Today

Administrative stuff

- Project starter code uploaded
- Backprop example uploaded
- A3 almost uploaded (tonight); no change to due date
- Reminder: ch5 of DLP has lots of helpful code for your projects

Topics

- Gradient descent basics
- Backpropagation basics

News

TensorFlow 2.0 has been released!

- Latest tutorials and guides

Upcoming books I’m excited for

- Deep Learning in JavaScript (bonus: should be familiar)
- TinyML (Pete is a world expert on low-powered ML)

Questions from OH

Administrative stuff

- Will upload custom project submission in a couple weeks
- (Not limited to image classification!)
- You can do anything you like.

HW3 Walkthrough

TensorBoard

- Tool to interactively visualize results (loss curves, histograms, etc) from your experiments.
- Popular with TensorFlow (and PyTorch) users.
- A bit tricky to use / has a startup cost, but may save you energy long term.
- Works inside Colab (previously, only option was to install it on your local machine).

github.com/tensorflow/tensorboard/tree/master/docs/r2
tensorflow.org/tensorboard

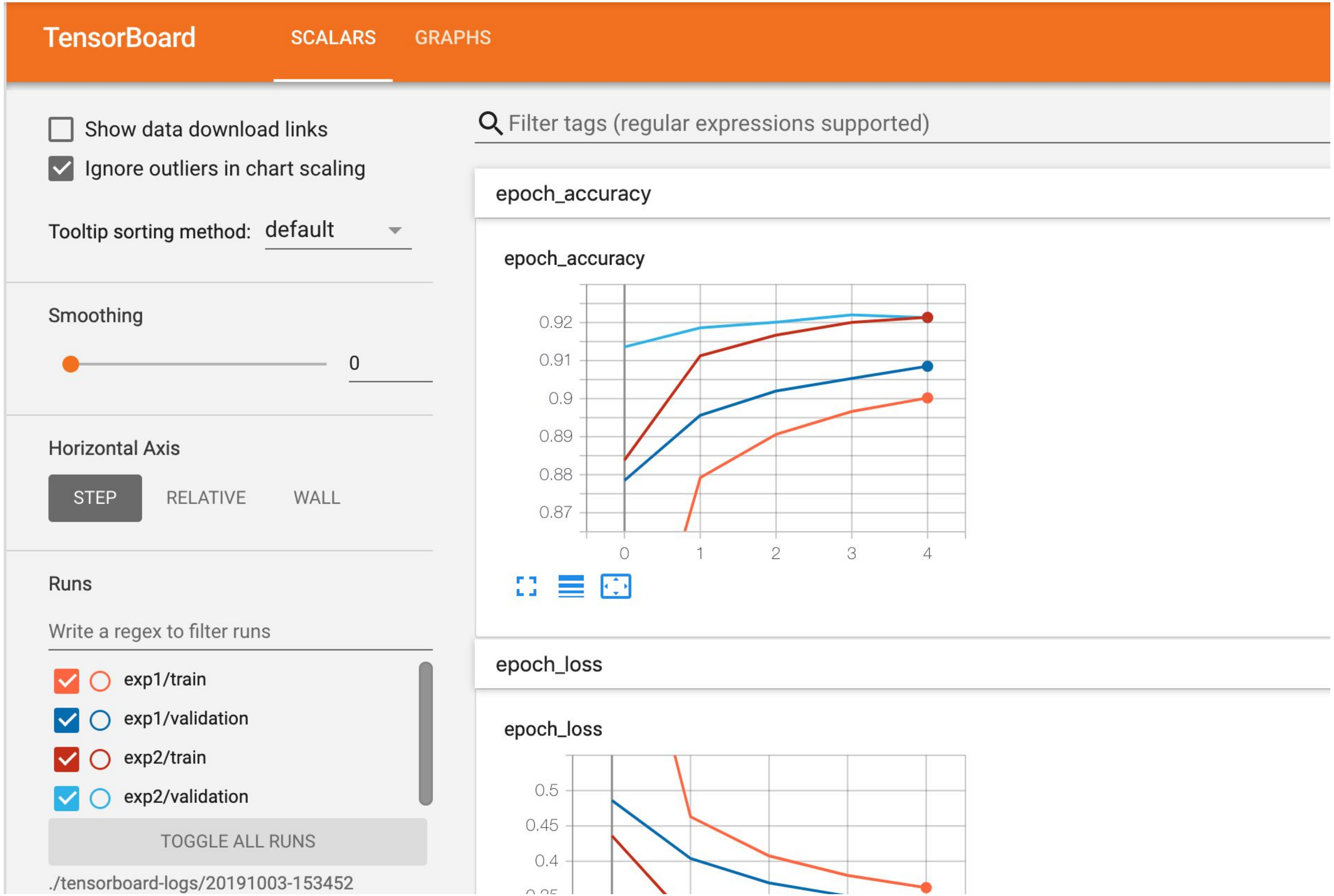
About HW3

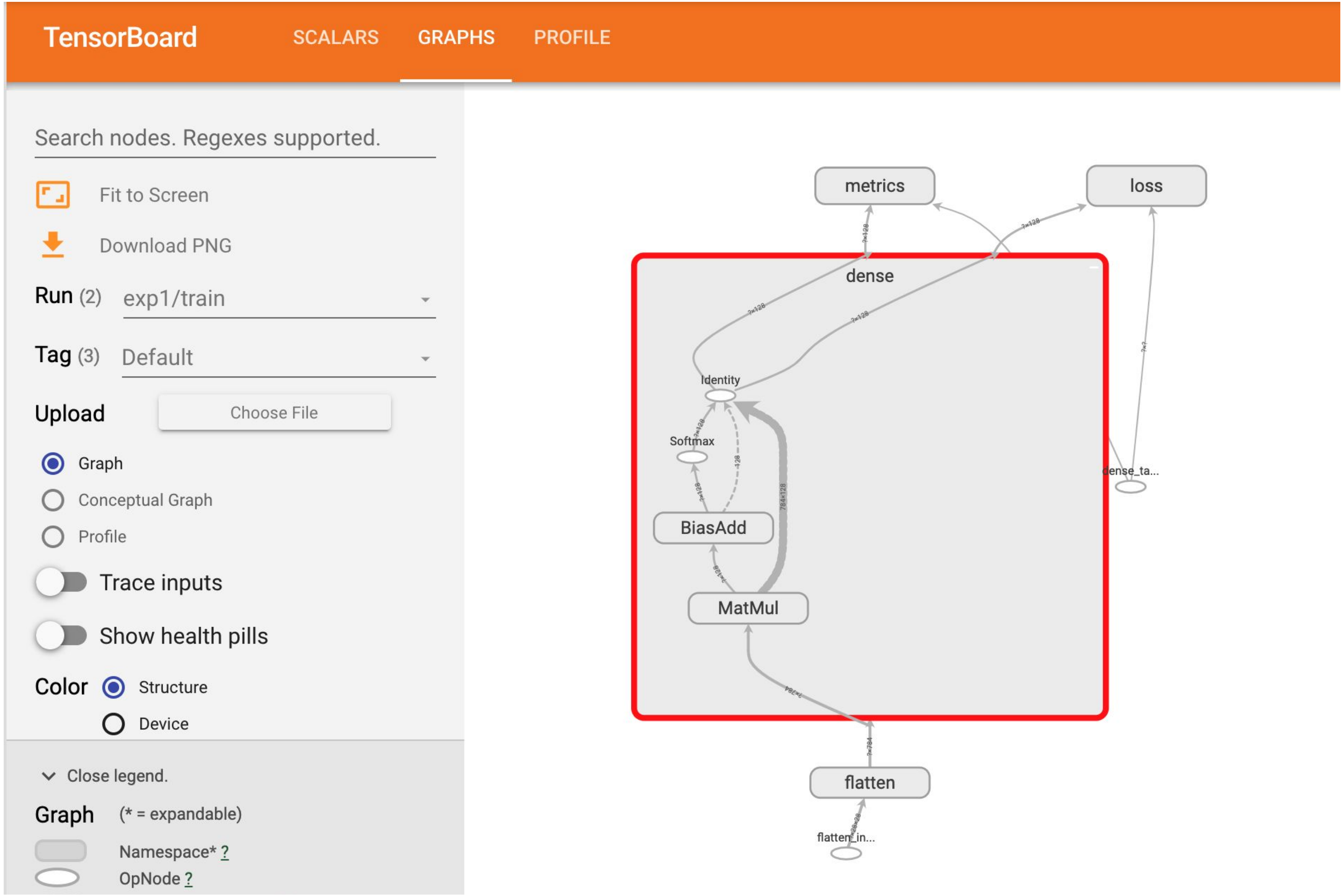
- Design and run experiments to explore a few common hyperparameters (learning rates, activation functions, etc)

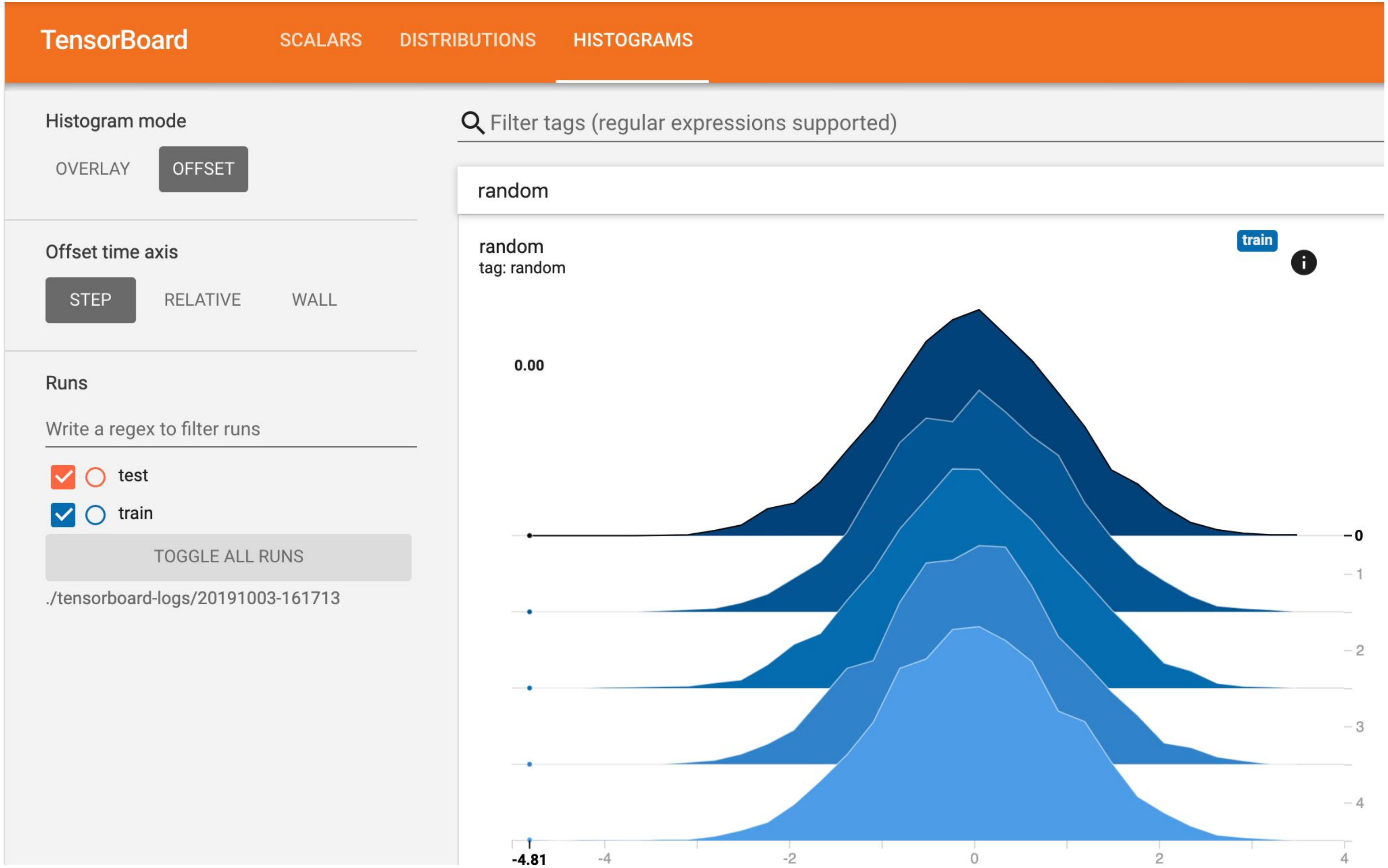
- Visualize the results in TensorBoard

- We have a bunch of reading this week, so a bit shorter than usual to make up for it.

Demo


TensorBoard
SCALARS
GRAPHS
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
exp1/train
exp1/validation
exp2/train
exp2/validation
TOGGLE ALL RUNS
./tensorboard-logs/20191003-153452
Filter tags (regular expressions supported)
epoch_accuracy
epoch_accuracy
0.92
0.91
0.9
0.89
0.88
0.87
0
1
2
3
4
epoch_loss
epoch_loss
0.5
0.45
0.4


TensorBoard
SCALARS
GRAPHS
PROFILE
Search nodes. Regexes supported.
Fit to Screen
Download PNG
Run (2)
exp1/train
Tag (3)
Default
Upload
Choose File
Graph
Conceptual Graph
Profile
Trace inputs
Show health pills
Color
Structure
Device
Close legend.
Graph
(* = expandable)
Namespace* ?
OpNode ?
metrics
loss
dense
Identity
Softmax
BiasAdd
MatMul
dense_ta...
flatten
flatten_in...


TensorBoard
SCALARS
DISTRIBUTIONS
HISTOGRAMS
Histogram mode
OVERLAY
OFFSET
Offset time axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
test
train
TOGGLE ALL RUNS
./tensorboard-logs/20191003-161713
Filter tags (regular expressions supported)
random
random
tag: random
train
0.00
0
1
2
3
4
-4.81
-4
-2
0
2
4

How to restore TensorBoard to a clean state

You will need to delete your logs directory, and kill the TensorBoard process

- Option one: use **Runtime -> Reset all runtimes** (to do both)

- Option two: delete your logs directory, then kill the process
  - !ps (to see processes running)
  - !kill [pid]

Questions from email

“Most of my experience is with sklearn. How do we classify structured data with DL?”

- Great, that’s where you should start. You may find it hard to beat a Random Forest baseline on many datasets.

- Here’s are three examples you can look at in the meantime.

- Note to self: explain feature columns using Facets.

Notes on numerical stability

You can read ater (I meant to include these earlier for you)

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.

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 $-\infty$.**

- Further operations may result in NAN (not a number).

Example

```
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

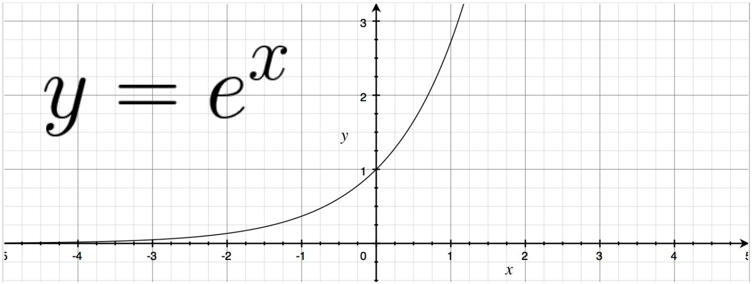
```
>>> import numpy as np
>>> np.exp(1000) / np.exp(1000)
nan

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

```python
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?

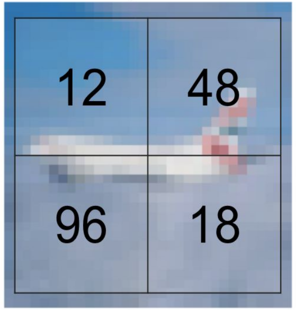
The "." is NumPy telling us this is a floating point value.

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

Underflow

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

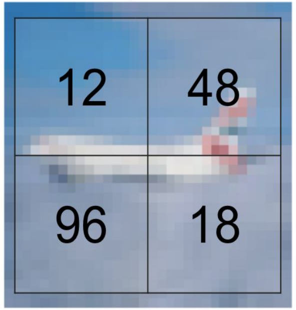
Recall: input to softmax is scores (Wx + b)



$$\underbrace{\begin{bmatrix} 1.4 & 0.5 & 0.7 & 1.2 \\ -2.0 & 0.1 & 0.2 & -0.7 \\ 0.2 & 0.9 & -0.2 & 0.5 \end{bmatrix}}_{\substack{W \\ \text{Weights}}} \underbrace{\begin{bmatrix} 12 \\ 48 \\ 96 \\ 18 \end{bmatrix}}_{\substack{x \\ \text{Inputs}}} + \underbrace{\begin{bmatrix} 0.5 \\ 1.2 \\ 0.2 \end{bmatrix}}_{\substack{b \\ \text{Bias}}} = \underbrace{\begin{bmatrix} 130.1 & \text{Plane} \\ -11.4 & \text{Car} \\ 12.8 & \text{Truck} \end{bmatrix}}_{\substack{\text{Output} \\ \text{Scores}}}$$

Quick discussion: Example that will cause overflow / underflow?

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



$$\underbrace{\begin{bmatrix} 1.4 & 0.5 & 0.7 & 1.2 \\ -2.0 & 0.1 & 0.2 & -0.7 \\ 0.2 & 0.9 & -0.2 & 0.5 \end{bmatrix}}_{\substack{W \\ \text{Weights}}} \underbrace{\begin{bmatrix} 12 \\ 48 \\ 96 \\ 18 \end{bmatrix}}_{\substack{x \\ \text{Inputs}}} + \underbrace{\begin{bmatrix} 0.5 \\ 1.2 \\ 0.2 \end{bmatrix}}_{\substack{b \\ \text{Bias}}} = \underbrace{\begin{bmatrix} 130.1 & \text{Plane} \\ -11.4 & \text{Car} \\ 12.8 & \text{Truck} \end{bmatrix}}_{\substack{\text{Output} \\ \text{Scores}}}$$

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

Underflow: Wx is very small (many weights / pixels close to zero)

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 0) 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).

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

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

```
>>> 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

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)

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

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)

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

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 now know the answer to.

Related concepts

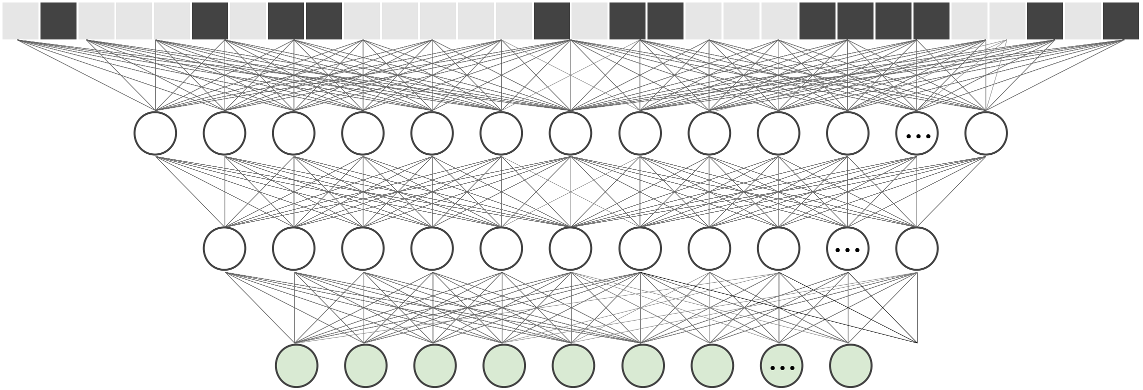
Why normalize input data?

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

Minimize (or maximize) a function iteratively.

How to find useful values for millions of weights?

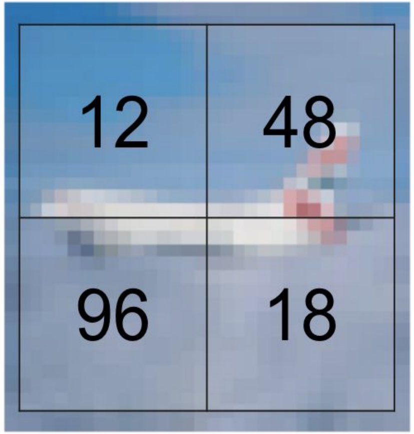


The fact that this is solvable is only apparent in retrospect. Not obvious that gradient descent was the way to go. Why? We learn lots of reasons why it probably won't work in a calculus class: e.g. local minimums would lead to a suboptimal solution. Not to mention, optimization by gradient descent often didn't work well in practice (trouble training deep networks). Empirically, you don't need a perfect solution, just a useful one.

