



# Сверточные нейронные сети



# Стандартное представление изображения



# Стандартное представление ч/б изображения

W

Черно-белая картинка — матрица из пикселей.

H — длина в пикселях,

W — ширина в пикселях.

Пиксель — число, означ. интенсивность цвета.

Интенсивность — число от 0 до 1.

Обычно ее кодируют числом от 0 до 255 (1 байт).

0	2	15	0	0	11	10	0	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10	0
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124	1
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255	49
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255	36
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235	62
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137	0
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6	0
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0	19
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7	0
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1	0
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0	4
0	22	206	252	246	251	241	100	24	113	255	245	255	194	9	0
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56	0
0	218	251	250	137	7	11	0	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61	0
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52	4
0	18	146	250	255	247	255	255	249	255	240	255	129	0	5	0
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12	0
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4	1
0	0	5	5	0	0	0	0	0	0	14	1	0	6	6	0



Черно-белая картинка —  
тензор размера (H, W), состоящий из uint8 чисел.



# Стандартное представление цветного изображения

Цветная картинка — матрица из пикселей,

$H$  — длина в пикселях,

$W$  — ширина в пикселях.

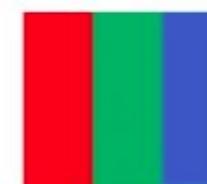
Пиксель обычно представляют в **RGB** формате:

массив интенсивностей **красного**, **зеленого** и **синего**.

**Интенсивность** — число от 0 до 1.

Обычно ее кодируют числом от 0 до 255 (1 байт).

Пиксель



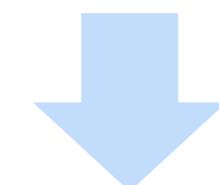
→ (167, 083, 151)

→ (000, 000, 000)

→ (255, 255, 255)

Цветная картинка —

тензор размера ( $H, W, 3$ ), состоящий из `uint8` чисел.





## Стандартное представление изображения

Цветная картинка — тензор размера ( $H, W, 3$ ), состоящий из чисел `uint8`.

Изображение можно разложить на 3 канала по размерности пикселя.



Цветное  
изображение

=



Интенсивности  
красного

+



Интенсивности  
зеленого

+



Интенсивности  
синего



# Представление изображения

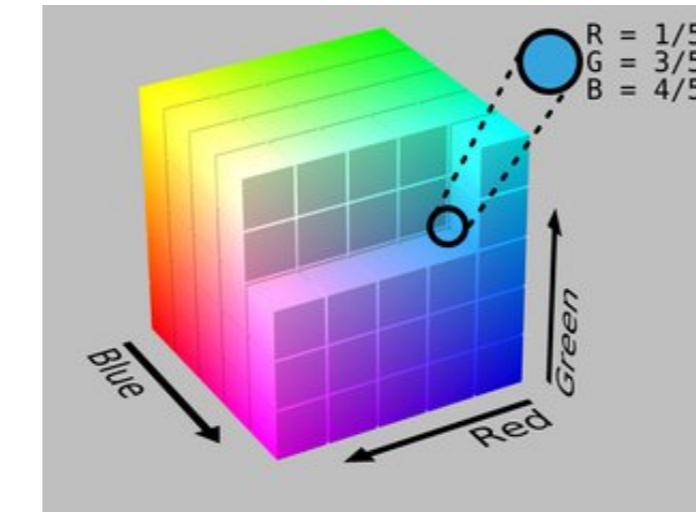
## Для справки

Мы рассматривали цветовую модель RGB, но есть и другие.

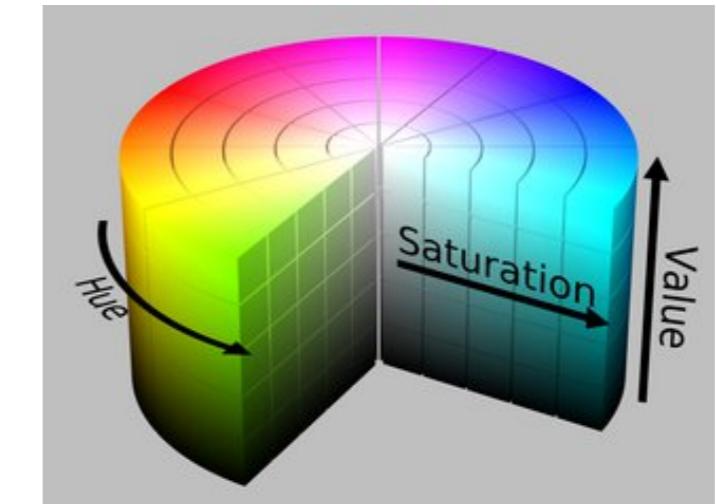
- RYB (red, yellow, blue) — красный, желтый, синий
- CMY / CMYK (cyan, magenta, yellow) — голубой, пурпурный, желтый
- HSL (hue, saturation, lightness) — оттенок, насыщенность, свет.
- HSV / HSB (hue, saturation, brightness) — оттенок, насыщенность, яркость

[Статья на Wikipedia](#)

Мы будем работать  
со стандартным RGB.



RGB



HSV



# Классификация изображений



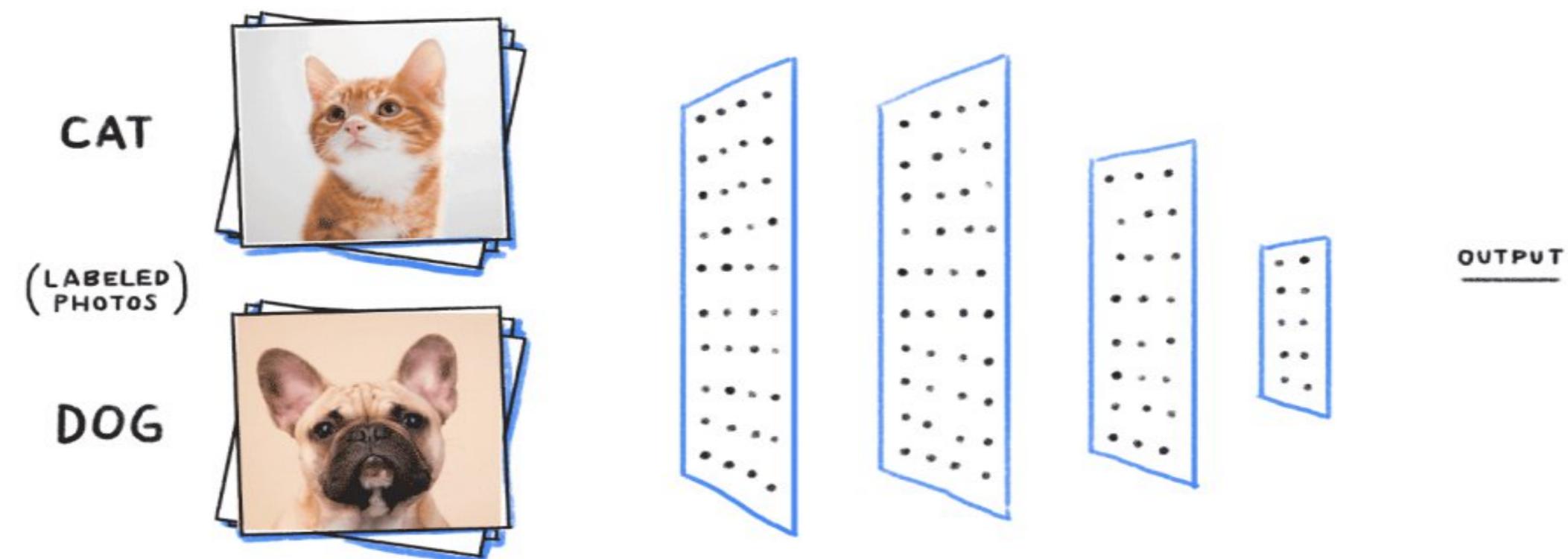
# Задача классификации изображений

$X$  — пространство картинок,

$Y$  — набор классов, например {кошка, собака}.

Требуется построить модель  $f: X \rightarrow Y$ ,

определяющую к какому классу относится объект на картинке.





# Проблемы классификации изображений

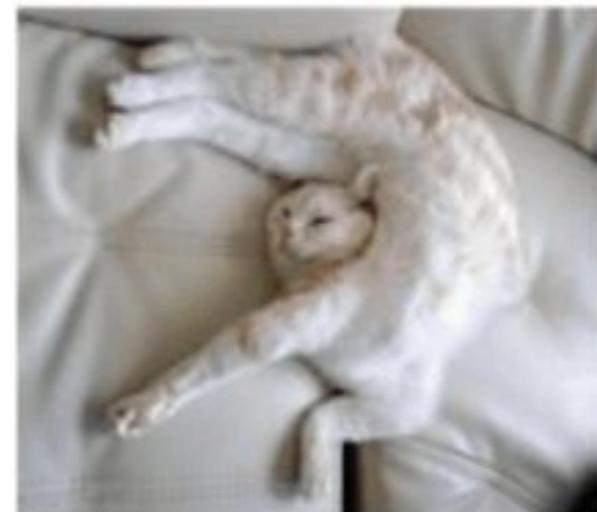
Разные углы обзора



Разный размер



Деформация



Перекрытие



Разная освещенность



Фоновые помехи



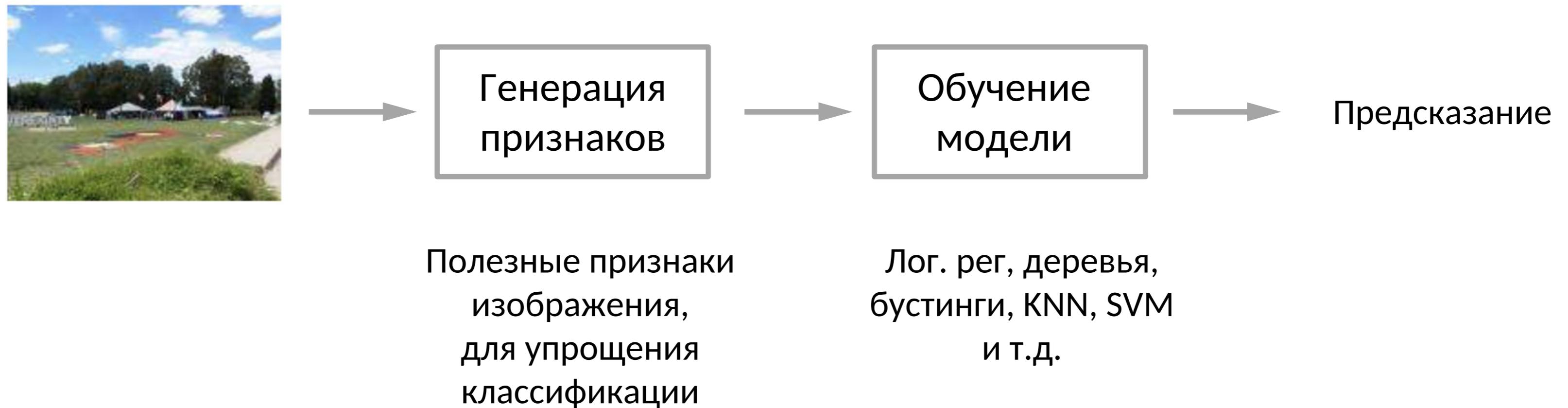
Разная форма





# Классификация изображений до нейросетей

Сгенерированные признаки подаем на вход модели машинного обучения.



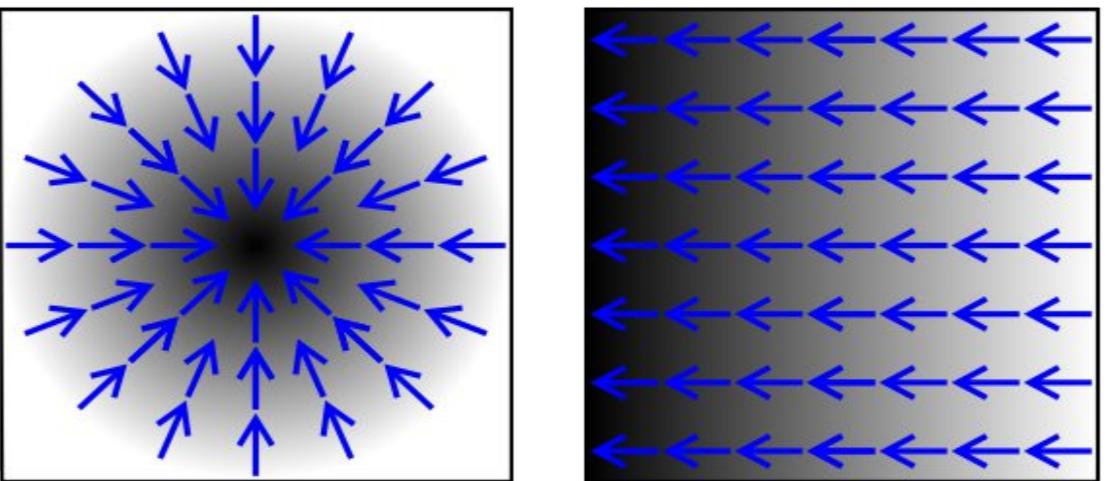


# Классификация изображений до нейросетей

## Генерация признаков

- Распределение цветов на картинке.
- Распределение градиентов интенсивности (HOG фичи).

Градиент в пикселе направлен в сторону наибольшего роста интенсивности в данной точке.



- Наличие и ориентация края в пикселе.

Есть различные алгоритмы для поиска краев, например детектор Canny.

- Наличие характерных фрагментов текстуры.

Задается большой словарь из характерных фрагментов разных текстур и осуществляется поиск похожих структур на изображении.

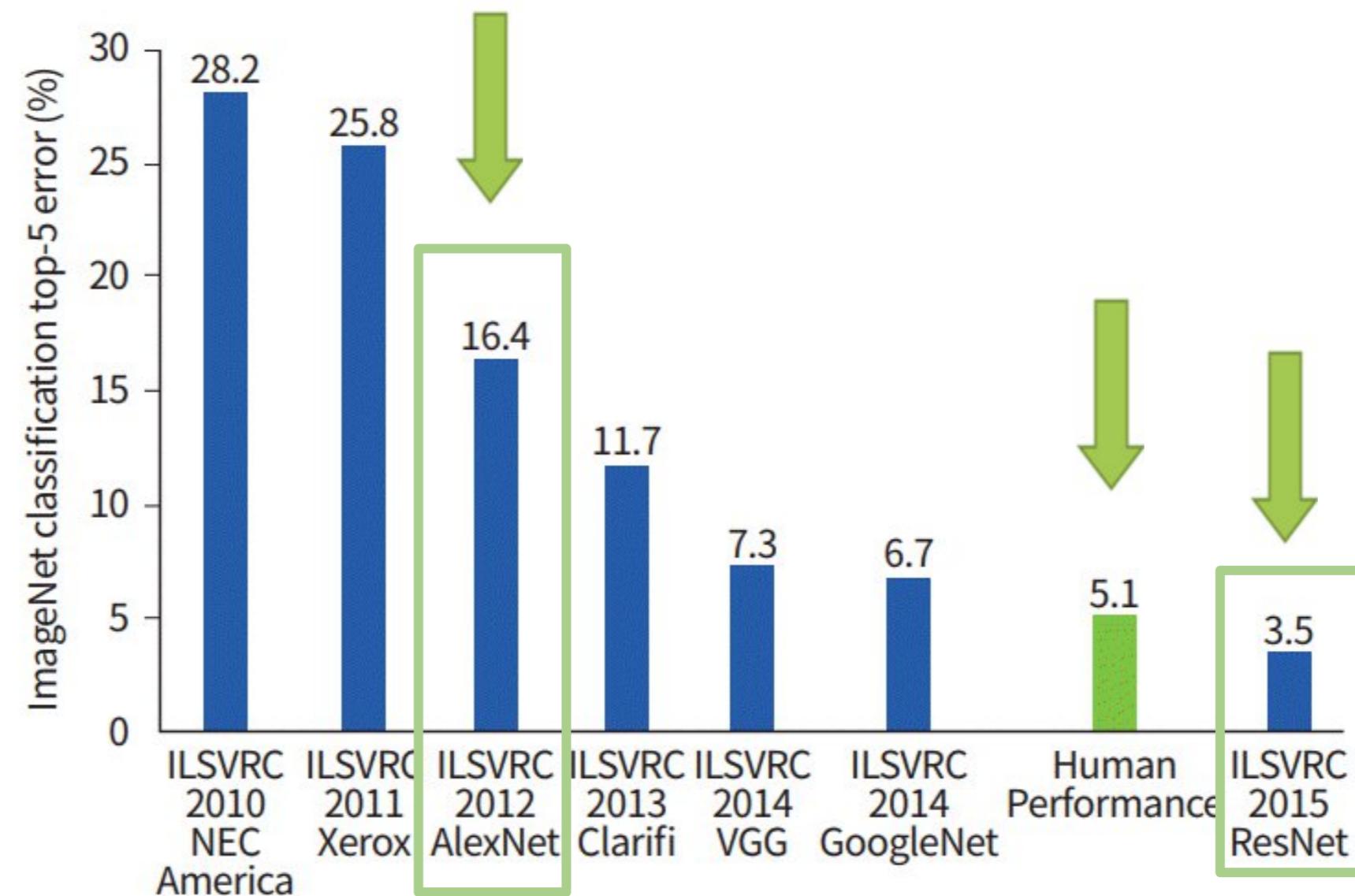
- Другие признаки. BoW, LBP признаки и прочее...





## Историческая справка

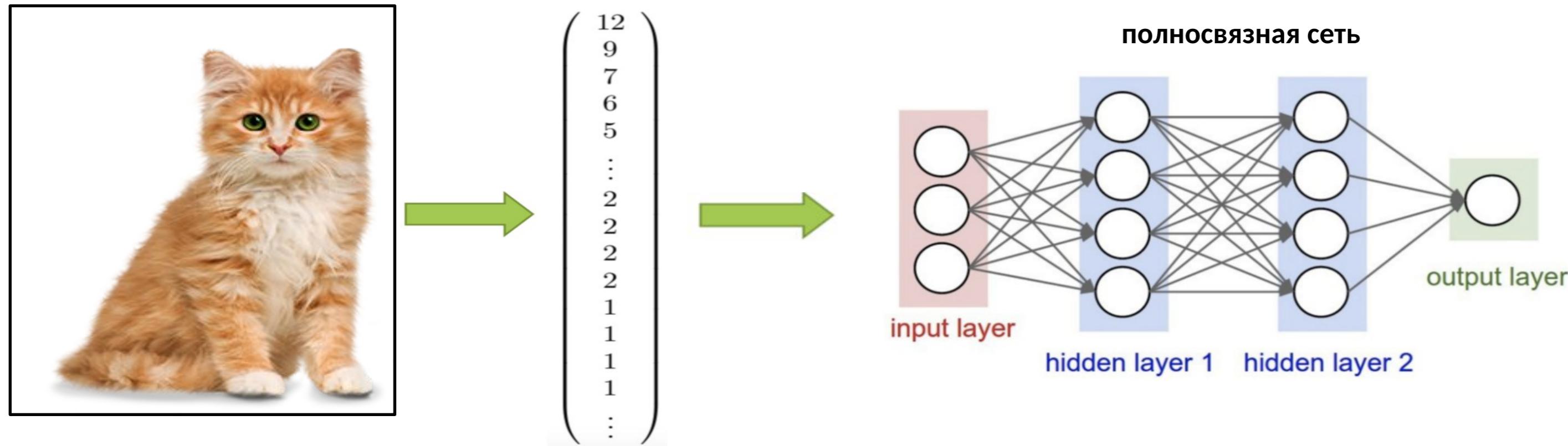
- 1998 год — Ян ЛеКун предложил архитектуру первой CNN - **LeNet**.
- ~2010 год — появилась имплементация **LeNet[5]**
- 2012 год — свёрточная нейронная сеть **AlexNet** победила в конкурсе ILSVRC.
- 2015 год — свёрточная нейронная сеть **ResNet** обогнала по качеству человека в конкурсе ILSVRC.





# Классификация изображений с помощью нейросетей

## Как классифицировать картинки?



## Почему такой подход плох?

1. Не учитывает явно взаимное расположение пикселей.
2. Расположение объекта на картинке не должно иметь значения для классификации.
3. Сеть нельзя сделать очень глубокой, она будет иметь слишком много параметров.



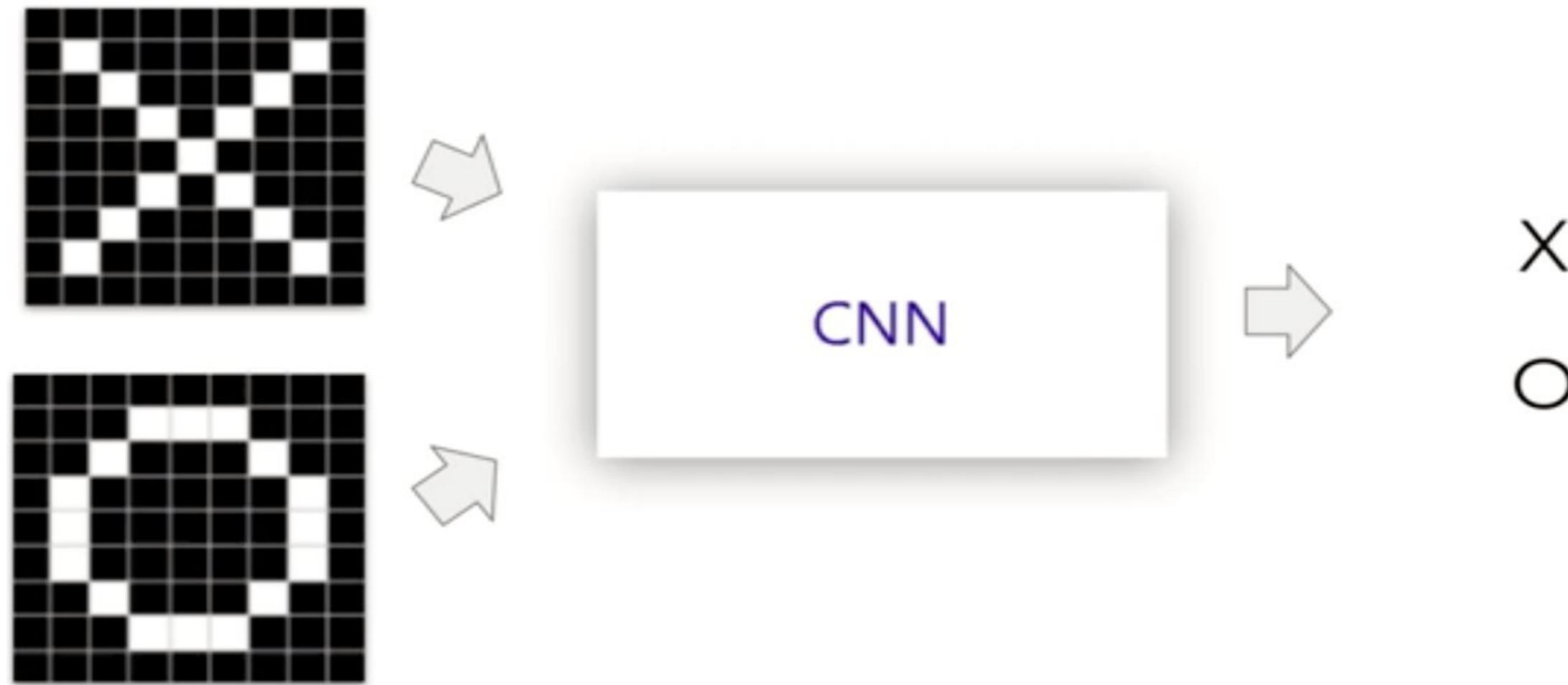
# Интуиция к CNN

## Поиграем в классификацию

Требуется уметь находить **крестики и нолики** на картинке.

Причем, объекты могут

- находиться в *разных местах* картинки,
- быть *не в точности равны* искомым паттернам.





# Классификация изображений с помощью нейросетей

## Идея: сделаем локальный поиск паттернов

Берем паттерн крестика и ищем похожий паттерн на картинке.

- Проходимся **окнами** по картинке
- Считаем **схожесть** этой части картинки и паттерна.
- Записываем результат в соотв. место в **матрице**.

Чем больше значение в матрице, тем больше похожа область картинки на паттерн.

0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	1	0
0	1	1	0	1	0	0
1	1	0	1	0	0	0

1	-1	1
-1	1	-1
1	-1	1

$K$

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

$I^*K$

$I$



# Классификация изображений с помощью нейросетей

## Как считать схожесть?

Чтобы оценить схожесть двух паттернов перемножим их **скалярно**:

1. Умножим матрицы поэлементно.
2. Сложим числа полученной матрицы.

0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	1	0
0	1	1	0	1	0	0
1	1	0	1	0	0	0

1	-1	1
-1	1	-1
1	-1	1

**K**

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

**I\*K**

**I**



# 2D свертка / 2D convolution



# 2D свертка / 2D convolution

Фильтр / ядро, представляющий из себя матрицу весов, пробегает по исходным данным и вычисляет скалярные произведения с той частью изображения, над которой он сейчас находится.

**Изображение**

$x_{11}$	$x_{12}$	$x_{13}$
$x_{21}$	$x_{22}$	$x_{23}$
$x_{31}$	$x_{32}$	$x_{33}$

\*

$w_{11}$	$w_{12}$
$w_{21}$	$w_{22}$

**Фильтр**

=

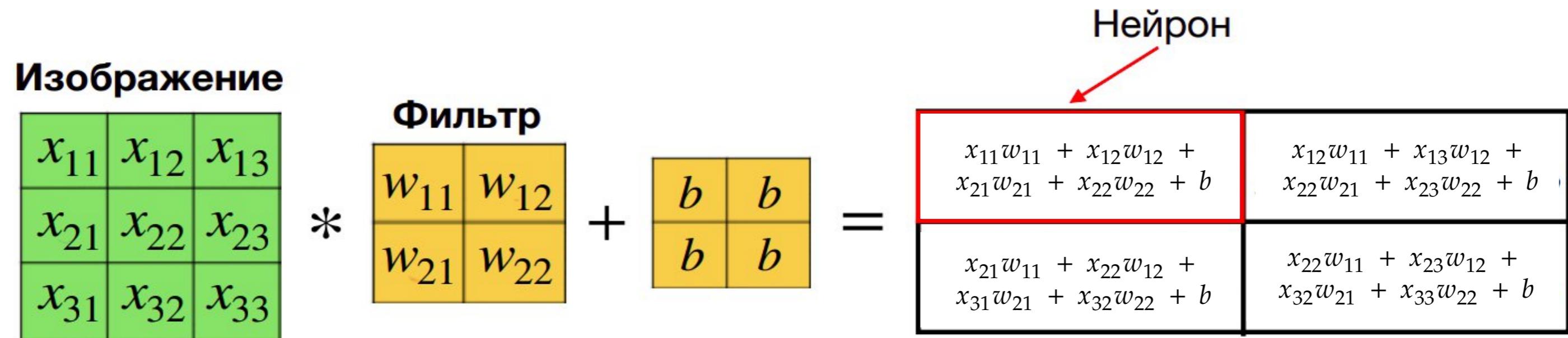
$x_{11}w_{11} + x_{12}w_{12} +$ $x_{21}w_{21} + x_{22}w_{22}$	



# 2D свертка / 2D convolution

Фильтр / ядро, представляющий из себя матрицу весов, пробегает по исходным данным и вычисляет скалярные произведения с той частью изображения, над которой он сейчас находится.

Ядро повторяет эту процедуру с каждой локацией, над которой оно скользит.



После чего к каждому элементу также добавляется смещение  $b$ .

Каждый отдельный элемент в полученной матрице — нейрон.

Формула свёртки:  $(I * F)_{m,n} = \sum_{i=1}^M \sum_{j=1}^M w_{ij} \cdot x_{m+i-1, n+j-1} + b$

где  $I$  — изображение,  $F$  — фильтр размера  $(M, M)$ .



# 2D свертка / 2D convolution

Веса в фильтре и смещение — обучаемые параметры.

Размер фильтра — гиперпараметр.

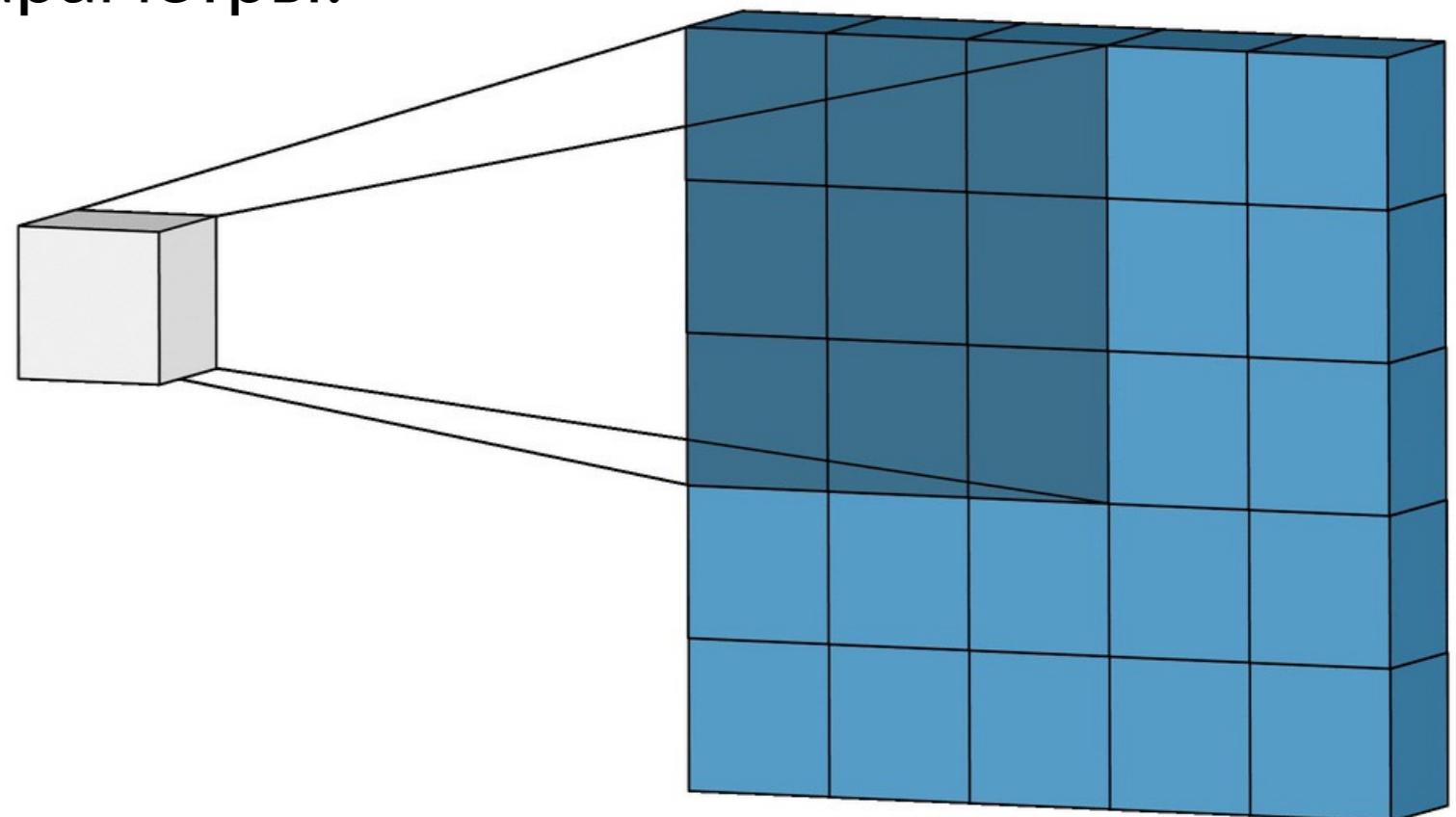
Исходные данные

3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

Ядро

Выход

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0





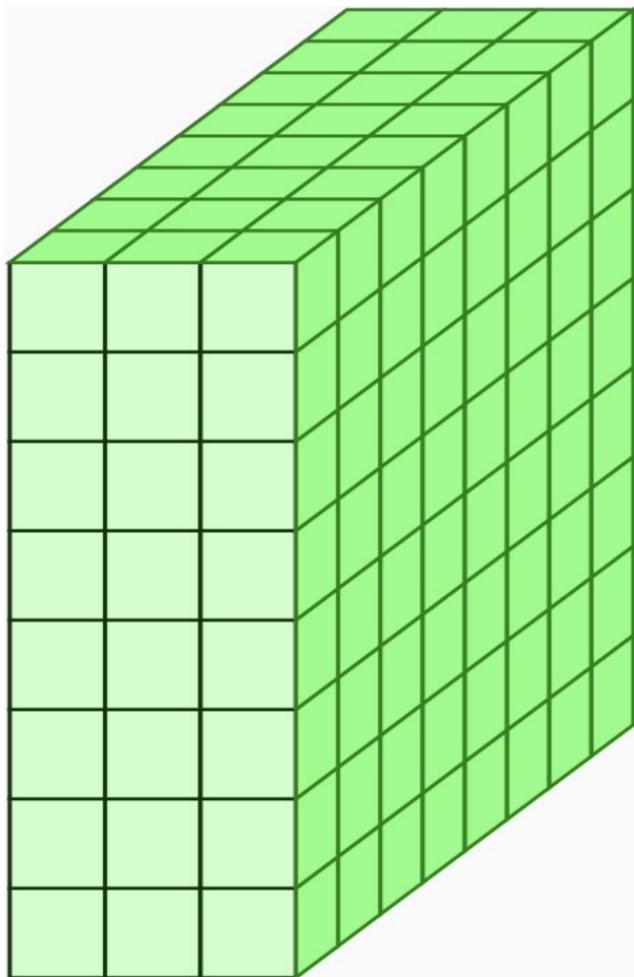
# Многоканальный вход

Вход — трехмерные тензоры размера  $H \times W \times C$  (высота, ширина, каналы).

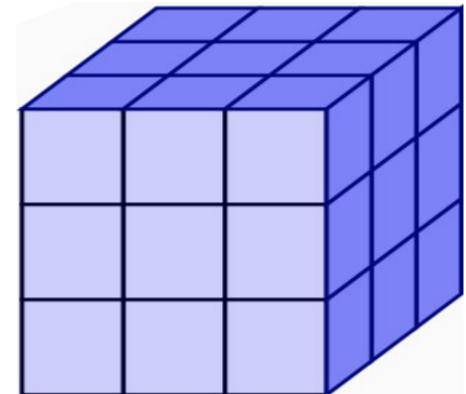
Ядро — трехмерная матрица размера  $k_x \times k_y \times C$ .

Обычно  $k_x = k_y$ , где  $k_x, k_y$  — гиперпараметры.

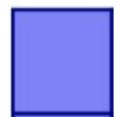
Изображение  $8 \times 8 \times 3$



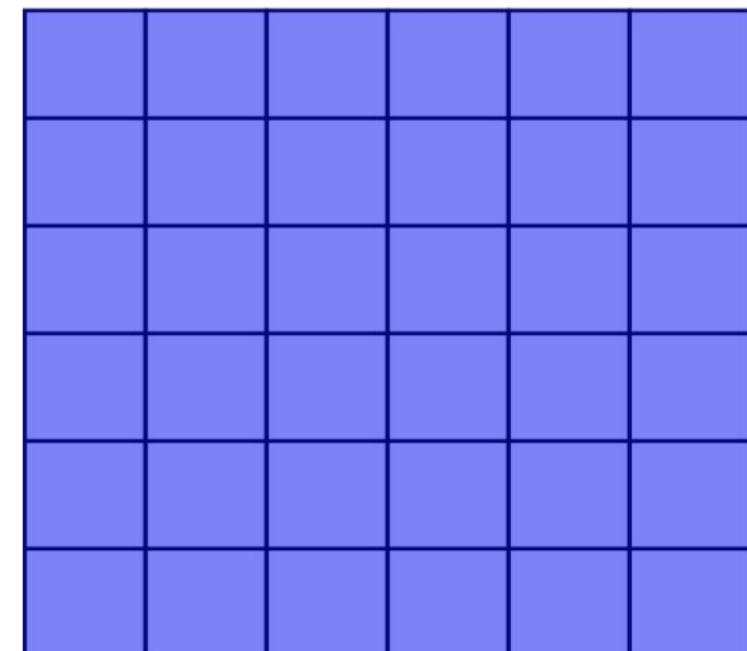
Фильтр  $3 \times 3 \times 3$



Смещение



Карта  $6 \times 6$





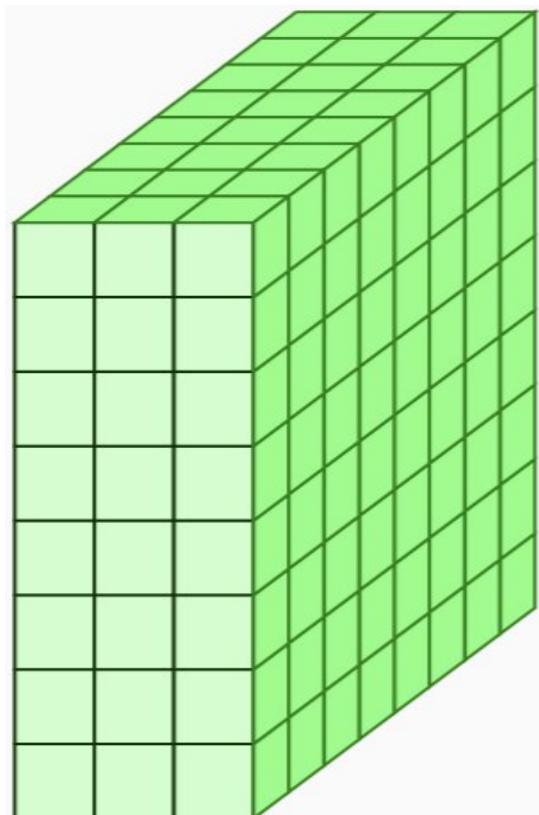
# Много фильтров

Один фильтр → одна карта, соответствующая одному паттерну.

