

# Stepik. Neural networks and NLP. 4. Language models and text generation - Part 2

## 4.4 Агрегация, механизм внимания

Всем привет! Поговорим о том, как получить один вектор, представляющий целое предложение или даже текст. То есть — как агрегировать глобальный контекст. На входе у нас есть матрица, представляющая текст. В ней столько же строк, сколько слов в тексте (например). Количество столбцов равно размеру [эмбеддинга](#). На выходе блоков агрегации получается либо матрица, сжатая по длине, то есть по количеству строк, либо вообще один вектор. Размерность эмбеддинга, при этом, не меняется. Агрегация или [пулинг](#) применяются тогда, когда нужно получить представление информации вне зависимости от того, где конкретно во входных данных эта информация встречалась — в начале текста или в конце. Другими словами, мы получаем полезную выжимку данных, то есть знаем, о чем идёт речь, и теряем пространственную информацию, то есть забываем, где в тексте об этом говорилось. Если мы сжимаем весь текст в один вектор, то этот вектор уже представляет глобальный контекст. Модуль агрегации или пулинга, в целом, работает аналогично свёрткам: мы проходим по данным скользящим окном и к каждому окну применяем некоторую операцию. Наиболее популярны два вида пулинга — агрегация через усреднение и агрегация через выбор максимума. В обоих видах пулинга нет обучаемых параметров. Функция усреднения или взятие максимума применяются по отдельности к каждому каналу, то есть к каждому столбцу матрицы признаков. Результатом применения функции является единственное число, которое записывается в соответствующую ячейку выходного вектора. Вектор получается посредством применения этой же функции к остальным каналам. При этом, пространственная информация теряется только частично — вектор содержит информацию из двух входных векторов (как нарисовано на слайде). При этом, он не содержит информации о том, в каком именно входном векторе что было. Это приводит к постепенному увеличению ширины учитываемого контекста без увеличения количества параметров нейросети — например, один вектор на третьем уровне учитывает информацию из четырёх векторов с первого уровня, то есть рецептивное поле равно 4, вместо 3 на втором уровне.

Вход

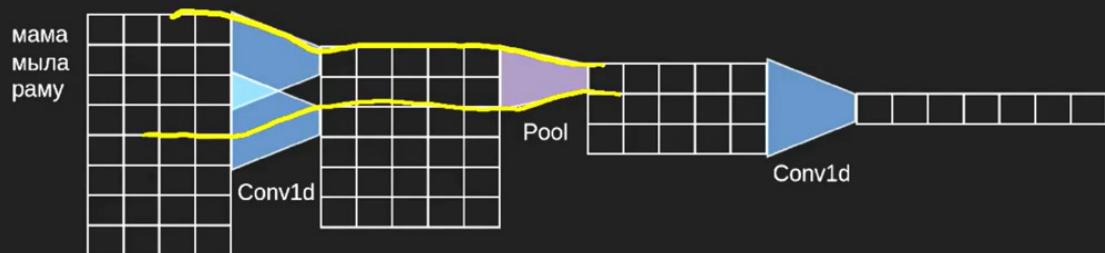
$Length \times EmbeddingSize$

Выход

$NewLength \times EmbeddingSize$  или  $EmbeddingSize$

Цель

- Расширение контекста с потерей пространственной информации
- Учет глобального контекста



- по аналогии со свёртками
- без параметров
- в рамках окна для каждого канала посчитать значение функции (avg, max) вывести в результирующий тензор
- пространственная информация частично теряется
- ширина контекста плавно увеличивается



Из комментариев:

Вопрос:

В чём отличие ембедингов от механизма внимания? Вроде и то, и другое в общем алгоритме применения нейросетей к тексту <https://stepik.org/lesson/247965/step/1?unit=220077> нужно для учёта контекста

Ответ (Романа Суворова):

Слово "эмбеддинг" в широком смысле - это просто какой-то вектор, который описывает объект (например, слово в тексте). В простейшем случае эмбеддинг слова можно получить из таблицы, как это говорится в шаге, который Вы упоминаете. Такой способ получения эмбеддинга не учитывает контекст. Чтобы учесть контекст, мы этот "первый вариант" эмбеддинга подаём в следующие слои нейросети, которые подмешивают в него информацию о текущем контексте. Таким образом получаются новые варианты эмбеддинга для того же слова. Механизм внимания - это один из способов учесть контекст, наряду с рекуррентными и свёрточными сетями.

Другими словами, в литературе слово "эмбеддинг" используется в двух смыслах: (1) векторное представление объекта; (2) самый простой способ получения этого векторного представления - таблица (в pytorch слой nn.Embedding).

Вопрос:

скользящее окно пулинга криво нарисовано, там ведь оно не пересекается между собой, ведь так?

Ответ (Романа Суворова):

может и пересекаться, зависит от настроек (размер окна и шаг окна). В PyTorch (агрегация через выбор максимума - [MaxPool](#), агрегация через усреднение - [AvgPool](#)) за это отвечают параметры *kernel\_size* (размер окна) и *stride* (шаг окна). Размер окна - количество элементов входной последовательности, которые будут агрегироваться за раз. Шаг окна - количество элементов, на которые окно будет сдвигаться. Возьмём для примера последовательность 121342 (одна цифра - один элемент).

Если *kernel\_size*=2 и *stride*=1, то на выходе MaxPool мы получим последовательность 22344:  $y_0 = \max(x_0, x_1)$ ,  $y_1 = \max(x_1, x_2)$  и т.п. - тут окна пересекаются.

Если *kernel\_size*=2 и *stride*=2, то на выходе MaxPool мы получим 234:  $y_0 = \max(x_0, x_1)$ ,  $y_1 = \max(x_2, x_3)$ ,  $y_2 = \max(x_4, x_5)$  - тут окна не пересекаются.

Если *kernel\_size*=3 и *stride*=1, то на выходе MaxPool мы получим 2344:  $y_0 = \max(x_0, x_1, x_2)$ ,  $y_1 = \max(x_1, x_2, x_3)$  - тут окна пересекаются на 2 позиции.

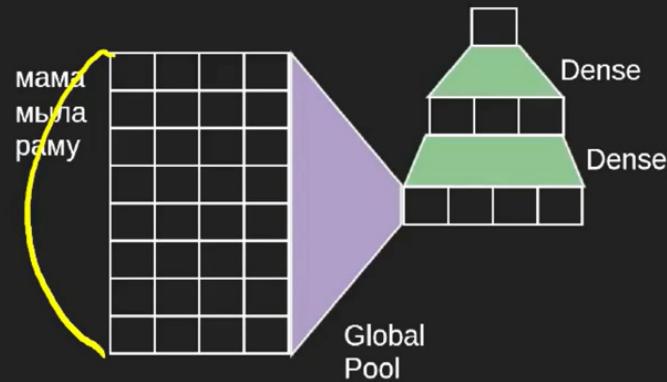
Если *kernel\_size*=3 и *stride*=2, то на выходе MaxPool мы получим 24:  $y_0 = \max(x_0, x_1, x_2)$ ,  $y_1 = \max(x_2, x_3, x_4)$  - тут окна пересекаются на 1 позицию (последний элемент тут отбрасывается и не влияет на результат из-за того, что длина входной последовательности не кратна).

Если *kernel\_size*=3 и *stride*=3, то на выходе MaxPool мы получим 24:  $y_0 = \max(x_0, x_1, x_2)$ ,  $y_1 = \max(x_3, x_4, x_5)$  - тут окна не пересекаются, все элементы последовательности учитываются.

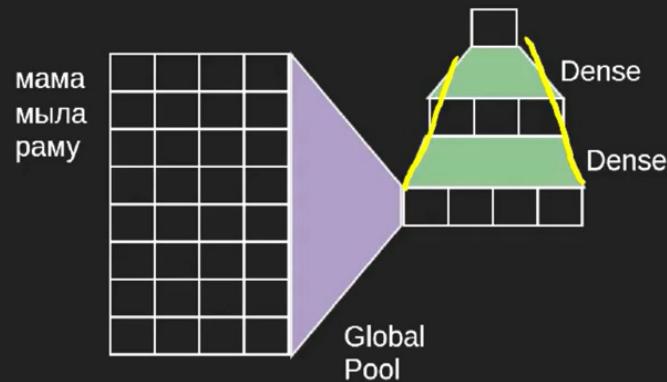
Можно сделать *stride* > *kernel\_size*, тогда не все элементы входной последовательности будут учитываться (окно будет перепрыгивать через *stride*-*kernel\_size* позиций).

Когда мы не хотим плавно понижать размерность, а хотим просто получить один вектор для всей последовательности, можно применять глобальный [пулинг](#). Как и локальный, он может делаться через усреднение или через выбор максимума. Отличие глобального пулинга от локального заключается в том, что размер окна всегда равняется длине входной последовательности. Если подают последовательность другой длины, то размер окна подстраивается под неё. В остальном же — всё то же самое. Количество каналов не меняется, обучаемых параметров нет. Пространственная информация, при этом, теряется полностью. Как

правило, глобальный пулинг используют ближе к последним слоям, то есть к выходу из сети. Таким образом получают представление, размер которого вообще не зависит от длины входной последовательности. Далее, к полученному вектору применяют один или несколько полносвязных слоёв, чтобы решать конечную задачу — например, задачу классификации текста. Вроде бы — всё хорошо, пулинг — это очень простая операция, которая позволяет скатить весь текст до одного вектора без обучаемых параметров, нет проблем с затуханием или [взрывом градиентов](#), гибкая — может принимать на вход последовательности разной длины, но есть пара проблем. Во-первых, каждый канал агрегируется независимо, то есть мы не можем агрегировать один канал в зависимости от значения другого канала. Это ограничивает нас. Во-вторых, единственное выразительное средство в пулинге — это амплитуда значения — она отвечает и за значимость, и за полезную информацию, которую нужно использовать в дальнейшем. Другими словами, пулинг (особенно глобальный) — операция достаточно слабая и грубоватая для многих задач.



- получение векторного представления для всего текста



- получение векторного представления для всего текста
- полная потеря пространственной информации
- ближе к выходу из нейросети
- часто дальше идут полно связные слои



- получение векторного представления для всего текста
- полная потеря пространственной информации
- ближе к выходу из нейросети
- часто дальше идут полносвязные слои

### Проблемы пулинга

- каждый канал агрегируется независимо
- амплитуда значения отвечает и за значимость и за полезную информацию одновременно
- слишком слабое преобразование



Умные люди придумали агрегацию получше — механизм внимания. Давайте представим, что входные векторы — это люди, которые что-то обсуждают вместе. Вы — один из участников такого обсуждения и вы хотите высказаться. Естественно, вы хотите чтобы вас услышали, но если все вокруг вас бурно дискутируют, бессмысленно в таком шуме что-то пытаться говорить. Сначала нужно привлечь к себе внимание, и убедиться, что вас слушают, и только тогда вы уже будете смело говорить то что хотите сказать. Таким образом, гораздо более эффективно разделить операции оценки значимости информации (то есть, привлечение внимания), и использования этой информации (то есть, передачи дальше по нейросети). Примерно такая идея заложена в основу механизма внимания. Давайте теперь рассмотрим его работу более формально. Итак, на вход механизма внимания поступает матрица размерности "длина текста" на "размер [эмбеддинга](#)". Сначала к вектору каждого слова мы, независимо, применяем некоторую нейросеть, у которой один выход — оценка значимости соответствующего элемента текста (слова, например). Про эту оценку мы знаем только то, что это вещественное число, ничем не ограниченное, и оно для более значимых слов — больше, а для менее значимых слов — меньше. Часто [релевантность](#) рассчитывается с помощью однослойной нейросети, но ничего не мешает использовать и более сложное преобразование. Релевантность для всех элементов текста рассчитывается с помощью одной и той же нейросети, её веса не зависят от каждого конкретного слова (это важно). Далее мы нормируем оценки релевантности так, чтобы их сумма равнялась единице. Самый удобный способ сделать это — применить [софтмакс](#). Далее мы домножаем оценки релевантности на соответствующие значения входных векторов и получаем результирующий вектор. Размерность результирующего вектора будет такая же,

как и размерность эмбеддингов слов на входе. Можно записать эту операцию через матричное произведение вектора на матрицу.

