Neural Networks

A basic and intuitive introduction to “shallow” fully-connected feed-forward networks

Lion Krischer

Goal for Now + First Part of the Afternoon

- Fully understand and implement a simple feed-forward neural network from scratch.
- Little math (and probably slightly wrong notation).
- Go slow and step by step.

A bit tedious at times but not very hard.

Modern and deep networks are a bit more complicated but not fundamentally different.

Machine Learning

Input $\longrightarrow$ f(x) $\longrightarrow$ Output

Input ⟶ **f(x)** ⟶ **Output**

Physical Model

Source Parameters
Earth Model

F(Earth Model, Source Parameters)

Seismograms

Input $\longrightarrow$ f(x) $\longrightarrow$ Output

Anything | Machine Learning Model | Anything

In a way we replace our knowledge of the physics or the logic of a problem with an arbitrary “learned” function.

Input → **f(x)** → **Output**

Input	f(x)	Output
Anything	Machine Learning Model	Anything

Input → **f(x)** → **Output**

Anything | Machine Learning Model | Anything

Data Driven

Input → f(x) → Output

Anything

Supervised Machine Learning Model
Neural Network

Anything

Data Driven

When is this a good idea/useful?

- If it is very hard to make a physical or logical model.
    - Image and speech recognition, self-driving cars, language translation, …
    - Phase detection and recognition (see Thursday), signal classification (see Wednesday), …
- If the physical model is very expensive to compute but the result is somehow “simple”.
- And: One has a LOT of data.

Neural Networks

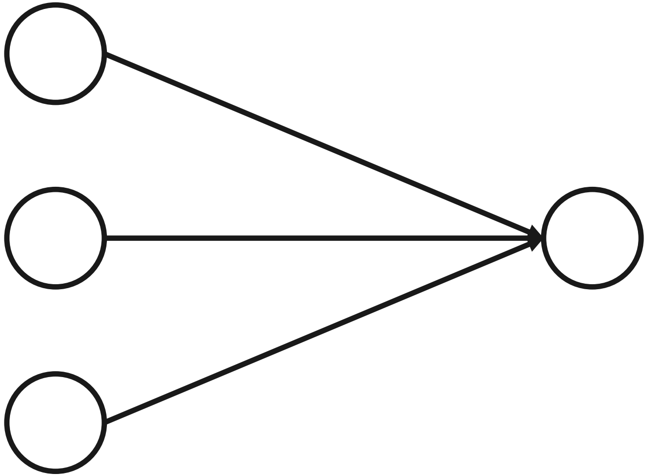
Universal Function Approximators

- **Universal approximation theorem**: Feed-forward networks of finite size with a single hidden layer and non-linear activation functions can approximate any continuous function (under some mathematical restrictions).
- Newer results reduce the necessary width of the networks under some conditions.
- Only a statement that one could always create a neural network to perform a certain action. Not if it is actually learnable.

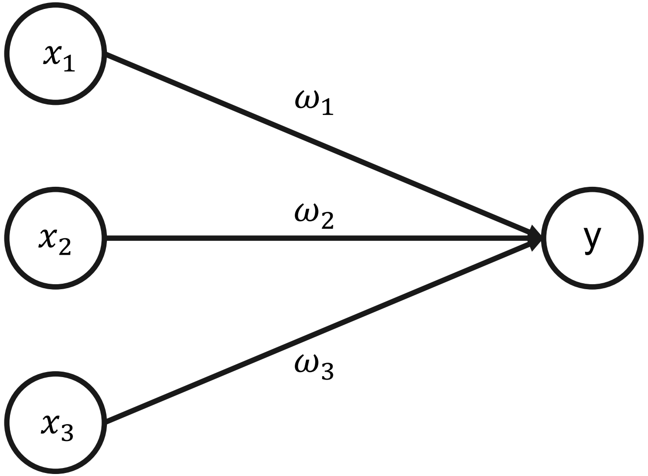
Thus the questions is not “Can a neural network do this?” but “Can I make it learn what I want it to do?”.

Neural Networks Step by Step

Weighted Sum

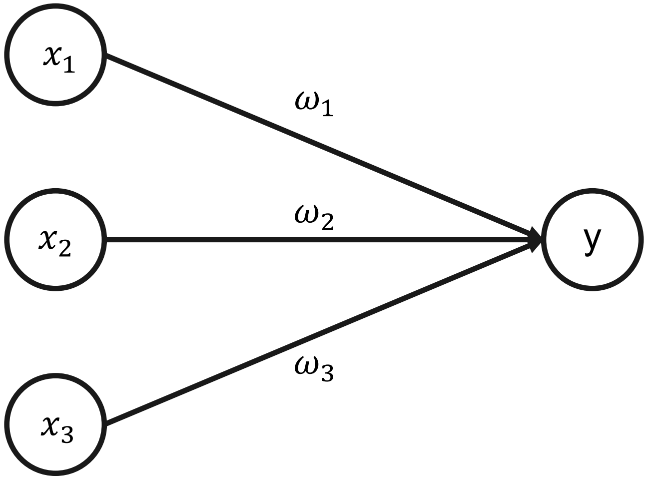


Weighted Sum



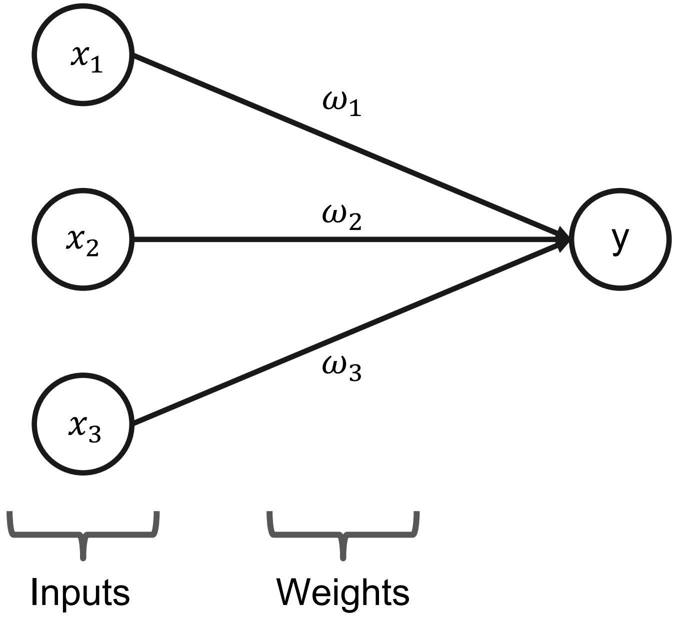
Weighted Sum

$$y = \omega_1 * x_1 + \omega_2 * x_2 + \omega_3 * x_3 = \sum_i \omega_i x_i$$



Weighted Sum

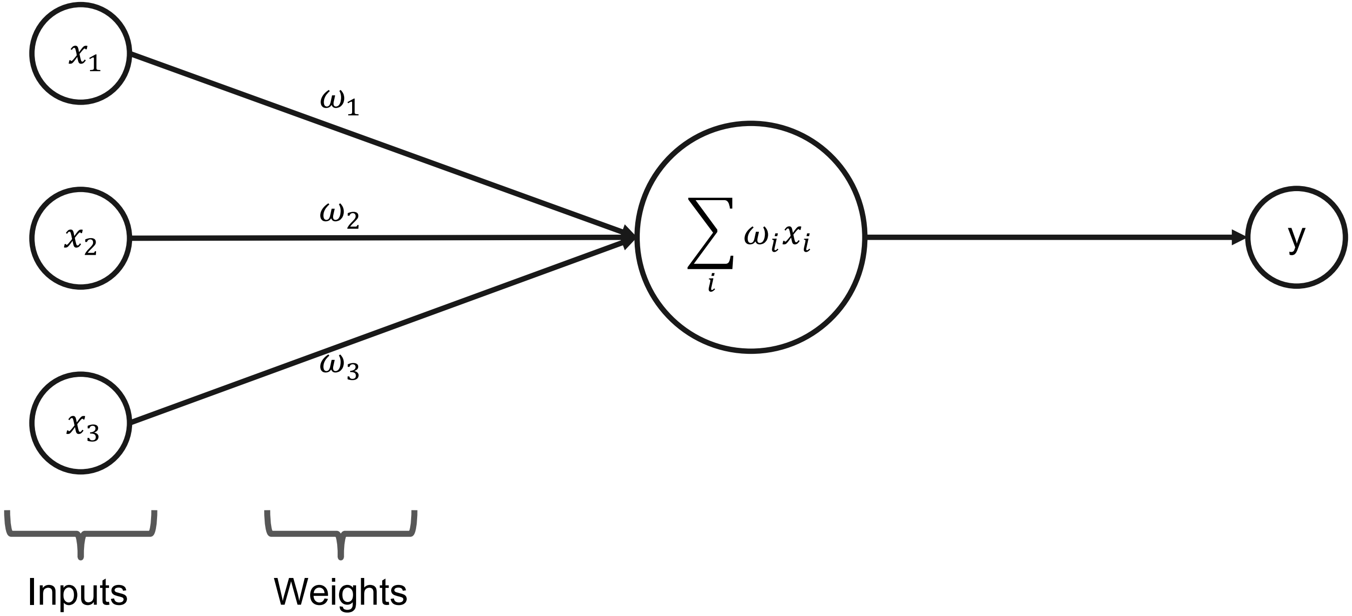
$$y = \omega_1 * x_1 + \omega_2 * x_2 + \omega_3 * x_3 = \sum_i \omega_i x_i$$



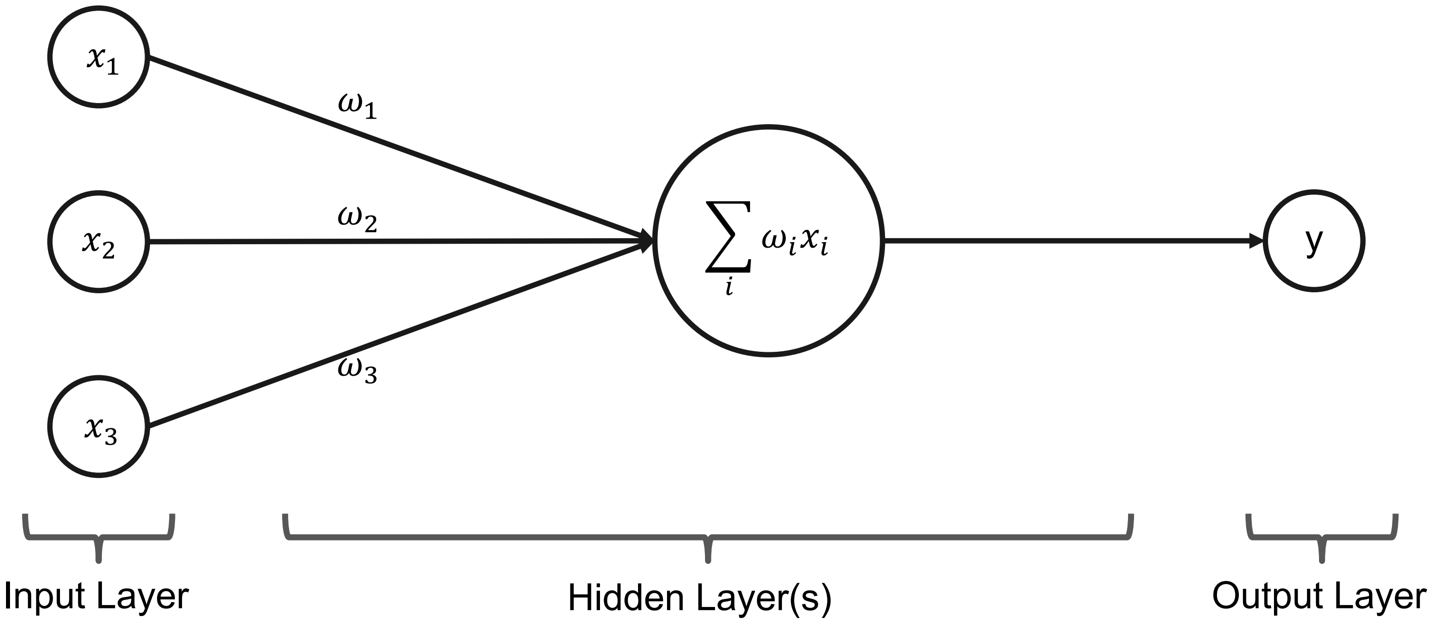
Weighted Sum

Not too useful yet – basically a dot product

Weighted Sum

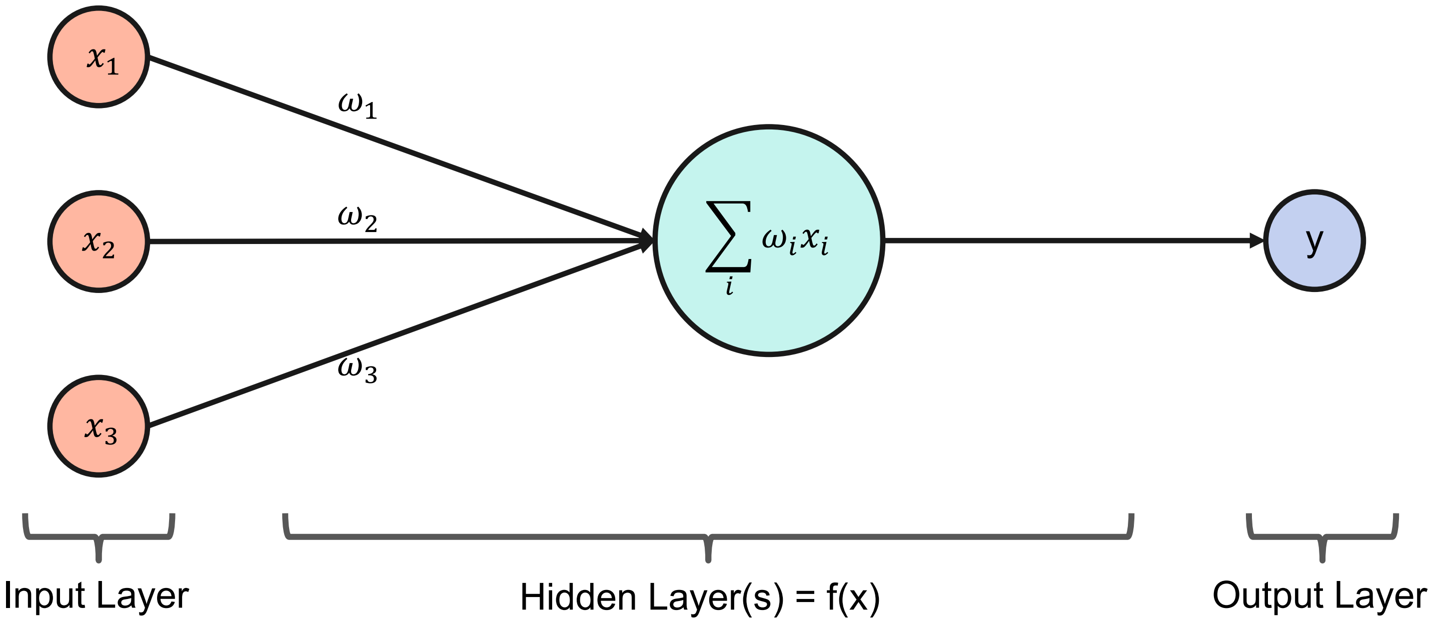


Weighted Sum



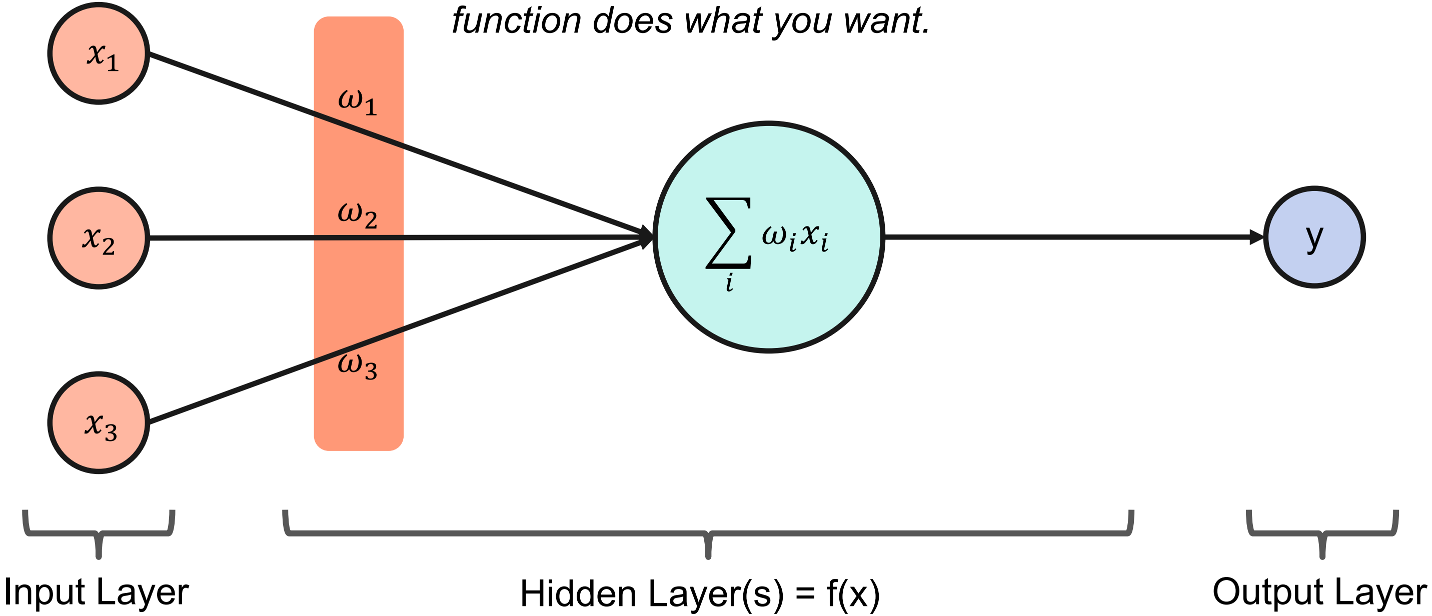
$$\text{Input} \longrightarrow f(x) \longrightarrow \text{Output}$$

