# Automatic differentiation

Jill-Jênn Vie

Oct 3, 2025

# Outline

Data $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ targets $\boldsymbol{y} \in \mathcal{Y}^n$
Model $f : \boldsymbol{x} \mapsto \hat{y} = f(\boldsymbol{x})$

## Activation and loss $\mathcal{L}$

Why sigmoid? Why ReLU?

## Optimizers

Why stochastic gradient descent?

## Automatic differentiation

Why backpropagation? What does `loss.backward()` do?

## Unsupervised, semi-supervised, self-supervised

What if we have very few labels $\boldsymbol{y}$ or only $\boldsymbol{X}$?


@untitled01ipynb

# Optimization

Find the best parameters $\boldsymbol{\theta}$ to reach a goal: making $f_{\boldsymbol{\theta}}(x)$ close to $y$

Usually, minimize a differentiable loss function

## Regression: continuous target

Ex. $y \in \mathbb{R}$ $\quad \mathcal{L} = \sum_i (\hat{y}_i - y_i)^2$ squared error

## Classification: discrete target

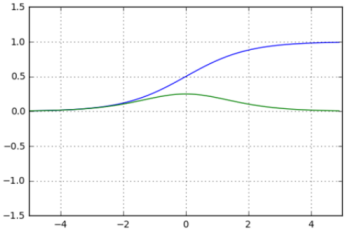
Ex. $y \in \{0, 1\}$

- Logistic regression $\hat{y} = \text{sigmoid}(\boldsymbol{w}^T \boldsymbol{x}) \in \{0, 1\}$
- Logistic loss: $\mathcal{L}_i(p, y) = y \log p + (1 - y) \log(1 - p)$

Ex. $y \in \{1, \ldots, C\}$

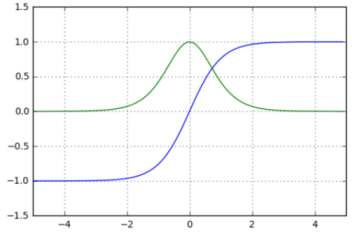
- Multinomial regression $\hat{y} = \text{softmax}(\boldsymbol{W}\boldsymbol{x} + b) \in \{0, 1\}^C$
- Cross-entropy loss: $\mathcal{L}_i(\boldsymbol{p}, y) = \sum_{c=1}^{C} y_c \log p_c = \log p_y$

# Activation / link functions; why sigmoid, softmax?



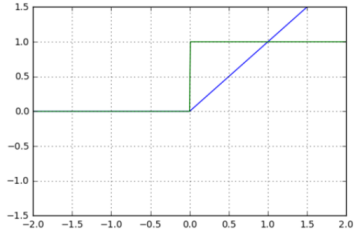

$$\text{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\text{sigm}'(x) = \text{sigm}(x)(1-\text{sigm}(x))$$




$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$



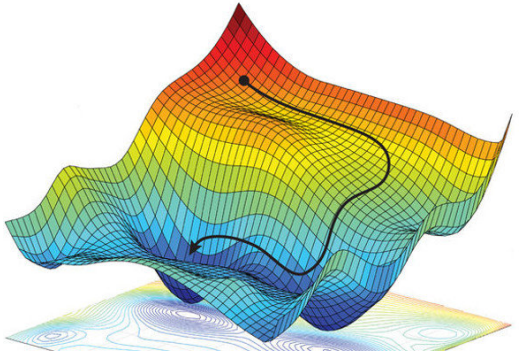

$$\text{relu}(x) = \max(0, x)$$

$$\text{relu}'(x) = 1_{x>0}$$

(Ollion & Grisel)

| | Binary | Multiclass |
|---|---|---|
| $f$ | softplus : $x \mapsto \log(1+\exp(x))$ | logsumexp$^+$ : $\boldsymbol{x} \mapsto \log(1+\sum_c \exp(x_c))$ |
| $f'$ | sigmoid : $x \mapsto 1/(1+\exp(-x))$ | softmax : $\boldsymbol{x} \mapsto \exp(\boldsymbol{x})/\sum_c \exp(x_c)$ |
| $f''$ | $x \mapsto \text{sigmoid}(x)(1-\text{sigmoid}(x))$ | $\boldsymbol{x} \mapsto \text{diag}(s(\boldsymbol{x})) - s(\boldsymbol{x})s(\boldsymbol{x})^T$ |

