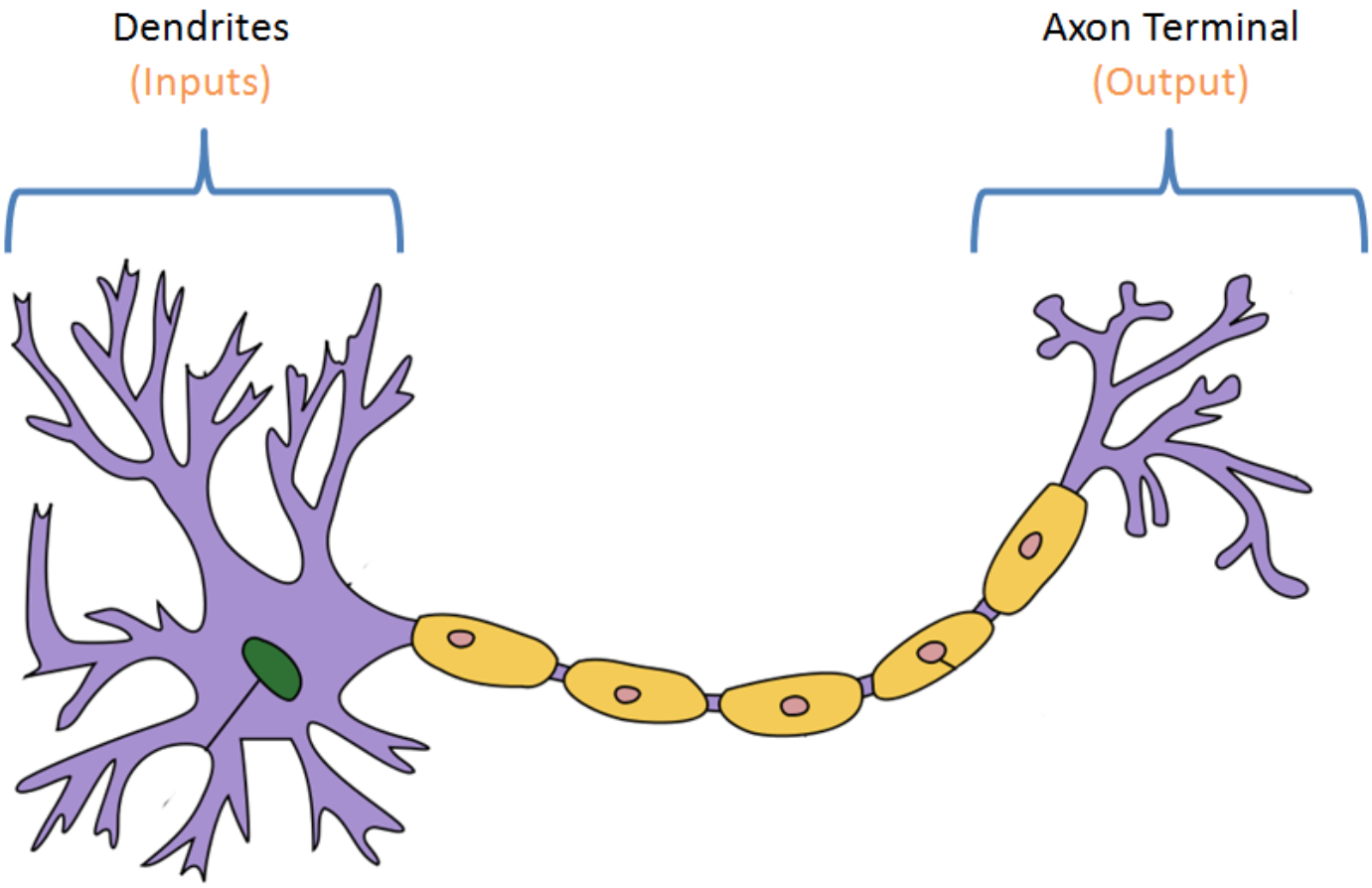
Neural Nets

Hey this is artificial intelligence, let's take a look at the gold standard of intelligence: the brain!

As we know, the brain consists of about 100 *billion* neurons that resemble a certain basic structure:



Here we have a basic neuron structure with dendrites, which collect the neurotransmitters from other connected neurons at their synapses...

...and if enough neurotransmitters are excitatory such that the neuron "fires," it will then release neurotransmitters of its own into the synapses connected to its axon terminals.

In simplest terms, we want to try to model certain computational problems using the input / output semantics of actual neurons to tackle tasks like classification and deep learning.

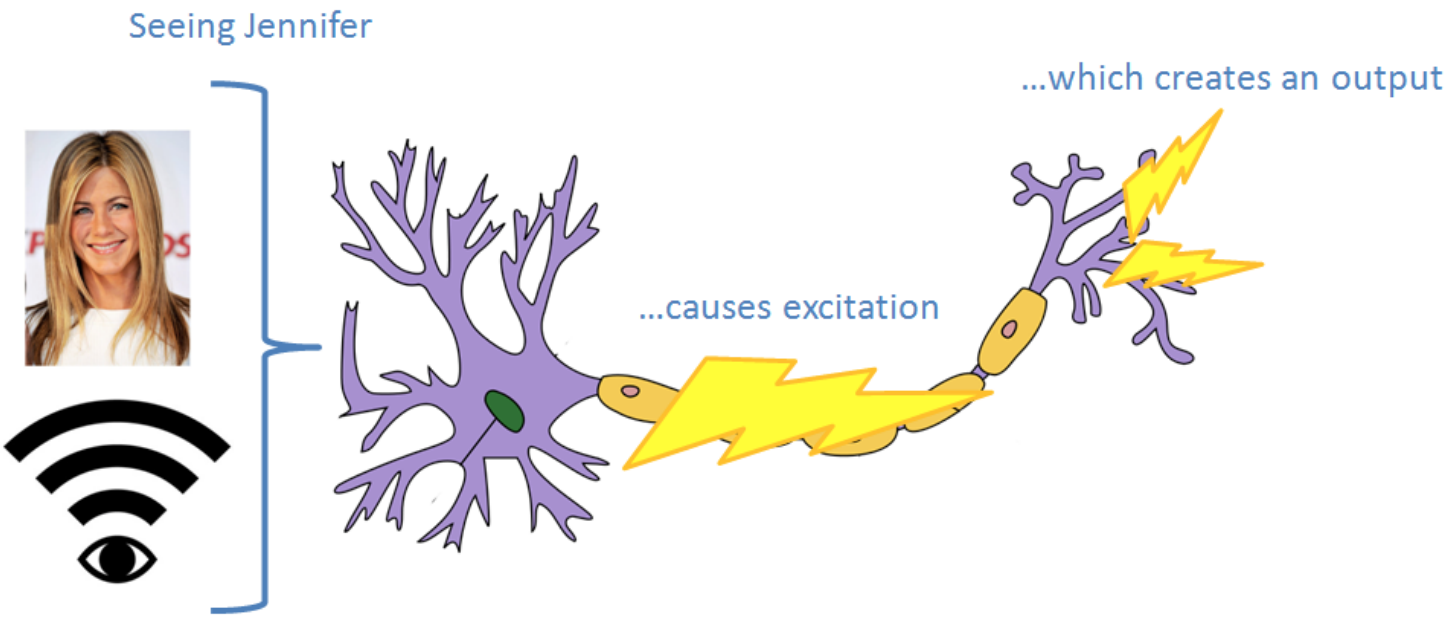
Let's root neural networks in a familiar problem: classification.

Remember that classification is the task of deciding whether our input attributes is indicative of a particular class, and returns a yes / no answer.

Example

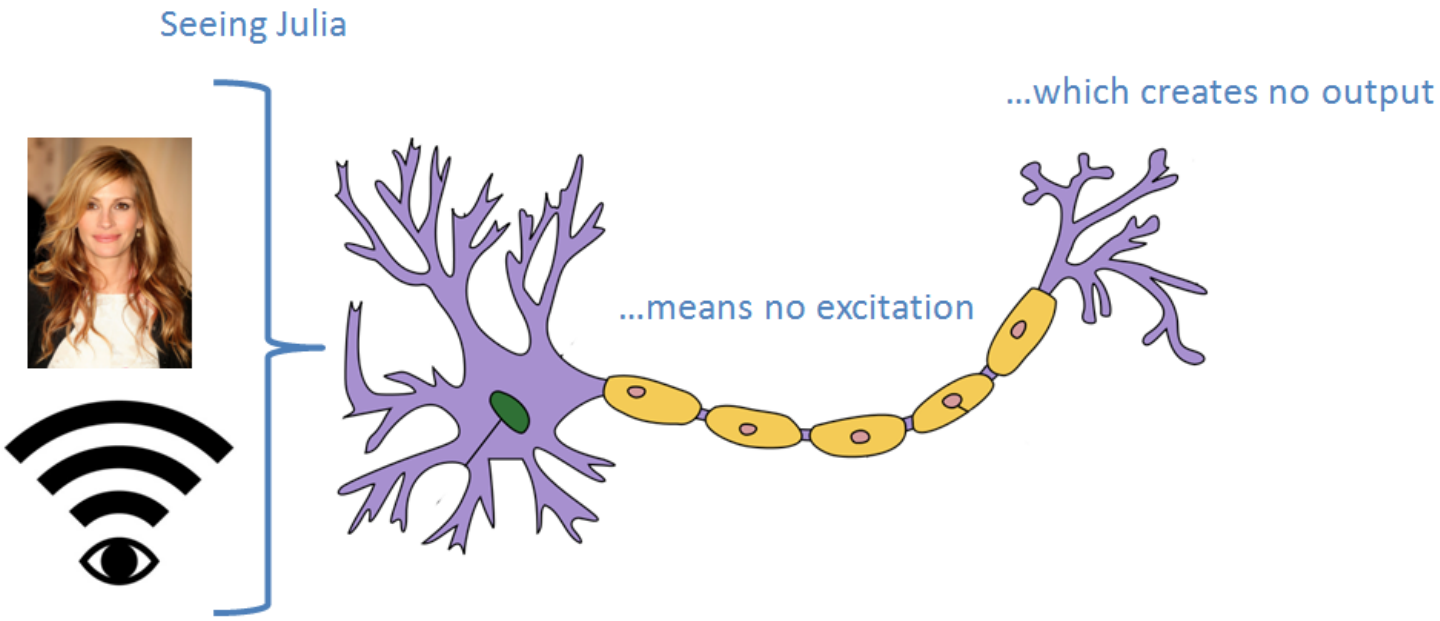
☑ Take, for example, a very simple neuron which (through some process irrelevant to us) gets activated whenever you're looking at Jennifer Aniston (this was a real study, look it up):

This might look like the following:



(somewhere, someone who actually studies neuroscience is reading this and cringing)

However, if that same neuron were to respond to input of seeing Julia Roberts instead (which is, I believe, what they did in the study) it might not activate!



It would be nice, then, if we could take a bunch of neurons with certain activation patterns, and combine their outputs to serve our computation interests.

Let's look at the basic case first...

