L6 – Gradients and Gradient Descent

Alexander Binder

October 29, 2025

links

know where to look for

- d2l.ai Appendix A.3 and A.4
- https://pytorch.org/

Takeaway points

Takeaway points

at the end of this lecture you should be able to:

- directional derivatives and gradient
- directional derivatives $\leftrightarrow$ partial derivatives
- further topics in the deep learning lecture:
  - vector-valued function $\rightarrow$ Jacobi-matrix
  - chain rule with directional argument written as matrix multiplications
  - derivative of a linear function and matrix multiplications

Linear and Bilinear mappings are the most common in deep learning. Combinations of these together with activation functions can be dealt with using the chain rule.

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

How to find the trainable parameters w 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 $w^* = \mathrm{argmin}_w g(w)$
- assumption: can compute $\nabla g|_w$ - the gradient in point w.

Remember Logistic regression

Example for such a $g(w)$ (assume binary classification with zero-one-labels $y_i \in \{0, 1\}$)

$$g(w, b) = \sum_{i=1}^{n} L(s(x_i), y_i) = \sum_{i=1}^{n} -y_i \ln s_{(w,b)}(x_i)$$
$$s_{(w,b)}(x) = \frac{1}{1 + e^{-w \cdot x - b}}$$

for a given dataset $D_n = \{(x_i, y_i)\}$. Once (w, b) is found, we use $s_{(w,b)}(x)$ as prediction mapping.

How to apply this:

- consider the loss g as function of w
- compute $\nabla g|_w = \nabla^{(w)} \sum_{i=1}^{n} L(s(x_i), y_i)$ - the gradient of the loss with respect to w

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Limits

- define sequence convergence
- define limit when input is moving to some value or to infinity

Limits

- a limit $\lim_{\epsilon \to a} g(\epsilon)$ exists if
  for each sequence of numbers $(s_n)_{n=1}^{\infty}$ which converges to a (.i.e. $\lim_{n\to\infty} s_n = a$)
  - the sequence $g(s_n)$ converges to a value z
  - and it is the same value z for all such sequences $(g(s_n))_{n=1}^{\infty}$

out of exams: the definition of convergence of a sequence

- $x : \ |x - e| < \delta$ are those points x which are at most δ far away from e
- a sequence $(s_n)_{n=1}^{\infty}$ converges to a value s if:

  for each value of $\delta > 0$ there exists an index K such that for all indices $n \geq K$:

  $$|s_n - e| < \delta \tag{1}$$

  ...

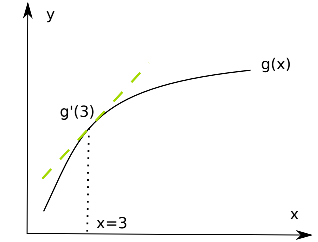
  - one writes this as: $\lim_{n\to\infty} s_n = e$
- intuition: no matter how small the distance δ we require, from some K on all points s_n will be closer to e than this δ

  Example: $s_n = 3 + (-1)^n \frac{1}{n}, \ s_n \xrightarrow{n\to\infty} 3$

Gradient in 1 dimension

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

$$\lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad (=: f'(x)) \tag{2}$$



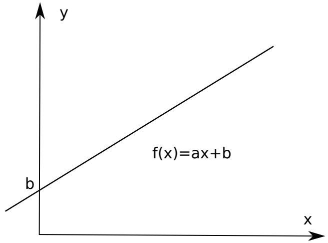
- intuition: slope of the function f at point x
- example: $f(x) = ax + b$ (affine with slope a),then

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = \frac{a(x+\epsilon) + b - (ax+b)}{\epsilon} \tag{3}$$

$$= \frac{ax + a\epsilon + b - ax - b}{\epsilon} \tag{4}$$

$$= \frac{a\epsilon}{\epsilon} = a \tag{5}$$

$$\Rightarrow \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} = a \tag{6}$$



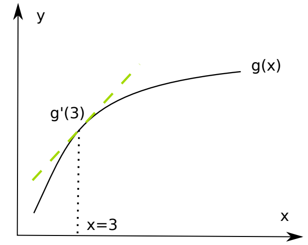
Gradient in 1 dimension

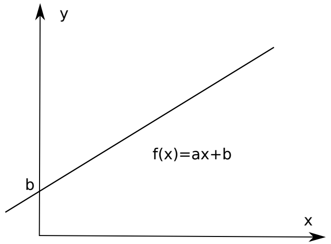
- $f : \mathbb{R}^1 \to \mathbb{R}^1$ is differentiable in input x if the limit exists:

$$\lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad (=: f'(x)) \tag{2}$$

- intuition: slope of the function f at point x
- example: $f(x) = ax + b$ (affine with slope a),then

$$\begin{aligned}
\frac{f(x+\epsilon) - f(x)}{\epsilon} &= \frac{a(x+\epsilon) + b - (ax+b)}{\epsilon} && (3)\\
&= \frac{ax + a\epsilon + b - ax - b}{\epsilon} && (4)\\
&= \frac{a\epsilon}{\epsilon} = a && (5)\\
\Rightarrow \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} &= a && (6)
\end{aligned}$$





Gradient in 1 dimension

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

$$\lim_{\epsilon \rightarrow 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad (=: f'(x)) \tag{2}$$

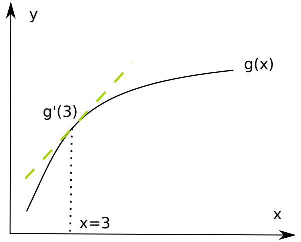
- intuition: slope of the function f at point x
- example: $f(x) = ax + b$ (affine with slope a),then

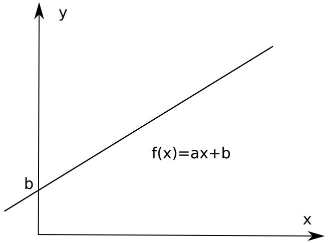
$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = \frac{a(x+\epsilon) + b - (ax+b)}{\epsilon} \tag{3}$$

$$= \frac{ax + a\epsilon + b - ax - b}{\epsilon} \tag{4}$$

$$= \frac{a\epsilon}{\epsilon} = a \tag{5}$$

$$\Rightarrow \lim_{\epsilon \rightarrow 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} = a \tag{6}$$

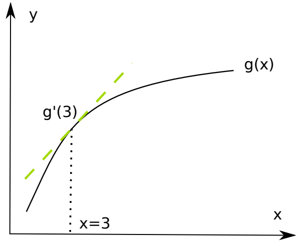




Gradient in 1 dimension

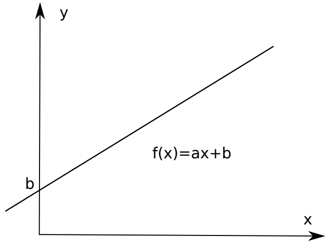
- $f : \mathbb{R}^1 \to \mathbb{R}^1$ is differentiable in input x if the limit exists:

$$\lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad (=: f'(x)) \tag{2}$$



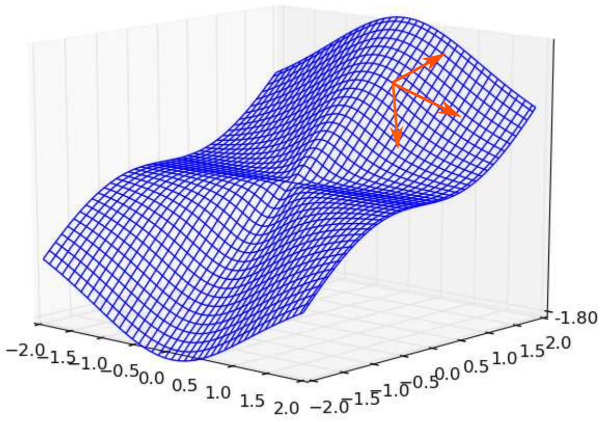
- intuition: slope of the function f at point x
- example: $f(x) = ax + b$ (affine with slope a),then

