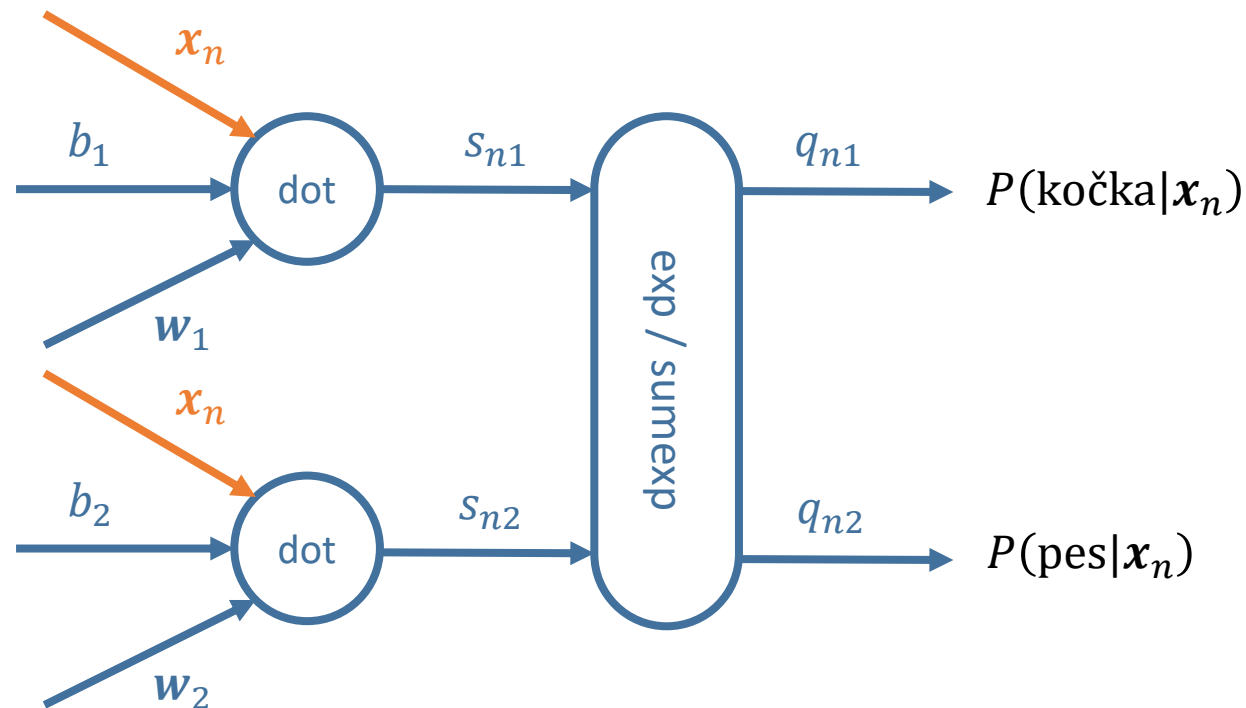


Aplikace neuronových sítí

Zpětná propagace

Multiclass logistická regrese (softmax klasifikátor)



$$q_{nc} = P(\text{třída } c|\mathbf{x}_n) = \frac{e^{s_{nc}}}{\sum_{i=1}^C e^{s_{ni}}}$$

Diferencovatelné výpočetní grafy

Řetízkové pravidlo (chain rule)

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



$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$



- “derivace vnější krát derivace vnitřní”

Příklad na řetízkové pravidlo

$$z = \underbrace{(x_1 + ax_2)}_g^f$$

vnitřní funkce g :

$$y = x_1 + ax_2$$

$$\frac{\partial y}{\partial x_1} = 1 \quad \frac{\partial y}{\partial x_2} = a$$

vnější funkce f :

$$z = y^2$$

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

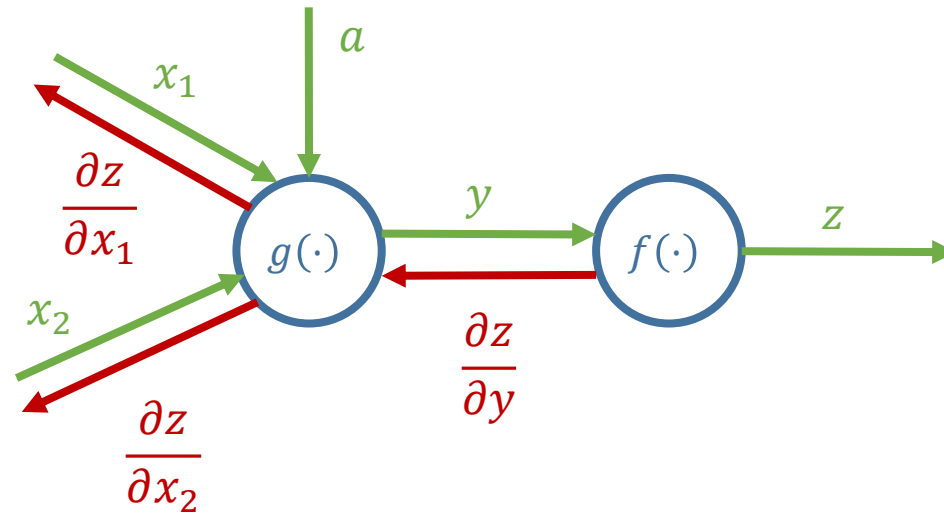
aplikace pravidla:

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = 2y \cdot 1 = 2(x_1 + ax_2)$$

$$\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_2} = 2y \cdot a = 2a(x_1 + ax_2)$$

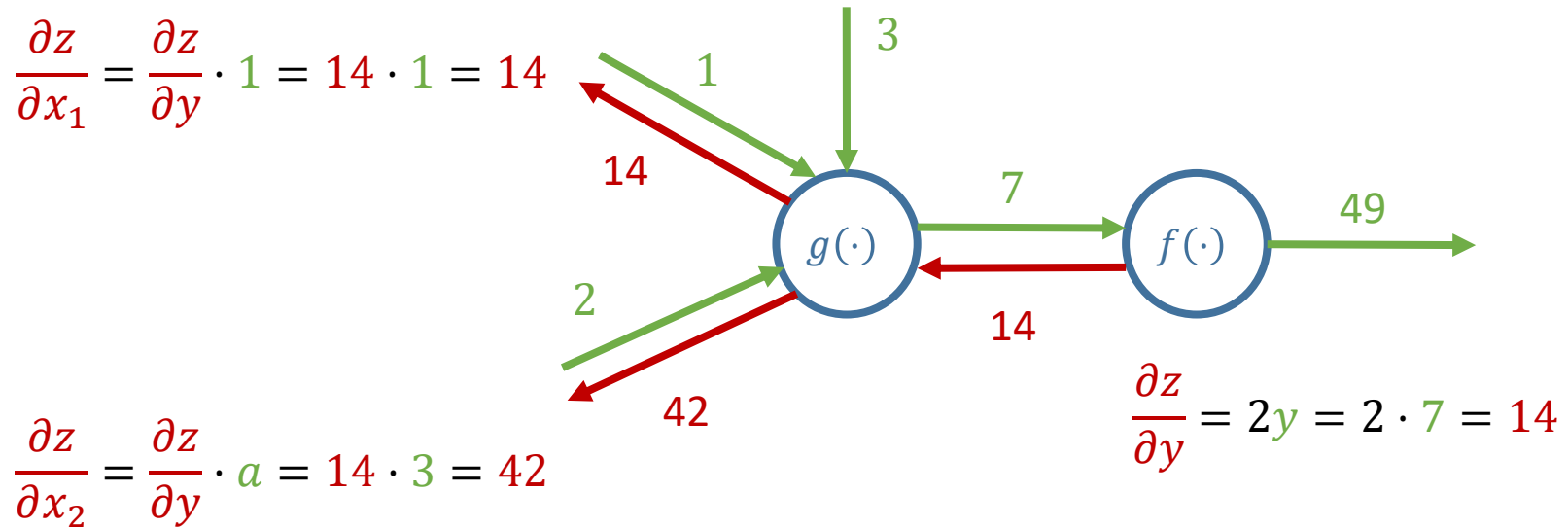
Funkce $z = (x_1 + ax_2)^2$ jako graf

$$z = (x_1 + ax_2)^2$$



Funkce $z = (x_1 + ax_2)^2$ jako graf

$$z = (x_1 + ax_2)^2$$



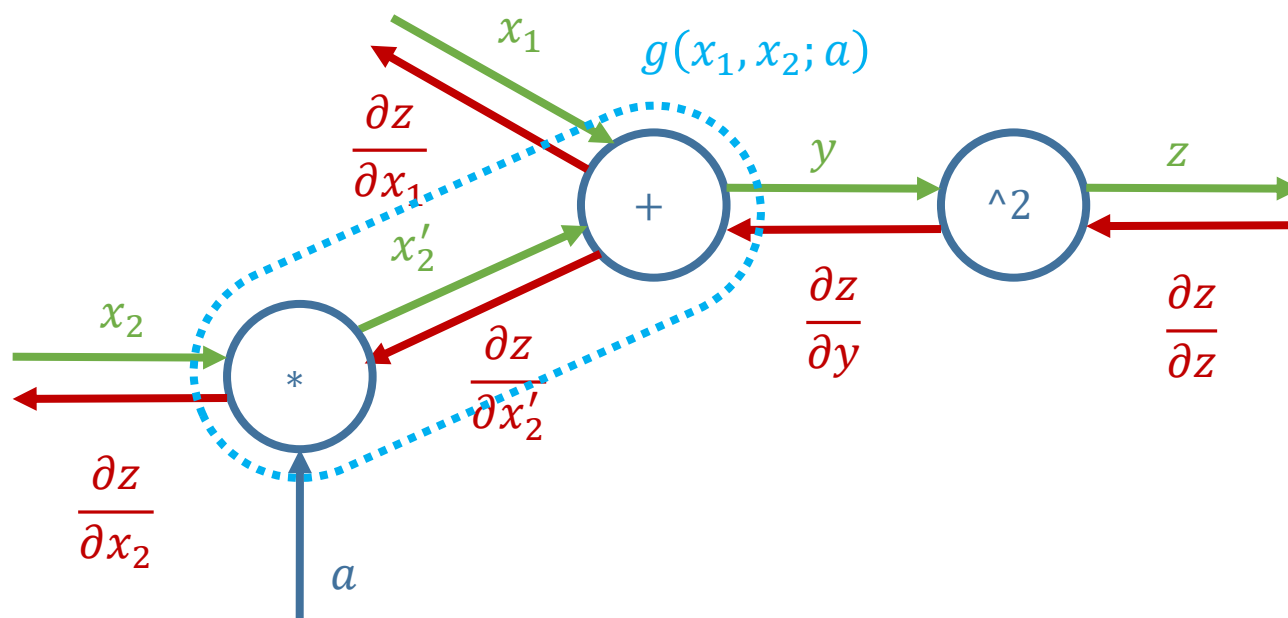
Funkce $z = (x_1 + ax_2)^2$ jako graf podrobně

dopředný průchod

- (1) $x'_2 \leftarrow ax_2$
- (2) $y \leftarrow x_1 + x'_2$
- (3) $z \leftarrow y^2$

zpětný průchod

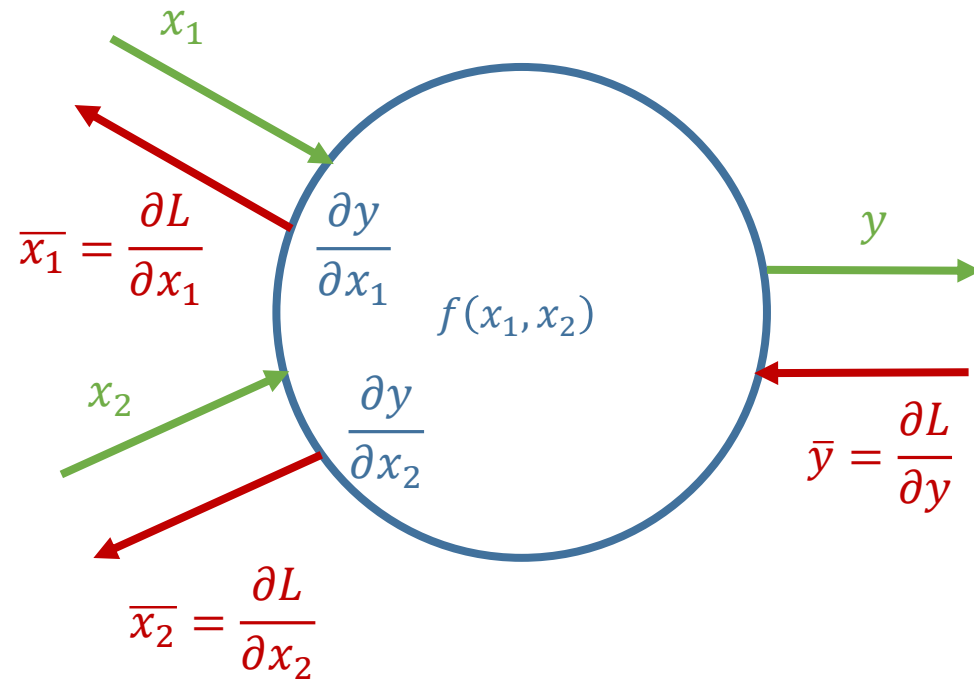
- (0) $\frac{\partial z}{\partial z} = 1$
- (1) $\frac{\partial z}{\partial y} \leftarrow \frac{\partial z}{\partial z} \cdot \frac{\partial z}{\partial y} = 2y$
- (2) $\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = \frac{\partial z}{\partial y}$
 $\frac{\partial z}{\partial x'_2} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x'_2} = \frac{\partial z}{\partial y}$
- (3) $\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial x'_2} \cdot \frac{\partial x'_2}{\partial x_2} = \frac{\partial z}{\partial x'_2} \cdot a$



Uzel grafu a lokální gradient

- Na funkci lze nahlížet jako na orientovaný výpočetní graf
- Jednotlivé operace jsou uzly
- Hrany reprezentují návaznosti vstupů a výstupů
- **Jakobián:**

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_C}{\partial x_1} & \cdots & \frac{\partial y_C}{\partial x_D} \end{bmatrix}$$



Python kód funkce $z = (x_1 + ax_2)^2$ se zpětnou propagací

dopředný průchod

- (1) $x'_2 \leftarrow ax_2$
- (2) $y \leftarrow x_1 + x'_2$
- (3) $z \leftarrow y^2$

zpětný průchod

- (0) $\frac{\partial z}{\partial z} = 1$
- (1) $\frac{\partial z}{\partial y} \leftarrow \frac{\partial z}{\partial z} \cdot \frac{\partial z}{\partial y} = 2y$
- (2) $\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x_1} = \frac{\partial z}{\partial y}$
 $\frac{\partial z}{\partial x'_2} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x'_2} = \frac{\partial z}{\partial y}$
- (3) $\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial x'_2} \cdot \frac{\partial x'_2}{\partial x_2} = \frac{\partial z}{\partial x'_2} \cdot a$

```
def forward(x1, x2, a):
```

```
    # (1) forward
```

```
    x2_ = a * x2
```

```
    # (2) forward
```

```
    y = x1 + x2_
```

```
    # (3) forward
```

```
    z = y * y
```

```
    cache = x2, a, y
```

```
    return z, cache
```

```
>>> x1, x2, a = 1, 2, 3
```

```
>>> z, cache = forward(x1, x2, a)
```

```
>>> dx1, dx2, da = backward(1.0, cache)
```

```
>>> dx1, dx2, da
```

```
(14.0, 42.0, 28.0)
```

```
def backward(dz, cache):
```

```
    x2, a, y = cache
```

```
    # (3) backward
```

```
    dy = dz * 2 * y
```

```
    # (2) backward
```

```
    dx1 = dy
```

```
    dx2_ = dy
```

```
    # (1) backward
```

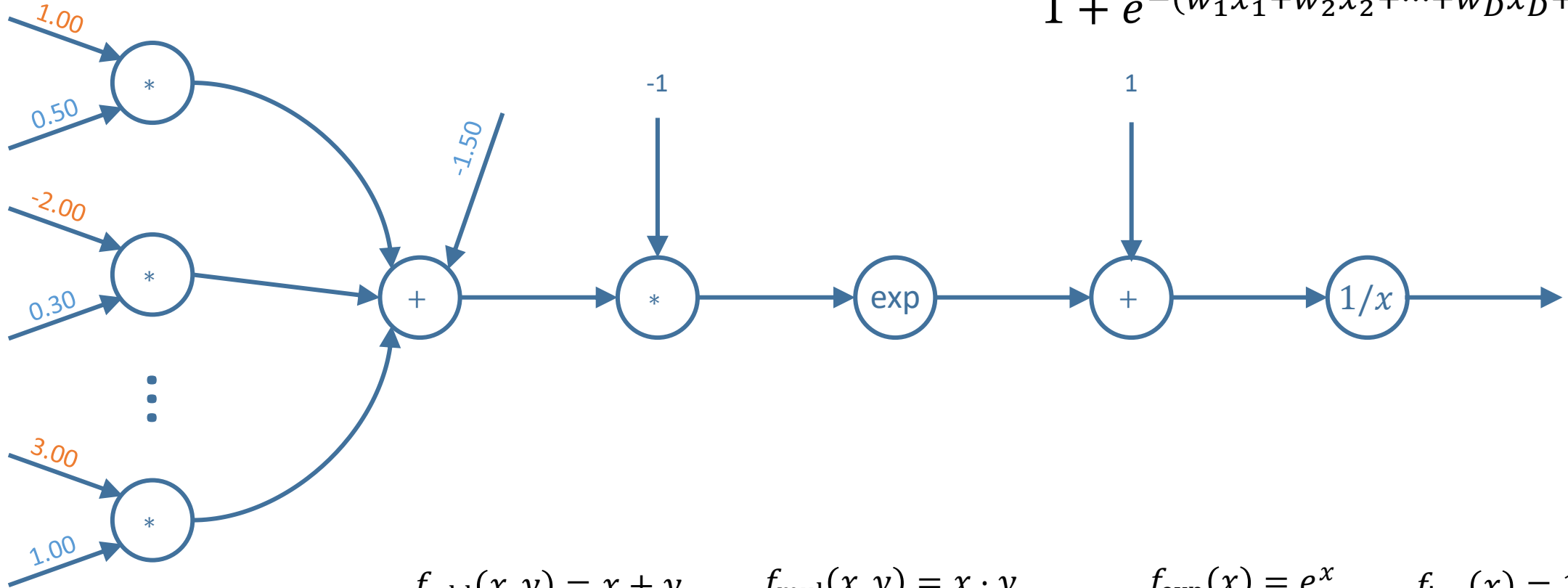
```
    dx2 = dx2_ * a
```

```
    da = dx2_ * x2
```

```
    return dx1, dx2, da
```

