

Эмбединги (векторное представление токена, описывающее его свойства)

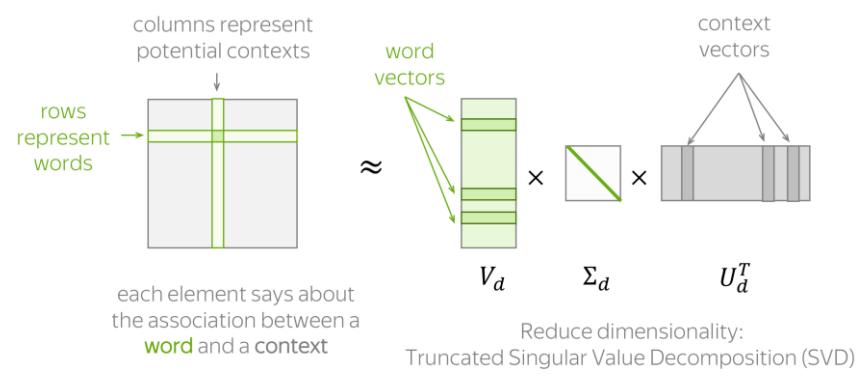
Самый основной вопрос – как работать со словами? Их нужно как-то кодировать, чтобы использовать их в нашем коде. То есть представлять слово числом или набором чисел.

ONE-HOT-кодирование (одно значение вектора 1, остальные 0) – все эмбединги на одинаковом расстоянии друг от друга, что не несет никакой полезной информации. То есть «собака» и «кот» на том же расстоянии, что «собака» и «вертолет». Такое кодирование порождает очень разреженную матрицу => такой **подход совсем не учитывает смысла слов** (а только их наличие)

Аналитический учет контекста (подсчетом) – идея состоит в пародировании людей. Мы понимаем незнакомые слова, встречая их в разном контексте => будем пытаться разворачивать идею «если у слов схожий контекст – значит слова тоже схожи». И начнем мы с простых подсчетных методов.

Таким образом: **хотим положить информацию о контексте слова в его векторное представление**. Хотим реализовать следующее разложение для матрицы слова*контекст на отдельные матрицы с векторами слов и контекстов:

Для реализации необходимо: понять как учитывать контекст и как заполнять эту матрицу



Простой метод частотный метод

Контекст – слова в скользящем окне размера L.

Элемент матрицы $N(w,C)$ – кол-во раз, когда слово w появилось в контексте C.

Положительная точечная взаимная информация (PPMI)

Контекст – слова в скользящем окне размера L.

Для элемента матрицы тут используется более сложная идея (SOTA-решение для до нейросетевых моделей) (формула для расчета слова внутри контекста)

- $PPMI(w, c) = \max(0, PMI(w, c))$, where

$$PMI(w, c) = \log \frac{P(w, c)}{P(w)P(c)} = \log \frac{N(w, c) / (w, c)}{N(w)N(c)}$$

Латентный семантический анализ (LSA) == tfidf

Контекст – документ d из коллекции документов D

Элемент матрицы – формула ниже

$$tf-idf(w, d, D) = tf(w, d) \cdot idf(w, D)$$

term frequency

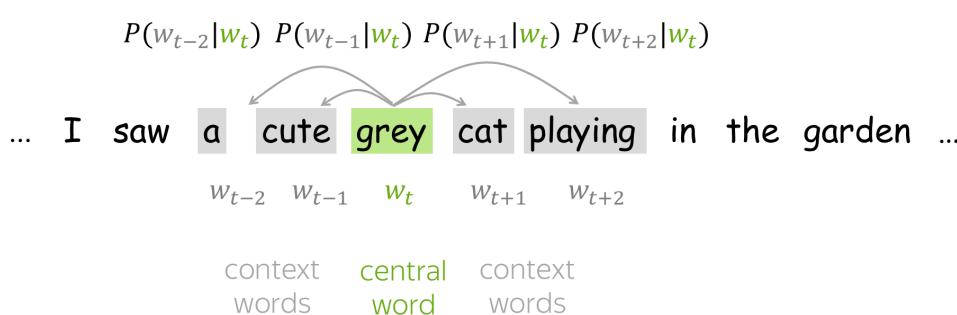
$$\log \frac{|D|}{|\{d \in D : w \in d\}|}$$

inverse document frequency

Word2Vec – предыдущие методы использовали

контекст напрямую, теперь идея в том, чтобы обучить векторы слов, чтобы они предсказывали контекст.

Хотим взять большой корпус текстов и передвигаясь по нему скользящим окном регулировать векторы слов так, чтобы увеличивать вероятности слов в контексте и уменьшать слова вне контекста. Эмпирически доказано, что размер окна очень влияет на обучение W2V: больший размер окна чаще выделяет тематическую похожесть слов («поварок», «собака», «лай»), в то время как меньшие окна выделяют синтаксическую и функциональную похожесть: («пудель», «питбуль»)



Как будем считать loss и какой функционал оптимизировать? Для каждой позиции $t = 1, \dots, T$ в корпусе текстов W2V предсказывает контекст в скользящем окне размера « m » с центральным словом w_t (тета – параметры модели, т е все векторы центрального и контекстного представлений). Можно написать правдоподобии для каждой позиции и в качестве лосса взять отрицательное лог-правдоподобие (тк хотим его минимизировать):

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m, \\ j \neq 0}} P(w_{t+j} | w_t, \theta)$$

$$\text{Loss} = J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m, \\ j \neq 0}} \log P(w_{t+j} | w_t, \theta)$$

↑ agrees with our plan above ↗ go over text ↑ with a sliding window ↗ compute probability of the context word given the central

Остается понять, как находить такие вероятности нахождения слова в контексте другого.

Для этого составим две матрицы: центральных и контекстных представлений с кол-вом строк = размеру словаря и кол-вом столбцов = желаемому размеру эмбединга. Под сходством слов будет понимать скалярное произведение их векторов (больше произведение – более похожие слова). Соответственно контекстная близость слова «о» (outer) для слова «с» (center) = центральный вектор «с» * контекстный вектор «о». С таким опр-м

близость вероятность «о» в контексте «с» (используем софт-макс, чтобы получить распределение по вероятностям, в числителе делим на суммарную похожесть «с» со всеми словами словаря:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Dot product: measures similarity of o and c
Larger dot product = larger probability
Normalize over entire vocabulary to get probability distribution

Как будем учить модель? Градиентным спуском, один шаг – одна пара центр. и контекстного слова

Для обучения модели создадим датасет из пар центральных слов «с» и каждого его контекстного слова («о», «с»). Подавать в модель будем также попарно.

Все параметры тета – векторы наших представлений, каждый из которых учится классич. град спуском:
 $\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$. (где альфа – learning rate)

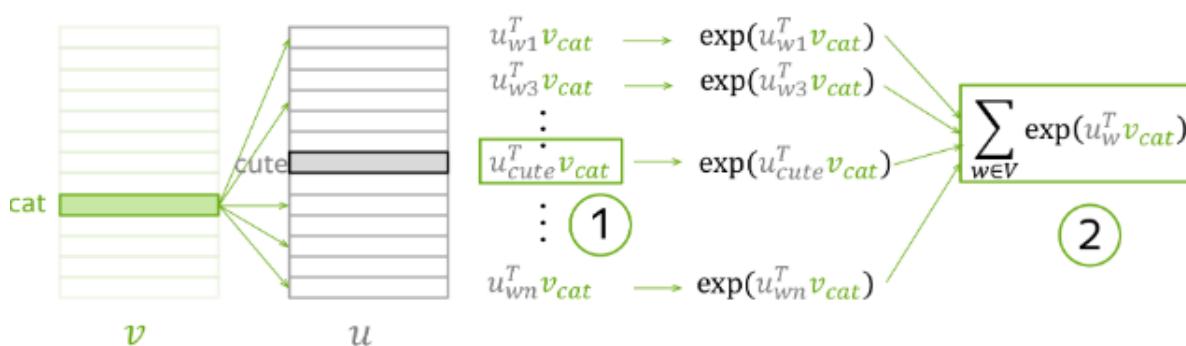
На каждом шаге град спуска будем обновлять одну пару центрального слова (например «cat») и контекстного слова (например «cute»).

Вспомним, как в общем выглядит функционал качества Loss, а далее выпишем лосс для конкретной пары выбранных слов $J_{t,j}(\theta)$:

$$\text{Loss} = J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m, j \neq 0}} \log P(w_{t+j} | w_t, \theta) = \frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m, j \neq 0}} J_{t,j}(\theta).$$

$$J_{t,j}(\theta) = -\log P(cute | cat) = -\log \frac{\exp u_{cute}^T v_{cat}}{\sum_{w \in V} \exp u_w^T v_{cat}} = -u_{cute}^T v_{cat} + \log \sum_{w \in V} \exp u_w^T v_{cat}.$$

На данном шаге в лоссе из центральных представлений присутствует только представление слова «cat», а из контекстных – все представления (для каждого слова словаря). Один такой шаг можно изобразить в виде схемы: берем центральное (зеленое) и контекстное слово (серый), считаем лосс, обновляем центральный вектор для «cat» и все контекстные векторы.



$$J_{t,j}(\theta) = -\underbrace{u_{cute}^T v_{cat}}_{1} + \log \underbrace{\sum_{w \in V} \exp(u_w^T v_{cat})}_{2}$$

$$v_{cat} := v_{cat} - \alpha \frac{\partial J_{t,j}(\theta)}{\partial v_{cat}}$$

$$u_w := u_w - \alpha \frac{\partial J_{t,j}(\theta)}{\partial u_w} \quad \forall w \in V$$

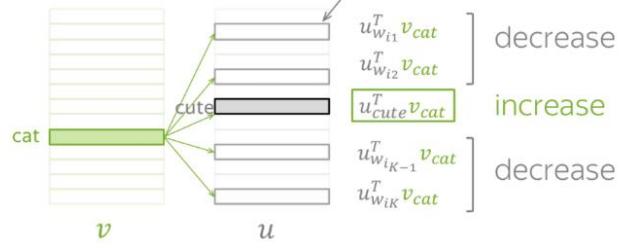
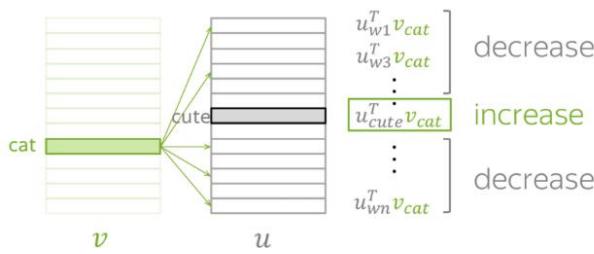
Можно задаться вопросом, зачем нам вообще две разные матрицы (центральная и контекстная)? Это можно объяснить тем, какие векторы обновляются на каждом шаге (только один центральный и все контекстные), поэтому именно центральные векторы отражают истинное значение слова, и именно центральную матрицу мы будем использовать как эмбеддинги слов после обучения (контекстную можно будет выкинуть).

А почему мы на данном одном шаге одновременно увеличиваем близость «cat» и «cute» и уменьшаем близость «cat» и всех других слов, включая ее другие контекстные слова («grey», «playing»)? На самом деле это не страшно, тк мы делаем апдейты для каждого центрального слова и для каждого его контекстного слова, и по итогу в среднем мы будем хорошо сближать слова в контексте и отдалять все другие.

Улучшение (Negative Samplings) – заметим, что до этого на каждом шаге град спуска мы обновляли векторы всей контекстной матрицы – это очень долго и неэффективно, тк время для каждого шага выполнялось за $O(N)$, где N – размер словаря (слишком долго).

Идея: вместо того, чтобы рассматривать все контекстные векторы на каждом шаге, хотим брать только K рандомных «негативных» (не контекстных) слов для текущего центрального. То есть на каждом шаге будем сближать векторы текущих центрального и контекстного слов и отдалять другие K рандомных.

Negative samples: randomly selected K words



Поскольку наш корпус текстов достаточно большой, то такое обновление не помешает нам эффективно обучать эмбеддинги для слов. При таком улучшении лосс-функция примет такой вид (где сигма = сигмоида, а также если заметить, что сигмоида(-x) = 1 – сигмоида(x), то можно переписать еще в ином виде):

$$J_{t,j}(\theta) = -\log \sigma(u_{cute}^T v_{cat}) - \sum_{w \in \{w_{i_1}, \dots, w_{i_K}\}} \log \sigma(-u_w^T v_{cat}),$$

$$J_{t,j}(\theta) = -\log \sigma(u_{cute}^T v_{cat}) - \sum_{w \in \{w_{i_1}, \dots, w_{i_K}\}} \log(1 - \sigma(u_w^T v_{cat})).$$

Дополнение: как мы отираем K рандомных? В основе лежит идея, что для каждого центр слова есть всего несколько true-контекстных, тогда рандомно взятое слово сильно более вероятно будет не контекстным. Слова будем брать случайно в соответствии с эмпирическим распределением слов в словаре. Пусть $U(w)$ – частоты слов w в словаре $\Rightarrow W2V$ использует видоизмененное распределение $U^{(3/4)}(w)$ для того, чтобы брать редко встречающиеся слова чаще.

Word2Vec подходы (Skip-Gram & C-Bow) –

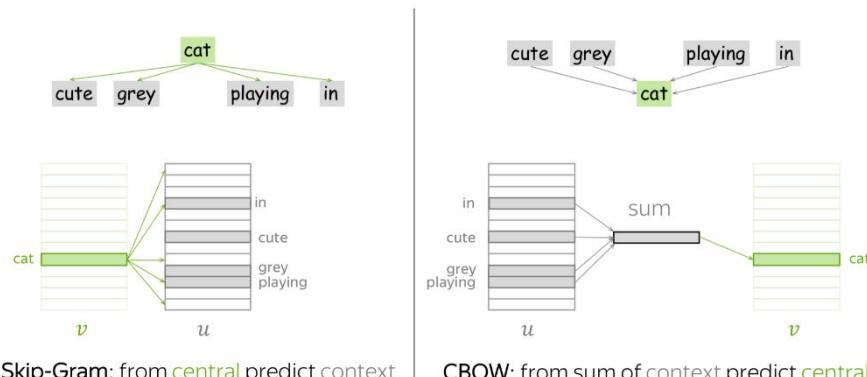
W2V включает в себя два основных подхода:

Skip-Gram – идея, которую мы рассматривали выше: предсказание контекста по

центральному слову. Skip-Gram с идеей negative samplings – самый популярный подход

C-Bow (Continuous Bag Of Words) – идея предсказание центрального слова по сумме его контекстных векторов

... I saw a cute grey cat playing in the garden ...



Skip-Gram: from central predict context (one at a time)

CBOW: from sum of context predict central

Гиперпараметры

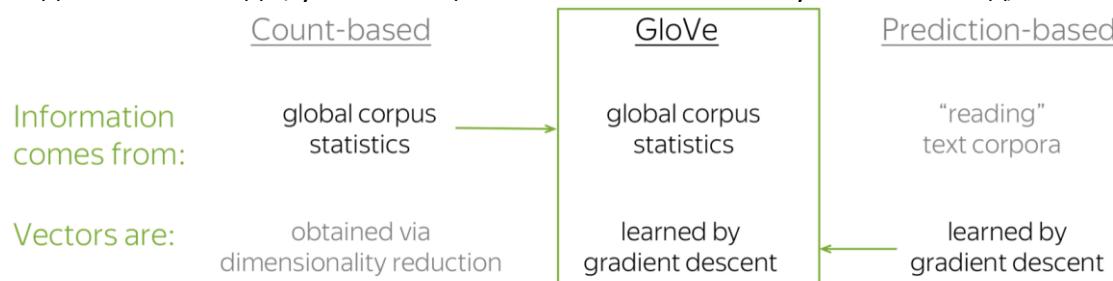
Модель – Skip-Gram with negative samplings

Кол-во negative samplings – маленькие датасеты (15-20), большие датасеты (2-5)

Размерность представлений – можно использовать (300, 150, 50)

Размер окна – 5-10

GloVe (Global Vectors for Word Representations) – идея данного подхода в том, чтобы совместить подсчетные методы, учитывающие частотность слов и обучаемый метод, такой как Word2Vec.



GloVe использует для расчета функции потерь ту же идею, что и первый подсчетный метод, учитывающий кол-во встречаемости данного слова в конкретном контексте. В ФП также остаются центр и контекстные векторы слов, но добавляется bias для каждого слова и учет частотности. Также контролирует влияние частых и редких слов: штрафуем редкие слова и не даем сильно вырастать очень частым словам. Сбор датасета и подача в модель – схожа с W2V. GloVe также использует обучаемые матрицы параметров – матрицу центральных и матрицу контекстных представлений.

$$J(\theta) = \sum_{w,c \in V} f(N(w, c)) \cdot (u_c^T v_w + b_c + \bar{b}_w - \log N(w, c))^2$$

Diagram illustrating the components of the loss function:

- context vector
- word vector
- bias terms (also learned)

Weighting function to:

- penalize rare events
- not to over-weight frequent events

$f(x)$ graph:

The graph shows a red curve starting at the origin (0,0) and increasing towards 1. At a point x_{max} , the curve levels off at a value of 1. A vertical dashed line connects x_{max} on the x-axis to the point where the curve meets the line $f(x)=1$. The area under the curve for $x < x_{max}$ is shaded blue.

$$\begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max}, \\ 1 & \text{otherwise.} \end{cases}$$

$\alpha = 0.75, x_{max} = 100$

Как проверить качество построения эмбедингов? Обычно есть два пути: проверить близость слов и качество построение представлений, основываясь на внутренних свойствах слов, на их значении (быстрый метод, не ничего не обещающий на практике); другой вариант построить разные эмбединги и посмотреть какой подход дает лучшее качество на финальной модели (четко говорит, что лучше на практике, но очень долгий и вычислительно тяжелый).

Классификация текстов

Задача классификации текстов очень популярна, и сама по себе, и как компонент сложных пайплайнов.

Постановка задачи – мы предполагаем что у нас есть коллекция документов, где каждый документ состоит из своего числа токенов $X = (x_1, \dots, x_n)$ и соответствующий single-таргет для каждого документа.

Типичная структура классификатора текстов:

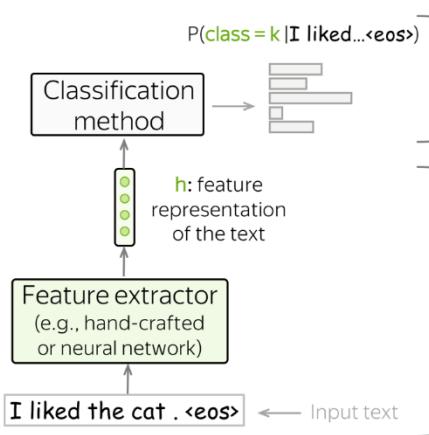
feature extractor – для извлечения фичей из текста / какого-либо превращения его в формат цифр. Может быть как ручным (в классич подходах – bow, tfidf и др), либо обучаемым (нейросети).

classifier – непосредственно сама модель классификации текста на основе фичей из feature extractor.

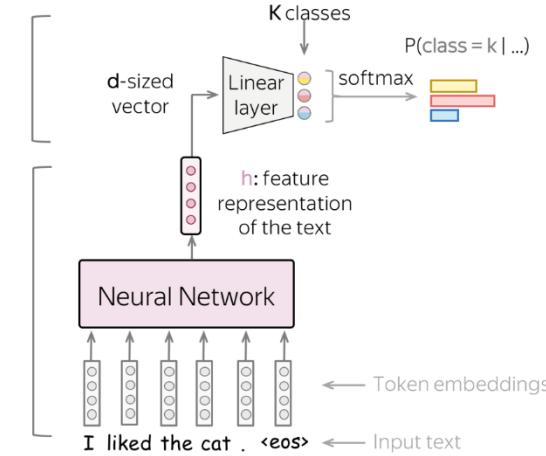
Классический ML – самый простой вариант решить данную задачу. Для данного подхода мы должны воспользоваться ручным извлечением фичей из текста (например, bow/tf-idf). В качестве моделей могут выступать Naïve Bayes, LogReg, SVM, Boostings

Нейросетевой подход – основной идеей нейросетевого подхода является предположение о том, чтобы запихнуть весь пайpline внутрь нейросети так, чтобы feature extraction из текста модель делала внутри себя. То есть в таком подходе на вход мы будем подавать эмбединги токенов текста, а модель сама будет преобразовывать их в признаковое представление для текста. Часть классификатора в нейросети изначально будем представлять достаточно простой структуры – просто как FC-сетка, выдающая на выход вероятности классов.

General Classification Pipeline



Classification with Neural Networks



Как происходит обучение? Обучение на классический лосс в виде cross-entropy:

$$\text{Loss}(p^*, p) = -p^* \log(p) = -\sum_{i=1}^K p_i^* \log(p_i)$$

В процессе обучения мы итеративно улучшаем веса модели: считаем лоссы для единичных примеров (или батчей) и делаем градиентные шаги. На каждом шаге мы максимизируем вероятность, которую модель присваивает правильному классу. В то же время мы минимизируем сумму вероятностей неправильных классов, поскольку сумма всех вероятностей постоянна, увеличивая одну вероятность, мы уменьшаем сумму всех остальных.

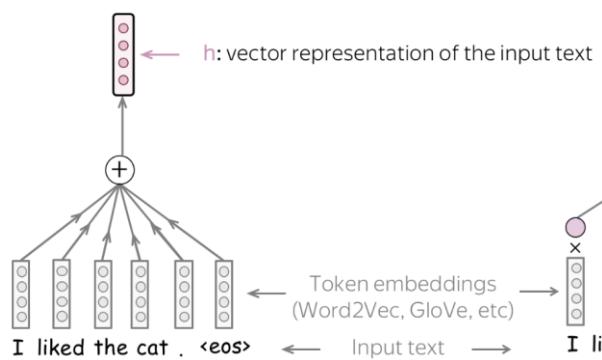
Нейросетевые модели для классификации текстов – мы хотим найти модели, которые умеют принимать наборы разной длины из эмбедингов слов в тексте и выводить вектор фиксированной длины на выход (для подачи в FC-сеть и предсказания класса)

Bag of Embeddings (BoE) and Weighted BoE – наша цель: получить векторное представление для текста и самый простой вариант – это придумать махинацию над эмбедингами токенов в тексте, чтобы превратить их в вектор фиксированной длины, не используя при этом никакую модель.

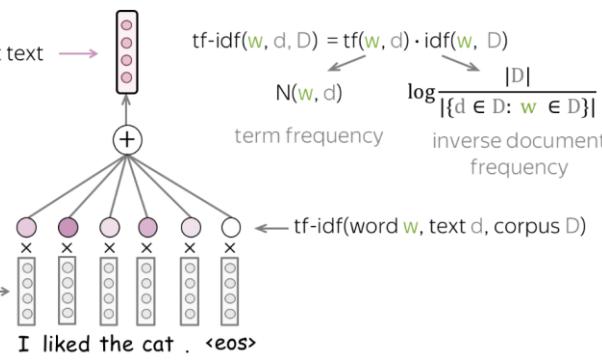
Например, можно использовать сумму всех эмбедингов слов в тексте (BoE) или их взвешенную сумму (Weighted BoE).

BoE вместе с (допустим, Naive Bayes) – это отличный **бейзлайн** для модели классификации текстов.