Доп. комментарий (от Романа Суворова):

Исходный вопрос: @Николай\_Капырин, вот это "Размерность результирующего вектора будет такая же, как и размерность эмбеддингов слов на входе" для меня оказалось неожиданным.

При виде софтмакса я решил, что мы ищем самое важное слово в тексте, которое как бы передает смысл текста...

@Anonymous\_117776891, всё верно, софтмакс ищет самое важное слово в тексте и на его выходе мы имеем вектор, в котором столько же элементов, сколько элементов во входной последовательности. Однако на софтмаксе работа механизма внимания не заканчивается: мы хотим не просто выбрать самые важные слова, но и агрегировать их признаки в один вектор. Для этого мы перемножаем выход софтмакса и входную матрицу признаков, в результате получаем один вектор, размерность которого совпадает с размерностью входного пространства (то есть количеству элементов в эмбеддинге одного слова).

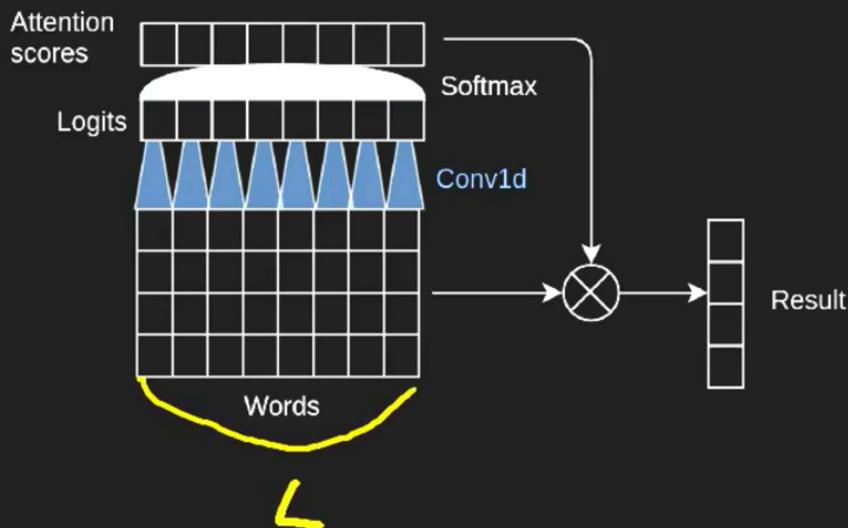
## Идея "умного" пулинга

SAMSUNG  
Research  
Russia

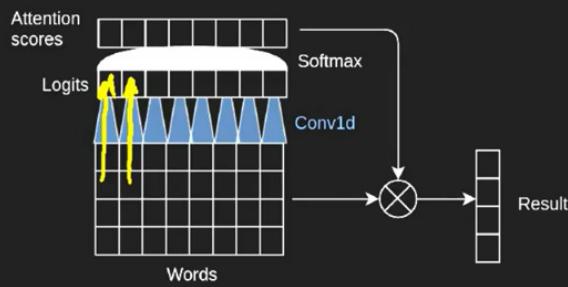
- мы - участники обсуждения и нам есть что сказать по теме
- мы хотим, чтобы нас услышали
- сначала мы должны привлечь к себе **внимание**
- только потом мы начнём **говорить**, убеждаясь, что нас слушают
- разделить операции **оценки значимости вектора** и **получения информации из него**



SAMSUNG  
Research  
Russia



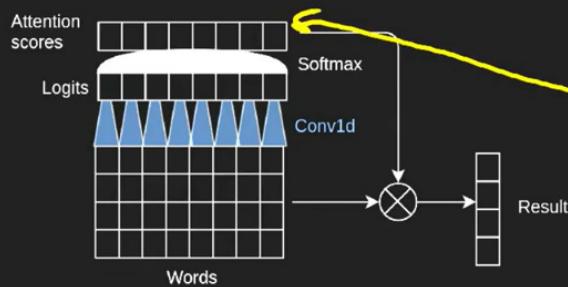
$$Words \in \mathbb{R}^{L \times S}$$



$$UnnormScores = Net(Words) \in \mathbb{R}^L$$



$$Words \in \mathbb{R}^{L \times S}$$

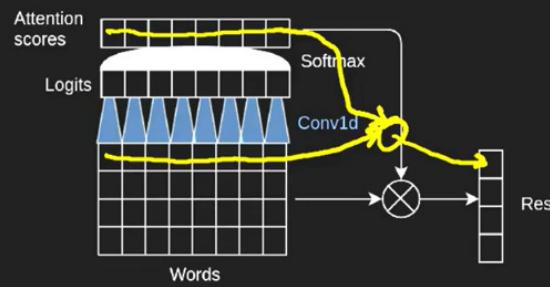


$$UnnormScores = Net(Words) \in \mathbb{R}^L$$

$$AttScores = \text{Softmax}(UnnormScores) \in \mathbb{R}^L$$

$$\text{Softmax}(x) = \left\{ \frac{e^{x_i}}{\sum_j e^{x_j}} \right\}_i$$





$UnnormScores = Net(Word) \in \mathbb{R}^L$

$AttScores = Softmax(UnnormScores) \in \mathbb{R}^L$

$Softmax(x) = \left\{ \frac{e^{x_i}}{\sum_j e^{x_j}} \right\}_i$

$Result = AttScores \cdot Words \in \mathbb{R}^S$



Из практической задачи:

Вспомните основную формулу из предыдущей задачи с глобальным avg-пулингом:

$avg(Input) = \frac{1}{k} \sum_{i=0}^{k-1} Input[i]$

Технически, основное отличие механизма внимания от avg-пулинга - наличие весов при слагаемых:

$Attention[Ch] = \sum_{i=0}^{InLen-1} AttScores[i] \cdot Input[i, Ch]$

где  $Input \in \mathbb{R}^{InLen \times EmbSize}$  - матрица признаков входной последовательности,  $Ch$  - номер столбца (номер признака),  $AttScores[i]$  - оценка значимости для  $i$ -го элемента входной последовательности ( $AttScores \in \mathbb{R}^{InLen}$ ,  $0 \leq AttScores[i] \leq 1$ ).

**Примените механизм внимания к входной матрице**

$Input = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 1 & 3 & 0 \\ 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{InLen \times EmbSize}$

с учётом оценок значимости  $AttScores = (0.1 \quad 0.5 \quad 0.3 \quad 0.1)$ .

В общем случае, механизм внимания состоит из следующих основных компонентов. Во-первых, на вход может подаваться не только матрица слов, то есть данные, но и отдельный вектор-запрос, на основе которого будут рассчитываться релевантности слов. Термин "релевантность" мы будем использовать для обозначения оценки значимости слова, то есть релевантность —

это какое-то число, чем она выше, тем более слово значимо. Второй компонент — это механизм расчёта релевантности. Как правило, это простое [скалярное произведение](#) или нейросеть. Третий компонент — это механизм вычисления значений, то есть векторов, которые, в дальнейшем, будут складываться с весами, полученными в результате расчёта релевантности. Четвёртый компонент — механизм получения единого вектора, то есть агрегация. Как правило, оценки релевантности сначала нормируются с помощью [софтмакса](#), а затем входные вектора домножаются на эти оценки и поэлементно складываются. В результате мы получаем гораздо большую гибкость, чем при обычной агрегации. В частности, это проявляется в том, что значения выходного вектора более не являются независимыми. Теперь одни элементы входных векторов могут работать на привлечение внимания, а другие на передачу полезной информации.

SAMSUNG  
Research  
Russia

#### Компоненты

- Входные данные - матрица слов и вектор-запрос (опционально)
  - Механизм расчёта релевантности
  - Механизм вычисления значений
  - Механизм агрегации (softmax + sum)
- 
- Большая гибкость по сравнению с avg/max pooling
  - Значения результирующего вектора вычисляются с учётом друг друга



Так как веса входных векторов вычисляются адаптивно, то, с ростом длины последовательности, внимание не начинает работать хуже. В отличие от механизма внимания, [пулинг](#) с усреднением, при росте длины последовательности, приводит к затуханию амплитуды значений, потому что знаменатель в формуле усреднения является большим числом. И вообще, оказывается, механизм внимания — операция универсальная и может заменить и [рекуррентки](#), и свёртки — можно всю нейросеть построить только из внимания, и она будет отлично работать. И ещё одна приятная плюшка напоследок: если в нашей нейросети есть механизм внимания, мы получаем возможность частично интерпретировать решения, принятые нейросетью. Мы можем понять, на какие участки входных данных она обратила больше внимания, а на которые — меньше (простите за тавтологию).

## Компоненты

- Входные данные - матрица слов и вектор-запрос (опционально)
- Механизм расчёта релевантности
- Механизм вычисления значений
- Механизм агрегации (softmax + sum)

- Большая гибкость по сравнению с avg/max pooling
- Значения результирующего вектора вычисляются с учётом друг друга
- Лучше работает для длинных последовательностей (по сравнению с пулингом и RNN)
- Универсальная операция
- Возможность интерпретации решений нейронной сети



Вариантов механизмов внимания — огромное множество. Давайте рассмотрим несколько популярных. Во-первых, для получения результирующего вектора можно использовать не исходные вектора, а сначала их как-то преобразовать с помощью другой нейросети, и только потом уже агрегировать. При этом веса у нейросетей, отвечающих за получение значений и за оценку [релевантности](#), отличаются. Это даёт нам ещё больше гибкости. Часто, задача поставлена так, что слова во входной последовательности не являются релевантными или не релевантными сами по себе — их релевантность оценивается относительно запроса, то есть в контексте конкретной потребности пользователя. Например, если мы сравниваем два текста, то вектор-запрос характеризует наш запрос (то есть то, что мы ищем), а матрица — текст-ответ (то есть, где мы ищем). Часто, в таких случаях релевантность оценивается простым [скалярным произведением](#) вектора запроса и векторов из входной матрицы. Это можно также записать как матричное произведение вектора на всю матрицу. Иначе говоря, чем ближе векторы входных слов к вектору запроса в смысле некоторой метрики (например, косинусной), тем более они значимы. Можно все выше приведённые варианты скомбинировать, чтобы получить ещё больше гибкости — в варианте на слайде и значения рассчитываются отдельно от ключей, и релевантность рассчитывается с учётом внешнего запроса. По сути, это означает, что механизм вычисления релевантности может быть нелинейным — когда вначале исходные вектора преобразуются, а затем уже сравниваются с запросом. А ещё можно не брать запрос извне, а вычислить самостоятельно — например, с помощью глобального [пулинга](#) с выбором максимума. Тогда оценка значимости каждого слова будет обусловлена на весь текст. Механизмы внимания, использующие в качестве ключей и значений элементы одного и того

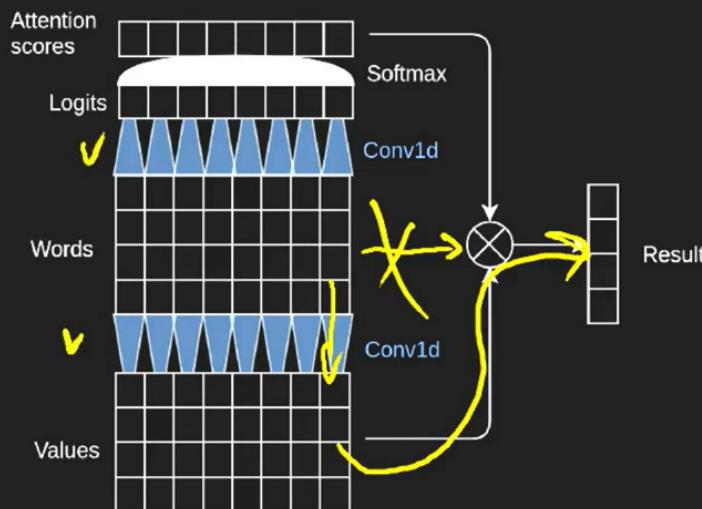
же текста ещё называются механизмами "[самовнимания](#)" (или "self-attention"). Такое внимание уже, в принципе, может заменить рекуррентную нейросеть в ряде задач.

