CONVOLUTIONAL ARCHITECTURES AND BEYOND

Sergey Nikolenko

Harbour Space University

June 16, 2026

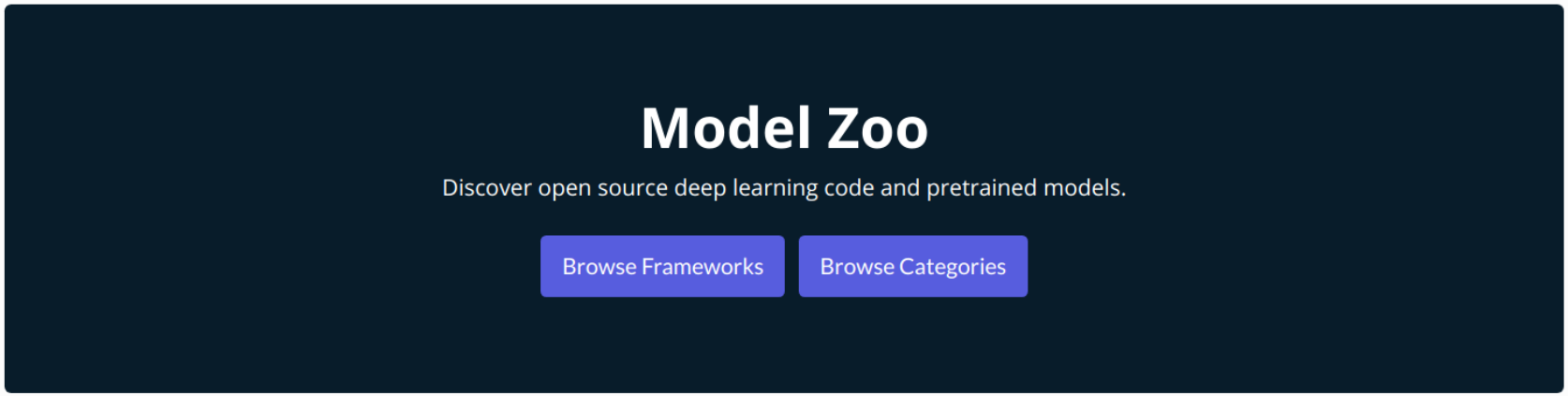
Random facts:

- on June 16, 1884, LaMarcus Thompson's *Switchback Railway* opened at Coney Island as the first purpose-built roller coaster; it ambled along at about six miles per hour for a nickel a ride, yet proved profitable enough to spawn an entire industry
- on June 16, 1904, James Joyce set the whole action of *Ulysses*, all seven hundred-odd pages of it, on this one ordinary Dublin day, because it was the date of his first walk out with Nora Barnacle; readers now retrace it each ``Bloomsday'' in Edwardian dress
- on June 16, 1911, the Computing-Tabulating-Recording Company was incorporated in New York, merging a scale maker, a time-clock maker, and Herman Hollerith's census tabulating-machine business; the ungainly conglomerate was not renamed IBM until 1924
- on June 16, 1960, Alfred Hitchcock's *Psycho* premiered in New York under a strict ``no one admitted after the start'' rule, enforced by Pinkerton guards and life-sized cardboard cutouts of the director; that gimmick is why films have had fixed start times ever since
- on June 16, 1963, Valentina Tereshkova, a textile worker and amateur parachutist, became the first woman in space aboard *Vostok 6*, circling the Earth 48 times in just under three days; no other woman would reach orbit for another 19 years

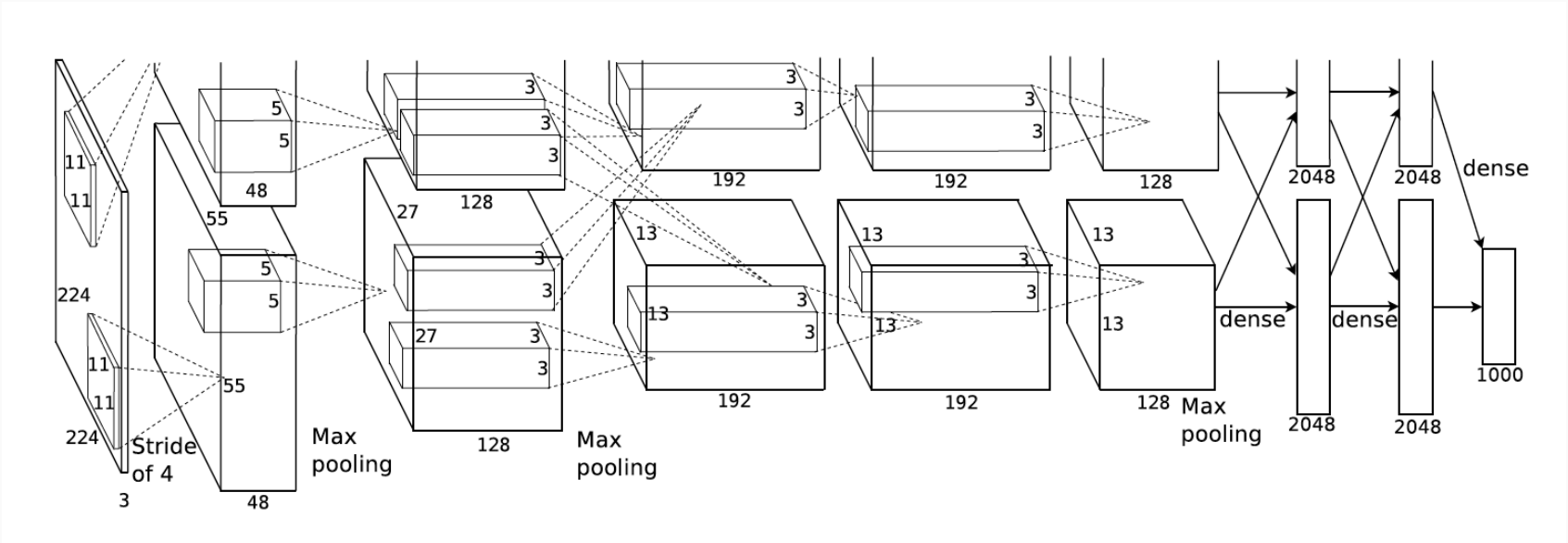
Important convolutional architectures

The Zoo

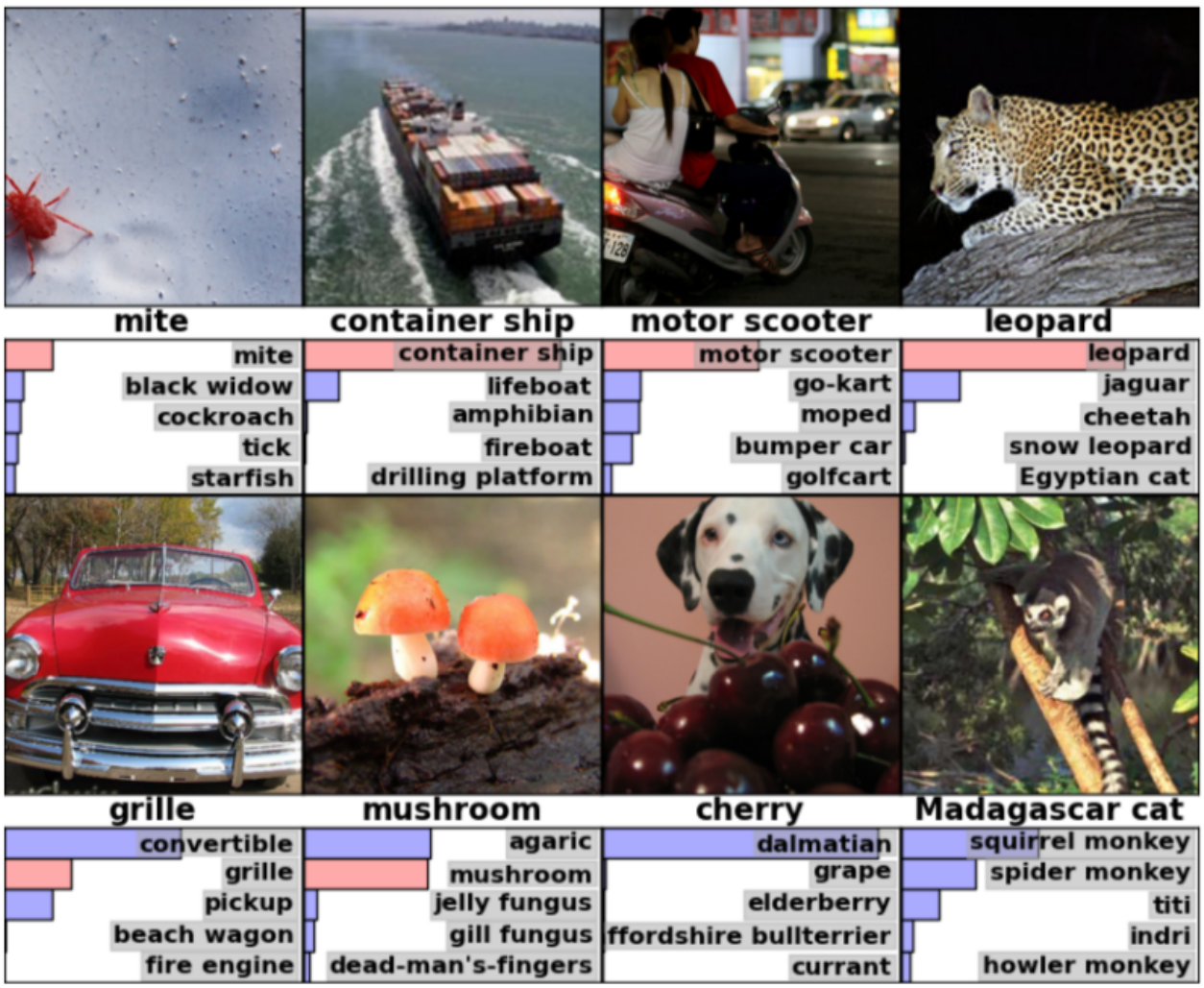
- There is a huge number of different models based on the CNN idea.
- Most of them are accompanied by code.
- Many provide models pretrained on the ImageNet or other large standard datasets.
- But usually they are all based on a few standard ideas, and that's exactly what we are going to consider.



(Krizhevsky, Sutskever, Hinton, 2012): AlexNet

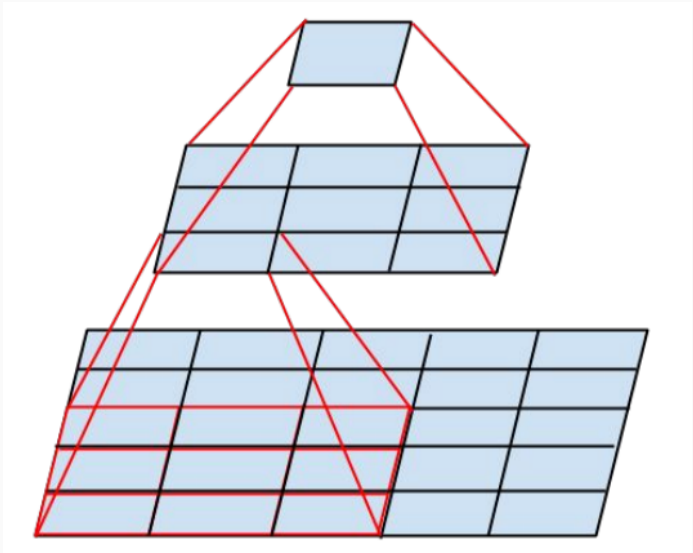


(Krizhevsky, Sutskever, Hinton, 2012): AlexNet



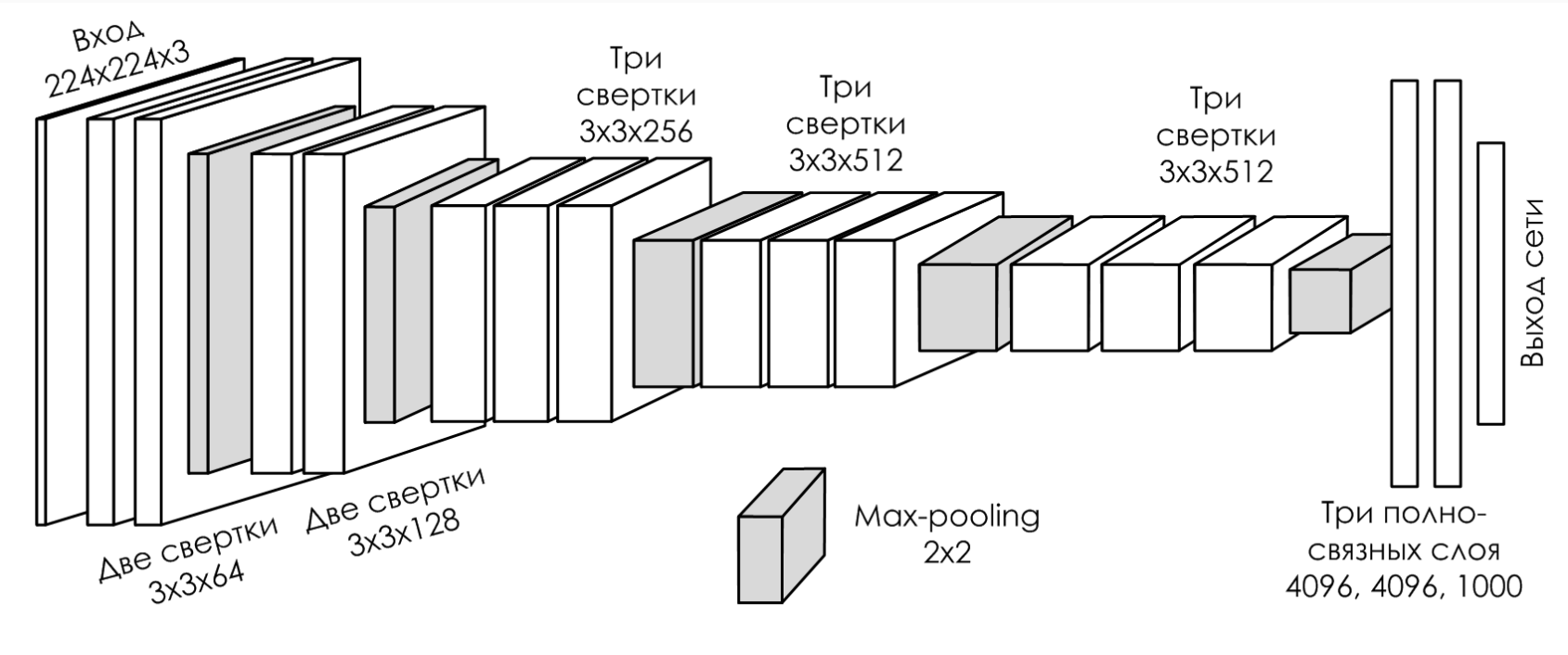
VGG

- VGG (Oxford Visual Geometry Group): let's represent large convolutions as compositions of 3×3 convolutions.
- This reduces the number of weights and makes the network deeper.



VGG

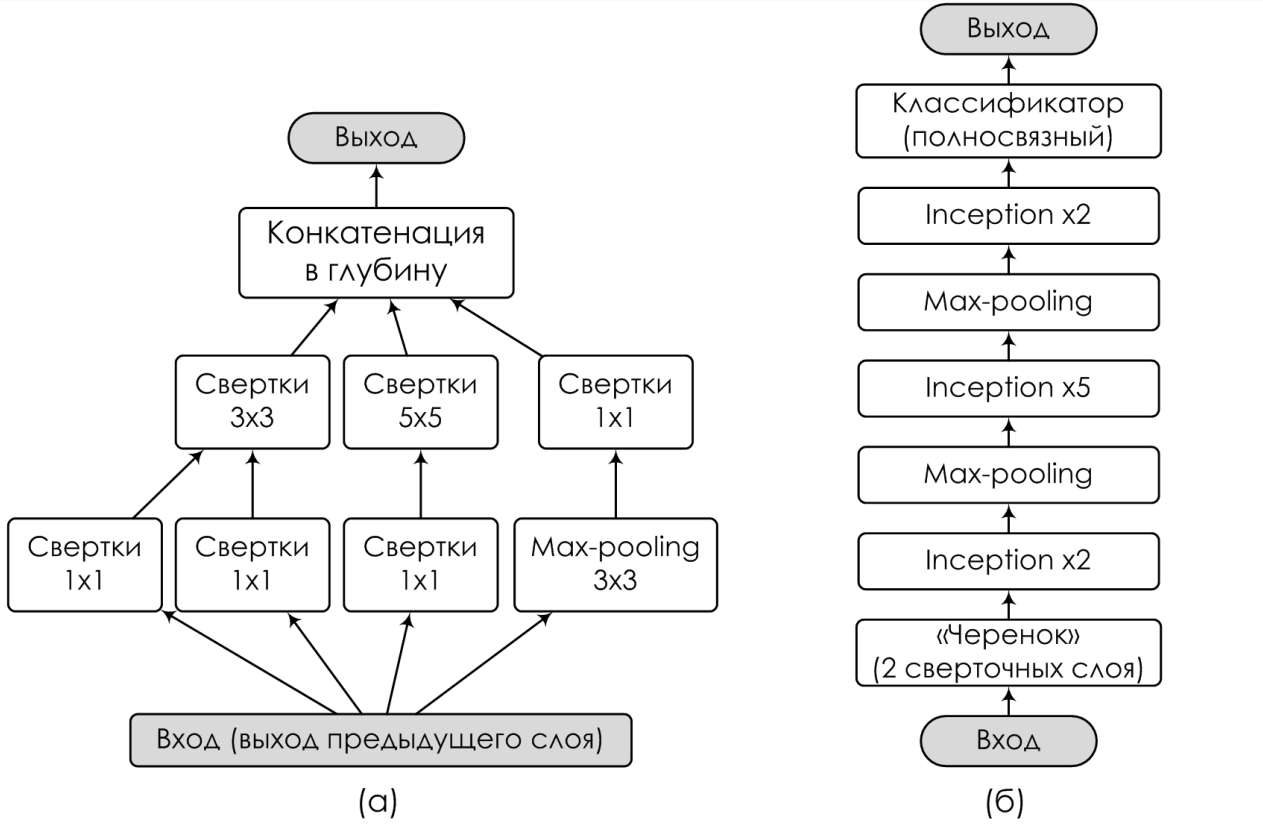
- The VGG network used in ImageNet Large Scale Visual Recognition Competition (ILSVRC-2014):



- NVIDIA has been optimizing GPUs especially for 3×3 convolutions.

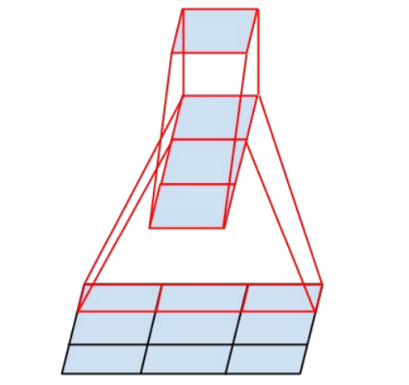
Inception

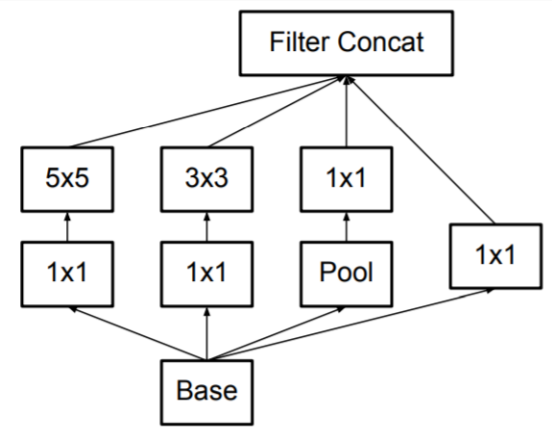
- *Inception* and *GoogLeNet* based on it: let's use the "network in network" idea to reduce the number of weights even further.
- Another idea – additional classifiers (additional terms for the objective function).

