

# Adversarial examples и другие идеи

---

Сергей Николенко

НИУ ВШЭ – Санкт-Петербург

10 октября 2020 г.

---

*Random facts:*

- 10 октября в ООН — Всемирный день психического здоровья; каждый год новая тема, обычно разные (в 2018 — «Молодые люди и психическое здоровье в изменяющемся мире», в 2019 — «Продвижение психического здоровья и предотвращение суицида»), но в 2020 проблем стало так много, что всё проще: «Move for mental health: Increased investment in mental health»
- 10 октября 1853 г. впервые встретились Рихард Вагнер и Ференц Лист; это произошло по инициативе 15-летней дочери Листа Козимы, которая вскоре вышла замуж за Ханса фон Бюлова, а через 17 лет, после долгой мыльной оперы — за Вагнера
- 10 октября 1865 г. Джон Хайат нашёл-таки замену слоновой кости и запатентовал билльярдный шар из целлулоида
- 10 октября 1874 г. в России было объявлено первое в истории штормовое предупреждение на Балтийском море
- 10 октября 1993 г. в 21:00 впервые вышел в эфир телеканал НТВ, а 10 октября 2006 г. начала работу социальная сеть ВКонтакте

# Современные свёрточные архитектуры

---

- Разных моделей, использующих CNN, очень много, и всё время появляются новые.
- Всё можно скачать, часто даже уже веса, предобученные на стандартных датасетах, есть.
- Но обычно они основаны на нескольких относительно стандартных идеях. Их-то мы и рассмотрим.

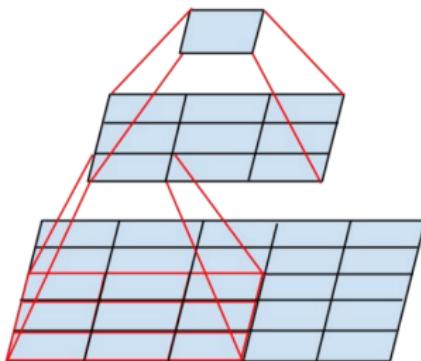
## Model Zoo

Discover open source deep learning code and pretrained models.

[Browse Frameworks](#)

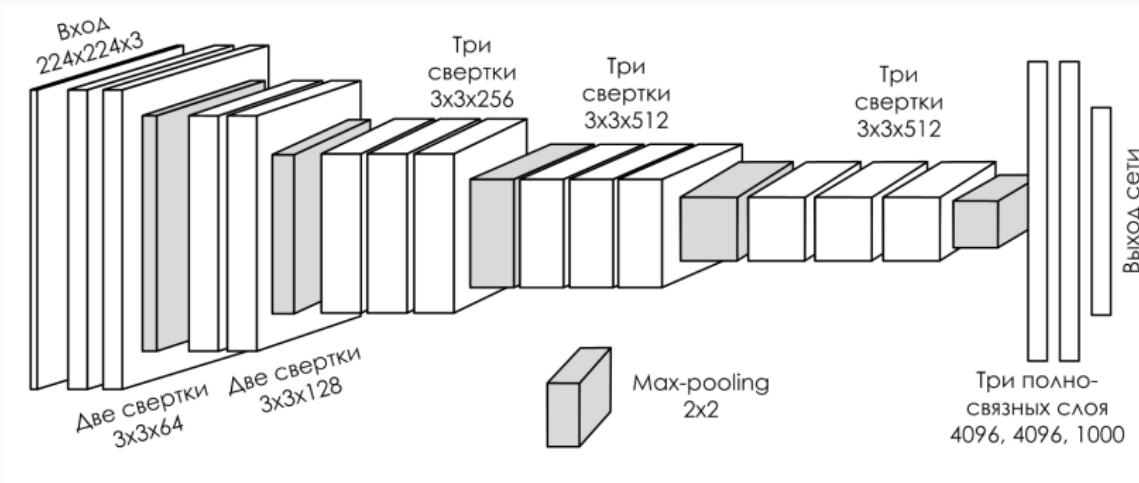
[Browse Categories](#)

- VGG (Oxford Visual Geometry Group): давайте представлять большие свёртки как комбинации свёрток  $3 \times 3$ .
- Это уменьшает число весов и делает сеть глубже.



# VGG

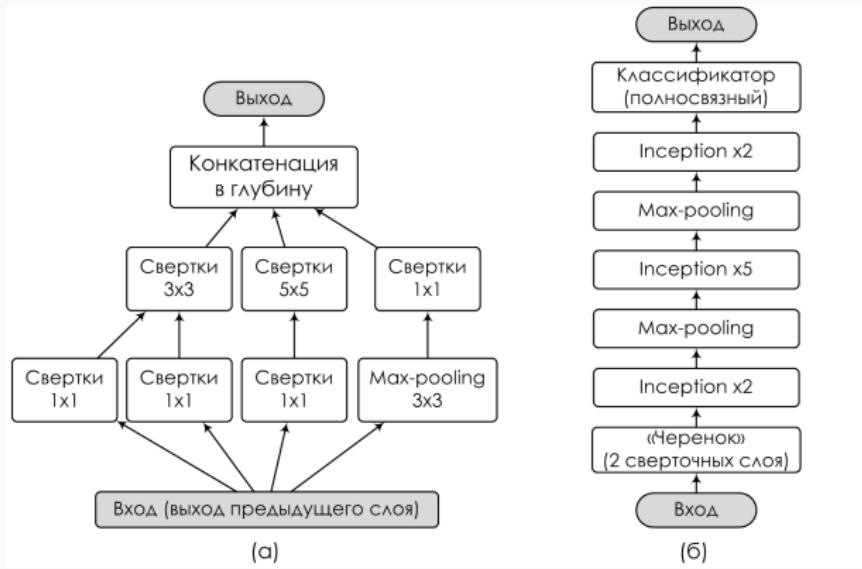
- Сеть VGG, использованная в ImageNet Large Scale Visual Recognition Competition (ILSVRC-2014).



- А сейчас NVIDIA специально оптимизирует GPU для свёрток  $3 \times 3$ .

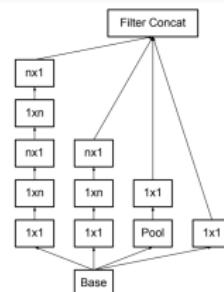
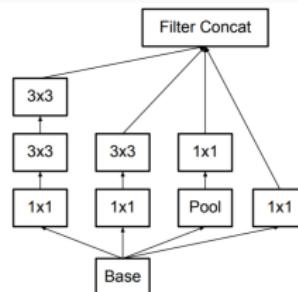
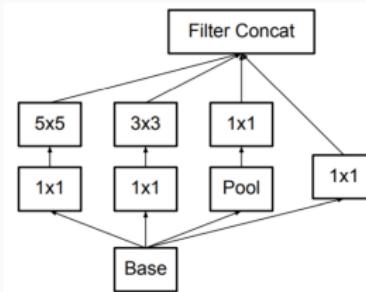
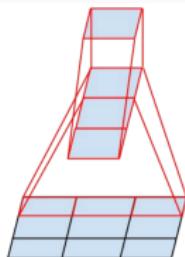
# Inception

- Inception, разработанная командой GoogLeNet: давайте используем идею “сеть в сети” (network in network), чтобы сократить веса ещё больше!
- И дополнительные классификаторы (просто лишние слагаемые в целевую функцию).



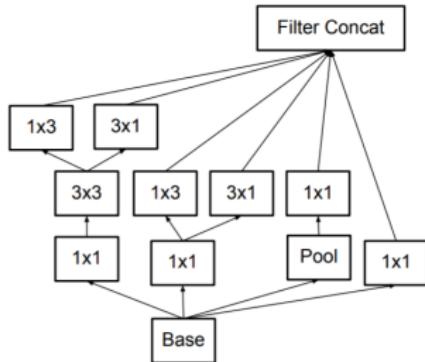
# Inception

- Во второй версии (Szegedy et al., 2015) свёртки  $n \times n$  заменили на комбинации свёрток  $n \times 1$  и  $1 \times n$ :



# Inception

- А банки фильтров в Inception v2 сделали широкими, а не глубокими:



- В той же статье – Inception v3; то же самое, но ещё RMSProp, факторизованные свёртки  $7 \times 7$ , batchnorm во вспомогательных классификаторах и...

- Label smoothing:
  - в обычном классификаторе мы обучаем результаты softmax с целевой переменной  $q(k) = [k = y]$ , т.е. аргументы softmax хотят расти неограниченно;
  - модели становятся слишком уверены в себе, оверфиттинг;
  - решение: давайте вместо этого оптимизировать

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

для какого-то априорного  $u(k)$ , например  $u(k) = \frac{1}{k}$ .

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	<b>18.77%</b>	<b>4.2%</b>

# ResNet

- Остаточное обучение (residual learning): давайте обучим разности между очередным уровнем и предыдущим.
- Тогда градиенты смогут беспрепятственно проходить куда надо.
- Функция, реализуемая остаточным блоком, выглядит как

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

где  $\mathbf{x}^{(k)}$  — входной вектор слоя  $k$ ,  $F(x)$  — функция, которую вычисляет слой нейронов, а  $\mathbf{y}^{(k)}$  — выход остаточного блока, который потом станет входом следующего слоя  $\mathbf{x}^{(k+1)}$ .