You're not limited to calculating gradients of weights

Recall: Loss is a function of weights and inputs

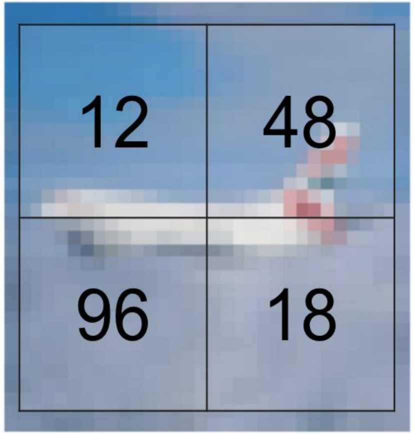
These are the parameters we can adjust



$$\underbrace{\begin{bmatrix} 1.4 & 0.5 & 0.7 & 1.2 \\ -2.0 & 0.1 & 0.2 & -0.7 \\ 0.2 & 0.9 & -0.2 & 0.5 \end{bmatrix}}_{\substack{W \\ \text{Weights}}} \underbrace{\begin{bmatrix} 12 \\ 48 \\ 96 \\ 18 \end{bmatrix}}_{\substack{x \\ \text{Inputs}}} + \underbrace{\begin{bmatrix} 0.5 \\ 1.2 \\ 0.2 \end{bmatrix}}_{\substack{b \\ \text{Bias}}} = \underbrace{\begin{bmatrix} 130.1 & \text{Plane} \\ -11.4 & \text{Car} \\ 12.8 & \text{Truck} \end{bmatrix}}_{\substack{\text{Output} \\ \text{Scores}}}$$

Recall: Loss is a function of weights and inputs

These are the parameters we can adjust

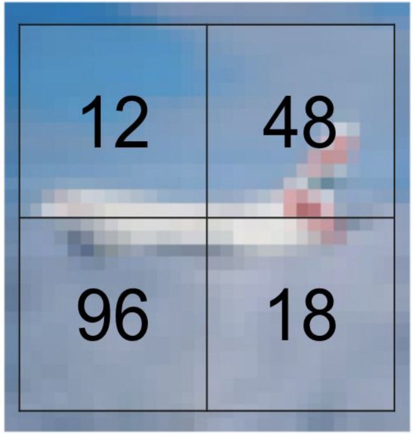


$$\underbrace{\begin{bmatrix} 1.4 & 0.5 & 0.7 & 1.2 \\ -2.0 & 0.1 & 0.2 & -0.7 \\ 0.2 & 0.9 & -0.2 & 0.5 \end{bmatrix}}_{\substack{W \\ \text{Weights}}} \underbrace{\begin{bmatrix} 12 \\ 48 \\ 96 \\ 18 \end{bmatrix}}_{\substack{x \\ \text{Inputs}}} + \underbrace{\begin{bmatrix} 0.5 \\ 1.2 \\ 0.2 \end{bmatrix}}_{\substack{b \\ \text{Bias}}} = \underbrace{\begin{array}{cc} \begin{bmatrix} 130.1 \\ -11.4 \\ 12.8 \end{bmatrix} & \begin{array}{l} \text{Plane} \\ \text{Car} \\ \text{Truck} \end{array} \end{array}}_{\substack{\text{Output} \\ \text{Scores}}}$$

Can’t change the inputs

Recall: Loss is a function of weights and inputs

These are the parameters we can adjust



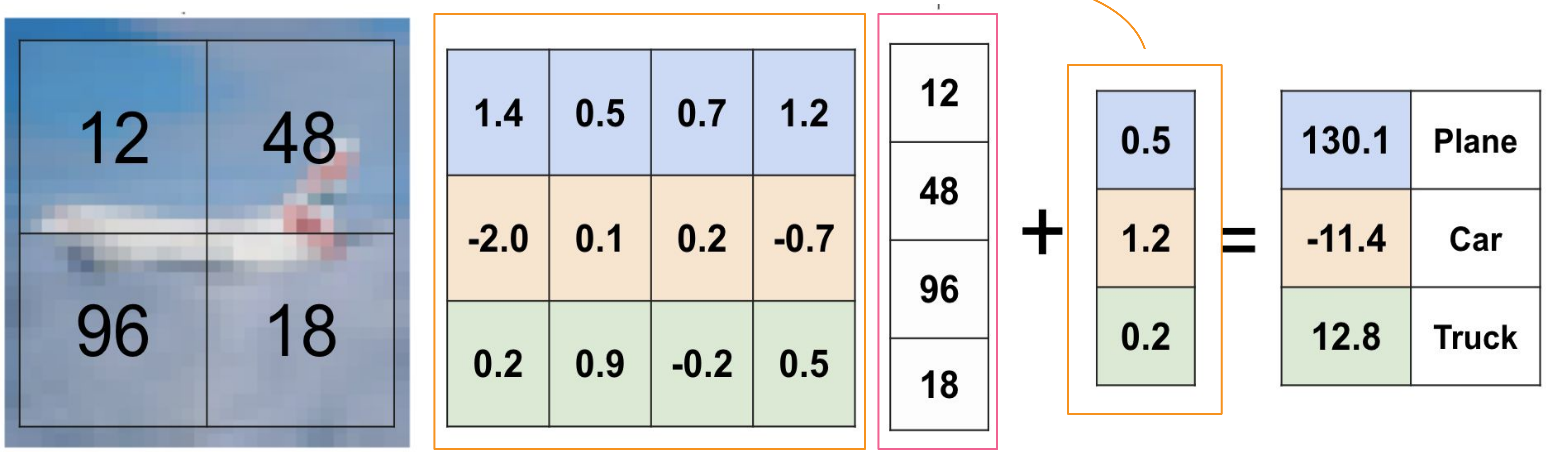
$$\underbrace{\begin{bmatrix} 1.4 & 0.5 & 0.7 & 1.2 \\ -2.0 & 0.1 & 0.2 & -0.7 \\ 0.2 & 0.9 & -0.2 & 0.5 \end{bmatrix}}_{\substack{W \\ \text{Weights}}} \underbrace{\begin{bmatrix} 12 \\ 48 \\ 96 \\ 18 \end{bmatrix}}_{\substack{x \\ \text{Inputs}}} + \underbrace{\begin{bmatrix} 0.5 \\ 1.2 \\ 0.2 \end{bmatrix}}_{\substack{b \\ \text{Bias}}} = \underbrace{\begin{bmatrix} 130.1 & \text{Plane} \\ -11.4 & \text{Car} \\ 12.8 & \text{Truck} \end{bmatrix}}_{\substack{\text{Output} \\ \text{Scores}}}$$

~~Can't change the inputs~~

Unless we're doing deep dream, or style transfer. In which case, we leave a pretrained network unchanged, and adjust the input image.

Recall: Loss is a function of weights and inputs

These are the parameters we can adjust



~~Can't change the inputs~~
Unless we're doing deep dream, or style transfer. In which case, we leave a pretrained network unchanged, and adjust the input image.

W
Weights

x
Inputs

b
Bias

Output
Scores

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

Four strategies

Four strategies

- **Random guess** (randomly guess values for weights)

Gradient descent. How do we get the gradient?

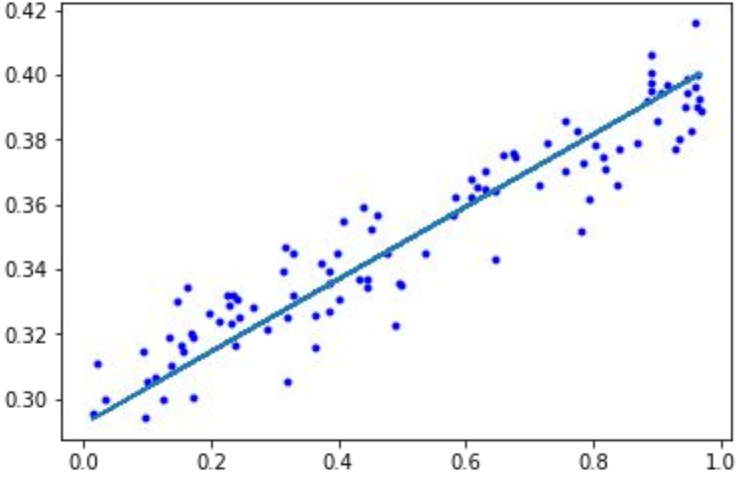
- **Numeric** (find gradient by nudging weights and forwarding the function, rinse, repeat).

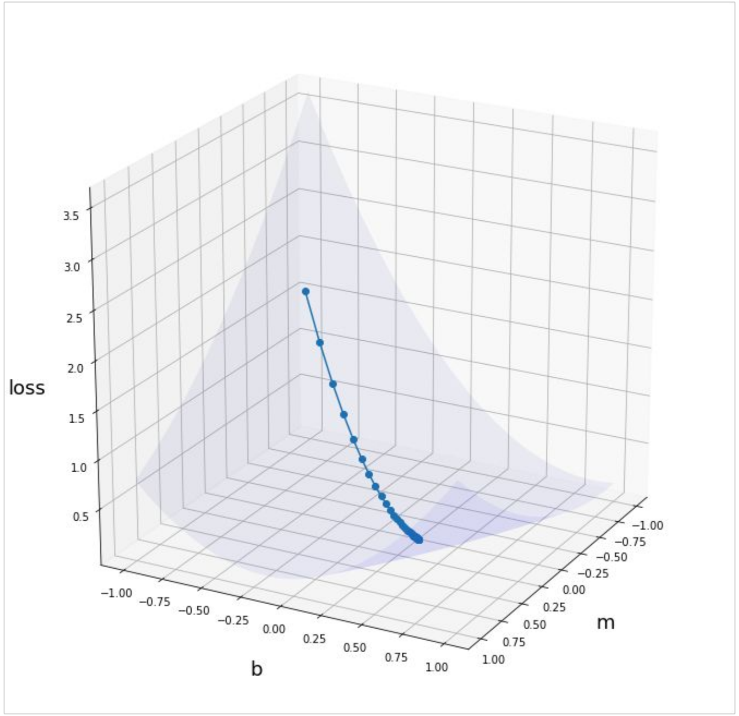
- **Analytic** (find gradient by using your knowledge of calculus).

- **Backprop** (find gradient algorithmically).

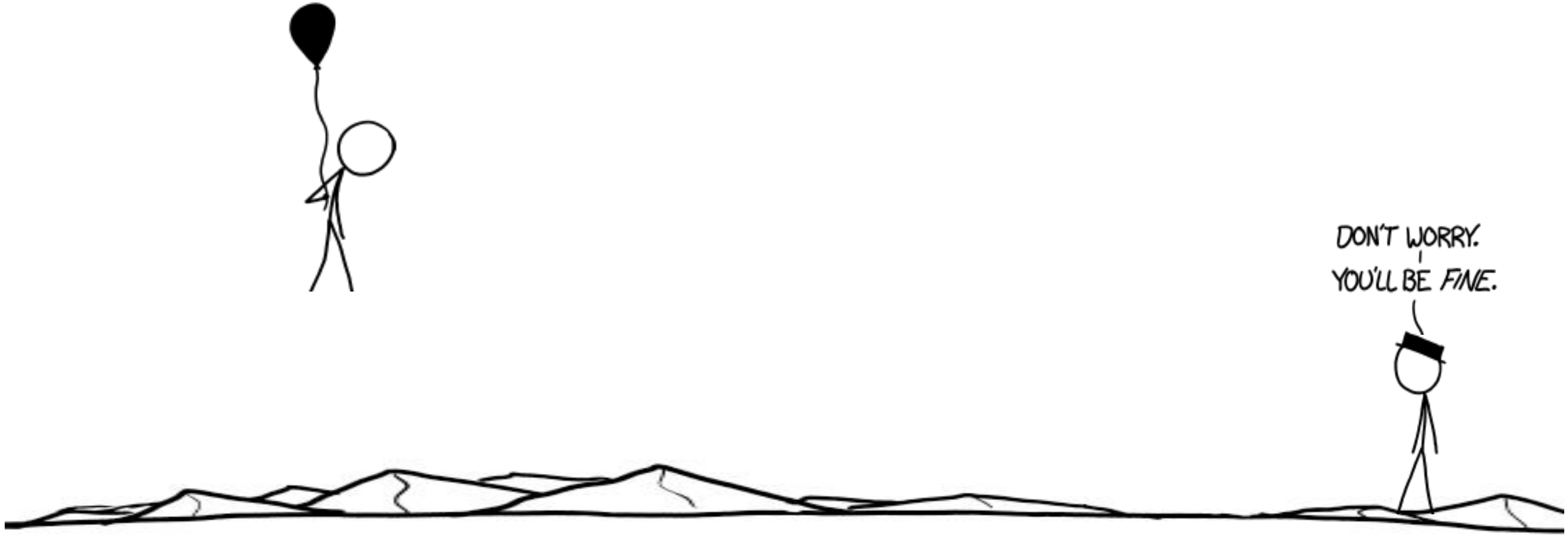
Basics, assuming you have the gradient

Worked example in the linear regression notebook on courseworks





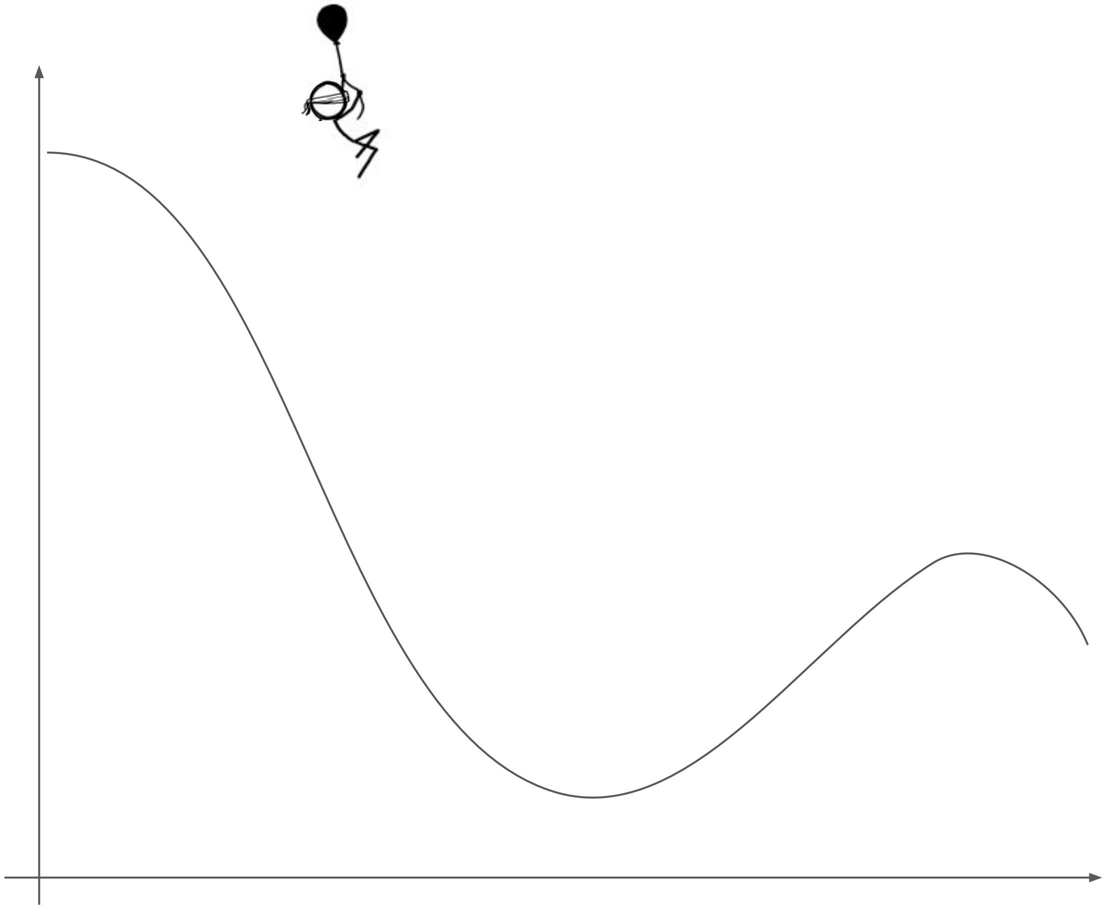
Review: you have been dropped into a mountain range

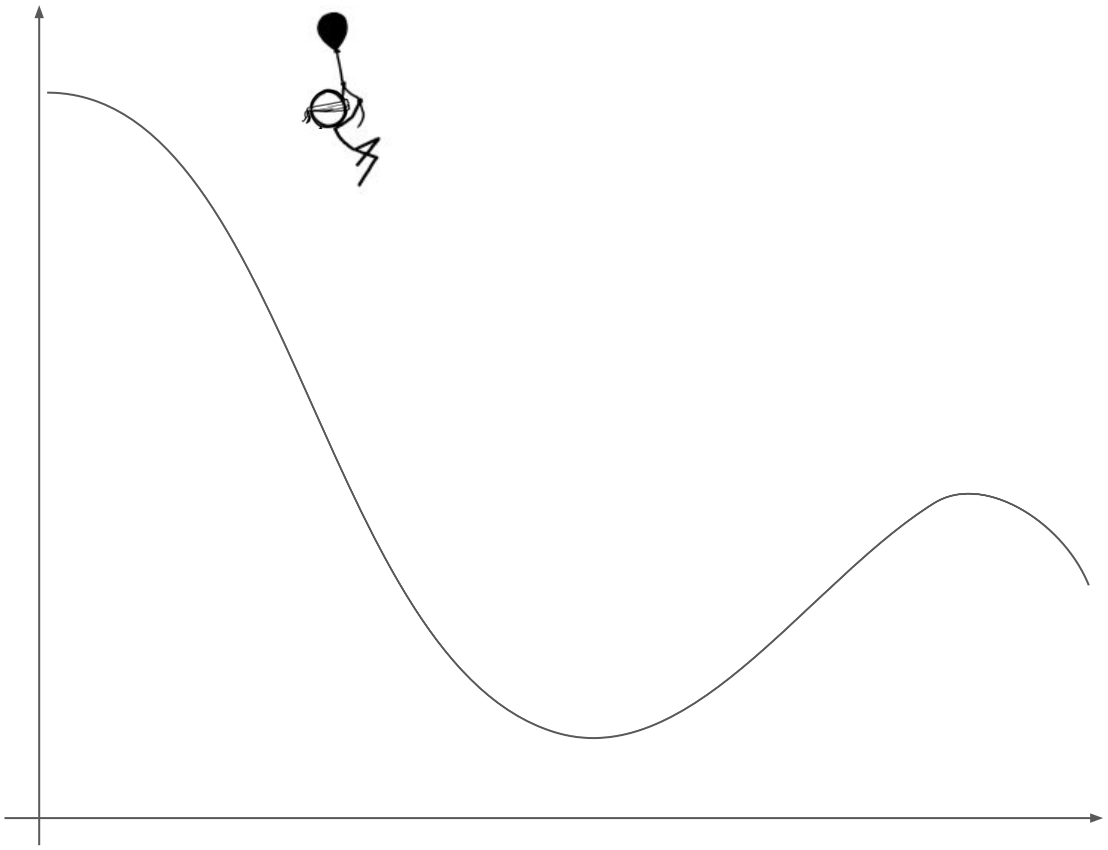


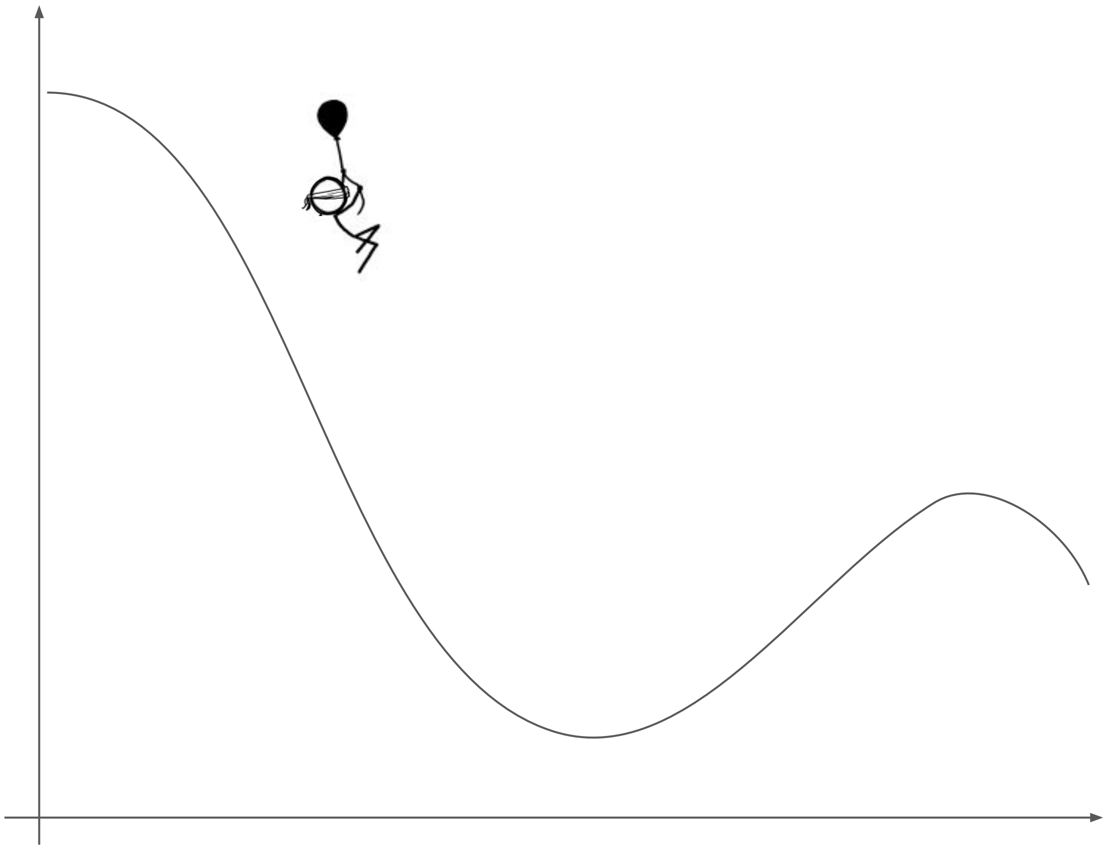
[XKCD What if]

