

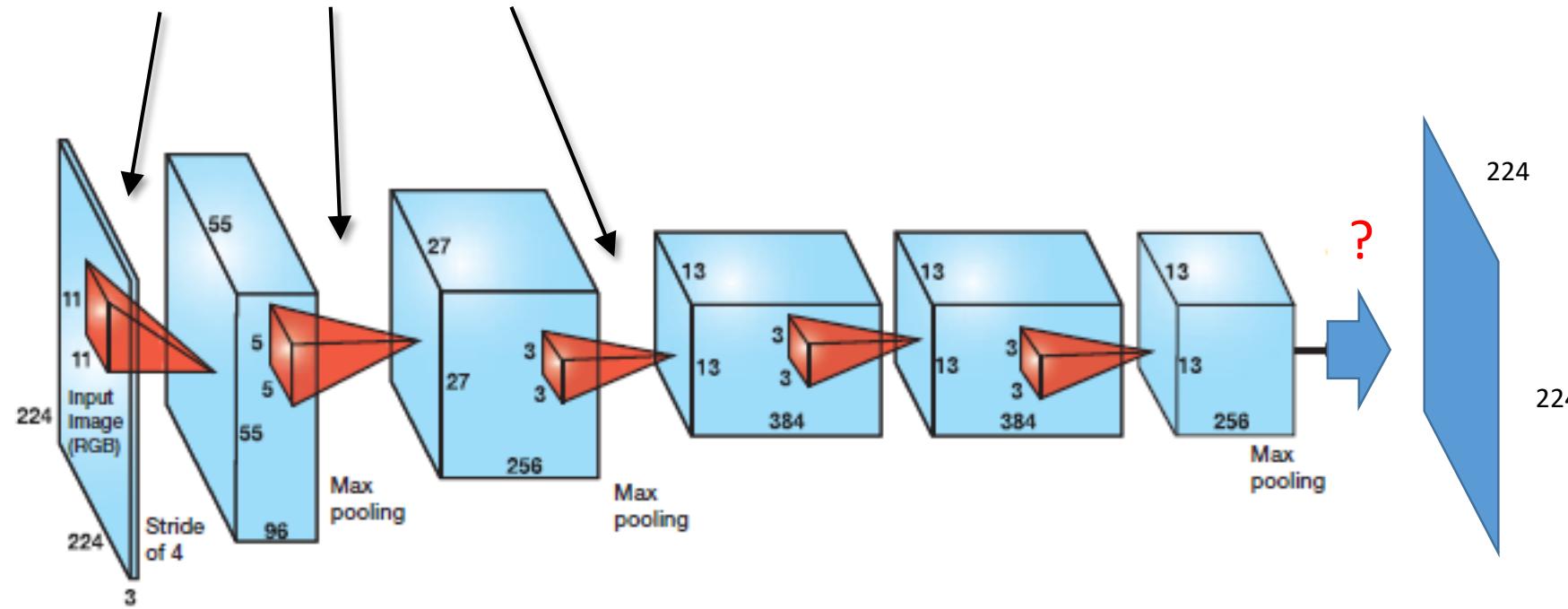
# Aplikace neuronových sítí

---

Transponovaná konvoluce, generativní modely

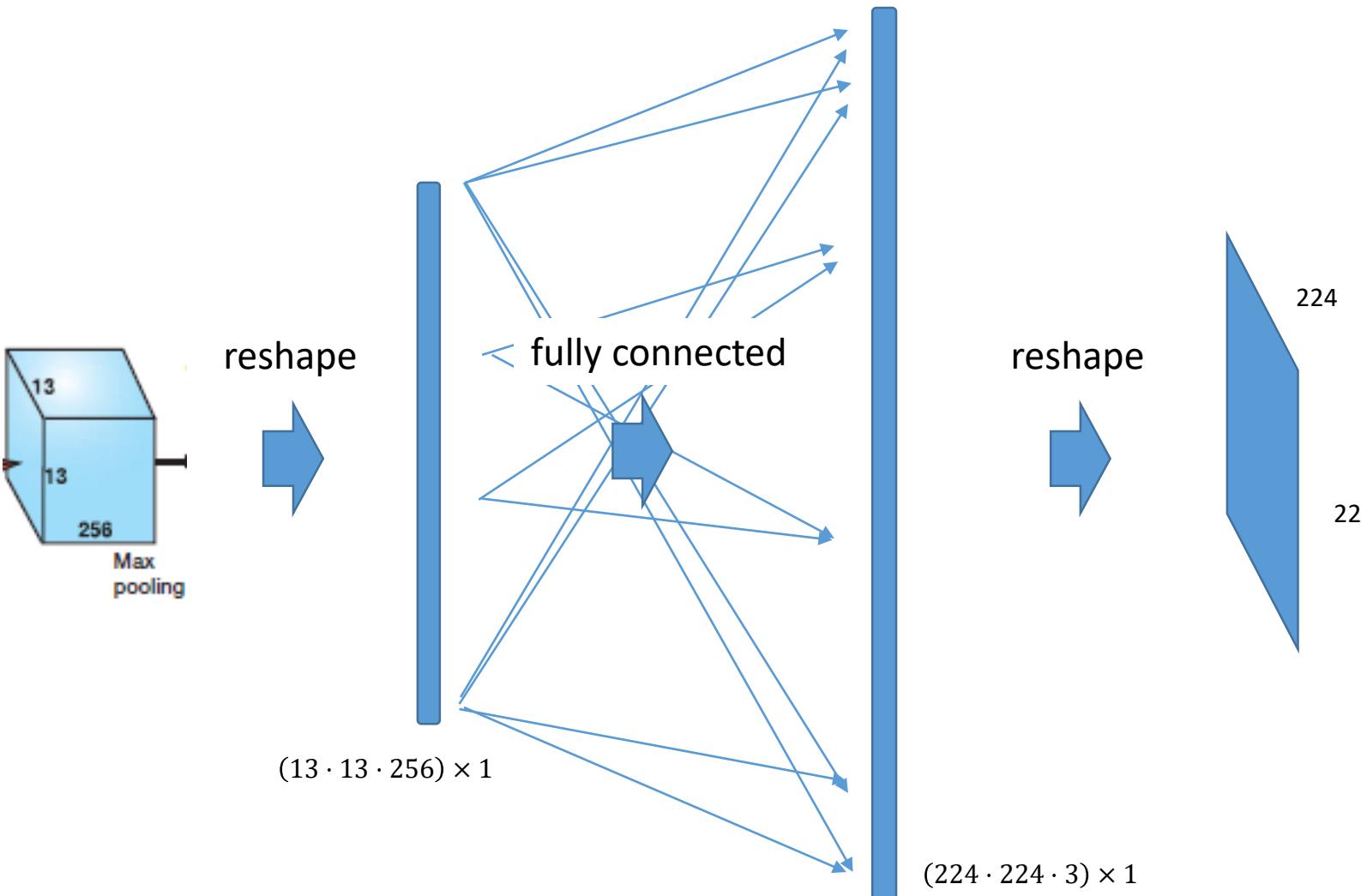
# Obrázek jako výstup

2D max pooling či konvoluce s krokem (stride) > 1 pouze zmenšují



jak vyrobit/rekonstruovat obrázek v původním  
nebo vyšším rozlišení, než je poslední vrstva?

# Triviálně fully connected vrstvou

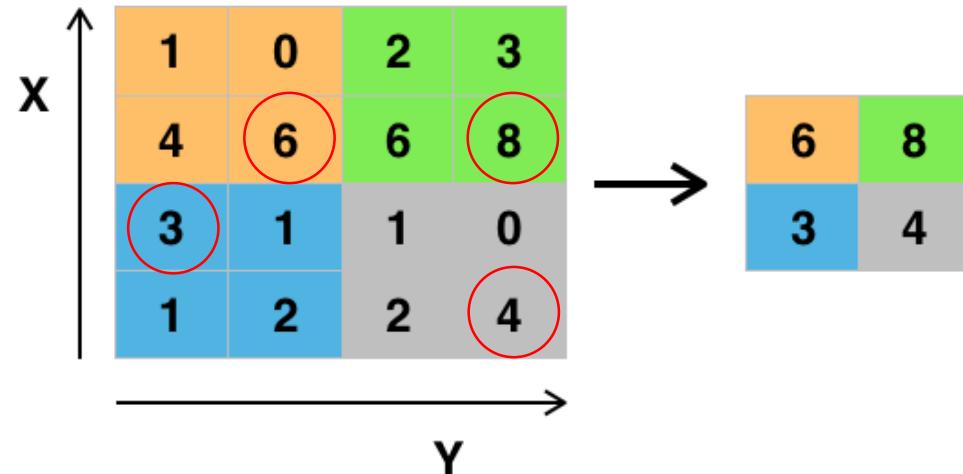


problém: počet parametrů =  $13 \cdot 13 \cdot 256 \cdot 224 \cdot 224 \cdot 3 = 6.5 \cdot 10^9$  (!!)

# Max Unpooling

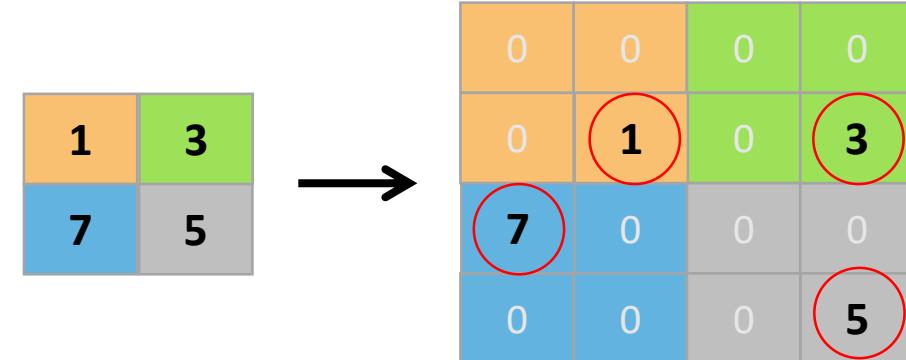
## Max pooling

Single depth slice



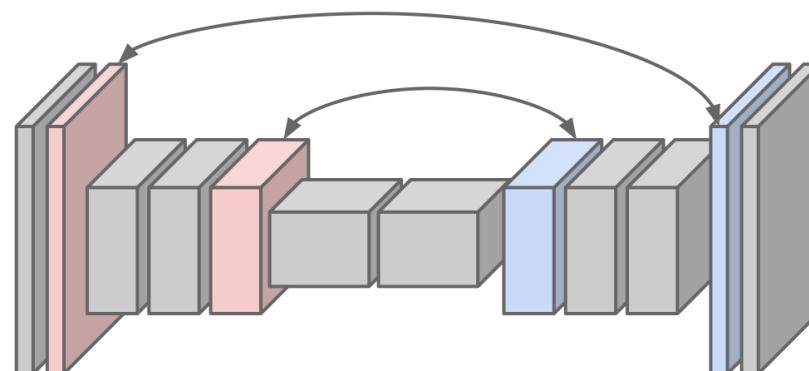
zapamatují se pozice maxim

## Max unpooling



předpoklad symetrické konvoluční sítě se stejným počtem poolingů i unpoolingu

hodnoty se zapíšou na pozice dle pozice maxim z odpovídajícího poolingu v první části sítě



všimněme si: připomíná  
zpětný průchod max poolingu

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

1	2
3	4

vstup: 2x2

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

výstup: 4x4

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

1	2
3	4

vstup: 2x2

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

výstup: 4x4

váhy konvolučního filtru

1	-1	2
1	-2	1
-1	2	-1

x 1



1	-1	2
1	-2	1
-1	2	-1

toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

1	2
3	4

vstup: 2x2

-2	1	0	0
2	-1	0	0
0	0	0	0
0	0	0	0

výstup: 4x4

1	-1	2
1	-2	1
-1	2	-1

x 1



1	-1	2
1	-2	1
-1	2	-1

toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

1	2
3	4

vstup: 2x2

-2	1	0	0
2	-1	0	0
0	0	0	0
0	0	0	0

výstup: 4x4

váhy konvolučního filtru

1	-1	2
1	-2	1
-1	2	-1

x 2



2	-2	4
2	-4	2
-2	4	-2

toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

1	2
3	4

vstup: 2x2

-2	3	-4	2
2	-3	4	-2
0	0	0	0
0	0	0	0

výstup: 4x4

váhy konvolučního filtru

1	-1	2
1	-2	1
-1	2	-1

x 2



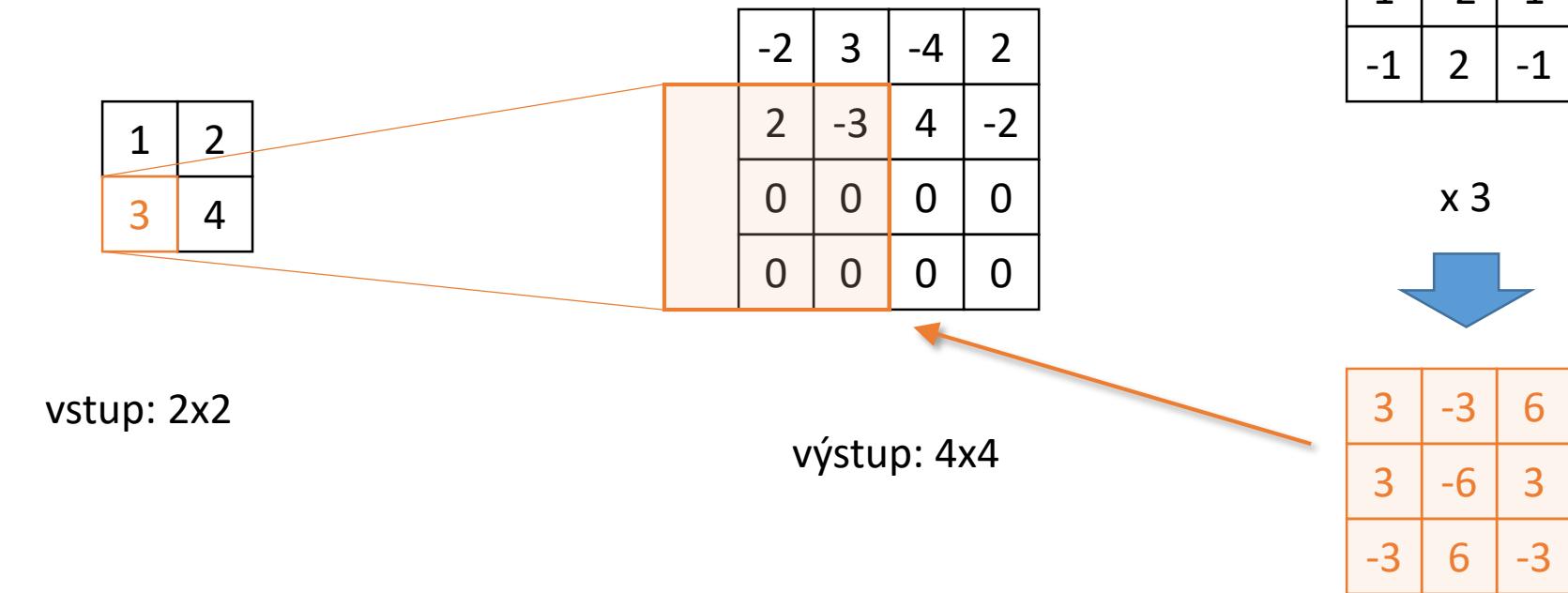
2	-2	4
2	-4	2
-2	4	-2

toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

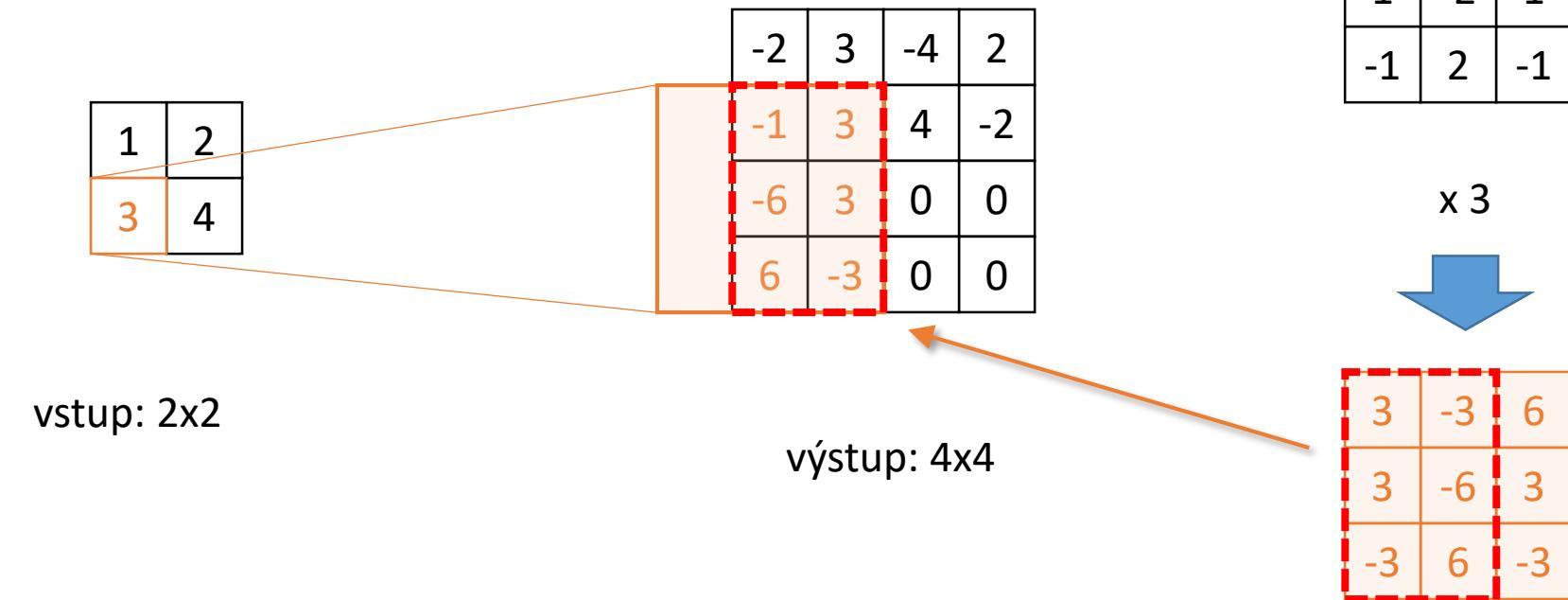


toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

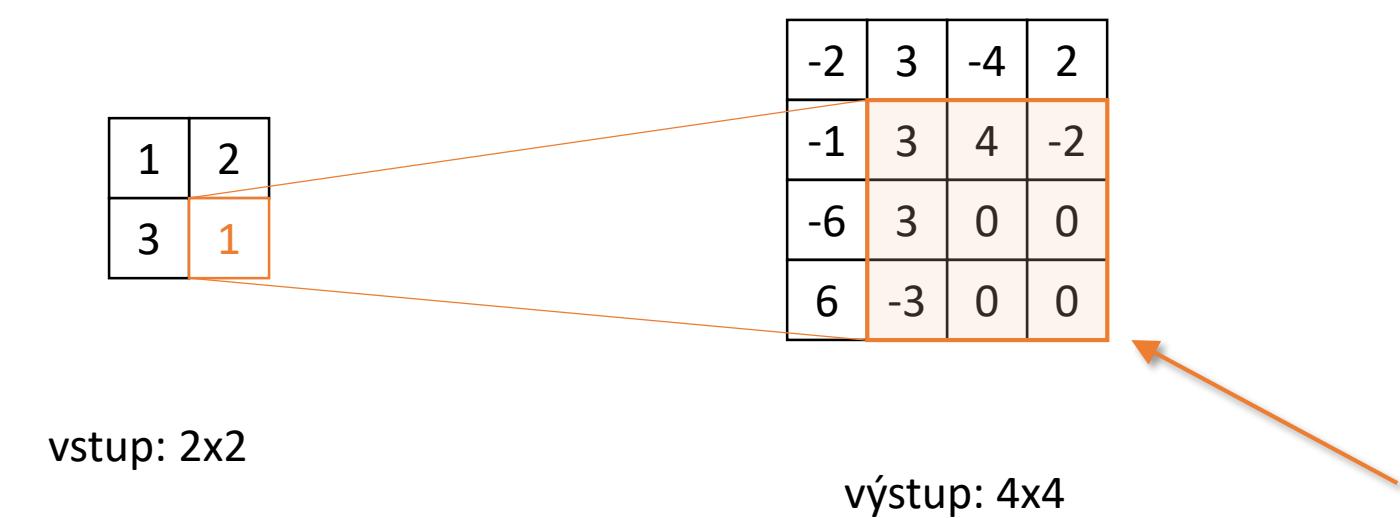


toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru



váhy konvolučního filtru

1	-1	2
1	-2	1
-1	2	-1

x 1

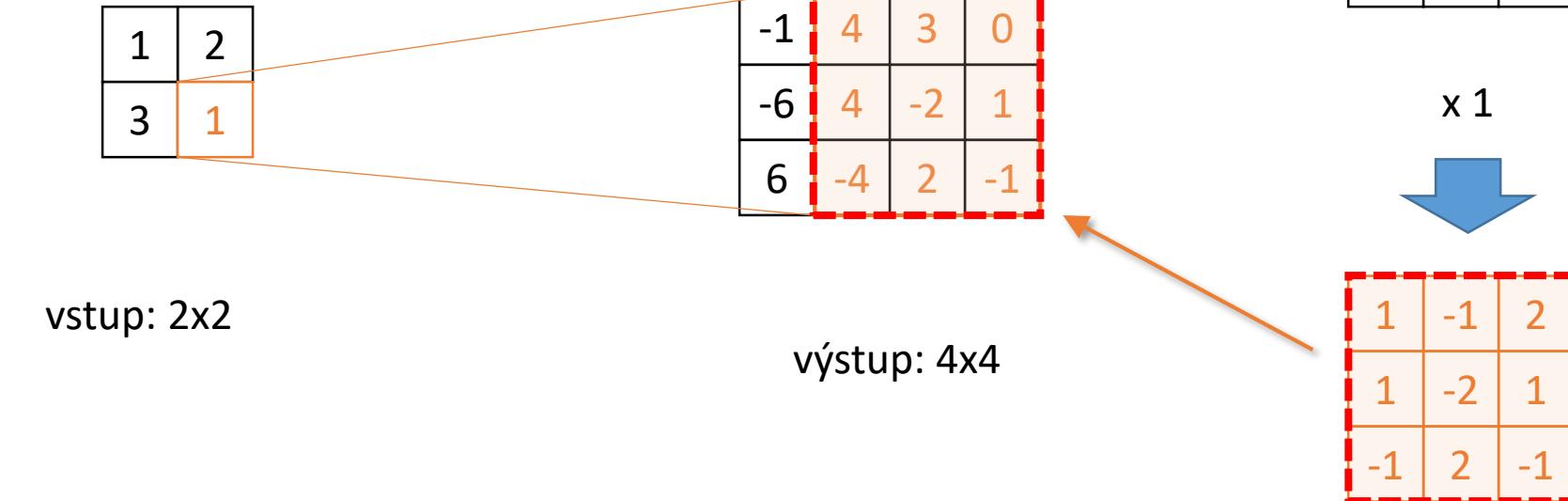
1	-1	2
1	-2	1
-1	2	-1

toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru



toto se přičte k výstupu  
na odpovídající pozici

# Transponovaná konvoluce

transponovaná konvoluce 3x3, stride 2, padding 1

všuse, kde filtr překrývá výstup, se  
přičte vstup krát váhy filtru

1	2
3	1

vstup: 2x2

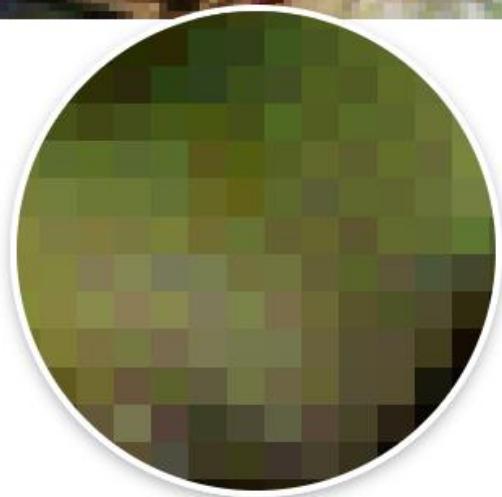
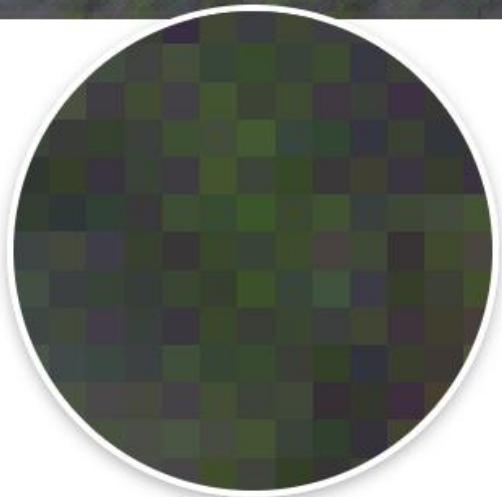
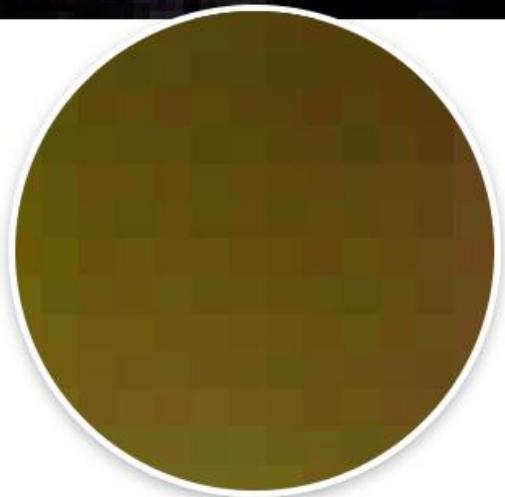
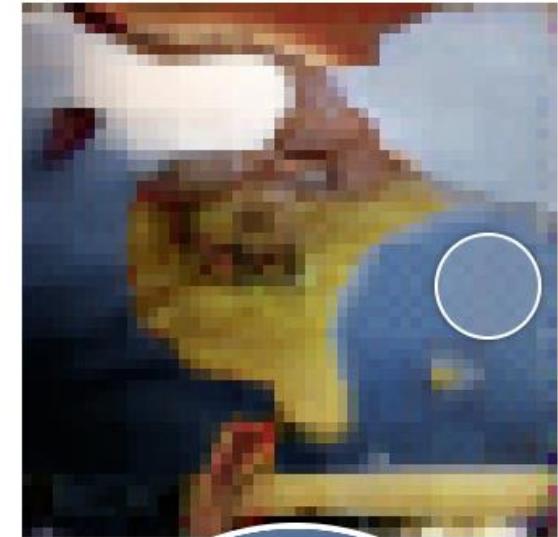
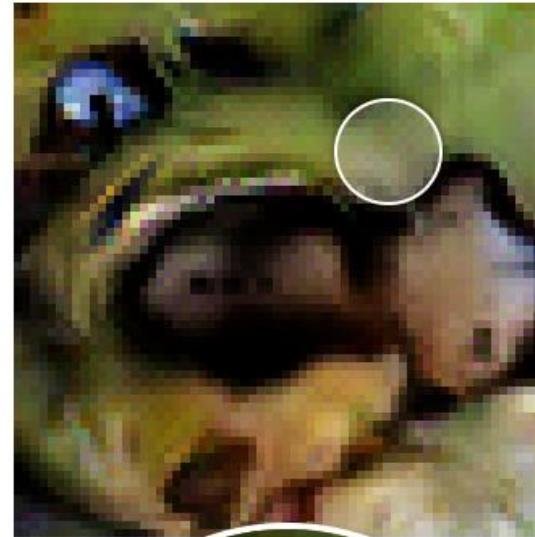
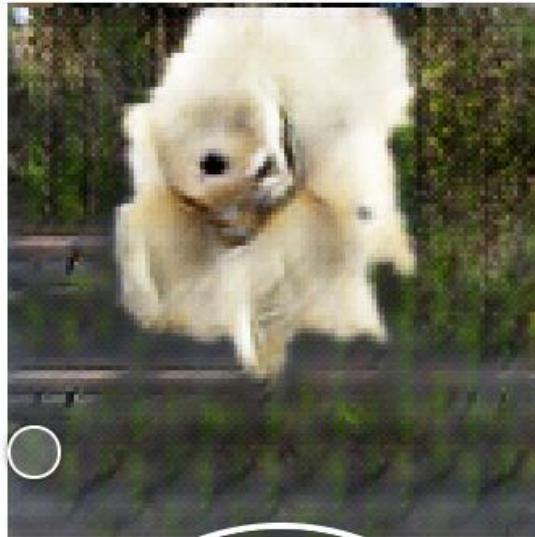
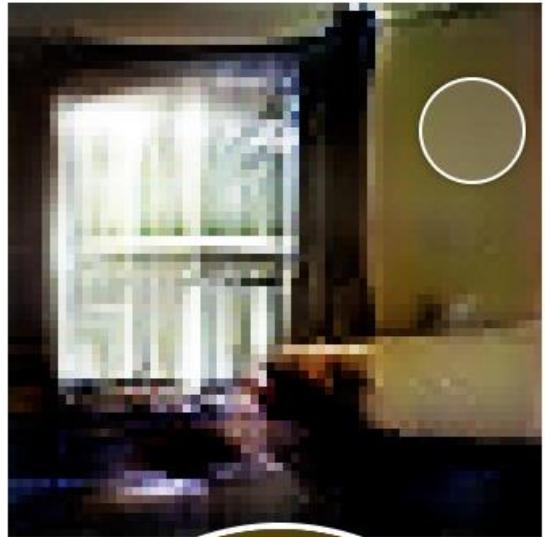
-2	3	-4	2
-1	4	3	0
-6	4	-2	1
6	-4	2	-1

