Gradient Descent, Properties of the Gradient, Backpropagation

Alexander Binder

University of Oslo (UiO)

February 16, 2023



UiO : Department of Informatics
University of Oslo

Learning goals

- **?!** directional derivatives and their relationship to the gradient
- **?!** multilinear algebra: derivatives of multi-linear functions and the product rule

Learning goals

- gradient descent as a vanilla solution strategy to find parameters in machine learning
- be able to explain the impact of large or small learning rates on the performance of gradient descent
- be able to explain the impact of different initialization points on the performance of gradient descent, and for what problems different initializations do not matter
- Batch vs Stochastic gradient descent

Outline

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

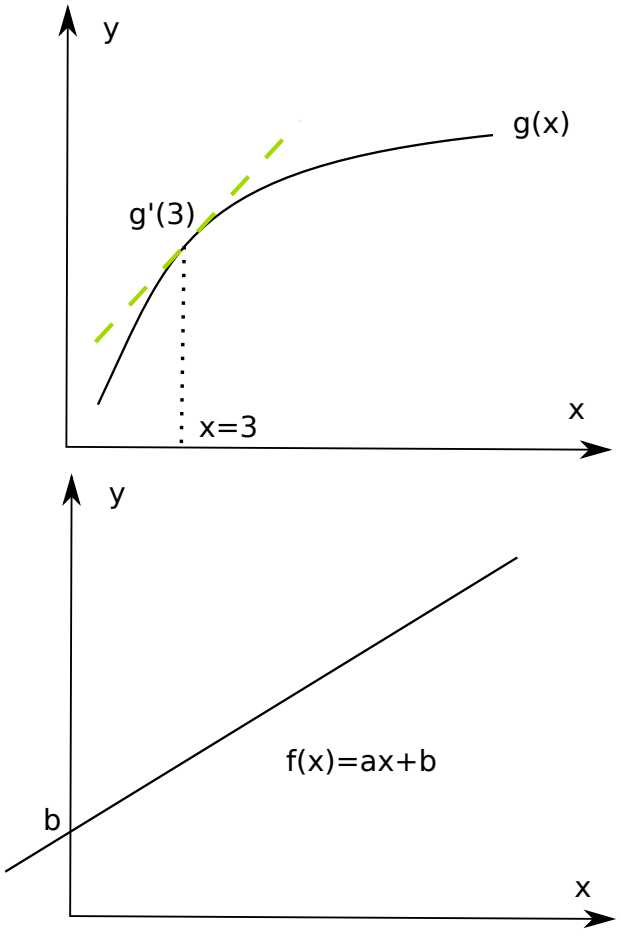
- $g : \mathbb{R}^1 \to \mathbb{R}^1$ is differentiable in input x if the limit exists:

$$\lim_{\epsilon \to 0} \frac{g(x+\epsilon) - g(x)}{\epsilon} \; (=: g'(x))$$

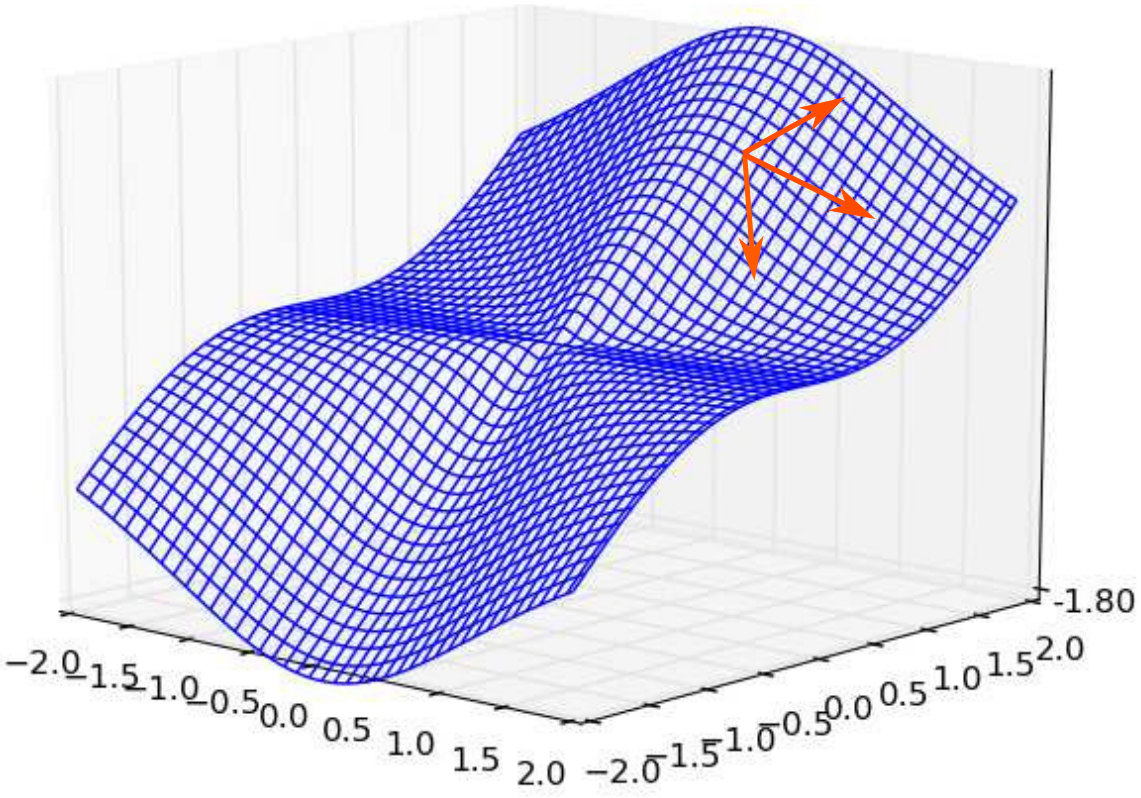
- example: $f(x) = ax + b$ (affine with slope a),then

$$\frac{f(x+\epsilon) - f(x)}{\epsilon}$$
$$= \frac{a(x+\epsilon) + b - (ax+b)}{\epsilon}$$
$$= \frac{a\epsilon}{\epsilon} = a$$
$$\Rightarrow \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} = a$$

- intuition: slope of the function g at point x



Function of 2 input variables: $f(x_1, x_2) \in \mathbb{R}^1$

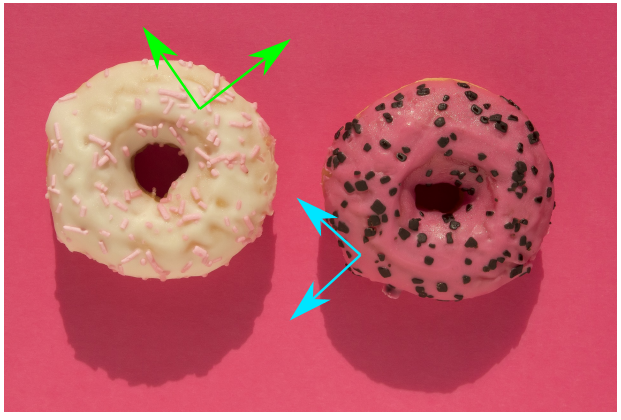


in every point (x_1, x_2): a two dimensional vector space of directions to move from it

in every direction there is a slope – the directional derivative

Function of n input variables: $f(x_1, x_2, \ldots, x_n) \in \mathbb{R}^1$

Example:



The surface of a donut is two dimensional at every point. At every point there is a two-dimensional space of directions to move, each with a slope.

Now take the product space of two donut surfaces. It consists of all pairs (p_1, p_2) such that $p_1 \in$ white donut surface, $p_2 \in$ red donut surface. At every pair (p_1, p_2) - the set of all directions is $n = 4$-dimensional!

in every point $(x_1, x_2, \ldots, x_n)$: a n-dimensional vector space of directions to move from it. In every direction there is a slope – the directional derivative – provides information about function value change in this direction

The directional derivative of function f in point x in direction v is defined as:

$$\delta_{\mathbf{v}} f(\mathbf{x}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{v}) - f(\mathbf{x})}{\epsilon}$$

Fact: If the function is differentiable in x, then the directional derivative in $\mathbf{x}$ in direction $\mathbf{v}$ is the inner product of the gradient of x in v:

$$\delta_{\mathbf{v}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

- directional derivatives tell you how the function grows from x in direction v when you take an infinitely small step
- the gradient contains information about all directional derivatives, if differentiable in x

- next step: define gradient via partial derivatives

$$\begin{aligned}\nabla f(\mathbf{x}) \cdot e_i &= \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon e_i) - f(\mathbf{x})}{\epsilon} \\ &= \lim_{\epsilon \to 0} \frac{f(x_1, \ldots, x_i + \epsilon, \ldots, x_D) - f(x_1, \ldots, x_i, \ldots, x_D)}{\epsilon} \\ &= \frac{\partial f}{\partial x_i}(\mathbf{x})\end{aligned}$$

therefore:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_D}(\mathbf{x}) \end{pmatrix}$$

therefore:

$$\delta_{\mathbf{v}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v} = \sum_d \frac{\partial f}{\partial x_d}(\mathbf{x}) v_d$$

Further consequence from:

$$\delta_{\mathbf{v}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

Which v maximizes $\nabla f(\mathbf{x}) \cdot \mathbf{v}$?

the optimization problem

$$\operatorname{argmax}_{v:\|v\|=1} w \cdot v$$

is solved by $v := \frac{w}{\|w\|}$.

Therefore: The gradient is the direction where the function increases maximally when taking an infinitesimally small step.

Analogously: the negative gradient is the direction where the function **decreases maximally when taking an infinitesimally small step**.

