Neural Networks

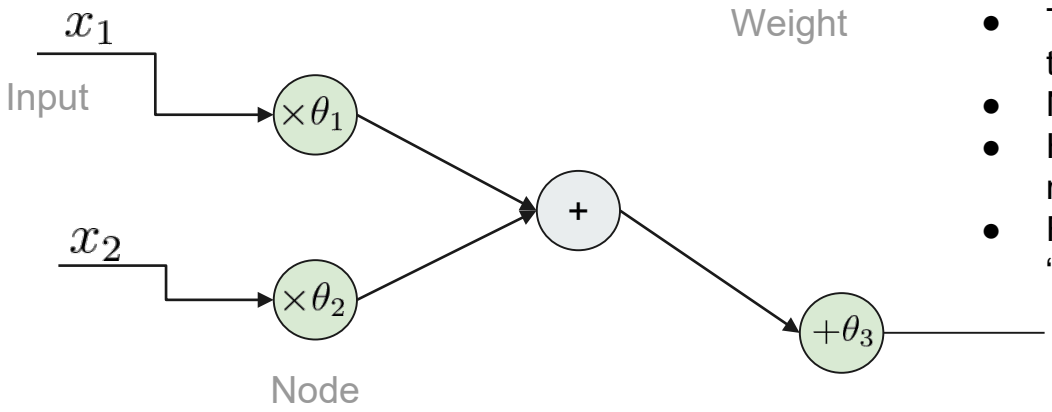
Bill Paivine, Howie Choset

Intro to Robotics 16 -311

NNs are computation graphs

A computation graph shows the structure of some computation in a directed graphical form, making partial results more clear.

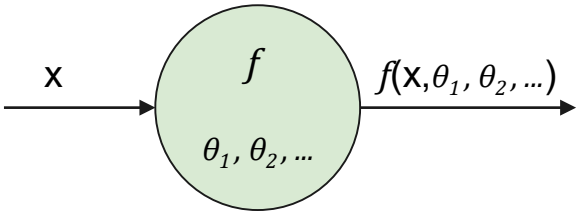
$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

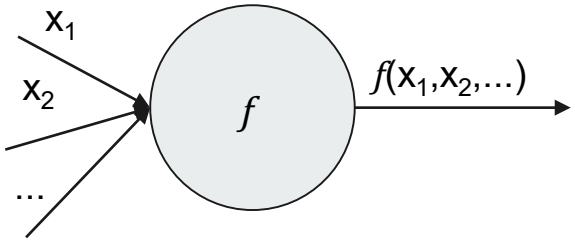


- The left computation graph corresponds to the computation which computes f
- Nodes represent “operations”
- Here, θ_i represents a “weight”, and x_i represents an “input”
- Following the arrow direction computes the “forward” pass of the graph

Structure of Computation Graph

Graph is directed and acyclic
Two types of nodes:





Trainable Node:
Contains one or more “trainable” parameters, which are variables affecting the output of the graph but can be changed during training.

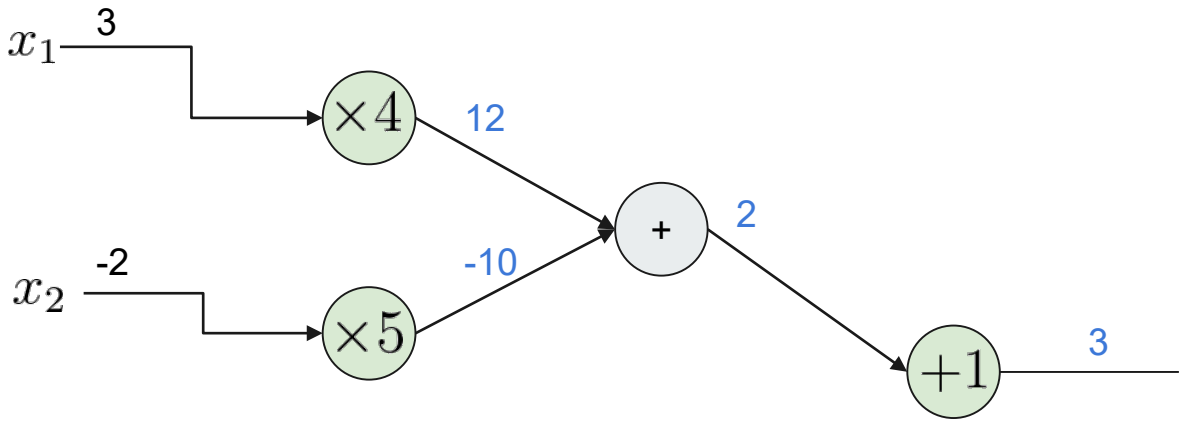
Non-trainable Node:
Can have several inputs, but contains no internal parameters.

The input to a node can be either an input to the entire graph, or it can be the output of another node.

The output of a node can feed into multiple other nodes.

Forward Pass Example

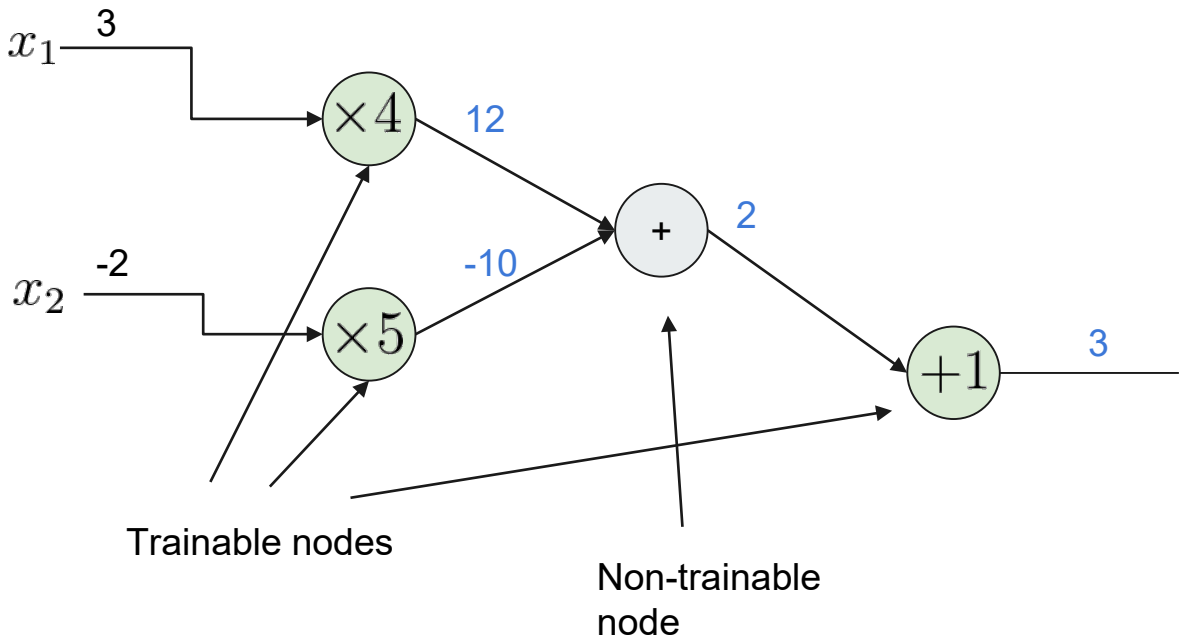
$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



$$\theta_1 = 4$$
$$\theta_2 = 5$$
$$\theta_3 = 1$$
$$x_1 = 3$$
$$x_2 = -2$$

Forward Pass Example

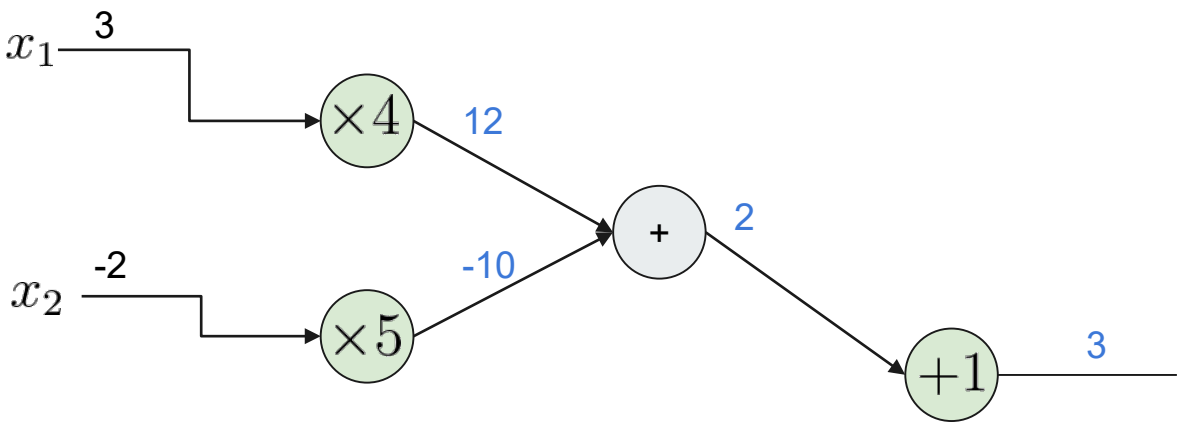
$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



$$\theta_1 = 4$$
$$\theta_2 = 5$$
$$\theta_3 = 1$$
$$x_1 = 3$$
$$x_2 = -2$$

Forward Pass Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



$$\theta_1 = 4$$
$$\theta_2 = 5$$
$$\theta_3 = 1$$
$$x_1 = 3$$
$$x_2 = -2$$

A **neural network** can be viewed as a computation graph, with possibly several outputs.

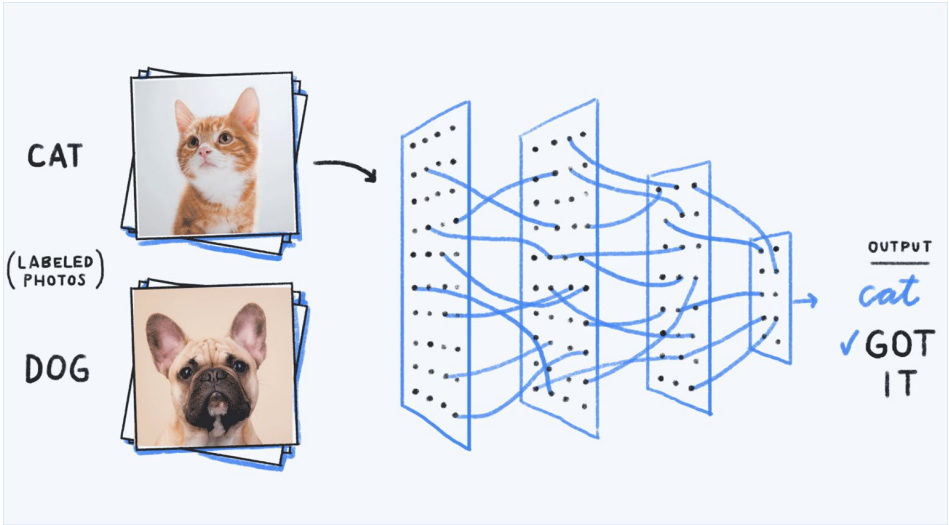
For the moment, let's say we have a desired output.

Training means adjusting the weights so the inputs produce the desired outputs (FOR ALL INPUTS)

- *we do not control the inputs (x_i) but we do control the weights (θ_i)*

Training neural networks

- Target: desired output for a specific input
- Examples of inputs: images and the neural net can be used for classification, ie the target would be the class



Training neural networks

- In the case of neural networks, we often have a “target”, or desired output for an arbitrary input (e.g. an image classification)
- The loss function L is a metric used to measure the “distance” between the output (can be multivalued) with the desired output. Example of a loss function is L2 distance. The loss function outputs a single real number which we try to minimize.

Training neural networks

- In the case of neural networks, we often have a “target”, or desired output for an arbitrary input (e.g. an image classification)
- The loss function L is a metric used to measure the “distance” between the output (can be multivalued) with the desired output. Example of a loss function is L2 distance. The loss function outputs a single real number which we try to minimize.
- To “train” a neural network, we want to adjust the weights of the network such that the network most accurately computes an arbitrary function (output of neural net), which is characterized by training data.

