Aplikace neuronových sítí

Zpětná propagace

Minule: Stochastic Gradient Descent (SGD)

- Optimalizujeme parametry $\boldsymbol{\theta}$, tj. např. váhovou matici a bias
- Aktuální hodnota je $\boldsymbol{\theta}_t$
- Nový odhad bude $\boldsymbol{\theta}_{t+1}$
- Všechno jsou vektory

gradient lossu vyhodnocený v bodě $\boldsymbol{\theta}_t$

$$\boldsymbol{\theta}_{t+1} \coloneqq \boldsymbol{\theta}_t - \gamma \cdot \boxed{\nabla L(\boldsymbol{\theta}_t)}$$

krok učení (learning rate)

Minule: Gradient

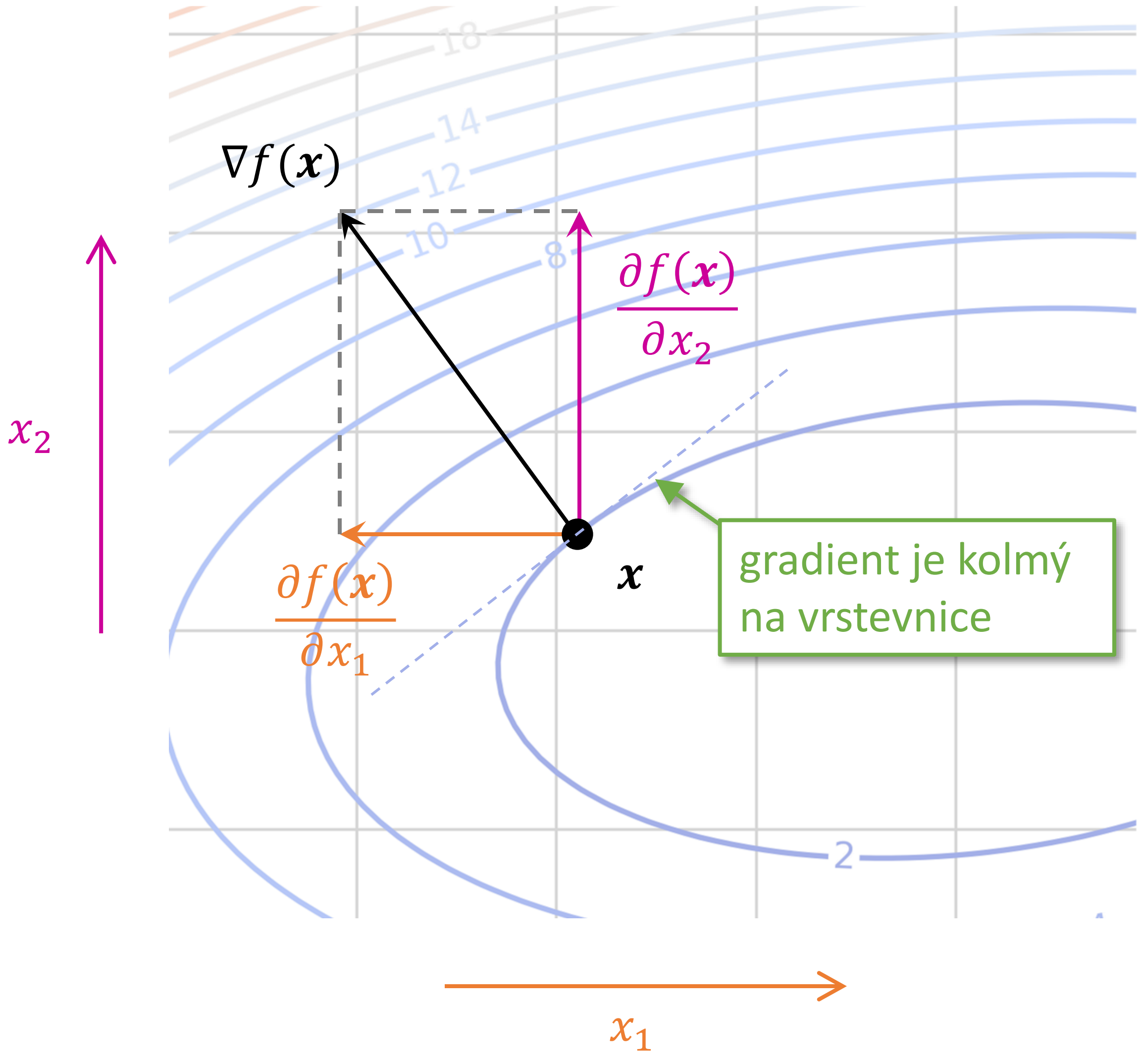
- U funkcí $f(\boldsymbol{x})\colon \mathbb{R}^D \to \mathbb{R}$, tj.

$$f(\boldsymbol{x}) = f(x_1, \dots, x_D)$$

  tedy funkcí s D vstupy a 1 výstupem můžeme derivovat vzhledem ke každému z jednotlivých x_d → **parciální derivace**

- Uspořádání všech D paricálních derivací do vektoru se nazývá **gradient**:

$$\nabla f(\boldsymbol{x}) = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \dots, \frac{\partial f(\boldsymbol{x})}{\partial x_D}\right]^{\mathsf{T}}$$



Minule: Manuální výpočet gradientu

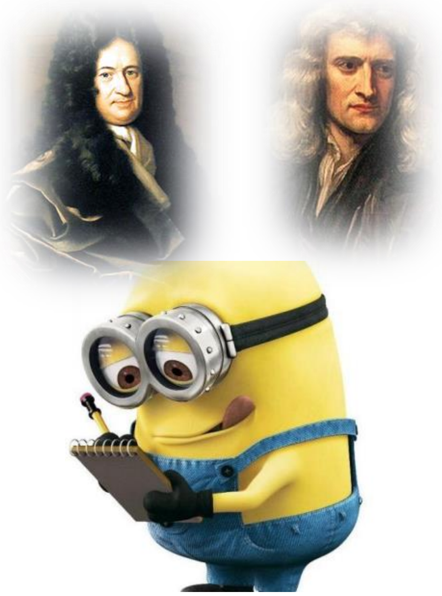
Optimalizovaná funkce (loss) je

$$L(\boldsymbol{w}, \boldsymbol{b}) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\left(\boldsymbol{w}_{y_n,:} \cdot \boldsymbol{x}_n + b_{y_n}\right)}{\sum_{k=1}^{K} \exp\left(\boldsymbol{w}_{k,:} \cdot \boldsymbol{x}_n + b_k\right)}$$

Potřebujeme

$$\frac{\partial L(\boldsymbol{w}, \boldsymbol{b})}{\partial \boldsymbol{w}} = ?$$

$$\frac{\partial L(\boldsymbol{w}, \boldsymbol{b})}{\partial \boldsymbol{b}} = ?$$



$K \times D$

$(K \times 1) \cdot (1 \times D)$

$$\frac{\partial L(\boldsymbol{w}, \boldsymbol{b})}{\partial \boldsymbol{w}} = \frac{1}{N}\sum_{n=1}^{N} (\hat{\boldsymbol{p}}_n - \boldsymbol{p}_n) \cdot \boldsymbol{x}_n^{\top}$$

$(K \times 1)$

$$\frac{\partial L(\boldsymbol{w}, \boldsymbol{b})}{\partial \boldsymbol{b}} = \frac{1}{N}\sum_{n=1}^{N} (\hat{\boldsymbol{p}}_n - \boldsymbol{p}_n)$$

$K \times 1$

Minule: Manuální výpočet gradientu

Optimalizovaná funkce (loss) je

$$L(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\left(\boxed{f(\boldsymbol{x}_n, \boldsymbol{\theta})}_{y_n}\right)}{\sum_{k=1}^{K} \exp\left(f(\boldsymbol{x}_n, \boldsymbol{\theta})_k\right)}$$

Co když $f(\boldsymbol{x}_n, \boldsymbol{\theta})$ potažmo $L(\boldsymbol{\theta})$ jsou složité funkce jako např. hluboké neuronové sítě?

Potřebujeme

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = ?$$

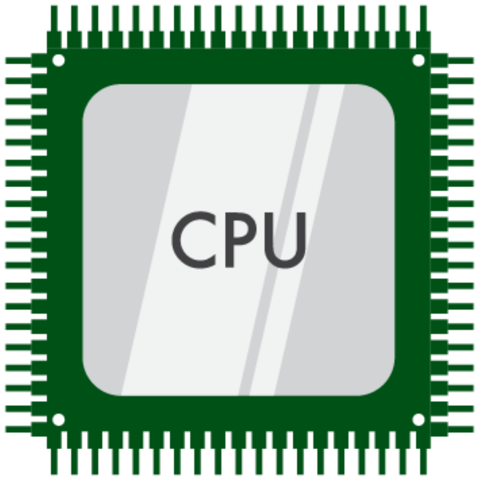
Dnes: Automatický výpočet gradientu

Optimalizovaná funkce (loss) je

$$L(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\left(f(\boldsymbol{x}_n, \boldsymbol{\theta})_{y_n}\right)}{\sum_{k=1}^{K} \exp\left(f(\boldsymbol{x}_n, \boldsymbol{\theta})_k\right)}$$

Potřebujeme

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = ? \longrightarrow$$



$$\longrightarrow \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \ \dots$$

Řetízkové pravidlo

Řetízkové pravidlo (chain rule)

- Pro výpočet derivace složených funkcí formy $z = f(y) = f\left(g(x)\right)$ používáme

$$\boxed{\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}}$$

- “derivace vnější krát derivace vnitřní”

Příklad na řetízkové pravidlo

$$z = (\underbrace{x_1 + 3 \cdot x_2}_{g})^{2} \quad f$$

vnitřní funkce g:

$$y = x_1 + 3 \cdot x_2$$

$$\frac{\partial y}{\partial x_1} = 1 \qquad \frac{\partial y}{\partial x_2} = 3$$

vnější funkce f:

$$z = y^2$$

$$\frac{\partial z}{\partial y} = 2 \cdot y$$

aplikace pravidla:

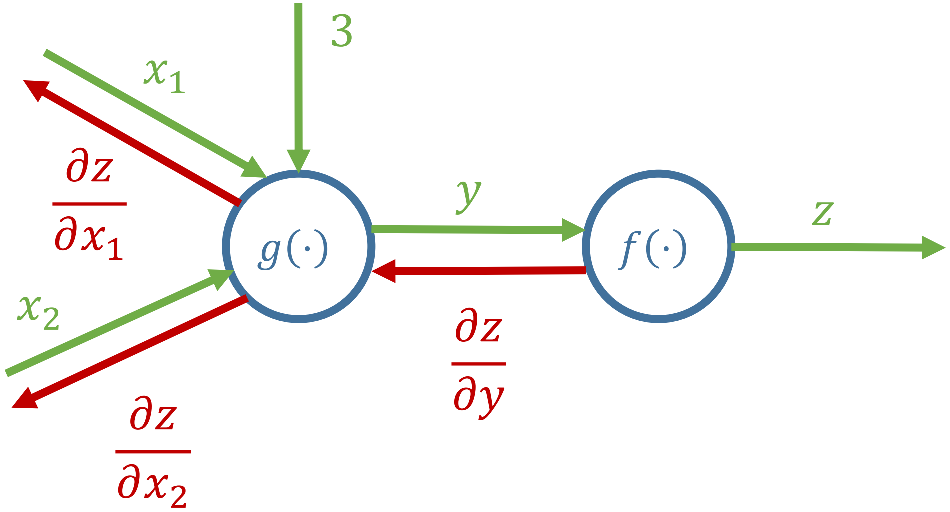
$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = 2 \cdot y \cdot 1 = 2 \cdot (x_1 + 3 \cdot x_2)$$

$$\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_2} = 2 \cdot y \cdot 3 = 2 \cdot 3 \cdot (x_1 + 3 \cdot x_2)$$

Diferencovatelné výpočetní grafy

Funkce $z = (x_1 + 3 \cdot x_2)^2$ jako graf

$$z = (x_1 + 3 \cdot x_2)^2$$



vnitřní funkce g:

$$y = x_1 + 3 \cdot x_2$$

vnější funkce f:

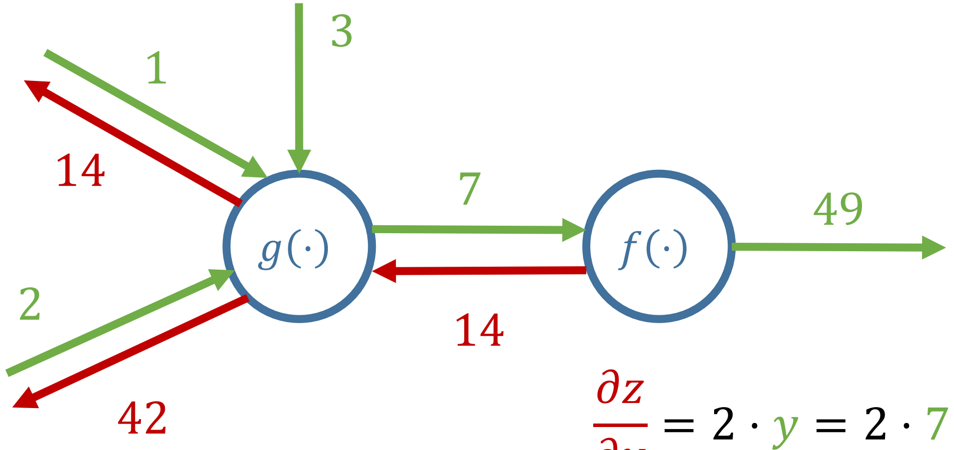
$$z = y^2$$

Funkce $z = (x_1 + 3 \cdot x_2)^2$ jako graf

$$z = (x_1 + 3 \cdot x_2)^2$$

$$\boxed{\begin{aligned} x_1 &= 1 \\ x_2 &= 2 \end{aligned}}$$

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y} \cdot 1 = 14 \cdot 1 = 14$$



$$\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y} \cdot 3 = 14 \cdot 3 = 42$$

$$\frac{\partial z}{\partial y} = 2 \cdot y = 2 \cdot 7 = 14$$

vnitřní funkce g:

$$y = x_1 + 3 \cdot x_2$$

vnější funkce f:

$$z = y^2$$

Funkce $z = (x_1 + 3 \cdot x_2)^2$ jako graf podrobně

dopředný průchod

(1) $x_2' \coloneqq 3 \cdot x_2$

(2) $y \coloneqq x_1 + x_2'$

(3) $z \coloneqq y^2$

zpětný průchod

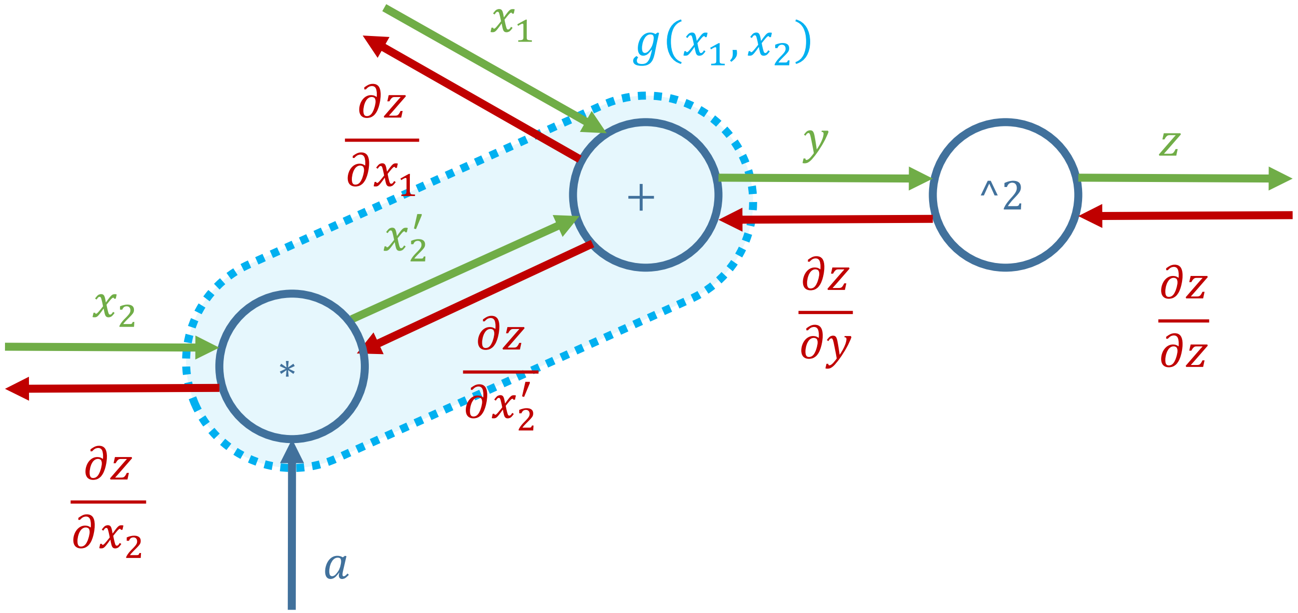
(4) $\frac{\partial z}{\partial z} \coloneqq 1$

(3) $\frac{\partial z}{\partial y} \coloneqq \frac{\partial z}{\partial z} \cdot \frac{\partial z}{\partial y} = 2 \cdot y$

(2) $\frac{\partial z}{\partial x_1} \coloneqq \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = \frac{\partial z}{\partial y}$

$\frac{\partial z}{\partial x_2'} \coloneqq \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_2'} = \frac{\partial z}{\partial y}$

(1) $\frac{\partial z}{\partial x_2} \coloneqq \frac{\partial z}{\partial x_2'} \cdot \frac{\partial x_2'}{\partial x_2} = \frac{\partial z}{\partial x_2'} \cdot 3$



Python kód funkce $z = (x_1 + 3 \cdot x_2)^2$ se zpětnou propagací

dopředný průchod

(1) $x_2' \coloneqq 3 \cdot x_2$

(2) $y \coloneqq x_1 + x_2'$

(3) $z \coloneqq y^2$

zpětný průchod

(4) $\frac{\partial z}{\partial z} \coloneqq 1$

(3) $\frac{\partial z}{\partial y} \coloneqq \frac{\partial z}{\partial z} \cdot \frac{\partial z}{\partial y} = 2 \cdot y$

(2) $\frac{\partial z}{\partial x_1} \coloneqq \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = \frac{\partial z}{\partial y}$

$\frac{\partial z}{\partial x_2'} \coloneqq \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_2'} = \frac{\partial z}{\partial y}$

(1) $\frac{\partial z}{\partial x_2} \coloneqq \frac{\partial z}{\partial x_2'} \cdot \frac{\partial x_2'}{\partial x_2} = \frac{\partial z}{\partial x_2'} \cdot 3$

