

Aplikace neuronových sítí

Vícevrstvé perceptrony a trénování sítí v praxi

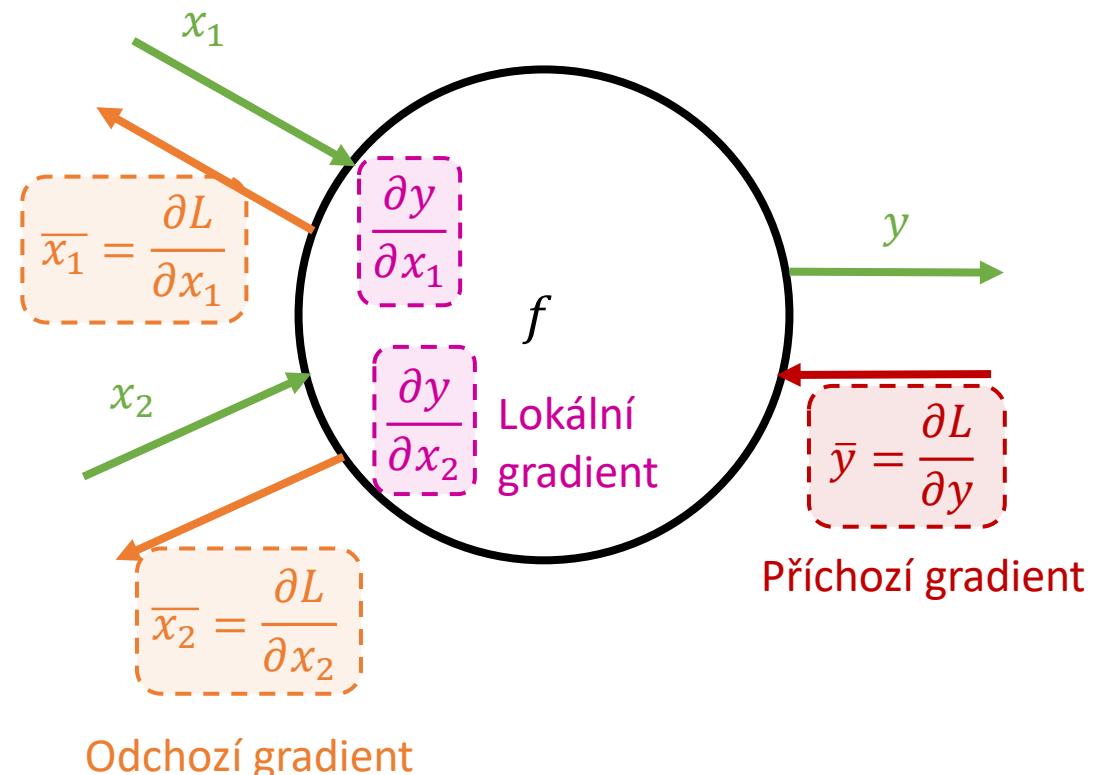
Minule: Uzel grafu a lokální gradient

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

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

- Zpětný průchod je aplikace řetízkového pravidla

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



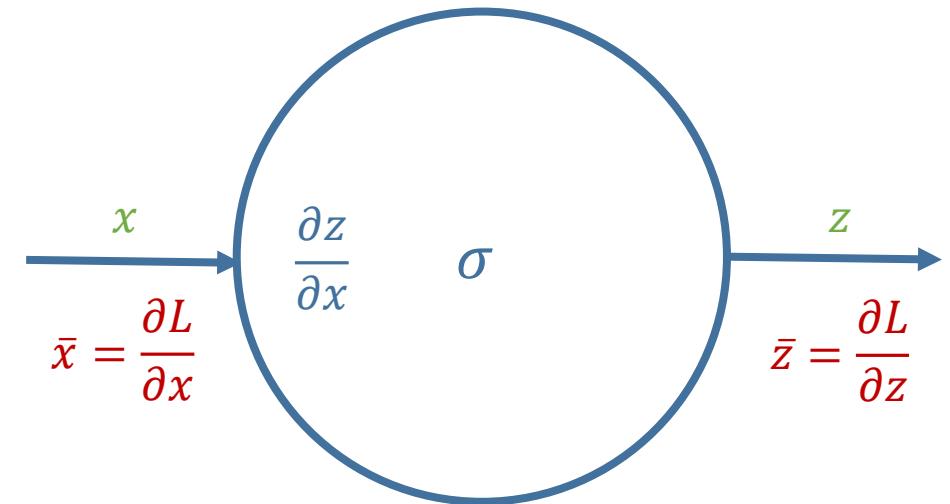
přehová notace: <https://people.maths.ox.ac.uk/gilesm/files/NA-08-01.pdf>

Minule: Funkce jako uzel ve výpočetním grafu

```
class Sigmoid(Function):
```

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

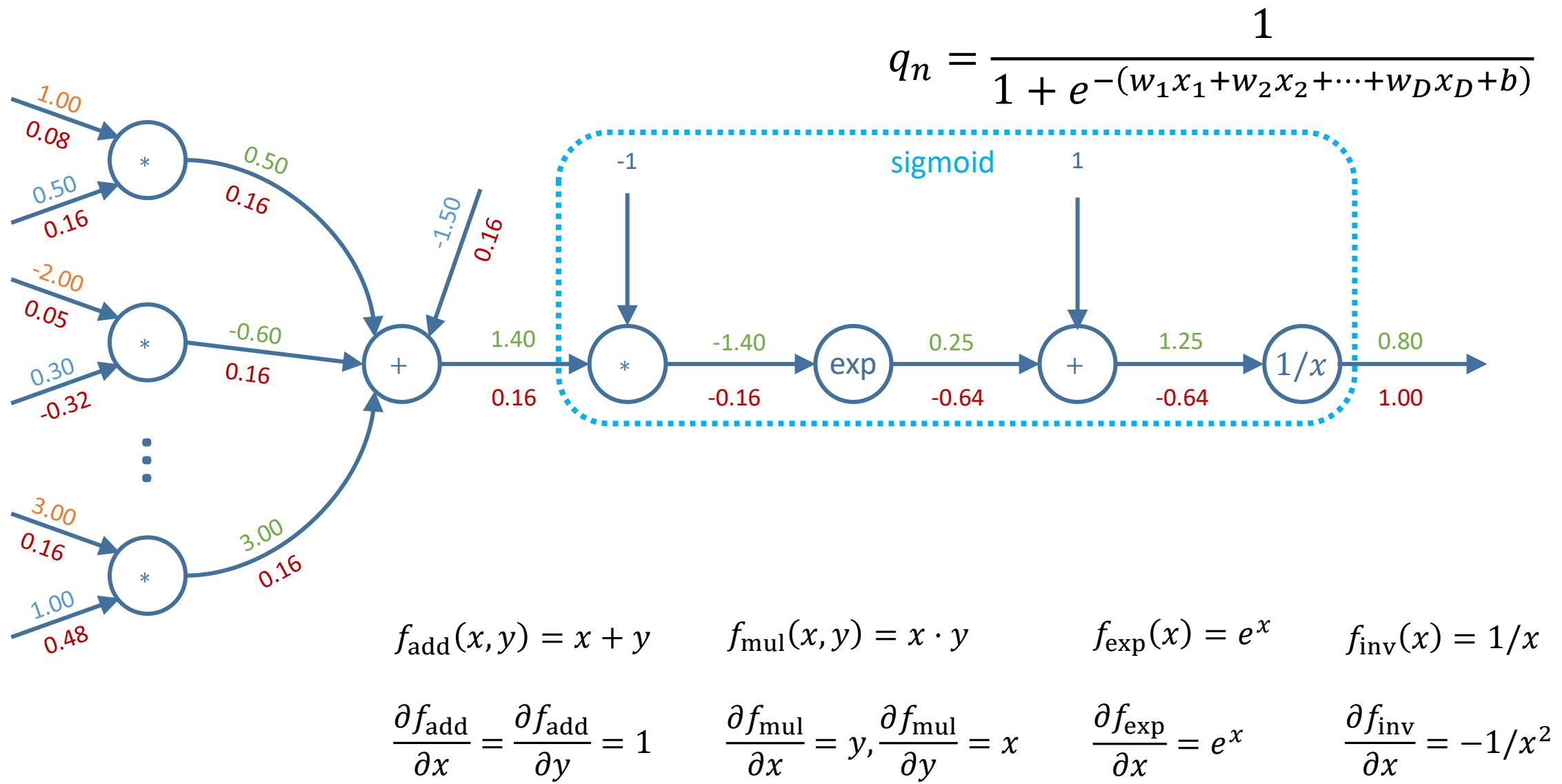
```
@staticmethod  
def backward(dz: float, cache: tuple):  
    z, = cache  
    dx = dz * z * (1 - z) # retizkove pravidlo  
    return dx
```



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** a **zpětný průchod**

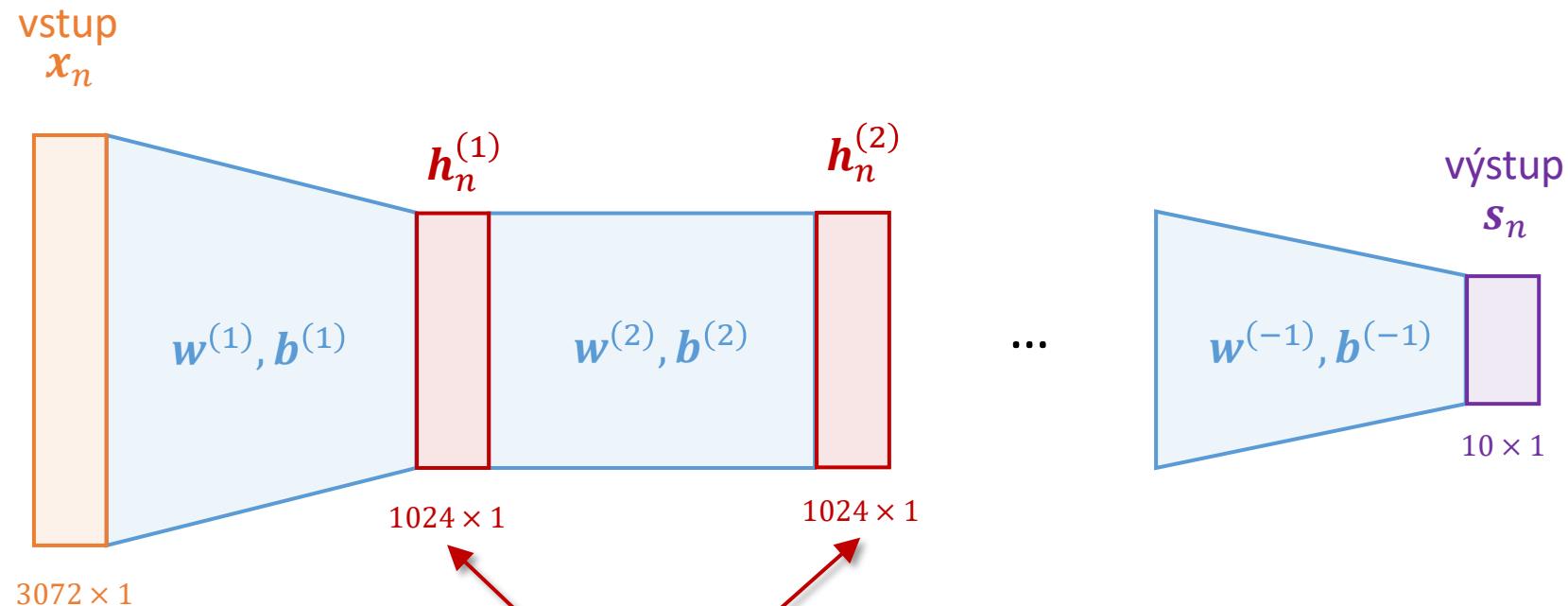
Minule: Binární logistická regrese se sigmoidem jako graf



Vícevrstvý perceptron

Vícevrstvý perceptron (Multi-Layer Perceptron, MLP)

- Označuje se také jako dopředná síť (Feed-Forward Network, FFN)
- Opakují se především dva+ typy vrstev: **plně propojená** a **aktivace**



Bloky sestávající z plně propojené vrstvy a aktivace
se obvykle označují jako vrstvy a jejich výstup jako
tzv. skrytá vrstva (h jako hidden)

Plně propojená vrstva

- Vyskytuje se pod mnoha různými názvy
 - Fully-Connected (FC) (MATLAB)
 - Affine
 - Linear (PyTorch)
 - Dense (Keras)
 - InnerProduct (Caffe)
- Dopředný průchod

$$\mathbf{z}_n = \mathbf{w} \cdot \mathbf{x}_n + \mathbf{b}$$

- Rozvinutý zápis

$$\begin{bmatrix} z_{n,1} \\ \vdots \\ z_{n,K} \end{bmatrix} = \begin{bmatrix} w_{1,1} & \dots & w_{1,D} \\ \vdots & \ddots & \vdots \\ w_{K,1} & \dots & w_{K,D} \end{bmatrix} \cdot \begin{bmatrix} x_{n,1} \\ \vdots \\ x_{n,D} \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix}$$

- Vstupy
 - \mathbf{x}_n je sloupcový vektor s rozměry $D \times 1$
- Parametry
 - \mathbf{w} je váhová matice s rozměry $K \times D$
 - \mathbf{b} je sloupcový vektor biasů s rozměry $K \times 1$
- Gradient na vstup

$$\bar{\mathbf{x}}_n = \mathbf{w}^T \cdot \bar{\mathbf{z}}_n$$

- Gradient na váhy

$$\bar{\mathbf{w}} = \bar{\mathbf{z}}_n \cdot \bar{\mathbf{x}}_n^T$$

- Gradient na bias

$$\bar{\mathbf{b}} = \bar{\mathbf{z}}_n$$

Sigmoid vrstva

- Dopředný průchod

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

Parametry?
Žádné!

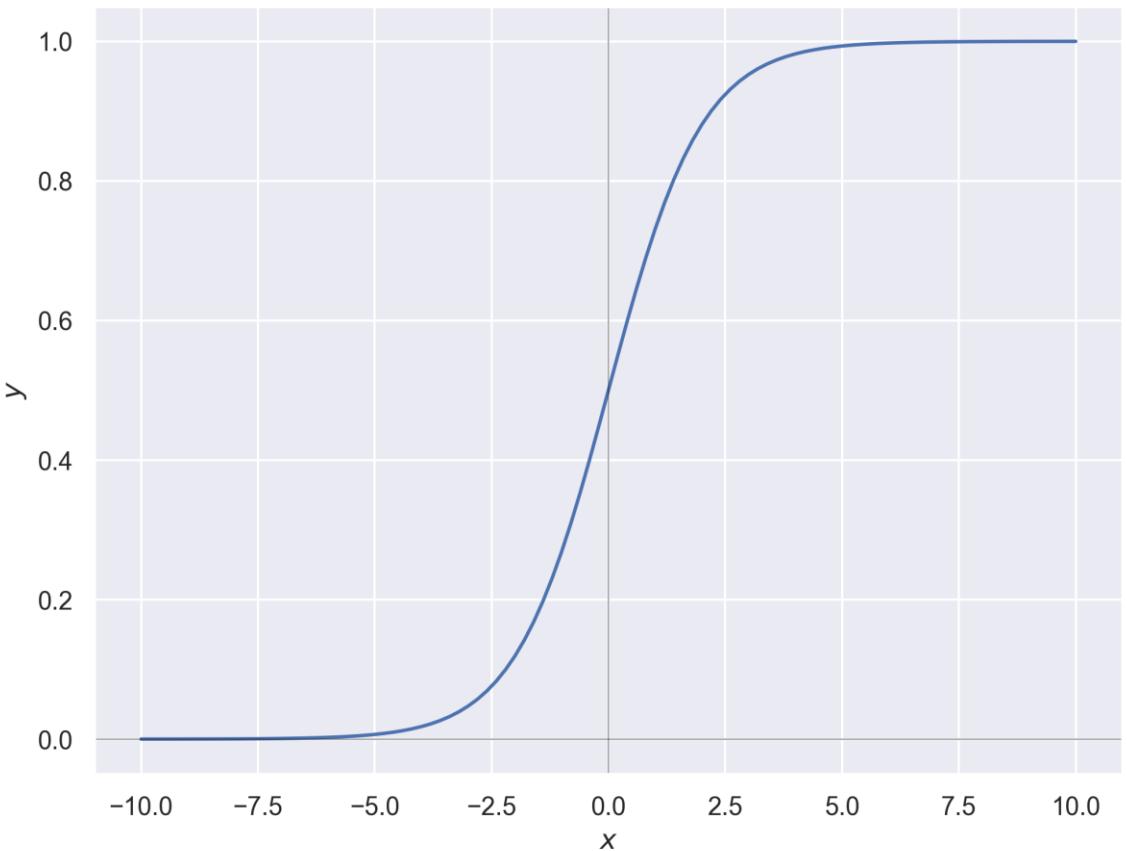
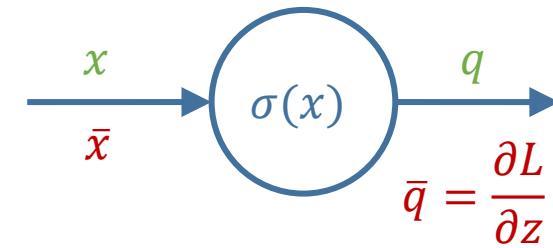
- Derivace funkce sigmoid

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

q ... cache

- Zpětný průchod

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



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 range(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

Proč aktivace?