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = \frac{a(x+\epsilon) + b - (ax+b)}{\epsilon} \tag{3}$$
$$= \frac{ax + a\epsilon + b - ax - b}{\epsilon} \tag{4}$$
$$= \frac{a\epsilon}{\epsilon} = a \tag{5}$$
$$\Rightarrow \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} = a \tag{6}$$



2-dimensions: directional derivatives

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

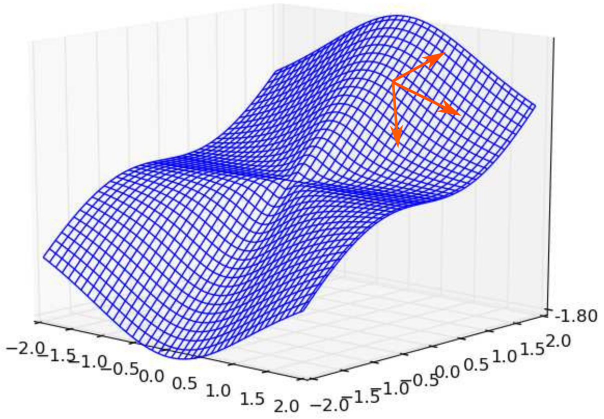


- in every point (x_1, x_2): a two dimensional vector space of directions to move away from (x_1, x_2)
- in every direction there is a slope (red arrows) – the directional derivative

n-dimensions: directional derivatives

Function of *n* input variables:

$$f(x_1, x_2, \ldots, x_n) \in \mathbb{R}^1$$

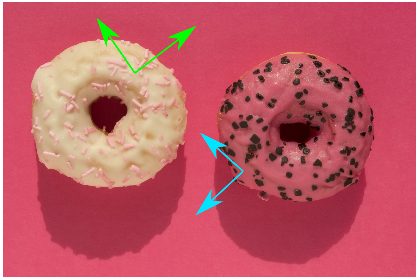


- in every point $(x_1, x_2, \ldots, x_n)$: a n-dimensional vector space of directions to move away from $(x_1, x_2, \ldots, x_n)$
- in every direction there is a slope (red arrows) – the directional derivative – provides information about function value change in this direction

Gradients

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 to move away from (p_1, p_2) is $d = 4$-dimensional!
- in every point $(x_1, x_2, \ldots, x_n)$: a n-dimensional vector space of directions to move away from $(x_1, x_2, \ldots, x_n)$

n-dimensions: directional derivatives and gradient

The directional derivative

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

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

next step:

- define partial derivatives
- define the gradient $\nabla f|_{\mathbf{x}}$ via partial derivatives
- establish relationship: directional derivatives $\delta_{\mathbf{v}} f|_{\mathbf{x}}$ versus the gradient $\nabla f|_{\mathbf{x}}$

n-dimensions: the partial derivative

- consider $f : \mathbb{R}^n \to \mathbb{R}^1$, $f(\mathbf{x}) = f(x_1, \dots, x_n) \in \mathbb{R}^1$
  The i-th one hot vector $\mathbf{e}_i$ is defined as:

$$\mathbf{e}_i = (0, \dots, \quad 0, \quad \underbrace{1}_{i} \quad, \quad 0 \quad, \dots, 0)^\top$$

- we can define a partial derivative for variable input x_i:

The partial derivative

$$\frac{\partial f}{\partial x_i}\Big|_{\mathbf{x}} = \delta_{\mathbf{e}_i} f\Big|_{\mathbf{x}} = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x})}{\epsilon}$$

note: $\mathbf{x} + \epsilon \mathbf{e}_i = (x_1, \dots, x_{i-1}, \underbrace{x_i + \epsilon}_{\text{i-th dim}}, x_{i+1}, \dots, x_n)^\top$

n-dimensions: the partial derivative

The partial derivative

$$\frac{\partial f}{\partial x_i}\Big|_{\mathbf{x}} = \delta_{\mathbf{e}_i} f\big|_{\mathbf{x}} = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x})}{\epsilon}$$

$$\text{note: } \mathbf{x} + \epsilon \mathbf{e}_i = (x_1, \dots, x_{i-1}, \underbrace{x_i + \epsilon}_{\text{i-th dim}}, x_{i+1}, \dots, x_n)^\top$$

- note: the partial derivative $\frac{\partial f}{\partial x_i}|_{\mathbf{x}}$ is the derivative of a function $g_{(x\setminus\{x_i\})}(t)$ in one dimension – for which all dimensions of vector $\mathbf{x} = (x_0, \dots, x_i, \dots, x_n)$ are fixed except the i-th dimension:

$$\begin{aligned} g_{(x\setminus\{x_i\})}(t) &= f(x_0, \dots, t, \dots, x_n) \\ \Rightarrow g'(x_i) &= \lim_{\epsilon \to 0} \frac{g(x_i + \epsilon) - g(x_i)}{\epsilon} = \frac{\partial f}{\partial x_i}\Big|_{\mathbf{x}} \end{aligned}$$

the gradient

- define the gradient $\nabla f|_{\mathbf{x}}$ of function f in $\mathbf{x}$ as the vector of all partial derivatives in input point $\mathbf{x}$:

$$\nabla f|_{\mathbf{x}} = \begin{pmatrix} \frac{\partial f}{\partial x_1}|_{\mathbf{x}} \\ \frac{\partial f}{\partial x_2}|_{\mathbf{x}} \\ \vdots \\ \frac{\partial f}{\partial x_n}|_{\mathbf{x}} \end{pmatrix} \tag{7}$$

- the gradient stores information about all slopes of a function at $\mathbf{x}$ for every direction $\mathbf{v}$ from $\mathbf{x}$:

directional derivatives and gradient

Fact: If the function is differentiable in x, then the directional derivative $\delta_{\mathbf{v}} f(|_{\mathbf{x}}$ is equal to the inner product of the gradient $\nabla f|_{\mathbf{x}}$ of f in $\mathbf{x}$ with $\mathbf{v}$

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

$$\nabla f|_{\mathbf{x}} \cdot \mathbf{e}^{(k)} = \frac{\partial f}{\partial x_k}|_{\mathbf{x}}$$

n-dimensions: directional derivatives and gradient

Take-away I

defined:

- directional derivative
- partial derivative
- gradient

Take-away II

- directional derivatives tell you how the function grows from $\mathbf{x}$ in direction $\mathbf{v}$ when you take an infinitely small step
- the gradient contains information about all directional derivatives, if f is differentiable in $\mathbf{x}$, via inner products of the gradient in $\mathbf{x}$ with directions $\mathbf{v}$

What does it mean to be differentiable ?

definition of differentiability

$f : \mathbb{R}^n \to \mathbb{R}^1$, $f(\mathbf{x}) = f(x_1, \ldots, x_n) \in \mathbb{R}^1$
is differentiable in input $\mathbf{x}$ if

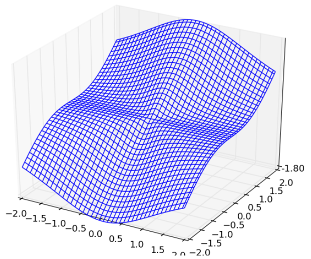
- all directional derivatives (for all vectors $\mathbf{v}$) exist
- the directional derivatives satisfy a linear relationship (with real numbers a_1, a_2)