Доп. комментарий:

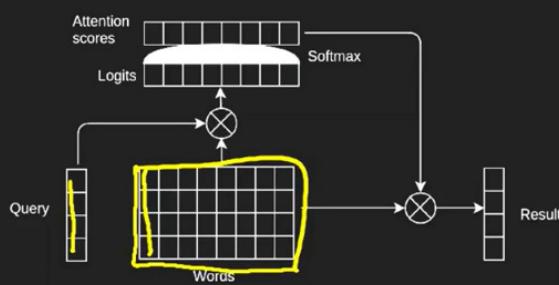
Исходный вопрос: @Николай\_Капырин, и что мы дальше делаем с result? Я из 4.4 этого не уловил

@Anonymous\_117776891, можно с ним делать всё что угодно. Поместить в декодер и раскрутить закодированное предложение в новую фразу (например, в машинном переводе или чат-боте), а можно поместить в полносвязную сеть и узнать что-то об авторе, или для кого текст написан, или классифицировать настроение текста...

Эта лекция, в основном -- про агрегацию, про то, как лучше "потерять" пространственную информацию о положении слов и сохранить больше "смысла".

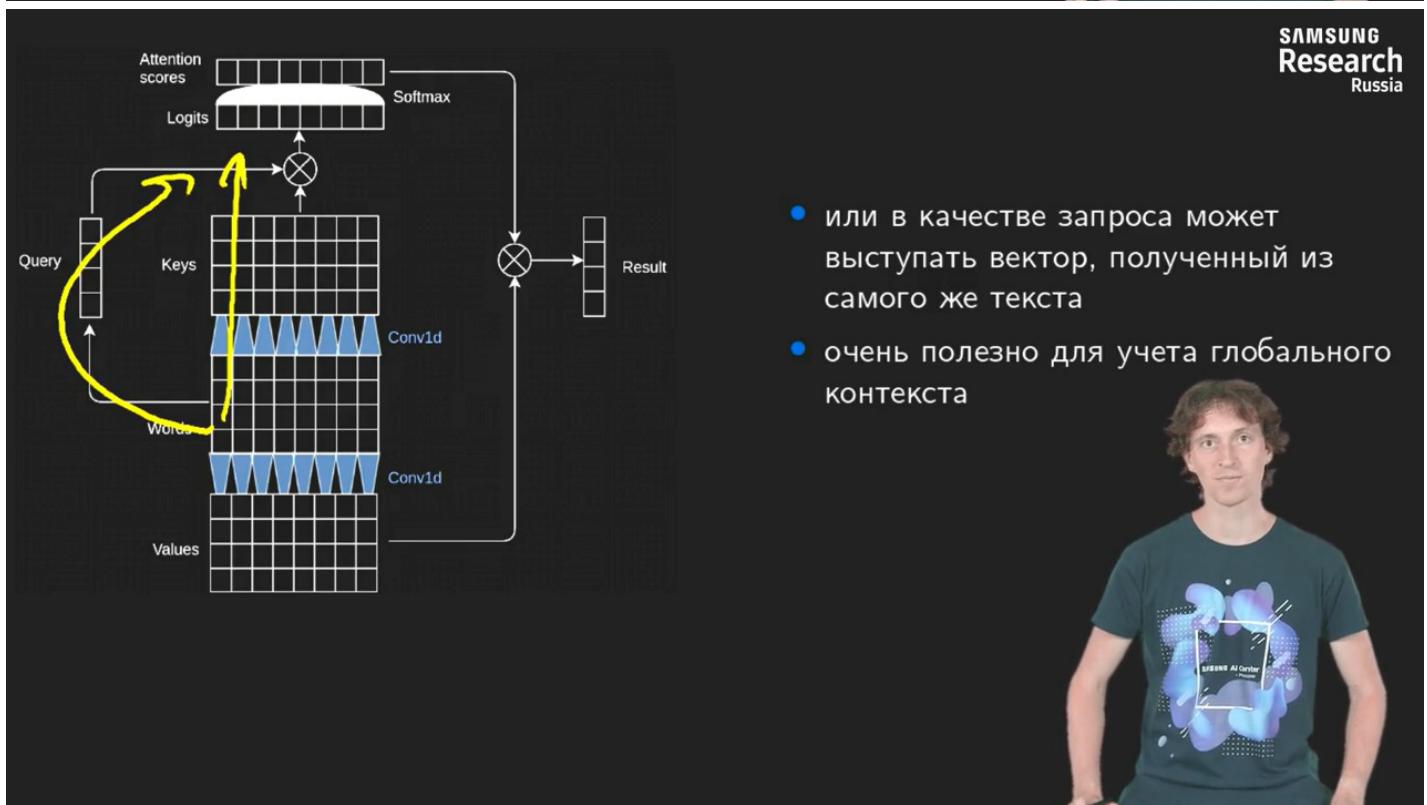
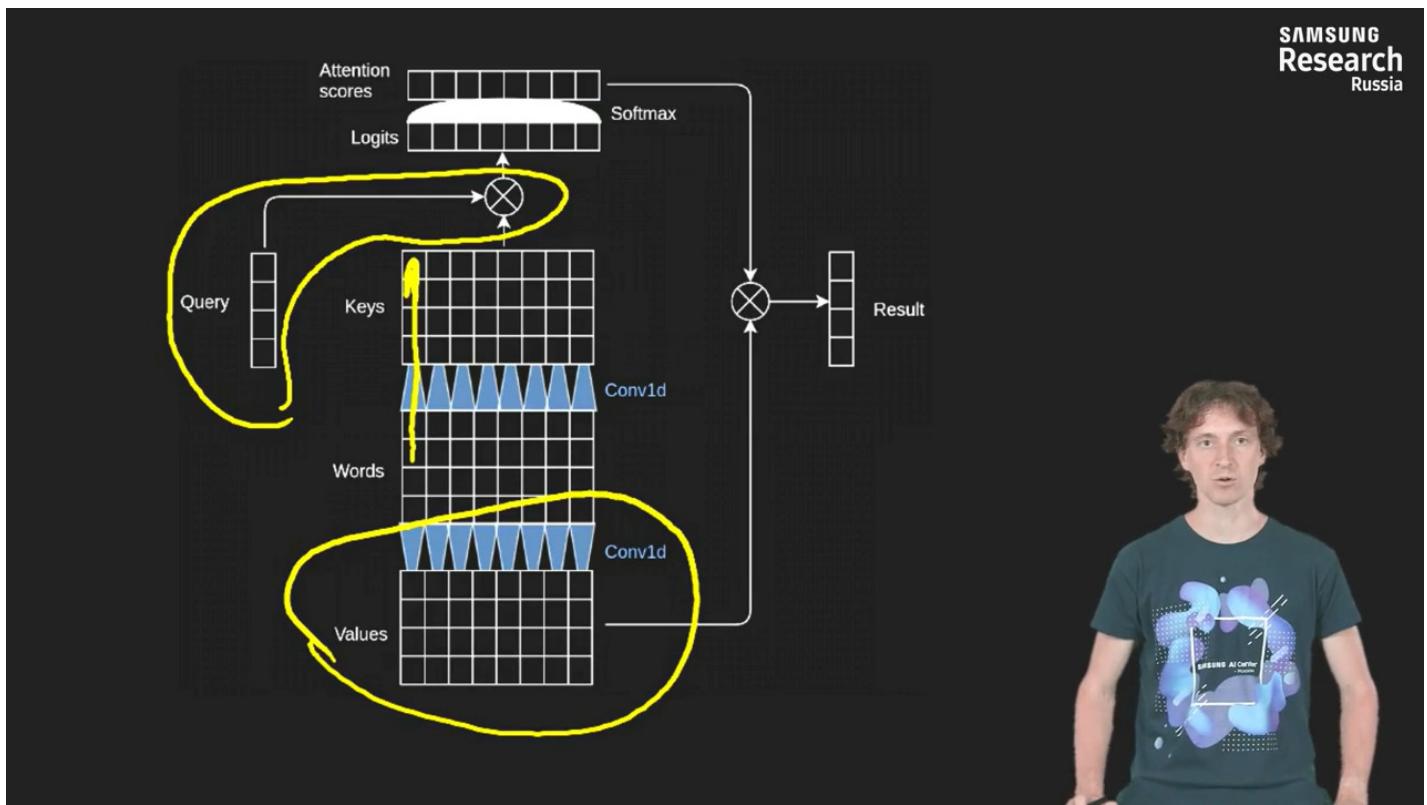


- больше гибкости - значения вычисляются



- запрос может прийти извне - при поиске, вопросе-ответе, сравнении текстов
- простое скалярное произведение для оценки значимости каждого слова относительно запроса





- или в качестве запроса может выступать вектор, полученный из самого же текста
- очень полезно для учета глобального контекста



Из комментариев:

Вопрос:

А можно ссылки на работы / статьи в блогах, чтобы подробнее прочитать? Или хотя бы хорошие ключевые слова.

Ответ (от студента):

[Attention Is All You Need](#) - классика

<https://habr.com/ru/post/341240/> тут и статья, и видео

Ответ (от Романа Суворова):

комментарий Сергея ссылается на self-attention - один из способов применения механизма внимания для того, чтобы найти признаки каждого слова (элемента последовательности) в контексте всех остальных слов (элементов последовательности). Другими словами, у self-attention и на входе и на выходе - последовательность. Self-attention с точки зрения применений - аналог рекурренток.

В этом уроке речь идёт о применении механизма внимания для агрегации - когда мы хотим взять последовательность, а на выходе получить один вектор. Оригинальная работа для такого применения механизма внимания - [Neural Machine Translation by Jointly Learning to Align and Translate, 2014](#), страница 3. [Хороший пост](#) про варианты такого внимания для seq2seq (генерации одной последовательности по другой, например машинный перевод).

Механизм self-attention и его реализация будет рассматриваться чуть позже в курсе ([уроки 4.5](#) и [4.6](#))

Ответ (от студента):

<http://nlp.seas.harvard.edu/2018/04/03/attention.html> — очень хорошая аннотация оригинальной статьи.

Вопрос:

Там где мы оцениваем релевантность относительно запроса, query обозначен как вектор, а текст как матрица. Query - это действительно одиночный вектор? И если да, то что это - одно слово, или результат агрегации какого-то текста-запроса?

Ответ (от Романа Суворова):

это зависит от ситуации. Можно его получить агрегацией (в этом же уроке такой вариант рассматривается), можно взять какое-то одно слово. Тут начинается architecture engineering - нужно Ваши знания о предметной области, о характере задачи, заложить в архитектуру.

Вопрос:

не совсем понятно откуда берутся запросы, если они из самого же текста, а не из вне. Ну и тут не совсем понятно, что же такое ключи

Ответ (от студента):

в self-attention ключи и запросы, а также и значения, действительно берутся из одного текста, но являются продуктами разных полносвязных слоёв.

Ответ (от студента):

спасибо большое за ответ. Я в более поздних видео примерно поняла, что это также слова из того самого контекста (предложения например), и имеем ли мы ключ, запрос, или значение получается в зависимости от того стоит ли слово как контекст или же наоборот (аналогия с word2vecом). Но у меня возникает тогда следующий вопрос: смотрим ли мы абсолютно все возможные сочетания слов? Это же сколько компьютерных операций тогда получается. Или я абсолютно неправильно поняла ситуацию с ключами, запросами в self-attention?

Из практического задания:

Рассмотрим механизм внимания, в котором релевантности элементов входной последовательности рассчитываются с помощью скалярного произведения с фиксированным вектором-запросом.

Как и раньше,  $Input \in \mathbb{R}^{InLen \times EmbSize}$  - входная матрица.

Релевантности элементов входной последовательности можно посчитать с помощью матричного произведения  $UnnormScores = Input \cdot Query$ , где  $Query \in \mathbb{R}^{EmbSize}$  - вектор-запрос, характеризующий значимость признаков. Тогда  $UnnormScores \in \mathbb{R}^{InLen}$  - вектор релевантностей элементов входной последовательности (чем больше  $UnnormScores[i]$ , тем значимее  $i$ -ый элемент).

Затем релевантности нормируют:  $AttScores = Softmax(UnnormScores)$ .