- Opět různé názvy, nejčastěji aktivace (activation) nebo nelinearity (non-linearity)
- Zajišťuje, aby výsledná síť byla schopna modelovat i nelineární funkce
- Bez aktivace by dopřednný průchod např. dvouvrstvého MLP byl

$$\begin{aligned} \mathbf{z}_n &= \mathbf{w}^{(2)} \cdot (\mathbf{w}^{(1)} \cdot \mathbf{x}_n + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} \\ &= [\mathbf{w}^{(2)} \cdot \mathbf{w}^{(1)}] \cdot \mathbf{x}_n + [\mathbf{w}^{(2)} \cdot \mathbf{b}^{(1)}] + [\mathbf{b}^{(2)}] \\ &= \mathbf{w}^{(\text{new})} \cdot \mathbf{x}_n + \mathbf{b}^{(\text{new})} \end{aligned}$$

druhá vrstva
první vrstva

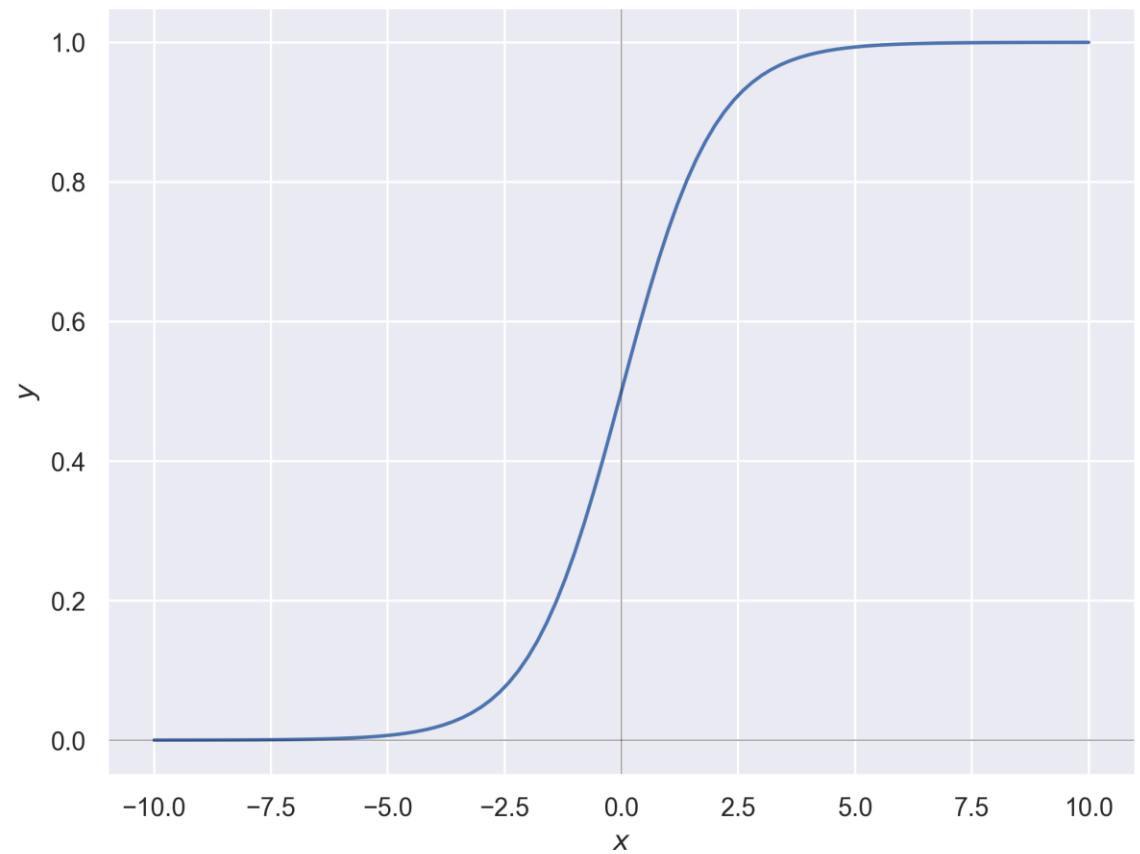
matice x matice = matice matice x vektor = vektor vektor

Více lineárních vrstev za sebou bez aktivace jsou stále jen lineární model

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

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

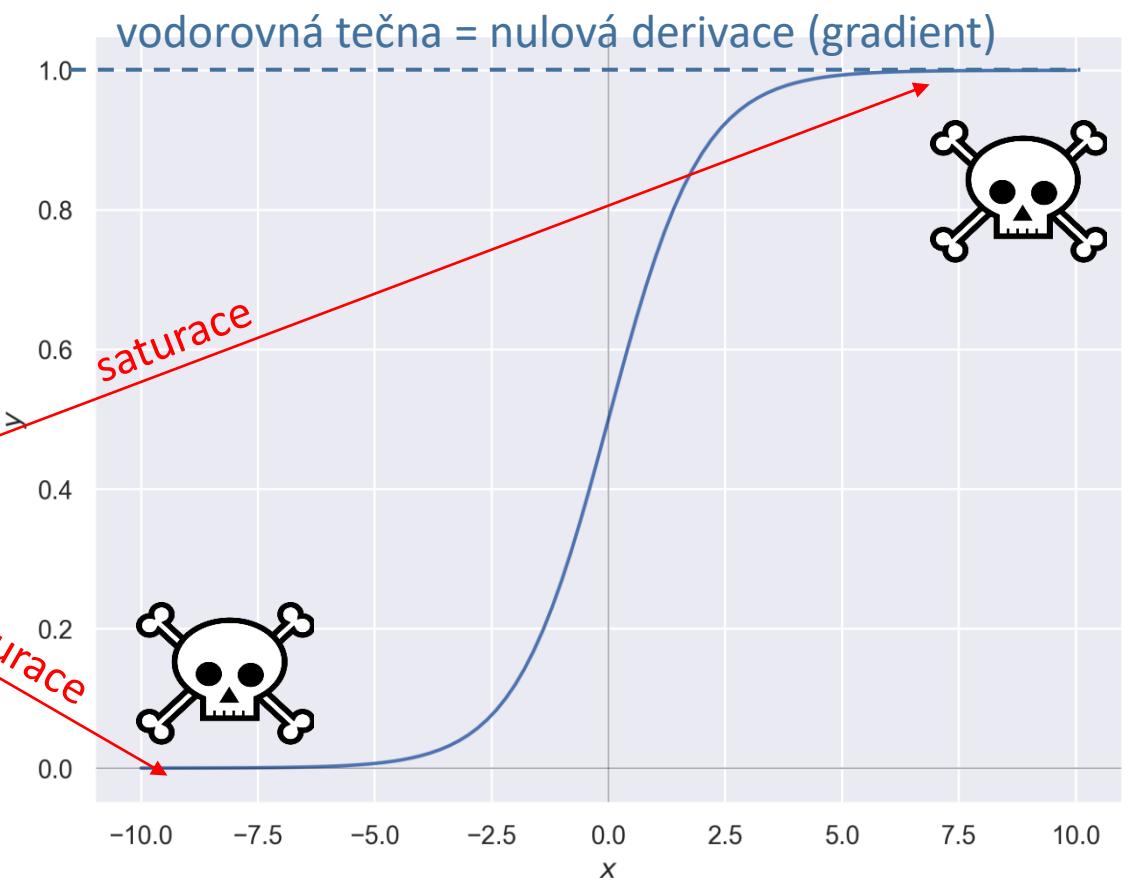


Sigmoid

- 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

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



Mizející gradient (vanishing gradient)

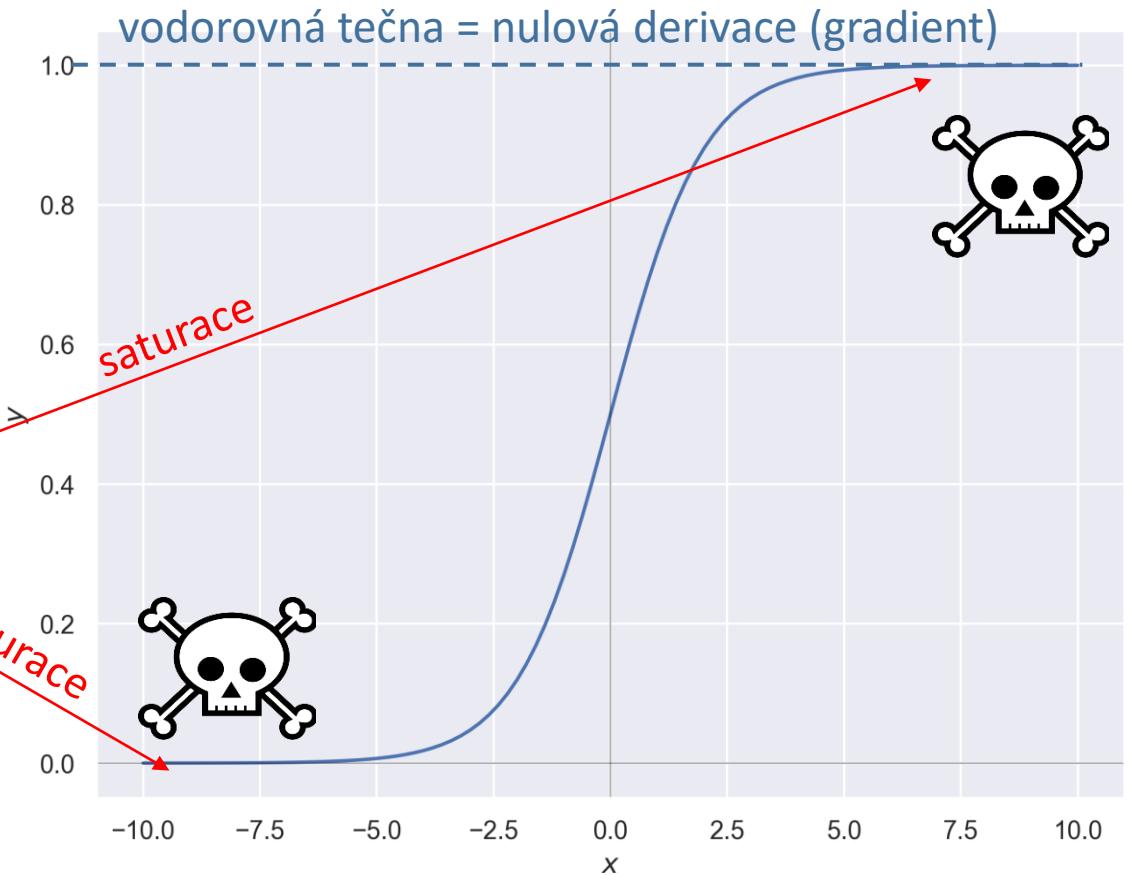
- Zpětný průchod sigmoidy je

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

- Např. $0.99 \cdot (1 - 0.99) = 0.01 \rightarrow$ příchozí gradient \bar{q} je o dva řády snížen v magnitudě
- Pokud je síť hluboká a toto se děje opakovaně, gradient přicházející do nejnižších vrstev bude téměř nula
- Síť se potom není schopna učit
- Jeden z hlavních důvodů, proč hluboké sítě tak dlouho nefungovaly



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



Hyperbolický tangens

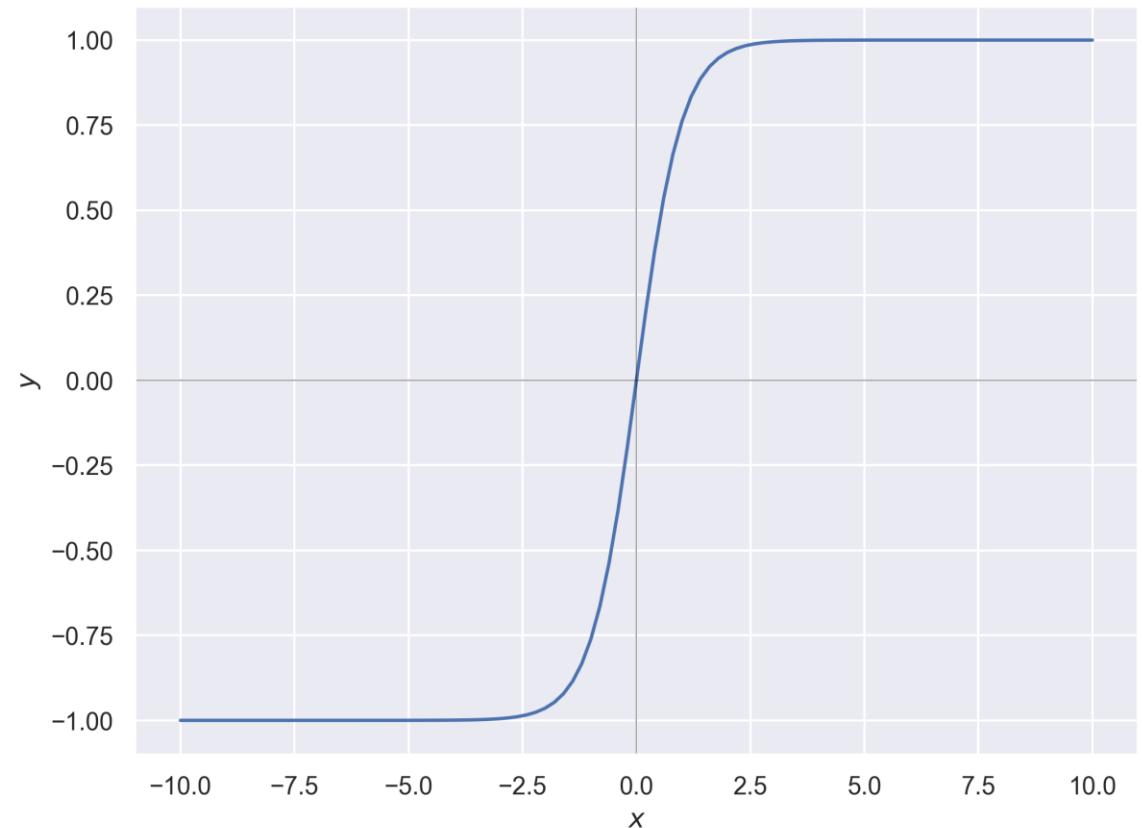
- Vlastně jen přeškálovaný sigmoid

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

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



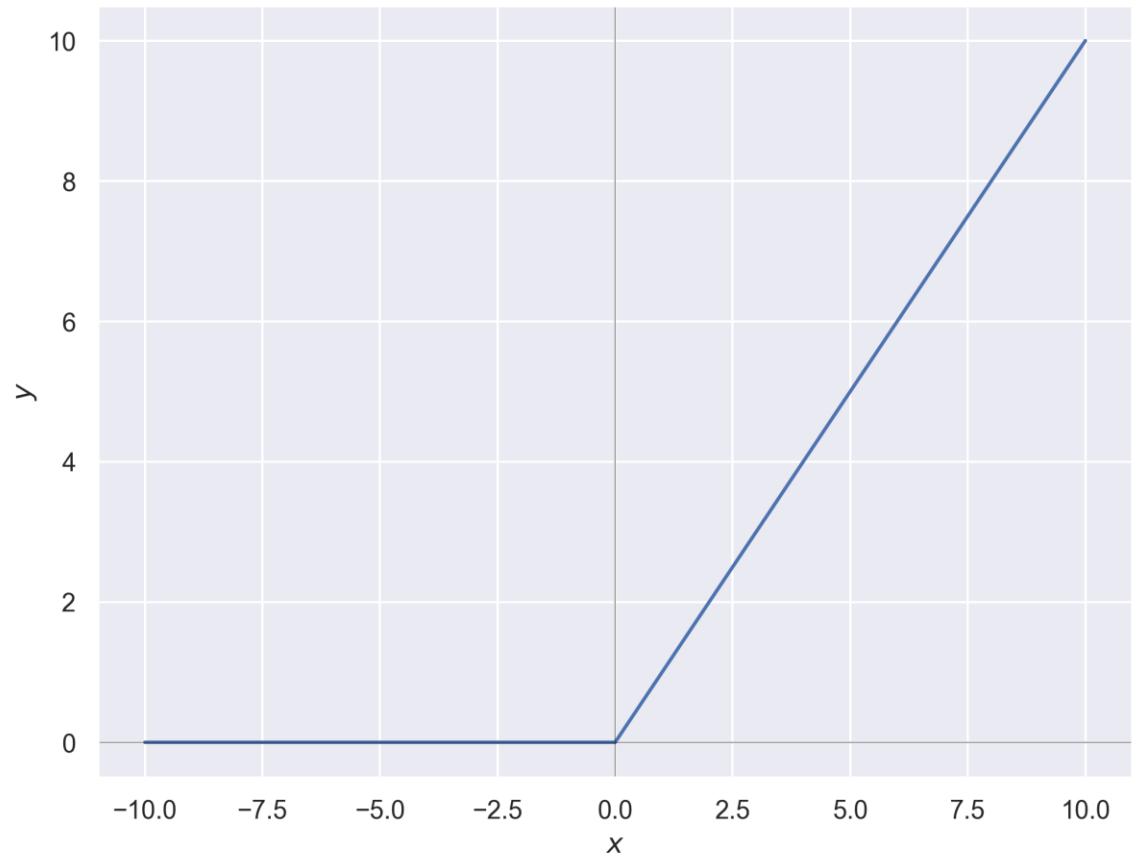
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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 nelinearity

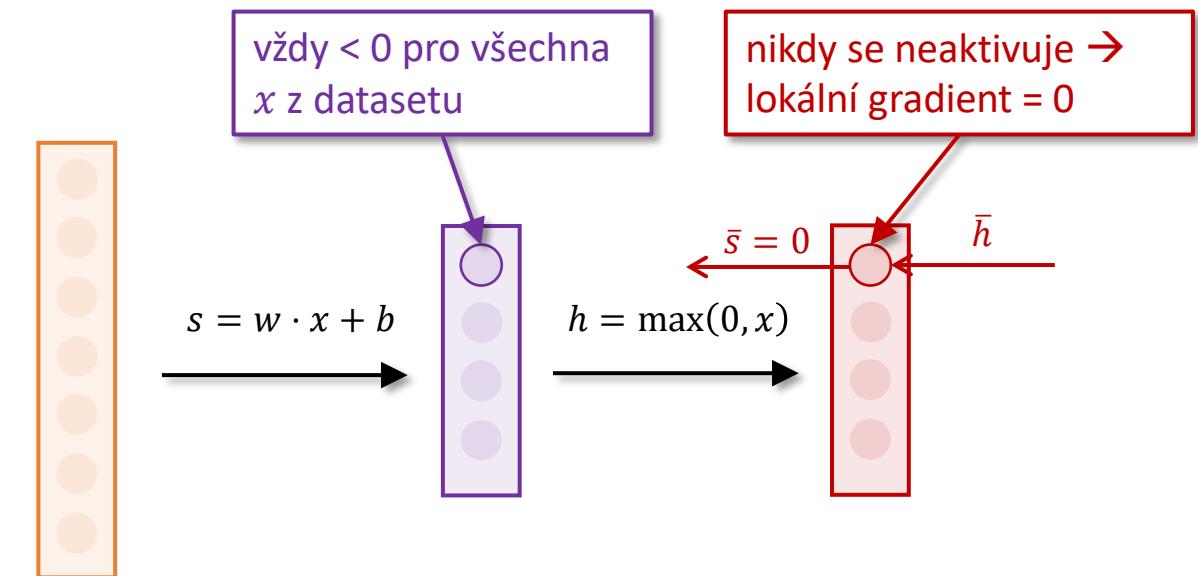


Dead ReLU

- Lokální gradient ReLU je

$$\frac{d\text{ReLU}(x)}{dx} = \begin{cases} 1 & \text{pokud } x > 0 \\ 0 & \text{jinak} \end{cases}$$