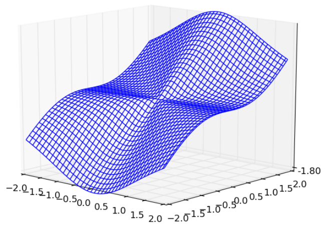
Linear relationship means that the following holds for all $a_1, a_2 \in \mathbb{R}$ and vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^n$:

$$\partial_{a_1\mathbf{v}_1 + a_2\mathbf{v}_2} f(\mathbf{x}) = a_1 \partial_{\mathbf{v}_1} f(\mathbf{x}) + a_2 \partial_{\mathbf{v}_2} f(\mathbf{x}) \tag{8}$$

n-dimensions: the partial derivative

- why we did not simply say f is differentiable if all its partial derivatives exist?
- $f((x,y)) = \begin{cases} \frac{y^3}{x^2+y^2} & \text{if } (x,y) \neq (0,0) \\ 0 & \text{if } (x,y) = (0,0) \end{cases}$
- both partial derivatives exist, but function has kinks in its *directional* derivatives
- linearity of directional derivatives (on the previous slide) not satisfied





Gradients

next: show why gradients can help to find direction where a function is increasing or decreasing

Gradients

Which direction $\mathbf{v}$ maximizes $\delta_{\mathbf{v}} f|_{\mathbf{x}} = \nabla f|_{\mathbf{x}} \cdot \mathbf{v}$?

$$\delta_{\mathbf{v}} f|_{\mathbf{x}} = \nabla f|_{\mathbf{x}} \cdot \mathbf{v}$$

Fact: the optimization problem

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

is solved by $\mathbf{v} := \frac{\mathbf{w}}{\|\mathbf{w}\|}$. This has an important consequence!: $\frac{\nabla f|_{\mathbf{x}}}{\|\nabla f|_{\mathbf{x}}\|}$ is the direction with the highest slope of f in $\mathbf{x}$.

Gradient and local function change

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**.

Gradients

Fact: the optimization problem

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

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

$$\mathbf{w}\cdot\mathbf{v} = \|\mathbf{w}\|\|\mathbf{v}\|\cos(\angle(\mathbf{w},\mathbf{v})) = \|\mathbf{w}\|1\cos(\angle(\mathbf{w},\mathbf{v}))$$

- $\mathbf{w}$ is not optimized over / it is a constant .
- The $\cos(\cdot)$ is maximized for an angle $= 0$.
- therefore $\mathbf{v} = c\mathbf{w},\ c > 0$. Now use the constraint: $\|\mathbf{v}\| = 1$. Implies $\mathbf{v} := \frac{\mathbf{w}}{\|\mathbf{w}\|}$
- analogously we obtain that $-1 * \frac{\mathbf{w}}{\|\mathbf{w}\|}$ is the direction of $\operatorname{argmin}_{\mathbf{v}:\|\mathbf{v}\|=1} \mathbf{w}\cdot\mathbf{v}$

Gradients

Gradient and local function change

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**.

Preview: This will be the key idea to use gradients to find points which minimize a loss:

- we will start at some point, then always take a small step in the direction of the negative gradient

n-dimensions: vector-valued functions

$$\begin{aligned} \mathbf{f}(\mathbf{x}) &= (f_1(x_1, \dots, x_n), f_2(x_1, \dots, x_n), \dots, f_s(x_1, \dots, x_n)) \\ \mathbf{f} &: \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{f}(\mathbf{x}) \in \mathbb{R}^s \end{aligned}$$

How to define a gradient for a vector-valued function $\mathbf{f}|_{\mathbf{x}}$?

n-dimensions: vector-valued functions

$$\begin{aligned}\mathbf{f}(\mathbf{x}) =& (f_1(x_1, \ldots, x_n), f_2(x_1, \ldots, x_n), \ldots, f_s(x_1, \ldots, x_n)) \\ \mathbf{f} :& \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{f}(\mathbf{x}) \in \mathbb{R}^s\end{aligned}$$

- Apply ∇ to every component f_i:

$$\begin{aligned} J(\mathbf{f})|_{\mathbf{x}} &= \left(\nabla f_1|_{\mathbf{x}}, \nabla f_2|_{\mathbf{x}}, \ldots, \nabla f_s|_{\mathbf{x}}\right) \\ &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_1}|_{\mathbf{x}}, \ldots, \frac{\partial f_s}{\partial x_1}|_{\mathbf{x}} \\ \frac{\partial f_1}{\partial x_2}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_2}|_{\mathbf{x}}, \ldots, \frac{\partial f_s}{\partial x_2}|_{\mathbf{x}} \\ \ldots \\ \frac{\partial f_1}{\partial x_n}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_n}|_{\mathbf{x}}, \ldots, \frac{\partial f_s}{\partial x_n}|_{\mathbf{x}} \end{pmatrix} \end{aligned}$$

We treat every component f_i separately

n-dimensions: vector-valued functions

Jacobi-Matrix

Let $\mathbf{f}$ be a vector-valued function, then

$$J(\mathbf{f})|_{\mathbf{x}} = \left(\nabla f_1|_{\mathbf{x}}, \nabla f_2|_{\mathbf{x}}, \dots, \nabla f_s|_{\mathbf{x}}\right)$$
$$= \begin{pmatrix} \frac{\partial f_1}{\partial x_1}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_1}|_{\mathbf{x}}, \dots, \frac{\partial f_s}{\partial x_1}|_{\mathbf{x}} \\ \frac{\partial f_1}{\partial x_2}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_2}|_{\mathbf{x}}, \dots, \frac{\partial f_s}{\partial x_2}|_{\mathbf{x}} \\ \dots \\ \frac{\partial f_1}{\partial x_n}|_{\mathbf{x}}, \frac{\partial f_2}{\partial x_n}|_{\mathbf{x}}, \dots, \frac{\partial f_s}{\partial x_n}|_{\mathbf{x}} \end{pmatrix}$$

is called the Jacobi-Matrix of $\mathbf{f}$ in $\mathbf{x}$

One can also write it by abuse of notation as $\nabla\mathbf{f}|_{\mathbf{x}}$

Its piece of cake!

Outline

1. Motivation
2. The derivative of a 1dim function and the Gradient
3. The directional derivative and Derivatives of Linear mappings
4. Chain Rule
5. Gradients in PyTorch and Autograd
6. Gradient Descent
7. Properties of Gradient Descent
8. Stochastic gradient descent

n-dimensions: derivatives as linear mappings into the space of all directional derivatives

- if f is differentiable in $\mathbf{x}$, then the directional derivatives satisfy a linearity condition. Therefore we can define a linear mapping $Df|_{\mathbf{x}}[\mathbf{v}]$ in point $\mathbf{x}$ using the directional derivatives:

$$Df|_{\mathbf{x}}[\mathbf{v}] := \nabla f|_{\mathbf{x}} \cdot \mathbf{v} \tag{9}$$

- $Df|_{\mathbf{x}}[\mathbf{v}]$ has two arguments
  - the point $\mathbf{x}$ in which the derivative is computed
  - the vector $\mathbf{v}$ for the direction, in which one wants to know the slope
- the mapping $Df|_{\mathbf{x}}[\mathbf{v}]$ is linear only in its second argument $\mathbf{v}$ (the direction)

n-dimensions: derivatives as linear mappings into the space of all directional derivatives

- $Df|_{\mathbf{x}}[\mathbf{v}]$ is linear in $\mathbf{v}$ means:

$$Df|_{\mathbf{x}}[c\mathbf{v}] = cDf|_{\mathbf{x}}[\mathbf{v}] \text{ for } c \in \mathbb{R} \tag{10}$$

$$Df|_{\mathbf{x}}[\mathbf{v}_1 + \mathbf{v}_2] = Df|_{\mathbf{x}}[\mathbf{v}_1] + Df|_{\mathbf{x}}[\mathbf{v}_2] \tag{11}$$

$$Df|_{\mathbf{x}}[\mathbf{0}] = 0 \tag{12}$$

n-dimensions: derivatives as linear mappings into the space of all directional derivatives

What is the definition

$$Df|_{\mathbf{x}}[\mathbf{v}] := \nabla f|_{\mathbf{x}} \cdot \mathbf{v} \tag{13}$$

good for ?