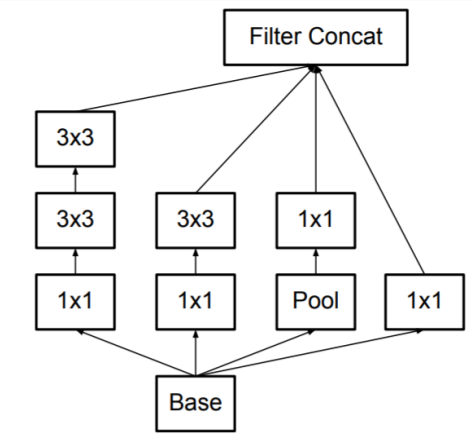


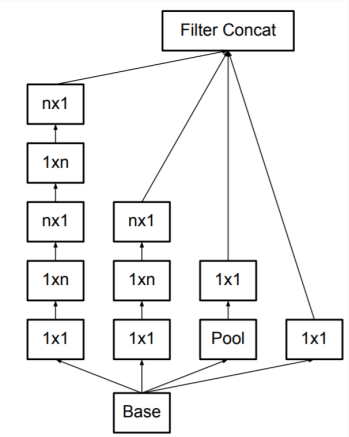
Inception

- In Inception v2 (Szegedy et al., 2015) $n \times n$ convolutions were replaced by $n \times 1$ and $1 \times n$ convolutions:



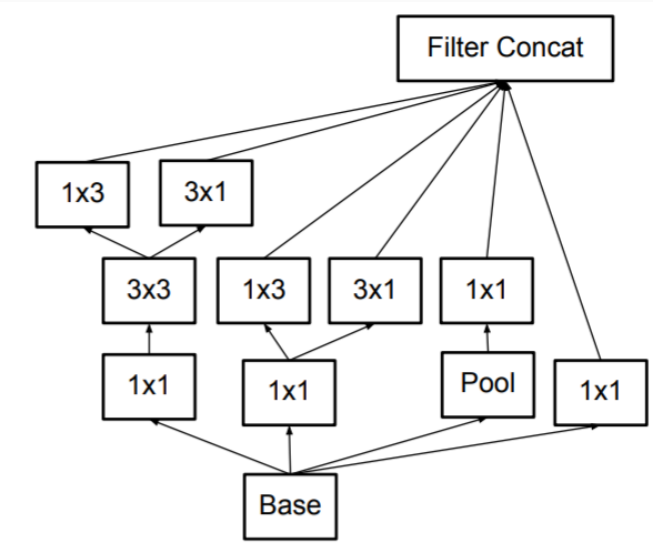






Inception

- Filter banks in Inception v2 became wide rather than deep:



- The same paper also introduced Inception v3; same thing, but with RMSProp, factorized 7×7 convolutions, batchnorm in additional classifiers and...

Inception

- Label smoothing:
  - in a regular classifier we train the softmax with hard objective $q(k) = [k = y]$, i.e., arguments of the softmax function "want" to grow unboundedly;
  - as a result, the models overfit and become too certain;
  - possible solution: let's instead optimize

$$q'(k) = (1 - \epsilon)[k = y] + \epsilon u(k)$$

for some prior distribution $u(k)$, say $u(k) = \frac{1}{K}$.

Network	Crops Evaluated	Top-5 Error	Top-1 Error
GoogLeNet [20]	10	-	9.15%
GoogLeNet [20]	144	-	7.89%
VGG [18]	-	24.4%	6.8%
BN-Inception [7]	144	22%	5.82%
PReLU [6]	10	24.27%	7.38%
PReLU [6]	-	21.59%	5.71%
Inception-v3	12	19.47%	4.48%
Inception-v3	144	**18.77%**	**4.2%**

ResNet

- Residual learning: let's train the *differences* (residues) between one layer and the next.
- Then the gradients will be able to flow with no obstacle.
- A function implemented by a residual unit looks like

$$\mathbf{y}^{(k)} = F(\mathbf{x}^{(k)}) + \mathbf{x}^{(k)},$$

  where $\mathbf{x}^{(k)}$ is the input vector of layer k, $F(x)$ is the function computed by the layer, and $\mathbf{y}^{(k)}$ is the output of the residual layer that will then become $\mathbf{x}^{(k+1)}$ for the next layer.
- Now the gradient can pass through and does not vanish when F becomes saturated:

$$\frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{x}^{(k)}} = 1 + \frac{\partial F(\mathbf{x}^{(k)})}{\partial \mathbf{x}^{(k)}}.$$

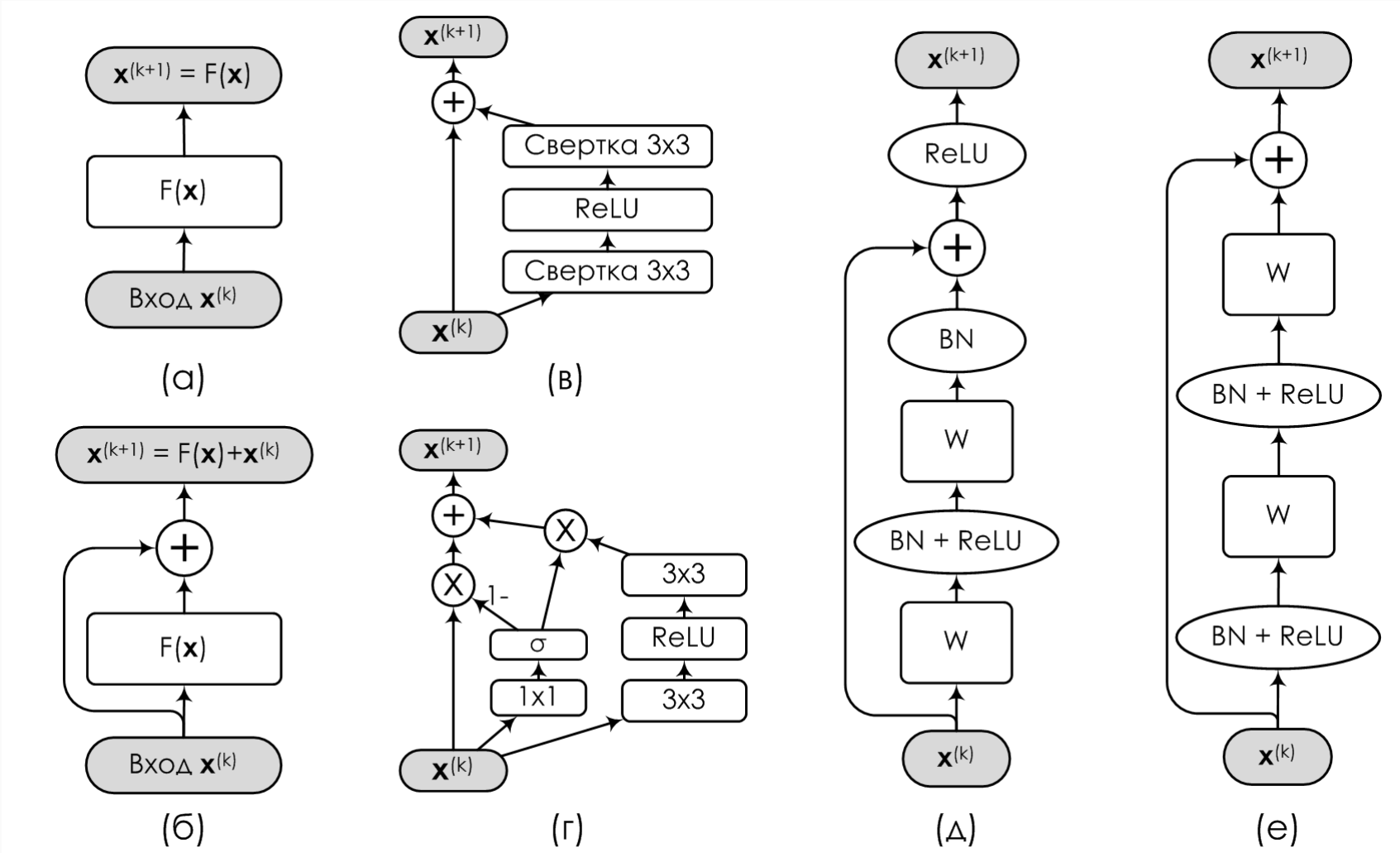
ResNet

- This has allowed for *very* deep networks.
- Another similar approach – *highway networks* by Jürgen Schmidhuber.
- We again represent $\mathbf{y}^{(k)}$, output of layer k, as a linear combination of $\mathbf{x}^{(k)}$ and $F(\mathbf{x}^{(k)})$, but in a different way:

$$\mathbf{y}^{(k)} = C(\mathbf{x}^{(k)})\mathbf{x}^{(k)} + T(\mathbf{x}^{(k)})F(\mathbf{x}^{(k)}),$$

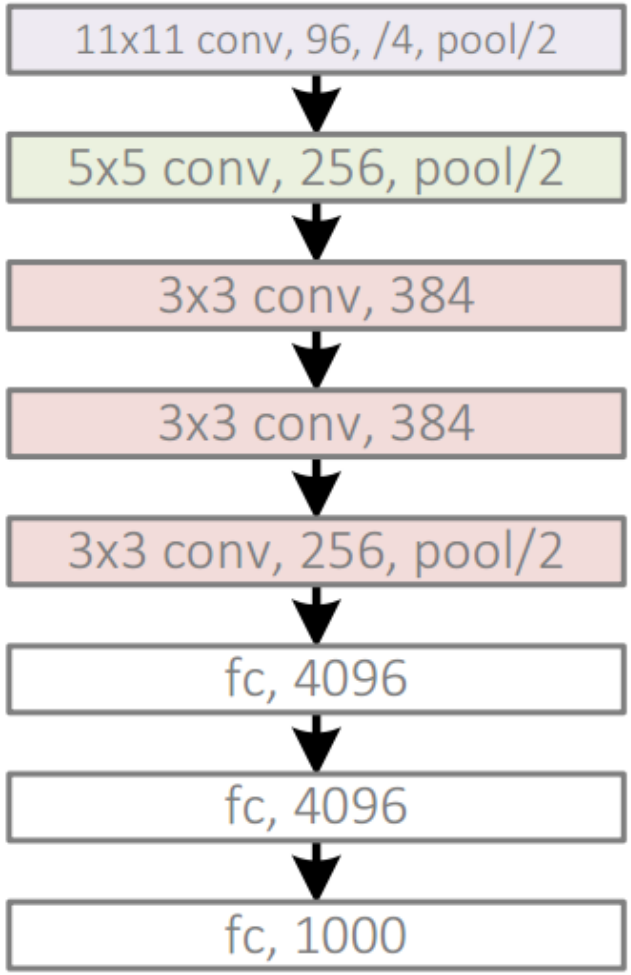
  where C is the *carry gate*, and T is the *transform gate*; usually it's a convex combination, $C = 1 - T$.
- Practice shows that the residual connections should be as "straight" as possible.

ResNet: variations



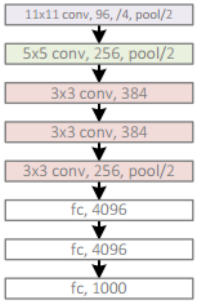
Revolution of Depth (Kaiming He)

AlexNet, 8 layers
(ILSVRC 2012)

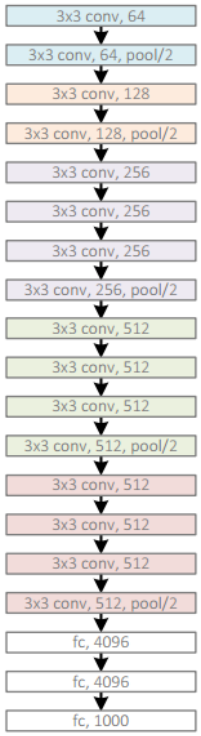


Revolution of Depth (Kaiming He)

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

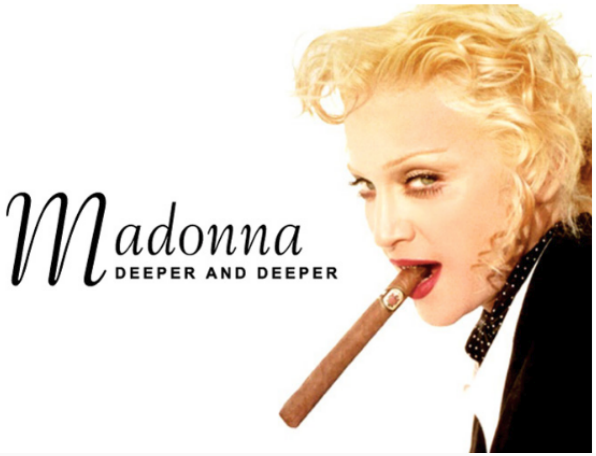
Revolution of Depth (Kaiming He)

ResNet led to the revolution of depth

AlexNet, 8 layers
(ILSVRC 2012)

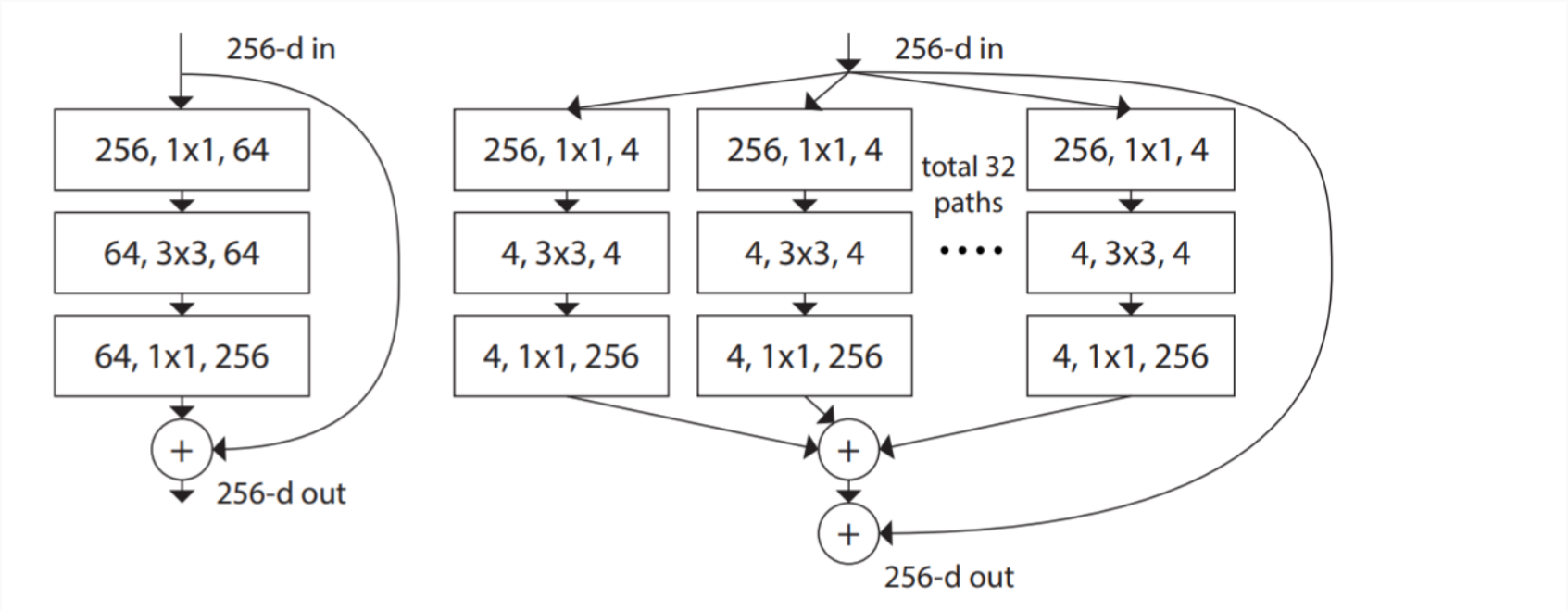
VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)



ResNeXt

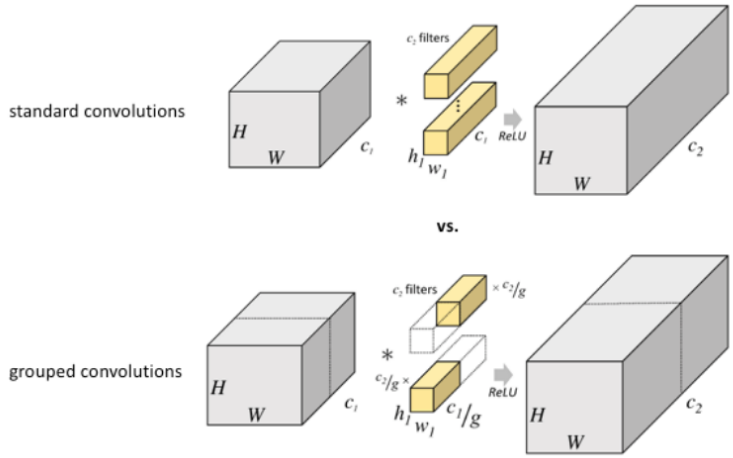
- ResNeXt (Xie et al., 2016): let's replace ResNet units with "split-transform-merge" units, similar to Inception.



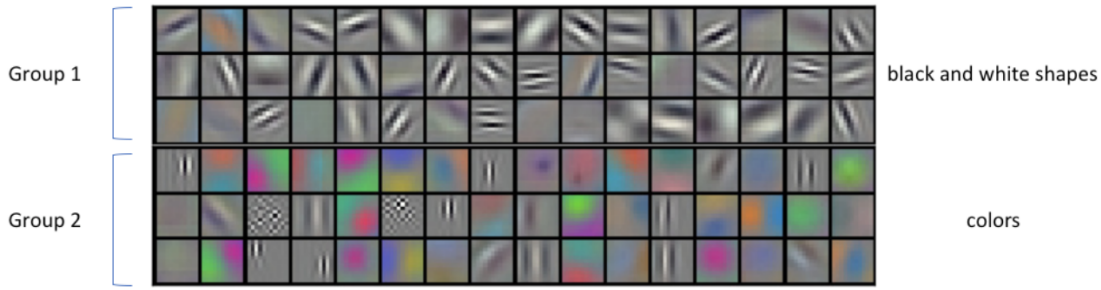
- The input is divided into blocks w.r.t. channels, and every block gets its own convolutions.