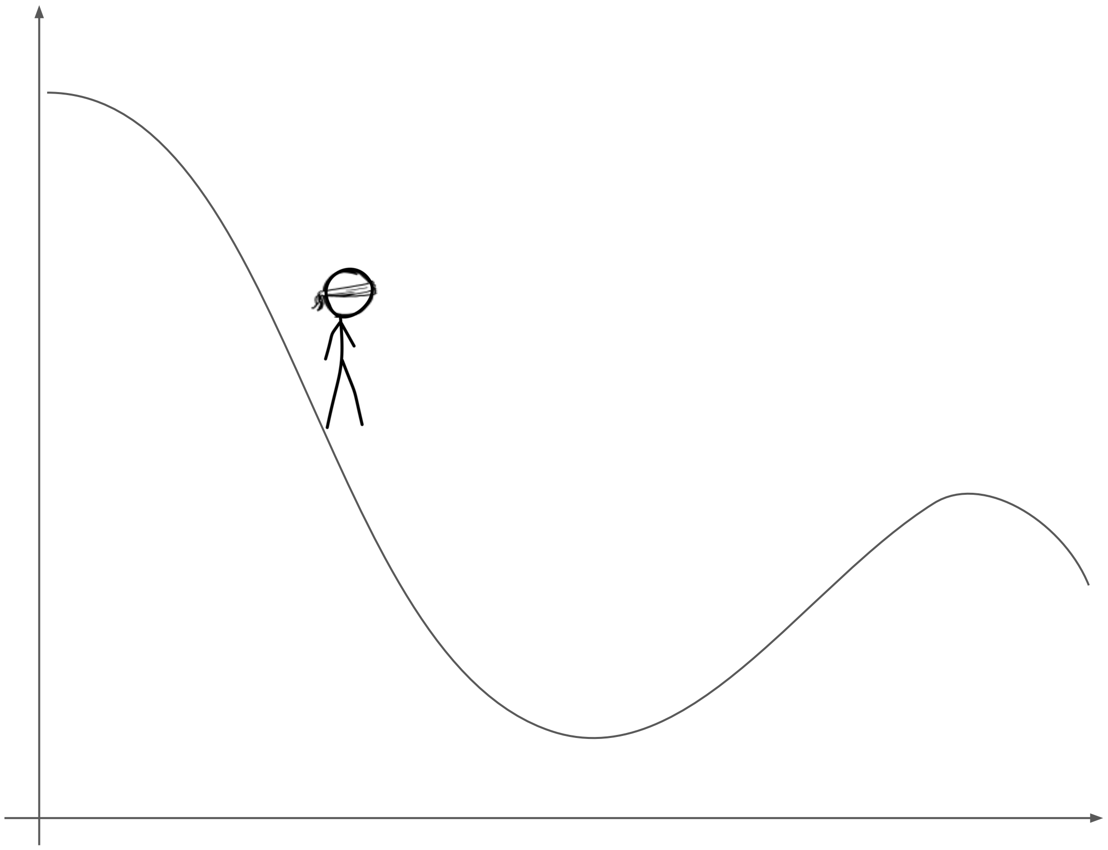
Blindfolded

Surprisingly, this is the classic analogy.

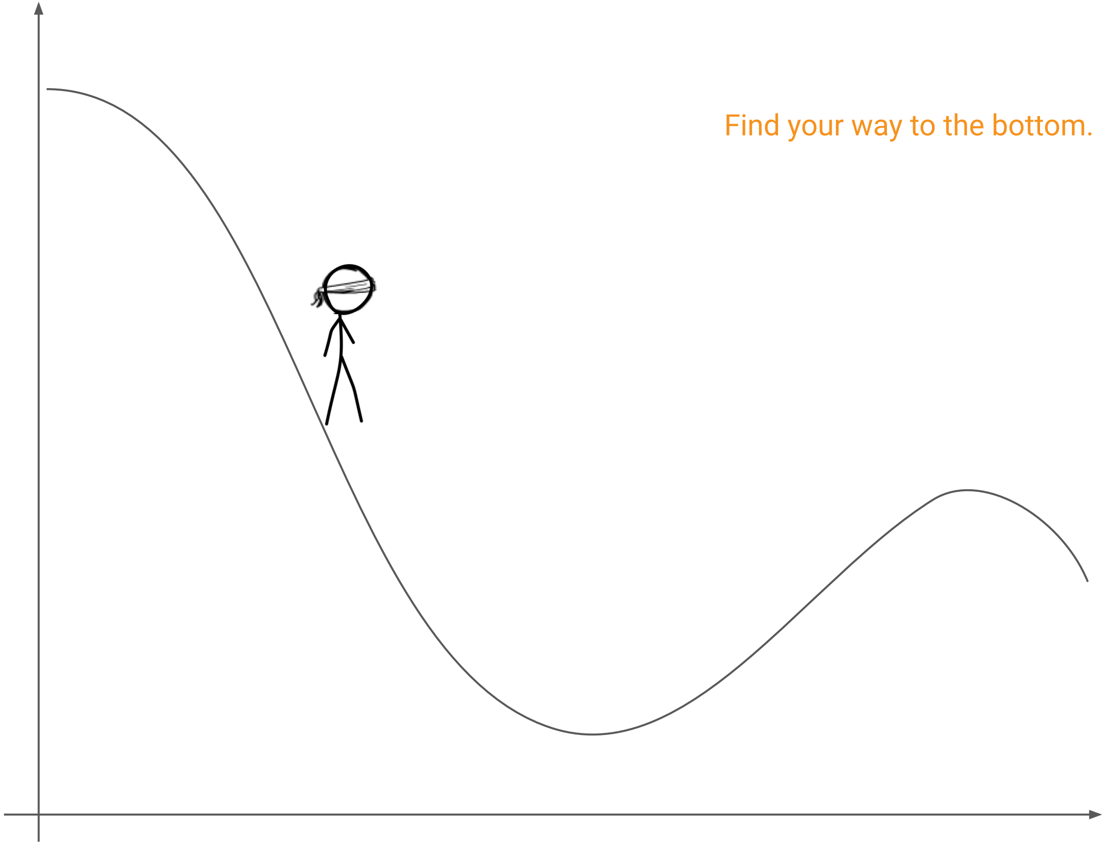






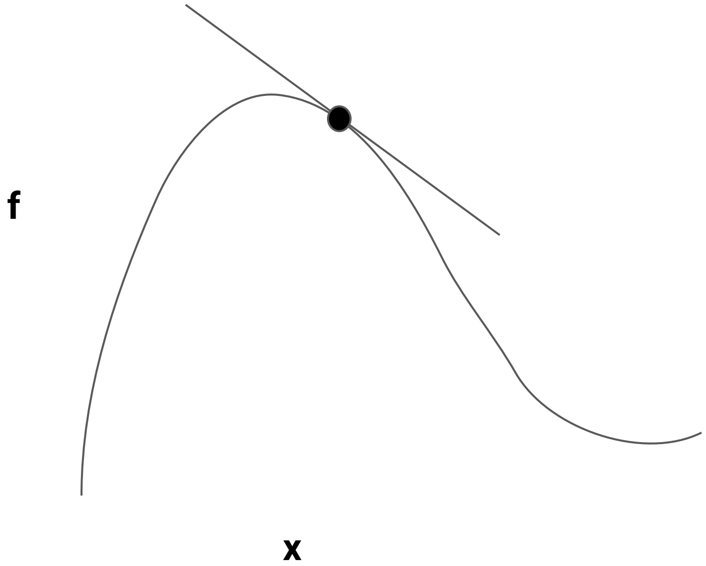


Find your way to the bottom.



Recall

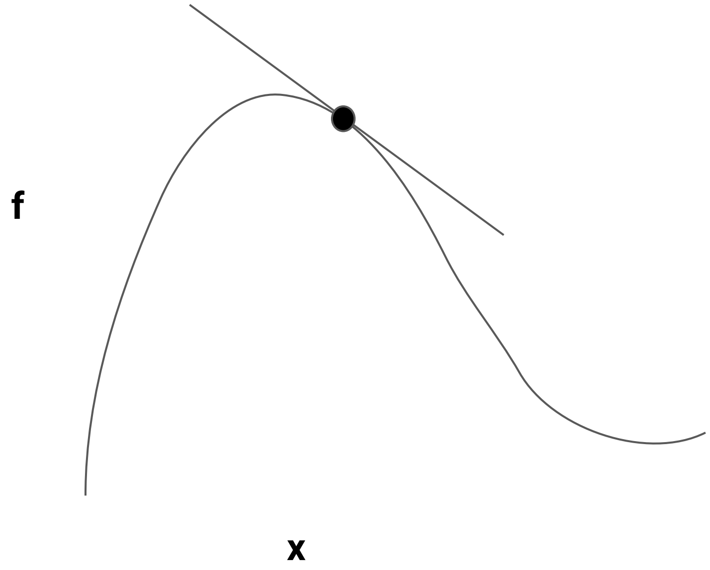
The derivative of f with respect to x tells us how a tiny change in x causes a tiny change in f. Gives us both direction and magnitude.



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

Recall

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 x at this point.

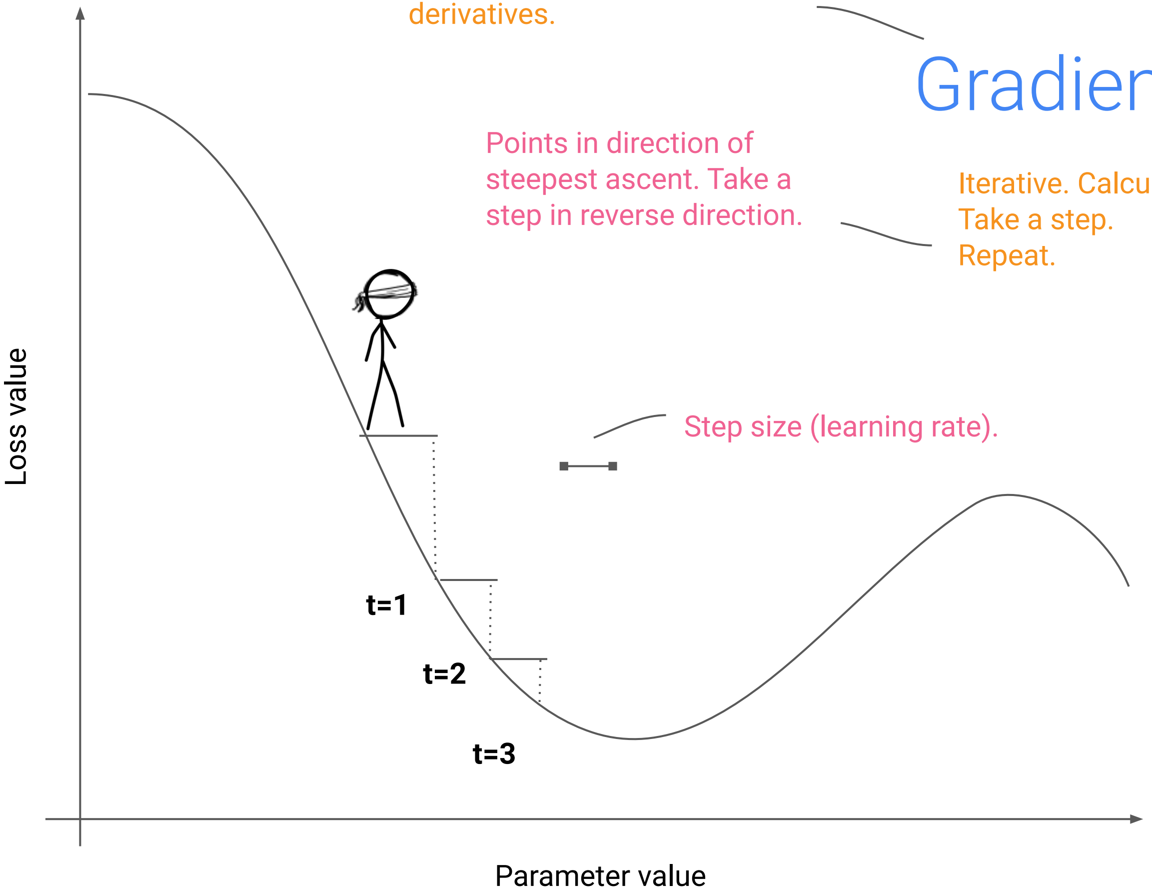
Gradient descent

Gradient = A vector of partial derivatives.

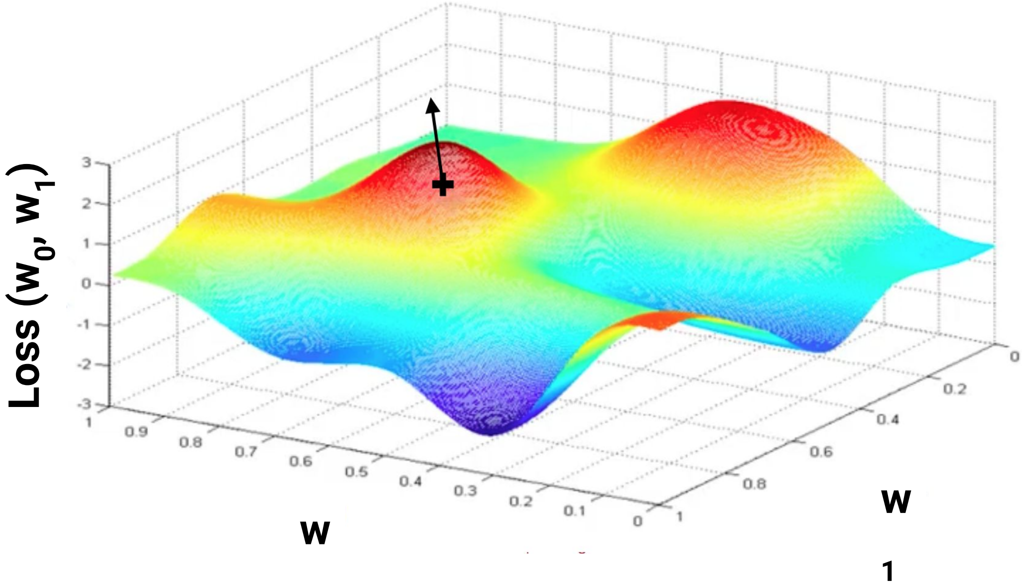
Points in direction of steepest ascent. Take a step in reverse direction.

Iterative. Calculate the gradient.
Take a step.
Repeat.

Step size (learning rate).



With two variables



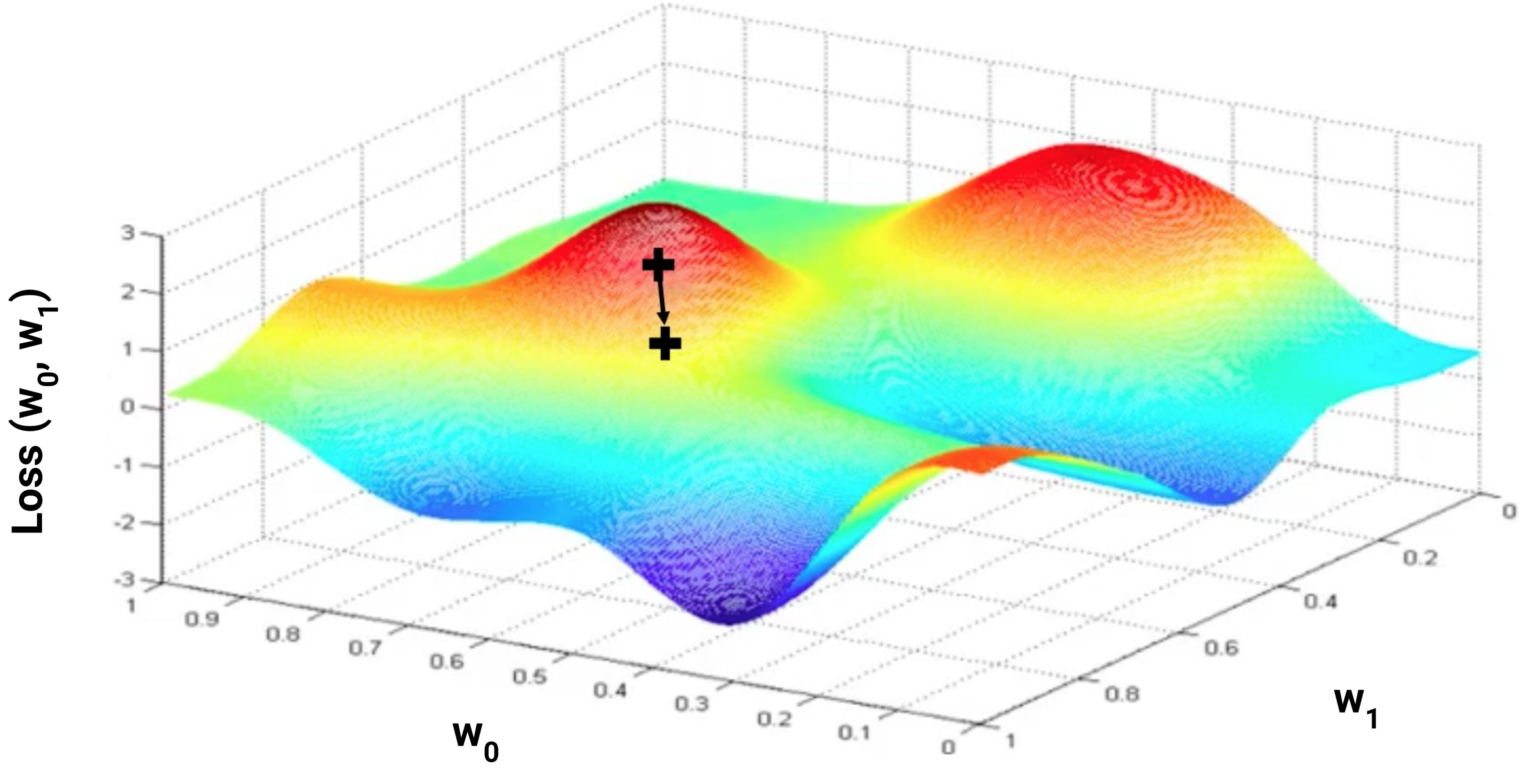
$$\nabla_w Loss = \frac{\partial Loss}{\partial w_0}, \frac{\partial Loss}{\partial w_1}$$

In case it's been a while, 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).