výstup: 4x4

váhy konvolučního filtru

1	-1	2
1	-2	1
-1	2	-1

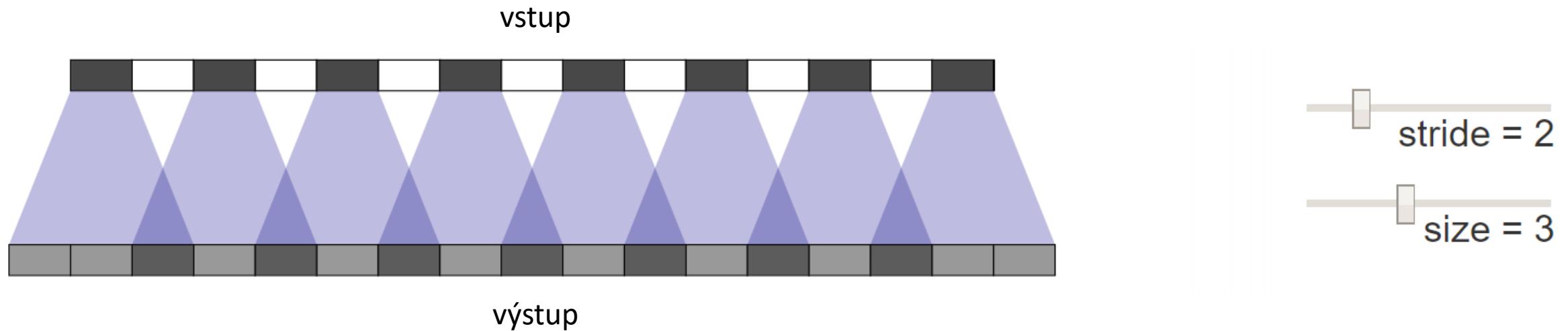
# Checkerboard artifact



obrázek: <https://distill.pub/2016/deconv-checkerboard/>

# Checkerboard artifact

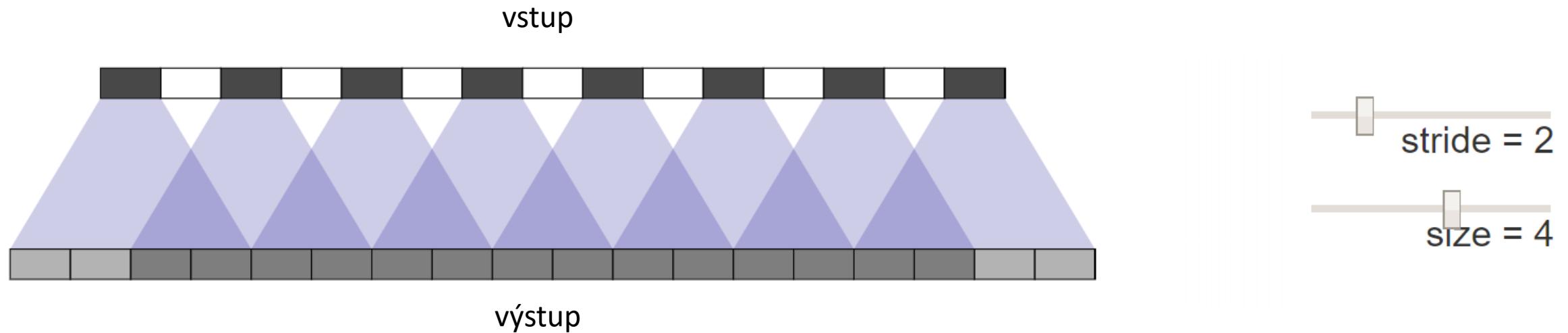
některé pixely výstupu sestávají z více součtů než ostatní → vzniká “šachovnicový” šum



obrázek: <https://distill.pub/2016/deconv-checkerboard/>

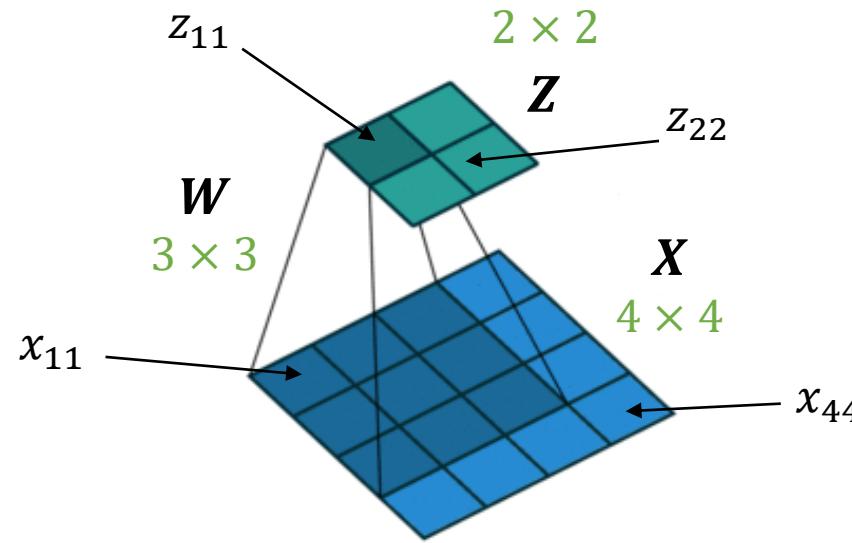
# Checkerboard artifact

pokud vhodně zvolíme kombinaci velikosti filtru a kroku, tento artefakt odstraníme



obrázek: <https://distill.pub/2016/deconv-checkerboard/>

# Konvoluce jako lineární vrstva



konvoluci na obrázku lze zapsat maticově:

$$2 \times 2 \rightarrow 4 \times 1$$

$$\begin{bmatrix} z_{11} \\ z_{12} \\ z_{21} \\ z_{22} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & 0 & w_{21} & w_{22} & w_{23} & 0 & w_{31} & w_{32} & w_{33} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & w_{13} & 0 & w_{21} & w_{22} & w_{23} & 0 & w_{31} & w_{32} & w_{33} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & w_{13} & 0 & w_{21} & w_{22} & w_{23} & 0 & w_{31} & w_{32} & w_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{11} & w_{12} & w_{13} & 0 & w_{21} & w_{22} & w_{23} & 0 & w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ \vdots \\ x_{42} \\ x_{43} \\ x_{44} \end{bmatrix}$$

... podobné lineární vrstvě

# Zpětný průchod konvoluce: gradient na vstup

- Připomeňme, že pro lineární vrstvu

$$z = \mathbf{W}x$$

$4 \times 1 \rightarrow 2 \times 2$   
 $4 \times 16$   
 $4 \times 4 \rightarrow 16 \times 1$

je gradient na vstup

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

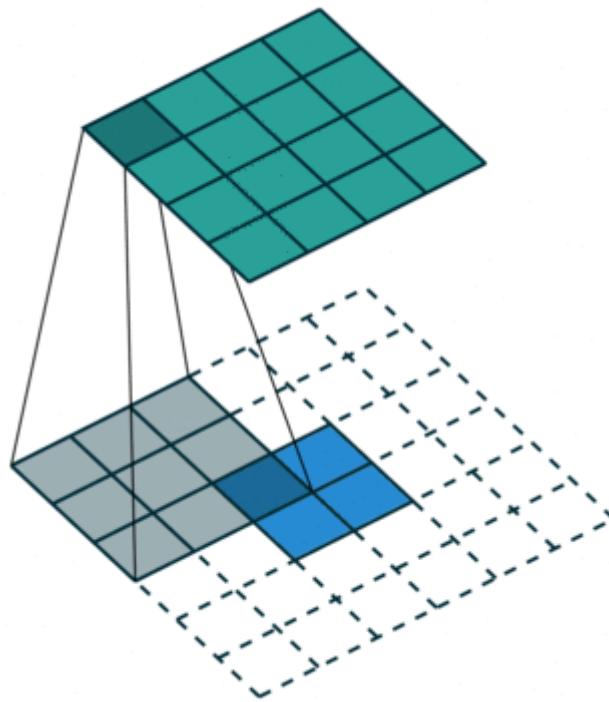
$16 \times 1 \rightarrow 4 \times 4$   
 $16 \times 4$   
 $2 \times 2 \rightarrow 4 \times 1$

- Zpětná propagace gradientu na vstup konvoluce je tedy opět konvoluce, jejíž lineární forma má transponovanou matici  $\mathbf{W}$
- Odtud anglický název **transposed convolution**
- “Obrácená” konvoluce: z tvaru výstupu  $z$  na tvar vstupu  $x$

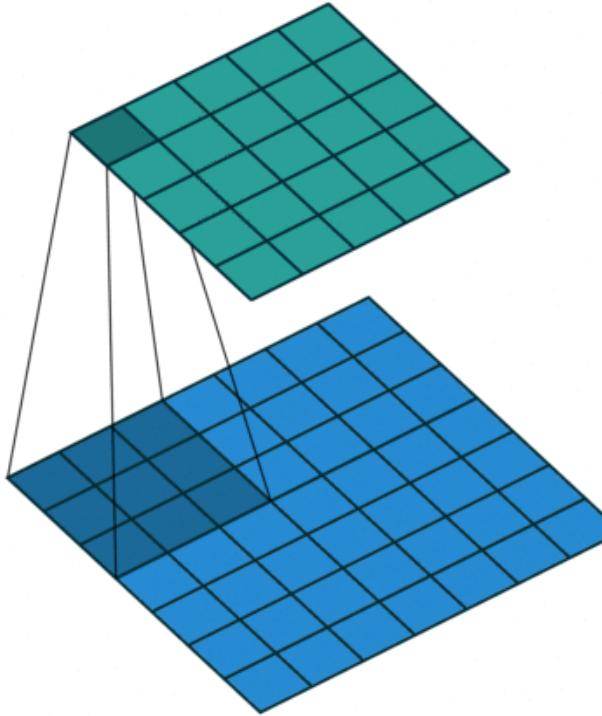
# Jiná vizualizace transponované konvoluce

pozn.: vizualizace odpovídají transponované variantě standardní konvoluce s uvedenými parametry!

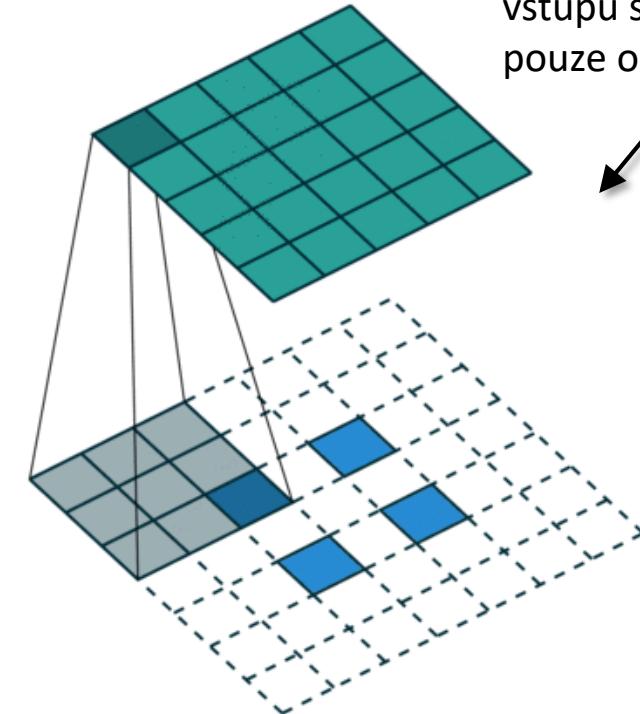
tj. kdybychom měli std. konvoluci s parametry uvedenými dole, tak její zpětný (transponovaný) průchod by vypadal právě takto



3x3, stride=1, padding=0



3x3, stride=1, padding=2



3x3, stride=2, padding=0  
(někdy jako à trous/dilated convolution)

pro každý posun +1 ve vstupu se posuneme pouze o  $\frac{1}{2}$  ve výstupu

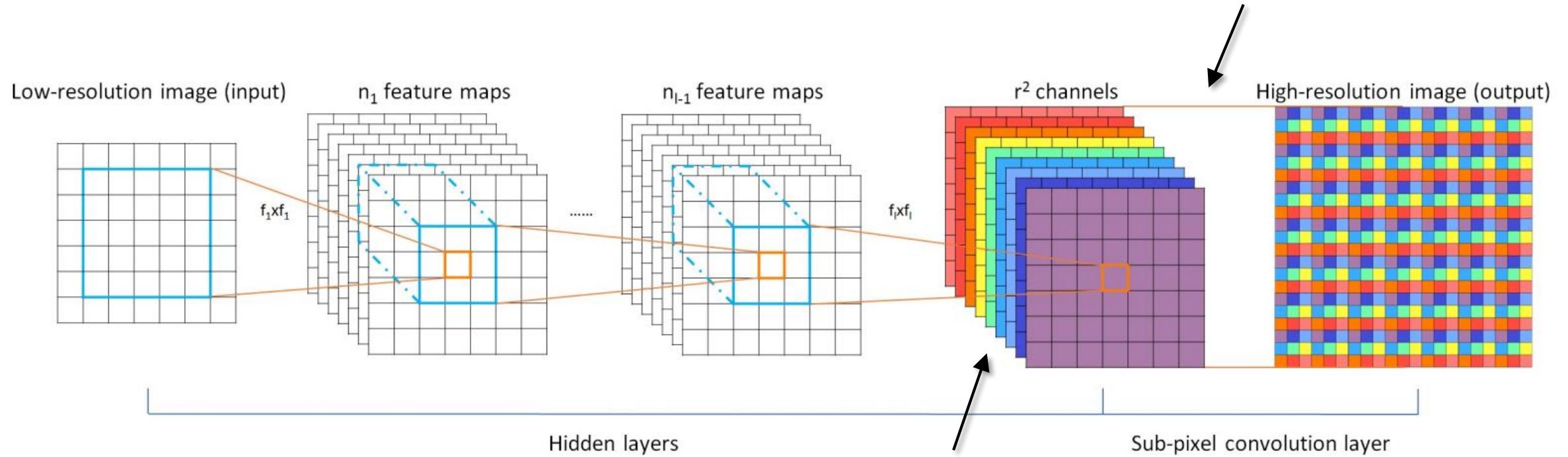
# Další názvy transponované konvoluce

- Transposed convolution
  - zřejmě “nejsprávnější” název
  - někdy se však používá jen pro speciální konfigurace stride a paddingu
- Deconvolution
  - špatně, tento termín je již zaveden pro jiný typ operace
- Upconvolution
  - Příliš se neuchytilo
- Fractionally strided convolution
  - vychází z definice kroku (stride) jako **poměru** pohybu po vstupu ku výstupu
  - u zpětného průchodu tak může být  $< 1$
- Upsampling
  - v některých frameworkech, v literatuře ne
- Dilated / à trous convolution
  - spec. případ transposed convolution se  $\text{stride} > 1$

# Sub-pixelová konvoluce

efektivnější než transponovaná konvoluce, přitom bez artefaktů!

poslední krok je vlastně jen „reshape“!



samostatný model pro každé  
požadované zvětšení  $r$

vygenerujeme  $r^2 C$ -kanálovou konvoluční mapu  
 $C$  je počet kanálů výstupu,  $r$  je faktor zvětšení

poslední reshape krok → v PyTorch třída [nn.PixelShuffle](#)

# Super resolution



- poměrně jednoduché nasbírat trénovací data, stačí jen obrázky
- učíme sítě z malého obrázku predikovat velký
- kritérium bud' mean square error (suma čtverců)
- nebo např. perceptual loss – podobné jako content loss u style transfer = porovnávají se konvoluční mapy
- namísto RGB se typicky používá YCbCr a zvětšuje se pouze kanál Y

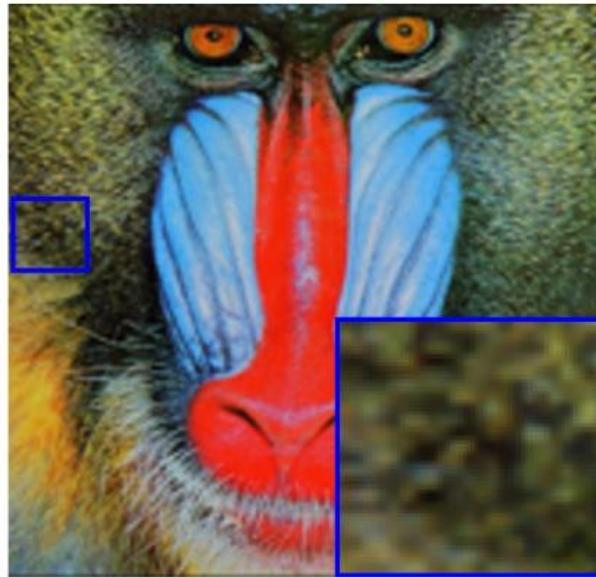
obrázek: <https://sites.google.com/a/udayton.edu/rhardie1/research/super-resolution>

# Super resolution

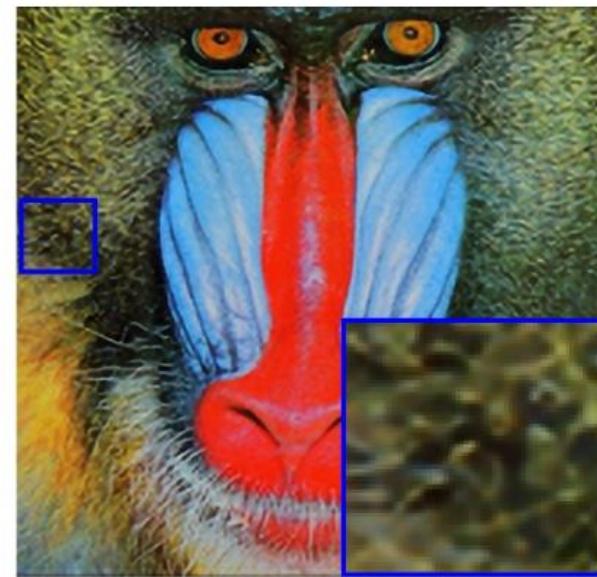
trojnásobné zvětšení



(a) Baboon Original



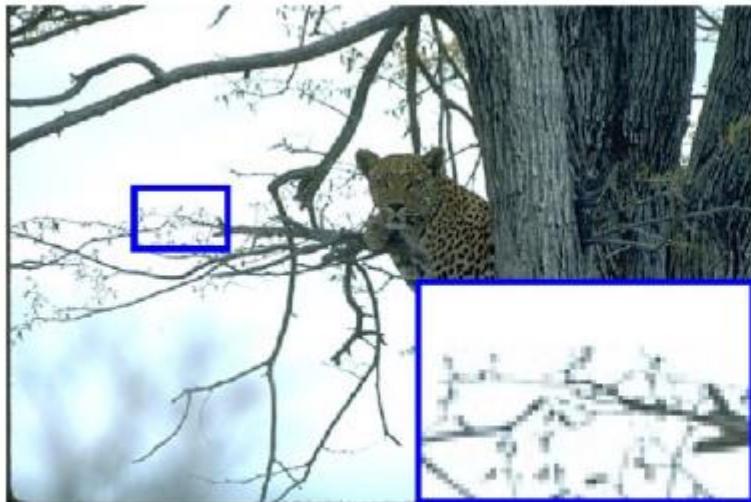
(b) Bicubic / 23.21db



(e) ESPCN / **23.72db**

# Super resolution

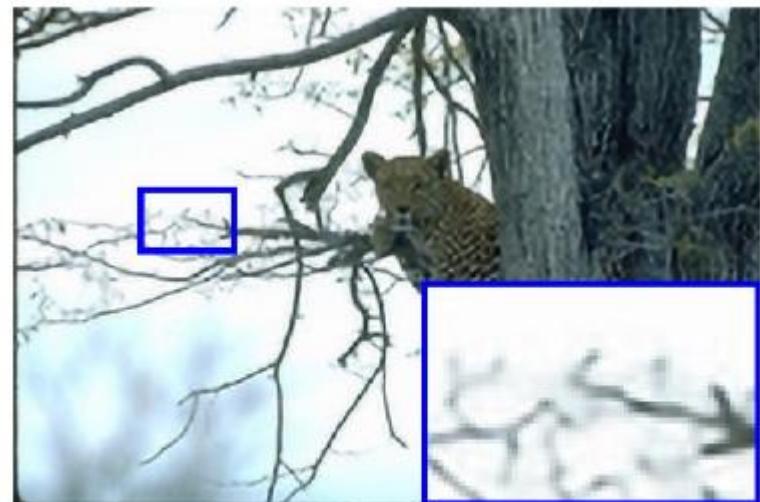
trojnásobné zvětšení



(f) 335094 Original



(g) Bicubic / 22.24db



(j) ESPCN / 24.14db

# Super resolution

trojnásobné zvětšení



(k) Monarch Original

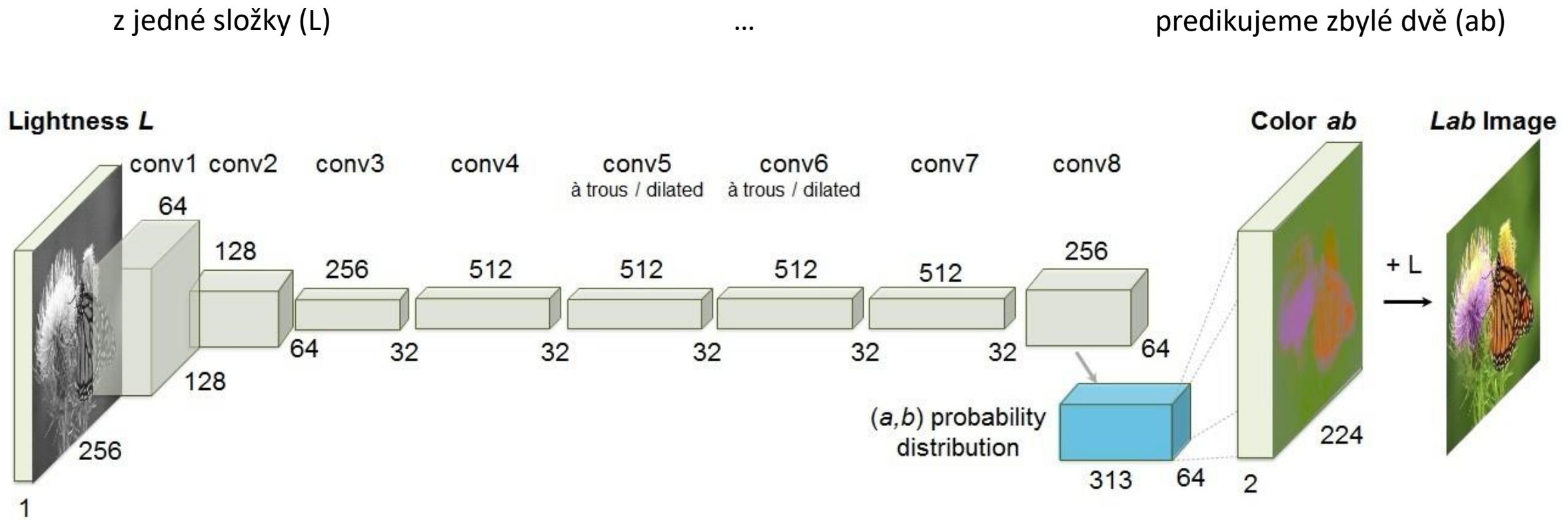


(l) Bicubic / 29.43db



(o) ESPCN / **33.66db**

# Automatické barvení obrázků (image colorization)



opět ne RGB, ale jiný obarevný prostor: zde typicky Lab či LUV (jas → dvě barevné složky)

# Automatické barvení obrázků (image colorization)



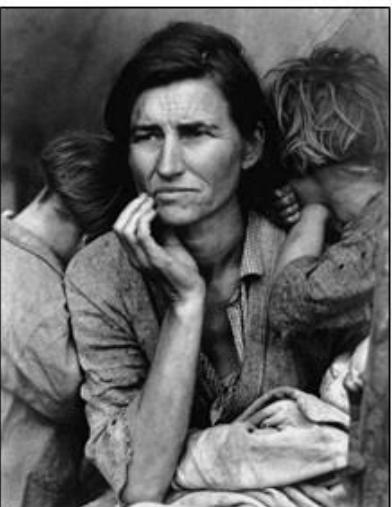
zdroj: [Zhang et al.: Colorful Image Colorization](#)

# Automatické barvení obrázků (image colorization)



zdroj: [Zhang et al.: Colorful Image Colorization](#)

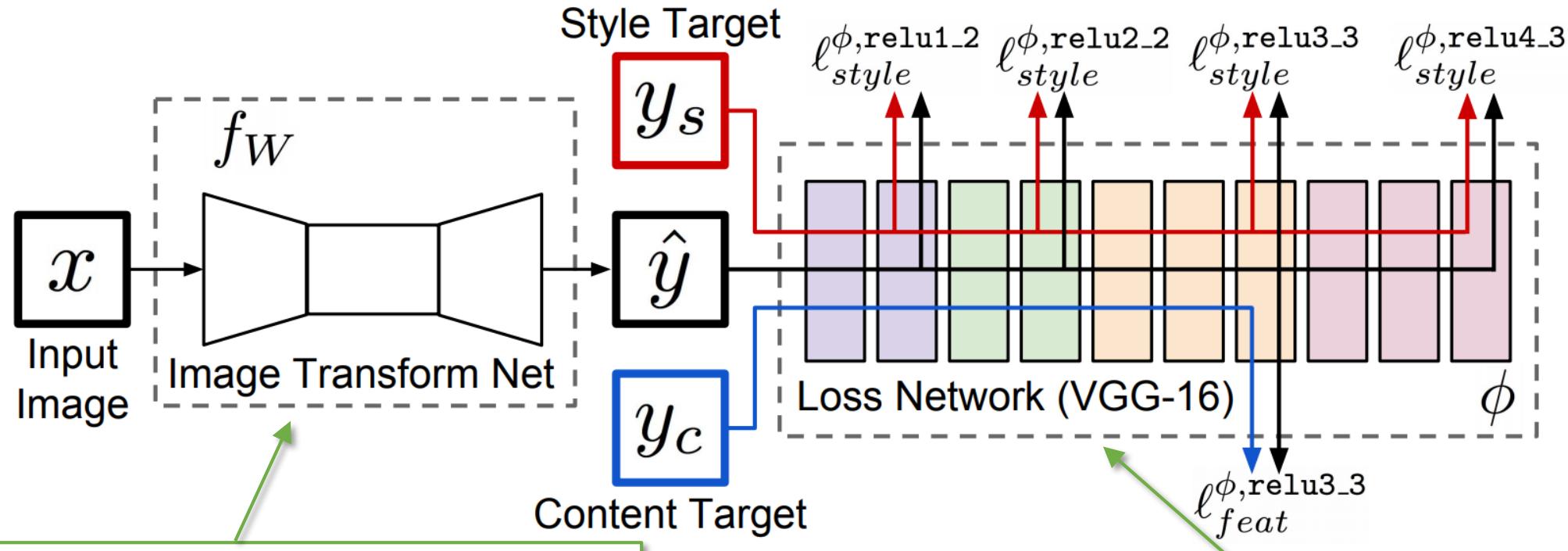
# Automatické barvení obrázků (image colorization)



zdroj: [Zhang et al.: Colorful Image Colorization](#)

# Fast style transfer

nahrazuje postupnou a pomalou optimalizaci vstupu učením sítě, která, jakmile naučená, stylizuje obrázek jediným průchodem



natrénovaná konvoluční síť, která ze vstupu  
přímo predikuje stylizovaný obrázek

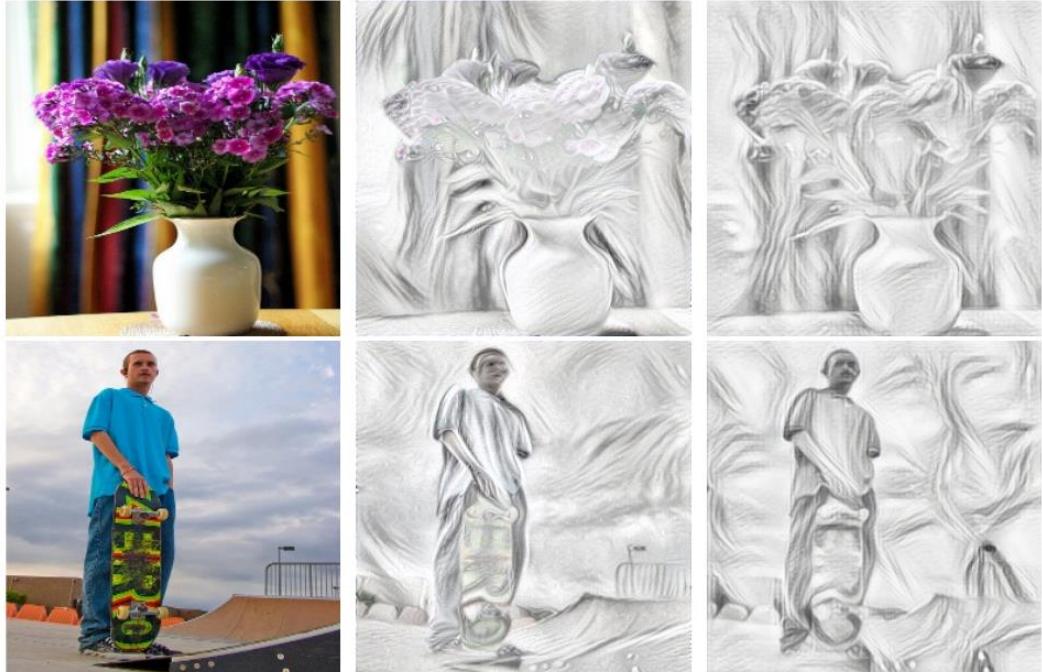
tato část zůstává stejná, tj. porovnávají se konvoluční  
mapy (perceptual loss) a jejich style (gram matice)

je potřeba samostatnou síť pro každý styl; toto omezení však bylo odstraněno v dalších pracích

článek: [Johnson et al.: Perceptual Losses for Real-Time Style Transfer and Super-Resolution](#)

# Fast style transfer

Style  
*Sketch*



obsah (content)

původní transfer

fast transfer

Style  
*The Simpsons*



obsah (content)

původní transfer

fast transfer

článek: [Johnson et al.: Perceptual Losses for Real-Time Style Transfer and Super-Resolution](#)

# Fast style transfer → super resolution

stejný algoritmus (perceptual loss) autoři aplikovali i na problém super resolution!



