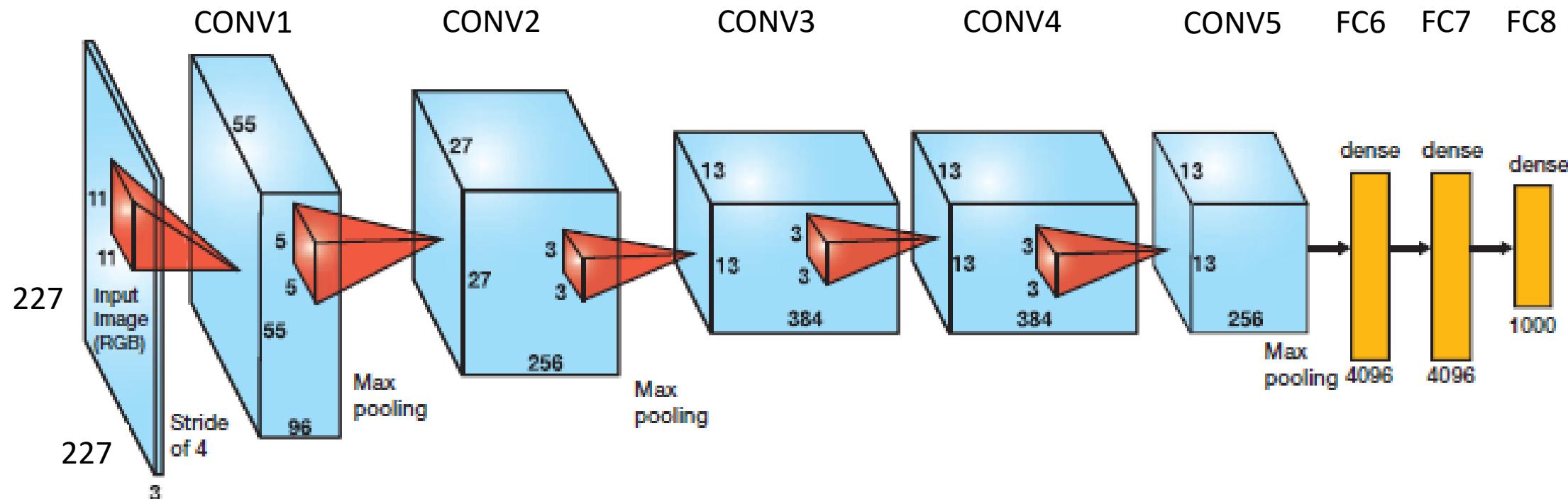


Aplikace neuronových sítí

Trénování sítí v praxi

Alexnet (2012)



- architektura: CONV-POOL-NORM-CONV-POOL-NORM-CONV-CONV-FC-FC-FC
- “naškálovaná” LeNet-5

Alexnet (2012)

- Krizhevsky, Sutskever, Hinton: “ImageNet Classification with Deep Convolutional Neural Networks”
 - ✓ Sít, která “nastartovala DNN/CNN revoluci”
 - ✓ Autoři nevyvinuli žádný nový algoritmus, “pouze” ukázali, jak správně CNN používat
 - ✓ Místo sigmoid aktivací přechod na ReLU
 - ✓ Kromě klasické L2 regularizace navíc Dropout
 - Výrazné umělé rozšiřování dat (data augmentation)
 - Místo SGD → Momentum SGD
 - Postupné snižování learning rate
 - Trénováno na dvou GTX 580 celkem 5-6 dní

Úprava dat

Předzpracování dat

- U konvolučních sítí pro obrazová data obvykle jen velmi omezené
- Cílem je end-to-end učení modelu
- 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

- Např. odečtení průměrného obrázku

```
out = rgb - mean_image
```

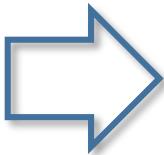
kde mean_image je 32x32x3

Umělé rozšiřování dat

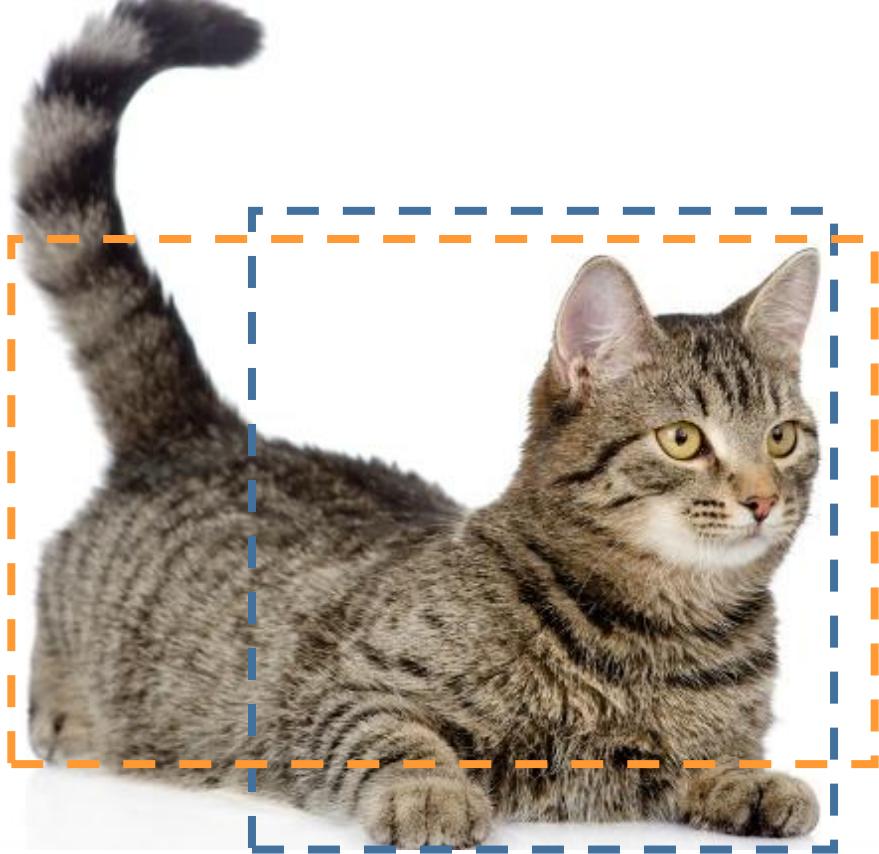
- Data augmentation
- U neurosítí téměř vždy platí: více dat = lepší výsledky
- Pokud reálná data nejsou, lze je “nafouknout” uměle, např. náhodnými transformacemi obrázků



Zrcadlení



Ořez



Další transformace

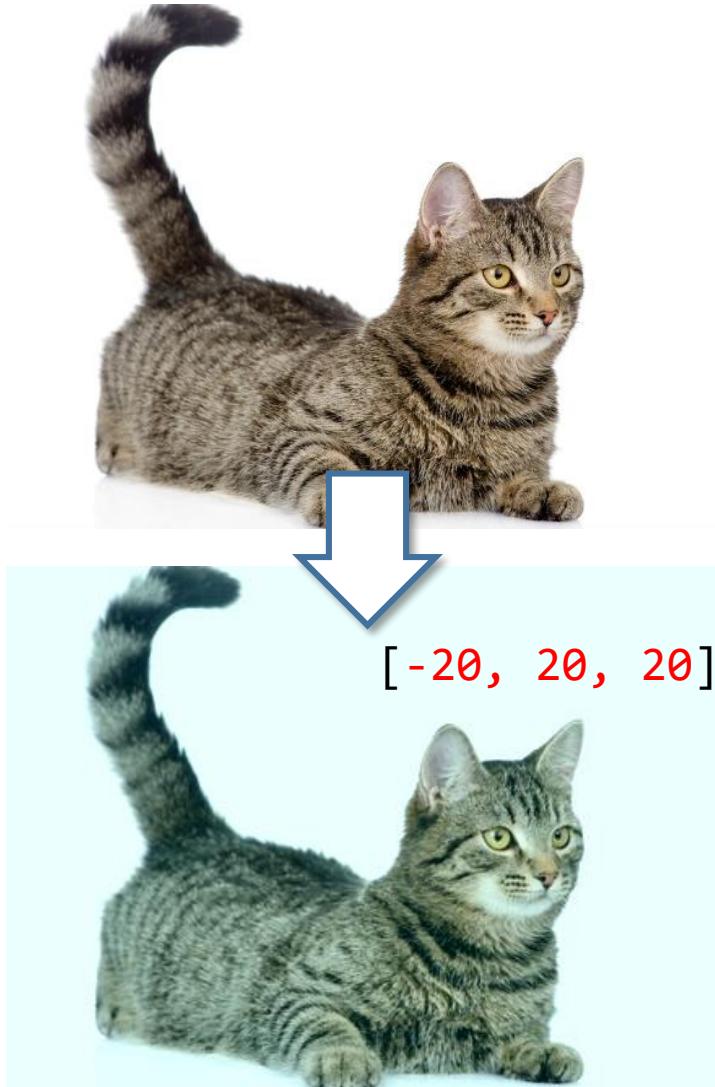
- Otočení
- Zkosení
- Lokální deformace a další
- Ne vždy ale pomáhají

Train augmentation

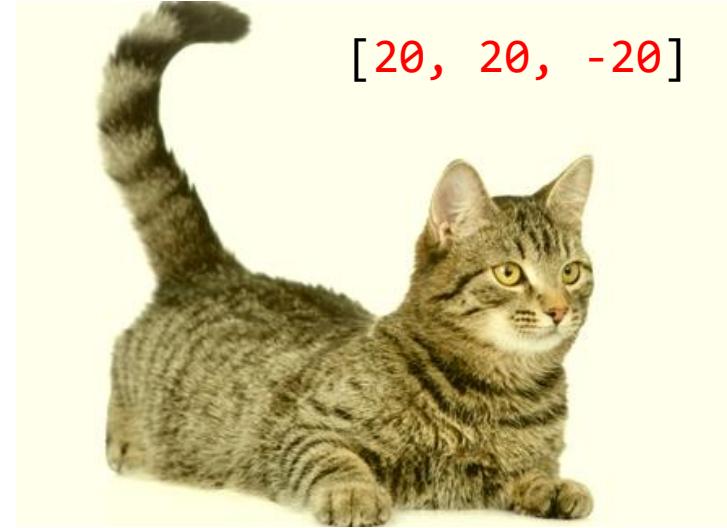
Name	Accuracy	LogLoss	Comments
Default	0.471	2.36	Random flip, random crop 128x128 from 144xN, N > 144
Drop 0.1	0.306	3.56	+ Input dropout 10%. not finished, 186K iters result
Multiscale	0.462	2.40	Random flip, random crop 128x128 from (144xN, - 50%, 188xN - 20%, 256xN - 20%, 130xN - 10%)
5 deg rot	0.448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

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

Posun barev



[-20, 20, 20]



[20, 20, -20]



[5, 0, 50]

```
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

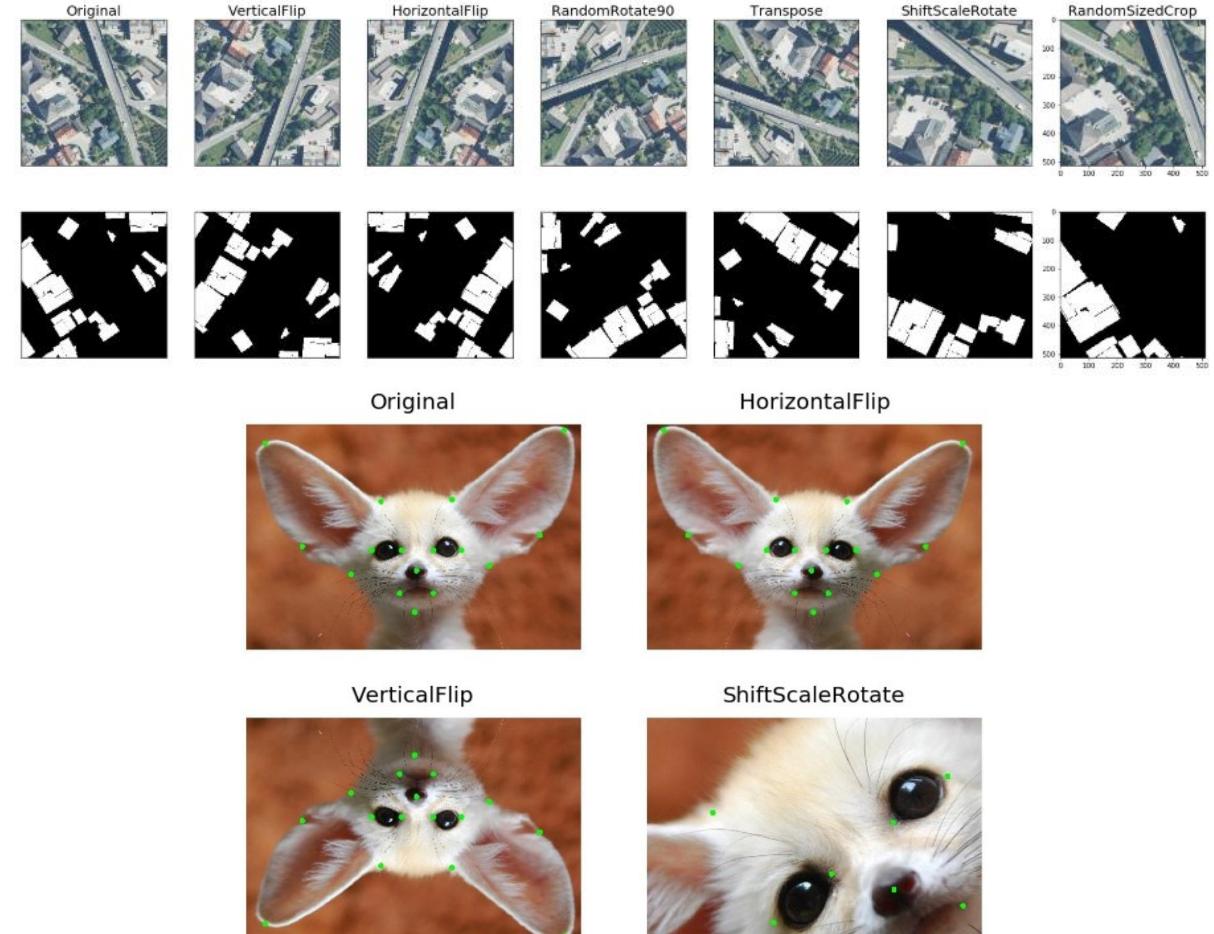
```
augment = transforms.Compose([
    transforms.Resize([256, 256]),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip()
])
prep = transforms.Compose([
    transforms.Resize([224, 224]),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 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, ...)



Optimalizační metody

Metoda největšího spádu (Gradient Descent)

Inicializace:

- parametry θ_0 na náhodné hodnoty

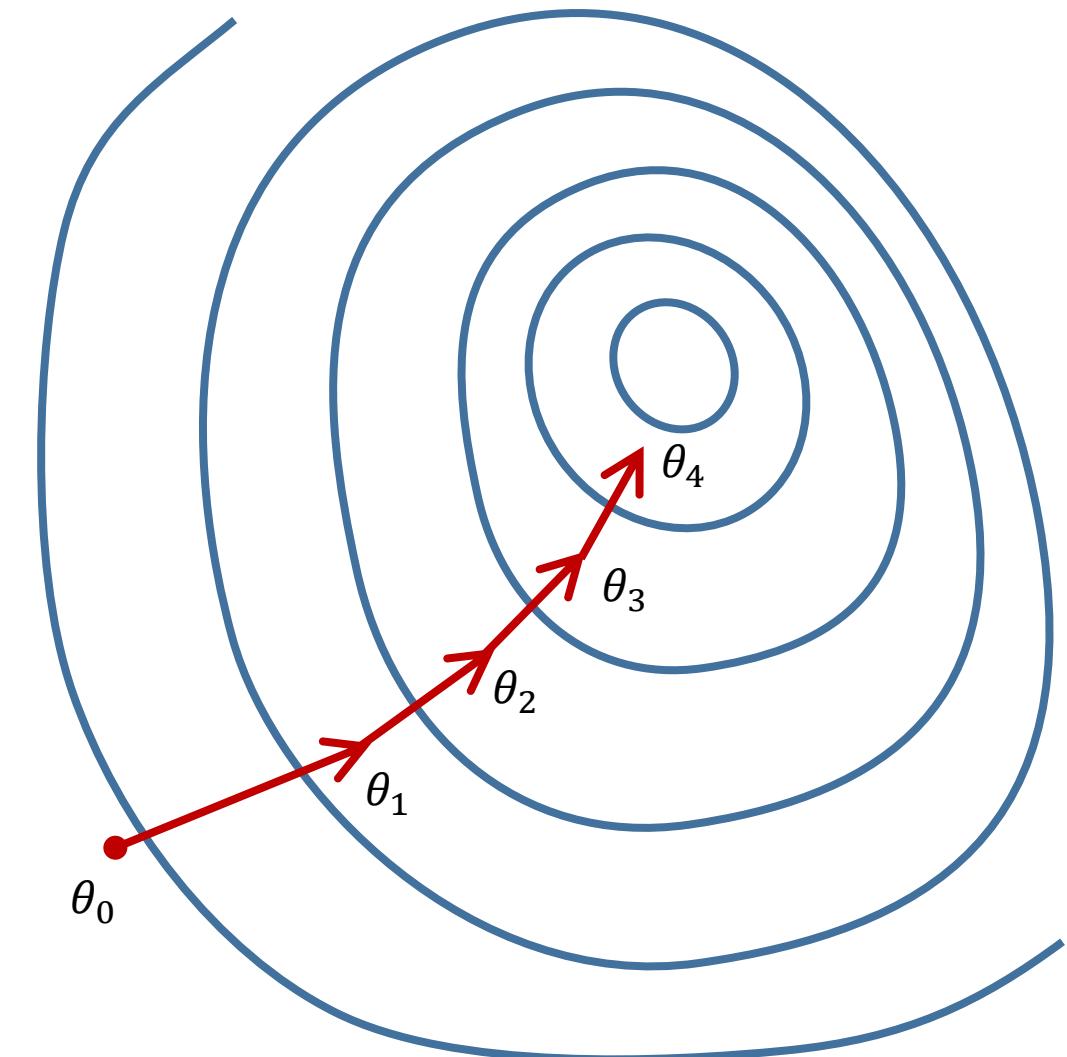
Opakujeme:

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

learning rate

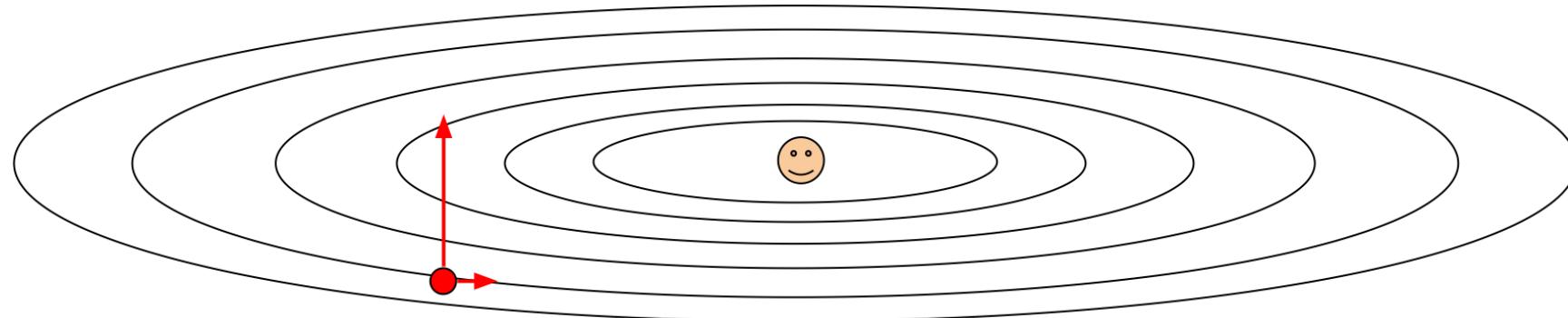
Skončíme:

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



SGD update

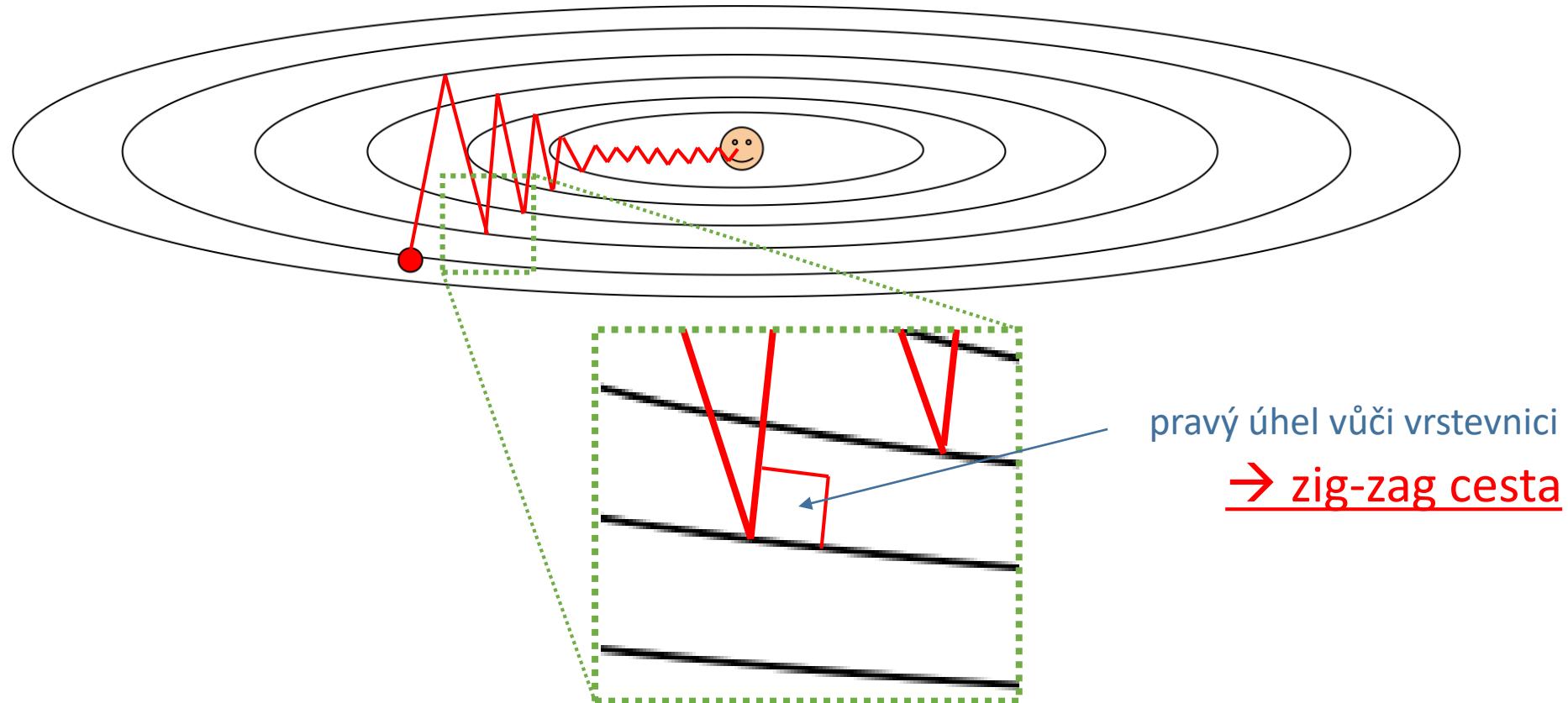
funkce, která je v jednom směru mnohem citlivější na změnu



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

SGD update

funkce, která je v jednom směru mnohem citlivější na změnu



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

Momentum SGD

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

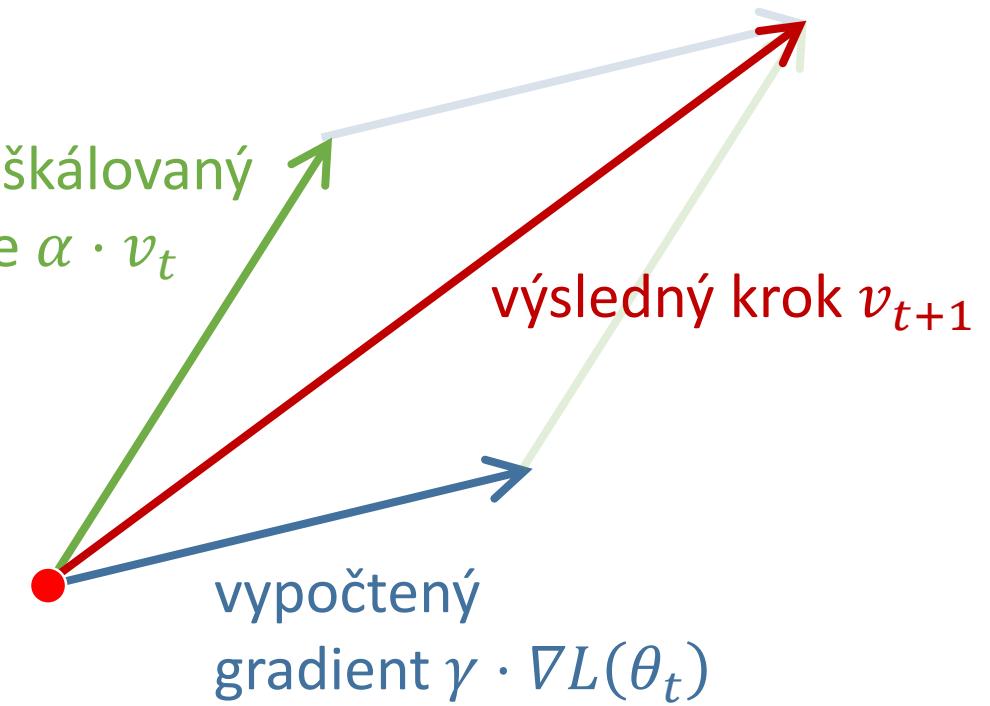
hyperparametr, např. $\alpha = 0.95$

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

learning rate (krok)

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

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



standardní SGD:

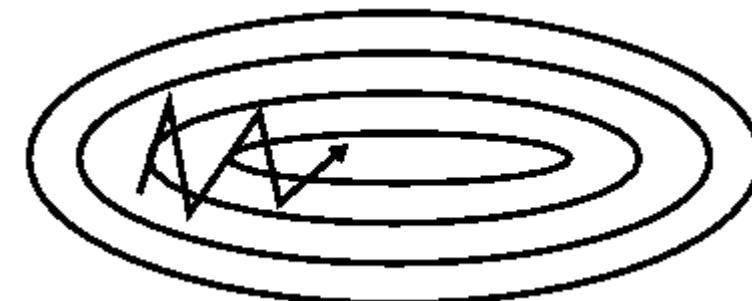
$$\theta_{t+1} := \theta_t - \gamma \cdot \nabla L(\theta_t)$$

Momentum SGD

Obyčejné SGD



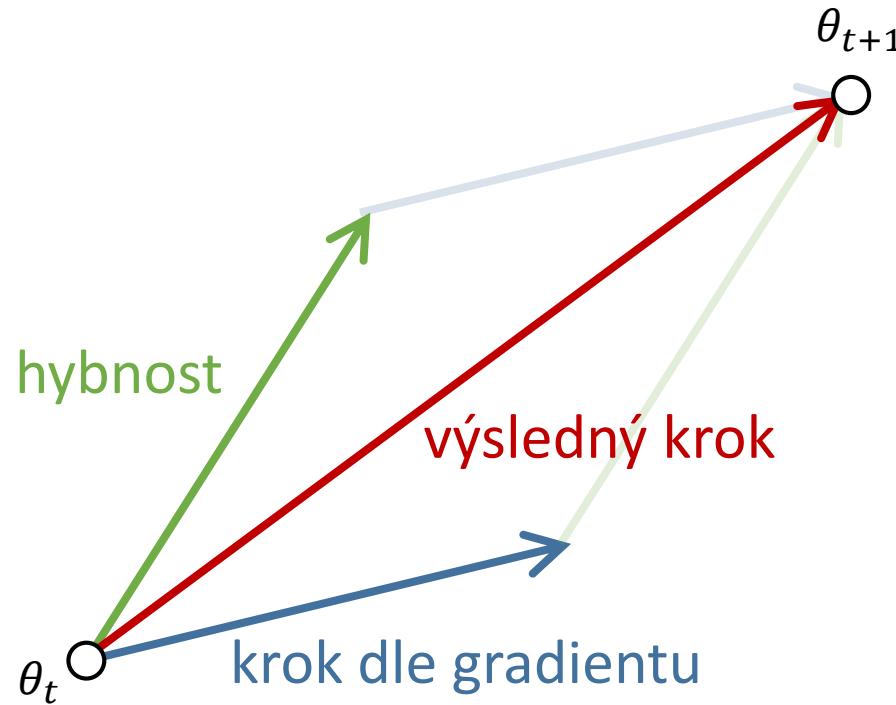
SGD + momentum



v horizontálním směru postupně nabírá rychlosť

Nesterov Accelerated Gradient (NAG)

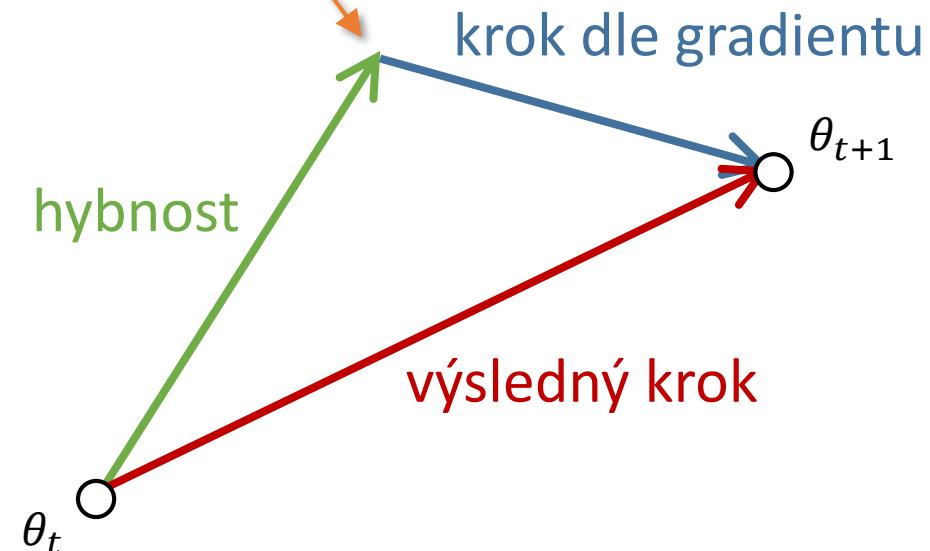
SGD + momentum



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

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

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

nejprve update hybností,
poté teprve výpočet gradientu

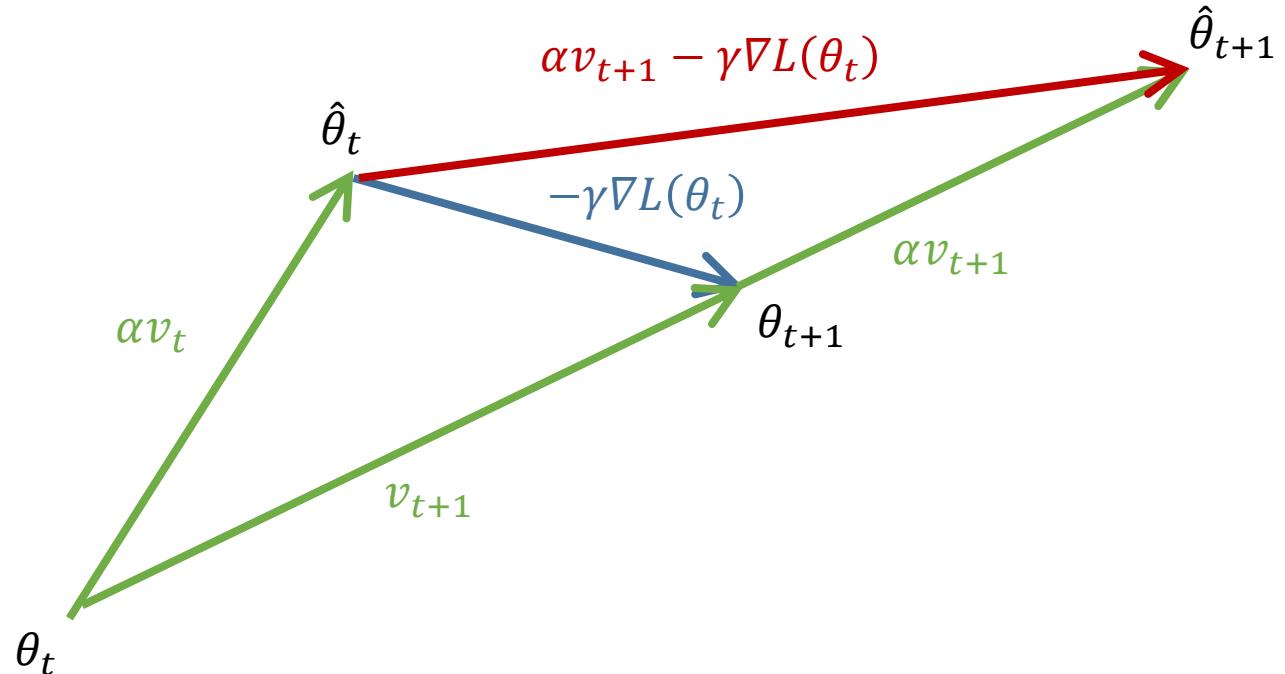
Nesterov Accelerated Gradient (NAG)

- Nesterov mechanismus:

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

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

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



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

Momentum SGD vs Nesterov

- Standard momentum SGD:

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

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

$$:= \theta_t + \alpha v_t - \gamma \nabla L(\theta_t)$$

- Nesterov s posunutými parametry:

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

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

$$:= \hat{\theta}_t - \gamma \nabla L(\hat{\theta}_t) + \alpha^2 v_t - \alpha \gamma \nabla L(\hat{\theta}_t)$$

$$:= \hat{\theta}_t + \alpha^2 v_t - \gamma(1 + \alpha) \nabla L(\hat{\theta}_t)$$

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. $\beta = 0.99$

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

průměrná norma gradientů $\rightarrow u_{t+1}$ (prvkově pro každý parameter → stejný rozměr jako gradient)

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

prvkově na druhou

Adaptive Momentum (Adam)

