

Aplikace neuronových sítí

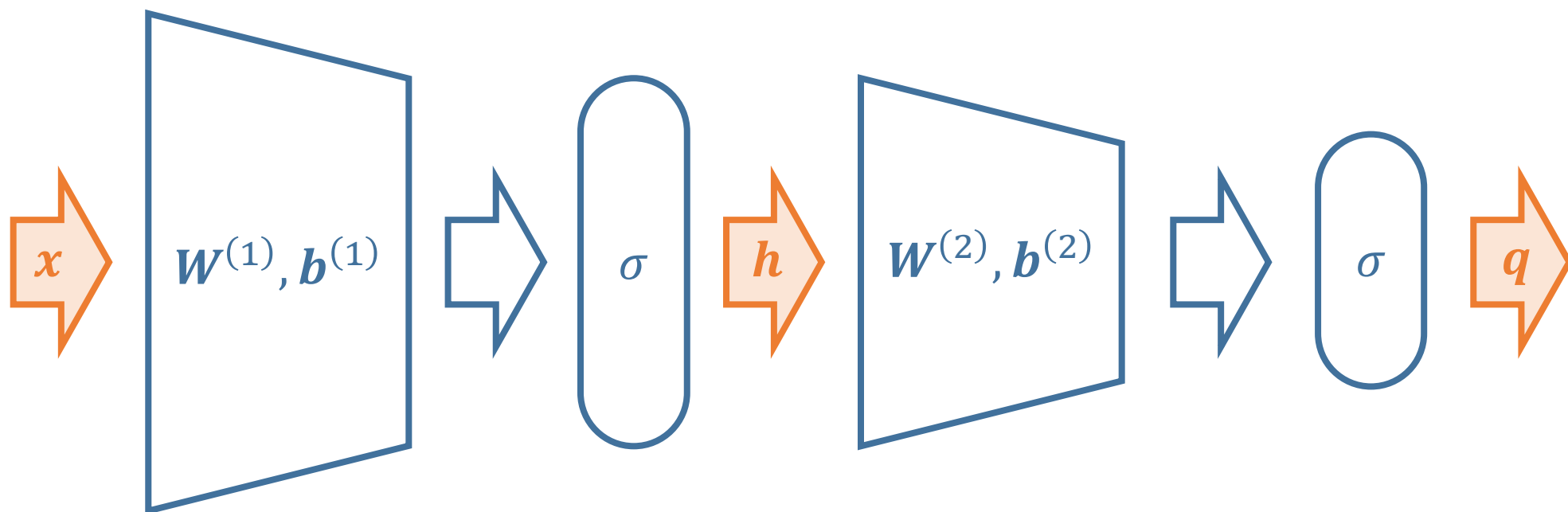
Trénování sítí

Vícevrstvý perceptron

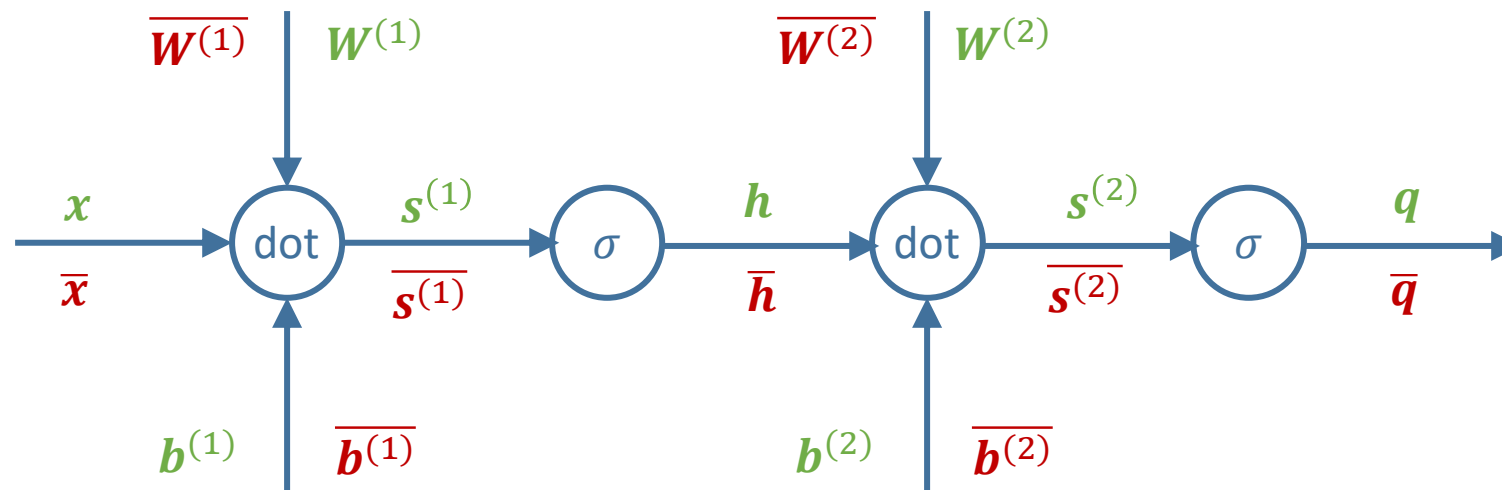
- Bloky s definovaným forward a backward chováním lze libovolně skládat za sebe
- Např. dvouvrstvý perceptron

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

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/>

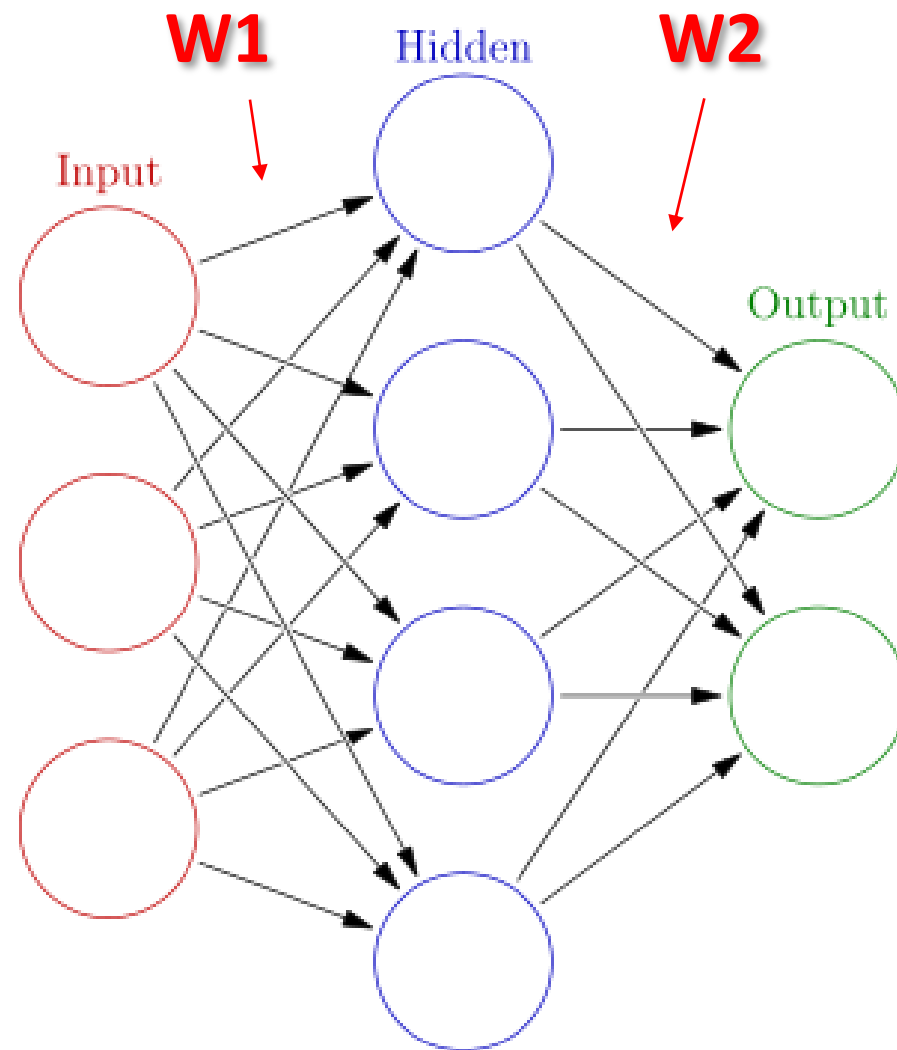
Inicializace

Náhodná inicializace

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

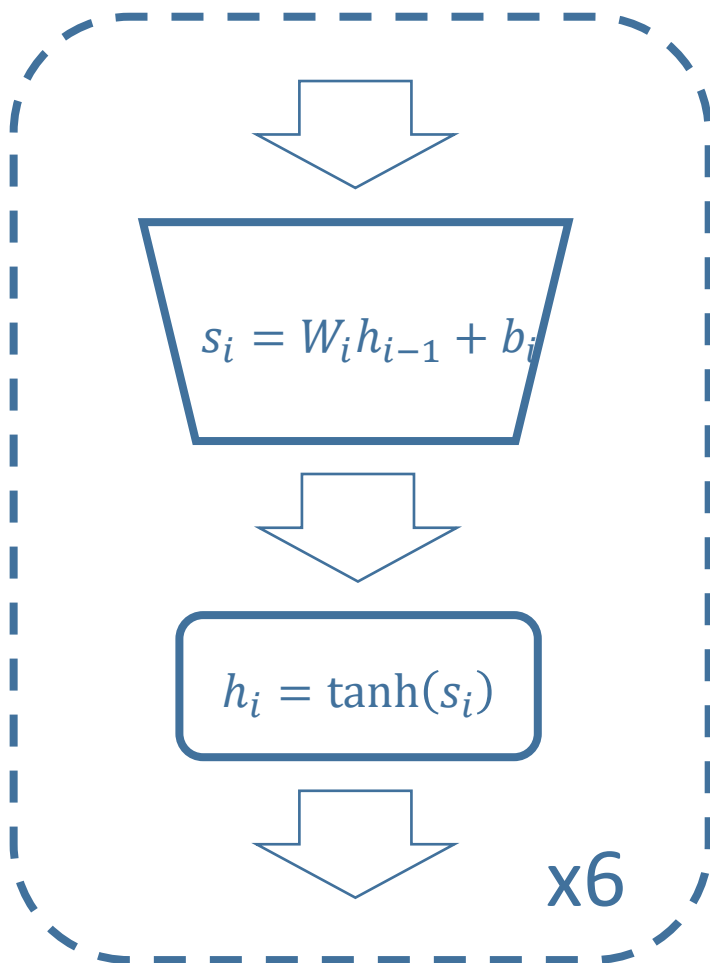
```
W2 = 0.01 * np.random.randn(4, 2)
```

- 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

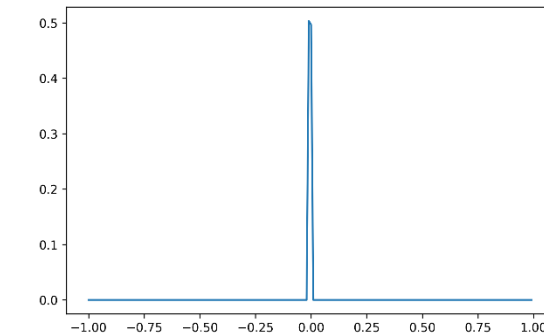
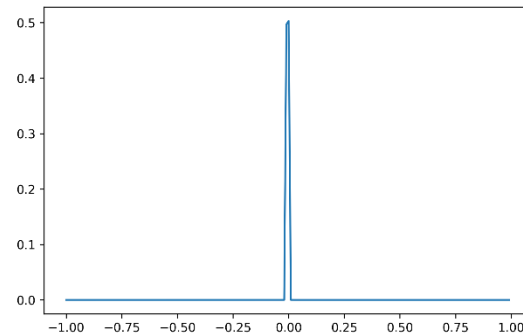
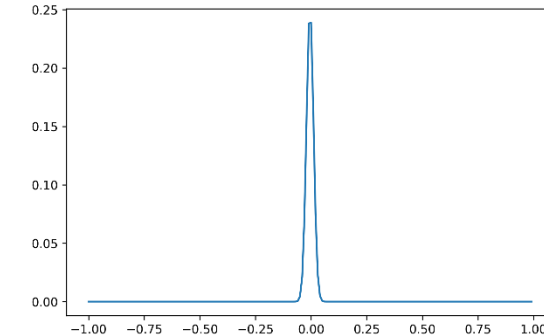
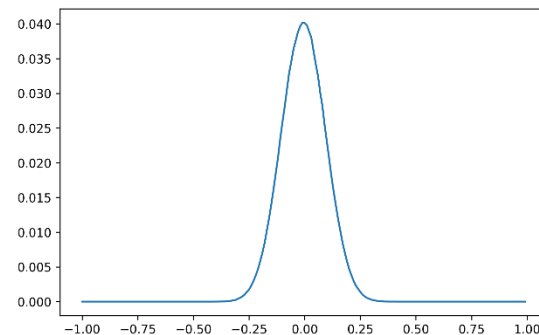
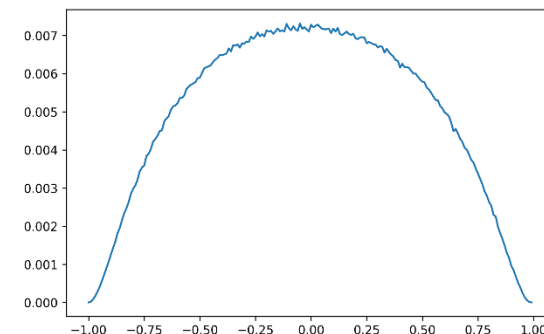
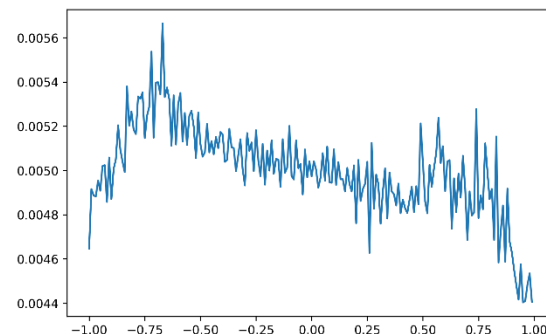


Příliš malé váhy

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

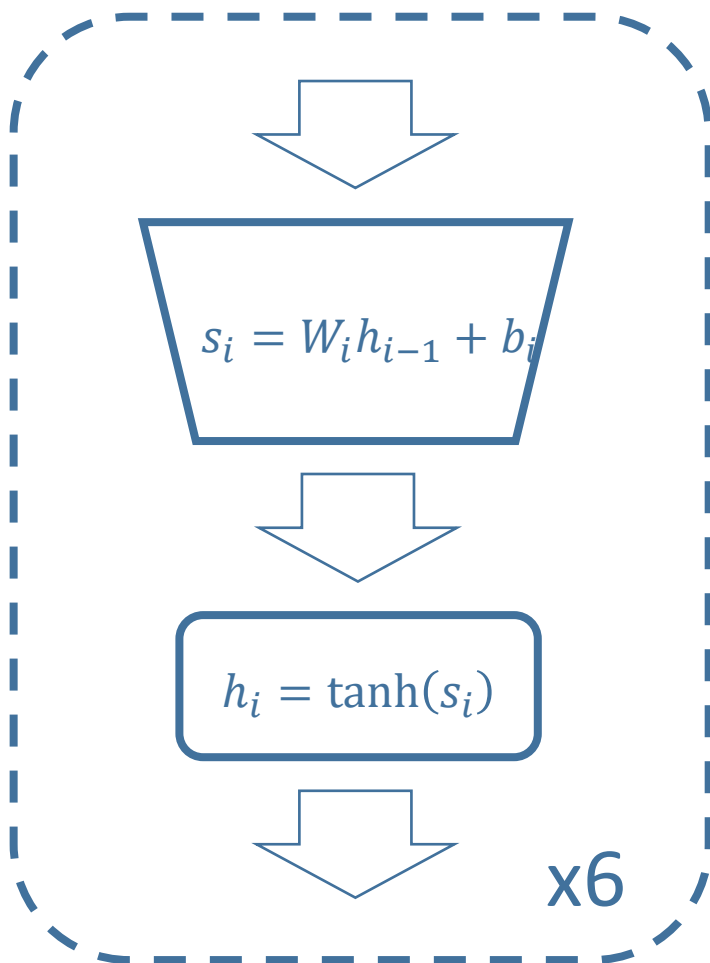


```
Wi = 0.01 * np.random.randn(fan_in, fan_out)
```

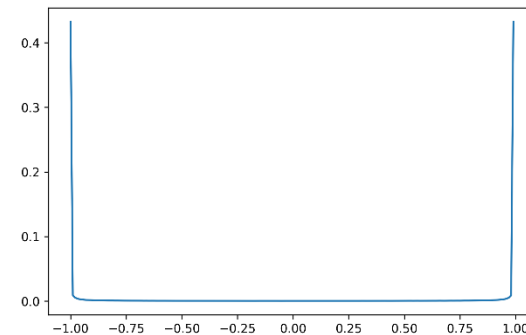
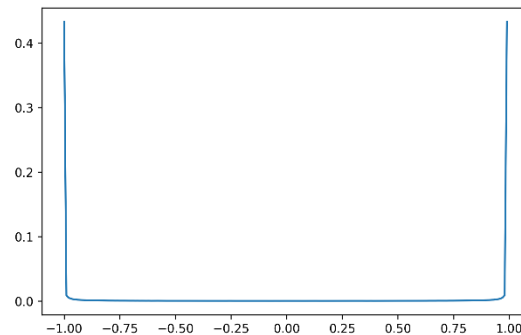
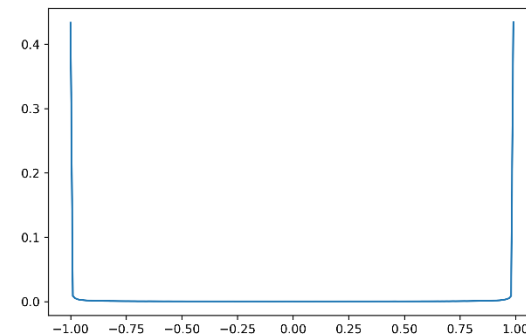
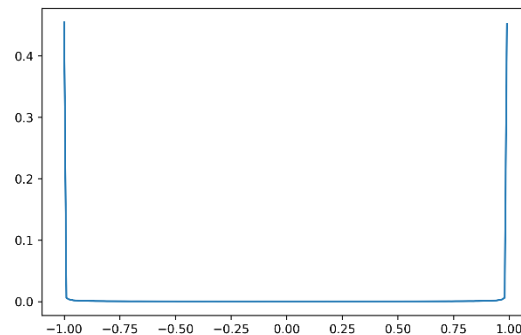
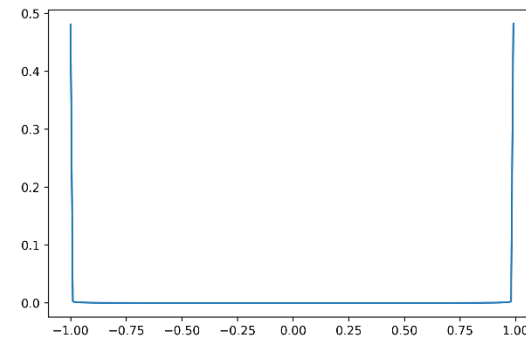
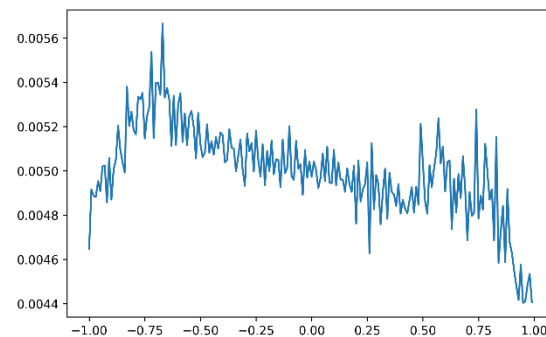