Outline

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

Why do you need that?

examples:

$$X \in \mathbb{R}^{d \times k}, \; D_w(X \cdot w) = X \text{ ok, easy}$$

$$D_w((Xw)^T \cdot (Xw)) = X^T Xw + w^T X^T X \text{ ?? shape mismatch !}$$

$$= (k, 1) + (1, k) = ???$$

$$A \in \mathbb{R}^{k \times d}, W \in \mathbb{R}^{d \times d}, C \in \mathbb{R}^{d \times m},$$

$$D_w(AWC) = A \,[\text{a hole here?}]\, C = \text{ nonsense ???}$$

Consider viewpoint directional derivatives:

$$Df(x)[h] = \nabla f(x) \cdot h$$

The derivative of f in x is a linear mapping of directions onto directional derivatives.
Rule 1: the derivative of a linear function is the linear function itself.

$$f(x) \text{ linear} \Rightarrow Df(x)[h] = f(h)$$
$$f(x) = Ax \Rightarrow Df(x)[h] = Ah$$

This holds also if f maps into vectors or matrices.

Rule 1: the derivative of a linear function is the linear function itself.

$$f(x) \text{ linear} \Rightarrow Df(x)[h] = f(h)$$

This holds also if f maps into vectors or matrices. Why?

$$\begin{aligned}
f(x) &= Ax \\
\Rightarrow f_j(x) &= (Ax)_j = \sum_k A_{jk} x_k \\
\frac{\partial f_j}{\partial x_m} &= A_{jm} \\
\nabla f_j(x) &= (A_{j1}, A_{j2}, \ldots, A_{jm}, \ldots A_{jd}) \\
Df_j(x)[h] &= \nabla f_j(x) \cdot h = \sum_k A_{jk} h_k = (Ah)_j \\
\Rightarrow Df(x)[h] &= Ah
\end{aligned}$$

$$D_w(AWC)[H] = ???$$

$$f(W) = AWC \text{ is linear in } W \text{ so:}$$
$$D_w(AWC)[H] = f(H) = AHC$$

its a no brainer!

You need a partial derivative?

$$W \in \mathbb{R}^{d \times d}, \Rightarrow \frac{\partial f}{\partial W_{ij}}(W) = A\mathbb{1}_{ij}C$$

$\mathbb{1}_{ij}$ is the matrix which is 1 in entry (i,j) and zero everywhere else

Rule 2: product rule applicable for any vector/matrix-vector/matrix multiplication

$$D_x(f(x) \cdot g(x))[h] = (Df(x)[h]) \cdot g(x) + f(x) \cdot (Dg(x)[h])$$

$$\begin{aligned} &= Df(x)[h] \cdot g(x) \\ &+ f(x) \cdot Dg(x)[h] \end{aligned}$$

This extends to multiplications of more than two terms:

$$\begin{aligned} D_x(f(x) \cdot g(x) \cdot r(x))[h] = & \\ &= Df(x)[h] \quad \cdot g(x) \quad \cdot r(x) \\ &+ f(x) \quad \cdot Dg(x)[h] \quad \cdot r(x) \\ &+ f(x) \quad \cdot g(x) \quad \cdot Dr(x)[h] \end{aligned}$$

$$D_w(w^T A w)[h] = ?$$

$$D_w(w^T A w)[h] = ?$$

Apply product rule to $w^\top * Aw$

$$
\begin{aligned}
D_w(w^T A w)[h] &= D_w(w^T \cdot Aw)[h] \\
&= D_w(w^T)[h] \cdot Aw + w^T \cdot D(Aw)[h] \\
&\text{both are linear mappings!} \\
D_w(w^T)[h] &= h^T \\
D(Aw)[h] &= Ah \\
\Rightarrow D_w(w^T A w)[h] &= h^T Aw + w^T Ah \\
D_w(w^T A w)[h] &= D_w(w^T A \cdot w)[h] \text{ same solution}
\end{aligned}
$$

Remember linear regression

Assumption for now: no bias. Parameters are w.

Goal: to find parameters which minimize the loss on this dataset:

$$(w^*) = \text{argmin}_w \sum_{i=1}^{n} L(f(x_i), y_i) = \text{argmin}_w \sum_{i=1}^{n} (x_i \cdot w - y_i)^2$$

for a given dataset $D_n = \{(x_i, y_i)\}$. Then $f_{w^*}(x) = x \cdot w^*$ is the selected mapping. Rare case: Can be solved explicitly for w.

- consider the loss L as function of w
- compute $\nabla L(w)$ - the gradient of the loss with respect to w
- solve $\nabla L(w) = 0$ for w.
- Solution can be maximum, minimum or saddle point. Verify that the Hessian in the solution point is positive definite. That is: the function has positive curvature in every direction. implies: must be a minimum.

Write in matrix form:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1^{(1)}, \ldots, x_1^{(D)} \\ x_2^{(1)}, \ldots, x_2^{(D)} \\ \vdots \\ x_n^{(1)}, \ldots, x_n^{(D)} \end{pmatrix} \in \mathbb{R}^{n \times D}$$

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

then:

$$L(w) = \sum_{i=1}^{n} (x_i \cdot w - y_i)^2 = (X \cdot w - Y)^T \cdot (X \cdot w - Y)$$

$$D_w((Xw - Y)^T(Xw - Y)) = ?$$

$$D_w((Xw - Y)^T \cdot (Xw - Y))[h] = D_w((Xw - Y)^T)[h] \cdot (Xw - Y) + (Xw - Y)^T \cdot D_w(Xw - Y)[h]$$

$$D_w((Xw - Y)^T)[h] = D_w((Xw)^T)[h] + D_w((-Y)^T)[h] = (Xh)^T + 0$$

$$\Rightarrow D_w((Xw - Y)^T \cdot (Xw - Y))[h] = (Xh)^T \cdot (Xw - Y) + (Xw - Y)^T \cdot (Xh)$$

Note both terms are real numbers:
$X \sim (n, d), w \sim (d, 1), Y \sim (n \times 1)$
$\Rightarrow Xw - Y \sim (n \times 1), (Xw - Y)^T \cdot (Xw - Y) \sim (1, n) \cdot (n, 1) = (1, 1)$

$$D_w((Xw - Y)^T(Xw - Y)) = ?$$

$$D_w((Xw - b)^T \cdot (Xw - Y))[h] = (Xh)^T \cdot (Xw - Y) + (Xw - Y)^T \cdot (Xh)$$

Note both terms are real numbers: $X \sim (n, d), w \sim (d, 1), Y \sim (n \times 1)$ therefore they are their own transpose!

$$\begin{aligned} D_w((Xw - b)^T \cdot (Xw - Y))[h] &= (Xh)^T \cdot (Xw - Y) \\ &\quad + ((Xw - Y)^T \cdot (Xh))^T \\ &= (Xh)^T \cdot (Xw - Y) \\ &\quad + (Xh)^T \cdot (Xw - Y) \\ = 2(Xh)^T \cdot (Xw - Y) &= h^T \cdot 2X^T(Xw - Y) \end{aligned}$$

$$\begin{aligned} D_w((Xw-b)^T \cdot (Xw-Y))[h] &= (Xh)^T \cdot (Xw-Y) \\ &\quad + ((Xw-Y)^T \cdot (Xh))^T \\ = 2(Xh)^T \cdot (Xw-Y) &= h^T \cdot 2X^T(Xw-Y) \end{aligned}$$

This is a linear operation in h: $h \mapsto h^T \cdot 2X^T(Xw-Y)$,

Therefore the gradient is the transpose of $2X^T(Xw-Y)$

Solve the minimization problem by computing the gradient for w and setting it to zero.

$$
\begin{aligned}
D_w((X \cdot w - Y)^T \cdot (X \cdot w - Y)) &= 2X^T(X \cdot w - Y) \\
2X^T(X \cdot w - Y) &= 0 \\
\Rightarrow (X^T X) \cdot w - X^T \cdot Y &= 0 \\
\Rightarrow (X^T X) \cdot w &= X^T \cdot Y \\
w &= (X^T \cdot X)^{-1} X^T \cdot Y
\end{aligned}
$$

if the matrix inverse $(X^T \cdot X)^{-1}$ exists.

$$L(w) = \sum_{i=1}^{n} (x_i \cdot w - y_i)^2 + \frac{\lambda}{2} \|w\|_2^2$$

$$= (X \cdot w - Y)^T \cdot (X \cdot w - Y) + \frac{\lambda}{2} w^T \cdot w$$

Gradient analogously:
This is a linear operation in h: $h \mapsto h^T \cdot 2X^T(Xw - Y) + h^T \cdot \lambda w$,

Therefore the gradient is the transpose of $2X^T(Xw - Y) + \lambda w$

$$
\begin{aligned}
2X^T \cdot (X \cdot w - Y) + \lambda w &= 0 \\
\Rightarrow (X^T X) \cdot w - X^T \cdot Y + \lambda \mathbb{I} w &= 0 \\
\Rightarrow (X^T \cdot X + \lambda \mathbb{I}) \cdot w &= X^T \cdot Y \\
w &= (X^T \cdot X + \lambda \mathbb{I})^{-1} X^T \cdot Y
\end{aligned}
$$

Advantage of ridge regression: This inverse always exists for $\lambda > 0$. Why ?