# Gradient descent



$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

```
for each epoch:
    for x, y in dataset:
        compute gradients ∂L/∂θ (f_θ(x), y)   # also noted ∇_θ L
        θ ← θ − γ ∇_θ L   # γ is the learning rate
```

# SGD in PyTorch

```
import torch

# define model, n_epochs, trainloader
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

for _ in range(n_epochs):
    for inputs, targets in training:
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

# Variants

## Batch gradient descent

Compute the gradient of loss on all examples and update parameters

## Gradient descent

For each example $\boldsymbol{x}_i, y$ update parameters

## Stochastic gradient descent

Sample examples $\boldsymbol{x}_i, y$ and update parameters

## Minibatch gradient descent

Sample a batch of examples, estimate full gradient from batch and update parameters

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \gamma \frac{n}{B} \sum_{i \in B} \mathcal{L}(\boldsymbol{x}_i, y)$$

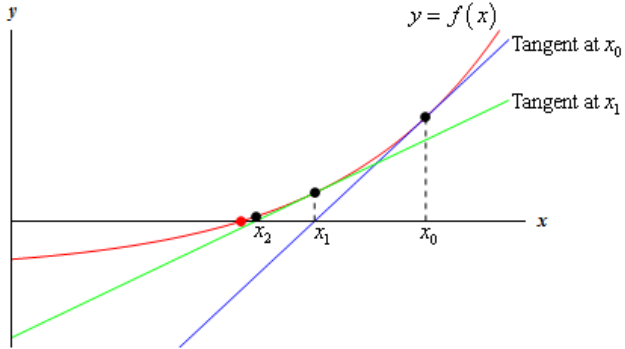
# Why SGD?

To minimize a function

Find the zeroes of its derivative

How to find the zeroes of a function?

# Newton's method: find $x$ such that $f(x) = 0$




$f : \mathbb{R} \to \mathbb{R}$ differentiable, $\nexists x, f'(x) = 0$

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$

Quadratic convergence

$$\exists C > 0, |x_{t+1} - \ell| \leq C|x_t - \ell|^2$$

# In higher dimension

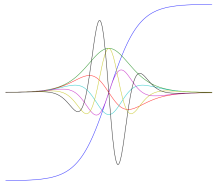
Let $g : \mathbb{R}^n \to \mathbb{R}$ twice differentiable, with $n$ big

What is the size of $g'(\boldsymbol{x})$? Usually noted $\dfrac{\partial g}{\partial \boldsymbol{x}}$ or $\nabla_{\boldsymbol{x}} g$.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \underbrace{g''(\boldsymbol{x}_t)^{-1}}_{\in \mathbf{R}^{n\times n},\ O(n^3)} \underbrace{g'(\boldsymbol{x}_t)}_{\in \mathbf{R}^n}$$

# How to compute gradients automatically? What's in `loss.backward()`?