Příliš velké váhy

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



```
Wi = 1.00 * np.random.randn(fan_in, fan_out)
```

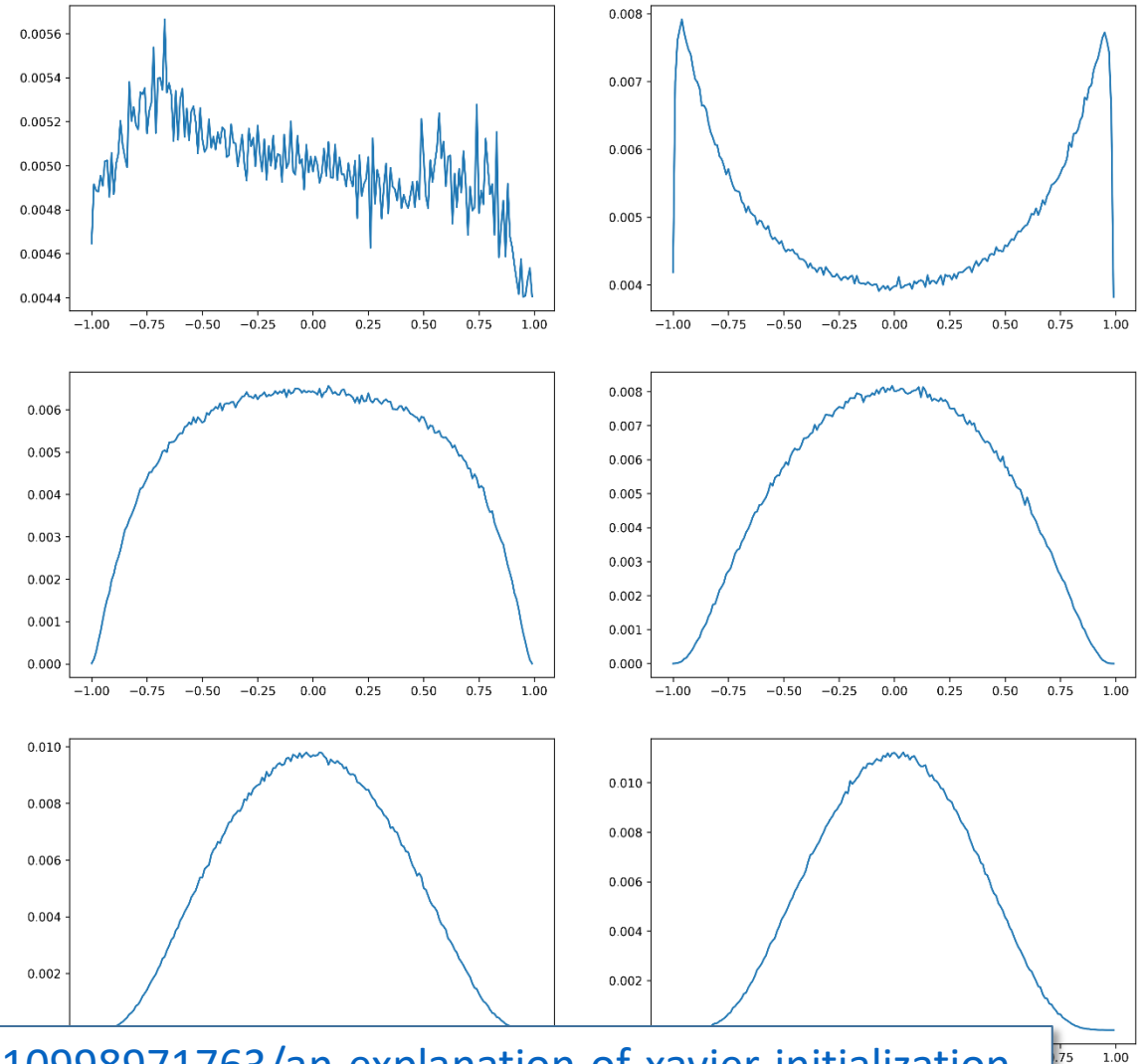
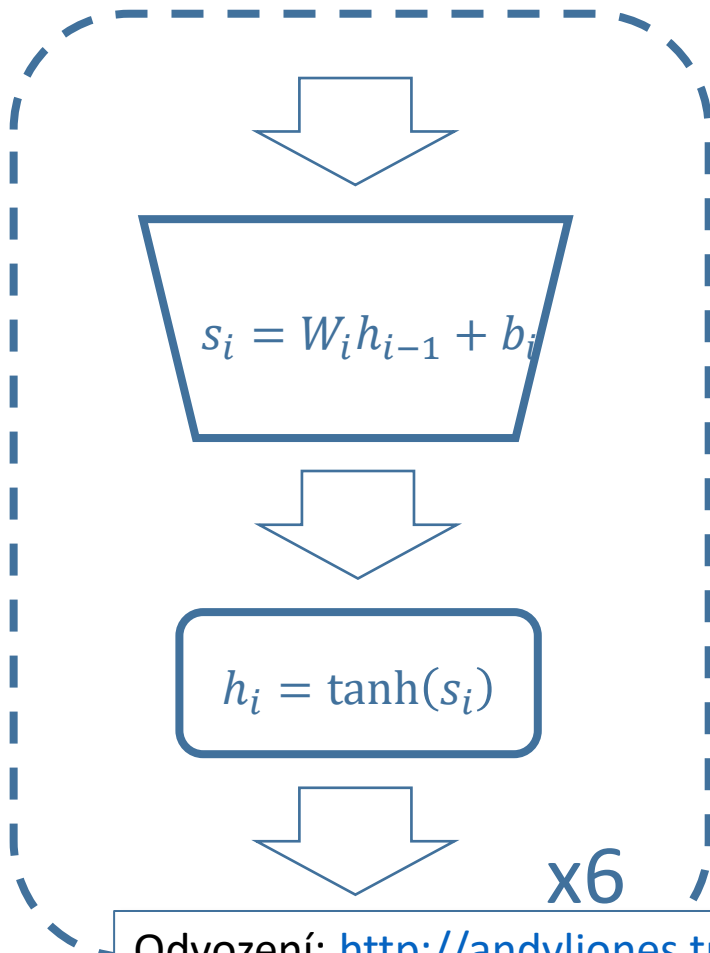


Xavier / Glorot inicializace



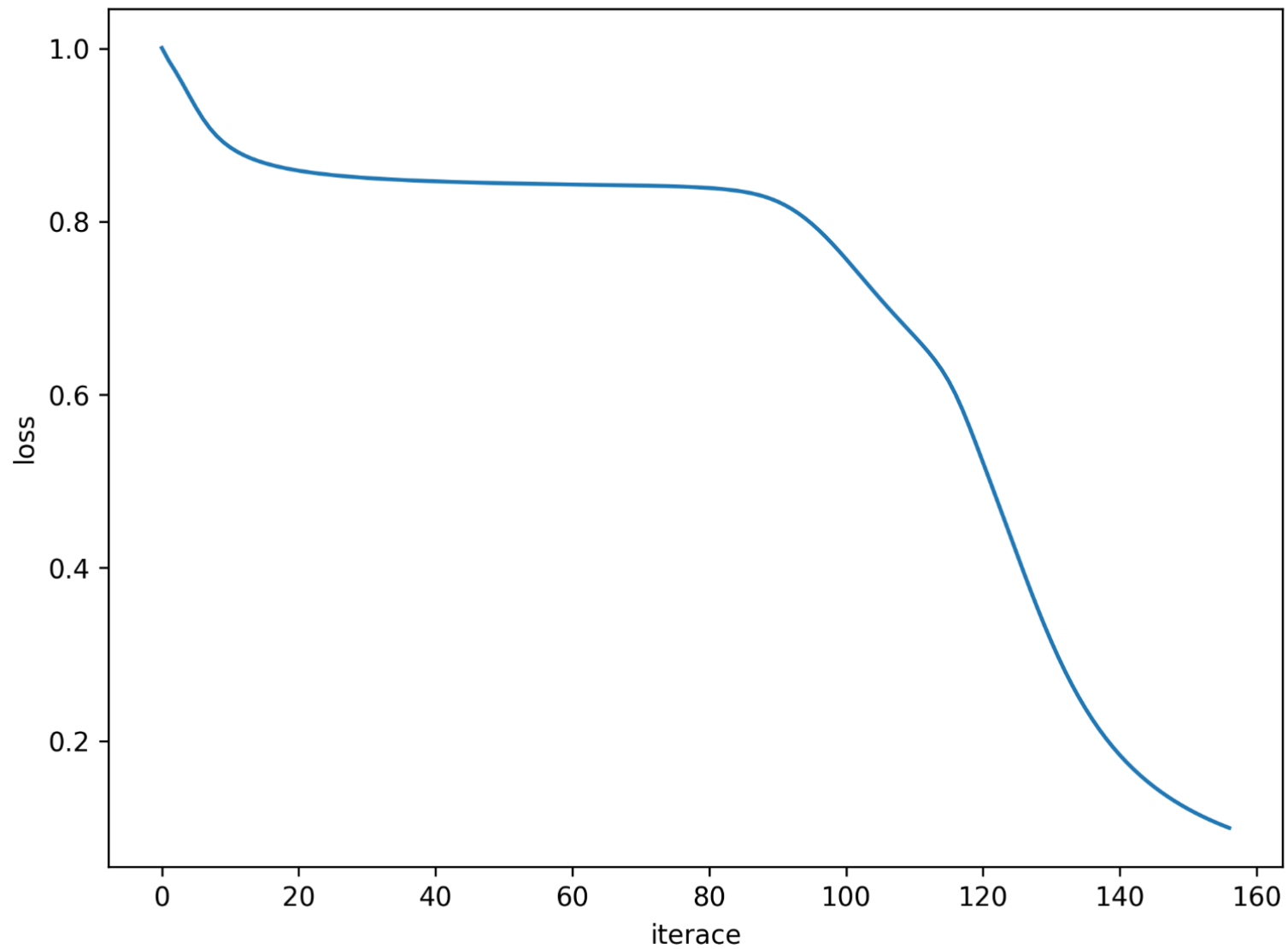
- 6 vrstev feed forward síť dle obr.
- histogram hodnot skrytých vrstev

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



Odvození: <http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization>

Špatná inicializace

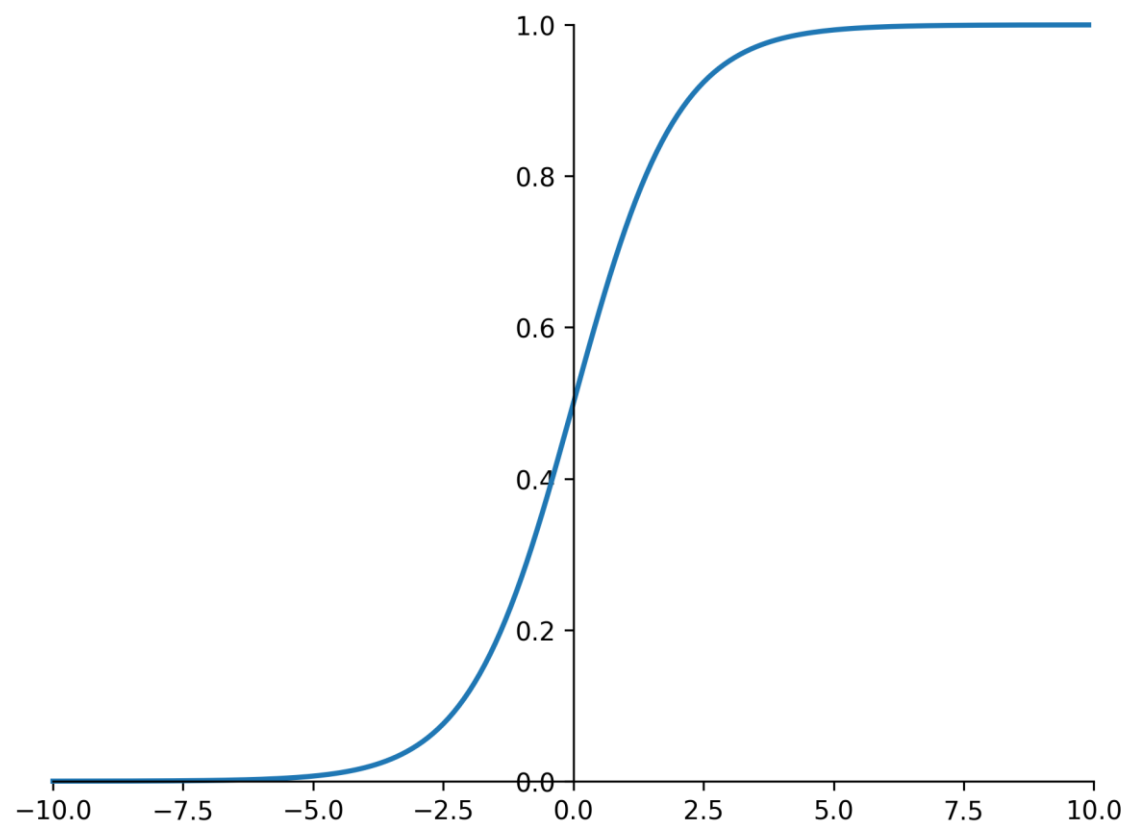


Aktivace

Sigmoid

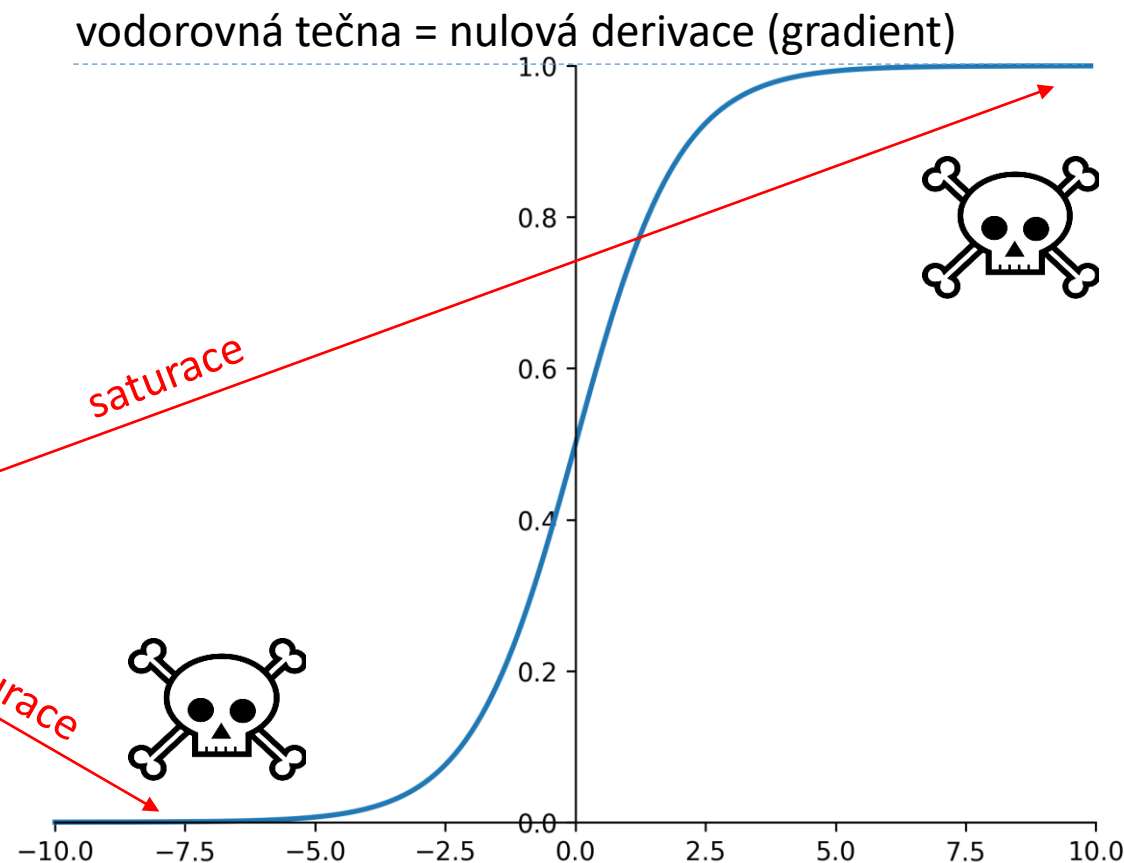
- Před Alexnet prakticky jediná používaná aktivace
- Převádí vstup na pravděpodobnost, tj. do intervalu $\langle 0, 1 \rangle$
- Vstup x jsou typicky skóre s z předchozí lineární vrstvy
- Problémy:
 1. "mizející" gradient
 2. pouze kladné hodnoty
 3. exp funkce zbytečně náročná na výpočet