- И градиент будет проходить через этот блок беспрепятственно и не будет затухать:

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

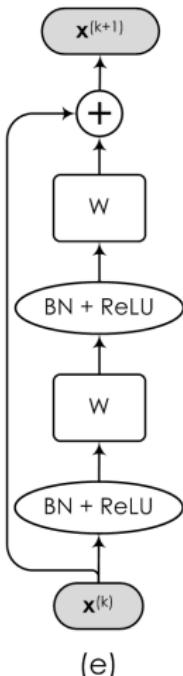
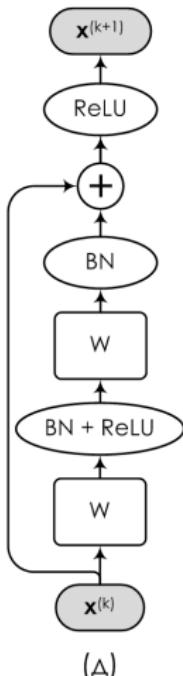
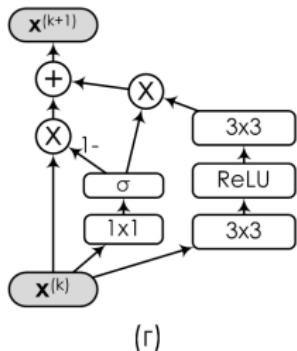
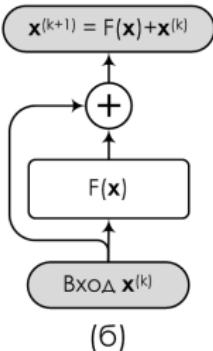
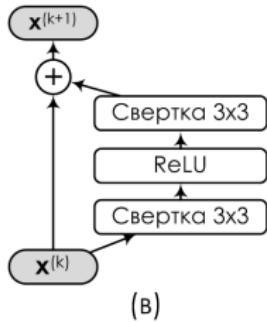
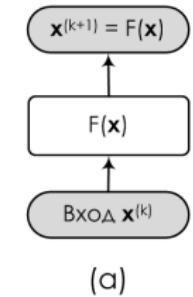
- Это привело к очень глубоким сетям.
- Другой аналогичный подход – *highway networks* (магистральные сети) Шмидхубера.
- Тоже представляем  $y^{(k)}$ , выход слоя  $k$ , как линейную комбинацию входа этого слоя  $x^{(k)}$  и результата  $F(x^{(k)})$ , веса которой управляются другими преобразованиями:

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

где  $C$  – это гейт переноса (carry gate), а  $T$  – гейт преобразования (transform gate); обычно комбинацию делают выпуклой,  $C = 1 - T$ .

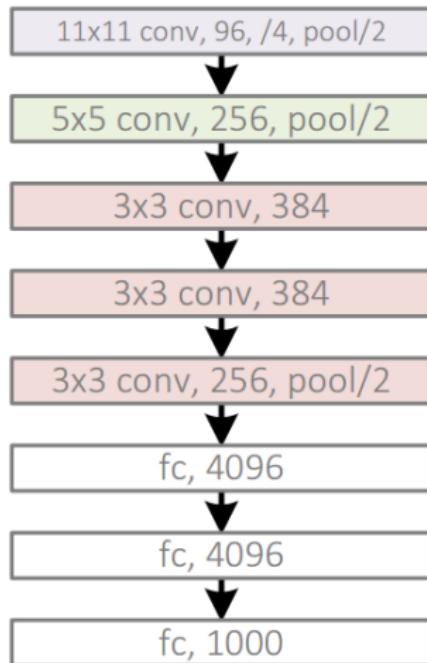
- Практика показывает, что чем «прямее», тем лучше.

# ResNet: варианты



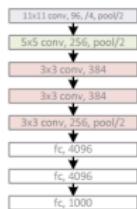
# Revolution of Depth (Kaiming He)

AlexNet, 8 layers  
(ILSVRC 2012)

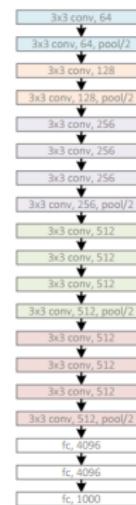


# Revolution of Depth (Kaiming He)

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

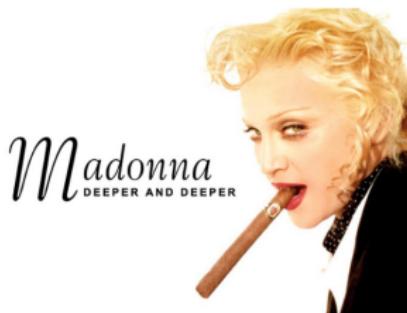
AlexNet, 8 layers  
(ILSVRC 2012)



VGG, 19 layers  
(ILSVRC 2014)

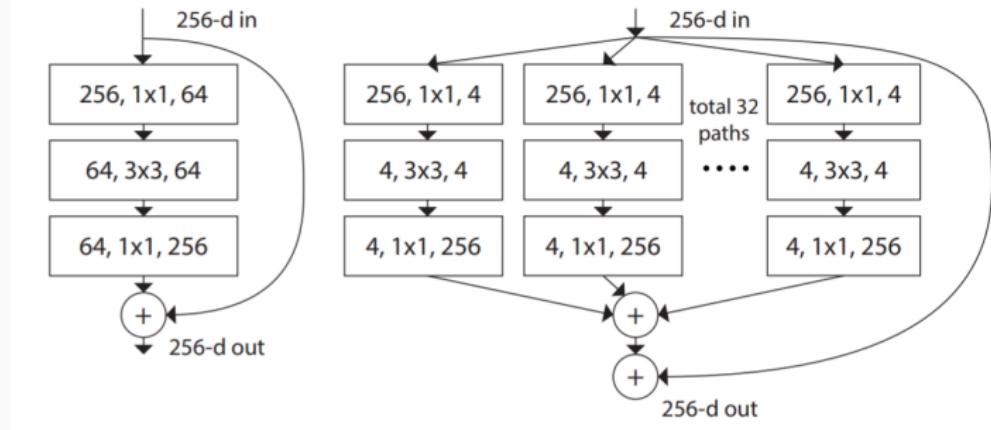


ResNet, **152 layers**  
(ILSVRC 2015)



# ResNeXt

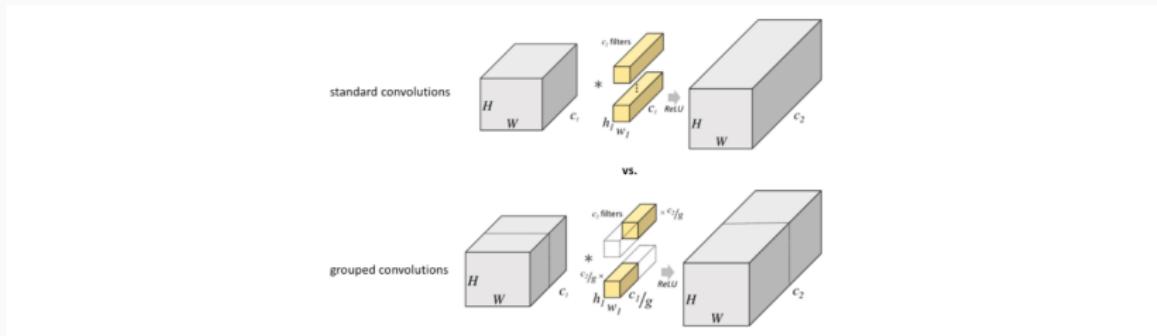
- ResNeXt (Xie et al., 2016): давайте заменим ResNet-блоки на "split-transform-merge" блоки, похожие на Inception.



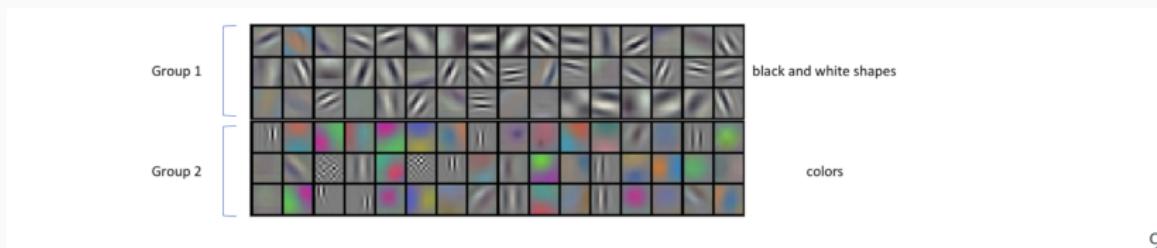
- Вход разбивается на блоки по каналам, к каждому блоку свои свёртки.

# ResNeXt

- Идея похожа на group convolutions, которые были ещё в AlexNet для параллелизации:

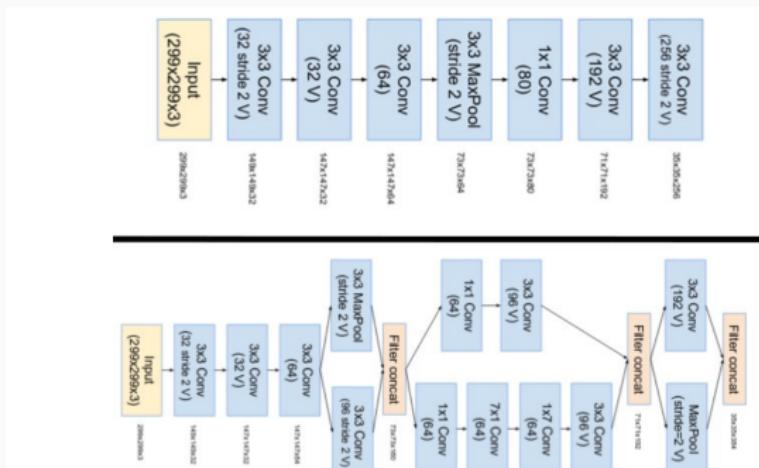


- И они дают специализацию в результатах:



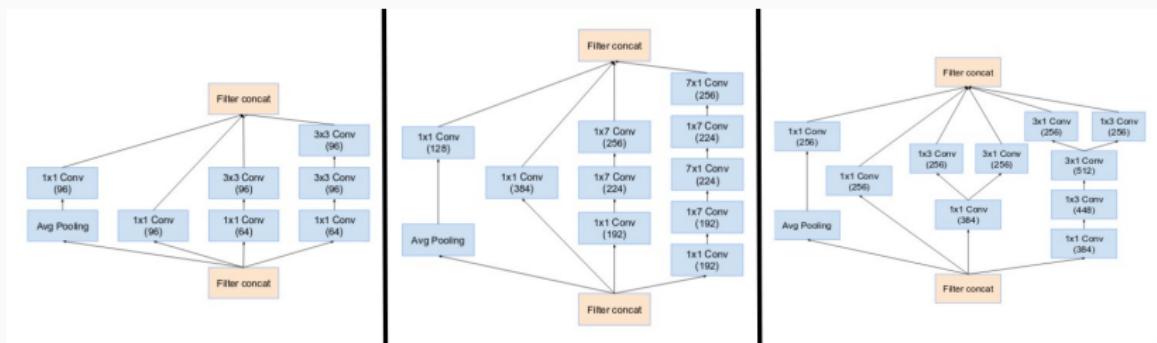
# Inception v4 и Inception ResNet

- Ещё одна классическая статья (Szegedy et al., 2016): вводятся Inception v4 и Inception ResNet.
- Inception v4 – идея в том, чтобы всё стандартизировать, упростить модули. Во-первых, «членок»:



