Artificial Neural Networks

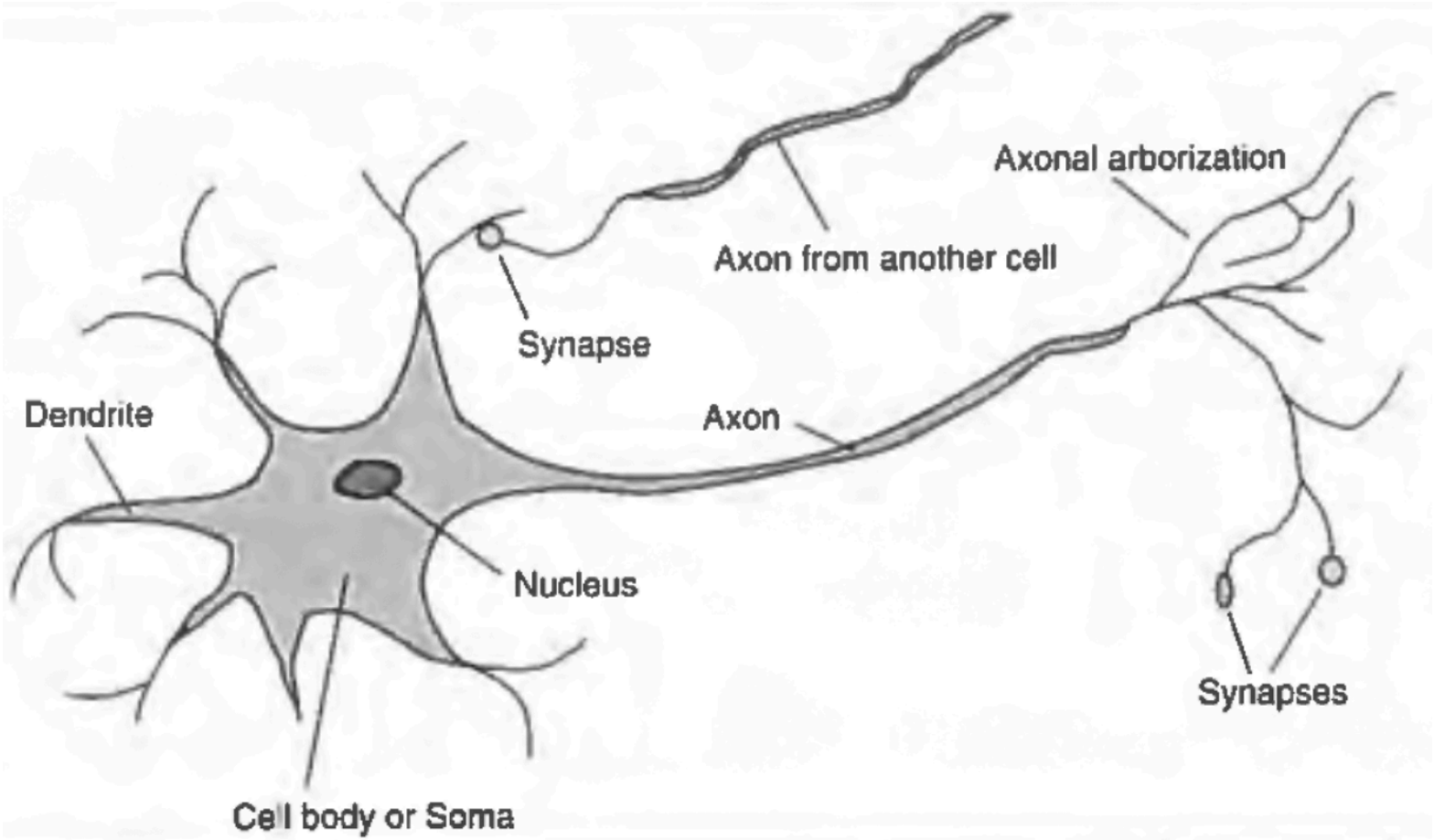
Deep Learning Course, RUC – Sep 21, 2017

Henning Christiansen

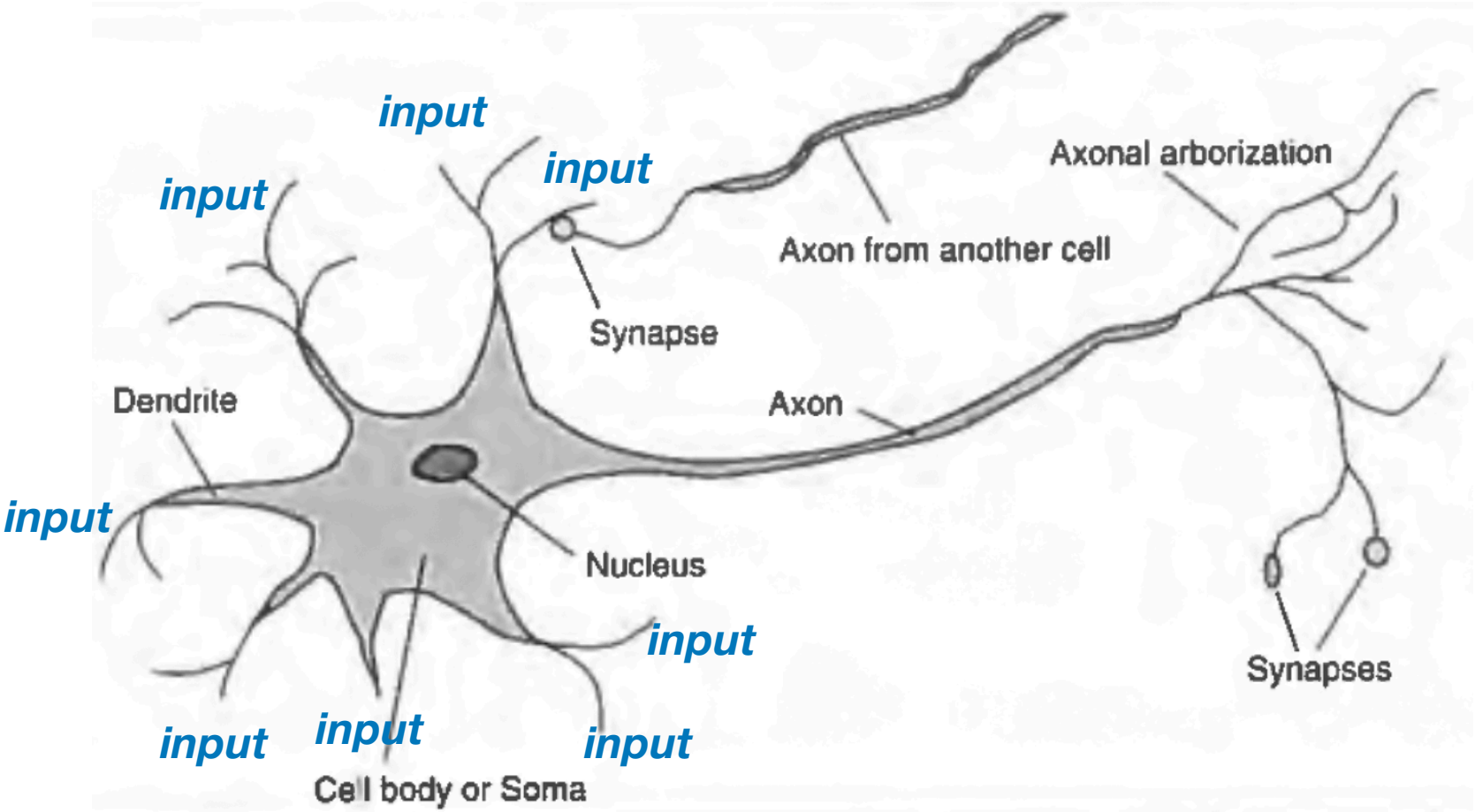
Overview

- Intro plus a bit of history
- The computational unit, a "neural node"
- Putting them together to form net
- How to evaluate a net ("feed forward")
- How to learn? The back-propagation algorithm (heavy stuff)
- Your exercise: Implementing back-propagation using Processing, starting from a program with data structures plus the feed forward algorithm

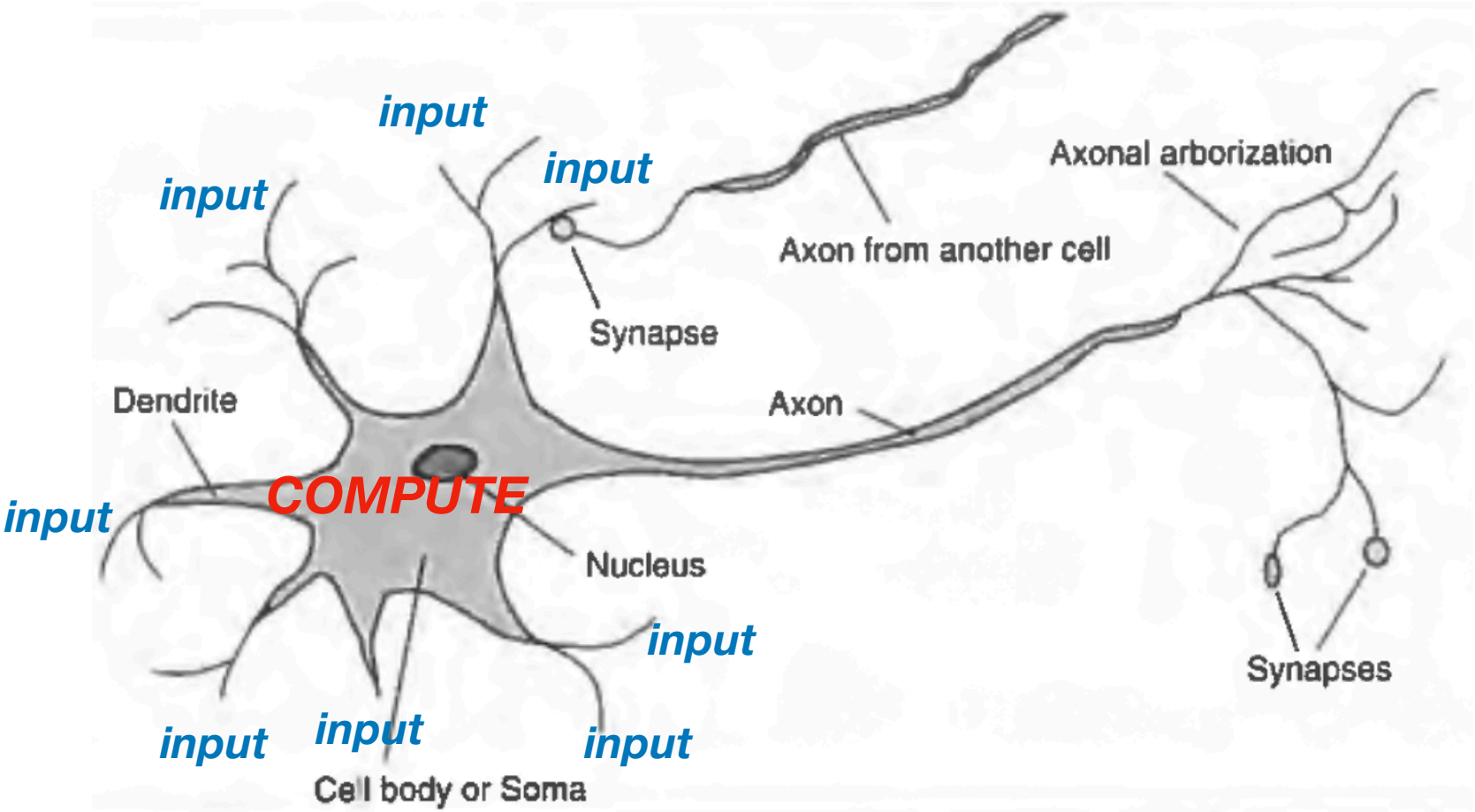
A brain-inspired model



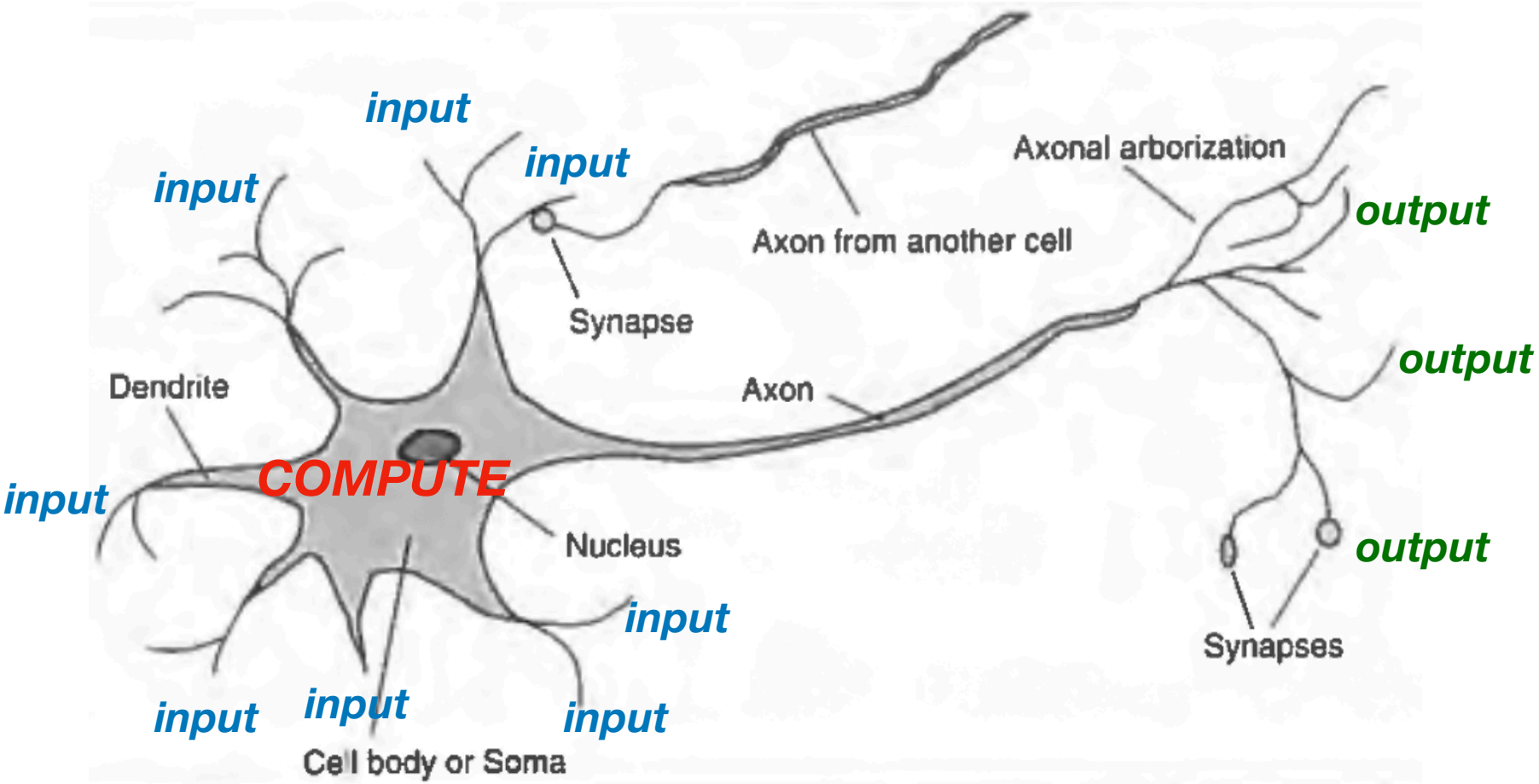
A brain-inspired model



A brain-inspired model



A brain-inspired model



Millions of those, connected in complicated ways – computing continually
Affected by other physiological phenomena

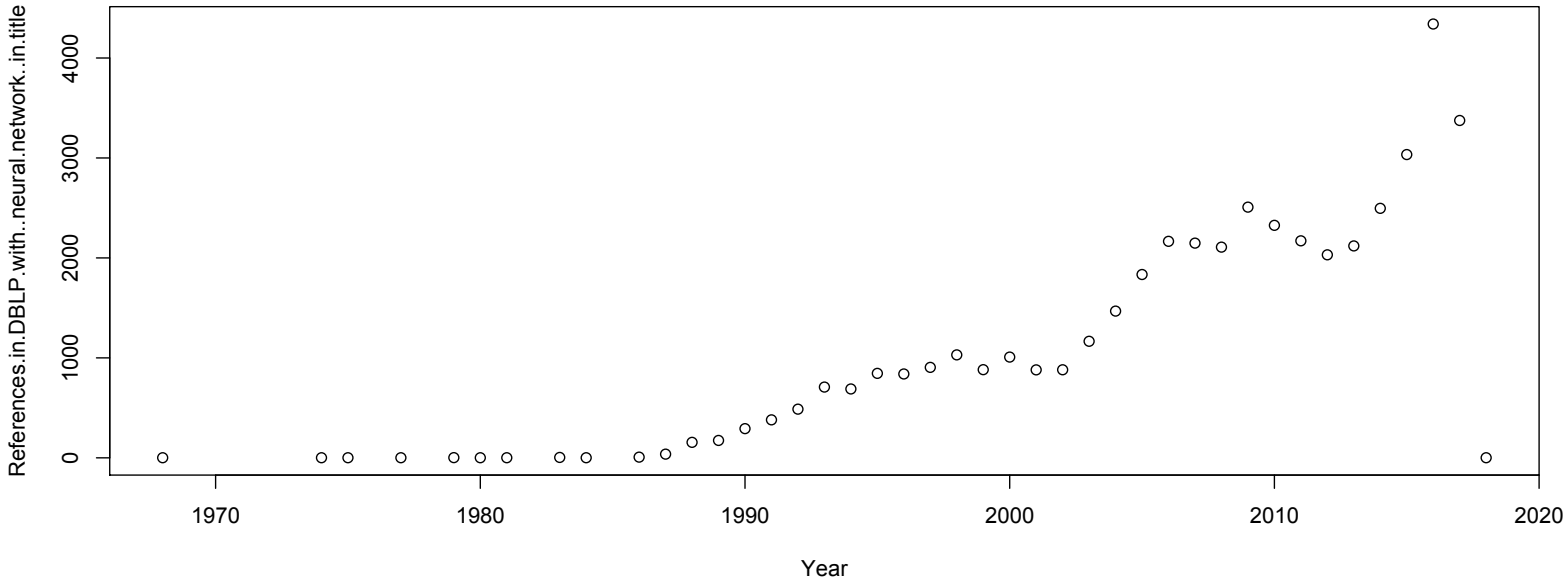
History, very briefly

- Psychology: Spencer (1872), James (1890), Freud (1895)
- Computational model: McCulloch, Pitts (1943)
- Ideas about learning: Hebb (late 1940s), Farley and Clark (1954), etc. etc.
- Backpropagation, the basic training algoritm: Werbos (1975)
- Deep learning: precursor by Schmidhuber (1992), Behnke (2003), Hinton et al. (2006) etc.