Binární logistická regrese se sigmoidem jako graf

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



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

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

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

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

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

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

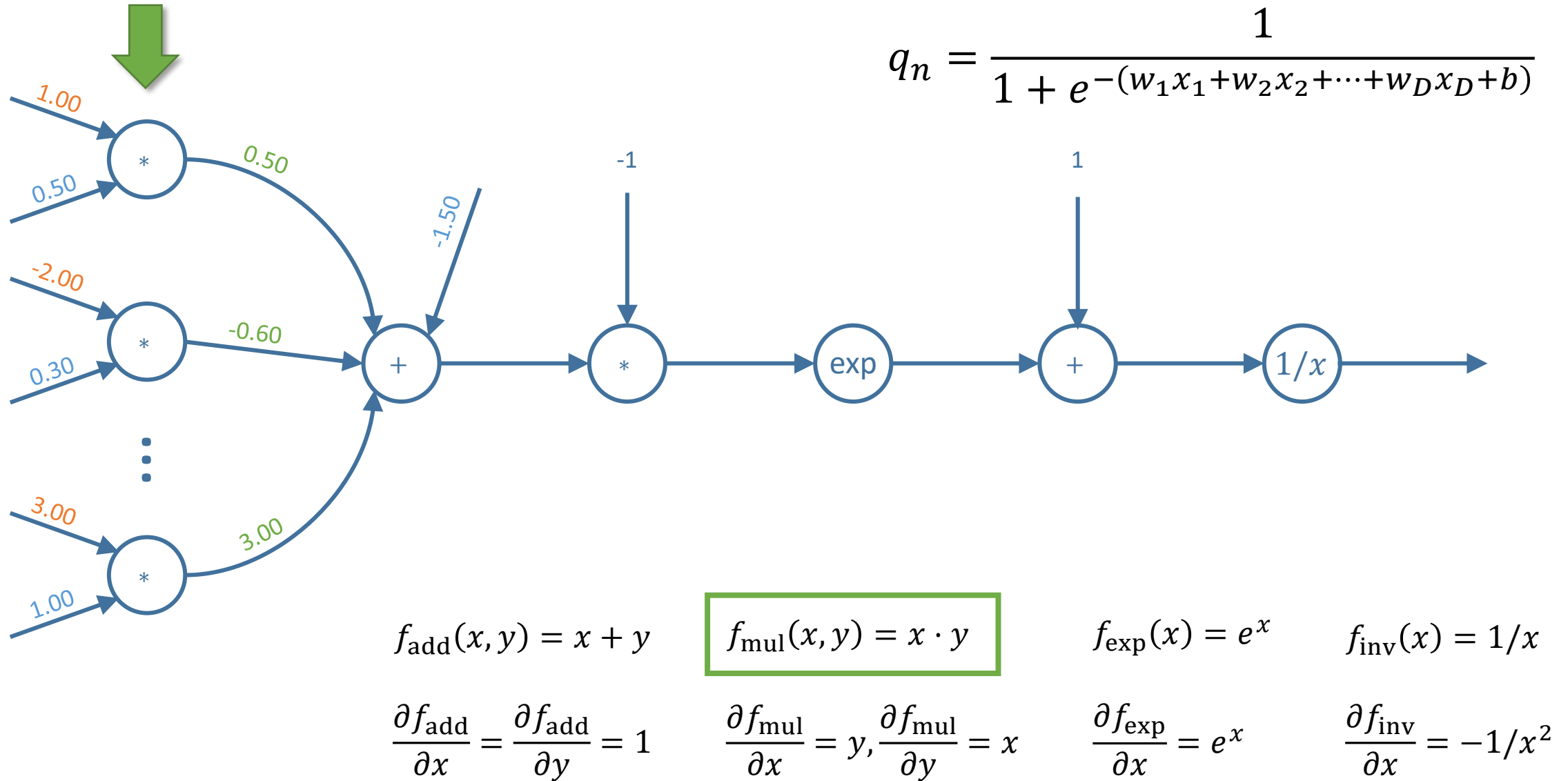
$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1$$

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

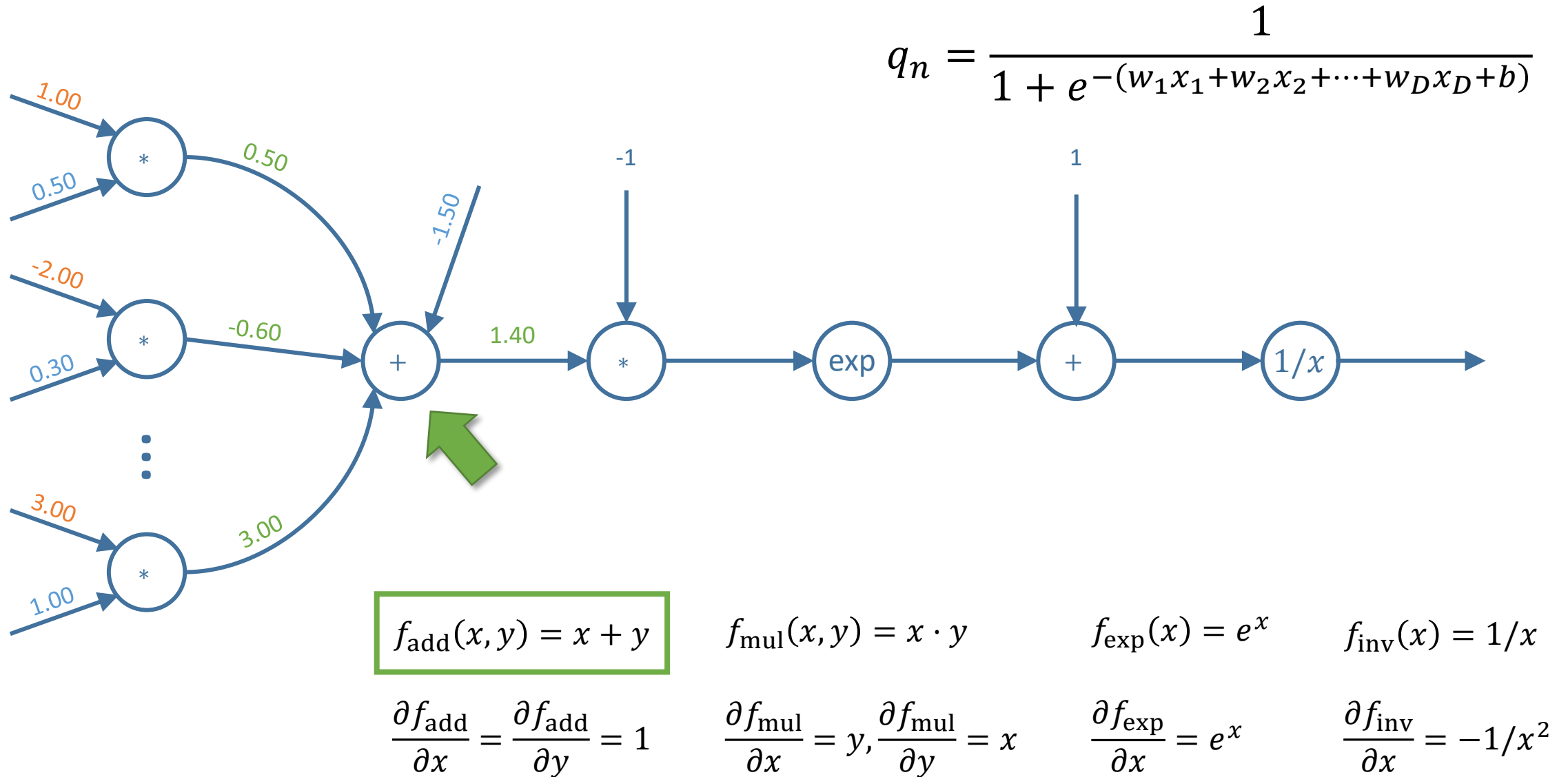
$$\frac{\partial f_{\text{exp}}}{\partial x} = e^x$$

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

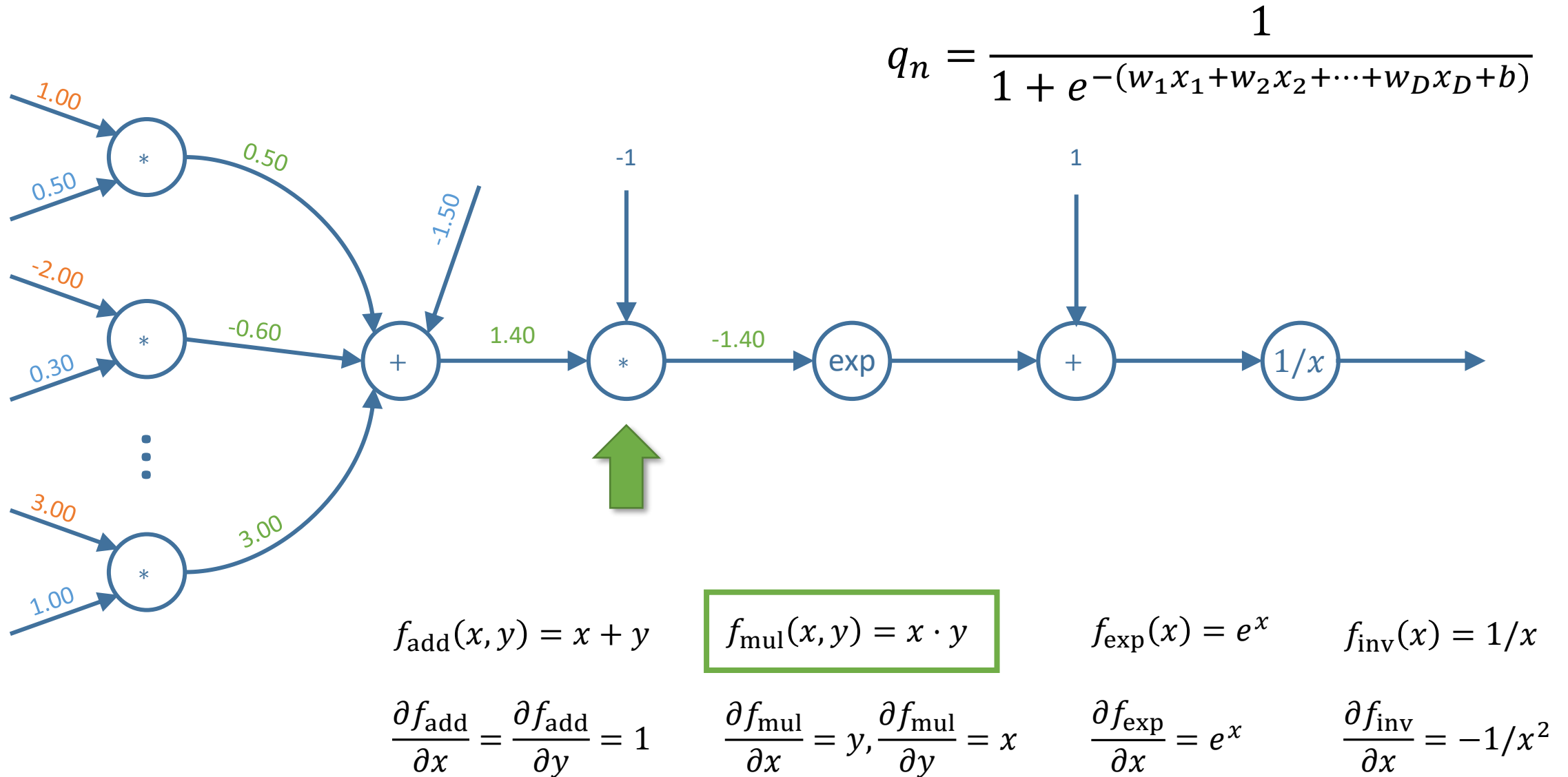
Binární logistická regrese se sigmoidem jako graf



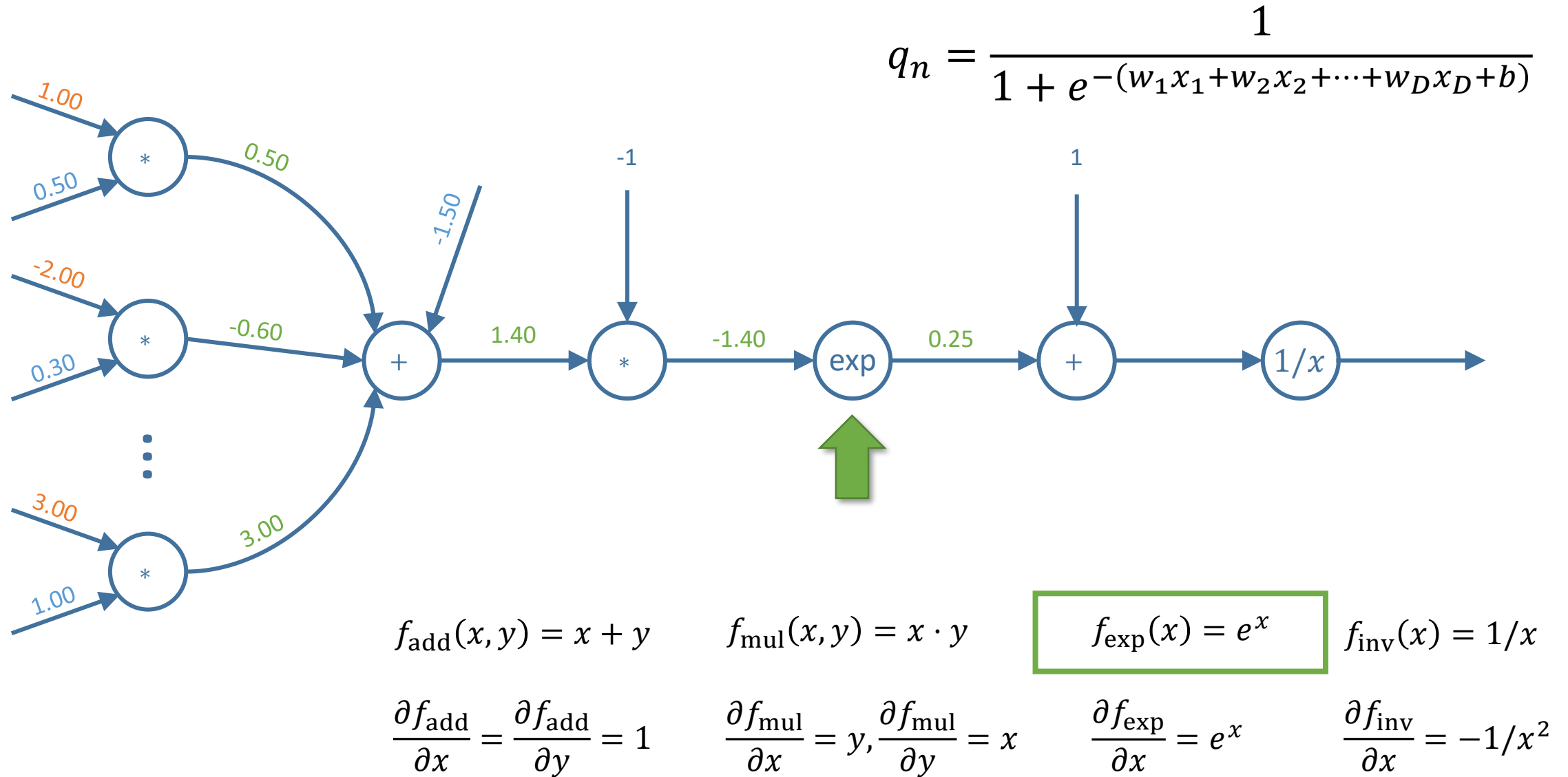
Binární logistická regrese se sigmoidem jako graf



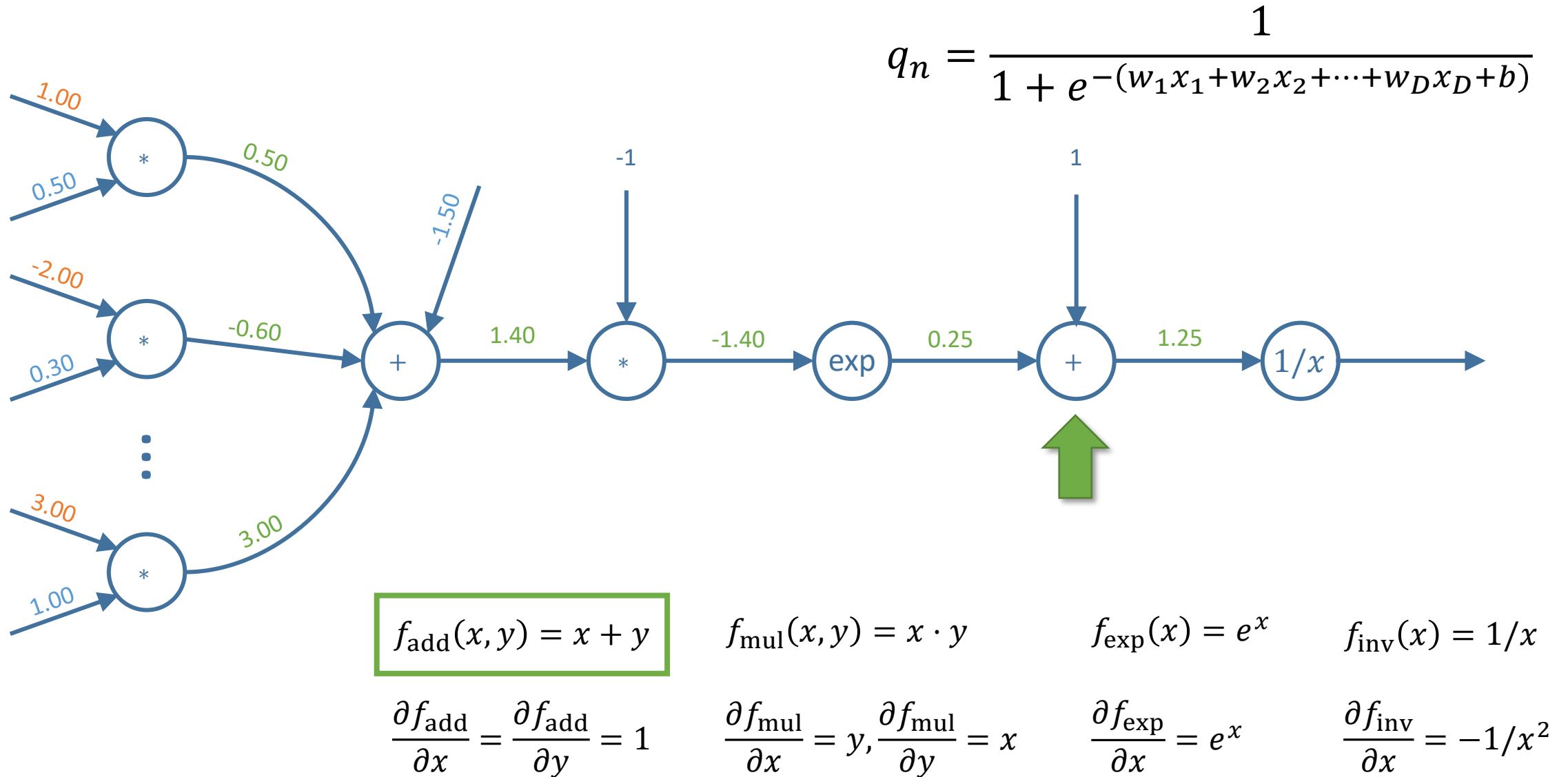
Binární logistická regrese se sigmoidem jako graf



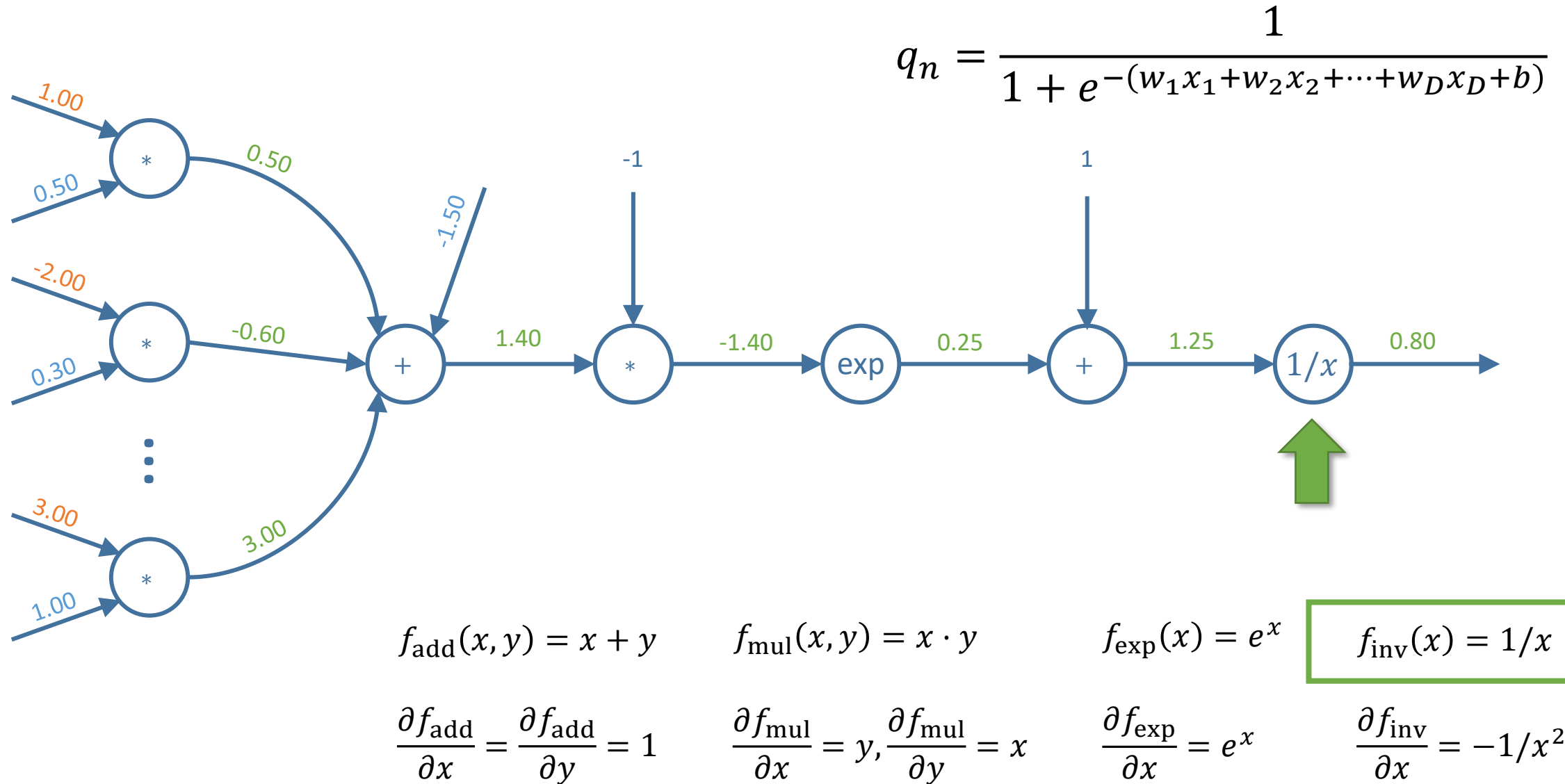
Binární logistická regrese se sigmoidem jako graf