Sum of embeddings
(Bag of Words, Bag of Embeddings)



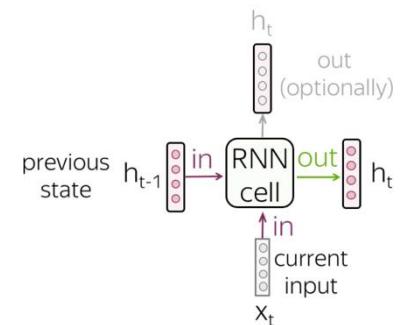
Weighted sum of embeddings
(e.g., using tf-idf weights)



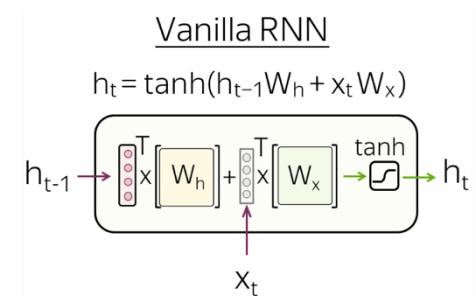
Замечание: важно понимать, что BoE и BoW – это сильно разные вещи. BoE – это сумма эмбедингов слов для текста, в то время как BoW – сумма one-hot векторов. Предобученные эмбединги (W2V, GloVe, FastText) чувствуют близость похожих слов, поэтому эта близость будет учитываться в BoE, в отличие от BoW, в котором эта близость не может быть детектирована посредством one-hot векторов.

Recurrent Models – идея RNN сетей в том, чтобы пародировать восприятия текста человека, то есть «читать и запоминать его» в некоторой последовательности, надеясь, что на каждом шагу сеть будет запоминать то, что «прочитала» до этого. На каждом шагу модель RNN получает («читает») новый вектор (эмбединг нового токена) и предыдущее скрытое состояние модели (свое прошлое состояние «памяти» до прочтения нового токена). На основе входных данных ячейка RNN обновляет свое скрытое состояние («свою память»), дополняя его информацией о новом поступившем векторе. Таким образом RNN итеративно «читает» весь текст, обновляя свое скрытое состояние на каждом шагу.

Замечание: сама структура ячейки RNN не изменяется, то есть техника обновления скрытого состояния остается постоянной.



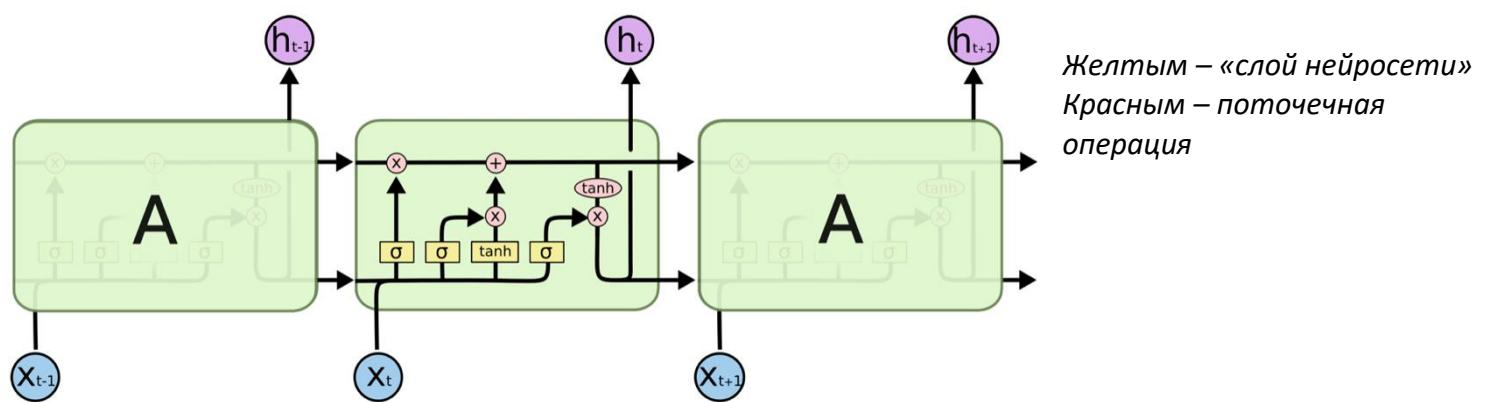
Vanilla RNN – самая простая архитектура RNN ячейки, в которой старое скрытое состояние линейно умножается на вектор весов и складывается с соотв произведением другого вектора весов на вектор эмбединга нового токена, а далее применяется нелинейность (чаще всего – \tanh). Vanilla RNNs страдает от проблем затухания и взрыва градиентов, из-за которых сеть либо «имеет слишком короткую память» (затухание), либо произойдет переполнение типа данных из-за слишком больших значений (взрыв). Для решения этих проблем существуют более сложные виды ячеек.



LSTM (Long short-term memory) – основной идеей RNNs является то, что сеть «запоминает» то, что она «прочла» до этого. И при стандартном Vanilla RNN подходе взвешенного складывания нового токена и предыдущего скрытого состояния сеть не имеет возможности четко запомнить информацию, далекую от текущей. И именно эту проблему «короткой памяти» решает структура LSTM, которая также, как и обычная RNN имеет chain-like структуру, но в отличие от RNN имеет не 1 слой внутри ячейки (\tanh), а целых 4, взаимодействующих друг с другом.

Ключевым компонентом LSTM является «пронос» предыдущего состояния ячейки (*cell state*) сквозь новую ячейку (верхняя горизонтальная линия в структуре ячейки). Такая «конвейерная лента» состояния ячейки позволяет проносить вперед предыдущую информацию практически без изменений вдоль всей RNN цепочки, воздействуя на предыдущее состояние только некоторыми линейными преобразованиями (нечто похожее на *residual connections* только для RNN).

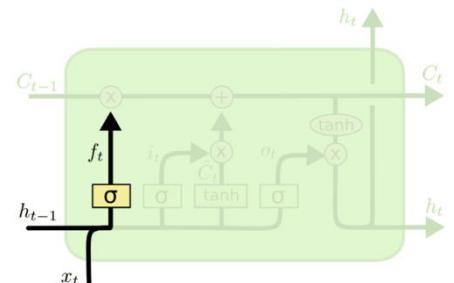
LSTM умеет «добавлять и стирать» информацию из состояния ячейки, данный функция аккуратно регулируется LSTM **фильтрами (gates)**, состоящими из сигмоиды и поточечного умножения. Сигмоида регулирует насколько сильно (от 0 до 1) мы хотим «пропустить» каждую из компонент состояния ячейки. У LSTM есть три такие фильтра для контроля и регулирования состояния ячейки.



Далее пошагово разберем всю структуру ячейки LSTM:

Шаг 1: решить, какую информацию из состояния ячейки $C(t-1)$ мы хотим выкинуть, эту функцию реализует слой сигмоиды (*forget gate layer*), который смотрит на предыдущее скрытое состояние h_t и новый токен x_t , а далее для каждой компоненты состояния ячейки выдает число от 0 до 1, фильтрующее каждую из компонент

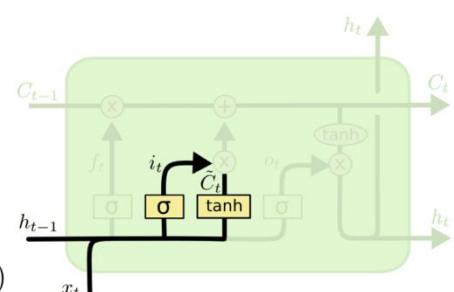
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$



Шаг 2: решить, какую новую информацию на основании h_t и x_t нам добавить в текущее состояние ячейки. Данный фильтр состоит из двух слоев: сигмоидальный (*input gate layer*), отвечающий за то, какие именно компоненты состояния ячейки мы будем обновлять; и слой с \tanh , отвечающий за составления новых значений для обновления. Далее мы объединяем эти слои и обновляем новое состояние ячейки.

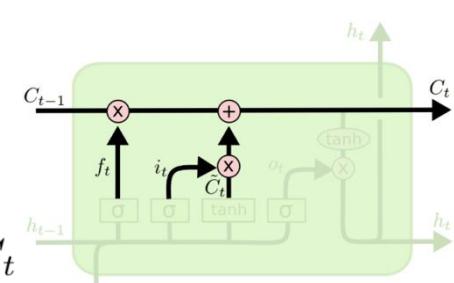
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

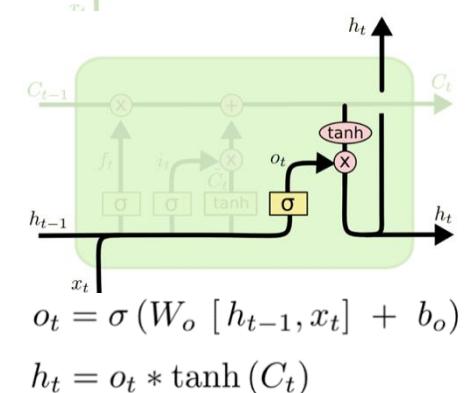


Шаг 3: обновить предыдущее состояние ячейки C_{t-1} на новое состояние C_t , выполнив шаги 1 и 2. То есть мы сначала стираем лишнее из состояния ячейки, умножая ее на f_t (шаг 1), а далее обновляем ее новыми значениями (шаг 2)

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



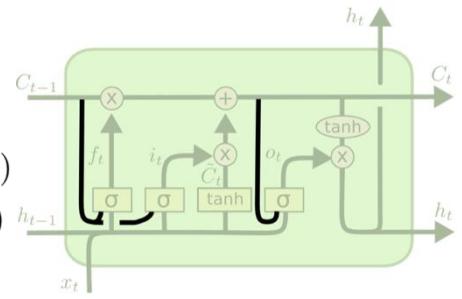
Шаг 4: решить, какой будет *output*, то есть какое будет текущее скрытое состояние h_t на данном шаге. Вывод скрытого состояния будет основан на текущем состоянии ячейки, с предварительно спроектированными всеми значениями ячейки в промежуток от -1 до 1 благодаря \tanh .
Далее фильтруем получившиеся значения еще одним сигмоидальным слоем, решающим какие компоненты состояния ячейки посыпать в *output*, так как нам для выхода не нужна вся запомненная информация из C_t , а лишь та часть, требуемая для решения задачи.



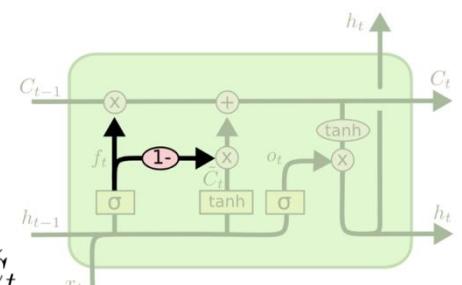
Вариации LSTM – вышеописанное пошаговое применение LSTM описано для самой классической структуры ячейки, которых бывает большое множество (на самом деле каждая новая публикация, использующая LSTM, привносит в структуру некоторые свои изменения). Ниже разобраны основные концепции.

Добавление «глазков» (reephole connections): в таком подходе каждый фильтр, состоящий из сигмоиды, может учитывать предыдущее состояние ячейки C_{t-1} . Такие «глазки» добавляются во все сигмоидальные фильтры

$$\begin{aligned} f_t &= \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \\ o_t &= \sigma(W_o \cdot [C_t, h_{t-1}, x_t] + b_o) \end{aligned}$$



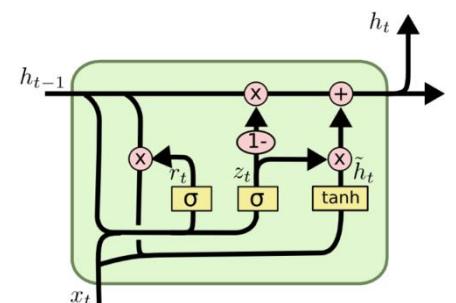
Объединение forget gate & input gate: такая концепция предлагает использовать один фильтр и для забывания, и для добавления новой информации в состояние ячейки (вместо независимого выбора удаление и вставки информации). То есть мы забудем только ту часть, на место которой впоследствии произойдёт вставка.



$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

Gated Recurrent Unit (GRU): в данном подходе предлагается скомбинировать *forget gate* & *input gate* в единый «*update gate*», а также смерджить состояние ячейки и скрытое состояние модели в одну сущность. Такая структура получается заметно проще классической структуры и, к тому же, имеет хороший потенциал для обучения.

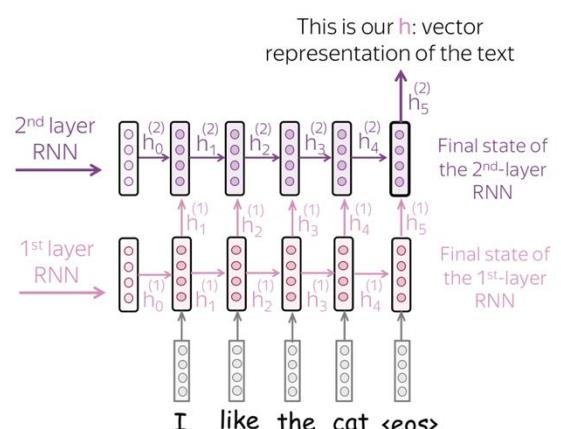
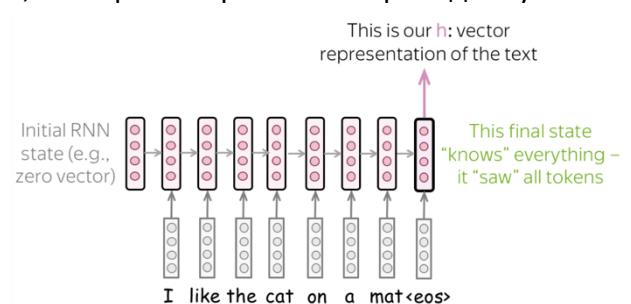
$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) & \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) & h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \end{aligned}$$



RNN для классификации текстов – вспомним, что для задачи классификации мы хотели научиться получать векторные представления фиксированной длины для текстов. Для этого можно использовать RNN модели, поскольку мы можем пользоваться скрытыми состояниями модели, векторы которых имеют фикс длину (гиперпараметр).

Однослочная RNN-сеть: в самом простом варианте мы, предварительно «прочитав» моделью нужный нам текст, возьмем на выходе финальное скрытое состояние модели, как некую кумулятивную информацию, полученную после «прочтения» текста.

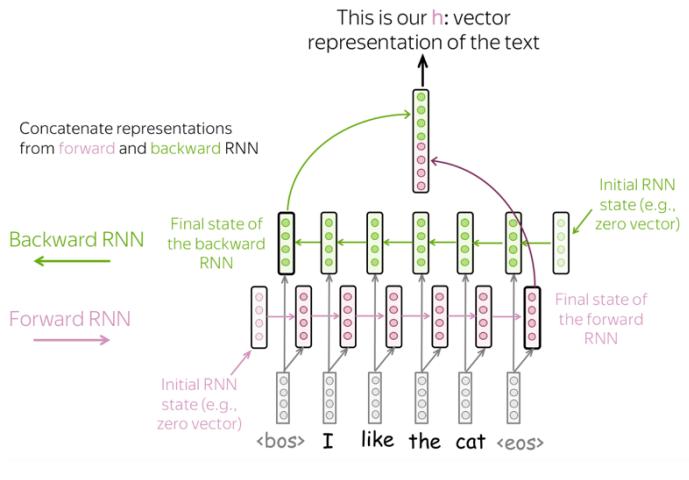
Многослойная RNN-сеть: для получения наилучшего векторного представления текста мы можем добавить поверх 1-го слоя, «читающего» изначальные токены текста, еще целый набор слоев, где каждый последующий слой будет «считывать» *output's* предыдущего слоя. Основная гипотеза в таком подходе состоит в том, что первые слои будут считывать локальные связи (фразы, словосочетания и т.д.), в то время как глубокие слои смогут увидеть более верхнеуровневые связи (темы, главные идеи, топики).



Двусторонняя RNN-сеть: идея в использовании финальных состояний и сети, «прочитавшей» текст слева направо, и сети, «прочитавшей» справа налево. Эта эвристика пытается избавиться от проблемы предыдущих подходов, в которой сеть может легко «забыть» смысл начальной информации. В таком подходе одна сеть будет лучше запоминать конец текста, а другая его начало.

Замечание: если текст большой, то ни один из этих способов не спасет нас от забывания «серединной» информации (даже Bidirectional LSTM)

Замечание: все вышеперечисленные идеи могут использовать не только по отдельности, но и вместе. Мы можем сделать многослойную сеть, в которой N слоев правосторонние, а M слоев – левосторонние.



CNN для классификации текстов – изначально CNN-сети были придуманы для решения CV-задач.

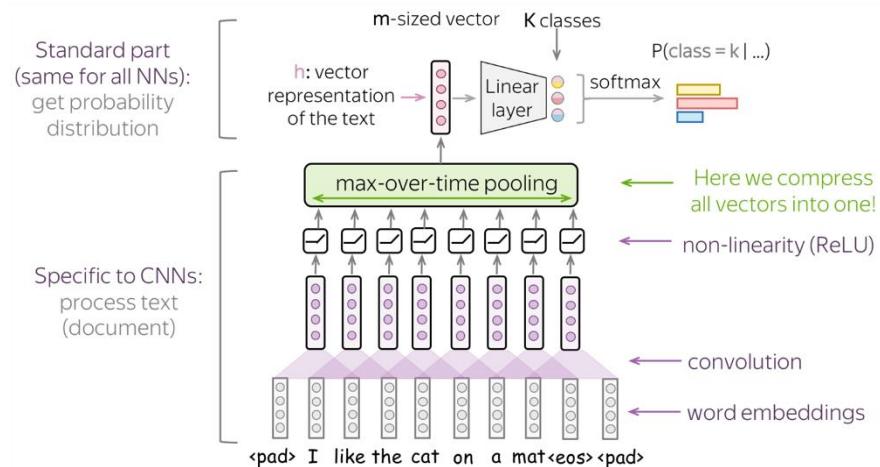
Идея сверток состоит в том (рассмотрим пример с изображением), что нам не важно, где именно изображен на картинке кот, если нам нужно понять есть ли он на картинке (нам важно, что он хоть где-то есть). Поэтому свертка – это проверка некоторой области данных (в данном случае – кусочек картинки) на соответствие с выбранным паттерном (паттерн – обучаемый). То есть каждая операция будет смотреть соответствие обучаемыми паттернами и модель будет понимать, какие из них окажутся наиболее полезными.

Как нам поможет такая идея в текстовых данных?

На самом деле схожий подход для текстов использовать можно, например для *sentiment* анализа текста (позитивный/негативный) нам не так важно, где встречаются опорные слова, а важно, что они действительно присутствуют в тексте.

Типичные блоки CNN-сети – хотим детектировать паттерны, и нас не волнует, где именно эти паттерны присутствуют. Эту идею воплотить в жизнь позволяют два блока:

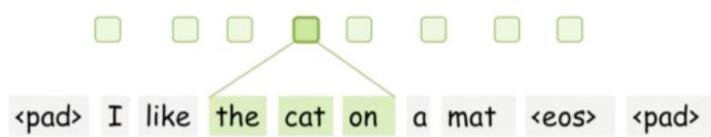
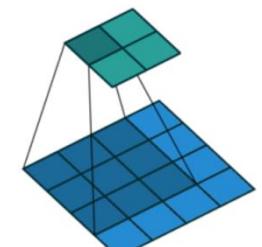
- **блок свертки:** позволяет детектировать совпадения с выбранными паттернами.
- **блок pooling:** агрегирует эти совпадения среди некоторой области.



В начале по набору эмбеддингов для токенов применяем *conv* слой и получаем наборы совпадений с какими-то паттернами, затем навешиваем нелинейность и агрегируем совпадения по области, чтобы получить векторное представление текста фиксированной длины и обработать его FC-сеткой.

Сверточный слой: для изображений идея состоит в том, чтобы пройтись по нему скользящим окном и итеративно применять **сверточный фильтр** к каждому окну. На изображении можно видеть пример: внизу начальная картинка, а сверху – отфильтрованный выход сверточного слоя с размером свертки 3x3. И поскольку начальное изображение имеет двумерное пространство, то и фильтр, и выход свертки будут также двумерными.

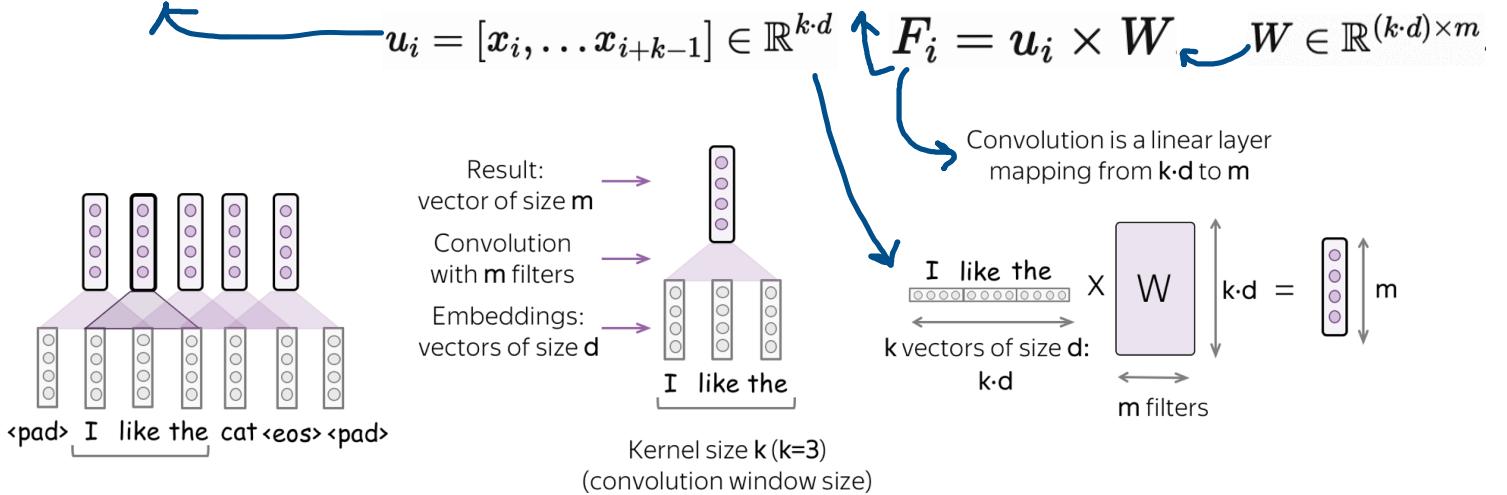
В случае с текстами мы имеем одномерное пространство, соответственно и фильтр, и выход слоя будут одномерными. В примере нарисовано применение свертки размером «3» к одномерному тексту.



Применение сверточного слоя для текста: свертка – это линейный слой (с последующей нелинейностью), применённый к каждому *input* окну. Предположим, что:

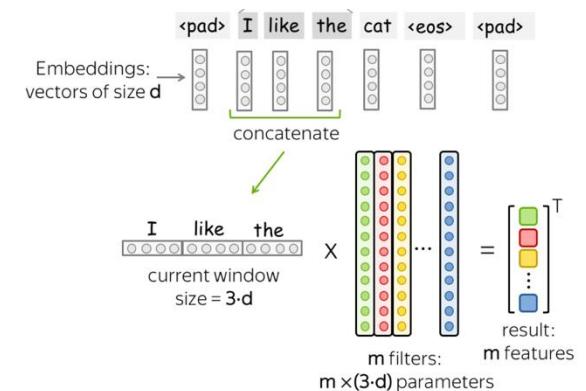
- x_1, \dots, x_n – представления входных слов
- d (*input channels*) – размерность входного эмбединга
- k (*kernel size*) – длина сверточного окна (для текста)
- m (*output channels*) – число сверточных фильтров

Тогда сверточный слой – это такое линейное преобразование, что для окна размером k мы соединяем все векторы окна в один u_i и умножаем его на матрицу фильтров W .



Таким образом мы проходим по всем *input* окнам и получаем столько выходных векторов размером m , сколько окон мы обработали.