Weighted Sum



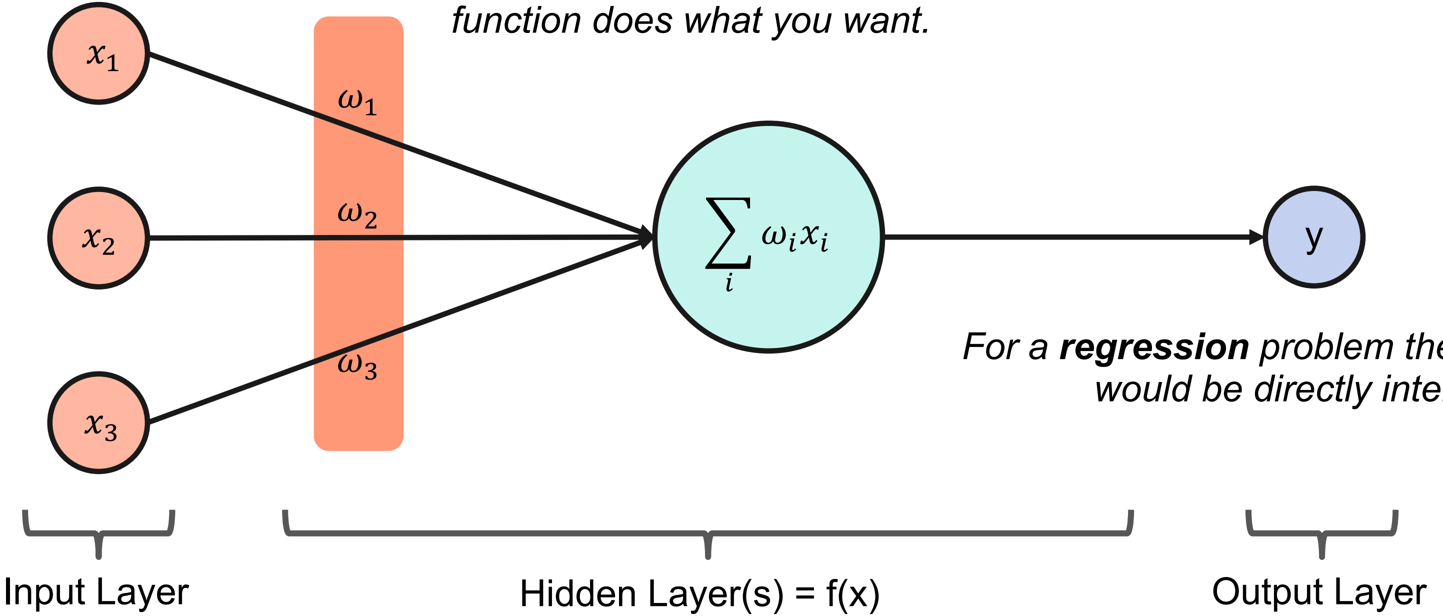
Weighted Sum

Change weights to change the behavior of the function. "Learning" would thus be to find better weights so the function does what you want.



Weighted Sum

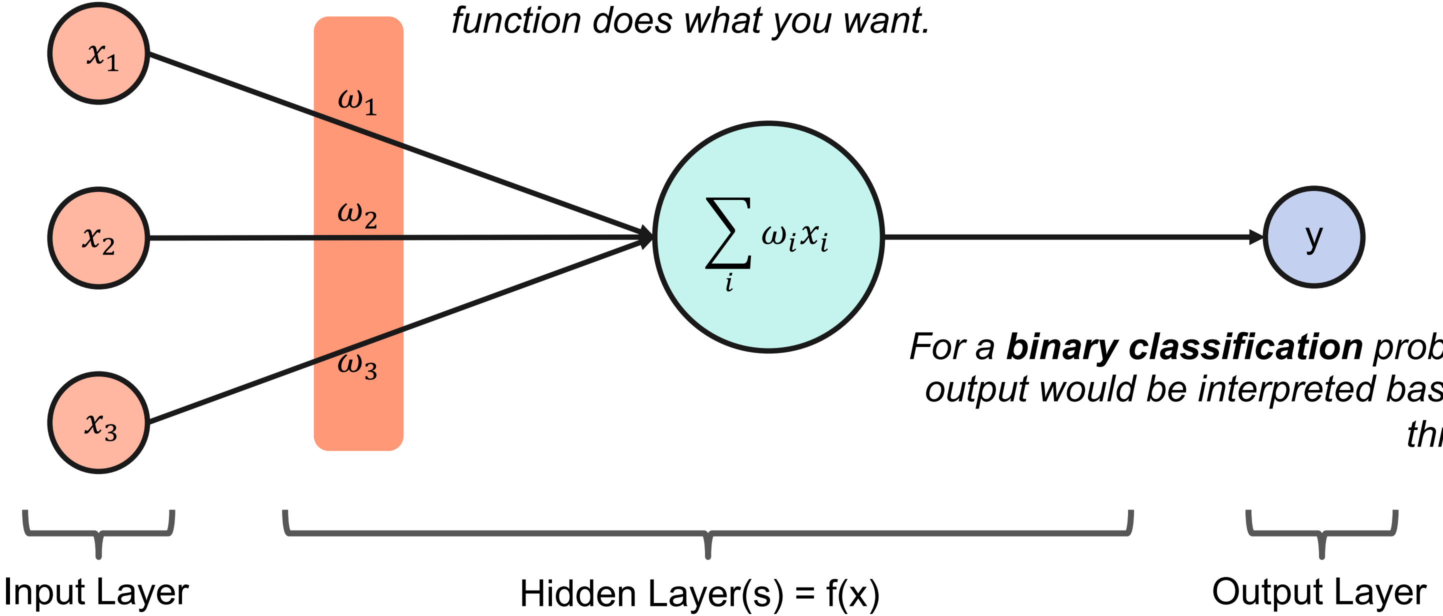
Change weights to change the behavior of the function. "Learning" would thus be to find better weights so the function does what you want.



*For a **regression** problem the output would be directly interpreted.*

Weighted Sum

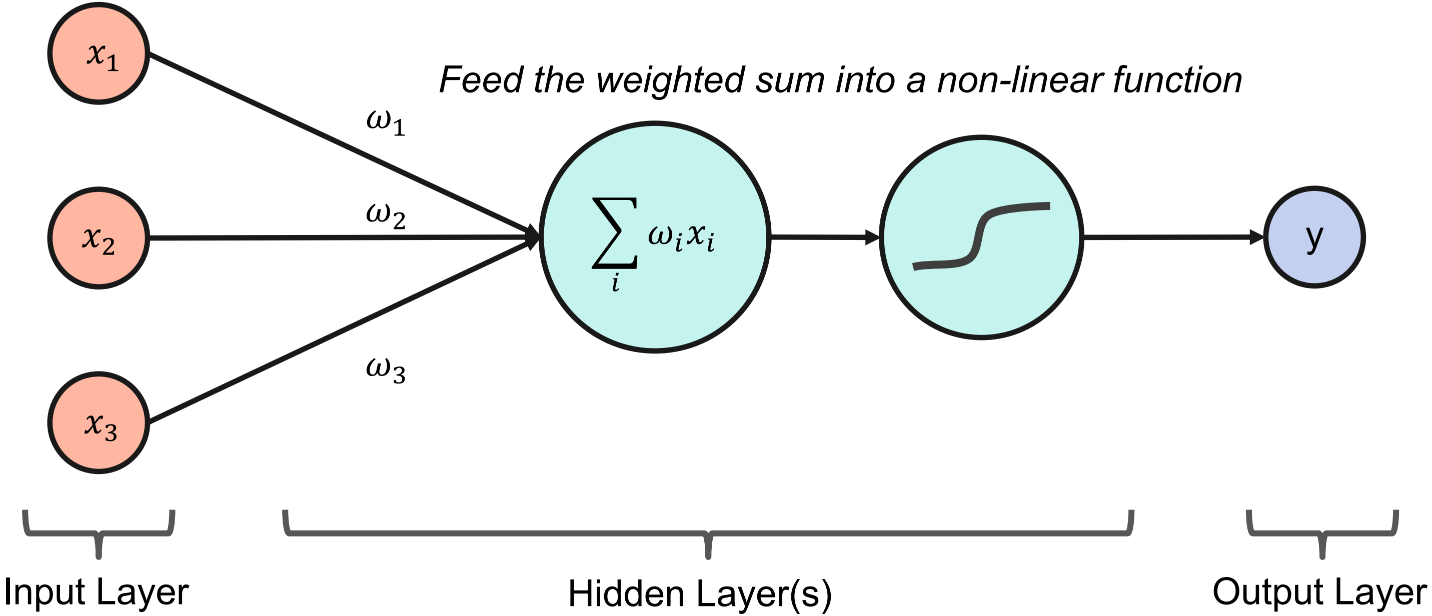
Change weights to change the behavior of the function. "Learning" would thus be to find better weights so the function does what you want.



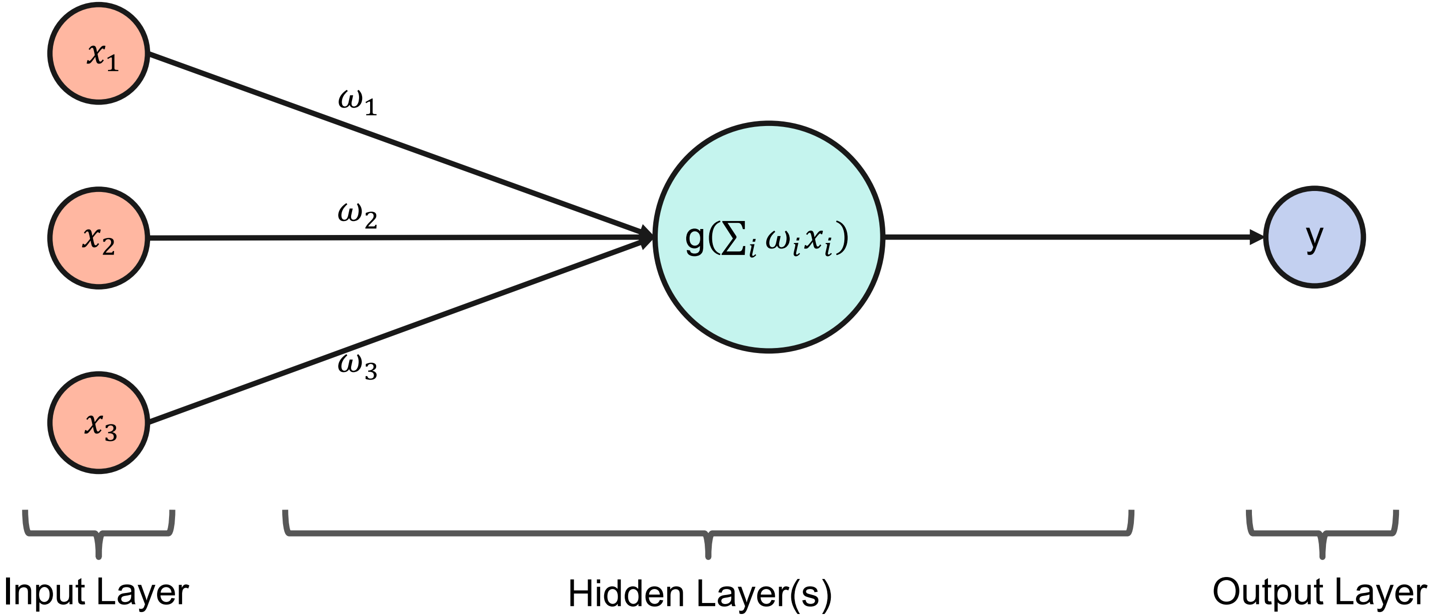
*For a **binary classification** problem the output would be interpreted based on a threshold.*

Computing something non-linear might be useful.

Activation Function



Activation Function



Why do we need an activation function?

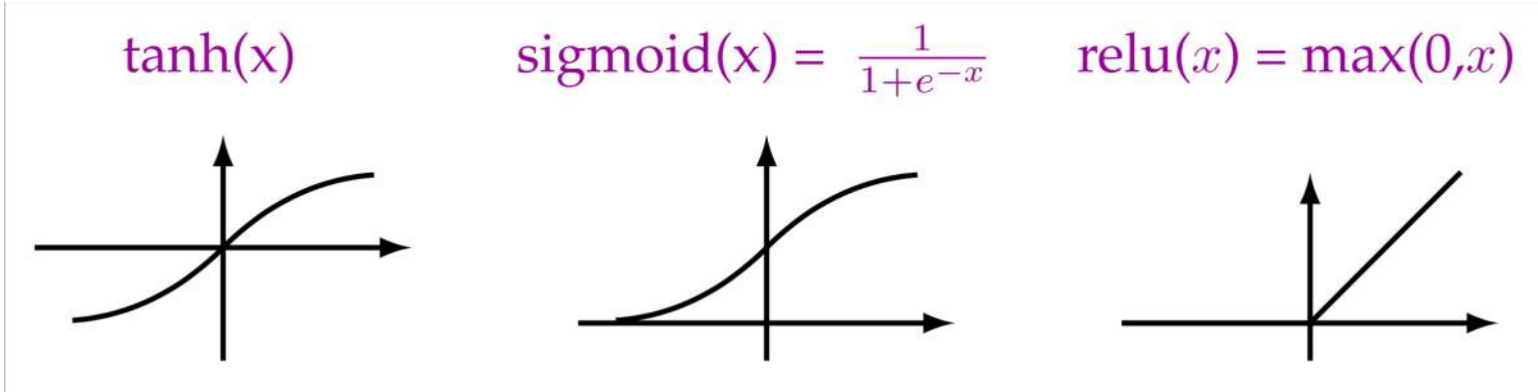
- Some non-linearity is needed to compute anything non-linear.

- More complex relationships between inputs and outputs cannot be represented by something purley linear.