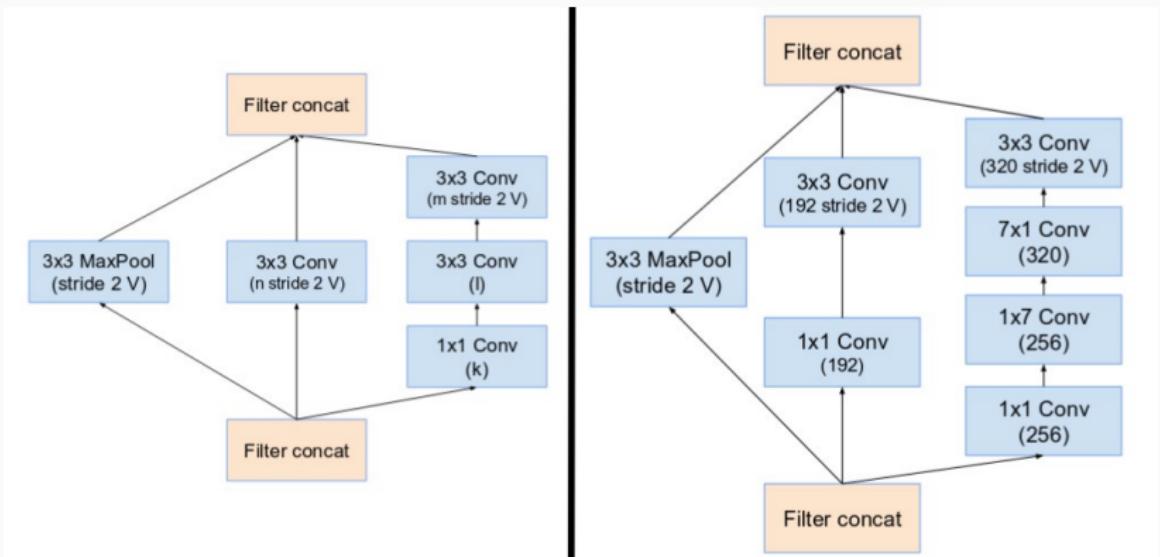
# Inception v4 и Inception ResNet

- Во-вторых, теперь три базовых блока A, B, C:



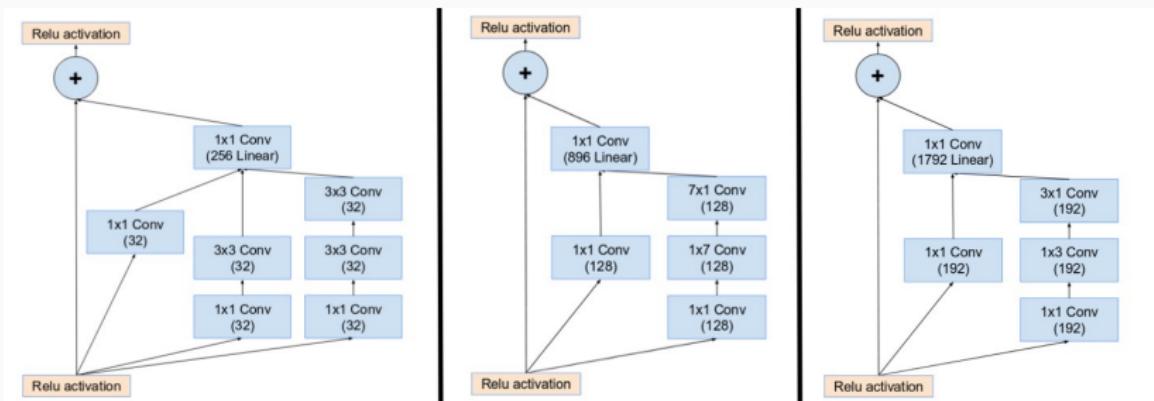
# Inception v4 и Inception ResNet

- И специальные reduction blocks для сокращения размерности сетки:



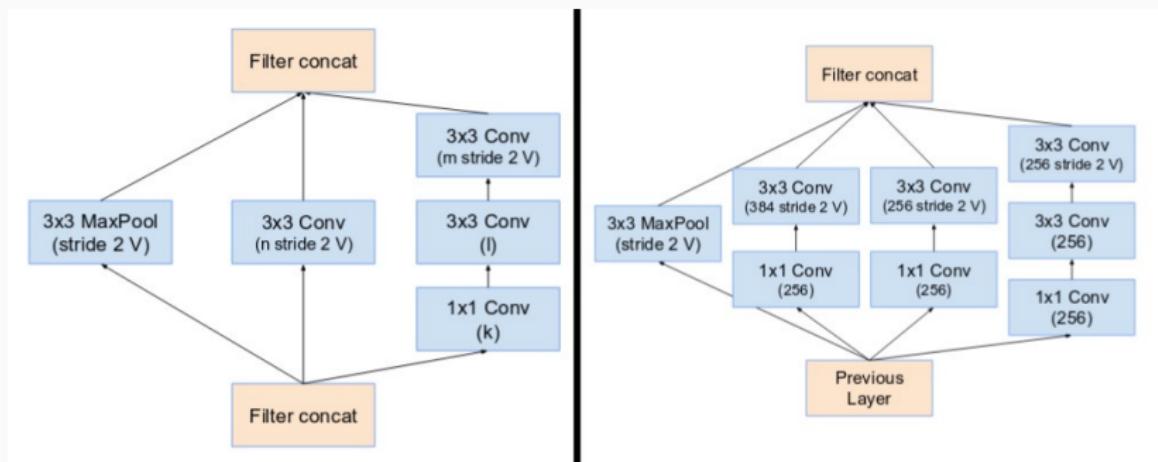
# Inception v4 и Inception ResNet

- В Inception ResNet к тем же блокам добавляются остаточные связи:



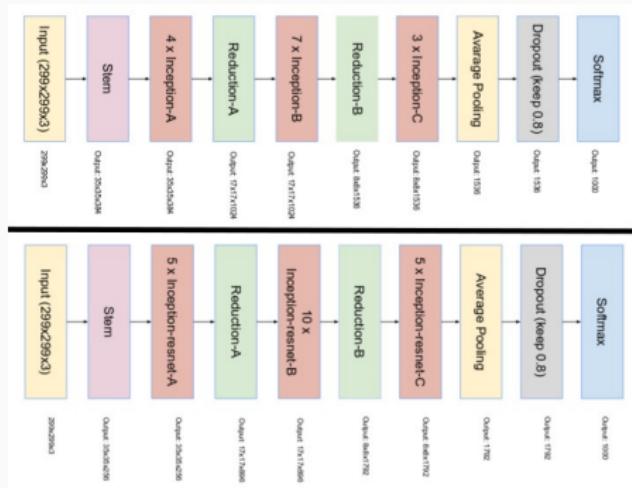
# Inception v4 и Inception ResNet

- Субдискретизации теперь нет, но по-прежнему есть reduction blocks:



# Inception v4 и Inception ResNet

- В итоге архитектура даже упростилась; сверху Inception v4, снизу Inception ResNet:



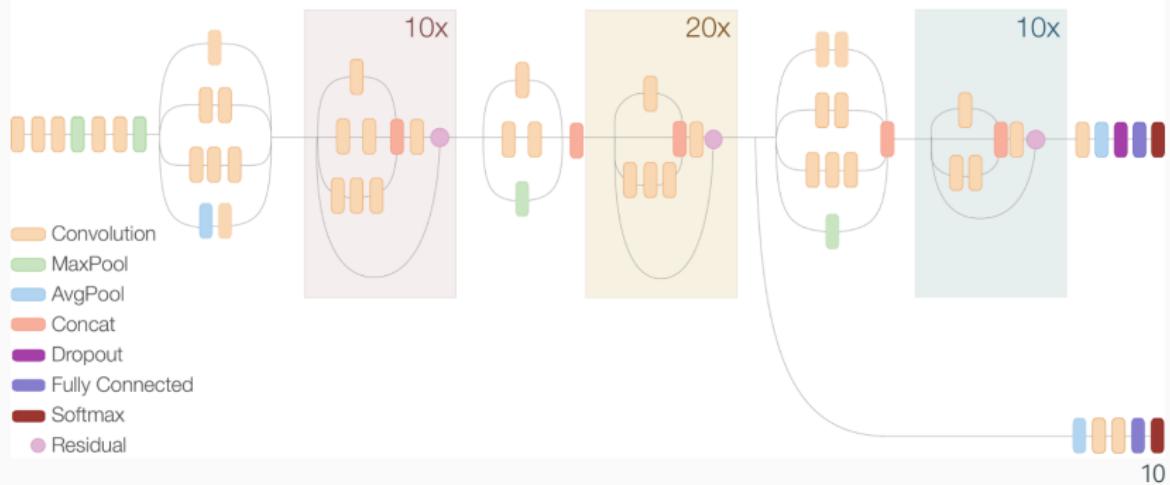
# Inception v4 и Inception ResNet

- Inception ResNet v2:

Inception Resnet V2 Network



Compressed View



# Inception v4 и Inception ResNet

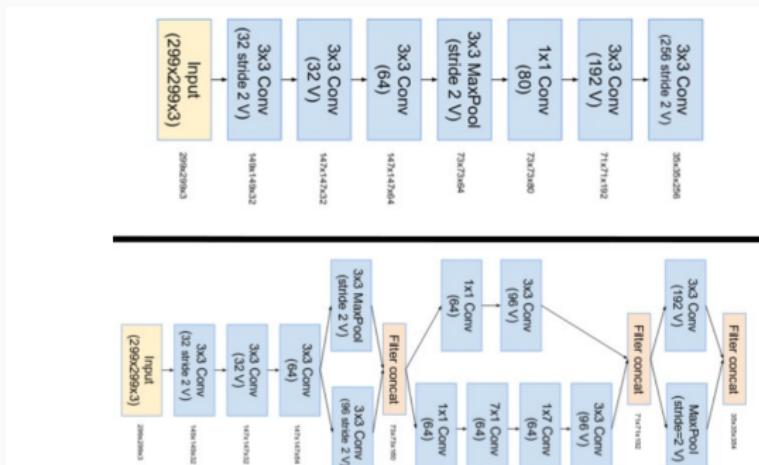
- И работает неплохо:

<b>Network</b>	Crops	<b>Top-1 Error</b>	<b>Top-5 Error</b>
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%

<b>Network</b>	Crops	<b>Top-1 Error</b>	<b>Top-5 Error</b>
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%

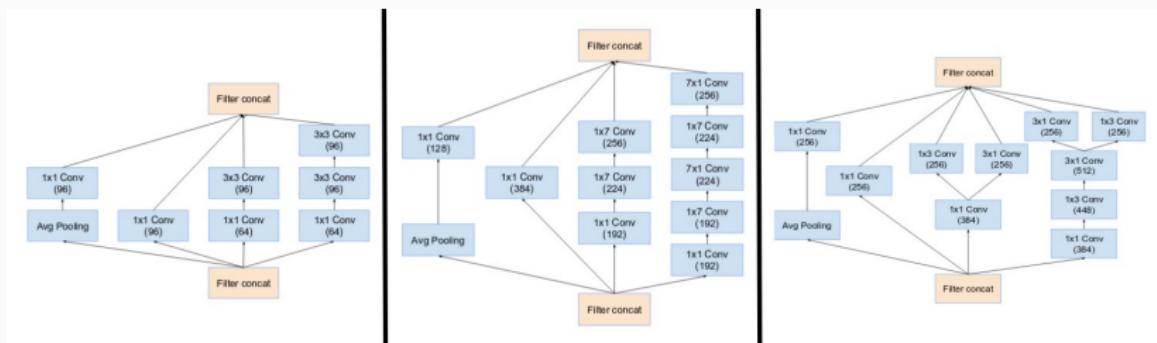
# Inception v4 и Inception ResNet

- Ещё одна классическая статья (Szegedy et al., 2016): вводятся Inception v4 и Inception ResNet.
- Inception v4 – идея в том, чтобы всё стандартизировать, упростить модули. Во-первых, «членок»:



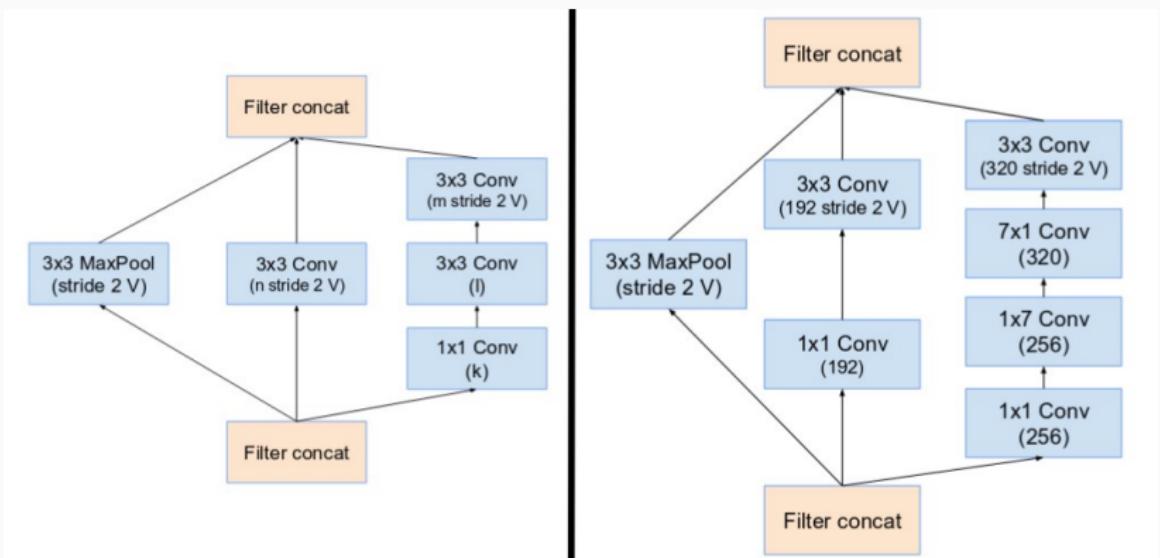
# Inception v4 и Inception ResNet

- Во-вторых, теперь три базовых блока A, B, C:



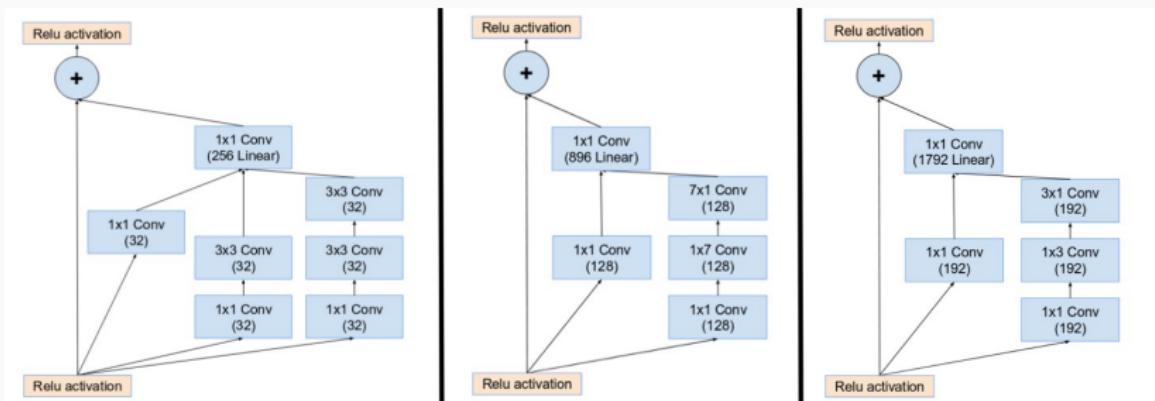
# Inception v4 и Inception ResNet

- И специальные reduction blocks для сокращения размерности сетки:



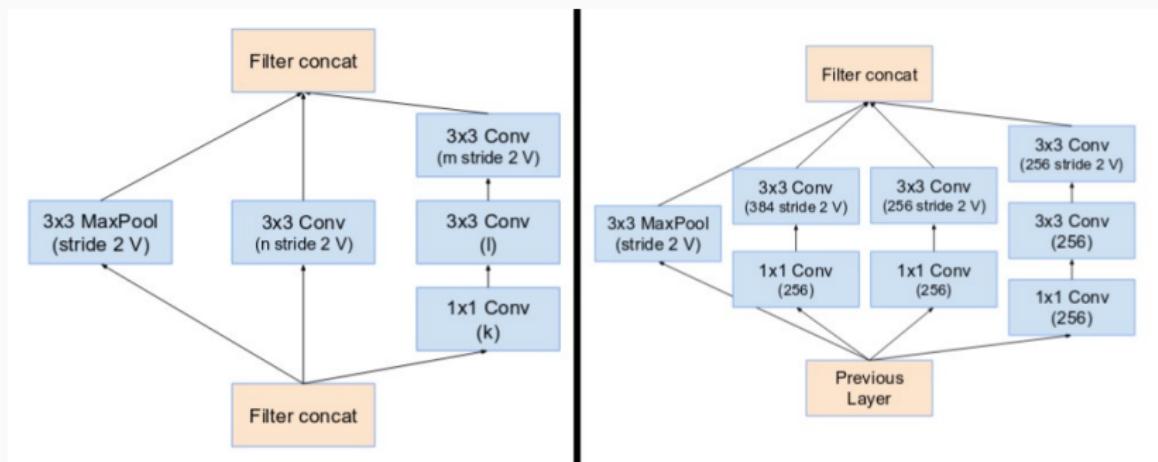
# Inception v4 и Inception ResNet

- В Inception ResNet к тем же блокам добавляются остаточные связи:



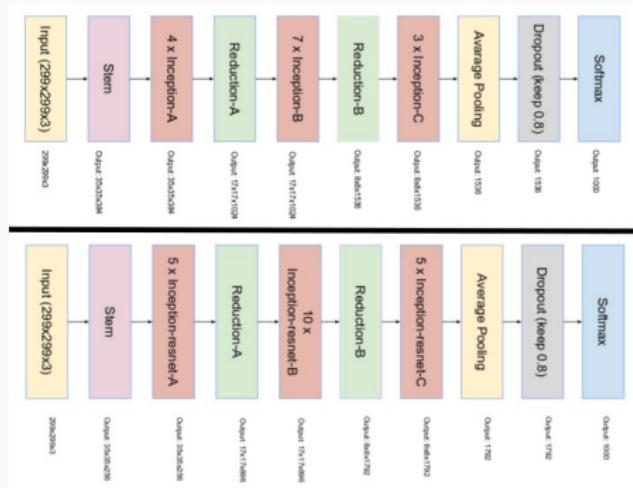
# Inception v4 и Inception ResNet

- Субдискретизации теперь нет, но по-прежнему есть reduction blocks:



## Inception v4 и Inception ResNet

- В итоге архитектура даже упростилась; сверху Inception v4, снизу Inception ResNet:



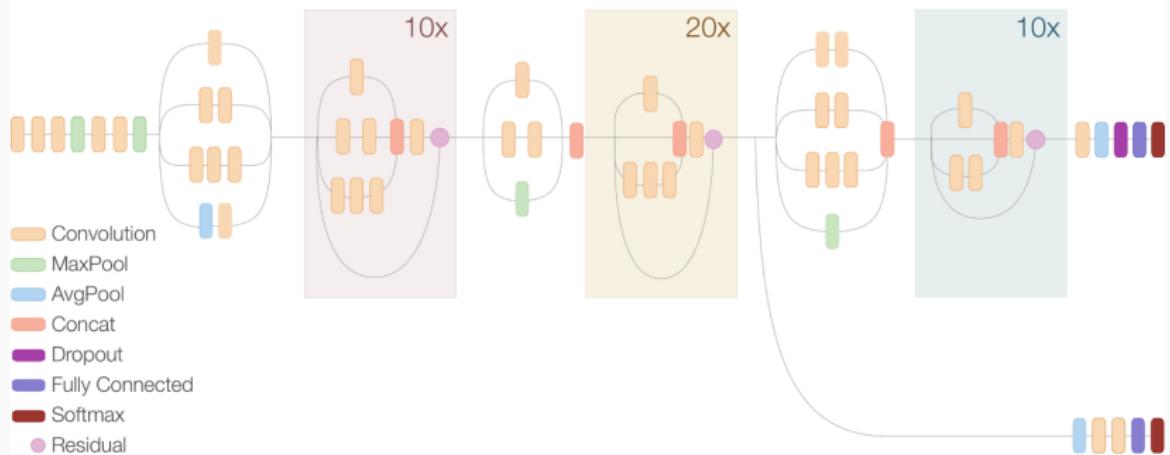
# Inception v4 и Inception ResNet

- Inception ResNet v2:

Inception Resnet V2 Network



Compressed View



# Inception v4 и Inception ResNet

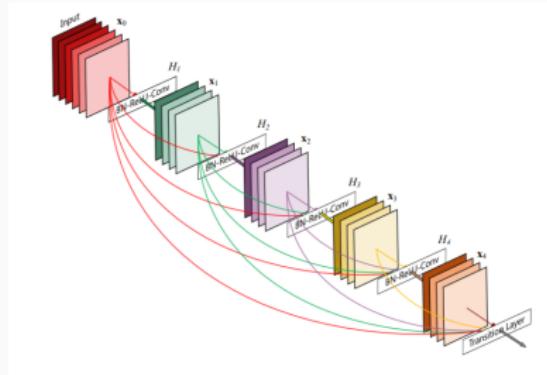
- И работает неплохо:

<b>Network</b>	Crops	<b>Top-1 Error</b>	<b>Top-5 Error</b>
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%

<b>Network</b>	Crops	<b>Top-1 Error</b>	<b>Top-5 Error</b>
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%

# DenseNet

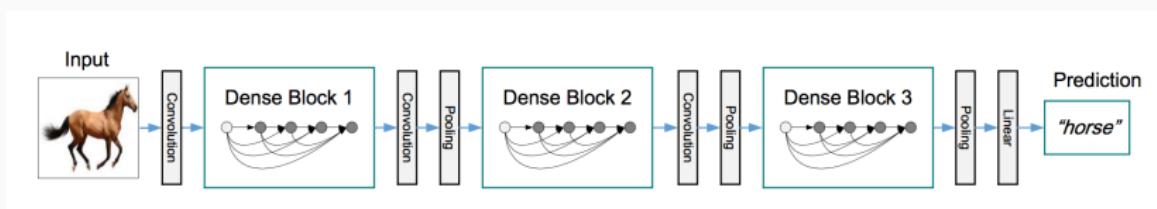
- DenseNet (Huang et al., 2017, best paper at CVPR 2017) – полезно включать не только последовательные соединения между слоями:



- Каждый с каждым! Выглядит странно – сколько же весов нужно?..

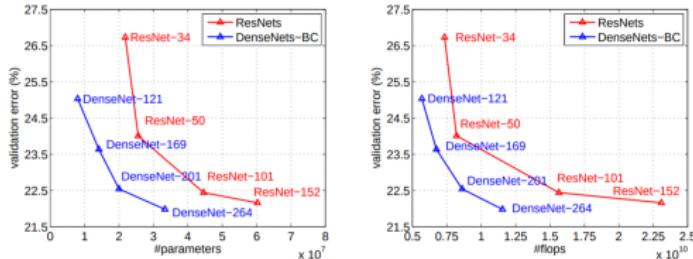
# DenseNet

- Но на самом деле нужно не так уж много; такие соединения позволяют мало фич использовать.
- Например,  $k = 12$  фильтров на каждый уровень – это значит, что на уровне  $m$  будет  $k_0 + k(m - 1)$  фич доступно.
- Такие конструкции объединяют в блоки:



# DenseNet

- В DenseNet меньше весов, обучается быстрее и лучше:



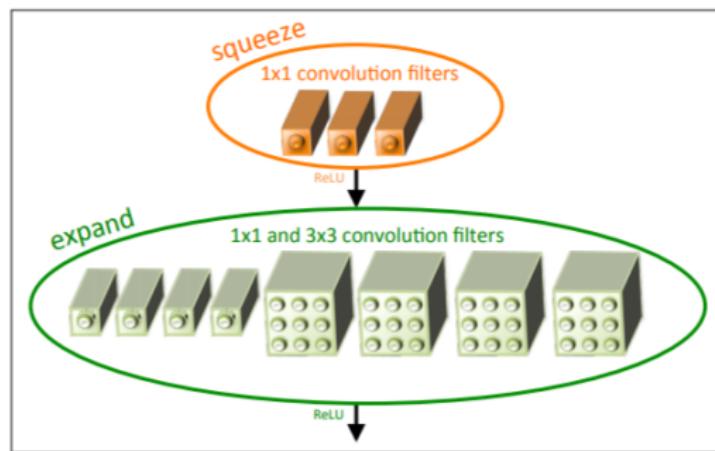
Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17] with Dropout/Drop-path	21 21	38.6M 38.6M	10.18 7.33	5.22 4.60	35.34 28.20	23.30 23.73	2.01 1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110 1202	1.7M 10.2M	11.66 -	5.23 4.91	37.80	24.58	1.75
Wide ResNet [42]	16 28 16	11.0M 36.5M 2.7M	- -	4.81 4.17	- -	22.07 20.50	-
with Dropout				-	-	-	1.64
ResNet (pre-activation) [12]	164 1001	1.7M 10.2M	11.26* 10.56*	5.46 4.62	35.58* 33.47*	24.33 22.71	-
DenseNet ( $k = 12$ )	40	1.0M	<b>7.00</b>	5.24	<b>27.55</b>	24.42	1.79
DenseNet ( $k = 12$ )	100	7.0M	<b>5.77</b>	<b>4.10</b>	<b>23.79</b>	<b>20.20</b>	1.67
DenseNet ( $k = 24$ )	100	27.2M	<b>5.83</b>	<b>3.74</b>	<b>23.42</b>	<b>19.25</b>	<b>1.59</b>
DenseNet-BC ( $k = 12$ )	100	0.8M	<b>5.92</b>	4.51	<b>24.15</b>	22.27	1.76
DenseNet-BC ( $k = 24$ )	250	15.3M	<b>5.19</b>	<b>3.62</b>	<b>19.64</b>	<b>17.60</b>	1.74
DenseNet-BC ( $k = 40$ )	190	25.6M	-	<b>3.46</b>	-	<b>17.18</b>	-

# SqueezeNet

- SqueezeNet (Iandola et al., 2017) – как уменьшить число параметров:
  - заменять  $3 \times 3$  фильтры на  $1 \times 1$ ;
  - уменьшать число входов для  $3 \times 3$ ;
  - делать downsampling как можно позже, чтобы карты активаций были побольше.

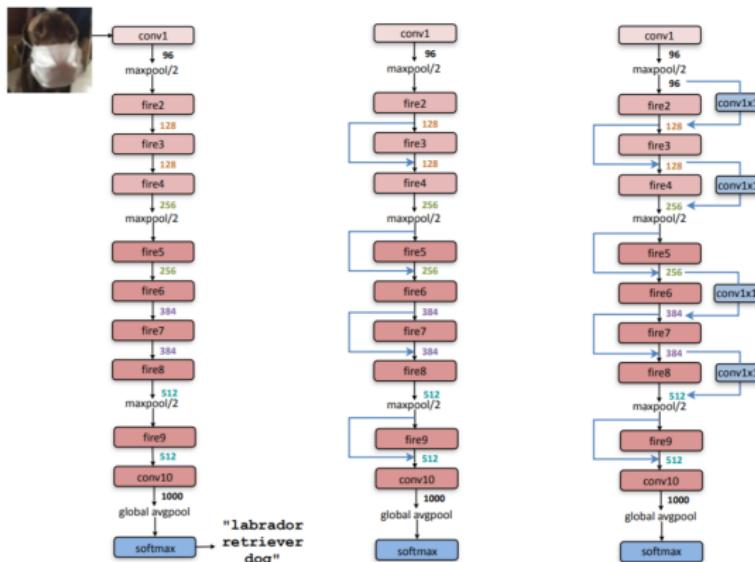
# SqueezeNet

- Fire module:
  - squeeze convolutional layer (только  $1 \times 1$ );
  - expand layer ( $1 \times 1$  и  $3 \times 3$ ).



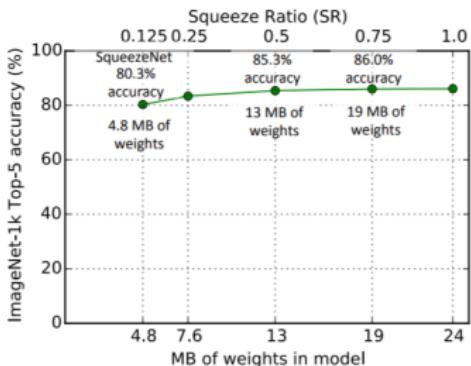
# SqueezeNet

- И общая архитектура SqueezeNet:

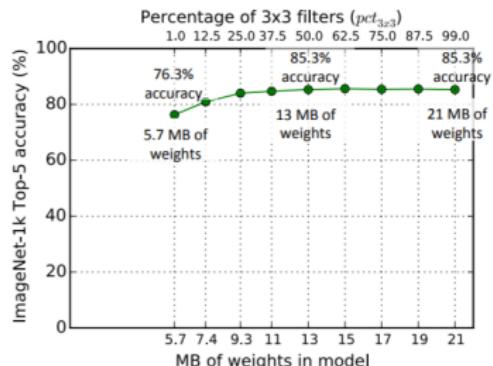


# SqueezeNet

- Получается в 50 раз меньше параметров, чем у AlexNet, но:



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

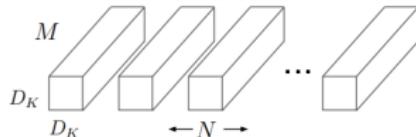


(b) Exploring the impact of the ratio of 3x3 filters in expand layers ( $pct_{3x3}$ ) on model size and accuracy.