```
from autograd import elementwise_grad as egrad
import matplotlib.pyplot as plt
x = np.linspace(-7, 7, 200)
plt.plot(x, tanh(x),
       x, egrad(tanh)(x),                                    # first
       x, egrad(egrad(tanh))(x),                             # second
       x, egrad(egrad(egrad(tanh)))(x),                      # third
       x, egrad(egrad(egrad(egrad(tanh))))(x),               # fourth
       x, egrad(egrad(egrad(egrad(egrad(tanh)))))(x),        # fifth
       x, egrad(egrad(egrad(egrad(egrad(egrad(tanh))))))(x))  # sixth
plt.show()
```



# Existing methods and their limitations

## Numerical differentiation

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

Round-off errors

## Symbolic differentiation

Have to keep symbolic expressions at each step of the process

## Automatic differentiation

# Chain rule

$$(f \circ g)' = g' \cdot (f' \circ g)$$

## Generalized chain rule

$$\frac{df_1}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3} \cdots \frac{df_n}{dx}$$

# Reminder: Jacobians

Consider differentiable $f : \mathbb{R}^n \to \mathbb{R}^m$, its Jacobian $J_f \in \mathbb{R}^{m \times n}$ contains its first-order partial derivatives $(J_f)_{ij} = \dfrac{\partial f_i}{\partial x_j}$:

$$J_f = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} & \cdots & \dfrac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

## Example: linear layer

$W$ is a $m \times n$ matrix

If $f(\boldsymbol{x}) = W\boldsymbol{x}$

$f_i = W_i \boldsymbol{x} = \sum_j W_{ij} x_j$ where $W_i$ is $i$th row of $W$

$(J_f)_{ij} = \dfrac{\partial f_i}{\partial x_j} = W_{ij}$ so $J_f = W$

# Generalized multivariate chain rule

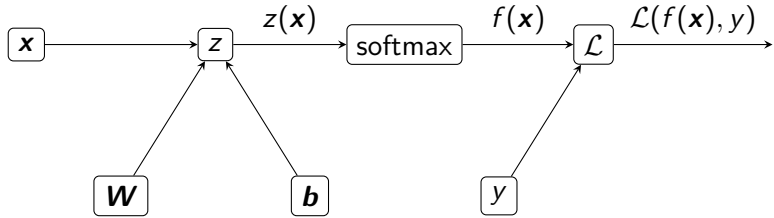
Consider differentiable $f : \mathbb{R}^m \to \mathbb{R}^k, g : \mathbb{R}^n \to \mathbb{R}^m$ and $\boldsymbol{a} \in \mathbb{R}^n$.

$$D_{\boldsymbol{a}}(f \circ g) = D_{g(\boldsymbol{a})} f \circ D_{\boldsymbol{a}} g$$