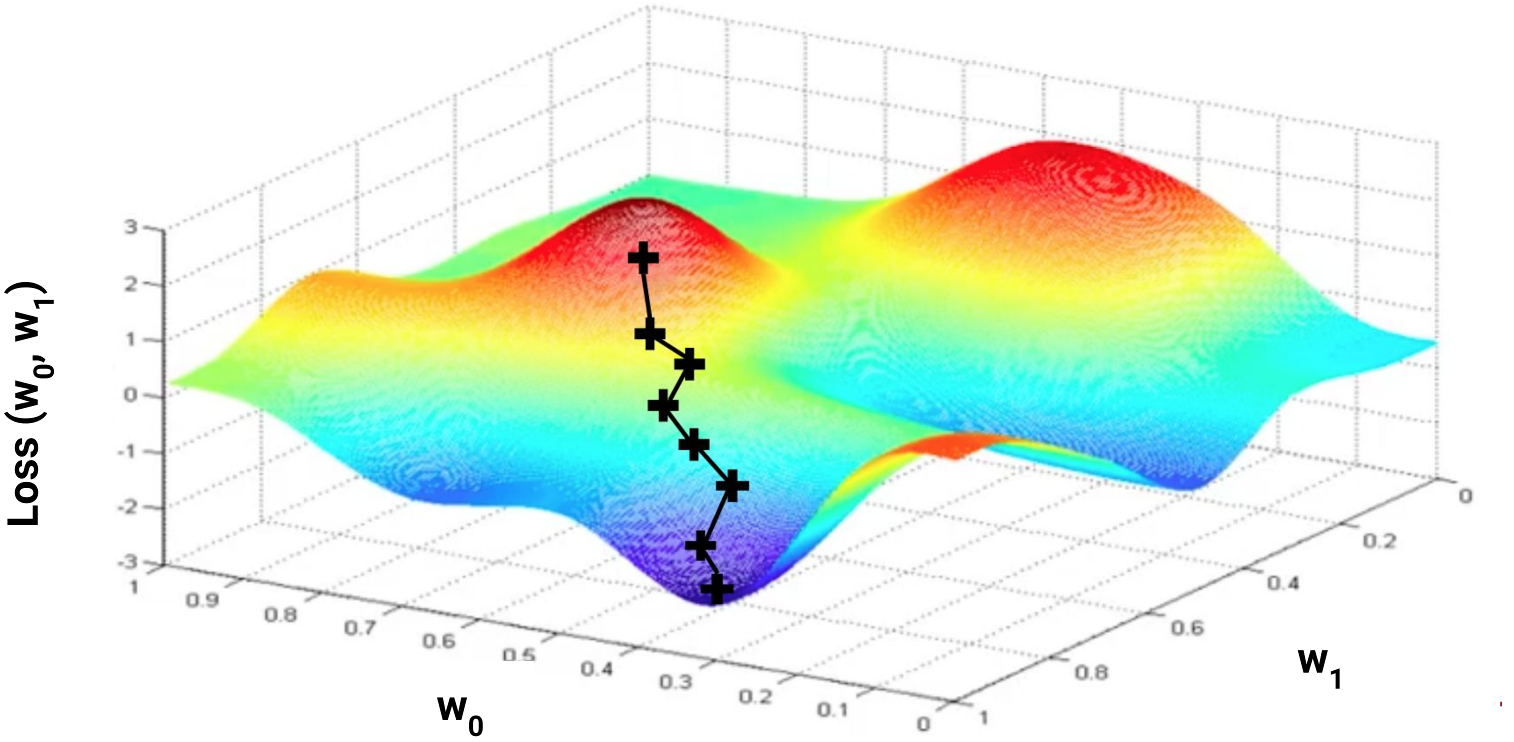
The gradient points in the direction of steepest ascent. We usually want to minimize a function (like loss), so we take a step in the opposite direction.

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

Take small step in opposite direction.

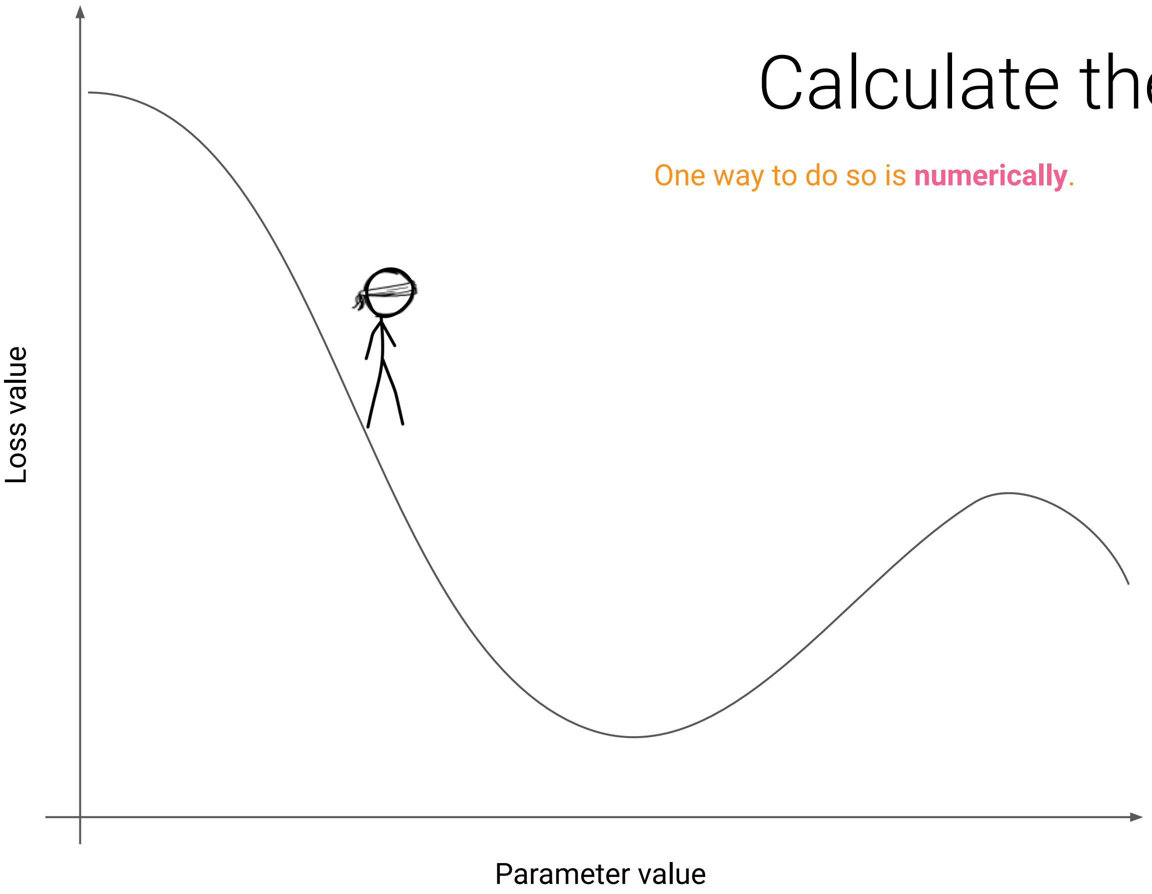


Repeat until convergence



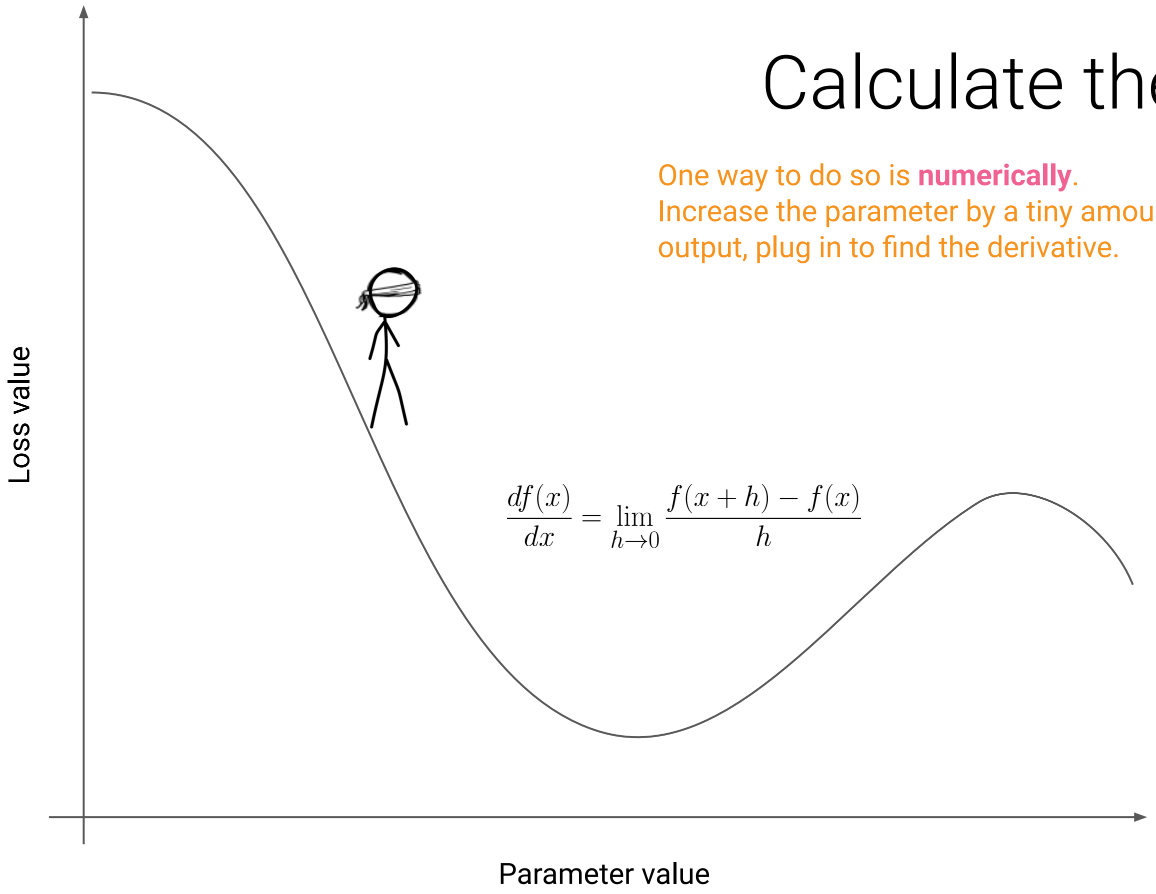
Calculate the gradient

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.



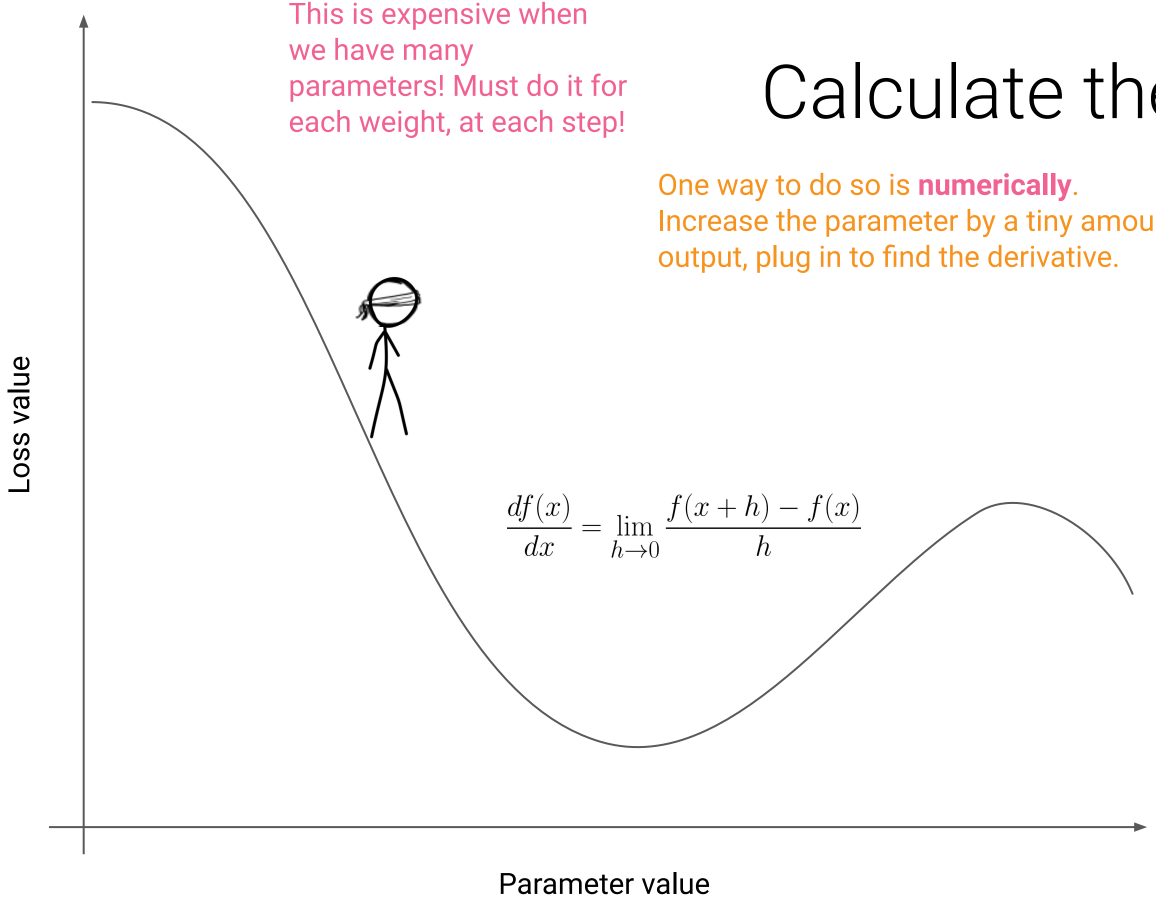
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Calculate the gradient

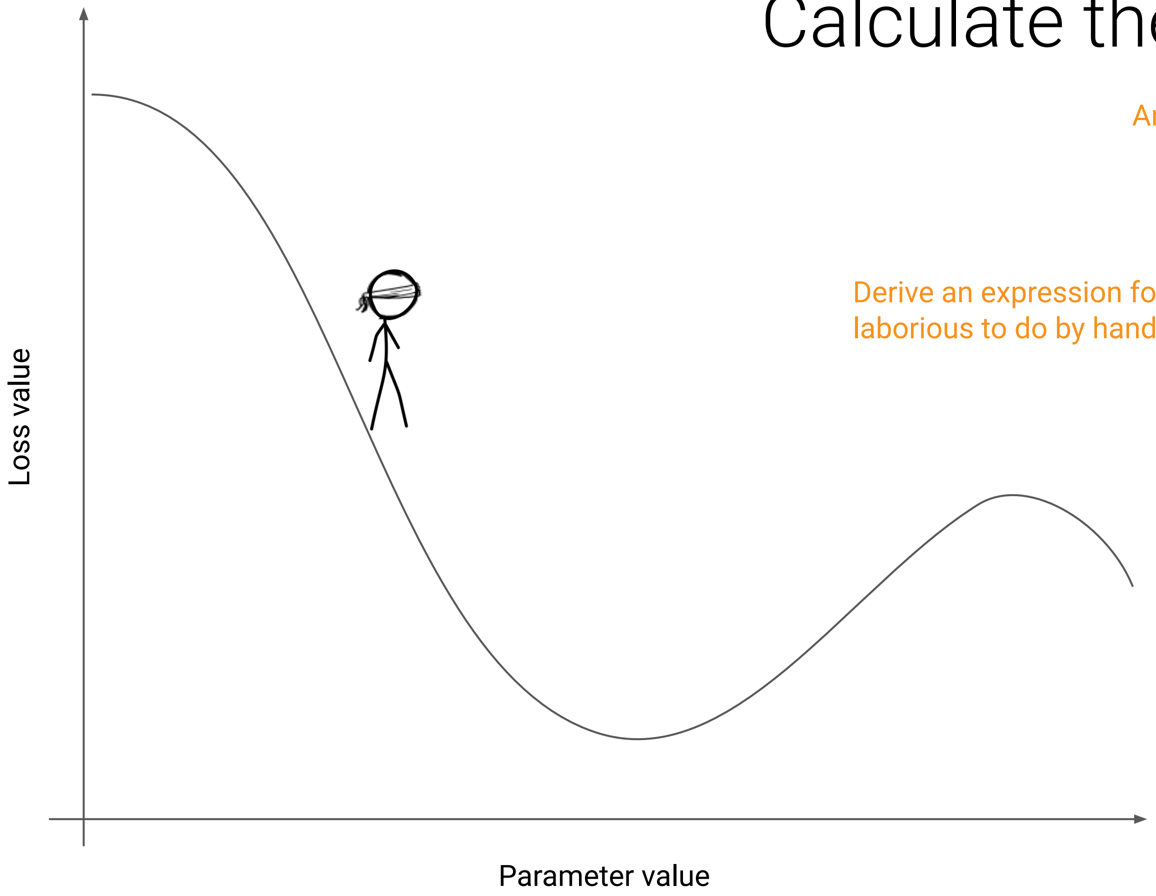
This is expensive when we have many parameters! Must do it for each weight, at each step!

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

Or, util other stopping criteria (number of steps, no further improvement after n successive steps, etc). Example: see early stopping callback.

3. Calculate gradient of loss w.r.t. weights.

$$\nabla_w Loss$$

4. Update weights.

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

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

Calculate the gradient

Two sources of complexity

1. The gradient descent algorithm itself (momentum, or adaptive learning rates - not bad, usually intuitive).

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

Backprop refers to the method of computing gradients (not the end-to-end optimization process) - that's gradient descent, which is independent of the method used to calculate the grads.

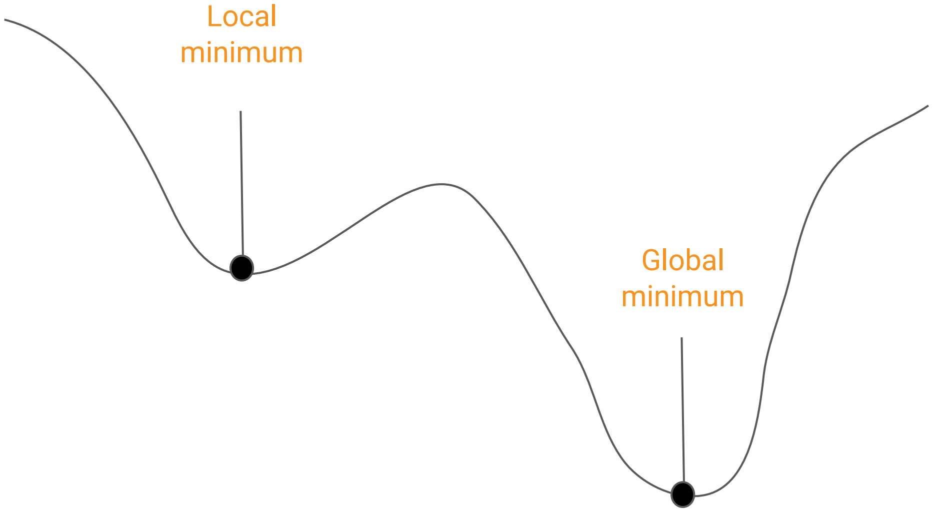
Many optimizers

- Adadelta
- Adagrad
- $\boxed{\text{Adam}}$ A good default choice.
- Adamax
- Nadam
- Optimizer
- RMSprop
- SGD

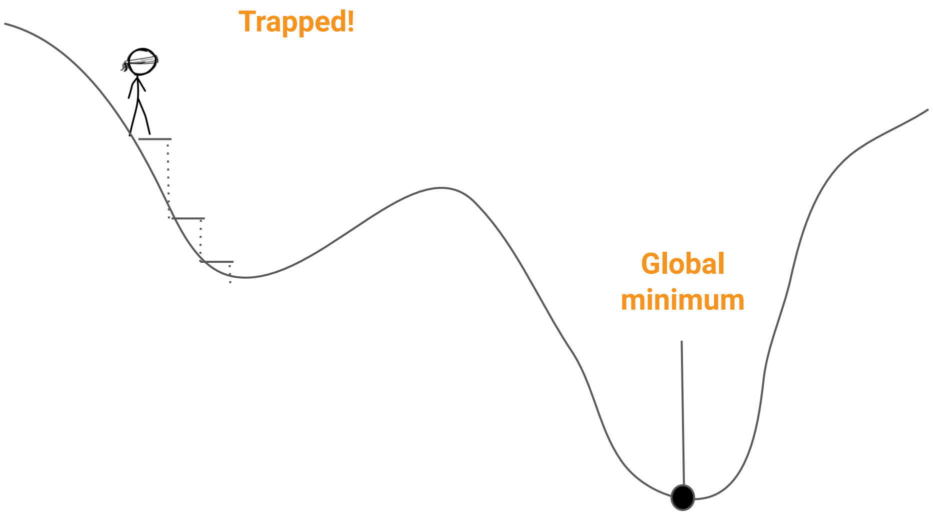
Trivia, which of these was published on slides during a class (rather than in a paper?)