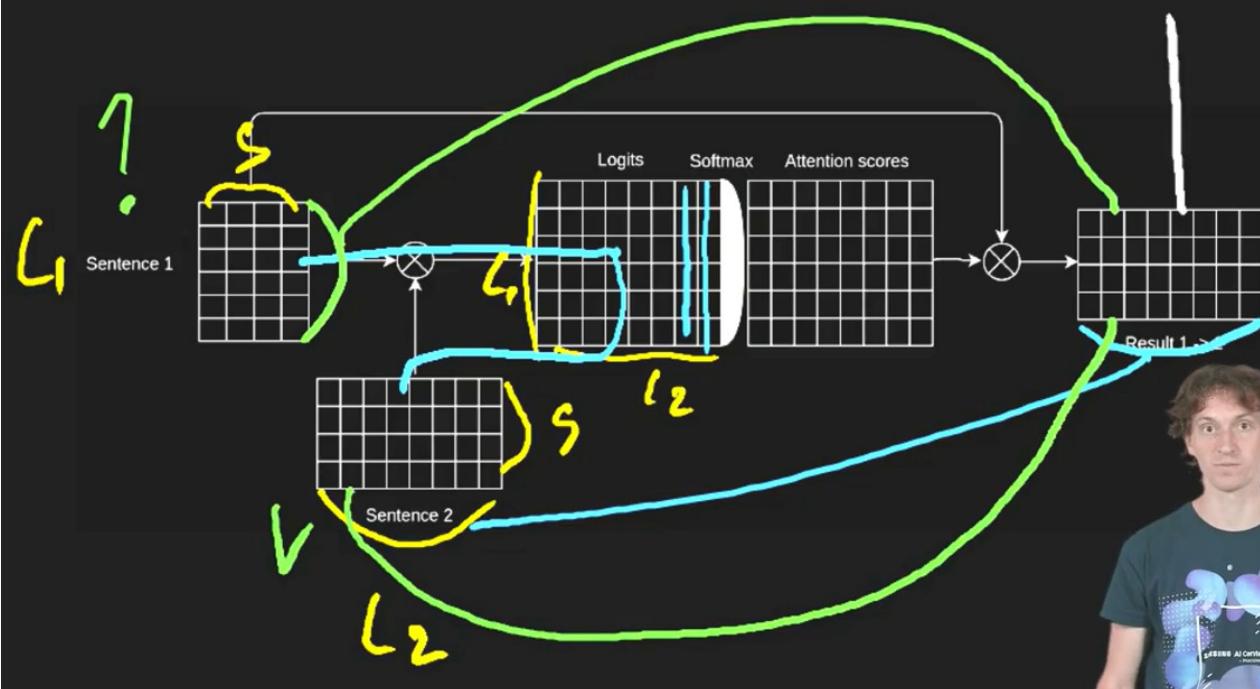
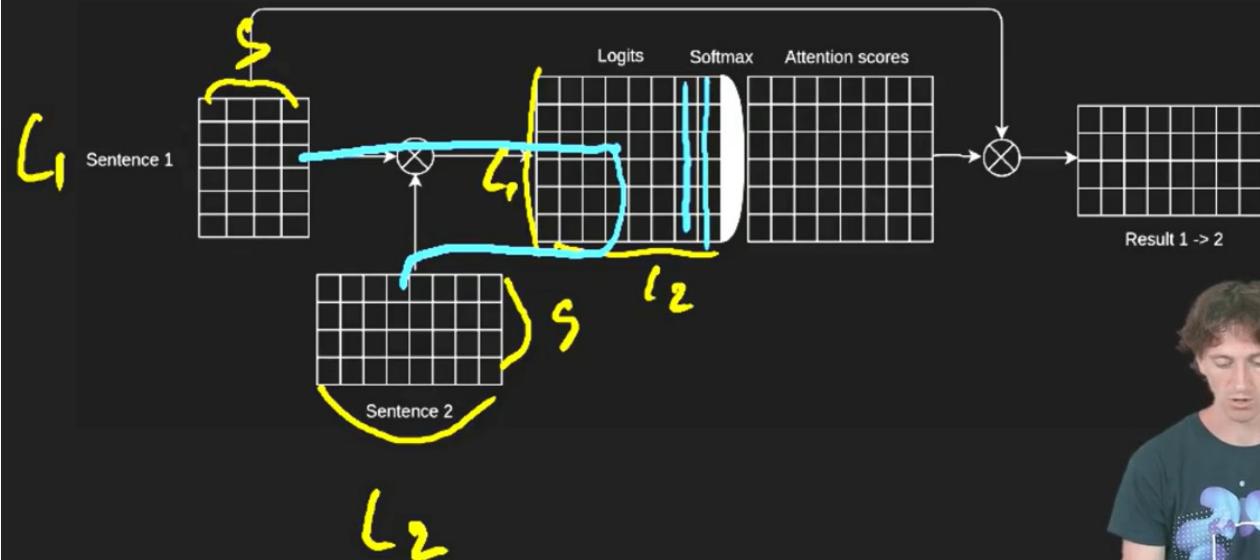
Ну и наконец, результат операции внимания считается по формуле из предыдущей задачи  $Attention[Ch] = \sum_{i=0}^{InLen-1} Input[i, Ch] \cdot AttScores[i]$ .

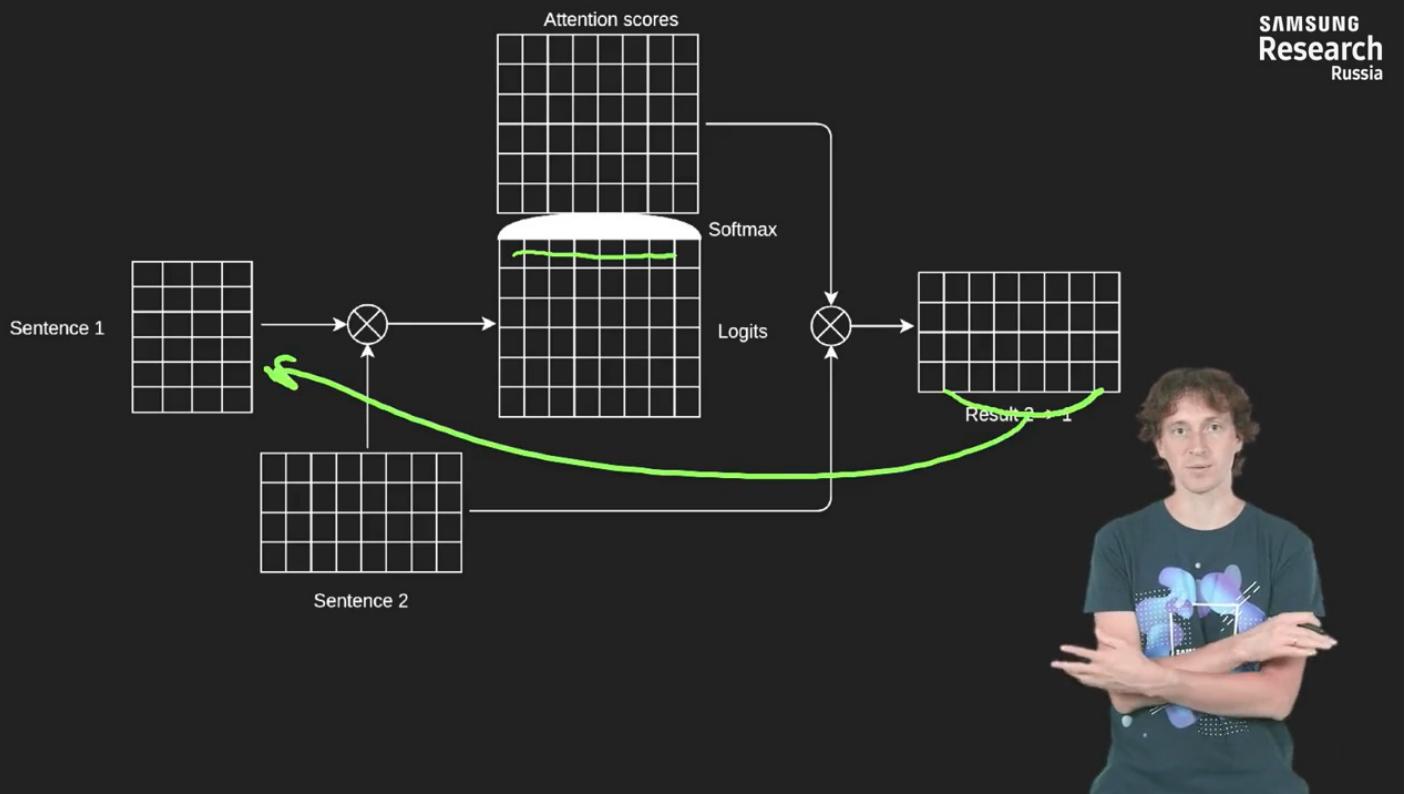
**Примените механизм внимания к входной матрице**

$$Input = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 1 & 3 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

с учётом вектора-запроса  $Query = (0 \ 0 \ 1)$ .

Внимание можно использовать для сравнения двух текстов — вот нам даны две матрицы, размерность представления у них одинаковая, а вот длины отличаются. Тогда мы можем их перемножить и получим большую матрицу размерности L1 на L2,  $i$ -й элемент этой матрицы несёт следующий физический смысл: насколько  $i$ -ое это слово из первого текста похоже на  $j$ -ое слово из второго текста. Далее у нас есть 2 варианта — два измерения, по которым можно нормализовать (то есть, к которым можно применить [софтмакс](#)). Если мы нормализуем так, как изображено на картинке, то есть по столбцам, то, в результате, получим матрицу, физический смысл которой будет следующим. Её размер будет соответствовать размеру второго предложения, а  $i$ -й столбец будет хранить представление всего первого текста относительно соответствующего слова второго текста. Такие представления могут быть крайне полезны, например, в вопросно-ответных системах, когда первое предложение обозначает вопрос, а второе — текст, в котором мы ищем ответ. Тогда, после такой агрегации по каждому вектору, в полученной матрице мы можем, например, предсказать, содержится ли в районе этого слова ответ на вопрос, или нет. Или же мы можем нормализовать матрицу [релевантности](#) по строкам — тогда получим матрицу, соответствующую размеру первого предложения, и физический смысл отобразится зеркально. Итак, в этом видео мы рассмотрели агрегацию или [пулинг](#) через усреднение, через выбор максимума, рассмотрели глобальный пулинг, а также механизм внимания — его различные варианты.





- различные варианты пулинга, агрегации
- механизм внимания - зачем, какие варианты его могут быть

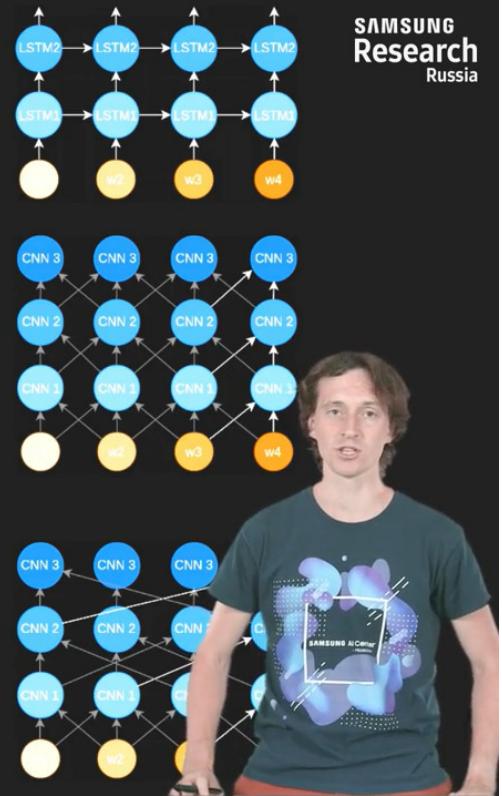


## 4.5 Трансформер и self-attention

Всем привет! В этом видео мы поговорим про одну из недавно предложенных моделей, которая очень хорошо встярхнула всю область обработки текстов, а именно — [трансформер](#) и