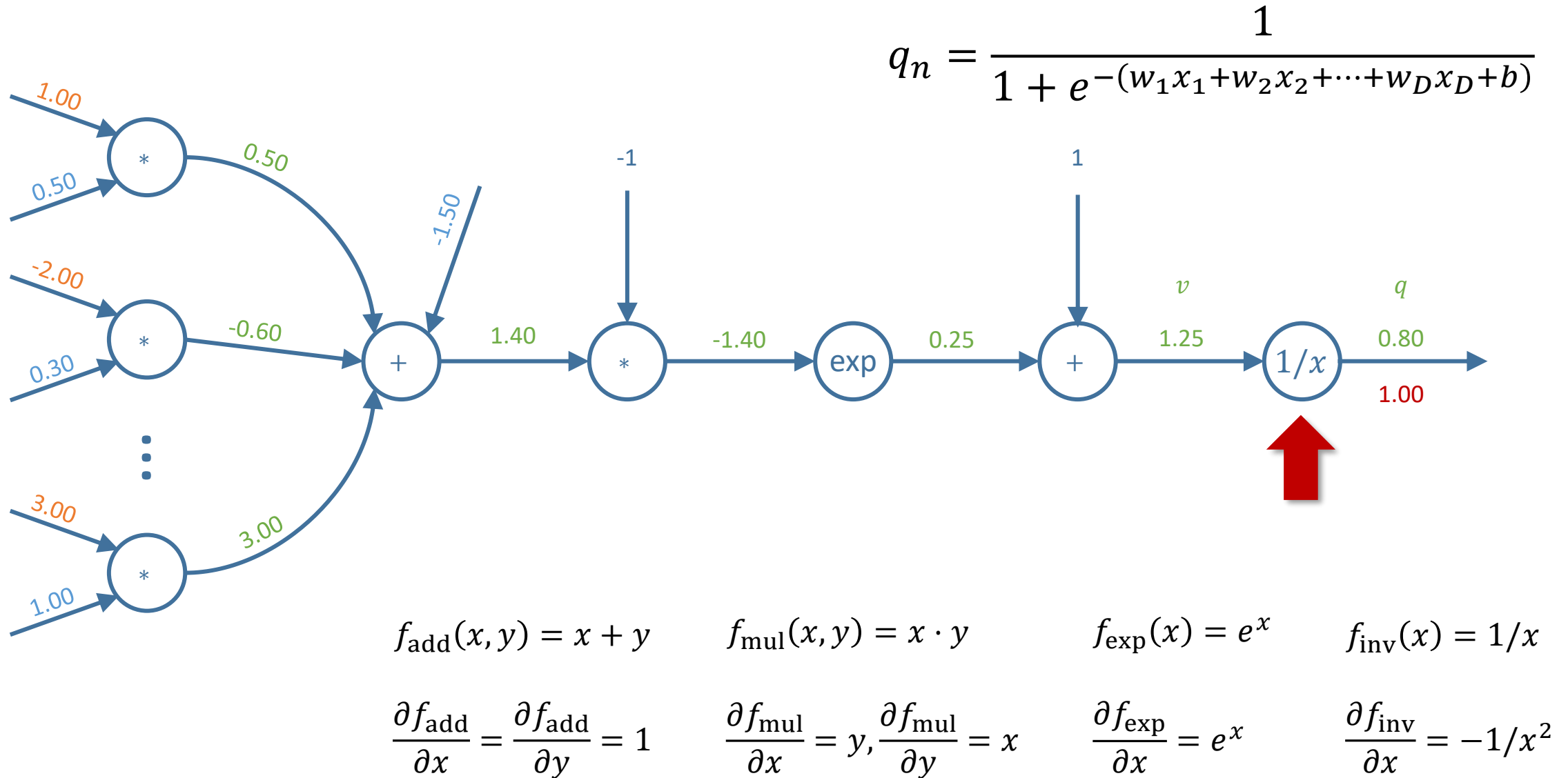
Binární logistická regrese se sigmoidem jako graf



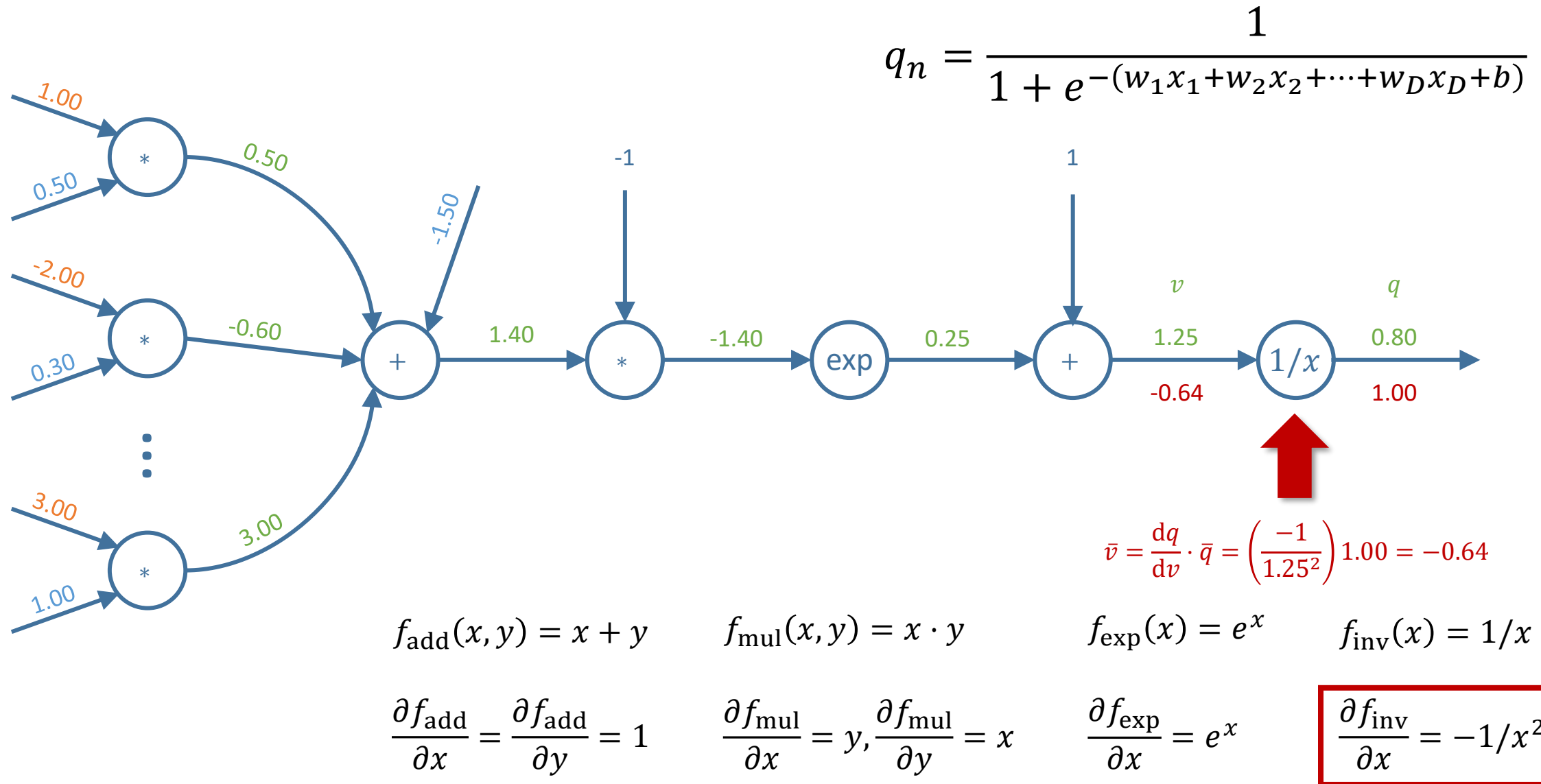
Binární logistická regrese se sigmoidem jako graf



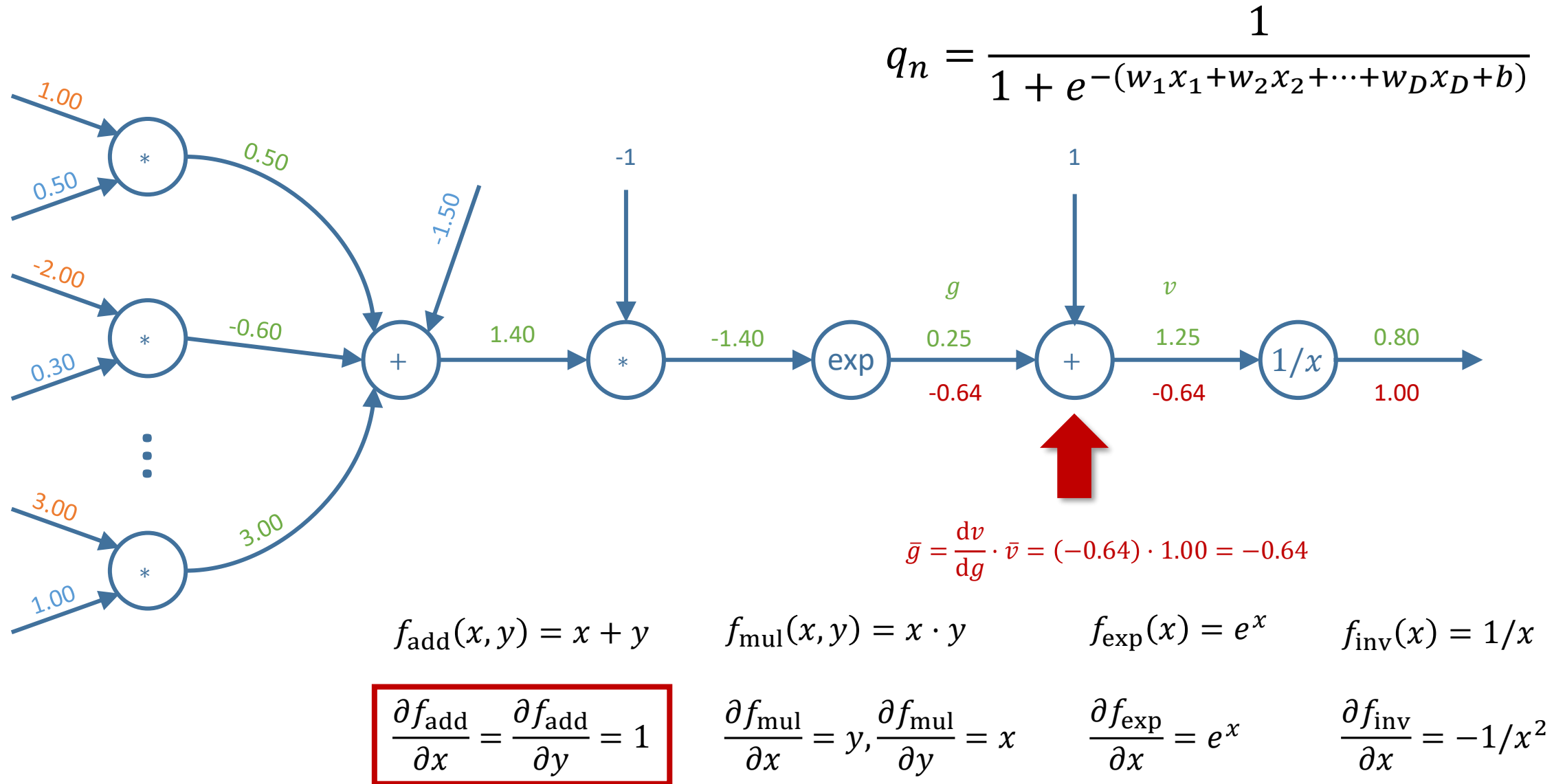
Binární logistická regrese se sigmoidem jako graf



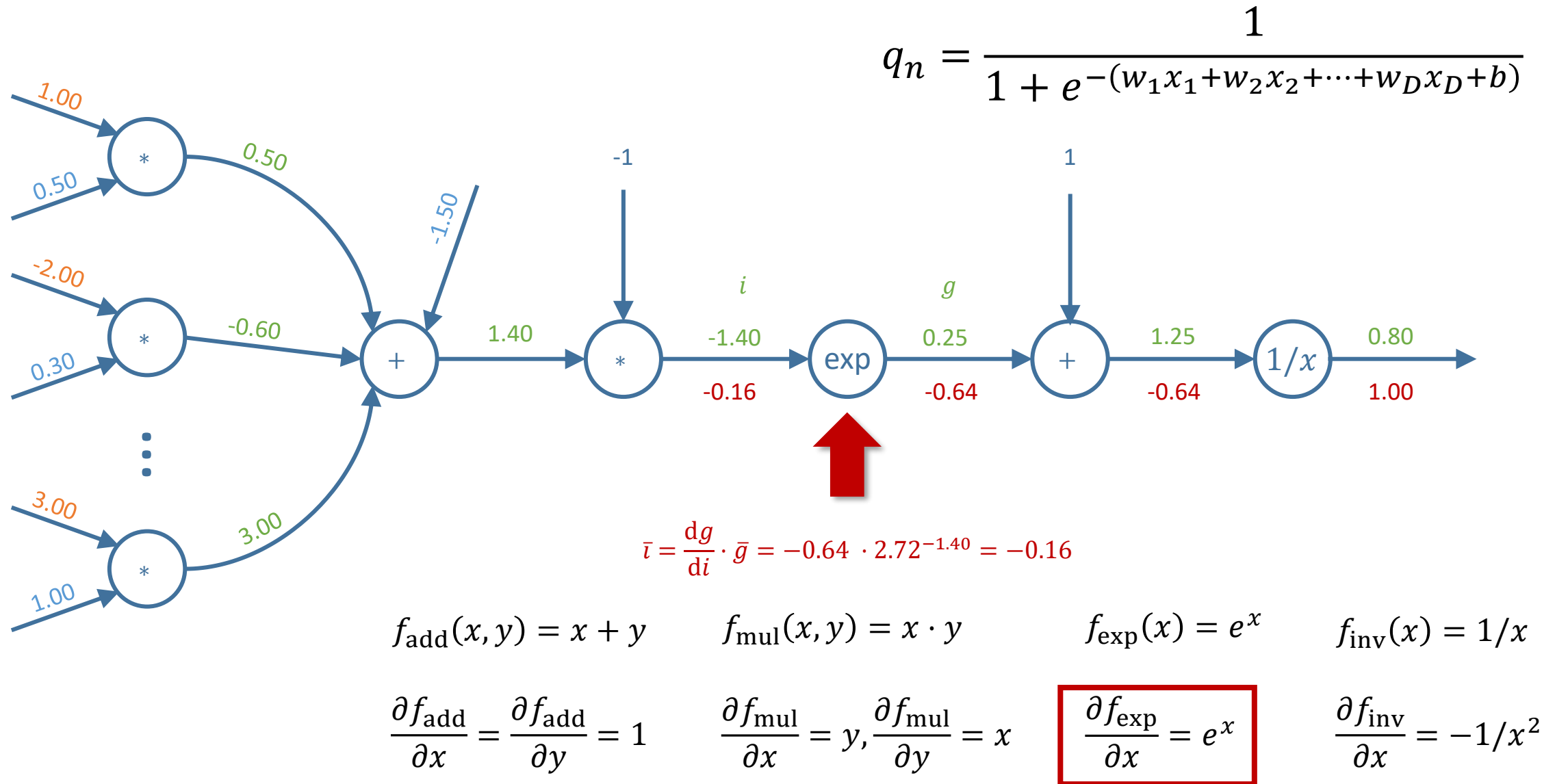
Binární logistická regrese se sigmoidem jako graf



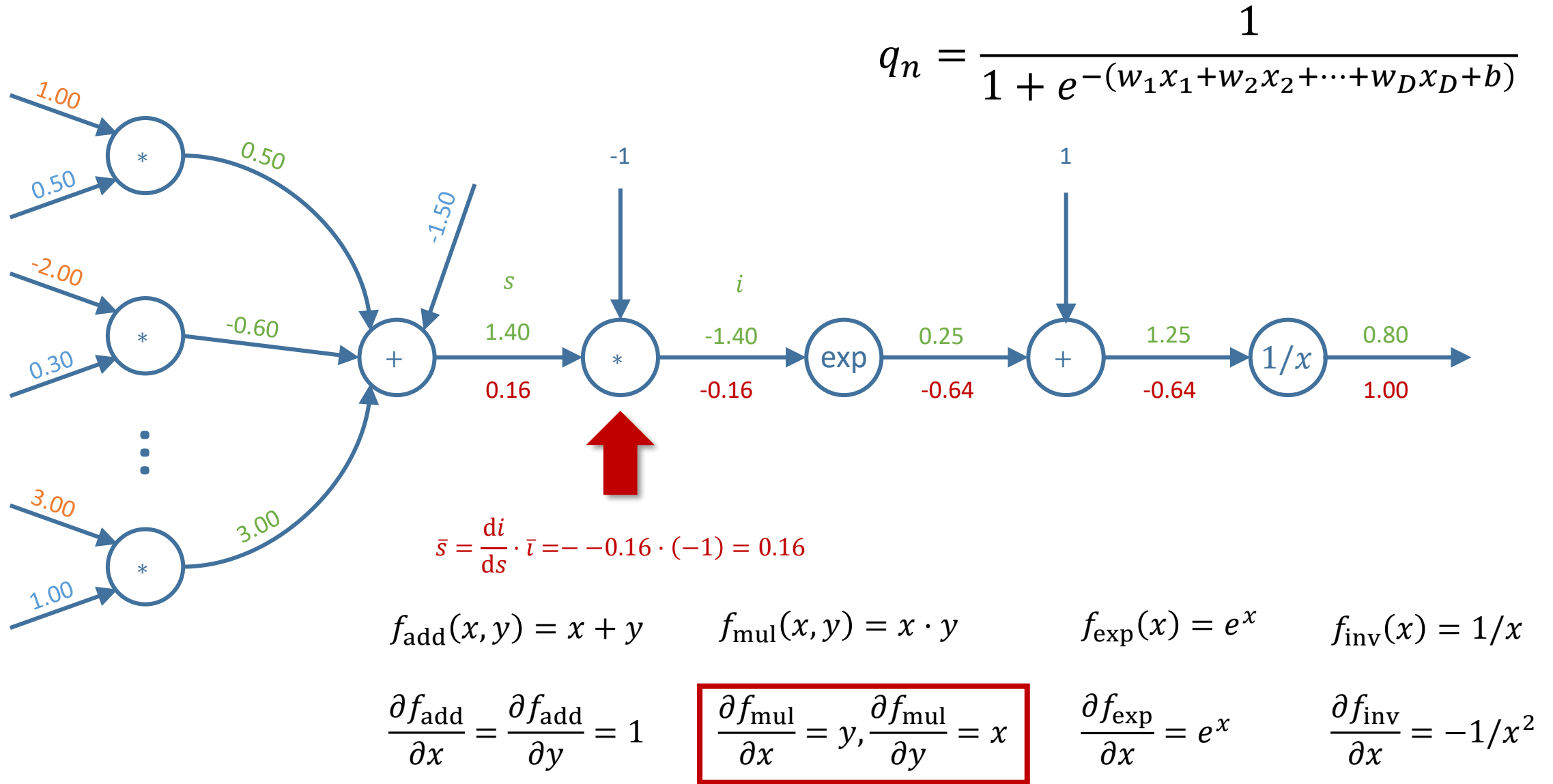
Binární logistická regrese se sigmoidem jako graf