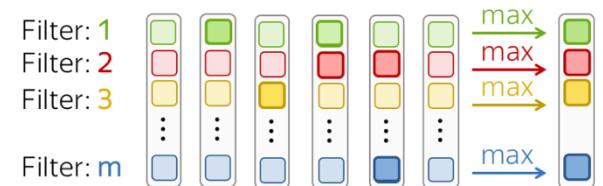
В сверточный фильтр заложена **идея**: один фильтр = один *feature extractor*, поэтому m фильтров = m *feature extractors*. Каждый фильтр – это один из столбцов в матрице W , каждый из которых при умножении на общий вектор окна фильтра дает **одно** значение в выходном векторе m . Соответственно, сколько столбцов в матрице W , столько же фильтров и, соответственно, столько же будет значений в выходном векторе.



Pooling слой: после того, как сверточный слой извлек m фичей из каждого окна и получил большой набор векторов размера m , слой пулинга нужен, чтобы агрегировать эти фичи по какому-то заданному размеру области. Данный слой используется, чтобы уменьшить размерность входного пространства, и соответственно уменьшить кол-во пар-в, используемых сетью.

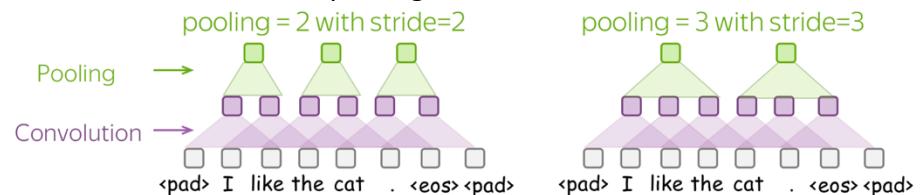
Основными видами пулинга являются: **max-pooling & mean-pooling**.

Макс-пулинг берет максимум по каждой оси пространства (то есть по каждой фиче), медианный-пулинг берет медиану по каждой фиче.

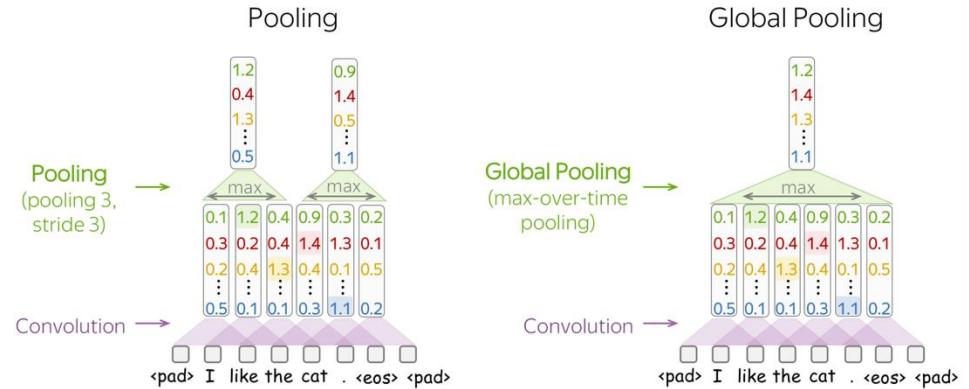


Pooling & Global pooling: также как и сверточный слой, слой пулинга применяется к некоторым окнам фиксированной длины. У этих обоих слоев есть параметр *stride*, контролирующий отступ одного окна от другого, также данный параметр позволяет контролировать степень пересечения окон (обычно используется пулинг так, чтобы окна не пересекались)

Разница между пулингом и глобальным пулингом состоит в том, что пулинг применяется независимо к каждому входному окну, в то время как глобальный пулинг применяется на весь *input*. Глобальный пулинг нужен для того, чтобы вне зависимости от длины текста, в определенный момент свести все к одному вектору, характеризующему весь текст. Также *global pooling* иногда называют как *max-over-time pooling*.



На данном изображении видно разницу между обычным пулингом с окном = 3 и stride = 3, выдающим на выход в данном случае 2 вектора (из 6) И глобальным пулингом, превращающим весь вход вне зависимости от его длины в один вектор на выходе.



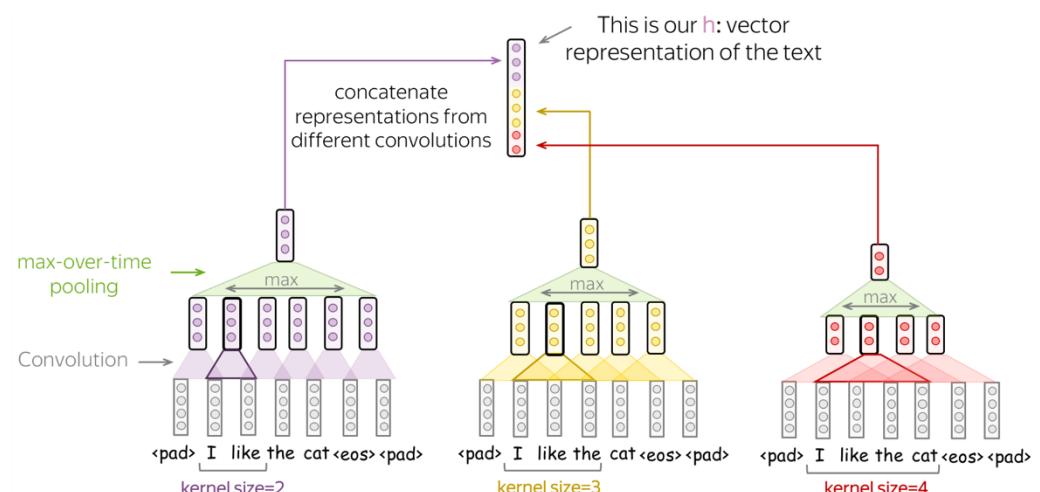
Типичная структура CNN-сети – разобрав каждый блок CNN-сети мы видим, что можно построить модель, которая в определённый момент выдаст нам вектор фиксированной длины как представление всего текста целиком. Превратить набор векторов фичей по каждому окну в единый вектор нам позволяет слой *max-over-time pooling* – это ключевой слой для CNN-сетей при работе с последовательностями.

Выше мы рассмотрели структуру сети только с одним сверточным слоем (в котором соотв все фильтры только с одним размером окна).

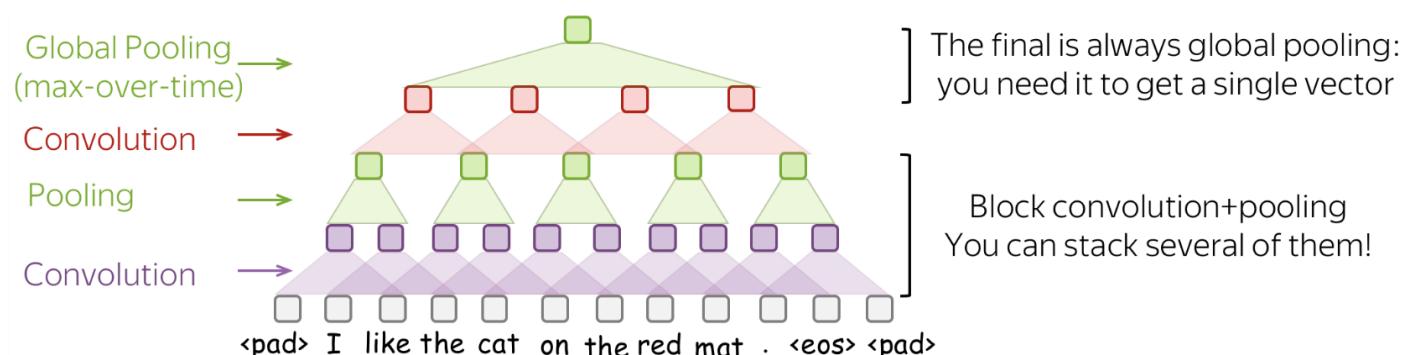
Как можно совместить несколько разных сверток для работы с последовательностью?

Разные свертки параллельно: если мы хотим применить к одной входной последовательности несколько разных наборов фильтров с разными размерами окон, то мы можем сделать это следующим образом.

Применим к одной посл-ти свертки с разными размерами окон, потом для каждой нелинейность, а затем глобал пулинг. Получим для каждой свертки свой набор фичей, а далее конкатенируем их.



Разные свертки последовательно: также мы можем захотеть использовать разные свертки друг за друг в погоне за попыткой уловить более верхнеуровневые паттерны. В таком варианте мы используем слои в такой же последовательности (свертка -> нелинейность -> пулинг -> повтор), а в конце берем глобал пулинг. В примере ниже изображен пример со сверткой из одного фильтра.



Multi-label классификация – такая формулировка задачи существенно отличается от single-label классификации.

Но в жизни встречается, не менее часто (когда у объекта может быть несколько правильных ответов).

Для такого типа задачи в сравнении с single-label подходом нам нужно поменять две вещи:

- модель и то, как она вычисляет вероятности классов

- лосс-функцию

Model: Вместо Softmax – Element-wise Sigmoid – после последнего FC-слоя мы имеем K значений,

соответствующих уверенностям модели в K классах, далее эти уверенности (логиты) мы превращаем в вероятности.

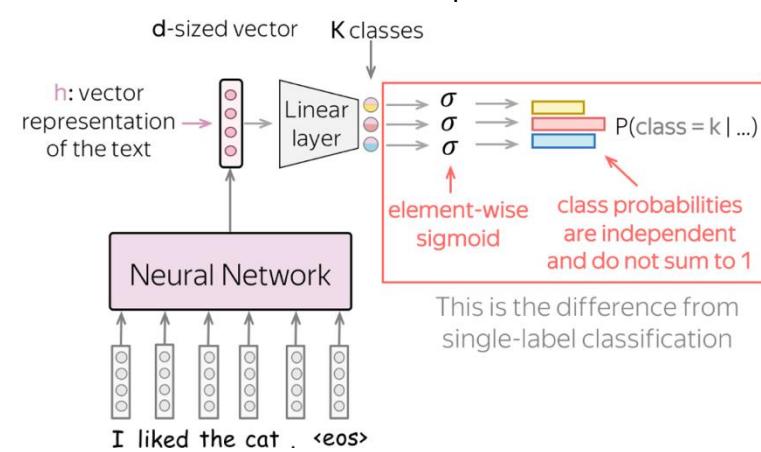
Для single-label задачи мы использовали softmax, который превращал логиты в суммирующуюся в 1 распределение вероятности. Это значит, что K классов делят между собой одну вероятностную массу (если у одного класса высокая вероятность, у других классов она должна быть низкой). Для нашей задачи такое не подходит, так как хотим иметь возможность, чтобы у нескольких классов была высокая вероятность.

Для multi-label задачи мы каждое из K значений логитов

будем превращать в вероятность независимо от других логитов, применяя к нему сигмоиду.

←
Это можно интерпретировать, как K независимых классификаторов, где каждый из которых решает, отнести ли объект к K-му классу.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Loss function: Binary Cross Entropy для каждого класса – раз уж мы стали рассматривать задачу как использование «K независимых классификаторов», то и лосс-функция для каждого класса должна быть независимой и учитывающей вероятность (*Binary Cross Entropy aka LogLoss*), а общим лоссом будет сумма, взятая со знаком «-», всех LogLoss'ов для каждого класса.

Model prediction:

$$P(y_i=1 | \dots \text{<eos>}), i = 1..K$$

Target:

$$p_i^*, i = 1..K$$



Binary cross-entropy loss for each class:

$$-\sum_{i=1}^K [p_i^* \cdot \log P(y_i = 1|x) + (1 - p_i^*) \cdot \log P(y_i = 0|x)] \rightarrow \min$$

where $P(y_i = 0|x) = 1 - P(y_i = 1|x)$

binary classifier loss for class i

Как работать с Word Embeddings? – на вход любой текстовой модели мы должны подать текст в виде каких-то эмбедингов для его слов (токенов).

У нас есть 3 варианты, чтобы получить их:

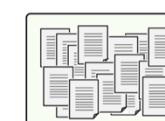
- обучать эмбединги с нуля как часть модели (текста может не хватить для того, чтобы достаточно обучить все представления, то есть некоторые слова могут не уловить всех связей из-за недостатка данных или из-за недостатка информации в тексте, если текст однообразен)
- взять предобученные (W2V, GloVe, FastText,...) (если их недообучать, то могут не уловить специфику конкретной нашей задачи)
- инициализировать их предобученными и далее дообучать вместе с моделью (*fine-tuning*) (отличный вариант, так как эмбединги уже много знают о мире, и к тому же мы их подкорректируем для конкретной задачи)
 - Train from scratch
 - Take pretrained (Word2Vec, GloVe)
 - Initialize with pretrained, then fine-tune

What they will know:



May be not enough to learn relationships between words

What they will know:



Know relationships between words, but are not specific to the task

What they will know:



Know relationships between words and adapted for the task

“Transfer” knowledge from a huge unlabeled corpus to your task-specific model

Аугментации данных – подход для «бесплатного» увеличения количества данных путем некоторого видоизменения начальных данных. Такой подход помогает увеличить **количество данных** (для DL-моделей это очень важно), а также **разнообразие данных** (увеличивая разнообразие, мы делаем нашу модель более устойчивой к реальным разнообразным данным, а значит более регуляризованной). Для изображений все достаточно понятно: берем картинки и крутим их в логически допустимых пределах, а также меняем цвета, яркость, контрастность, добавляем шум, растягиваем в разные стороны и т.д. **Как быть с текстовыми данными?**

Word dropout является наиболее простым и популярным решением для регуляризации, в котором для каждого набора данных каждое слово с фиксированной вероятностью (допустим, 10%) либо заменяется на токен типа «Unknown», либо на рандомный токен из словаря.

The movie about cats was absolutely great, and the cats were cute.

replace with UNK

pick several words randomly

replace with random words

The movie UNK cats was absolutely
UNK, and the UNK were cute.

The movie mejorate cats was absolutely
fellows, and the mak were cute.

Идея тут проста: мы хотим научить модель воспринимать текст целиком, воспринимать широкий контекст, а не только улавливать связь между соседними словами.

(аналогом для изображений является замена, некоторых рандомных зон пикселей либо на черный цвет, либо на рандомный цвет – для того, чтобы научить модель воспринимать картинку целиком)

External resources (e.g., thesaurus) – более сложное решение, в котором слова или фразы заменяются синонимами. Для таких фокусов нам нужны внешние источники этих самых синонимов, которых мало для языков != английскому. Также это может сыграть злую шутку при работе с языками с богатой морфологией, из-за которой замена синонимом просто нарушит грамматическую согласованность.

Different models – еще более мудреный метод состоит в том, чтобы «перефразировать» текст, используя какие-то внешние модели. Одним из популярных методом для перефразирования является перевод текста на какой-нибудь язык, а затем назад.

Языковые модели (Language models)

Когда мы создаем модель какого-либо процесса, мы хотим, чтобы данная модель хорошо предсказывала наиболее вероятные события в соответствии с выбранным процессом.

Языковая модель – не исключение, здесь надо только уточнить, что *событием* в данном случае будет являться какой-то лингвистический элемент (текст, предложение, слово, символ). Таким образом цель LM – уметь корректно предсказывать вероятности таких событий.

Как предсказать вероятность текста? – у нас нет и не будет текста со всеми возможными предложениями в мире, по которому можно было бы посчитать вероятность текста целиком. Поэтому разложим вероятности текстов/предложений на вероятности более мелких токенов (слов/символов).

Для формальной постановки задачи: пусть

y_1, \dots, y_n – токены предложения;

$P(y_1, \dots, y_n)$ – вер-ть увидеть эти токены в соотв. последовательности.

Тогда получается:

$$P(y_1, y_2, \dots, y_n) = P(y_1) \cdot P(y_2|y_1) \cdot P(y_3|y_1, y_2) \cdots \cdots P(y_n|y_1, \dots, y_{n-1}) = \prod_{t=1}^n P(y_t|y_{<t})$$

Таким образом мы разложили вероятность какого текста/предложения на произведение вероятностей более мелких токенов.

Подходы LM – первоначально рассмотрим *left-to-right* идею языковой модели. Метода тут 2: основанный на N-граммах и нейросетевой. И разница в них только в том, как именно они вычисляют вероятность $P(y_t|y_1, \dots, y_{t-1})$

Need to define:

- how to compute
 $P(y_t|y_1, y_2, \dots, y_{t-1})$

N-gram models

Neural models

$$P(y_t|y_1, \dots, y_{t-1})$$

Непосредственное применение LM состоит в том, чтобы «генерировать» каждый новый токен последовательно. Изначально проанализировав контекст, сэмлировать (брать в соответствии с набором вероятностей) новый токен из распределения вероятностей новых токенов. Далее повторить уже с обновленным контекстом (включающим новый токен).

Замечание 1: далее будет показано, что использовать жадный алгоритм, выбирая на каждом шагу *наиболее вероятный токен* далеко не всегда является хорошей идеей и может давать плохие результаты.

Замечание 2: помимо *left-to-right* идеи генерации новых токенов существуют и другие – более умные подходы, о них мы упомянем далее в курсе.

N-gram Language Models – более нативный подход для *left-to-right* идеи, в котором условные вероятности слов считаются на основе общих статистик корпуса текстов (схоже с подсчетными методами для эмбедингов). Для построения N-gram LMs мы должны обсудить две ключевые компоненты для такого рода языковых моделей.

Markov property (предположение о независимости) – для прямого вычисления нам требуется кол-во раз, когда некая большая посл-ть встречалась в корпусе текстов. Выше было сказано, что кол-во зачастую будет = 0 \Rightarrow такое решение $P(y_t|y_1, \dots, y_{t-1}) = \frac{N(y_1, \dots, y_{t-1}, y_t)}{N(y_1, \dots, y_{t-1})}$ нам не подходит.

Поэтому используется Марковское предположение:

«вероятность слова зависит только от фиксированного

количество слов перед ним». Таким образом, модель N-gram предполагает, что:

Можно также выписать предположение для:

- модели 3-грам ($n = 3$)
- модели 2-грам ($n = 2$)
- модели 1-грам ($n = 1$)

Используя данное предположение (допустим для 3-грам) можно на примере с текстом показать, как изменится вероятностное представление текста: «*I saw a cat on a mat*»

$$P(y_t|y_1, \dots, y_{t-1}) = P(y_t|y_{t-n+1}, \dots, y_{t-1})$$

$n=3$ (trigram model): $P(y_t|y_1, \dots, y_{t-1}) = P(y_t|y_{t-2}, y_{t-1})$

$n=2$ (bigram model): $P(y_t|y_1, \dots, y_{t-1}) = P(y_t|y_{t-1})$,

$n=1$ (unigram model): $P(y_t|y_1, \dots, y_{t-1}) = P(y_t)$.

$$P(I \text{ saw a cat on a mat}) =$$

- $P(I)$
- $\cdot P(\text{saw} | I)$
- $\cdot P(a | I \text{ saw})$
- $\cdot P(\text{cat} | I \text{ saw a})$
- $\cdot P(\text{on} | I \text{ saw a cat})$
- $\cdot P(a | I \text{ saw a cat on})$
- $\cdot P(\text{mat} | I \text{ saw a cat on a})$

$$P(I \text{ saw a cat on a mat}) =$$

- $P(I)$
- $\cdot P(\text{saw} | I)$
- $\cdot P(a | I \text{ saw})$
- $\cdot P(\text{cat} | I \text{ saw a})$
- $\cdot P(\text{on} | I \text{ saw a cat})$
- $\cdot P(a | I \text{ saw a cat on})$
- $\cdot P(\text{mat} | I \text{ saw a cat on a})$

ignore use



Smoothing (перераспределение вероятностной массы) – допустим мы работаем с моделью 4-грамм и рассматриваем следующий пример:

Что если числитель или знаменатель в этой формуле будет = 0?

$$P(\text{mat} \mid \text{I saw a cat on a}) = P(\text{mat} \mid \text{cat on a}) = \frac{N(\text{cat on a mat})}{N(\text{cat on a})}$$

Эти оба случая плохи для нас. Для того, чтобы этого избежать принято использовать **smoothings (сглаживания)**. Данный трюк позволяет «украсть» вероятностной массы с уже увиденных событий в пользу еще неувиденных. Перед нами стоят две разные задачи: избавить от нулей в числителе и в знаменателе.

Нули в знаменателе: если фраза «*cat on a*» никогда не встречалась в нашем тексте, то нас в знаменателе ожидает 0. Что с этим делать?

$$N(\text{cat on a}) = 0 \longrightarrow \text{try "on a"}$$

Backoff (aka Stupid Backoff) – идея в том, чтобы брать меньше контекста для неизвестного нам контекста. Идея достаточно проста и глупа, но работает хорошо. Также существует более умные подходы, например: **Kneser-Ney Smoothing** (но его мы разбирать тут не будем)

$$P(\text{mat} \mid \text{cat on a}) \approx P(\text{mat} \mid \text{on a})$$

$$N(\text{on a}) = 0 \longrightarrow \text{try "a"}$$

$$P(\text{mat} \mid \text{on a}) \approx P(\text{mat} \mid \text{a})$$

Linear interpolation – более интересное решение состоит в том, чтобы смиксовать все вероятности: 1-граммы, 2-граммы, 3-граммы, и т.д.

То есть нам будут нужны суммирующиеся в 1 коэффициенты $\lambda_1, \dots, \lambda_{(n-1)}$.

И тогда мы сможем переписать нашу вероятность, как:



Данный коэффициенты подбираются как гиперпараметры (например, на кросс-валидации)

$$\hat{P}(\text{mat} \mid \text{cat on a}) \approx \lambda_3 P(\text{mat} \mid \text{cat on a}) + \lambda_2 P(\text{mat} \mid \text{on a}) + \lambda_1 P(\text{mat} \mid \text{a}) + \lambda_0 P(\text{mat})$$

Нули в числителе: если фраза «*cat on a mat*» никогда не встречалась, числитель = 0 \Rightarrow все дроби = 0. Но это совсем не значит, что рассматриваемая фраза невозможна! (значит, нужно как-то поправить)

Самым простым способом решить проблемы является

Laplace smoothing (aka add-one smoothing). Его суть в том, чтобы «притвориться», что мы видели все N-граммы хотя бы 1 раз (либо вместо 1 можно добавить любую δ)

$$\hat{P}(\text{mat} \mid \text{cat on a}) = \frac{\delta + N(\text{cat on a mat})}{\delta \cdot |V| + N(\text{cat on a})}$$

Генерация текста в модели N-gram – идея генерации текста в данной модели полностью совпадает с общей идеей: имея в наличии какой-то контекст, мы будем выдавать распределение вероятностей следующего токена (среди всех токенов словаря). Далее мы будем сэмплировать новый токен из этого распределения, расширять им наш контекст, и повторять все процедуру заново.

Если посмотреть на примеры, сгенерированный моделями N-грамм, то можно заметить, что сгенерированный текст получать очень несвязным, модель совсем не чувствует хоть небольшой контекст, а только связывает слова, основываясь на самых близких связях. **Отсутствие чувства контекста** – основной минус данной модели.

Замечание 1: если вместо сэмплирования жадно брать наивероятнейший новый токен, то получиться, что сгенерированные тексты получаются очень короткими (_end_of_str – популярный токен) и очень однообразными.

NN Models – при использовании нейросетей будем также рассматривать *left-to-right* подход LM. И основной задачей модели все также является предсказание вероятности события при учете контекста. $P(y_t \mid y_{<t})$

Будем тренировать модели, чтобы они умели предсказывать эту вероятность.