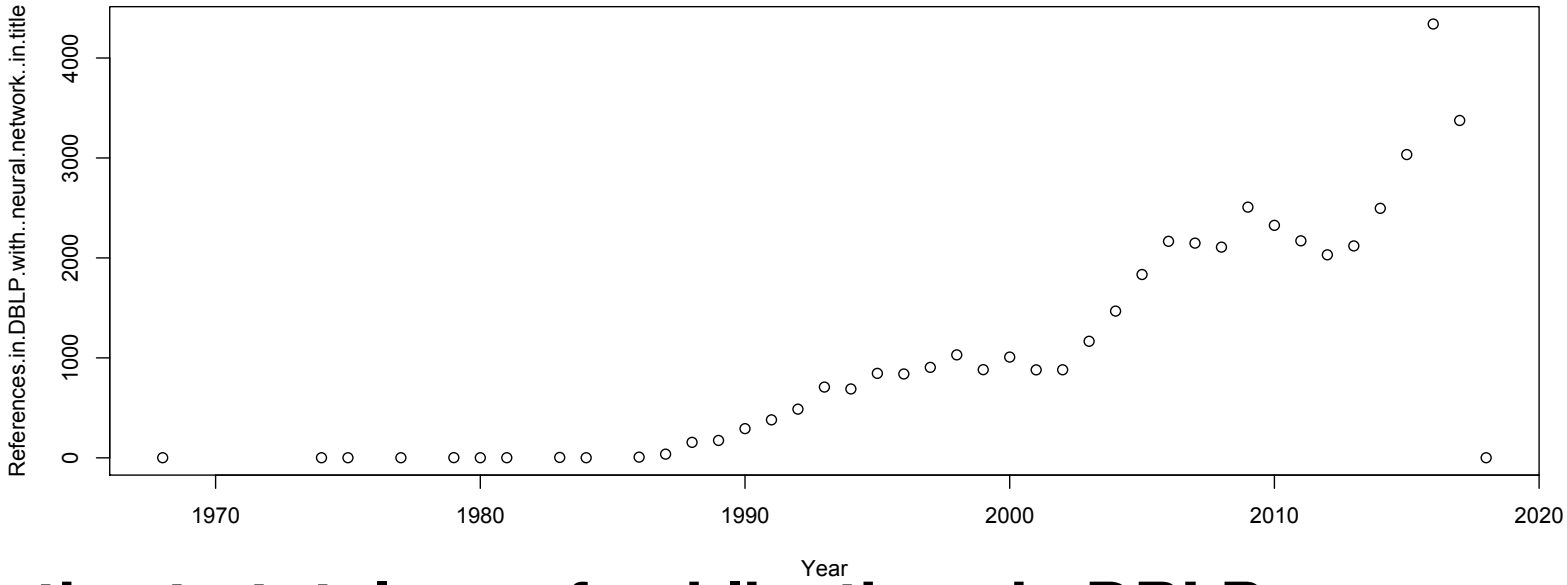
(source: wikipedia ;-)

- Some statistics about publications ....

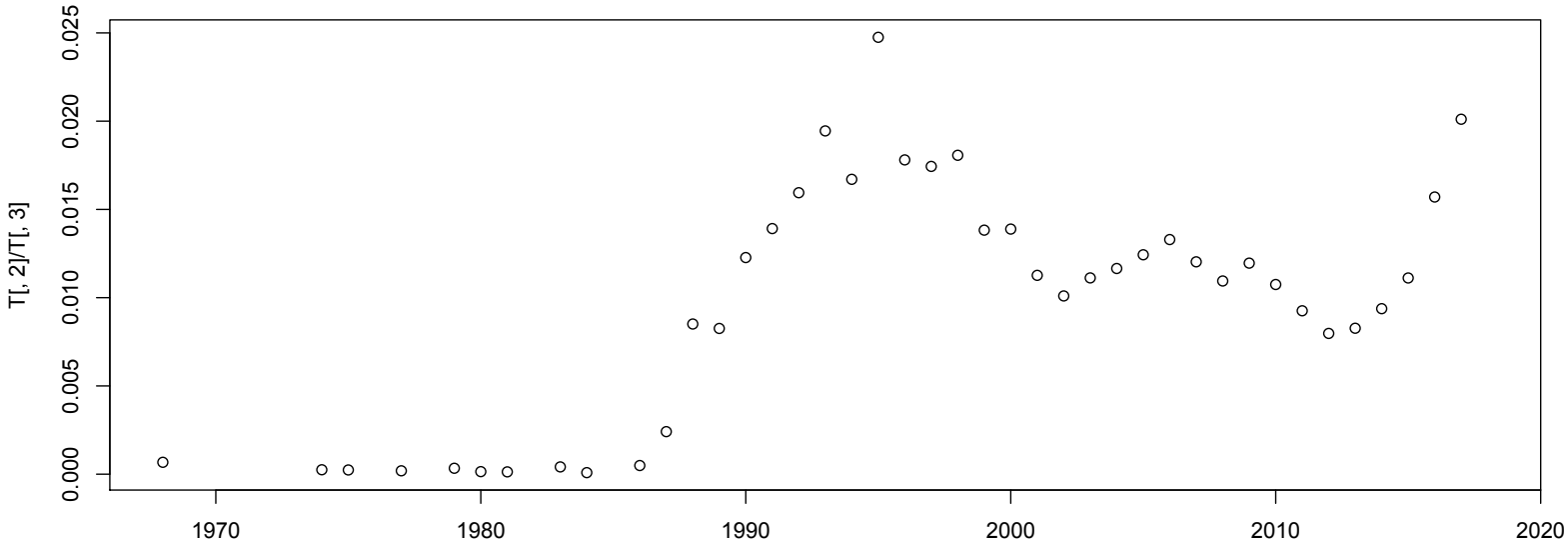
Publications in DBLP with 'neural networks' in title



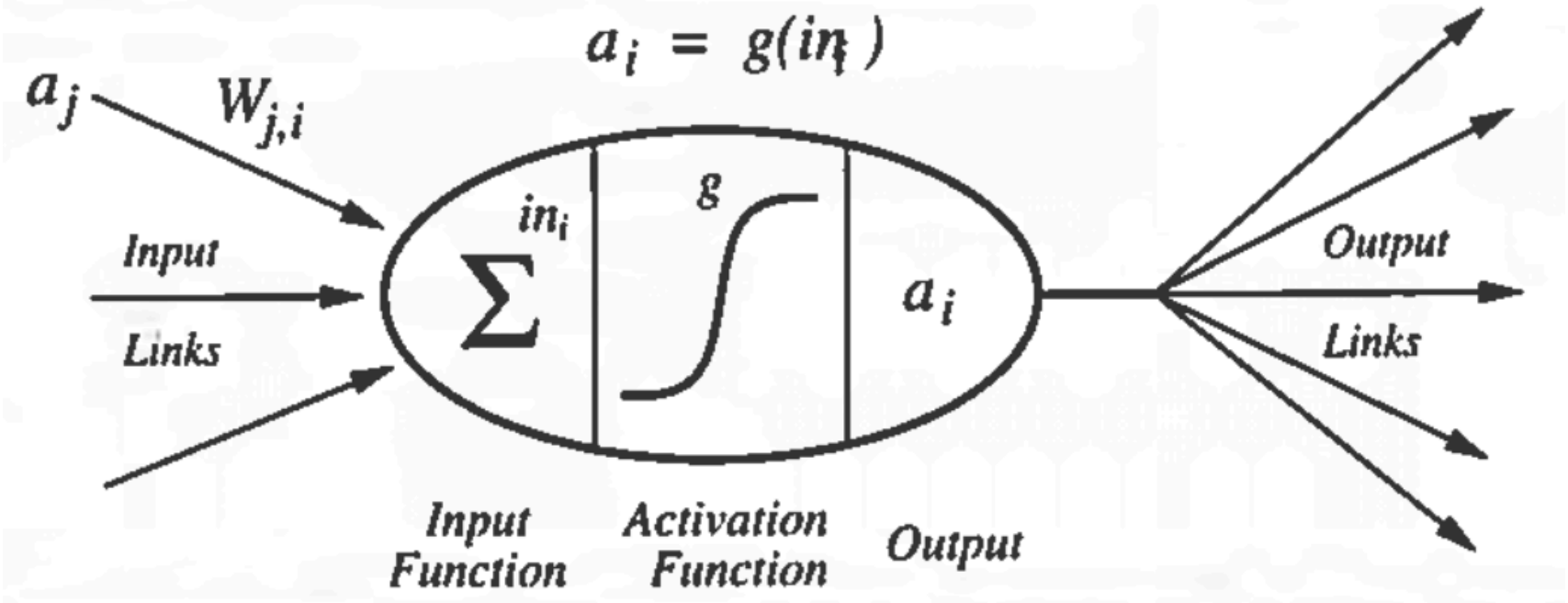
Publications in DBLP with 'neural networks' in title



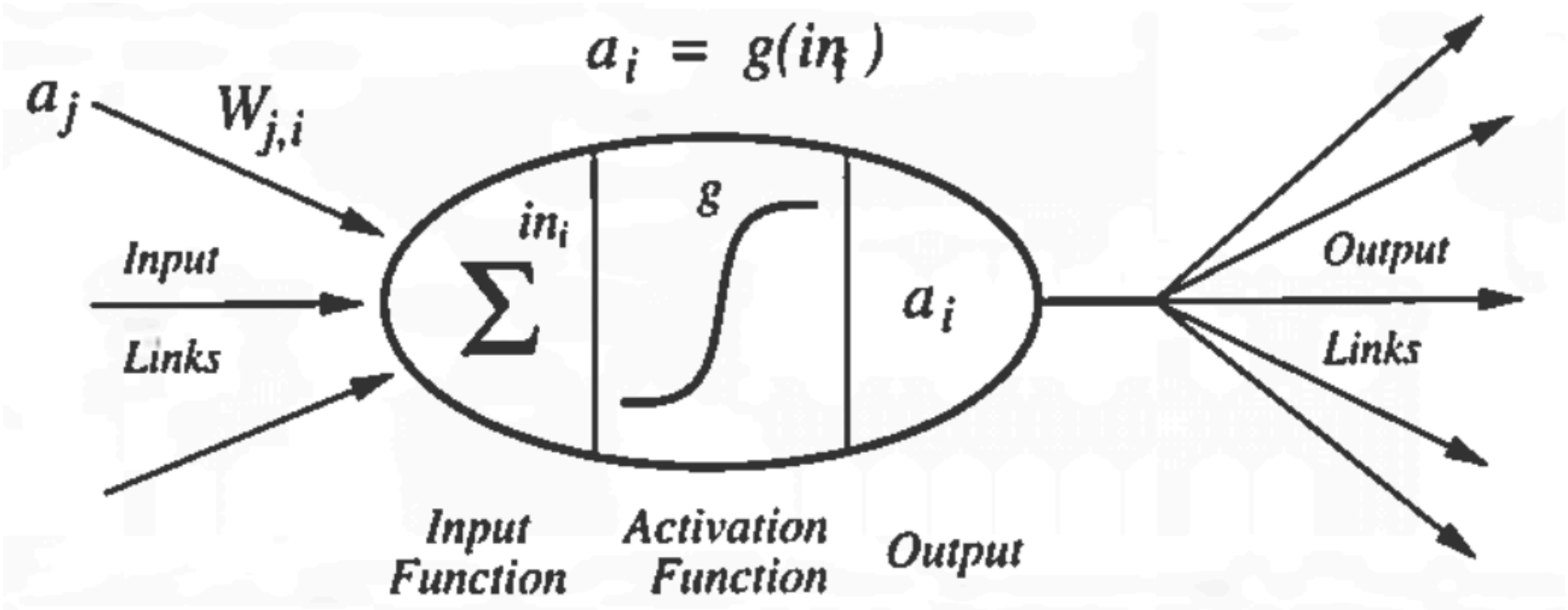
Relative to total no. of publications in DBLP



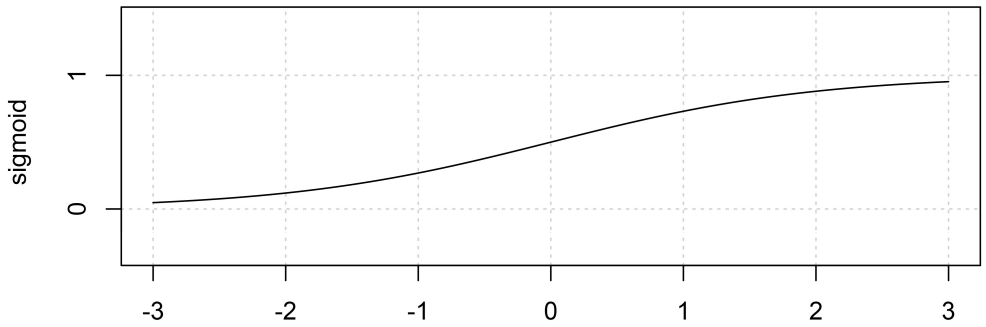
The computational version of "a neuron"



The computational version of "a neuron"