<b>Ground Truth</b>	<b>Bicubic</b>	<b>Ours (<math>\ell_{pixel}</math>)</b>	<b>SRCCN [11]</b>	<b>Ours (<math>\ell_{feat}</math>)</b>
This image	31.78 / 0.8577	31.47 / 0.8573	32.99 / 0.8784	29.24 / 0.7841
Set5 mean	28.43 / 0.8114	28.40 / 0.8205	30.48 / 0.8628	27.09 / 0.7680

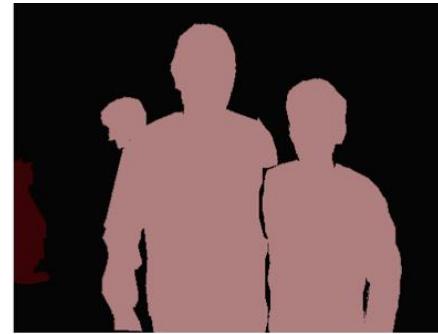
# Fast style transfer → super resolution

stejný algoritmus (perceptual loss) autoři aplikovali i na problém super resolution!



článek: [Johnson et al.: Perceptual Losses for Real-Time Style Transfer and Super-Resolution](#)

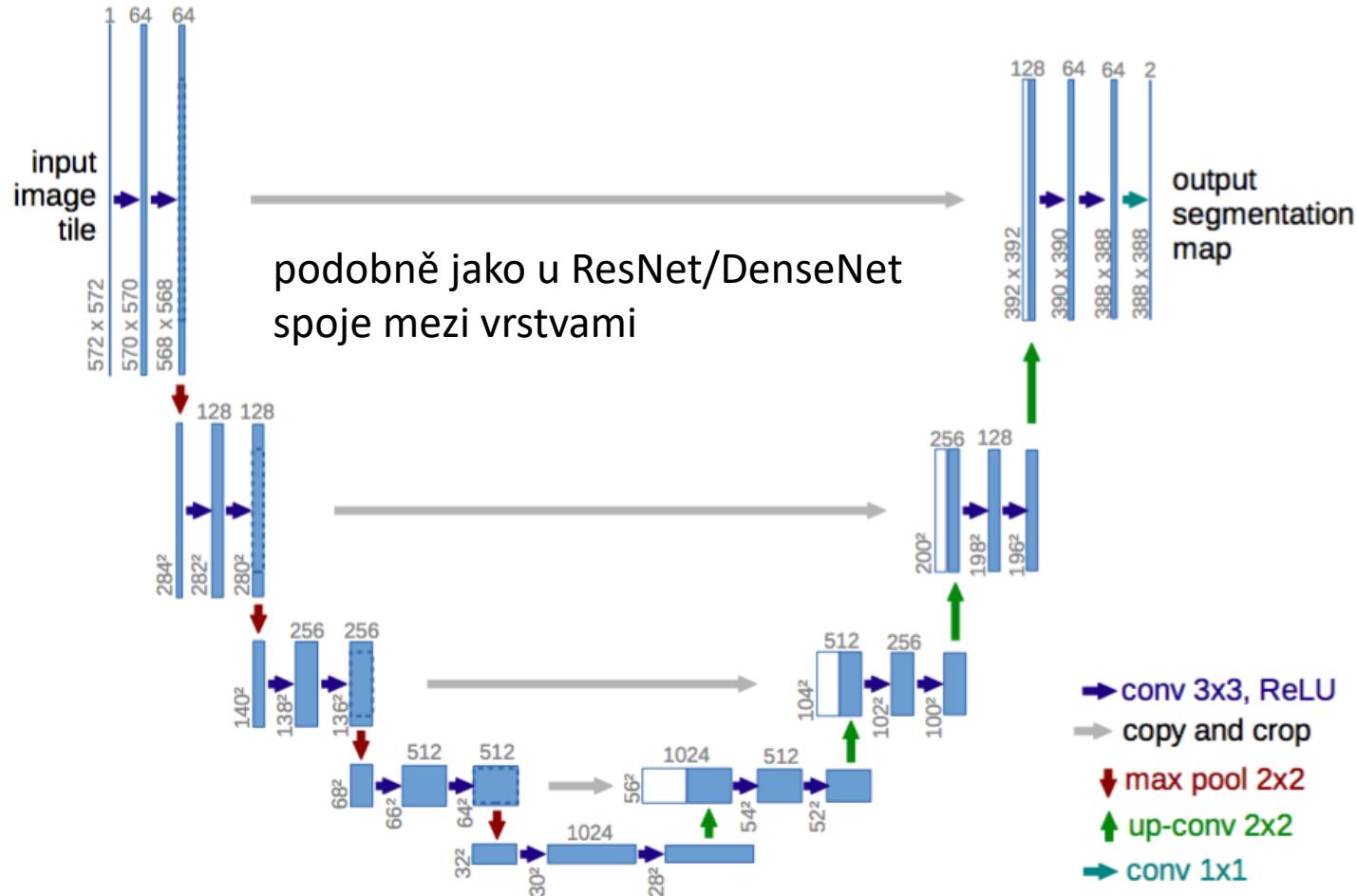
# Sémantická segmentace



- lze zformulovat jako klasifikaci
- výstup ze sítě je predikce třídy pro každý pixel
- lze tedy použít standardní softmax, loss pro obrázek je pak suma přes všechny pixely
- drahá trénovací data – potřebujeme manuálně per-pixel segmentované obrázky!

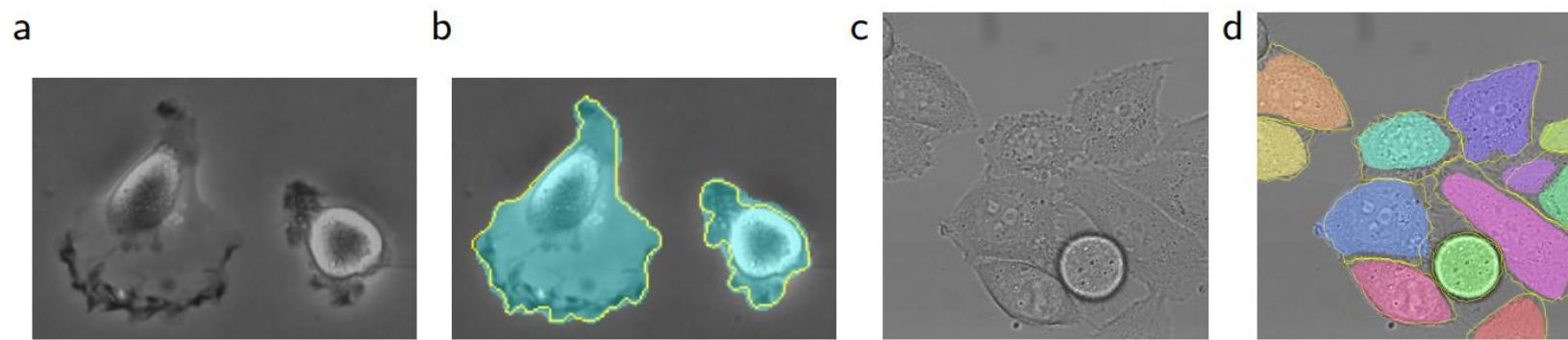
# Sémantická segmentace U-net

architektura typu enkodér - dekodér



článek: [Ronneberger et al.: U-Net: Convolutional Networks for Biomedical Image Segmentation](#)

# Sémantická segmentace U-net



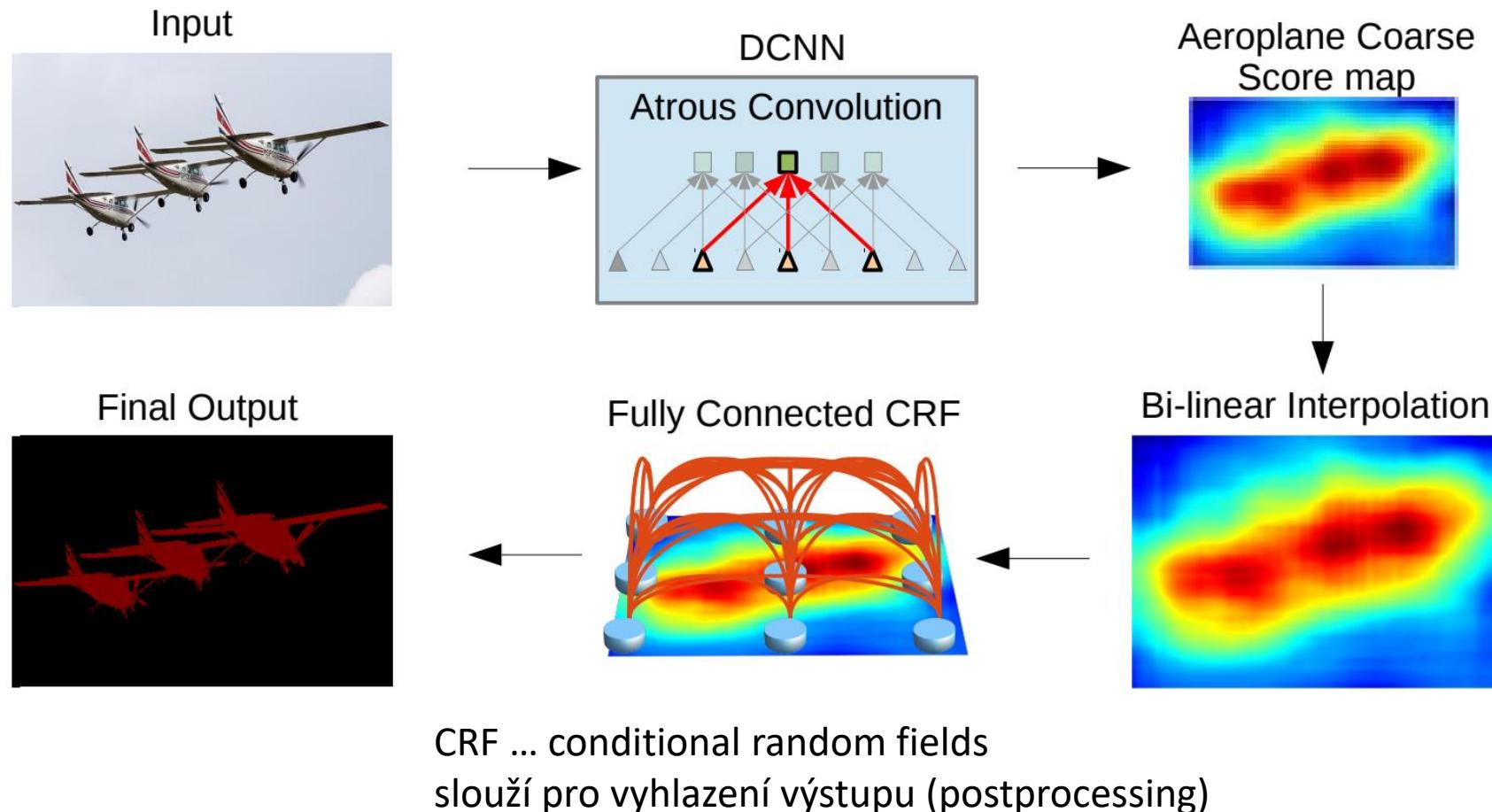
**Fig. 4.** Result on the ISBI cell tracking challenge. (a) part of an input image of the “PhC-U373” data set. (b) Segmentation result (cyan mask) with manual ground truth (yellow border) (c) input image of the “DIC-HeLa” data set. (d) Segmentation result (random colored masks) with manual ground truth (yellow border).

**Table 2.** Segmentation results (IOU) on the ISBI cell tracking challenge 2015

Name	PhC-U373	DIC-HeLa
IMCB-SG (2014)	0.2669	0.2935
KTH-SE (2014)	0.7953	0.4607
HOUS-US (2014)	0.5323	-
second-best 2015	0.83	0.46
u-net (2015)	<b>0.9203</b>	<b>0.7756</b>

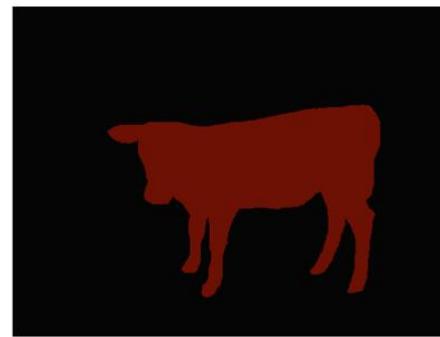
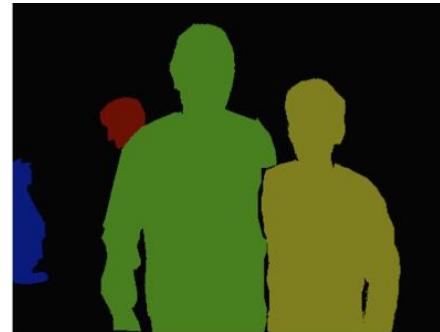
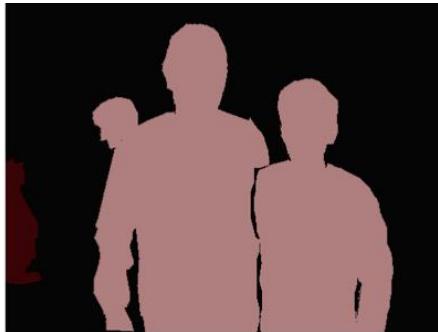
# Sémantická segmentace skrze dilated konvoluce

- používá VGG, ze které autoři odstranili všechny max poolingy a nahradili konvolucemi se stride > 1
- upsampling potom opět pomocí dilated konvolucí = transponovaná konvolece se stride > 1



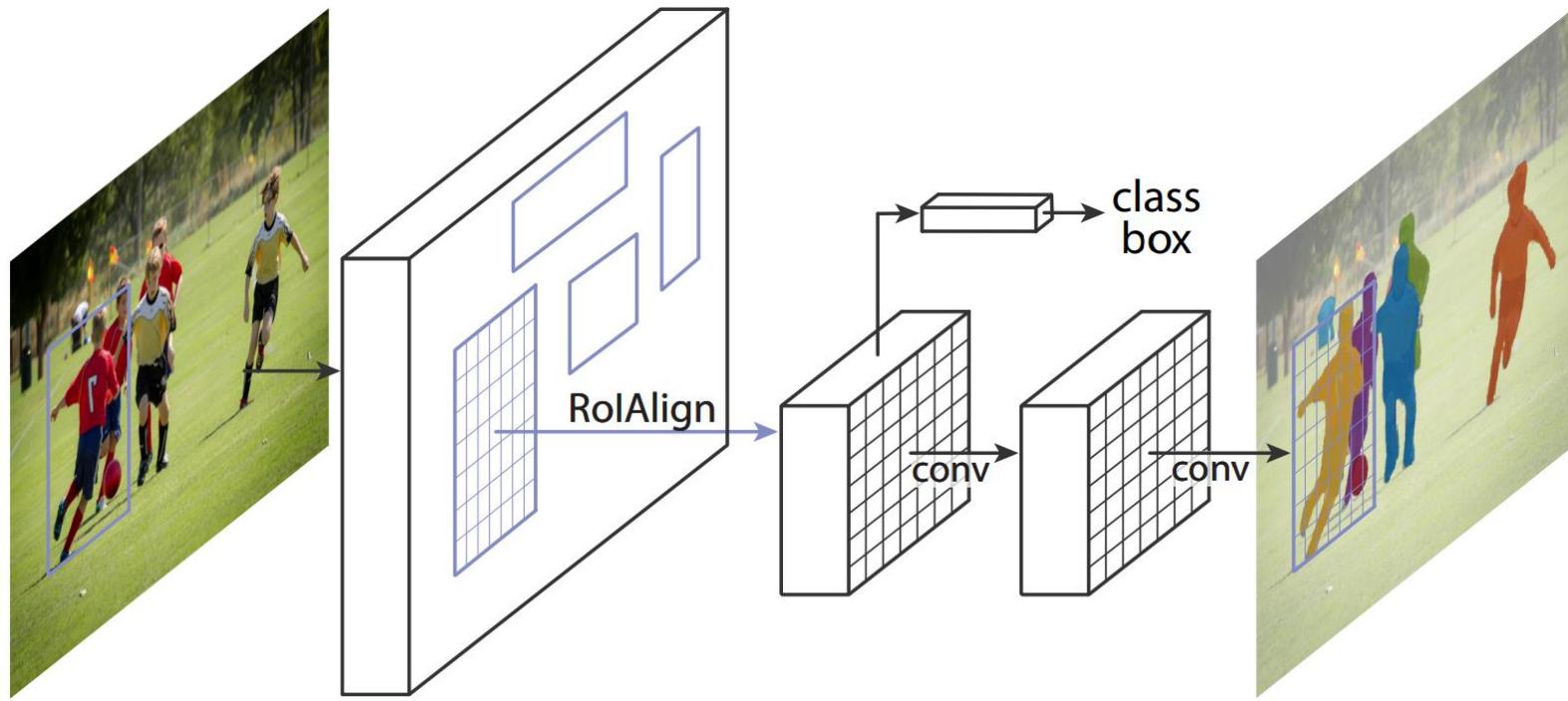
článek: [Chen et al: DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs](#)

# Segmentace objektů (instance segmentation)



- na rozdíl od sémantické segmentace rozlišujeme mezi jednotlivými instancemi tříd (objekty)!

# Mask R-CNN



- kombinace Faster R-CNN pro detekci objektů a segmentace
- pro každý výstupní region je zároveň predikovaná i maska → segmentace objektů, nikoliv tříd
- segmentace per-pixelově predikuje objekt vs neobjekt
- oproti Faster R-CNN tedy obsahuje jednu segmentační větev navíc

článek: [He et al.: Mask R-CNN](#)

# Mask R-CNN

	backbone	AP <sup>bb</sup>	AP <sup>bb</sup> <sub>50</sub>	AP <sup>bb</sup> <sub>75</sub>	AP <sup>bb</sup> <sub>S</sub>	AP <sup>bb</sup> <sub>M</sub>	AP <sup>bb</sup> <sub>L</sub>
Faster R-CNN+++ [19]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [27]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [21]	Inception-ResNet-v2 [41]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [39]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	<b>52.1</b>
Faster R-CNN, RoIAlign	ResNet-101-FPN	37.3	59.6	40.3	19.8	40.2	48.8
<b>Mask R-CNN</b>	ResNet-101-FPN	38.2	60.3	41.7	20.1	41.1	50.2
<b>Mask R-CNN</b>	ResNeXt-101-FPN	<b>39.8</b>	<b>62.3</b>	<b>43.4</b>	<b>22.1</b>	<b>43.2</b>	51.2

Table 3. **Object detection single-model** results (bounding box AP), vs. state-of-the-art on test-dev. Mask R-CNN using ResNet-101-FPN outperforms the base variants of all previous state-of-the-art models (the mask output is ignored in these experiments). The gains of Mask R-CNN over [27] come from using RoIAlign (+1.1 AP<sup>bb</sup>), multitask training (+0.9 AP<sup>bb</sup>), and ResNeXt-101 (+1.6 AP<sup>bb</sup>).

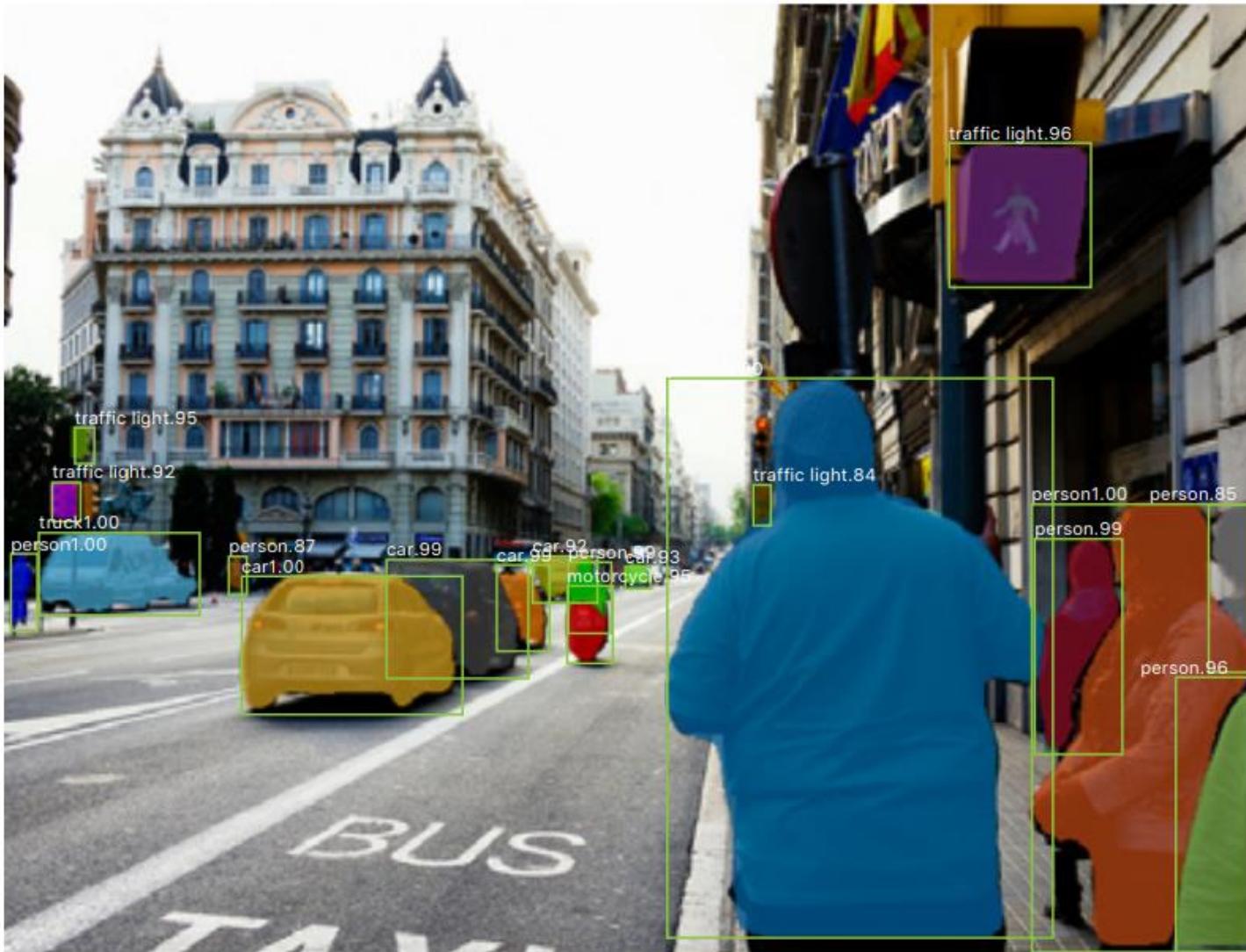
- dosahuje state-of-the-art i pro obyčejnou detekci objektů (bounding boxy!)
- cca 200 ms (5 FPS) na obrázek na NVidia Tesla M40 GPU
- cca 400 ms (2.5 FPS) na obrázek, pokud region proposal net (RPN) a segmentační části sítě nesdílí váhy

# Mask R-CNN



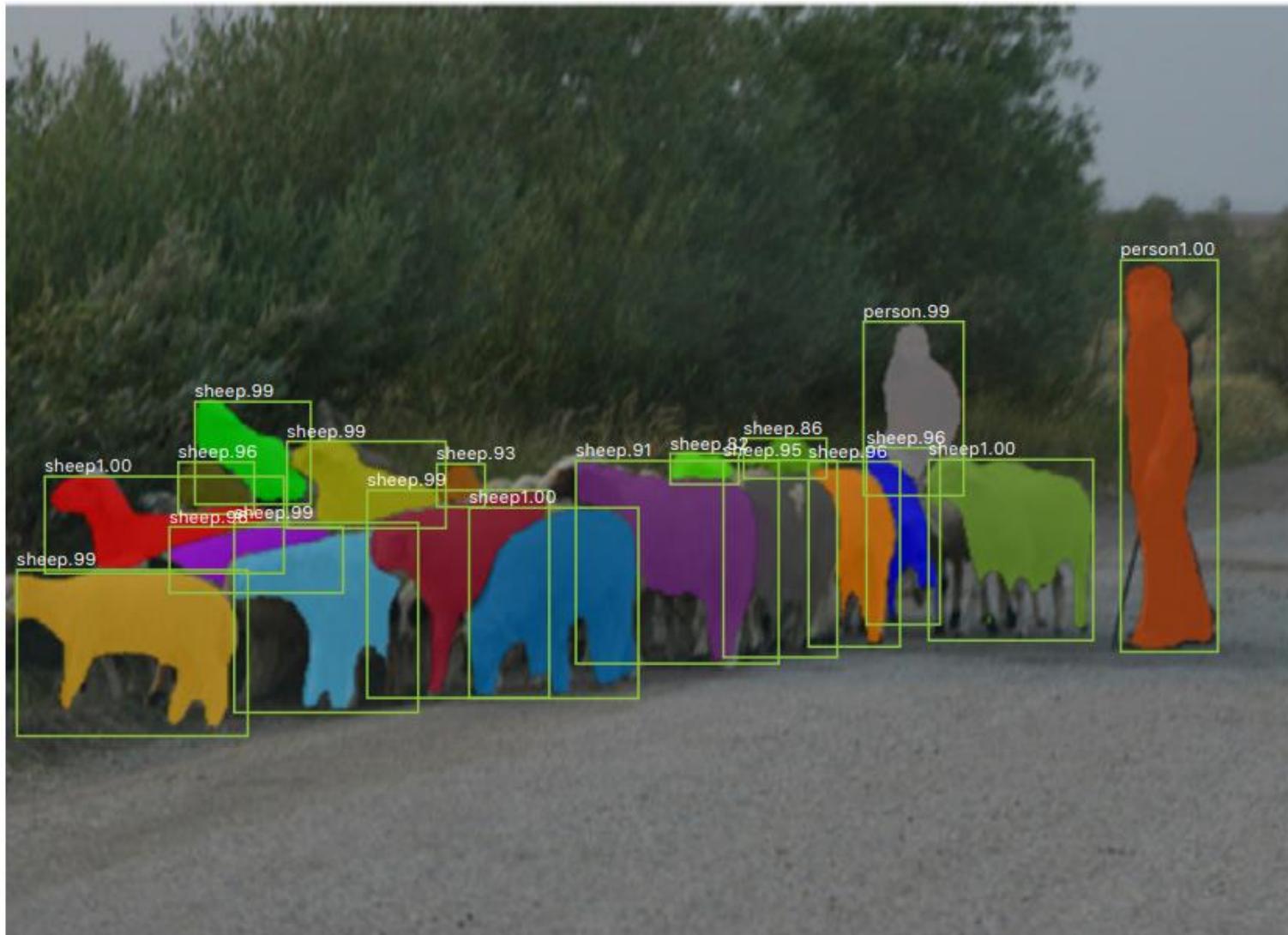
článek: [He et al.: Mask R-CNN](#)

# Mask R-CNN



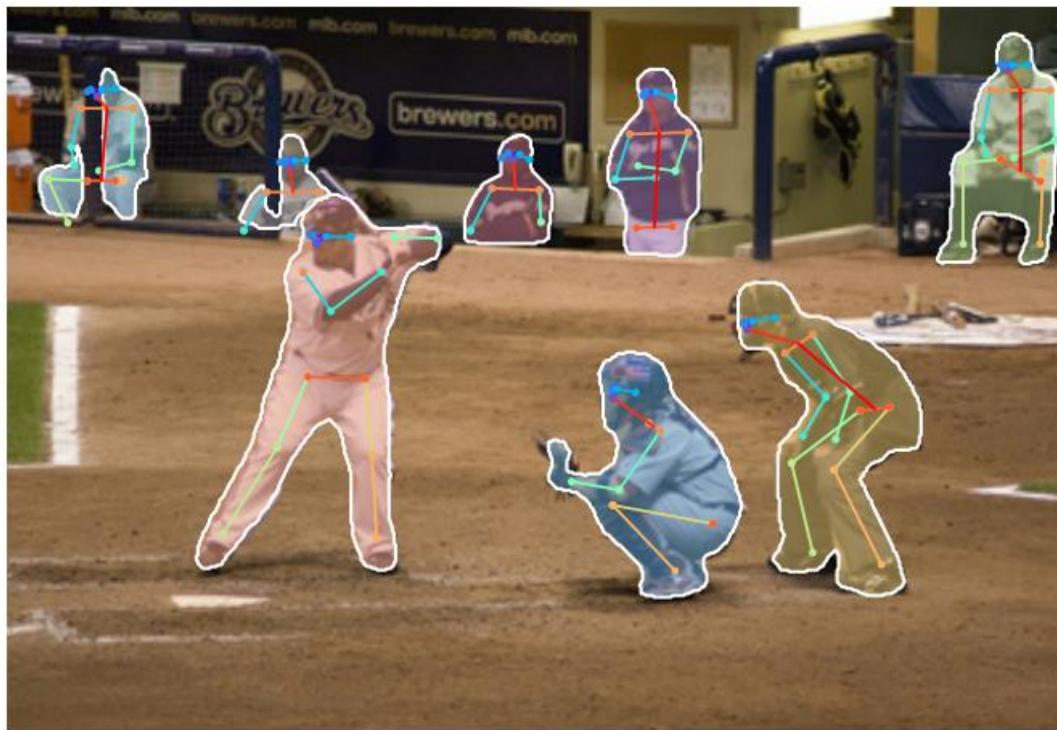
článek: [He et al.: Mask R-CNN](#)

# Mask R-CNN



článek: [He et al.: Mask R-CNN](#)

# Mask R-CNN pro pose estimation



- autoři rozšířili i pro pose estimation
- pro každý kloub (keypoint) síť predikuje one-hot masku = pouze jediný pixel je “1”, ostatní “0”