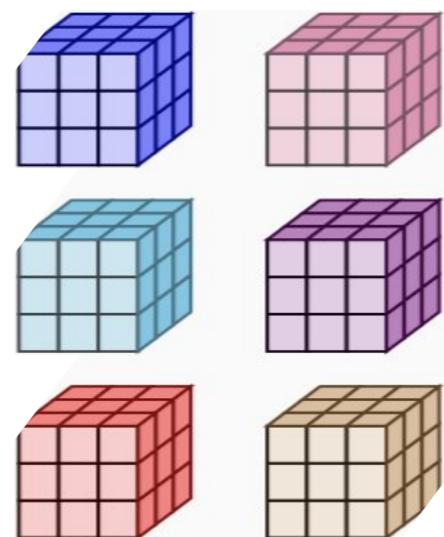
Возьмем  $K$  разных фильтров и применим свёртку с ними.

Получим  $K$  карт на выходе.

Изображение  $8 \times 8 \times 3$



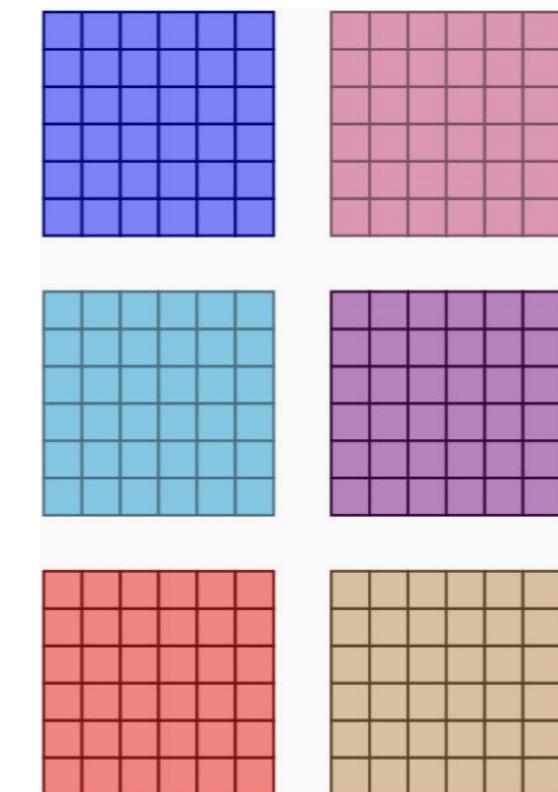
6 фильтров  $3 \times 3 \times 3$



Смещения



6 карт  $6 \times 6$





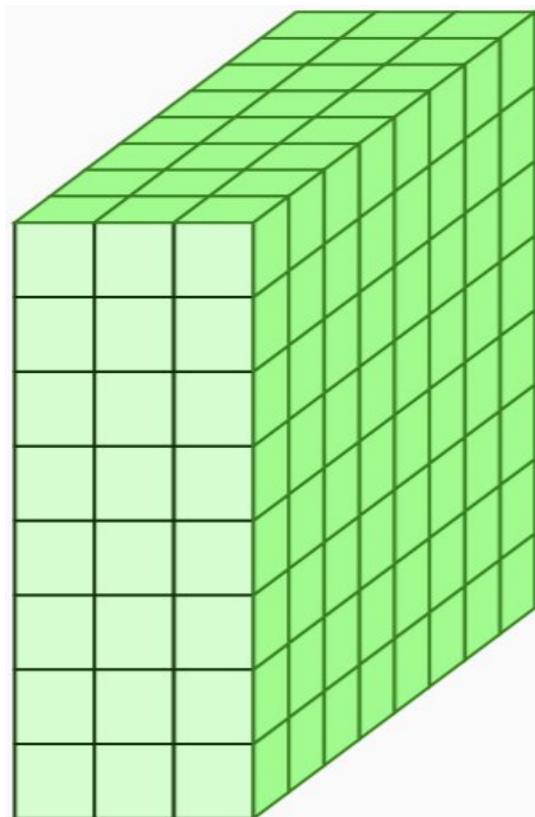
# Много фильтров

Один фильтр → одна карта, соответствующая одному паттерну.

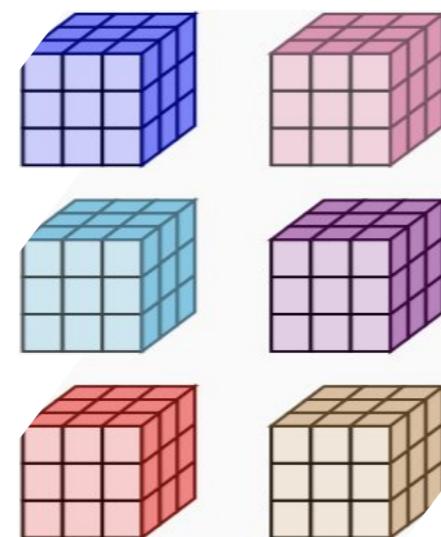
Возьмем  $K$  разных фильтров и применим свёртку с ними.

Получим  $K$  карт на выходе.

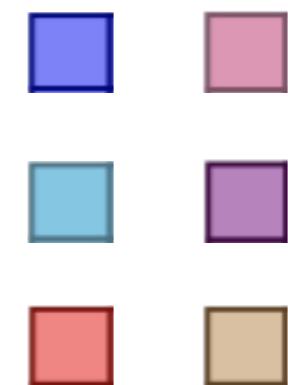
Изображение  $8 \times 8 \times 3$



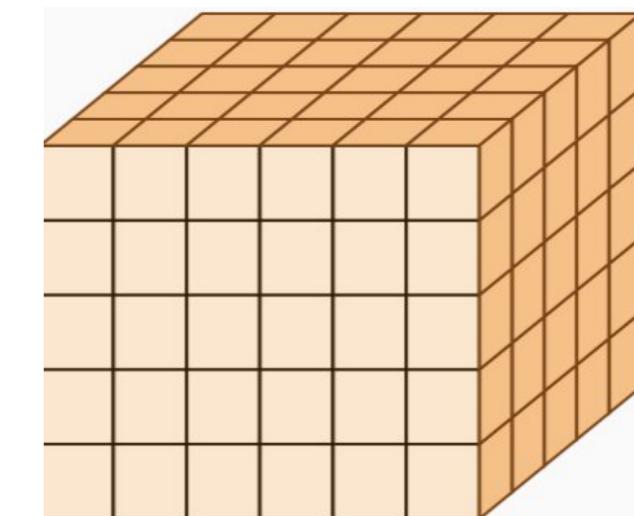
6 фильтров  $3 \times 3 \times 3$



Смещения



6 карт  $6 \times 6$



Выходную матрицу  $6 \times 6 \times 6$  можно использовать  
как вход следующего свёрточного слоя.



# Отступ / Padding

## Проблема

1. Крайние пиксели никогда не оказываются в центре ядра.
2. Выходной размер получается меньше входного.

**Padding** добавляет по краям поддельные пиксели.

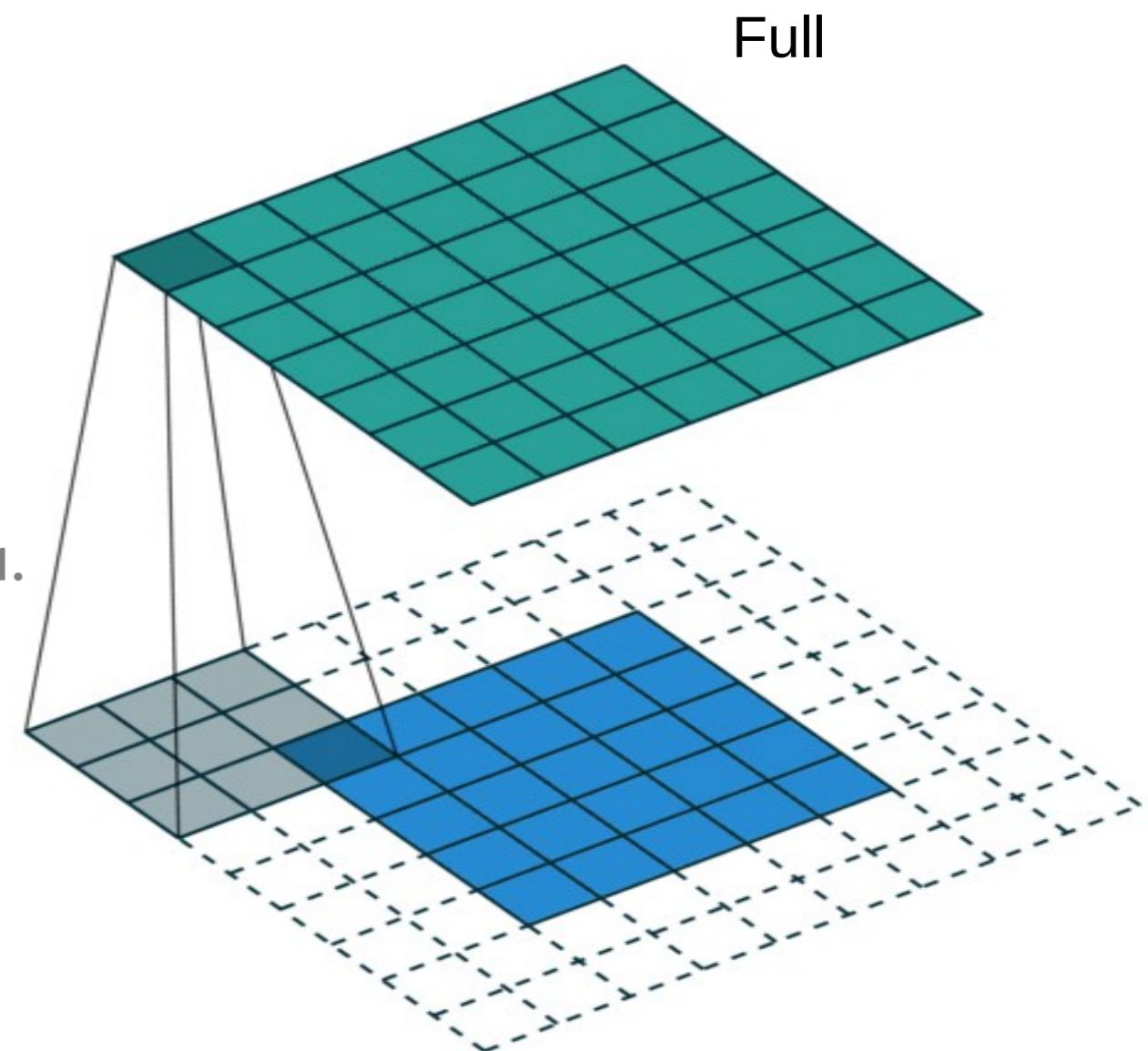
Тогда позволяем краевым неподдельным пикселям оказываться в центре ядра.

## Какими значениями заполнять?

- Нулями (zero-padding)  
Самый популярный вариант.  
Сеть учится понимать, что окно находится на границе картинки.
- Значением ближайшего пикселя

## Виды

- Same: высота и ширина выходн. и входн.изобр. совпадают.
- Valid: отсутствие паддинга.
- Full: высота и ширина выходн. изобр. больше входн.





# Шаг / Striding

**Идея.**

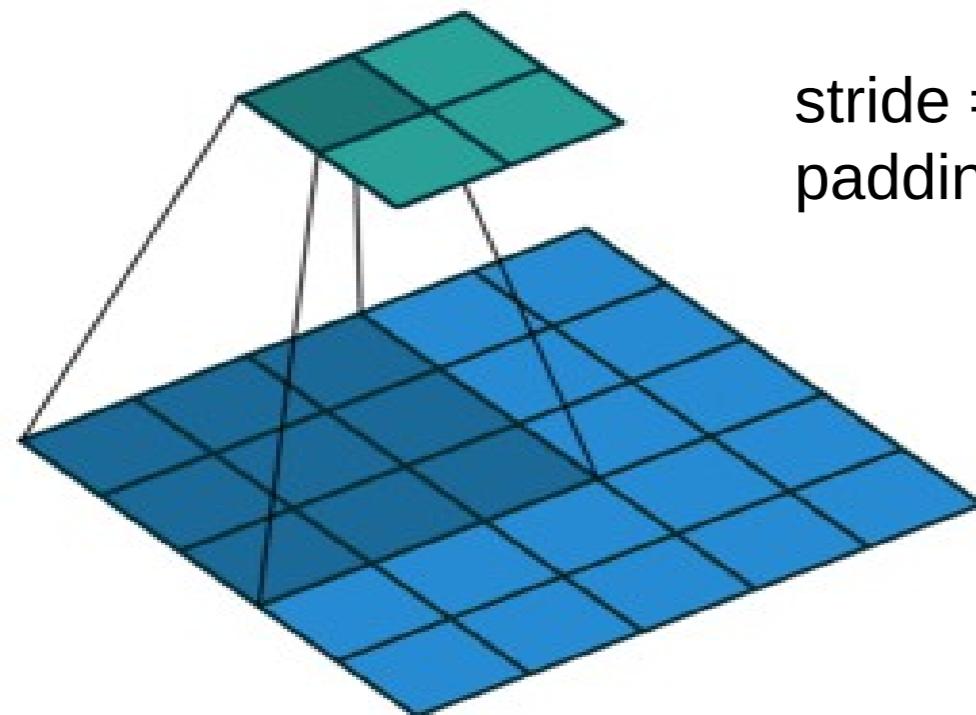
Пропустим некоторые области, над которыми скользит ядро.

Stride = 1 - окна сдвигаются на 1 пиксель.

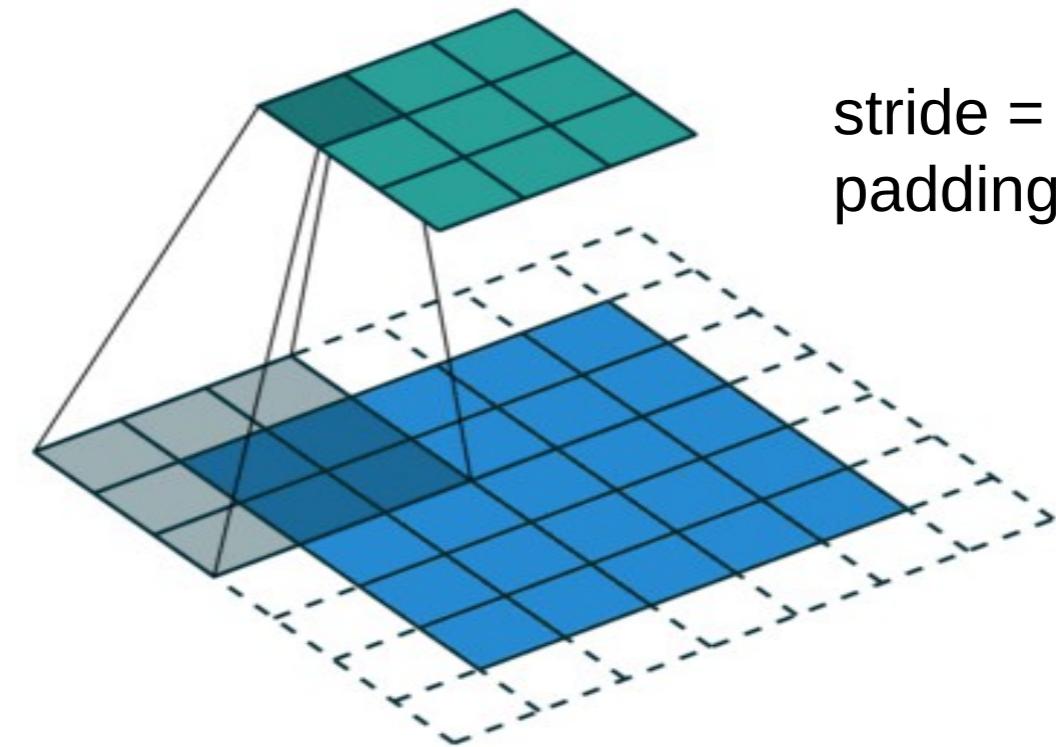
Stride = 2 - окна сдвигаются на 2 пикселя.

Таким образом, размер выходной матрицы уменьшается.

Искомые паттерны всё равно должны найтись,  
если исходное изображение достаточно большое.



stride = 2  
padding=0



stride = 2  
padding = 1



# Больше сверток

- Одной свёрткой можем узнать только наличие простых паттернов на картинке.
- Одной свёрткой не можем найти сложные паттерны.

Если фильтр будет изображать лицо кота, то мы вряд ли найдем что-то похожее на картинке с котом, ведь коты бывают разные.

## Идея

Сделаем несколько сверток подряд.

- Первая свёртка будет искать простые паттерны на исходной картинке.
- Вторая будет искать простые паттерны уже на картах после первой свёртки. Простые паттерны из простых паттернов — уже более сложные паттерны.
- Третья ...





# Область видимости / Receptive field

Область видимости / *Receptive field* –  
размер области исходного изображения, которую видит один нейрон.

## Пример

Рассмотрим одноканальное изображение  
и многослойную свёрточную сеть с фильтрами свёртки размера .

Будем рассматривать разные слои и отвечать на вопрос:

**“Сколько пикселей исходного изображения доступно нейрону этого слоя?”**



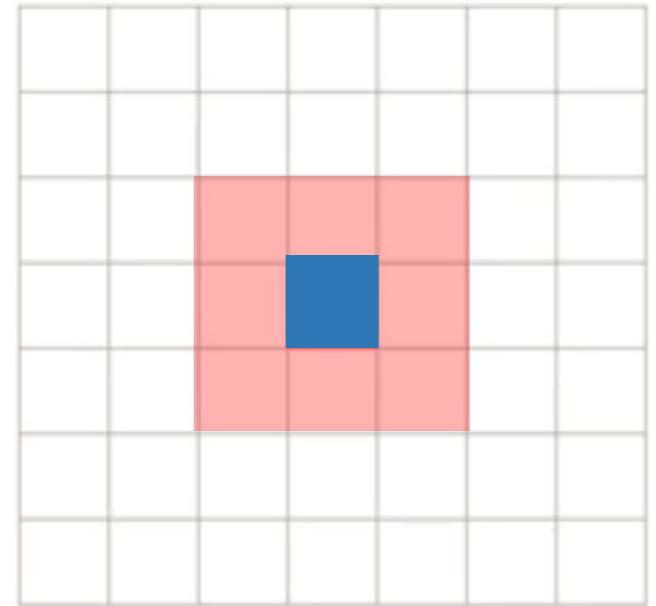
# Область видимости / Receptive field

## Область видимости слоя 1

Нейрон на слое 1

— скалярное произведение фильтра  
на область изображения.

Его область видимости — 9 пикселей.





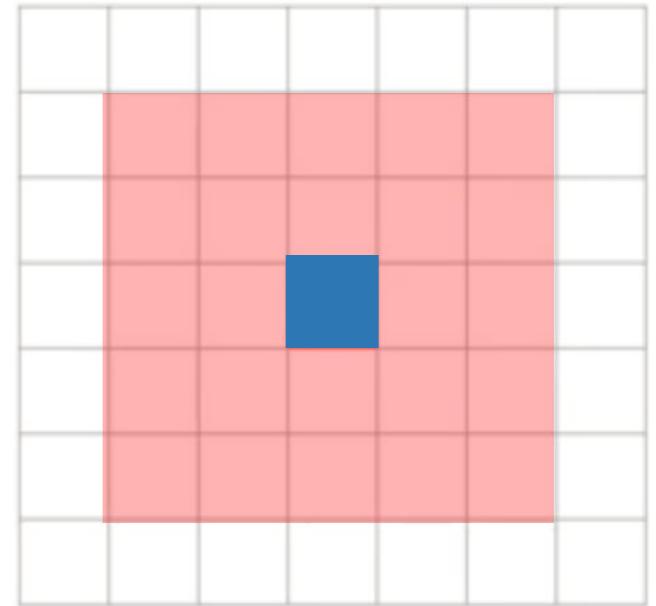
# Область видимости / Receptive field

**Область видимости слоя 2**

Нейрон на слое 2

— скалярное произведение фильтра  
на область слоя 1.

Его область видимости — 25 пикселей.



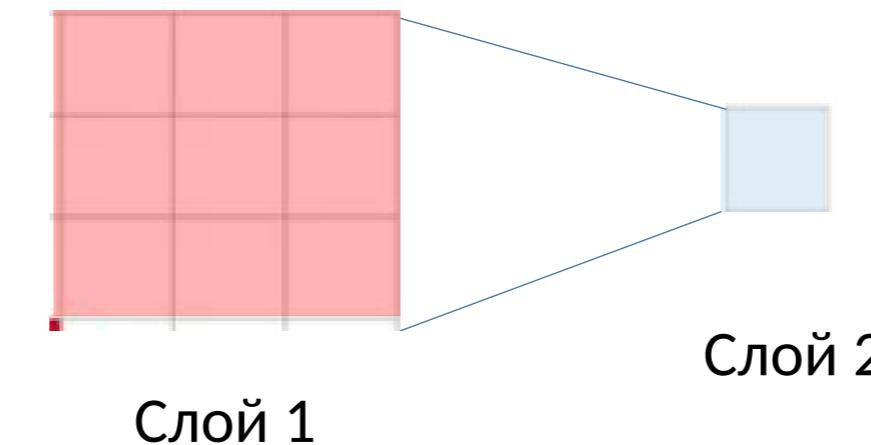
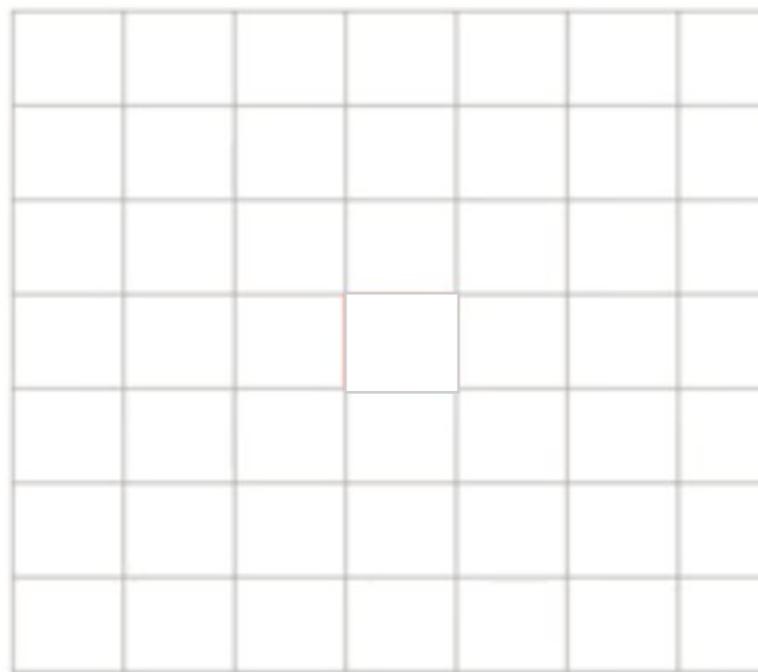


# Область видимости / Receptive field

Какой будет область видимости **слоя 2**, если использовать свертку с **шагом 2**?

Нейрон на слое 2 – скалярное произведение фильтра на область слоя 1.

Для слоя 2 область видимости состоит из 9 нейронов 1 слоя



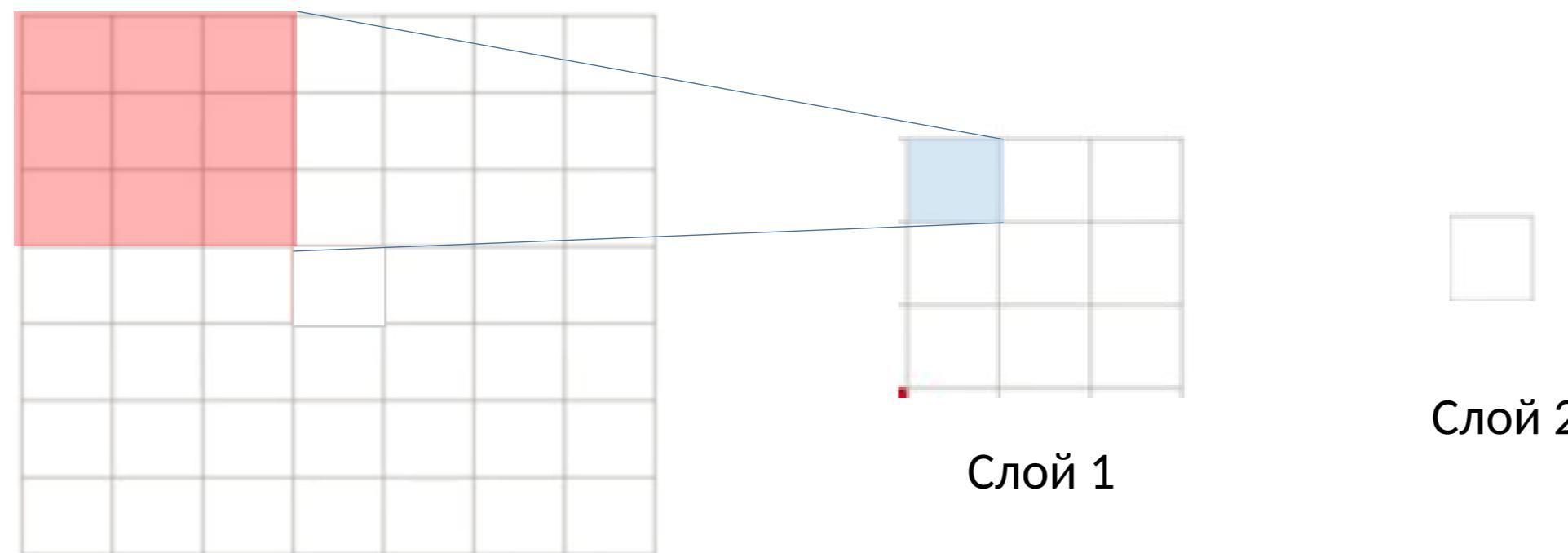


# Область видимости / Receptive field

Какой будет область видимости **слоя 2**, если использовать свертку с **шагом 2**?

Нейрон на слое 2 – скалярное произведение фильтра на область слоя 1.

Для слоя 1 область видимости состоит из 9 исходного изображения.



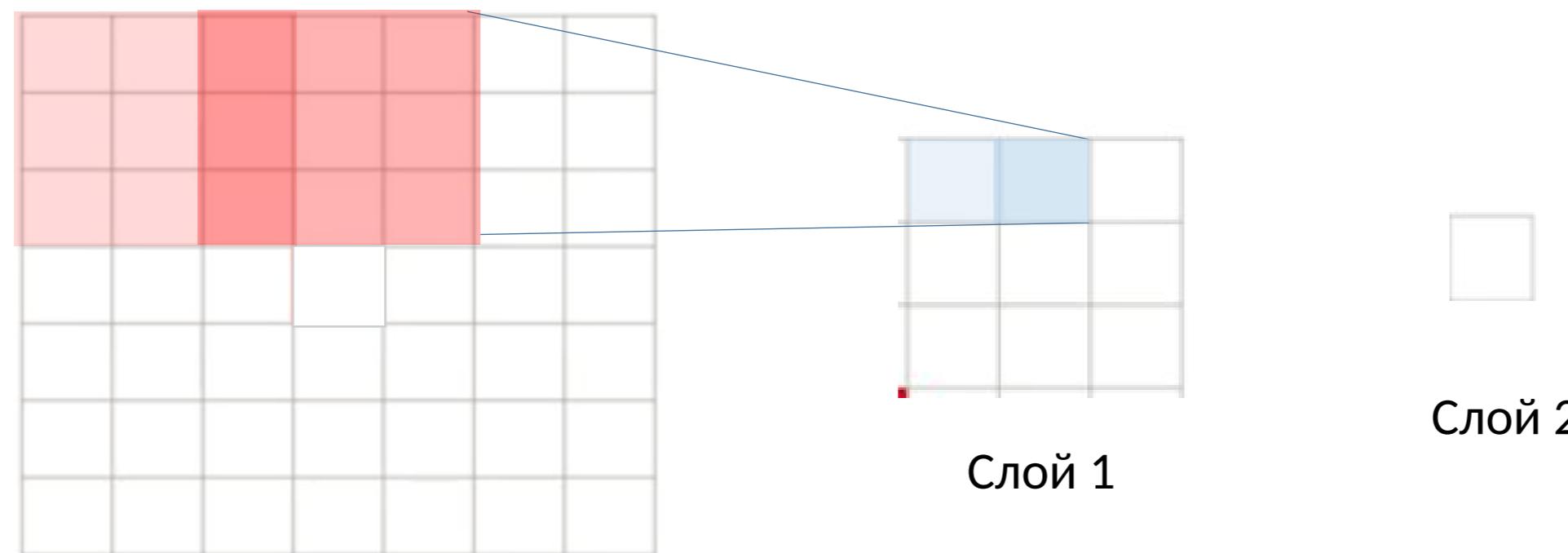


# Область видимости / Receptive field

Какой будет область видимости **слоя 2**, если использовать свертку с **шагом 2**

Нейрон на слое 2 – скалярное произведение фильтра на область слоя 1.

Для слоя 1 область видимости состоит из 9 исходного изображения.



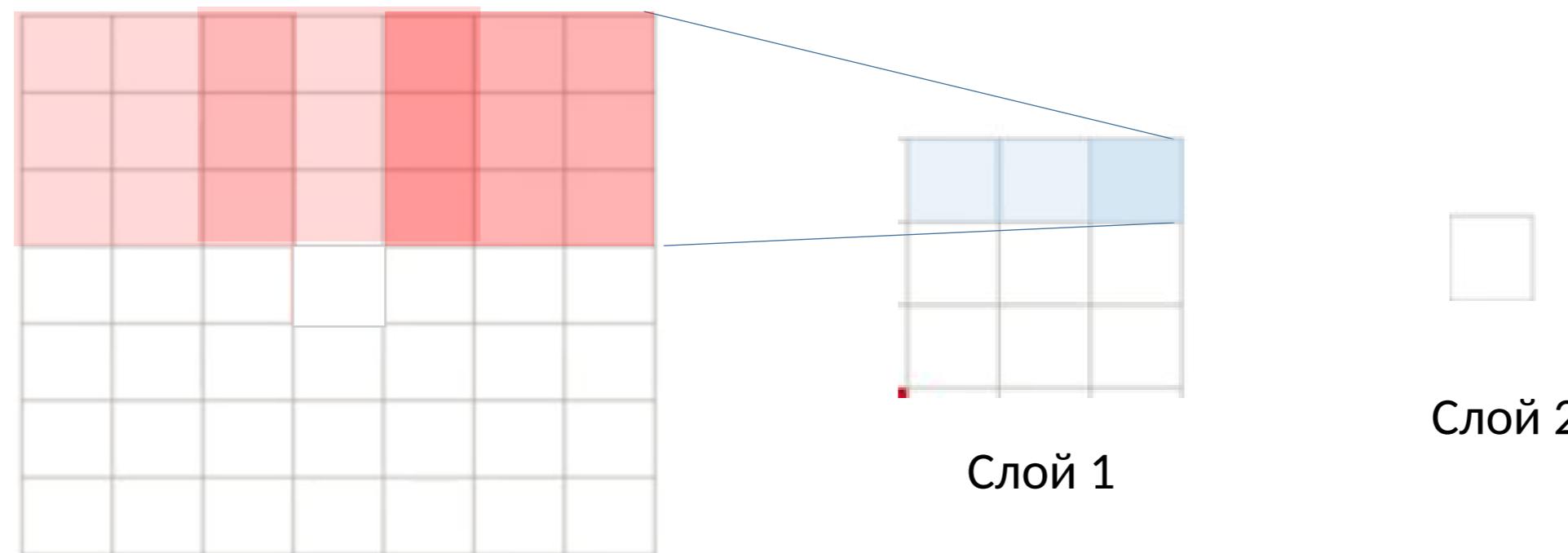


# Область видимости / Receptive field

Какой будет область видимости **слоя 2**, если использовать свертку с **шагом 2**?

Нейрон на слое 2 – скалярное произведение фильтра на область слоя 1.

Для слоя 1 область видимости состоит из 9 исходного изображения.

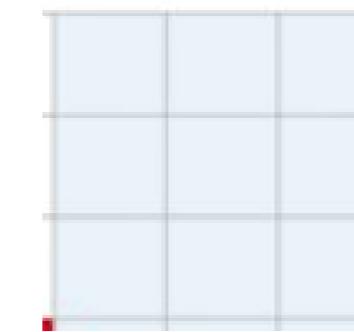




# Область видимости / Receptive field

Какой будет область видимости **слоя 2**, если использовать свертку с **шагом 2**?

Для слоя 2 область видимости состоит из 49 пикселей исходного изображения. В случае, когда  $\text{stride} = 1$  она была равна 25 пикселям.



Слой 1



Слой 2



Почему для операции свёртки выполним backpropagation?

Backpropagation

Посмотрим на пример свертки.

**Изображение**

$x_{11}$	$x_{12}$	$x_{13}$
$x_{21}$	$x_{22}$	$x_{23}$
$x_{31}$	$x_{32}$	$x_{33}$

**Фильтр**

$w_{11}$	$w_{12}$
$w_{21}$	$w_{22}$

$*$

$+$

$b$	$b$
$b$	$b$

=

$x_{11}w_{11} + x_{12}w_{12} + x_{21}w_{21} + x_{22}w_{22} + b$	$x_{12}w_{11} + x_{13}w_{12} + x_{22}w_{21} + x_{23}w_{22} + b$
$x_{21}w_{11} + x_{22}w_{12} + x_{31}w_{21} + x_{32}w_{22} + b$	$x_{22}w_{11} + x_{23}w_{12} + x_{32}w_{21} + x_{33}w_{22} + b$

↑  
операция  
свертки

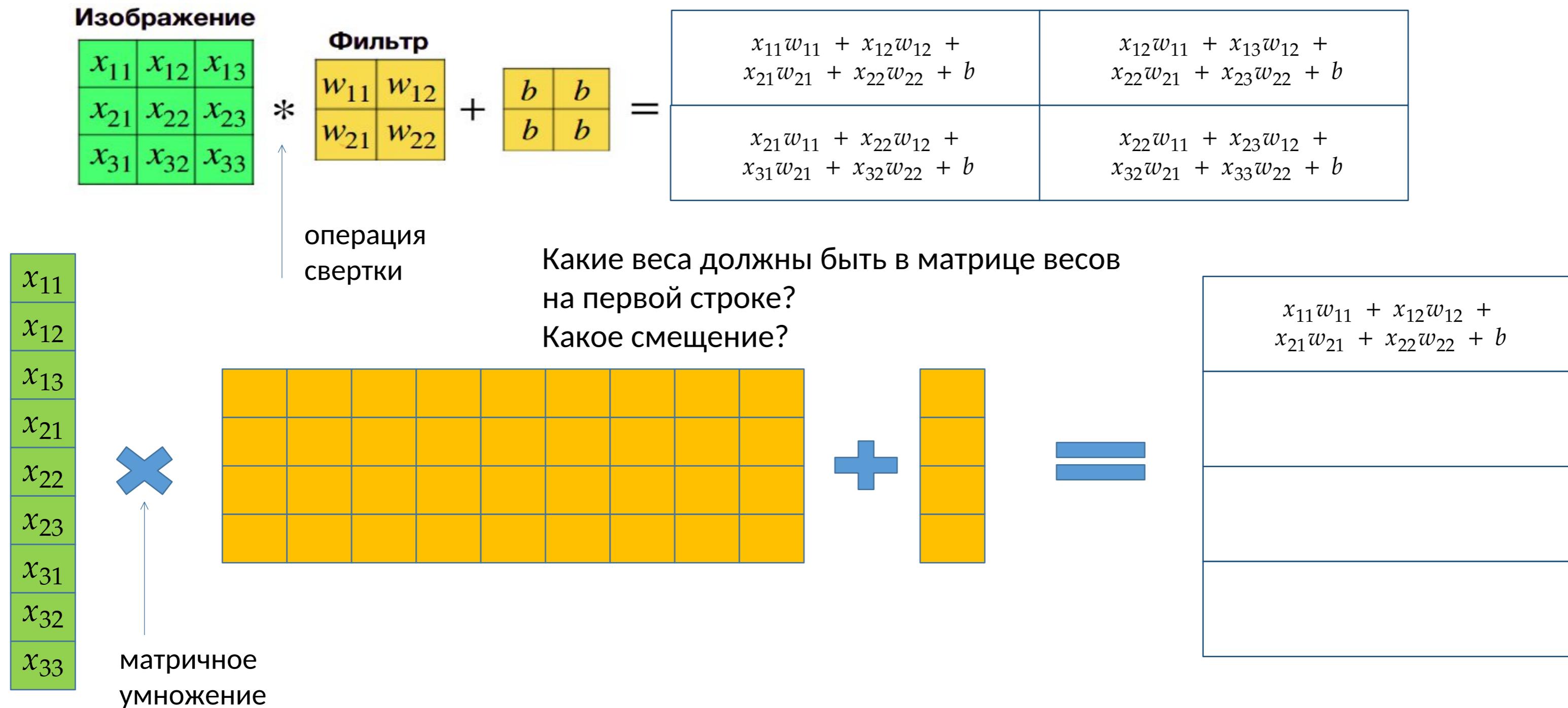
Результат похож на **линейное преобразование** исходной картинки.



# Backpropagation

Применим полносвязный слой.

1. Вытянем картинку в вектор.
2. Полученный вектор домножим на матрицу весов и добавим смещение.