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



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



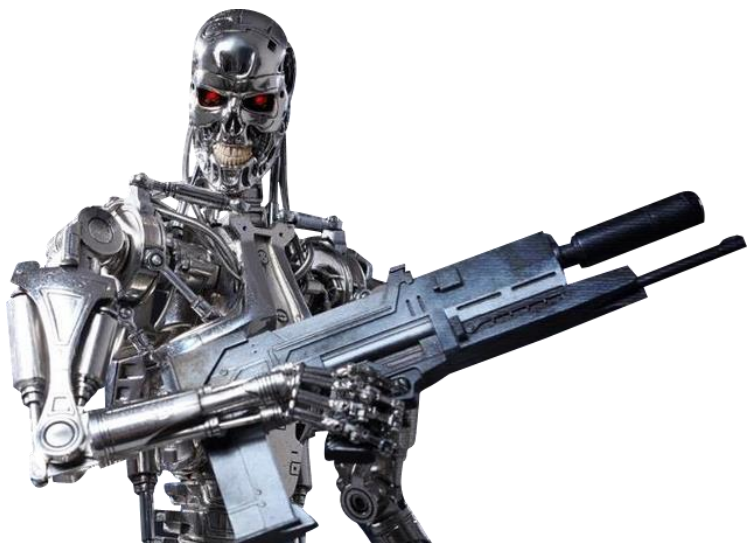
vanishing gradient

Hyperbolický tangens

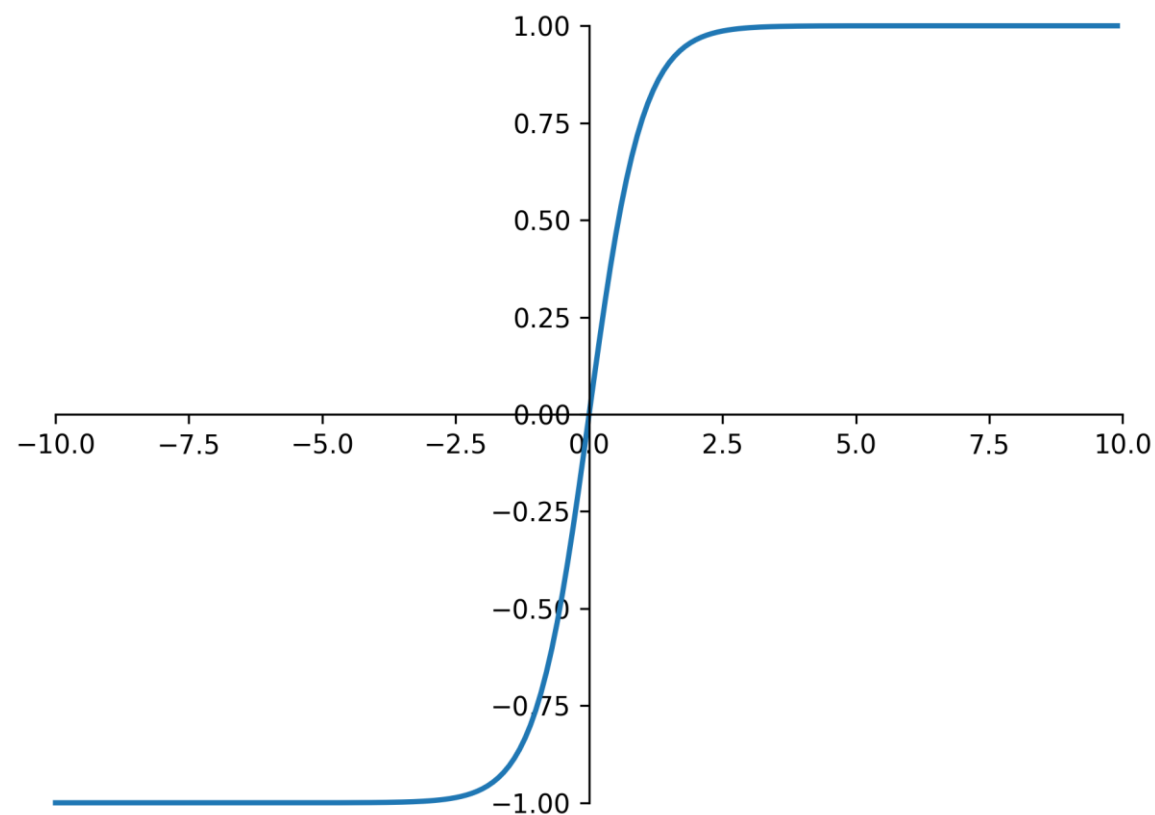
- Vlastně jen přeškálovaný sigmoid

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

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



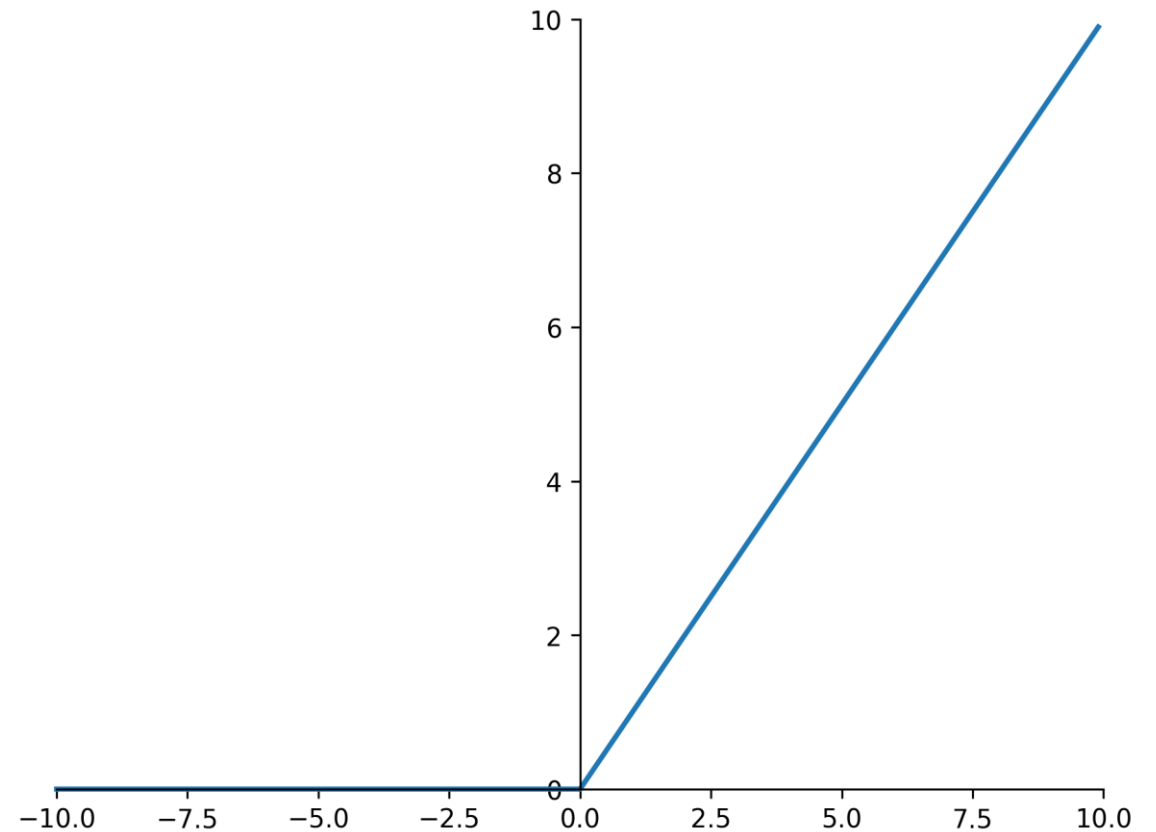
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



ReLU

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

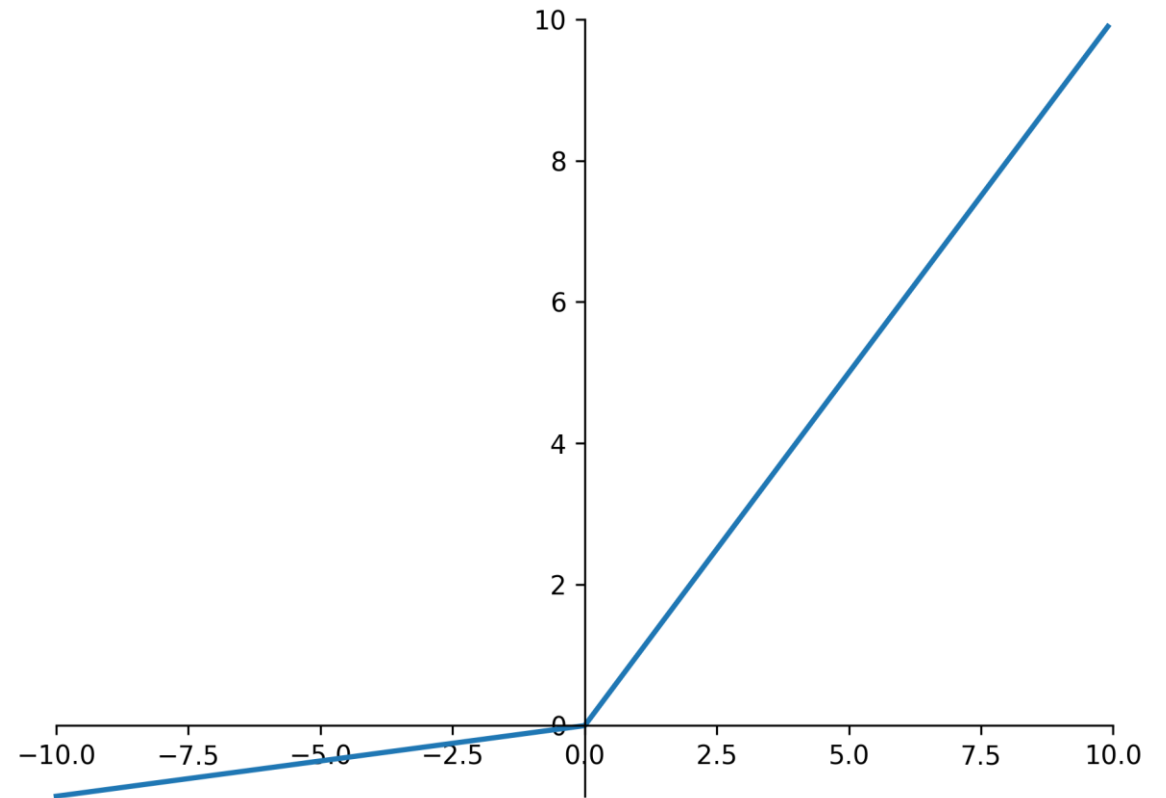
- Rectified Linear Unit
- Saturuje pouze v záporu
- Výpočetně nenáročné (bez expů)
- Trénování je mnohem rychlejší než se sigmoid či tanh
- Defaultní volba pro vnitřní nonlinearity



Leaky/Parametric ReLU

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

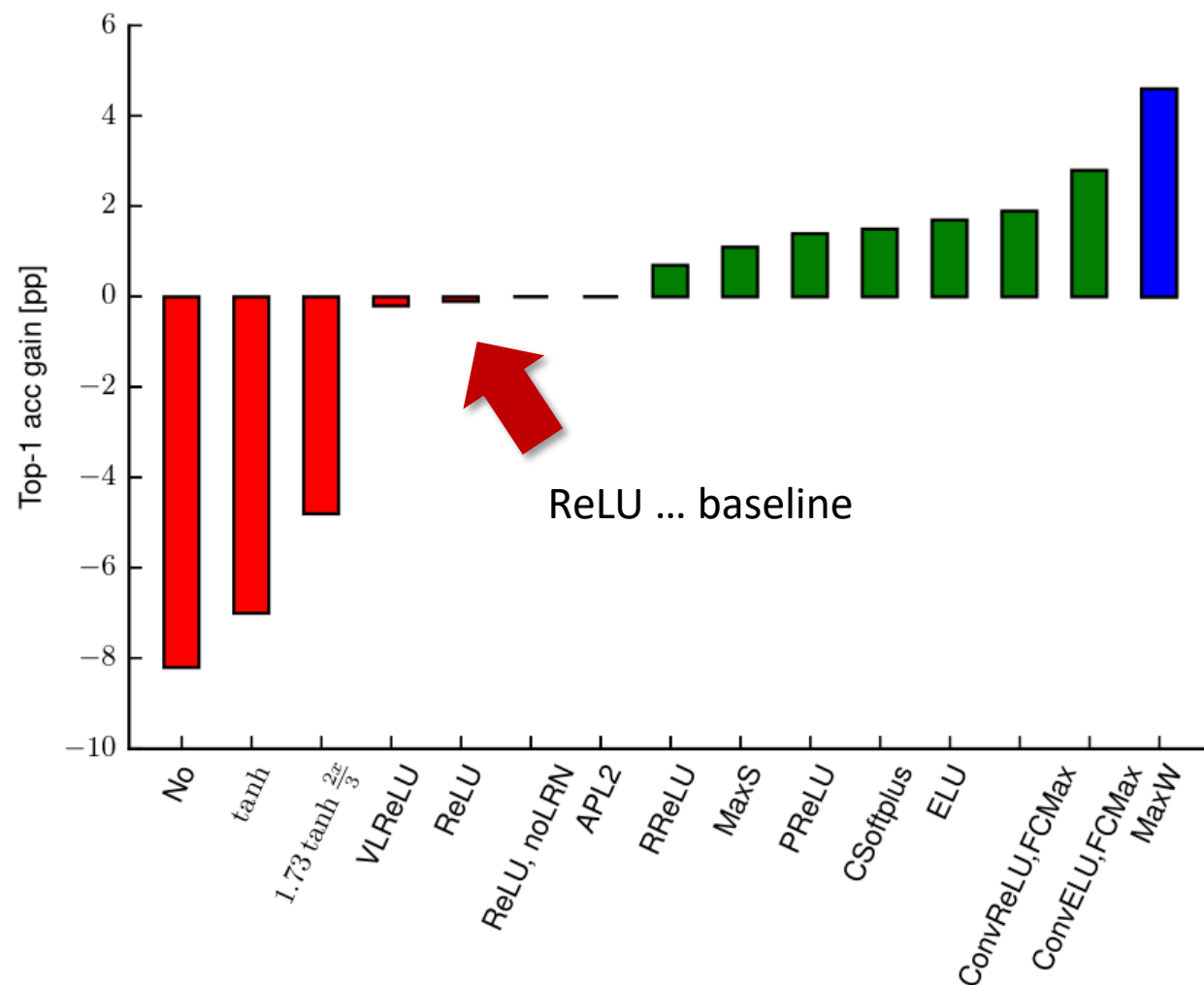
- Snaží se řešit problémy ReLU
 - nemá saturaci → neumírá gradient
 - zachovává rychlost
- Parametr α buď jako
 - konst. (např. 0.01) → Leaky ReLU
 - učitelný parameter → Parametric ReLU



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



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)`

Regularizace

Vliv přeškálování parametrů

$$\mathbf{s}_n = \mathbf{W}\mathbf{x}_n$$
$$\mathbf{q}_n = \frac{e^{\mathbf{s}_n}}{\sum_{c=1}^C e^{\mathbf{s}_{nc}}}$$

Matice parametrů
vynásobená 10x



$$\begin{bmatrix} 0.21 \\ 0.09 \\ -0.40 \end{bmatrix} = \begin{bmatrix} 0.016 & -0.002 & 0.003 \\ 0.003 & -0.006 & 0.006 \\ -0.004 & -0.006 & -0.008 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$
$$\begin{bmatrix} 0.41 \\ 0.36 \\ 0.22 \end{bmatrix} = \text{Softmax} \left(\begin{bmatrix} 0.21 \\ 0.09 \\ -0.40 \end{bmatrix} \right)$$

$$\begin{bmatrix} 2.05 \\ 0.90 \\ -4.00 \end{bmatrix} = \begin{bmatrix} 0.16 & -0.02 & 0.03 \\ 0.03 & -0.06 & 0.06 \\ -0.04 & -0.06 & -0.08 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$
$$\begin{bmatrix} 0.76 \\ 0.24 \\ 0.00 \end{bmatrix} = \text{Softmax} \left(\begin{bmatrix} 2.05 \\ 0.90 \\ -4.00 \end{bmatrix} \right)$$

- Přeškálováním parametrů se zvýrazní rozdíly, ale nezmění znaménko skóre (logitů) ani pořadí pravděpodobností na výstupu, tj. klasifikátor predikuje stále stejně
- Hodnota kritéria (lossu) se ale přitom zmenší
- Aby nedocházelo k optimalizaci lossu pouhým škálováním velikosti parametrů, měli bychom nějak toto chování penalizovat

Regularizace

- Do kritéria (lossu) zavedeme člen penalizující velikost vah pomocí L2 normy

$$R(\mathbf{W}) = \|\mathbf{W}\|_2^2 = \sum_{c=1}^C \sum_{d=1}^D w_{cd}^2$$

- Celkově pak úloha je

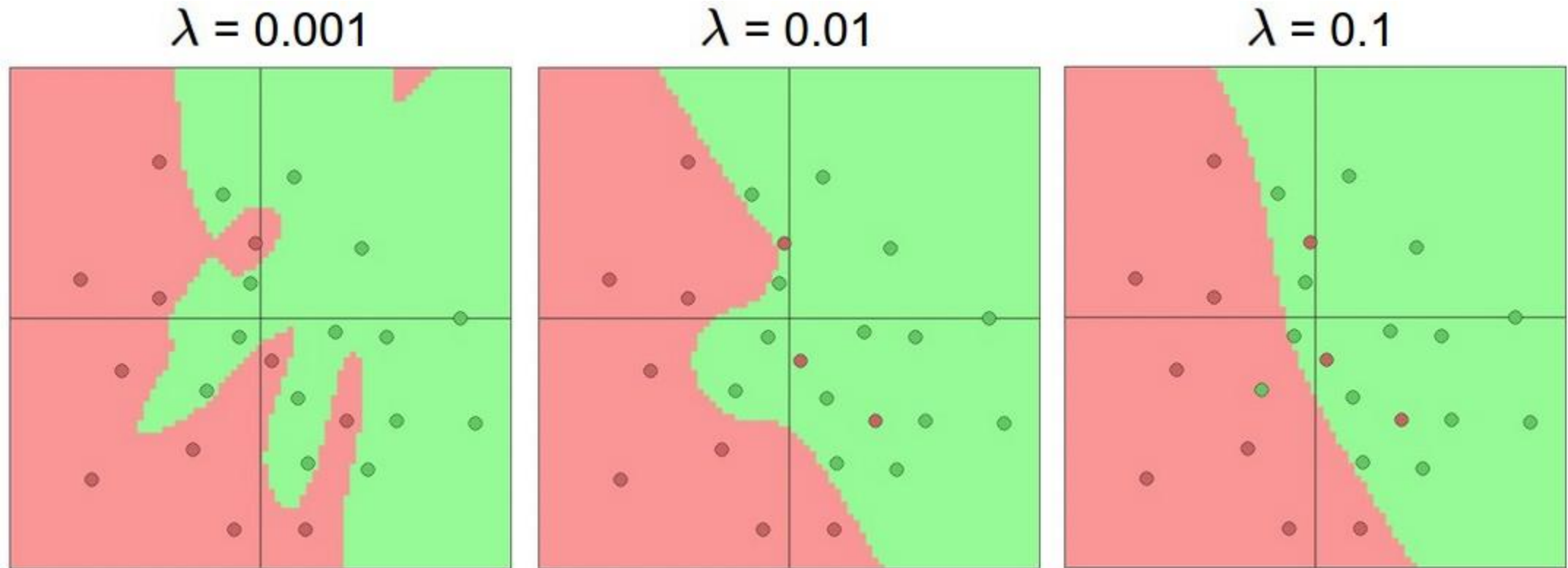
$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) + \lambda R(\mathbf{W})$$

