ТРАНСФОРМЕР И КОМПЬЮТЕРНОЕ ЗРЕНИЕ

Сергей Николенко

ИТМО — Санкт-Петербург

2 ноября 2026 г.





Random facts:

- 2 ноября 1721 г. Пётр I принял титул Императора Всероссийского, и Россия стала империей
- 2 ноября 1937 г. на башнях Московского Кремля впервые зажглись рубиновые звёзды
- 2 ноября 1959 г. вратарь «Монреаля» Жак Плант, получив шайбой в лицо и семь швов, вернулся на лёд в маске, которую сделал сам; теперь без маски не играет никто
- 2 ноября 1960 г. британский суд присяжных постановил, что роман Дэвида Лоуренса «Любовник леди Чаттерлей» не является непристойным, и Penguin смог издать его без купюр
- 2 ноября 1988 г. червь, запущенный студентом Робертом Моррисом, вывел из строя заметную часть тогдашнего интернета
- 2 ноября 2000 г. на МКС прибыл первый экипаж, и с тех пор люди живут на станции непрерывно
- 2 ноября 2016 г. «Чикаго Кабс» выиграли Мировую серию впервые за 108 лет и сняли «проклятие козла»

Краткое резюме предыдущей серии

- В прошлый раз мы прошли рекуррентные сети, LSTM и GRU, механизмы внимания поверх рекуррентных сетей и дошли до конструкции трансформера.
- Сегодня повторим самовнимание и соберём трансформер целиком, а потом посмотрим, каким он стал сегодня: *современный рецепт*, к которому сошлись почти все большие модели.
- Потом трансформеры придут в зрение: ViT, Swin и CLIP.
- А дальше — задачи компьютерного зрения, где ответ сложнее одной метки: сначала *распознавание объектов*, от двухэтапных детекторов до DETR.

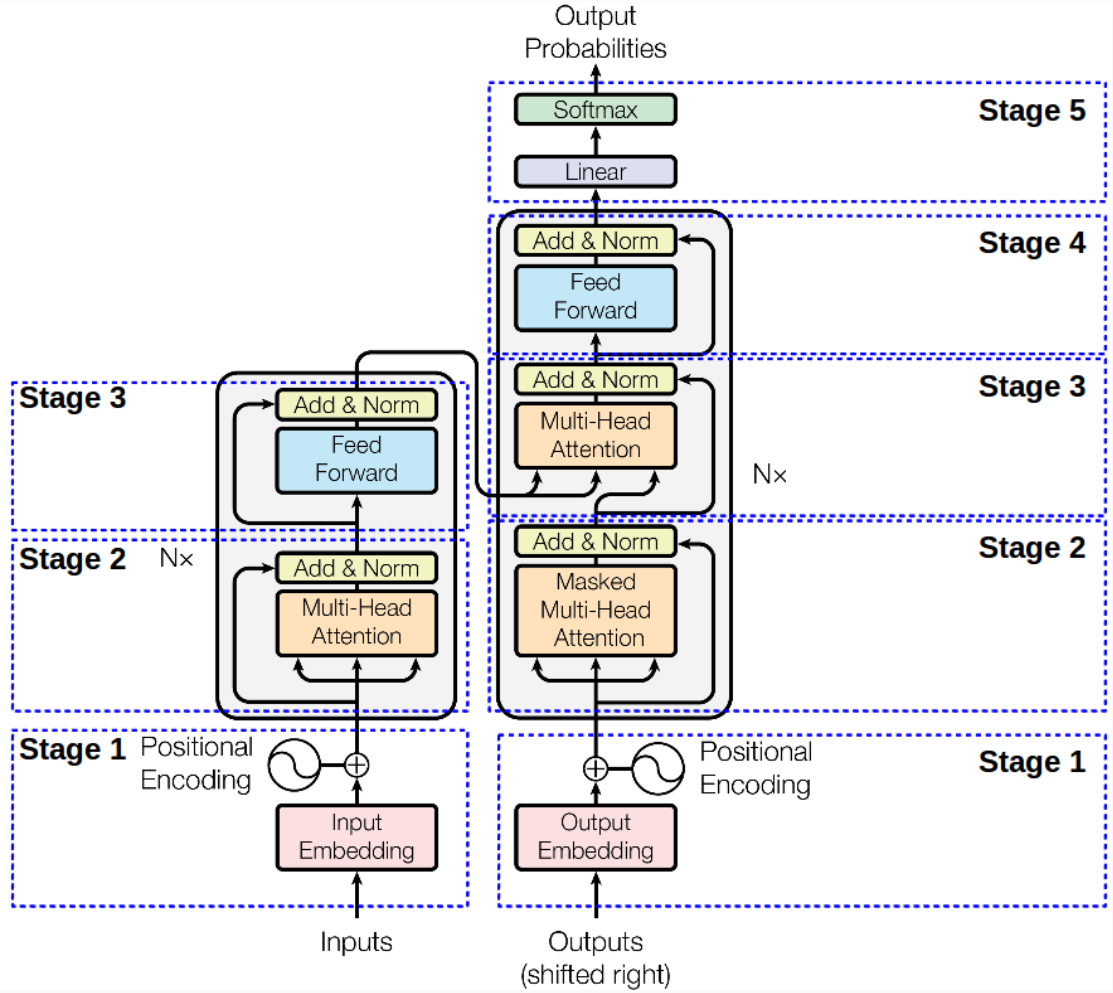
Самовнимание и Transformer

Transformer

- Мы изучали перевод на рекуррентных сетях и дошли до архитектуры Google NMT
- Но в 2017 году оказалось, что всё может быть ещё проще и интереснее
- Google: «Attention is all you need» (Vaswani et al., 2017)
- Основная идея – self-attention; оказывается очень плодотворной для всевозможных seq2seq задач
- Главная мотивация – попробовать всё-таки уйти от кодирования вектором постоянной длины

Transformer

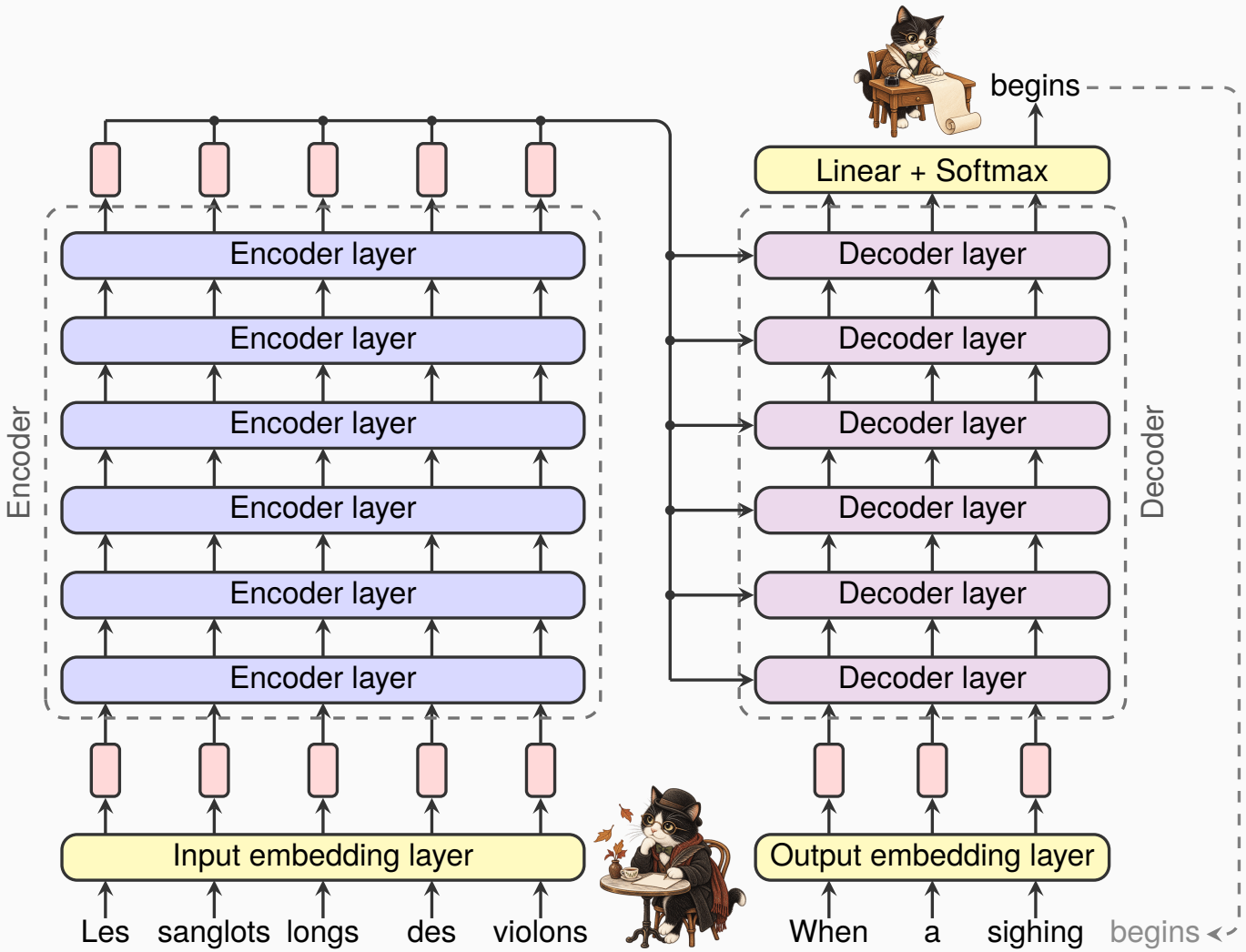
- Общая схема:



- Теперь подробнее...

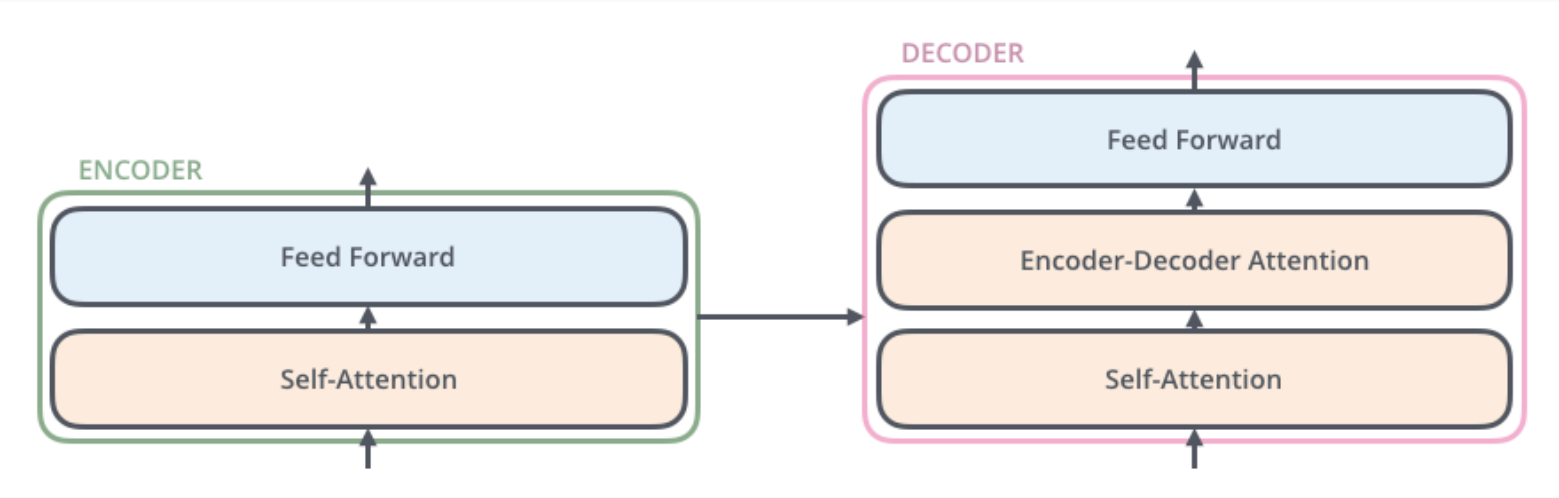
Transformer

- Суть, как и раньше, – encoder-decoder:



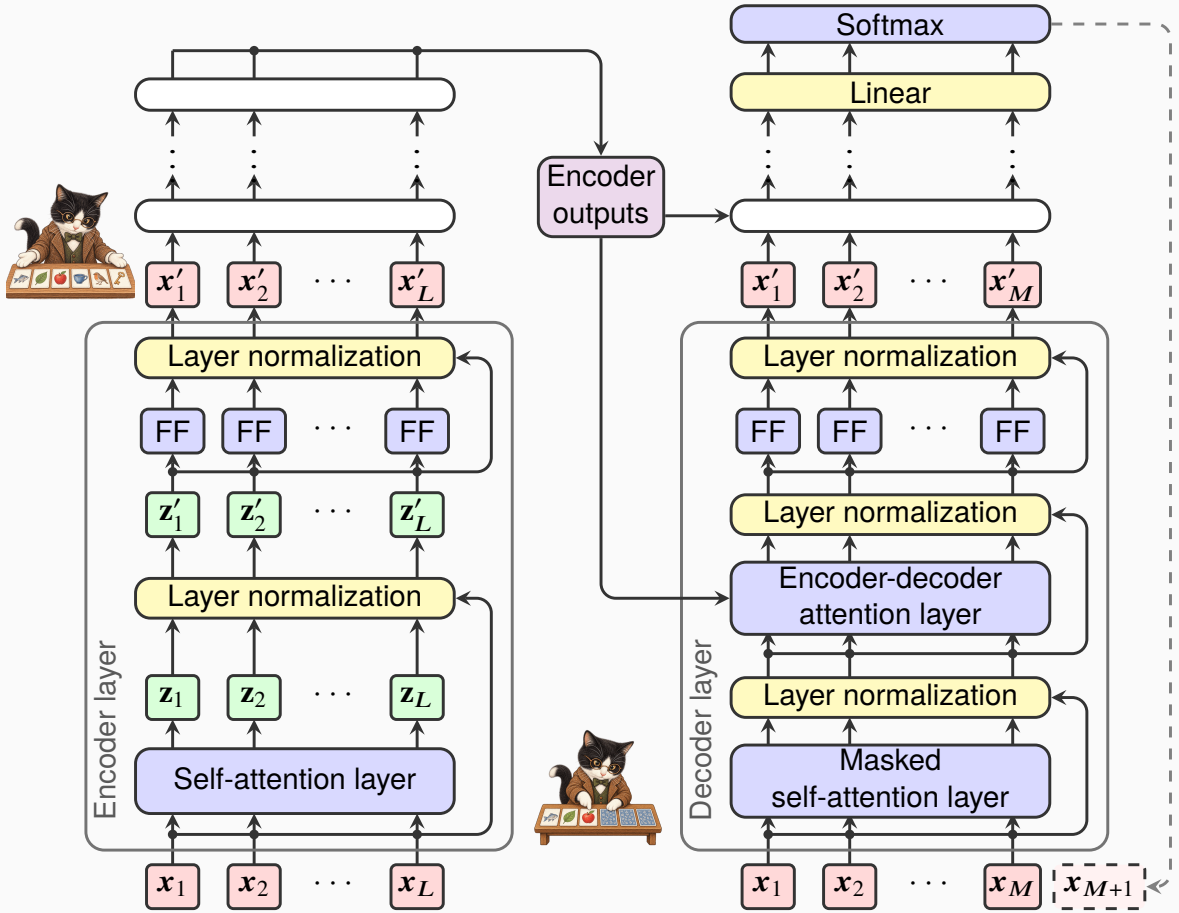
Transformer

- В каждом слое – слой self-attention, а потом feedforward layer, который независимо применяется к каждой позиции входа. У декодера ещё есть attention между ними:



Transformer

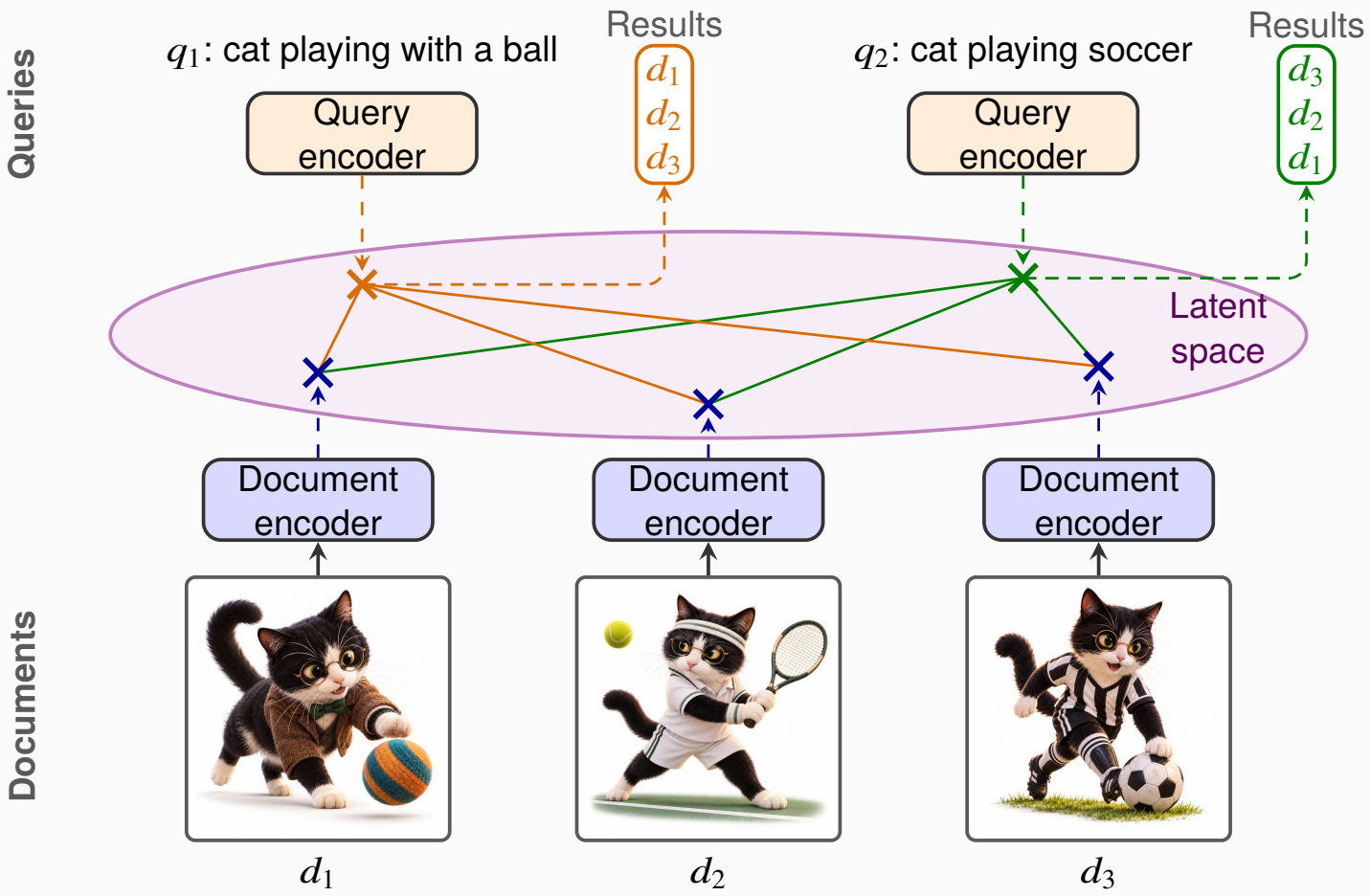
- Слова, естественно, представляются векторами, в feedforward слое всё параллельно:



- Но что же это такое – self-attention?

Self-attention

- Чтобы понять самовнимание (self-attention), начнём с информационного поиска (information retrieval)



Self-attention

- Запросы и документы отображаются в одно и то же латентное пространство (но разными кодировщиками — это могут быть вообще объекты разной природы)
- Запрос идёт через query encoder
- Документы (на иллюстрации это картинки) — через другой encoder, но в то же пространство
- Чтобы найти самые релевантные документы, мы ищем ближайших соседей в латентном пространстве среди документов; часто предполагают, что латентное пространство линейно, и расстояние там — это просто скалярное произведение $\mathrm{dist}(q, d) = \mathrm{Enc}_q(q)^\top \mathrm{Enc}_d(d)$.

Self-attention

- В самовнимании эта интуиция используется очень абстрактно. Слой получает на вход последовательность векторов $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_L$, т.е. матрицу $X \in \mathbb{R}^{d \times L}$.
- Запросы, ключи и документы берутся из самих $\mathbf{x}_i$:
  - умножая $\mathbf{x}_1$ на матрицу весов W^Q, мы получаем вектор *запроса* $\mathbf{q}_1 = W^Q \mathbf{x}_1$; обратите внимание, что размерность q векторов запросов, $\mathbf{q}_i \in \mathbb{R}^q$, может отличаться (обычно меньше) от входной размерности d, $\mathbf{x}_i \in \mathbb{R}^d$;
  - умножая $\mathbf{x}_i$ на матрицу весов W^K, мы получаем векторы *ключей* $\mathbf{k}_i = W^K \mathbf{x}_i$ для $i = 1, \dots, L$; поскольку мы хотим, чтобы запросы и ключи находились в одном латентном пространстве, ключи имеют ту же размерность, что и запросы, $\mathbf{k}_i \in \mathbb{R}^q$, таким образом $W^K \in \mathbb{R}^{q \times d}$;
  - третья матрица весов W^V даёт векторы *значений* $\mathbf{v}_i = W^V \mathbf{x}_i$ для $i = 1, \dots, L$; это те *документы*, которые мы будем «извлекать» с помощью ключей $\mathbf{k}_i$; формально мы имеем другую размерность v для значений, $\mathbf{v}_i \in \mathbb{R}^v$ и $W^V \in \mathbb{R}^{v \times d}$; на практике обычно $v = q$.

Self-attention

- Затем мы выполняем поиск, вычисляя оценки внимания как скалярные произведения между запросами и документами
- Нормализуем $\mathbf{q}_i^\top \mathbf{k}_j$, деля на $\sqrt{q}$, и получаем оценки через softmax, чтобы преобразовать их в вероятности:

$$\alpha_{ij} = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right)_j,$$

где $K \in \mathbb{R}^{q \times L}$ — это все ключи, объединенные в матрицу, $K = W^K X$.

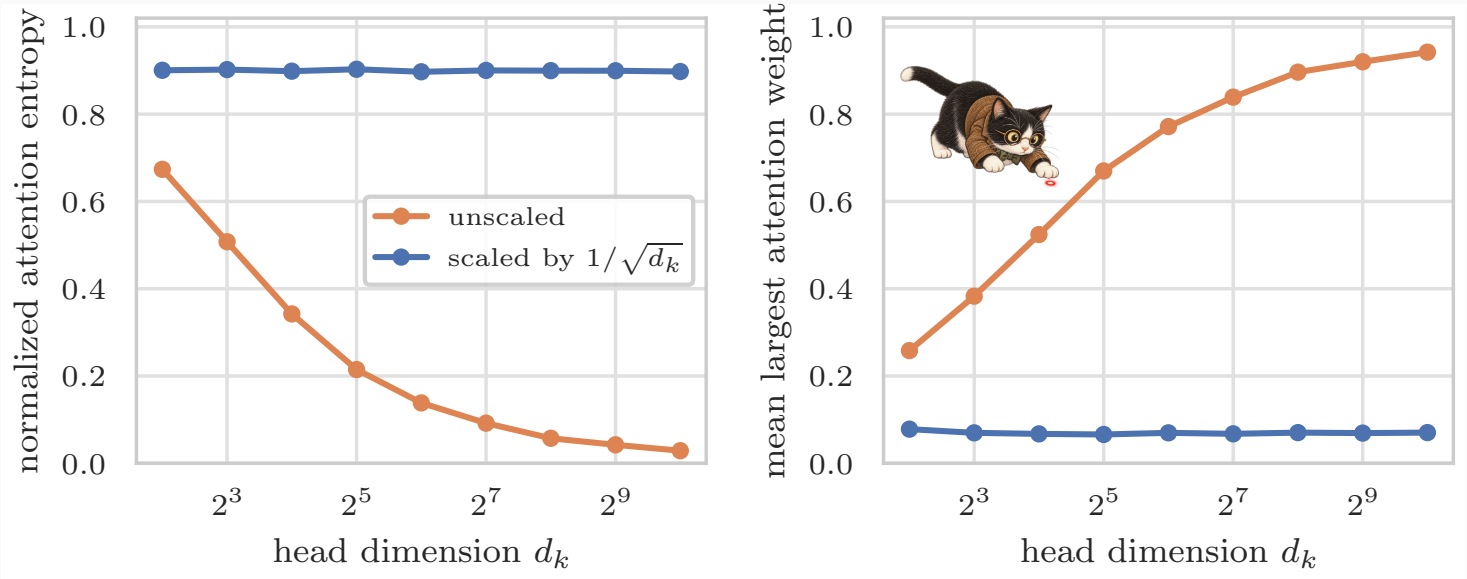
- Затем используем результат в качестве коэффициентов для выпуклой комбинации *значений* $\mathbf{v}_j$:

$$\mathbf{z}_i = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right) V,$$

где $V \in \mathbb{R}^{q \times L}$ — значения (values), $V = W^V X$.

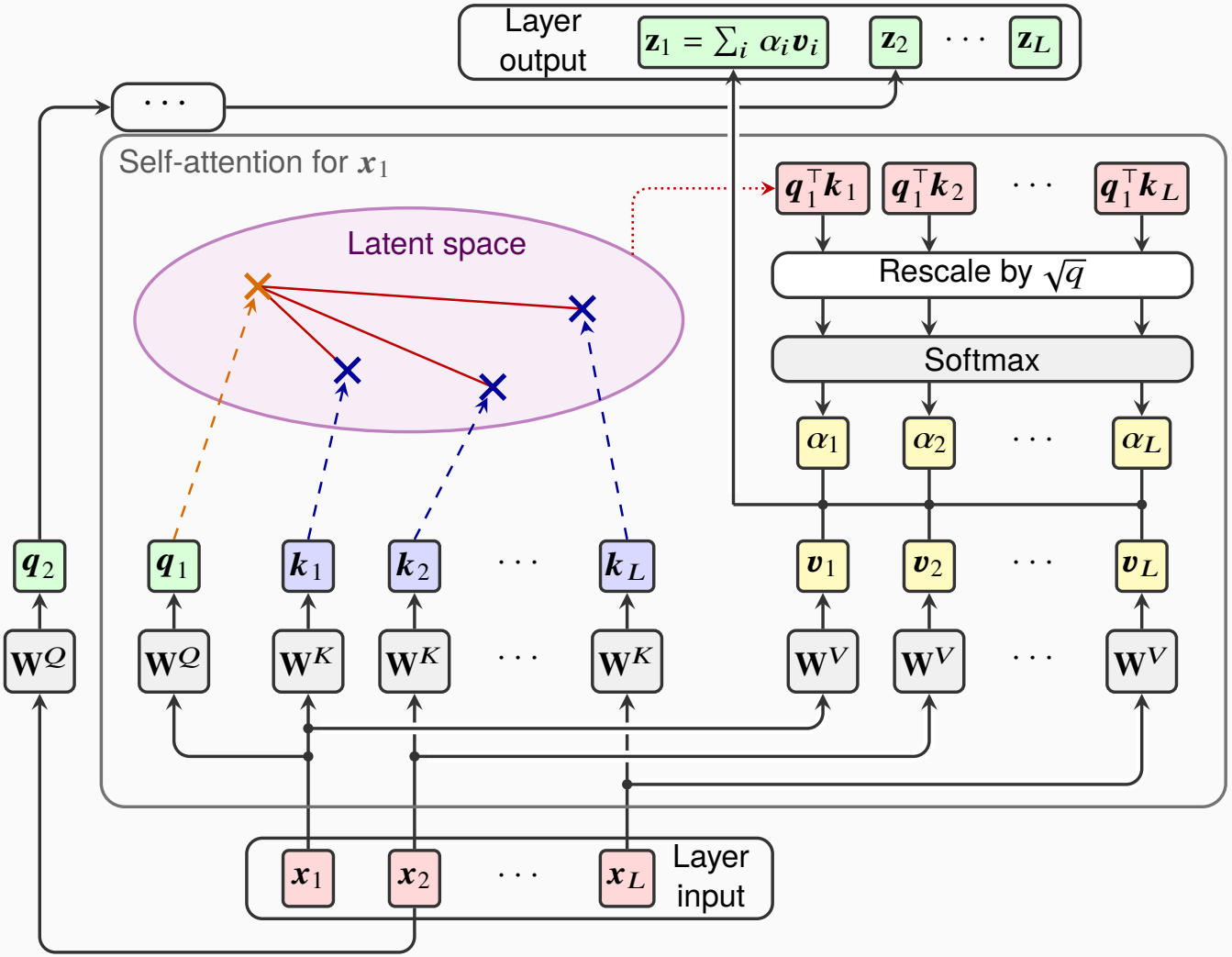
Self-attention

- Зачем делить на $\sqrt{q}$? Скалярное произведение $\mathbf{q}_i^\top \mathbf{k}_j = \sum_{r=1}^{q} q_{ir} k_{jr}$ — сумма q примерно независимых слагаемых с дисперсией около 1, так что оно порядка $\sqrt{q}$.
- Без нормировки softmax насыщается, и внимание превращается в почти детерминированный поиск; слева — нормированная энтропия весов, справа — самый большой вес:



Self-attention

- Общая структура слоя самовнимания (self-attention):



Self-attention

- И это вся интуиция! Мы можем объединить вычисления каждого $\mathbf{z}_i$ в одну формулу в матричном виде:

$$Z = \text{softmax}\left(\frac{1}{\sqrt{q}} QK^\top\right) V.$$

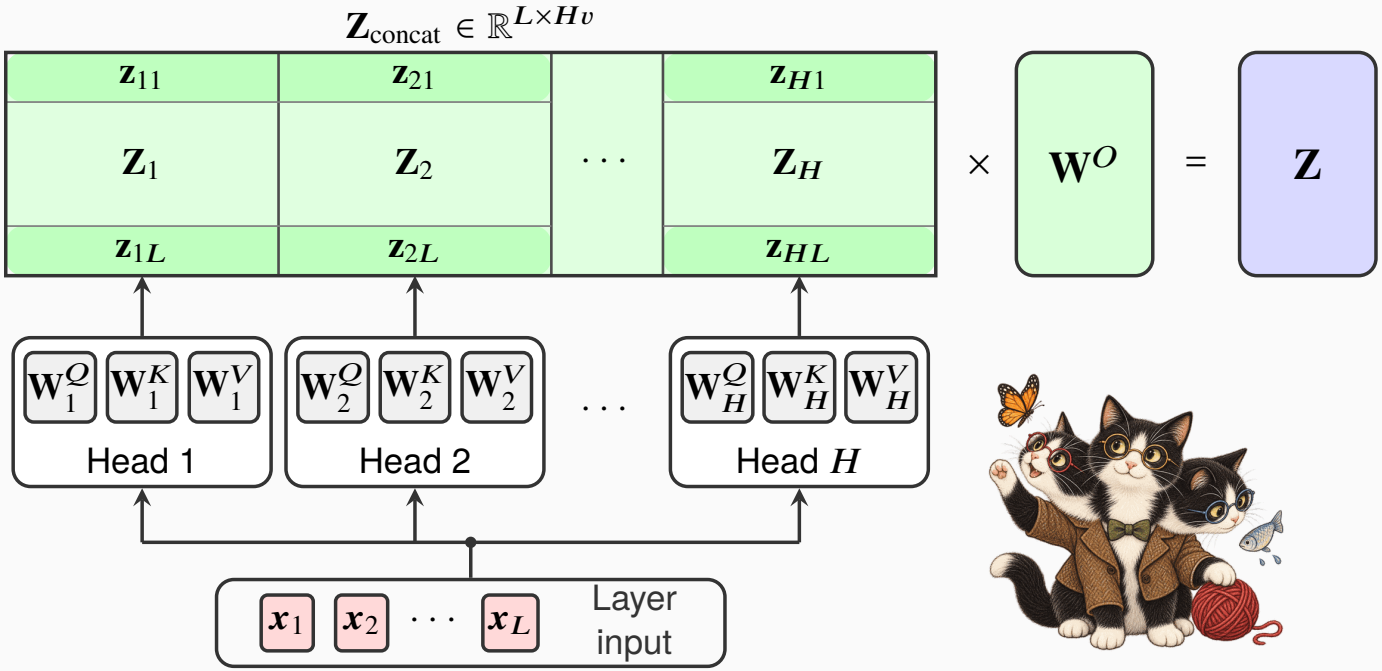
- Но это только один способ «смотреть» на входные векторы; то, что мы сейчас определили, — это не полный слой самовнимания, а только одна *голова* (self-attention head)
- Можно распараллелить эти вычисления вдоль H различных голов, используя разные матрицы весов $W_1^Q, \dots W_H^Q$, $W_1^K, \dots W_H^K$ и $W_1^V, \dots W_H^V$. Это *многоголовое внимание* (multi-head attention)

Self-attention

- После H параллельных голов мы получаем матрицы выходов $Z_1, Z_2, \ldots, Z_H$ размерности $v \times L$, объединяем их все в $Z_{\text{concat}} \in \mathbb{R}^{L \times Hv}$ и добавляем ещё одну матрицу весов $W^O \in \mathbb{R}^{Hv \times d}$, чтобы сжать результат обратно до необходимой размерности:
$$Z = \left(Z_{\text{concat}} W^O\right)^\top.$$
- С помощью W^O слой самовнимания может комбинировать представления, полученные из различных голов внимания; это также важный способ добавить гибкость и выразительность в архитектуру.

Self-attention

- Многоголовое внимание:



- Это самая важная часть, но это ещё не всё.

Два вида смешивания: токены и каналы

- Блок трансформера удобно понимать как чередование двух операций:
  - *внимание смешивает токены*: каждая позиция собирает информацию из других;
  - *FFN смешивает каналы*: каждая позиция обрабатывается отдельно, $d \to 4d \to d$.
- С нормализацией N и остаточными связями:

$$A = X + \mathrm{MHA}(N(X)), \qquad Y = A + \mathrm{FFN}(N(A)).$$

- Одного внимания мало: без остаточных связей и FFN стопка слоёв самовнимания вырождается в матрицу ранга 1, причём дважды экспоненциально быстро (Dong et al., 2021).

Два вида смешивания: токены и каналы

- Где параметры? При $d_{\text{ff}} = 4d$ у FFN около $8d^2$ весов, у W^Q, W^K, W^V, W^O — около $4d^2$; большая часть модели — в FFN.
- Где вычисления? Проекции — $4Ld^2$, $QK^\top$ и умножение на V — $2L^2d$, FFN — $8Ld^2$; квадратичная часть доминирует, только когда L в несколько раз больше d.
- А вот с памятью хуже: наивная реализация хранит HL^2 весов внимания (к этому мы ещё вернёмся).
- Удобный взгляд — *residual stream*: каждый слой читает из общего потока и пишет в него. Голова внимания — маленькая ассоциативная память: запросы и ключи решают, *откуда* читать, значения — *что* скопировать.

Декодер

- Декодер включает два дополнительных типа слоёв: *маскированное самовнимание* (masked self-attention) и *encoder-decoder attention*
- Маскированное внимание в основном означает, что, поскольку декодер работает авторегрессионно, он не должен смотреть на токены, которые ещё не породил
- Для этого проще всего ввести всю последовательность и замаскировать будущие позиции внутри слоёв самовнимания
- Формально это означает, что мы задаем аргументы softmax как $-\infty$ для будущих токенов, что означает, что их веса внимания всегда будут нулевыми

Декодер

- Encoder-decoder attention — это вариация механизма самовнимания, которая опирается на результаты кодировщика
- Звучит сложно... но на самом деле это почти тривиальное изменение! Мы используем разные векторы в качестве входных данных в том же самовнимании:
  - для создания запросов используем векторы из предыдущего слоя, то есть текущие представления уже порождённых выходных токенов;
  - но для «документов» в задаче «поиска», то есть для векторов ключей и значений, мы используем векторы из энкодера
- Неформально это значит, что мы выполняем «поиск» на выходе энкодера с запросами, состоящими из уже порождённых токенов
- Формально все размерности хорошо совпадают: в аргументе softmax есть L элементов для каждого вектора, но число запросов и, следовательно, число выходов совпадает с числом входов

Декодер

- Обучается декодер параллельно, хотя модель авторегрессионная: подаём сдвинутую последовательность $\langle \mathrm{BOS} \rangle, y_1, \ldots, y_{T-1}$ и предсказываем $y_1, \ldots, y_T$ сразу во всех позициях (teacher forcing).
- Маскированное внимание — одна треугольная матрица:

$$\mathrm{Attn}_{\mathrm{causal}}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{q}} + M\right) V,$$

$$M_{ij} = 0 \text{ при } j \le i, \qquad M_{ij} = -\infty \text{ при } j > i.$$

- А порождение — честно по одному токену; ключи и значения префикса хранятся в *KV-кэше*, чтобы не пересчитывать их на каждом шаге.
- Тонкость: при обучении префикс написан человеком, а при порождении — самой моделью, и ранняя ошибка остаётся в контексте (exposure bias).

Токенизация

- Мы используем вложения (embeddings), но как мы делим текст на токены?
- *Byte-pair encoding* (BPE): старая идея из сжатия данных (Gage, 1994), которую для перевода приспособили Sennrich et al. (2016): жадно склеиваем самую частую пару символов
- Давайте начнем с построения словаря:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\}.$$

Original words:	cat		pet		mat		rat		eats
Frequencies:	10		12		5		8		4
Characters:	c	a	t	p	e	m	r	s	
Frequencies:	10	27	39	12	16	5	8	4	
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts	
Frequencies:	10	**27**	12	12	5	8	4	4	



Токенизация

- Затем разделим его на символы и подсчитаем пары символов:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\},$$

$$\{c : 10, a : 27, t : 39, p : 12, e : 16, m : 5, r : 8, s : 4\},$$

$$\{ca : 10, at : 27, pe : 12, et : 12, ma : 5, ra : 8, ea : 4, ts : 4\}.$$

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4



Токенизация

- Теперь возьмем наиболее частую пару («*at*») и перекодируем её в новый символ (Z), который добавляется в словарь:

$$\{\mathrm{cZ} : 10, \mathrm{pet} : 12, \mathrm{mZ} : 5, \mathrm{rZ} : 8, \mathrm{eZs} : 4\},$$
$$\{c : 10, Z : 27, t : 12, p : 12, e : 16, m : 5, r : 8, s : 4\},$$
$$\{cZ : 10, pe : 12, et : 12, mZ : 5, rZ : 8, eZ : 4, Zs : 4\}.$$

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4



Токенизация

- Можно теперь выбрать самую частую пару — *pe* или *et* — и заменить её другим новым символом *Y*:

$$\{\mathrm{cZ}:10, \mathrm{Yt}:12, \mathrm{mZ}:5, \mathrm{rZ}:8, \mathrm{eZs}:4\},$$
$$\{c:10, Z:27, Y:12, t:12, e:4, m:5, r:8, s:4\},$$
$$\{cZ:10, Yt:12, mZ:5, rZ:8, eZ:4, Zs:4\}.$$

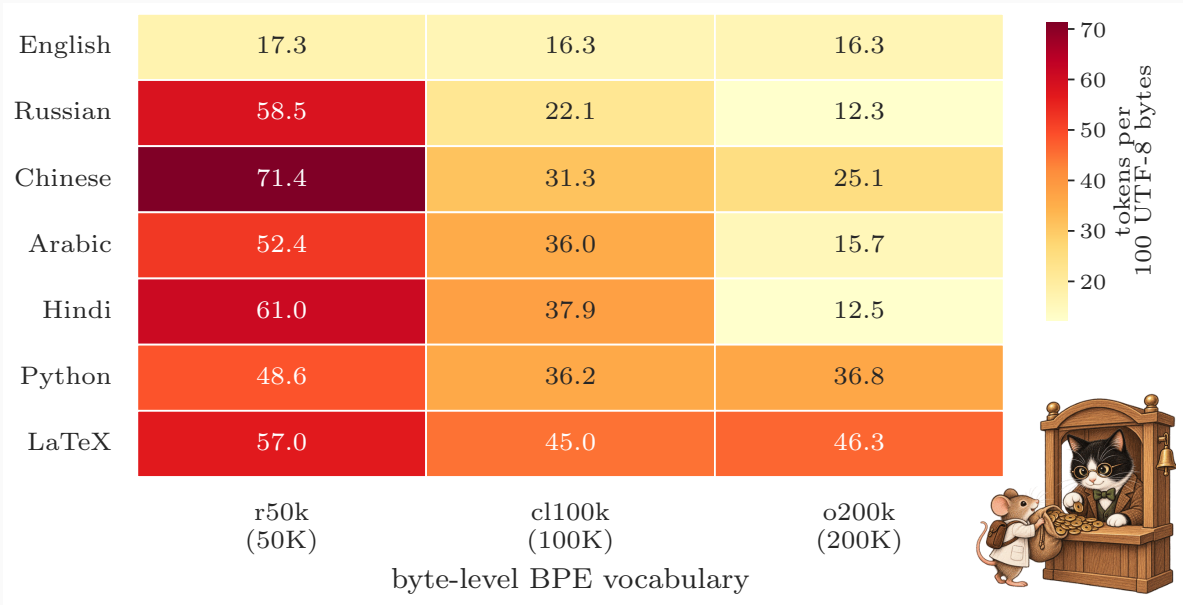
- И так далее, пока не достигнем требуемого размера словаря

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4



Токенизация

- Варианты: byte-level BPE (GPT-2), WordPiece (BERT), Unigram, SentencePiece.
- Токенизатор — часть архитектуры: чем больше словарь, тем короче последовательности, но больше таблица вложений и softmax на выходе.
- И налог на разные языки неравномерный; токенов на 100 байт UTF-8 в трёх поколениях BPE из tiktoken:



Positional encodings

- В трансформере каждый входной токен может “посмотреть” на любой другой *напрямую*, через любое число других токенов
- Это большое преимущество по сравнению с RNN!
- Но это значит, что у нас фиксированное ограниченное *context window* и, главное, поскольку веса внимания покрывают все токены равномерно, мы *теряем последовательность*: слой self-attention “не знает”, что некоторые токены стоят рядом, а некоторые далеко друг от друга
- Нужно как-то дать понять трансформеру, что у последовательности есть порядок, и это делается через *positional encodings*
- Формально: $\mathrm{Attn}(PX) = P\,\mathrm{Attn}(X)$, вход для самовнимания — множество

Positional encodings

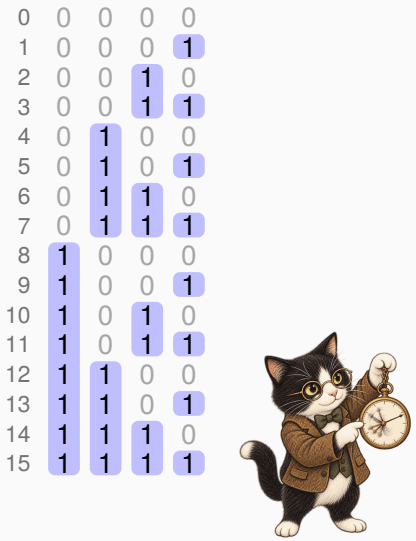
- Positional encodings — векторы, добавляемые ко вложениям, которые отражают порядковый номер токена
- Как его закодировать? Можно просто число увеличивать, но это не так просто:
  - если использовать увеличивающуюся последовательность вроде $1, 2, 3, \ldots$, она не обобщится на последовательности длиннее, чем часто бывало в датасете, да и вообще поведение будет плохо определено для больших значений;
  - если уложить в интервал вроде $[0, 1]$, то positional encoding не будет знать, сколько на самом деле слов между токенами: расстояние от 0 до $\frac{1}{2}$ может быть 1 токен или 100.

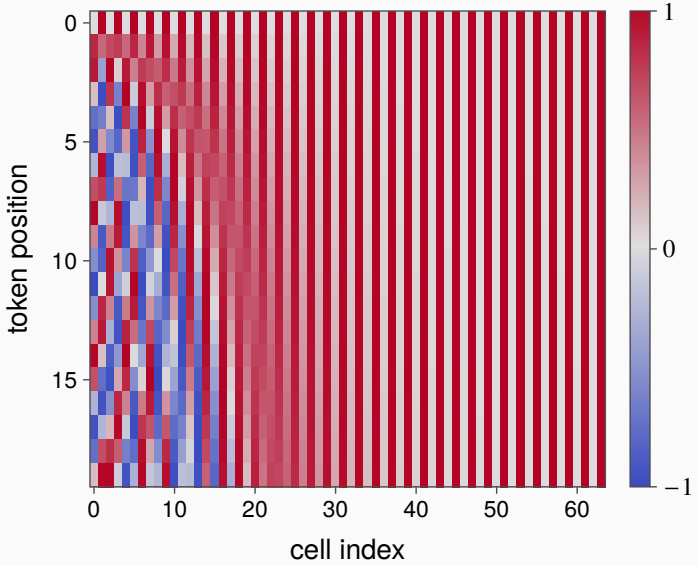
Positional encodings

- Так что мы используем нечто вроде позиционной системы счисления, только непрерывную:

$$\mathrm{PE}\left(\mathrm{pos}, 2i\right) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right), \quad \mathrm{PE}\left(\mathrm{pos}, 2i+1\right) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right),$$

где pos — позиция токена, d — размерность вложения.

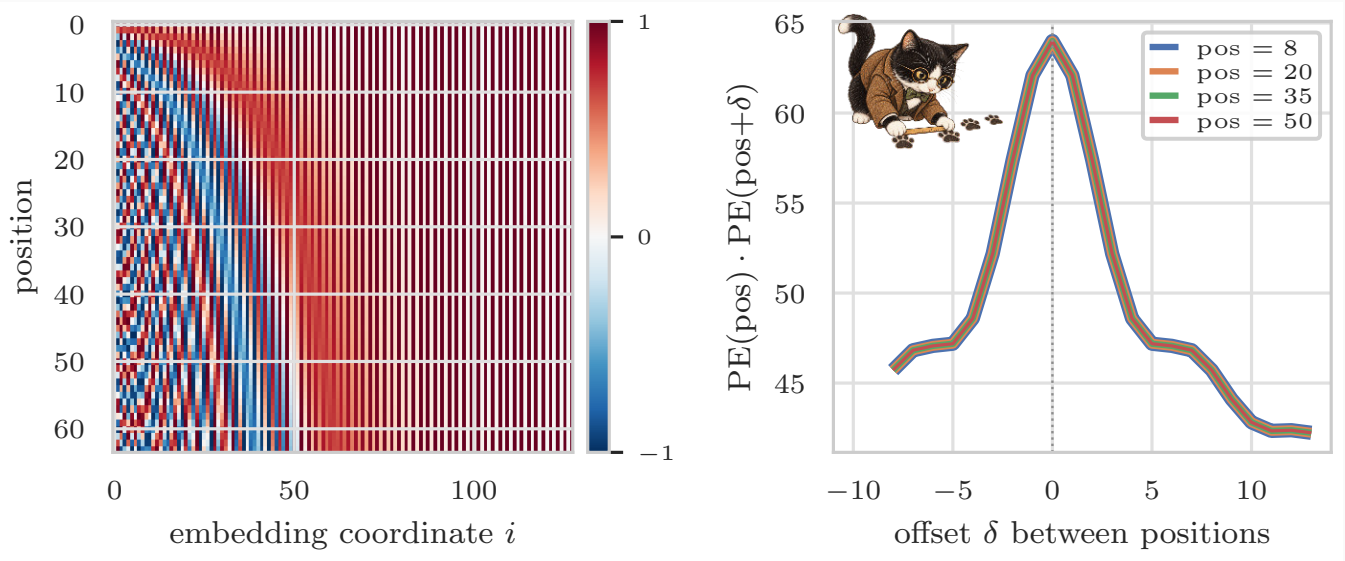




Positional encodings

- Главное свойство: $\mathrm{PE}(p)^\top \mathrm{PE}(p+\delta)$ зависит только от сдвига δ, ведь

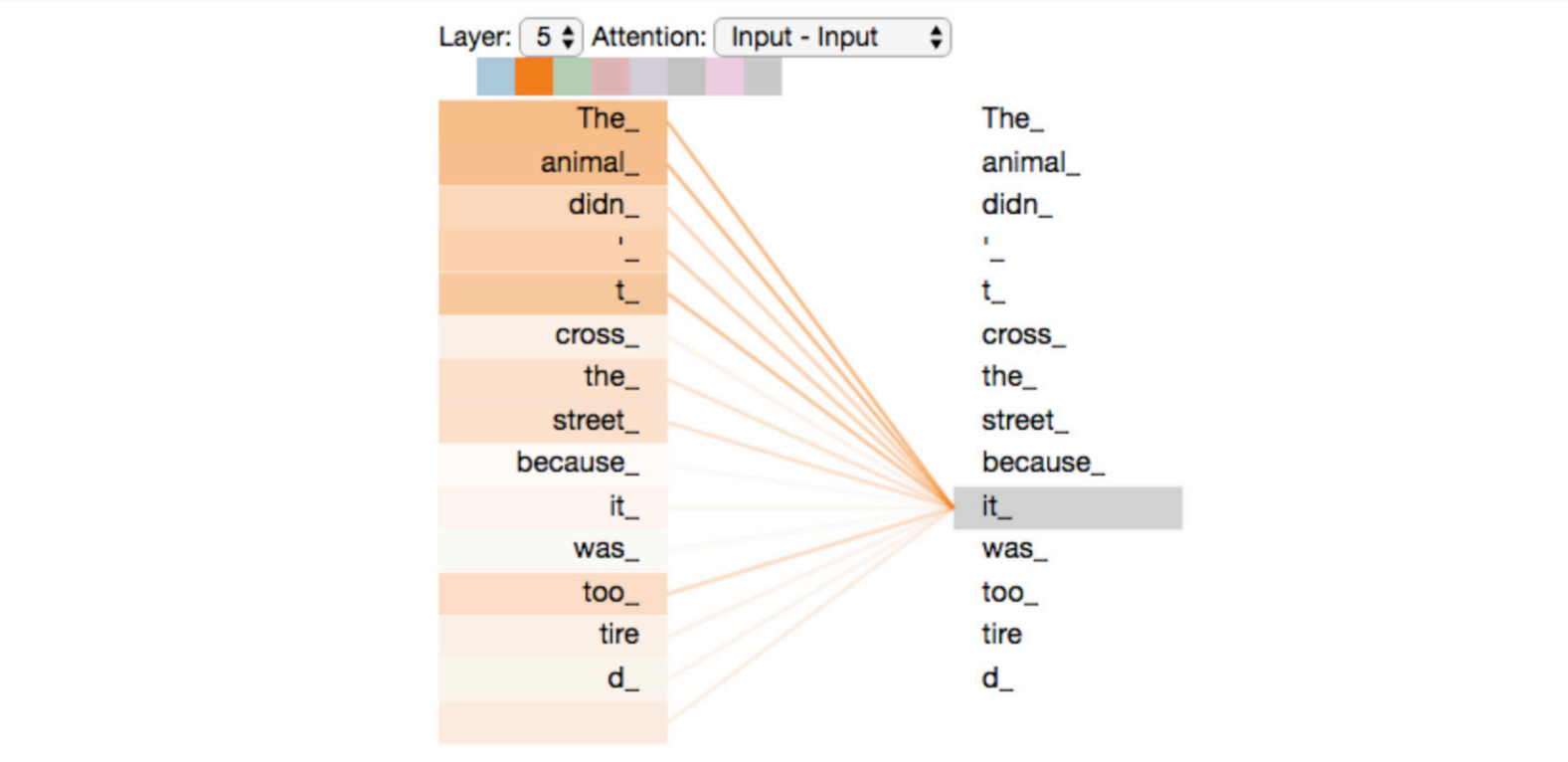
$$\sin(p\omega_i)\sin((p+\delta)\omega_i) + \cos(p\omega_i)\cos((p+\delta)\omega_i) = \cos(\delta\omega_i).$$



- Позиционные кодировки не конкатенируют, а *складывают* с вложениями токенов; выученные позиционные векторы работают почти так же (Vaswani et al., 2017).