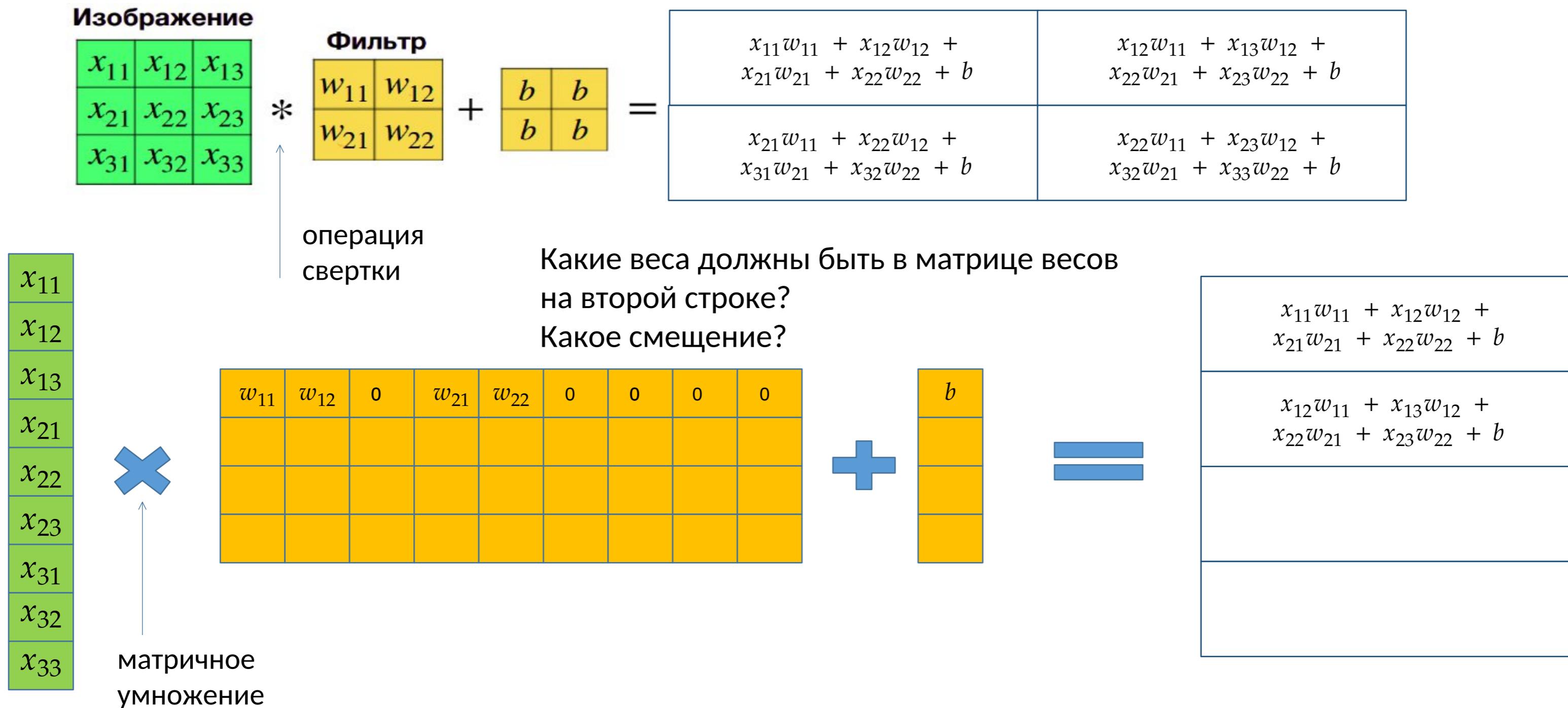




# Backpropagation

Применим полносвязный слой.

1. Вытянем картинку в вектор.
2. Полученный вектор домножим на матрицу весов и добавим смещение.



# Backpropagation

Применим полносвязный слой.

1. Вытянем картинку в вектор.
  2. Полученный вектор умножим на матрицу весов и добавим смещение.





# Backpropagation

Применим полносвязный слой.

1. Вытянем картинку в вектор.
2. Полученный вектор домножим на матрицу весов и добавим смещение.

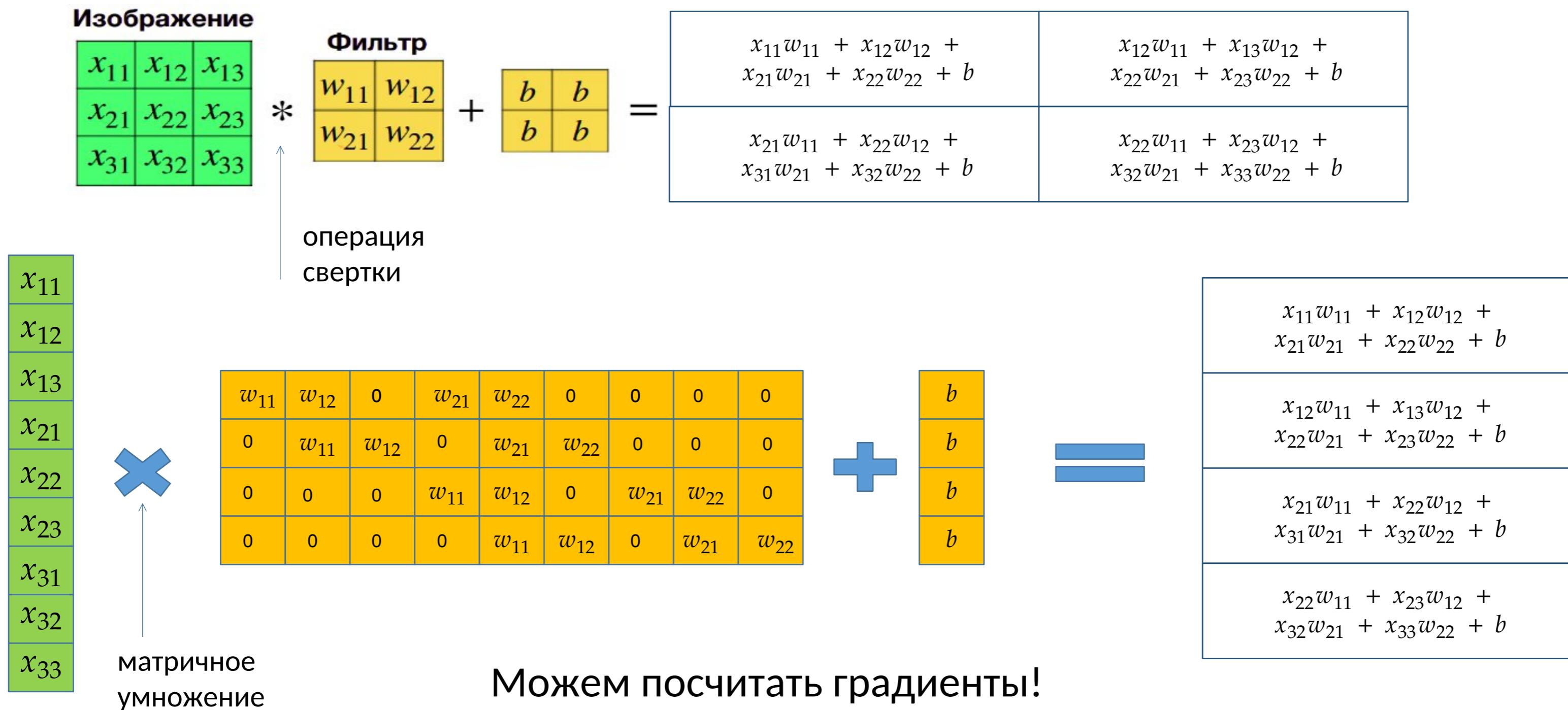




# Backpropagation

Применим полносвязный слой.

1. Вытянем картинку в вектор.
2. Полученный вектор домножим на матрицу весов и добавим смещение.
3. Результат можно снова собрать в картинку.





# Подсчет размеров

Пусть входная картинка имеет размер  $H_{in} \times W_{in} \times C_{in}$ .

Размер применяемого фильтра  $M \times M \times C_{in}$ .

## Пример 1

Применяем операцию свертки с **stride=1, padding=0**.

Какой размер будет у выхода после свертки?

Размер карты:

$$H_{out} = H_{in} - M + 1$$

$$W_{out} = W_{in} - M + 1$$

Применив  $C_{out}$  фильтров, получим размер

$$H_{out} \times W_{out} \times C_{out}$$



# Подсчет размеров

Пусть входная картинка имеет размер  $H_{in} \times W_{in} \times C_{in}$ .

Размер применяемого фильтра  $M \times M \times C_{in}$ .

## Пример 2

Применяем операцию свертки с **stride=S, padding=0**.

Какой размер будет у выхода после свертки?

Окна, с которыми берем скаляр. произвед. имеют координаты по горизонтали

$[1, M], [1 + S, S + M], [1 + 2S, 2S + M], \dots, [1 + kS, kS + M]$ ,

где  $k$  – такое целое число, при котором  $[1 + kS, kS + M]$  – последнее окно.

Тогда должно быть выполнено:  $kS + M \leq H_{in}$ , значит

$$H_{out} = k + 1 = \lfloor (H_{in} - M) / S \rfloor + 1$$

Аналогично

$$W_{out} = k + 1 = \lfloor (W_{in} - M) / S \rfloor + 1$$

Применив  $C_{out}$  фильтров, получим размер

$$H_{out} \times W_{out} \times C_{out}$$



# Подсчет размеров

Пусть входная картинка имеет размер  $H_{in} \times W_{in} \times C_{in}$ .

Размер применяемого фильтра  $M \times M \times C_{in}$ .

## Пример 3

Применяем операцию свертки с **stride=S**, **padding=P**.

Какой размер будет у выхода после свертки?

Добавляем  $P$  пикселей с каждой стороны входного изображения

$$H_{out} = \lfloor (H_{in} - M + 2P) / S \rfloor + 1$$

Аналогично

$$W_{out} = k + 1 = \lfloor (W_{in} - M + 2P) / S \rfloor + 1$$

Применив  $C_{out}$  фильтров, получим размер

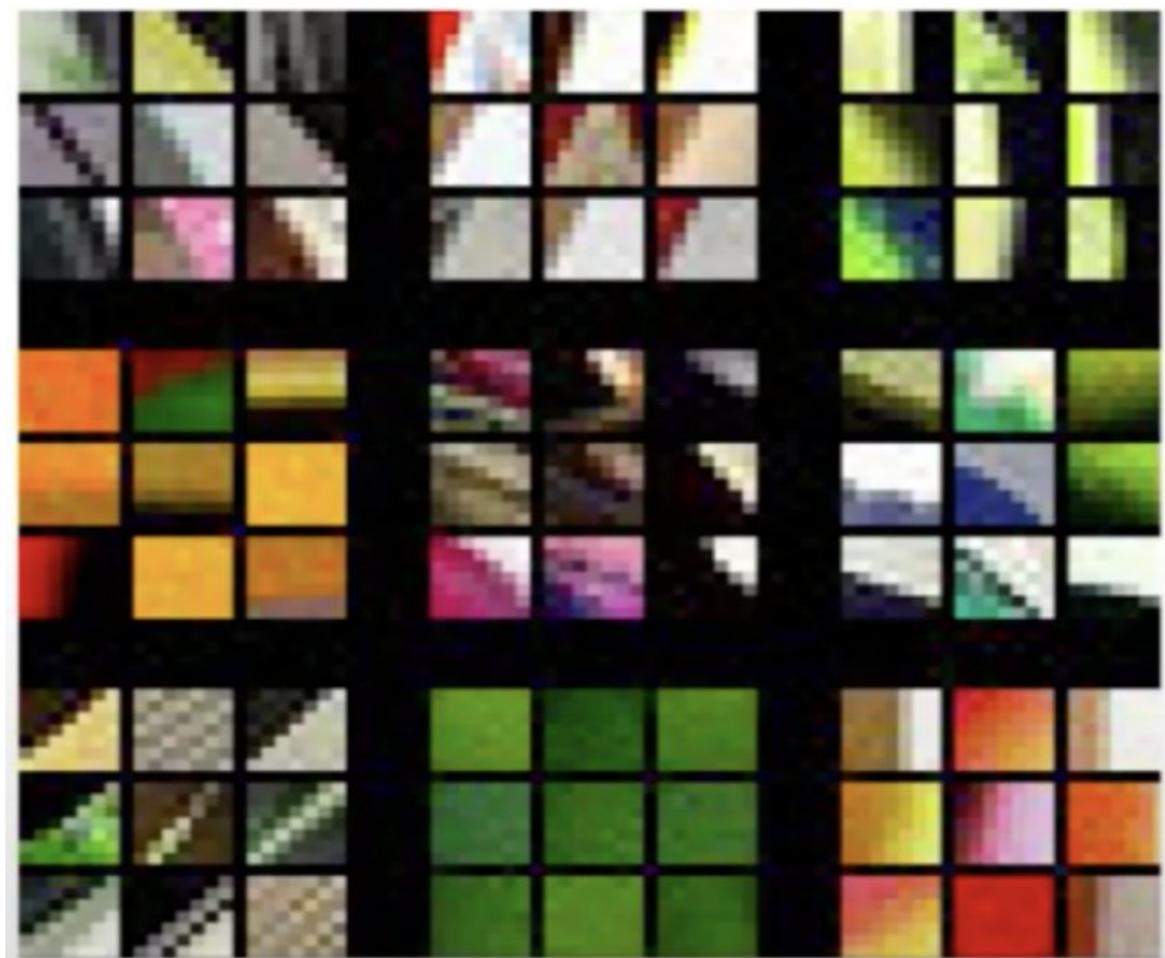
$$H_{out} \times W_{out} \times C_{out}$$



# Интерпретируемость

- Рассматриваем некоторый нейрон в свёртке.
- Для данного нейрона возьмем объекты, на которых значения выхода нейрона самое большое.
- Визуализируем receptive field этого нейрона у этих объектов, то есть посмотрим какая часть изображения видна данному нейрону.

**Первый  
свёрточный слой**



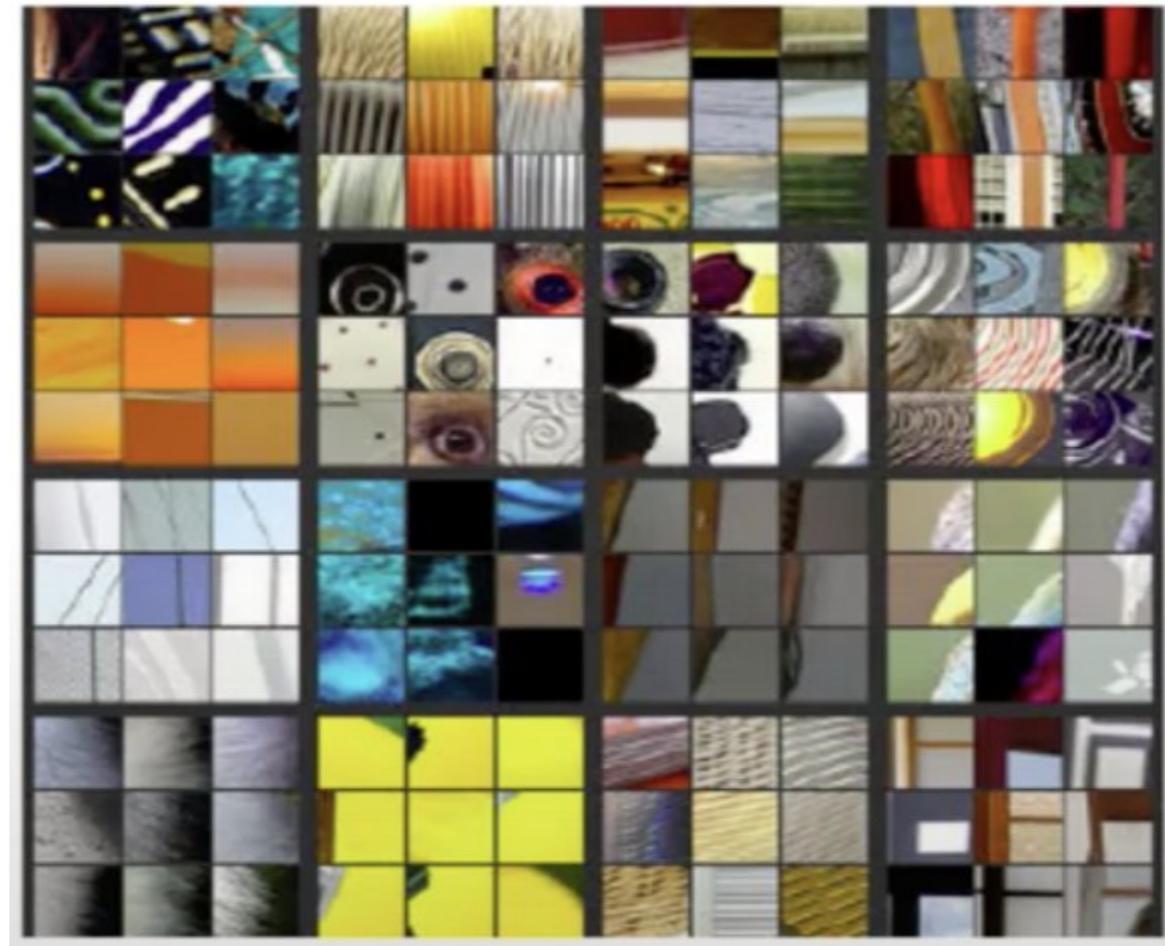
Визуализация  
receptive field  
для одного нейрона



# Интерпретируемость

- Рассматриваем некоторый нейрон в свёртке.
- Для данного нейрона возьмем объекты, на которых значения выхода нейрона самое большое.
- Визуализируем receptive field этого нейрона у этих объектов, то есть посмотрим какая часть изображения видна данному нейрону.

**Второй  
свёрточный слой**



Визуализация  
receptive field  
для одного нейрона



# Интерпретируемость

- Рассматриваем некоторый нейрон в свёртке.
- Для данного нейрона возьмем объекты, на которых значения выхода нейрона самое большое.
- Визуализируем receptive field этого нейрона у этих объектов, то есть посмотрим какая часть изображения видна данному нейрону.

**Третий  
свёрточный слой**

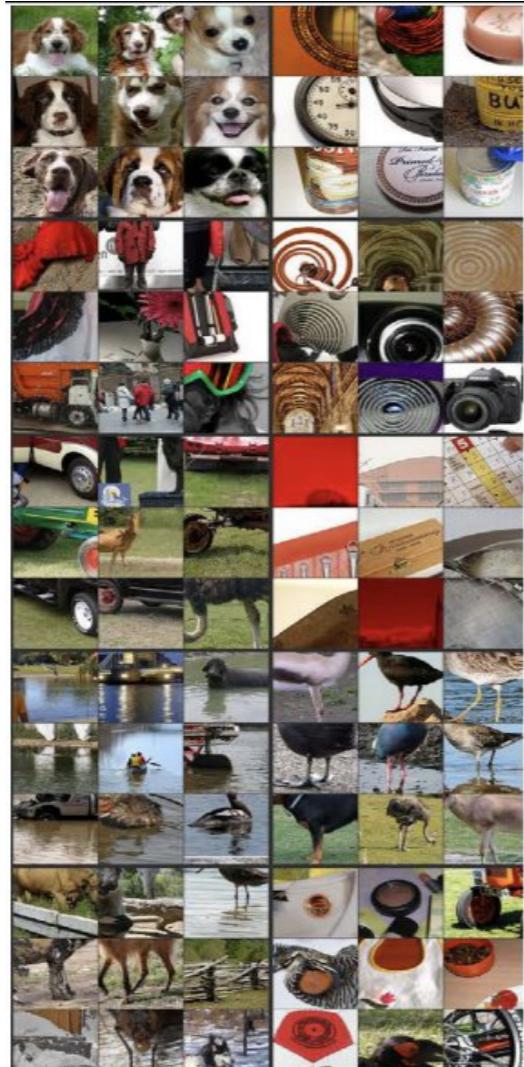




# Интерпретируемость

- Рассматриваем некоторый нейрон в свёртке.
- Для данного нейрона возьмем объекты, на которых значения выхода нейрона самое большое.
- Визуализируем receptive field этого нейрона у этих объектов, то есть посмотрим какая часть изображения видна данному нейрону.

**Четвертый  
свёрточный слой**



**Пятый  
свёрточный слой**





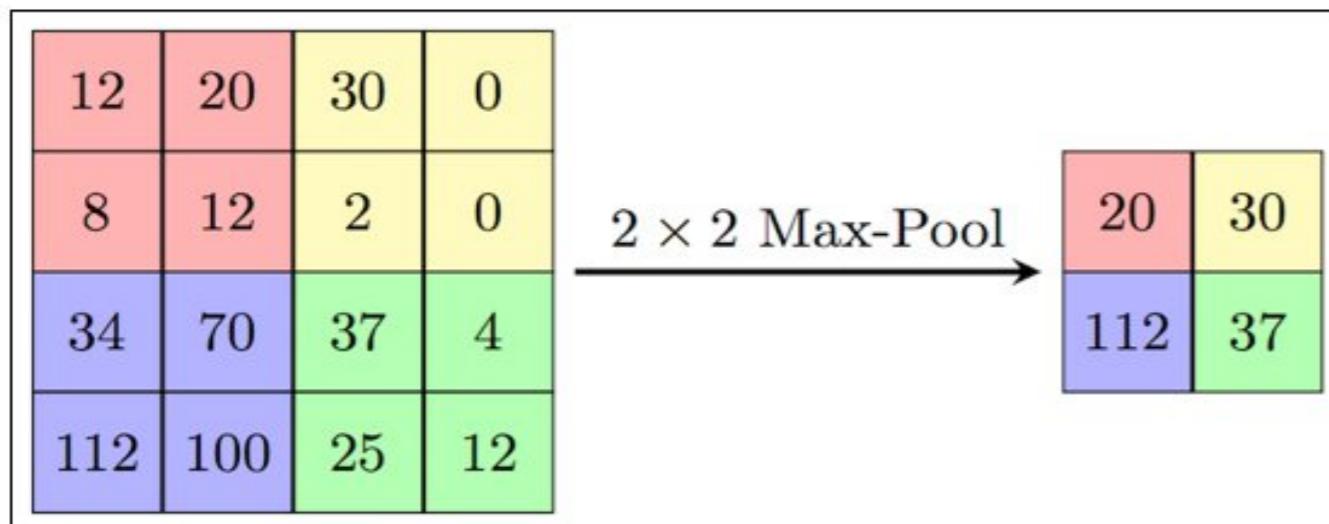
# 2D Pooling



# Pooling

Скользим окном по входным данным и вычисляем некоторую функцию от элементов в окне.

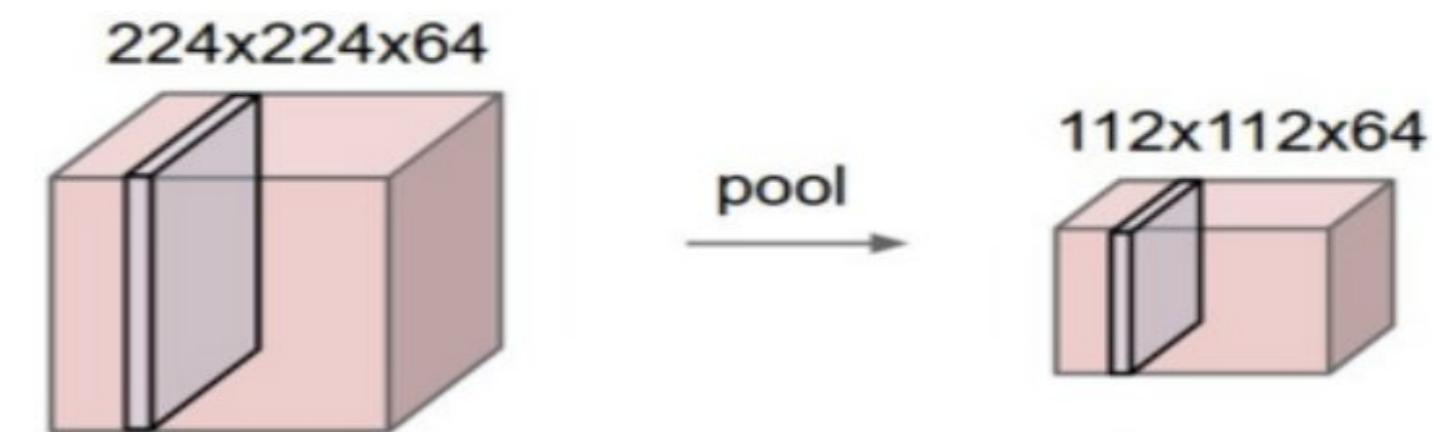
## Пример



Цель: уменьшение размерности.

Применяется обычно *после свёрточного слоя*.

После свёртки имеем большое количество признаков, для дальнейшей работы настолько подробные признаки уже не нужны. уменьшим количество признаков, оставляя важную информацию.

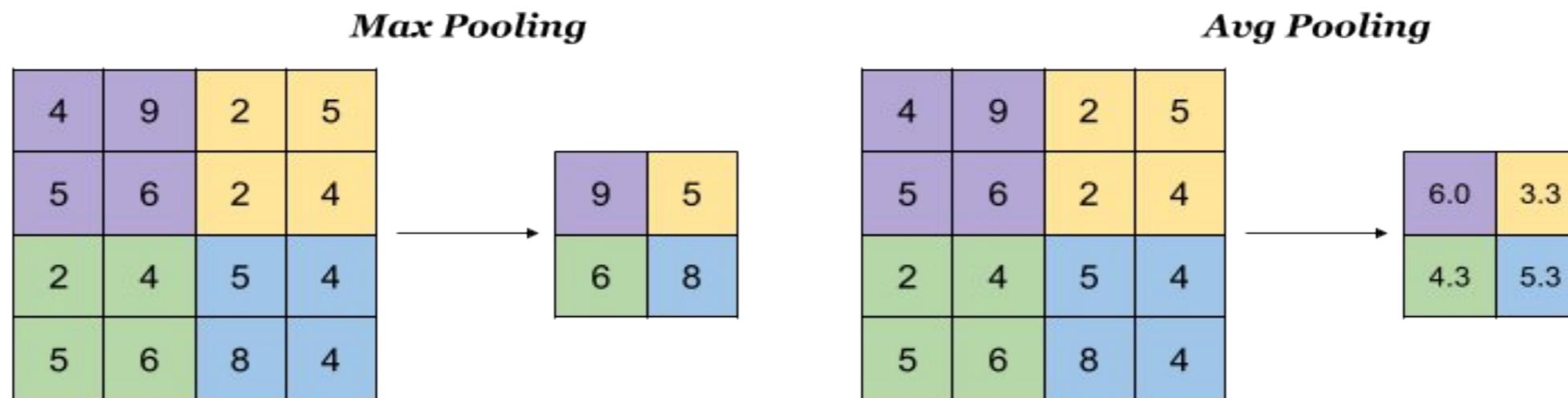




# Pooling

## Гиперпараметры:

- **Размер окна.** Обычно 2
- **Stride.** Шаг, с которым будем перемещать окно. Обычно 2.
- **Функция,** которую будем применять к элементам в окне.
  1. Max (самый популярный вариант)
  2. Average
  3. Sum



<https://indoml.com>



## Pooling. Цели

- Увеличиваем **receptive field** следующих слоев.
- Уменьшаем **изображение**, чтобы следующие свёрточные слои оперировали над большей областью исходного изображения.
- **Ускорение вычислений** за счет уменьшения количества признаков.  
Такая фильтрация помогает не переобучаться, т.к. выкидываем часть информации.
- **Помогаем модели понять был ли найден паттерн.**  
Значения в картах сверки отвечают за схожесть с паттерном.  
При взятии максимального значения в окне, оставляем максимальную схожесть, встреченную в этом окне. Нам не важны остальные значения, по максимуму можно понять был ли встречен паттерн в данном окне.  
Помогли сети, выкинув ненужную информацию.  
Применение максимума в данной ситуации более логично, именно поэтому это самый популярный вариант.



# Pooling. Backpropagation

Функция потерь не зависит от значений, не выбранных в качестве максимума.

Градиент по ним будет равен 0.  
Т.е. градиент потечет назад  
только через значения,  
выбранные в качестве максимумов.

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters  
and stride 2

Forward

6	8
3	4

6	8
3	4

Backpropagation

0	0	0	0
0	dout	0	dout
dout	0	0	0
0	0	0	dout



# Нормализация



# Батч-нормализация

## Достоинства

- Нейроны будут иметь один и тот же сдвиг и масштаб.
- Делая scale и shift перепараметризуем сеть:  
веса ( $\gamma, \beta$ ) отвечают за масштаб и сдвиг,  
веса слоя, выход которого мы нормализовали,  
отвечают за направление.



более однородные данные  
идут на вход след. слою



сетка быстрее учится

## Недостатки

- Если использовать маленькие батчи,  
то шум становится слишком большим,  
нормализация работает плохо.

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;  
Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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



## Послойная (Layer) нормализация

Нормализация происходит по признакам для каждого элемента батча.  
Если работаем с картинками, то усредняем по каналам изображения.

$$\begin{aligned}\mu_i &= \frac{1}{m} \sum_{j=1}^m x_{ij} \\ \sigma_i^2 &= \frac{1}{m} \sum_{j=1}^m (x_{ij} - \mu_i)^2 \\ \hat{x}_{ij} &= \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}\end{aligned}$$

Лучше работает для рекуррентных сетей

## Объектная (Instance / contrast) нормализация

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_{ti})^2.$$

Применяется для тензоров картинок.

Нормализация происходит для каждого элемента батча для каждого канала.

Усредняем по высоте и ширине тензора картинки.

Делает сеть инвариантной к изображениям разной контрастности.



# Нормализация

- Бывает еще несколько видов нормализации:  
Weight Normalization,  
Group-Normalization,  
Batch-Instance Normalization...
- Можно комбинировать нормализации на разных слоях сети.
- Эмпирически было установлено,  
что для задач классификации изображений и детекции объектов,  
на первых слоях хорошо работает instance-нормализация,  
в средине — batch-нормализация,  
в конце — layer-нормализация.

[Статья](#)



# Развитие сверточных нейронных сетей



# Развитие сверточных нейронных сетей

**ImageNet** – база данных из 14 млн изображений, размеченных на 20 тысяч классов.

**ILSVRC** (The ImageNet Large Scale Visual Recognition Challenge) –

конкурс, использующий 1 млн. изображений с 1000 классов базы ImageNet.





# Convolution neural networks

AlexNet

VGG

Inception

ResNet

Inception-ResNet

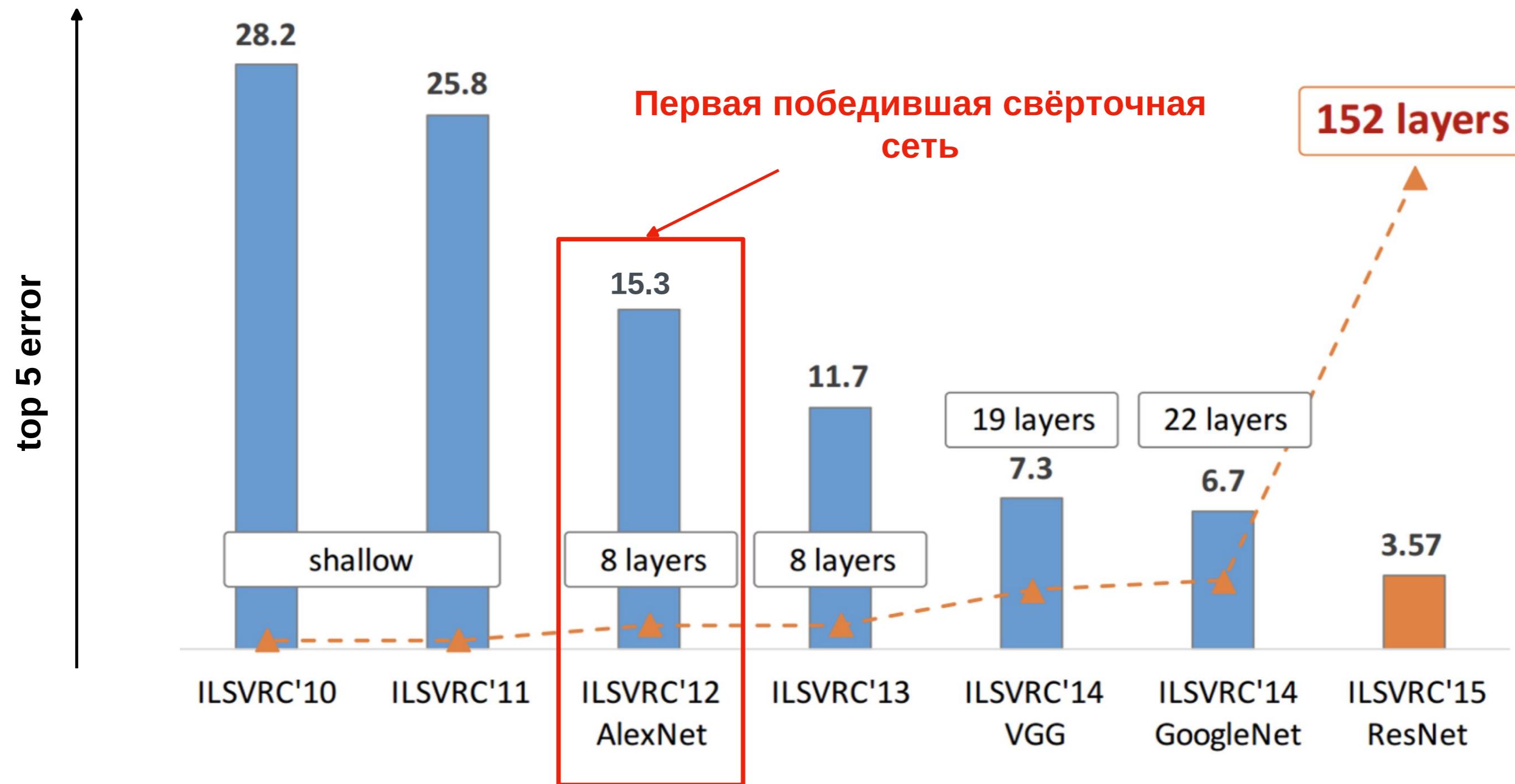
EfficientNet



# AlexNet

Победила в соревновании LSVRC-2012 с ошибкой 15.3%,  
против ошибки 26.2% на втором месте.

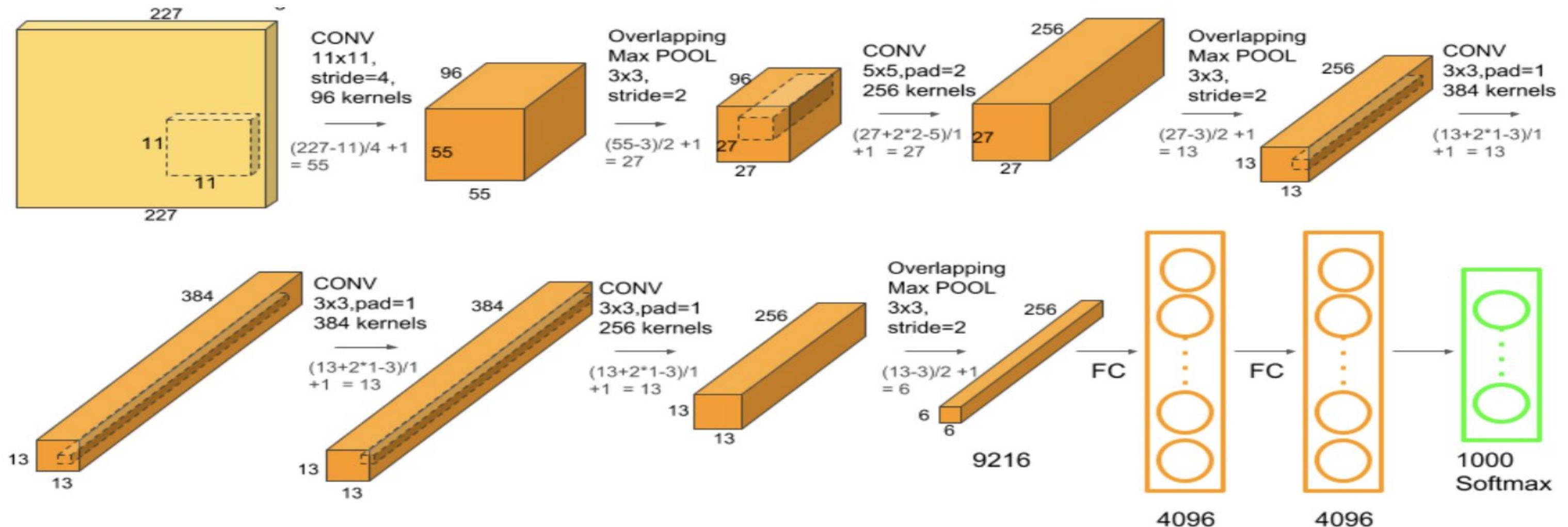
[Статья](#)





# AlexNet

- Решается задача классификации на 1000 классов.
- Архитектура состоит из пяти свёрточных слоев и трех полно связных.
- За первыми двумя свёрточными слоями следует Max Pooling с пересекающимися окнами.
- Свёрточные слои номер 3, 4, 5 связаны напрямую (без Max Pooling).
- После FC-слоев применяется Dropout с  $p = 0.5$ .
- Специальная нормализация — Local Response Normalization





# AlexNet

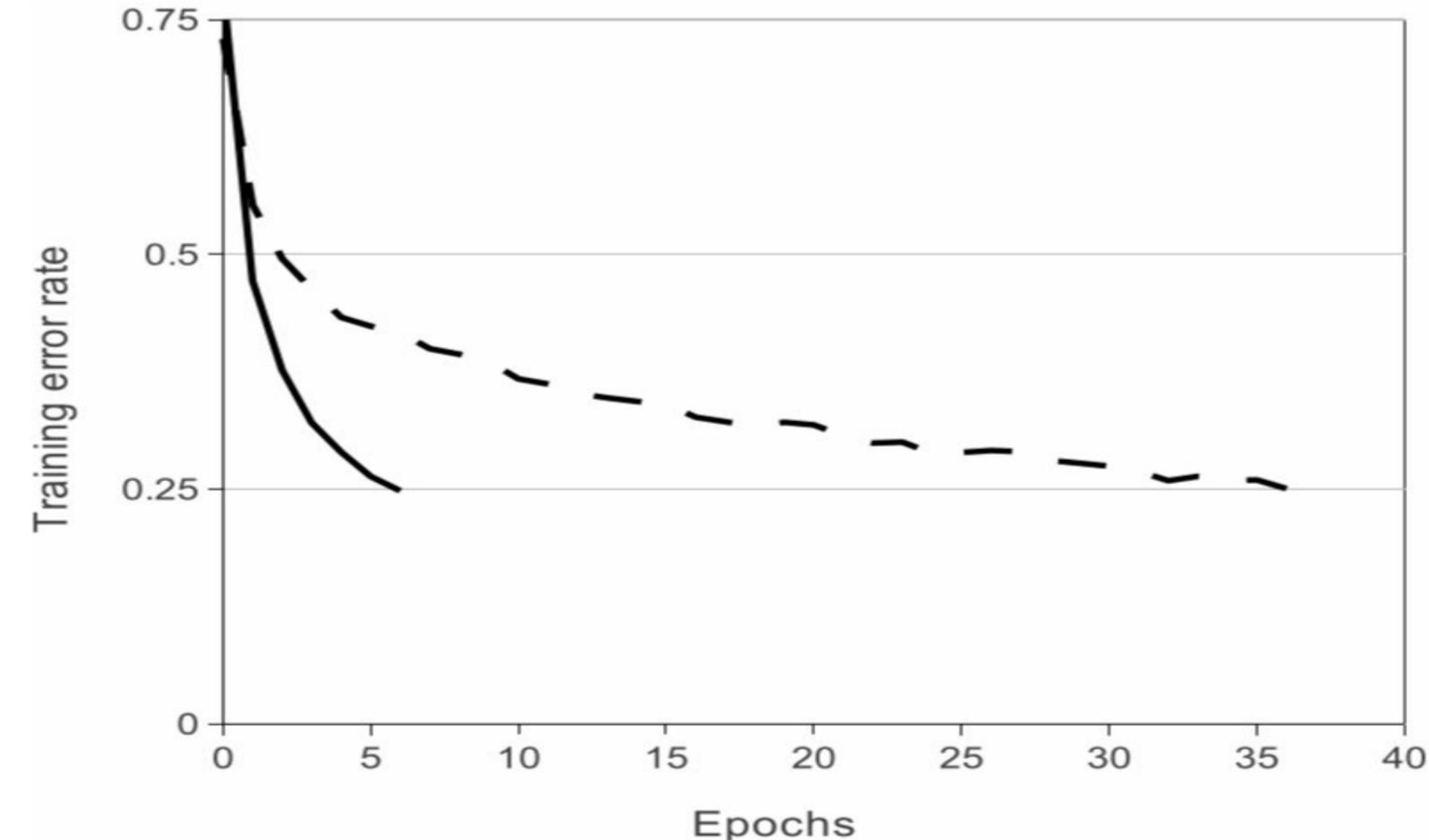
- В качестве функции активации используется **ReLU**.