- Kombinace Momentum SGD + RMSprop

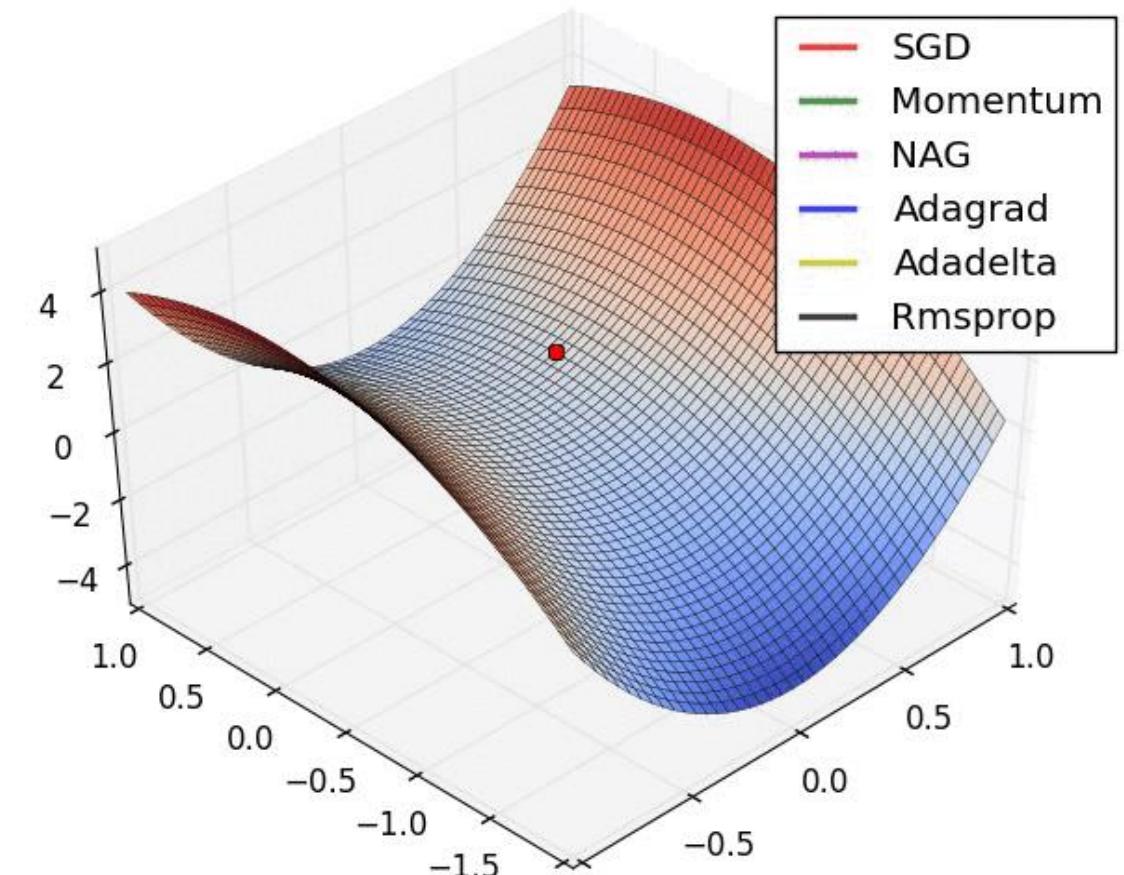
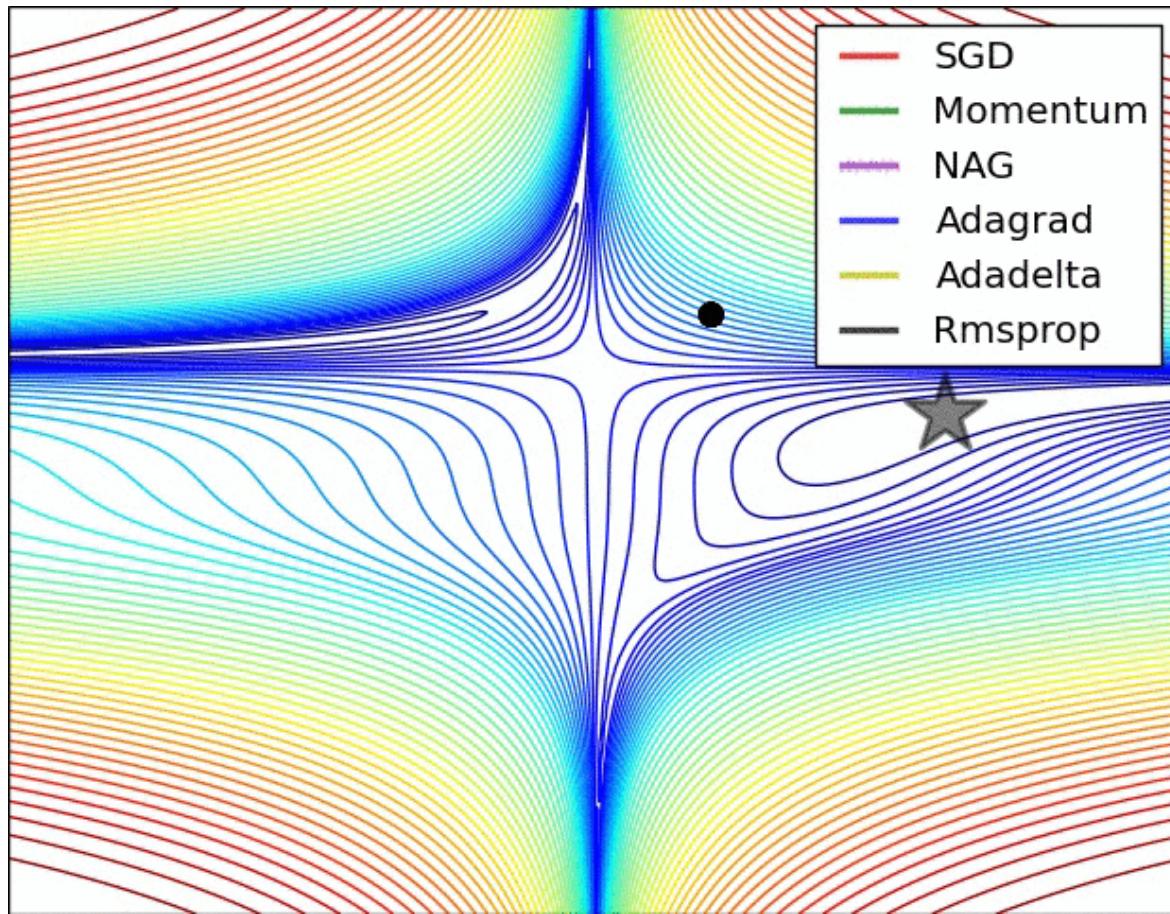
$$\text{momentum*}: \quad v_{t+1} := \alpha \cdot v_t + (1 - \alpha) \cdot \nabla L(\theta_t)$$

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

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

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

Vizualizace



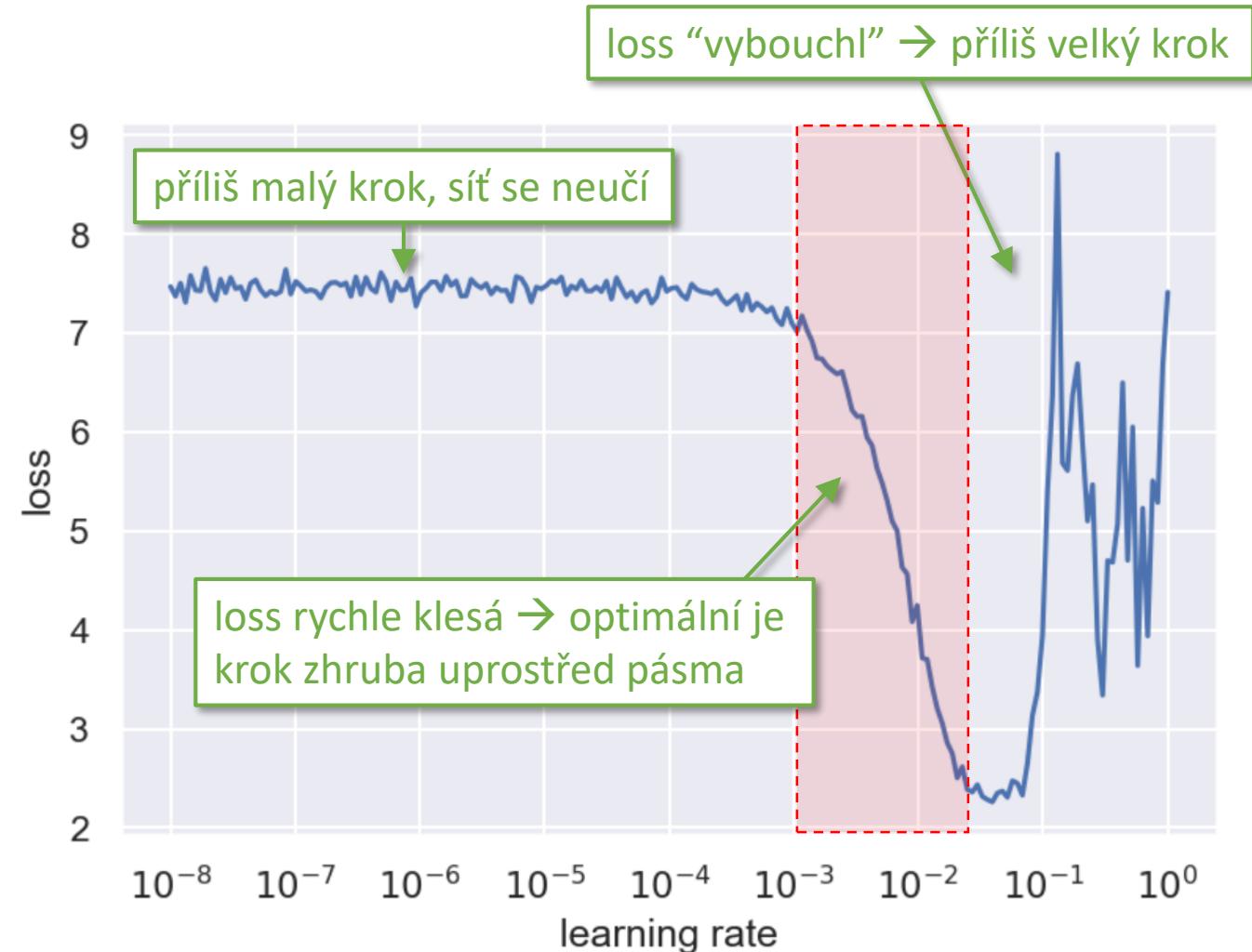
animace: <https://ruder.io/optimizing-gradient-descent/>

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),))

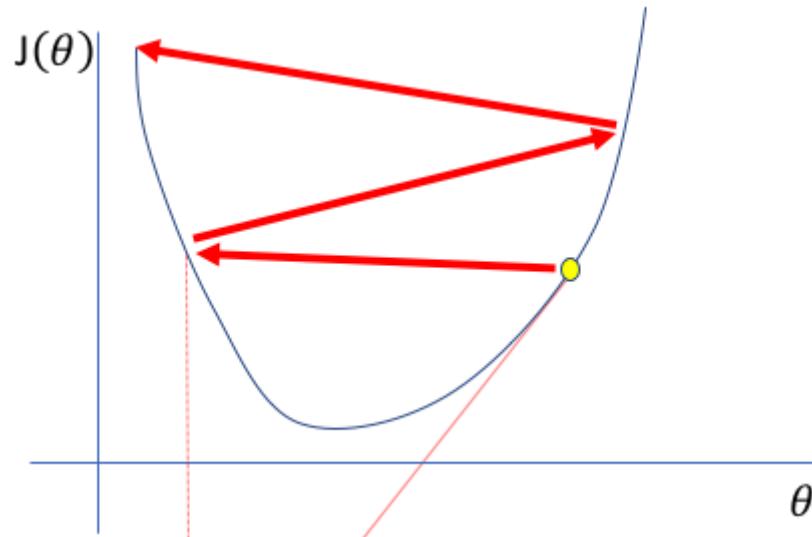
for i, le in enumerate(learning_exps):
    inputs, targets = next(data_iter)
    scores = model(inputs)
    loss = crit(scores, targets)
    losses[i] = float(loss)
    optimizer.zero_grad()
    loss.backward()
    for pg in optimizer.param_groups:
        pg['lr'] = 10 ** le
    optimizer.step()