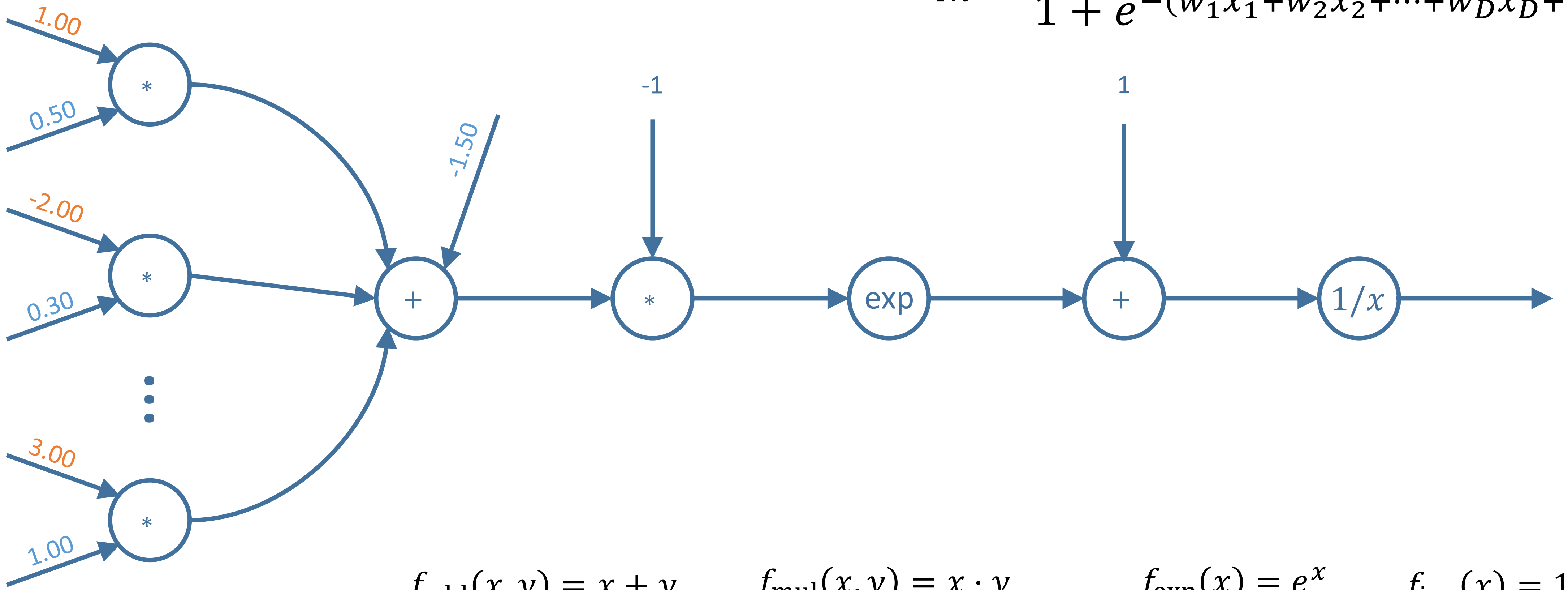
```python
def foo_with_backprop(x1, x2):
    # forward
    x2_ = 3. * x2  # (1)
    y = x1 + x2_   # (2)
    z = y ** 2.    # (3)

    # backward
    dz = 1.             # (4)
    dy = dz * 2. * y    # (3)
    dx1 = dy            # (2)
    dx2_ = dy           # (2)
    dx2 = dx2_ * 3.     # (1)

    return z, (dx1, dx2)
```

```
>>> foo_with_backprop(1., 2.)
(49.0, (14.0, 42.0))
```

Binární logistická regrese* se sigmoidem jako graf

$\boldsymbol{x}_n = [1.00, -2.00, \dots, 3.00]^T$

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



$f_{\text{add}}(x, y) = x + y$

$f_{\text{mul}}(x, y) = x \cdot y$

$f_{\exp}(x) = e^x$

$f_{\text{inv}}(x) = 1/x$

$\boldsymbol{w} = [0.50, 0.30, \dots, 1.00]^T$

$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1$

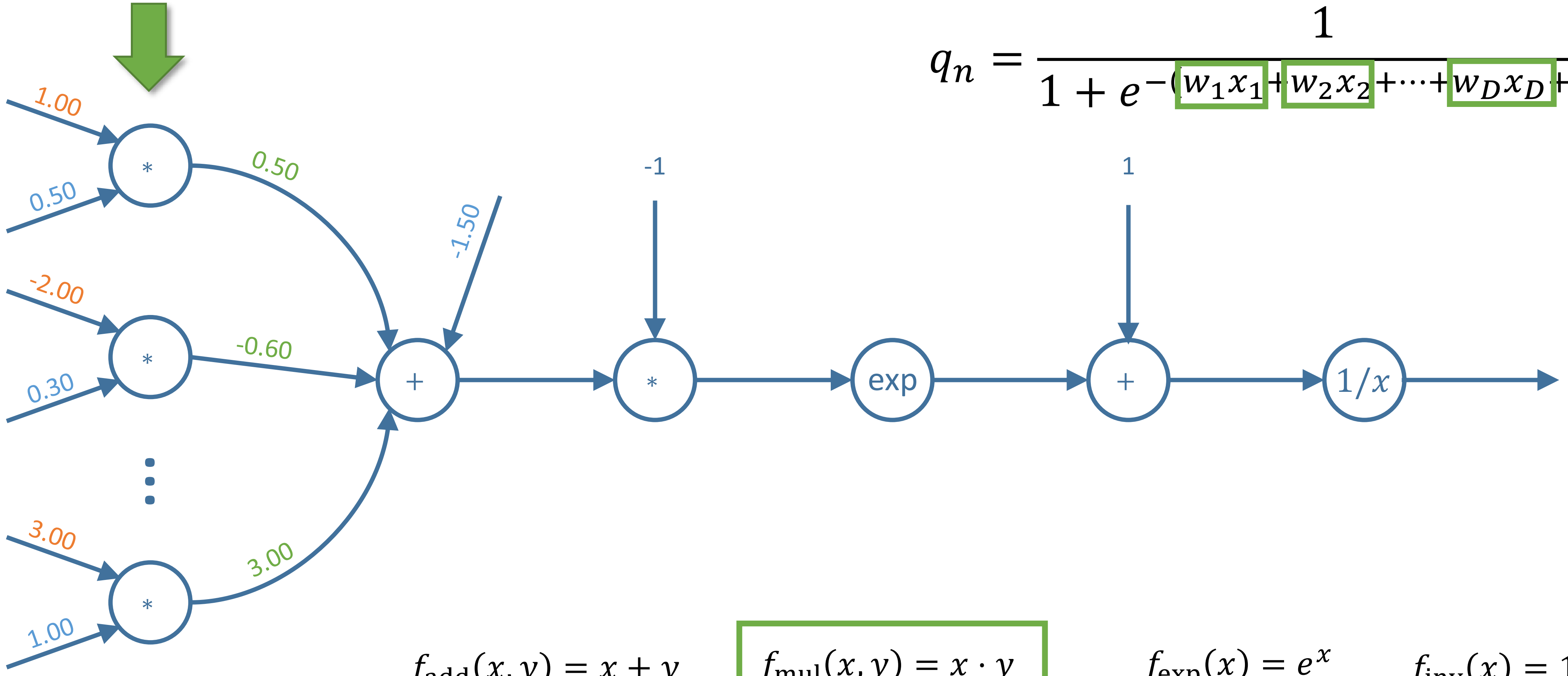
$\frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x$

$\frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x$

$\frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(\boxed{w_1 x_1} + \boxed{w_2 x_2} + \cdots + \boxed{w_D x_D} + b)}}$$



$f_{\mathrm{add}}(x, y) = x + y$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1$$

$\boxed{f_{\mathrm{mul}}(x, y) = x \cdot y}$

$$\frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x$$

$f_{\exp}(x) = e^x$

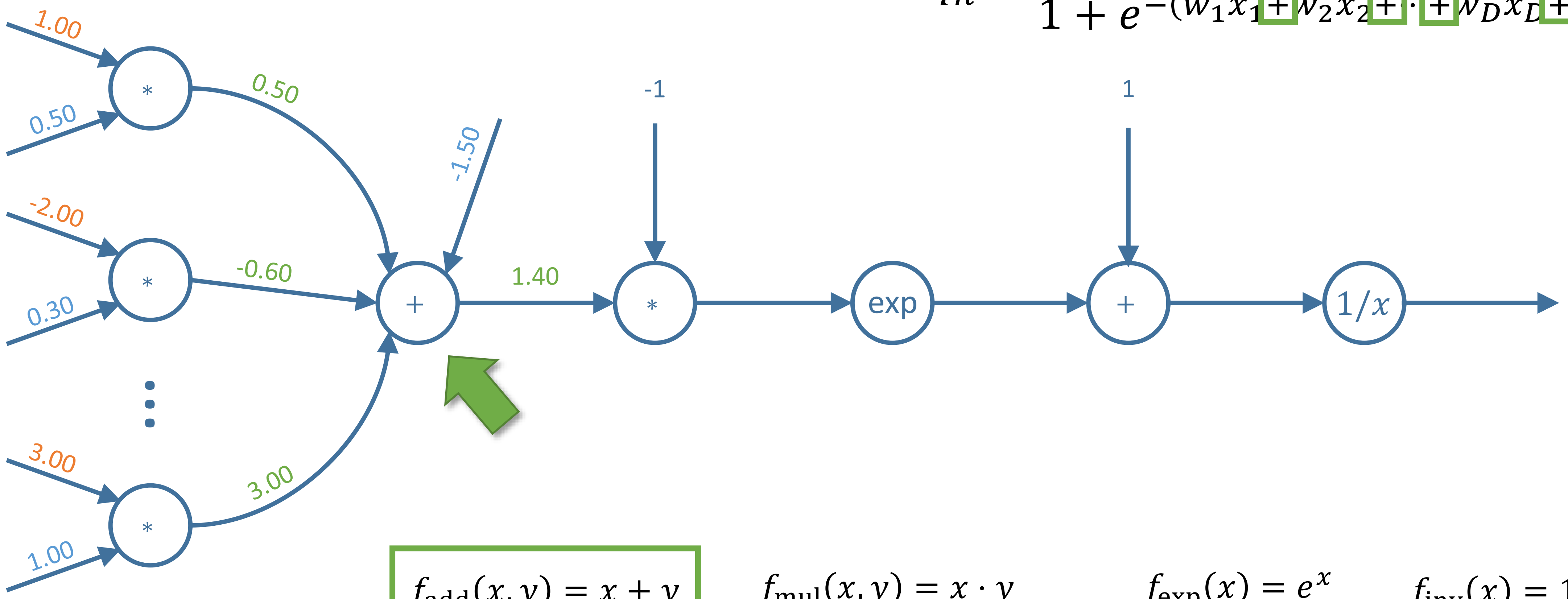
$$\frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x$$

$f_{\mathrm{inv}}(x) = 1/x$

$$\frac{\mathrm{d} f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 \boxed{+} w_2 x_2 \boxed{+} \cdots \boxed{+} w_D x_D \boxed{+} b)}}$$

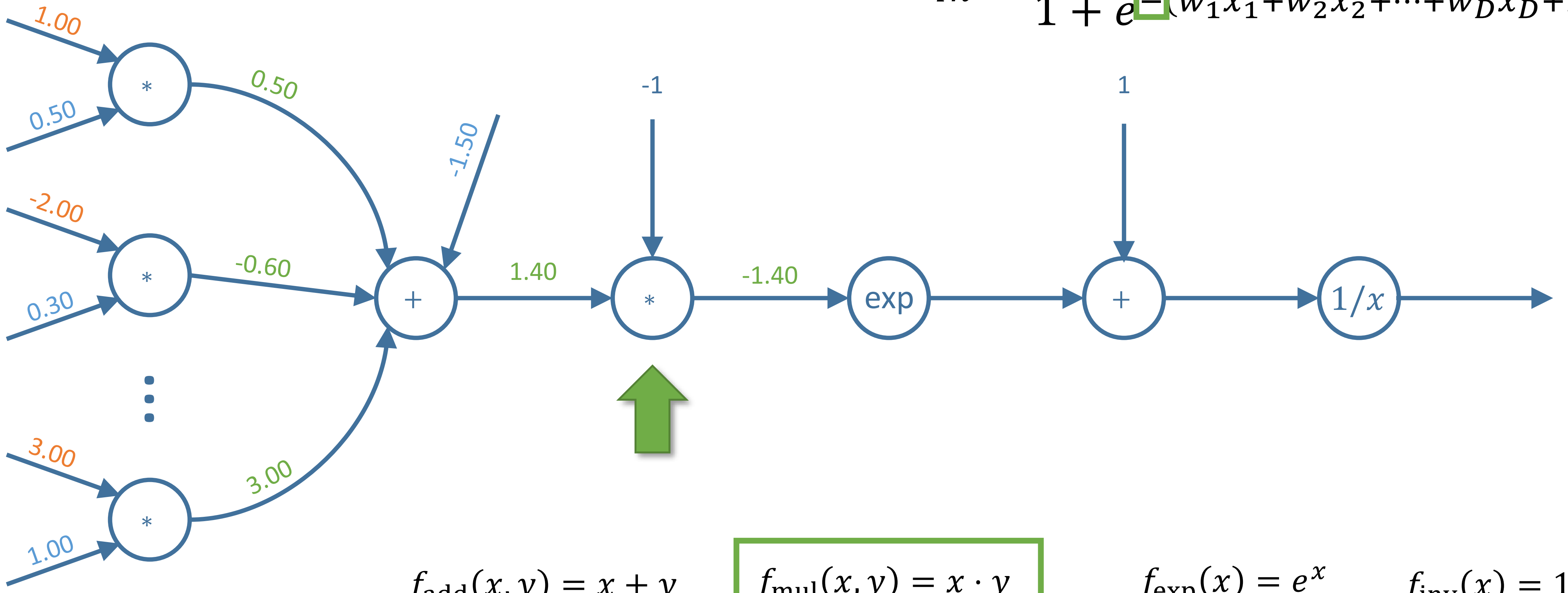


$$\boxed{f_{\mathrm{add}}(x, y) = x + y} \qquad f_{\mathrm{mul}}(x, y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\mathrm{inv}}(x) = 1/x$$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{\boxed{-}(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

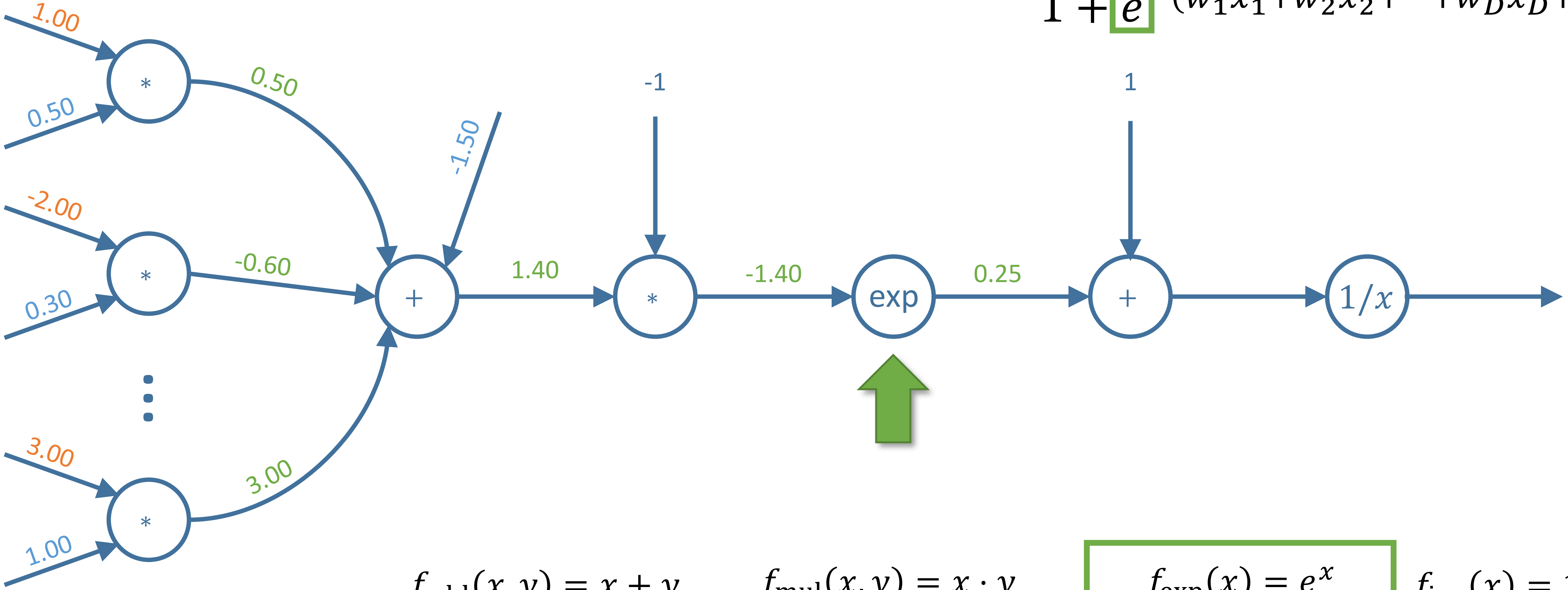


$f_{\text{add}}(x, y) = x + y$ $\quad$ $\boxed{f_{\text{mul}}(x, y) = x \cdot y}$ $\quad$ $f_{\exp}(x) = e^x$ $\quad$ $f_{\text{inv}}(x) = 1/x$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + \boxed{e}^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

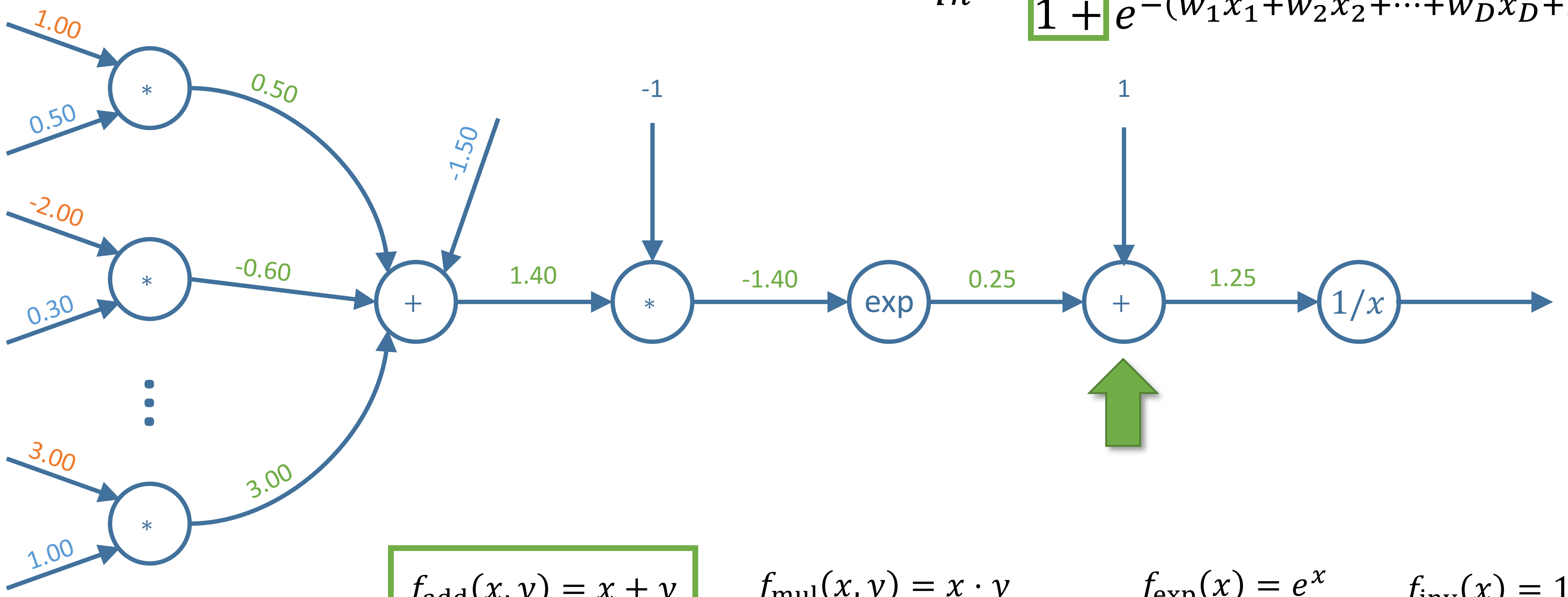


$$f_{\text{add}}(x, y) = x + y \qquad f_{\text{mul}}(x, y) = x \cdot y \qquad \boxed{f_{\exp}(x) = e^x} \qquad f_{\text{inv}}(x) = 1/x$$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{\boxed{1+} e^{-(w_1x_1+w_2x_2+\cdots+w_Dx_D+b)}}$$