# Učení bez učitele a generativní modely

---

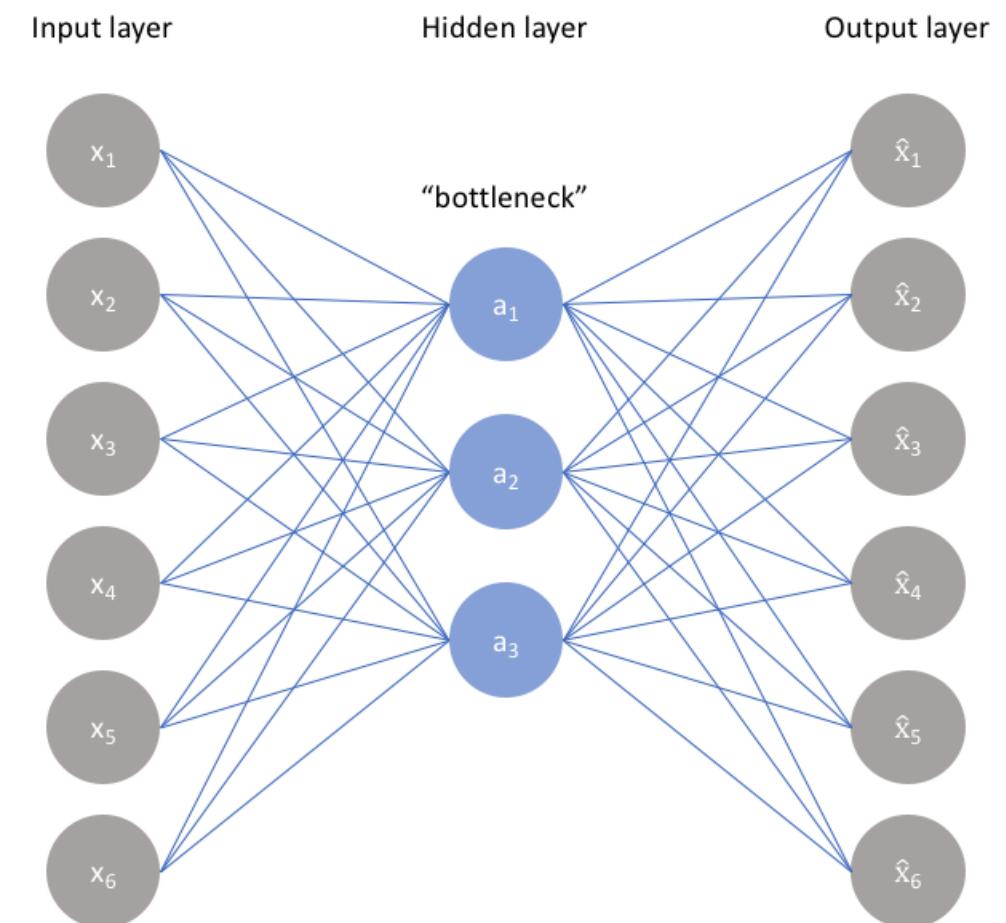
# Autoenkodér

- úkolem rekonstrukce vstupních dat
- architektura typu encoder - dekodér
- vstupní vektor/obrázek je redukován enkodérem na nějaký *bottleneck*
- poté znova rekonstruován dekodérem
- lze nahlížet jako na kompresi
- cílem co nejmenší rekonstrukční odchylka, např. mean square error (MSE)

$$L(x, x') = \frac{1}{D} \|x - x'\|^2$$

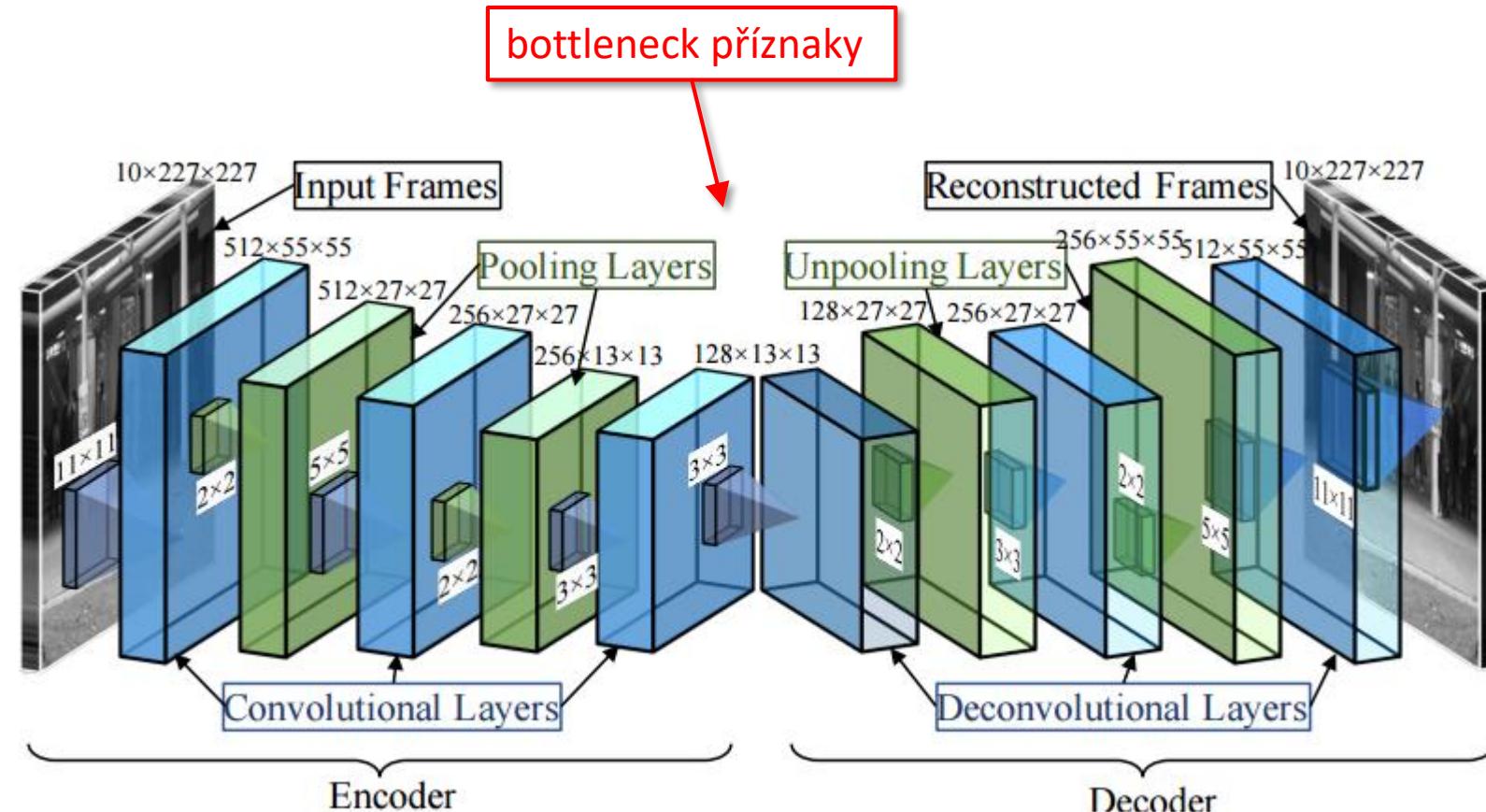
kde  $D$  je délka vektoru  $x$

- Spec. případ s jednou skrytou vrstvou a bez nelinearity  
≈ PCA
  - nejsou to samé, ale výsledné váhy pokrývají stejný prostor
  - PCA má navíc omezení jejich vzájemné ortogonality



obrázek: <https://www.jeremyjordan.me/autoencoders/>

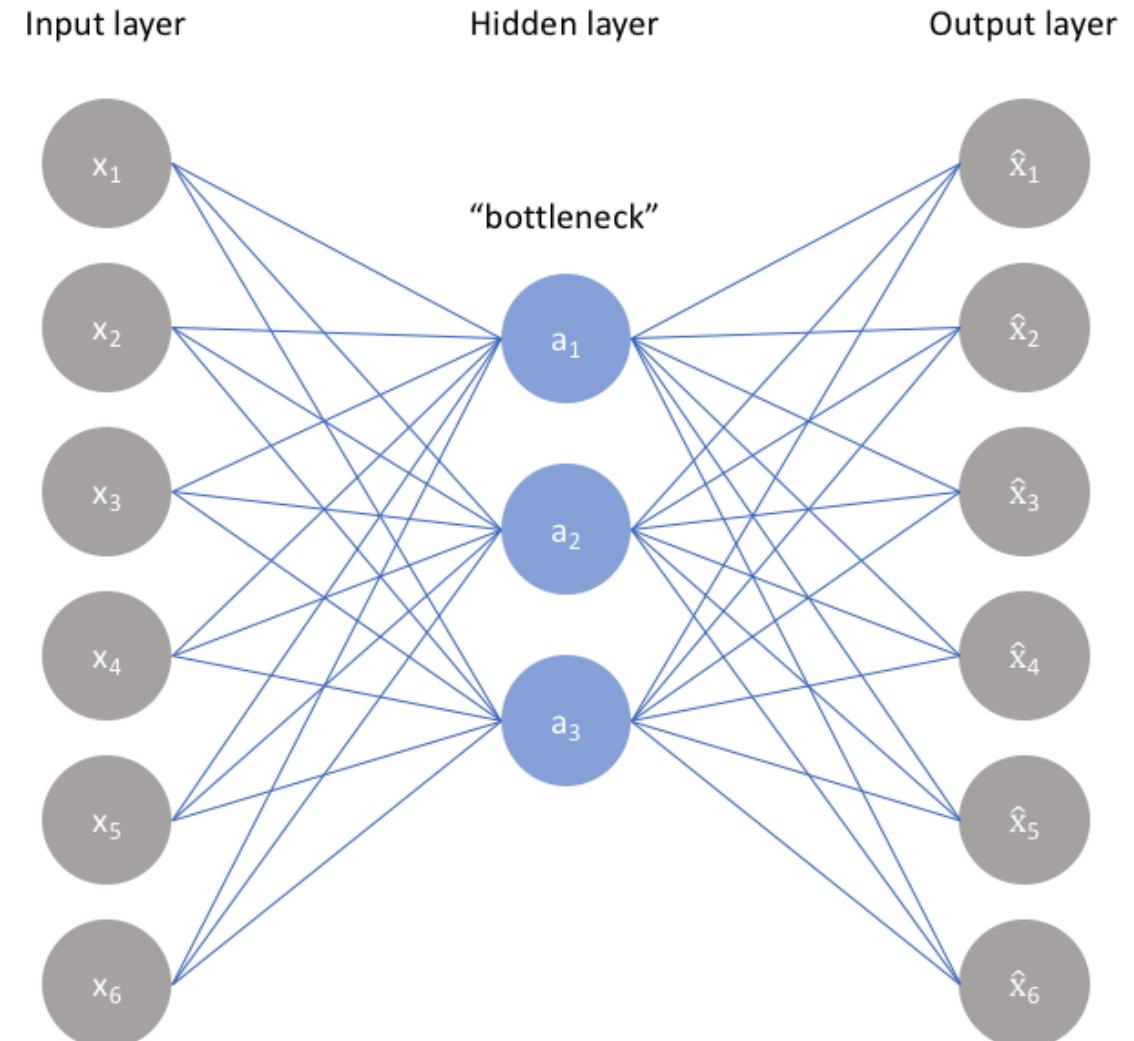
# Konvoluční autoenkodér



- nahrazuje fully connected vrstvy konvolučními

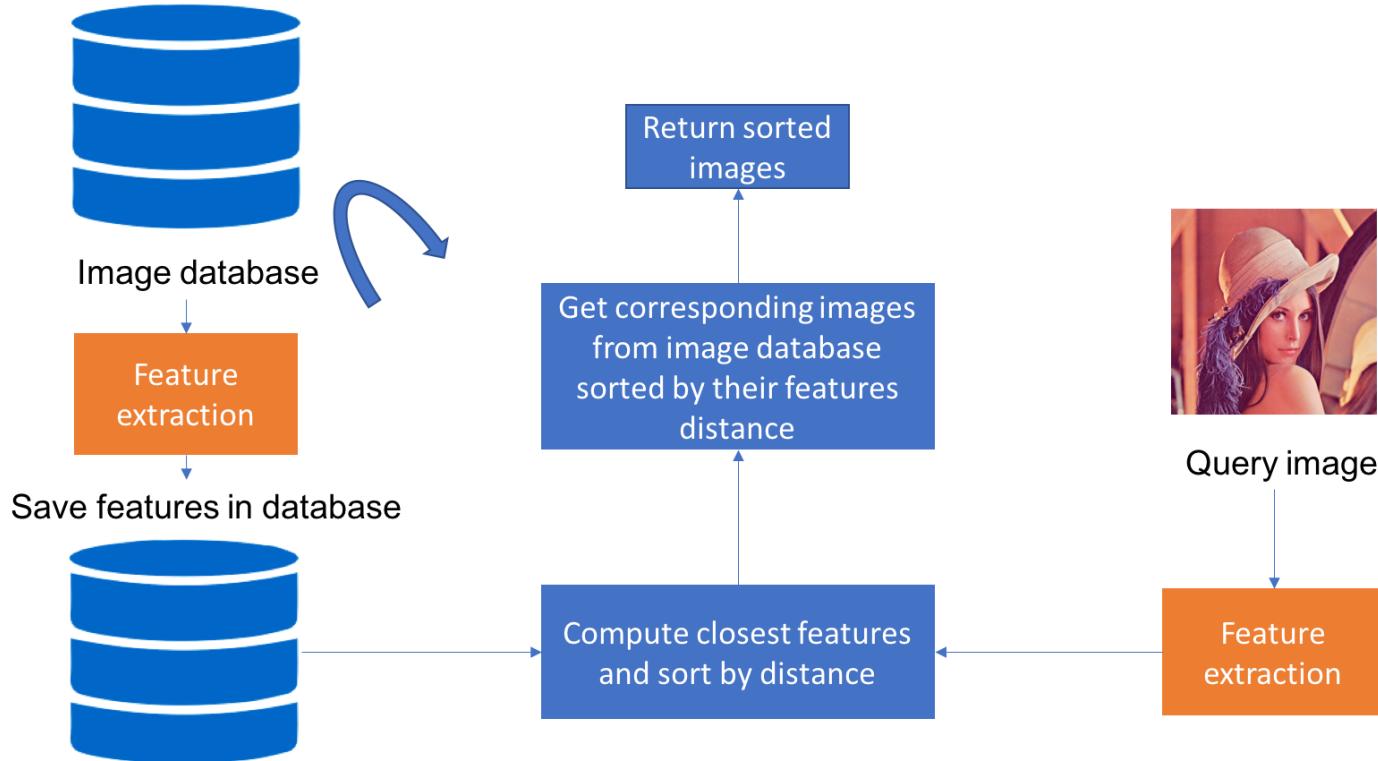
# Autoenkodér pro detekci anomálií

- myšlenka: natrénujeme na “čistých” datech
- v testovací fázi:
  - známá data, která odpovídají trénovacím → malá rekonstrukční odchylka
  - neznámá data (anomálie) → velká odchylka
- V literature jako anomalies či outlier detection
- lze využít např. pro detekci nečistot v materiálu apod.

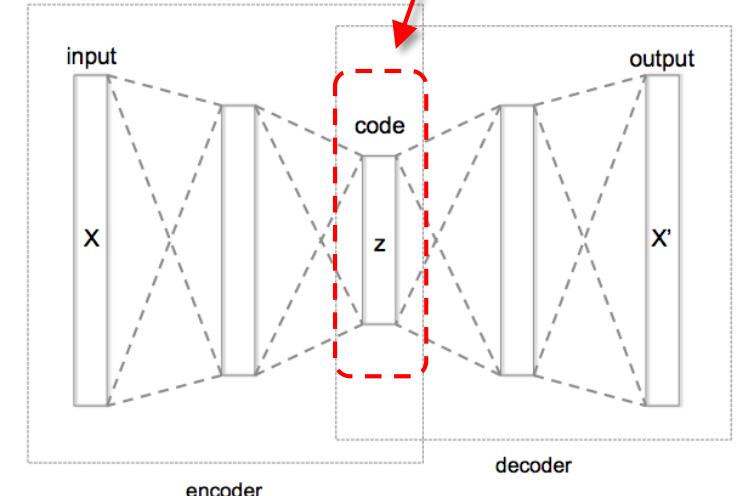


# Content-based image retrieval

- myšlenka: autoenkovdér natrénovaný na velkém množství obrázků použít pro extrakci příznaků



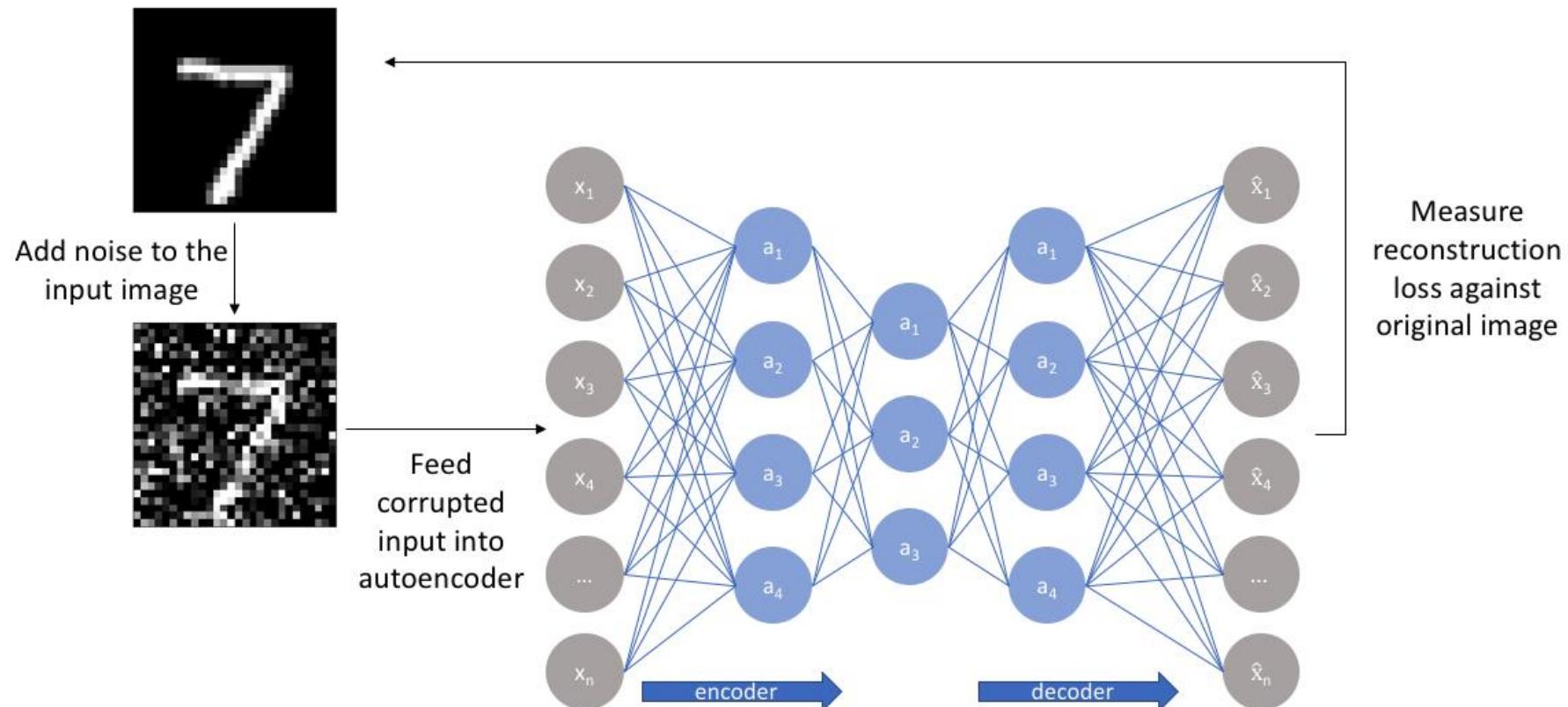
bottleneck vektor jako příznaky



- nejpodobnější obrázek vyhledáme např. metodou nejbližšího soused nad extrahovanými příznaky

obrázek: <https://blog.sicara.com/keras-tutorial-content-based-image-retrieval-convolutional-denoising-autoencoder-dc91450cc511>

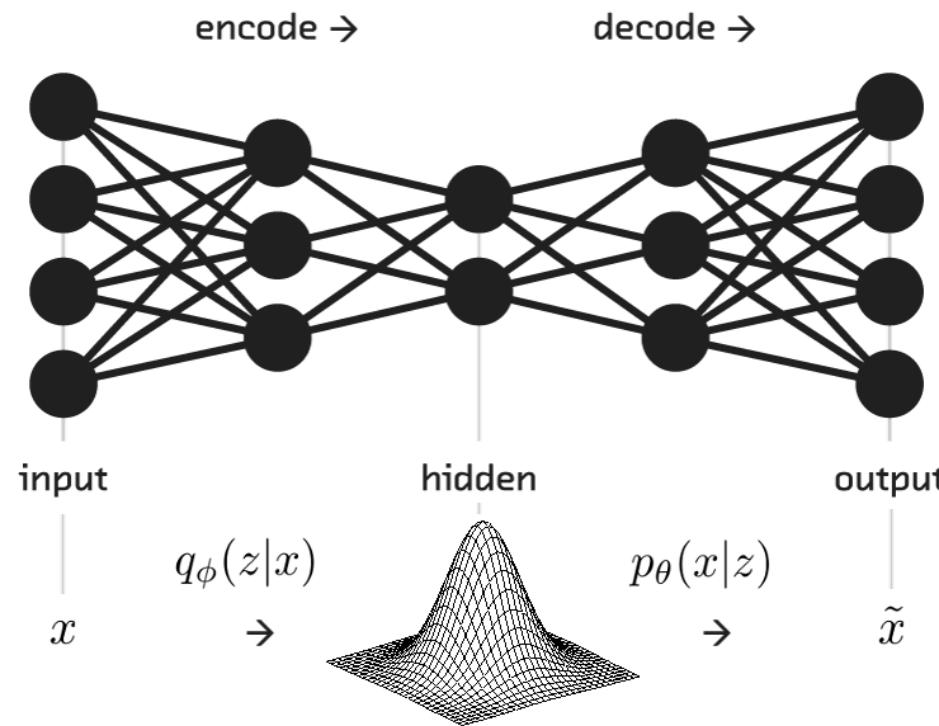
# Denoising autoencoder



- vstup je zašuměný → síť rekonstruuje čistý obrázek
- natrénovaný autoenkovdér odstraní šum
- lze využít i pro inpainting apod.
- princip formulace podobný jako u colorization, super resolution, ...

obrázek: <https://www.jeremyjordan.me/autoencoders/>

# Variační autoenkodér



celkový loss

$$\mathcal{L}(x, \hat{x}) + \sum_j KL(q_j(z|x) || p(z))$$

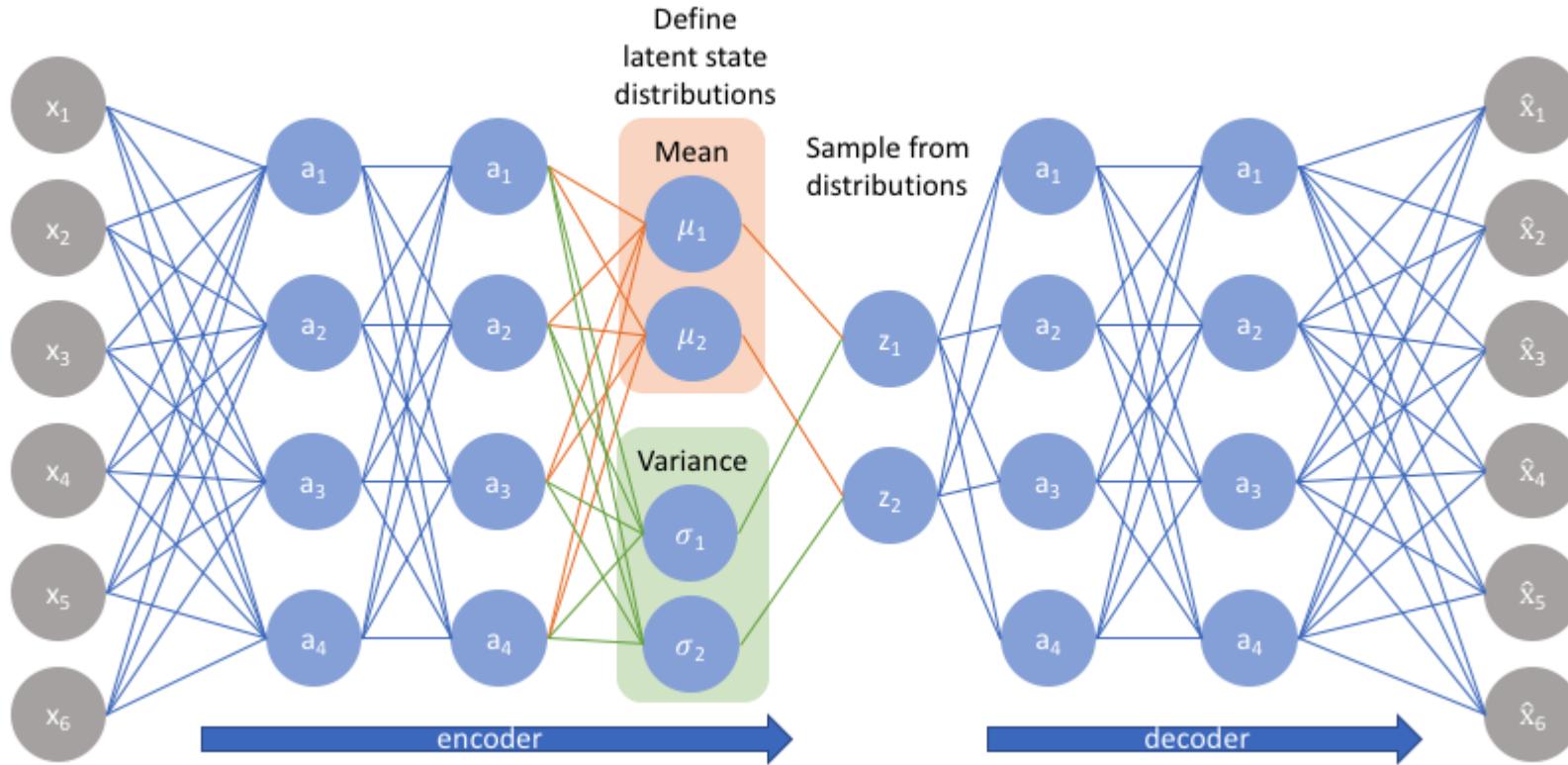
Kullback-Liebler divergence

- omezuje bottleneck příznaky na gaussovské (normální) rozložení
- kromě rekonstrukční odchylky navíc Kullback-Liebler divergence jako kritérium na bottleneck

$$D_{\text{KL}}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

... měří podobnost, na kolik je bottleneck ( $Q$ ) gaussovský ( $P$ )

