Today's agenda

Administrative stuff

- A1 due tomorrow
- A2 released (will add some more EC in a day or two)

Topics

- Numerical stability
- Gradient descent basics
- Backpropagation basics

Overflow and underflow pop up frequently when training NNs. Good to be aware of these I will save you headaches down the road.

Assignment 2 Walkthrough

Questions from email

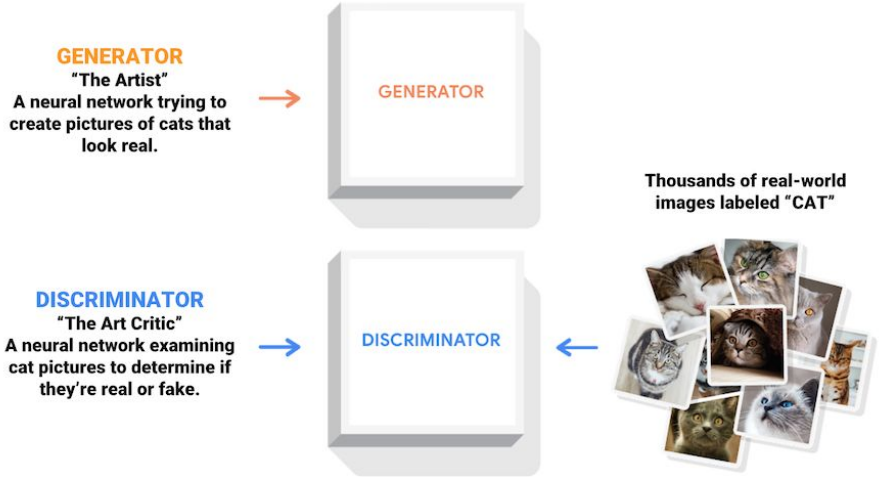
Most of my experience is with sklearn. How do we classify CSVs with DL?

- Wrote a quick **example** for you.
- You may find it hard to beat a Random Forest baseline (w/o a ton of data) on many datasets.
- We have a class on these topics in a few weeks.

Quick demo with Facets.

Updated tutorial for GANs

- Generating Handwritten Digits with DCGAN



Numerical stability

Our friends overflow and underflow

Underflow

- **Numbers near zero are rounded to zero**. Problem for functions that behave differently when their argument is zero (instead of a small positive value).

- Example: when computing loss, we take the log of the Softmax output.

  - Softmax is supposed to be $0 < x < 1$.

  - log(0) is undefined.

Underflow example

```
import numpy as np
np.exp(-1), np.exp(-10), np.exp(-100), np.exp(-1000)
0.36, 4.53e-05, 3.72e-44, 0.0
```

Not actually zero! This is an error due to the floating point representation.

Overflow

- **Large numbers (but not infinite numbers) are approximated as ∞ or −∞**.
- Further operations may result in NAN (not a number).

Overflow

```
import numpy as np
np.exp(1), np.exp(100), np.exp(1000)
2.71, 2.68e+43, inf
```

Of course not actually inf.

Non-a-number

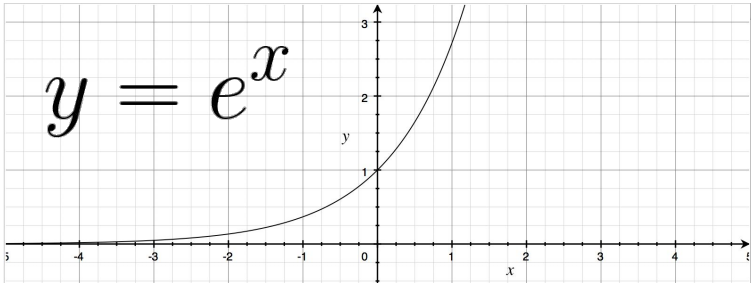
```
>>> import numpy as np
>>> np.exp(1000) / np.exp(1000)
nan

RuntimeWarning: overflow encountered in exp
```

Sometimes you'll get useful warnings.

Recall softmax

The last activation in a network used for classification. Normalizes each output to 0 < x < 1, and such that they sum to 1.

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$



Reminder

Why exp? No prediction will have zero or negative probability.

Notice, no parameters to learn - just a function to convert scores to probabilities.

Naive implementation / works well so far…

```
import numpy as np
softmax = np.exp(scores) / np.sum(np.exp(scores))
```

```
>>> softmax([1,2,3])
# 0.09, 0.24, 0.66
```

Higher scores increase output multiplicatively

```
>>> softmax([0, 0, 10])
# 4.53e-05 4.53-05 9.99e-01
```

Outputs may approach 1 (but will always be less than 1, rounding errors aside)

```
>>> softmax([-10, -5, -8])
# 0.006 0.946 0.047
```

No output ever has zero or negative probability

… but suffers from overflow & underflow

```
import numpy as np
softmax = np.exp(scores) / np.sum(np.exp(scores))
```

```
>>> softmax([1000,1000])
array([nan, nan])
```

Overflow

Discussion, when would we see large inputs to softmax like this?

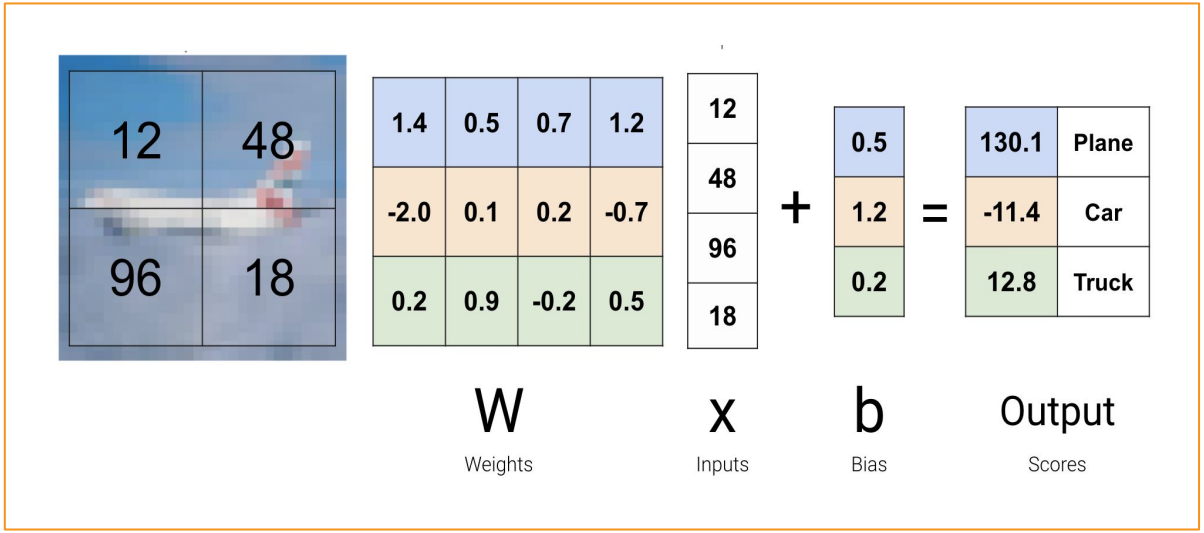
The "." is NumPy telling us this is a floating point value.

```
>>> softmax([-1000, -10, -8])
array([0., 0.11, 0.88])
```

Underflow

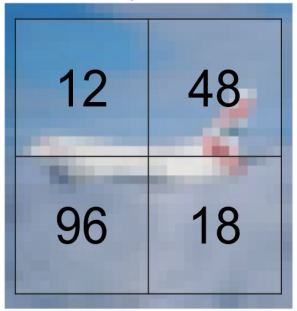
Discussion, when would we see small inputs to softmax like this?

Recall: input to softmax is scores (Wx + b)



Quick discussion: Example that will cause overflow / underflow?

Recall: input to softmax is scores (Wx + b)



1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7
0.2	0.9	-0.2	0.5

W
Weights

12
48
96
18

x
Inputs

+

0.5
1.2
0.2

b
Bias

=

130.1	Plane
-11.4	Car
12.8	Truck

Output
Scores

Overflow: Wx is very large (imagine a large image / large weights)

Underflow: Wx is very small (most weights / pixels close to zero) - or network is super confident it's not a certain class

Stabilizing softmax

```
import numpy as np
softmax = np.exp(scores) / np.sum(np.exp(scores))
```

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

```
>>> softmax([1, 1, 1])
0.33, 0.33, 0.33
```

```
>>> softmax([2, 2, 2])
0.33, 0.33, 0.33
```

Changing each input by a constant doesn't effect the result.

Stabilizing softmax

```
import numpy as np
softmax = np.exp(scores) / np.sum(np.exp(scores))
```

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

```
>>> softmax([1, 1, 1])
0.33, 0.33, 0.33
```

```
>>> softmax([2, 2, 2])
0.33, 0.33, 0.33
```

Compute softmax(z) instead. This prevents overflow (largest term is 1) and prevents dividing by zero due to underflow - at least one term in the denominator is 1).

$$z = \mathbf{x} - max_i x_i$$

Subtract the max score from all the scores before computing softmax

Stabilizing softmax

Underflow remains a problem! Later, the loss function may attempt log(0) if the numerator underflows and becomes 0 (not fixed by this trick).

```
import numpy as np
softmax = np.exp(scores) / np.sum(np.exp(scores))
```

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

```
>>> softmax([1, 1, 1])
0.33, 0.33, 0.33
```

```
>>> softmax([2, 2, 2])
0.33, 0.33, 0.33
```

$$z = \mathbf{x} - max_i x_i$$

Recall: Cross Entropy

Each example has a label in a one-hot format

This is a bird

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	0	0	0	0	0	True probabilities
0.1	0.2	0.6	0.2	0.0	0.0	0.0	0.0	0.0	0.0	Predicted probabilities

Rounded! Softmax output is always $0 < x < 1$

Cross entropy loss for a batch of examples

True prob (either 1 or 0) in our case!

$$L = -\sum \hat{y}_i \log(y_i)$$

Sum over all examples

Predicted prob (between 0-1)

Stabilizing cross entropy

Cross entropy loss for a batch of examples

True prob (either 1 or 0) in our case!

$$L = -\sum \hat{y}_i \, log(y_i)$$

Sum over all examples

Predicted prob (between 0-1)

1) Where is the problem?

2) What can we do to prevent it?

Stabilizing cross entropy

Cross entropy loss for a batch of examples

True prob (either 1 or 0) in our case!

$$L = -\sum \hat{y}_i \boxed{log(y_i)}$$

Sum over all examples

Predicted prob (between 0-1)

May attempt log(0) if Softmax underflows and returns 0 probability for the true class.

1) Where is the problem?

2) What can we do to prevent it?

Clipping

Guarantees all values in softmax output are 0 < x < 1
before computing cross entropy, so we never attempt log(0).

```
softmax_output = tf.clip_by_value(softmax_output, _epsilon, 1. - _epsilon)
```

Epsilon is a small value (like 0.0001)

Clipping

Guarantees all values in softmax output are $0 < x < 1$ before computing cross entropy, so we never attempt $\log(0)$.

```
softmax_output = tf.clip_by_value(softmax_output, _epsilon, 1. - _epsilon)
return - tf.reduce_sum(target * tf.log(softmax_output), axis)
```

Epsilon is a small value (like 0.0001)

Now compute cross entropy.

Clipping

Guarantees all values in softmax output are 0 < x < 1 before computing cross entropy, so we never attempt log(0).

```
softmax_output = tf.clip_by_value(softmax_output, _epsilon, 1. - _epsilon)
return - tf.reduce_sum(target * tf.log(softmax_output), axis)
```

Epsilon is a small value (like 0.0001)

This technique may feel a bit simplistic (by that I mean - our field is young, and perhaps there's a better way yet to be discovered!) A similar trick is used to deal with exploding gradients.

https://github.com/keras-team/keras/blob/master/keras/backend/tensorflow_backend.py#L3266
https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/clip_by_value

The function you may want to examine is 'categorical_crossentropy' in case that line number changes.

Why is there no support for directly computing cross entropy? #2462

Closed ushnish opened this issue on May 22, 2016 · 7 comments

ushnish commented on May 22, 2016

I see that we have methods for computing softmax and sigmoid cross entropy, which involve taking the softmax or sigmoid of the logit vector and then computing cross entropy with the target, and the weighted and sparse implementations of these. But what if I simply want to compute the cross entropy between 2 vectors?

18

mrry commented on May 22, 2016 Member

We provide optimized cross-entropy implementations that are fused with the softmax/sigmoid implementations because their performance and numerical stability are critical to efficient training.

If however you are just interested in the cross entropy itself, you can compute it directly using code from the beginners tutorial:

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

N.B. DO NOT use this code for training. Use `tf.nn.softmax_cross_entropy_with_logits()` instead.

17 11

A common question you should now know the answer to.

Related concepts

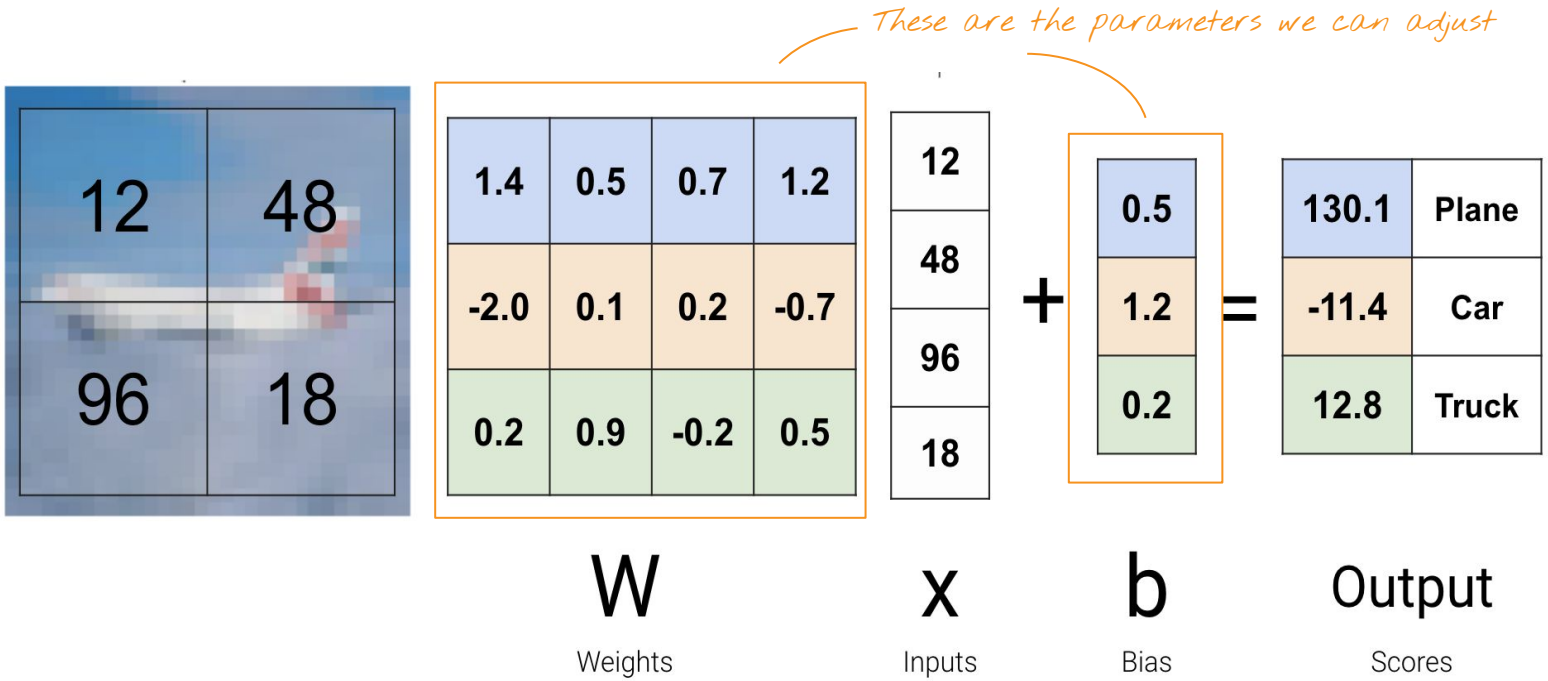
- **Vanishing gradients** (TLDR: the product of a series of numbers less than zero, result heads to zero).

- **Why normalize input data**? Say a column in your input data ranges between -10^6 and 10^6. Instead of feeding these raw values to the network, first subtract mean and divide by standard deviation.

Optimization

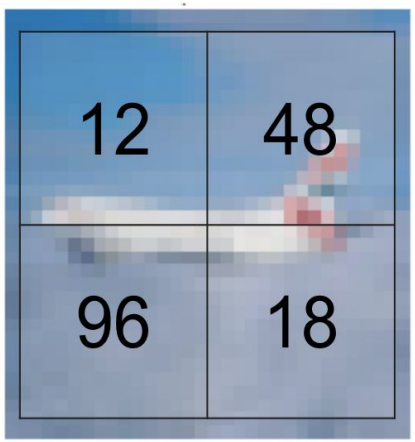
Minimizing (or maximizing) a function by nudging the parameters.

Recall: Loss is a function of weights and inputs



Recall: Loss is a function of weights and inputs

These are the parameters we can adjust



1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7
0.2	0.9	-0.2	0.5

W
Weights

12
48
96
18

x
Inputs

+

0.5
1.2
0.2

b
Bias

=

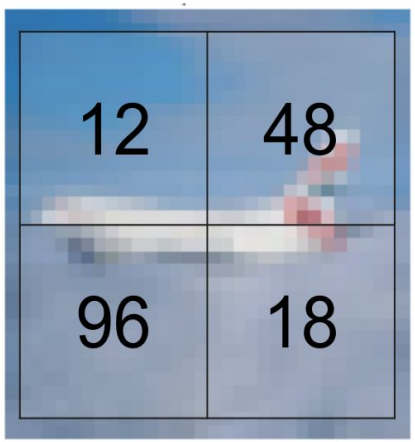
130.1	Plane
-11.4	Car
12.8	Truck

Output
Scores

Can't change the inputs

Recall: Loss is a function of weights and inputs

These are the parameters we can adjust



1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7
0.2	0.9	-0.2	0.5

W
Weights

12
48
96
18

x
Inputs

+

0.5
1.2
0.2

b
Bias

=

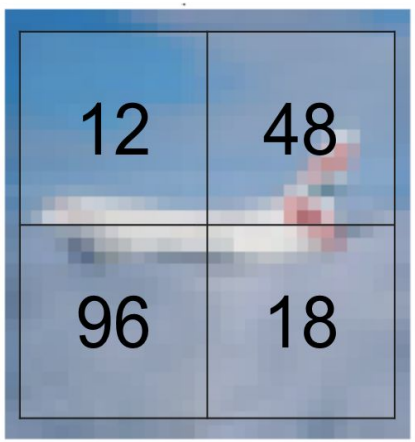
130.1	Plane
-11.4	Car
12.8	Truck

Output
Scores

~~Can't change the inputs~~
Unless we're doing deep dream of style transfer.

Recall: Loss is a function of weights and inputs

These are the parameters we can adjust



1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7
0.2	0.9	-0.2	0.5

12
48
96
18

+

0.5
1.2
0.2

=

130.1	Plane
-11.4	Car
12.8	Truck

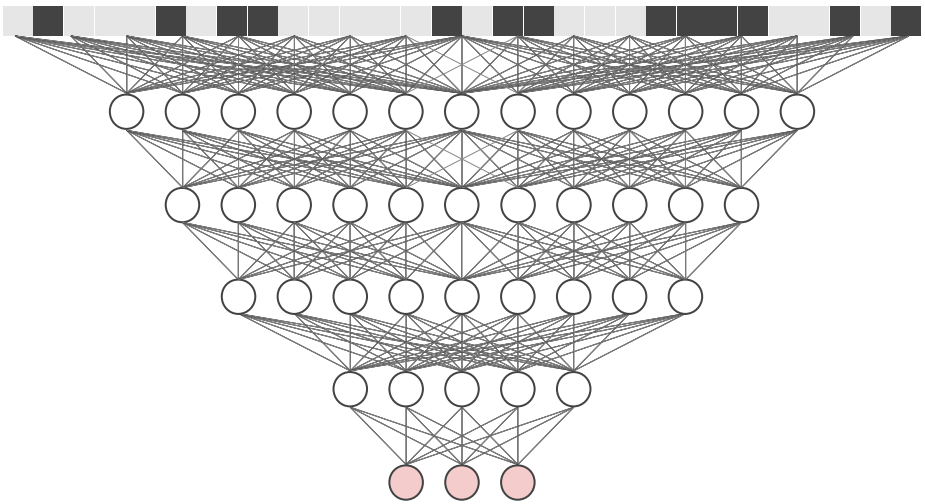
~~Can't change the inputs~~
Unless we're doing deep dream of style transfer.

W	x	b	Output
Weights	Inputs	Bias	Scores

Given a starting value for a weight, what can we do?

How can we find **useful** values for all these weights?

Empirically, don't need a perfect solution to build something that works well in practice.



The fact that this is solvable is only apparent in retrospect (not obvious that it's feasible to do using the gradient - many layers / local minimums, computationally expensive, etc).