Результаты

- В результате трансформер обучает веса того, как одни слова ”участвуют в обработке” других слов:



Результаты

- Бонус от self-attention – во-первых, вычислительный, во-вторых, сокращает пути между словами, в-третьих, потенциальная интерпретируемость:

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Результаты

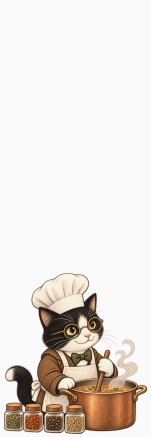
- Работает лучше, обучается в сто раз быстрее:

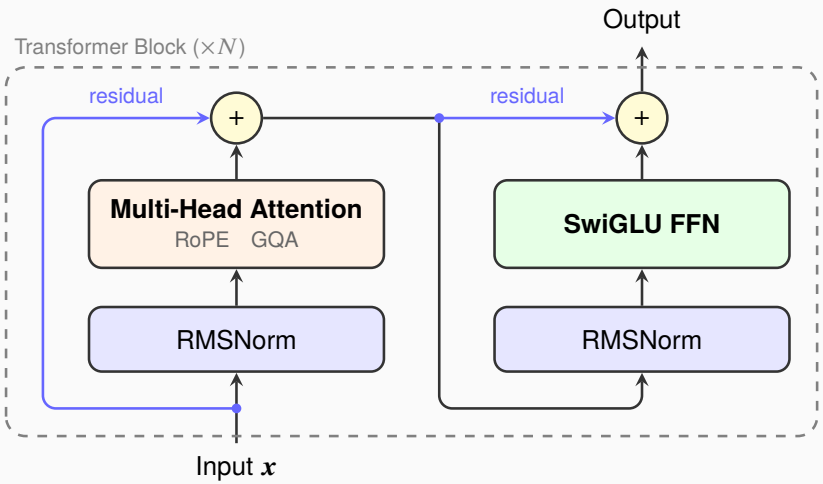
Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Современный рецепт трансформера

Что изменилось с 2017 года

- Изменений к исходному трансформеру предлагали десятки, и большинство из них стабильно не помогает (Narang et al., 2021).
- Но LLaMA, Mistral, PaLM, GPT-NeoX, Pythia и другие пришли примерно к одним и тем же ответам:
  - как кодировать позицию — RoPE;
  - где и как нормализовать — Pre-LN с RMSNorm;
  - каким сделать FFN — SwiGLU;
  - сколько хранить голов ключей и значений — GQA;
  - как считать внимание, не переполняя память — FlashAttention.





Rotary position embeddings (RoPE)

- Синусоиды и выученные позиционные векторы кодируют *абсолютную* позицию; а вниманию на самом деле нужна *относительная*.
- RoPE (Su et al., 2021): не прибавляем позицию ко входу, а *поворачиваем* запросы и ключи:

$$\mathbf{q}'_m = R_m \mathbf{q}, \qquad \mathbf{k}'_n = R_n \mathbf{k},$$

$$R_m = \begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & & & \\ \sin m\theta_0 & \cos m\theta_0 & & & \\ & & \cos m\theta_1 & -\sin m\theta_1 & \\ & & \sin m\theta_1 & \cos m\theta_1 & \\ & & & & \ddots \end{pmatrix},$$

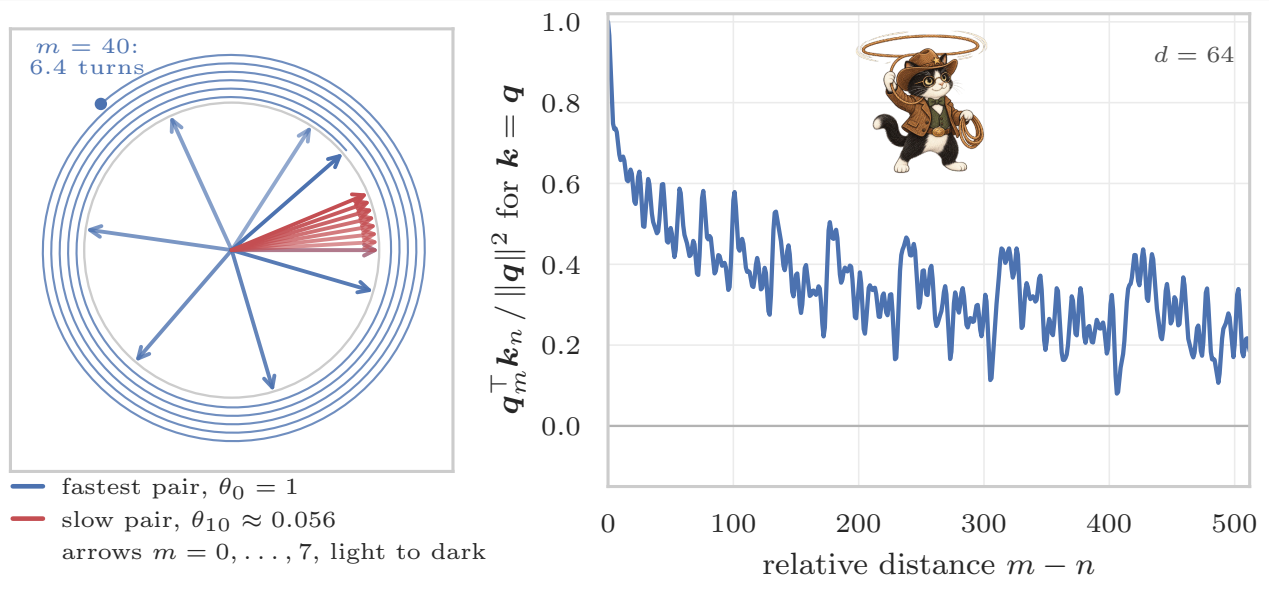
где $\theta_i = 10000^{-2i/d_h}$ — те же частоты, что у синусоид: пара координат i поворачивается на угол $m\theta_i$.

Rotary position embeddings (RoPE)

- Теперь в скалярном произведении

$$\mathbf{q}_m'^{\top}\mathbf{k}_n' = \mathbf{q}^{\top}R_m^{\top}R_n\mathbf{k} = \mathbf{q}^{\top}R_{n-m}\mathbf{k}$$

  остаётся только *относительная* позиция $n - m$; и никаких новых параметров.
- Быстрые пары координат делают много оборотов за предложение, медленные почти не двигаются:



Rotary position embeddings (RoPE)

- Беда та же, что у синусоид: обычный RoPE плохо экстраполирует на длины больше, чем при обучении.
- Отсюда целая индустрия трюков: position interpolation, NTK-aware scaling, YaRN — растягиваем или переинтерполируем частоты.
- Главный гиперпараметр — основание: 10000 досталось от синусоид, а в длинноконтекстных моделях его поднимают; у Llama 3 оно 500000.
- Альтернатива попроще — ALiBi (Press et al., 2022): просто штраф, линейный по расстоянию, прямо в оценки внимания (BLOOM).
- Но в современных LLM по умолчанию — RoPE.

Нормализация: RMSNorm и Pre-LN

- LayerNorm вычитает среднее и делит на стандартное отклонение; RMSNorm (Zhang, Sennrich, 2019) не вычитает среднее:

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \odot \gamma, \qquad \mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}.$$

- Дешевле и работает примерно так же.
- Важнее, *где* нормализовать. В исходном трансформере — *после* подслоя (Post-LN): $\mathbf{x} + \mathrm{Attn}(\mathbf{x})$, потом нормализация.
- Xiong et al. (2020): у Post-LN при инициализации большие градиенты у выхода, отсюда и нужда в аккуратном warmup.
- Pre-LN, нормализация *перед* подслоем, $\mathbf{x} + \mathrm{Attn}(N(\mathbf{x}))$: градиенты хорошие с самого начала, warmup почти не нужен.

Нормализация: RMSNorm и Pre-LN

- С Pre-LN выход подслоя прибавляется к остаточному потоку без нормализации, и в очень глубоких моделях поток может разрастись так, что поздние слои его почти не меняют.
- Поэтому многие нормализуют ещё и *выход* подслоя: Swin v2, Gemma 2 и 3 (RMSNorm и до, и после), OLMo 2 (только после).
- А можно вообще без нормализации: Zhu et al. (2025) заменили её на поэлементный

$$\mathrm{DyT}(\mathbf{x}) = \gamma \odot \tanh(\alpha \mathbf{x}) + \beta$$

  и получили те же результаты на зрении и языке.
- Так что где нормализовать и нормализовать ли вообще — вопрос до сих пор открытый.

SwiGLU

- FFN в исходном трансформере:
  $\mathrm{FFN}(\mathbf{x}) = \mathrm{ReLU}(\mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$.
- SwiGLU (Shazeer, 2020) — FFN с *гейтом*:

$$\mathrm{SwiGLU}(\mathbf{x}) = \left(\mathrm{Swish}(\mathbf{x}W_1) \odot (\mathbf{x}W_3)\right) W_2, \qquad \mathrm{Swish}(x) = x \cdot \sigma(x).$$

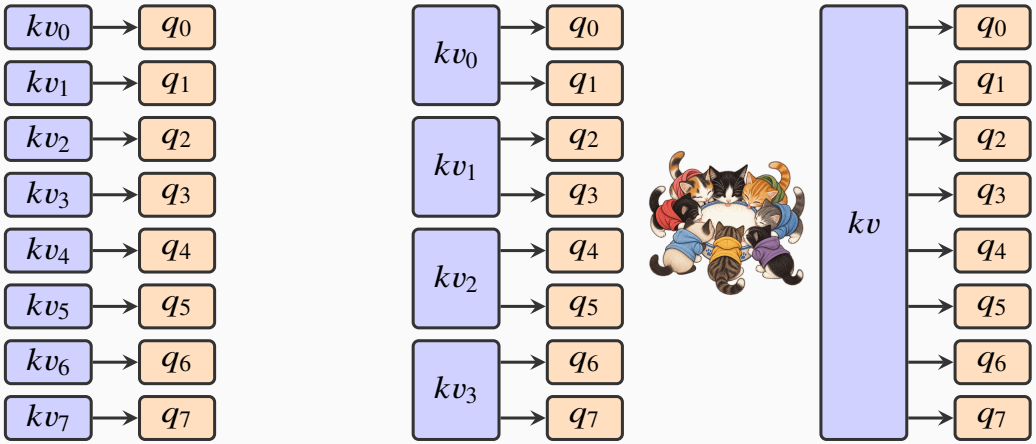
- Ветка $\mathbf{x}W_3$ работает как гейт, модулирующий сигнал из ветки со Swish; это gated linear unit, тот же принцип, что в LSTM.
- Почему работает? Статья честно объясняет успех «as all else, to divine benevolence».
- Лишняя матрица W_3 окупается сужением: у ReLU-FFN ширины d_{ff} примерно $2dd_{\text{ff}}$ весов, у SwiGLU ширины m — $3dm$, так что берут $m \approx \frac{2}{3}d_{\text{ff}}$.
- Теперь это стандарт: PaLM, LLaMA, Mistral, почти все современные LLM (у Gemma — GEGLU).

KV-кэш и grouped-query attention

- При развёртывании главное узкое место — не арифметика, а *пропускная способность памяти*.
- Два режима вывода:
  - *prefill*: весь промпт за один проход, упираемся в вычисления, $O(L^2 d)$ один раз;
  - *decode*: по одному токену, и на каждом шаге надо прочитать из памяти все веса и весь KV-кэш.
- KV-кэш — это $2\, n_{\text{layers}} L H_{kv} d_h$ чисел на последовательность.
- Llama 2 7B: 32 слоя, 32 головы, $d_h = 128$, т.е. $2 \cdot 32 \cdot 32 \cdot 128 \approx 262$ тысячи чисел, полмегабайта на *каждый токен*.
- Одна последовательность из 4096 токенов — 2 ГБ кэша, батч из $16 - 32$ ГБ, больше чем вдвое больше самих весов (13.5 ГБ).

KV-кэш и grouped-query attention

- Multi-query attention (MQA, Shazeer, 2019): одна голова ключей и значений на все головы запросов; кэш меньше в H раз, а качество теряется не так уж сильно.
- Grouped-query attention (GQA, Ainslie et al., 2023) — середина: $1 < H_{kv} < H_q$, одна голова ключей и значений на H_q / H_{kv} голов запросов.



- MHA можно «доучить» в GQA за $\approx 5\%$ исходного обучения; GQA в Llama 2 70B, во всей Llama 3 (восемь KV-голов), Mistral, Mixtral...

KV-кэш и grouped-query attention

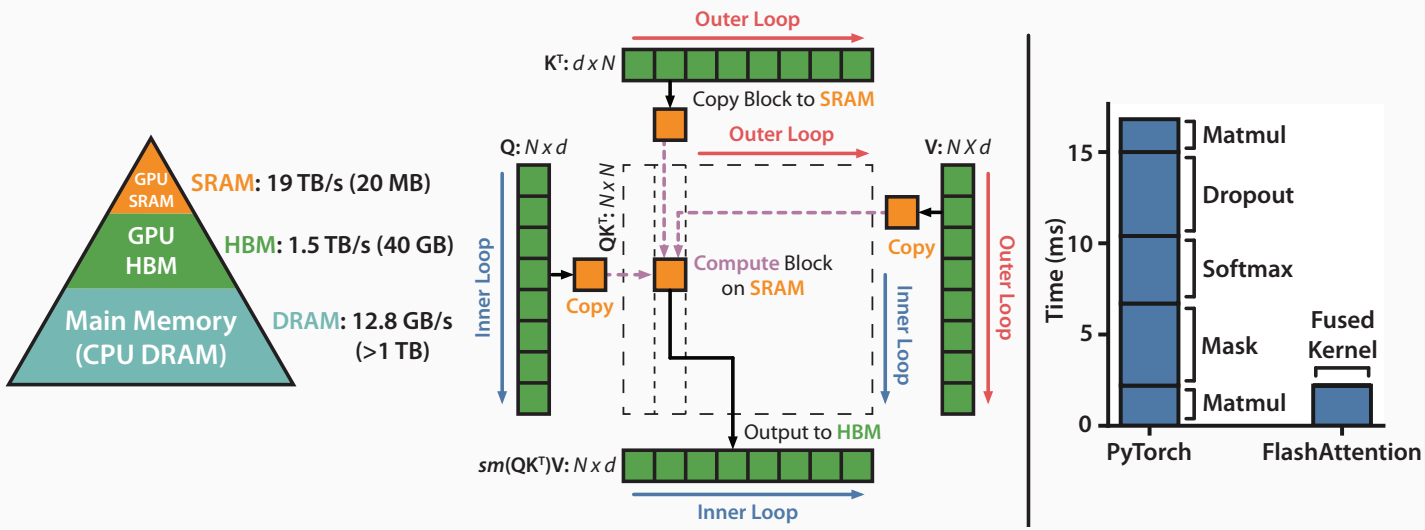
- Multi-head latent attention (MLA, DeepSeek-V2): ключи и значения сжимаются в латентное пространство малой размерности *до* кэширования.
- Кэш меньше на 90% с лишним, а качество почти не страдает; матрицы «разжатия» можно алгебраически вмонтировать в соседние проекции.
- И трюки на уровне систем: paged attention (кэш хранится несмежными блоками), квантизация кэша до 8 или 4 бит.

FlashAttention

- Самовнимание строит матрицу $L \times L$: при $L = 8192$ и 32 головах в FP16 это $8192^2 \cdot 32 \cdot 2$ байта $= 4$ ГиБ на *один* тензор.
- FlashAttention (Dao et al., 2022) считает *точное* внимание, никогда не храня эту матрицу целиком.
- Идея — *IO-awareness*: у GPU маленькая быстрая SRAM на кристалле и большая медленная HBM снаружи.
- Считаем внимание *блоками*: загружаем куски Q, K, V в SRAM, считаем частичные выходы и аккумулируем, используя онлайн-softmax.
- Дополнительная память $O(L^2) \to O(L)$, обращений к медленной памяти гораздо меньше; арифметика та же, а по времени обычно в 2–4 раза быстрее.

FlashAttention

- Иерархия памяти, разбиение на блоки и одно слитое ядро вместо пяти проходов по медленной памяти:



- Это не новая архитектура, а та же функция; но теперь FlashAttention фактически часть спецификации трансформера.
- FlashAttention-2 — лучше параллелизм (50–73% пиковой производительности GPU); FlashAttention-3 — под Hopper и FP8, ещё в 1.5–2 раза.

Другие улучшения

- *Параллельные внимание и FFN* (GPT-J, PaLM): вместо последовательного блока

$$\mathbf{z} = \mathbf{x} + \mathrm{Attn}(N(\mathbf{x})) + \mathrm{FFN}(N(\mathbf{x}));$$

  оба подслоя читают один и тот же нормализованный вход, цепочка зависимостей короче.
- *QK-нормализация*: на больших масштабах логиты $\mathbf{q}^\top \mathbf{k}/\sqrt{q}$ растут, softmax становится почти one-hot, и обучение расходится. Нормализуем запросы и ключи перед скалярным произведением; к 2025 году это стандарт (Gemma 3, OLMo 2, Qwen3).
- Альтернатива — soft-capping логитов $c\tanh(x/c)$ (Gemma 2).
- *Без смещений*: вслед за PaLM большинство моделей убирают bias из линейных слоёв и нормализаций; стабильнее и ничего не стоит.

Другие улучшения

- *Локальное и глобальное внимание*: самая дешёвая запись в KV-кэше — та, которую не храним.
- Слой со скользящим окном смотрит только на W предыдущих токенов: кэш не больше W, а рецептивное поле всё равно растёт на W с каждым слоем, как у свёрток.
- Mistral 7B: окно 4096 во всех слоях; Gemma 2: локальные и глобальные слои через один; Gemma 3: пять локальных с окном 1024 на один глобальный.
- И позиционная информация делится между ними: в Command A и Llama 4 глобальные слои вообще без позиционных кодировок (NoPE), порядок слов дают локальные.

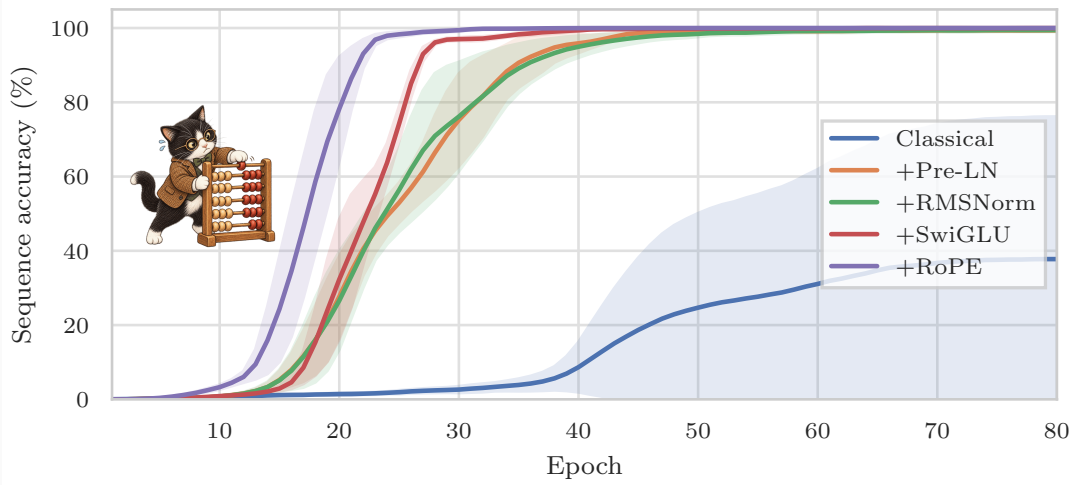
Рецепт в деле: эксперимент

- Маленький трансформер ($d = 64$, 4 головы, $2 + 2$ слоя), пять вариантов, каждый добавляет один ингредиент:
  - Classical (Post-LN, LayerNorm, ReLU, синусоиды) $\rightarrow$ +Pre-LN $\rightarrow$ +RMSNorm $\rightarrow$ +SwiGLU $\rightarrow$ +RoPE.
- Две задачи: развернуть строку из 10 символов и сложить два трёхзначных числа.

		Разворот		Сложение	
Вариант	Параметров	Точн., %	Эп.$\rightarrow$90%	Точн., %	Эп.$\rightarrow$90%
Classical	237 533	100.0	17	37.7 ± 38.3	—
+Pre-LN	237 789	100.0	13	99.4 ± 0.2	34
+RMSNorm	237 021	100.0	13	99.5 ± 0.4	35
+SwiGLU	233 693	100.0	14	100.0 ± 0.0	26
+RoPE	233 693	98.5	33	100.0 ± 0.0	21

Рецепт в деле: эксперимент

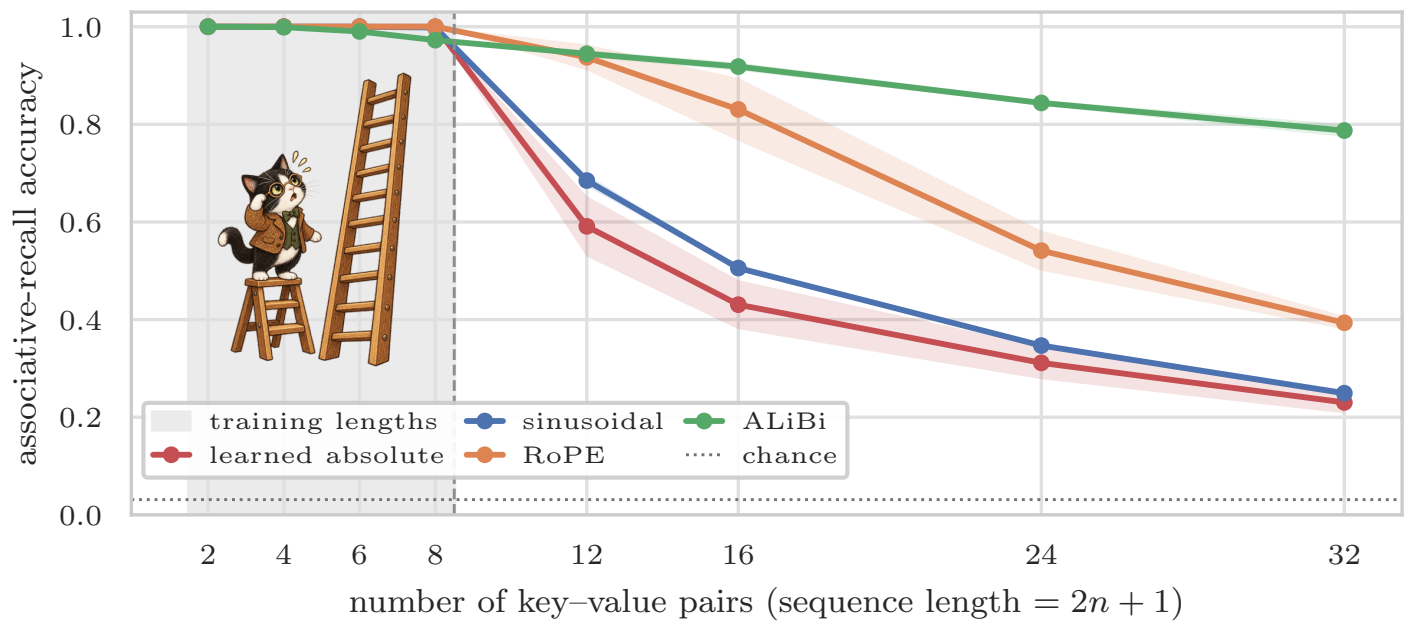
- Разворот: Pre-LN — самый большой выигрыш в скорости; а RoPE *замедляет*: задаче нужны абсолютные позиции («символ 0 идёт на место 9»).
- Сложение: Classical нестабилен (одно зерно выучило перенос разрядов, два — нет), Pre-LN это полностью лечит, а RoPE теперь быстрее всех:



- В этих двух задачах Pre-LN стабильно помогает; SwiGLU даёт небольшой выигрыш, а RoPE помогает или мешает в зависимости от задачи.

Экстраполяция на длину

- Ассоциативное припоминание: $[k_1, v_1, \ldots, k_n, v_n, k_q] \mapsto v_q$; учим на $2 \le n \le 8$ парах, проверяем до $n = 32$.
- Четыре модели отличаются только позиционным механизмом: выученные позиции, синусоиды, RoPE, ALiBi.



- Идеальная точность на длинах из обучения ничего не гарантирует за их пределами; лучше всех обобщает ALiBi.

Экстраполяция на длину

- А что внутри? Естественная двухслойная схема (*induction head*): на первом слое значение v_i читает стоящий перед ним ключ k_i, на втором запрос k_q находит значение с нужным ключом.
- В обученной модели с RoPE такая схема действительно появляется; на длине из обучения точность почти 100%, на вдвое большей — около 75%:



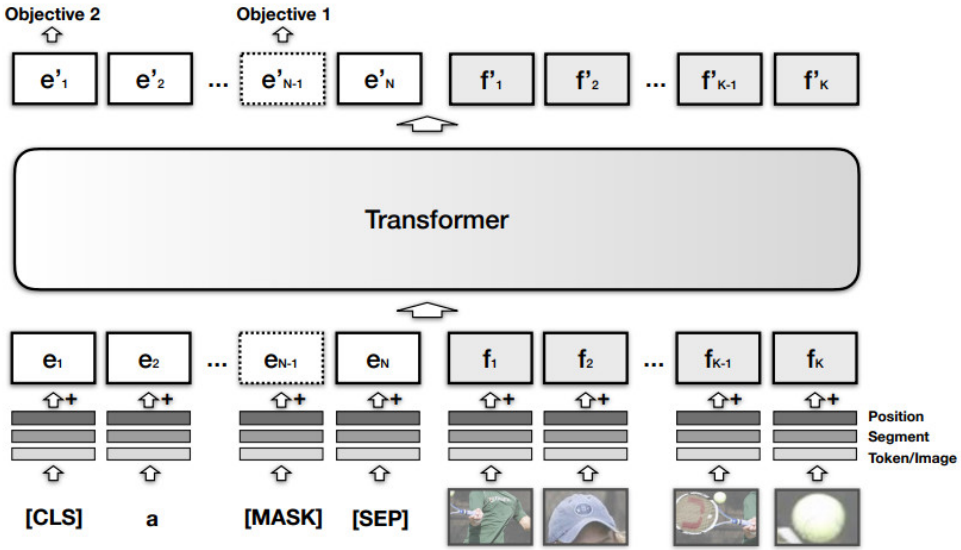
- Но веса внимания — не объяснение: на втором слое все головы смотрят куда надо, а выключение любой одной почти ничего не меняет; нужны и визуализация, и абляции.

Vision Transformers

Visual BERT

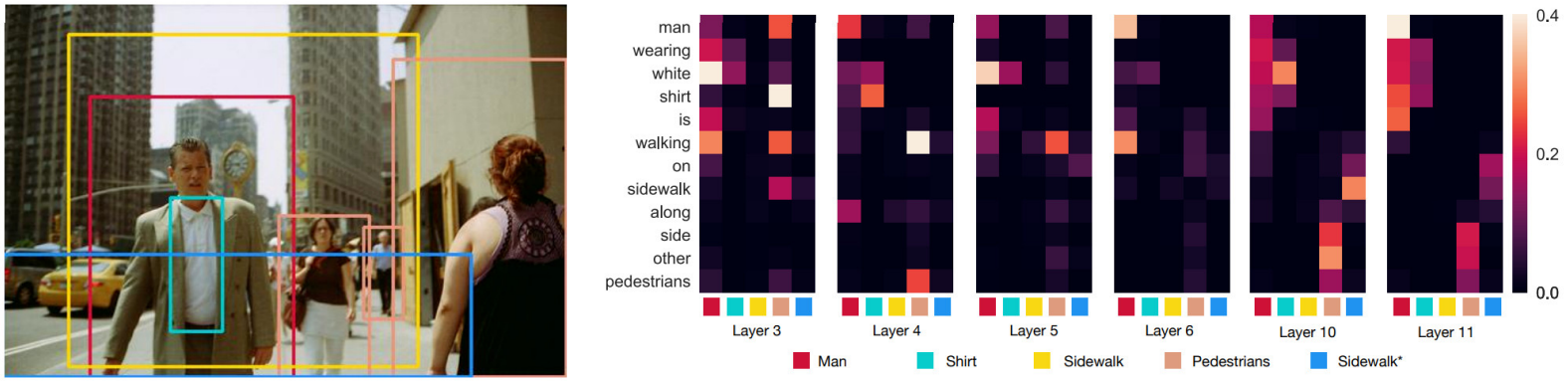
- Картинки — это же не последовательности? Ну да, но...
- Visual BERT: давайте вместе моделировать картинки и подписи к ним
- Используем предобученный object detection вроде Faster R-CNN, вырезаем объекты, строим их вложения через CNN и positional embeddings, подаём в единый трансформер с подписью



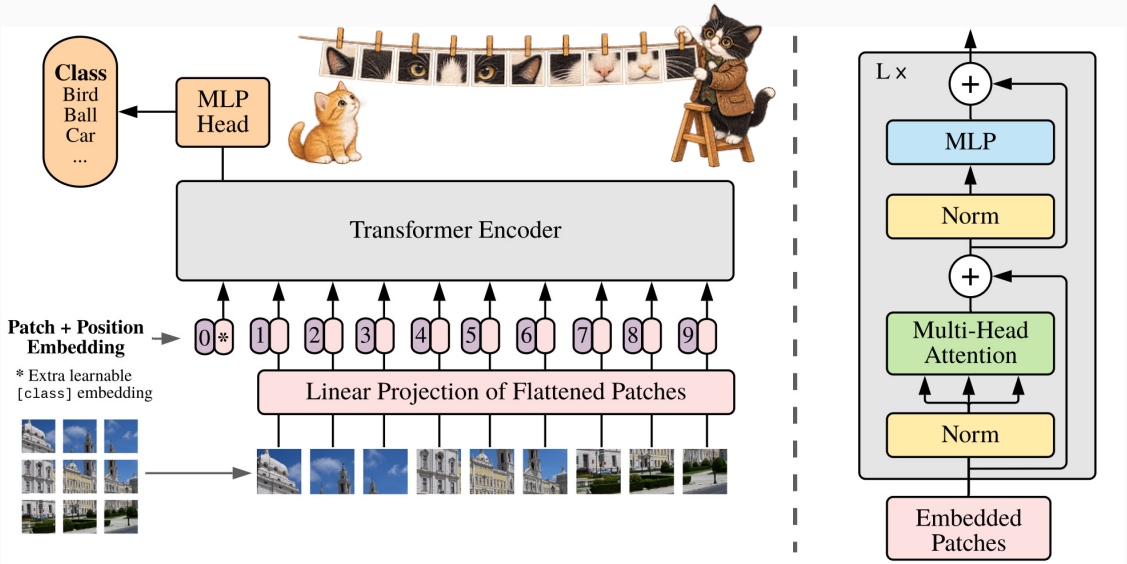


Visual BERT

- Предобучение: masked language modeling + sentence-image prediction (подходит ли подпись к рисунку?)
- Примеры голов внимания показывают, как слова из подписи «смотрят» на соответствующие объекты

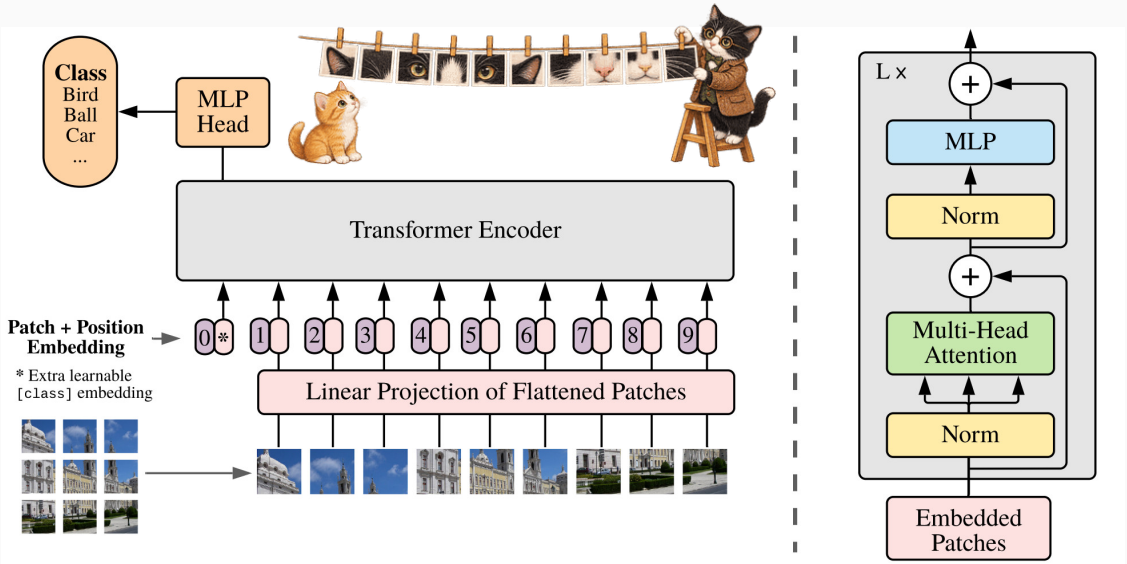


Visual Transformer (ViT)



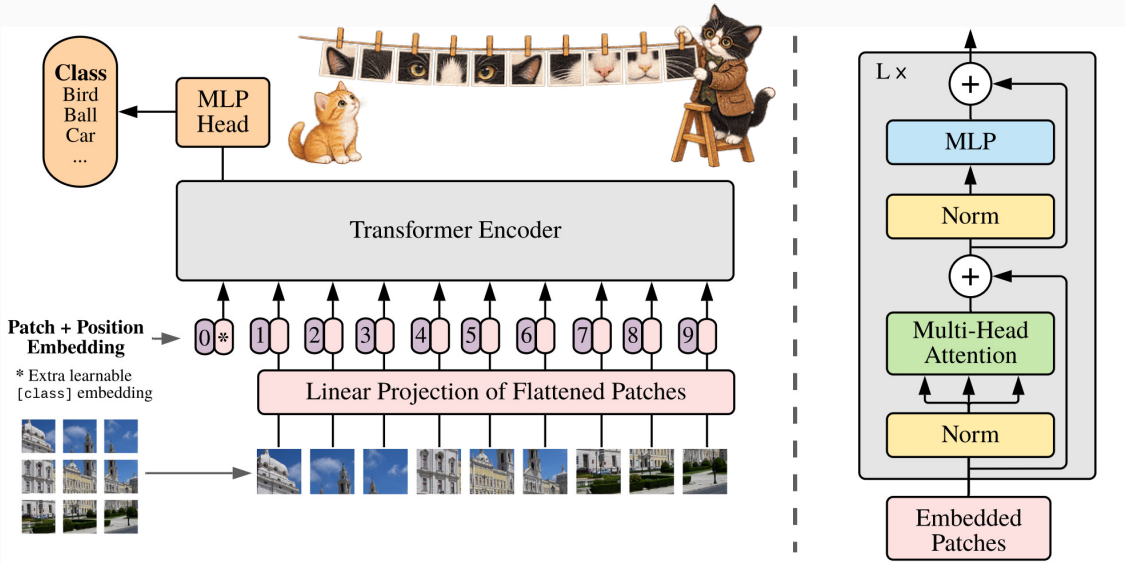
- Самая успешная Transformer-архитектура для картинок: *Vision Transformer* (ViT; Dosovitsky et al., 2020)
- ViT — прямолинейная модификация BERT, но теперь на входе токены из изображения, а не текста

Visual Transformer (ViT)



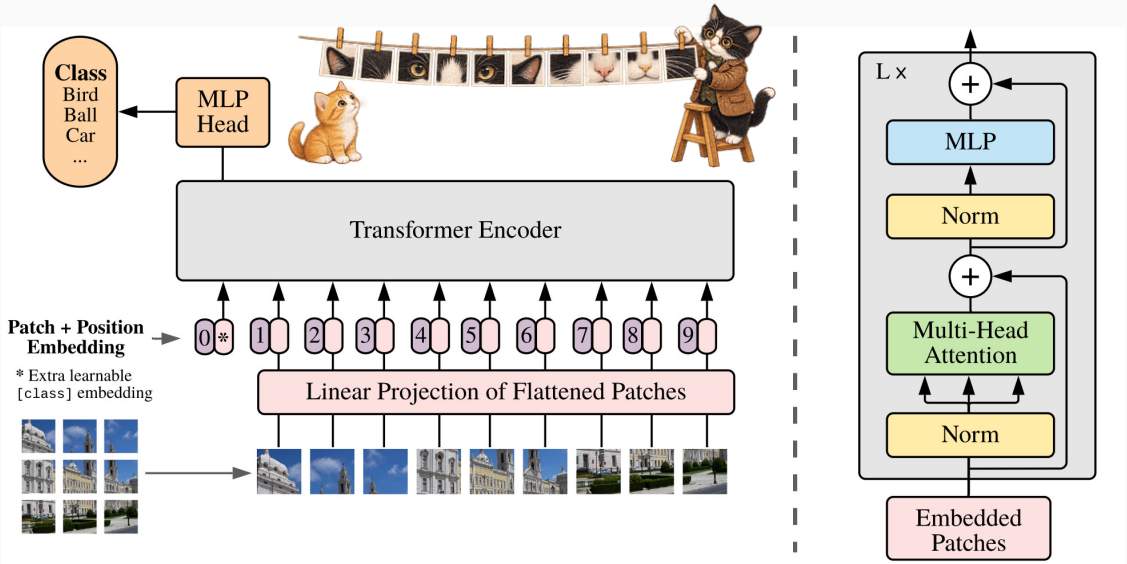
- Вход режется на маленькие патчи: $H \times W$ картинка с C каналами $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ превращается в последовательность патчей $\mathbf{x}_p \in \mathbb{R}^{N \times P^2 \times C}$, где $N = HW/P^2$ — число $P \times P$ патчей

Visual Transformer (ViT)



- Патчи превращаются во вложения через обычную линейную проекцию, последовательность векторов подаётся в кодировщик трансформера
- Основной ViT предобучали на классификации с учителем; маскирование патчей — отдельная линия предобучения

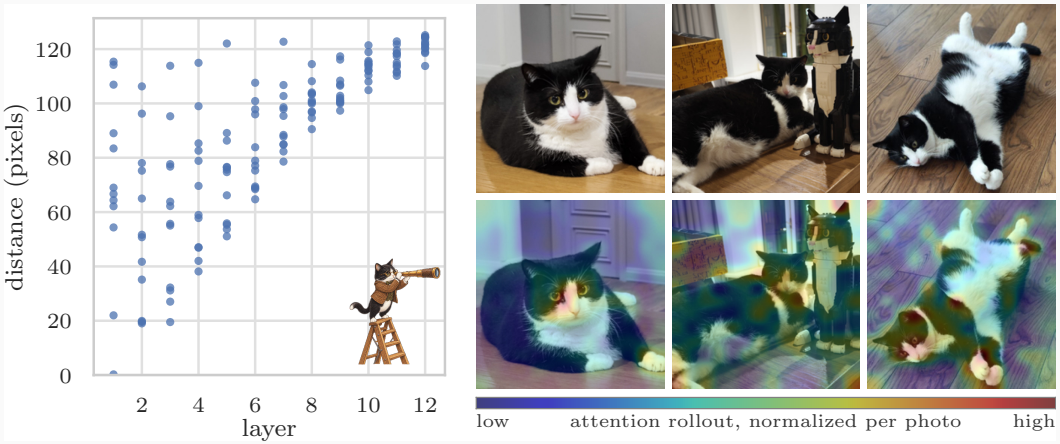
Visual Transformer (ViT)



- ViT использует positional encodings, но те же самые, линейные!
- Dosovitsky et al. экспериментировали с 2D-кодировками, но никакого улучшения не нашли...

Что видит ViT

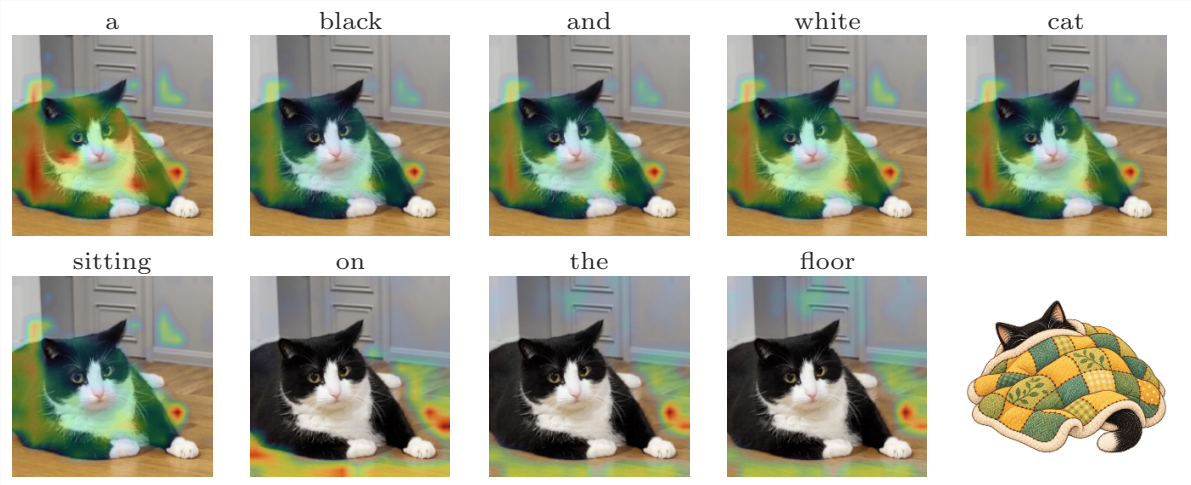
- Индуктивных смещений у ViT почти нет, и за это платят данными: ViT-Base, обученный с нуля на ImageNet-1K, проигрывает хорошему ResNet, а предобученные на JFT-300M (300 млн картинок) ViT-Large и ViT-Huge обходят лучшие CNN.
- Что внутри? Мы прогнали исходный ViT-B/16 и перерисовали две картинки из статьи: среднее расстояние внимания каждой головы по слоям (слева) и attention rollout токена `[CLS]` на фотографиях кота (справа):



- В нижних слоях одни головы смотрят на всю картинку, другие локально, а к последней трети сети глобальны все.

ViT и декодер: подписи к картинкам

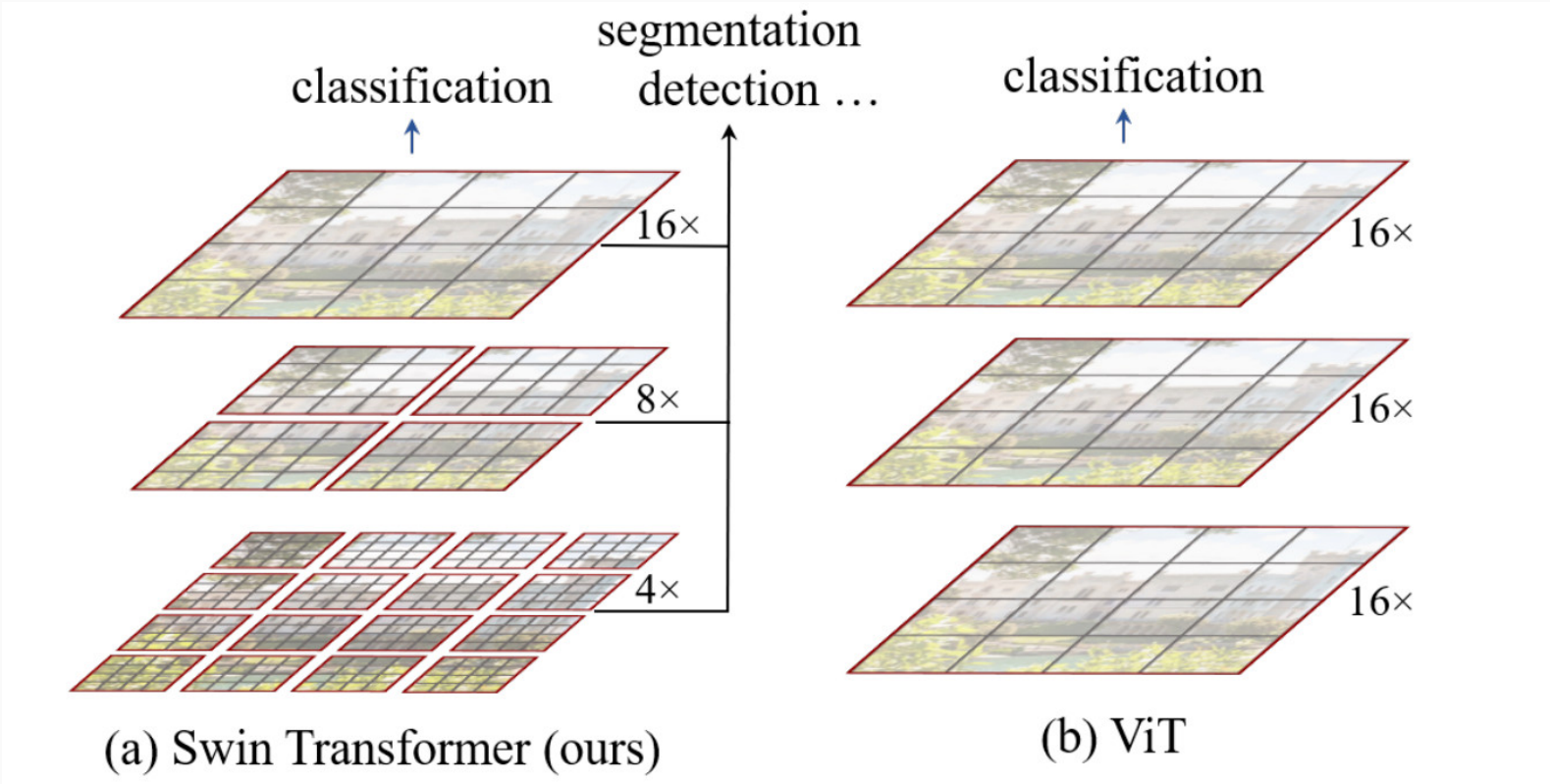
- ViT не обязан заканчиваться классификатором: повесим на него декодер трансформера с cross-attention к патчам, и получится прямой потомок Show, Attend and Tell.
- Тот же кот, что в прошлый раз: старая модель выдумала «a cat laying on a couch with a remote», а ViT с декодером GPT-2 пишет «a black and white cat sitting on the floor».



- Зато внимание гораздо более размытое: почти на каждом слове оно накрывает всего кота. Качество выросло, а читаемость карт внимания — нет.