So the Jacobians verify:

$$J_{f \circ g} = (J_f \circ g) J_g$$

# Computation graph



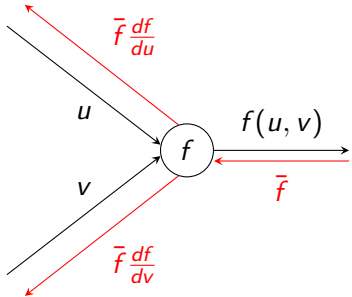

$$\frac{d\mathcal{L}}{db_1} = \frac{d\mathcal{L}}{df}\frac{df}{db_1} = \frac{d\mathcal{L}}{df}\left(\frac{df}{dz}\frac{dz}{db_1}\right) \text{ (forward)}$$
$$= \frac{d\mathcal{L}}{dz}\frac{dz}{db_1} = \left(\frac{d\mathcal{L}}{df}\frac{df}{dz}\right)\frac{dz}{db_1} \text{ (backward)}$$

Properly written: $J_{\mathcal{L}\circ\text{softmax}\circ z} = (J_{\mathcal{L}} \circ f)(J_{\text{softmax}} \circ z)J_z$

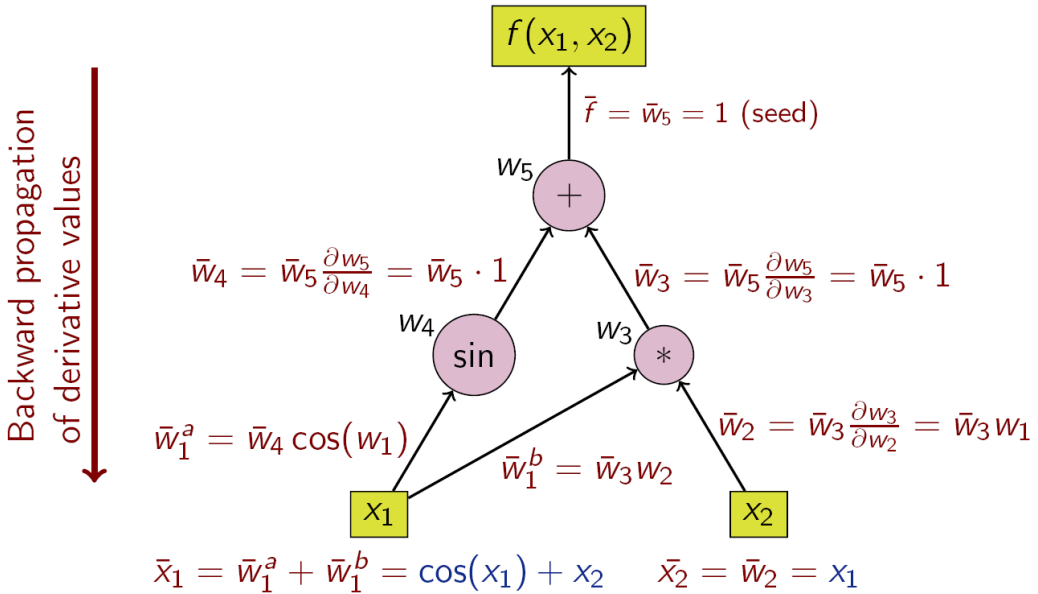
Given that $\boldsymbol{x} \in \mathbb{R}^d, z(\boldsymbol{x}), f(\boldsymbol{x}) \in \mathbb{R}^{d_2}, \mathcal{L}(f(\boldsymbol{x}), y) \in \mathbb{R}$,
which order is better? This is why backpropagation.

# Reverse accumulation (in $\mathbb{R}$)

Let us note the adjoint $\overline{f} \triangleq \frac{d\mathcal{L}}{df}$.

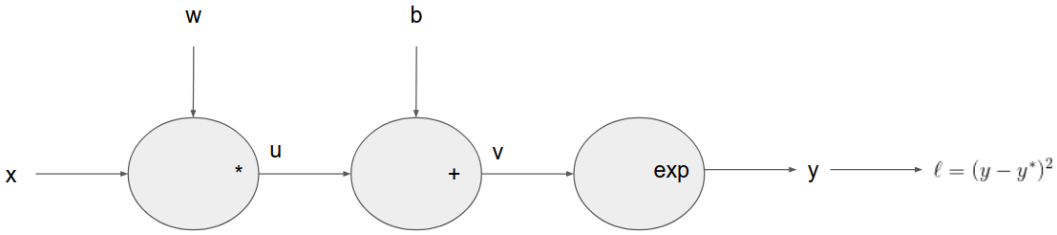


# A complete example (Wikipedia)



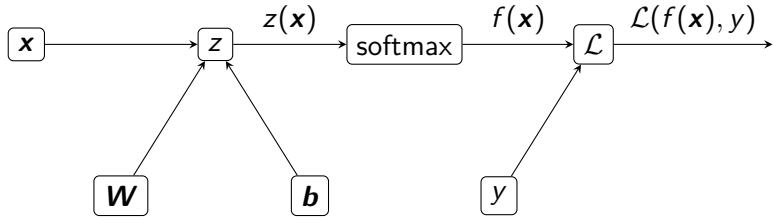
# Notebook: compute gradients

FORWARD PASS




$u = wx$
$v = u + b$
$y = e^v$
$\ell = (y - y^*)^2$

# Computation graph: classifier




Here we define $\mathcal{L}$ as cross-entropy:

$$\mathcal{L}(f(\boldsymbol{x}), y) = -\sum_{c=1}^{K} \mathbf{1}_{y=c} \log f(\boldsymbol{x})_c = -\log f(\boldsymbol{x})_y$$

Compute $\frac{d\mathcal{L}}{dz_c}$

# Unsupervised, semi-supervised, self-supervised

What if we have very few labels $\boldsymbol{y}$ or only $\boldsymbol{X}$?

# Next week: introduction to reinforcement learning

Dynamic programming's *Principle of Optimality* (Bellman, 1952)

$$\left.\begin{array}{rl} \text{value function} & v_\pi(s) \\ \text{action-value function} & q_\pi(s,a) \end{array}\right\} \text{ for policy } \pi : S \to A \to R \to S' \to A'$$

Q-learning (1992), SARSA (1996), policy gradient (2000)