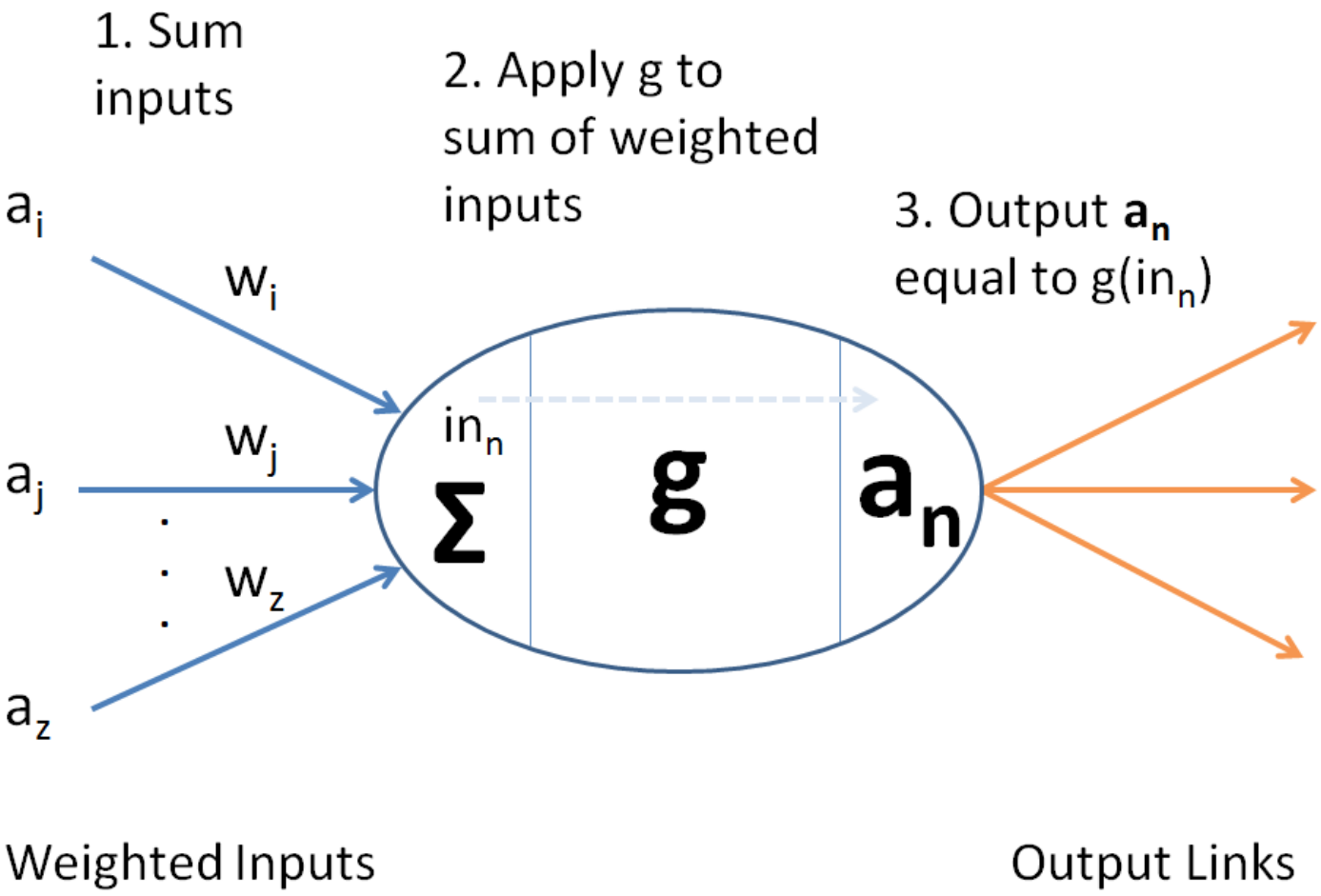
The anology to our neural networks is as follows:

- Artificial neural networks have **input** elements representing our attributes of interest (like the dendrites).

- Artificial neurons have **activation functions** that determine whether or not, based on the input, they will fire.

- Artificial neural networks have **output** elements that can be used for classification or simply returning results of the internal computations...

Let's look at a very simple artificial neuron and then look at how to use it.



What you choose for the activation function is up to you, but typical choices are the following:

An activation function that is a **step / threshold function** will only produce an output when some minimum value is reached for its sum of weighted inputs.

Typically, step functions output 0 if we haven't reached some threshold yet, or 1 otherwise.

These bear a close semblance to how actual neurons operate.

An activation function that is a **logistic function** (the inverse of the natural logit function) produces an s-curve used to convert the logarithm of odds into a probability (positive inputs approach a ceiling of 1, negative inputs approach a floor of -1).

Why don't we look at a single node neuron and see how this all fits together.

Here are the steps for computation at a single neuron:

Step One: Sum weighted inputs for n input links

$$in_j = \sum_{i=0}^{n} w_i * a_i$$

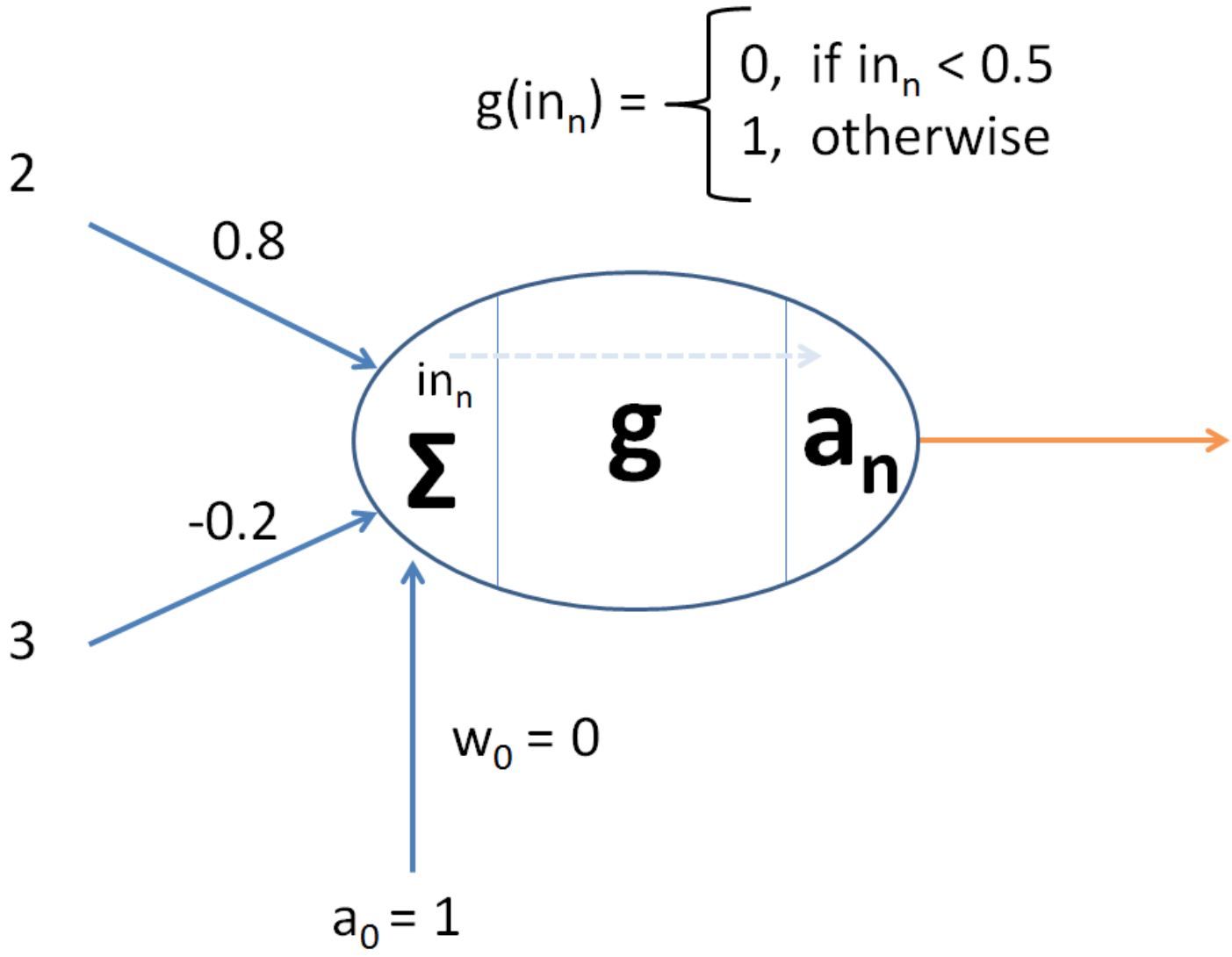
Step Two: Apply $g(in_j)$ where g is the activation function

$$a_j = g(in_j)$$

Step Three: Propagate the output a_j to any output links

Example

☑ What will the following neuron output for the given inputs, weights, and activation function?

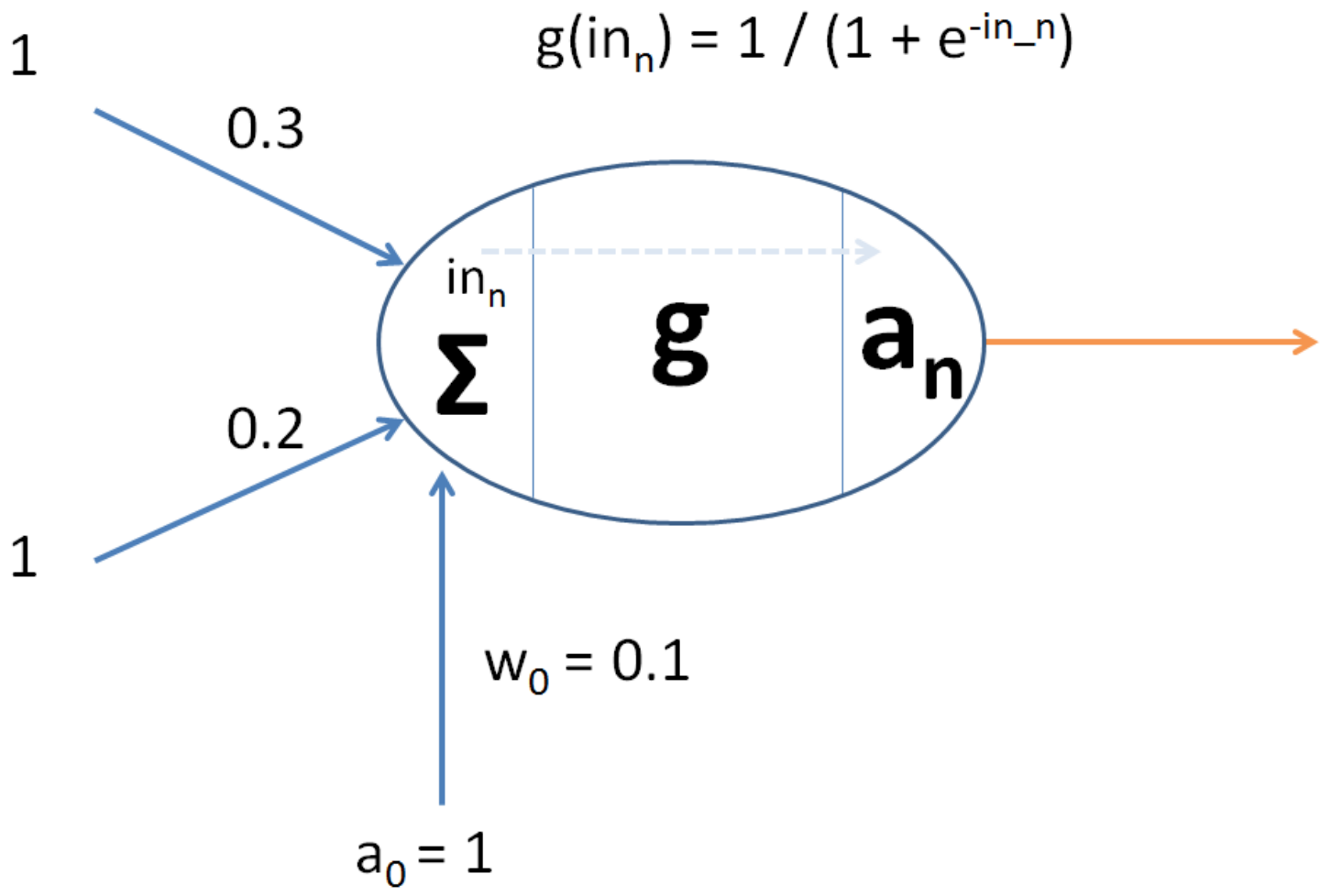


The above neuron, which is a single unit whose activation function is a step function, is called a **perceptron**.

Some things to notice:

- Weights can be negative (to immitate an inhibitory effect like our own neurons have)
- Weights can be 0
- (not shown) Weights can be any real number, not necessarily percentage values
- The bias input value is always 1 though its weight can be modified
- Other input values can be whatever you'd like to suit the problem at hand

Example
What will the following neuron output for the given inputs, weights, and activation function?



The above neuron, which is a single unit whose activation function is a logistic function, is called a **sigmoid perceptron**, sometimes used as a probability calculator.

The **bias** input is simply a means of shifting our sigmoid curve left and right by manipulating its weight, and (described later) for handling learning where our inputs are 0.