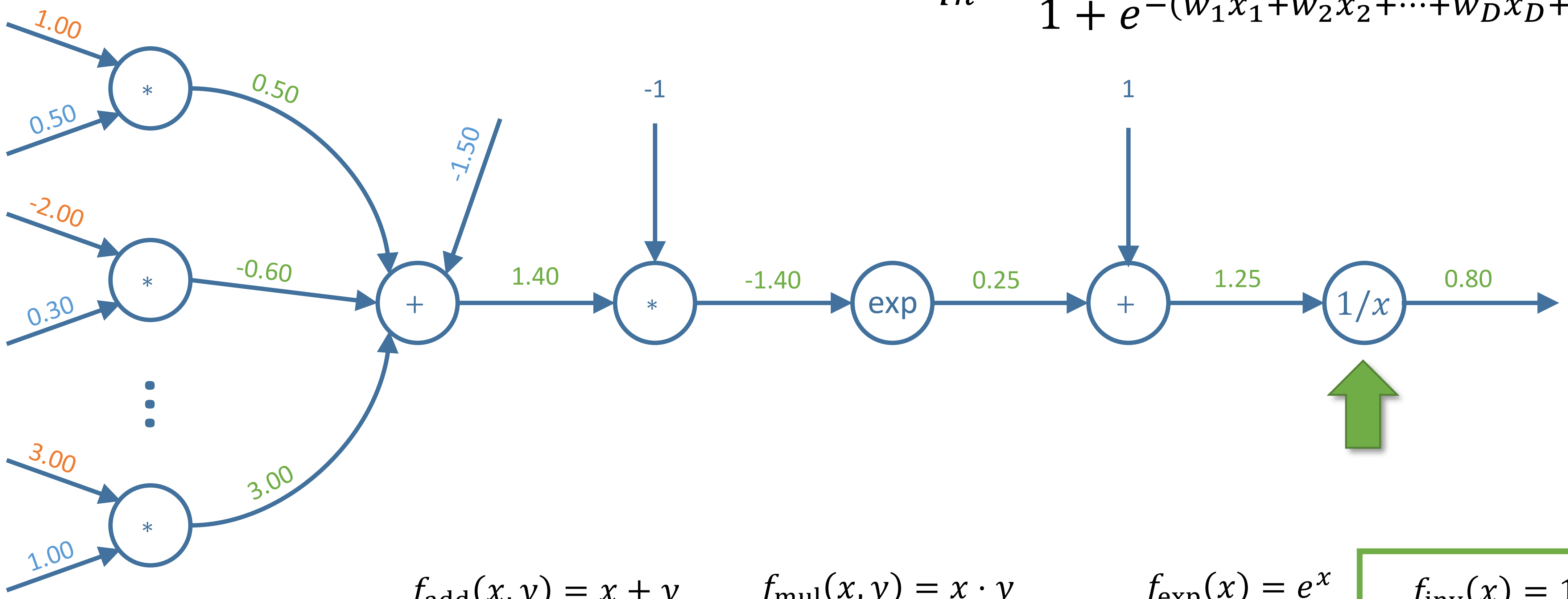


$$\boxed{f_{\text{add}}(x,y) = x + y} \qquad f_{\text{mul}}(x,y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\text{inv}}(x) = 1/x$$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

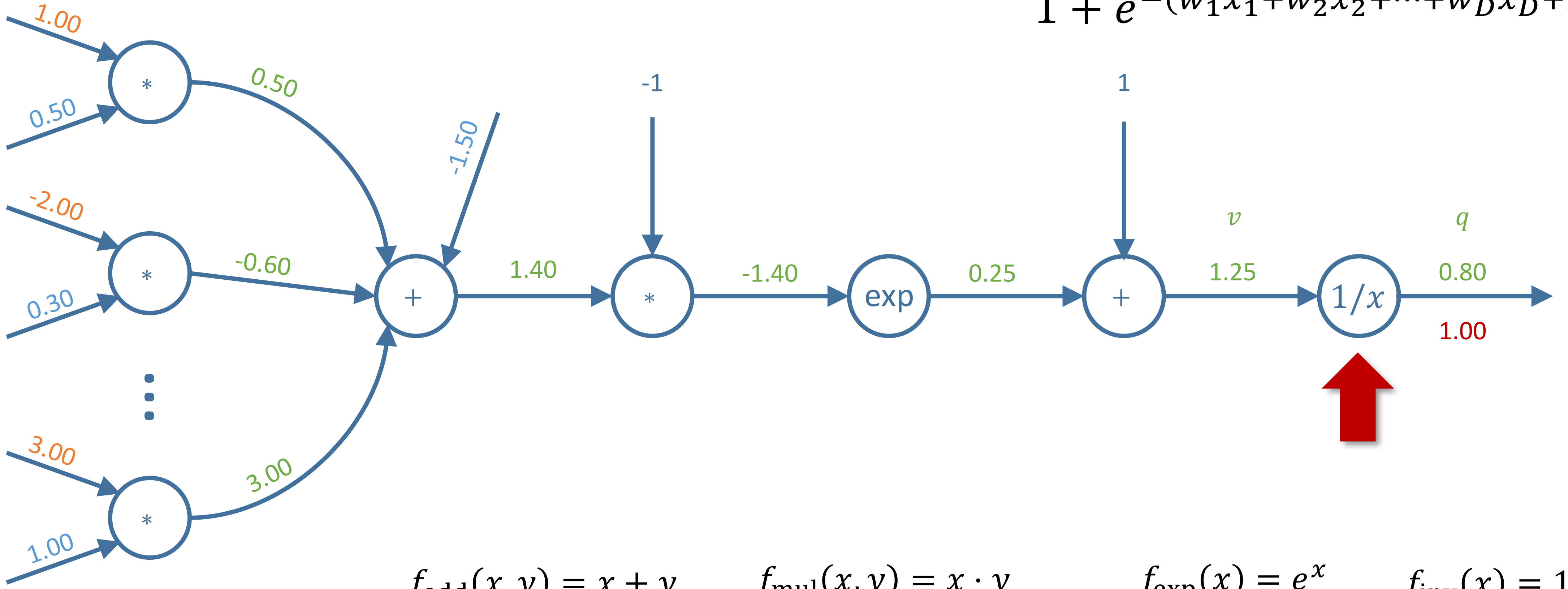


$$f_{\text{add}}(x, y) = x + y \qquad f_{\text{mul}}(x, y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad \boxed{f_{\text{inv}}(x) = 1/x}$$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

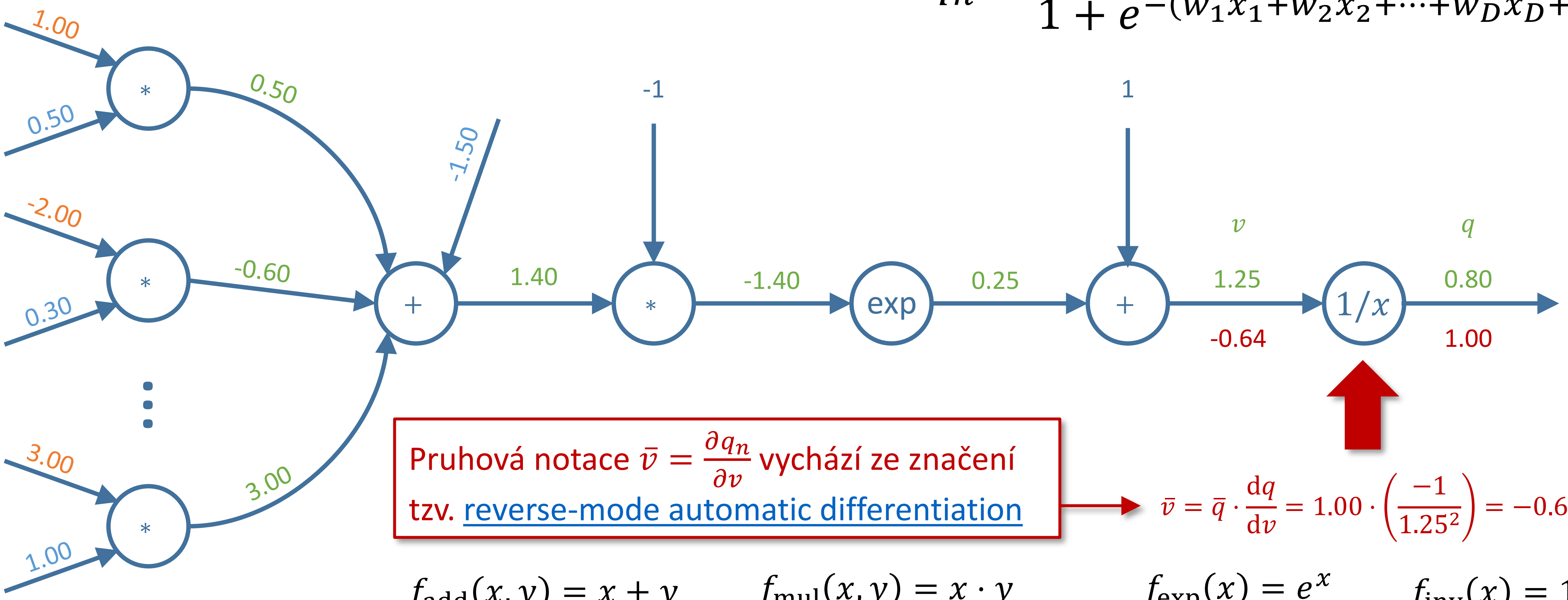


$$f_{\text{add}}(x, y) = x + y \qquad f_{\text{mul}}(x, y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\text{inv}}(x) = 1/x$$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



Pruhová notace $\bar{v} = \frac{\partial q_n}{\partial v}$ vychází ze značení tzv. reverse-mode automatic differentiation

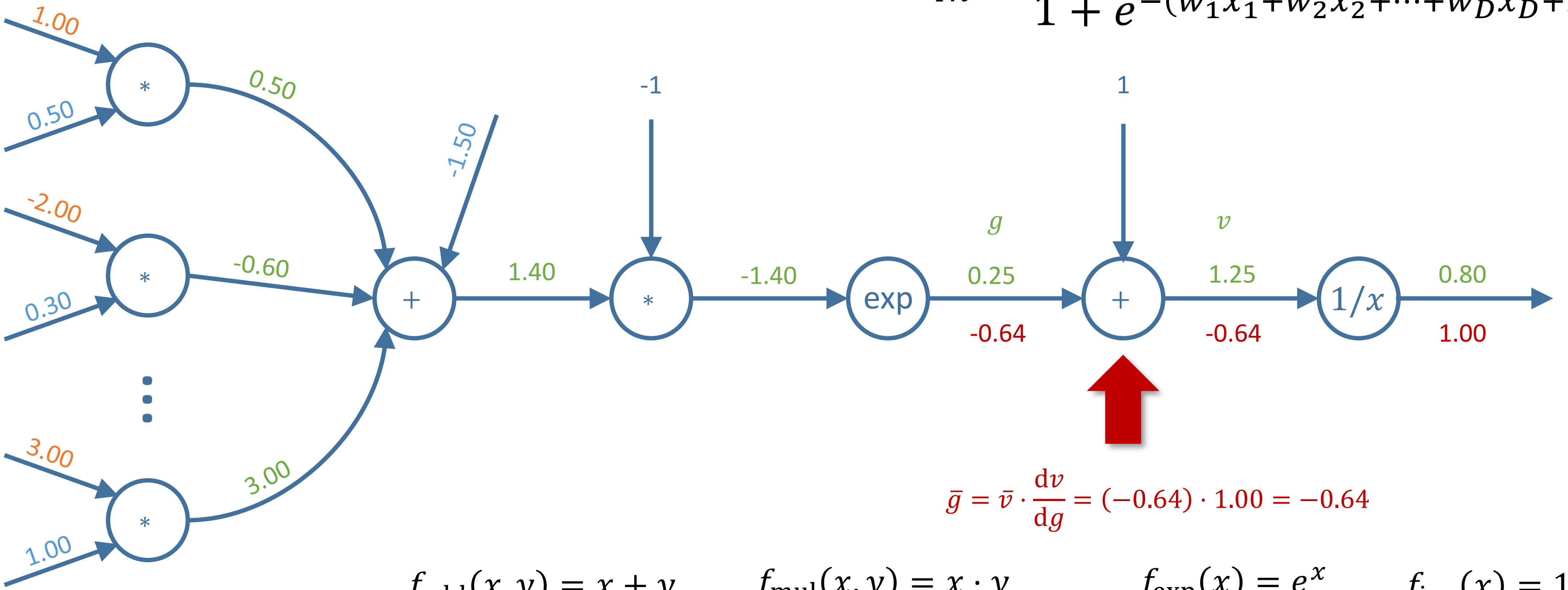
$$\bar{v} = \bar{q} \cdot \frac{\mathrm{d}q}{\mathrm{d}v} = 1.00 \cdot \left(\frac{-1}{1.25^2}\right) = -0.64$$

$f_{\mathrm{add}}(x, y) = x + y$ $\quad f_{\mathrm{mul}}(x, y) = x \cdot y$ $\quad f_{\exp}(x) = e^x$ $\quad f_{\mathrm{inv}}(x) = 1/x$

$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1$ $\quad \frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x$ $\quad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x$ $\quad \boxed{\frac{\mathrm{d} f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2}$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



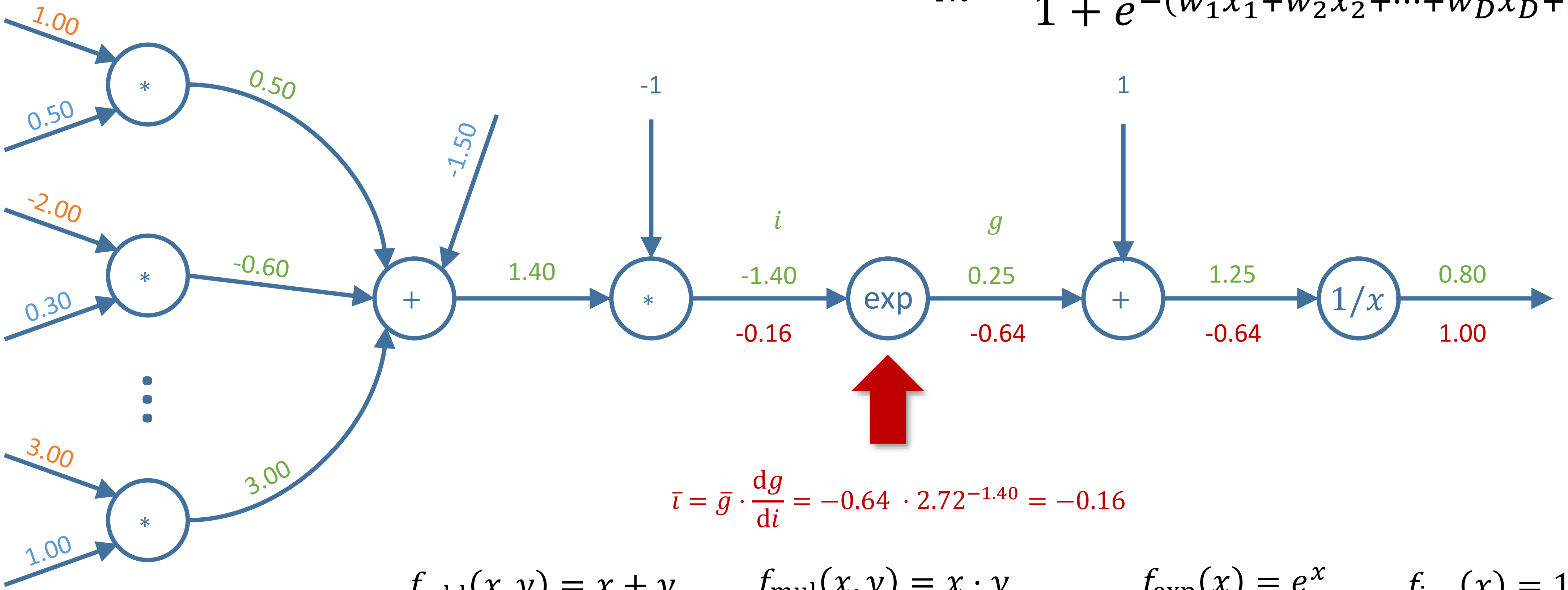
$$\bar{g} = \bar{v} \cdot \frac{\mathrm{d}v}{\mathrm{d}g} = (-0.64) \cdot 1.00 = -0.64$$

$$f_{\text{add}}(x, y) = x + y \qquad f_{\text{mul}}(x, y) = x \cdot y \qquad f_{\text{exp}}(x) = e^x \qquad f_{\text{inv}}(x) = 1/x$$

$$\boxed{\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1} \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\text{exp}}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