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

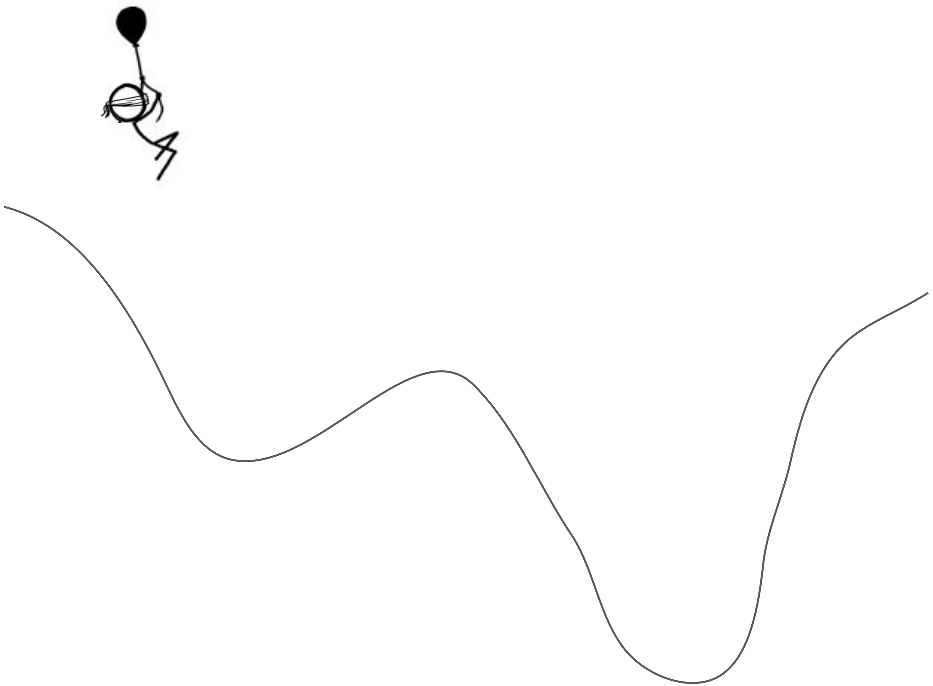
Local and global minimum



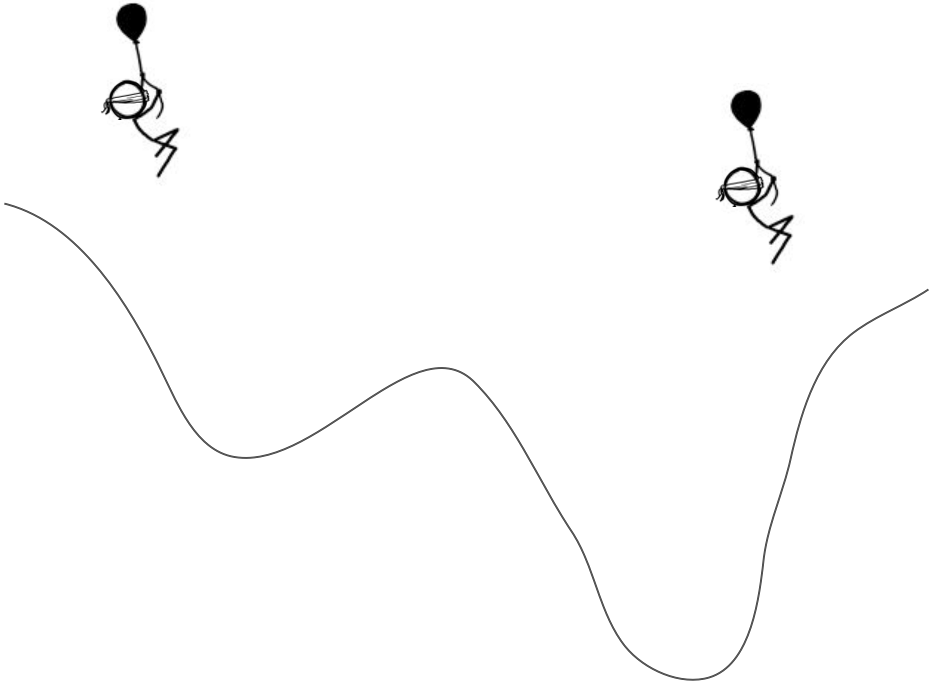
Local and global minimum



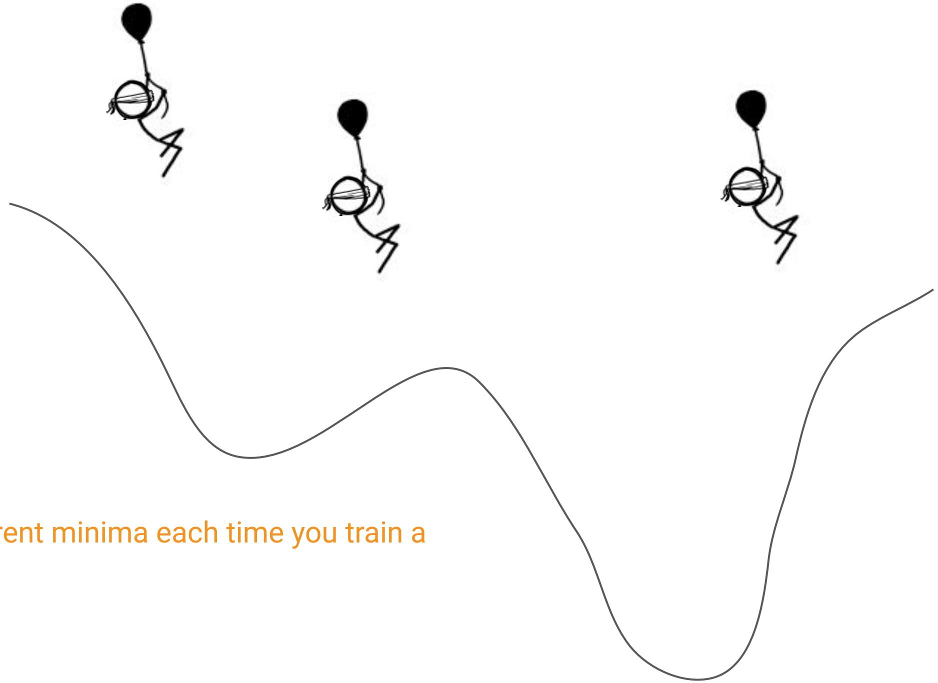
Result depends on starting point



Result depends on starting point



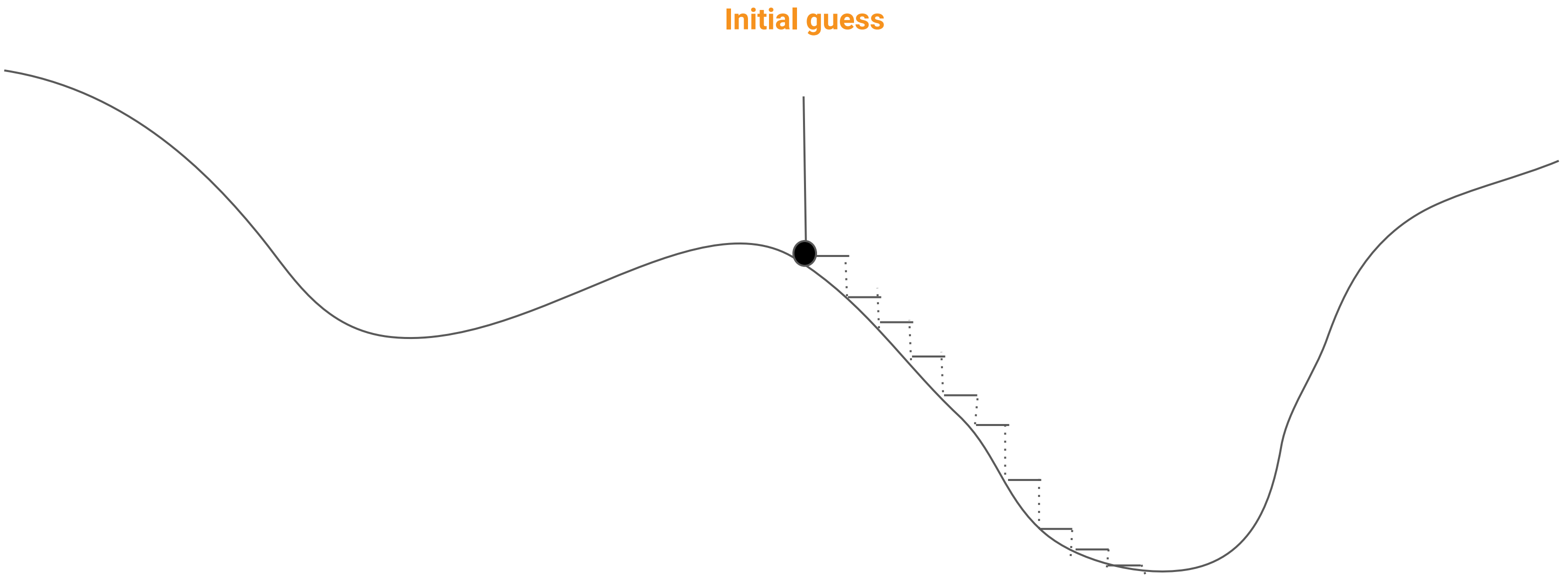
Result depends on starting point



You may end up in a different minima each time you train a network.

Learning rates

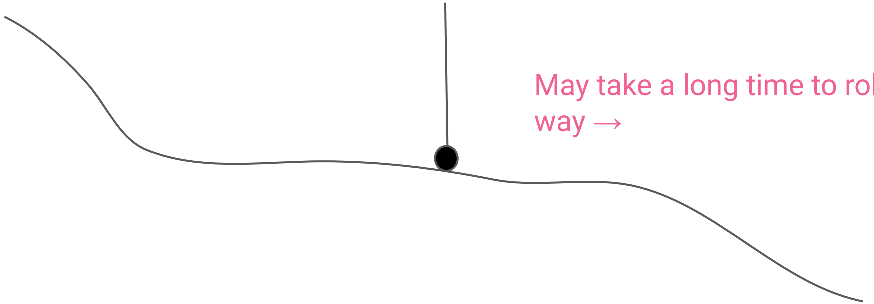
A low learning rate could take many steps to converge



Learning rates

Or, stall in regions where the gradient is small.

Initial guess

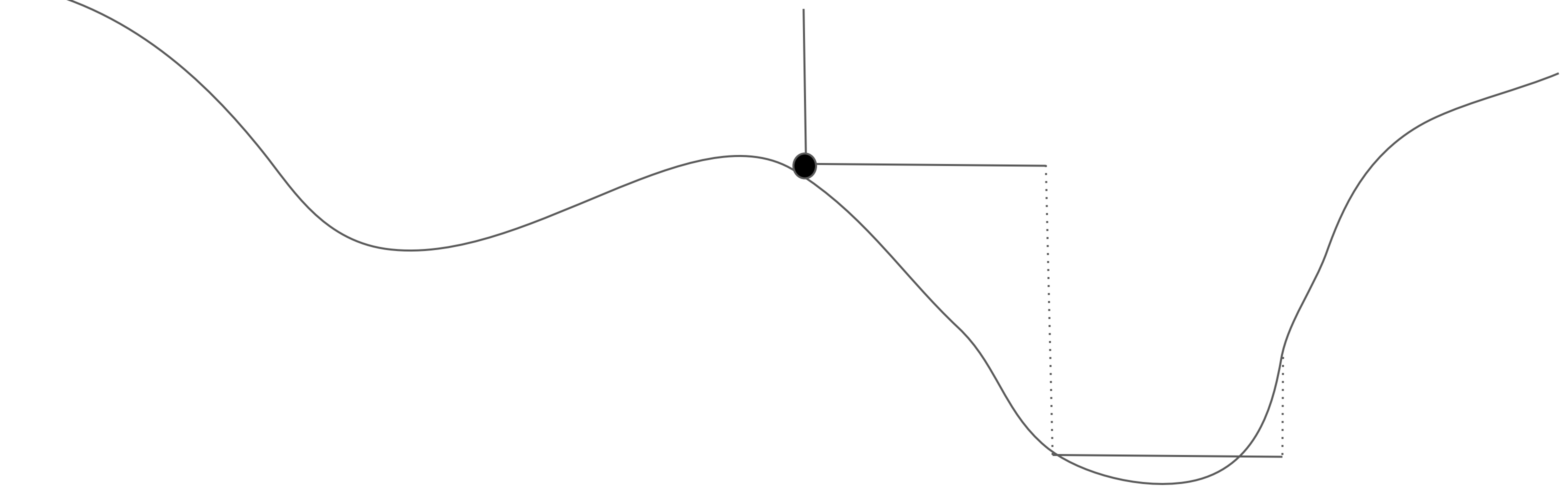


May take a long time to roll this way →

Learning rates

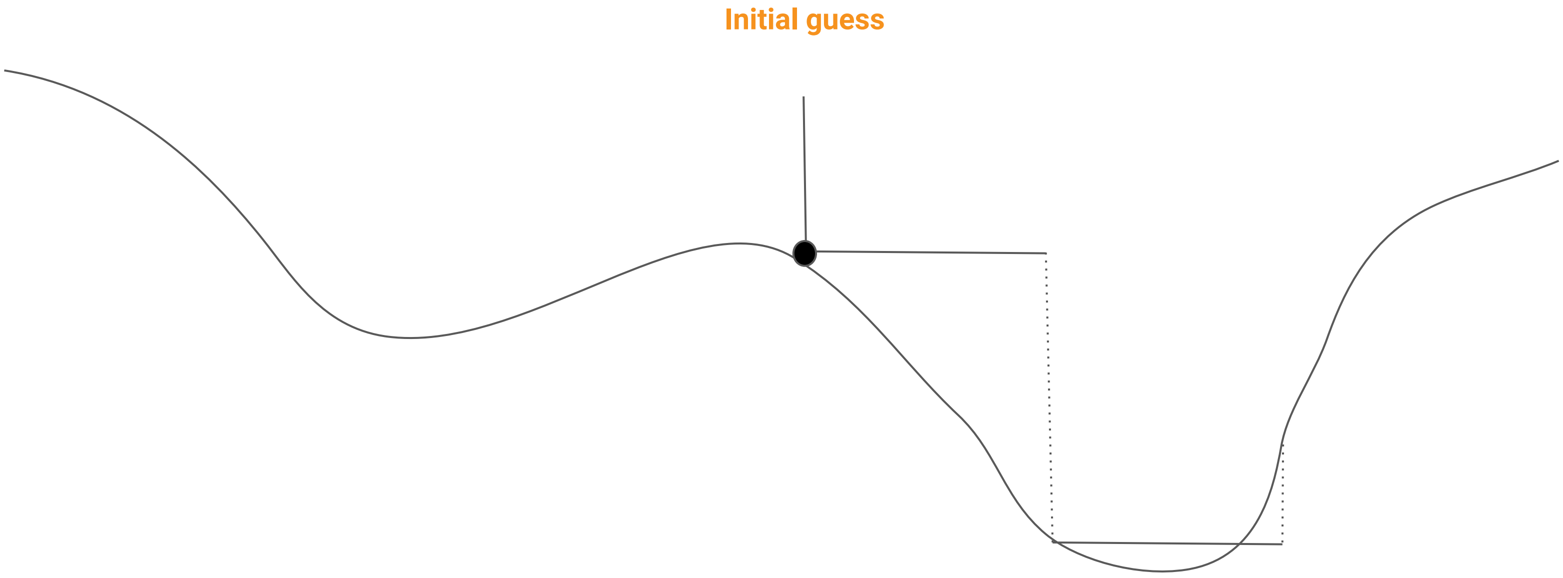
A high learning rate could jump over the minimum!

Initial guess



Learning rates

Or oscillate around it, never converging.



Demo: high learning rates

playground.tensorflow.org

Momentum

Image 2: SGD without momentum

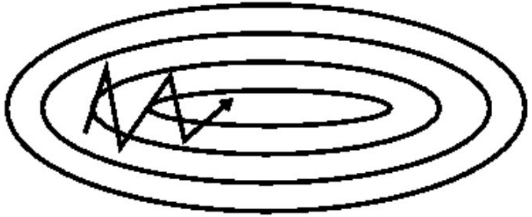


Image 3: SGD with momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

ruder.io/optimizing-gradient-descent/index.htm

Options

Note: update weights using the average gradient (not average loss). We have to compute the gradient for each example in a batch.

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).

Options

Note: update weights using the average gradient (not average loss). We have to compute the gradient for each example in a batch.

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.

How many updates to your weights per epoch? Say you have 128 examples in your training set, and batch size of 32

Batch

- ?

Stochastic

- ?

Mini-batch

- ?

How many updates to your weights per epoch? Say you have 128 examples in your training set, and batch size of 32

Batch (full)

- 1 (we update the weights once, with the average gradient for all 128 examples)

Stochastic (one at a time)

- 128 (we classify an example, calculate the gradient, update the weights, then move on to the next one)

Mini-batch (some)

- 4 (we classify 32 examples, calculate the avg gradient, update weights, then move to next 32)

Say you have two GPUs, how can you go faster?

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.

Increase effective batch size

- Say one GPU can handle 64 examples / iteration
- Use two for an effective batch size of 128
- Average gradients before applying

Break, time to work on HW3
A3_starter.ipynb on CourseWorks

Questions that came up a lot

How much data

Will x be ported to TF2 and why

Adam / TensorBoard paper

Zuckerman

Calculating the numerical gradient

Demo

Computational graphs are a helpful abstraction.

Running example

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

Running example

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

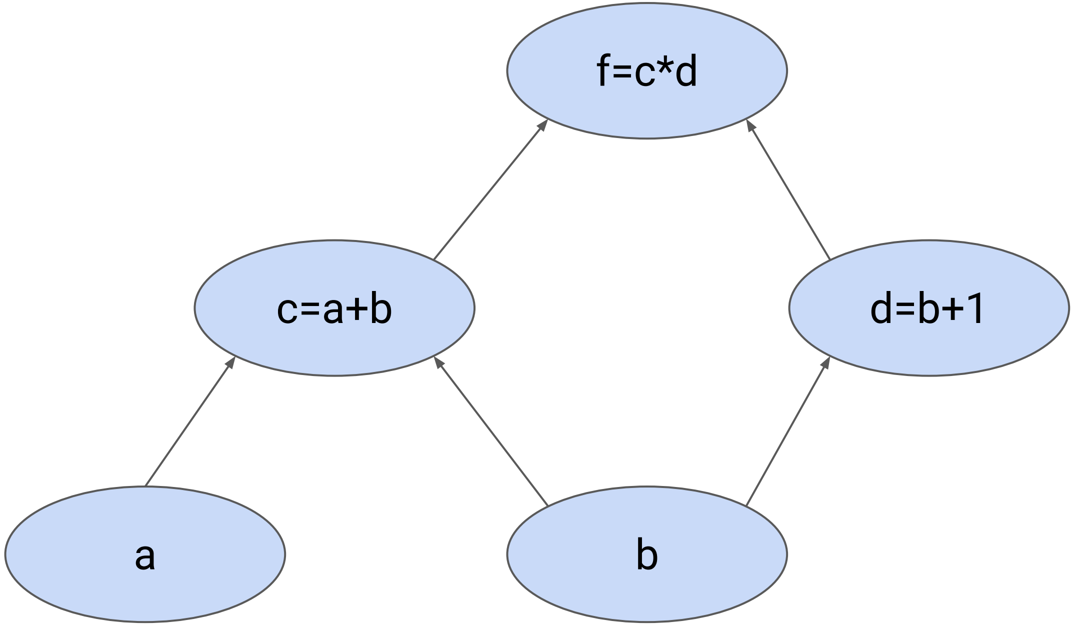
To compute f we need to perform three operations (two additions, one multiplication).

```
c = a + b

d = b + 1

f = c * d
```

Introduce intermediate variables (one for each operation).