- computing derivatives sometimes clearer when starting from the directional derivatices (example $\mathbf{F}(\mathbf{X}) = \mathbf{AXC}$ below) :
- can help getting the gradient written using linear algebra, without many sum terms (example of the gradient of a quadratic function $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{Ax}$ below):

the derivative of a linear function

- let f be a linear mapping in $\mathbf{x}$. Then we can write it as:

$$f(\mathbf{x}) = \mathbf{u} \cdot \mathbf{x} = \sum_i u_i x_i \tag{14}$$

- then $\frac{\partial f}{\partial x_i}|_{\mathbf{x}} = u_i$ and therefore:

$$Df|_{\mathbf{x}}[\mathbf{v}] = \nabla f|_{\mathbf{x}} \cdot \mathbf{v} = \mathbf{u} \cdot \mathbf{v} = f(\mathbf{v}) \tag{15}$$

- the derivative of a linear function is a linear function
- practical for matrix algebra!

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{z}^\top \mathbf{x} \\ \Rightarrow Df|_{\mathbf{x}}[\mathbf{v}] &=? \\ \nabla f(\mathbf{x}) &=? \\ \mathbf{f}|_{\mathbf{x}} &= \mathbf{A}^\top \mathbf{x} \\ \Rightarrow Df|_{\mathbf{x}}[\mathbf{v}] &=? \\ \nabla f_k|_{\mathbf{x}} &=? \end{aligned}$$

can get all directional derivatives as matrix-multiplications (GPU-implementations!)

example $\mathbf{A} \in \mathbb{R}^{k\times m}, \mathbf{X} \in \mathbb{R}^{m\times l}, \mathbf{C} \in \mathbb{R}^{l\times r}$

$$\mathbf{F}(\mathbf{X}) = \mathbf{AXC}$$

$$\Rightarrow D\mathbf{F}|_{\mathbf{X}}[\mathbf{V}] =$$

example $\mathbf{A} \in \mathbb{R}^{k \times m}, \mathbf{X} \in \mathbb{R}^{m \times l}, \mathbf{C} \in \mathbb{R}^{l \times r}$

$$\mathbf{F}(\mathbf{X}) = \mathbf{AXC}$$

linear in $\mathbf{X}$!!!

$$\Rightarrow D\mathbf{F}|_{\mathbf{X}}[\mathbf{V}] =$$

example $\mathbf{A} \in \mathbb{R}^{k\times m}, \mathbf{X} \in \mathbb{R}^{m\times l}, \mathbf{C} \in \mathbb{R}^{l\times r}$

$$\begin{aligned} \mathbf{F}(\mathbf{X}) &= \mathbf{AXC} \\ &\quad \text{linear in } \mathbf{X} \text{ !!!} \\ \Rightarrow D\mathbf{F}|_{\mathbf{X}}[\mathbf{V}] &= \mathbf{AVC} \end{aligned}$$

sooo easy!!!

From here getting partial derivatives is easy:

$$\mathbf{F}(\mathbf{X}) = \mathbf{AXC}$$
$$\frac{\partial \mathbf{F}}{\partial X_{kl}}|_{\mathbf{X}} = D\mathbf{F}|_{\mathbf{X}}[\mathbf{1}^{(kl)}] = \mathbf{A}\mathbf{1}^{(kl)}\mathbf{C}$$
$$\text{where } \mathbf{1}_{ij}^{(kl)} = 1 \text{ if } i = k \text{ and } l = j, \text{ and 0 else}$$

Outline

1. Motivation
2. The derivative of a 1dim function and the Gradient
3. The directional derivative and Derivatives of Linear mappings
4. Chain Rule
5. Gradients in PyTorch and Autograd
6. Gradient Descent
7. Properties of Gradient Descent
8. Stochastic gradient descent

Why chainrule?

Why ?

- Use case 1: being able to compute directional derivatives and gradients for a single neuron $r(\mathbf{x})$ with respect to inputs $\mathbf{x}$ and trainable weights $\mathbf{w}$

$$g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + \mathbf{b}$$
$$r(\mathbf{x}) = f(\mathbf{w} \cdot \mathbf{x} + \mathbf{b}) = f(g(\mathbf{x}))$$

- Use case 2: being able to compute directional derivatives and gradients for a generalized quadratic function

$$r(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} = Matmul(\mathbf{x}^\top, \mathbf{A}\mathbf{x}) = f(g(\mathbf{x}), h(\mathbf{x}))$$
$$\text{with } g(\mathbf{x}) = \mathbf{x}^\top, \; h(\mathbf{x}) = \mathbf{A}\mathbf{x}$$

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

- 1-dim case:

$$r(x) = f(g(x)) \tag{16}$$
$$r'(x) = f'(g(x))g'(x) \tag{17}$$

- n-dim case:

$$h(\mathbf{x}) = (f \circ g)(\mathbf{x}) = f(g(\mathbf{x})) \tag{18}$$
$$Dh|_{\mathbf{x}}[\mathbf{v}] = ? \tag{19}$$

- Good news: you can remember it by drawing a graph, and assigning partial derivatives to its edges

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

- n-dim case:

$$\begin{aligned} h(\mathbf{x}) &= (f \circ g)(\mathbf{x}) = f(g(\mathbf{x})) \\ Dh|_{\mathbf{x}}[\mathbf{v}] &= ? \end{aligned}$$

- high level view: $f(x) \longrightarrow \nabla f_{\mathbf{x}} \longrightarrow Df|_{\mathbf{x}}[v] = \nabla f_{\mathbf{x}} \cdot v$
  - $Df|_{\mathbf{x}}[v] = \nabla f|_{\mathbf{x}} \cdot v$ computes a linear approximation to f at x by encoding all possible slopes in all possible directions
- if we compute the slopes for the concatenation $h(x) = f(g(x))$,

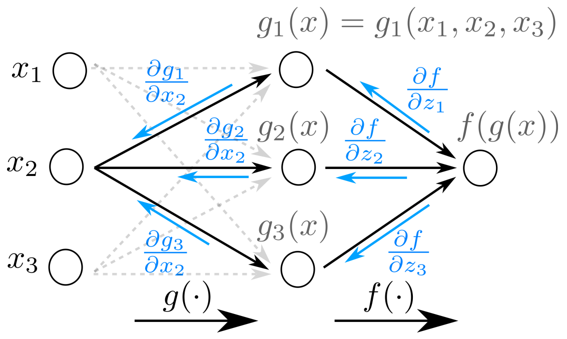
  ... expect to use the concatenation of linear approximations $Df|_{g(x)}[\cdot], Dg|_x[\cdot]$ for the respective component functions f and g

  informally:

$$D(f \circ g)|_{\mathbf{x}}[\mathbf{v}] = Df|_{g(\mathbf{x})}[Dg|_{\mathbf{x}}[\mathbf{v}]] = [\nabla f|_{g(\mathbf{x})}]^\top [Jg|_{\mathbf{x}}]^\top \mathbf{v}$$

- Good news: you can remember it by drawing a graph, and assigning partial derivatives to its edges

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



$g_1(x) = g_1(x_1, x_2, x_3)$
x_1
x_2
x_3
$\frac{\partial g_1}{\partial x_2}$
$\frac{\partial g_2}{\partial x_2}$
$\frac{\partial g_3}{\partial x_2}$
$g_2(x)$
$g_3(x)$
$\frac{\partial f}{\partial z_1}$
$\frac{\partial f}{\partial z_2}$
$\frac{\partial f}{\partial z_3}$
$f(g(x))$
$g(\cdot)$
$f(\cdot)$

$$\frac{\partial f \circ \mathbf{g}}{\partial x_2}(\mathbf{x}) = \frac{\partial f}{\partial z_1}\bigg|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_1}{\partial x_2} + \frac{\partial f}{\partial z_2}\bigg|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_2}{\partial x_2} + \frac{\partial f}{\partial z_3}\bigg|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_3}{\partial x_2}$$