[механизм внутреннего внимания](#) (или self-attention). Давайте вспомним основные архитектуры нейросетей, применяемые в обработке текстов, и посмотрим на них с нового ракурса. В первую очередь — это, конечно, [рекуррентки](#), рабочая лошадка. Рекуррентки обрабатывают один токен за шаг, обновляя своё скрытое состояние. То есть, для выполнения очередного шага, необходимо полностью завершить предыдущий. Это приводит к тому, что рекуррентки работают последовательно, и возможности современных видеокарт используется не на полную мощность. Не так давно были предложены варианты рекурренток с улучшенным параллелизмом. Например, "simple recurrent unit" позволяет параллельно обсчитывать разные элементы скрытого состояния. В результате, следующий шаг можно начать считать раньше чем полностью досчитается предыдущий. Но параллелизм, по-прежнему, неполный. Некоторые важные закономерности в [естественном языке](#) носят нелокальный характер. Например, я сейчас вам про что-то рассказываю, и мне не надо после каждого слова напоминать вам тему лекции. Вы её прочитали в начале, или услышали, и, скорее всего, вы её помните — вы находитесь в контексте. Поэтому хорошая архитектура для обработки текстов должна уметь выделять далёкие закономерности, а именно — связи между словами, отстоящими друг от друга на 50, 100, или даже более слов. Чтобы рекуррентка увидела такую закономерность (от слова в начале длинного текста до слова в конце), нужно взять, и прошагать от первого слова до последнего (при этом, обработав все слова). Таким образом, для учёта закономерности длины  $N$  требуется порядка  $N$  операций. Следующий классический вид модулей — свёртки. Они очень хорошо распараллеливаются в рамках одного слоя — все токены обрабатываются независимо. Слои также могут частично считаться параллельно, так как вычисления зависят от данных только в небольшой окрестности. В отличие от рекурренток, чтобы увеличить длину учитываемых [свёрточными сетями](#) зависимостей, нужно неизбежно увеличить число параметров сети — либо нужно увеличить размер ядра, либо увеличить количество слоёв. В результате получается, что на каждом слое свёрток с непрерывными ядрами длина зависимости увеличивается примерно на размер ядра. Если мы используем [прореженные \(то есть, "dilated"\) свёртки](#), то получаем чуть более эффективную картину — у нас шаг прореживания растёт, увеличивается в два раза на каждом слое и, таким образом, чтобы учесть зависимость длины  $N$ , нам нужно порядка логарифма от  $N$  слоёв. Есть гипотеза о том, что чем больше нам нужно сделать шагов, чтобы учесть какую-то зависимость, чтобы перенести информацию из одной точки в другую, тем выше вероятность того, что мы потеряем много информации на этом пути, и, следовательно, будем хуже учитывать далёкие зависимости. И нейросетки либо медленно учатся, но компактнее (как, например, рекуррентки), либо лучше распараллеливаются, но требуют больше вычислений (как, например, свёртки).

- Рекуррентные блоки (RNN)
  - Последовательное вычисление (LSTM, GRU)
  - Улучшенный параллелизм (SimpleRU)
  - Стоимость учета зависимости длины  $n$  —  $O(n)$
- Свёрточные блоки (CNN)
  - Параллельное вычисление
  - Длина учитываемых последовательностей зависит от глубины сети —  $k$
  - Непрерывные свёртки —  $O(n/k)$
  - Прореженные (dilated) свёртки —  $O(\log_k(n))$
- Проблемы
  - Информация о далёких зависимостях теряется
  - Медленно учится (RNN)

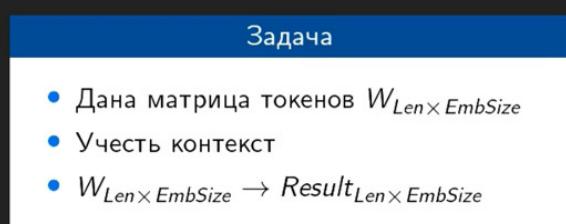
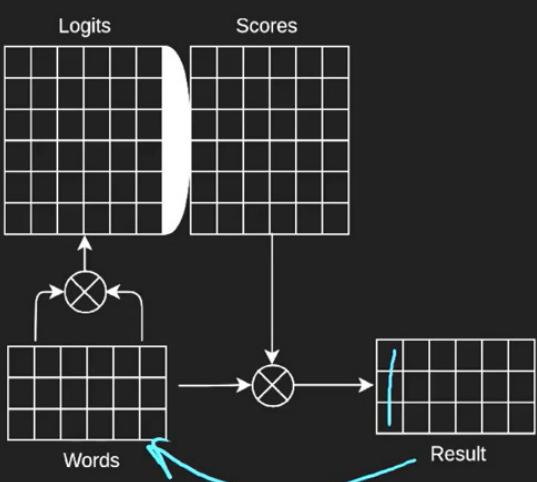
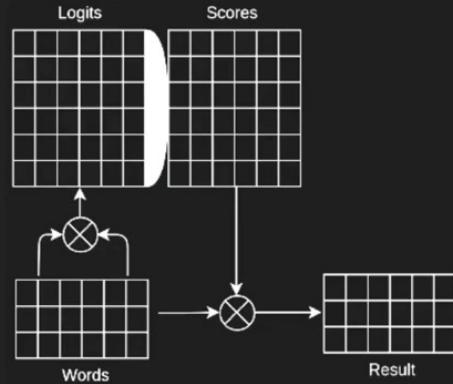
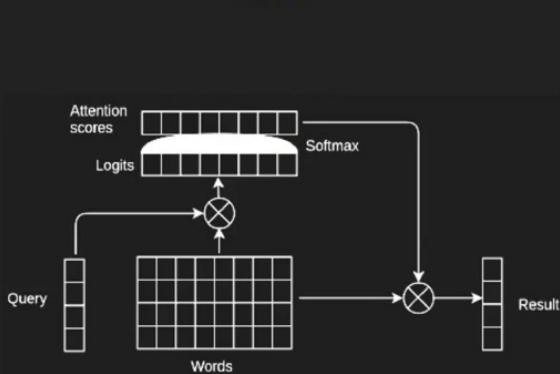


В 2017 году ребята из Google обратили внимание на механизм внимания<sup>[1]</sup>, а именно — на механизм внутреннего внимания (или "self-attention" — его ещё называют "intra attention"), который, в целом, как свёртки, только дешевле и позволяет учитывать зависимости любой длины. Механизм внутреннего внимания принимает на вход матрицу признаков токенов. Вот тут столбцы соответствуют токенам, а строки — признакам токенов. В результате применения "внимания" получается матрица такого же размера, что и входная матрица, но в ней уже каждый вектор содержит информацию о значении соответствующего токена в контексте всех остальных токенов. Алгоритм работы внутреннего внимания принципиально ничем не отличается от обычного внимания. Но особенность заключается в том, что и в качестве запросов, и в качестве ключей, используются одни и те же данные, одни и те же элементы, то есть сами токены. На первом шаге рассчитывается квадратная матрица попарного сходства каждого токена с каждым. Затем эта матрица, с помощью софтмакса, нормируется по строкам или по столбцам так, чтобы сумма весов по строке или столбцу равнялась единице. В данном случае мы нормируем по столбцам. Затем исходные векторы выступают в роли значений. Они взвешиваются с помощью полученных весов и складываются. Эту же операцию можно записать с помощью матричного произведения. Получаем такую вот итоговую формулу — это самый простой вариант внутреннего внимания, он демонстрирует основной принцип работы. На практике же используют более навороченную схему. Отличие, на самом деле, минимальное и заключается в том, что перед тем, как считать попарное сходство токенов, мы преобразовываем исходные признаки токенов, причём делаем это двумя независимыми нейросетями. Часто используется простое линейное преобразование — домножение на

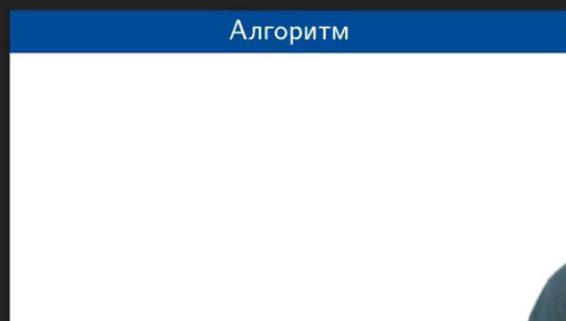
матрицу. Здесь матрицы проекции — квадратные, и длина их стороны равняется размеру эмбеддинга токена. Это даёт гораздо больше гибкости и при расчёте релевантности токен будет иметь разные признаки, в зависимости от того, выступает он в качестве запроса или в качестве ключа. Такое преобразование можно понимать как применение одномерной свёртки с ядром размера 1. Тут не нарисовано, но аналогично может преобразовываться и результат. Итоговая формула выглядит практически так же, как и в самом простом случае — здесь только добавились матрицы проекций.

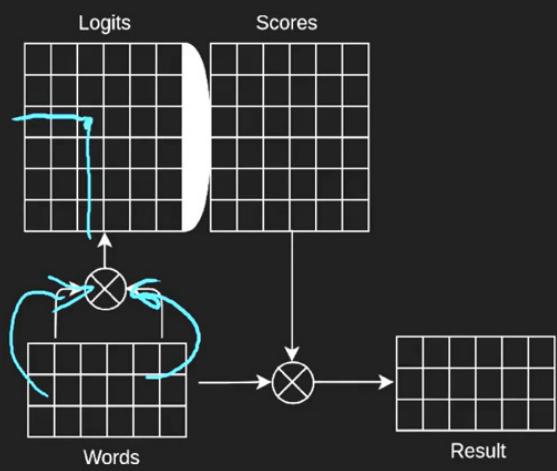
[1] Vaswani A. et al. Attention is all you need //Advances in neural information processing systems. – 2017. – С. 5998-6008. (<https://arxiv.org/abs/1706.03762>)

- Механизм внимания
- Механизм внутреннего внимания self-attention (intra-attention)



SAMSUNG  
Research  
Russia





### Задача

- Данна матрица токенов  $W_{Len \times EmbSize}$
- Учесть контекст
- $W_{Len \times EmbSize} \rightarrow Result_{Len \times EmbSize}$

### Алгоритм

- ➊ Сходство токенов - каждый с каждым  
 $Logits = W^T \cdot W$



image.png

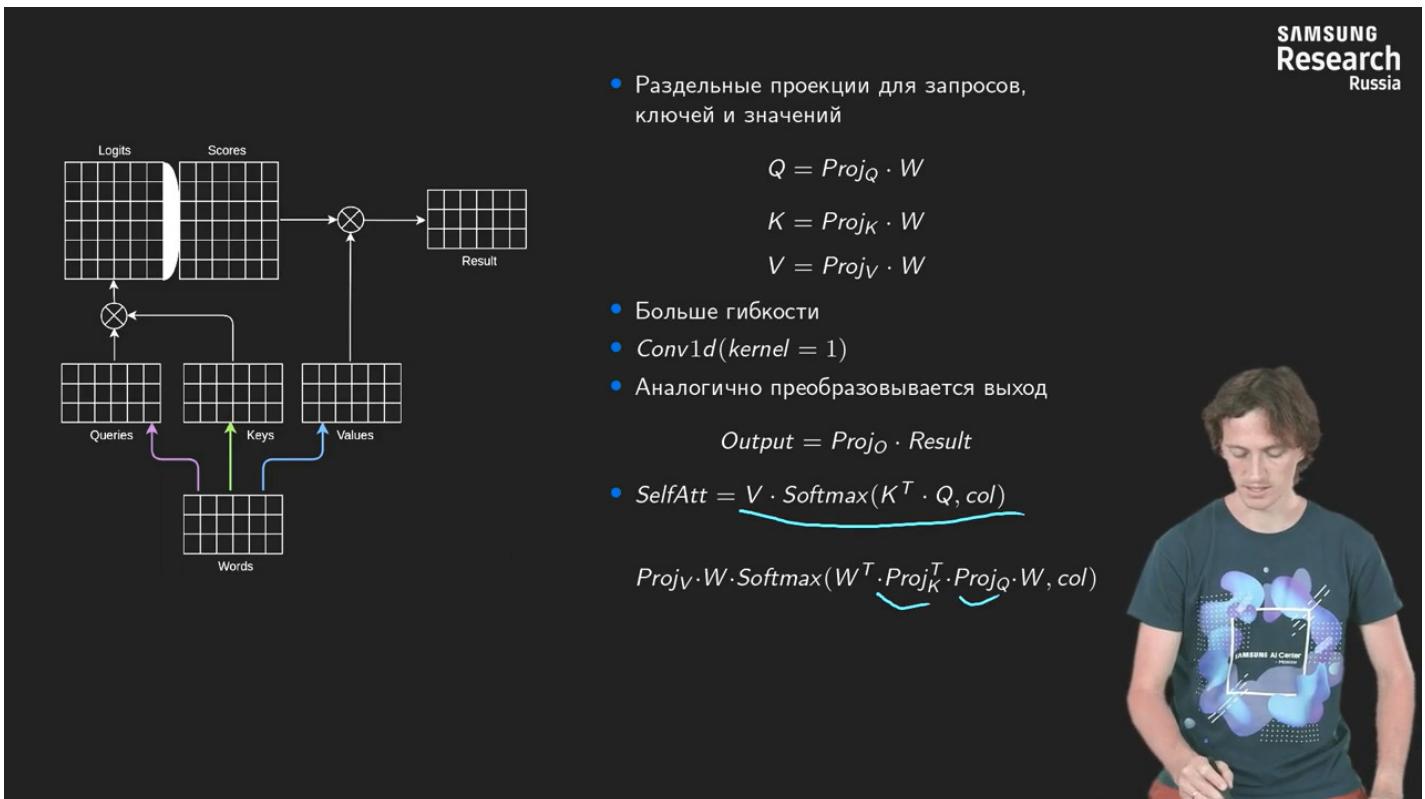
443 kB

- Раздельные проекции для запросов, ключей и значений

$$Q = \underbrace{Proj_Q}_{W} W$$

$$K = \underbrace{Proj_K}_{W} W$$

$$V = \underbrace{Proj_V}_{W} W$$



Из комментариев:

Вопрос:

А вот интересно. Если на входе не embedding , А матрица векторов полученная кодированием токена в one\_hot\_encoding( единичный)

1 0 0 0 0

0 0 1 0 0

0 1 0 0 0

Или вообще wordbag

1 1 1 0 0

Эта штука работать будет?