Observe how the output of a sigmoid perceptron changes by manipulating the bias input:

Say for perceptron n with 2 inputs and 1 bias input: (where sig() is the sigmoid logistic function and $w_0 * a_0$ is the bias term)

$$\begin{aligned} a_n = sig(\; & w_i * a_i \\ & + w_j * a_j \\ & + w_0 * a_0) \end{aligned}$$

$$\begin{aligned} a_n = sig(\; & 0.3 * 1 \\ & + 0.2 * 1 \\ & + 0.0 * 1) = 0.62 \end{aligned}$$

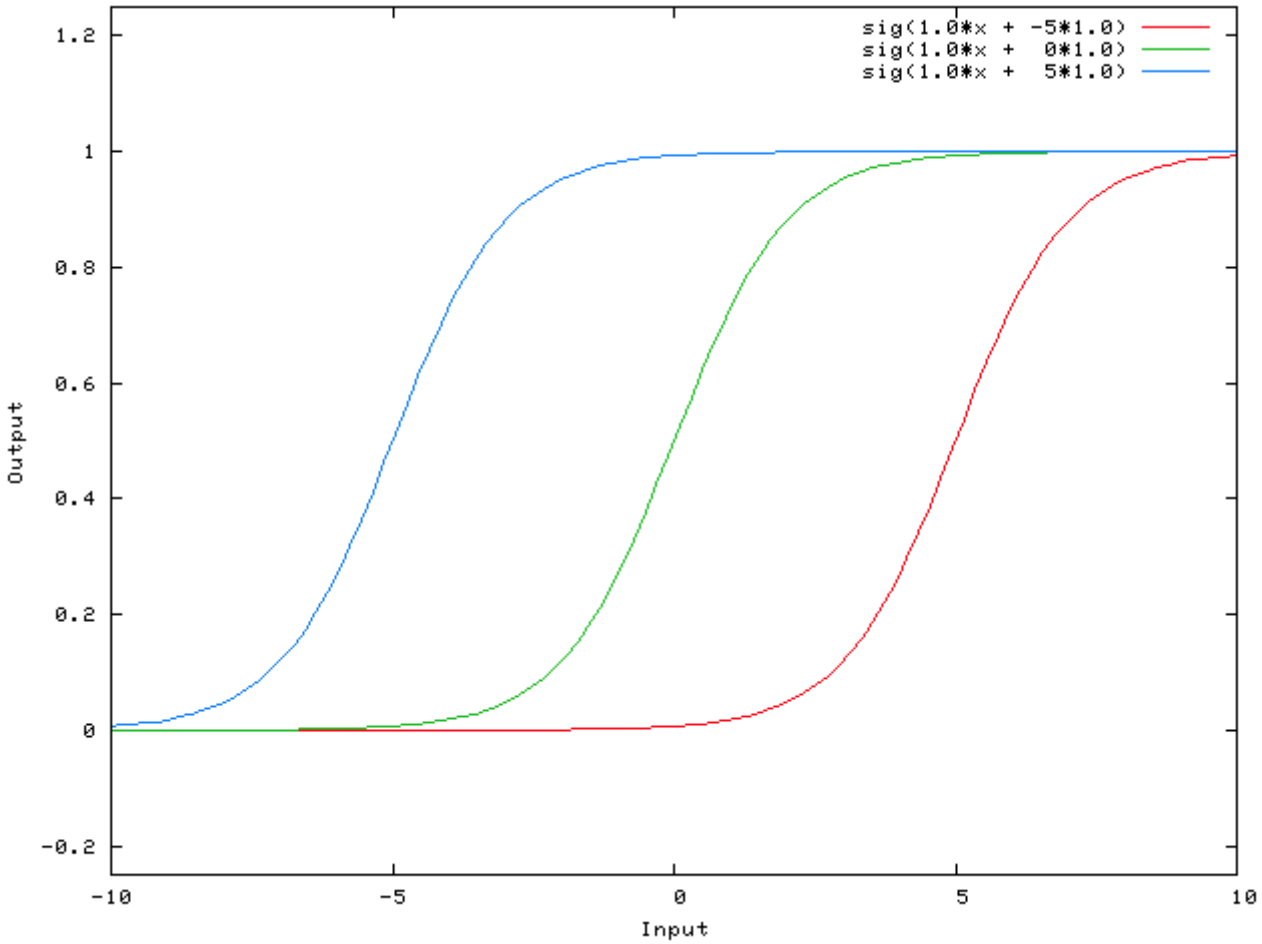
Change w_0 to 0.5 shifts right

$$\begin{aligned} a_n = sig(\; & 0.3 * 1 \\ & + 0.2 * 1 \\ & + 0.5 * 1) = 0.73 \end{aligned}$$

Change w_0 to -0.5 shifts left

$$\begin{aligned} a_n = sig(\; & 0.3 * 1 \\ & + 0.2 * 1 \\ & - 0.5 * 1) = 0.5 \end{aligned}$$

Visually, we can see the shift as the following (depicted below: only one input and taken from a really good SO article on Bias inputs, linked):



(http://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks)
Typically you will have one bias input connected to all non-input neurons, though you can have one per level as well.

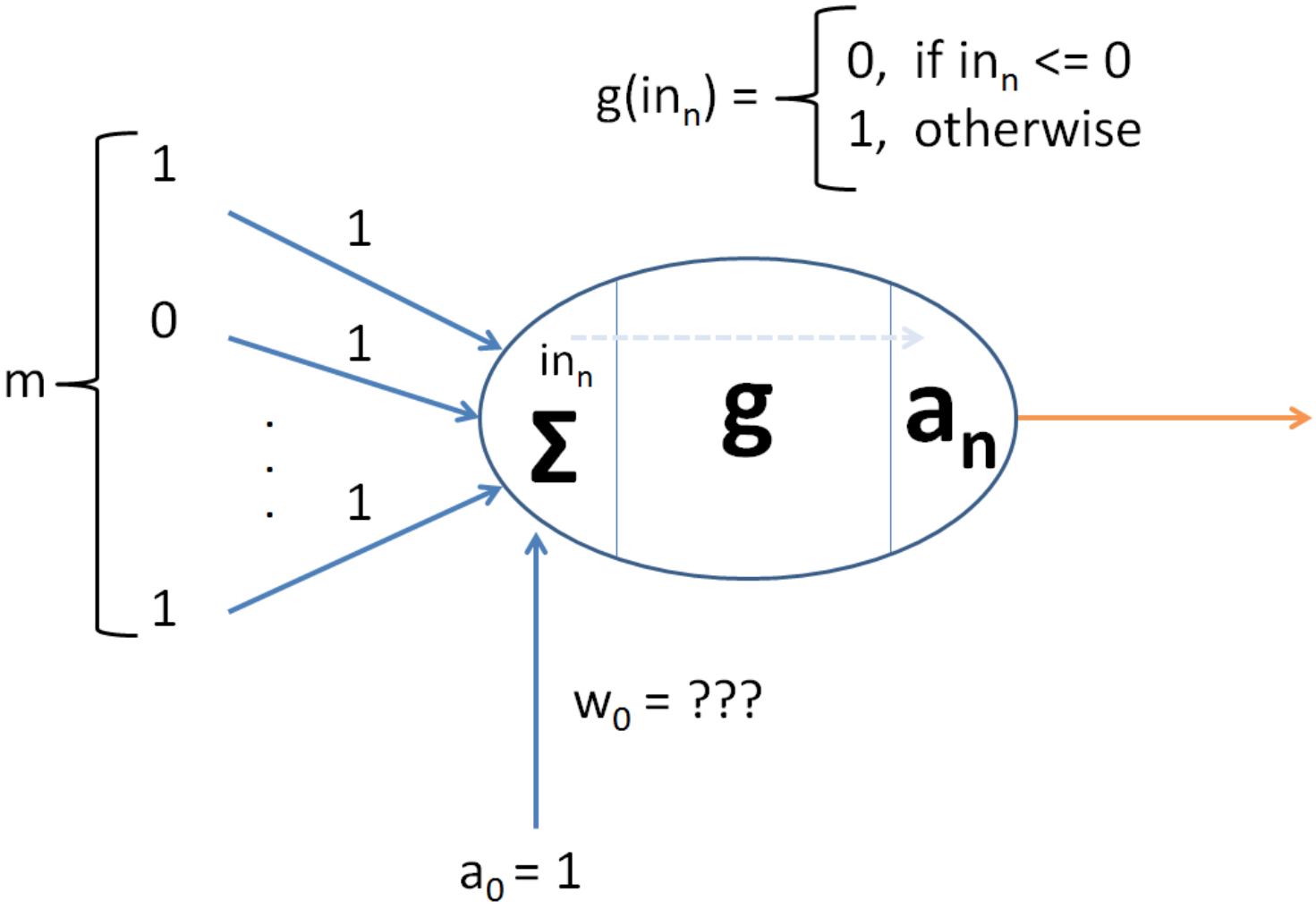
Succinctly, the output of some neuron n is given by:

For i input links:

$$a_n = g(\sum_i w_i * x_i)$$

Perceptrons on their own can do many tasks quite well. The majority function is one of them:

Example
☑ Consider the following perceptron (step activation function) that implements the **majority function**, outputting 1 whenever the majority (MORE than half) of its inputs are also 1. Determine the weight of the bias input for m inputs (each with weight and activation values of 1).

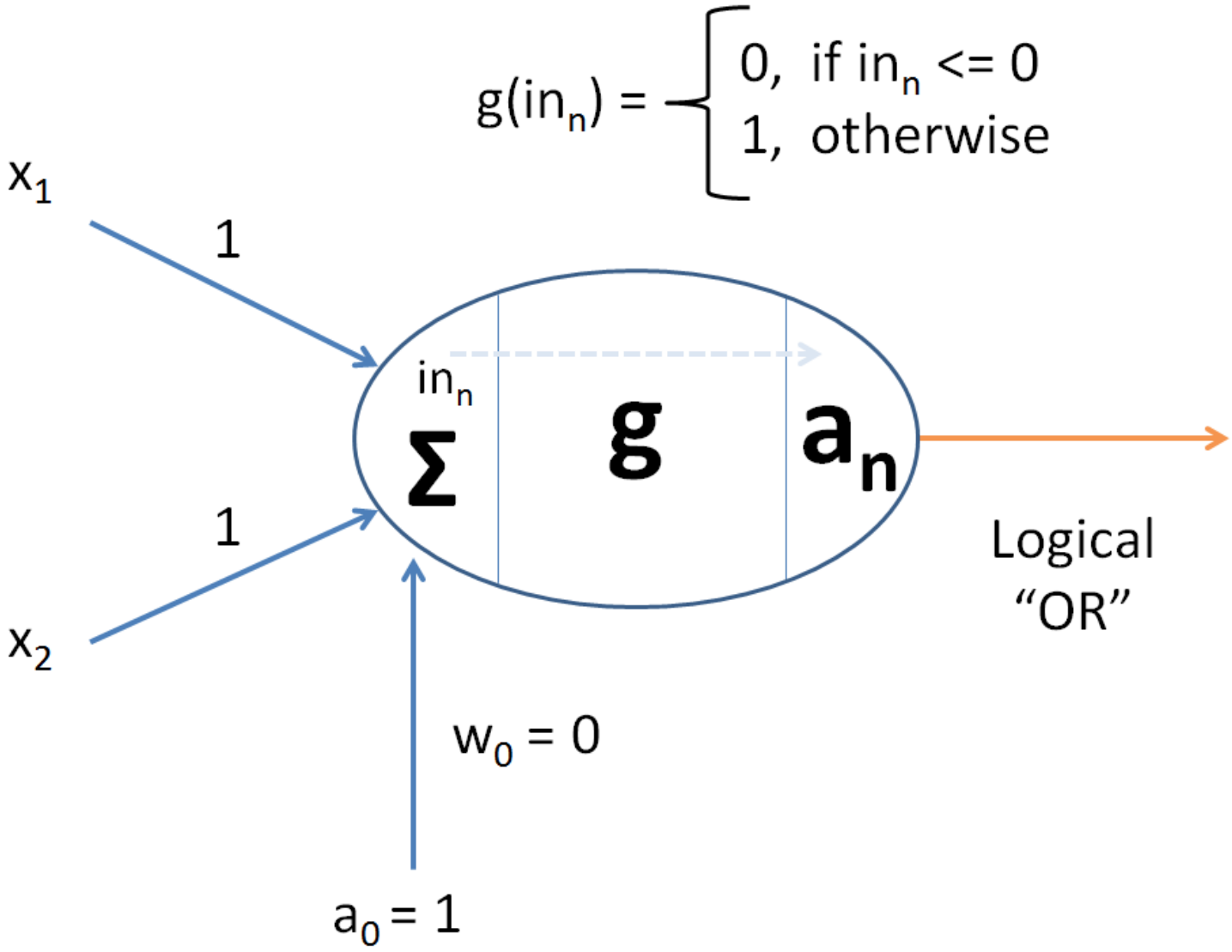


Need a hint? Consider m = 10 with 2 separate cases: in Case 1, 5 of those inputs are 1 and the others 0; in Case 2, 6 of those inputs are 1 and the others 0.