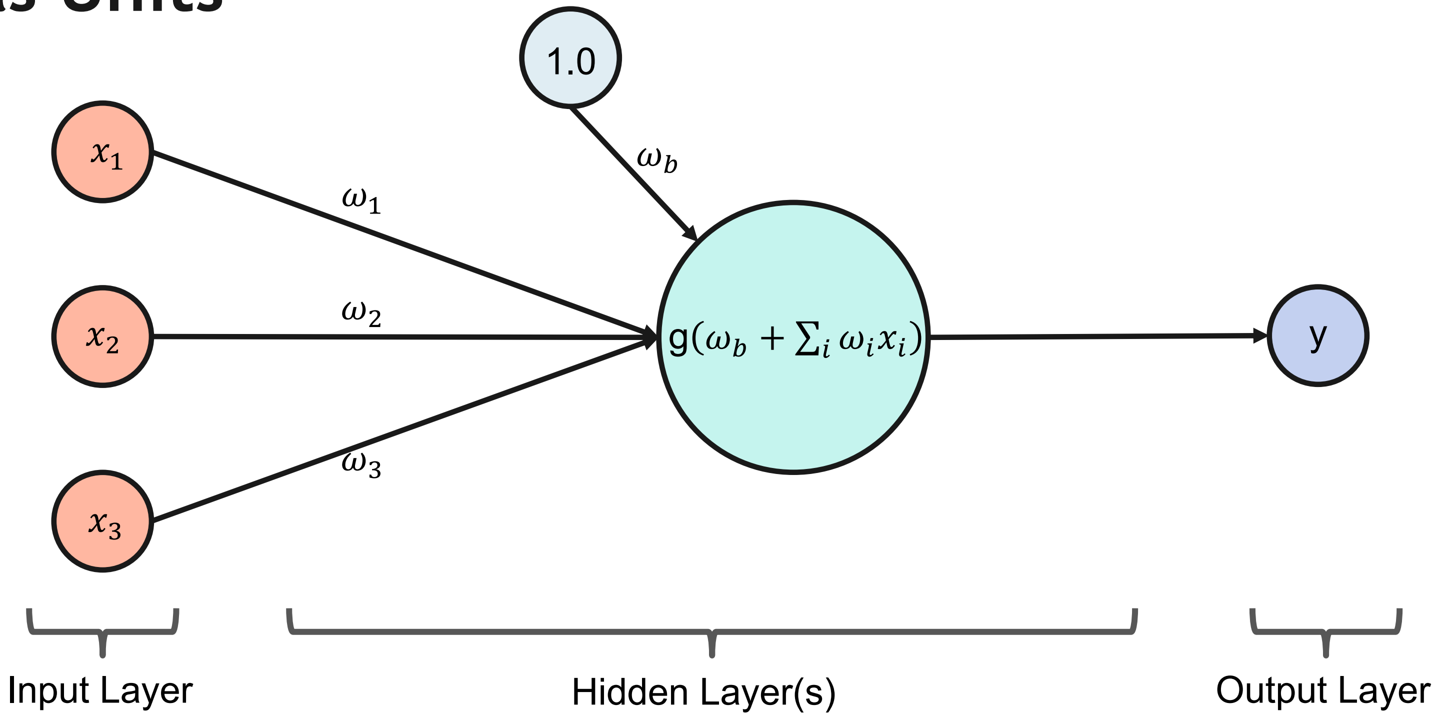
- (I think) they are called activation functions because the first one used was a Heaviside function and thus either “activated” a neuron or not for a given set of inputs.

Popular Choices

- Features people are looking for in activation function:
  - At least piecewise differentiable for reasons we'll elaborate upon later
  - Monotonous
  - Helpful if they bound the output (but not all modern choices do that)
  - Computationally cheap

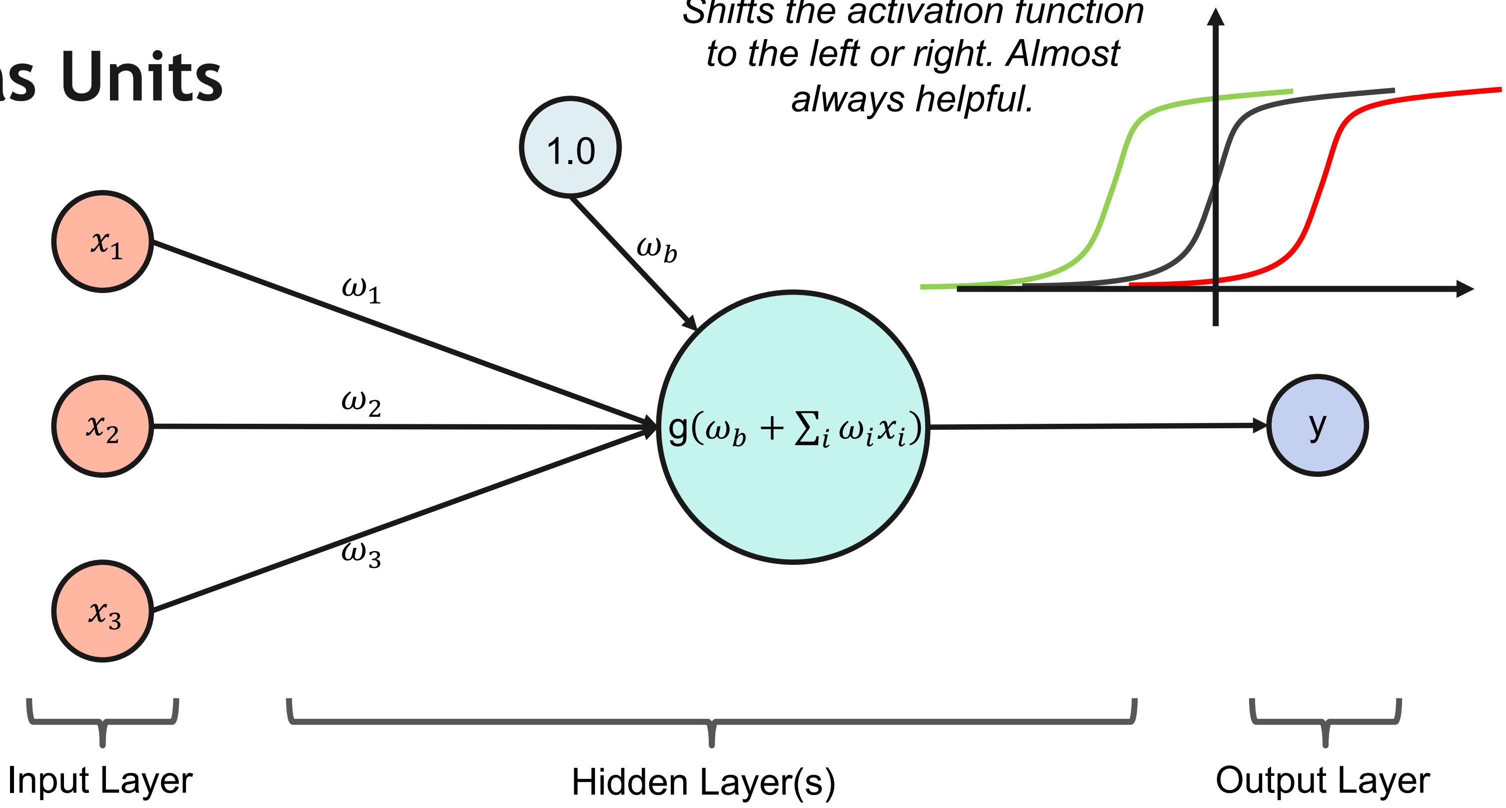


Bias Units


1.0
x_1
x_2
x_3
ω_b
ω_1
ω_2
ω_3
$g(\omega_b + \sum_i \omega_i x_i)$
y
Input Layer
Hidden Layer(s)
Output Layer

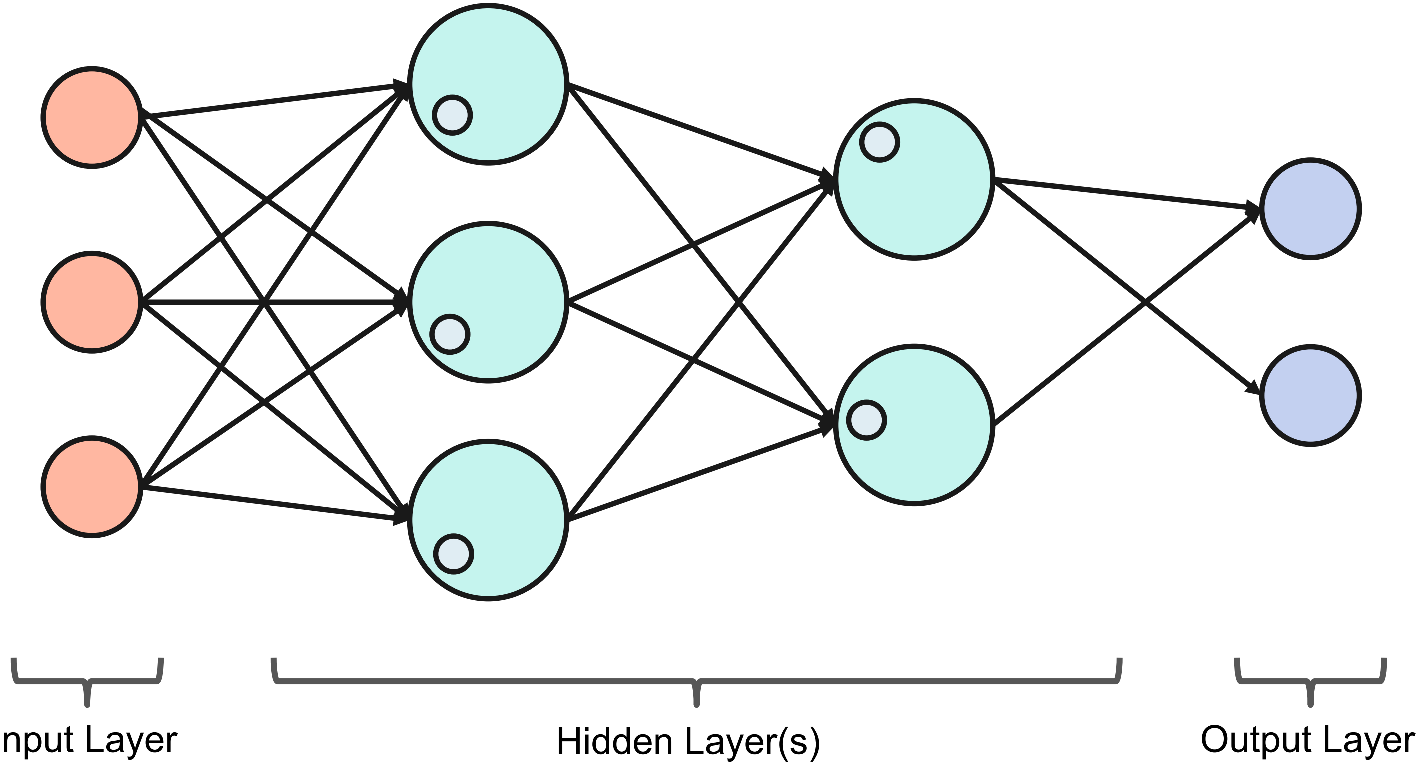
Bias Units

Shifts the activation function to the left or right. Almost always helpful.



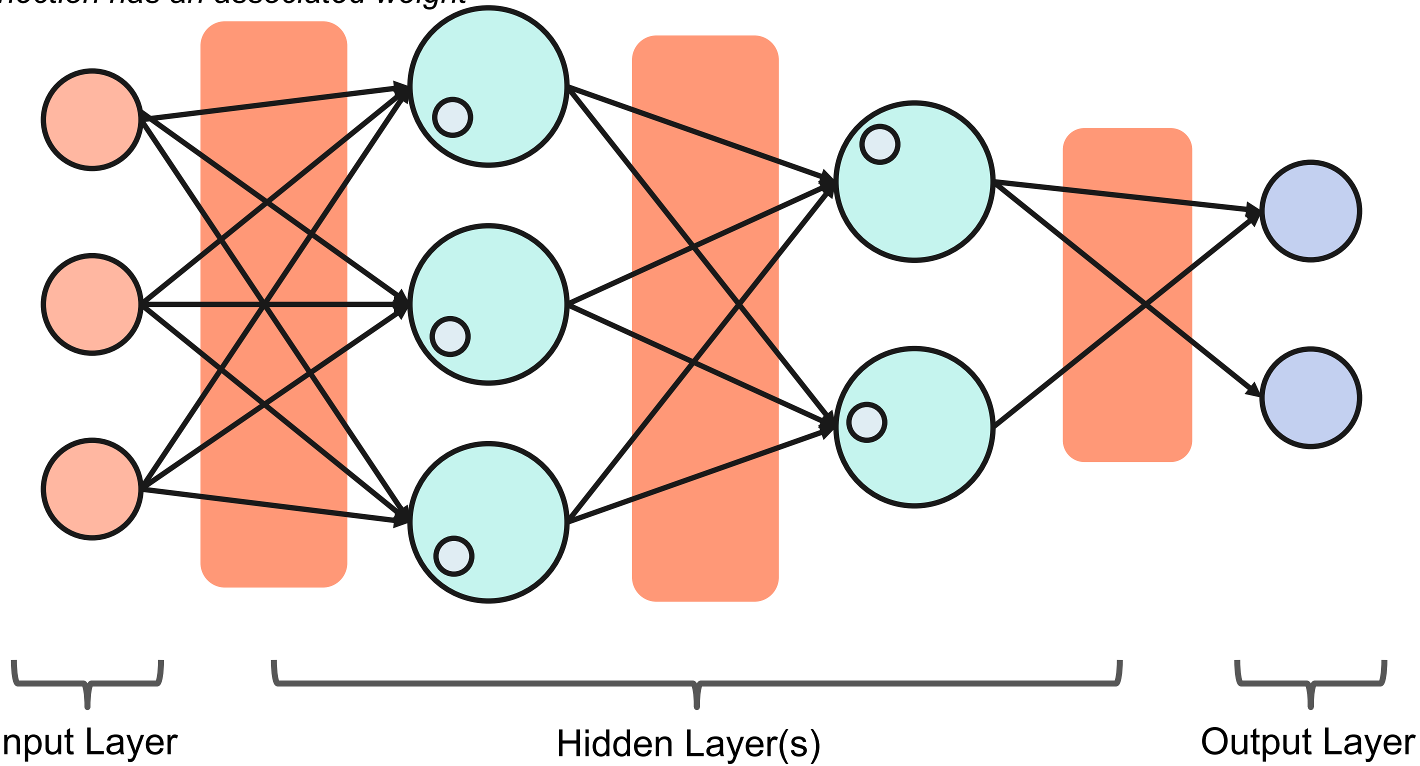
Now let’s start to build a proper network.