Swin Transformers

- *Swin Transformer* (Liu et al., 2021; **s**hifted **win**dows): похож на ViT, но иерархический



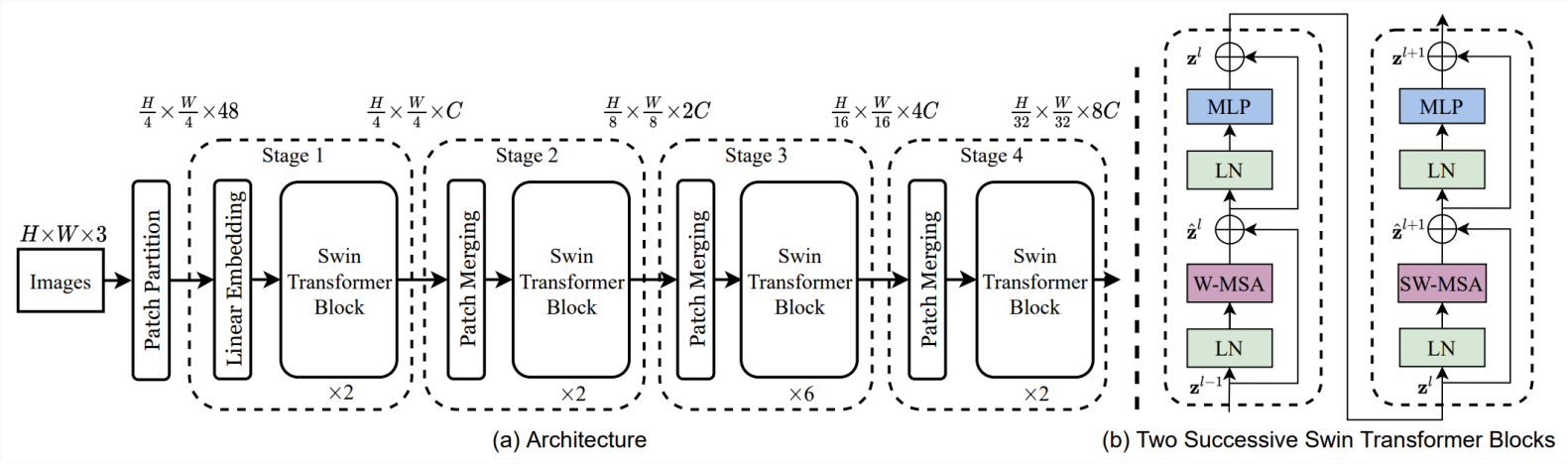
Swin Transformers

- Обрабатывает патчи на нескольких масштабах, вычисляя самовнимание по патчам с CNN-подобной архитектурой
- Окна сдвигаются между слоями, и это даёт связь между частями изображения



Swin Transformers

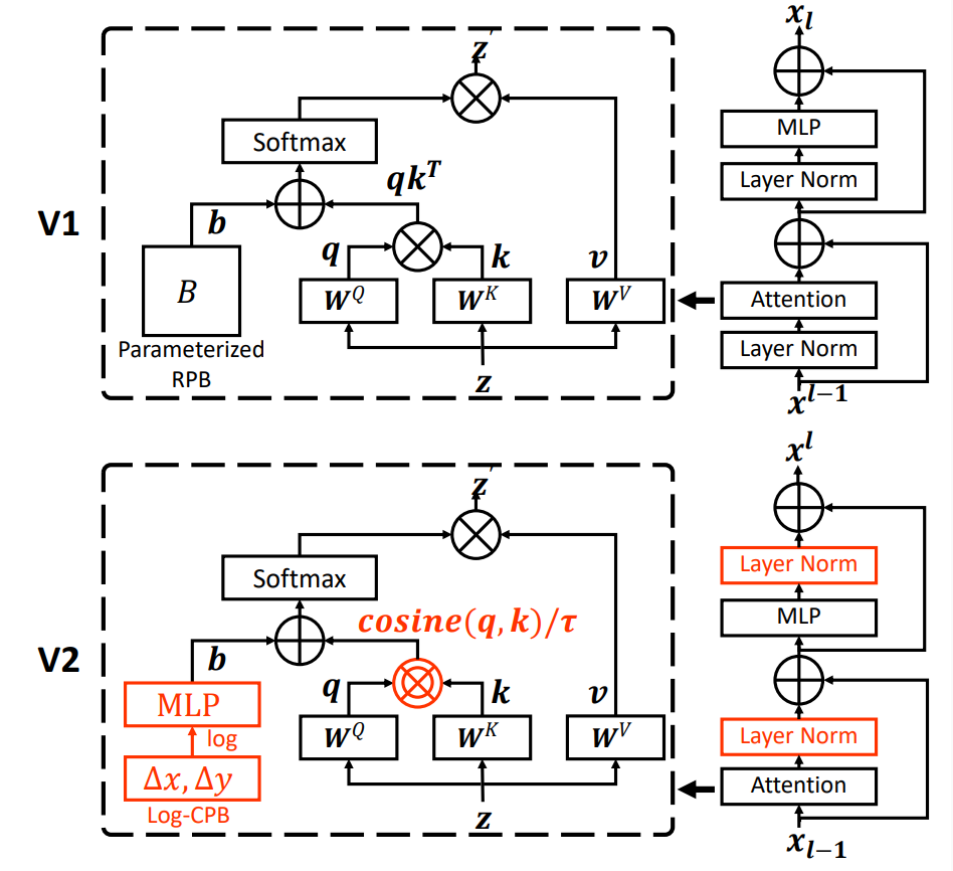
- Сама архитектура сокращает геометрию, как классические CNN backbones:



- Swin может масштабироваться до высоких разрешений и может использоваться для задач распознавания объектов и сегментации

Swin Transformers

- Потом вышел *Swin Transformer v2* (Liu et al., 2022) — это Swin Transformer, отмашстабированный до 3B параметров, на картинках до 1536×1536 пикселей, что ещё сильнее улучшает обработку изображений во всех задачах



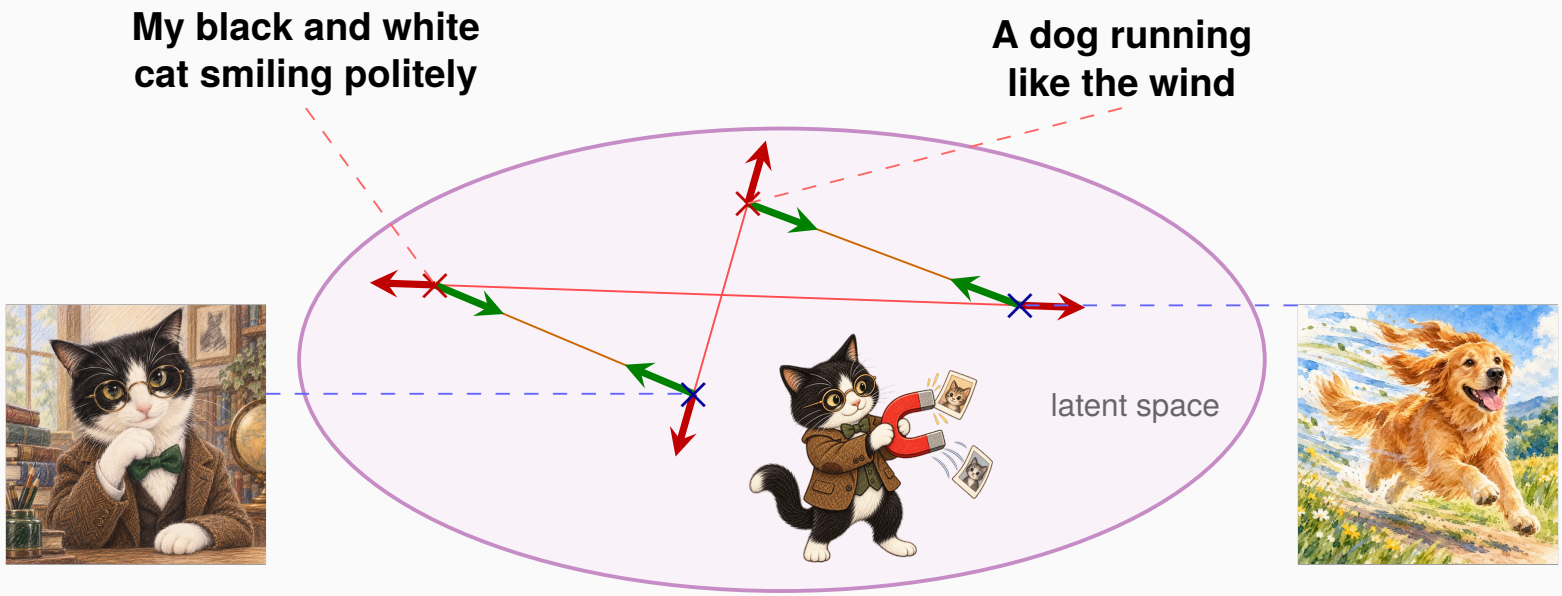
CLIP и BLIP

Мультимодальные латентные пространства

- У нас есть много картинок и текстов, свободно доступных в интернете
- Как использовать текст в паре с изображениями так, чтобы получить хорошее мультимодальное латентное пространство?
- Авторы CLIP пишут, что их первой идеей было обучить свёрточную сеть для изображений и трансформер для текста, чтобы предсказывать подпись к изображению
- Но это не сработало: предсказывать точную подпись очень сложно (практически безнадёжно), и это не совсем то, что нам нужно в этой модели, нам просто нужны хорошие мультимодальные представления

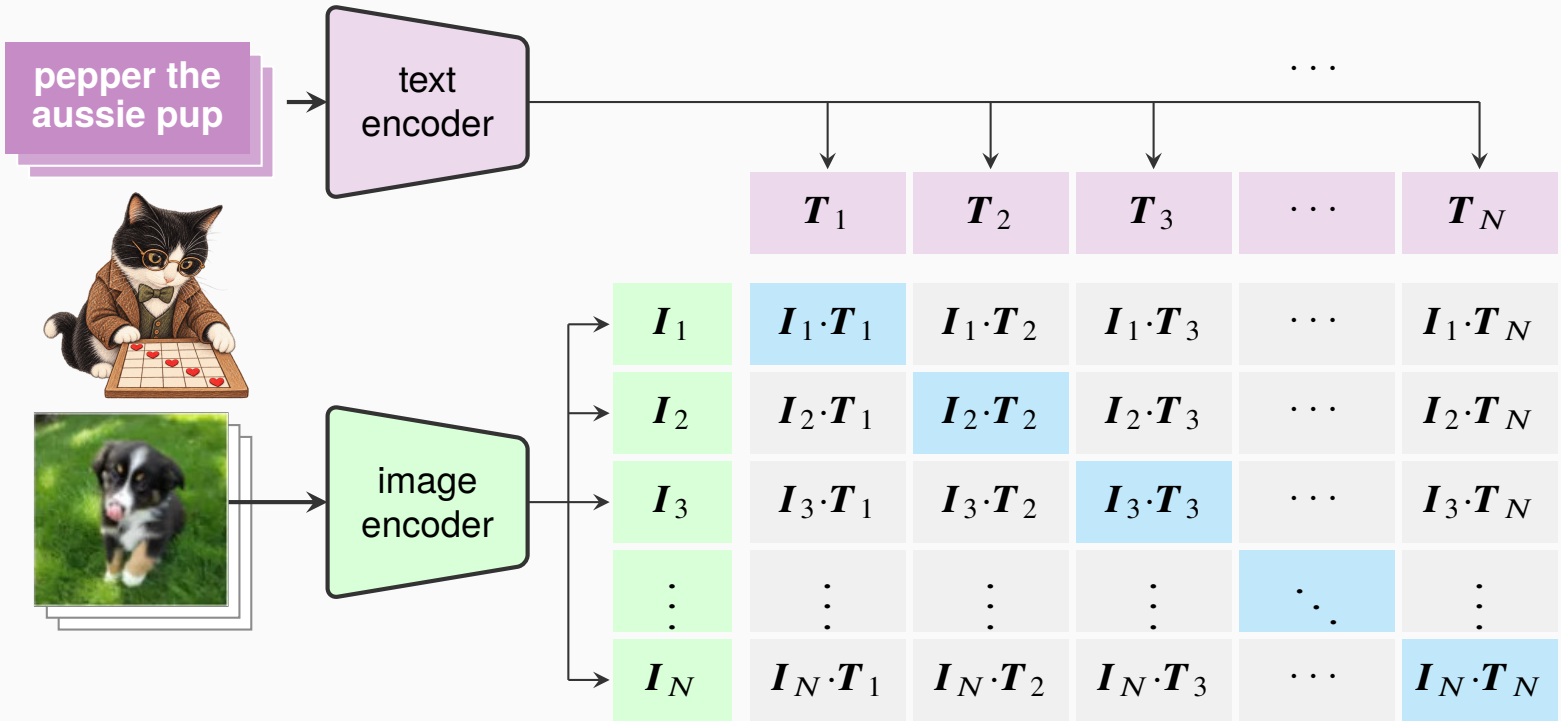
CLIP

- Контрастивное предобучение (contrastive pretraining): моделируем силы притяжения и отталкивания в совместном латентном пространстве



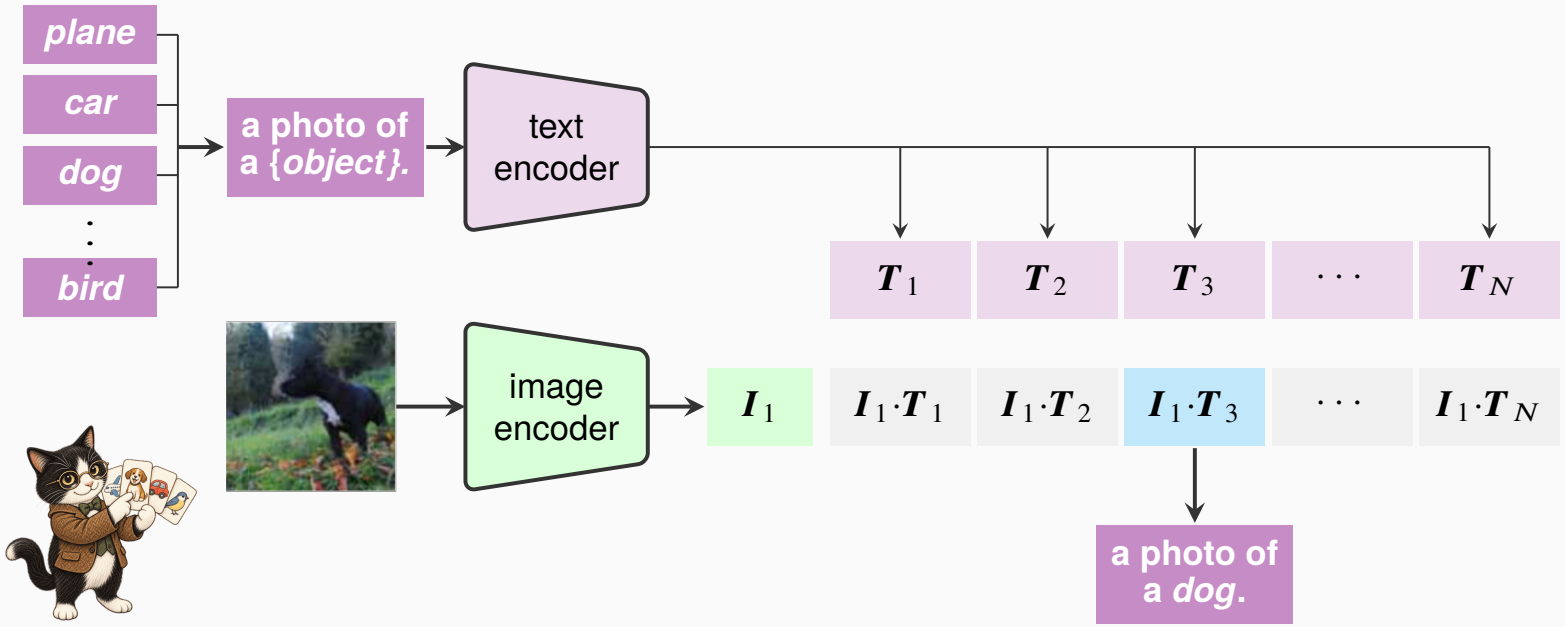
CLIP

- CLIP (Radford et al., 2021; OpenAI), Contrastive Language–Image Pretraining; использовали пары изображение-текст, которые были просто собраны из интернета; кодировщик изображений — тот самый ViT:



CLIP

- В итоге работала даже zero-shot классификация – преобразуем метки классов в простые запросы:



CLIP

- Попробуем сами: самый дешёвый CLIP (ViT-B/32, около 150M параметров) на шести фотографиях моего кота, восемь запросов вида «a photo of a ...»:



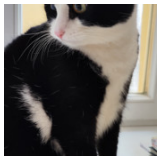
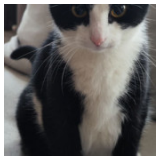
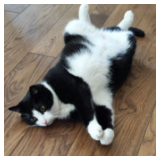
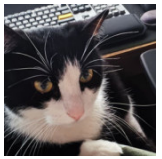
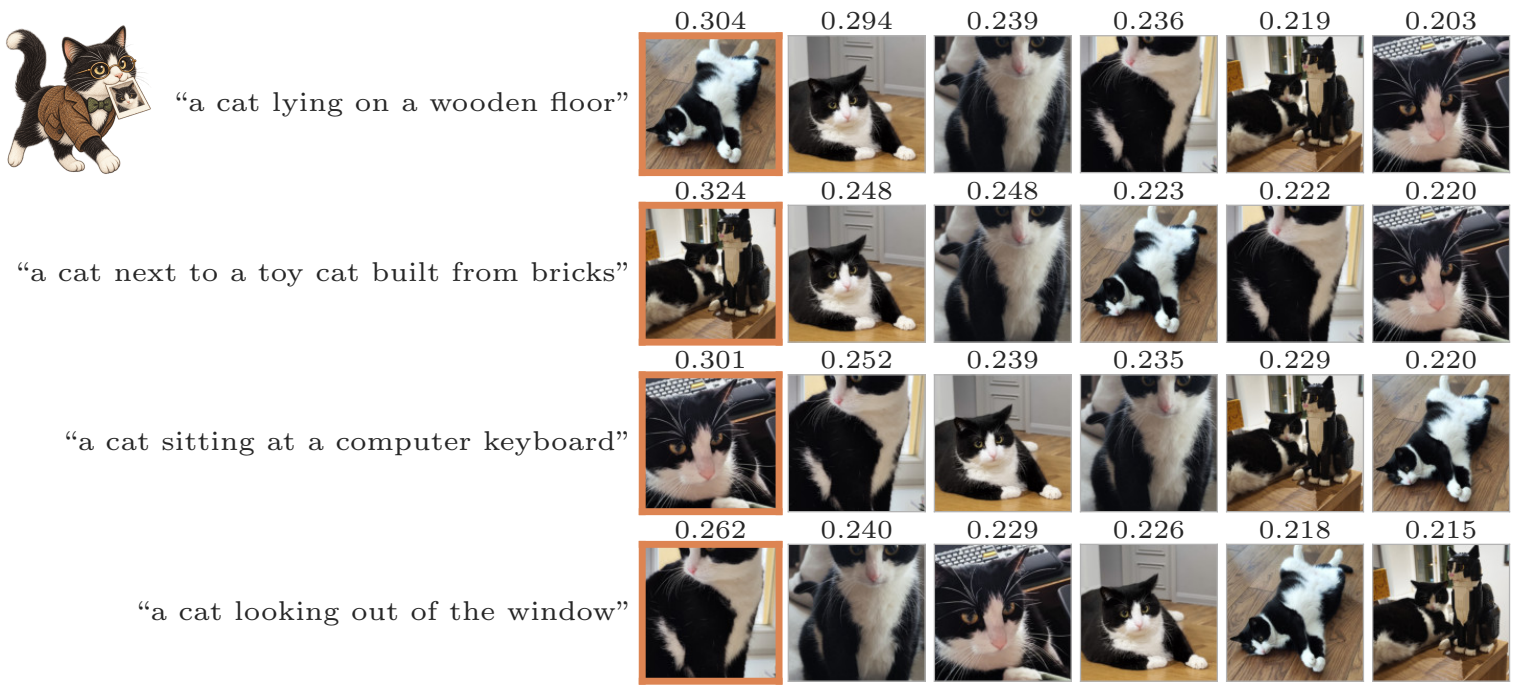






	image 1	image 2	image 3	image 4	image 5	image 6
“a photo of a cat”	0.96	0.89	0.97	0.98	0.95	0.95
“a photo of a dog”	0.01	0.00	0.01	0.01	0.01	0.00
“a photo of a tiger”	0.01	0.00	0.01	0.00	0.01	0.02
“a photo of a rabbit”	0.00	0.00	0.01	0.01	0.00	0.00
“a photo of a LEGO set”	0.00	0.04	0.00	0.00	0.00	0.00
“a photo of a cardboard box”	0.01	0.06	0.00	0.00	0.00	0.00
“a photo of a computer keyboard”	0.00	0.00	0.00	0.00	0.00	0.02
“a photo of a wooden floor”	0.01	0.00	0.00	0.00	0.03	0.00

- Кот побеждает везде (от 0.89), но модель заметила и LEGO с коробкой, и клавиатуру: в совместном пространстве вся сцена.

CLIP

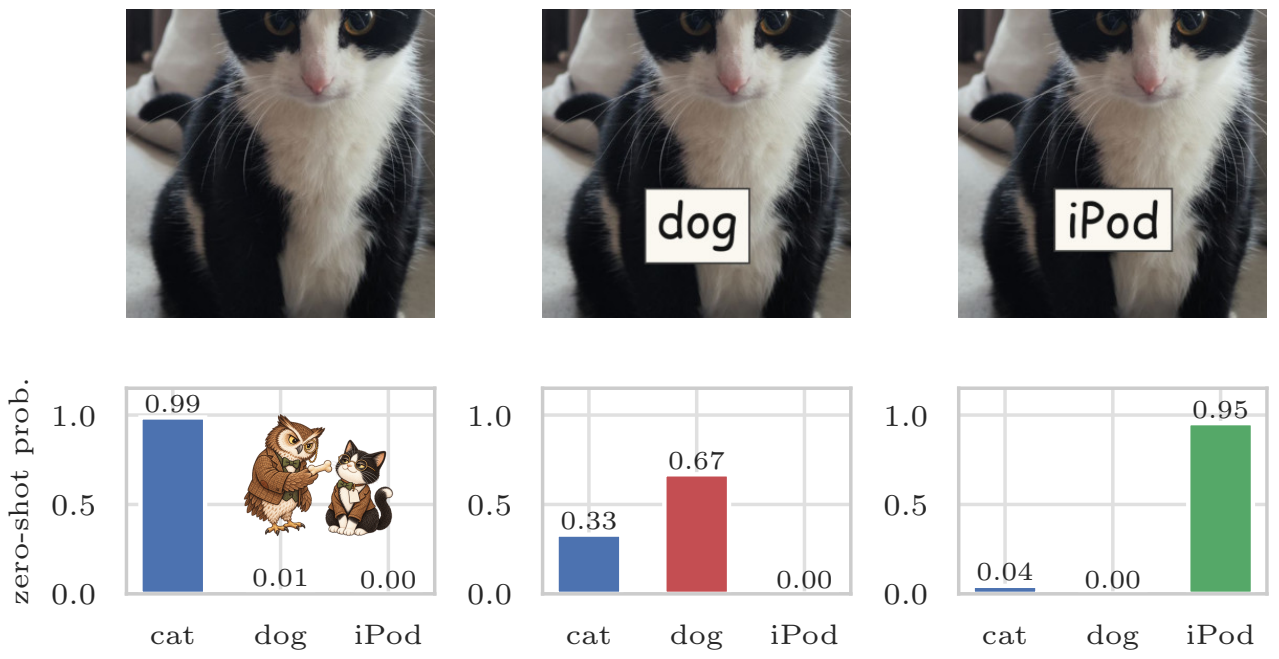
- Поэтому CLIP хорош для поиска по свободному тексту: каждый запрос находит нужную фотографию, хотя ни один не похож на метку класса:



- Но отрывы маленькие: у победителя сходство около 0.3, у остальных около 0.22.

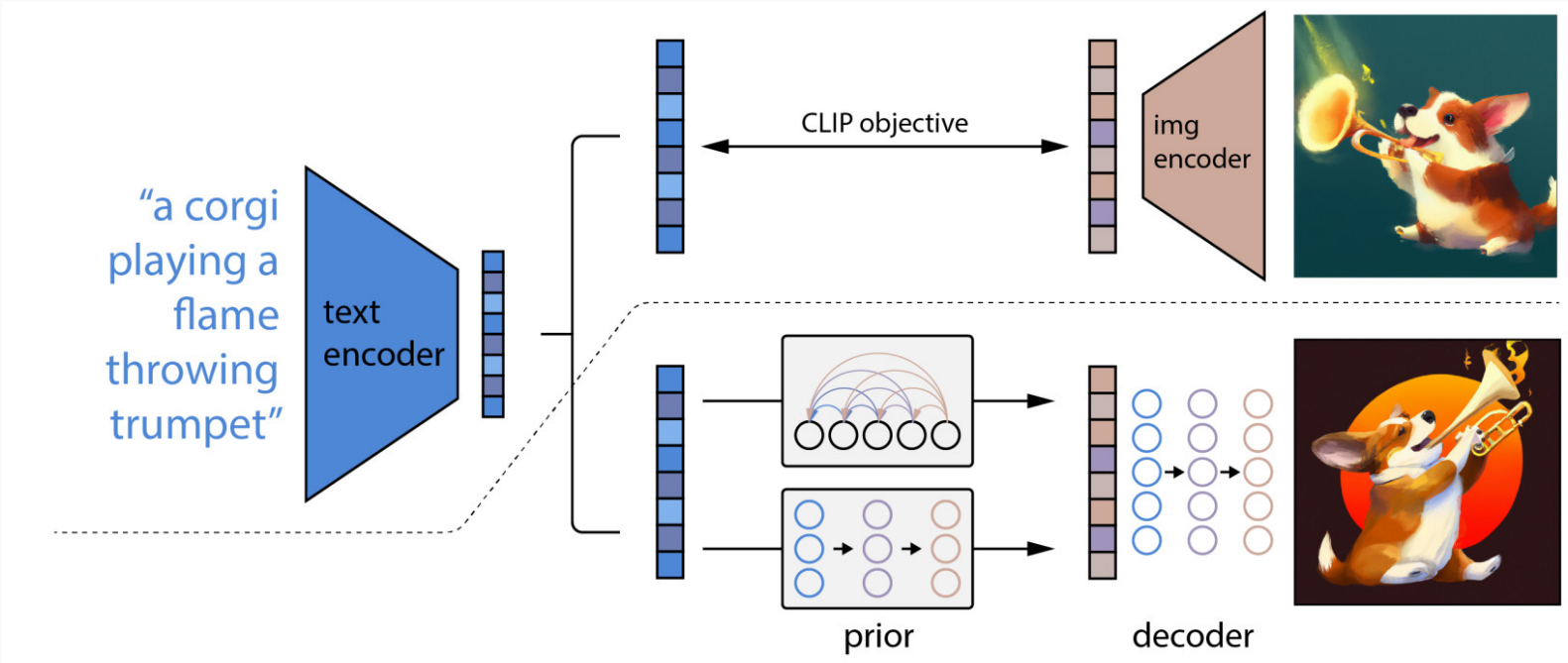
CLIP

- И забавный провал — типографская атака (Goh et al., 2021): CLIP читает текст на картинке так же, как видит объекты.
- Наклеили бумажку «dog» — две трети вероятности уходят к собаке; бумажку «iPod» — 0.95 за плеер, хотя ни один пиксель кота не изменился:



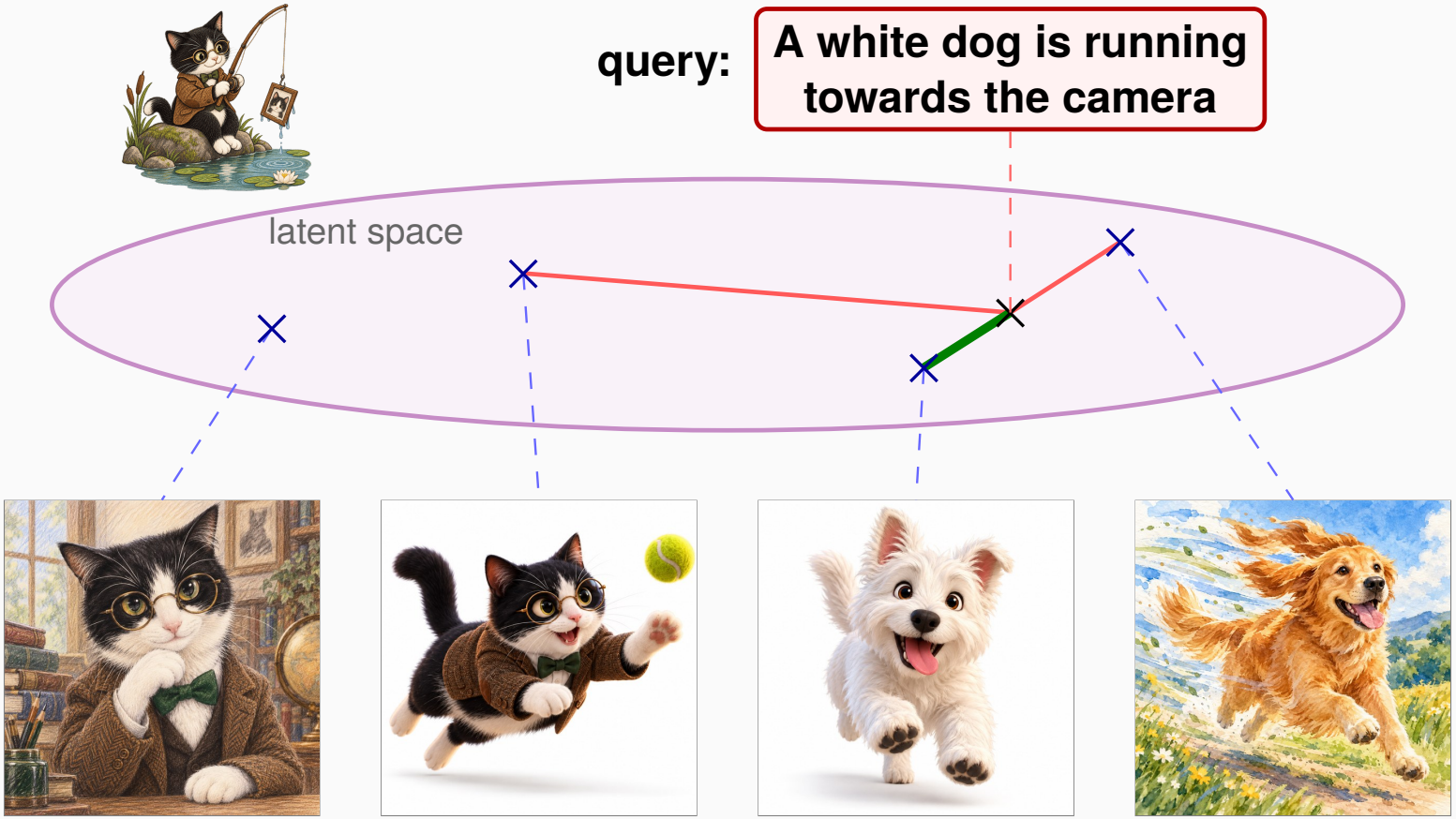
CLIP

- CLIP стал популярным источником совместных представлений; например, DALL-E 2 рисовал картинки в латентном пространстве CLIP:



CLIP

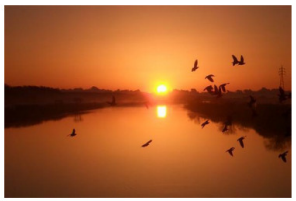
- Основное применение CLIP — это обеспечение информационного поиска и порождающих моделей:



BLIP

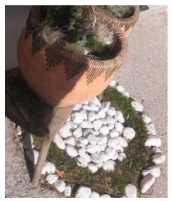
- BLIP (Li et al., 2022) – Bootstrapping Language-Image Pretraining; порождаем синтетические подписи и фильтруем их





T_w: "from bridge near my house"

T_s: "a flock of birds flying over a lake at sunset"



T_w: "in front of a house door in Reichenfels, Austria"

T_s: "a potted plant sitting on top of a pile of rocks"



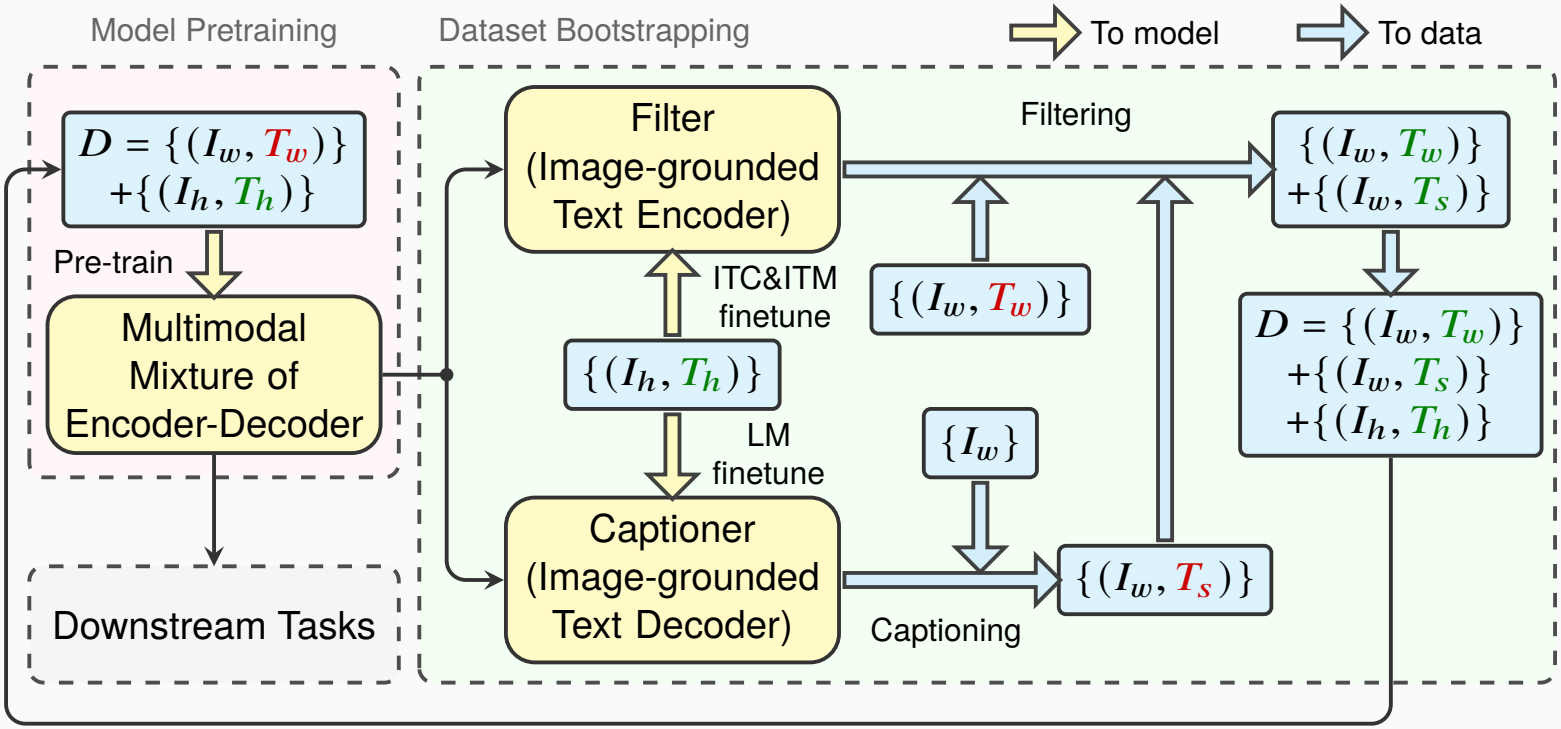
T_w: "the current castle was built in 1180, replacing a 9th century wooden castle"

T_s: "a large building with a lot of windows on it"



BLIP

- Структура обучения:

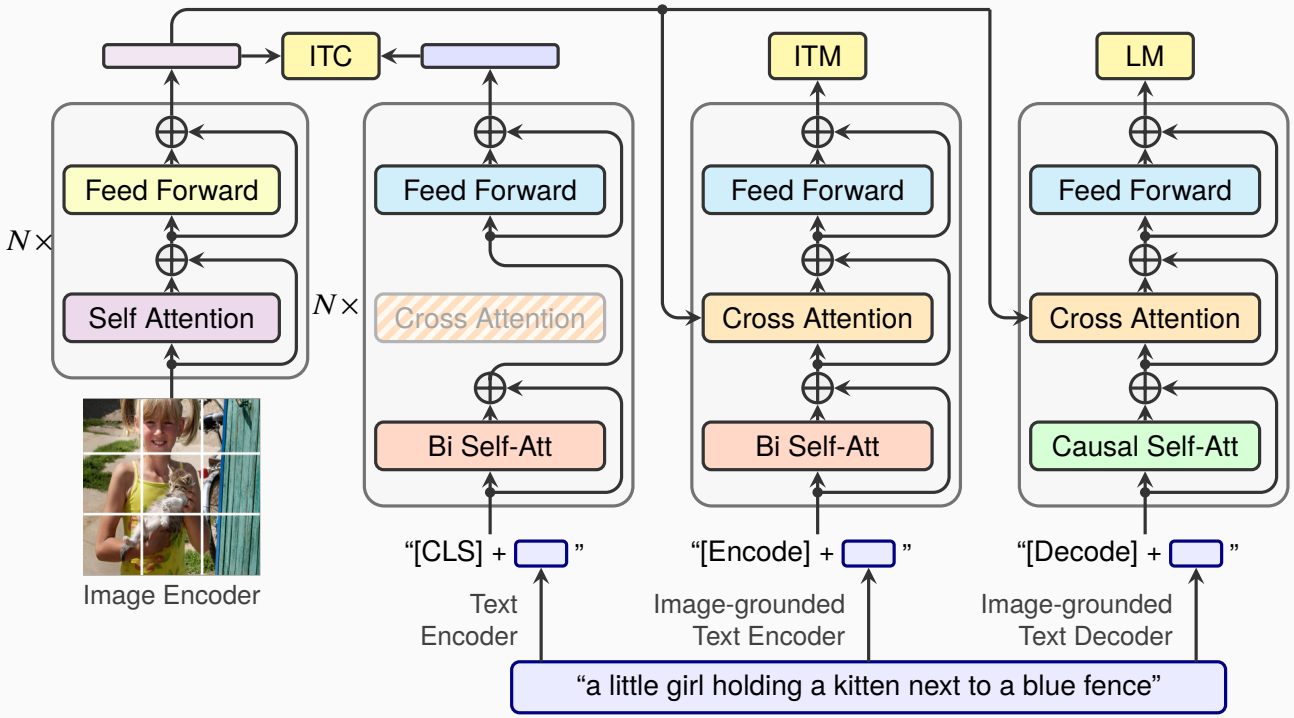


I_w: web images I_h: human-annotated images T_h: human-annotated texts

T_w: web texts T_w: filtered web texts T_s: synthetic texts T_s: filtered synthetic texts

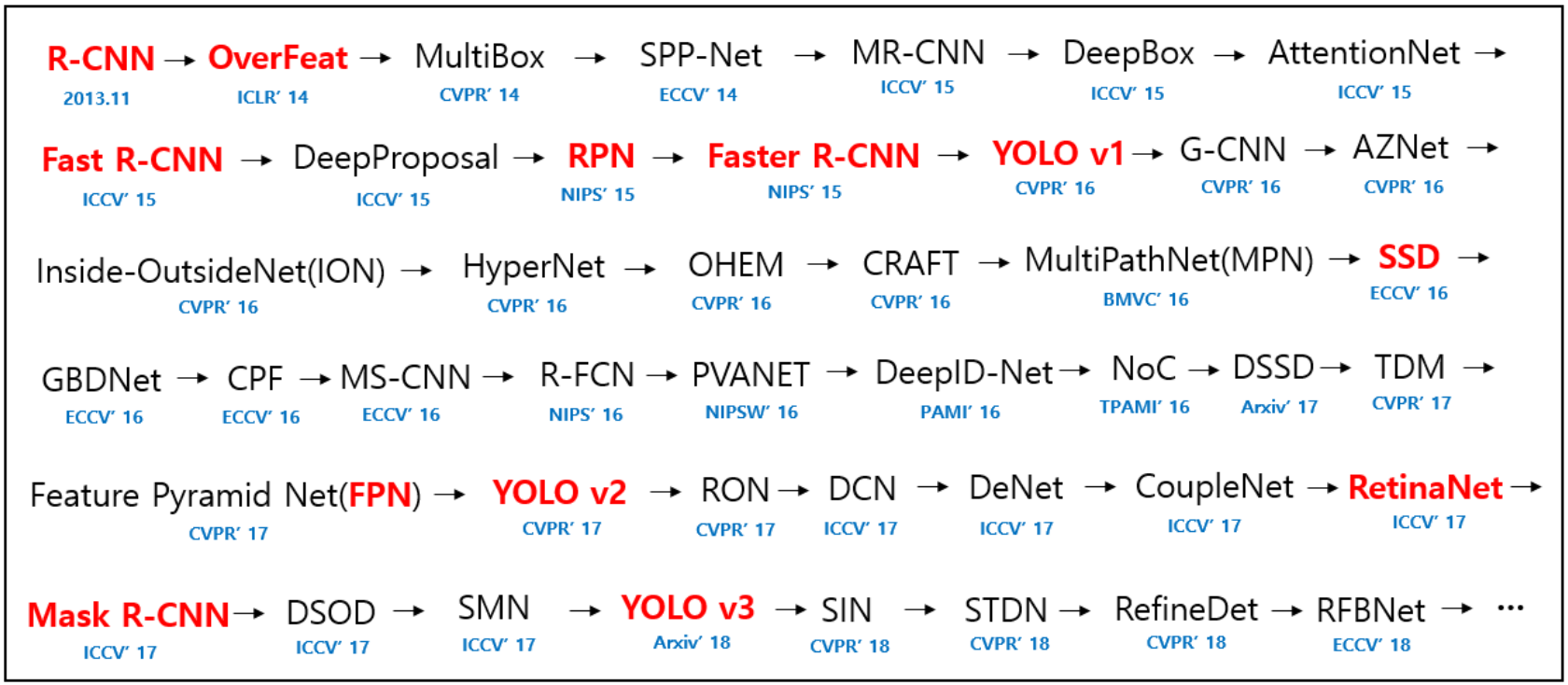
BLIP

- Три разных кодировщика и декодер: ITC – контрастивная функция потерь для изображения и текста (сопоставление представлений изображения и текста), ITM – функция потерь на соответствие изображения и текста (различение положительных и отрицательных пар), LM – языковое моделирование (авторегрессивное порождение подписей)



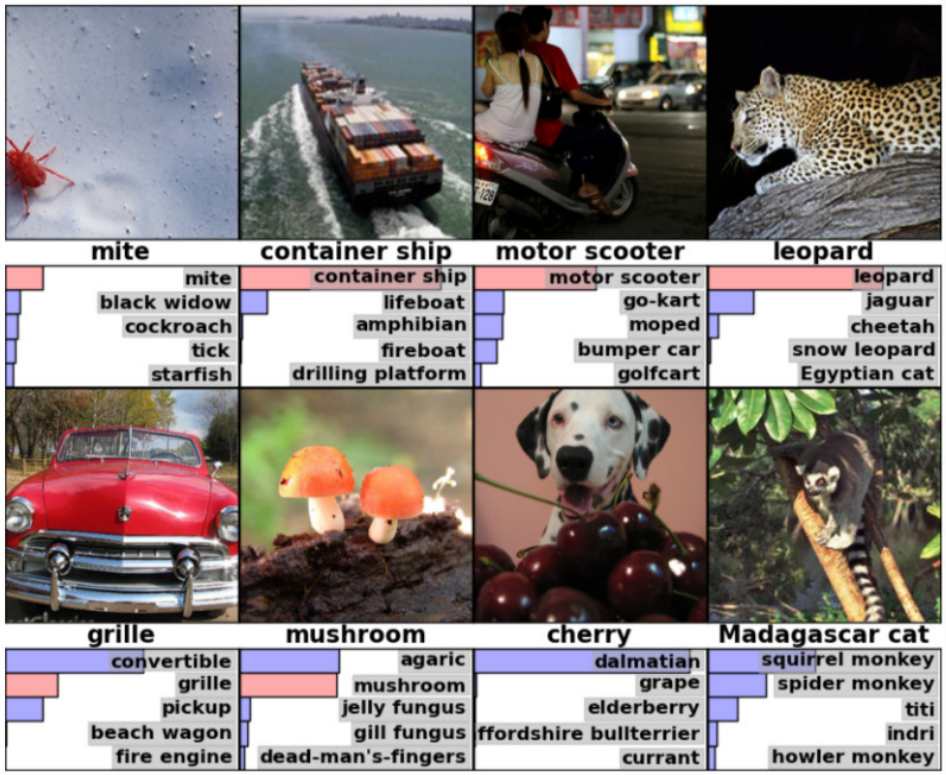
Распознавание объектов

План


R-CNN 2013.11
OverFeat ICLR' 14
MultiBox CVPR' 14
SPP-Net ECCV' 14
MR-CNN ICCV' 15
DeepBox ICCV' 15
AttentionNet ICCV' 15
Fast R-CNN ICCV' 15
DeepProposal ICCV' 15
RPN NIPS' 15
Faster R-CNN NIPS' 15
YOLO v1 CVPR' 16
G-CNN CVPR' 16
AZNet CVPR' 16
Inside-OutsideNet(ION) CVPR' 16
HyperNet CVPR' 16
OHEM CVPR' 16
CRAFT CVPR' 16
MultiPathNet(MPN) BMVC' 16
SSD ECCV' 16
GBDNet ECCV' 16
CPF ECCV' 16
MS-CNN ECCV' 16
R-FCN NIPS' 16
PVANET NIPSW' 16
DeepID-Net PAMI' 16
NoC TPAMI' 16
DSSD Arxiv' 17
TDM CVPR' 17
Feature Pyramid Net(FPN) CVPR' 17
YOLO v2 CVPR' 17
RON CVPR' 17
DCN ICCV' 17
DeNet ICCV' 17
CoupleNet ICCV' 17
RetinaNet ICCV' 17
Mask R-CNN ICCV' 17
DSOD ICCV' 17
SMN ICCV' 17
YOLO v3 Arxiv' 18
SIN CVPR' 18
STDN CVPR' 18
RefineDet CVPR' 18
RFBNet ECCV' 18
...

Постановка задачи и датасеты

- Мы обсуждали классификацию: Imagenet (> 15M картинок, ≈ 22K категорий).
- Размечены вручную (Amazon Mechanical Turk). Стандартный датасет для обучения всех свёрточных сетей для обработки фотографий.



Постановка задачи и датасеты

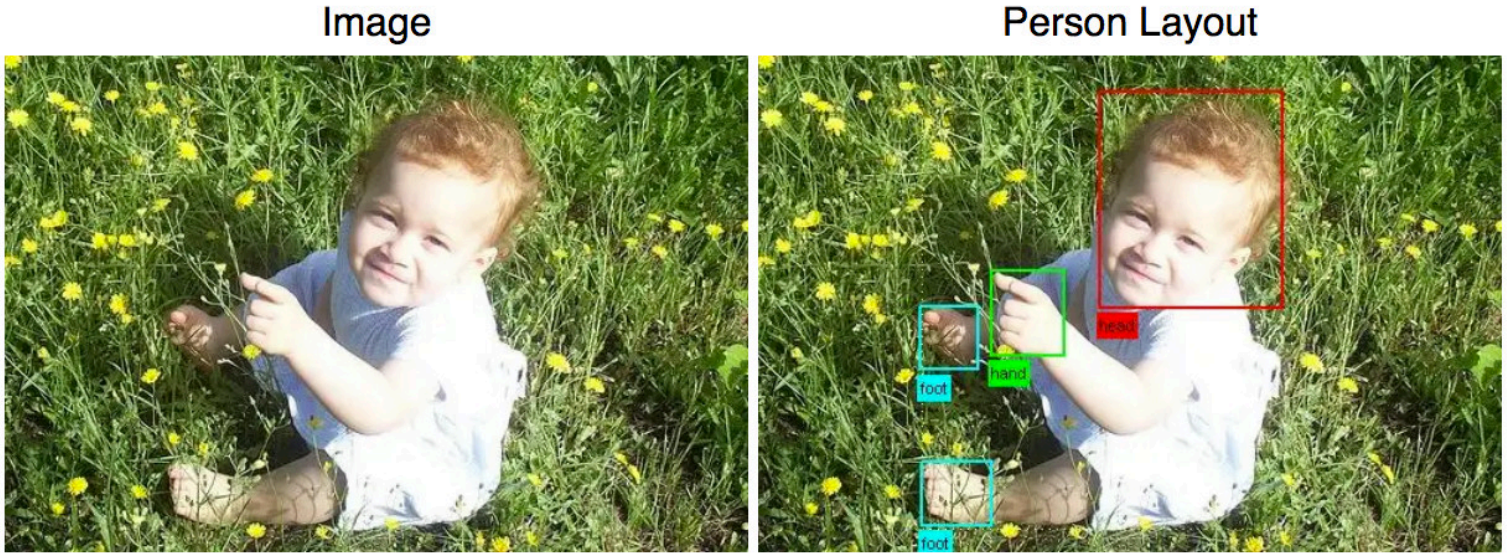
- Но ведь надо же ещё и понять, *где* эти объекты (и распознавать несколько объектов на одной и той же фотографии).
- PASCAL VOC для сегментации:



- Относительно маленький, так что сначала всё предобучается на *ImageNet*.