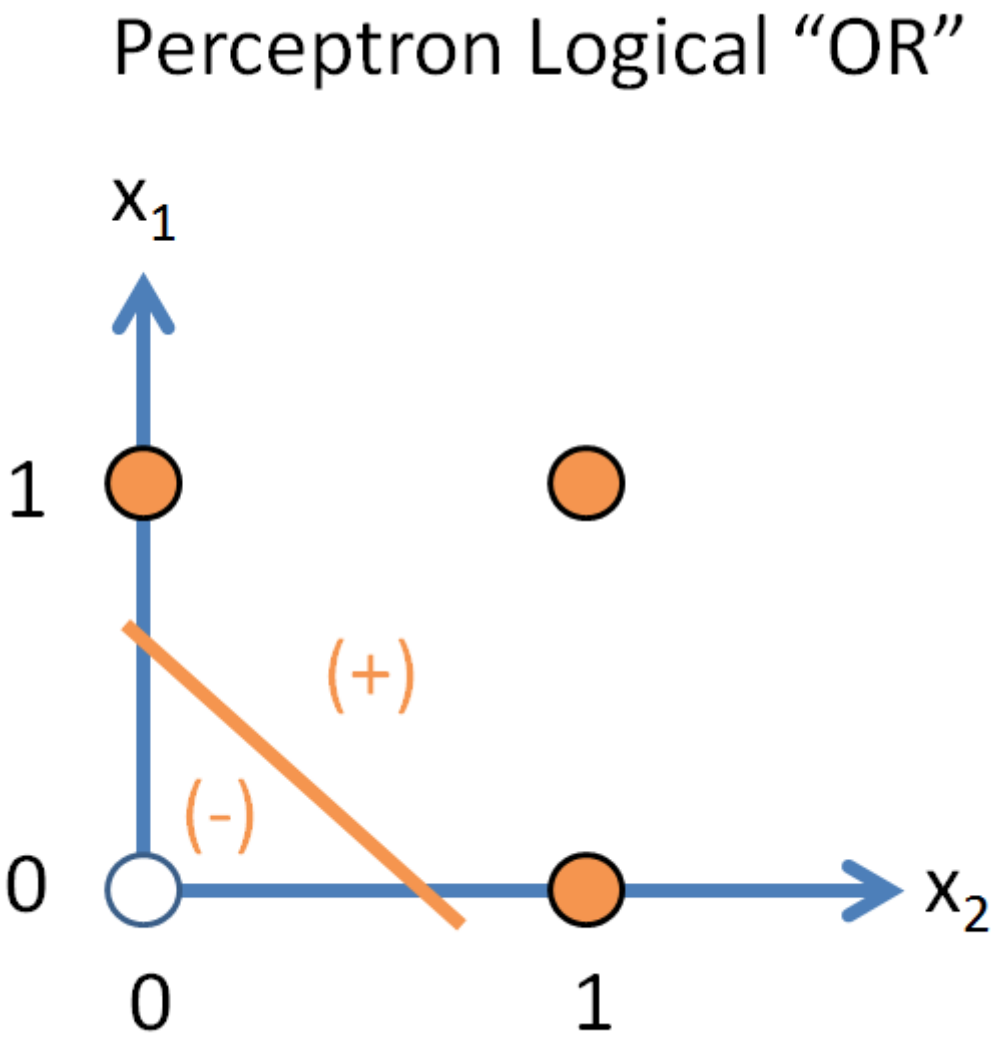
Click for solution.

Now there is one problem to discuss with perceptrons. Let's condider the perceptron that performs a logical, binary OR, i.e., outputs 1 if either of its inputs (or both) are 1.

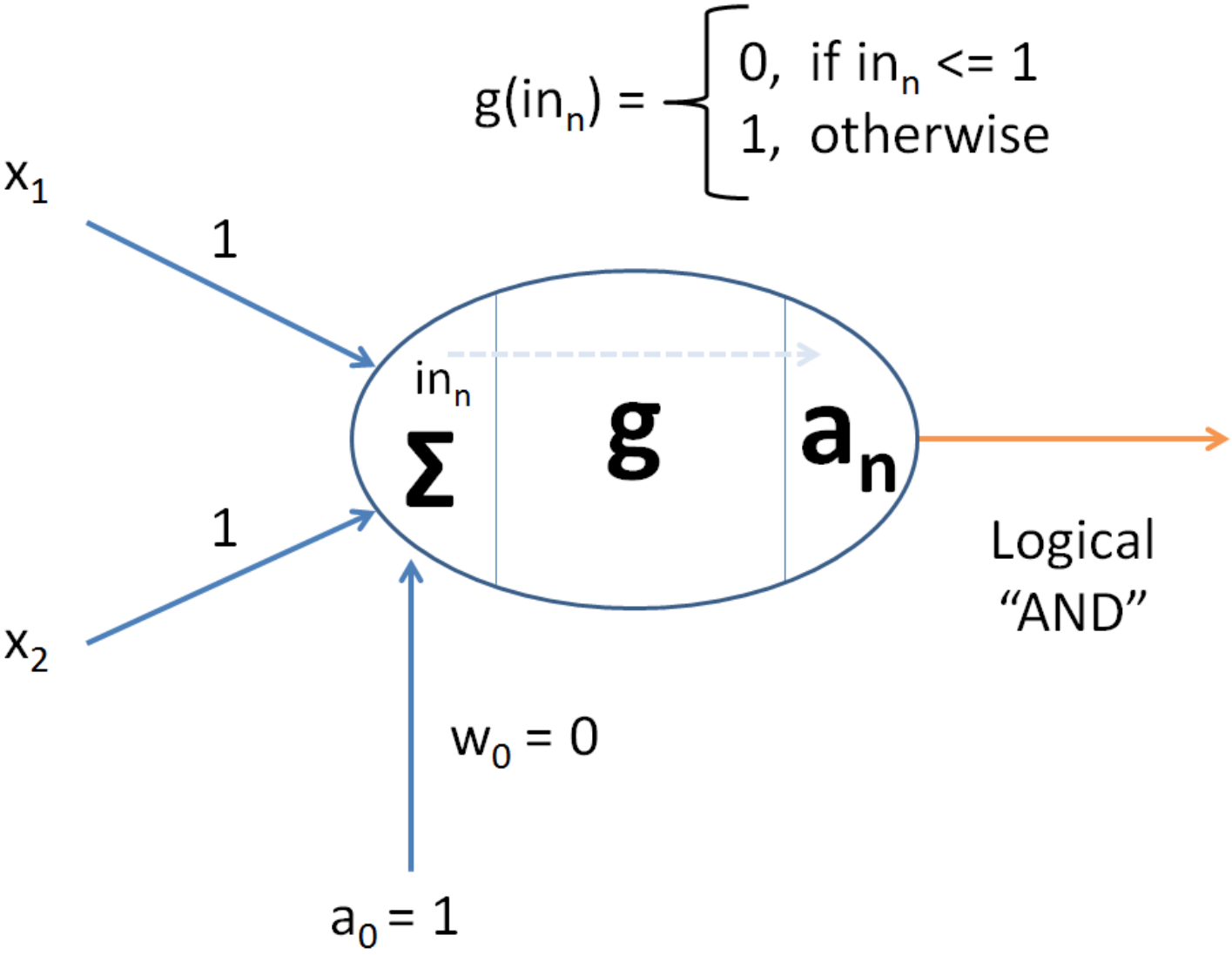
We might create a perceptron that looks like:



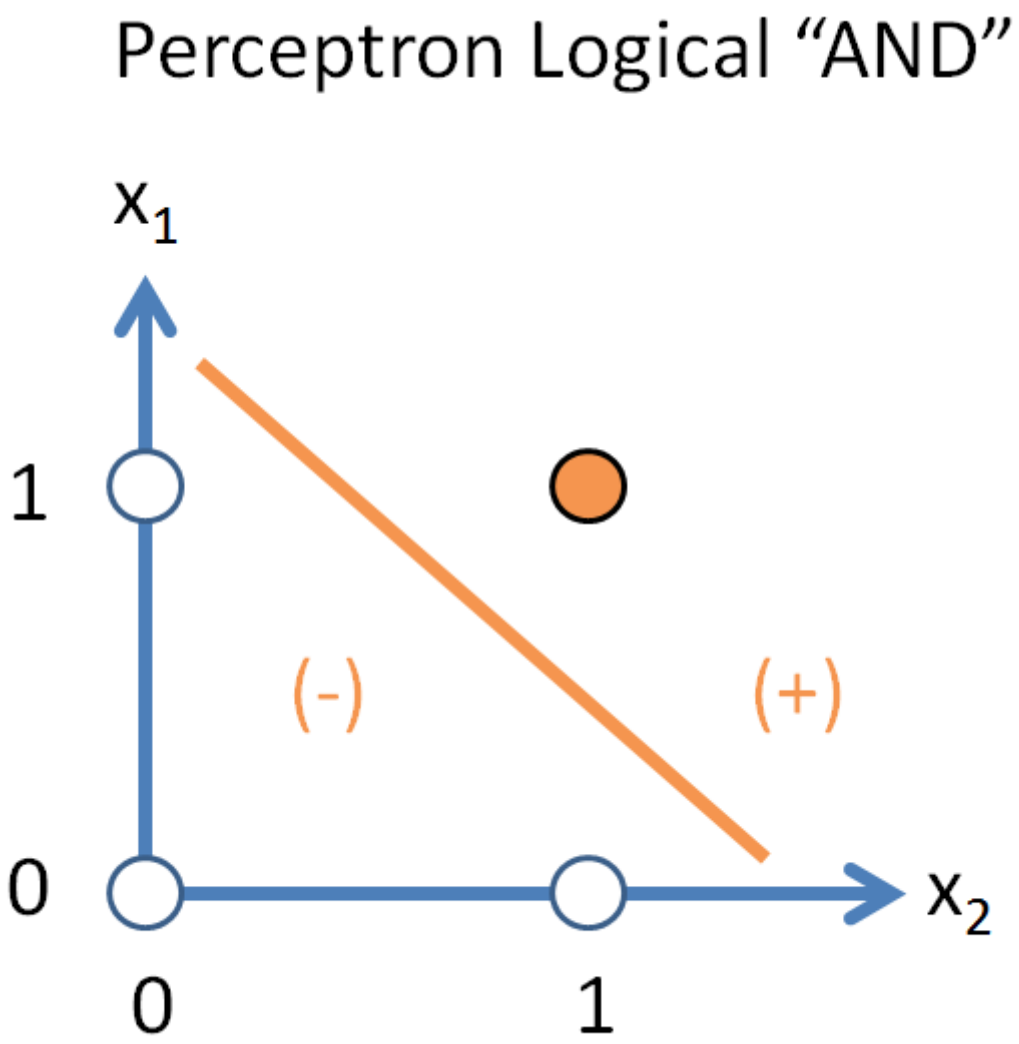
Which works out for the classification task of separating the 0 outcomes of a binary logical OR from the 1 outcomes:



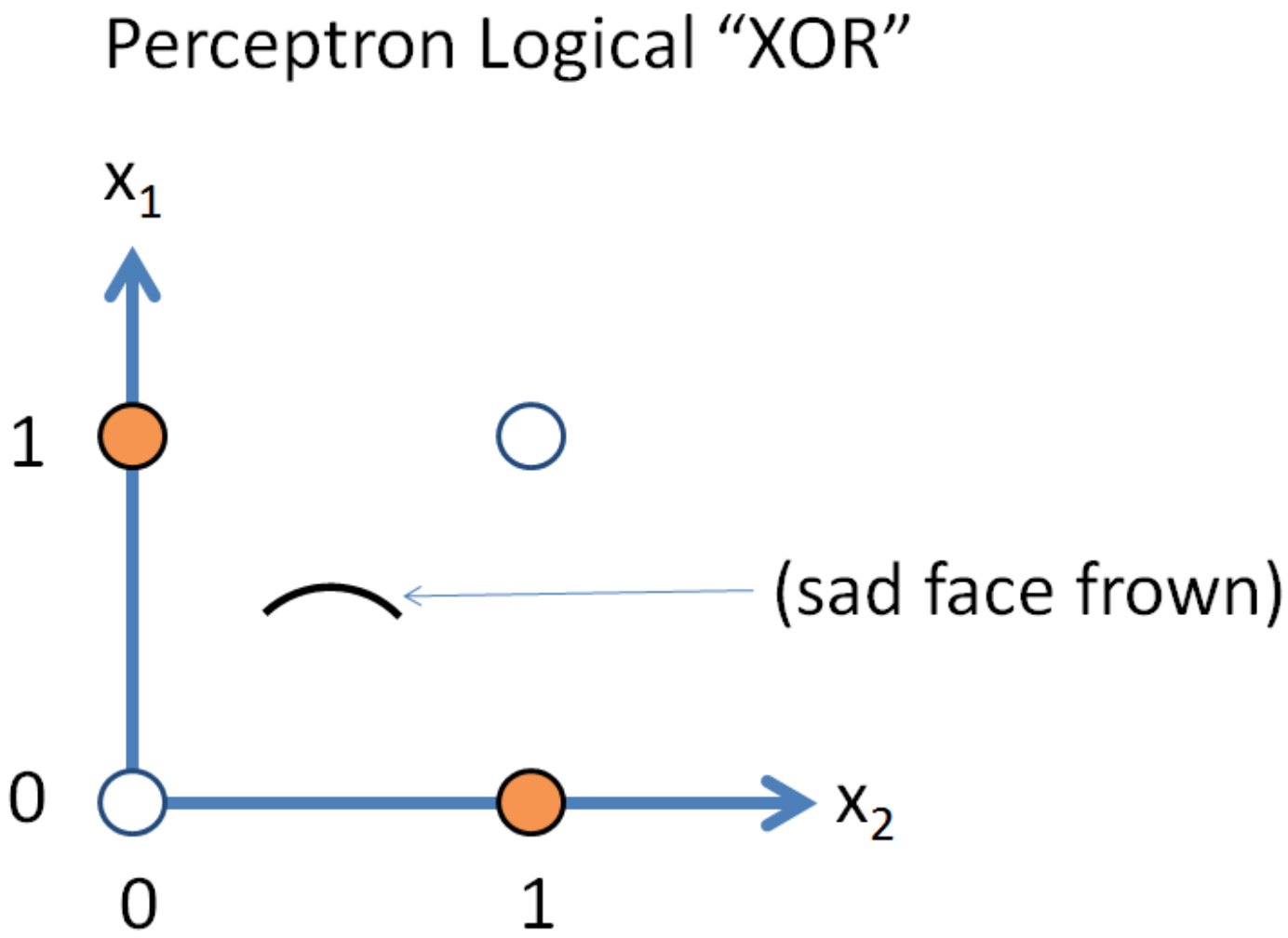
The same can be said of a perceptron that performs a logical, binary AND, i.e., outputs 1 iff both inputs are 1.



Which works out for the classification task of separating the 0 outcomes of the AND operation from the 1 outcomes:



But what about logical XOR? i.e., returning 1 iff both inputs are different values:



Can't do it with a single dividing line like we did with AND and OR!

> Perceptrons by themselves can only solve classification problems that are **linearly-separable**.

For this reason, we need to talk about **networks** of perceptrons that handle **non-linear regression.**

Feed-Forward Networks

Now that we have some concept of the individual perceptrons, we can talk about tying them together to handle interesting tasks!

> A **feed-forward neural network** is one with some number of input and output nodes with edges that form a directed, acyclic graph.

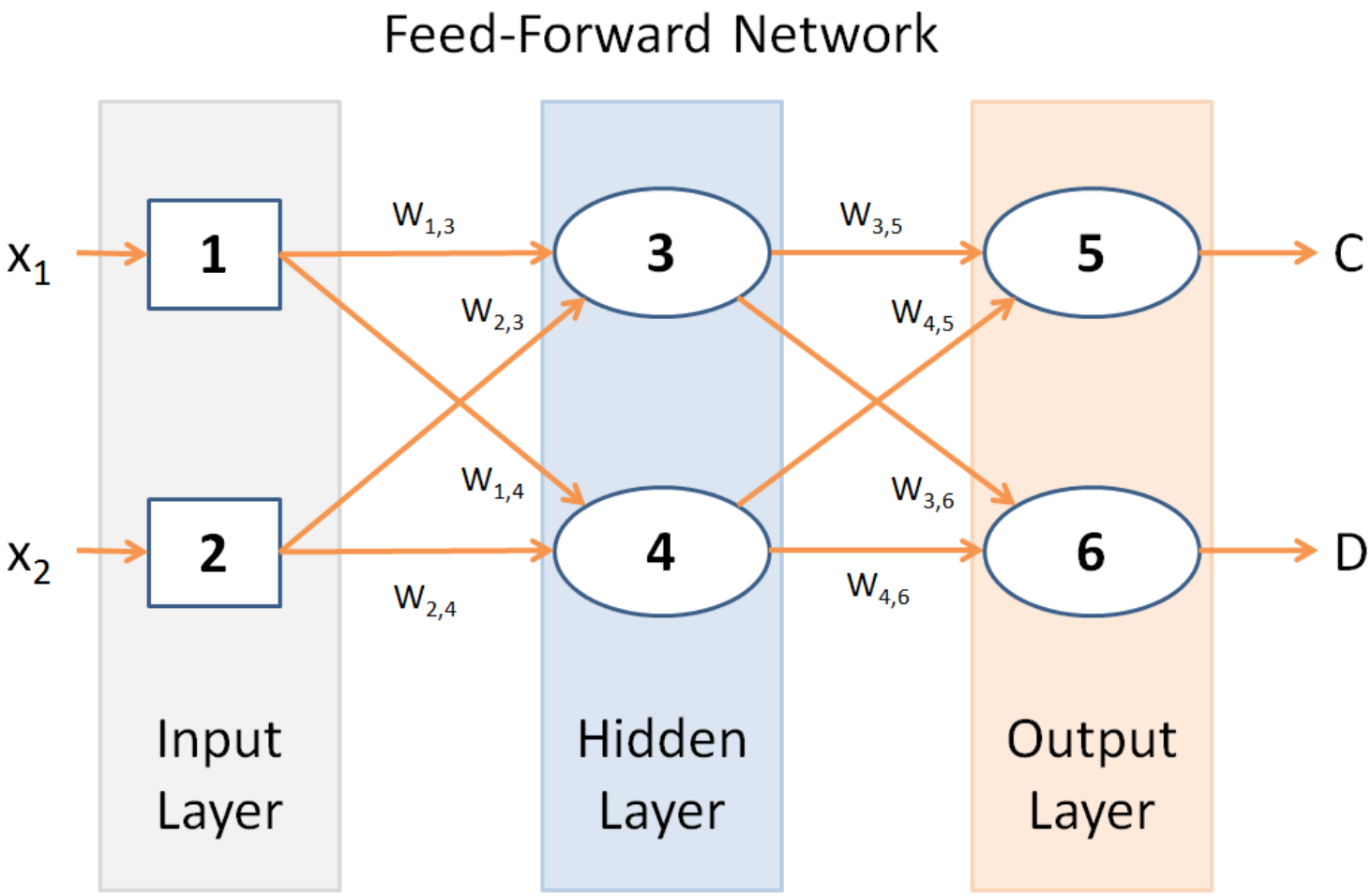
In other words, our feed-forward networks only propagate their outputs towards the output nodes.

As such, we can connect layer after layer of neuron to one another with the outputs of neurons at level i serving as inputs to neurons at level $i + 1$.

A network's **input layer** for some vector of inputs $X = (x_1, x_2, \dots)$ sets each neuron's output in the input layer to its corresponding input value.

So, for input neurons $1, 2, \dots, n$ the outputs of each is: $(a_1, a_2, \dots, a_n) = (x_1, x_2, \dots, x_n)$

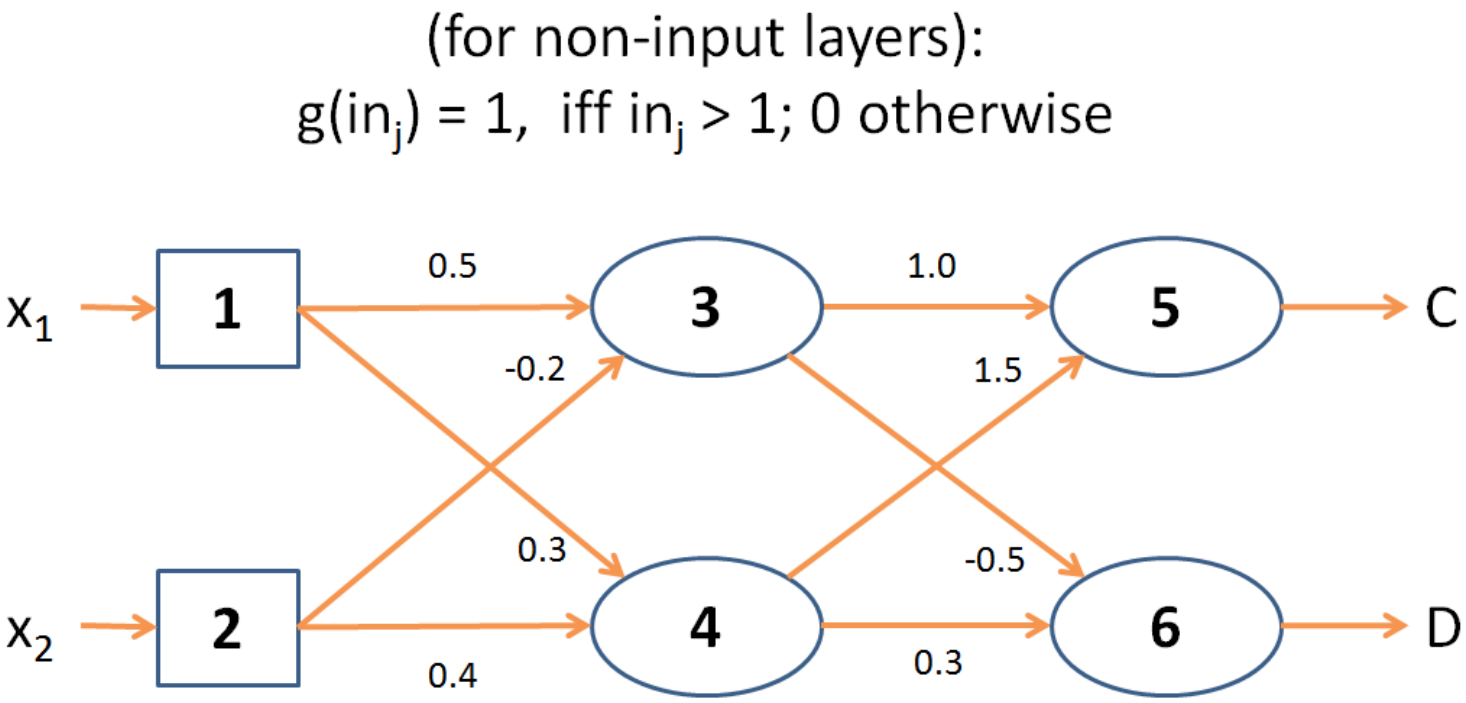
A network's **hidden layer** performs the internal computation for our network's task and are all neurons between the input and output layers.



Here's an example:

Example

Say we had a simple, 3 layer network with 2 inputs, x1 and x2, and then 4 perceptrons (each with activation function g, which is a step function) with various output weights. What is the output of 5 and 6 (i.e., output values C and D), when x1 = 1 and x2 = 2? Assume a 0 weight for the bias input.