Ответ (Романа Суворова):

будет, но будет есть очень много памяти. На самом деле, Embedding - это эффективный (по количеству операций) способ реализовать линейный слой для one-hot векторов.

Допустим, некоторое слово изначально (на входе) представляется каким-то вектором  $x$ , как обычно мы его сразу хотим преобразовать:  $Wx$  (матричное произведение матрицы весов  $W$  и вектор-столбца  $x$ ). А теперь представим, что  $x$  - это one hot вектор, то есть в нём все элементы нули, кроме  $i$ -го элемента, который равен 1. Тогда результат этого матричного произведения будет равен  $i$ -й строке матрицы  $W$ . Тогда если мы заранее знаем, что  $x$  - это one hot, тогда почему бы просто не выбирать строчки из таблицы вместо дорогостоящего перемножения? а для эффективной реализации wordbag в pytorch тоже есть embedding-слой - [EmbeddingBag](#).

Из комментариев к практическому заданию:

Вопрос:

Какой физический смысл умножения матрицы на себя же транспонированную? Как можно в нашем случае интерпретировать получившуюся матрицу?

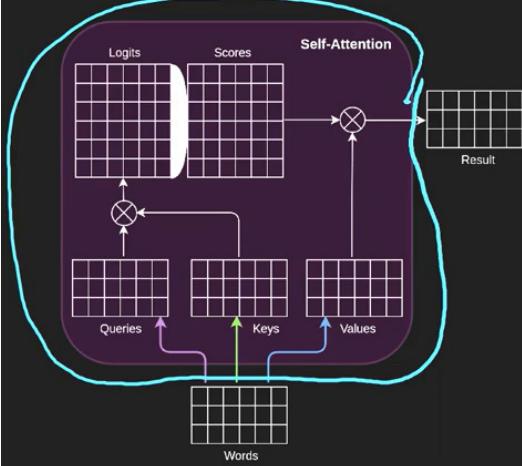
Ответ (Романа Суворова):

Давайте вспомним, как выполняется матричное произведение. Допустим, у нас есть две матрицы  $A_{n,m}$  и  $B_{m,k}$ , тогда результат их произведения - матрица  $A_{n,m} \cdot B_{m,k} = C_{n,k}$ , в которой каждый элемент равен скалярному произведению соответствующей строки матрицы  $A_{n,m}$  и столбца матрицы  $B_{m,k}$ :  $c_{ij} = \sum_{q=1}^m a_{iq} b_{qj}$ . Физический смысл скалярного произведения - оценка сходства векторов (если вектора нормированы на их L2-длину, то их скалярное произведение - это косинус угла между ними).

Если мы перемножаем матрицу  $A_{n,m}$  на себя саму транспонированную (как в этой задаче), то мы получаем квадратную матрицу  $A_{n,m} \cdot A_{n,m}^T = C_{n,n}$ , содержащую оценки сходства для каждой пары строк матрицы  $A_{n,m}$  (матрицу попарного сходства векторов в исходном наборе векторов  $A_{n,m}$ ). Такая матрица ещё называется [матрицей Грама](#), у неё достаточно много разных применений. В контексте механизма внимания смысл матрицы *Logits* - сходство каждого слова (или элемента последовательности) со всеми остальными словами.

Об этом говорится [и в лекции, начиная примерно с 0:54](#).

Итак, вот эта часть на схеме — это и есть [механизм внутреннего внимания](#). Это основной строительный блок [трансформера](#) и многих других современных архитектур для обработки текстов. Однако механизм внутреннего внимания, сам по себе, обладает рядом недостатков. Корень этих недостатков заключается в том, что операция усреднения, пусть даже взвешенного — это достаточно грубая операция и мы теряем много информации. Кроме того, если какой-то токен получил большой вес, то все остальные сразу получили вес меньше — даже если в них есть полезная информация. И такой механизм внимания позволяет учесть только один аспект, он измеряет сходство токенов друг с другом только один раз, хотя, казалось бы, они могут быть похожи множеством разных способов. Короче говоря, механизм внутреннего внимания теряет информацию и усложняет сходимость и решение задачи, если он используется как единственное выразительное средство в нейросети, в чистом виде.



- Раздельные проекции для запросов, ключей и значений

$$Q = Proj_Q \cdot W$$

$$K = Proj_K \cdot W$$

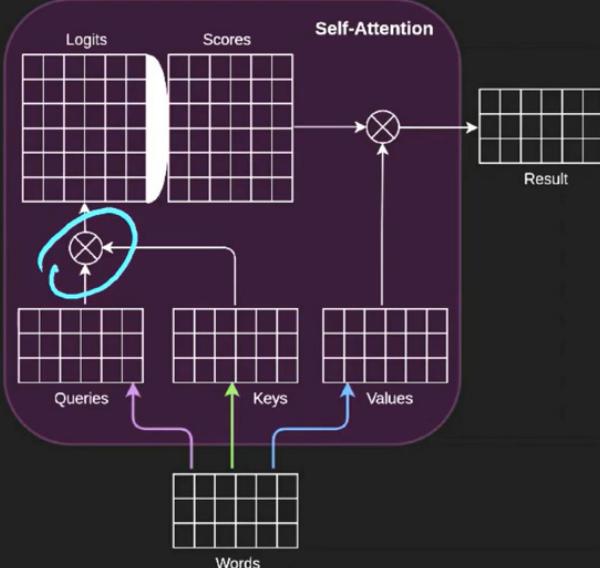
$$V = Proj_V \cdot W$$

- Больше гибкости
- $Conv1d(kernel = 1)$
- Аналогично преобразовывается выход

$$Output = Proj_O \cdot Result$$

- $SelfAtt = V \cdot \text{Softmax}(K^T \cdot Q, col)$

$$Proj_V \cdot W \cdot \text{Softmax}(W^T \cdot Proj_K^T \cdot Proj_Q \cdot W, col)$$



### Недостатки Self-Attention

- Потеря информации
- Только один аспект



Из практического задания:

Давайте теперь посмотрим, как работает более общий вариант self-attention - когда в качестве ключей, запросов и значений используются разные матрицы.

На вход мы получаем всё ту же матрицу

$$Input = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}$$

Общий алгоритм состоит из следующих основных шагов:

1. Найти значения ключей, запросов и значений, используя линейное преобразование

$$Keys = Input \cdot Proj_K + Bias_K$$

$$Queries = Input \cdot Proj_Q + Bias_Q$$

$$Values = Input \cdot Proj_V + Bias_V$$

2. Найти матрицу попарного сходства, используя полученные матричное произведение запросов и ключей

$$Logits = Queries \cdot Keys^T$$

3. Найти коэффициенты усреднения, нормировав матрицу попарного сходства с помощью softmax по строкам

$$AttScores = softmax(Logits, rows)$$

4. Найти результат с помощью матричного произведения матриц значений и коэффициентов

$$Result = AttScores \cdot Values$$

Вам требуется найти значение матрицы  $Result \in \mathbb{R}^{InLen \times EmbSize}$  с учётом следующих параметров преобразования:

$$Proj_K = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

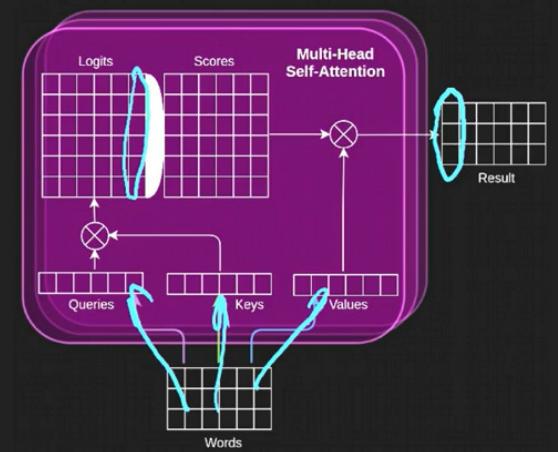
$$Proj_Q = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

$$Proj_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$Bias_K = Bias_Q = Bias_V = (0 \quad 0)$$

Авторы [трансформера](#) это заметили и предложили модифицировать механизм внимания — а именно, предложили ввести несколько "голов".<sup>[1]</sup> Головы — это точно такие же механизмы [внутреннего внимания](#), но в разных головах для получения проекций используются разные веса. И нормирование выполняется независимо в каждой голове. Это позволяет строить результирующий [вектор признаков](#), учитывая сразу множество аспектов, а не только один (как если бы мы использовали обычный механизм внутреннего внимания с одной "головой"). Каждая "голова" работает с пространством признаков меньшего размера. Потом, на выходе, они конкатенируются — так, что размер выходной матрицы остаётся прежним — таким же, каким и был на входе. Именно такой вариант внутреннего внимания и используется в трансформере (и в большинстве современных архитектур).

[1] Attention and its Different Forms (<https://towardsdatascience.com/attention-and-its-different-forms-7fc3674d14dc>)