В то время в основном использовались  $\tanh$  и  $\text{sigmoid}$ .

В сравнении с  $\tanh$  сеть с ReLU достигла ошибки 0.25% в **6 раз быстрее**.

Это связано с горизонтальными асимптотами у  $\tanh$  и  $\text{sigmoid}$ ,

что замедляет сходимость градиентного спуска.





# AlexNet

AlexNet содержит 62.3 миллионов параметров.

Conv: 3.7million (6%), FC: 58.6 million (94%)

Параметров много нужно бороться с переобучением.

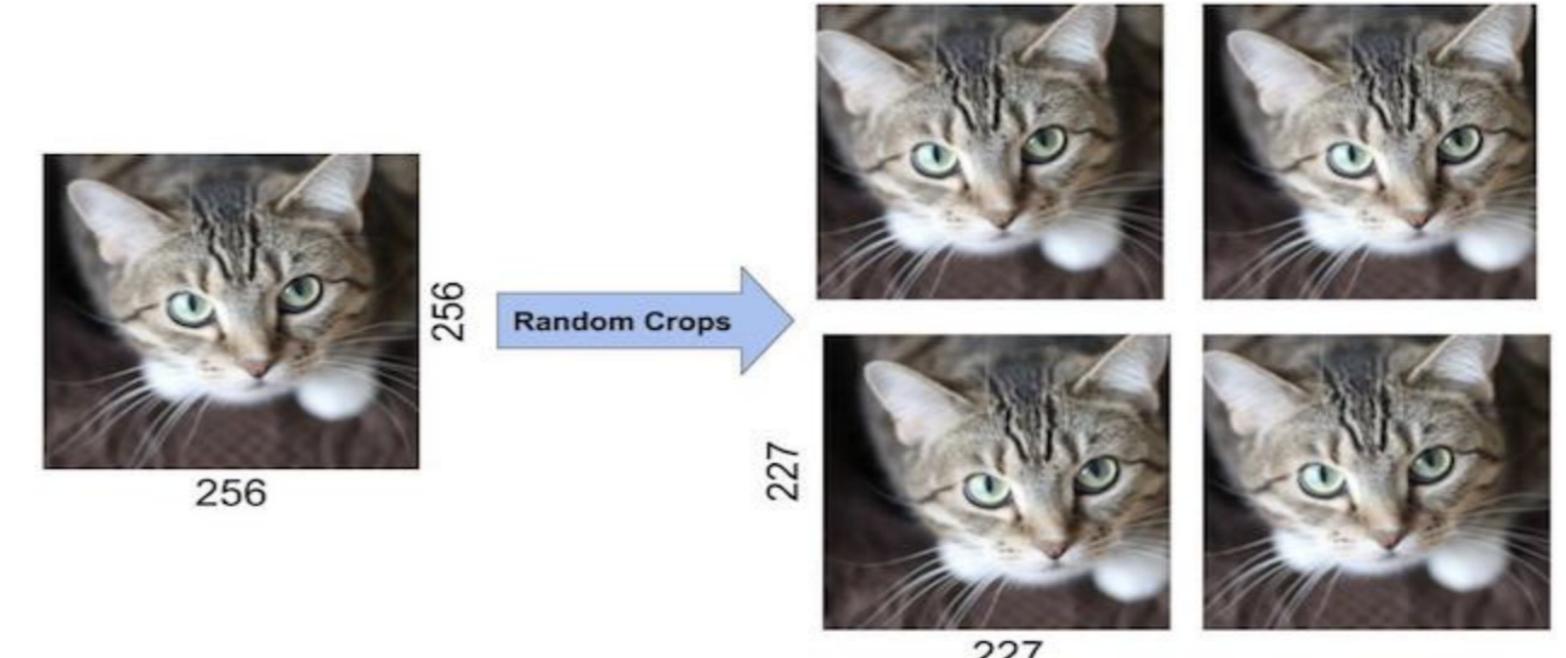
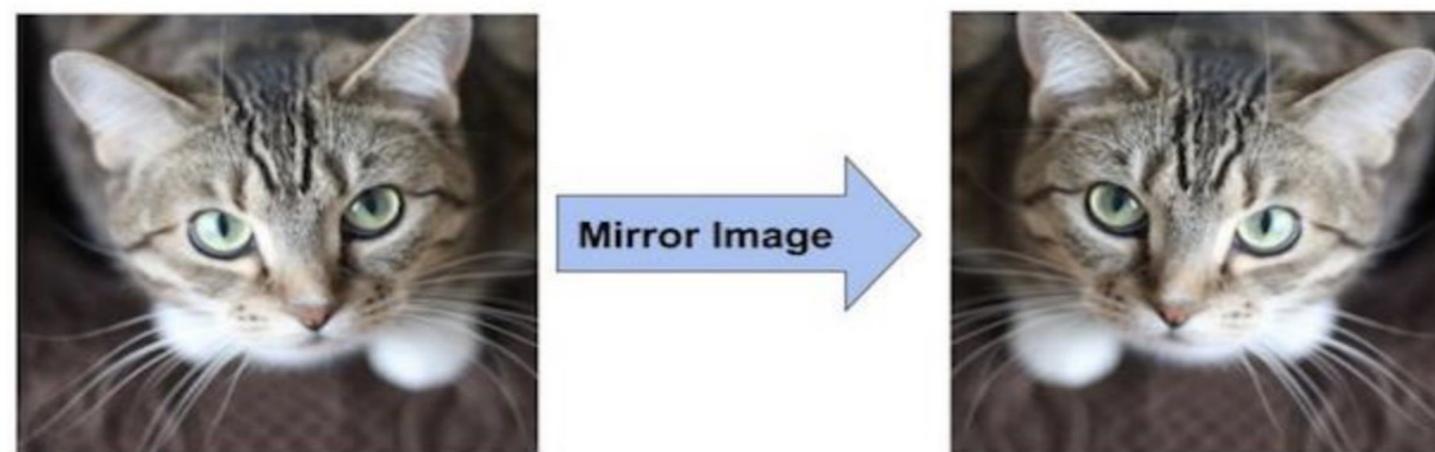
- Dropout

Совсем новый для того времени метод.

- Data Augmentation (Аугментация данных)
  1. Добавили вертикальное отображение картинок к датасету.
  2. Вырезали случайные подкартинки большого размера из исходного изображения.

Это позволяет модели понять, что класс объекта инвариантен

к отражению и к положению объекта на изображении.



256

227

227



# Аугментации данных

Аугментация данных — техника искусственного **увеличения выборки**.  
Помогает предотвратить переобучение нейросетей.

*Какие бывают типы аугментации изображений?*



# Аугментации данных

Аугментация данных — техника искусственного **увеличения выборки**.  
Помогает предотвратить переобучение нейросетей.

Какие бывают типы аугментации изображений?

Geometry based



Color based





# Convolution neural networks

AlexNet

VGG

Inception

ResNet

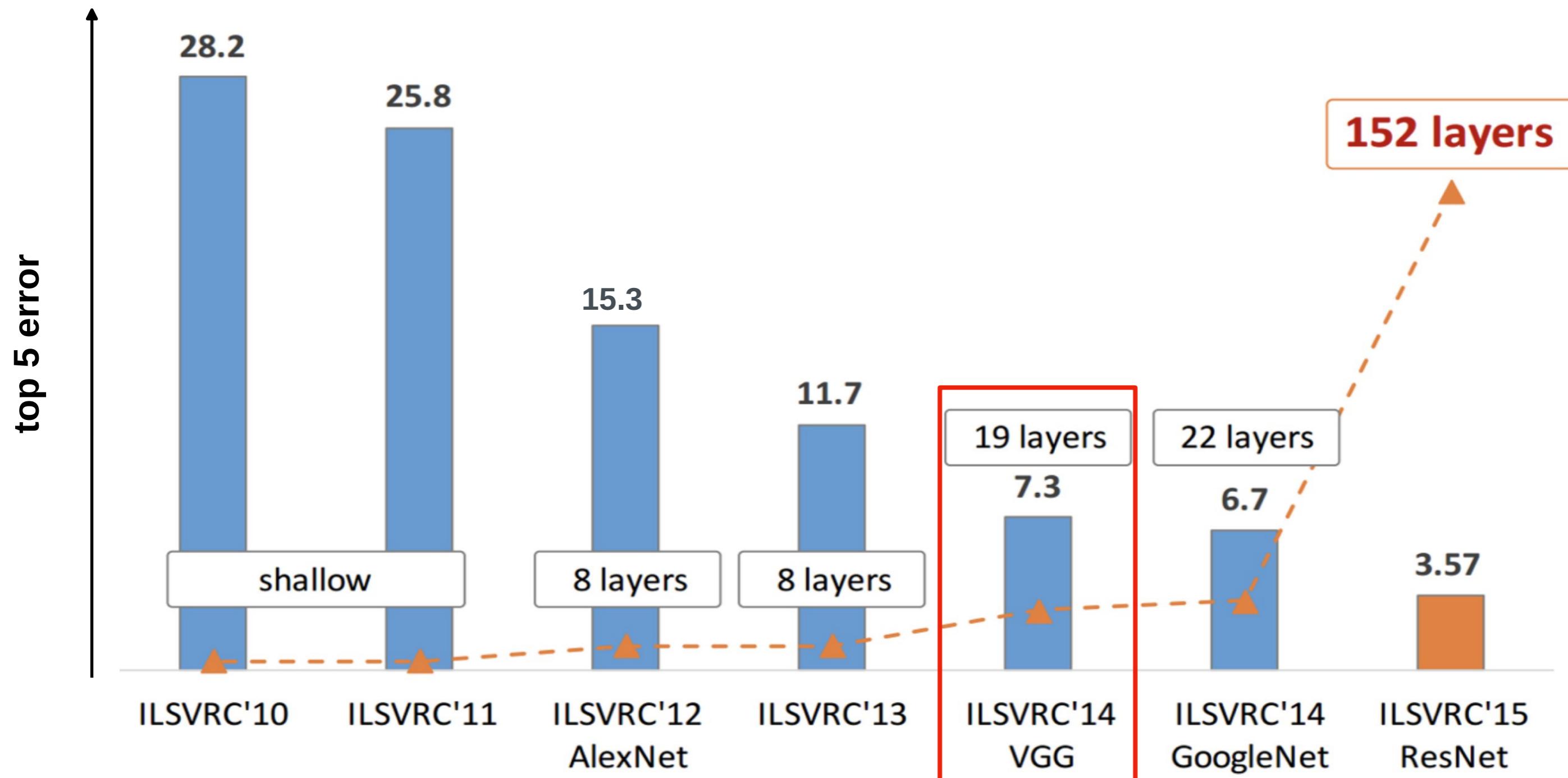
Inception-ResNet

EfficientNet



# VGG

[Статья](#)





# VGG

- Сеть становится **глубже**.
- Свёртки с широкими ядрами заменяются на **свертки с ядрами 3x3**.

Широкие свёртки у AlexNet нужны для увеличения *области видимости*.

*Чем полезно использовать несколько маленьких свёрток?*

1. При увеличении числа слоев в сети **receptive field тоже увеличивается**

Например, 3 свёртки подряд с фильтрами 3x3 с stride=1 имеют такой же receptive field, что и одна свёртка с фильтром 7x7.

## 2. Уменьшается число параметров

Пусть кол-во каналов до и после каждой свёртки - n .

Кол-во параметров у трёх свёрток 3x3 :

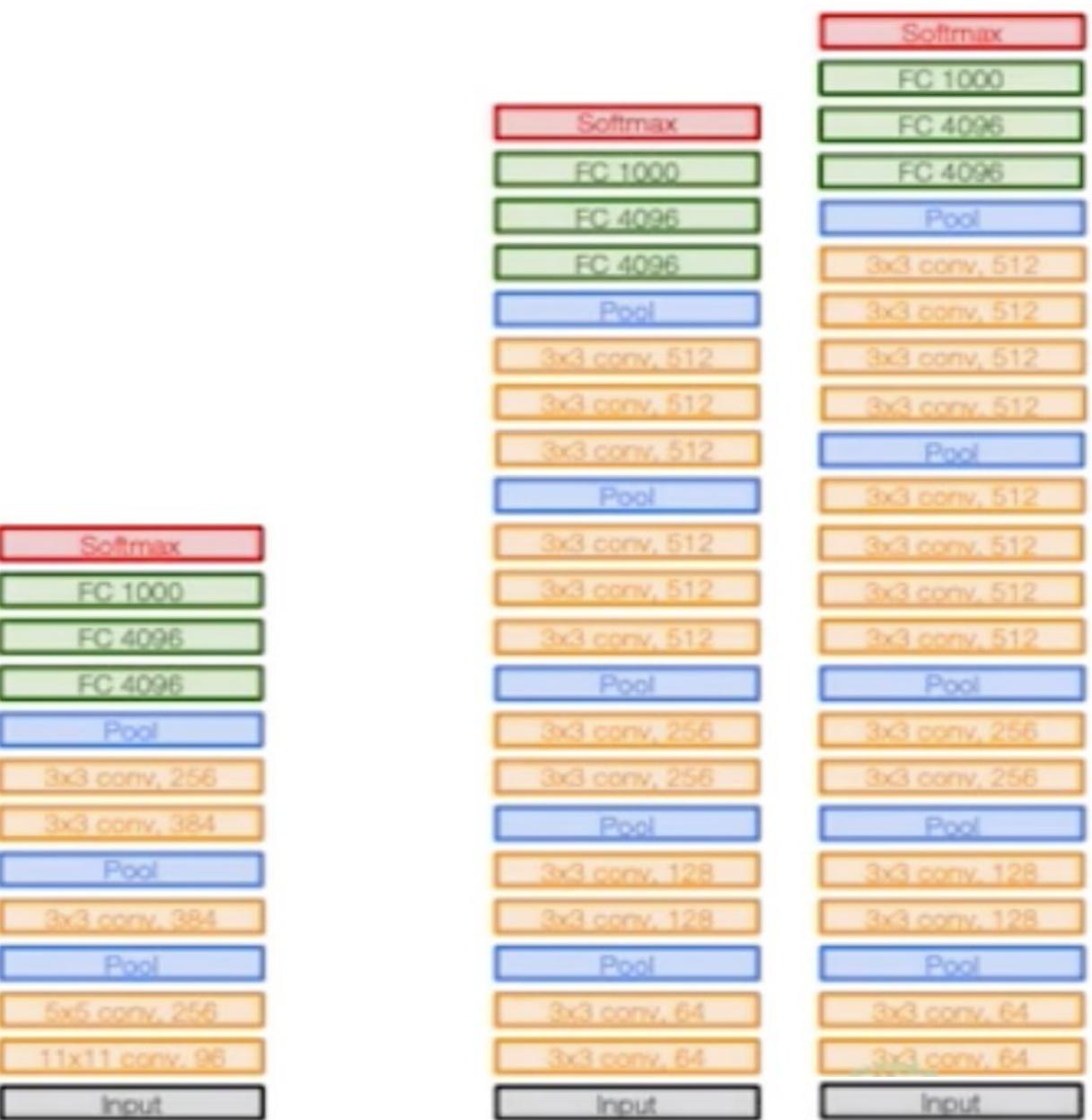
$$3 * 3 * 3 * n = 27n$$

Кол-во параметров у одной свёртки 7x7:

$$1 * 7 * 7 * n = 49n$$

## 3. Больше нелинейности

Между тремя слоями свёртки есть 3 нелинейности, что позволяет сети представлять более сложные функции.



AlexNet

VGG-16

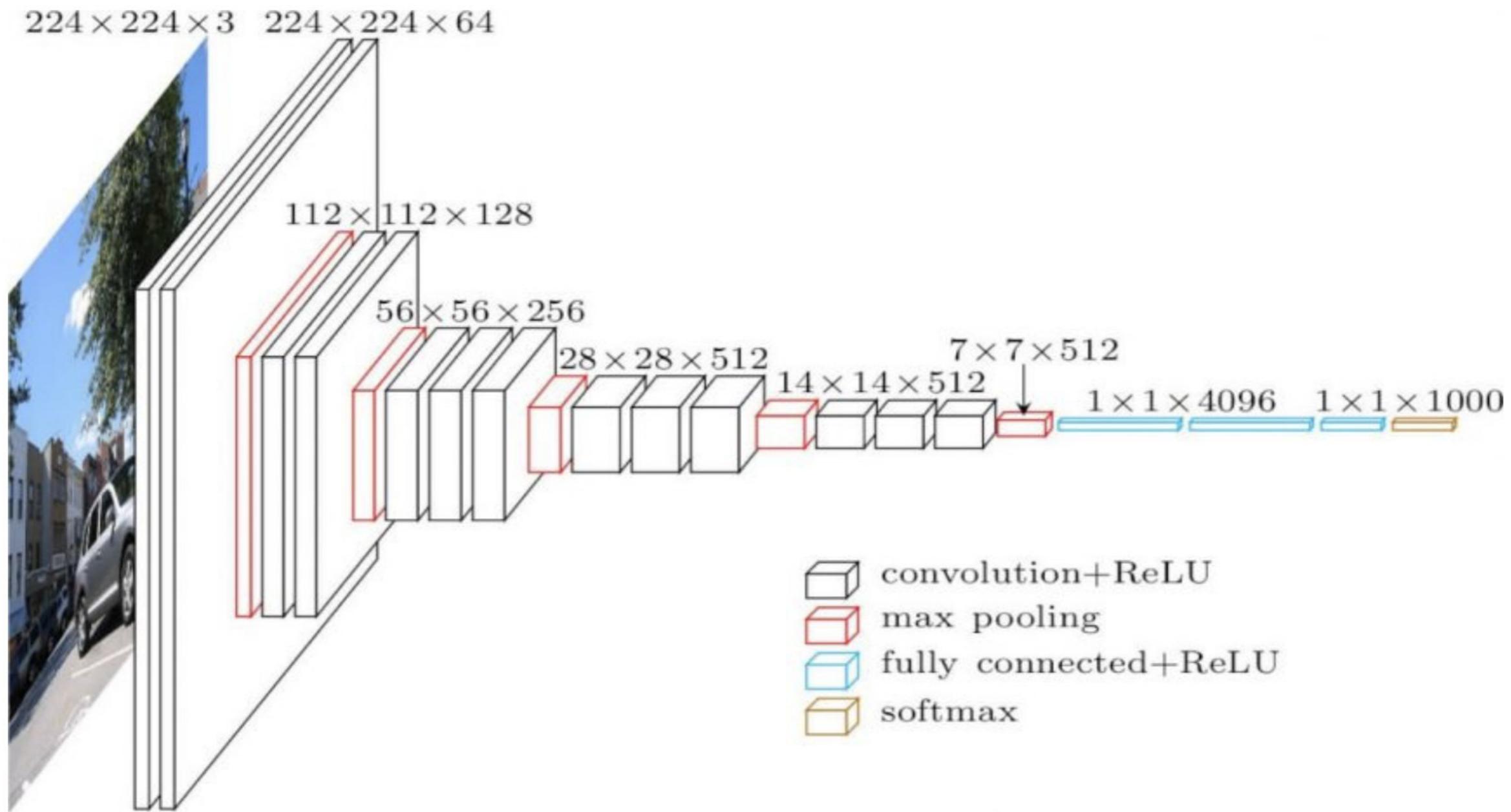
VGG-19



# VGG

## Детали:

- Функция активации ReLU.
- Аугментация данных.
- Нет нормализации, как у AlexNet.





# VGG

## Сравнение моделей AlexNet и VGG:

### AlexNet:

- Ошибка 15.3%
- 8 слоев
- 62М
- Использование памяти во время обучения:  
8 МВ на картинку

### VGG:

- Ошибка 7.3%
- 16/19 слоев
- 138М / 143М
- Использование памяти во время обучения:  
192 / 202 МВ на картинку



# VGG

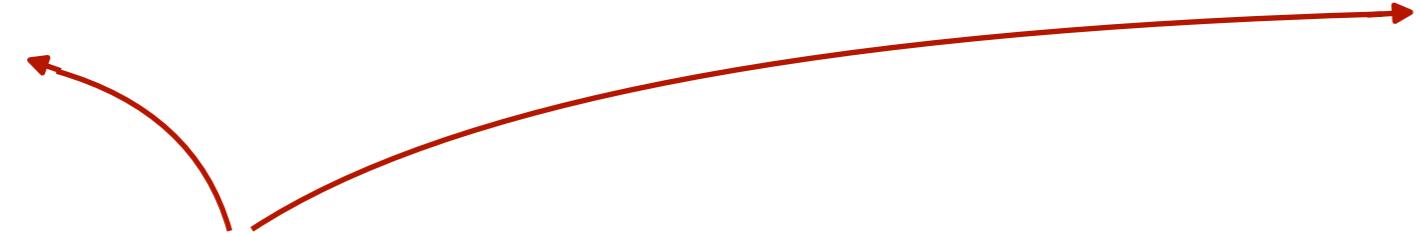
## Сравнение моделей AlexNet и VGG:

### AlexNet:

- Ошибка 16.4%
- 8 слоев
- 62M
- Использование памяти во время обучения:  
8 MB на картинку

### VGG:

- Ошибка 7.3%
- 16/19 слоев
- 138M / 143M
- Использование памяти во время обучения:  
192 / 202 MB на картинку



Как это считается?



# VGG

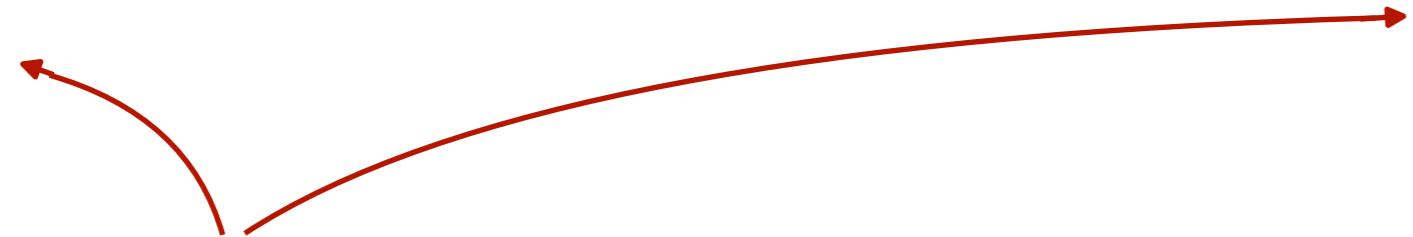
## Сравнение моделей AlexNet и VGG:

### AlexNet:

- Ошибка 16.4%
- 8 слоев
- 62М
- Использование памяти во время обучения:  
8 МВ на картинку

### VGG:

- Ошибка 7.3%
- 16/19 слоев
- 138М / 143М
- Использование памяти во время обучения:  
192 / 202 МВ на картинку



### Как это считается?

Вспомним, что при обучении нам необходимо сохранить

- все промежуточные выходы сети во время прямого прохода,
- все градиенты выходов по входу во время обратного прохода.

Поэтому для подсчета использованной памяти необходимо сложить занимаемые памяти каждого из хранимых тензоров.



# Convolution neural networks

AlexNet

VGG

Inception

ResNet

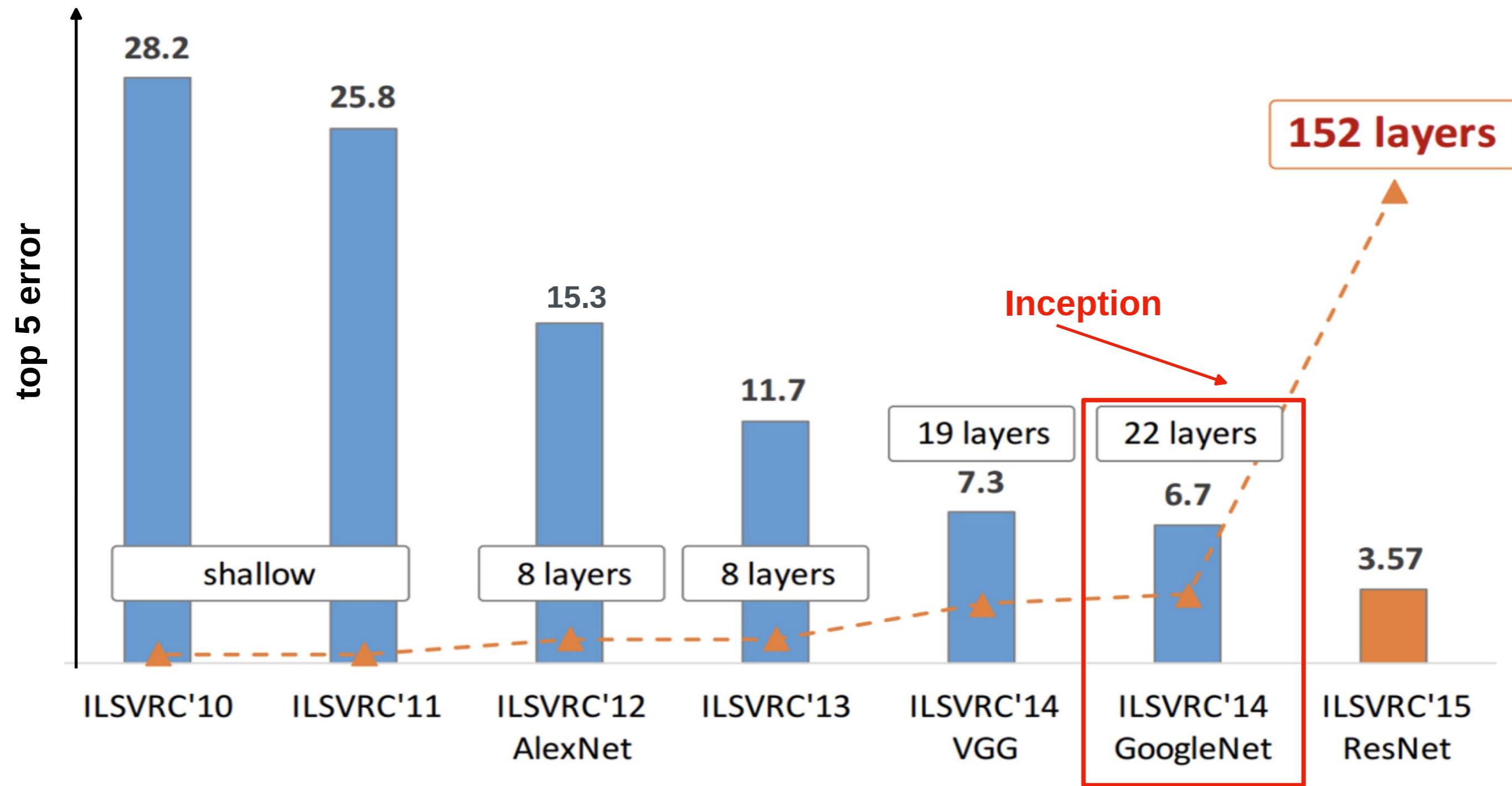
Inception-ResNet

EfficientNet



# Inception

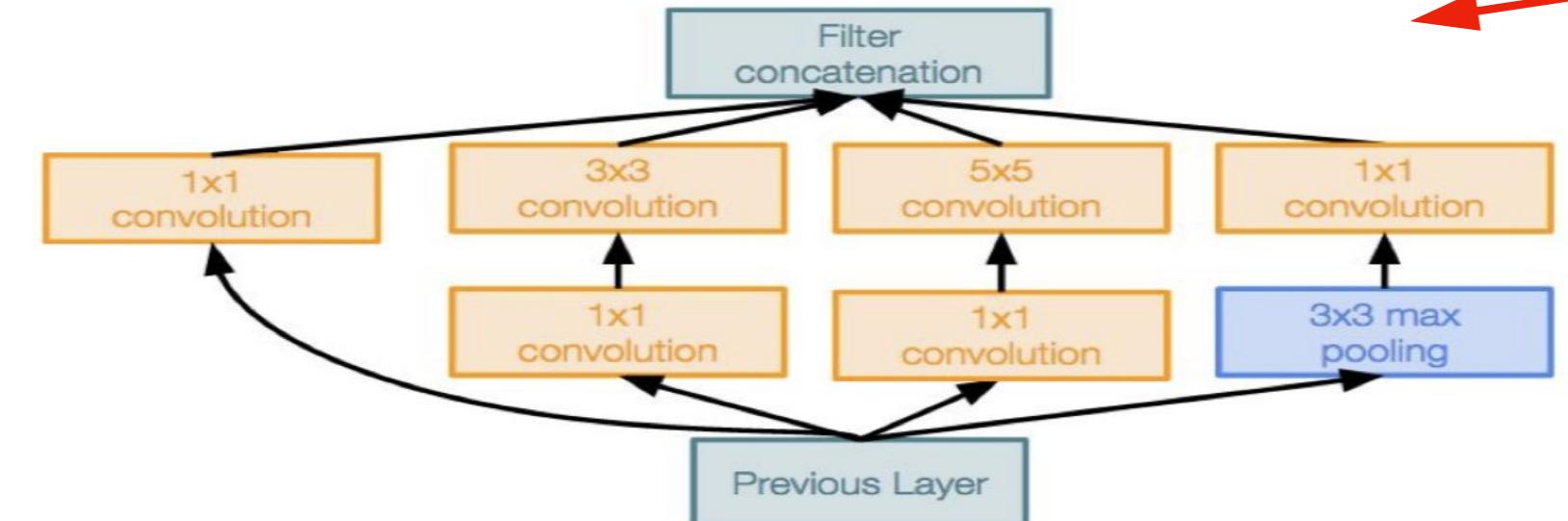
[Статья](#)



# Inception

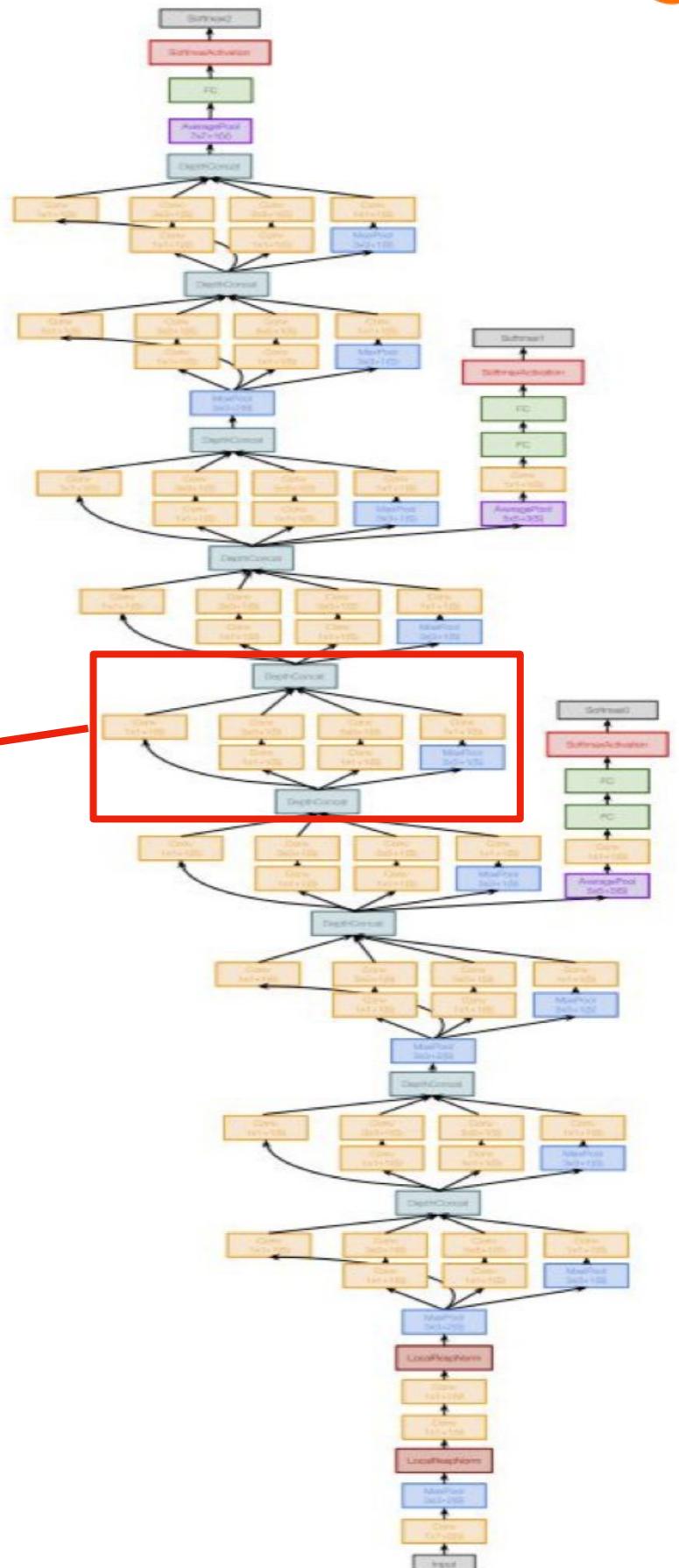


- Достигает ошибки 6.7%
  - 22 слоя
  - Содержит только 5M параметров  
В 12 раз меньше, чем AlexNet
  - Архитектура построена из одинаковых модулей  
(между которыми иногда добавляется pooling)



# Inception module

Применяем один и тот же Inception module много раз.





# Inception

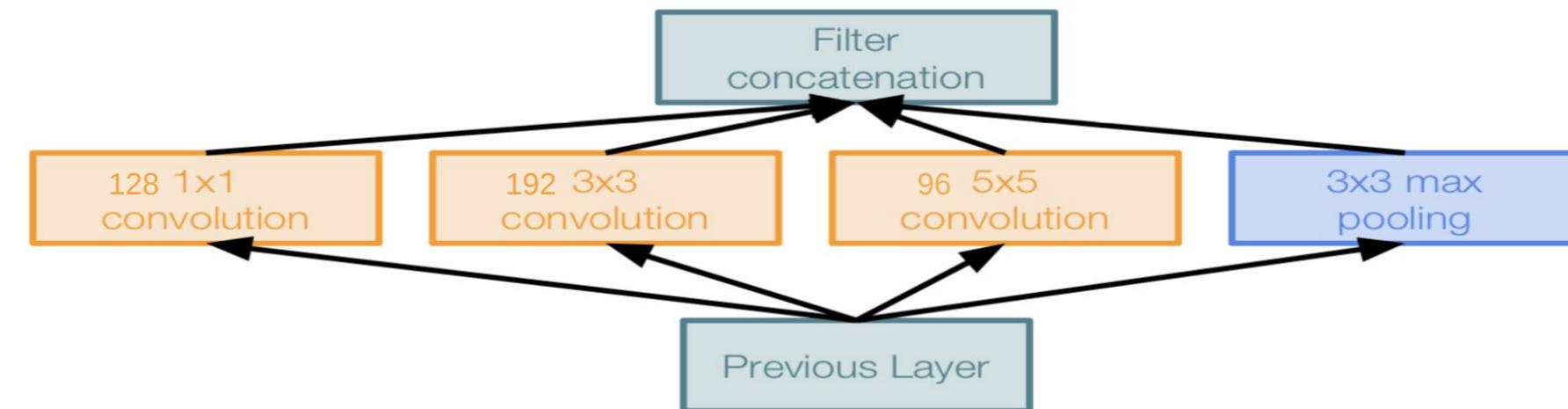
## Что такое Inception module?

- Применяем несколько свёрток с разными размерами к входу параллельно.

Посмотрим как на небольшие паттерны, так и на паттерны побольше.

- Применяем Max pooling к входу параллельно со свёртками.
- Сконкатенируем полученные результаты.

Конкатенируем по размерности каналов.



Naive Inception module

## Какие возникнут проблемы?

- Ширина и высота выходных карт не соответствует друг другу.

Можно решить добавлением padding.

- После конкатенации кол-во каналов станет большим.

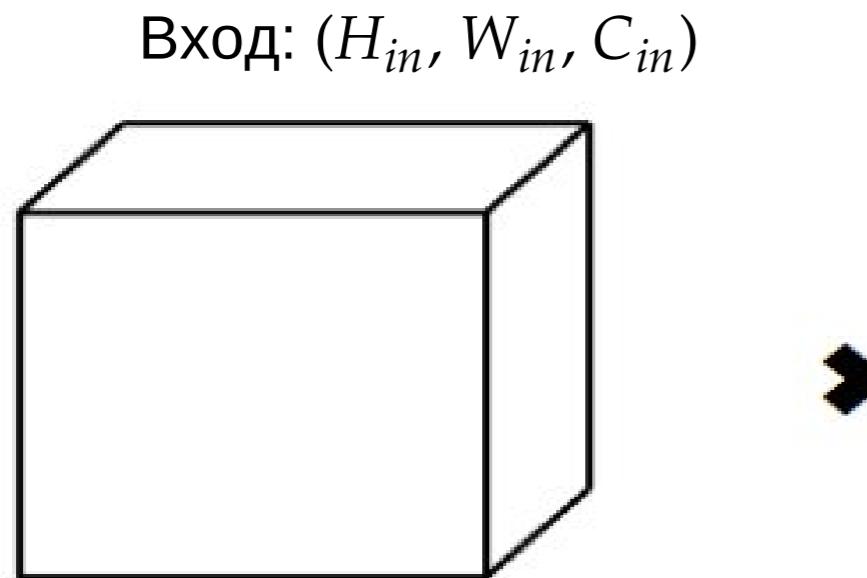
Решение - свёртка 1x1.