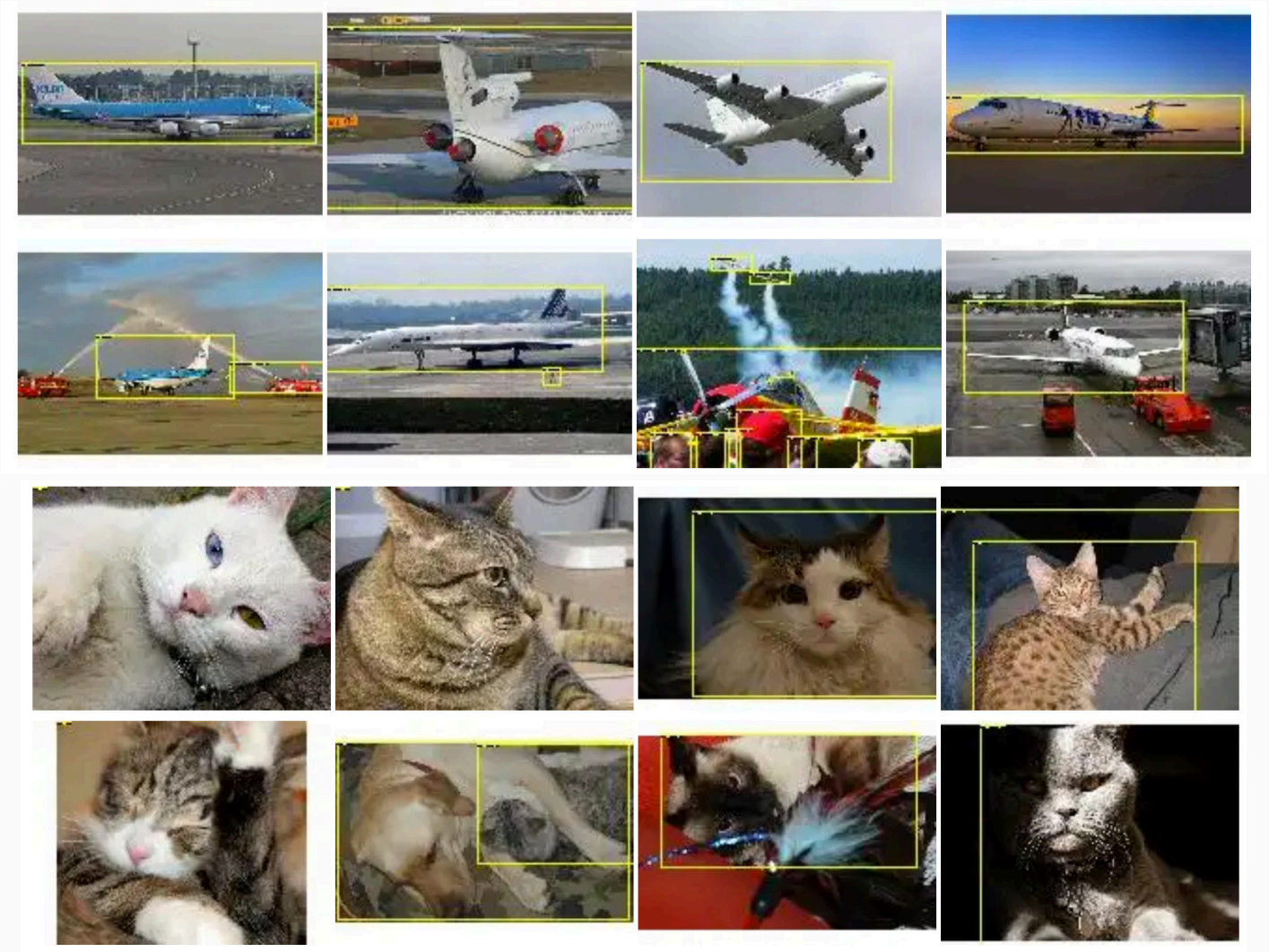
Постановка задачи и датасеты

- Классические статьи используют VOC07 (9963 изображений, 24640 объектов):



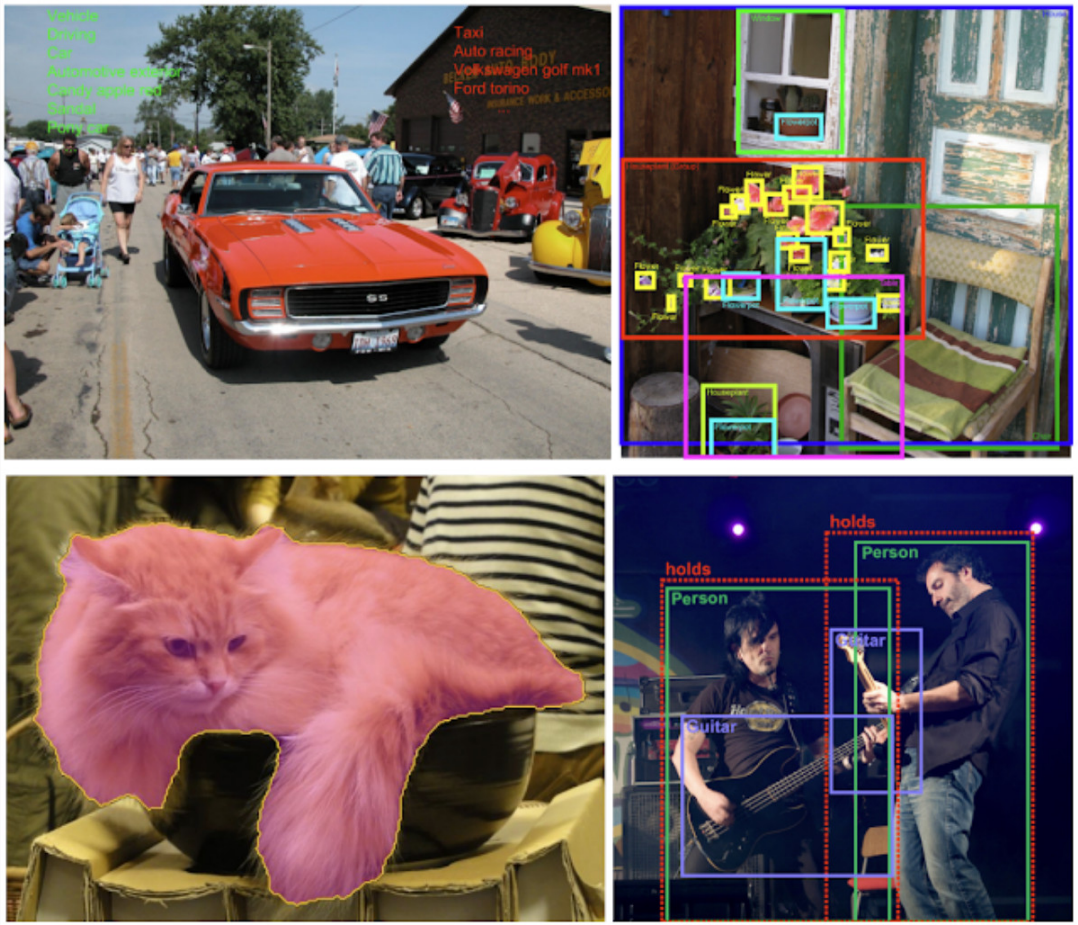
Постановка задачи и датасеты

- Или VOC12 (11540 изображений, 27450 объектов):



Постановка задачи и датасеты

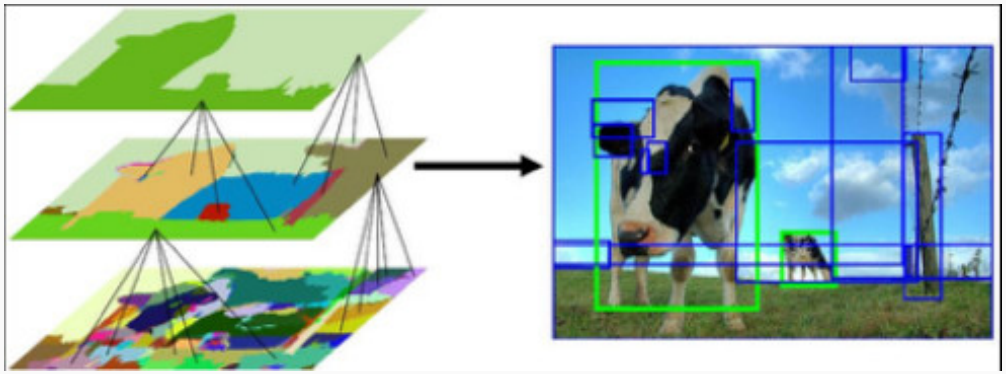
- Сейчас главный датасет — наверное, *OpenImages* от *Google*
- OpenImages V6: 15M bbox'ов, 2.7M сегментированных изображений, 60M меток



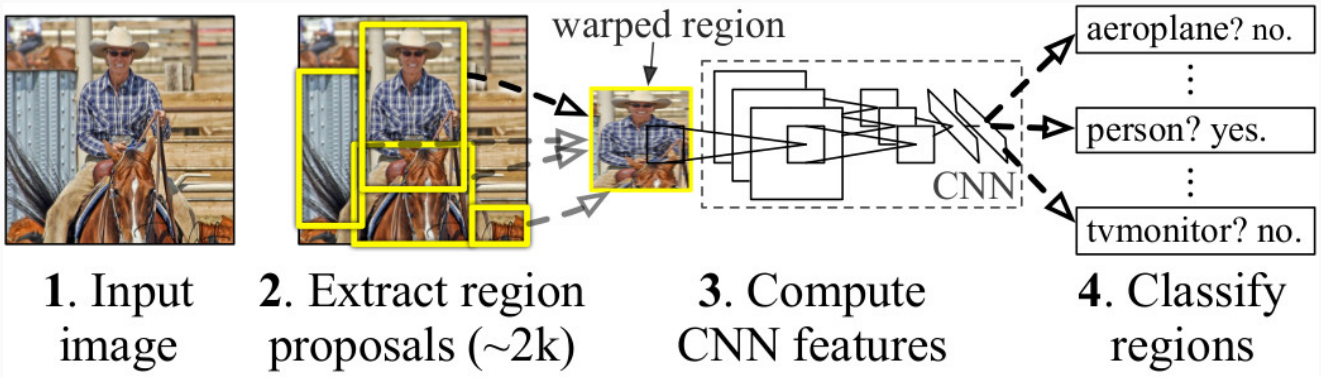
Двухэтапное распознавание объектов

R-CNN

- R-CNN: Region-based Convolutional Network method (Girshick et al., 2014).
- Идея: будем выделять участки каким-то внешним алгоритмом (например, selective search).



- Затем на них выделять признаки CNN (предобученной на ImageNet и fine-tuned на нужном) и классифицировать.



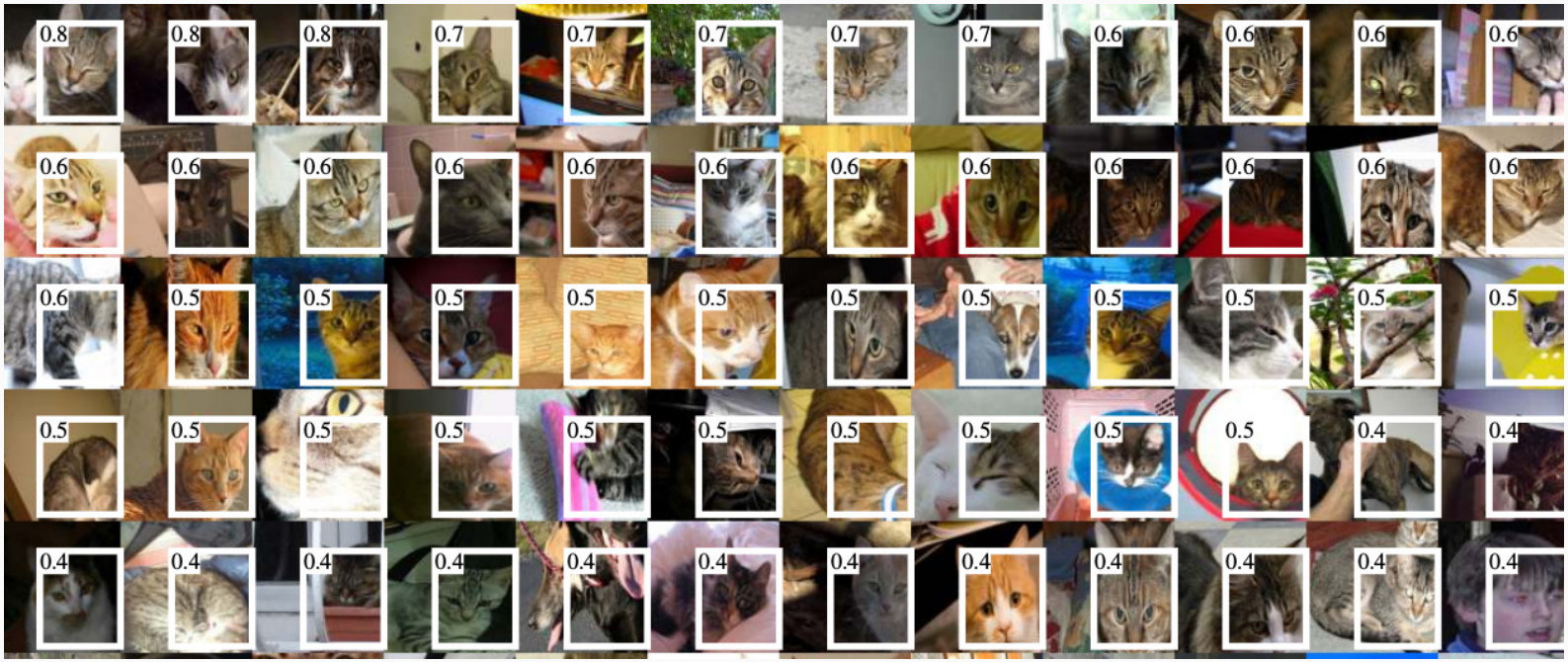
R-CNN

- Можно визуализировать: берём нейрон высокого уровня, показываем, на каких участках он активируется сильнее.



R-CNN

- Можно визуализировать: берём нейрон высокого уровня, показываем, на каких участках он активируется сильнее.

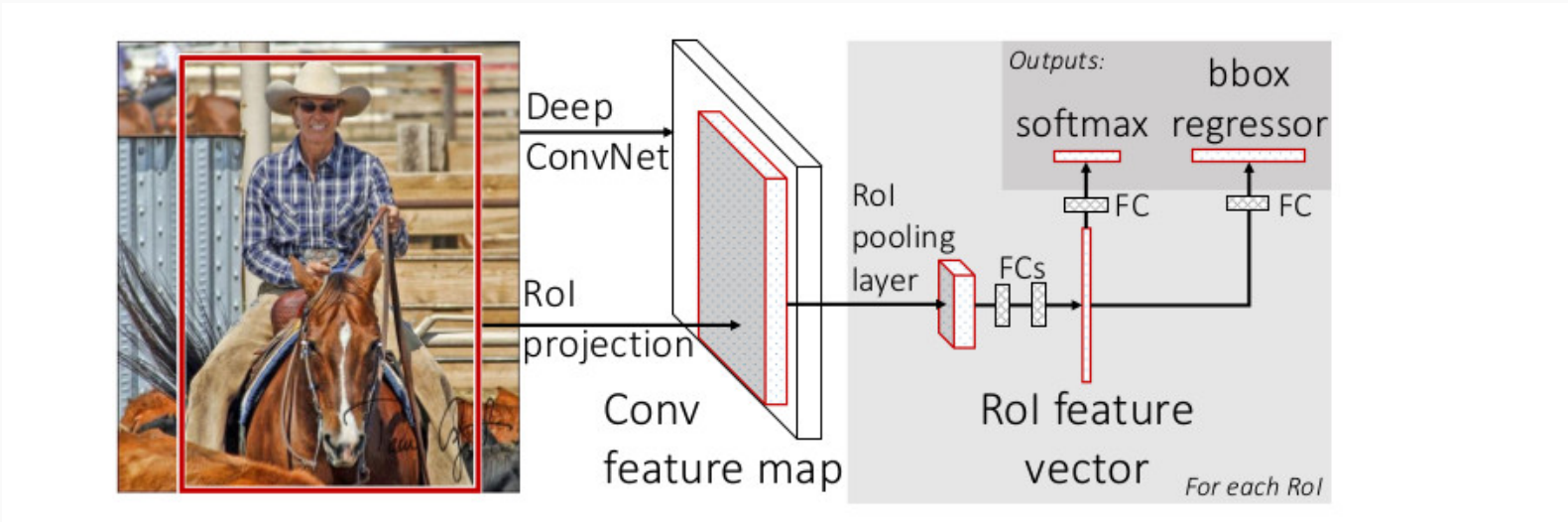


Fast R-CNN

- Но у R-CNN есть проблемы:
  - надо обучать в несколько приёмов (сначала CNN, потом SVM на её признаках, потом bounding box регрессоры)
  - обучать долго (надо выделить признаки из каждого участка в тренировочном датасете и записать на диск для обучения SVM и регрессоров);
  - распознавать очень медленно (47с на картинку на GPU).
- Причина в том, что для каждого участка надо делать проход по CNN.

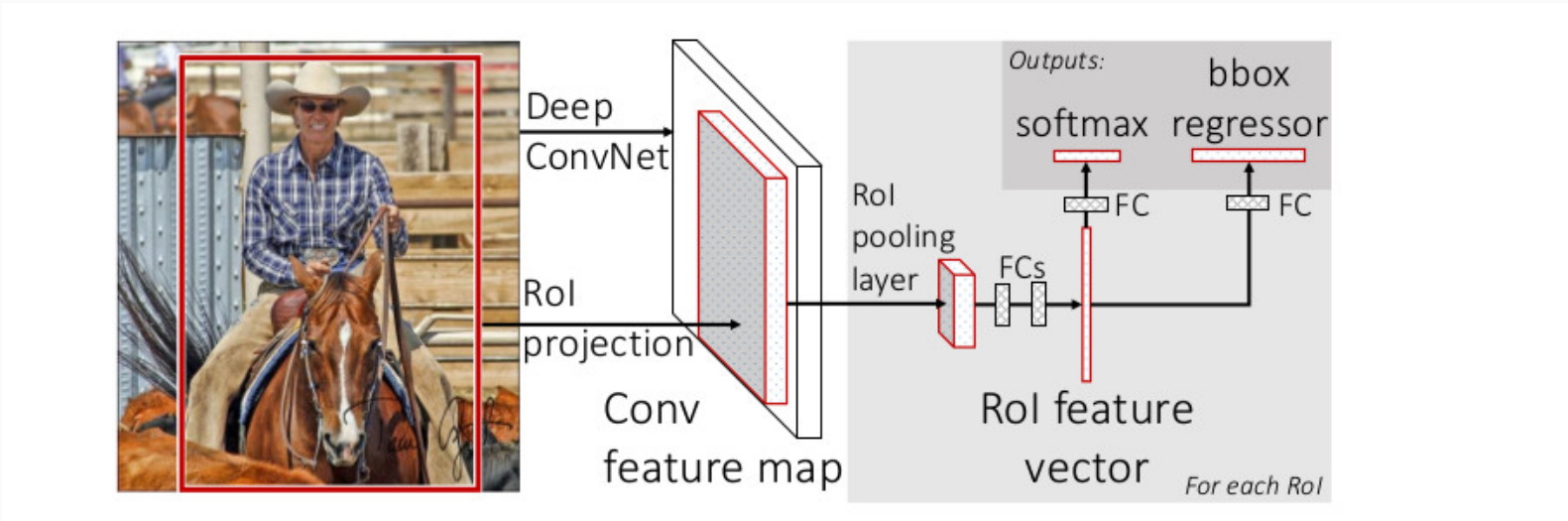
Fast R-CNN

- Поэтому – *Fast R-CNN*.
- RoI (region of interest) projection собирает признаки из заданного участка.
- Теперь нужно считать признаки для всей картинки только один раз.



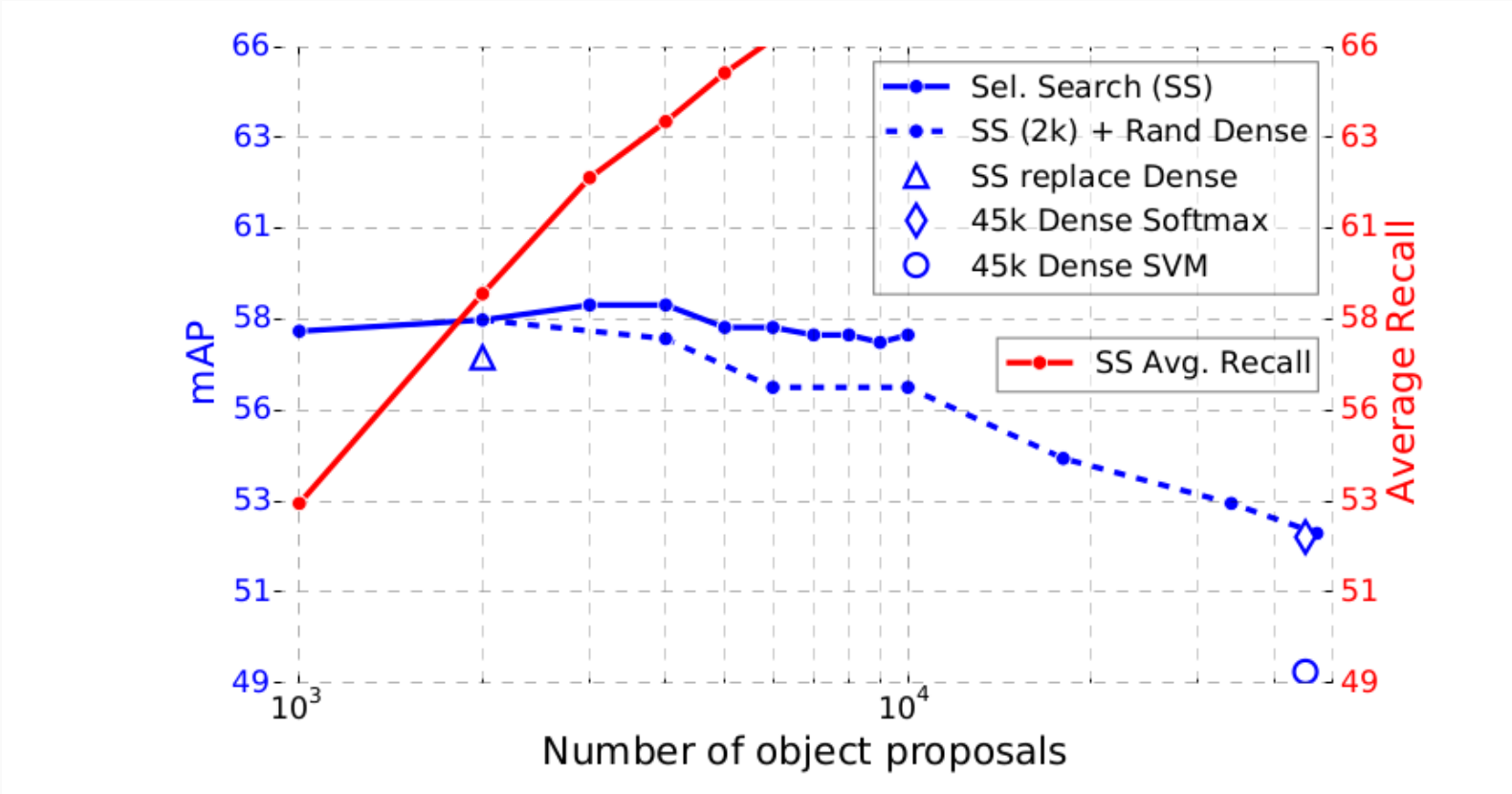
Fast R-CNN

- Функция потери складывается из двух: ошибка классификации и ошибка bounding box регрессии.
- Получается на два порядка быстрее и ничем не хуже (иногда лучше) по качеству.



Fast R-CNN

- Сами кандидаты в участки по-прежнему из selective search.
- Выяснилось, что добавлять больше кандидатов не очень помогает.

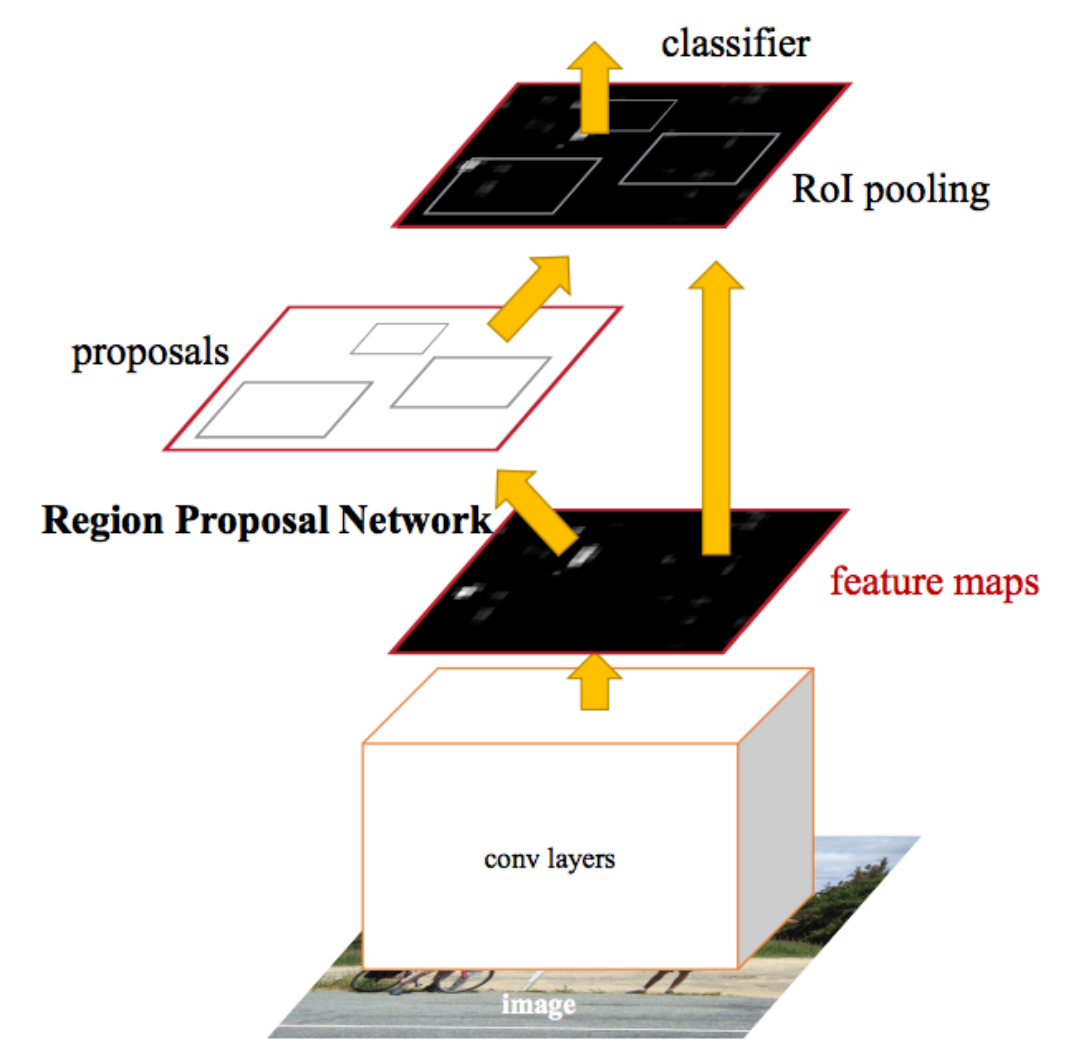


Faster R-CNN

- Осталось одно узкое место: selective search для выбора участков.
- Оказывается, это тоже можно вставить в сеть!
- Получается *Faster R-CNN* (Ren et al., 2015).

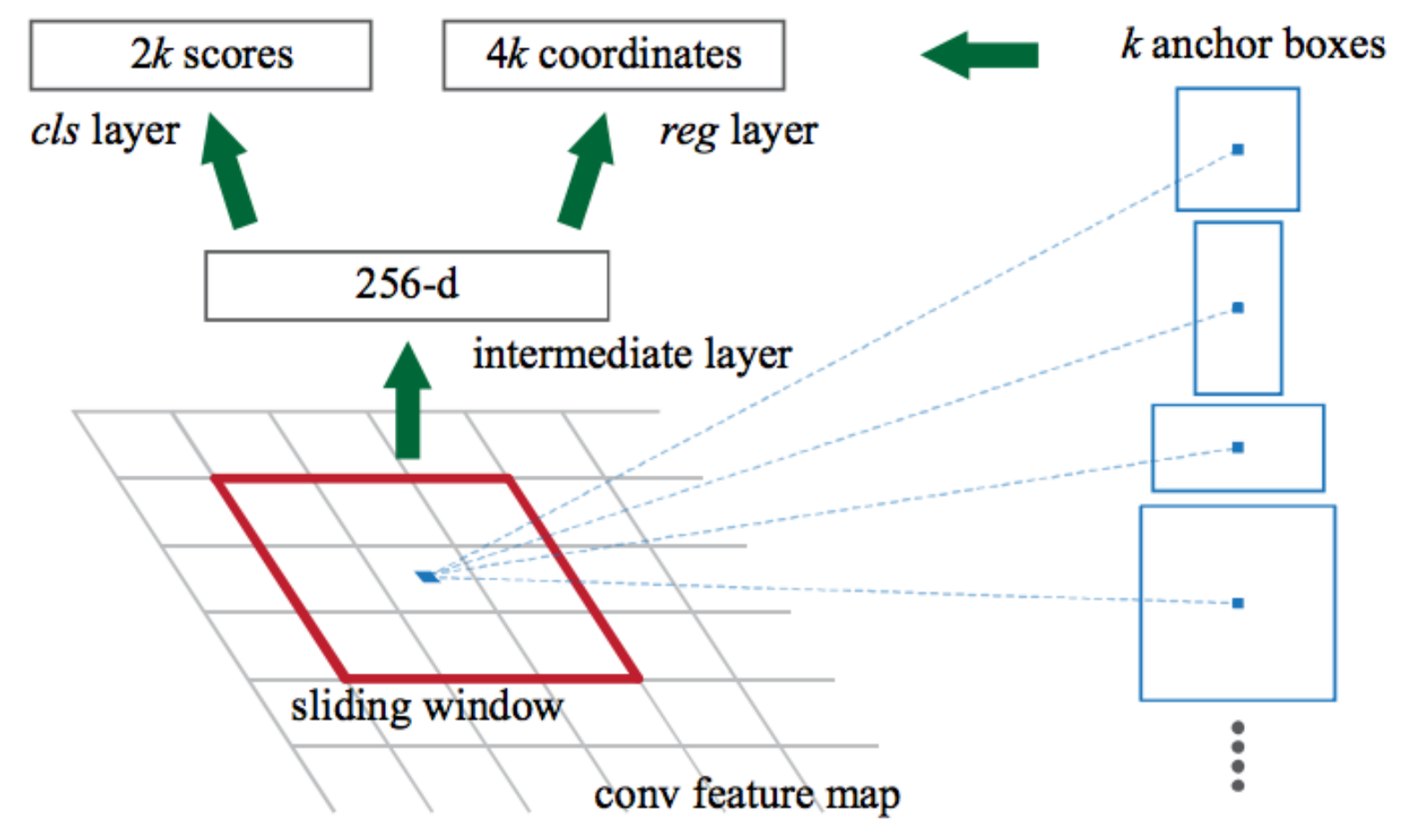
Faster R-CNN

- Отдельная Region Proposal Network:



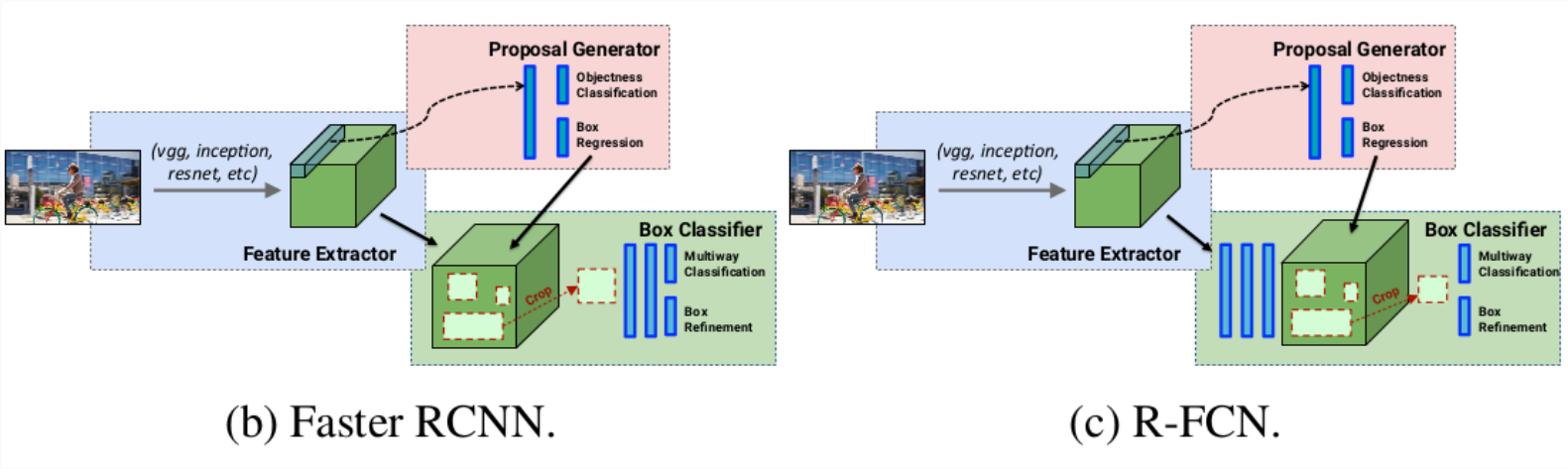
Faster R-CNN

- Оценивает каждое отдельное окно из существующих anchor boxes:



Faster R-CNN

- Можно и ещё больше сэкономить, не вычисляя на каждом участке вообще никаких сложных слоёв.
- R-FCN (Region-based Fully Convolutional Network) вырезает признаки из самого последнего перед классификацией слоя (Dai et al., 2016).
- Вот в чём разница:



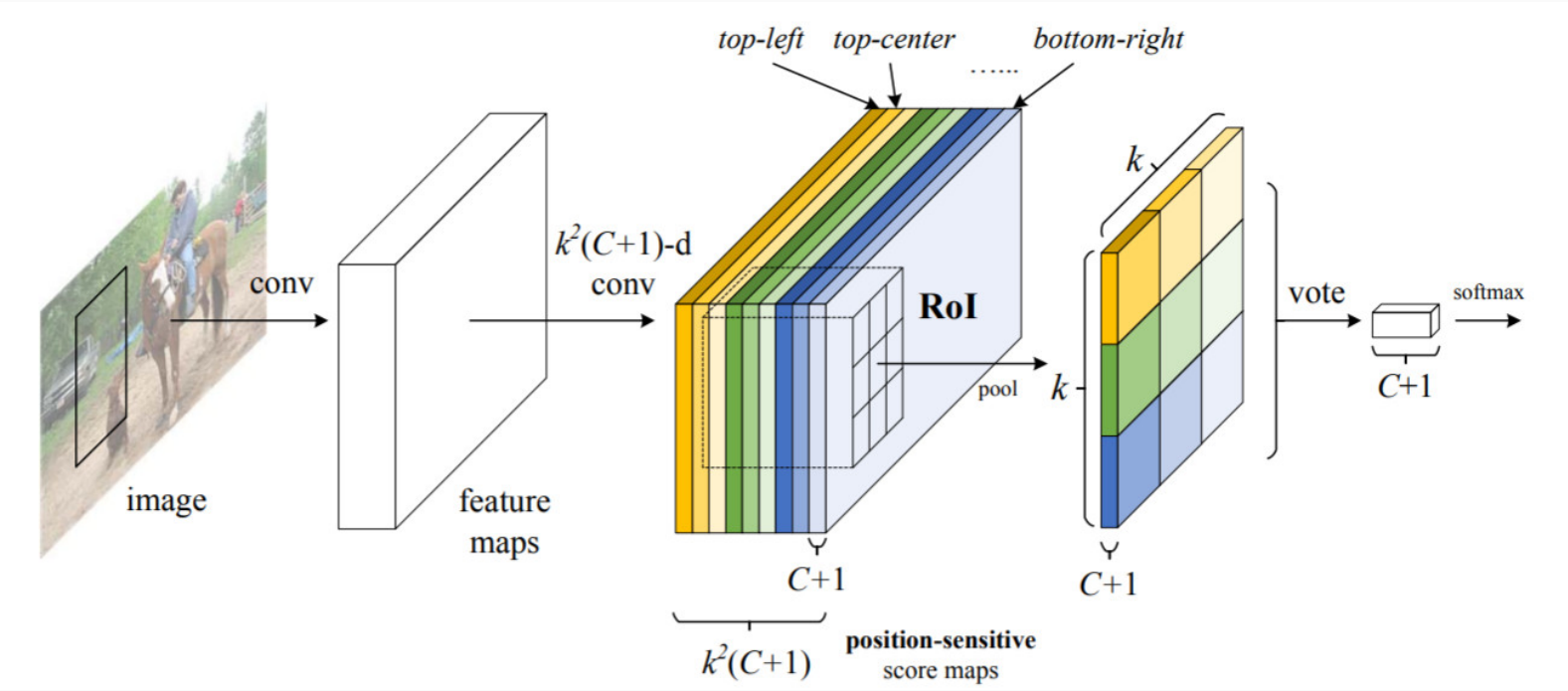
(b) Faster RCNN. (c) R-FCN.

R-FCN

- (Dai et al., 2016), R-FCN (Region-based Fully Convolutional Network)
- Можно ещё больше сэкономить в Faster R-CNN, не вычисляя на каждом участке вообще никаких сложных слоёв.
- Но как? Идея такая:
  - лучшие CNN для классификации (ResNet, GoogLeNet) давно fully convolutional, т.е. полносвязный слой только в конце
  - но с object detection почему-то это не работает
  - Faster R-CNN пришлось вставить противоестественный region proposal слой
  - проблема в том, что CNN дают translation invariance, а для object detection этого как раз не надо

R-FCN

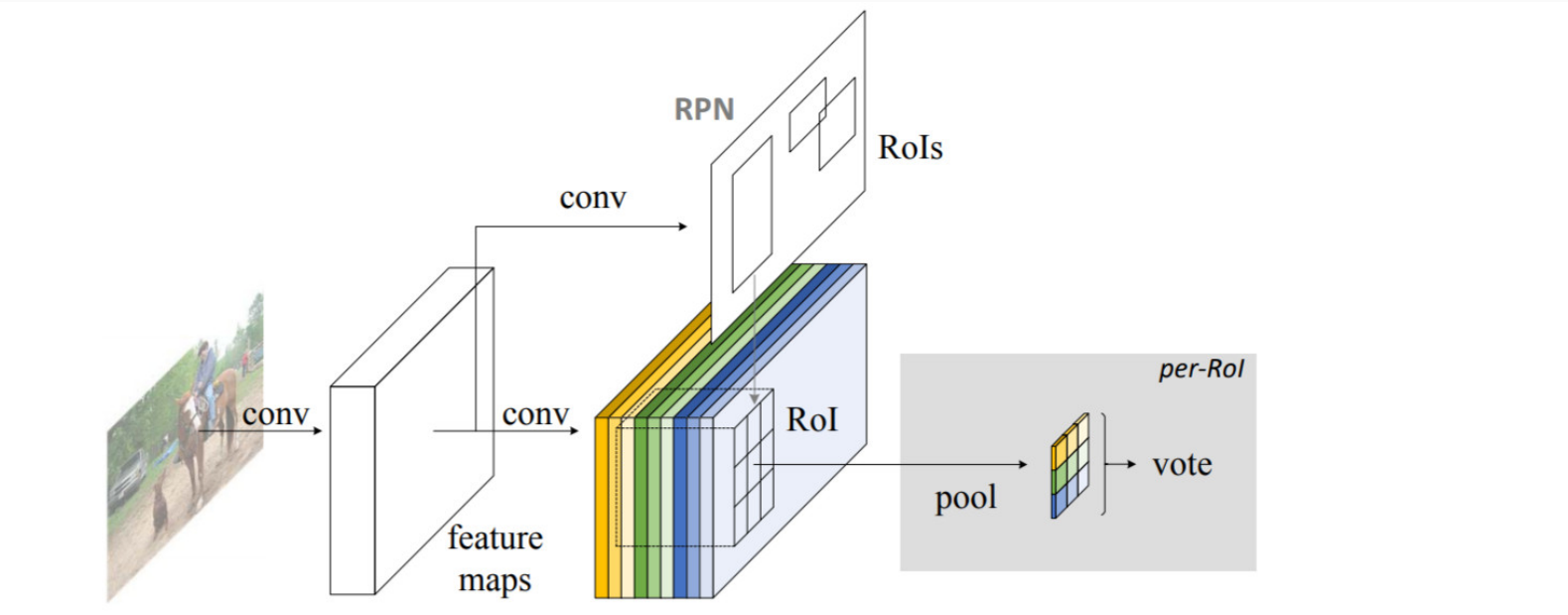
- Решение R-FCN в том, чтобы сделать специальные карты признаков, которые как раз position-sensitive:



- Пытаемся их обучить так, чтобы они кодировали информацию относительно конкретной позиции («левая часть объекта», «правая часть объекта»)

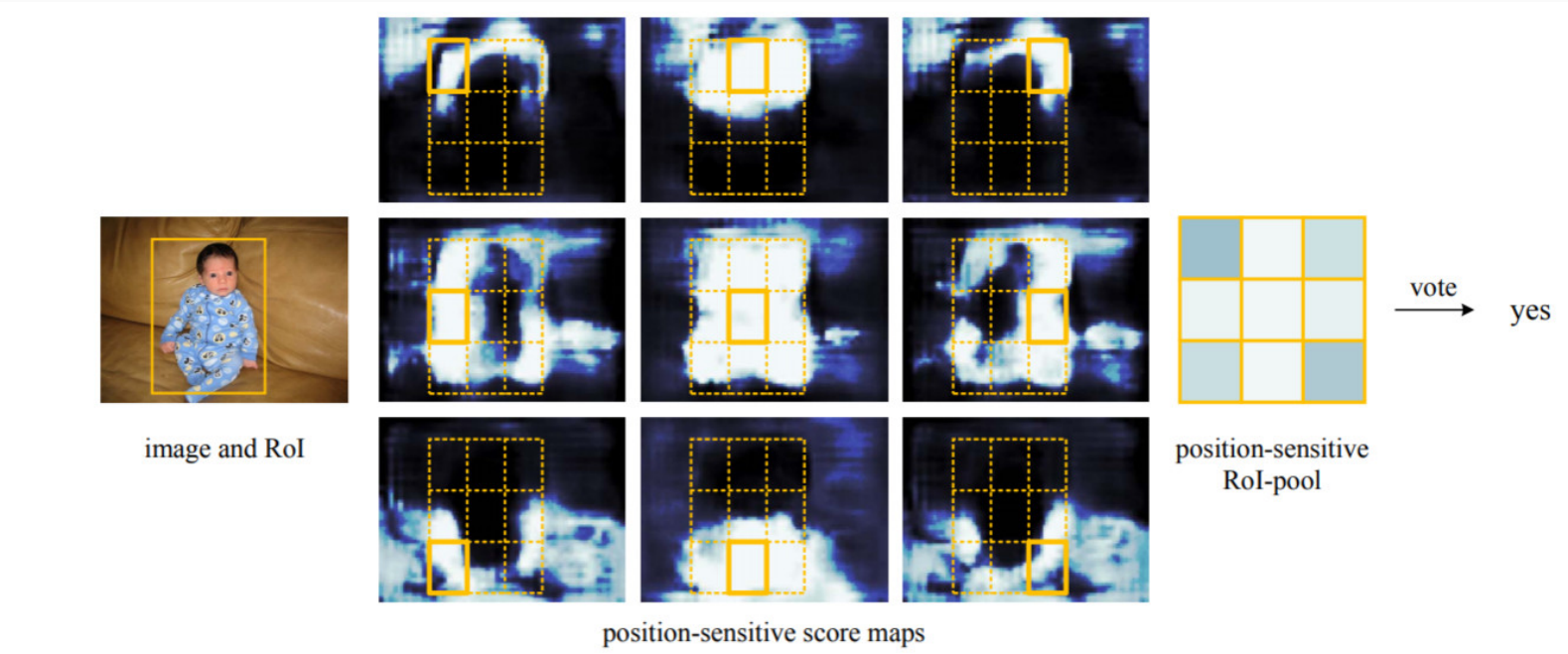
R-FCN

- В остальном как Faster R-CNN – есть отдельная RPN
- А потом RoI pooling, который учитывает наши новые feature maps:



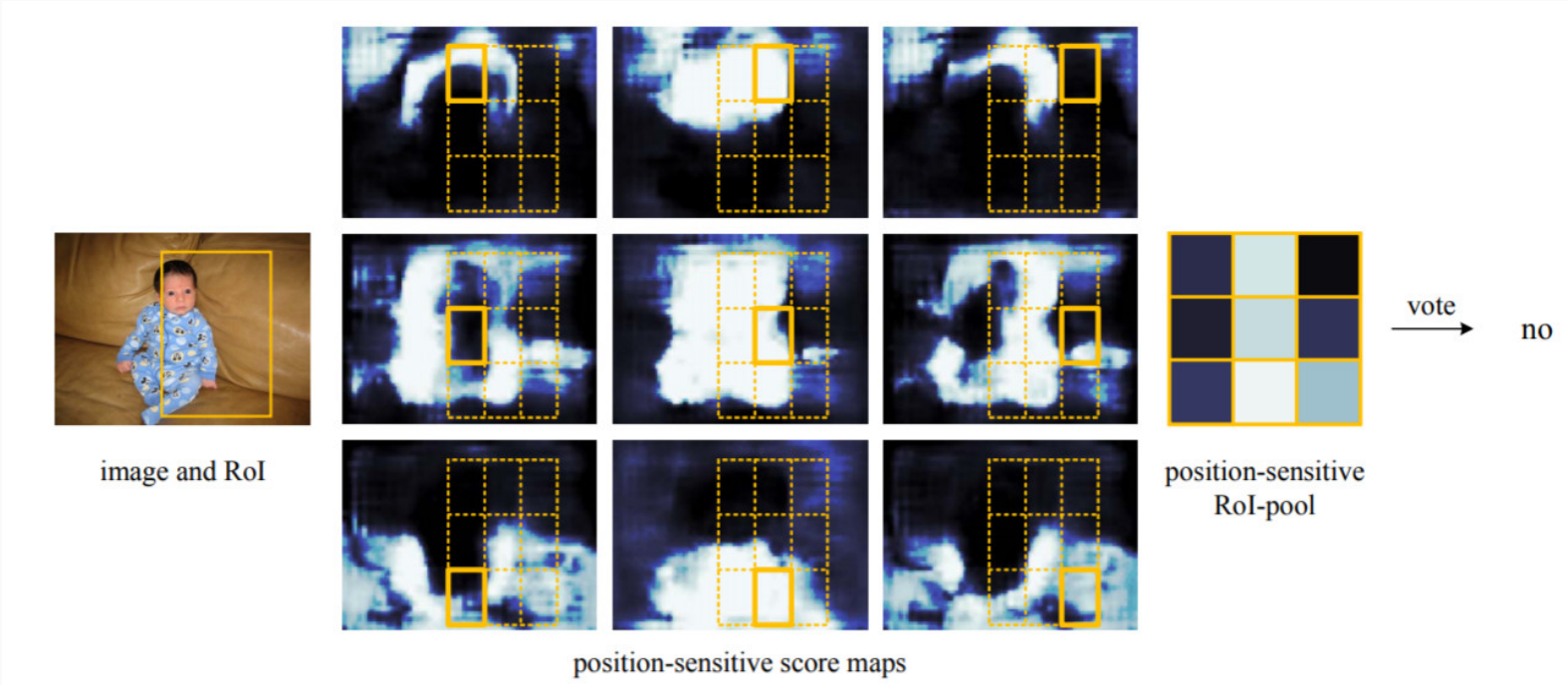
R-FCN

- Визуализация R-FCN активаций для 3×3 position-sensitive карт признаков:



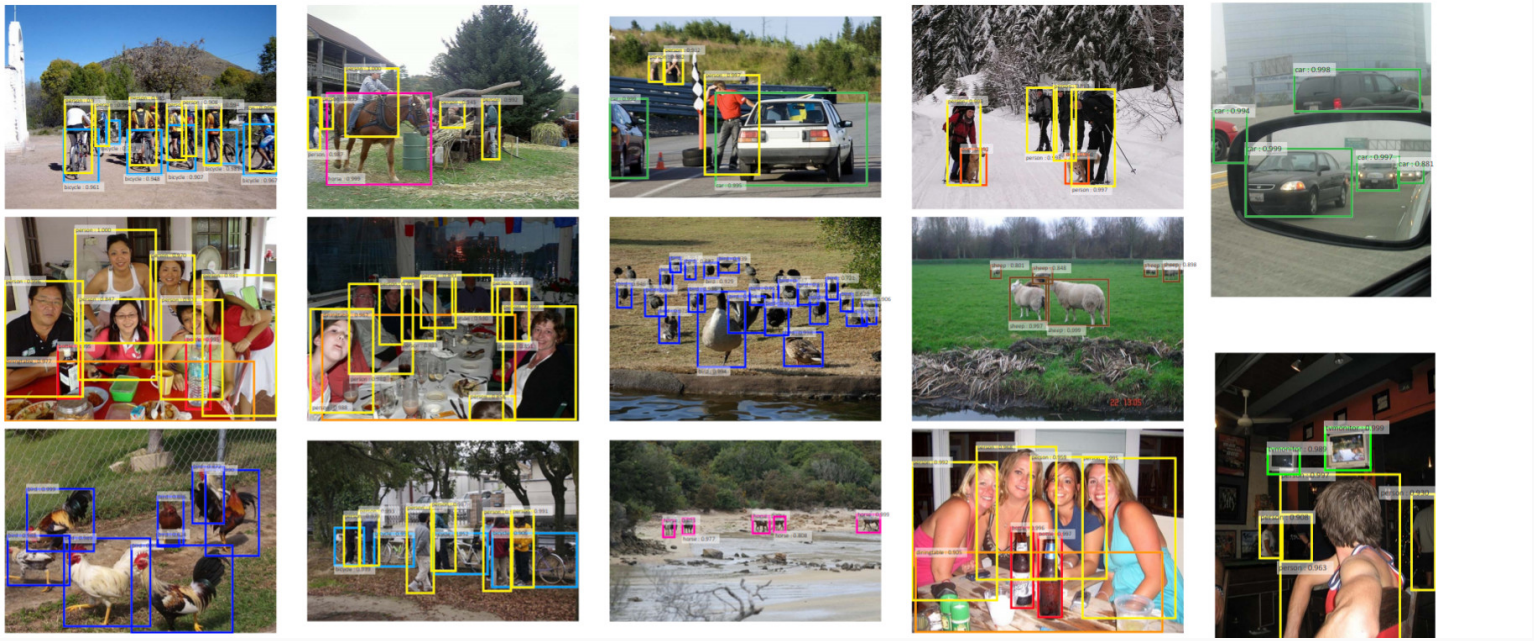
R-FCN

- А вот что будет для неправильного окна:



R-FCN

- Это неплохая архитектура и по сегодняшним меркам:



R-FCN

- Циферки тоже отличные:

	depth of per-RoI subnetwork	training w/ OHEM?	train time (sec/img)	test time (sec/img)	mAP (%) on VOC07
Faster R-CNN	10		1.2	0.42	76.4
R-FCN	0		0.45	0.17	76.6
Faster R-CNN	10	✓ (300 RoIs)	1.5	0.42	79.3
R-FCN	0	✓ (300 RoIs)	0.45	0.17	**79.5**
Faster R-CNN	10	✓ (2000 RoIs)	2.9	0.42	*N/A*
R-FCN	0	✓ (2000 RoIs)	0.46	0.17	79.3

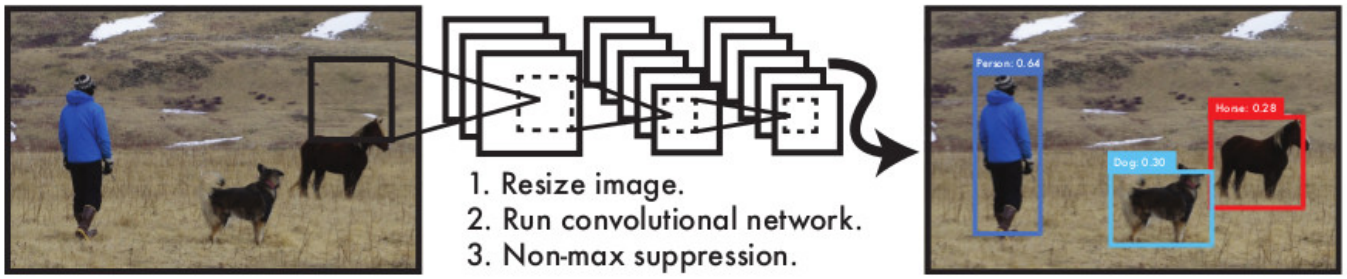
	training data	mAP (%)	test time (sec/img)
Faster R-CNN [9]	07+12	76.4	0.42
Faster R-CNN +++ [9]	07+12+COCO	**85.6**	3.36
R-FCN	07+12	79.5	0.17
R-FCN multi-sc train	07+12	80.5	0.17
R-FCN multi-sc train	07+12+COCO	**83.6**	0.17

	training data	mAP (%)	test time (sec/img)
Faster R-CNN [9]	07++12	73.8	0.42
Faster R-CNN +++ [9]	07++12+COCO	**83.8**	3.36
R-FCN multi-sc train	07++12	77.6†	0.17
R-FCN multi-sc train	07++12+COCO	**82.0**‡	0.17

Одноэтапное распознавание объектов

YoLo

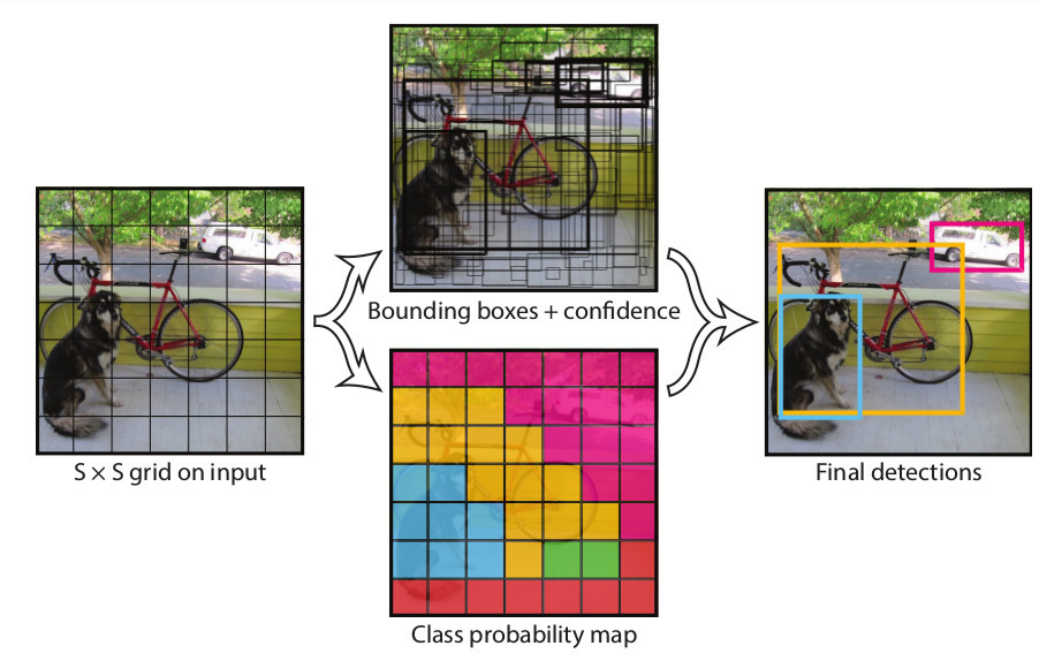
- You only look once (YOLO; Redmon et al., 2016).
- За один проход ищет и прямоугольники с объектами (bounding boxes), и сами объекты.