- $\lambda R(\mathbf{W})$ je regularizační člen
 - λ je hyperparametr, často označovaný jako weight decay
 - Obvykle aplikován pouze na váhy, na bias nikoliv

Regularizace: příklad

- Např. $\mathbf{x} = [1, 1, 1, 1]^T$ a dvojce různé parametry:
 - $\mathbf{w}_1 = [1, 0, 0, 0]^T$
 - $\mathbf{w}_2 = [0.25, 0.25, 0.25, 0.25]^T$
- Přestože $\mathbf{w}_1^T \mathbf{x} = \mathbf{w}_2^T \mathbf{x} = 1$, preferujeme \mathbf{w}_2
- Brání přeučení
 - \mathbf{w}_2 má menší normu
 - \mathbf{w}_1 sází všechno na jeden příznak, zatímco \mathbf{w}_2 důležitost rozkládá
 - normu $\|\mathbf{w}\|_2$ lze interpretovat jako apriorní pravděpodobnost
 - regularizace brání změnám \rightarrow trénování méně reaguje na změny

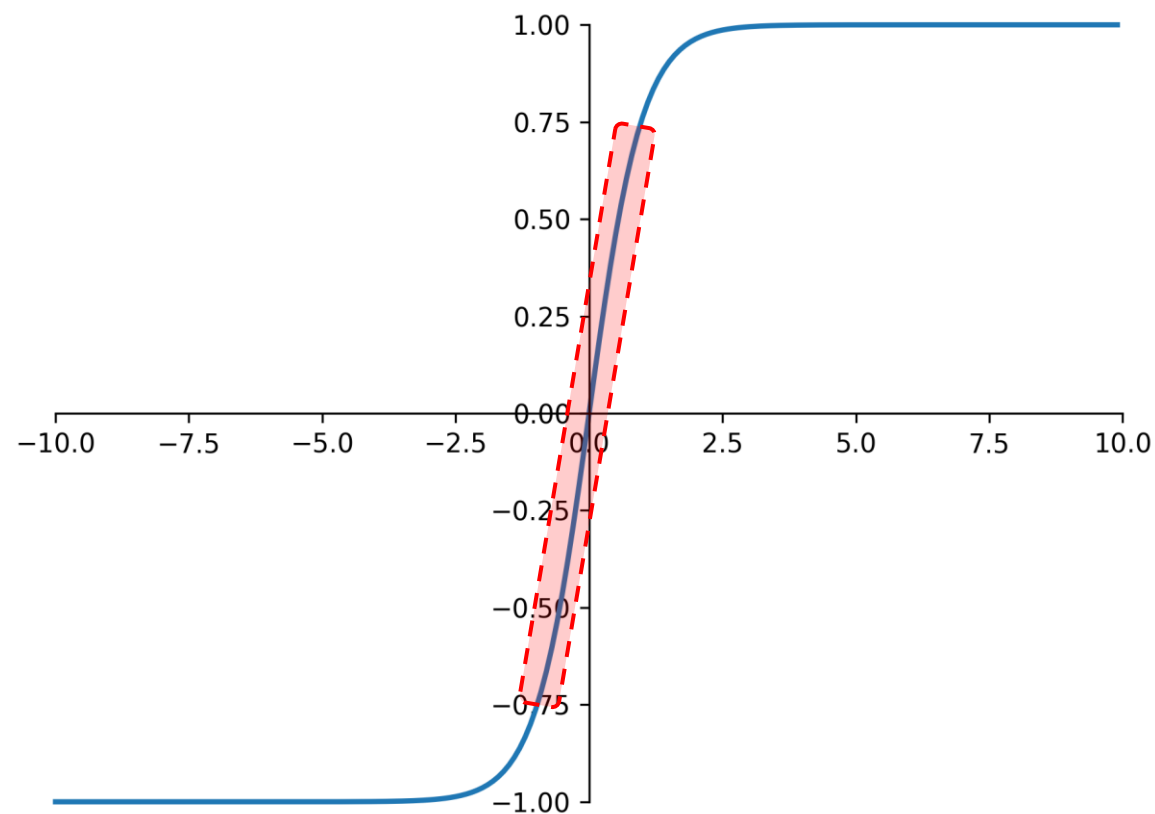
Vliv regularizace



<http://cs231n.github.io/neural-networks-1/>

Proč regularizace pomáhá proti overfittingu?

- Např. pro tanh
- Čím menší hodnoty, tím větší pravděpodobnost, že se budeme nacházet v lineární části aktivační funkce
- Méně nelineární = jednodušší funkce



Proč weight decay?

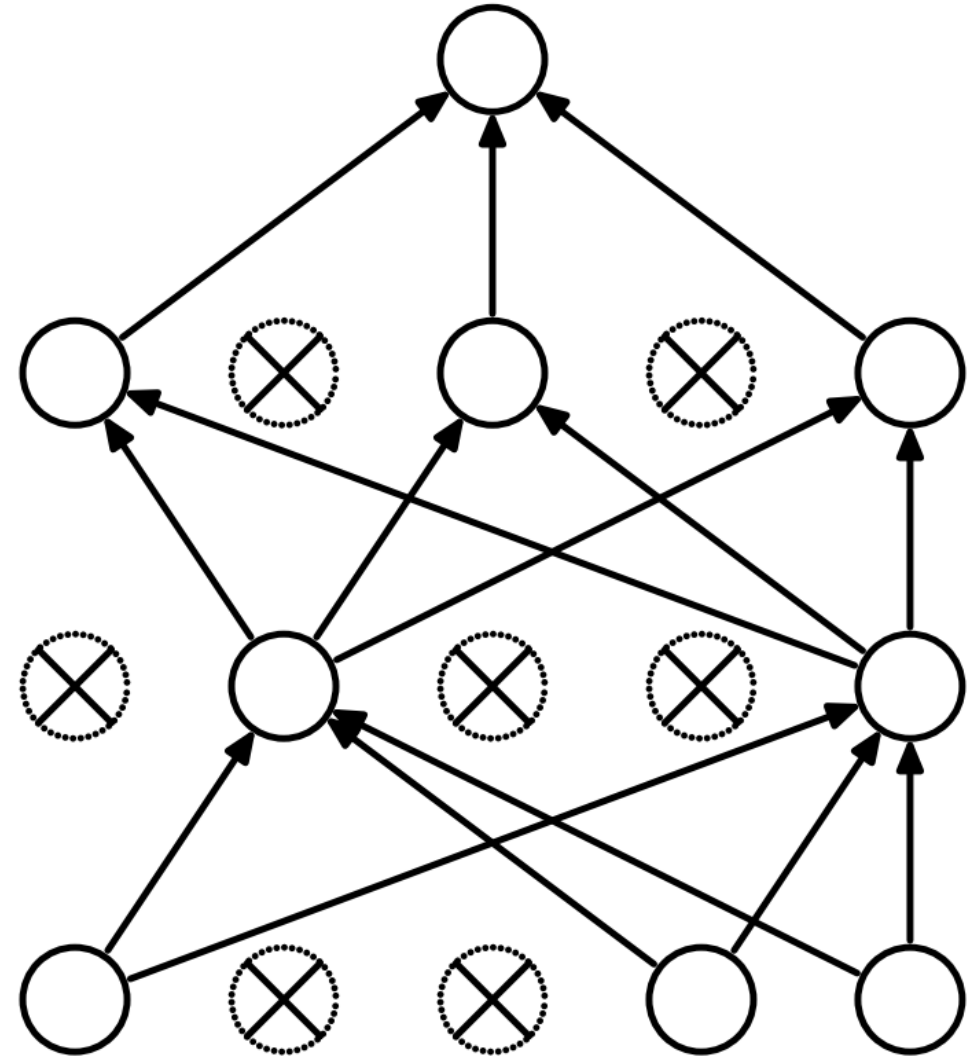
- Standardní update SGD $W := W - \gamma \text{d}W$
- Pokud L2, pak navíc $W := W - \gamma \text{d}W - \gamma 2\lambda W$
 - protože $\frac{\text{d}}{\text{d}W}(\lambda W^2) = 2\lambda W$
- To jde napsat jako $W := (1 - 2\gamma\lambda)W - \gamma \text{d}W$
- člen $(1 - 2\gamma\lambda)$ je $< 1 \rightarrow$ odečítá část W , proto decay

Dropout

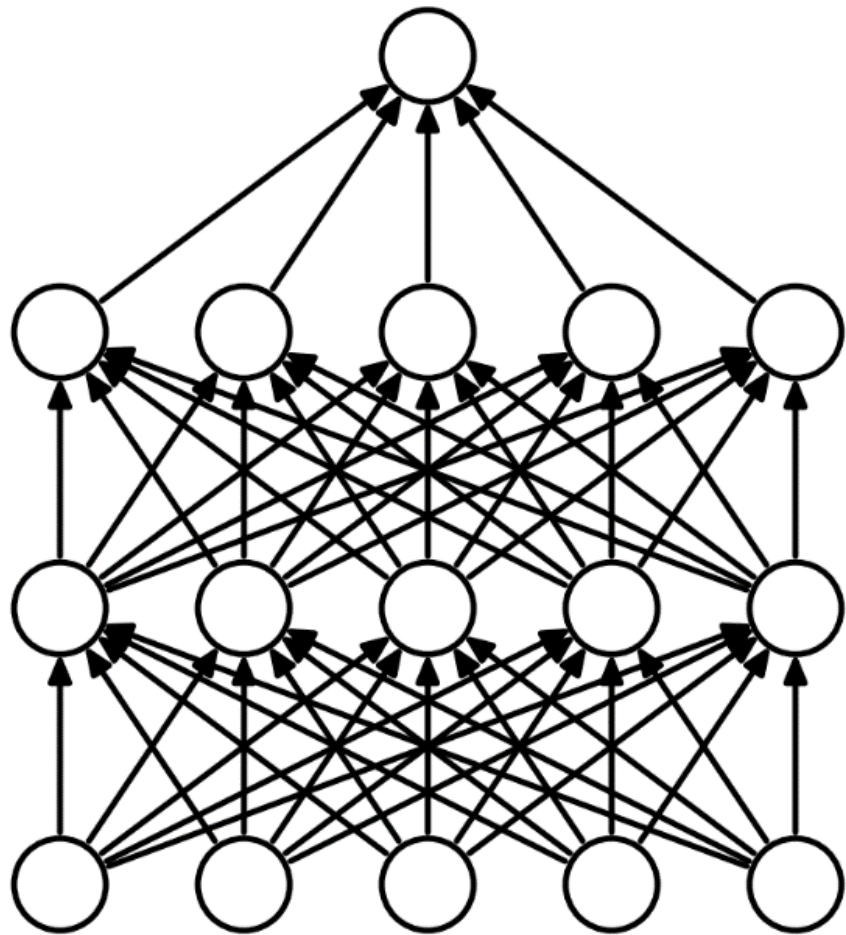
- Náhodně nastavuje výstupy na nulu
- Např. s pravděpodobností 40 %:

```
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.
```

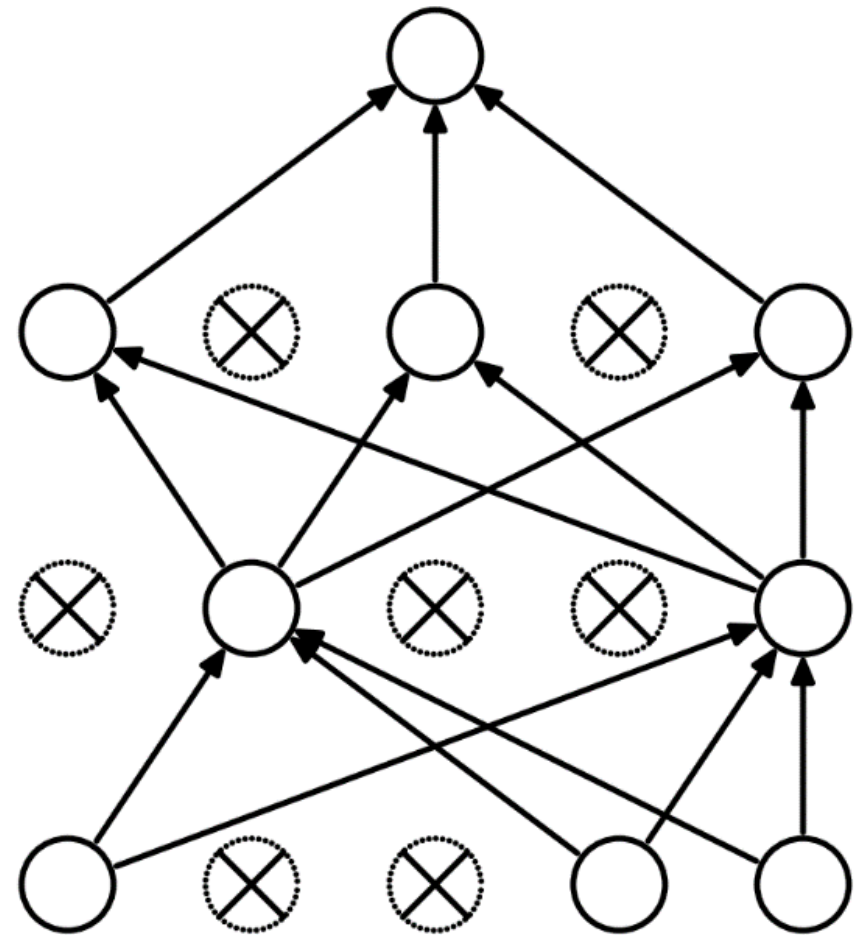
- slouží jako regularizace → prevence overfitu



Dropout



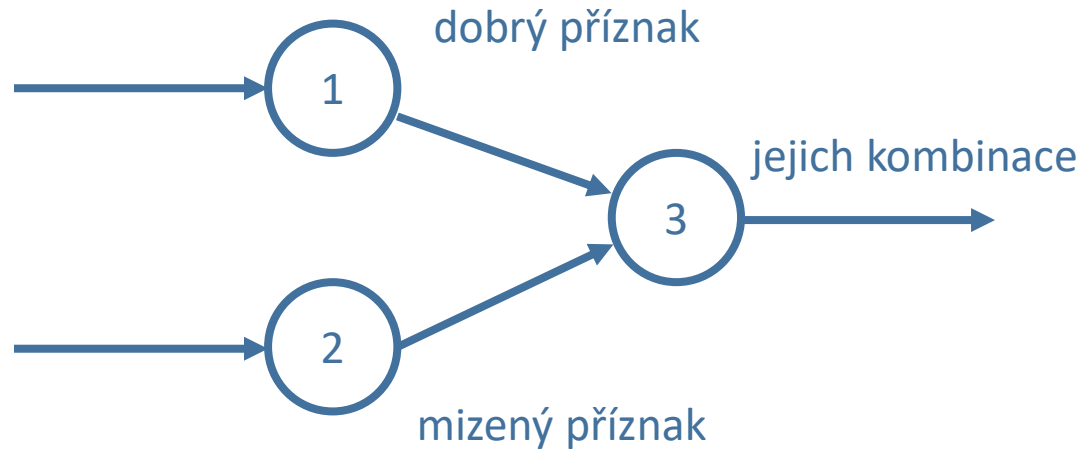
(a) Standard Neural Net



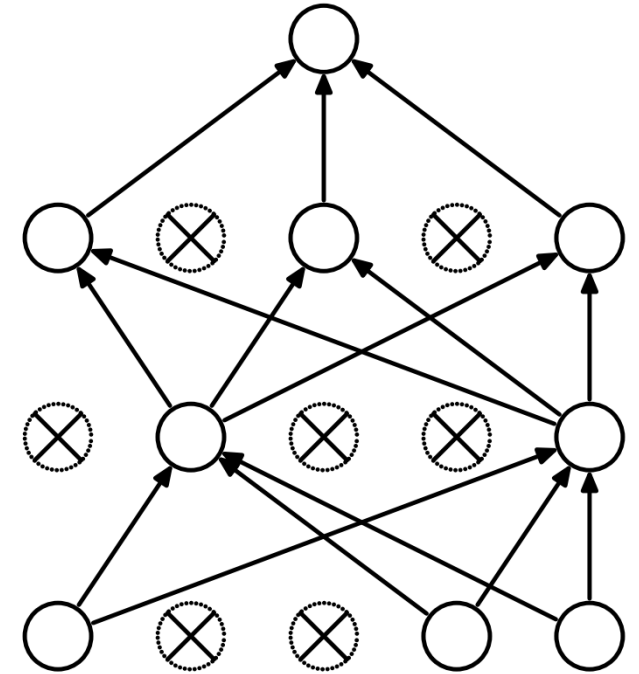
(b) After applying dropout.