Four strategies

Four strategies

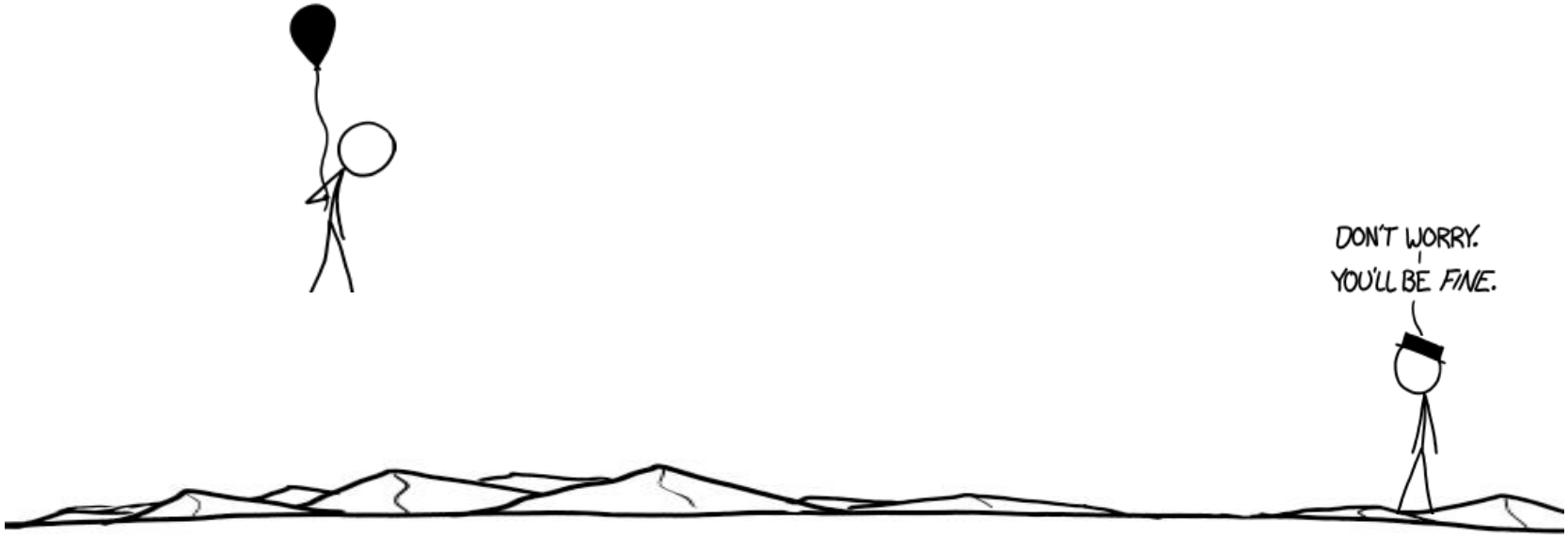
- **Random guess** (randomly guess weights).

Gradient descent

- **Numeric** (calc. gradient by nudging weights and forwarding the function).
- **Analytic** (calc gradient by hand using rules you remember from calculus).
- **Backprop** (compute gradient automatically, using recursive application of chain rule on computational graph).

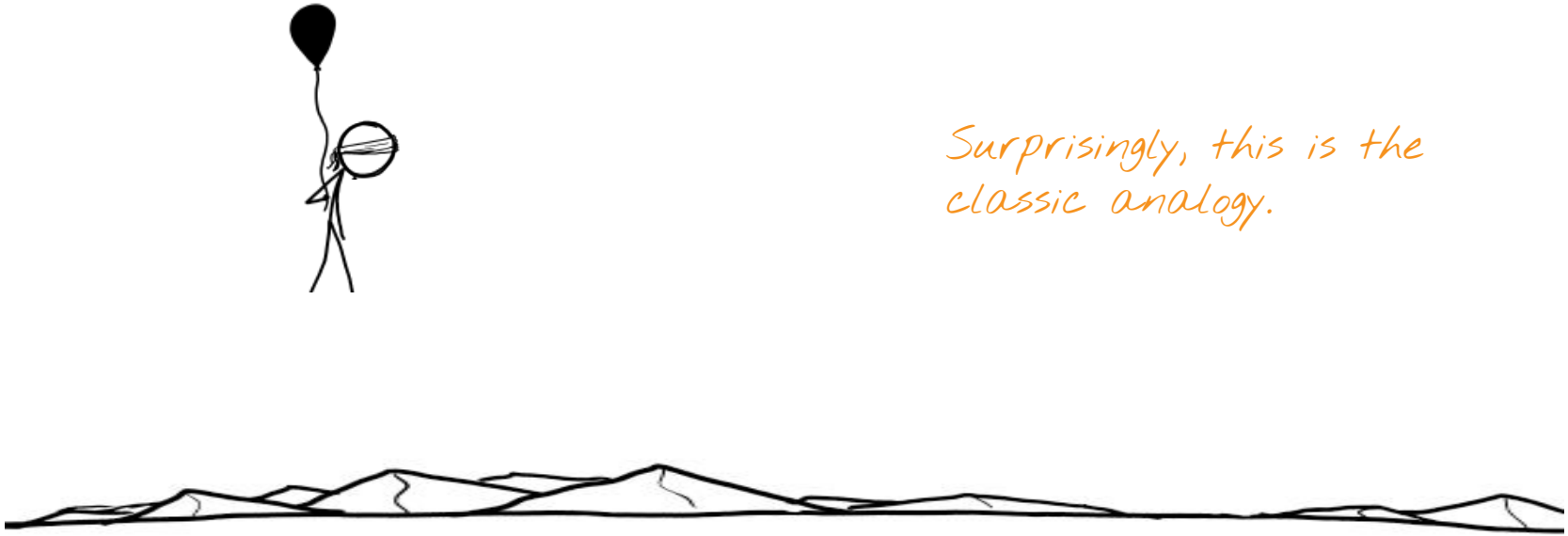
Gradient descent basics

You have been dropped into a mountain range

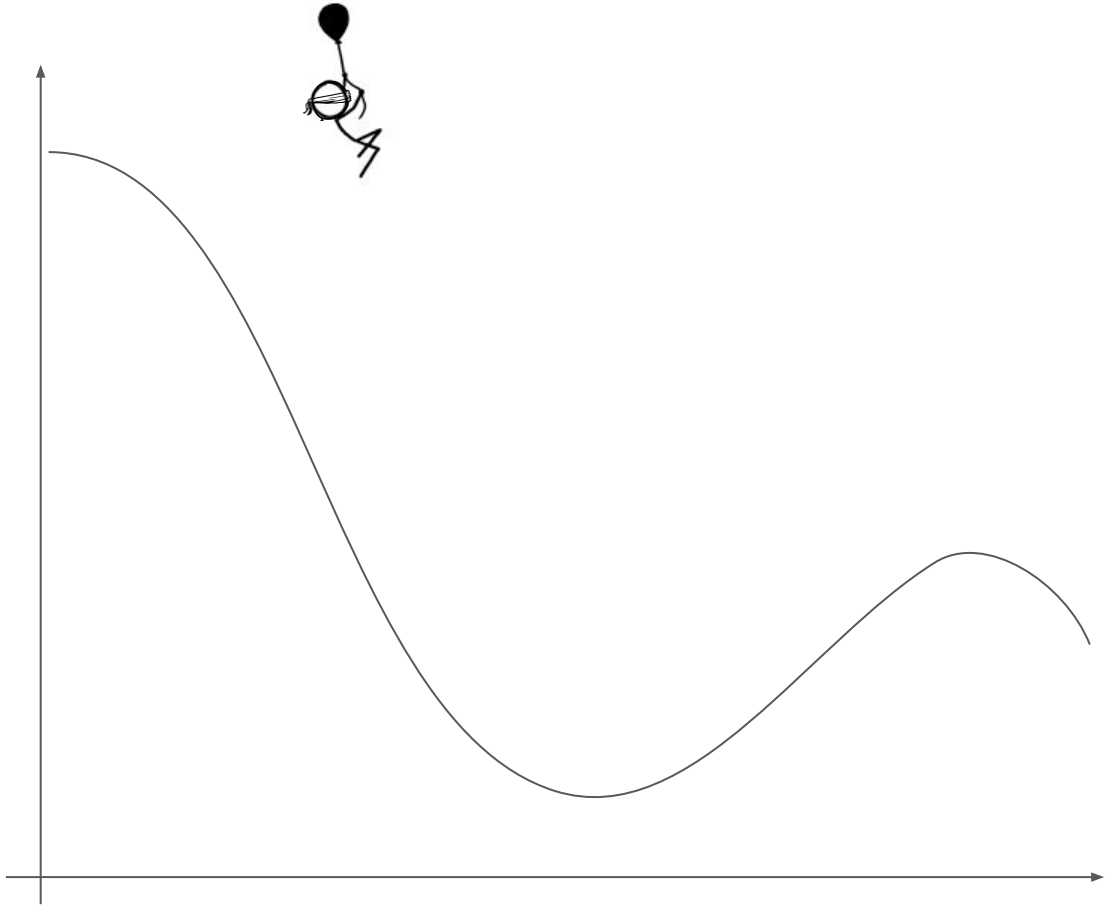


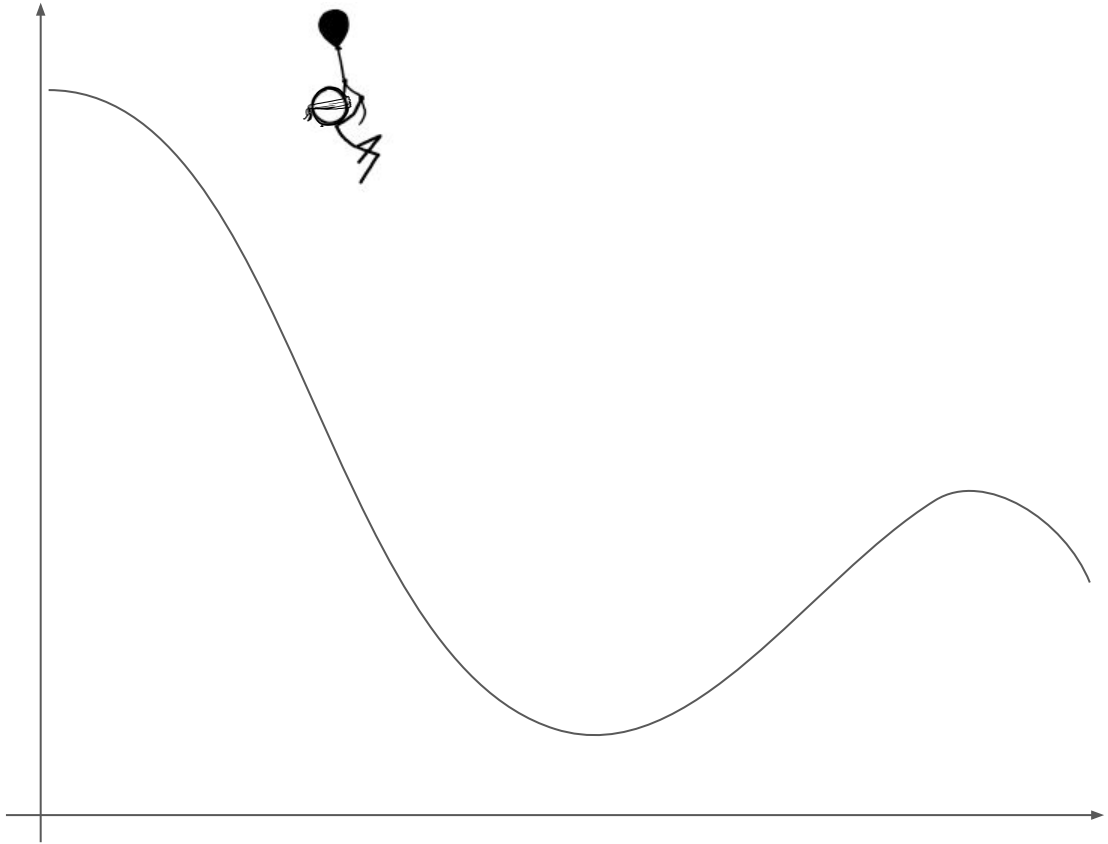
XKCD What if

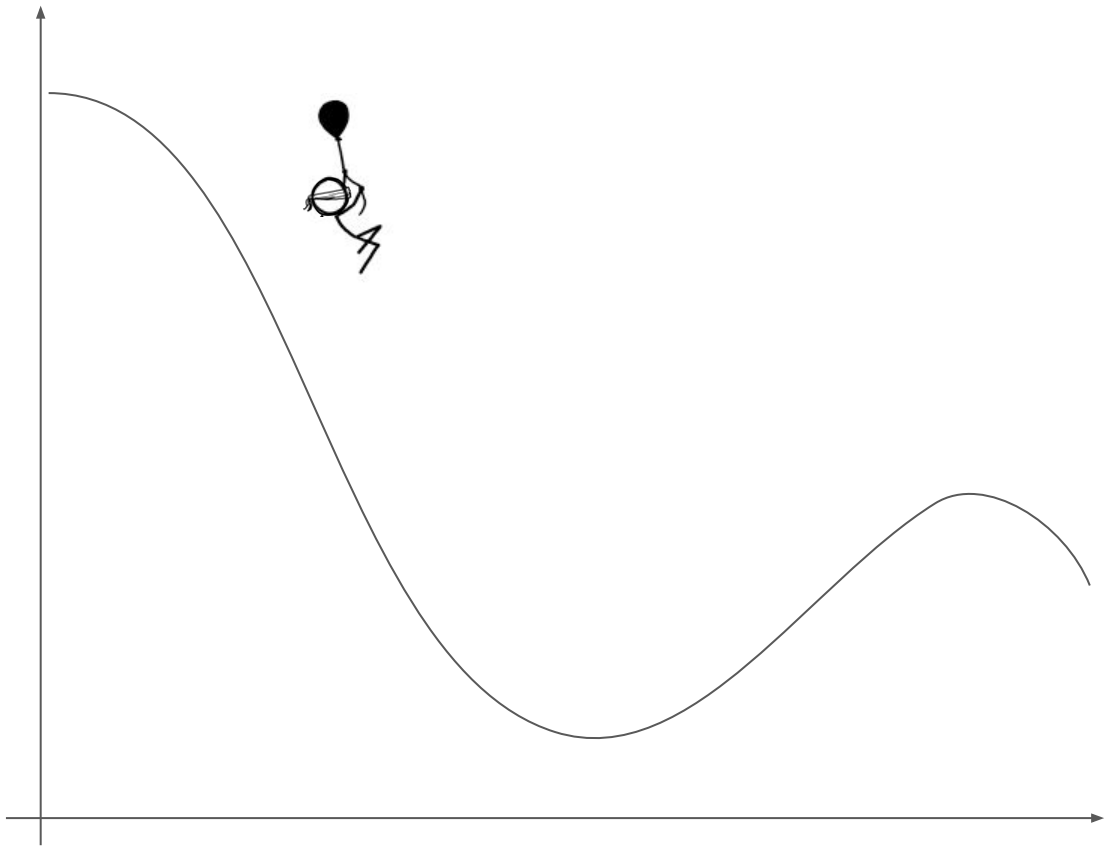
Blindfolded

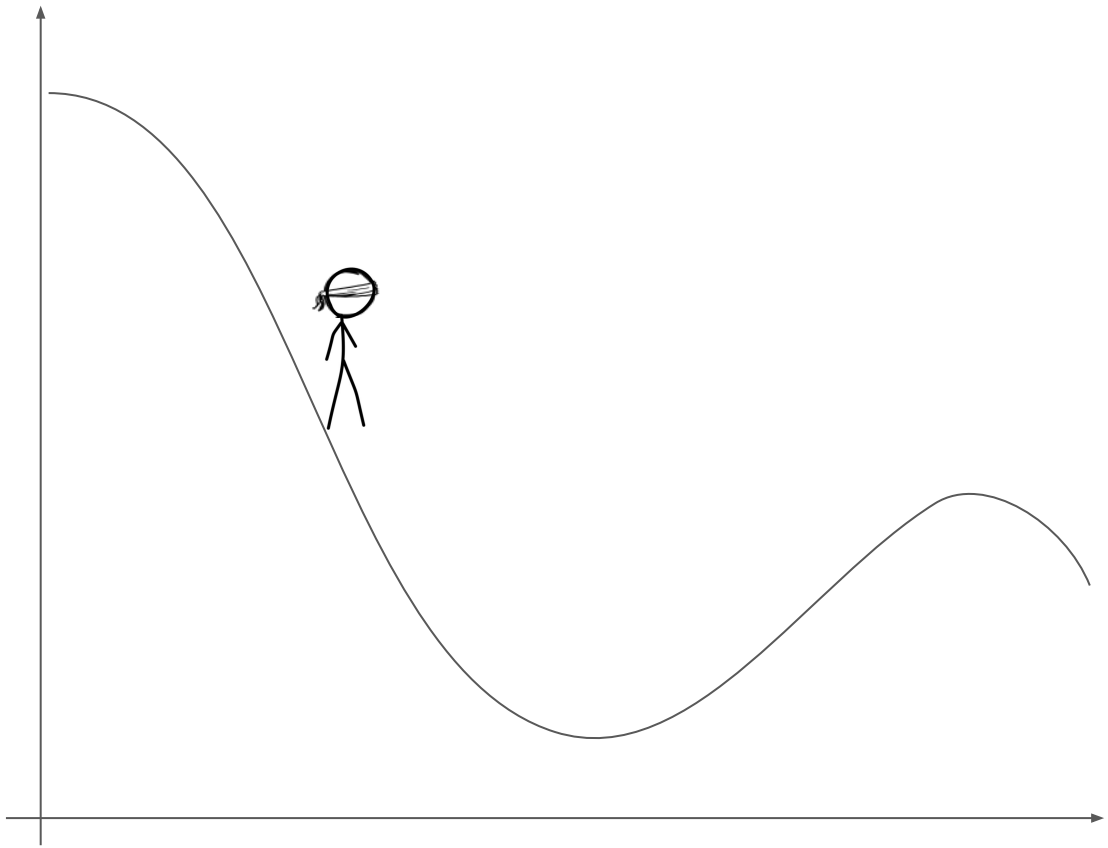


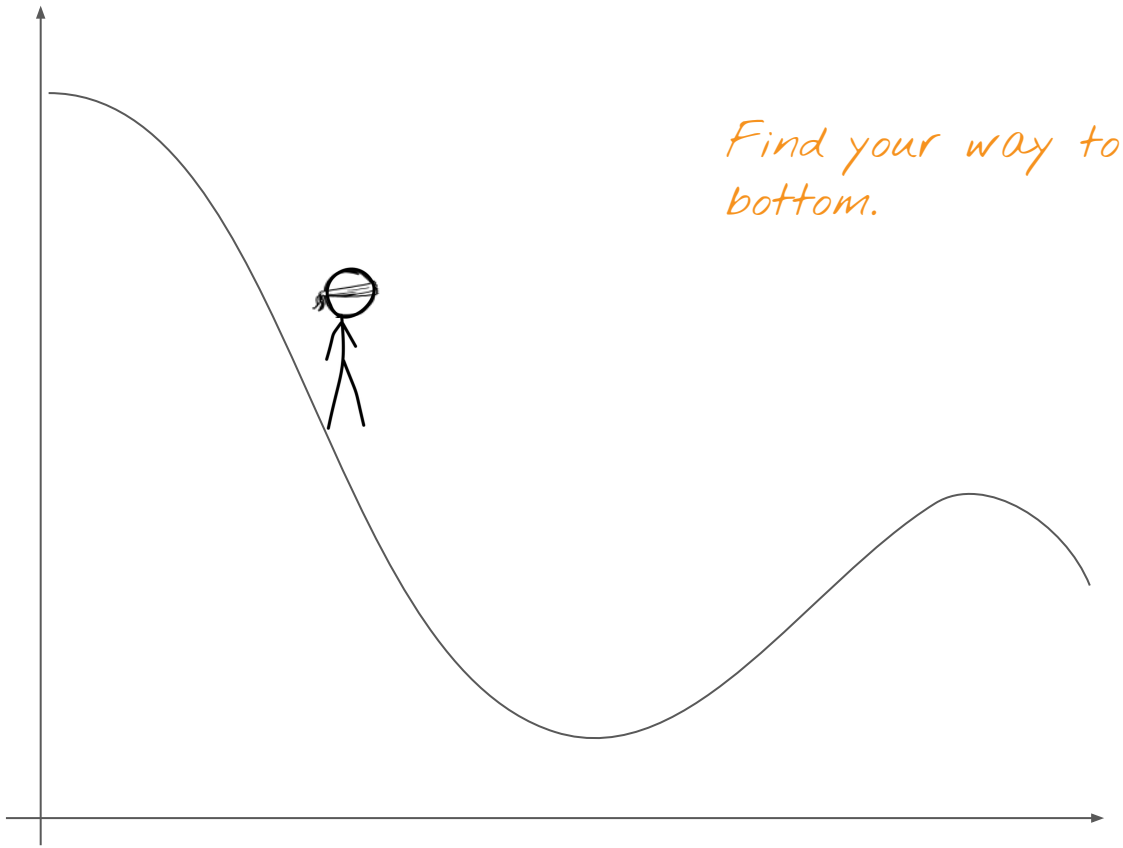
Surprisingly, this is the classic analogy.







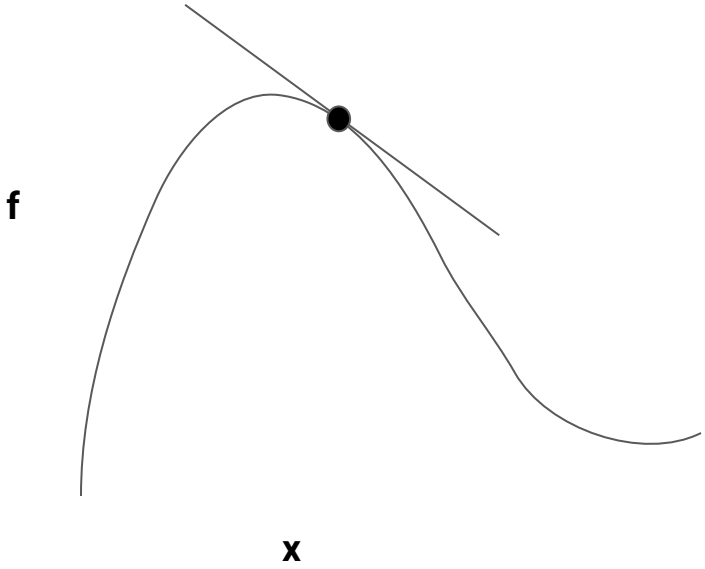




Find your way to the bottom.