Training neural networks

- In the case of neural networks, we often have a “target”, or desired output for an arbitrary input (e.g. an image classification)
- The loss function L is a metric used to measure the “distance” between the output (can be multivalued) with the desired output. Example of a loss function is L2 distance. The loss function outputs a single real number which we try to minimize.
- To “train” a neural network, we want to adjust the weights of the network such that the network most accurately computes an arbitrary function (output of neural net), which is characterized by training data.
- This can be done by starting with some initial weights, then iteratively updating the weights using the gradient calculated from each training example (gradient descent).

Training neural networks

- In the case of neural networks, we often have a "target", or desired output for an arbitrary input (e.g. an image classification)
- The loss function L is a metric used to measure the "distance" between the output (can be multivalued) with the desired output. Example of a loss function is L2 distance. The loss function outputs a single real number which we try to minimize.
- To "train" a neural network, we want to adjust the weights of the network such that the network most accurately computes an arbitrary function (output of neural net), which is characterized by training data.
- This can be done by starting with some initial weights, then iteratively updating the weights using the gradient calculated from each training example (gradient descent).
- To update the weights, we need to calculate $\frac{\delta L}{\delta \theta_i}$. We can do this for any computation graph by using the partial derivative of each node and the chain rule to "propagate" the gradient backwards through the network. This process of computing the gradients is called backpropagation.

Gradient with respect to output

Training

Compute a gradient for weights (back propagation, explained by example)

Update trainable parameters according to update rule:

$$\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}$$

Our goal is to minimize the function L, by changing only θ, while only knowing the gradient of L

Gradient descent
What is a gradient?
How do we use it?

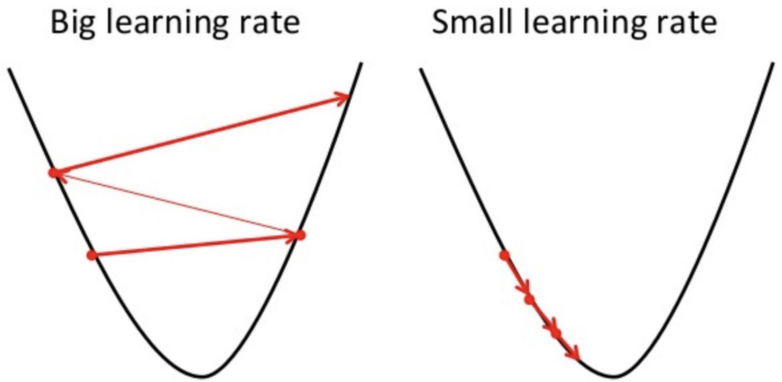
Gradient Descent

$$\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}$$

The gradient tells which direction to update θ, but since the neural network is nonlinear, this direction is only valid for small changes in θ. Therefore, we weight the “gradient update” to θ with a small (generally ~0.001) **learning rate** (α) to ensure that the updates do not overshoot and therefore perform adversarial updates.

Gradient descent is like trying to find your way down a mountain without an overall map--you simply follow which direction the slope points until you reach the bottom.

Note that gradient descent only finds a **local** minimum, not the global minimum. However, in many cases, the local minimum will actually be the global minimum.



Simple Gradient Descent Example

Suppose: $L(\theta, x) = \theta x + \theta^2 x^2$

$$\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}$$

Where $\theta = 3$ and $x = -2$ (L(3,-2) = 30)

The gradient will be $\frac{\partial L}{\partial \theta} = x + 2\theta x^2 = -2 + 2(3)(4) = 22$

If we choose $\alpha = 0.1$, then our update proceeds as $\theta \leftarrow 3 - (0.1)(22) = 0.8$

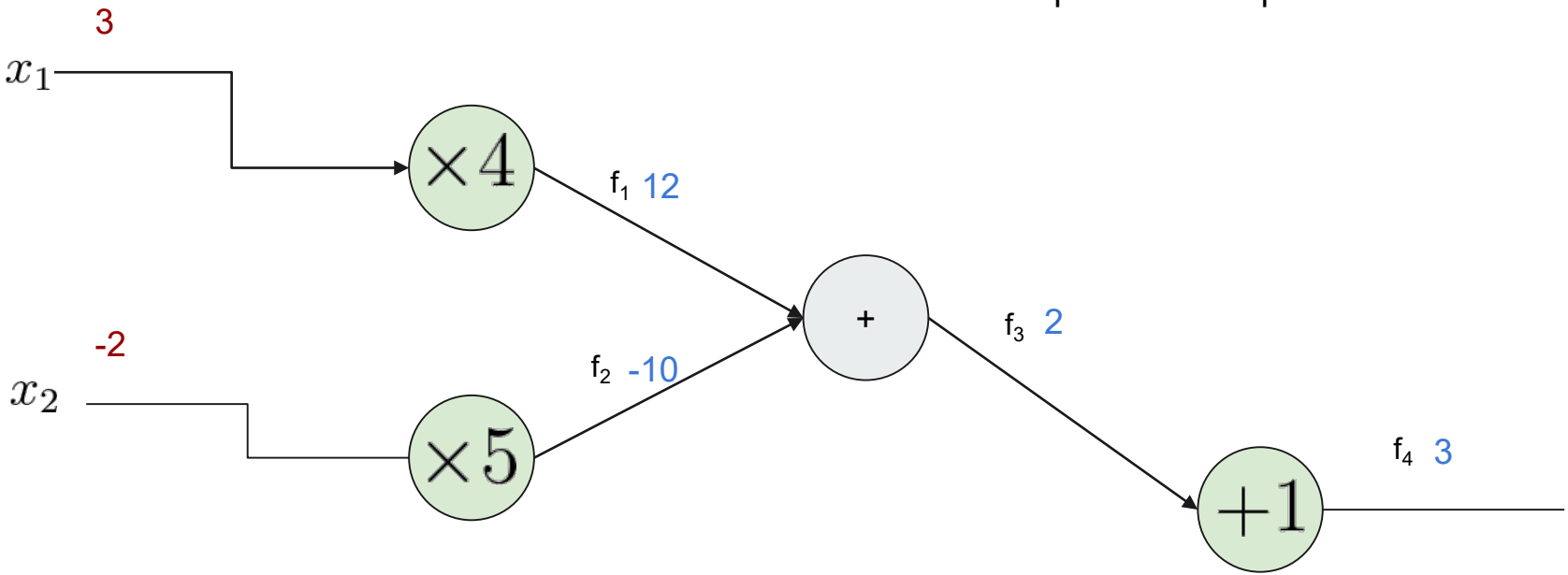
The new theta value should decrease the output of L. Checking (L(0.8, -2) = 0.96) shows that it does.

We can repeat this process until the minimum (θ=0.25) is reached.

Backpropagation Example

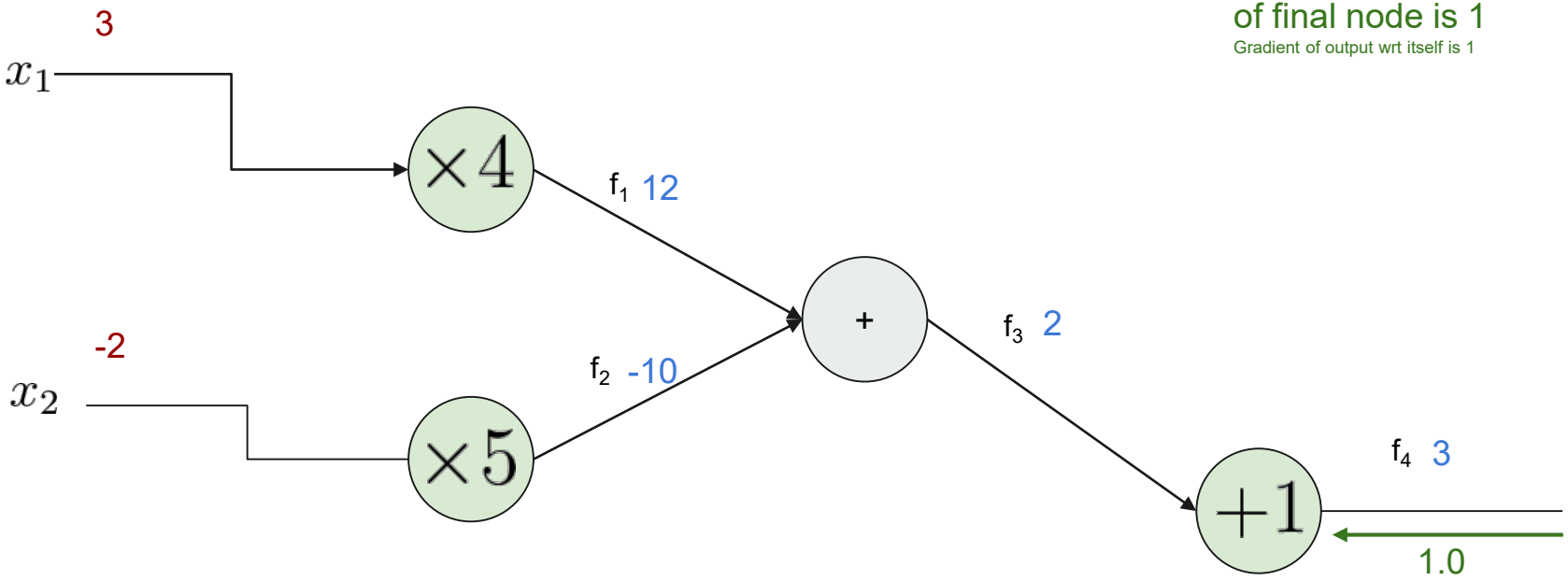
$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

We will compute the gradients of the weights of this graph with respect to its output.