A computational graph

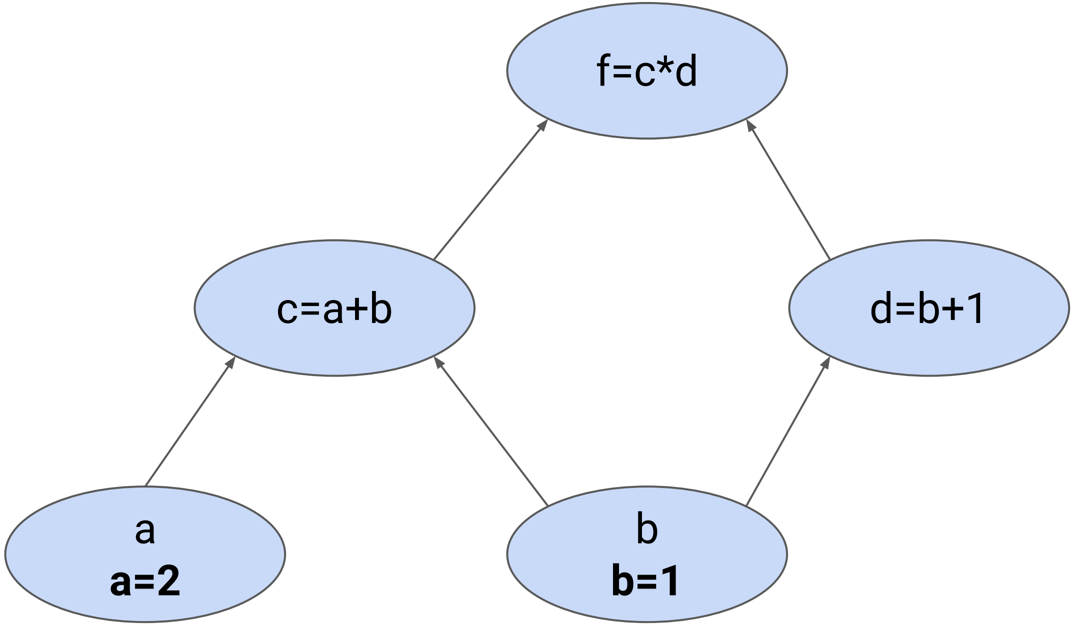
Closely related to a dependency graph.



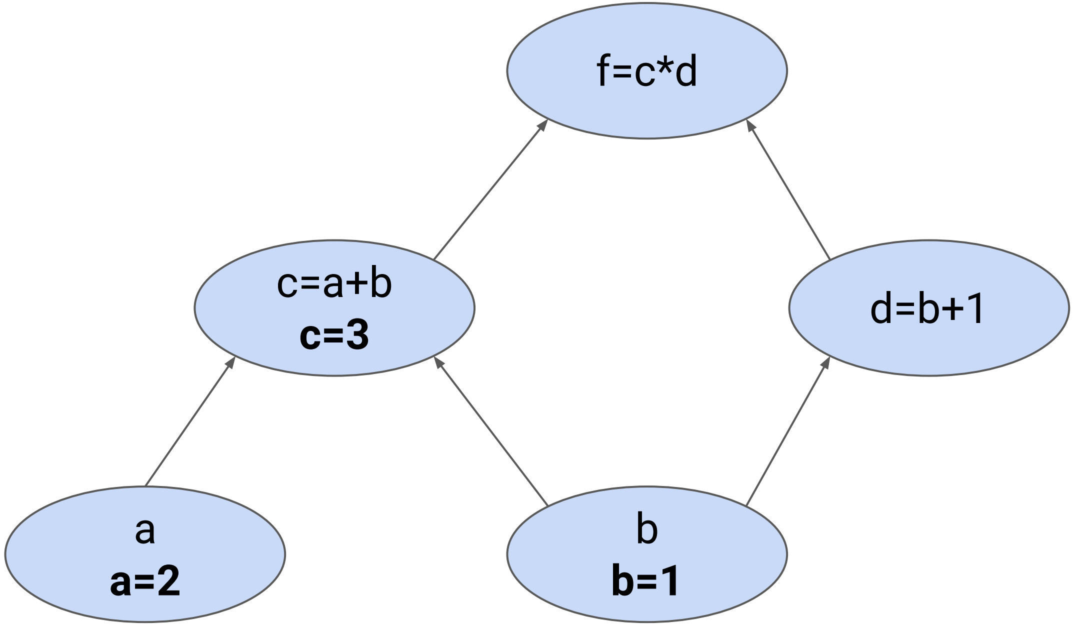
The name "**TensorFlow**" comes from the idea of tensors (n-dimensional arrays) flowing on a graph.

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

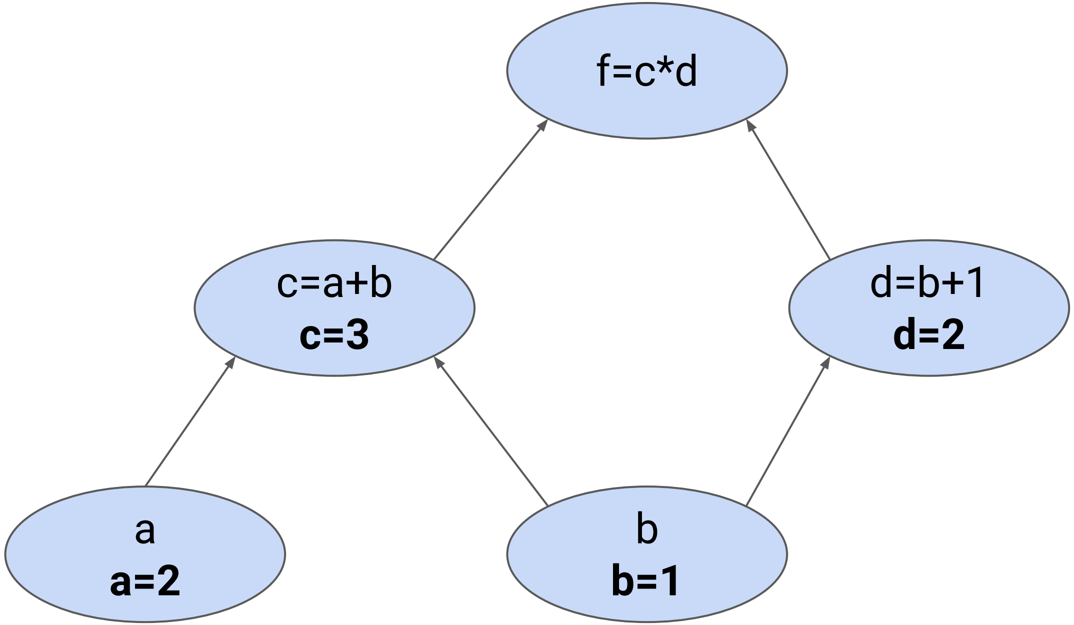
Forward pass



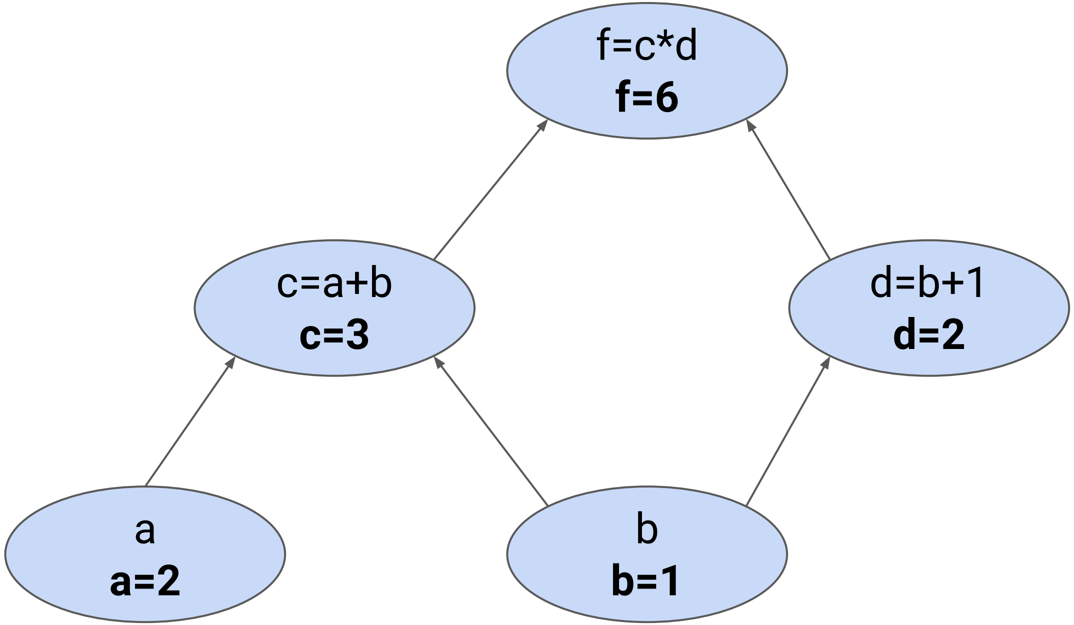
Forward pass



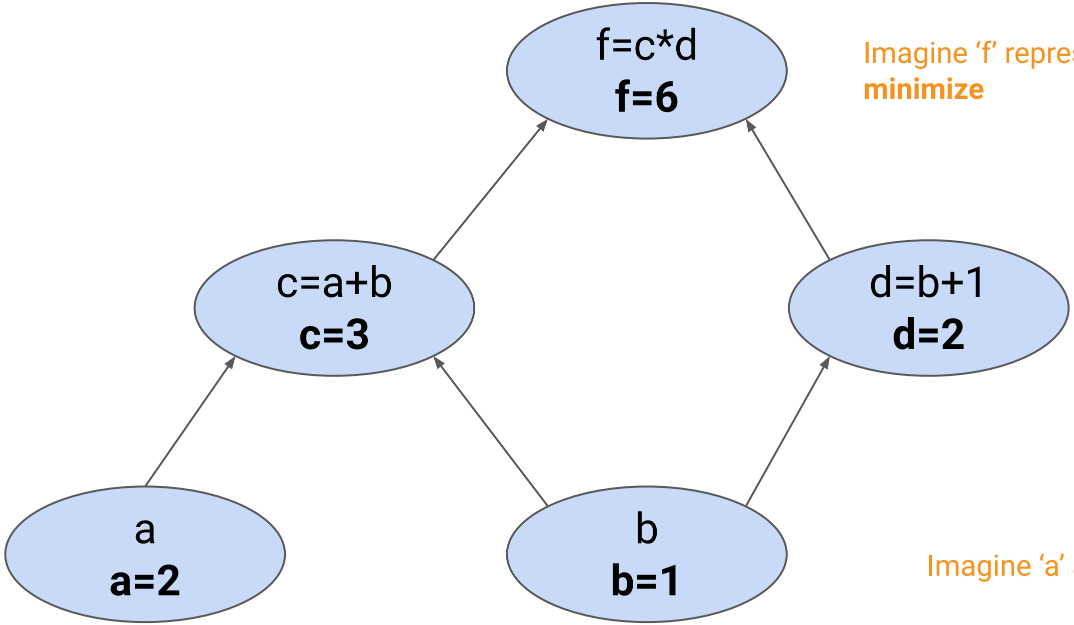
Forward pass



Forward pass



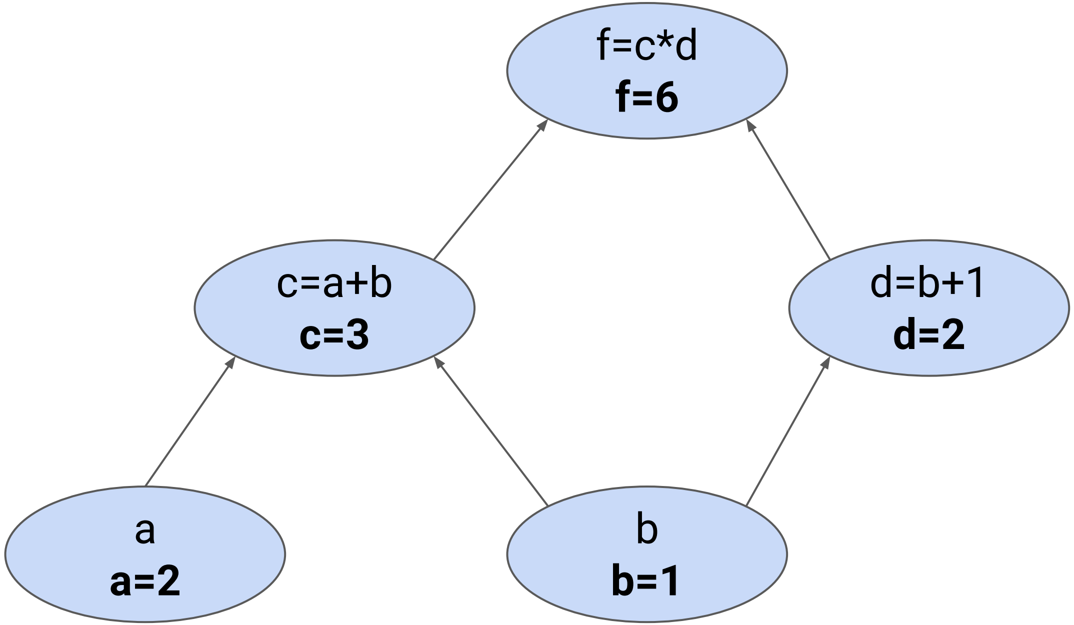
How does adjusting the weights affect the output?



Imagine ‘f’ represents a loss we want to **minimize**

Imagine ‘a’ and ‘b’ represent our **weights.**

Need to find the gradient



Read as gradient of the loss w.r.t. the weights.

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

Read: 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’?) Ditto for b.

Numerically

```python
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.

Worked example on CourseWorks

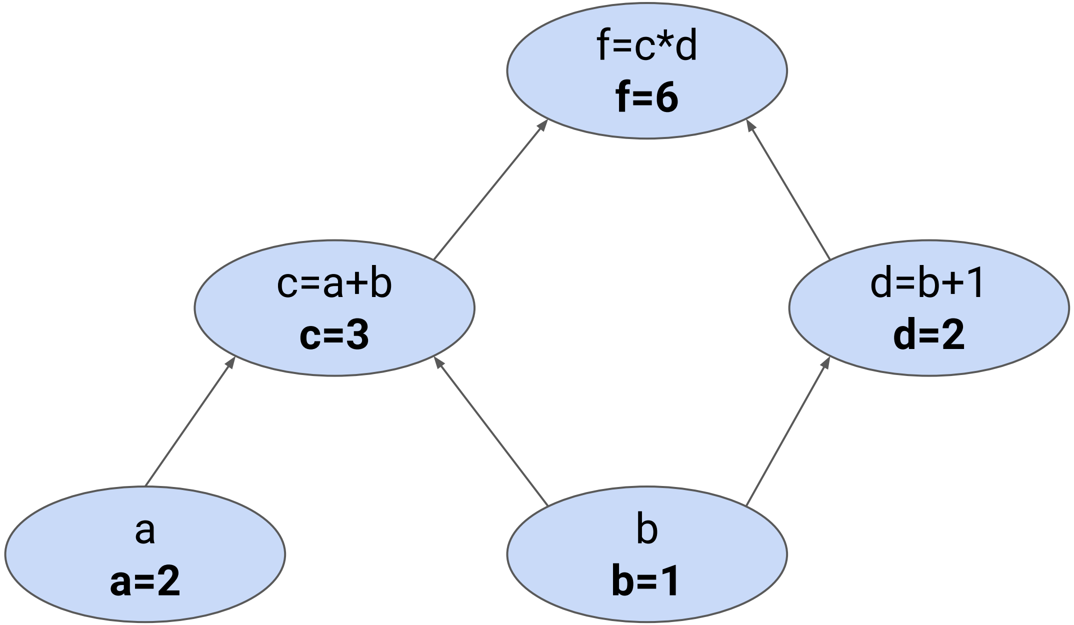
```python
def numeric_gradient(f, params, h=1e-4):
  grad = np.zeros_like(params) # Vector of partial derivatives
  for i in range(len(params)): # Loop over weights
    orginal_val = params[i]
    params[i] += h
    plus_h = f(*params) # f(x + h)
    params[i] = orginal_val
    params[i] -= h
    minus_h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

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

Central difference as it happens, but idea is the same w/ others

Numerically: it's 2,5



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

Meaning:

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

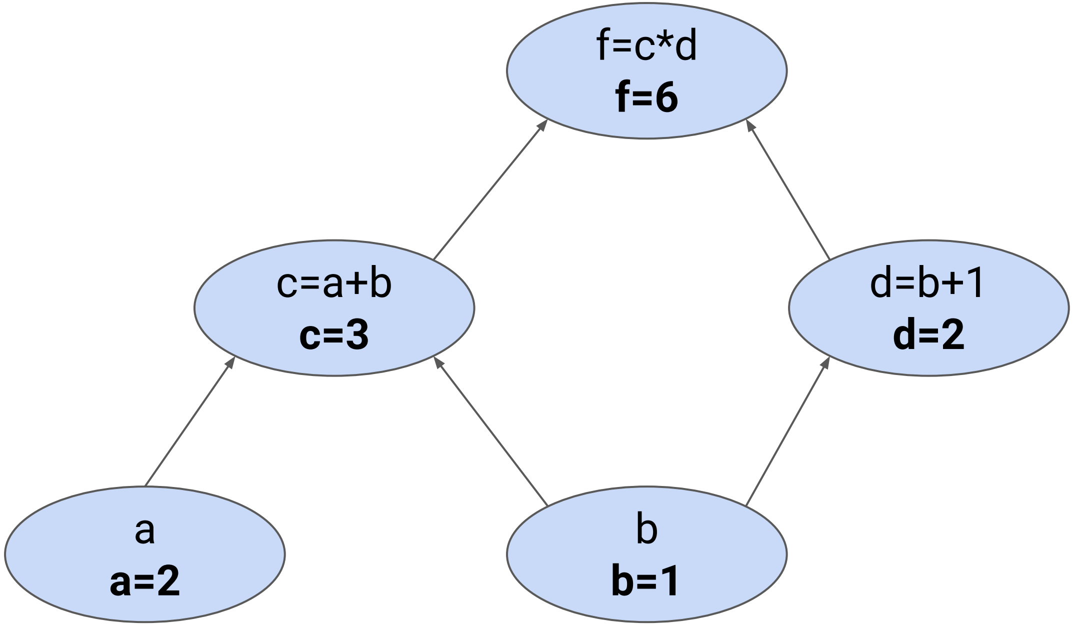
For every weight in the network:

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

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

Backprop

Backprop: Efficiently calculate gradients by recursive application of the chain rule on a computational graph.

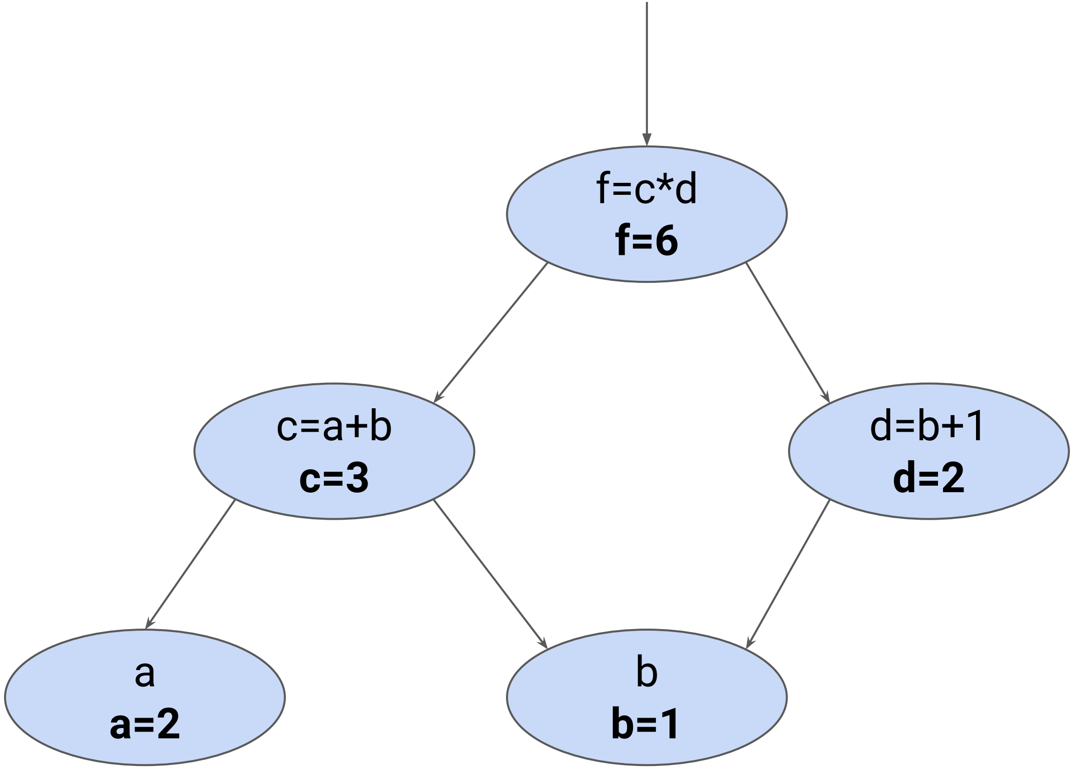


1) Compute the forward pass.