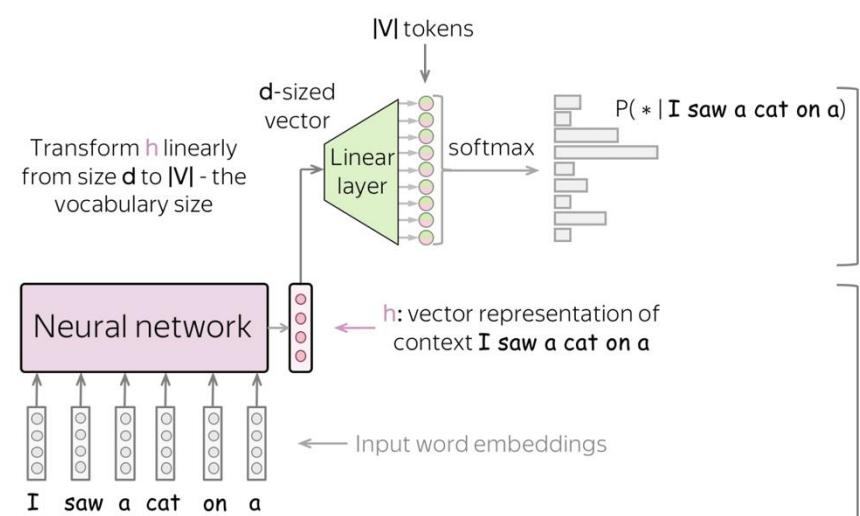
По сути, модель должна уметь делать 2 вещи:

- **model-specific:** «читать» предыдущий контекст и на его основе составлять векторное представление прочитанного контекста. Используя это представление модель сможет предсказывать распределение вероятности следующего токена. Структура модели для этой задачи может быть разной (и RNN, и CNN, и вообще то, что мы захотим)

- **model-agnostic:** генерировать распределение вероятностей следующего токена

Структура нейросетевой LM – то есть, по сути, это задача классификации, где на каждом шагу мы должны научиться предсказывать следующий токен. И поэтому пайпайн построения модели очень схож с разделом *Классификации Текстов*. А именно:

- «читать» моделью предыдущий контекст
- получить векторное представление контекста
- по нему предсказать вероятностное распределение нового токена



Обучение NN-LM – так как мы рассматриваем задачу классификации, то обучать будем также на *Cross-Entropy*.

Формально, если y_1, \dots, y_n – последовательность тренировочных токенов и в определенный момент времени t модель выдает прогноз

$$p^{(t)} = p(* | y_1, \dots, y_{t-1})$$

И таргетом на этом моменте времени является

$$p^* = \text{one-hot}(y_t)$$

То есть модель хочет предсказать 1 истинному токену y_t , и 0 всем остальным токенам.

Стандартно тут используется *Cross-Entropy Loss*:

$$\text{Loss}(p^*, p) = -p^* \log(p) = -\sum_{i=1}^{|V|} p_i^* \log(p_i)$$

С другой стороны, на формулу лосса можно взглянуть по-

другому: если p & p^* – два распределения, то эта формула представляет из себя эквивалент дивергенции Кульбака-Лейблера.

$$D_{KL}(p^* || p)$$

Поэтому стандартную NN-LM оптимизацию можно еще интерпретировать, как максимальное «сближение» предсказанного моделью распределения и эмпирического распределения таргета.

Generation and Sampling Strategies – мы можем колдовать различными способами над вероятностными распределениями, которые нам выдает модель, чтобы сгенерировать текст с определенными особенностями. И когда специфичные особенности текста чаще зависят непосредственно от поставленной задачи, существуют два показателя сгенерированного текста, за которыми нам следует следить в любом случае.

- **coherent**: сгенерированный текст должен иметь смысл (для N-грамм модели когерентность была очень плоха)
- **diverse**: модель должна уметь генерировать различный текст.

Standard Sampling – самый классический путь: сэмплирование нового токена из предсказанного моделью вероятностного распределения безо всяких изменений (на всем количестве токенов).

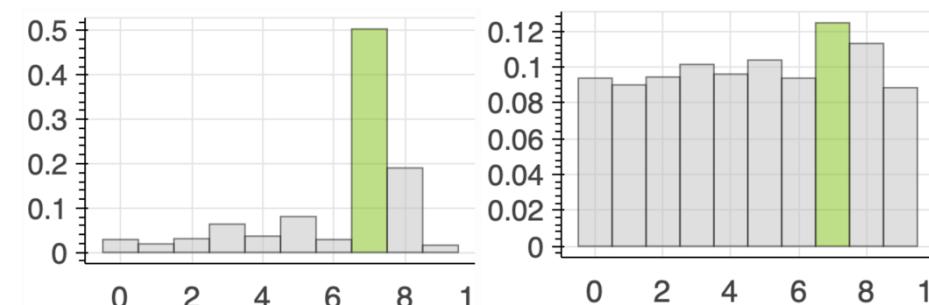
Sampling with temperature – чтобы легче всего изменить поведение генерации текста LM, чаще всего используется метод **изменения softmax-температуры**. То есть прежде, чем применять softmax к финальным логитам, все логиты делятся на некоторую «температуру» τ .

$$\tau - \text{softmax temperature}$$

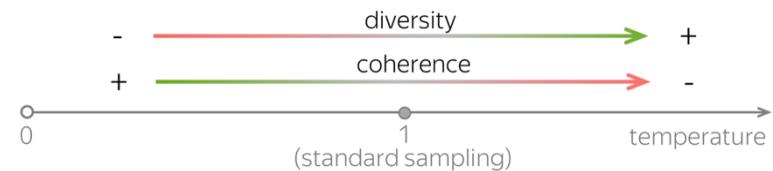
$$\frac{\exp(h^T w)}{\sum_{w_i \in V} \exp(h^T w_i)} \rightarrow \frac{\exp\left(\frac{h^T w}{\tau}\right)}{\sum_{w_i \in V} \exp\left(\frac{h^T w_i}{\tau}\right)}$$

Какие изменения дает такое изменение температуры?

На графиках можно увидеть распределение для одних и тех же логитов, но с разной температурой softmax (слева $\tau = 0.5$, справа $\tau = 5$). Таким образом высокая температура повышает *diverse*, поскольку она выравнивает распределение вероятностей, и, соответственно, увеличивает случайность и вариабельность сэмплов. Но в то же время, такая высокая вариабельность делает сгенерированный текст абсолютно бессмысленным (что подтверждается примерами).



С другой стороны, если мы берем очень низкую температуру, то мы теряем любую вариабельность, и алгоритм просто превращается в жадный (т.к. одна вероятность всегда на порядок выше других) и текст получается хоть и осмысленным, но всегда очень похожим (из одних фрагментов фраз).



Top-K sampling – данная простая эвристика заключается в том, что мы будем сэмплировать только **по первым K наиболее вероятных токенам**. Тем самым мы оставляем модели возможность выбора, но делаем так, чтобы максимально неподходящие слова не были сэмплированы.

Замечание 1: фиксированное

значение K – не самая лучшая

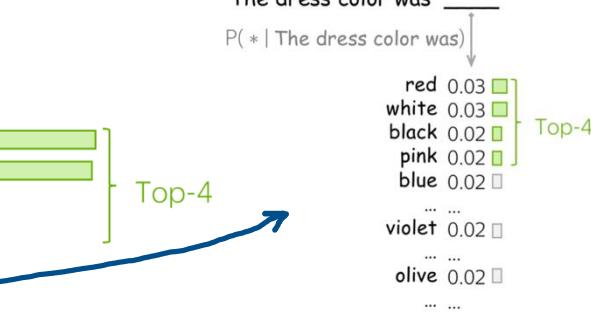
идея. Это объясняется тем, что:

- либо содержат нежелательные токены (в более острых расп-х)
- либо покрывают слишком мало вероятностной массы (в более плоских расп-х)

The light was _____

$P(* | \text{The light was})$ get probability distribution

on	0.45
off	0.44
in	0.01
at	0.01
too	0.01
...	...



Top-P (aka Nucleus) sampling – некоторая модификация предыдущей эвристики, в которой мы теперь будем сэмплировать **по токенам, попавшим в P% вероятностной массы**.

The light was _____

$P(* | \text{The light was})$ get probability distribution

on	0.45
off	0.44
in	0.01
at	0.01
too	0.01
...	...

Top-80%

The dress color was _____

$P(* | \text{The dress color was})$

red	0.03
white	0.03
black	0.02
pink	0.02
blue	0.02
...	...
violet	0.02
...	...
olive	0.02
...	...

языковых

Оценка (уже обученных) Language Models – при создании

моделей мы закладывали в них ключевую идею: модель должна хорошо предсказывать наиболее вероятные события. Другими словами, хорошая модель какого-либо процесса должна хорошо согласоваться с реальным процессом. Так же и **хорошая языковая модель должна хорошо согласоваться с реальным текстом** – это является основной идеей для оценки языковых моделей.

⇒ то есть мы должны понять, ожидает ли модель увидеть текст, соответствующий реальному.

Допустим, у нас есть текст $y_{(1:M)} = (y_1, \dots, y_M)$.

Тогда вероятность того, насколько «модель ожидает увидеть

данного текста» можно вычислить по **лог-правдоподобию**:

(то есть cross-entropy * -1).

Наряду с лог-правдоподобием текст можно оценить такой величиной как **перплексия**.

Таким образом хорошая модель будет иметь высокое лог-правдоподобие и низкую перплексию.

$$L(y_{1:M}) = L(y_1, y_2, \dots, y_M) = \sum_{t=1}^M \log_2 p(y_t | y_{<t})$$

Log-likelihood of the text

$$\text{Perplexity}(y_{1:M}) = 2^{-\frac{1}{M} L(y_{1:M})}$$

Замечание 1: для того, чтобы лучше почувствовать величину **перплексии** рассмотрим крайние значения, которые она может принимать. Наиболее высокое значение = 1 (т.к. если наша модель дает вероятности = 1 истинным токенам на каждом шаге, то лог-правдоподобие = 0 ⇒ перплексия = 1). Наиболее низкое значение перплексия будет принимать, если модель совершенно не понимает текст и думает, что все токены имеют одинаковую вероятность $1/|V|$, где V – словарь.

$$\text{Perplexity}(y_{1:M}) = 2^{-\frac{1}{M} L(y_{1:M})} = 2^{-\frac{1}{M} \sum_{t=1}^M \log_2 p(y_t | y_{1:t-1})} = 2^{-\frac{1}{M} \cdot M \cdot \log_2 \frac{1}{|V|}} = 2^{\log_2 |V|} = |V|$$

В таком случае перплексия ↗

Sequence to sequence (Seq2Seq) and Attention

Для обзора задачи seq2seq отлично подойдет задача машинного перевода, в которой по входной последовательности (на одном языке), нужно выдать новую последовательность (уже на другом языке), причем чаще всего эти последовательности имеют разную длину на разных языках.



Основы Seq2Seq – формально задачу машинного перевода можно описать так: у нас есть входная последовательность x_1, \dots, x_m и выходная последовательность y_1, \dots, y_n (вероятно, с разными длинами). Перевод можно рассматривать как учитывающее входную последовательность нахождение наиболее вероятной выходной последовательности.

Формально, это можно записать так:

$$y^* = \arg \max_y p(y|x)$$

Где y – желаемая выходная последовательность, а x – входная последовательность. То есть мы максимизируем условную вероятность получить последовательность « y » при условии последовательность « x ». При постановке задачи для обучения машины мы должны обучить некоторую функцию, где θ – набор параметров, а далее найти argmax от этой функции.

Human Translation

$$y^* = \arg \max_y p(y|x)$$

The “probability” is intuitive and is given by a human translator’s expertise

Machine Translation

model → parameters

$$y' = \arg \max_y p(y|x, \theta)$$

Questions we need to answer

- **modeling**

How does the model for $p(y|x, \theta)$ look like?

- **learning**

How to find θ ?

- **search**

How to find the argmax?

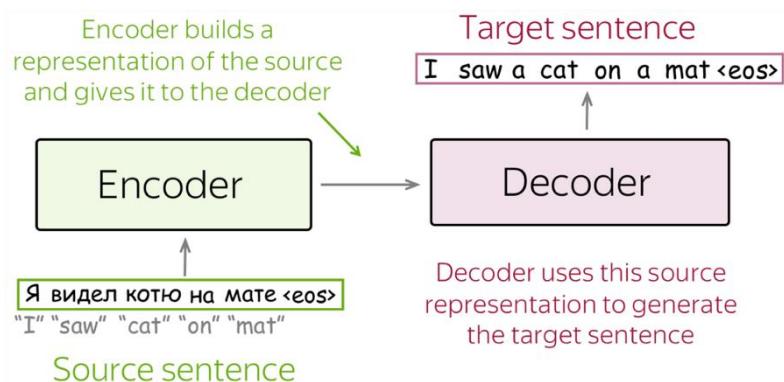
Чтобы построить такого рода модель нужно суметь сделать 3 вещи:

- **modelling**: научиться строить модель для $p(y|x, \theta)$
- **learning**: научиться оптимизировать параметры θ
- **inference**: научиться находиться наилучший « y » из возможных (наилучшую выходную последовательность из возможных)

Парадигма Encoder-Decoder

– самая стандартная парадигма для построения seq2seq моделей, состоящая из 2-х основных компонент:

- **encoder**: «читает» входную последовательность и производит ее векторное представление
- **decoder**: использует это представление, чтобы «раскодировать» его и вывести выходную последовательность



Conditional Language Models (CLM) – в прошлом разделе мы работали с языковыми моделями (LM), предсказывающими вероятность $p(y)$ набора токенов $y = (y_1, \dots, y_n)$ (то есть мы смотрим только на выходную последовательность и никаку более, даже когда мы генерируем текст и учитываем предыдущий контекст, а он является частью выходной последовательности, мы все равно смотрим только на элементы одной последовательности). Поэтому в LM используется **безусловная вероятность**.

Теперь же мы хотим решать задачи, в которых требуется учитывать входную последовательность, а значит, что для seq2seq моделей мы будем вычислять **условную вероятность**. CLM-модели функционируют схожим образом с LM-моделями, но они дополнительно учитывают входную последовательность.

$$\text{Language Models: } P(y_1, y_2, \dots, y_n) = \prod_{t=1}^n p(y_t | y_{<t})$$

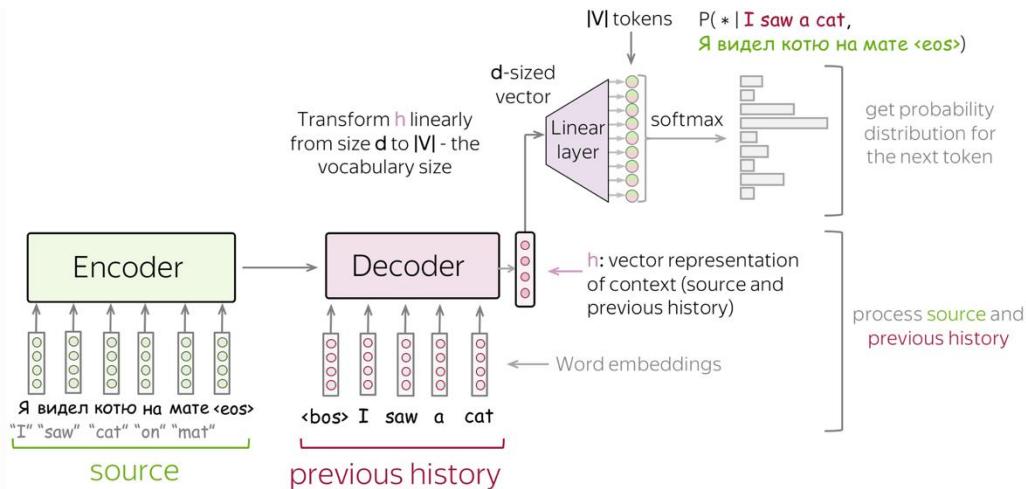
$$\text{Conditional Language Models: } P(y_1, y_2, \dots, y_n, |x) = \prod_{t=1}^n p(y_t | y_{<t}, x)$$

condition on source x

Поскольку разница LM & CLM только в « x », то верхнеуровневый пайплайн обучения и применения CLM-моделей будет очень похож на LM-модели.

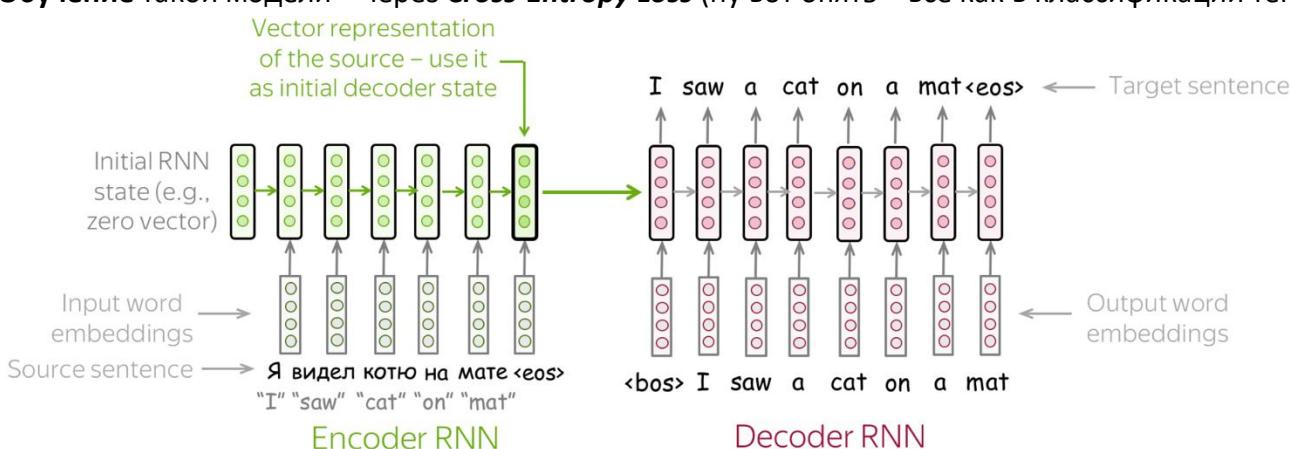
А именно:

- «читать» моделью источник (входную последовательность) и все предыдущие уже сгенерированные элементы выхода
- получить векторное представление контекста (и источника, и предыдущих выходов модели)
- пользуясь этим представлением, предсказать новый токен для выхода



The Simplest Model (две RNNs для Encoder и Decoder) – самая простая структура *encoder-decoder* модели состоит из 2-х RNNs (LSTMs), одна из которых является энкодером, другая – декодером. Энкодер «читает» источник (входную последовательность) и его финальное скрытое состояние используется как векторное представление источника. И тут мы надеемся, что этот эмбединг вместит в себя весь смысл источника (наивно, правда?), а декодер успешно раскодирует один лишь вектор в новую последовательность (и тут наивно :D). Такую модель можно модифицировать известными нам методами (например, добавить по несколько слоев)

Обучение такой модели – через **Cross-Entropy Loss** (ну вот опять – все как в классификации текстов и LM)



Inference (Greedy Decoding & Beam Search) – следующим вопросом, встающим на нашем пути, является:

Как мы будем выбирать новый токен из вероятностного распределения?

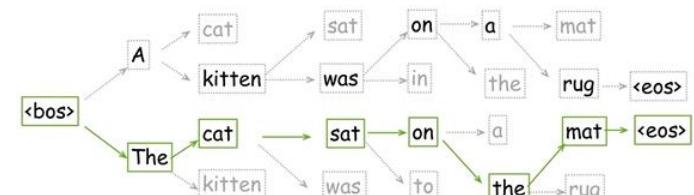
Или же, переформулировав: как искать argmax?

$$y' = \arg \max_y p(y|x) = \arg \max_y \prod_{t=1}^n p(y_t|y_{<t}, x)$$

Тут есть два подхода (тема схожа с соответствующей в LM):

Greedy decoding (жадно) – на каждом шагу выбирать новый токен, как наиболее вероятный из распределения. Это может быть неплохим бейзлайном, но решение не самое лучшее (наилучший токен сейчас \neq наилучшее предложение в будущем).

Beam search (следить за разными наиболее вероятными последовательностями) – подход, при котором мы будем трекать несколько «вариантов перевода», и на каждом шагу оставлять лишь топ-N из них. На каждом шаге инференса параллельно запускаем каждый из предложенных ранее вариантов и будем смотреть на кумулятивное произведение вероятностей токенов каждой последовательности (некое правдоподобие) и по этим коэффициентам оставляем топ лучших. Это была жадная разновидность бимсерча. Сюда же можно прикрутить и топ-K и топ-P% и сэмплировать из данного пула. Обычно beam-size берут в районе 4-10.

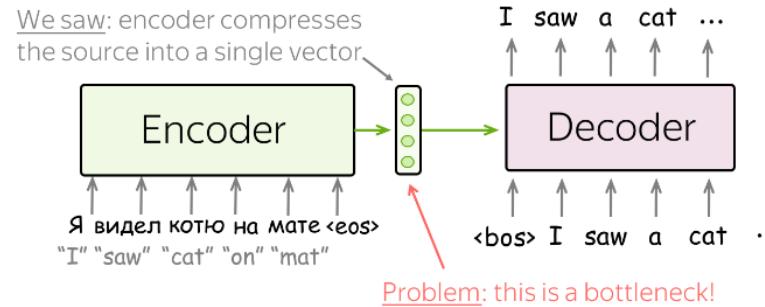


Attention (механизм внимания) – прежде чем разбираться, как мы будем решать проблему, поймем, с какой именно проблемой в классическом RNN исполнении парадигмы *encoder-decoder* мы сталкиваемся.

Мы ее сформулировали чуть выше:

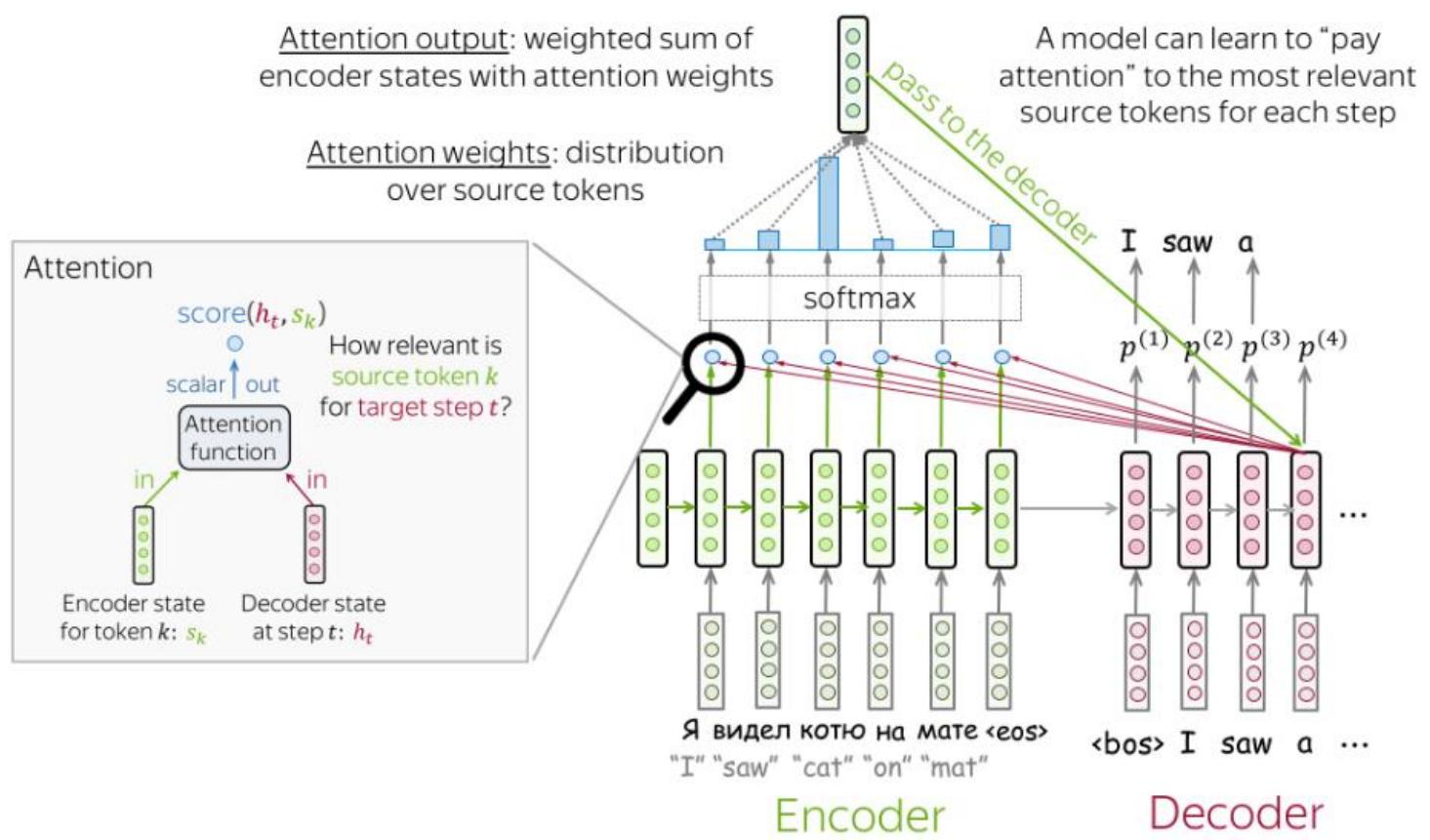
- для энкодера проблема в том, что он должен сжать всю информацию об источнике в один вектор
- для декодера проблема в том, что корректно «распаковать» один вектор в разумную выходную последовательность (**но на различных шагах некоторая информация из энкодера может нам быть более полезной, чем другая**)

Именно эту вторую проблему, позволяет решить *механизм внимания*. Он не спасает нас от того, что энкодер должен уместить всю информацию в один вектор, но зато он помогает декодеру корректно распаковывать этот вектор, зная на каких словах источника лучше сфокусироваться на определенных шагах декодировки (теперь мы будем ориентироваться больше не один вектор энкодера, а на сумму всех состояний энкодера, взвешенную по релевантности каждого в конкретный момент)



Attention Structure

Снизу приведена картинка для объяснения работы механизма внимания.