Three steps:

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

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

- next step: Write the result as a linear operation between two directional derivative terms

$$\frac{\partial f \circ \mathbf{g}}{\partial x_2}|_{\mathbf{x}} = \frac{\partial f}{\partial z_1}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_1}{\partial x_2}|_{\mathbf{x}} + \frac{\partial f}{\partial z_2}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_2}{\partial x_2}|_{\mathbf{x}} + \frac{\partial f}{\partial z_3}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_3}{\partial x_2}|_{\mathbf{x}}$$

$$= \sum_{k=1}^{3} \frac{\partial f}{\partial z_k}\Big|_{\mathbf{z}=\mathbf{g}(x)} \frac{\partial g_k}{\partial x_2}|_{\mathbf{x}}$$

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

- next step: Write the result as a linear operation between two directional derivative terms

$$\frac{\partial f \circ \mathbf{g}}{\partial x_2}|_{\mathbf{x}} = \frac{\partial f}{\partial z_1}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_1}{\partial x_2}|_{\mathbf{x}} + \frac{\partial f}{\partial z_2}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_2}{\partial x_2}|_{\mathbf{x}} + \frac{\partial f}{\partial z_3}\Big|_{\mathbf{g}(\mathbf{x})} \frac{\partial g_3}{\partial x_2}|_{\mathbf{x}}$$

$$= \sum_{k=1}^{3} \frac{\partial f}{\partial z_k}\Big|_{\mathbf{z}=\mathbf{g}(x)} \frac{\partial g_k}{\partial x_2}|_{\mathbf{x}}$$

$$= \left(\frac{\partial f}{\partial \mathbf{z}_{[:]}}\right) \cdot \left(\frac{\partial \mathbf{g}_{[:]}}{\partial x_2}\right)$$ as inner product, [:] denotes what makes it a vector

$$= \nabla f|_{\mathbf{g}(\mathbf{x})} \cdot \left(\frac{\partial \mathbf{g}_{[:]}}{\partial x_2}|_{\mathbf{x}}\right)$$ gradient of f and a slice of the Jacobi-matrix of g

$$= \nabla f|_{\mathbf{g}(\mathbf{x})} \cdot (J(\mathbf{g}|_{\mathbf{x}})^\top e^{(i)}) = \nabla f|_{\mathbf{g}(\mathbf{x})}^\top J(\mathbf{g}|_{\mathbf{x}})^\top e^{(2)}$$ chain of linear mappings

$$= J(f|_{\mathbf{g}(\mathbf{x})})^\top J(\mathbf{g}|_{\mathbf{x}})^\top e^{(2)}$$

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

One can remember the chain rule also as follows (now f is also vector valued):

$$D(\mathbf{f} \circ \mathbf{g})|_{\mathbf{x}}[\mathbf{e}^{(l)}] =$$

$$= J(\mathbf{f}|_{\mathbf{g(x)}})^\top J(\mathbf{g}|_{\mathbf{x}})^\top e^{(l)} \qquad \text{... as chain of Matmul of Jacobi}^\top$$

$$= D\mathbf{f}|_{\mathbf{g(x)}}[D\mathbf{g}|_{\mathbf{x}}[\mathbf{e}^{(l)}]] \qquad \text{...as chain of linear operations}$$

Chain rule

for any general direction vector $\mathbf{v}$:

$$D(\mathbf{f} \circ \mathbf{g})|_{\mathbf{x}}[\mathbf{v}] =$$

$$= J(\mathbf{f}|_{\mathbf{g(x)}})^\top J(\mathbf{g}|_{\mathbf{x}})^\top \mathbf{v} \qquad \text{... as chain of Matmul of Jacobi}^\top$$

$$= D\mathbf{f}|_{\mathbf{g(x)}}[D\mathbf{g}|_{\mathbf{x}}[\mathbf{v}]] \qquad \text{...as chain of linear operations}$$

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

Chain rule

for any general direction vector $\mathbf{v}$:

$$D(\mathbf{f} \circ \mathbf{g})|_{\mathbf{x}}[\mathbf{v}] =$$
$$= J(\mathbf{f}|_{\mathbf{g}(\mathbf{x})})^\top J(\mathbf{g}|_{\mathbf{x}})^\top \quad \text{... as chain of Matmul of Jacobi}^\top$$
$$= D\mathbf{f}|_{\mathbf{g}(\mathbf{x})}[D\mathbf{g}|_{\mathbf{x}}[\mathbf{v}]] \quad \text{...as chain of linear operations}$$

This extends to 3 or more mappings:

$$D(\mathbf{f} \circ \mathbf{g} \circ \mathbf{h} \circ \mathbf{k})|_{\mathbf{x}}[\mathbf{v}] =$$
$$= J(\mathbf{f})^\top J(\mathbf{g})^\top J(\mathbf{h})^\top J(\mathbf{k})^\top \mathbf{v}$$

with the respective function arguments, i.e. f at $(\mathbf{g} \circ \mathbf{h} \circ \mathbf{k})(\mathbf{x})$

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

Chain rule

for any general direction vector $\mathbf{v}$:

$$D(\mathbf{f}\circ\mathbf{g})|_{\mathbf{x}}[\mathbf{v}] =$$

$$= J(\mathbf{f}|_{\mathbf{g}(\mathbf{x})})^\top J(\mathbf{g}|_{\mathbf{x}})^\top \mathbf{v} \qquad \text{... as chain of Matmul of Jacobi}^\top$$

$$= D\mathbf{f}|_{\mathbf{g}(\mathbf{x})}[D\mathbf{g}|_{\mathbf{x}}[\mathbf{v}]] \qquad \text{...as chain of linear operations}$$

high level:

Concatenation of functions $f\circ g(\mathbf{x}) \leftrightarrow$ Concat their linearizations $Df|_{g(\mathbf{x})}[\cdot]$, $Df|_{\mathbf{x}}[\cdot]$ / respectively $J(\mathbf{f})^\top$, $J(\mathbf{g})^\top$

- There is a summing (k) which implements a chaining of linear mappings $D(\mathbf{f}\circ\mathbf{g})|_{\mathbf{x}}[\mathbf{e}^{(l)}] = \frac{\partial \mathbf{f}\circ\mathbf{g}}{\partial x_l}|_{\mathbf{x}} = \sum_k \frac{\partial \mathbf{f}}{\partial z_k}|_{\mathbf{g}(\mathbf{x})}\frac{\partial g_k}{\partial x_l}|_{\mathbf{x}}$
- The summing runs for the outer function (f) over the partial derivatives of the input variables
- The summing runs for the inner function ($\mathbf{g}$) over its output components.

Chain rule

- next: motivate why the chaining of linear mappings is something to be expected.
- consider the 1-dim case:

$$r(x) = f(g(x)) \tag{20}$$
$$r'(x) = f'(g(x))g'(x) \tag{21}$$

this is almost a concatenation of linear mappings!