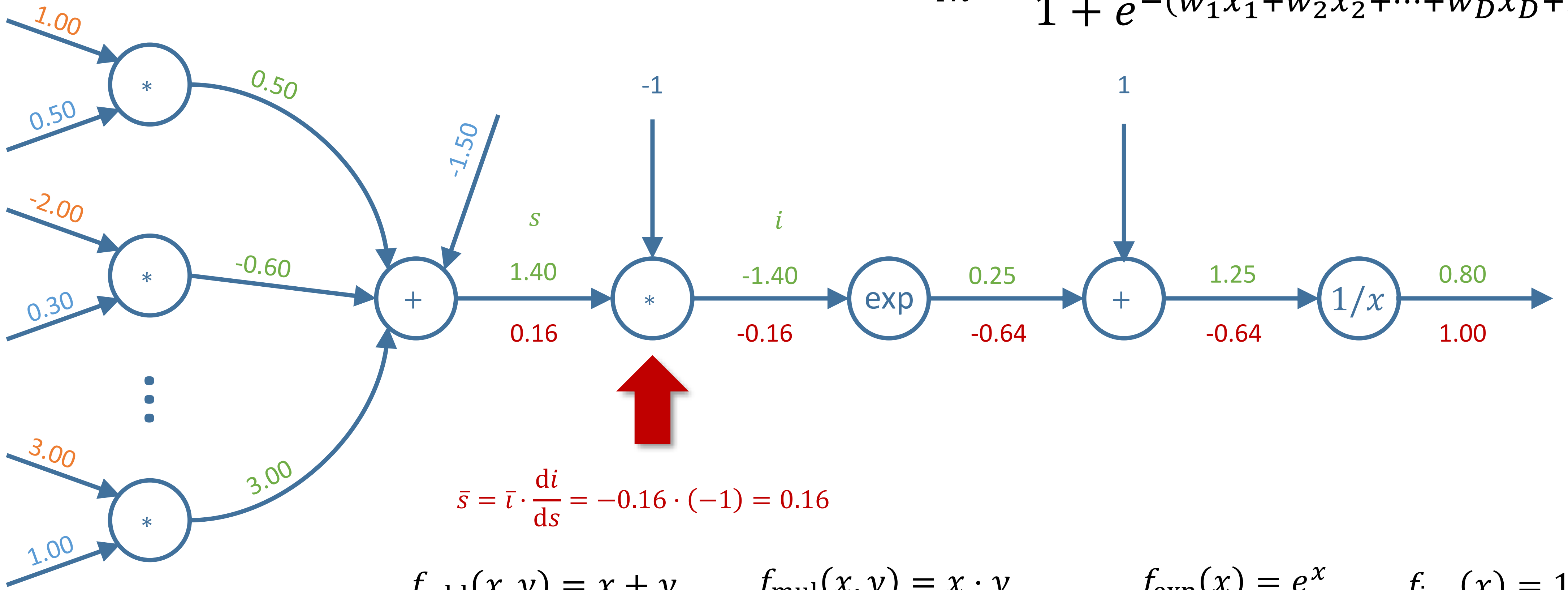
$$\bar{\imath} = \bar{g} \cdot \frac{\mathrm{d}g}{\mathrm{d}i} = -0.64 \cdot 2.72^{-1.40} = -0.16$$

$$f_{\mathrm{add}}(x, y) = x + y \qquad f_{\mathrm{mul}}(x, y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\mathrm{inv}}(x) = 1/x$$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1 \qquad \frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x \qquad \boxed{\frac{\mathrm{d}f_{\exp}}{\mathrm{d}x} = e^x} \qquad \frac{\mathrm{d}f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + \cdots + w_Dx_D + b)}}$$



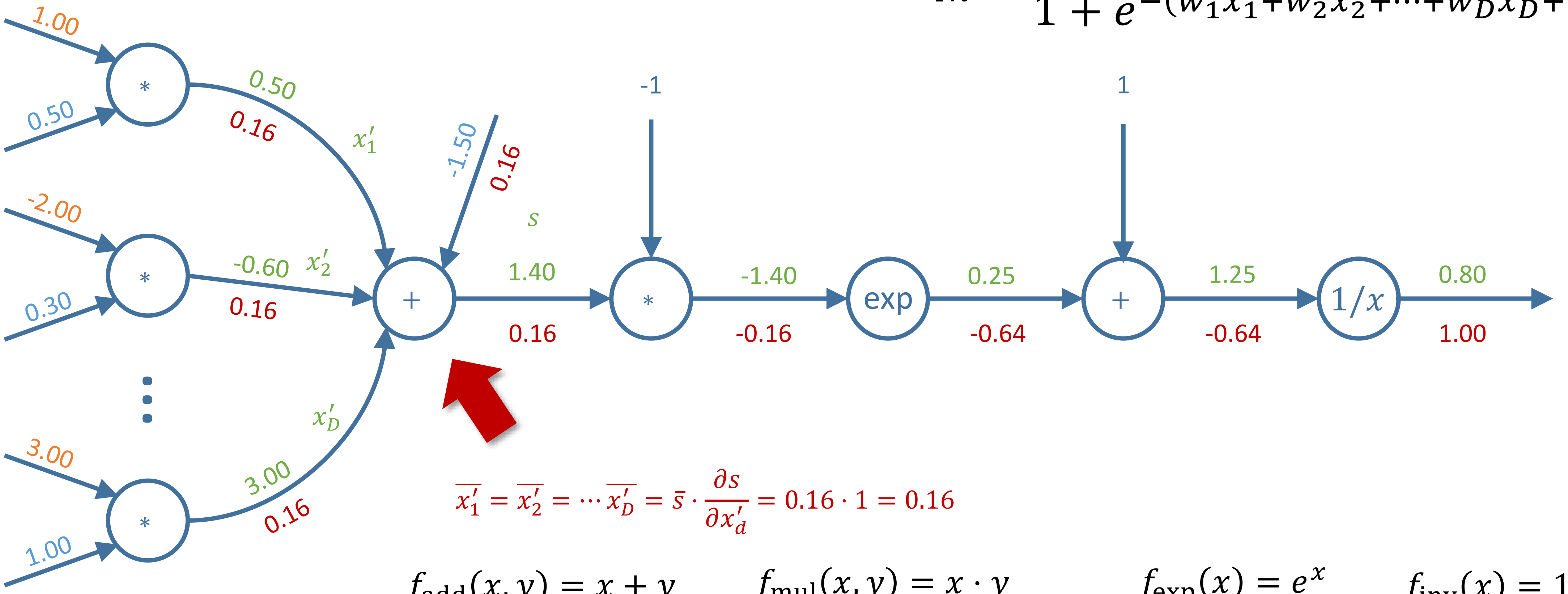
$$\bar{s} = \bar{\imath} \cdot \frac{\mathrm{d}i}{\mathrm{d}s} = -0.16 \cdot (-1) = 0.16$$

$f_{\mathrm{add}}(x, y) = x + y$ $\quad f_{\mathrm{mul}}(x, y) = x \cdot y$ $\quad f_{\exp}(x) = e^x$ $\quad f_{\mathrm{inv}}(x) = 1/x$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1 \qquad \boxed{\frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x} \qquad \frac{\mathrm{d}f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d}f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



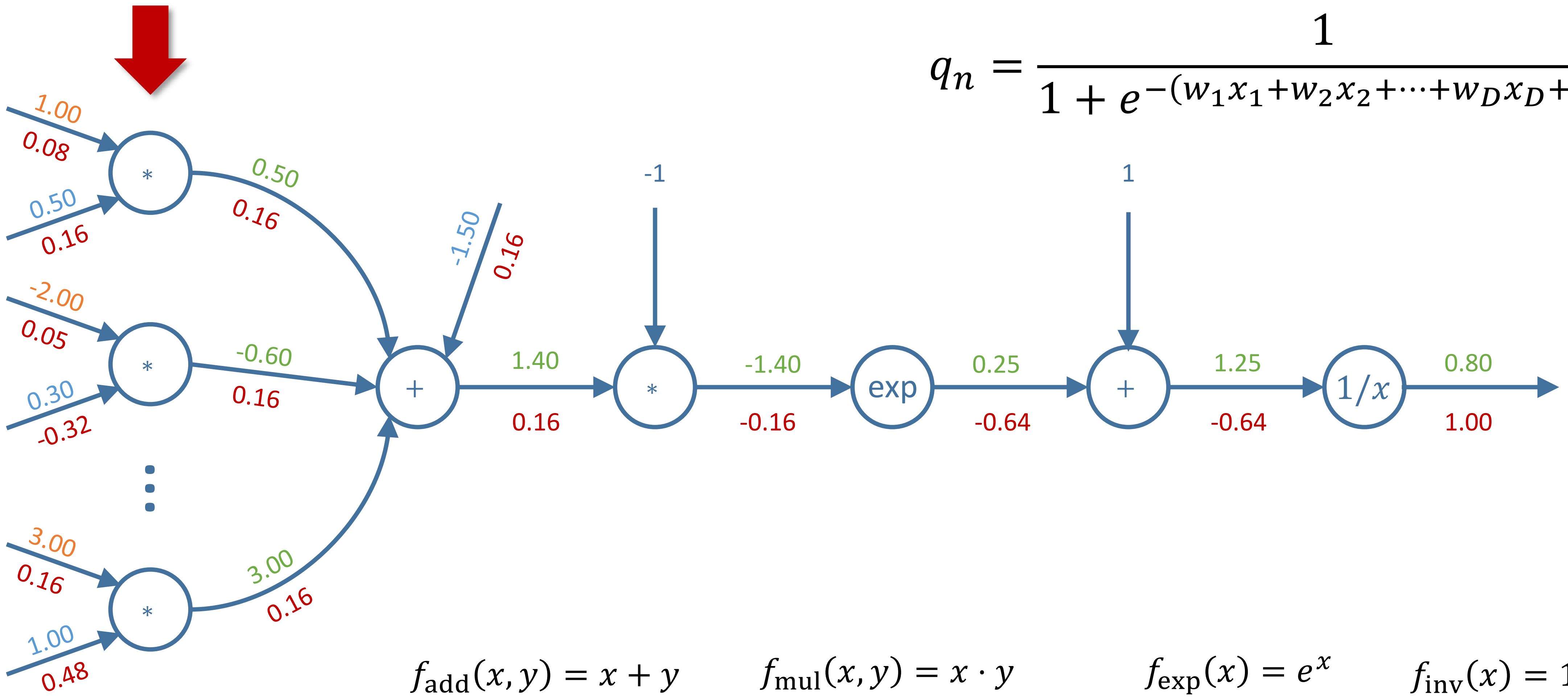
$$\overline{x_1'} = \overline{x_2'} = \cdots \overline{x_D'} = \overline{s} \cdot \frac{\partial s}{\partial x_d'} = 0.16 \cdot 1 = 0.16$$

$$f_{\text{add}}(x, y) = x + y \qquad f_{\text{mul}}(x, y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\text{inv}}(x) = 1/x$$

$$\boxed{\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1} \qquad \frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\text{inv}}}{\mathrm{d}x} = -1/x^2$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



$$f_{\mathrm{add}}(x,y) = x + y \qquad f_{\mathrm{mul}}(x,y) = x \cdot y \qquad f_{\exp}(x) = e^x \qquad f_{\mathrm{inv}}(x) = 1/x$$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1 \qquad \boxed{\frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x} \qquad \frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x \qquad \frac{\mathrm{d} f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

Zpětná propagace pro binární logistickou regresi* v Pythonu

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

```python
def linear_sigmoid_with_backprop(x, w, b):
    # forward
    x_ = [xi * wi for (xi, wi) in zip(x, w)]  # (1) mul1
    l = sum(x_)                               # (2) add1
    s = l + b                                 # (3) add2
    i = -1. * s                               # (4) mul2
    g = 2.718 ** i                            # (5) exp1
    v = g + 1.                                # (6) add3
    q = 1. / v                                # (7) inv1

    # backward
    dq = 1
    dv = dq * -1. / v ** 2                          # (7) inv1
    dg = dv                                         # (6) add3
    di = dg * g                                     # (5) exp1
    ds = -di                                        # (4) mul2
    db = ds                                         # (3) add2
    dx_ = [ds] * len(x_)                            # (2) add1
    dw = [dx_i * xi for (dx_i, xi) in zip(dx_, x)]  # (1) mul1
    dx = [dx_i * wi for (dx_i, wi) in zip(dx_, w)]  # (1) mul1

    return q, (dx, dw, db)
```

$$f_{\mathrm{add}}(x, y) = x + y$$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1$$

$$f_{\mathrm{mul}}(x, y) = x \cdot y$$

$$\frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x$$

$$f_{\exp}(x) = e^x$$

$$\frac{\mathrm{d} f_{\exp}}{\mathrm{d}x} = e^x$$

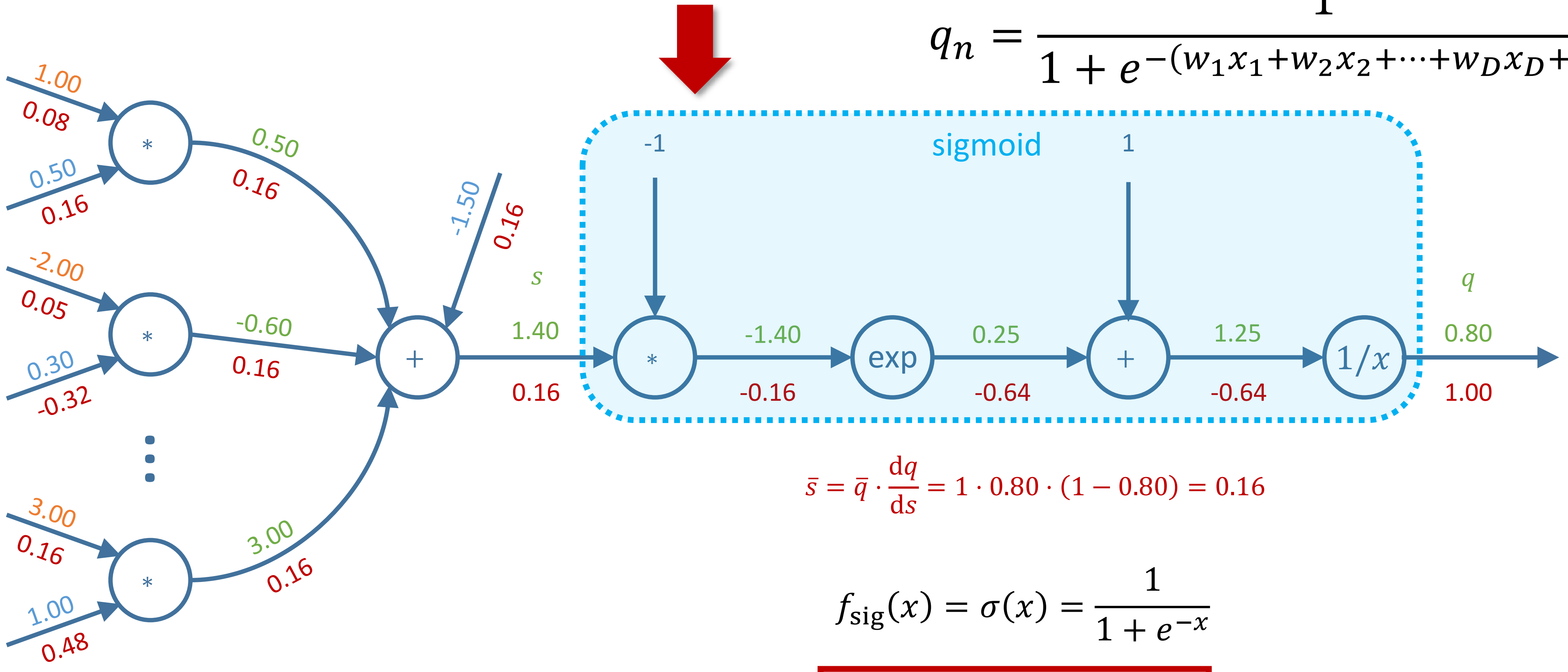
$$f_{\mathrm{inv}}(x) = 1/x$$

$$\frac{\mathrm{d} f_{\mathrm{inv}}}{\mathrm{d}x} = -1/x^2$$

```python
>>> linear_sigmoid_with_backprop(
>>>     [1., -2., 3.],
>>>     [0.5, 0.3, 1.],
>>>     -1.5
>>> )
(0.802,
 ([0.0793, 0.0476, 0.1586988],
  [0.1586988, -0.317, 0.476],
  0.15869881881212394)
)
```

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



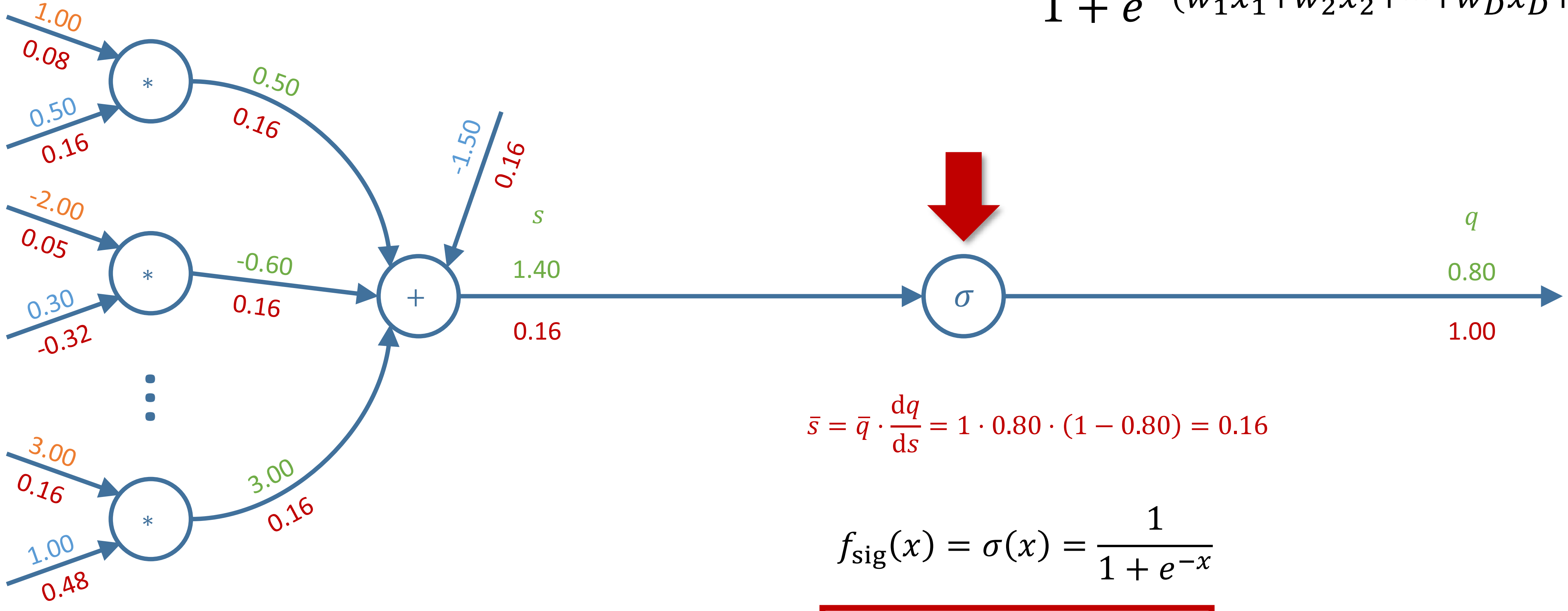
$$\bar{s} = \bar{q} \cdot \frac{\mathrm{d}q}{\mathrm{d}s} = 1 \cdot 0.80 \cdot (1 - 0.80) = 0.16$$

$$f_{\text{sig}}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\boxed{\frac{\mathrm{d}f_{\text{sig}}}{\mathrm{d}x} = \sigma(x)\big(1 - \sigma(x)\big)}$$

