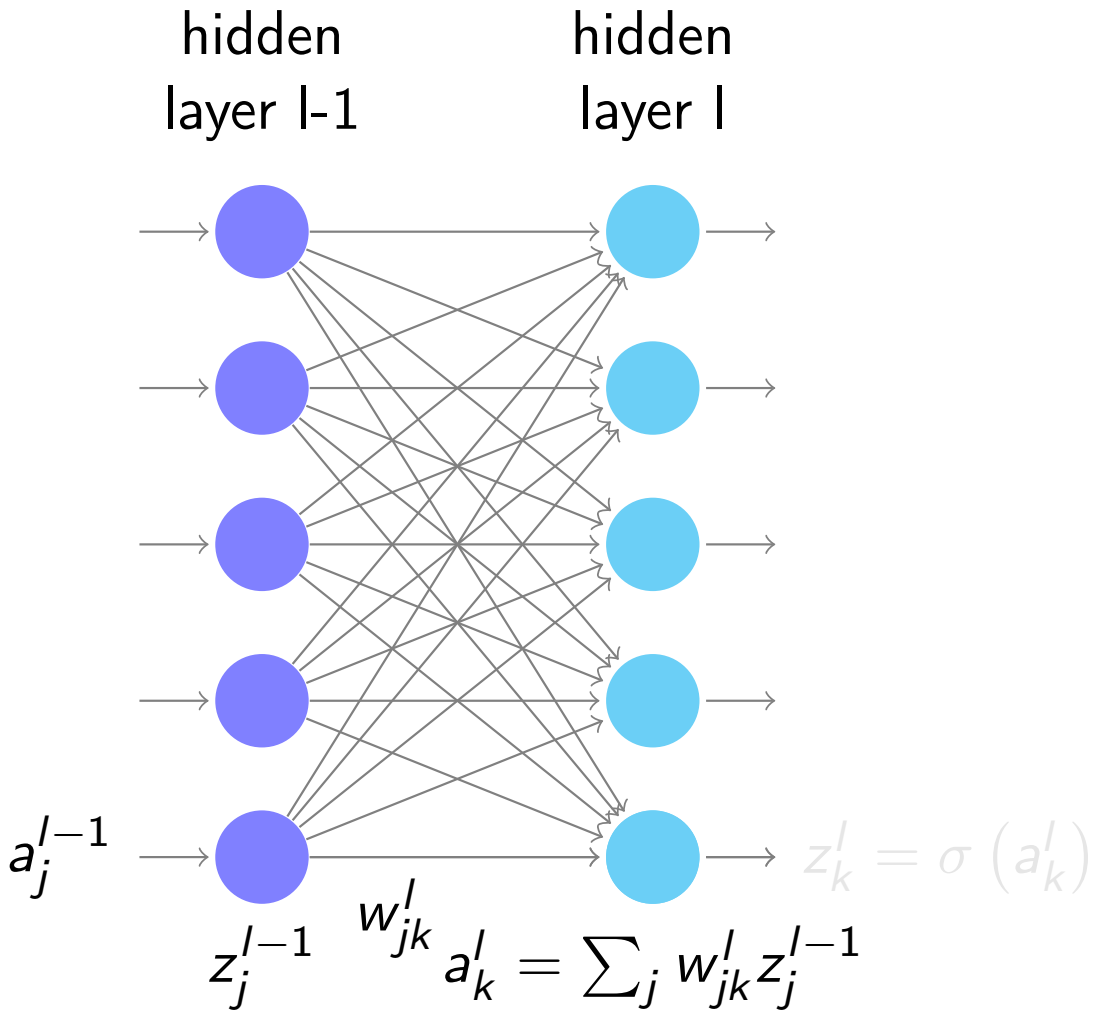
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



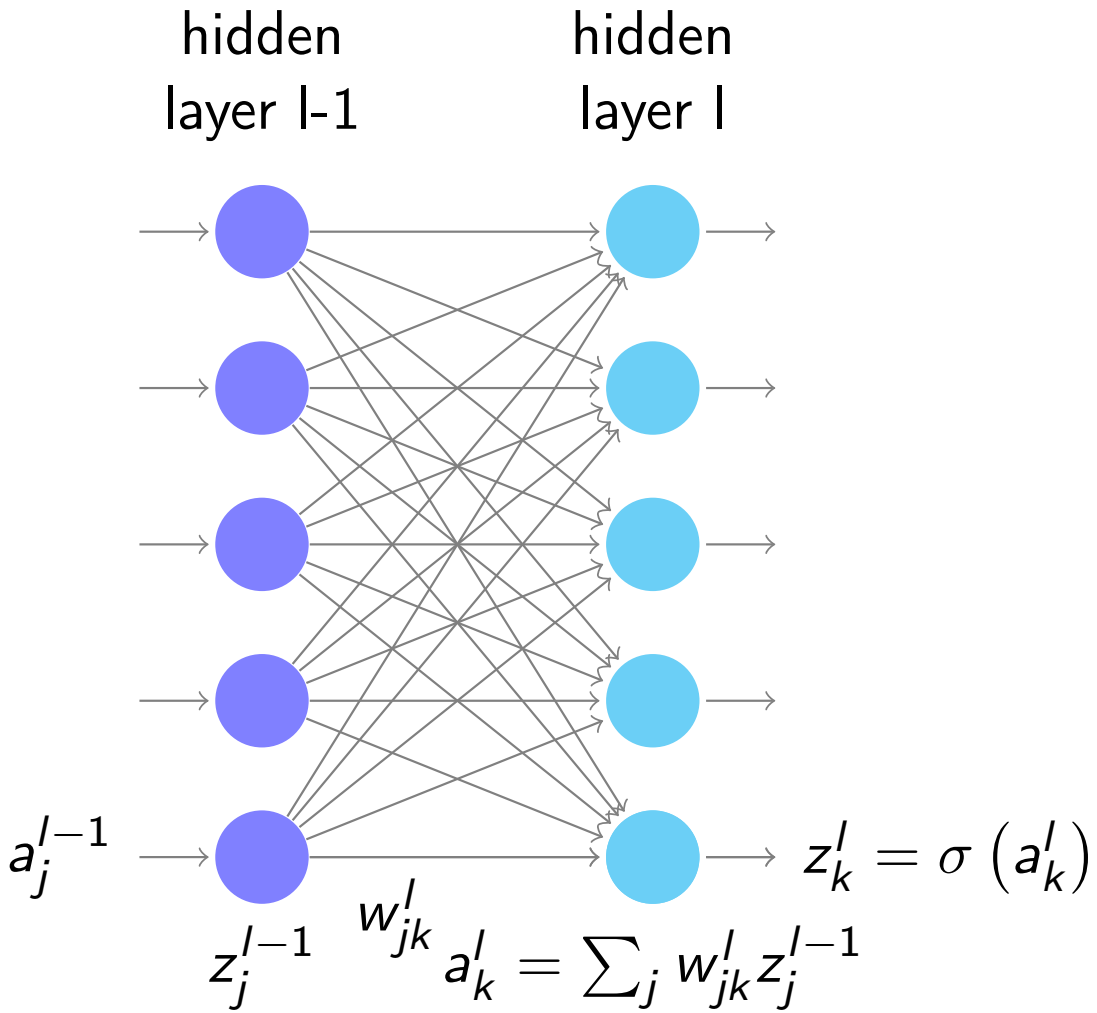
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



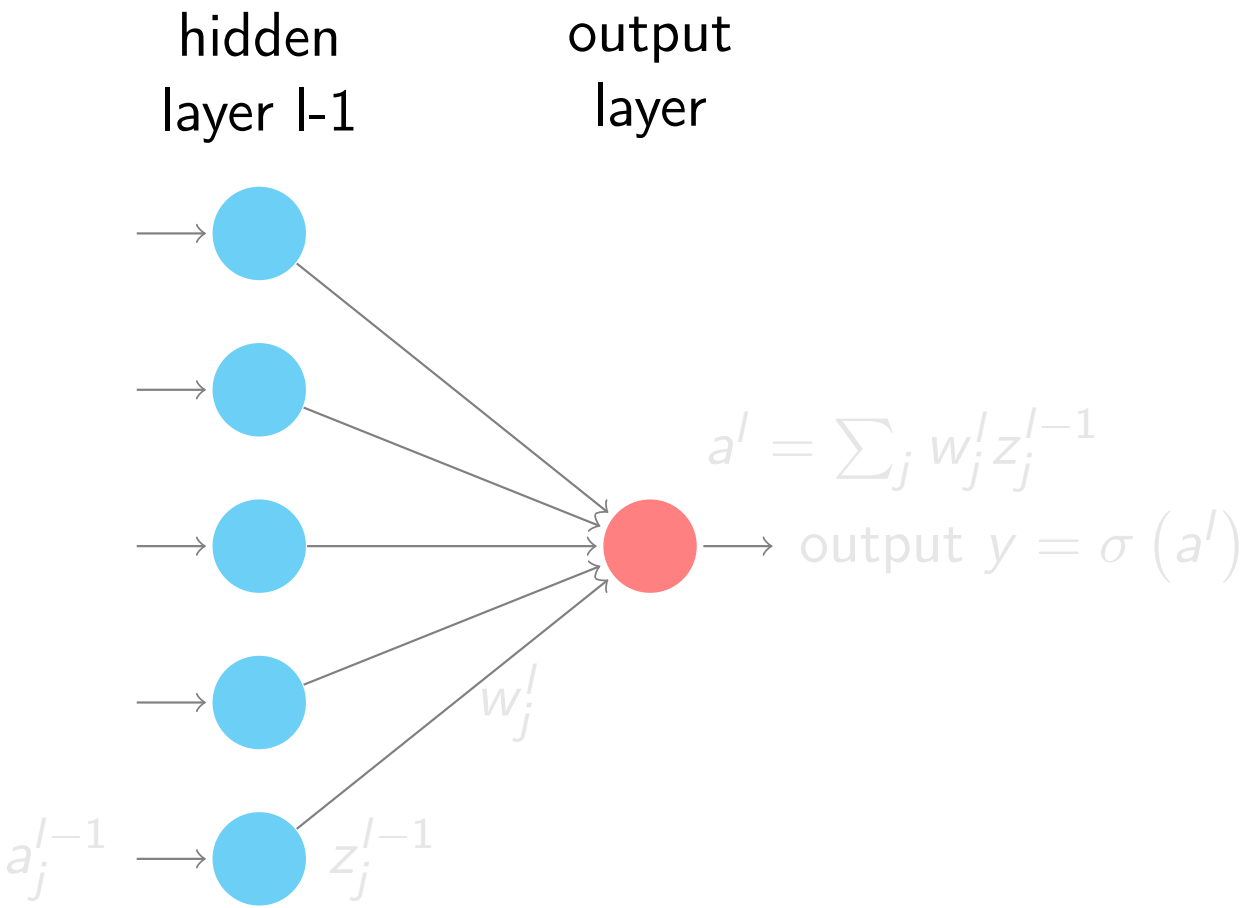
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Hidden Layer