ResNeXt

- The idea is similar to group convolutions, used already in AlexNet for parallelization:

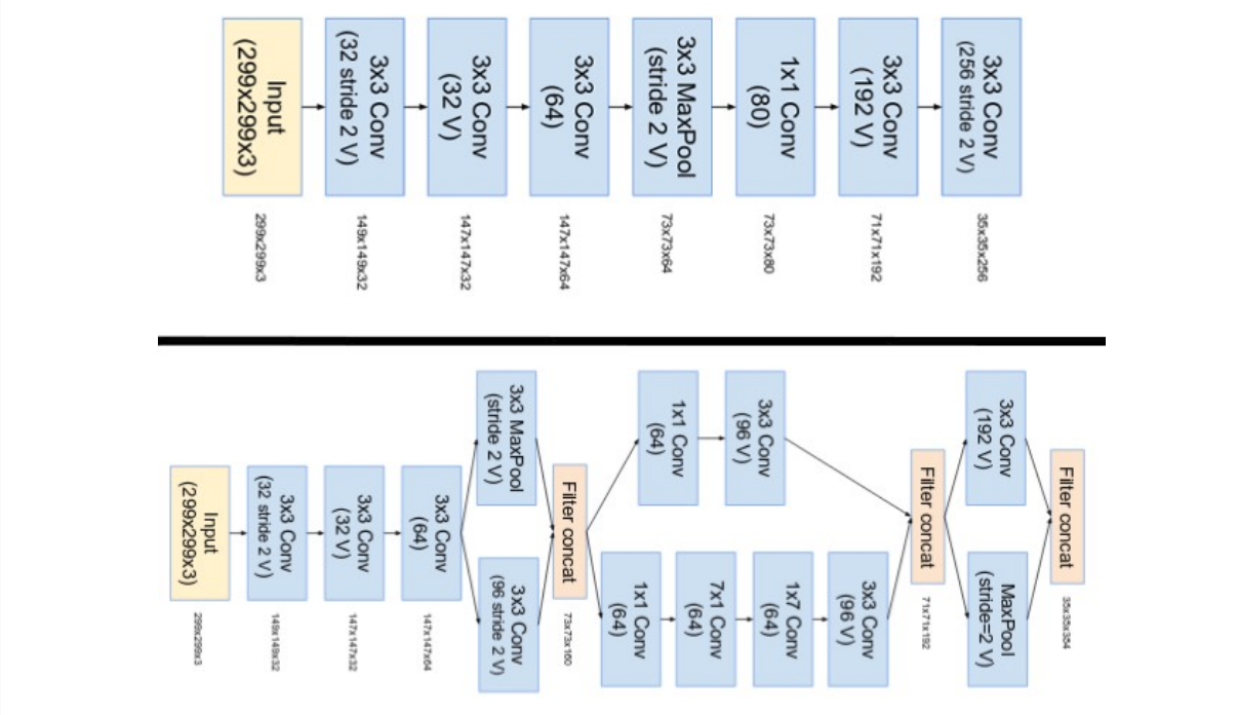


- They do yield a kind of a specialization in the results:



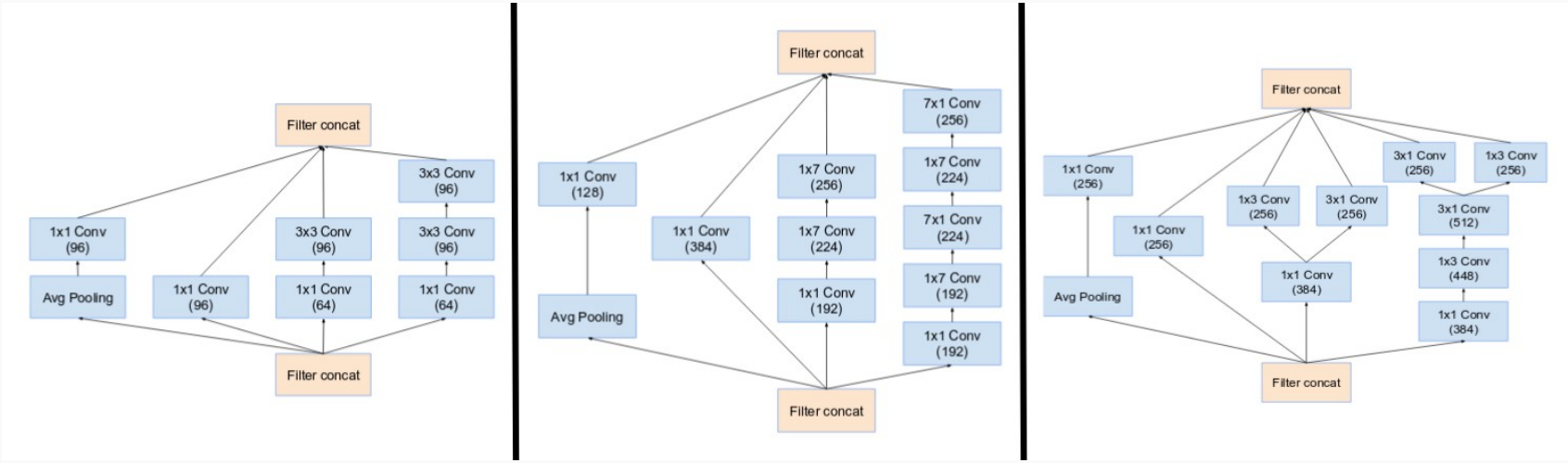
Inception v4 and Inception ResNet

- Another classic paper (Szegedy et al., 2016) introduced Inception v4 and Inception ResNet.
- Inception v4 – let's standardize everything and simplify the units. First, the "stem":



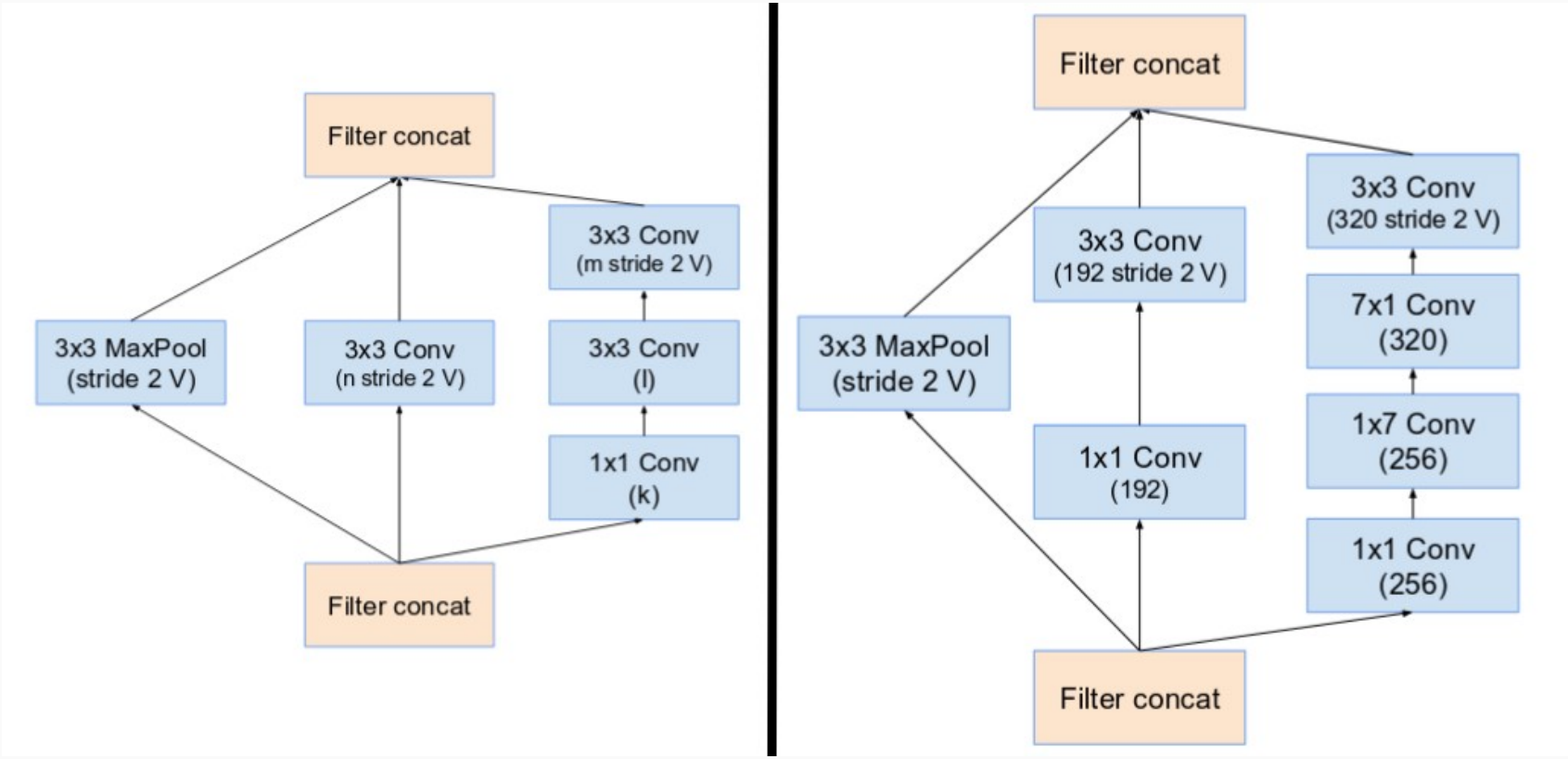
Inception v4 and Inception ResNet

- Second, we now have three basic blocks A, B, and C:



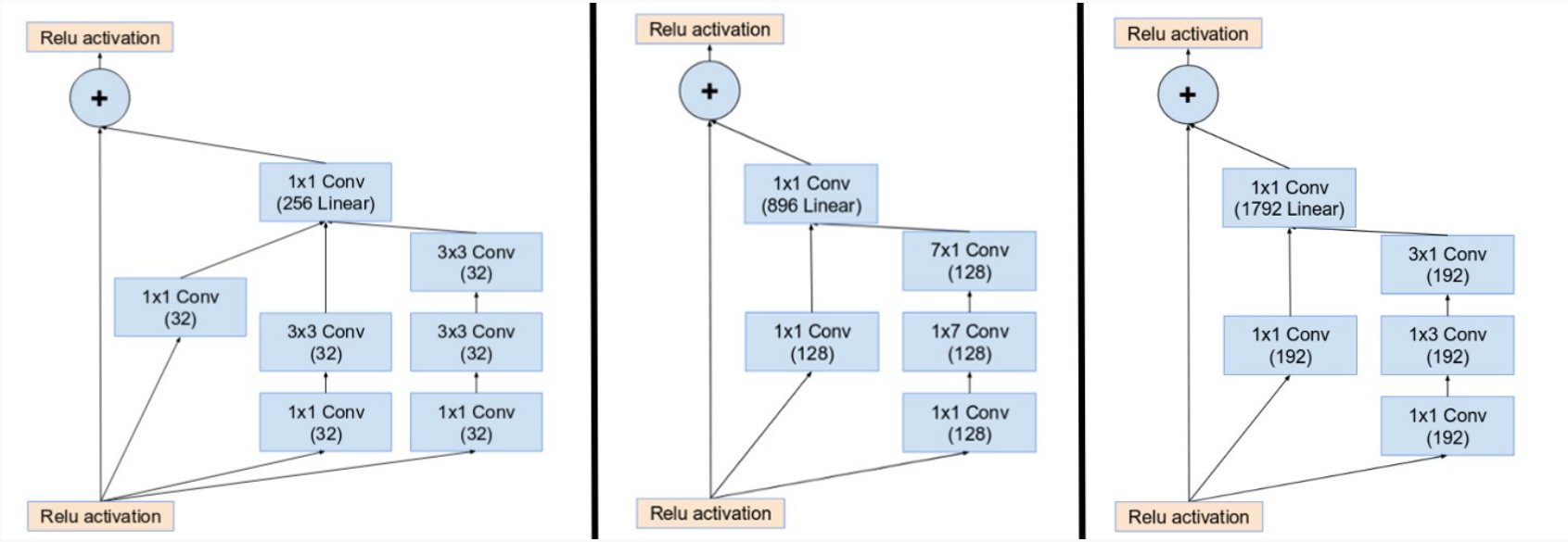
Inception v4 and Inception ResNet

- And special reduction blocks to reduce the dimensions:



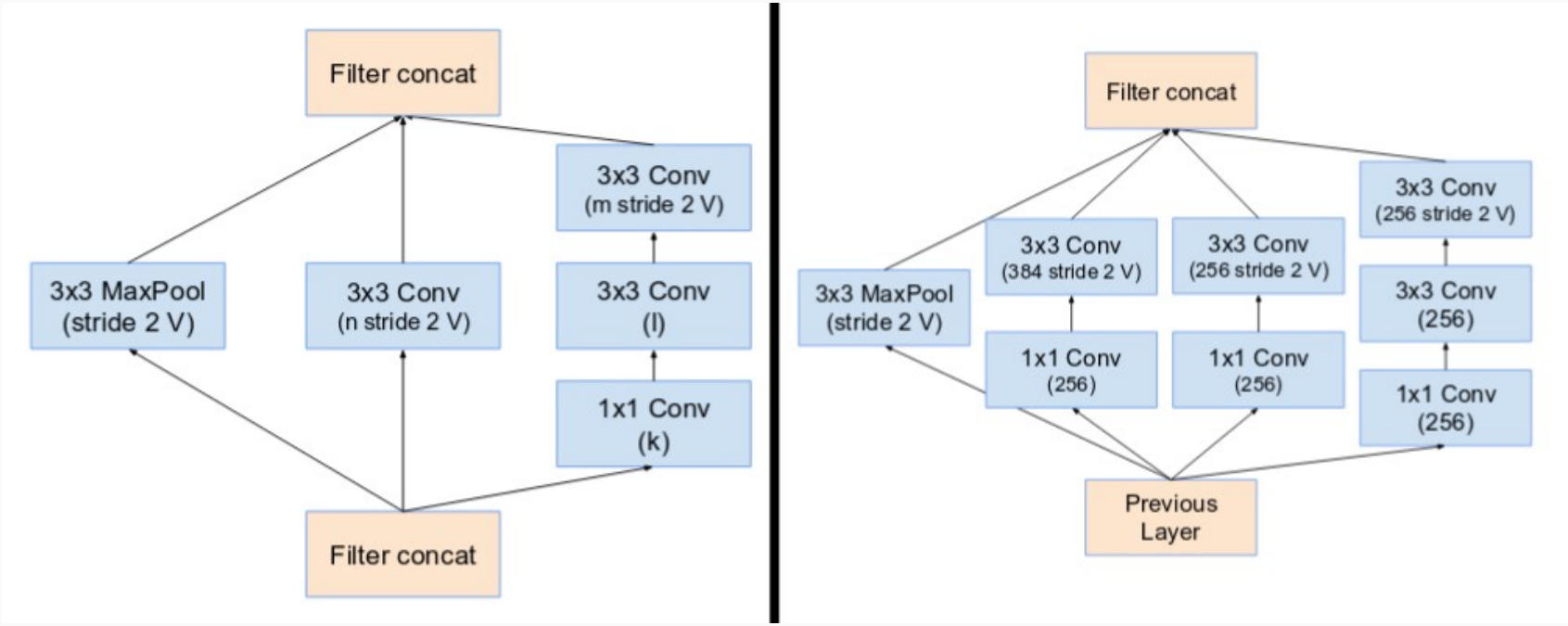
Inception v4 and Inception ResNet

- Inception ResNet adds residual connections to these blocks:



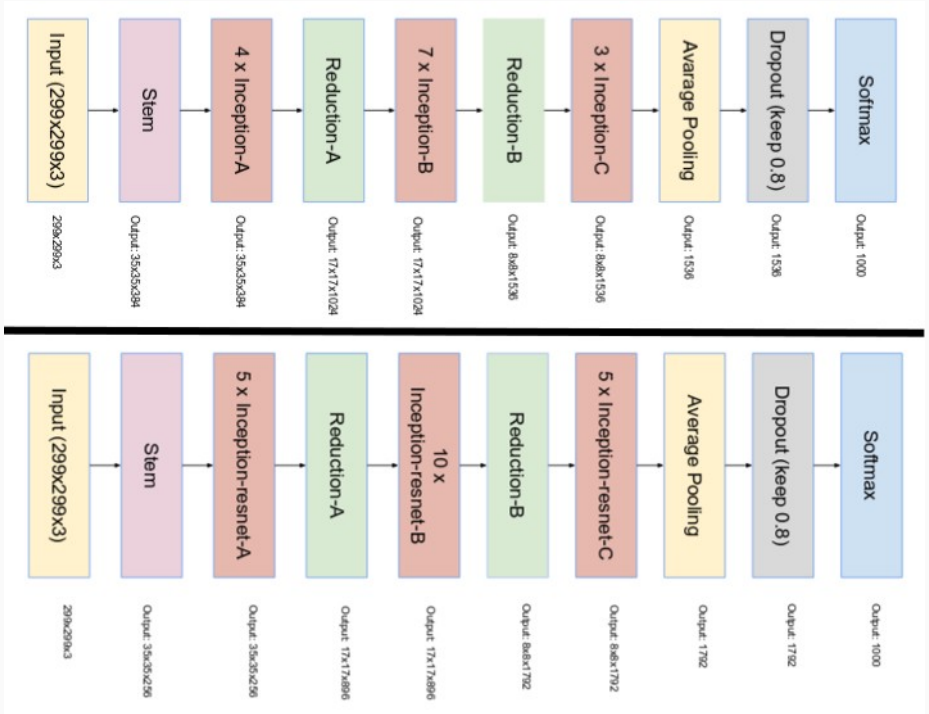
Inception v4 and Inception ResNet

- There is no pooling now but there are still reduction blocks:



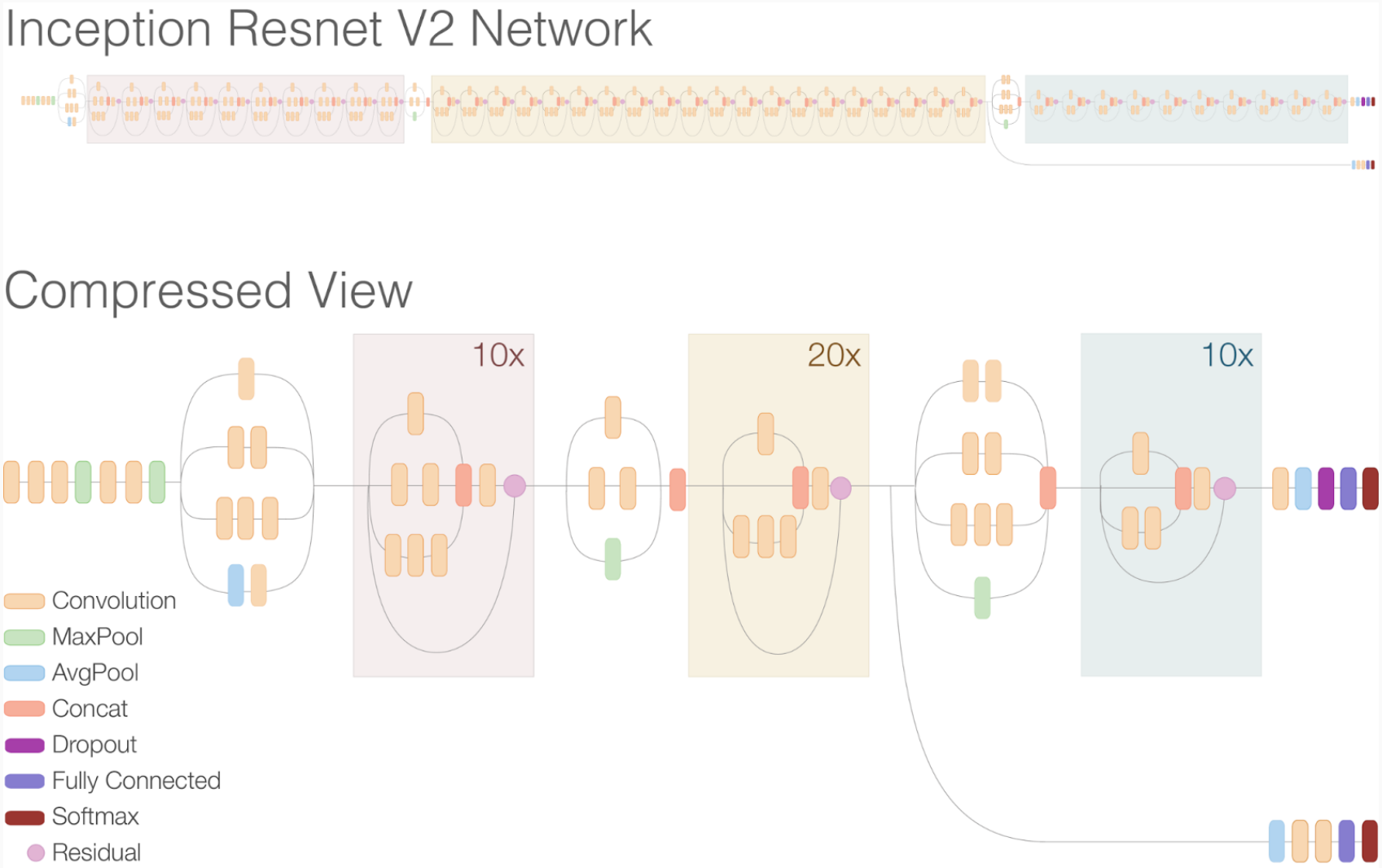
Inception v4 and Inception ResNet

- As a result, the architecture has become even simpler; Inception v4 (top), Inception ResNet (bottom):



Inception v4 and Inception ResNet

- Inception ResNet v2:



Inception v4 and Inception ResNet

- And it works quite well:

Network	Crops	Top-1 Error	Top-5 Error
ResNet-151 [5]	10	21.4%	5.7%
Inception-v3 [15]	12	19.8%	4.6%
Inception-ResNet-v1	12	19.8%	4.6%
Inception-v4	12	18.7%	4.2%
Inception-ResNet-v2	12	18.7%	4.1%

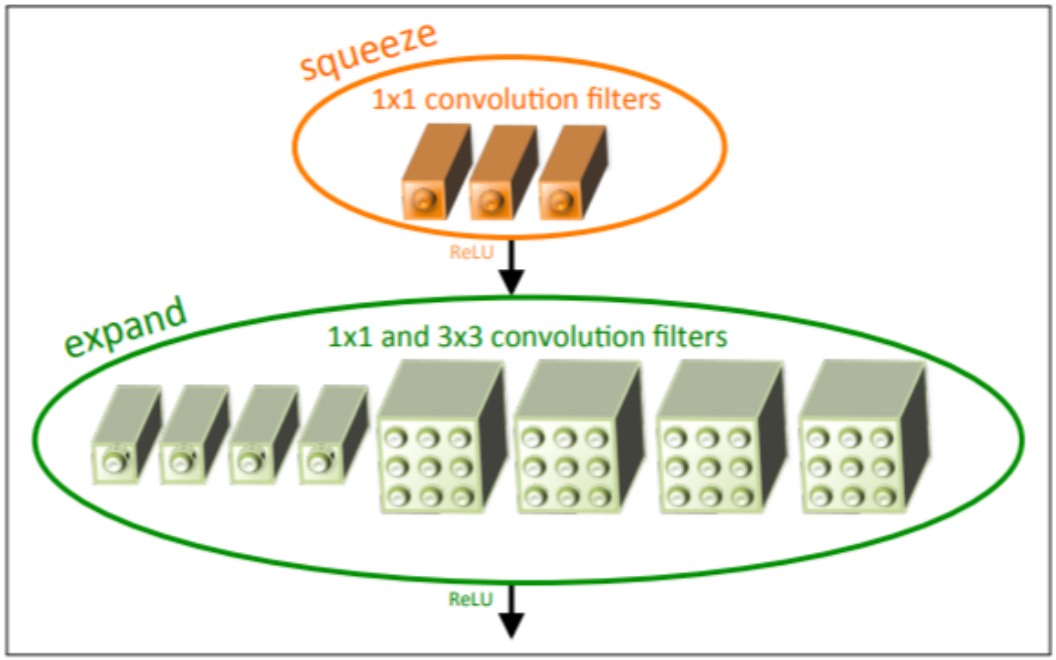
Network	Crops	Top-1 Error	Top-5 Error
ResNet-151 [5]	dense	19.4%	4.5%
Inception-v3 [15]	144	18.9%	4.3%
Inception-ResNet-v1	144	18.8%	4.3%
Inception-v4	144	17.7%	3.8%
Inception-ResNet-v2	144	17.8%	3.7%

SqueezeNet

- SqueezeNet (Iandola et al., 2017) – how to reduce the number of parameters:
  - replace 3×3 filters with 1×1;
  - reduce the number of inputs for 3×3 convolutions;
  - delay downsampling as late as possible to increase the size of activation maps.