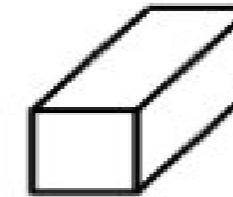
# Inception

**Что такое свёртка 1x1?**

- Фильтр имеет размер  $(1, 1, C_{in})$ .

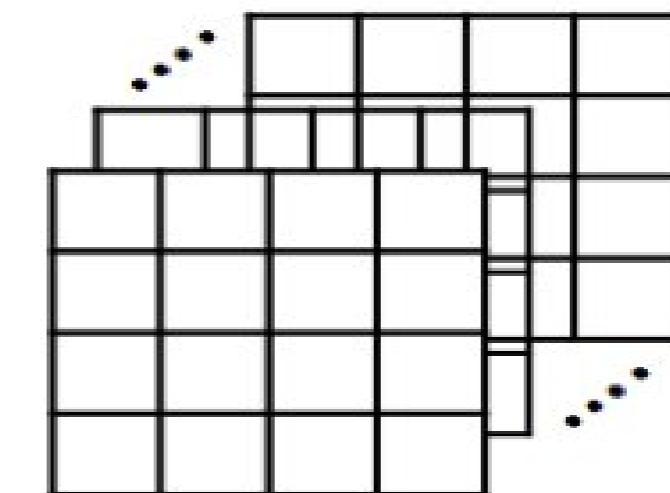


Фильтр:  $(1, 1, C_{in})$



$C_{out}$  фильтров

=



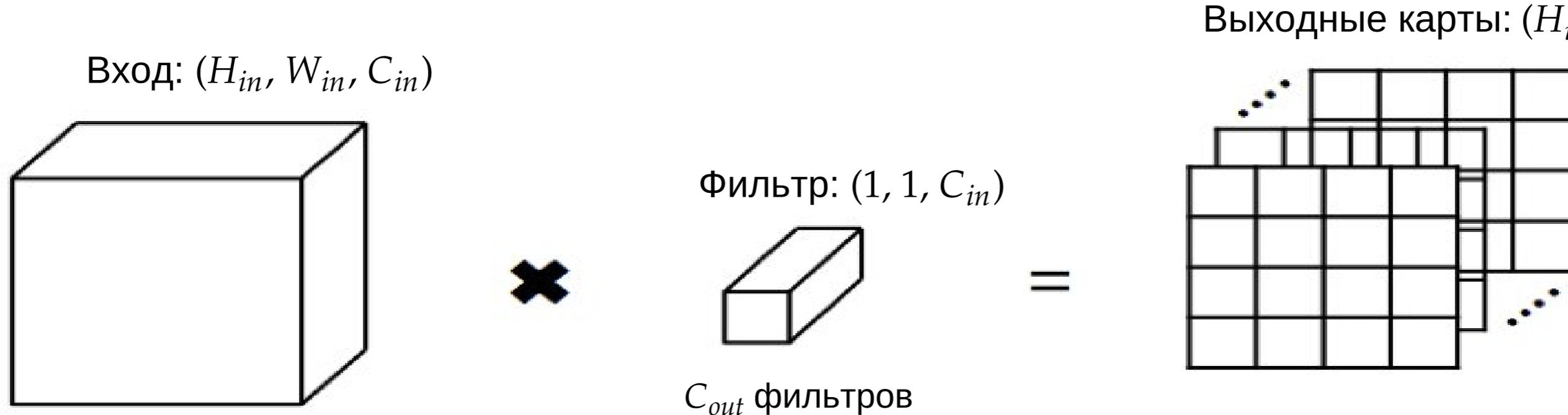
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

- Фильтр имеет размер  $(1, 1, C_{in})$ .
- Фильтр умножается на часть входной картинки размера  $(1, 1, C_{in})$  и получается одно число.



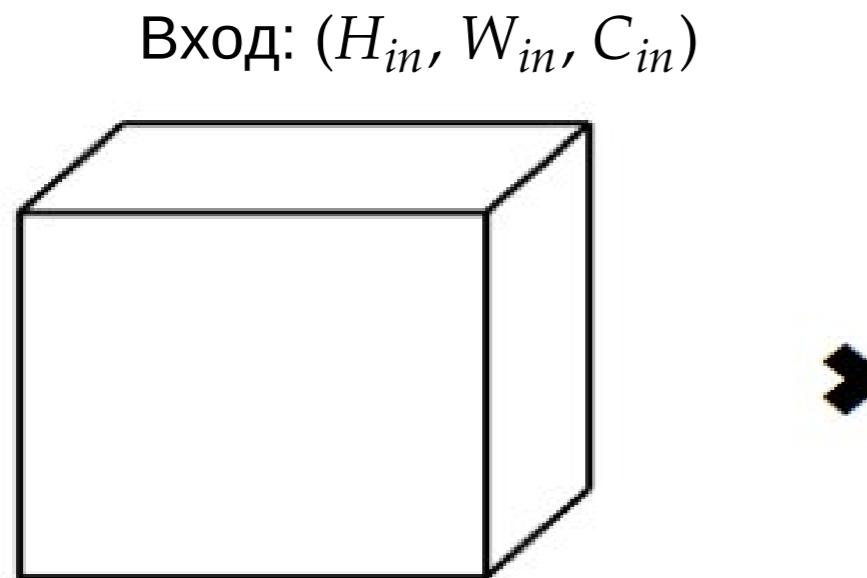
\*Для простоты на картинке уплачено смещение, которое должно быть.



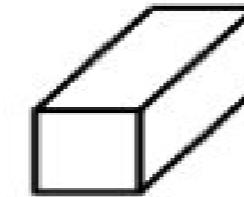
# Inception

## Что такое свёртка $1 \times 1$ ?

- Фильтр имеет размер  $(1, 1, C_{in})$ .
- Фильтр умножается на часть входной картинки размера  $(1, 1, C_{in})$  и получается одно число.
- После применения  $C_{out}$  фильтров к данной части картинки получим вектор размера  $(1, 1, C_{out})$ .

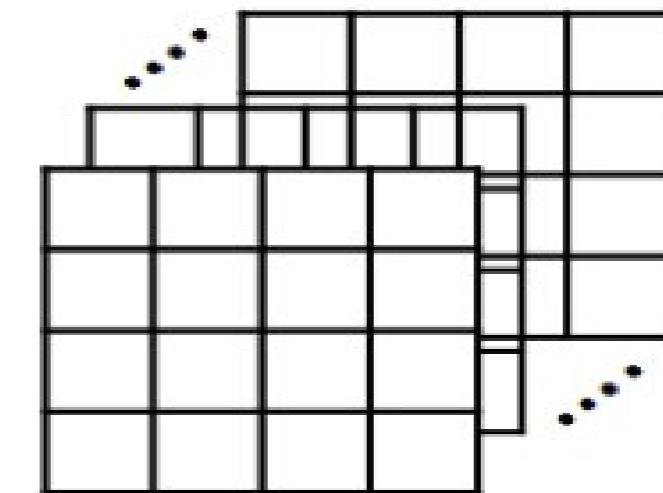


Фильтр:  $(1, 1, C_{in})$



$C_{out}$  фильтров

=



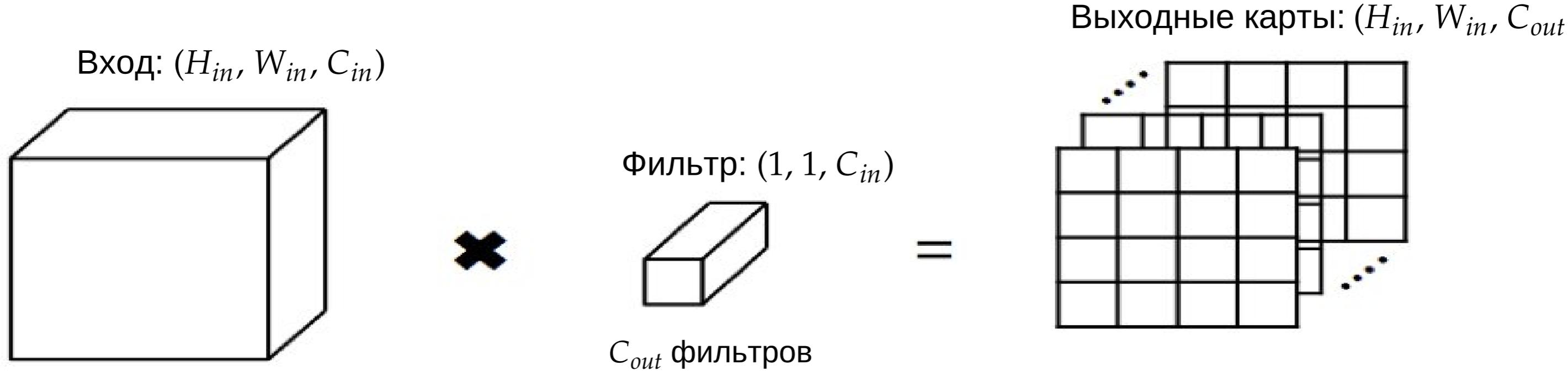
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка 1x1?

- Фильтр имеет размер  $(1, 1, C_{in})$ .
- Фильтр умножается на часть входной картинки размера  $(1, 1, C_{in})$  и получается одно число.
- После применения  $C_{out}$  фильтров к данной части картинки получим вектор размера  $(1, 1, C_{out})$ .
- Применив фильтры ко всем частям получим выход размера  $(H_{in}, W_{in}, C_{out})$ .



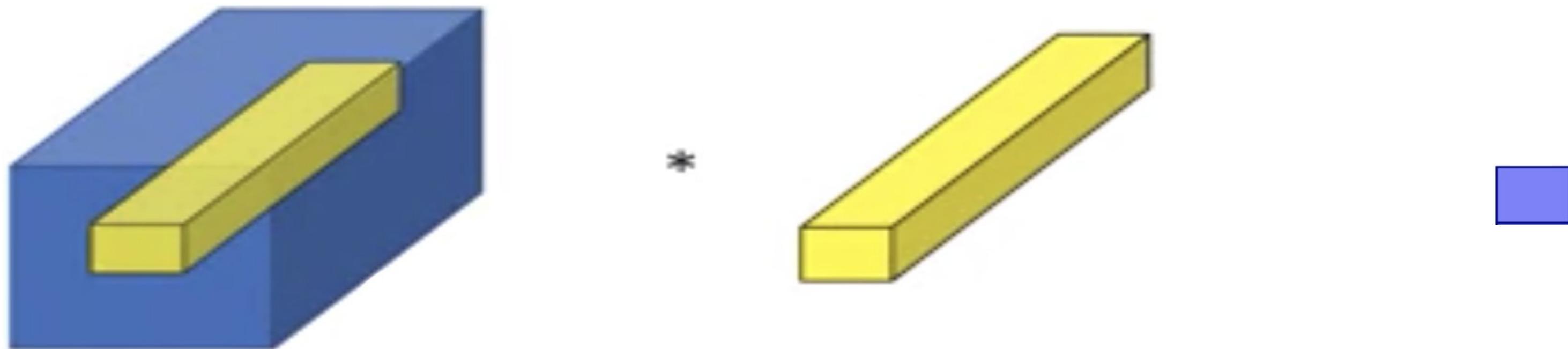
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



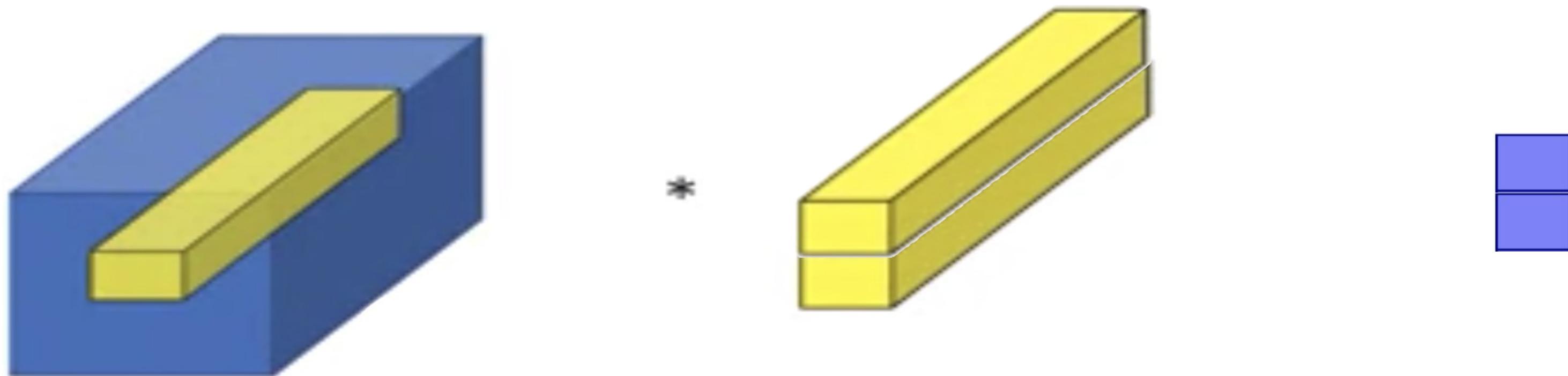
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



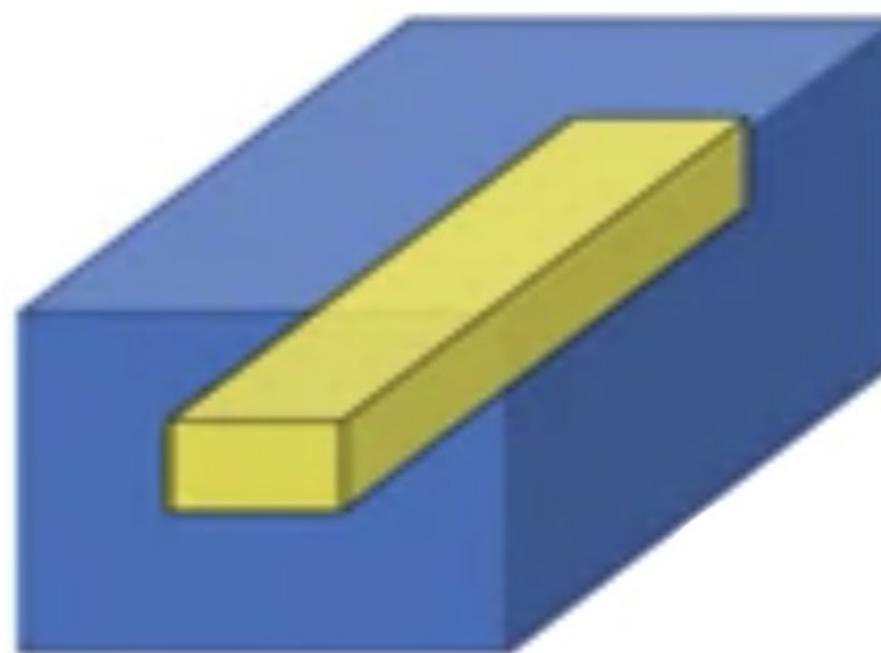
\*Для простоты на картинке упущено смещение, которое должно быть.



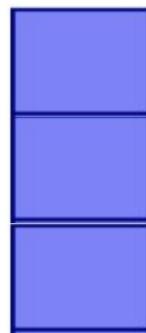
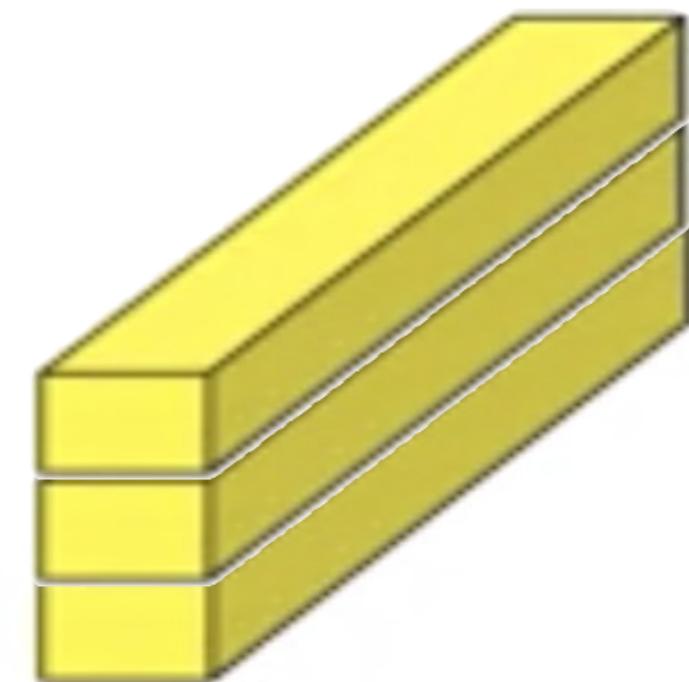
# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



\*



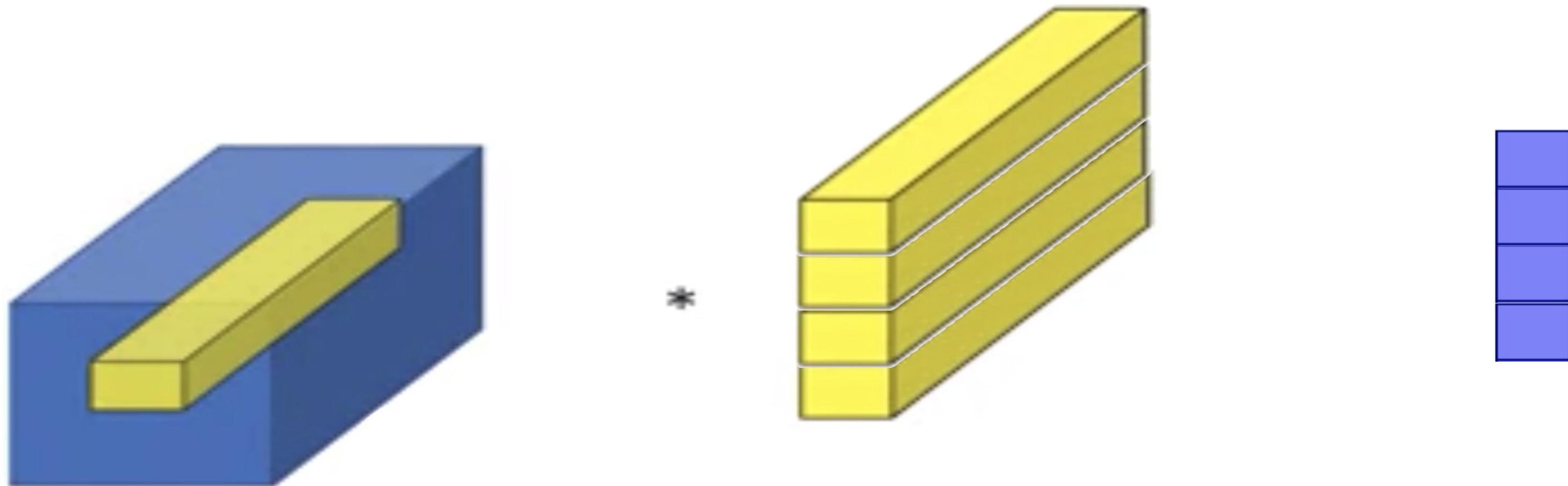
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



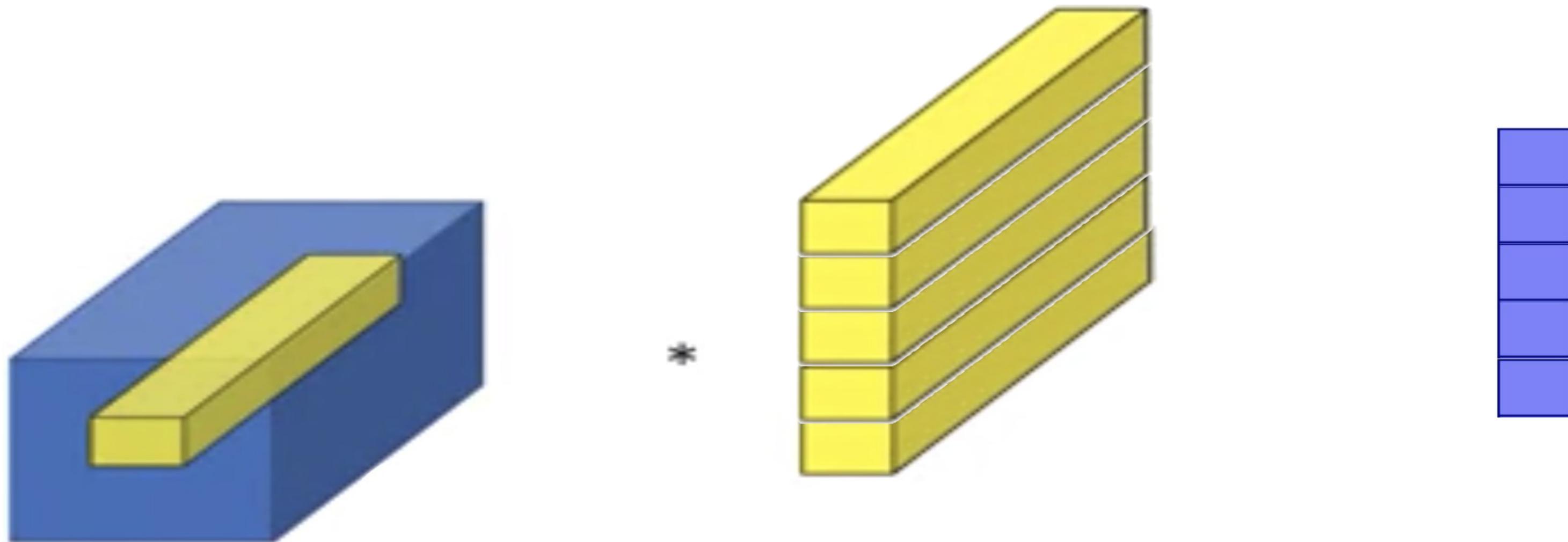
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



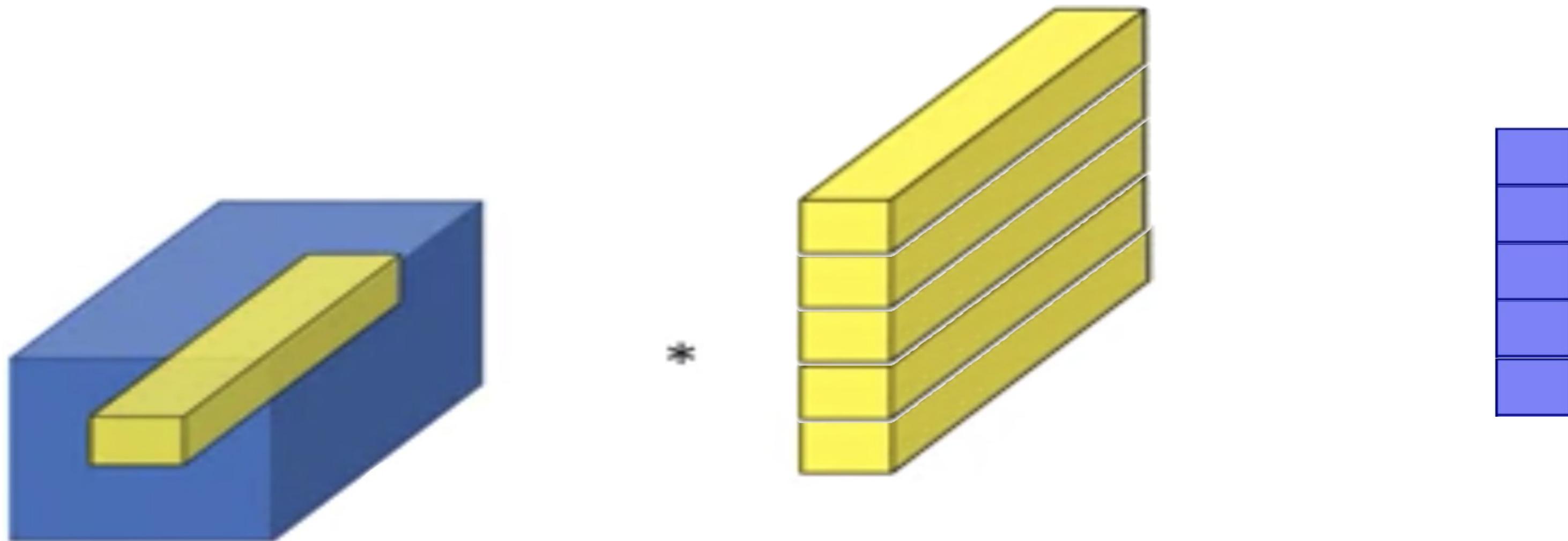
\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

## Что такое свёртка $1 \times 1$ ?

Применение фильтров к одной части изображения размера эквивалентно применению полносвязного слоя к этой части.



Заметим, что это то же самое, что умножение желтой матрицы на вектор из входной картинки.

\*Для простоты на картинке упущено смещение, которое должно быть.



# Inception

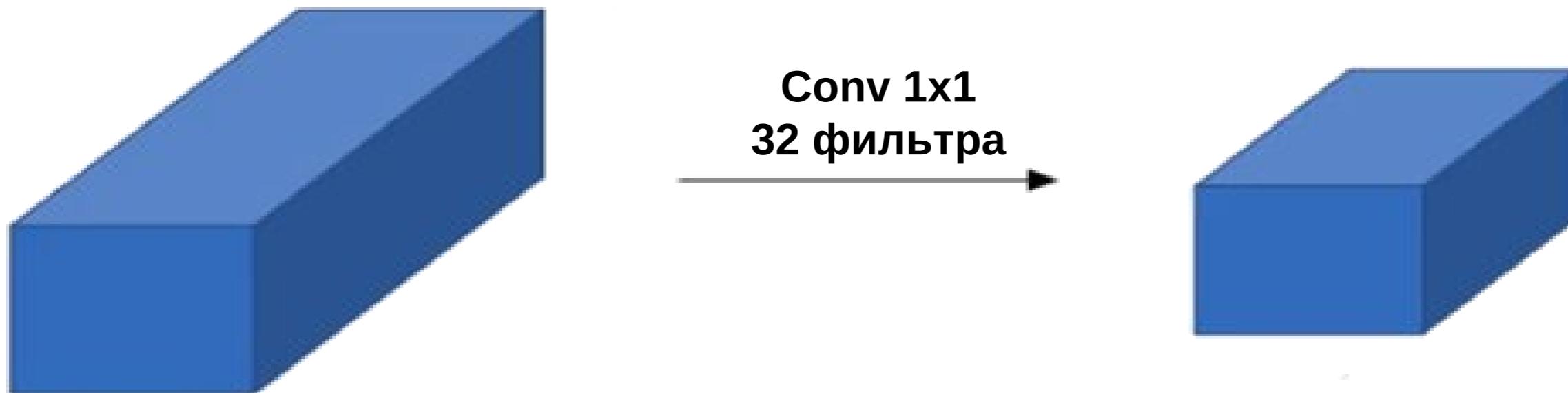
**Зачем нужна свёртка 1x1?**



# Inception

## Зачем нужна свёртка $1 \times 1$ ?

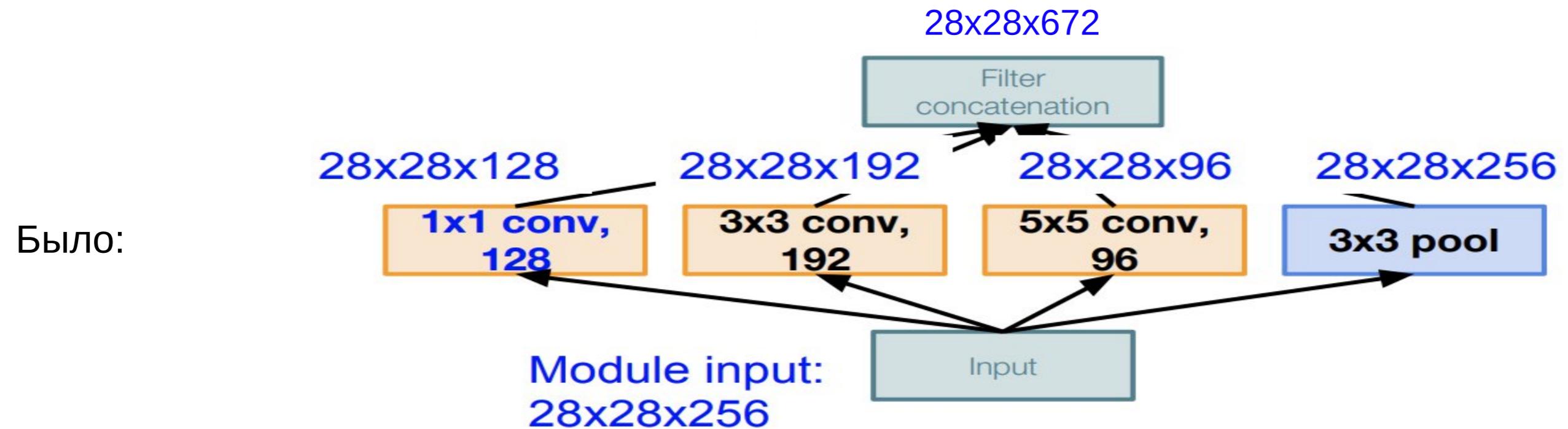
С помощью  $1 \times 1$  сверток можно сильно уменьшить количество каналов у изображения, не меняя при этом receptive field.





# Inception

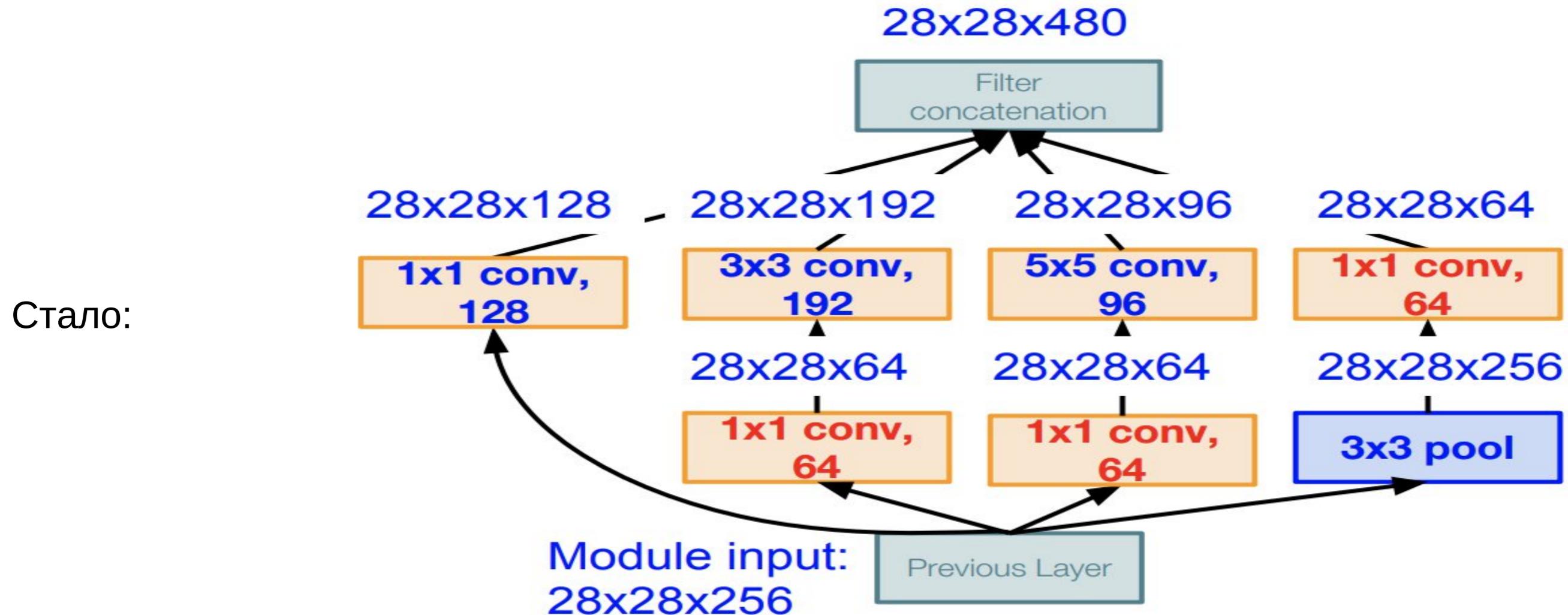
Применим  $1 \times 1$  свёртки в Inception модуле для уменьшения размерностей.





# Inception

Применим  $1 \times 1$  свёртки в Inception модуле для уменьшения размерностей.

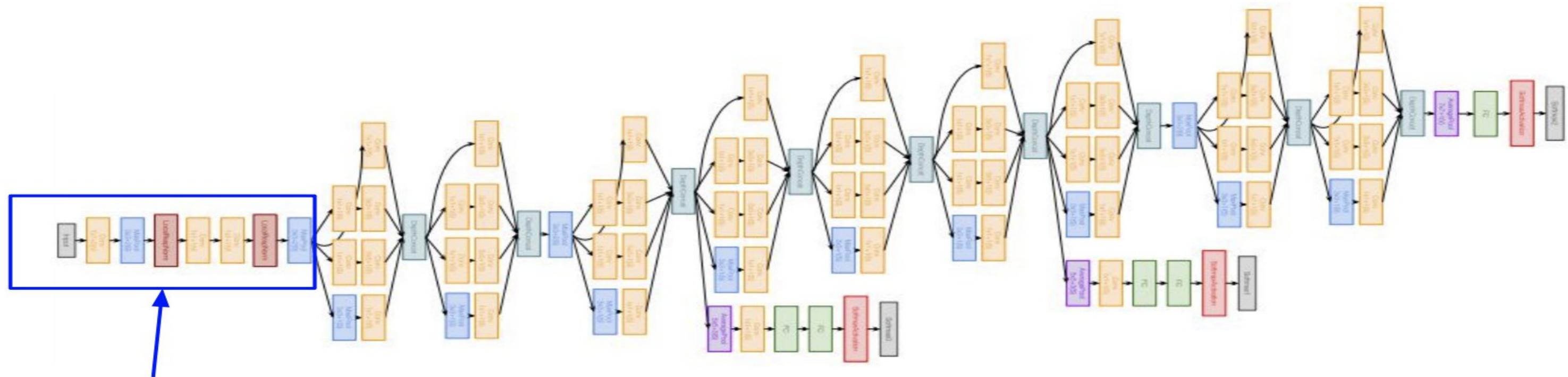


- Уменьшаем выходную размерность после  $3 \times 3$  pool.
- Уменьшаем количество обучаемых параметров.
- Уменьшаем количество вычислений.