# Implementace variačního autoenkodéru

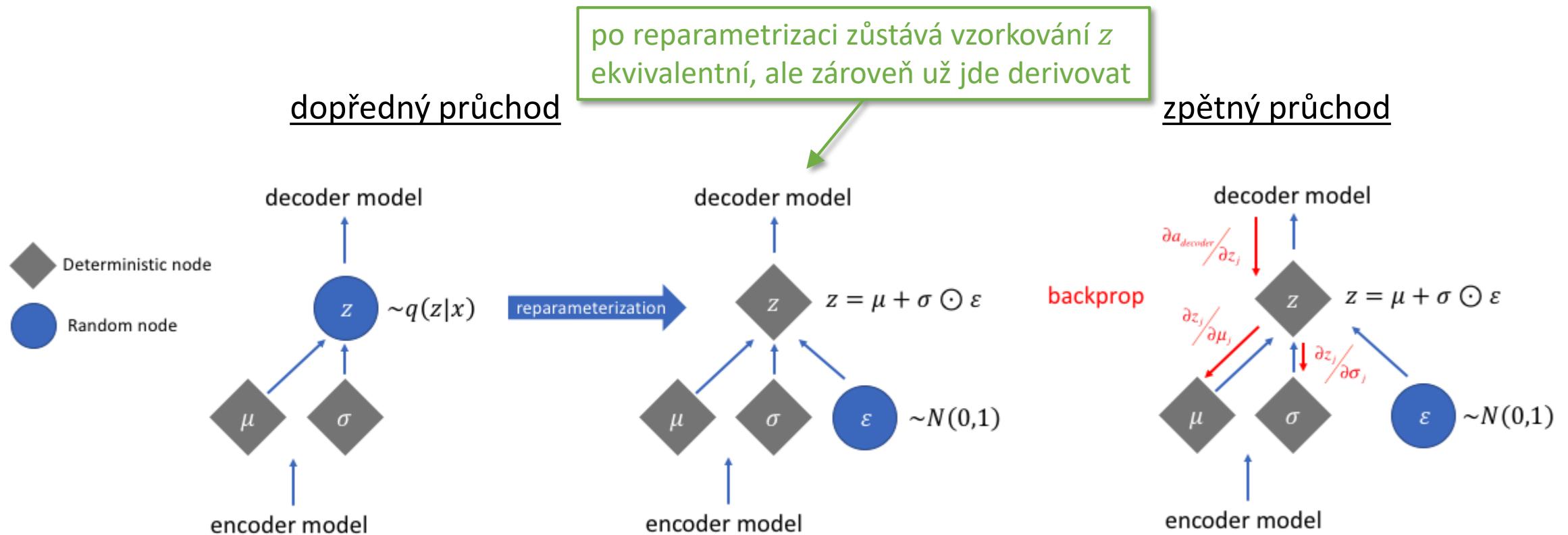


- KL nelze v praxi efektivně vyhodnotit, je to střední hodnota (integrál) přes všechny možné hodnoty
- namísto sítí vygeneruje dva vektory průměru  $\mu$  a rozptylu  $\sigma^2$  gaussovského rozložení
- pak se využije následujícího:

$$-\mathcal{D}_{KL}(q_\phi(z|x)||p_\theta(z)) = \frac{1}{2} \sum (1 + \log(\sigma^2) - \mu^2 - \sigma^2)$$

obrázek: <https://www.jeremyjordan.me/variational-autoencoders/>

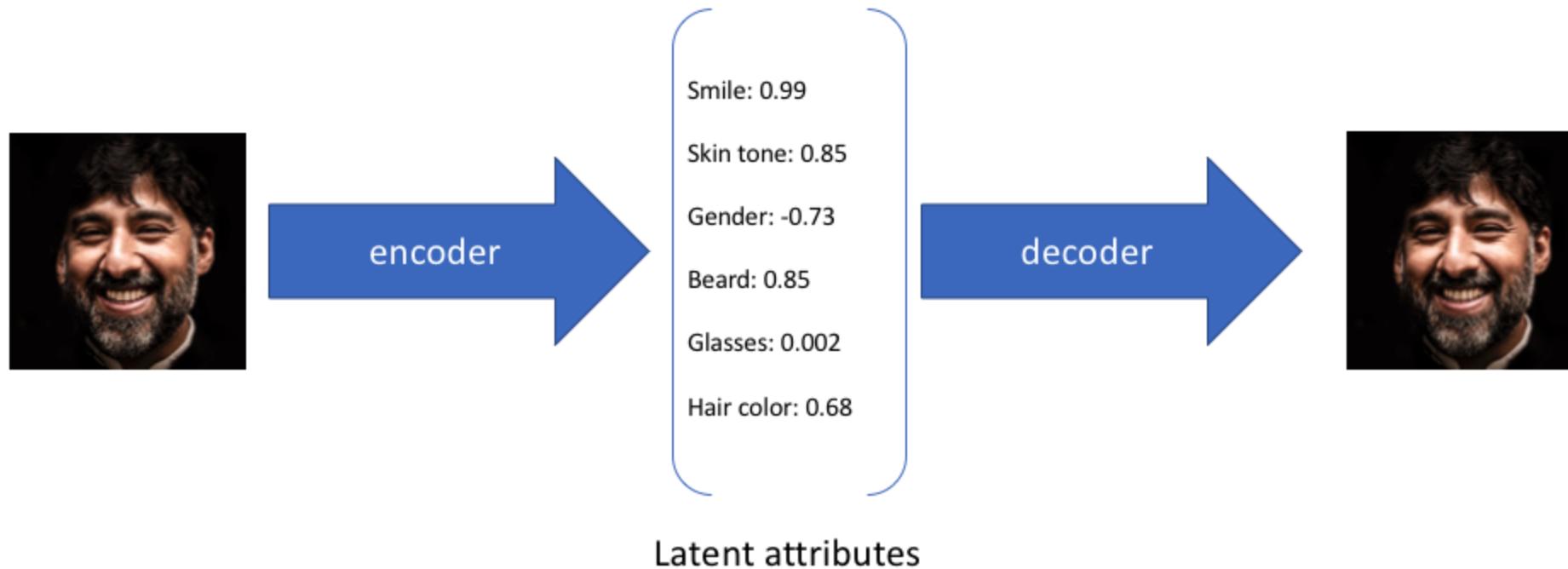
# Reparametizační trik



- vzorkuje se vždy ze standardního  $N(0, 1)$  gaussovského rozložení s nulovým průměrem a jednotkovým rozptylem
- průměr a rozptyl se řeší affinní transformací (škálování = std. odchylka, posun = průměr) vzorkovaných dat

obrázek: <https://www.jeremyjordan.me/variational-autoencoders/>

# Interpretace latentních “bottleneck” parametrů

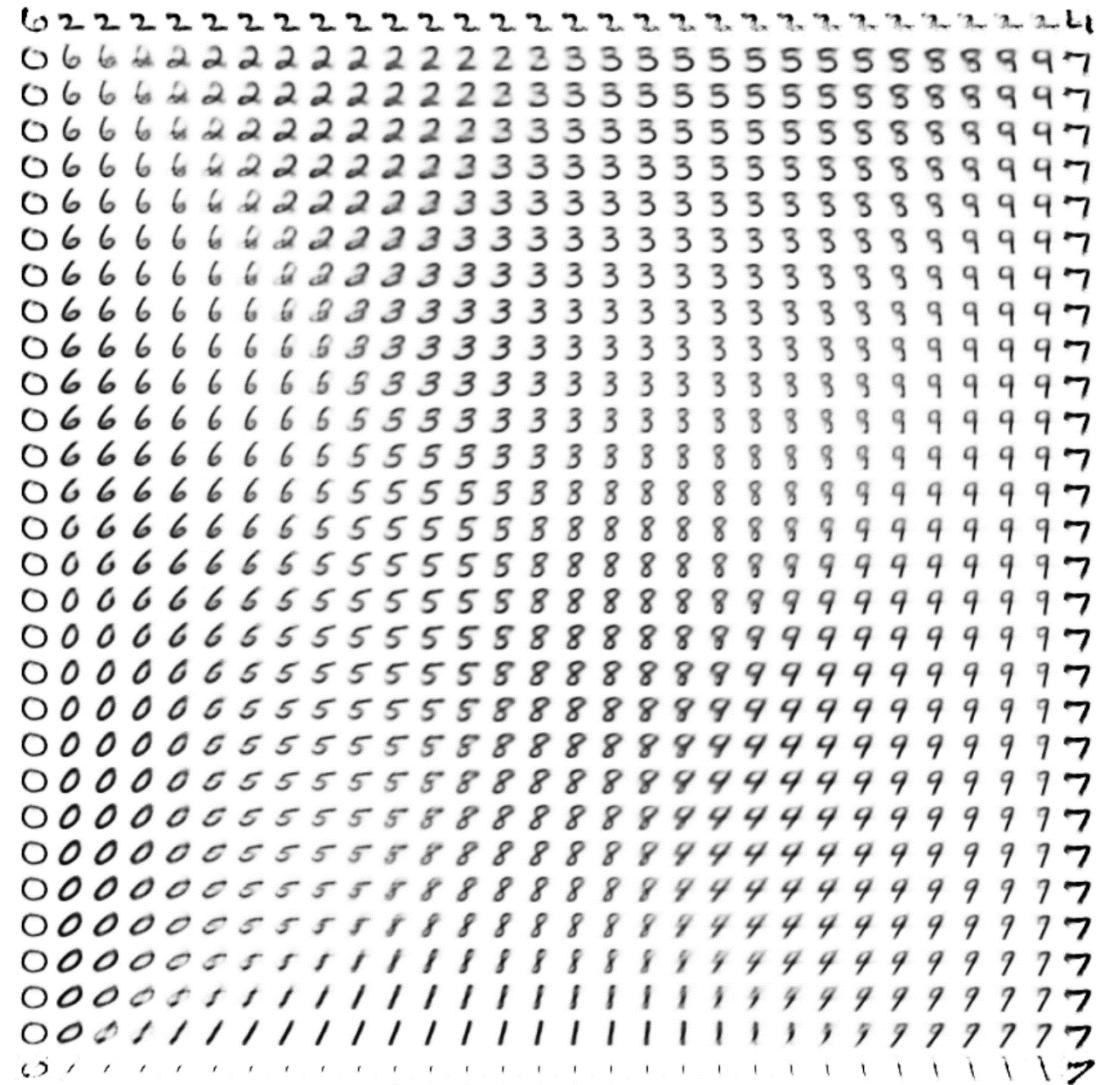


- na bottleneck vektor lze nahlížet jako na skryté parametry
- samplováním parametrů z normálního rozložení lze inicializovat dekodér a generovat různé obrázky
- variační autoenkovdér tedy patří mezi generativní modely

obrázek: <https://www.jeremyjordan.me/variational-autoencoders/>

# Skryté parametry variačního autoenkodéru

- na obrázku příklad dvourozměrného normálního rozložení (dva skryté parametry)
- pohybem v tomto 2D prostoru generujeme různé MNIST číslice



obrázek: <http://blog.fastforwardlabs.com/2016/08/22/under-the-hood-of-the-variational-autoencoder-in.html>

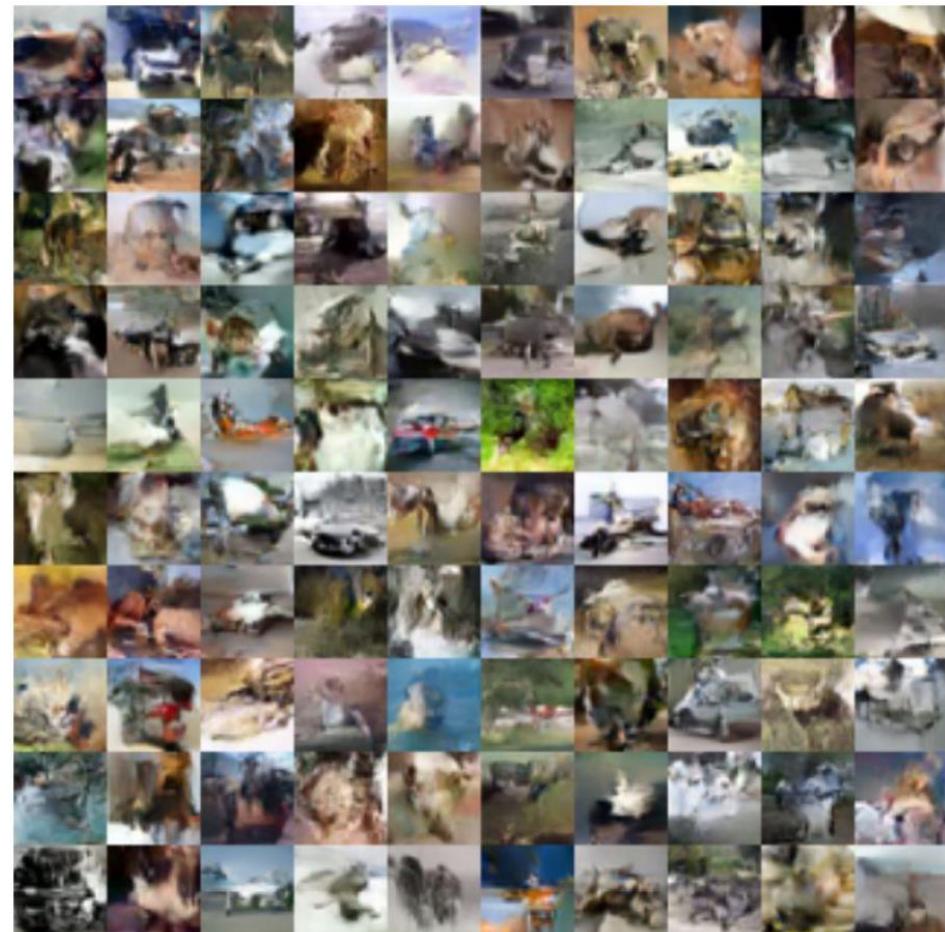
# Skryté parametry variačního autoenkodéru



# Obrázky vygenerované variačním autoenkodérem



Labeled faces in the wild

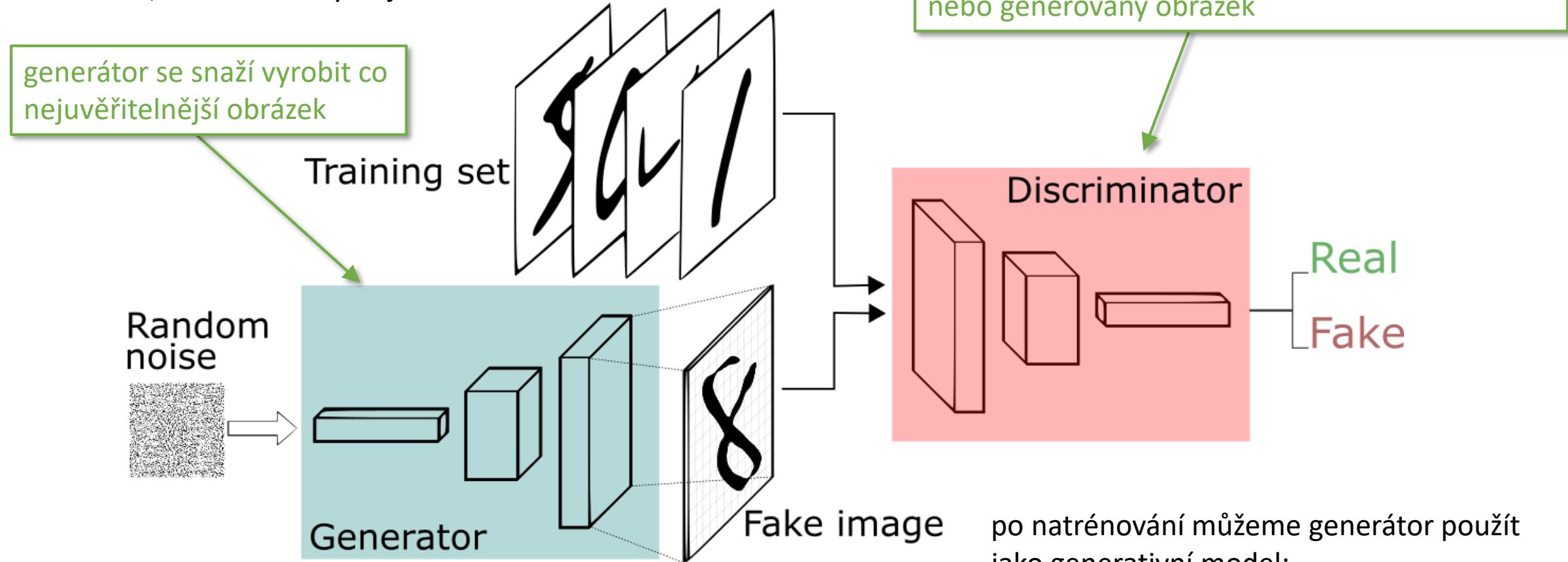


CIFAR-10

obrázek: <https://jaan.io/what-is-variational-autoencoder-vae-tutorial/>

# Generative adversarial network (GAN)

- [Goodfellow et al.: Generative Adversarial Networks](#)
- učení dvou sítí zároveň: generátor vs diskriminátor
- "soutěž", která z nich vyhraje



obrázek: <https://deeplearning4j.org/generative-adversarial-network>

# Kritérium generativní adversarial sítě

celkový loss je

nezávislé na generátoru →  
generátor lze updatovat oddeleně

$$L_{GAN}(G, D) = \boxed{\mathbb{E}_{x \sim p_{data}(x)} [\log D(x)]} + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

$D(x)$  ... výstup diskriminátoru

$G(z)$  ... výstup generátoru

rozdíl oproti klasické optimizaci:

- loss  $L_{GAN}(G, D)$  maximalizujeme přes diskriminátor tak, aby  $D(x)$  bylo 1 pro reálné obrázky, 0 pro fake
- generátor se naopak snaží, aby  $D(G(z))$  bylo 1 pro falešné obrázky → přes generátor minimalizujeme

# Trénování generativní adversarial sítě

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

for number of training iterations do

  for  $k$  steps do

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
- Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(x)$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

update diskriminátoru  
generátor je zafixovaný

  end for

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

gradient na diskriminátor

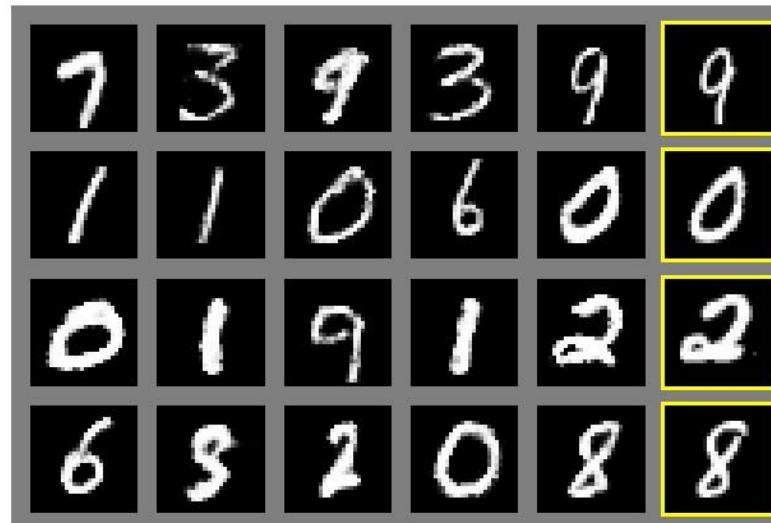
update generátoru  
diskriminátor je zafixovaný

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

# GAN ukázky v původním článku

MNIST



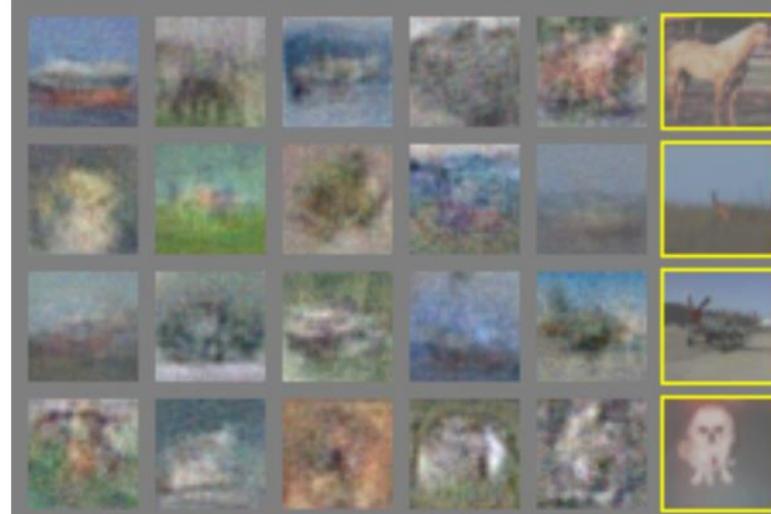
a)

TFD



b)

CIFAR



c)  
zdroj: [Goodfellow et al.: Generative Adversarial Networks](#)



d)

# Vektorová aritmetika



smiling  
woman



neutral  
woman



neutral  
man

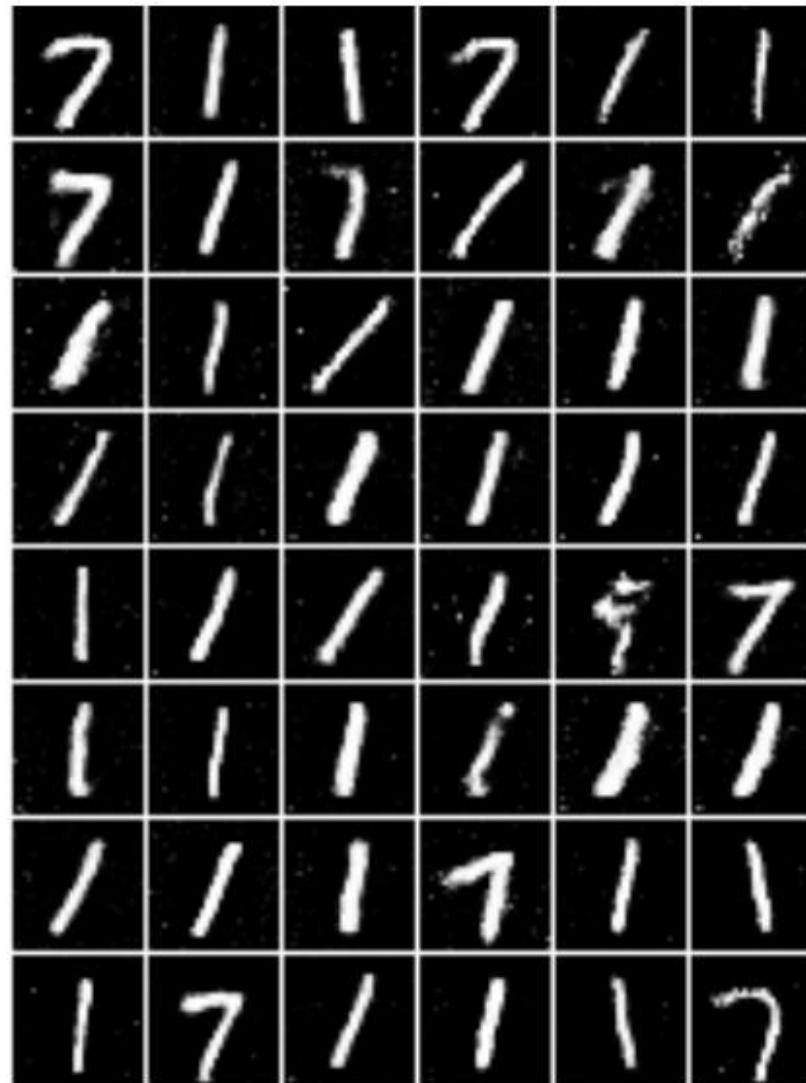


smiling man

Vektor náhodného vstupního šumu se po natrénování chová jako skryté proměnné, které odpovídají reálným interpretovatelným faktorům ovlivňujícím výsledný vzhled (např. pohlaví, vlasy, smích, ...)

# Typické problémy při trénování GAN

- Mizející gradient (vanishing gradients)
  - Pokud diskriminátor je příliš dobrý
  - Pravděpodobnosti  $D(G(z))$  jsou pak příliš blízko 0 nebo 1 → viz sigmoid gradient
- Mode collapse
  - Některé např. číslice se generují snáze
  - Generátor to zjistí a jiné přestane produkovat
  - Diskriminátor se chytne do lokálního minima



obrázek: [Wu et al.: Deep Compressed Sensing](#)

# Wasserstein GAN (2017)

- [Arjovsky et al: Wasserstein GAN](#)
- Namísto mini/max  $D(z)$  a  $D(G(z))$  optimalizuje
$$\mathbb{E}_x[D(x)] - \mathbb{E}_z[D(G(z))]$$
- Maximalizace rozdílu skóre pro reálná vs fake data
- Výstup není pravděpodobnost, ale skóre  $(-\infty, +\infty)$

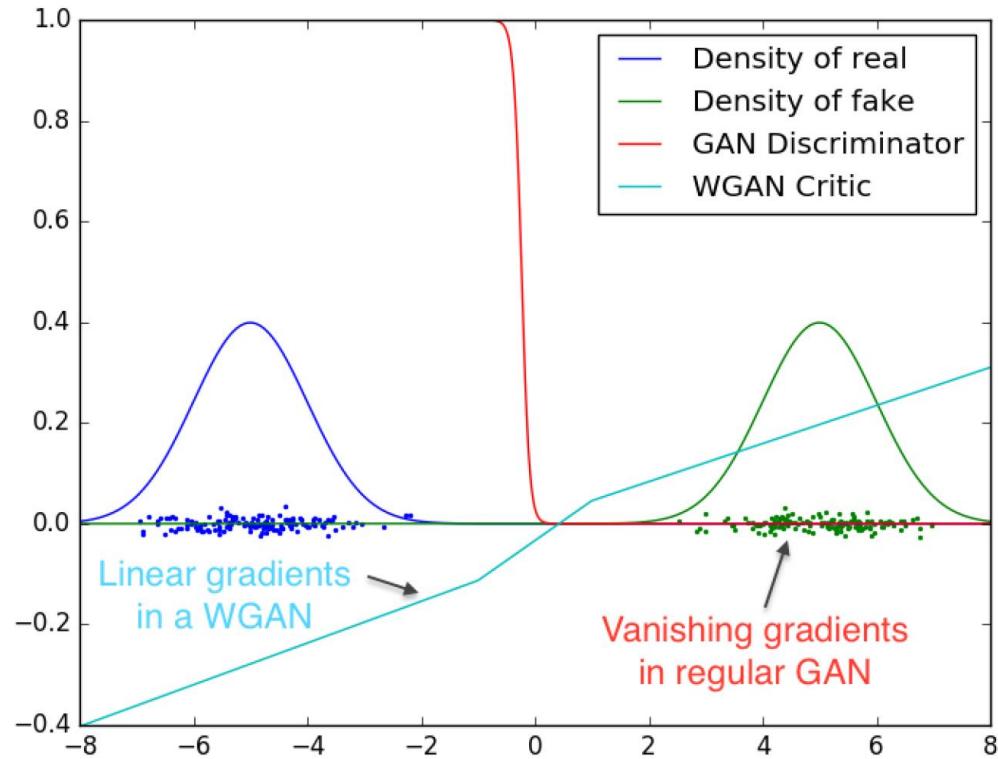


Figure 2: Optimal discriminator and critic when learning to differentiate two Gaussians. As we can see, the discriminator of a minimax GAN saturates and results in vanishing gradients. Our WGAN critic provides very clean gradients on all parts of the space.

# pix2pix (2016)

- Isola et al: Image-to-Image Translation with Conditional Adversarial Networks
- Unifikovaný framework pro úlohy typu *výstupní\_obrázek = funkce(vstupní\_obrázek)*
- **Není unsupervised**: dataset jsou páry obrázků (vstup, výstup)

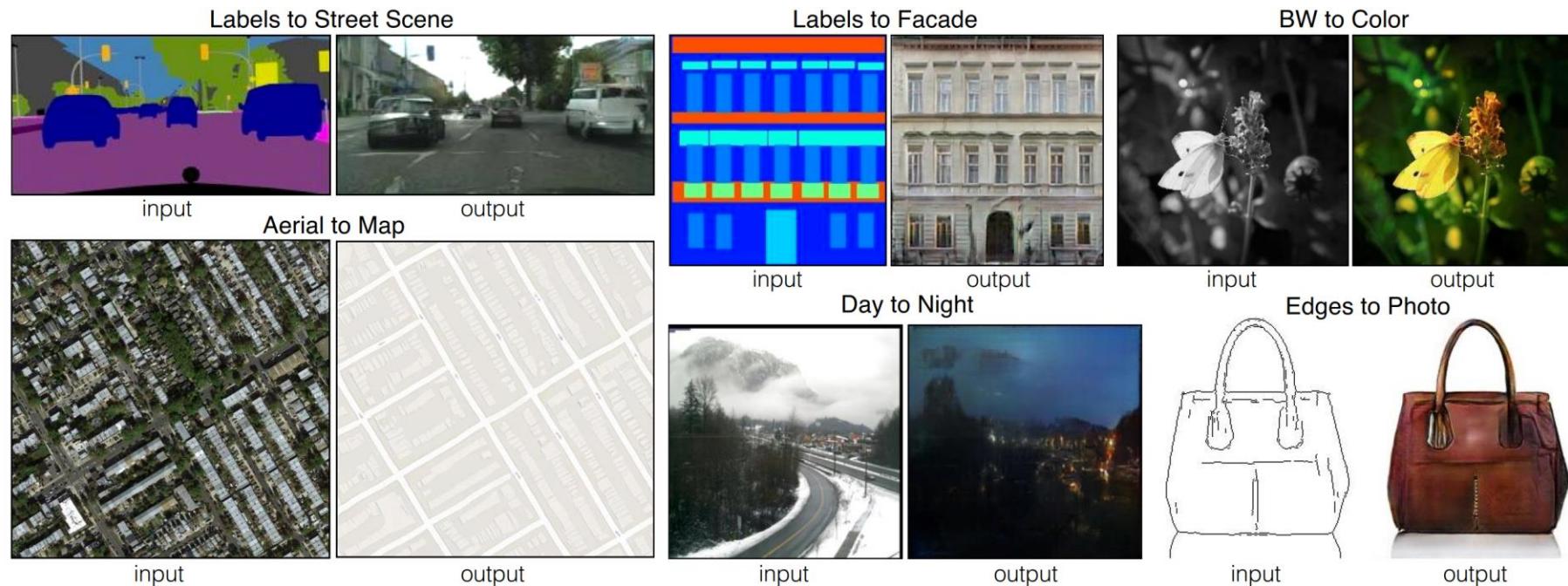
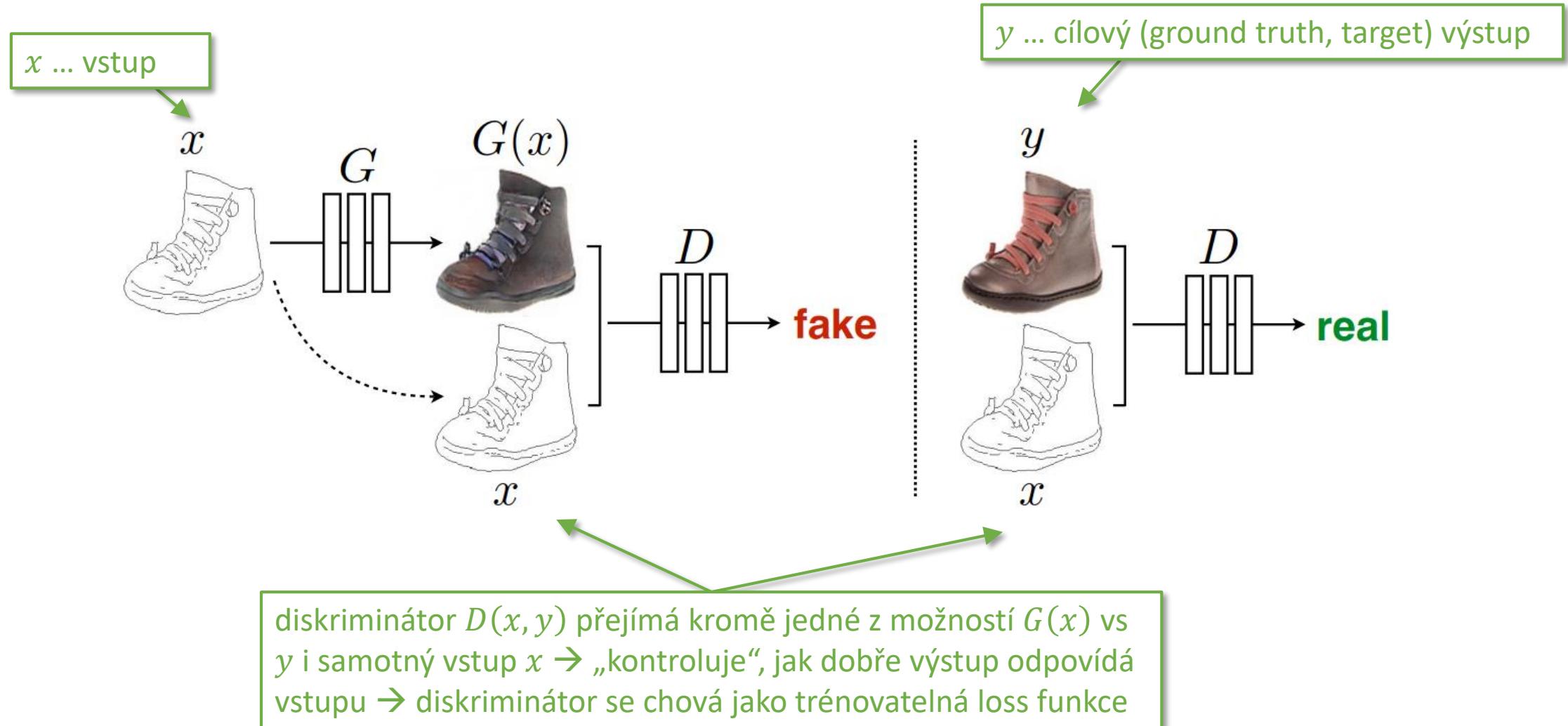


Figure 1: Many problems in image processing, graphics, and vision involve translating an input image into a corresponding output image. These problems are often treated with application-specific algorithms, even though the setting is always the same: map pixels to pixels. Conditional adversarial nets are a general-purpose solution that appears to work well on a wide variety of these problems. Here we show results of the method on several. In each case we use the same architecture and objective, and simply train on different data.