Outline

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

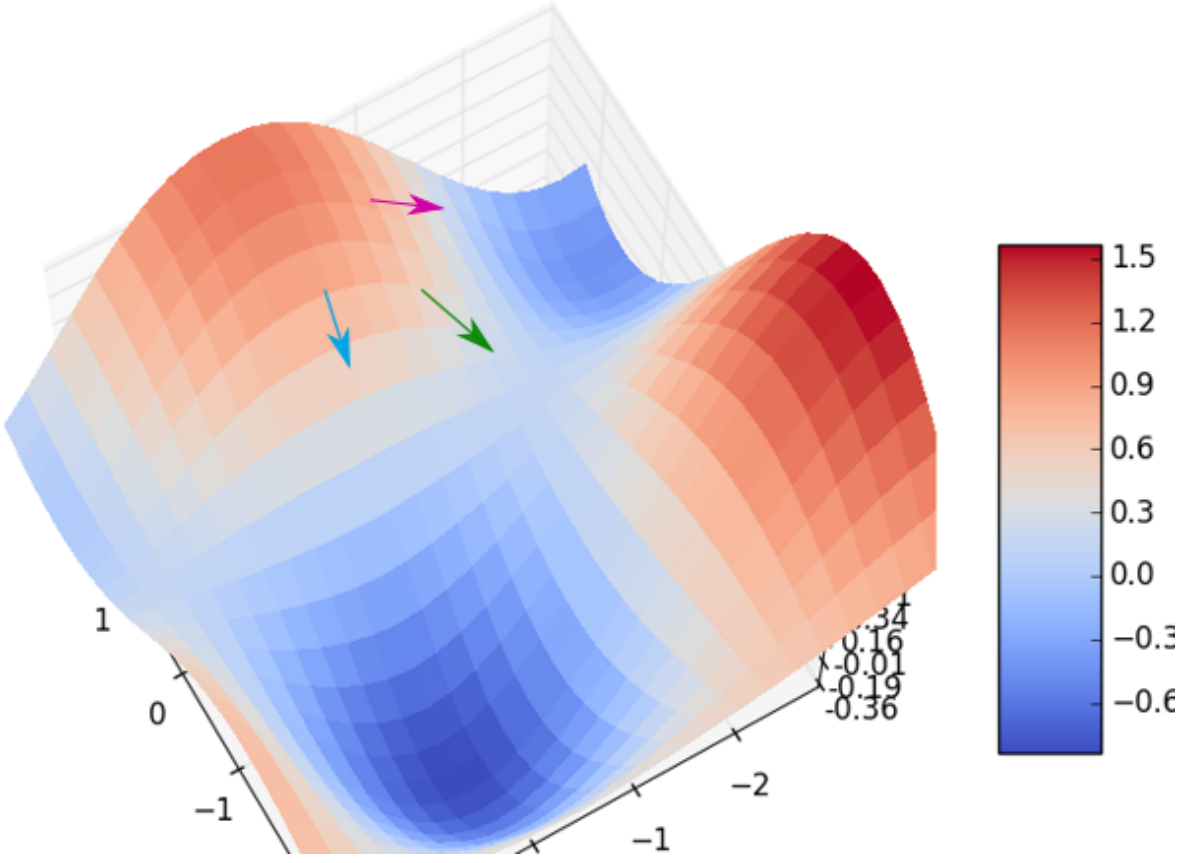
How to find these parameters (w, b) in classification or regression?

Problem setting for application of gradient-based minimization

The Problem Setting, in which gradient methods can be used:

- given some function $g(w)$
- goal: find $w^* = \text{argmin}_w g(w)$
- assumption: can compute $\nabla g(w)$ - the gradient in point w.

Idea: negative gradient at a point w is the direction of locally steepest function decrease from w.



Note: the direction of locally steepest decrease does not point to a global or local minimum.

Gradient Descent

Basic Algorithm: name: **Gradient Descent**:

- given: step size parameter η, initialize start vector w_0 as something
- run while loop, until function value changes very little (δ_g), do at iteration t:
  - $w_{t+1} = w_t - \eta \nabla_w g(w_t)$
  - compute change to last value: $\delta_g = \|g(w_{t+1}) - g(w_t)\|$

Consequences I:

- minimizing the gradient on training data ensures low loss on training data
- does not guarantee low losses on new unseen test data
- a gap between training and test loss is known as overfitting (training loss below test loss).

Minimize

$$L(w, b) = \frac{1}{n} \sum_{(x_i, y_i) \in D} \max(0, 1 - y_i f_{w,b}(x_i))$$

Use $L(w, b)$ with the algorithm box above ... gradient descent with respect to (w, b) for obtaining $(w^*, b^*) = \operatorname{argmin}_{w,b} L(w, b)$

Gradient Descent

Basic Algorithm: name: **Gradient Descent**:

- given: step size parameter η, initialize start vector w_0 as something
- run while loop, until function value changes very little (δ_g), do at iteration t:
  - $w_{t+1} = w_t - \eta \nabla_w g(w_t)$
  - compute change to last value: $\delta_g = \|g(w_{t+1}) - g(w_t)\|$

- the initialization of w_0 matters a lot in deep neural networks! see later exercise. E.g. https://arxiv.org/abs/1502.01852 "Kaiming-He"-initialization ... related to vanishing gradients problem, symmetry breaking

Gradient Descent

Basic Algorithm: name: **Gradient Descent**:

- given: step size parameter η, initialize start vector w_0 as something
- run while loop, until function value changes very little (δ_g), do at iteration t:
  - $w_{t+1} = w_t - \eta \nabla_w g(w_t)$
  - compute change to last value: $\delta_g = \|g(w_{t+1}) - g(w_t)\|$

When does it converge ? When the change δ_g is small, that is when $\eta \|\nabla_w g(w_t)\|$ becomes small. That means $\nabla_w g(w_t) \approx 0$, so a local optimum. Since we always went down, it must be local minimum, or a saddle point.

possible problems of gradient descent:

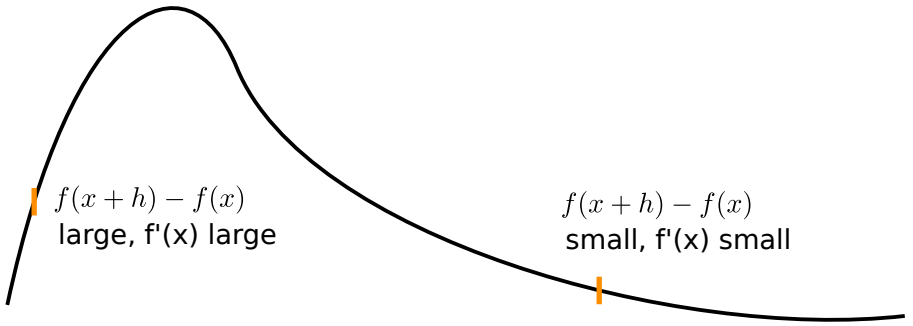
- we find a local minimum, not the global minimum of a function, can be good or bad.
- effects of bad stepsize: divergence – no solution, or slow convergence
- effects of starting point – which ones?

Lets explore these effects:

- in `learnThu8.py` run `tGD([stepsize])` to see the effect of different stepsizes. Why a too large stepsize can lead to numeric overflows? This is a common effect in deep learning training: too large stepsize, then training error will not go down.
- run `tGD2([initvalue])` with *initvalue* $\in [-4, +4]$ to see the effect of a constant stepsize, but different starting points – see in what minimum you end up.

Two Properties and how to deal with them:

1. convergence to global optimum only if the function is convex and the stepsize is sufficiently small. In general one reaches only local minima, sometimes saddle points.
2. the size of the update step $w_{t+1} = w_t - \eta \nabla_w g(w_t)$ depends on the norm of the gradient, too. So when starting in a steep region, even a small stepsize can bring trouble.



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

the gradient stepsize depends not only on the stepsize parameter but also on the norm of the gradient

learning rate adjustment schemes / learning rate annealing schemes

In practice: one starts with a learning rate, and decreases it over time, either with a polynomial decrease, or by a factor every N iterations.

$$\text{polynomial: } \lambda(t) = c_0 * \quad (t+1)^{-\alpha},\ \alpha > 0$$

$$\text{regular step at each } T\text{: } \lambda(t) = c_0 * \qquad c^{\lfloor t/T \rfloor},\ c \in (0,1)$$

This enforces convergence (not necessarily to a good point).[a]

in code:
pytorch: `torch.optim.lr_scheduler`

[a] What happens if one decreases the learning rate very fast?

Outline

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

Consider setting with an average of losses over training samples:

$$\frac{1}{n}\sum_{i=1}^{n}\ell(f(x_i), y_i)$$

The application of vanilla gradient descent to this function results in the following algorithm:

$$w_{t+1} = w_t - \eta\nabla_w\left(\frac{1}{n}\sum_{i=1}^{n}\ell(f_{w_t}(x_i), y_i)\right)$$

This is called **batch gradient descent** because it uses the set of **all training data samples** to compute the gradient in each step.

The alternative is **stochastic gradient descent** (SGD).

stochastic gradient descent computes in every iteration a gradient using only one sample every iteration, or, it uses a mini-batch like $k = 64$ samples – such that this set is chosen randomly from the set of all training samples. This is the default in deep learning.

stochastic gradient descent for an average of losses

The core idea of **Stochastic gradient descent** is to compute the gradient only over a randomly selected subset of samples. Stochastic gradient descent when starting at index m and using the next k samples is:

$$\nabla_w \frac{1}{k} \sum_{i=m+0}^{m+k-1} \ell(f_w(x_i), y_i)$$

stochastic gradient descent for an average of losses

use a randomly selected subset of samples:

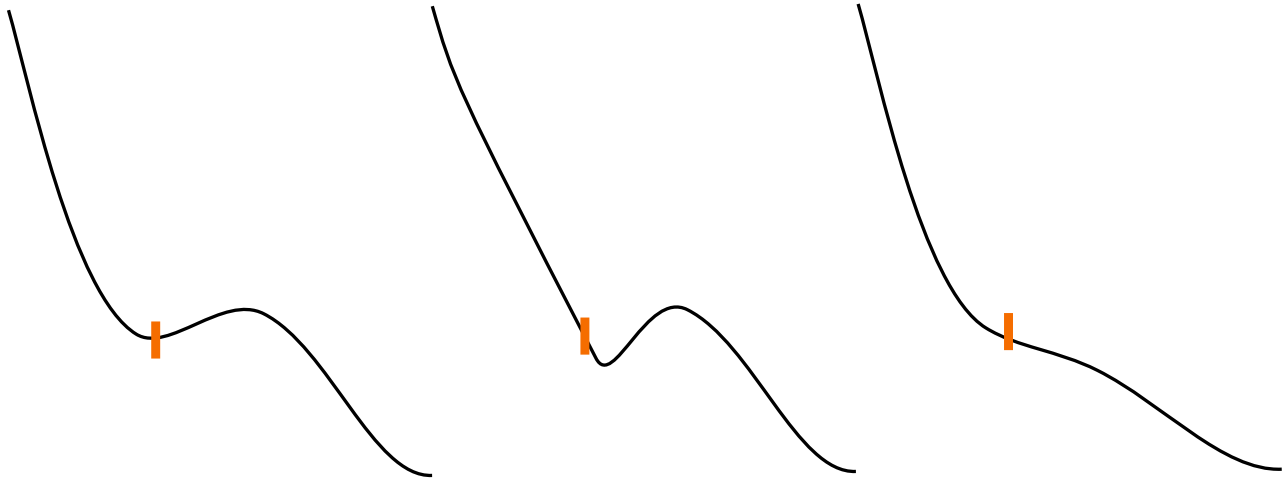
$$\nabla_w \frac{1}{k} \sum_{i=m+0}^{m+k-1} \ell(f_w(x_i), y_i)$$