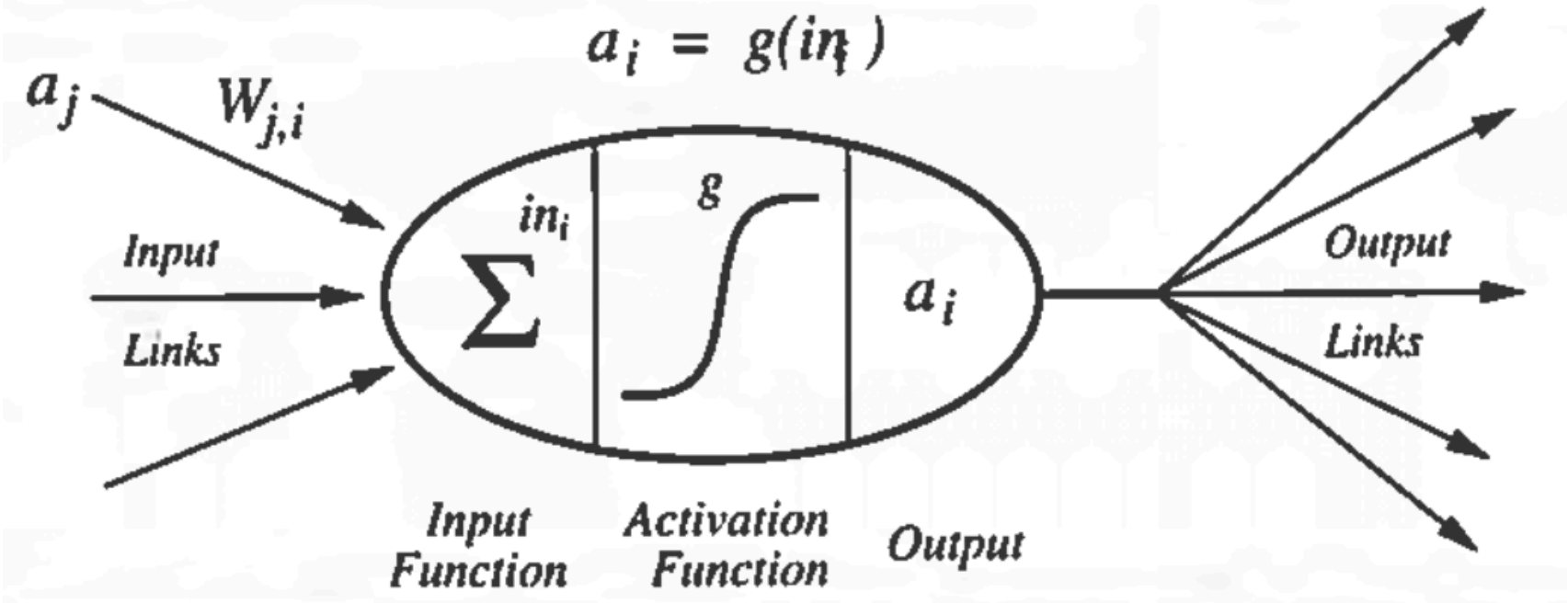


- Function "g" is a sigmoid (S-shaped), for example $1/(1+e^{-x})$

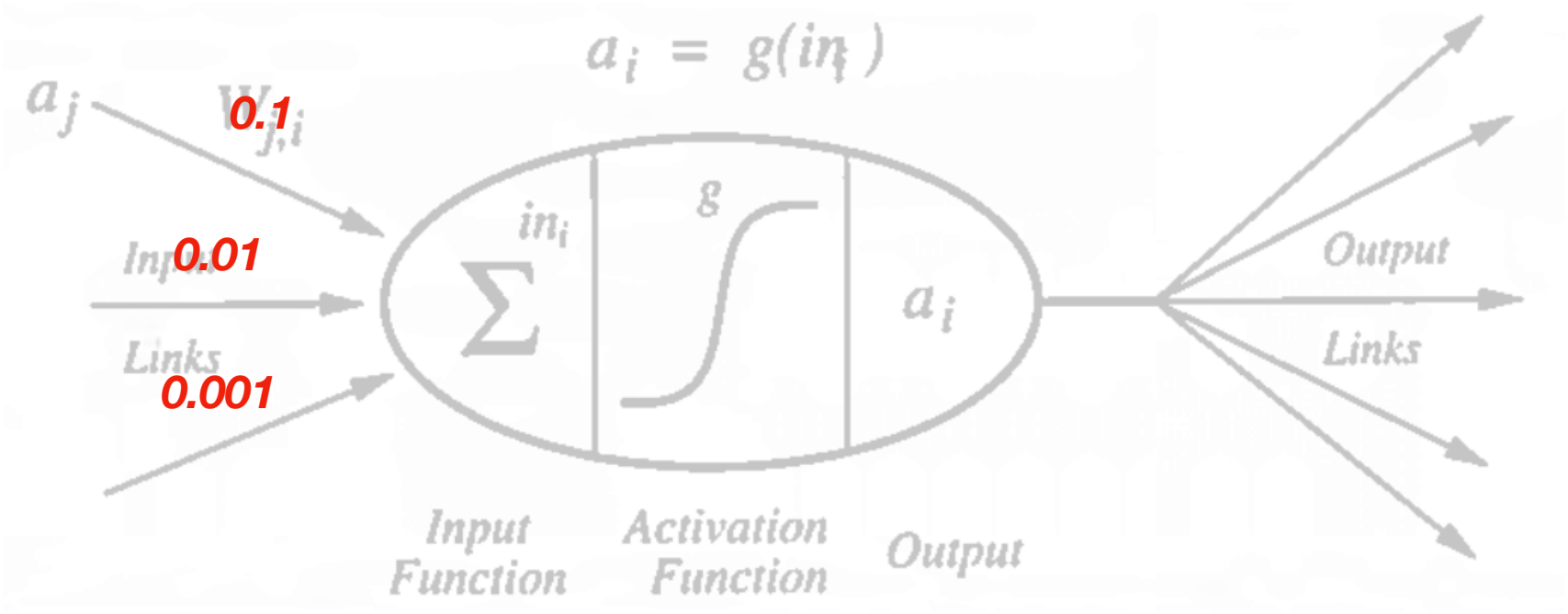


- Compresses the output interval

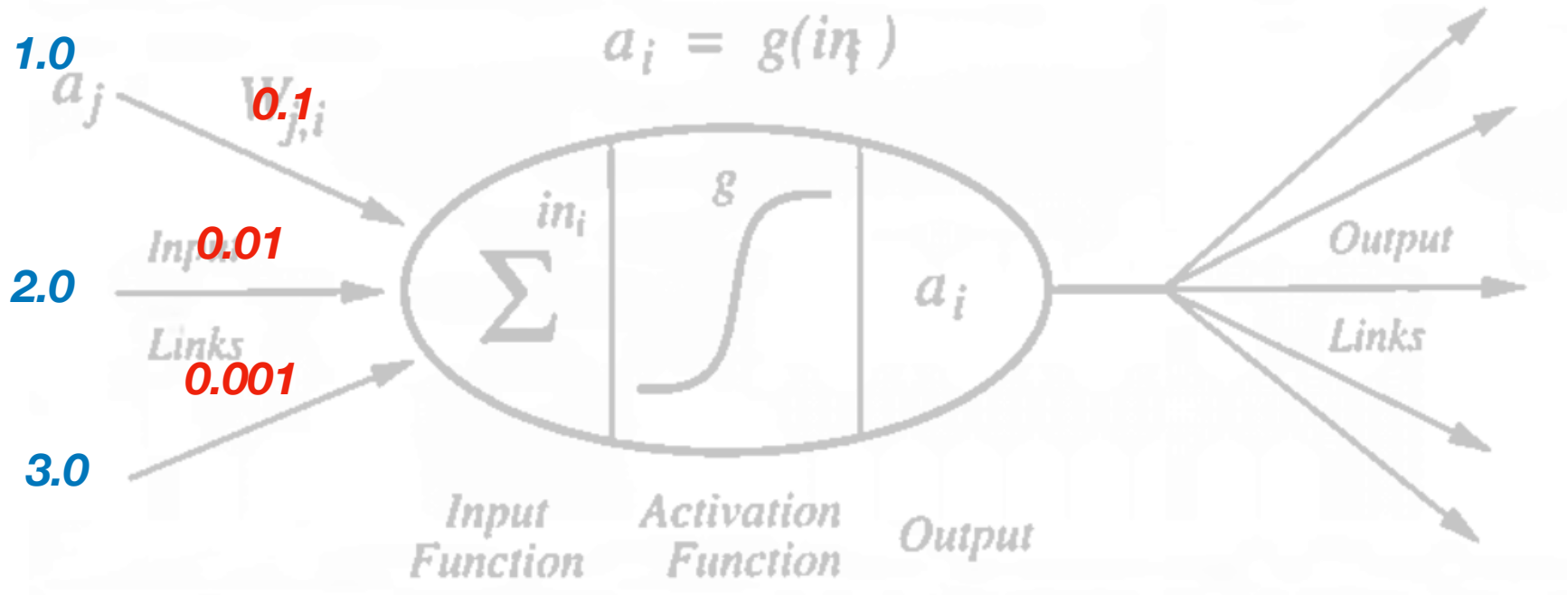
Example of evaluation a neuron



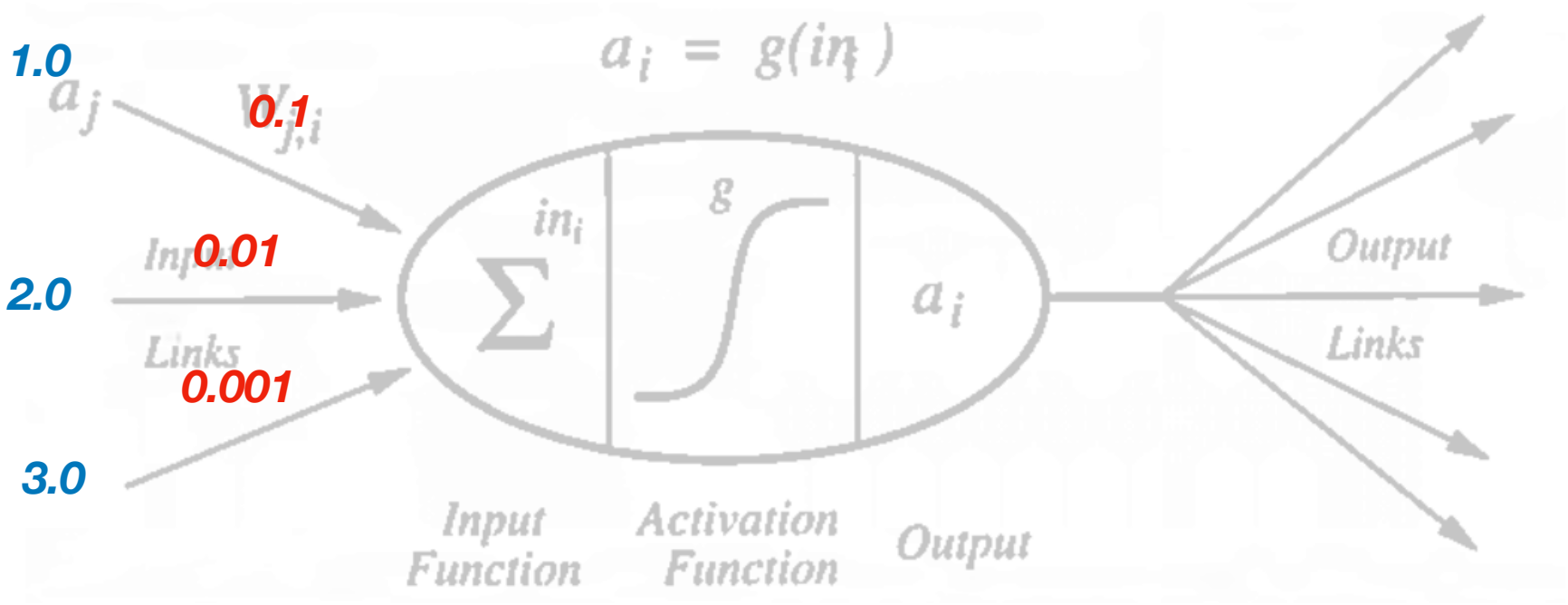
Example of evaluation a neuron



Example of evaluation a neuron

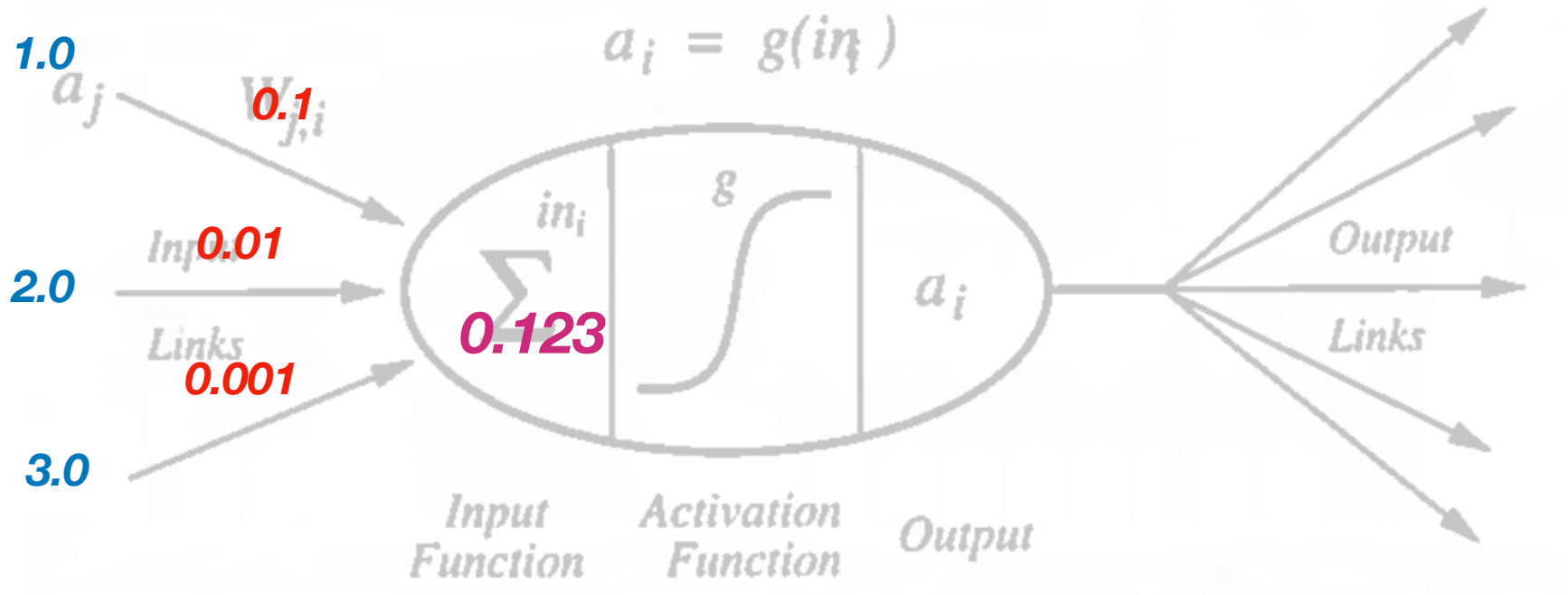


Example of evaluation a neuron



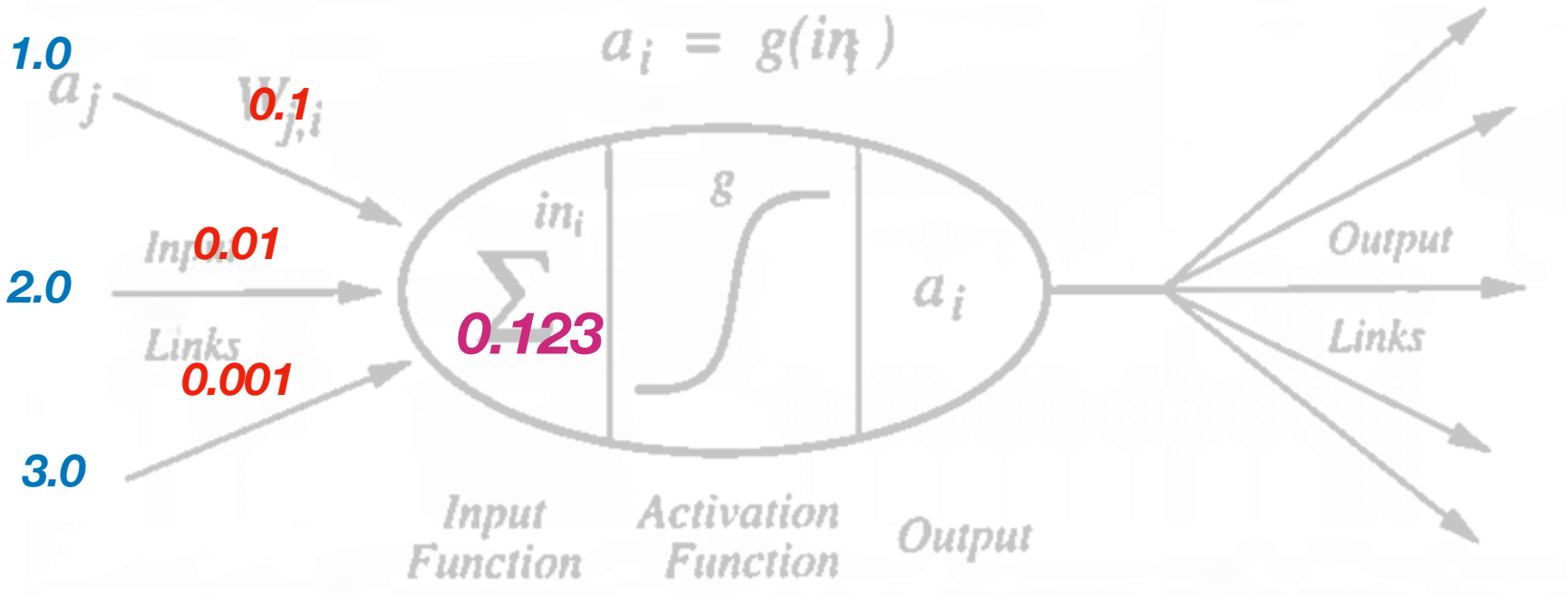
1.0 x 0.1 + 2.0 x 0.001 + 3.0 x 0.001 = 0.123