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

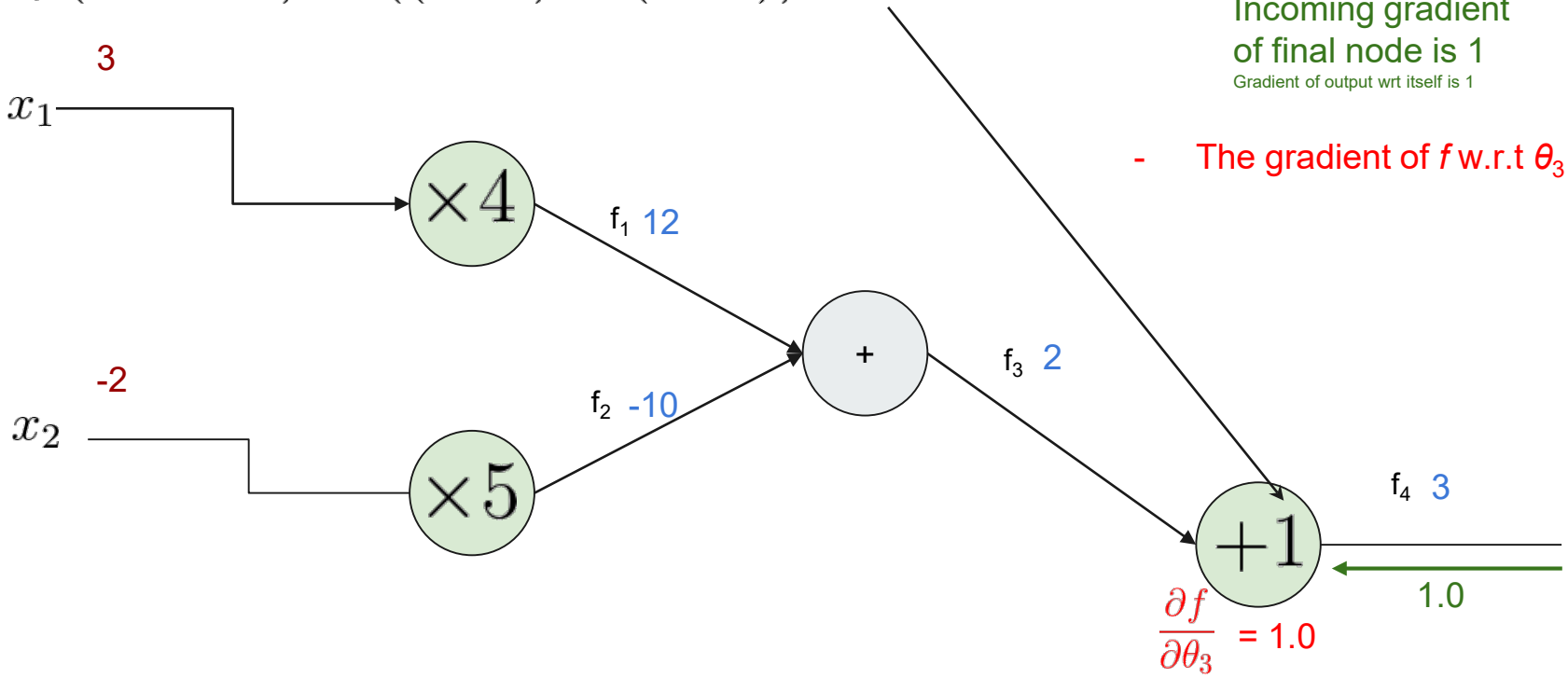


Incoming gradient of final node is 1

Gradient of output wrt itself is 1

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



Incoming gradient of final node is 1

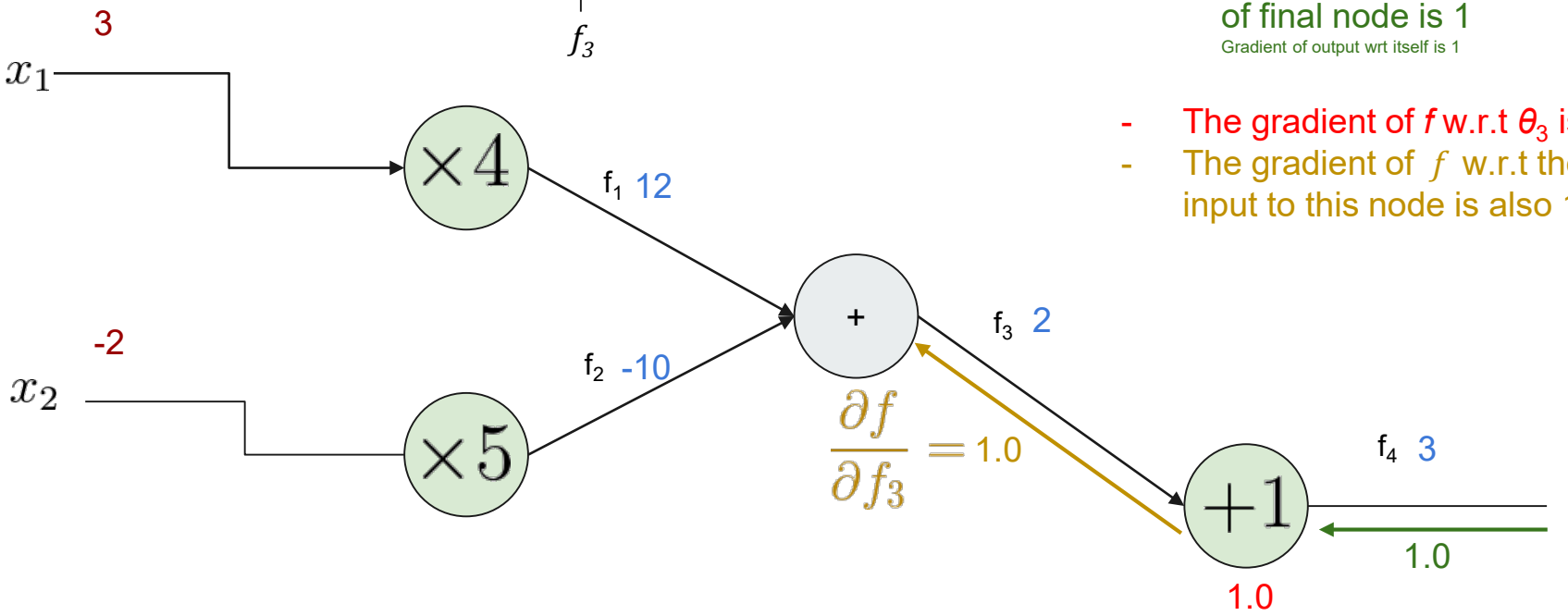
Gradient of output wrt itself is 1

- The gradient of *f* w.r.t θ_3 is 1

Backpropagation Example

$$f(x_1, x_2, \theta) = (((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

f_3



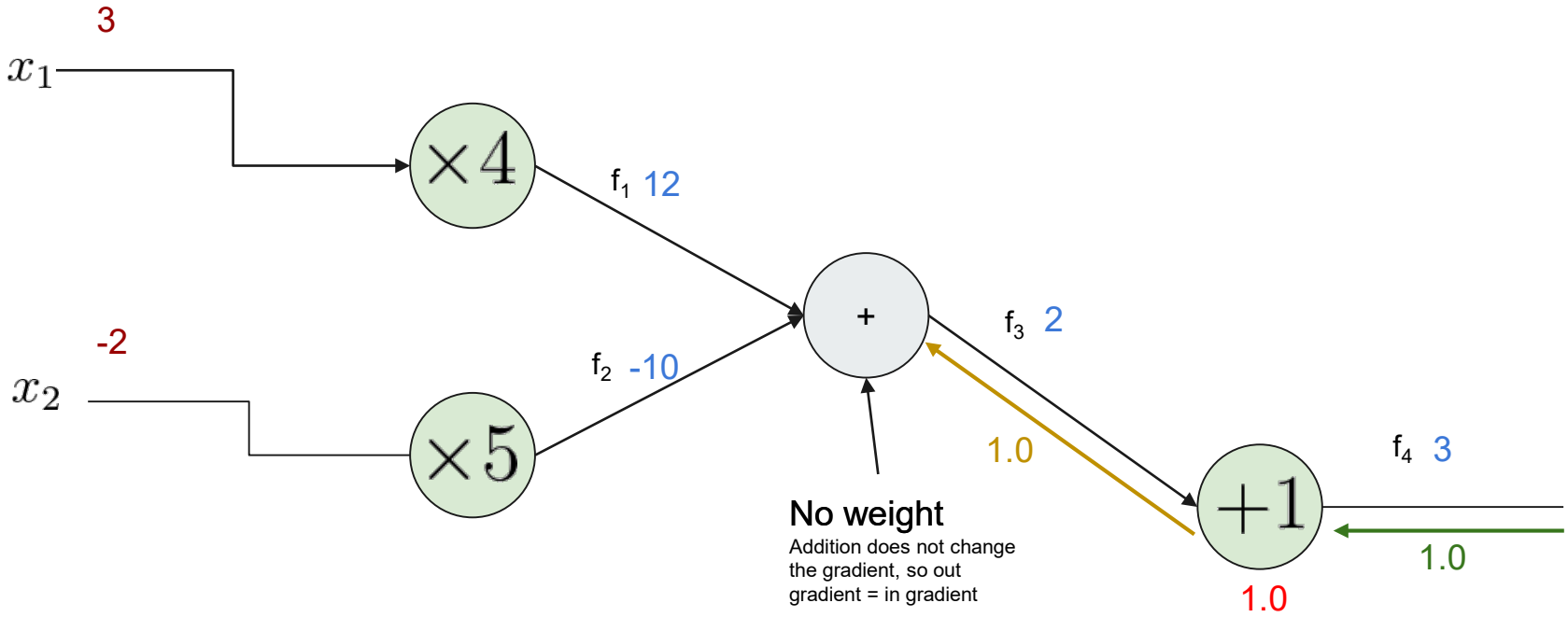
Incoming gradient of final node is 1

Gradient of output wrt itself is 1

- The gradient of *f* w.r.t θ_3 is 1
- The gradient of f w.r.t the input to this node is also 1.

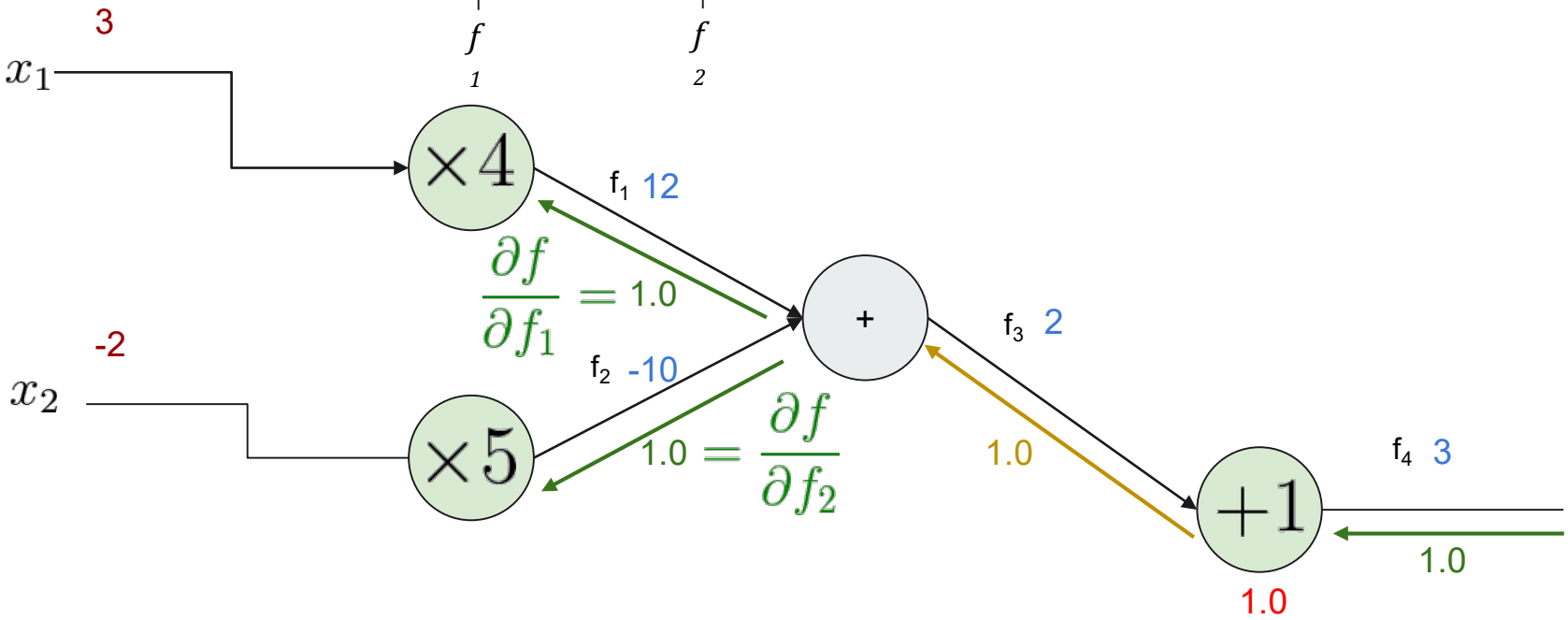
Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



Backpropagation Example

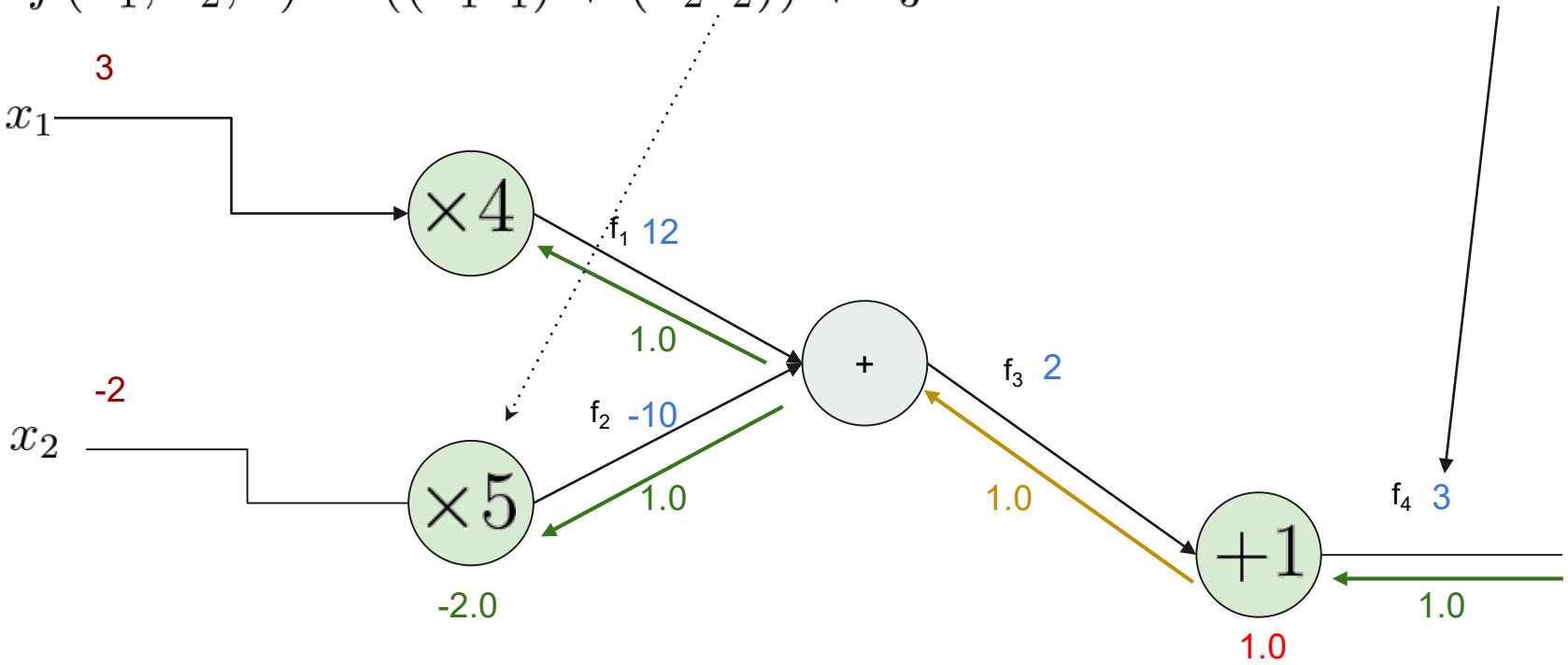
$$f(x_1, x_2, \theta) = (\underbrace{(x_1\theta_1)}_{f_1} + \underbrace{(x_2\theta_2)}_{f_2}) + \theta_3$$