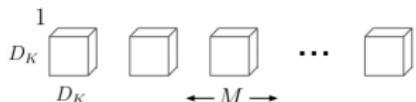
Architecture	Top-1 Accuracy	Top-5 Accuracy	Model Size
Vanilla SqueezeNet	57.5%	80.3%	4.8MB
SqueezeNet + Simple Bypass	<b>60.4%</b>	<b>82.5%</b>	4.8MB
SqueezeNet + Complex Bypass	58.8%	82.0%	7.7MB

# MobileNet

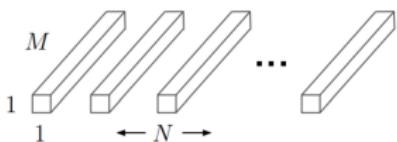
- MobileNet (Howard et al., 2017): сети для мобильных устройств.
- Depthwise separable convolutions: давайте разложим свёртку на depthwise (один фильтр к каждому каналу) и  $1 \times 1$ .



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



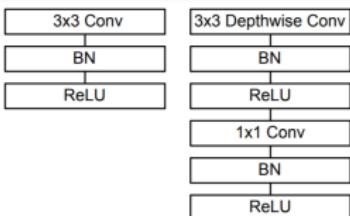
(c)  $1 \times 1$  Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

# MobileNet

- Тогда структура слоя будет чуть сложнее (но весов меньше), и общая структура не слишком глубокая:

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× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
5× 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 $7 \times 7$	$7 \times 7 \times 1024$
FC / s1	1024 × 1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$



# MobileNet

- И получается, что можно сэкономить очень много параметров ценой небольшого ухудшения качества:

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

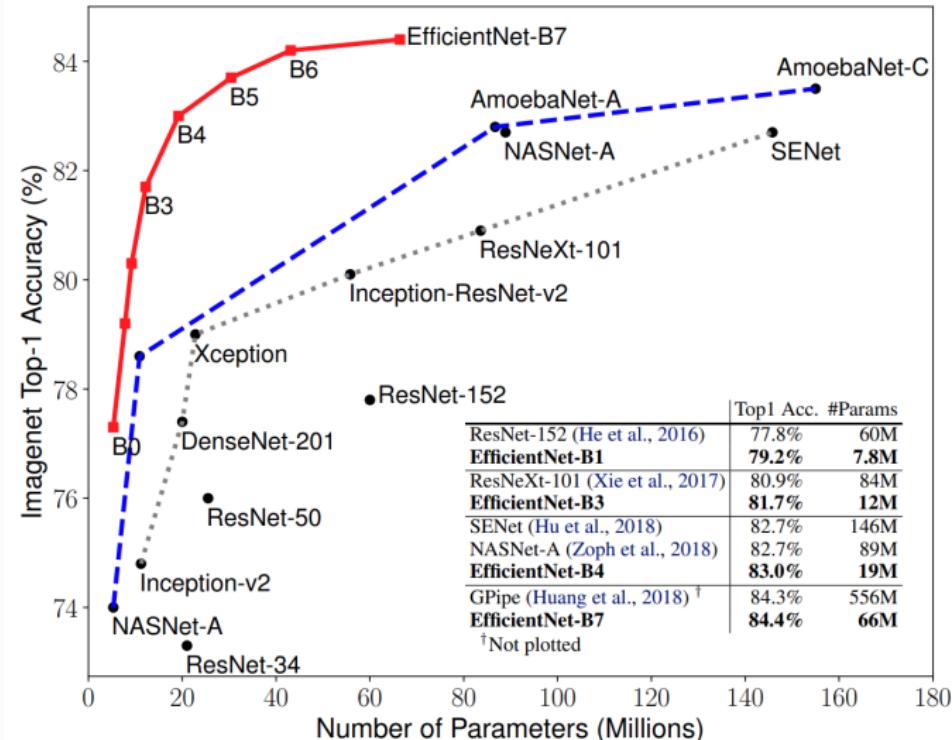
Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.50 MobileNet-160	60.2%	76	1.32
SqueezeNet	57.5%	1700	1.25
AlexNet	57.2%	720	60

Table 10. MobileNet for Stanford Dogs

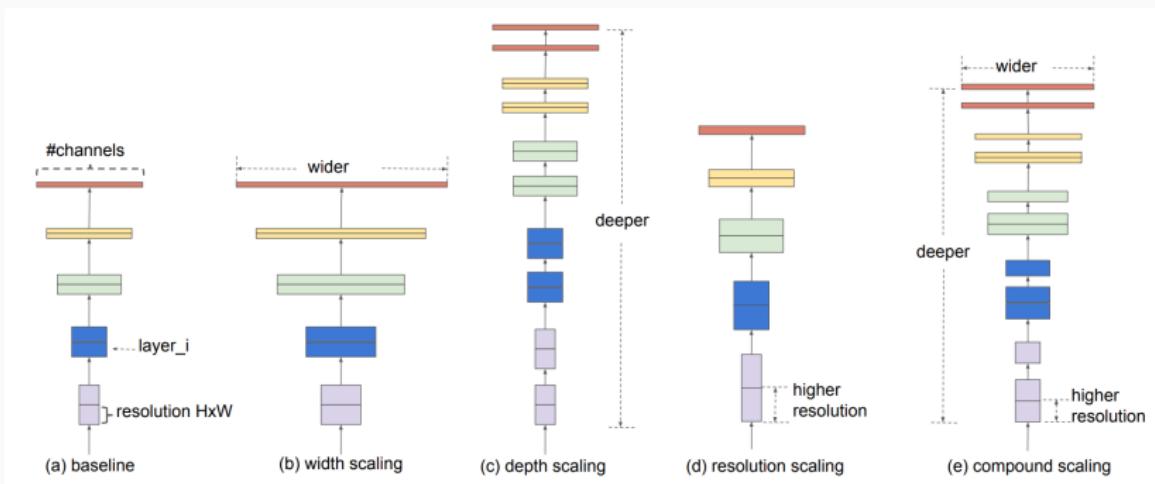
Model	Top-1 Accuracy	Million Mult-Adds	Million Parameters
Inception V3 [18]	84%	5000	23.2
1.0 MobileNet-224	83.3%	569	3.3
0.75 MobileNet-224	81.9%	325	1.9
1.0 MobileNet-192	81.9%	418	3.3
0.75 MobileNet-192	80.5%	239	1.9

# EfficientNet

- EfficientNet (Tan, Le, 2019): а вот и у neural architecture search дошли руки до свёрточных архитектур



- Разные виды масштабирования для моделей:

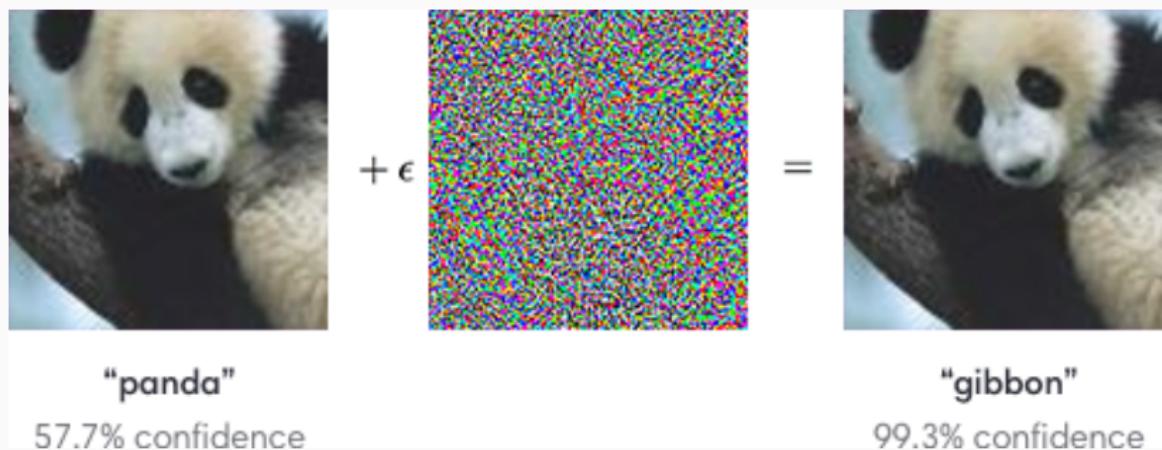


## Adversarial examples

---

## Adversarial examples

- Интересная особенность глубоких сетей: можно обмануть любую сеть, сделать картинку подходящей под любой класс неразличимым на человеческий взгляд шумом.



- Как?..

## Adversarial examples

- Давайте сделаем градиентный спуск не по весам сети  $\theta$ , а по входу  $x$ !
- Надо только контролировать, чтобы новый пример  $\hat{x}$  оставался похож на исходный  $x$ , например чтобы  $\|\hat{x} - x\|_\infty \leq \epsilon$ .
- Более того, можно попробовать сделать  $\hat{x}$  устойчивым ко всяким преобразованиям вроде поворотов.
- Кстати, а это как сделать?
- Давайте посмотрим на пример...

# Adversarial examples

- Направление началось в *Intriguing properties of neural networks* (Szegedy et al., 2013). Вообще очень интересная статья...
- Например, мы с вами анализировали “значения” нейронов, находя сильнейшие их активации.
- Т.е. предполагается, что если мы проанализируем нейроны последнего уровня, то это будет правильный базис в латентном пространстве, на котором легко выделить семантику.
- Правильно?..

# Adversarial examples

- ...не совсем:



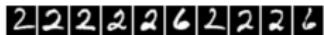
(a) Unit sensitive to lower round stroke.



(b) Unit sensitive to upper round stroke, or lower straight stroke.



(c) Unit sensitive to left, upper round stroke.



(d) Unit sensitive to diagonal straight stroke.

Figure 1: An MNIST experiment. The figure shows images that maximize the activation of various units (maximum stimulation in the natural basis direction). Images within each row share semantic properties.



(a) Direction sensitive to upper straight stroke, or lower round stroke.



(b) Direction sensitive to lower left loop.



(c) Direction sensitive to round top stroke.