Fully-connected Feed-forward Neural Network

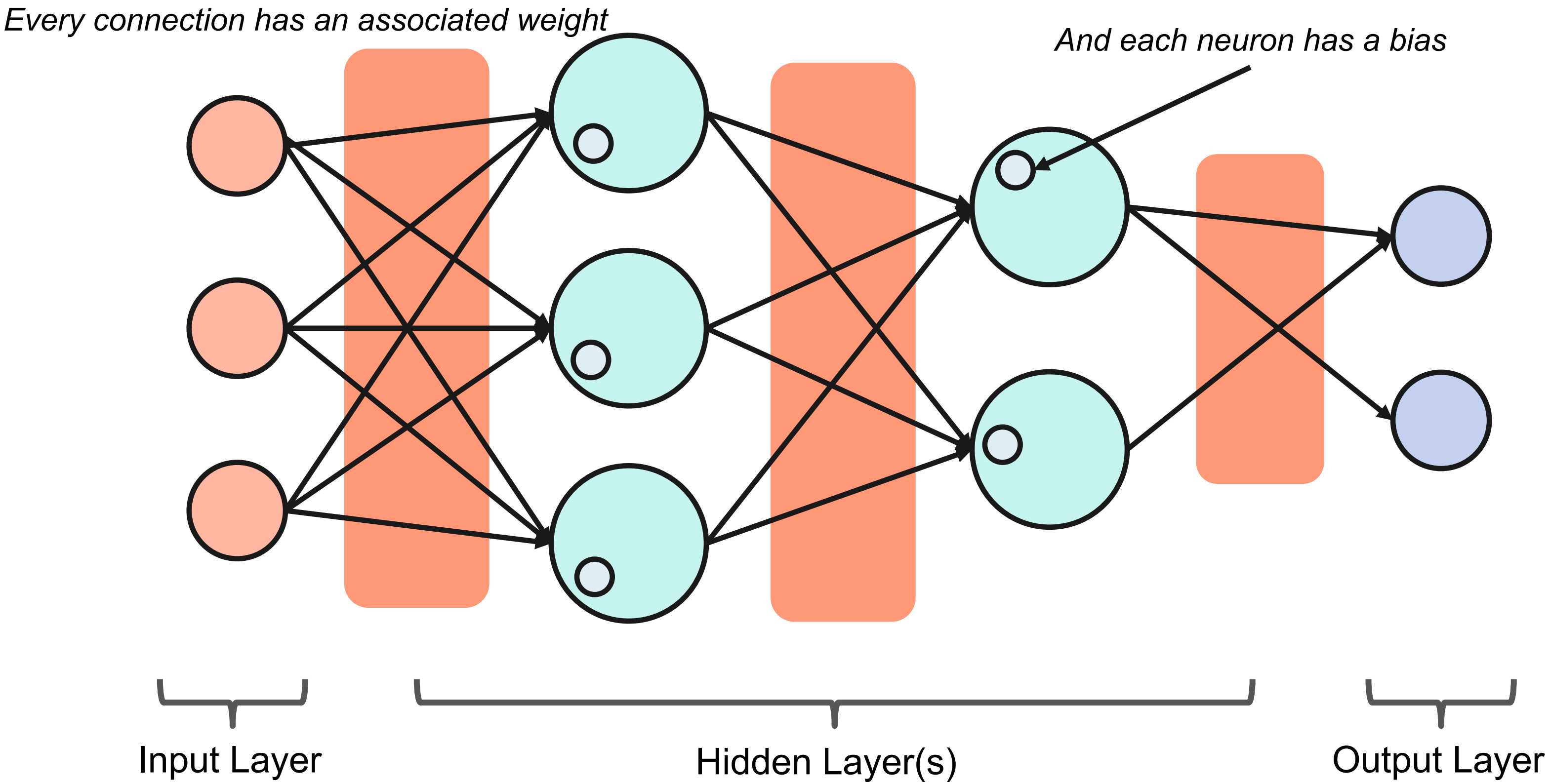


Fully-connected Feed-forward Neural Network

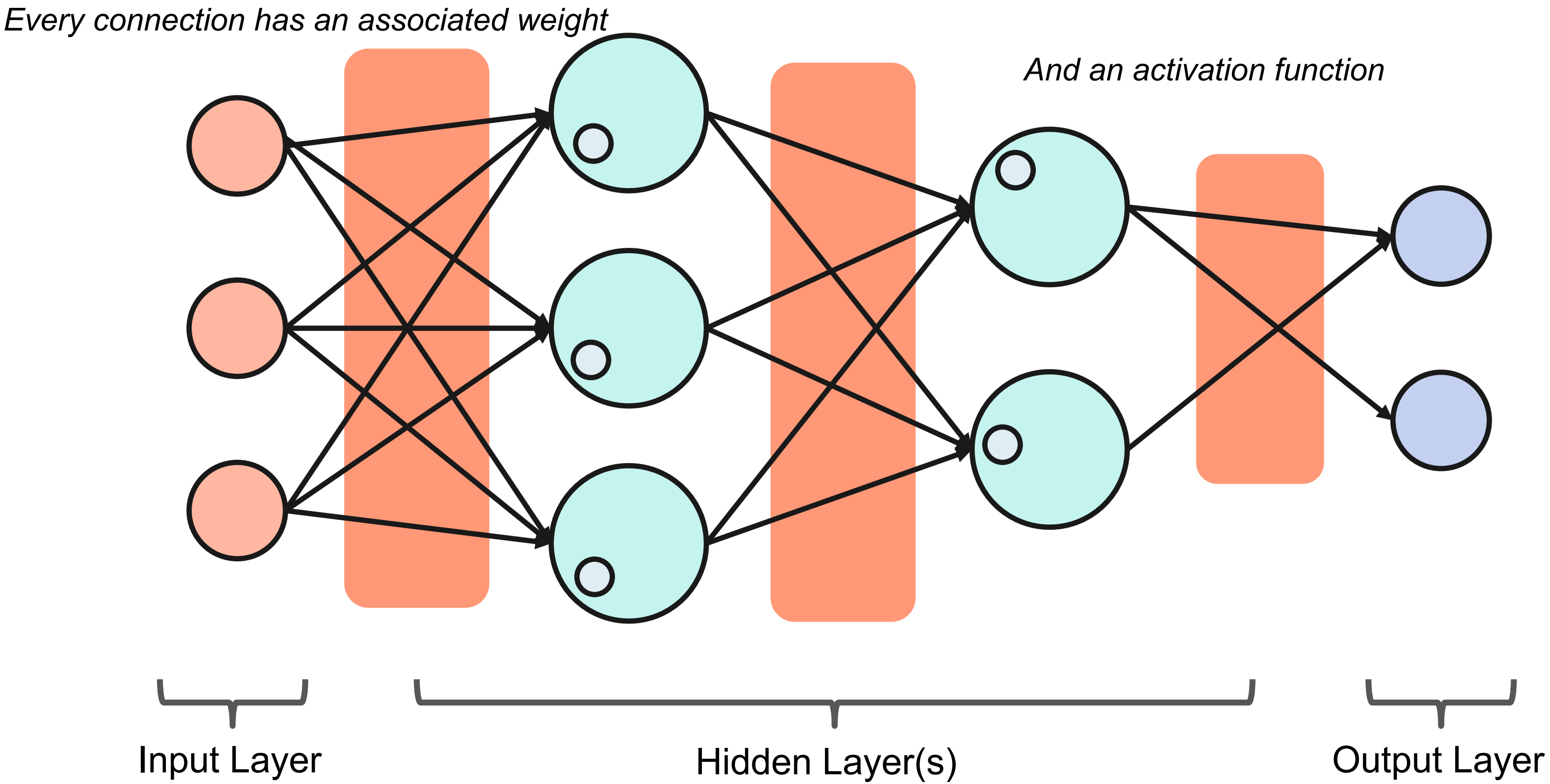
Every connection has an associated weight



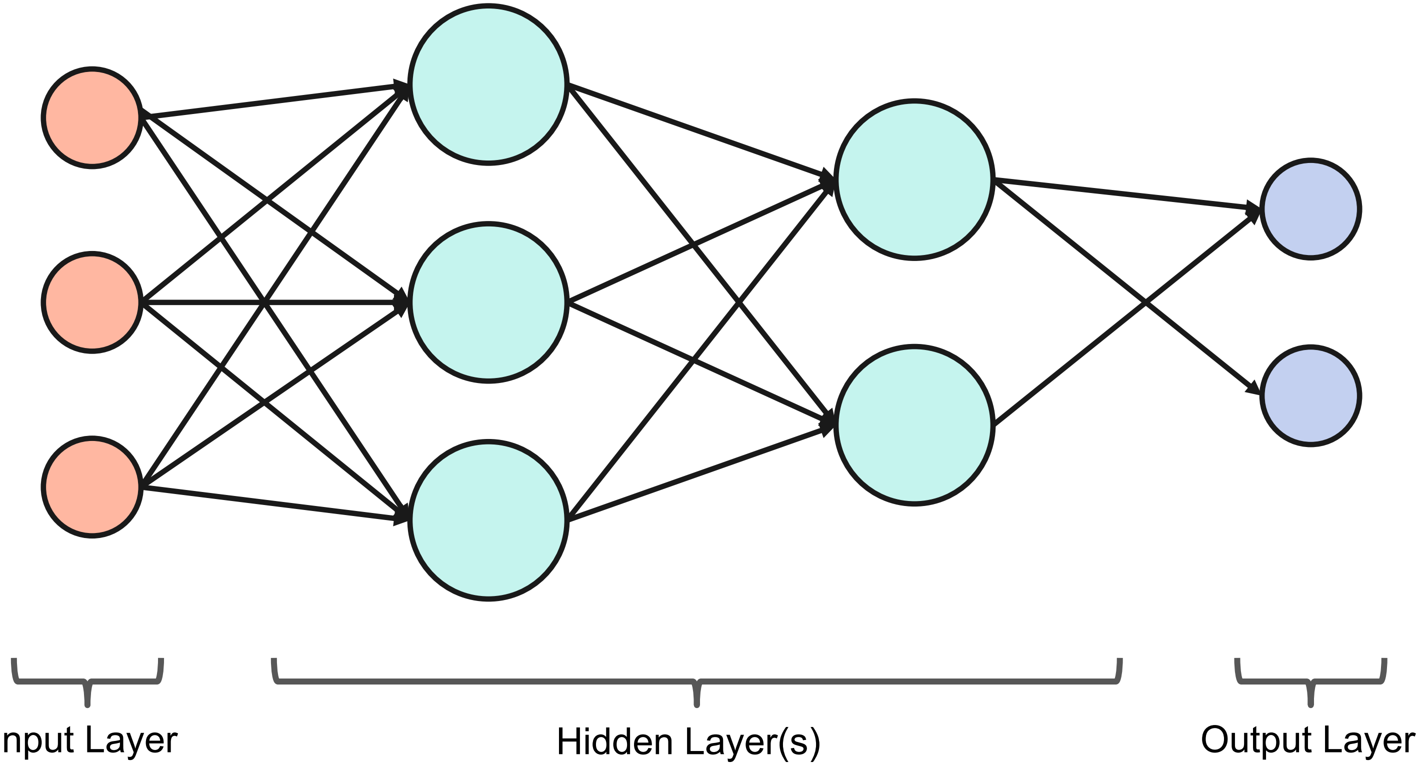
Fully-connected Feed-forward Neural Network



Fully-connected Feed-forward Neural Network



Fully-connected Feed-forward Neural Network



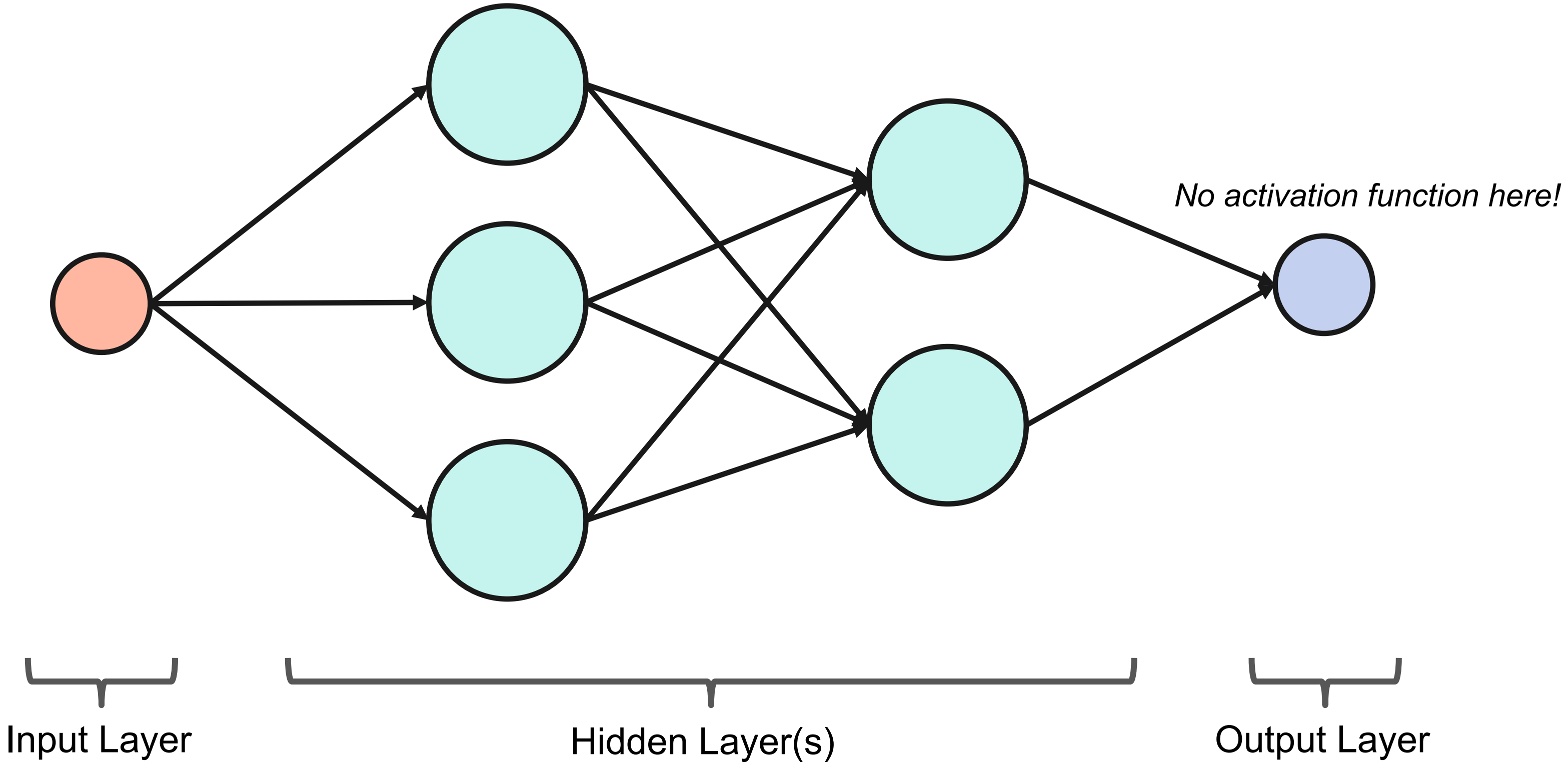
What is gained by adding more “depth”?

- Remember that everything can also be done with a single layer but this layer might get very large.
- If done properly the hope is to decompose a problem in many separate processing or computation steps.
- More layers thus means more steps which allows more complicated functions.
- This generalizes better and also requires fewer parameters.
- Deep networks are discussed in depth on Thursday.

Now we know how a neural network computes something.

Let’s implement this!

Exercise: Forward Pass of this Network



Open the Neural Networks from Scratch Notebook.

How to make it do what we want?

So far we just multiplying numbers together…

$$\text{Input} \longrightarrow f(x) \longrightarrow \text{Output}$$

Input $\longrightarrow$ f(x) $\longrightarrow$ Output

Known input data

Known output data

Input ⟶ f(x) ⟶ Output

Known input data

Optimize for the parameters of f(x)

Known output data

Known input data

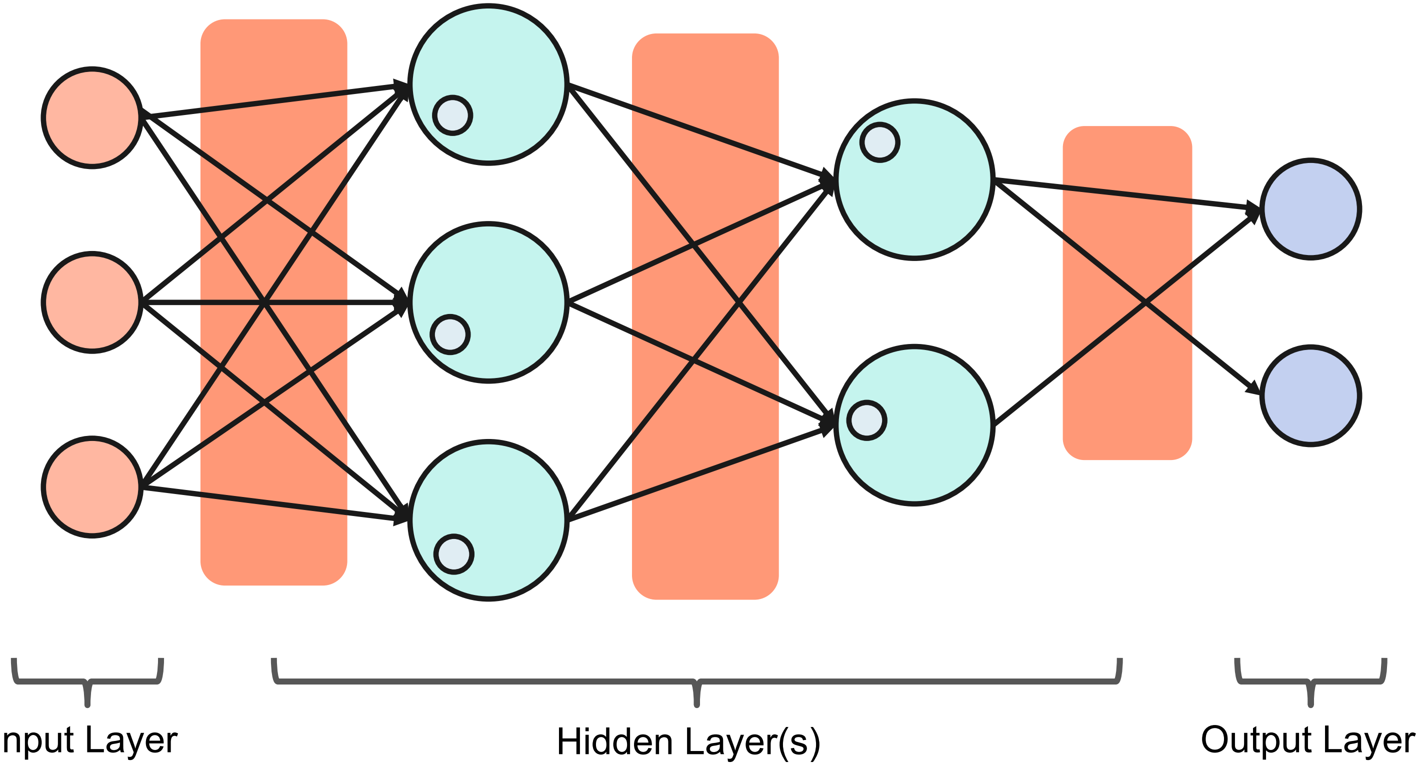
Optimize for the parameters of f(x)

Known output data

Objective for neural networks: Find a set of weights so that it can convert the inputs to output with as little error as possible.

The actual network layout is fixed.

Fully-connected Feed-forward Neural Network



Numerical Optimization Problem

- These networks can have a lot of parameters => Global search methods infeasible
- Derivatives can be computed analytically
    - => Local gradient based optimization

We require two ingredients:

- A way to compute derivatives (back-propagation)
- A way to perform the actual optimization (here we use simple gradient descent)

Back-Propagation with Pen & Paper

What is Back-Propagation?

- Returns the analytic gradients of some objective function/misfit measurement/loss function with respect to the weights of a neural network
- Thus tells us how we have to change the network parameters to increase that loss function. We of course go the other way.
- It is really just the repeated application of the chain rule.
- Works just the same as auto-differentiation does.