plt.plot(learning_exps, losses)
```



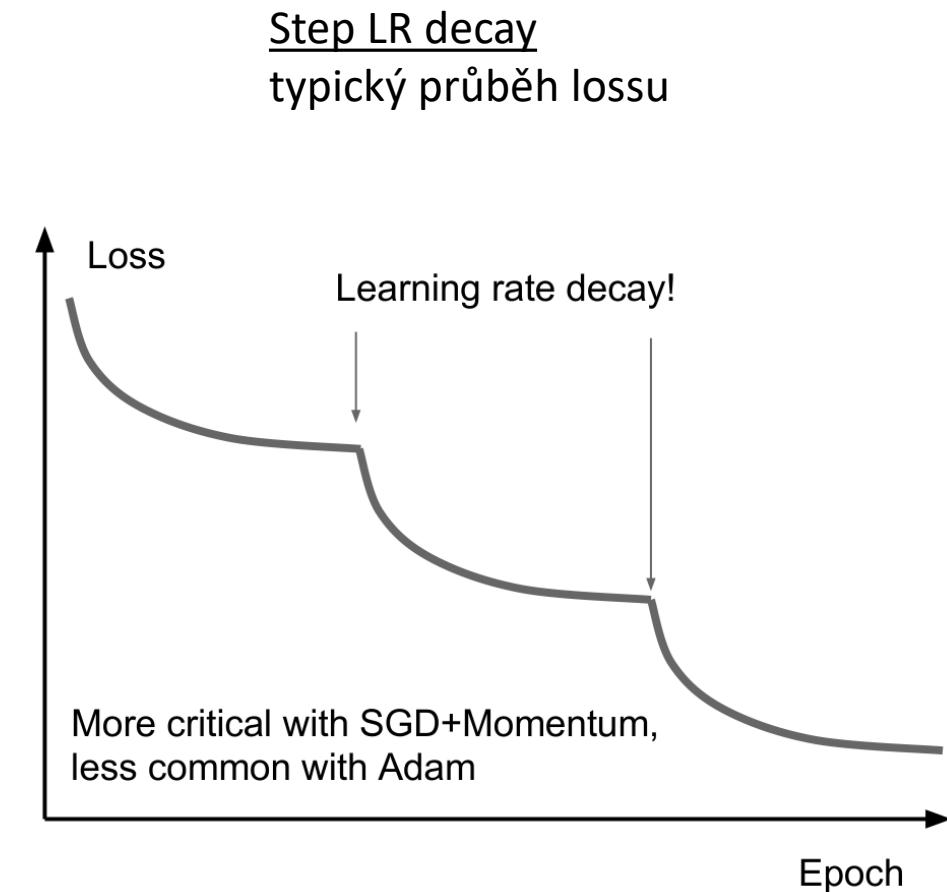
Dynamický krok učení (learning rate scheduling/annealing)

- Pokud během trénování loss neklesá



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

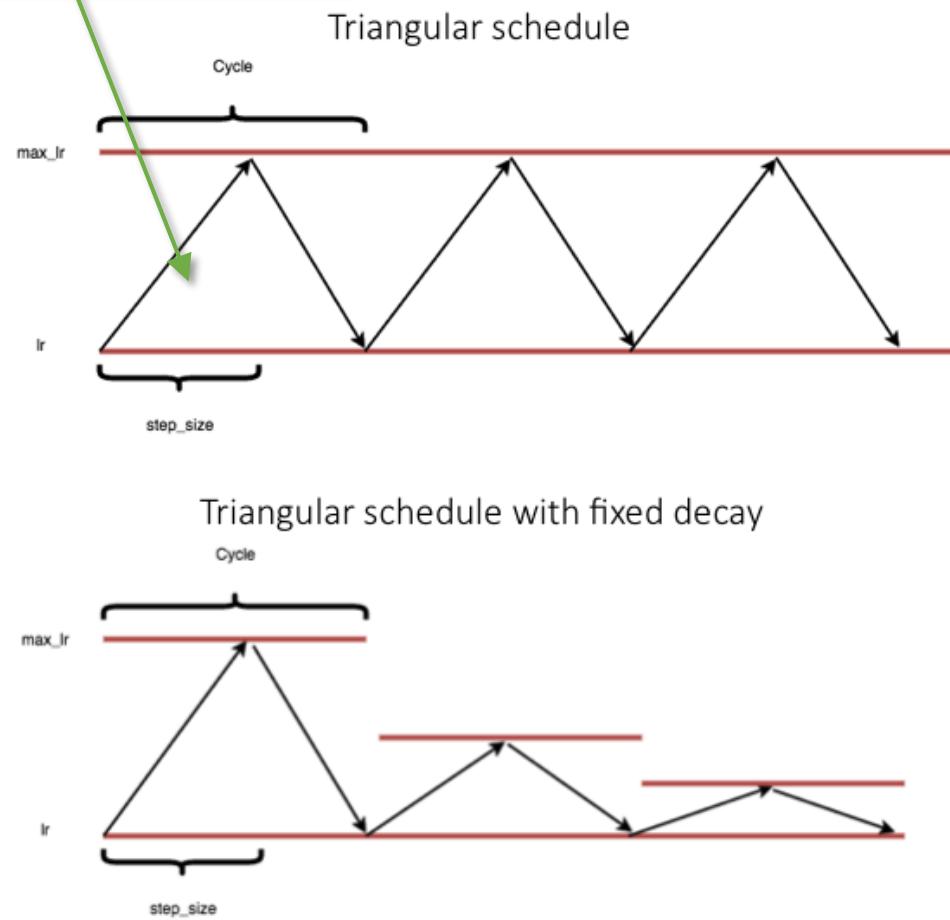
- Je možné zkusit zmenšit learning rate, obvykle např. na polovinu nebo desetinu
- Lze opakovat vícekrát
- Např. 50 epoch lr=0.1, pak 25x lr=0.01 a nakonec 25x lr=0.001 (celkem 100 epoch)



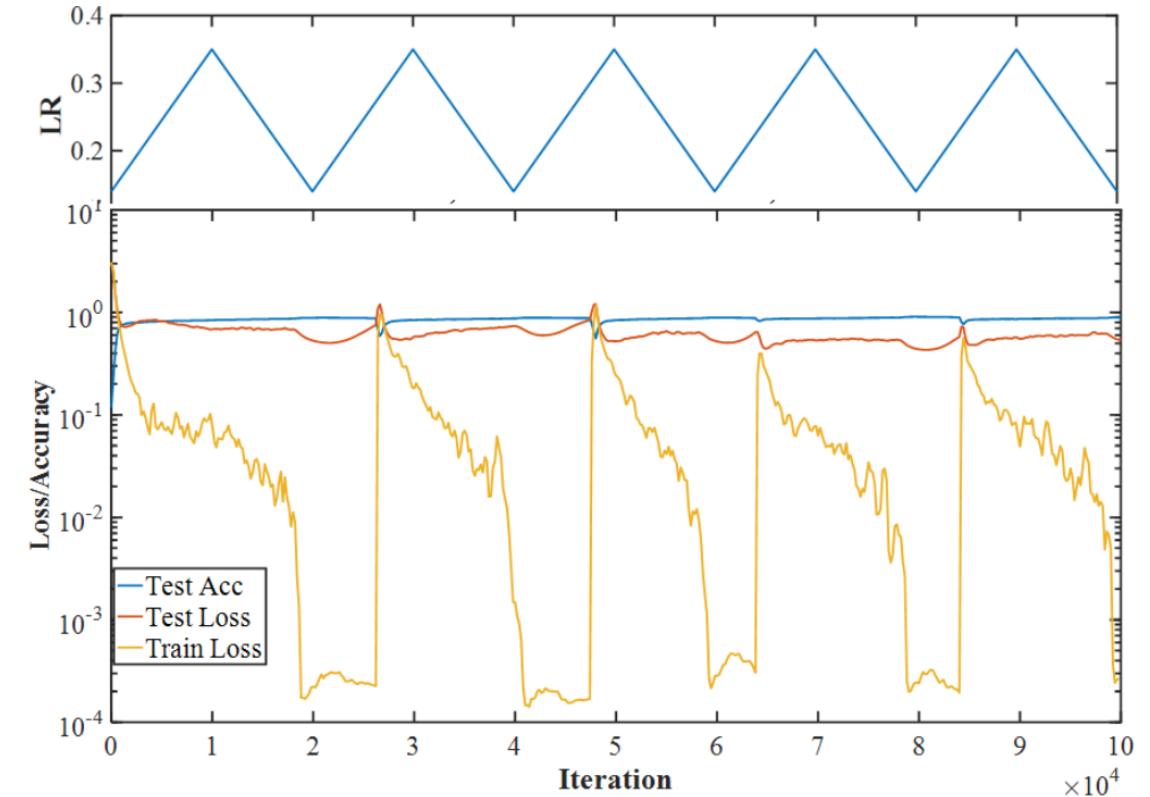
obrázek: <http://cs231n.stanford.edu/>

Cyklický krok učení

batch iterace, ale i epochy



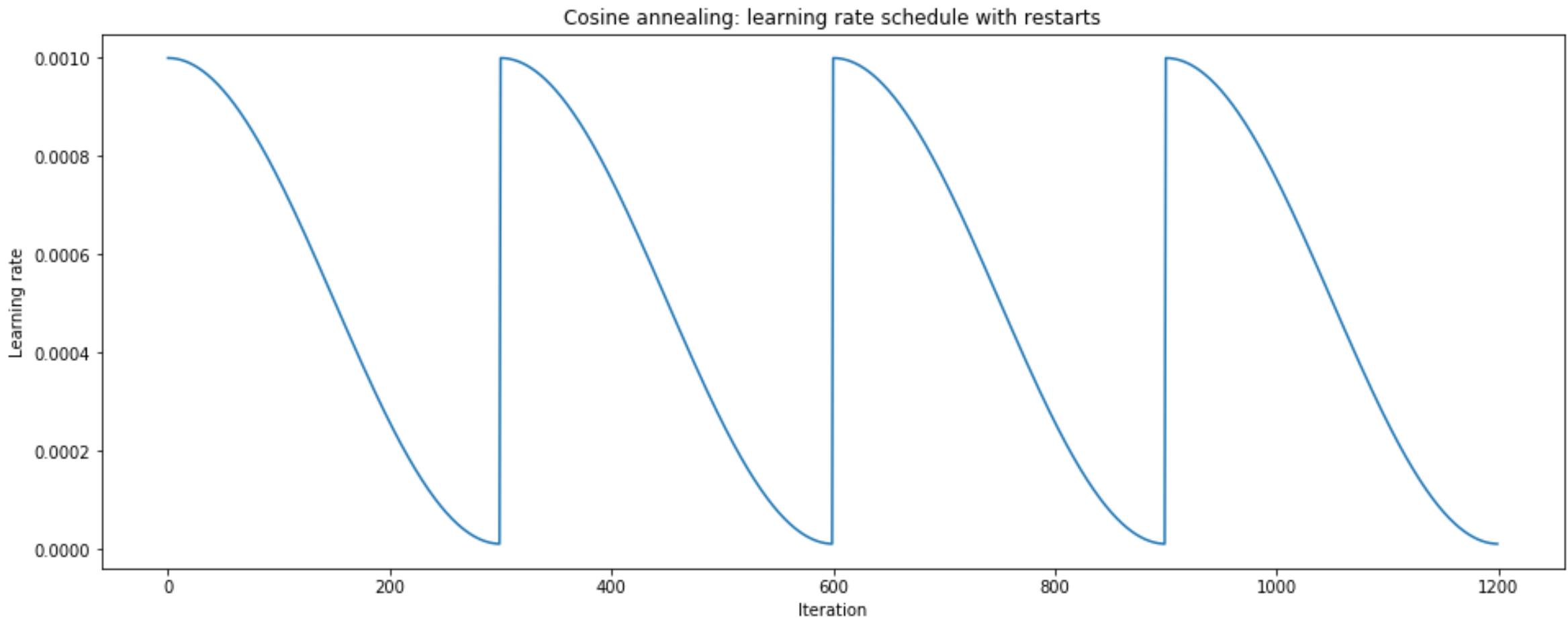
obrázek: <https://www.jeremyjordan.me/nn-learning-rate/>



(a) Cyclical learning rate between LR=0.1 and LR=0.35 with stepsize=10K.

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 podobné rozložení hodnot v různých vrstvách tak, aby žádná vrstva “nezabíjela” gradient
- Obtížné zajistit inicializací a aktivacemi / nelinearitami
- Co prostě výstup vrstvy normalizovat?
- Např. na nulový průměr a std. odchylku 1:

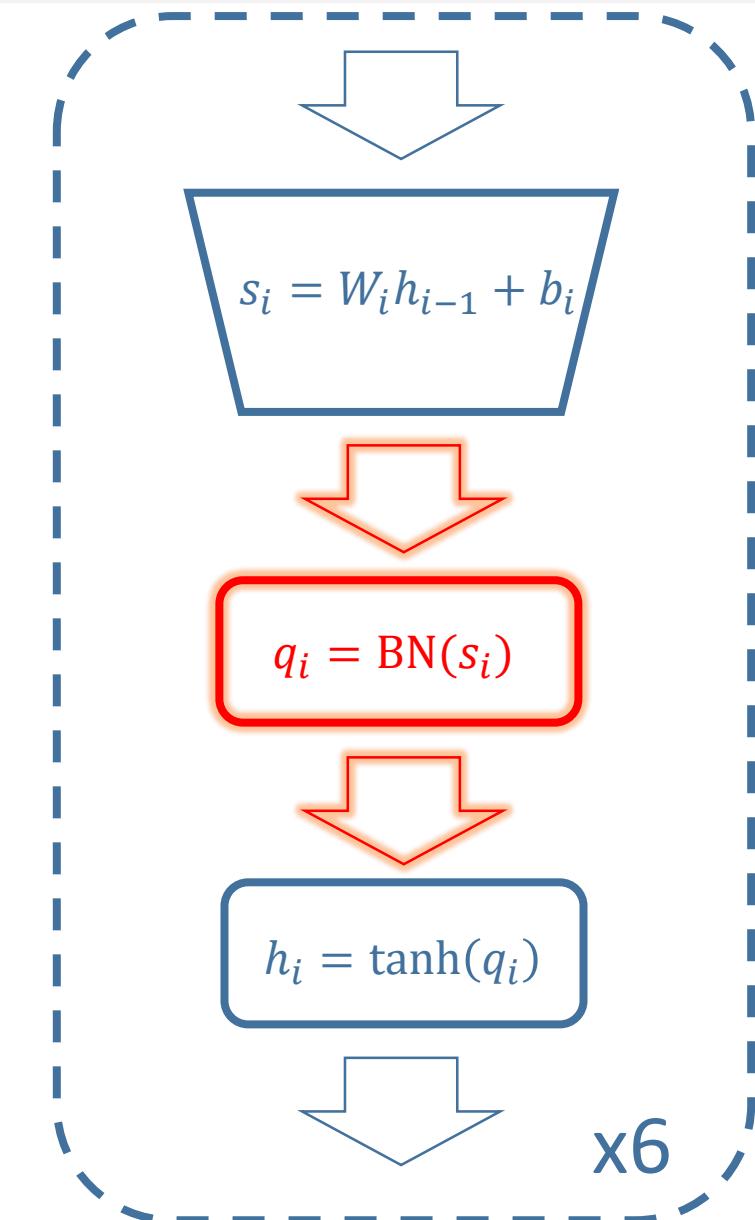
$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x]}}$$

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 normalization

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

Parameters to be learned: γ, β

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

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

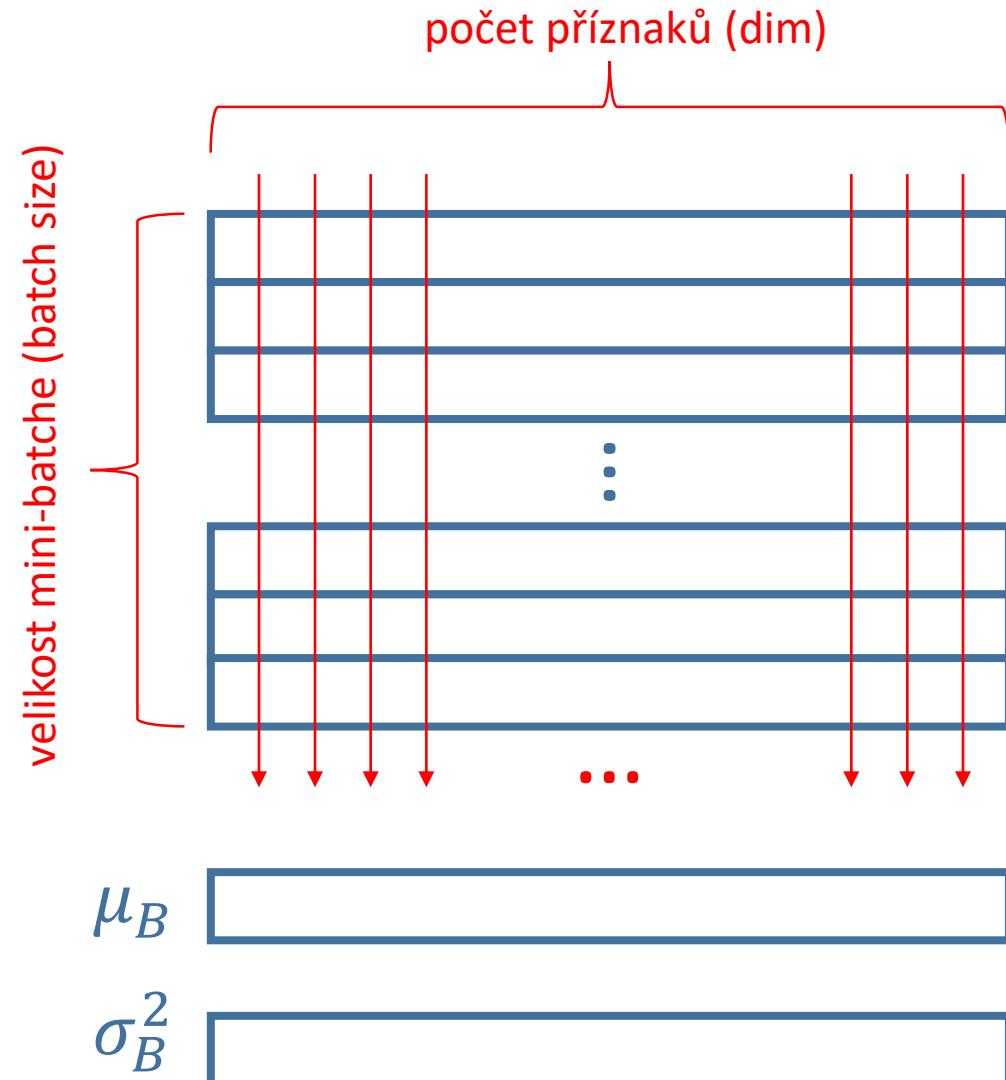
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

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

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

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 normalization 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
- 😢 zpomaluje
- 😢 take 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>

Další normalizace

Fully connected

- Batch normalizace
 - x $N \times D$
 - μ, σ $1 \times D$
- Layer normalizace
 - x $N \times D$