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

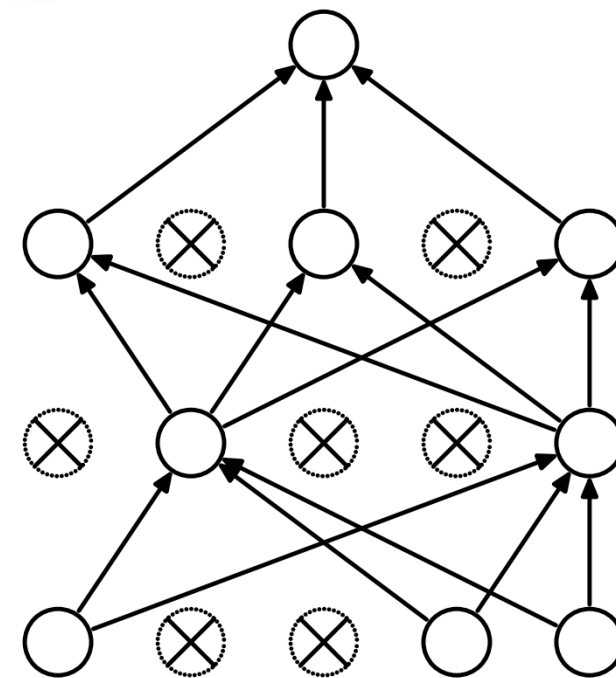
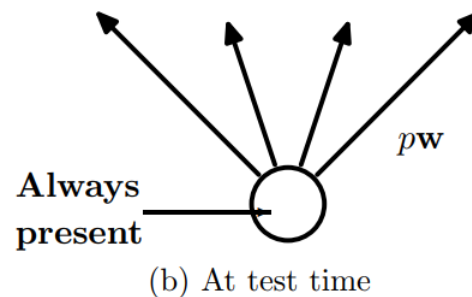
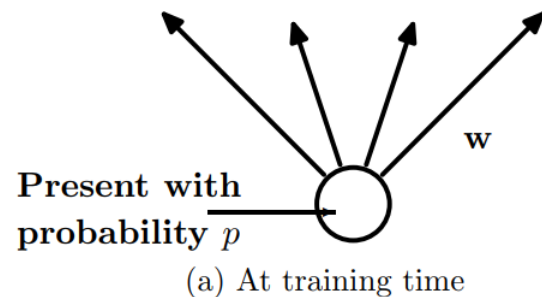
**Rozdílné chování v
trénovací a testovací fázi!**

- 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 → tomu se přizpůsobí váhy a aktivace
- bez dropoutu v testovací fázi je pak příliš velká “energie” výstupu do další vrstvy




Dropout v trénovací a testovací fázi

- Režim 1: základní (direct) dropout

- v testovací fázi vynásobíme výstup dropout pravděpodobností $1 - p$


```
1 def train(x):
2     hidden = relu(np.dot(x, w1) + b1)
3     mask = np.random.rand(*hidden.shape) < 0.4
4     hidden[mask] = 0.
5     prob = np.dot(hidden, w2) + b2
6
7     # loss
8     # update gradientu
9
10 def predict(x):
11     hidden = relu(np.dot(x, w1) + b1)
12     hidden *= (1 - 0.4)
13     prob = np.dot(hidden, w2) + b2
14
15     # argmax / klasifikace
```



- Režim 2: inverted dropout

- vyřešíme už v trénovací fázi, kde výstup vydělíme $1 - p$, aby měl stejnou “energii”, jako kdyby žádný dropout nebyl

```
1 def train(x):
2     hidden = relu(np.dot(x, w1) + b1)
3     mask = np.random.rand(*hidden.shape) < 0.4
4     hidden[mask] = 0.
5     prob = np.dot(hidden / (1 - 0.4), w2) + b2
6
7     # loss
8     # update gradientu
9
10 def predict(x):
11     hidden = relu(np.dot(x, w1) + b1)
12     prob = np.dot(hidden, w2) + b2
13
14     # argmax / klasifikace
```



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/>

Optimalizační metody

Metoda největšího spádu (Gradient Descent)

Inicializace:

- parametry θ_0 na náhodné hodnoty

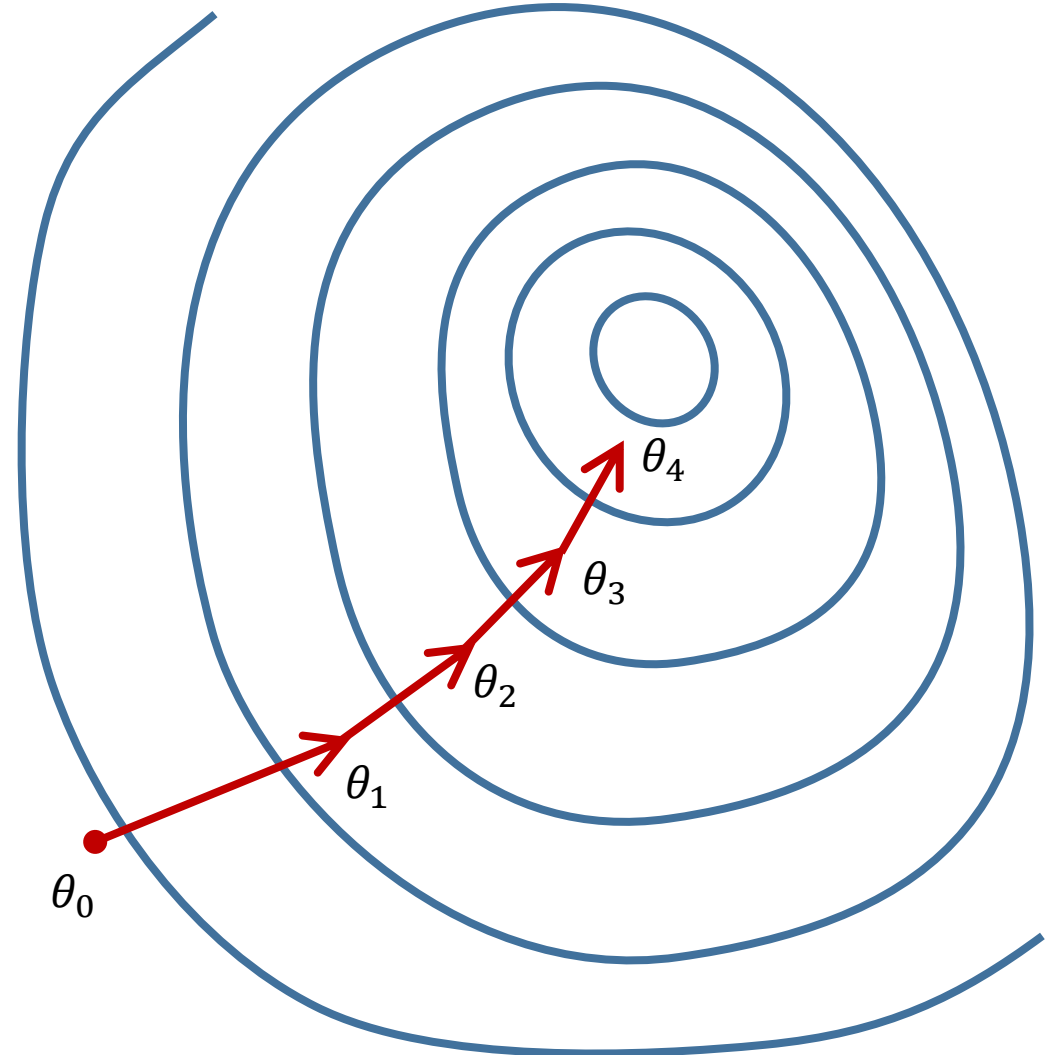
Opakujeme:

learning rate

$$\theta_{t+1} = \theta_t - \gamma \cdot \nabla L(\theta_t)$$

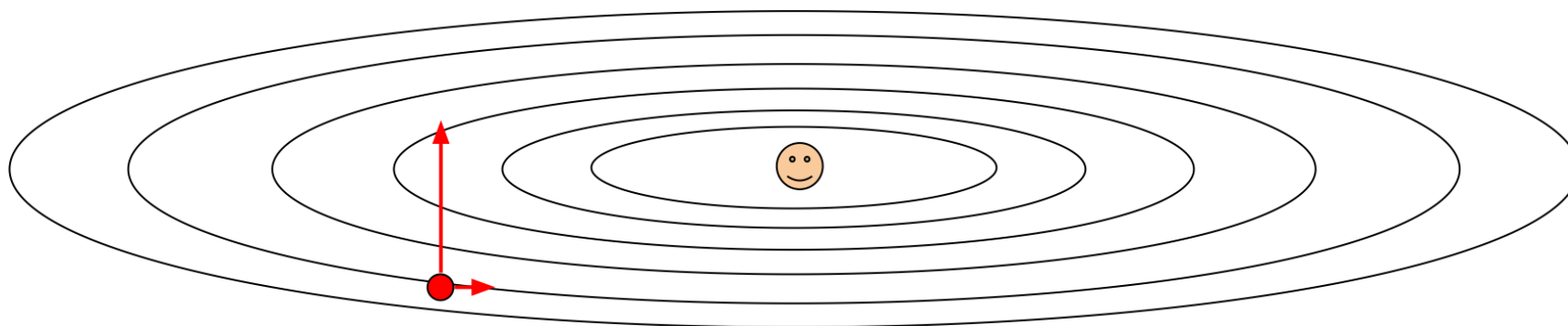
Skončíme:

- po vykonaném počtu kroků
- loss již neklesá, $L(\theta_t) \geq L(\theta_{t-k}) \forall k = 1, \dots, K$



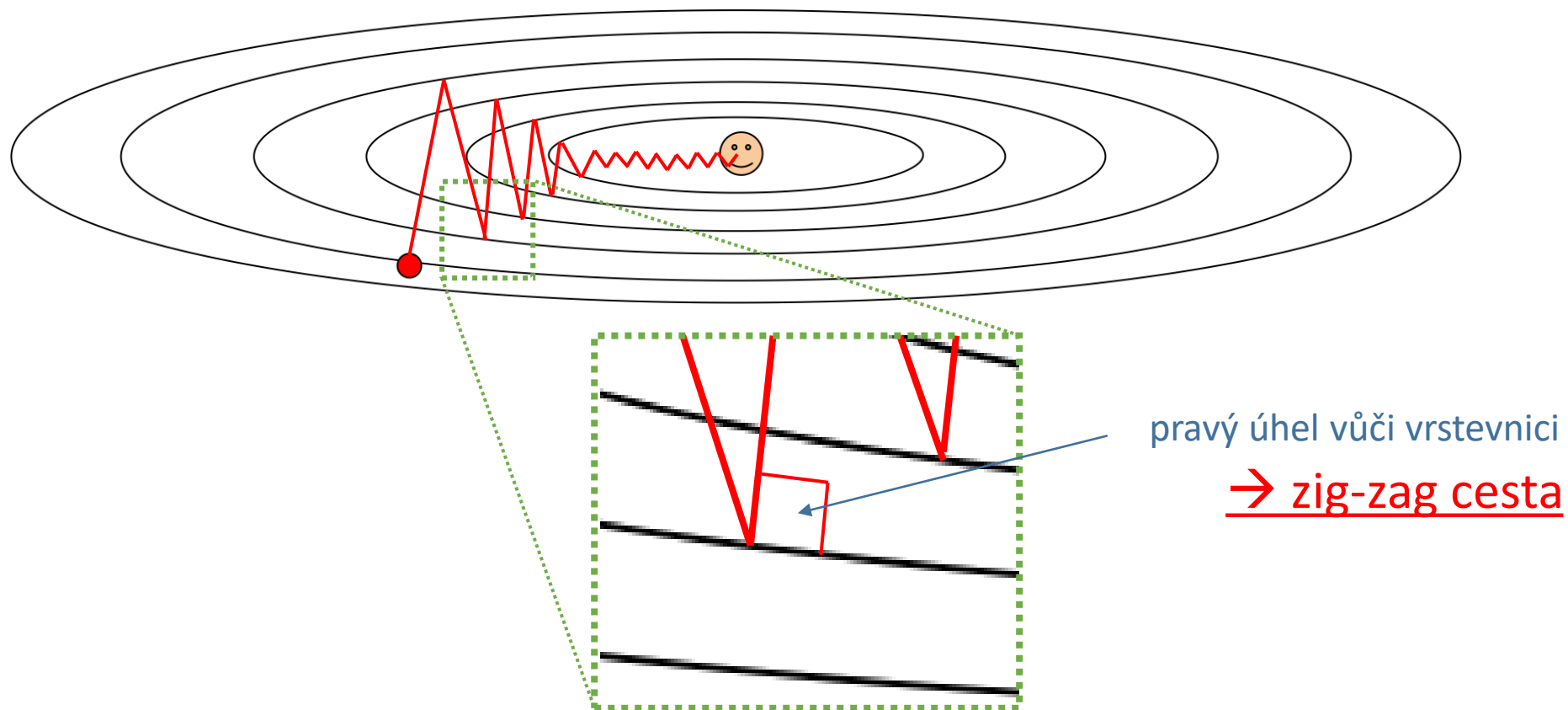
SGD update

funkce, která je v jednom směru mnohem citlivější na změnu



SGD update

funkce, která je v jednom směru mnohem citlivější na změnu



Momentum SGD

- Pamatuje si předchozí update
- Přičítá k novému
- Momentum = hybnost, aneb simuluje “koulení” z kopce
- Obvykle konverguje rychleji

hyperparametr, např. $\alpha = 0.95$

learning rate (krok)

$$(1) \quad v_{t+1} := \alpha \cdot v_t - \gamma \cdot \nabla L(\theta_t)$$

$$(2) \quad \theta_{t+1} := \theta_t + v_{t+1}$$

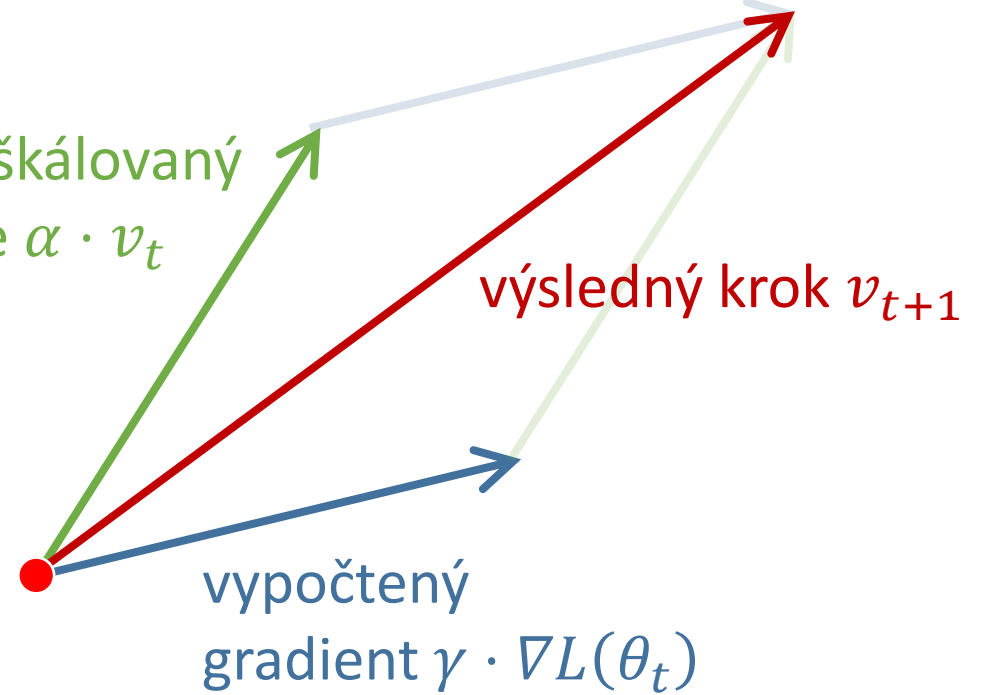
hybnost = přeškálovaný
minulý update $\alpha \cdot v_t$

výsledný krok v_{t+1}

vypočtený
gradient $\gamma \cdot \nabla L(\theta_t)$

standardní SGD:

$$\theta_{t+1} := \theta_t - \gamma \cdot \nabla L(\theta_t)$$

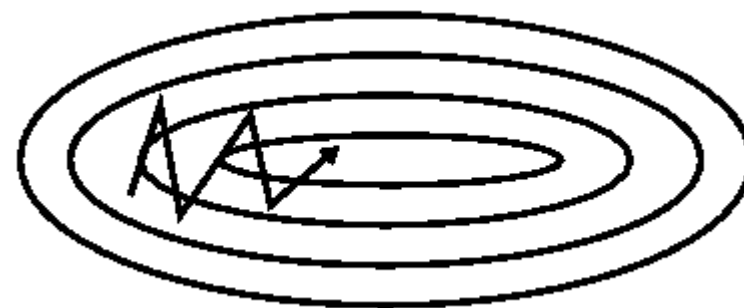


Momentum SGD

Obyčejné SGD



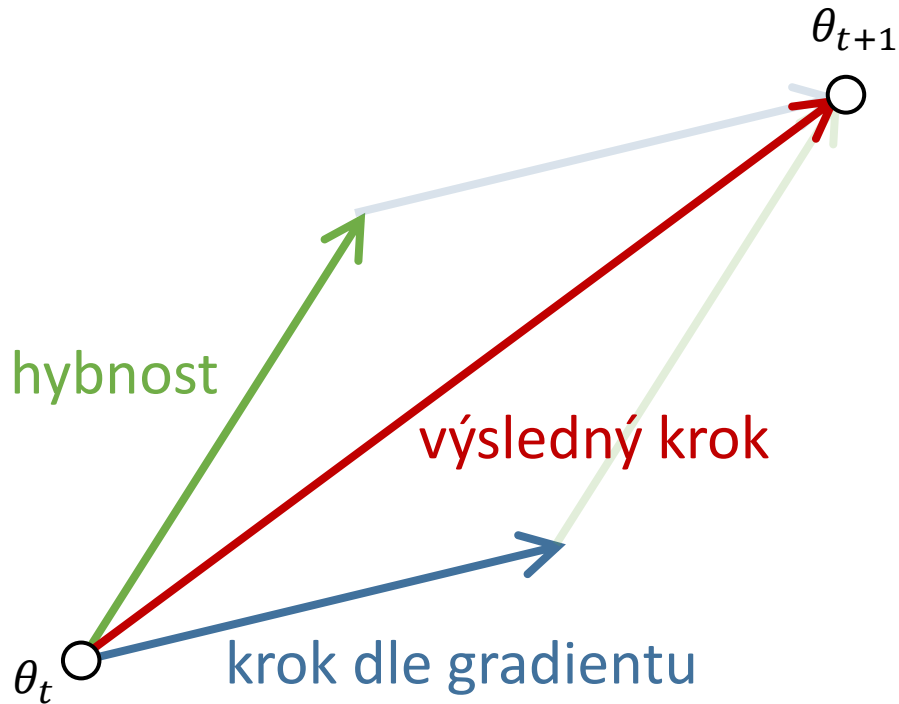
SGD + momentum



v horizontálním směru postupně nabírá rychlost

Nesterov Accelerated Gradient (NAG)