Simple Example

$$f(x, y, z) = ((x + y)z)^2$$

Simple Example

$$f(x, y, z) = \big((x + y)z\big)^2$$

Goal:

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

Simple Example

$$f(x, y, z) = \big((x + y)z\big)^2$$

Goal:

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

Chain rule:

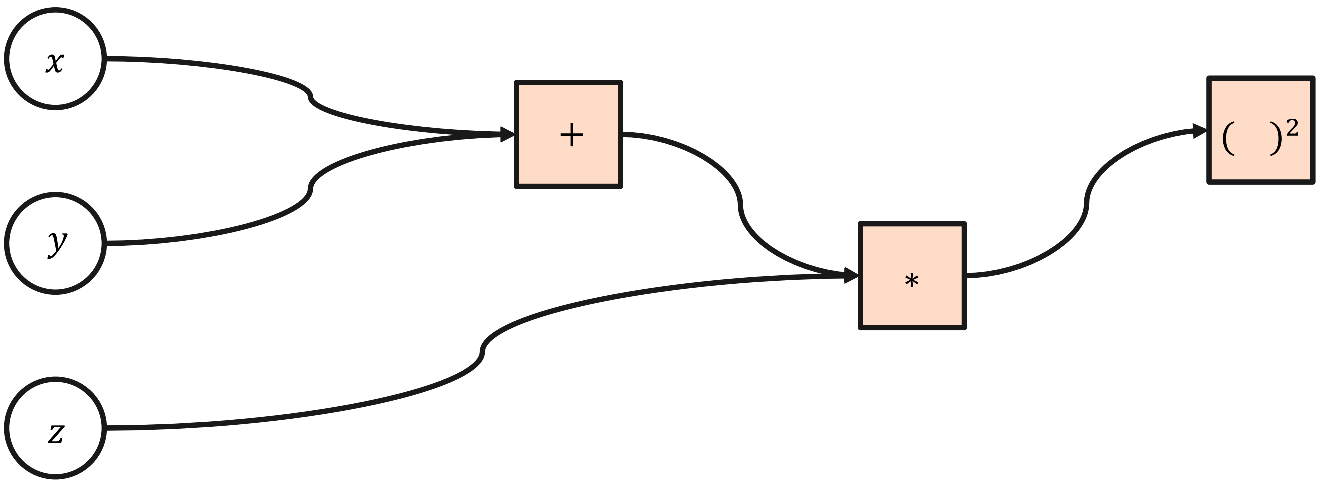
$$\frac{\partial f}{\partial z} = 2\,\big((x + y)z\big)(x + y)$$

of course works, but tedious for deeply nested computations.

And inefficient, assuming that we already calculated for example $(x + y)$ if we evaluated the function once.

