Machine Learning I

Lecture 11: Artificial Neural Networks

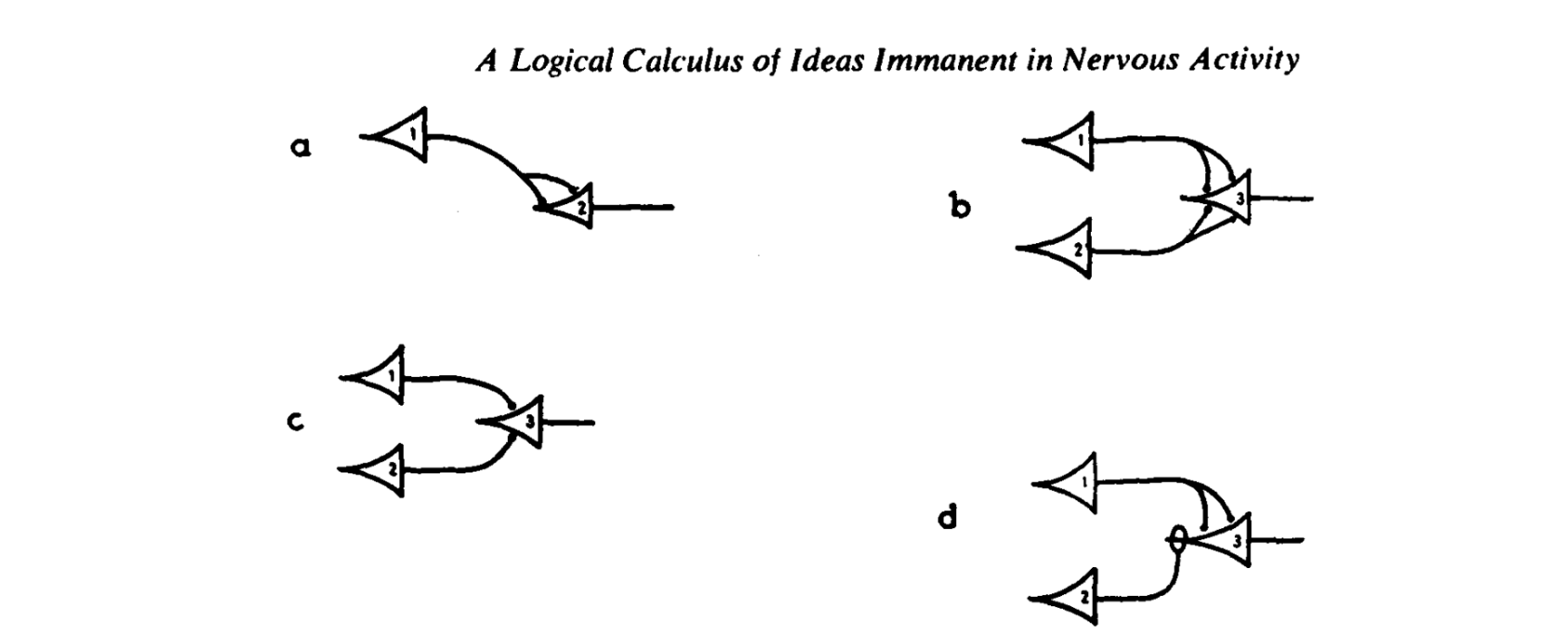
Nathaniel Bade

Northeastern University Department of Mathematics

Table of contents

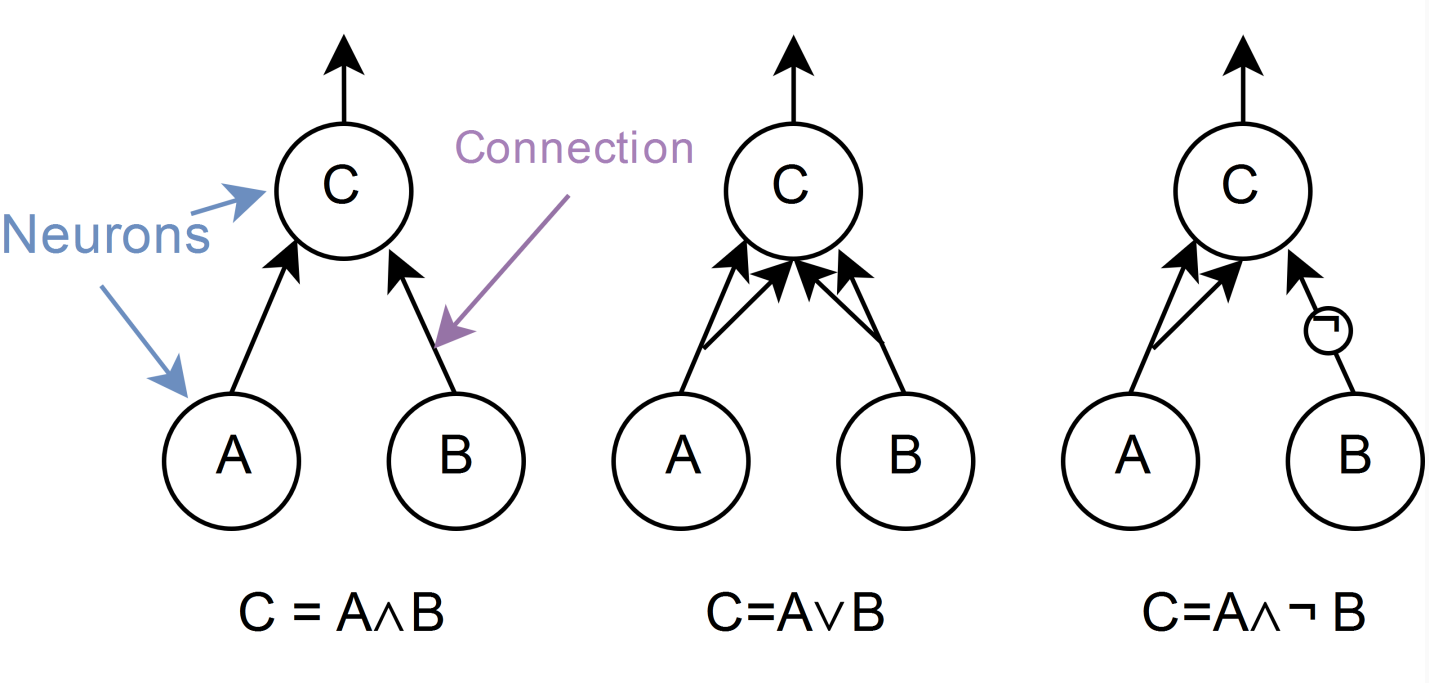
Artificial Neural Networks

Neural Networks



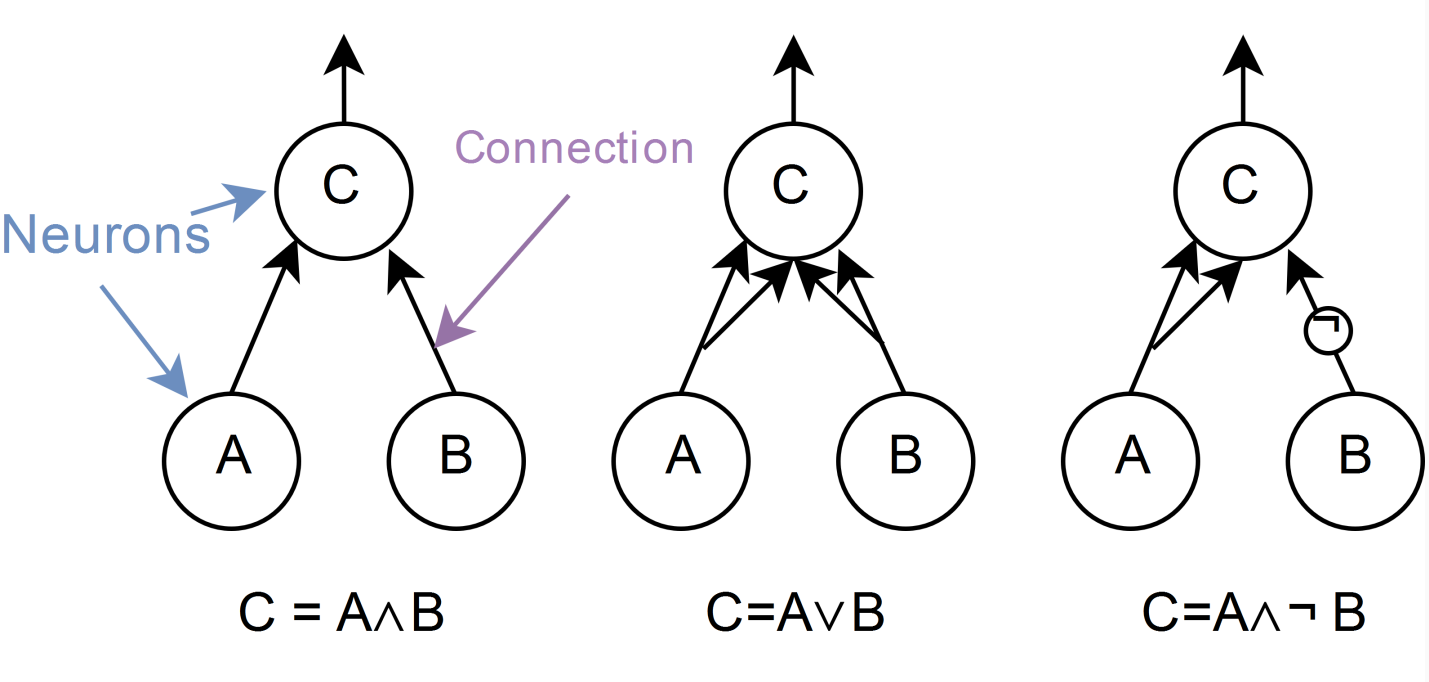
Neural Networks are actually quite old, being introduced in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts to model neurons in the brain using electrical circuits.

Neural Networks as Logic Gates



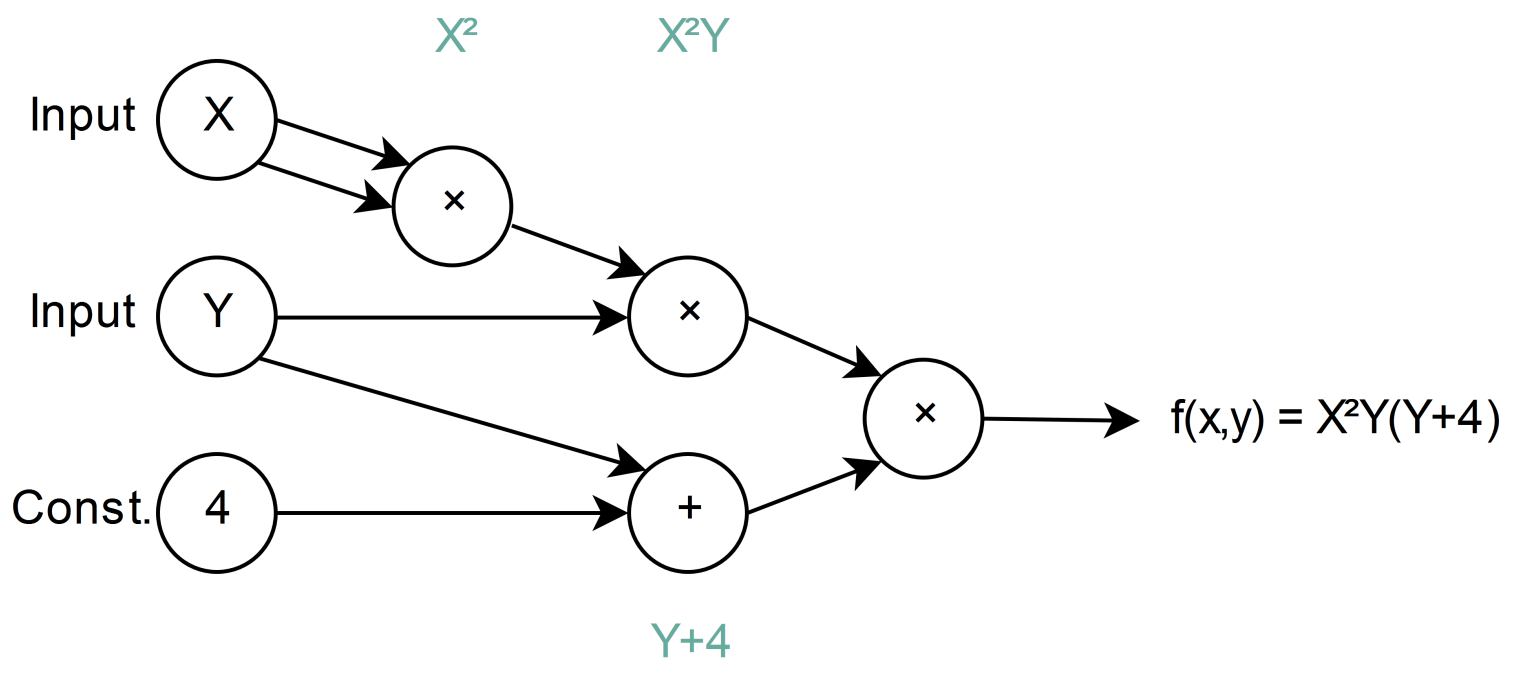
The original neural networks were directed graphs. Each node has a state **on** (firing) or **off** (no firing). Node's also have a threshold t and only fire if more than t imputing nodes are firing.