- initialize start vector w_0 as something, choose step size parameter η
- run while loop, until function value changes very little (δ), do at iteration t:
  - select a random subset of k samples (usually by a random ordering of all training samples)
  - $w_{t+1} = w_t - \eta \nabla_w (\frac{1}{k} \sum_{i=m+0}^{m+k-1} \ell(f_w(x_i), y_i))$
  - compute change to last value: $\delta = \frac{1}{k} \| \sum_{i=m+0}^{m+k-1} \ell(f_{w_{t+1}}(x_i), y_i)) - \ell(f_{w_t}(x_i), y_i)) \|$

Advantages of stochastic gradient descent

- Full-batch is often too costly to compute a gradient using all samples when its more than tens of thousands
- SGD is a noisy, approximated version of the batch gradient.
- injecting small noise is one way to prevent overfitting! Sometimes SGD can be better than full batch gradient descent in finding *good* local optima.

- An illustration why noise to the loss function (e.g. by randomized sampling of training batches) may help to jump out of bad local optima



Left: a loss surfaces at some point (orange). Middle and Right: changes in the loss surfaces as different training data subsets are used. In the middle the gradient norm is much larger compared to the left case – allows to jump out of the local minimum.

- Insight: stochastic gradient descent is a noisy approximation (subset of all samples) to the batch gradient.

The batch gradient can be seen as an expected value: non-zero probability only for our training data points $(x_i, y_i) \in D_n$

$$P_{D_n}(x, y) = \begin{cases} \frac{1}{n} & \text{if } (x, y) = (x_i, y_i) \text{ for } (x_i, y_i) \in D_n \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \nabla_w \frac{1}{n} \sum_{(x_i, y_i) \in D_n} \ell(f_w(x_i), y_i) &= \sum_{(x_i, y_i) \in D_n} \nabla_w \ell(f_w(x_i), y_i) \ \frac{1}{n} \\ &= \sum_{(x_i, y_i) \in D_n} \nabla_w \ell(f_w(x_i), y_i) P_{D_n}((x_i, y_i)) \\ &= E_{(x_i, y_i) \sim P_{D_n}}[\nabla_w \ell(f_w(x_i), y_i)] \end{aligned}$$

This is an expectation of a gradient function $\nabla_w \ell(f_w(x_i), y_i)$.

Remember here for a discrete set of values (x_i, y_i)

$$\sum_i r(x_i, y_i) P((x_i, y_i)) = E[r(x, y)]$$

When performing stochastic gradient descent, we use a subset of samples from D_n in every step for computing the gradient. Using only a subset of samples from D_n is an approximation to the expectation shown in the last slide!

$$E_{(x_i,y_i)\sim P_{D_n}}[\nabla_w \ell(f_w(x_i), y_i)] \approx \sum_{i=m+0}^{m+k-1} \nabla_w \ell(f_w(x_i), y_i) \frac{1}{k}$$

Approximation holds due to the central limit theorem! No convergence here[1]

[1] Why there is none?

$$E_{(x_i,y_i)\sim P_{D_n}}[\nabla_w \ell(f_w(x_i), y_i)] \approx \sum_{i=m+0}^{m+k-1} \nabla_w \ell(f_w(x_i), y_i) \frac{1}{k}$$

Approximation holds due to central limit theorem! No convergence here[2]

Recap on central limit theorem

If you have a population with mean μ and standard deviation σ and you draw n random samples from the population - statistically independently, then the distribution of the sample means will be **approximately** normally distributed **with mean** μ **and standard deviation** $\frac{\sigma}{\sqrt{n}}$

[2] Why there is none?

The only thing that is necessary for the expectation to hold is that

- drawing pairs (x_i, y_i), (x_k, y_k) is statistically independent

This noise from approximation can sometimes act as regularization – prevents to look at the data too closely.

Outline

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

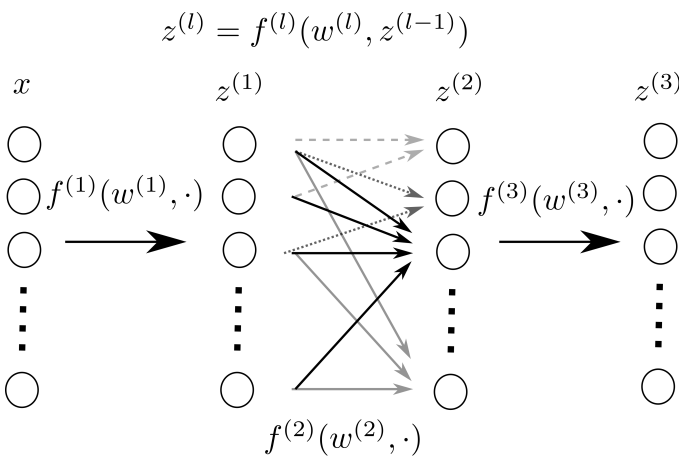
Learning goals

- be able to use the chain rule on functions
- Backpropagation as an algorithm to compute derivatives in a neural network (w.r.t. network parameters, and inputs)
- Backpropagation inside amounts to applying chain rule top-down along the graph structure of the network
- the backprop comes with an efficient mechanism to reuse computed partial derivatives for computing new derivatives further down in the network.
- you should be able to derive the derivative of a loss of a neural network as a sum-product of derivatives of single neurons with respect to their inputs and parameters

- A single neural network layer (fully connected / linear):

$$z^{(l)} = g(w^{(l)} \cdot z^{(l-1)} + b) = f(w^{(l)}, z^{(l-1)})$$
$$z^{(l-1)} \in \mathbb{R}^{d_1},\ w^{(l)} \in \mathbb{R}^{d_2 \times d_1},\ b \in \mathbb{R}^{d_2},\ z^{(l)} \in \mathbb{R}^{d_2},$$

- $g(w^{(l)} \cdot z + b)$ is the layer definition, $f(w^{(l)}, z^{(l-1)})$ is a notation to denote dependency of the definition on parameters $w^{(l)}$ and the input to the network layer $z^{(l-1)}$.

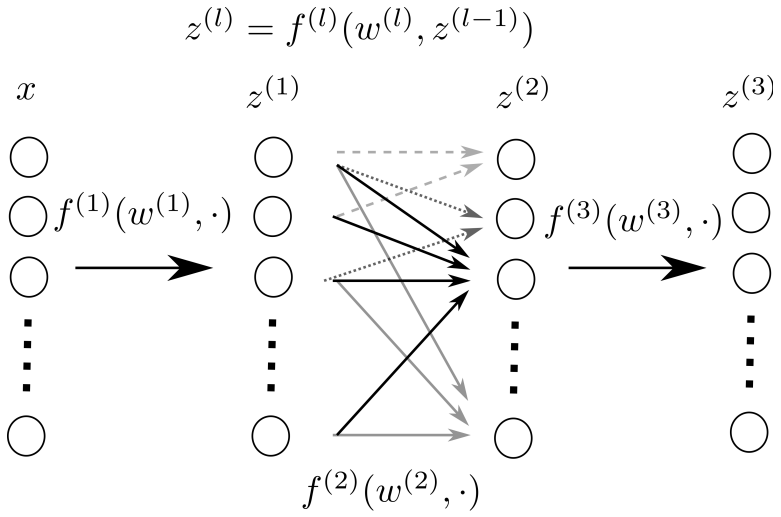


- A three layer network can be described as:

$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

- An iterative writing would be:

$$z^{(l)} = f^{(l)}(w^{(l)}, z^{(l-1)})$$



- A three layer network can be described as:

$$z^{(l)} = f^{(l)}(w^{(l)}, z^{(l-1)})$$
$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

- goal: to compute $\frac{\partial y}{\partial w_d^{(l)}}$ efficiently.

How to compute gradients generically for some function (if we had no autograd) ?

Finite difference method:

$$\frac{\partial f}{\partial w_d}(x, w) = \lim_{\epsilon \to 0} \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d)}{\epsilon} \tag{1}$$

$$\approx \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d)}{\epsilon} \tag{2}$$

better is however:

$$\frac{\partial f}{\partial w_d}(x, w) \approx \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d - \epsilon)}{2\epsilon} \tag{3}$$

Issue: for every partial derivative we have to make a call to the whole neural network!

How can we compute the derivative $\frac{\partial L}{\partial w_d}$ of a loss $L(y, y_{gt})$ with respect to parameters w_d once we have

- $\frac{\partial y_k}{\partial w_d}$
- and $\frac{\partial L}{\partial y_k}$?

How can we compute the derivative $\frac{\partial L}{\partial w_d}$ of a loss $L(y, y_{gt})$ with respect to parameters w_d once we have

- $\frac{\partial y_k}{\partial w_d}$
- and $\frac{\partial L}{\partial y_k}$?

$$\frac{\partial L}{\partial w_d} = \sum_k \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial w_d} \tag{4}$$

chain rule.