Binární logistická regrese* se sigmoidem jako graf

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$



$$\bar{s} = \bar{q} \cdot \frac{\mathrm{d}q}{\mathrm{d}s} = 1 \cdot 0.80 \cdot (1 - 0.80) = 0.16$$

$$f_{\mathrm{sig}}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\boxed{\frac{\mathrm{d}f_{\mathrm{sig}}}{\mathrm{d}x} = \sigma(x)\big(1 - \sigma(x)\big)}$$

Zpětná propagace pro binární logistickou regresi* v Pythonu

$$q_n = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b)}}$$

```
def linear_sigmoid_with_backprop(x, w, b):
    # forward
    x_ = [xi * wi for (xi, wi) in zip(x, w)]  # (1) mul1
    l = sum(x_)                               # (2) add1
    s = l + b                                 # (3) add2
    q = 1. / (1. + 2.718 ** (-s))             # (4) sig1

    # backward
    dq = 1
    ds = dq * q * (1. - q)                          # (4) sig1
    db = ds                                         # (3) add2
    dx_ = [ds] * len(x_)                            # (2) add1
    dw = [dx_i * xi for (dx_i, xi) in zip(dx_, x)]  # (1) mul1
    dx = [dx_i * wi for (dx_i, wi) in zip(dx_, w)]  # (1) mul1

    return q, (dx, dw, db)
```

$$f_{\mathrm{add}}(x, y) = x + y$$

$$\frac{\partial f_{\mathrm{add}}}{\partial x} = \frac{\partial f_{\mathrm{add}}}{\partial y} = 1$$

$$f_{\mathrm{mul}}(x, y) = x \cdot y$$

$$\frac{\partial f_{\mathrm{mul}}}{\partial x} = y, \frac{\partial f_{\mathrm{mul}}}{\partial y} = x$$

$$f_{\mathrm{sig}}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{\mathrm{d} f_{\mathrm{sig}}}{\mathrm{d}x} = \sigma(x)\big(1 - \sigma(x)\big)$$

```
>>> linear_sigmoid_with_backprop(
>>>     [1., -2., 3.],
>>>     [0.5, 0.3, 1.],
>>>     -1.5
>>> )
(0.802,
 ([0.0793, 0.0476, 0.1586988],
  [0.1586988, -0.317, 0.476],
  0.15869881881212394)
)
```

Zpětná propagace *skoro* obecně (zrhuba a ošklivě)

- Pokud máme složenou funkci

$$y_N = f_N(f_{\dots}(f_2(f_1(\boldsymbol{x}), \boldsymbol{x}), \boldsymbol{x}), \boldsymbol{x})$$

kde

- $\boldsymbol{x} = [x_1, \dots, x_D]^\top$
- $y_N \in \mathbb{R}$

Uvažujeme M vstupů (vektor) a 1 výstup (skalár)

- Funkci f můžeme rozvinout jako

$$\boldsymbol{y}_1 \coloneqq f_1(\boldsymbol{x})$$
$$\boldsymbol{y}_2 \coloneqq f_2(\boldsymbol{y}_1, \boldsymbol{x})$$
$$\dots$$
$$y_N \coloneqq f_N(\boldsymbol{y}_{N-1}, \boldsymbol{x})$$

- Chceme spočítat gradient

$$\overline{\boldsymbol{x}} = [\overline{x_1}, \dots, \overline{x_D}]^\top = \nabla y_N = \left[\frac{\partial y_N}{\partial x_1}, \dots, \frac{\partial y_N}{\partial x_D}\right]^\top$$

- Zpětná propagace pro výpočet $\overline{\boldsymbol{x}}$ spočívá v aplikaci řetízkového pravidla jednotlivých dílčích funkcí f_n v pořadí $f_N, f_{N-1}, \dots, f_1$, tj.

$$\overline{y_{N-1}} \coloneqq \frac{\partial f_N(\boldsymbol{y}_{N-1}, \boldsymbol{x})}{\partial \boldsymbol{y}_{N-1}}$$

$$\overline{y_{N-2}} \coloneqq \overline{y_{N-1}} \cdot \frac{\partial f_{N-1}(\boldsymbol{y}_{N-2}, \boldsymbol{x})}{\partial \boldsymbol{y}_{N-2}}$$

$$\dots$$

$$\overline{y_1} \coloneqq \overline{y_2} \cdot \frac{\partial f_2(\boldsymbol{y}_1, \boldsymbol{x})}{\partial \boldsymbol{y}_1}$$

$$\overline{\boldsymbol{x}} \coloneqq \overline{y_1} \cdot \frac{\partial f_1(\boldsymbol{x})}{\partial \boldsymbol{x}} + \sum_{n=2}^{N} \overline{y_n} \cdot \frac{\partial f_n(\boldsymbol{y}_{n-1}, \boldsymbol{x})}{\partial \boldsymbol{x}}$$

Zpětná propagace *skoro* obecně (zrhuba a ošklivě)

- Pokud máme složenou funkci

$$y_N = f_N(f_{\dots}(f_2(f_1(\boldsymbol{x}), \boldsymbol{x}), \boldsymbol{x}), \boldsymbol{x})$$

kde

  - $\boldsymbol{x} = [x_1, \dots, x_D]^\top$
  - $y_N \in \mathbb{R}$

- Funkci f můžeme rozvinout jako

$$\begin{aligned} \boldsymbol{y}_1 &:= f_1(\boldsymbol{x}) \\ \boldsymbol{y}_2 &:= f_2(\boldsymbol{y}_1, \boldsymbol{x}) \\ &\dots \\ y_N &:= f_N(\boldsymbol{y}_{N-1}, \boldsymbol{x}) \end{aligned}$$

Každá f_n může být obecně závislá na jakémkoliv vstupu (ale i mezivýstupu, což ve vztazích není zohledněno)

- Chceme spočítat gradient

$$\overline{\boldsymbol{x}} = [\overline{x_1}, \dots, \overline{x_D}]^\top = \nabla y_N = \left[\frac{\partial y_N}{\partial x_1}, \dots, \frac{\partial y_N}{\partial x_D}\right]^\top$$

- Zpětná propagace pro výpočet $\overline{\boldsymbol{x}}$ spočívá v aplikaci řetízkového pravidla jednotlivých dílčích funkcí f_n v pořadí $f_N, f_{N-1}, \dots, f_1$, tj.

$$\overline{\boldsymbol{y}_{N-1}} := \frac{\partial f_N(\boldsymbol{y}_{N-1}, \boldsymbol{x})}{\partial \boldsymbol{y}_{N-1}}$$

$$\overline{\boldsymbol{y}_{N-2}} := \overline{\boldsymbol{y}_{N-1}} \cdot \frac{\partial f_{N-1}(\boldsymbol{y}_{N-2}, \boldsymbol{x})}{\partial \boldsymbol{y}_{N-2}}$$

$$\dots$$

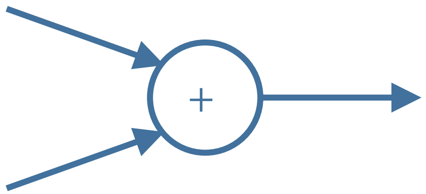
$$\overline{\boldsymbol{y}_2} \cdot \frac{\partial f_2(\boldsymbol{y}_1, \boldsymbol{x})}{\partial \boldsymbol{y}_1}$$

$$\overline{\boldsymbol{x}} := \overline{\boldsymbol{y}_1} \cdot \frac{\partial f_1(\boldsymbol{x})}{\partial \boldsymbol{x}} + \sum_{n=2}^{N} \overline{\boldsymbol{y}_n} \cdot \frac{\partial f_n(\boldsymbol{y}_{n-1}, \boldsymbol{x})}{\partial \boldsymbol{x}}$$

Modularizace zpětné propagace

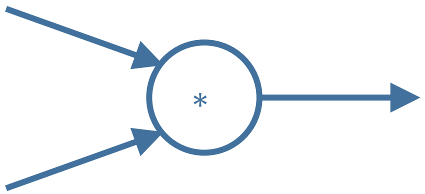
Druhy operací

- Např. ve funkci linear_sigmoid se v grafu opakují pouze 4 typy operací (**"vrstev"**) i jejich řetízkových pravidel:
  1. součet
  2. násobení
  3. exponenciální funkce
  4. inverze



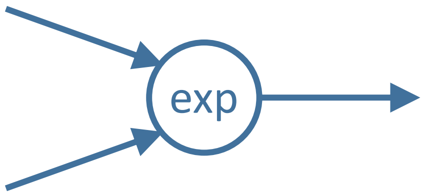
$$f_{\text{add}}(x, y) = x + y$$

$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1$$



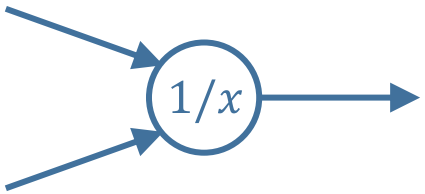
$$f_{\text{mul}}(x, y) = x \cdot y$$

$$\frac{\partial f_{\text{mul}}}{\partial x} = y, \frac{\partial f_{\text{mul}}}{\partial y} = x$$



$$f_{\text{exp}}(x) = e^x$$

$$\frac{\partial f_{\text{exp}}}{\partial x} = e^x$$



$$f_{\text{inv}}(x) = 1/x$$

$$\frac{\partial f_{\text{inv}}}{\partial x} = -1/x^2$$

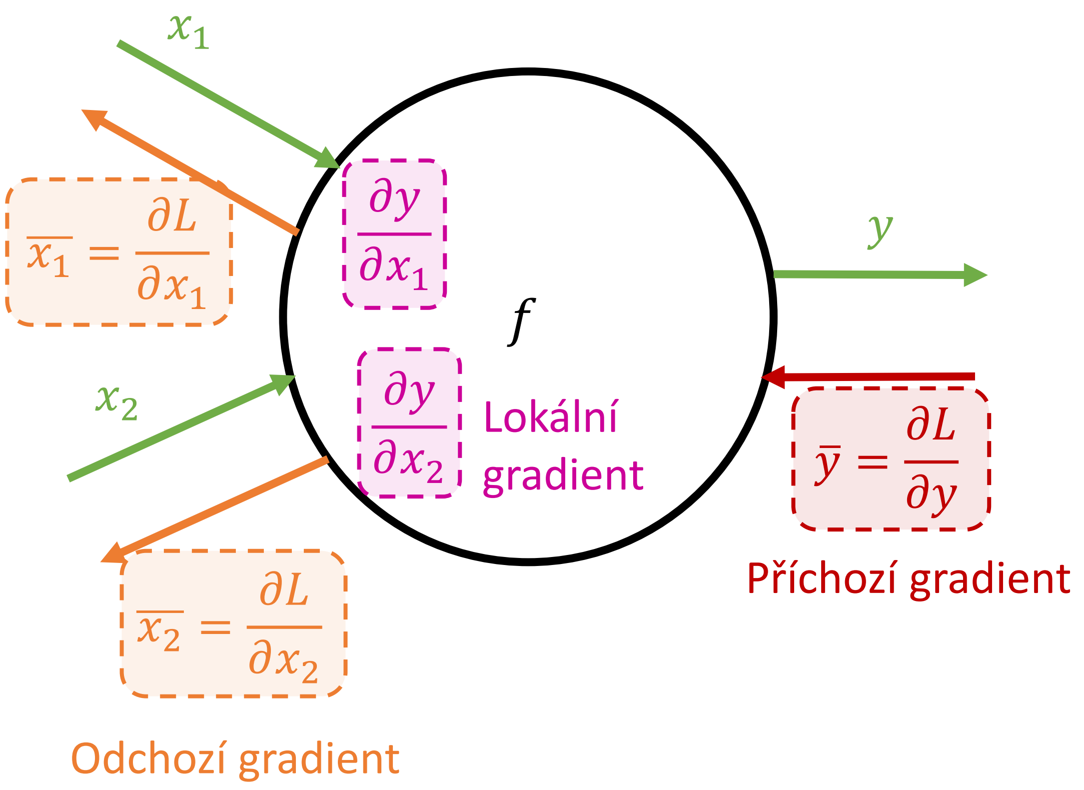
Uzel grafu a lokální gradient

- Na funkce lze nahlížet jako na orientovaný výpočetní graf
- Jednotlivé operace jsou uzly
- Hrany jsou proměnné a zároveň repezentují návaznosti vstupů a výstupů
- Dopředný průchod je

$$y = f(x_1, \dots, x_D)$$

- Zpětný průchod je aplikace řetízkového pravidla

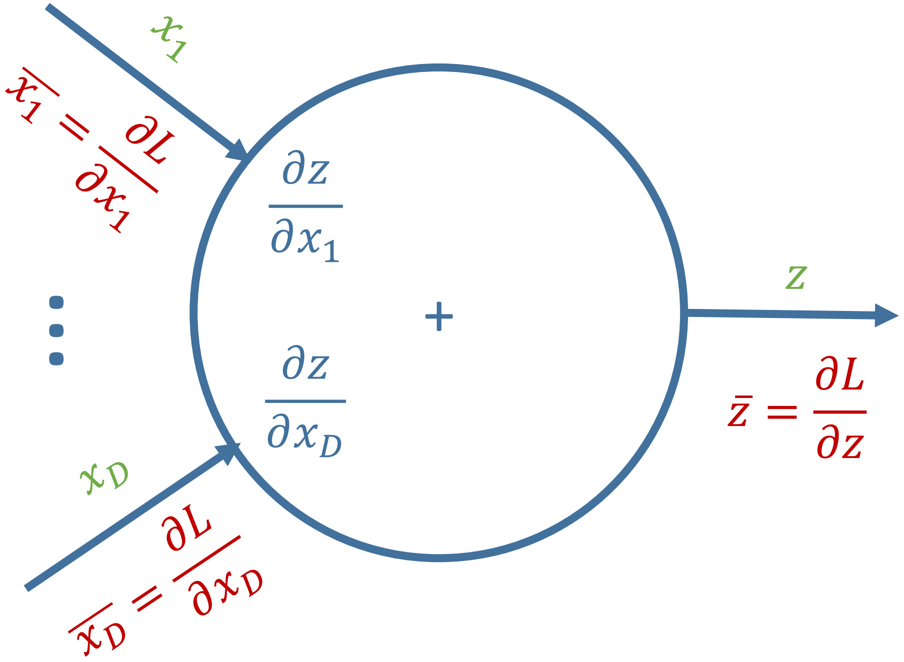
$$\boxed{\frac{\partial L}{\partial x_d}} = \boxed{\frac{\partial L}{\partial y}} \cdot \boxed{\frac{\partial y}{\partial x_d}}$$



pruhová notace: https://people.maths.ox.ac.uk/gilesm/files/NA-08-01.pdf

Příklad: Operace sčítání jako uzel ve výpočetním grafu

```
class Sum(DifferentiableFunction):

    @staticmethod
    def forward(x: np.ndarray):
        z = np.sum(x)
        cache = x.shape,
        return z, cache

    @staticmethod
    def backward(dz: float, cache: tuple):
        x_shape, = cache
        dx = dz * np.ones(x_shape)  # retizkove pravidlo
        return dx
```



forward:

$$\boxed{z = \sum_{d=1}^{D} x_d}$$

lokální gradient:

$$\frac{\partial z}{\partial x_d} = 1$$

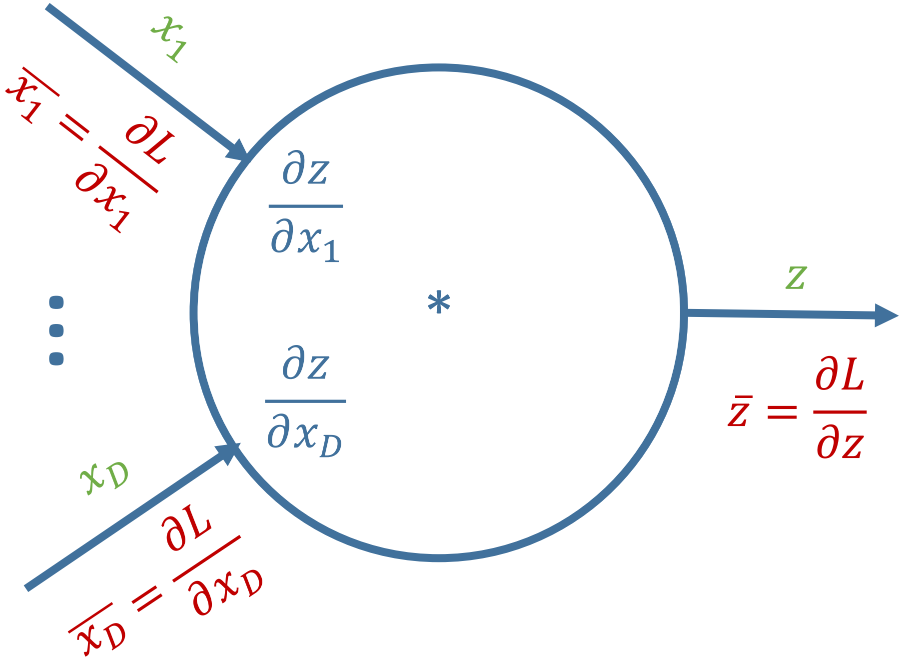
backward:

$$\boxed{\overline{\boldsymbol{x}} = [\bar{z}, \dots, \bar{z}]^\top}$$

ve zpětném režimu “rozdistribuje” příchozí gradient do všech vstupů:

Příklad: Operace násobení jako uzel ve výpočetním grafu

```python
class Multiply(DifferentiableFunction):

    @staticmethod
    def forward(x: np.ndarray):
        z = np.prod(x)
        cache = x, z
        return z, cache

    @staticmethod
    def backward(dz: float, cache: tuple):
        x, z = cache
        dx = dz * z / x  # retizkove pravidlo
        return dx
```



forward:

$$\boxed{z = \prod_{d=1}^{D} x_d}$$

lokální gradient:

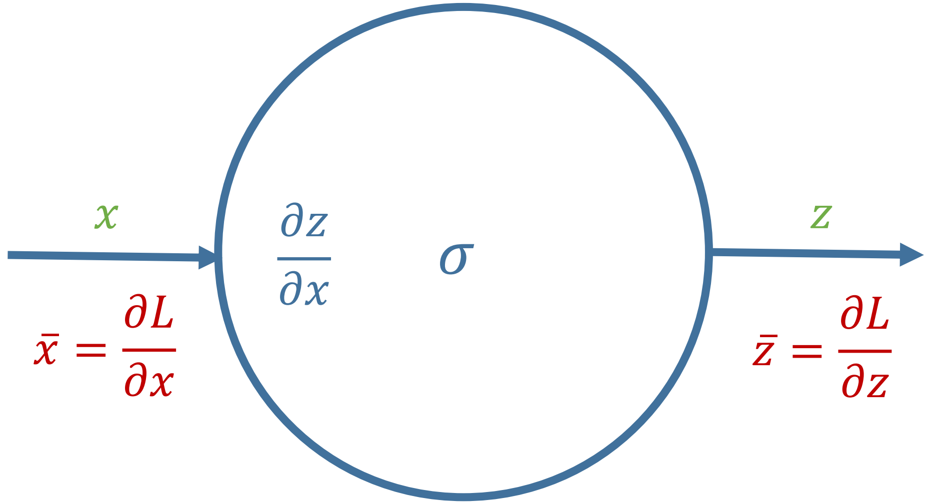
$$\frac{\partial z}{\partial x_d} = \prod_{i \neq d} x_i = \frac{z}{x_d}$$

backward:

$$\boxed{\bar{\boldsymbol{x}} = \left[\frac{\bar{z} \cdot z}{x_1}, \dots, \frac{\bar{z} \cdot z}{x_D}\right]^\top}$$