Example of evaluation a neuron



1.0 x 0.1 + 2.0 x 0.001 + 3.0 x 0.001 = 0.123

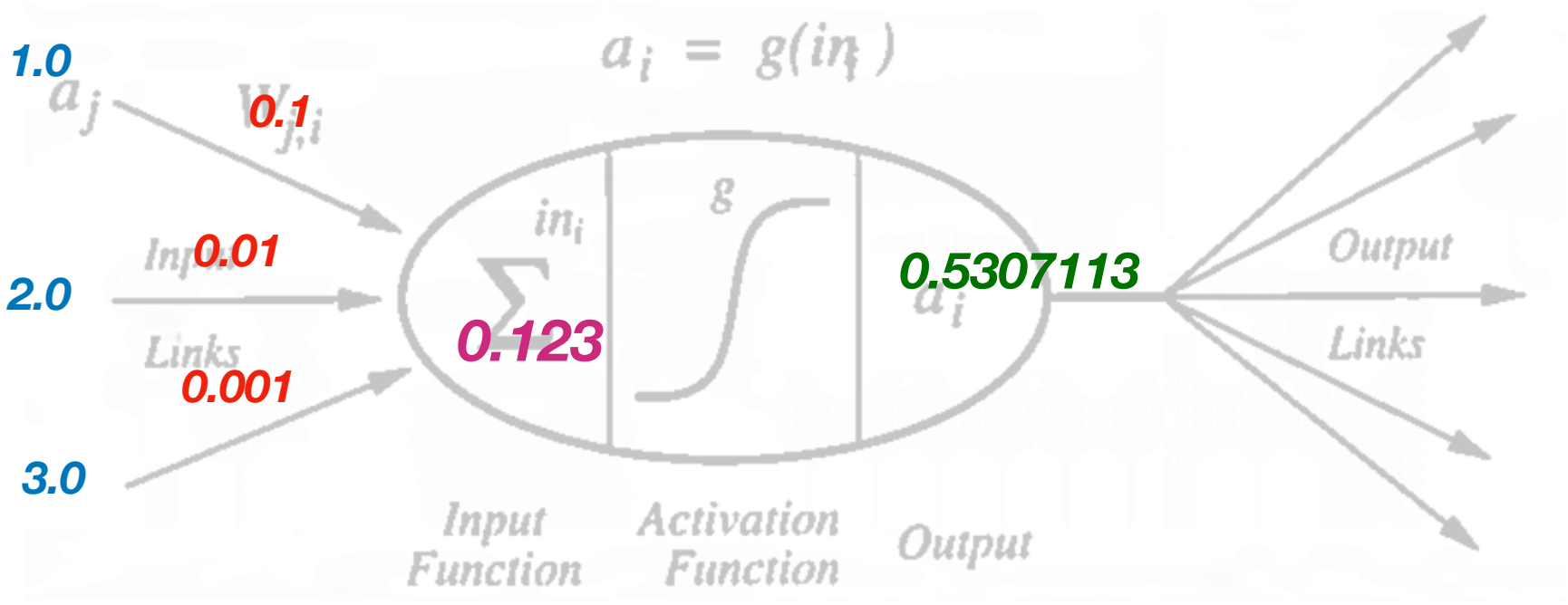
Example of evaluation a neuron



1.0 x 0.1 + 2.0 x 0.001 + 3.0 x 0.001 = 0.123

sigmoid(0.123) = $1/(1+e^{-0.123})$ = 0.5307113

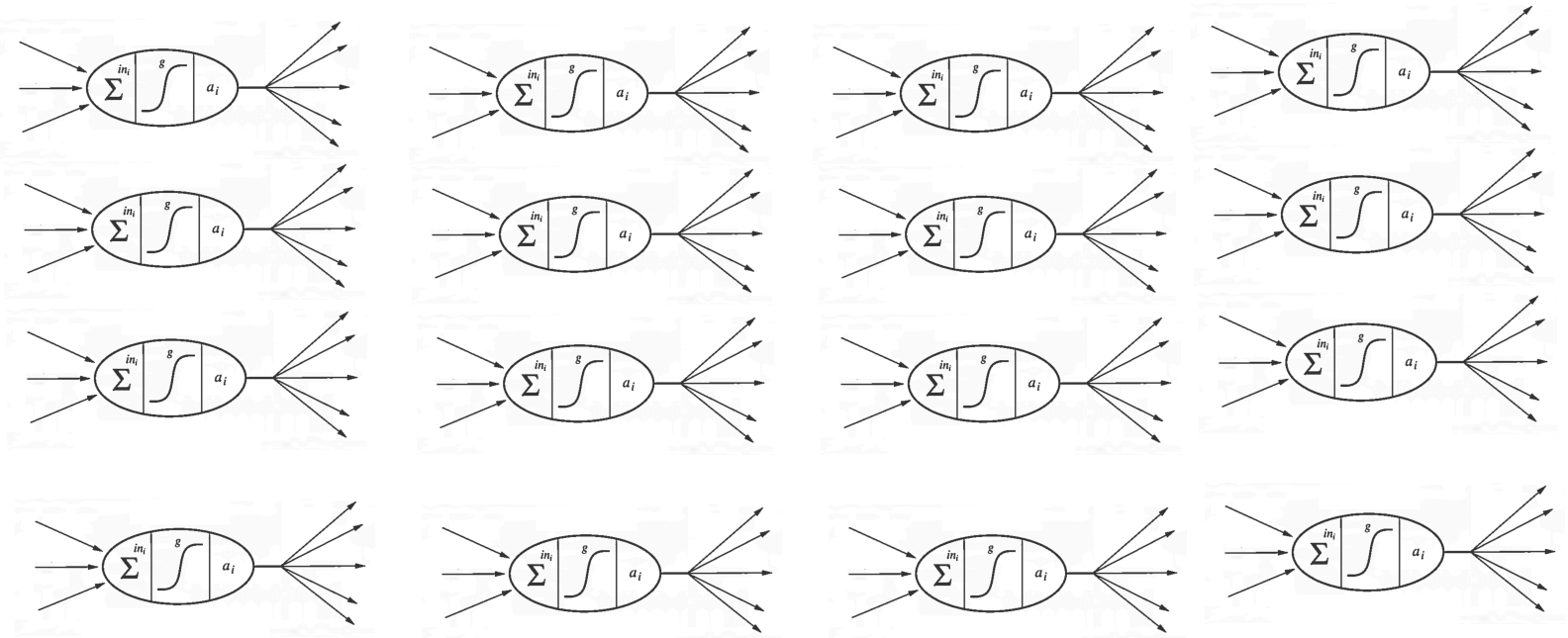
Example of evaluation a neuron



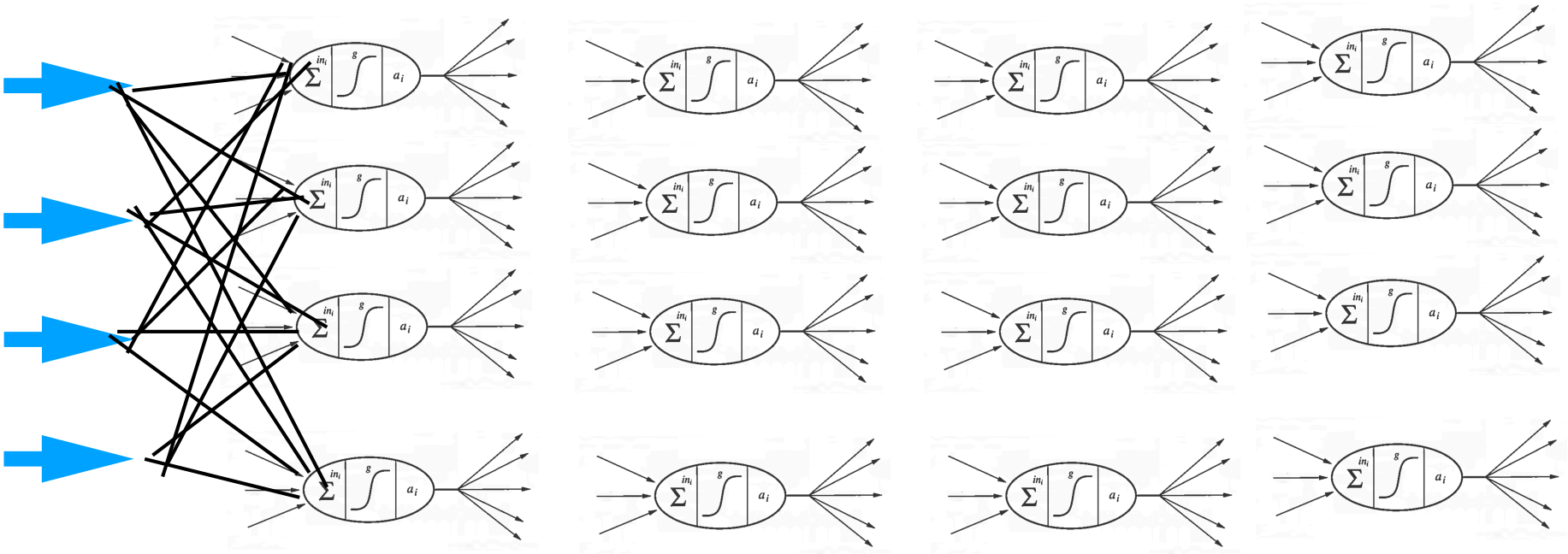
1.0 x 0.1 + 2.0 x 0.001 + 3.0 x 0.001 = 0.123

sigmoid(0.123) = $1/(1+e^{-0.123})$ = 0.5307113

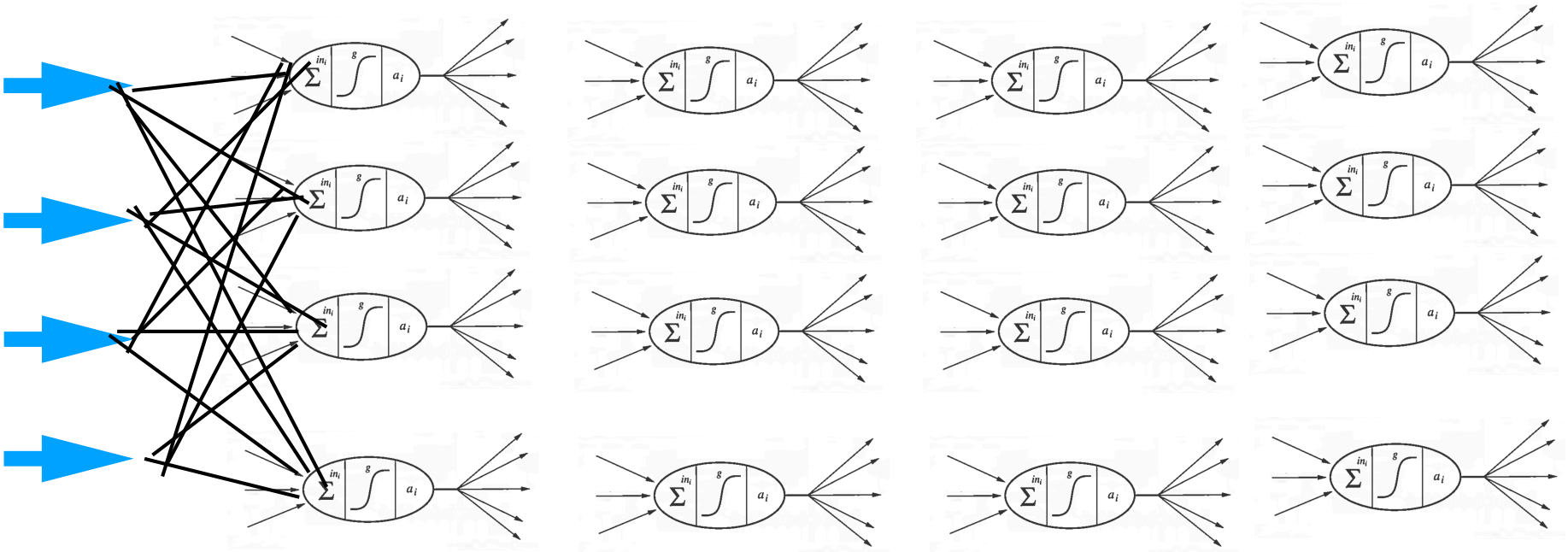
Putting them together in a network – typically organized in *layers*, evaluation as *feed-forward*



Putting them together in a network – typically organized in *layers*, evaluation as *feed-forward*

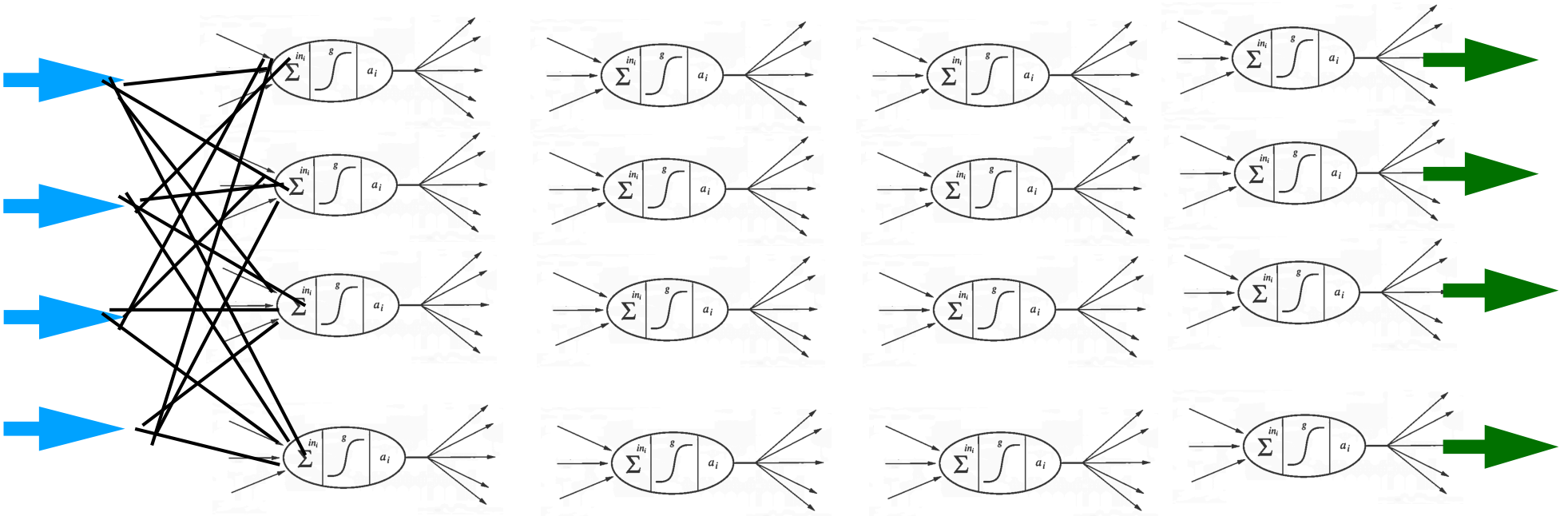


Putting them together in a network – typically organized in *layers*, evaluation as *feed-forward*



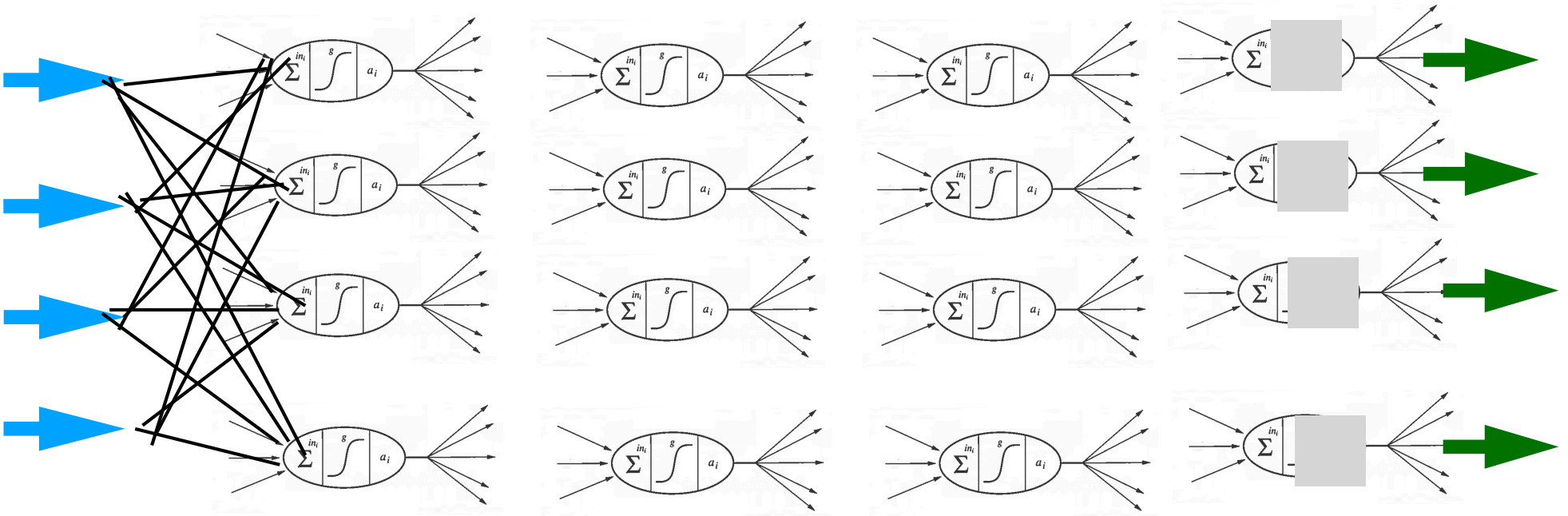
- typically all nodes in one layer connected to all in next layer

Putting them together in a network – typically organized in *layers*, evaluation as *feed-forward*



- typically all nodes in one layer connected to all in next layer

Putting them together in a network – typically organized in *layers*, evaluation as *feed-forward*



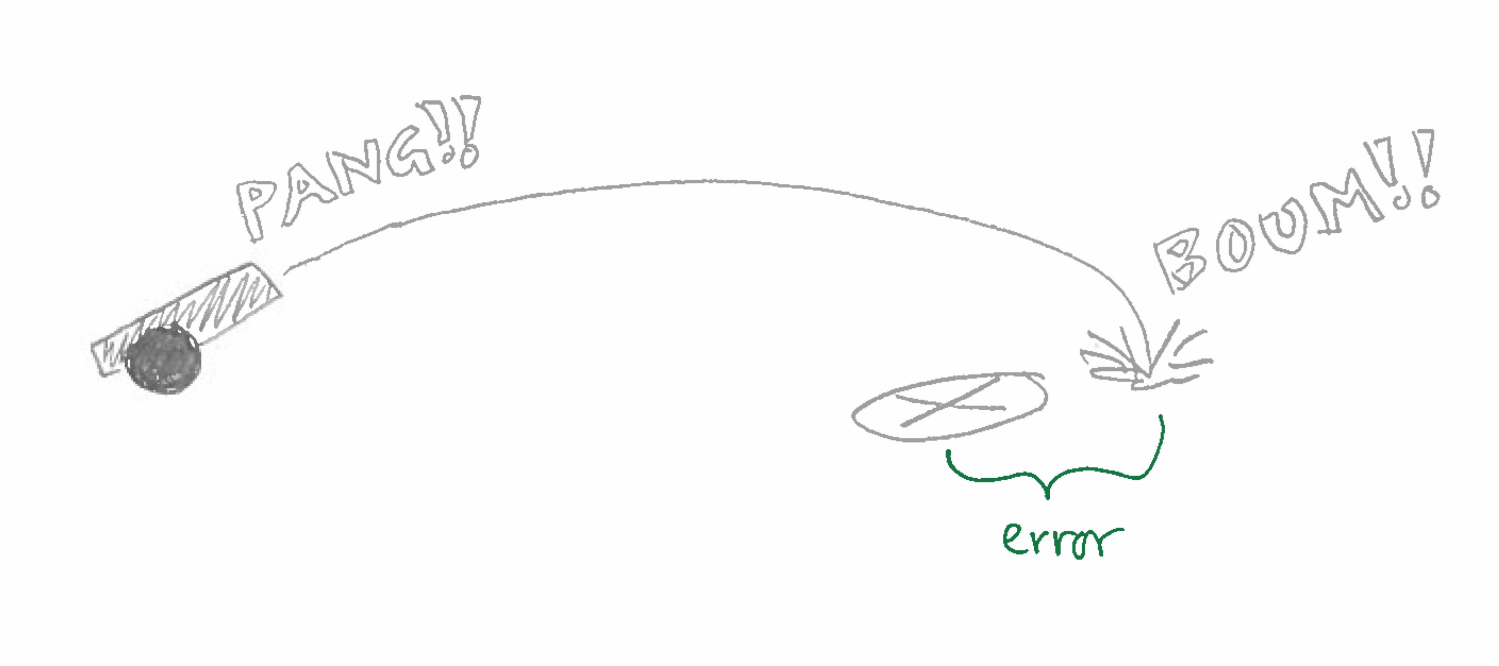
- typically all nodes in one layer connected to all in next layer
- drop the sigmod for the final output