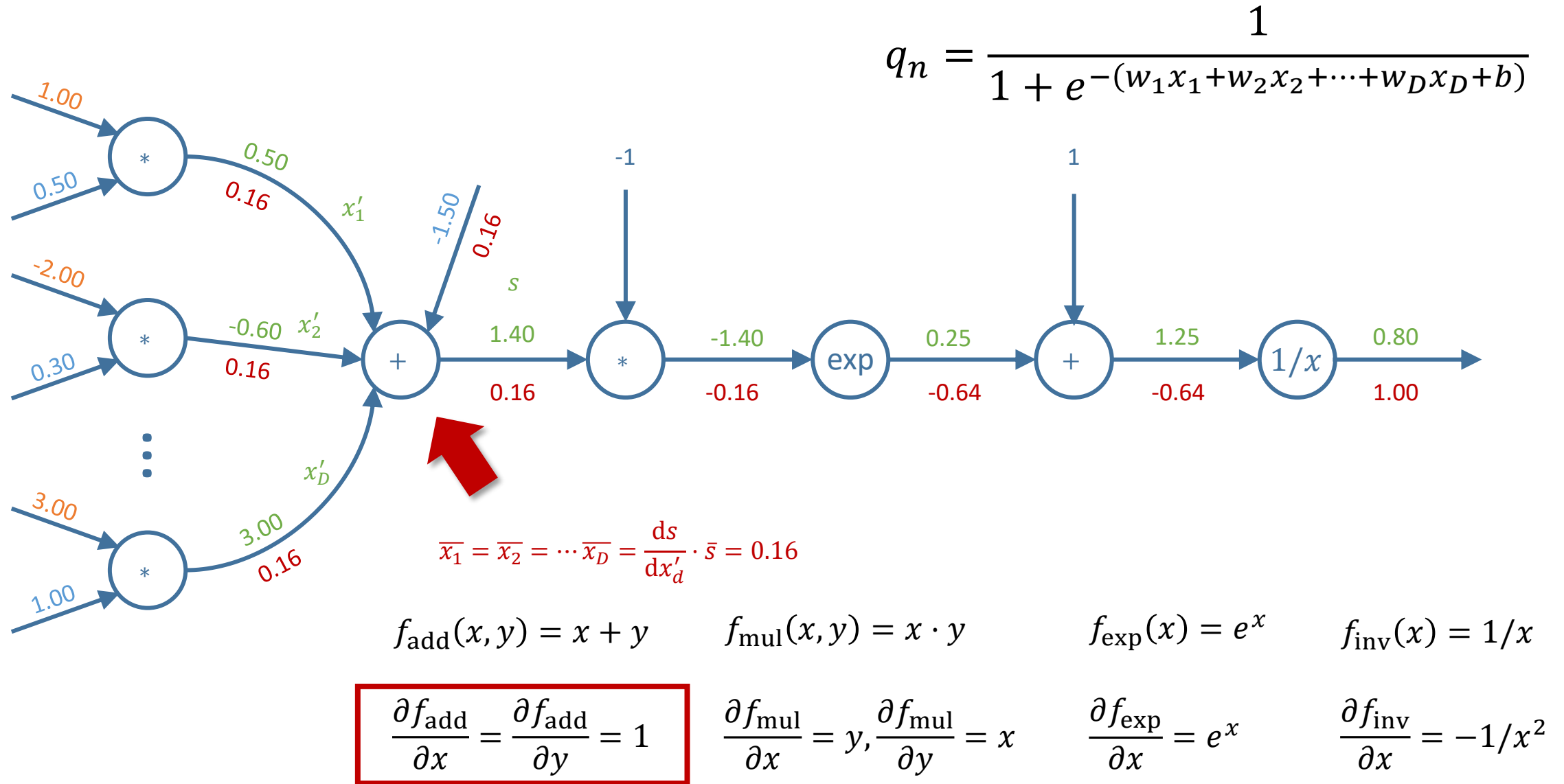
Binární logistická regrese se sigmoidem jako graf



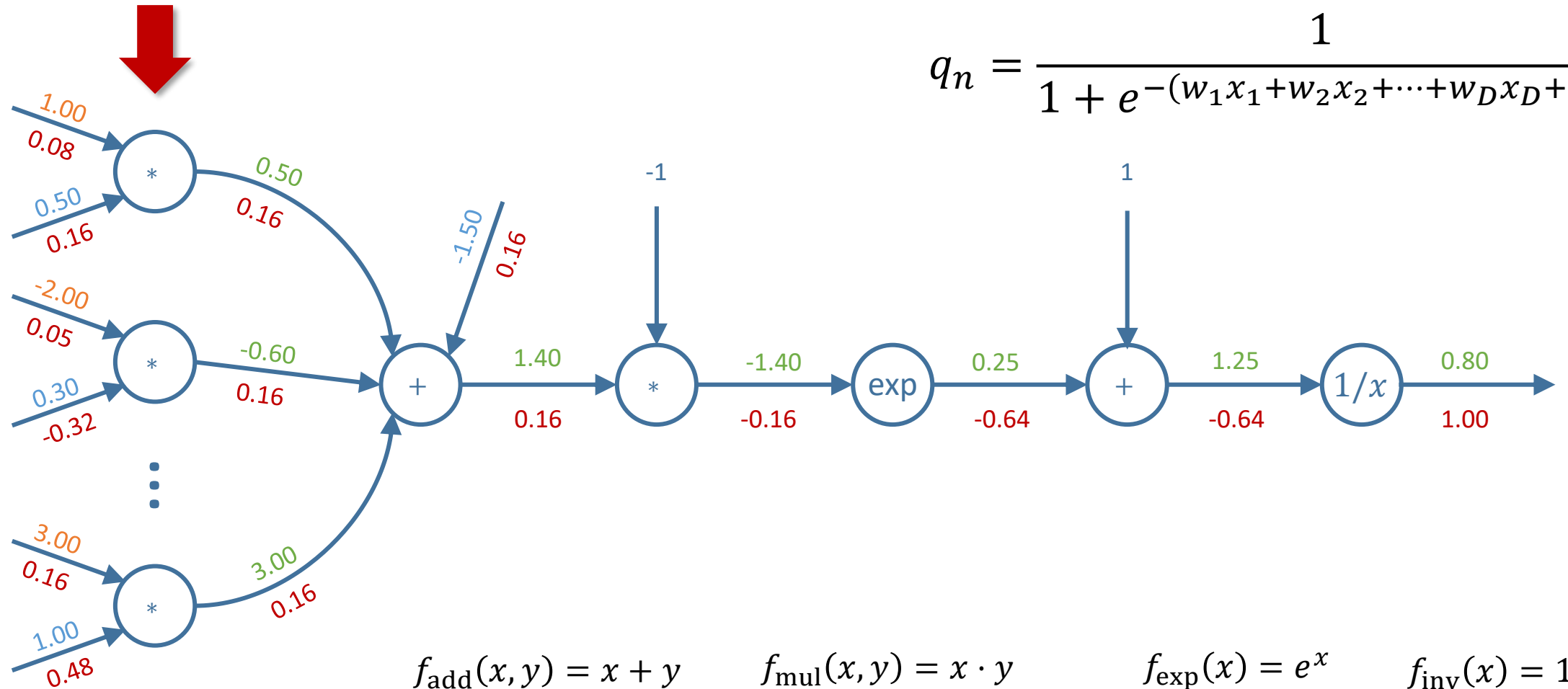
Binární logistická regrese se sigmoidem jako graf



Binární logistická regrese se sigmoidem jako graf



Binární logistická regrese se sigmoidem jako graf



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

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

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

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

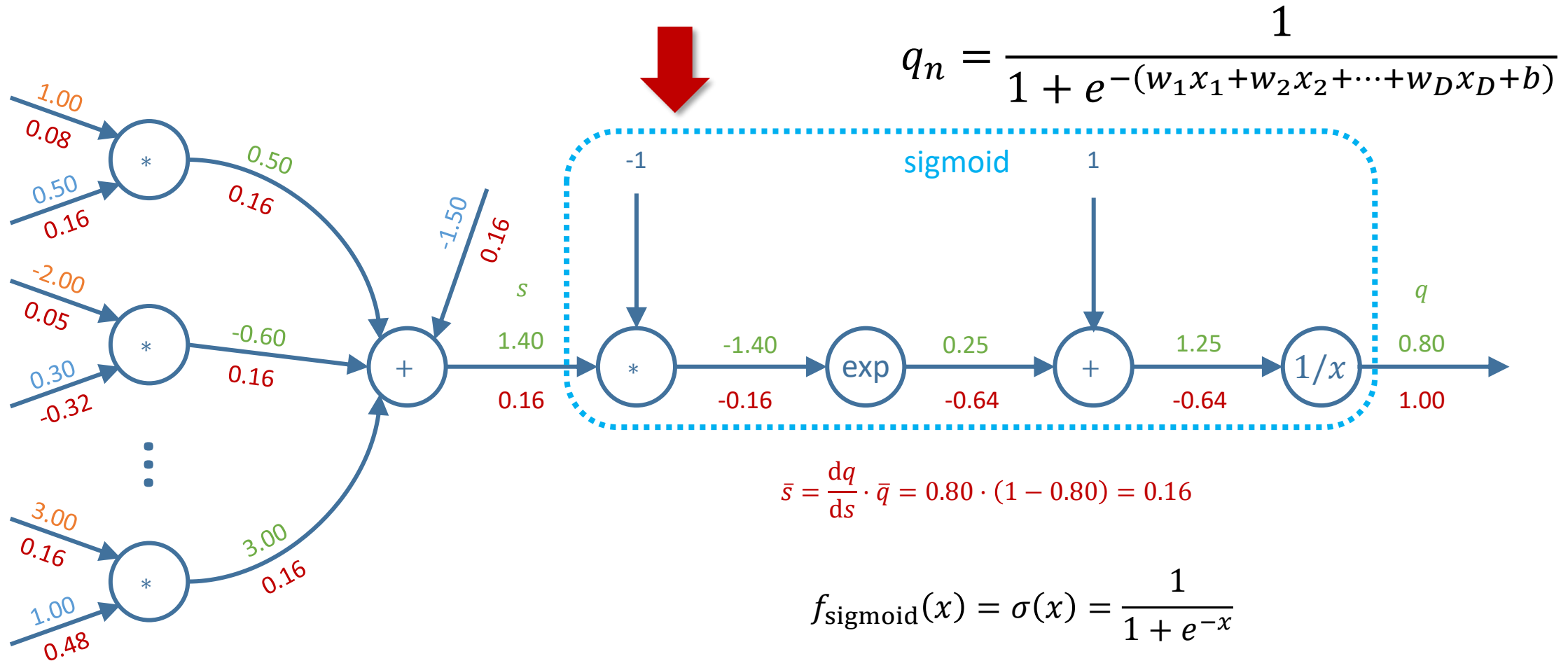
$$\frac{\partial f_{\text{add}}}{\partial x} = \frac{\partial f_{\text{add}}}{\partial y} = 1$$

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

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

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

Binární logistická regrese se sigmoidem jako graf



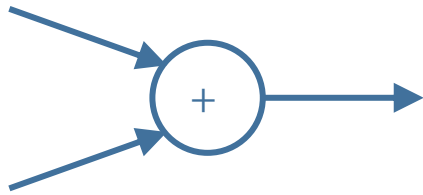
$$\bar{s} = \frac{dq}{ds} \cdot \bar{q} = 0.80 \cdot (1 - 0.80) = 0.16$$

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

$$\frac{\partial f_{\text{sigmoid}}}{\partial x} = \sigma(x)(1 - \sigma(x))$$

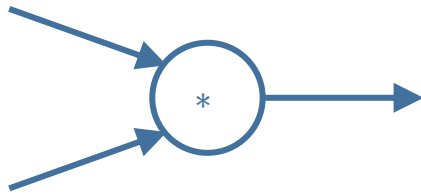
Druhy operací

- Operace v grafu se opakují → platí i pro výpočet gradientu
- V grafu se opakují pouze 4 typy **vrstev**:
 1. součet
 2. násobení
 3. exponenciální funkce
 4. inverze



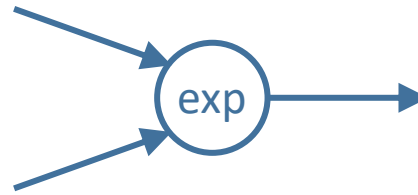
$$f_{\text{add}}(x, y) = x + y$$

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



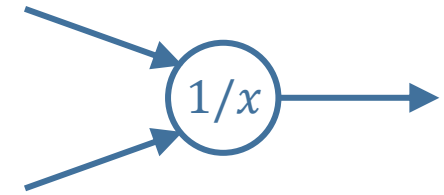
$$f_{\text{mul}}(x, y) = x \cdot y$$

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



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

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



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

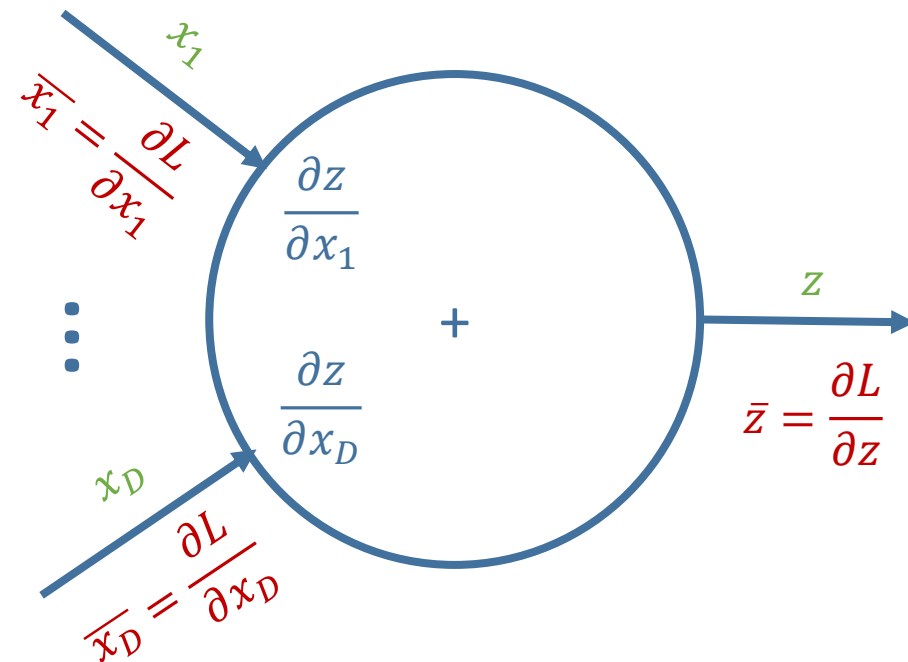
$$\frac{\partial f_{\text{inv}}}{\partial x} = -1/x^2$$

Sčítání

```
class AddNode(object):  
  
    def forward(x_vec):  
        # cache  
        self.dim = x_vec.shape[0]  
        z = np.sum(x_vec)  
        return z  
  
    def backward(dz):  
        dx_vec = dz * np.ones(self.dim)  
        return dx_vec
```

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

$$\overline{x_d} = \bar{z}, \quad d = 1, \dots, D$$

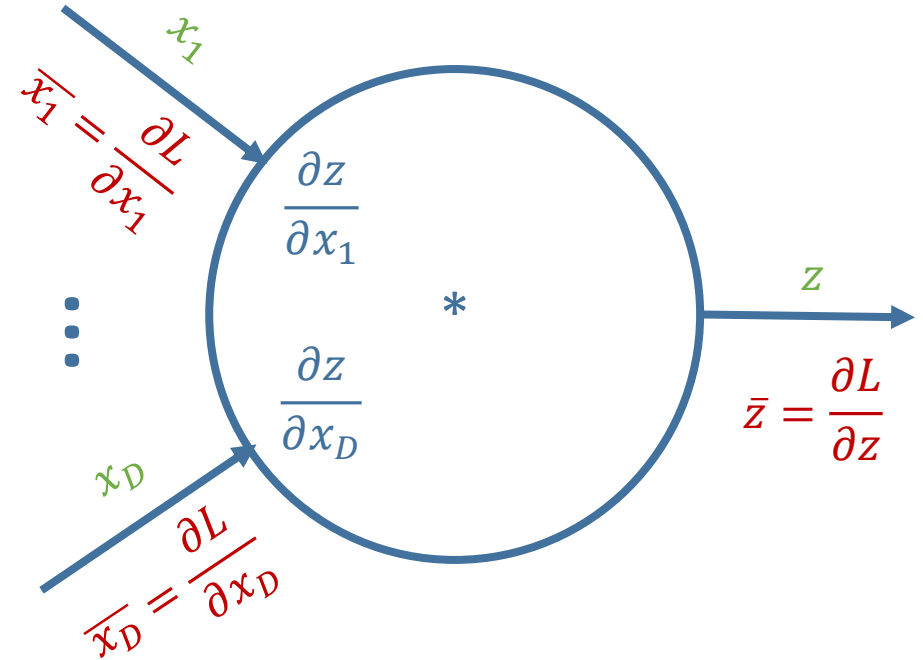


forward: $z = \sum_{d=1}^D x_d$

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

Násobení

```
class MultiplyNode(object):  
  
    def forward(x_vec):  
        # cache  
        self.x = x_vec  
        self.z = np.prod(self.x)  
        return self.z  
  
    def backward(dz):  
        # dx1 = (x1 * x2 * ... xD) / x1  
        dx_vec = dz * self.z / self.x  
        return dx_vec
```



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

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

Výpočetní graf pro zabalení funkcí do jednoho bloku

```
class ComputationalGraph(object):
    def add_node(node, *parents):
        self.nodes.append(node)
        self.parents[node] = parents

    def forward(*inputs):
        for node in topologically_sorted(self.nodes):
            inputs = [self.outputs[p] for p in self.parents[node]]
            self.outputs[node] = node.forward(*inputs)
        return top_nodes_outputs # napr. loss (treba cross entropy)

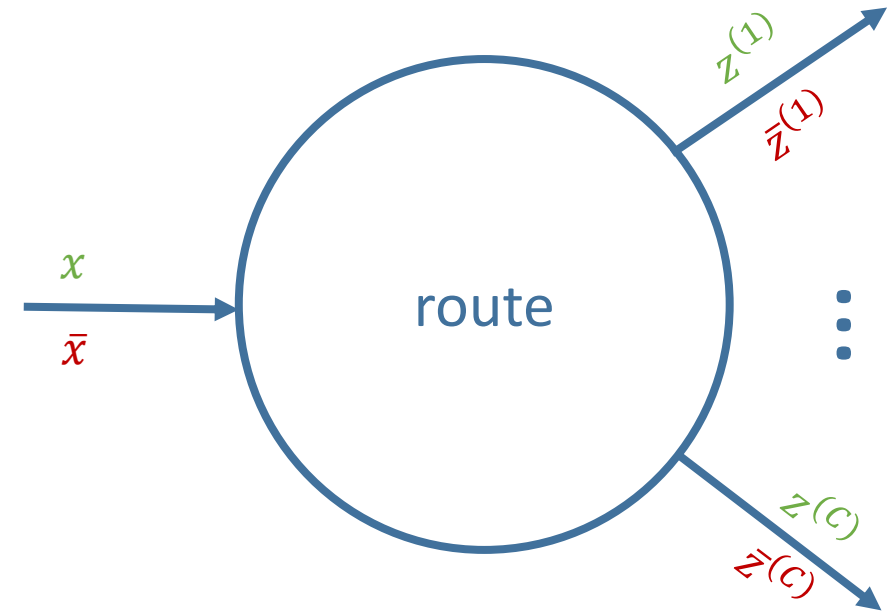
    def backward(dout):
        for node in reversed(topologically_sorted(self.nodes)):
            node_grads = node.backward(self.grads[node]) # retizkove pravidlo
            for p, g in zip(self.parents[node], node_grads):
                self.grads[p] = g
        return bottom_nodes_grads
```

graf nesmí obsahovat cykly!