 - μ, σ $N \times 1$

Konvoluční vrstvy

- Spatial batchnorm a.k.a. BatchNorm2d
 - x $N \times C \times H \times W$
 - μ, σ $1 \times C \times 1 \times 1$
- Instance normalizace
 - x $N \times C \times H \times W$
 - μ, σ $N \times C \times 1 \times 1$

Další normalizace

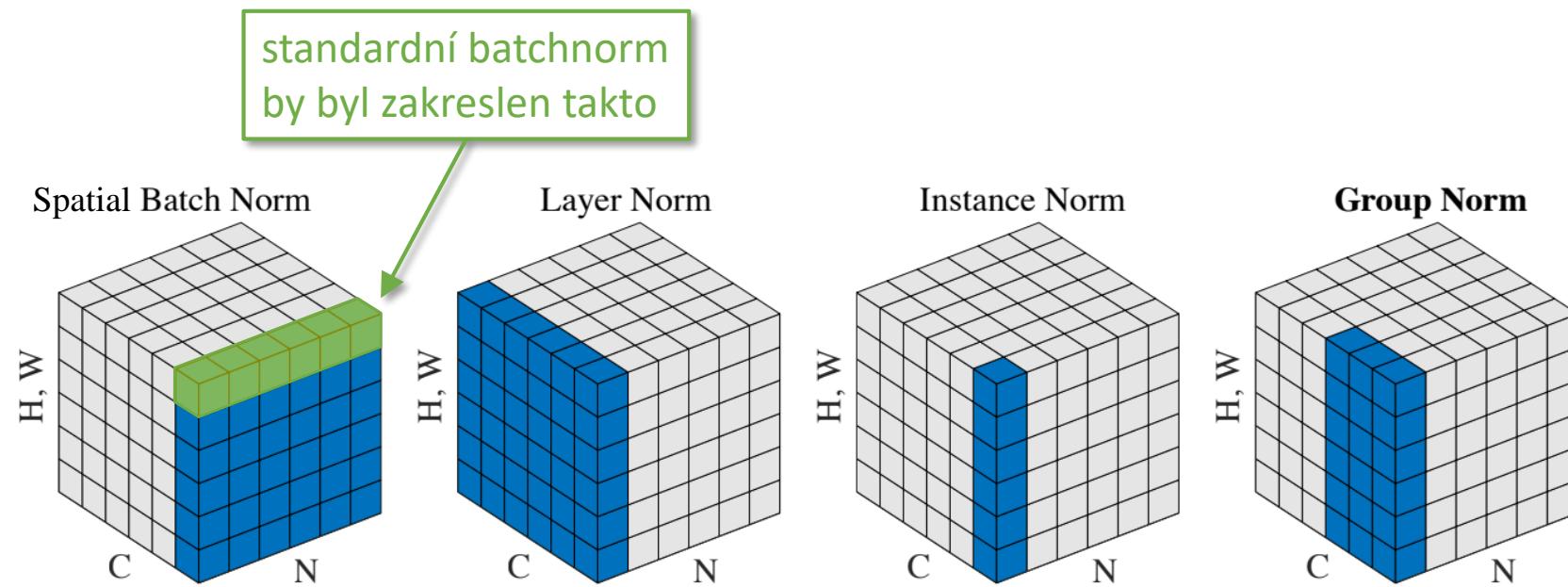


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

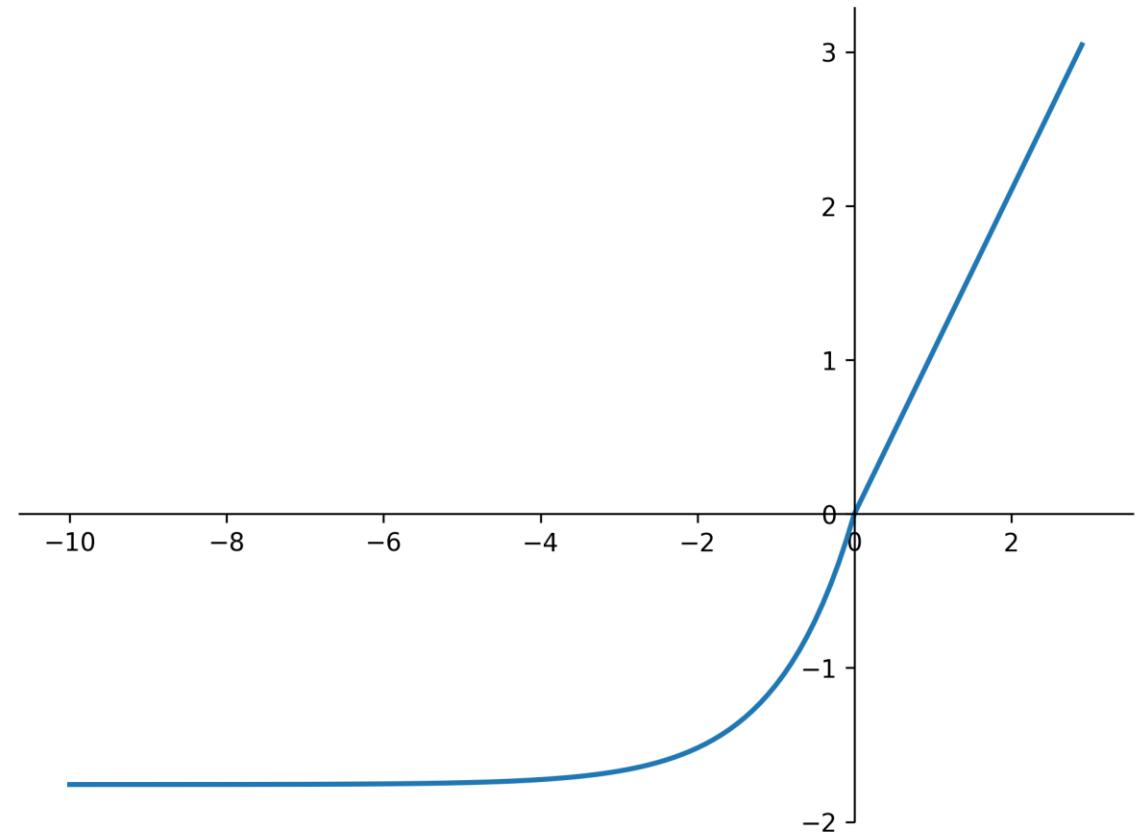
obrázek: [Wu, He: Group Normalization](#)

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)

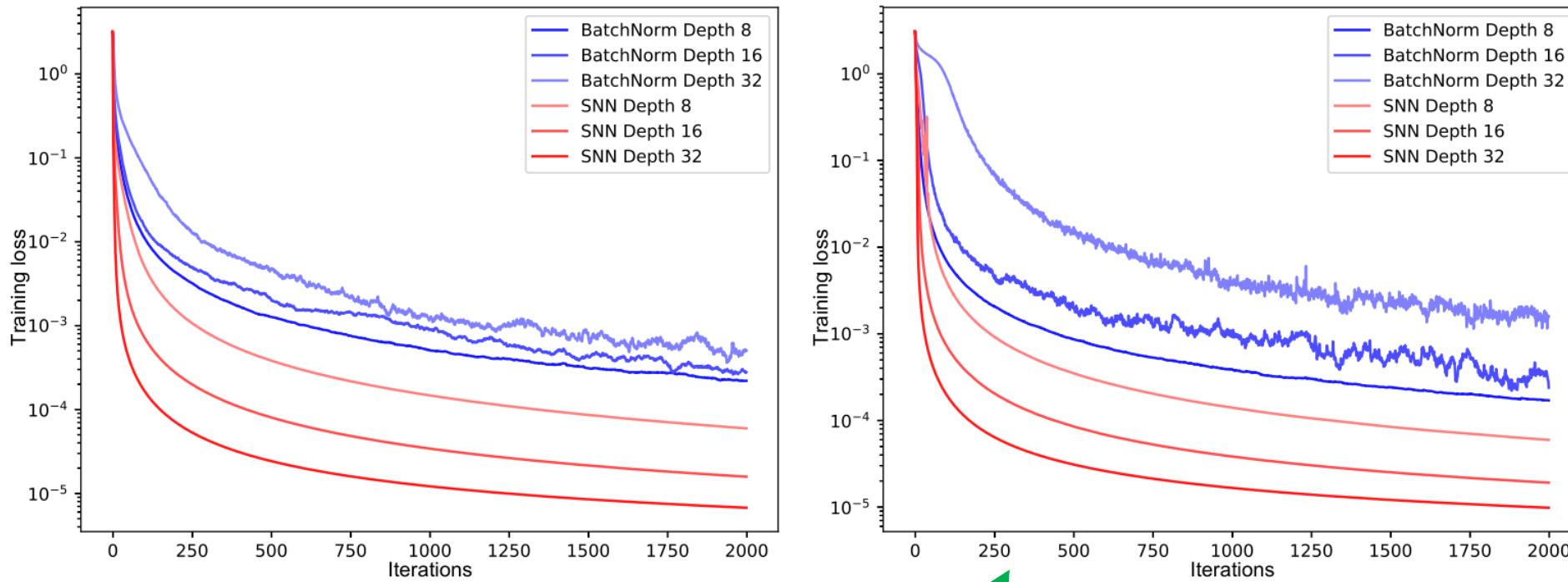
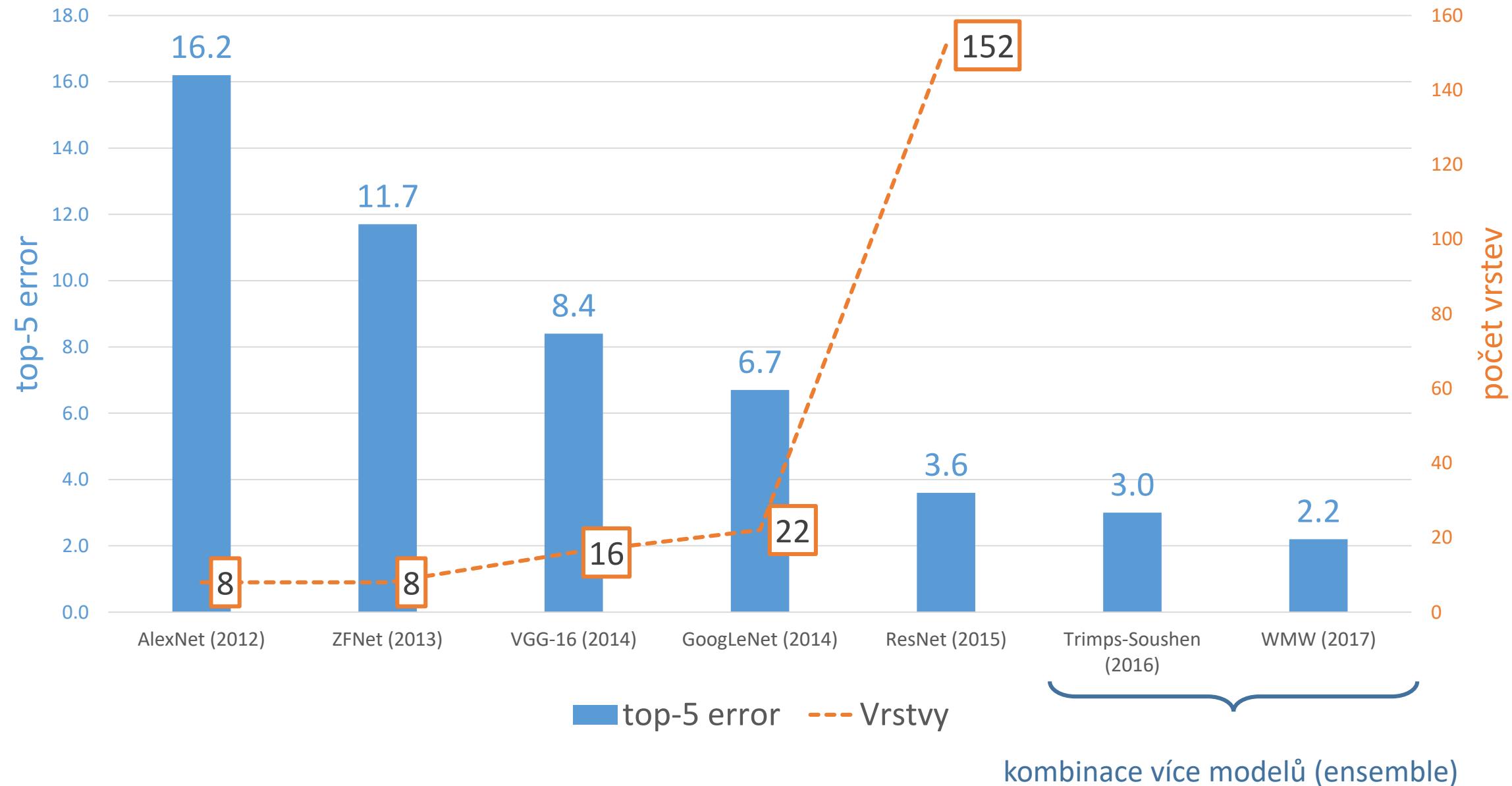


Figure 1: The left panel and the right panel show the training error (y-axis) for feed-forward neural networks (FNNs) with batch normalization (BatchNorm) and self-normalizing networks (SNN) across update steps (x-axis) on the **MNIST** dataset the **CIFAR10** dataset, respectively. We tested networks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

Další konvoluční sítě

ImageNet klasifikace



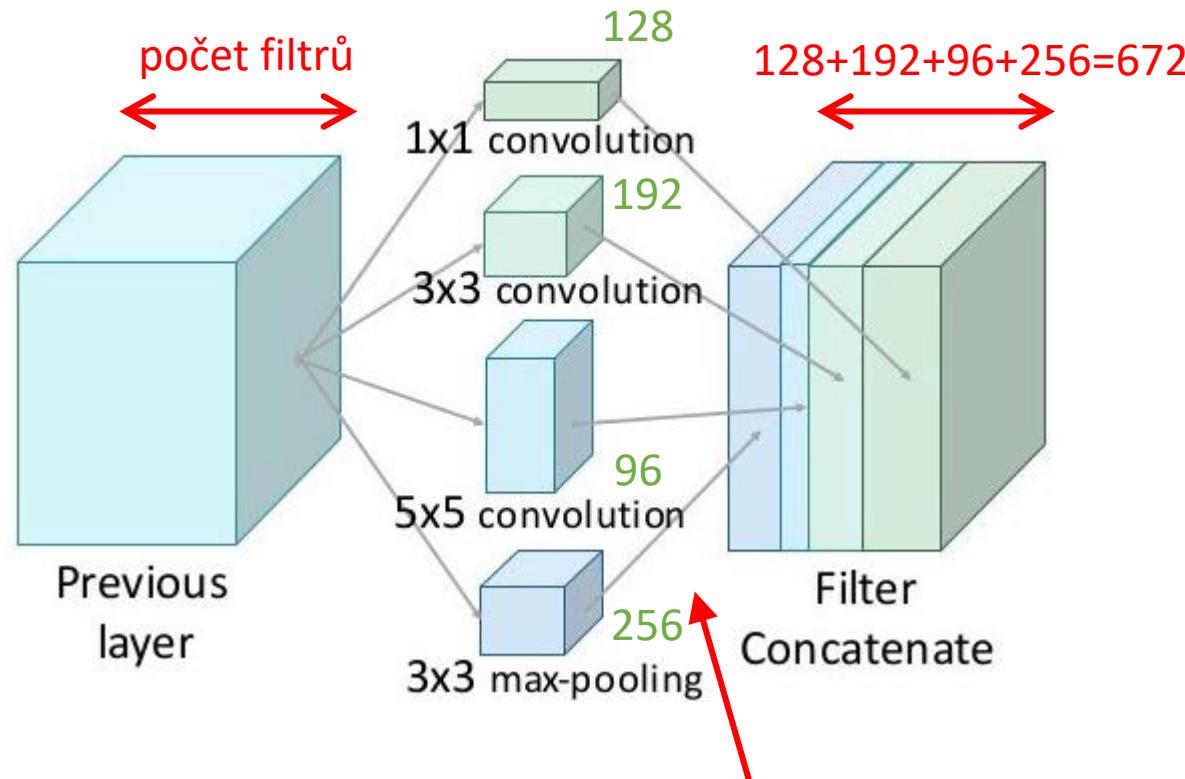
GoogLeNet (2014)

- Szegedy et al.: Going Deeper with Convolutions
- Navrženo s ohledem na výpočetní náročnost a celkový počet parametrů
- Skládá se z tzv. **Inception** modulů, které kombinují více typů konvolucí v jedné vrstvě (vychází z Lin et al.: “Network in network”)



obrázek: <http://knowyourmeme.com/memes/we-need-to-go-deeper>

Inception modul v1

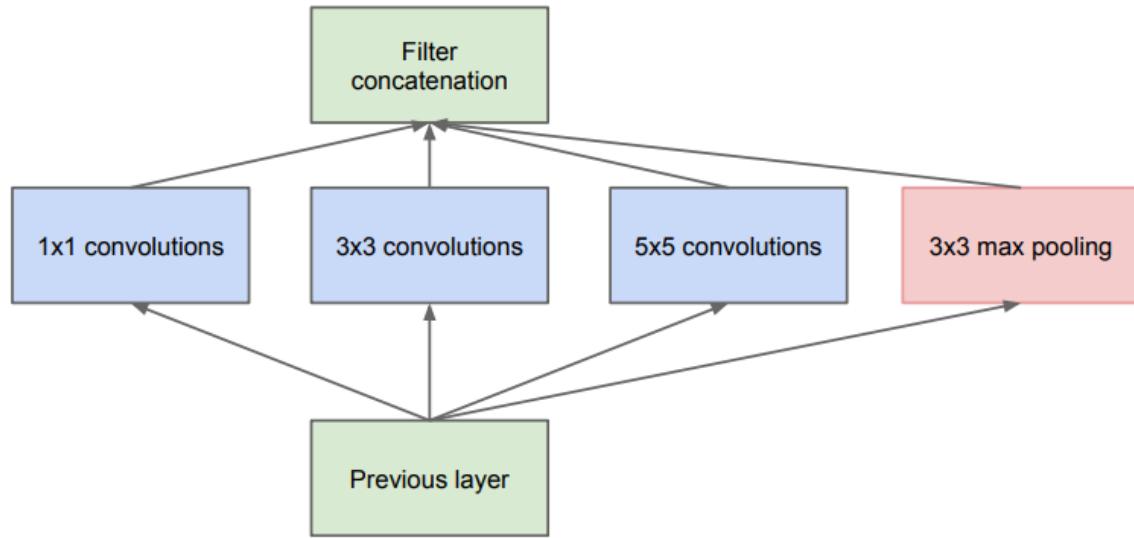


zero padding tak, aby výsledek konvoluce
měl vždy stejnou velikost (mode='same')

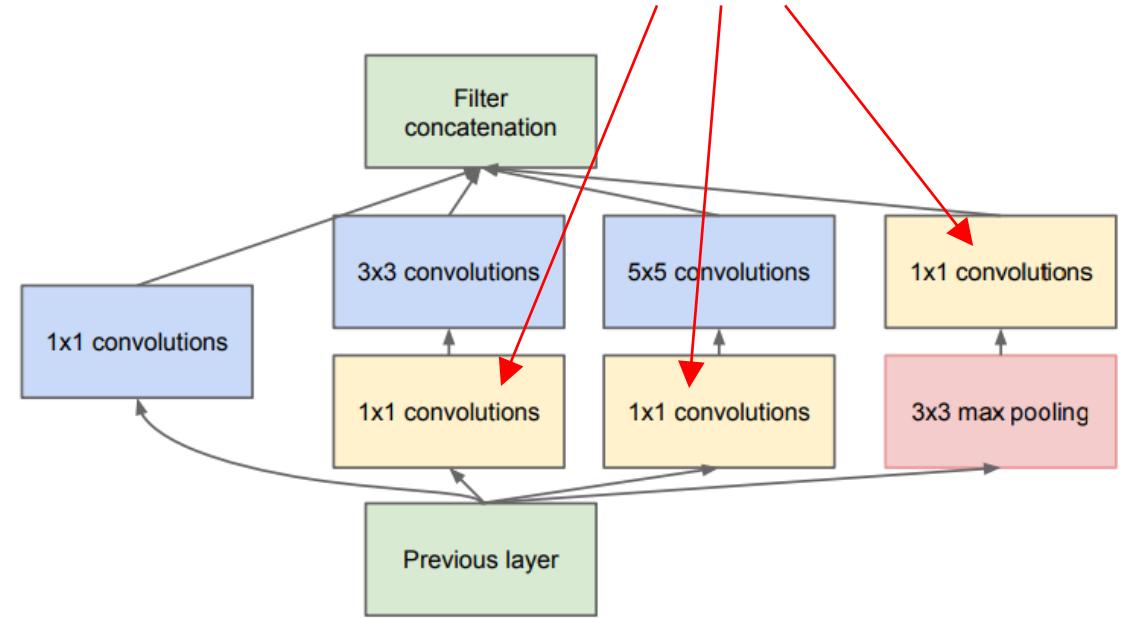
Optimalizace Inception modulu

[Szegedy et al.: Going Deeper with Convolutions](#)

1x1 “bottleneck” vrstvy s méně filtry: redukují dimenze a urychlují



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Figure 2: Inception module

obrázek: <https://arxiv.org/abs/1409.4842>

ve skutečnosti použita tato varianta

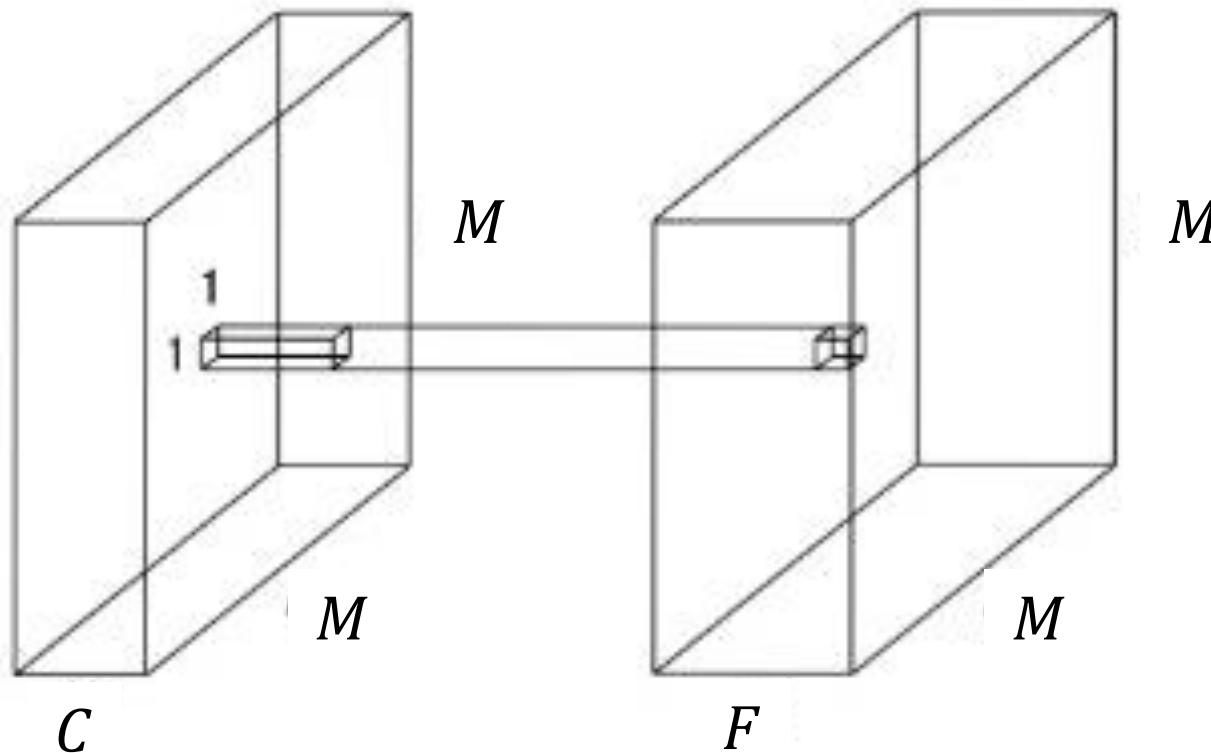


1×1 konvoluce