Příklad: Funkce sigmoid jako uzel ve výpočetním grafu

```
class Sigmoid(DifferentiableFunction):

    @staticmethod
    def forward(x: float):  # zatim pouze skalary
        z = 1 / (1 - math.exp(-x))
        cache = z,
        return z, cache

    @staticmethod
    def backward(dz: float, cache: tuple):
        z, = cache
        dx = dz * z * (1 - z)  # retizkove pravidlo
        return dx
```



forward:

$$z = \frac{1}{1 + e^{-x}}$$

lokální gradient:

$$\frac{\partial z}{\partial x} = z \cdot (1 - z)$$

backward:

$$\bar{\boldsymbol{x}} = \bar{z} \cdot z \cdot (1 - z)$$

Příklad: definice vlastní funkce v PyTorch

```python
class MulConstant(torch.autograd.Function):
    @staticmethod
    def forward(tensor, constant):
        return tensor * constant

    @staticmethod
    def setup_context(ctx, inputs, output):
        # ctx is a context object that can be used to stash information
        # for backward computation
        tensor, constant = inputs
        ctx.constant = constant

    @staticmethod
    def backward(ctx, grad_output):
        # We return as many input gradients as there were arguments.
        # Gradients of non-Tensor arguments to forward must be None.
        return grad_output * ctx.constant, None
```

Definujeme dopředný průchod násobení

Definujeme zpětný průchod násobení

Příklad: https://pytorch.org/docs/stable/notes/extending.html

Příklad: implementace lineární vrstvy v PyTorch pro MKLDNN

https://github.com/pytorch/pytorch/blob/5ed60477a7cd930298dbdab71046ec62024427c4/aten/src/ATen/native/mkldnn/Linear.cpp#L59

```
Tensor mkldnn_linear(
    const Tensor& self,
    const Tensor& weight_t, const c10::optional<Tensor>& bias_opt) {
  // See [Note: hacky wrapper removal for optional tensor]
  c10::MaybeOwned<Tensor> bias_maybe_owned = at::borrow_from_optional_tensor(bias_opt);
  const Tensor& bias = *bias_maybe_owned;
...
  ideep::tensor y;
  if (bias.defined()) {
    const ideep::tensor b = itensor_from_tensor(bias);
    ideep::inner_product_forward::compute(x, w, b, y);
  } else {
    ideep::inner_product_forward::compute(x, w, y);
  }
```

Definice dopředného průchodu

$$f(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{b}) = \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b}$$

Příklad: implementace lineární vrstvy v PyTorch pro MKLDNN

https://github.com/pytorch/pytorch/blob/5ed60477a7cd930298dbdab71046ec62024427c4/aten/src/ATen/native/mkldnn/Linear.cpp#L167C13-L167C13

```
std::tuple<Tensor, Tensor, Tensor> mkldnn_linear_backward(
    const Tensor& input, const Tensor& grad_output,
    const Tensor& weight, std::array<bool,3> output_mask) {
  Tensor grad_input, grad_weight, grad_bias;
  if (output_mask[0]) {
    grad_input = at::mkldnn_linear_backward_input(input.sizes(), grad_output, weight);
  }
  if (output_mask[1] || output_mask[2]) {
    std::tie(grad_weight, grad_bias) = at::mkldnn_linear_backward_weights(grad_output, input, weight, output_mask[2]);
  }
  return std::tuple<Tensor, Tensor, Tensor>{grad_input, grad_weight, grad_bias};
}
```

Gradient na vstup x

Gradient na parametry (váhy a bias)

Příklad: implementace lineární vrstvy v PyTorch pro MKLDNN

https://github.com/pytorch/pytorch/blob/5ed60477a7cd930298dbdab71046ec62024427c4/aten/src/ATen/native/mkldnn/Linear.cpp#L108

```
Tensor mkldnn_linear_backward_input(
    IntArrayRef input_size, const Tensor& grad_output, const Tensor& weight_t){
  TORCH_CHECK(grad_output.is_mkldnn(),
      "mkldnn_linear_backward: grad_output needs to be mkldnn layout");
  TORCH_CHECK(weight_t.device().is_cpu() && weight_t.scalar_type() == kFloat,
      "mkldnn_linear_backward: weight_t needs to be a dense tensor");
  auto grad_output_reshaped = grad_output.dim() > 2 ?
    grad_output.reshape({-1, grad_output.size(grad_output.dim() - 1)}) : grad_output;

  ideep::tensor& grady = itensor_from_mkldnn(grad_output_reshaped);
...
  ideep::tensor gradx;
  ideep::inner_product_backward_data::compute(
    grady, w, {input_reshaped_size.begin(), input_reshaped_size.end()}, gradx);
```

Řetízkové pravidlo schované v knihovně https://github.com/intel/ideep

Řetízkové pravidlo pro funkce $\mathbb{R}^D \to \mathbb{R}^K$

Jakobián, sigmoid, maticové násobení

Opakované použití jedné proměnné

- Jeden vstup $x \in \mathbb{R}$ je použitý pro výpočet více mezivýsledků $z_k \in \mathbb{R}$

$$z_k = g_k(x)$$

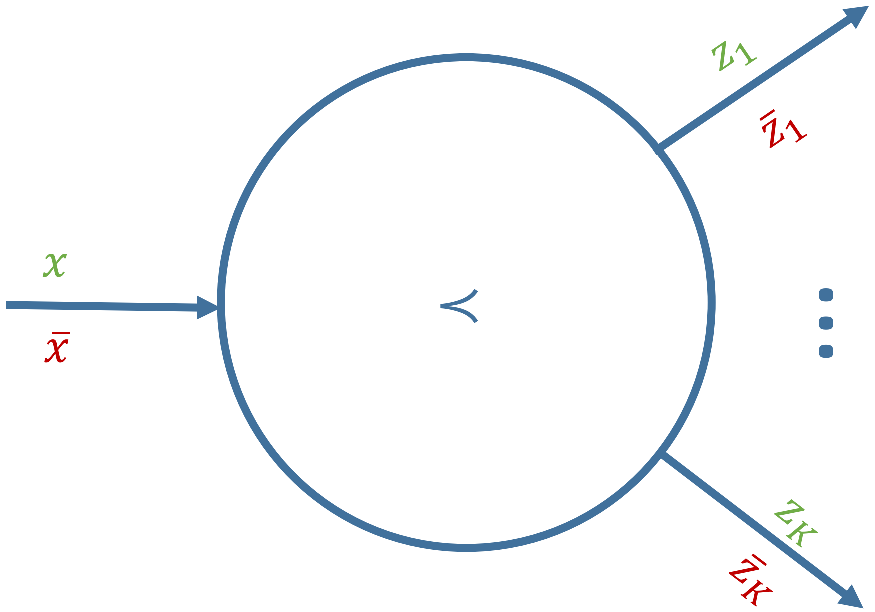
- Konečný výsledek (např. loss) je závislý na obou mezivýsledcích

$$\begin{aligned} L &= f(z_1, \dots, z_K) \\ &= f\big(g_1(x), \dots, g_K(x)\big) \end{aligned}$$

- Aplikuje se řetízkové pravidlo pro funkce více proměnnných

$$\begin{aligned} \frac{\partial L}{\partial x} &= \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial x} + \cdots + \frac{\partial L}{\partial z_K} \cdot \frac{\partial z_K}{\partial x} \\ &= \bar{z}_1 \cdot 1 + \cdots + \bar{z}_K \cdot 1 \end{aligned}$$

- Všimněme si duality vůči bloku sčítání



Příchozí gradienty $\bar{z}_k$ se sečtou

forward: $\boldsymbol{z} = [x, \dots, x]^\top$

backward: $\bar{x} = \sum_{k=1}^{K} \bar{z}_k$

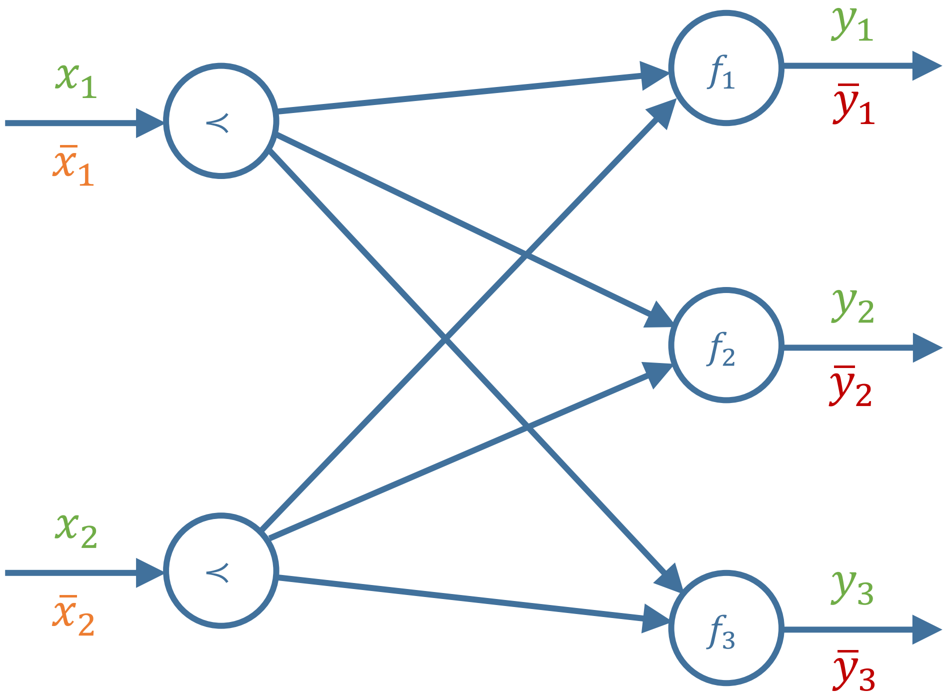
https://math.stackexchange.com/questions/1566376/derivation-of-the-multivariate-chain-rule

Řetízkové pravidlo pro funkce více proměnných

$$\frac{\partial y_k}{\partial x_d}$$

$$\bar{x}_1 = \bar{y}_1 \cdot \frac{\partial y_1}{\partial x_1} + \bar{y}_2 \cdot \frac{\partial y_2}{\partial x_1} + \bar{y}_3 \cdot \frac{\partial y_3}{\partial x_1}$$

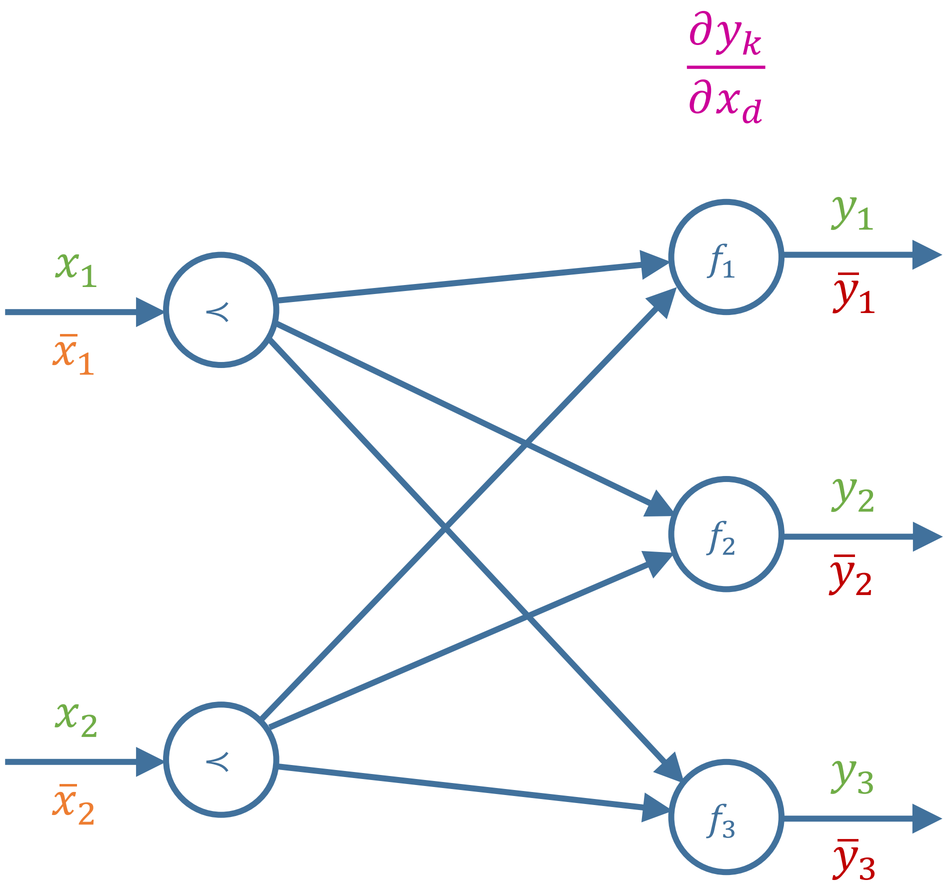
$$\bar{x}_2 = \bar{y}_1 \cdot \frac{\partial y_1}{\partial x_2} + \bar{y}_2 \cdot \frac{\partial y_2}{\partial x_2} + \bar{y}_3 \cdot \frac{\partial y_3}{\partial x_2}$$



Řetízkové pravidlo pro funkce více proměnných

$$\bar{x}_1 = \begin{bmatrix} \bar{y}_1 & \bar{y}_2 & \bar{y}_3 \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \frac{\partial y_2}{\partial x_1} \\ \frac{\partial y_3}{\partial x_1} \end{bmatrix}$$

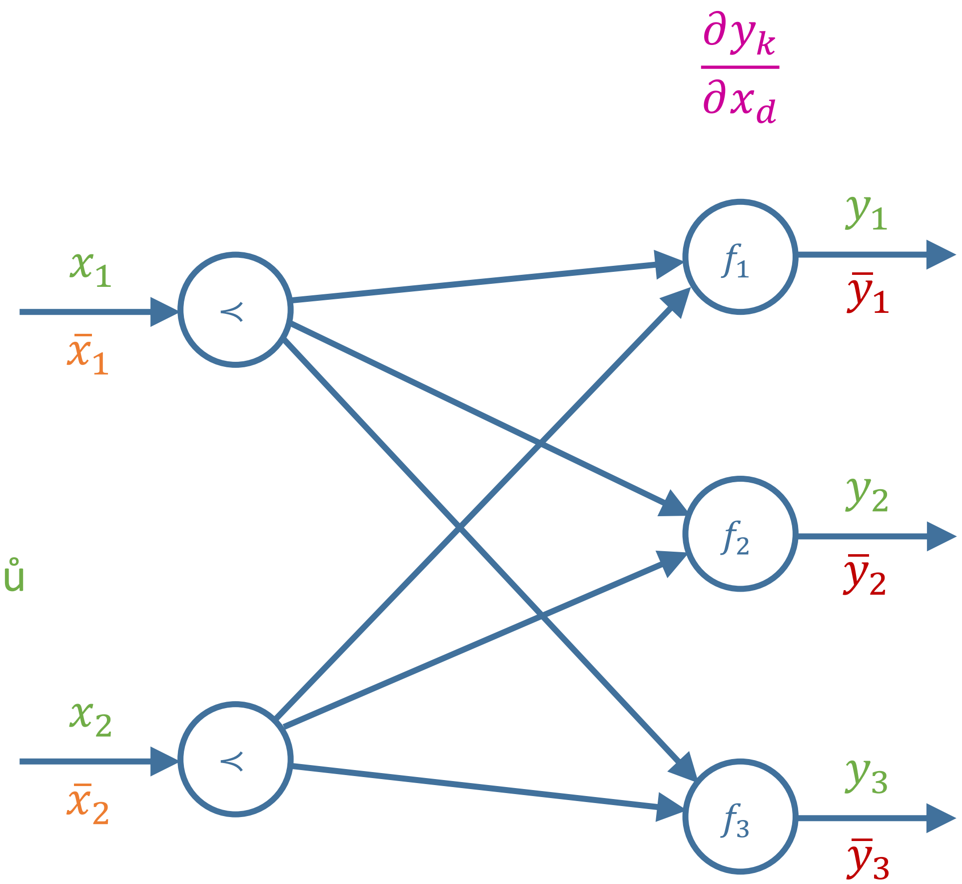
$$\bar{x}_2 = \begin{bmatrix} \bar{y}_1 & \bar{y}_2 & \bar{y}_3 \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_2} \\ \frac{\partial y_3}{\partial x_2} \end{bmatrix}$$



Řetízkové pravidlo pro funkce více proměnných

Matice paricálních derivací všech výstupů y_k vůči všem vstupům x_d se nazývá **jakobián**

$$[\bar{x}_1 \quad \bar{x}_2] = [\bar{y}_1 \quad \bar{y}_2 \quad \bar{y}_3] \cdot \left.\underbrace{\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} \end{bmatrix}}_{\text{počet vstupů}}\right\} \text{počet výstupů}$$



řetízkové pravidlo:

odchozí_gradient = příchozí_gradient x lokální_gradient

Řetízkové pravidlo pro funkce více proměnných

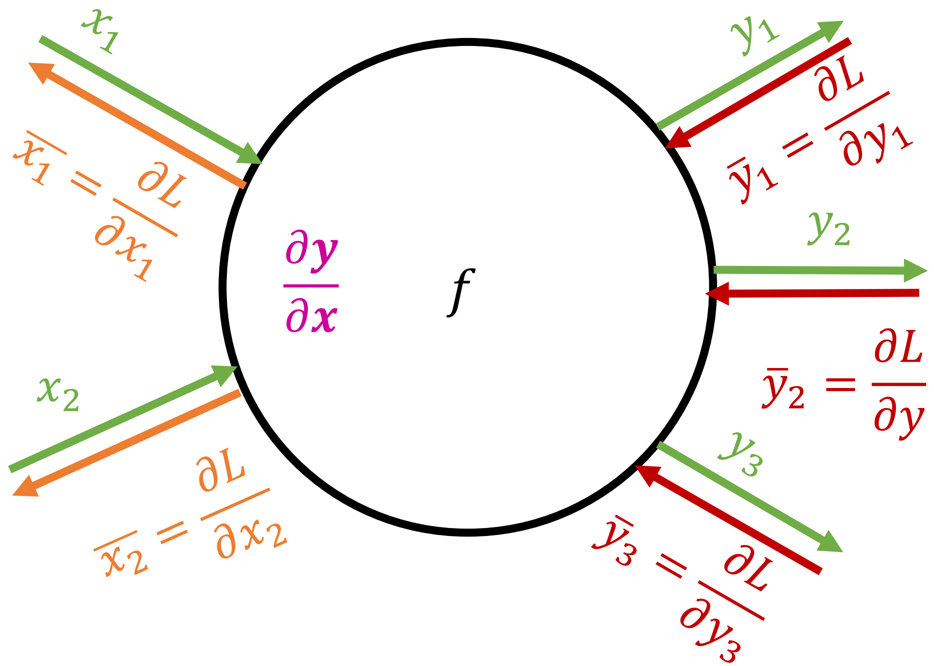
Matice paricálních derivací všech výstupů y_k vůči všem vstupům x_d se nazývá **jakobián**

$$[\bar{x}_1 \quad \bar{x}_2] = [\bar{y}_1 \quad \bar{y}_2 \quad \bar{y}_3] \cdot \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} \end{bmatrix}$$