YoLo

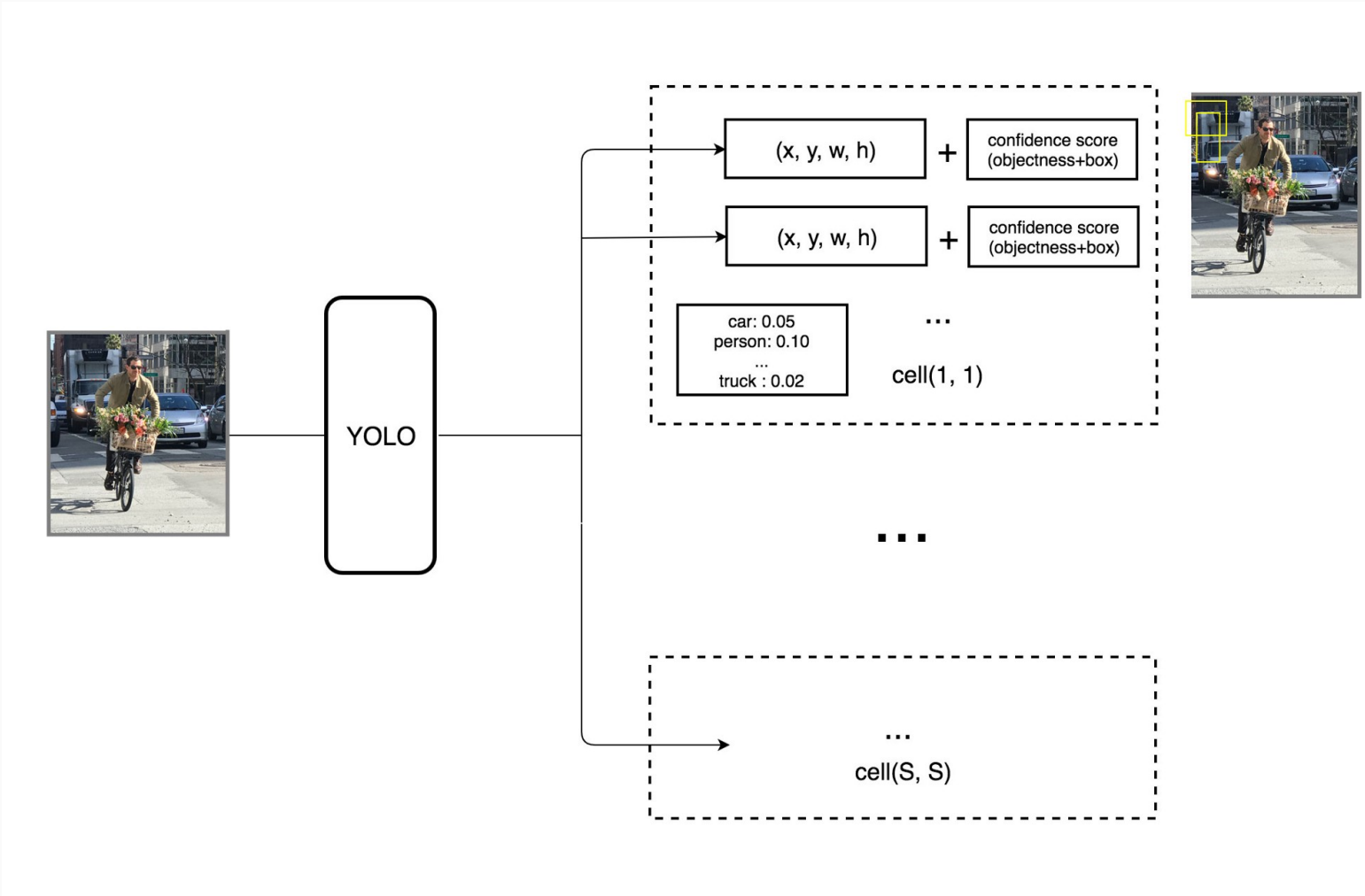
- Разбивает картинку на решётку $S \times S$.
- В каждой ячейке предсказывает и прямоугольники, и вероятности; и потом просто

$$p(\text{class}_i \mid \text{obj})p(\text{obj})p(\text{bbox}).$$



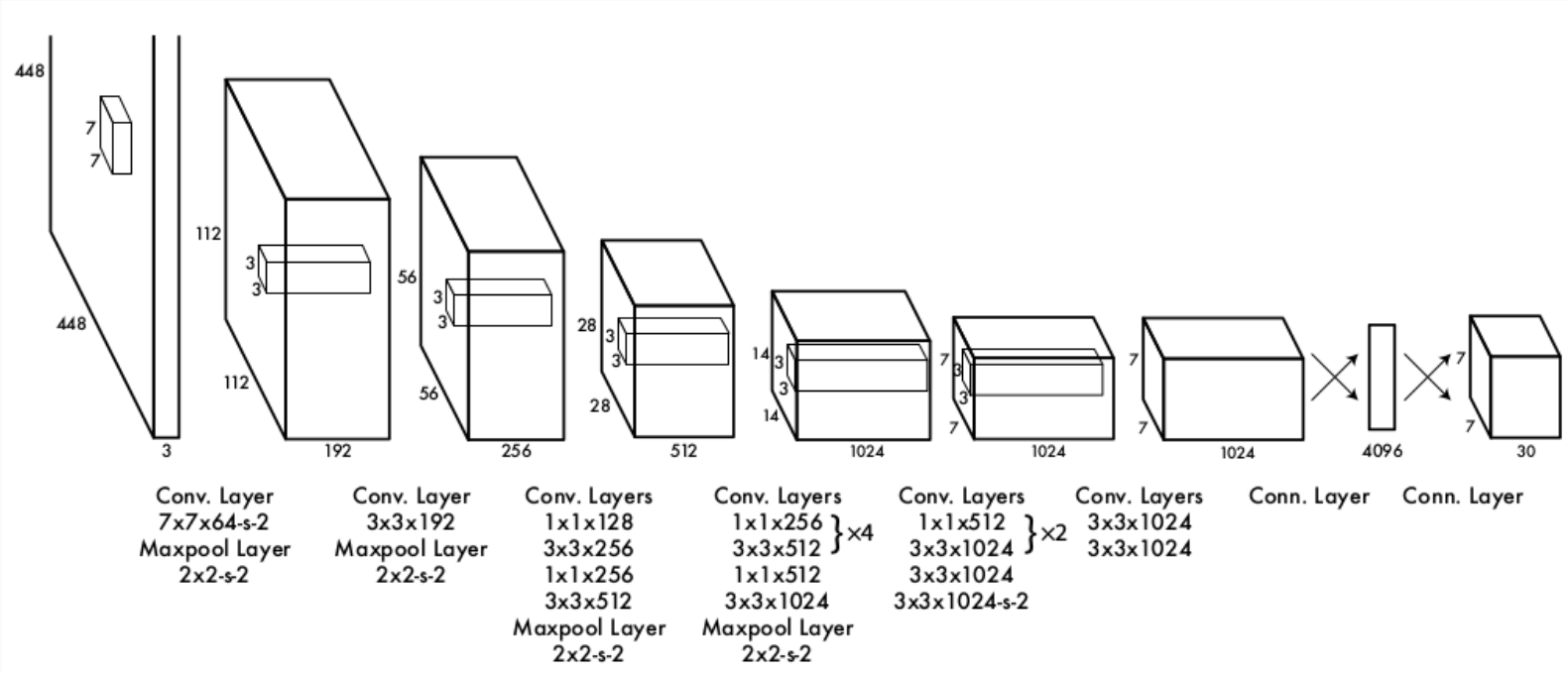
YoLo

- Вот так:


(x, y, w, h)
+
confidence score
(objectness+box)
(x, y, w, h)
+
confidence score
(objectness+box)
car: 0.05
person: 0.10
...
truck : 0.02
...
cell(1, 1)
YOLO
...
...
cell(S, S)

YoLo

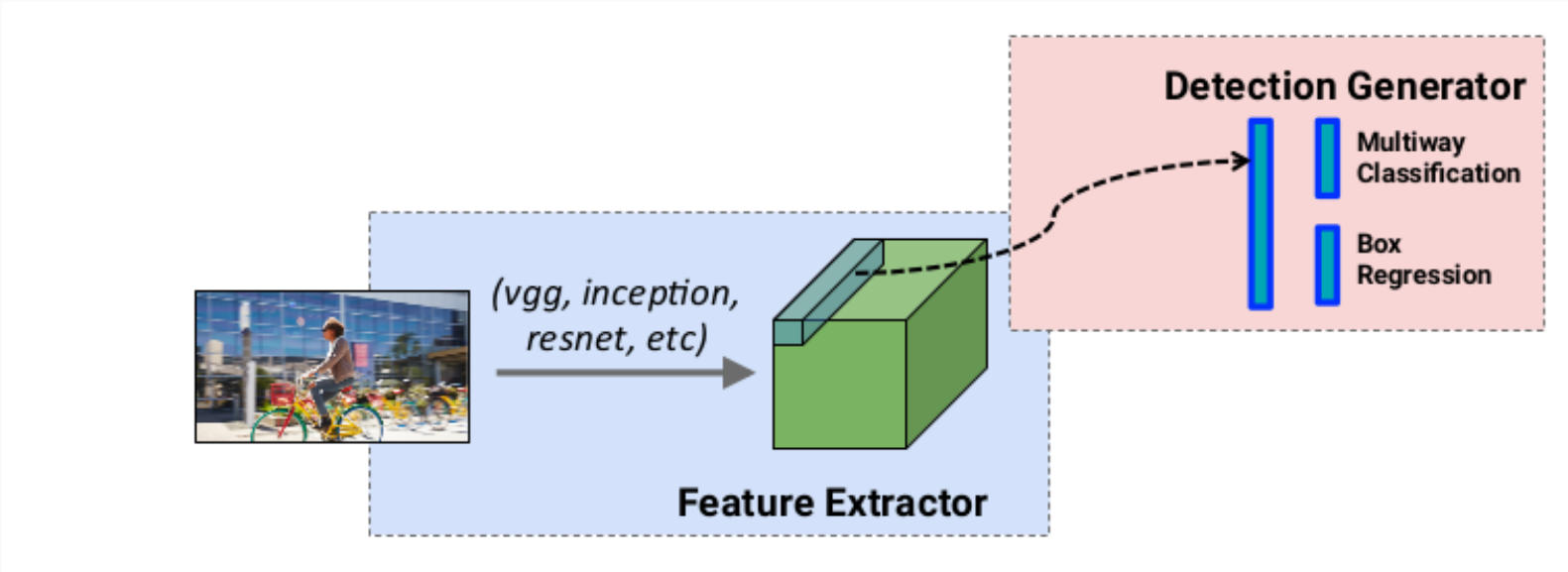
- Архитектура YOLO.



- Первый YOLO уступал по точности, но работал быстро.

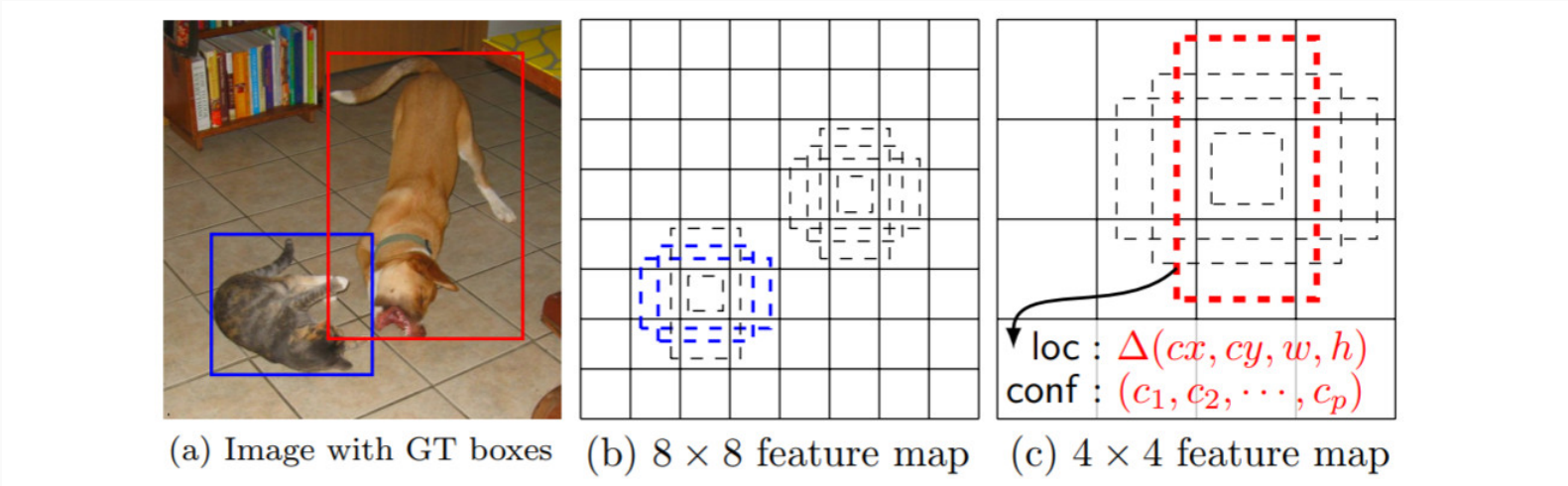
SSD

- Дальнейшее развитие идеи предсказывать всё сразу – single-shot detectors (SSD; Liu et al., 2016).
- Одна сеть, которая предсказывает
  - несколько меток классов;
  - несколько соответствующих позиций для anchor boxes (bounding boxes нескольких подходящих заранее выбранных размеров).



SSD

- Чтобы получить правильные ответы, мы сопоставляем стандартные anchor boxes с настоящими, и это правильные значения offsets и всего такого:

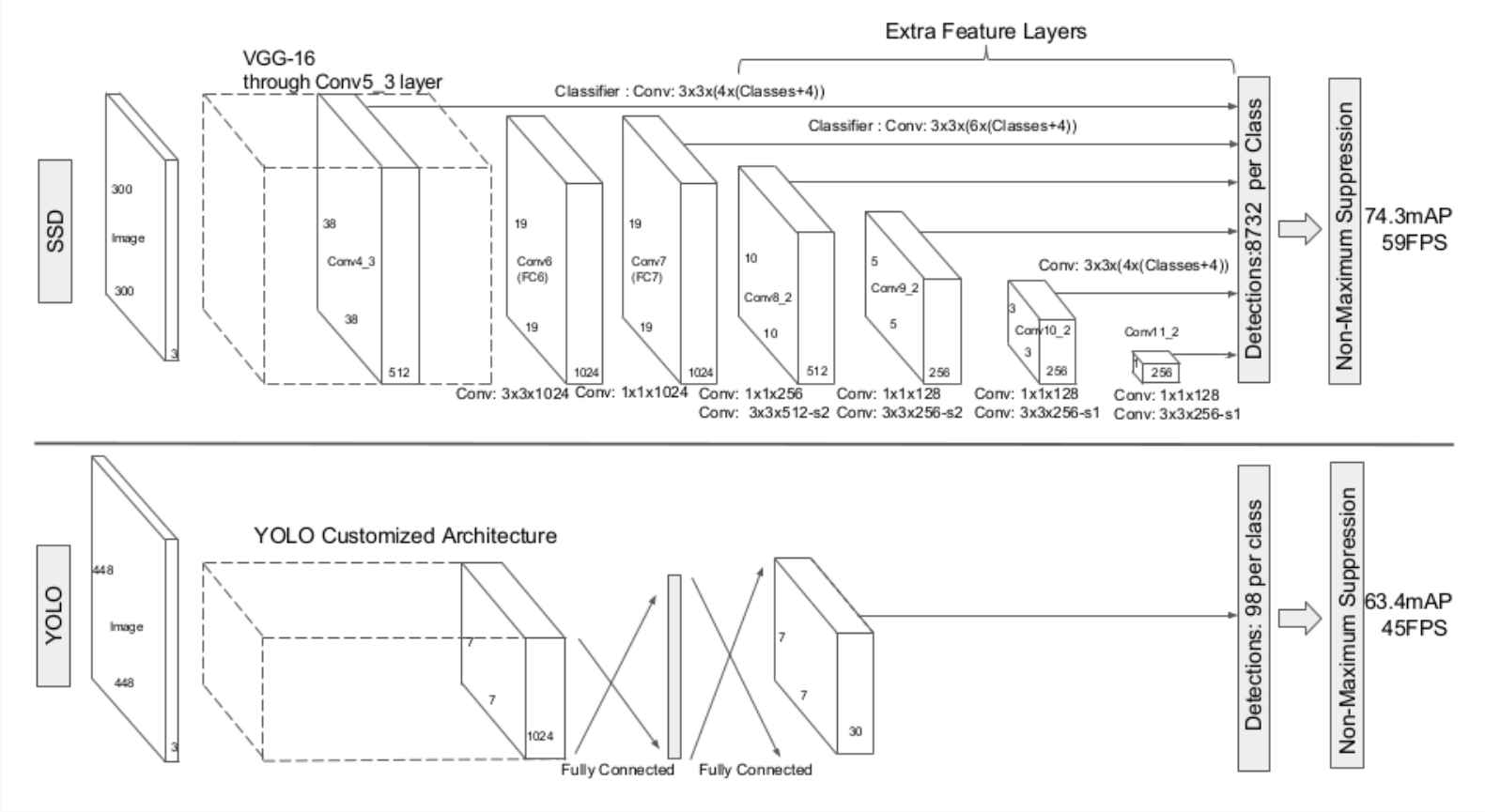


(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

- Целевая функция – как MultiBox, но с учётом классов

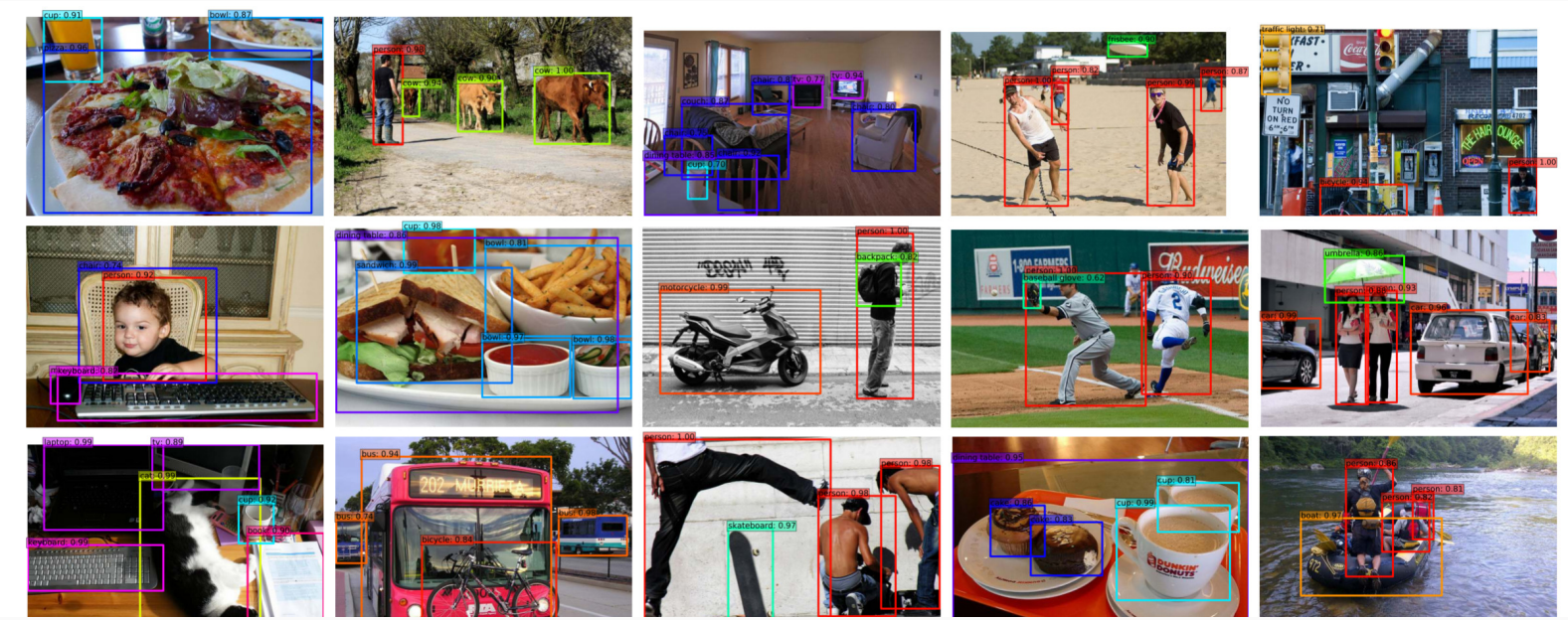
SSD

- Архитектура чуть посложнее YoLo, очень много выходов:



SSD

- Опять же, всё неплохо работает, но теперь уже везде всё будет неплохо работать:



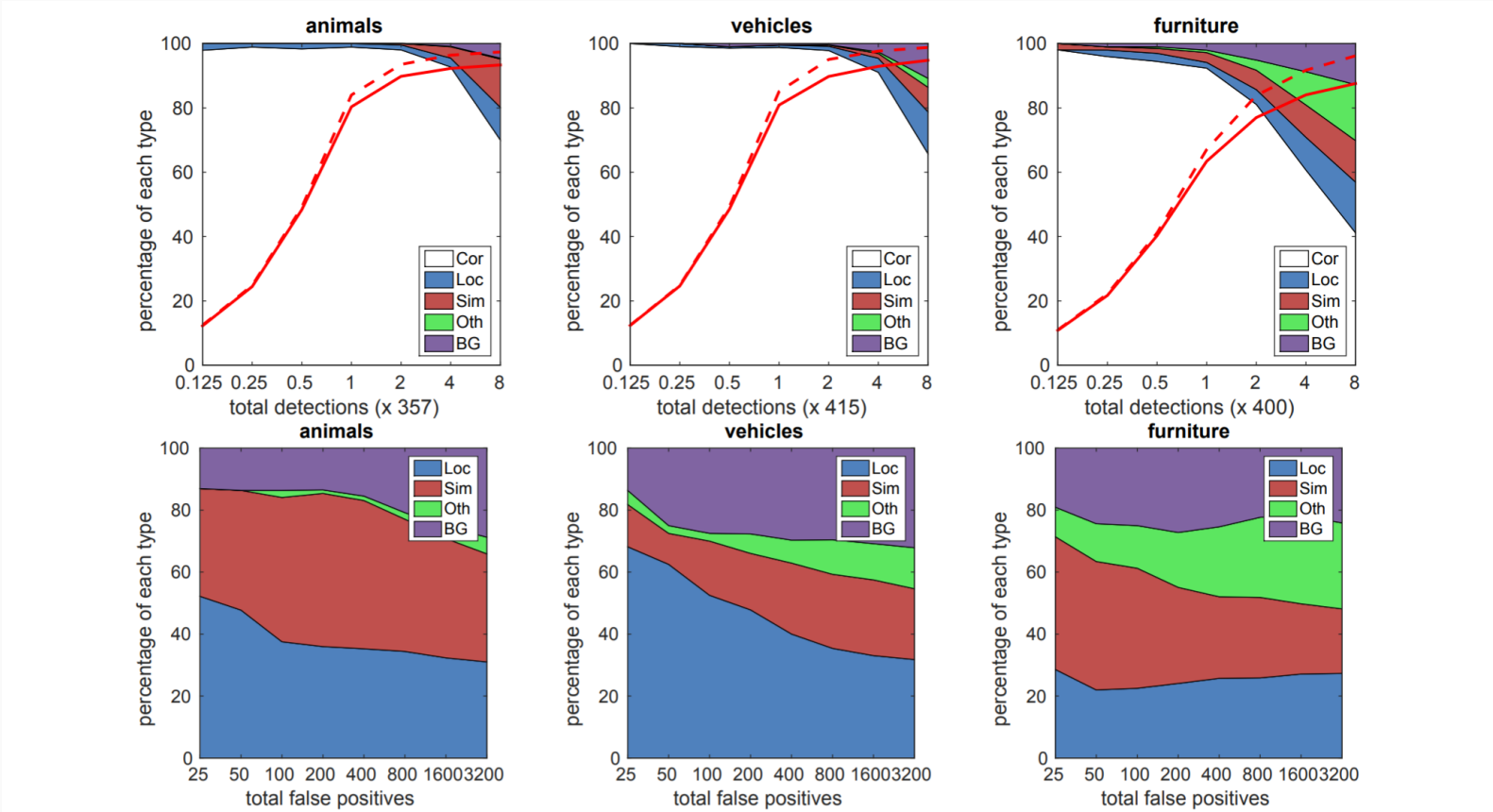
SSD

- На тот момент был единственным real-time детектором, который давал $> 70\%$ mAP на Pascal VOC2007:

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	$\sim 1000 \times 600$
Fast YOLO	52.7	155	1	98	448×448
YOLO (VGG16)	66.4	21	1	98	448×448
SSD300	74.3	46	1	8732	300×300
SSD512	76.8	19	1	24564	512×512
SSD300	74.3	59	8	8732	300×300
SSD512	76.8	22	8	24564	512×512

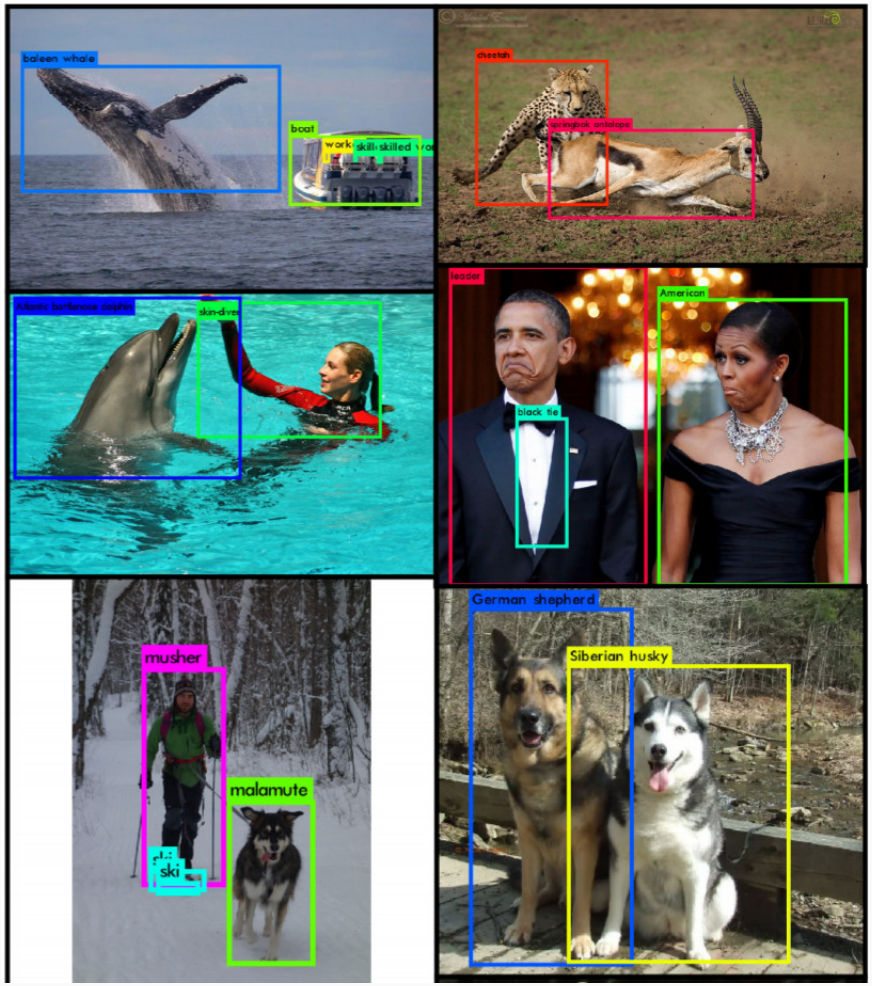
SSD

- Можно проанализировать, где ошибки возникают:



YoLo v2, YOLO9000, YoLo v3

- (Redmon, Farhadi, 2016): следующая итерация YoLo, гораздо больше классов (9000):



YoLo v2, YOLO9000, YoLo v3

- Что было не так: много ошибок локализации и низкий recall по сравнению с Faster R-CNN
- Что сделали:
  - добавили batchnorm на всех свёрточных уровнях, $+2\%$ mAP сразу
  - сделали отдельные классификатор высокого разрешения (448×448) вместо предобученного на ImageNet (256×256), ещё $+4\%$ mAP
  - YoLo предсказывал координаты bbox'ов, а лучше предсказывать offsets в каждой позиции свёрточным образом; класс теперь для каждого anchor box тоже предсказывается
  - размеры anchor boxes выбраны по bbox'ам из тренировочного набора, кластеризацией
  - добавили skip connection признаков с более раннего слоя, чтобы увеличить геометрическое разрешение
  - multi-scale training: обучаемся на разных размерах картинок

YoLo v2, YOLO9000, YoLo v3

- Получается очень круто:

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 288×288	2007+2012	69.0	91
YOLOv2 352×352	2007+2012	73.7	81
YOLOv2 416×416	2007+2012	76.8	67
YOLOv2 480×480	2007+2012	77.8	59
YOLOv2 544×544	2007+2012	**78.6**	40

YoLo v2, YOLO9000, YoLo v3

- И всё это реально помогает:

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?								✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

- По категориям:

Method	data	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
Fast R-CNN [5]	07++12	68.4	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	87.5	80.5	80.8	72.0	35.1	68.3	65.7	80.4	64.2
Faster R-CNN [15]	07++12	70.4	84.9	79.8	74.3	53.9	49.8	77.5	75.9	88.5	45.6	77.1	55.3	86.9	81.7	80.9	79.6	40.1	72.6	60.9	81.2	61.5
YOLO [14]	07++12	57.9	77.0	67.2	57.7	38.3	22.7	68.3	55.9	81.4	36.2	60.8	48.5	77.2	72.3	71.3	63.5	28.9	52.2	54.8	73.9	50.8
SSD300 [11]	07++12	72.4	85.6	80.1	70.5	57.6	46.2	79.4	76.1	89.2	53.0	77.0	60.8	87.0	83.1	82.3	79.4	45.9	75.9	69.5	81.9	67.5
SSD512 [11]	07++12	74.9	87.4	82.3	75.8	59.0	52.6	81.7	81.5	90.0	55.4	79.0	59.8	88.4	84.3	84.7	83.3	50.2	78.0	66.3	86.3	72.0
ResNet [6]	07++12	73.8	86.5	81.6	77.2	58.0	51.0	78.6	76.6	93.2	48.6	80.4	59.0	92.1	85.3	84.8	80.7	48.1	77.3	66.5	84.7	65.6
YOLOv2 544	07++12	73.4	86.3	82.0	74.8	59.2	51.8	79.8	76.5	90.6	52.1	78.2	58.5	89.3	82.5	83.4	81.3	49.1	77.2	62.4	83.8	68.7

YoLo v2, YOLO9000, YoLo v3

- А как сделать много классов? Новые проблемы: например, классы больше не взаимоисключающие («собака» и сотни пород собак)
- ImageNet берёт классы из WordNet – сети понятий, в которой есть иерархия, в виде направленного графа
- В YOLO сначала сделали из WordNet дерево, WordTree, а потом предсказывают условные вероятности на каждом уровне:

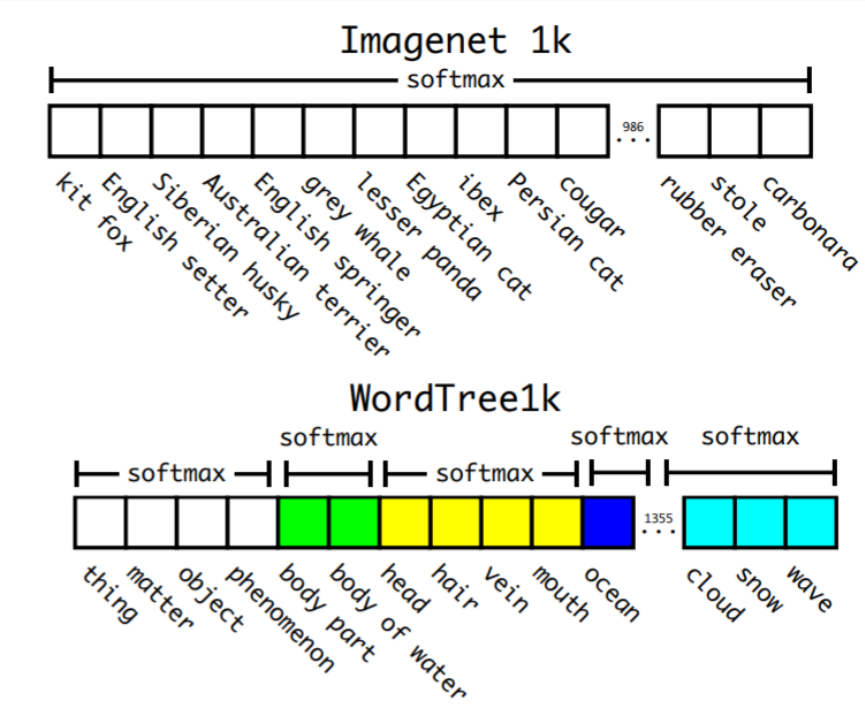
$$Pr(\text{Norfolk terrier}|\text{terrier})$$
$$Pr(\text{Yorkshire terrier}|\text{terrier})$$
$$Pr(\text{Bedlington terrier}|\text{terrier})$$
$$\ldots$$

и потом

$$Pr(\text{Norfolk terrier}) = Pr(\text{Norfolk terrier}|\text{terrier}) * Pr(\text{terrier}|\text{hunting dog}) * \ldots * Pr(\text{mammal}|Pr(\text{animal}) * Pr(\text{animal}|\text{physical object})$$

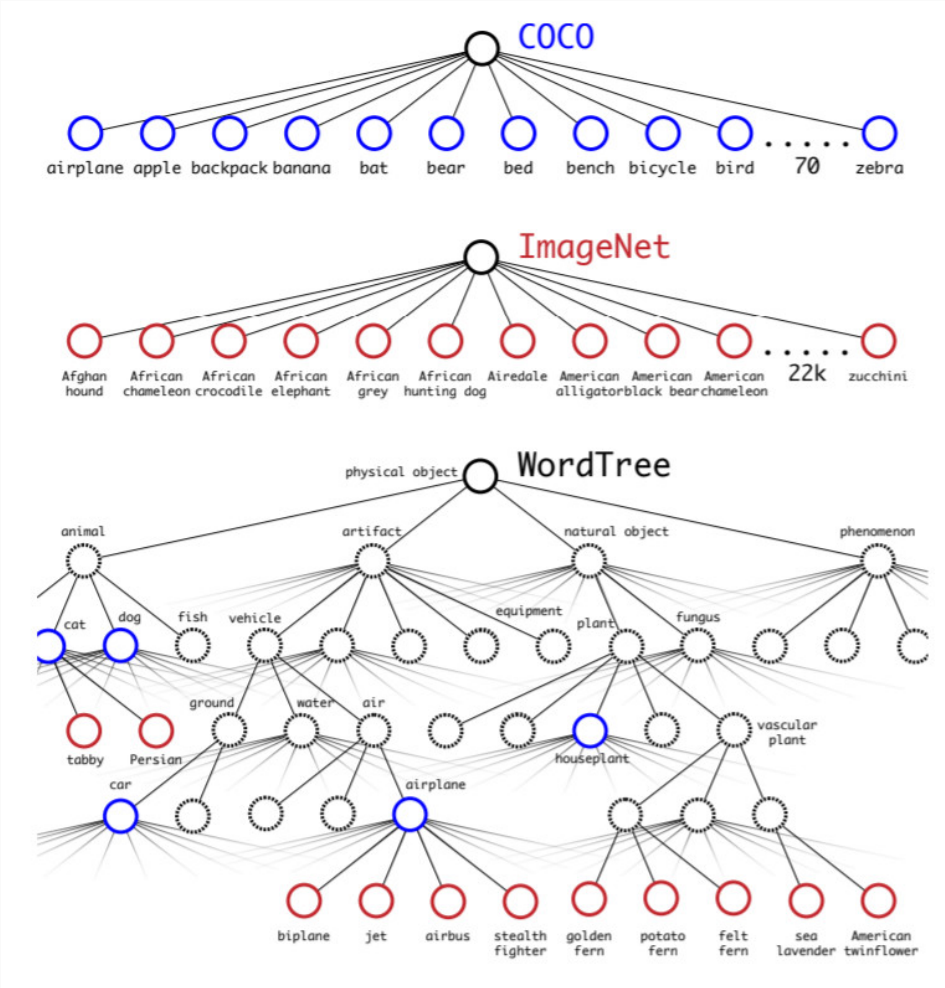
YoLo v2, YOLO9000, YoLo v3

- Это позволяет строить много маленьких softmax'ов, которые проще обучать:



YoLo v2, YOLO9000, YoLo v3

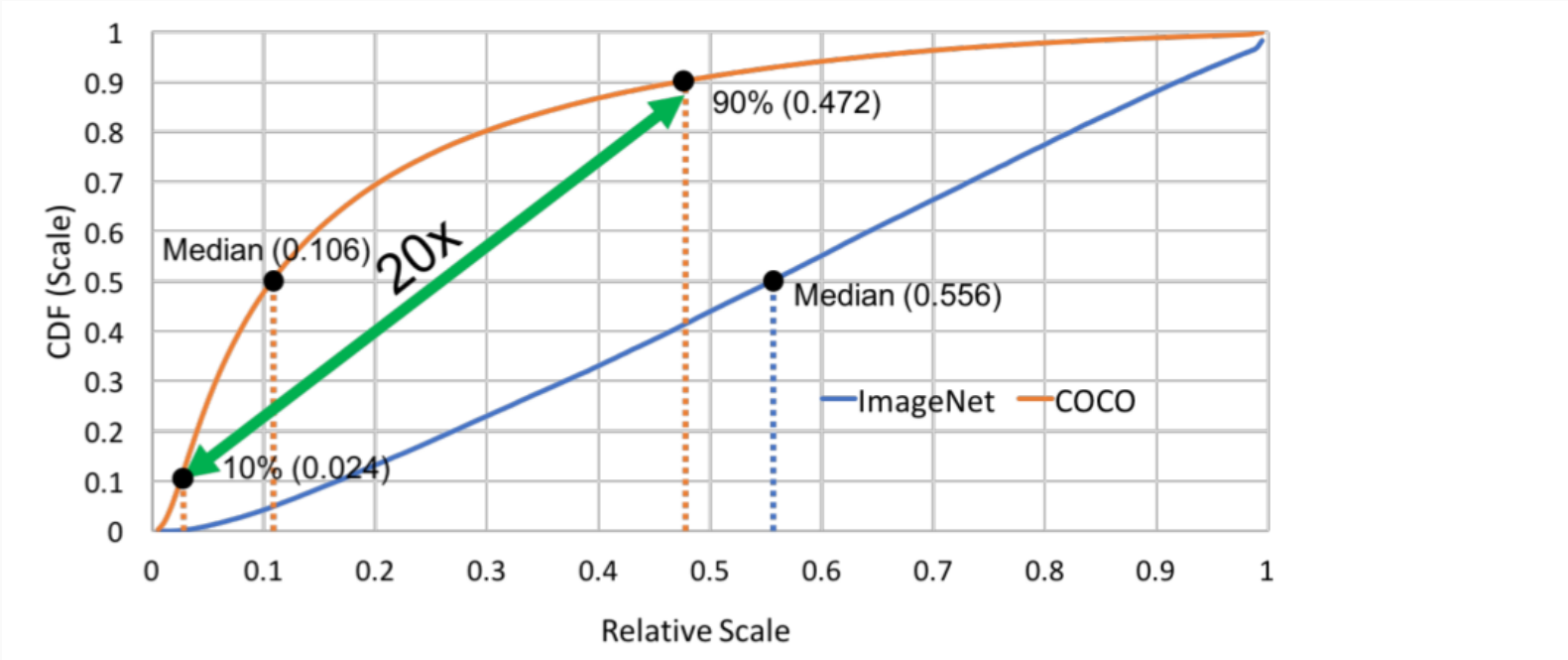
- И датасеты можно объединить даже



Проблемы с масштабом

Проблемы с масштабом

- Детекторы все хорошие, но почему-то распознают максимум 50-60% объектов. Почему на COCO такие плохие результаты?
- Потому что маленькие объекты плохо распознаются, а их в жизни много, и разброс масштабов очень большой.



- Почему так?

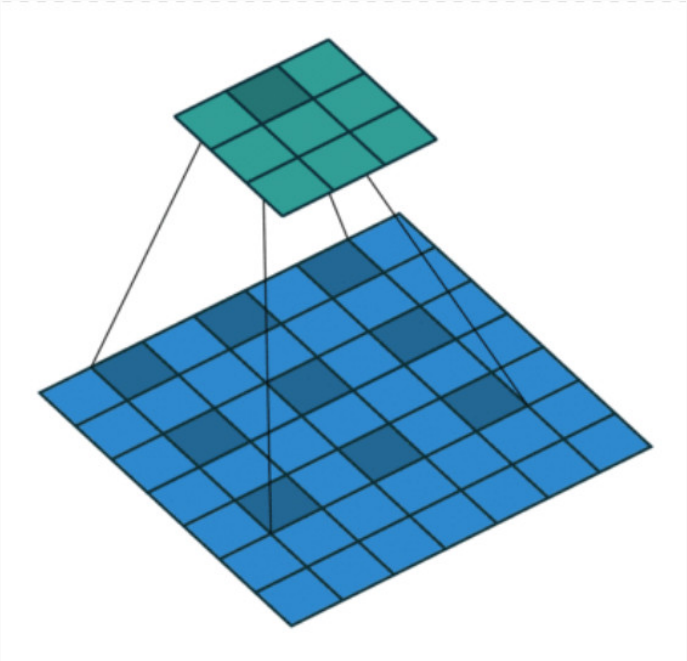
Проблемы с масштабом

- Effective stride: к моменту region proposal network уже геометрия исходной картинки сжимается.
- В базовой конструкции Faster R-CNN получается effective stride 16, т.е. от объекта 16×16 пикселей остаётся 1 пиксель.
- А если пытаться уменьшить stride, то сети выходят совсем гигантские и неподъёмные. Что делать?..



Проблемы с масштабом

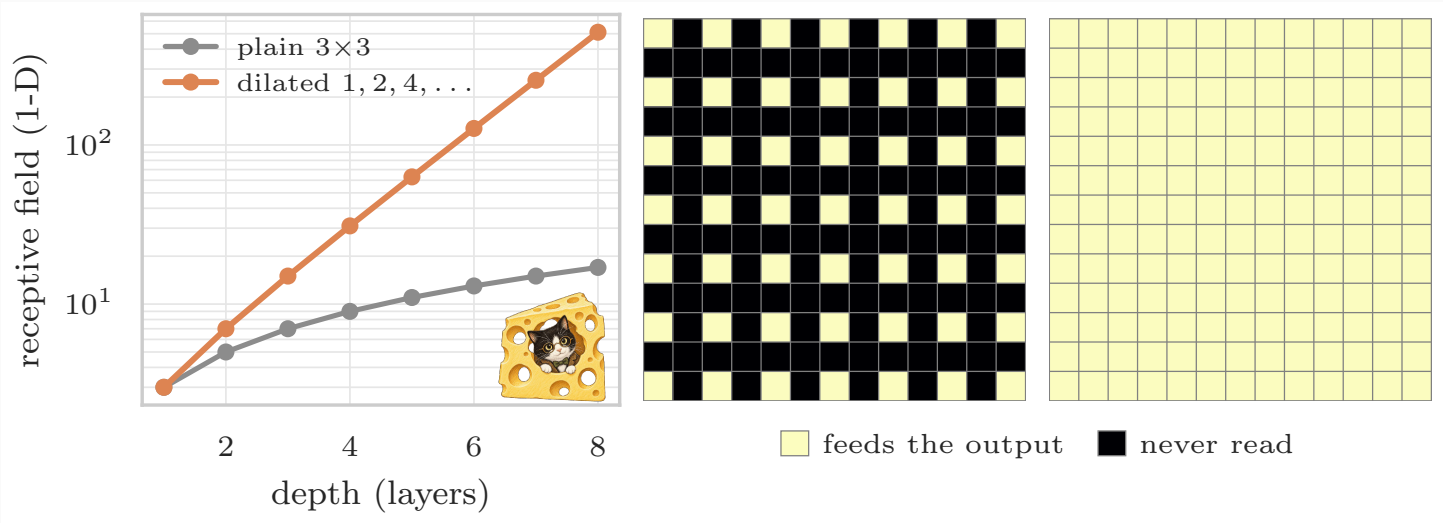
- Во-первых, свёртки в таких сетях часто немного другие. Dilated/atrous convolutions:



- Кстати: `https://github.com/vdumoulin/conv_arithmetic`
- Зачем нужны dilated convolutions?