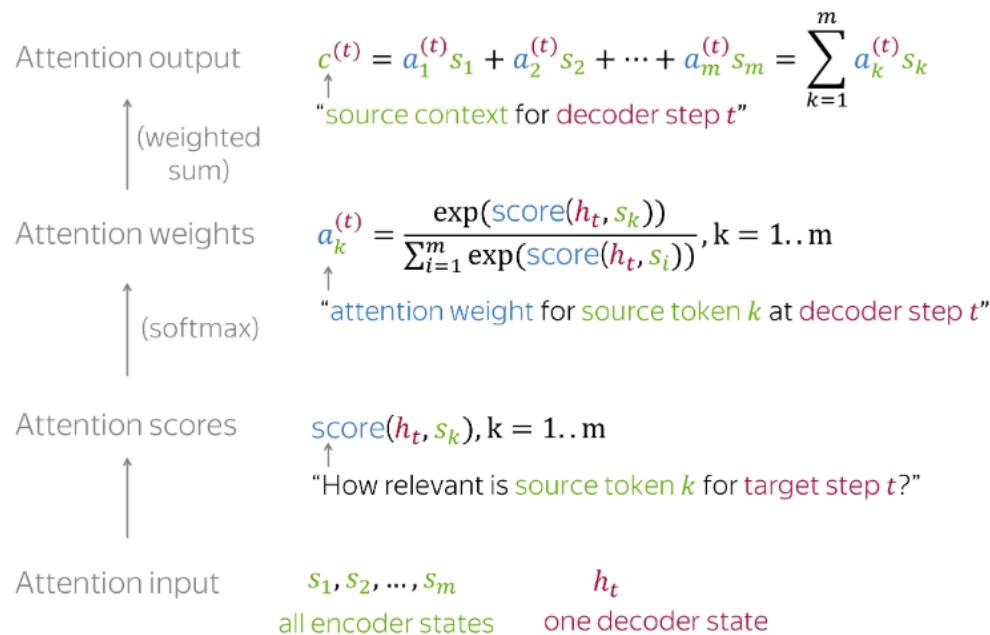
На t -ом шаге энкодера механизм *attention*:

- получает предыдущее состояние декодера $h(t-1)$ (для предсказания токена t) и все состояния энкодера (s_1, \dots, s_m)
 - для каждого состояния энкодера s_k *attention* вычисляет $\text{score} = \text{«релевантность»}$ состояния s_k для раскодирования t -го токена выходной последовательности при помощи $h(t-1)$.
- Формально, *attention* вычисляет нек. функцию $\text{score}(h(t-1), s_k)$ для всех s_k из входной последовательности.
- вычисляет *attention weights*: вероятностное распределение важностей каждого из состояний энкодера ($\text{score} + \text{softmax}$)
 - выдает вектор релев. контекста = взвешенной сумме состояний энкодера с коэффициентами *attention weights*.

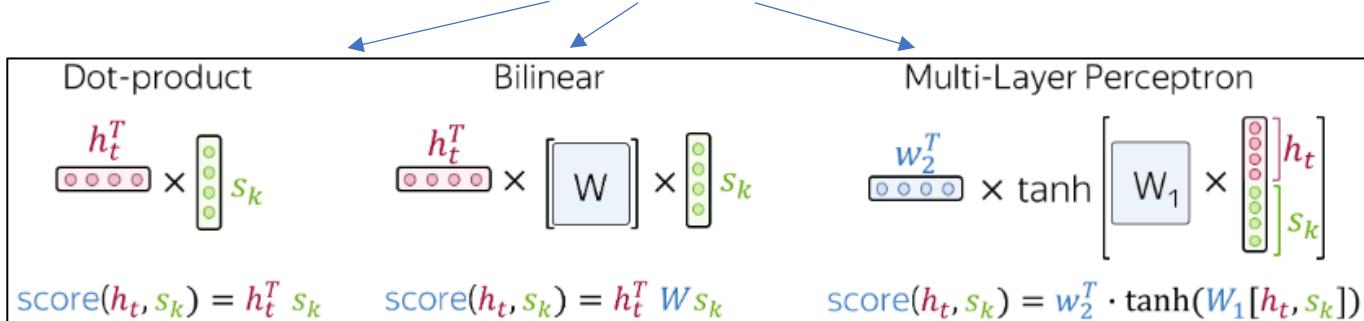
Более детально работа механизма внимания для декодирования t-го токена разобрана на следующей картинке:

Видно, что на выходе *attention* выдает вектор актуального контекста, который складывается из векторов всех состояний энкодера (s_i) с соответствующими релевантностями (a_i)

Основная фишка – такую модель можно учить *end-to-end*, т.к. все компоненты модели дифференцируемы, то есть модель сама научиться брать нужную ее информацию на каждом шаге.

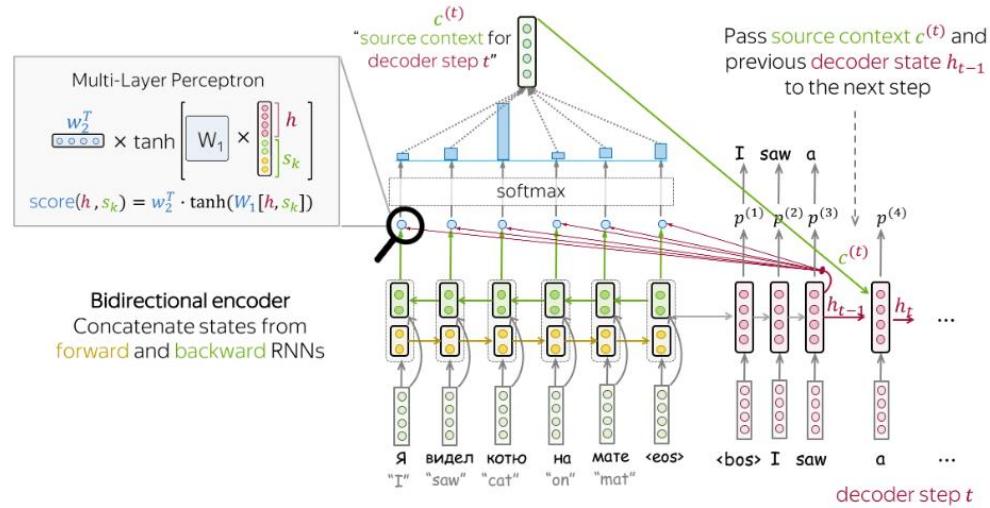


Как вычислять Attention Score? – в обзоре механизма внимания на картинках выше мы не уточнили, как именно вычисляется на шаге t «релевантность» состояния энкодера s_k для декодирования предыдущего скрытого состояния декодера $h(t-1)$. Здесь можно использовать сколь угодно замысловатую функцию, но обычно обходятся одним из нескольких простых вариантов: скалярное произведение (классика), bilinear function (aka Luong attention), multi-layer perceptron (MLP) (aka Bahdanau attention)



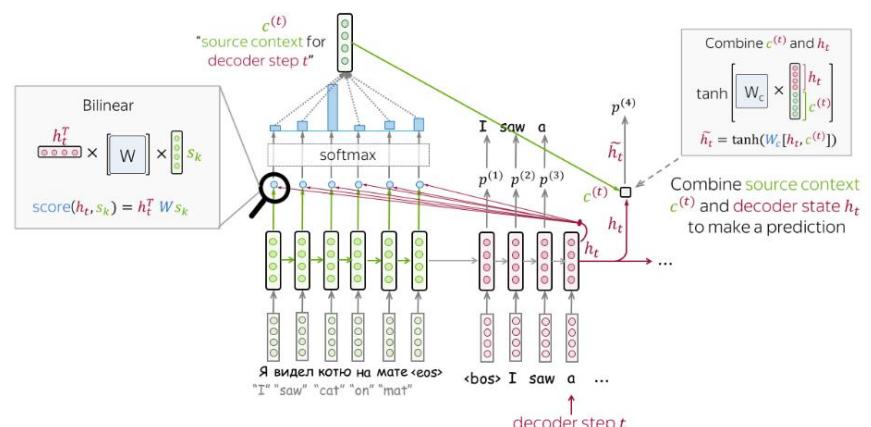
Bahdanau Attention Model:

- *encoder*: двусторонняя RNN. Финальное состояние каждого токена = конкатенация forward и backward сетей
- *attention score*: многослойная полносвязная сеть
- *использование attention*: между шагами декодера – механизм внимания использует состояние $h(t-1)$ для нахождения актуального контекста $c(t)$ на шаге t , далее и $c(t)$, и $h(t-1)$ передаются в сеть декодера, а далее по выходу $h(t)$ делается прогноз.



Luong Attention Model:

- *encoder*: односторонняя RNN
- *attention score*: bilinear function
- *использование attention*: между выходом декодера $h(t)$ и предсказанием на шаге t . То есть *attention* использует $h(t)$, чтобы получить $c(t)$, а далее мы объединяем $h(t)$ и $c(t)$, и на основе h _крышка делаем прогноз для t .

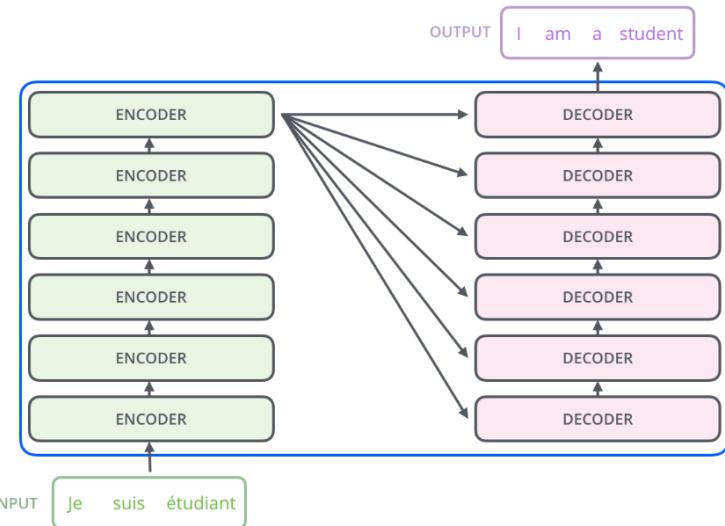


NMT Transformer: “Attention is all you need” – новая концепция, в которой предлагается отказаться от всего, кроме механизма внимания и наделить последнего новым крутым функционалом, при этом также руководствуясь парадигмой *encoder/decoder*. Сейчас, в 2023г, модели, построенные на базе трансформеров, являются флагманами среди ML/DL-моделей, которые также хорошо работают для задач LM. Трансформеры – новый подход к решению seq2seq. Как будет устроен наш новый black-box? Во-первых, заметим, что энкодер и декодер здесь будут работать по-разному, соответственно и разбирать мы их будем по отдельности.

	Seq2seq without attention	Seq2seq with attention	Transformer
processing within encoder	RNN/CNN	RNN/CNN	attention
processing within decoder	RNN/CNN	RNN/CNN	attention
decoder-encoder interaction	static fixed-sized vector	attention	attention

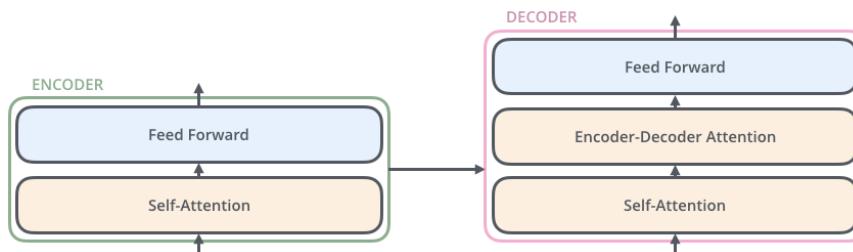
Encoder в трансформере будет на каждом из своих шагов (слоев энкодера) все больше и больше «разбираться» во входной последовательности (узнавать токены все лучше в контексте всего предложения), обновляя на каждом шаге каждое из представлений начальных токенов (путем слоя *self-attention*) в соответствии с контекстом входной последовательности. А далее пропускаться через FC-сетку для получения окончательных представлений энкодера для текущего шага.

На каждом слое энкодера новые представления токенов будут вычисляться одновременно (то есть токены взаимодействуют друг с другом параллельно), в отличие от RNN, в котором взаимодействие токенов происходит последовательно.



Decoder в трансформере будет работать похожим образом с *encoder*'ом, то есть также на каждом шаге все лучше «разбираясь» в выходных токенах (через *self-attention*), но после *self-attention* они также на каждом шаге они будут заглядывать в *encoder* благодаря *Encoder-Decoder Attention* (то есть будут смотреть на все представления токенов в финальном слое энкодера). После всех заглядываний они также будут пропускаться через FC-сеть, чтобы получить финальные представления декодера для текущего шага.

Encoder-Decoder Attention – это наш старый добрый *attention*, который мы использовали в RNN+Attention.



Слой Self-Attention: the “Look at Each Other” part – это одна из ключевых компонент структуры трансформера. Разница между *attention* и *self-attention* состоит в том, что второй механизм взаимодействует с представлениями той же природы, например: всеми представлениями токенов энкодера на определенном слое (а обычный *attention* используется в декодере и смотрит на представления энкодера)

Decoder-encoder attention is looking	Self-attention is looking
<ul style="list-style-type: none"> from: one current decoder state at: all encoder states 	<ul style="list-style-type: none"> from: each state from a set of states at: all other states in the same set

Основной идеей self-attention является взаимодействие токенов друг с другом для того, чтобы каждый «лучше понял себя» в контексте всей входной последовательности и обновил свое представление.

Как он работает внутри? Рассмотрим механизм *self-attention* по шагам (пример после описания):

Шаг 1 (расчет query, key, value): для каждого токена поступающего на вход слою *self-attention* вычисляем три вектора-помощника, которые будут нам в дальнейшем помогать: *Query*, *Key*, *Value*.

- *query*: вектор «запроса» – нужен для того, чтобы запрашивать информацию по другим токенам
- *key*: вектор «ключа» - нужен для того, чтобы давать некоторую информацию о токене по запросу (*query*)
- *value*: вектор «значения» - характеризует прибавку токена в контексте другого токена

Каждый из векторов вычисляется при произведении вектора токена на соответствующую обучаемую матрицу W_q , W_k , W_v . Размер каждого вектора обычно на порядок меньше, чем размер изначального эмбединга (эмбединг ~ 512 , каждый из векторов ~ 64)

Шаг 2 (расчет score): допустим мы хотим обновить представление для j-го токена во входной последовательности. Мы должны посчитать «важности» каждого i-го слова входной последовательности, для обновления j-го токена. «Важность» токена i-го для обновления j-го будем считать как скалярное произведение: ⇒ для j-го слова получим N «важностей» - насколько каждый токен входа нужен для корректного контекста j-го токена.

$$\text{Важность}(i \text{ для } j) = \langle \text{query}(j), \text{key}(i) \rangle$$

Шаг 3 (учет размерности и softmax): после того, как мы получили «важности» каждого входного токена для обновления j-го токена, нам стоит привести их в более универсальный и стабильный вид:

- разделить каждую «важность» на \sqrt{d} , где d – размерность векторов key и query (эвристика для стабильности)
- взять от этого всего softmax

После преобразования получаем уже окончательные коэффициенты важностей каждого входного токена для обновления j-го токена.

$$\text{Важность}(i \text{ для } j) = w_{i,j} = \frac{\exp(\langle \text{query}(j), \text{key}(i) \rangle / \sqrt{d})}{\sum_{p=1}^n \exp(\langle \text{query}(j), \text{key}(p) \rangle / \sqrt{d})}$$

Шаг 4 (получение нового представления): когда все нужные важности будут получены, остается только лишь обновить представление j-го токена на новое, учитывая весь требуемый нам контекст, умножив важность каждого токена на его value-вектор.

Таким образом для j-го токена получили новое векторное представление, учитывющее контекст входной последовательности.

$$Z_j = \sum_{p=1}^n w_{p,j} v_p$$

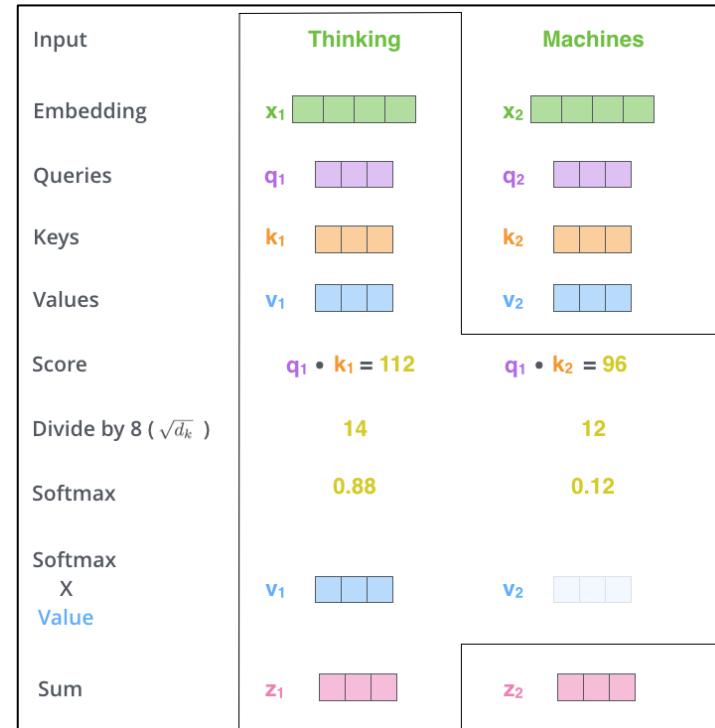
Набор новых представлений для всех входных токенов называется **Attention Head (результат слова)**

Замечание: мы рассмотрели схему применения слоя *self-attention* для пересчета представления j-го токена, на практике мы одновременно пересчитываем все векторы входной последовательности (именно этим параллелизмом и удобна такая архитектура). Для RNN один тренировочный шаг занимает $O(\text{len(source)} + \text{len(target)})$, в то время как для трансформера один шаг занимает $O(1)$.

Рассмотрим пример работы слоя *self-attention*:

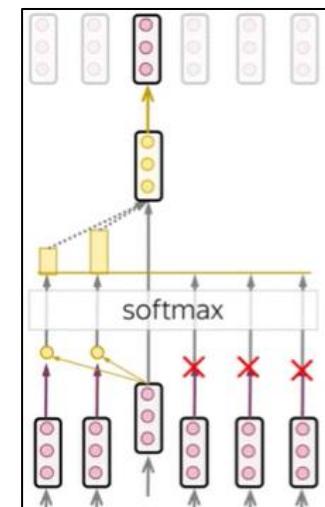
Входная последовательность состоит всего из двух слов: «Thinking Machines» и допустим мы хотим обновить. Что мы тут делаем?

- сперва вычисляем query, key, value векторы для обоих слов из входа
- далее считаем важность 1-го слова для обновления 1-го слова, и важность 2го слова для обновления 1-го
- делим на корень из размерности и обрамчиваем в softmax (получаем значения от 0 до 1, суммирующиеся в 1) – получаем финальные важности
- умножаем ранее полученный вектор key каждого слова на его финальную важность для обновления 1-го слова.
- суммируем эти произведения и получим новое векторное представление для 1-го слова
- на выходе получаем z1,z2 – являющиеся *attention-head* для нашей входной последовательности



Masked Self-Attention: “Don’t Look Ahead” for the decoder – некоторая модификация слоя *self-attention*, применяемая внутри декодера. Такое изменение структуры слоя требуется из-за того, что в декодере мы генерируем каждый новый символ последовательно, не зная будущих токенов (в отличие от энкодера, в котором мы изначально знаем всю входящую последовательность и поэтому при *self-attention*’е мы разрешаем смотреть на все токены).

Именно для этого используется данная модификация, в которой при использовании слоя *self-attention* мы запрещаем вычислять важности для впереди стоящих токенов. И получается, что в декодере вычислять новое представление для j-го токена мы будем, учитывая важности и key-векторы только позади стоящих токенов. Здесь также используется хэширование предыдущих состояний токенов (см. *past_key_values* в разделе Transformer Remarks)



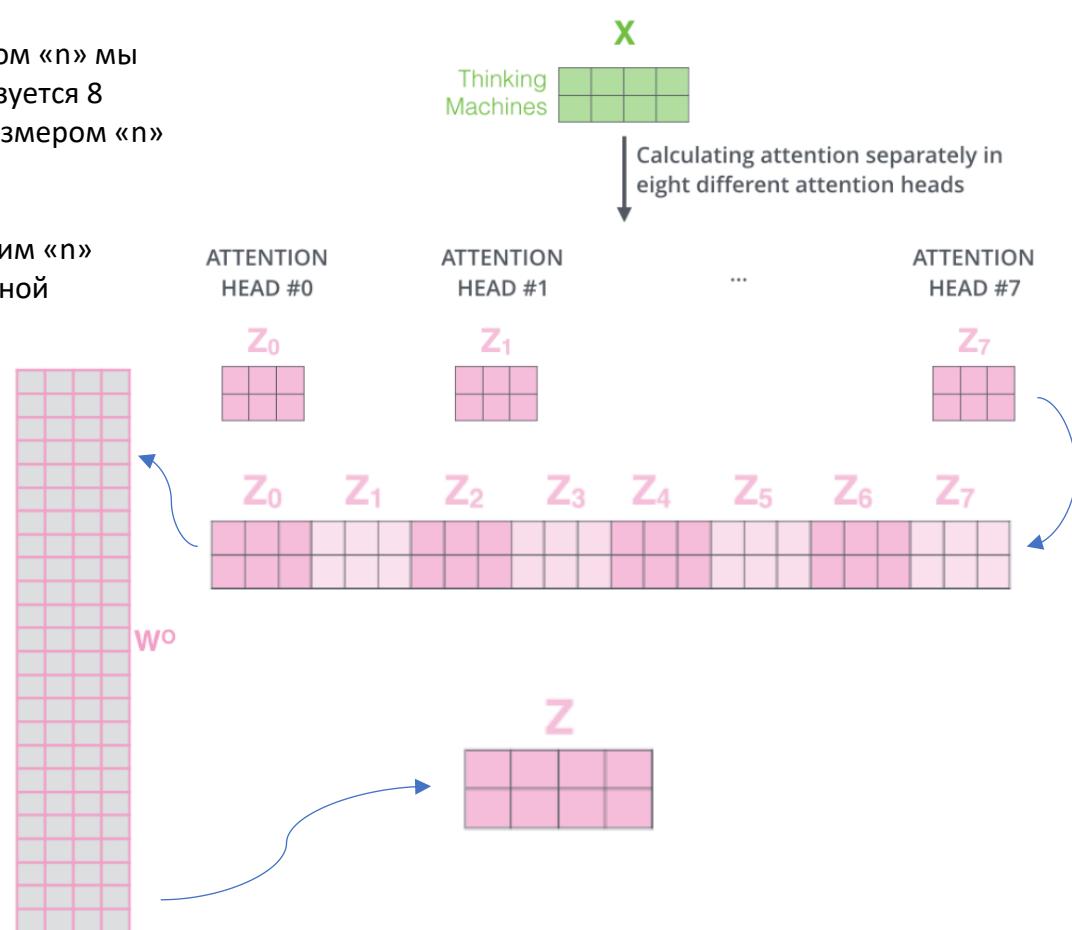
Multi-Head Attention: Independently Focus on Different Things – еще одна модификация слоя *self-attention* в классическом DL-стиле: если что-то хорошо работает, то возьмем такого побольше. С другой стороны на эту модификацию можно посмотреть так: каждое слово в предложении может быть связано с контекстом очень многими связями (по части речи, по роду/полу, по падежу, по локальному смыслу и т.д.) \Rightarrow надо дать модели возможность одновременно фокусироваться на разных вещах \Rightarrow будем вычислять не одну *attention-head*, а сразу несколько, после – объединять во что-то единое.