SGD + momentum



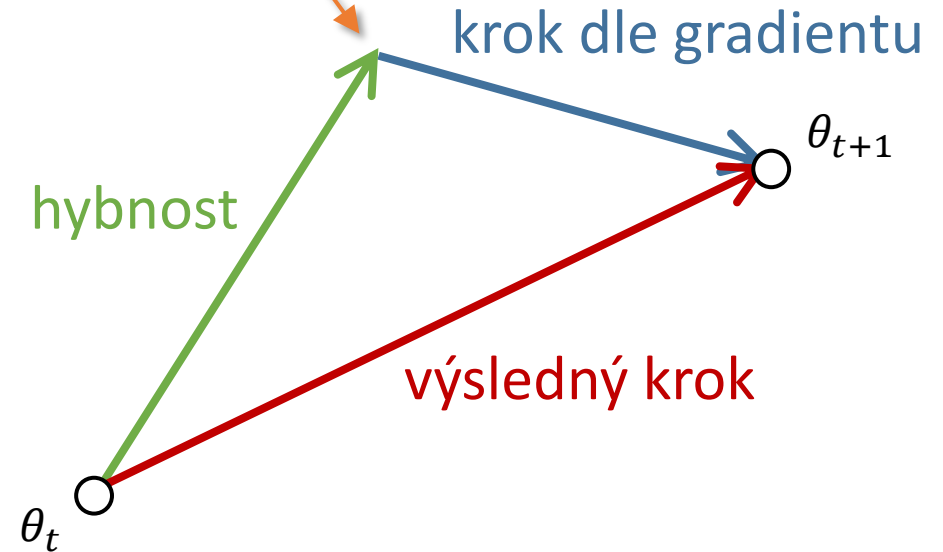
$$v_{t+1} := \alpha \cdot v_t - \gamma \cdot \nabla L(\theta_t)$$

$$\theta_{t+1} := \theta_t + v_{t+1}$$

Nesterov

nejprve update hybností,
poté teprve výpočet gradientu

gradient se vyhodnocuje
v bodě $\theta_t + \alpha v_t$



$$v_{t+1} := \alpha \cdot v_t - \gamma \cdot \nabla L(\theta_t + \alpha v_t)$$

$$\theta_{t+1} := \theta_t + v_{t+1}$$

Nesterov Accelerated Gradient (NAG)

- Nesterov mechanismus:

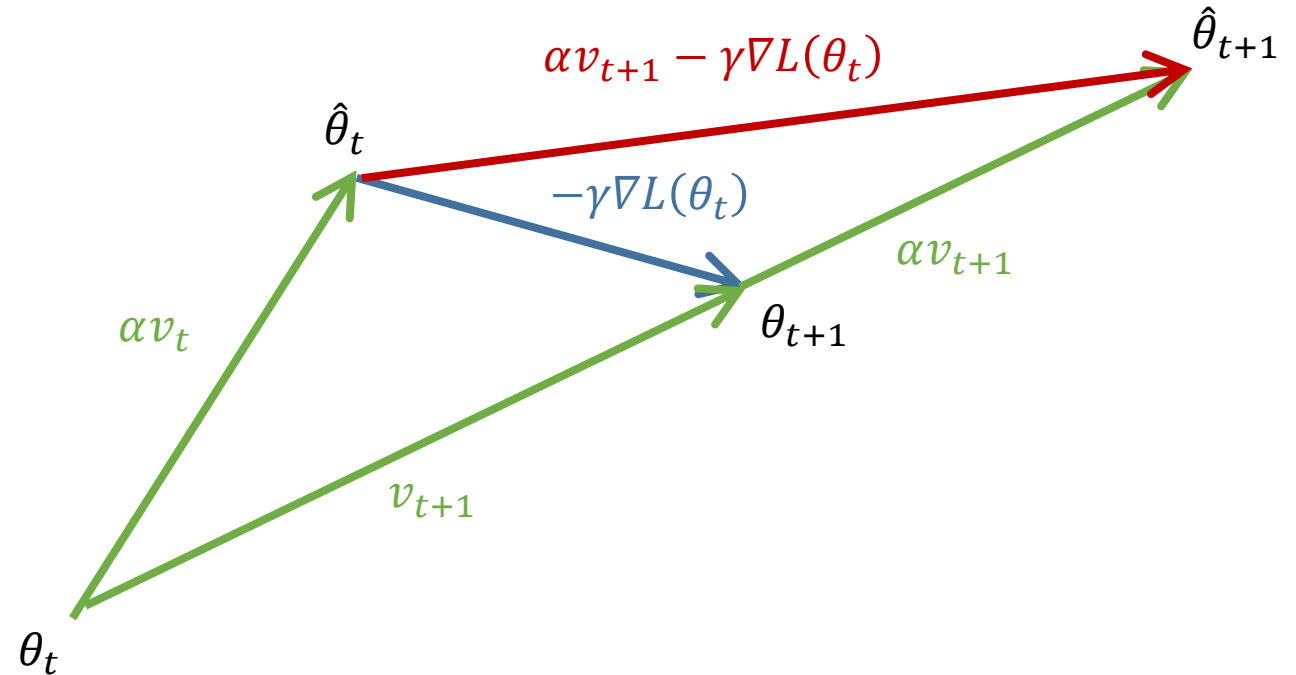
$$v_{t+1} := \alpha v_t - \gamma \nabla L(\theta_t + \alpha v_t)$$

$$\theta_{t+1} := \theta_t + v_{t+1}$$

- Pokud substituujeme za “posunuté” parametry $\hat{\theta}_t = \theta_t + \alpha v_t$:

$$v_{t+1} := \alpha v_t - \gamma \nabla L(\hat{\theta}_t)$$

$$\hat{\theta}_{t+1} := \hat{\theta}_t + \alpha v_{t+1} - \gamma \nabla L(\hat{\theta}_t)$$



- Tj. výpočet hybnosti zůstává stejný, ale změní se update parametrů

Momentum SGD vs Nesterov

- Standard momentum SGD:

$$\begin{aligned}v_{t+1} &:= \alpha v_t - \gamma \nabla L(\theta_t) \\ \theta_{t+1} &:= \theta_t + v_{t+1} \\ &:= \theta_t + \alpha v_t - \gamma \nabla L(\theta_t)\end{aligned}$$

- Nesterov s posunutými parametry:

$$\begin{aligned}v_{t+1} &:= \alpha v_t - \gamma \nabla L(\hat{\theta}_t) \\ \hat{\theta}_{t+1} &:= \hat{\theta}_t - \gamma \nabla L(\hat{\theta}_t) + \alpha v_{t+1} \\ &:= \hat{\theta}_t - \gamma \nabla L(\hat{\theta}_t) + \alpha^2 v_t - \alpha \gamma \nabla L(\hat{\theta}_t) \\ &:= \hat{\theta}_t + \alpha^2 v_t - \gamma(1 + \alpha) \nabla L(\hat{\theta}_t)\end{aligned}$$

Jelikož $\alpha < 1$, Nesterov snižuje vliv hybnosti a naopak zvyšuje vliv lokálního gradientu

RMSprop

- Root mean square
- Vychází z adaptivních technik jako např. AdaGrad
- Upravuje krok pro jednotlivé parametry, postupně sčítá jejich kvadráty (sleduje energii)
- Pokud je gradient v některých směrech neustále vyšší než jiné → normalizace
- Tzn. zmenšuje “protáhlé” dimenze = zvětšuje “splácnuté” → narovnává

decay rate ... hyperparametr, typ. $\beta = 0.99$

průměrná norma gradientů $\rightarrow u_{t+1} := \beta \cdot u_t + (1 - \beta) \cdot (\nabla L(\theta_t))^2$
(prvkově pro každý parameter
 \rightarrow stejný rozměr jako gradient)

prvkově na druhou

$$\theta_{t+1} := \theta_t - \gamma \cdot \frac{\nabla L(\theta_t)}{\sqrt{u_{t+1} + \epsilon}}$$

Adaptive Momentum (Adam)

- Kombinace Momentum SGD + RMSprop

momentum*: $v_{t+1} := \alpha \cdot v_t + (1 - \alpha) \cdot \nabla L(\theta_t)$

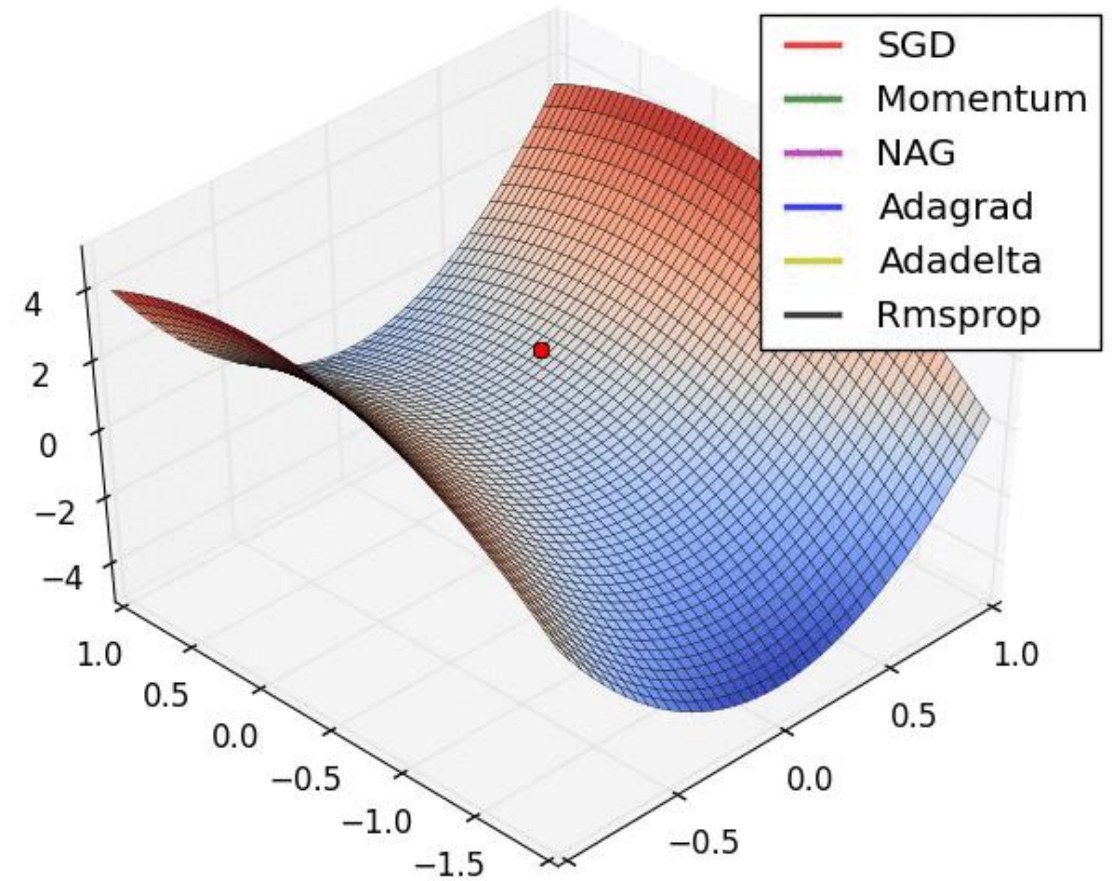
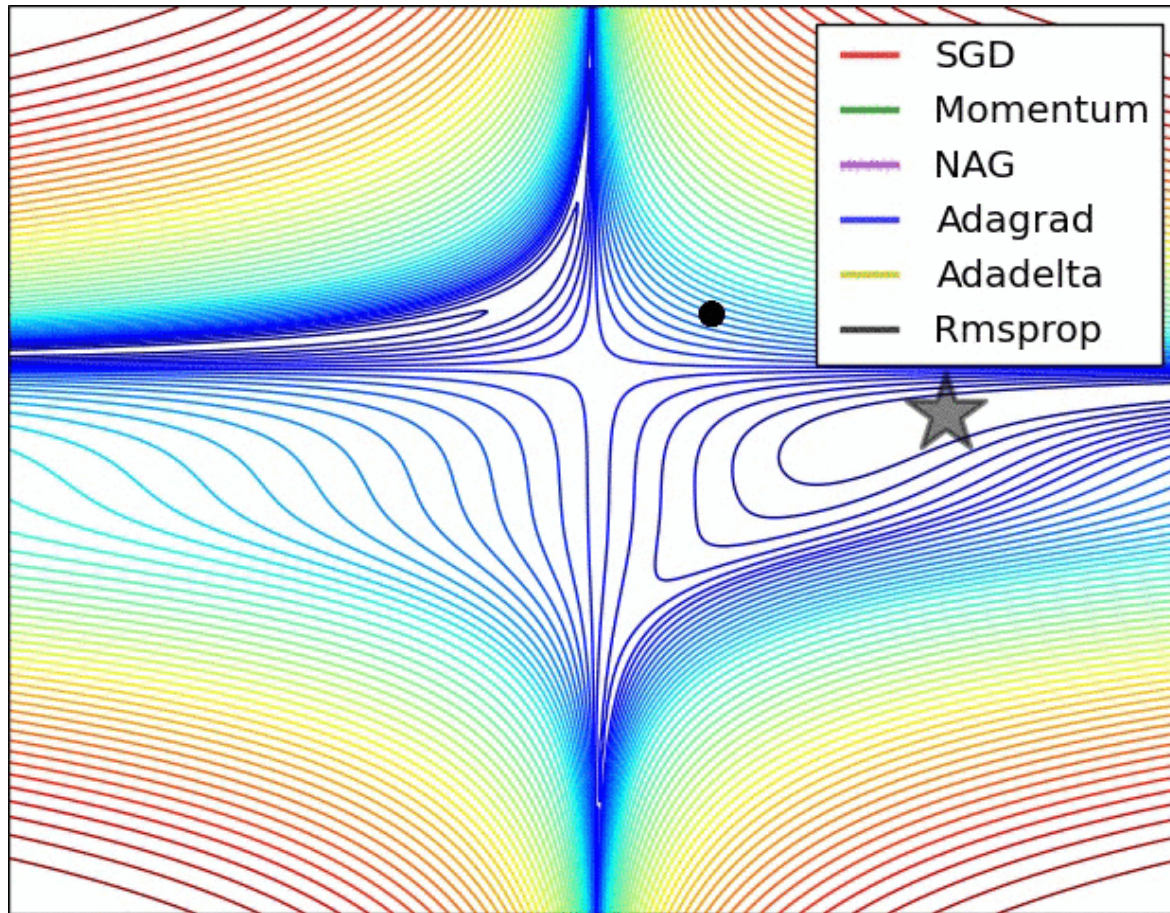
rmsprop: $u_{t+1} := \beta \cdot u_t + (1 - \beta) \cdot (\nabla L(\theta_t))^2$

Adam update: $\theta_{t+1} := \theta_t - \gamma \cdot \frac{v_{t+1}}{\sqrt{u_{t+1} + \epsilon}}$

- Obvykle funguje dobře i s výchozím nastavením hyperparametrů
- Dobrá výchozí volba
- Overview dalších metod např. zde: <https://ruder.io/optimizing-gradient-descent>

[Kingma, Ba: Adam: A Method for Stochastic Optimization](#)

Vizualizace



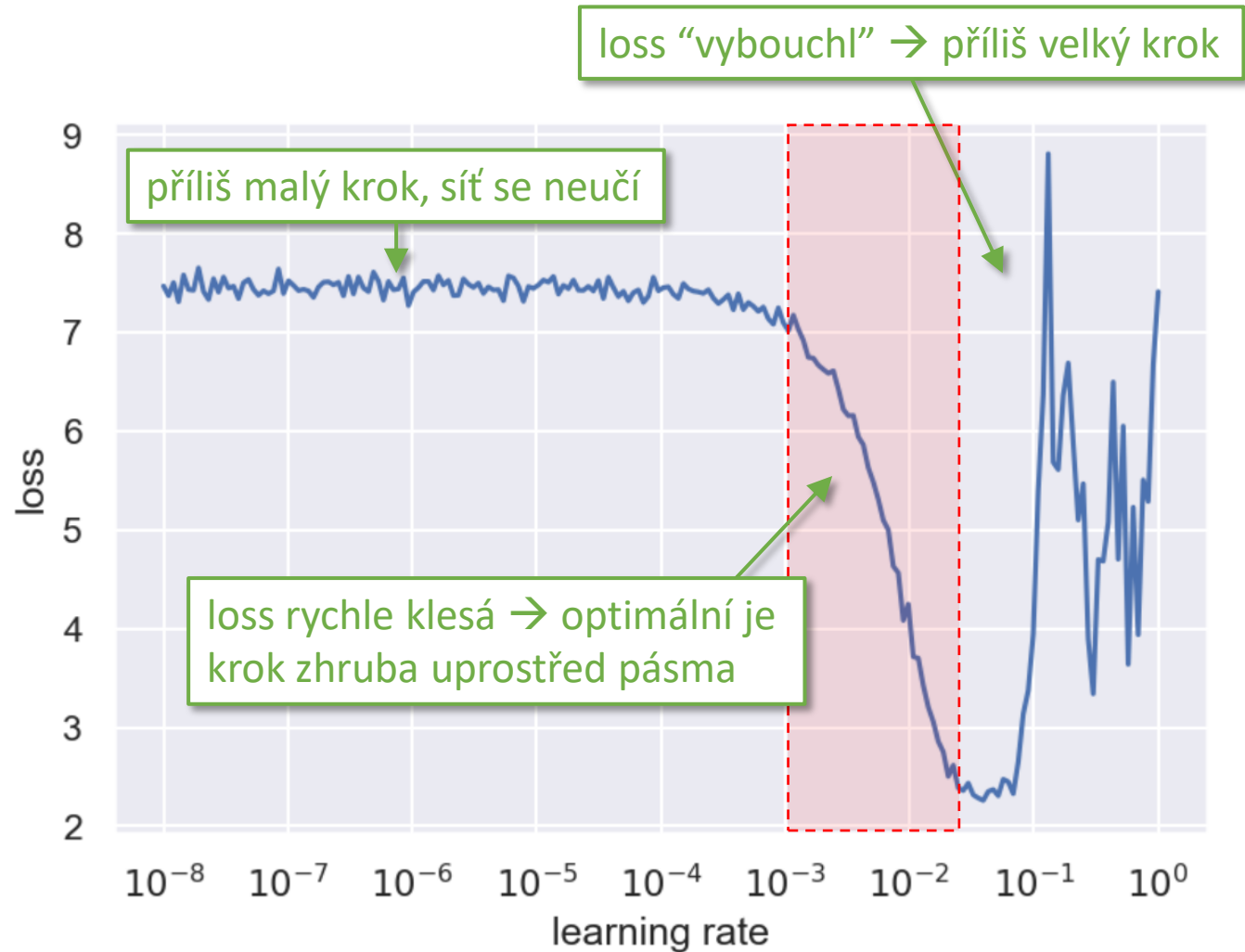
animace: <https://ruder.io/optimizing-gradient-descent/>