vzpomeňme: filtr vždy zahrnuje všechny kanály

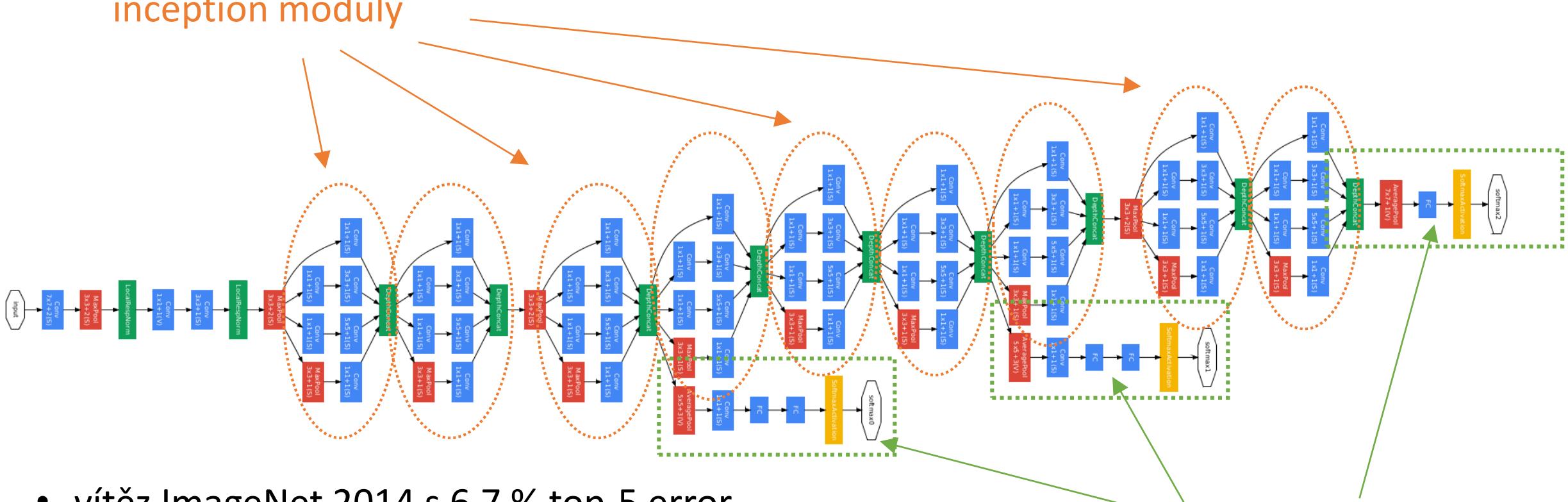
tzn., že i při 1×1 konvoluci (bez okolí) je výsledek stále lineární kombinací přes kanály

filtr je tedy $1 \times 1 \times C \rightarrow$ počet parametrů je $C \cdot F$



GoogLeNet (2014)

inception moduly



- vítěz ImageNet 2014 s 6.7 % top-5 error
- celkem 22 vrstev
- minimum fully-connected (lineárních) vrstev, téměr plně konvoluční síť
- 12x méně parametrů než AlexNet

loss na více místech sítě, ne
pouze na vrchu

ResNet (2015)

- He et al.: “Deep Residual Learning for Image Recognition”
- Cílem návrhu být co nejhlubší → 152 vrstev!
- Vítěz ImageNet 2015 ve všech kategoriích
- Vítěz MS COCO challenge
- 3.6 % top-5 error na ImageNet: lepší než člověk (cca 5 %)
- Problém: přidávání vrstev pomáhá jen do určité chvíle, pak už ne
- Overfitting?

Příliš mnoho vrstev

Overfitting? Ne: kromě testovací chyby s více vrstvami roste i trénovací chyba!

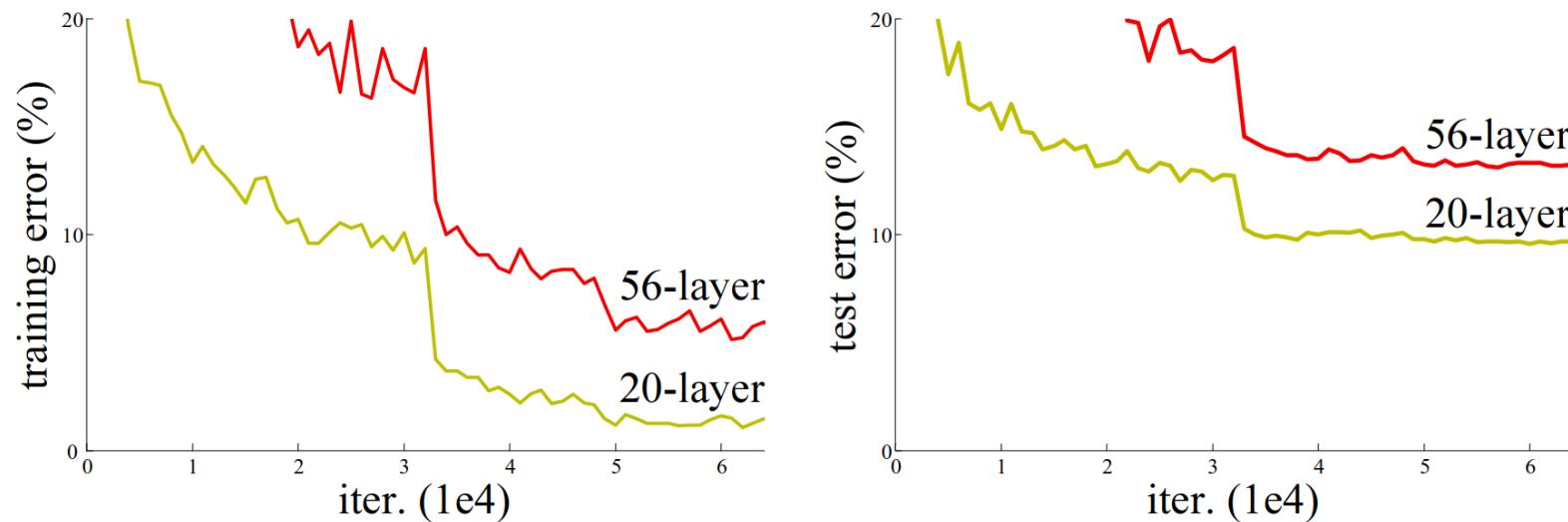


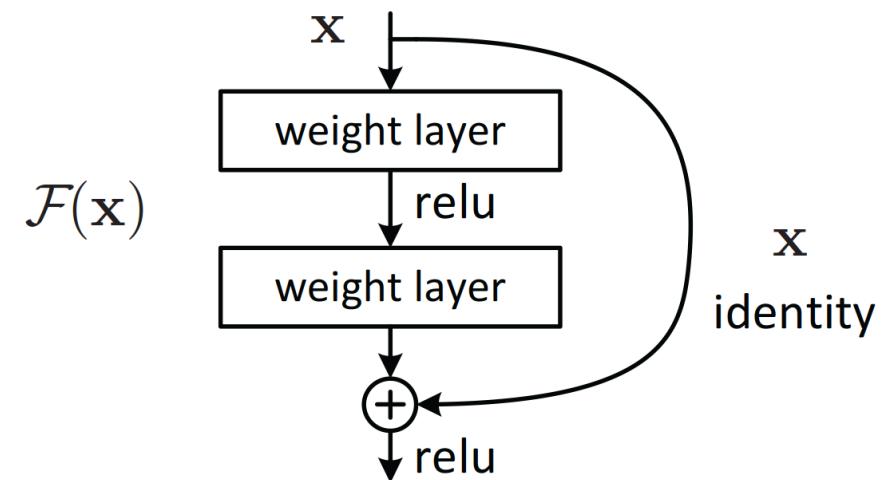
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Reziduální blok

- Podobně jako inception používá složitější bloky
- Výstup sestává ze součtu konvoluce a přímo mapovaného vstupu (identity)
- Sítí se tedy učí pouze rezidua

$$\mathcal{F}(x) = \mathcal{H}(x) - x$$

- “Naučit se nuly je jednodušší než identitu”



$$\mathcal{H}(x) = \mathcal{F}(x) + x$$

Figure 2. Residual learning: a building block.

Bottleneck residual blok

Podobně jako u Inception i ResNet
optimalizuje pomocí 1×1 bottleneck
vrstev uvnitř reziduálních bloků

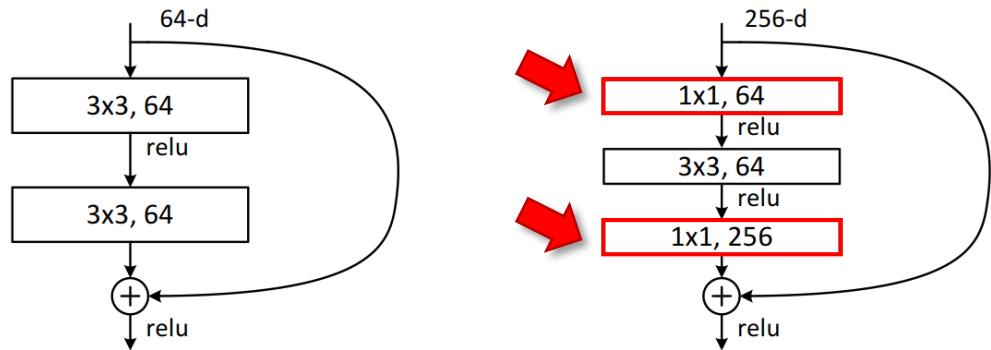
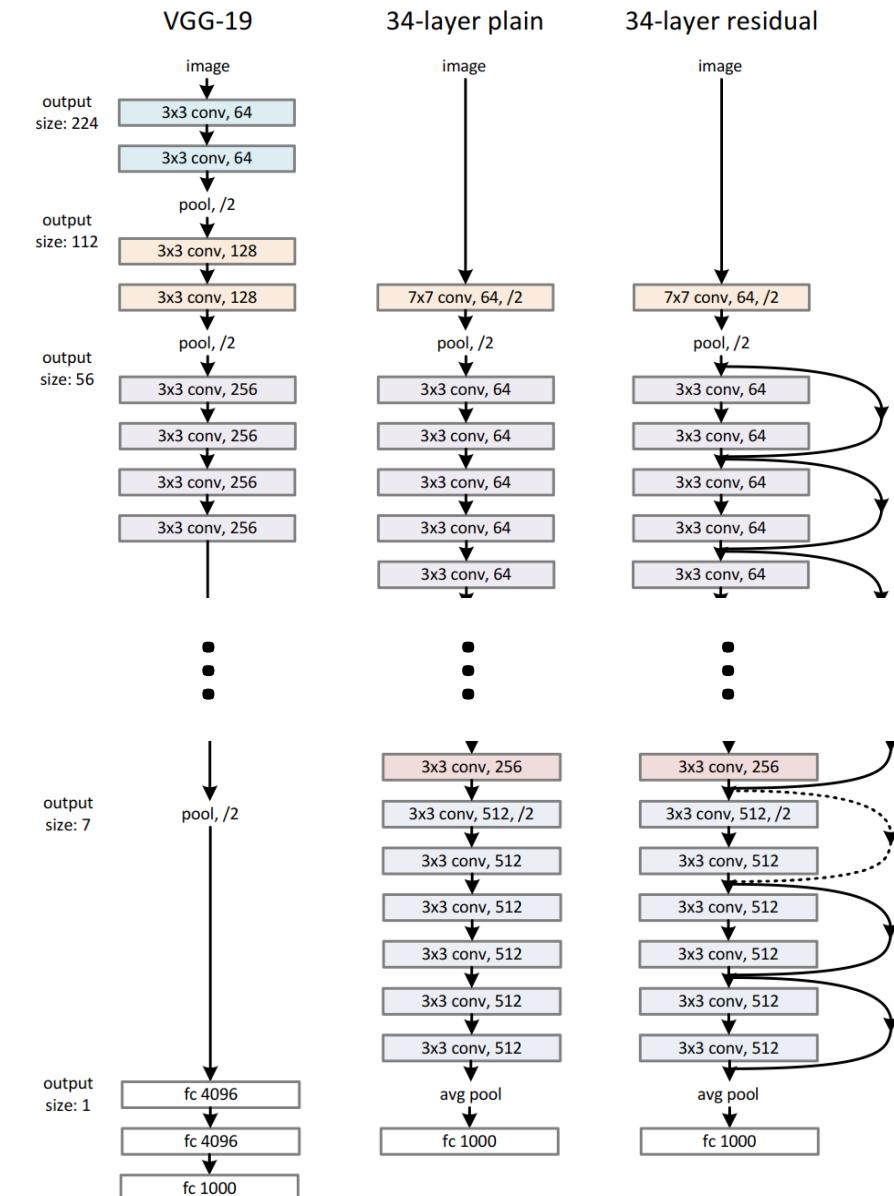


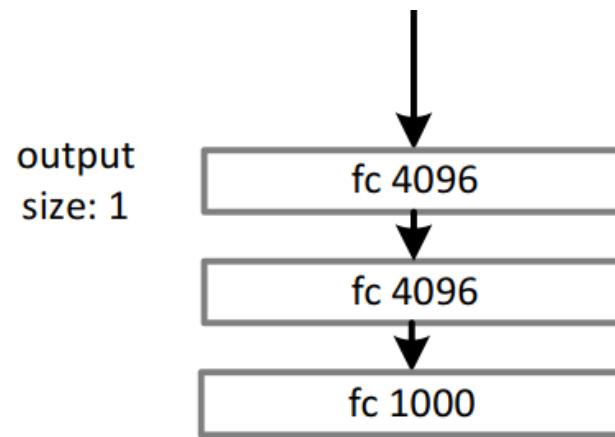
Figure 5. A deeper residual function \mathcal{F} for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.



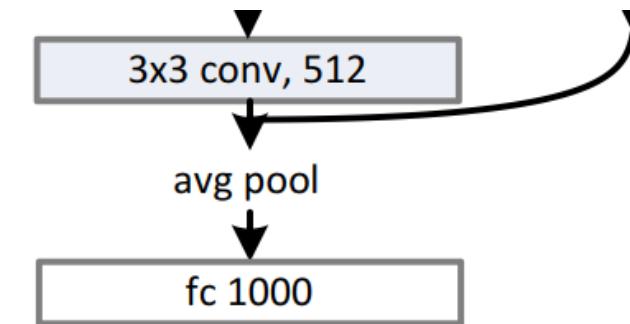
Téměr plně konvoluční

bez skrytých lineárních vrstev

VGG:



ResNet:



pouze lineární klasifikátor

Average pooling