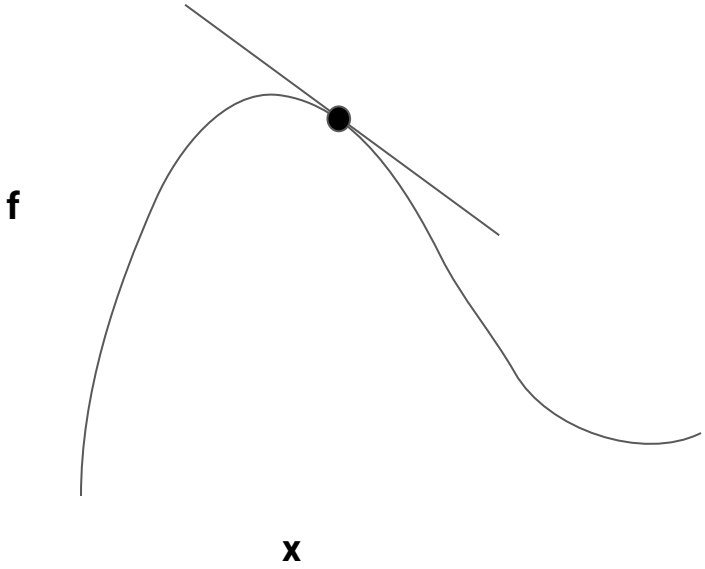
Follow the slope

The derivative of f with respect to x tells us how a tiny change in x causes a tiny change in f. Gives us both direction and magnitude.



$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Follow the slope



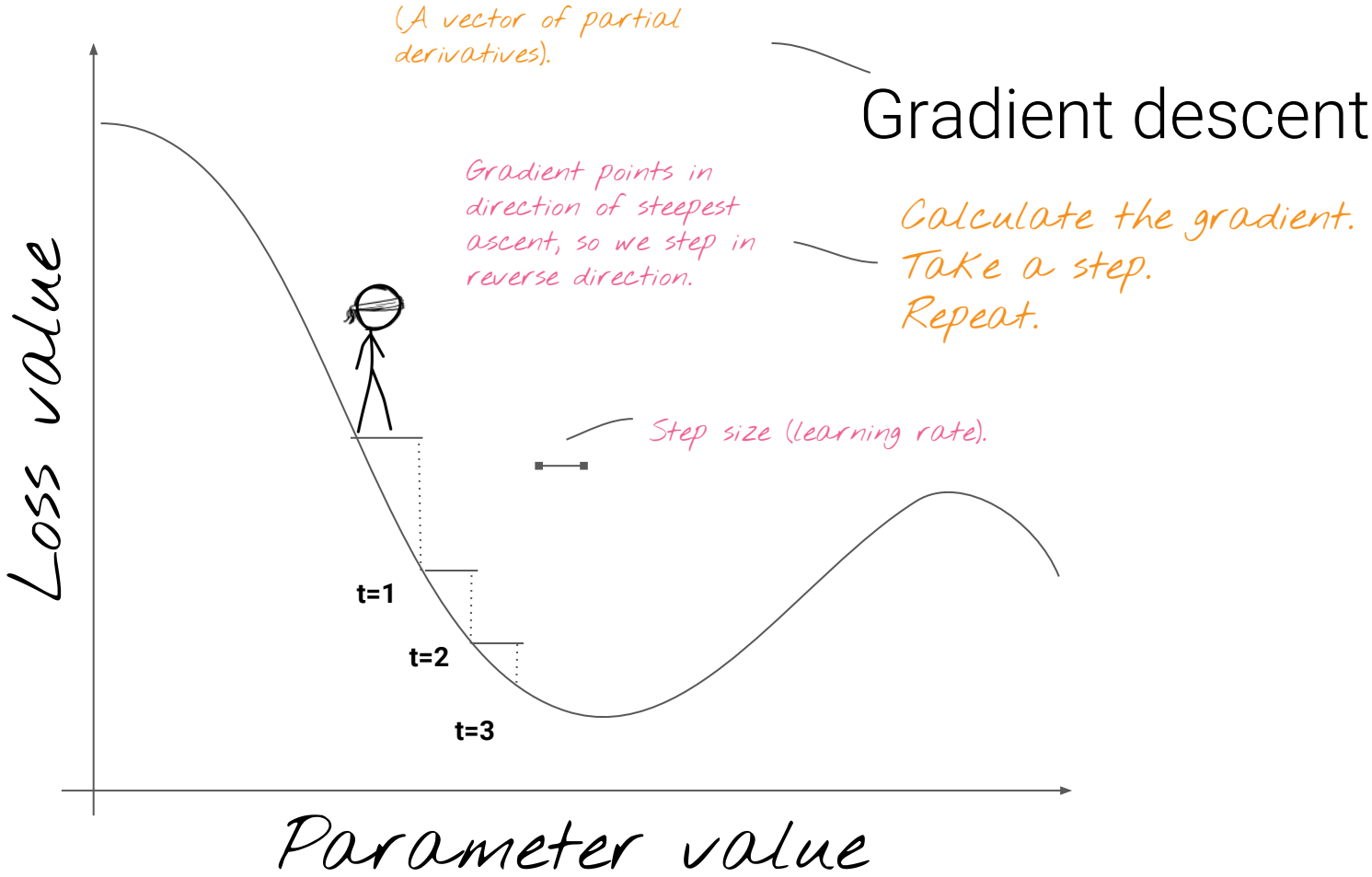
The derivative of f with respect to x tells us how a tiny change in x causes a tiny change in f. Gives us both direction and magnitude.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

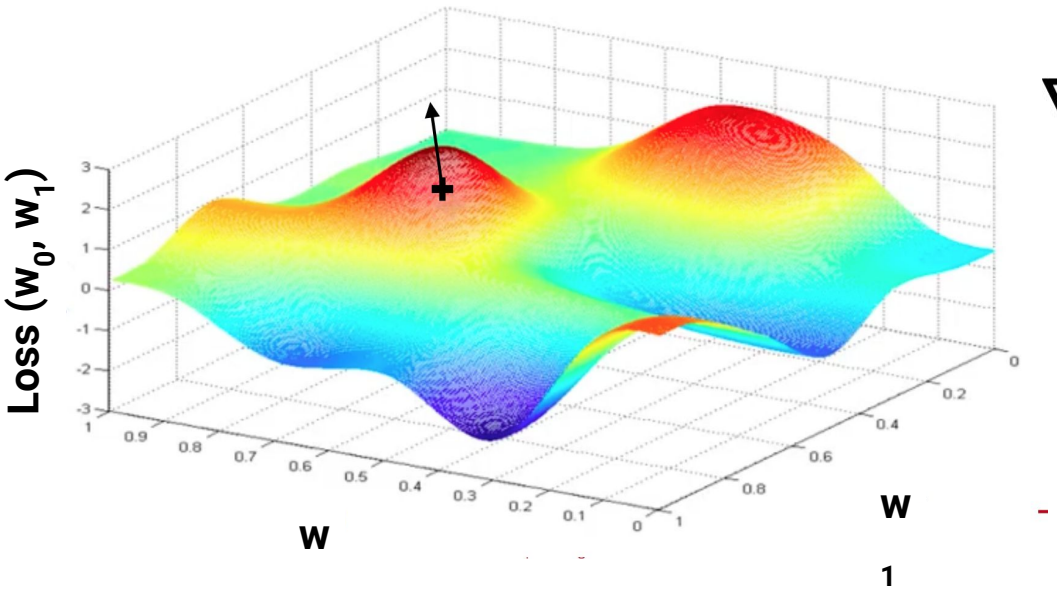
Direction: if negative (as shown to the left), increasing x will decrease f. If positive, increasing x will increase f.

Magnitude: the absolute value of the derivative tells us how quickly f changes proportional to at this point.

Gradient descent



With >1 variable, we need the gradient



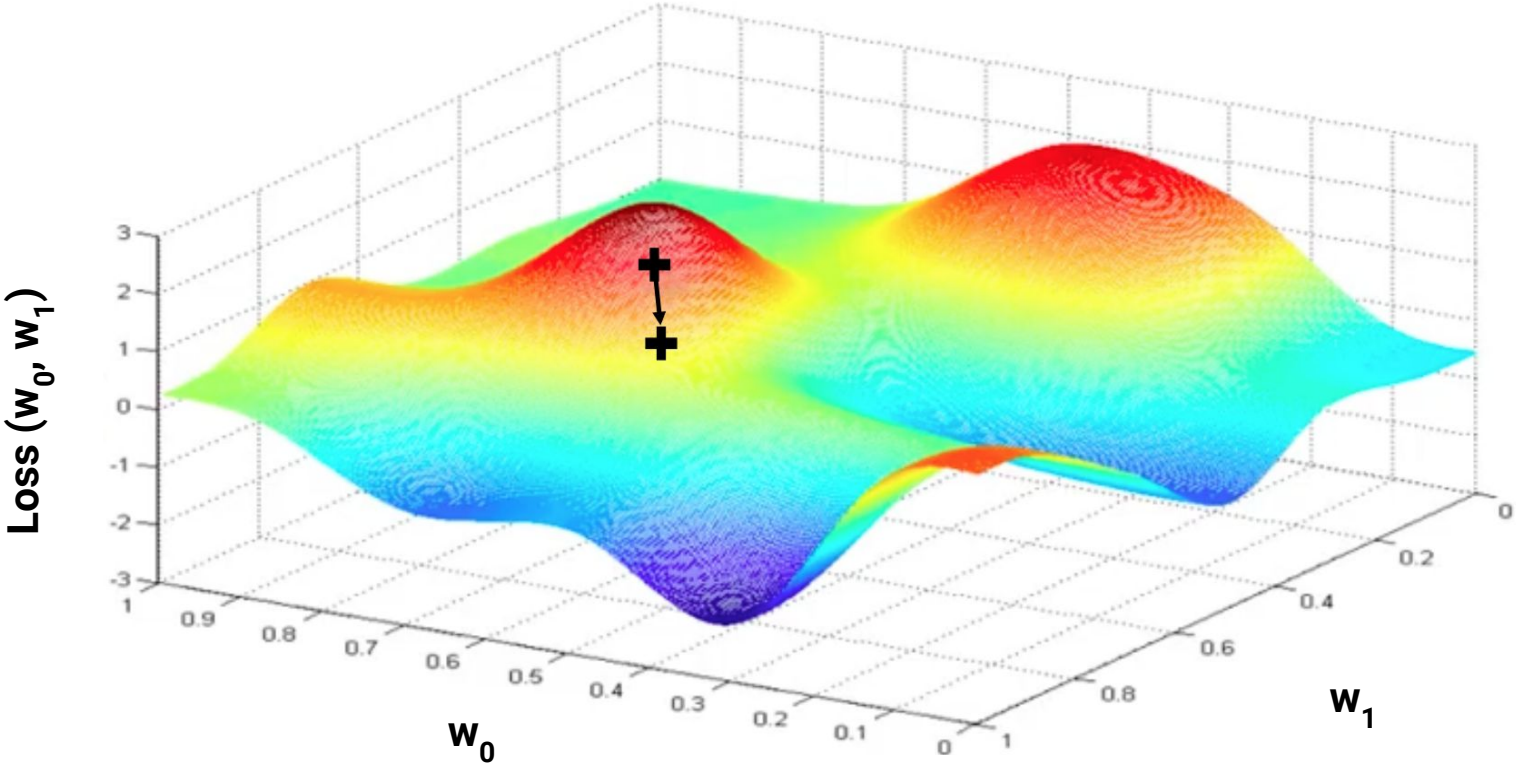
$$\nabla_w Loss = \frac{\partial Loss}{\partial w_0}, \frac{\partial Loss}{\partial w_1}$$

The gradient is a vector of partial derivatives (these are the derivative of a function w.r.t. each variable, while the others are held constant).

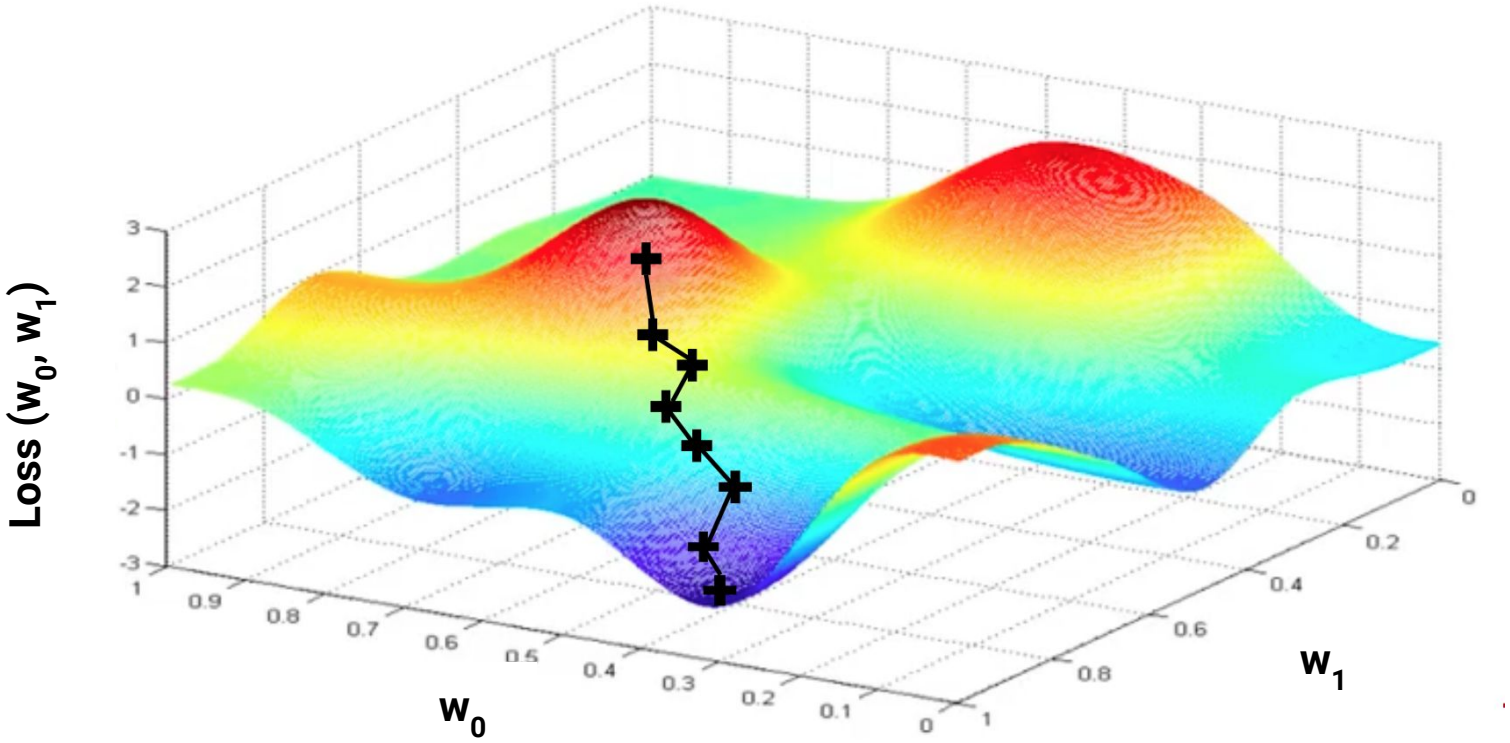
The gradient points in the direction of steepest ascent. We usually want to minimize a function (like loss), so we take a step in the opposite direction..

You'll often see loss abbreviated as "J", and the weights of our model written as θ (theta).

Take small step in opposite direction of gradient.

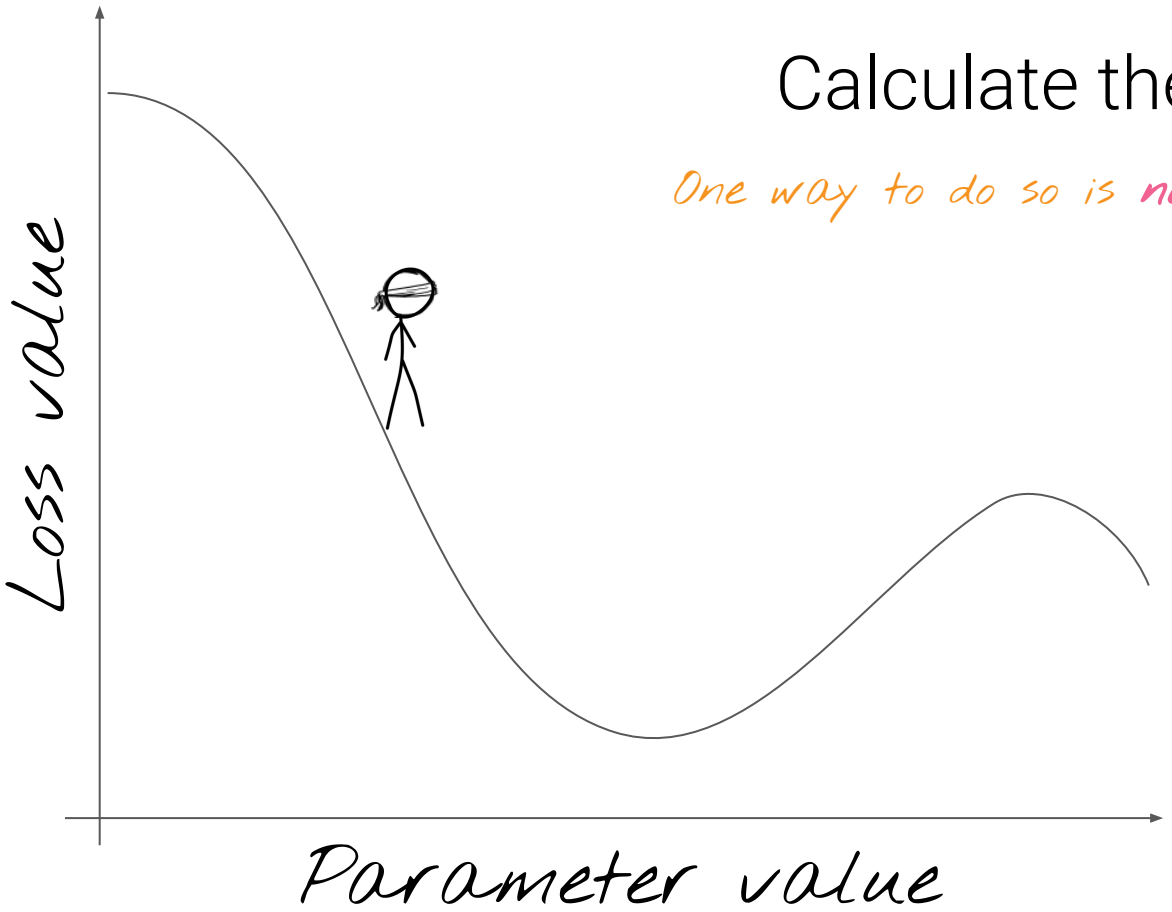


Repeat until convergence



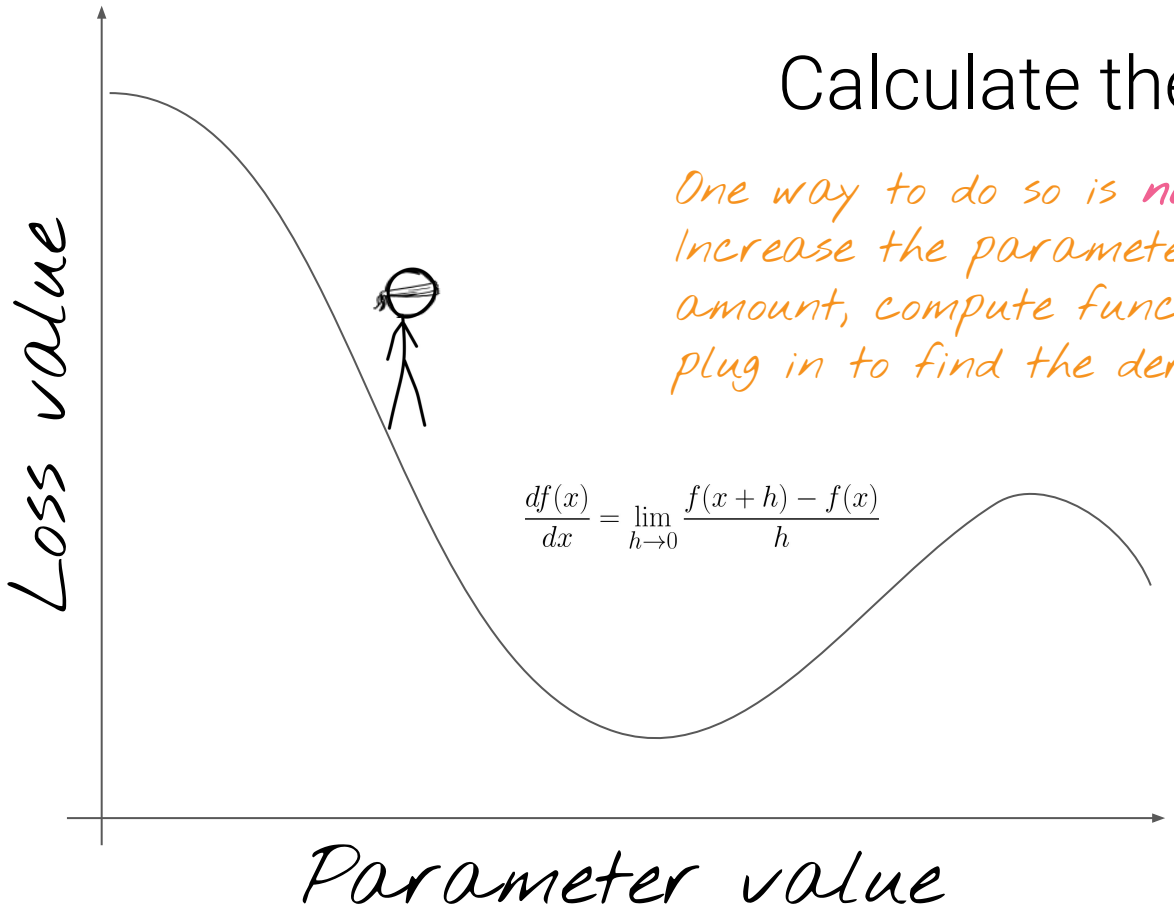
Calculate the gradient

One way to do so is numerically.



Calculate the gradient

One way to do so is **numerically**. Increase the parameter by a tiny amount, compute function output, plug in to find the derivative.