- 1-dim case: $x \in \mathbb{R}$, $g(x) \in \mathbb{R}$

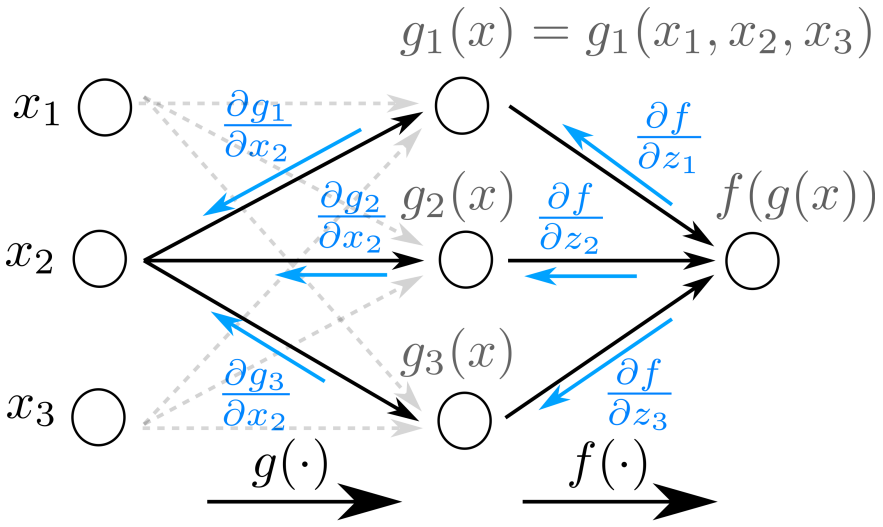
$$h(x) = f \circ g(x) = f(g(x))$$
$$\frac{\partial f \circ g}{\partial x}(x) = f'(g(x))g'(x) = \frac{\partial f}{\partial z}(g(z))\frac{\partial g}{\partial x}(x)$$

- n-dim case, when shown using partial derivatives:

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \ldots, z_m) \in \mathbb{R} \tag{5}$$
$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \ldots, x_d) \in \mathbb{R}^m \tag{6}$$
$$f \circ g(x) = f(g(x)) \tag{7}$$
$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x))\frac{\partial g_r}{\partial x_k}(x) \tag{8}$$

Contains a non-intuitive summing: derivatives over input components of f and over output components of g.

Recovering the chainrule by writing $f(g(x))$ as a graph



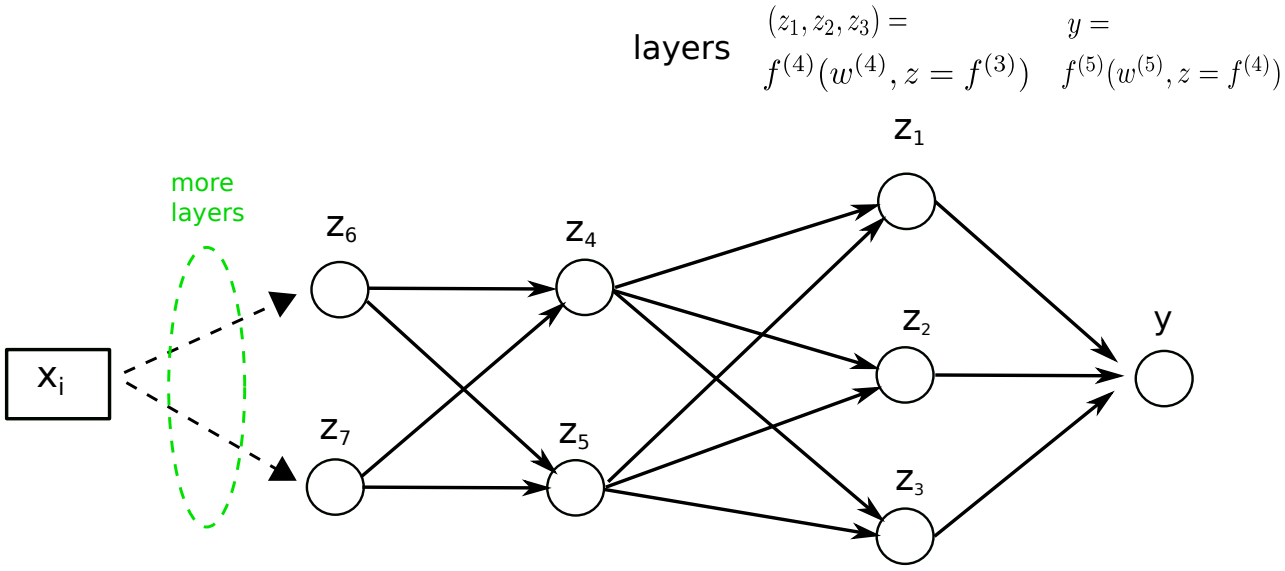
$$\frac{\partial f \circ g}{\partial x_2}(x) = \left.\frac{\partial f}{\partial z_1}\right|_{z=g(x)} \frac{\partial g_1}{\partial x_2} + \left.\frac{\partial f}{\partial z_2}\right|_{g(x)} \frac{\partial g_2}{\partial x_2} + \left.\frac{\partial f}{\partial z_3}\right|_{g(x)} \frac{\partial g_3}{\partial x_2}$$

Three steps:

- we assign to an edge $z_i \mapsto h(z_i, \text{other vars})$ the term: $\frac{\partial h}{\partial z_i}$
- we multiply all edge terms along a backward path $x_2 \to y$
- at a node x_2 we sum terms from all paths

Backpropagation computes gradients via chainrule along the (directed) edges in the neural network graph.
Backprop = chainrule on a DAG

Executing chainrule along a directed graph.

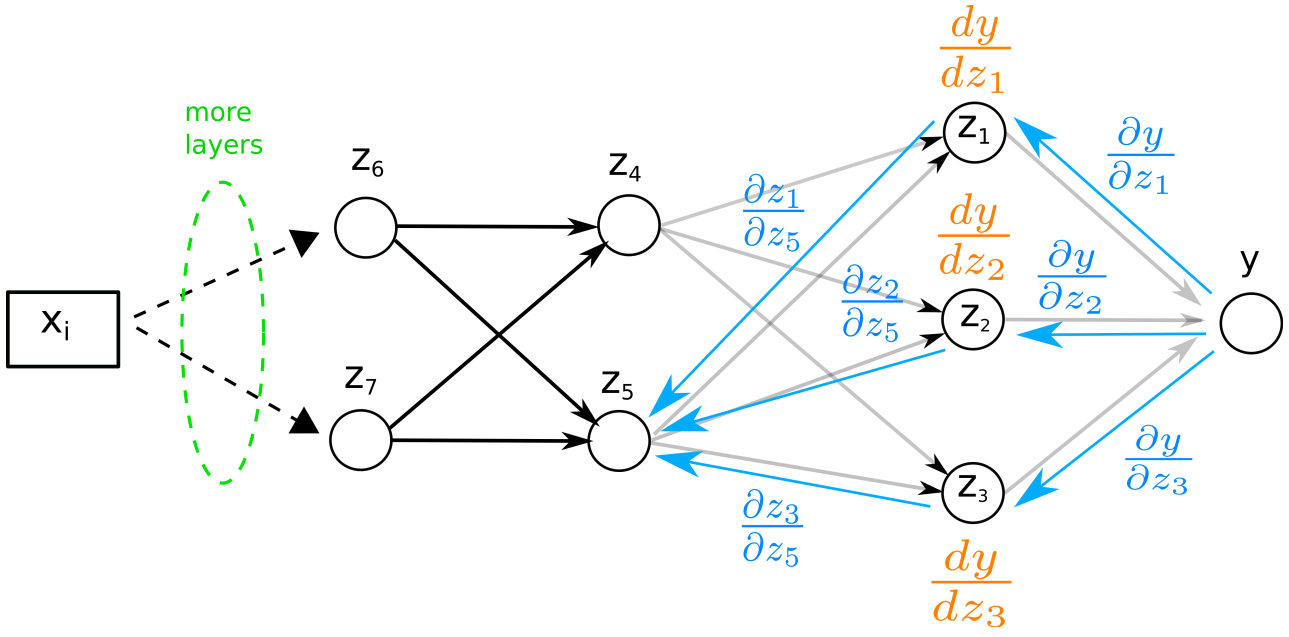


$$\frac{\partial y}{\partial z_1} = \text{ ? ... compute directly}$$

$$y = \sum_i w_i z_i + b$$

$$\frac{\partial y}{\partial z_i} = w_i \text{ for } i = 1, 2, 3$$

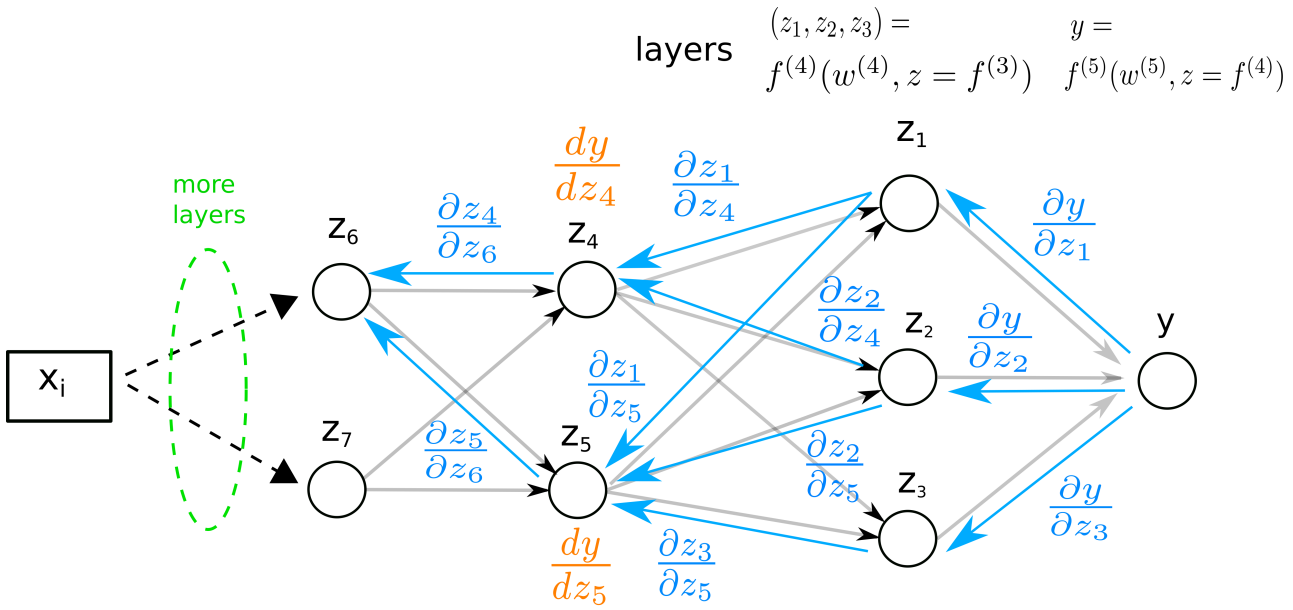
Executing chainrule along a directed graph.