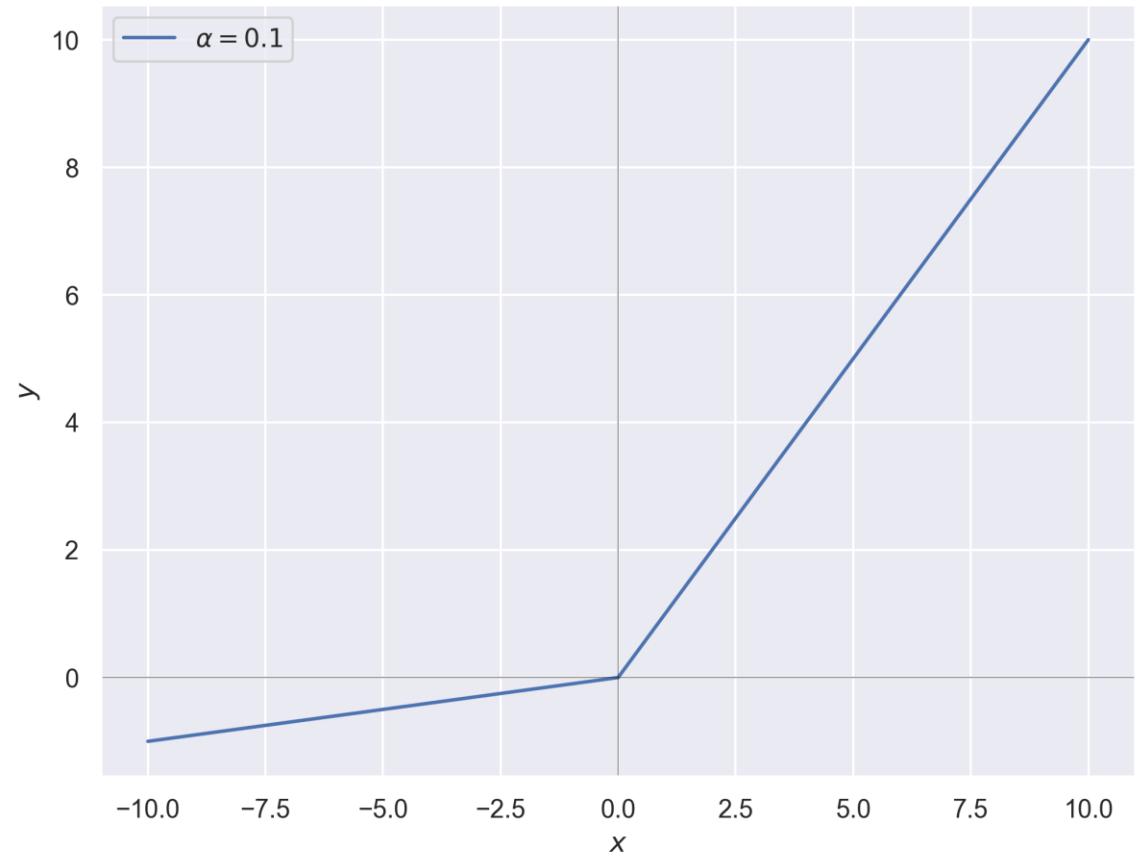
- Dokud na trénovacím datasetu existují příchozí hodnoty $x > 0$, ReLU propouští gradient → ReLU je aktivní
- Může se ale stát, že předchozí vrstva pro daný neuron na trénovacím datasetu nikdy nevygeneruje kladné číslo
- Vstup do ReLU je pak vždy $x < 0$ → lokální gradient do řetízkového pravidla je nula → netrénuje → ReLU je “mrtvá”



Leaky/Parametric ReLU

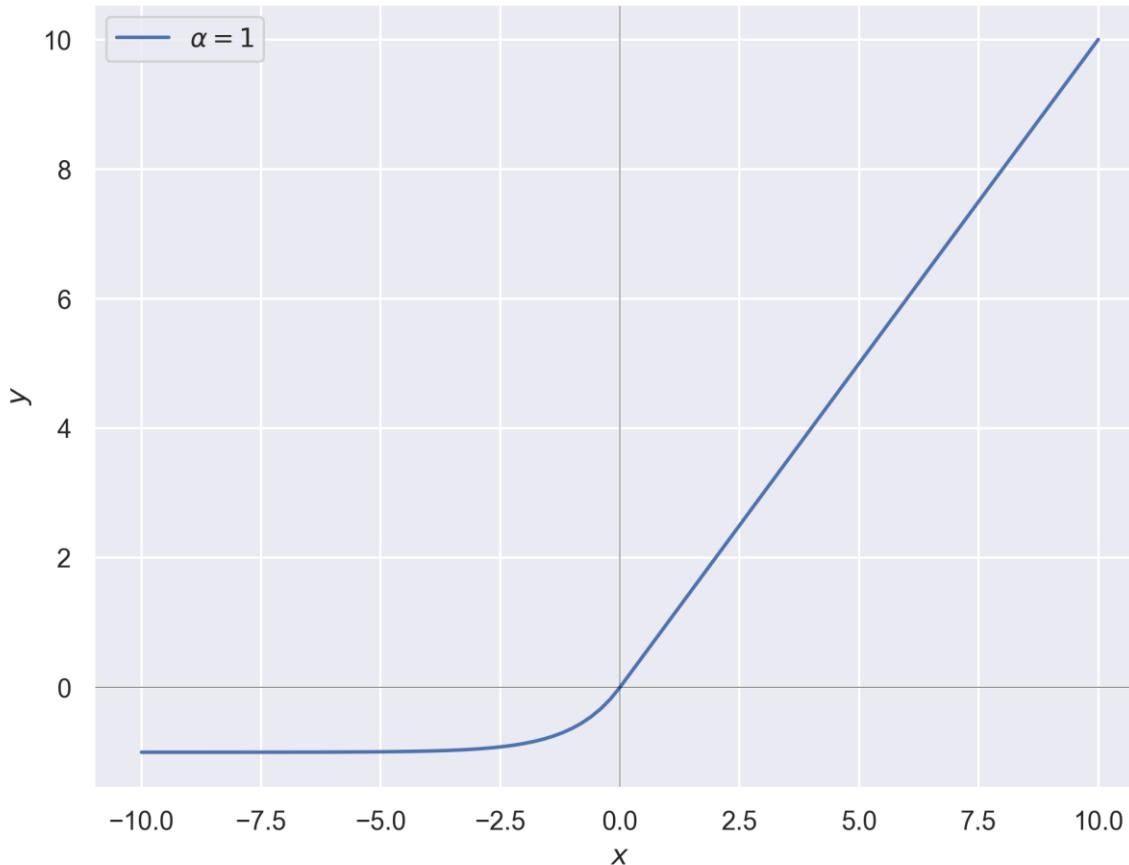
- Snaží se řešit problémy ReLU
 - nemá saturaci → neumírá gradient
 - zachovává rychlosť
- Parametr α budť jako
 - konst. (např. 0.01) → Leaky ReLU
 - učitelný parameter → Parametric ReLU

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

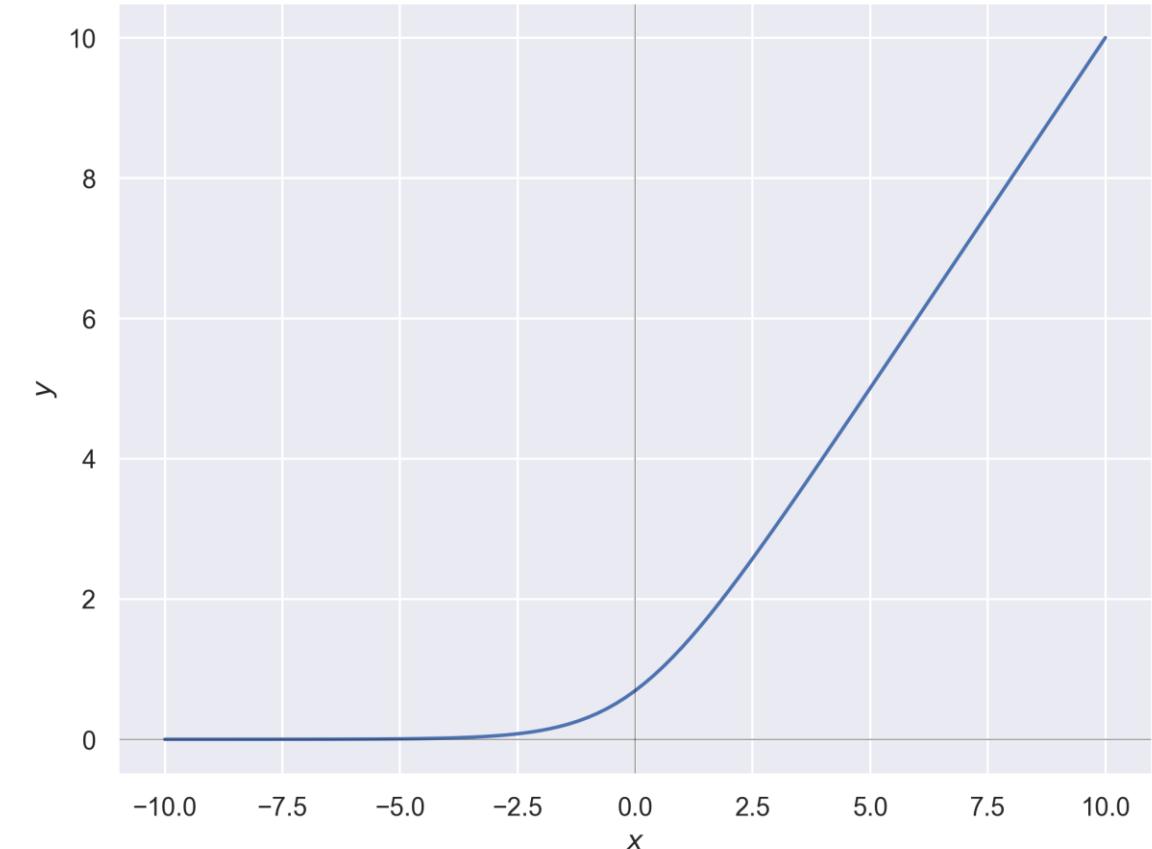


Další nelinearity

$$\text{ELU}(\textcolor{brown}{x}) = \begin{cases} \textcolor{brown}{x} & \text{pokud } x > 0 \\ \alpha \cdot e^{\textcolor{brown}{x}} - \alpha & \text{pokud } x \leq 0 \end{cases}$$

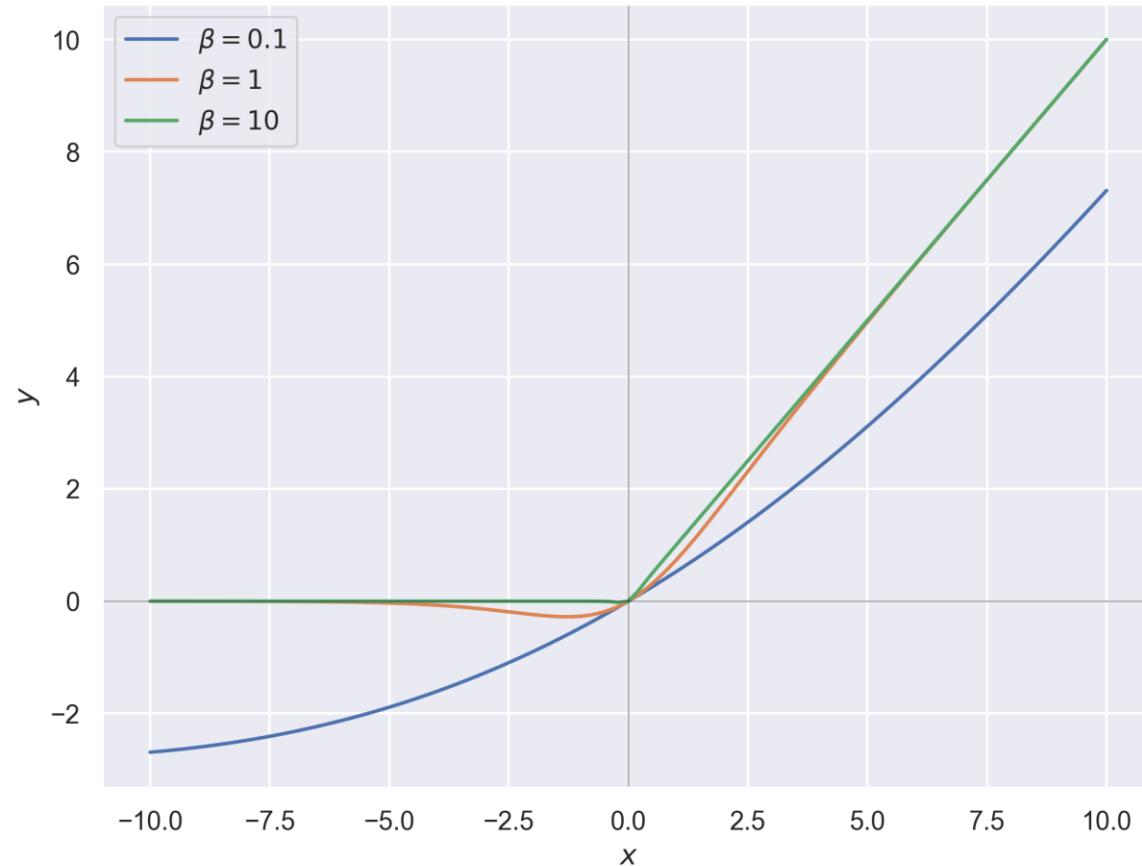


$$\text{softplus}(\textcolor{brown}{x}) = \log(1 + e^{\textcolor{brown}{x}})$$

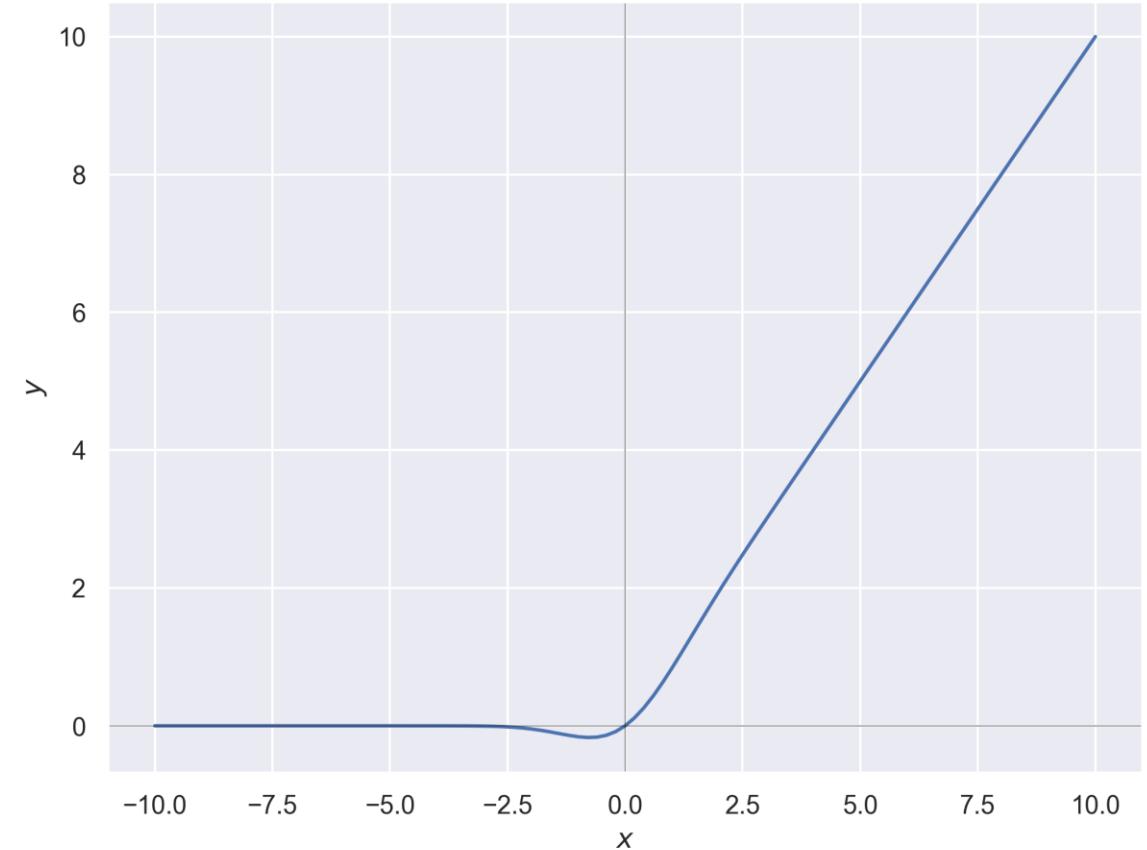


Další nelinearity

$$\text{swish}(x) = x \cdot \sigma(\beta \cdot x)$$



$$\text{GELU}(x) = x \cdot \text{CDF}_{\mathcal{N}(0,1)}(x)$$



Další nelinearity

Model	ResNet	WRN	DenseNet
LReLU	94.2	95.6	94.7
PReLU	94.1	95.1	94.5
Softplus	94.6	94.9	94.7
ELU	94.1	94.1	94.4
SELU	93.0	93.2	93.9
GELU	94.3	95.5	94.8
ReLU	93.8	95.3	94.8
Swish-1	94.7	95.5	94.8
Swish	94.5	95.5	94.8

Table 4: CIFAR-10 accuracy.

Model	Top-1 Acc. (%)			Top-5 Acc. (%)		
LReLU	73.8	73.9	74.2	91.6	91.9	91.9
PReLU	74.6	74.7	74.7	92.4	92.3	92.3
Softplus	74.0	74.2	74.2	91.6	91.8	91.9
ELU	74.1	74.2	74.2	91.8	91.8	91.8
SELU	73.6	73.7	73.7	91.6	91.7	91.7
GELU	74.6	-	-	92.0	-	-
ReLU	73.5	73.6	73.8	91.4	91.5	91.6
Swish-1	74.6	74.7	74.7	92.1	92.0	92.0
Swish	74.9	74.9	75.2	92.3	92.4	92.4

Table 6: Mobile NASNet-A on ImageNet, with 3 different runs ordered by top-1 accuracy. The additional 2 GELU experiments are still training at the time of submission.

Model	Top-1 Acc. (%)			Top-5 Acc. (%)		
LReLU	79.5	79.5	79.6	94.7	94.7	94.7
PReLU	79.7	79.8	80.1	94.8	94.9	94.9
Softplus	80.1	80.2	80.4	95.2	95.2	95.3
ELU	75.8	79.9	80.0	92.6	95.0	95.1
SELU	79.0	79.2	79.2	94.5	94.4	94.5
GELU	79.6	79.6	79.9	94.8	94.8	94.9
ReLU	79.5	79.6	79.8	94.8	94.8	94.8
Swish-1	80.2	80.3	80.4	95.1	95.2	95.2
Swish	80.2	80.2	80.3	95.0	95.2	95.0

Table 7: Inception-ResNet-v2 on ImageNet with 3 different runs. Note that the ELU sometimes has instabilities at the start of training, which accounts for the first result.