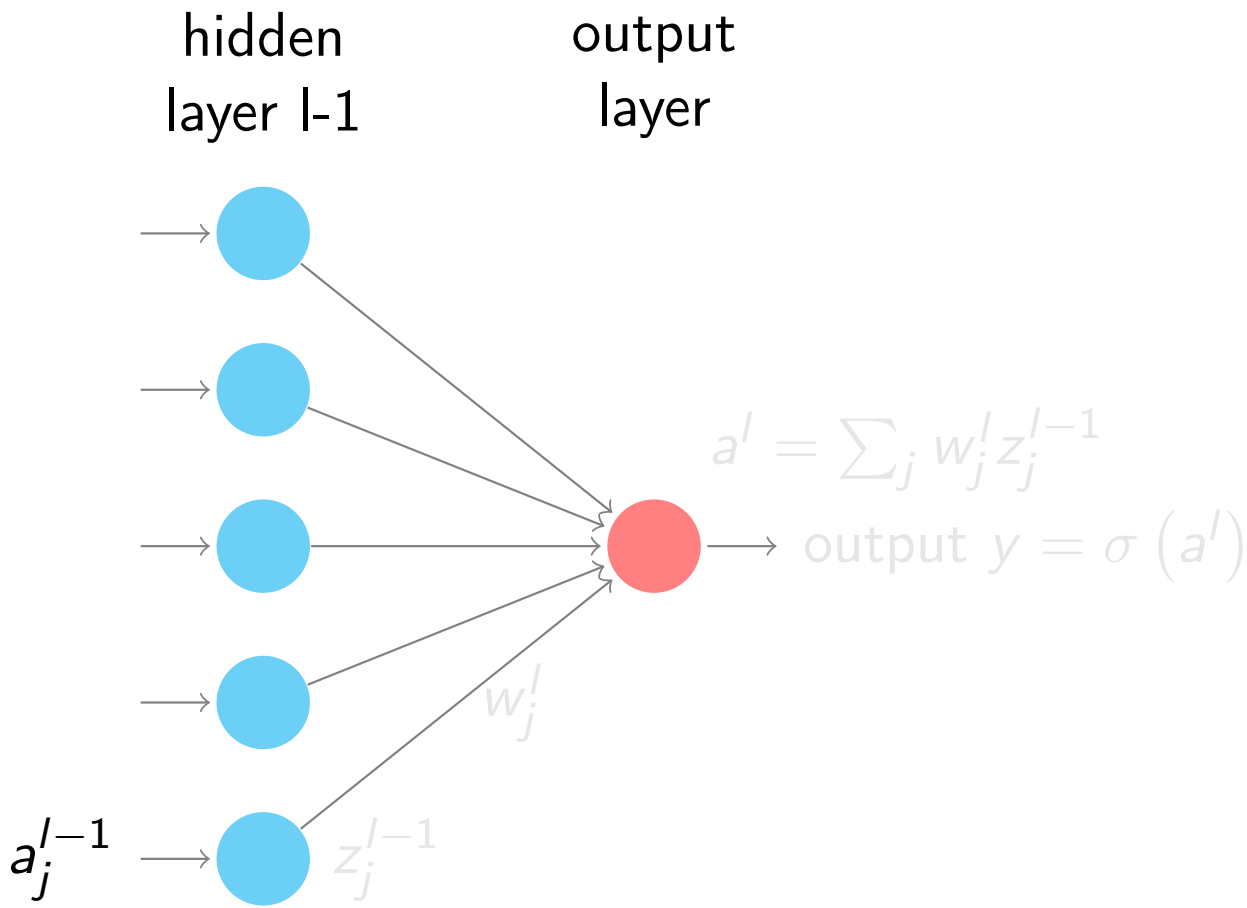
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



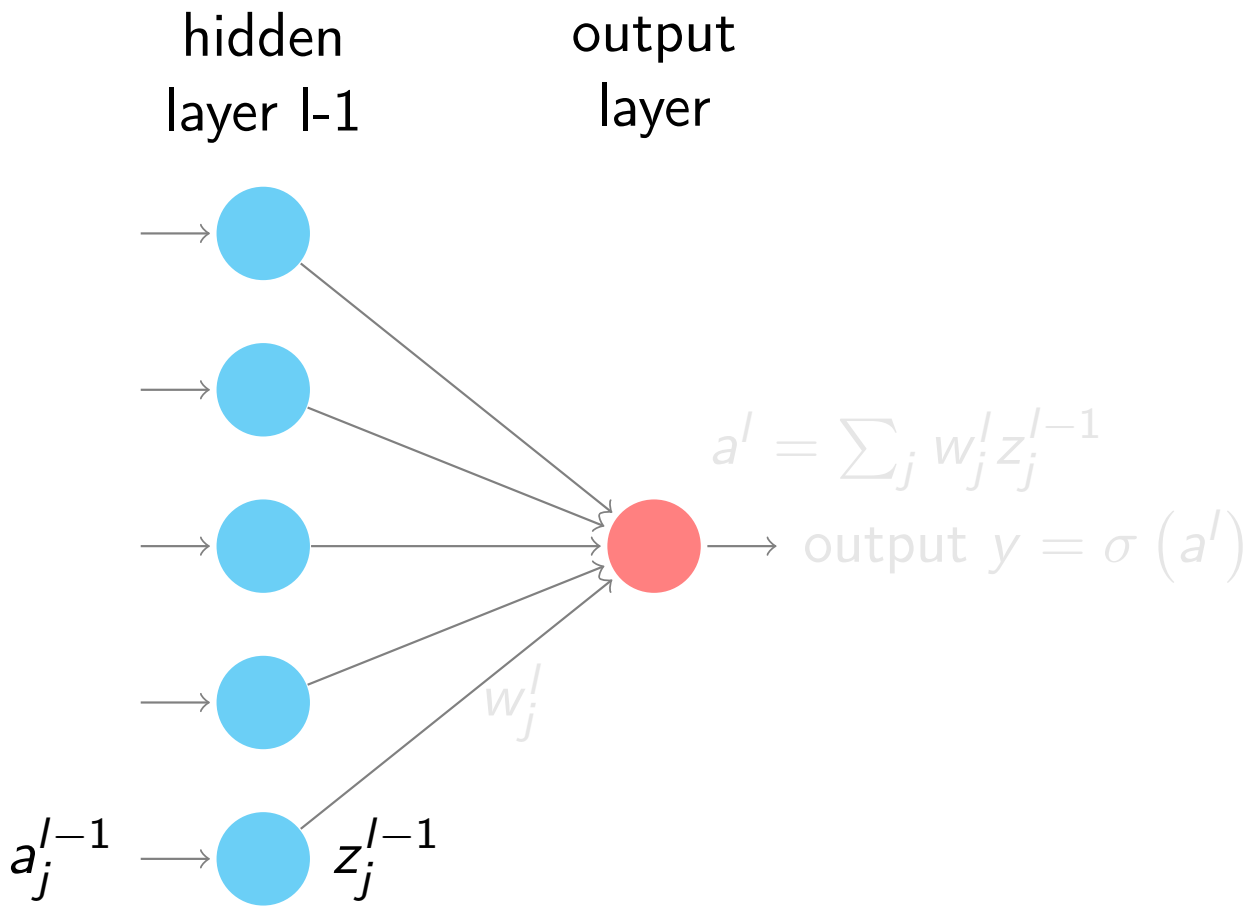
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



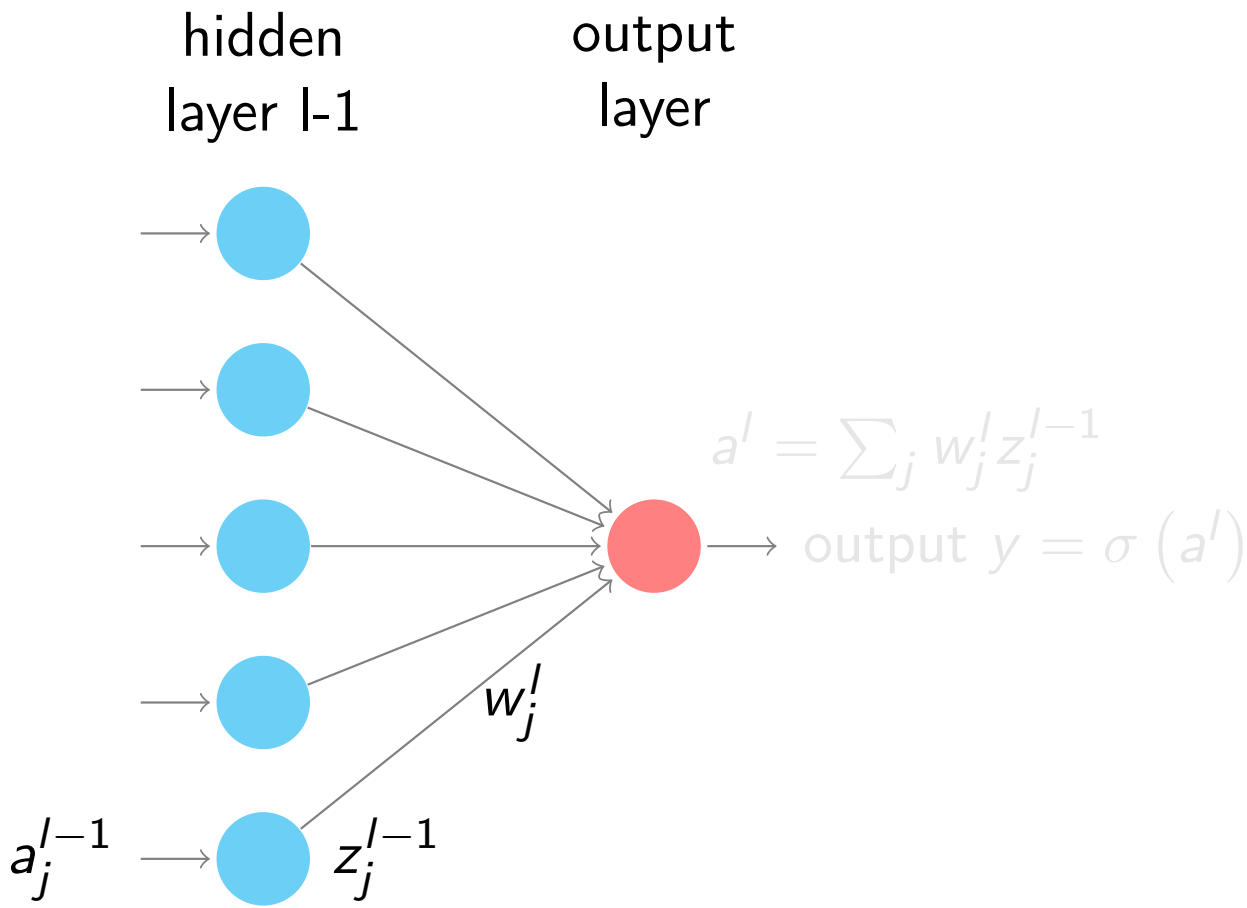
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