  - every real number $a \in \mathbb{R}$ defines a linear mapping via:

$$L_a : \mathbb{R} \to \mathbb{R},\ x \in \mathbb{R} :\ L_a(x) = ax$$

- $f'(g(x))$ defines a linear mapping $L_{f'(g(x))}[v] = f'(g(x))v$
- $g'(x)$ defines a linear mapping $L_{g'(x)}[h] = g'(x)h$
- now:

$$r'(x)h = f'(g(x))g'(x)h = f'(g(x))L_{g'(x)}[h] = L_{f'(g(x))}[L_{g'(x)}[h]]$$

we have a concatenation of mappings already in the 1-dim case

Chain rule

N-dim case: concatenation of linear mappings $Df|_{\mathbf{g}(\mathbf{x})}[\cdot]$ and $D\mathbf{g}|_{\mathbf{x}}[\cdot]$

$$\begin{aligned} e(\mathbf{x}) &= f(\mathbf{g}(\mathbf{x})) && (22)\\ De|_{\mathbf{x}}[\mathbf{v}] &= Df|_{\mathbf{g}(\mathbf{x})}[D\mathbf{g}|_{\mathbf{x}}[\mathbf{v}]] && (23)\\ Df &\text{ is derivated at point } \mathbf{g}(\mathbf{x}) \text{ in direction } \mathbf{c} = D\mathbf{g}|_{\mathbf{x}}[\mathbf{v}] && (24) \end{aligned}$$

This linear algebra perspective can be useful for functions written in linear algebra terms (see below)

Preview:

- Backpropagation computes gradients via chainrule along the (directed) edges in the neural network graph.
- additional feature: made efficient by expressing the chainrule terms via matrix multiplications

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Takeaway points

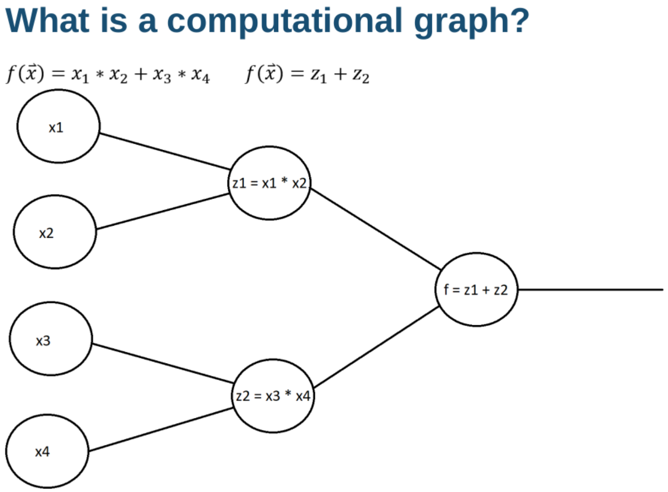
Takeaway points

at the end of this lecture you should be able to:

- computing gradients in pytorch

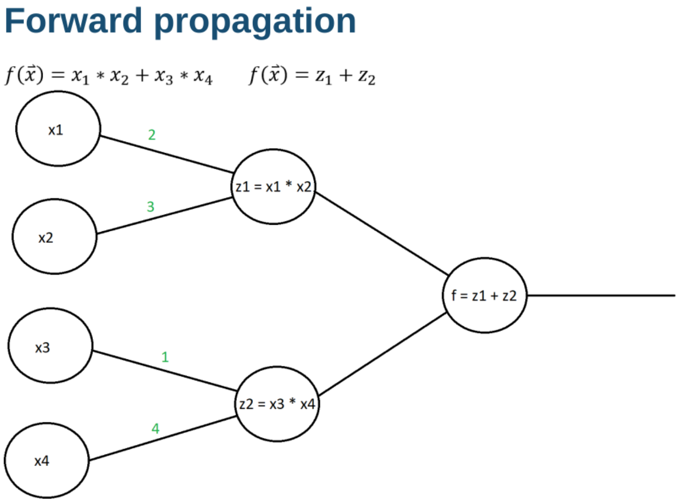
The computational graph

A directed-graph representation of computations done.



The computational graph

Naming convention: Forward pass: compute your function



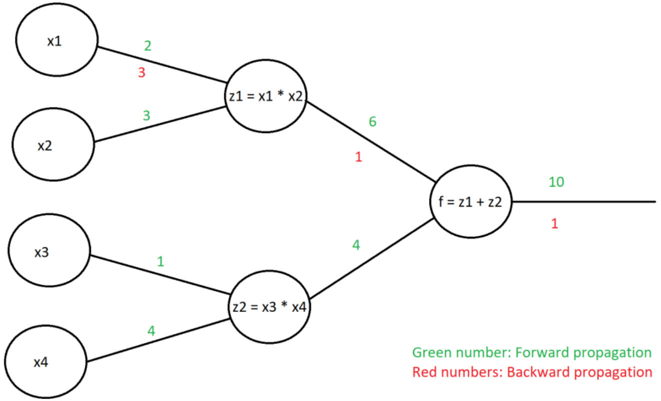
The computational graph

Naming convention: Backward pass: computing derivatives of your function

Backward propagation

What if we want to get the derivative of *f* with respect to the *x1*?

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2 \qquad \frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$


x1
x2
x3
x4
z1 = x1 * x2
z2 = x3 * x4
f = z1 + z2
2
3
3
6
1
1
4
4
10
1
Green number: Forward propagation
Red numbers: Backward propagation

Autograd

How to get some gradients of some computations ?

- You can define a sequence of computations
- then call `.backward()` or `torch.autograd.grad(...)`.

```
import torch
import numpy as np

a=torch.tensor(0.25*np.ones((2),dtype=np.float32) ,requires_grad=True)
b=torch.tensor( 2*np.ones((2),dtype=np.float32),requires_grad=True)
c=torch.tensor( 3*np.ones((2),dtype=np.float32),requires_grad=True)

d=a*b #element-wise product of two (2)-dim vectors
e=torch.dot ( d,c) #the computations until here
print('e.requires_grad? ',e.requires_grad)

e.backward() # computes gradients

print( 'de/dc', c.grad) #gradients in all tensors marked with
print( 'de/db', b.grad) #requires_grad=True
print( 'de/da', a.grad)
```

Autograd

When gradients are computed and stored for a tensor ?

- If tensors are leaf tensors and have their `requires_grad=True` attribute set, then they are marked for tracking operations along the computation sequence for later gradient computation.
- leaf tensor: A tensor created by the user .

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py

Autograd

Autograd: Automatic differentiation with respect to tensors used in computations.

How does it work?

- It records all operations when a function is executed as a graph

```
import torch
import numpy as np

def print_graph(g, level=0):
    if g == None:
      return
    print('*'*level*1, g)
    for subg in g.next_functions:
        print_graph(subg[0], level+1)

if __name__ == '__main__':
  a=torch.ones((2,1),requires_grad=True) #requires_grad=True !!
  b=torch.tensor( 2*np.ones((2,1),dtype=np.float32),requires_grad=True)
  c=torch.tensor( 3*np.ones((2,1),dtype=np.float32),requires_grad=True)
  #computations
  d=a+b
  e=d*c
  print(e)
  print_graph(e.grad_fn, 0)
```

Autograd

Autograd: Automatic differentiation with respect to tensors used in computations.

How does it work?

- It records all operations when a function is executed as a graph
- when one asks Pytorch to compute a gradient, it used the graph and performs chain rule along the graph to get the gradient

Autograd: Automatic differentiation with respect to tensors used in computations.

Autograd:

- You can define a function or any compute sequence
- then call `.backward()` or `torch.autograd.grad(...)`.

```
import torch
import numpy as np

def somefunction(a):
  b= a[:,1] #torch.sum(a,dim=1) #b.shape = (3)
  e= b[0]**2 -2.0*torch.exp(b[1]) +3.0*b[2]
  return e

if __name__ == '__main__':
  a=torch.randn( (3,2) )
  a.requires_grad=True

  r = somefunction(a)
  r.backward() # computes gradients

  print( 'dr/da', a.grad  )
  print( 'input value of tensor a as numpy: a.data.numpy(): \n',  a.data.cpu().numpy()  )
  print( 'de/da as numpy: a.grad.numpy():\n ', a.grad.cpu().numpy()  )
```

Autograd: Automatic differentiation with respect to tensors used in computations.

Autograd:

- You can define a sequence of computations
- then call `.backward()` or `torch.autograd.grad(...)`.

```
import torch
import numpy as np

def somefunction2(a,x):
  b= torch.mm(  x.unsqueeze(0), a ).squeeze(0)
  e = 2*b[0]+b[1]**2
  return e

if __name__ == '__main__':
  a=torch.randn( (3,2) )
  a.requires_grad=True
  x=torch.randn( (3) )
  x.requires_grad=True

  r = somefunction2(a,x)
  r.backward() # computes gradients

  print( 'dr/da', a.grad  )
  print( 'dr/dx', x.grad  )
```

Autograd

...

if `e` is a tensor with 1 element, then `e.backward()` computes the gradient of `e` with respect to all its inputs that are leaf tensors involved in computing `e` and are marked with `requires_grad=True`.