Back-propagation – the intuition

- Desired behaviour vs. actual behaviour

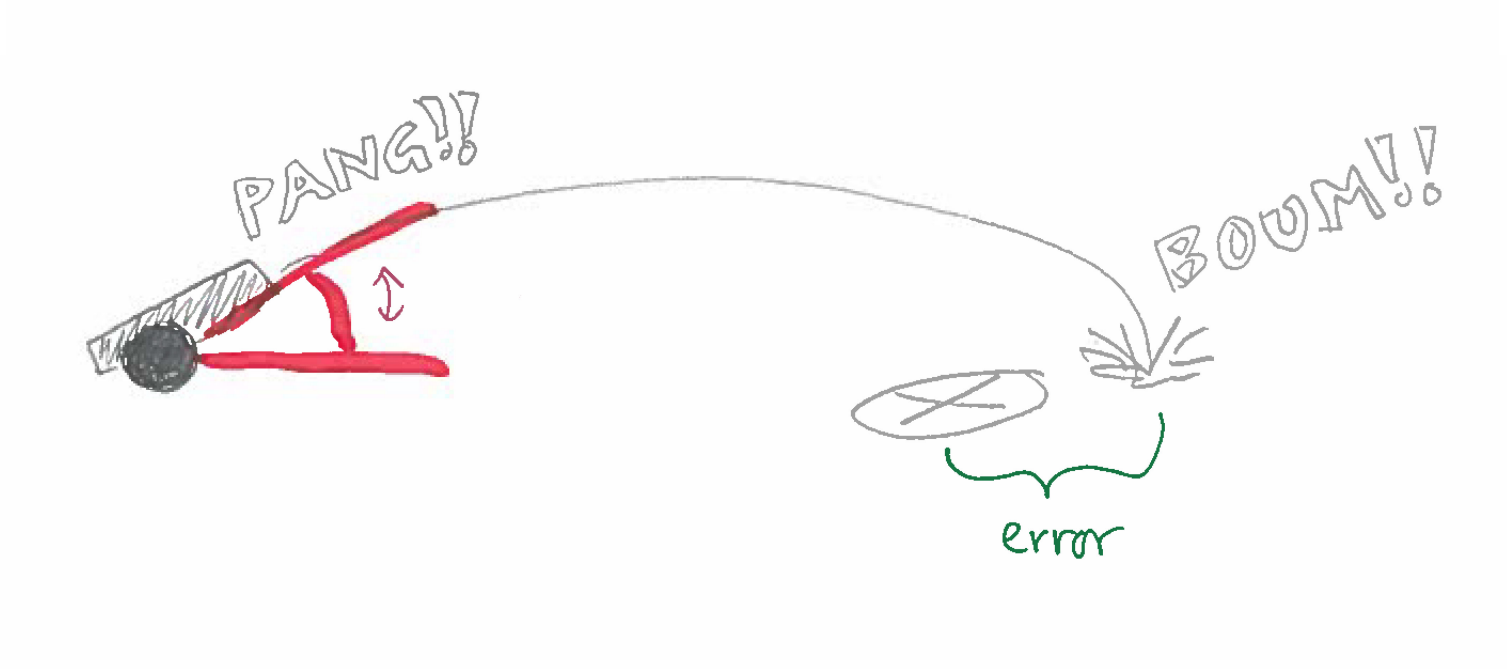
Back-propagation – the intuition

- Desired behaviour vs. actual behaviour



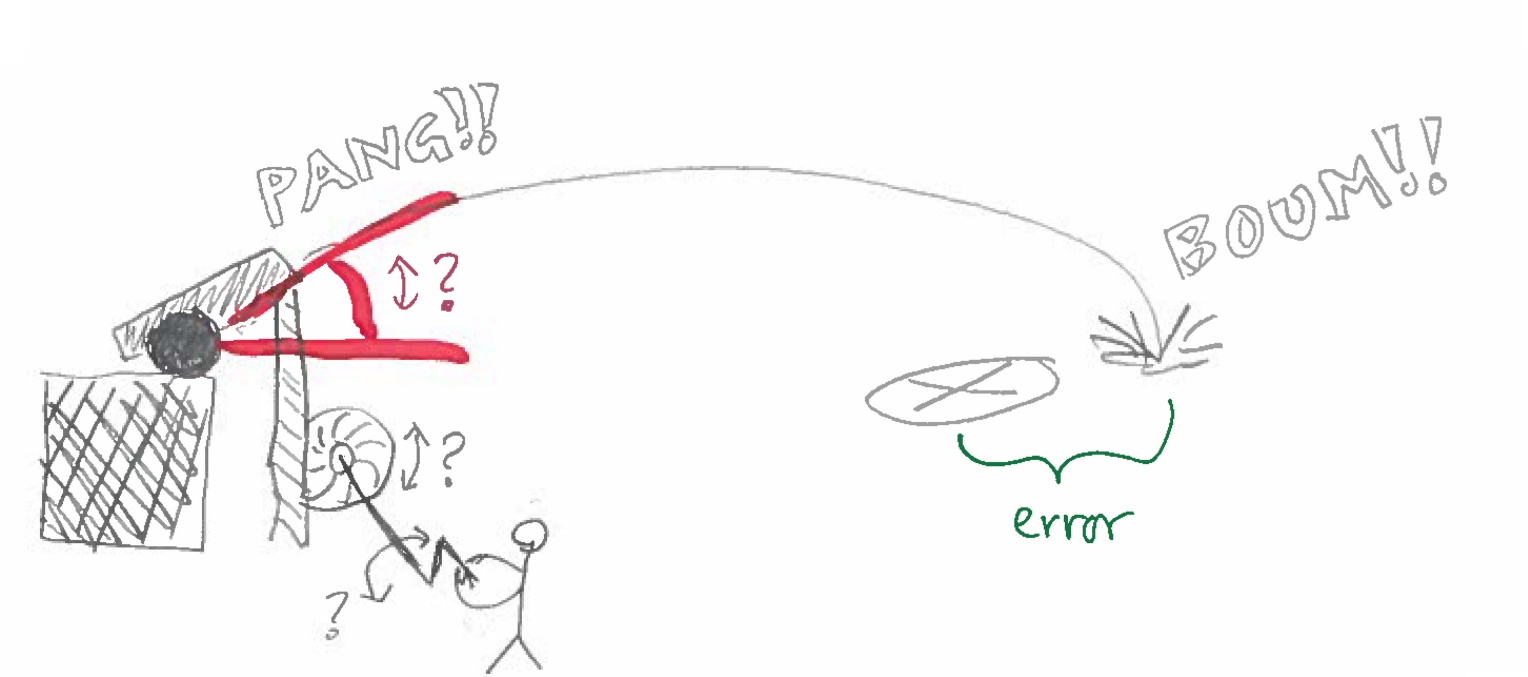
Back-propagation – the intuition

- Desired behaviour vs. actual behaviour



Back-propagation – the intuition

- Desired behaviour vs. actual behaviour



- moving the error "backwards" in the system...

Back-propagation – the algorithm

- It is simple, but easy to get confused (i.e., it is not so easy)
- Here we will understand it mechanically
- Do not try do understand why it is as it is, or why it works!!
- You will understand its mechanics better, when you have implemented it (exercise)

Very abstract version; only one hidden layer

function BACK-PROP-UPDATE(network, examples, α) **returns** a network with modified weights
inputs: *network*, *examples*, the learning rate α

repeat {
for each $e = \langle \mathbf{I}^e, \mathbf{T}^e \rangle$ **in** examples **do** {
/* *Compute the output e* */
$\mathbf{O}^e \leftarrow$ RUN-NETWORK$(network, \mathbf{I}^e)$

/* *Compute the error and* Δ *for units in the output layer* */
$\mathbf{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}^e$

/* *Update the weights leading to the output layer* */
for each *unit i* **in** *the output layer* **and** *unit j in the hidden layer* **do** $W_{j,i} \leftarrow W_{j,i} + \cdots$

/* *Update weights from input layer to hidden layer* */

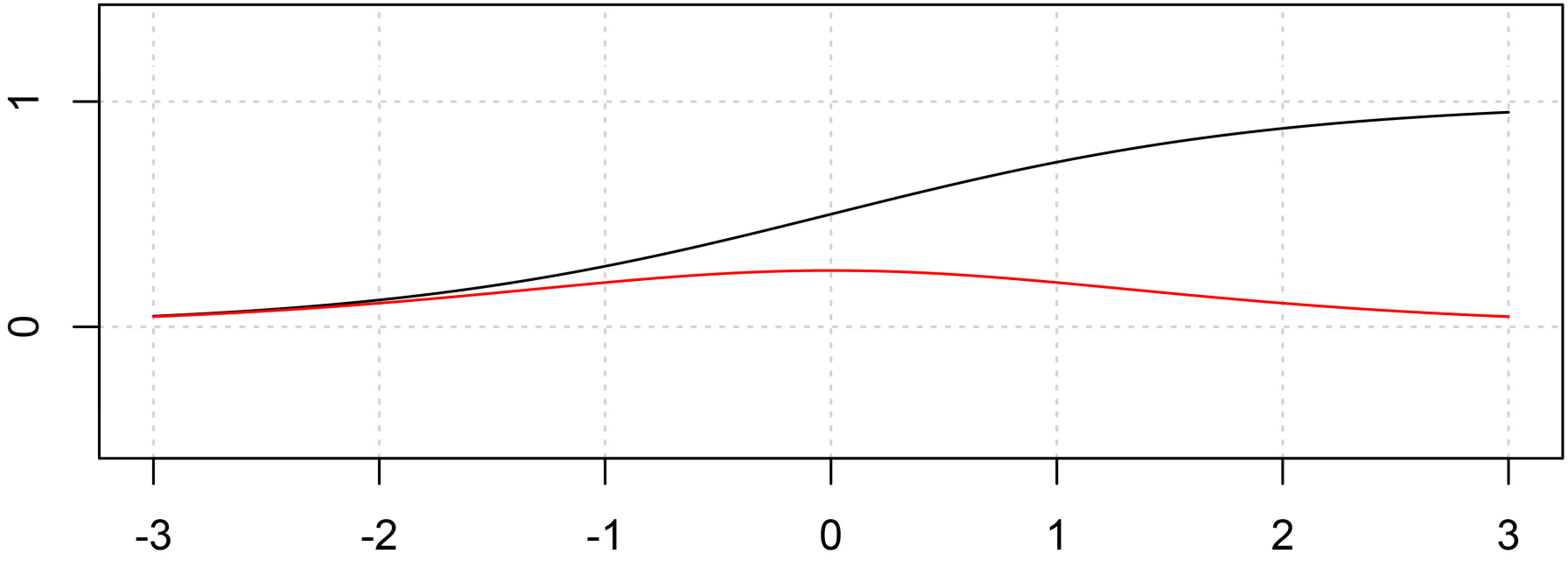
for each *node j* **in** *hidden layer* **do** {
$\cdots$

/* *Update weights from input layer to hidden layer* */
for each *unit k in input layer* **do**
$W_{k,j} \leftarrow W_{k,j} + \cdots$
} /* *end: for each node j ...* */

} /* *end: for each e ...* */
} /* *end: repeat* */
until *stop-condition*

For the details, we need ...

- Learning rate alpha; between 0 and 1; 0.2 often seen
- Delta$_i$ – a portion of the error associated with node *i*
- The sigmoid and its derivative (slope; differentiated); referred to as *g* and *g*'



With the details ...

function BACK-PROP-UPDATE(network, examples, α) **returns** a network with modified weights

inputs: *network*, *examples*, the learning rate α

repeat {

for each $e = \langle \mathbf{I}^e, \mathbf{T}^e \rangle$ **in** examples **do** {

/ Compute the output for this example, storing values of "in_i" and "a_i" for each node */*

$\mathbf{O}^e \leftarrow$ RUN-NETWORK$(network, \mathbf{I}^e)$

/ Compute the error and Δ for units in the output layer */*

$\mathbf{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}^e$

for each *unit i* **in** *the output layer* **do** $\Delta_i \leftarrow Err_i^e \times g'(in_i)$

/ Update the weights leading to the output layer*/*

for each *unit i* **in** *the output layer* **and** *unit j in the hidden layer* **do** $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$

/ Update the weights from input to hidden layers */*

/ Compute the error term at each node */*

for each *node j* **in** *hidden layer* **do** {

$\Delta_j \leftarrow g'(in_j) \times \sum_i W_{j,i} \Delta_i$, *i running over all nodes in the output layer*

/ Update weights from input layer to hidden layer */*

for each *unit k in input layer* **do**

$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$

} */* end: for each node j ... */*

} */* end: for each e ... */*

} */* end: repeat */*

until *stop-condition*

An animation of how *one* tuple is processed

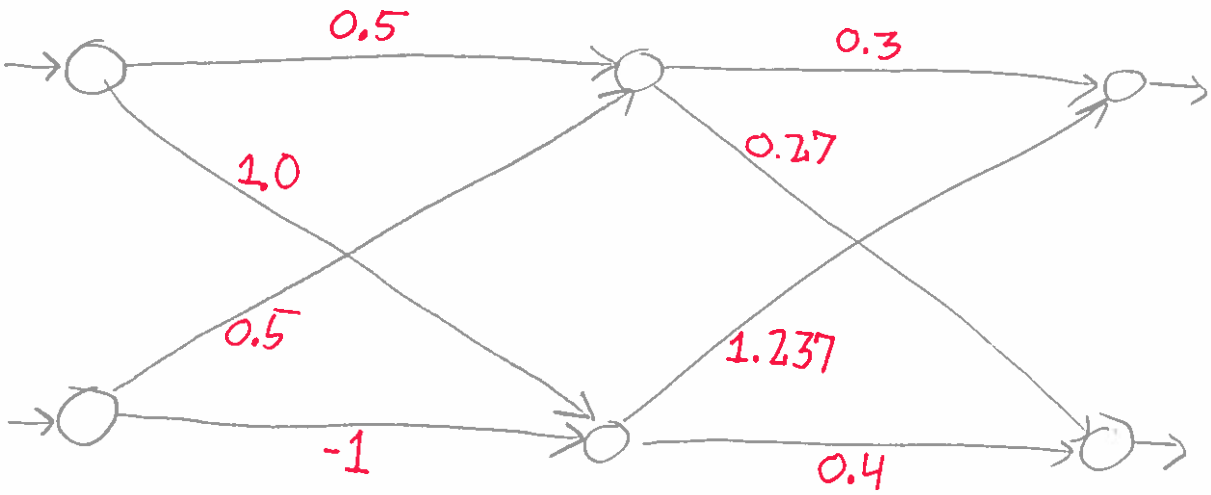
- 2 input nodes
- 1 hidden layer with 2 hidden nodes
- 2 output nodes
- alpha = 0.5

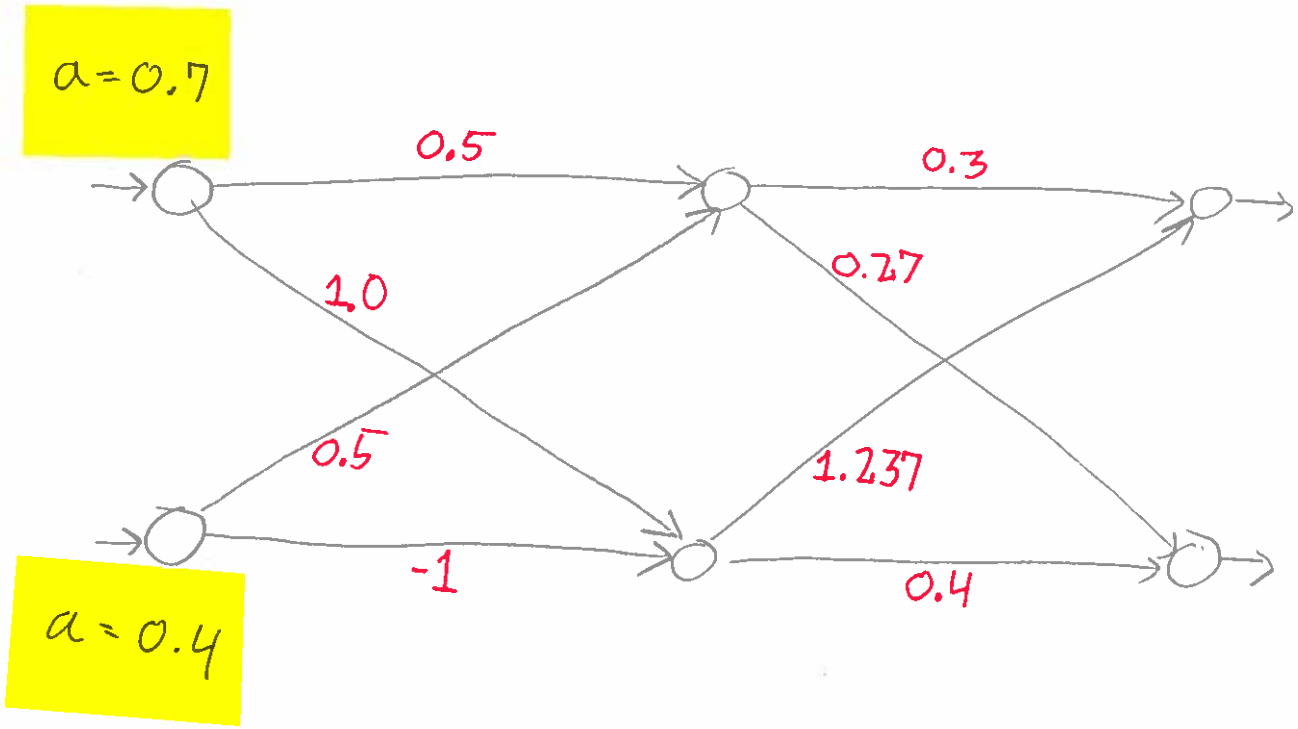
- Assume we are trying to learn the function