$$f(x,y,z) = \left((x+y)z\right)^2$$

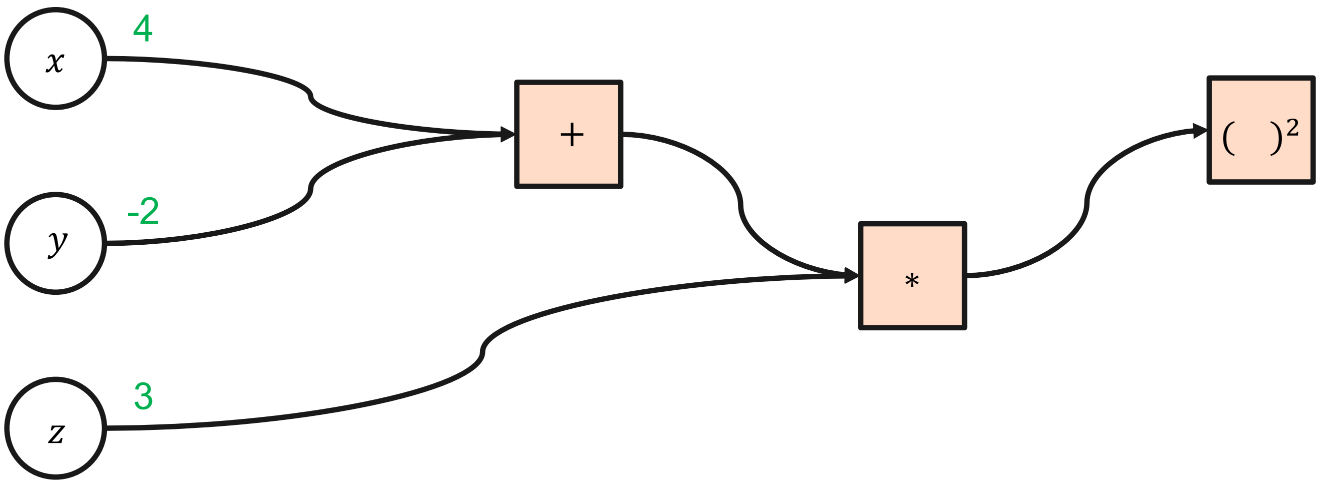
Simple Example



Simple Example

$$f(x, y, z) = \left((x + y)z\right)^2$$

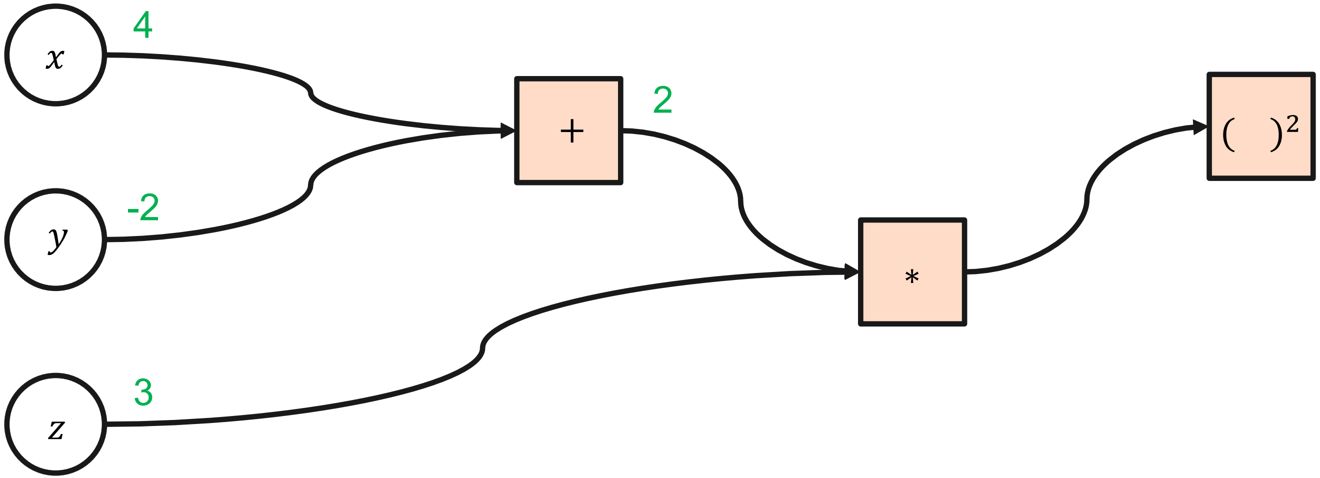
Forward Pass



$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

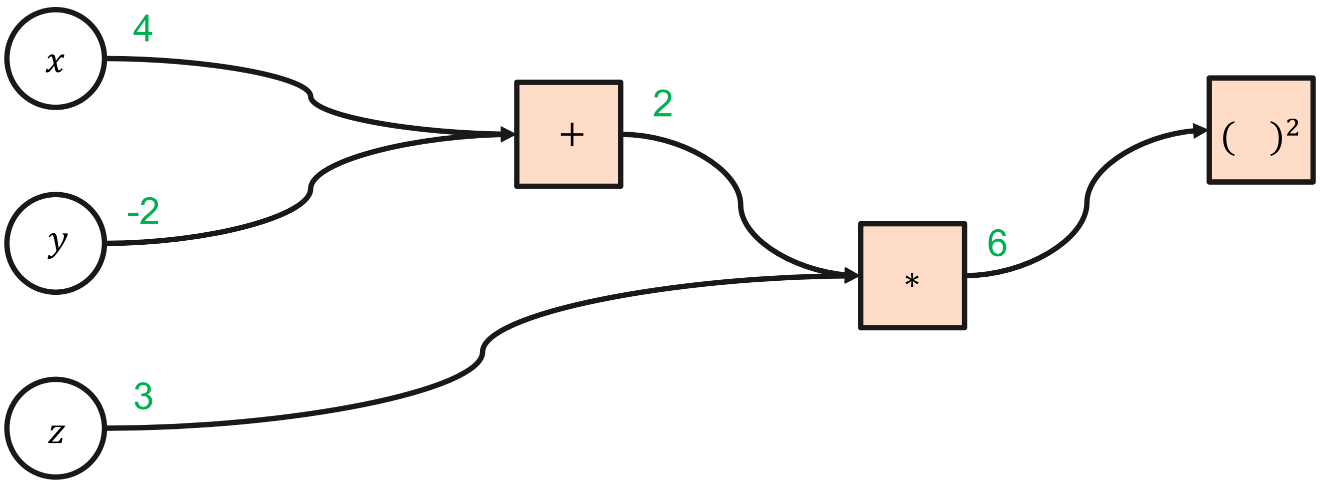
Forward Pass



$$f(x, y, z) = ((x + y)z)^2$$

Simple Example

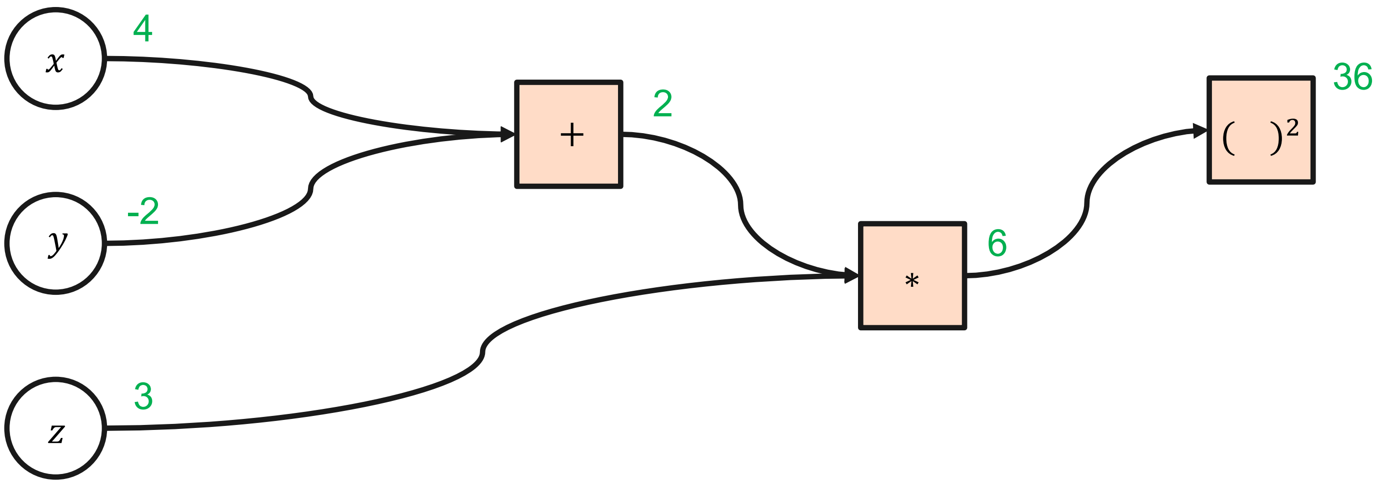
Forward Pass



$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

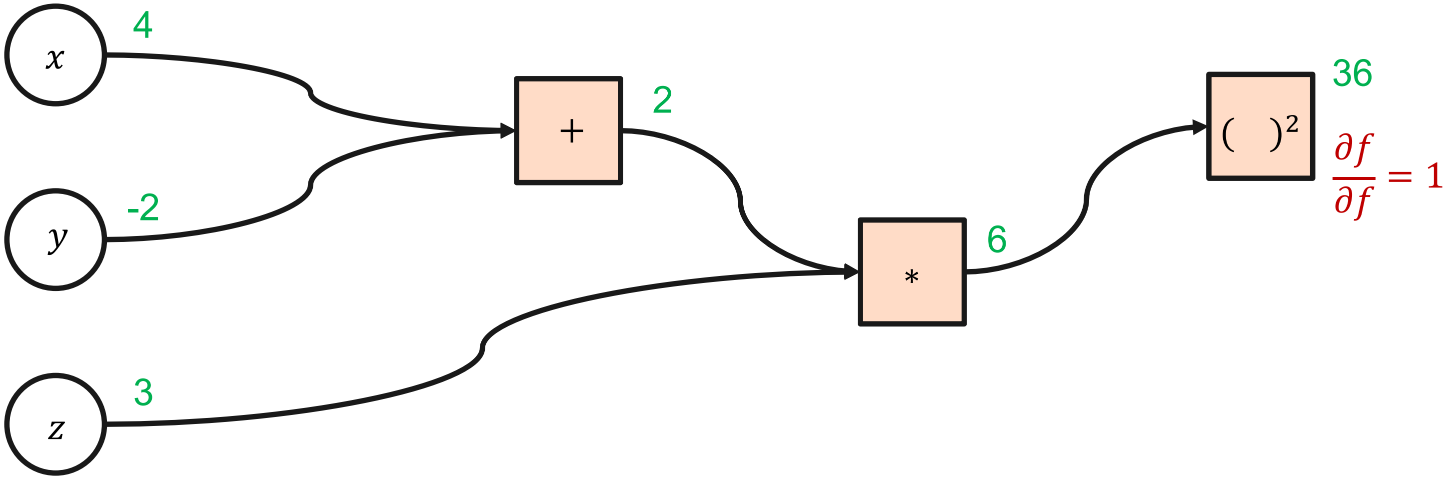
Forward Pass



$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

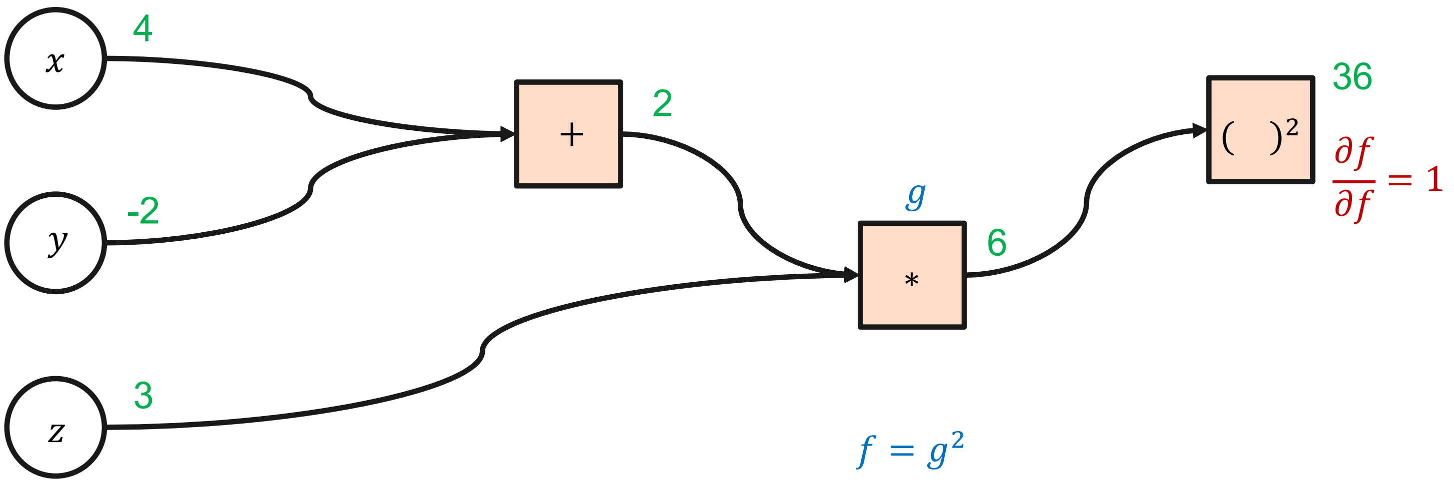
Forward Pass
Backward Pass



$$f(x, y, z) = \left((x + y)z\right)^2$$

Simple Example

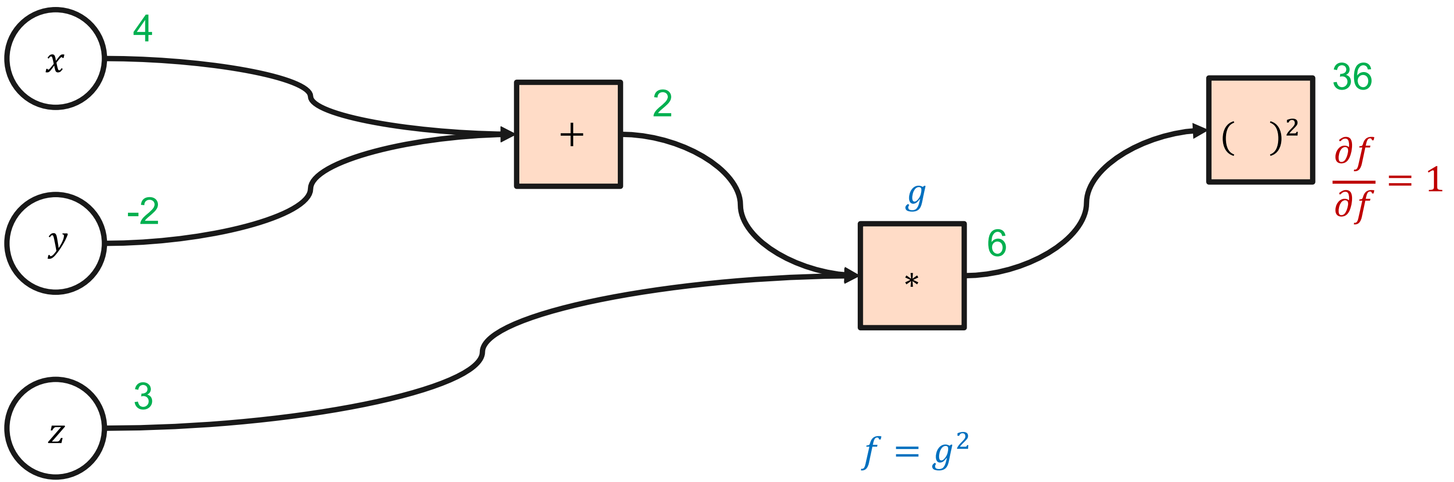
Forward Pass
Backward Pass



$$f(x,y,z) = \big((x+y)z\big)^2$$

Simple Example

Forward Pass
Backward Pass



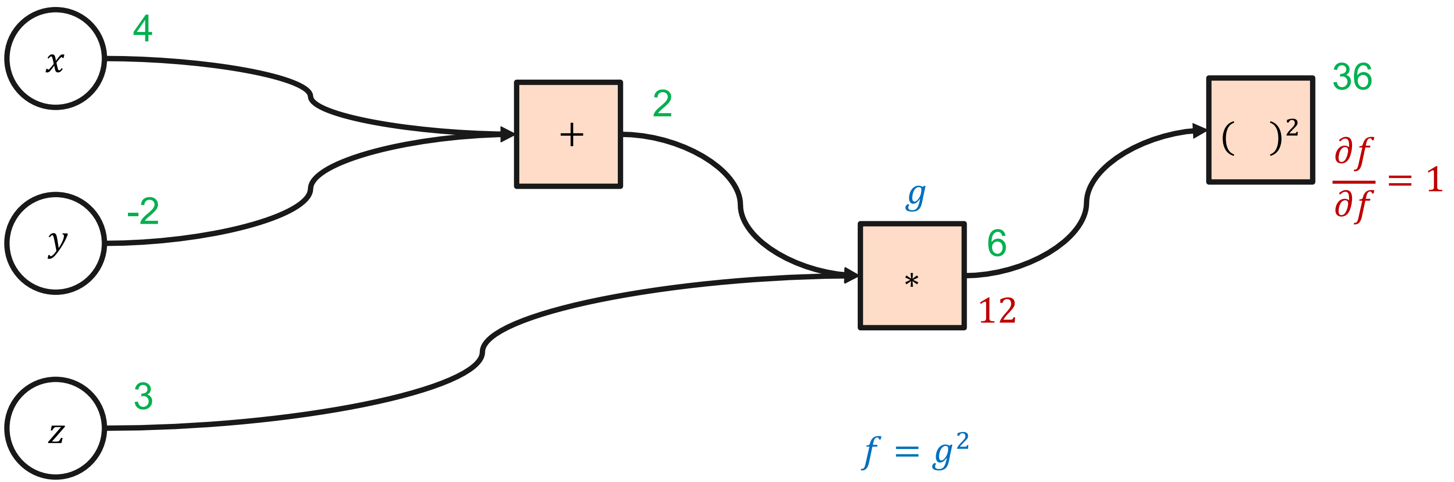
$$f = g^2$$

$$\frac{\partial f}{\partial g} = 2g$$

$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

Forward Pass
Backward Pass



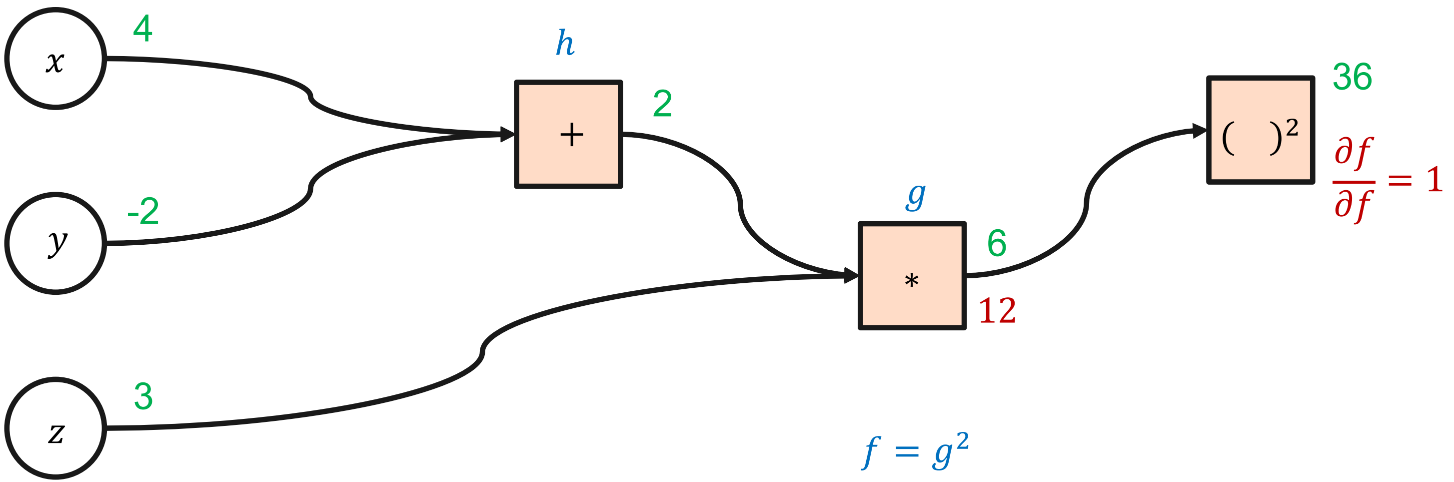
$$f = g^2$$

$$\frac{\partial f}{\partial g} = 2g$$

$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

Forward Pass
Backward Pass



$$f = g^2$$

$$\frac{\partial f}{\partial g} = 2g$$