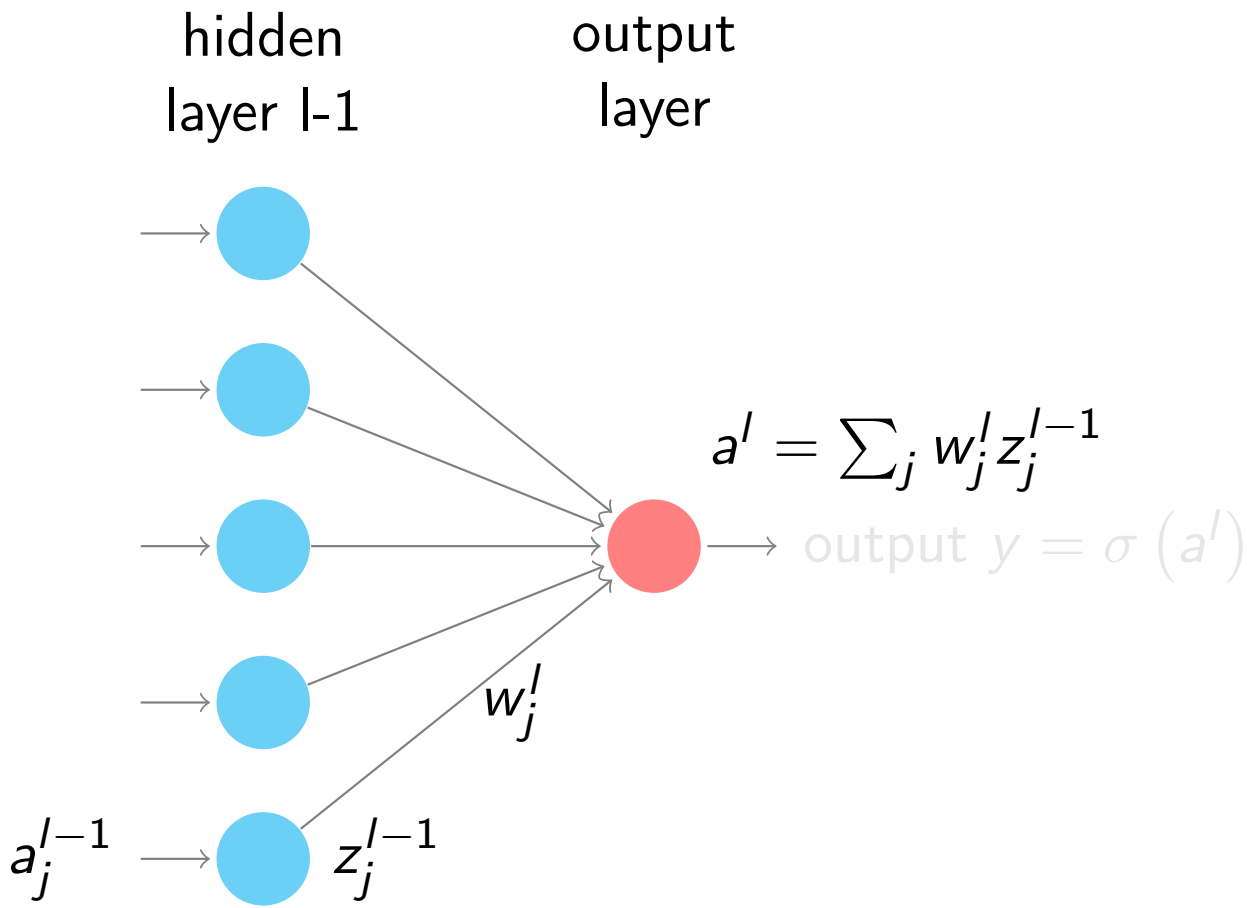
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



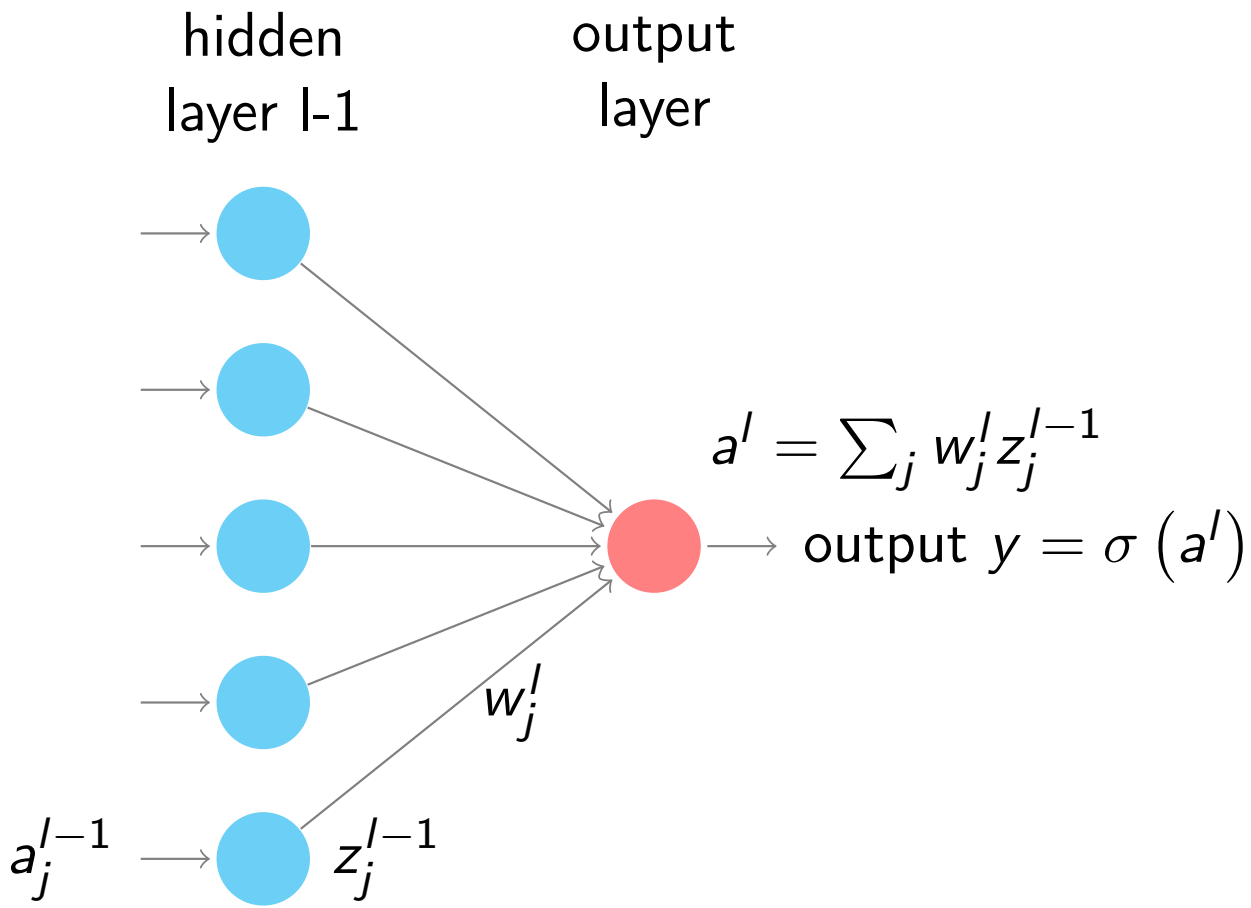
propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



propagation of the neuron activations in network for training instance **x**

Activation Propagation for Prediction: Output Layer



propagation of the neuron activations in network for training instance **x**

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we learn weights in MLP ?

online minimisation of the MSE $\Rightarrow$ consider one instance $(\mathbf{x}, t)$ at a time

- propagate $\mathbf{x}$ to compute neuron activations a_k^l and outputs z_k^l
- compare the network output y with the expected target t
- compute the adaptation Δw_{jk}^l for each connection weight w_{jk}^l

How can we obtain the adaptations Δw_{jk}^l ?

stochastic gradient descent rule: $w_{jk}^l \leftarrow w_{jk}^l - \alpha_t \frac{\delta E}{\delta w_{jk}^l}$ where $E = \frac{1}{2}(t-y)^2$

- the step size must decrease $\Rightarrow \alpha_t$ is decreased at each step t
- derivative $\frac{\delta E}{\delta w_{jk}^l}$ must be computed for each connection in each layer

Gradient Computation and Responsibility

How can we compute the derivative $\frac{\delta E}{\delta w_{jk}^l}$?

use the chain rule to decompose derivative as

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l}$$

where $\frac{\delta E}{\delta a_k^l} = \delta_k^l =$ "responsibility of neuron k in layer l for the error E" and

$$\frac{\delta a_k^l}{\delta w_{jk}^l} = \begin{cases} z_j^{l-1} & \text{for hidden and output neurons} \\ x_j & \text{for input neurons} \end{cases}$$

How can we compute the responsibility δ_k^l ?

- propagation: from left to right (forward) to compute output y
- back-propagation: from right to left (backward) to get responsibility δ_k^l

Gradient Computation and Responsibility

How can we compute the derivative $\frac{\delta E}{\delta w_{jk}^l}$?

use the chain rule to decompose derivative as

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l}$$

where $\frac{\delta E}{\delta a_k^l} = \delta_k^l =$ "responsibility of neuron k in layer l for the error E" and

$$\frac{\delta a_k^l}{\delta w_{jk}^l} = \begin{cases} z_j^{l-1} & \text{for hidden and output neurons} \\ x_j & \text{for input neurons} \end{cases}$$

How can we compute the responsibility δ_k^l ?

- propagation: from left to right (forward) to compute output y
- back-propagation: from right to left (backward) to get responsibility δ_k^l

Gradient Computation and Responsibility

How can we compute the derivative $\frac{\delta E}{\delta w_{jk}^l}$?

use the chain rule to decompose derivative as

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l}$$

where $\frac{\delta E}{\delta a_k^l} = \delta_k^l =$ "responsibility of neuron k in layer l for the error E" and

$$\frac{\delta a_k^l}{\delta w_{jk}^l} = \begin{cases} z_j^{l-1} & \text{for hidden and output neurons} \\ x_j & \text{for input neurons} \end{cases}$$

How can we compute the responsibility δ_k^l ?

- propagation: from left to right (forward) to compute output y
- back-propagation: from right to left (backward) to get responsibility δ_k^l

Gradient Computation and Responsibility

How can we compute the derivative $\frac{\delta E}{\delta w_{jk}^l}$?

use the chain rule to decompose derivative as

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l}$$

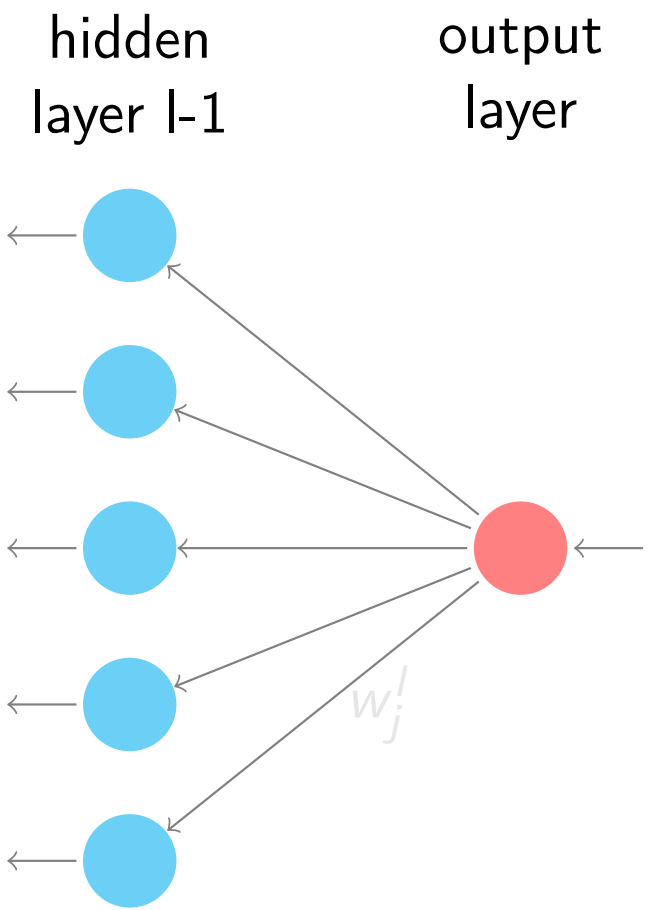
where $\frac{\delta E}{\delta a_k^l} = \delta_k^l =$ "responsibility of neuron k in layer l for the error E" and

$$\frac{\delta a_k^l}{\delta w_{jk}^l} = \begin{cases} z_j^{l-1} & \text{for hidden and output neurons} \\ x_j & \text{for input neurons} \end{cases}$$

How can we compute the responsibility δ_k^l ?