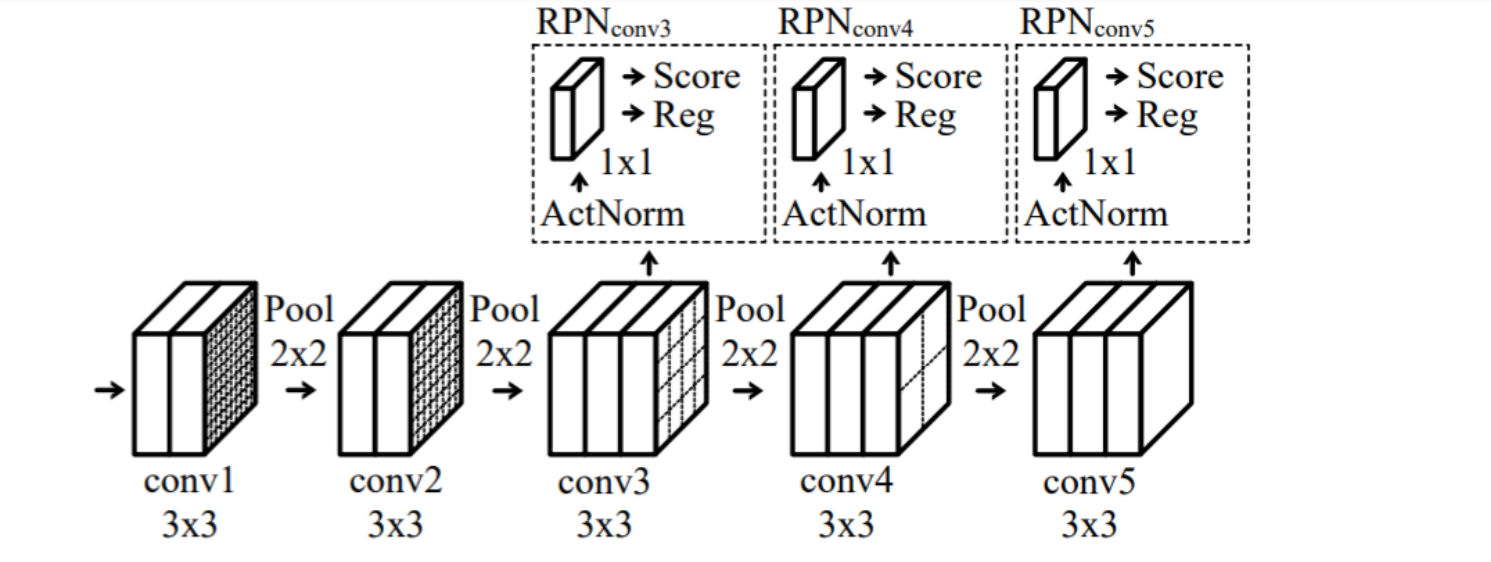
Проблемы с масштабом

- А чтобы покрыть гораздо большее receptive field за меньшее число слоёв: с темпами $1, 2, 4, \ldots$ оно растёт с глубиной экспоненциально, а не линейно (слева), и без единого лишнего веса.
- Но если у всех слоёв один и тот же темп, получается решётка, где три четверти входов никогда не читаются (в центре, темпы $2, 2, 2$); лечится темпами без общего делителя ($1, 2, 3$, справа):



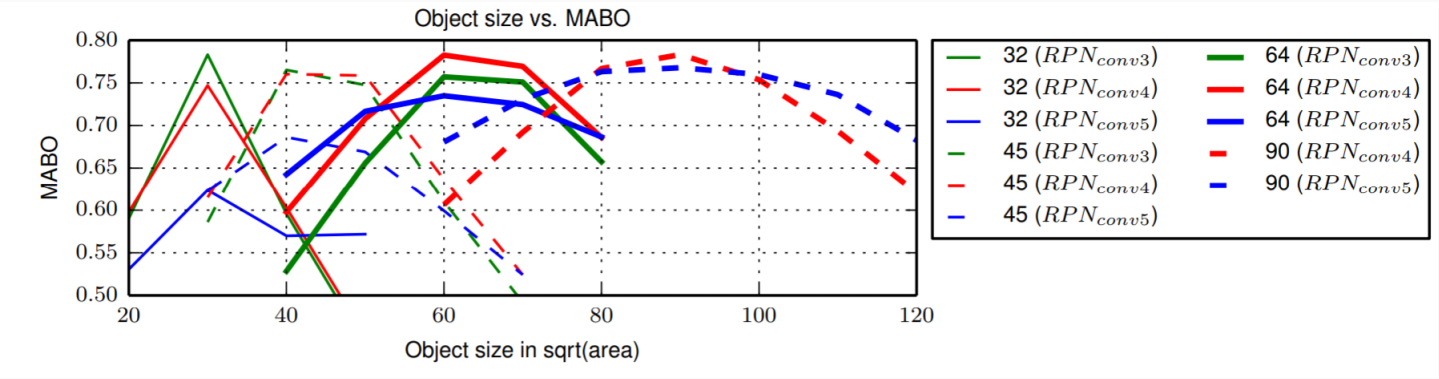
Проблемы с масштабом

- Dilated convolutions используются в Faster R-CNN, R-FCN и других.
- (Yu, Koltun, 2016): Multi-scale context aggregation by dilated convolutions; собирают весь контекст с разных dilated convolutions вместе, применяют к сегментации.
- Можно ещё пытаться сделать несколько разных RPN; (Eggert et al., 2017) – возьмём базовый Faster R-CNN и встроим в него три RPN с разным масштабом.

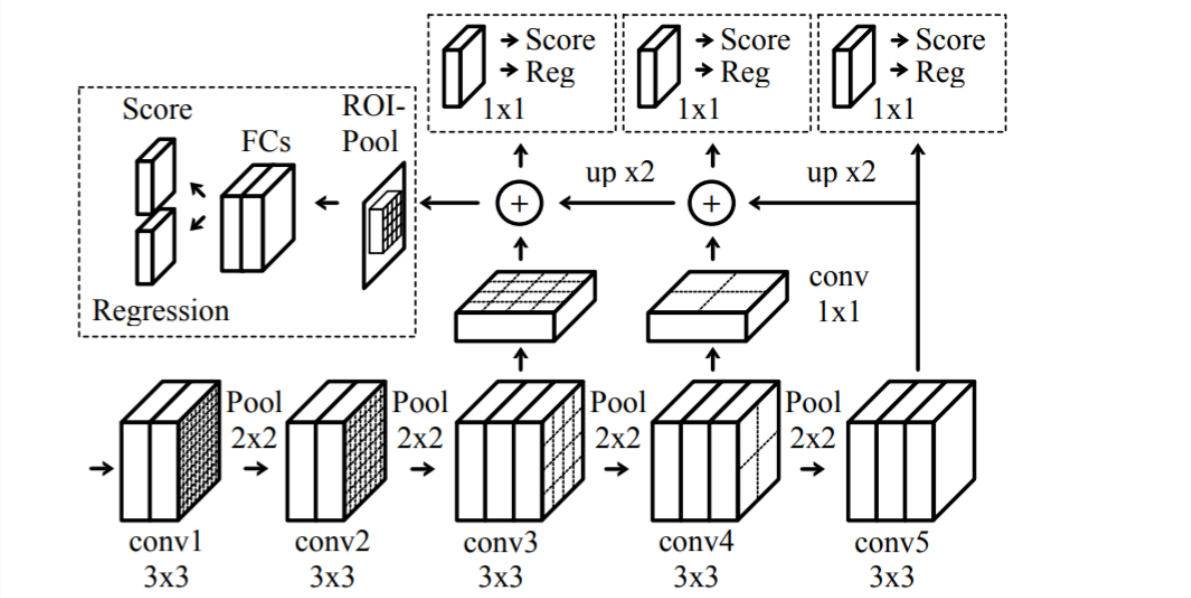


Проблемы с масштабом

- Разные RPN нужны для объектов разного размера:

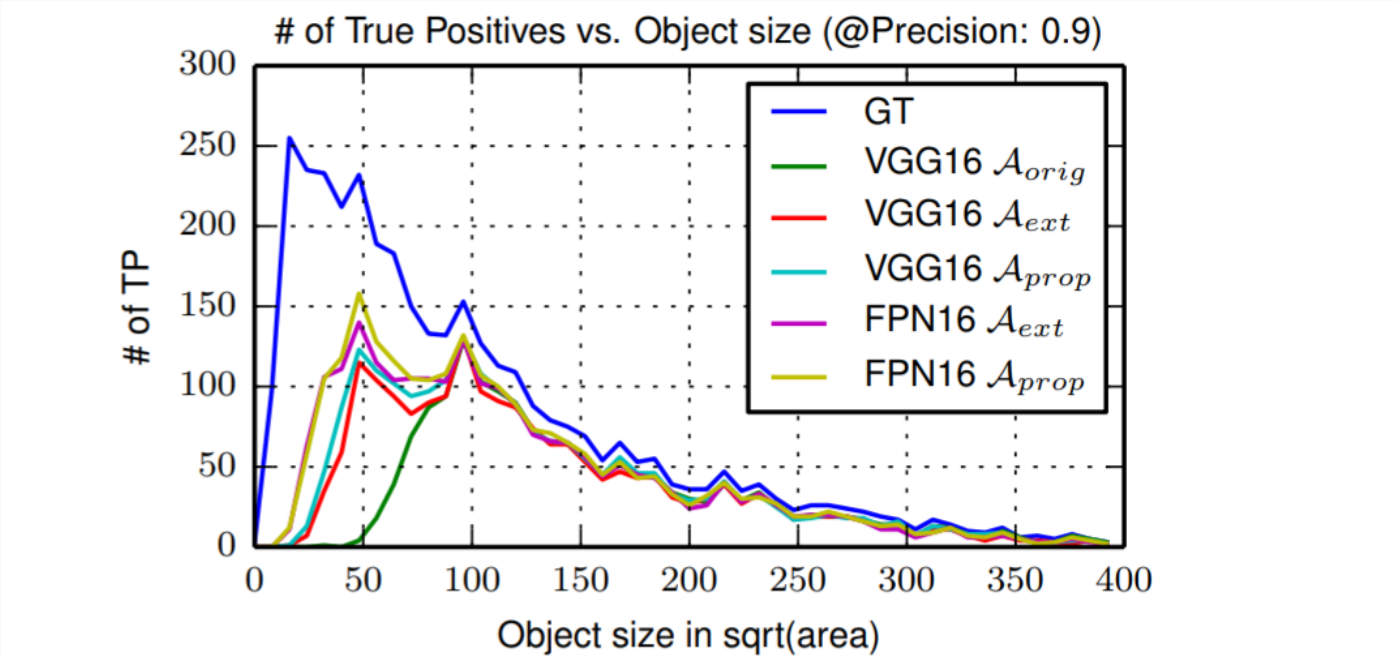


- И делают нечто похожее на FPN, чтобы объединить признаки:



Проблемы с масштабом

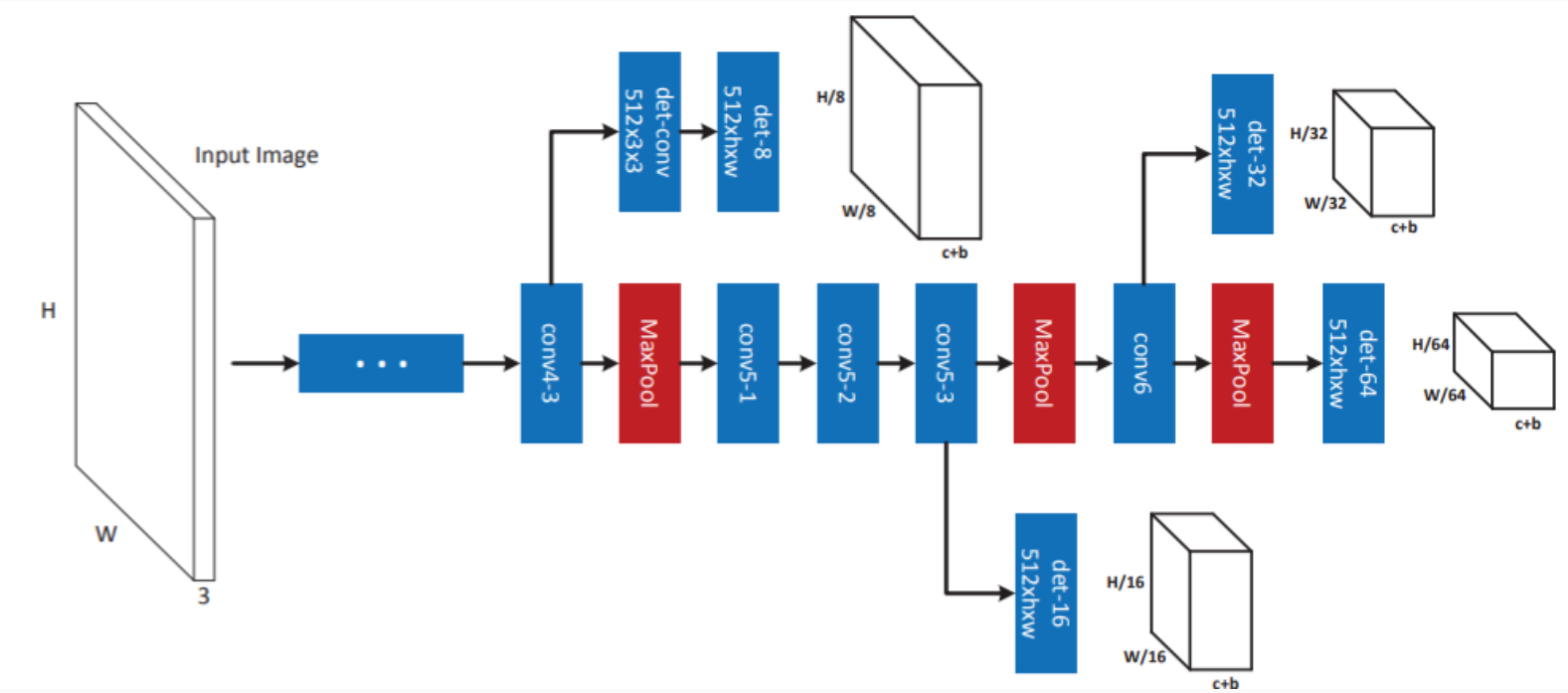
- Получается хорошо именно за счёт маленьких объектов:





Проблемы с масштабом

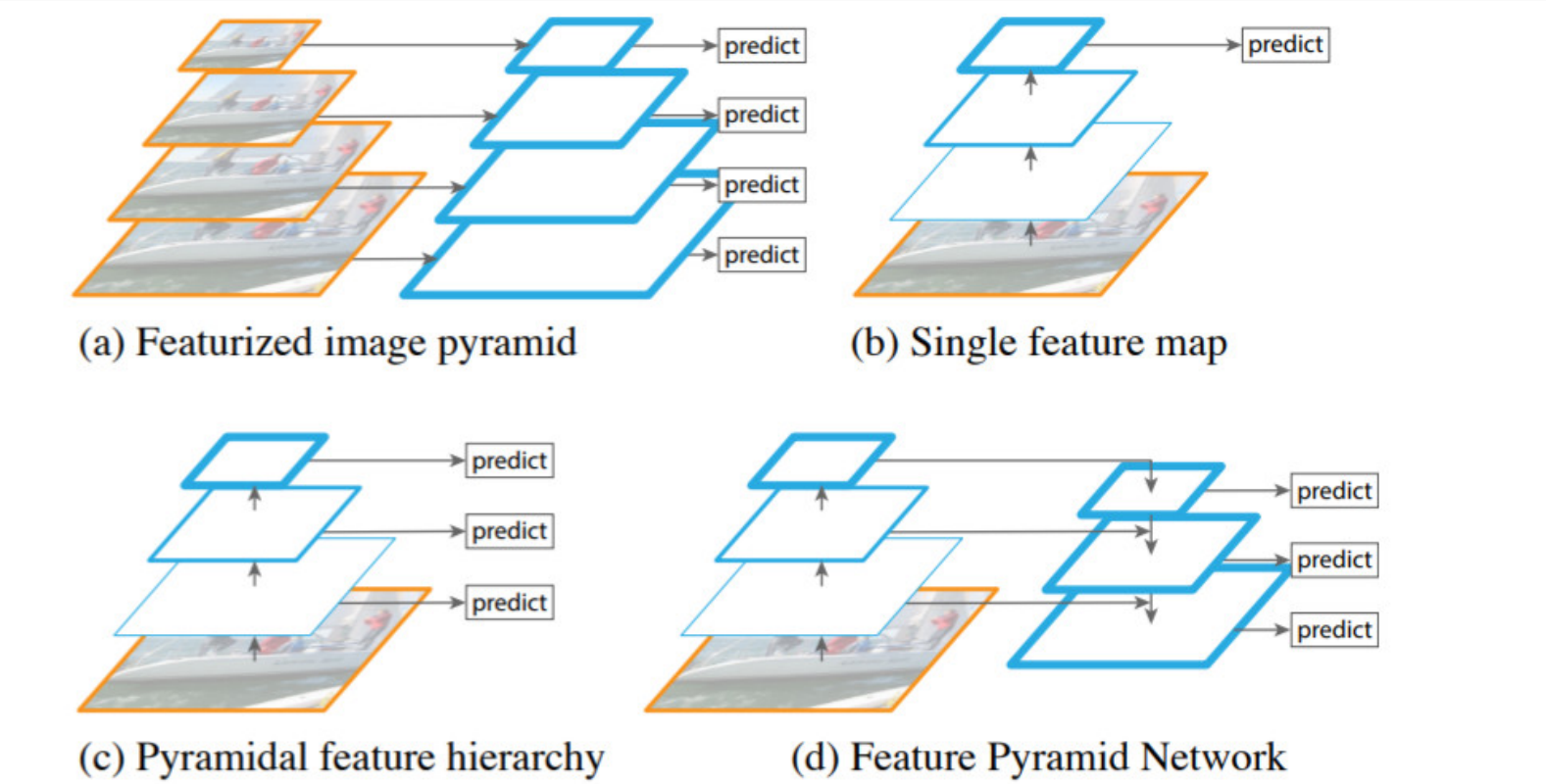
- (Cai et al., 2016): A unified multiscale deep convolutional neural network for fast object detection, аналогичная идея
- У object proposal network несколько веток с выходами разных масштабов:



Пирамиды признаков

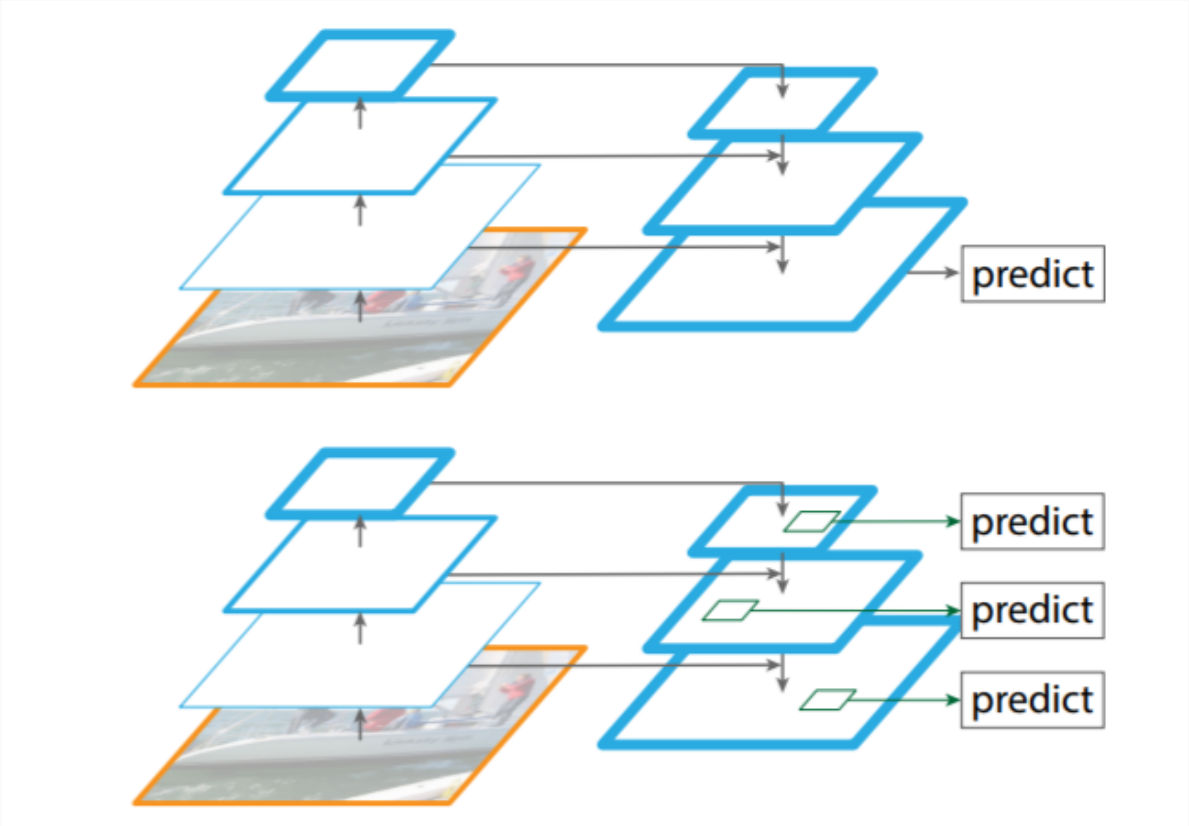
Feature Pyramid Networks

- (Lin et al., 2017): Feature Pyramid Networks for Object Detection
- Базовая идея – строим пирамиду признаков и используем их все:



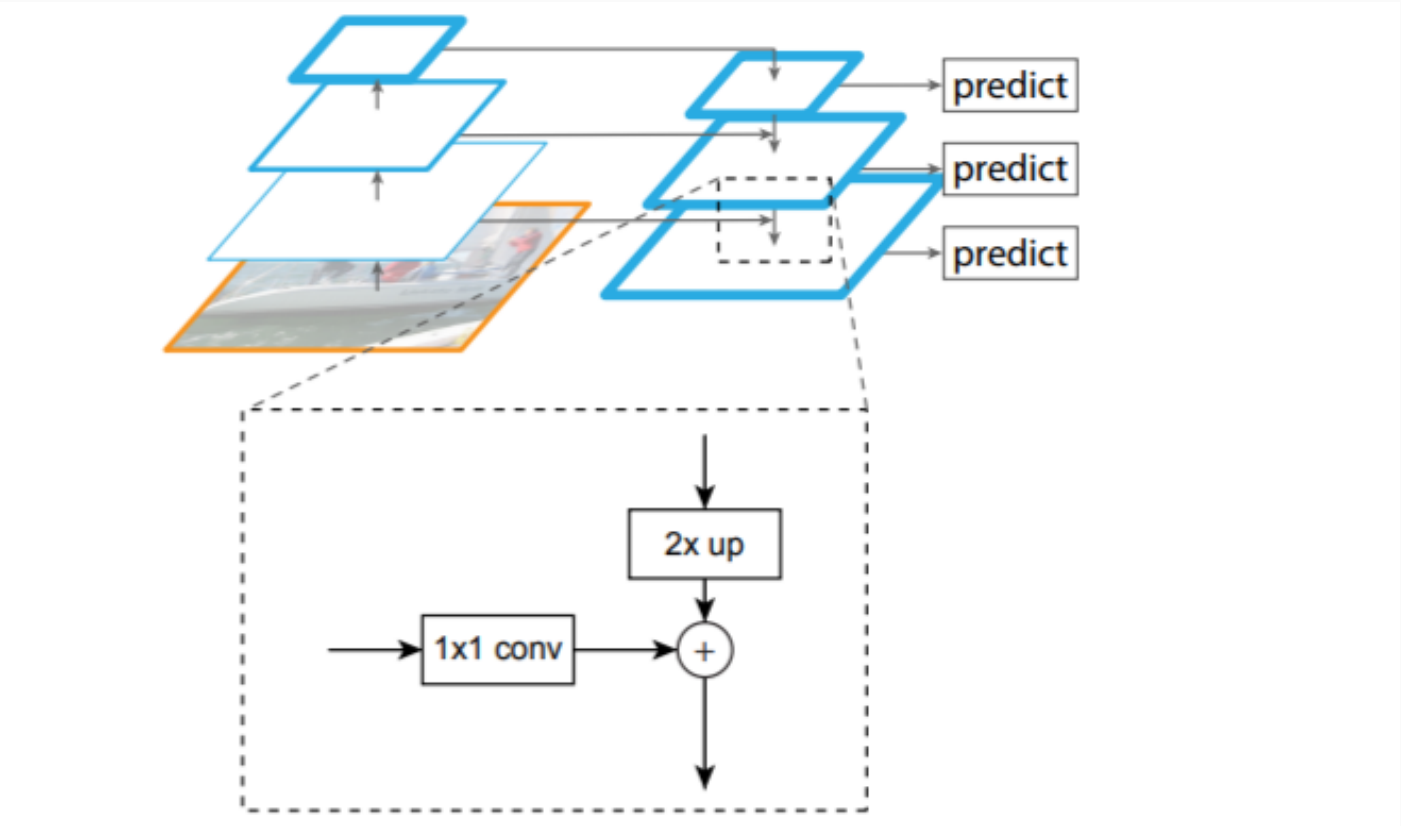
Feature Pyramid Networks

- Раньше так не делали, потому что очень долго и сложно
- FPN пытается сделать пирамиду признаков, у которой было бы много семантики на каждом уровне; сверху обычная архитектура, снизу FPN:



Feature Pyramid Networks

- Когда идём сверху вниз, добавляем признаки сверху, в них сила, то бишь семантика:



- Эту идею можно куда угодно применить, и в RPN, и в Faster R-CNN

Feature Pyramid Networks

- И, опять же, очень круто получается:

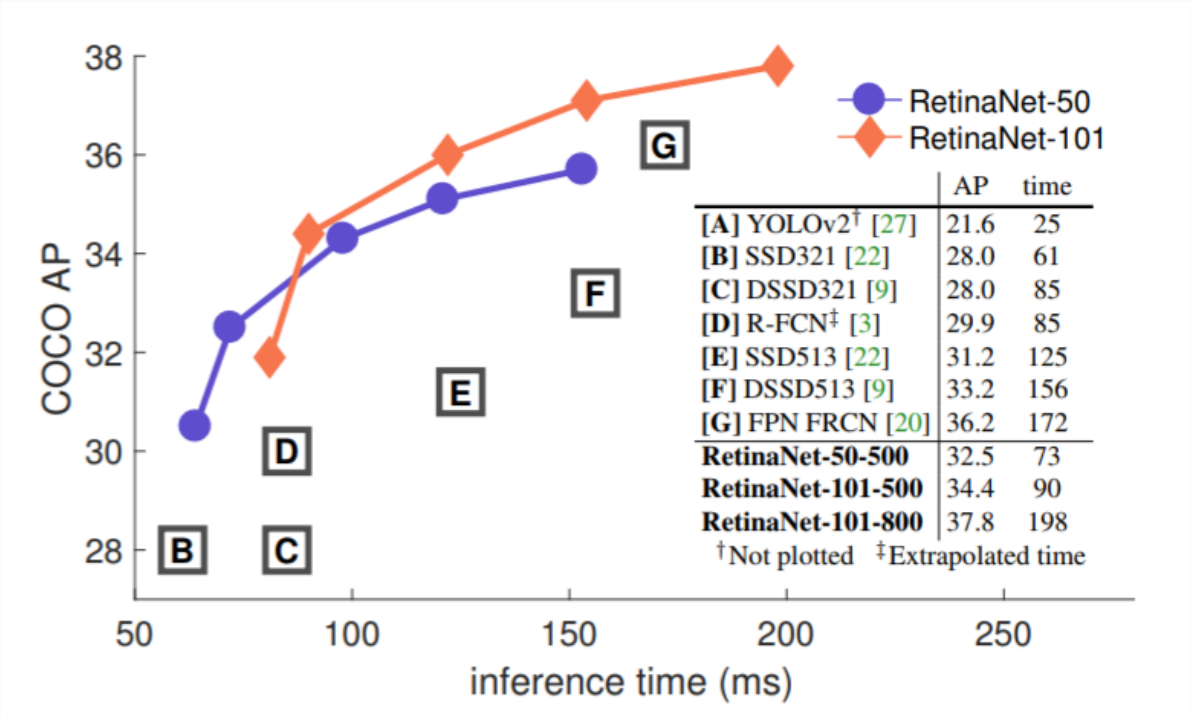
method	backbone	competition	image pyramid	test-dev AP@.5	test-dev AP	test-dev AP_s	test-dev AP_m	test-dev AP_l	test-std AP@.5	test-std AP	test-std AP_s	test-std AP_m	test-std AP_l
ours, Faster R-CNN on **FPN**	ResNet-101	-		**59.1**	**36.2**	**18.2**	**39.0**	48.2	**58.5**	**35.8**	**17.5**	**38.7**	47.8
*Competition-winning **single-model** results follow:*													
G-RMI†	Inception-ResNet	2016		-	34.7	-	-	-	-	-	-	-	-
AttractioNet‡ [10]	VGG16 + Wide ResNet§	2016	✓	53.4	35.7	15.6	38.0	**52.7**	52.9	35.3	14.7	37.6	**51.9**
Faster R-CNN +++ [16]	ResNet-101	2015	✓	55.7	34.9	15.6	38.7	50.9	-	-	-	-	-
Multipath [40] (on minival)	VGG-16	2015		49.6	31.5	-	-	-	-	-	-	-	-
ION‡ [2]	VGG-16	2015		53.4	31.2	12.8	32.9	45.2	52.9	30.7	11.8	32.8	44.8

- И можно убедиться, что действительно все эти пирамиды нужны, без них хуже:

Fast R-CNN	proposals	feature	head	lateral?	top-down?	AP@0.5	AP	AP_s	AP_m	AP_l
(a) baseline on conv4	RPN, $\{P_k\}$	C_4	conv5			54.7	31.9	15.7	36.5	45.5
(b) baseline on conv5	RPN, $\{P_k\}$	C_5	2*fc*			52.9	28.8	11.9	32.4	43.4
(c) **FPN**	RPN, $\{P_k\}$	$\{P_k\}$	2*fc*	✓	✓	**56.9**	**33.9**	**17.8**	**37.7**	**45.8**
Ablation experiments follow:										
(d) bottom-up pyramid	RPN, $\{P_k\}$	$\{P_k\}$	2*fc*	✓		44.9	24.9	10.9	24.4	38.5
(e) top-down pyramid, w/o lateral	RPN, $\{P_k\}$	$\{P_k\}$	2*fc*		✓	54.0	31.3	13.3	35.2	45.3
(f) only finest level	RPN, $\{P_k\}$	P_2	2*fc*	✓	✓	56.3	33.4	17.3	37.3	45.6

Detectron

- (Lin et al., 2018): Focal Loss for Dense Object Detection
- Изменили функцию ошибки, стали работать гораздо быстрее и лучше:



	AP	time
[A] YOLOv2† [27]	21.6	25
[B] SSD321 [22]	28.0	61
[C] DSSD321 [9]	28.0	85
[D] R-FCN‡ [3]	29.9	85
[E] SSD513 [22]	31.2	125
[F] DSSD513 [9]	33.2	156
[G] FPN FRCN [20]	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	37.8	198

†Not plotted ‡Extrapolated time

- Как это?

Detectron

- Что мешает one-shot detectors? Почему они хуже, чем two-stage (типа Faster R-CNN)?
- Lin et al. утверждают, что мешает class imbalance: надо очень много выходов сразу предсказать, и в основном это очень простые примеры из чистого background
- В two-stage детекторах есть фильтрация, region proposal networks, т.е. у них imbalance совсем не такой сильный

- Чтобы с этим справиться, предлагается focal loss. Начинаем с перекрёстной энтропии:

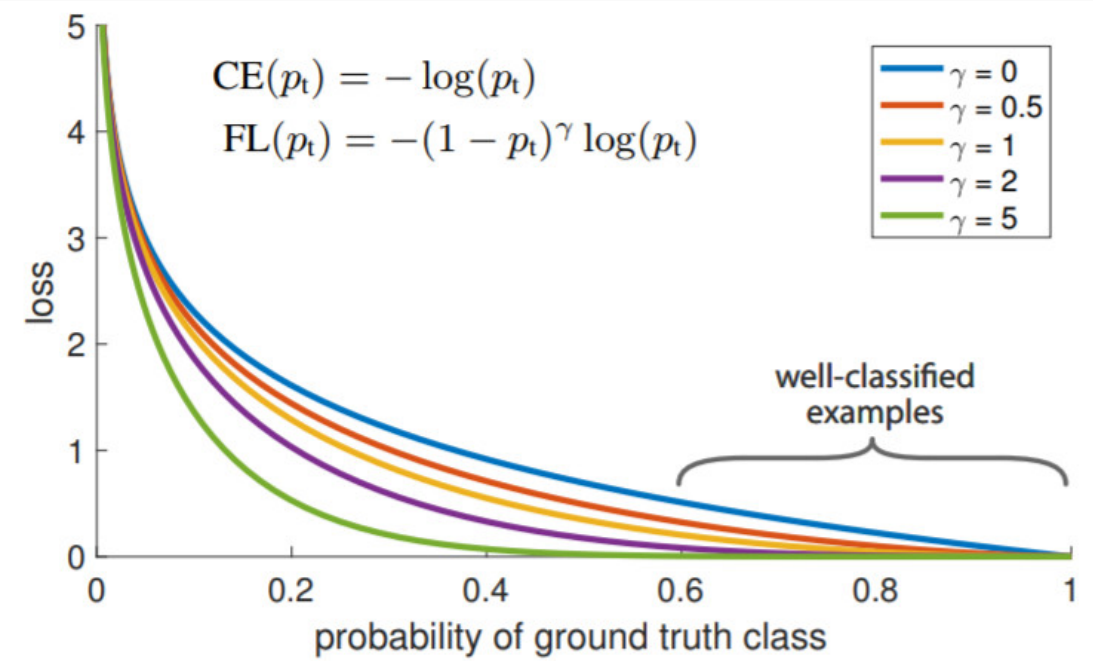
$$\mathrm{CE}(p, y) = \begin{cases} -\log(p), & \text{если } y = 1, \\ -\log(1-p), & \text{в противном случае;} \end{cases}$$

  переобозначим $p_t = [y = 1]p + [y = 0](1 - p)$, и тогда $\mathrm{CE}(p, y) = -\log(p_t)$.
- Проблема тут в том, что даже простые примеры с $p_t > 0.8$ дают нетривиальную потерю, и если всё просуммировать, то забивают «сигнал».

- Focal loss:

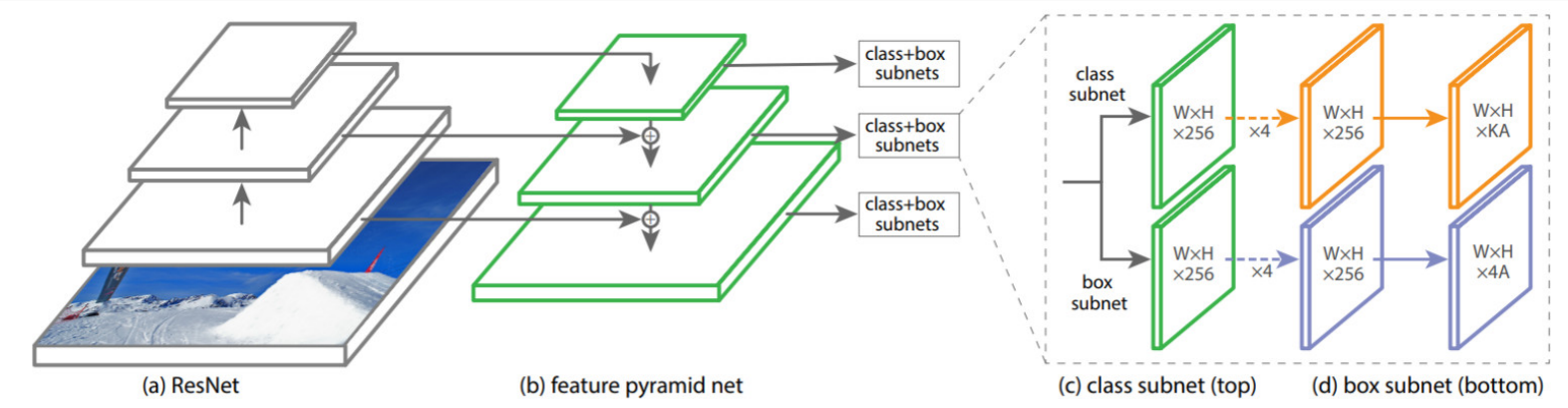
$$\mathrm{FL}(p, y) = -(1 - p_t)^{\gamma} \log(p_t).$$

- Просто перевзвешиваем так, чтобы потерь было меньше на хорошо классифицируемых примерах.



Detectron

- RetinaNet: FPN c focal loss



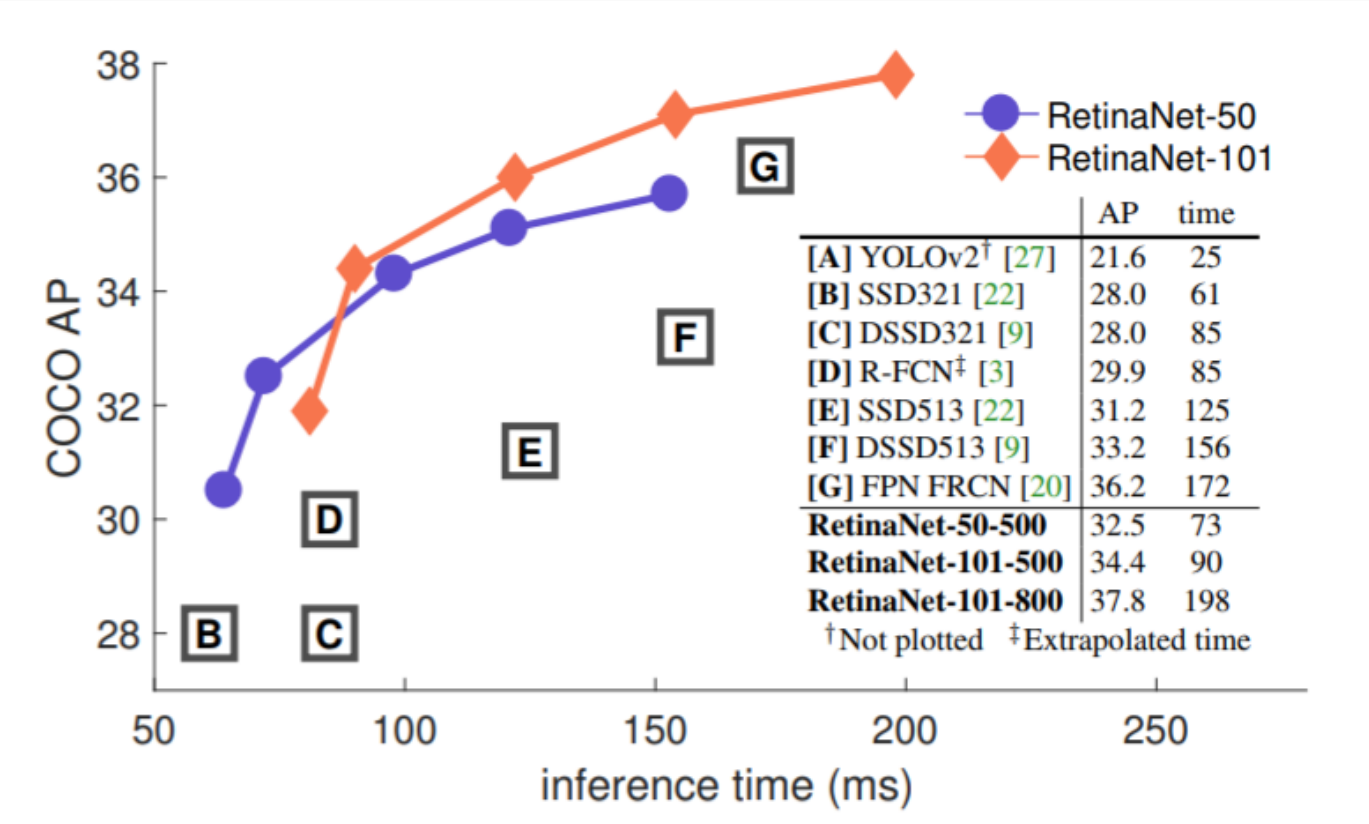
Detectron

- Получается очень хорошо:

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [16]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [20]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [17]	Inception-ResNet-v2 [34]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [32]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [27]	DarkNet-19 [27]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [22, 9]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [9]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet (ours)	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet (ours)	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2

Detectron

- Очень хорошо...

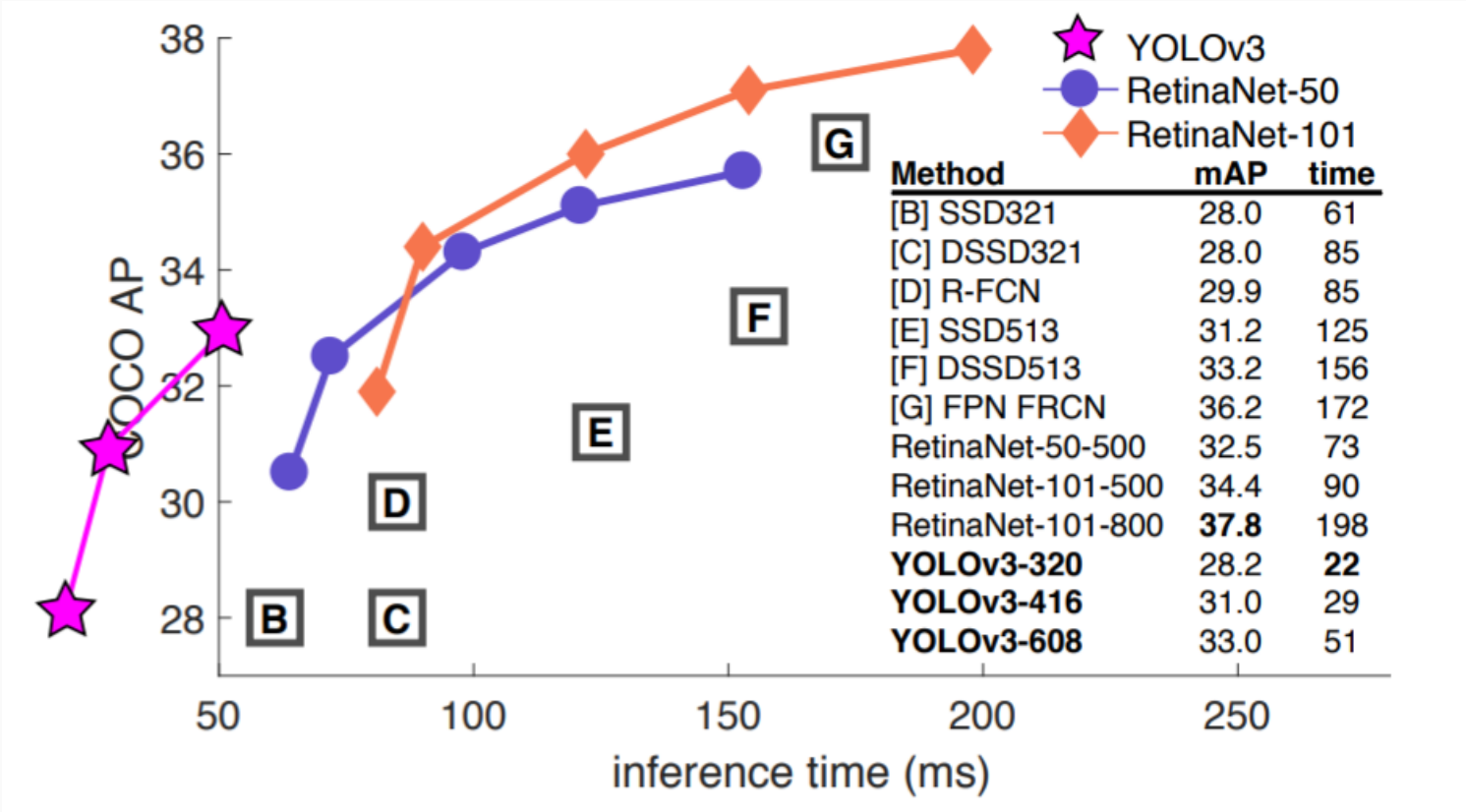


	AP	time
[A] YOLOv2† [27]	21.6	25
[B] SSD321 [22]	28.0	61
[C] DSSD321 [9]	28.0	85
[D] R-FCN‡ [3]	29.9	85
[E] SSD513 [22]	31.2	125
[F] DSSD513 [9]	33.2	156
[G] FPN FRCN [20]	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	37.8	198

†Not plotted ‡Extrapolated time

YOLOv3

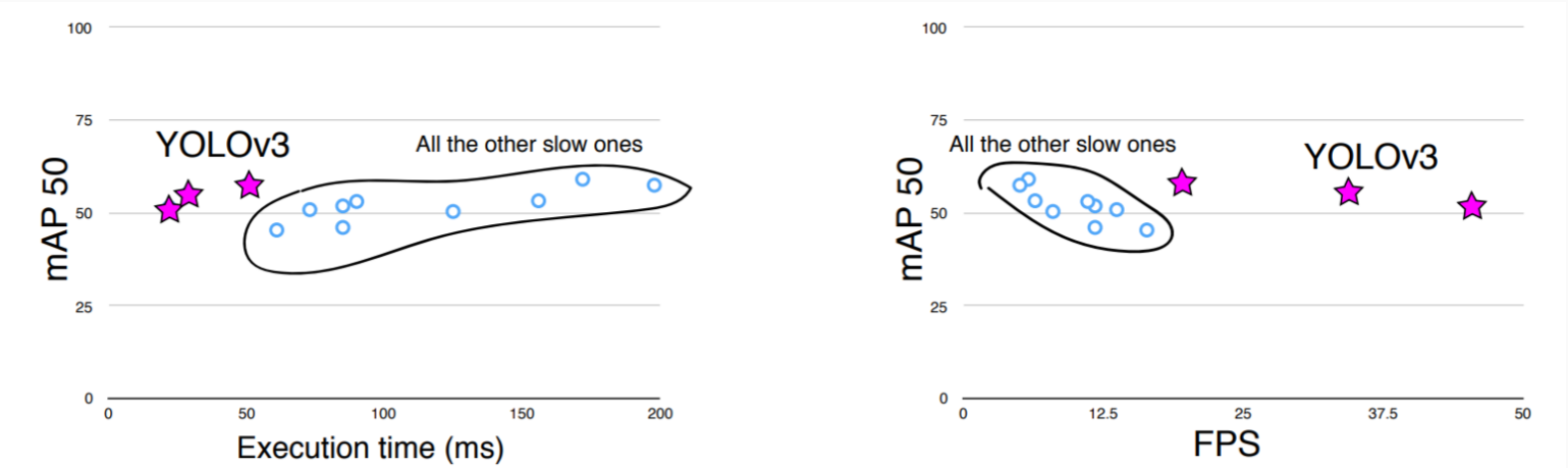
- …но YOLO всё равно лучше



Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

YOLOv3

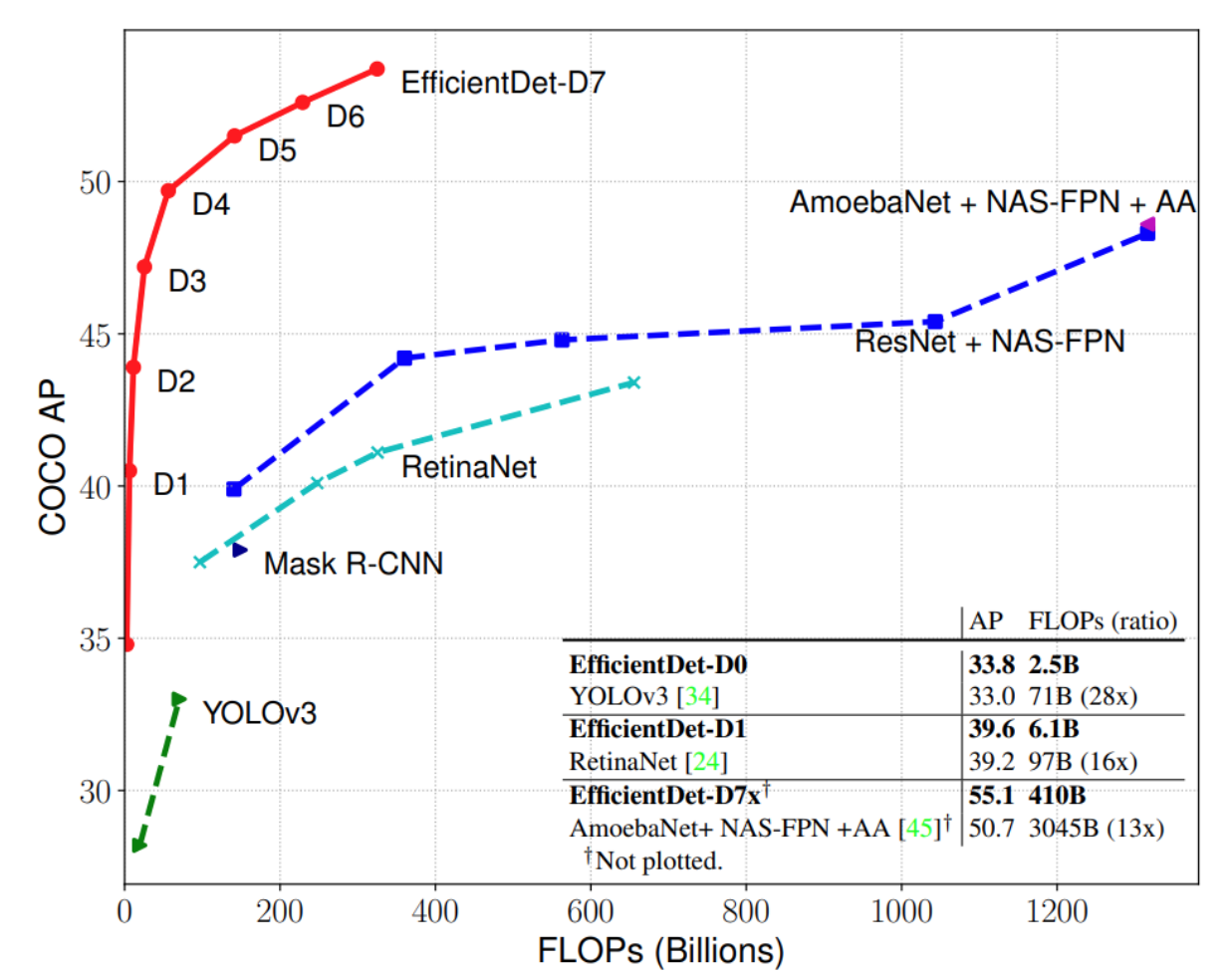
- YOLOv3 – это новый вариант YOLO, в котором использовали focal loss.
- Быстро:



- И хорошо: `https://www.youtube.com/watch?time_continue=2&v=MPU2HistivI`

EfficientDet

- (Tan et al., 2019): EfficientDet – neural architecture search добрался и до object detection, конечно

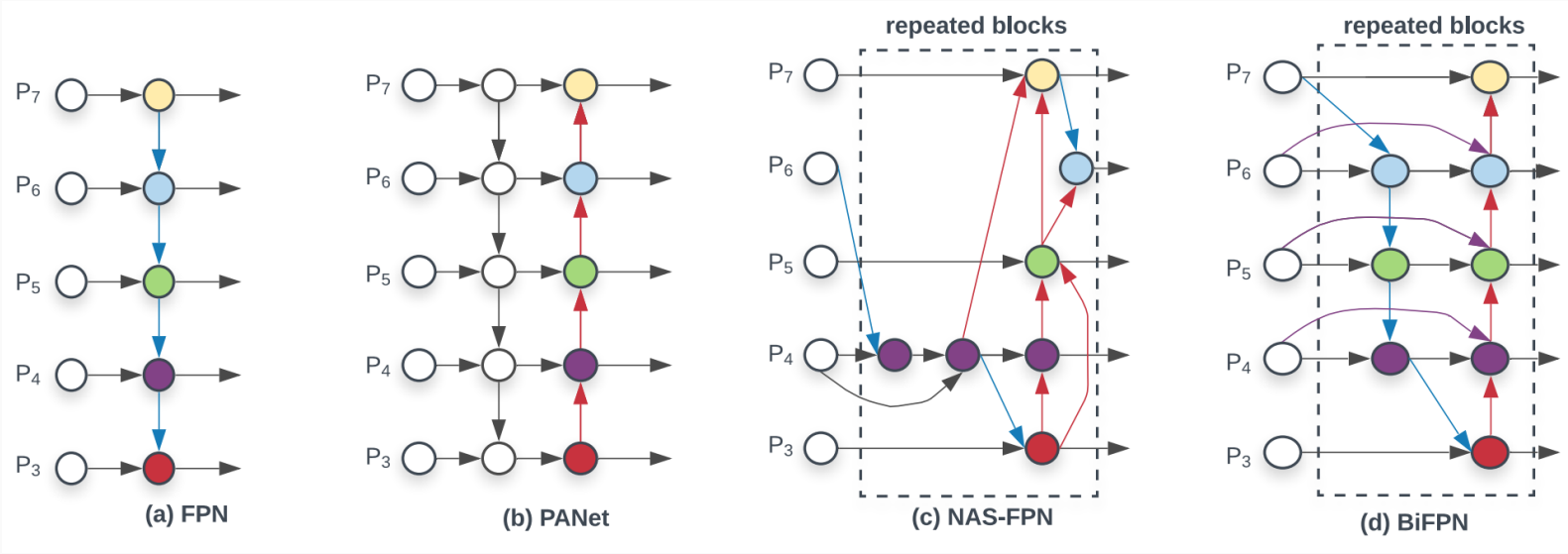


	AP	FLOPs (ratio)
EfficientDet-D0	**33.8**	**2.5B**
YOLOv3 [34]	33.0	71B (28x)
EfficientDet-D1	**39.6**	**6.1B**
RetinaNet [24]	39.2	97B (16x)
EfficientDet-D7x†	**55.1**	**410B**
AmoebaNet+ NAS-FPN +AA [45]†	50.7	3045B (13x)

†Not plotted.