Opakované použití jednoho výstupu

- Výstup vstupuje do více než jednoho bloku v další vrstvě
- Konečný výsledek (např. loss) je závislý na obou mezivýsledcích
- Všimněme si duality vůči bloku sčítání



forward: $z^{(c)} = x$

backward: $\bar{x} = \sum_{c=1}^c \bar{z}^{(c)}$

Příklad: “lineární” vrstva, gradient na vstup

- Lineární (afinní) vrstva

$$\mathbf{s} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- Rozvinutý zápis

$$\begin{bmatrix} s_1 \\ \vdots \\ s_C \end{bmatrix} = \begin{bmatrix} w_{11} & \dots & w_{1D} \\ \vdots & \ddots & \vdots \\ w_{C1} & \dots & w_{CD} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_C \end{bmatrix}$$

- c -tý výstup

$$s_c = \sum_{d=1}^D w_{cd}x_d + b_c$$

- Gradient pro jeden vstup a výstup

$$\frac{\partial s_c}{\partial x_d} = w_{cd}$$

- Každé x_d je ale použito C -krát: vyskytuje se ve výpočtech všech s_1, \dots, s_C , a proto

$$\bar{x}_d = \sum_{c=1}^C w_{cd}\bar{s}_c = \mathbf{w}_{:,d}\bar{\mathbf{s}}$$

kde $\mathbf{w}_{:,d}$ je d -tý sloupec \mathbf{W}

- Celý gradient vektorově:

$$\bar{\mathbf{x}} = \mathbf{W}^T \bar{\mathbf{s}}$$

Příklad: “lineární” vrstva, gradient na váhy


- Pro c -tý výstup pouze c -tý řádek – např. s_1 nezávisí na w_{21}, \dots, w_{2D}

$$s_c = \sum_{d=1}^D w_{cd} x_d + b_c \quad \Rightarrow \quad \frac{\partial s_c}{\partial w_{id}} = \begin{cases} x_d & \text{když } i = c \\ 0 & \text{když } i \neq c \end{cases}$$

- Celkový gradient tedy je

$$\overline{w_{cd}} = \sum_{i=1}^C \overline{s_i} \cdot \frac{\partial s_i}{\partial w_{cd}} = \overline{s_c} \cdot x_d$$

\mathbf{x} co bylo na vstupu
při dopředném
průchodu → cache!



- Vektorově

$$\overline{\mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \overline{s_1} \\ \vdots \\ \overline{s_C} \end{bmatrix} \cdot [x_1, \dots, x_D] = \begin{bmatrix} \overline{s_1} \cdot x_1 & \dots & \overline{s_1} \cdot x_D \\ \vdots & \ddots & \vdots \\ \overline{s_C} \cdot x_1 & \dots & \overline{s_C} \cdot x_D \end{bmatrix} = \overline{\mathbf{s}} \cdot \mathbf{x}^\top$$

Sigmoid jako vrstva

- Dopředný průchod

$$q = \sigma(x) = \frac{1}{1 + e^{-x}}$$

- Derivace funkce sigmoid

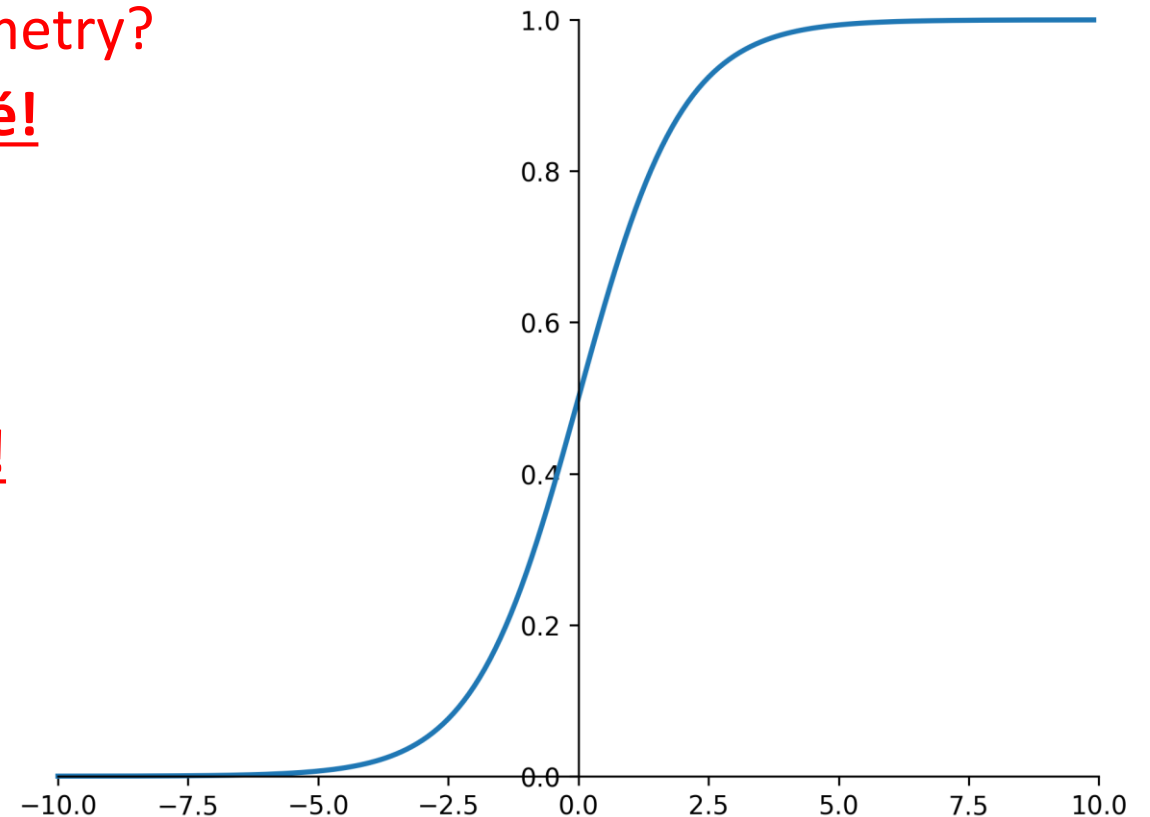
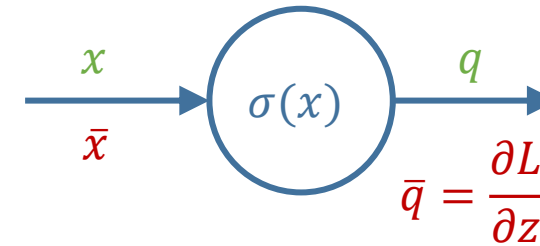
$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

- Zpětný průchod

$$\bar{x} = \bar{q} \cdot \frac{dq}{dx} = \bar{q} \cdot q \cdot (1 - q)$$

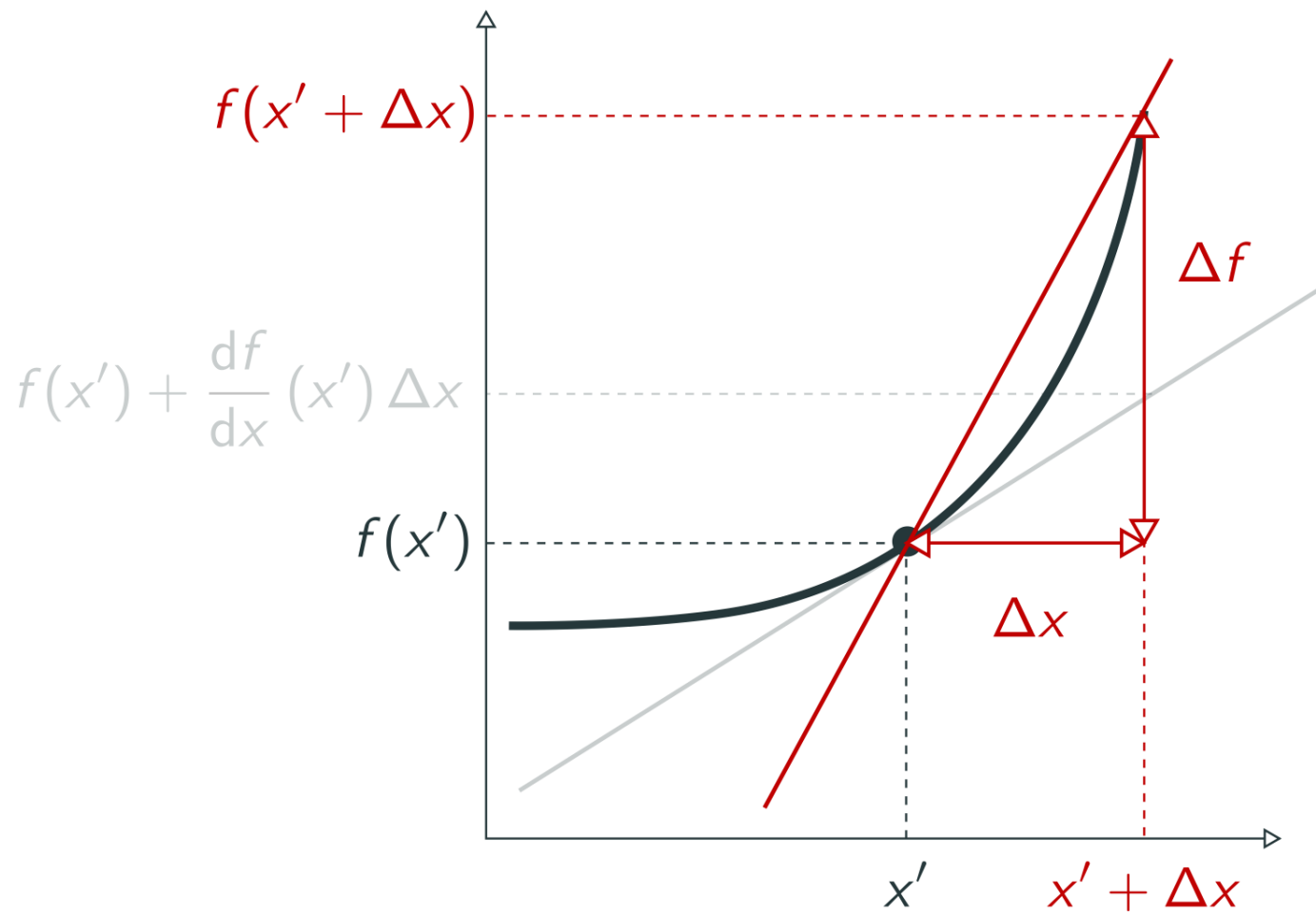
Parametry?
Žádné!

q ... cache!



Numerický gradient

Aproximace derivace dopřednou diferencí



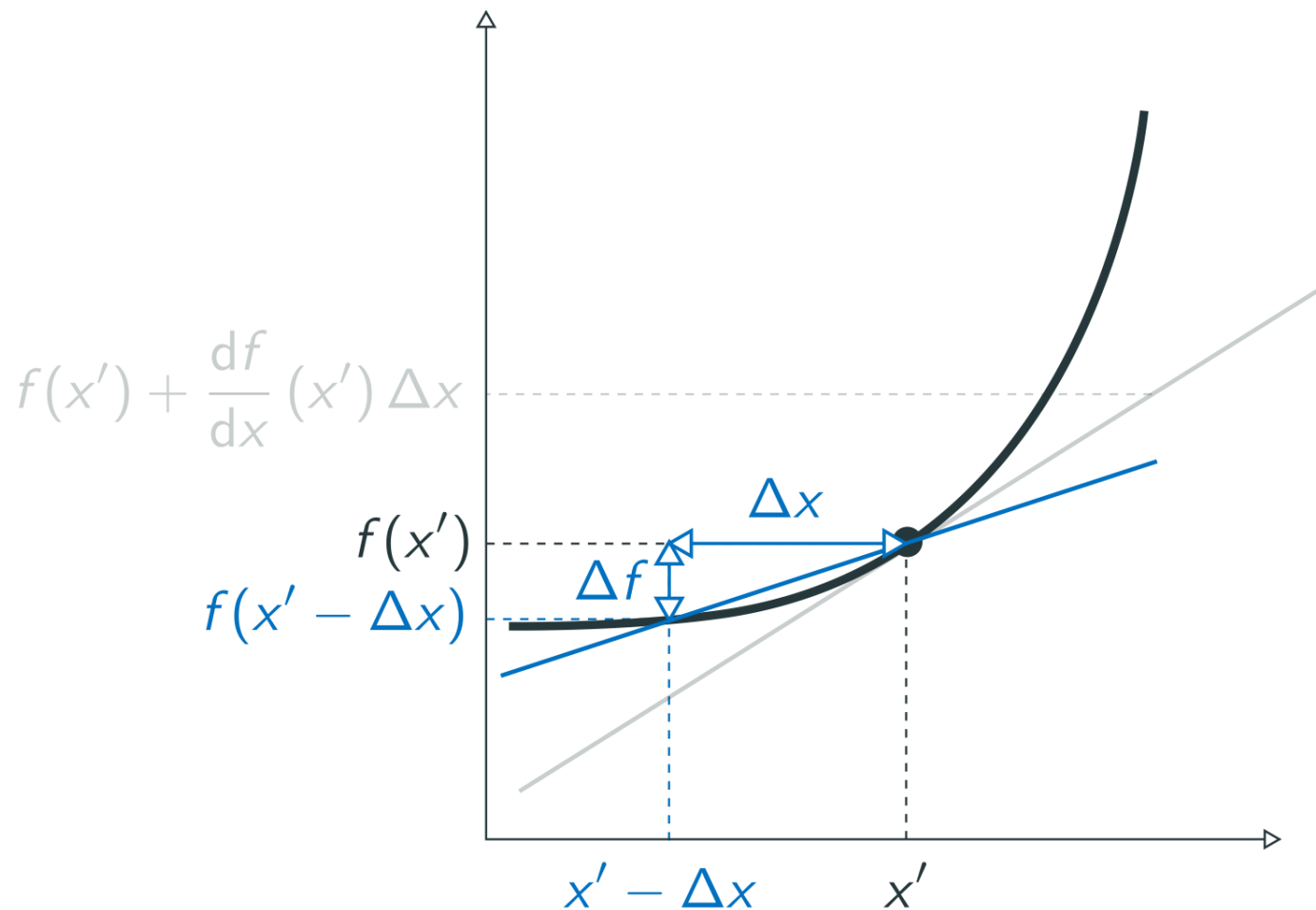
Derivace:

$$\frac{df}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta f}{\Delta x}$$

Aproximace dopředně:

$$\frac{df}{dx} \approx \frac{f(x' + \Delta x) - f(x')}{\Delta x}$$

Aproximace derivace zpětnou diferencí



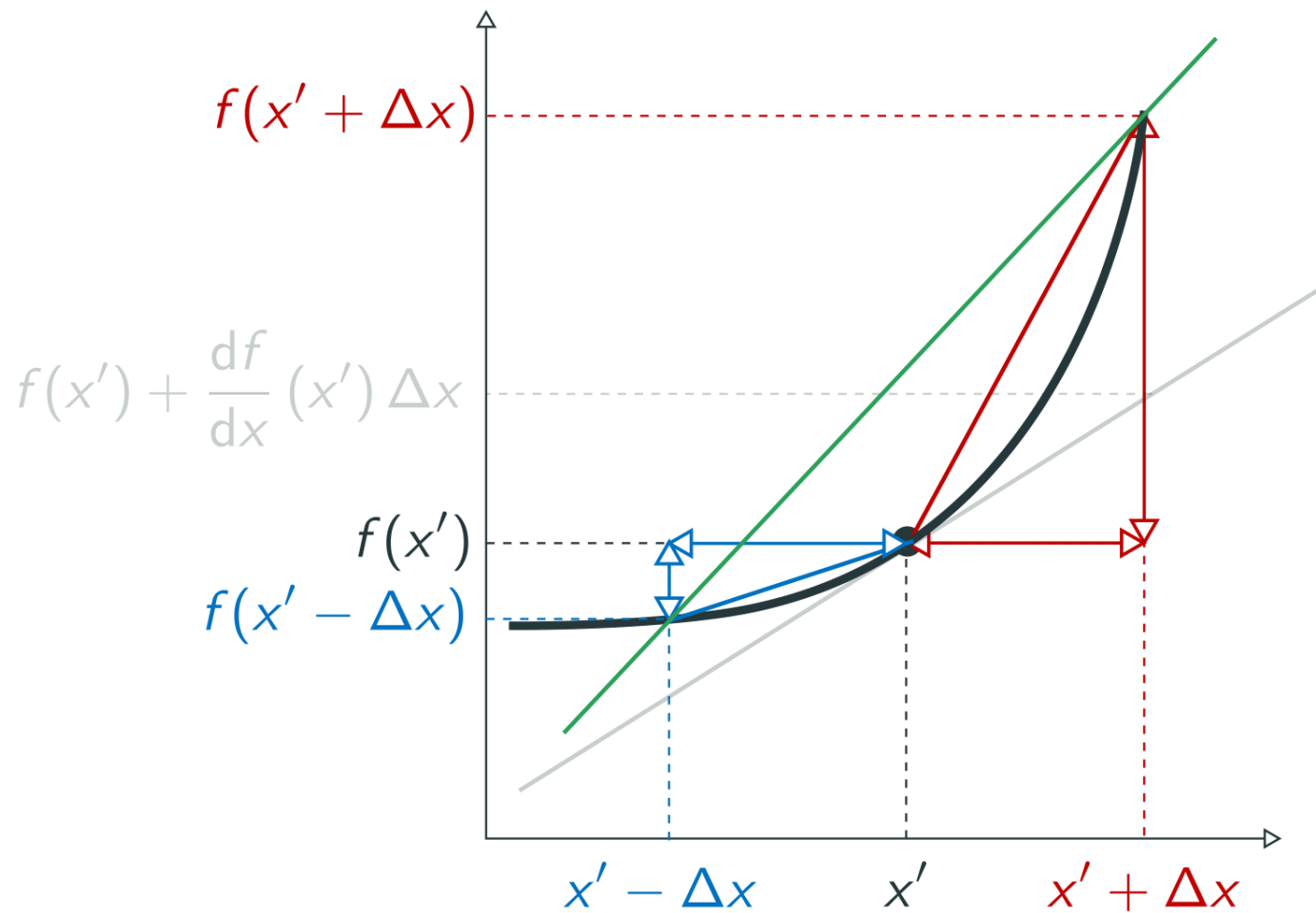
Derivace:

$$\frac{df}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta f}{\Delta x}$$

Aproximace zpětně:

$$\frac{df}{dx} \approx \frac{f(x') - f(x' + \Delta x)}{\Delta x}$$

Aproximace derivace centrální diferencí



Derivace:

$$\frac{df}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta f}{\Delta x}$$

Aproximace centrálně:

$$\frac{df}{dx} \approx \frac{f(x' + \Delta x) - f(x' - \Delta x)}{\Delta x}$$

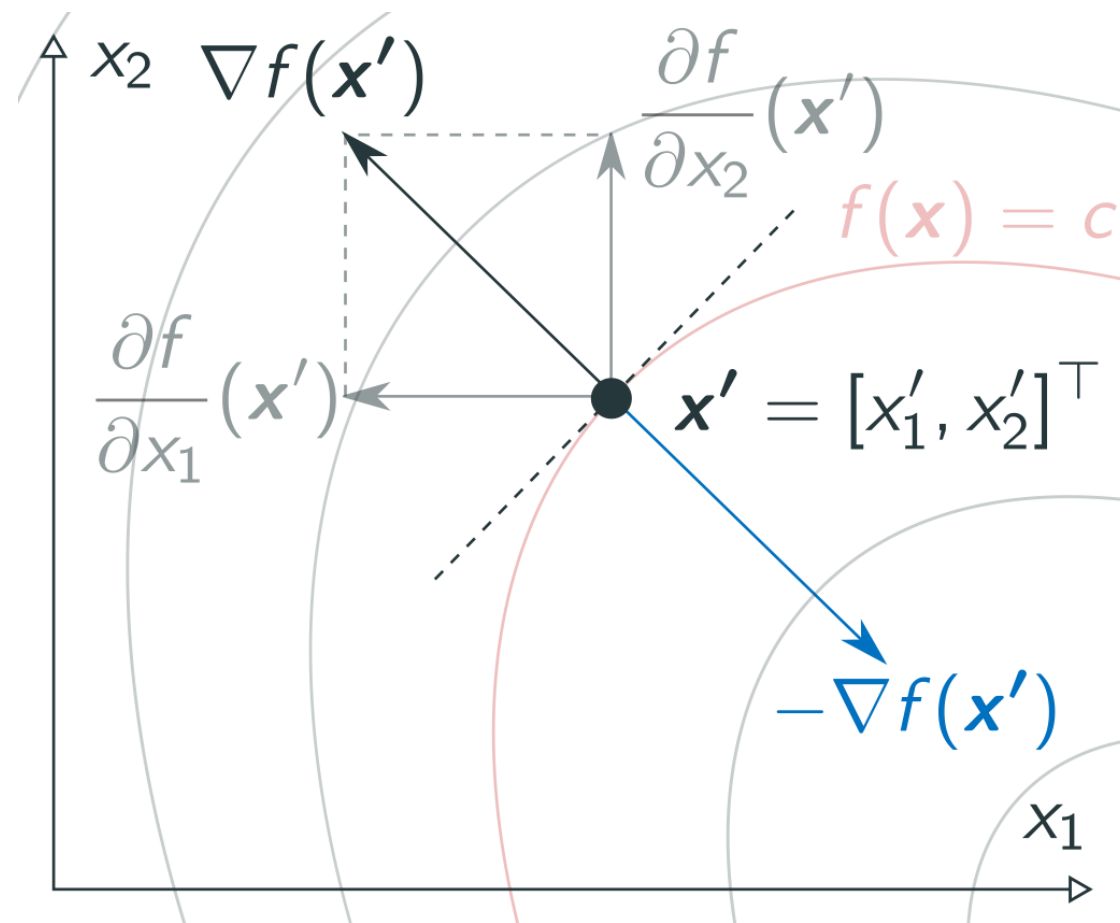
výhodou menší chyba aproximace $O(\Delta x^2)$ → častější použití než jednostranné