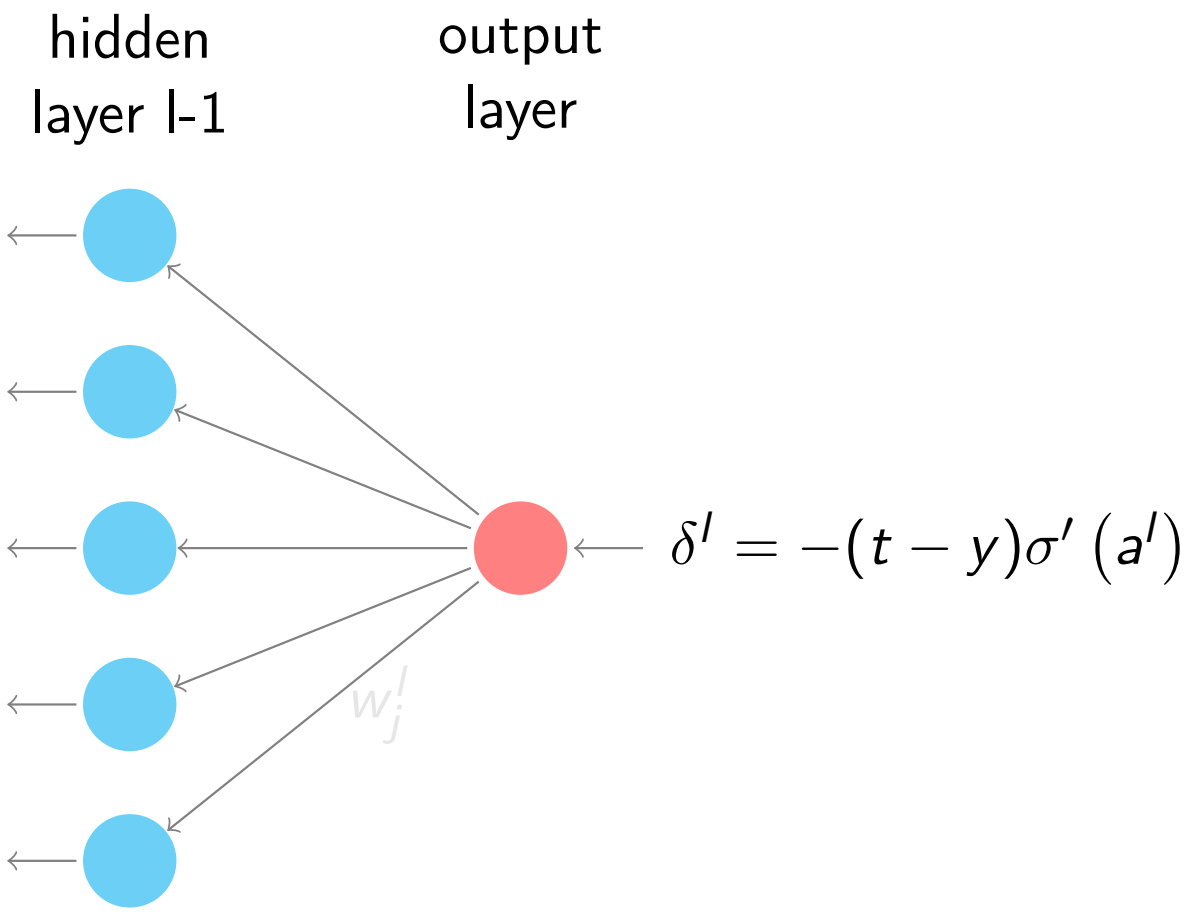
- propagation: from left to right (forward) to compute output y
- back-propagation: from right to left (backward) to get responsibility δ_k^l

Error Back-Propagation: Output Layer



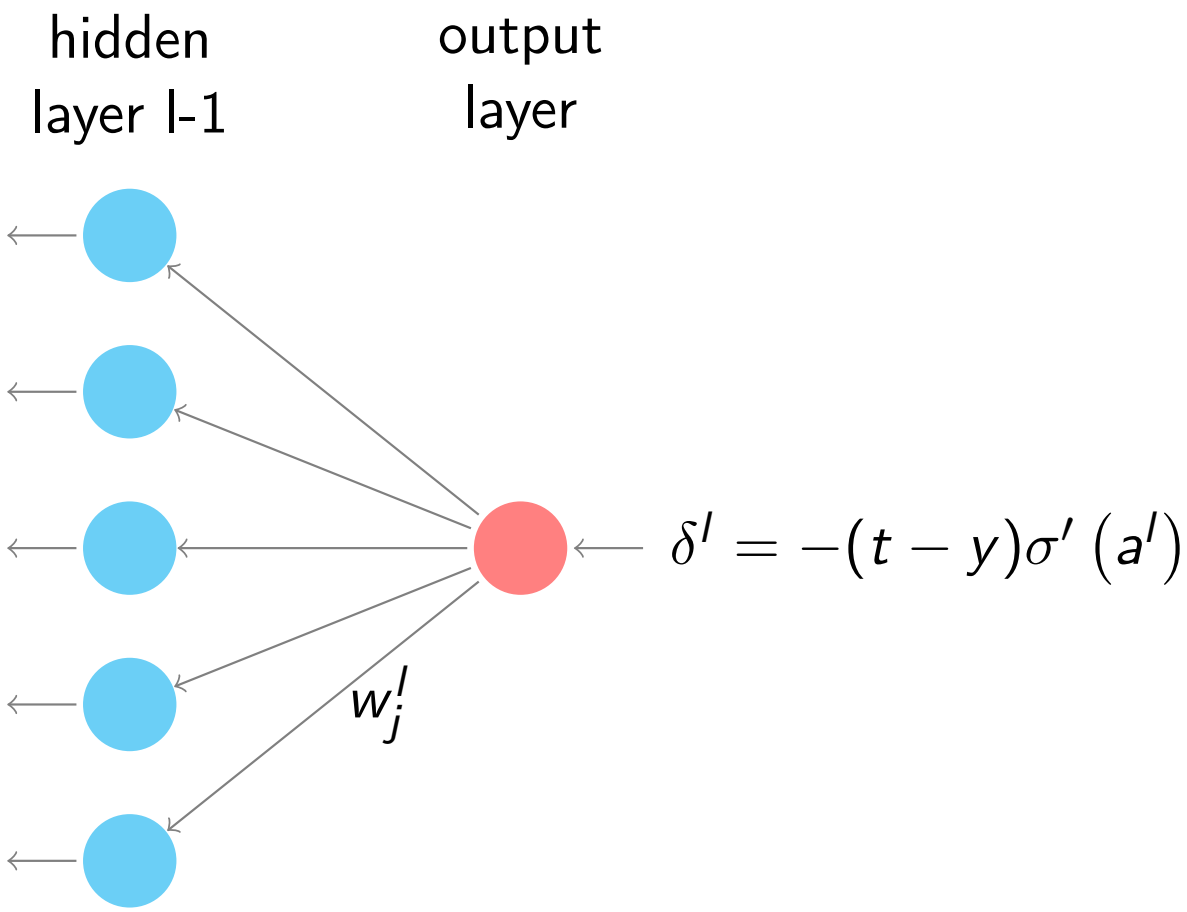
back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

Error Back-Propagation: Output Layer



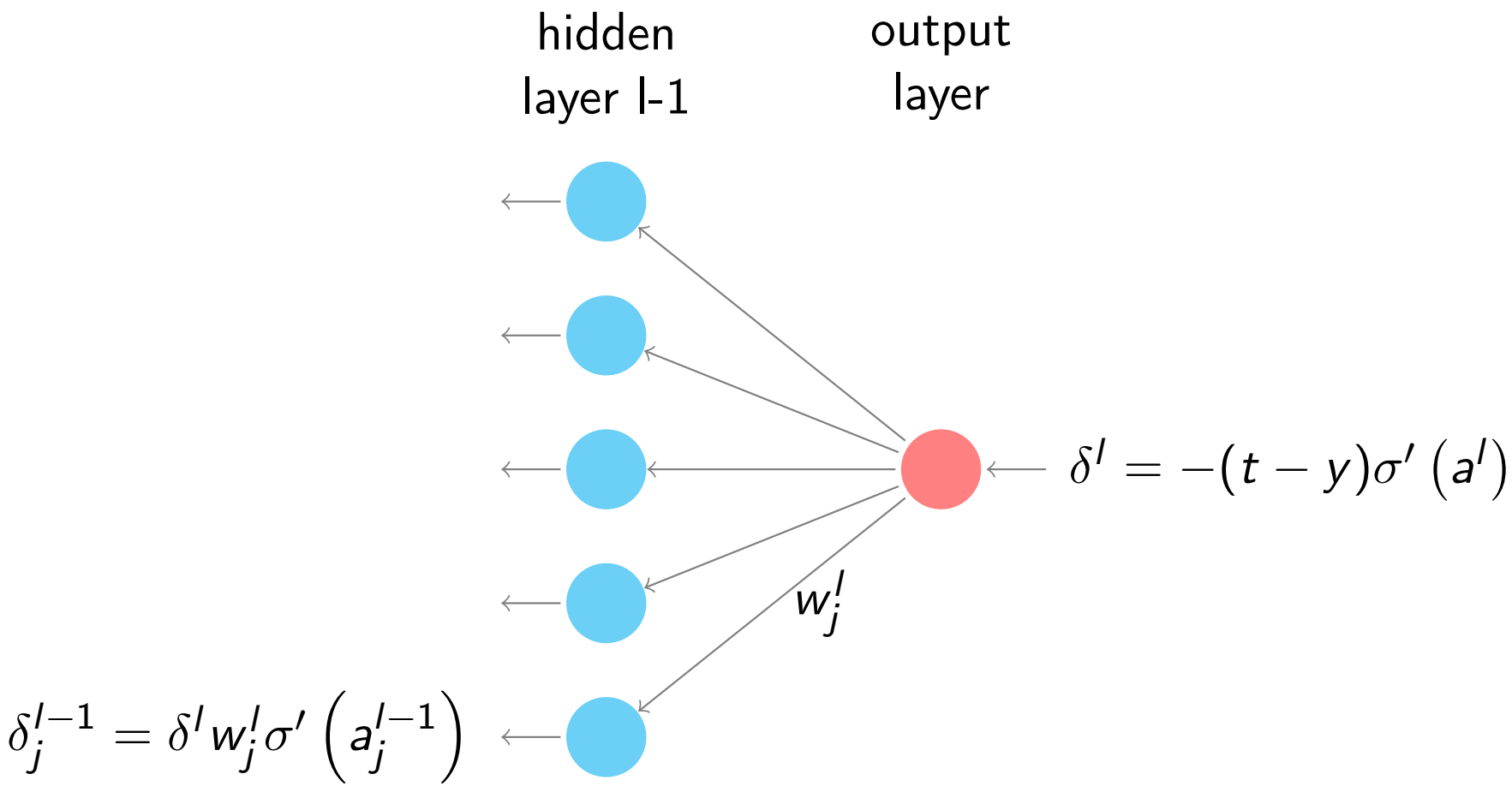
back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