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

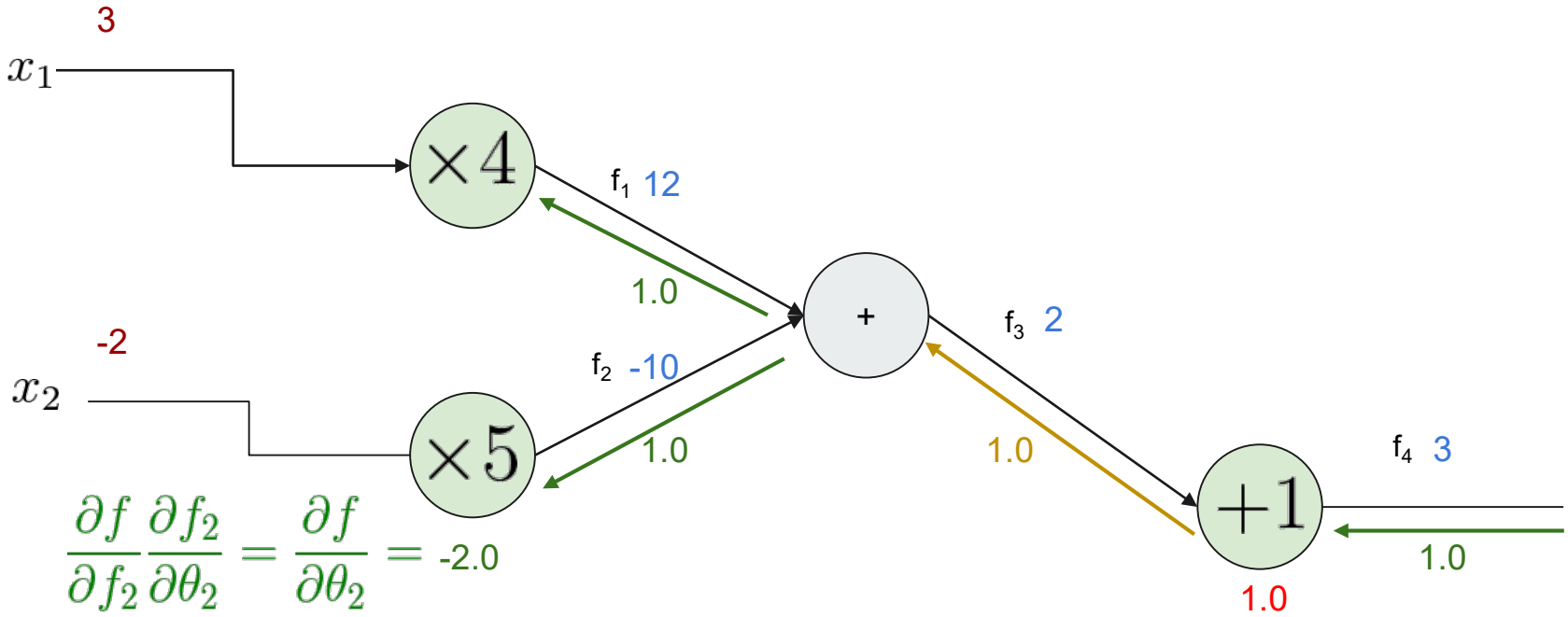
Now we calculate the gradient of f w.r.t θ_2



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

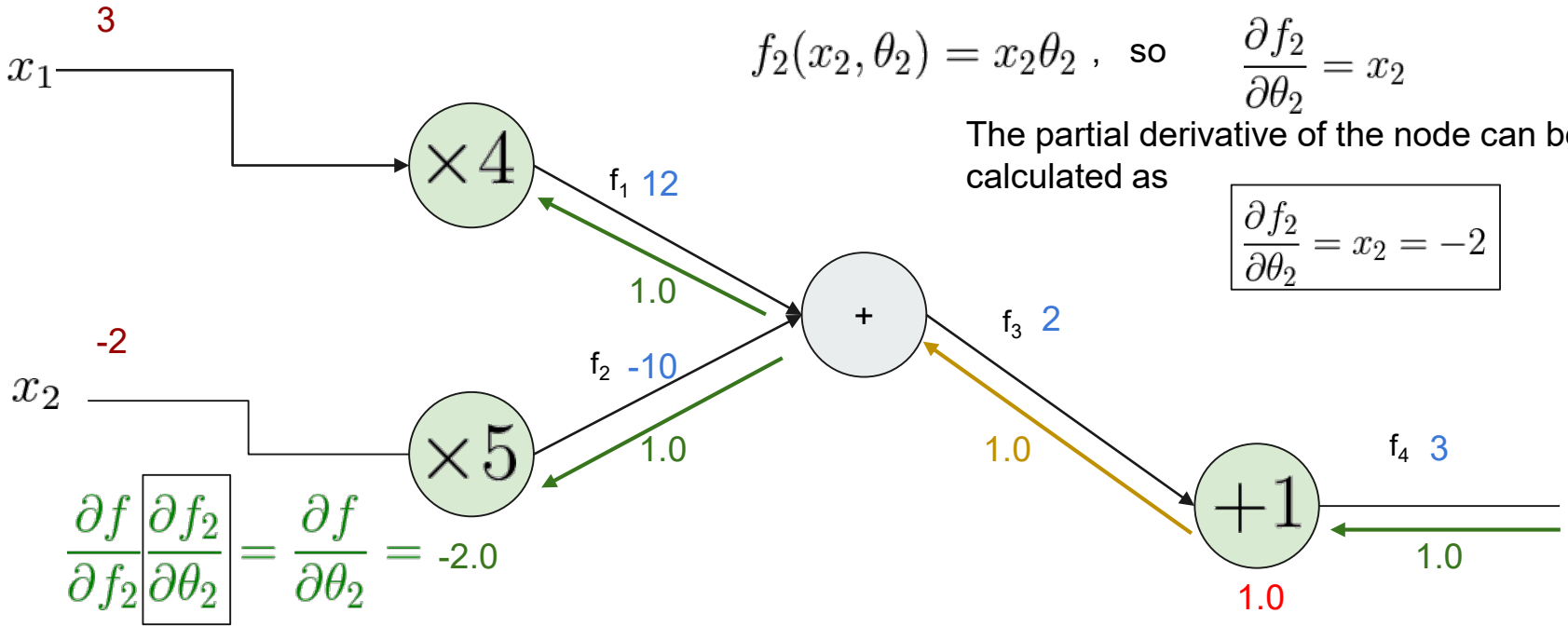
Now we calculate the gradient of y w.r.t θ_2



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

Now we calculate the gradient of y w.r.t θ_2



$$f_2(x_2, \theta_2) = x_2\theta_2 \text{ , so } \frac{\partial f_2}{\partial \theta_2} = x_2$$

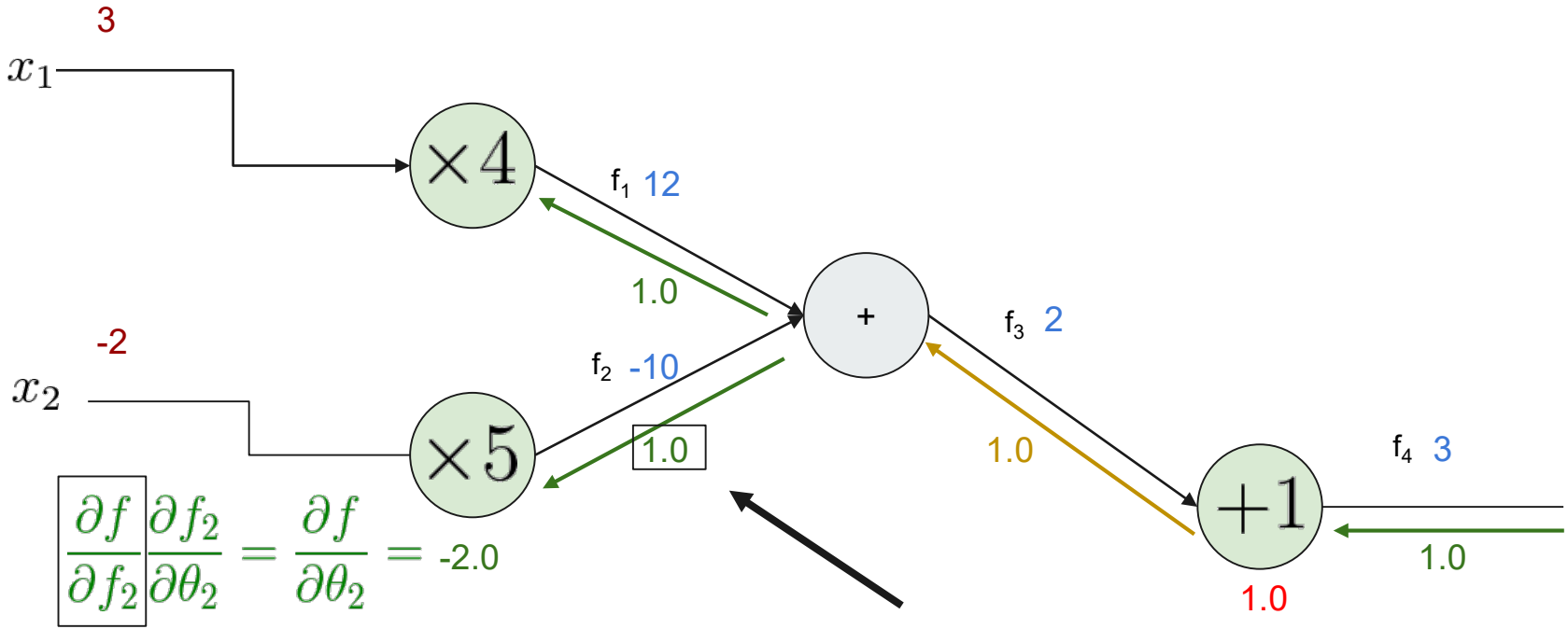
The partial derivative of the node can be calculated as