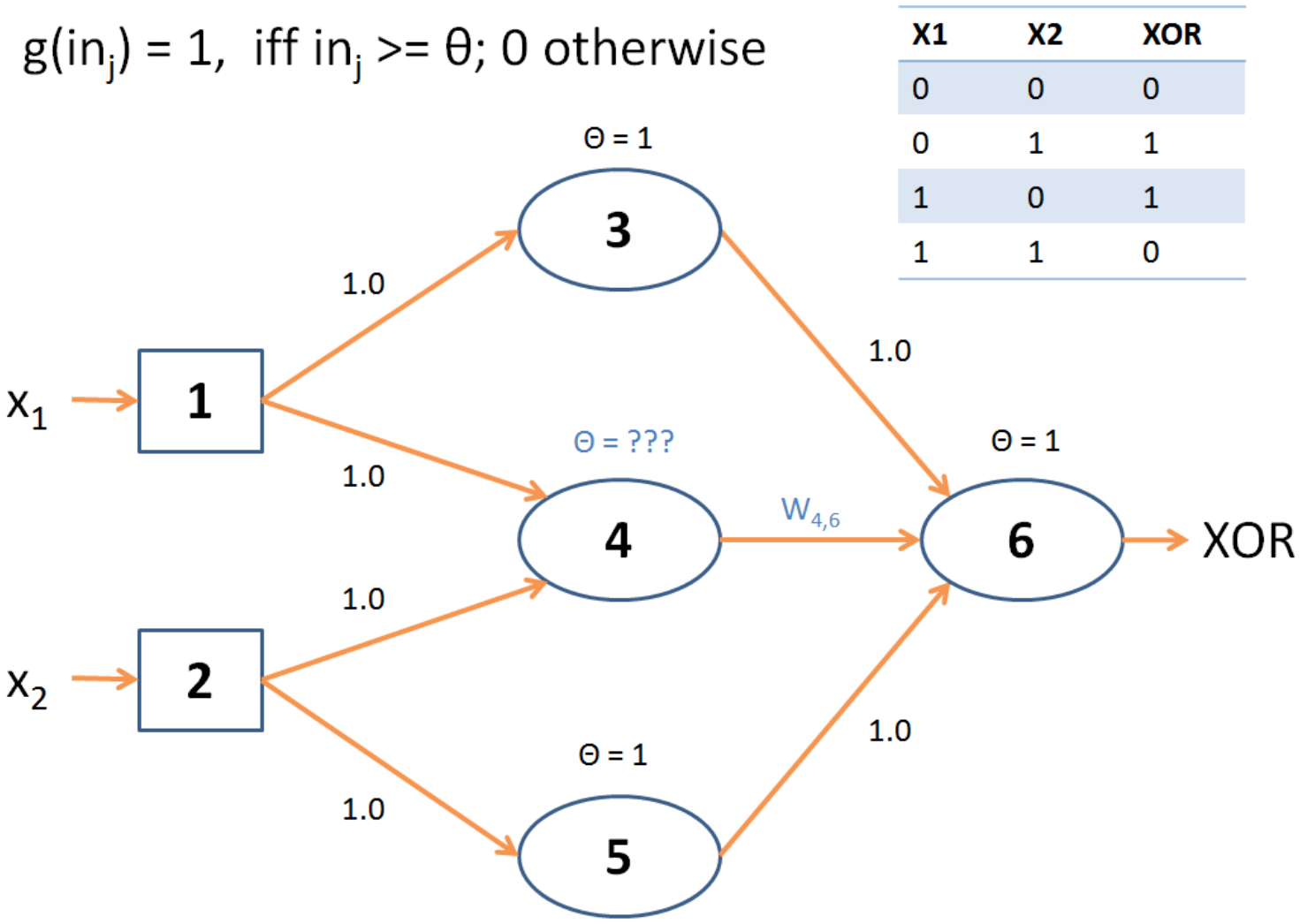


NOTE: In the above, our input layer still consists of neurons who just happen to be set to our input; they're still neurons due to their multiple output links, which isn't the case for the input values x1 and x2.

Now that we know about multi-layer networks, we can solve our XOR problem! Let's do that now.

Example

Complete the following feed-forward perceptron network to elicit the XOR behavior of two binary inputs, x1 and x2. i.e., solve for θ of node 4 and $W_{4,6}$



X1	X2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

❓ Click for solution.

So, we see that our neural *networks* are really just **non-linear regressors** since we can find separations in classification problems that are non-linear so long as we use multiple neurons.

This brings us to the next section: how do we train our networks? What learning tools do we have available to us?

How to Train Your Neurons

One of the most powerful features of Neural Networks is to be able to learn a classification problem.

As a form of supervised learning, we can give our networks a vector of inputs X and a vector of the expected outputs for that input Y, and train our network to make correct output.

ℹ The learning **parameters** (i.e., what we have liberty to change in order to achieve our correct classification) in a neural network are its **weights**

Furthermore, since each weight affects only one other neuron at its link's terminus, for m weights in the network we actually perform m separate learning problems!

So, how do we fine tune our weights in order to get the proper output?

The trick is: we're going to guess!

We'll start by guessing the correct answer, and depending on how wildly we were wrong from the expected answer, we will then **update** our weights.

ℹ **Back-propagation learning** is the learning task whereby we successively update our network weights based on the given, expected outputs for given inputs.

The technique is called "back-propagation" because we always start by comparing our output guess to the actual answer, and then attempt to modify our weights to minimize error.

In this way, we are propagating the output error backwards in levels until we have changed the greatest weight-offenders (that might have led to a wrong answer) to their correct values.

One thing to note: although we're attempting to discover the proper weights that are appropriate given our network, we have no means of determining the actual network structure.

ℹ **Cross validation** techniques allow us to try several putative models, run back-prop and determine which had the best performance.

But, for now, we'll just assume that we had settled on some sort of network structure and now need to find the proper weights.

The backprop strategy is interested in the derivatives of our activation functions, and will use these to find a sort of gradient descent to minimizing error.

ℹ For this reason, the activation functions of our nodes must be **differentiable.**

Luckily, logistic functions have a nice derivative that will prove handy for an example. Before we look at the algorithm, a review of notation:

- $w_{i,j}$ = the weight associated with a link from node i to node j
- x_i = an input from a training data point's input vector corresponding to input node i
- in_j = the sum of weighted inputs to node j
- $g(in_j)$ = the activation function for node j which is a function of the input weighted sum
- a_j = the output of node j equivalent to g(in_j)
- $g'(in_j)$ = the derivative of the activation function of node j
- y_j = an output from a training data point's output vector corresponding to output node i
- $\Delta[j]$ = the error term associated with node j, i.e., how distant its output is compared to the expected one
- α = the **learning rate**, which is usually a small number like 0.01, which dictates how quickly our weights will change based on updates

Let's take a look at the algorithm and then start off an example in gory detail.

```
; Takes the network structure (network) and a set
; of training data (examples) to return a network
; with proper weight parameters set

function BACK-PROP-LEARNING (examples, network)
local variables: Δ, a vector of errors indexed by node

repeat
    for each weight w_i,j in network do
        ; Seed network with our random guesses
        w_i,j ← a small random number

    ; Example data point from training set with
    ; input vector x and expected output y
    for each example (x, y) in examples do

        ; Propagate the inputs, x, forward to
        ; obtain our outputs, and then compare to
        ; the expected
        for each node i in the input layer do
            a_i = x_i

        ; Go through each hidden layer up to
        ; the output one, L
        for layer = 2 in L do

            ; Compute the inputs to every node
            for each node j in layer do
                in_j ← Σ_i w_i,j * a_i
                a_j ← g(in_j)

        ; At this point, we can compare the outputs
        ; that we got to the outputs we expected (y)
        ; and get an error term to store in the delta

        for each node j in output layer do

            ; Here g'(in_j) is the derivative of the
            ; activation function g(in_j)
            Δ[ j ] ← g'(in_j) * (y_j - a_j)

        ; Populate the errors of each node in the hidden
        ; layer
        for layer = L - 1 to 1 do
            for each node i in layer do
                Δ[ i ] ← g'(in_i) * Σ_j w_i,j * Δ[ j ]

        ; Finally, update the weights, where α is our
        ; learning rate constant
```

```
        for each weight w_i,j in network do
            w_i,j ← w_i,j + α * a_i * Δ[ j ]

until (MAX_ITERATIONS reached) OR
      (Global error over Δ below some minimum)

return network
```

Whew! That's a big algorithm! So notationally heavy!

Care to just glance at a small example (and not all of one, just a few steps)

Consider the following network and training sample where each activation function is the logistic function:

Logistic function:

$$g(in_j) = \frac{1}{(1 + exp(-in_j))}$$

Derivative of logistic function:

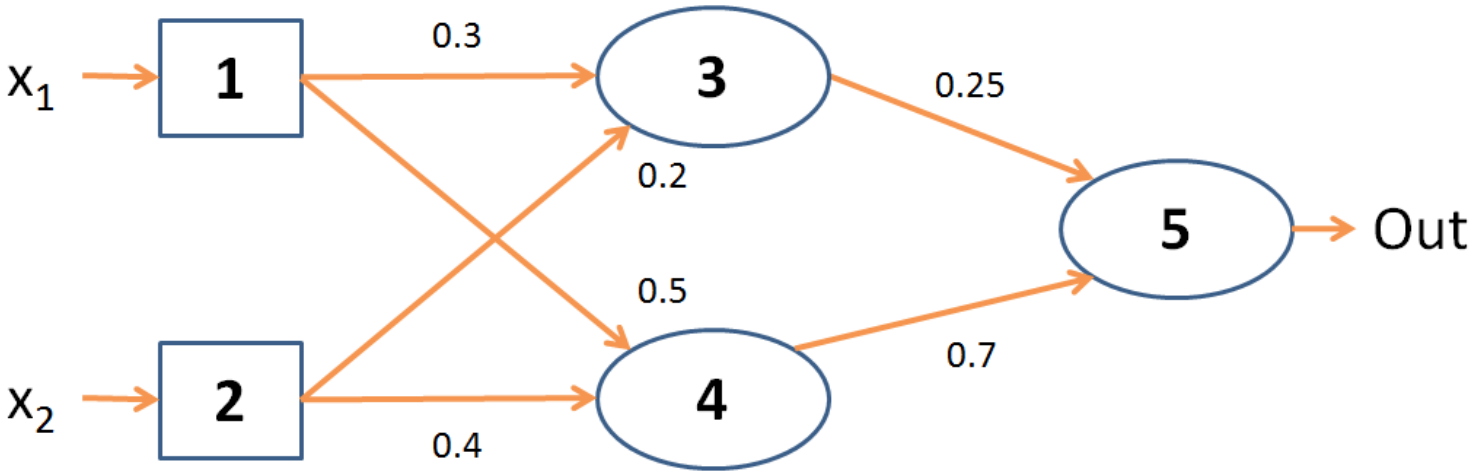
$$g'(in_j) = g(in_j) * (1 - g(in_j))$$

Now, consider the following (arbitrary) training set with input vector X and expected output vector Y.

Case	**X**	**Y**
1	{0.5, 1.0}	{0.2}
2	{0.3, 0.5}	{0.15}
3	{0.9, 0.4}	{0.8}
n	...	...