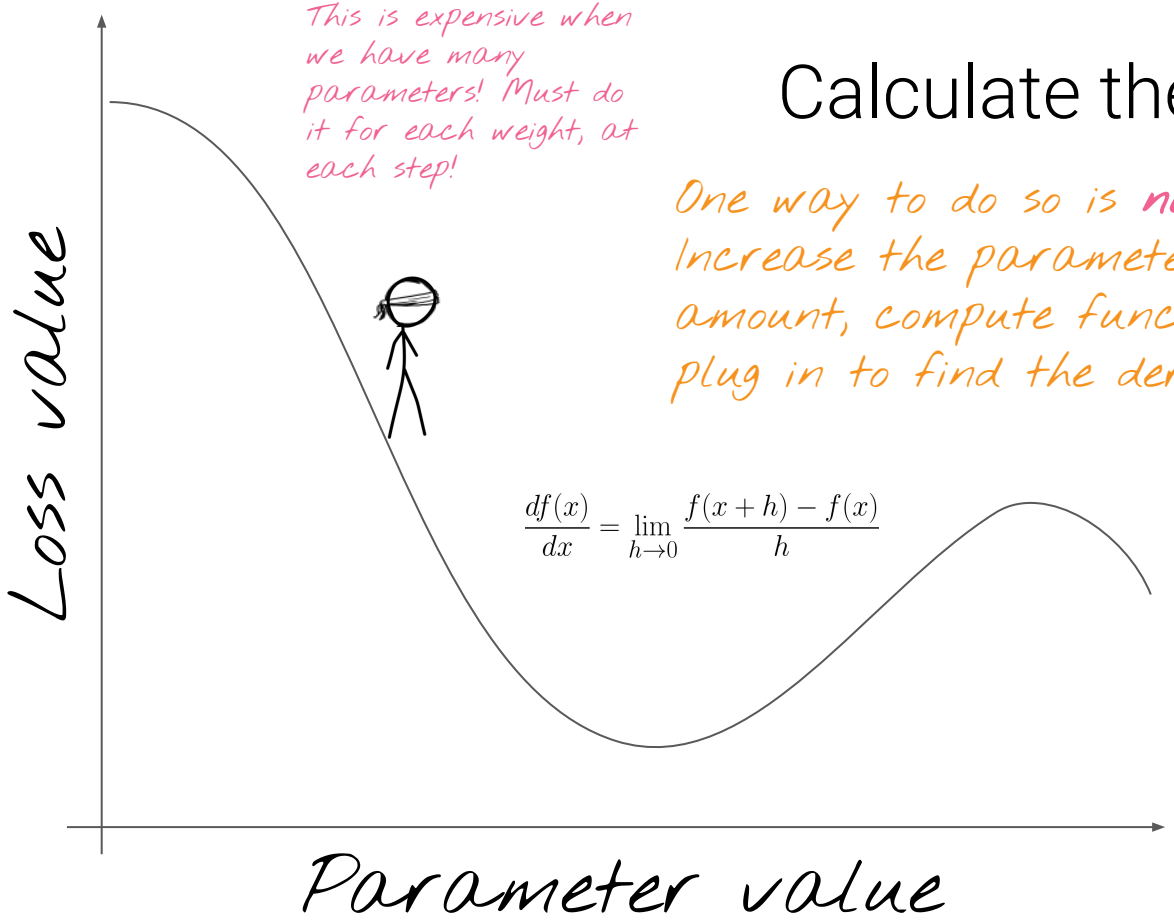


$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Calculate the gradient

This is expensive when we have many parameters! Must do it for each weight, at each step!

One way to do so is **numerically**. Increase the parameter by a tiny amount, compute function output, plug in to find the derivative.

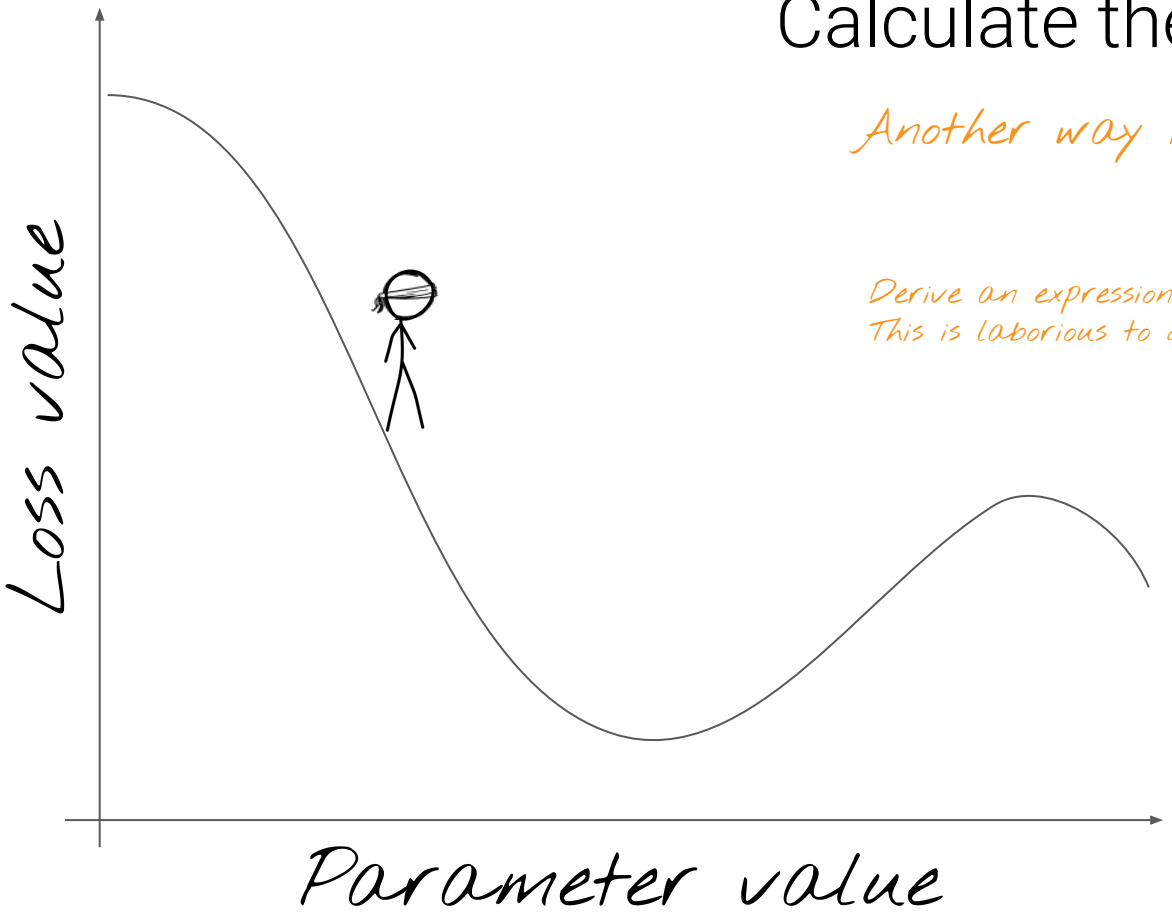


$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Calculate the gradient

Another way is analytically.

Derive an expression for the gradient.
This is laborious to do by hand.



Basically

1. Initialize weights randomly
2. Repeat until convergence
3. Calculate gradient of loss w.r.t. weights. $\nabla_w Loss$
4. Update weights.

$$w_i \leftarrow w_i - \eta \frac{\partial Loss}{\partial w_i}$$

Or other stopping criteria, like max steps, or no further improvement after K successive steps.

Eta (learning rate, or step size - sometimes written as alpha - same thing.)

Two sources of complexity

1. The gradient descent algorithm itself momentum, or adaptive learning rates).

2. The method of computing the gradients themselves (backprop).

Backprop refers only to the method of computing gradients (not the end-to-end optimization process).

Many optimizers

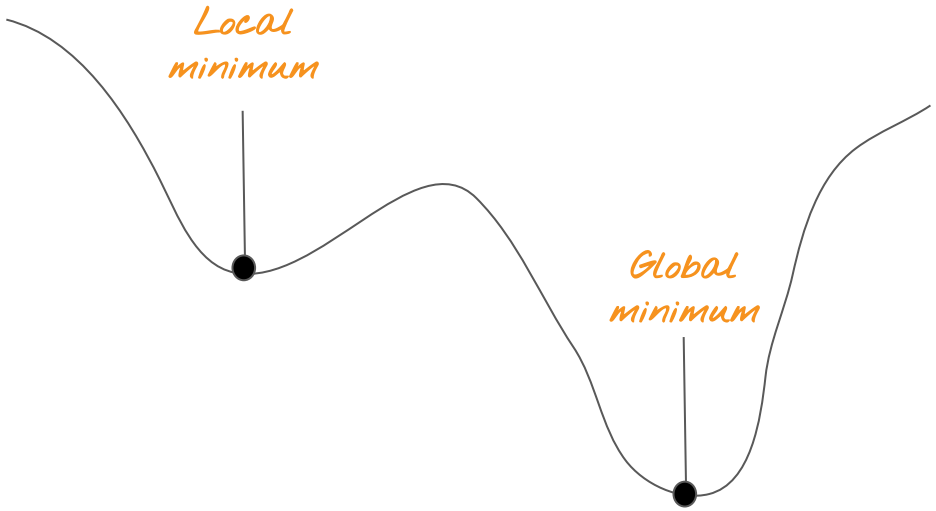
And new ones all the time (see linked papers if you'd like to read ahead).

- Adadelta
- Adagrad
- Adam
- Adamax
- Nadam
- Optimizer
- RMSprop
- SGD

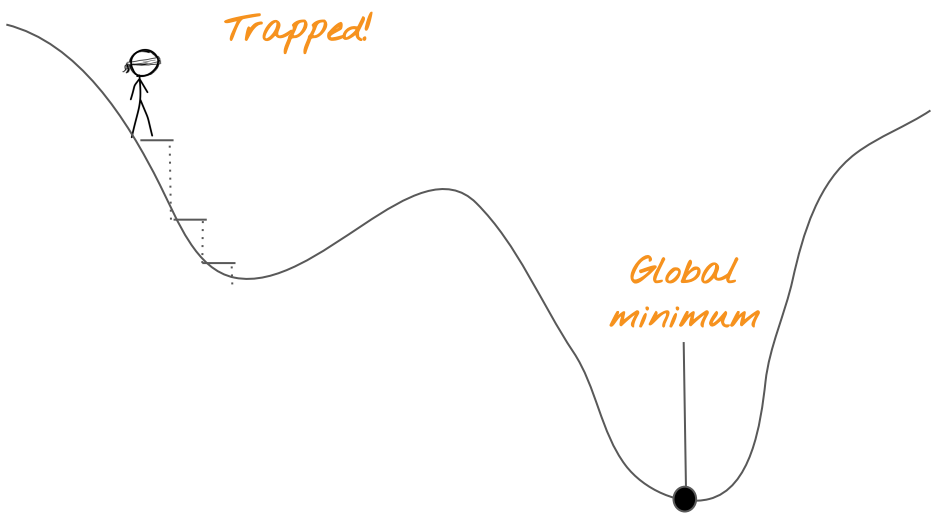
A good default choice.

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/optimizers

Local and global minimum

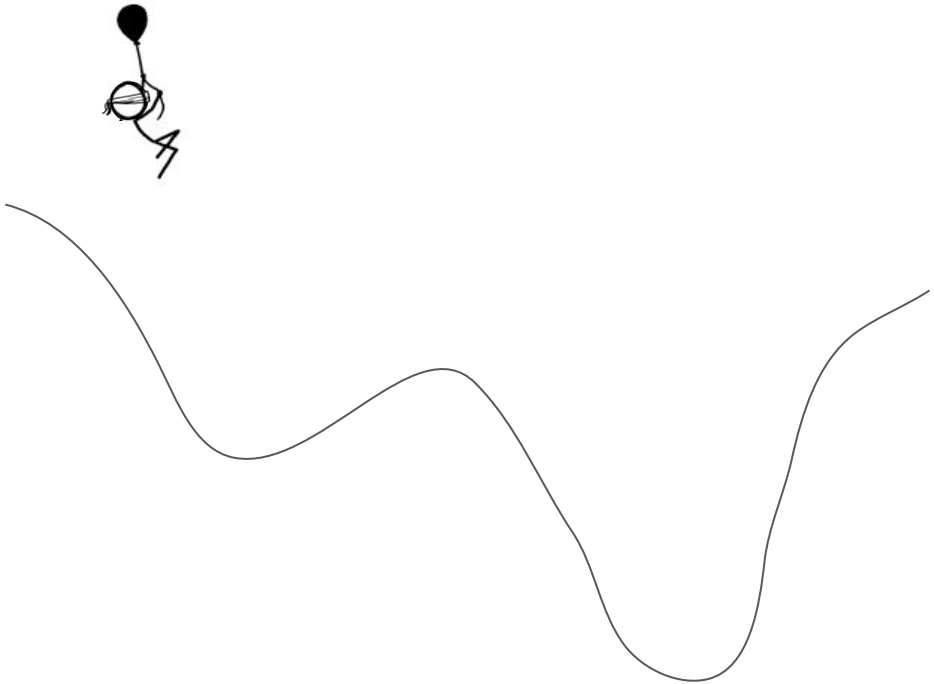


Local and global minimum



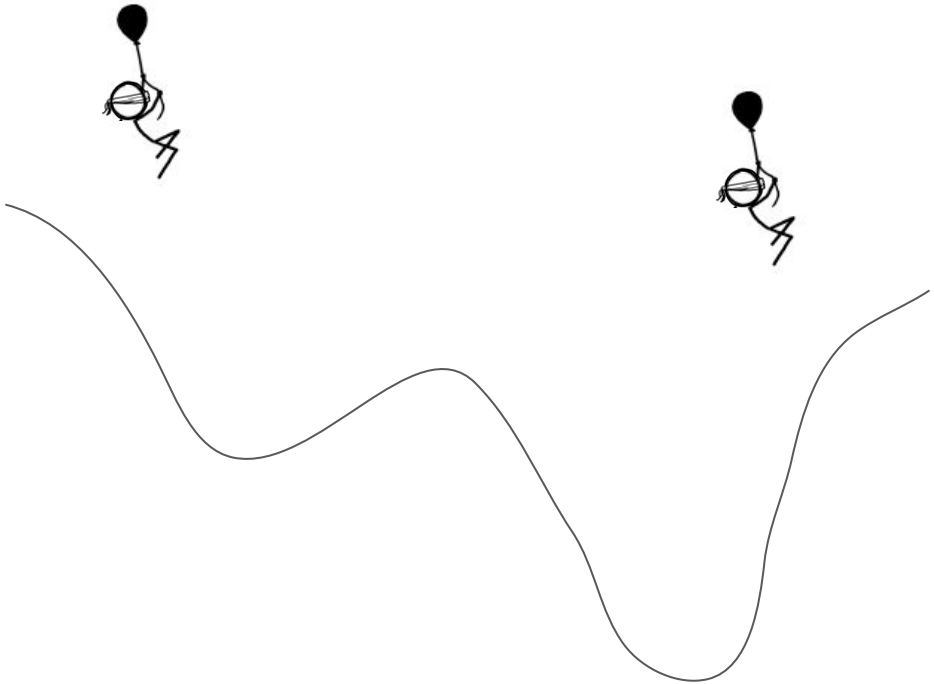
One solution (random restart)

Not often needed, just FYI.



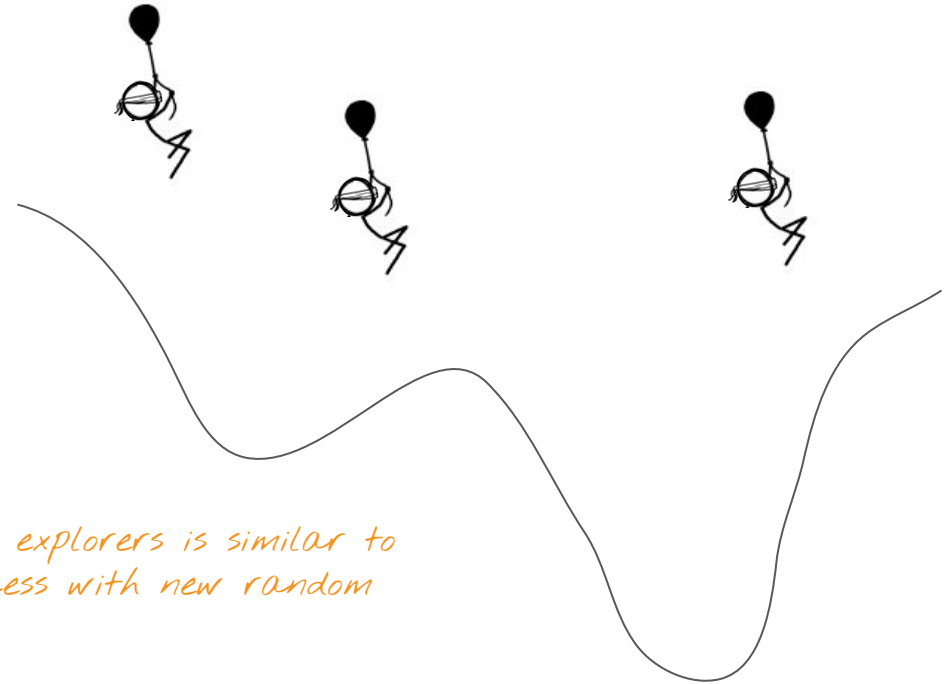
One solution (random restart)

Not often needed, just FYI.



One solution (random restart)

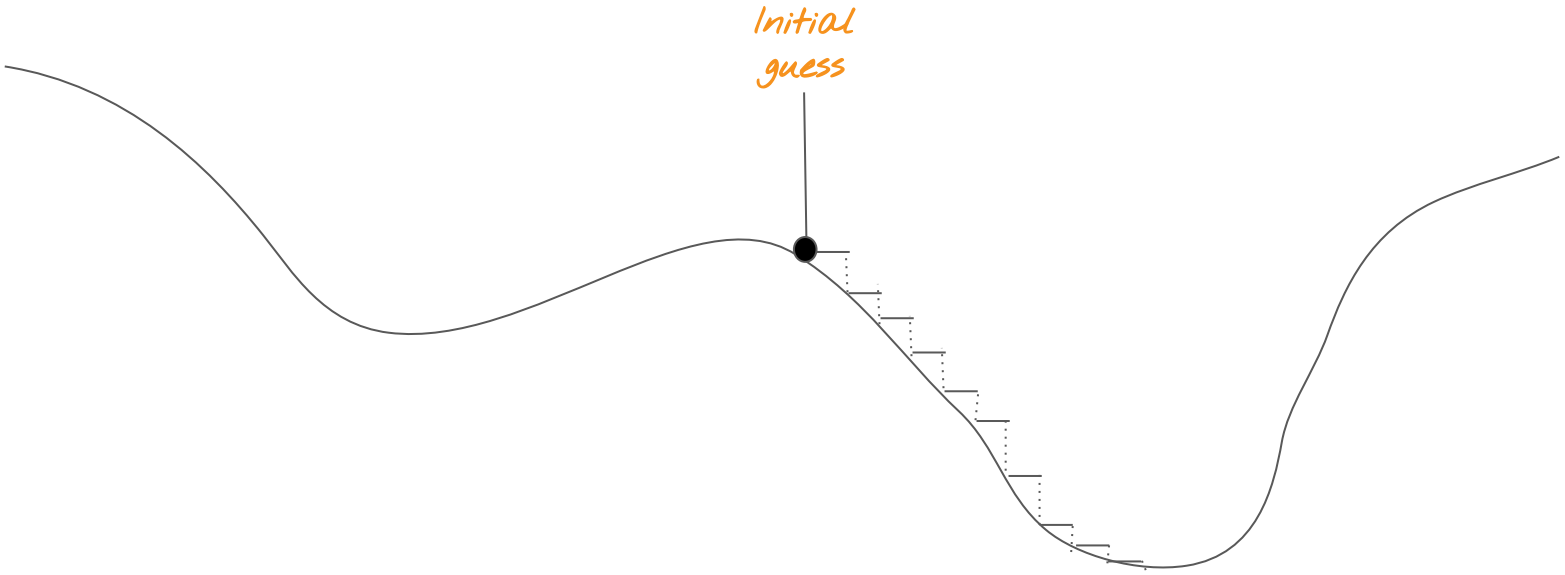
Not often needed, just FYI.



Parachuting many explorers is similar to restarting the process with new random weights.