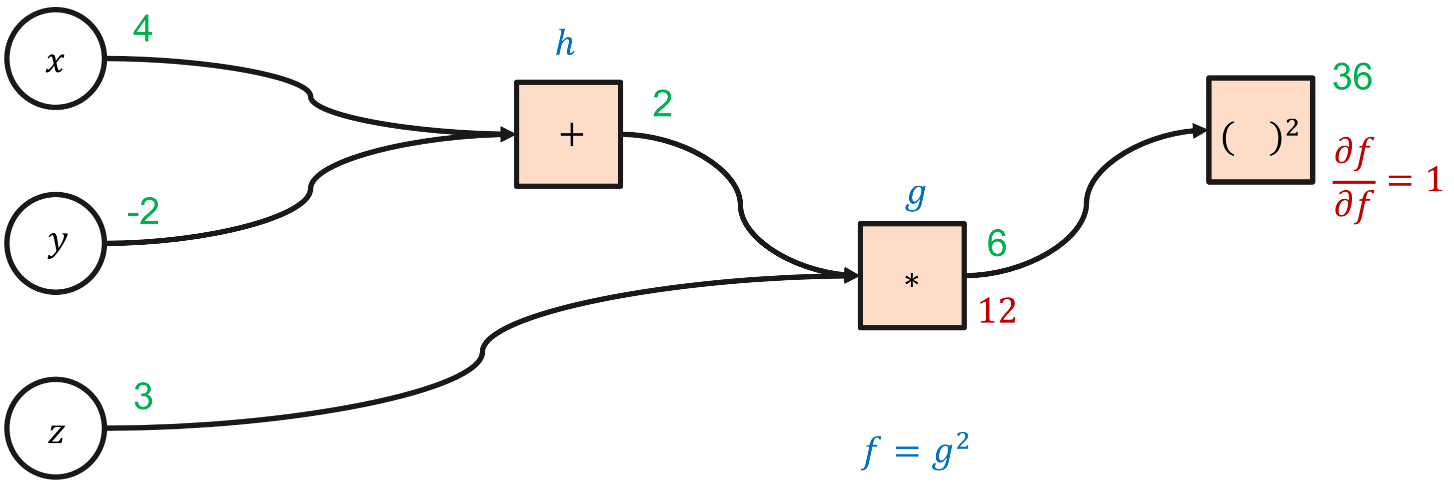
Simple Example

$$f(x, y, z) = ((x + y)z)^2$$

$$g = hz$$

$$\frac{\partial g}{\partial z} = h$$

Forward Pass
Backward Pass



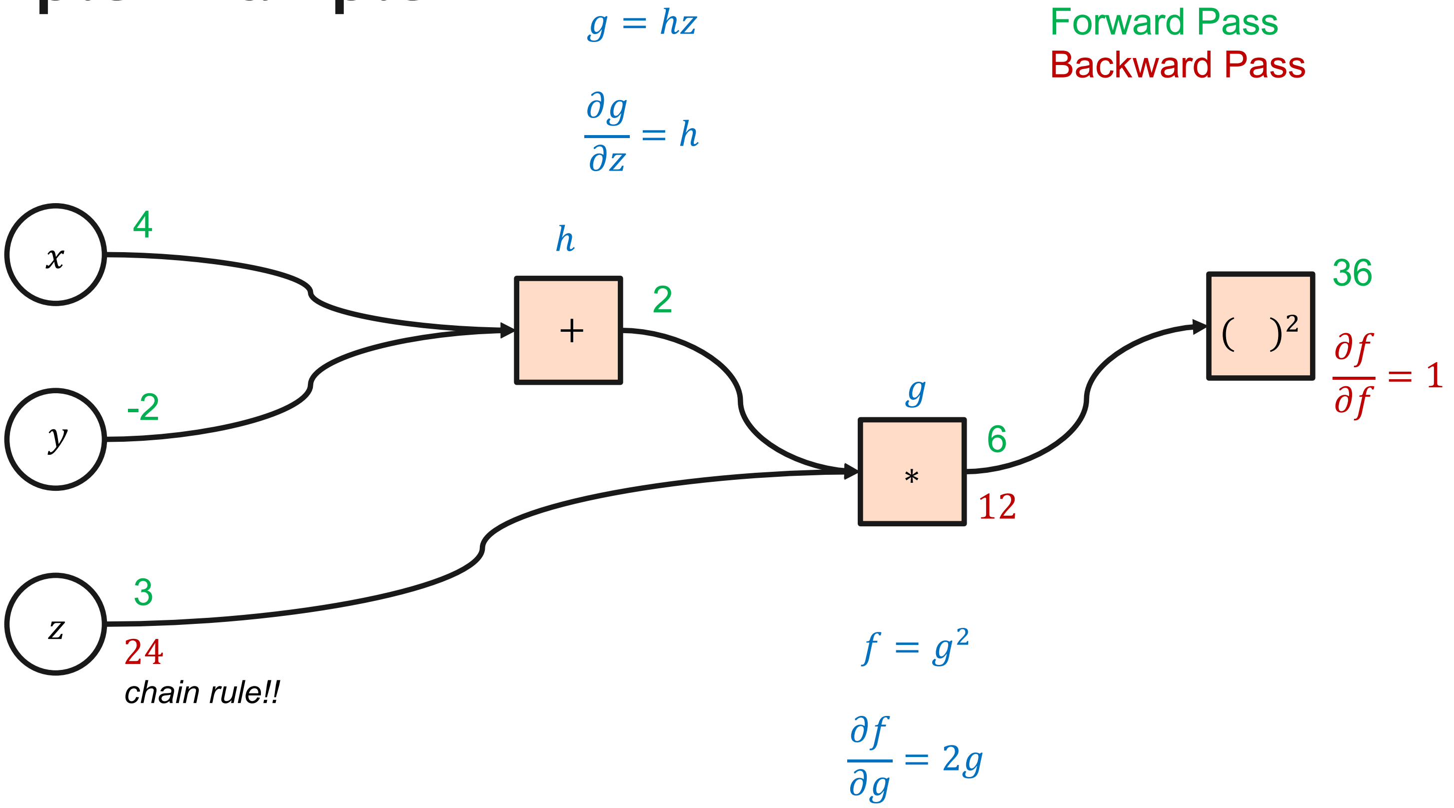
$$f = g^2$$

$$\frac{\partial f}{\partial g} = 2g$$

$$f(x,y,z) = \left((x+y)z\right)^2$$

Simple Example

Forward Pass
Backward Pass



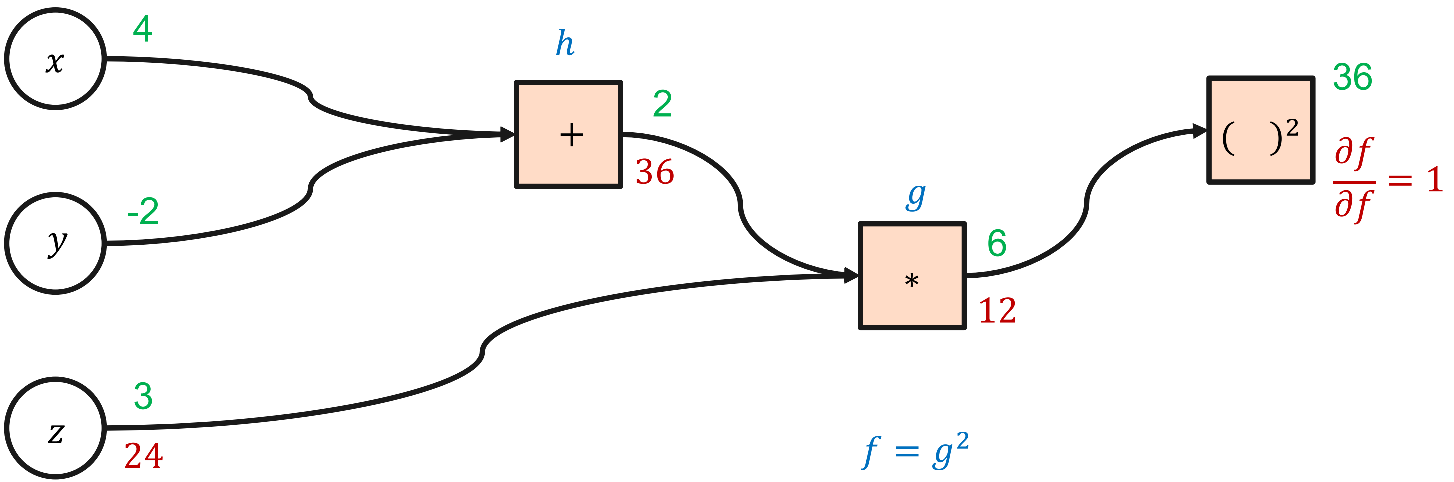
Simple Example

$$f(x, y, z) = \big((x + y)z\big)^2$$

Forward Pass
Backward Pass

$$g = hz$$

$$\frac{\partial g}{\partial z} = h$$

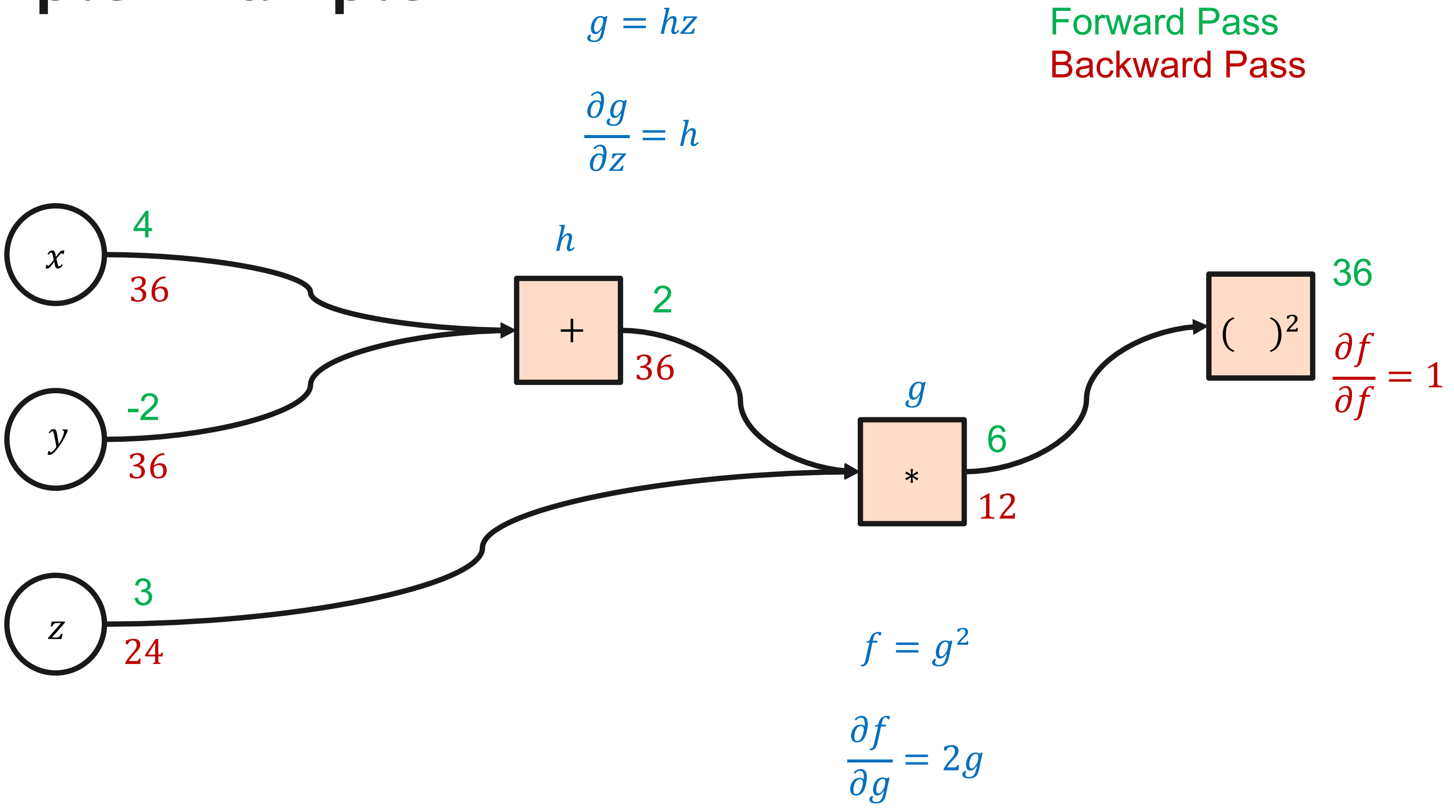


$$f = g^2$$

$$\frac{\partial f}{\partial g} = 2g$$

$$f(x, y, z) = \left((x + y)z\right)^2$$

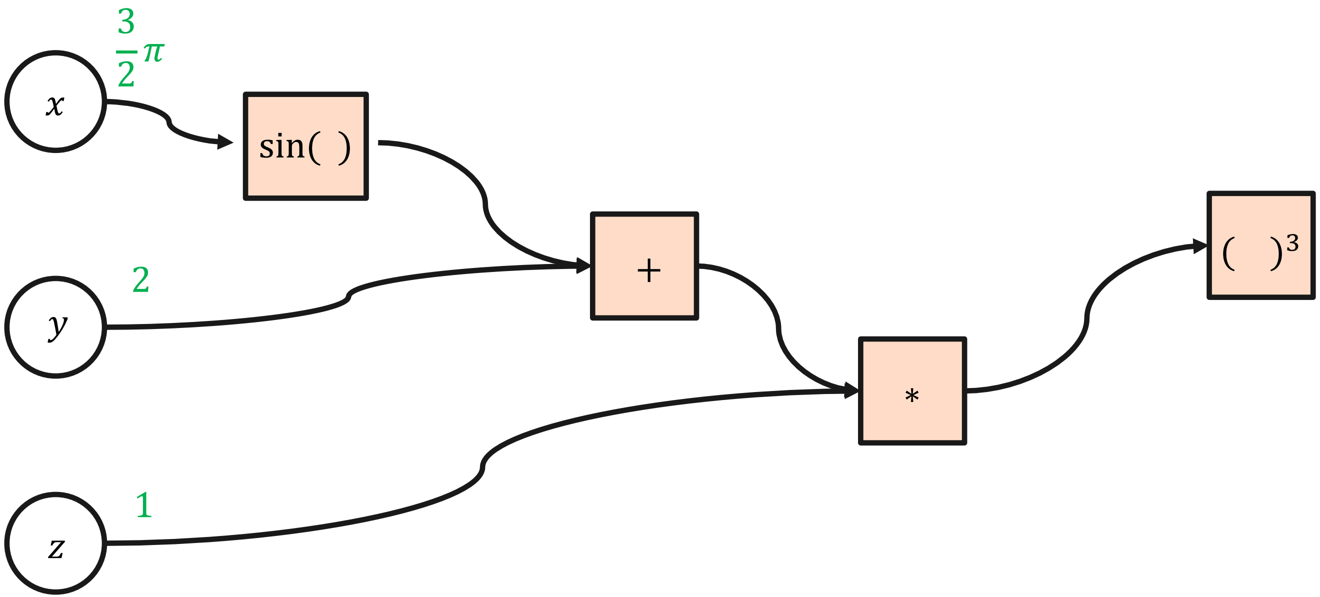
Simple Example



Exercise

$$f(x, y, z) = \left((\sin(x) + y)z\right)^3$$

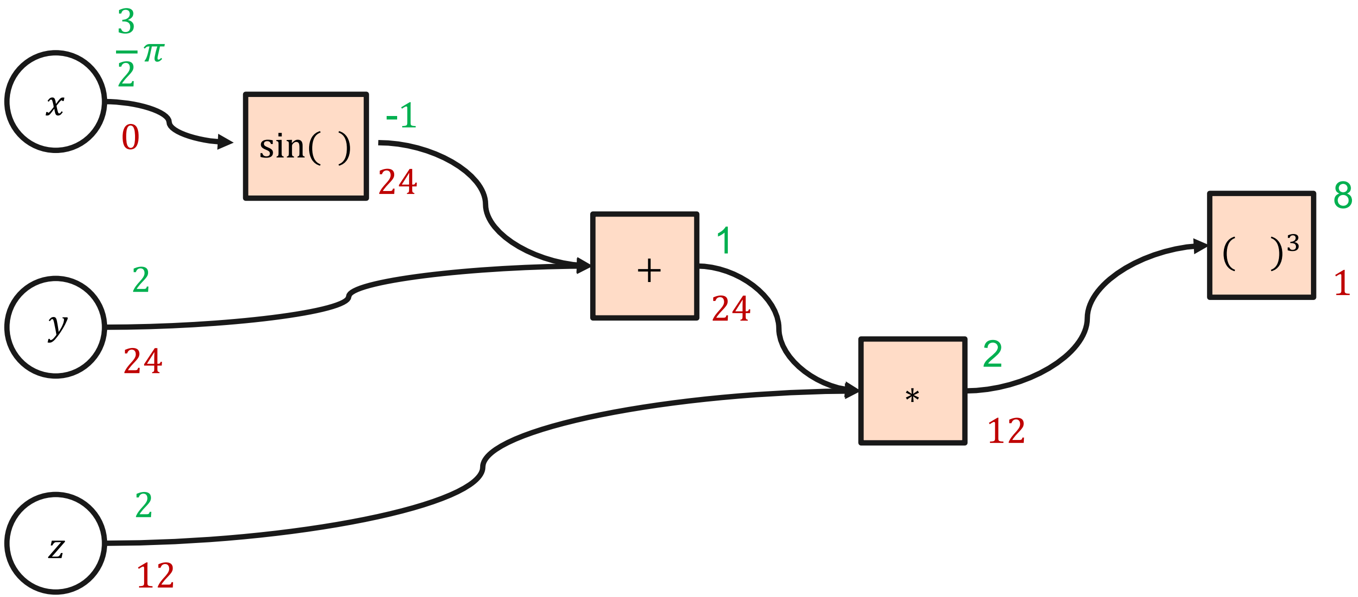
Forward Pass



$$f(x,y,z) = \left((\sin(x) + y)z\right)^3$$

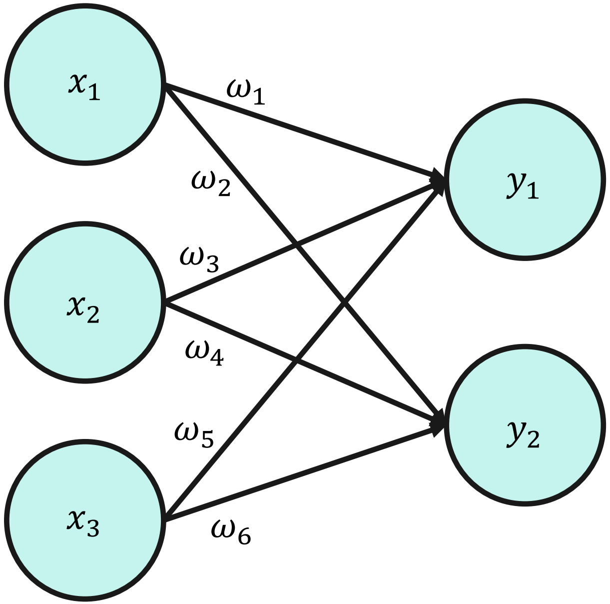
Exercise

Forward Pass
Backward Pass

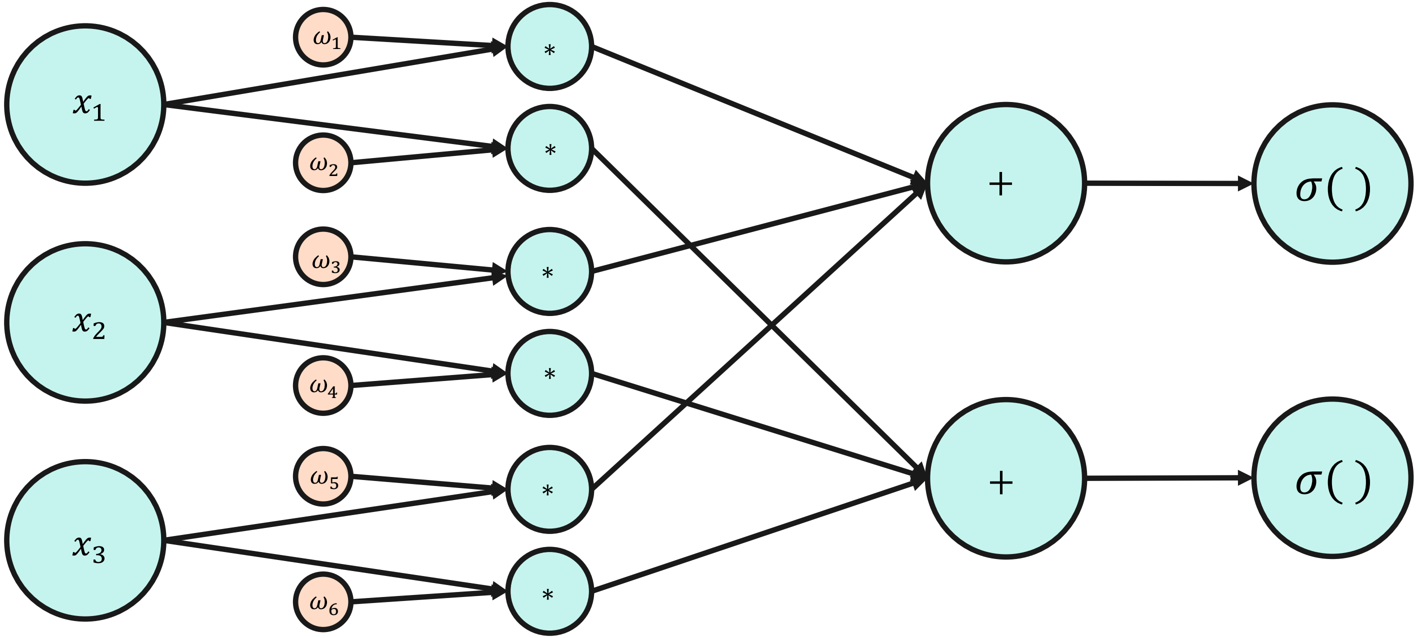


How to do Backpropagation Through This Layer?

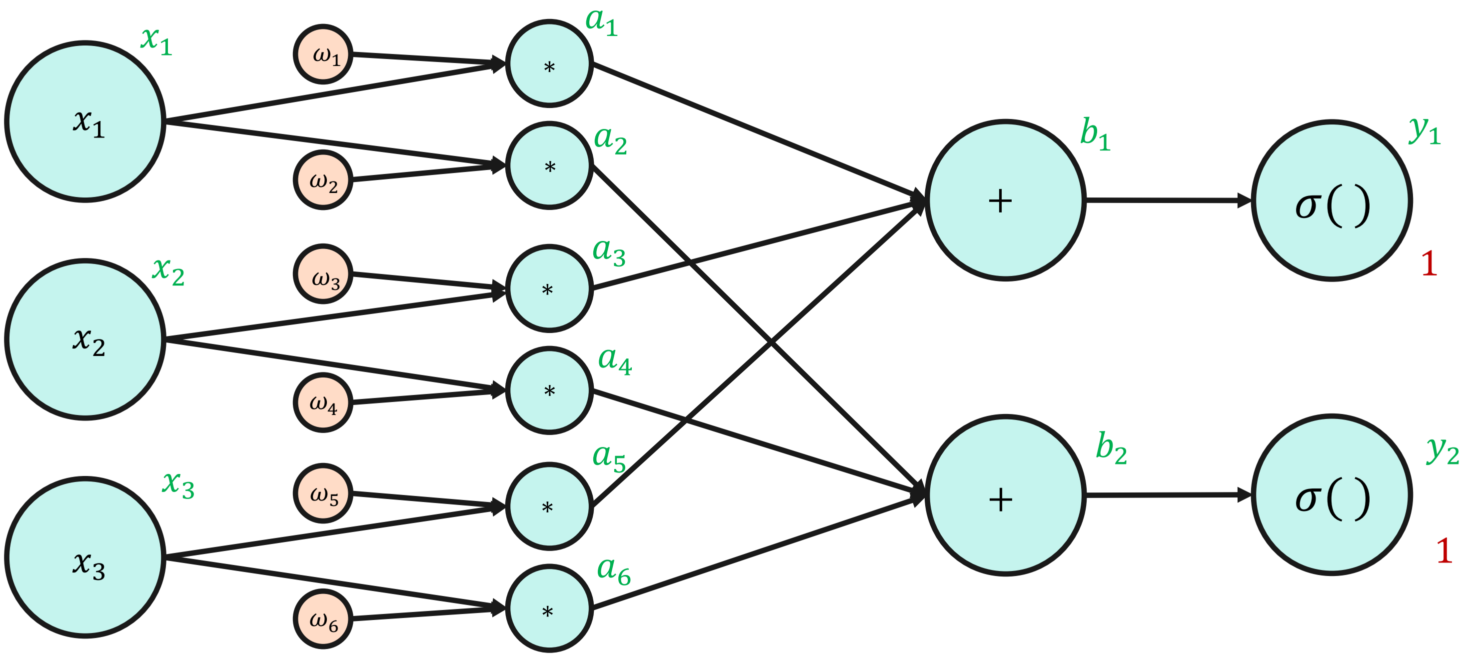
activation function already applied



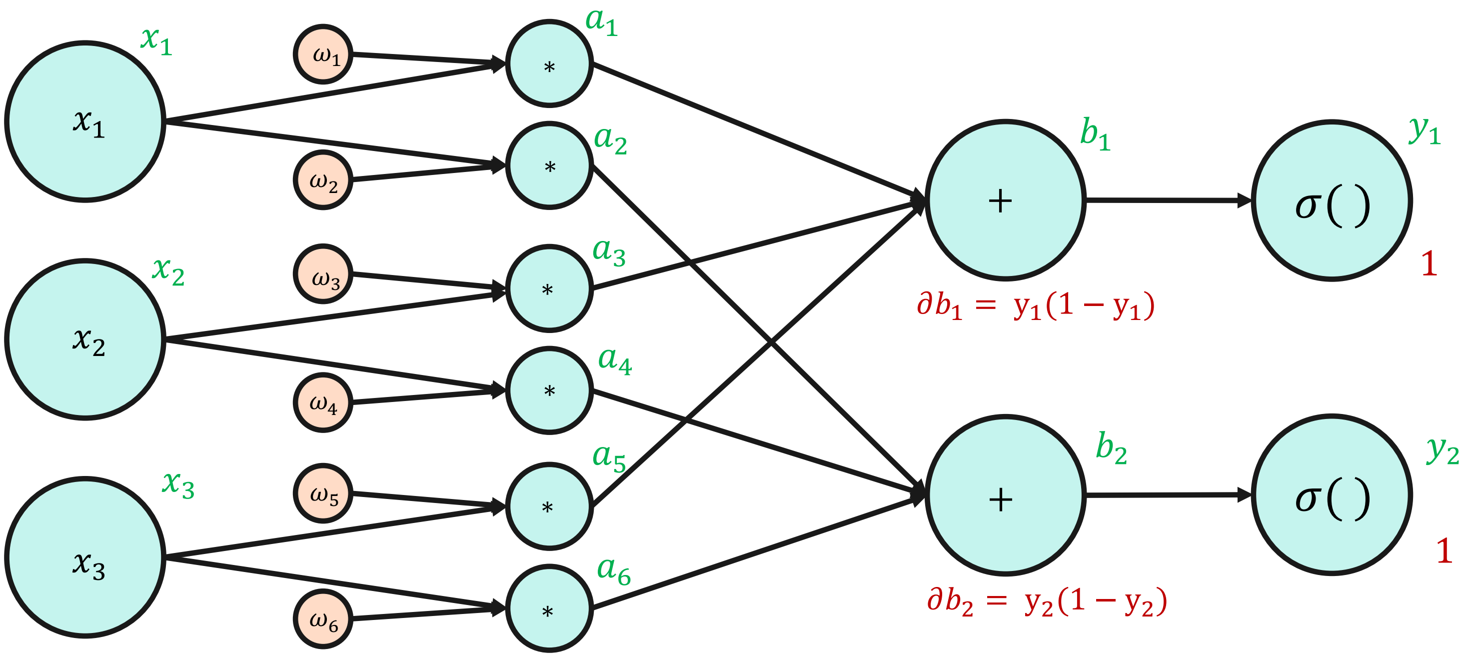
requires activation function



x_1
x_2
x_3
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
*
*
*
*
*
*
+
+
$\sigma(\,)$
$\sigma(\,)$