Další nelinearity

Model	ResNet	WR	%)	Top-5 Acc. (%)
LReLU	94.2	95.	79.6	94.7 94.7 94.7
PReLU	94.1	95.	80.1	94.8 94.9 94.9
Softplus	94.6	94.	80.4	95.2 95.2 95.3
ELU	94.1	94.	80.0	92.6 95.0 95.1
SELU	93.0	93.	79.2	94.5 94.4 94.5
GELU	94.3	95.	79.9	94.8 94.8 94.9
ReLU	93.8	95.	79.8	94.8 94.8 94.8
Swish-1	94.7	95.	80.4	95.1 95.2 95.2
Swish	94.5	95.	80.3	95.0 95.2 95.0

Table 4: CIFAR-10



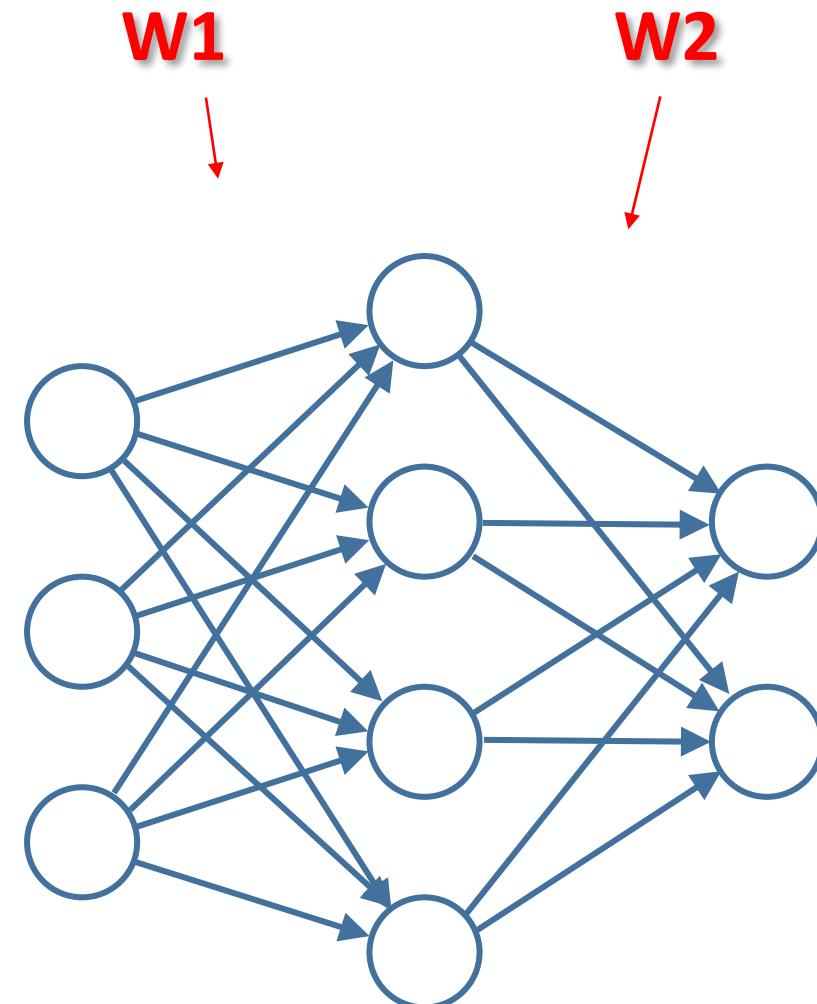
et-v2 on ImageNet
Note that the ELU
lies at the start of
for the first result.

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 sítě
 - 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

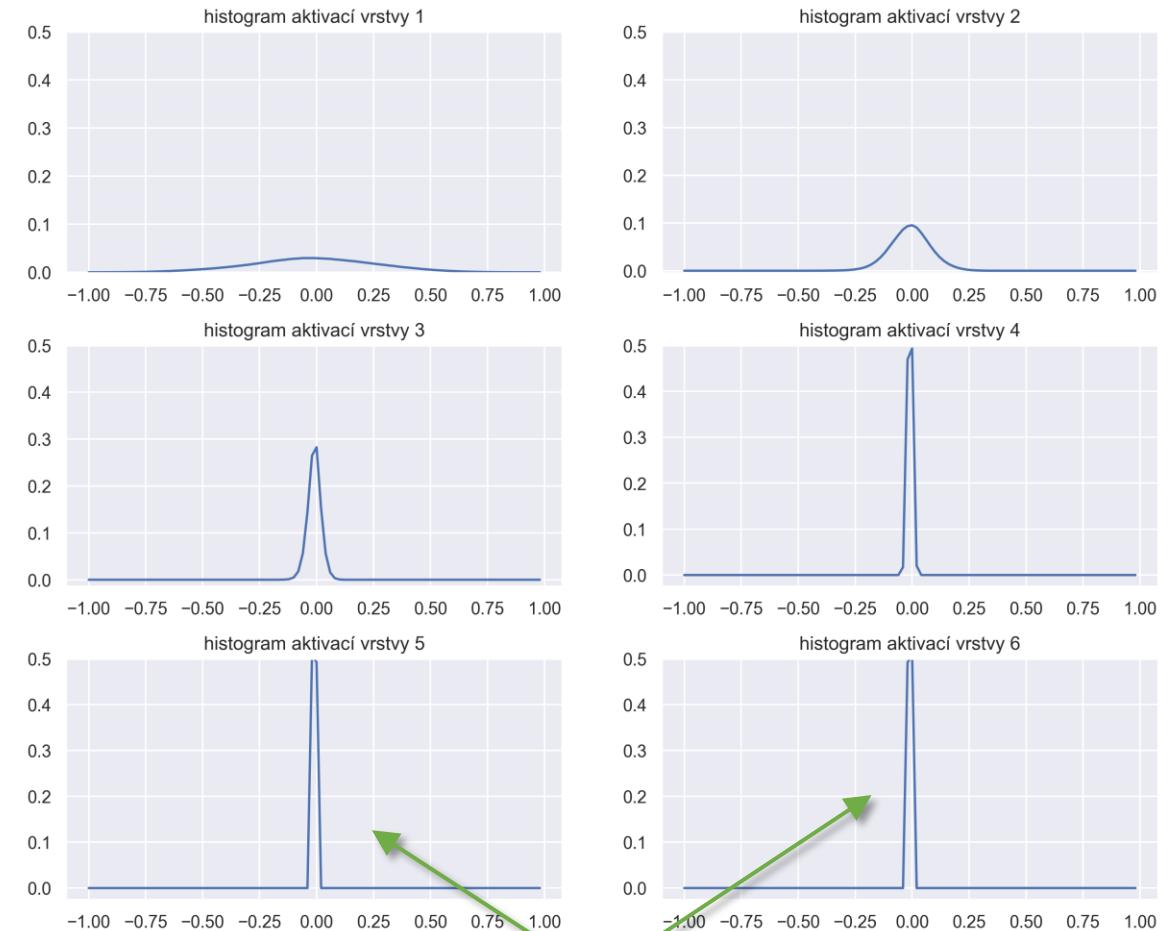
- 6-vrstvý MLP typu linear+tanh
- histogram hodnot aktivací skrytých vrstev
- data jsou CIFAR10 vydělená 255

```
dims = [3072, 1024, 1024, 1024, 1024, 1024, 1024]
weights = [
    0.01 * np.random.randn(dims[i-1], dims[i])
    for i in range(1, len(dims))]
]

bins = np.linspace(-1., 1., 101)
histograms = [np.zeros(100) for w in weights]

for t in range(10):
    i = np.random.choice(50000, size=256, replace=False)
    x = x_train[i]
    for w, h in zip(weights, histograms):
        x = tanh(np.dot(x, w))
        h += np.histogram(x.ravel(), bins=bins)[0]
```

$$w_i = 0.01 * np.random.randn(dim_in, dim_out)$$



nízké aktivace → lineární oblast tanh → příliš lineární model

Příliš velké váhy

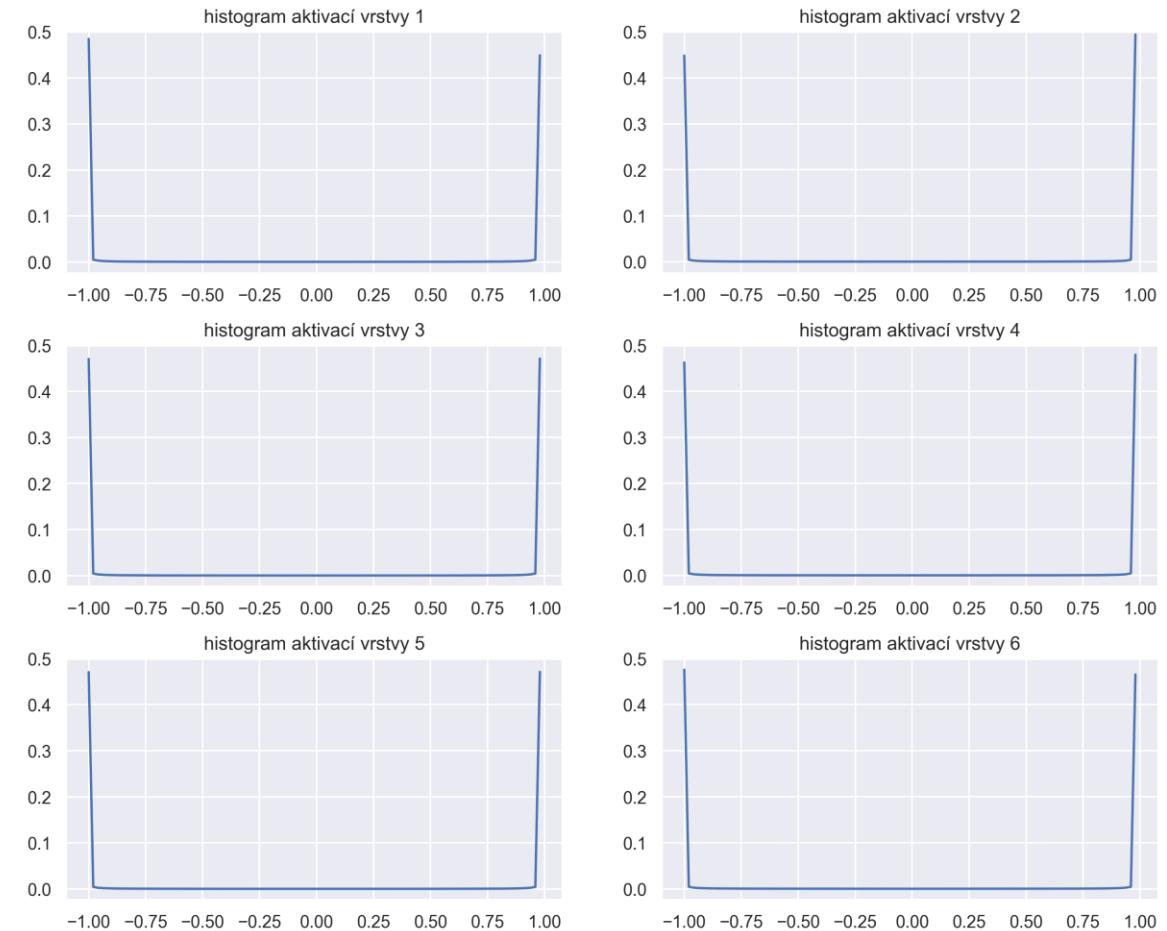
- 6-vrstvý MLP typu linear+tanh
- histogram hodnot aktivací skrytých vrstev
- data jsou CIFAR10 vydělená 255

```
dims = [3072, 1024, 1024, 1024, 1024, 1024, 1024]
weights = [
    1.00 * np.random.randn(dims[i-1], dims[i])
    for i in range(1, len(dims))
]

bins = np.linspace(-1., 1., 101)
histograms = [np.zeros(100) for w in weights]

for t in range(10):
    i = np.random.choice(50000, size=256, replace=False)
    x = x_train[i]
    for w, h in zip(weights, histograms):
        x = tanh(np.dot(x, w))
        h += np.histogram(x.ravel(), bins=bins)[0]
```

$$w_i = \boxed{1.00} * np.random.randn(dim_in, dim_out)$$



všude příliš saturovaná aktivace → mizející gradient

Xavier / Glorot inicializace

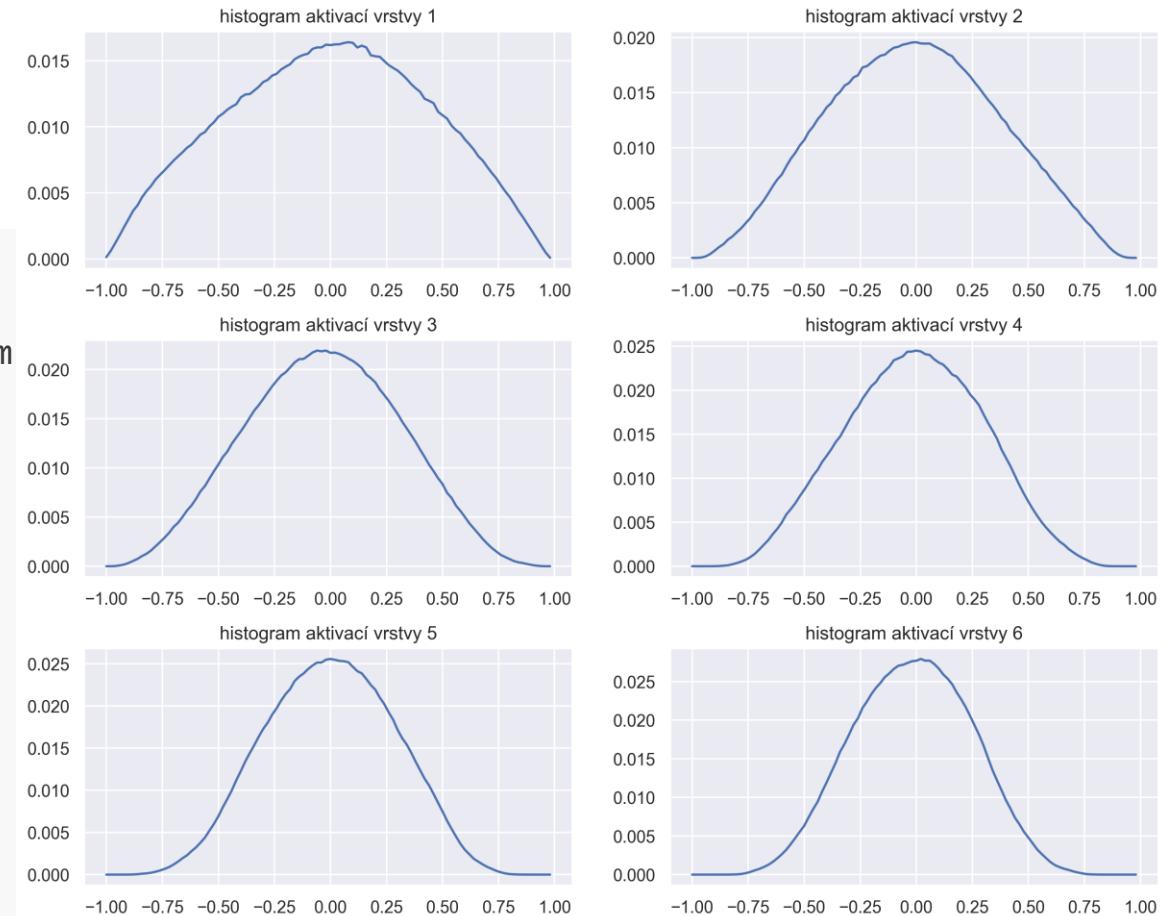
- 6-vrstvý MLP typu linear+tanh
- histogram hodnot aktivací skrytých vrstev
- data jsou CIFAR10 vydělená 255

```
dims = [3072, 1024, 1024, 1024, 1024, 1024, 1024]
weights = [
    1 / np.sqrt(dims[i-1]) * np.random.randn(dims[i-1], dim
    for i in range(1, len(dims)))
]

bins = np.linspace(-1., 1., 101)
histograms = [np.zeros(100) for w in weights]

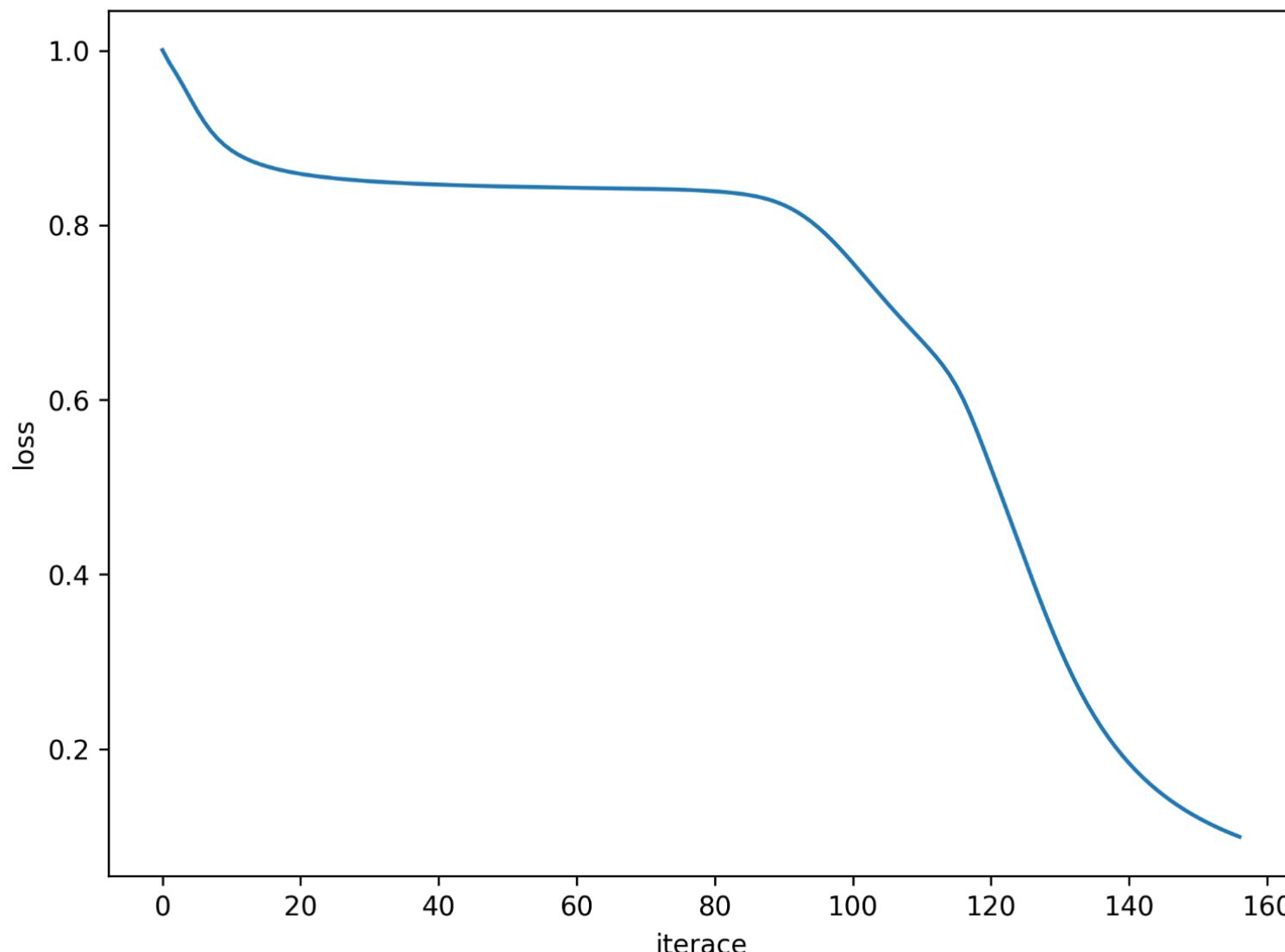
for t in range(10):
    i = np.random.choice(50000, size=256, replace=False)
    x = x_train[i]
    for w, h in zip(weights, histograms):
        x = tanh(np.dot(x, w))
        h += np.histogram(x.ravel(), bins=bins)[0]
```

$$W_i = \frac{1}{\sqrt{\text{fan_in}}} * \text{np.random.randn}(\text{dim_in}, \text{dim_out})$$



rozložení aktivací je rovnoměrnější

Špatná inicializace a její vliv na průběh lossu



Aktivace & Inicializace

- Velmi důležité pro optimální učení sítě
- Nejjednodušší zkusit ReLU → když funguje, lze vyzkoušet i cool hipster alternativy
- U ReLU namísto Glorot inicializace vhodnější **He**:

```
Wi = np.random.randn(dim_in, dim_out) / np.sqrt(dim_in / 2.)
```