Neural Networks as Logic Gates



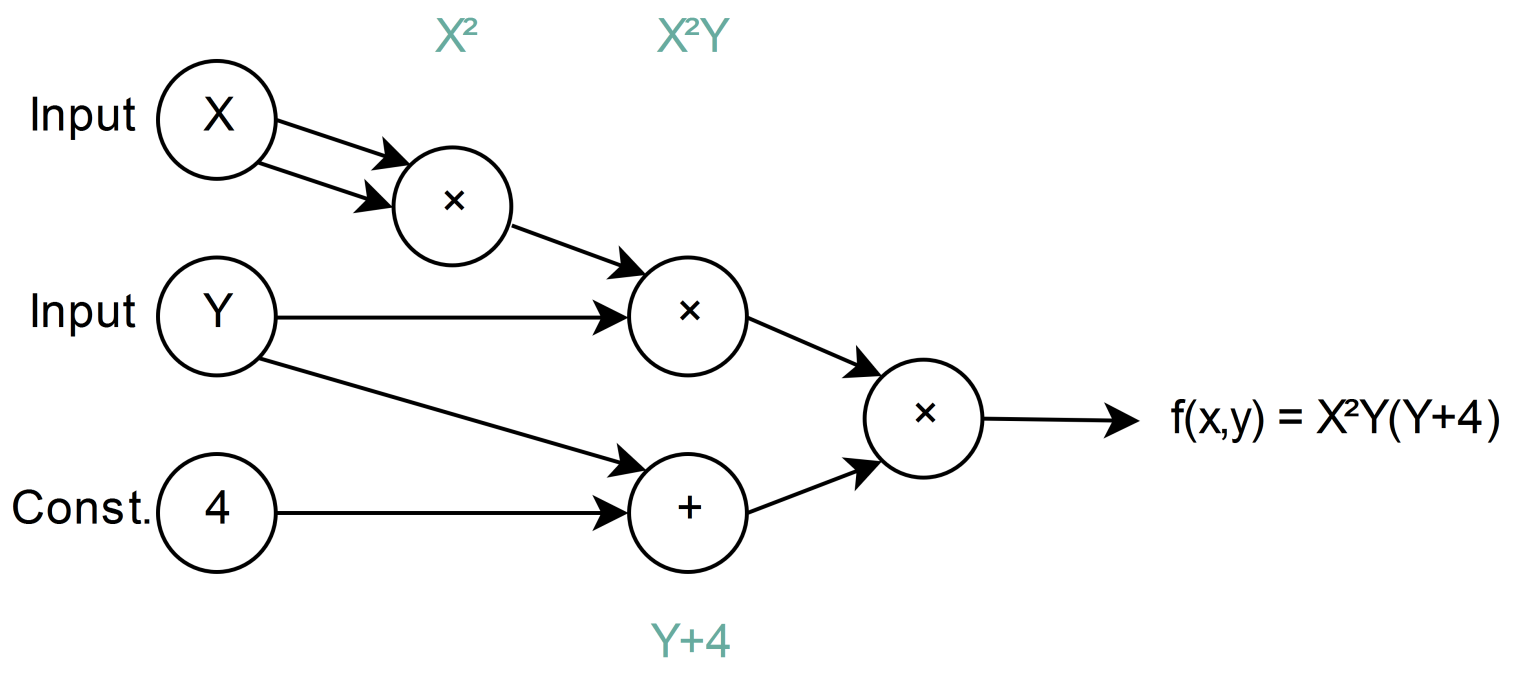
McCulloch and Pitts showed that the standard operations of propositional logic could be built out of strings of such neurons. In the third case, the $\neg$ is a inhibitor connection which stops C from firing. Such connections do appear in biological neurons.

Neural Networks as Operation Trees



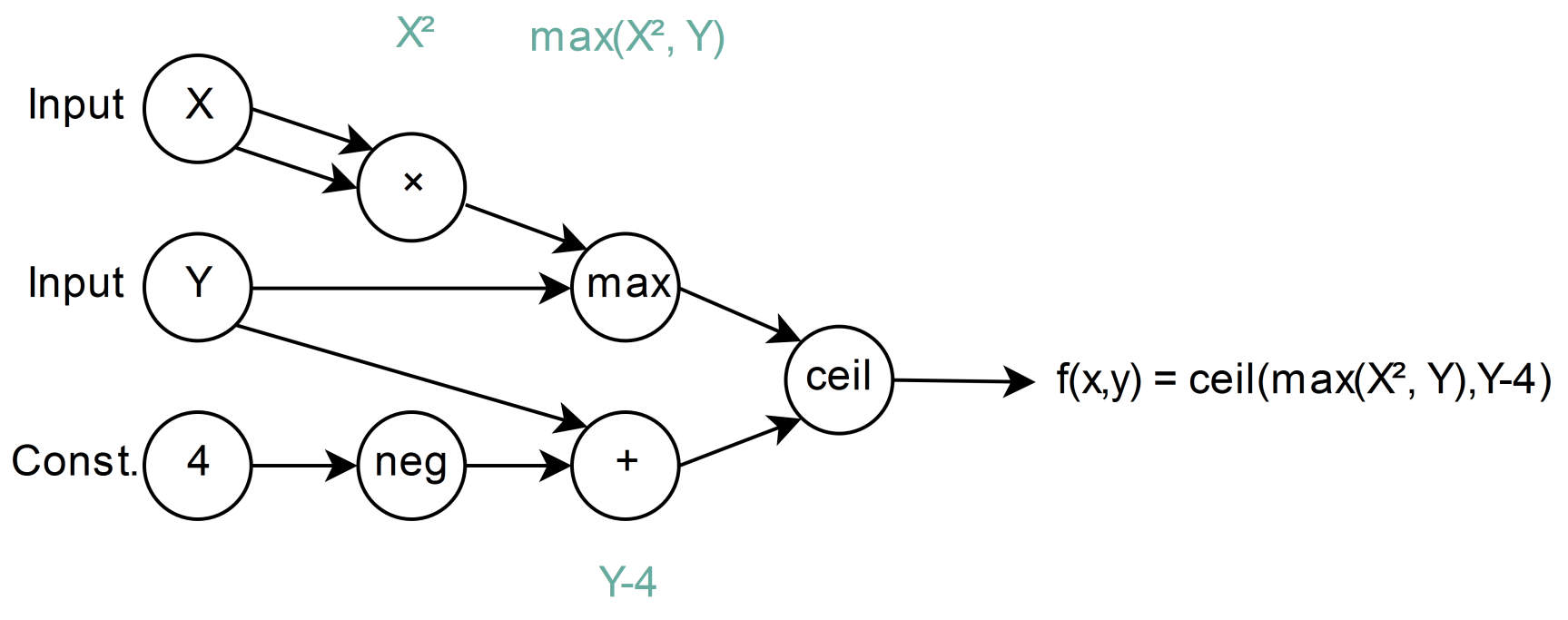
Another view of neural networks (the view most implementations take) is as operation trees. In an operation tree, one constructs a function by giving a sequential list of instructions. Each node represents an operation with a fixed number of inputs and outputs. In an operation tree there is no threshold, and the nodes may output any data type.

Neural Networks as Operation Trees



Viewing neural networks as operation trees give us an intuitive way to build functions out of concrete operational pieces. Furthermore, depending on the allowed nodes it gives a clear representation of nondifferentiable functions.

Neural Networks as Operation Trees



Viewing neural networks as operation trees give us an intuitive way to build functions out of concrete operational pieces. Furthermore, depending on the allowed nodes it gives a clear representation of nondifferentiable functions.

Neural Networks as Operation Trees

Its important to point out that these are two wildly different objects mathematically and philosophically:

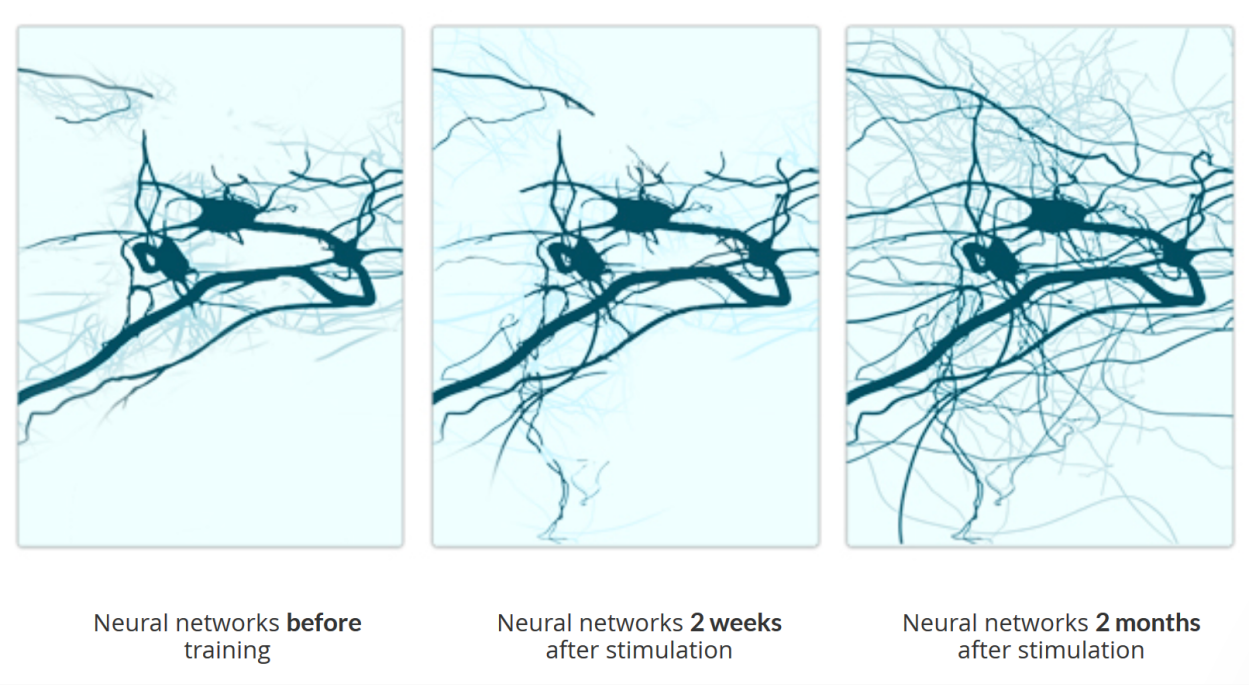
Neural Networks:

- Finite number of types of nodes.
- Binary input and output.
- Interesting behavior comes from topologically.
- Idea: To show how a brain can be constructed from simple pieces.

Operation Trees:

- Nodes are arbitrary functions.
- Arbitrary input and output.
- Topologically trivial as graphs.
- Any function can be constructed so no philosophical interest at full generality.

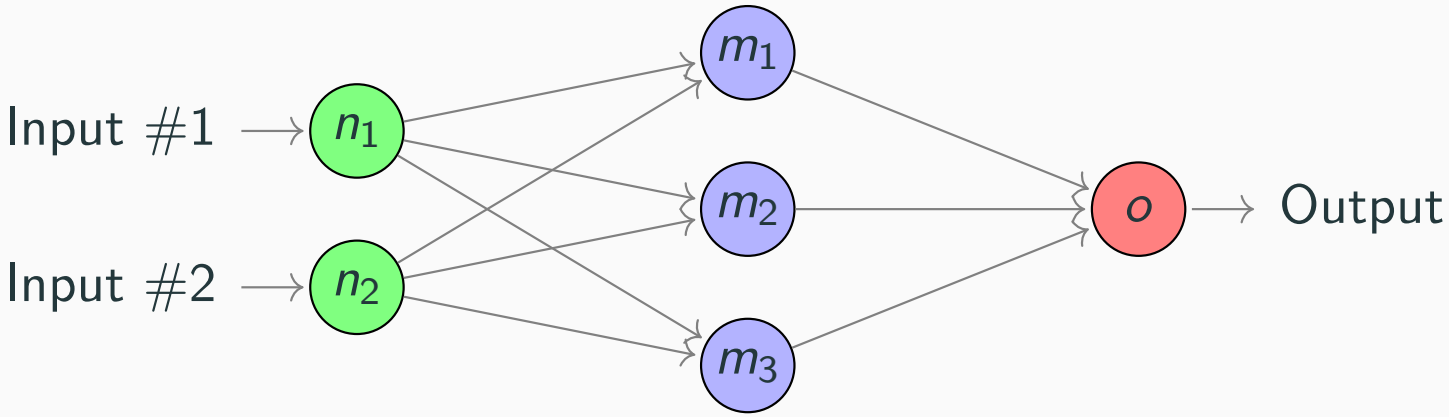
Towards Modern Neurons



In 1949, psychologist Donald Hebb pointed out the neural pathways are strengthened each time they were used, a concept which gave a potential mechanism to human learning.

Towards Modern Neurons

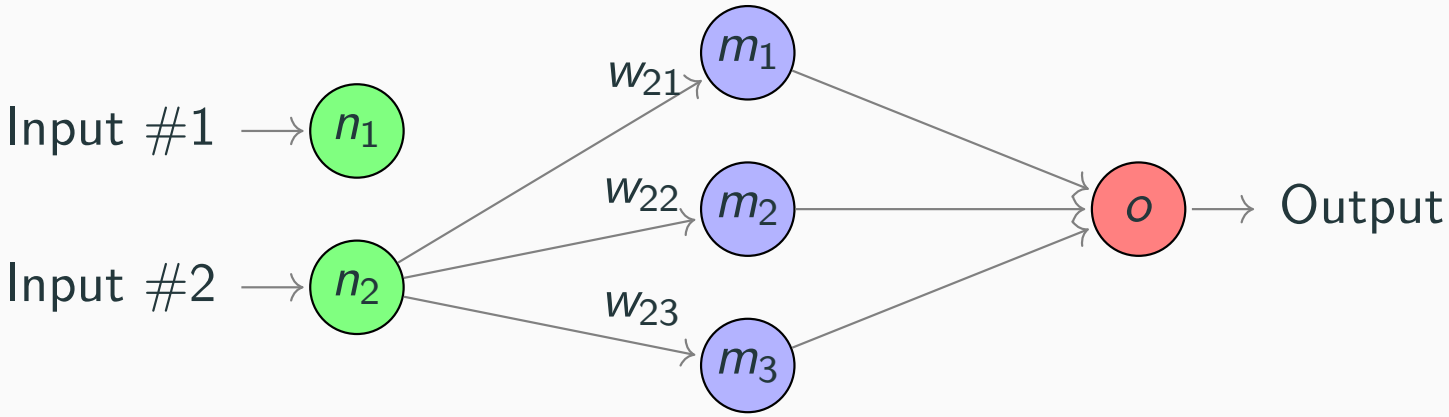
Simple Neural Network



In 1949, psychologist Donald Hebb pointed out the neural pathways are strengthened each time they were used, a concept which gave a potential mechanism to human learning. Practically, this means that each of the edges connecting the nodes can carry a multiplicitive weight which appropriately scales the input.