Aproximace ve více rozměrech

- Gradient je vektor parciálních derivací

$$\nabla f(x') = \left[\frac{\partial f}{\partial x_1}(x'_1), \dots, \frac{\partial f}{\partial x_D}(x'_D) \right]^\top$$

- Aproximace tedy nutné provést **pro každou** proměnnou funkce f



Příklad aproximace derivace numerickou diferencí

$$f(\mathbf{x}) = \sum_{d=1}^D x_d^2 \quad \mathbf{x} = [1.0, 0.8, 1.3]^\top$$

numerický odhad:

$$\widehat{\nabla f(\mathbf{x})} = \begin{bmatrix} \frac{f(x_1 + \Delta x, x_2, x_3) - f(x_1, x_2, x_3)}{\Delta x} \\ \frac{f(x_1, x_2 + \Delta x, x_3) - f(x_1, x_2, x_3)}{\Delta x} \\ \frac{f(x_1, x_2, x_3 + \Delta x) - f(x_1, x_2, x_3)}{\Delta x} \end{bmatrix} = \begin{bmatrix} \frac{f(1.0 + 0.001, 0.8, 1.3) - f(1.0, 0.8, 1.3)}{0.001} \\ \frac{f(1.0, 0.8 + 0.001, 1.3) - f(1.0, 0.8, 1.3)}{0.001} \\ \frac{f(1.0, 0.8, 1.3 + 0.001) - f(1.0, 0.8, 1.3)}{0.001} \end{bmatrix} = \begin{bmatrix} 2.001 \\ 1.601 \\ 2.601 \end{bmatrix}$$

skutečný gradient:

$$\nabla f(\mathbf{x}) = [2x_1, 2x_2, 2x_3]^\top = [2.0, 1.6, 2.6]^\top$$

hluboké neurosítě běžně miliony parametrů → pro vyčíslení gradientu nutné vyhodnotit jejich výstup min. jednou pro každý parametr a to v každém kroku gradient descentu!

Centrální difference v numpy

```
def eval_numerical_gradient_array(f, x, df, h=1e-5):
    grad = np.zeros_like(x)
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        ix = it.multi_index

        oldval = x[ix]
        x[ix] = oldval + h
        pos = f(x).copy()
        x[ix] = oldval - h
        neg = f(x).copy()
        x[ix] = oldval

        grad[ix] = np.sum((pos - neg) * df) / (2 * h)
        it.iternext()
    return grad
```

<http://cs231n.github.io/>

Kontrola gradientu

- Numerická aproximace je sice **pomalá**, ale vždy až na toleranci **správná** → vhodná **pouze pro kontrolu** implementace backward metody

- V příkladu vyšlo

$$\widehat{\nabla f(\mathbf{x})} = [2.001, 1.601, 2.601]^T$$
$$\nabla f(\mathbf{x}) = [2.0, 1.6, 2.6]^T$$

- Relativní chyba

$$\delta = \max \frac{|\widehat{\nabla f(\mathbf{x})} - \nabla f(\mathbf{x})|}{|\widehat{\nabla f(\mathbf{x})}| + |\nabla f(\mathbf{x})|} = \max \left(\frac{0.001}{4.001}, \frac{0.001}{3.201}, \frac{0.001}{5.201} \right) = 3.124 \cdot 10^{-4}$$

- Kód v numpy z předmětu [cs231n Stanfordovy univerzity](#):

```
>>> def rel_error(x, y):
>>>     return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
>>> rel_error(np.array([2.001, 1.601, 2.601]), np.array([2.0, 1.6, 2.6]))
0.00031240237425800993
```

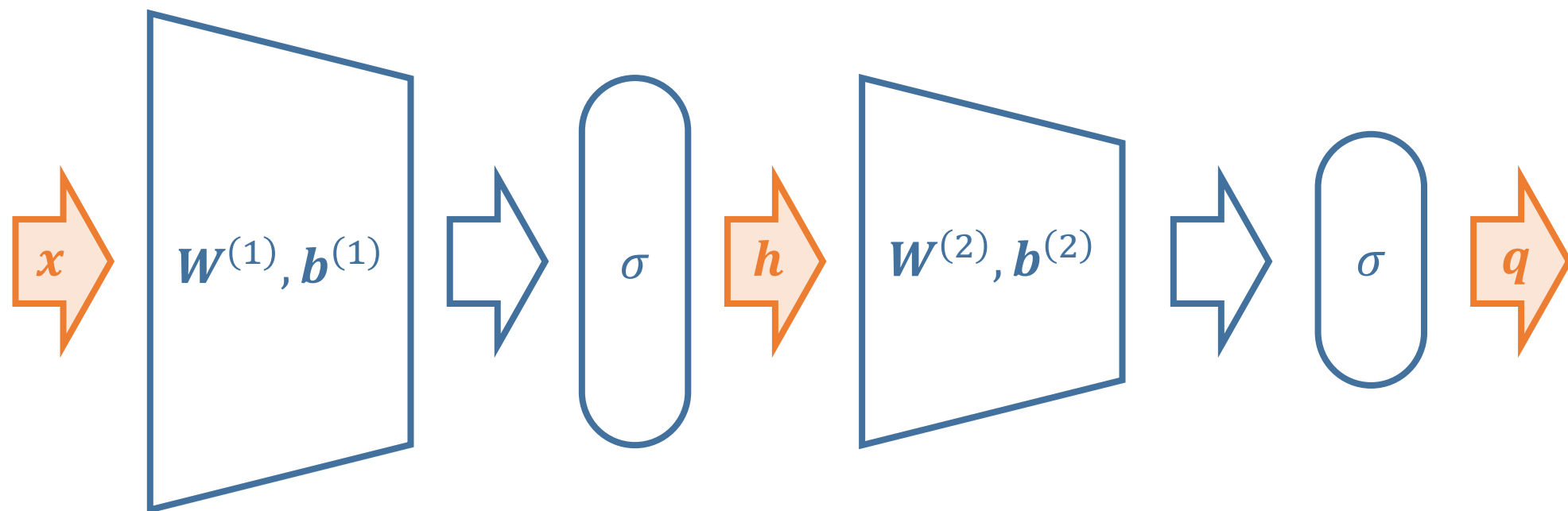
Vícevrstvé sítě

Vícevrstvý perceptron

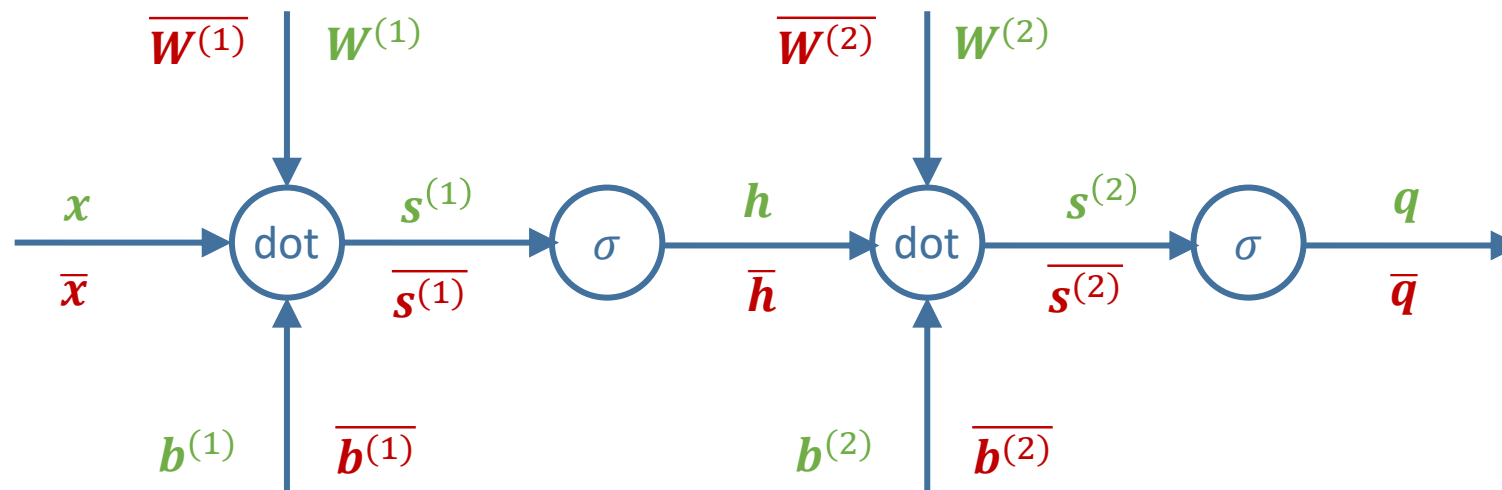
- Bloky s definovaným forward a backward chováním lze libovolně skládat za sebe
- Např. dvouvrstvý perceptron

$$q = \sigma\{W^{(2)} \sigma(W^{(1)}x + b^{(1)}) + b^{(2)}\}$$

h ... skrytá reprezentace



Vícevrstvý perceptron



forward:

$$\begin{aligned} & \xrightarrow{x} s^{(1)} \leftarrow W^{(1)}x + b^{(1)} \quad \Rightarrow \quad h \leftarrow \sigma(s^{(1)}) \quad \Rightarrow \quad s^{(2)} \leftarrow W^{(2)}h + b^{(2)} \quad \Rightarrow \quad q \leftarrow \sigma(s^{(2)}) \end{aligned}$$

backward:

$$\begin{aligned} & \bar{x} \leftarrow W^{(1)\top} \bar{s}^{(1)} \quad \leftarrow \quad \bar{s}^{(1)} \leftarrow h(1-h)\bar{h} \quad \leftarrow \quad \bar{h} \leftarrow W^{(2)\top} \bar{s}^{(2)} \quad \leftarrow \quad \bar{s}^{(2)} \leftarrow q(1-q)\bar{q} \quad \leftarrow \quad \bar{q} \\ & \bar{W}^{(1)} \leftarrow \bar{s}^{(1)} h^\top \quad \quad \quad \bar{W}^{(2)} \leftarrow \bar{s}^{(2)} h^\top \\ & \bar{b}^{(1)} \leftarrow \bar{s}^{(1)} \quad \quad \quad \bar{b}^{(2)} \leftarrow \bar{s}^{(2)} \end{aligned}$$

Dvouvrstvý perceptron v numpy na 11 řádků

```
X = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
y = np.array([[0, 1, 1, 0]]).T
syn0 = 2*np.random.random((3, 4)) - 1
syn1 = 2*np.random.random((4, 1)) - 1
for j in xrange(60000):
    l1 = 1 / (1 + np.exp(-(np.dot(X, syn0))))
    l2 = 1 / (1 + np.exp(-(np.dot(l1, syn1))))
    l2_delta = (y - l2) * (l2 * (1 - l2))
    l1_delta = l2_delta.dot(syn1.T) * (l1 * (1 - l1))
    syn1 += l1.T.dot(l2_delta)
    syn0 += X.T.dot(l1_delta)
```

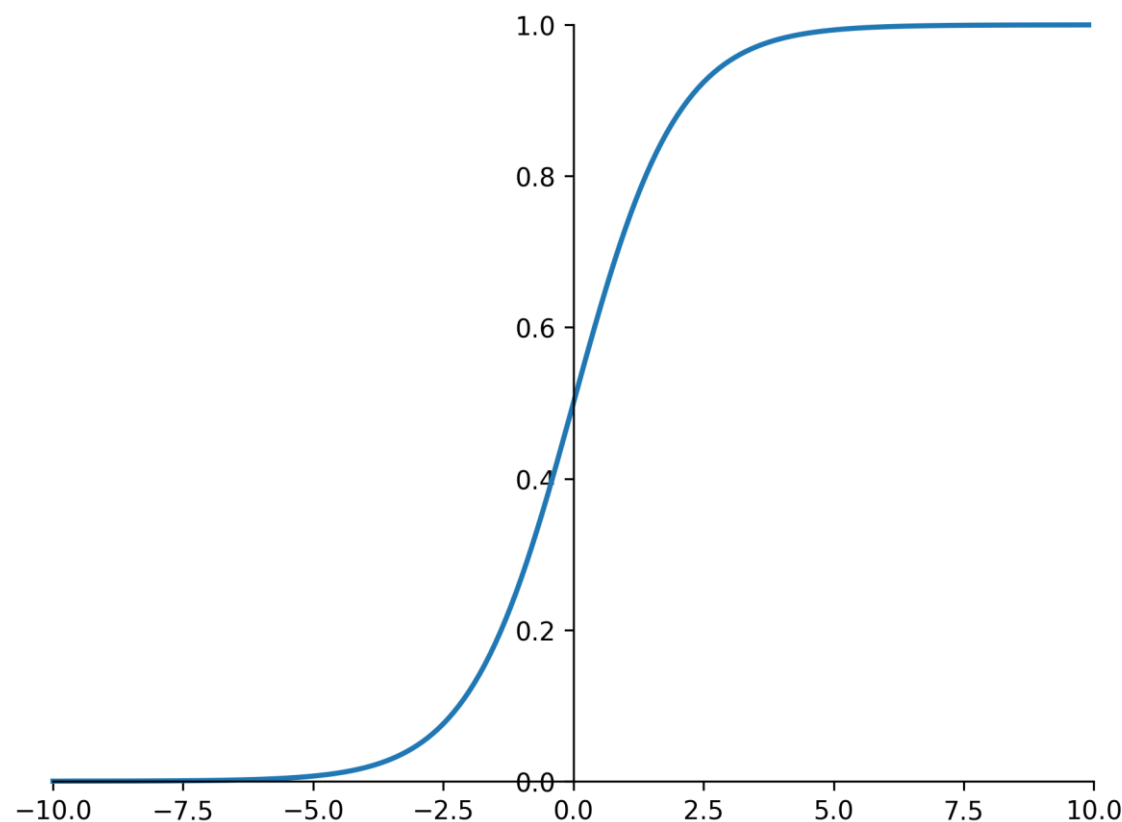
<http://iamtrask.github.io/2015/07/12/basic-python-network/>

Aktivace

Sigmoid

- Před Alexnet prakticky jediná používaná aktivace
- Převádí vstup na pravděpodobnost, tj. do intervalu $\langle 0, 1 \rangle$
- Vstup x jsou typicky skóre s z předchozí lineární vrstvy
- Problémy:
 1. "mizející" gradient
 2. pouze kladné hodnoty
 3. exp funkce zbytečně náročná na výpočet

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

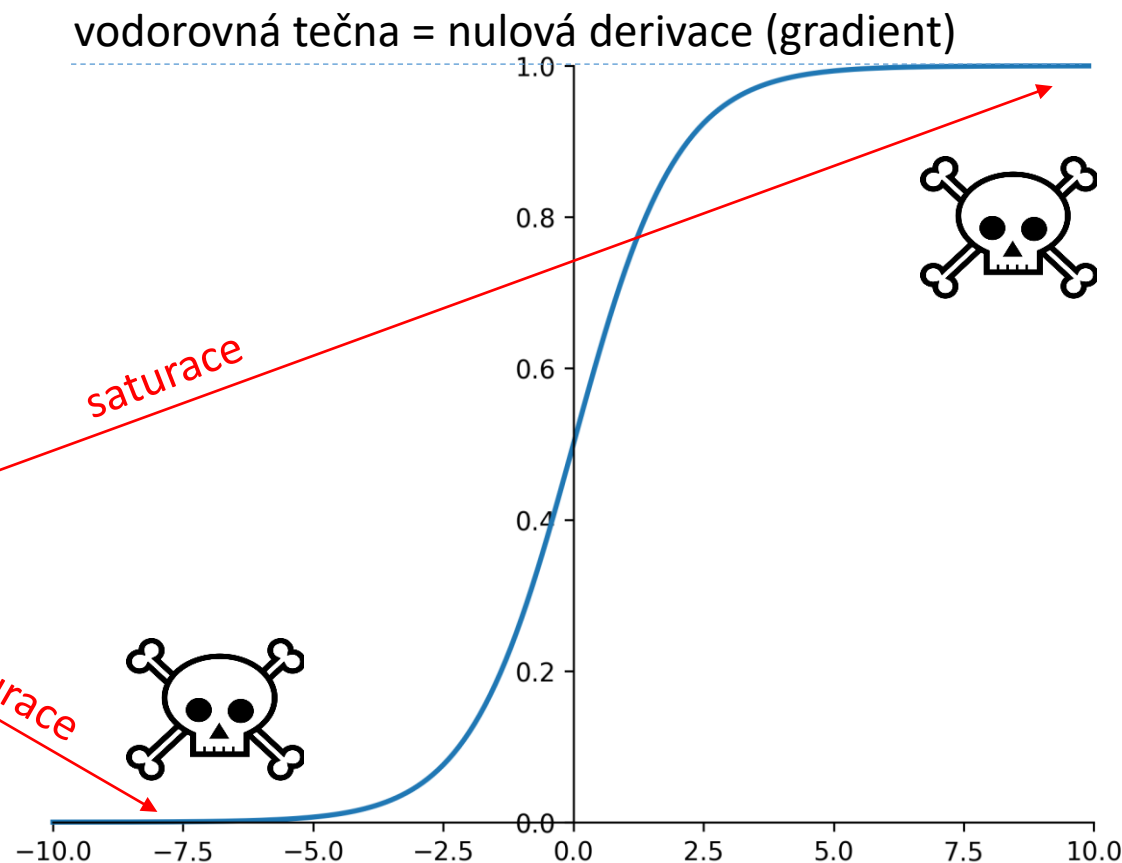


Sigmoid

- Před Alexnet prakticky jediná používaná aktivace
- Převádí vstup na pravděpodobnost, tj. do intervalu $\langle 0, 1 \rangle$
- Vstup x jsou typicky skóre s z předchozí lineární vrstvy
- Problémy:
 1. **“mizející” gradient**
 2. pouze kladné hodnoty
 3. exp funkce zbytečně náročná na výpočet