$$\boxed{\frac{\partial f_2}{\partial \theta_2} = x_2 = -2}$$

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

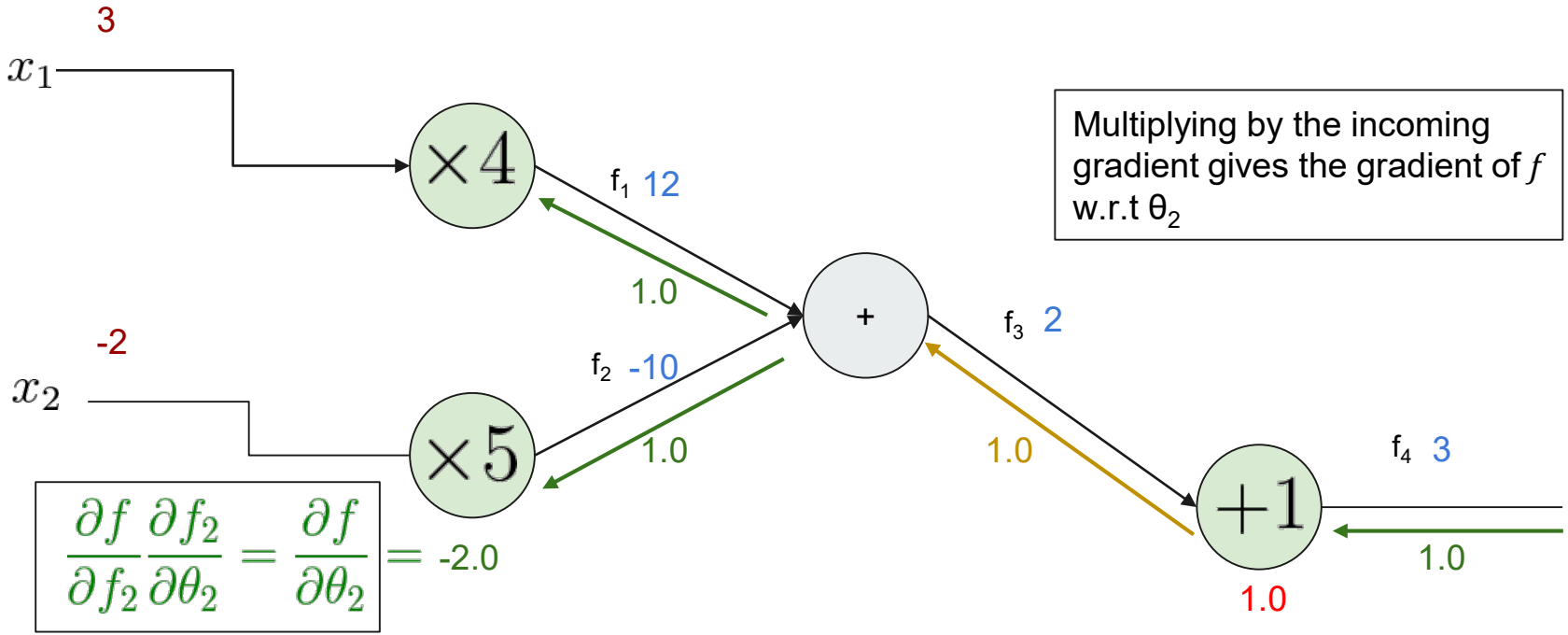
Now we calculate the gradient of y w.r.t θ_2



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

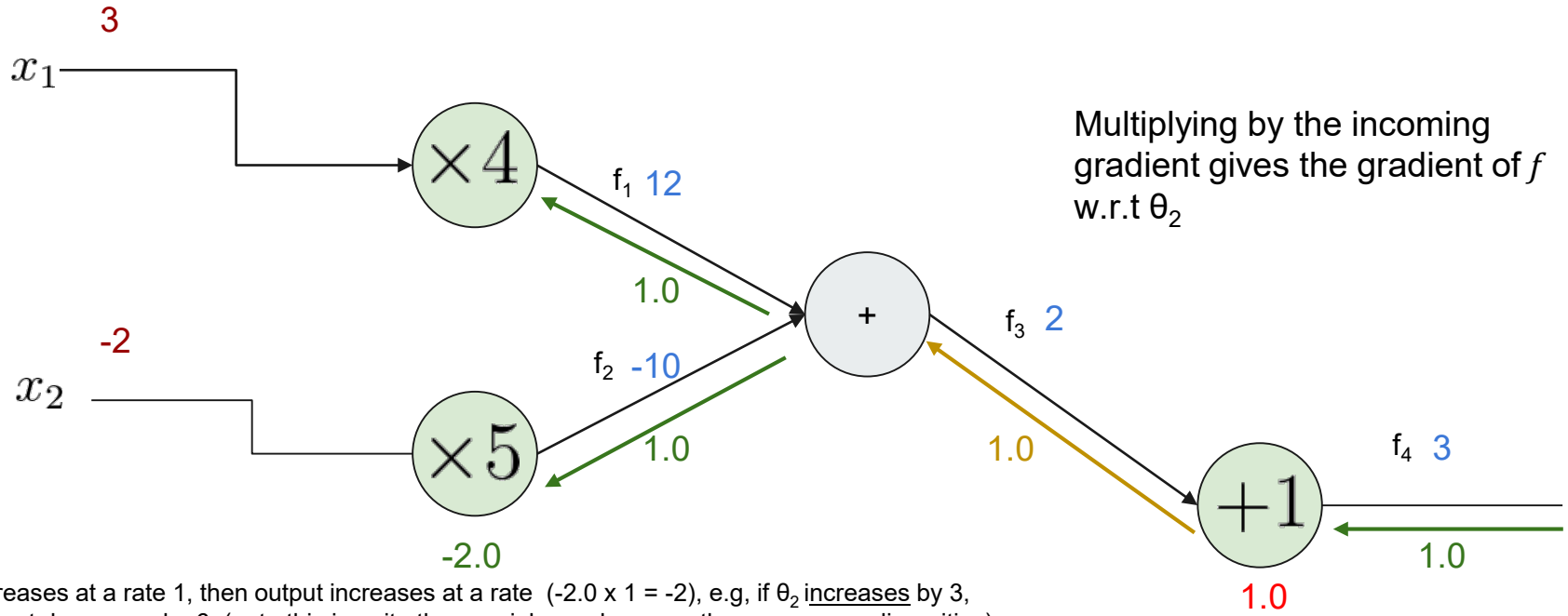
Now we calculate the gradient of y w.r.t θ_2



Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

Now we calculate the gradient of y w.r.t θ_2



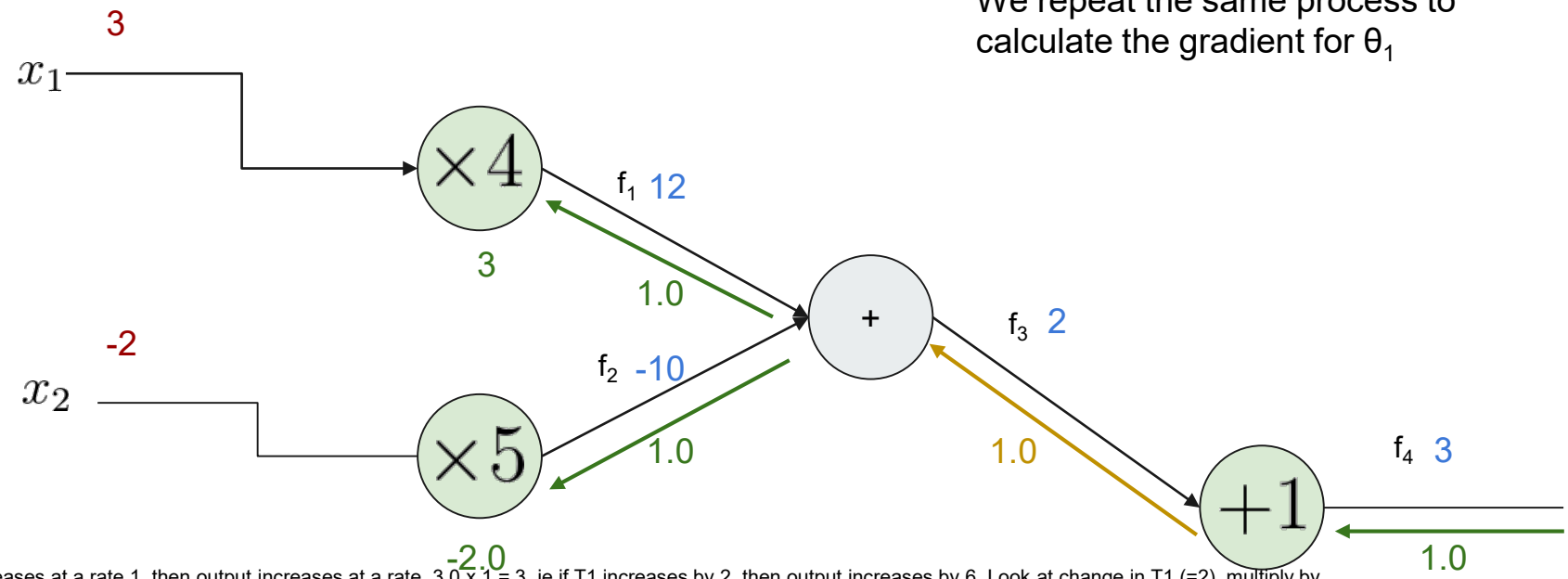
Multiplying by the incoming gradient gives the gradient of f w.r.t θ_2

If θ_2 increases at a rate 1, then output increases at a rate (-2.0 x 1 = -2), e.g, if θ_2 increases by 3, then output decreases by 6. (note this is quite the special case because there are no nonlinearities)

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

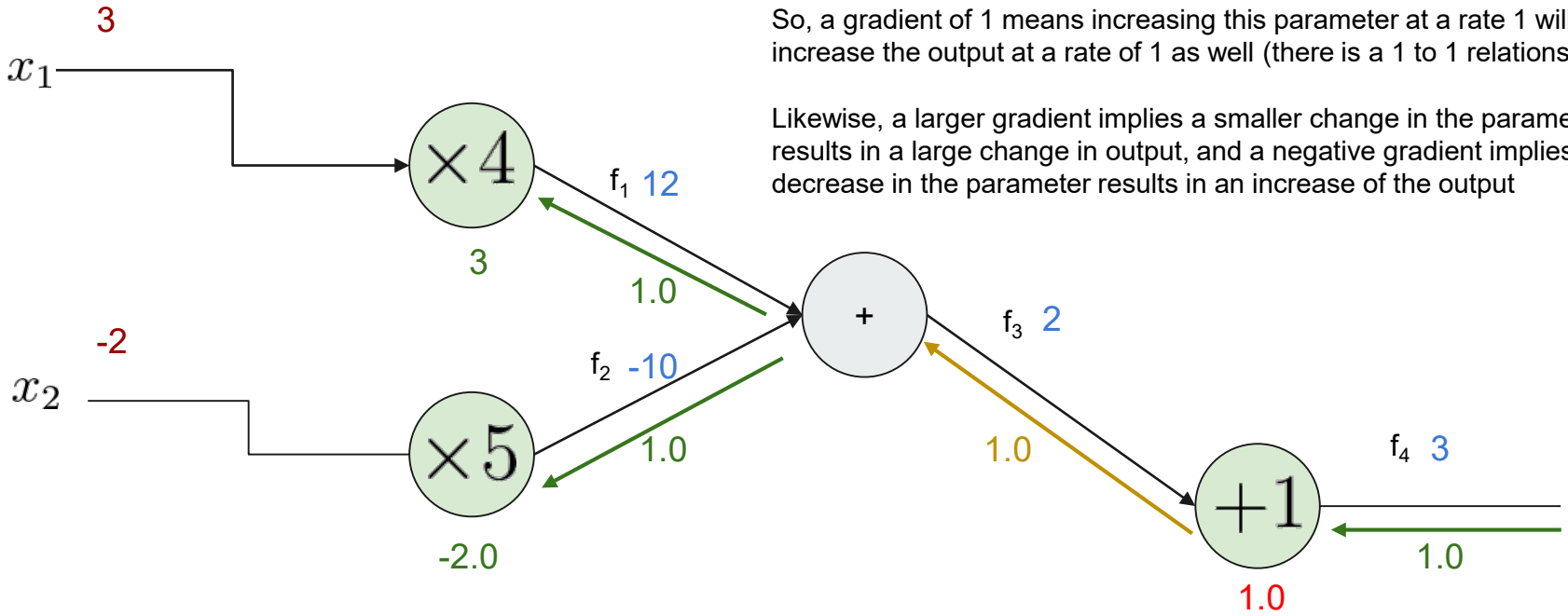
We repeat the same process to calculate the gradient for θ_1