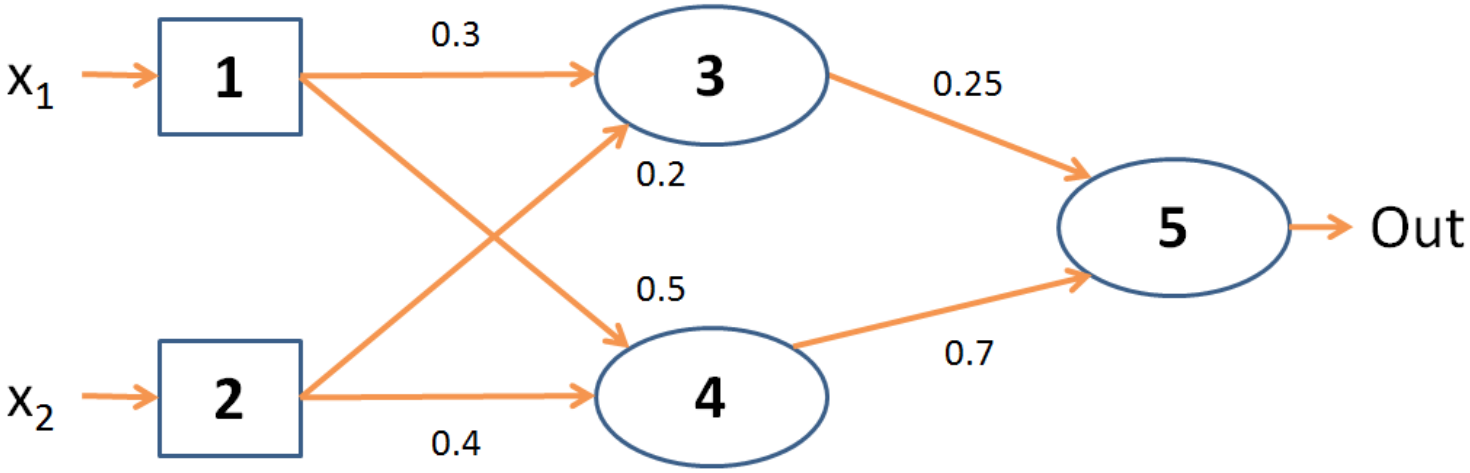
And then we had the following network:

Step One: Seed weights with small random #s



Step Two: Pick sample input X and propagate to Out

$X = \{0.5, 1.0\}$



$in_3 = 0.3 * 0.5 + 0.2 * 1.0 = 0.35;\ g(in_3) = 0.59 = a_3$

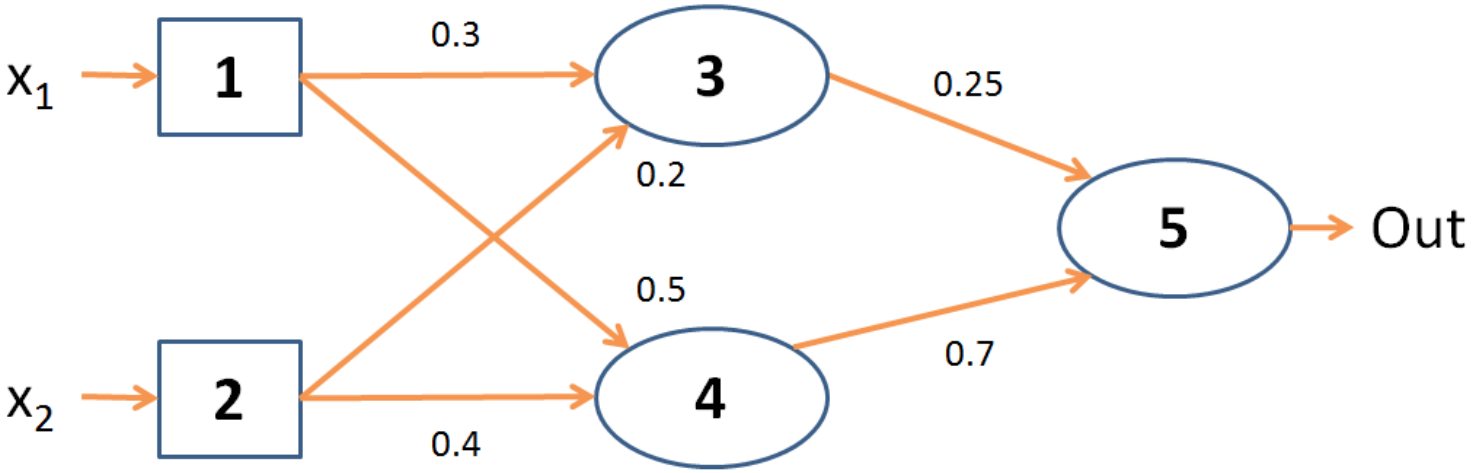
$in_4 = 0.5 * 0.5 + 0.4 * 1.0 = 0.65;\ g(in_4) = 0.66 = a_4$

$in_5 = 0.25 * 0.35 + 0.7 * 0.65 = 0.54;\ g(in_5) = 0.63 = a_5$

*Not shown: Bias node and weight (avoiding clutter)

Step Three: Find error in output compared to Y

Y = 0.2; Out = 0.63



$\Delta[5]$

$= g'(in\_5) * (y - Out)$

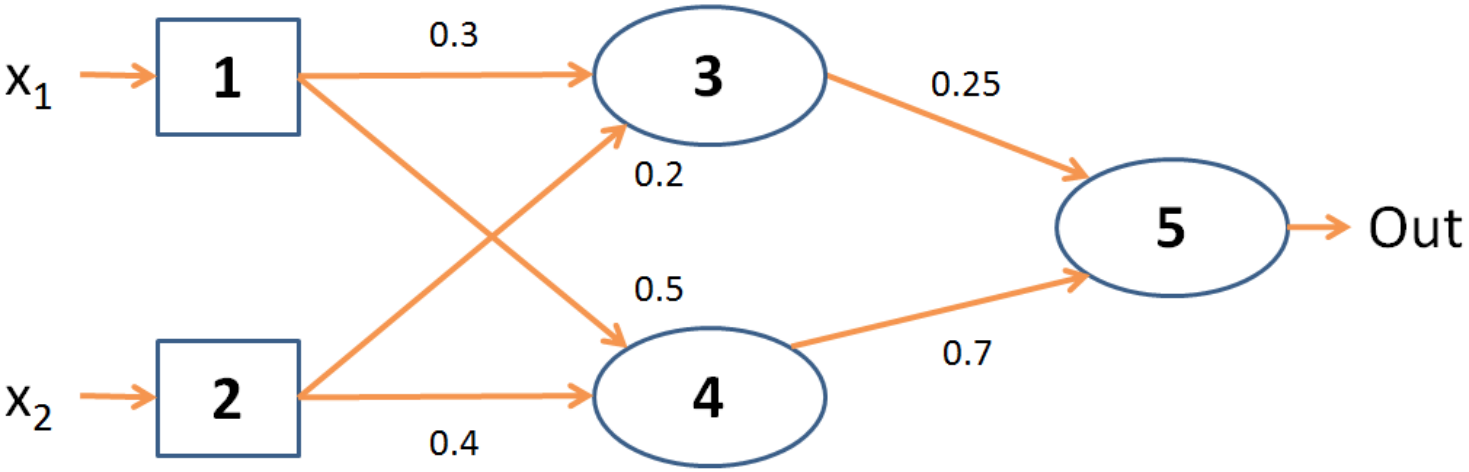
$= g(in\_5) * (1 - g(in\_5)) * (y - Out)$

$= 0.63 * (1 - 0.63) * (0.2 - 0.63) = -0.1$

*Not shown: Bias node and weight (avoiding clutter)

Step Four: Find deltas for each node (propagate back)!

$$\Delta[i] = g'(in_i) * \Sigma_j w_{i,j} * \Delta[j]$$



$\Delta[3]$ (Do this for every node; not shown here because I'm lazy)

$= g'(in\_3) * w_{i,j} * \Delta[j]$

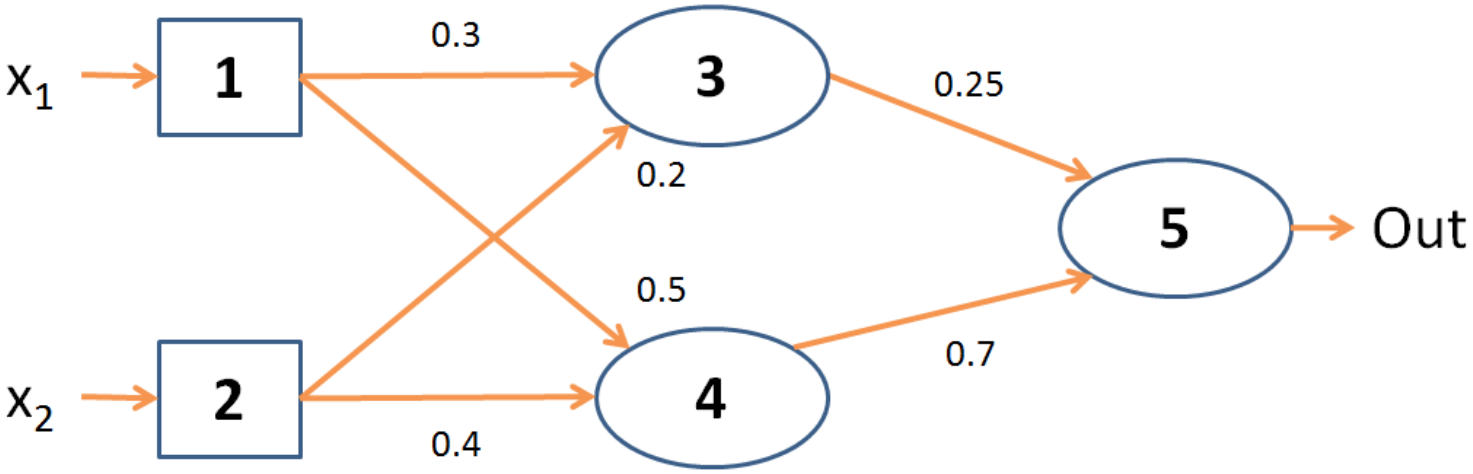
$= g(in\_3) * (1 - g(in\_3)) * w_{3,5} * \Delta[5]$

$= 0.59 * (1 - 0.59) * 0.25 * (-0.1) = -0.01$

*Not shown: Bias node and weight (avoiding clutter)

Step Five: Update weights

$$w_{i,j} = w_{i,j} + \alpha * a_i * \Delta[\, j \,]$$



$w_{3,5}$ (Do this for every weight INCLUDING bias; not shown here because I'm lazy)

$= w_{3,5} + \alpha * a_3 * \Delta[\, 5 \,]$

$= 0.25 + 0.1 * 0.59 * (-0.1)$; for $\alpha = 0.1$, (big α)

$= 0.244$

*Not shown: Bias node and weight (avoiding clutter)

Wow... that is just... wow.

Let's let that simmer for a bit...

At the end of this whole process, we end up with a network whose weights have hopefully been adjusted to fit our output requirements!

> This is why, like Naive Bayes, neural networks are members of the **Probably, Approximately Correct (PAC)** algorithms and structures.

We can train to our sample data enough to generalize but we don't want to overtrain so we lose all flavor of generalizability.

Knowledge in Learning

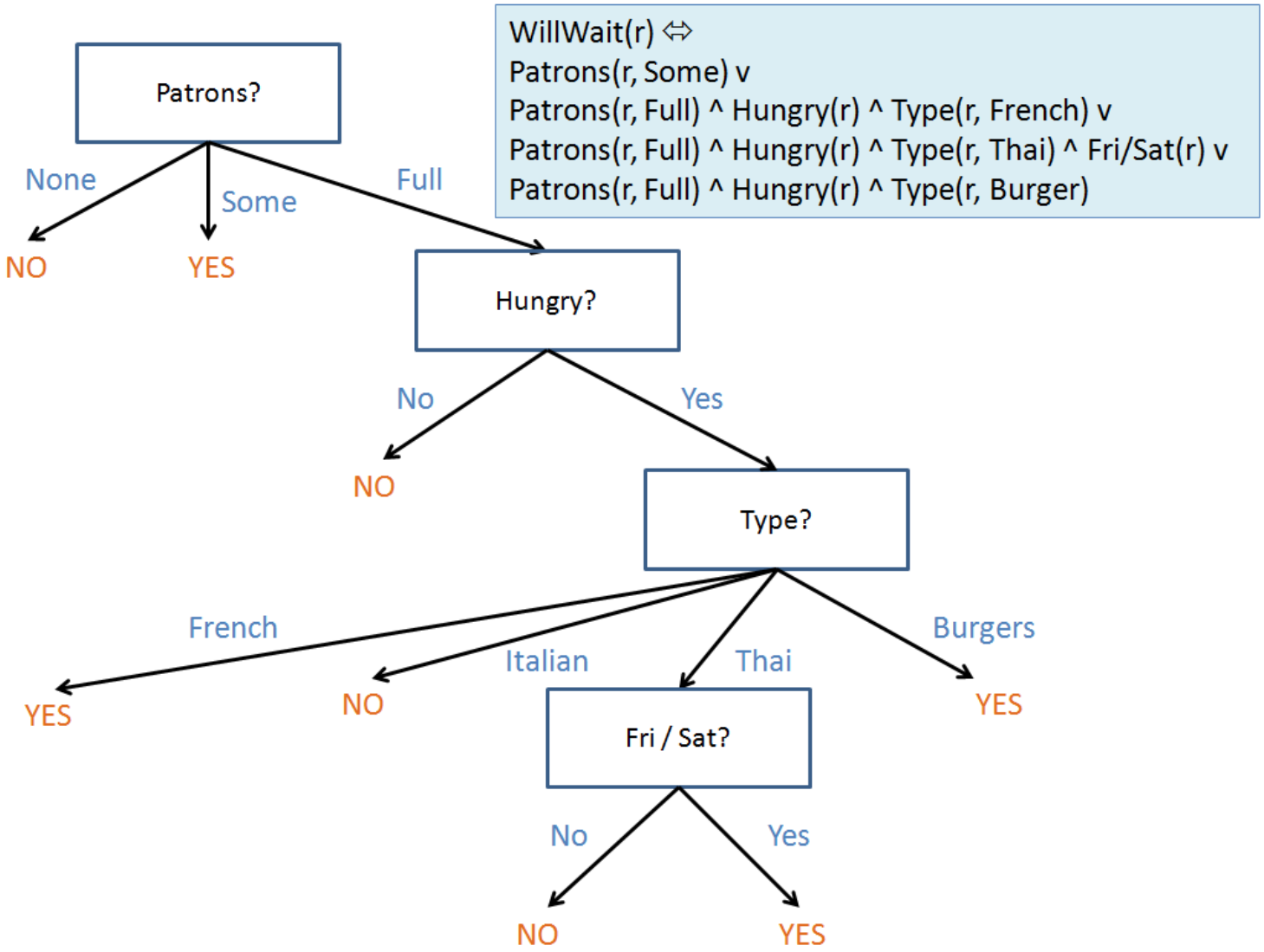
Our previous efforts to tackle learning problems has been data-driven and simply formulated around the examples in our training sets themselves.