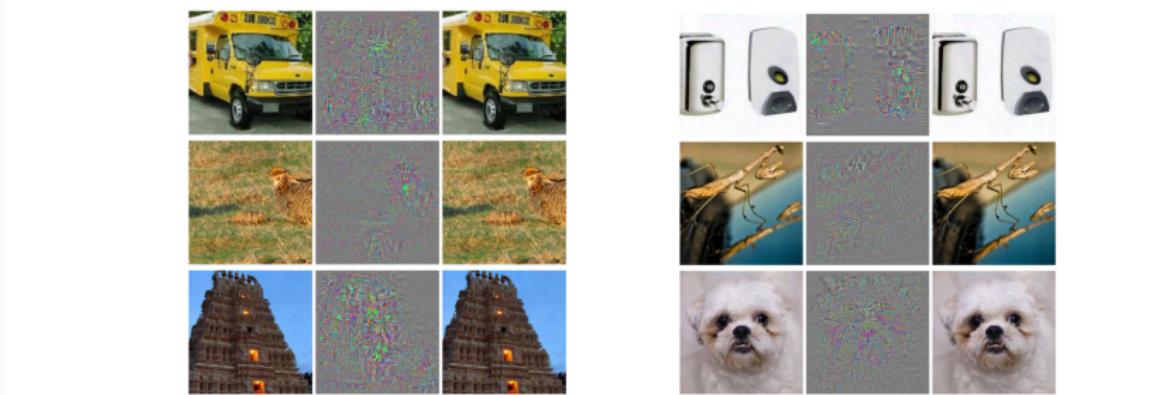
(d) Direction sensitive to right, upper round stroke.

Figure 2: An MNIST experiment. The figure shows images that maximize the activations in a random direction (maximum stimulation in a random basis). Images within each row share semantic properties.

- Т.е. у обычных сетей никакого нет disentanglement, пространство признаков хорошее, но базис в нём не лучше случайного.

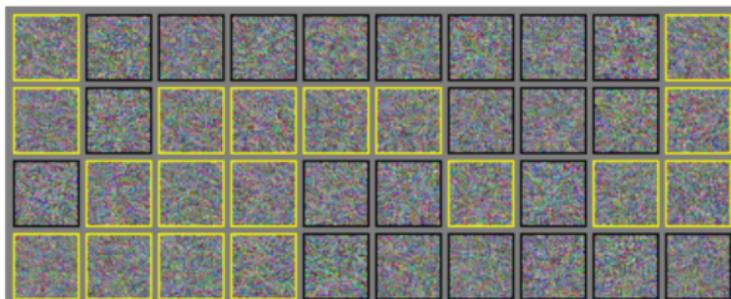
# Adversarial examples

- И там же adversarial attacks появились; для AlexNet всё то, что справа – страус:



# Adversarial examples

- Дальше в (Goodfellow, Shlens, Szegedy, 2014); всё, что выделено – самолёт:



# Adversarial examples

- Выводы (Goodfellow, Shlens, Szegedy, 2014):
  - объясняют на уровне линейных классификаторов: для  $\hat{x} = x + z$  мы хотим сдвинуть  $w^\top \hat{x} = w^\top x + w^\top z$ , т.е. просто берём  $z = \text{sign}(w)$  и применяем ограничения на норму отклонения;
  - то же самое можно сделать в любой сети, приблизив линейно в окрестности:

$$z = \epsilon \text{sign}(\nabla_x L(\theta, x, y));$$

- т.е. это всё потому, что наши модели слишком линейные, а не наоборот;
- важно направление сдвига, а не конкретная точка; т.е. можно даже обобщить adversarial сдвиг на разные чистые примеры;
- и можно попытаться регуляризовать, добавив adversarial сдвиг в целевую функцию:

$$L'(\theta, x, y) = \alpha L(\theta, x, y) + (1 - \alpha) L(\theta, x + \epsilon \text{sign}(\nabla_x L(\theta, x, y)), y).$$

- Но на этом история не заканчивается...

# Adversarial examples

- Варианты атак:
  - Deep Fool attack (Bastani et al., 2016): двигаем пример к гиперплоскости, разделяющей классы,  $z = \frac{f(x_0)}{\|w\|_2^2} w$  для линейного классификатора и  $z_i = \frac{f(x_i)}{\|\nabla f(x_i)\|_2^2} \nabla f(x_i)$  для любой функции;
  - (Carlini, Wagner, 2016): ищем минимальные исправления на основе  $L_0$ ,  $L_2$  и  $L_\infty$ -норм, до сих пор одни из лучших атак;
  - а можно искать не исправления входа, а признаки, которые полезно исправлять;
  - (Papernot et al., 2016): выясним, какие пиксели сильнее всего влияют, и будем их сдвигать;
  - и очень, очень много чего ещё, это мы ещё про GAN'ы не начинали говорить...

# Adversarial examples

- Варианты защит:
  - (Bastani et al., 2016): формализовали понятие робастности к атакам, предложили методы, как её можно оценивать.
  - (Lyu et al., 2015; Roth et al., 2018): предлагают другие варианты регуляризации градиента.
  - (Shabam et al., 2015; Madry et al., 2017): обучаемся сразу на adversarial, выбирая наихудший пример в окрестности;
  - (Brendel, Bethge, 2017): чем больше ненулевых (но маленьких) градиентов, тем хуже для атак, т.е. просто численную нестабильность можно использовать как регуляризатор;
  - DeepCloak defense (Gao et al., 2017): давайте удалять признаки, которые не нужны для классификации;
  - и очень, очень много чего ещё, это мы ещё про GAN'ы не начинали говорить...

# Adversarial examples

- (Kurakin, Goodfellow, Bengio, 2016): атаки в реальном мире!  
Более того, black box: делаем атаки на одной модели, а проверяем на другой.
- Вот приложение, которое портит картинку:



# Adversarial examples

- Ещё круче – тут adversarial example распечатали и сфотографировали... и всё равно часто ломаются сети!



(a) Printout



(b) Photo of printout



(c) Cropped image

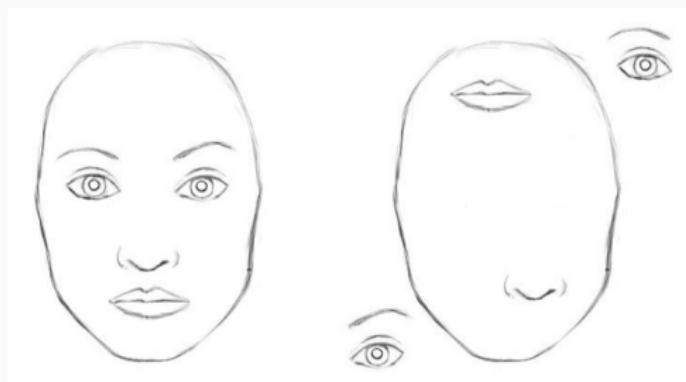
- Насколько это реалистично – пока непонятно, но есть некие общие соображения, почему это всё работает...

## Капсулевые сети

---

# Капсулные сети

- У свёрточных сетей есть проблемы. Например, признаки важны, но их взаимное расположение – не очень:



- Почему так?

## Капсулльные сети

- В основном из-за субдискретизации, того самого свойства max-pooling забывать, откуда пришёл признак.
- Хинтон: “The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster”.
- <https://www.youtube.com/watch?v=rTawFwUvnLE&feature=youtu.be>
- И даже если делать полносвязные сети, они всё равно не смогут отразить пространственные соотношения между признаками/объектами.
- Routing problem: как отдать выходы нейронов именно тем нейронам, которым они нужны?

# Капсулевые сети

- Мы с вами, судя по всему, строим представление трёхмерного мира в голове, как бы обратный рендеринг (inverse geometry).
- Поэтому нам легко распознавать объекты с разных сторон.



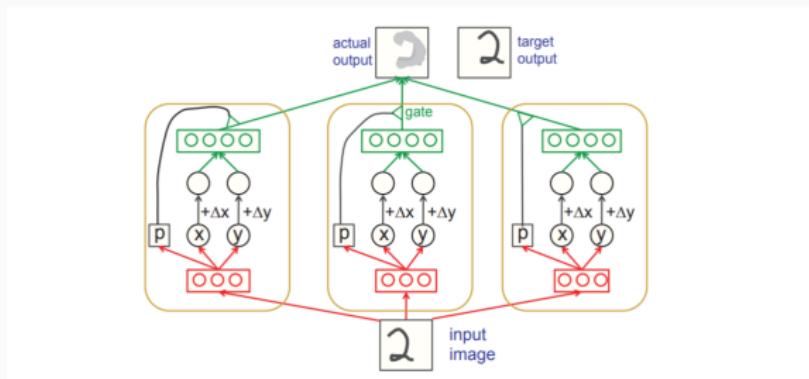
- Но CNN этого ничего не могут.
- Хорошо бы научиться как-то делать это в нейронных сетях, но до очень недавнего времени не было метода, как это можно пытаться сделать.

- Капсулевые сети (capsule networks):
  - проблема с обычными CNN в том, что нейрон выдаёт всего один скаляр;
  - хочется сделать небольшую группу нейронов, которая будет отражать конкретную (неявно заданную, конечно) сущность;
  - капсула – группа нейронов, которая выдаёт вероятность того, что эта сущность присутствует, и группу свойств этого объекта (положение, ориентацию, масштаб, скорость движения, цвет...);
  - соответственно, капсула получает выходы капсул предыдущего уровня и ищет, где есть “совпадения высокой размерности”, т.е. где сразу много свойств совпадают с тем, что ей надо; это значит, что тут есть её объект, определяемый сразу несколькими признаками.
- Пока ничего не понятно, правда?..

- Первая статья о капсулах – *Transforming Auto-encoders* (Hinton et al., 2011):
  - предположим, что всё это работает;
  - но как обучить первый уровень капсул? ему-то на вход пиксели подают;
  - идея – давайте обучать автокодировщики, которые явным образом производят преобразования.

# Капсулльные сети

- Здесь мы явно распознаём положение цифры на входе, потом явным образом прибавляем сдвиг, а на выходе хотим получить сдвинутую цифру:



# Капсулевые сети

- И получается неплохо:



- Можно то же сделать с трёхмерными свойствами:



# Капсулевые сети

- То есть можно начать с чего-то. А дальше капсула работает так:
  - капсула выдаёт вектор;
  - длина вектора – это уверенность в том (вероятность того), что объект есть;
  - а координаты вектора кодируют как раз свойства объекта, например расположение.
- Если капсула распознаёт лицо, и мы начнём лицо двигать по картинке, вектор должен поворачиваться куда надо.