1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

max pooling

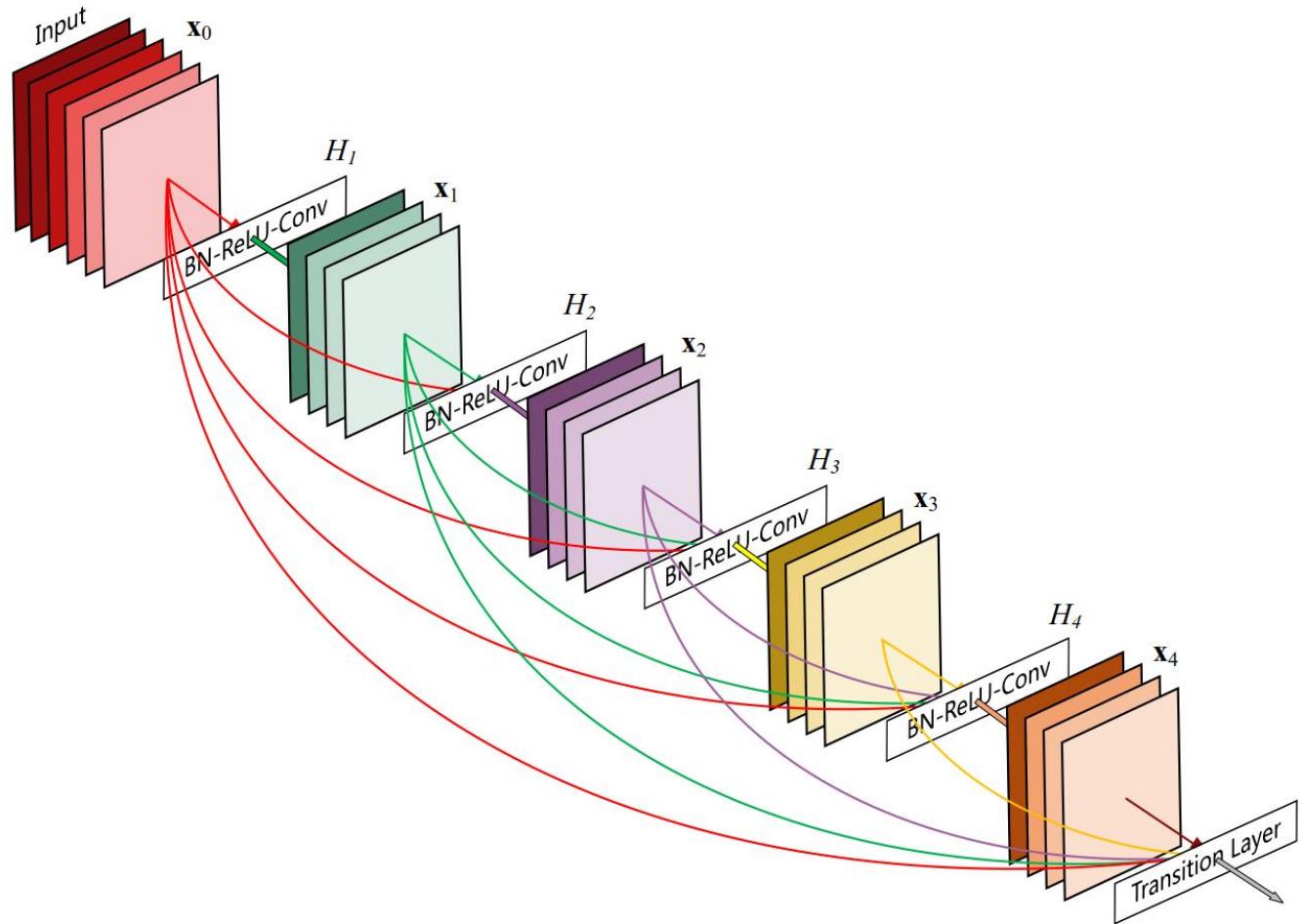
6	8
3	4

average pooling

3	5
2	2

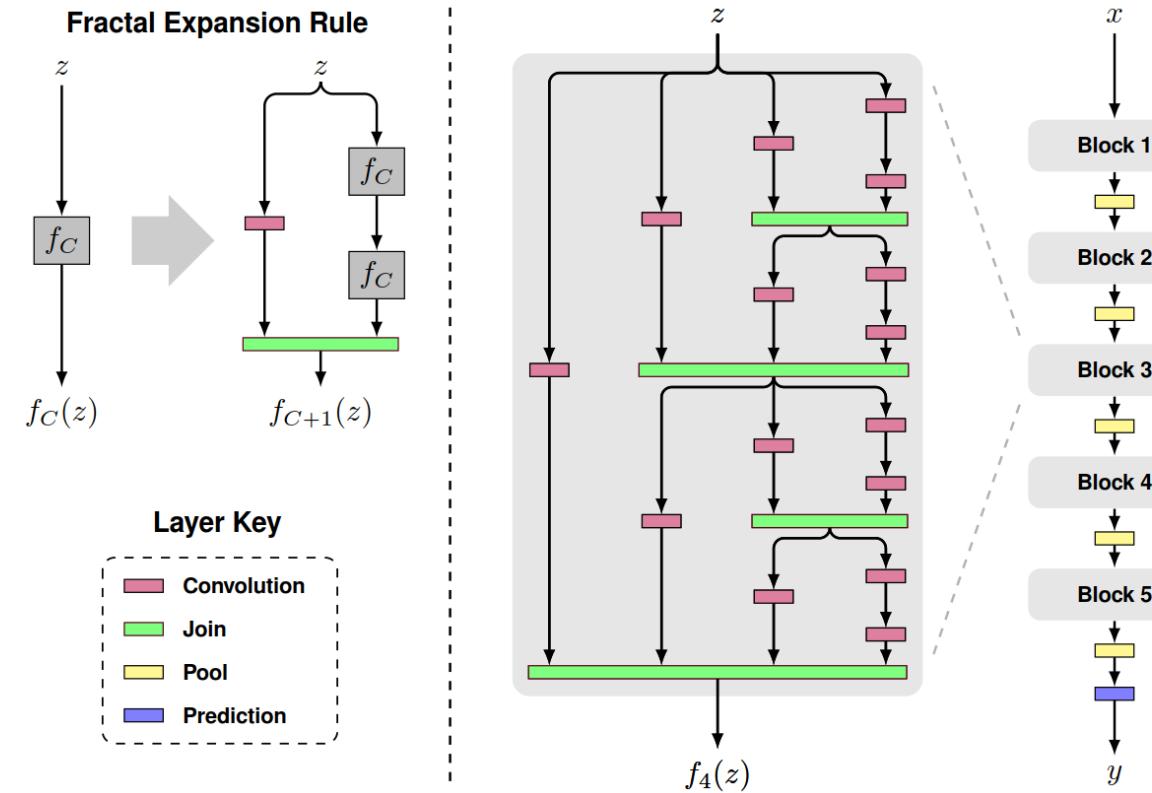
DenseNet (2016)

- Huang et al.: “Densely Connected Convolutional Networks”
- Výstup vrstvy je připojen na vstup každé další vrstvy
- “ResNet do extrému”



FractalNet (2017)

- Larsson et al.: “FractalNet: Ultra-Deep Neural Networks without Residuals”
- “Rekurzivní ResNet”



SqueezeNet (2016)

- Iandola et al: “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”
- Cílem co nejmenší a nejfektivnější síť

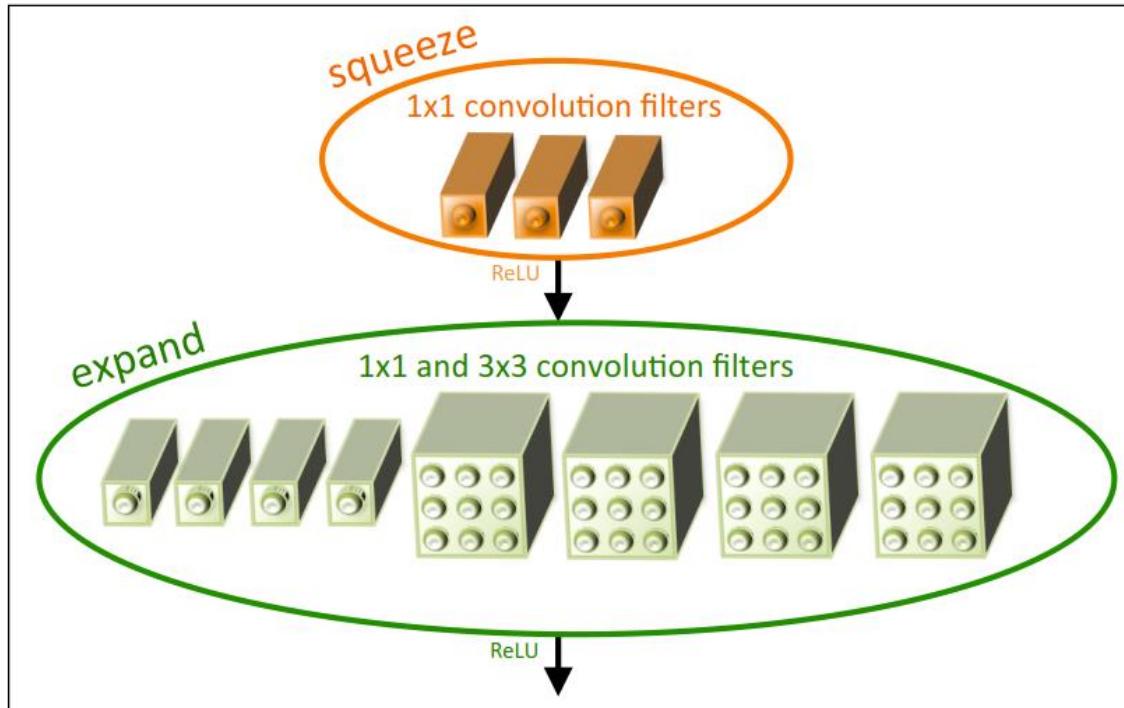
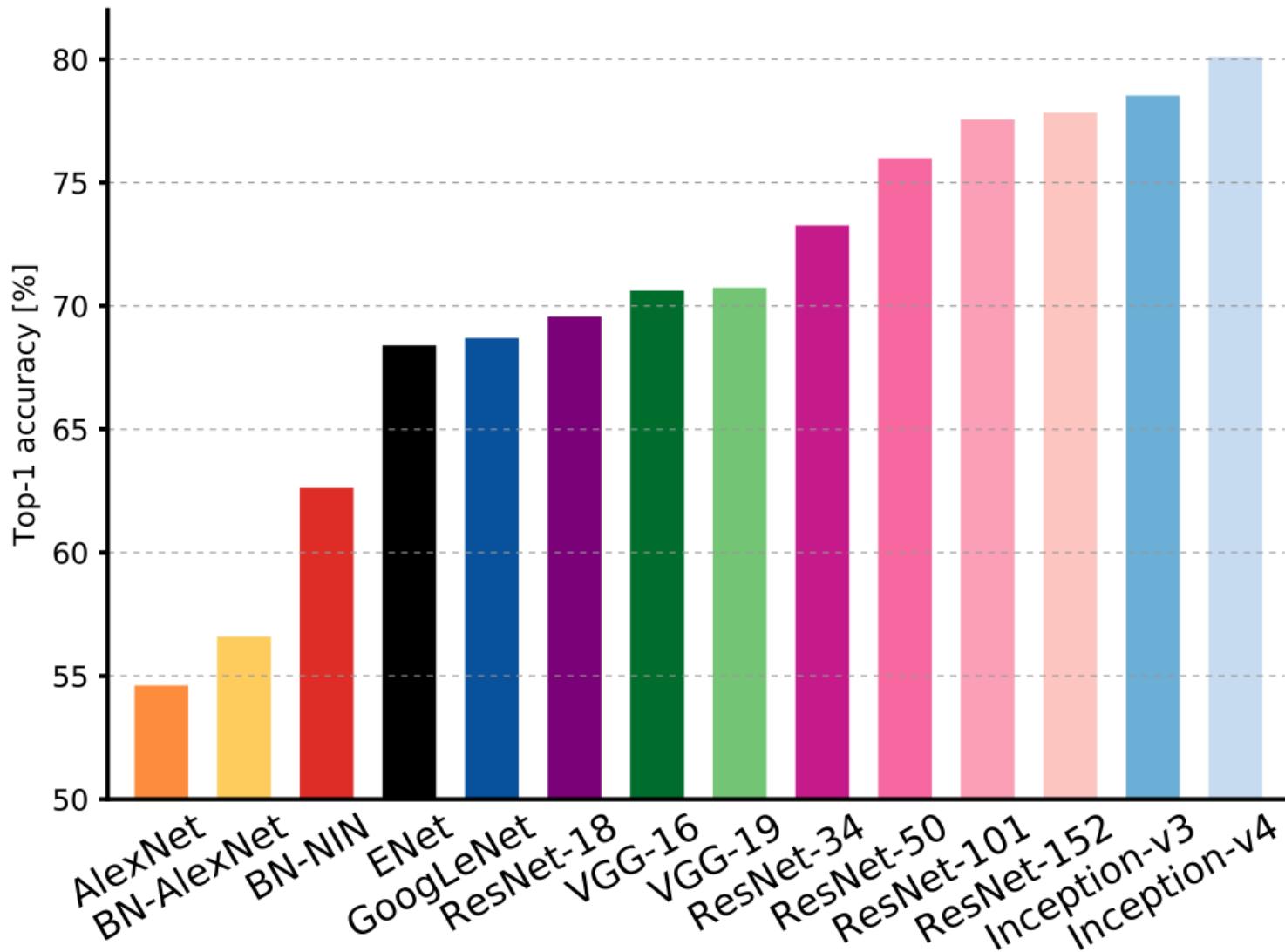


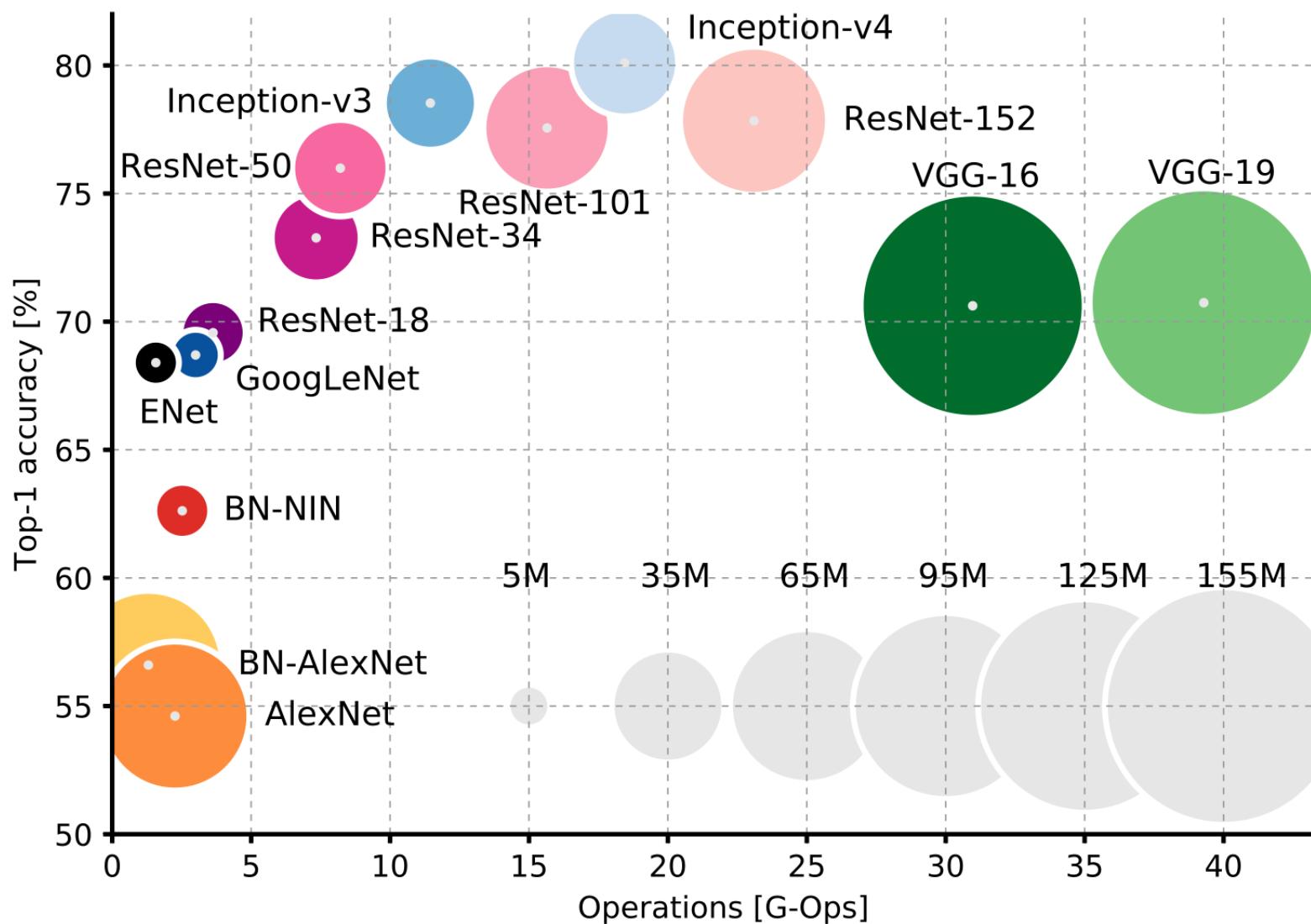
Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

Srovnání nejpoužívanějších CNN architektur



obrázek: [Canziani et al.: "An Analysis of Deep Neural Network Models for Practical Applications"](#)

Srovnání nejpoužívanějších CNN architektur



velikost znázorňuje
celkový počet
parametrů

obrázek: [Canziani et al.: "An Analysis of Deep Neural Network Models for Practical Applications"](#)

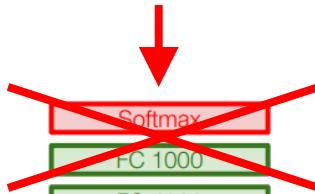
Transfer learning

Trénování konvolučních sítí při málo datech

- Popsané architektury mají obvykle miliony parametrů
- Malé datasety na jejich trénování nestačí → výrazný overfit
- I pokud data máme: trénování VGG na ImageNet trvalo autorům 2-3 týdny, a to i s 4x NVIDIA Titan Black GPU
- **Naštěstí lze obejít!**
 1. Můžeme vzít existující již natrénovaný model (např. VGG-16)
 2. Odstraníme poslední klasifikační vrstvu
 3. Nahradíme vlastní

Transfer learning

1000 tříd pro ImageNet

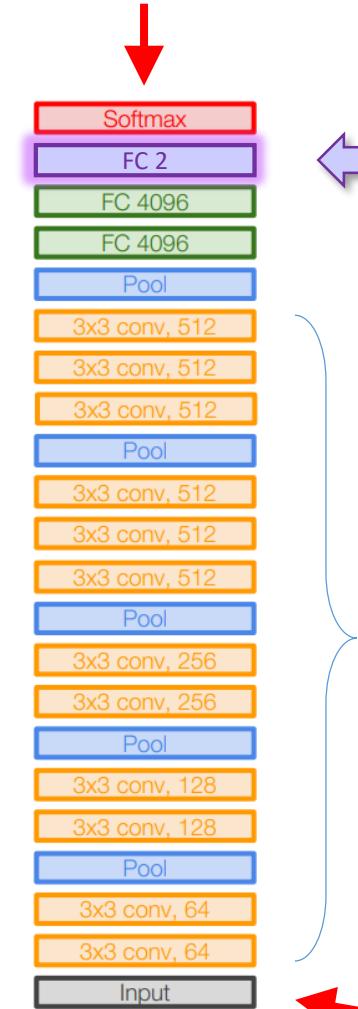


poslední lineární vrstvu
tvaru 4096×1000
zahodíme

ImageNet data

VGG16

2 třídy: kočky vs psi



připojíme vlastní lineární vrstvu
tvaru 4096×2 a náhodně
Inicializujeme

pokud máme málo dat, je lepší
konvoluční vrstvy netrénovat →
layer freeze

můžeme použít vlastní data

Transer learning v PyTorch