řetízkové pravidlo:

$$\underset{1 \times D}{\boxed{\frac{\partial L}{\partial \boldsymbol{x}}}} = \underset{1 \times K}{\boxed{\frac{\partial L}{\partial \boldsymbol{y}}}} \cdot \underset{K \times D}{\boxed{\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}}}$$

Lokální gradient je jakobián
$K \times D$



Odchozí gradient
$1 \times D$

Příchozí gradient
$1 \times K$

Řetízkové pravidlo pro funkce více proměnných sloupcově

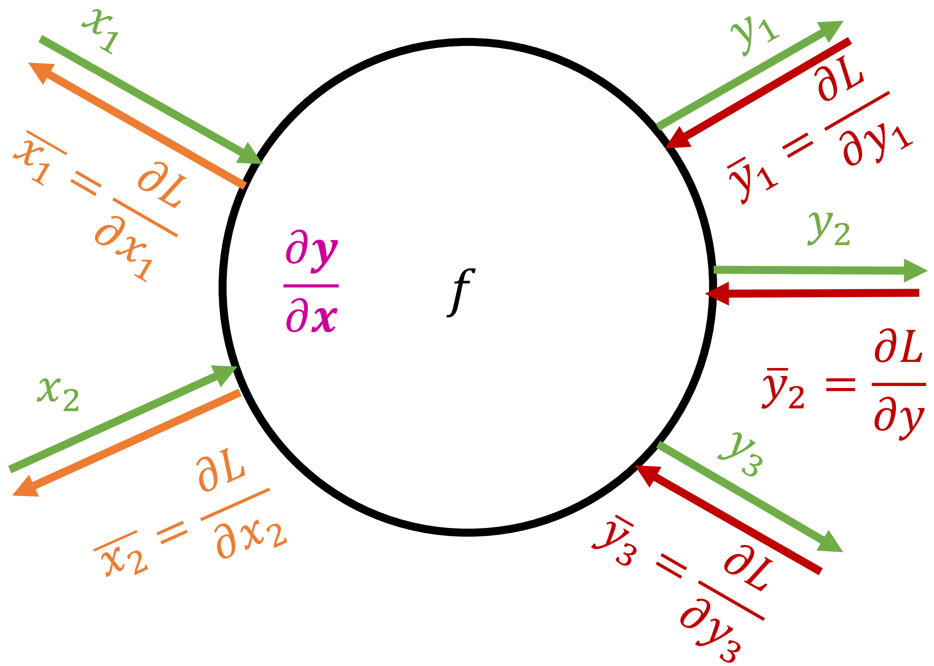
Matice paricálních derivací všech výstupů y_k vůči všem vstupům x_d se nazývá **jakobián** (transponovaný)

$$\begin{bmatrix}\bar{x}_1 \\ \bar{x}_2\end{bmatrix} = \begin{bmatrix}\frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_3}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_3}{\partial x_2}\end{bmatrix} \cdot \begin{bmatrix}\bar{y}_1 \\ \bar{y}_2 \\ \bar{y}_3\end{bmatrix}$$

řetízkové pravidlo sloupcově:

$$\underbrace{\frac{\partial L}{\partial \boldsymbol{x}}}_{D \times 1} = \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top}}_{D \times K} \cdot \underbrace{\frac{\partial L}{\partial \boldsymbol{y}}}_{K \times 1}$$

Lokální gradient je jakobián
$K \times D$



Odchozí gradient
$1 \times D$

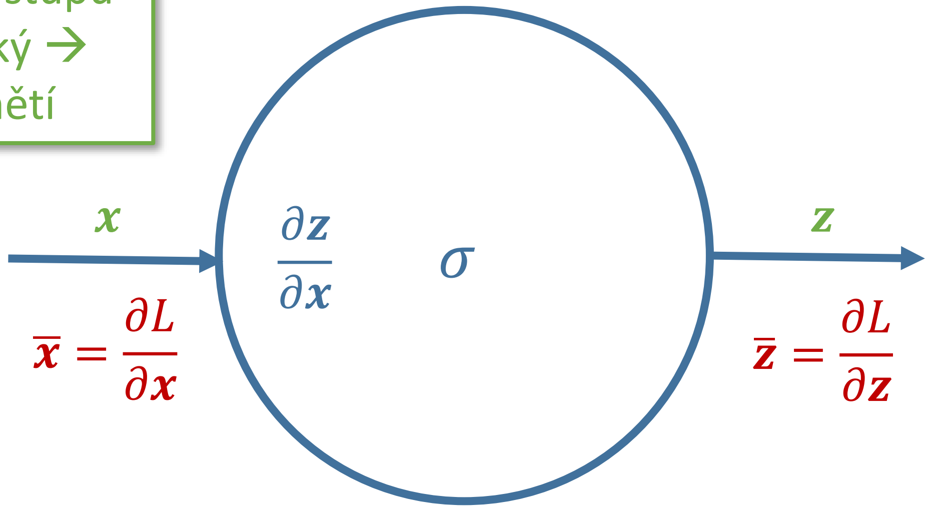
Příchozí gradient
$1 \times K$

Příklad: <u>**Vektorová**</u> funkce sigmoid jako uzel ve výpočetním grafu

forward:

$$\begin{bmatrix} z_1 \\ \vdots \\ z_D \end{bmatrix} = \sigma\left(\begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}\right) = \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_D) \end{bmatrix}$$

Jakobián má rozměr počet_výstupů x počet_vstupů, přitom je řídký → velmi neefektivní práce s pamětí



lokální gradient (jakobián):

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \begin{bmatrix} z_1 \cdot (1 - z_1) & 0 & 0 \\ 0 & z_2 \cdot (1 - z_2) & 0 \\ 0 & 0 & z_3 \cdot (1 - z_3) \end{bmatrix}$$

backward (řetízkové pravidlo):

$$\begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_D \end{bmatrix} = \begin{bmatrix} z_1 \cdot (1 - z_1) & 0 & 0 \\ 0 & z_2 \cdot (1 - z_2) & 0 \\ 0 & 0 & z_3 \cdot (1 - z_3) \end{bmatrix} \cdot \begin{bmatrix} \bar{z}_1 \\ \vdots \\ \bar{z}_D \end{bmatrix}$$

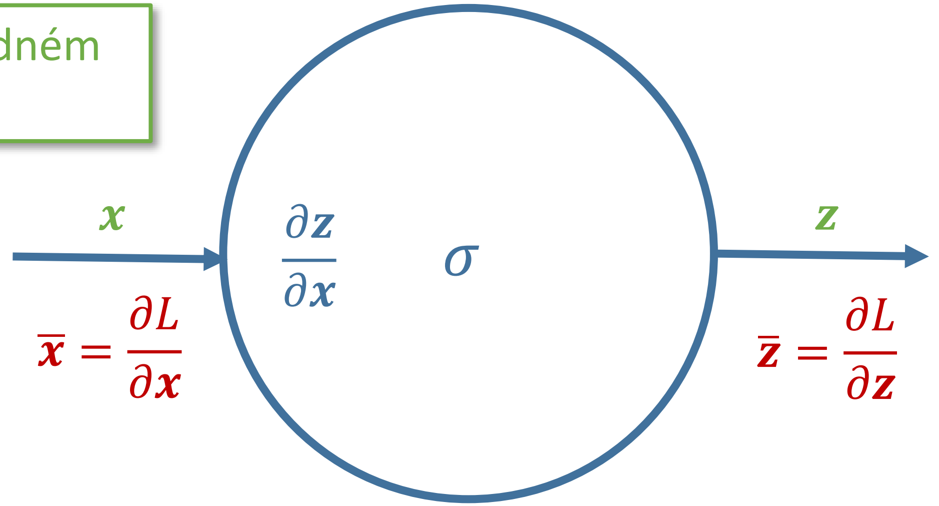
Navíc spousta zbytečného násobení

Příklad: <u>**Vektorová**</u> funkce sigmoid jako uzel ve výpočetním grafu

forward:

$$\begin{bmatrix} z_1 \\ \vdots \\ z_D \end{bmatrix} = \sigma\left(\begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}\right) = \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_D) \end{bmatrix}$$

z_k je závislé pouze na x_k a žádném jiném vstupu



lokální gradient prvkově:

$$\frac{\partial z_k}{\partial x_d} = \begin{cases} z_k \cdot (1 - z_k) & k = d \\ 0 & k \neq d \end{cases}$$

backward (řetízkové pravidlo aplikované po prvcích):

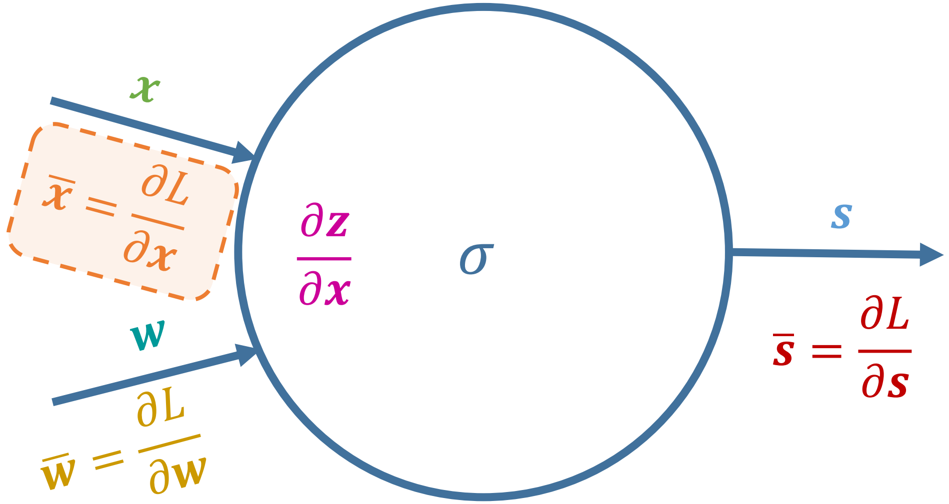
$$\begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_D \end{bmatrix} = \begin{bmatrix} z_1 \cdot (1 - z_1) \\ \vdots \\ z_K \cdot (1 - z_K) \end{bmatrix} \odot \begin{bmatrix} \bar{z}_1 \\ \vdots \\ \bar{z}_D \end{bmatrix}$$

Pouze tolik násobení, kolik je prvků vstupu → mnohem efektivnější, přitom ekvivalentní řetízkovému pravidlu s jakobiánem

Příklad: Maticové násobení

forward:

$$\underset{K \times N}{\begin{bmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \end{bmatrix}} = \underset{K \times D}{\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}} \cdot \underset{D \times N}{\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{bmatrix}}$$



lokální gradient (jakobián):

řádky = počet_výstupů = $K \cdot N$

sloupce = počet_vstupů = $(K \cdot D + D \cdot N)$

rozměr jakobiánu = $(K \cdot N) \times (D \cdot N)$

backward (řetízkové pravidlo) na vstup $\boldsymbol{x}$:

$$\begin{bmatrix} \bar{x}_{1,1} \\ \vdots \\ \bar{x}_{D,N} \end{bmatrix} = \begin{bmatrix} \frac{\partial s_{1,1}}{\partial x_{1,1}} & \cdots & \frac{\partial s_{1,1}}{\partial w_{K,D}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s_{K,N}}{\partial x_{1,1}} & \cdots & \frac{\partial s_{K,N}}{\partial x_{1,1}} \end{bmatrix} \cdot \begin{bmatrix} \bar{s}_{1,1} \\ \vdots \\ \bar{s}_{K,N} \end{bmatrix}$$

- Běžné hodnoty pro neuronové sítě jsou např. $N = 64$, $D = K = 4096$
- Jakobián by pak měl $4096 \cdot 64 \cdot (4096 \cdot 4096 + 4096 \cdot 64) = 4.5 \cdot 10^{12}$ prvků a v paměti při float32 zabíral např. 16640 GB!
- Navíc je spousta prvků v matici nulová → zbytečná násobení
- Mechanická aplikace řetízkového pravidla s jakobiánem proto není praktická

Příklad: Maticové násobení

forward:

$$\begin{bmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{bmatrix}$$

$K \times N$ $\quad$ $K \times D$ $\quad$ $D \times N$

lokální gradient prvkově:

$$\frac{\partial s_{k,m}}{\partial x_{d,n}} = \begin{cases} w_{k,d} & \text{pokud } m = n \\ 0 & \text{jinak} \end{cases}$$

hodnoty v m-tém sloupci $\boldsymbol{s}$ závisejí pouze na m-tém sloupci $\boldsymbol{x}$ → jinde je gradient nula

backward (řetízkové pravidlo pro $\bar{\boldsymbol{x}}$):

$$\underset{D \times N}{\bar{\boldsymbol{x}}} = \underset{D \times K}{\boldsymbol{w}^\top} \cdot \underset{K \times N}{\bar{\boldsymbol{s}}}$$

aplikujeme $\boldsymbol{w}_{:,d} \cdot \bar{\boldsymbol{s}}_{:,n}$ pro všechny (d, n)

pro každé $\bar{x}_{d,n}$ aplikujeme řetízkové pravidlo pro funkce více proměnných:

$$\begin{aligned}
\bar{x}_{d,n} &= \bar{s}_{1,1} \cdot \frac{\partial s_{1,1}}{\partial x_{d,n}} + \cdots + \bar{s}_{K,D} \cdot \frac{\partial s_{K,D}}{\partial x_{d,n}} \\
&= \bar{s}_{1,n} \cdot \frac{\partial s_{1,n}}{\partial x_{d,n}} + \cdots + \bar{s}_{K,n} \cdot \frac{\partial s_{K,n}}{\partial x_{d,n}} \\
&= \bar{s}_{1,n} \cdot w_{1,d} + \cdots + \bar{s}_{K,n} \cdot w_{K,d} \\
&= \boldsymbol{w}_{:,d} \cdot \bar{\boldsymbol{s}}_{:,n}
\end{aligned}$$

vyškrtnuty nulové $\frac{\partial s_{k,m}}{\partial x_{d,n}}$

d-tý sloupec $\boldsymbol{w}$ krát n-tý sloupec $\bar{\boldsymbol{s}}$

Příklad: Maticové násobení

forward:

$$\begin{bmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{bmatrix}$$

$$\underset{K \times N}{} \qquad \underset{K \times D}{} \qquad \boxed{D \times N}$$

Jakákoliv proměnná x a gradient na ní $\bar{x} = \partial L / \partial x$ musí mít vždy shodné rozměry!

lokální gradient prvkově:

$$\frac{\partial s_{k,m}}{\partial x_{d,n}} = \begin{cases} w_{k,d} & \text{pokud } m = n \\ 0 & \text{jinak} \end{cases}$$

hodnoty v m-tém sloupci $\boldsymbol{s}$ závisejí pouze na m-tém sloupci $\boldsymbol{x}$ → jinde je gradient nula

backward (řetízkové pravidlo pro $\bar{\boldsymbol{x}}$):

$$\underset{\boxed{D \times N}}{\bar{\boldsymbol{x}}} = \underset{D \times K}{\boldsymbol{w}^\top} \cdot \underset{K \times N}{\bar{\boldsymbol{s}}}$$

aplikujeme $\boldsymbol{w}_{:,d} \cdot \bar{\boldsymbol{s}}_{:,n}$ pro všechny (d, n)

backward (řetízkové pravidlo pro $\bar{\boldsymbol{w}}$):

$$\underset{K \times D}{\bar{\boldsymbol{w}}} = \underset{K \times N}{\bar{\boldsymbol{s}}} \cdot \underset{N \times D}{\boldsymbol{x}^\top}$$

Reverzní automatické derivování

Autograd

Reverzní automatické derivování

Popsaný způsob zpětné propagace, kdy máme funkci $f: \mathbb{R}^D \rightarrow \mathbb{R}$

$$z = f(x_1, \dots, x_D)$$

která mapuje

- vektor vstupů $\boldsymbol{x} = [x_1, \dots, x_D]^\top$ ($\boldsymbol{x}$ mohou být např. parametry klasifikátoru)
- na skalár z (např. hodnota klasifikačního lossu na trénovacím datasetu)

a my opakovanou aplikací řetízkového pravidla zpětně od z k $\boldsymbol{x}$ počítáme gradient

$$\nabla f(\boldsymbol{x}) = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_D}\right] = \overline{\boldsymbol{x}}$$

Reverse-mode AD počítá
“jak každý ze vstupů funkce ovlivňuje
její výstupní (skalární) hodnotu?”

se označuje jako **reverzní automatické derivování**

- Reverse-mode automatic differentiation
- https://en.wikipedia.org/wiki/Automatic_differentiation
- Funkce f může být libovolně složitá, musí ovšem být složena ze základních diferencovatelných operací, u nichž umíme spočítat zpětný průchod řetízkovým pravidlem

Implementace reverzního automatického derivování

- **Problém: jak budeme reprezentovat výpočetní graf?**
- Existuje mnoho různých implementací
- <u>**Deklarativní způsob**</u>
  - např. Google tensorflow v1
  - nejprve sestavíme výpočetní graf z připravených bloků ve frameworku
  - poté voláme s konkrétními daty
  - graf se po vytvoření už nemění

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

```
# Construct model
logits = neural_net(X)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
```

Implementace reverzního automatického derivování

- **Problém: jak budeme reprezentovat výpočetní graf?**
- Existuje mnoho různých implementací
- **Imperativní způsob**
  - např. PyTorch nebo tensorflow v2, tzv. eager mode
  - Každá proměnná (např. váhová matice klasifikátoru) je speciální objekt
  - Při jakékoliv operaci nad proměnnou (např. sčítání, násobení, umocňování atd.) v dopředném průchodu si uložíme informaci, co se stalo, které proměnné byly vstupy a které výstupy a jak bude vypadat zpětný průchod
  - Zpětný průchod zavoláme, kdy potřebujeme a dostaneme zpět gradienty
  - graf vzniká "za pochodu"

s = f(a, b)

```
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
q = 3*a**3 - b**2
s = q.sum()
s.backward()
a.grad, b.grad
```

```
(tensor([36., 81.]), tensor([-12.,  -8.]))
```

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

Každá proměnná v grafu bude objekt třídy `Var`, který uchovává:
`value` … svojí hodnotu, např. váhová matice
`children` … odkaz na potomky v grafu jako seznam

Třída `Var` přepisuje metody operací jako sčítání, násobení, apod.