Autograd

See the pytorch fmnist training code for logistic regression or the 2-/3-layer NN:

Question: Why using `.backward()` does compute gradients with respect to all model parameters?

- it knows what was defined as instances of the `class torch.nn.Parameter`
- useful: you can list all parameters of a neural network:

```
for nm,param in module.named_parameters():
  print('name:',nm, ' shape: ', param.shape)
```

Autograd for tensors with more than one element

If the result of a computation is a tensor $\mathbf{e}(x)$ of $n \geq 2$ elements, then the gradient of it is a matrix, the Jacobi-matrix. Example for 2 elements:

$$\mathbf{e}(\mathbf{x}) = (e_1(\mathbf{x}), e_2(\mathbf{x}))$$
$$\nabla \mathbf{e}|_{\mathbf{x}} = (\nabla e_1|_{\mathbf{x}}, \nabla e_2|_{\mathbf{x}})$$

For result tensors with more than one element

Important: !!! In this case cannot apply `e.backward()` directly.

- instead need to provide a weight tensor $\mathbf{v}$ with the same number of elements as the output $\mathbf{e}(x)$, and to multiply the output $\mathbf{e}(x)$ with $\mathbf{v}$ as inner product.
- the inner product $\mathbf{e}(x) \cdot \mathbf{v}$ is again a tensor with 1 element.
- Then compute the gradient of the scalar $\mathbf{e}(x) \cdot \mathbf{v} = e_1 v_1 + e_2 v_2 + \ldots + e_n v_n$ with respect to the input tensors $\mathbf{x}$ of $\mathbf{e}(\mathbf{x})$.

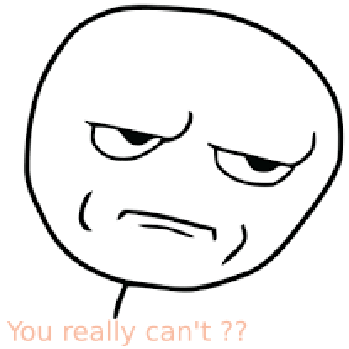
Autograd for tensors with more than one element

- example of a function which returns a tensor with more than one element:

```
import torch
import numpy as np

def somefunction2(a,x):
    b= torch.mm(  x.unsqueeze(0), a ).squeeze(0) # 2-tensor
    return b

if __name__=='__main__':

    a=torch.randn( (3,2) )
    a.requires_grad=True
    x=torch.randn( (3) )
    x.requires_grad=True

    r = somefunction2(a,x)
    #r.backward() # ERROR
    weight = torch.tensor([2., 1.])  #r inner product weight = scalar!
    r.backward(weight) # computes gradients

    print( 'dr/da', a.grad  )
    print( 'dr/dx', x.grad  )
```

Autograd

If you feel not up to computing derivatives for some functions



Pytorch can do that for you!

autograd

Autograd

- Autograd tracks the graph of computations when one runs code
- Tracked computations will be used to compute a gradient automatically
- use `with torch.no_grad():` environment to **not** record computations for gradient calculations for some larger block of code that is reused – use case: everything outside of handling training data, e.g. computing validation or test scores.[a]

[a]Why you dont want to track gradient computations in this case?

- out of exams: for GAN-training `sometensor.detach()` prevents the gradient flowing from `sometensor` to all those parts used to compute sometensor.

autograd

Note: If you have a tensor with attached gradient, then the `.data` stores the tensor values, and `.grad.data` the gradient values

```
vals=x.data.cpu().numpy() #exports function values to numpy
g_vals=x.grad.data.cpu().numpy() #exports gradient values to numpy
```

I hope this messing around with gradients helps you!!

gradients are not that dangerous

Takeaway:

- directional derivatives $\leftrightarrow$ partial derivatives
- directional derivatives and gradient
- other topics ergarding gradients:
  - vector-valued function $\rightarrow$ Jacobi-matrix
  - next lecture: derivative of a bilinear function
- Computing gradients in PyTorch

Takeaway:

https://www.youtube.com/watch?v=dp8zV3YwgdE

Are we all doomed ?

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Problem

How to find the trainable parameters **w** in classification ? We have, for example for logistic regression:

$$f(x) = \mathbf{w} \cdot \mathbf{x} + b$$
$$g(x) = \frac{1}{1 + e^{-f(x)}}$$

and some training, test and validation samples (x, y)

- first attempt: compute an average loss function on a training data set D of size n

$$L = \frac{1}{n} \sum_{(x,y) \in D} (-y \ln g(\mathbf{x}) - (1 - y) \ln(1 - g(\mathbf{x})))$$

- L depends on trainable parameters $u = (w, b)$
- next step: find parameters (w, b) which minimize this loss

A coarse solution

How to find the trainable parameters w 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 $L(u)$
- goal: find $u^* = \mathrm{argmin}_u L(u)$
- assumption: can compute $\nabla^{(u)} L(v)$ - the gradient in point $u = v$ with respect to the variables u.
- use a variant of Gradient Descent

Remember Logistic regression

- next: why is the gradient with respect to u useful to find a local minimum of function $L(u)$?

Gradients

Which direction $\mathbf{v}$ minimizes $\delta_{\mathbf{v}} f|_{\mathbf{x}} = \nabla f|_{\mathbf{x}} \cdot \mathbf{v}$
Fact: the optimization problem

$$\mathrm{argmin}_{\mathbf{v}:||\mathbf{v}||=1} \mathbf{w} \cdot \mathbf{v}$$

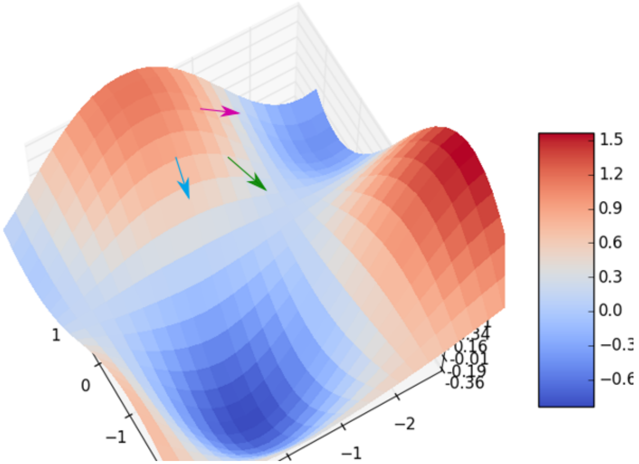
is solved by $\mathbf{v} := -1 * \frac{\mathbf{w}}{||\mathbf{w}||}$. This has an important consequence!: $-\frac{\nabla f|_{\mathbf{x}}}{||\nabla f|_{\mathbf{x}}||}$ is the direction with the highest negative slope of f in $\mathbf{x}$.

Gradient and local function change

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

why the gradient can inform optimization

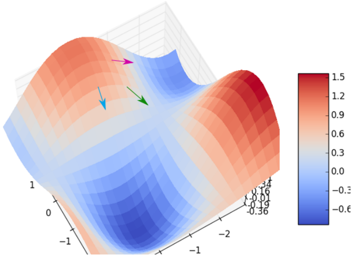
Idea: negative gradient at a point *w* is the direction of steepest function decrease from *w*, if the step size is sufficiently small.



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

Why the gradient can be used for optimization

Idea: negative gradient at a point u is the direction of steepest function decrease from u, if the step size is sufficiently small.