$\frac{\partial y}{\partial z_5} = ???$ $\frac{\partial y}{\partial z_5} = \frac{dy}{dz_1}\frac{\partial z_1}{\partial z_5} + \frac{dy}{dz_2}\frac{\partial z_2}{\partial z_5} + \frac{dy}{dz_3}\frac{\partial z_3}{\partial z_5}$

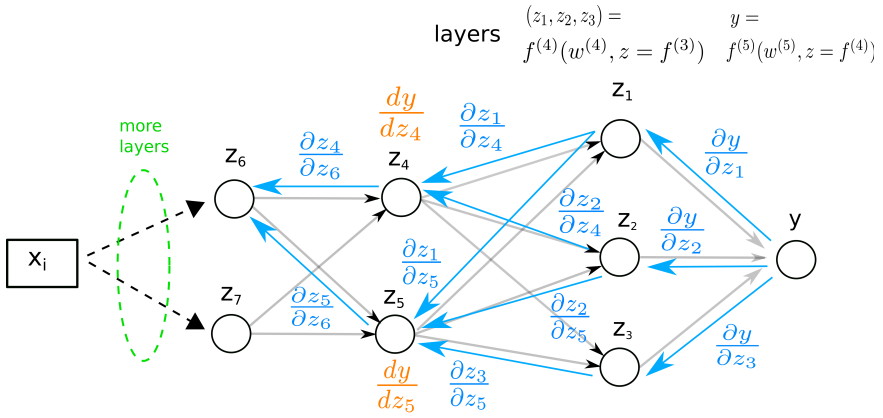
- flow of grads: each edge $z_k \to z_i$ has a partial derivative $\frac{\partial z_i}{\partial z_k}$ flowing backwards
- each path (e.g. $z_4 \to y$) is the product of its associated edge terms $\frac{\partial z_i}{\partial z_k}$.
- each node z_k has a total derivative $\frac{dy}{dz_k}$ accumulated

Executing chainrule along a directed graph.



$\frac{\partial y}{\partial z_6}$ =??? The flow principle continues: $\frac{dy}{dz_6} = \frac{\partial z_4}{\partial z_6}\frac{dy}{dz_4} + \frac{\partial z_5}{\partial z_6}\frac{dy}{dz_5}$

- how to use it: walk backwards along the graph edges and compute layer by layer
- at first $\frac{\partial y}{\partial z_1}, \frac{\partial y}{\partial z_2}, \frac{\partial y}{\partial z_3}$
- then $\frac{\partial y}{\partial z_4}, \frac{\partial y}{\partial z_5}$ from the previous $\frac{\partial y}{\partial z_i}$
- then $\frac{\partial y}{\partial z_6}, \frac{\partial y}{\partial z_7}$ from the previous $\frac{\partial y}{\partial z_i}$



...

- each edge $z_k \to z_i$: a partial derivative $\frac{\partial z_i}{\partial z_k}$ flowing backwards
- each path (e.g. $z_6 \to y$) is the product of its associated edge terms $\frac{\partial z_i}{\partial z_k}$. In particular:
- the term flowing back along $z_k \to z_i$ is the product of the last edge $\frac{\partial z_i}{\partial z_k}$ with the term $\frac{dy}{dz_i}$ stored in z_i
- the partial derivative in z_k is the sum of all terms flowing back into z_k from all $z_k \to z_i$

In practice walking down is done layer by layer. Define a layer index.

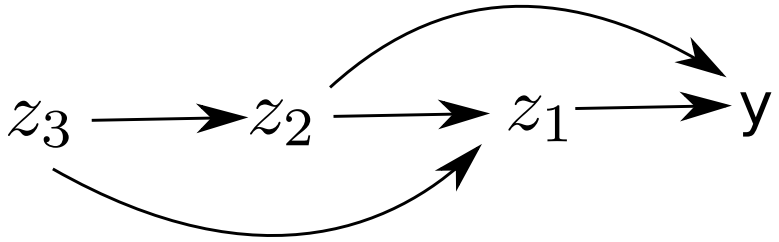
The precise definition is not important. We just need an ordering of neurons into layers.

Auxiliary: Layer index (backward def.)

Suppose the graph has inputs X, outputs Y. The outputs $y_i \in Y$ have no connections to each other.

- Then assign $y_i \in Y$ as the layer index $l(y_i)$ the longest path length from any $x_k \in x$ to y_i.
- for any other neuron n in the graph, assign as layer index $l(n)$:

$$l(n) = \min_{y_i \in Y} l(y_i) - \max_{\text{paths } p: n \to y_i} |p|$$



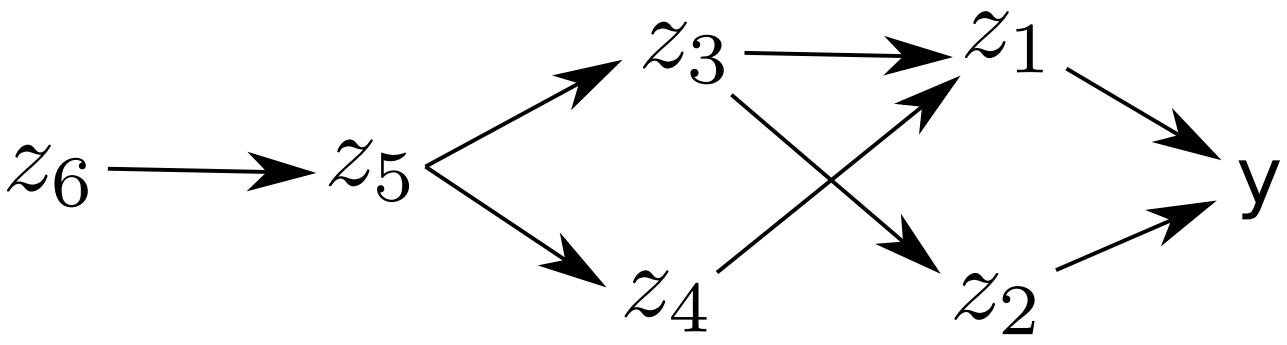
in practice walking down is done layer by layer

Backpropagation – not yet efficient

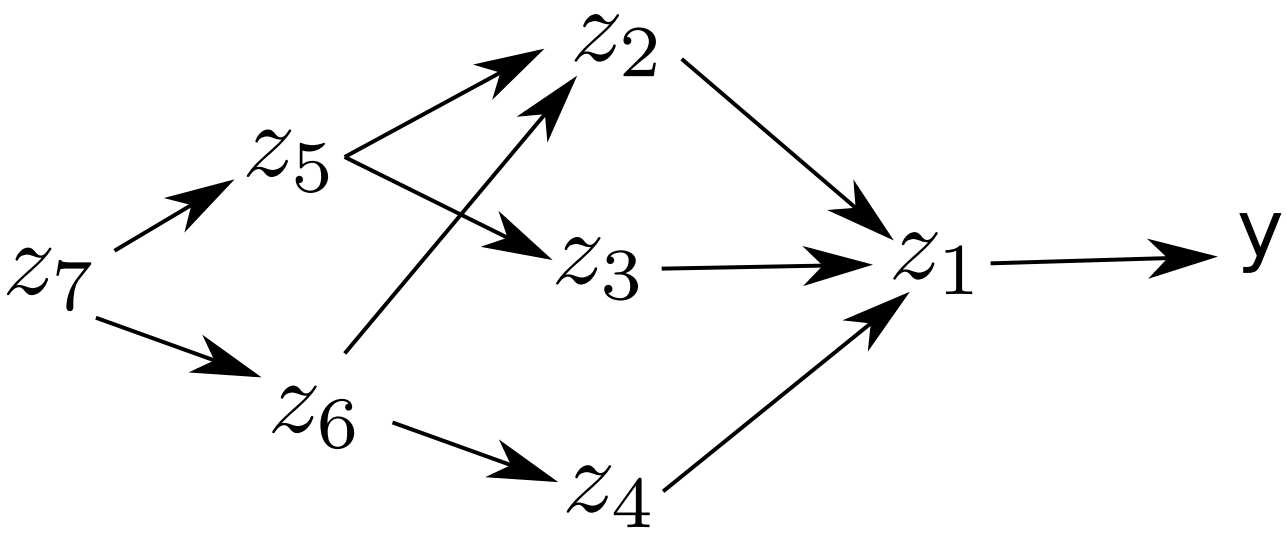
1. Start at the top by finding the set of neurons z_l such that the loss function directly depends on their inputs $L = L(z_l)$. Let the last layer index be M.
2. for $i \in range(M-1, -1, step = -1)$