Towards Modern Neurons

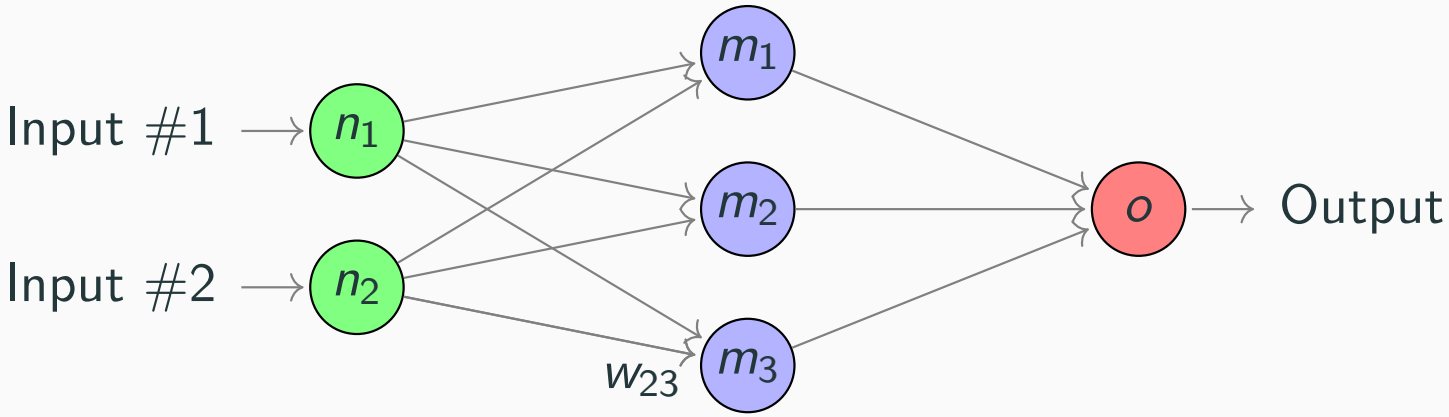
Simple Neural Network



While the output was still binary, the weights allowed the connections to be evaluated non-symmetrically. In addition to the weights, these early neural networks incorporated a **Bayesian** firing mechanisms. Both the weights and the firing probability were increased each time the pair of nodes successfully fired.

Towards Modern Neurons

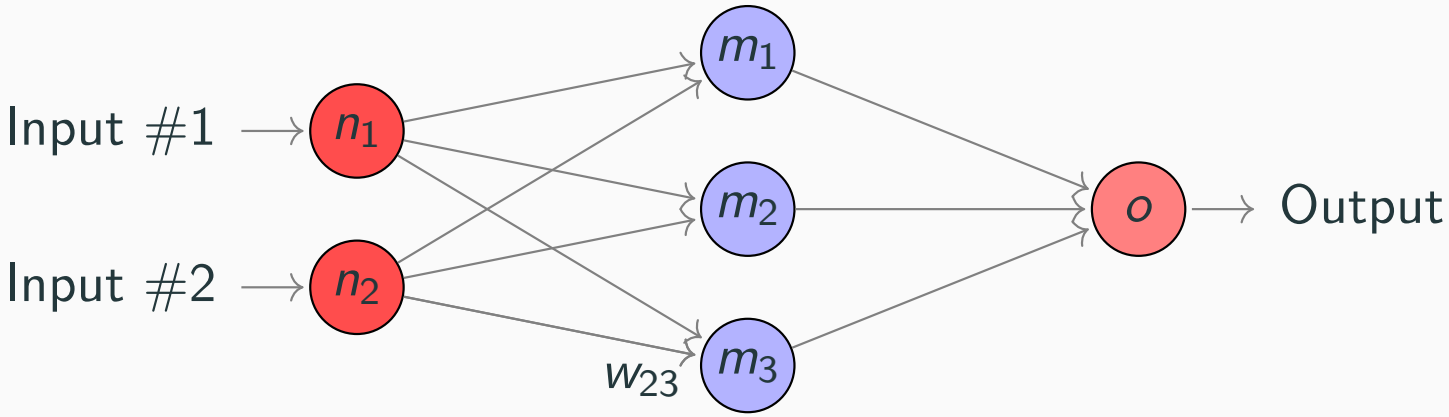
Simple Neural Network



While the output was still binary, the weights allowed the connections to be evaluated non-symmetrically. In addition to the weights, these early neural networks incorporated a **Bayesian** firing mechanisms. Both the weights and the firing probability were increased each time the pair of nodes successfully fired.

Towards Modern Neurons

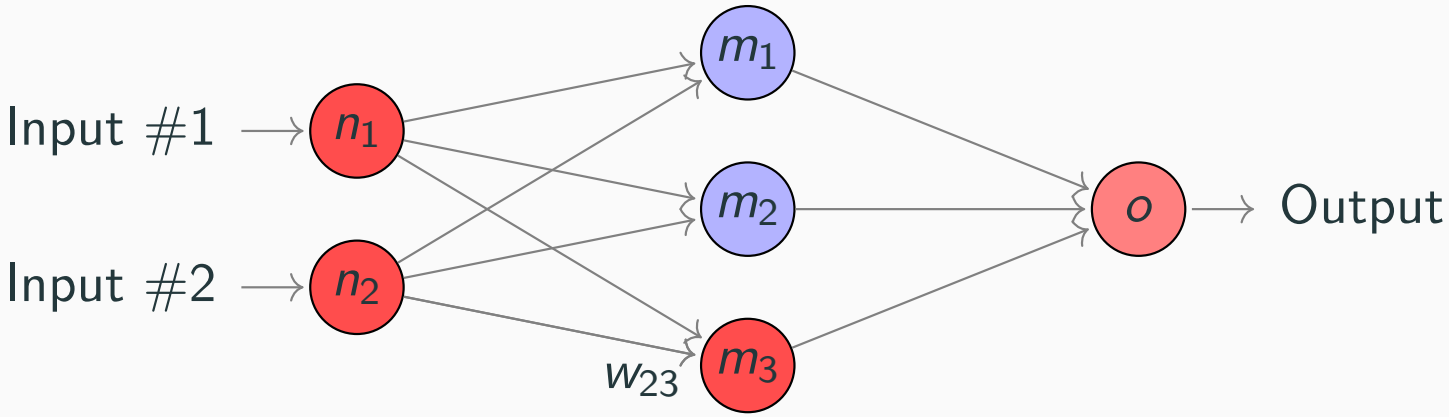
Simple Neural Network



While the output was still binary, the weights allowed the connections to be evaluated non-symmetrically. In addition to the weights, these early neural networks incorporated a **Bayesian** firing mechanisms. Both the weights and the firing probability were increased each time the pair of nodes successfully fired.

Towards Modern Neurons

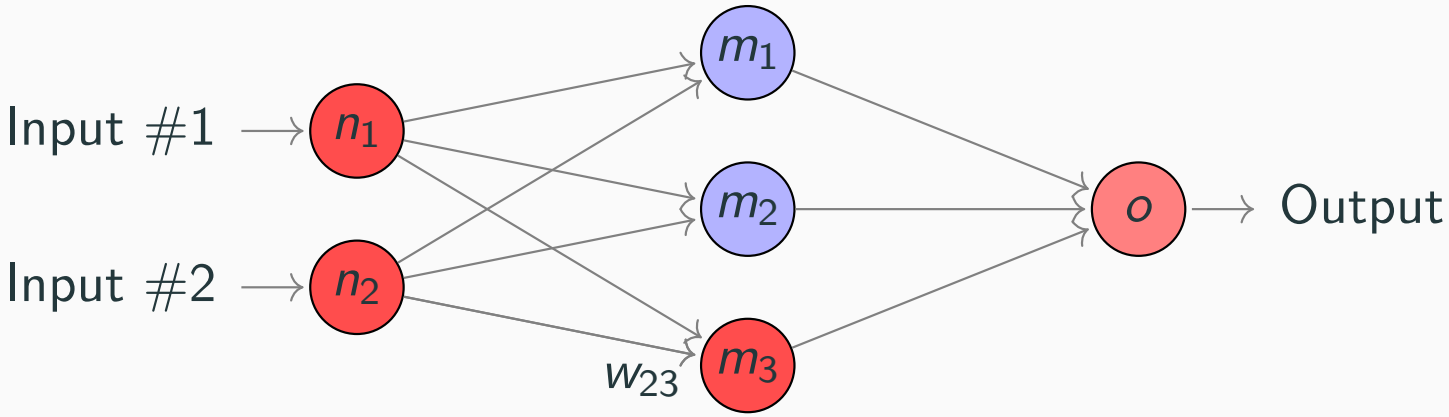
Simple Neural Network



While the output was still binary, the weights allowed the connections to be evaluated non-symmetrically. In addition to the weights, these early neural networks incorporated a **Bayesian** firing mechanisms. Both the weights and the firing probability were increased each time the pair of nodes successfully fired.

Towards Modern Neurons

Simple Neural Network

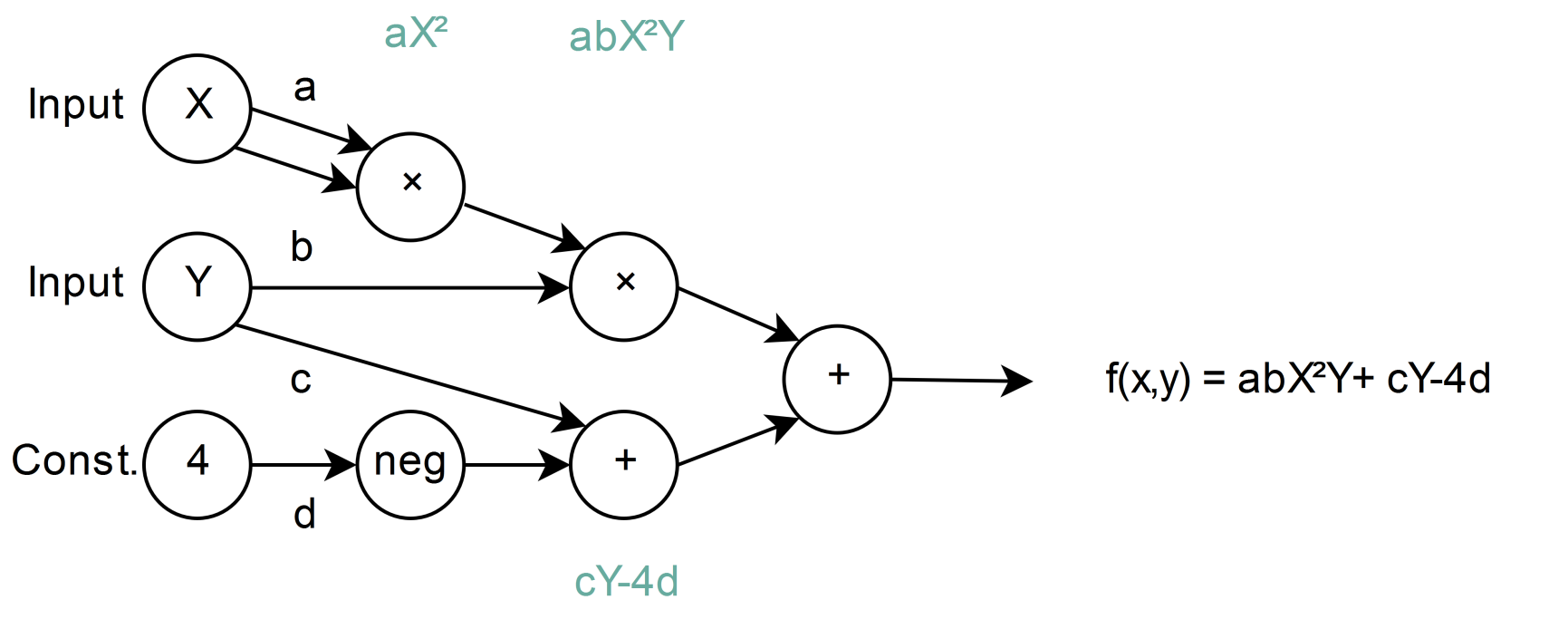


$n_2 \uparrow$, $w_{23} \uparrow$, $m_3 \uparrow$

Here, n_i and m_j represent the firing probabilities. In addition, n_1 and w_{13} would increase in this model.

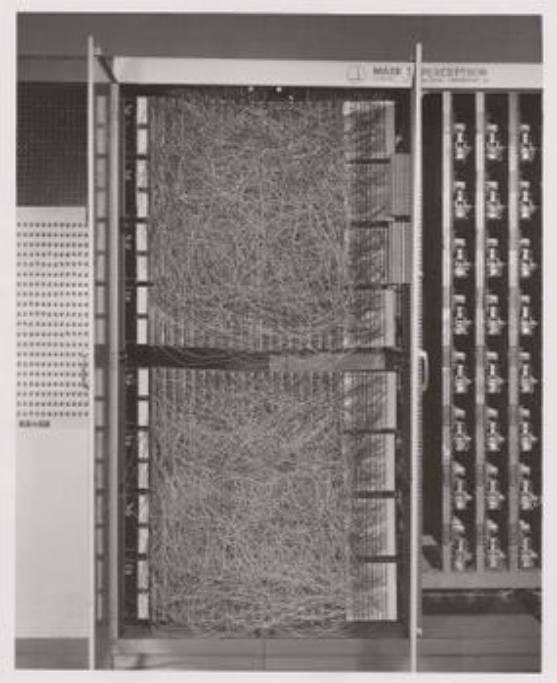
Linear Classifier as a Neural Networks

Towards Modern Neurons



Combining weights with operation trees leads to parameter dependent functions. In the above, the redundant weights have been removed for clarity leaving an expression for $f(X, Y)$ in terms of the variables (X, Y) and the constant parameters a, b, c, d.