# Inception

## Полная архитектура

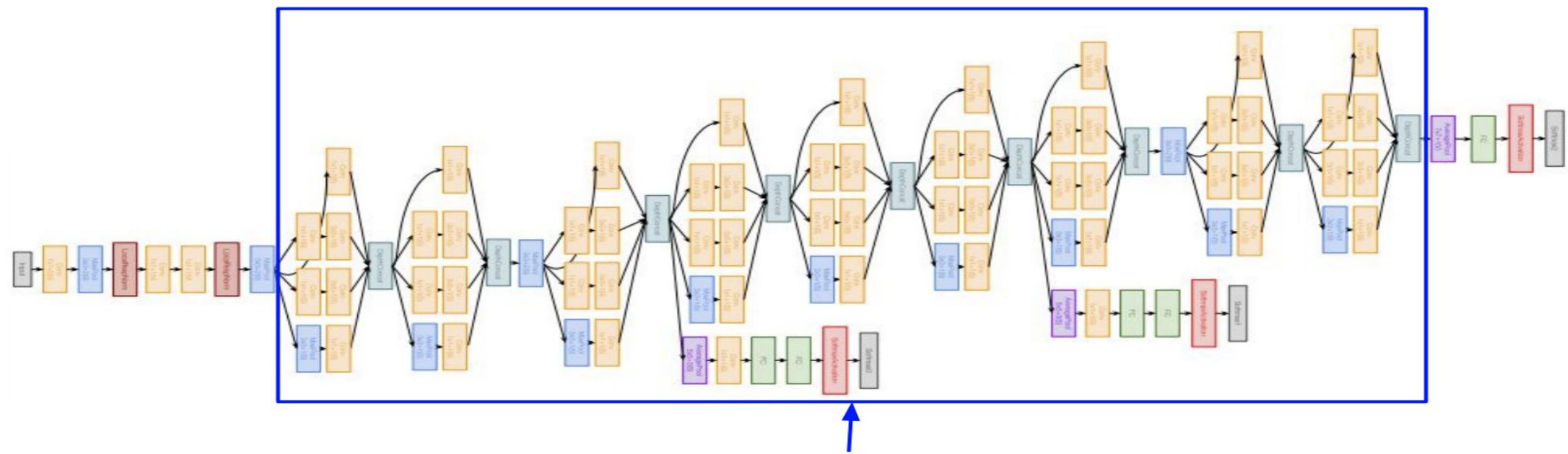


Начальная сеть:  
Conv-Pool-Norm-Conv-Norm-Pool



# Inception

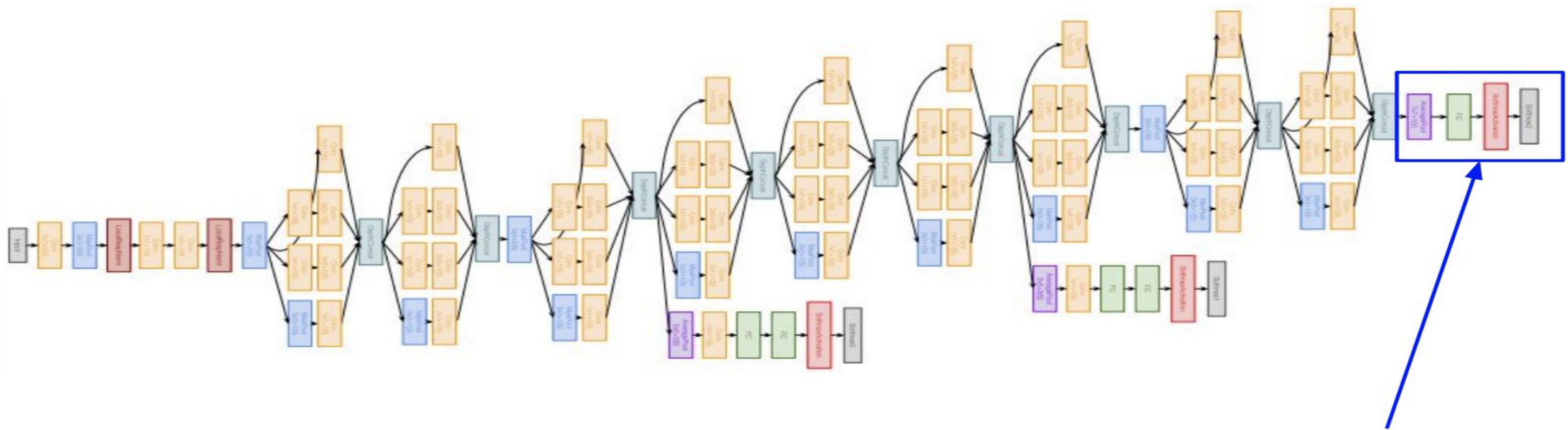
## Полная архитектура





# Inception

## Полная архитектура

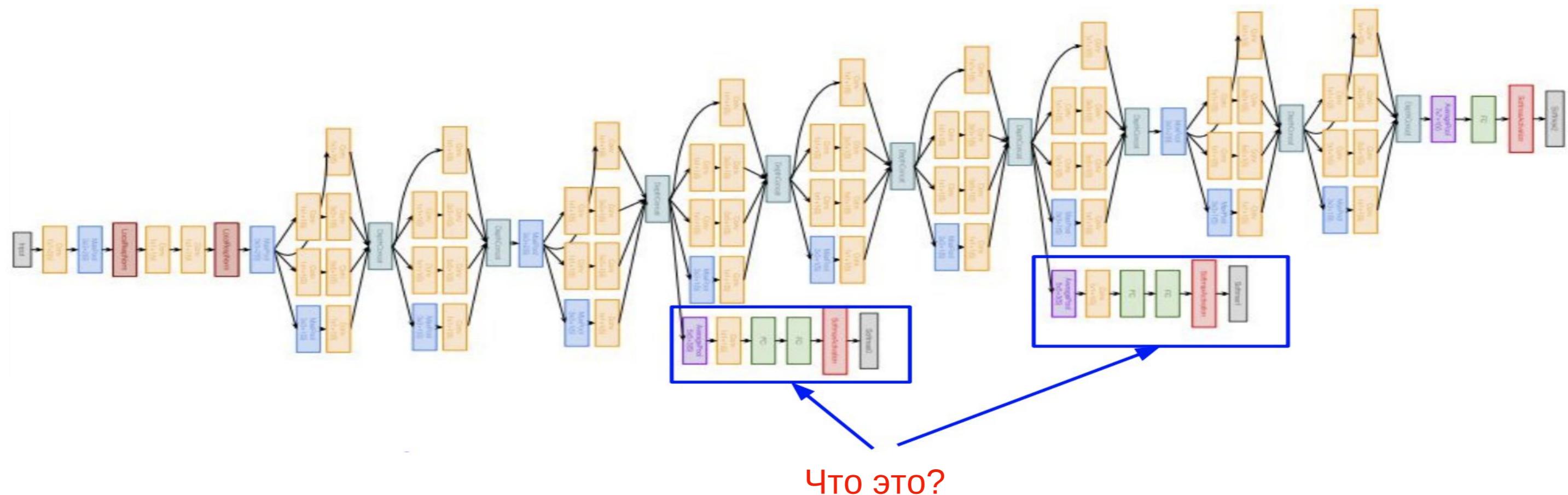


Выходной классификатор



# Inception

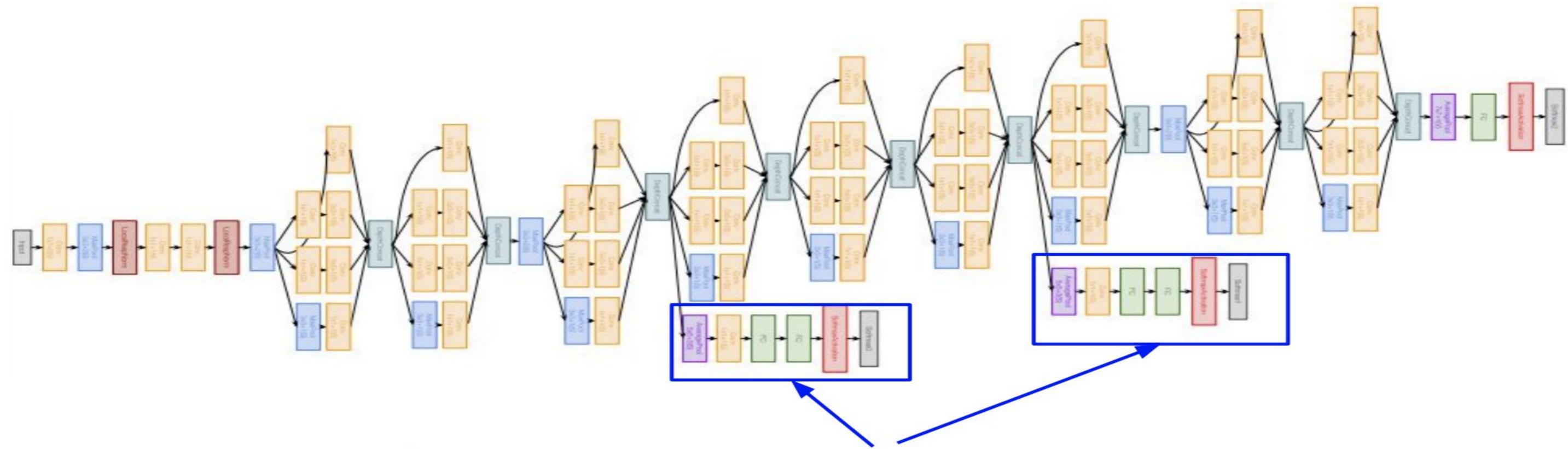
## Полная архитектура





# Inception

## Полная архитектура



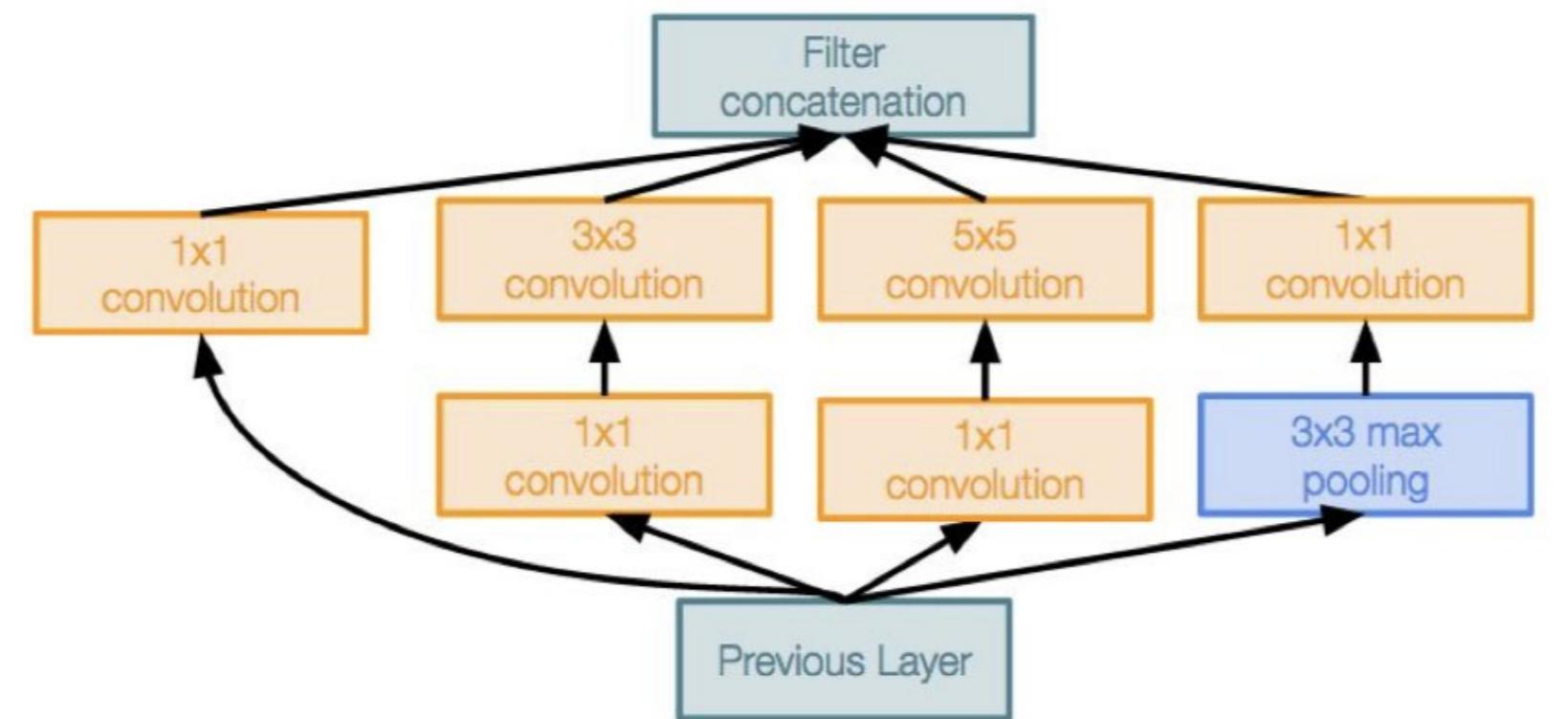
Дополнительные классификаторы.  
Они обучаются вместе со всей сетью.

В глубоких нейросетях возникает проблема затухания градиента.  
Такие “головы” помогают градиенту дотекать до начала сети.

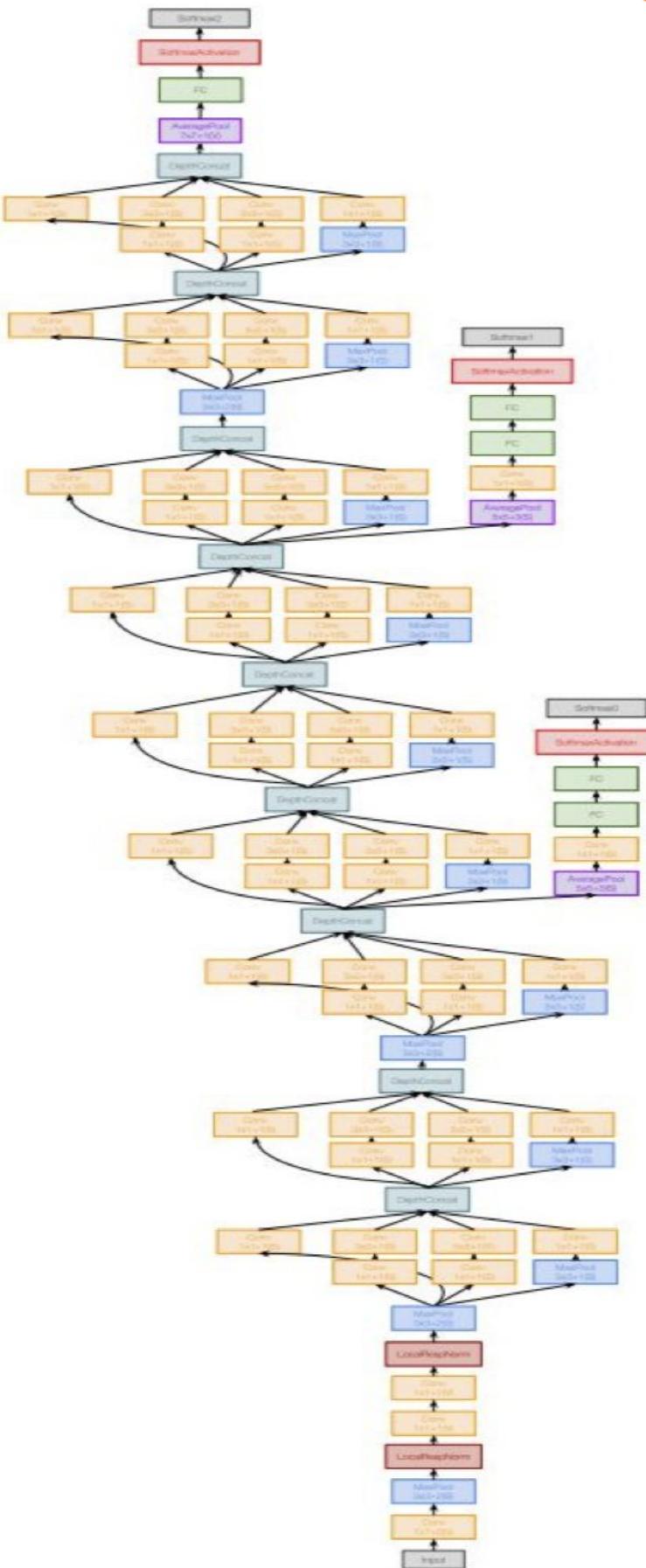


# Inception

- Достигает ошибки 6.7%
  - 22 слоя
  - Содержит только 5М параметров
- В 12 раз меньше, чем AlexNet
- Архитектура построена из одинаковых модулей



Inception module





# Convolution neural networks

AlexNet

VGG

Inception

ResNet

Inception-ResNet

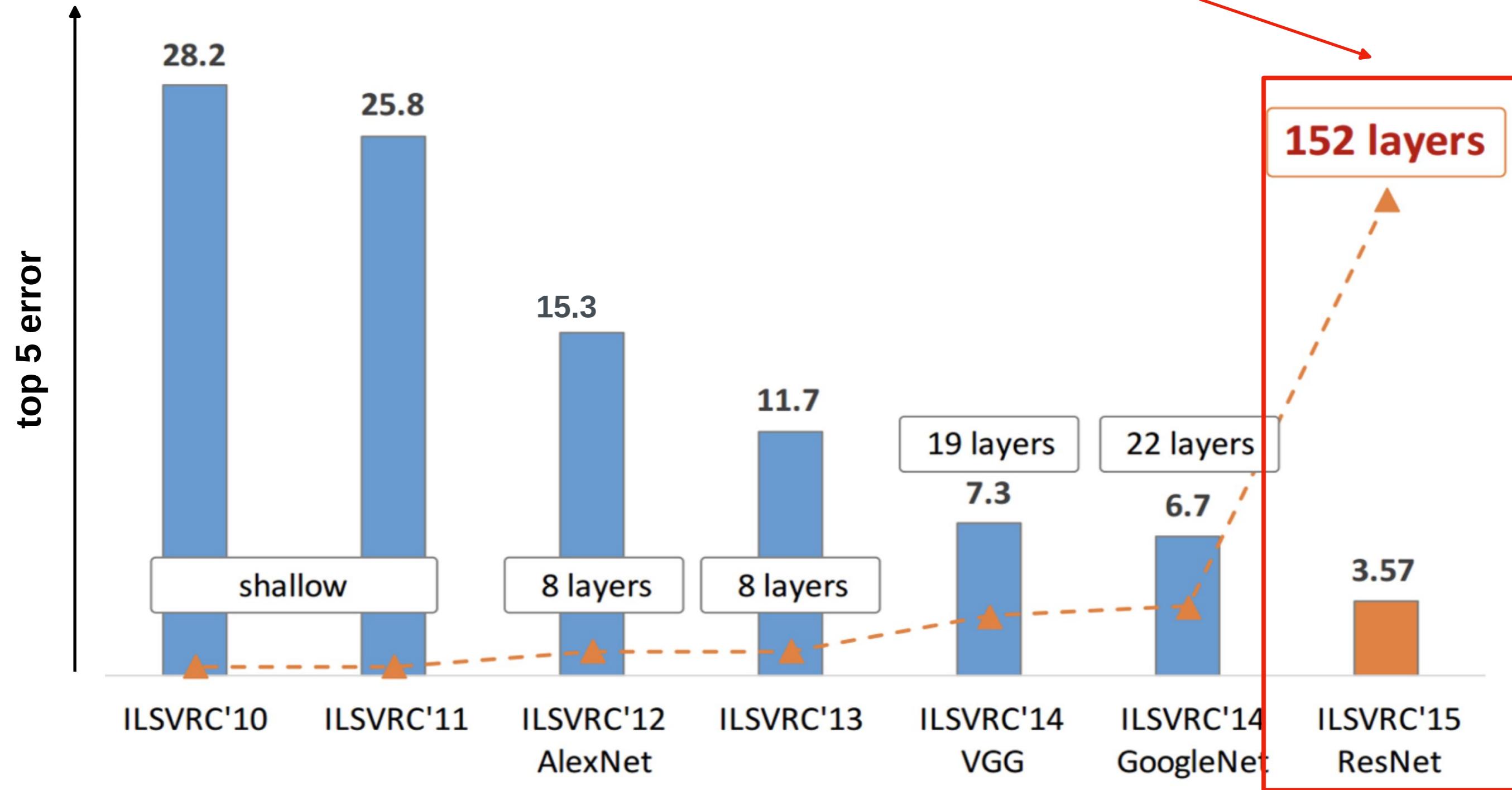
EfficientNet



# ResNet

[Статья](#)

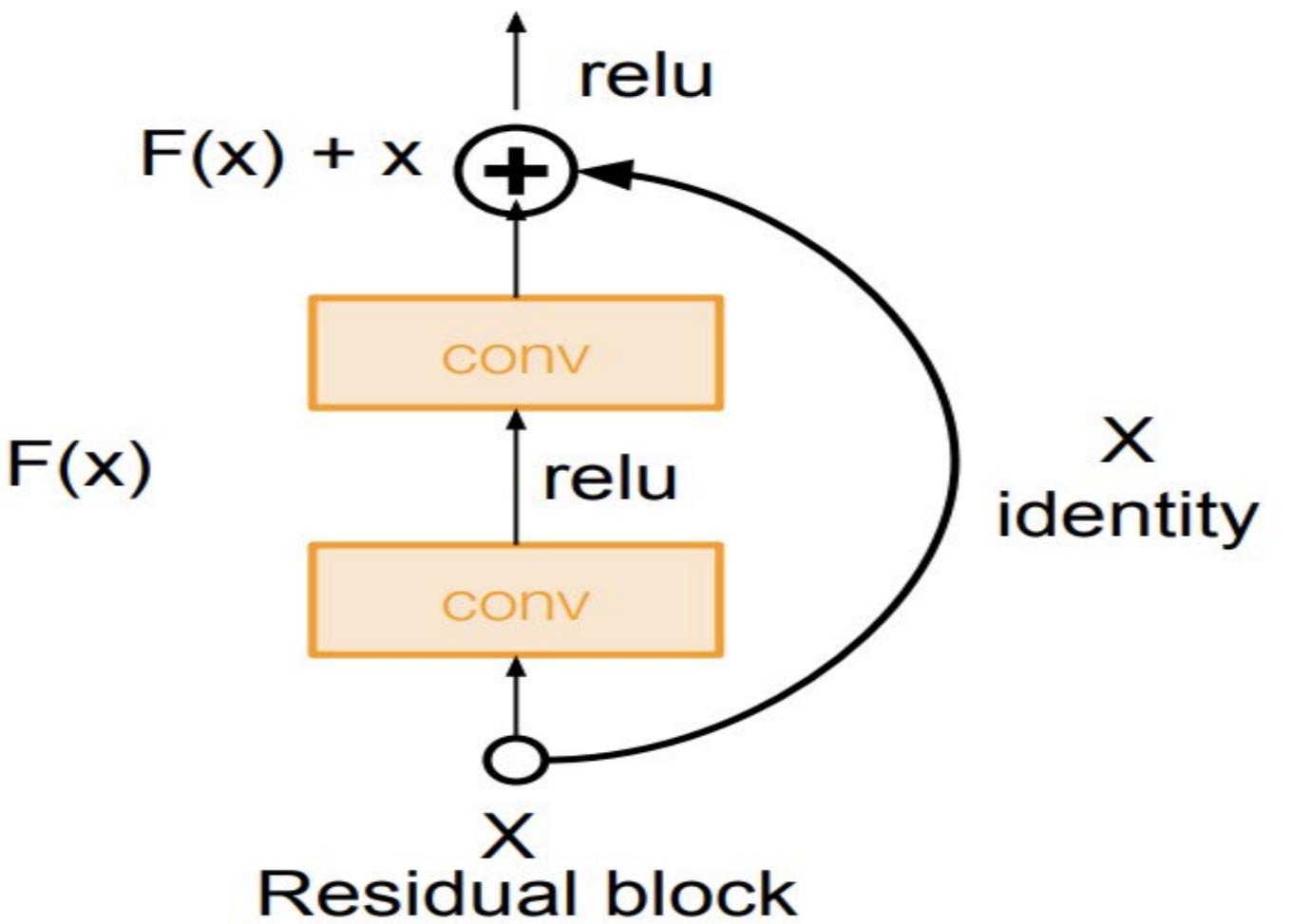
Очень глубокая нейронная сеть



# ResNet



- Достигает ошибки 3.57%
- Ошибка обученного человека - 5%.
- 152 слоя
- Состоит из одинаковых повторяющихся блоков



# ResNet



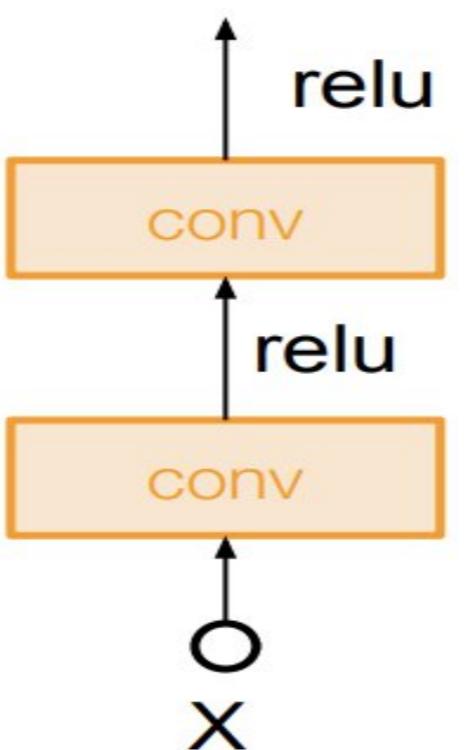
**Идея:**

Будем использовать такие блоки, что на выходе пространственная размерность такая же, что и на входе:

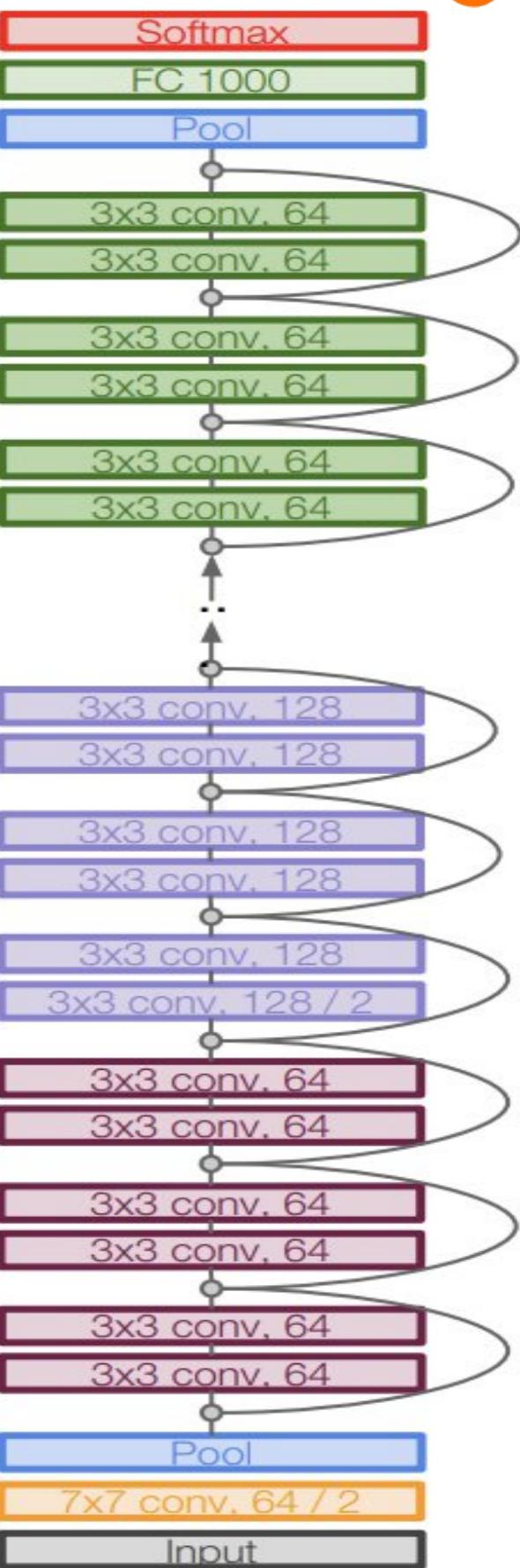
Это достигается при помощи использования **паддинга**.

Пространственная размерность не уменьшается — всегда можем применить ещё свёртки.

Тогда **можем сделать много таких блоков подряд**.



Построим глубокую сеть из таких блоков.

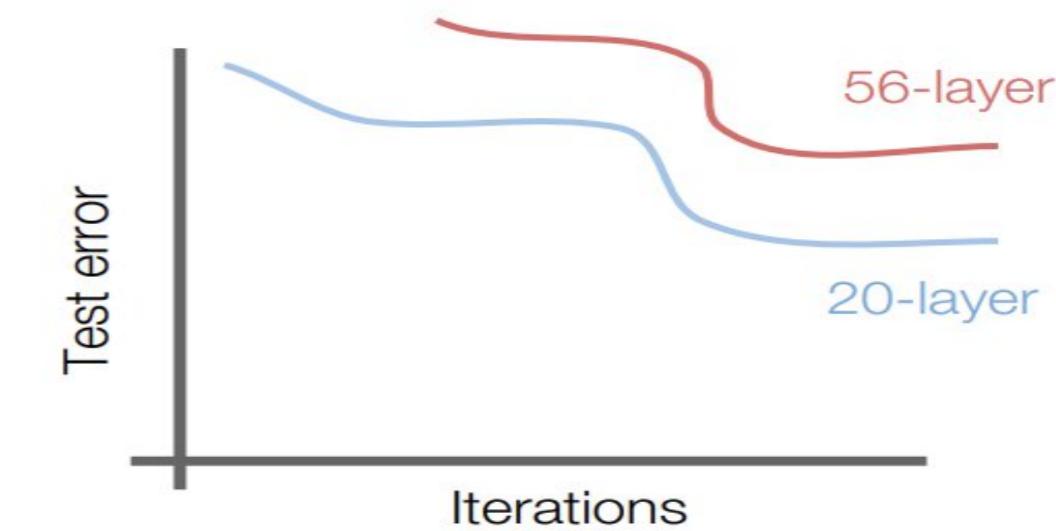
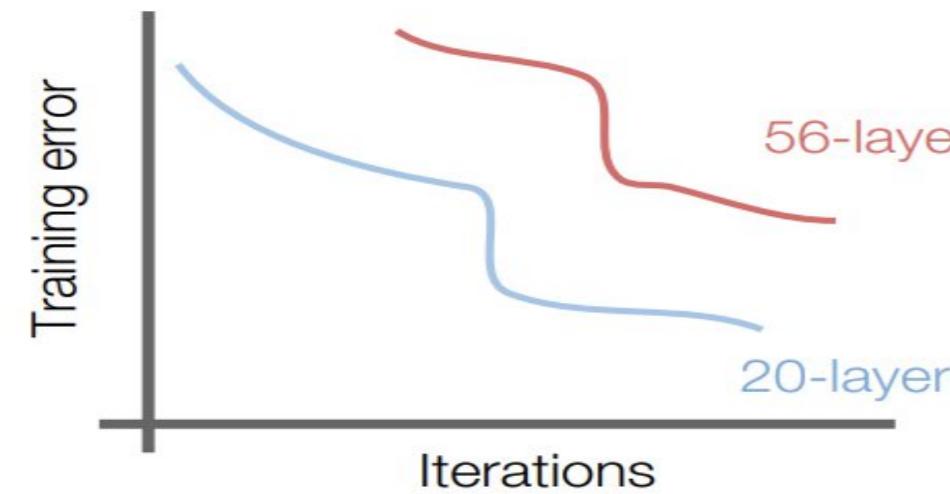




# ResNet

Что происходит во время обучения глубокой нейросети?

Посмотрим на график качества от номера итерации.



**Что странного в этих графиках?**

- Глубокая сеть по определению может представить функции, представляемые менее глубокими нейросетями.

Например, можем взять сеть, в которой первые слои совпадают с маленькой сетью, а далее идут слои, являющиеся identity преобразованием.

- Данное поведение не связано с переобучением.

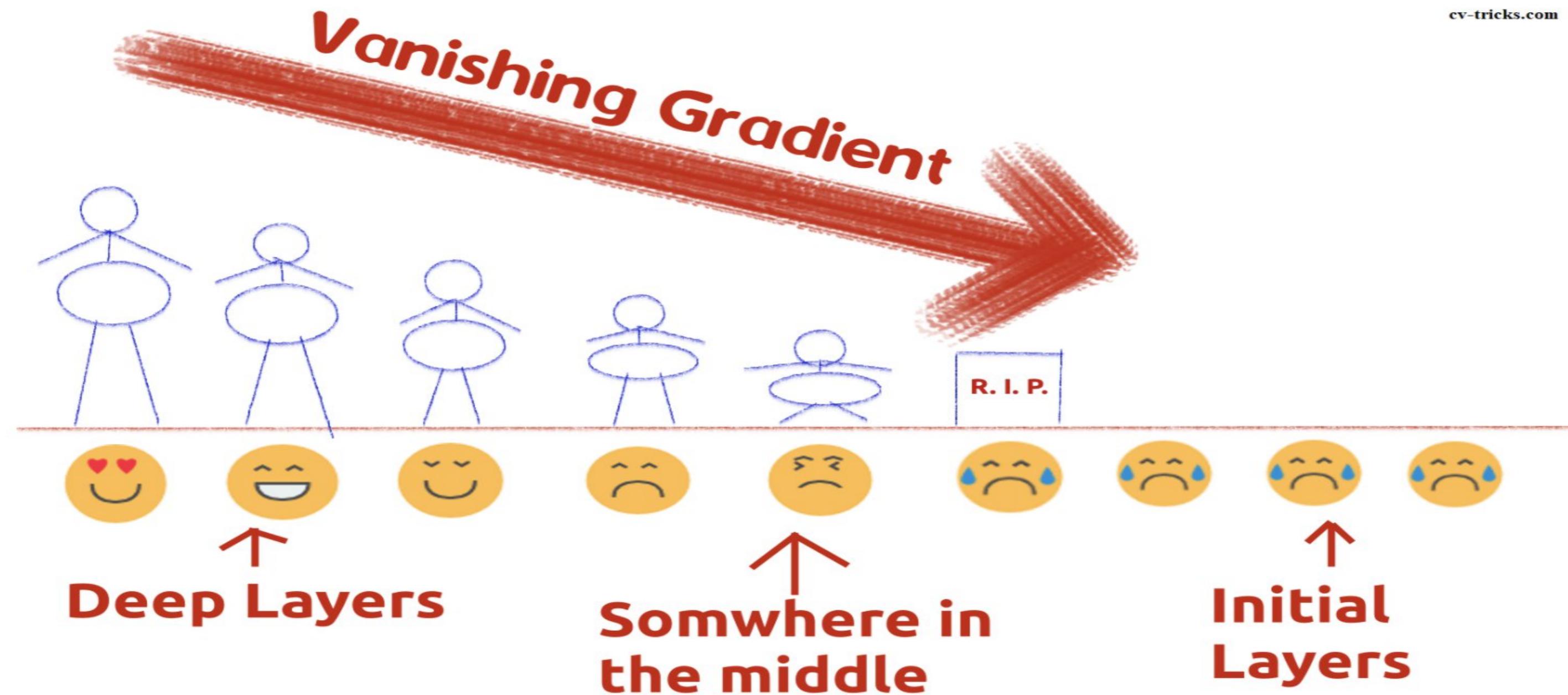
Ошибка на обучении ведет себя так же, как и на teste.



# ResNet

Из-за чего такая проблема?

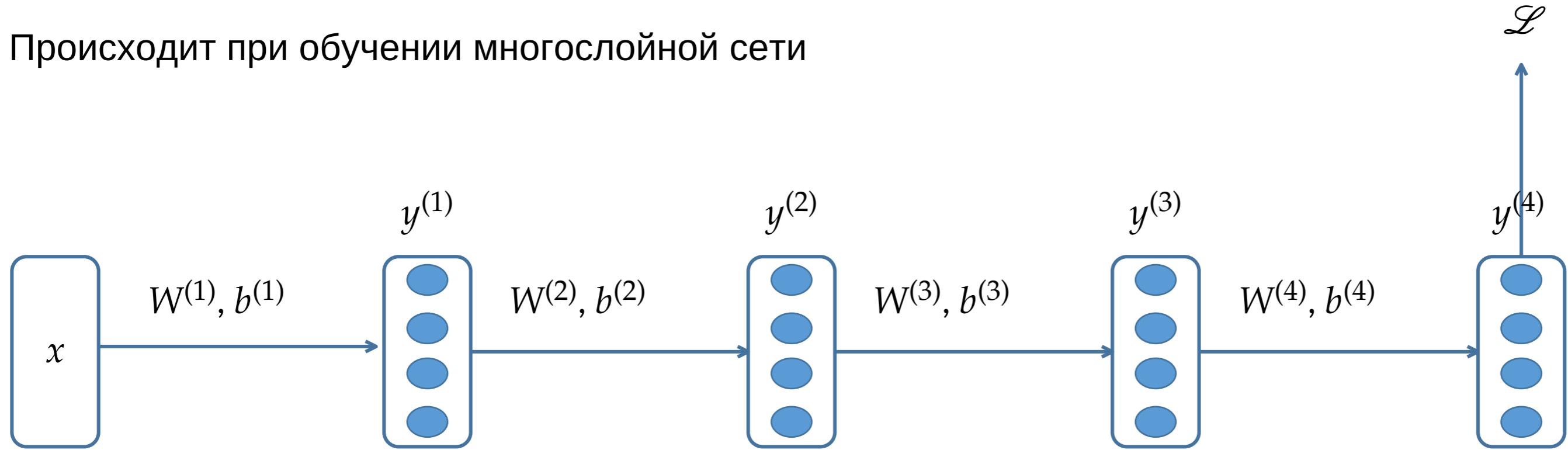
Сеть плохо учится из-за затухания градиента





# Затухание градиента

Происходит при обучении многослойной сети



Сделали forward pass.

Теперь хотим обновить параметры.

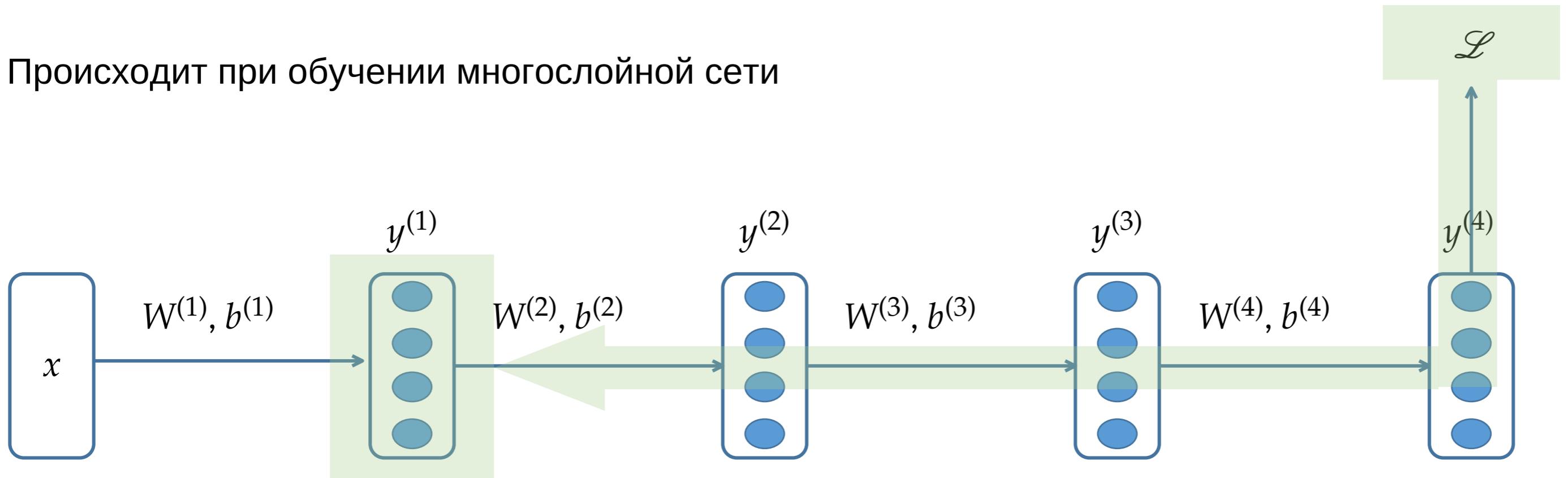
Для этого делаем backward pass.

Посмотрим, как будут считаться градиенты для  $y^{(1)}$ .