However, what we haven't previously asserted is any knowledge that we might have a priori of seeing the data and converting our problems into FOL formulations.

> A **hypothesis** is a FOL description of the set of attributes that satisfy a positive classification on some learning task.

For example, remember our Dining Dillema example with decision trees as to whether or not one would wait for a seat?

Consider the following conversion from decision tree to FOL hypothesis:



Here, in our FOL predicates, r represents a sample from the training set, and the predicates indicate whether or not that sample r has the attribute in question.

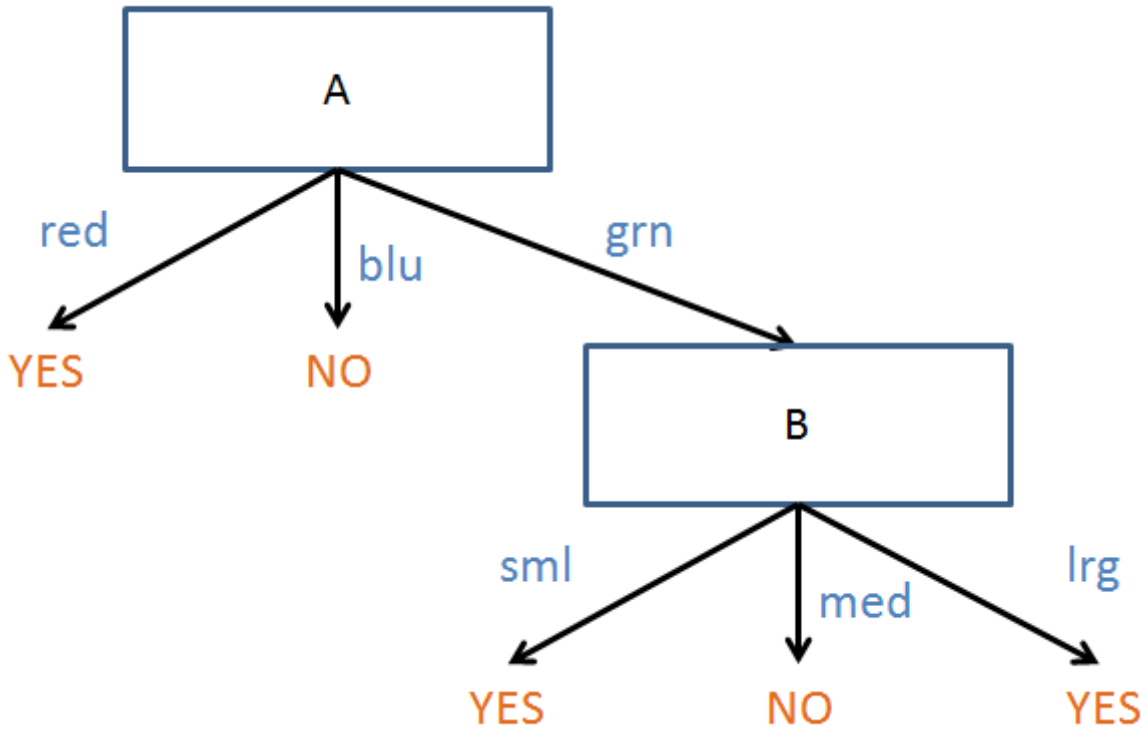
So, in general, our hypotheses are in the form:

```
Classification(r) ⇔ DNF

; ...for some CNF that might look like:

Classification(r) ⇔ Pred1(r) ∧ Pred2(r) ∧ ... ∨
                    Pred1(r) ∧ Pred3(r) ∧ ... ∨
                    ...
```

ℹ The terms of the DNF (disjunctive normal form) sentence that compose the hypothesis are called the hypothesis' **extensions**, which each cover the positive classifications of some number of disjoint sample examples.

As such, two hypotheses with at least one different extension will be inconsistent since they must disagree on the classification of at least one sample element.

Example
☑ Generate a FOL hypothesis representation of the following decision tree for some classification predicate $X(r)$ on sample r and trinary attributes A and B.



Click for solution.

As we iterate through training set examples, we can modify our hypotheses in one of two ways:

Generalization extends the number of positive classifications to samples by dropping a condition from an extension.

```
; Initial hypothesis:
X(r) ⇔ A(r, red) ∧ B(r, sml)

; Generalization: drop condition:
X(r) ⇔ A(r, red)
```

Specialization reduces the number of positive classifications to samples by adding a condition to an extension.

```
; Initial hypothesis:
X(r) ⇔ Patrons?(r, full)

; Specialization: add condition:
X(r) ⇔ Patrons(r, full) ∧ Hungry?(r)
```

Therefore, if we're interested in finding a hypothesis that suits our training set, we can iterate through each sample and attempt to modify our best hypothesis to do the job.

ℹ The **current best hypothesis** tries to maintain a single hypothesis by adjusting for inconsistent classifications that might arise with each sample element.

A crude outline of the algorithm goes something like:

```
function Current-Best-Learning(examples, h)
    for each ex in examples
        if ex consistent with h
            continue
        else ex false positive for h
            ; Here we try to perform specialization on h,
            ; which must satisfy not only ex but all
            ; previous examples
            try to specialize h, or return fail
              if we cannot
        else ex false negative for h
            try to generalize h, or return fail
              if we cannot
```

Some things to note:

- Very computationally expensive to not only specialize and generalize according to a new inconsistent sample, but also to double check that a modification correctly works for all previous examples
- Might have to back-track to a choice point to generalize / specialize since there are often multiple plausible attribute splits that might satisfy an example
- In light of multiple-viable attribute splits, we would simply choose one non-deterministically and hope that we chose correctly for future examples

Let's try it once on a very simple example:

Example
☑ Find a hypothesis that meets the following example training set for classification task X. Use the current best hypothesis algorithm.

A	B	C	X?
red	sml	0	Yes
red	med	1	No
grn	sml	0	No
red	med	0	Yes

Now we go example by example attempting to generate a hypothesis:

Example 1

A	B	C	X?
red	sml	0	Yes

❓ Give a hypothesis h_1 over the first example on attribute A of the format:

$$h_1 : X(r) \Leftrightarrow ???$$

Now continue to next example:

Example 2

A	B	C	X?
red	med	1	No

❓ Give a hypothesis h_2 modifying $h_1 : X(r) \Leftrightarrow A(r, red)$ over the first TWO examples on attributes A and B of the format:

$$h_2 : X(r) \Leftrightarrow ???$$

So far so good, demonstrably h2 now satisfies the classification outcome of examples 1 and 2... let's keep going...

Example 3

A	B	C	X?
grn	sml	0	No

❓ Is Example 3 consistent with $h_2 : X(r) \Leftrightarrow A(r, red) \wedge B(r, sml)$? Do we need to specialize or generalize?

Example 4

A	B	C	X?
red	med	0	Yes

❓ Give a hypothesis h_3 modifying $h_2 : X(r) \Leftrightarrow A(r, red) \wedge B(r, sml)$ over the first FOUR examples on attributes A, B, and C of the format:

$$h_3 : X(r) \Leftrightarrow ???$$

That looks like it fits! Notice, however, that only the latter extension is required to match all of our classifications... so current-best will detect that on a recursive call (see book pg. 771) and reduce to:

$$h4 : X(r) \Leftrightarrow A(r, red) \wedge C(r, 0)$$

Looks good!

Like we said, however, there are many efficiency concerns with this nondeterministic attribute selection, since *many* different hypotheses might satisfy the sample that we have.

Using Relevance Information

Now comes the part where we can actually assert some knowledge on our hypothesis search by limiting the attributes that might contribute to predicting a classification:

> **Determinations** assert which attributes are truly predictive of what other attributes or classifications.

Sometimes these are called **functional dependencies** since they assert that classifications are simply a "function" of a set of attribute settings.

```
; We denote a determination by the syntax:
Predictor1 ^ Predictor2 ^ ... ^ PredictorN > Classification

; For example, we know that Conductance of
; a particular material is a function of its
; Material type and Temperature:

Material(x, m) ^ Temperature(x, l) > Conductance(x, ϱ)

; In our last example:

A(r, a) ^ C(r, c) > X(r)
```

As such, we'd be interested in, given some training set over some number of attributes, which attributes are determinations of the classification.

Furthermore, if there does exist some set of attributes that are determinations for the classification, which has the fewest attributes?

The brute-force approach, described in the book on page 786, is simply to attempt to assemble sets of attributes that correctly predict the classification, starting with sets of size 1 and working up from there.

Naturally, this algorithm's complexity is a function of the size of the minimal determination (the number of attributes in it, let's say p) and n, the total number of attributes: O(n^p)

Odds & Ends

Just to comment on the philosophical analyses of artificial intelligence, John Searle's Chinese Room Argument poses a thought-provoking glance into the boundaries of intelligence.

Questions about just what it meant for a machine to be "intelligent" arose after Turing's famous test for machine intelligence was posed, and continue to this day.

The Chinese Room scenario was posed by Searle in response to Turing's assertion that "intelligence" can be measured by the yardstick functionalism:

> A **mental state** is the abstract notion of a thought.

> **Functionalism** asserts that a mental state is any intermediate causal condition between input and output.

Example
Consider being given some visual input like seeing a flower, which evokes some thought about the input (the mental state), and then produces some behavioral output (like picking the flower).

In this capacity, if we were able to surgically replace all of the neurons in someone's brain with devices that reproduced the exact input-output of the neurons they replaced, then functionalists would argue that the mental state (i.e., the thought aspect of consciousness) remains unchanged.

Searle, and other **biological naturalists** like him, argue that consciousness would be disrupted, and someone who has undergone the above procedure would slowly lose their capacity for control.

> **Biological naturalists** claim that mental state is not simply a neuronal-level product, but rather a higher level consequence of **patterns** of neural activation and unspecified properties of neurons.

In other words, the **high-level** notion of thought is a result of a bunch of separate interactions of the **low-level** neuronal activations.

Furthermore, there is something in the **unspecified** properties of neurons that allow for the notion of thought that we cannot encapsulate in a machine replacement.

The big difference in the two perspectives is this:

- Functionalists are OK with behavioral evidence of intelligence so long as a machine can reproduce human-like responses to input.
- Biological naturalists claim that, although the input / output might be replicable, the mental state is not. As a tag phrase: "Brains (the physical) cause minds (thought)"

Put simply, machines might one day be able to mimic a human's behavior, but Searle claims that they'll never be able to rationalize why they are doing so or to think like a human does.