If θ_1 increases at a rate 1, then output increases at a rate 3.0 x 1 = 3, ie if T1 increases by 2, then output increases by 6. Look at change in T1 (=2), multiply by gradiet (3) = 6. Multiply by 1.0. OR cascade down (input is 3, the weight is 6, 3x6 = 18. 18 - 10 = 8. 8 + 1 = 9. . (note this is quite the special case because there are no nonlinearities)

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



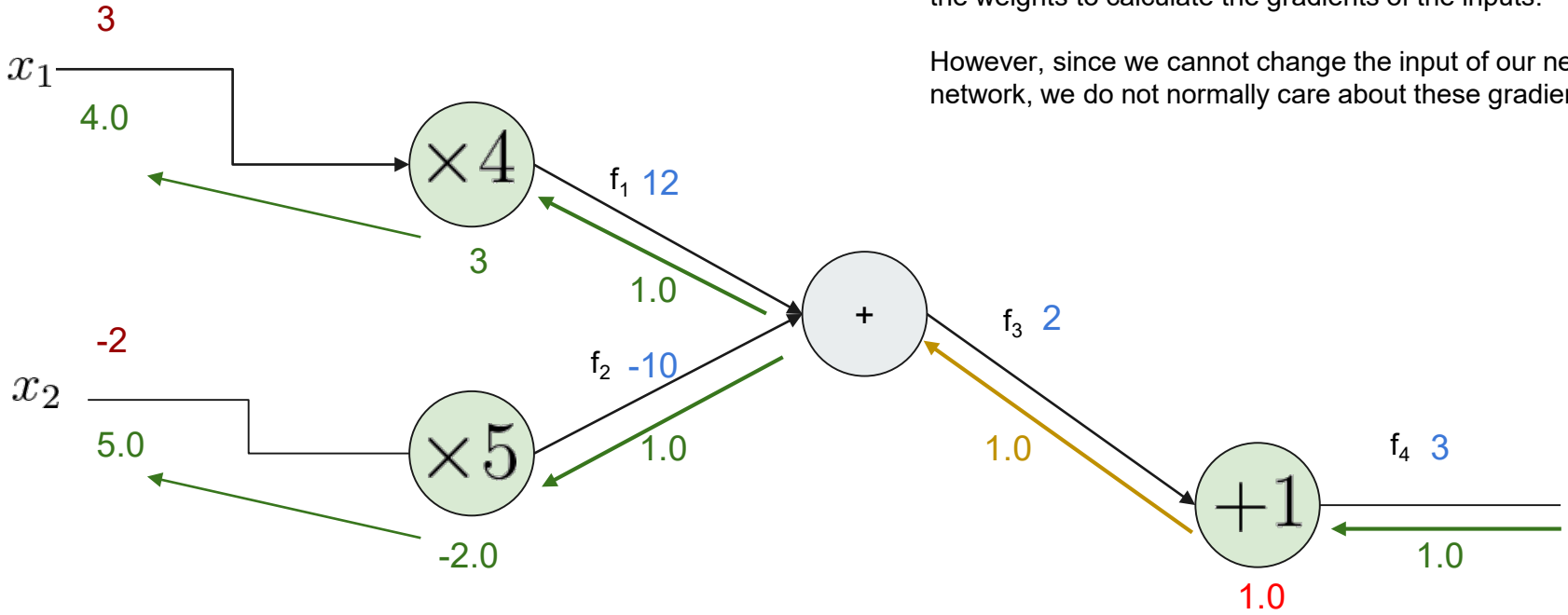
Intuition behind the gradient:
It gives us a number which describes how a change to that number affects the output

So, a gradient of 1 means increasing this parameter at a rate 1 will increase the output at a rate of 1 as well (there is a 1 to 1 relationship)

Likewise, a larger gradient implies a smaller change in the parameter results in a large change in output, and a negative gradient implies that a decrease in the parameter results in an increase of the output

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$



We can apply the same process for calculating the gradient of the weights to calculate the gradients of the inputs.

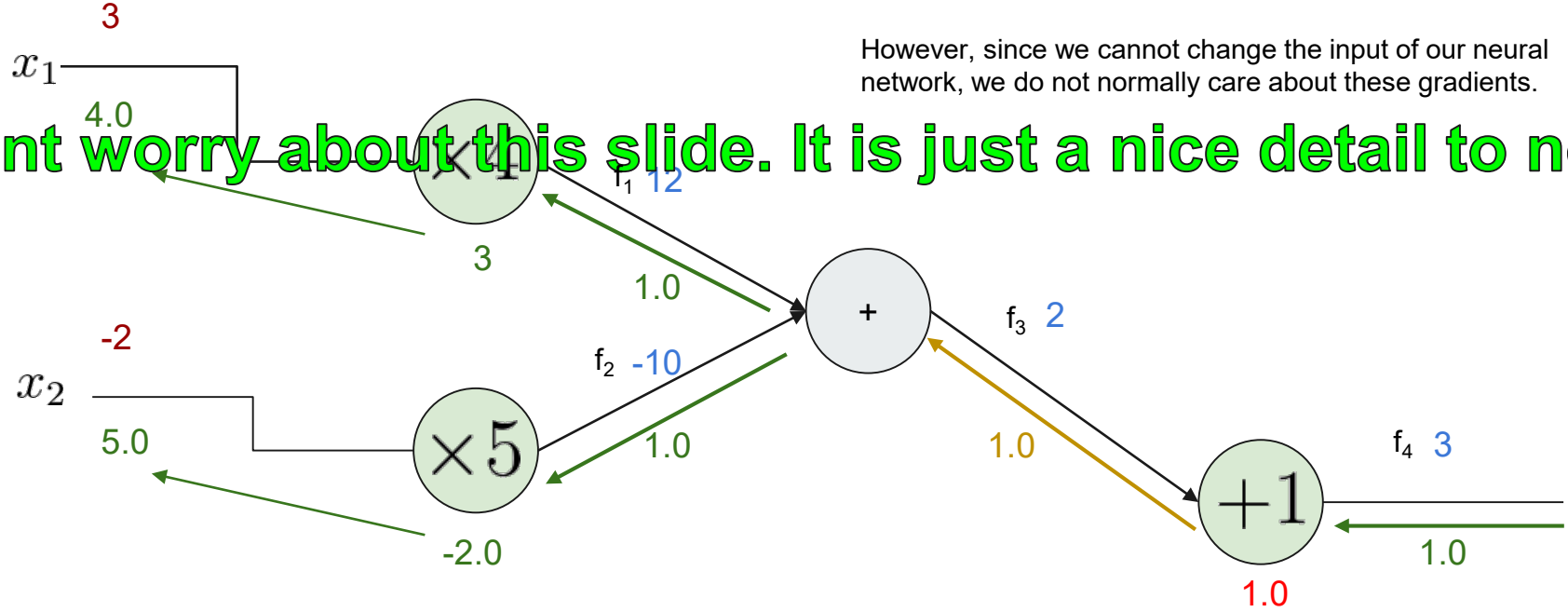
However, since we cannot change the input of our neural network, we do not normally care about these gradients.

Backpropagation Example

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

We can apply the same process for calculating the gradient of the weights to calculate the gradients of the inputs.

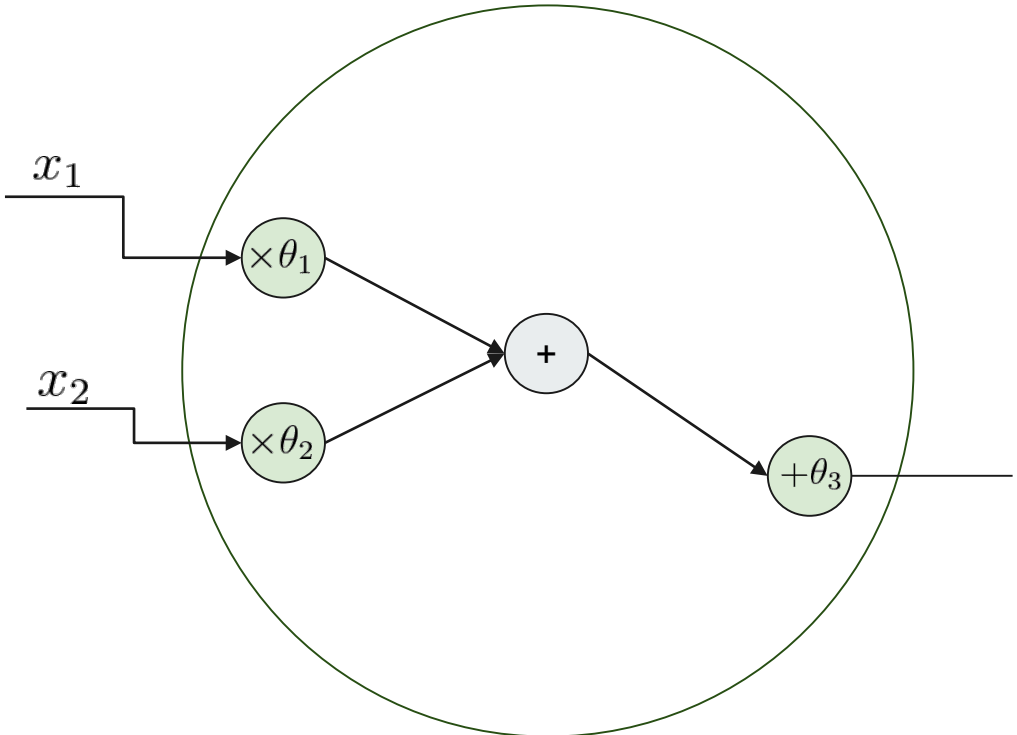
However, since we cannot change the input of our neural network, we do not normally care about these gradients.



Dont worry about this slide. It is just a nice detail to note

More complicated functions

$$f(x_1, x_2, \theta) = ((x_1\theta_1) + (x_2\theta_2)) + \theta_3$$

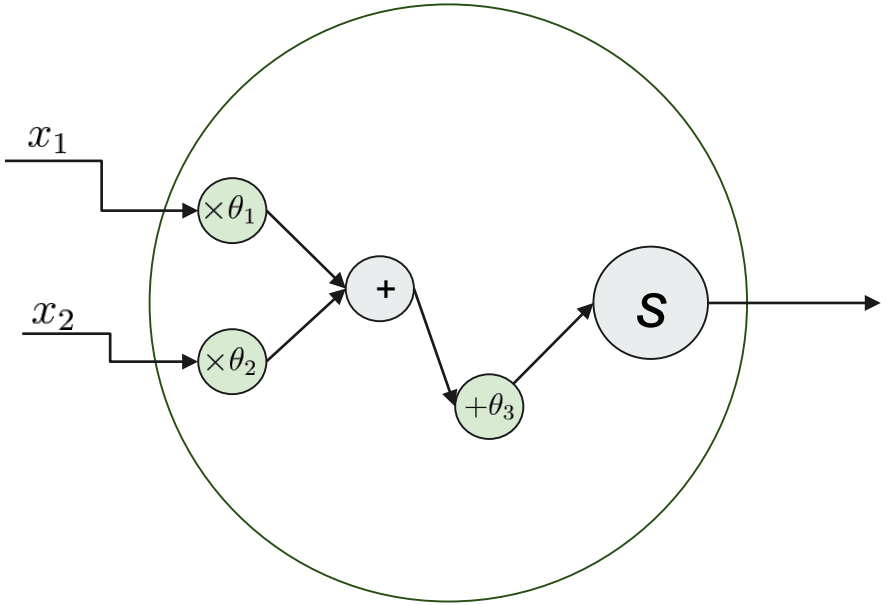


We can group a collection of nodes in the computation graph together to form structures which perform more complicated tasks.

A **neuron** in a neural network is one of these structures.

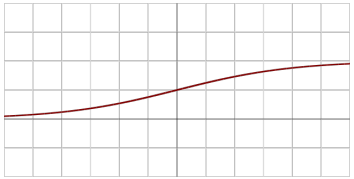
Neural network node

$$f(x_1, x_2, \theta) = \frac{1}{1 + e^{-(x_1\theta_1 + x_2\theta_2 + \theta_3)}}$$



A node in a neural network uses nearly the same computation graph as before, but the output is passed through a nonlinear activation function, which can be either a trainable or non-trainable node.

In this case, the activation function is a sigmoid function (non-trainable):



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

This structure of nodes, when grouped together, forms a single neuron in the network.

Generally, a neuron is a linear combination of many inputs composed with its nonlinear activation function.