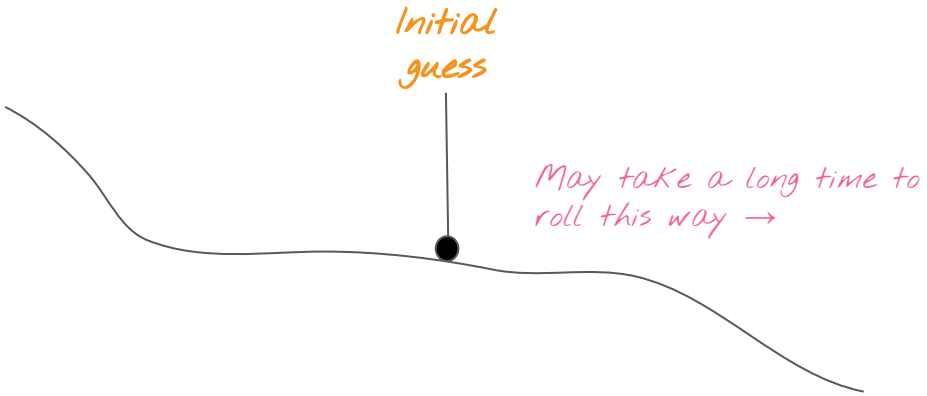
Learning rates

A low learning rate could take many steps to converge



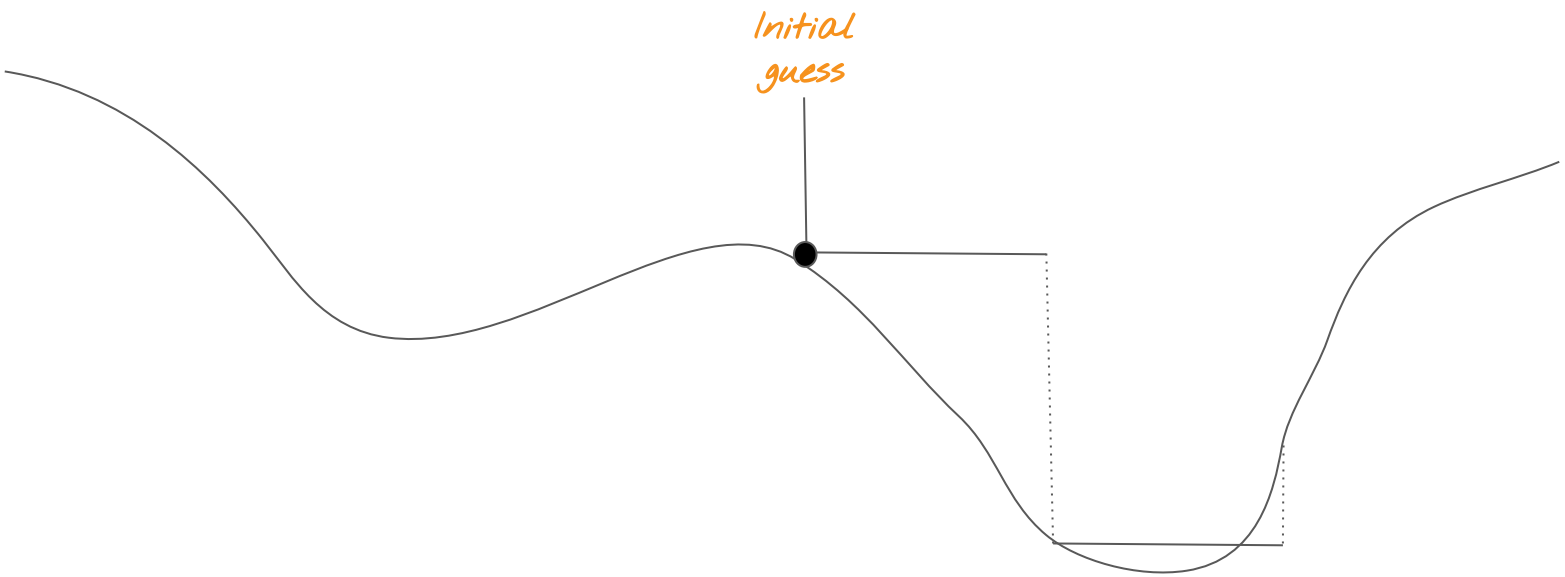
Learning rates

Or, stall in regions where the gradient is small.



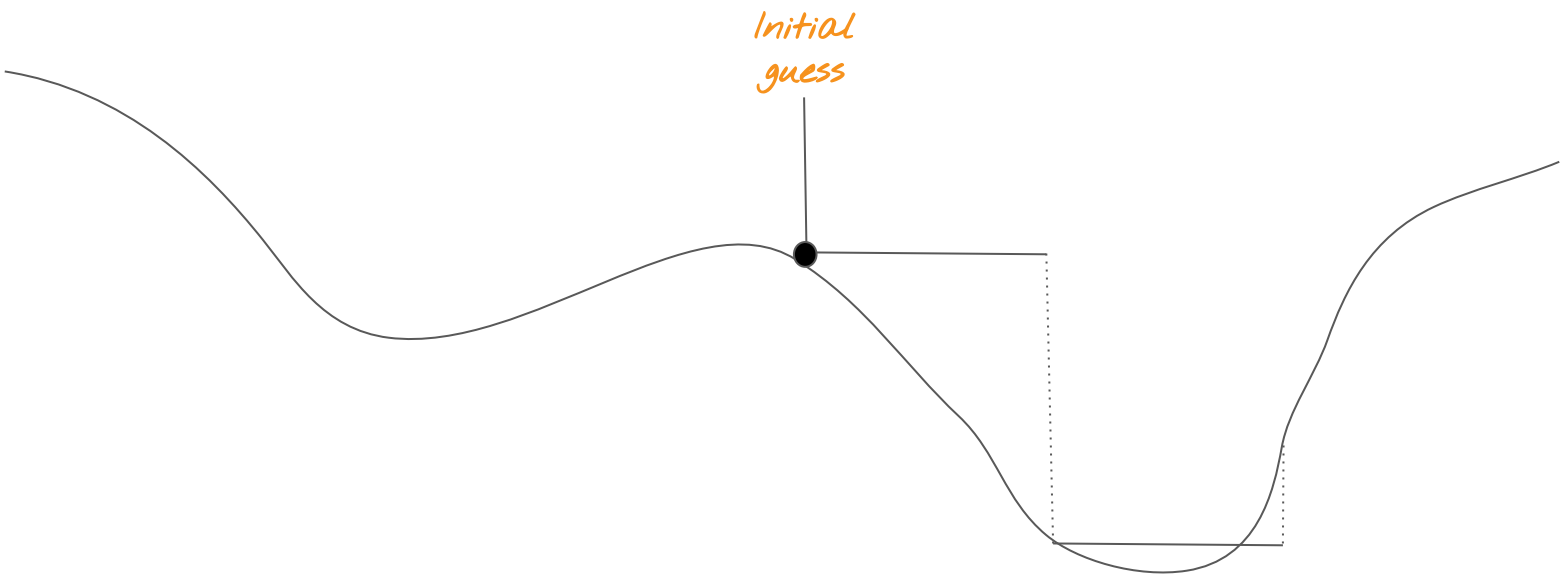
Learning rates

A high learning rate could jump over the minimum!



Learning rates

Or oscillate around it, never converging.



Batch vs mini-batch vs stochastic

These names are silly. I agree w/ Aurelien:

Existing names

Note: we update weights using the average gradient. We have to compute the gradient for each example in a batch (we can't just compute loss, then backprop once).

Batch

- Use entire training set to compute gradient.

Stochastic

- Use a single training example at a time.

Mini-batch

- Use a small batch of data (typically ~32 to ~ 128 examples).

Existing names

Note: we update weights using the average gradient. We have to compute the gradient for each example in a batch (we can't just compute loss, then backprop once).

Batch

- Use entire training set to compute gradient.

Stochastic

- Use a single training example at a time.

Faster, noisier updates.

Mini-batch

- Use a small batch of data (typically ~32 to ~ 128 examples) at a time.

Use mini-batch in practice.

Batch

Pros: accurate updates. *Cons*: slow to compute.

- Use entire training set to compute gradient.

Stochastic

Pros: fast to compute. *Cons*: the gradient of one example is a poor estimate of the entire training set.

- Use a single training example at a time.

Mini-batch

Balanced, more likely to approximate true gradient.

- Use a small batch of data (~32 to ~ 128 examples) at a time.

Bonus: mini-batch gives a “simple” way to parallelize training

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Some of your training data

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Say one machine can handle a maximum batch size of 5.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

With two machines, you can send one mini-batch to each!

You can make updates to your weights at roughly the same speed as one machine, with effectively double the batch size = more accurate updates.

Questions from email

Why do models have more parameters than necessary?

- Gradient descent is unreliable when the network is small, particularly when the network is just the right size to learn the problem, but not larger (more likely to get stuck in a local minimum).

- Solution: make the network larger than necessary, then regularize it.

- See this talk from Yann Lecun.

Calculating the numerical gradient

Computational graphs are a helpful abstraction.

Running example

```
f = (a + b) * (b + 1)
```

Running example

```
f = (a + b) * (b + 1)
```

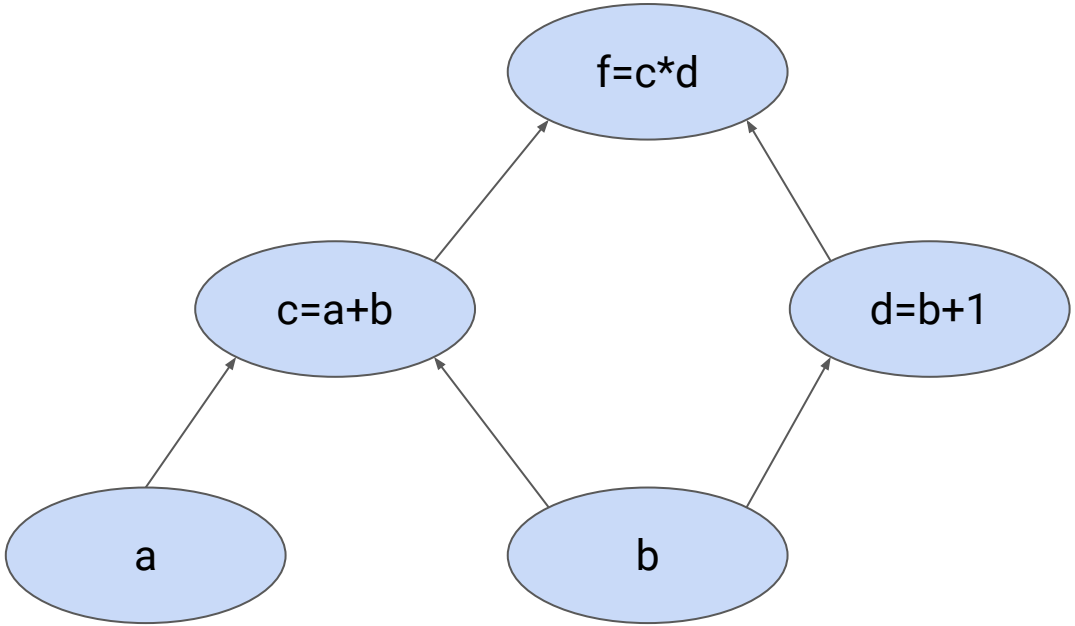
To compute f we need to perform three operations (two additions, one multiplication).

```
c = a + b

d = b + 1

f = c * d
```

Introduce intermediate variables (one for each operation).

A computational graph

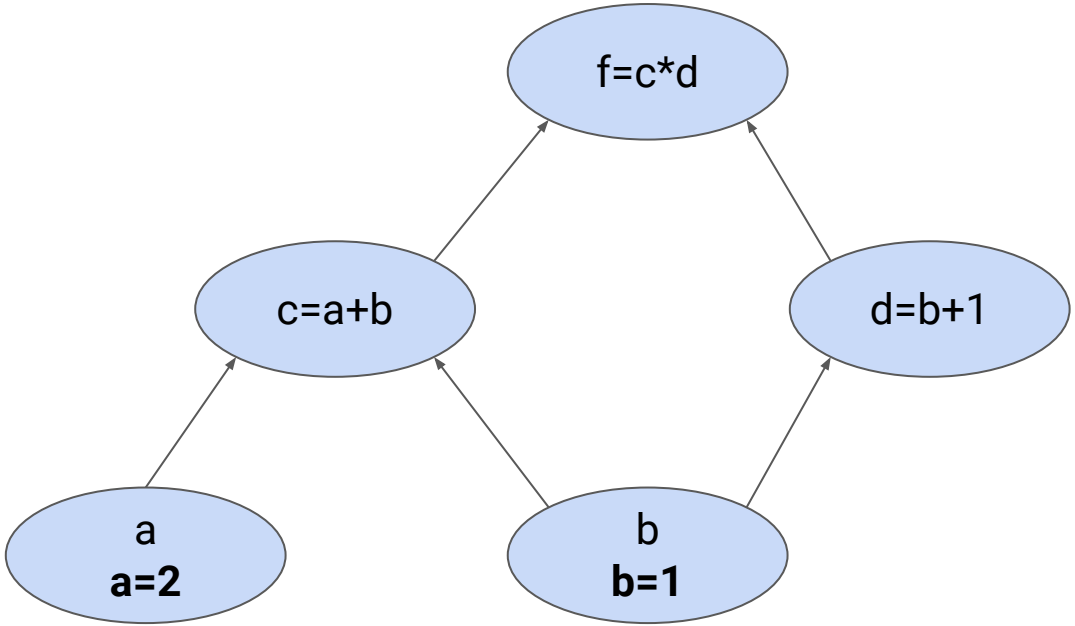
Closely related to a dependency graph.



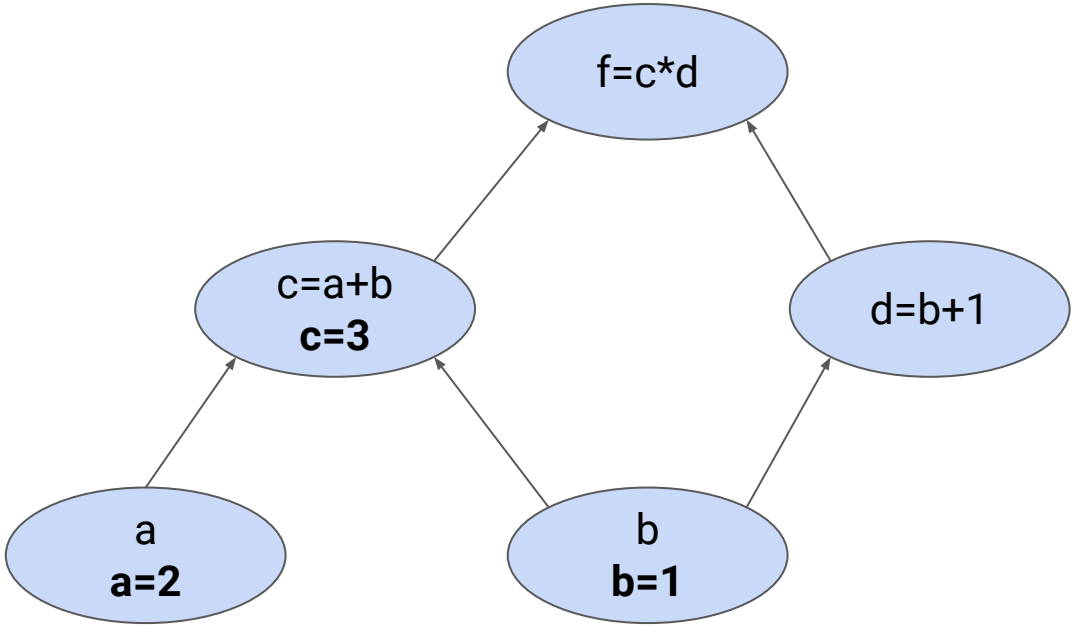
Trivia: the name "**TensorFlow**" comes from the idea of tensors (n-dimensional arrays) flowing on a graph. Here, we're using scalars.

Diagram from Colah's blog. I thought it might be helpful to show how we can solve this in several ways.

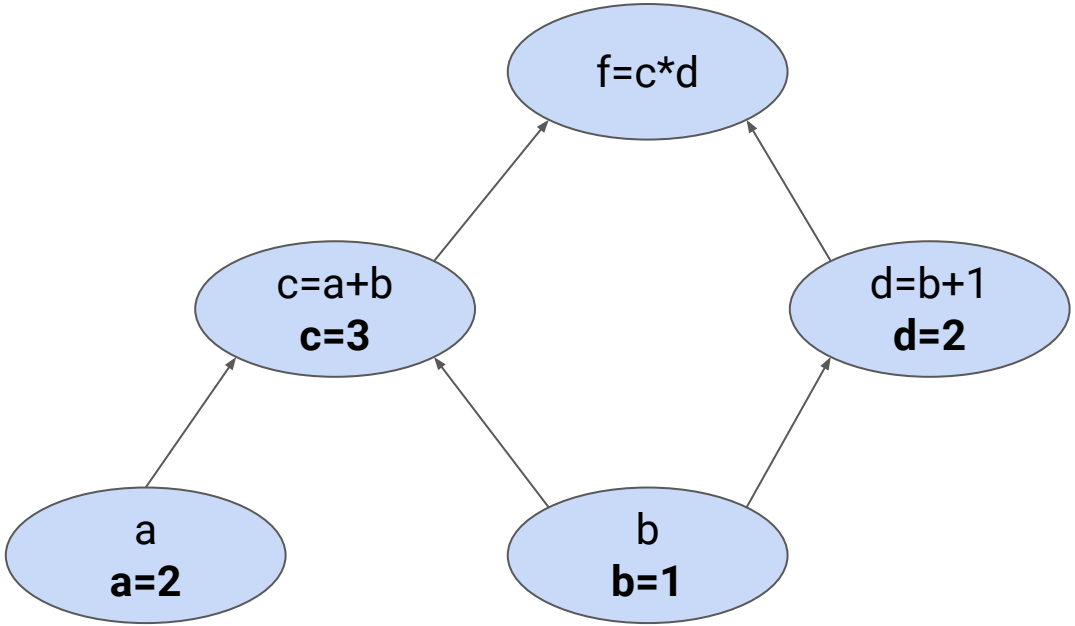
Forward pass



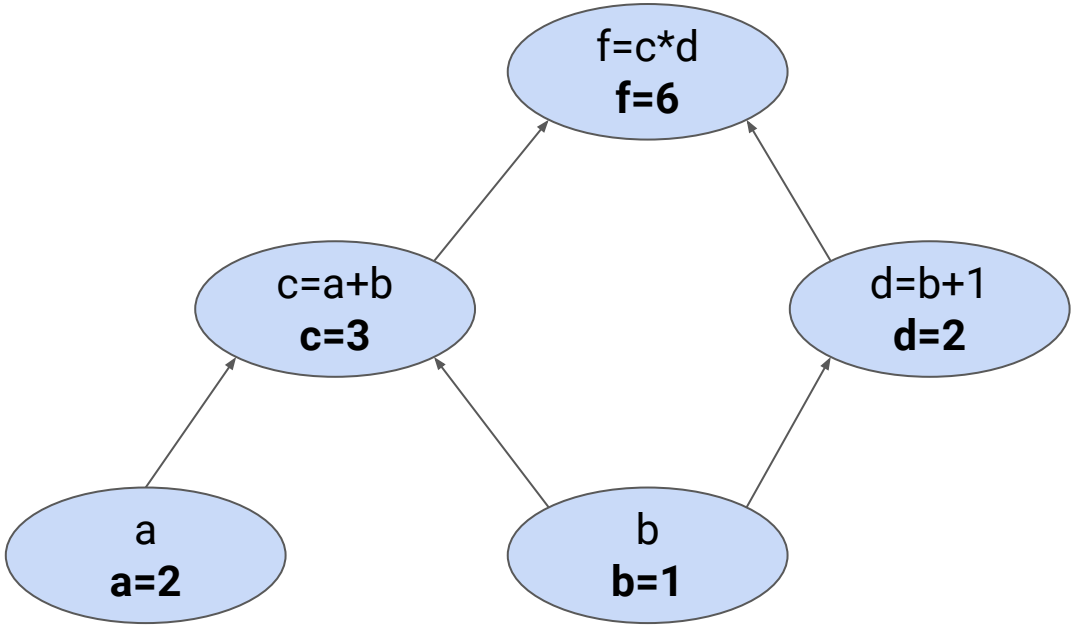
Forward pass



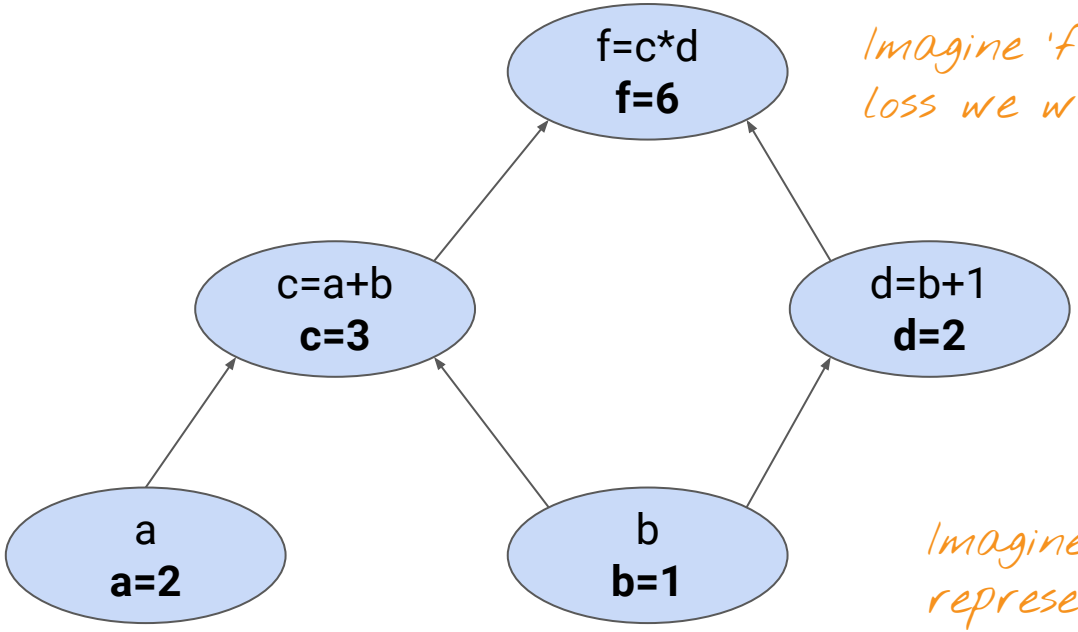
Forward pass



Forward pass



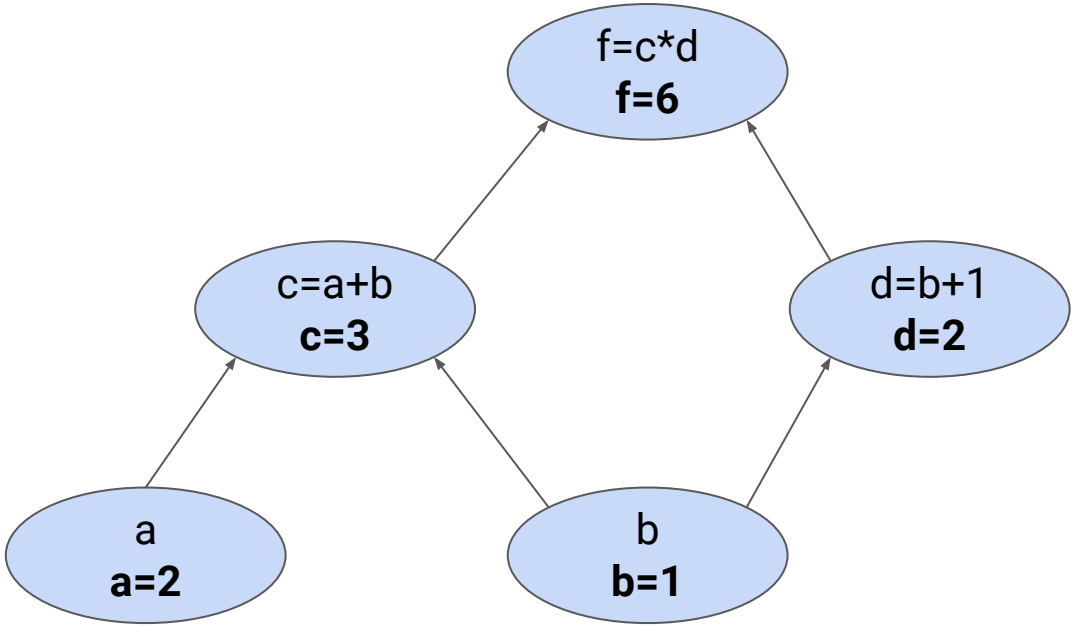
How does adjusting the weights affect the output?