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

Přepíšeme např. operátor násobení

Druhý potomek je `other` a jeho lokální gradient je hodnota `self`

První potomek je `self` a jeho lokální gradient je hodnota `other`

Nová hodnota je součin `self` a `other`

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

Zpětná propagace spočívá v rekurzivním procházení grafu a u každého potomka aplikaci řetízkového pravidla

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

```python
# Example: f(x, y) = x * y + sin(x)
x = Var(2)
y = Var(3)
f = x * y + x.sin()

# Calculation of partial derivatives
f.calc_grad()

print("f =", f.value)      # f = 6.909297426825682
print("∂f/∂x =", x.grad)  # ∂f/∂x = 2.5838531634528574
print("∂f/∂y =", y.grad)  # ∂f/∂y = 2
```

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

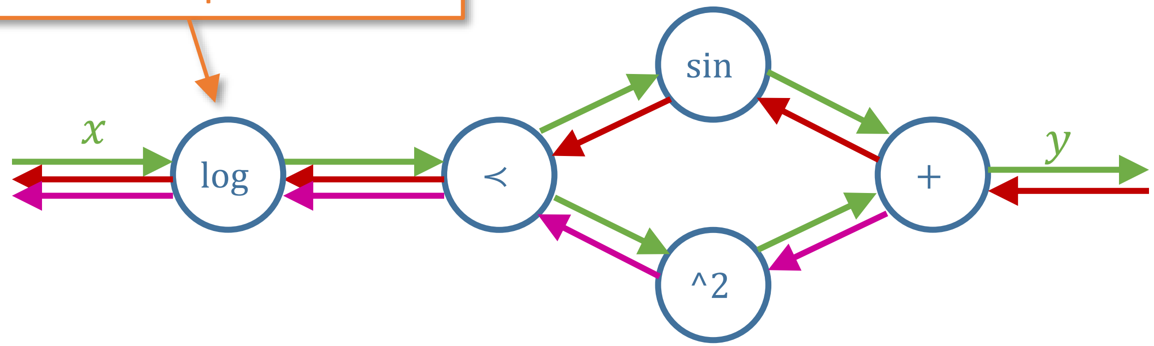
Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

Problém

$$y = \sin(\log(x)) + \log(x)^2$$

Zde řetízkové pravidlo 2x!



Graf se prochází depth-first a pokud se nějaká proměnná vyskytuje vícekrát, bude se vícekrát i volat její backward a celý její pod-strom

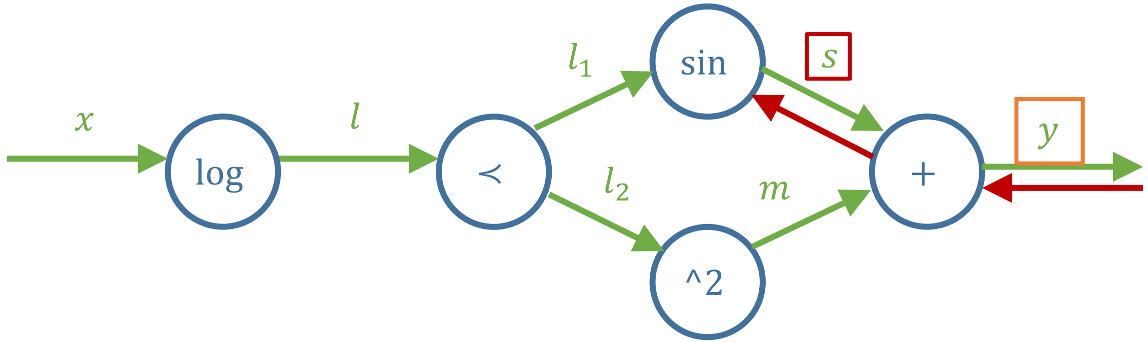
Řetízkové pravidlo může být výpočetně náročné pro velké matice → chceme počítat max. jednou

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad   # no recursion
```

$y = \sin(\log(x)) + \log(x)^2$



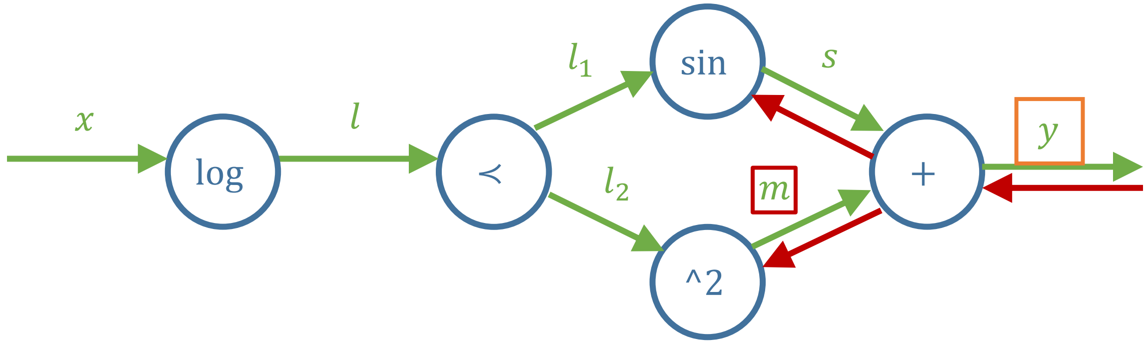
variables = $[x, l, l_1, l_2, \boxed{s}, m, \boxed{y}]$

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz Google colab notebook

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

$$y = \sin(\log(x)) + \log(x)^2$$




```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad  # no recursion
```

variables = $[x, l, l_1, l_2, s, \boxed{m}, \boxed{y}]$

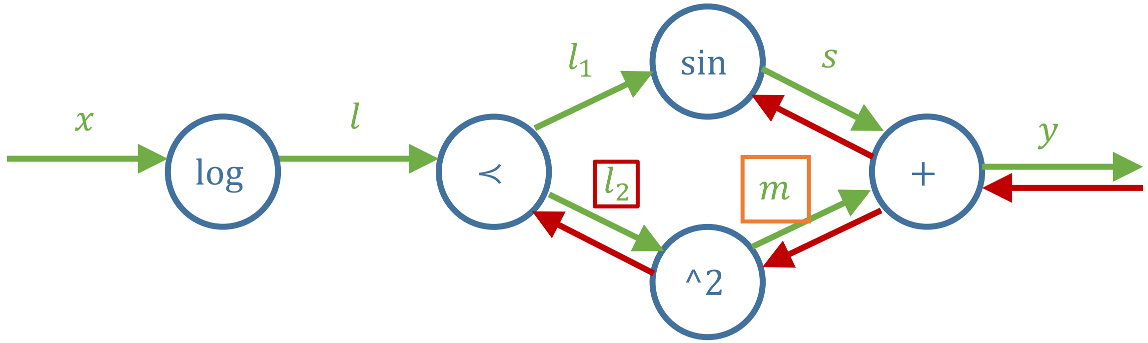
V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz Google colab notebook

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad   # no recursion
```

$$y = \sin(\log(x)) + \log(x)^2$$



variables = $[x, l, l_1, \boxed{l_2}, s, \boxed{m}, y]$

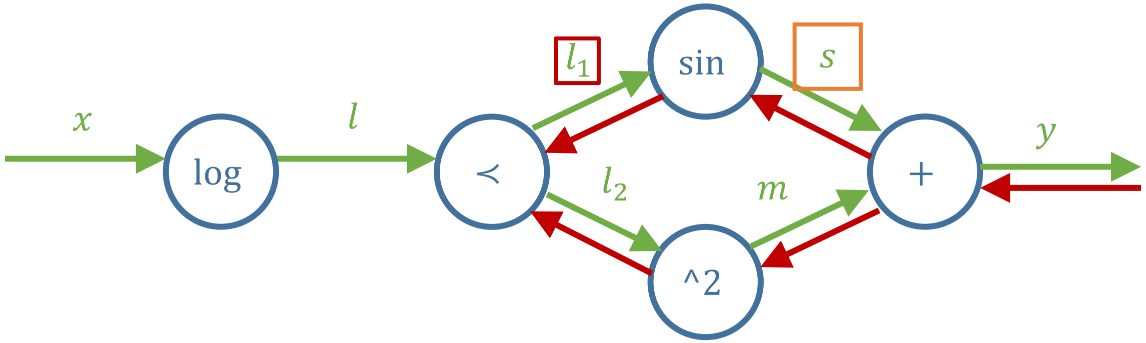
Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz [Google colab notebook]()

Imperativní reverzní automatické derivování v Pythonu

```
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad   # no recursion
```

$$y = \sin(\log(x)) + \log(x)^2$$



variables = $[x, l, \boxed{l_1}, l_2, \boxed{s}, m, y]$

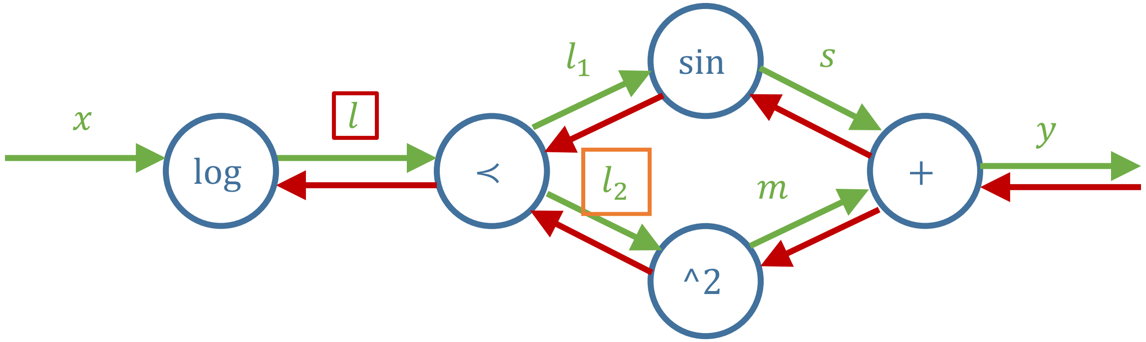
V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz [Google colab notebook]

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad  # no recursion
```

$$y = \sin(\log(x)) + \log(x)^2$$



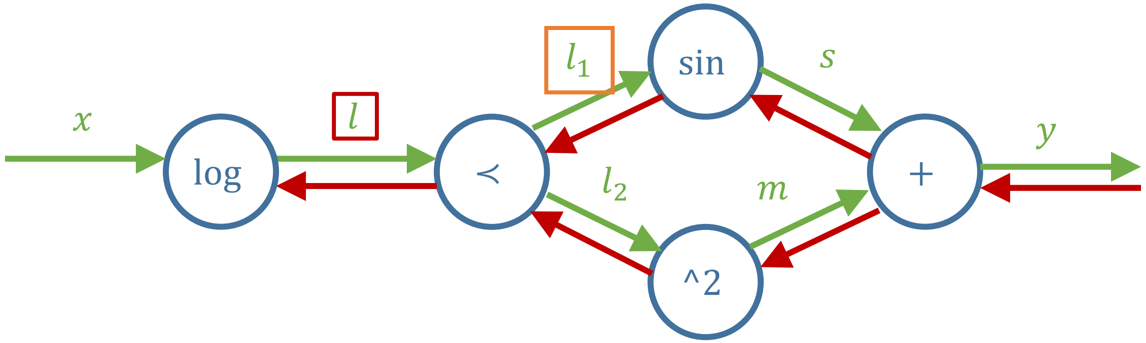
variables = $[x, \boxed{l}, l_1, \boxed{l_2}, s, m, y]$

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz [Google colab notebook]()

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

$$y = \sin(\log(x)) + \log(x)^2$$




```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad  # no recursion
```

variables = $[x, \boxed{l}, \boxed{l_1}, l_2, s, m, y]$

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz Google colab notebook

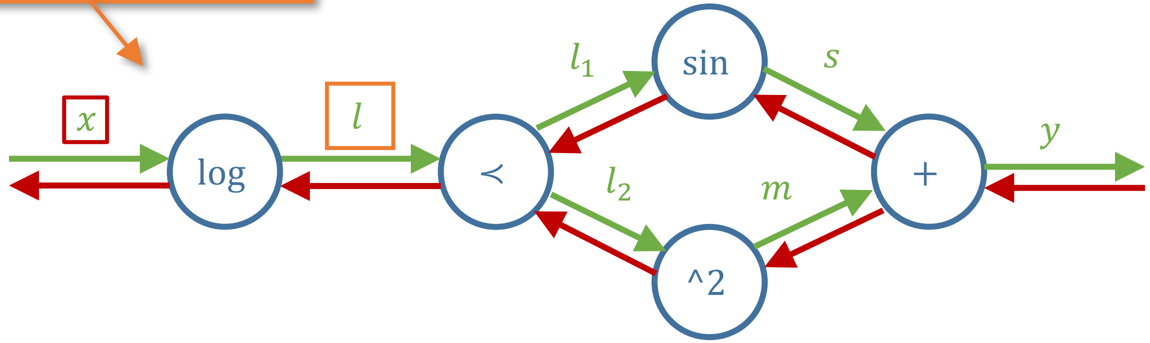
Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad  # no recursion
```

Gradient plně akumulován = řetízkové pravidlo pouze 1x

$y = \sin(\log(x)) + \log(x)^2$



variables = $[\boxed{x}, \boxed{l}, l_1, l_2, s, m, y]$

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz [Google colab notebook]

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Imperativní reverzní automatické derivování v Pythonu

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        variables = self.topologically_sorted_list_of_graph_variables()
        for var in reversed(variables):
            for coef, child in var.children:
                child.grad += coef * var.grad  # no recursion
```

Implementace třídy Variable bude náplní cvičení

Jiná možnost je sestavit seznam rekurzivním procházením atributů children

variables = $[x, l, l_1, l_2, s, m, y]$

V PyTorch se seznam postupně tvoří během dopředného průchodu a označuje se jako gradient tape, pro tutorial viz [Google colab notebook]()

Příklad:
https://en.wikipedia.org/wiki/Automatic_differentiation#Python

Závěr

Zpětná propagace a řetízkové pravidlo

- Zpětná propagace je způsob, jak u libovolně složité diferencovatelné funkce $f: \mathbb{R}^D \to \mathbb{R}$

$$z = f(x_1, \dots, x_D)$$

získat gradient

$$\nabla z = \left[\frac{\partial z}{\partial x_1}, \dots, \frac{\partial z}{\partial x_D}\right]$$

- Pokud lze funkci rozdělit na jednodušší diferencovatelné bloky

$$z = f\big(g(x_1, \dots, x_D)\big)$$

pak spočívá v opakované aplikaci řetízkového pravidla

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial g} \cdot \frac{\partial g}{\partial x}$$

odchozí_gradient = příchozí_gradient x lokální_gradient

v každém uzlu výpočetního grafu, tzv. reverzním automatickým derivováním

Funkce $y = f(x)$ a její derivace v závislosti na rozměrovosti

$x \in \mathbb{R}, y \in \mathbb{R}$

derivace je **jednorozměrná**, tj.

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \mathbb{R}$$

Jak se změní y, pokud o trochu změníme x

$x \in \mathbb{R}^D, y \in \mathbb{R}$

derivace je **gradient**, tj.

$$\frac{\partial y}{\partial x} = \mathbb{R}^D$$

$$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x_1}, \dots, \frac{\partial y}{\partial x_D}\right]$$

Jak se změní y, pokud o trochu změníme d-tý prvek vstupu x_d - pro všechna d

$x \in \mathbb{R}^D, y \in \mathbb{R}^K$

derivace je **jakobián**, tj.

$$\frac{\partial y}{\partial x} = \mathbb{R}^{K \times D}$$

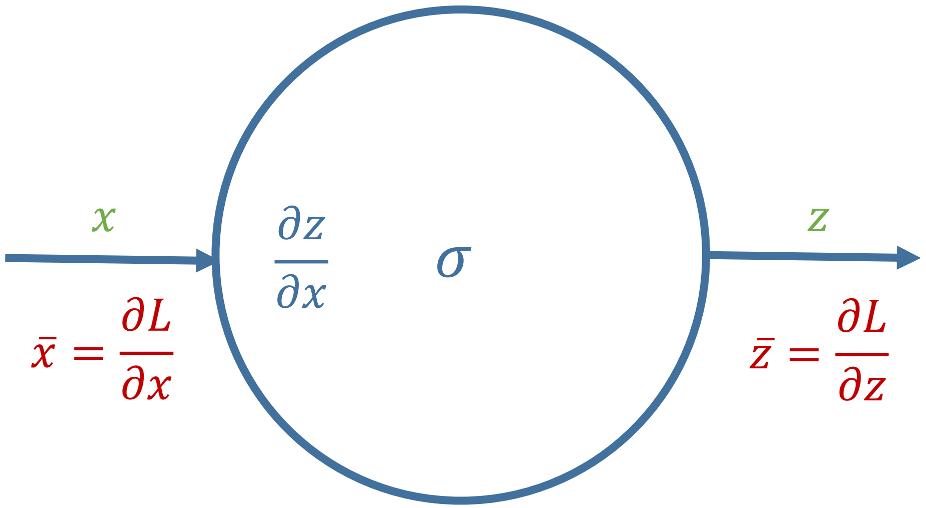
$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \cdots & \frac{\partial y_K}{\partial x_D} \end{bmatrix}$$

Jak se změní k-tý výstup y_k, pokud o trochu změníme d-tý vstup x_d - pro všechny dvojice (k, d)

slide: https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/598_WI2022_lecture06.pdf

Funkce jako uzel ve výpočetním grafu

```python
class Sigmoid(Function):

    @staticmethod
    def forward(x: float):  # zatim pouze skalary
        z = 1 / (1 - math.exp(-x))
        cache = z,
        return z, cache

    @staticmethod
    def backward(dz: float, cache: tuple):
        z, = cache
        dx = dz * z * (1 - z)  # retizkove pravidlo
        return dx
```



zpětný průchod je řetízkové pravidlo

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$$

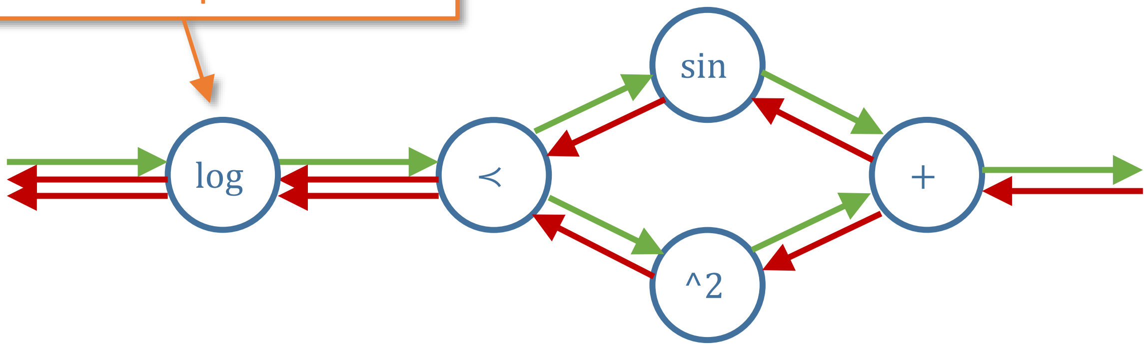
Každá funkce, kterou chceme použít jako stavební blok, musí mít definovaný

dopředný průchod

zpětný průchod

Řetízkové pravidlo volat pro každý uzel právě jednou

Zde řetízkové pravidlo 2x!



S **<u>topologickým řazením</u>** gradient plně akumulován = řetízkové pravidlo pouze 1x