Jak zvolit krok učení (learning rate finder)

```
learning_exps = np.linspace(-8, 0, num=200)
data_iter = iter(train_loader)
model = torchvision.models.resnet18()
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=10**learning_exps[0]
)
losses = np.zeros((len(learning_exps),))

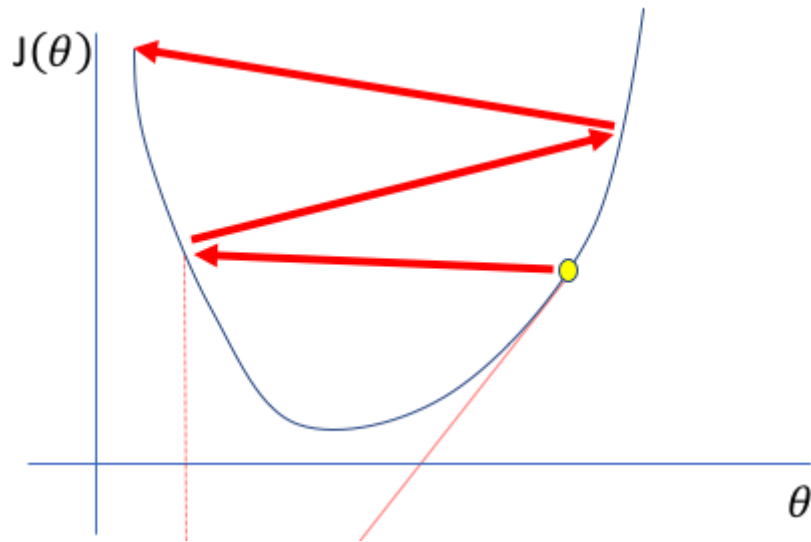
for i, le in enumerate(learning_exps):
    inputs, targets = next(data_iter)
    scores = model(inputs)
    loss = crit(scores, targets)
    losses[i] = float(loss)
    optimizer.zero_grad()
    loss.backward()
    for pg in optimizer.param_groups:
        pg['lr'] = 10 ** le
    optimizer.step()

plt.plot(learning_exps, losses)
```



Dynamický krok učení (learning rate scheduling/annealing)

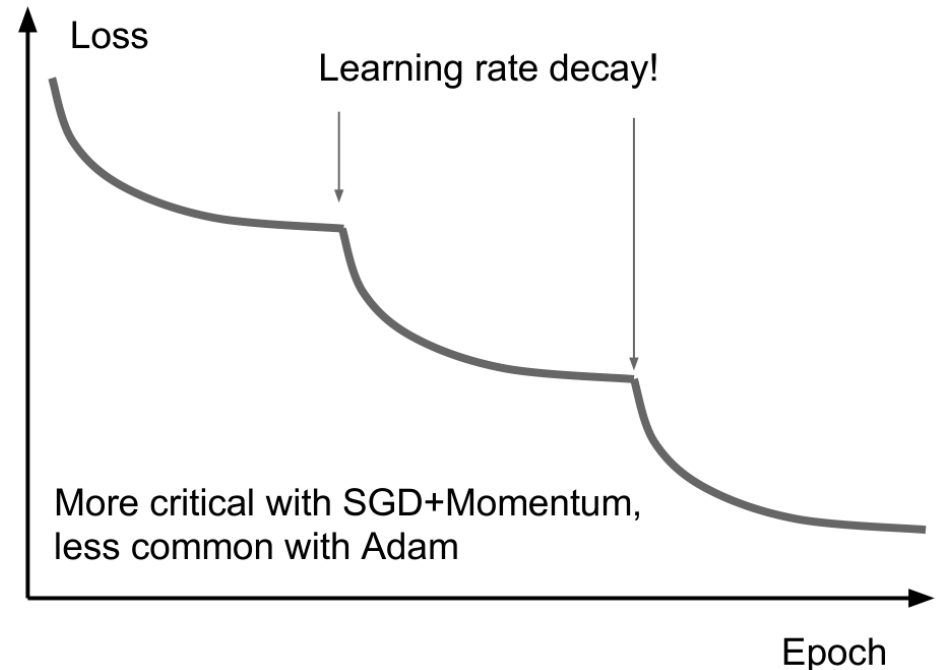
- Pokud během trénování loss neklesá



obrázek: <https://www.jeremyjordan.me/nn-learning-rate/>

- Je možné zkusit zmenšit learning rate, obvykle např. na polovinu nebo desetinu
- Lze opakovat vícekrát
- Např. 50 epoch $lr=0.1$, pak 25x $lr=0.01$ a nakonec 25x $lr=0.001$ (celkem 100 epoch)

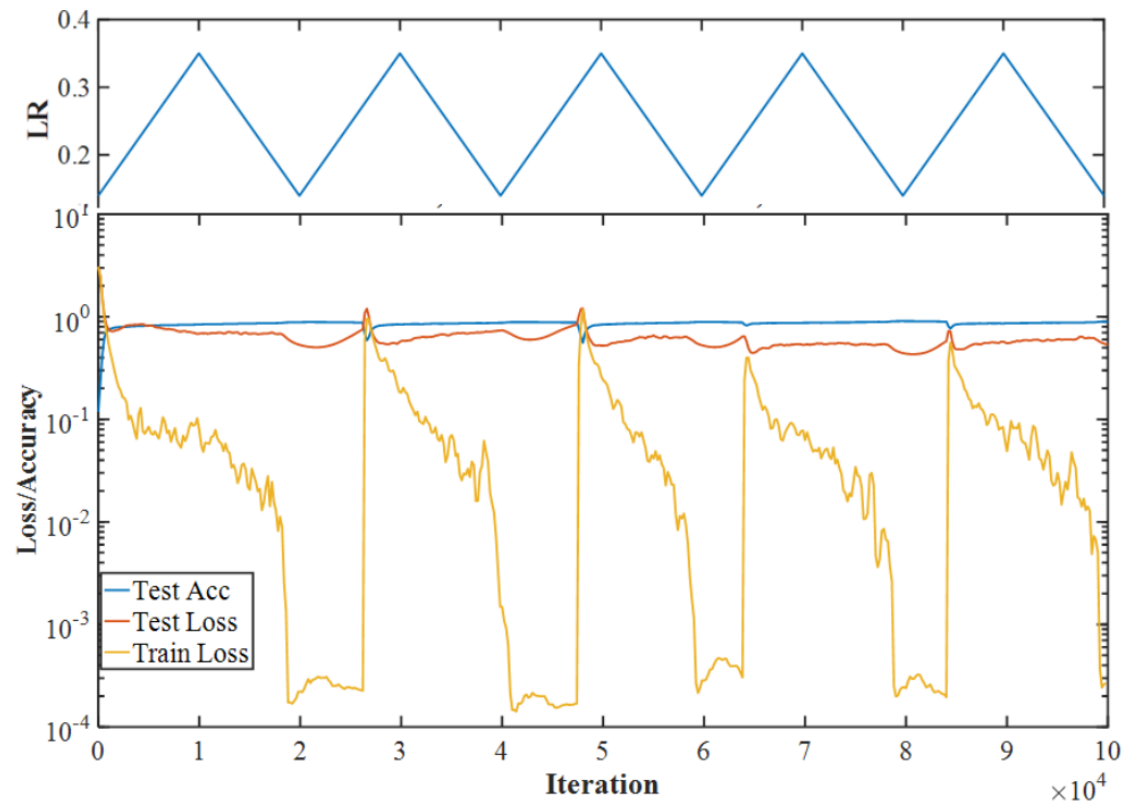
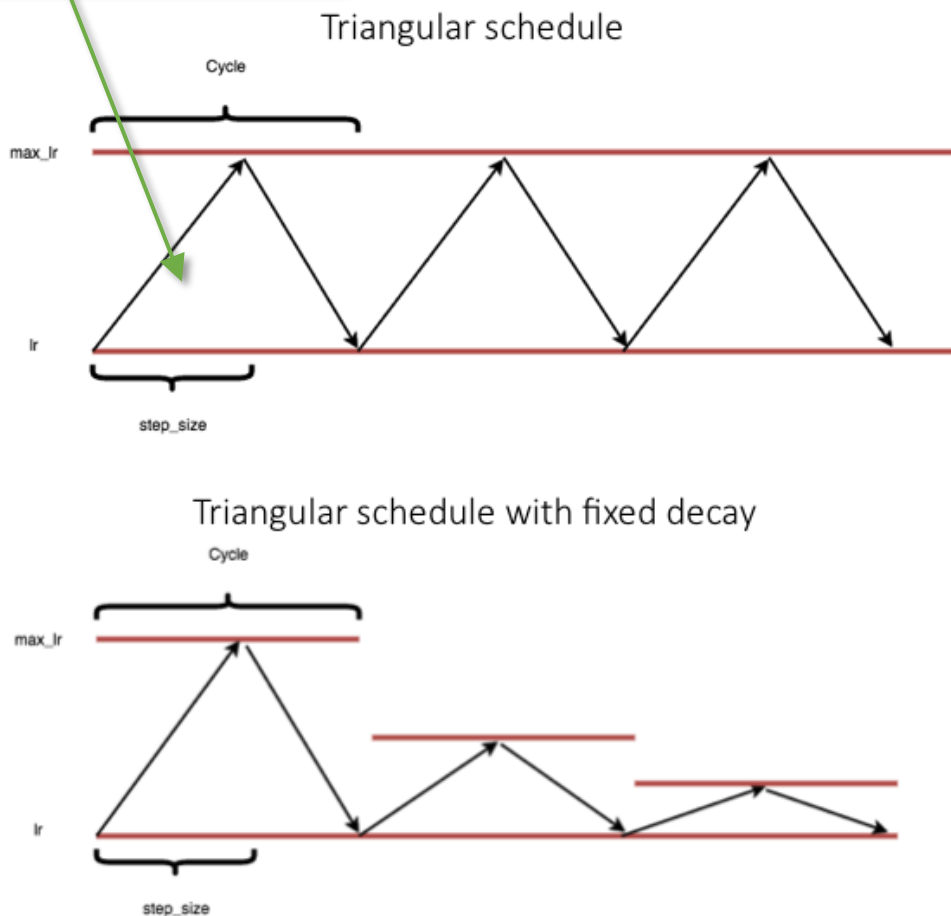
Step LR decay
typický průběh lossu



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

Cyklický krok učení

batch iterace, ale i epochy

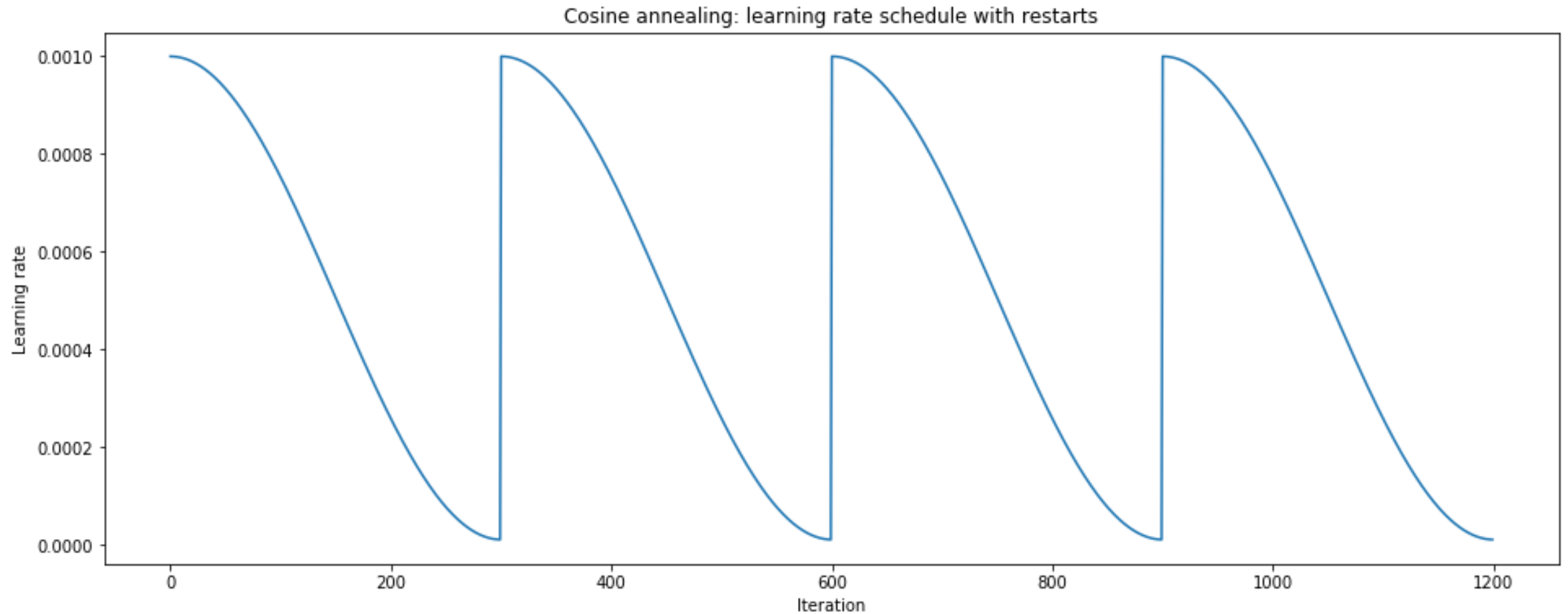


(a) Cyclical learning rate between $\text{LR}=0.1$ and $\text{LR}=0.35$ with $\text{stepsize}=10\text{K}$.

obrázek: <https://www.jeremyjordan.me/nn-learning-rate/>

[Smith, Topin: Exploring loss function topology with cyclical learning rates](#)

Stochastic Gradient Descent with Warm Restarts (SGDR)



obrázek + výborný přehled: <https://www.jeremyjordan.me/nn-learning-rate/>

Batch normalizace, SELU

aneb další triky v rukávu

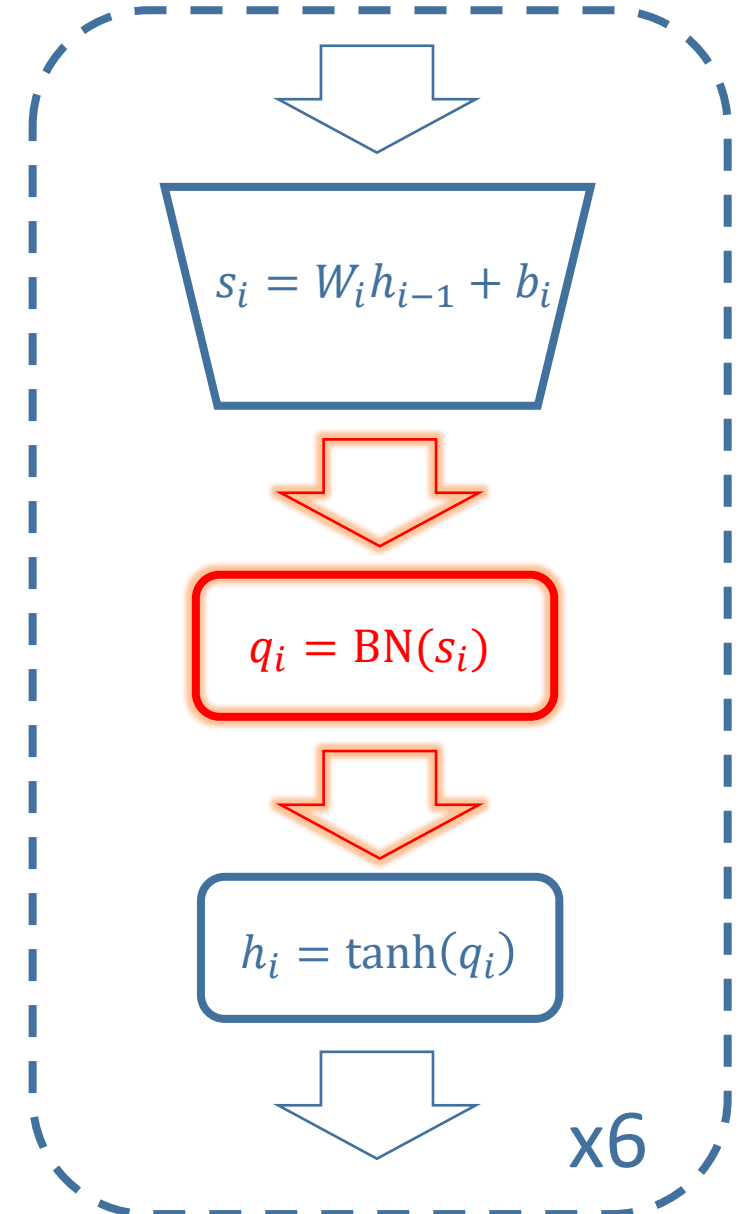
Batch normalizace (BN)

- [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 inicializací 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 normalizace

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

Parameters to be learned: γ, β

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

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

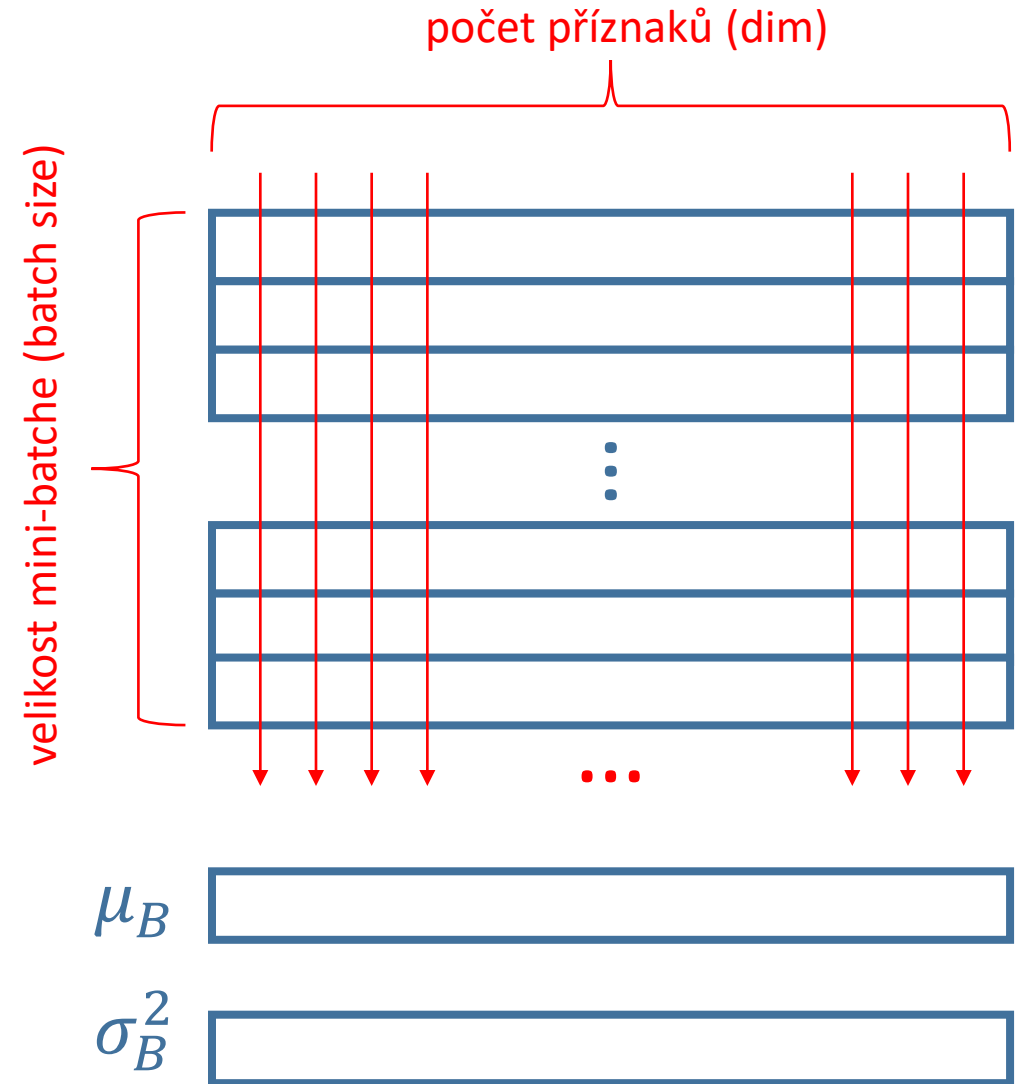
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