Также такая модификация дает модели больше свободы, так для каждой *attention-head* используется свой набор матриц W_q , W_k , W_v для вычисления *Query*, *Key*, *Value*. Каждая из таких матриц инициализируется рандомно, поэтому связи, которые мы подцепим каждым набором матриц могут получиться разными.

Таким образом для входа размером « n » мы получим (в Трансформере используется 8 голов) 8 разных *attention-heads* размером « n » каждая.

Далее сконкатенируем их и получим « n » векторов, но уже в 8 раз увеличенной размерности.

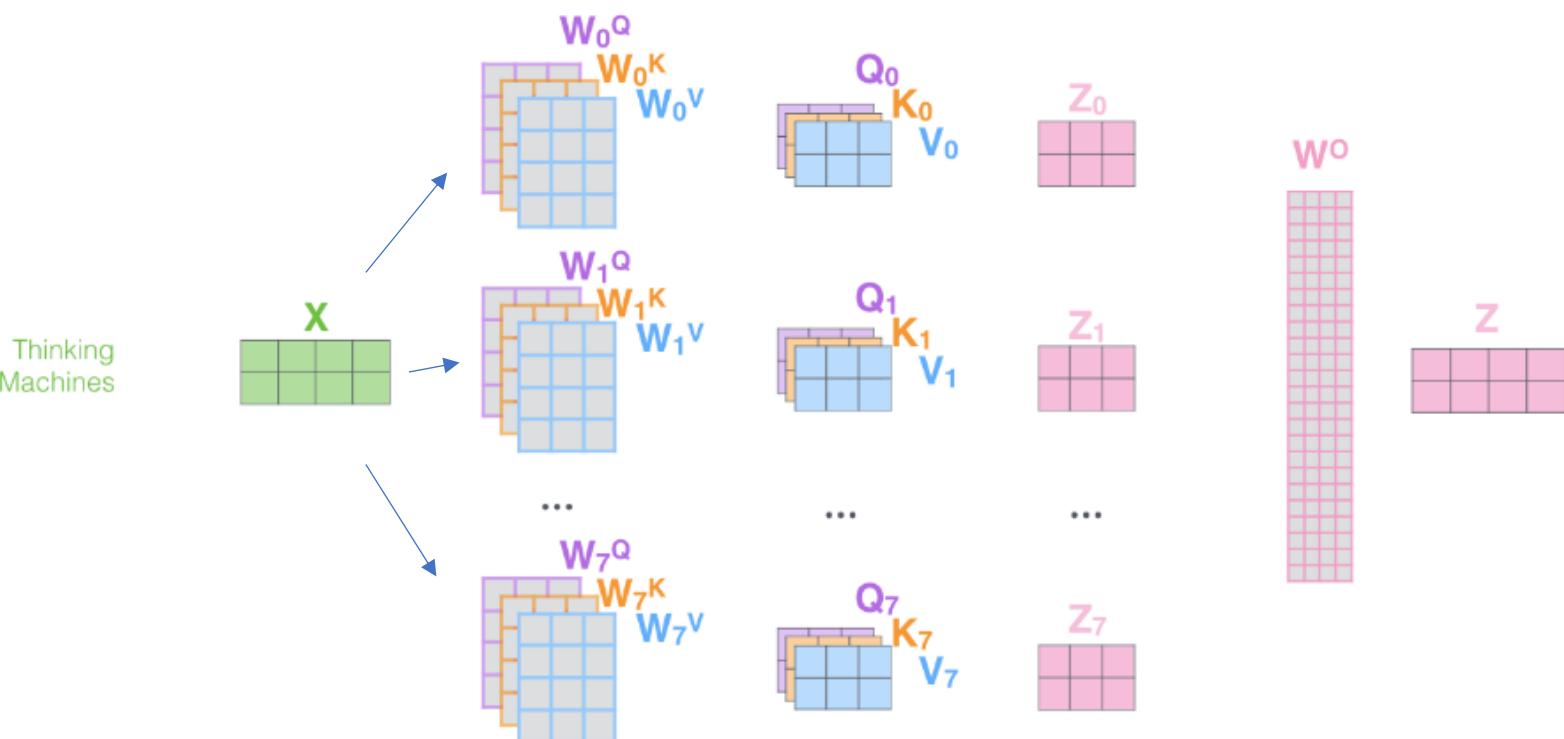
Умножим матрицу скокатенированных векторов на обучаемую матрицу W_o , чтобы уменьшить размерность и в результате получим Z – выход слова *multi-headed self-attention*



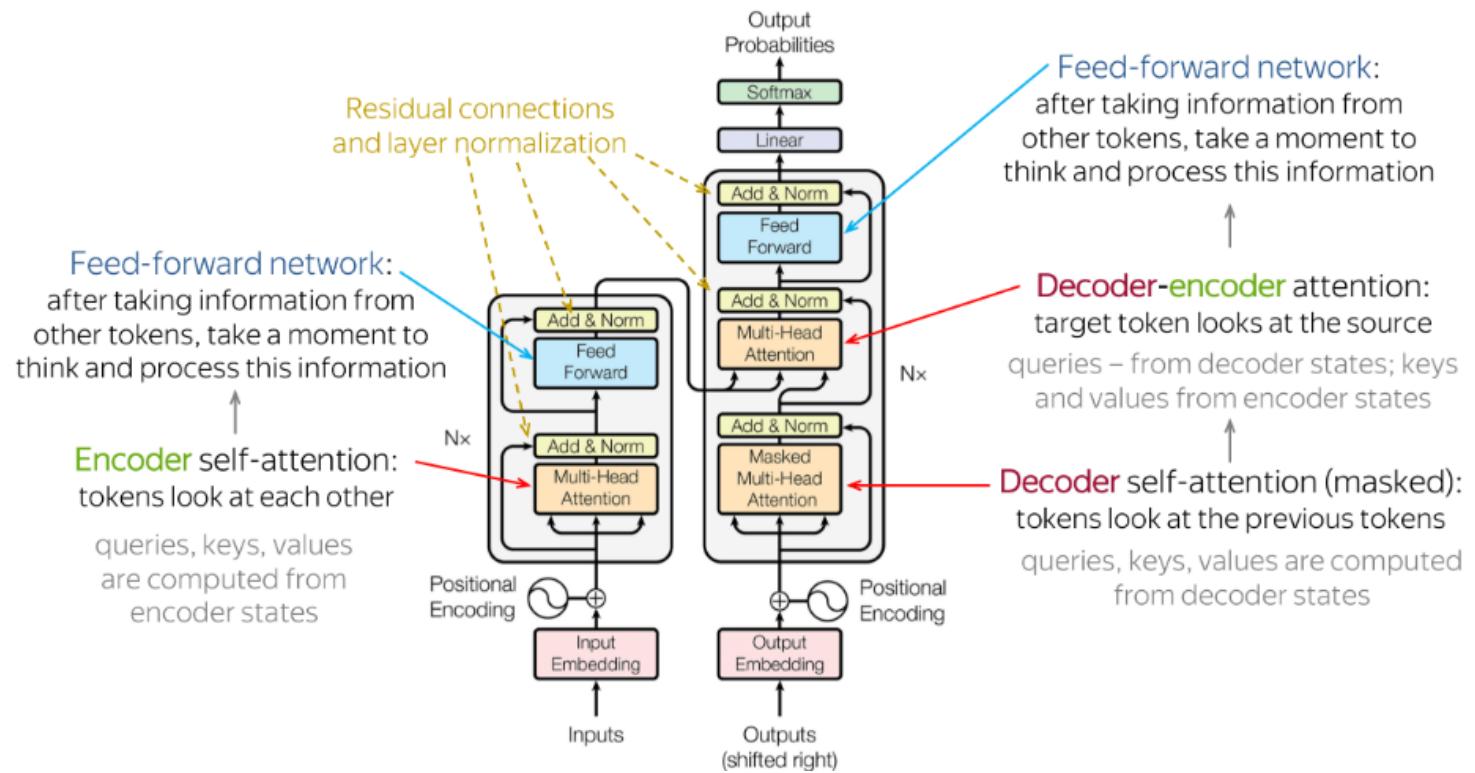
Финальный пример работы данной модификации:

X – последовательность входных представлений в слой *multi-headed self-attention*

Z – выходная последовательность слова



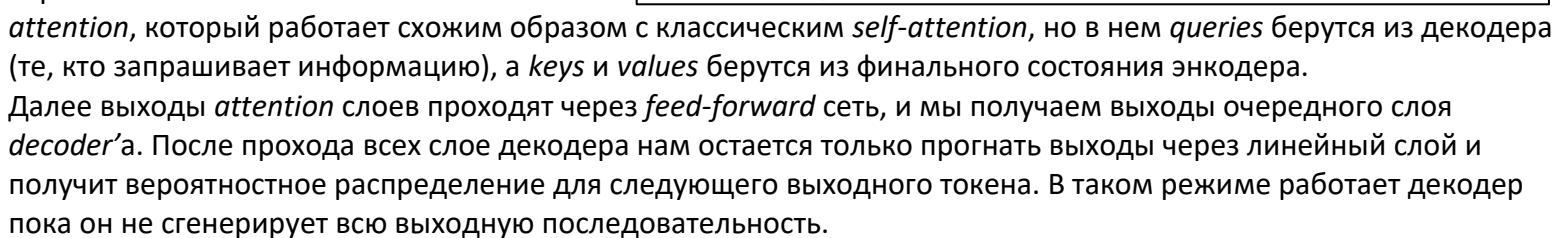
Transformer Model Structure – понимая как устроены основные компоненты трансформера (*multi-headed self-attention(encoder)* & *masked multi-headed self-attention(decoder)*) далее можем взглянуть целостную архитектуру модели.



Интуитивно видно, что модель работает точно в соответствии с тем, что мы обсудили выше.

В *encoder*'е токены взаимодействуют только друг с другом и на каждом слое все больше обновляют свои представления после прохода через *feed-forward* сеть.

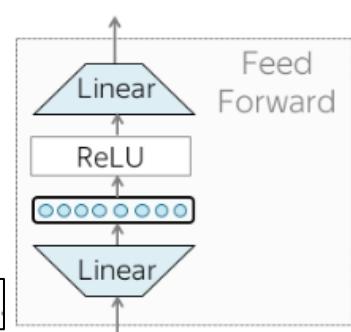
В *decoder*'е токены сначала взаимодействуют с предыдущими сгенерированными токенами выходной последовательности через *masked multi-headed self-attention*, а затем со всеми финальными представлениями токенов источника (из последнего слоя *encoder*'а) через *multi-headed encoder-decoder attention*



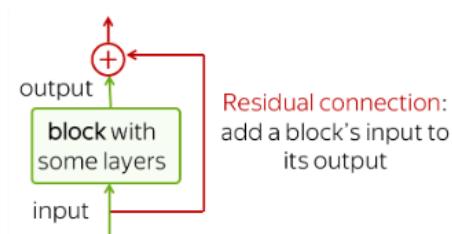
Далее разберем некоторые нюансы архитектуры модели, которые можно заметить на картинках.

Feed Forward Blocks: в каждом слое энкодера и декодера помимо различных вариаций механизмов внимания присутствуют *feed-forward* блоки. Чаще всего они из себя представляют 2-х слойную полносвязную сеть (лин.слой + нелинейность + лин.слой). После механизма внимания, на котором токены «смотрят друг на друга и лучше понимают себя» модель использует данные блоки, чтобы обработать эту новую информацию и выдать новые окончательные представления токенов. FFN – “Let me think and process this information”

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



Residual Connections: все то же самое, что мы встречаем в ResNet'е при работе со сверточными сетями. Идея заключается в пробросе предыдущих состояний сетки вперед, тем самым очень упрощая процесс потока обратного распространения ошибки, позволяя градиенту не затухнуть, даже в глубоких сетях.



В трансформерах *residual connections* используются после каждого *attention* и *FFN* слоев. На схеме трансформера видно, что каждый такой проброс ведет нас в слой *Add & Norm*, в котором вектора сначала конкатенируются, а затем нормализуются (см. след блок)

Layer Normalization (LayerNorm): тут речь про “*Norm*” часть слоя *Add & Norm*, в которой происходит независимая нормализация каждого вектора в батче (для того, чтобы стабилизировать процесс обучения).

В трансформере нам нужно нормализовать представления каждого токена. Также тут присутствуют обучаемые параметры *scale & bias* (общие для всего слоя), которые используются после нормализации для изменения масштаба выходов.

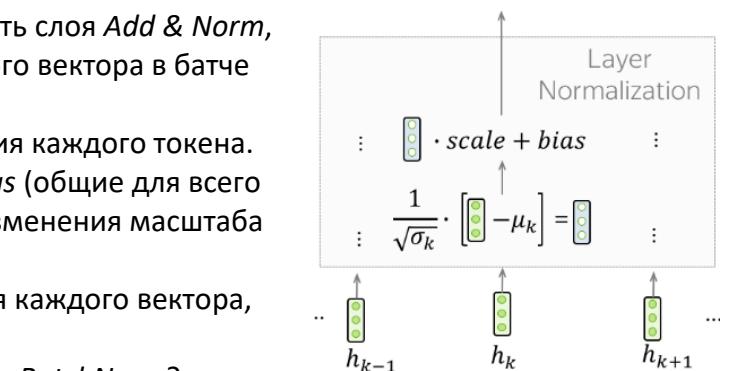
Замечание 1: $\mu(k)$ & $\sigma(k)$ вычисляются отдельно для каждого вектора, *scale & bias* – обучаемые параметры для слоя.

Замечание 2: почему используем именно *LayerNorm*, а не *BatchNorm*?

Ответ – из-за идеи, по которой мы хотим нормализовать каждую **фичу** по-отдельности, а это именно *LayerNorm*, а не *BatchNorm*. Вспомним разницу между ними:

BatchNorm – способ нормализации, при котором сначала вычисляется *среднее и дисперсия для всех элементов батча по одной фиче (каждая фича независимо от другой)*. А далее отнормализованные данные по каждой фиче приводятся к обучаемым *scale & bias* (*для всех фичей в BatchNorm-слое одни обучаемые параметры*).

LayerNorm – способ нормализации, при котором *среднее и дисперсия вычисляются независимо по каждому элементу батча (по всем фичам элемента)*. А далее отнормализованные данные по каждому объекту приводятся к обучаемым *scale & bias* (*для всех объектов в LayerNorm-слое одни обучаемые пары*)
 \Rightarrow видно, что оба слоя нормализуют, а потом приводят распределение к обучаемым параметрам, но различие у них в том, что нормализация и приведение к новым параметрам происходит в разных пространствах.
LayerNorm чаще используется в NLP-задачах, поскольку длина последовательностей зачастую варьируется и использование *BatchNorm* может привести к нестабильности (*В CV-задачах большее предпочтение наоборот отдают BatchNorm'у*)



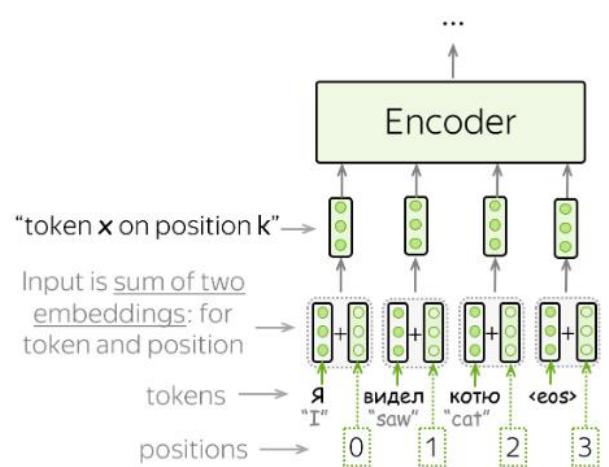
Positional Encoding: заметим, что изначальной архитектуре (т.к. мы не используем RNN или CNN) трансформер не может ничего знать о взаимном расположении токенов между собой \Rightarrow мы должны модели помочь в этом и как-то подсказывать их позиции.

Для этого на вход в модель будем подавать два набора эмбедингов: для токенов (классические эмбединги для слов) и для их позиций (наша новая эвристика). Тогда каждый входной токен будет равняться **сумме 2-х эмбедингов: токена и позиции**.

Позиционные эмбединги можно также обучать, но эмпирически было установлено, что фиксированные позиционные эмбединги показывают себя на практике не хуже.

Вычисляются они по следующим формулам:

Где pos – позиция токена, i – размерность векторного представления (каждая фича в позиционном эмбединге относится к синусоиде, а длины волн формируются в геом.прогрессии от



$$PE_{pos,2i} = \sin(pos/10000^{2i/d_{model}}),$$

$$PE_{pos,2i+1} = \cos(pos/10000^{2i/d_{model}}),$$

²*Пи до 10000*2*Пи

Transformer Remarks

Безусловно, трансформеры – это прорыв в моделях для работы с последовательностями. Но и тут есть тонкости, на которые стоит обращать внимание и, по возможности, пытаться улучшать.

Квадратичная сложность механизма attention

Если вспомнить общую формулу для расчета attention (q, k, v – матрицы с размерностями (длина последовательности) * (размер скрытого представления), то можно заметить, что произведение матриц в числителе дроби имеет квадратичную сложность (с точки зрения подаваемой последовательности), а также дальнейшее произведение на матрицу « v »

⇒ не сможем работать с длинными последовательностями (у BERT'a ограничение около 2тыс. токенов)

$$\text{Attention}(q, k, v) = \underbrace{\text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)}_{\text{from } q \text{ to } k} v$$

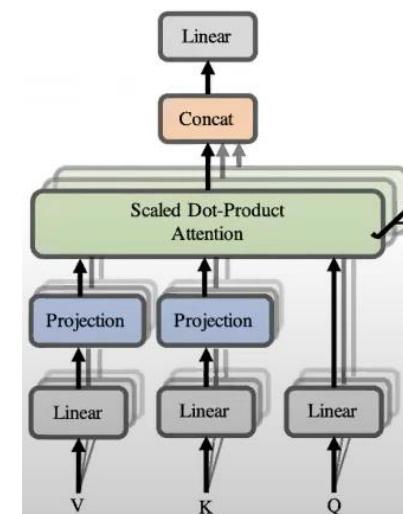
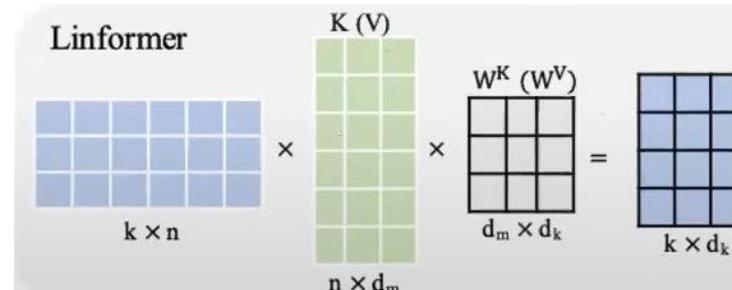
Attention weights
vector dimensionality of K, V

Linformer – идея при создании такой модификации трансформера состояла в том, чтобы уменьшить сложность вычисления двух вышеописанных последовательных произведений матриц путем проекций.

То есть у нас есть матрицы: $Q(n, k)$, $K(n, k)$, $V(n, k)$, где n – длина последовательности, k – размер скрытого представления. Поэтому при умножении $Q^*Kt = \text{матрица}(n, n)$ (сложность квадратичная).

Хотим теперь сделать проекцию матриц K и V так, чтобы аппроксимировать их матрицами $K'(\text{const}, k)$, $V'(\text{const}, k)$, тем самым сделав сложность такого умножения линейной.

Точность в таком подходе оказалось существенно хуже, но идея хорошо, а значит – можно дальше развивать.



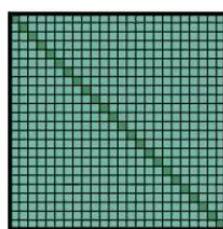
Longformer – еще раз напомним, что квадратичная сложность из-за того, что мы хотим получить матрицу «сходства» = матрицу attention (n, n). А точно ли она нам вся нужна? Точно ли слова отстоящие друг от друга очень далеко действительно важны друг другу?

⇒ вместо обычного *attention* будет *sliding window attention*, в котором будем считать «важности» только для близких слов (рисунок «b»). В таком случае мы можем подавать на вход любую последовательность, поскольку сложность ее обработки будет линейной.

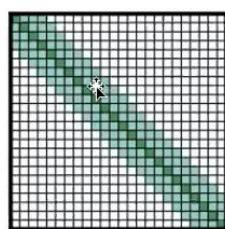
⇒ следующая небольшая модификация состоит в том, чтобы смотреть не прям на соседние, а в *dilation* варианте (шаг через 1) (рисунок «c»)

⇒ *global+sliding window* модификация обобщает подход со скользящим окном для того, чтобы у нас в последовательности были токены, «знающие» про вся последовательность. Это нужно, когда мы хотим извлекать информацию из всего текста (например, в классификации)

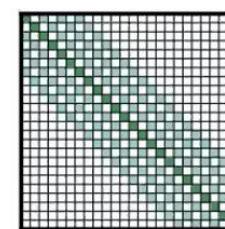
На тесте Longformer показал качество также хуже, чем обычный трансформер, но его большой плюс в том, что он умеет работать с большими текстами, которые нельзя засунуть в обычный трансформер.



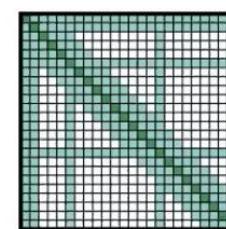
(a) Full n^2 attention



(b) Sliding window attention



(c) Dilated sliding window



(d) Global+sliding window

Performer – продвинутый аналог *Linformer'a*, в котором размерность для проекции не обучаются как гиперпараметры, а для ее нахождения используются случайные признаки Фурье. Качество очень возрастает по сравнению с *Linformer'ом* (релиза пока что нет)

F-NET (Fourier-Net) – добавили attention и все стало очень хорошо работать.