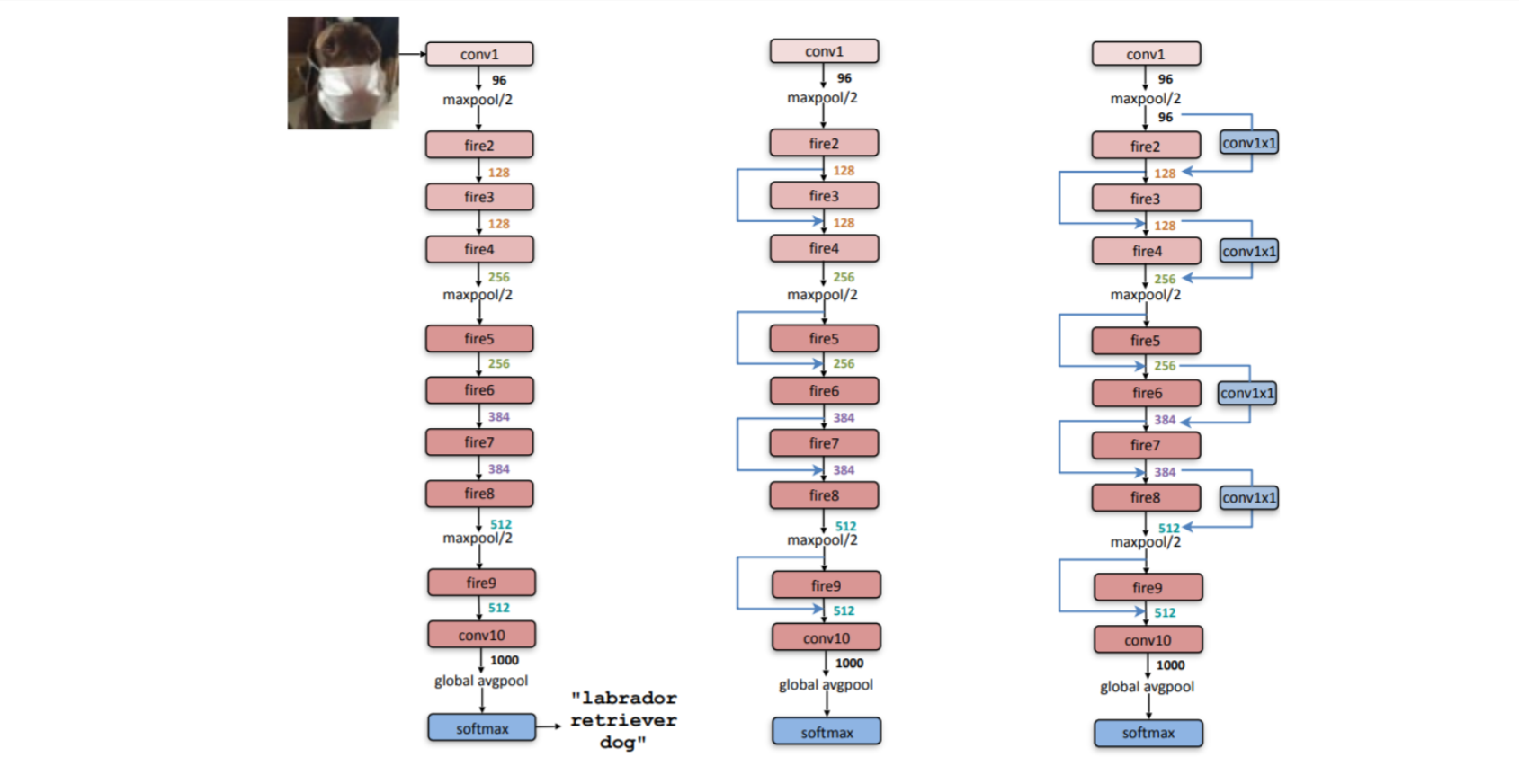
SqueezeNet

- Fire module:
  - squeeze convolutional layer (1×1 only);
  - expand layer (1×1 and 3×3).



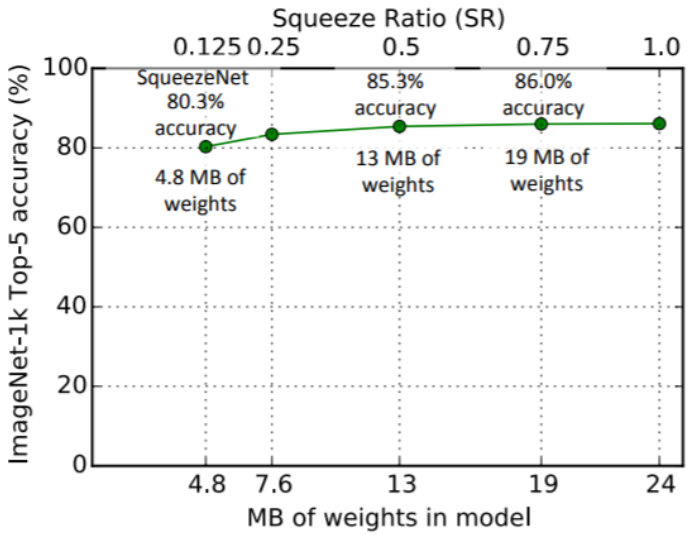
SqueezeNet

- General SqueezeNet architecture:

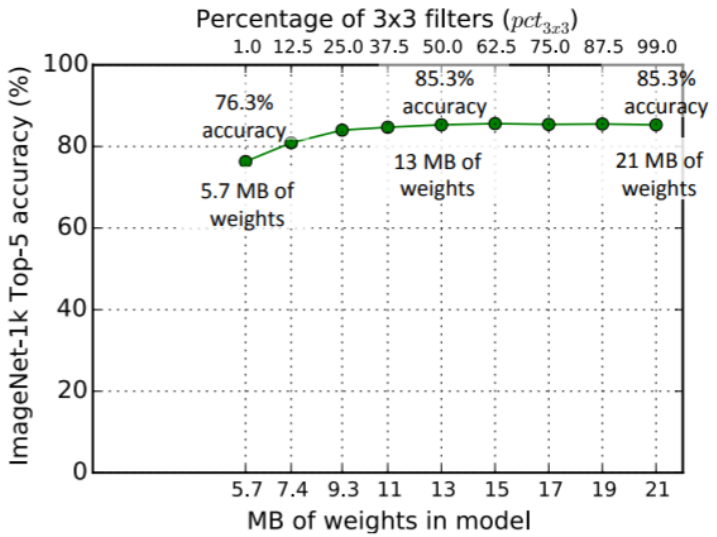


SqueezeNet

- We get 50x fewer parameters than AlexNet. But:



(a) Exploring the impact of the squeeze ratio (SR) on model size and accuracy.

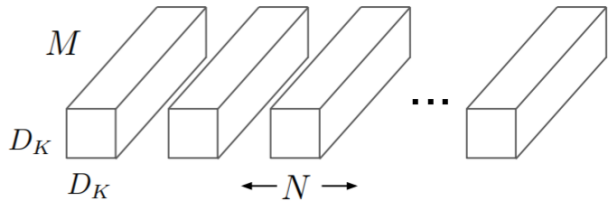


(b) Exploring the impact of the ratio of 3x3 filters in expand layers (pct_{3x3}) on model size and accuracy.

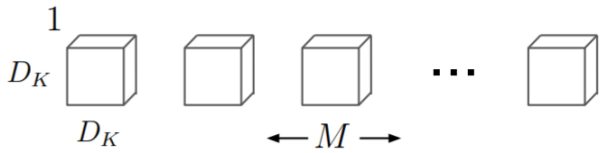
Architecture	Top-1 Accuracy	Top-5 Accuracy	Model Size
Vanilla SqueezeNet	57.5%	80.3%	4.8MB
SqueezeNet + Simple Bypass	**60.4%**	**82.5%**	4.8MB
SqueezeNet + Complex Bypass	58.8%	82.0%	7.7MB

MobileNet

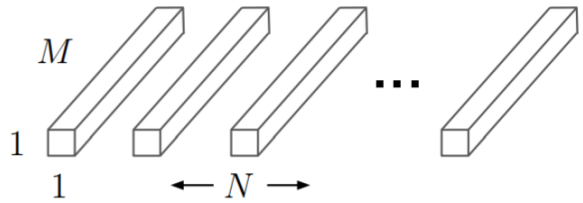
- MobileNet (Howard et al., 2017): networks for mobile devices.
- Depthwise separable convolutions: let's decompose a convolution into a depthwise convolution (one filter for each channel) and a 1×1 convolution.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

MobileNet

- Then the structure of a layer will be more complex (but with fewer weights), and the overall architecture is not so deep:

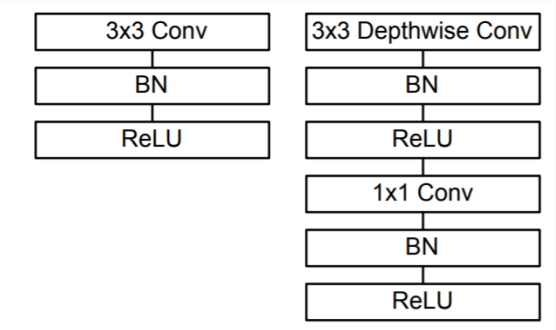


Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

MobileNet

- We see that we can save a lot of parameters at the price of a small decrease in quality:

Table 8. MobileNet Comparison to Popular Models

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Table 9. Smaller MobileNet Comparison to Popular Models

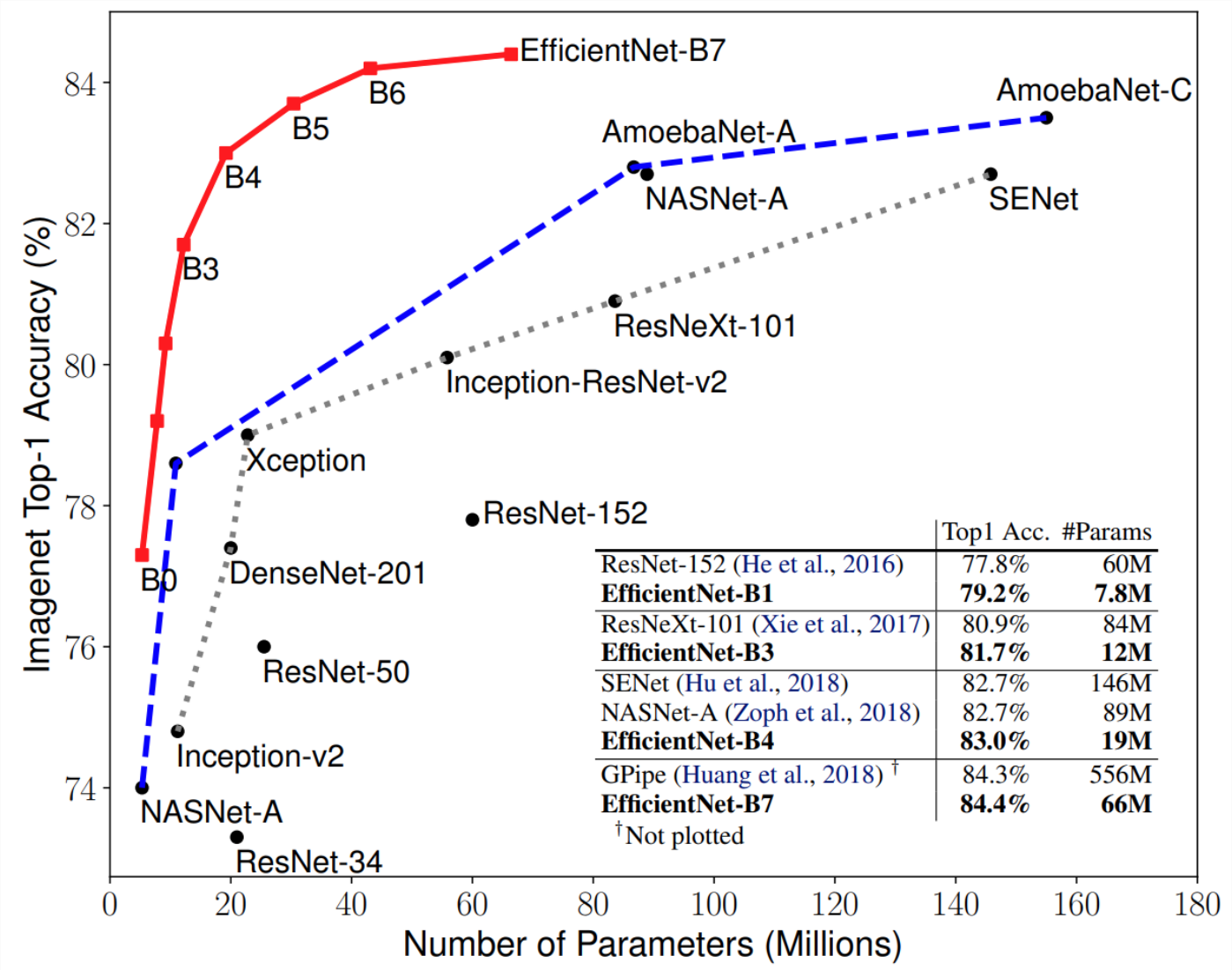
Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60

Table 10. MobileNet for Stanford Dogs

Model	Top-1 Accuracy	Million Mult-Adds	Million Parameters
Inception V3 [18]	84%	5000	23.2
1.0 MobileNet-224	83.3%	569	3.3
0.75 MobileNet-224	81.9%	325	1.9
1.0 MobileNet-192	81.9%	418	3.3
0.75 MobileNet-192	80.5%	239	1.9

EfficientNet

- EfficientNet (Tan, Le, 2019): neural architecture search to design state of the art networks at a fraction of the cost

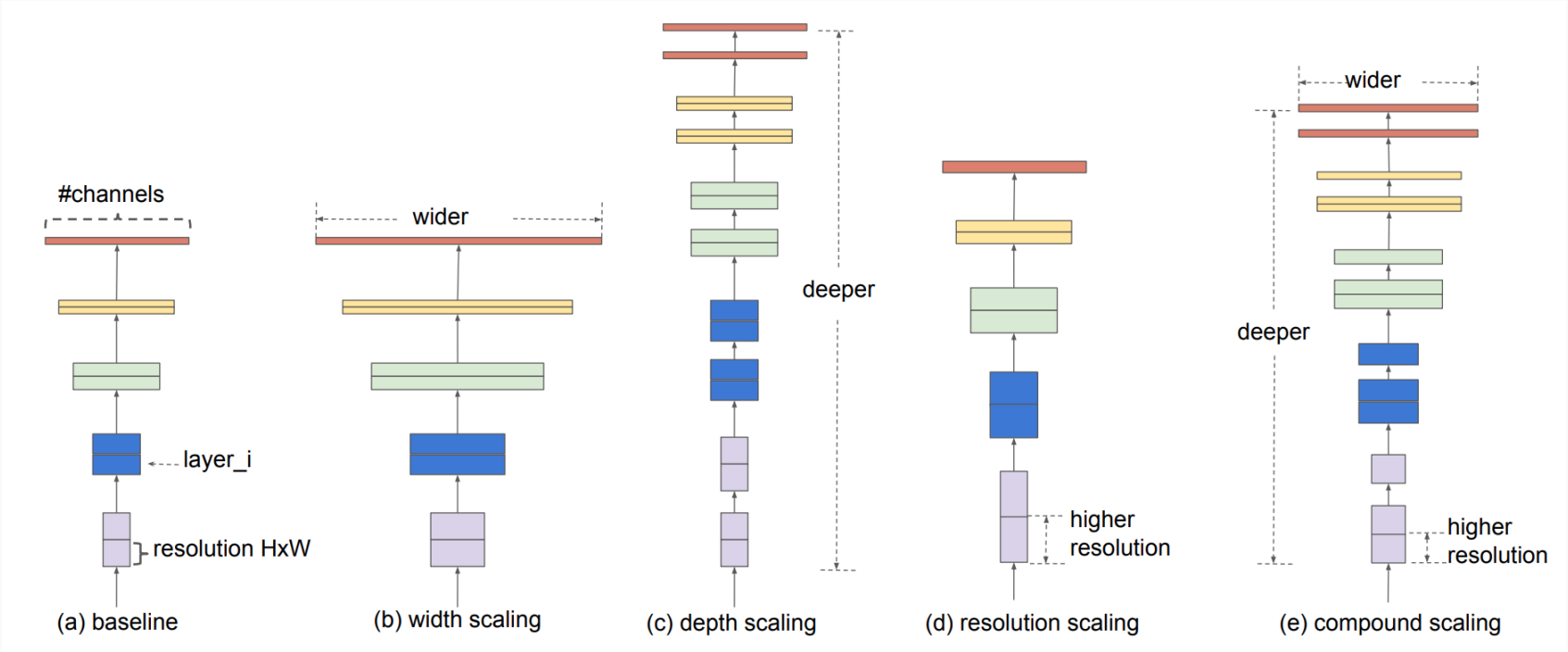


	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

EfficientNet

- Different kinds of scaling for the models:



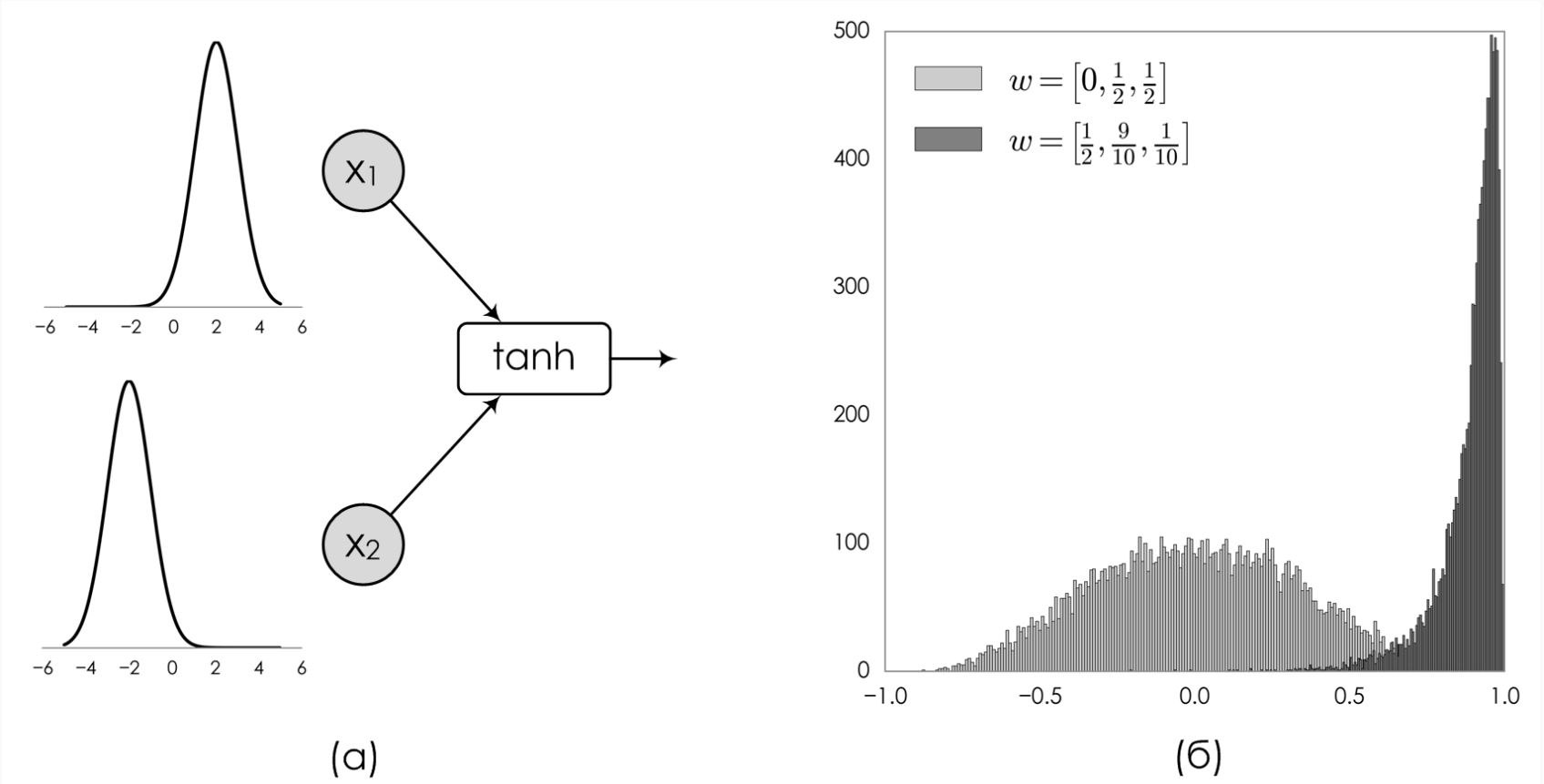
Batch normalization

Batch normalization

- Important problem in deep neural networks: *internal covariate shift*.
- When we change the weights of a layer, the distribution of its outputs changes.
- This means that the next layer has to re-train almost from scratch, it did not expect these outputs.
- Moreover, these neurons might have already reached saturation, so they can't re-train quickly.
- This seriously impedes learning.

Batch normalization

- A characteristic example; note how different the distributions are:



- What can we do?

Batch normalization

- We could try to normalize (whiten) after every layer.
- Does not work: consider a layer that simply adds a bias b to its inputs $\mathbf{u}$:

$$\hat{\mathbf{x}} = \mathbf{x} - \mathbb{E}[\mathbf{x}], \text{ where } \mathbf{x} = \mathbf{u} + b.$$

- On the next gradient descent step, we'll have $b := b + \Delta b$...
- ...but $\hat{\mathbf{x}}$ will not change:

$$\mathbf{u} + b + \Delta b - \mathbb{E}[\mathbf{u} + b + \Delta b] = \mathbf{u} + b - \mathbb{E}[\mathbf{u} + b].$$

- So the biases will simply increase unboundedly, and that's all the training we'll get; not a good thing.

Batch normalization

- We can try to add normalization as a layer:

$$\hat{\mathbf{x}} = \mathrm{Norm}(\mathbf{x}, X).$$

- But note that the entire dataset X is required here.
- So on the gradient descent step we'll need to compute $\frac{\partial \mathrm{Norm}}{\partial \mathbf{x}}$ and $\frac{\partial \mathrm{Norm}}{\partial X}$, and also the covariance matrix

$$\mathrm{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in X}\left[\mathbf{x}\mathbf{x}^\top\right] - \mathbb{E}\left[\mathbf{x}\right]\mathbb{E}\left[\mathbf{x}\right]^\top.$$

- Definitely won't work.

Batch normalization

- The solution is to normalize each component separately, and not over the whole dataset but over the current mini-batch; hence *batch normalization*.
- After batch normalization we get

$$\hat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\mathrm{Var}[x_k]}},$$

  where the statistics are computed over the current mini-batch.
- However, one more problem: now nonlinearities disappear!
- E.g., we will almost always get into the region where σ is very close to linear.

Batch normalization

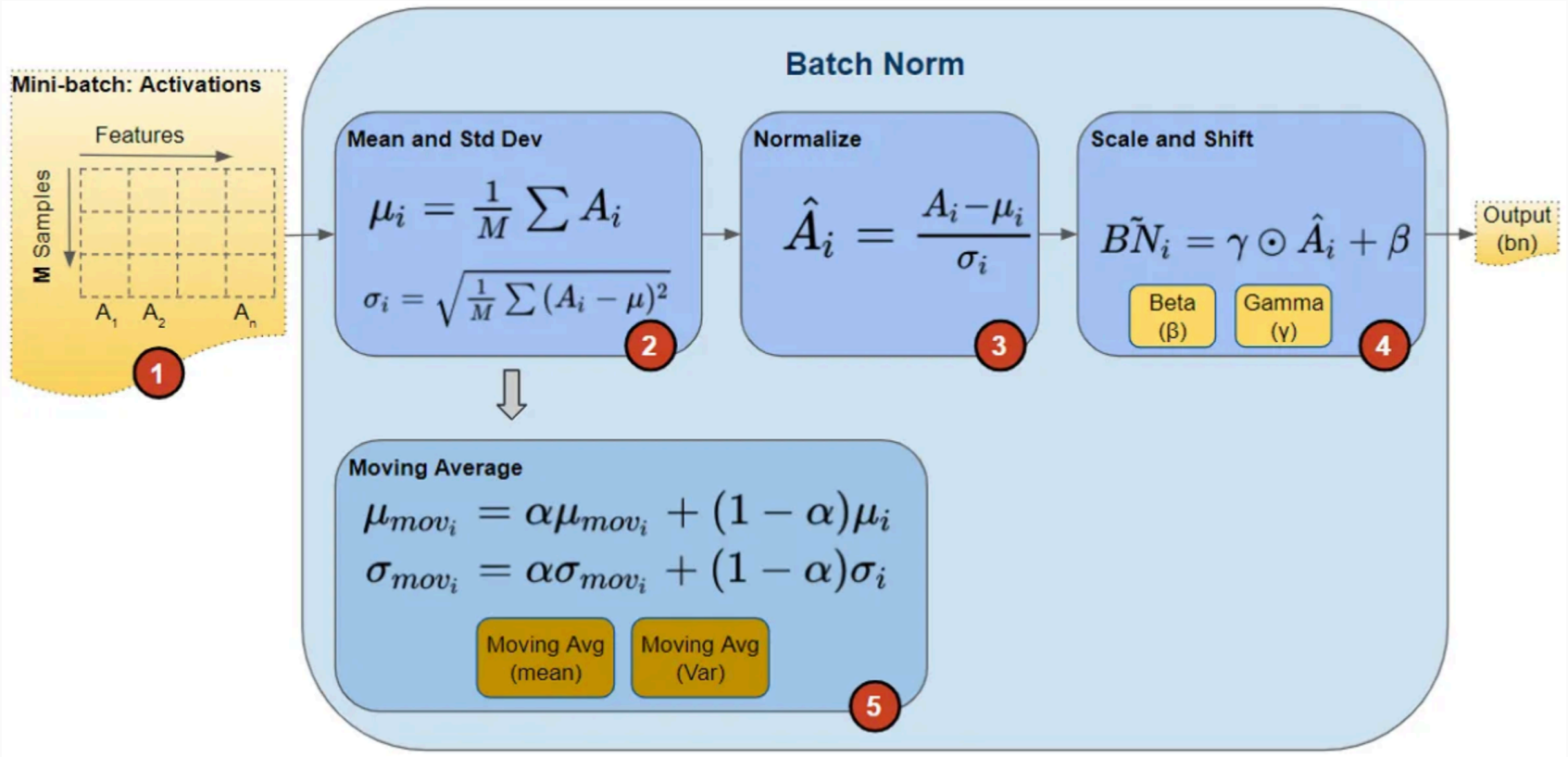
- To fix this, we have to allow the batchnorm layer enough flexibility to sometimes do *nothing* with the inputs.
- So we introduce additional shift and scale parameters:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\mathrm{Var}[x_k]}} + \beta_k.$$

- γ_k and β_k are new variables and will be trained just like the weights.

Batch normalization

- Final scheme:



What BN actually does

- The original idea was to reduce internal covariate shift.
- But it turned out that BN's effectiveness does not really depend on this; still, BN helps with regularization, makes dropout unnecessary, and so on. Possible reasons:
  - BN smooths the gradients, reducing the Lipschitz constant (Santurkar et al., 2019);
  - BN decouples the learning of the direction and the length of the weight vectors, which improves training;
  - there are results showing that BN helps achieve linear convergence in ordinary gradient descent.

Variants

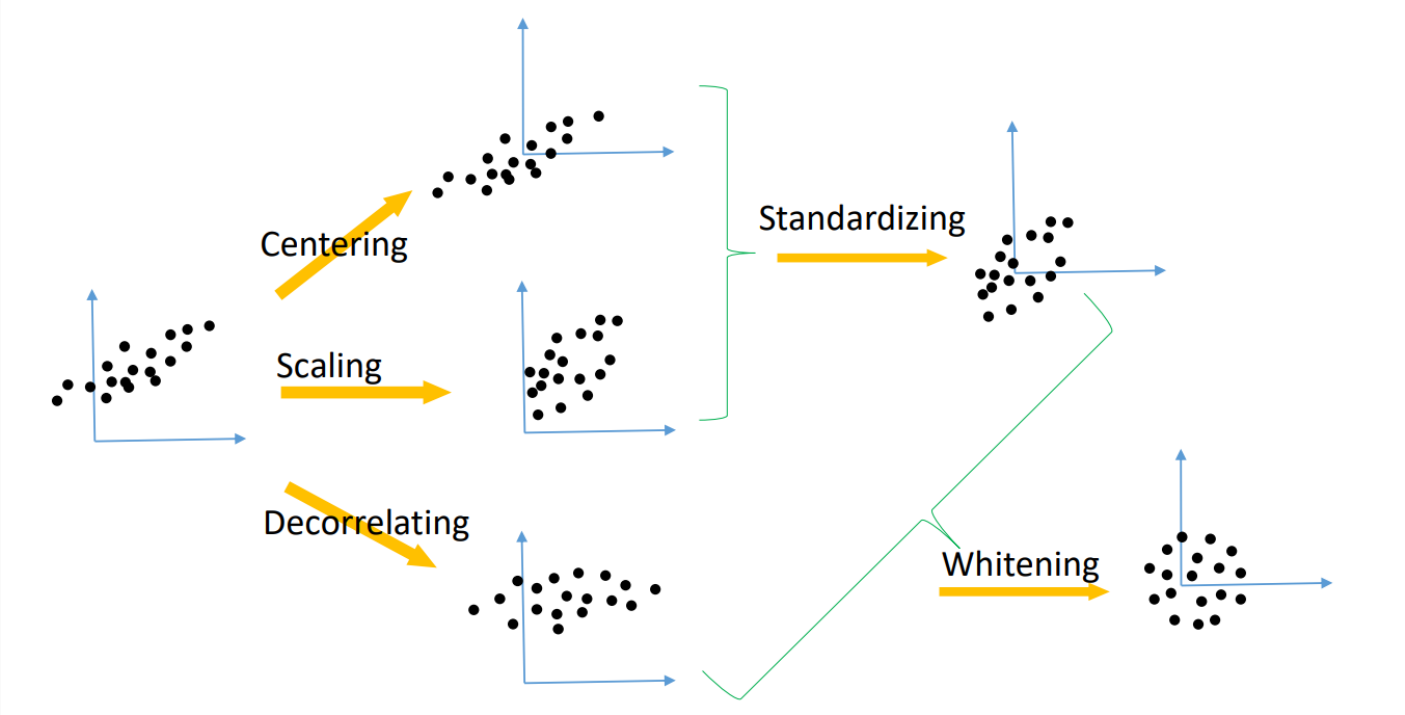
- Mini-batch normalization has by now become essentially the standard.
- Yet another very cool story, much like dropout.
- But the idea goes further:
  - (Laurent et al., 2016): BN does not help recurrent networks;
  - (Cooijmans et al., 2016): recurrent batch normalization;
  - (Salimans and Kingma, 2016): weight normalization to improve training — let us add the weights as

$$\mathbf{h}_i = f\left(\frac{\gamma}{\|\mathbf{w}_i\|}\mathbf{w}_i^\top \mathbf{x} + b_i\right);$$

    then we rescale the gradient, stabilize its norm, and bring its covariance matrix closer to the identity, which improves training.
- Let us make sense of the landscape of normalization methods

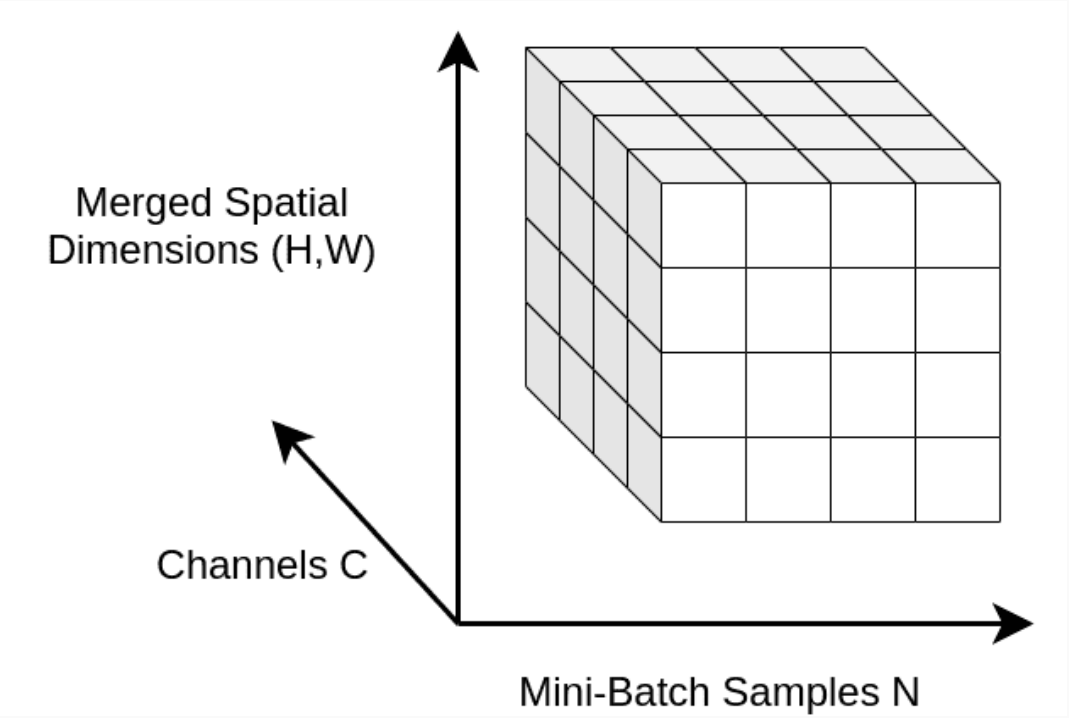
Variants

- In general, all these methods use the same operations:



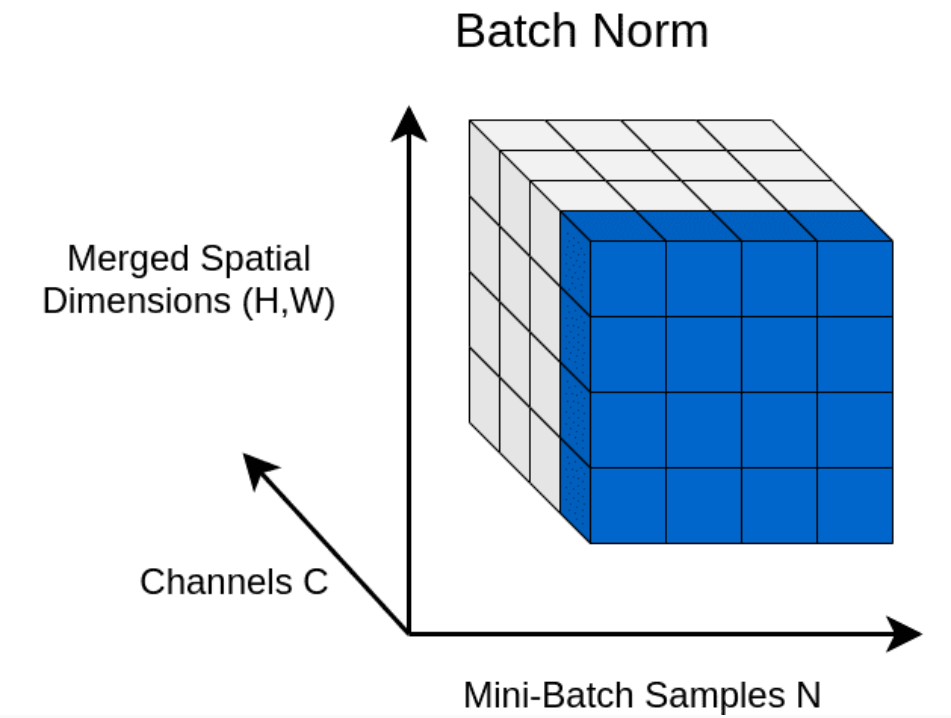
Variants

- and differ in the set of operations and in which dimensions they are applied over:



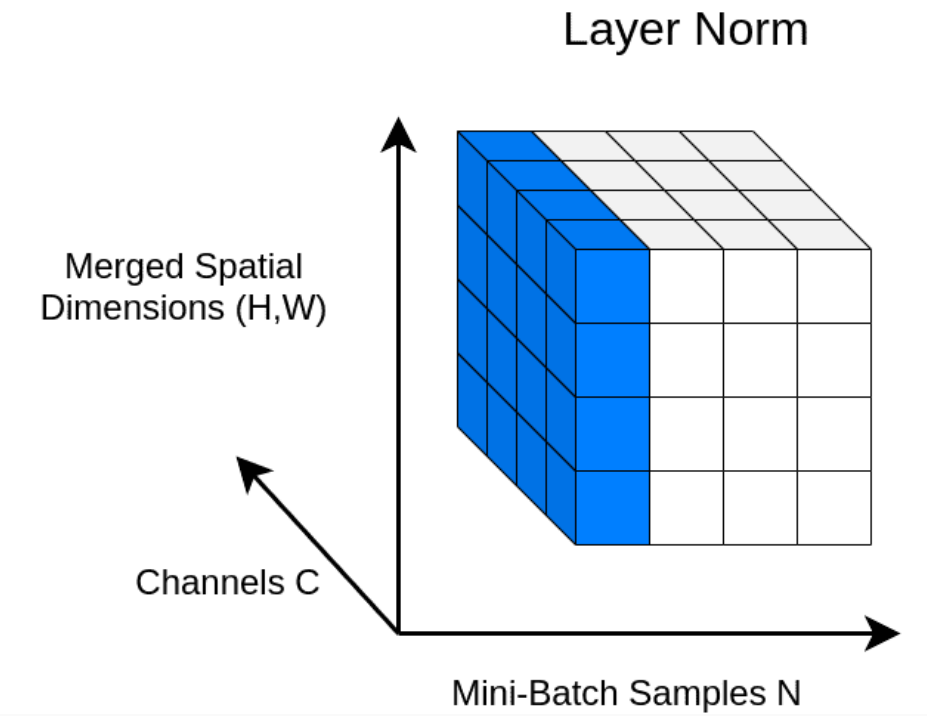
Variants

- Batch norm – over the mini-batch (often used for images):