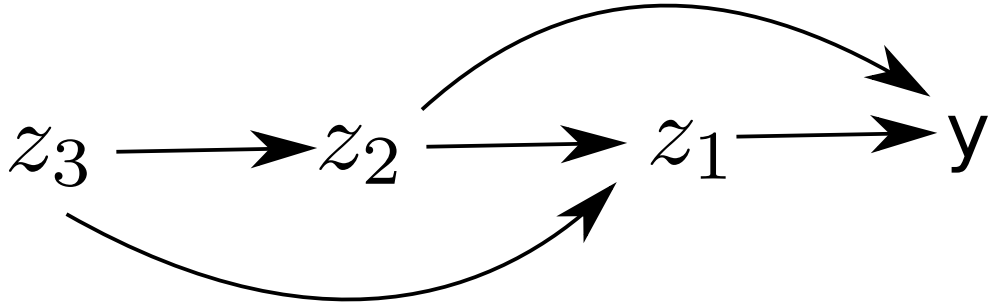
$$\forall z_k \in \text{Layer}_i : \frac{dL}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dL}{dz_l}\frac{\partial z_l}{\partial z_k} \tag{9}$$

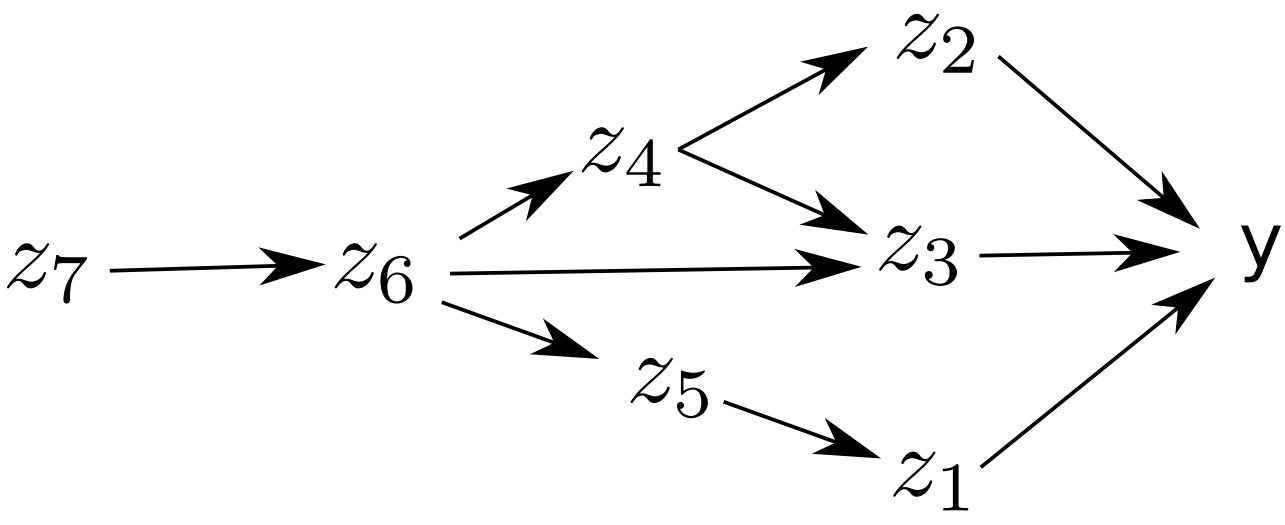
3. next: repeat step 2. for the z_k until you reach the bottom / or until you have covered all paths backwards from E to your weight of interest w_k
4. Finish at the bottom by $\frac{dL}{dw_k} = \frac{dL}{dz_k}\frac{\partial z_k}{\partial w_k}$


z_6
z_5
z_3
z_4
z_1
z_2
y

Exercise example for backprop




z_3
z_2
z_1
y


z_2
z_4
z_7
z_6
z_3
y
z_5
z_1

Above algorithm is not efficient yet!

Next steps:

- rewrite coordinate-wise chain rule as matrix multiplications
- convert

$$\frac{dL}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dL}{dz_l}\frac{\partial z_l}{\partial z_k}$$

  into layer-wise matrix multiplications

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \ldots, z_m) \in \mathbb{R} \tag{10}$$

$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \ldots, x_d) \in \mathbb{R}^m \tag{11}$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}|_{g(x)} \frac{\partial g_r}{\partial x_k}|_x \tag{12}$$

$$\nabla(f|_{g(x)})_r := \frac{\partial f}{\partial z_r}|_{g(x)} \tag{13}$$

$$J_{kr} := \frac{\partial g_r}{\partial x_k}|_x \tag{14}$$

$$\Rightarrow \frac{\partial f \circ g}{\partial x_k}|_x = \sum_{r=1}^{m} J_{kr} \nabla(f|_{g(x)})_r = (J \star \nabla f|_{g(x)})_k \tag{15}$$

Observation: the k-th partial derivative is the k-th component of matrix-vector product $J \star v$, $\star$ explicit for matrix multiplication

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \ldots, z_m) \in \mathbb{R} \tag{16}$$

$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \ldots, x_d) \in \mathbb{R}^m \tag{17}$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x) = (J \star \nabla f|_{g(x)})_k \tag{18}$$

$$\Rightarrow \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = J \star \nabla f|_{g(x)} \tag{19}$$

- by definition:

$$\begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}|_x \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}|_x \end{pmatrix} = \nabla (f \circ g)|_x$$

- what is J?

- what is J? $J_{kr} \overset{\text{by def}}{=} \frac{\partial g_r}{\partial x_k}\big|_x$.

$$\Rightarrow J = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}\big|_x, \ldots, \frac{\partial g_m}{\partial x_1}\big|_x \\ \frac{\partial g_1}{\partial x_2}\big|_x, \ldots, \frac{\partial g_m}{\partial x_2}\big|_x \\ \vdots \\ \frac{\partial g_1}{\partial x_d}\big|_x, \ldots, \frac{\partial g_m}{\partial x_d}\big|_x \end{pmatrix} = \underbrace{(\nabla g_1|_x, \ldots, \nabla g_m|_x)}_{\text{Jacobi-matrix (or its transpose)}} \tag{20}$$

- matrix of gradients $\nabla g_i|_x$ for each output component g_i

- function f maps onto real numbers ($f(x) \in \mathbb{R}$)
  $\tilde{\leftrightarrow}$ apply ∇, Derivative: gradient

$$f(x_1, \ldots, x_d) \in \mathbb{R} \Rightarrow \nabla f|_x = \begin{pmatrix} \frac{\partial f}{\partial x_1}|_x \\ \vdots \\ \frac{\partial f}{\partial x_d}|_x \end{pmatrix} \text{ is the gradient}$$

- function $g = (g_1, \ldots, g_m)$ maps onto vectors ($g(x) \in \mathbb{R}^m$)
  $\tilde{\leftrightarrow}$ apply ∇ to each component g_k, Derivative: Jacobi-matrix.

Jacobi-Matrix $J^{(g)}$

is a name for the matrix $(\nabla g_1|_x, \ldots, \nabla g_m|_x)$ of concatenated gradients for each output component g_i of $g = (g_1, \ldots, g_m)$ (applied ∇ to each output component)
The definition is transposed sometimes!

The definition is transposed sometimes!

$$J_{kr} \overset{\text{by def}}{=} \frac{\partial g_r}{\partial x_k}|_x$$

$$\text{or } J_{rk} \overset{\text{by def}}{=} \frac{\partial g_r}{\partial x_k}|_x$$

chain rule *n*-dim case as matrix multiplications

assume:

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R} \tag{21}$$

and if ∇f is a column vector and if g is a row vector (22)

$$g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) = (g_1, \ldots, g_m) \in \mathbb{R}^m \tag{23}$$

$$\text{then: } \nabla(f \circ g)|_x = (\nabla g_1|_x, \nabla g_2|_x, \ldots, \nabla g_m|_x) \star \nabla f|_{g(x)} \tag{24}$$

$$= (\text{inner function } g \text{ gradients}) \star (\text{outer function } f \text{ gradients}) \tag{25}$$

Mind here that the shapes must be correctly in this form: gradients are assumed to be column vectors, while $g.shape = (1, m)$.

If instead we would have gradients as row vectors $(\nabla f).shape = (1, m)$, $(\nabla g_i).shape = (1, d)$ and $g.shape = (m, 1)$: then J and ∇f are transposed to the above (!), then one has to use ... see on the next slides

if gradients are column vectors

$$\nabla(f \circ g)|_x = (\nabla g_1|_x, \nabla g_2|_x, \ldots, \nabla g_m|_x) \star \nabla f|_{g(x)} \tag{26}$$

if gradients are row vectors instead, then one has to use

$$\nabla(f \circ g)|_x = \nabla f|_{g(x)} \star \begin{pmatrix} \nabla g_1|_x \\ \nabla g_2|_x \\ \vdots \\ \nabla g_m|_x \end{pmatrix} = \nabla f|_{g(x)} \star J \tag{27}$$

Why? Equations are equiv under transposition

$$AB = CD^\top$$
$$\Leftrightarrow B^\top A^\top = DC^\top$$

How to remember that? consider the shapes:

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R}, \ \nabla f(x) \in \mathbb{R}^m \tag{28}$$

$$g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) \in \mathbb{R}^m, \ \nabla g(x) = J^{(g)} \in \mathbb{R}^{m*d} \tag{29}$$

if gradients are columns: $\nabla f \circ g \sim (d, 1)$, $\nabla f \sim (m, 1)$

$$(d, 1) \quad \tilde{=} (d, m) \star (m, 1) \text{ only one way for } J \tag{30}$$

$$\nabla(f \circ g)|_x = (\nabla g_1|_x, \nabla g_2|_x, \ldots, \nabla g_m|_x) \star \nabla f|_{g(x)} \tag{31}$$

if gradients are rows: $\nabla f \circ g \sim (1, d)$, $\nabla f \sim (1, m)$

$$(1, d) \quad \tilde{=} (1, m) \star (m, d) \text{ ...transpose of the above} \tag{32}$$

$$\nabla(f \circ g)|_x = \nabla f|_{g(x)} \star \begin{pmatrix} \nabla g_1|_x \\ \nabla g_2|_x \\ \vdots \\ \nabla g_m|_x \end{pmatrix} \tag{33}$$

the result (II): chain rule in matrix form (transposed version)

chain rule n-dim case as matrix multiplications (transposed version)

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R}, \; g : \mathbb{R}^d \mapsto \mathbb{R}^m, \tag{34}$$

$$\text{if } g(x) = \begin{pmatrix} g_1 \\ \vdots \\ g_m \end{pmatrix} \in \mathbb{R}^m \text{ is a column vector and if} \tag{35}$$

$$\nabla f = \text{ is a row vector } \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_m} \right) \tag{36}$$

$$\text{then: } \nabla (f \circ g)|_x = \nabla f|_{(g(x)} \star J = \nabla f|_{g(x)} \star \begin{pmatrix} \nabla g_1|_x \\ \nabla g_2|_x \\ \vdots \\ \nabla g_m|_x \end{pmatrix} \tag{37}$$

$$= (\text{outer function } f \text{ gradients}) \star (\text{inner function } g \text{ gradients}) \tag{38}$$

Next step:

Backprop as recursion, using matrix algebra expressions

$\nabla^{(z)} L$ denotes: compute Jacobi-Matrix of L with respect to variable z.