Activation functions

There are many choices of activation functions

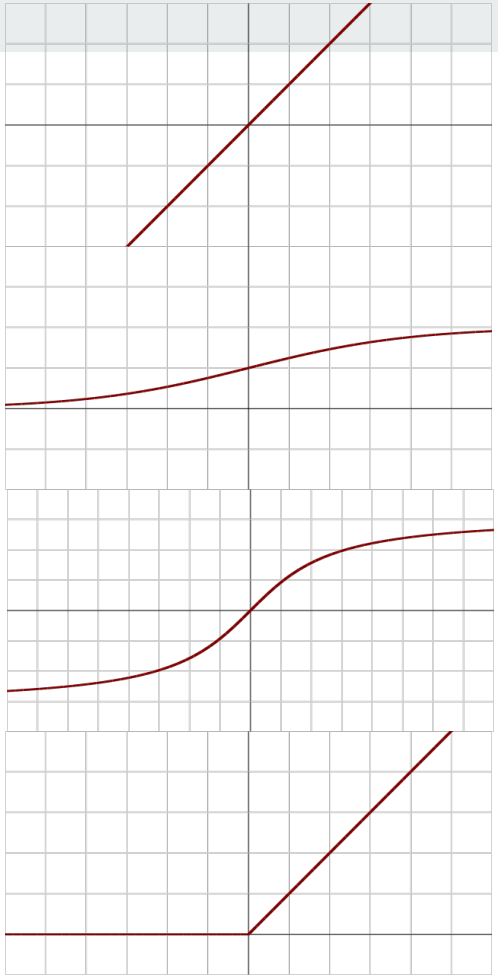
- See wikipedia article on activation functions

Common activation functions include

- identity $f(x) = x$
- sigmoid $f(x) = 1 / (1+e^{-x})$
- inverse tangent $f(x) = \tan^{-1}(x)$
- rectified linear unit $f(x) = \max(0, x)$
- and more...

The choice of activation function is very important! For hidden layers, the activation function **must** be nonlinear. For all activation functions, they ought to be differentiable.

The most common practice is to use ReLU on hidden layers, and identity on your output layer.



Backpropagation in neural networks

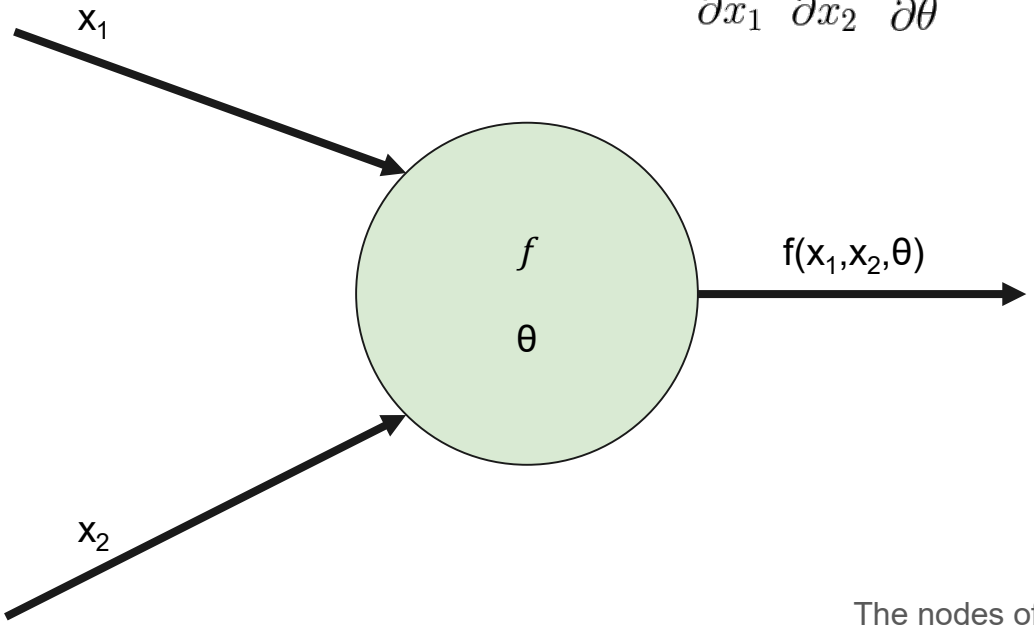
- In the case of neural networks, we often have a “target”, or desired output for an arbitrary input (e.g. an image classification)
- **The loss function L is a metric used to measure the “distance” between the output (can be multivalued) with the desired output. Example of a loss function is L2 distance. The loss function outputs a single real number which we try to minimize.**
- To “train” a neural network, we want to adjust the weights of the network such that the network most accurately computes an arbitrary function (output of neural net), which is characterized by training data.
- This can be done by starting with some initial weights, then iteratively updating the weights using the gradient calculated from each training example (gradient descent).

- To update the weights, we need to calculate $\frac{\delta L}{\delta \theta_i}$. We can do this for any computation graph by using the partial derivative of each node and the chain rule to “propagate” the gradient backwards through the network.

Backpropagation at a single node

Goal: Calculate

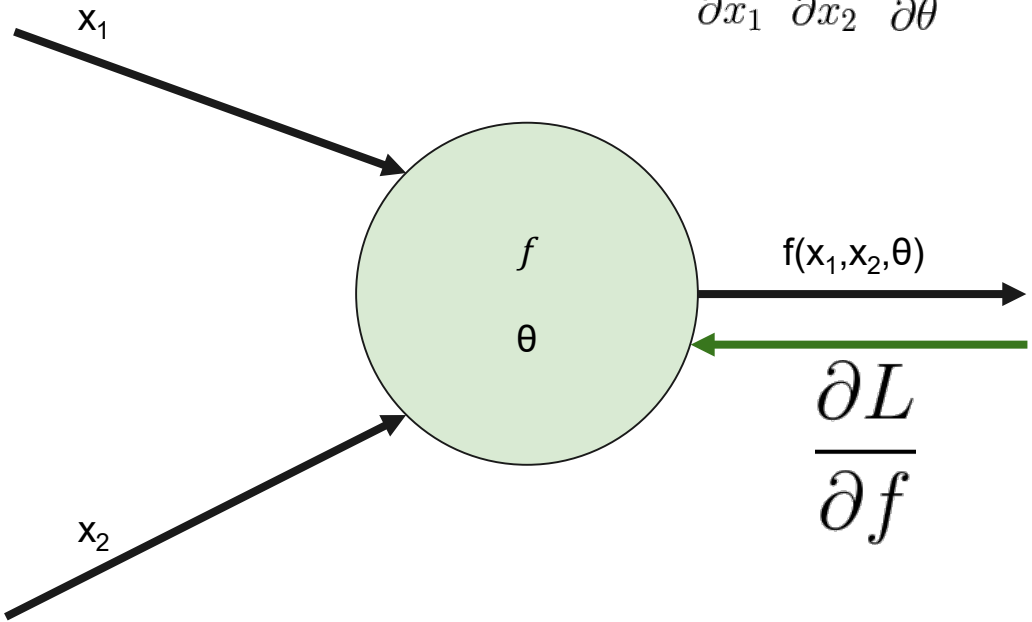
$$\frac{\partial L}{\partial x_1} \quad \frac{\partial L}{\partial x_2} \quad \frac{\partial L}{\partial \theta}$$



The nodes of a neural network must be differentiable!

Backpropagation at a single node

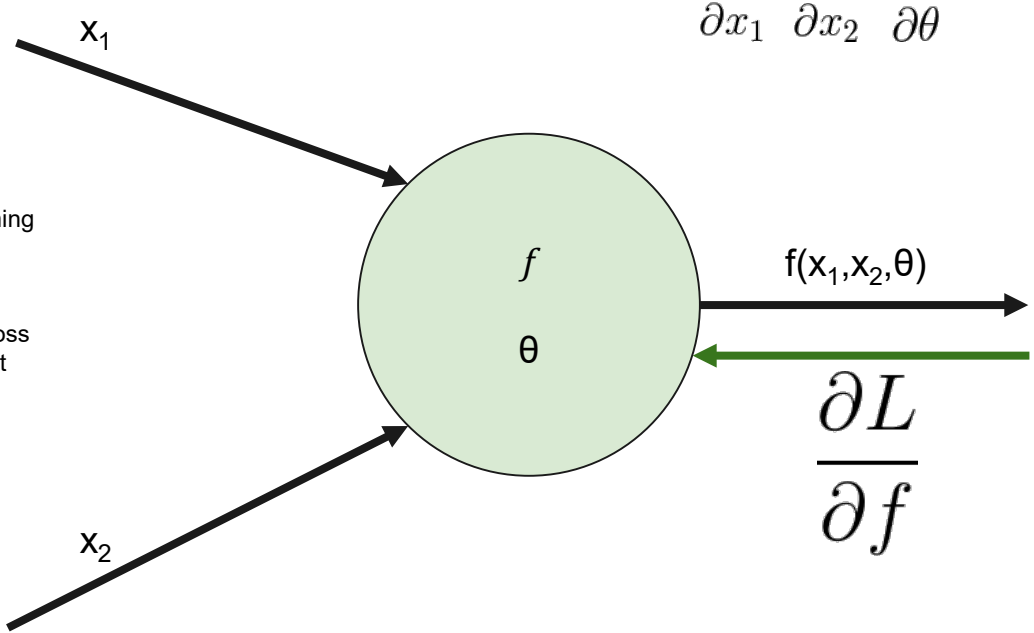
Goal: Calculate $\frac{\partial L}{\partial x_1}$ $\frac{\partial L}{\partial x_2}$ $\frac{\partial L}{\partial \theta}$



Backpropagation at a single node

Goal: Calculate

$$\frac{\partial L}{\partial x_1} \quad \frac{\partial L}{\partial x_2} \quad \frac{\partial L}{\partial \theta}$$



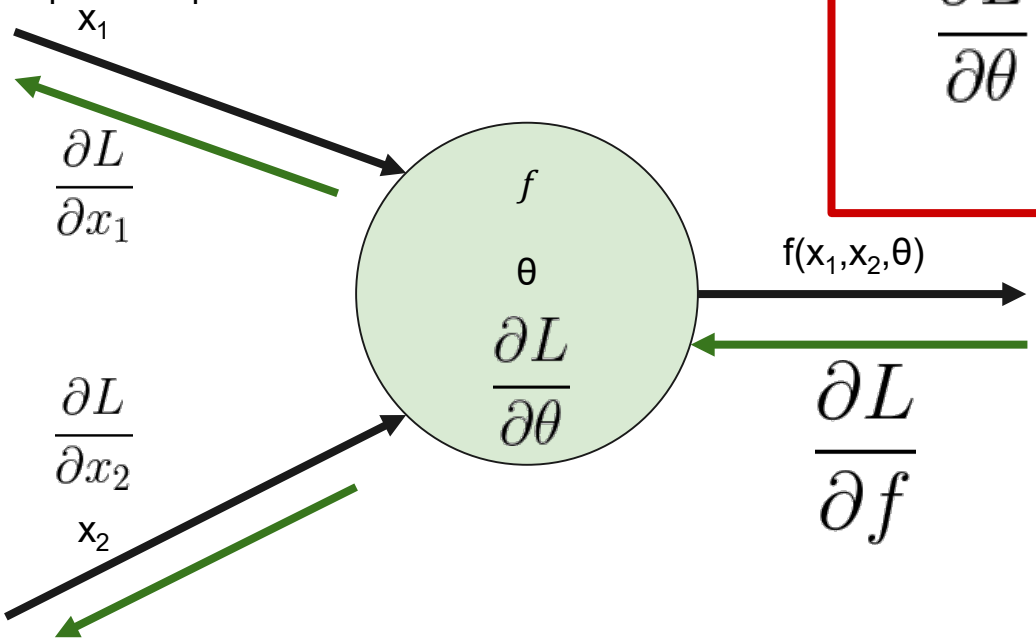
The green line represents the incoming gradient w.r.t the loss function.

If this node is the final node in the graph (i.e. this node computes the loss function), then the incoming gradient will be 1

Backpropagation at a single node

First, we calculate the gradient for the trainable parameters by using the chain rule, multiplying the incoming gradient with the partial w.r.t theta. We make sure to, save these for the update step later.