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

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

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	

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 zavrhané 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	

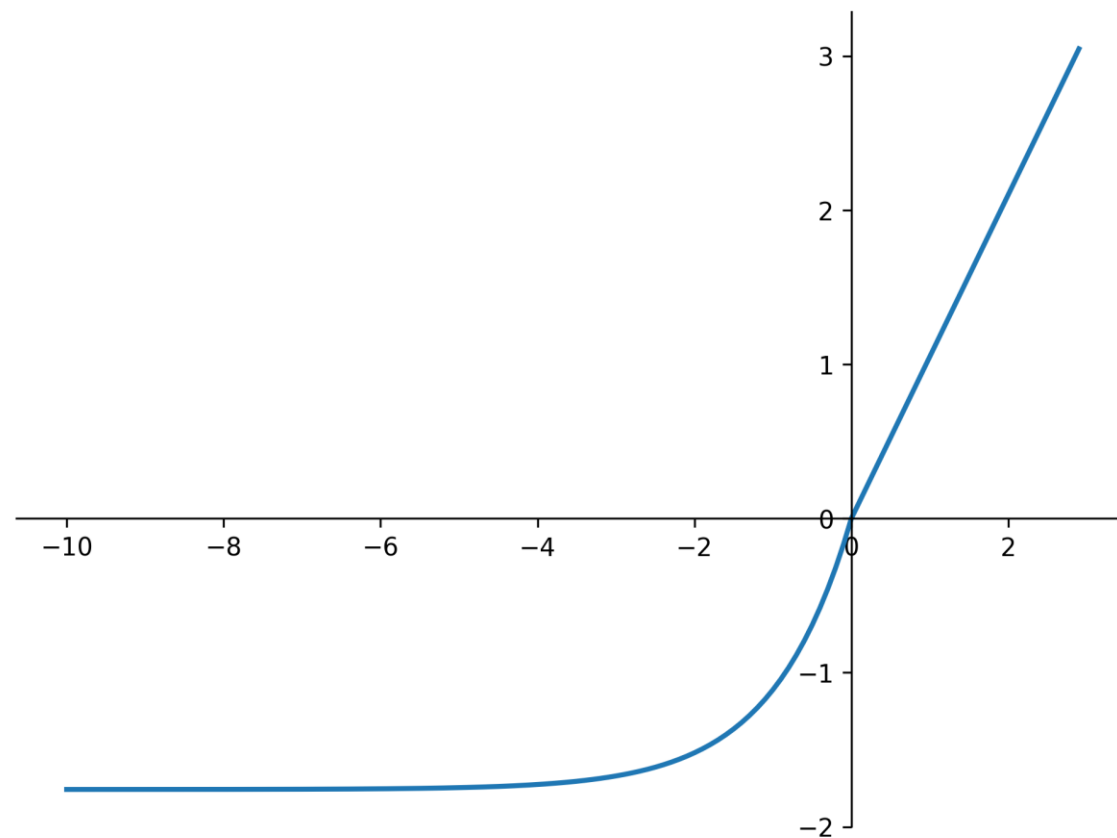
zdroj + další výsledky: <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md>

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!
 - $\lambda = 1.0507009873554804934193349852946$
 $\alpha = 1.6732632423543772848170429916717$



$$\text{ELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha \exp(x) - \alpha & \text{if } x \leq 0 \end{cases}$$



Scaled Exponential Linear Unit (SELU)

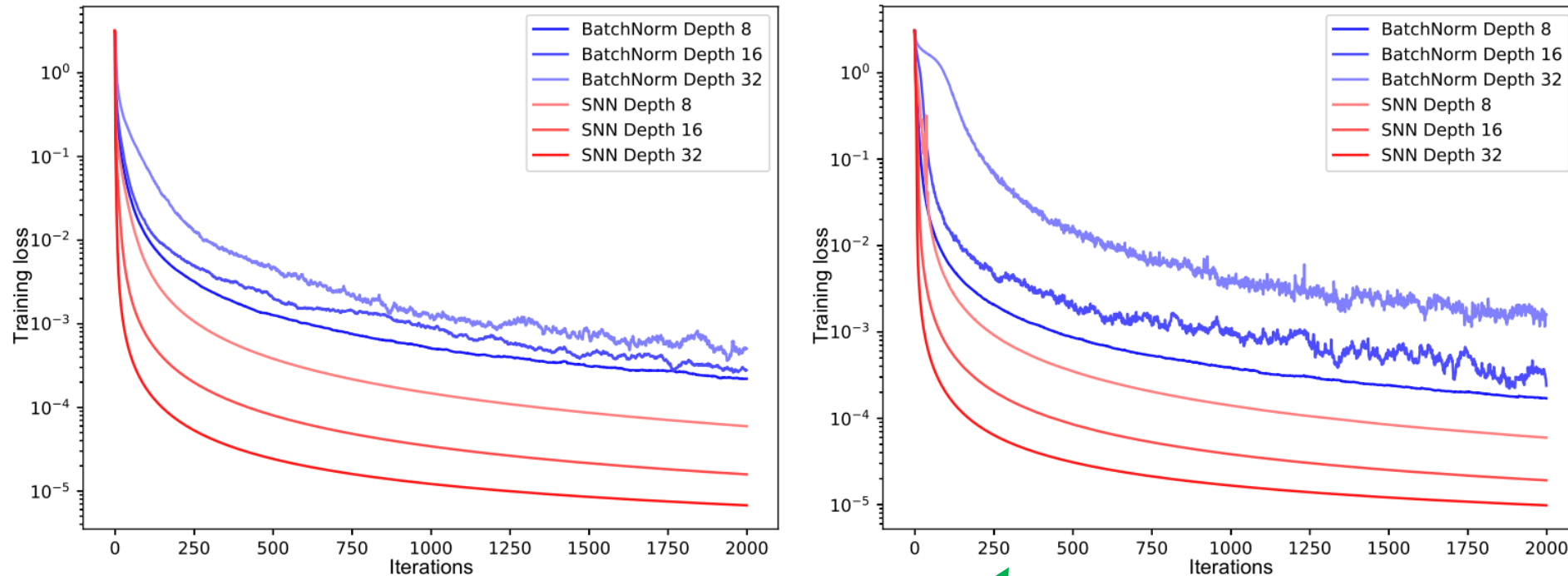


Figure 1: The left panel and the right panel show the training error (y-axis) for feed-forward neural networks (FNNs) with batch normalization (BatchNorm) and self-normalizing networks (SNN) across update steps (x-axis) on the **MNIST** dataset the **CIFAR10** dataset, respectively. We tested networks 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" \(2017\)](#)

Trénování sítě v praxi

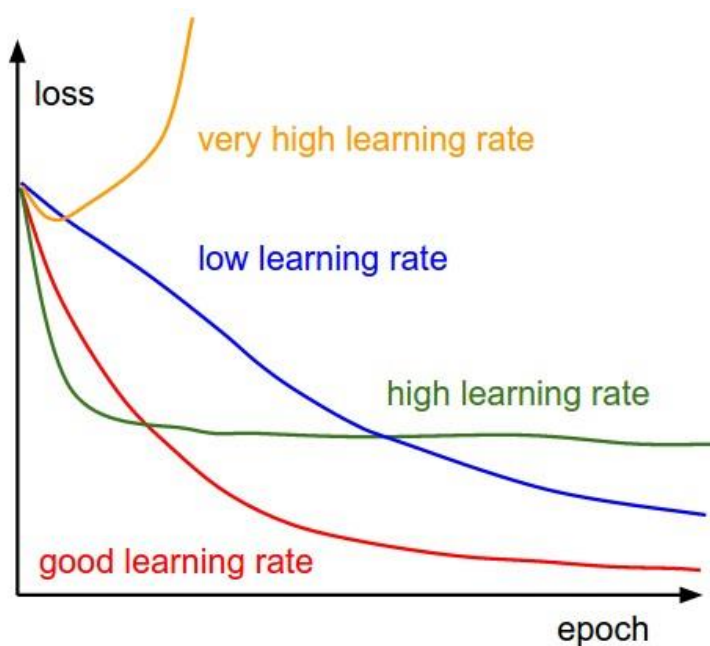
Volba sítě a nastavení trénování

- 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)

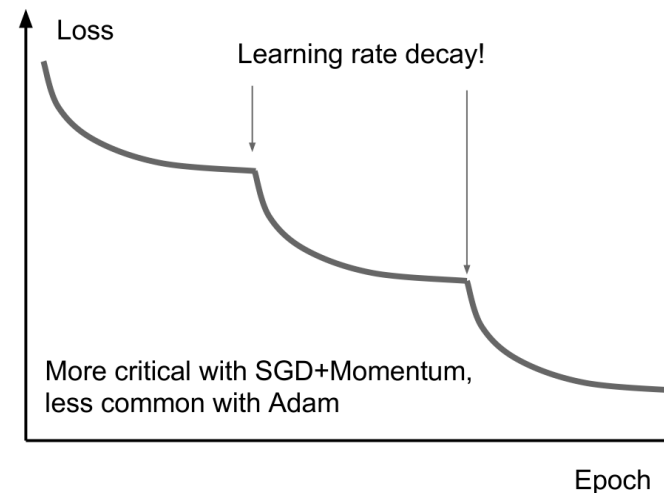
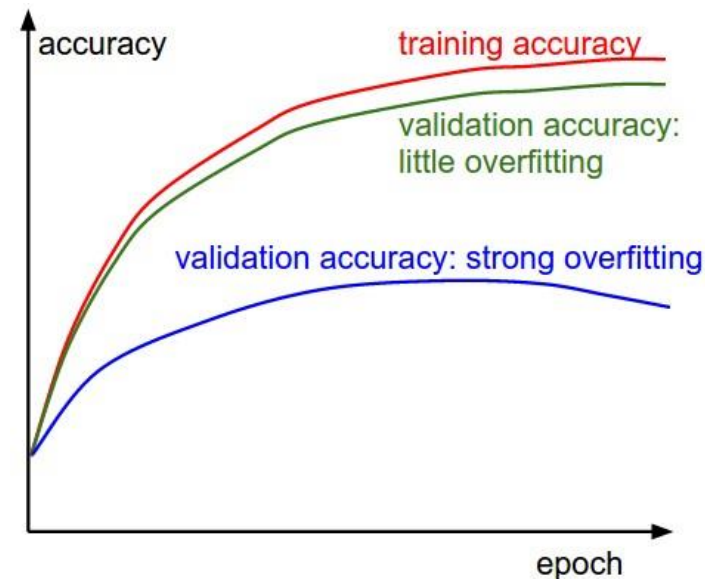
- Zkontrolovat poslední vrstvu, typ a numerický rozsah výstupů
 - Klasická chyba např. sigmoid/ReLU jako výstup, když je úloha regrese do reálných čísel $-\infty$ až $+\infty$
 - Nebo naopak nenormalizovaná lineární skóre, když výstupem má být pravděpodobnost
- Zkontrolovat hodnotu lossu na první dávce 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 \rightarrow výstupní vektor pravděpodobností je v průměru $\mathbf{q}_n \approx [0.1, \dots, 0.1]^T$ a proto SCE loss by měl začínat kolem $L_n = -\log q_{ny_n} \approx 2.302$
- Zkusit overfit na jedné dávce dat
 - Naučit jednu dávku (batch) na hodnotu lossu 0 nebo přesnost 100 %
 - Kontrola, zda “pipeline” funguje a zda je model schopný se učit

Trénování



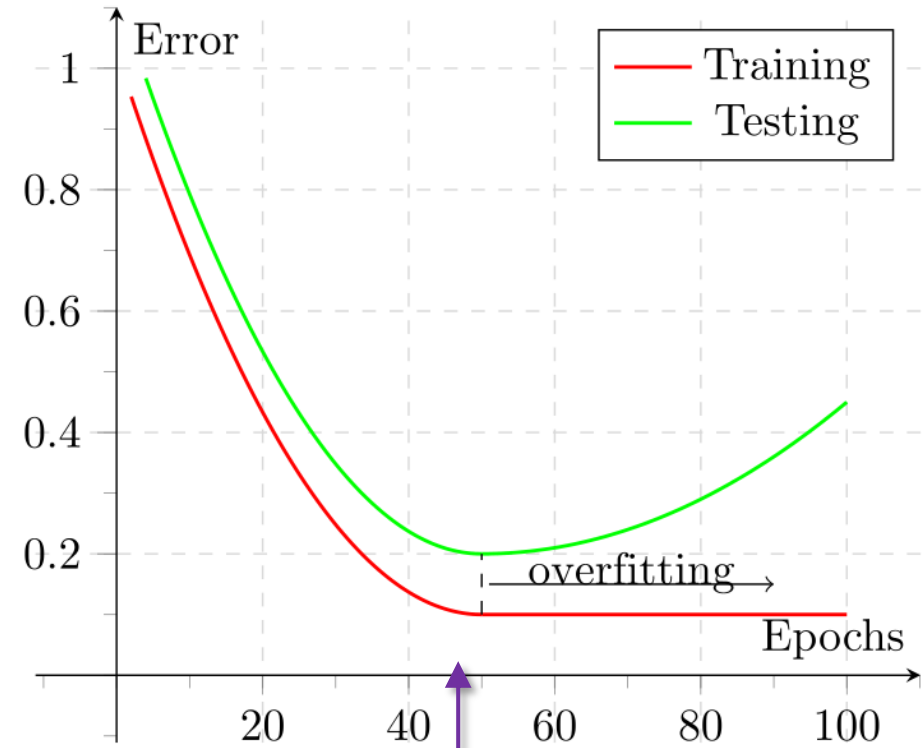
- Monitorovat hodnotu lossu a podle toho nastavit lr
- Nebo lze použít automatické hledání lr
- Pokud funguje, zkusit lr decay

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



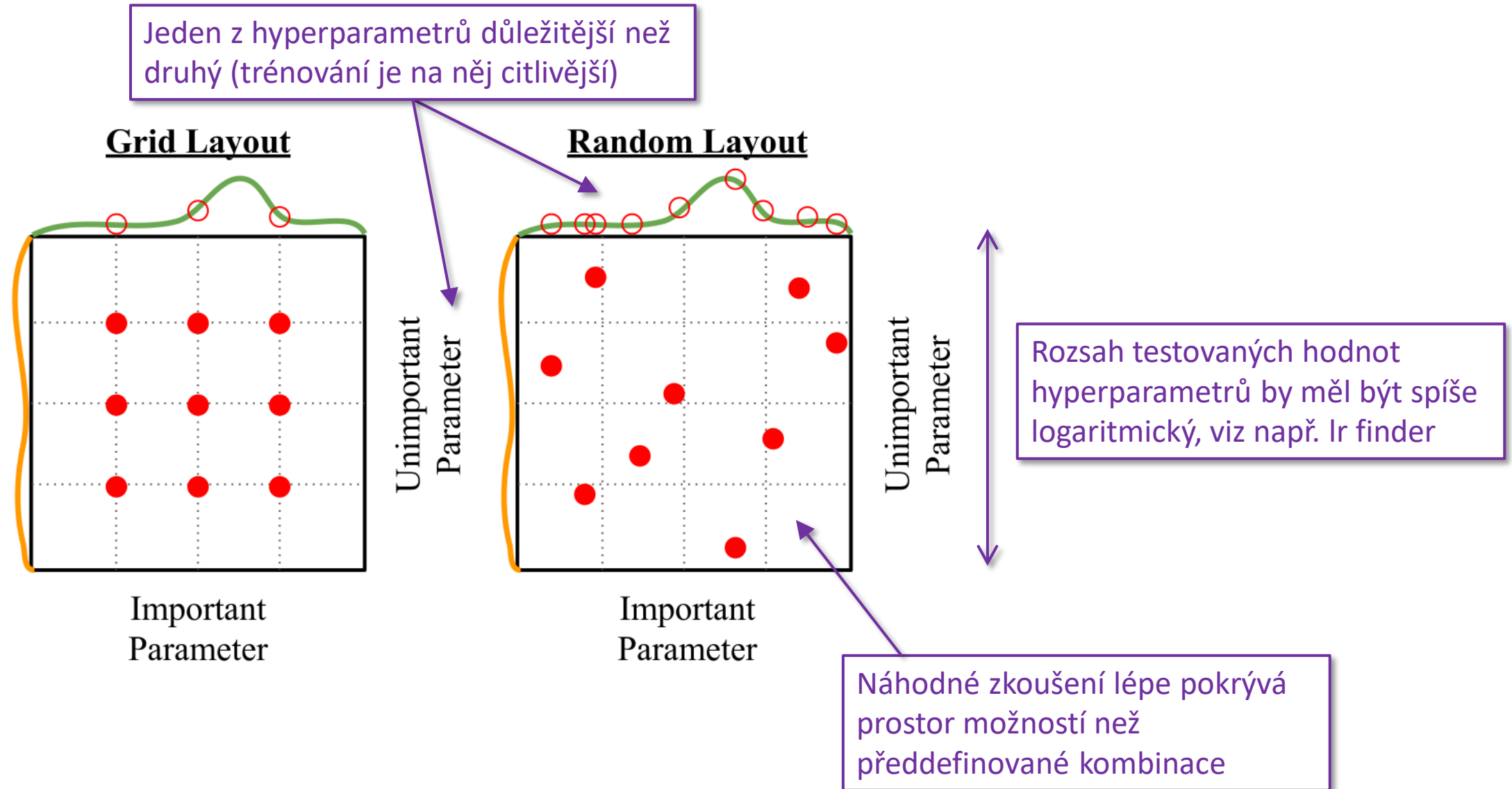
Prevence overfitu & optimalizace skóre

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



early stopping = zastavíme ještě před overfitem

Až když vše funguje: optimalizace hyperparametrů



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/>