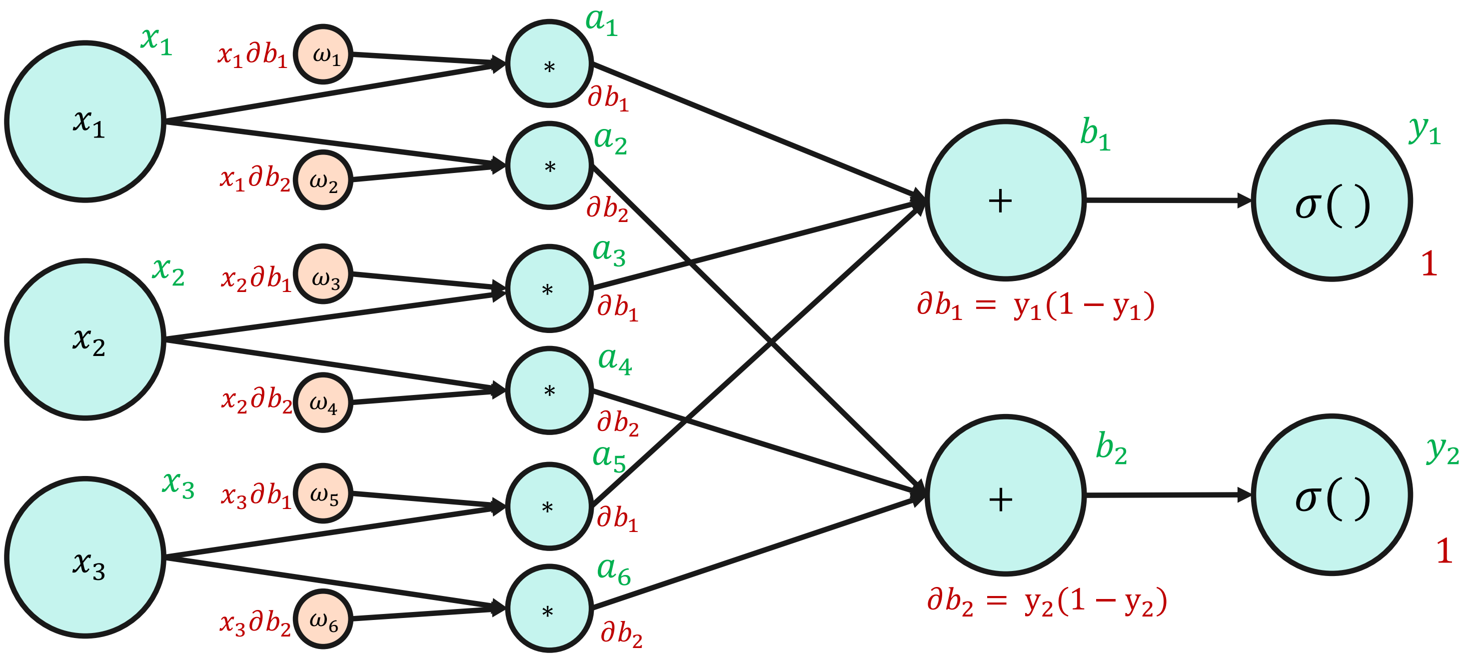

x_1
x_1
x_2
x_2
x_3
x_3
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
a_1
a_2
a_3
a_4
a_5
a_6
*
b_1
b_2
+
y_1
y_2
$\sigma(\,)$
1
1

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

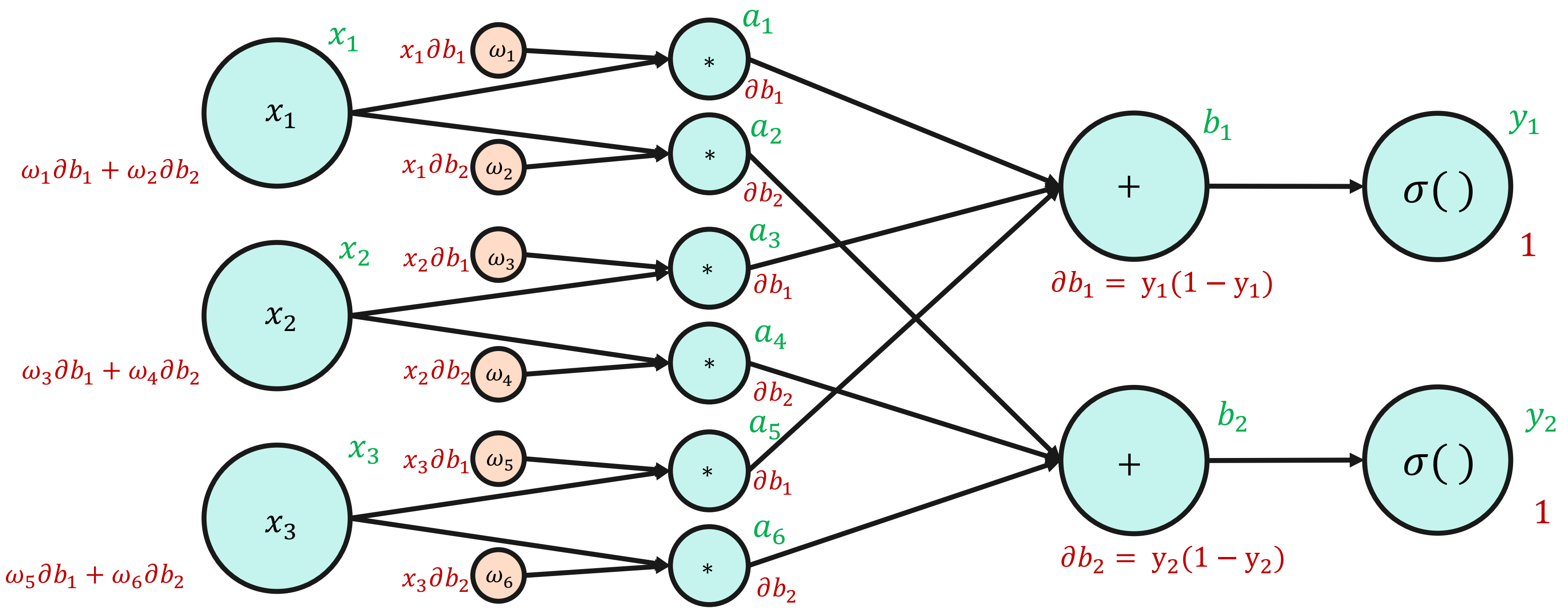


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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



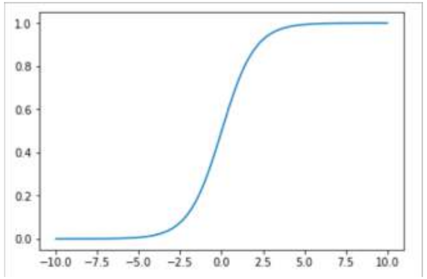
$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$



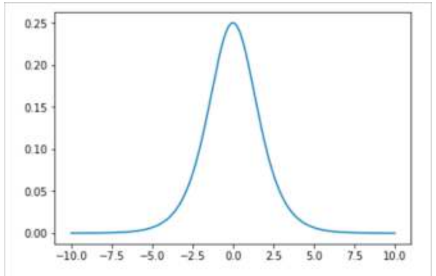
Everything can again be expressed using matrix multiplications.

Also note that we require all x and y values from the forwards pass!

Sigmoid Activation Function



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



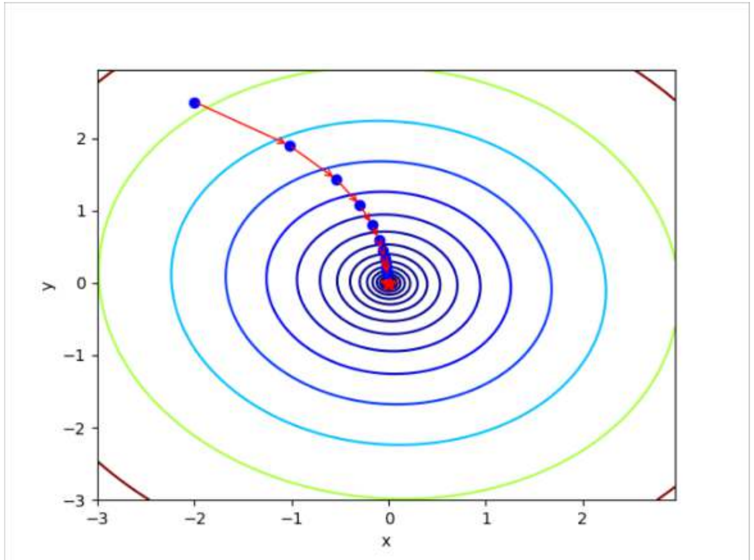
- Remember that back-propagation always multiplies with the previous gradients
- The maximum slop of the sigmoid function is 0.25 => very slow training for the first layers in deep network if sigmoid or similar activation functions are used.
- Gradients also very small for large absolute values
- Vanishing gradients
- Other activation functions are used for deep learning (see Thursday)

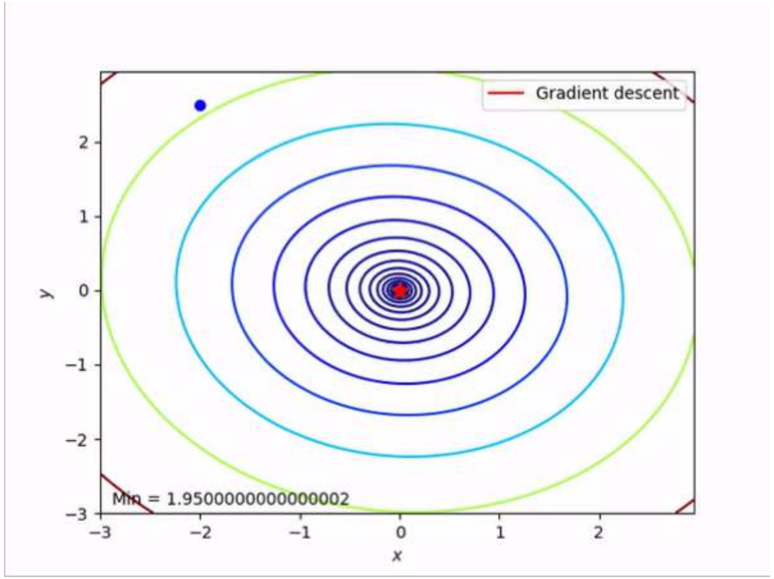
Let’s look at some code.

Gradient Descent

- Recall that we ruled out global optimization as there are just too many parameters in the network.
- And we have an analytic way to compute exact derivatives to all the parameters in our network.
- The most basic gradient based optimization algorithm is gradient descent:
    - Compute the gradient of the objective/loss function at the current point
    - Choose a step-length and walk in the direction of the negative gradient
    - Rinse and repeat

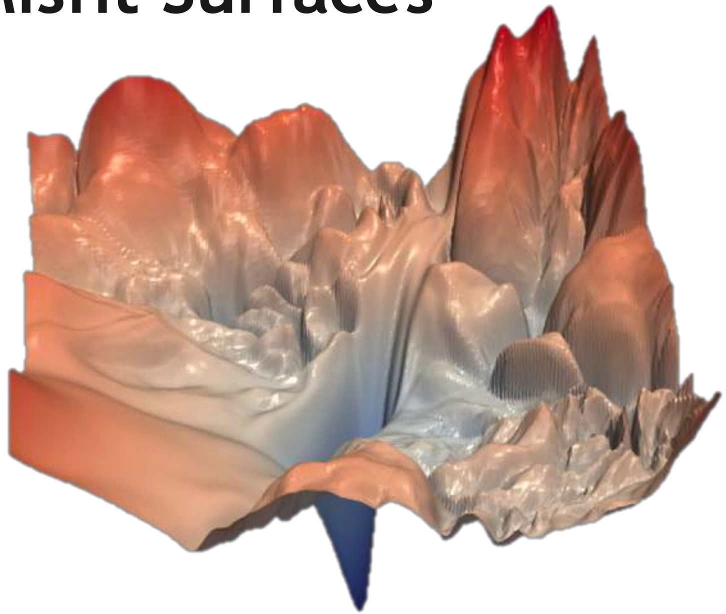
Gradient Descent Illustrations

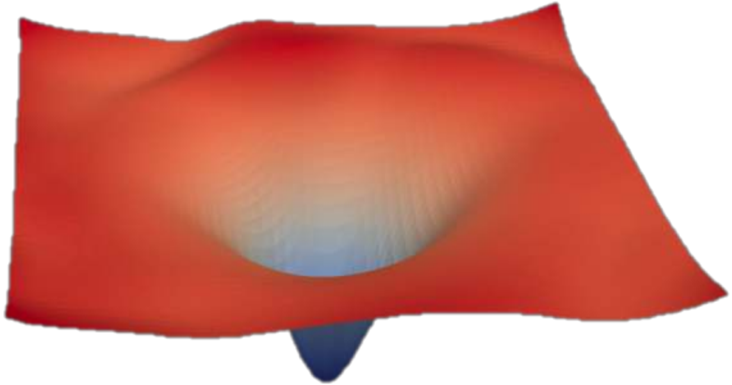




https://jed-ai.github.io/py1_gd_animation/

Misfit Surfaces





Li, Hao, et al. "Visualizing the loss landscape of neural nets." Advances in Neural Information Processing Systems. 2018.

Exercise: Implement Gradient Descent

How to Learn a Neural Network

- Design the network architecture (e.g number of inputs and outputs, number of layers, neurons in each layer, activation functions, ...).
- Loop until convergence:
    - Take a data sample, run a forward pass
    - Compute the loss of the result by comparing to the training data
    - Compute the gradient of the loss function with respect to the weights of the neural network via backpropagation
    - Update the weights with a chosen gradient based optimization algorithm

Have another look at the code.

Practicalities

- Straightforward gradient descent is rarely used. People instead use some for of stochastic gradient descent
- Higher-order methods like L-BFGS are also rarely used.
- The vanishing gradient problem needs to be dealt with.
- Regularization is important and all kinds of fancy tricks are done.
- Inputs and outputs are usually normalized in some fashion. Thus to actually use a neural network the outputs then have to be “denormalized” again.

Stochastic Gradient Descent

- Instead of computing the cumulative gradient of the whole training data set at once, compute it in so-called mini batches and update after each batch.
- This is called mini-batch gradient descent (stochastic gradient descent would be to update every time).
- Some momentum term is usually added to the optimization algorithm to deal with outliers in the gradients.

- Ends up being very fast (more efficient on GPUs to run many in parallel).
- Results in faster updates for highly redundant data.
- As the gradient is slightly different for each batch it can potentially get out of local minima again.

- State-of-the-art are adaptive mini-batched gradient descent methods like Adam, AdaGrad, RMSProb, …

http://ruder.io/optimizing-gradient-descent

Thank you.