vanishing gradient

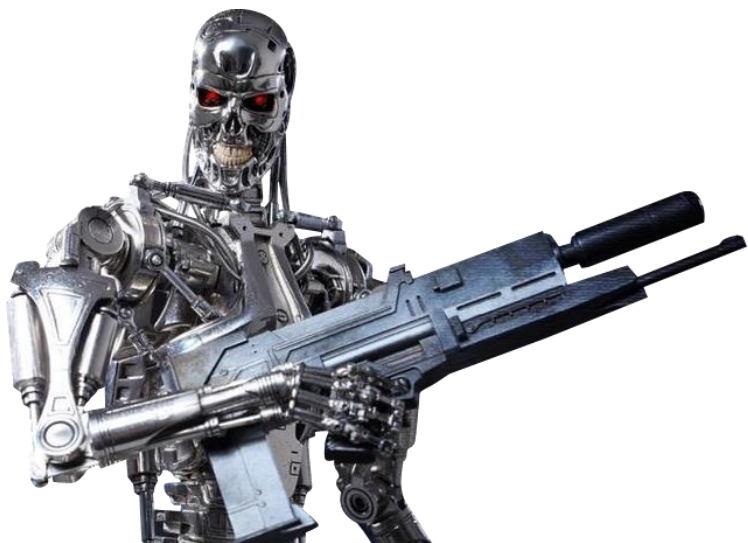


Hyperbolický tangens

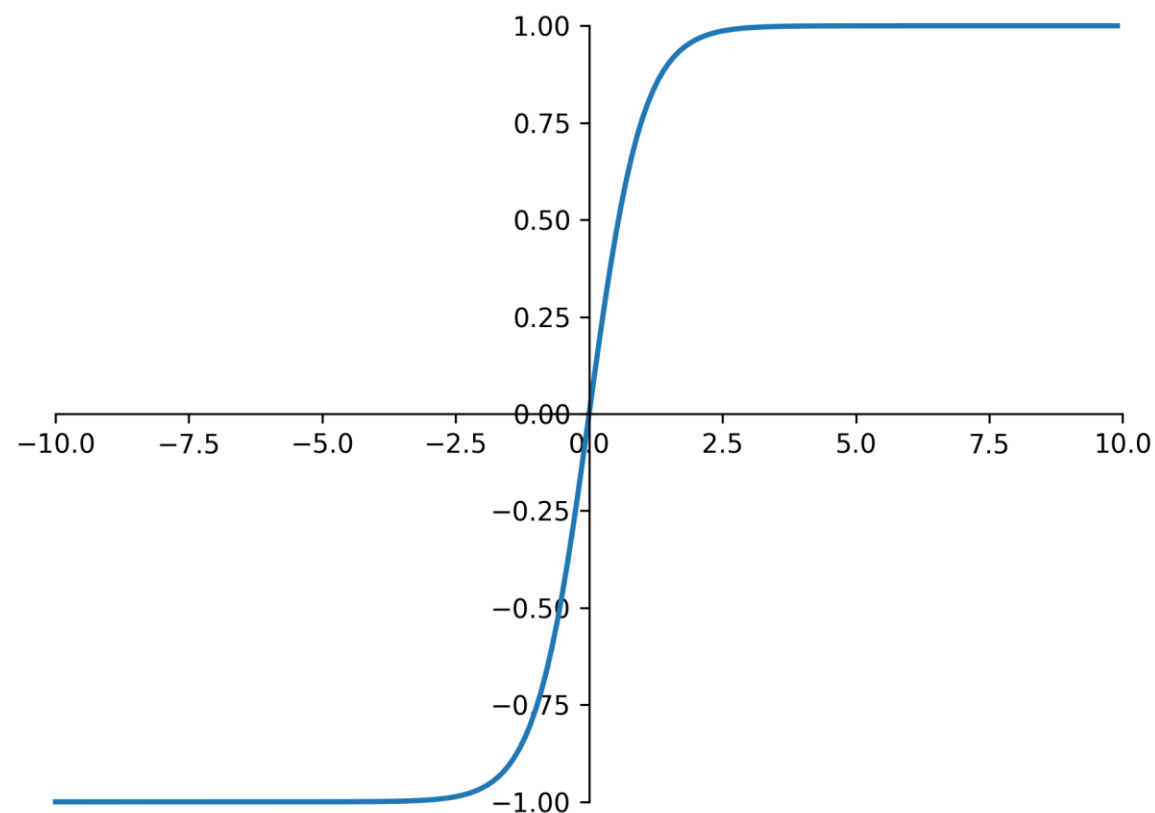
- Vlastně jen přeškálovaný sigmoid

$$\tanh(x) = 2 \cdot \sigma(2x) - 1$$

- Pouze tedy vycentruje sigmoid
- Ale stále “zabíjí” gradient



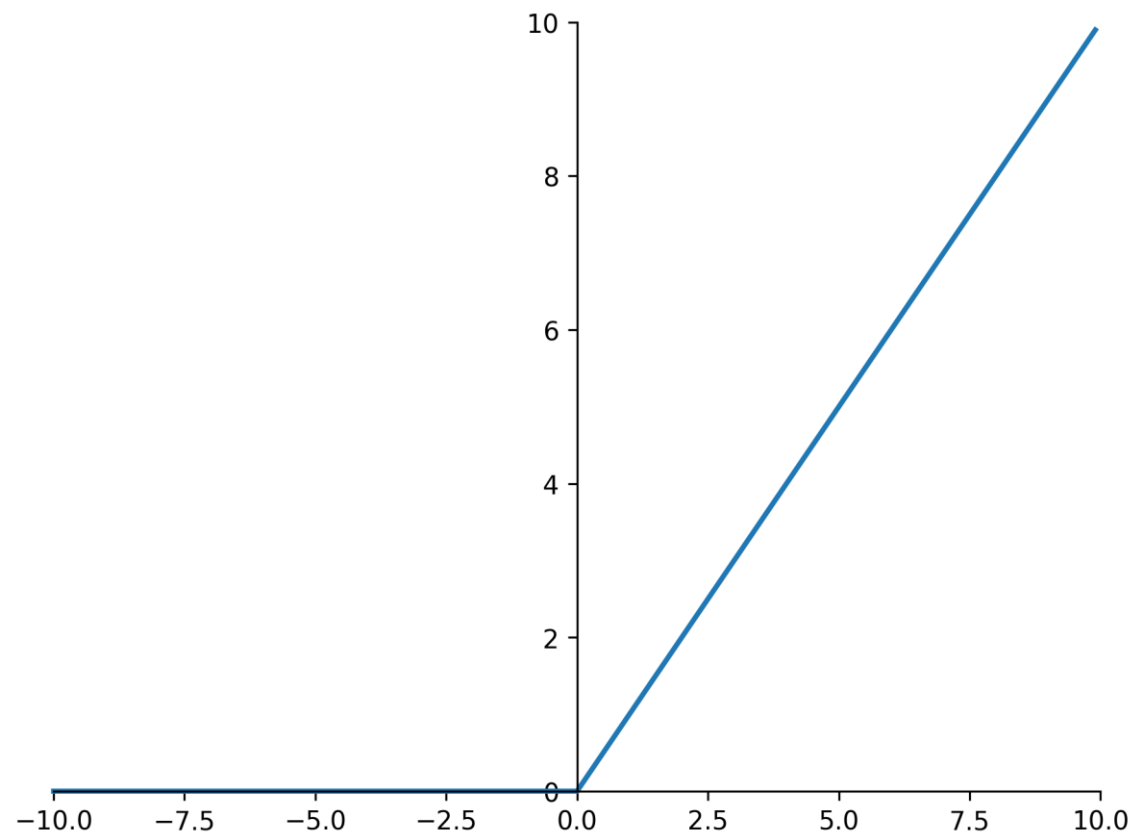
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



ReLU

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

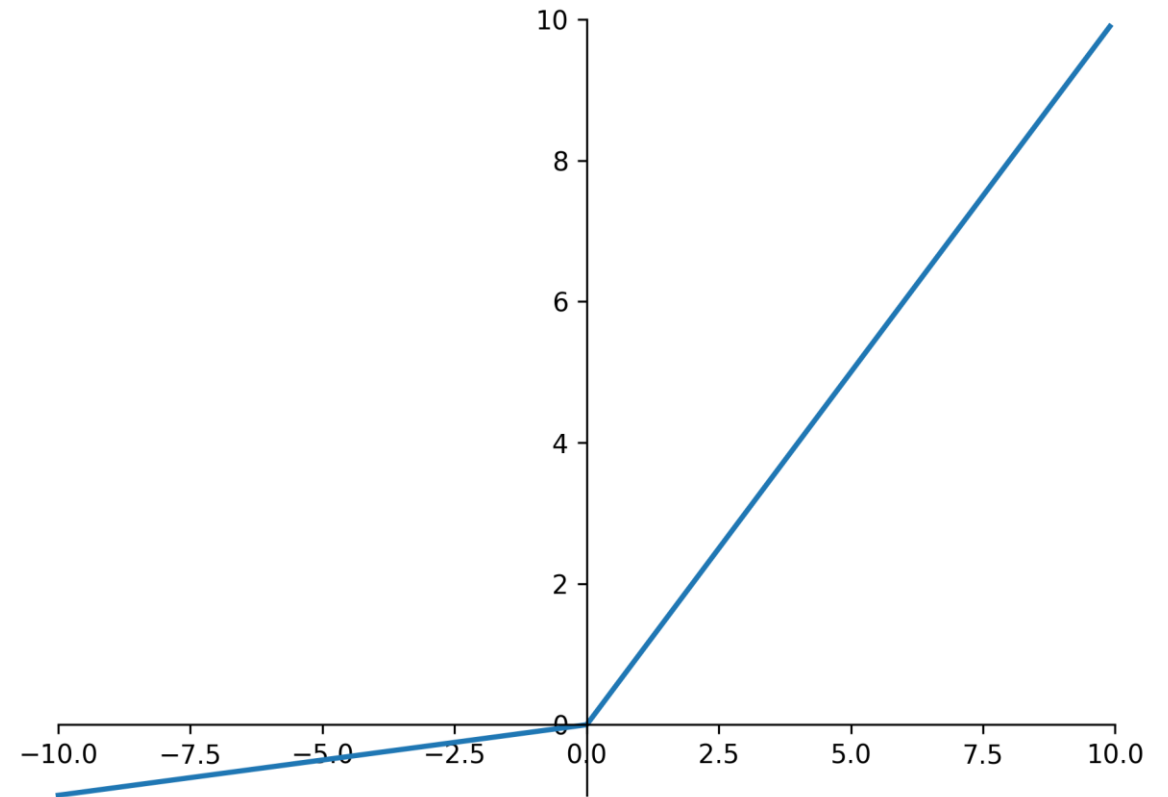
- Rectified Linear Unit
- Saturuje pouze v záporu
- Výpočetně nenáročné (bez expů)
- Trénování je mnohem rychlejší než se sigmoid či tanh
- Defaultní volba pro vnitřní nonlinearity



Leaky/Parametric ReLU

$$\text{PReLU}(x) = \max(\alpha x, x)$$

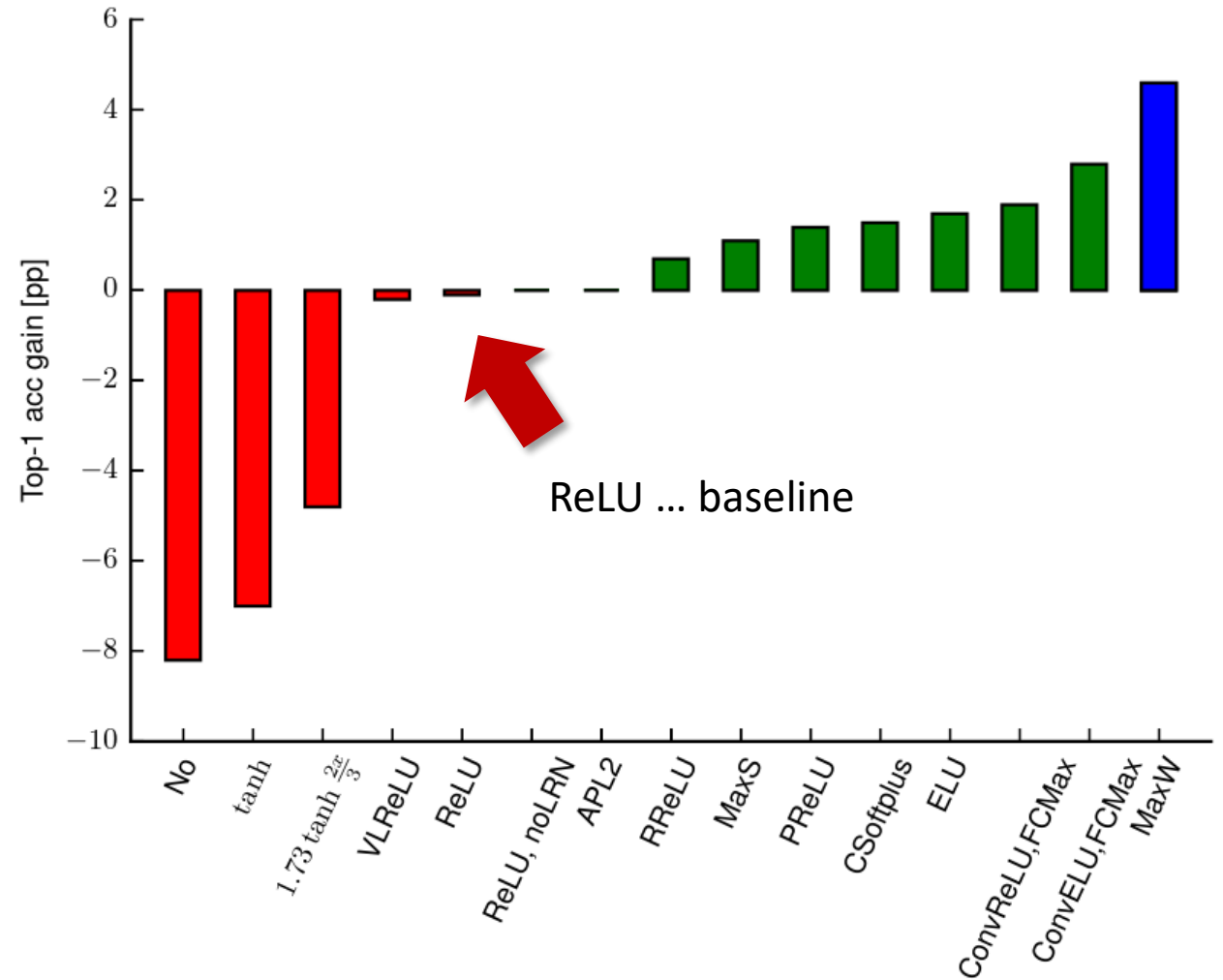
- Snaží se řešit problémy ReLU
 - nemá saturaci → neumírá gradient
 - zachovává rychlost
- Parametr α buď jako
 - konst. (např. 0.01) → Leaky ReLU
 - učitelný parameter → Parametric ReLU



Další nonlinearity

Table 3: Non-linearities tested.

Name	Formula	Year
none	$y = x$	-
sigmoid	$y = \frac{1}{1+e^{-x}}$	1986
tanh	$y = \frac{e^{2x}-1}{e^{2x}+1}$	1986
ReLU	$y = \max(x, 0)$	2010
(centered) SoftPlus	$y = \ln(e^x + 1) - \ln 2$	2011
LReLU	$y = \max(x, \alpha x), \alpha \approx 0.01$	2011
maxout	$y = \max(W_1x + b_1, W_2x + b_2)$	2013
APL	$y = \max(x, 0) + \sum_{s=1}^S a_i^s \max(0, -x + b_i^s)$	2014
VReLU	$y = \max(x, \alpha x), \alpha \in 0.1, 0.5$	2014
RReLU	$y = \max(x, \alpha x), \alpha = \text{random}(0.1, 0.5)$	2015
PReLU	$y = \max(x, \alpha x), \alpha \text{ is learnable}$	2015
ELU	$y = x, \text{ if } x \geq 0, \text{ else } \alpha(e^x - 1)$	2015



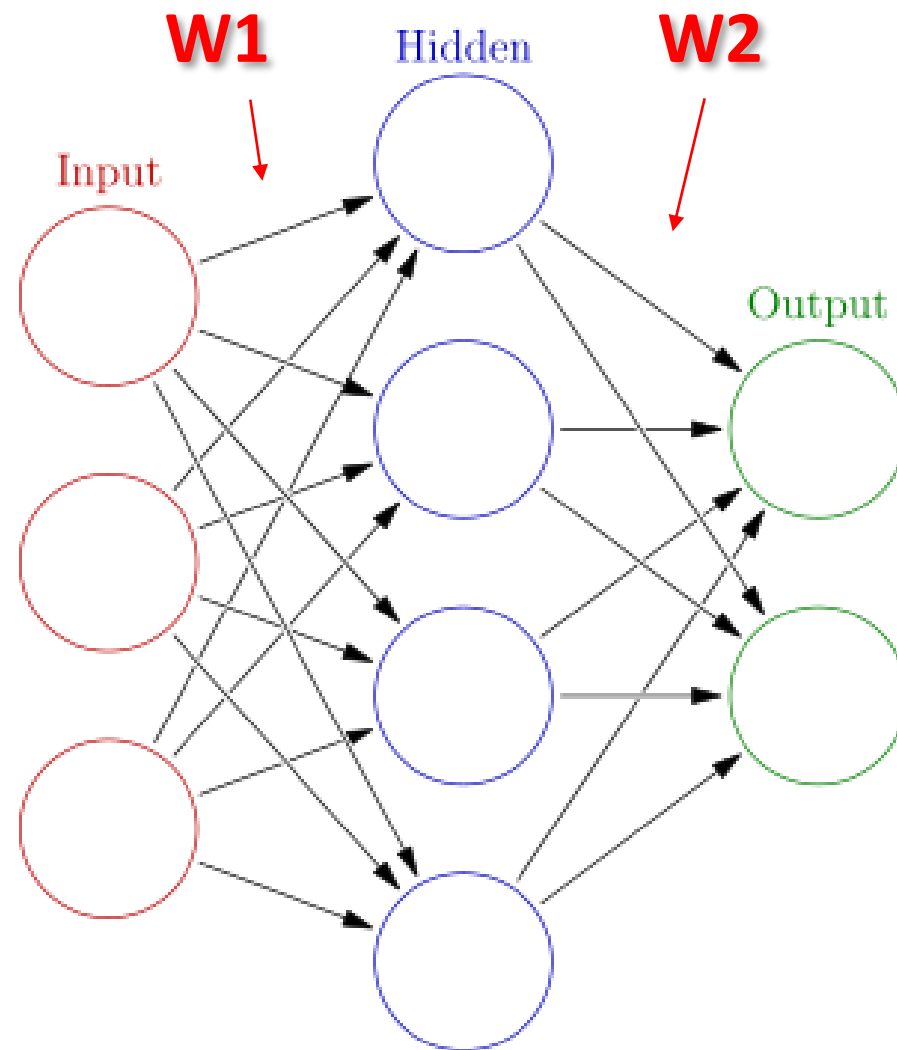
Inicializace

Náhodná inicializace

```
W1 = 0.01 * np.random.randn(3, 4)
```

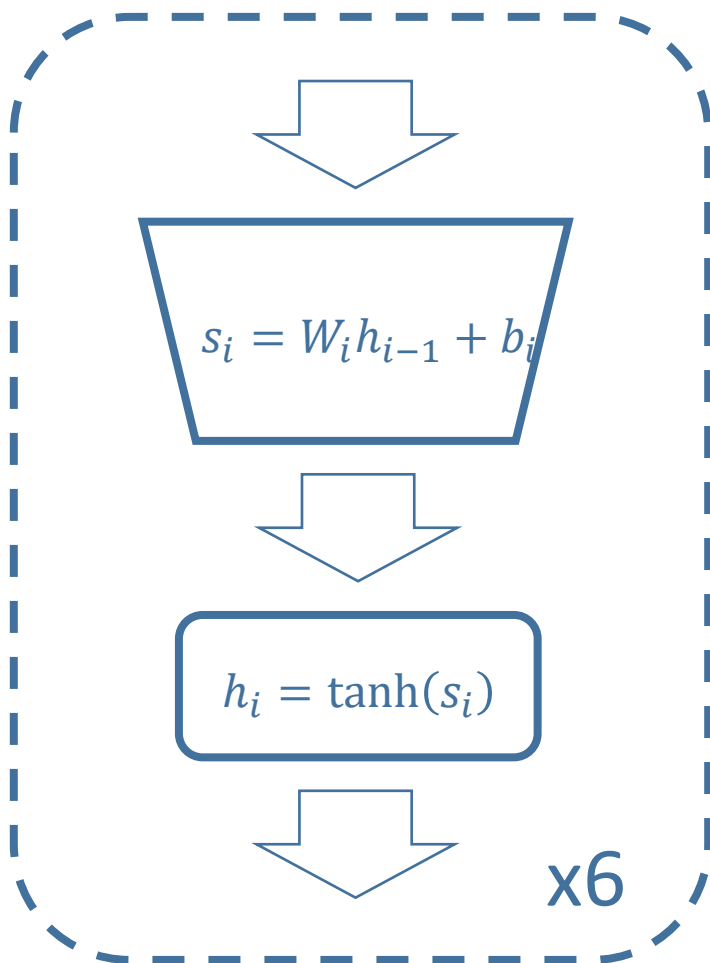
```
W2 = 0.01 * np.random.randn(4, 2)
```