Imagine 'f' represents a loss we want to **minimize**

Imagine 'a' and 'b' represent our **weights.**

The gradient gives us the answer



The gradient of the loss w.r.t. the weights.

$$\nabla_W L = \frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}$$

For example, if we increase 'a' by a little bit, how does this affect L? (Does L increase, or decrease, and at what rate compared to our increase in 'a'?)

Numeric gradient

```
def forward(a, b):
  c = a + b
  d = b + 1
  f = c * d
  return f


forward(a=2, b=1) # 6
```

Define a function for your forward pass.

From hello-backprop.ipynb, on CourseWorks,

```python
def numeric_gradient(f, params, h=1e-4):
  grad = np.zeros_like(params) # Vector of partial derivatives
  for i in range(len(params)): # Loop over weights
    orginal_val = params[i]
    params[i] += h
    plus_h = f(*params) # f(x + h)
    params[i] = orginal_val
    params[i] -= h
    minus_h = f(*params) #f(x - h)
    params[i] = orginal_val # Reset the weight
    grad[i] = (plus_h - minus_h) / (2 * h) # Partial derivative
  return grad
```

This code is computing:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$$

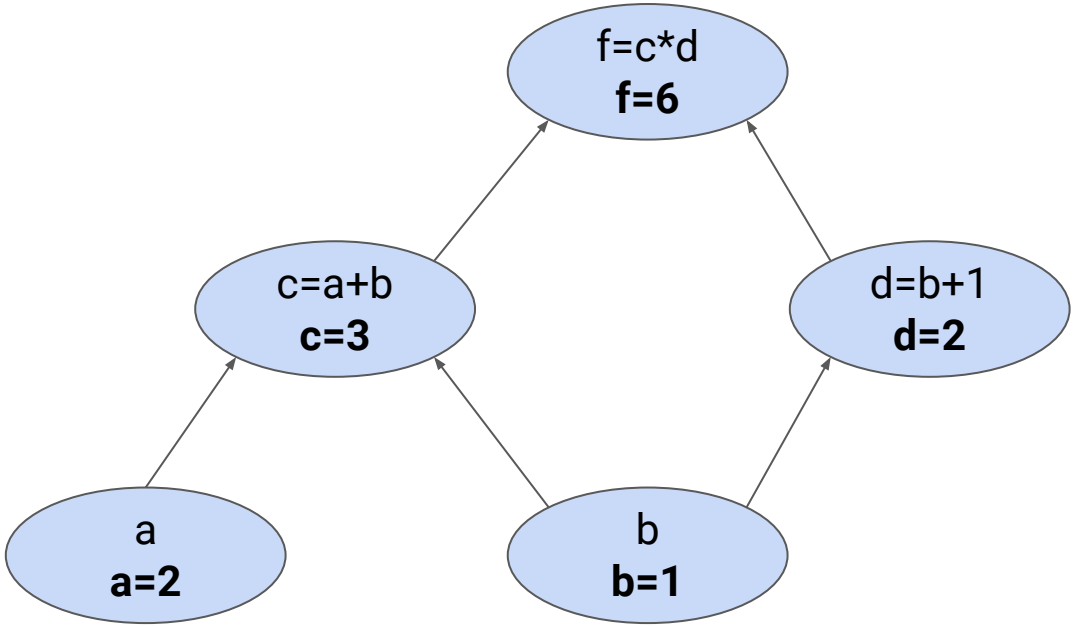
Numeric gradient

```
da, db = numeric_gradient(f=forward, params=[2.0, 1.0])
print ("Numeric gradient. da %0.2f, db %0.2f" % (da, db)) #2.0, #5.0
```

Note: this is the gradient evaluated at a=2, b=1. It will change for other values!

The gradient gives us the answer

$$\nabla_W L = \frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}$$



We got [2, 5].

If we increase a by epsilon, we expect f to increase by 2 * epsilon.

Likewise, if we increase b by epsilon, f should increase by 5 * epsilon.

Complexity of calculating the numerical gradient

Any ideas?

- For example, say we have a tiny network with 1,000 weights.
- How many forward passes do we need to do?

Complexity of calculating the numerical gradient

(For one example)

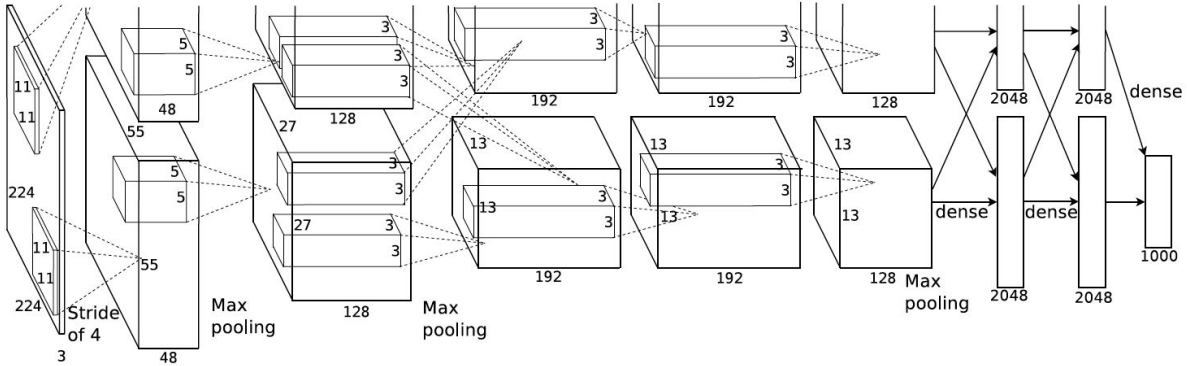
For every weight in the network, we need to:

- Increase its value
- Forward prop, calculate loss
- Decrease its value
- Forward propagate, calculate loss
- Calculate the gradient.

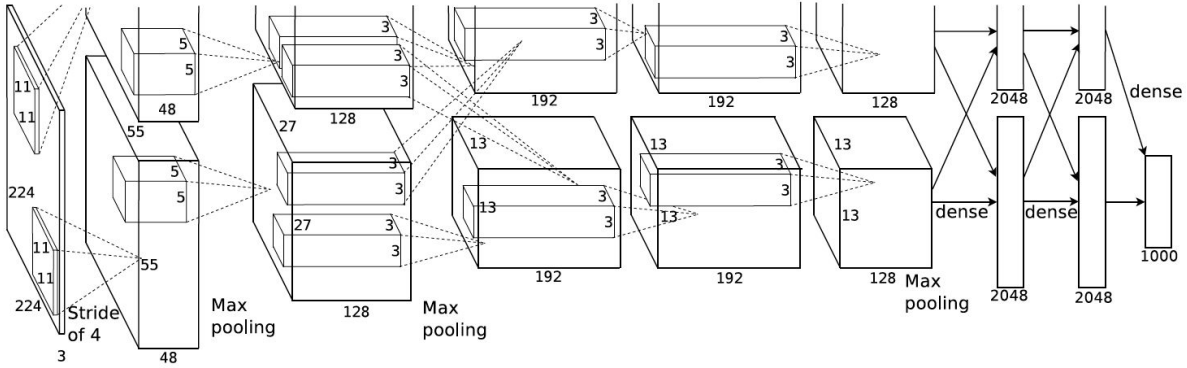
We also need to do this for each weight, one at a time!

Not feasible (but a great way to check your backprop code!)

Complexity of calculating the analytic gradient (by hand?)



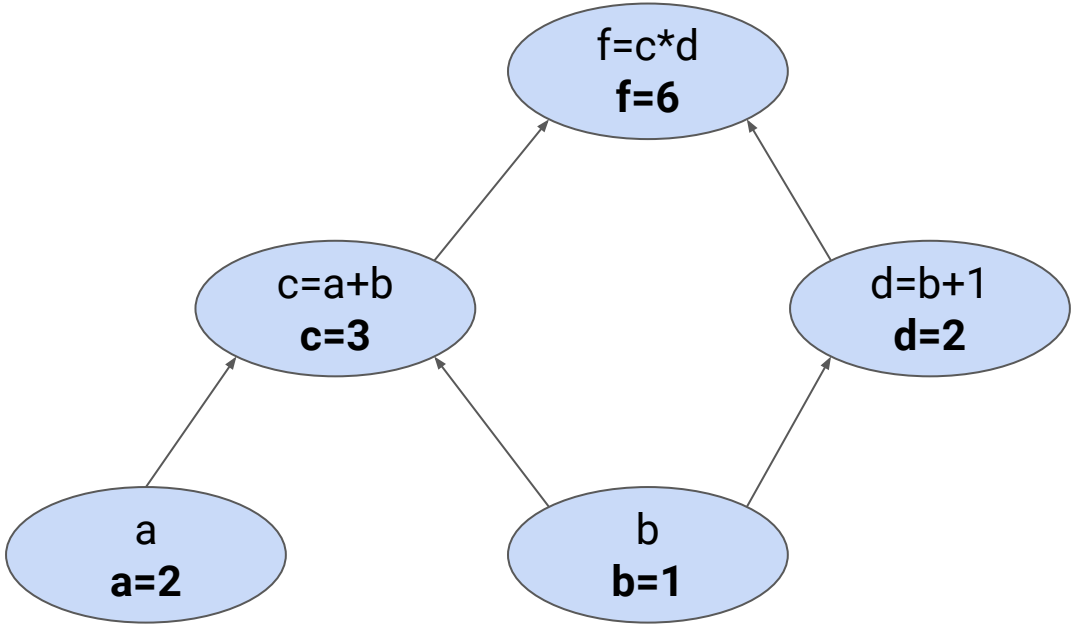
Complexity of calculating the analytic gradient (by hand?)



There are reasons we don't want to do this by hand other than "it's hard". If we have an 'autodiff' implementation in software, we can experiment with new architectures quickly.

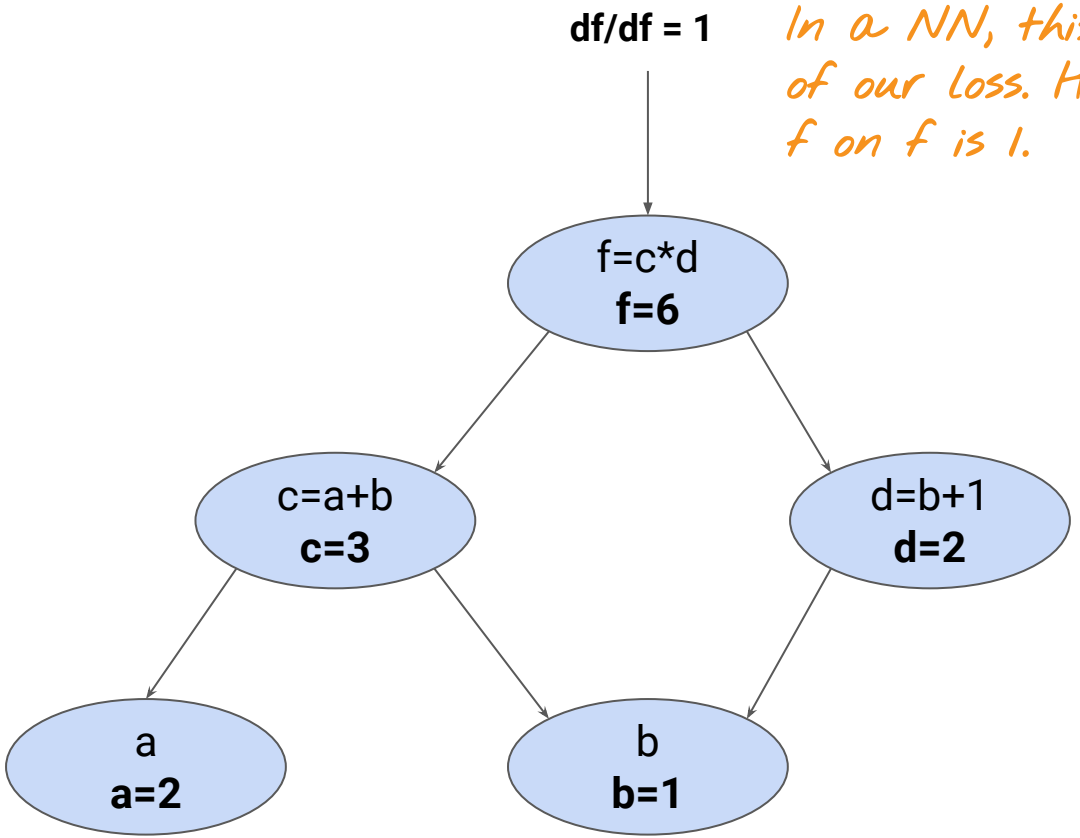
Backprop

Backprop: A method for efficiently computing gradients (by recursive application of the chain rule on a computational graph).



1) Compute the forward pass (finished here).

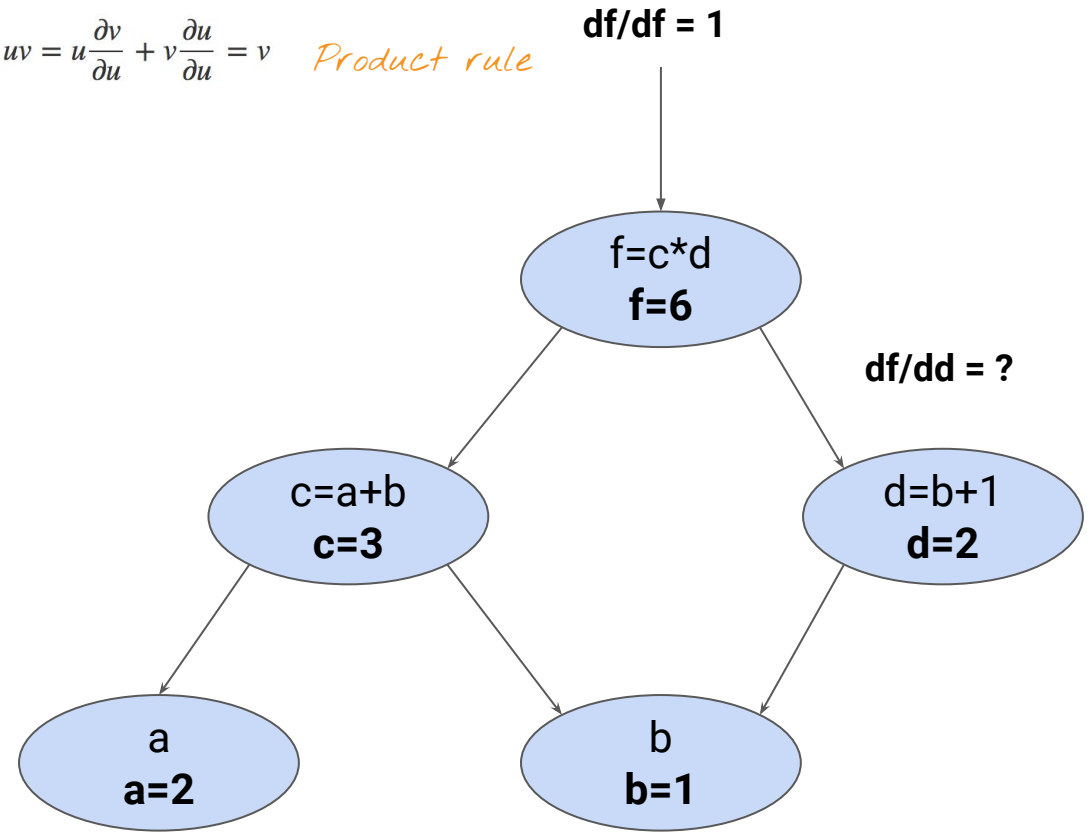
2) Starting from the output, begin propagating gradients backward along **edges in the graph.**



In a NN, this would be the value of our loss. Here, the gradient of f on f is 1.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

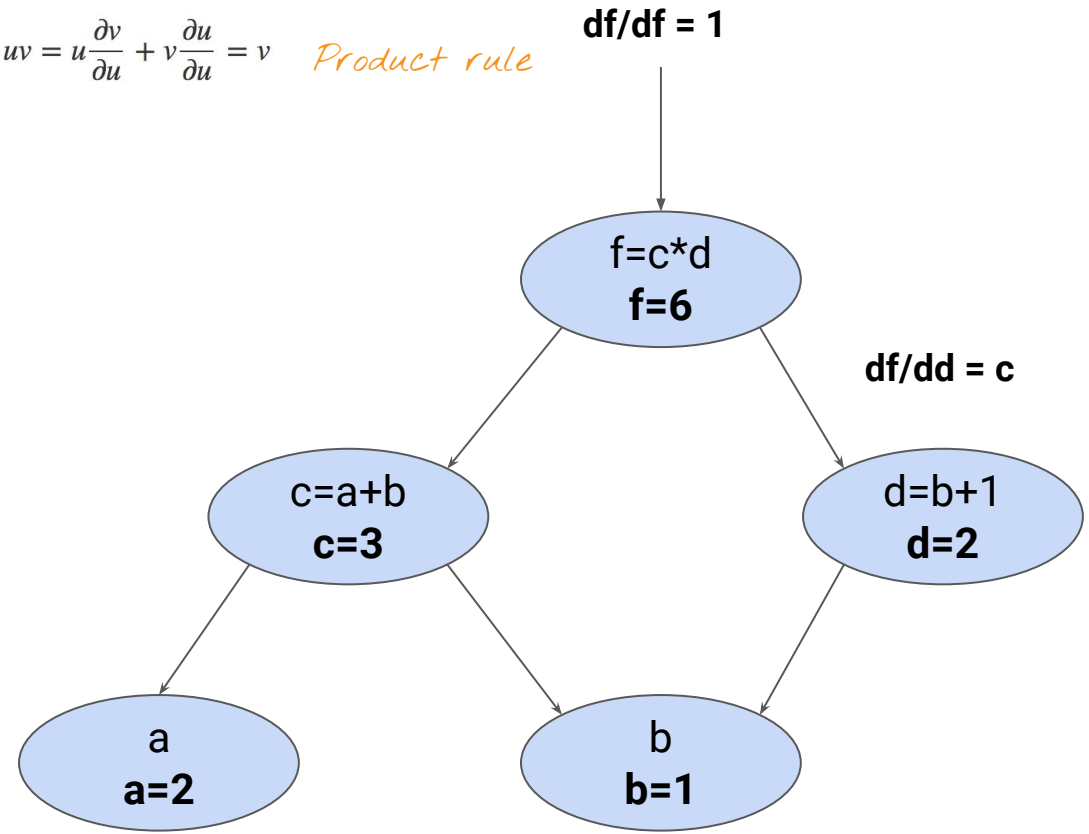
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



If we increase d by a little, then f increases at a rate of ?. So the gradient on this edge is ?.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

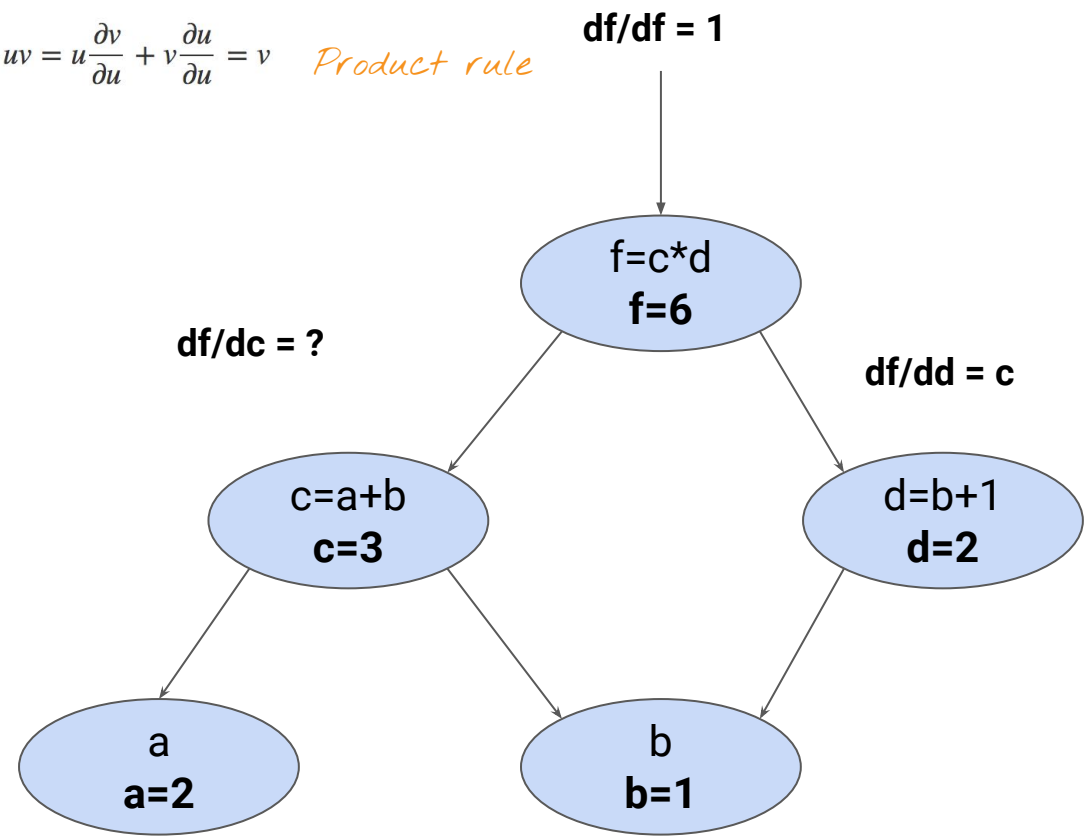
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