Towards Modern Neurons

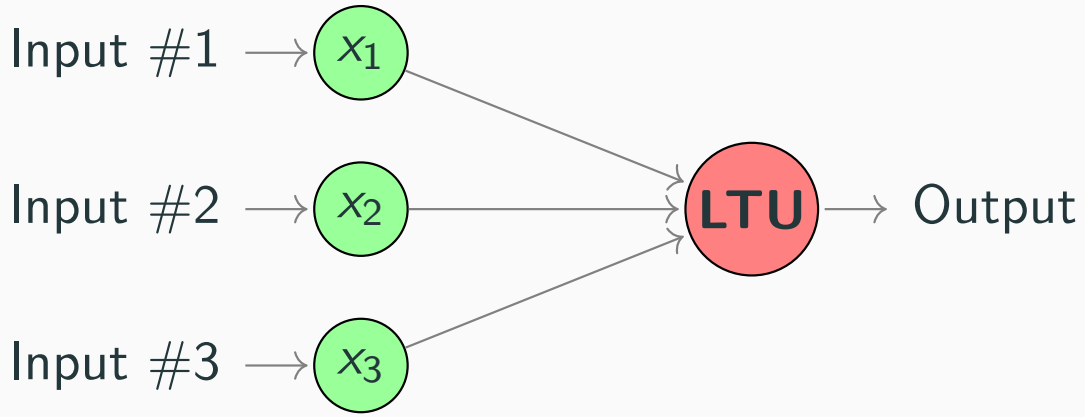


In 1957 Frank Rosenblatt introduced a linear algorithm into the ANN literature called the Perceptron. The perceptron actually entered the world as a hardware device, not as a software algorithms.

In 1957 Frank Rosenblatt introduced a linear algorithm into the ANN literature called the Perceptron. The perceptron is based around a **linear threshold unit** (**LTU**).

Perceptron

Simple Neural Network



The LTU sums the inputs and then applies a step function:

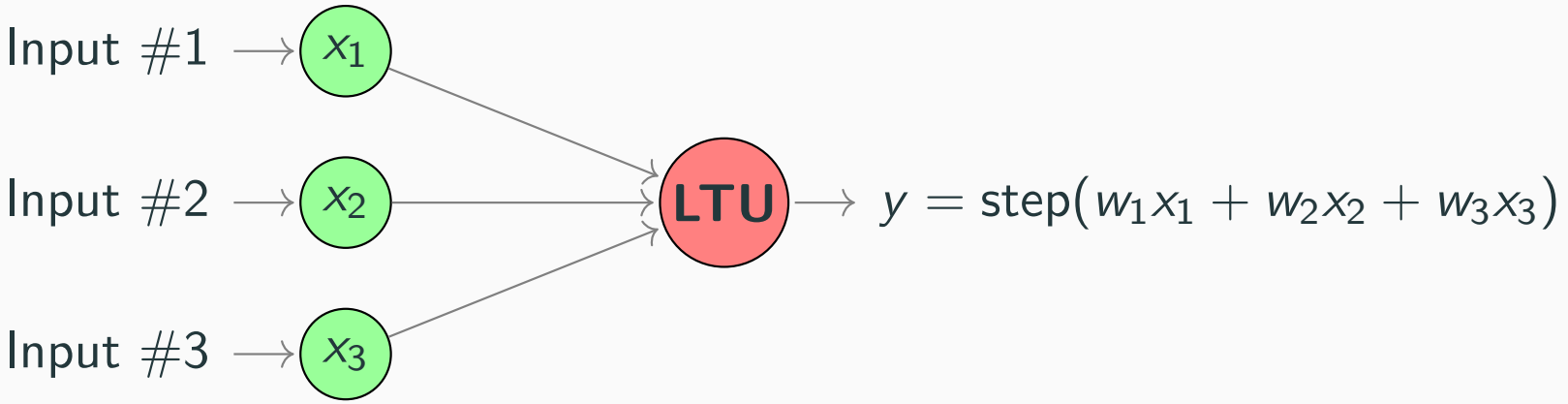
$$\textbf{LTU} : y = \text{step}(w_1 x_1 + w_2 x_2 + \ldots + w_p x_p)\,,$$

where $\text{step}(x)$ is the Heaviside step function.

$$\text{step}(x) = \begin{cases} 0 & \text{if } x < 0\,, \\ 1 & \text{if } x \geq 0\,. \end{cases}$$

Perceptron

Perceptron

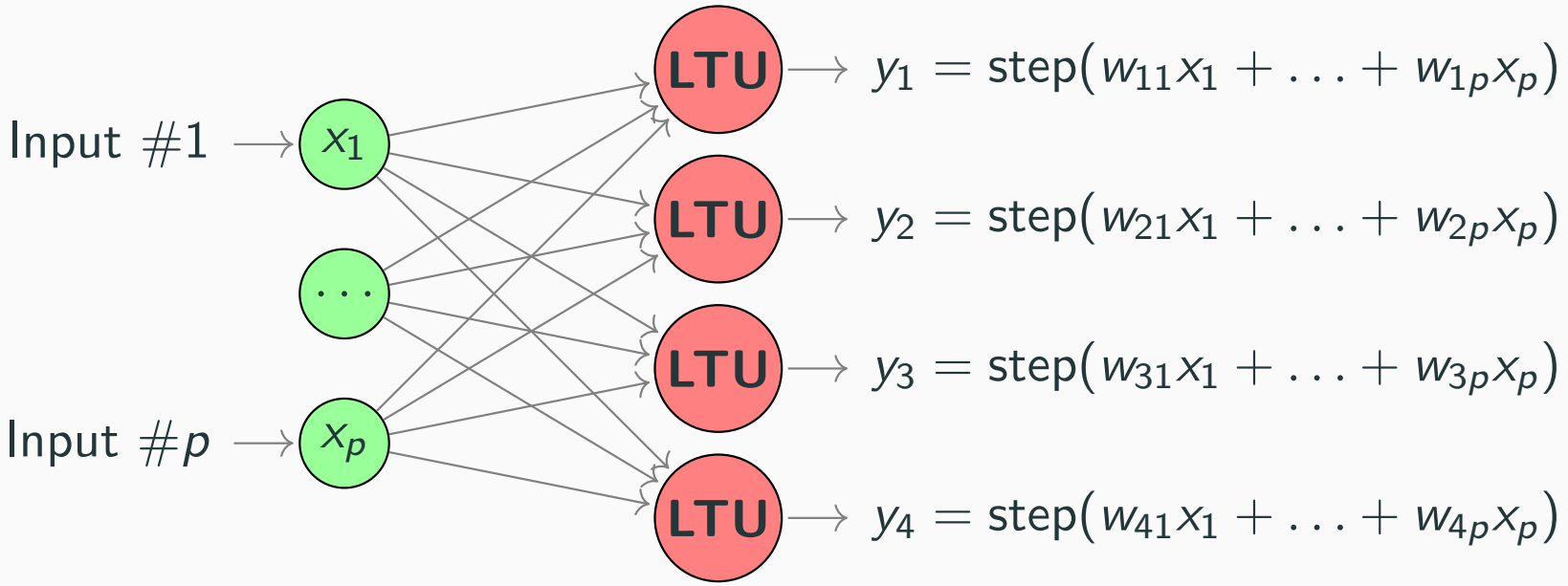


A single LTU gives a binary classifier using the sum of linear functions. Notice that functionally this give the same hypothesis class as (non-affine) linear regression

$$y = x_1\beta_1 + \ldots + x_p\beta_p$$

Perceptron

Multilabel Perceptron

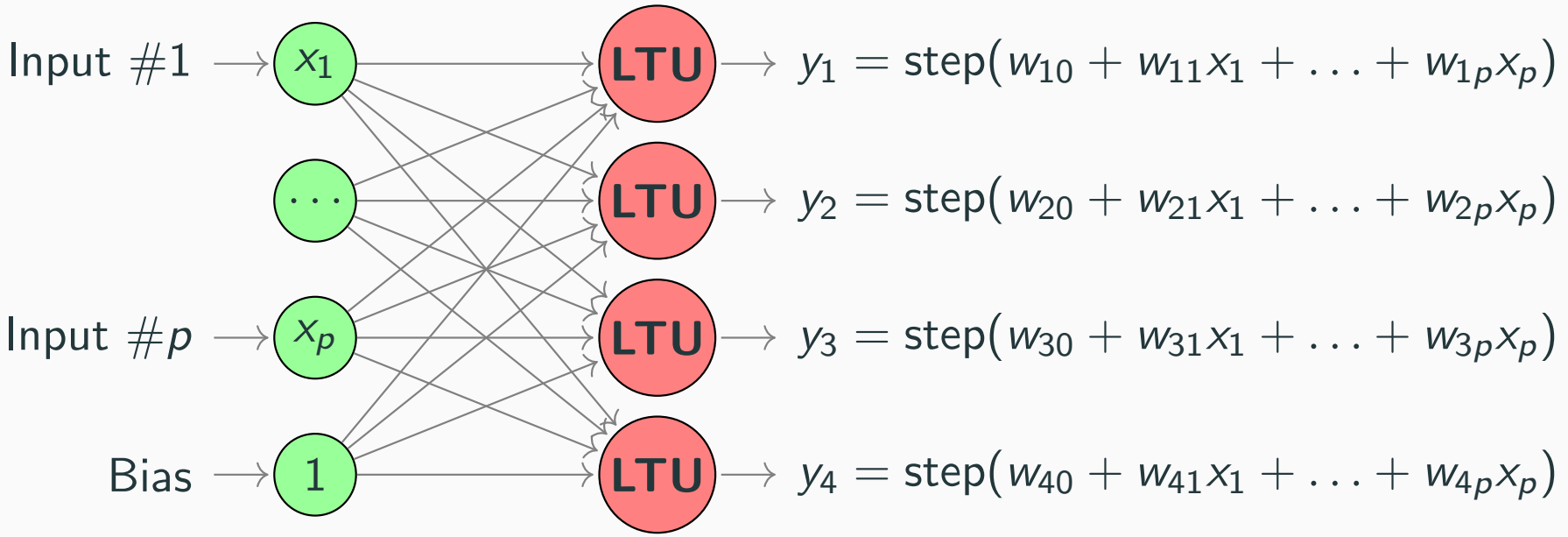


A single LTU gives a binary classifier using the sum of linear functions. For multilable classification we add a LTU nodes for each label.

Notice that we are not post-composing with argmax so it possible that multiple (or none) of the outputs could be 1.

Perceptron

Multilabel Perceptron



Finally, we can add a constant bias neuron to produce the affine terms w_{i0}.

Training the Perceptron

The perceptron training algorithm proposed by Rosenblatt followed Hebb's rule of organization, that is that when one neuron triggers another the connect between then is strengthened.

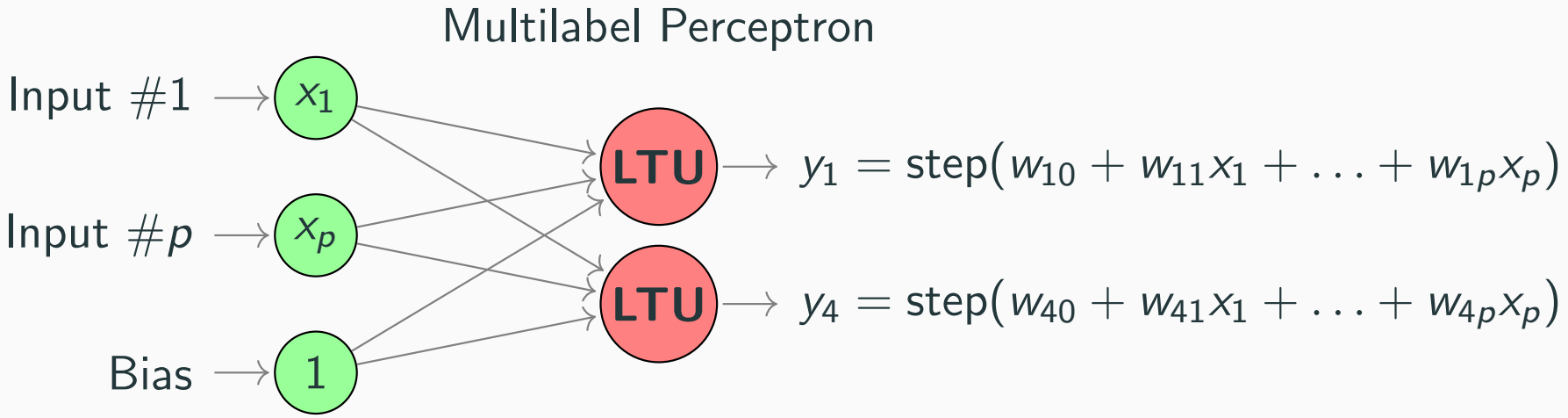
Rosenblatt also wanted to include feedback into the training, only strengthening connections the lead to correct predictions.

This leads to the learning rule

$$w_{ij}^{(n+1)} = w_{ij} + \eta(y_j - \hat{y}_j)x_i \,,$$

with learning rate η, a single training instance (x, y) and the predicted label $\hat{y}_i$.