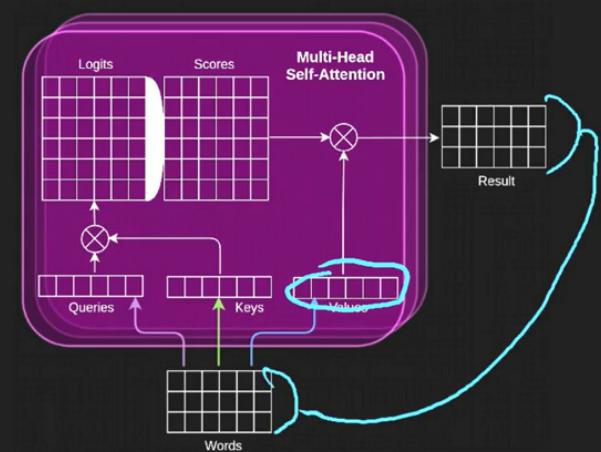


### Недостатки Self-Attention

- Потеря информации
- Только один аспект

### Несколько "голов"

- Различные веса



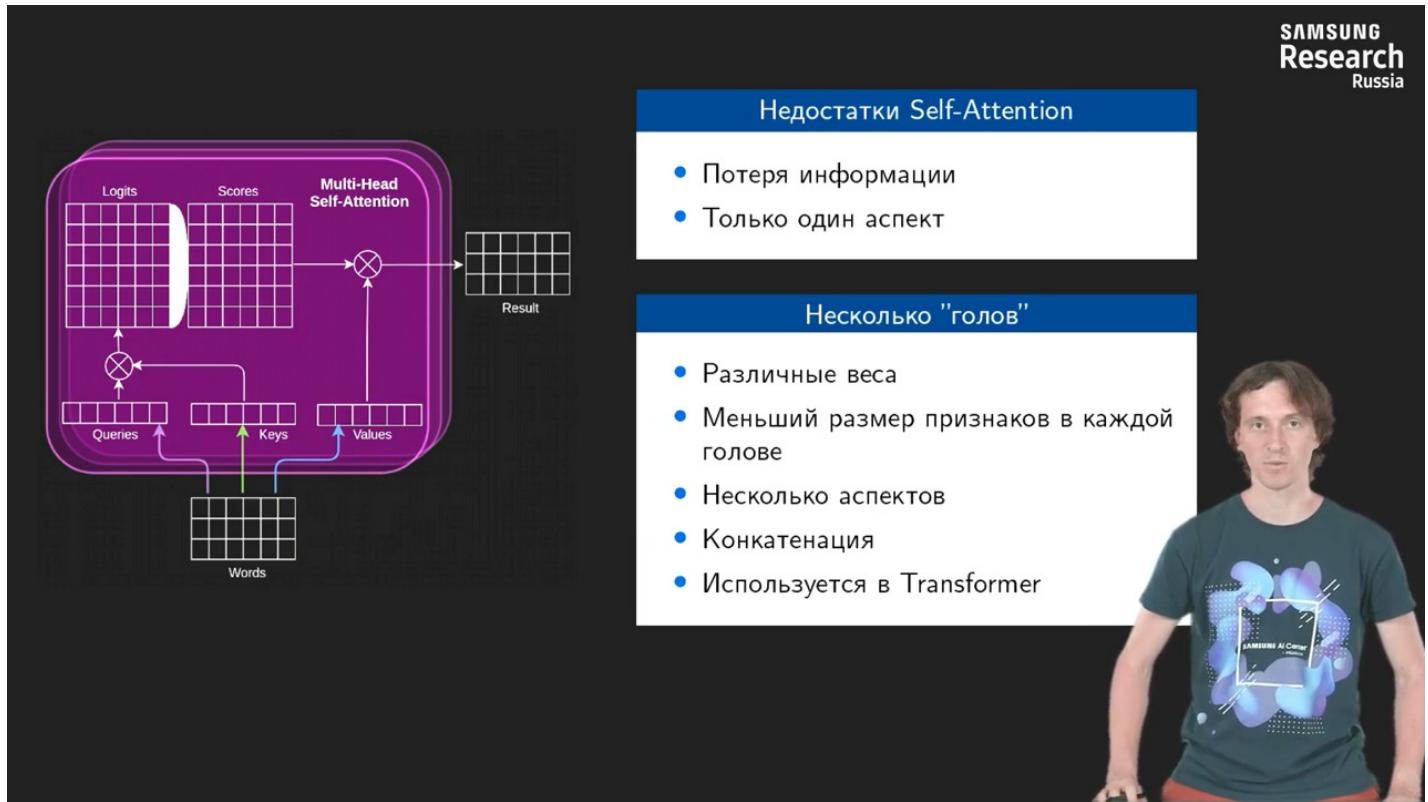
### Недостатки Self-Attention

- Потеря информации
- Только один аспект

### Несколько "голов"

- Различные веса
- Меньший размер признаков в каждой голове





Из комментариев:

Вопрос:

Я правильно понимаю, что нет смысла брать количество голов большим, чем EmbSize?

Ответ (романа Суворова):

это и не получится, потому что эффективный размер векторов внутри каждой головы - EmbSize / HeadsN. В целом о количестве голов надо думать в духе "в каком количестве разных аспектов я хочу сравнивать слова друг с другом?". Даже в больших моделях голов не очень много, 6-12.

Из практического задания:

Ещё один шаг - перейдём от простого self-attention к multihead self-attention.

Общий алгоритм - точно такой же, как и в предыдущей задаче. Отличие в том, что нам нужно несколько раз применить механизм внимания с разными параметрами преобразований  $Result^i = Self\ Attention(Input, Proj_K^i, Proj_Q^i, Proj_V^i)$ ,  $Result^i \in \mathbb{R}^{InLen \times \frac{EmbSize}{HeadsN}}$ .

Результат  $MHResult \in \mathbb{R}^{InLen \times EmbSize}$  получается конкатенацией  $Result^i$  по столбцам:  $MHResult = [Result^1, Result^2, \dots, Result^{HeadsN}]$ .

Вам требуется найти  $MHResult$  для входных данных

$$Input = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}$$

с учётом количества "голов"  $HeadsN = 2$  и параметров преобразований

$$Proj_K^1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, Proj_K^2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix},$$

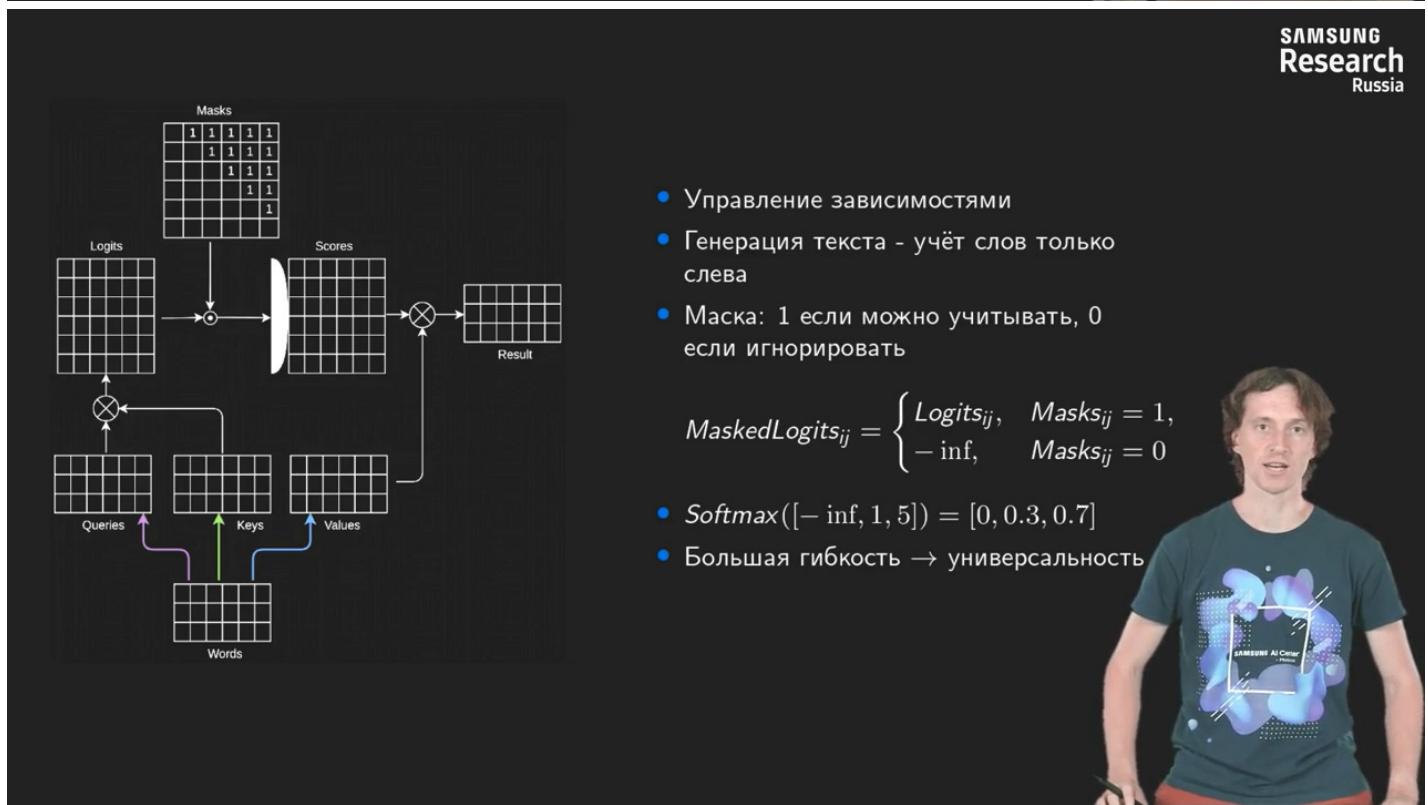
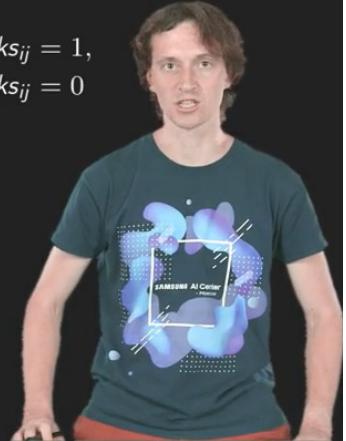
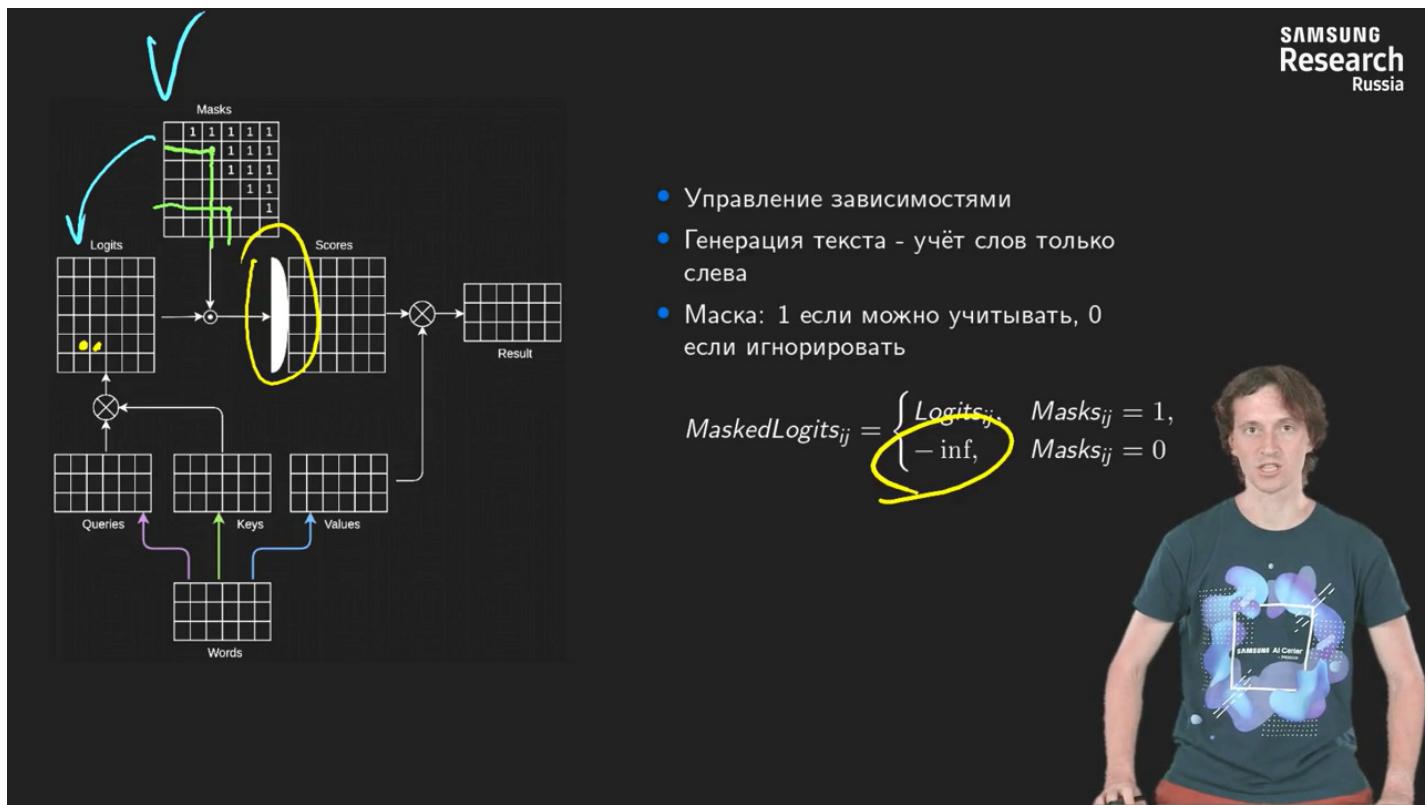
$$Proj_Q^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Proj_Q^2 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

$$Proj_V^1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, Proj_V^2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$Bias_K^i = Bias_Q^i = (0 \ 0), \quad Bias_V^i = 0$$

Результат запишите в виде матрицы, на одной строке - элементы одной строки матрицы, разделённые пробелами. В качестве десятичного разделителя используйте точку. Ответ округлите до не менее чем двух знаков после запятой.