# Затухание градиента

Происходит при обучении многослойной сети

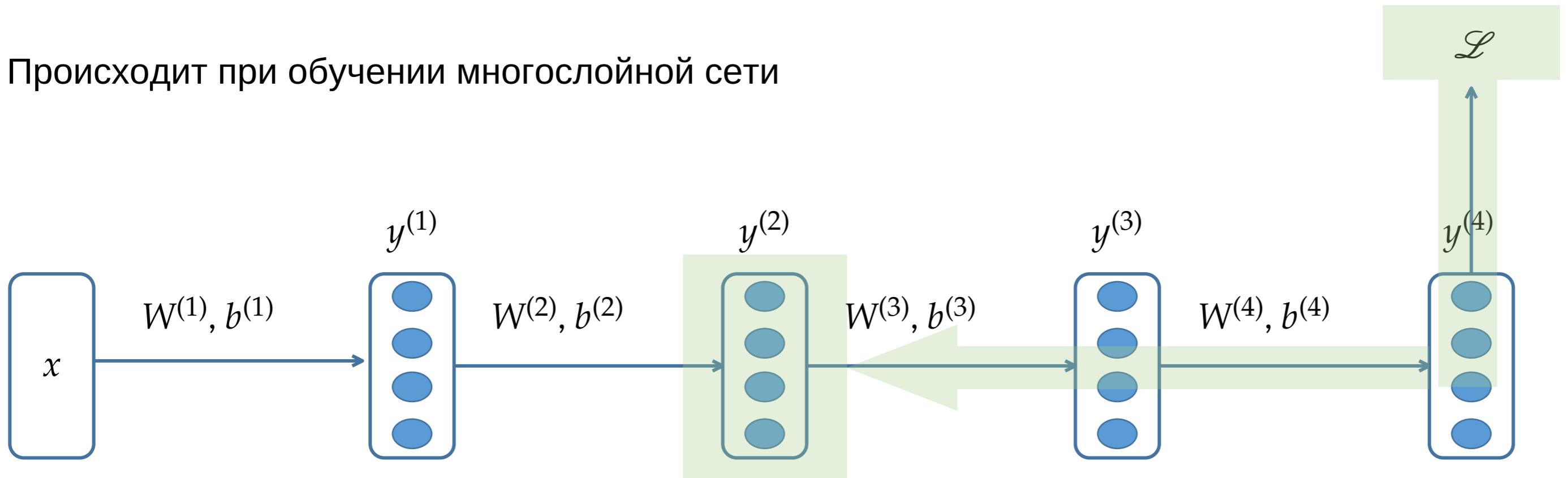


$$\frac{\partial \mathcal{L}}{\partial y^{(1)}}$$



# Затухание градиента

Происходит при обучении многослойной сети

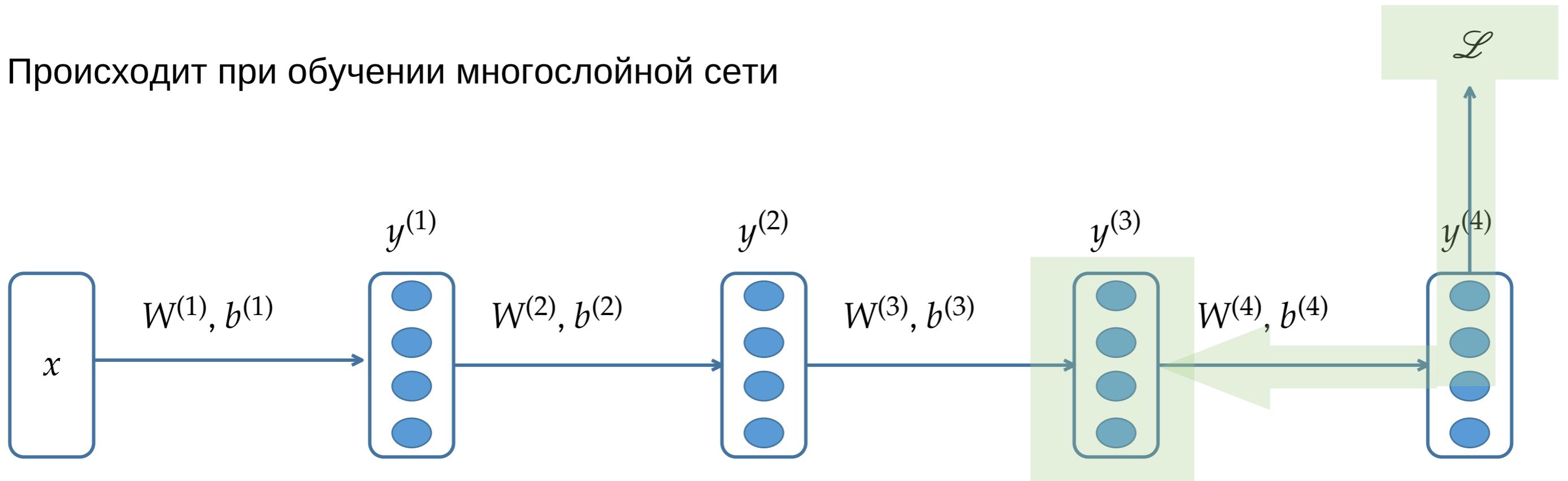


$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}}$$



# Затухание градиента

Происходит при обучении многослойной сети

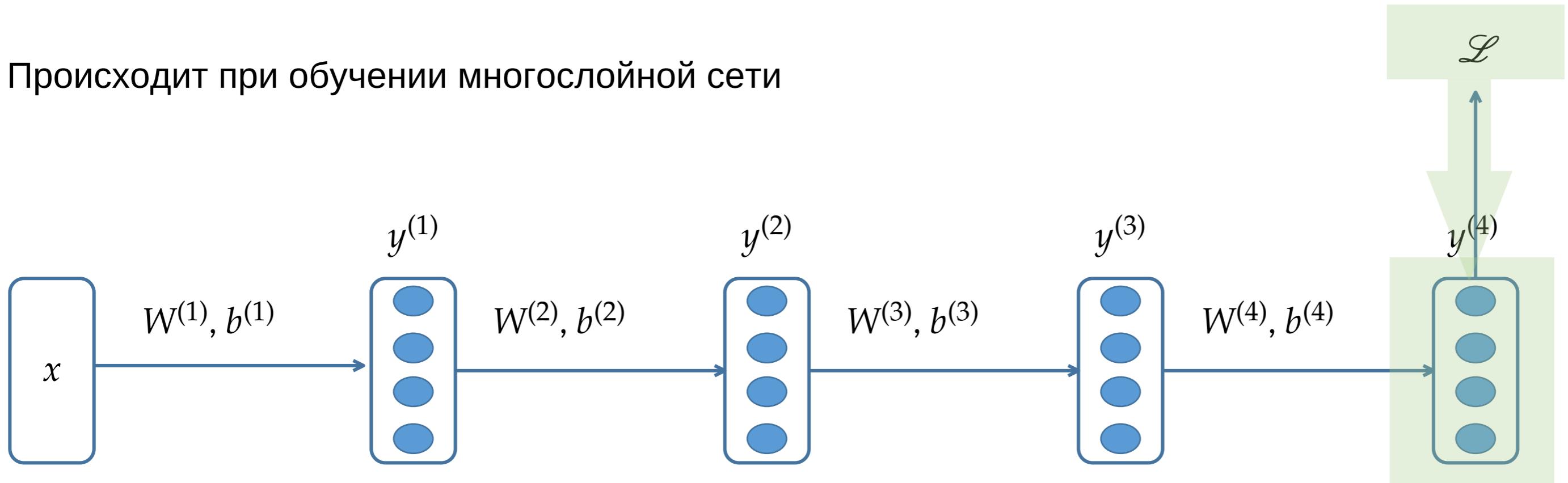


$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(3)}}$$



# Затухание градиента

Происходит при обучении многослойной сети

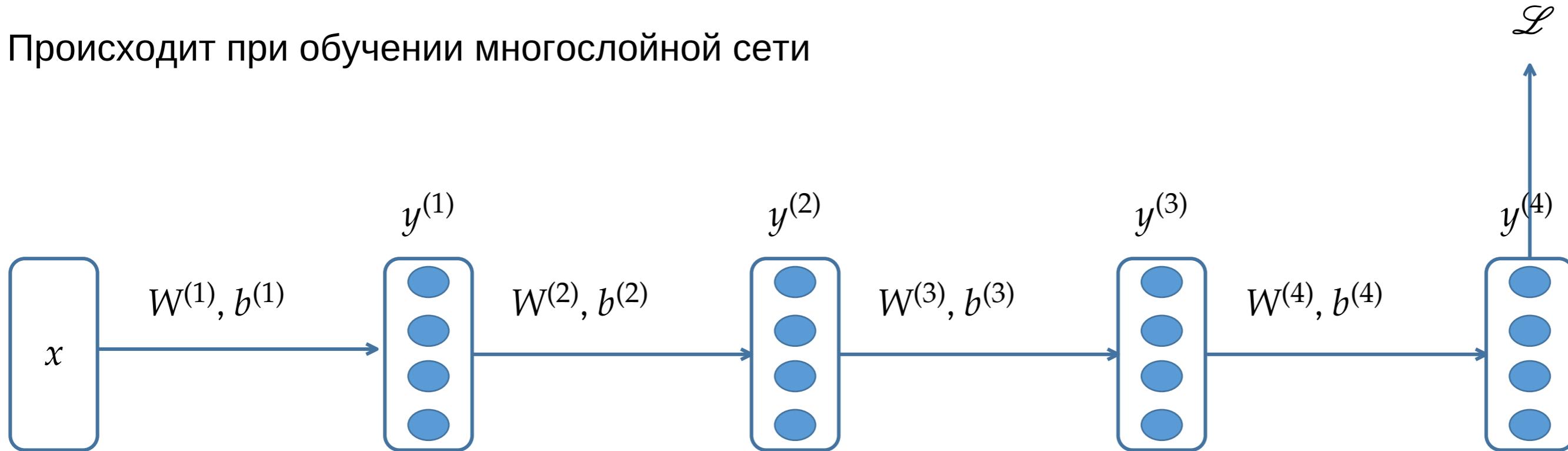


$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(3)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial y^{(4)}}{\partial y^{(3)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(4)}}$$



# Затухание градиента

Происходит при обучении многослойной сети



$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial y^{(4)}}{\partial y^{(3)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(4)}} \sim 0$$

Если градиенты очень маленькие, то из-за пределов вычислительной точности они превращаются в 0.

# ResNet



- Один блок теперь выучивает не функцию  $H(x)$ , а функцию  $F(x) = H(x) - x$ .

Ранее блок пытался предсказать некоторое преобразование  $H(x)$  **полностью**,  
а теперь блок пытается предсказать **лишь необходимую добавку** ко входу.

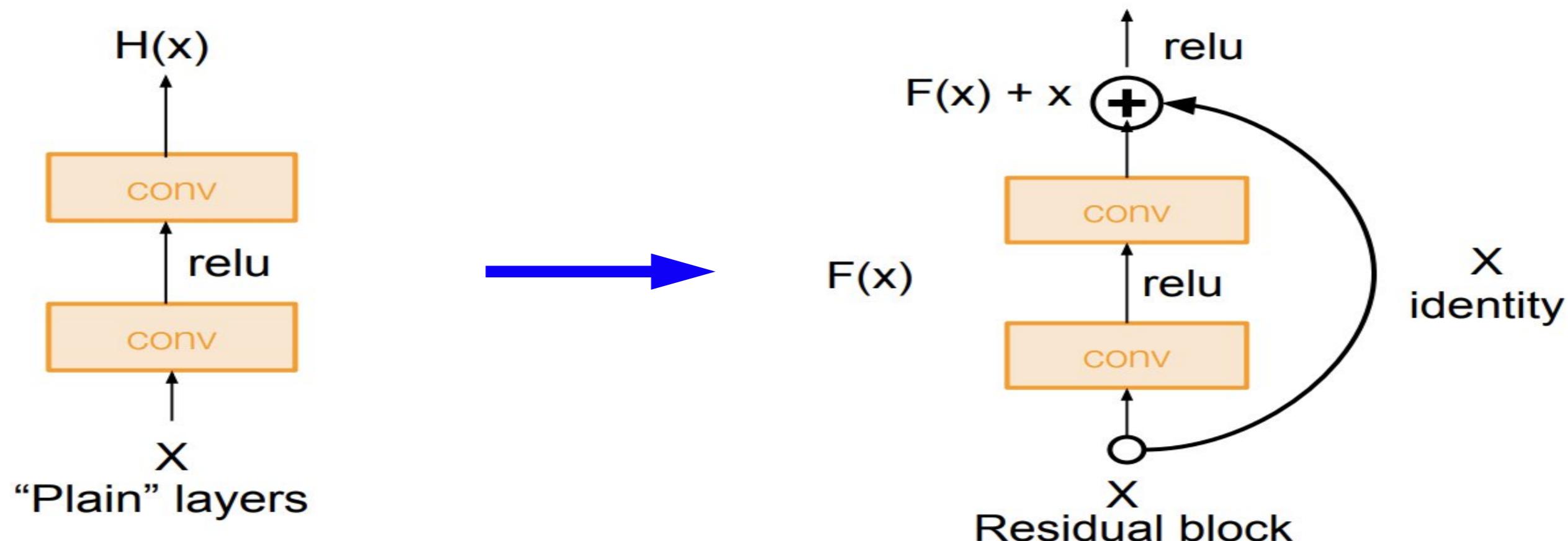
- Сети обычно **проще предсказывать остаток**.

Сеть уже выучила неплохие признаки, поэтому признаки на выходе после блока  
скорей всего должны содержать и признаки, которые были до этого.

Если сети не нужна будет добавка  $F(x)$ , то сеть просто положит  $F(x) = 0$ .

- При обратном распространении ошибки **градиенты практически не могут зануляться**.

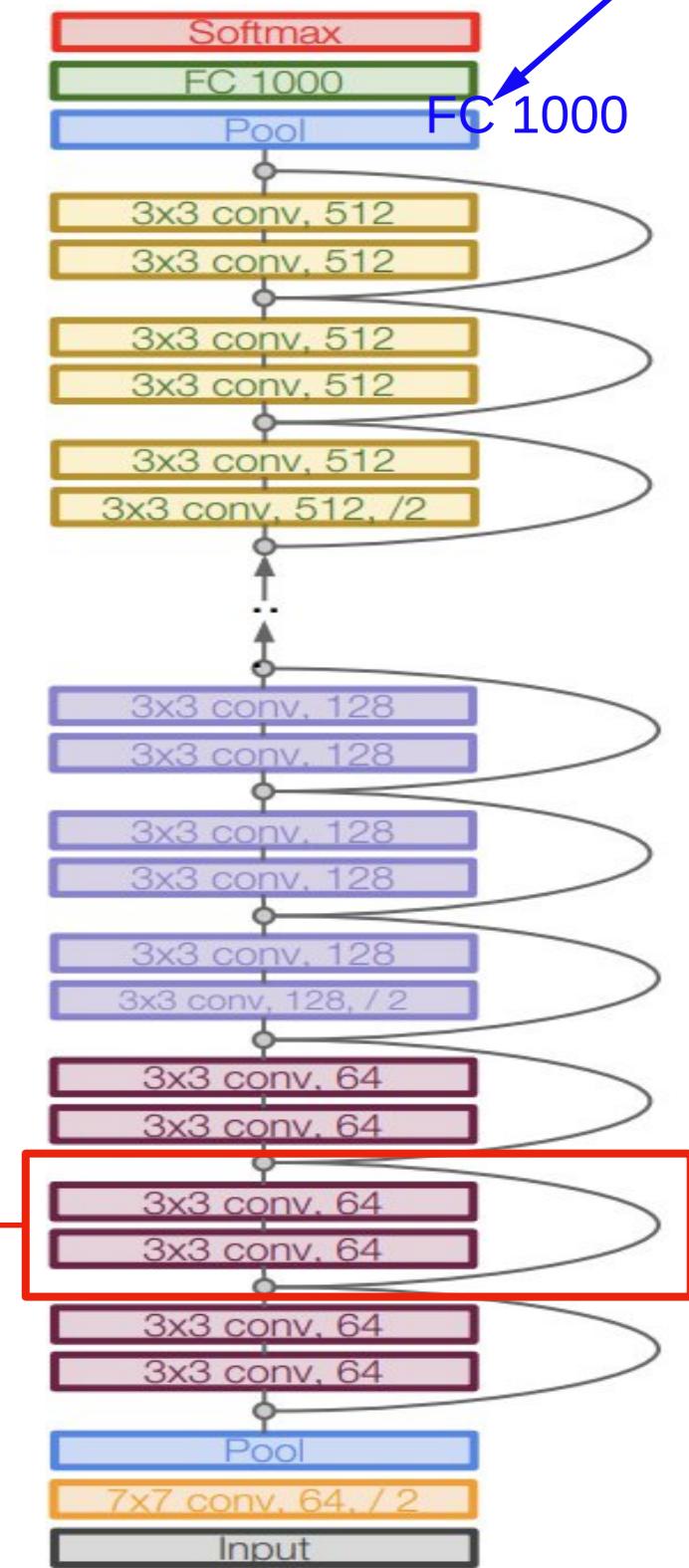
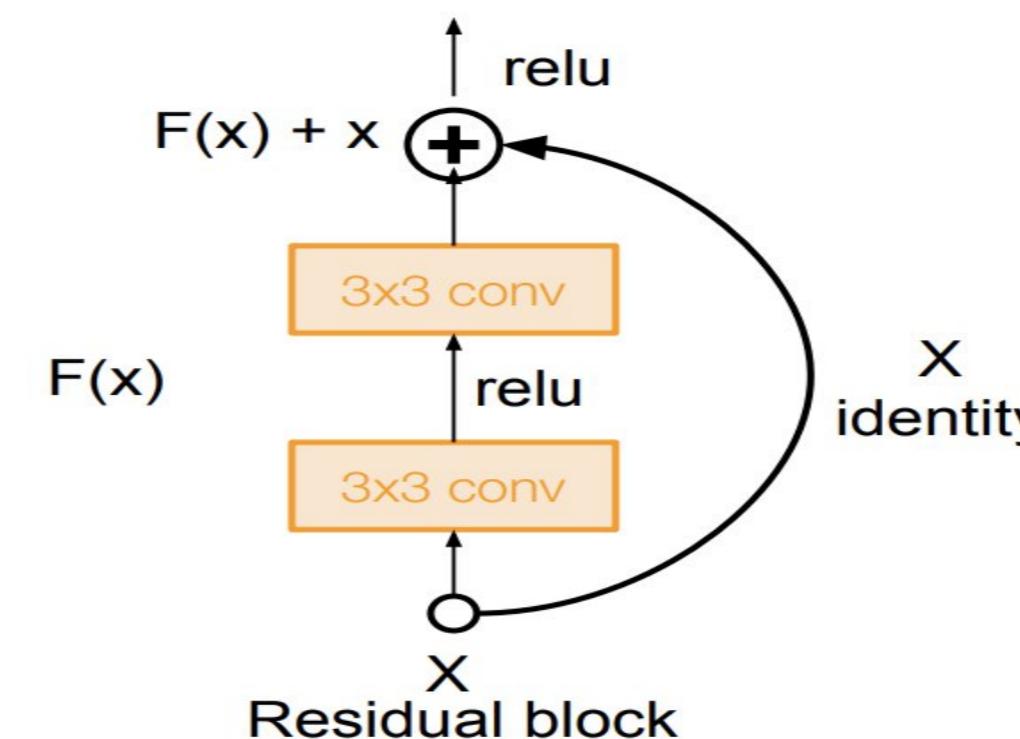
За счет чего появляется возможность обучать **глубокие нейросети**.





# ResNet

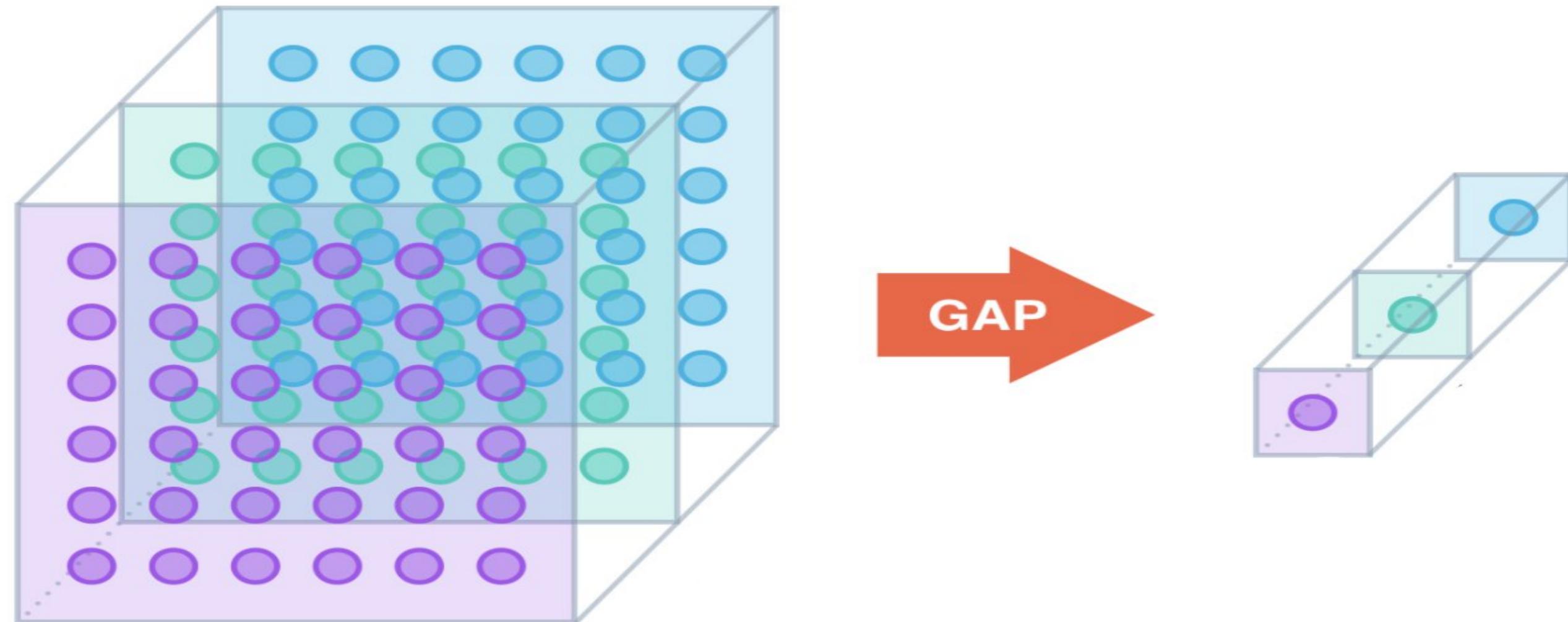
- Стакаем блоки друг за другом.
- В большинстве блоков размерности выхода и входа совпадают.
- Периодически удваиваем количество каналов и уменьшаем  $H$  и  $W$  в 2 раза за счет применения  $stride=2$ .
- В начале стоит один свёрточный слой и max pooling.
- После последнего блока имеем трёхмерную картинку. К ней применяется Global Average Pooling.  
Он переводит картинку размера  $(H_{in}, W_{in}, C_{in})$  в вектор размера  $C_{in}$ .
- В конце применяется лишь один Dense слой.





# ResNet

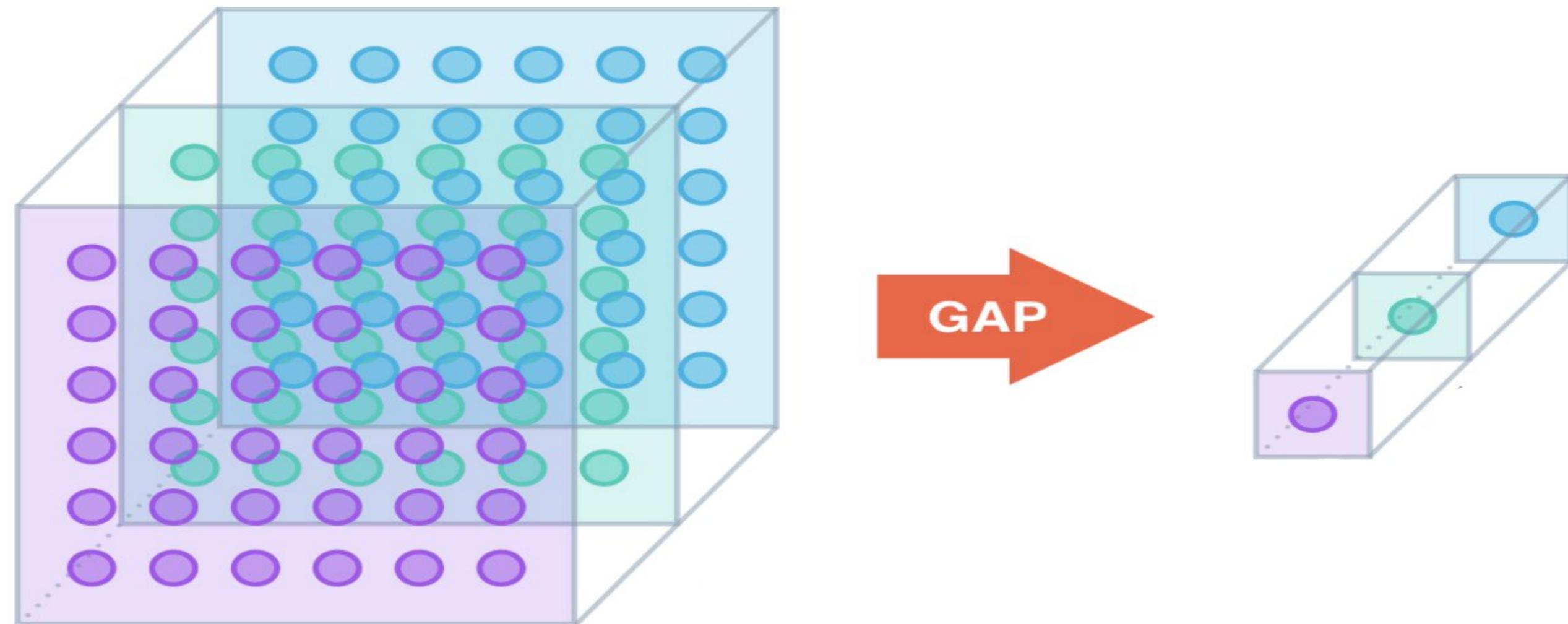
## Global Average Pooling





# ResNet

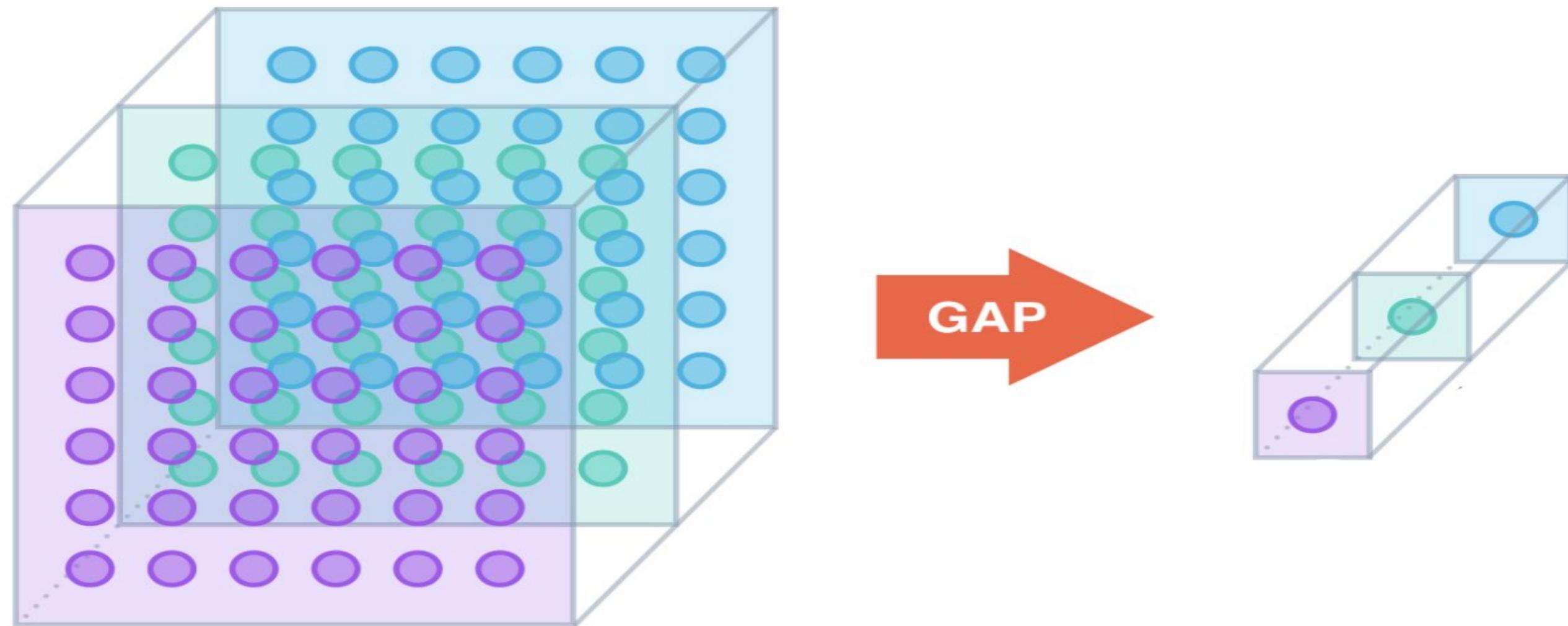
## Global Average Pooling





# ResNet

## Global Average Pooling



- По каждой карте считаем среднее значение.
- Переводим картинку размера в выход размера .

*Замечание:*

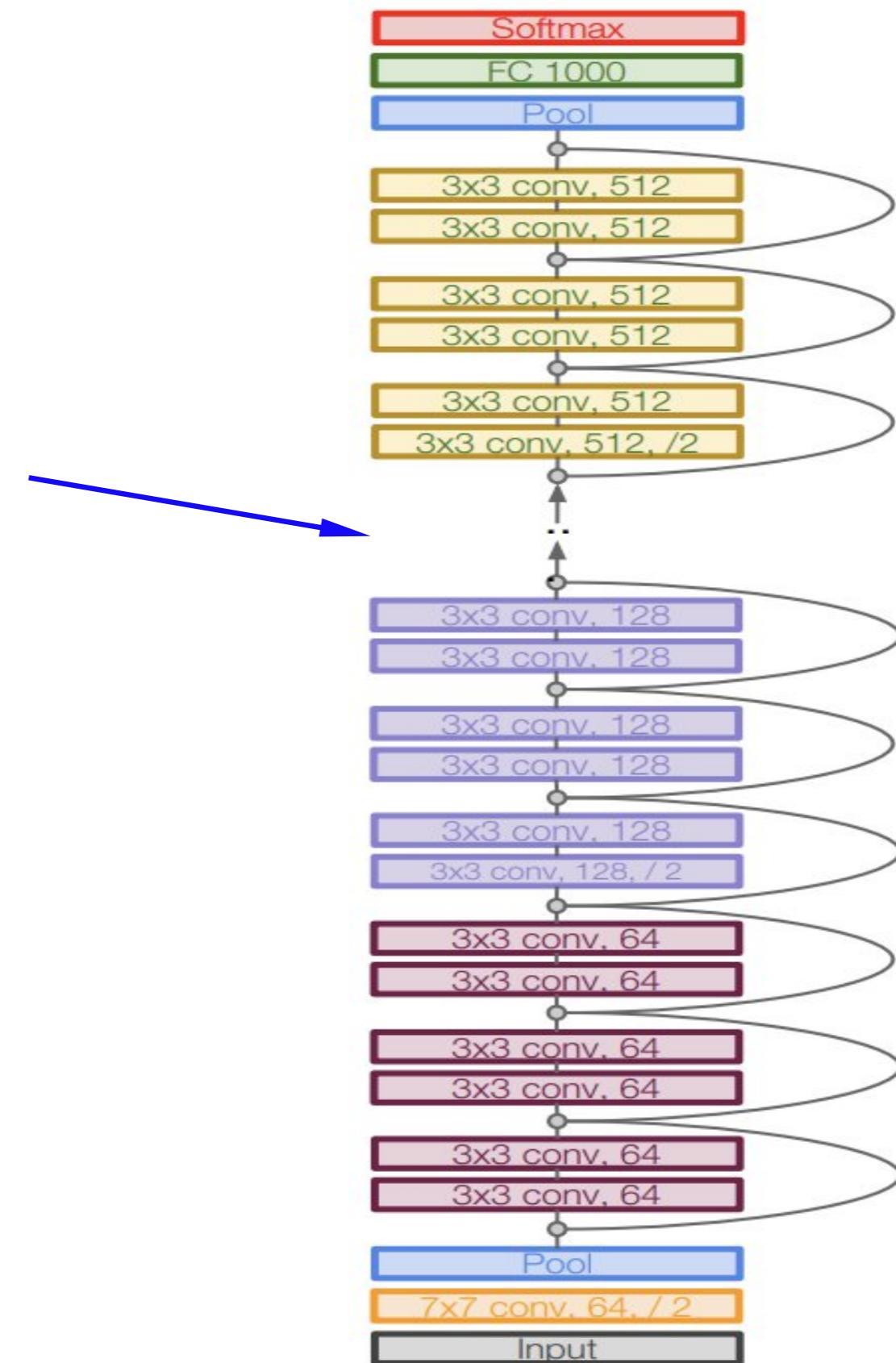
Вместо подсчета среднего можно считать и другую функцию.

Например, максимум.



# ResNet

Итоговая глубина - 34, 50, 101 или 152 слоя.

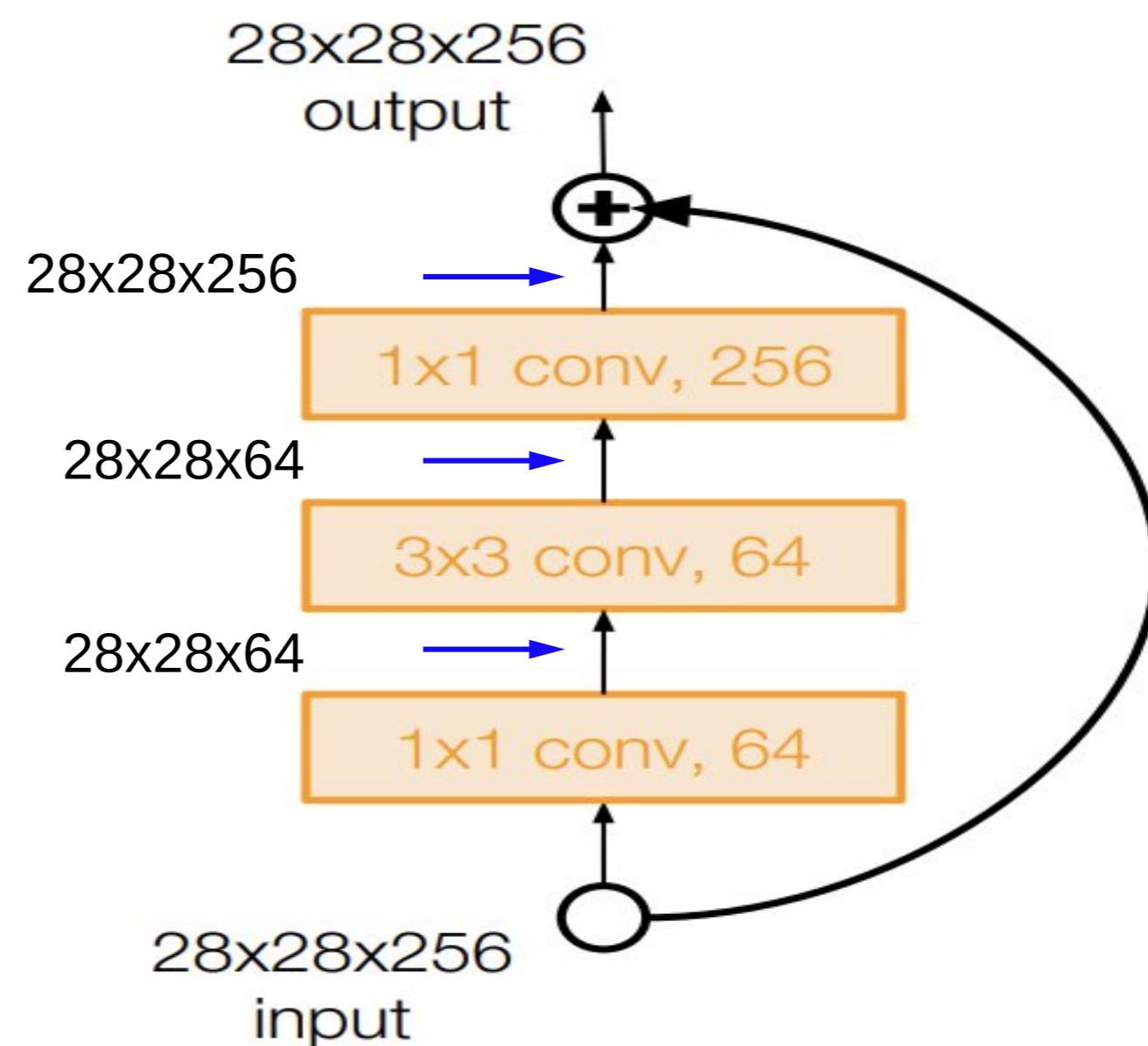




# ResNet

Для глубоких сетей (50+ слоев) используем свёртки  $1 \times 1$  для вычислительной эффективности.

Блок выглядит следующим образом:



- Уменьшается количество параметров
- Более быстрый подсчет при прямом и обратном распространении.



# ResNet

- Глубокие сети теперь могут обучаться.
- Глубокие сети могут достичь меньшей ошибки, чем неглубокие.
- ResNet победил сразу в пяти соревнованиях по различным видам задач.

## MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: “Ultra-deep” (quote Yann) **152-layer nets**
  - ImageNet Detection: **16%** better than 2nd
  - ImageNet Localization: **27%** better than 2nd
  - COCO Detection: **11%** better than 2nd
  - COCO Segmentation: **12%** better than 2nd



# Convolution neural networks

AlexNet

VGG

Inception

ResNet

Inception-ResNet

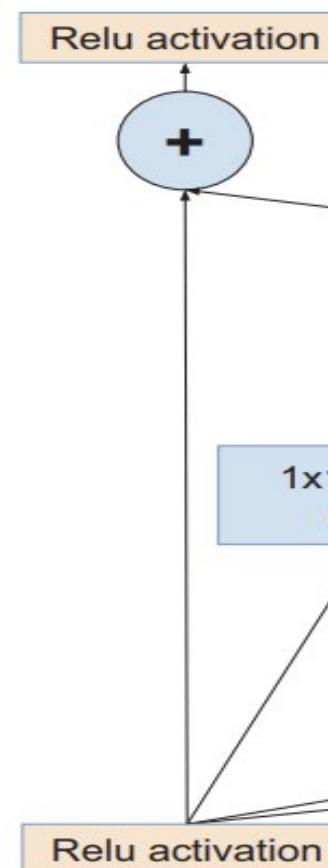
EfficientNet



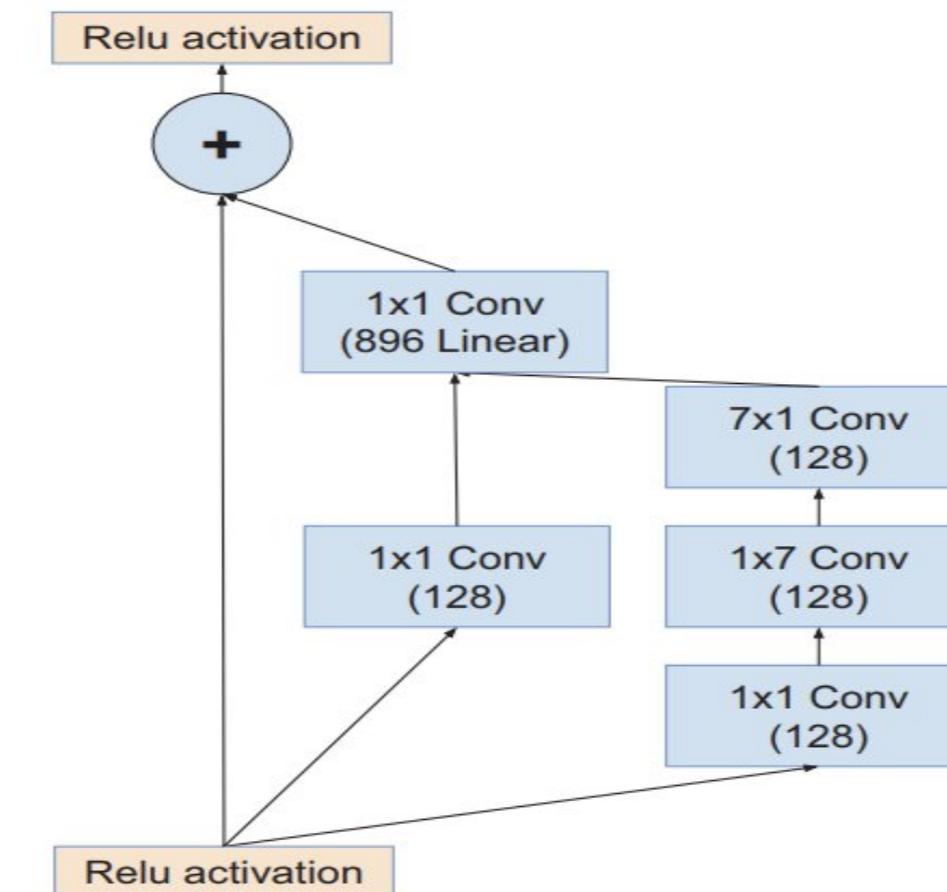
# Inception-ResNet

Почему бы не построить другие блоки с skip connection?

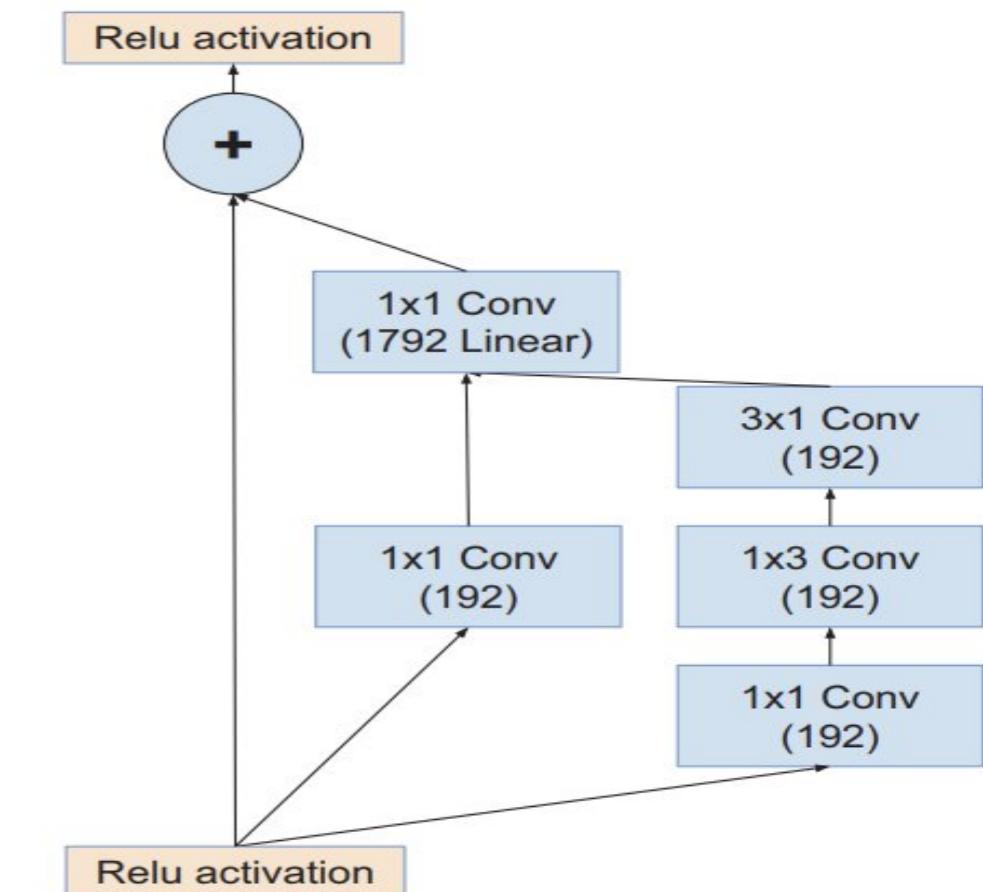
Возьмем блок у Inception и добавим к нему skip connection.



Block A



Block B



Block C

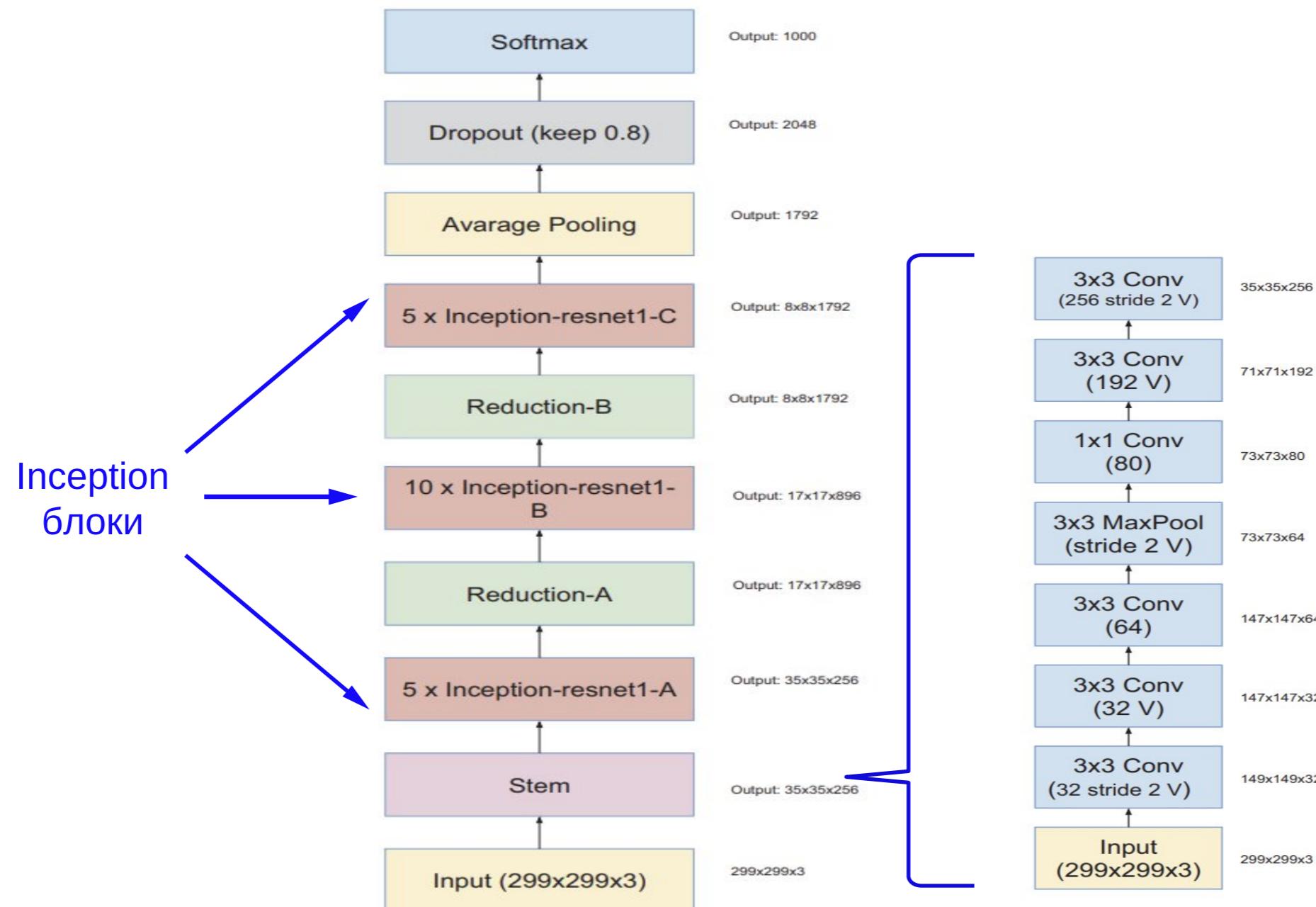


# Inception-ResNet

Почему бы не построить другие блоки с skip connection?

Возьмем блок у Inception и добавим к нему skip connection.

И, как и ранее, настакаем такие блоки друг на друга.





# Convolution neural networks

AlexNet

VGG

Inception

ResNet

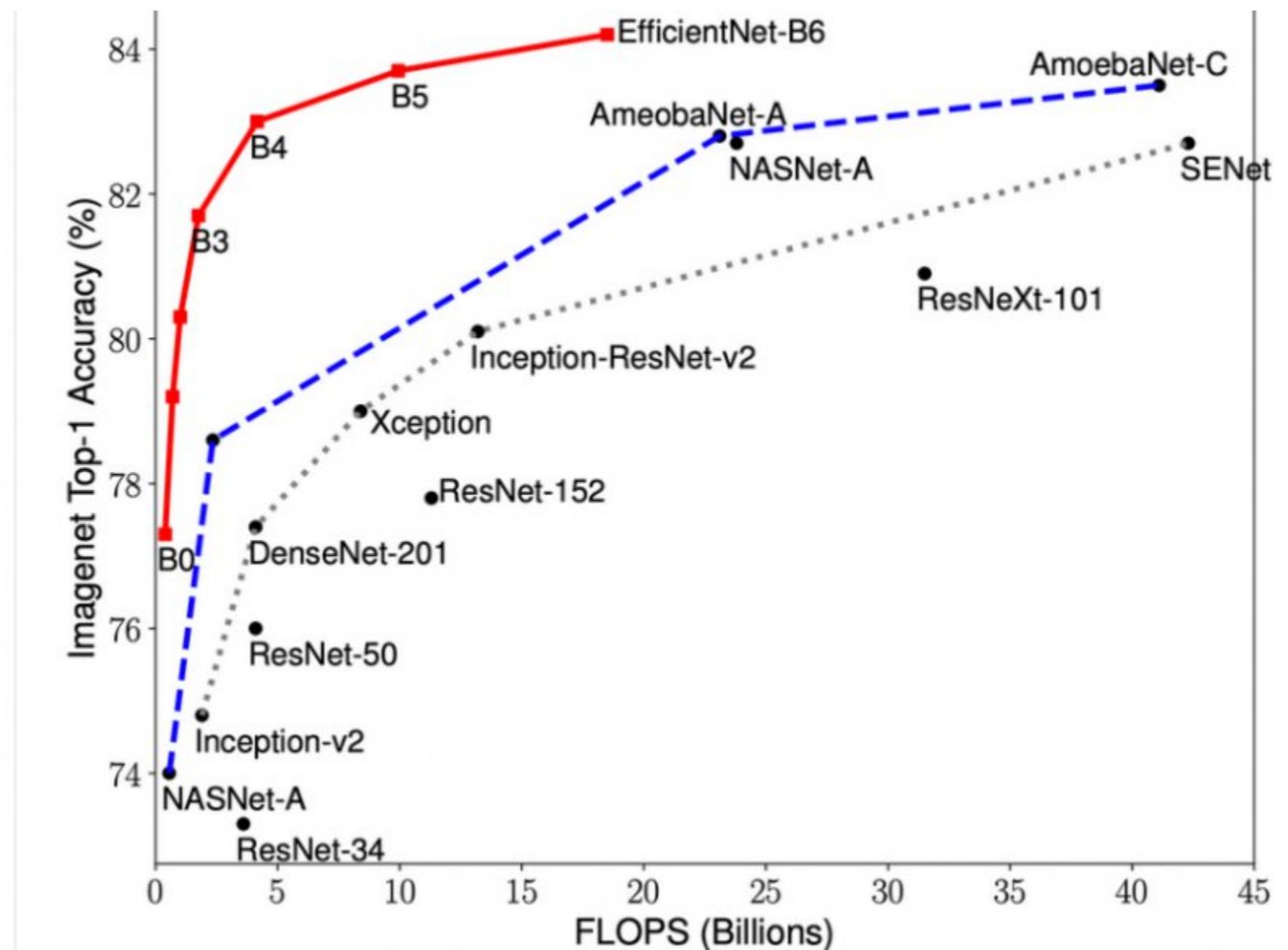
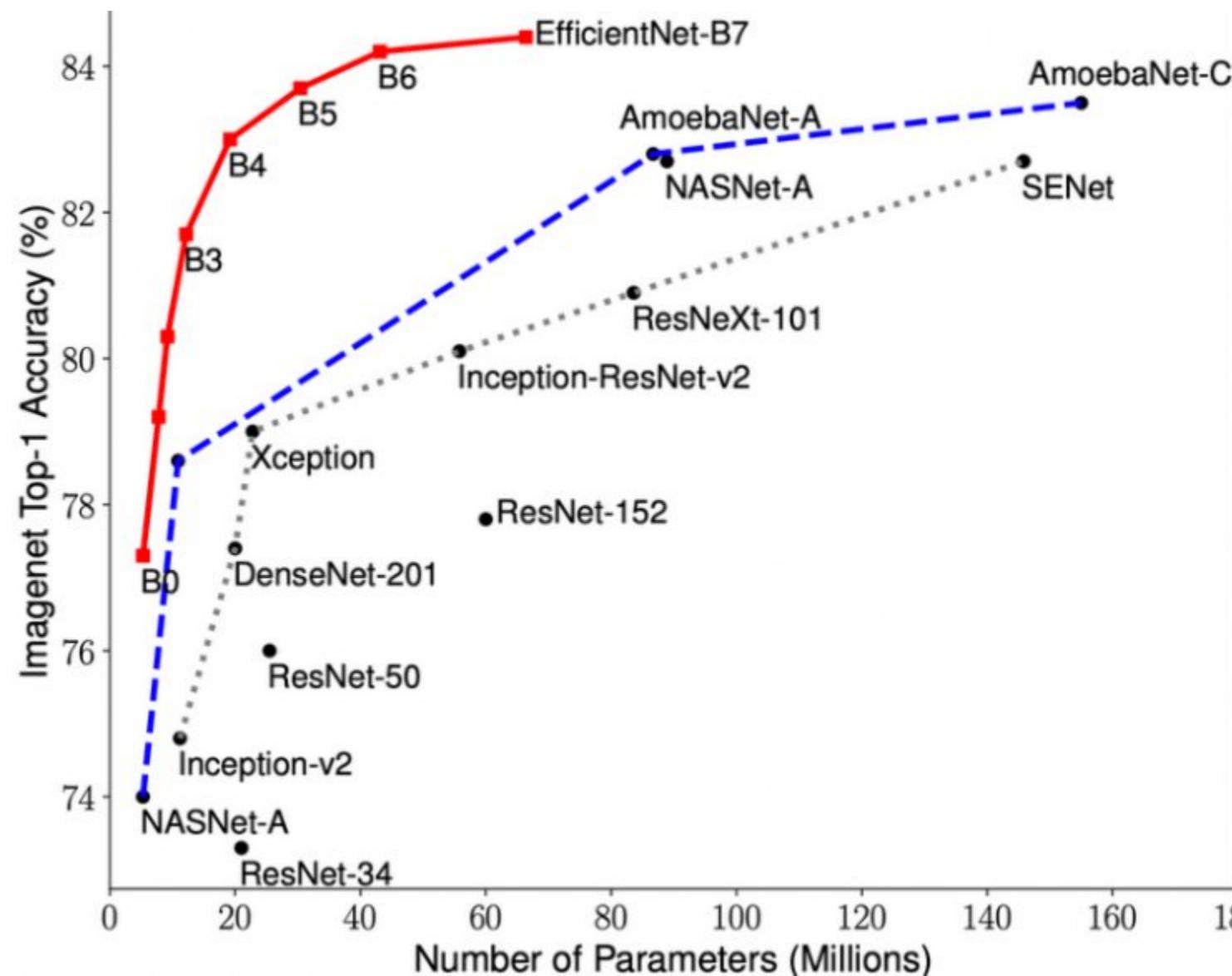
Inception-ResNet

EfficientNet



# EfficientNet

Авторы [статьи](#) предлагают новый метод составного масштабирования (compound scaling method), который равномерно масштабирует глубину/ширину/разрешение с фиксированными пропорциями между ними. Архитектура подбирается с помощью «Neural Architecture Search».



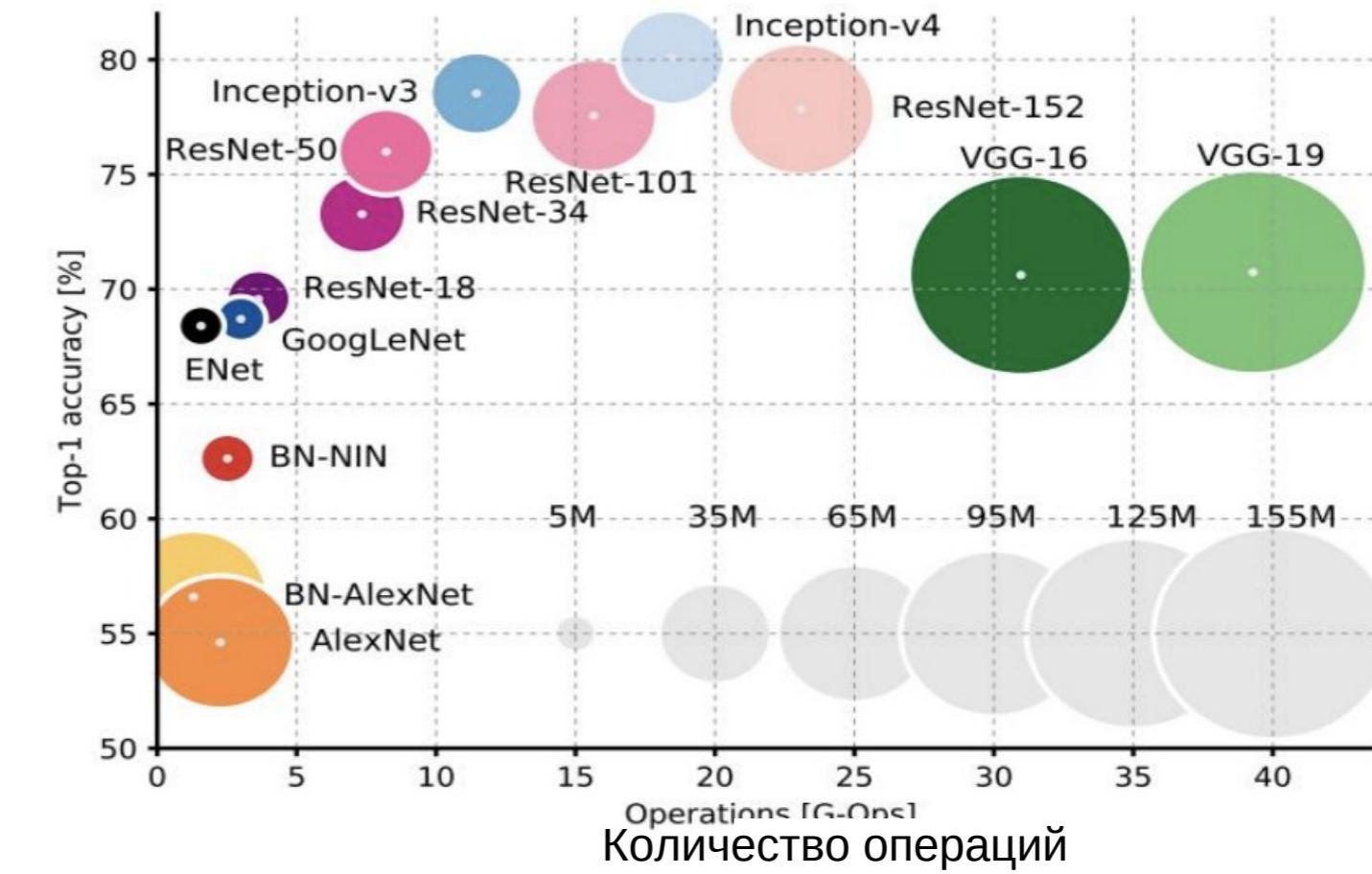
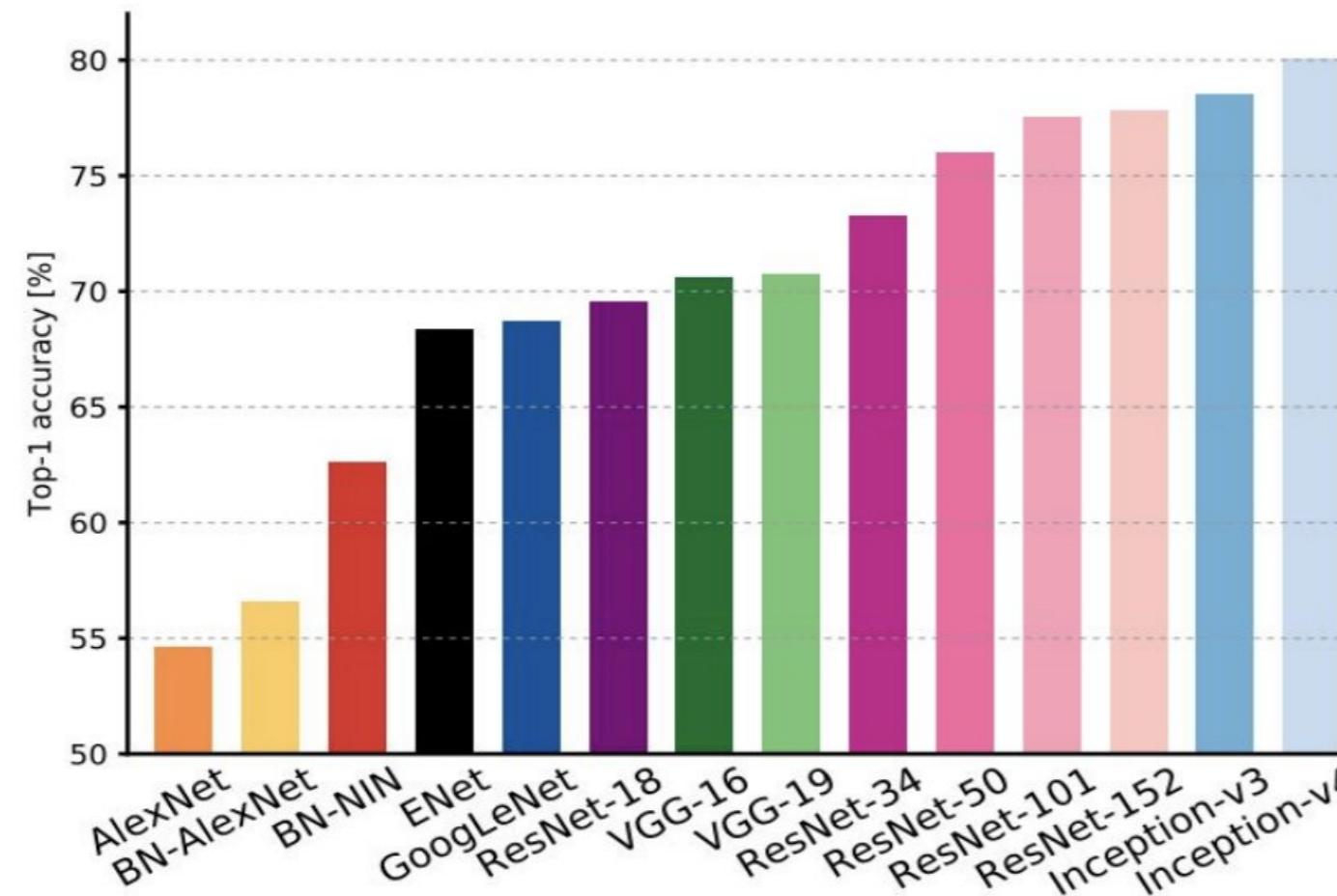


# Сравнение моделей



# Сравнение

Сравним модели по качеству и вычислительной сложности.

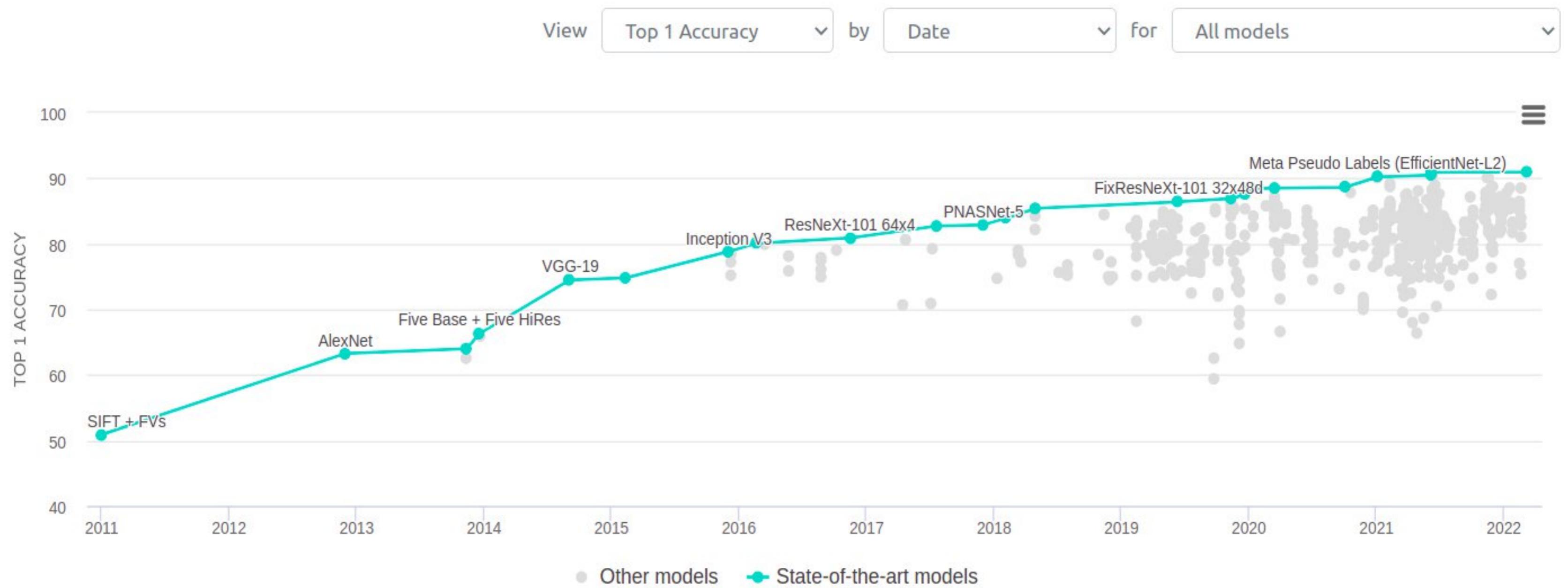


Размер круга - затраты по памяти.



# Сравнение

Результаты соревнования по годам: top 1 accuracy

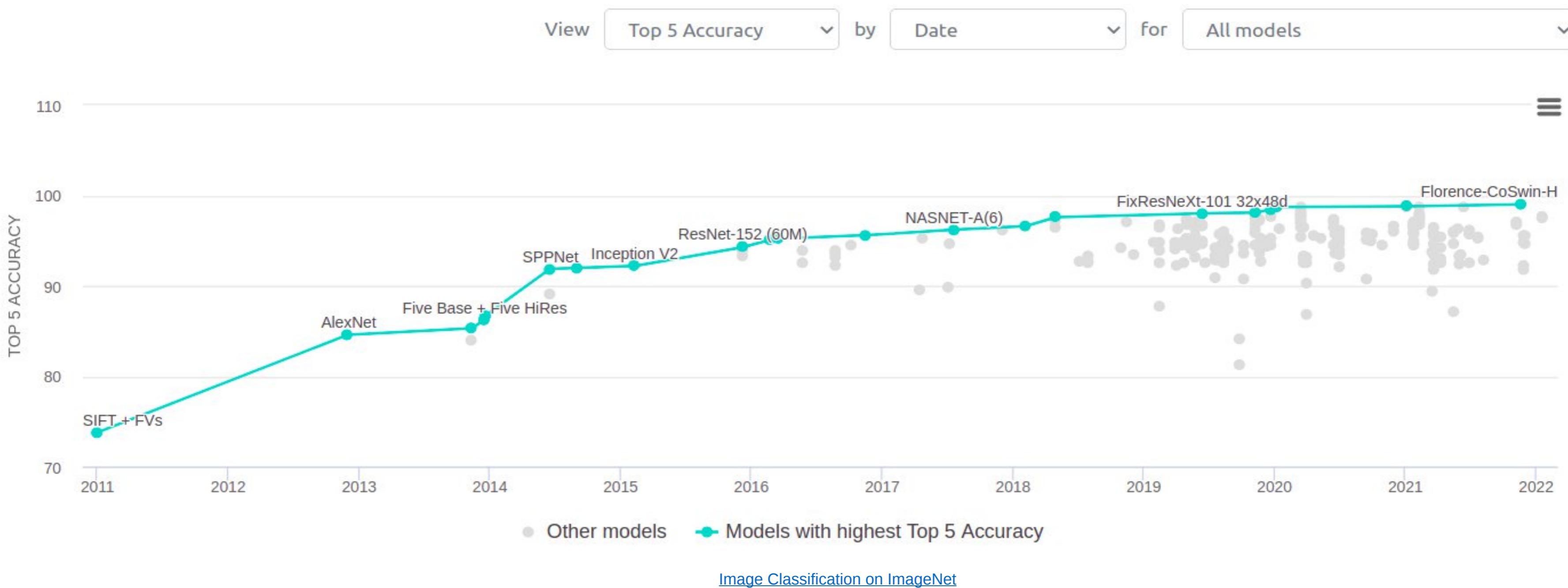


[Image Classification on ImageNet](#)



# Сравнение

Результаты соревнования по годам: top 5 accuracy





## Другие применения свёрток



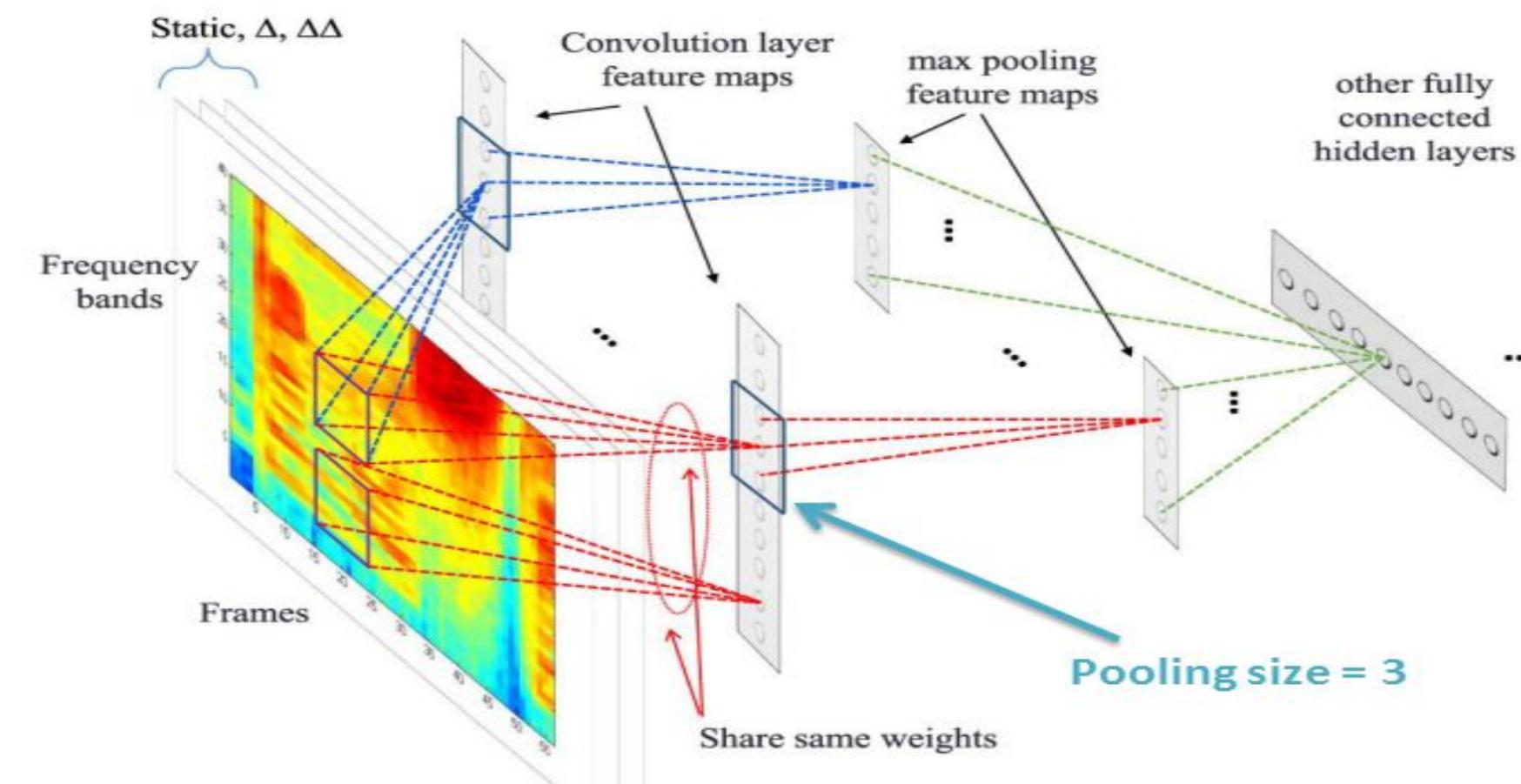
# Распознавание речи

На вход подается картинка размера, называемая **спектрограммой звука**.

Каждый канал соответствует картине распределения частот для промежутка времени.

Кол-во каналов - число таких промежутков времени.

С таким изображением можно работать как с обычным изображением и подать на вход свёрточной сети.

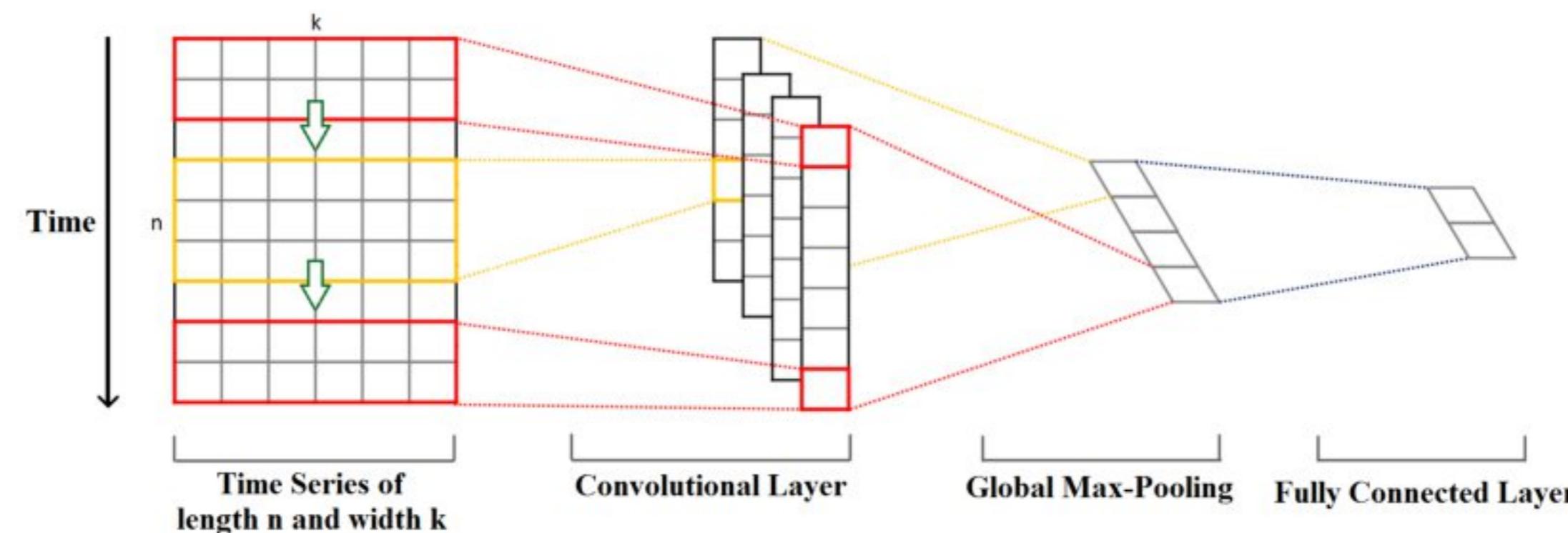




# Временные ряды

Используются 1-D свертки.

Таким образом объединяется информация за промежуток времени.





BIGE!