Error Back-Propagation: Output Layer



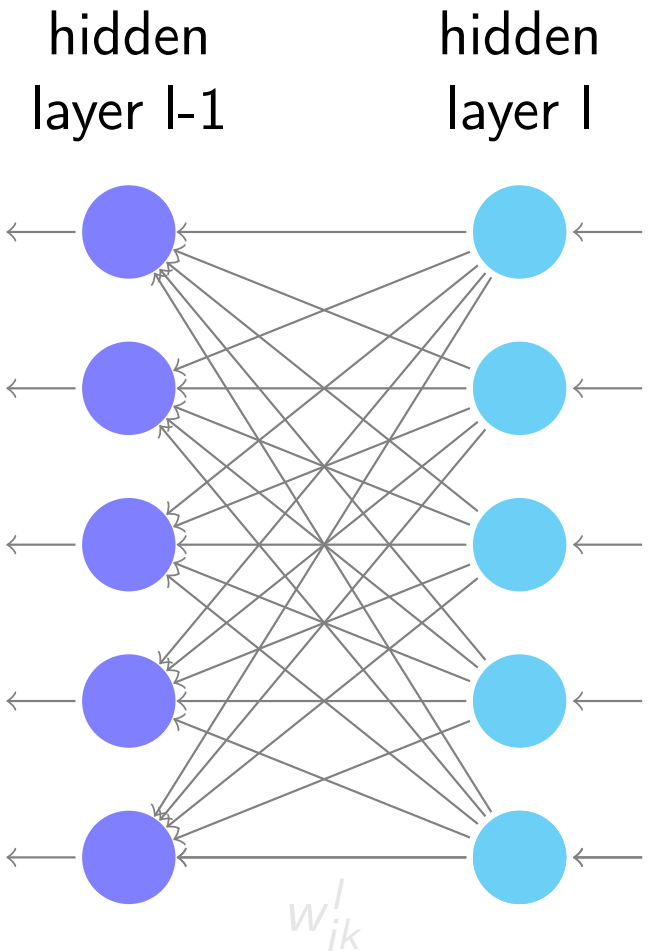
back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

Error Back-Propagation: Output Layer



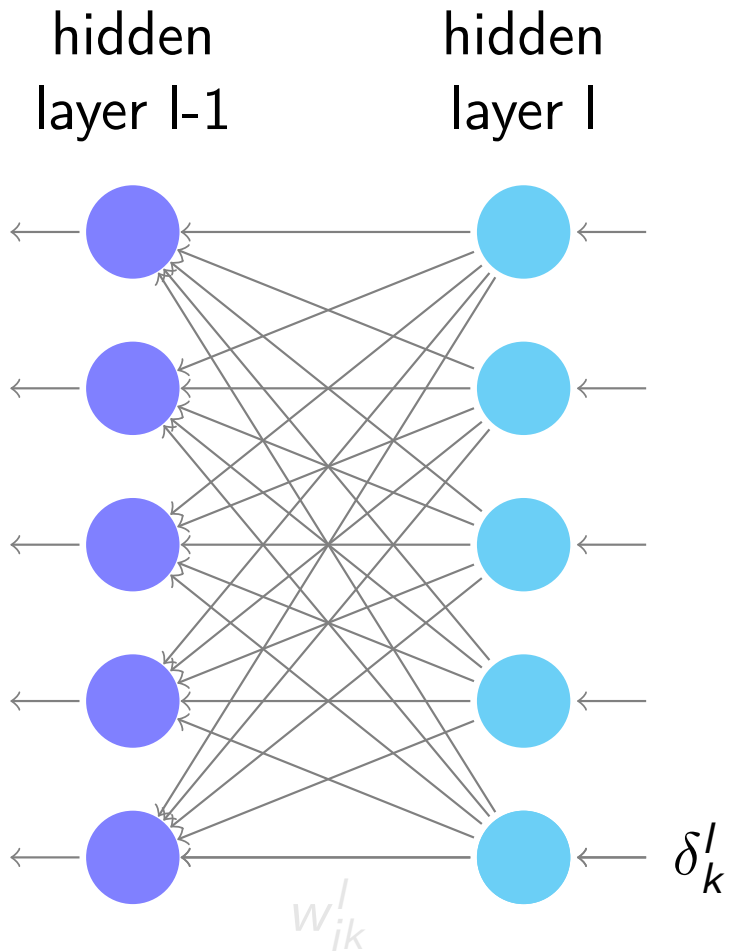
back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

Error Back-Propagation: Hidden Layer



back-propagation of the responsibility in network for error $E = \frac{1}{2}(t - y)^2$

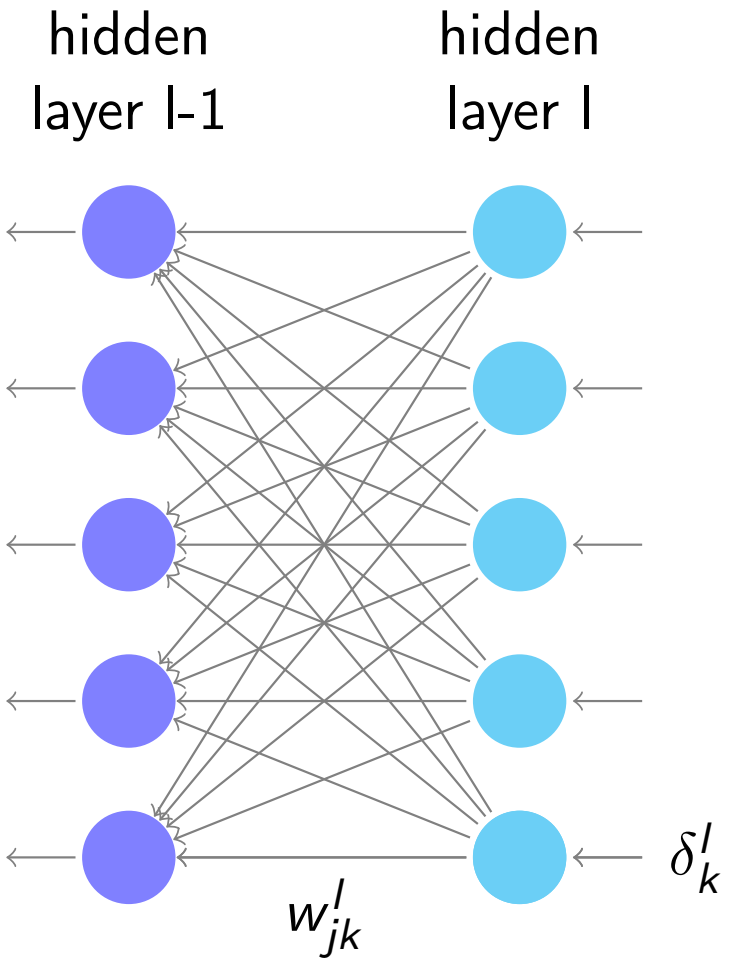
Error Back-Propagation: Hidden Layer



$$\delta_j^{l-1} = \sigma'\left(a_j^{l-1}\right) \sum_k \delta_k^l w_{jk}^l$$

back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

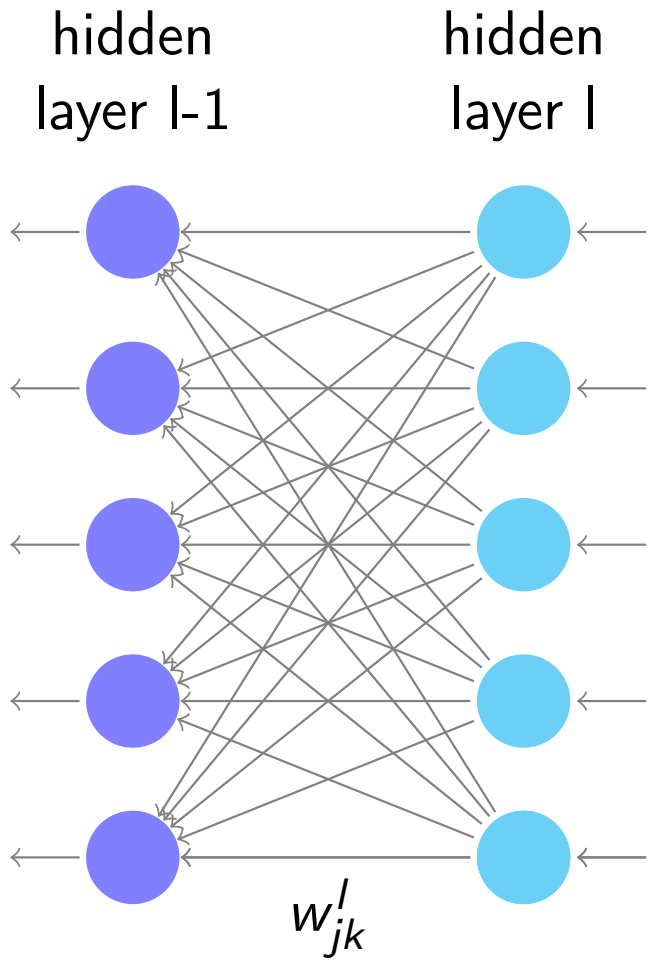
Error Back-Propagation: Hidden Layer



$$\delta_j^{l-1} = \sigma'\left(a_j^{l-1}\right) \sum_k \delta_k^l w_{jk}^l$$

back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

Error Back-Propagation: Hidden Layer



$$\delta_j^{l-1} = \sigma'\left(a_j^{l-1}\right) \sum_k \delta_k^l w_{jk}^l$$

back-propagation of the responsibility in network for error $E = \frac{1}{2}(t-y)^2$

About the Activation Functions

in regression, MLP often have sigmoid hidden units and a linear output unit

for the linear output unit:

$$\delta^l = -(t - y) = y - t$$

for sigmoid hidden units:

$$\delta_j^{l-1} = \sigma\left(a_j^{l-1}\right)\left[1 - \sigma\left(a_j^{l-1}\right)\right]\sum_k \delta_k^l w_{jk}^l$$