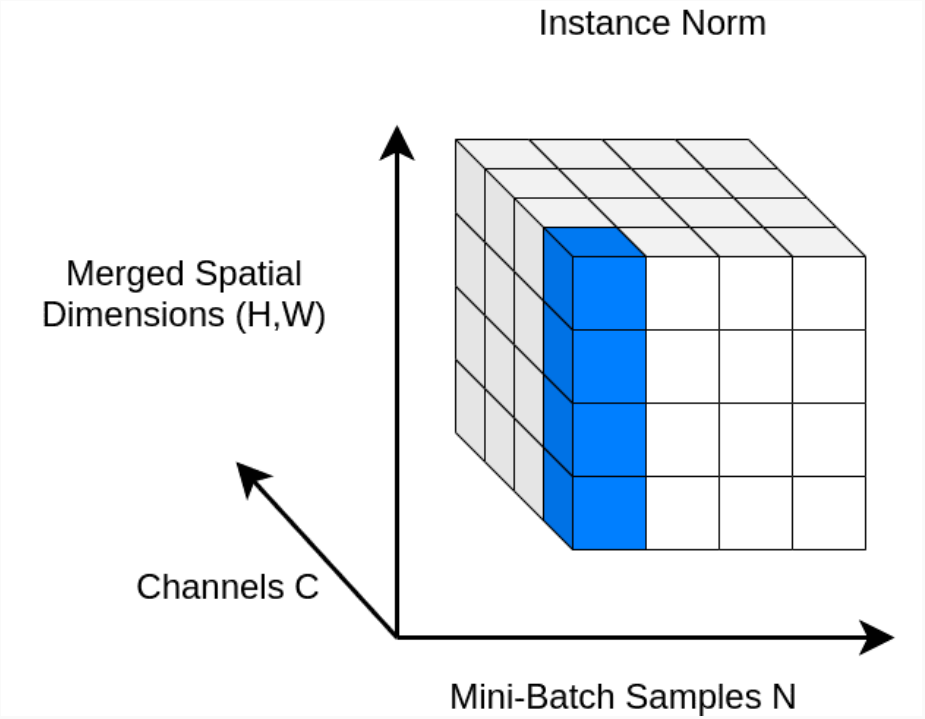
Variants

- Layer norm – over the channels and spatial dimensions, i.e., over all outputs of a layer (used mostly for vectors, in particular in transformers):



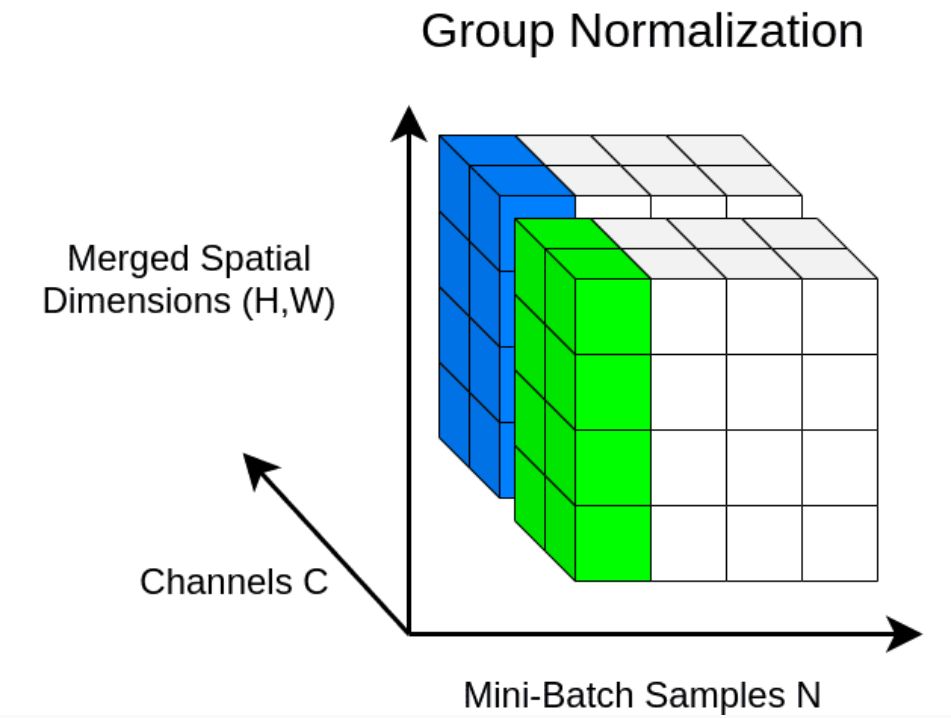
Variants

- Instance norm – only over the spatial dimensions, i.e., we drop the channels (used, e.g., in AdaIN for style transfer):



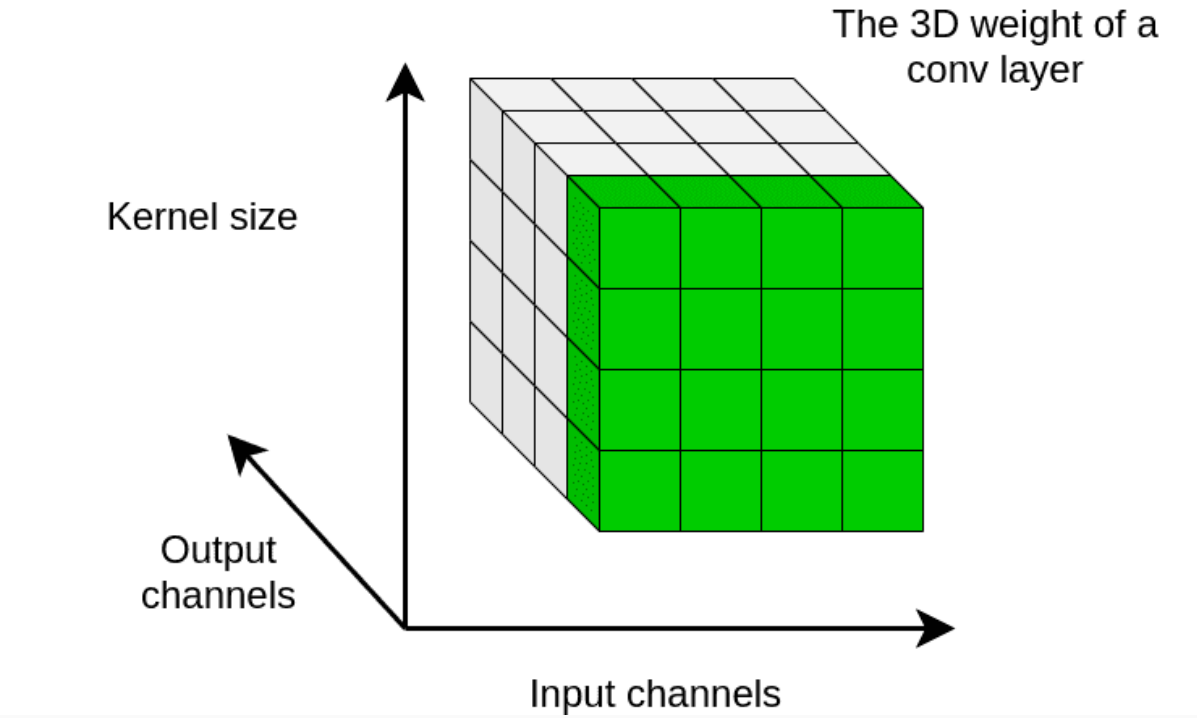
Variants

- Group norm – we can split the channels into groups:



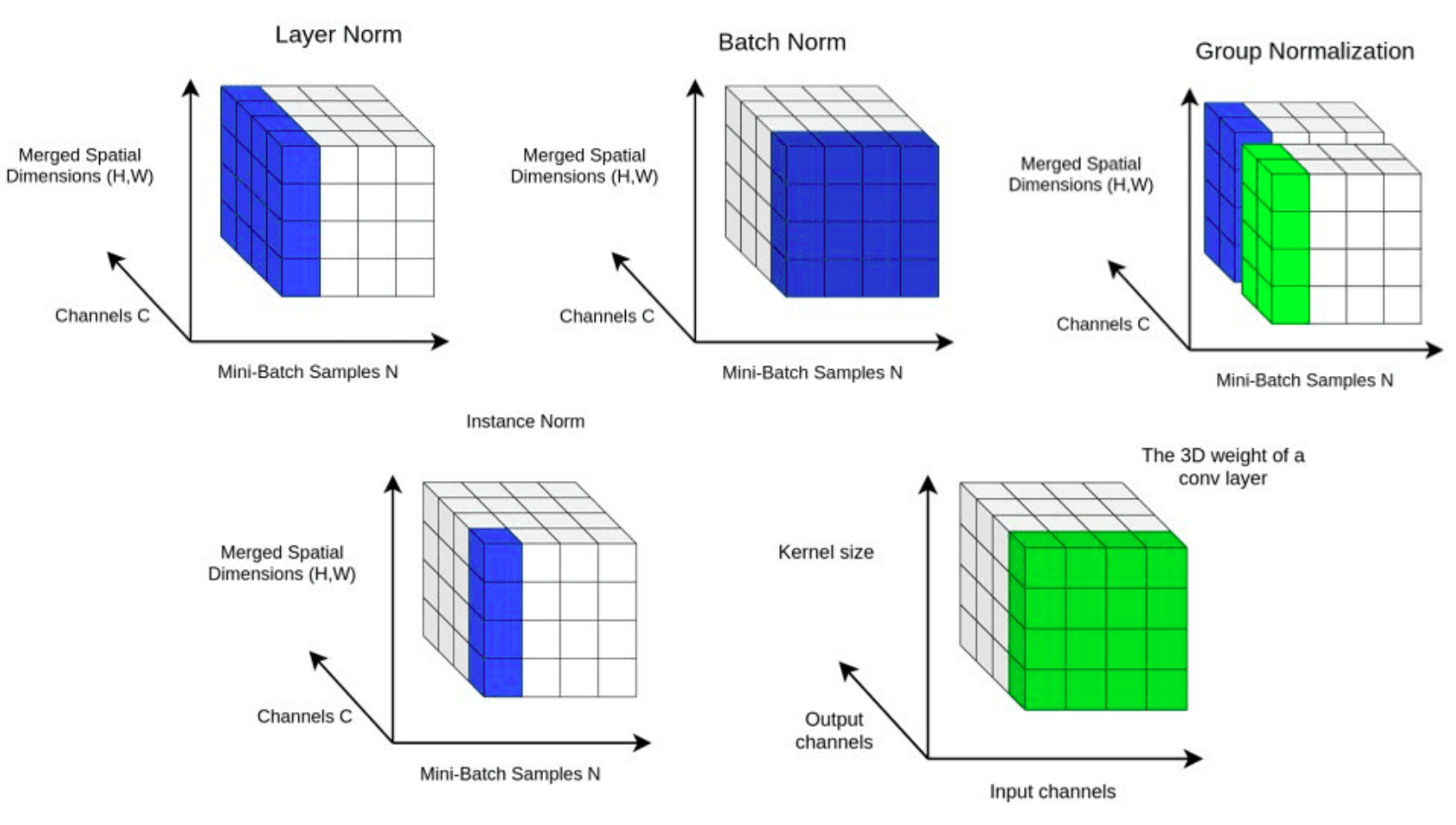
Variants

- Weight standardization – we standardize the weights rather than the activations, separately for each channel, i.e., we also normalize the gradients during backpropagation (group norm + weight standardization were used in BiT – Big Transfer):



Variants

- In summary:



Variants

- But in general this is, as always, a big science (Huang et al., 2020):

Method	NAP	NOP	NRR	Published In
Local contrast normalization [108]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$: local spatial positions	Standardizing	No	ICCV, 2009
Local response normalization [75]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$: local channels	Scaling	No	NeurIPS, 2012
Batch normalization (BN) [8]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICML, 2015
Mean-only BN [21]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Centering	No	NeurIPS, 2016
Layer normalization (LN) [20]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
Instance normalization (IN) [67]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
L^p-Norm BN [90]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing with L^p-Norm divided	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
Divisive normalization [76]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$: local spatial positions and channels	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICLR, 2017
Conditional IN [68]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Side information	ICLR, 2017
Dynamic LN [101]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	INTERSPEECH, 2017
Conditional BN [107]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Side information	NeurIPS, 2017
Pixel normalization [72]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$	Scaling	No	ICLR, 2018
Decorrelated BN [34]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	ZCA whitening	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2018
Group normalization (GN) [22]	$\Pi_{GN}(\mathbf{X}) \in \mathbb{R}^{mg \times shw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2018

Variants

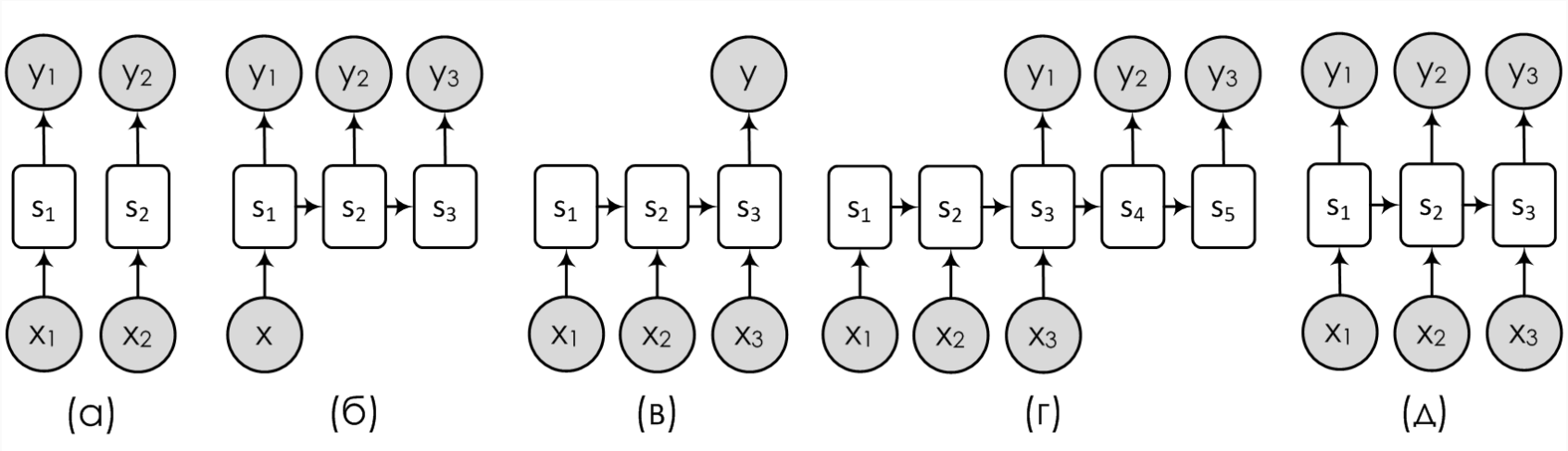
- And really big it is:

Adaptive IN [69]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2018
L^1-Norm BN [91], [92]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing with L^1-Norm divided	Learnable $\gamma, \beta \in \mathbb{R}^d$	NeurIPS, 2018
Whitening and coloring BN [79]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	CD whitening	Color transformation	ICLR, 2019
Generalized BN [93]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	General standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	AAAI, 2019
Iterative normalization [81]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	ZCA whitening by Newton's iteration	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2019
Instance-level meta normalization [103]	$\Pi_{LN}(\mathbf{X})/\Pi_{IN}(\mathbf{X})/\ \Pi_{GN}(\mathbf{X})$	Standardizing	Learnable & generated $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2019
Spatially adaptive denormalization [109]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^{d \times h \times w}$	CVPR, 2019
Position normalization (PN) [71]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	NeurIPS, 2019
Root mean square LN [97]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Scaling	Learnable $\gamma \in \mathbb{R}^d$	NeurIPS, 2019
Online normalization [99]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Scaling	No	NeurIPS, 2019
Batch group normalization [73]	$\Pi_{BGN}(\mathbf{X}) \in \mathbb{R}^{g_m g \times s_m shw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICLR, 2020
Instance enhancement BN [105]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Learnable & generated $\gamma, \beta \in \mathbb{R}^d$	AAAI, 2020
PowerNorm [96]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Scaling	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICML, 2020
Local contex normalization [74]	$\Pi_{GN}(\mathbf{X}) \in \mathbb{R}^{mg \times shw}$: local spatial positions and channels	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2020
Filter response normalization [100]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Scaling	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2020
Attentive normalization [106]	$\Pi_{BN}(\mathbf{X})/\Pi_{IN}(\mathbf{X})/$ $\Pi_{LN}(\mathbf{X})/\Pi_{GN}(\mathbf{X})$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2020

Working with sequences

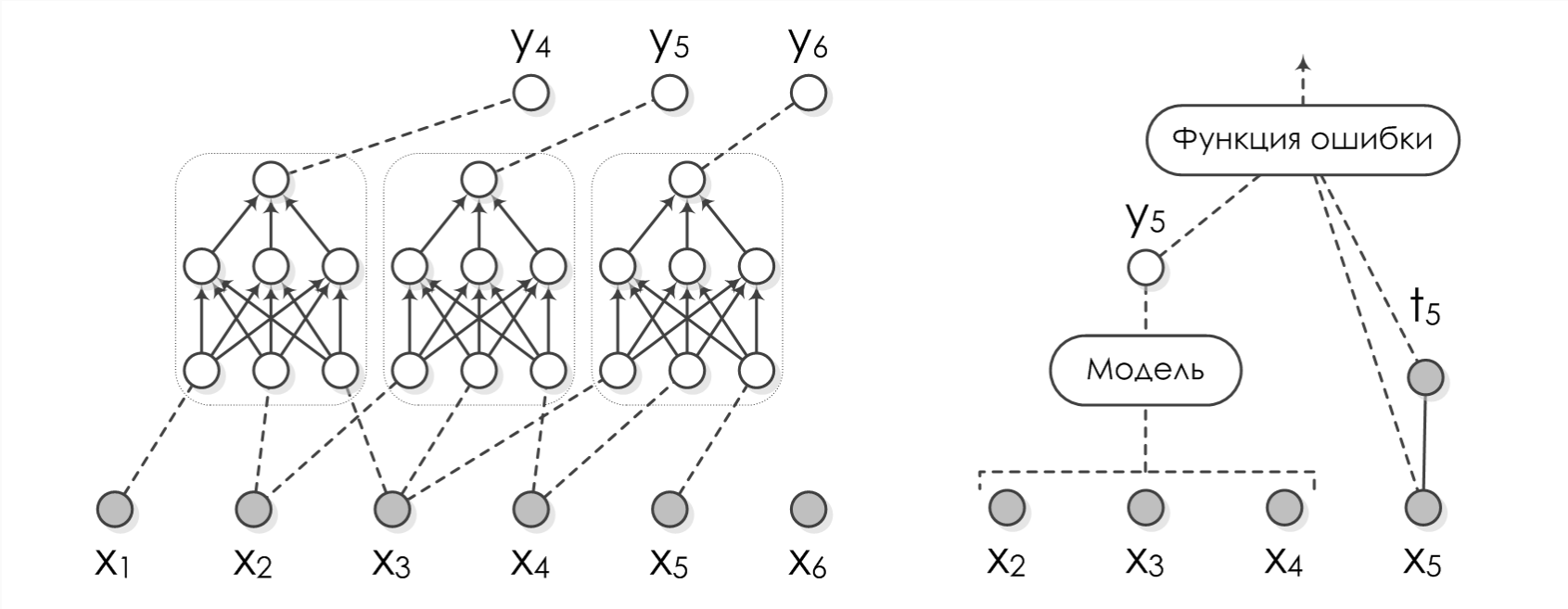
Sequences

- Sequences: financial time series, language, sound...
- Different kinds of sequence-based problems:



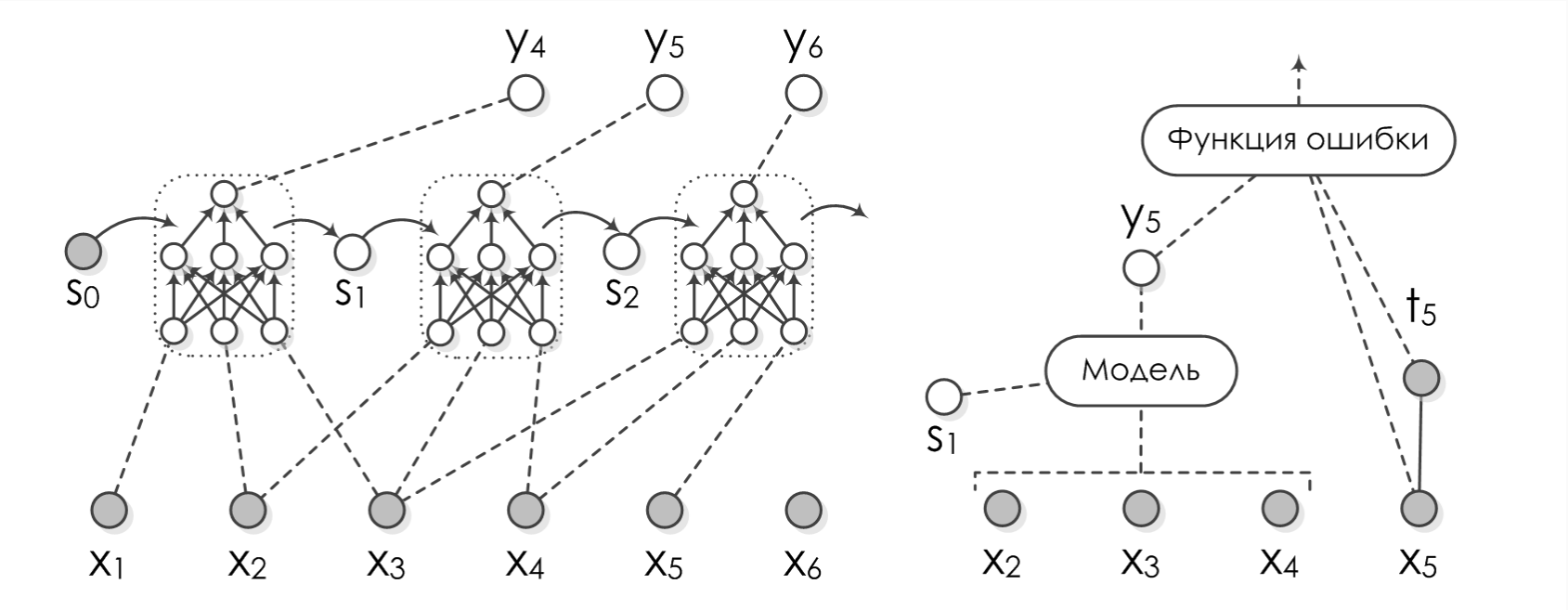
Sequences

- How to we apply a neural network to a sequence-based problem?
- We can try to use a sliding window:



Sequences

- ...but it would be better to have a hidden state and update it.
- This is the whole idea of *recurrent neural networks* (RNN).



- But how do we do backprop? Doesn't the computational graph have cycles now that

$$s_i = h(x_i, x_{i+1}, x_{i+2}, s_{i-1})?$$

Sequences

- ...not really. We can "unroll" it back:

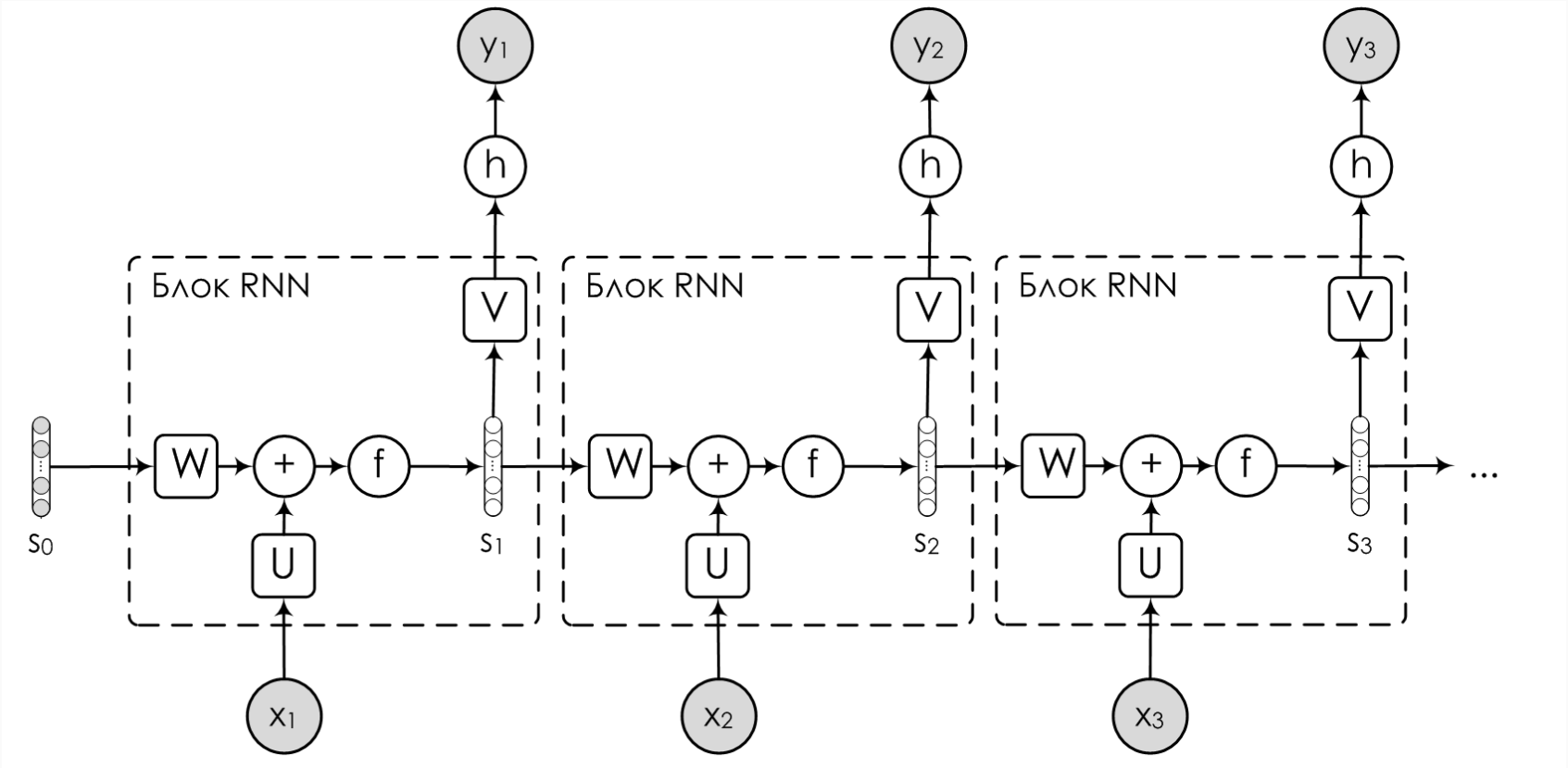
$$y_6 = f(x_3, x_4, x_5, s_2) = f(x_3, x_4, x_5, h(x_2, x_3, x_4, s_1)) = \\ = f(x_3, x_4, x_5, h(x_2, x_3, x_4, h(x_1, x_2, x_3, s_0))).$$

- So formally there is no problem.
- But, of course, lots of practical issues – it's a *very* deep network with lots of shared weights...

Recurrent Neural Networks

Simple RNN

- “Simple” RNN:



Simple RNN

- Formally:

$$\begin{aligned}\mathbf{a}_t &= \mathbf{b} + W\mathbf{s}_{t-1} + U\mathbf{x}_t,\\ \mathbf{s}_t &= f(\mathbf{a}_t),\\ \mathbf{o}_t &= \mathbf{c} + V\mathbf{s}_t,\\ \mathbf{y}_t &= h(\mathbf{o}_t),\end{aligned}$$

  where f is the recurrent nonlinearity, and h is the output function.

- Two problems:
  - exploding gradients;
  - vanishing gradients.

Exploding gradients

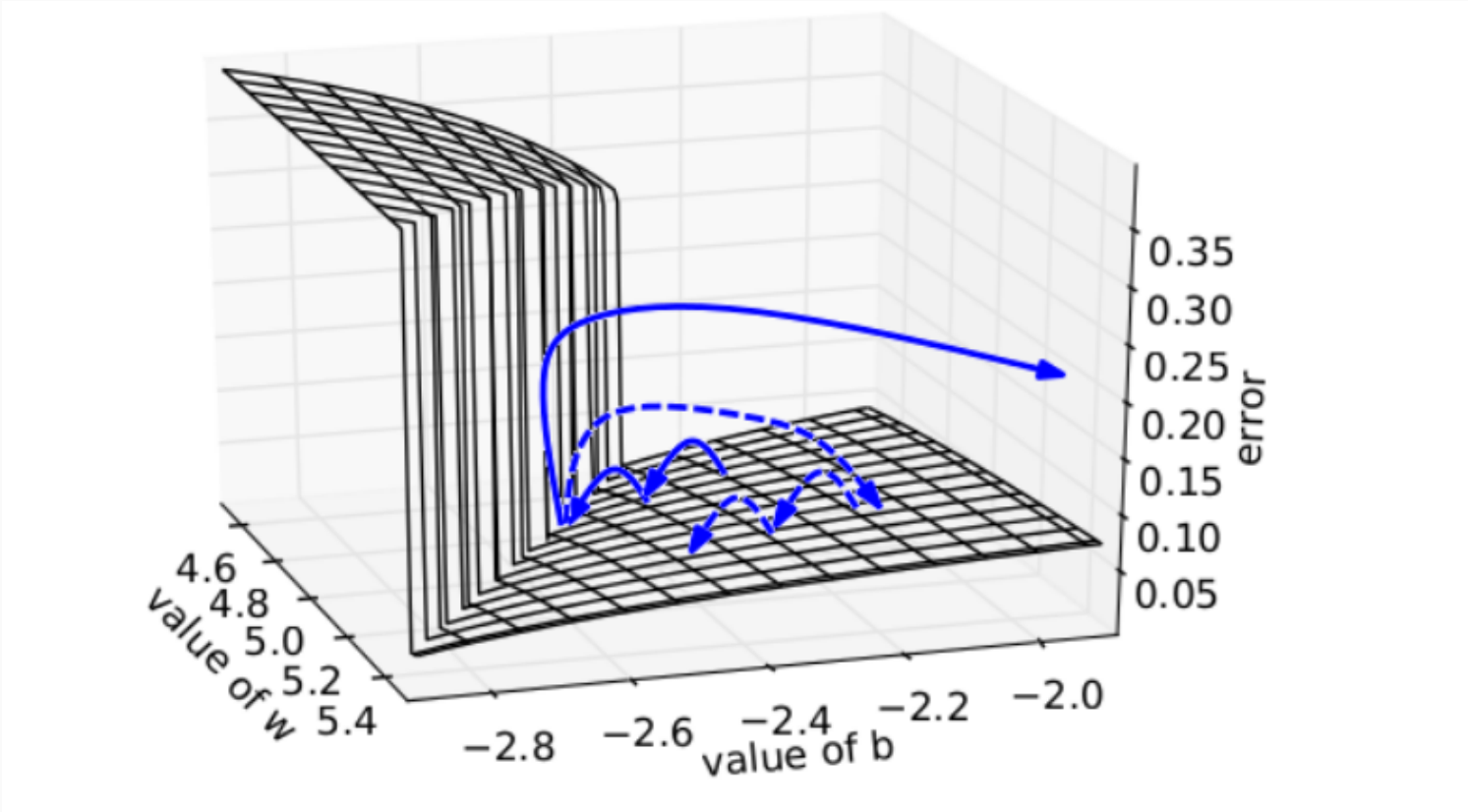
- We multiply by the same W, and the norm of the gradient grows or decreases exponentially.
- What do we do when it explodes?

Exploding gradients

- We can simply bound the gradients from above!
- Two versions — bound either the norm or every value:
  - `sgd = optimizers.SGD(lr=0.01, clipnorm=1.)`
  - `sgd = optimizers.SGD(lr=0.01, clipvalue=.05)`

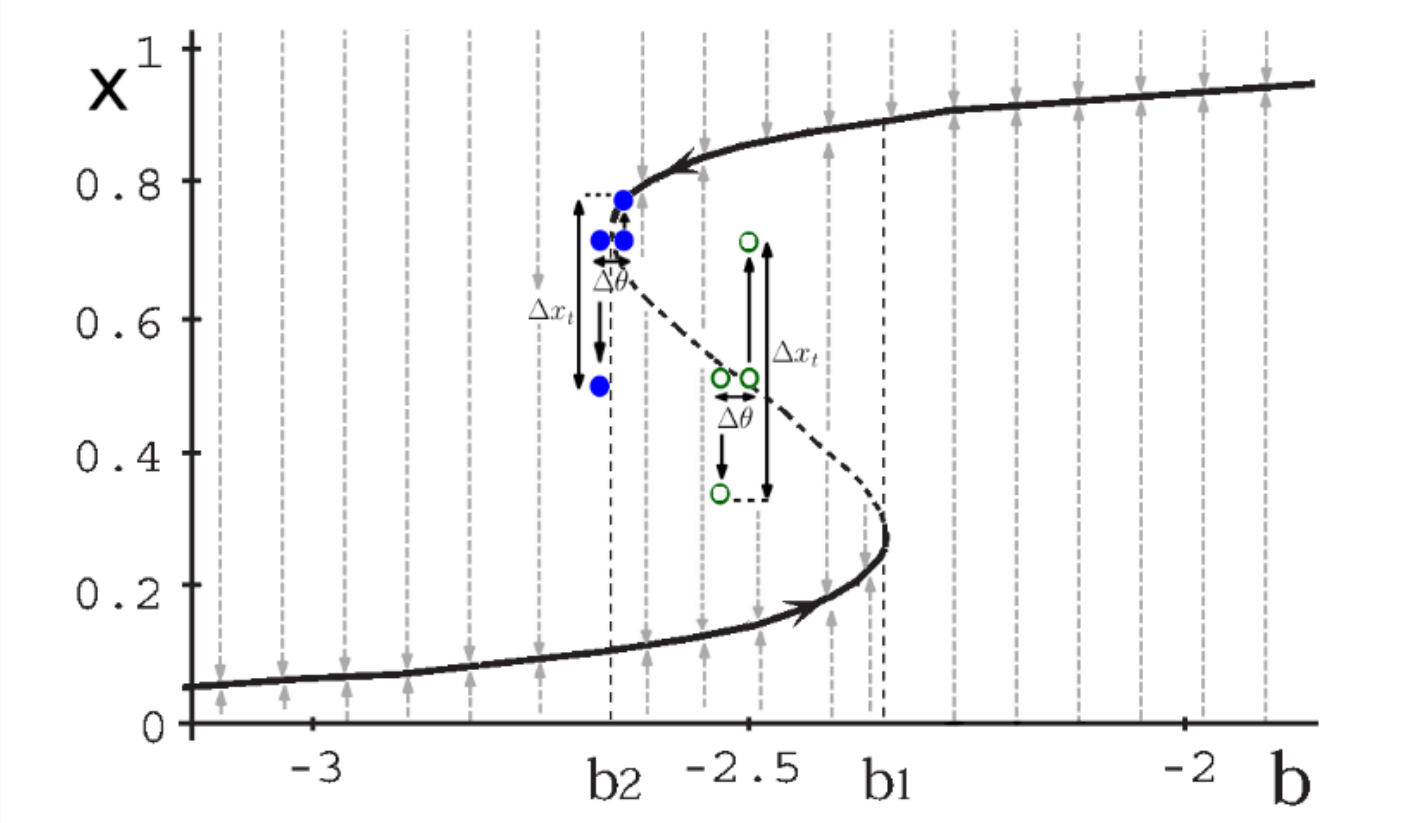
Exploding gradients

- (Pascanu et al., 2013) – nice pictures about it:



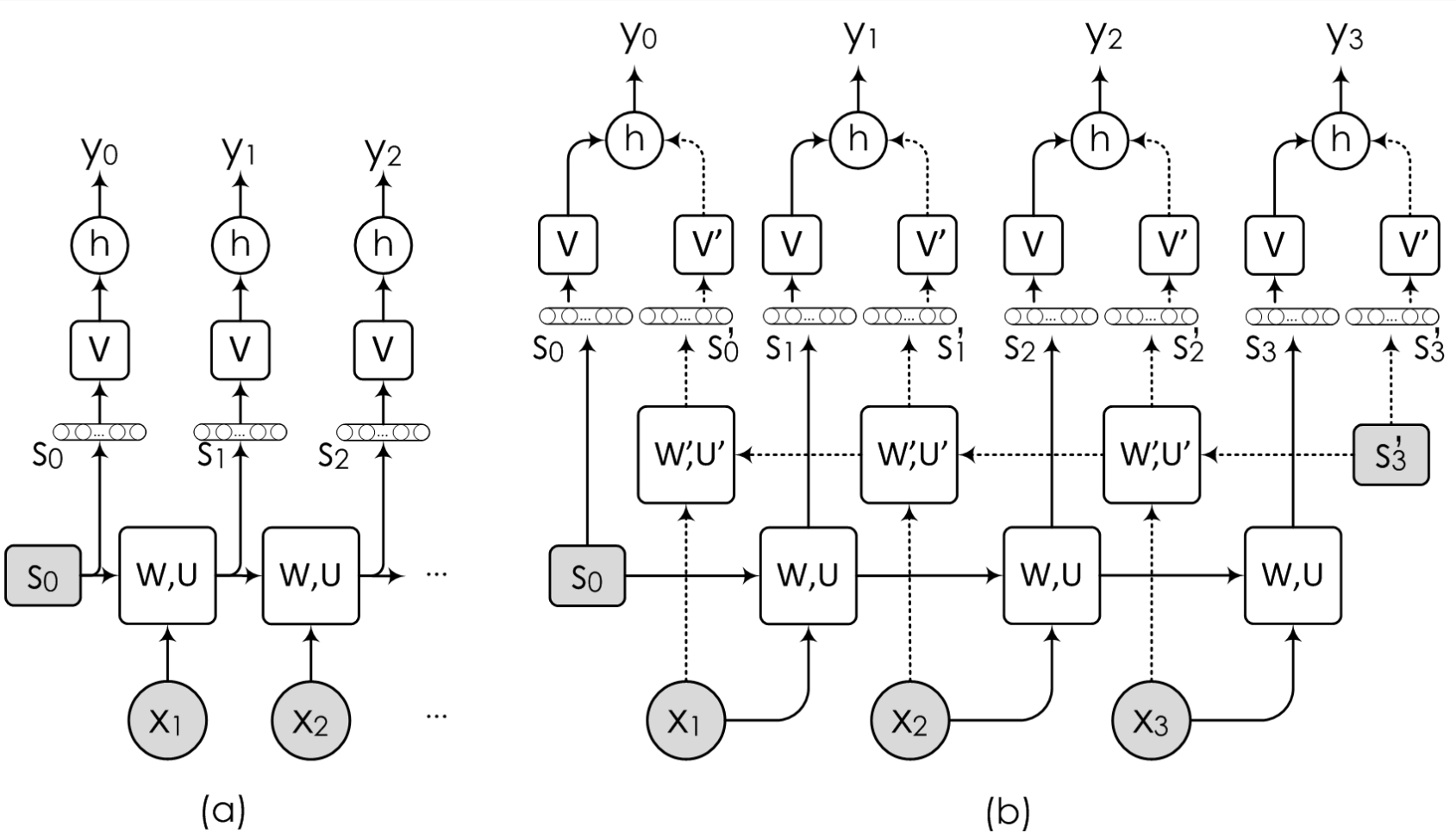
Exploding gradients

- Also explains where such gaps come from – RNNs have "bifurcation points":



Bidirectional RNN

- Sometimes we need context from both directions:



Bidirectional RNN

- Formally:

$$\begin{aligned}
\mathbf{s}_t &= \sigma\left(\mathbf{b} + W\mathbf{s}_{t-1} + U\mathbf{x}_t\right), \\
\mathbf{s}'_t &= \sigma\left(\mathbf{b}' + W'\mathbf{s}'_{t+1} + U'\mathbf{x}_t\right), \\
\mathbf{o}_t &= \mathbf{c} + V\mathbf{s}_t + V'\mathbf{s}'_t, \\
\mathbf{y}_t &= h\left(\mathbf{o}_t\right).
\end{aligned}$$

- This, of course, generalizes to any other sort of constructions.