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

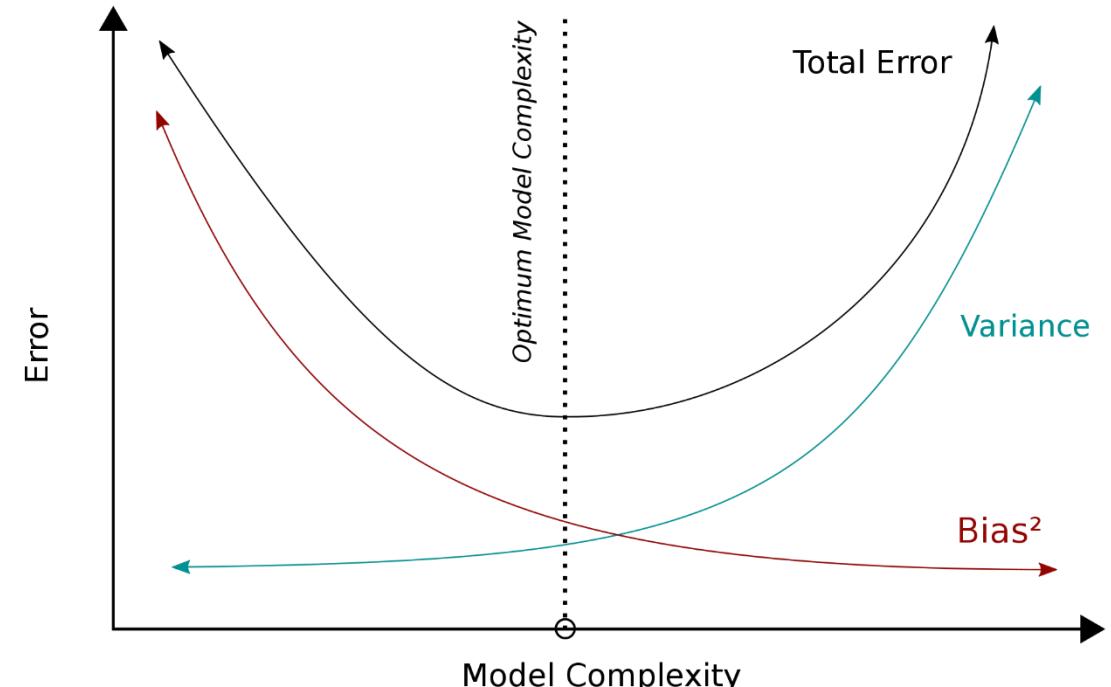
Regularizace

Bias/variance trade-off, L2 weight decay

Bias vs variance

- Bias zahrnuje předpoklady modelu
- Velký bias
 - = nedoučení (underfitting)
 - model nezachycuje důležité závislosti
 - nízká úspěšnost na trénovacích datech
- Lze odhadnout tak, že stanovíme horní mez úspěšnosti pro daný problém
- Typicky se malý vzorek datasetu nechá vyhodnotit člověkem. Skóre potom slouží jako benchmark
- Pokud úspěšnost modelu << úspěšnost člověka, pak lze usuzovat na příliš velký bias

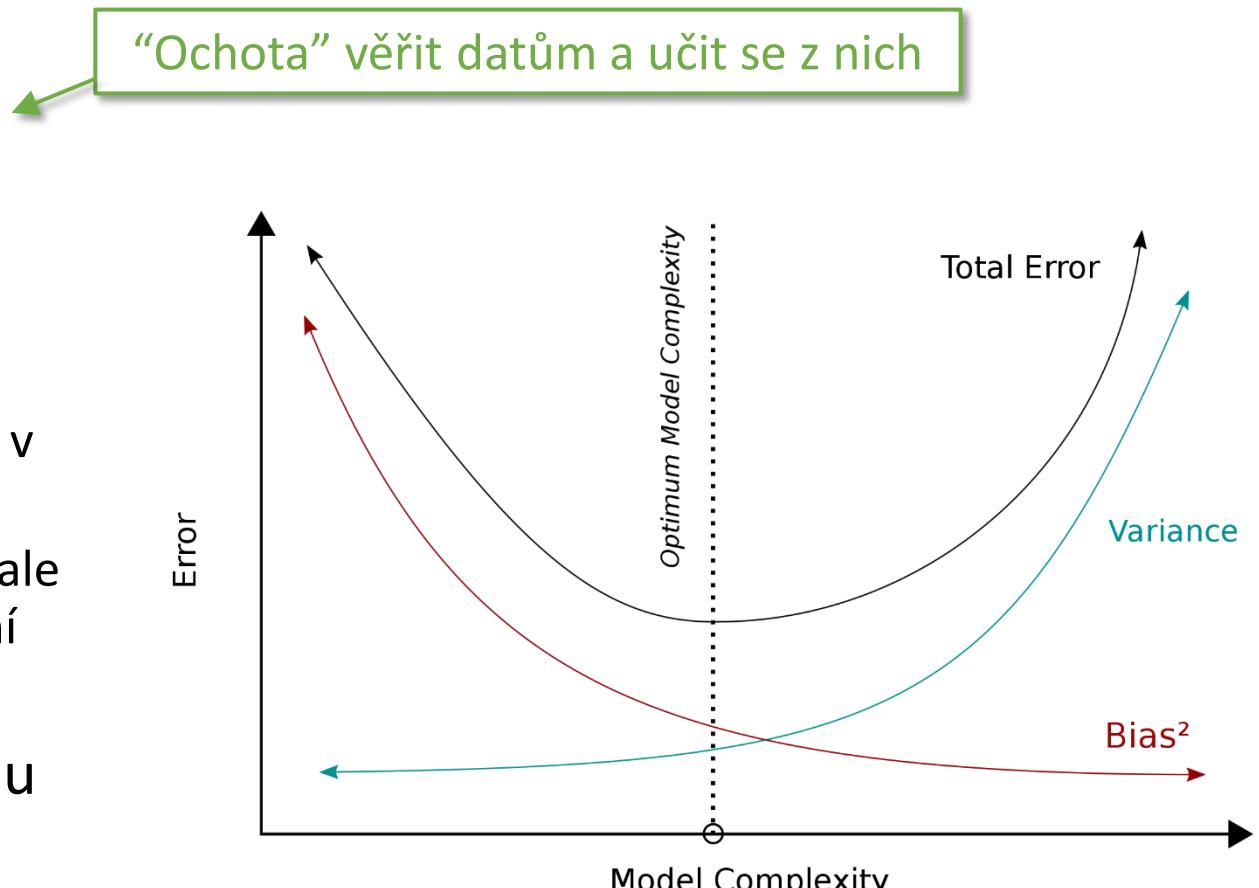
“Předsudky” vůči datům



obrázek: [Wikipedia](#)

Bias vs variance

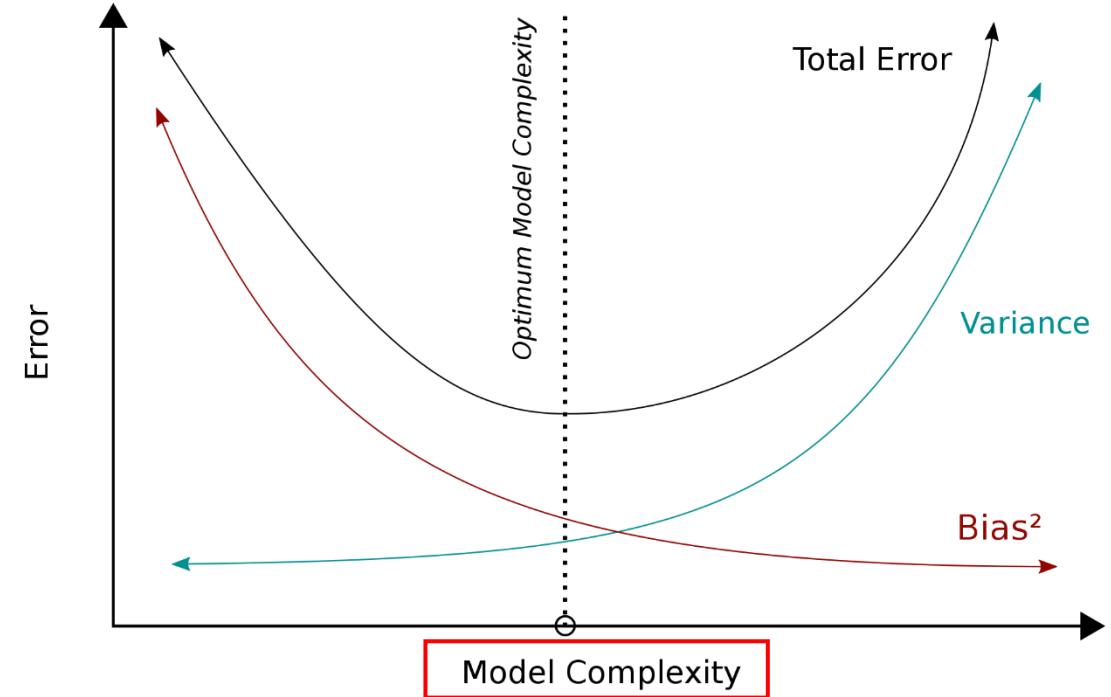
- Variance vyjadřuje citlivost modelu na malé změny v trénovací množině
- Velká variance
 - = přeучení (overfitting)
 - velká změna v úspěšnosti při malé změně v trénovacích datech
 - vysoká úspěšnost na trénovacích datech, ale malá na validačních = špatná generalizační schopnost
- Příliš velkou varianci lze poznat z rozdílu úspěšnosti na trénovací a validační/testovací množině



obrázek: [Wikipedia](#)

Bias vs variance trade-off

- Bias a variance dohromady tvoří chybovost modelu na neviděných datech
- Obě veličiny souvisí s komplexitou modelu
- Obvykle platí
 - jednodušší model = vyšší bias
 - složitější model = vyšší variance
- **Lze ladit regularizací**
- U neurosítí trénovaných na velkých datasetech lze často dosáhnout malého biasu i variance



obrázek: [Wikipedia](#)

Škálování vah aditivní regularizací

- Spočívá v penalizaci příliš vysokých vah zavedením dodatečného členu do lossu
- Menší váhy = jednodušší model (viz přednášku 1 o lineárních modelech)

Celkový loss na trénovacích datech X
a s parametry θ včetně regularizace

$$L(X, \theta) = \frac{1}{N} \sum_{n=1}^N L_n(x_n, \theta)$$

Loss vypočtený z dat (např.
softmax cross entropy)

Loss vypočtený z velikosti
parametrů

$$+ \lambda \cdot R(\theta)$$

- λ je hyperparametr 😔
- často jako weight decay*
- ovlivňuje sílu regularizace
- typicky malé číslo $\lambda \approx 10^{-3}$

Nejčastěji L2 regularizace

$$R(w) = \sum_{k,d} w_{k,d}^2$$

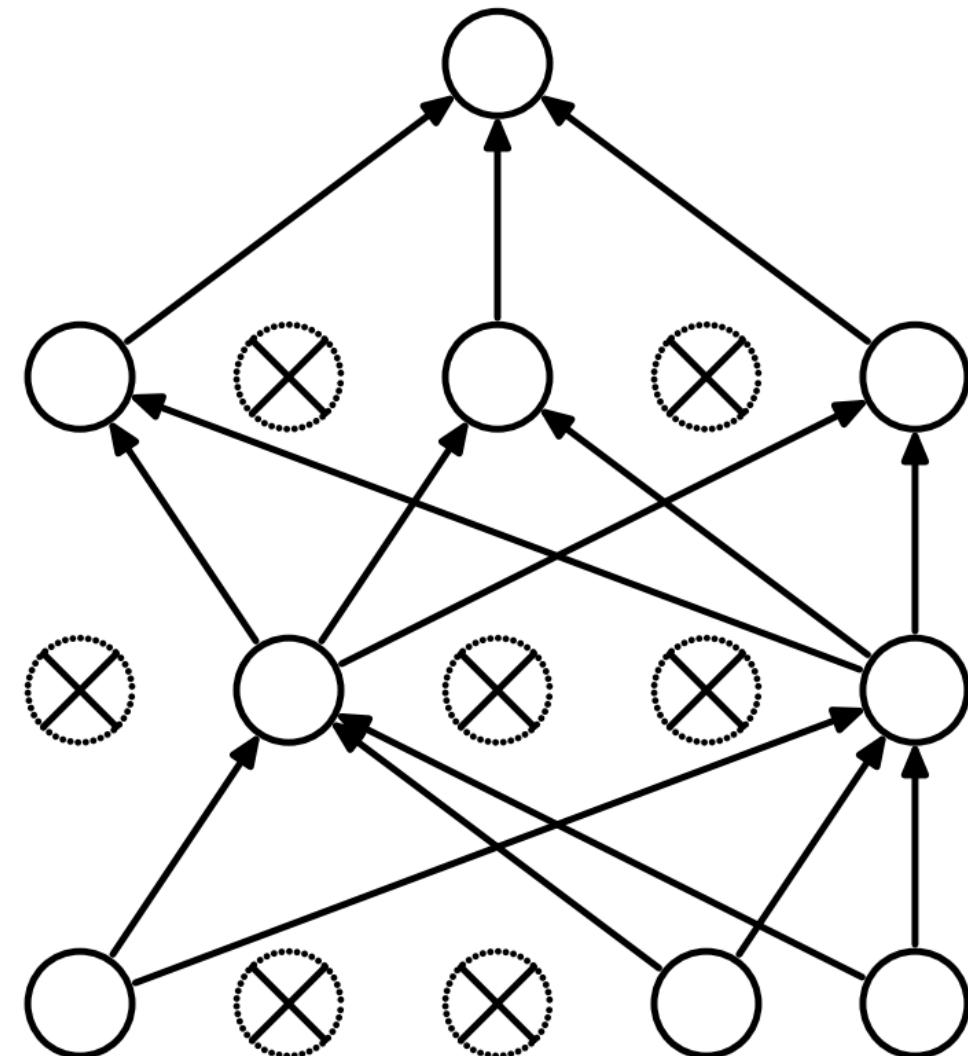
Regularizace: dropout

Dropout

- Náhodně nastavuje výstupy na nulu
- Např. s pravděpodobností 40 %:

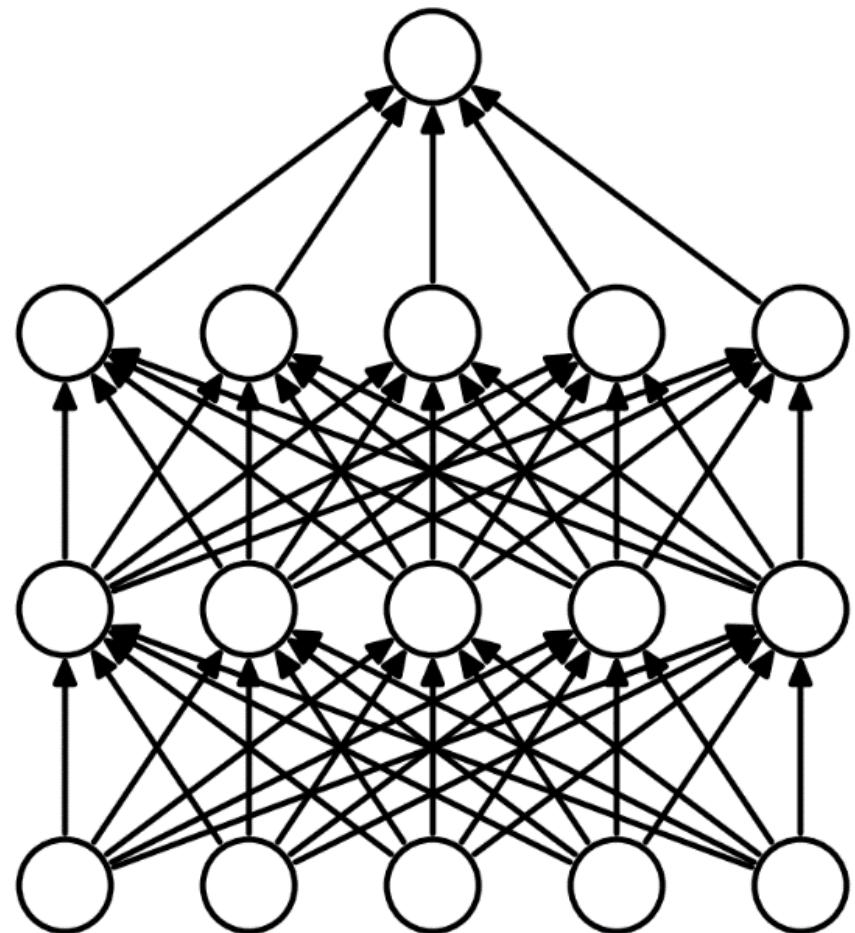
```
1 scores = np.dot(x, w) + b
2 hidden = np.maximum(0., scores)
3 mask = np.random.rand(*hidden.shape) < 0.4
4 hidden[mask] = 0.
```