Product rule. Intuition: if we increase d by a little, then f increases at a rate of c. So the gradient on this edge is c.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

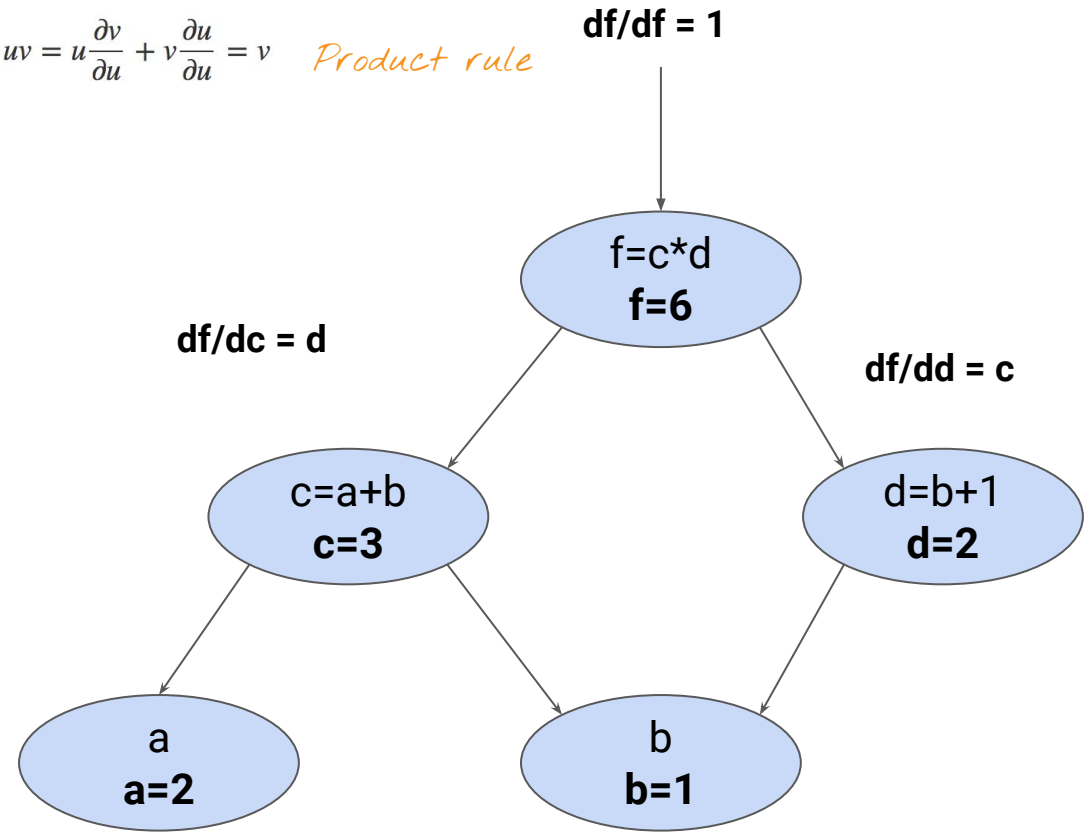
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



If we increase c by a little, then f increases at a rate of ?. So the gradient on this edge is ?.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

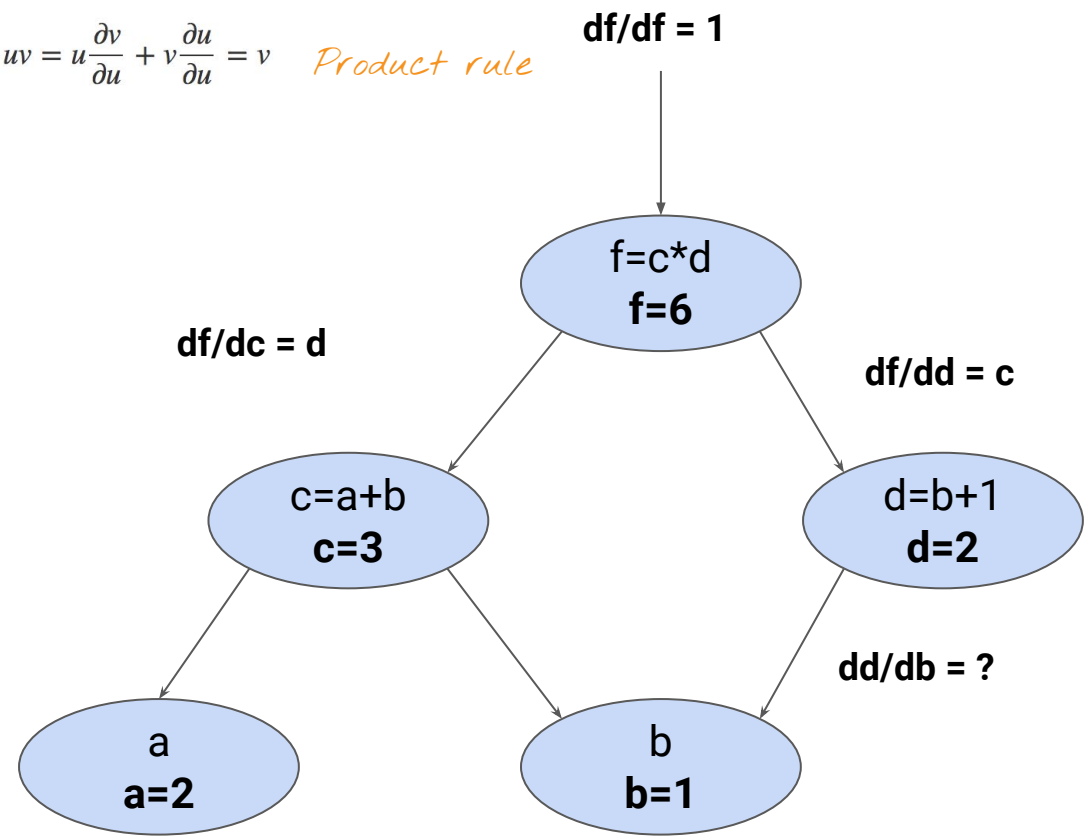
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



Product rule. Intuition: if we increase c by a little, then f increases at a rate of d. So the gradient on this edge is d.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

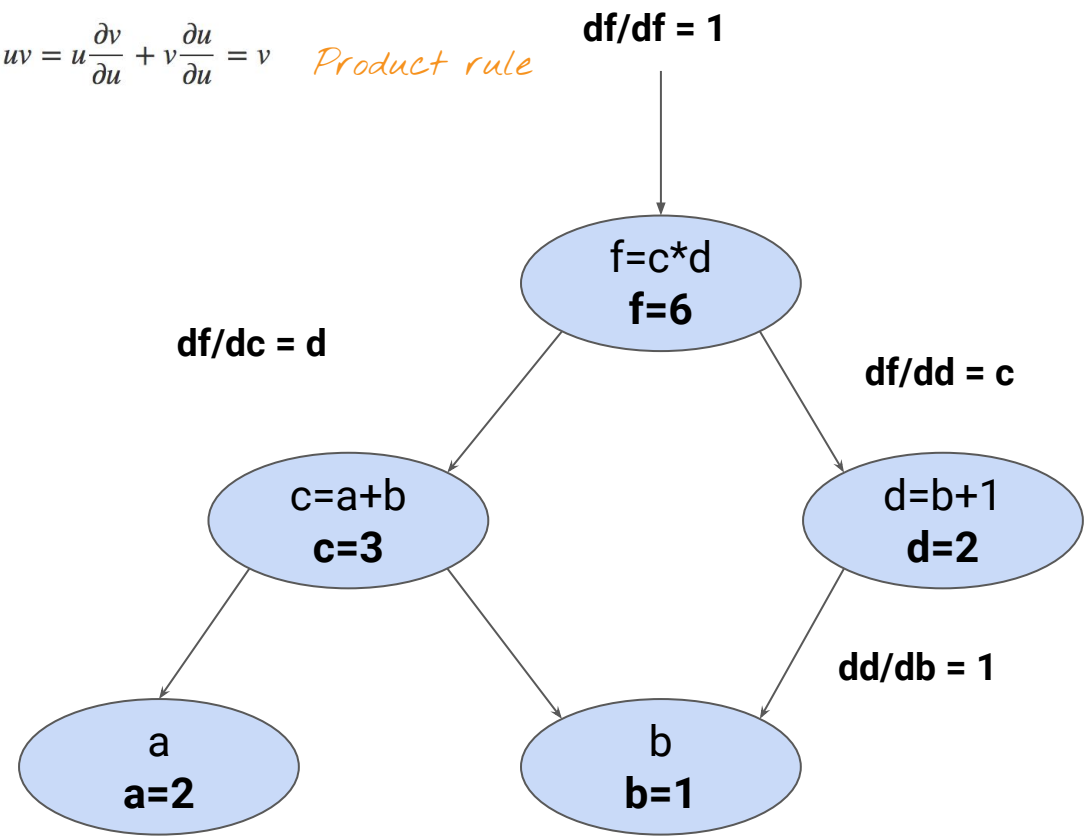
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



Sum rule. Intuition: if we increase b by a little, how much does d change?

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

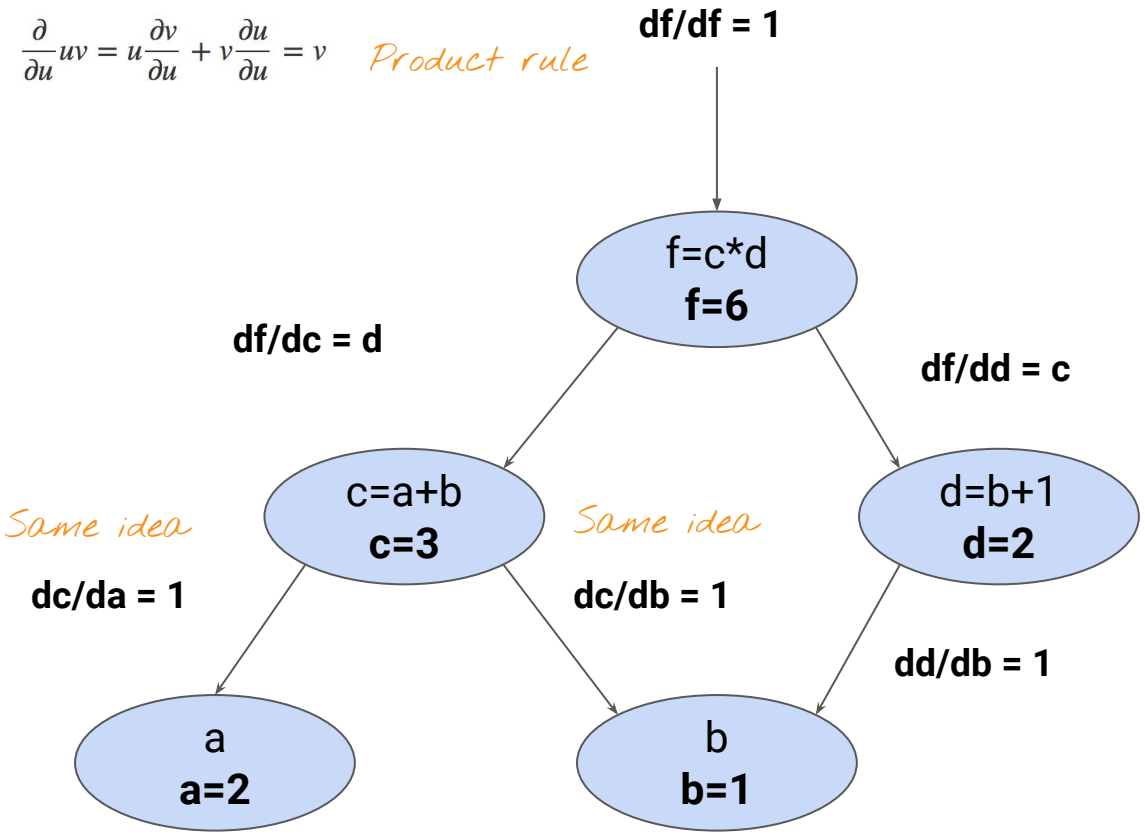
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



Sum rule. Intuition: if we increase b by a little, how much does d change? The same amount.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

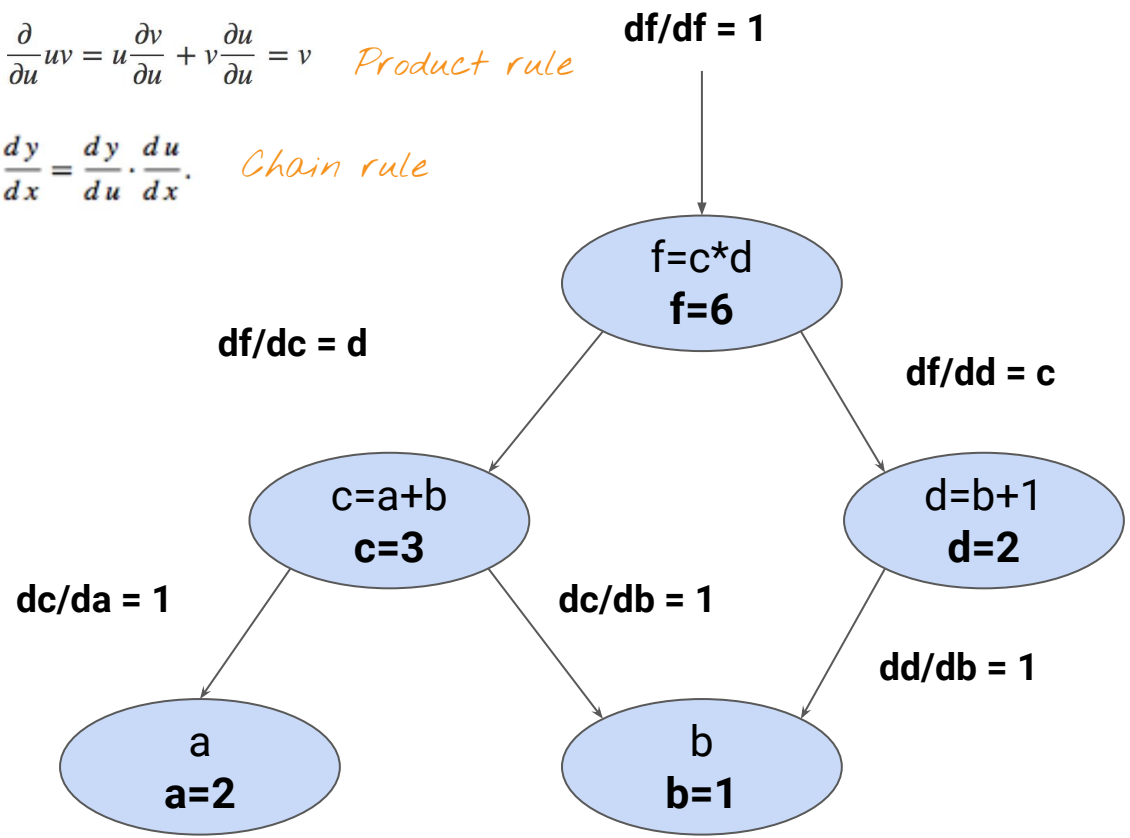


Sum rule. Intuition: if we increase b by a little, how much does d change? The same amount.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule

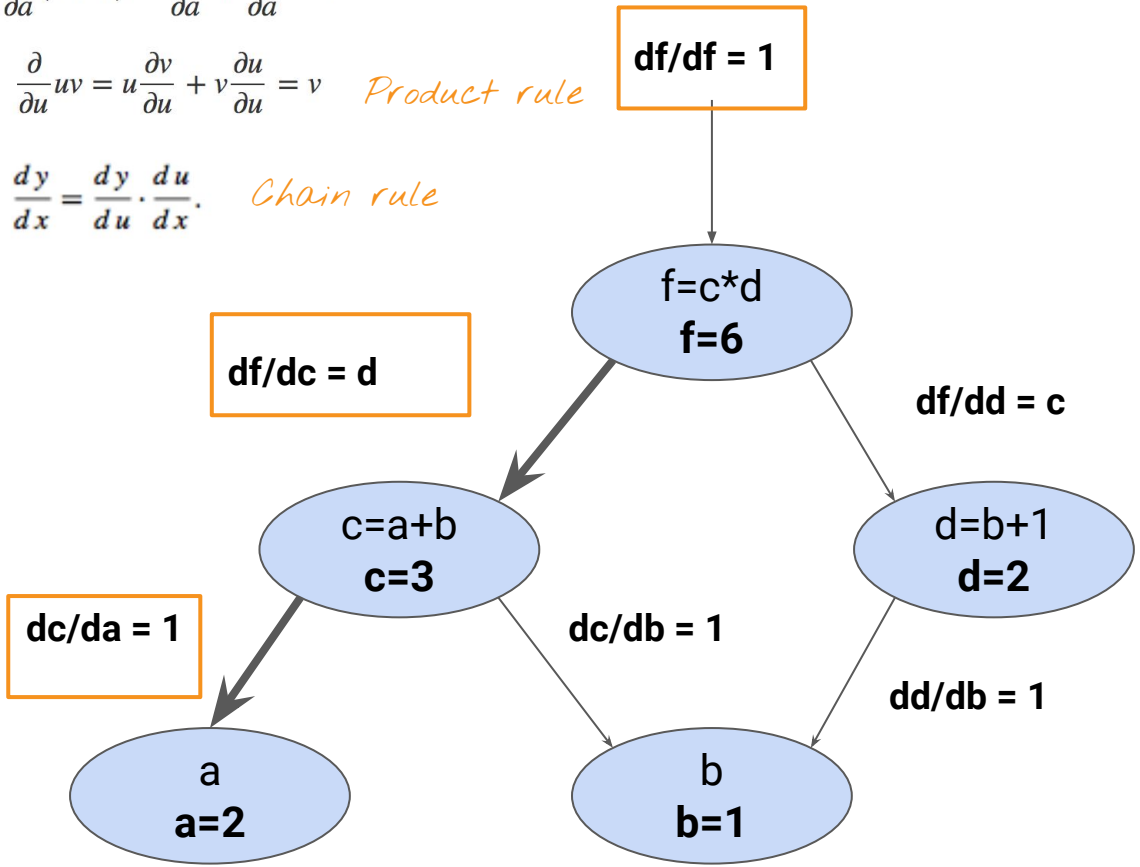


Now we can compute the gradient as the product along paths (another way of thinking about the chain rule!)

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule



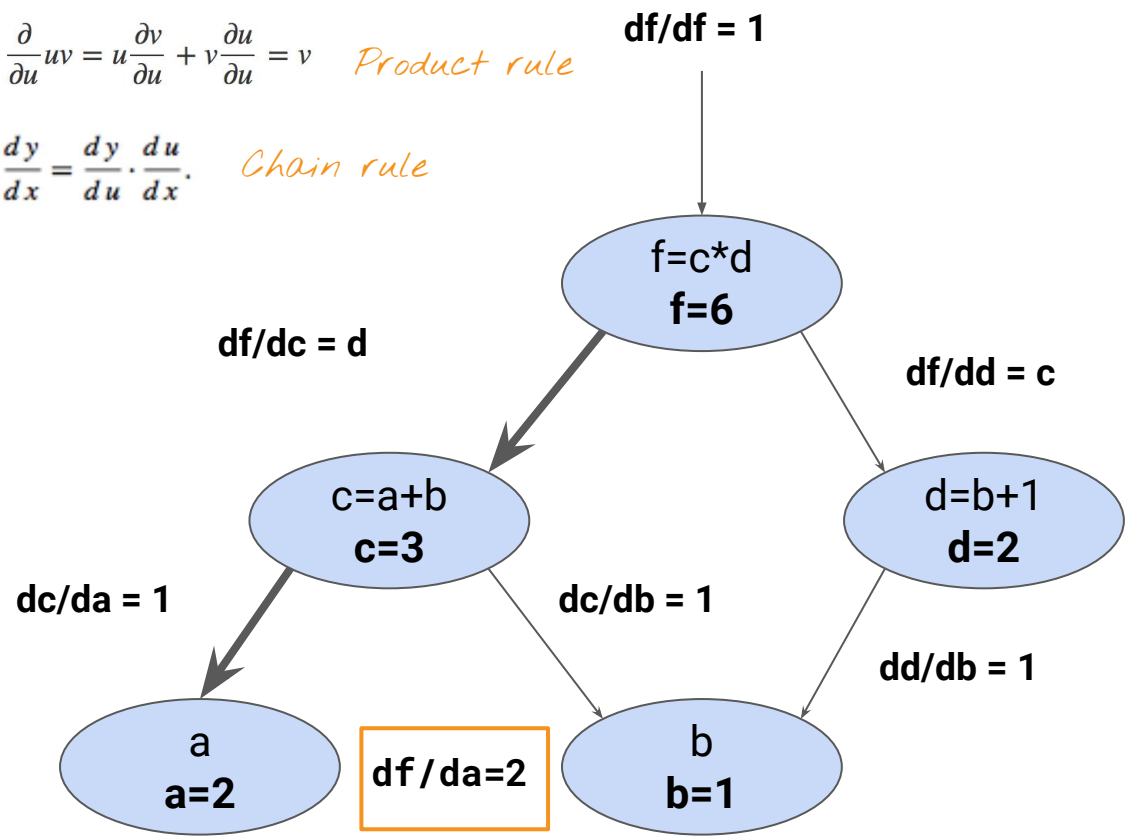
Now we can compute the gradient as the product along paths (another way of thinking about the chain rule!)

df/da = df/df * df/dc * dc/da

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule



Now we can compute the gradient as the product along paths (another way of thinking about the chain rule!)

```
df/da = df/df * df/dc * dc/da
      = 1 * d * 1
      = 1 * 2 * 1
      = 2
```

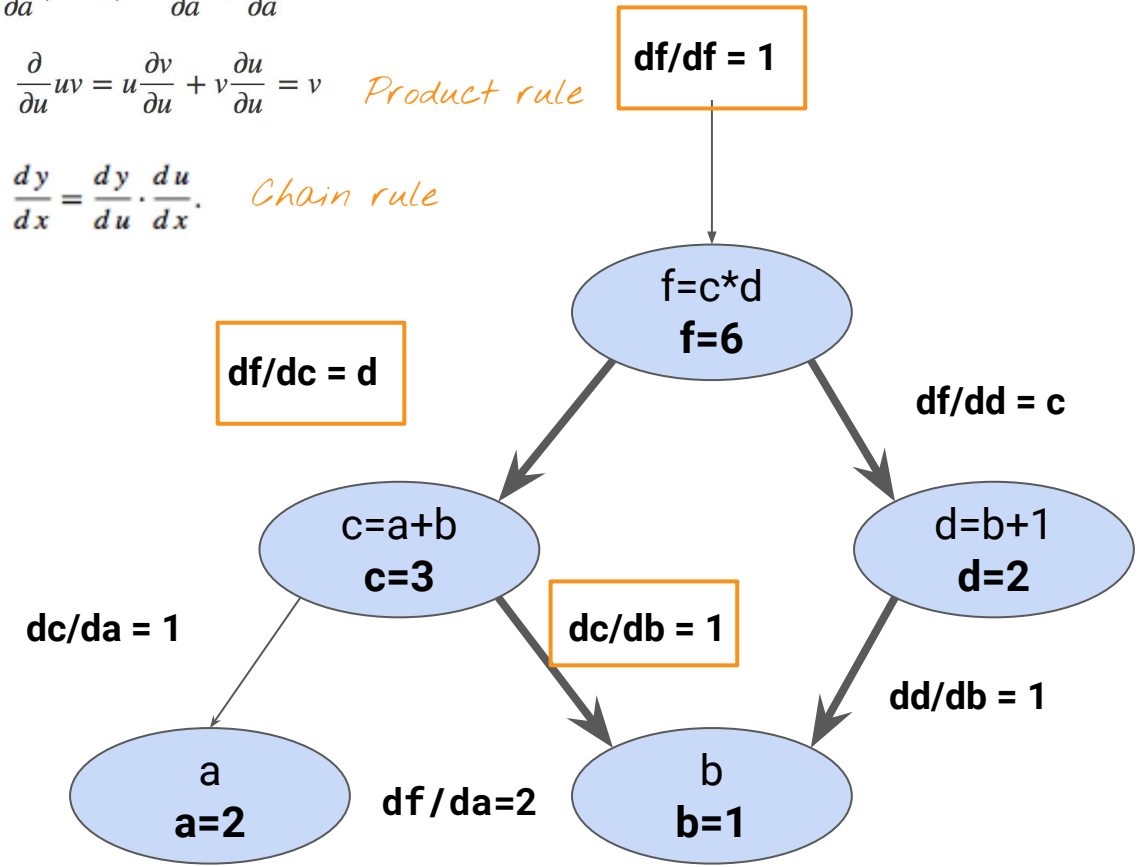
$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule

df/db = df/df * df/dc * dc/db

There are two paths to b.