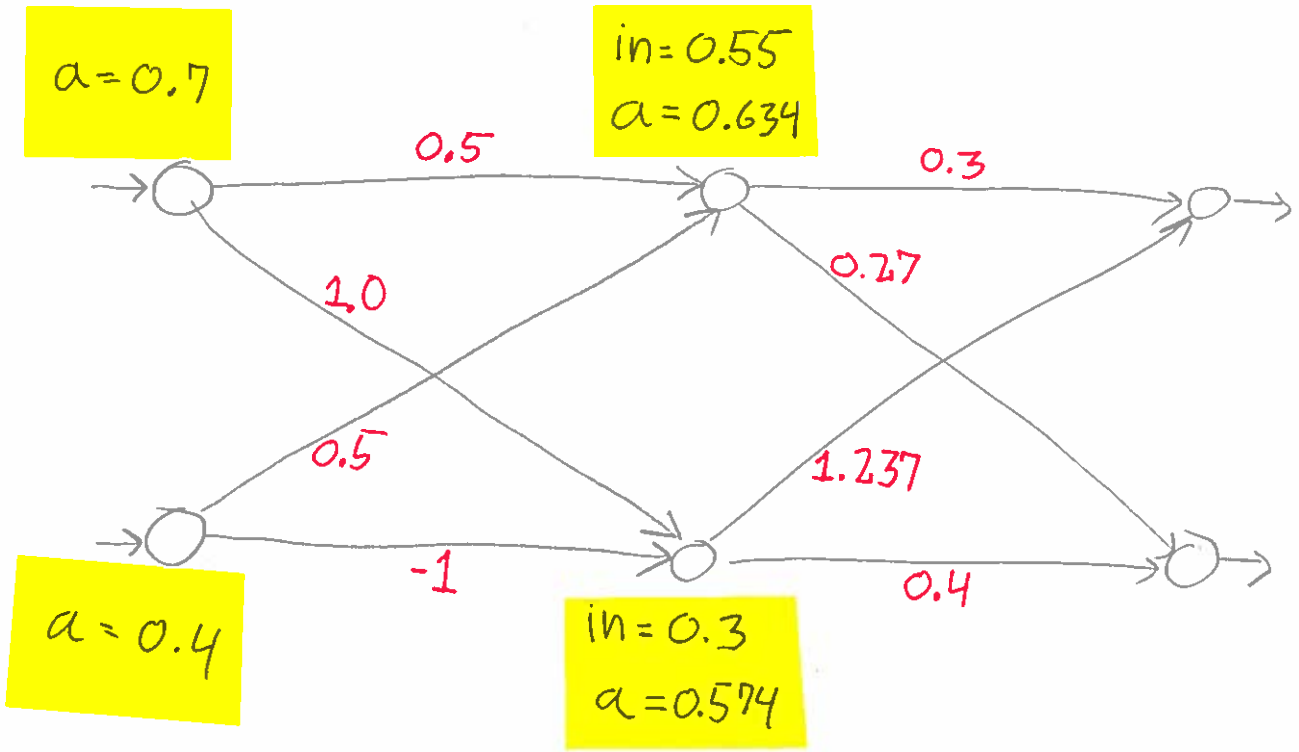
  <x, y> --> <x+y, x-y>

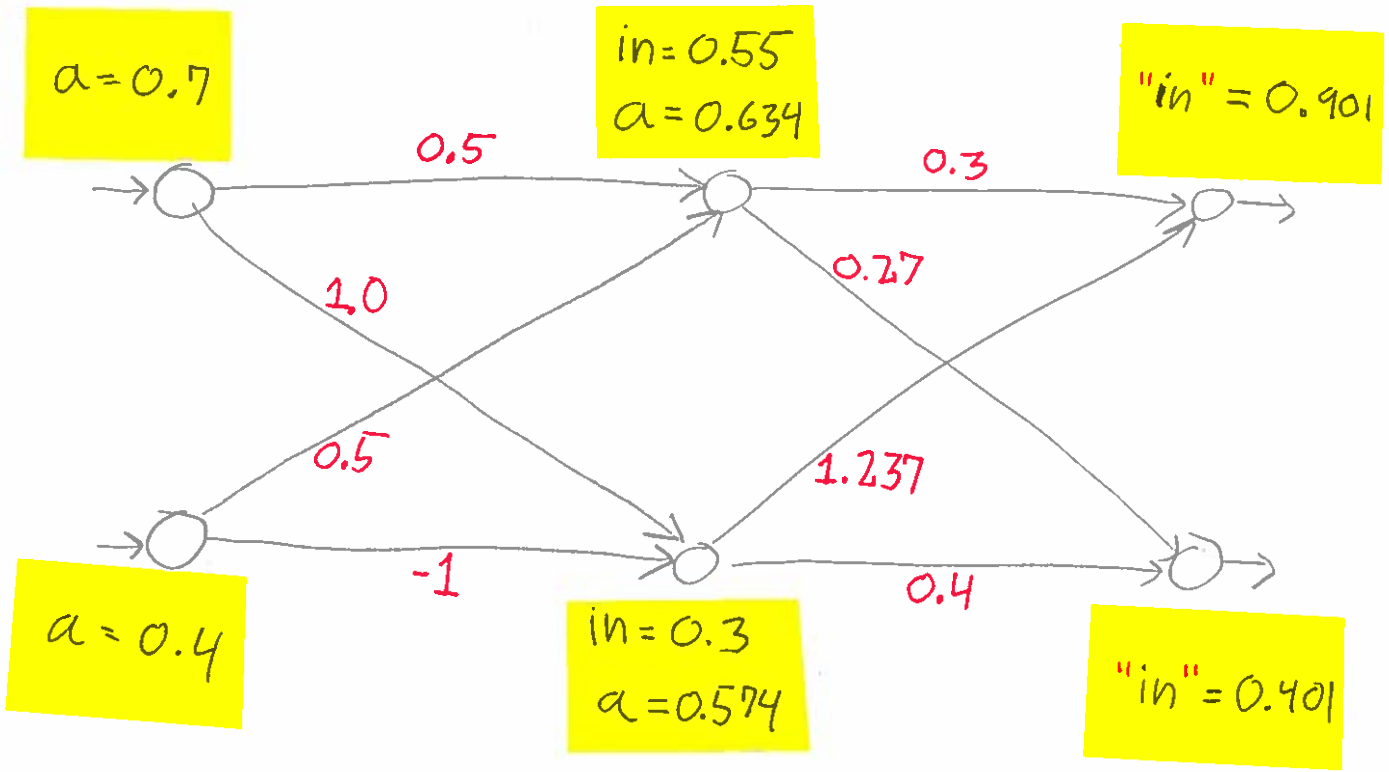
- and we consider this particular training tuple