- slouží jako regularizace → prevence overfitu

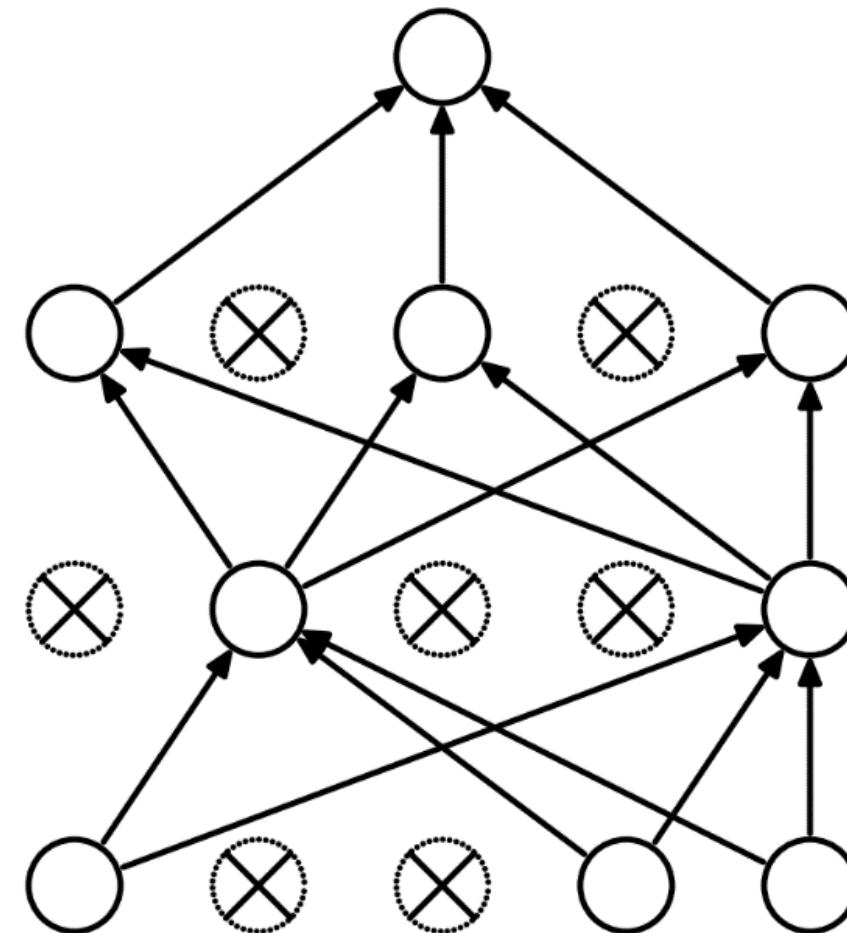


obrázek: <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>

Dropout



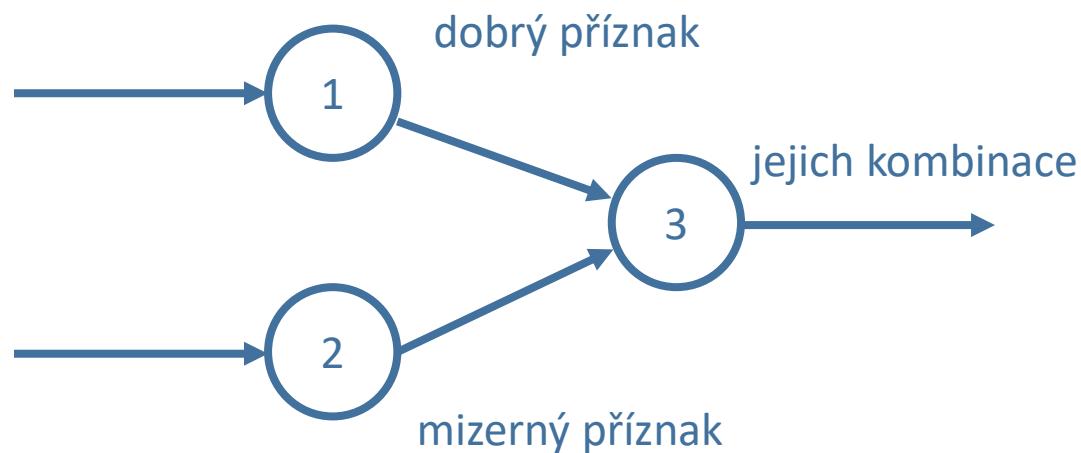
(a) Standard Neural Net



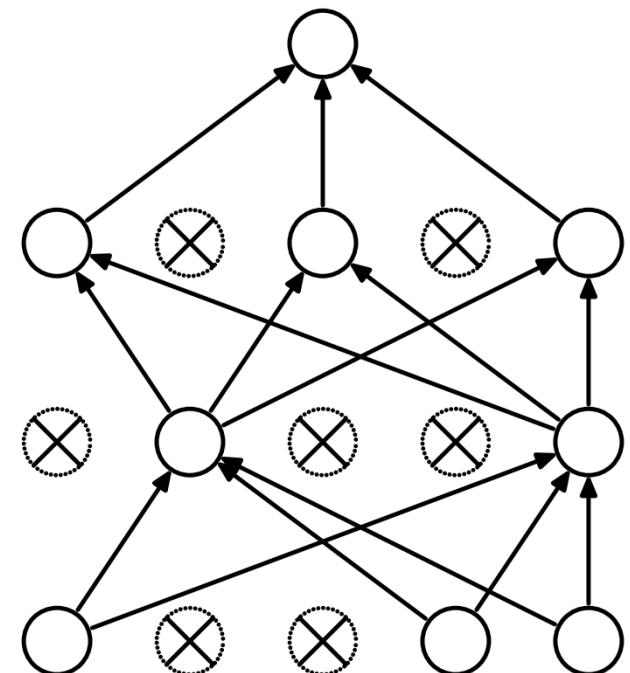
(b) After applying dropout.

Dropout

- Nutí síť vytvářet robustní a redundantní příznaky
 - náhodně vypadávají → tlak, aby **všechny** dobře reprezentovaly



- Model ensemble
 - de facto vytváří kombinaci více modelů, které sdílejí váhy
 - každá maska reprezentuje jednu síť
 - jedna síť = jeden trénovací vektor



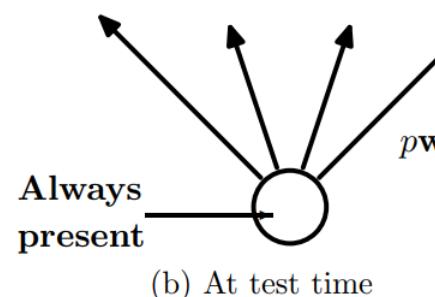
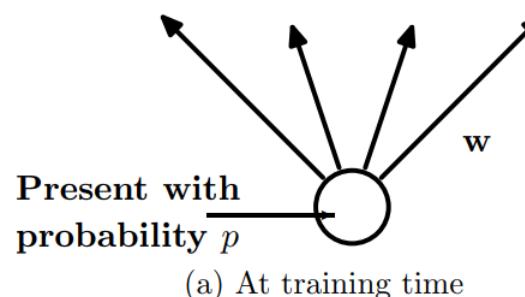
Dropout v trénovací a testovací fázi

- Model ensemble

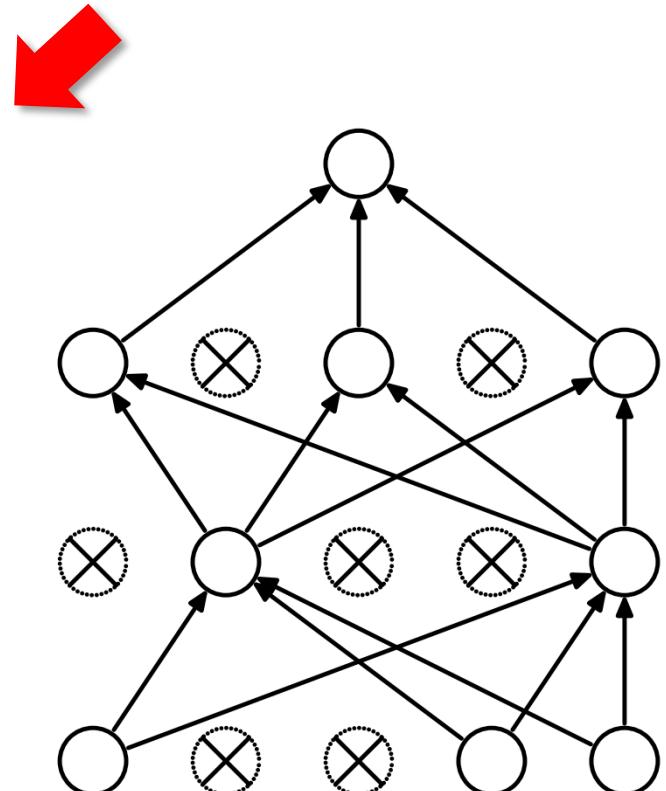
- teoreticky v test. fázi forward např. 100x a zprůměrovat → pomalé
- chceme pouze jeden forward průchod → **v testu se dropout nedělá**

- Problém

- $s = w_1x_1 + w_2x_2 + w_3x_3$
- $p = 1/3 \rightarrow$ v průměru jedno w_i vypadne
- průměrná hodnota s bude o $1/3$ nižší \rightarrow tomu se přizpůsobí váhy a aktivace
- bez dropoutu v testovací fázi je pak příliš velká “energie” výstupu do další vrstvy



Rozdílné chování v trénovací a testovací fázi!



Dropout v trénovací a testovací fázi

- Režim 1: základní (direct) dropout
 - **v testovací fázi** vynásobíme výstup dropout pravděpodobností $1 - p$
- Režim 2: inverted dropout
 - vyřešíme už **v trénovací fázi**, kde výstup vydělíme $1 - p$, aby měl stejnou “energií”, jako kdyby žádný dropout nebyl

```
def direct_dropout_forward(  
    inpt: np.ndarray,  
    training: bool = True,  
    p_drop: float = 0.4  
):  
    if training:  
        mask = np.random.rand(*inpt.shape) < p_drop  
        output = inpt.copy()  
        output[mask] = 0.  
    else:  
        output = (1. - p_drop) * inpt  
    return output
```

```
def inverted_dropout_forward(  
    inpt: np.ndarray,  
    training: bool = True,  
    p_drop: float = 0.4  
):  
    if training:  
        mask = np.random.rand(*inpt.shape) < p_drop  
        output = inpt / (1. - p_drop)  
        output[mask] = 0.  
    else:  
        output = inpt  
    return output
```

- testovací fáze pak nemusí být upravována
- škáluje → vhodné použít ještě s další regularizací

Jak moc dropoutu?



- Optimální většinou 40-60 %, ale není pravidlem 😞
- Nejlépe nahlížet jako na hyperparametr → křížová validace
- Při správném nastavení obvykle přinese cca 2 % accuracy navíc, někdy ale nic či dokonce zhorší
- Diskuze např. zde:
https://www.reddit.com/r/MachineLearning/comments/3oztvk/why_50_when_using_dropout/
<https://pgaleone.eu/deep-learning/regularization/2017/01/10/anaysis-of-dropout/>

Regularizace: normalizace dávky

Batch normalization, SELU

Batch normalizace (BN)

- [Ioffe, Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#)
- Chceme, aby vstup do vrstvy měl pro každou dávku podobné rozložení hodnot, aby se parametry neustále nemusely přizpůsobovat měnícím se datům
- Obtížné zajistit inicializaci a aktivacemi / nelinearitami
- Co prostě výstup vrstvy normalizovat?
- Např. na nulový průměr a std. odchylku 1:

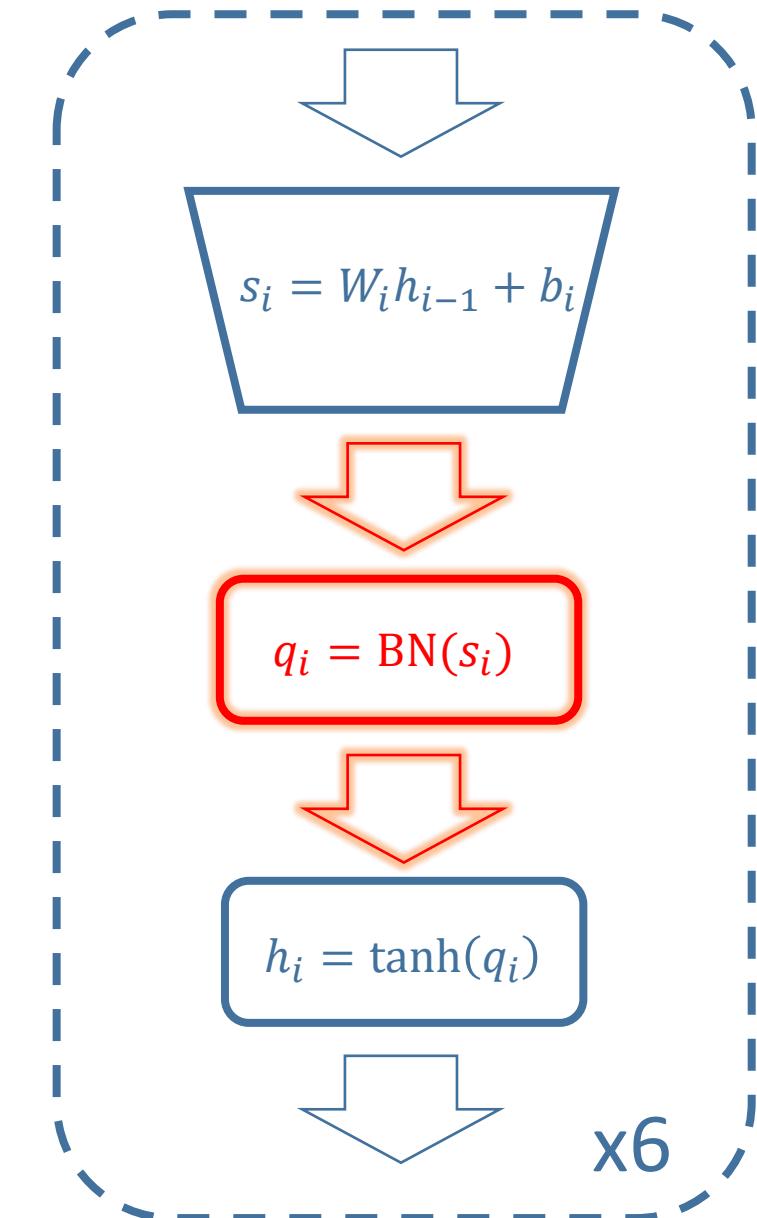
$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x]}}$$

kde

$E[x]$ je střední hodnota

$\text{Var}[x]$ je rozptyl

- operace je diferencovatelná! → lze počítat gradient
 - <https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>



Dopředný průchod batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

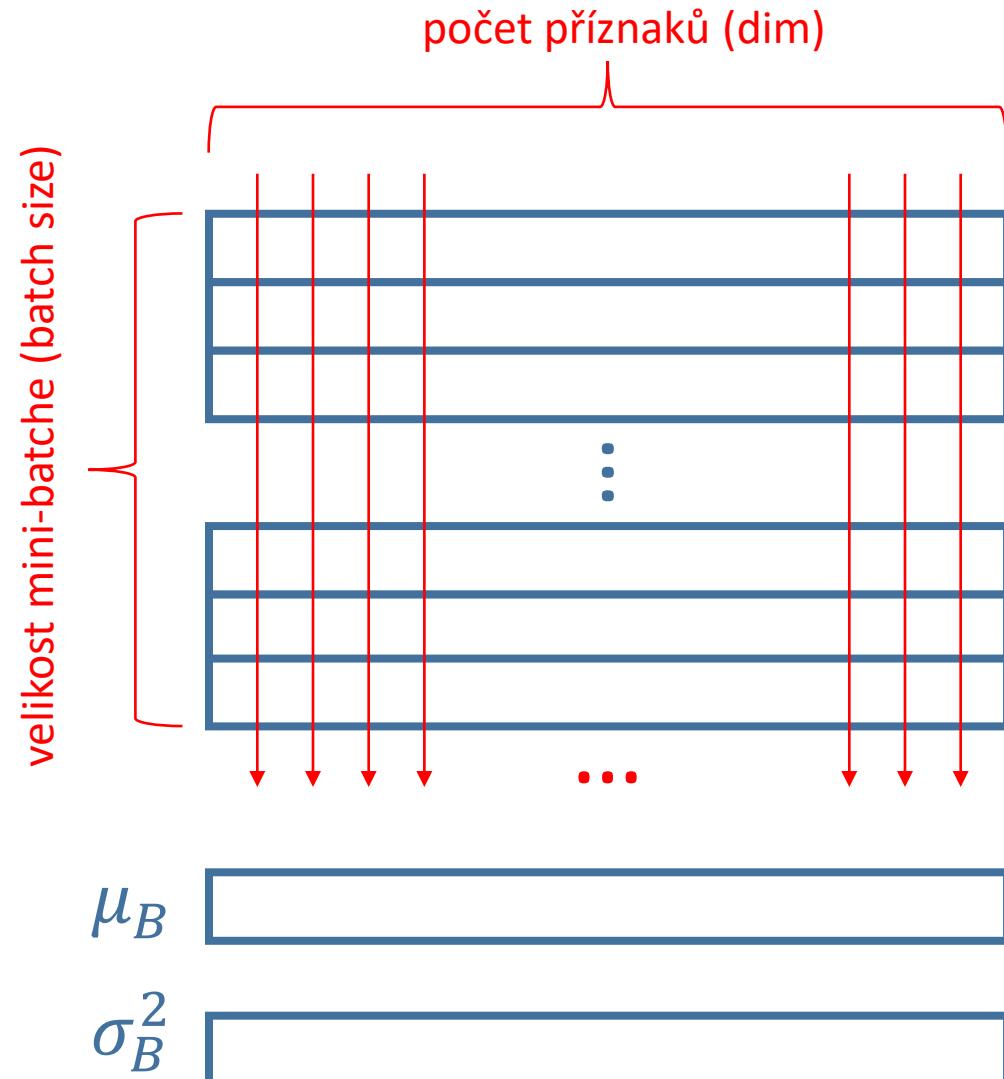
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

naučitelné parametry γ a β , které umožňují nastavit si statistiky výstupu tak, jak to síti vyhovuje



Kam umístit batch normalizaci?

- Umístit BN před nebo po nelinearitě?
- Doporučuje se různě
- Např. dle cs231n 2016/lec 5/slide 67 před
- V původním BN článku rovněž před
- **Dle výsledků však lepší po**
- Dává smysl: snažíme se, aby vstup do každé další vrstvy měl požadované rozložení, ale ReLU po BN zahodí záporné hodnoty

Výsledky pro RELU na ImageNet:

Name	Accuracy	LogLoss	Comments
Before	0.474	2.35	As in paper
Before + scale&bias layer	0.478	2.33	As in paper
After	0.499	2.21	
After + scale&bias layer	0.493	2.24	

zdroj: <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md>

Trénování vs testování BN

- Podobně jako dropout, Batchnorm se chová rozdílně v trénování a v testování
- V testovací fázi se batch statistiky nepočítají
- Použije se naučený průměr a rozptyl z trénovacích dat
- Výpočet např. průměrováním se zapomínáním
- Nebo např. jedním průchodem natrénované sítě trénovacími daty

Proč batch normalizace funguje?