- Proč náhodně? Proč ne všechno na nuly?
- Potřebujeme narušit symetrii (break the symmetry)
 - Řekněme, že váhy jsou nuly a hidden vrstva používá sigmoid
 - do output pak jdou samé 0.5
 - tzn., že i gradient bude všude téměř stejný
 - **chceme opak:** aby každý neuron dělal něco jiného

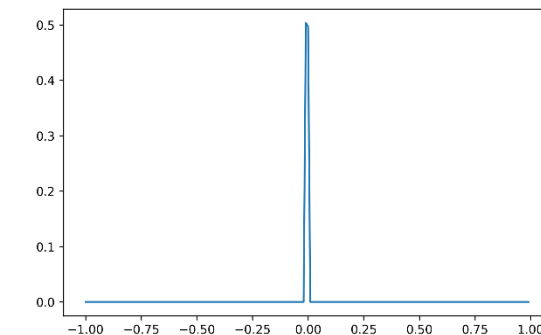
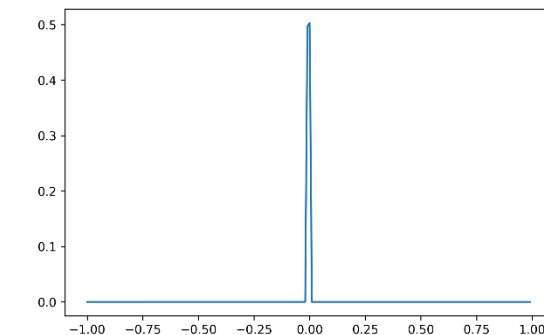
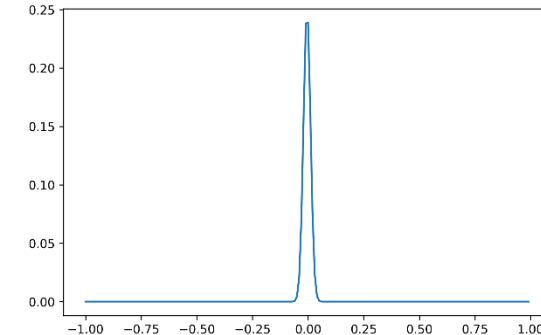
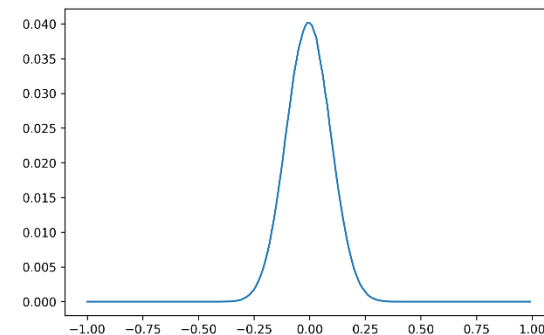
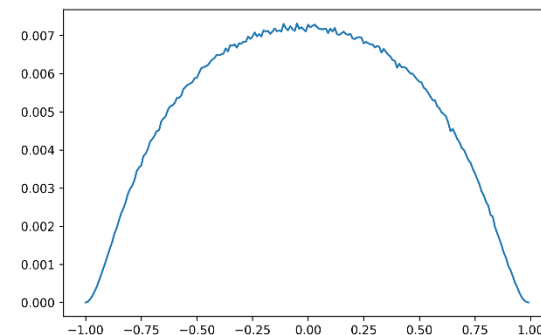
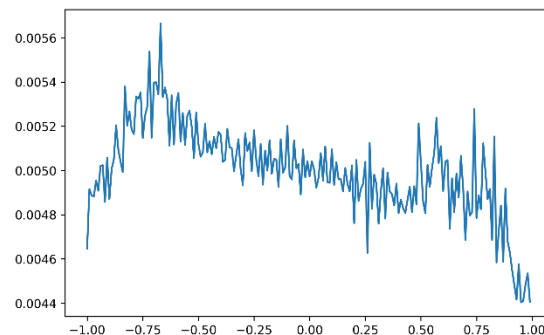


Příliš malé váhy

- 6 vrstev feed forward síť dle obr.
- histogram hodnot skrytých vrstev

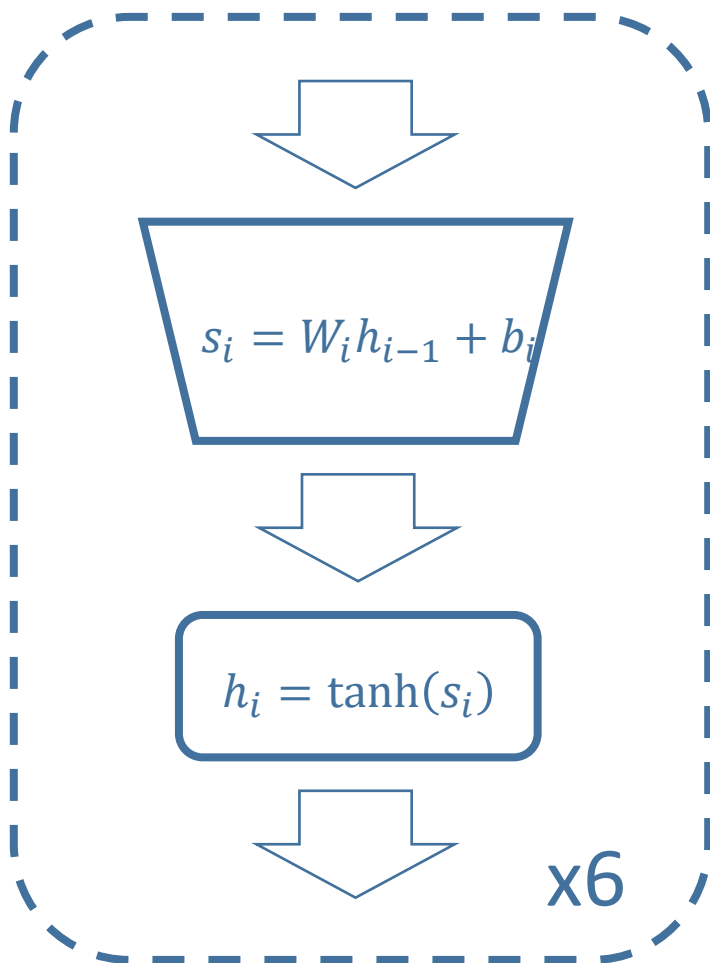


```
Wi = 0.01 * np.random.randn(fan_in, fan_out)
```

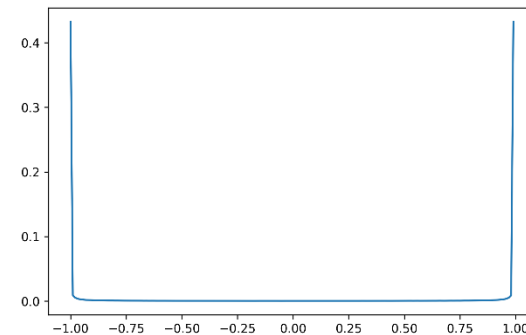
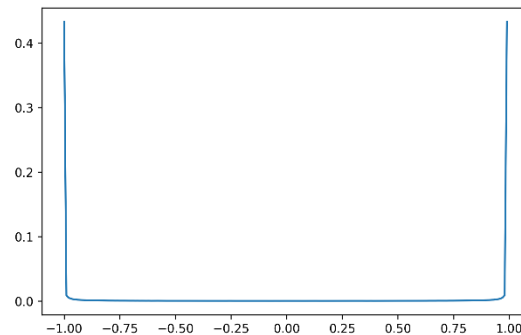
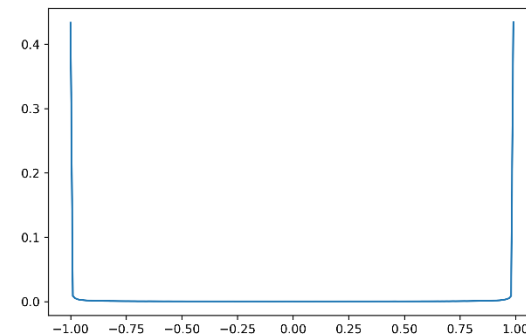
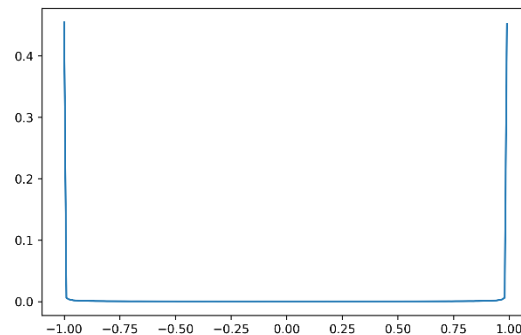
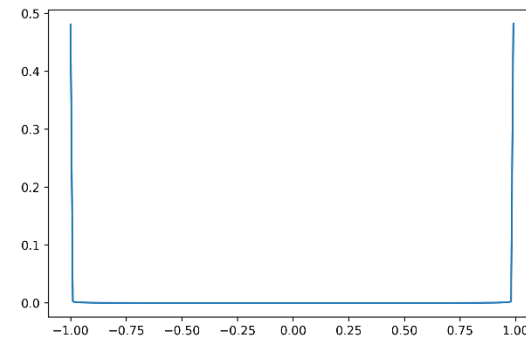
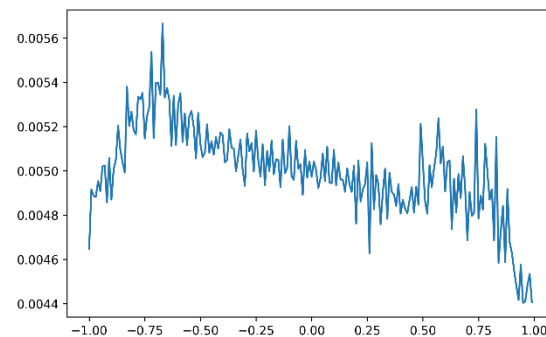


Příliš velké váhy

- 6 vrstev feed forward síť dle obr.
- histogram hodnot skrytých vrstev



$W_i = 1.00 * \text{np.random.randn}(\text{fan_in}, \text{fan_out})$

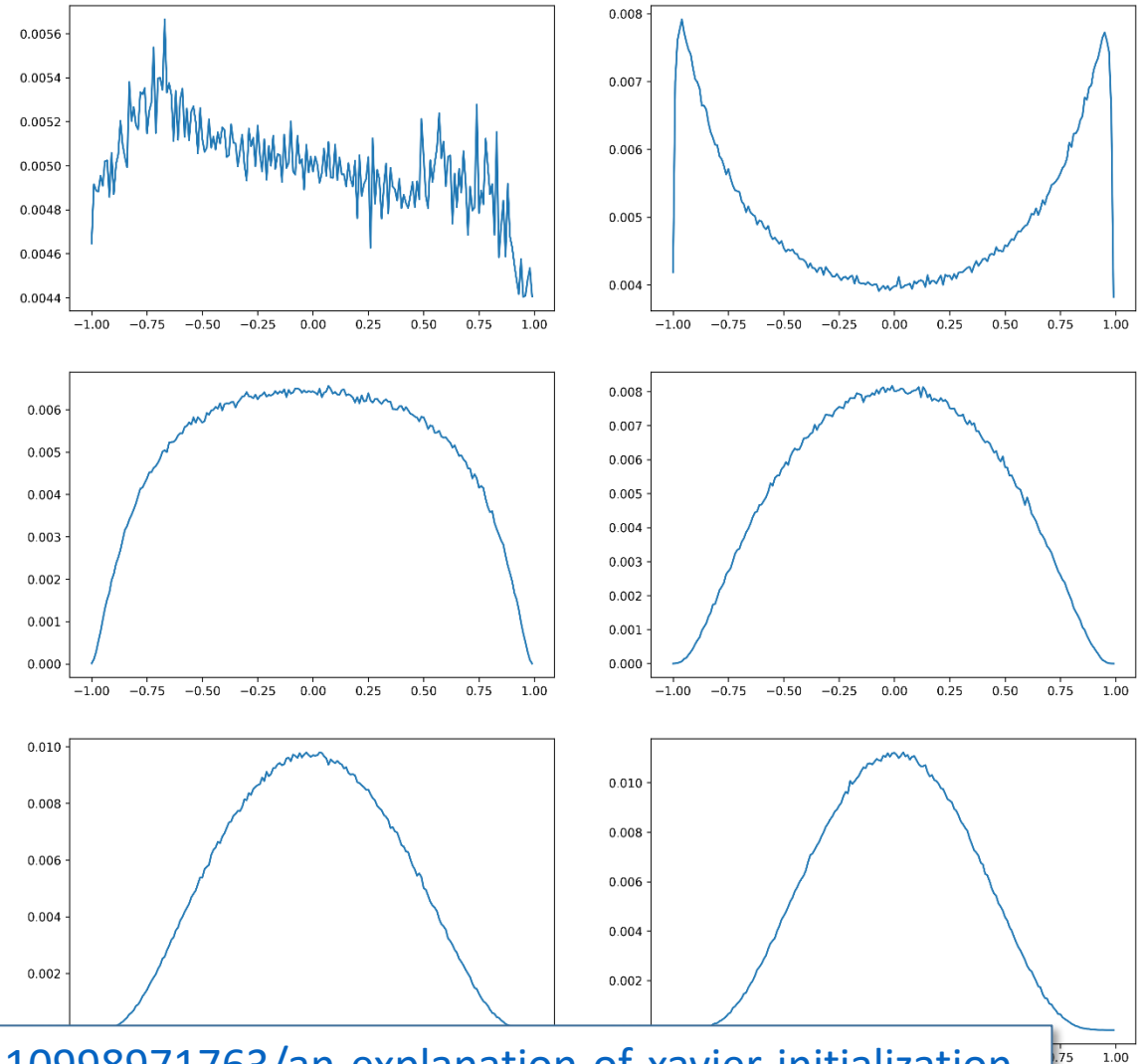
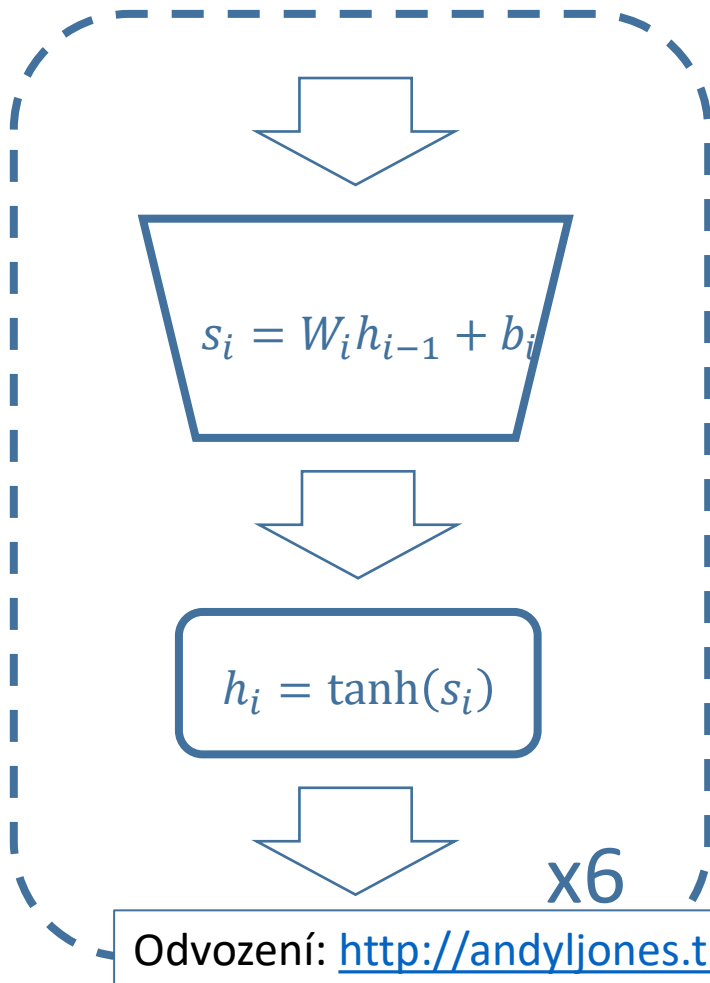


Xavier / Glorot inicializace

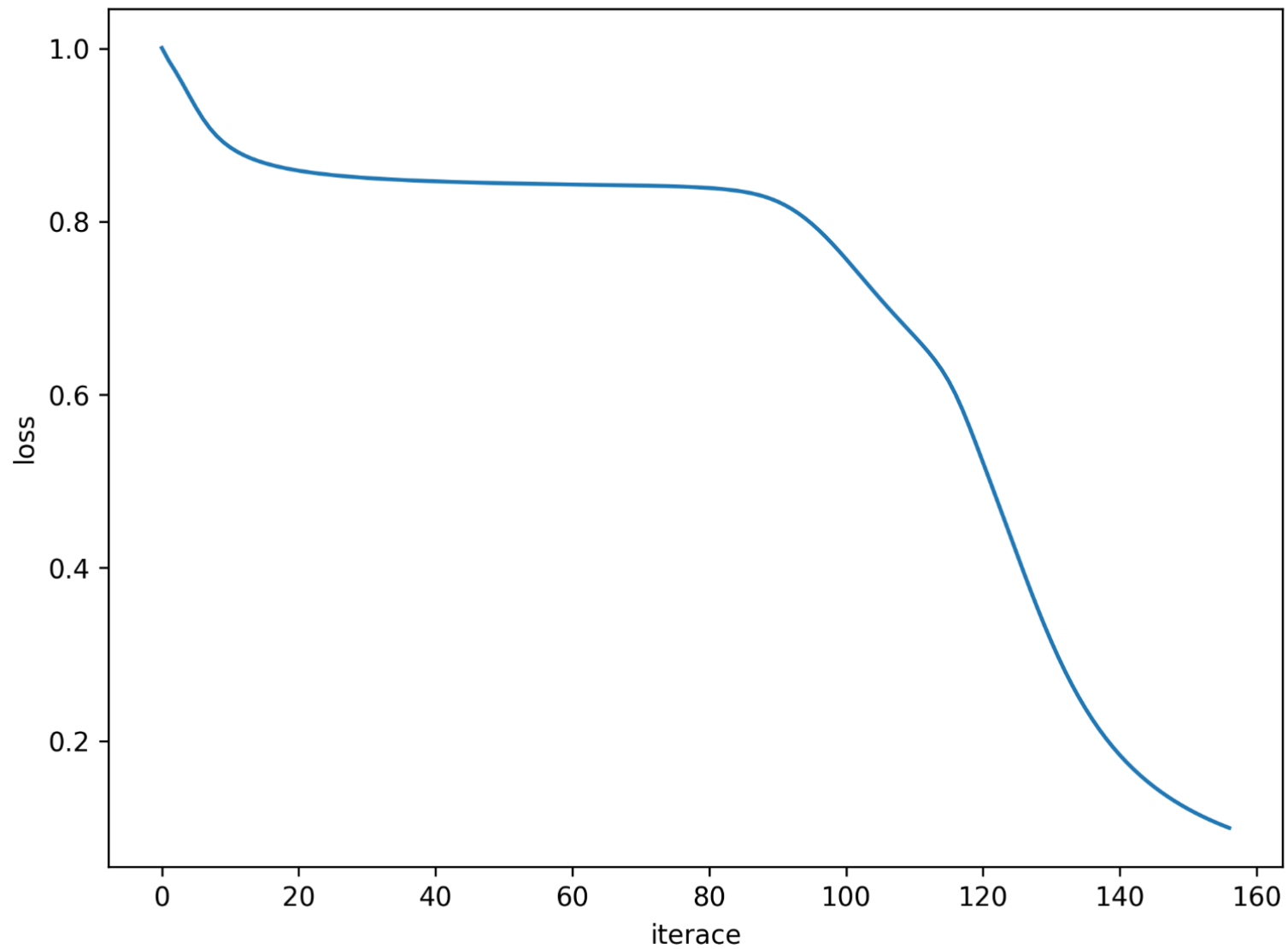


- 6 vrstev feed forward síť dle obr.
- histogram hodnot skrytých vrstev

```
Wi = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```



Špatná inicializace



Aktivace & Inicializace

- Nejjednodušší zkusit ReLU → když funguje, vyzkoušet její varianty nebo např. SELU
- U ReLU namísto Glorot inicializace vhodnější **He**:

```
Wi = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in / 2.)
```

- Odvození: <http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization>
- Co biasy?
 - většinou inicializujeme na nuly: `b = np.zeros(fan_out)`
 - U ReLU možné malé kladné hodnoty: `b = 0.01 * np.ones(fan_out)`