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

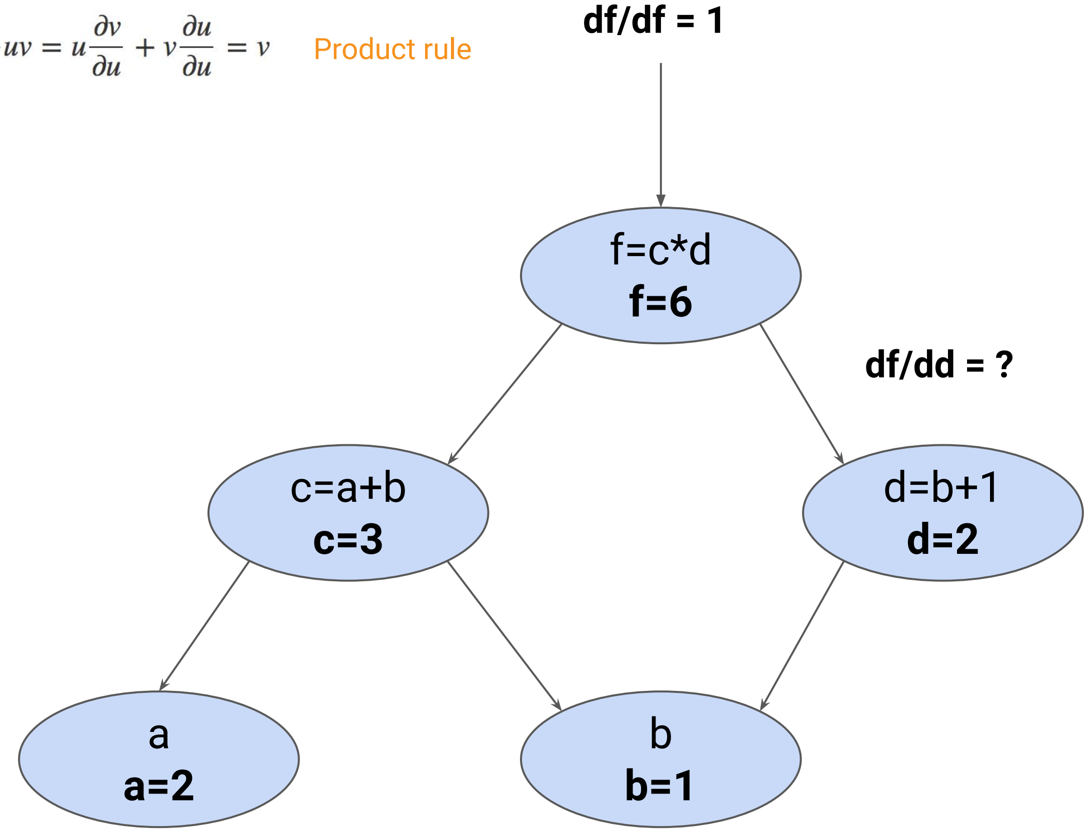
df/df = 1

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

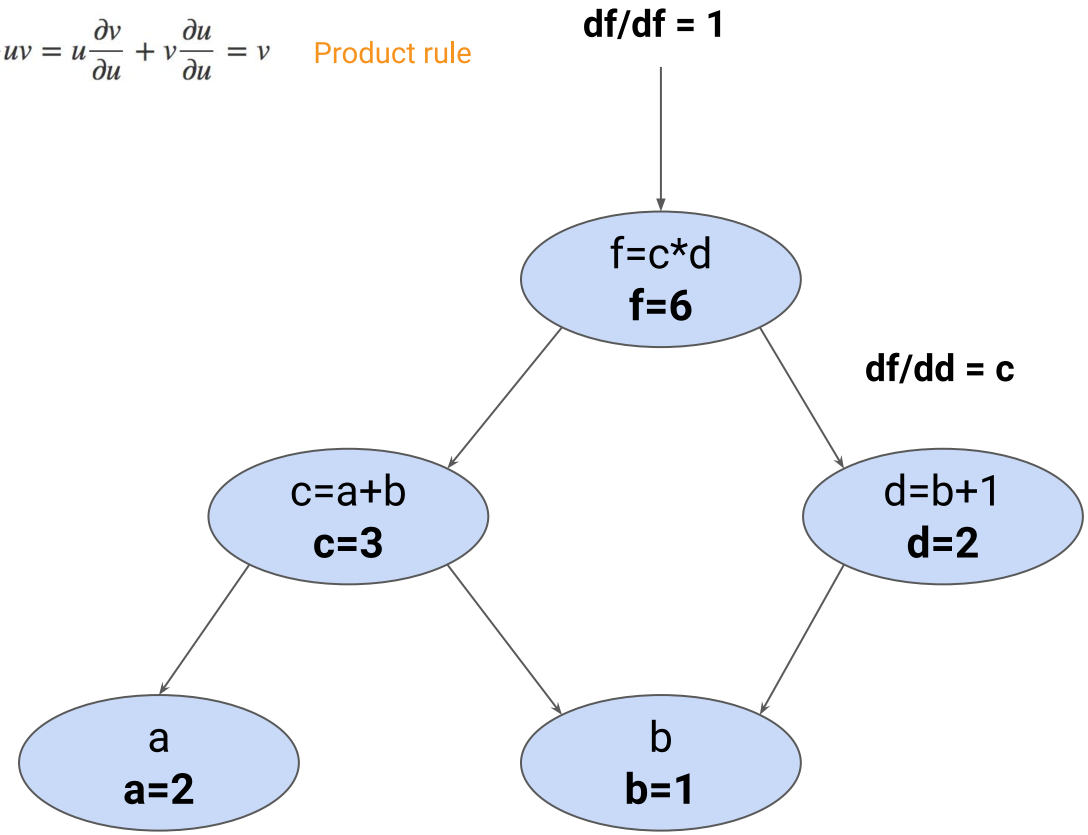
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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

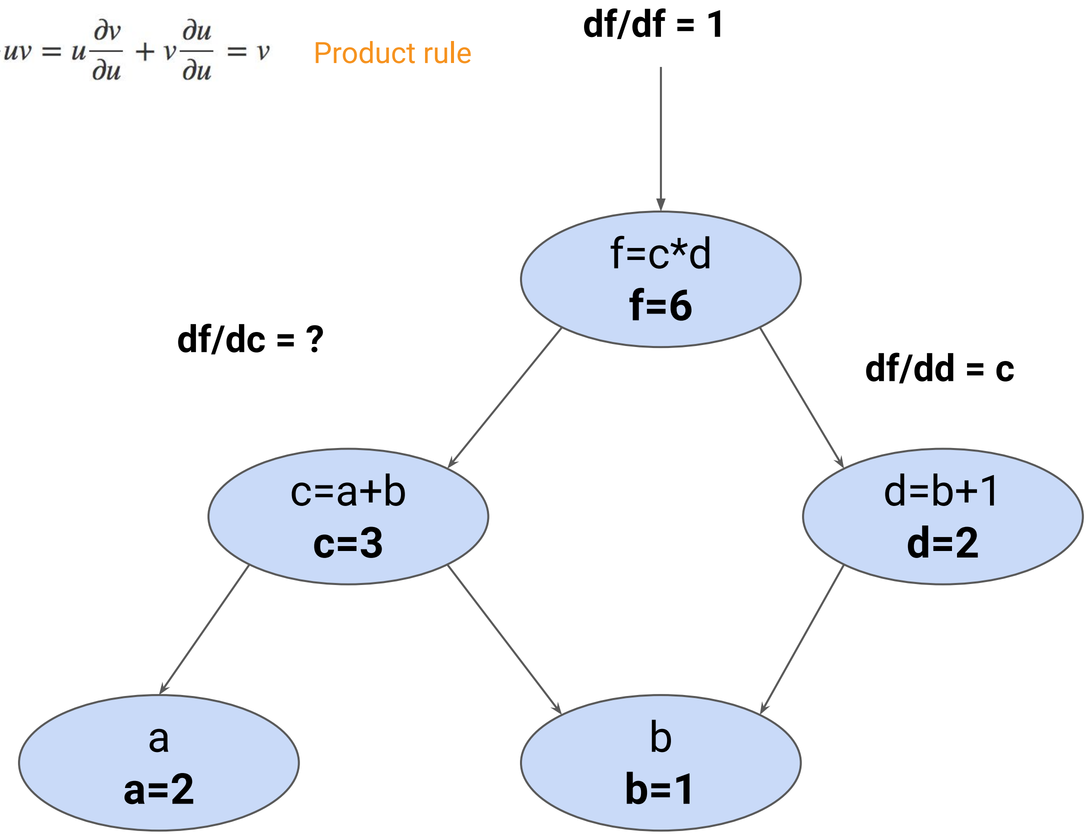
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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

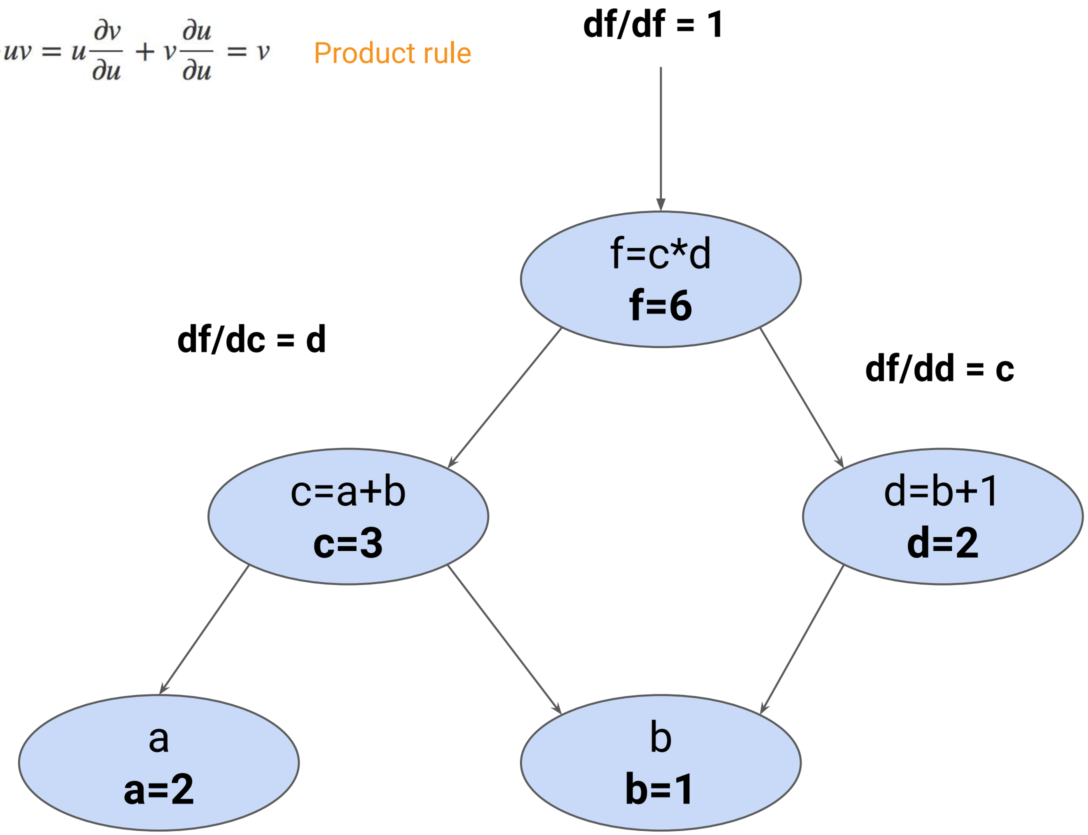
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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

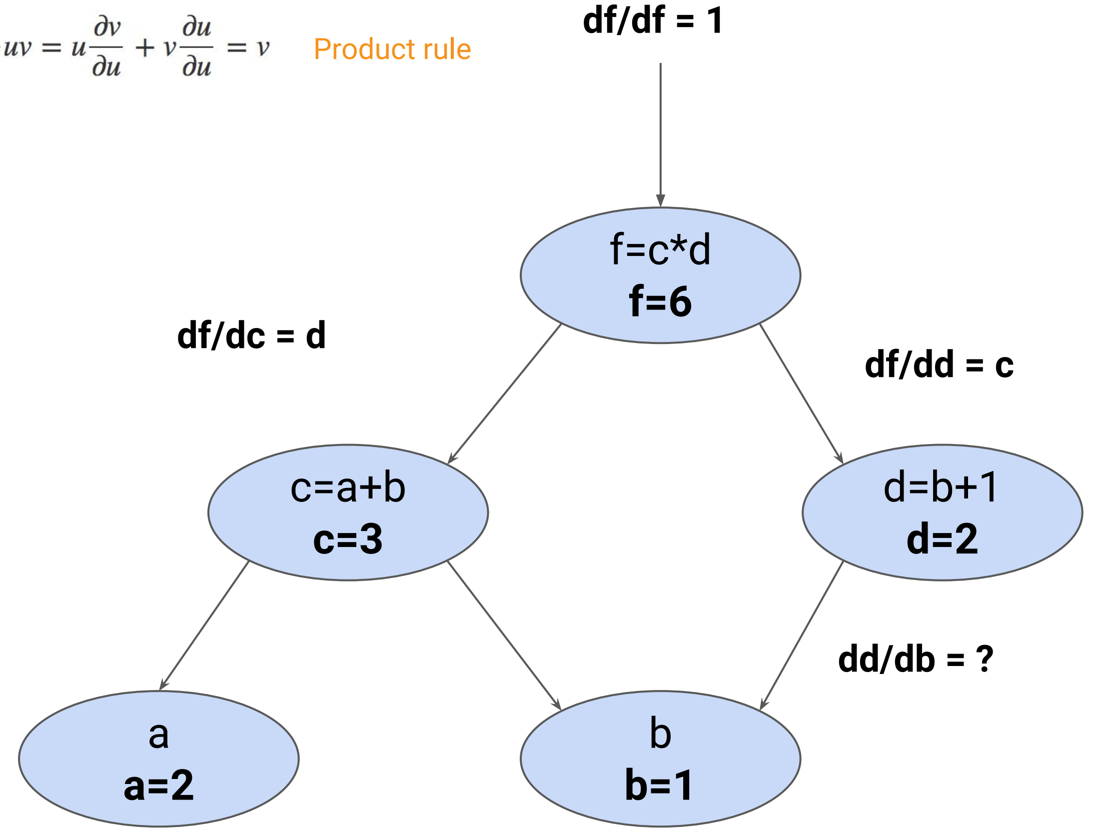
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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

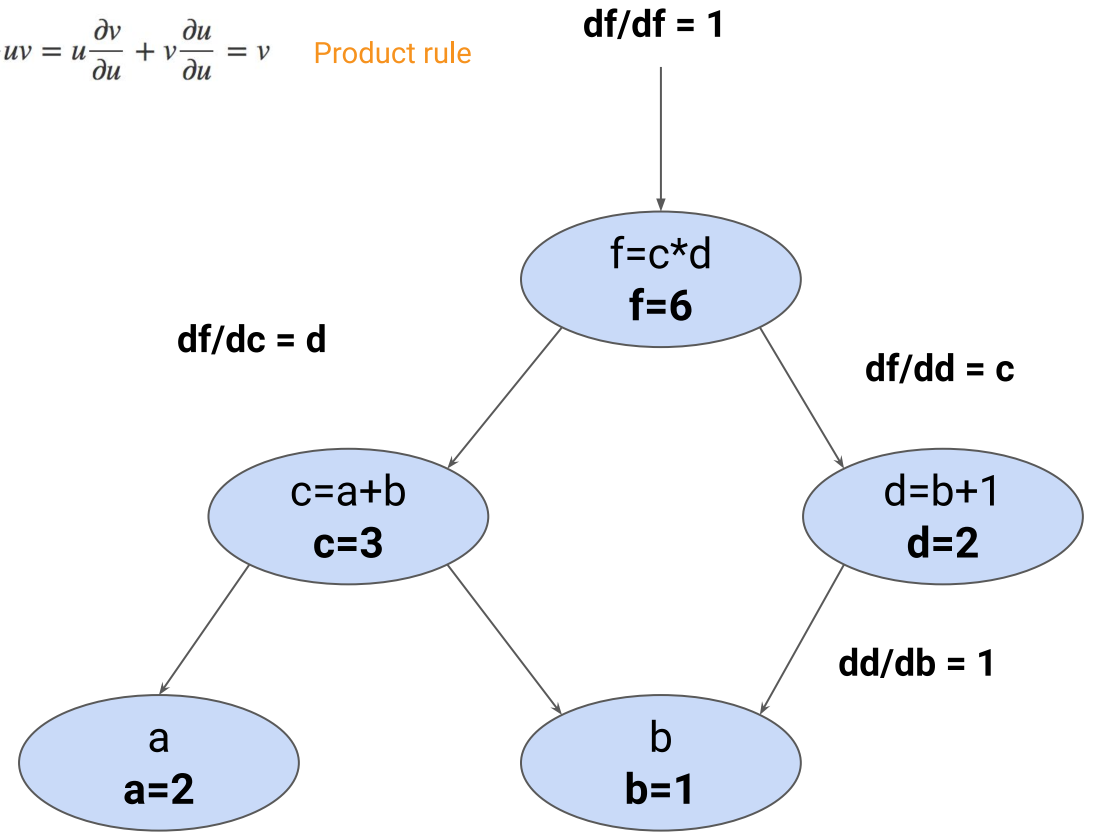
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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

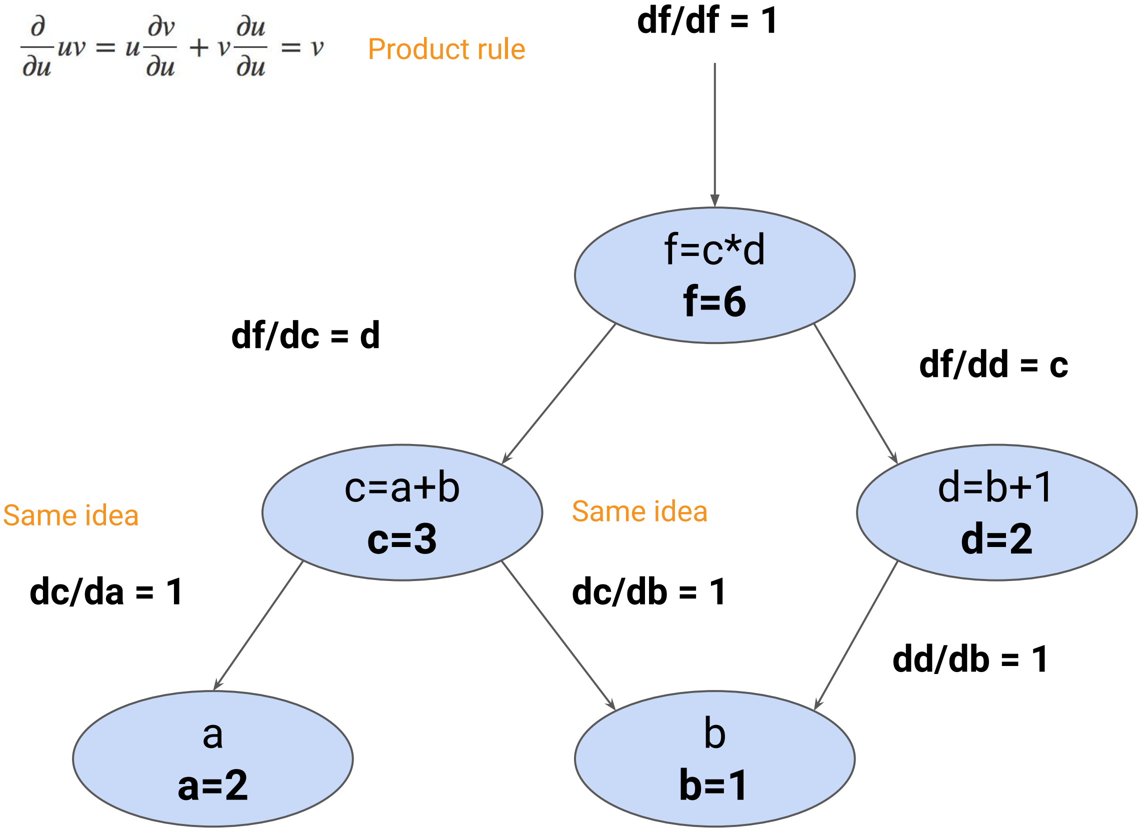
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ Product rule