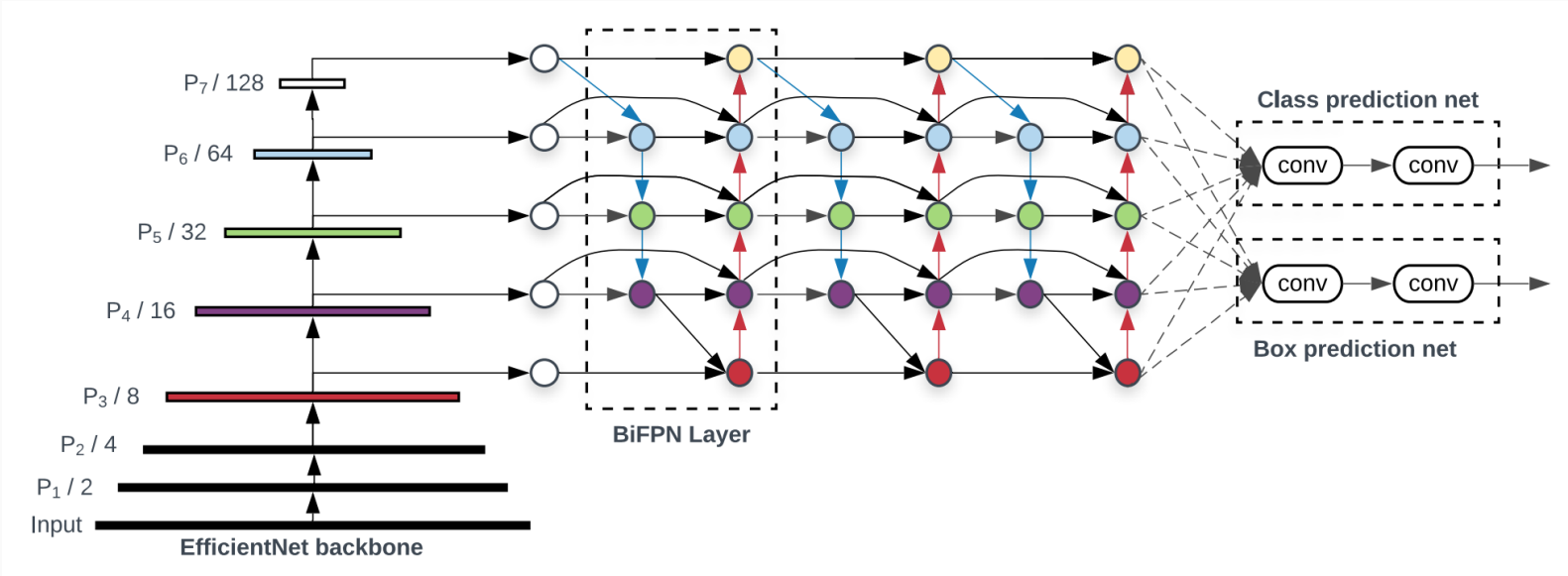
EfficientDet

- Новые типы блоков для пирамид признаков:



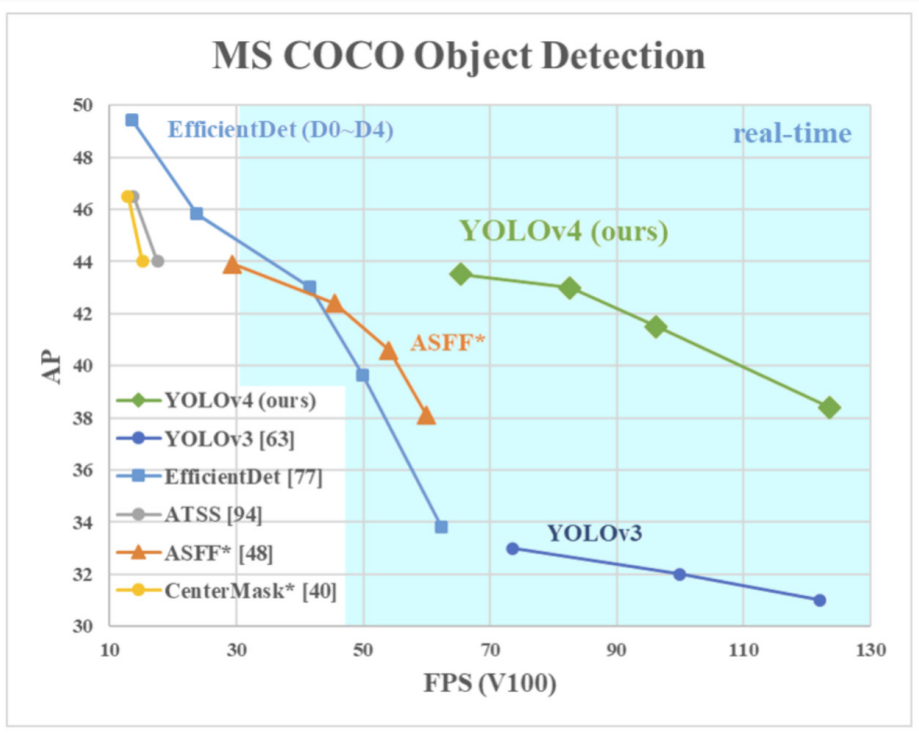
EfficientDet

- А дальше просто подставим EfficientNet в качестве backbone:



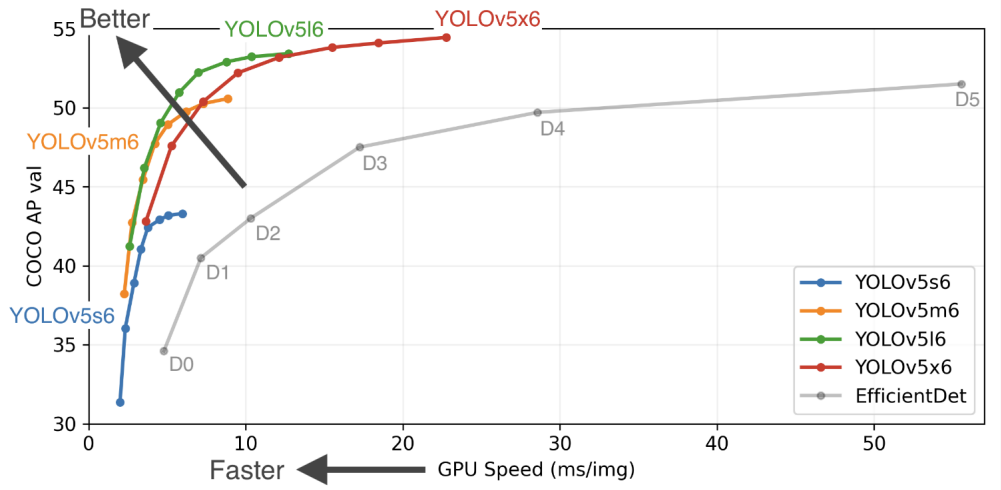
YOLOv4, YOLOv5, PP-YOLO...

- YOLOv4 (Bochkovskiy et al., 2020): again a new iteration with lots of small improvements
- Bag-of-freebies (augmentation incl. self-adversarial training, label smoothing, objective function etc.) vs. bag-of-specials (activation functions, attention mechanism, weighted skip connections etc.)

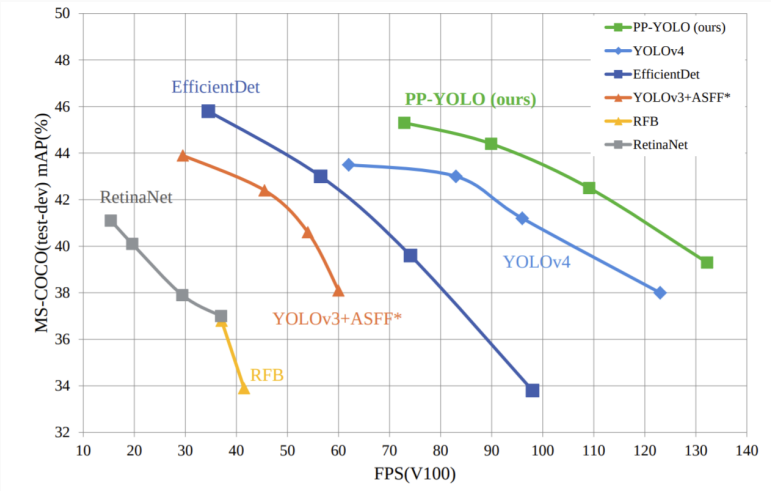


YOLOv4, YOLOv5, PP-YOLO...

- YOLOv5 only took a couple of months, but there is a story here...



- Things get even murkier with PP-YOLO (Long et al., 2020)



YOLO сегодня

А что с YOLO сейчас?

- Мы остановились на YOLOv4/v5; с тех пор прошло шесть лет, и «YOLO» из одной архитектуры превратилось в бренд.
- Джозеф Редмон, автор v1–v3, ушёл из computer vision в 2020 году по этическим соображениям и больше к YOLO отношения не имеет.
- Имя подхватили все кому не лень:
  - v4 и v7 — Бочковский, Ванг и Ляо;
  - v5, v8, YOLO11 — компания Ultralytics (статей нет, есть репозиторий);
  - v6 — Meituan, v9 — снова Ванг и Ляо, v10 — Университет Цинхуа.

А что с YOLO сейчас?

- Две идеи из этого потока действительно важны.
- *NMS-free детекция* (YOLOv10, 2024): две головы при обучении, one-to-many для сигнала и one-to-one для инференса — и non-maximum suppression больше не нужен, а значит, детектор становится честным end-to-end.
- *Внимание просочилось внутрь*: YOLOv12 (Tian et al., 2025) называет себя attention-centric — area attention A^2 режет карту признаков на полосы, чтобы внимание стоило линейно, плюс R-ELAN для стабильного обучения; 40.6% mAP при 1.64 мс.
- YOLOv13 (2025) пошёл дальше и собирает контекст гиперграфом (HyperACE).
- YOLOv12: `https://arxiv.org/abs/2502.12524`
- YOLOv13: `https://arxiv.org/abs/2506.17733`

YOLO26 и YOLO27

- YOLO26 (Ultralytics, январь 2026) — то, что сейчас стоит брать по умолчанию:
  - end-to-end без NMS прямо из коробки;
  - выкинули distribution focal loss — регрессия рамки стала проще и дружелюбнее к железу;
  - обучение на MuSGD (SGD, скрещённый с Muon из мира LLM), ProgLoss и STAL — последнее специально про мелкие объекты.
- 40.9–57.5% mAP на COCO при 1.7–11.8 мс на T4; на CPU до 43% быстрее YOLO11.
- `https://docs.ultralytics.com/models/yolo26`

YOLO26 и YOLO27

- Одна модель теперь закрывает семь задач: детекция, instance- и semantic-сегментация, классификация, позы, повёрнутые рамки (OBB) и даже монокулярная глубина.
- А 13 сентября 2026 года на YOLO Vision анонсировали YOLO27: четыре размера, те же семь задач, NMS-free; YOLO27l заявлен как первая модель Ultralytics выше 60 mAP на COCO (61.2). Скачать пока нельзя — обещают «до конца года».
- `https://docs.ultralytics.com/models/yolo27`

Можно ли ещё без трансформеров?

- Короткий ответ: да, и это по-прежнему выбор по умолчанию, если вам нужен real-time на видеопотоке или деплой на edge.
- Свёрточные детекторы выигрывают по задержке, по размеру и по тому, насколько предсказуемо они квантуются в INT8 и экспортируются в ONNX/TensorRT.
- Даже открытый словарь теперь доступен без трансформерного детектора: YOLOE (Wang et al., ICCV 2025) добавляет к YOLO три режима — текстовый промпт (RepRTA), визуальный промпт-пример (SAVPE) и вообще без промпта (LRPC со встроенным словарём на 4585 имён).
- YOLOE: `https://arxiv.org/abs/2503.07465`

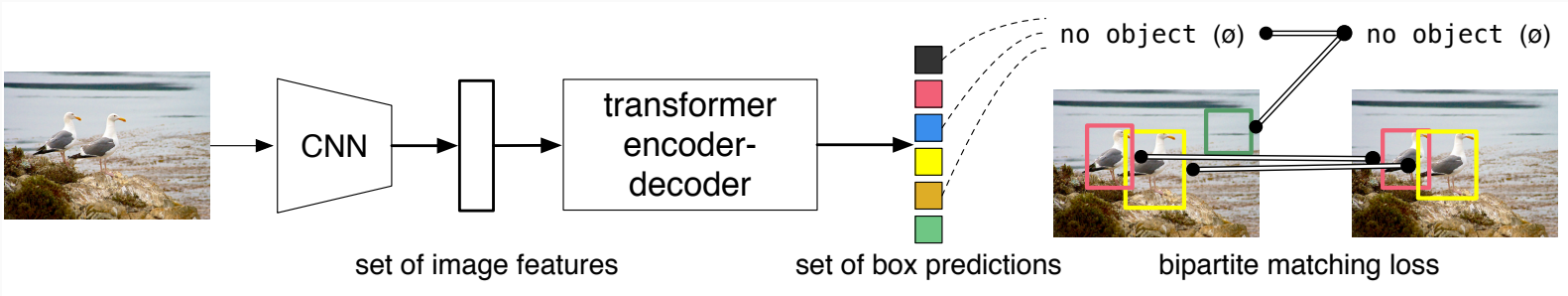
Можно ли ещё без трансформеров?

- Но честно скажем и обратное: на верхней полке качества трансформеры.
- RF-DETR первым перешагнул 60 AP на COCO, оставаясь real-time; это DINOv2-бэкбон плюс деформируемое внимание в декодере.
- SAM 3 (Meta, ноябрь 2025) сегментирует по текстовому понятию сразу все экземпляры на картинке и в видео, а SAM 3.1 (март 2026) добавил трекинг нескольких объектов с общей памятью.
- Так что граница проходит не по «свёртки против внимания», а по тому, что вам важнее: миллисекунды и джоули или открытый словарь и качество масок.
- `https://ai.meta.com/blog/segment-anything-model-3/`

Распознавание объектов с трансформерами

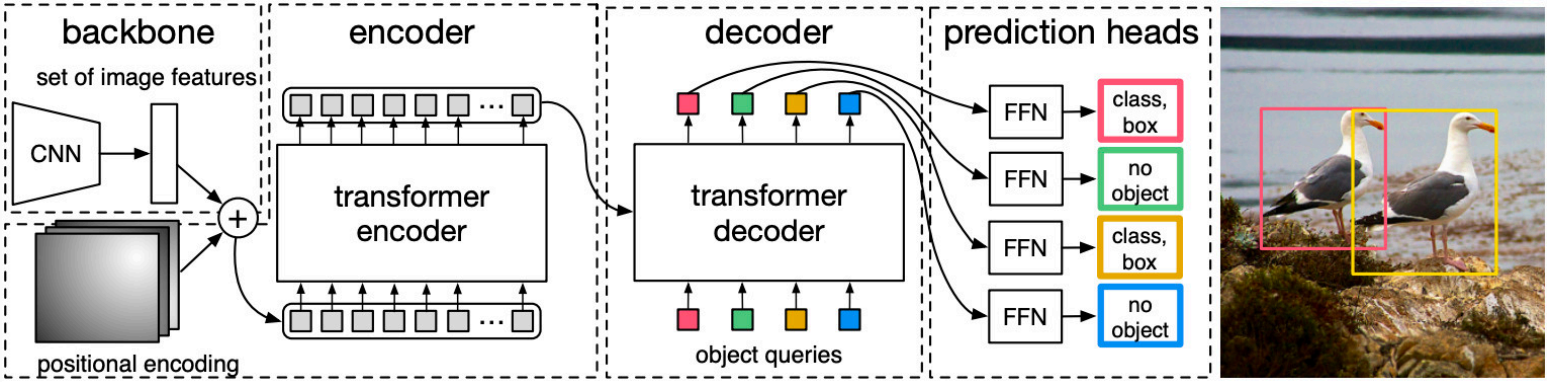
DETR и аналоги

- DETR (DEtection TRansformer; Carion et al., 2020) – прямая адаптация трансформеров
- Распознавание объектов — задача предсказания множества; DETR добавляет основанную на множествах глобальную функцию потерь, которая даёт уникальные предсказания через паросочетания, и архитектуру кодировщик-декодировщик на базе трансформера



DETR и аналоги

- DETR использует традиционный сверточный backbone (ResNet у Carion et al., но можно взять что угодно)
- Затем DETR выравнивает признаки, добавляет позиционное кодирование и перемешивает признаки с помощью кодировщика
- Декодировщик берёт на вход небольшое число *объектных запросов*, обучаемых позиционных вложений, с вниманием на выход кодировщика
- Выход проходит через маленькую feedforward сеть, которая предсказывает класс и bbox (или “нет объекта”)



DETR и аналоги

- Функция потерь происходит из паросочетания:
  - для множества предсказаний $\hat{y} = \{\hat{y}_i\}_{i=1}^N$ и истинных объектов $\mathbf{y} = \{y_i\}_{i=1}^N$ (плюс $\emptyset$, т.к. N должно быть большим), ищем перестановку $\sigma_* = \arg\min_\sigma \sum_{i=1}^N \ell_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ с помощью венгерского алгоритма, где ℓ_{match} – это стоимость сопоставления;
  - ℓ_{match} объединяет точность предсказаний для $y_i = (c_i, b_i)$, где c_i – метка класса (возможно, $\emptyset$), а $b_i \in \mathbb{R}^4$ – bbox:

$$\ell_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) = -[c_i \neq \emptyset]\hat{p}_{\sigma(i)}(c_i) + [c_i \neq \emptyset]\ell_{\text{box}}(b_i, \hat{b}_{\sigma(i)});$$

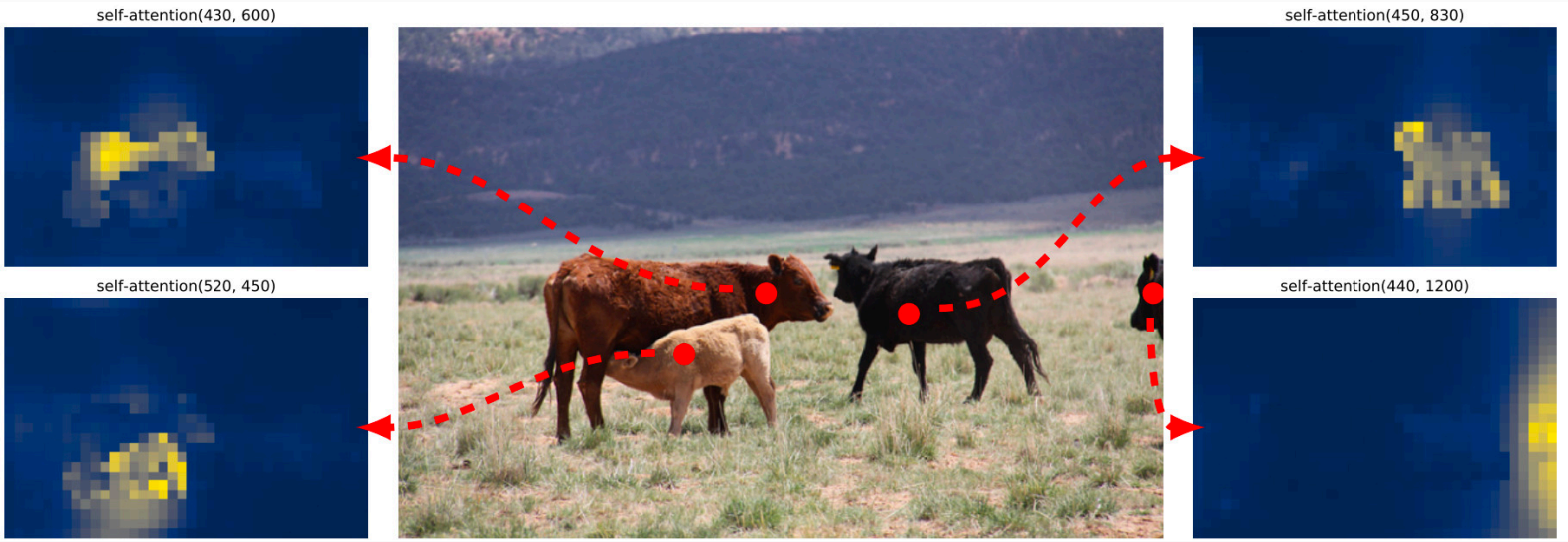
    здесь ℓ_{box} – комбинация функций потерь от IoU и L_1;
  - после сопоставления фактическая функция потерь

$$\ell_{\text{Hungarian}}(\mathbf{y}, \hat{y}) = \sum_{i=1}^N \left(-\log \hat{p}_{\sigma_*(i)}(c_i) + [c_i \neq \emptyset]\ell_{\text{box}}(b_i, \hat{b}_{\sigma_*(i)})\right);$$

    теперь мы предсказываем и $c_i = \emptyset$, но на практике они получают меньший вес.

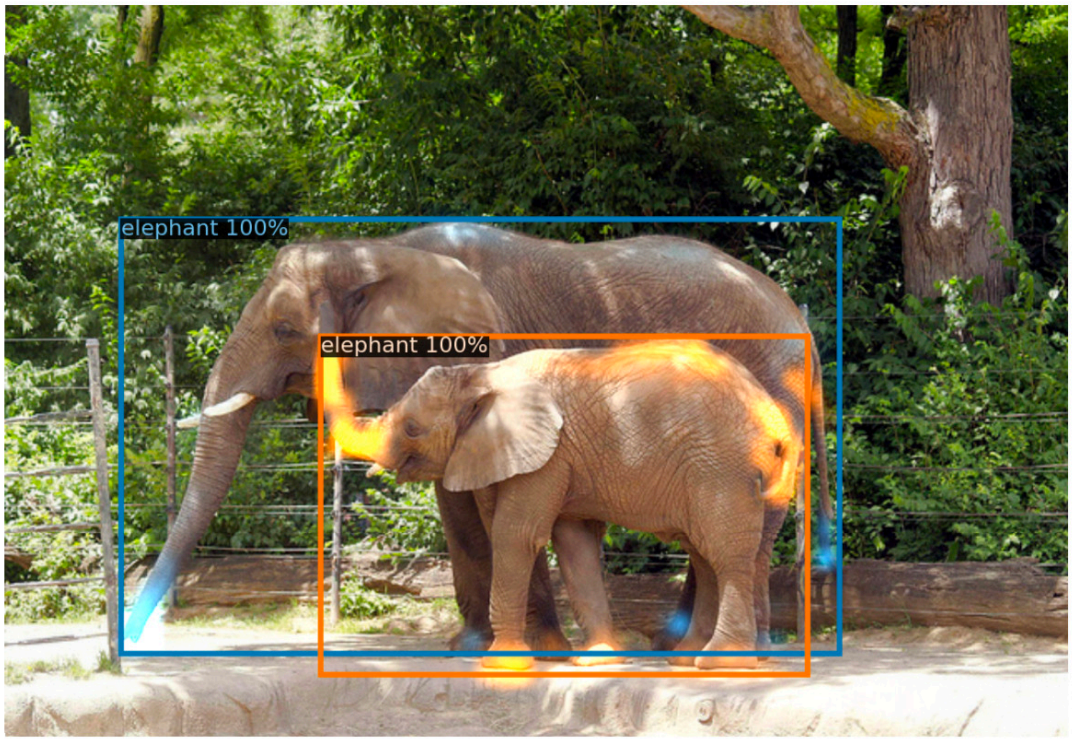
DETR и аналоги

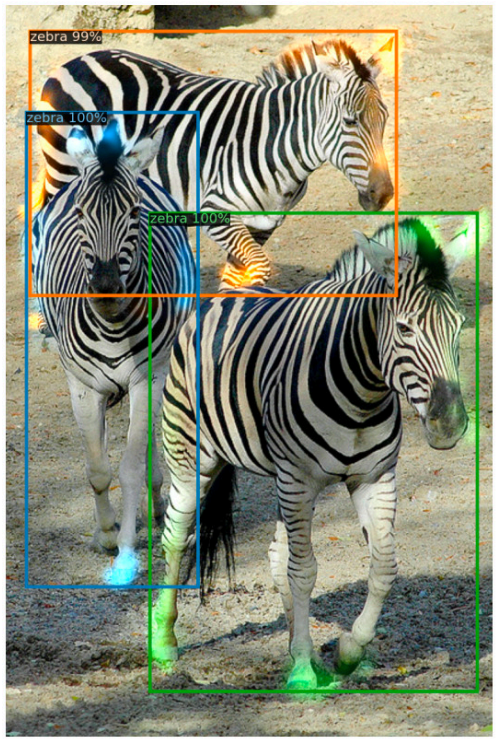
- Хорошие результаты, и DETR способен распределять экземпляры объектов по головам внимания:



DETR и аналоги

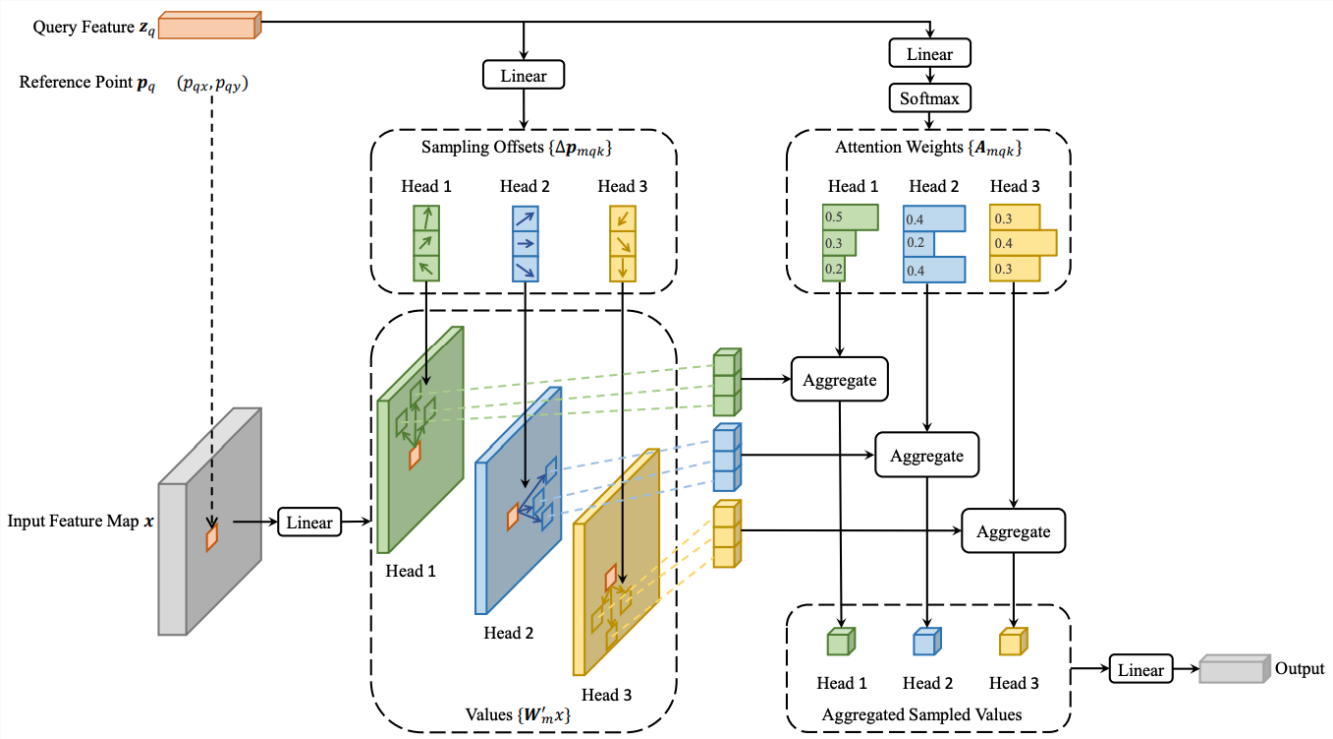
- Внимание декодера для каждого объекта локализовано на границах
- Скорее всего, DETR разделяет экземпляры глобальным вниманием в кодировщике, а декодеру остается только выяснить классы и границы объектов





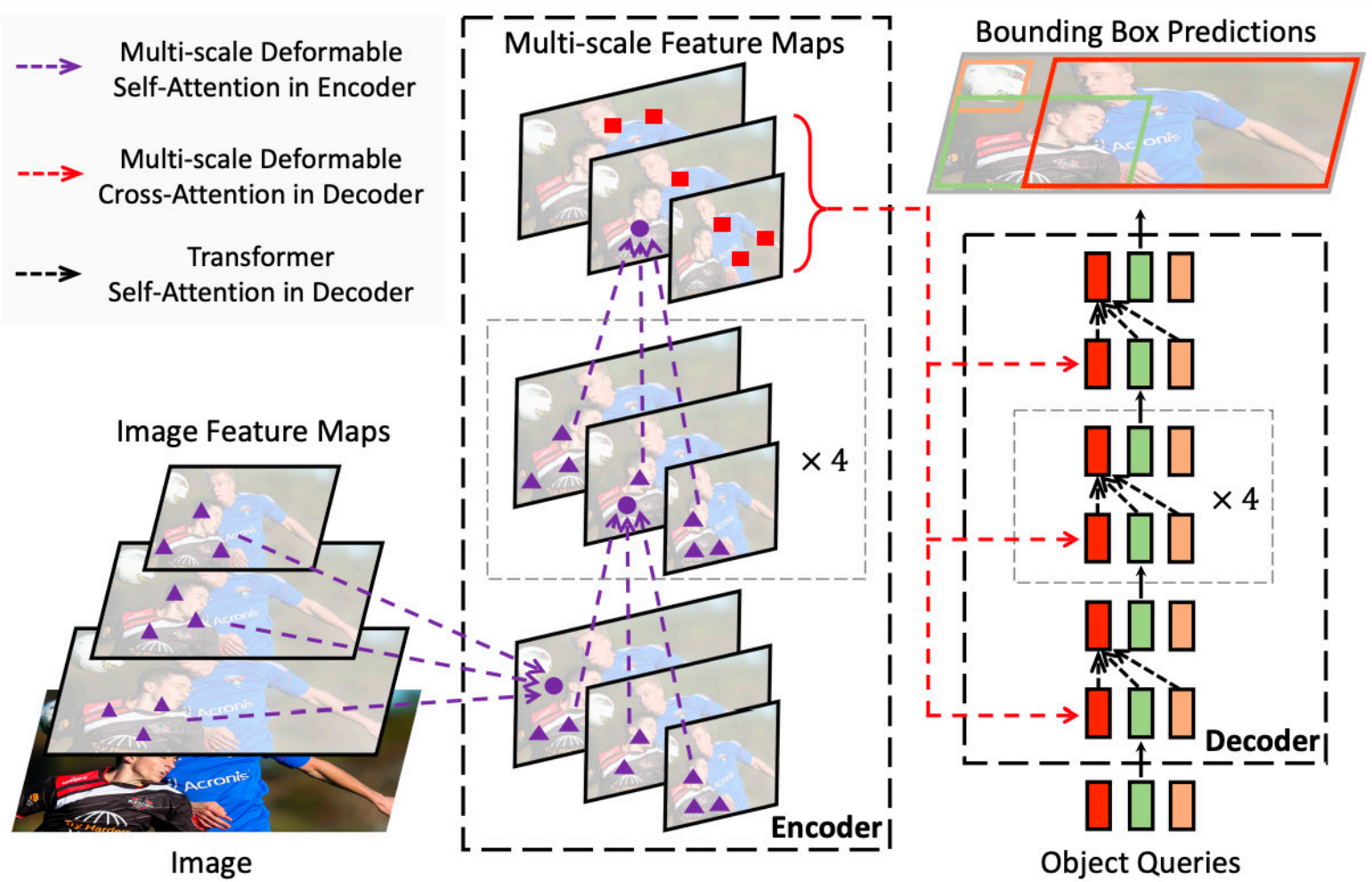
DETR и аналоги

- DETR работает хорошо, но не очень хорошо на маленьких объектах, а квадратичная сложность делает мультимасштабное распознавание очень сложной задачей
- Deformable DETR (Zhu et al., 2021): давайте используем деформируемое внимание и будем обращать внимание только на несколько пикселей



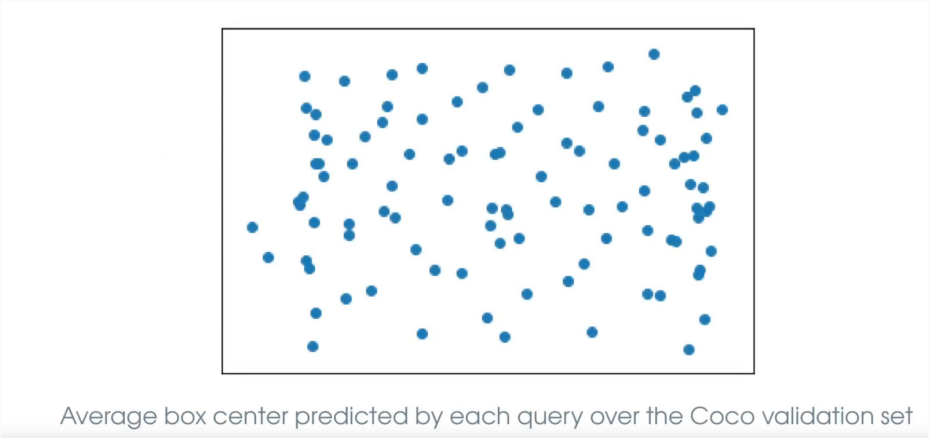
DETR и аналоги

- Общая архитектура:



DETR и аналоги

- Позиционные запросы DETR – это в основном гибкие обучаемые якоря для ограничивающих прямоугольников; они обучаются глобально и не зависят от текущего изображения

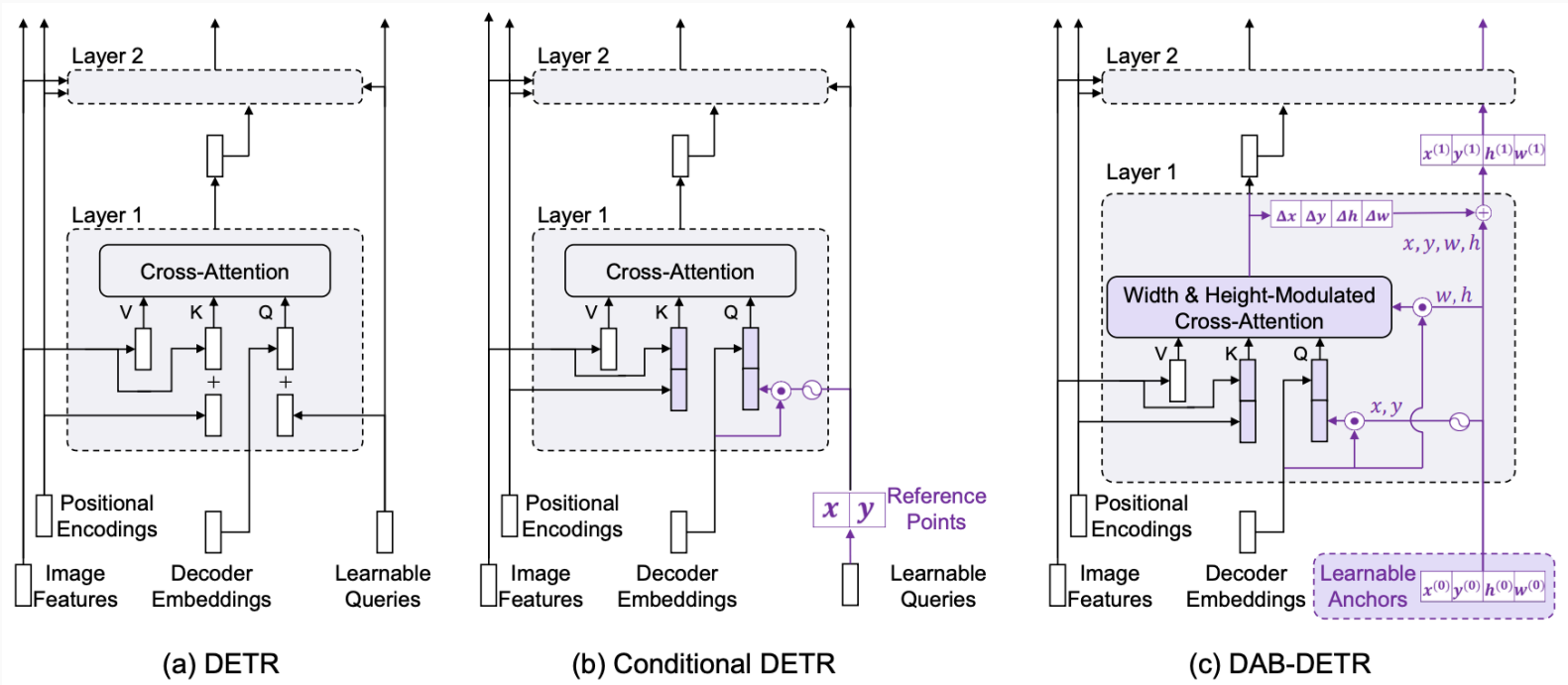


Average box center predicted by each query over the Coco validation set

- Обучение пространственных запросов на данных (без каких-либо фиксированных anchor boxes) – важная часть DETR, но DETR обучается очень, очень медленно, и пространственные запросы – большой фактор в этом
- Как мы можем это улучшить?

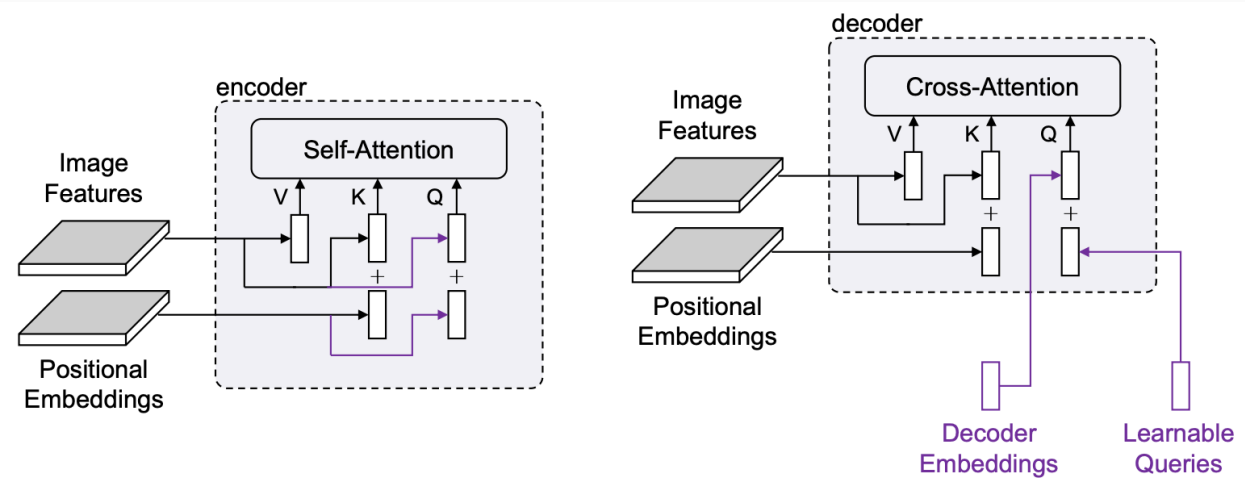
DETR и аналоги

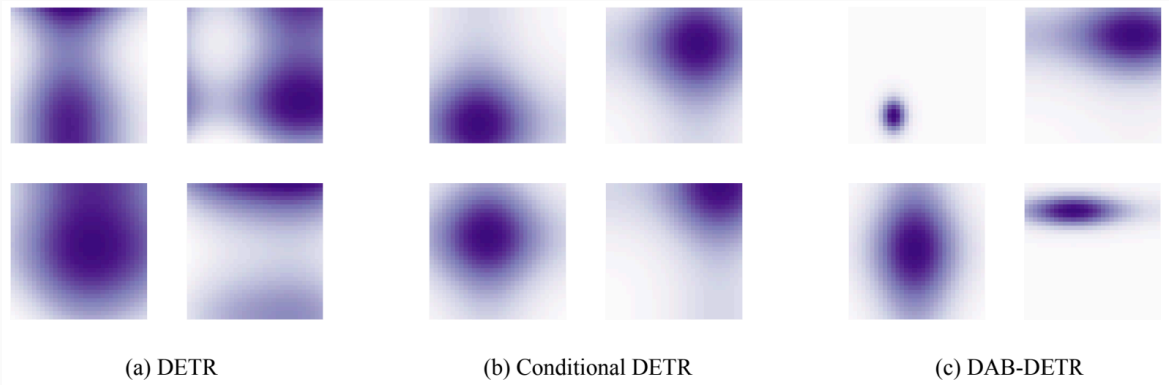
- DAB-DETR (Dynamic Anchor Box DETR; Liu et al., 2022): давайте вернемся к идее anchor boxes
- Обучаемые якоря постепенно уточняются от слоя к слою в декодере, и теперь они имеют определенную семантику:



DETR и аналоги

- Ширина и высота anchor box используются для модуляции перекрестного внимания, поэтому внимание становится гораздо лучше локализованным, и это помогает ускорить обучение

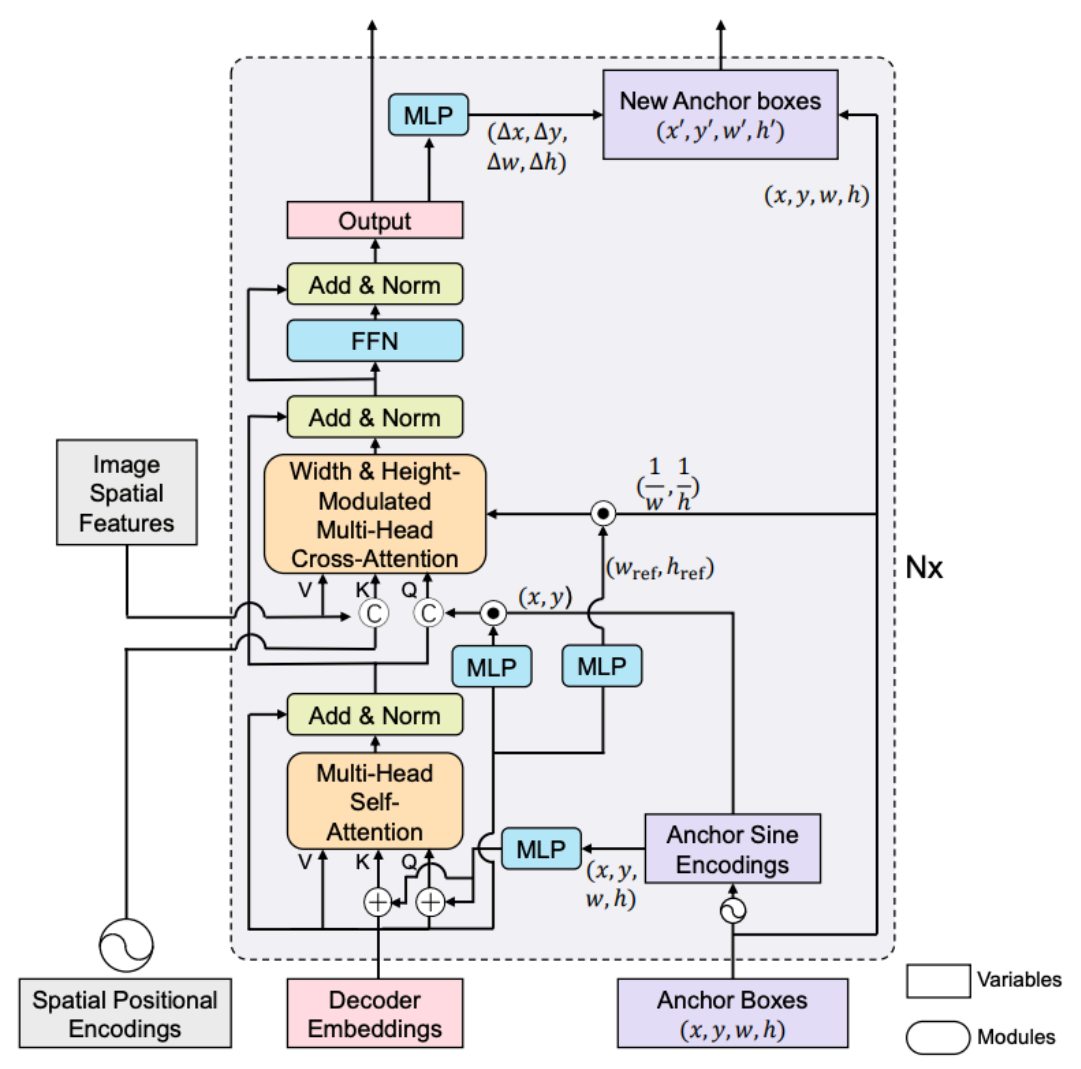




(a) DETR (b) Conditional DETR (c) DAB-DETR

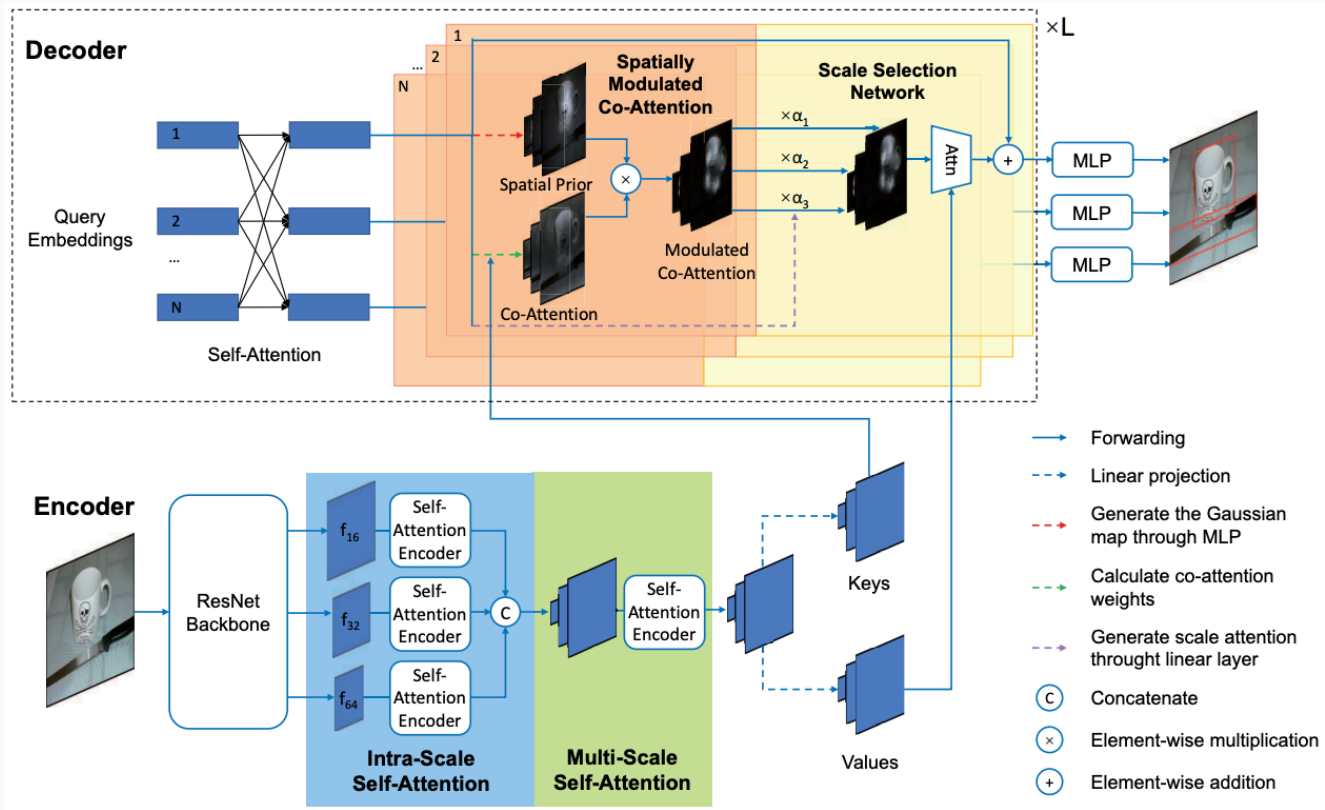
DETR и аналоги

- Обзор:



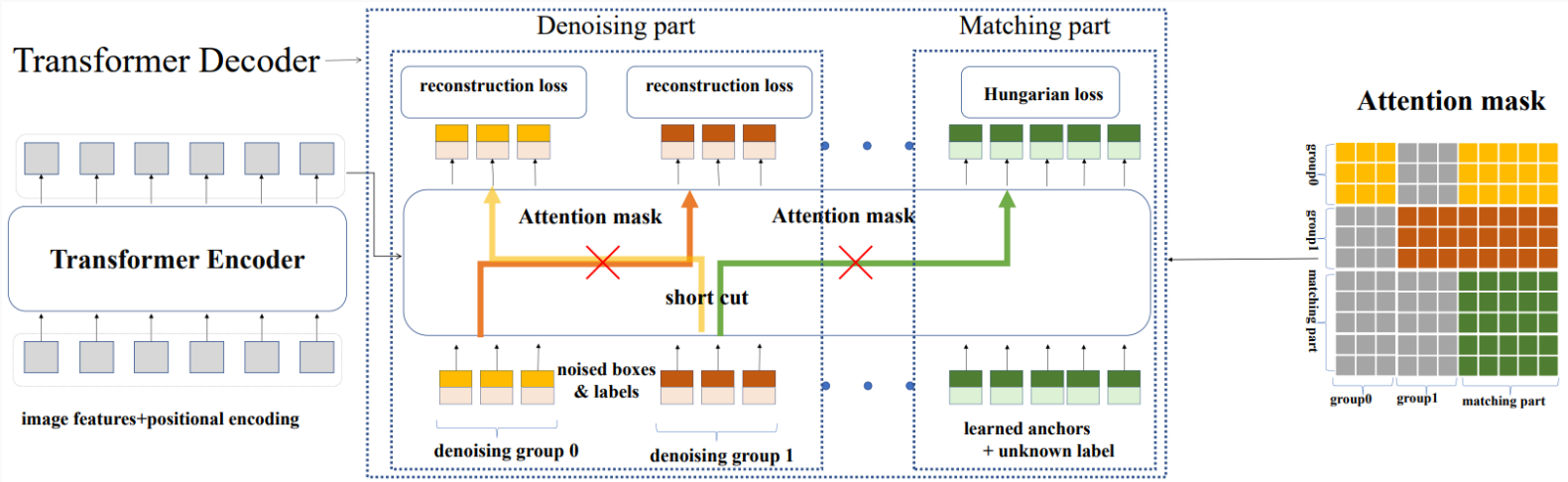
DETR и аналоги

- SMCA DETR (Spatially Modulated Co-Attention DETR; Gao et al., 2021) модулирует веса внимания через пространственные априорные распределения; то, что рядом, более важно: пространственные веса на основе предсказанных смещений относительно опорных точек



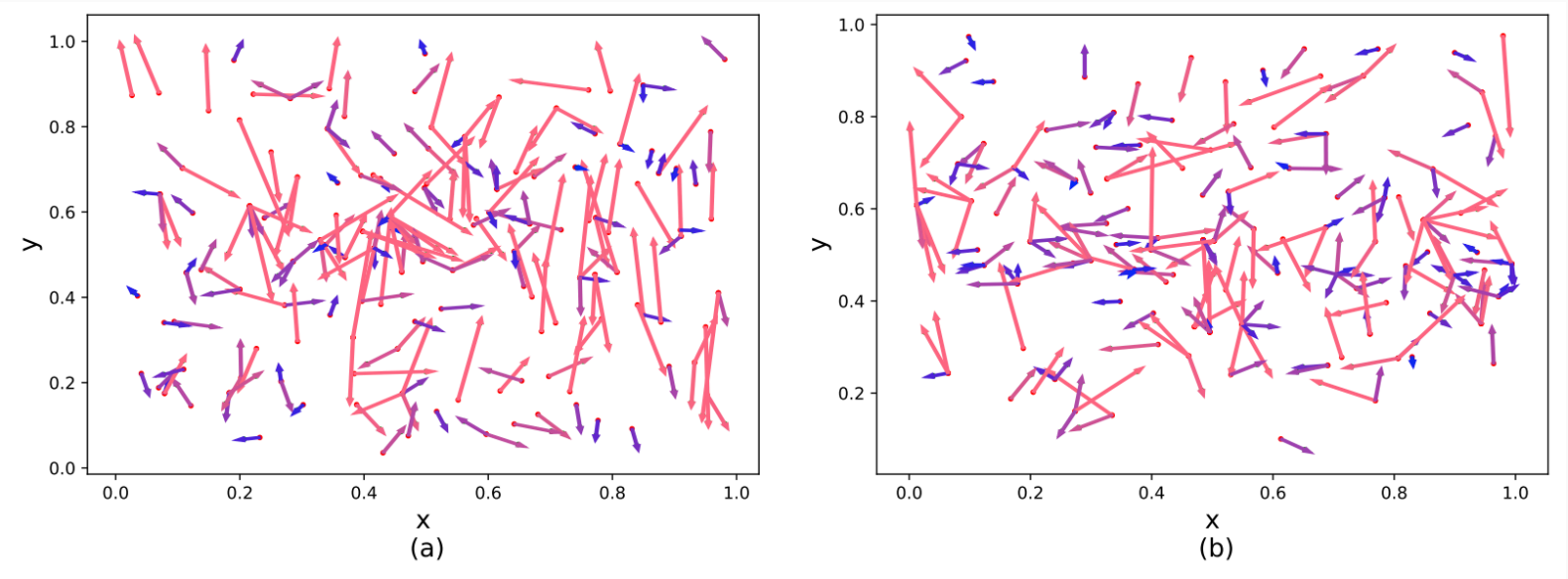
DETR и аналоги

- Другая проблема происходит от оптимального паросочетания: небольшое изменение весов может привести к совершенно другому результату, предсказания нестабильны, особенно в начале обучения
- DN-DETR (Li et al., 2022): давайте добавим другую цель обучения – реконструкцию зашумленных истинных прямоугольников, которые подаются с дополнительными запросами



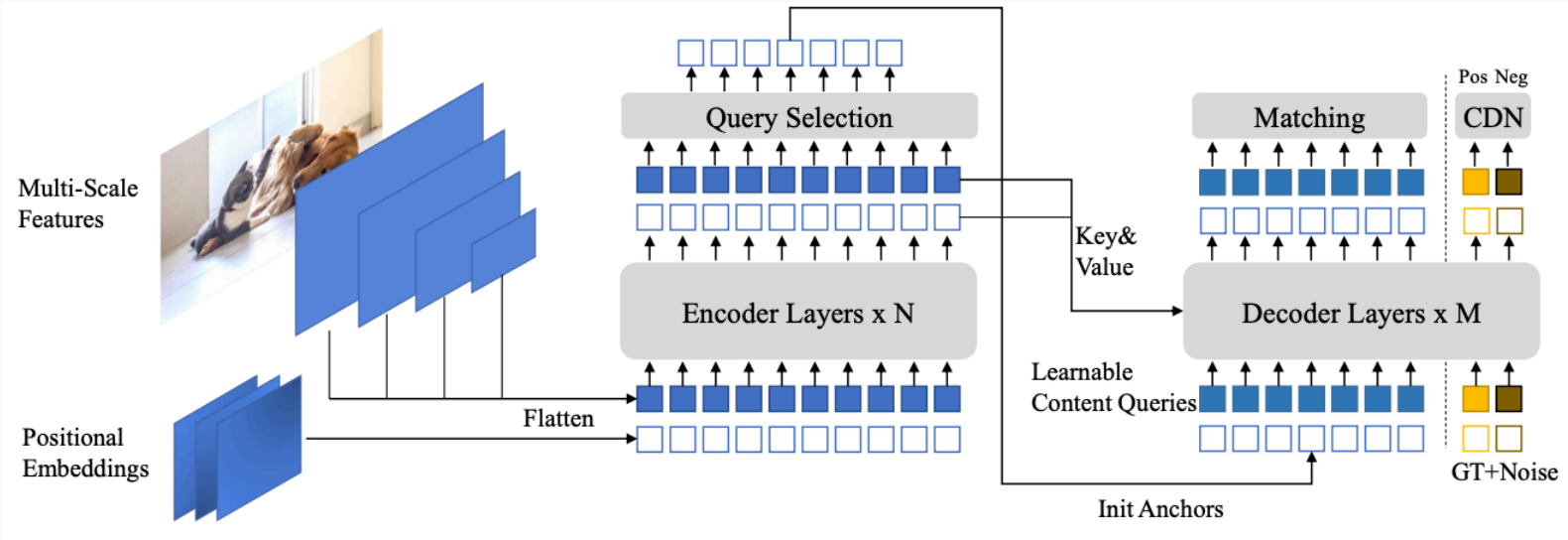
DETR и аналоги

- Это улучшает предсказание смещений: каждый запрос должен предсказывать очень разные смещения из-за нестабильности сопоставления, а запросы шумоподавления служат “хорошими якорями” для предсказания смещений
- В результате якоря располагаются ближе к своим целям (DAB-DETR слева, DN-DETR справа):



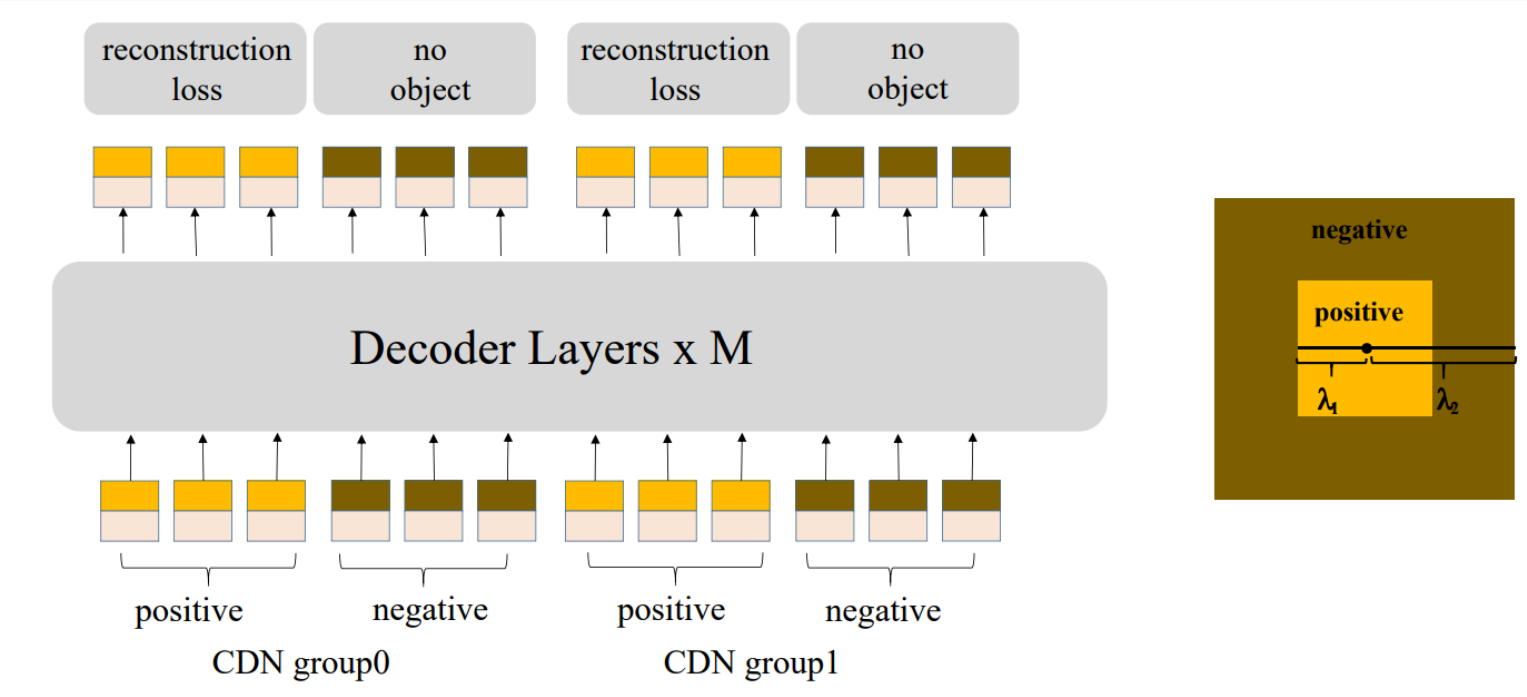
DETR и аналоги

- DINO (DETR with Improved DeNoising Anchor Boxes; Zhang et al., 2022): модификация DETR, основанная на идее DN-DETR
- Базовая структура более или менее та же:



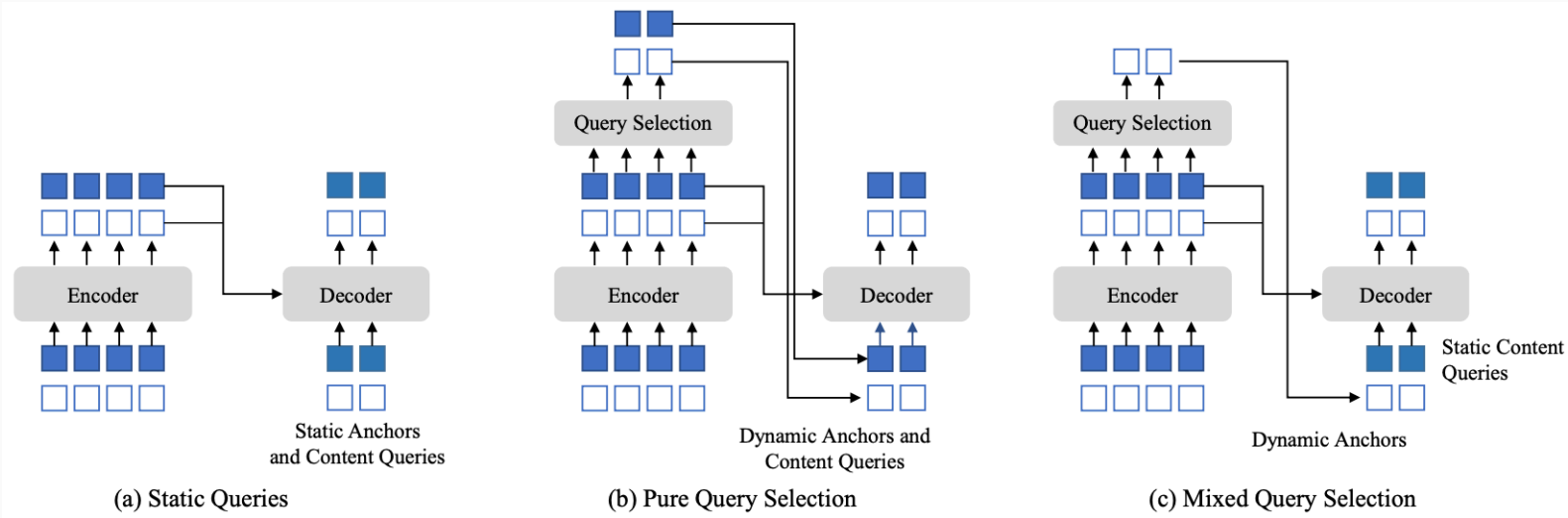
DETR и аналоги

- DN-DETR нужны хорошие якоря для улучшения обучения смещений; но что насчет плохих якорей, у которых нет близких истинных объектов?
- DINO добавляет отрицательные примеры для смягчения этого; часть входов – это якоря с большим шумом, которые должны быть отклонены



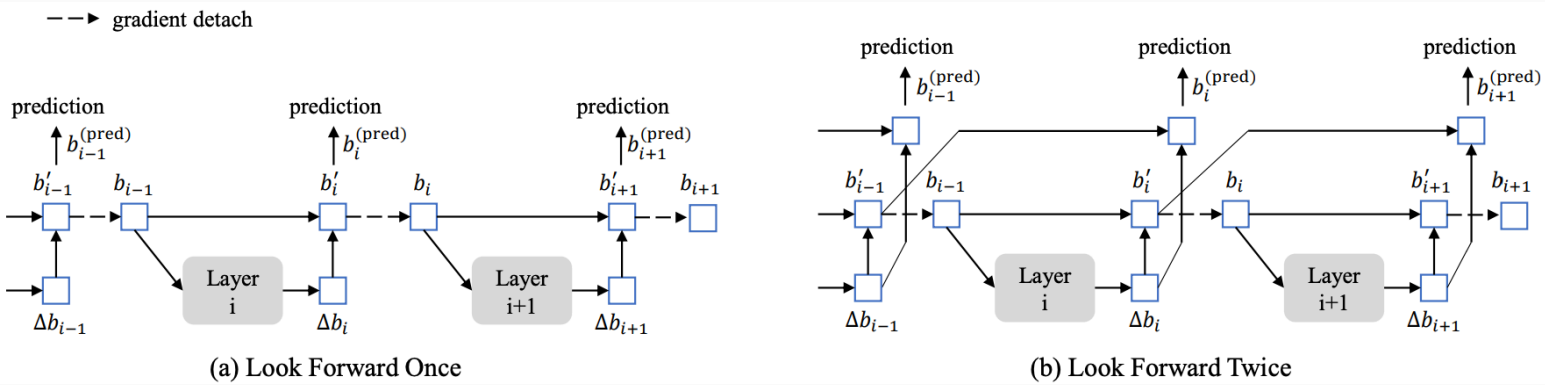
DETR и аналоги

- Смешанный выбор запросов: позиционные запросы инициализируются из предложений кодировщика, а содержательные запросы остаются независимыми



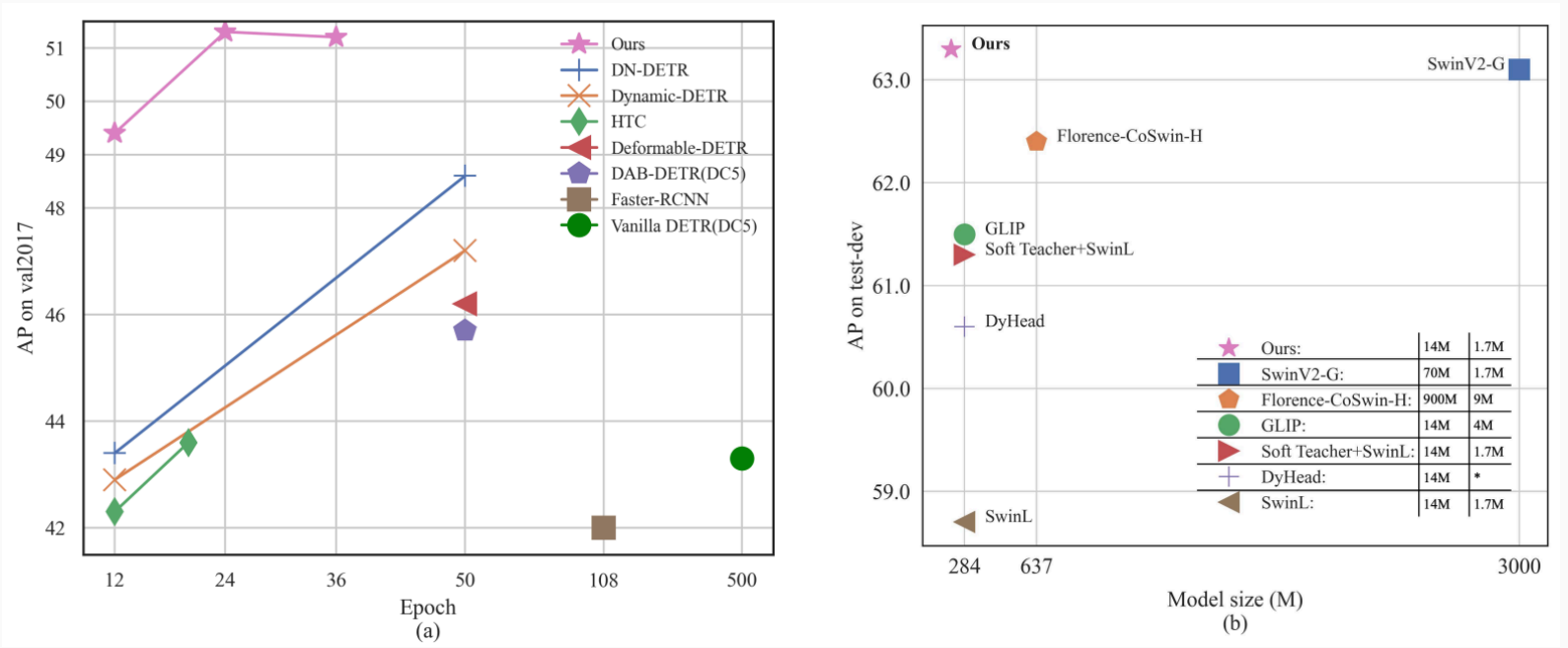
DETR и аналоги

- Двойной lookahead: качество ограничивающего прямоугольника зависит от текущего якоря (позиционного запроса) и текущего предсказания смещения
- Deformable DETR блокировал обратное распространение градиента для стабилизации обучения: параметры слоя i обновляются на основе вспомогательной функции потерь от прямоугольников b_i
- DINO блокирует градиенты еще на один шаг дальше, поэтому параметры слоя i обновляются на основе вспомогательной функции потерь от прямоугольников b_i и b_{i+1}



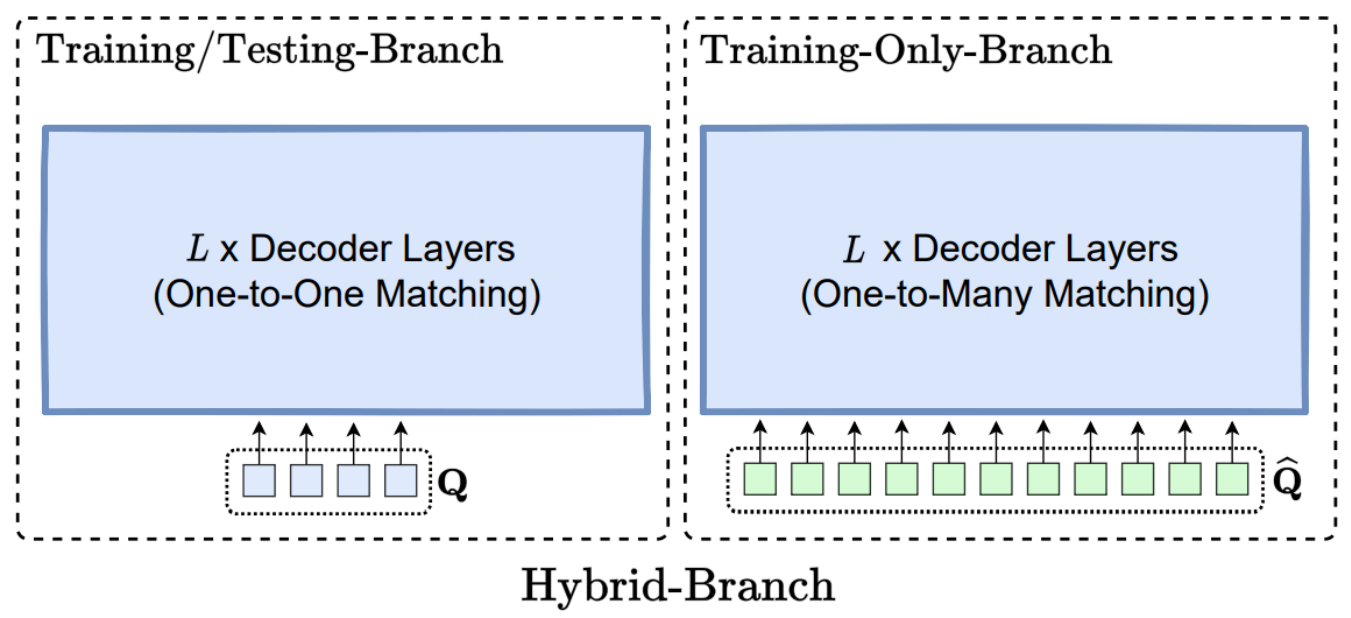
DETR и аналоги

- Очень хорошие результаты, быстрое обучение (слева с ResNet-50 backbone)



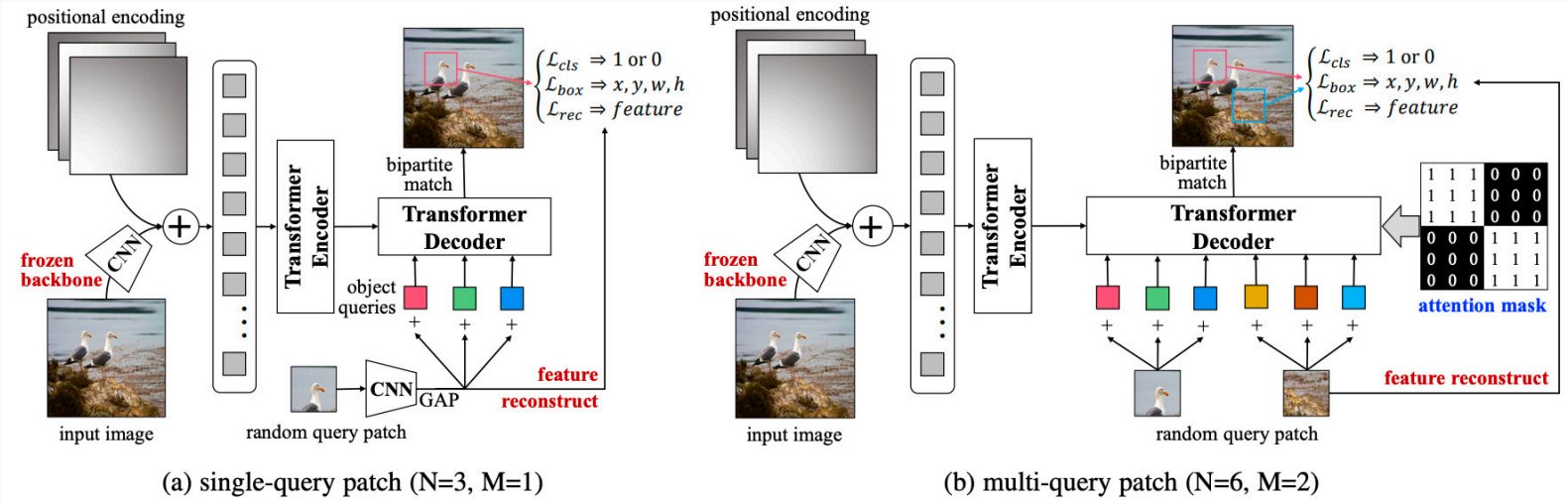
DETR и аналоги

- H-Deformable-DETR (Jia et al., 2023) – очень простая и интересная идея
- У нас очень мало положительных примеров, так что давайте их копировать!
- При обучении мы добавляем дополнительные запросы, которые должны предсказывать те же истинные прямоугольники, с масками внимания, которые разделяют группы запросов



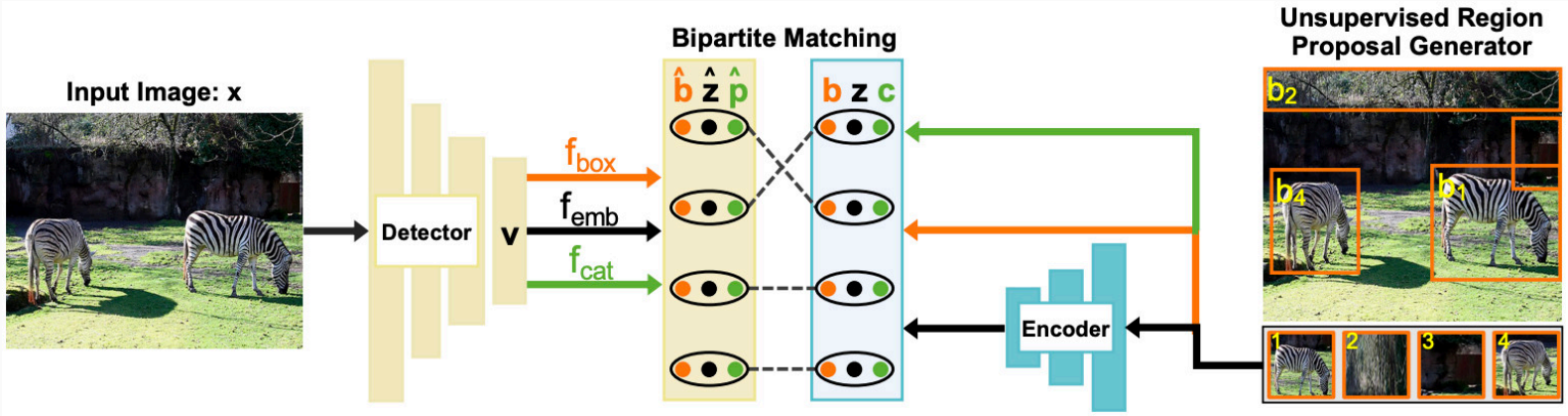
DETR и аналоги

- Как еще мы можем исправить долгое время обучения DETR?
- UP-DETR (Dai et al., 2021) – давайте добавим предобучение без обучения, self-supervised; мы берем случайный кусок и просим DETR предсказать его положение и восстановить его признаки (это нужно, потому что у нас нет меток классификации, но нужно сохранить семантику):



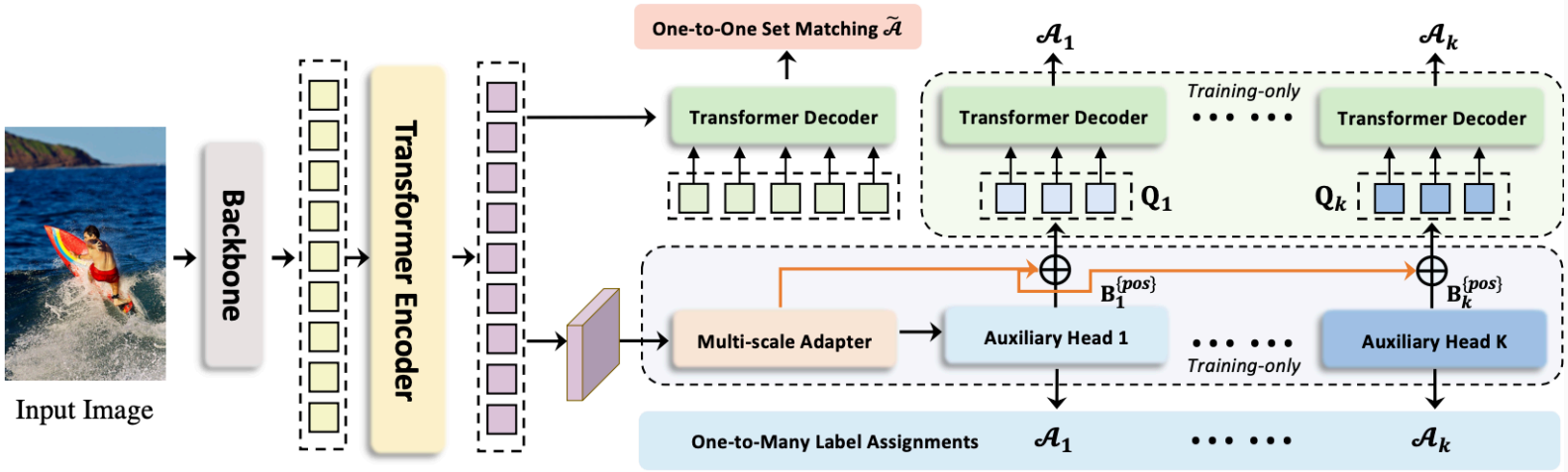
DETR и аналоги

- DETReg (Bar et al., 2022) – вместо случайных кропов давайте используем объекты или области, найденные другими алгоритмами, возможно, даже selective search
- Мы также можем предсказывать вложения для этих псевдо-объектов



DETR и аналоги

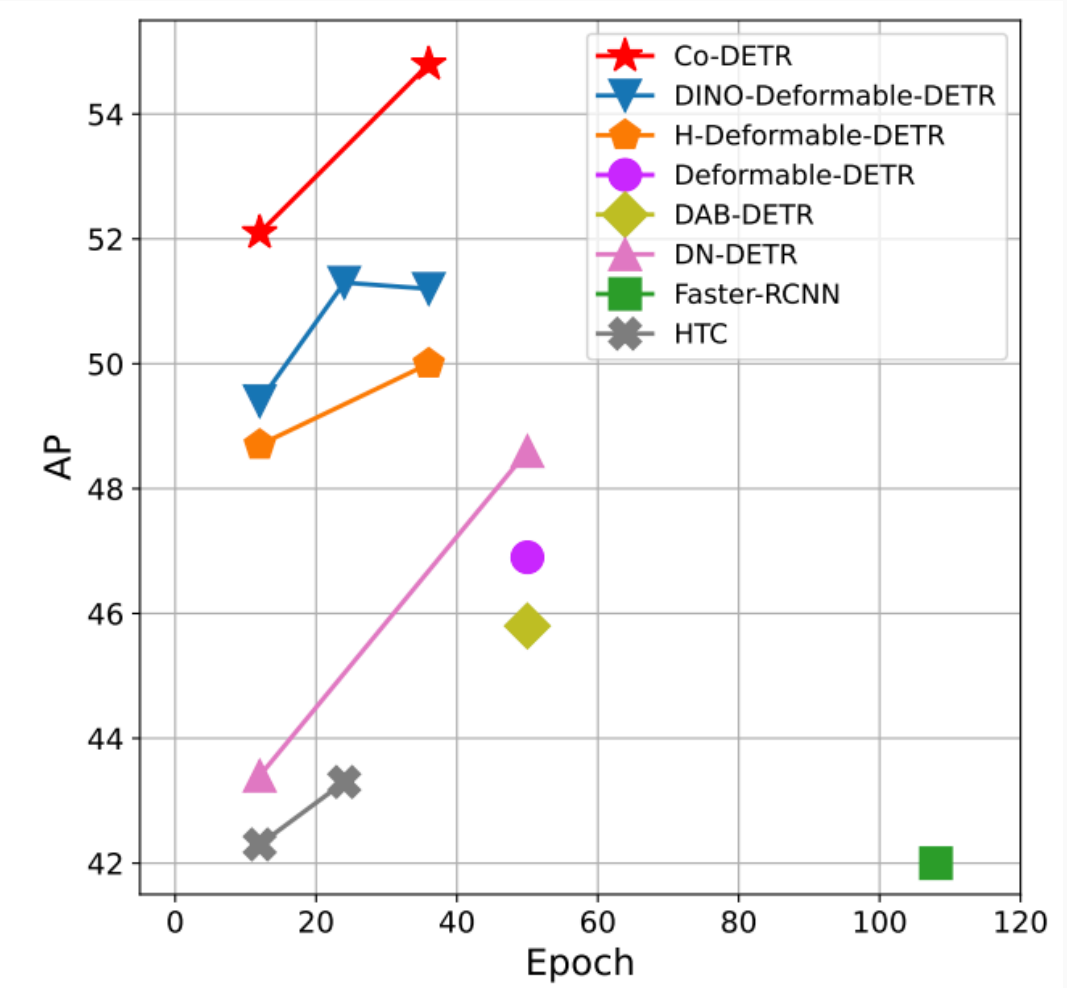
- Co-DETR (Zong et al., 2023) добавляет несколько вспомогательных голов из других object detection моделей; новые положительные примеры через паросочетания



Head i	Loss $\mathcal{L}_i$	Assignment $\mathcal{A}_i$		
		$\{pos\}$, $\{neg\}$ Generation	$\mathbf{P}_i$ Generation	$\mathbf{B}_i^{\{pos\}}$ Generation
Faster-RCNN [27]	cls: CE loss, reg: GIoU loss	$\{pos\}$: IoU(proposal, gt)>0.5 $\{neg\}$: IoU(proposal, gt)<0.5	$\{pos\}$: gt labels, offset(proposal, gt) $\{neg\}$: gt labels	positive proposals (x_1, y_1, x_2, y_2)
ATSS [40]	cls: Focal loss reg: GIoU, BCE loss	$\{pos\}$:IoU(anchor, gt)>(mean+std) $\{neg\}$: IoU(anchor, gt)<(mean+std)	$\{pos\}$: gt labels, offset(anchor, gt), centerness $\{neg\}$: gt labels	positive anchors (x_1, y_1, x_2, y_2)
RetinaNet [21]	cls: Focal loss reg: GIoU Loss	$\{pos\}$: IoU(anchor, gt)>0.5 $\{neg\}$: IoU(anchor, gt)<0.4	$\{pos\}$: gt labels, offset(anchor, gt) $\{neg\}$: gt labels	positive anchors (x_1, y_1, x_2, y_2)
FCOS [31]	cls: Focal Loss reg: GIoU, BCE loss	$\{pos\}$: points inside gt center area $\{neg\}$: points outside gt center area	$\{pos\}$: gt labels, ltrb distance, centerness $\{neg\}$: gt labels	FCOS point (cx, cy) $w = h = 8 \times 2^{2+j}$

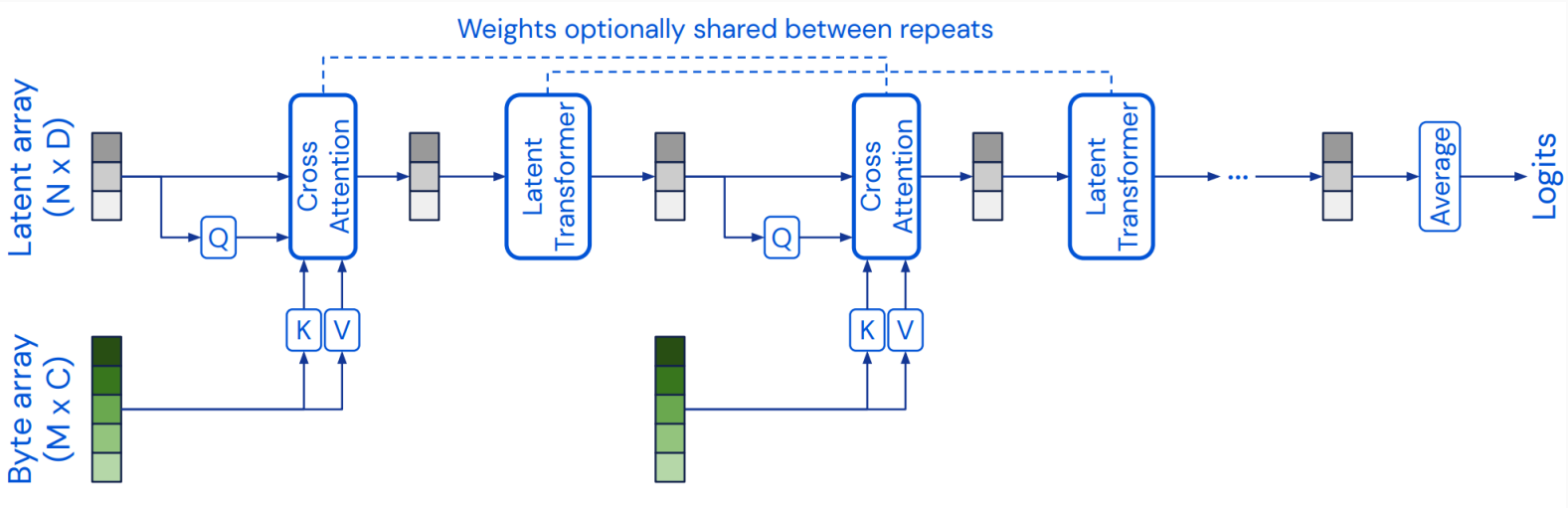
DETR и аналоги

- График для ResNet-50 backbone:



Perceiver

- *Perceiver* от *DeepMind* (Jaegle et al., 2021a) – это архитектура общего назначения, которая может обрабатывать множество различных модальностей



- Основная идея – избежать квадратичной сложности, используя латентные единицы меньшей размерности
- Это квадратично по числу запросов, поэтому мы используем небольшой вектор латентных переменных для запросов и добавляем большие массивы для K и V

Perceiver

- Обучен на изображениях, облаках точек, аудио и видео, без специальных кодировщиков и без изменения архитектуры

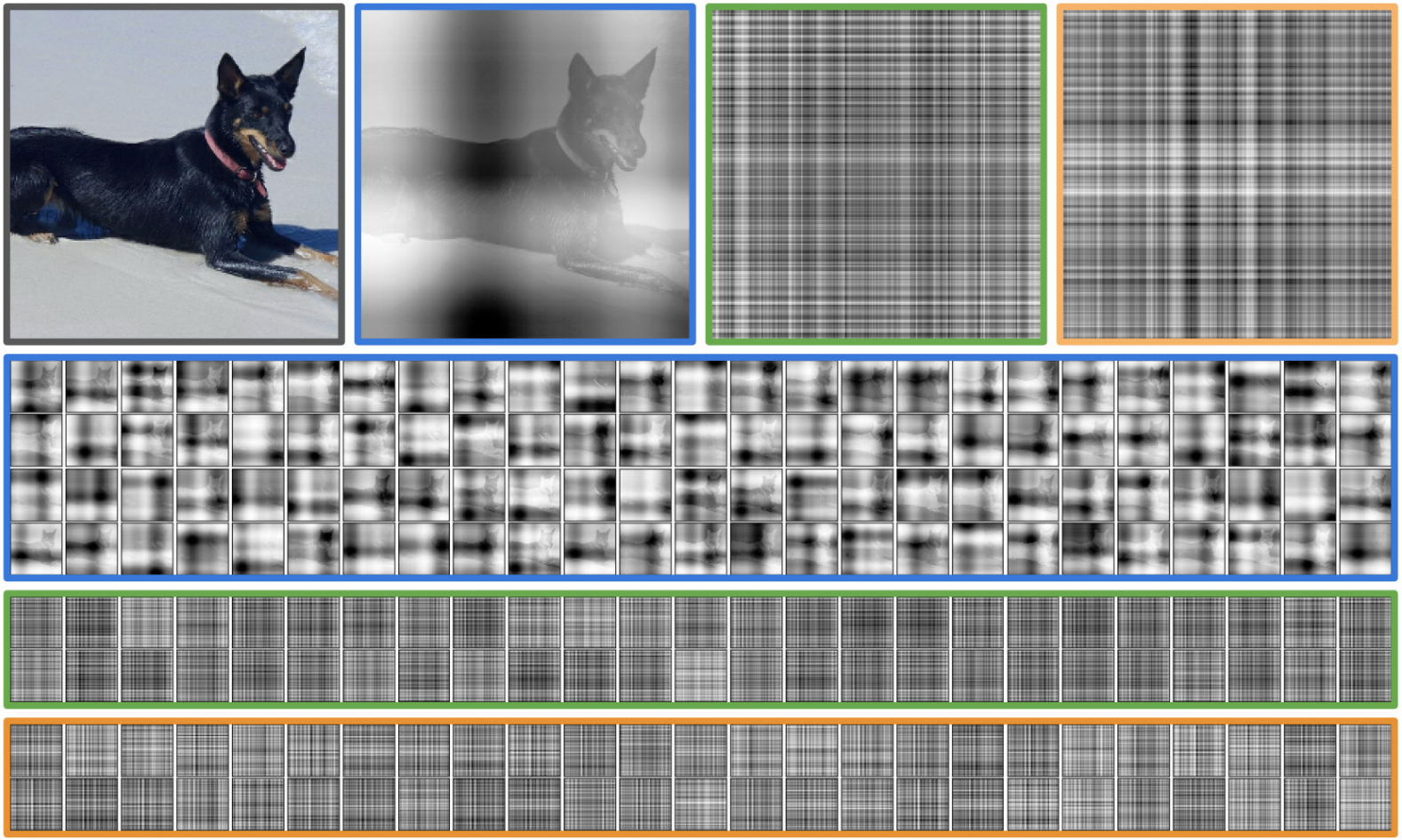


ResNet-50 (He et al., 2016)	77.6
ViT-B-16 (Dosovitskiy et al., 2021)	77.9
ResNet-50 (FF)	73.5
ViT-B-16 (FF)	76.7
Transformer (64x64, FF)	57.0
Perceiver (FF)	78.0

	Raw	Perm.	Input RF
ResNet-50 (FF)	73.5	39.4	49
ViT-B-16 (FF)	76.7	61.7	256
Transformer (64x64) (FF)	57.0	57.0	4,096
Perceiver:			
(FF)	78.0	78.0	50,176
(Learned pos.)	70.9	70.9	50,176

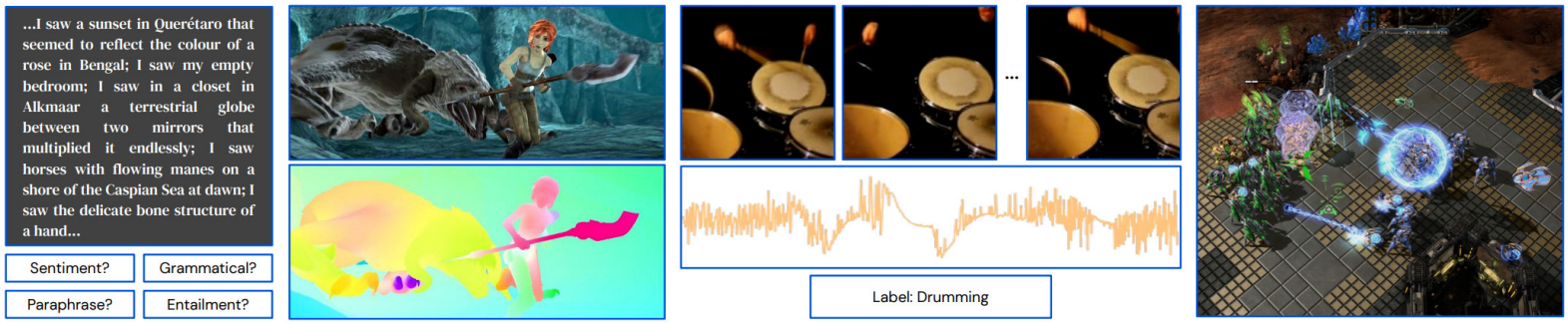
Perceiver

- Карты внимания опускаются до отдельных пикселей:



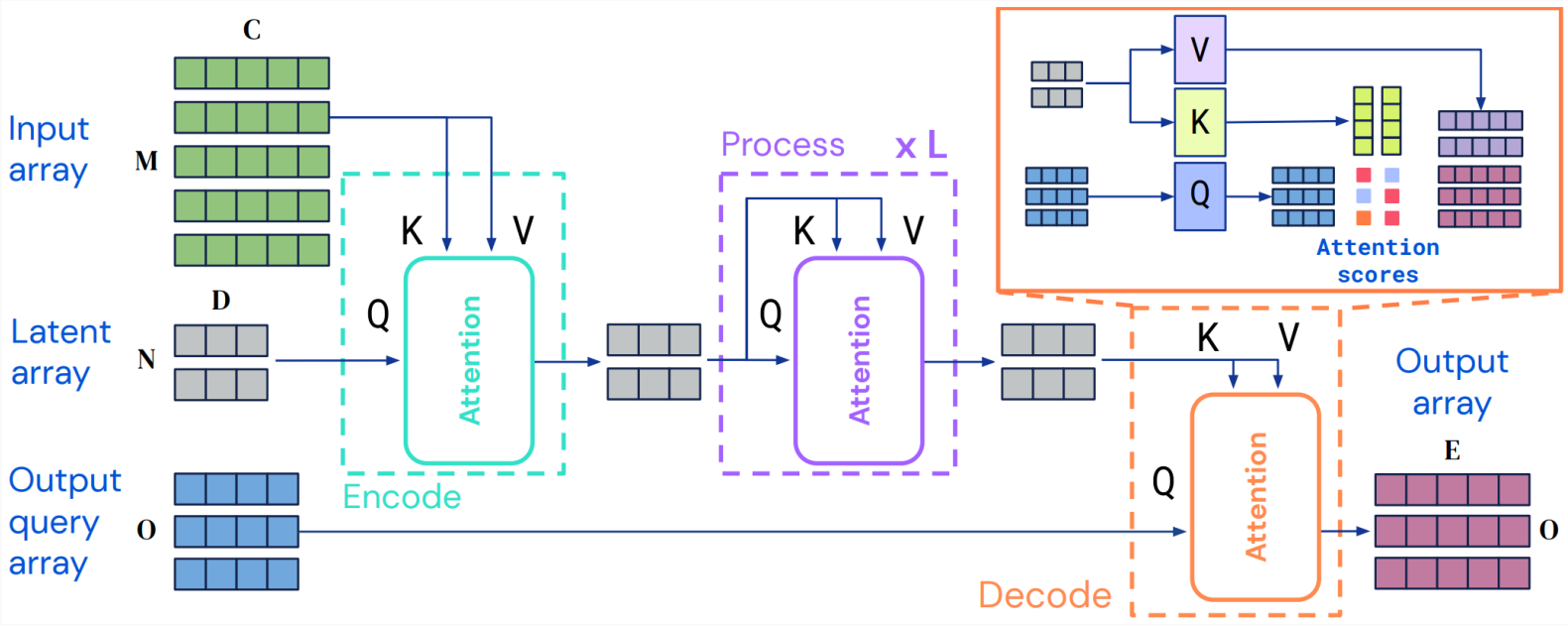
Perceiver

- *Perceiver IO* (Jaegle et al., 2021b), следующая версия, которая также позволяет строить большие *выходы*
- Может обрабатывать еще более разнообразные структурированные данные: многозадачное понимание языка, плотные визуальные задачи (например, optical flow), гибридные плотные/разреженные мультимодальные задачи (например, автокодировщик для видео+аудио), задачи с символьными выходами (например, StarCraft II) и так далее



Perceiver

- Архитектура Perceiver IO:



- Фактические выходные запросы конструируются автоматически путем комбинирования набора векторов, которые описывают свойства текущего выхода

Спасибо!

Спасибо за внимание!