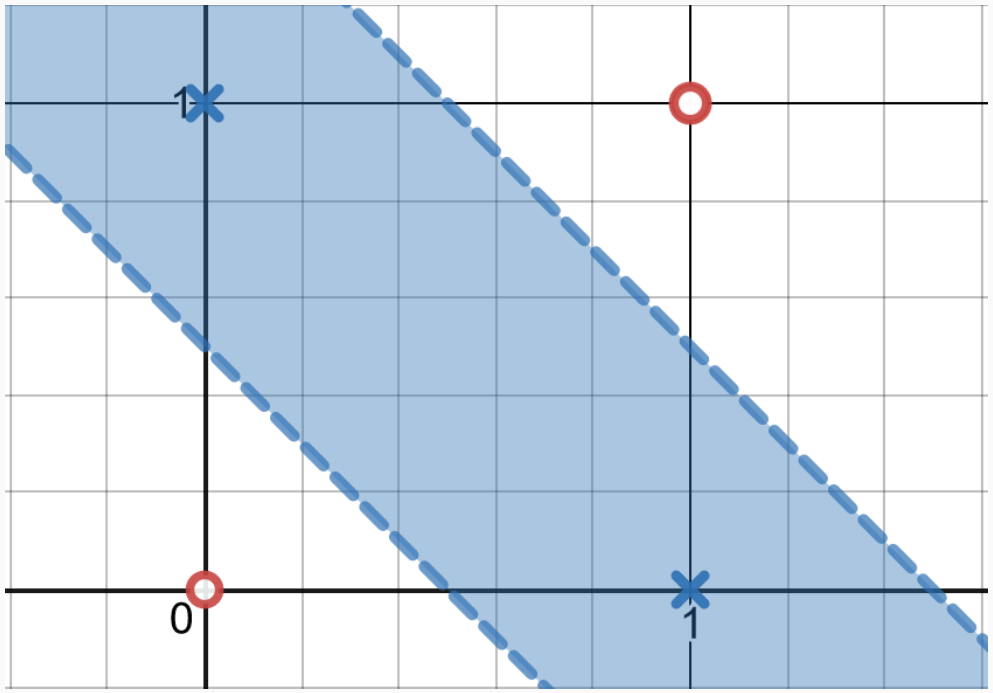
Perceptron



Learning Rule: $w_{ij}^{(n+1)} = w_{ij} + \eta(y_j - \hat{y}_j)x_i\,.$

Pass a single training instance x through the network:
If the prediction on label i is correct $\hat{y}_i = y_i$ and we do not update the weights w_{ij}.
If the prediction on label i is incorrect, $\hat{y}_i \neq y_i$. So if the true labeling is $y_i = 1$, add ηx_i to all weights w_{ij}.If $y_i = 0$, subtract ηx_i from all weights w_{ij}.
This is exactly the rule we wrote as a linear program.

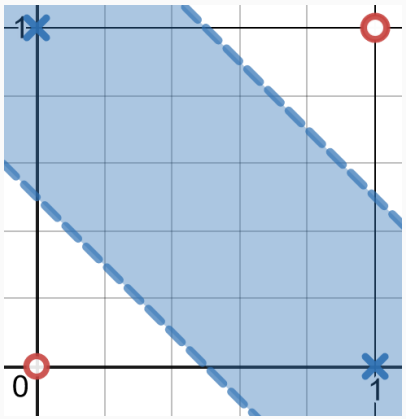
Perceptron



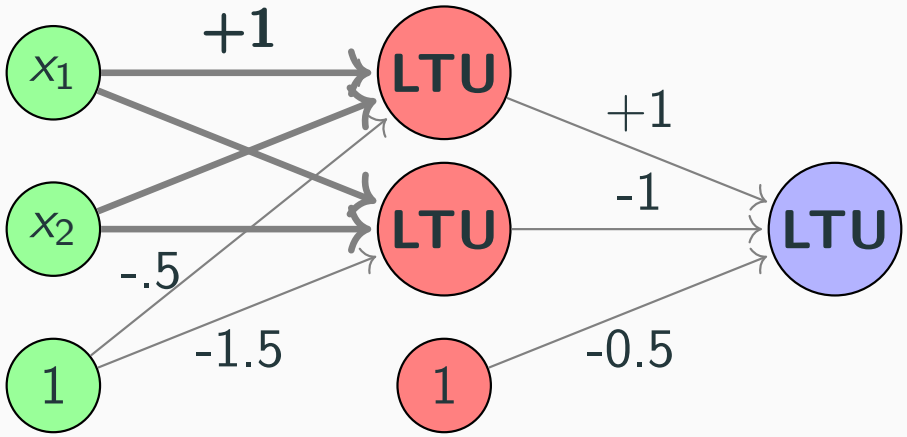
The perceptron itself is one of the first examples of SGD, but it has some serious problems. In particular, it cannot solve some trivial problems like the xor labeling above.

Although now unsurprising (no linear classifier can solve xor) the exceptions for the perceptron were high and when this problem was uncovered in 1969 it lead most researchers to abandon neural networks in favor of functional and logical methods.

Perceptron



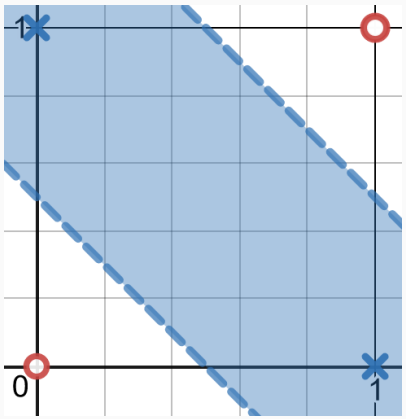
Multilayer Perceptron



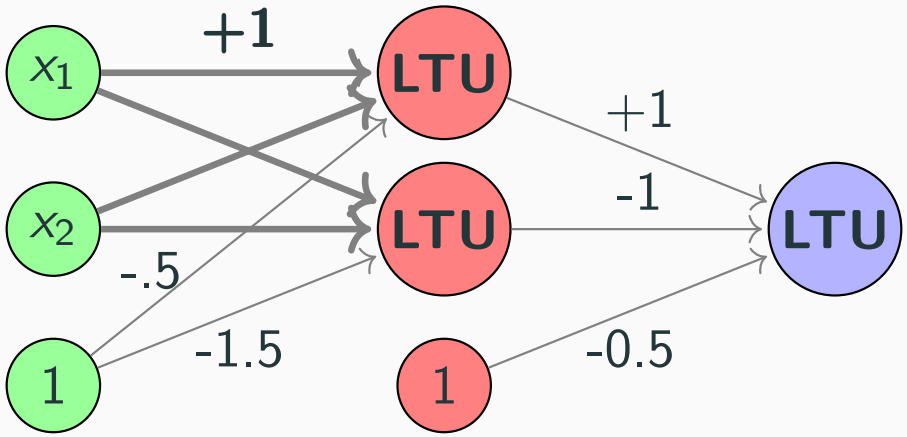
However it was premature to abandon the perceptron: it turns out that if you stack perceptrons you can fit the xor distribution. Consider sending the point $(0, 0)$ through:

$$(0,0) \rightarrow \text{step}(x_1 + x_2 - 0.5, x_1 + x_2 - 1.5) = (0,0)$$
$$\rightarrow \text{step}[(0) - (0) + (-0.5)] = 0$$

Perceptron



Multilayer Perceptron

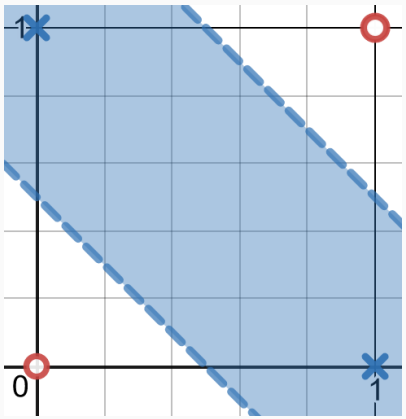


Sending (1,1) through, we get

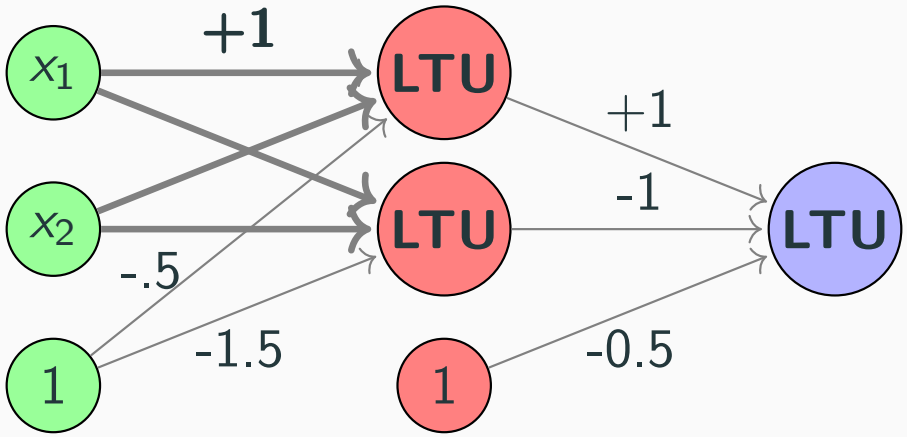
$$(1,1) \rightarrow \text{step}(1+1-0.5, 1+1-1.5) = (1,1)$$
$$\rightarrow \text{step}[(1)-(1)+(-0.5)] = 0$$

(**Exercise:**) Verify that (0,1) and (1,0) are labeled 1.

Perceptron



Multilayer Perceptron



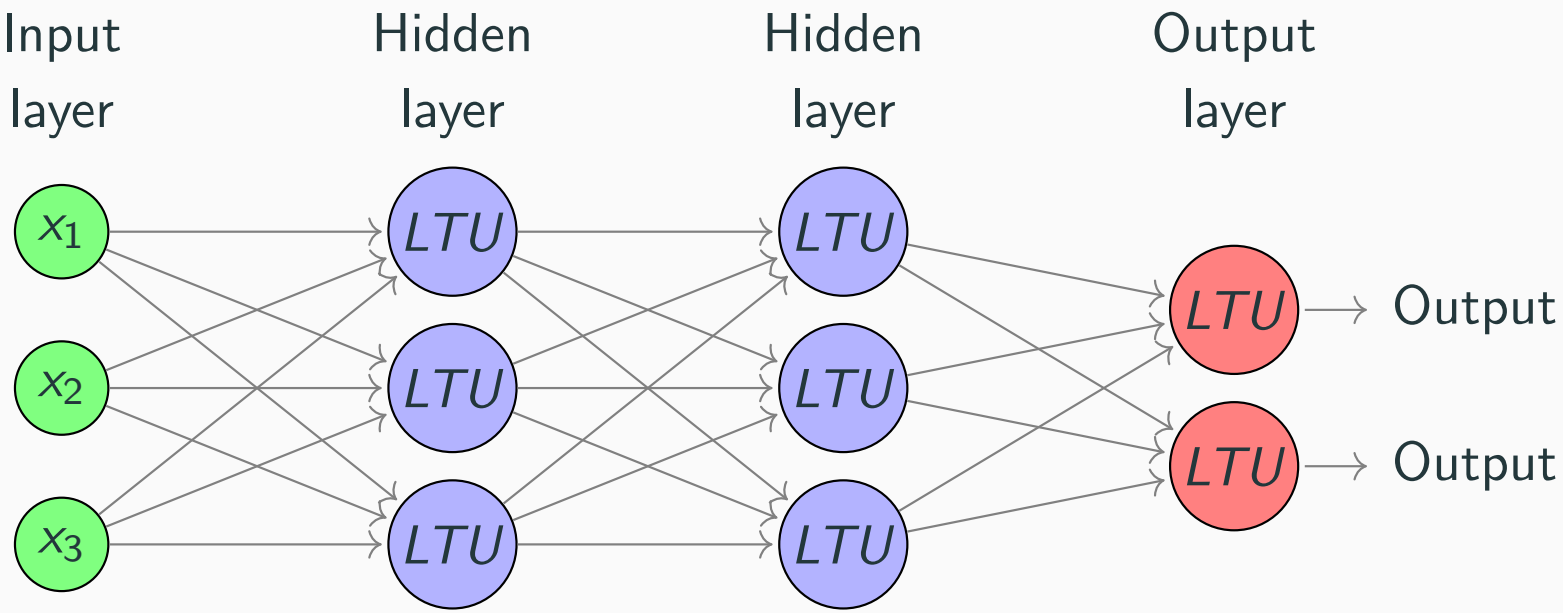
With a little bit of work, one can show that this multilayer perceptron actually is the fitting on the left:

$$\hat{h}(x_1, x_2) = \begin{cases} 1 & 0.5 - x_1 \leq x_2 \leq 1.5 - x_1 \\ 0 & \text{otherwise} \end{cases}$$

Questions: What is the hypothesis class of all two layer two variable perceptrons? How would we construct a perceptron with three linear conditions?