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



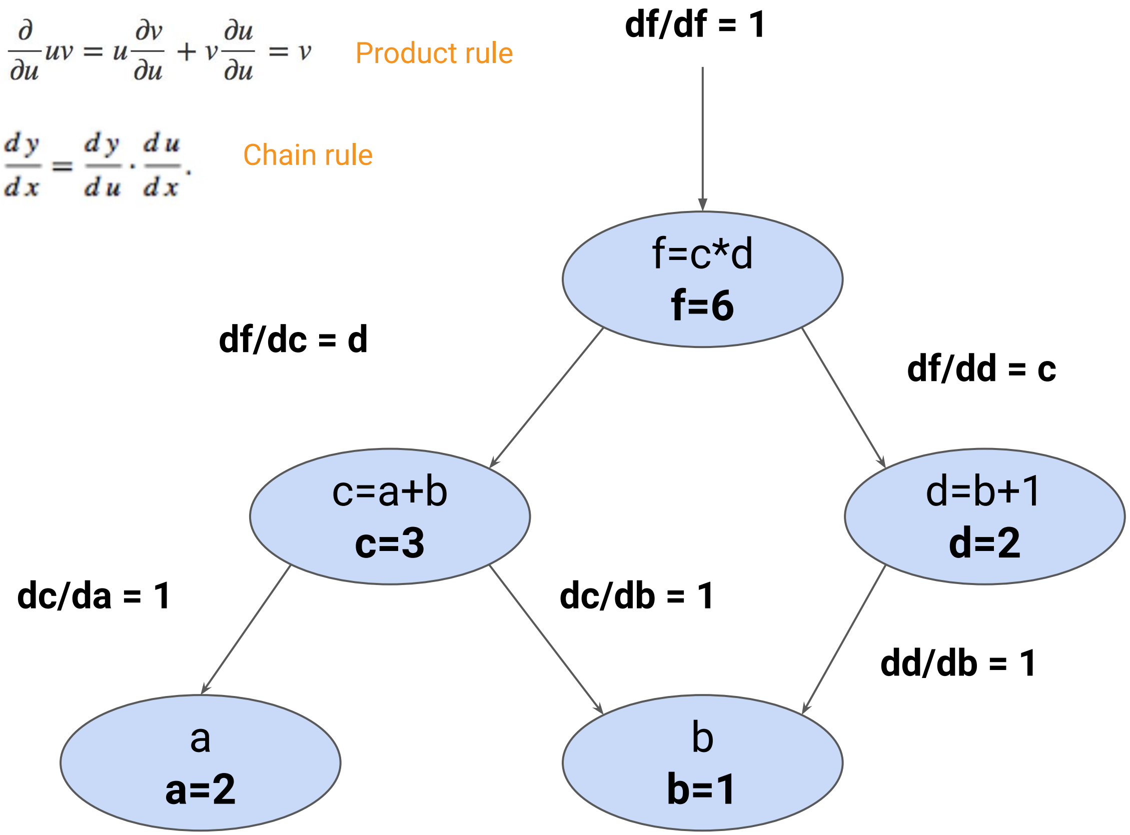
Sum rule. Intuition: if we increase b by a little, how much does d change? The same amount.

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

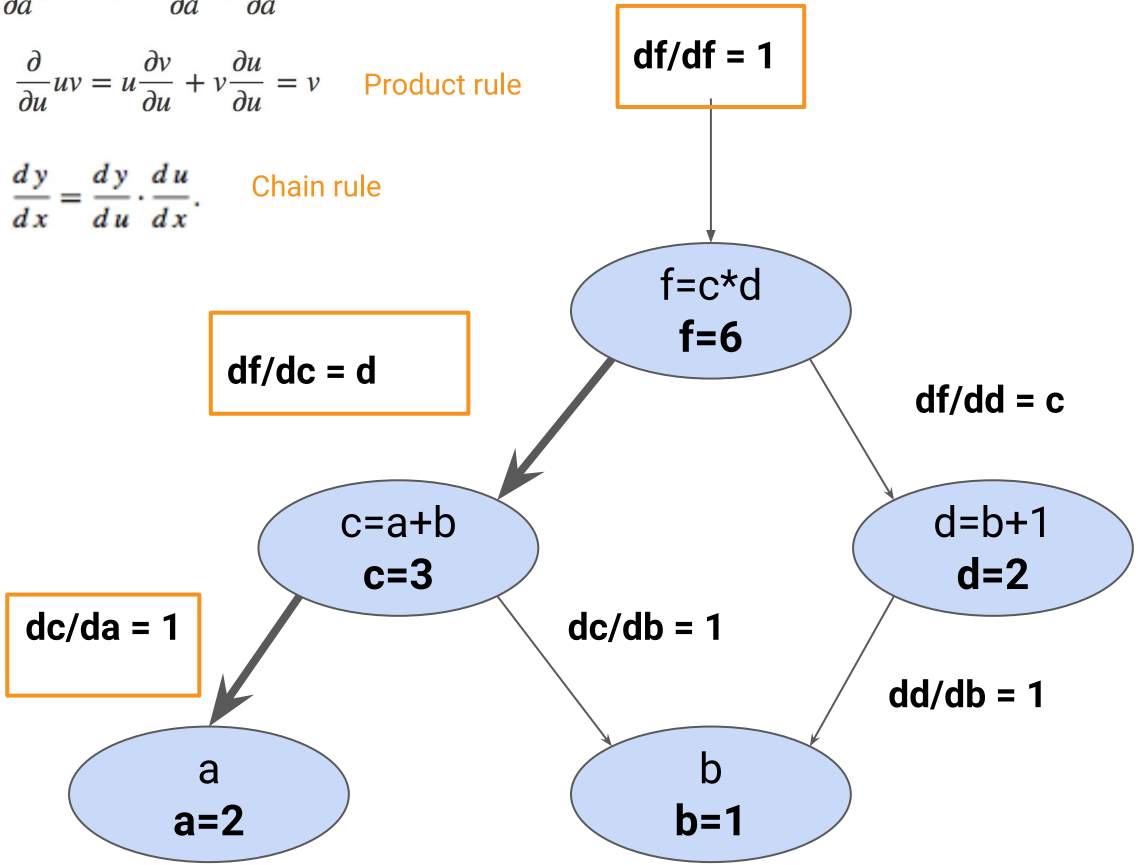
Now we can compute the gradient as the product along paths (another way of thinking about the chain rule!)



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

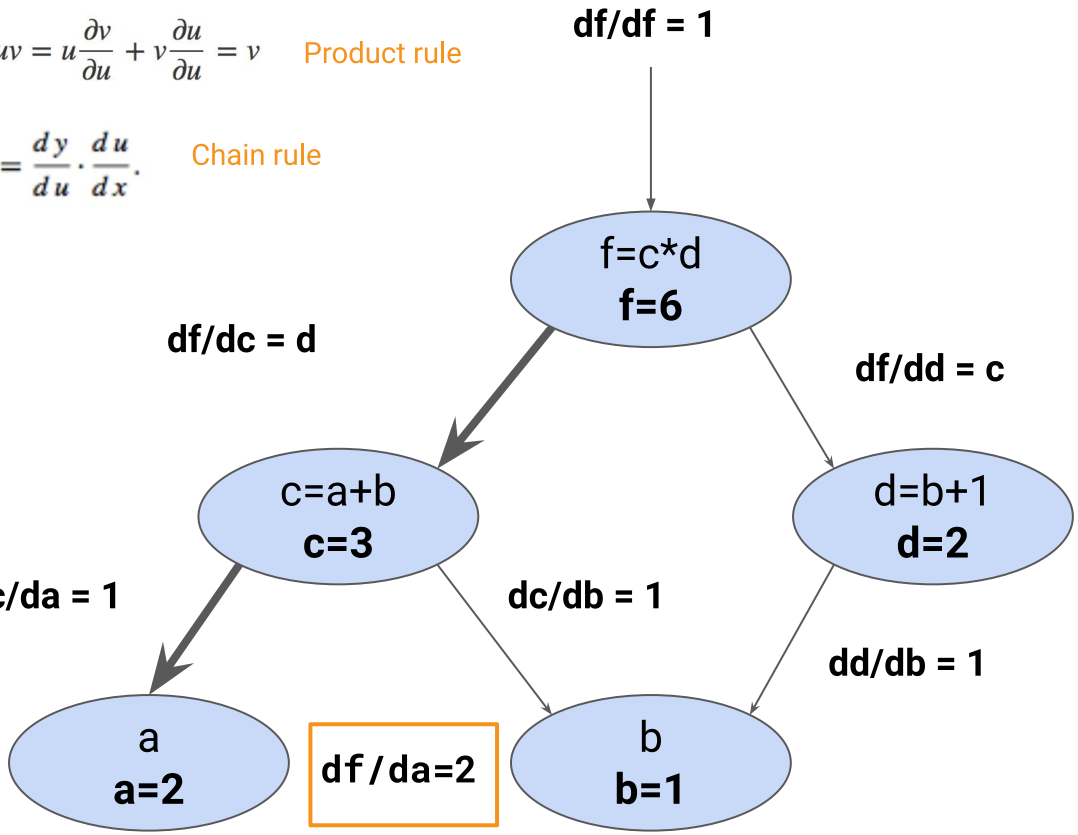
df/da = df/df * df/dc * dc/da

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

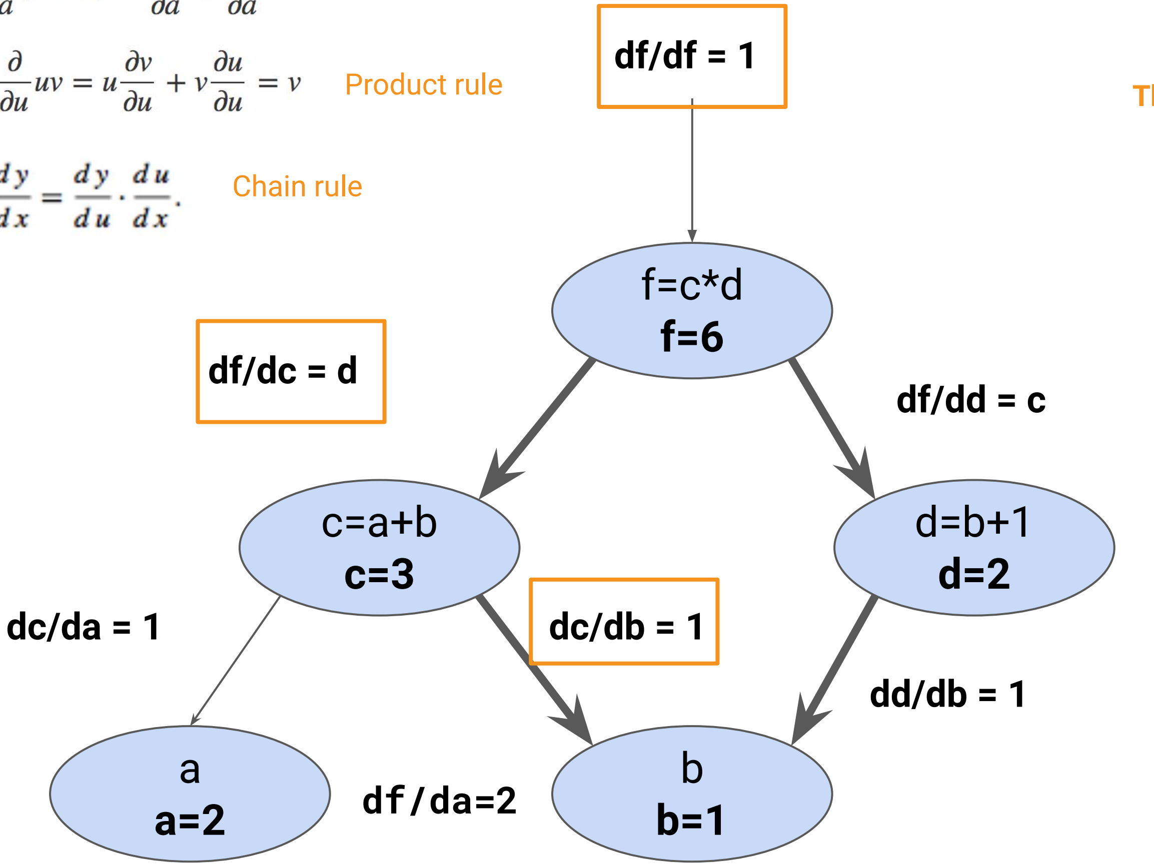
$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ 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.



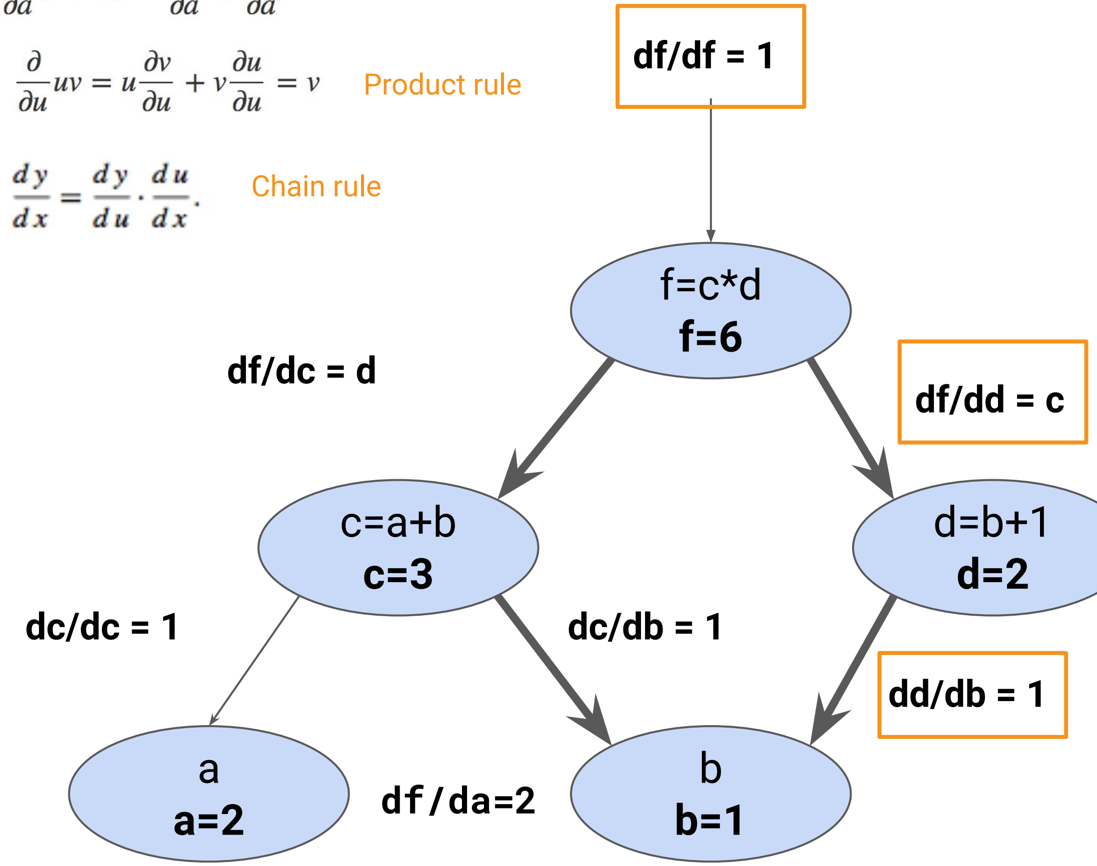
$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ 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.

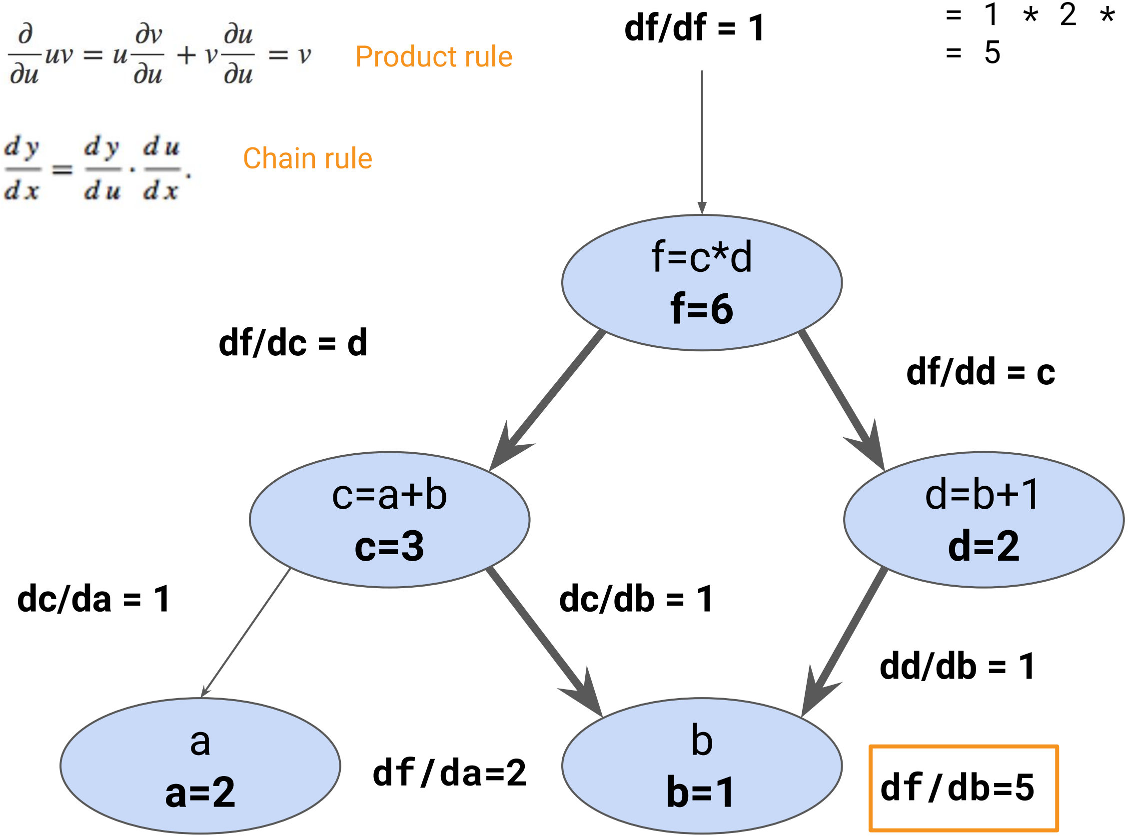


$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$ Sum rule

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$ 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 at once**

Dynamic programming -> linear in the number of edges on the graph.

Insight from Chris that's worth reading

"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 only needs the gradient of it's parent, and basic calculus rules - no knowledge of the overall graph is required).

Reading

Gradient descent

- An overview of gradient descent optimization algorithms
- Deep Learning: 4.3, 8.3 (Basic Algorithms); 8.4 (Parameter Initialization Strategies); 8.5 (Algorithms with Adaptive Learning Rates)

Backprop

- Calculus on Computational Graphs
- Course notes on backprop from CS231n (they're excellent)
- Yes you should understand backprop (same author as above)

Papers

- Understanding the difficulty of training deep feedforward neural networks