- how to use it for an algorithm ?
- take a small step into the direction of the negative gradient:
- Let $u^{(old)}$ be the parameters at the current step , then

$$u^{(new)} = u^{(old)} - \eta \nabla^{(u)} L(u^{(old)}), \ \eta > 0 \text{ is the stepsize}$$

- next: write this as an algorithm!

Gradient Descent Algorithm

Gradient Descent

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

- given: step size parameter η, initialize start vector $u^{(0)}$ to a value.
- run while loop, until function value changes (δ_L) drop below a threshold, do at iteration t:
  - $u^{(t+1)} = u^{(t)} - \eta \nabla^{(u)} L(u^{(t)})$
  - compute change of objective to last value: $\delta_L = \|L(u^{(t+1)}) - L(u^{(t)})\|$

immediate conclusions:

- minimizing the gradient on training data ensures low loss on training data
- does not guarantee low losses on new unseen test data (cf. overfitting)

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Gradient Descent Algorithm

Gradient Descent

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

- given: step size parameter η, initialize start vector $u^{(0)}$ to a value.
- run while loop, until function value changes (δ_L) drop below a threshold, do at iteration t:
  - $u^{(t+1)} = u^{(t)} - \eta \nabla^{(u)} L(u^{(t)})$
  - compute change of objective to last value: $\delta_L = \|L(u^{(t+1)}) - L(u^{(t)})\|$

questions:

- sensitivity to starting point
- sensitivity to learning rate
- quality of obtained solutions

Gradient Descent: sensitivity to starting point

Lets explore starting point effects:

- in `learnThu8.py` 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.

possible problems of gradient descent I

- we find a local minimum, not the global minimum of a function. Local optimum can be good or bad.
- effects of starting point – for non-convex functions: different starting point leads to possibly different solutions u^*.

Gradient Descent: sensitivity to starting point

Lets explore starting point effects:

- in `learnThu8.py` 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.

solutions with respect to starting point

- Do not train from scratch for any practical applications. ALWAYS fine-tune a neural net pretrained on a large corpus like ImageNet (later lecture)
- if – in rare cases – one would train from scratch (e.g. a totally new architecture), then careful initialization, and special learning rate treatments are important (later lecture)

see e.g. https://arxiv.org/abs/1502.01852 for examples how initialization matters for training deep neural networks

training deep neural networks from scratch without care about initialization can result in very bad performance

Gradient Descent: sensitivity to learning rate

Lets explore learning rate effects:

- in `learnThu8.py` run `tGD([stepsize])` to see the effect of different stepsizes.

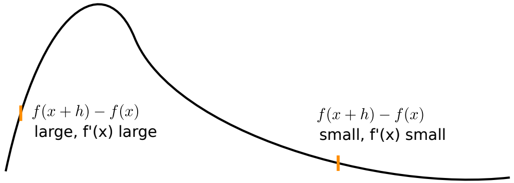
effects of bad stepsize choices:

- too large $\Rightarrow$ divergence/no solution.
- too small $\Rightarrow$ slow convergence

Gradients

possible problems of gradient descent

- the size of the update step $u_{t+1} = u_t - \eta \nabla_u L(u_t)$ depends on the norm of the gradient $\|\nabla L(u)\|$, too. So when starting in a steep region ($\|\nabla L(u)\|$ is large), even a small stepsize can lead to divergence.



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

learning rate adjustment

- one way to deal with the question of how to set the stepsize, is to reduce it over time:

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.

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

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

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

a[1]

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

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Batch gradient descent

Consider setting with an average of **losses over all training samples**, and a predictor which depends on trainable parameters u:

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

The application of gradient descent to a loss computed over all training samples results in the following algorithm:

$$u_{t+1} = u_t - \eta \nabla_u \left(\frac{1}{n}\sum_{i=1}^{n} L(f_{u_t}(x_i), y_i) \right)$$

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

Stochastic gradient descent / minibatches

The alternative is **stochastic gradient descent** (SGD). 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 all training samples. After updating the parameters, one draws a new randomly selected subset of all training samples for gradient computation.

SGD in practice

- shuffle/permute your training data $\{z_i = (x_i, y_i), i = 0, \ldots, n-1\}$[2],
- iterate over minibatches , computing the gradient of the loss in each iteration – until all data has been used once
- repeat the two above steps

For example, stochastic gradient descent when starting at index m and using the next k samples uses as update:

$$u_{t+1} = u_t - \eta \nabla_u \frac{1}{k} \sum_{i=m+0}^{m+k-1} L(f_{u_t}(x_i), y_i)$$

[2]make sure that features x and labels y of one sample (x, y) will stay together!!

Gradients

stochastic gradient descent for an average of losses

- initialize start vector u_0 as something, choose step size parameter η
- shuffle/permute your training data $\{z_i = (x_i, y_i), i = 0, \ldots, n-1\}$
- until all data has been used once:
  - compute the gradient of the loss for the next k samples (here it starts at an index m)
  - apply it to update the parameters of the mapping f:

$$u_{t+1} = u_t - \eta \nabla_u \frac{1}{k} \sum_{i=m+0}^{m+k-1} L(f_{u_t}(x_i), y_i)$$

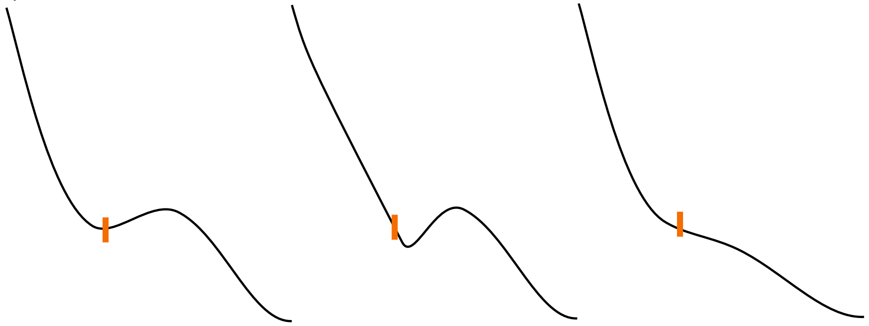
- measure loss on validation data . If low enough, stop.
- otherwise repeat from the shuffle step

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 (it is based on a random subset, that causes the noise).
    - injecting small noise is one way to prevent overfitting! For deep neural networks SGD can be better than full batch gradient descent in finding *good* local optima.

Advantages of stochastic gradient descent

- 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 right it allows to jump out of the local minimum.

Outline

1 Motivation

2 The derivative of a 1dim function and the Gradient

3 The directional derivative and Derivatives of Linear mappings

4 Chain Rule

5 Gradients in PyTorch and Autograd

6 Gradient Descent

7 Properties of Gradient Descent

8 Stochastic gradient descent

Why gradient descent is hard in practice?

What makes it hard to use gradient descent in deep neural networks?

- one cannot simply stack convolution layers: see for example Fig 1 in https://arxiv.org/abs/1512.03385
- the problem of vanishing or exploding gradients
  - the graphics on gradient magnitudes in http://neuralnetworksanddeeplearning.com/chap5.html

Challenge 1: vanishing gradients

What makes it hard to use gradient descent in deep neural networks?

- the problem of vanishing or exploding gradients
  - vanishing gradients: scale of gradient gets smaller and smaller as neural network architectures become deeper - in particular in layers closer to the input

Challenge 2: exploding gradients

What makes it hard to use gradient descent in deep neural networks?

- the problem of vanishing or exploding gradients
  - exploding gradients: scale of gradient gets unbounded as neural network architectures become deeper, if learning rates are too high

Challenge 3: imbalanced gradient scales

What makes it hard to use gradient descent in deep neural networks?

- the problem of vanishing or exploding gradients
  - imbalanced gradient scales (e.g. Bjorck et al, Understanding Batchnorm https://arxiv.org/abs/1806.02375): the scale of gradients has large variation across neurons - this means some neurons have faster updates than others