```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False
# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)
model_conv = model_conv.to(device)
criterion = nn.CrossEntropyLoss()
# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)
```

“zmrazení” vrstev, nebudou se trénovat a zůstávají konst. → síť pouze jako extractor příznaků

poslední vrstvu klasifikující do 1000 ImageNet tříd nahradíme vlastní, která má pouze 2 třídy

jako seznam parametrů pro optimalizaci předáváme pouze poslední lineární vrstvu (pouze pro urychlení)

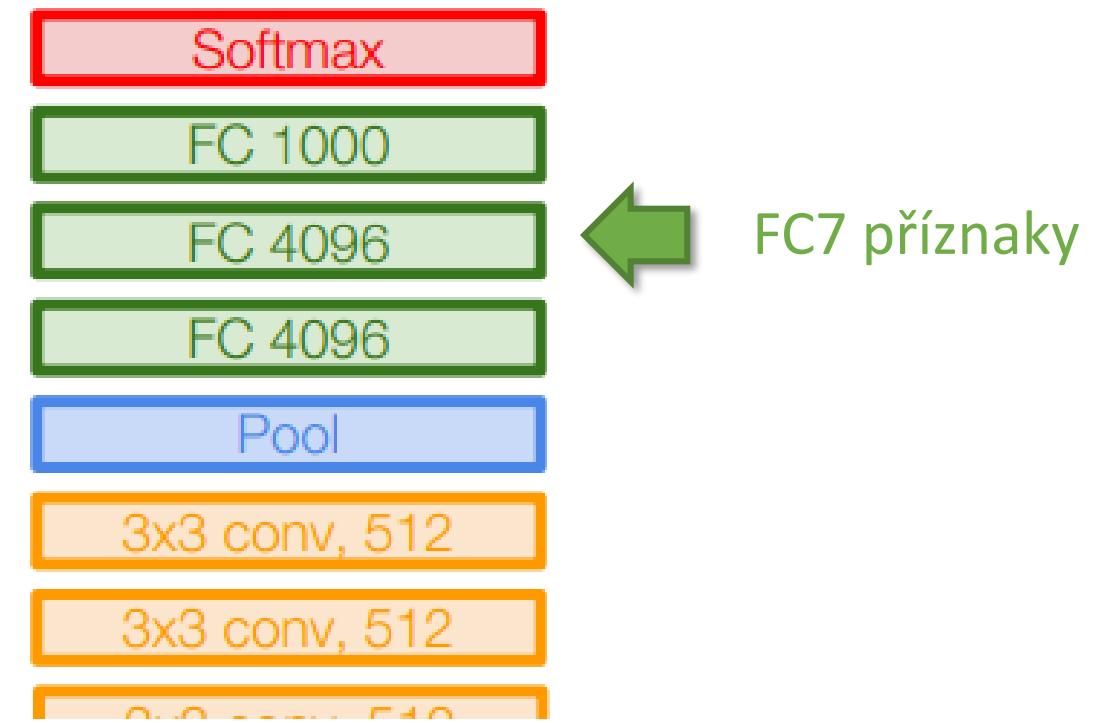
- poté, co je poslední vrstva natrénovaná, je možné opět uvolnit (“rozmrazit”) i konvoluční vrstvy a model dále zlepšit (fine tuning)

https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

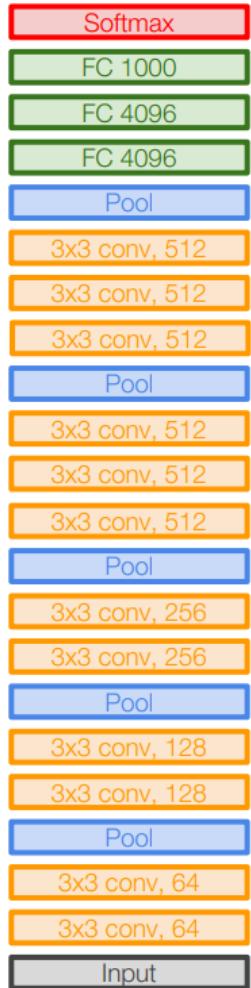
CNN příznaky

- Výstup z posledních lineárních vrstev lze použít např. jako příznaky (tzv. FC7) → CNN jako “feature extractor”
- Např. VGG-16 předposlední vrstva má rozměr 4096
- Nad těmito příznaky je možné natrénovat libovolný klasifikátor, třeba i rozhodovací stromy/lesy, bayesovské klasifikátory, ...
- Lze také využít pro urychlení trénování: celý dataset projet sítí a pro každý obrázek uložit na disk FC7 příznaky
- Během trénování se pak nemusí znova a znova provádět dopředný průchod celou sítí, pouze těmi posledními

např. VGG-16:



Transfer learning: shrnutí



specifické příznaky
(obličeje, text, ...)

obecné příznaky
(hrany, bloby, ...)

	podobná data	odlišná data
málo dat	trénovat spíše jen poslední vrstvu	problém 😊
hodně dat	fine tune několika vrstev (lze ale i celou síť)	fine tune více vrstev nebo i celé sítě

Shrnutí

Návrh vlastní sítě

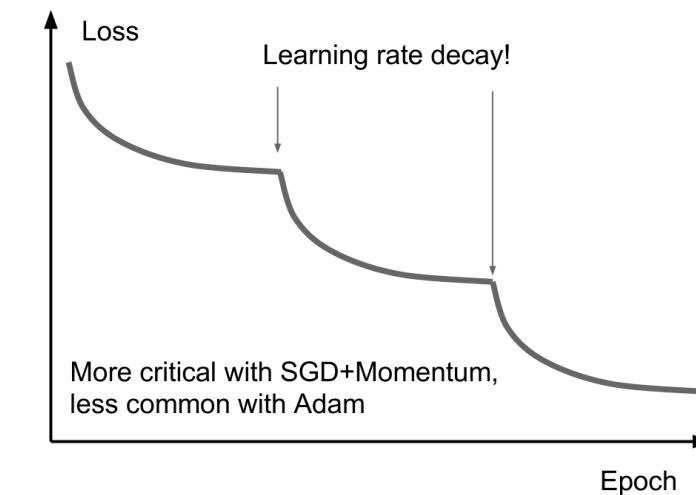
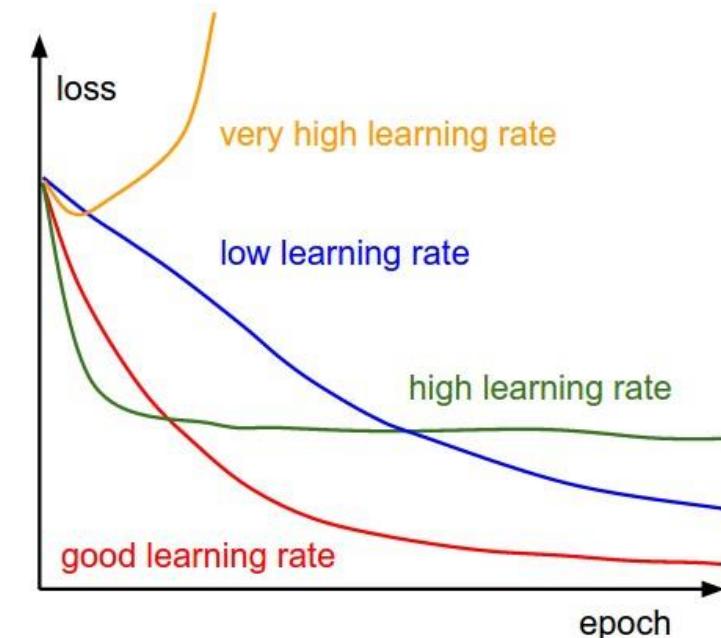
- méně parametrů, více nelinearit
- využívat sdílení parametrů → např. více konvoluce, méně lineárních vrstev
- použít spíše ReLU-like nonlinearity, sigmoid ne
- použít batchnorm, pravděpodobně pomůže
- vršek sítě dle úlohy:
 - klasifikace = sigmoid / softmax
 - regrese = bez nonlinearity
- pokud navrhujeme vlastní vrstvy a nelze využít autograd → gradient check

Aplikace existující sítě (transfer learning)

- raději implementace na githubu než se snažit o vlastní – často velmi obtížné replikovat pouze z článku
- stáhnout předučené váhy
- u menších sítí jako AlexNet lze natrénovat “from scratch”, u větších velmi obtížné
- pokud máme málo dat, zablokovat trénování u nižších vrstev

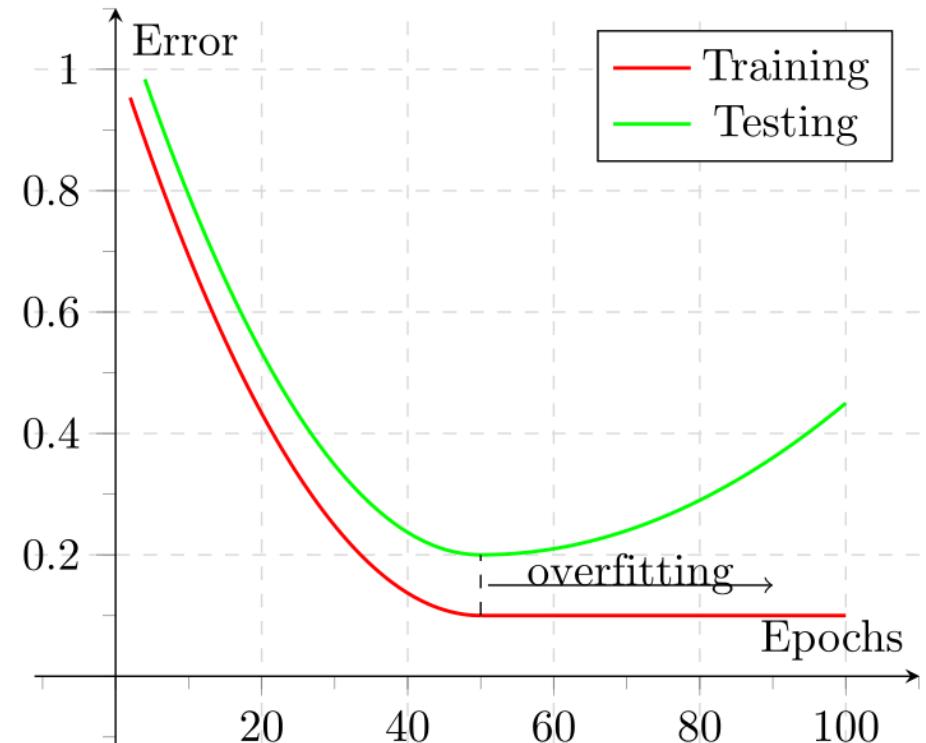
Trénování

- Ověřit overfittingem: zkusit malý vzorek, na němž model musí dosáhnout 100%
- Monitorovat hodnotu lossu a podle toho nastavit learning rate
- Poté případně podle průběhu postupně learning rate snižovat
 - existují i alternující schémata, viz např. [Smith: "Cyclical Learning Rates for Training Neural Networks"](#)
- Pokud vše funguje, zkusit optimalizovat hyperparametry



Prevence overfitu & optimalizace skóre

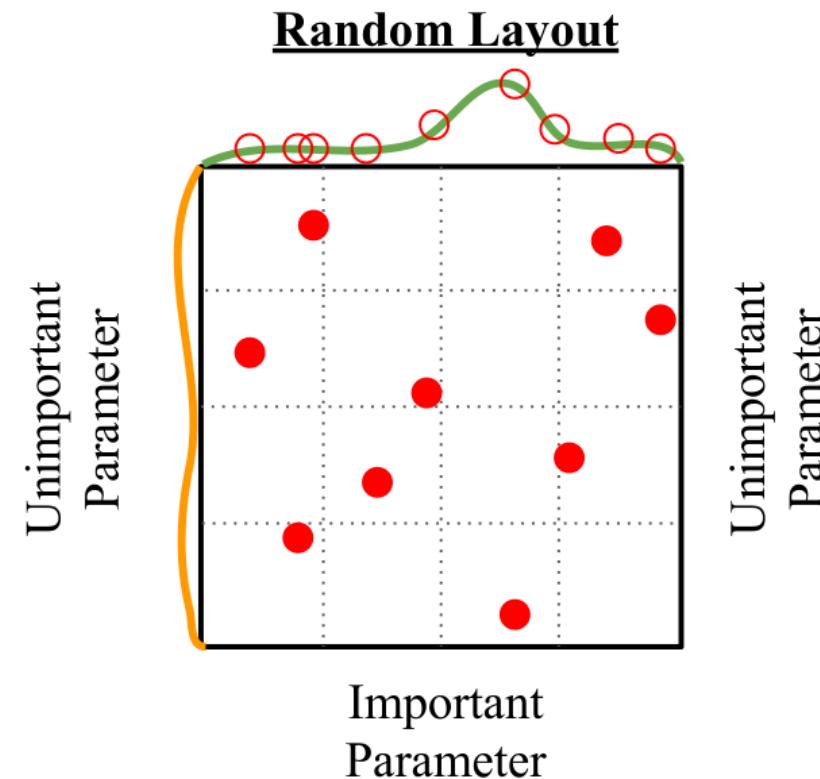
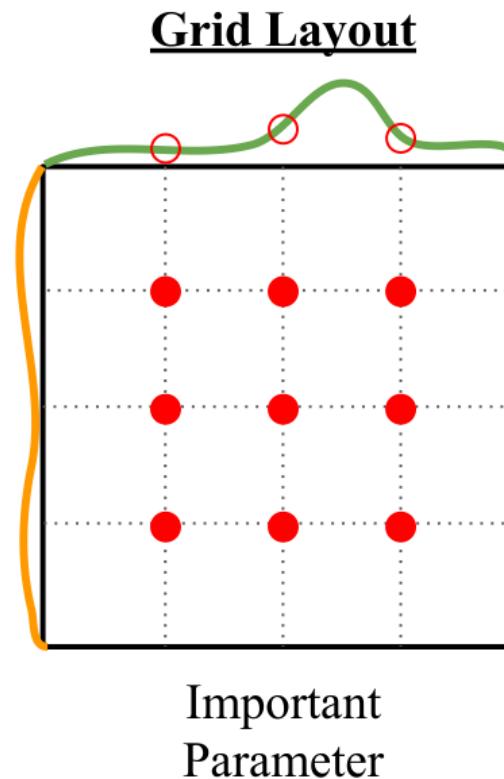
1. Nasbírat více dat
2. Uměle rozšířit data
3. Aplikovat vhodnější architekturu sítě
4. Pokud overfit: regularizace, dropout
5. Pokud stále overfit: zmenšit síť



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

Optimalizace hyperparametrů

Co když výsledné skóre závisí více na jednom parametru než na jiném?



Náhodné zkoušení lépe pokrývá prostor možností než předdefinované kombinace

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