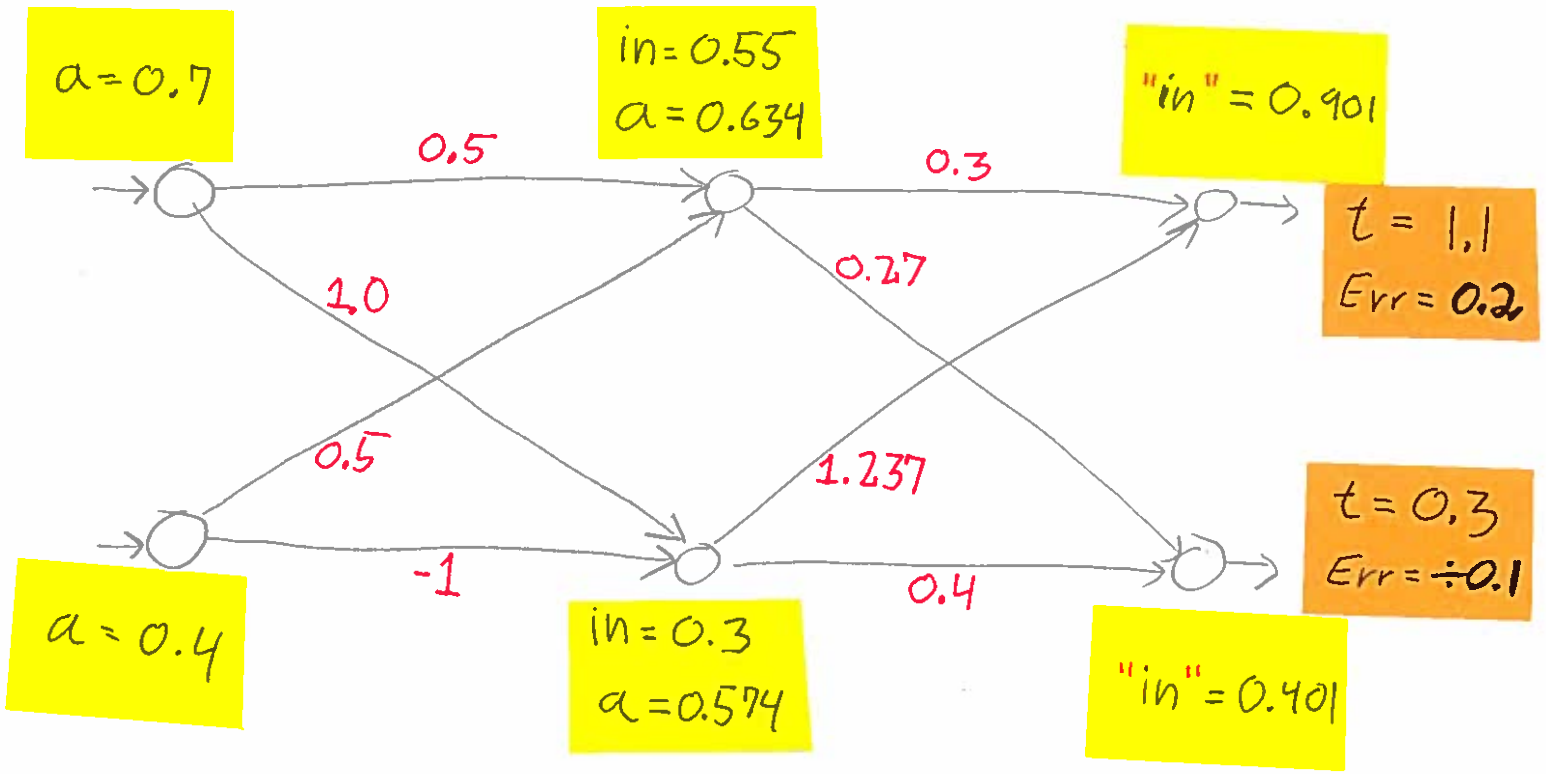
  <<0.7, 0.4>, <1.1, 0.3>>

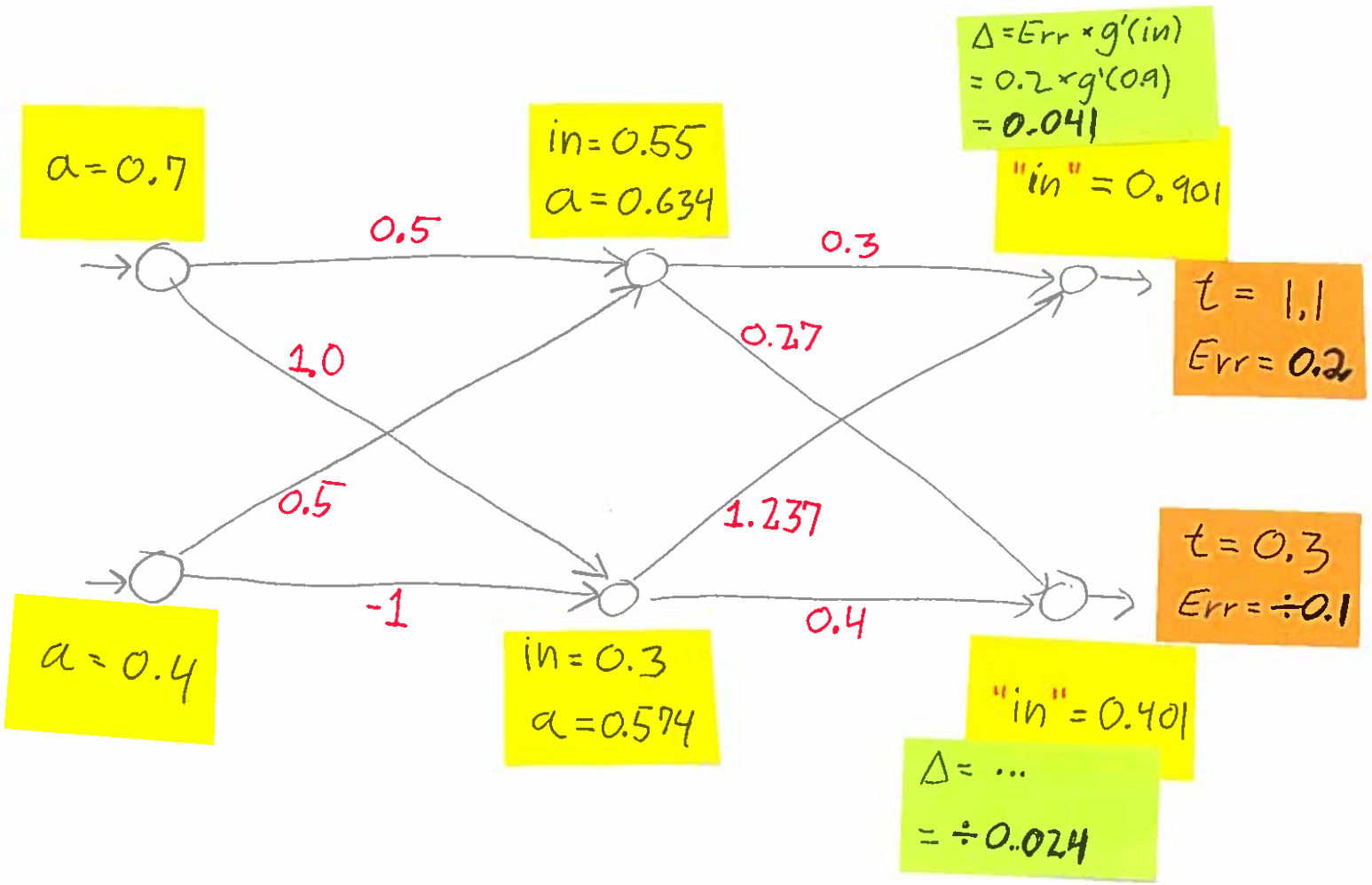

0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4

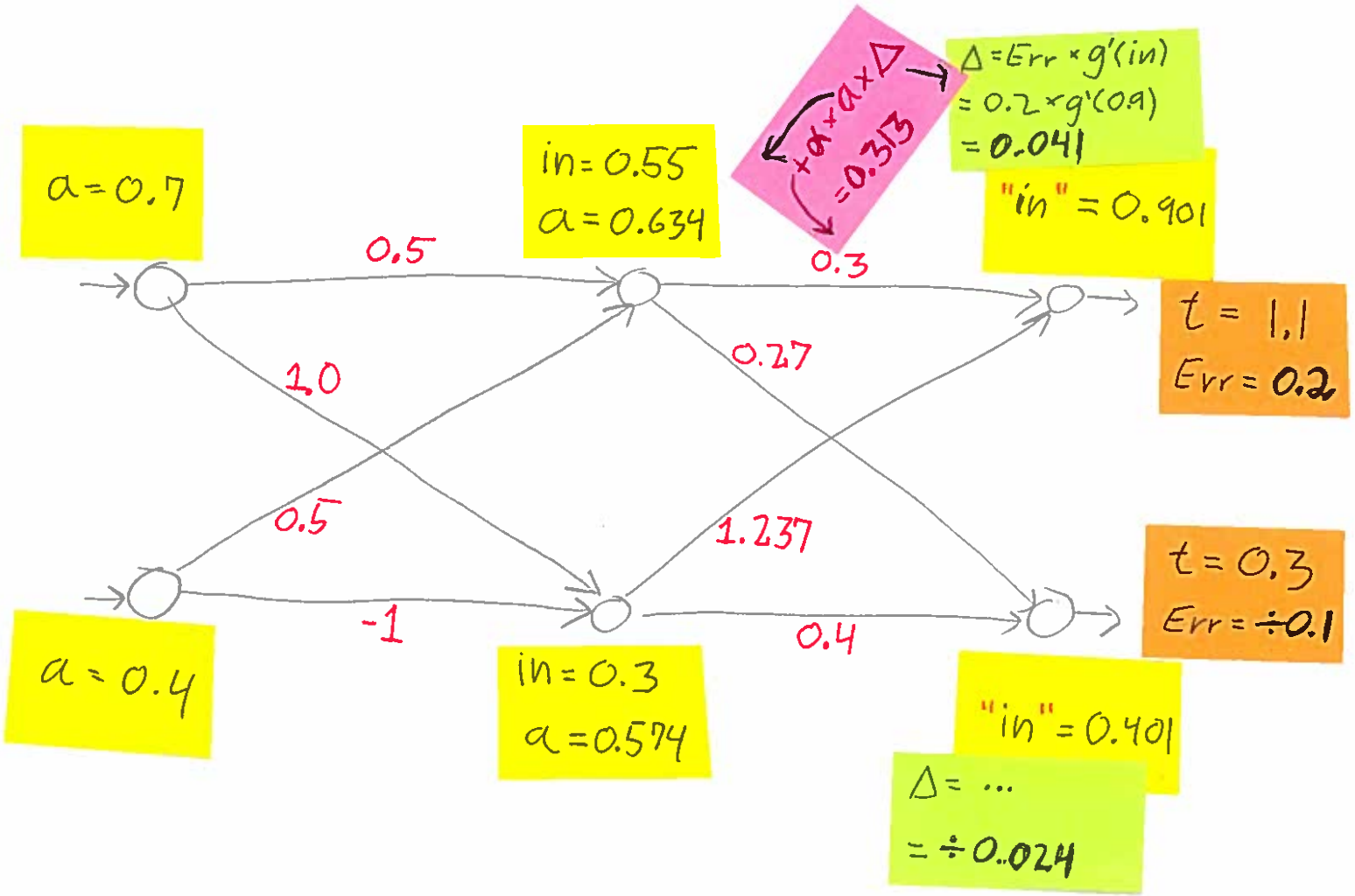

a = 0.7
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4

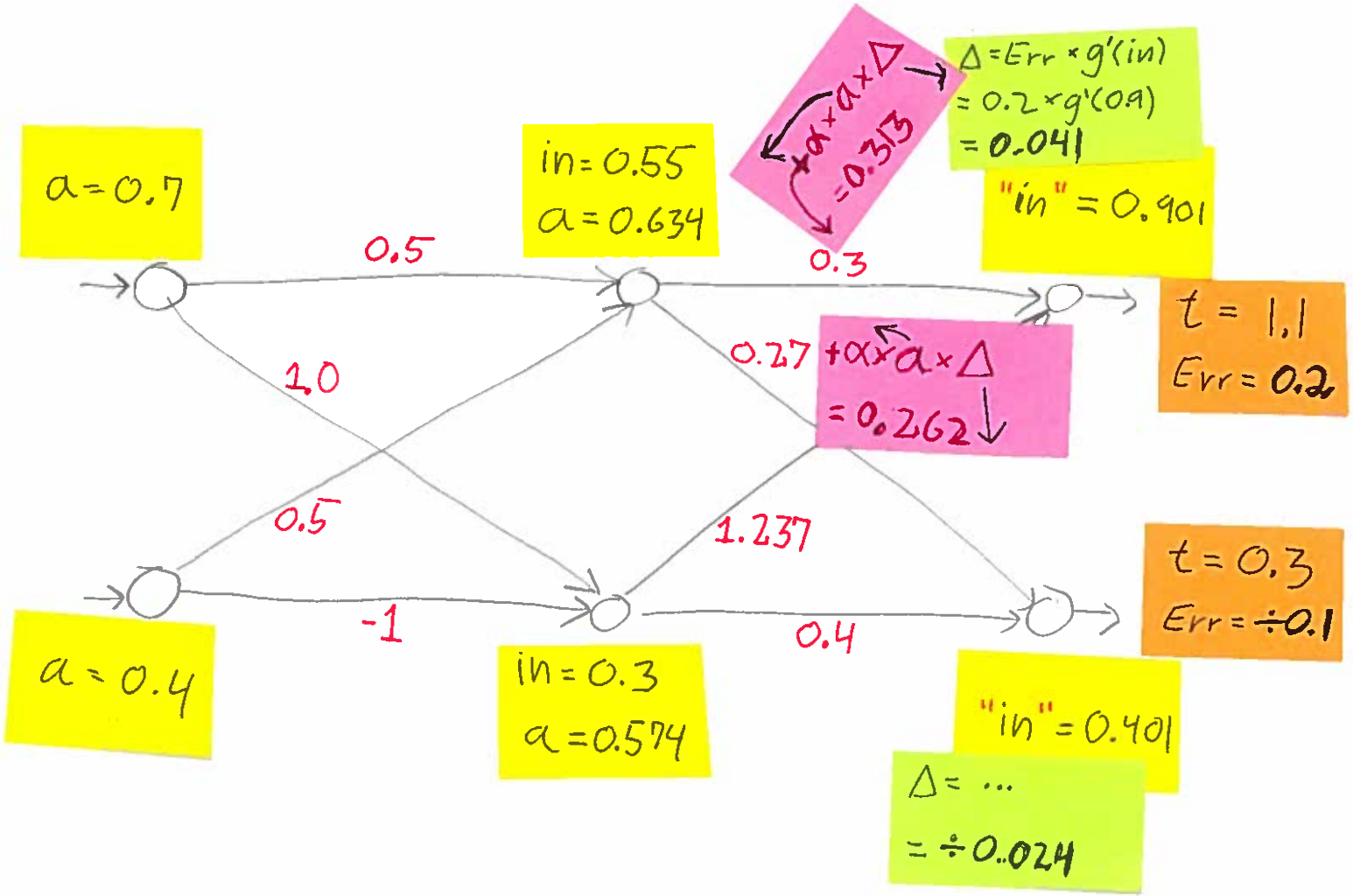

a = 0.7
in = 0.55
a = 0.634
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4
in = 0.3
a = 0.574

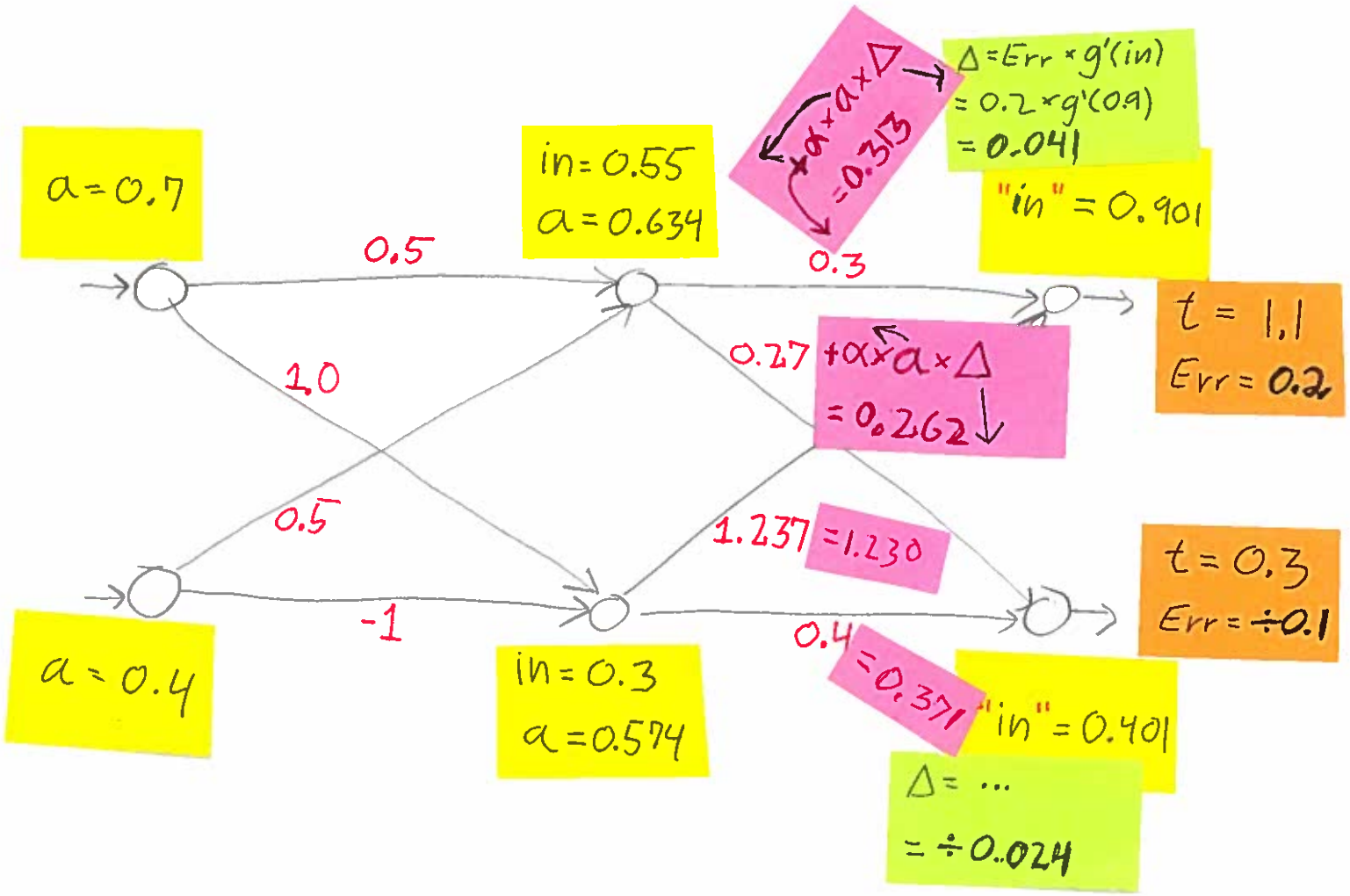

a = 0.7
in = 0.55
a = 0.634
"in" = 0.901
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4
in = 0.3
a = 0.574
"in" = 0.401

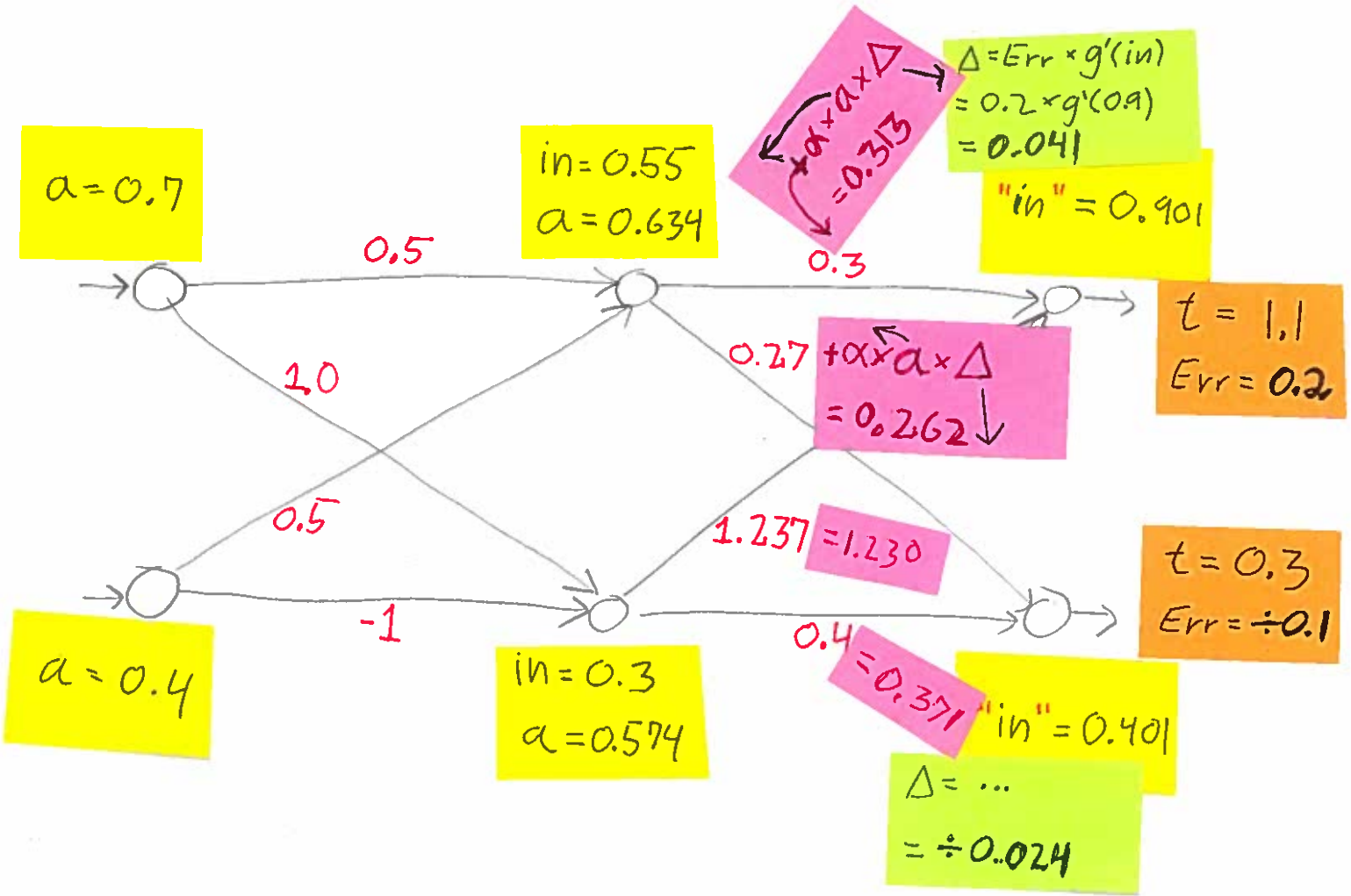

a = 0.7
in = 0.55
a = 0.634
"in" = 0.901
t = 1.1
Err = 0.2
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4
in = 0.3
a = 0.574
"in" = 0.401
t = 0.3
Err = -0.1


a = 0.7
in = 0.55
a = 0.634
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
t = 1.1
Err = 0.2
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4
in = 0.3
a = 0.574
t = 0.3
Err = -0.1
"in" = 0.401
Δ = ...
= -0.024

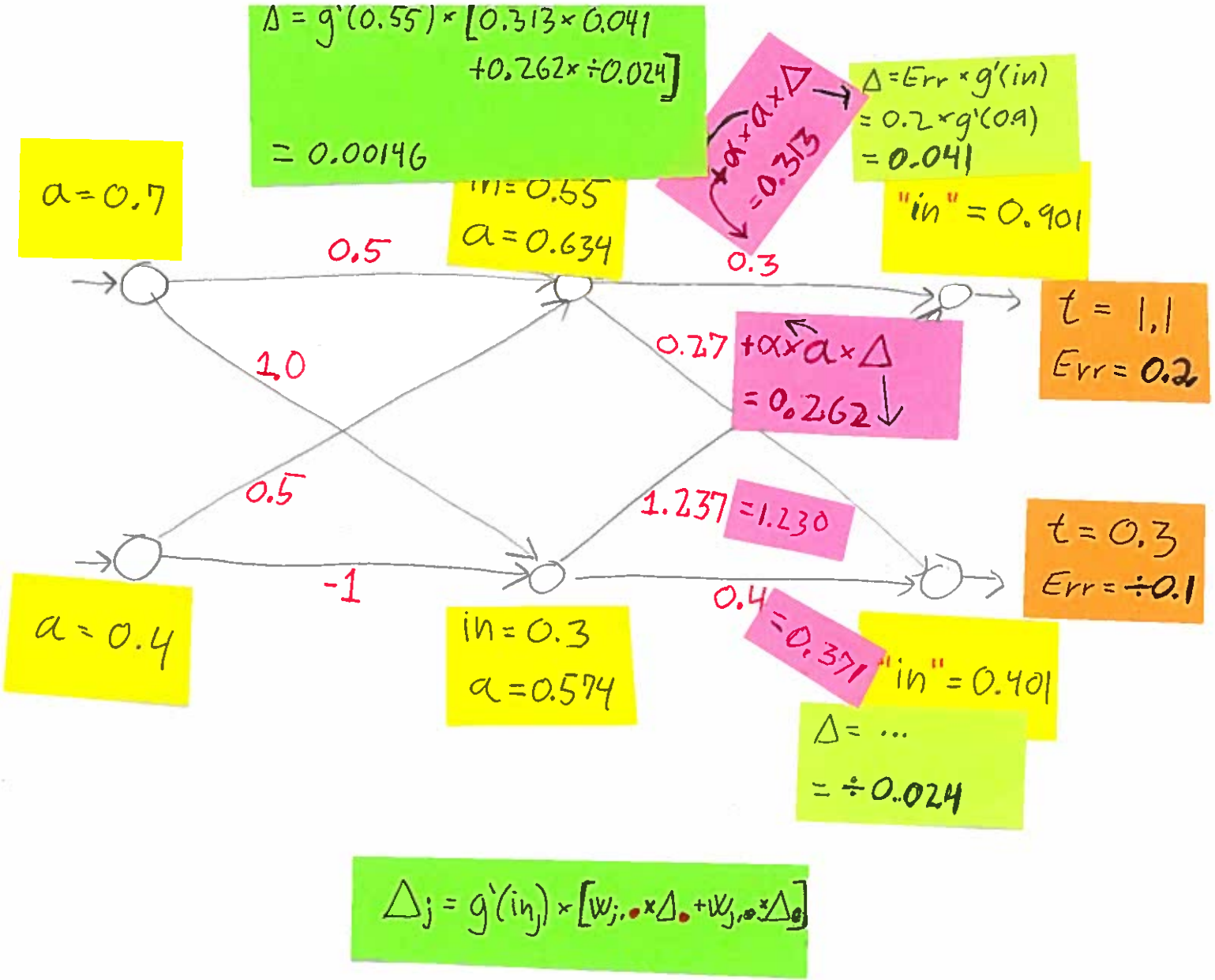

a = 0.7
in = 0.55
a = 0.634
+α × a × Δ
= 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
t = 1.1
Err = 0.2
0.5
0.3
1.0
0.27
0.5
1.237
-1
0.4
a = 0.4
in = 0.3
a = 0.574
t = 0.3
Err = -0.1
"in" = 0.401
Δ = ...
= -0.024