# pix2pix (2016)



# pix2pix (2016)

- Conditional GAN (cGAN)

- generátor i diskriminátor jsou podmíněny („vidí“) vstupem  $x$ , které není  $\sim \mathcal{N}(0,1)$
- nepodmíněná GAN: generátor generuje náhodně (vstupem je random  $z$ ), diskriminátor vidí jen  $G(z)$

- GAN kritérium pix2pix je

$$L_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z} \left[ \log \left( 1 - D(x, G(x, z)) \right) \right]$$

- Zároveň ale chce, aby výstup generátoru byl co nejblíže ground truth  $y$ , tj.

$$L_{L1}(G) = \mathbb{E}_{x,z}[\|y - G(x, z)\|_1]$$

- Celková loss funkce tedy je

$$G^* = \arg \min_G \max_D L_{cGAN}(G, D) + \lambda L_{L1}(G)$$

# pix2pix generátor

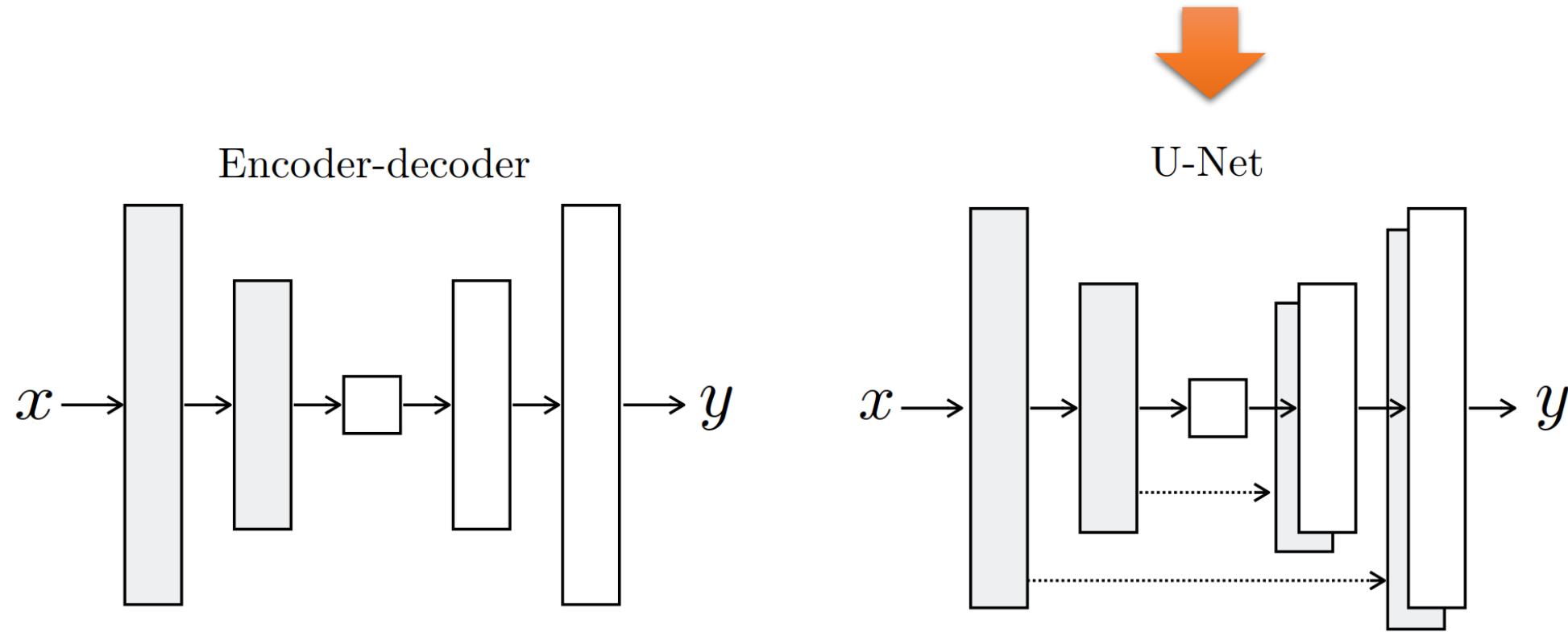
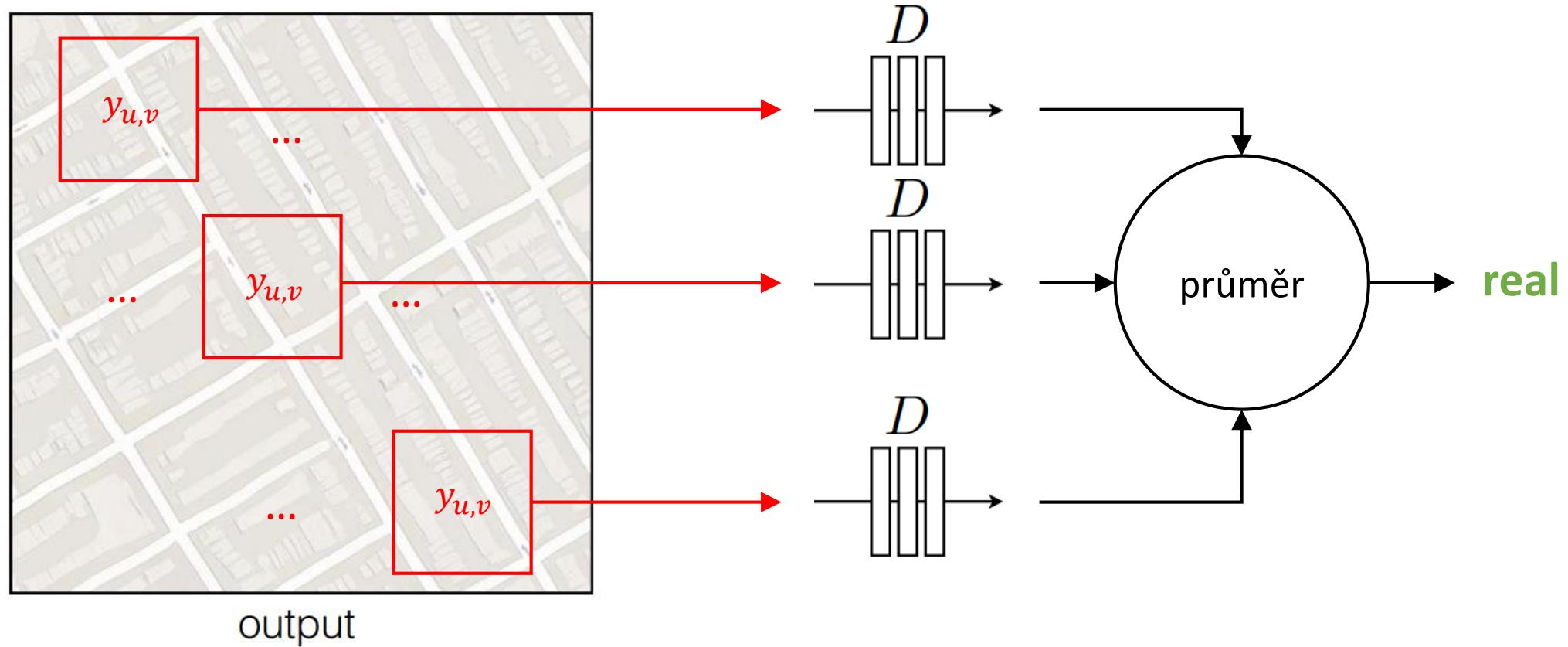


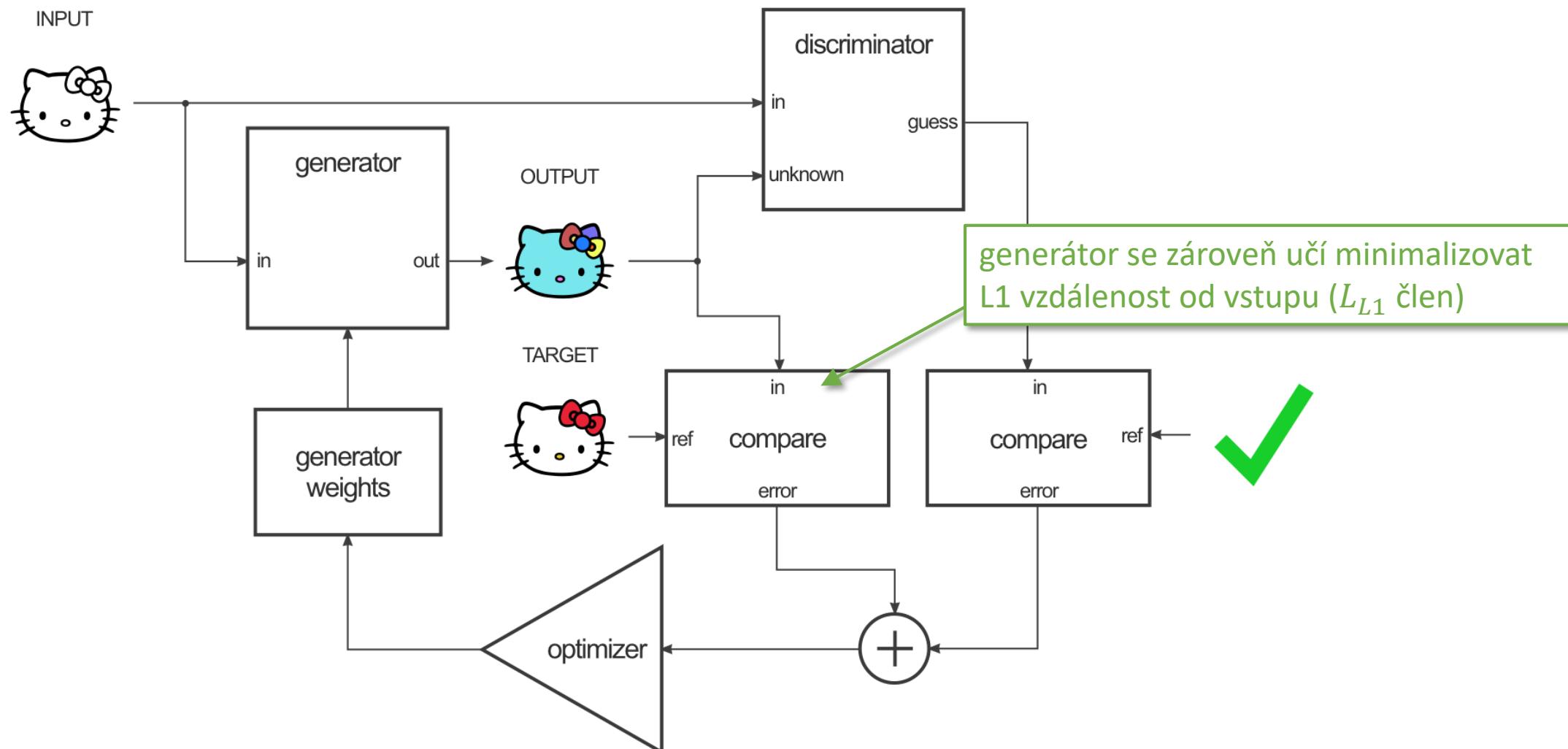
Figure 3: Two choices for the architecture of the generator. The “U-Net” [50] is an encoder-decoder with skip connections between mirrored layers in the encoder and decoder stacks.

# pix2pix diskriminátor

PatchGAN ... diskriminátor (konvoluční síť) namísto celého obrázku klasifikuje pouze jednotlivé patche a pak průměruje

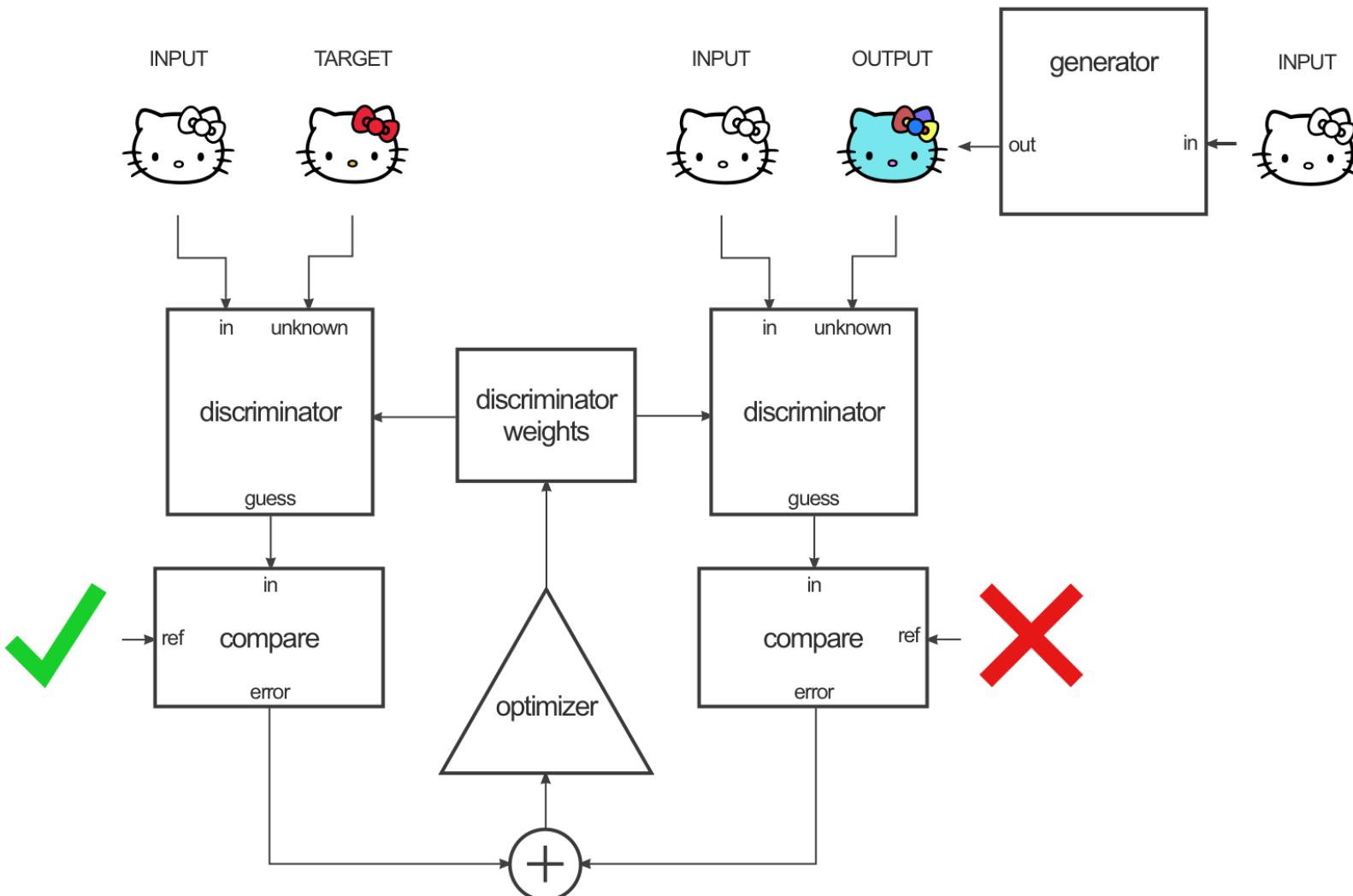


# pix2pix trénování generátoru



obrázek: <https://affinelayer.com/pix2pix/>

# pix2pix trénování diskriminátoru



obrázek: <https://affinelayer.com/pix2pix/>

# pix2pix (2016)



Figure 8: Example results on Google Maps at  $512 \times 512$  resolution (model was trained on images at  $256 \times 256$  resolution, and run convolutionally on the larger images at test time). Contrast adjusted for clarity.

# pix2pix (2016)

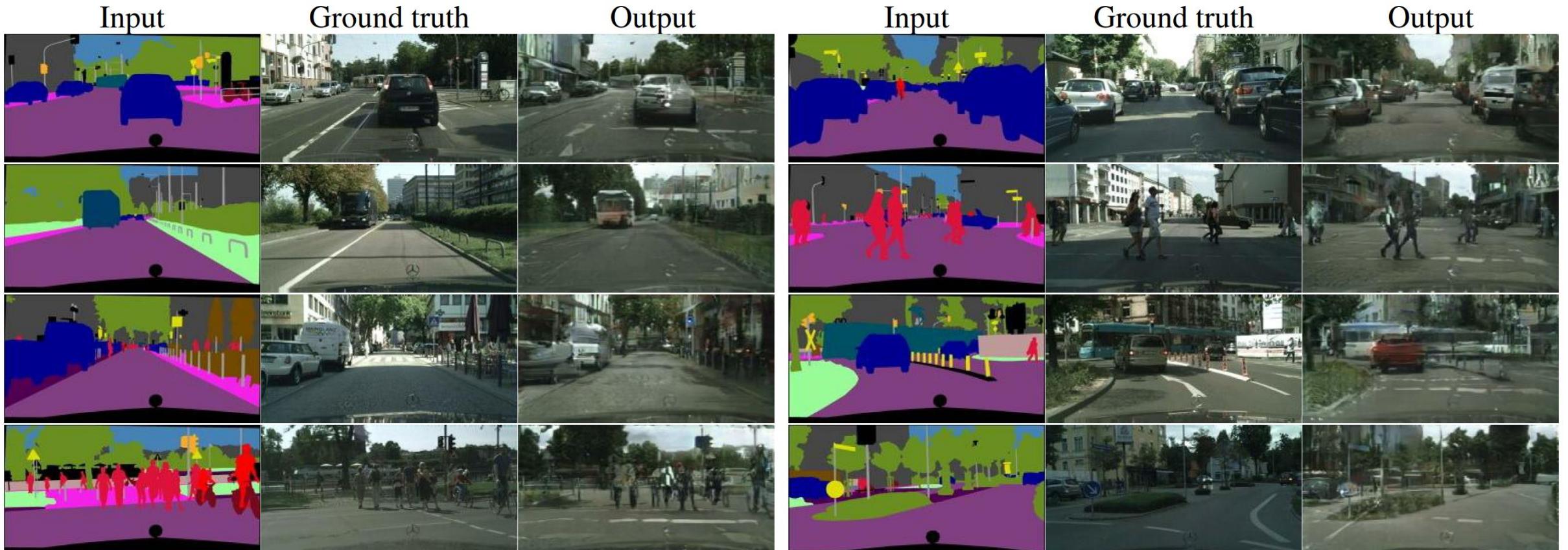


Figure 13: Example results of our method on Cityscapes  $\text{labels} \rightarrow \text{photo}$ , compared to ground truth.

# pix2pix (2016)



Figure 15: Example results of our method on day→night, compared to ground truth.

# pix2pix (2016)



Figure 16: Example results of our method on automatically detected edges→handbags, compared to ground truth.

# pix2pix (2016)



Figure 17: Example results of our method on automatically detected  $\text{edges} \rightarrow \text{shoes}$ , compared to ground truth.

# pix2pix (2016)

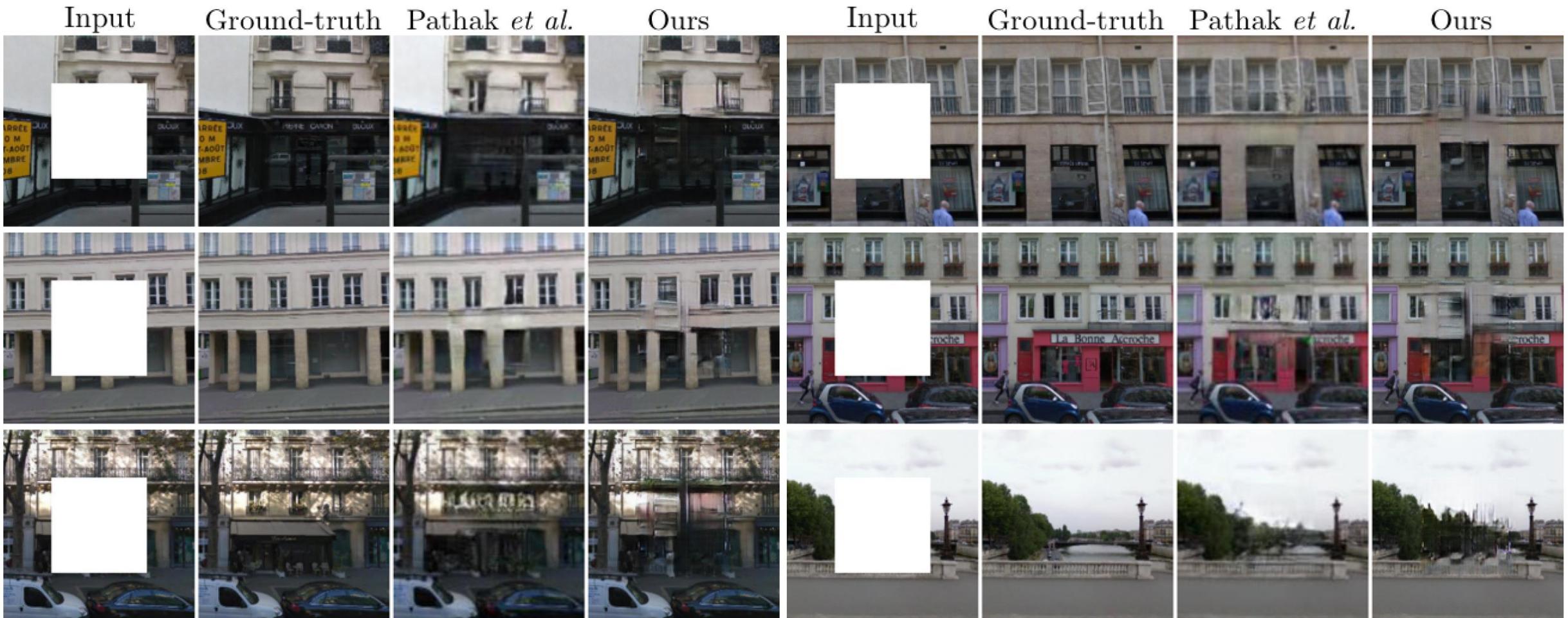


Figure 19: Example results on photo **inpainting**, compared to [43], on the Paris StreetView dataset [14]. This experiment demonstrates that the U-net architecture can be effective even when the predicted pixels are not geometrically aligned with the information in the input – the information used to fill in the central hole has to be found in the periphery of these photos.

# pix2pix (2016)

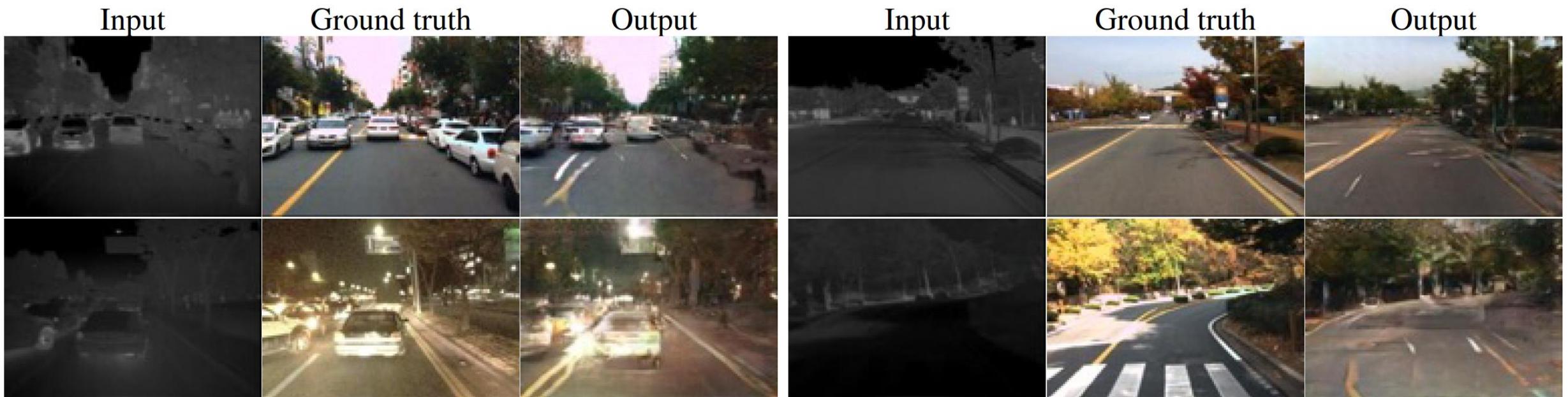
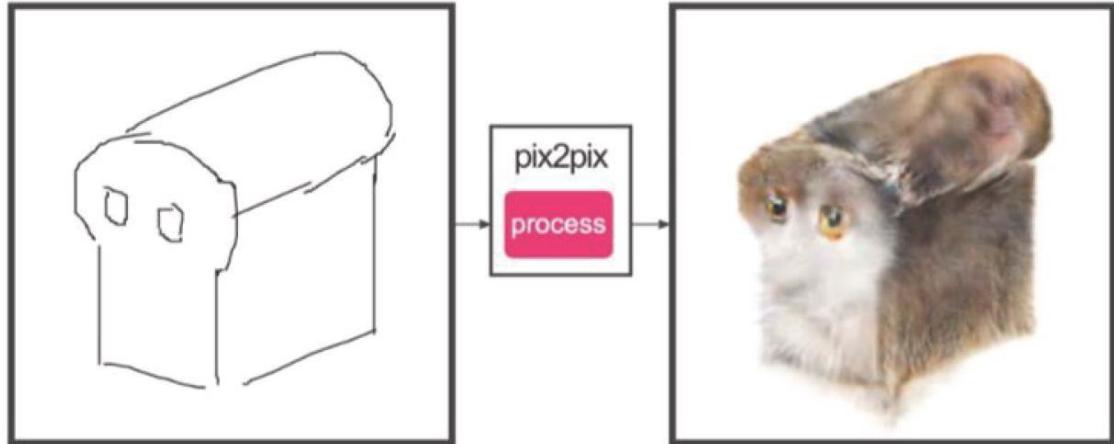


Figure 20: Example results on translating thermal images to RGB photos, on the dataset from [27].