- Internal Covariate Shift
 - v původním článku, dnes spíše zavržované vysvětlení
 - s každou dávkou (batch) dat se mírně změní statistické rozložení hodnot (platí i pro aktivace vnitřních vrstev)
 - parametry vrstvy se těmto změnám neustále musí přizpůsobovat
 - BN toto napravuje a hodnoty standardizuje na nulový* průměr a jednotkový* rozptyl
 - optimalizací parametrů γ a β jednodušeji upravuje statistiky výstupů jednotlivých vrstev
- Vyhlazení povrchu optimalizované funkce (lossu)
 - [Santurkar et al.: How Does Batch Normalization Help Optimization?](#)
 - Hladký povrch = pokud půjdeme ve směru gradientu delší vzdálenost, hodnota lossu klesne
 - Nehladký povrch = hodnota lossu s vyšším krokem vzroste, pak klesne, apod.
 - Vyhlazením jsou gradienty spolehlivější → optimalizace stabilnější, lze použít vyšší learning rate
- Regularizační efekt
 - odhad průměru i rozptylu závisí na konkrétní dávce a jsou pokaždé trochu jiné
 - to způsobuje šum, který má regularizační efekt, podobně jako např. přidávání šumu do gradientu

Výhody a nevýhody batch normalizace

- 😊 obvykle zvyšuje úspěšnost
- 😊 urychluje trénování, lze vyšší learning rate
- 😊 snižuje potřebu dropout
- 😊 snižuje závislost na inicializaci → robustnější
- 😊 stabilní pokud batch size dostatečně velká
- 😢 nepříliš vhodná pro rekurentní sítě
- 😢 nic moc pro malé batche
- 😢 různé chování v train a test (bugy)
- 😢 zpomaluje
- 😢 také závisí na nelinearitě

BN and activations

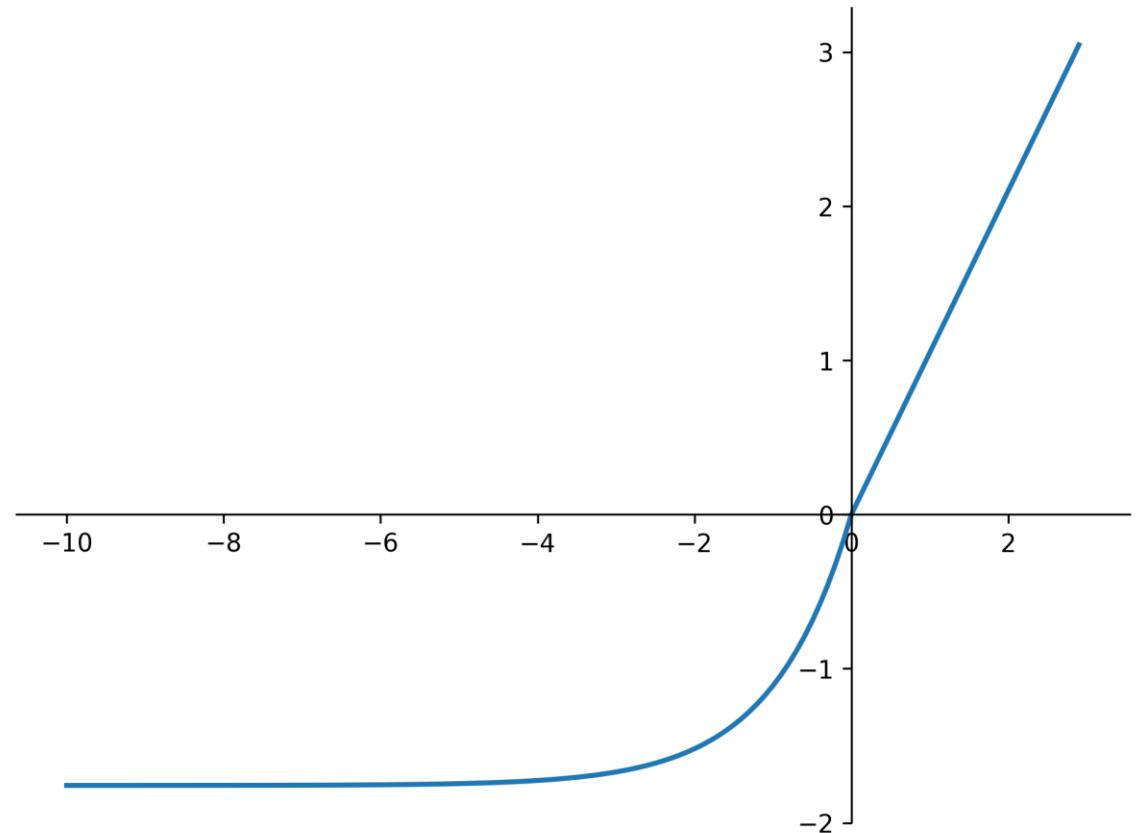
Name	Accuracy	LogLoss	Comments
ReLU	0.499	2.21	
RReLU	0.500	2.20	
PReLU	0.503	2.19	
ELU	0.498	2.23	
Maxout	0.487	2.28	
Sigmoid	0.475	2.35	
TanH	0.448	2.50	
No	0.384	2.96	

zdroj + další výsledky: <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md>

(Scaled) Exponential Linear Unit (ELU)

- Snaží se kombinovat lineární a exp aktivace
- Přibližuje výstupní hodnoty nulovému průměru
- Scaled exponential linear units (SELU)
 - [Klambauer et al.: “Self-Normalizing Neural Networks” \(2017\)](#)
 - Cílem dosáhnout $m=0$ a $std=1$
 - Při správném nastavení λ a α nahrazuje batch normalizaci! → navíc urychluje!
 - $\lambda = 1.0507009873554804934193349852946$
 $\alpha = 1.6732632423543772848170429916717$

$$\text{ELU}(x) = \lambda \cdot \begin{cases} x & \text{pokud } x > 0 \\ \alpha \cdot e^x - \alpha & \text{pokud } x \leq 0 \end{cases}$$



Scaled Exponential Linear Unit (SELU)

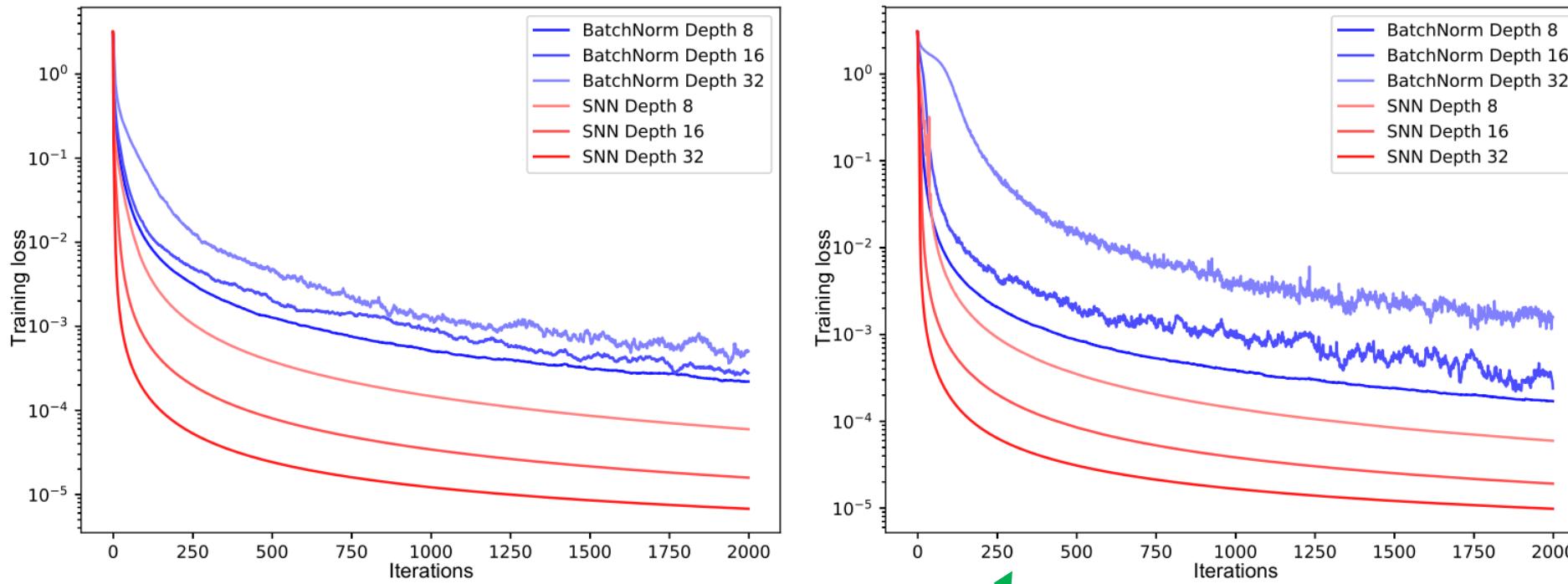
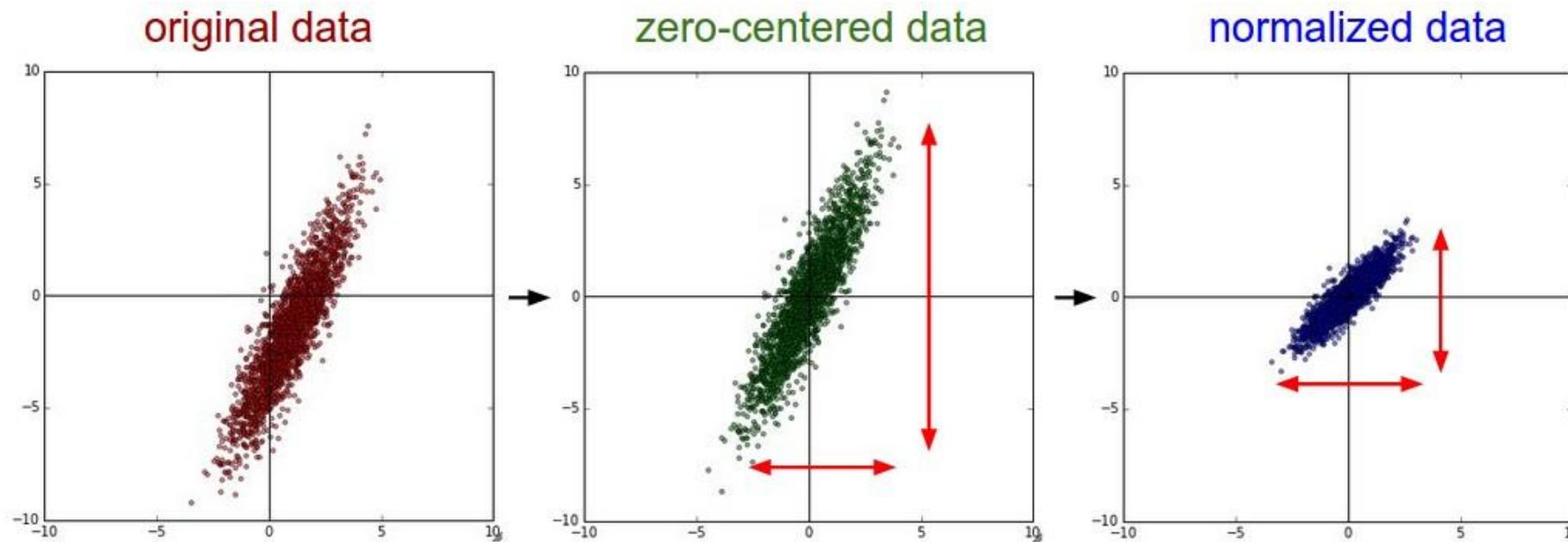


Figure 1: The left panel and the right panel show the training error (y-axis) for feed-forward neural networks (FNNs) with batch normalization (BatchNorm) and self-normalizing networks (SNN) across update steps (x-axis) on the **MNIST** dataset the **CIFAR10** dataset, respectively. We tested networks with 8, 16, and 32 layers and learning rate $1e-5$. FNNs with batch normalization exhibit high variance due to perturbations. In contrast, SNNs do not suffer from high variance as they are more robust to perturbations and learn faster.

zdroj: [Klambauer et al.: “Self-Normalizing Neural Networks” \(2017\)](#)

Regularizace: předzpracování a rozšiřování dat

Předzpracování dat



- Obvykle jen velmi omezené
- Cílem je end-to-end učení modelu
- Odečítání průměru
- Normalizace standardní odchylkou

Průměr a standardní odchylka by měly být spočítané pouze na trénovací sadě, tj. bez "koukání" na validační data

Předzpracování dat u konvolučních sítí

- U konvolučních sítí často používané konstantní hodnoty

- Odečtení průměrného pixelu

`out = rgb - mean_pixel`

kde `mean_pixel` je trojice [r, g, b]

často lze vidět konkrétní hodnoty [123.68, 116.779, 103.939],
které jsou průměrným pixelem na databázi ImageNet

- Méně časté: odečtení průměrného obrázku

`out = rgb - mean_image`

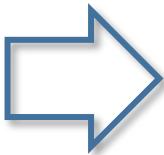
kde `mean_image` je 32x32x3

Umělé rozšiřování dat

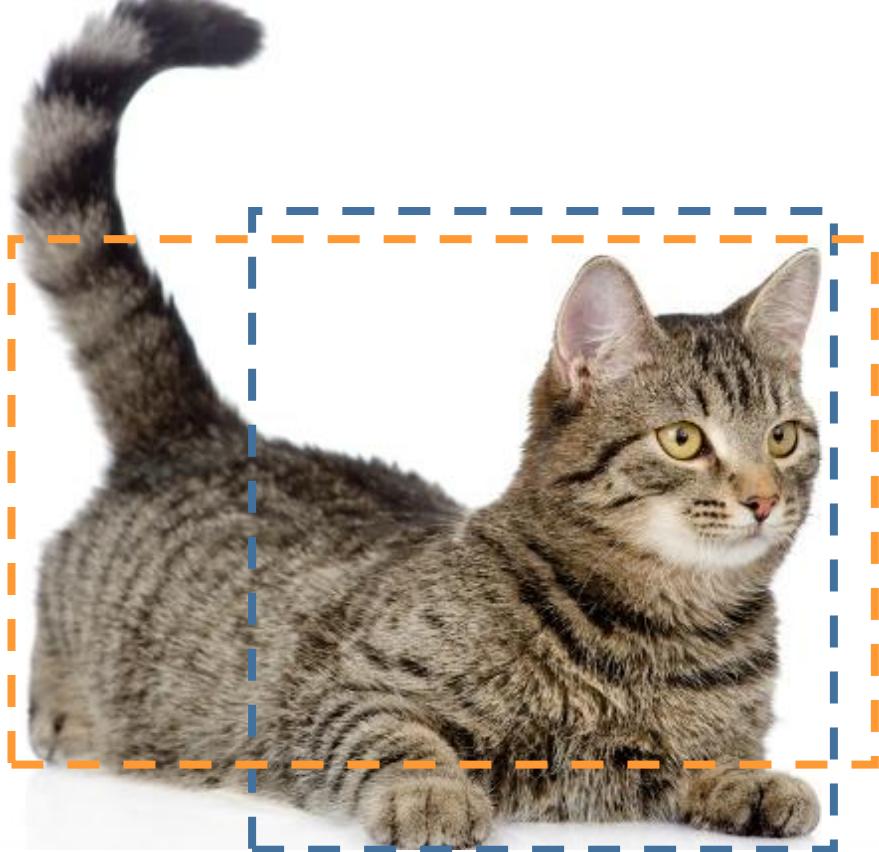
- Data augmentation
- U neurosítí téměř vždy platí: více dat = lepší výsledky
- Pokud reálná data nejsou, lze je “nafouknout” uměle, např. náhodnými transformacemi obrázků



Zrcadlení



Ořez



Další transformace

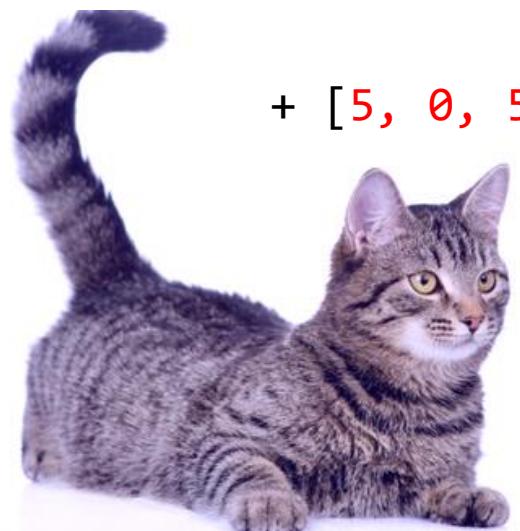
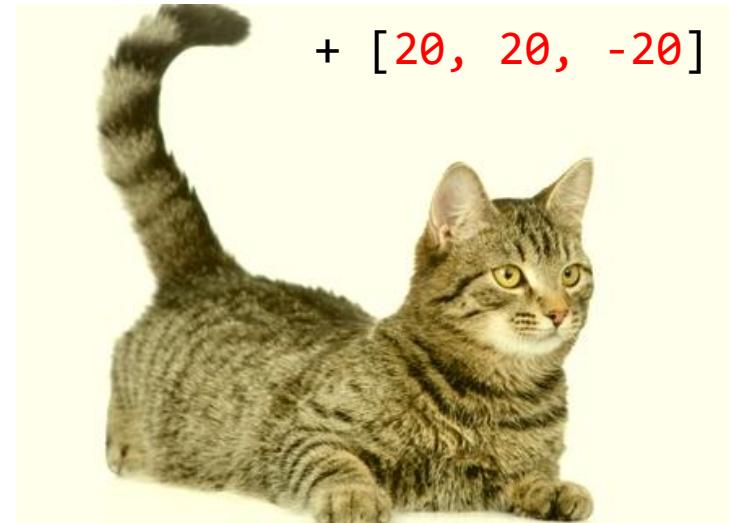
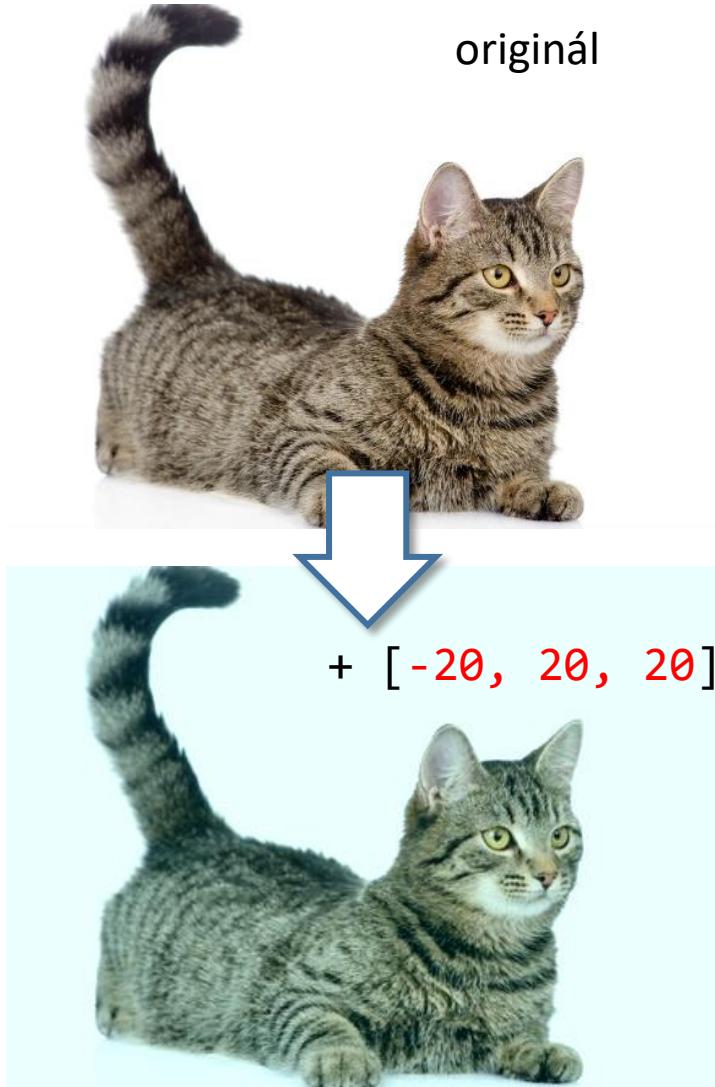
- Otočení
- Zkosení
- Lokální deformace a další
- Ne vždy ale pomáhají

Train augmentation

Name	Accuracy	LogLoss	Comments
Default	0.471	2.36	Random flip, random crop 128x128 from 144xN, N > 144
Drop 0.1	0.306	3.56	+ Input dropout 10%. not finished, 186K iters result
Multiscale	0.462	2.40	Random flip, random crop 128x128 from (144xN, - 50%, 188xN - 20%, 256xN - 20%, 130xN - 10%)
5 deg rot	0.448	2.47	Random rotation to [0..5] degrees.