Gradient Decent and Back Propagation

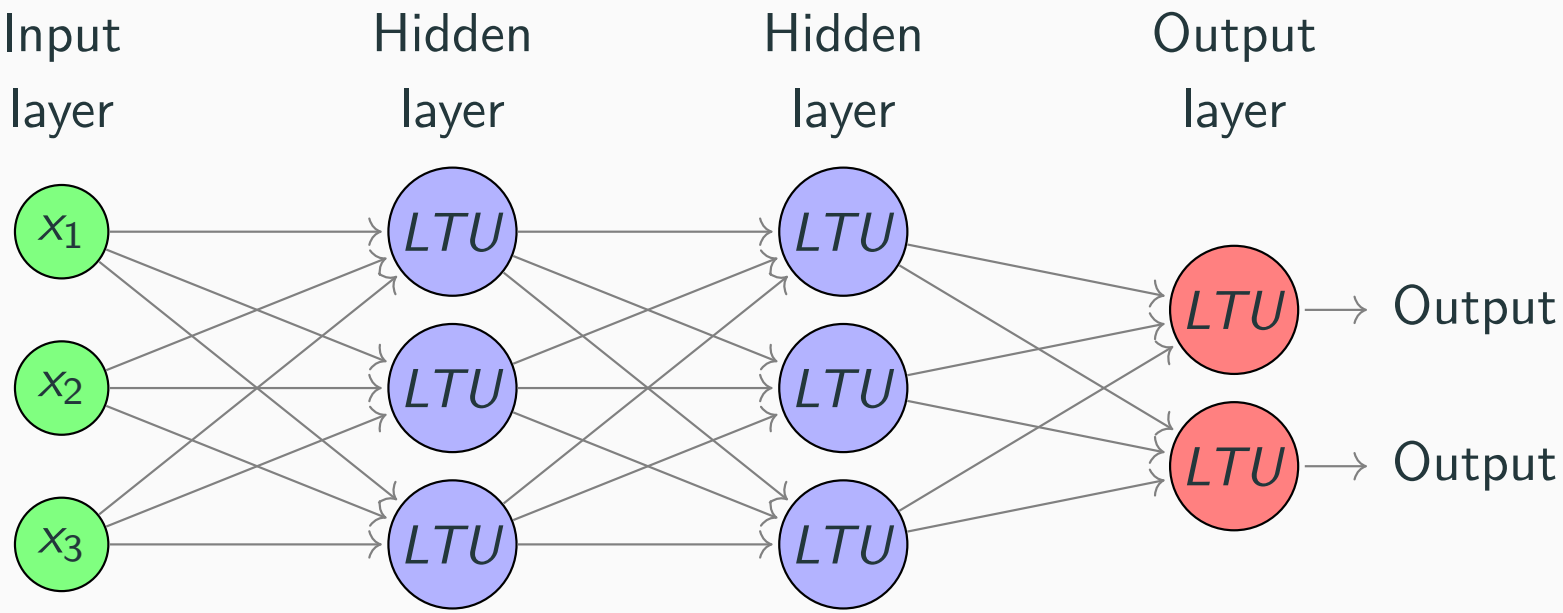
Training the Multilayer Perceptron



A multilayer perceptron (MLP) is composed of one input layer, multiple hidden layers and an output layer. If the network has more than one hidden layer it is called a **deep neural network**.

Learning how to effectively train MLP took another 20 years. In 1986, D. E. Rumelhart introduced the idea of **back propagation** to train MLPs.

Training the Multilayer Perceptron

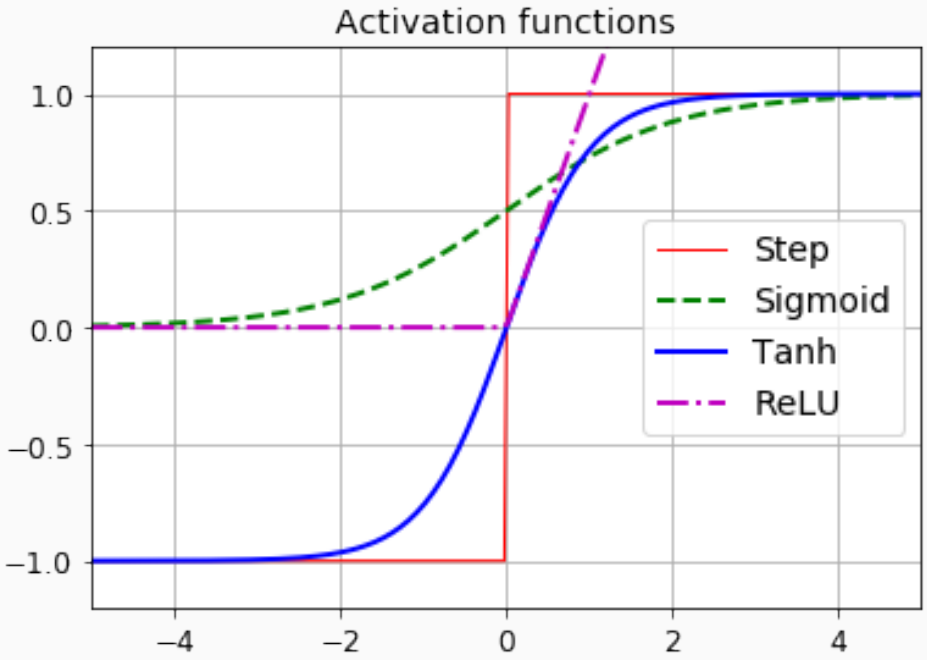


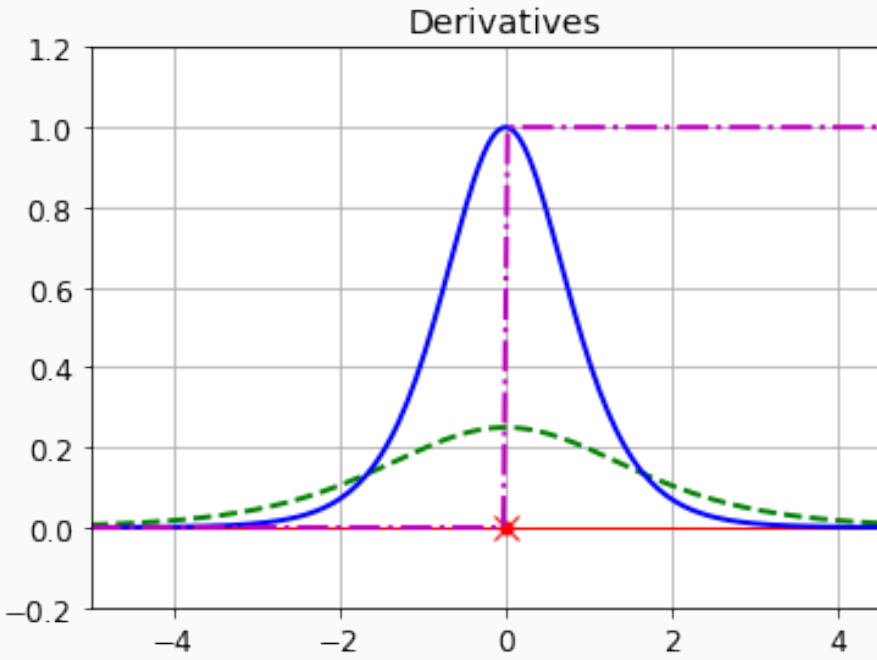
In order to compute the gradient, an important modification was made to the LSU: the step function was replaced by the **logistic** (aka **sigmoid** function

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

The problem here wasn't the discontinuity, but the fact that the step function has 0 gradient everywhere, so there is no indication of which direction to move.

Implementing Gradient Decent





Other popular alternatives to the step function are

$$\tanh(x) = 2\sigma(2x) - 1$$

and

$$\text{ReLU}(x) = \max\{0, x\}$$

Implementing Gradient Decent

There are several popular ways of implementing gradient decent. For a loss function like MSE, gradient decent updates the parameters at each step by

$$\beta^{(n+1)} = \beta^{(n)} - \eta \nabla_\beta \text{MSE}(\beta^{(n)}).$$

If there is a closed form formula for MSE, the update step can be computed explicitly. This is known as **manual differentiation** or **pen and paper differentiation**.

If the nodes are fairly out of the box, we can try to use automatic **symbolic differentiation** to pass through the graph and create a gradient. There are some decent solvers out there and this is an active area of development, but unfortunately it often yields huge redundant computations that allow errors to propagate. It is also must be implemented manually on arbitrary code.

Numerical Differential

Numerical differentiation is the simplest general solver. For a network $Y = h_\beta(X)$ with inputs X and outputs Y we implement partial differentiation by passing a data point (or points) through the network and seeing how a small change in β effects it

$$\frac{\partial h_\beta}{\partial \beta} \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_{\beta+\epsilon}(X_i) - h_\beta(X_i)}{\epsilon}, \qquad \epsilon \ll 1 .$$

For a large network, this takes many calls to the function while also being numerically suspect. It is however a good first pass test and it is easy to implement.

Autodiff

Forward mode autodiff is a mixture of numerical and symbolic differentiation. It relies on **dual numbers** (**infinitesimals** in the math context), that is we write

$$a + b\epsilon$$

where ϵ is "small," ie $\epsilon^2 = 0$. In memory, dual numbers are stored as a pair of floats (a, b).

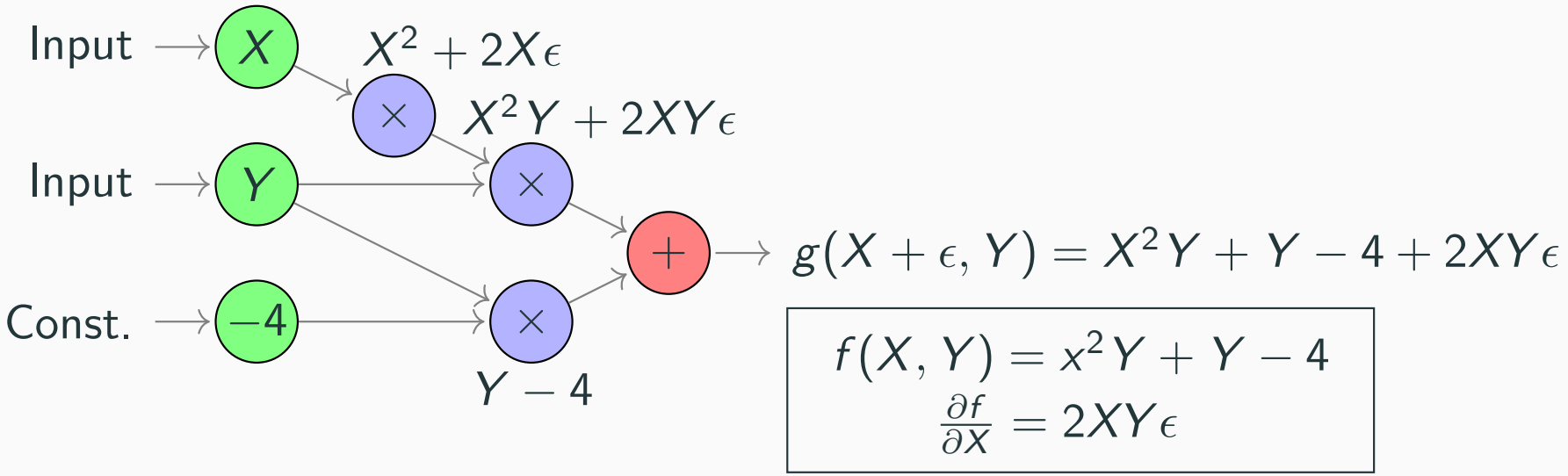
Dual numbers have slightly nonstandard multiplication and exponentiation that encode the rules of differentiation implicitly:

$$(a + b\epsilon)(c + d\epsilon) = ac + (bc + ad)\epsilon\,, \qquad e^{a+b\epsilon} = e^a(1 + b\epsilon)\,,$$

where the second equation is derive by expanding $e^{b\epsilon}$ and using $\epsilon^2 = 0$. In fact, if $f(x)$ is smooth than by Taylor theorem

$$f(a + b\epsilon) = f(a) + b\epsilon f'(a)\,.$$

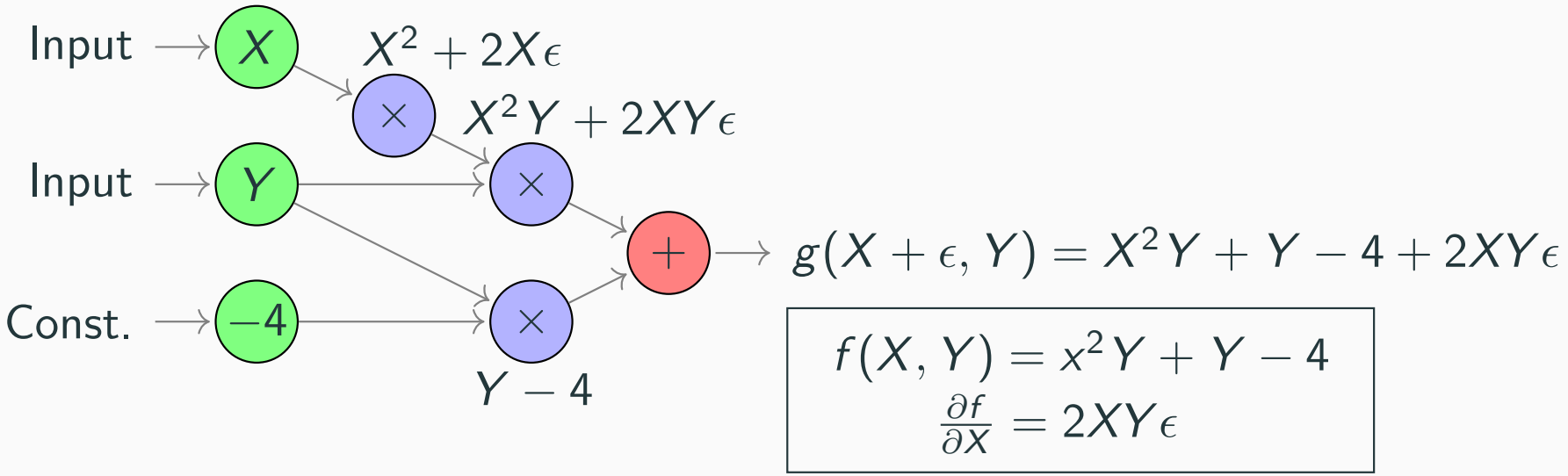
Forward Autodiff



Since $f(a + b\epsilon) = f(a) + b\epsilon f'(a)$ gives you both $f(a)$ and $f'(a)$ in one go, implementing forward autodiff comes down to creating a new graph that computes the result of passing the dual number through the graph.