# pix2pix (2016)

#edges2cats by Christopher Hesse



Background removal



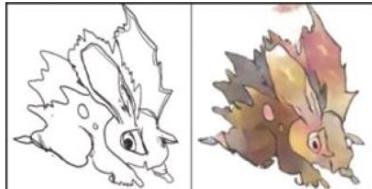
Palette generation



Sketch→Portrait



Sketch → Pokemon



“Do as I do”



#fotogenerator

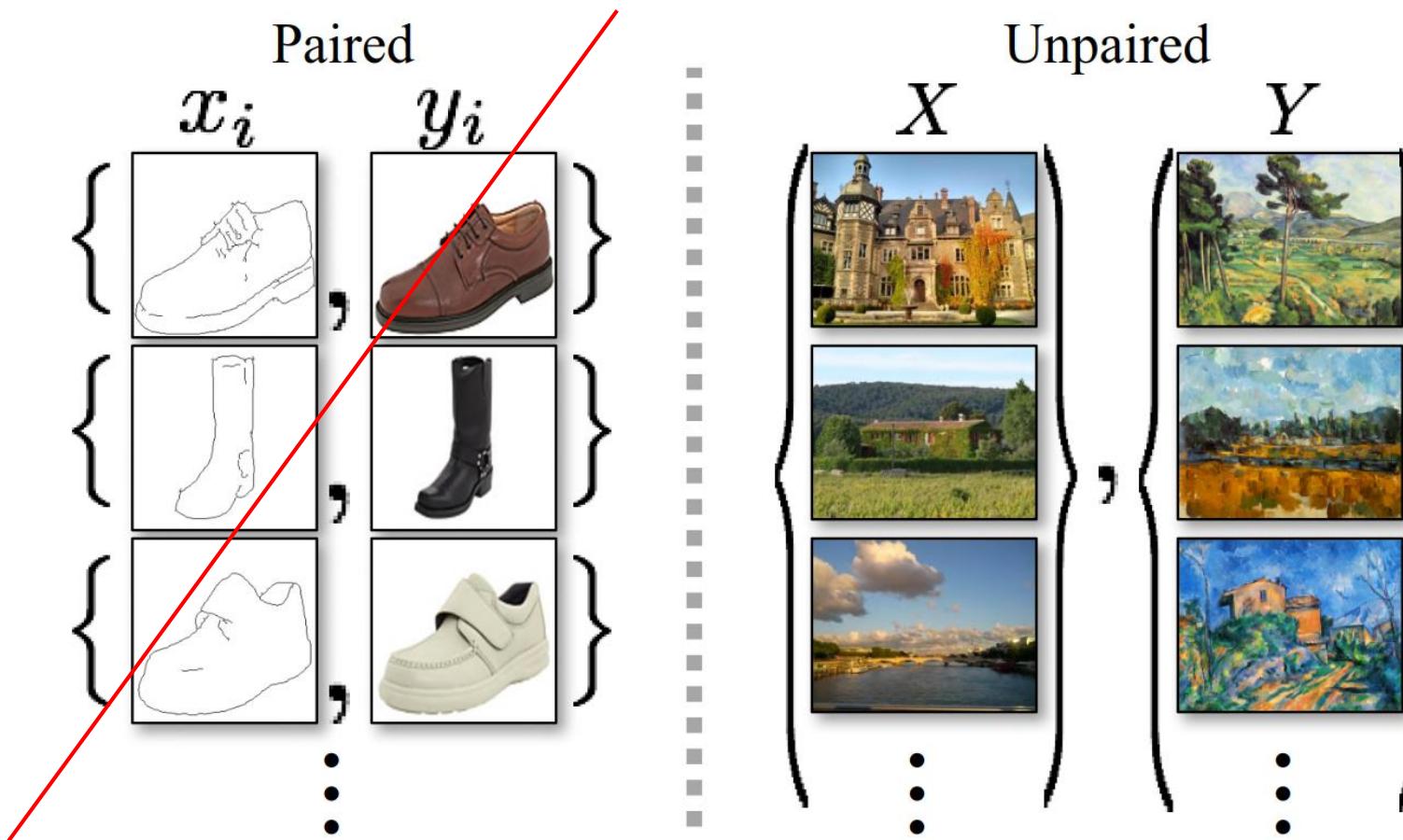


Figure 11: Example applications developed by online community based on our pix2pix codebase: #edges2cats [3] by Christopher Hesse, Background removal [6] by Kaihu Chen, Palette generation [5] by Jack Qiao, Sketch → Portrait [7] by Mario Klingemann, Sketch→Pokemon [1] by Bertrand Gondouin, “Do As I Do” pose transfer [2] by Brannon Dorsey, and #fotogenerator by Bosman et al. [4].

online demo: <https://affinelayer.com/pixsrv/>

# CycleGAN (2017)

- Zhu et al.: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- Nepotřebuje dataset párů (vstup, výstup), stačí nespárované kolekce
- Např. sada letních obrázků a druhá sada zimních → **unsupervised\***



# CycleGAN (2017)

- Obsahuje 2 generátory  $G, F$  a 2 diskriminátory  $D_X, D_Y$
- Vstup se převádí z domény  $X$  do domény  $Y$  a **zároveň i zpět** → cyklický loss
  - Ve výstupu generátoru musí zůstat informace vstupu a nemůže se naučit generovat pouze náhodně

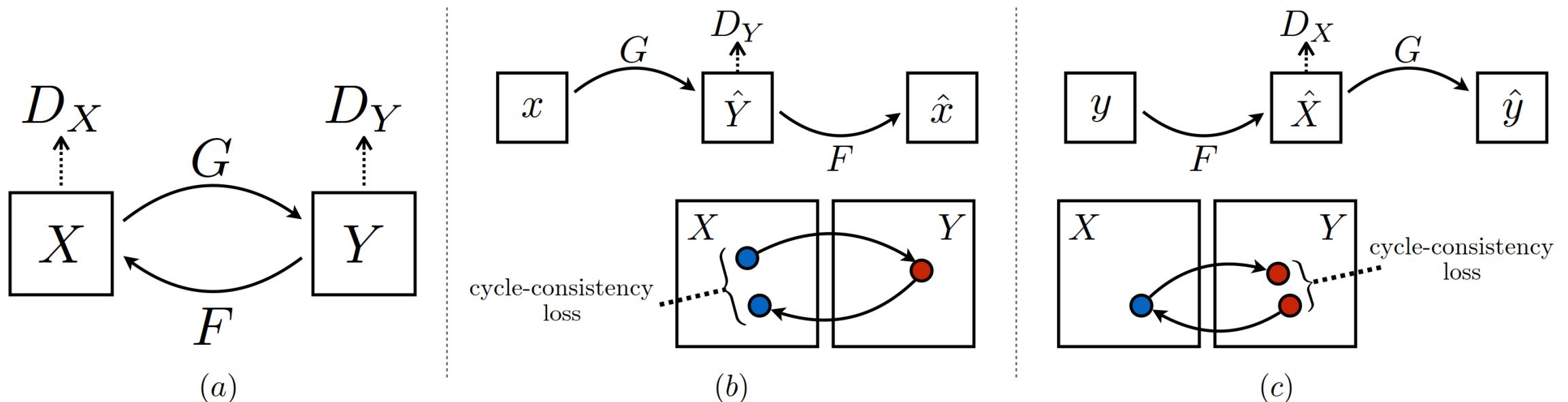


Figure 3: (a) Our model contains two mapping functions  $G : X \rightarrow Y$  and  $F : Y \rightarrow X$ , and associated adversarial discriminators  $D_Y$  and  $D_X$ .  $D_Y$  encourages  $G$  to translate  $X$  into outputs indistinguishable from domain  $Y$ , and vice versa for  $D_X$  and  $F$ . To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss:  $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$ , and (c) backward cycle-consistency loss:  $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

# CycleGAN (2017)

- GAN loss

$$L_{GAN}(G, D_Y) = \mathbb{E}_{y \sim Y} [\log D_Y(y)] + \mathbb{E}_{x \sim X} \left[ \log \left( 1 - D_Y(G(x)) \right) \right] \quad \text{← foto} \rightarrow \text{malba}$$

$$L_{GAN}(F, D_X) = \mathbb{E}_{x \sim X} [\log D_X(x)] + \mathbb{E}_{y \sim Y} \left[ \log \left( 1 - D_X(F(y)) \right) \right] \quad \text{← malba} \rightarrow \text{foto}$$

- Cyklický loss

$$L_{cyc}(G, F) = \underbrace{\mathbb{E}_{x \sim X} \left[ \|F(G(x)) - x\|_1 \right]}_{\text{true foto vs rekonstr. foto}} + \underbrace{\mathbb{E}_{y \sim Y} \left[ \|G(F(y)) - y\|_1 \right]}_{\text{true malba vs rekonstr. malba}}$$

- Celkový loss

$$\begin{aligned} L_{CycleGAN}(G, F, D_X, D_Y) &= L_{GAN}(G, D_Y) \\ &+ L_{GAN}(F, D_X) \\ &+ L_{cyc}(G, F) \end{aligned}$$

# CycleGAN (2017)

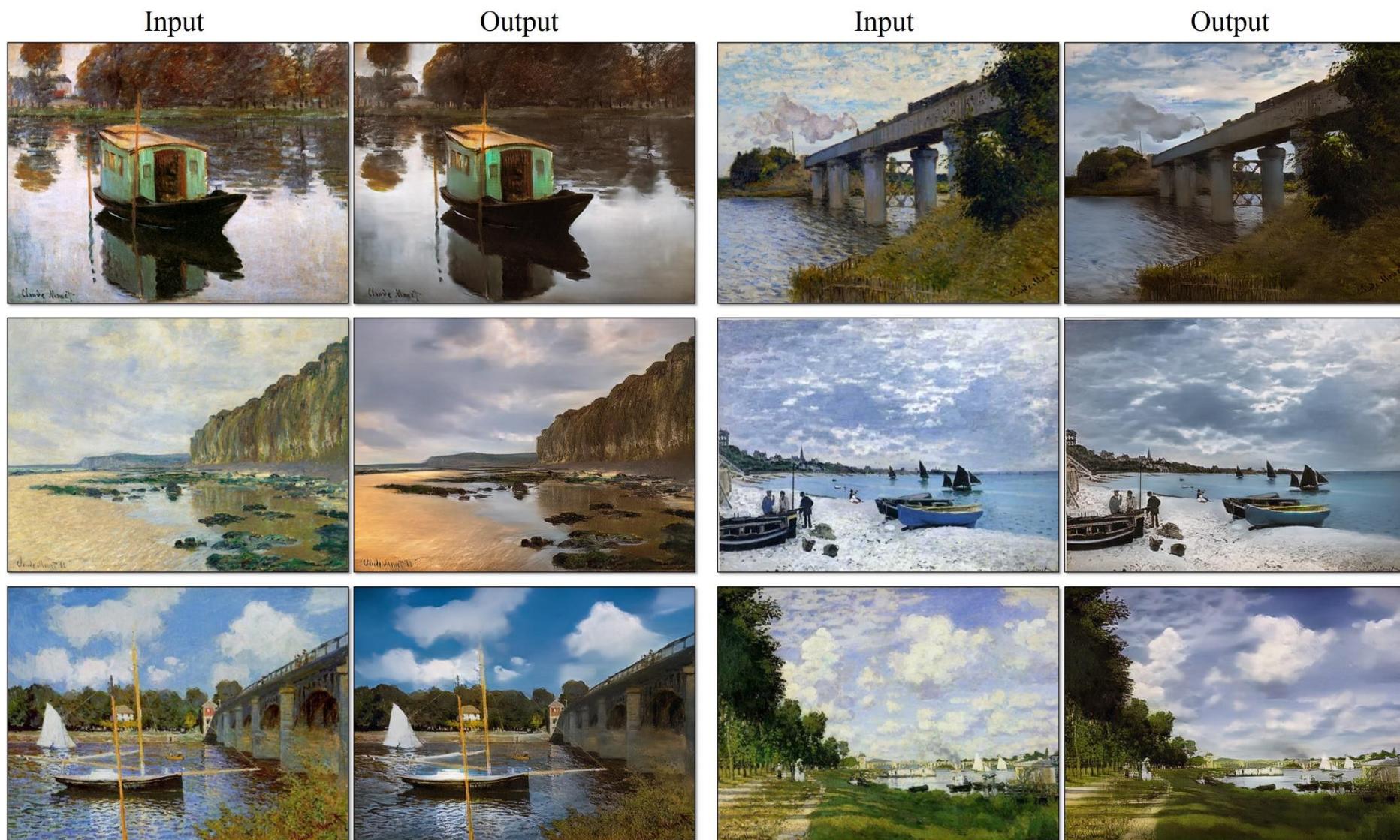


Figure 7: Different variants of our method for mapping labels $\leftrightarrow$ photos trained on cityscapes. From left to right: input, cycle-consistency loss alone, adversarial loss alone, GAN + forward cycle-consistency loss ( $F(G(x)) \approx x$ ), GAN + backward cycle-consistency loss ( $G(F(y)) \approx y$ ), CycleGAN (our full method), and ground truth. Both *Cycle alone* and *GAN + backward* fail to produce images similar to the target domain. *GAN alone* and *GAN + forward* suffer from mode collapse, producing identical label maps regardless of the input photo.

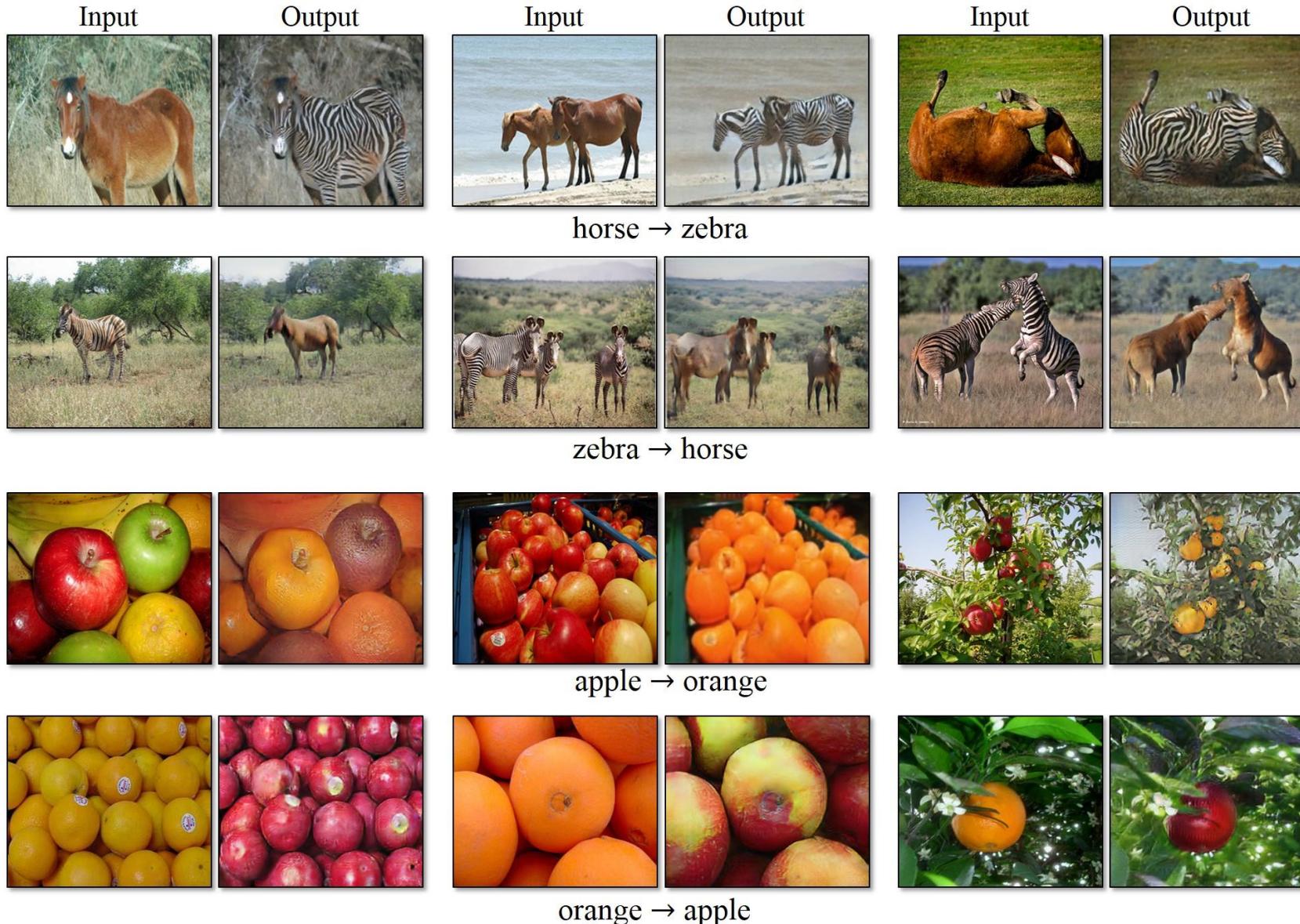
# CycleGAN (2017): Style transfer



# CycleGAN (2017): Monet → fotografie



# CycleGAN (2017): Transmogrifikace



# CycleGAN (2017): Transmogrifikace



<https://junyanz.github.io/CycleGAN/>

# CycleGAN (2017): Když to není dokonalé

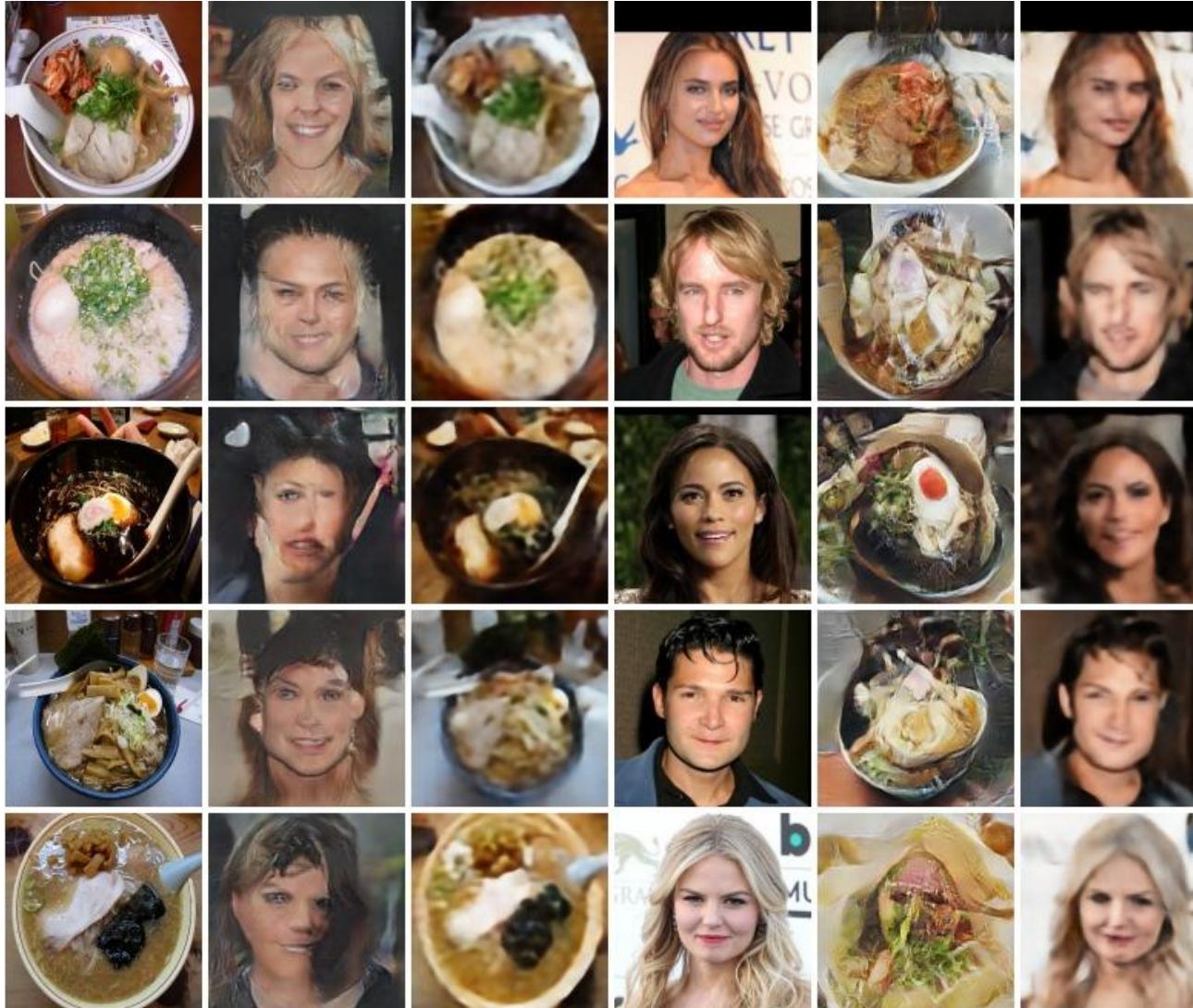


# CycleGAN (2017): Když to není dokonalé



<https://junyanz.github.io/CycleGAN/>

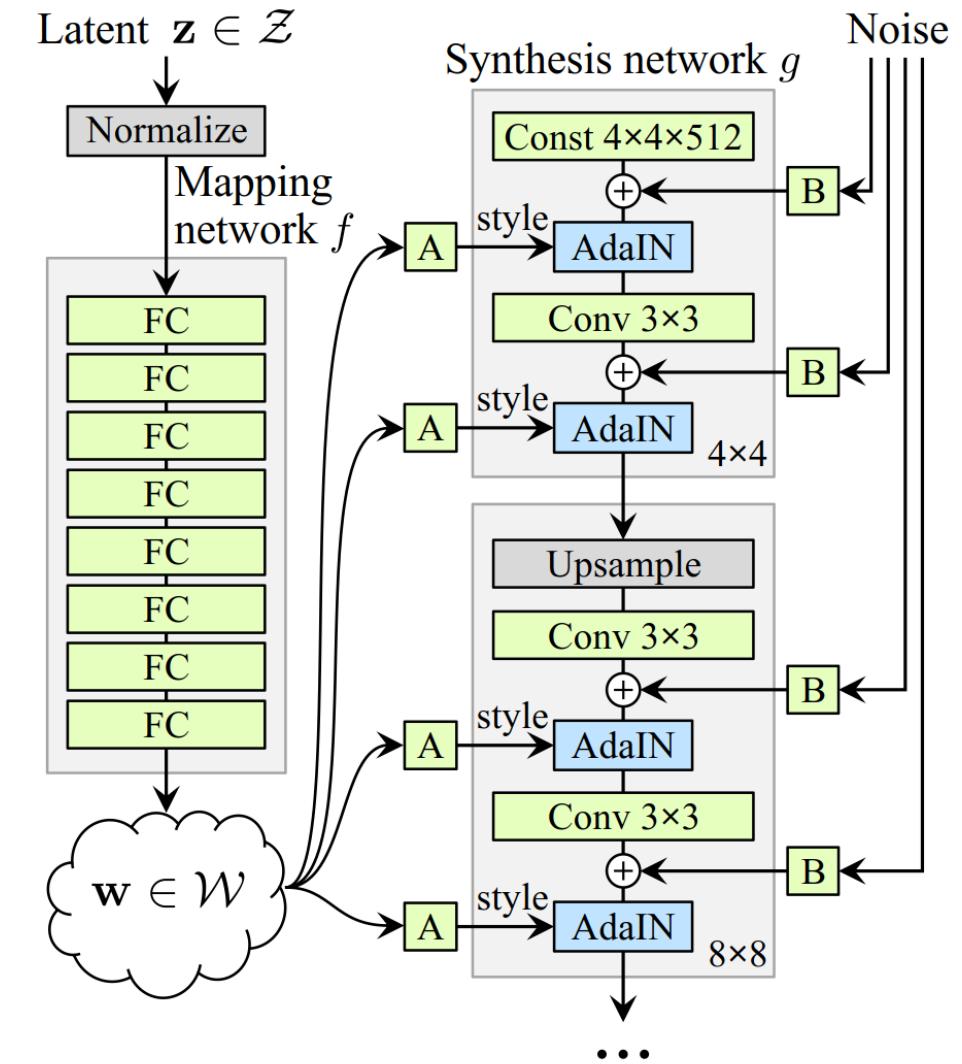
# CycleGAN (2017): Face to Ramen



<https://junyanz.github.io/CycleGAN/>

# StyleGAN (2018)

- [Karras et al.: A Style-Based Generator Architecture for Generative Adversarial Networks](#)
- Unsupervised generování, nepodmíněná GAN
- Mapovací síť  $f$  převádějící prostor  $z$  na „styl“  $w$
- Progresivní zvyšování rozlišení



(b) Style-based generator

# Instance normalization

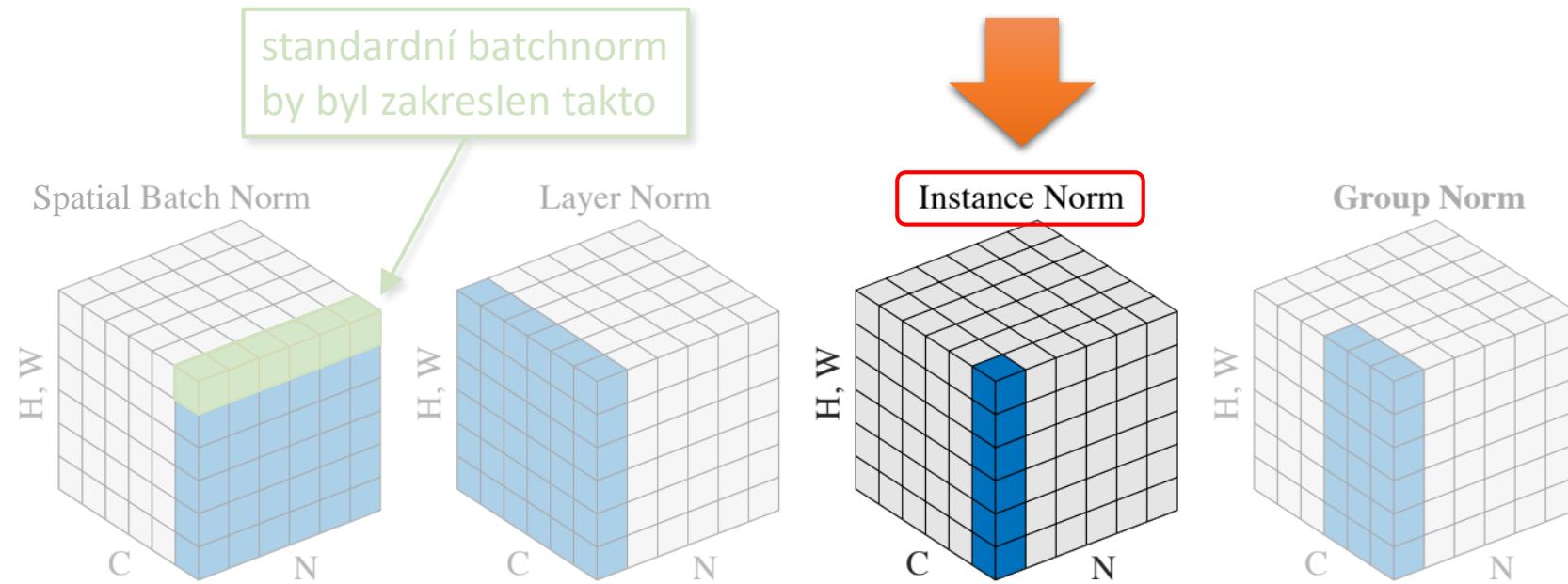


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with  $N$  as the batch axis,  $C$  as the channel axis, and  $(H, W)$  as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

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

# StyleGAN (2018)

- Adaptive instance normalization

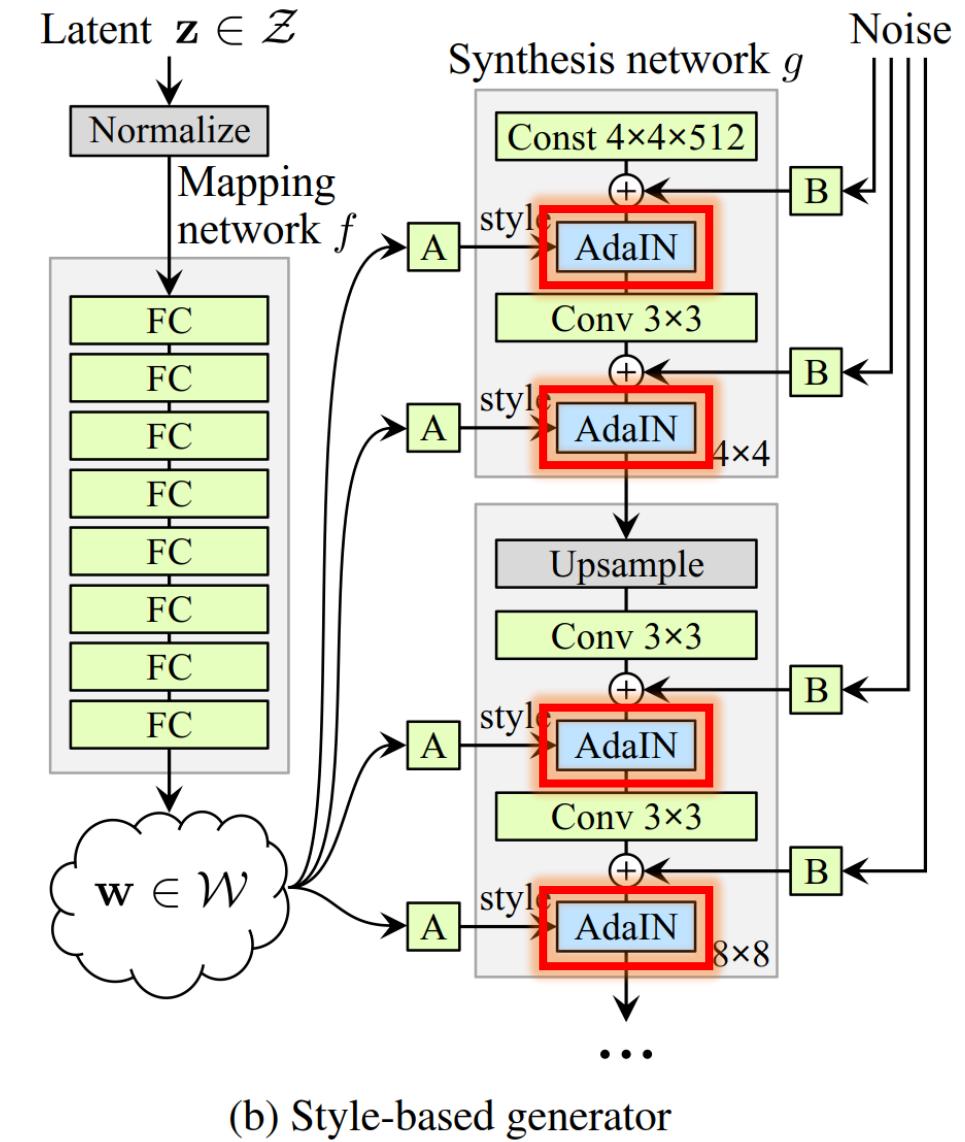
$$\text{AdaIN}(x, y) = y_{s,i} \frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i}$$