<https://github.com/ducha-aiki/caffenet-benchmark/blob/master/Augmentation.md>

Posun barev



```
out = np.clip(rgb + np.array([r, g, b]), 0, 255).astype(np.uint8)
```

Úprava dat v PyTorch

- Pro obrazová data implementováno v doplňkové knihovně Torchvision

```
augment = transforms.Compose([
    transforms.Resize([256, 256]),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip()
])
prep = transforms.Compose([
    transforms.Resize([224, 224]),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
train_dataset = datasets.Imagefolder(
    root=data_folder,
    transform=transforms.Compose([augment, prep]))
)
valid_dataset = datasets.ImageFolder(
    root=data_folder,
    transform=prep
)
```

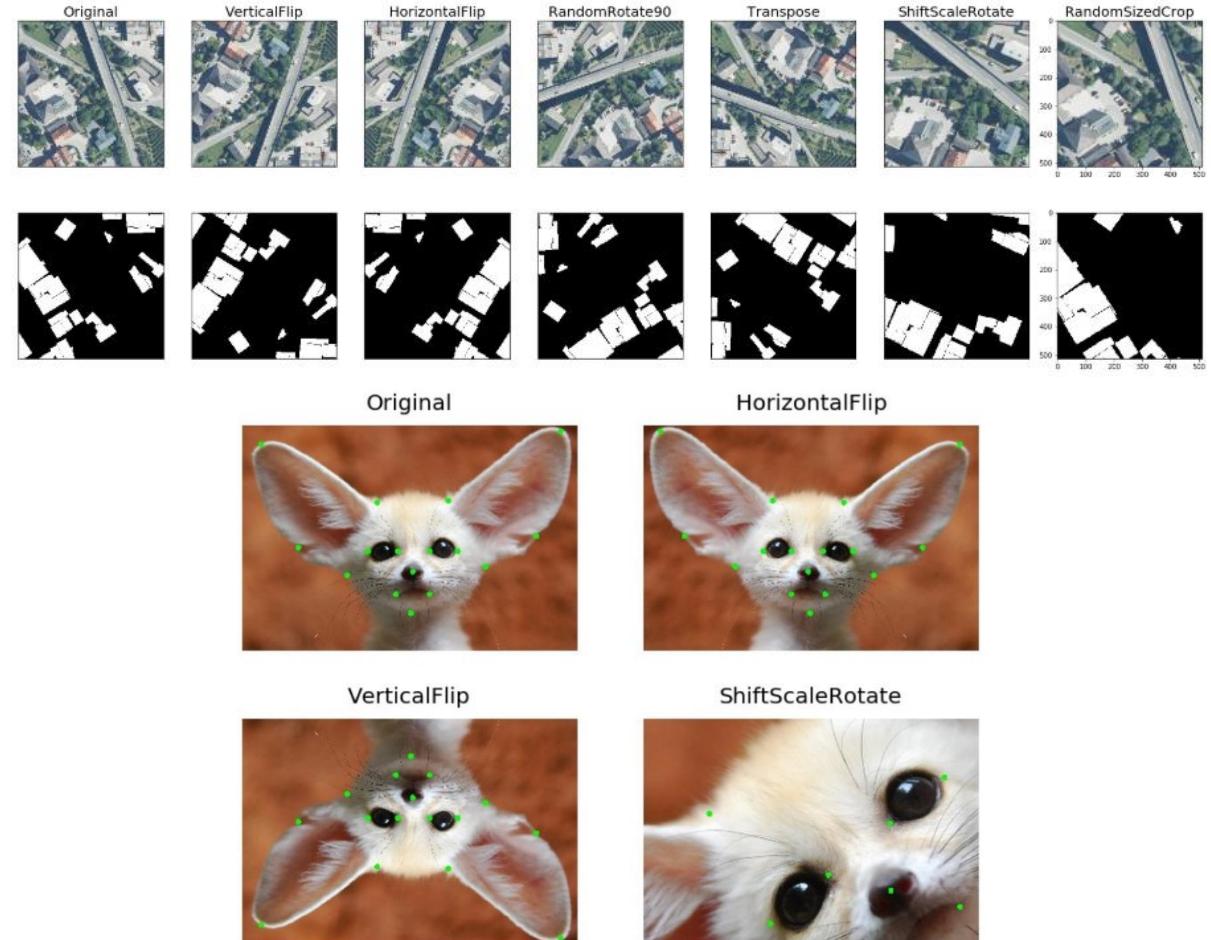
opět ImageNet hodnoty, pro obrázky s RGB pixely [0...1]



Úprava dat v PyTorch: albumentations



včetně anotací (segmentační masky, klíčové body, bounding boxy objektů, ...)



Trénování sítě v praxi

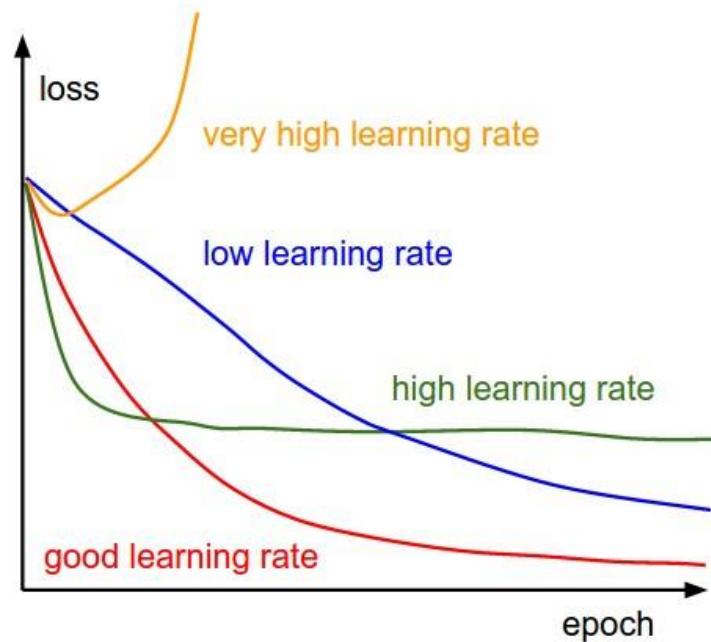
Volba sítě a nastavení trénování

- Hlavní idea: získat funkční baseline a až teprve poté zkoušet po jednom vylepšovat
- Ideální zvolit triviální síť, u které je riziko bugů minimální
- Jako optimizer nejlépe Adam, např. s learning rate (lr) $3 \cdot 10^{-4}$
- Pro začátek vypnout regularizaci a další vyfikundace jako lr scheduling apod.

Kontrola před (sanity check)

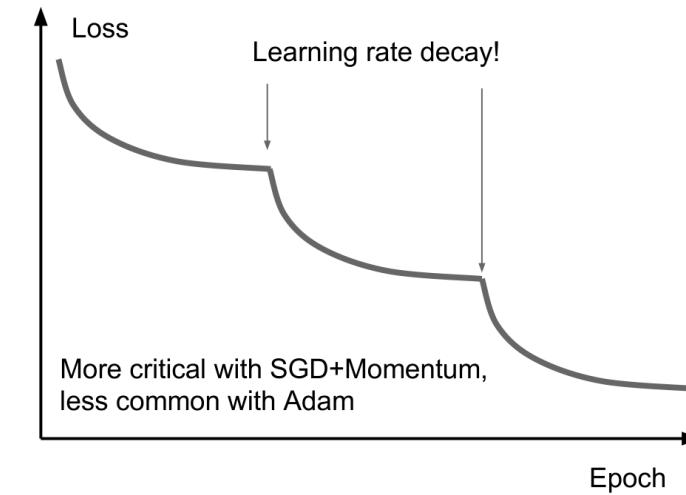
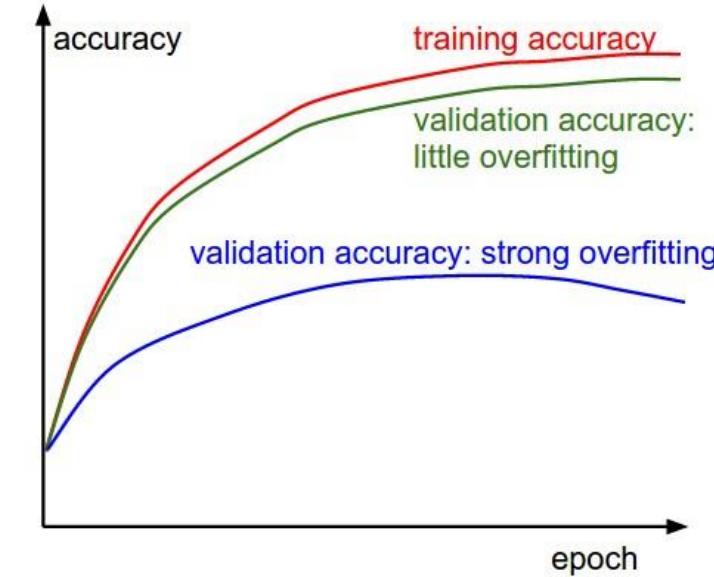
- Zkontrolovat poslední vrstvu, typ a numerický rozsah výstupů
 - Klasická chyba např. sigmoid/ReLU jako výstup, když je úloha regrese do reálných čísel $-\infty$ až $+\infty$
 - Nebo naopak nenormalizovaná lineární skóre, když výstupem má být pravděpodobnost
- Zkontrolovat hodnotu lossu na první dátce učení ještě před updatem parametrů
 - Po inicializaci by predikce měly být náhodné
 - Tomu by měla odpovídat i hodnota lossu
 - Např.: CIFAR-10 klasifikace do 10 tříd → výstupní vektor pravděpodobností je v průměru $\mathbf{q}_n \approx [0.1, \dots, 0.1]^T$ a proto SCE loss by měl začínat kolem $l_n = -\log q_{n,y_n} \approx 2.302$
- Zkusit overfit na jedné dátce dat
 - Naučit jednu dátku (batch) na hodnotu lossu 0 nebo přesnost 100 %
 - Kontrola, zda “pipeline” funguje a zda je model schopný se učit

Trénování



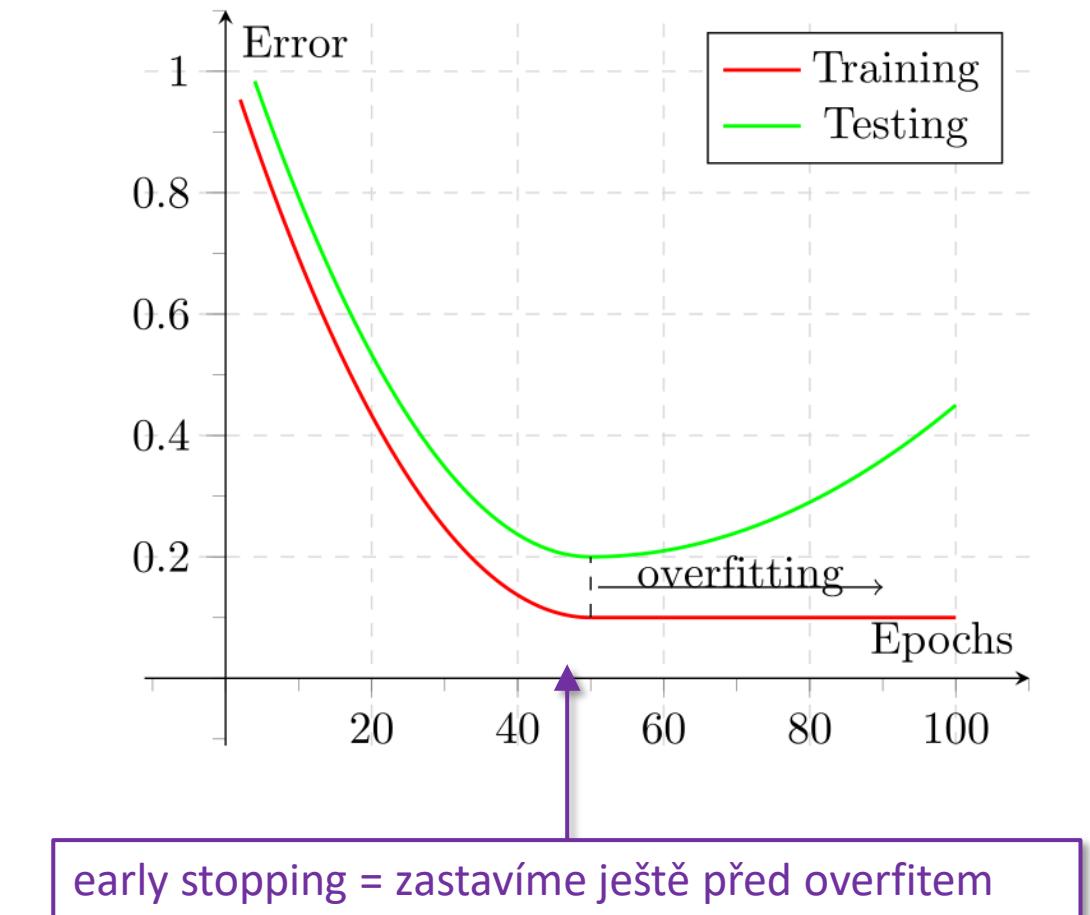
- Monitorovat hodnotu lossu a podle toho nastavit lr
- Nebo lze použít automatické hledání lr
- Pokud funguje, zkusit lr decay

obrázky: <https://cs231n.github.io/neural-networks-3/>



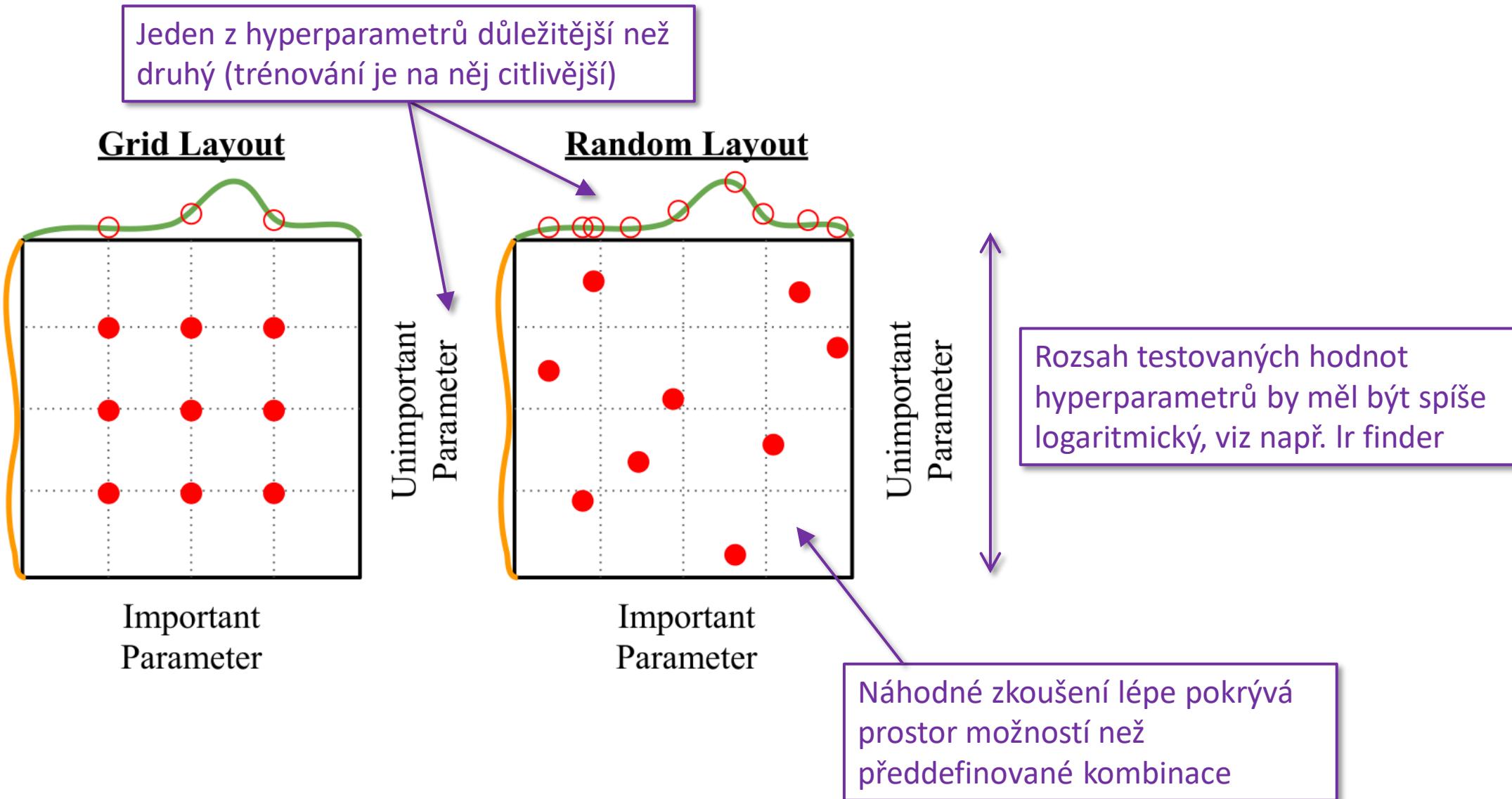
Prevence overfitu & optimalizace skóre

1. Nasbírat více dat
2. Uměle rozšířit data
3. Aplikovat vhodnější architekturu sítě
 - Nebo předtrénovanou síť (transfer learning)
4. Pokud overfit
 - zvýšit regularizaci (weight decay)
 - dropout
 - early stopping



obrázek: <https://commons.wikimedia.org/wiki/File:2d-epochs-overfitting.svg>

Až když vše funguje: optimalizace hyperparametrů



obrázek: <http://cs231n.stanford.edu/>

Další detaily a tipy ve zdrojích

- <https://cs231n.github.io/>
 - <https://cs231n.github.io/neural-networks-1/>
 - <https://cs231n.github.io/neural-networks-2/>
 - <https://cs231n.github.io/neural-networks-3/>
- <http://karpathy.github.io/2019/04/25/recipe/>