А точно ли из-за того, что сам механизм внимания настолько крут?

Или же тут дело в том, что мы разрешили токенам смотреть друг на друга и «расширять кругозор»?

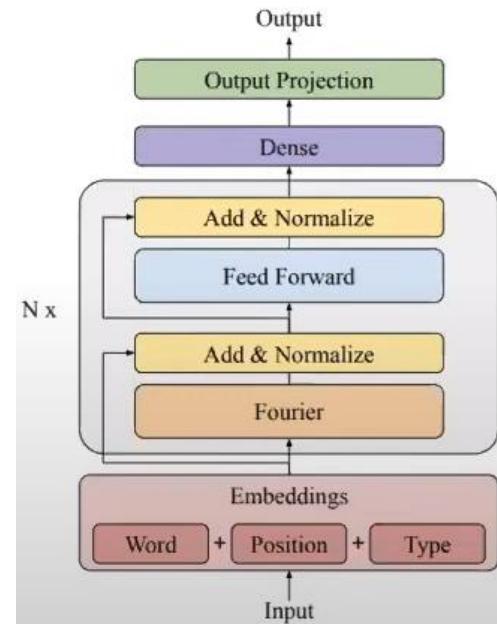
Может есть какие-то более интересные способы перемешивать информацию о токенам и учитьывать ее всю?
Так и появилась F-NET (*перемешивать информацию через преобразование Фурье*):

- заменим Attention-layer на Fourier-layer

- само Фурье-преобразование изображено справа:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}, \quad 0 \leq k \leq N - 1$$

- будем сначала перемешивать информацию внутри компонент эмбединга для токена, а далее перемешиваем информацию по всем токенам
- сложность данного *Fast Fourier Transform* = $N * \log(N) \Rightarrow$ тем самым уходим от квадратичной сложности в классическом трансформере
 \Rightarrow Благодаря такому подходу удалось добиться 92% точности BERT'а при ускорении работы в 7 раз (и снижения по памяти в 2 раза) даже для коротких последовательностей.



Past_Key_Values – или как ускорить decoder в трансформере?

Сейчас мы рассмотрим некоторую эвристику, позволяющую ускорить работу энкодера в трансформере.

Вспомним, что в дэкордере используется Masked Self-Attention, в котором мы смотрим только на предыдущие элементы контекста, поэтому нам нет смысла на каждой новой итерации предсказания токена перерасчитывать старые представления (так как они будут рассчитываться так же, как и раньше) \Rightarrow будем хэшировать скрытые состояния и на каждом новом предсказании токена просто подтягивать их из хэш-таблицы.

Пример: если мы рассчитали все представления для 100-го токена (для всех его слоев), то после предсказания 1010-го токена нам нет смысла перерасчитывать векторы для 100-го, так как он как раньше основывался только на 1-99, также и сейчас он просто повторит свои вычисления. Поэтому будем хэшировать, а потом просто подтягивать.

Способы распараллеливания трансформеров (3 способа)

Распараллелить по батчу: реплицируем модель на несколько ГПУ, распиливаем батч и для каждого отдельного «куска» батча делаем forward-pass на своей ГПУ. После этого считаем ошибку, собираем градиенты и обновляем все копии моделей.

Распараллелить по слоям: распиливаем модель на несколько групп по слоям и ложим каждую группу на свою ГПУ. Каждая группа делает свои вычисления и передает результат другой группе. Такой способ используется, если модель не влезает на одну ГПУ.

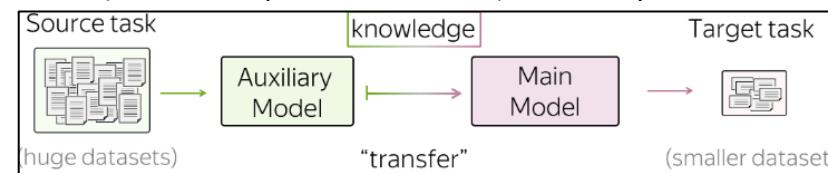
Распараллелить по головам: поскольку каждая голова attention делает вычисления независимо от другой, то нет смысла выполнять вычисления последовательно

Transfer Learning

В данном разделе мы будем говорить именно о применение ранее изученных подходов и моделей, а не про координально новые механизмы.

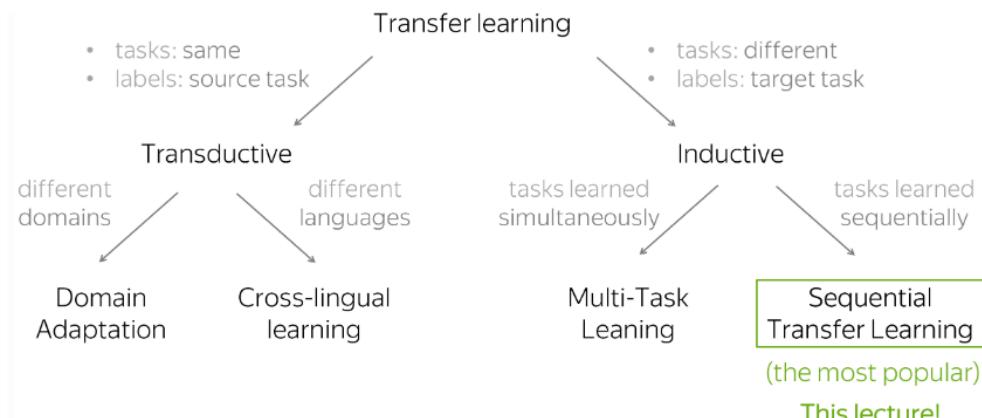
Основной идеей «*transfer learning*» является «перенос знаний» от одной модели к другой. Допустим мы хотим построить модель для решения нашей специфичной задачи, но, к сожалению, мы не располагаем достаточным количеством данных для того, чтобы построить хорошо работающую модель. *Что нам делать?*

Но зато у нас есть уже обученные модели для других целей (для классификации, LM и тд) ⇒ почему бы нам не воспользоваться их «умениями»? Таким образом можем перенести «знания» модели о неинтересующей нас задачи (*source task*) для решения интересующей задачи (*target task*).



Какие бывают типы Transfer Learning?

- изначально делится на *transductive* (задачи одинакового типа и таргеты есть только у *source task*) и *inductive* (задачи разного типа и таргеты есть только у *target task*)
- нас больше всего будет интересовать *Sequential Transfer Learning*



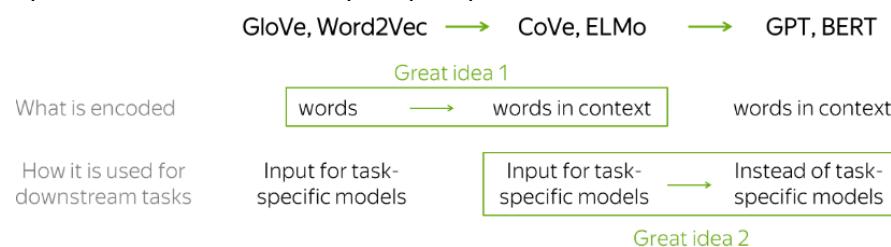
Pretrained Models – идею, которой мы пользовались при использовании предобученных эмбедингов (знающих много о вещах и мире) для решения нашей специфичной задачи, можно распространить на любые предобученные модели.

Далее мы рассмотрим 4 вида моделей:

(*CoVe, ELMo, GPT, BERT*).

У данных моделей бывает очень большое множество модификаций (от тренировочных данных до любых внутренних эвристик), но, грубо говоря, можно выделить 2 основные прогрессивные идеи, позволившие WordEmbedding'ам эволюционировать в *CoVe, ELMo*, а затем в *GPT, BERT*.

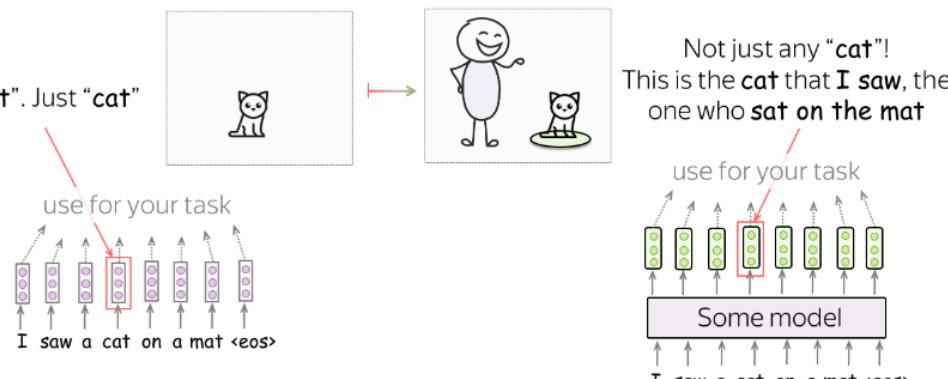
- что кодируем: **«слова -> слова в контексте»** (*Word2Vec, GloVe -> CoVe, ELMo*)
- использование: **«используем только эмбединги слов -> используем полностью предобученные модели»** (*CoVe, ELMo -> GPT, BERT*)



Идея №1: “From Words To Words-in-Context” – как мы можем видеть, идея «переноса знаний» посредством векторных представлений появилась сильно раньше, чем идея по «переносу» с помощью цельных моделей. ⇒ вместо того, чтобы переносить знания об отдельных словах (эмбединги слов), будем переносить умение воспринимать их различных контекстах (умение «читать текст»)

Следует вспомнить о том, какую выгоду мы получили, начав изучать нейро языковые модели (NLM)? С помощью них мы научились «читать» текст и воспринимать слова не как просто «слова с некоторым смыслом», а как часть последовательности.

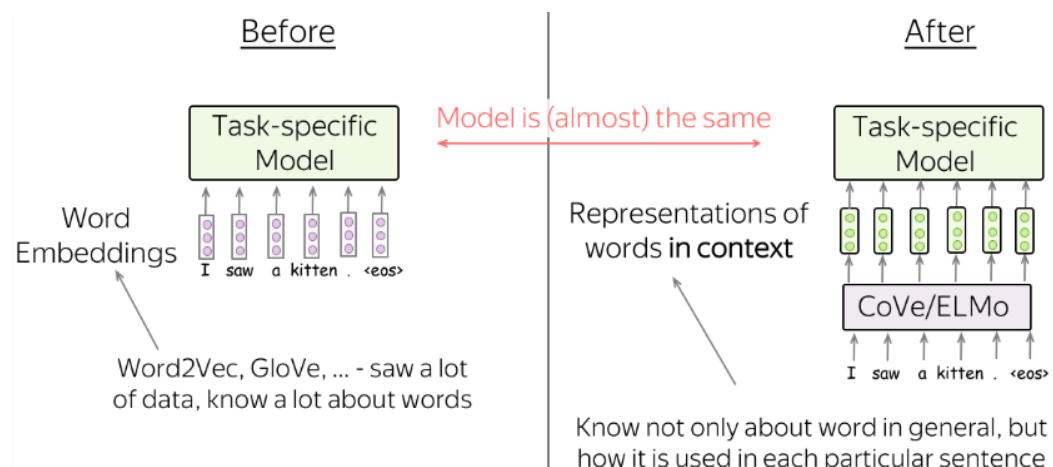
Можно обратиться к примеру справа, чтобы получше вспомнить эту разницу. «cat». Just «cat» Если мы оперируем просто предобученными эмбедингами (W2V, GloVe и т.д.), то мы будем воспринимать «кошку» просто как некоторое животное: маленькое, пушистое и красивое.



Если мы будем «читать» текст моделью, то «кошку» мы будем воспринимать не абы какую, а конкретно, которую мы видим, и которая сидит на коврике. Тем самым в дальнейшем мы будем лучше и глобальнее воспринимать любой текст.

На этой идеи основаны две

модели, которые будут разобраны далее: *CoVe*, *ELMo*. Они помогают нам получить новые, более осмысленные эмбеддинги для слов из текста. Дальнейшая работа с этими эмбеддингами будет почти такой же, как работа с эмбеддингами из W2V или GloVe (отличается лишь способ их получения).



CoVe (Contextualized Word Vectors Learned in Translation) – наконец-то первая модель 2017 года, в которой была реализована идея перехода от обычных эмбеддингов к контекстным эмбеддингам.

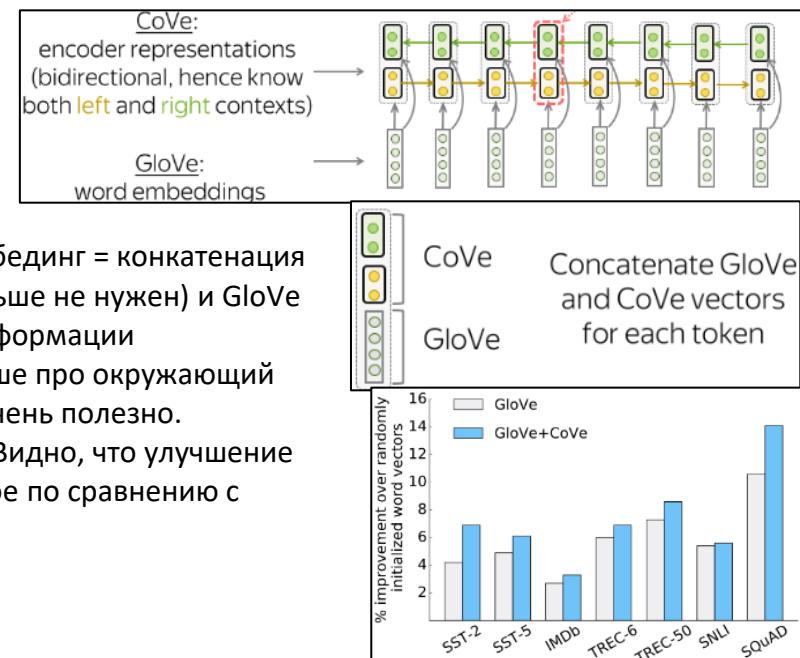
Что это и как тренировать? Neural Machine Translation (LSTMs and Attention) (Bahdanau Attention)

Таким образом будем тренировать такую систему для решения задачи машинного перевода (NMT – перевод с одного языка на другой), а далее будем использовать энкодер (LSTMs, GRU, и т.д.) такой системы для получения эмбеддингов, связанных с контекстом. На вход в модель будем подавать предобученные векторы GloVe и далее обрабатывать их энкодером.

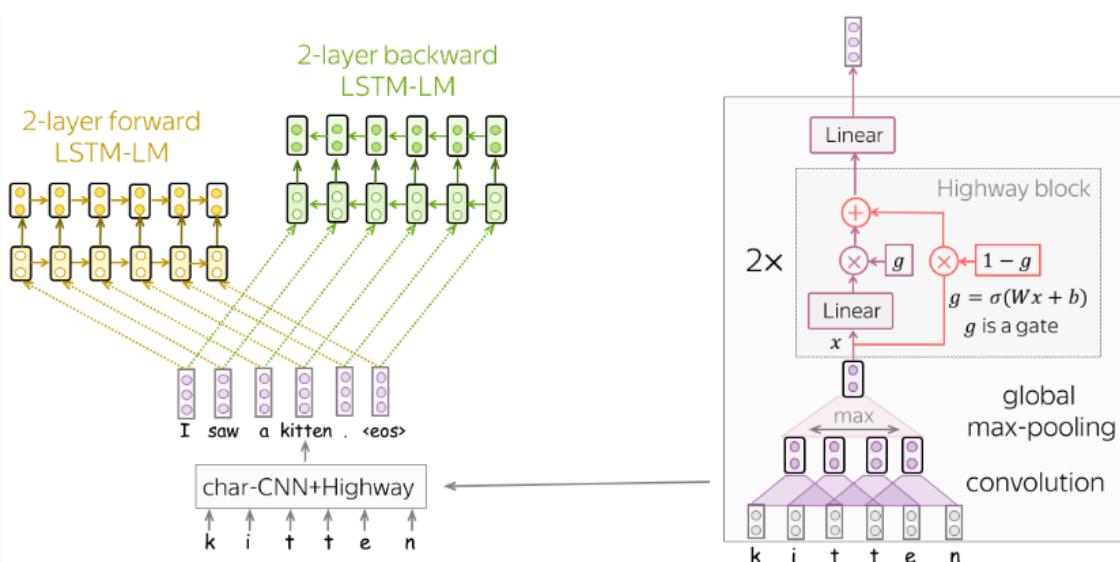
Что за энкодер? – отсылаясь к *Bahdanau Attention*, можно вспомнить, что энкодер в такой системе – *Bidirectional LSTM*. А поэтому выходные векторы из такой RNN содержат информацию и предыдущего, и последующего контекста.

Получение финальных эмбеддингов – финальный эмбеддинг = конкатенация выходов энкодера (декодер после обучения нам больше не нужен) и GloVe векторов. Идея в том, что GloVe содержит больше информации непосредственно о слове, а выходы энкодера – больше про окружающий контекст \Rightarrow совместное использование может быть очень полезно.

Результаты – можно наблюдать на картинке справа. Видно, что улучшение качества эмбеддингов, учитывающих контекст заметное по сравнению с обычными GloVe-эмбеддингами.



ELMo (Embeddings from Language Models) – в отличии от *CoVe*, *ELMo* использует для обучения языковую модель (LM), а не NMT модель. Такая замена GloVe векторов, представлениями из LM произвела колоссальное улучшение качества моделей для широкого спектра NLP-задач.



Что это и как тренировать? Forward and Backward LSTM-LMs on top of char-CNN

Структура модели состоит из двух двуслойных разнонаправленных LSTM. Интересная особенность предложенной структуры в том, как авторы получают начальные эмбединги слов. Напомним, что классические эмбединги слов тренируют один уникальный вектор для каждого слова, при этом в таком варианте:

- эмбединги знают смысл слова и не знают, из каких букв состоит слово (они не знают, что: *represent*, *represents*, *represented*, *representation* близки по написанию)

- эмбединги ничего не могут сделать со словами, которых нет в словаре (out-of-vocabulary)

Для того, чтобы попробовать обойти эти проблемы, авторы предлагают в качестве эмбединга слова использовать выходы простой посимвольной CNN-сети, состоящей из сверток, глобал пулинга, highway connections и FC-слоя.

Получение финальных эмбедингов – окончательный эмбединг для каждого слова будет состоять из 3-х вещей:

- эмбединг, полученный из char-CNN сети

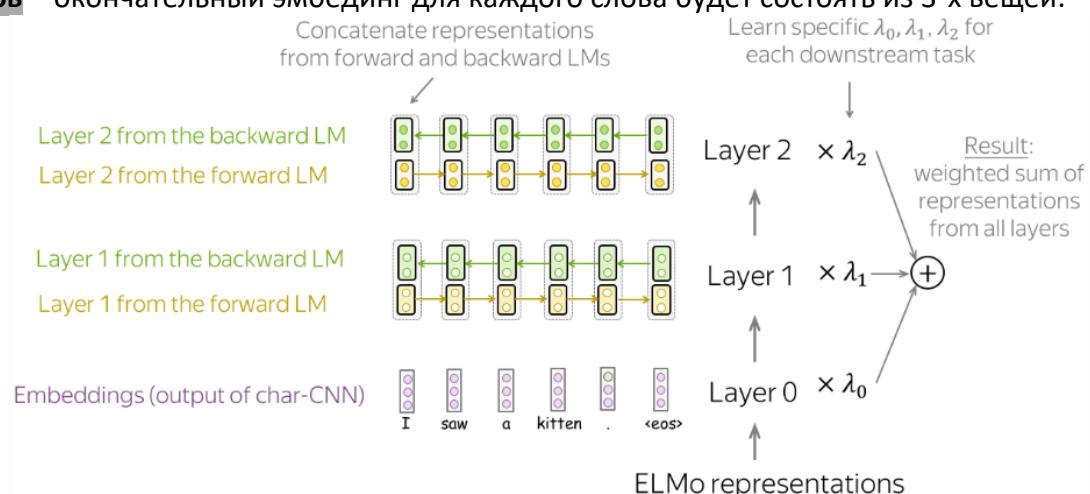
- сконкатенированные представления выходов 1-х слов forward и backward сеток

- сконкатенированные представления выходов 2-х слов forward и backward сеток

Идея нам не нова: каждый из эмбедингов несет информацию разного уровня \Rightarrow объединим их.

В 1-м слое информация о более низкоуровневом контексте, во 2-м слое о более верхнеуровневом.

Далее все 3 слоя складываются с различными весами (λ_0 , λ_1 , λ_2). Поскольку дальнейшее применение эмбедингов зависит от target-task, то и веса (λ_0 , λ_1 , λ_2) будут обучаемыми для того, чтобы иметь возможность сместить акцент на какую-либо часть предоставленной информации.



Идея №2: "Refuse From Task Specific Models" – следующие два класса моделей, которые мы рассмотрим (*GPT*, *BERT*), очень отличаются тем, как они используются для решения target-task'a.

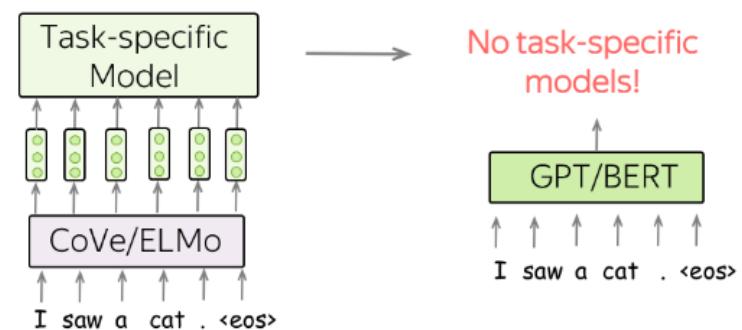
\Rightarrow CoVe/ELMo заменяют эмбединги слов, GPT/BERT заменяют целые модели

Повторим еще раз:

Было: (Specific model architecture for each downstream task) – CoVe/ELMo преимущественно заменяли embedding layer, после которого применялась новая модель для решения target-task. Разные target-task – разные task-specific models.

Стало: (Single model which can be fine-tuned for all tasks) –

в отличии от предыдущих моделей, GPT/BERT используются не для замены эмбедингов слов, а сразу для замены task-specific model. Таким образом сначала модели обучаются на огромном количестве неразмеченных данных, а далее fine-tune для каждой downstream-task.



GPT (Generative Pre-Training for Language Understanding)

GPT – это языковая *left-to-right* модель на базе трансформера с 12-ти слойным transformer decoder (без decoder-encoder attention). Заметим, что декодер-онли архитектура больше предназначена для задач генерации и подобных (в отличие от энкодер-декодер, которая используется, например, для задачи машинного перевода). У GPT есть только одна парадигма обучения, совпадающая с обучением LM \Rightarrow «читаем» текст и учимся предсказывать следующий токен. Таким образом обучение происходит на классической кросс-энтропии, а также лосса для всего предложения:

$$L_{xent} = - \sum_{t=1}^n \log(p(y_t | y_{<t}))$$

$$L = L_{xent} + \lambda \cdot L_{task}$$

Fine-Tuning (Using GPT for Downstream Tasks) – лосс для fine-tuning'а состоит из лосс вышеупомянутой языковой модели, а также task-specified лосса:

На этапе fine-tuning'а в архитектуре модели изменяется только финальный линейный слой.