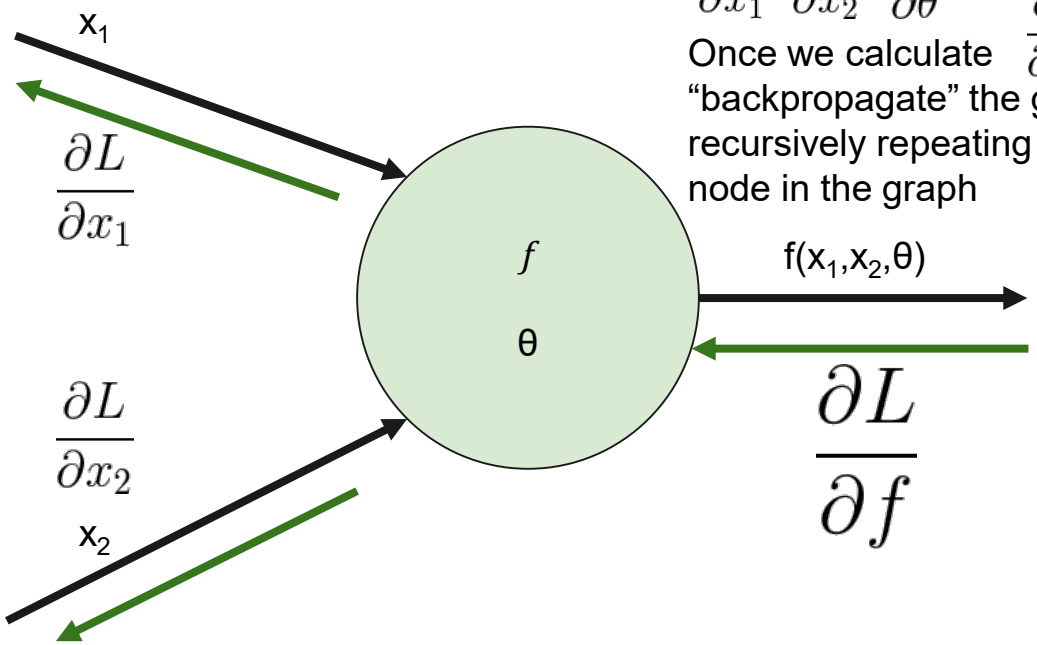
$$\frac{\partial L}{\partial \theta} = \frac{\partial f}{\partial \theta}\frac{\partial L}{\partial f}$$



Backpropagation at a single node

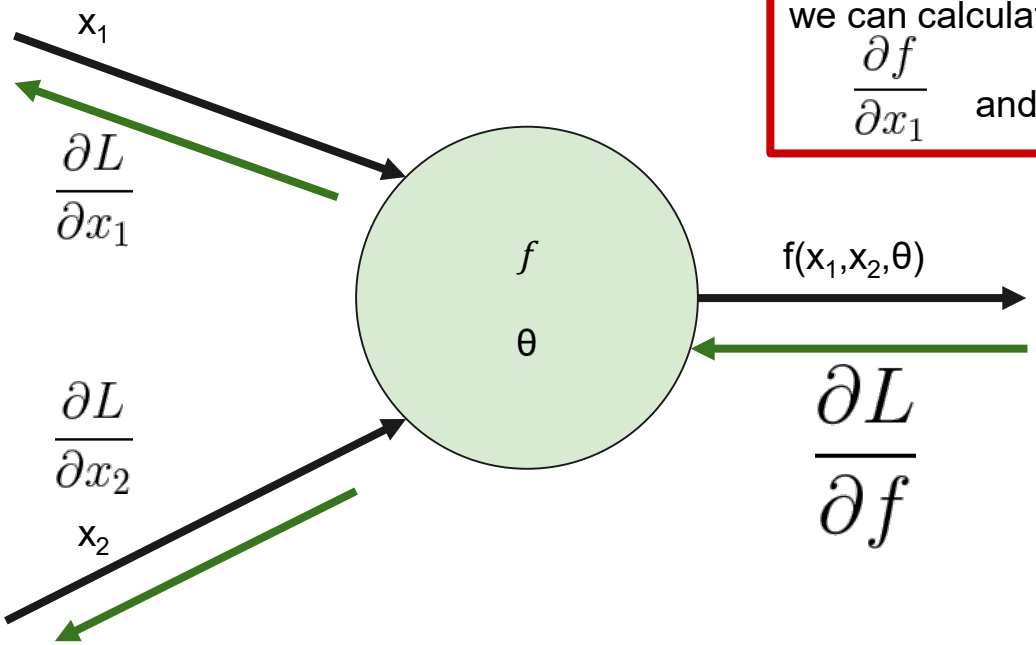
Goal: Calculate $\frac{\partial L}{\partial x_1}$ $\frac{\partial L}{\partial x_2}$ $\frac{\partial L}{\partial \theta}$

Once we calculate $\frac{\partial L}{\partial x_1}$ and $\frac{\partial L}{\partial x_2}$ we can "backpropagate" the gradients to earlier nodes, recursively repeating this process for each node in the graph



Backpropagation at a single node

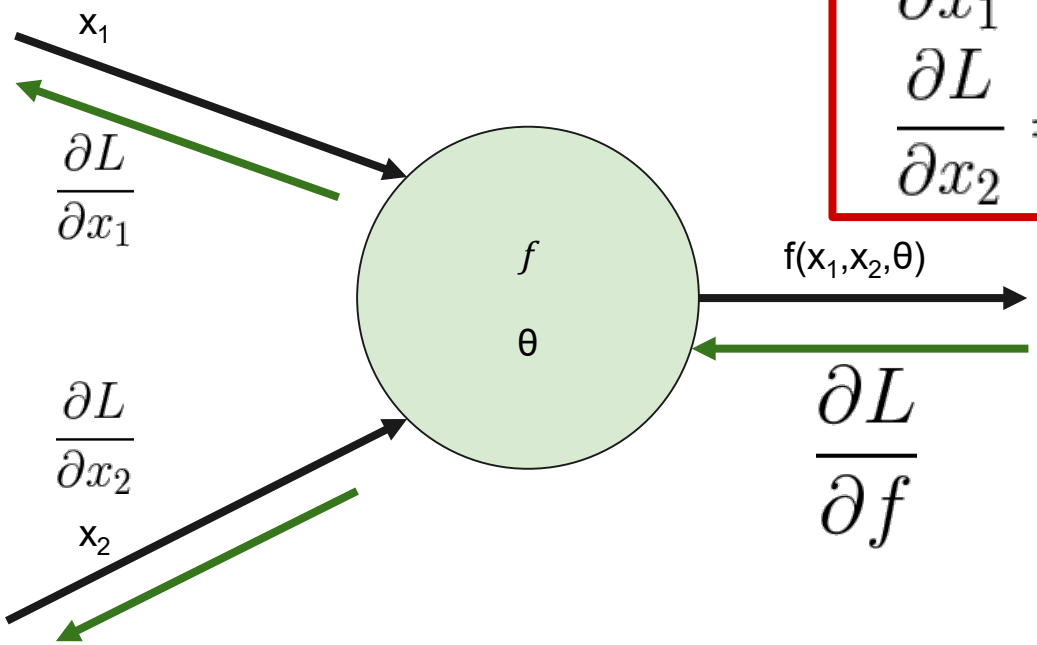
Since $f(x_1, x_2, \theta)$ is differentiable, we can calculate $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$



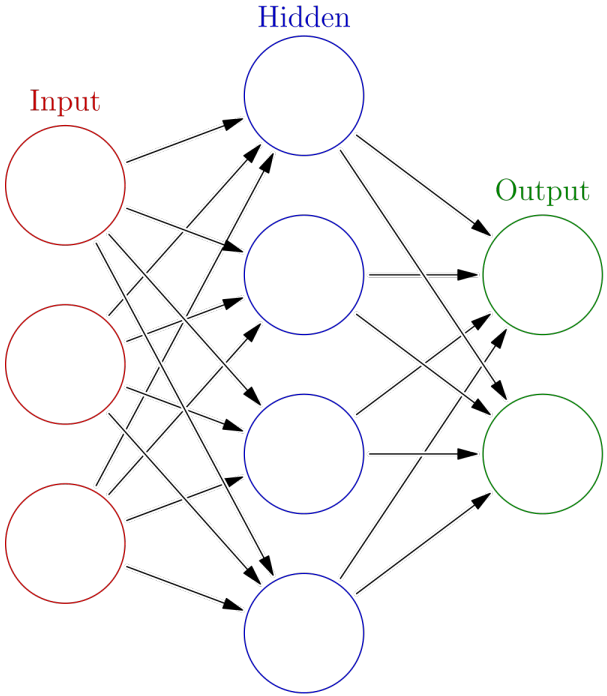
Backpropagation at a single node

Using the chain rule:

$$\frac{\partial L}{\partial x_1} = \frac{\partial f}{\partial x_1}\frac{\partial L}{\partial f}$$
$$\frac{\partial L}{\partial x_2} = \frac{\partial f}{\partial x_2}\frac{\partial L}{\partial f}$$



Structure of a neural network



A feed-forward artificial neural network is a computation graph which consists of neurons, each computing a linear combination of the previous layer composed with an activation function.

The final layer is considered the output layer.

If there are any layers of the network which are not the input or output layers, those are considered hidden layers, and the network is a “deep” neural network.

Note that the input layer does not have any weights associated with it.

Image from wikipedia: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg

Training a neural network

The purpose of the neural network is so we can create arbitrary functions, which capture a relationship described in training data. (e.g. a point on a topographical map vs the elevation of that point)

The loss function quantifies the performance of the neural network, by “scoring” the output of the neural network compared to some desired output (e.g. the absolute difference between the elevation predicted by the network, and true elevation of the point)

Backpropagation gives us a gradient which tells how changing our trainable parameters will reduce the loss.

If we minimize the loss function over all of the training data, the neural network becomes the best possible approximation of the function we are trying to train.

Training a neural network

Backpropagation gives us a gradient which tells how changing our trainable parameters will reduce the loss.

In other words, for each trainable parameter θ, and each training example *i*, we can calculate $\frac{\partial L_i}{\partial \theta}$, where L_i is the loss function for the network on input *i*.

Using this gradient, we can perform gradient descent to iteratively improve the overall performance of the neural network until the loss function is minimized, using the following update rule, applied to all parameters at each step:

$$\theta \leftarrow \theta - \alpha \frac{\partial L_i}{\partial \theta}$$

Really, this is stochastic gradient descent

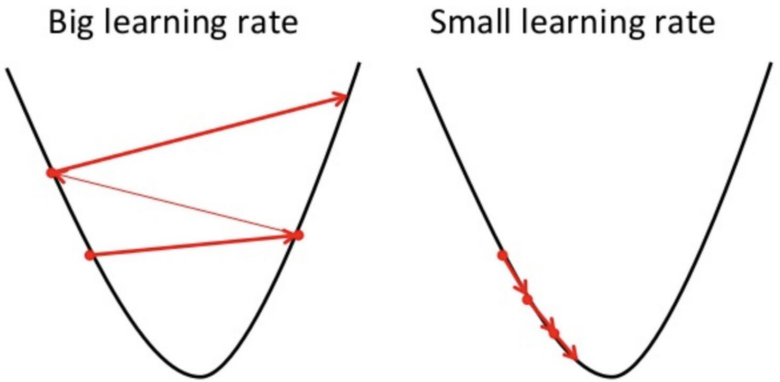
Training a neural network

$$\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}$$

The gradient tells which direction to update θ, but since the neural network is nonlinear, this direction is only valid for small changes in θ. Therefore, we weight the “gradient update” to θ with a small (generally ~0.001) **learning rate** (α) to ensure that the updates do not overshoot and therefore perform adversarial updates.

Gradient descent is like trying to find your way down a mountain without an overall map--you simply follow which direction the slope points until you reach the bottom.

Note that gradient descent only finds a **local** minimum, not the global minimum. However, in many cases, the local minimum will actually be the global minimum.



What next

- There have been many modifications, tweaks, etc. to the general ideas presented here, and other material may appear slightly differently, even if the core ideas are the same
- While the general framework for artificial neural networks have been described, this is not a hard-set rule which must be followed. Many of the most successful and most popular advances in machine/deep learning start from these ideas, and try more radical modifications to achieve better results.
- For more resources/to learn more, see wikipedia, and Google (there are many resources which go into far greater depth online).
- Additionally, take 10-601, 10-701, or other MLD courses