- Remember: $f^{(l)} = f^{(l)}(w^{(l)}, f^{(l-1)})$, $f^{(l-1)} = f^{(l-1)}(w^{(l-1)}, f^{(l-2)})$
- assume we have already computed the gradient with respect to inputs to layer l: $\nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)})$.
- want: $\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)})$

  By previous slide:

$$\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{f^{(l-2)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)}) \tag{39}$$

- compute $\nabla^{f^{(l-2)}} f^{(l-1)}$ analytically, multiply
- have now: $\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l-1)})$
- iterate further down ($+$ reusing)

$$\nabla^{f^{(l-3)}}(L \cdots \circ f^{(l-1)} \circ f^{(l-2)}) = \nabla^{f^{(l-3)}} f^{(l-2)} \star \nabla^{f^{(l-2)}}(L \cdots \circ f^{(l-1)}) \tag{40}$$

- One thing to finish: we need gradients with respect to the trainable parameters $\nabla^{w^{(l-1)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)})$

$$f^{(l-1)} = f^{(l-1)}(w^{(l-1)}, f^{(l-2)}) \tag{41}$$

$$\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{f^{(l-2)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)}) \tag{42}$$

$$\nabla^{w^{(l-1)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{w^{(l-1)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)}) \tag{43}$$

Backpropagation

The last two equations are the iterative version of backprop in matrix algebra. (!) mind the shapes (gradients are column or row vectors)

Why is this efficient ?

Exemplary three layer network:

$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

goal: to compute $\nabla^{w_d} y$ – using chain rule, $\nabla^{(z)} y$ denotes which vector z of variables the gradient of y is computed for.

$$\nabla^{w^{(3)}} y \rightarrow \text{ compute directly}$$

$$\begin{aligned}
\nabla^{w^{(2)}} y &= \nabla^{w^{(2)}}(f^{(3)} \circ f^{(2)}) = \nabla^{w^{(2)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)} \\
&= \nabla^{w^{(2)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)}(w^{(3)}, f^{(2)}(\ldots)) \\
\nabla^{w^{(1)}} y &= \nabla^{w^{(1)}}(f^{(3)} \circ f^{(2)} \circ f^{(1)}) = \nabla^{w^{(1)}} f^{(1)} \star \nabla^{f^{(1)}}(f^{(3)} \circ f^{(2)}) \\
&= \nabla^{w^{(1)}} f^{(1)} \star \nabla^{f^{(1)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)}(w^{(3)}, f^{(2)}(\ldots))
\end{aligned}$$

Observation

We need to have the Jacobi-matrix $\nabla^{f^{(k-1)}} f^{(k)}(w^{(k)}, f^{(k-1)}(\ldots))$ for computing the gradients for all parameters $w^{(k-1)}, w^{(k-2)}, \ldots, w^{(1)}$ in layers closer to the input. Backprop: compute it once, reuse it implicitly by recursion.

Single layer gradients $\nabla^{f^{(l-2)}} f^{(l-1)}$

Piece of cake

$$f^{(l-1)} = g(f^{(l-2)} \star w^{(l-1)} + b^{(l-1)}),\ g(\cdot) \text{ activation} \tag{44}$$

Note $f^{(l-1)}$ is a $(1, m)$-shaped vector, and $w^{(l-1)}$ is a matrix (n, m) , and $f^{(l-2)}$ is a $(1, n)$-shaped vector.

Consider a single output neuron $f^{(l-1)}[0, k]$ of the layer $f^{(l-1)}$:

$$\text{then: } f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{45}$$

$$\nabla^{f^{(l-2)}} f^{(l-1)}[0, k] = ? \tag{46}$$

$$\nabla^{w^{(l-1)}[:,k]} f^{(l-1)}[0, k] = ? \text{ the two red must match} \tag{47}$$

This is because slice $w^{(l-1)}[:, k]$ is used only when computing $f^{(l-1)}[0, k]$ and not for other indices $i : i \neq k$ of $f^{(l-1)}[0, i]$

Single layer gradients $\nabla^{f^{(l-2)}} f^{(l-1)}$

Piece of cake

$$f^{(l-1)} = g(f^{(l-2)} \star w^{(l-1)} + b^{(l-1)}),\ g(\cdot) \text{ activation} \tag{48}$$

Note $f^{(l-1)}$ is a $(1, m)$-shaped vector, and $w^{(l-1)}$ is a matrix (n, m) , and $f^{(l-2)}$ is a $(1, n)$-shaped vector.

Consider a single output neuron $f^{(l-1)}[0, k]$ of the layer $f^{(l-1)}$:

$$\text{then: } f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{49}$$

$$\nabla^{f^{(l-2)}} f^{(l-1)}[0, k] = \left(\frac{\partial f^{(l-1)}[0, k]}{\partial f^{(l-2)}[0, d]}, d = 1, \dots \right) \tag{50}$$

$$f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{51}$$

$$\frac{\partial f^{(l-1)}[0, k]}{\partial f^{(l-2)}[0, d]} = g'(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) w^{(l-1)}[d, k] \tag{52}$$

Piece of cake

$$f^{(l-1)} = g(f^{(l-2)} \star w^{(l-1)} + b^{(l-1)}),\ g(\cdot) \text{ activation} \tag{53}$$

Note $f^{(l-1)}$ is a $(1, m)$-shaped vector, and $w^{(l-1)}$ is a matrix (n, m) , and $f^{(l-2)}$ is a $(1, n)$-shaped vector.

Consider a single output neuron $f^{(l-1)}[0, k]$ of the layer $f^{(l-1)}$:

$$\text{then: } f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{54}$$

$$\nabla^{w^{(l-1)}[:,k]} f^{(l-1)}[0, k] = ? \text{ the two red must match} \tag{55}$$

$$f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{56}$$

$$\Rightarrow \frac{\partial f^{(l-1)}[0, k]}{\partial w^{(l-1)}[d, k]} = g'(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) f^{(l-2)}[0, d] \tag{57}$$

Piece of cake

$$f^{(l-1)} = g(f^{(l-2)} \star w^{(l-1)} + b^{(l-1)}),\ g(\cdot) \text{ activation} \tag{58}$$

Note $f^{(l-1)}$ is a $(1, m)$-shaped vector, and $w^{(l-1)}$ is a matrix (n, m) , and $f^{(l-2)}$ is a $(1, n)$-shaped vector.

Consider a single output neuron $f^{(l-1)}[0, k]$ of the layer $f^{(l-1)}$:

$$\text{then: } f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) \tag{59}$$

$$\frac{\partial f^{(l-1)}[0, k]}{\partial b^{(l-1)}[0, k]} = g'(f^{(l-2)} \star w^{(l-1),[:,k]} + b^{(l-1)}[0, k]) * 1 \tag{60}$$

Now: Chain rule for more than two functions using linear algebra ?

assume: gradients are column vectors here. Then:

$$\nabla(f \circ g)|_x = (\nabla g_1|_x, \nabla g_2|_x, \ldots, \nabla g_m|_x) \star \nabla f|_{g(x)} \tag{61}$$

$$= (\text{inner function } g \text{ gradients}) \star (\text{outer function } f \text{ gradients}) \tag{62}$$

This chains to more than two functions:

$$\nabla(f \circ g \circ t)|_x = \nabla t|_x \star \nabla(f \circ g)|_{t(x)} = \tag{63}$$

$$= (\nabla t_1|_x, , \ldots, \nabla t_n|_x) \star (\nabla g_1|_{t(x)}, \ldots, \nabla g_m|_{t(x)}) \star \nabla f|_{g(t(x))} \tag{64}$$

$$= (\text{inner gradients}) \star (\text{mid gradients}) \star (\text{outer gradients}) \tag{65}$$

$$= (\text{inner gradients})_{|x} \star (\text{mid gradients})_{|t(x)} \star (\text{outer gradients})_{|g(t(x))} \tag{66}$$

1. A recap on gradients, part I
2. Gradients for multilinear Algebra
3. Gradient Descent and its properties
4. Stochastic gradient descent
5. Backpropagation
6. Preview on Training and Initialization

The most important lessons

(1) Do not train from scratch. Initialize your model with weights pretrained on a large dataset (Usually from ImageNet-1K)

(2) If you would train from scratch, success or failure of training is very sensitive on how you initialize the neural network

- many tutorials in the internet on simple datasets (Fashion-MNIST, CIFAR-10) which train from scratch with sloppy initializations are totally misleading for more realistic and harder problems (!!)

- more details in the next lectures
- alternatives are available such as self-supervised pretraining, but this also needs sufficiently large datasets

This is for the case if you would train from scratch. No need to memorize the exact constant in the variance term.

Initialization for ReLU networks

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
- conv layers: draw weights from a normal with standard deviation equal to $\sigma = \sqrt{\frac{2}{n}}$
  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init
- older Xavier-Glorot style initialization is considered obsolete for deep networks while it still works for shallow networks

This is for the case if you would train from scratch. No need to memorize the exact constant in the variance term.

Initialization for pReLU networks

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init

Questions?!