About the Activation Functions

in regression, MLP often have sigmoid hidden units and a linear output unit

for the linear output unit:

$$\delta^l = -(t - y) = y - t$$

for sigmoid hidden units:

$$\delta_j^{l-1} = \sigma\left(a_j^{l-1}\right)\left[1 - \sigma\left(a_j^{l-1}\right)\right]\sum_k \delta_k^l w_{jk}^l$$

About the Activation Functions

in regression, MLP often have sigmoid hidden units and a linear output unit

for the linear output unit:

$$\delta^l = -(t - y) = y - t$$

for sigmoid hidden units:

$$\delta_j^{l-1} = \sigma\left(a_j^{l-1}\right)\left[1 - \sigma\left(a_j^{l-1}\right)\right]\sum_k \delta_k^l w_{jk}^l$$

Gradient Computation and Responsibility

reminder: we used the chain rule to decompose derivative as

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l}$$

where $\frac{\delta E}{\delta a_k^l} = \delta_k^l =$ "responsibility of neuron k in layer l for the error E"

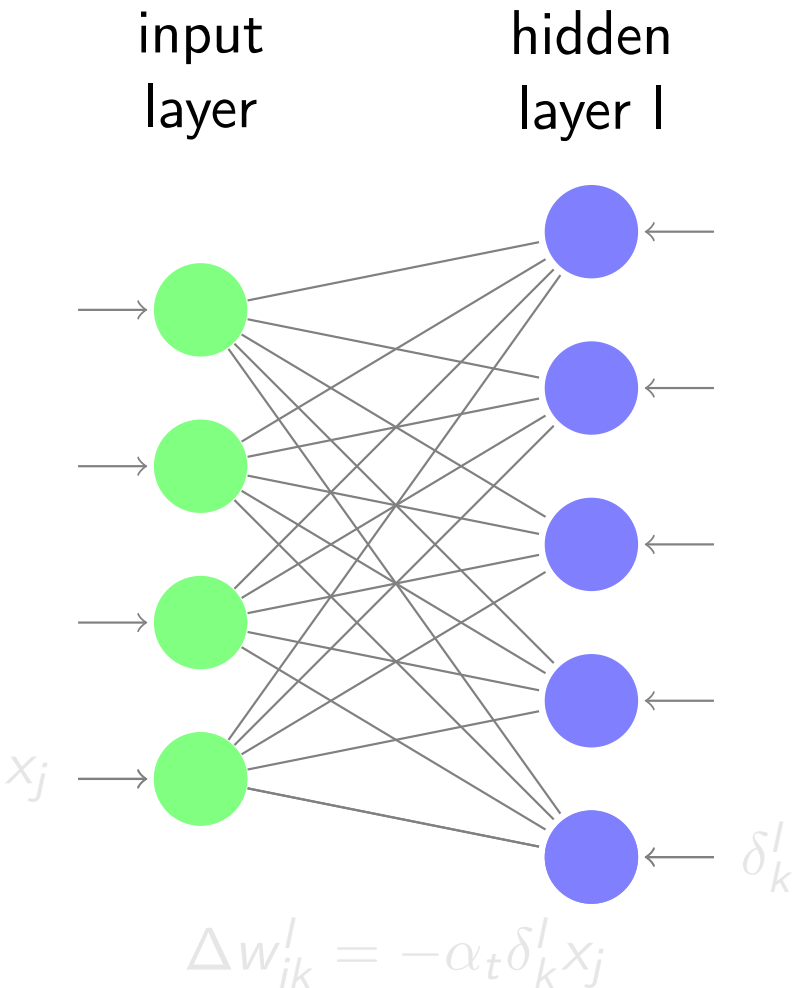
for first layer:

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l} = \delta_k^l x_j$$

for next layers:

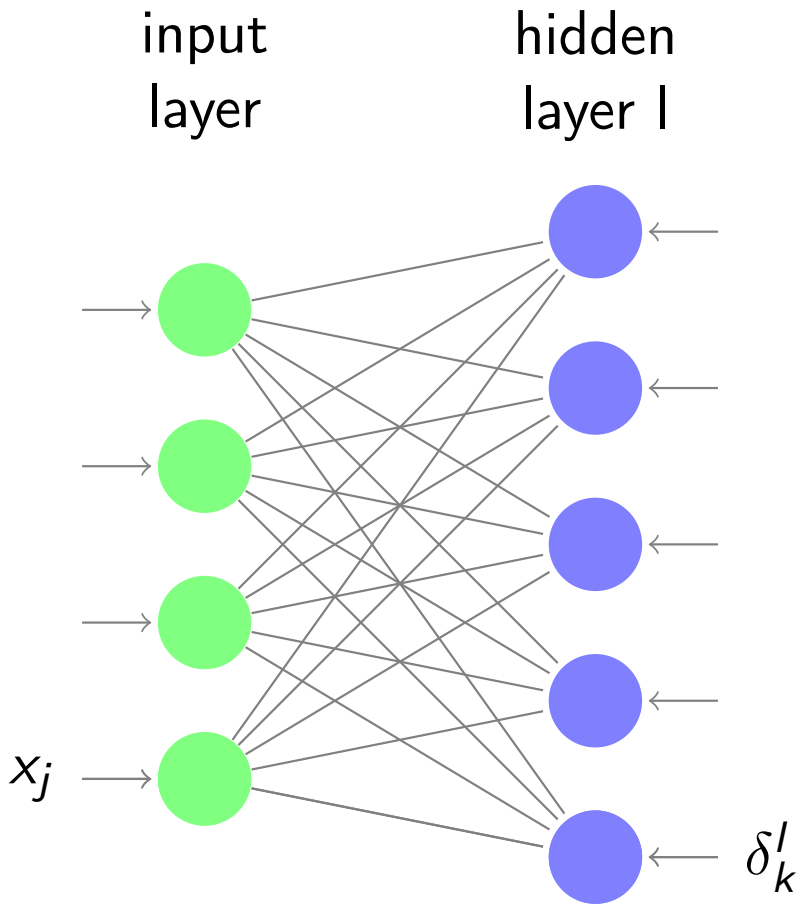
$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_k^l} \times \frac{\delta a_k^l}{\delta w_{jk}^l} = \delta_k^l z_j^{l-1}$$

Gradient Computation: Input Layer



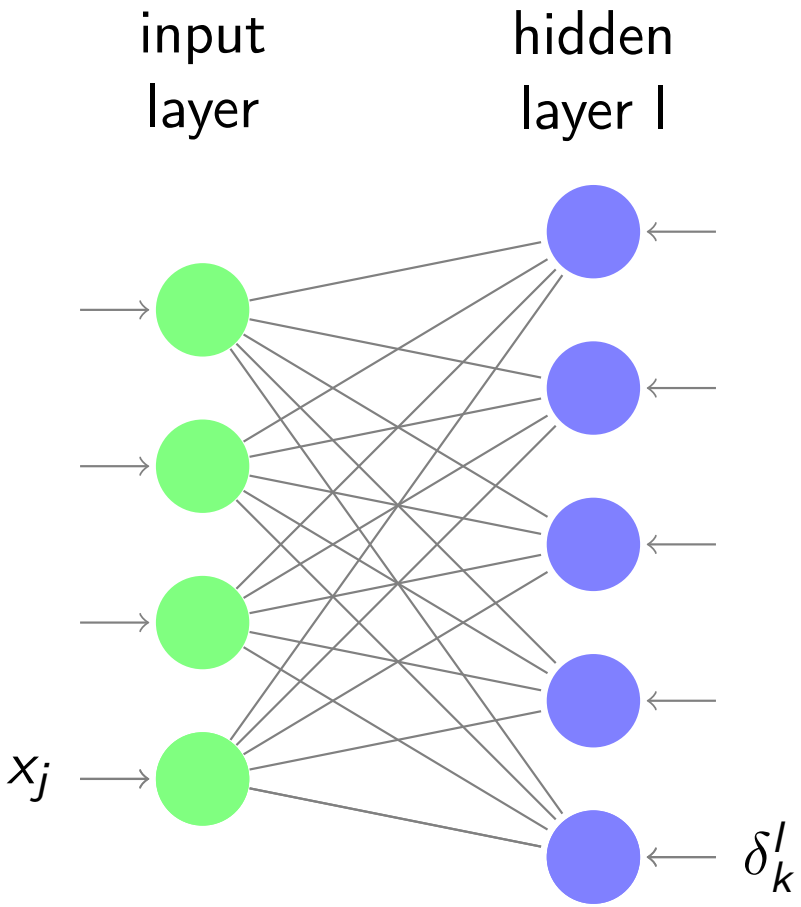
computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Gradient Computation: Input Layer



computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

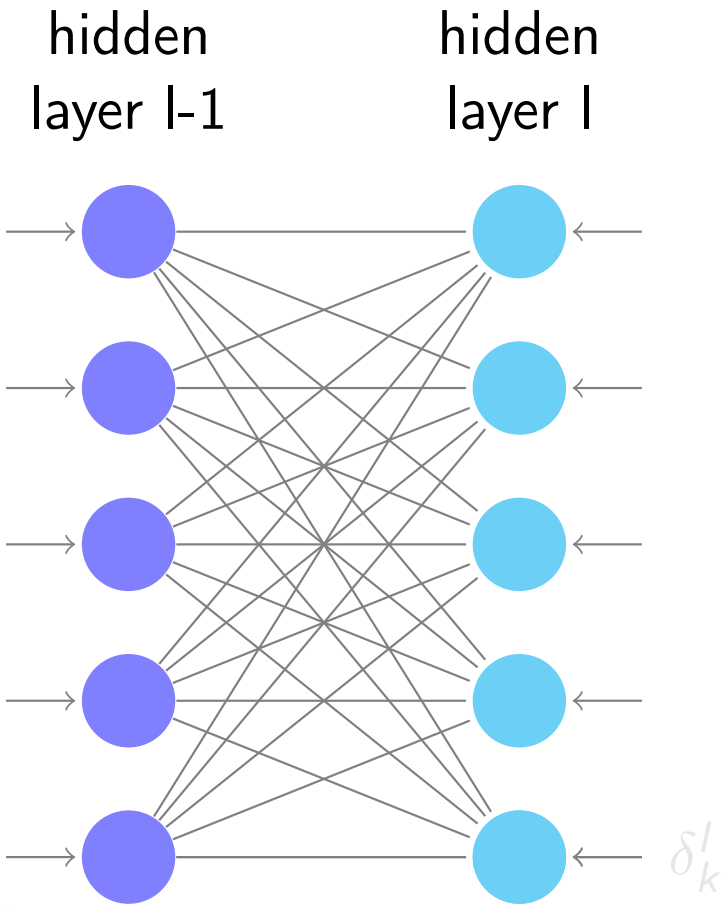
Gradient Computation: Input Layer



$$\Delta w_{jk}^l = -\alpha_t \delta_k^l x_j$$

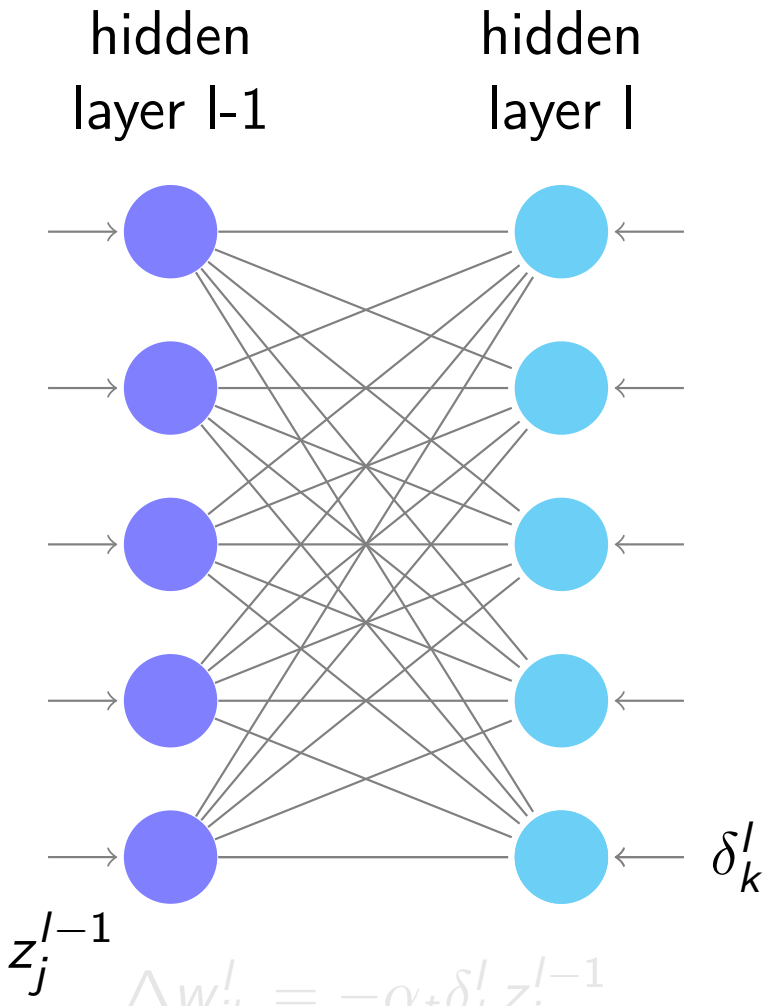
computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Gradient Computation: Hidden Layer



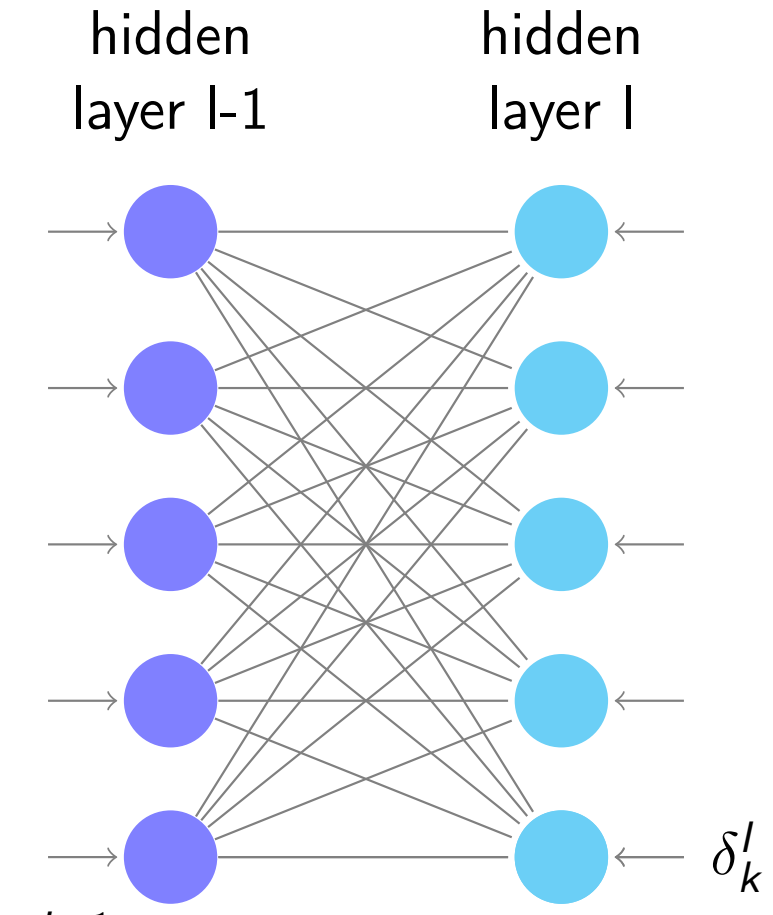
computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Gradient Computation: Hidden Layer



computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

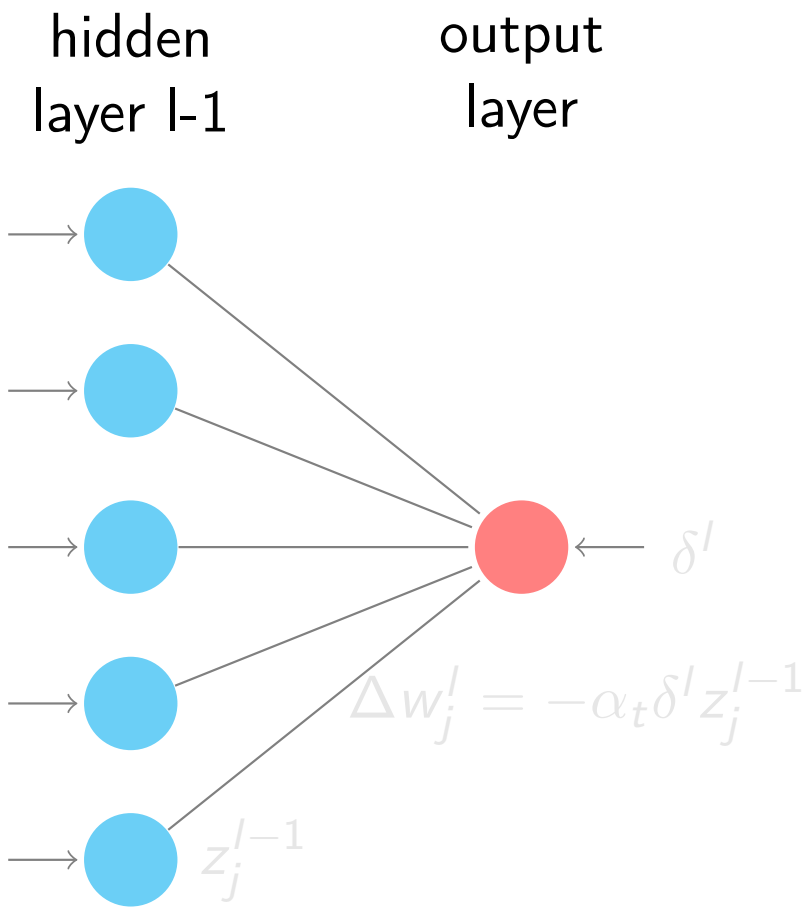
Gradient Computation: Hidden Layer



$$\Delta w_{jk}^l = -\alpha_t \delta_k^l z_j^{l-1}$$

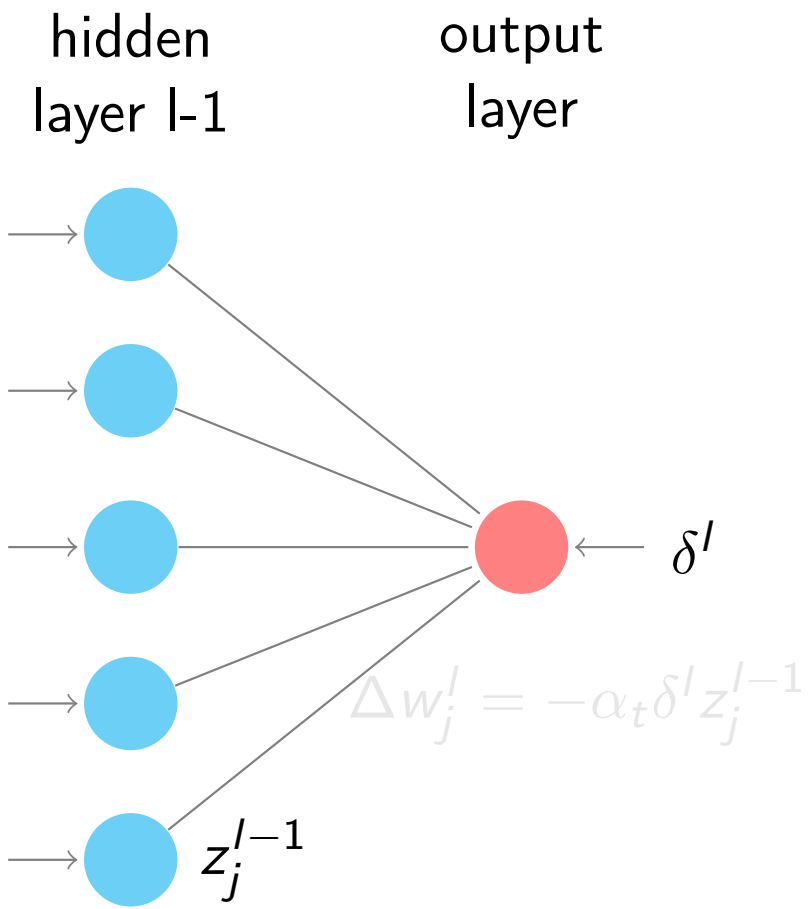
computation of the weight adaptations in network for error $E = \frac{1}{2}(t-y)^2$

Gradient Computation: Output Layer



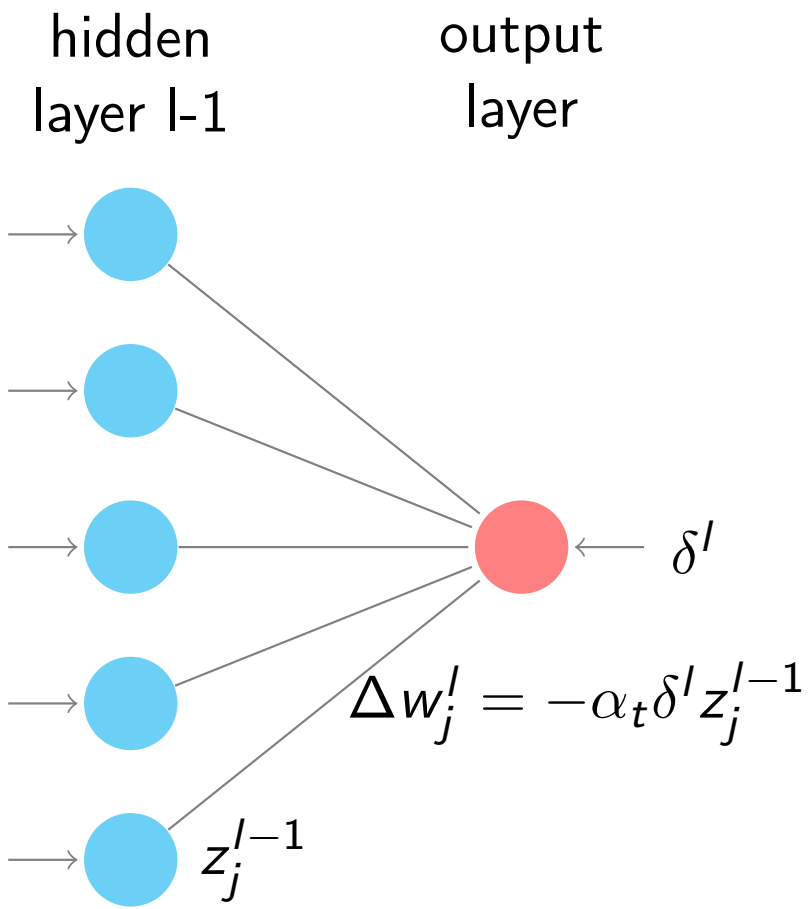
computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Gradient Computation: Output Layer



computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Gradient Computation: Output Layer



computation of the weight adaptations in network for error $E = \frac{1}{2}(t - y)^2$