# Капсулевые сети

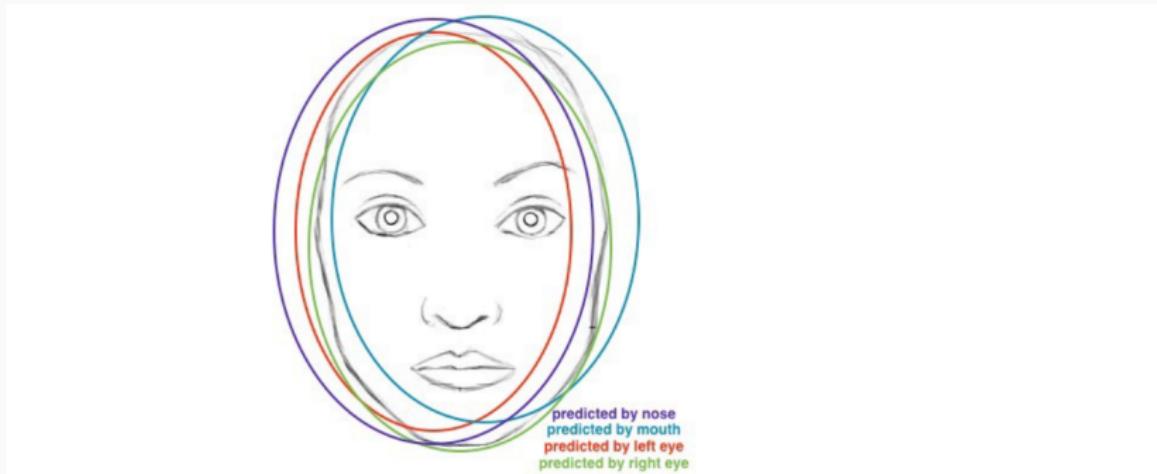
- Формально выглядит так:

Capsule vs. Traditional Neuron			
	Input from low-level capsule/neuron	vector( $\mathbf{u}_i$ )	scalar( $x_i$ )
Operation	Affine Transform	$\hat{\mathbf{u}}_{j i} = \mathbf{W}_{ij}\mathbf{u}_i$	—
	Weighting	$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j i}$	$a_j = \sum_i w_i x_i + b$
	Sum		
	Nonlinear Activation	$\mathbf{v}_j = \frac{\ \mathbf{s}_j\ ^2}{1+\ \mathbf{s}_j\ ^2} \frac{\mathbf{s}_j}{\ \mathbf{s}_j\ }$	$h_j = f(a_j)$
Output		vector( $\mathbf{v}_j$ )	scalar( $h_j$ )

- Матрица весов показывает, что сделать с признаками на входе (какое между ними должно быть соотношение).
- После умножения получается предсказание значения признака более высокого уровня.

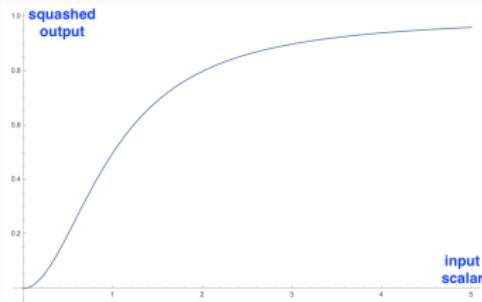
# Капсулевые сети

- Т.е.  $\hat{u}_1$  показывает, где лицо “по мнению глаз”,  $\hat{u}_2$  – “по мнению носа” и т.п.
- Если эти мнения примерно сходятся, признак активируется:



# Капсулевые сети

- Веса  $c_{ij}$  показывают, куда предыдущая капсула посыпает свой  $u_i$ , а куда нет; это и есть routing, чуть позже.
- А дальше новая векторная нелинейность “squash”: нормируем в единицу и ещё масштабируем длину:



# Капсулевые сети

- Самое интересное – routing, откуда взять веса  $c_{ij}$ :
  - $c_{ij} \geq 0$ , и для каждой капсулы нижнего уровня  $i$   $\sum_j c_{ij} = 1$ ;
  - т.е. это вероятность того, что выход  $\mathbf{u}_i$  принадлежит капсуле верхнего уровня  $j$ .
- Dynamic routing (Sabour, Frosst, Hinton, 2017) – отправляем капсулы туда, где с ними согласны:

---

## Procedure 1 Routing algorithm.

---

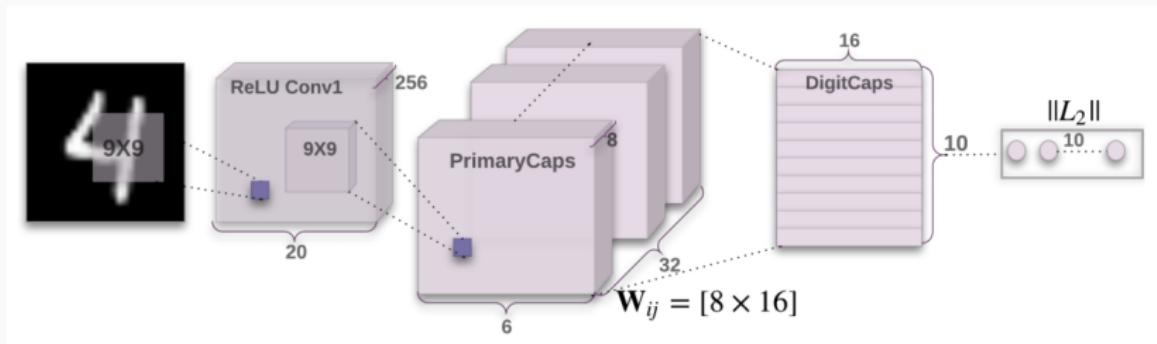
```
1: procedure ROUTING( $\hat{\mathbf{u}}_{j|i}, r, l$ )
2:   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow 0$ .
3:   for  $r$  iterations do
4:     for all capsule  $i$  in layer  $l$ :  $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$             $\triangleright \text{softmax computes Eq. 3}$ 
5:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$ 
6:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$             $\triangleright \text{squash computes Eq. 1}$ 
7:     for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$ 
return  $\mathbf{v}_j$ 
```

---

- Скалярное произведение увеличивает веса капсул, которые похожи на выход соответствующей капсулы верхнего уровня.

# Капсулевые сети

- CapsNet:



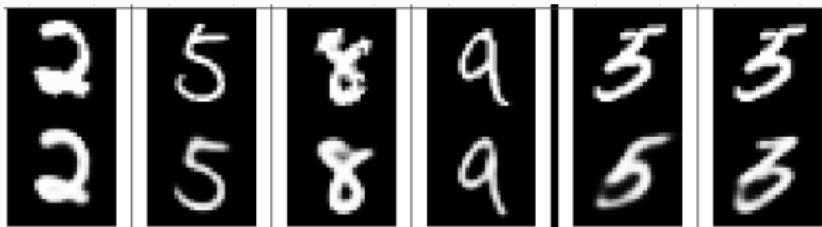
- На выходе у нас 10 векторов размерности 16 каждый (DigitCaps). Функция потерь напоминает SVM:

$$L_k = T_k \max(0, m^+ - \|\mathbf{v}_k\|)^2 + \lambda(1 - T_k) \max(0, \|\mathbf{v}_k\| - m^-)^2,$$

где  $T_k$  – правильный ответ,  $m^+ = 0.9$ ,  $m^- = 0.1$ ; т.е. у правильной цифры длинный вектор, у неправильных короткий.

## Капсулные сети

- И ещё reconstruction loss добавляют: берут выход только правильной капсулы и пытаются реконструировать цифру.

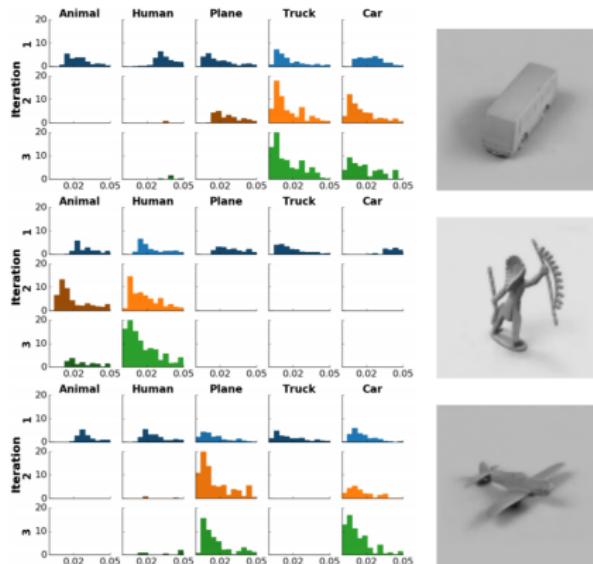


- И действительно получается, что размерности вектора чему-то соответствуют:

Scale and thickness	6 6 6 6 6 6 6 6 6 6
Localized part	6 6 6 6 6 6 6 6 6 6
Stroke thickness	5 5 5 5 5 5 5 5 5 5
Localized skew	4 4 4 4 4 4 4 4 4 4
Width and translation	3 3 3 3 3 3 3 3 3 3
Localized part	2 2 2 2 2 2 2 2 2 2

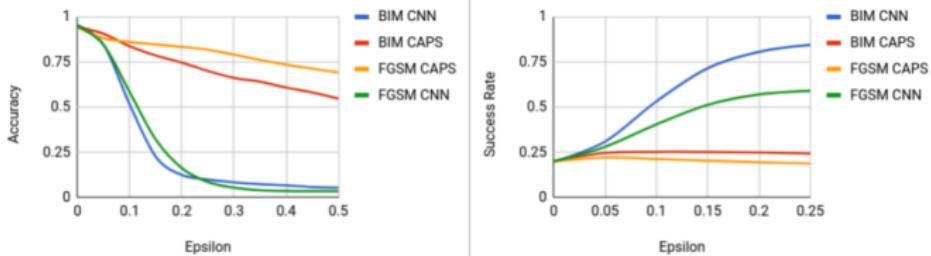
# Капсулевые сети

- И последнее, совсем свежая статья (Hinton, Sabour, Frosst, ICLR 2018).
- Сложная часть капсулевых сетей – это routing; там предлагается EM-схема для апдейта и распределения весов между соседними уровнями.



# Капсулевые сети

- И интересно, что капсулевые сети, похоже, более устойчивы к тем самым adversarial attacks:



- Но взлетит ли всё это на самом деле – пока неясно...

Спасибо!

Спасибо за внимание!