LSTM and GRU

LSTM

- Vanishing gradients: we have to multiply by W every time.
- This makes it impossible to have long-term memory.

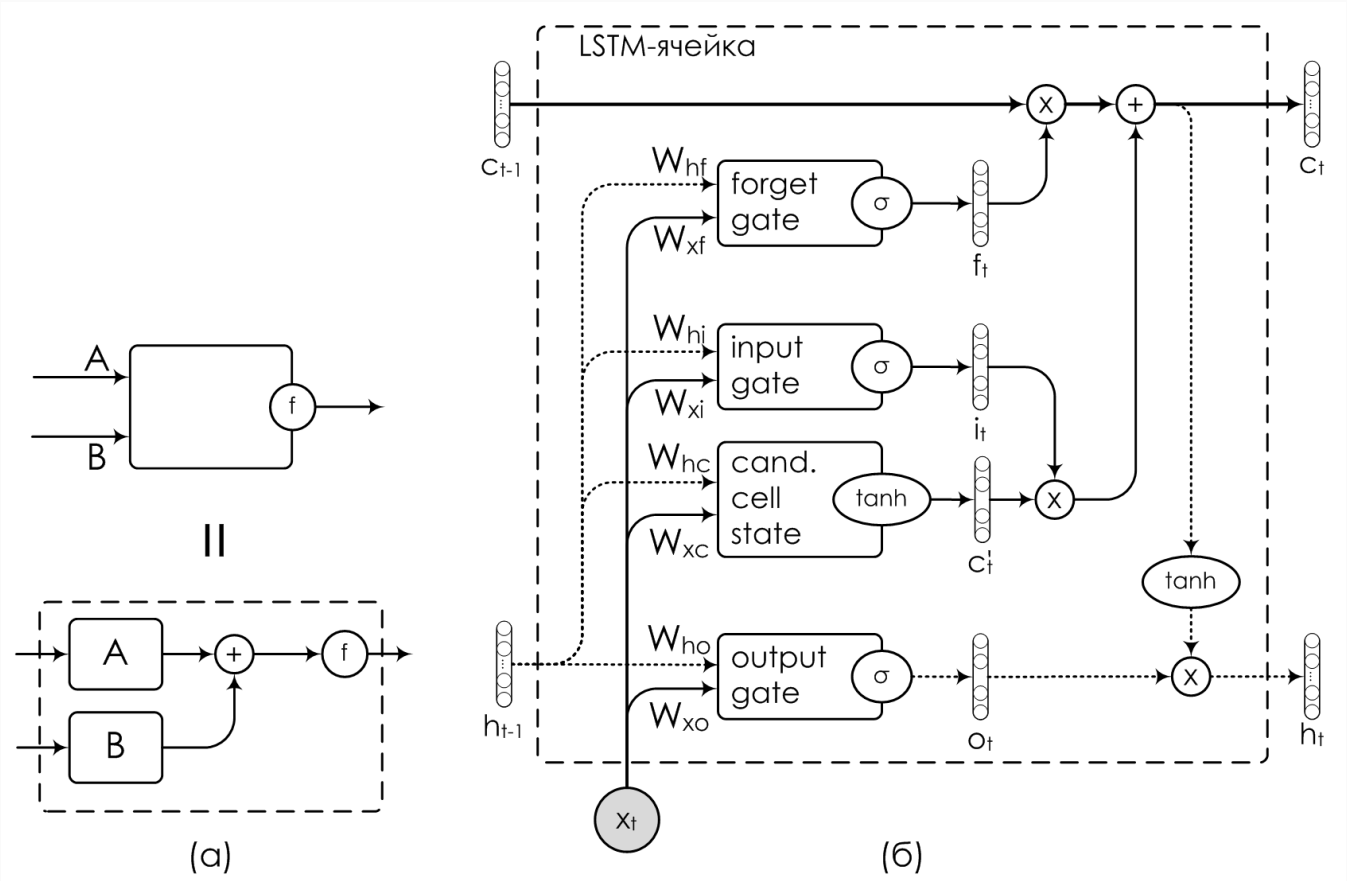
- What do we do?..

LSTM

- ...we have seen the basic idea in ResNet: we have to provide the "constant error carousel", a shortcut for the gradients.
- Idea from the mid-1990s (Schmidhuber): let's construct RNNs from a more complex unit that has the shortcut and controls memory explicitly.
- LSTM (long short-term memory).

LSTM

- "Vanilla" LSTM: $\mathbf{c}_t$ is the cell state, and $\mathbf{h}_t$ is the hidden state.
- Input gate and forget gate control whether we change $\mathbf{c}_t$ to the new candidate cell state.



LSTM

- Formally:

$$
\begin{aligned}
\mathbf{c}'_t &= \tanh\left(W_{xc}\mathbf{x}_t + W_{hc}\mathbf{h}_{t-1} + \mathbf{b}_{c'}\right) && \textit{candidate cell state} \\
\mathbf{i}_t &= \sigma\left(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i\right) && \textit{input gate} \\
\mathbf{f}_t &= \sigma\left(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f\right) && \textit{forget gate} \\
\mathbf{o}_t &= \sigma\left(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o\right) && \textit{output gate} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t, && \textit{cell state} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \textit{block output}
\end{aligned}
$$

- So the LSTM cell is able to control the cell state with the hidden state and weights.
- Very flexible, and if the forget gate is closed ($\mathbf{f}_t = 1$), we have

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t, \text{ so } \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = 1.$$

- The constant error carousel!

LSTM

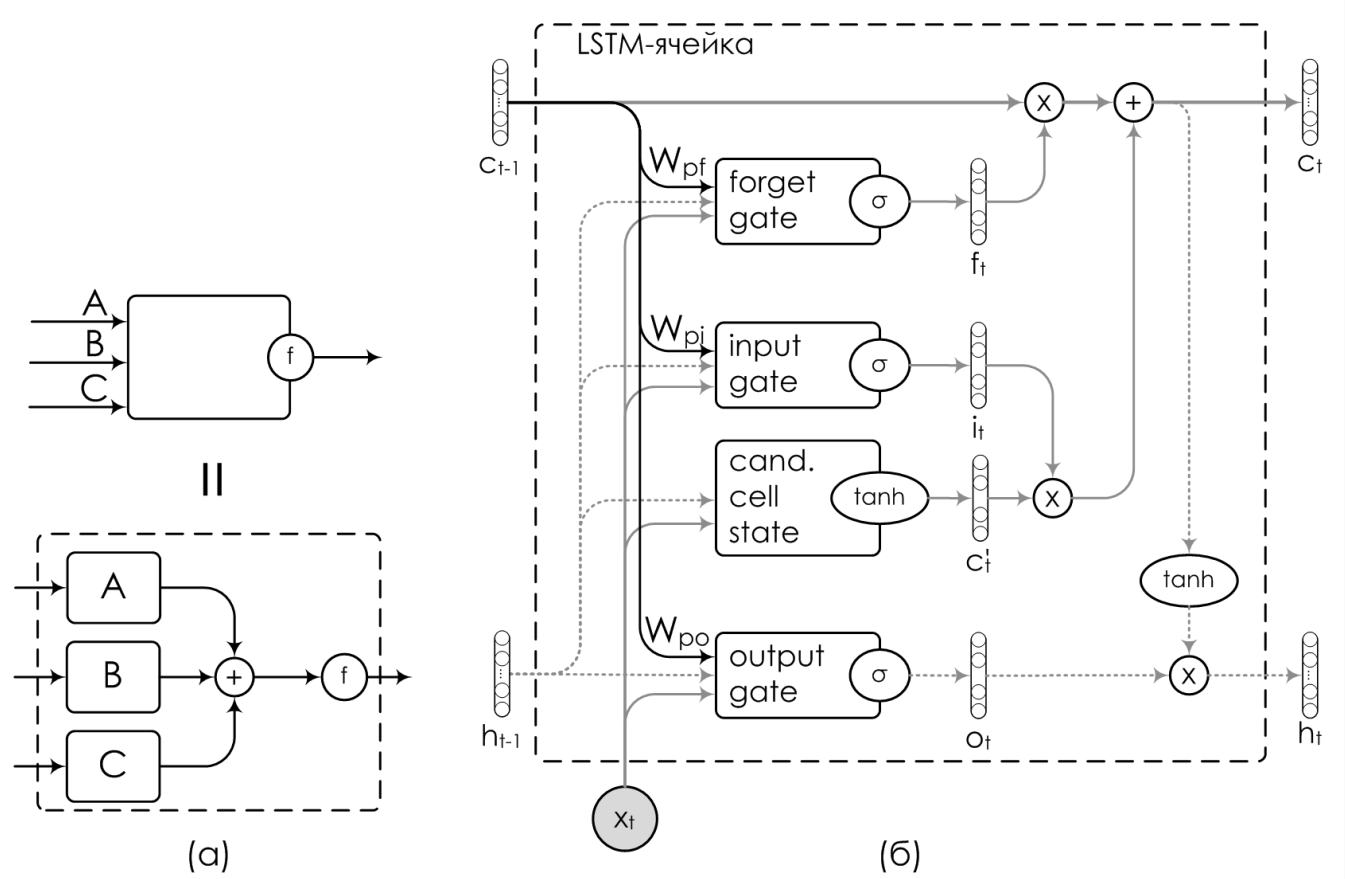
- LSTM was developed in mid-1990s (Hochreiter and Schmidhuber, 1995; 1997).
- Completely in its modern form in (Gers, Schmidhuber, 2000).
- One problem: we want to control $\mathbf{c}$, but it's unavailable for the gates! They only see $\mathbf{h}_{t-1}$, which is

$$\mathbf{h}_{t-1} = \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}).$$

- So if the output gate is closed the behaviour of LSTM does not depend on cell state at all.
- This is not good...

LSTM

- ...so of course we add a few more weight matrices (peepholes).



LSTM

- Formally:

$$\begin{aligned}
\mathbf{i}_t &= \sigma\left(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + W_{pi}\mathbf{c}_{t-1} + \mathbf{b}_i\right) \\
\mathbf{f}_t &= \sigma\left(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + W_{pf}\mathbf{c}_{t-1} + \mathbf{b}_f\right) \\
\mathbf{o}_t &= \sigma\left(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + W_{po}\mathbf{c}_{t-1} + \mathbf{b}_o\right)
\end{aligned}$$

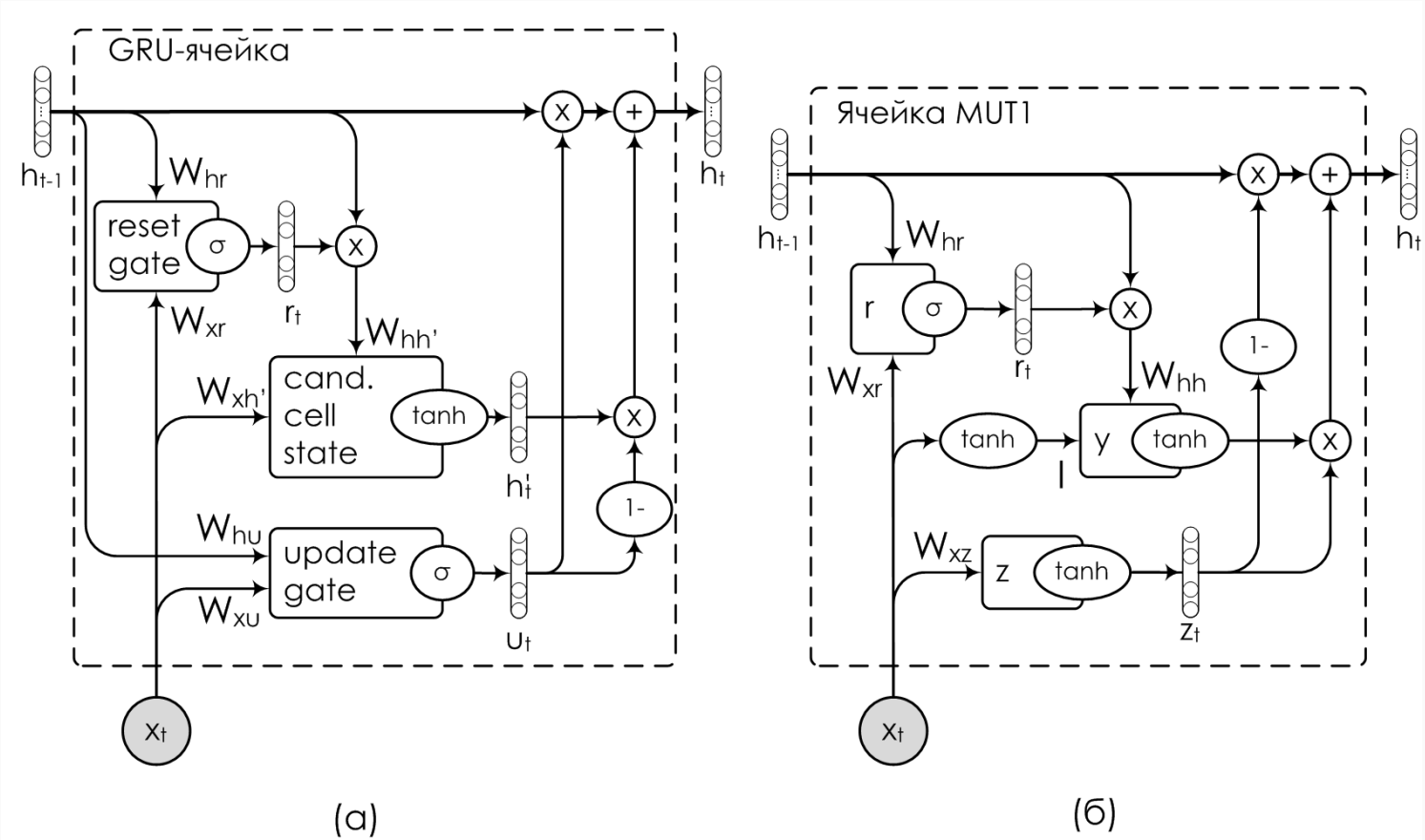
- Huge number of LSTM variations: we can basically remove any kind of gate, add or remove each peephole, add or remove activation functions etc.
- How do we choose?

LSTM

- «LSTM: a Search Space Odyssey» (Greff et al., 2015).
- Shows an experimental comparison of many variations.
- In particular, some significantly simpler architectures (with one less gate!) did not lose to the vanilla LSTM much.
- This brings us to...

GRU

- ...Gated Recurrent Units (GRU); developed in (Cho et al., 2014).
- Also implements the constant error carousel, but simpler than LSTM.



GRU

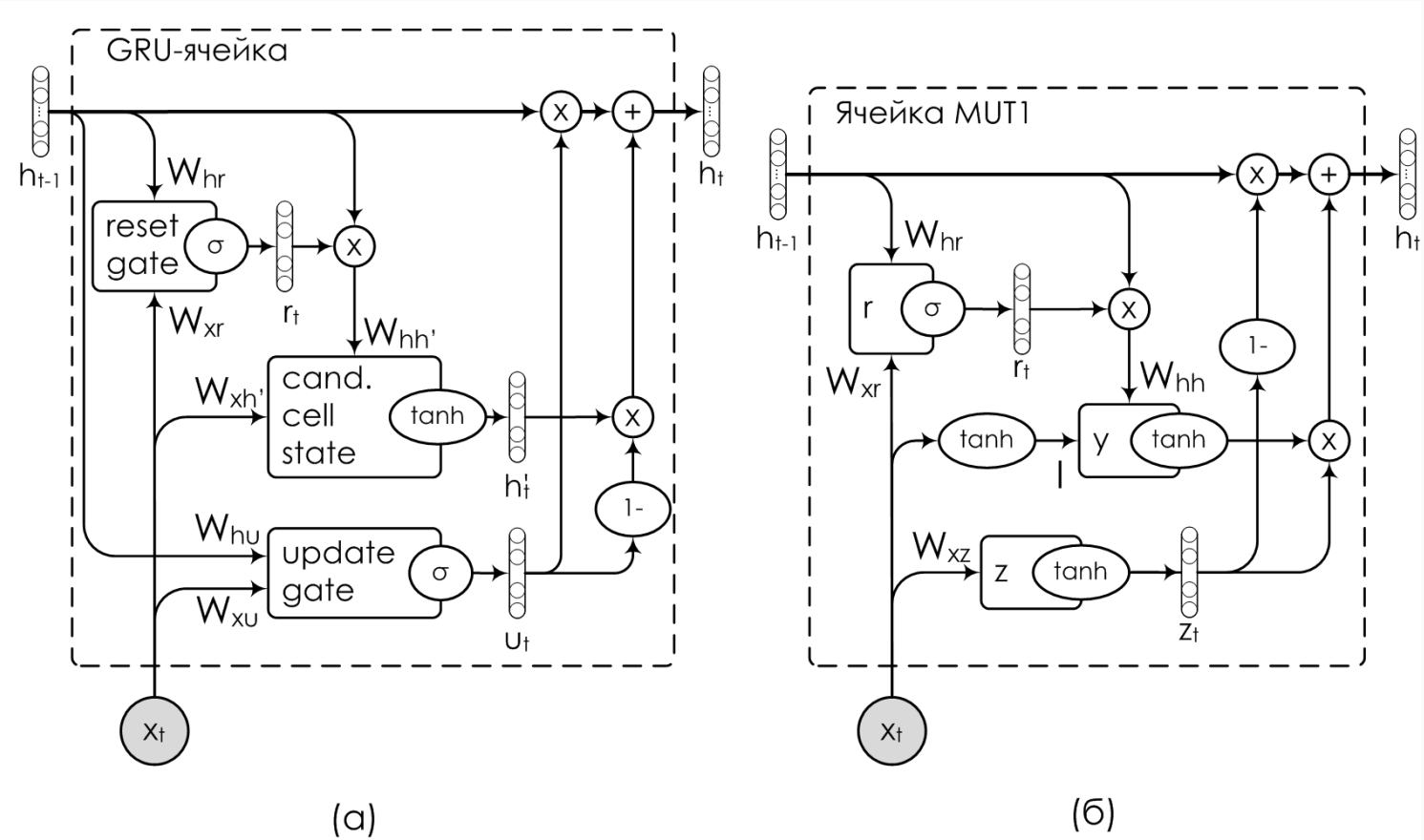
- Formally:

$$\begin{aligned}
\mathbf{u}_t &= \sigma(W_{xu}\mathbf{x}_t + W_{hu}\mathbf{h}_{t-1} + \mathbf{b}_u) \\
\mathbf{r}_t &= \sigma(W_{xr}\mathbf{x}_t + W_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) \\
\mathbf{h}'_t &= \tanh(W_{xh'}\mathbf{x}_t + W_{hh'}(\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}'_t + \mathbf{u}_t \odot \mathbf{h}_{t-1}
\end{aligned}$$

- Update gate and reset gate; there is no distinction between $\mathbf{c}_t$ and $\mathbf{h}_t$.
- Fewer matrices (6 vs. 8 or 11 with peepholes), fewer weights, but only very slightly worse than LSTM.
- So you can fit more GRUs and have better networks on a given computational budget.

GRU

- There are other variations too.
- (Józefowicz, Zaremba, Sutskever, 2015): huge experimental comparison, with an evolutionary approach.
- Identified three interesting architectures.



Thank you for your attention!