Pros and Cons of Learning MLPs with Back-Propagation

Advantages

- ✓ not so hard to implement, once you have grasped the basics
- ✓ each step is cheap, only δ_k^l and $\sigma'\left(a_j^{l-1}\right)$ must be computed

Limitations

- × many steps are necessary to converge (thousands or worse)
- × gradient descent is likely to get stuck in local minima of the MSE
- × MSE has many local optima (e.g. because of symmetry in MLPs)
- × very hard to find a good initial choice of weights

Pros and Cons of Learning MLPs with Back-Propagation

Advantages

- ✓ not so hard to implement, once you have grasped the basics
- ✓ each step is cheap, only δ_k^l and $\sigma'\left(a_j^{l-1}\right)$ must be computed

Limitations

- × many steps are necessary to converge (thousands or worse)
- × gradient descent is likely to get stuck in local minima of the MSE
- × MSE has many local optima (e.g. because of symmetry in MLPs)
- × very hard to find a good initial choice of weights

Pros and Cons of Learning MLPs with Back-Propagation

Advantages

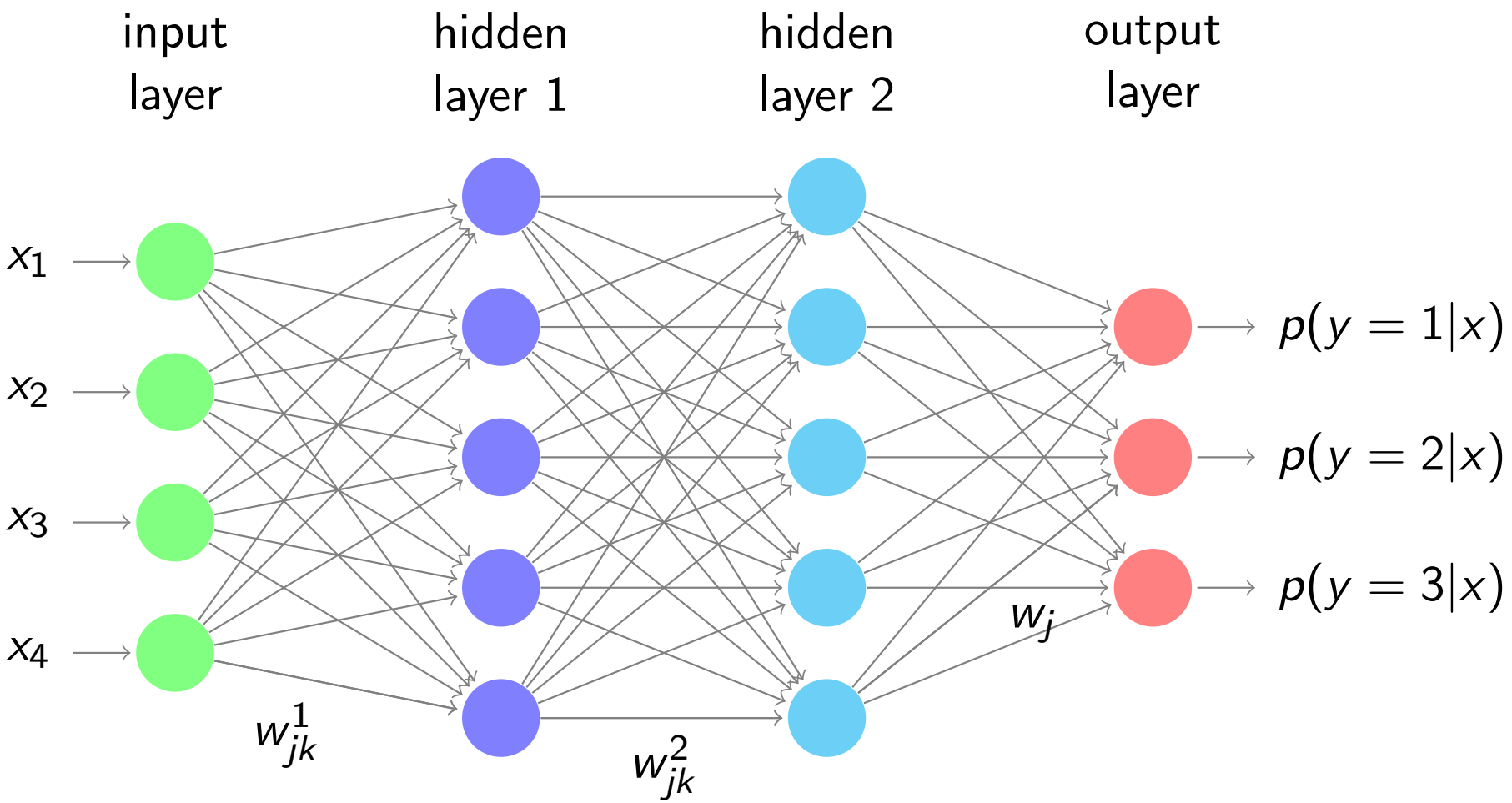
- ✓ not so hard to implement, once you have grasped the basics
- ✓ each step is cheap, only δ_k^l and $\sigma'\left(a_j^{l-1}\right)$ must be computed

Limitations

- × many steps are necessary to converge (thousands or worse)
- × gradient descent is likely to get stuck in local minima of the MSE
- × MSE has many local optima (e.g. because of symmetry in MLPs)
- × very hard to find a good initial choice of weights

Classification with Multi-Layer Perceptrons

Probabilistic MLP with Sigmoid Output Units



output k = conditional probability of class $k = \frac{1}{1+\exp(-a_k^l)}$

The Future of Neural Networks

Beyond Back-Propagation: Deep Learning ?

The second death of artificial neural networks

in the mid 90's, new models appeared: the support vector machines

- easier to optimise since their objective function is convex
- no local minima, no endless gradient descent, no α_t scheme tuning
- kernels can be used to learn from non-numerical data

in the 2000s, researchers abandoned ANNs and they "died" once again

Deep learning: the second renaissance ?

deep learning is a recent, efficient way to learn hidden weights

- hidden layers are used to perform feature extraction
- learning of hidden weights is unsupervised $\Rightarrow$ data representation
- successful in computer vision, speech recognition and NLP

Google TechTalk: `https://www.youtube.com/watch?v=AyzOUbkUf3M`

Beyond Back-Propagation: Deep Learning ?

The second death of artificial neural networks

in the mid 90's, new models appeared: the support vector machines

- easier to optimise since their objective function is convex
- no local minima, no endless gradient descent, no α_t scheme tuning
- kernels can be used to learn from non-numerical data

in the 2000s, researchers abandoned ANNs and they "died" once again

Deep learning: the second renaissance ?

deep learning is a recent, efficient way to learn hidden weights

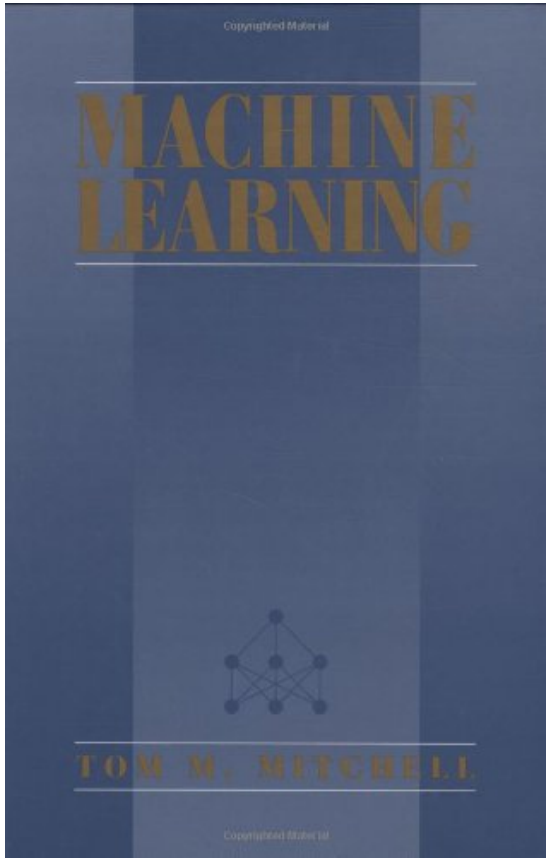
- hidden layers are used to perform feature extraction
- learning of hidden weights is unsupervised $\Rightarrow$ data representation
- successful in computer vision, speech recognition and NLP

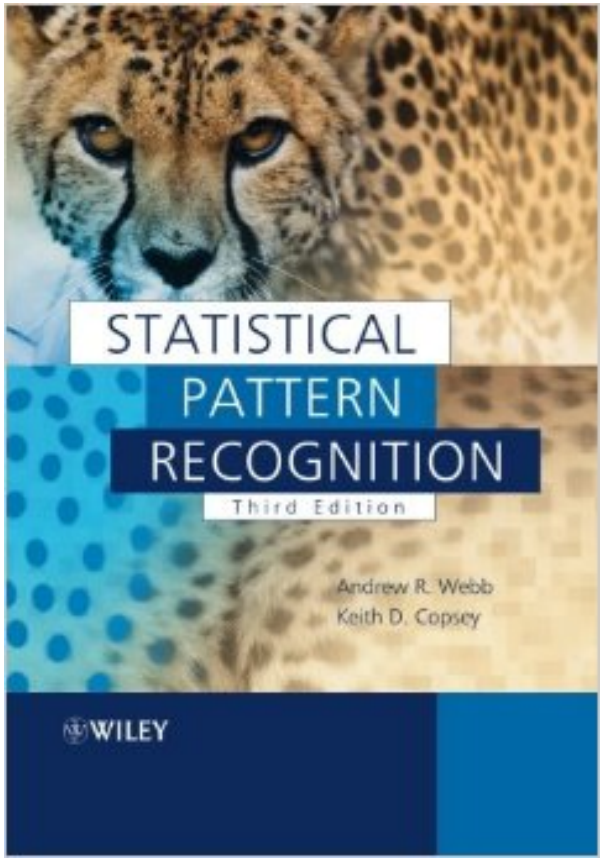
Google TechTalk: `https://www.youtube.com/watch?v=AyzOUbkUf3M`

Outline of this Lesson

- a bit of history
- single-layer neural networks
  - simple adaptation rules
  - the first AI winter
- multi-layer neural networks
  - error back-propagation
  - classification with multi-layer perceptrons
- the future of neural networks

References


Copyrighted Material
MACHINE LEARNING
TOM M. MITCHELL
Copyrighted Material


STATISTICAL
PATTERN
RECOGNITION
Third Edition
Andrew R. Webb
Keith D. Copsey
WILEY