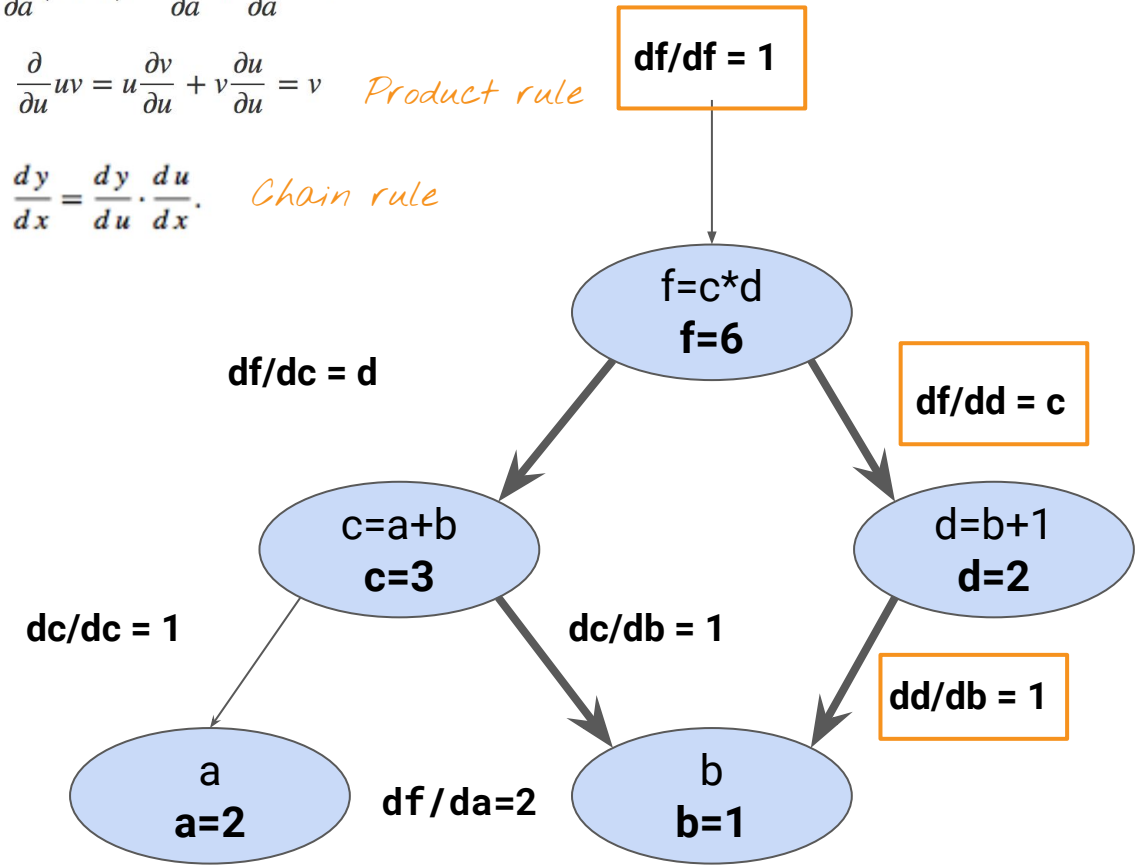


$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule

df/db = df/df * df/dc * dc/db + df/df * df/dd * dd/db



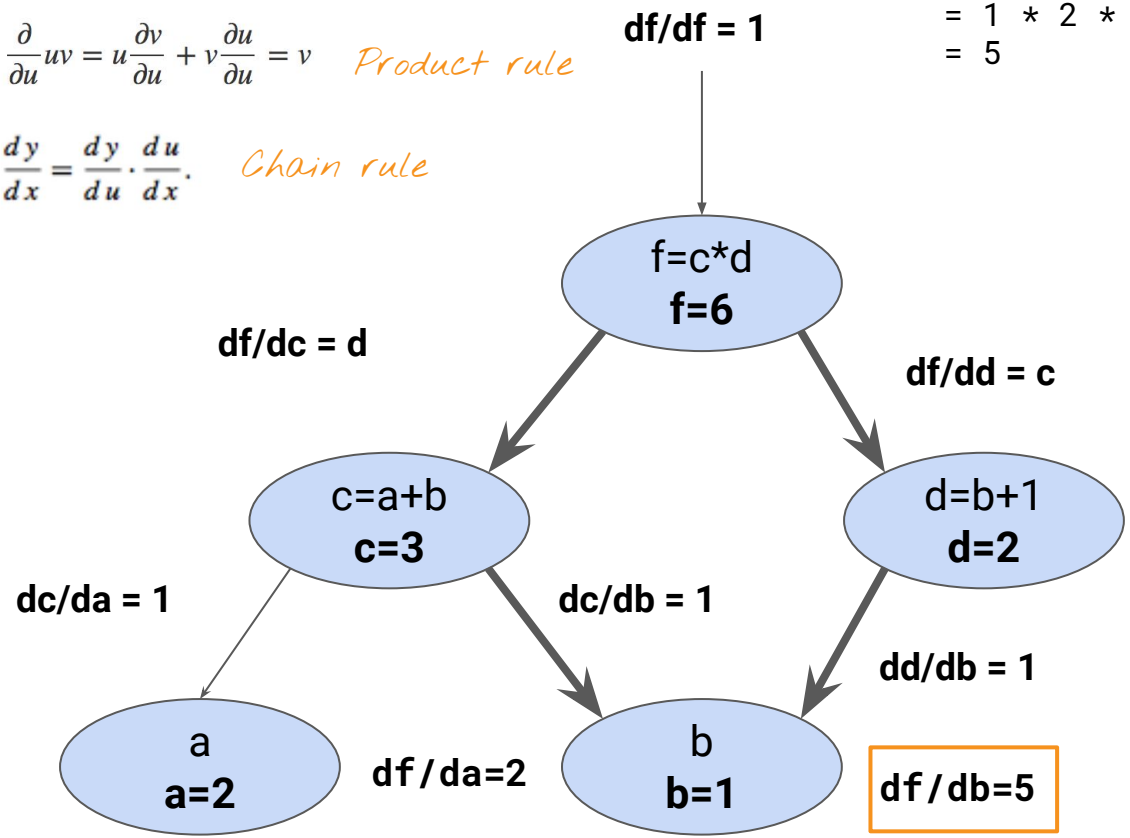
The correct thing to do is sum over them.

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$ Chain rule

```
df/db = df/df * df/dc * dc/db + df/df * df/dd * dd/db
      = 1 * d * 1 + 1 * c * 1
      = 1 * 2 * 1 + 1 * 3 * 1
      = 5
```



Same answer as the numeric gradient!

Complexity of backprop (any ideas?)

Any ideas?

- For example, say we have a tiny network with 1,000 weights.

How many forward passes do we need to do?

Complexity of backprop (any ideas?)

(For one example)

- Forward pass to compute loss.
- Backward pass to compute gradients of **every weight**

Linear in the number of edges on the computational graph(!)

Insight from Chris that's worth repeating

"When I first understood what backpropagation was, my reaction was: "Oh, that's just the chain rule! How did it take us so long to figure out?" I'm not the only one who's had that reaction. It's true that if you ask "is there a smart way to calculate derivatives in feedforward neural networks?" the answer isn't that difficult.

But I think it was much more difficult than it might seem. You see, at the time backpropagation was invented, people weren't very focused on the feedforward neural networks that we study. It also wasn't obvious that derivatives were the right way to train them. Those are only obvious once you realize you can quickly calculate derivatives. There was a circular dependency.

Worse, it would be very easy to write off any piece of the circular dependency as impossible on casual thought. Training neural networks with derivatives? Surely you'd just get stuck in local minima. And obviously it would be expensive to compute all those derivatives. It's only because we know this approach works that we don't immediately start listing reasons it's likely not to.

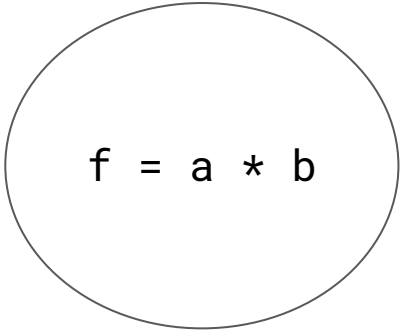
That's the benefit of hindsight. Once you've framed the question, the hardest work is already done."

Summary

- Backprop is a method to efficiently calculate gradients by recursive application of the chain rule on a computation graph.

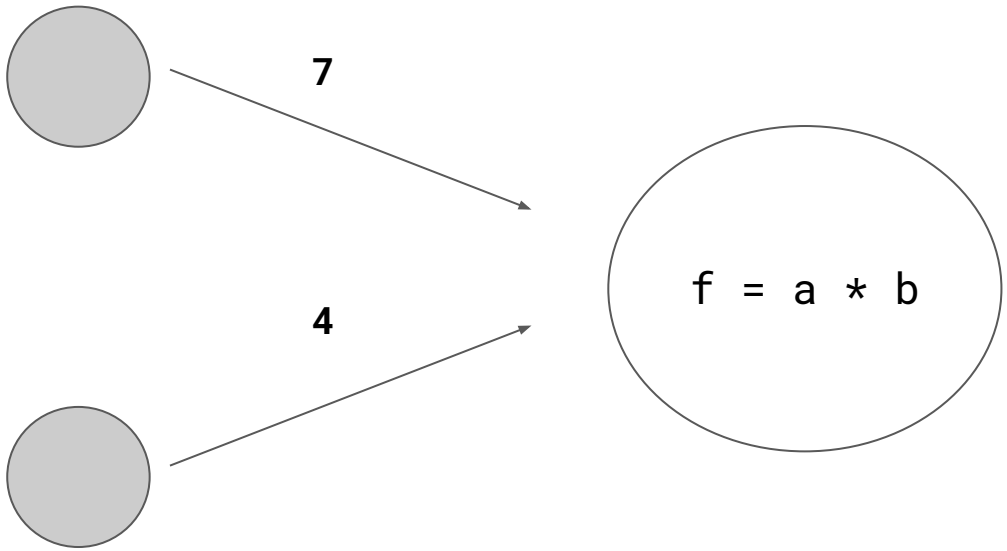
- The key is to realize each node on the graph can calculate its local gradient independently (it doesn't need to know anything about the structure of the graph).

The key is to realize each node on the graph can calculate its local gradient independently



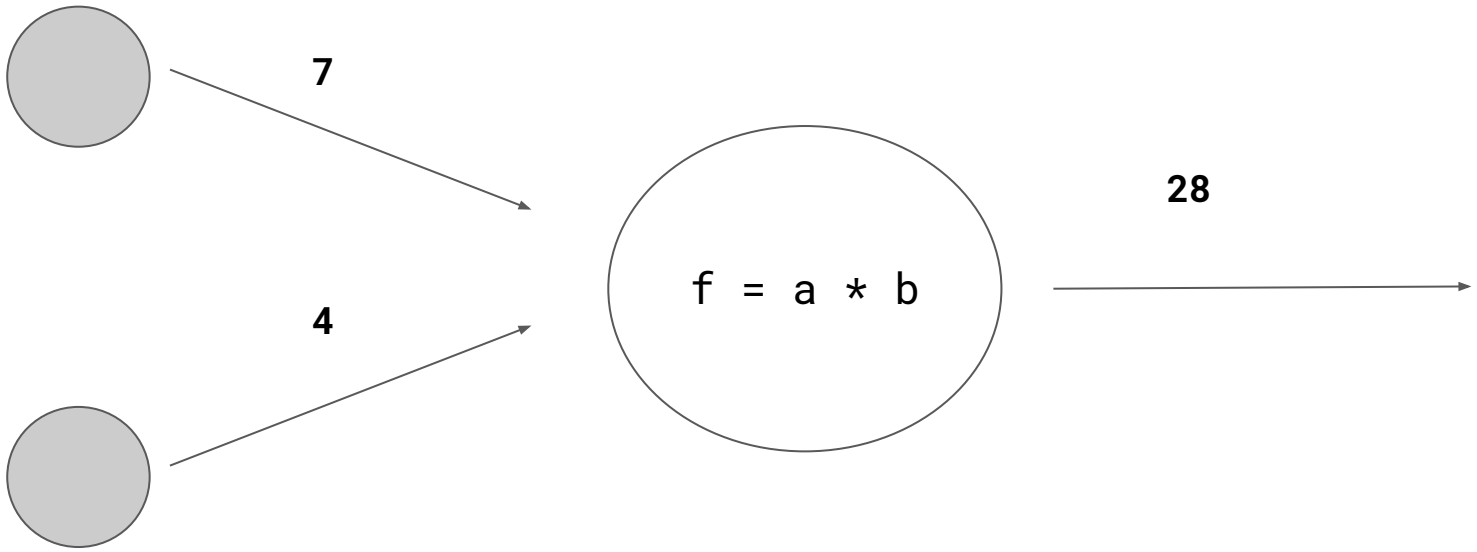
A humble multiplication operation

Knows how to compute its forward pass based on inputs from other nodes in the graph

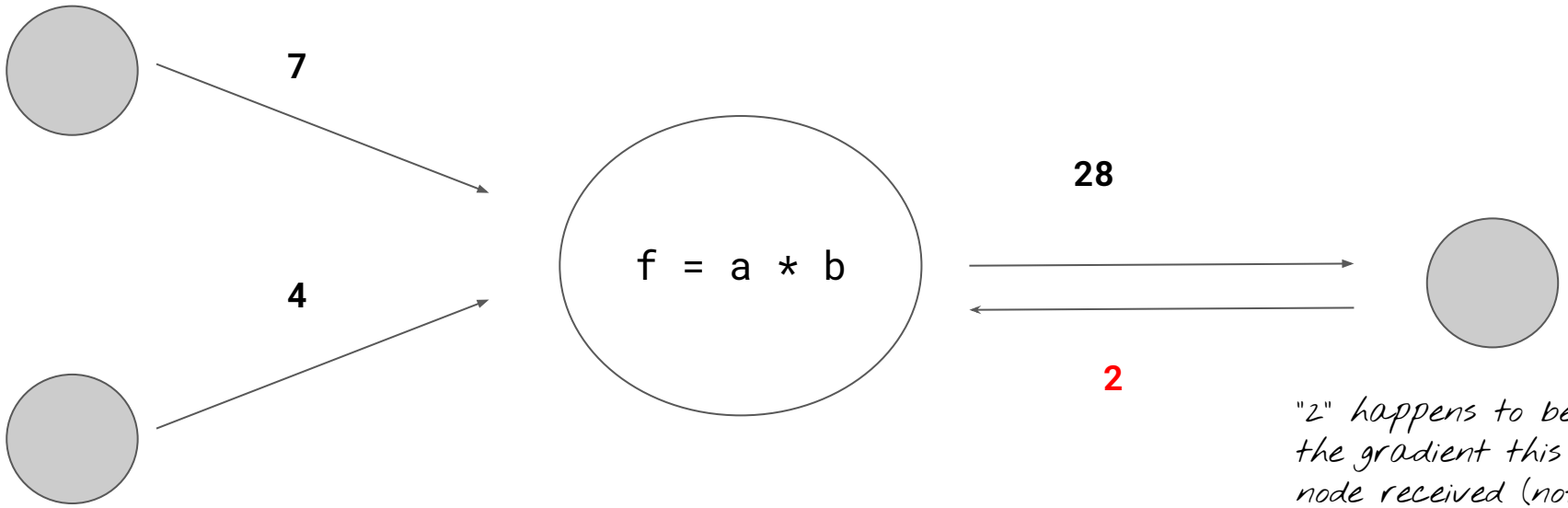


Other nodes (we don't need to know anything about them other than the value they propagate forward).

Knows how to compute its forward pass based on inputs from other nodes in the graph



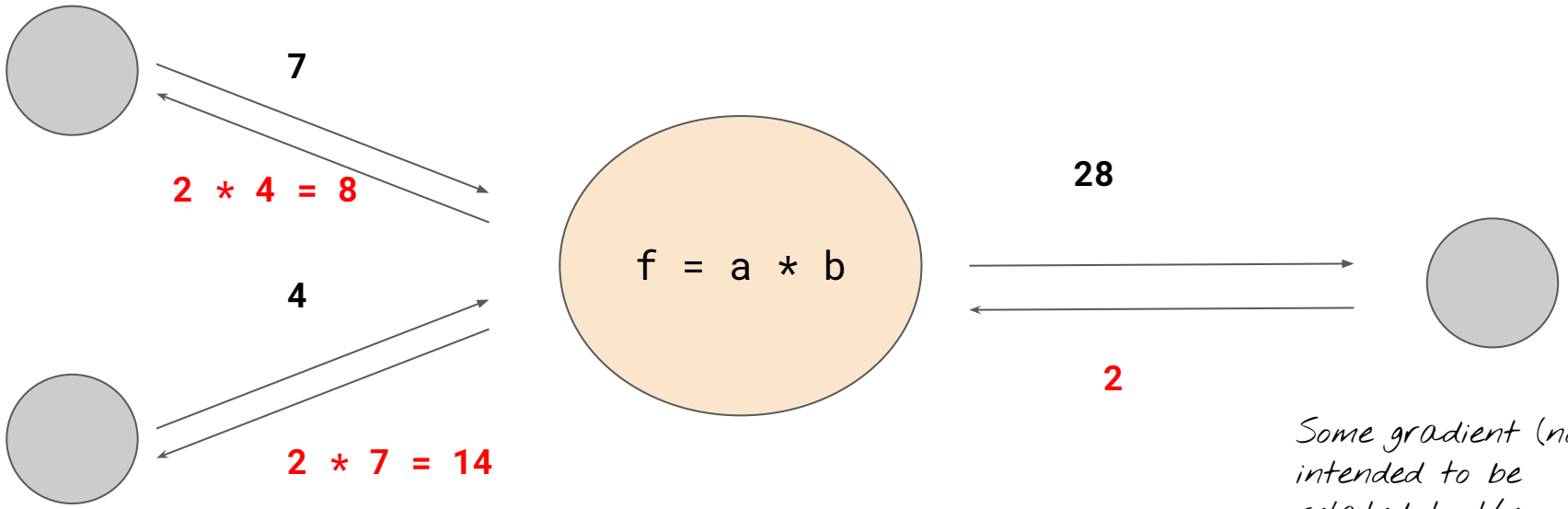
Knows how to distribute gradient it receives in during backward pass.



"2" happens to be the gradient this node received (not relate to the other examples in slides)

Our node doesn't need to know anything about the nodes upstream, only the value of the gradient it receives.

Knows how to distribute gradient it receives in during backward pass.



Some gradient (not intended to be related to the numbers in this example).

Complex functions can be broken down into simpler operations

Every primitive operation (and some complex ones for speed) have gradients defined so they can be used in a computational graph.

Code for three methods of computing the gradient is on our GitHub site.

For next time

Reading

- Deep Learning 4.1, 4.3
- Yes you should understand backprop
- Hacker's guide to Neural Networks which became cs231n (course notes are **amazing**)

Assignments

- **A1 is due today**