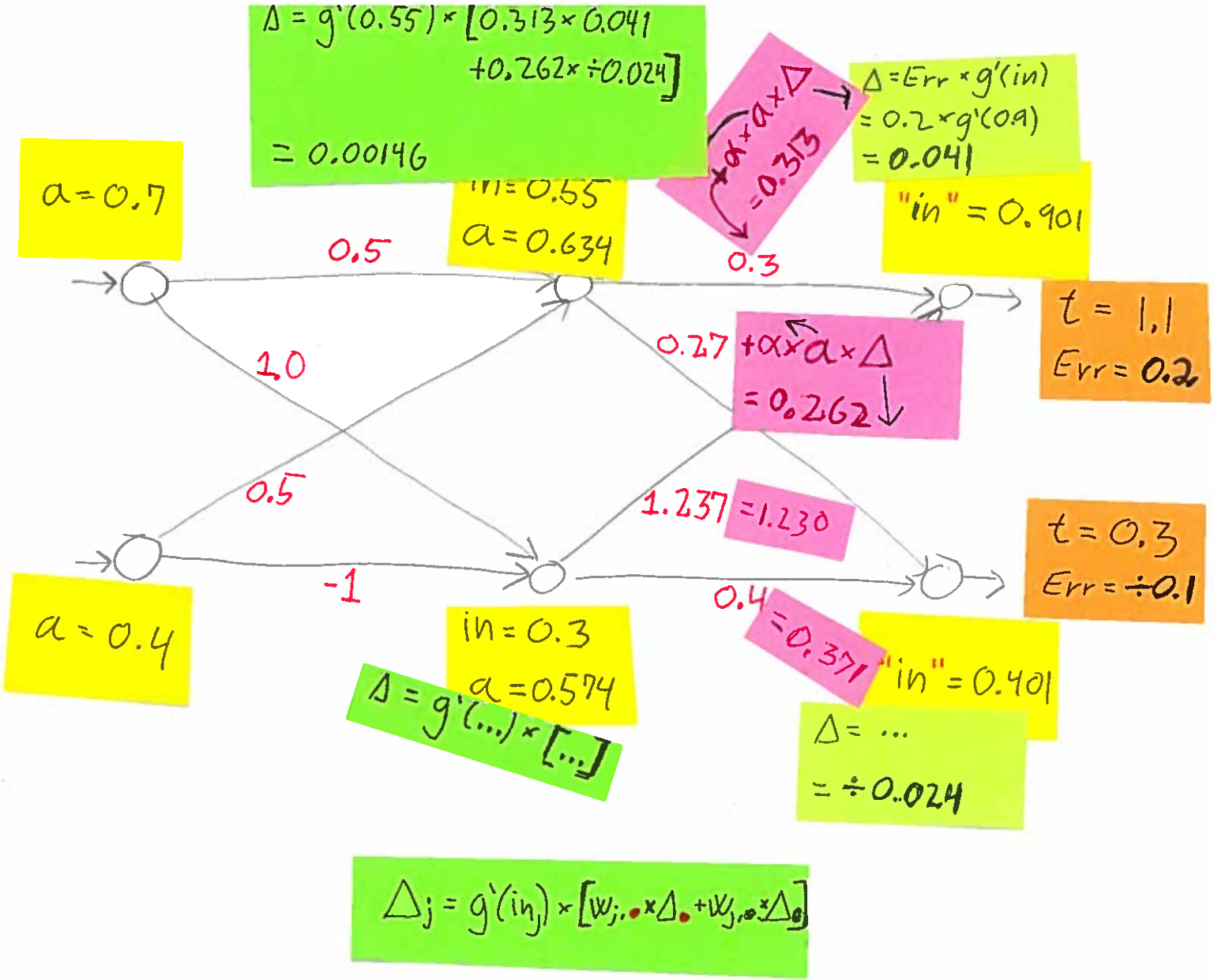

a = 0.7
0.5
in = 0.55
a = 0.634
α × a × Δ
= 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
0.3
t = 1.1
Err = 0.2
1.0
0.27
+α × a × Δ
= 0.262
0.5
1.237
a = 0.4
-1
in = 0.3
a = 0.574
0.4
t = 0.3
Err = -0.1
"in" = 0.401
Δ = ...
= -0.024

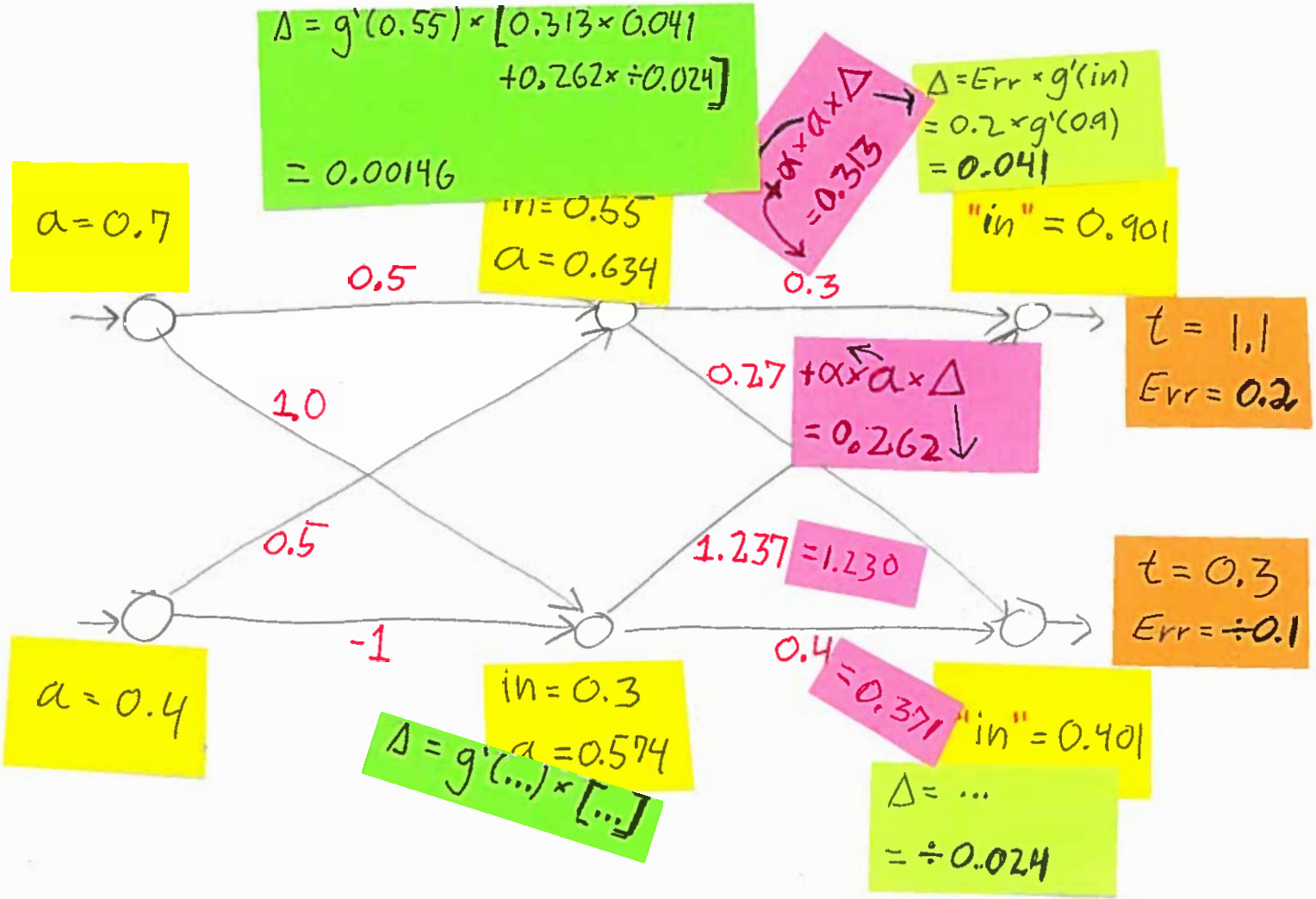

a = 0.7
0.5
in = 0.55
a = 0.634
+α × a × Δ
= 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
0.3
t = 1.1
Err = 0.2
1.0
0.27
+α × a × Δ
= 0.262
0.5
1.237
=1.230
t = 0.3
Err = -0.1
-1
a = 0.4
in = 0.3
a = 0.574
0.4
=0.371
"in" = 0.401
Δ = ...
= -0.024



$$\Delta_j = g'(in_j) \times [w_{j,\bullet} \times \Delta_\bullet + w_{j,\bullet} \times \Delta_\bullet]$$

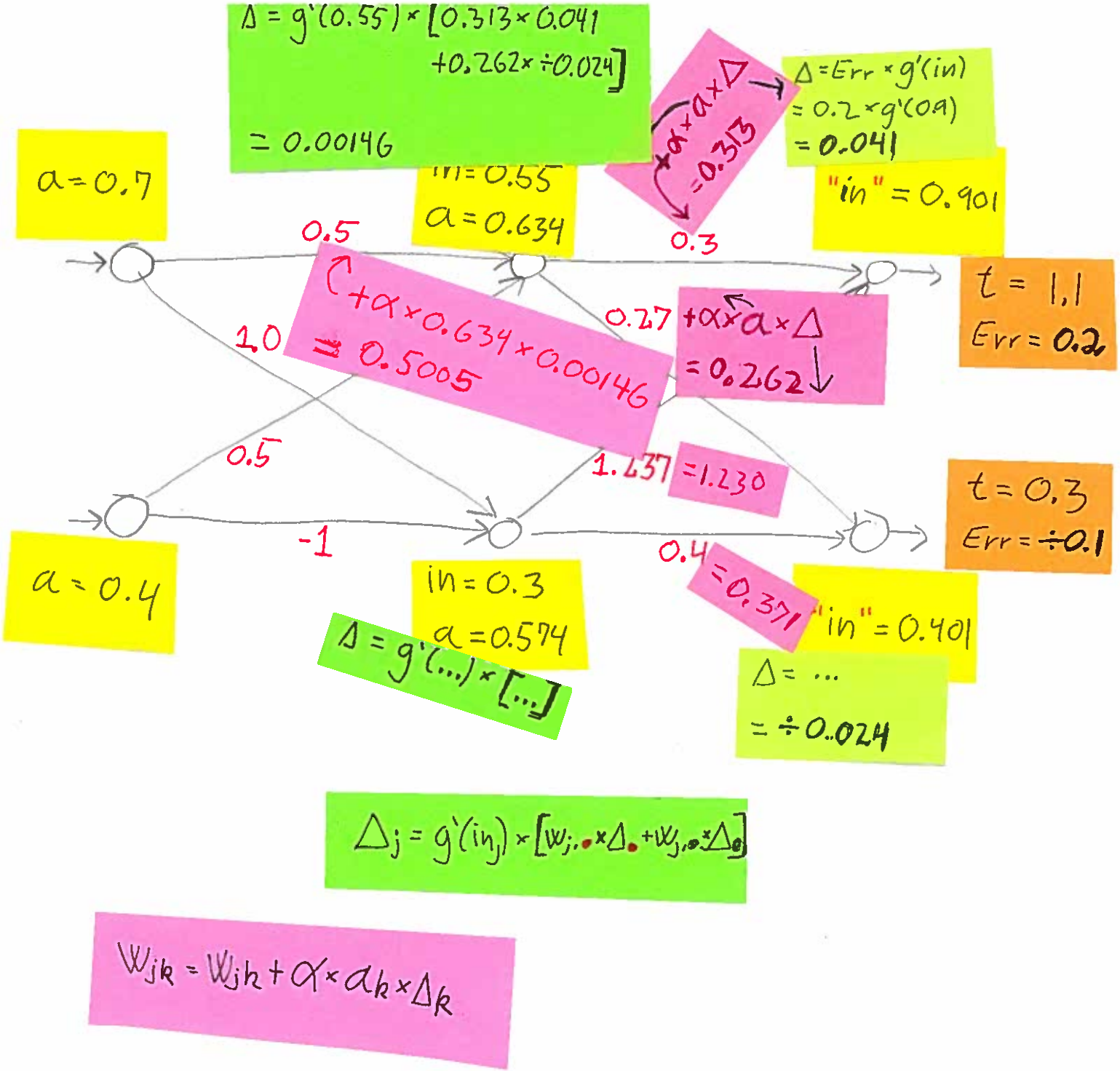

Δ = g'(0.55) × [0.313 × 0.041
+0.262 × ÷0.024]
= 0.00146
a = 0.7
in = 0.55
a = 0.634
0.5
1.0
0.5
-1
a = 0.4
in = 0.3
a = 0.574
+α × a × Δ
= 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
0.3
t = 1.1
Err = 0.2
0.27
+α × a × Δ
= 0.262
1.237 = 1.230
0.4
= 0.371
t = 0.3
Err = ÷0.1
"in" = 0.401
Δ = ...
= ÷0.024
Δj = g'(inj) × [wj,• × Δ• + wj,• × Δ•]

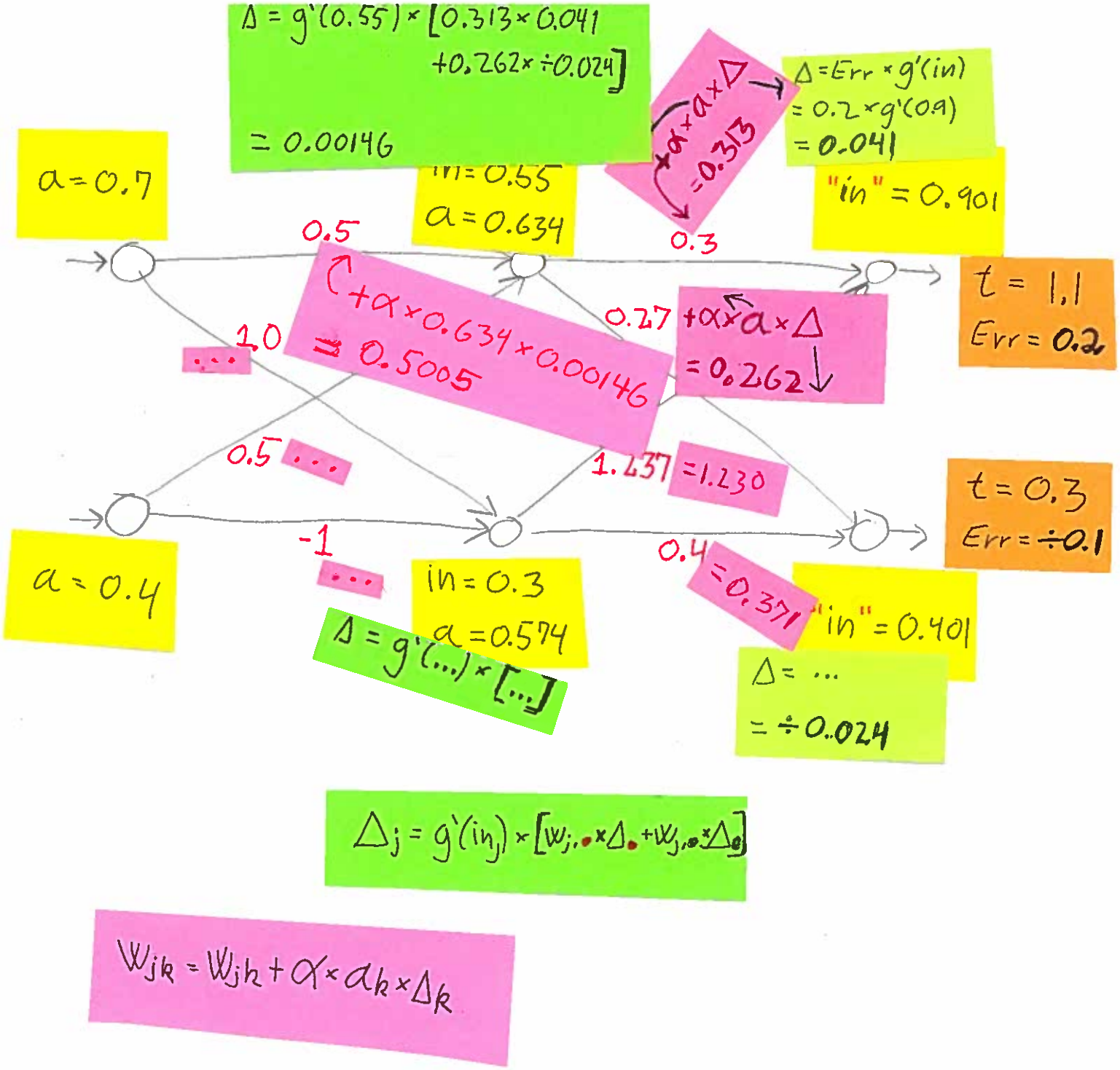

Δ = g'(0.55) × [0.313 × 0.041
+0.262 × ÷0.024]
= 0.00146
a = 0.7
0.5
in = 0.55
a = 0.634
+α × a × Δ
= 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
0.3
t = 1.1
Err = 0.2
1.0
0.27
+α × a × Δ
= 0.262
0.5
1.237 = 1.230
t = 0.3
Err = ÷0.1
-1
a = 0.4
in = 0.3
a = 0.574
Δ = g'(...) × [...]
0.4
= 0.371
"in" = 0.401
Δ = ...
= ÷0.024
Δj = g'(inj) × [wj,• × Δ• + wj,• × Δ•]



$$\Delta_j = g'(in_j) \times [w_{j,\bullet} \times \Delta_\bullet + w_{j,\bullet} \times \Delta_\bullet]$$

$$W_{jk} = W_{jk} + \alpha \times a_k \times \Delta_k$$


Δ = g'(0.55) × [0.313 × 0.041 + 0.262 × ÷0.024]
= 0.00146
+α × a × Δ = 0.313
Δ = Err × g'(in)
= 0.2 × g'(0.9)
= 0.041
"in" = 0.901
a = 0.7
in = 0.55
a = 0.634
0.5
0.3
t = 1.1
Err = 0.2
0.27
+α × a × Δ
= 0.262
1.0
+α × 0.634 × 0.00146
= 0.5005
0.5
1.237
=1.230
t = 0.3
Err = ÷0.1
a = 0.4
-1
in = 0.3
a = 0.574
0.4
=0.371
"in" = 0.401
Δ = g'(...) × [...]
Δ = ...
= ÷0.024
Δj = g'(inj) × [wj,• × Δ• + wj,• × Δ•]
Wjk = Wjk + α × ak × Δk



$$\Delta_j = g'(in_j) \times [w_{j,\bullet} \times \Delta_\bullet + w_{j,\bullet} \times \Delta_\bullet]$$

$$W_{jk} = W_{jk} + \alpha \times a_k \times \Delta_k$$

Training of a network

- initialize all weights by random numbers
- repeating epochs
    - each running through all training tuples
- and stopping when:
    - the square of errors has converged

- Notice: under training, errors are signed (pos or neg), but squaring for stop criterion eliminates the sign!!!
- The stopping error measured either for all training data, or for another independent sample

Training of a network

- initialize all weights by random numbers
- repeating epochs
    - each running through all training tuples
- and stopping when:
    - the square of errors has converged

- Notice: under training, errors are signed (pos or neg), but squaring for stop criterion eliminates the sign!!!
- The stopping error measured either for all training data, or for another independent sample

A simpler stop criterion for the exercise: repeat 1000 times and hope the best ;-)