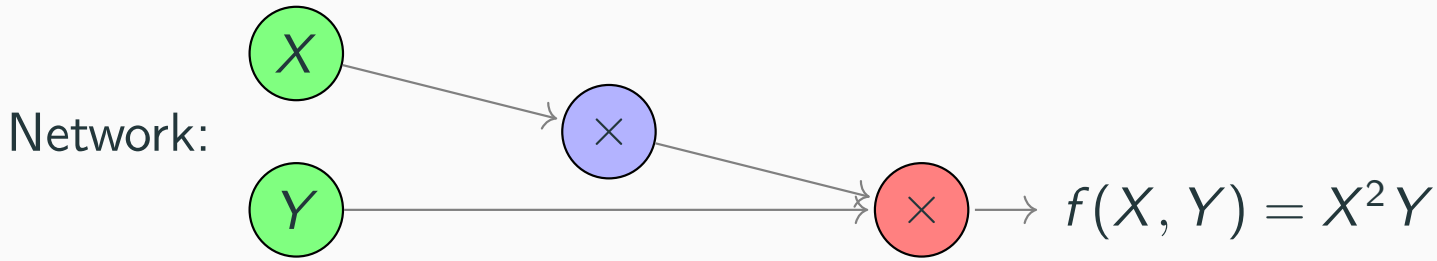
Of course, we intend to differentiate with respect to the weights, but that is equivalent to treating the weights as inputs.

Reverse Autodiff



Forward autodiff is exact (unlike numerical approximation) but it still involves passing through the graph once for each weight β. For a graph with many connection weights this cause take a long time. **Reverse autodiff** on the other hand passes through the graph exactly twice.

Reverse Autodiff

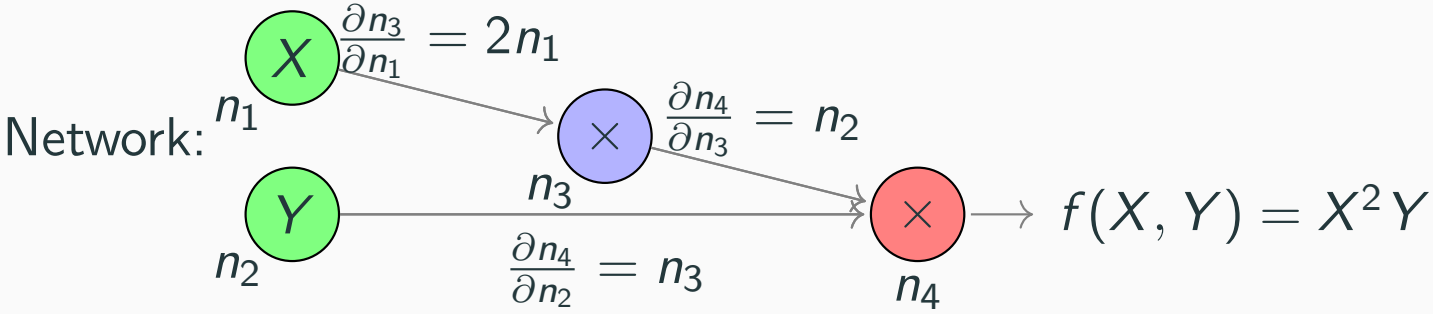


Reverse autodiff constructs a new graph containing the derivative of each input node with respect to each output node. It then uses the chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n}\frac{\partial n}{\partial x}$$

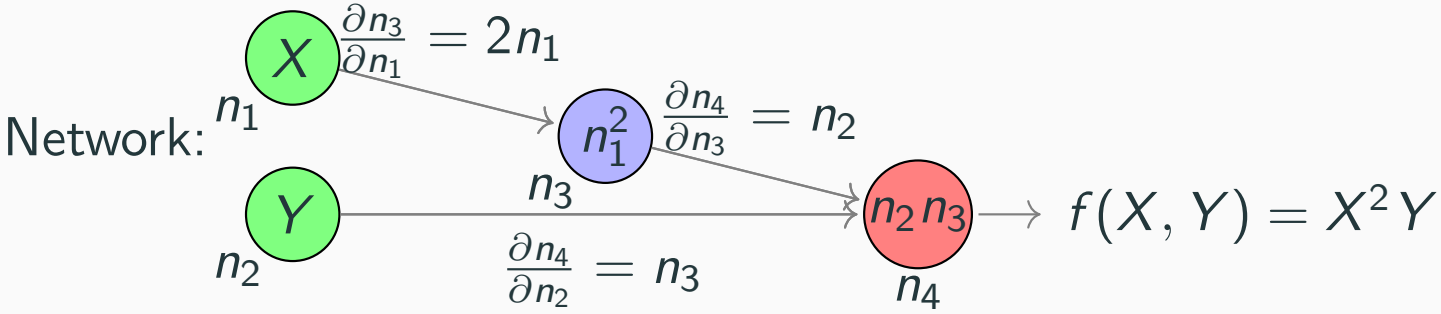
to compute the derivative by filling out a dual graph one step at a time.

Reverse Autodiff



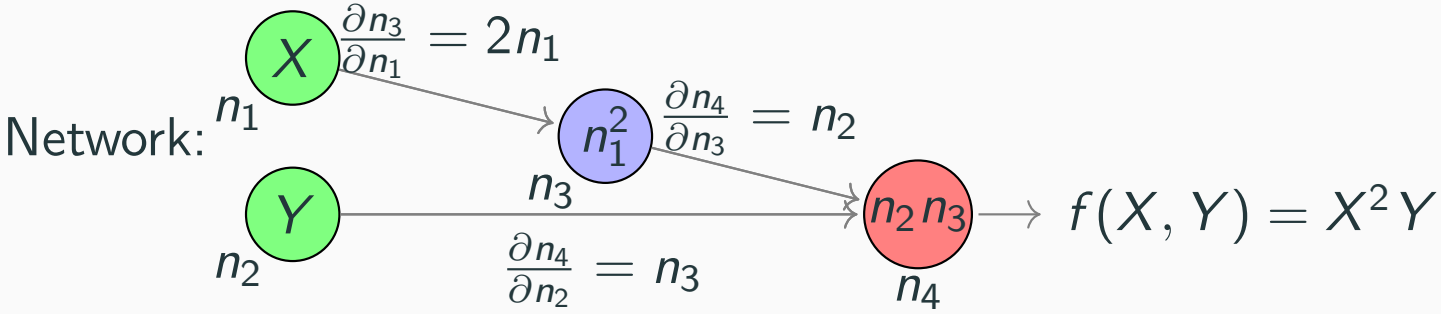
First, denote the value of each node by n_i.

Reverse Autodiff



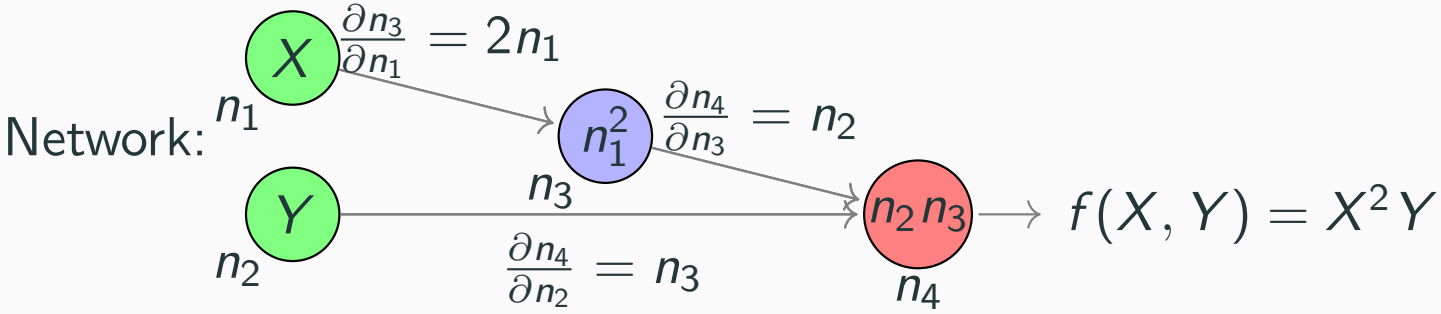
First, denote the value of each node by n_i. Second, propagating the node values through, compute the partial derivative at each connection. This can be done locally at each node upon constructing the graph.

Reverse Autodiff



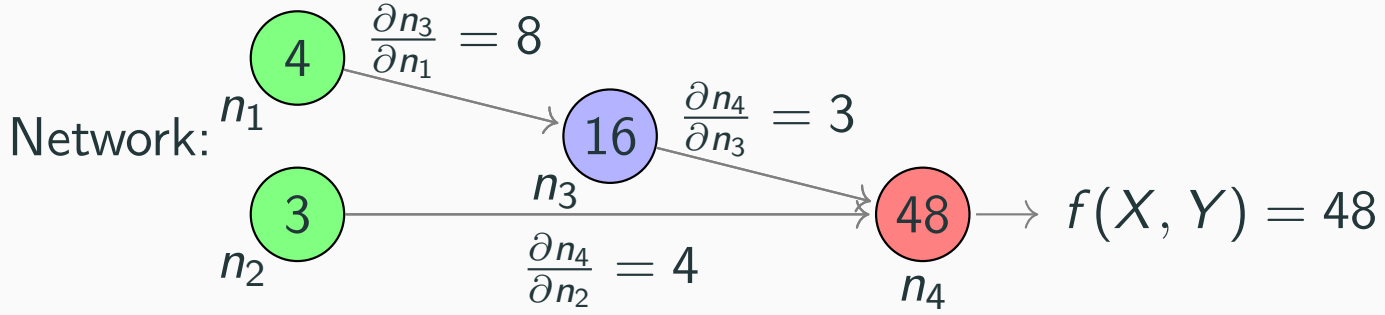
First, denote the value of each node by n_i. Second, propagating the node values through, compute the partial derivative at each connection. This can be done locally at each node upon constructing the graph. This constructs the gradient graph.

Reverse Autodiff



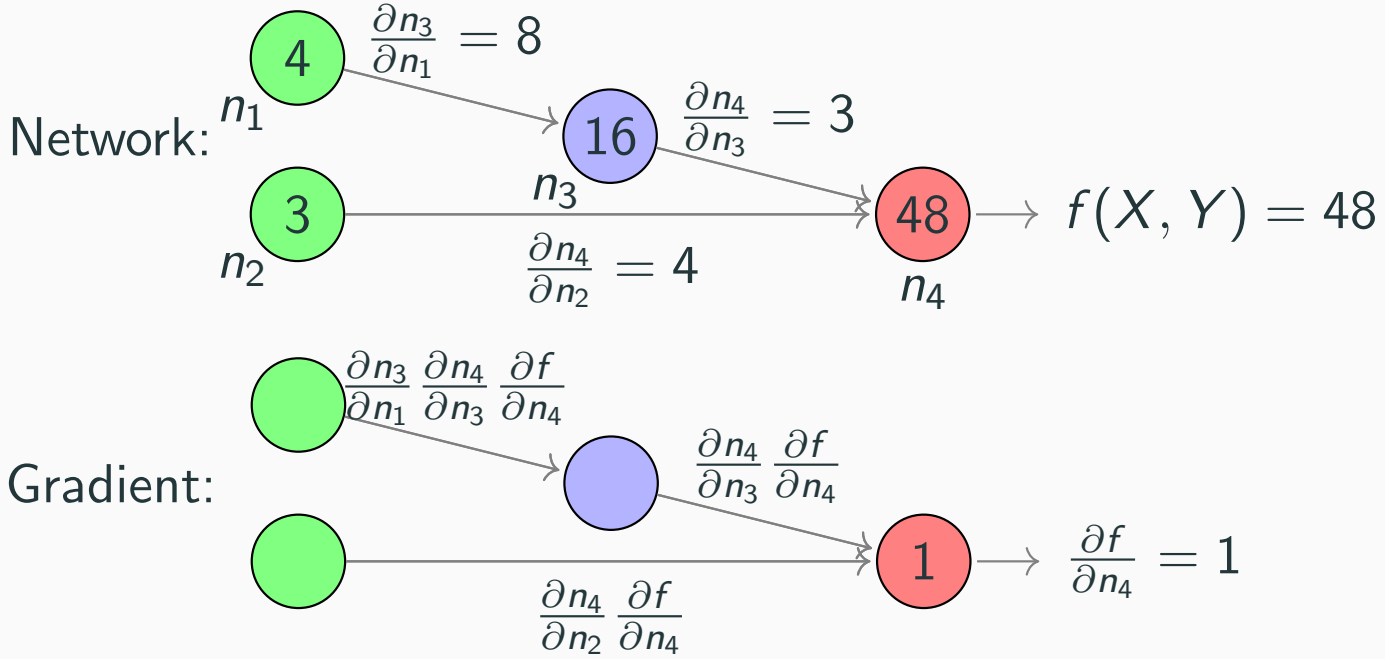
At runtime, for fixed (X, Y), say (4,3), we compute the gradient as follows. Pass (4,3) through the graph, filling all nodes.