$x$  ... feature mapa předchozí vrstvy generátoru

$y$  ... vytvořeno z  $w$  – scale a bias pro každý kanál

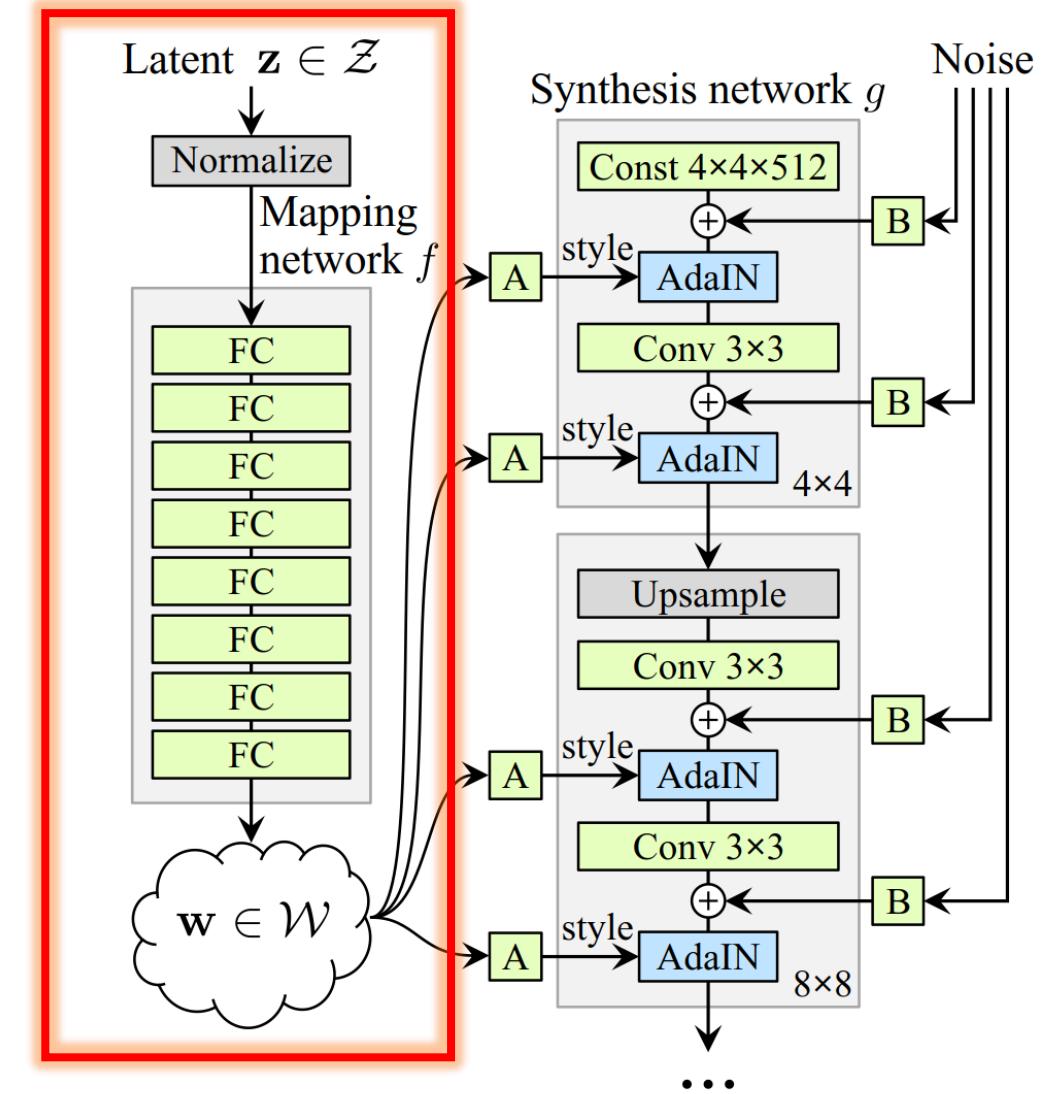
- Funguje jako styl

- V původním článku na style transfer použity gram matice = kovariance kanálů
- Transfer stylu ale funguje i s jinými statistikami, např. průměr a rozptyl (variance)
- Tj. pokud vynormalizujeme kanály feature map z obr. A tak, aby měly stejný průměr a rozptyl jako mapy z jiného obrázku B, přeneseme styl z B do A
- Zde ovšem cílové statistiky nepřenášíme z jiného obrázku, nýbrž generujeme mapovací síť  $y = A(f(z))$



# StyleGAN (2018)

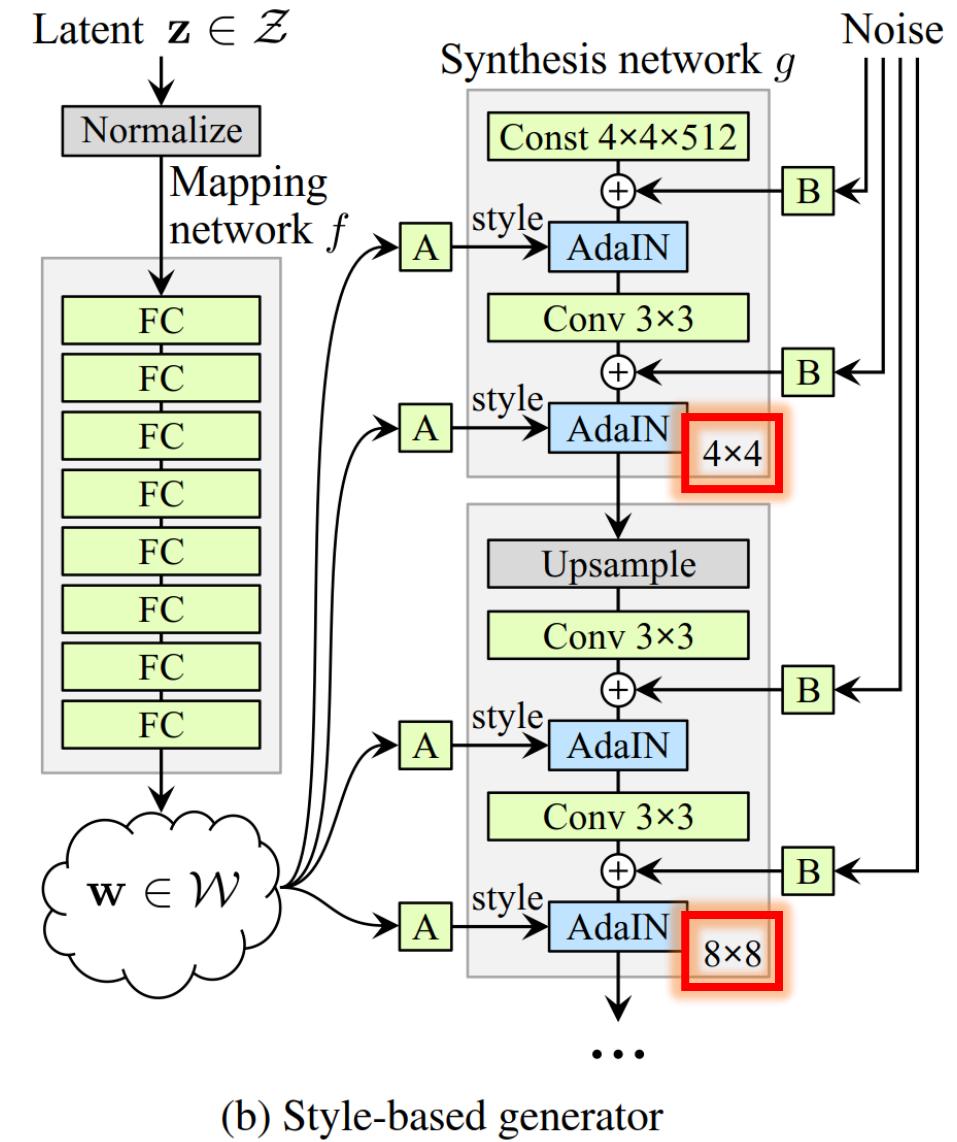
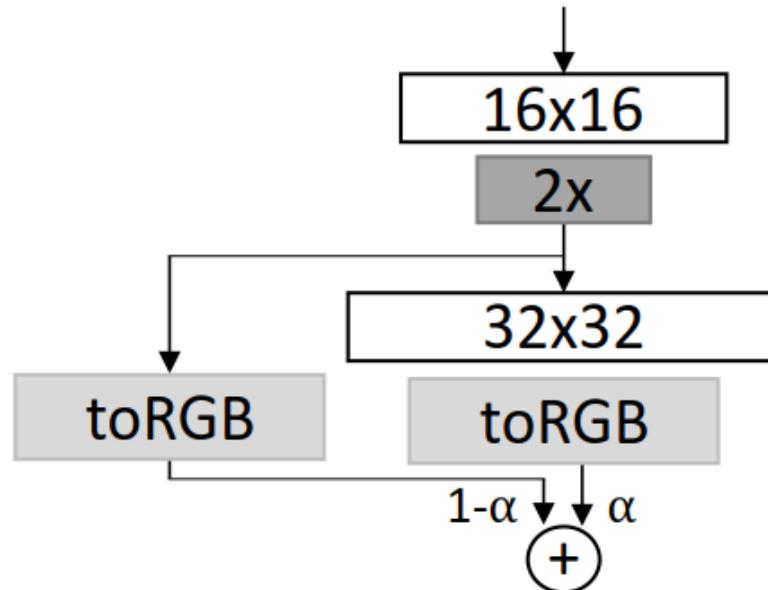
- Latentní proměnné  $z \sim p(z)$  nejsou na začátku generátoru, ale nejprve převedny na mezi-proměnné  $w$
- Ty jsou pak dodány do různých vrstev skrze affiní transformaci (bloky A)  $y = Aw + b$
- Výstup  $y$  pak funguje jako styl



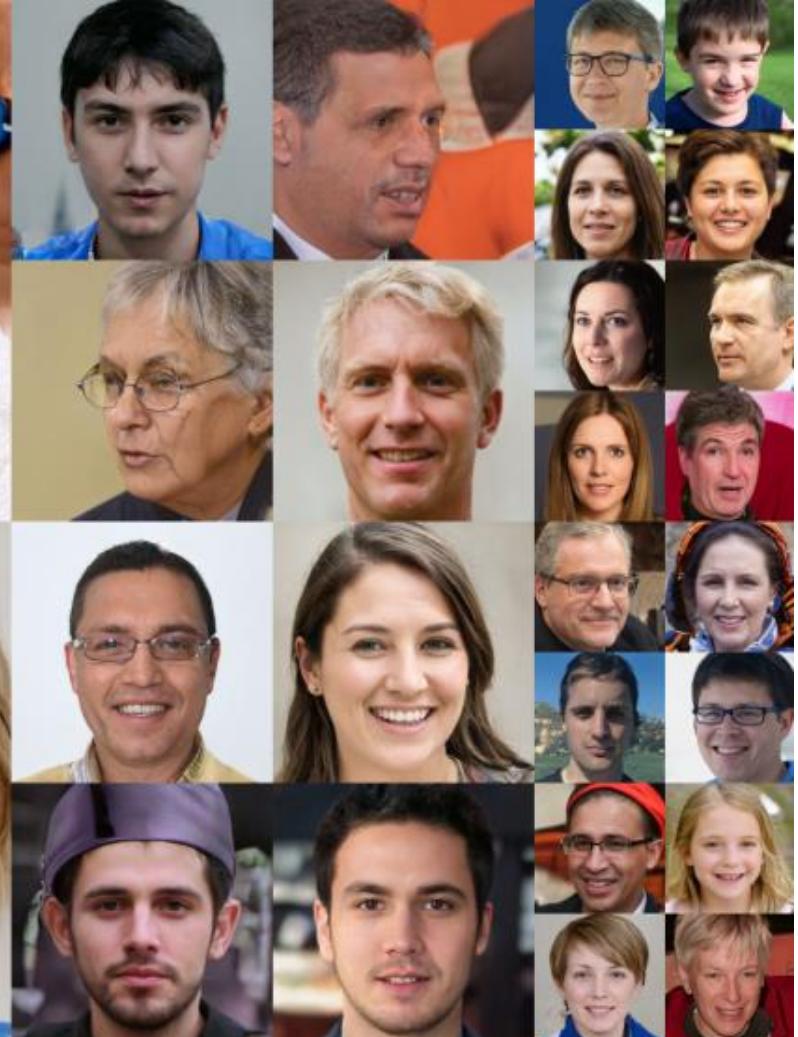
(b) Style-based generator

# StyleGAN (2018)

- Trénování po vrstvách
- Nejprve naučí generovat  $4 \times 4$ , počkají až zkonzervuje, poté přidají Upsample vrstvu  $8 \times 8$  a opakují až do  $1024 \times 1024$
- Není to „natvrdo“, ale ResNet-like prolínáním:

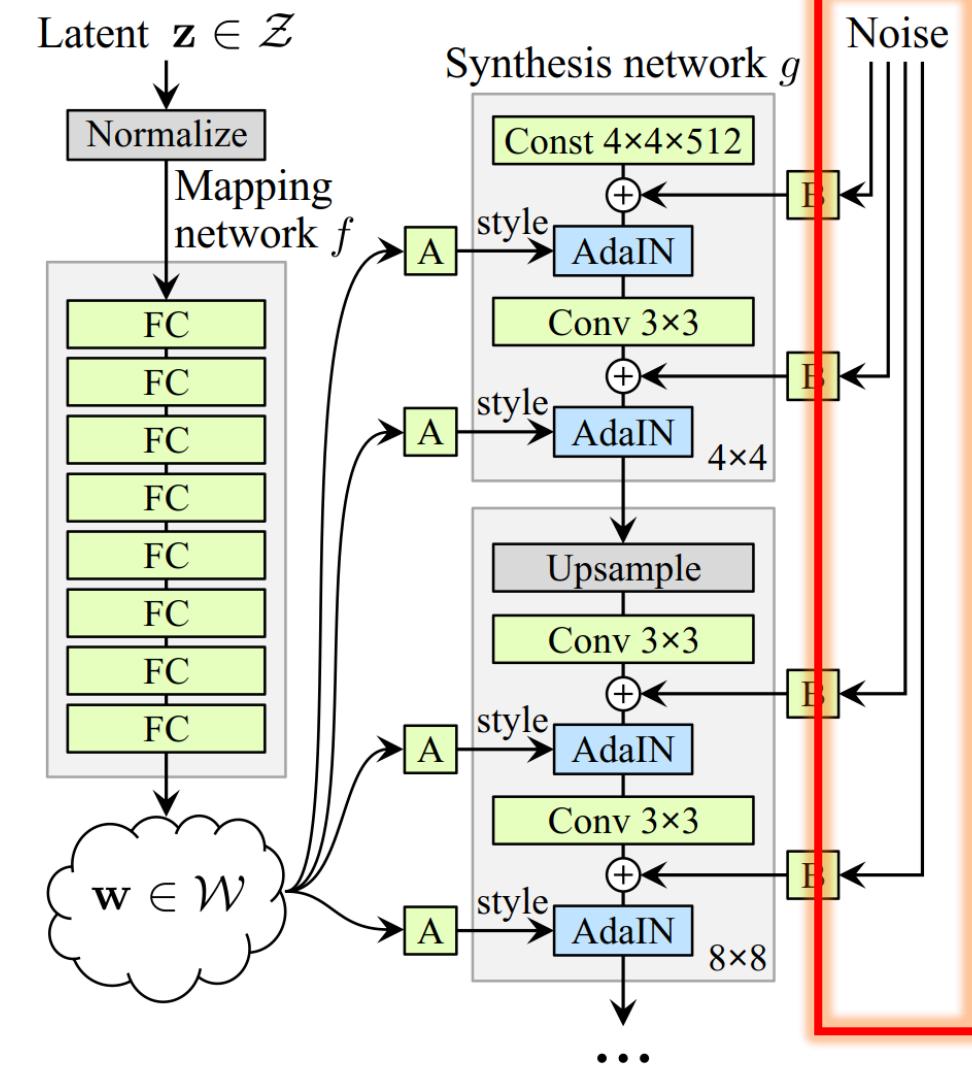


# StyleGAN (2018)



# StyleGAN (2018)

- Šum dodán pro mírnou stochasticitu
  - Ovlivní pouze drobné detaily

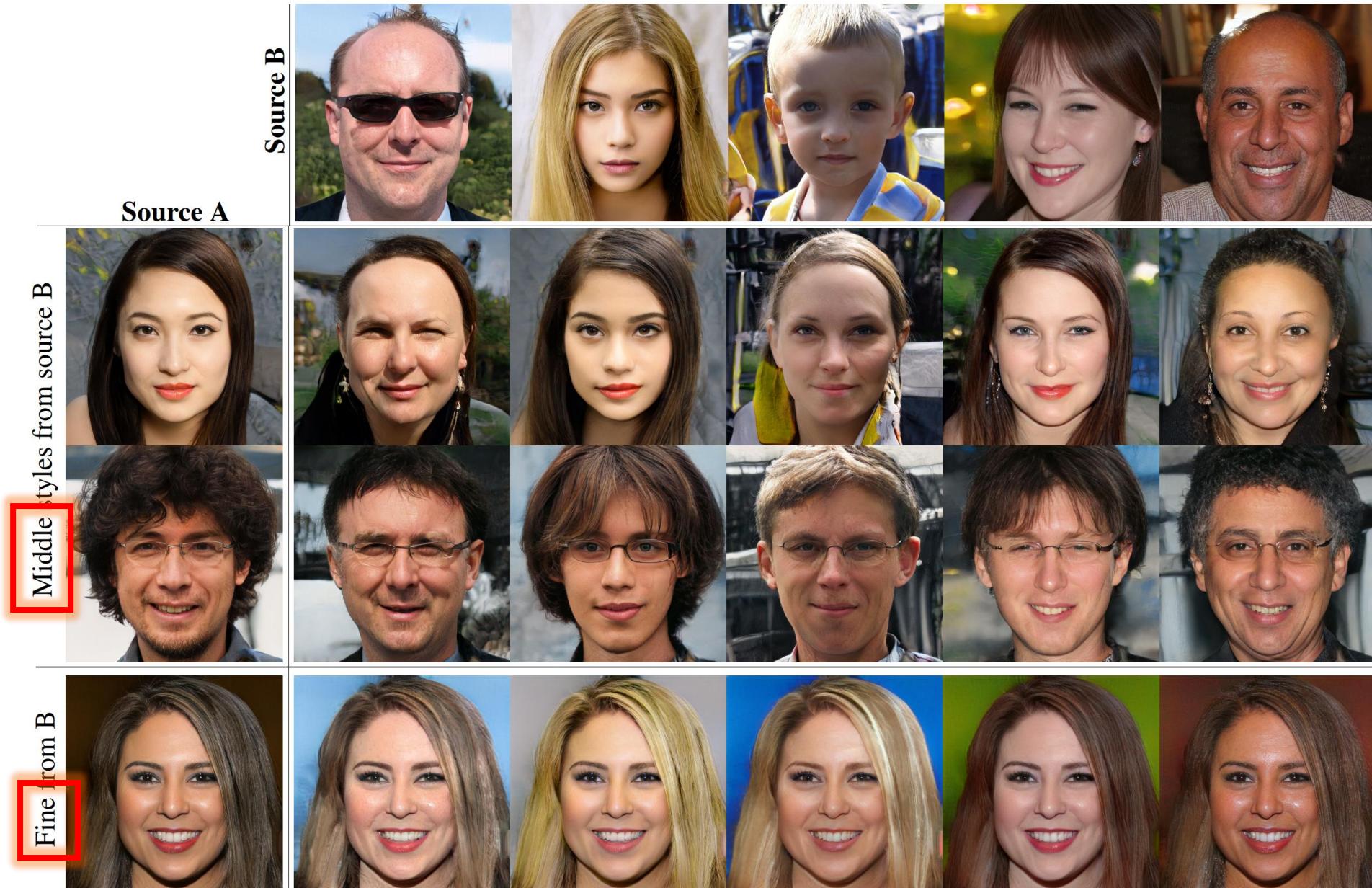


### (b) Style-based generator

# StyleGAN (2018): mixování stylů



# StyleGAN (2018): mixování stylů



# StyleGAN2 (2019)

- [Karras et al.: Analyzing and Improving the Image Quality of StyleGAN](#)
- Vylepšená verze
- Nevyužívá progresivního zvyšování rozlišení, trénuje najednou
- Snaží se především opravit artefakty



# StyleGAN2 (2019)

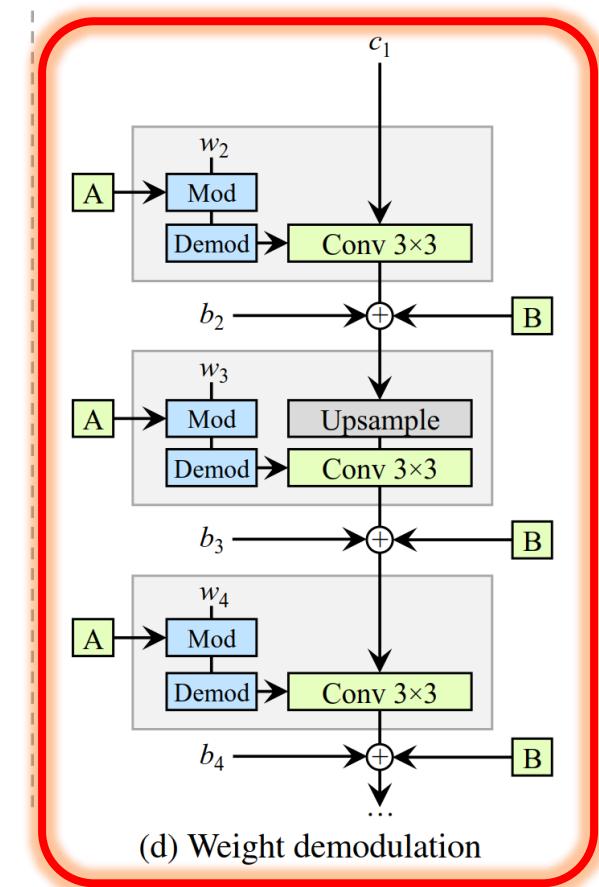
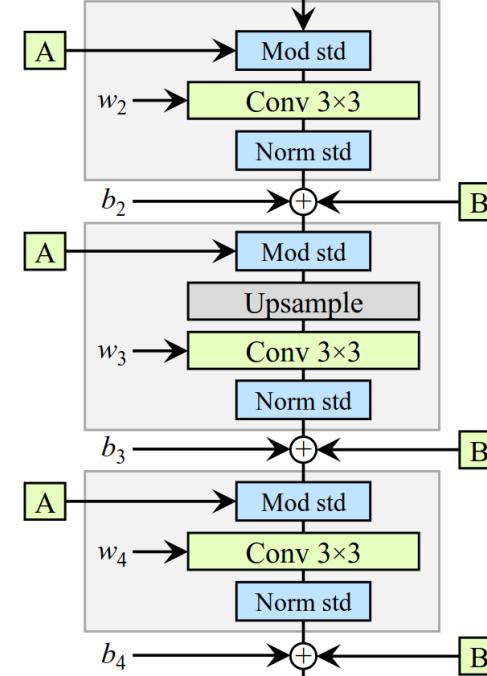
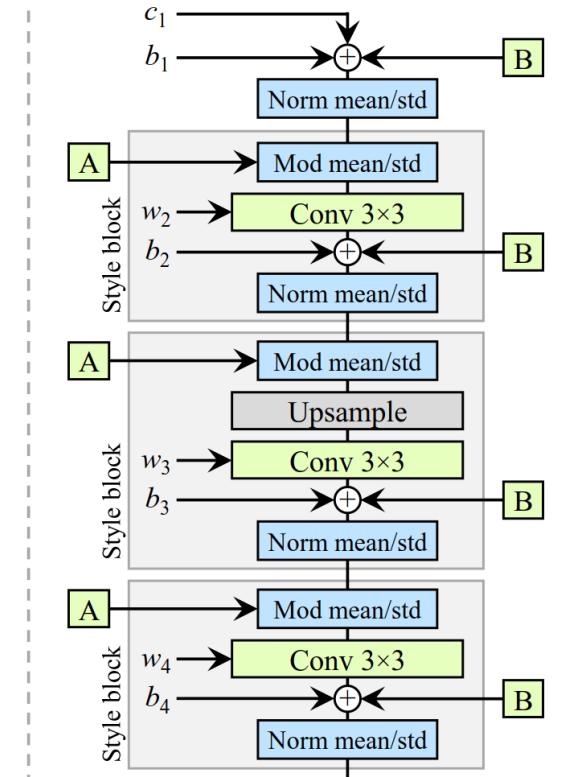
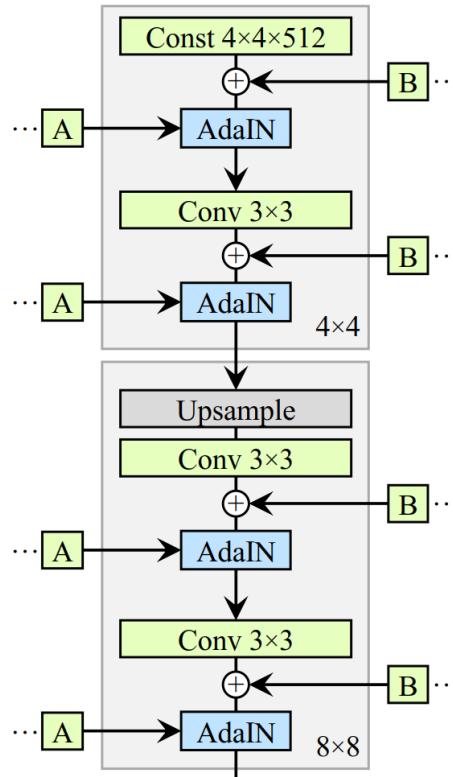
- Progresivní zvyšování způsobuje „fázové“ artefakty



Figure 6. Progressive growing leads to “phase” artifacts. In this example the teeth do not follow the pose but stay aligned to the camera, as indicated by the blue line.

# StyleGAN2 (2019)

- Změny v architektuře generátoru
- Normalizace zahrnuta přímo do vah konvolučního filtru



# StyleGAN2 (2019)



<https://thispersondoesnotexist.com/>

# StyleGAN2 (2019)



<https://thiscatdoesnotexist.com/>

# StyleGAN2 (2019)

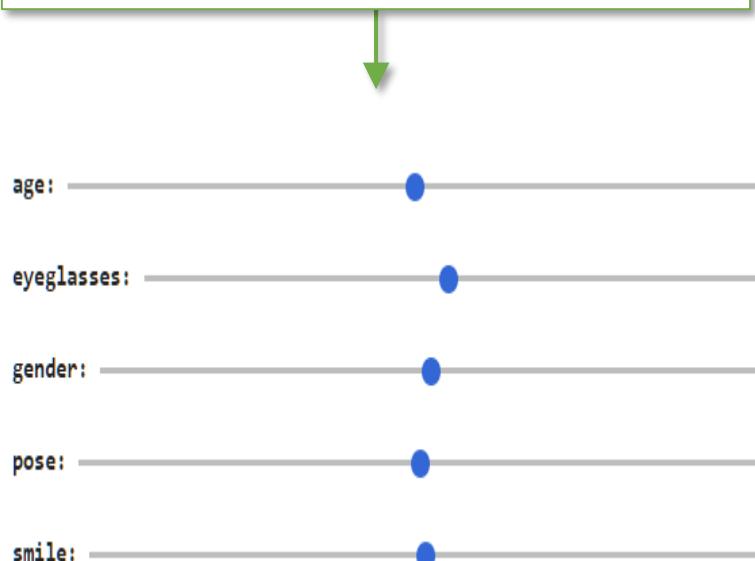


<https://thishorsedoesnotexist.com/> (thank god)

# InterFaceGAN (2019)

- [Shen et al.: Interpreting the Latent Space of GANs for Semantic Face Editing](#)
- Prostor latentních proměnných z dobře odpovídá faktorům variability v reálném světě

Směr jednotlivých os zjištěn  
natrénováním lineárního SVM nad  
vektory  $z$



# InterFaceGAN (2019)

- Pro úpravu existující fotky potřebujeme nějak zjistit odpovídající „kód“ z
- Např. optimalizací: učení vektoru  $z$  tak, aby feature mapy nebo pixely byly co nejpodobnější
- Nebo síť, která přímo převádí obrázek  $\rightarrow$  vektor  $z$
- Vznikne rekonstrukce  $\rightarrow$  na ní provedeme úpravy  $\rightarrow$  profit

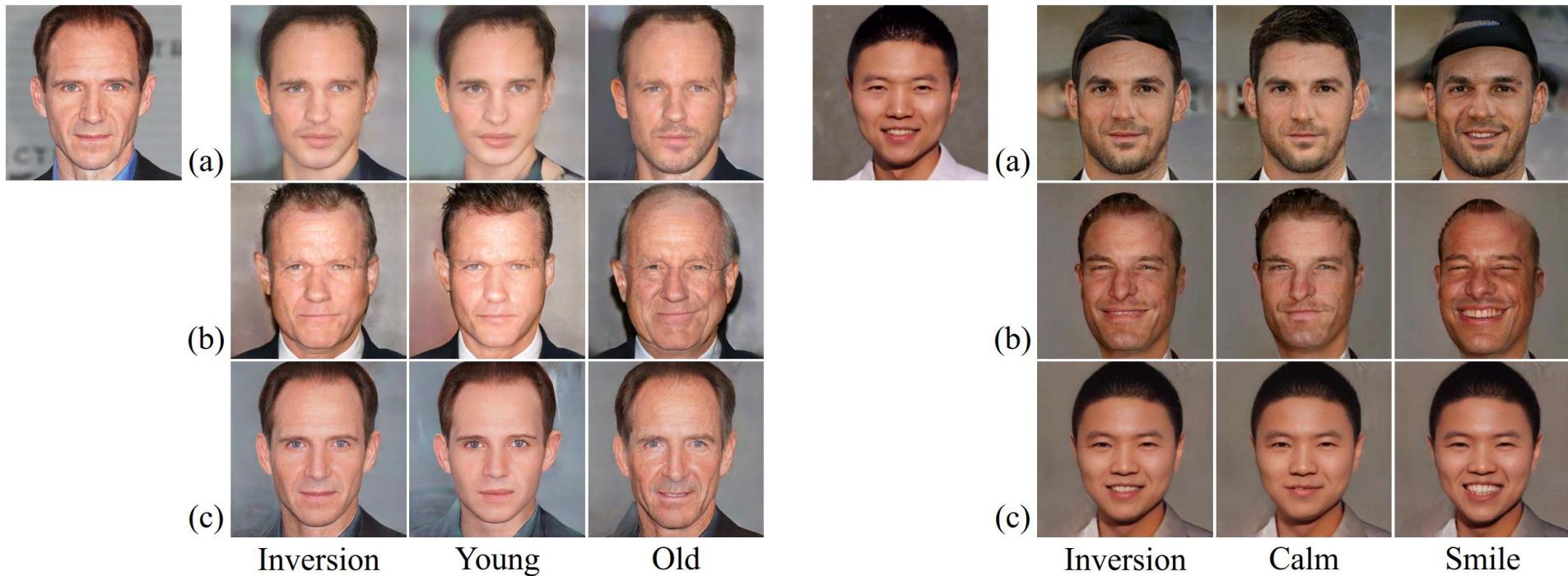


Figure 10: Manipulating real faces with respect to the attributes age and gender, using the pre-trained PGGAN [21] and StyleGAN [22]. Given an image to edit, we first invert it back to the latent code and then manipulate the latent code with InterFaceGAN. On the top left corner is the input real face. From top to bottom: (a) PGGAN with optimization-based inversion method, (b) PGGAN with encoder-based inversion method, (c) StyleGAN with optimization-based inversion method.