Справа представлена схема для GPT fine-tuning'a

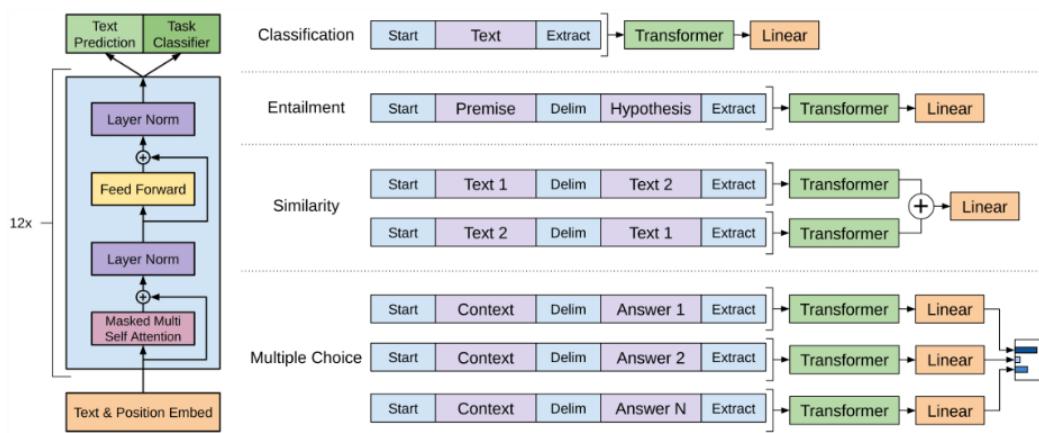
Разбор ниже под картинкой

1) Single sentence classification: для классификации предложений

просто «считаем» моделью нужный текст, а далее будем предсказывать класс, основываясь на финальном представлении последнего токена.

Примеры задач:

- SST-2 – оценить положительный или отрицательный текст
- CoLA (Corpus of Linguistic Acceptability) – понять лингвистически корректен ли текст



2) Sentence pairs classification: для классификации пар предложений «скормим» оба текста разделенных специальным токеном-сепаратором (delim). Далее будем делать прогнозы также основываясь на финальном представлении последнего токена.

Примеры задач:

- SNLI-2 – задача на логическое следование. Нужно решить второе предложение является ли логическим следствием, либо противоположностью, либо оно нейтрально
- QQP (Quora Question Pairs) – понять, являются ли тексты семантически одинаковыми
- STS-B – оценить похожесть текстов по шкале от 1 до 5

3) Question answering and commonsense reasoning: в такого рода задачах дается некоторый текст контекста Z, спрашиваемый вопрос q и набор возможных ответов A_k. Будем конкатенировать документ и вопрос и через delim ставить один из ответов. Для каждого возможного ответа будем пропускать соответствующие наборы «контекст+вопрос+ответ» через GPT-сеть, а дальше брать softmax по выходам набора. Далее – принимать решение.

Примеры задач:

- RACE (reading comprehension) – дан текст, вопрос и множество ответов. Нужно выбрать корректный.
- Story Cloze – дана история из 4-х предложений и 2 возможных окончания истории. Нужно выбрать корректный

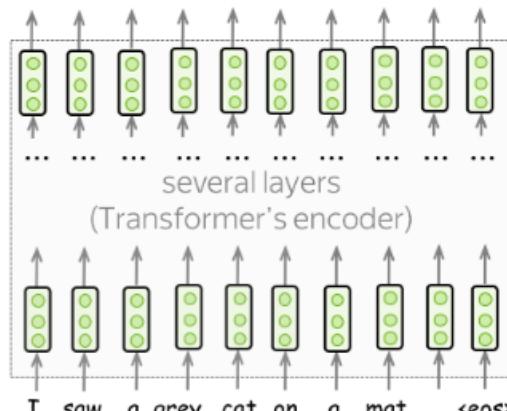
BERT (Bidirectional Encoder Representations from Transformers)

В отличии от GPT – языковой модели на базе трансформера, представляющей собой просто transformer decoder, BERT представляет собой transformer encoder (в котором self-attention накапливает не только предыдущий контекст, как в декодере, а полный контекст предложения)

Как обучать? – вопрос достаточно актуальный, поскольку до этого мы обучали только left-to-right LM (для декодер-онли моделей подход такой же, так как модель оперирует только предыдущими токенами и не заглядывает вперед, а для энкодер-онли моделей нужны эвристики).

Pre-Training Objectives: Next Sentence Prediction (NSP) Objective

В данной парадигме обучения BERT'а будем искать связи между двумя предложениями. Предложения будут передаваться парами, разделенными специальными токеном-сепаратором. Также в дополнение к эмбедингам слов и позиционным эмбедингам, также будем передавать сегментные эмбединги (к какому предложению относится слово). Самым первым токеном в модель подается токен <CLS>.



Model architecture:

- Transformer's encoder

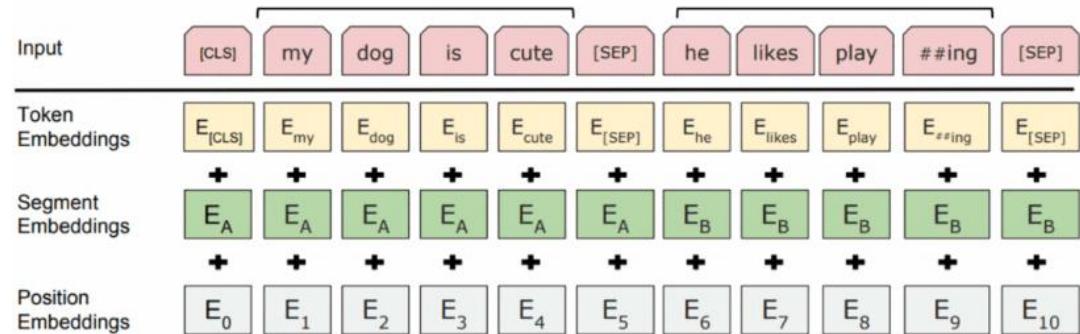
What is special about it:

- Training objectives
 - MLM: Masked language modeling
 - NSP: Next sentence prediction
- The way it is used
 - No task-specific models

Такое предсказание по наличию связи с последующим предложением является задачей бинарной классификации. По финальному представлению токена `<CLS>` модель предсказывает, могут ли эти предложения поданные в модель быть последовательными (друг относительно друга). При таком обучении 50% примеров тренировочный выборки состоят из последовательных предложений, извлеченных из тренировочных текстов, а другие 50% из рандомных пар предложений.

⇒ такой подход учит модель понимать связи между предложениями (учим воспринимать одно как контекст другого)

Pair of sentences: either consecutive or random (50%/50%)



Pre-Training Objectives: Masked Language Modeling (MLM) Objective

BERT имеет две основные парадигмы обучения, и самой популярной из них является MLM, в которой на каждом шагу происходит следующее:

- выбирается некоторое количество токенов (каждый токен выбирается с вероятностью 15%)
- заменяют выбранные токены (специальным токеном `<mask>` с $p=80\%$, рандомным токеном с $p=10\%$, оставляем нетронутым с $p=10\%$)
- пытаемся предсказать оригинальные токены на месте замененных

MLM-парадигма все также является языковой моделью, поскольку мы требуем предсказать какие-то замаскированные токены, основываясь на какой-то части данного текста.

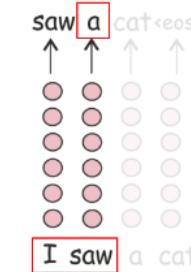
Сравним MLM и LM по-подробнее:

Классическая LM предсказывает следующий токен, основываясь на только на предыдущих, таким образом все финальные представления токенов будут учитывать только предыдущий контекст.

MLM, с другой стороны, видит весь текст целиком (только без замаскированных слов), таким образом все представления пытаются учитывать и предыдущий, и будущий контекст (поэтому BERT – bidirectional) (и в отличие от ELMo, которой для учета и предыдущего, и будущего контекста требовалось две разные LSTM сети с дальнейшей конкатенацией их представлений, у BERT'а все иначе: требуется лишь одна модель).

Language Modeling

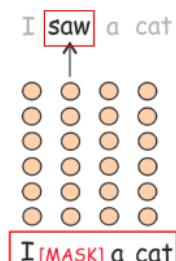
- Target: next token
- Prediction: $P(*) | I \text{ saw}$



left-to-right, does not see future

Masked Language Modeling

- Target: current token (the true one)
- Prediction: $P(*) | I [\text{MASK}] \text{ a cat}$



sees the whole text, but something is corrupted

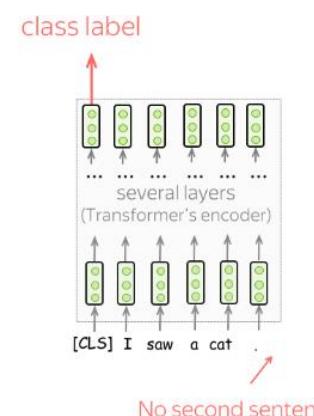
Fine-Tuning (Using BERT for Downstream Tasks)

Далее посмотрим, как применять такую архитектуру для различного рода задач. Сейчас рассмотрим только самый простой вариант *fine-tuning'a* модели для каждой из downstream tasks.

1) Single sentence classification: для классификации посредством BERT'а просто «считаем» им всем предложение, поставим в самое начало `<CLS>` токен, в котором будем аккумулировать информацию, и на основе представления которого будем делать классификационный прогноз.

Примеры задач:

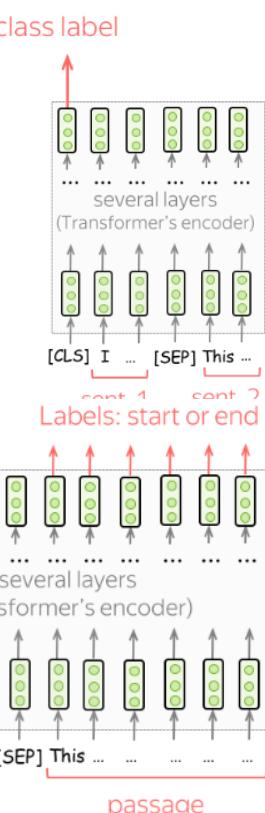
- SST-2 – оценить положительный или отрицательный текст
- CoLA (Corpus of Linguistic Acceptability) – понять лингвистически корректен ли текст



2) Sentence pairs classification: для классификации пар будем «скормливать» оба предложения так же, как мы делали на обучении. Схожим образом с задачей выше, классификационное предсказание будем проводить с помощью финального представления токена `<CLS>`.

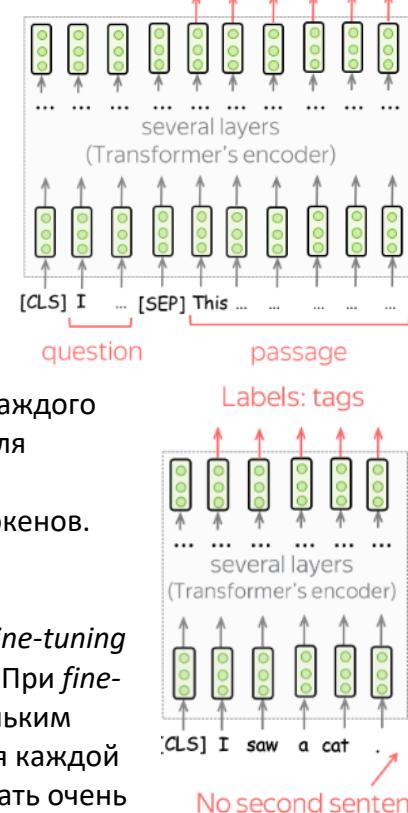
Примеры задач:

- SNLI-2 – задача на логическое следование. Нужно решить второе предложение является ли логическим следствием, либо противоположностью, либо оно нейтрально
- QQP (Quora Question Pairs) – понять, являются ли тексты семантически одинаковыми
- STS-B – оценить похожесть текстов по шкале от 1 до 5



3) Question answering: для QA авторы BERT'a использовали только SQuAD датасет. В нем каждый раз дается текст и вопрос по этому тексту. Во всех случаях ответом является маленькая часть текста, и поэтому задача модели – корректно выделить эту часть.

Чтобы использовать BERT для решения этой задачи, «скормим» в модель вопрос, а затем текст. Далее будем использовать все финальные представления токенов текста (именно текста, не вопроса) для того, чтобы понять является ли токен началом или концом нужного сегмента текста (ответа на вопрос).



4) Single sentence tagging: в такого рода задачи требуется предсказать класс для каждого токена последовательности. Например, в задаче Named Entity Recognition (NER) для каждого токена требуется его класс (персона, локация и т.д.). Соответственно для предсказания будем использовать все финальные представления каждого из токенов.

A Bit Adapters (Parameter Efficient Transfer) – до этого мы рассматривали только *fine-tuning* как метод «передачи знаний» обученной модели для решения downstream tasks. При *fine-tuning*'e мы берем уже обученную модель и дообучаем все ее параметры с маленьким learning rate для нужной задачи ⇒ дообучаем всю полную модель и к тому же для каждой новой задачи нам нужна будет новая копия предобученной модели (будет занимать очень много места).

К счастью, в 2019 году появилась альтернатива: перенос «знаний» модели с *adapters modules*.

В таком подходе параметры

преобученной модели остаются неизменными и нам остается дообучить лишь несколько параметров, называемых «*adapters*». Таким образом перенос «знаний» становится очень эффективным, поскольку для всех downstream tasks будет использоваться одна нетронутая предобученная модель.

На картинке справа изображен пример adapter'a и самого слоя трансформера с интегрированными вовнутрь адаптерами.

Можно видеть, что сама структура адаптера очень проста: 2-х слойная FC сетка с нелинейностью внутри.

Важной особенностью

является то, что размерность скрытого слоя очень маленькая ⇒ нужно для того, чтобы общее число обучаемых параметров было также маленьким. Именно это делает эксплуатацию адаптеров очень эффективной.

Finetuning:

- need to update the whole (huge!) model for each task

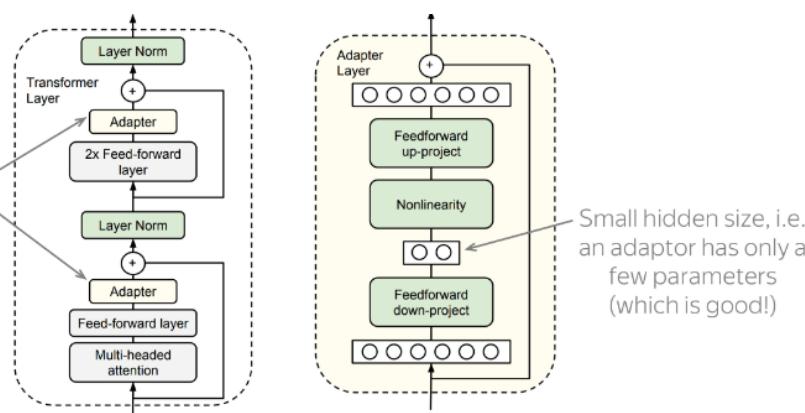
Parameters updated: 100% - **inefficient**

Adapters:

- model is fixed, train only small adapters

Parameters updated: e.g., ≈1% - **efficient**

Only these are trained,
everything else is fixed and
is the same for all tasks



Метрики качества для NLP-моделей

NLP-модели так же, как и другие модели всегда решают ту или иную задачу. И в зависимости от типа решаемой задачи для отслеживания качества и прогресса нам могут быть нужны различные метрики. Также важно заметить, что мы будем говорить именно о ML-метриках, которые используются исследователя во время жизненного цикла моделей, но также существуют и бизнес-метрики, на которые важно обращать внимания во время, перед и после деплоя модели в продакшн.

- *intrinsic evaluation* (с фокусом на промежуточные характеристики – МЛ метрики)

- *extrinsic evaluation* (оцениваем финальный продукт – бизнесовые метрики)

Далее рассмотрим основные типы решаемых задач и соответствующие им метрики:

- **задача классификации:** тут все по классике – accuracy, precision, recall, roc-auc, f1-score и т.д.

- **задача регрессии:** все тоже уже знаем – mse, mae, rmse, mape и т.д.

- **задача ранжирования:** списочные и каскадные метрики – pres@k, AP@k, mAP@k, DCG, pFound

- **задачи для сравнения текстов:** также существуют метрики для сравнения текстов друг с другом в прикладных NLP задачах (разные задачи – разные метрики)

BLEU (Bilingual Evaluation Understudy) – метрика, используемая для сравнения какого-либо перевода с другим предложенным переводом. И несмотря на то, что данная метрика чаще используется в задачах машинного перевода, с ее помощью можно оценивать сгенерированный текст для различного рода задач (переформулирование, суммаризация и др.)

Принимает значения от 0 до 1, где значения ~ 0.6-0.7 считаются отличным результатом, поэтому результат около 1 стоит расценивать как нереалистичный и означающий переобучение. И перед тем, как разобраться с расчетом Bleu-Score вспомним про две концепции: N-грамы и Precision.

1) под **N-граммами** будем понимать упорядоченную подпоследовательность из N токенов в тексте

Пример: предложение = "The ball is blue", тогда:

- 1-gram (unigram): "The", "ball", "is", "blue"

- 2-gram (bigram): "The ball", "ball is", "is blue"

- 3-gram (trigram): "The ball is", "ball is blue"

- 4-gram: "The ball is blue"

2) под **Precision (Clipped Presion)** – долю слов в предсказанном предложении, которые также встречаются хотя бы в одном из авторских переводов. Также чтобы не получить такой проблемы: если мы предскажем "Не Не Не" для предложения "He eats an apple" precision покажет нам 3/3 = 1 ⇒ это плохо (нужно ограничить повторы). Будем ограничивать каждое корректно угаданное слово максимальным числом раз, когда оно встречается в каком-либо авторском переводе (то есть в нашем примере clipped precision = 1/3)

Как теперь считать BLEU?

- рассчитать *precision* для N(1,4)-грам ⇒ получим 4 различных значения (для каждого N)

- вычисляем геометрическое среднее для данных 4-х метрик *precision*

- будем наказывать очень короткие предсказания (в них, как можно догадаться можно получить идеальные метрики). Справа приведена формула для «наказания», где «c» - длина предсказанного предложения, «r» - длина авторского перевода.

$$\text{Brevity Penalty} = \begin{cases} 1, & \text{if } c > r \\ e^{(1-r/c)}, & \text{if } c \leq r \end{cases}$$

Итоговая формула:

$$\text{Bleu}(N) = \text{Brevity Penalty} \cdot \text{Geometric Average Precision Scores}(N)$$

Финальным значением метрики будет

являться произведение геометрического среднего для *precision* и наказания за короткую длину предсказания.

Разновидности:

Обычно BLEU-score вычисляют для разного значения N (размер N-грамм)

- BLEU-1: использует только 1-граммы в *precision*

- BLEU-2: использует геометрическое среднее для N(1,2)-грамм

- BLEU-3: использует геометрическое среднее для N(1,3)-грамм

- BLEU-4: использует геометрическое среднее для N(1,4)-грамм

Плюсы: легко вычисляемая и интерпретируемая метрика; метрика похожа на то, как сам человек оценивал бы качество перевода; независим от языка; можно использовать, когда присутствуют различные вариации перевода

Минусы: не учитывает смысл слов (то есть замена синонимами для человека приемлема, а тут она будет ошибкой); смотрим только на совпадения 1в1 ("rain" != "is raining"); не учитывает важности слов (перепутать артикли не так страшно); не учитывает порядок слов (можно накидать правильных слов в глупом порядке)

METEOR (Metric for Evaluation of Translation with Explicit Ordering) – является производной метрикой от BLEU-score, чтобы исправить его некоторые недостатки. METEOR является recall-oriented (как и ROUGE) метрикой, в то время как BLEU является precision-oriented. Под капотом, данная метрика вычисляет и precision, и recall, а далее объединяет их в F-метрику (похожую на F1-score, но с сильным перевесом в recall сторону (обычно альфа = 9).

$$F_{mean} = \frac{P \cdot R}{\alpha P + (1-\alpha)R}$$

Также как и BLEU, METEOR наказывает слишком короткие предсказания посредством *Chunk Penalty*. Для расчета используется: «um» – количество униграм в предсказании и «c» – число *chunks* (количество правильных последовательных «наборов»). В идеальном случае (при совпадении перевода и истины) chunk всего один, в наихудшем случае, когда нет даже двух последовательно правильных слов chunk amount = len(предсказание).

$$p = 0.5 \left(\frac{c}{u_m} \right)^3$$

Далее финальная METEOR-метрика вычисляется как:

$$M = F_{mean}(1 - p)$$

NIST (by US National Institute of Standards and Technology) – еще одна некая производная метрика от BLEU-score, в которой каждая совпадающая N-грамма берется с соответствующим ей весом (так как биграмма из artikelей менее ценна, чем биграмма из важных слов). Веса вычисляются исходя из частоты N-граммы (чем реже N-грамма, тем у нее больше веса). Также эта метрика не так сильно наказывает короткие предсказание, в отличие от BLEU-score.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) – на самом деле это набор метрик, самые популярные из которых мы сейчас рассмотрим. Цель этих метрик – сравнить предсказанное предложение с истинным переводом слова (очень схожа с BLEU-score).

ROUGE-N: вычисляем набор совпадающих N-грамм у предсказанного текста с истинным. В отличие от BLEU, ROUGE-N вычисляет не только precision для каждого набора N-грамм, а также recall и f1-score.

ROUGE-L: основывается на longest common subsequence (LCS) между предсказанным и истинным переводом. Для такой метрики также можно вычислять и precision, и recall, и f1-score.

ROUGE-S: тут будем вычислять совпадения не просто по N-граммам, а по skip-граммам. Тут также можно вычислить и precision, и recall, и f1-score.

Плюсы: легкий; похож на оценку машинного перевода человеком; независим к языку.

Минусы: не учитывает синонимы и смыслы слов.

BLEU vs ROUGE: у BLEU больше фокус именно на precision, ROUGE больше фокусируется на recall.

WER (Word Error Rate) – существуют задачи, когда предсказанный текст не должен иметь возможности «двойкой» трактовки (в машинном переводе – это норм), но, допустим, в распознавании речи (Speech-to-Text) – совсем не норм (мы должно точь-в-точь распознать слова). В таких задачах BLEU-score нам не очень подойдет. Для такого рода задач используются другие метрики, например: *Word Error Rate (WER)* or *Character Error Rate (CER)*. В них мы сравниваем предсказанное предложение и истинное word by word и считаем «число различий» между ними. Под различиями будем понимать:

- если слово есть в оригинале, но нет в предсказании (*Deletion*)
- если слова нет в оригинале, но есть в предсказании (*Insertion*)
- если слово предсказано с некоторыми изменениями (*Substitution*)

Тогда метрика будет выглядеть так:



Виды NLP-задач

- категоризация текстов
- классификация последовательности символов (распознавание именованных сущностей, определение частей речи слов)
- распознавание фраз
- извлечение информации из текста
- синтаксическая аннотация
- семантическая аннотация
- генерирование текста (генерация на основе распознанной речи, машинный перевод)

Методы предобработки текста

Для корректной работы и последующей обработки моделями мы должны предобрабатывать поступающий к нам текст. Под основными пунктами можно выделить:

- перевод всех букв в тексте в нижний или верхний регистры
 - удаление цифр (чисел) или замена на текстовый эквивалент (обычно используются регулярные выражения)
 - удаление пунктуации. Обычно реализуется как удаление из текста символов из заранее заданного набора
 - удаление пробельных символов (whitespaces)
 - токенизация (обычно реализуется на основе регулярных выражений)
 - удаление стоп слов
 - стемминг
 - лемматизация
 - векторизация
-

Вопросы?

- как применять идею negative-samplings при обучении word2vec? Если данная модификация корректирует лосс-функцию, то, как ее менять? При классическом обучении ФК-сетки и финальном софтмаксе (я так учил w2v) получается мы используем не оптимизированную версию? Как тогда внутри сетки скорректировать функцию потерь?

- в чем эксплуатационная разница decoder-only, encoder-only, decoder-encoder transformer based моделей? Как конкретно учится каждый подтип моделей? Зачем нас эти разные варианты? Для каких задач какая из них лучше?

<https://datascience.stackexchange.com/questions/65241/why-is-the-decoder-not-a-part-of-bert-architecture>