Reverse Autodiff



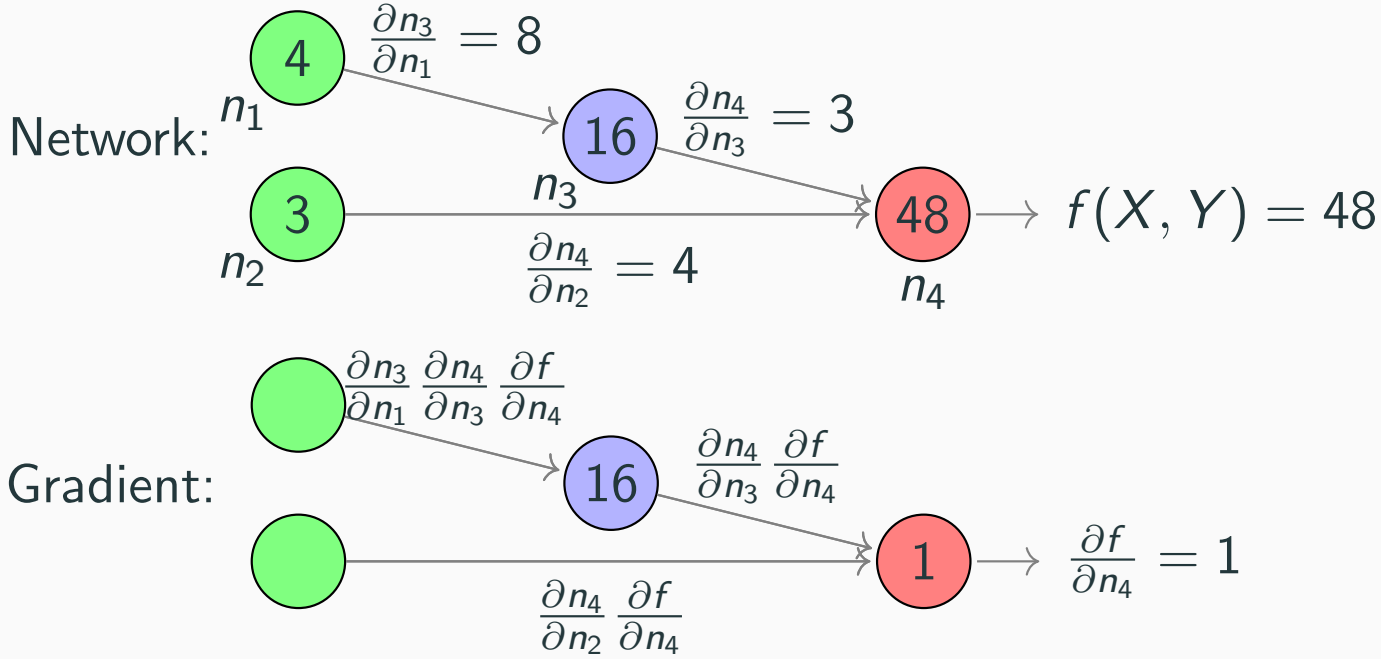
At runtime, for fixed (X, Y), say (4,3), we compute the gradient as follows. Pass (4,3) through the graph, filling all nodes.

Reverse Autodiff



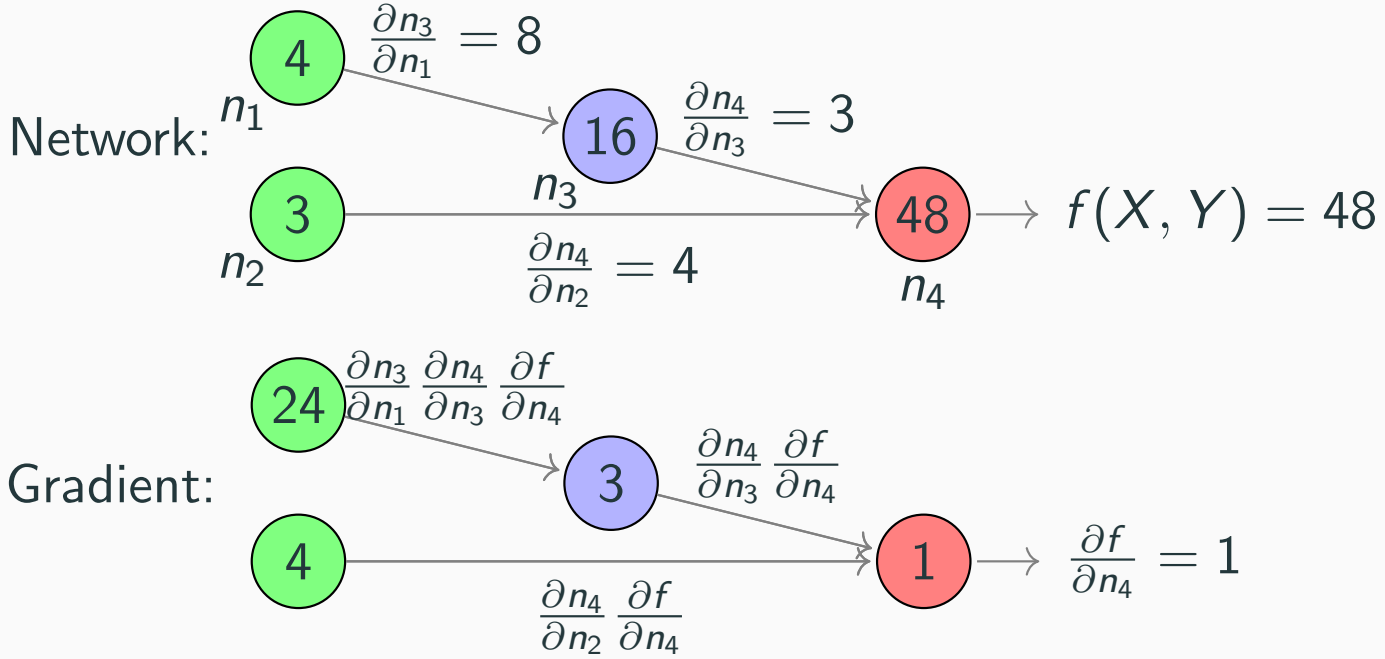
At runtime, for fixed (X, Y), say (4,3), we compute the gradient as follows.
Pass (4,3) through the graph, filling all nodes.
Second, introduce a new graph and back fill it using the values computed in the forward graph. Starting with $\frac{\partial f}{\partial n_4} = 1$, multiply backwards until the graph is filled.

Reverse Autodiff



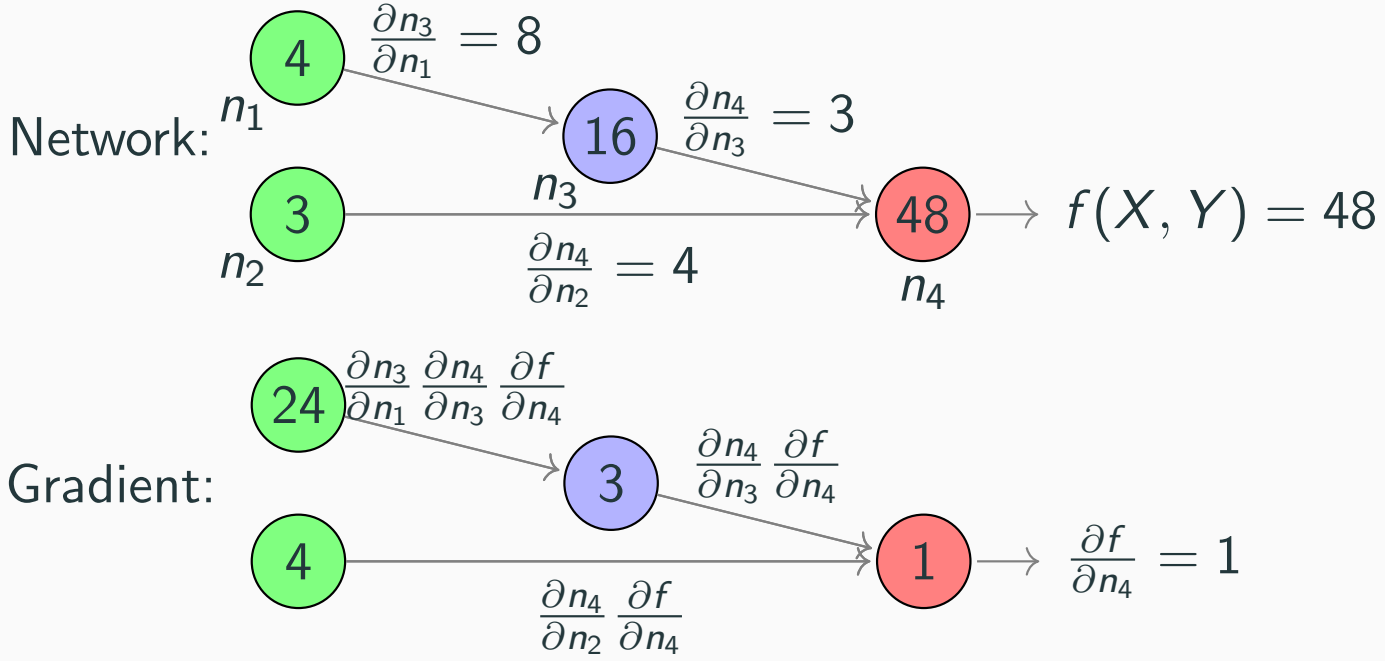
At runtime, for fixed (X, Y), say (4,3), we compute the gradient as follows.
Pass (4,3) through the graph, filling all nodes.
Second, introduce a new graph and back fill it using the values computed in the forward graph. Starting with $\frac{\partial f}{\partial n_4} = 1$, multiply backwards until the graph is filled.

Reverse Autodiff



At runtime, for fixed (X, Y), say (4,3), we compute the gradient as follows.
Pass (4,3) through the graph, filling all nodes.
Second, introduce a new graph and back fill it using the values computed in the forward graph. Starting with $\frac{\partial f}{\partial n_4} = 1$, multiply backwards until the graph is filled.

Reverse Autodiff

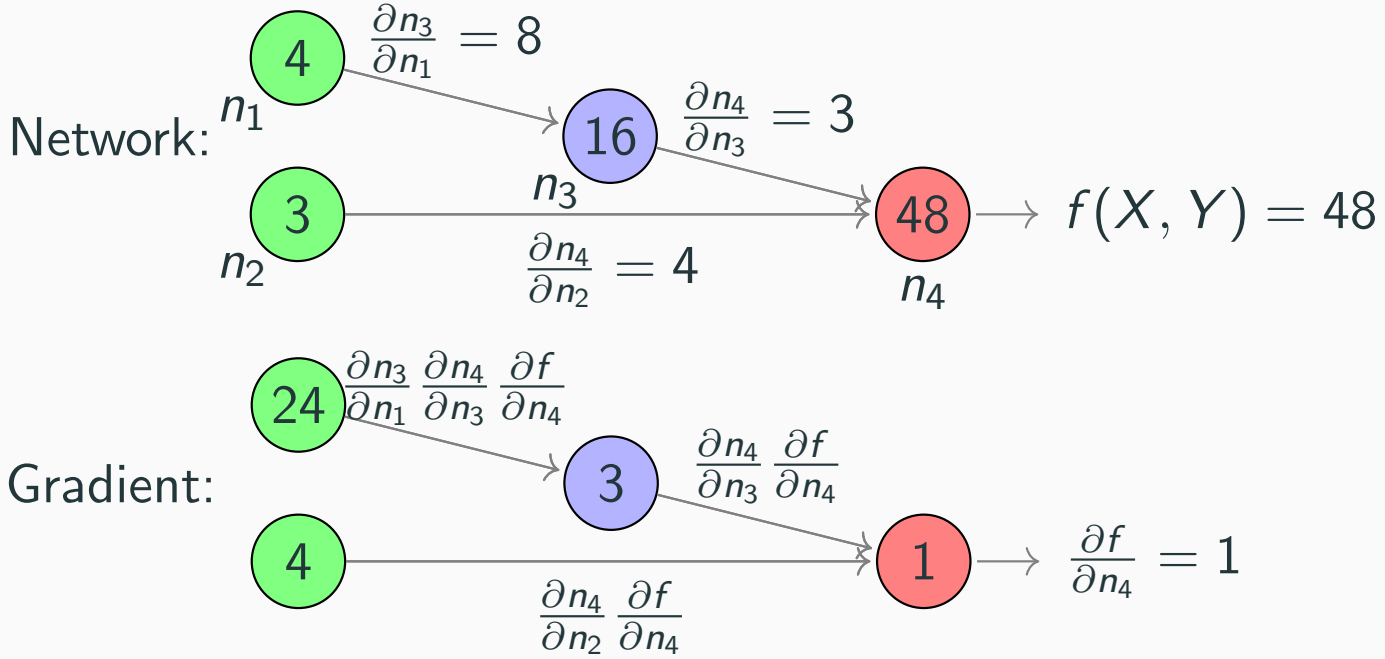


We are just computing the chain rule

$$\frac{\partial f}{\partial X}(4,3) = \frac{\partial n_1}{\partial x}\frac{\partial n_3}{\partial n_1}\frac{\partial n_4}{\partial n_3}\frac{\partial f}{\partial n_4}(4,3) = 24$$

By precomputing the derivatives at each node, reverse autodiff allows us to very quickly compute the whole gradient in two passes though the graph.

Reverse Autodiff



There is of course one caveat: If we have many output labels f_i or a node that is not differentiable we may not be able to use reverse autodiff.

The first problem is simply an limitation of the algorithm, but the second we can often smooth away by replacing a node with a smooth function, or by using a subgradient (?).

Training Neural Networks

Table 9-2. Main solutions to compute gradients automatically

Technique	Nb of graph traversals to compute all gradients	Accuracy	Supports arbitrary code	Comment
Numerical differentiation	$n_{\text{inputs}} + 1$	Low	Yes	Trivial to implement
Symbolic differentiation	N/A	High	No	Builds a very different graph
Forward-mode autodiff	n_{inputs}	High	Yes	Uses *dual numbers*
Reverse-mode autodiff	$n_{\text{outputs}} + 1$	High	Yes	Implemented by TensorFlow

The four nontrivial methods of computing gradients for gradient decent are summarized in this tabel taken from Chapter 9 or Geron. Tensorflow has automatically implemented reverse autodiff and as long as you use it's constructors it can compute gradients for you.

If you intend to implement a new kind of neuron you will have to code in compatibility with autodiff by hand, but after you do it once tensorflow will take it forward.

Reference

McCulloch and Pitts "A logical calculus of ideas immanent is nervous activity."
`http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf`

Neuroplacsicity Picture:
`https://www.cognifit.com/brain-plasticity-and-cognition`

This lecture is based on Chapters 9 - 10 of Geron "Hands on Machine Learning with Scikit-Learn and Tensorflow."