Механизм [внутреннего внимания](#) также предоставляет очень важную возможность, а именно — возможность управления зависимостями. Например — допустим, что мы делаем модель для генерации текста и хотим предсказывать следующее слово на основе предыдущих и чтобы остальные слова, которые есть на входе сети, вообще никак не учитывались. Например, если мы генерируем третье слово, то мы хотим чтобы учитывались только первое и второе. В механизме внимания для работы с подобными ситуациями используется маски. Маска — это матрица такой же размерности, что и маска "сходство токенов", то есть — квадратная, со стороной, равной длине входной последовательности. На  $i$ -ой/ $j$ -ой позиции в этой матрице ставится единичка, если для  $j$ -го токена надо учитывать  $i$ -ый токен, и нолик — если не нужно учитывать. Маска применяется перед нормализацией матрицы сходства — то есть, перед [софтмаксом](#). Маскирование заключается в том, что мы затираем некоторые значения матрицы сходства и ставим в эти места "минус бесконечность". Применение софтмакса к матрице с "минус бесконечностями" приводит к тому, что на их месте появляются нули. Механизм маскирования делает архитектуру с внутренним вниманием очень гибкой и практически универсальной. Маски можно менять динамически, для каждого слова можно формировать маску независимо от других, и так далее. Этим свойством [трансформера](#) пользуются авторы некоторых крутых архитектур, которые сейчас показывают самое лучшее качество решения многих задач.



Из комментариев:

Вопрос:

Можно ли рассматривать маски как аналог дропаута в обычных нейросетях?

Ответ (Николая Копырина):

интересная мысль, как и любая аналогия. Разница будет в том, что мы не контролируем, какие нейроны выключаются в дропауте, и нем полезно что они выключаются в случайной

комбинации, а тут мы пытаемся выключить из рассмотрения некоторые шаблоны намеренно. Не пытаемся внести неопределённость в работу системы, а наоборот -- упорядочиваем таким образом датасет.

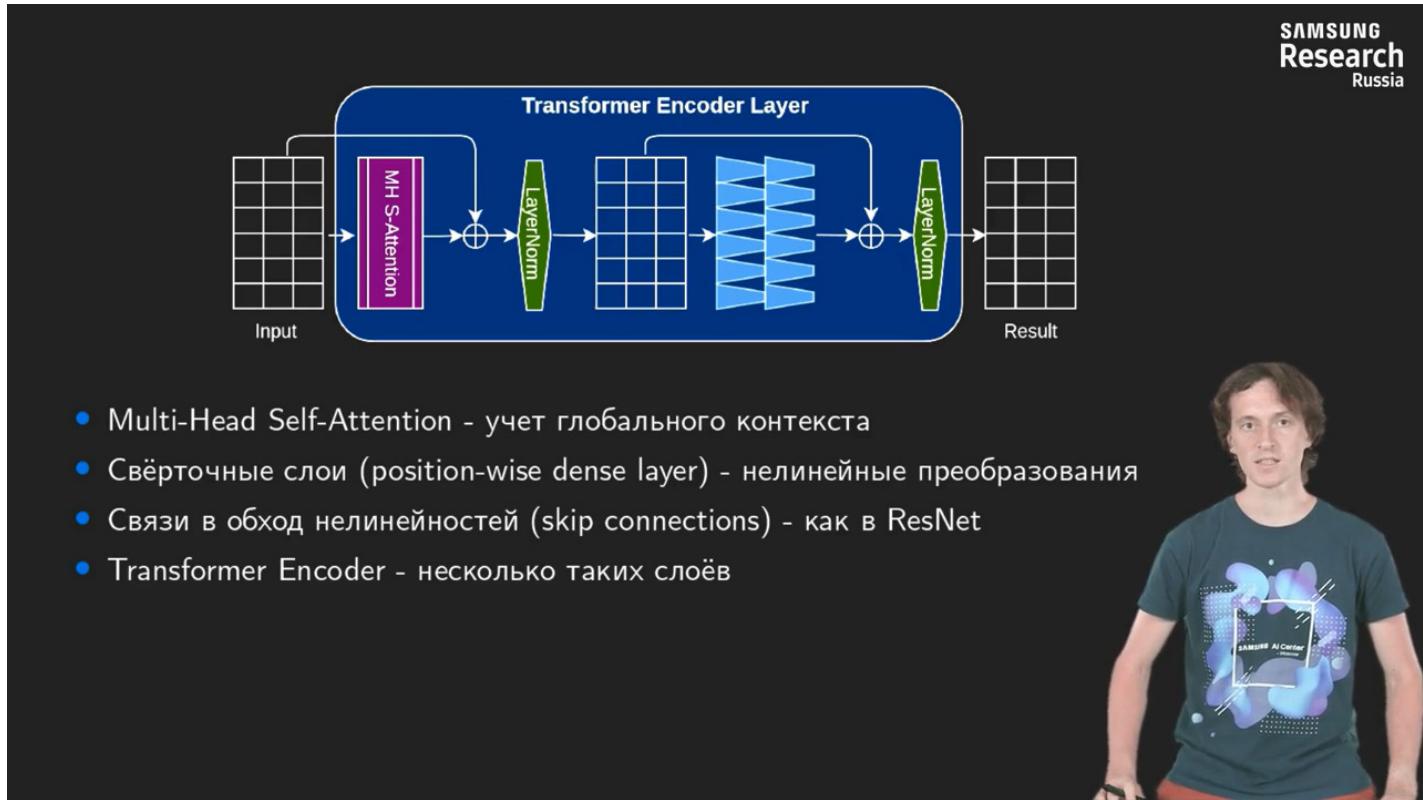
Внутреннее внимание с несколькими головами<sup>[1]</sup> — это ключевая идея [трансформера](#), но он состоит не только из этих блоков. Он состоит из нескольких слоёв следующего вида: сначала используется внимание для учёта глобального контекста, затем — признаки каждого токена независимо преобразовываются с помощью двухслойной нейросети. Ко всем токенам сеть применяется с одними и теми же параметрами. По смыслу это похоже на одномерные свёртки с размером ядра "1". Ещё тут есть связи в обход нелинейностей, как в [ResNet](#) для классификации изображений — это ускоряет процесс обучения за счёт лучшего протекания градиентов. Ну, и таких слоёв становится несколько, один поверх другого. Эти слои не меняют размерность данных. Наверное, вы уже догадываетесь, что здесь что-то не так. Рекуррентки обрабатывают текст слева направо, свёртки сравнивают каждое слово со словами слева и справа, а механизм внимания вообще ничего не знает про позиции слов в тексте: он не знает, какое слово рядом с каким стоит. Это же важно... Особенно это важно для таких языков как английский, да и для всех остальных тоже. Сам по себе, механизм внимания безразличен к позиции. Но нам важно учитывать и относительное расположение слов. Ну так давайте, запихнём эту информацию прямо в признаки! За это отвечает так называемое позиционное кодирование.<sup>[2]</sup> Код позиции — это вектор такого же размера что и [эмбеддинг](#) токена и чтобы получить признаки токена мы просто складываем эмбеддинг токена embedding позиции. В принципе, коды позиций можно хранить в таблице эмбеддингов — так же, как и эмбеддинги токенов. Но в ряде случаев это неудобно — например, если мы обучались на примерах размера не более 10, то у нас просто нет в этой таблице эмбеддингов вектора для одиннадцатой позиции и тексты такой длины мы уже не можем обрабатывать. Ну... и не проблема, давайте тогда просто сгенерируем какой-нибудь периодический сигнал. Авторы, например, предлагают использовать синусоиды и косинусоиды разных частот — это чем-то напоминает базис Фурье. На практике эти два подхода — обучаемые коды и периодически сигнал — работают примерно одинаково, но сигнал гораздо удобнее использовать. Вот так выглядит общая архитектура трансформера, как её нарисовали авторы. Они изначально решали задачу машинного перевода, то есть генерации предложения по предложению на другом языке. Поэтому их архитектура состоит из энкодера (мы его уже рассматривали) и декодера. Декодер отличается тем, что в нём используются маски. А ещё, в него подаются выходы энкодера. Но, в остальном, это практически одна и та же нейросеть (но параметры, конечно же, у них разные). И в энкодере, и в декодере, используется много слоёв вот такого типа — в оригинальной статье по 6, а потом стали делать ещё больше — по 12, по 24... Такая архитектура позволяет решать задачи, связанные с анализом последовательностей (не только с

текстами), часто — лучше, чем это получается делать с помощью рекурренток, и, при этом, можно тратить меньше времени на обучение.

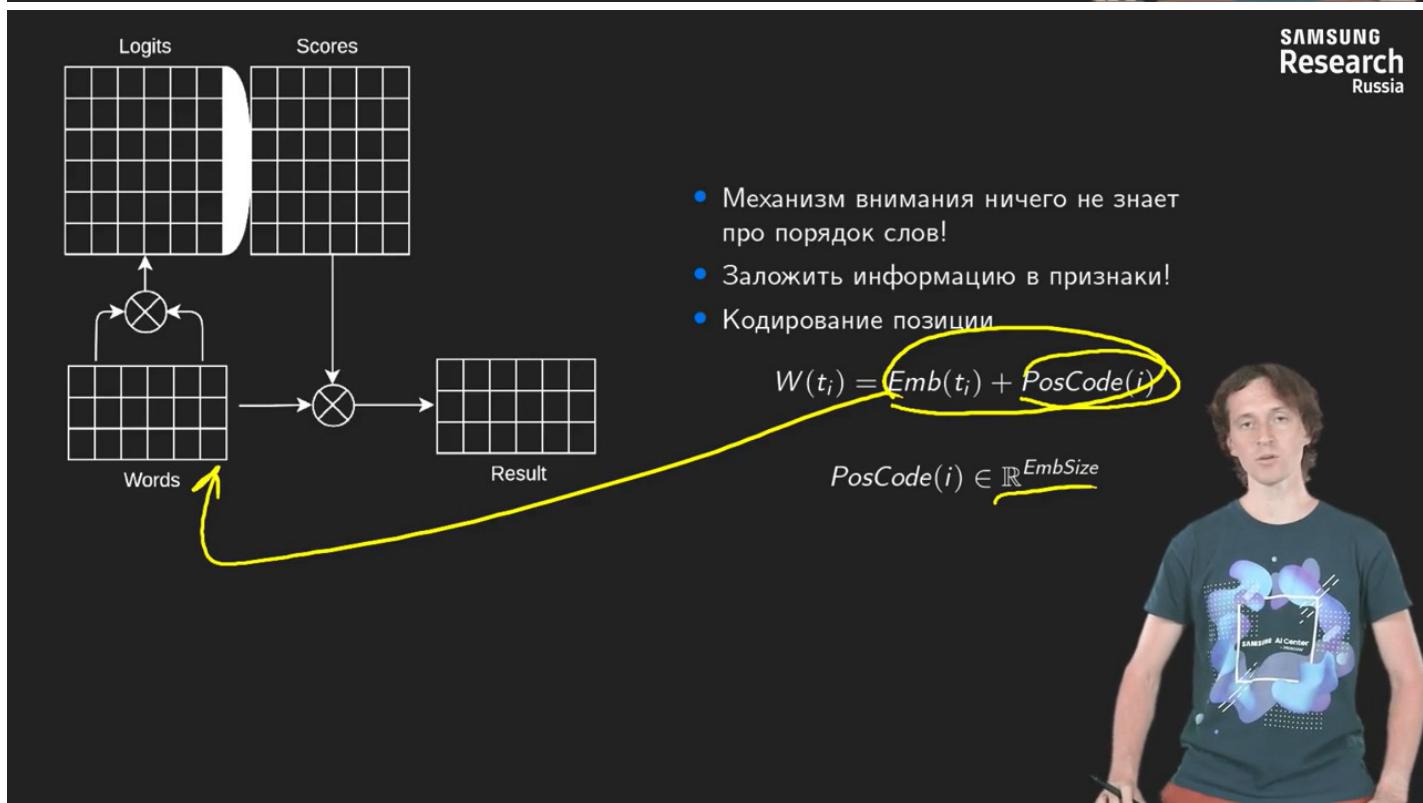
[1] Attention and its Different Forms (<https://towardsdatascience.com/attention-and-its-different-forms-7fc3674d14dc>)

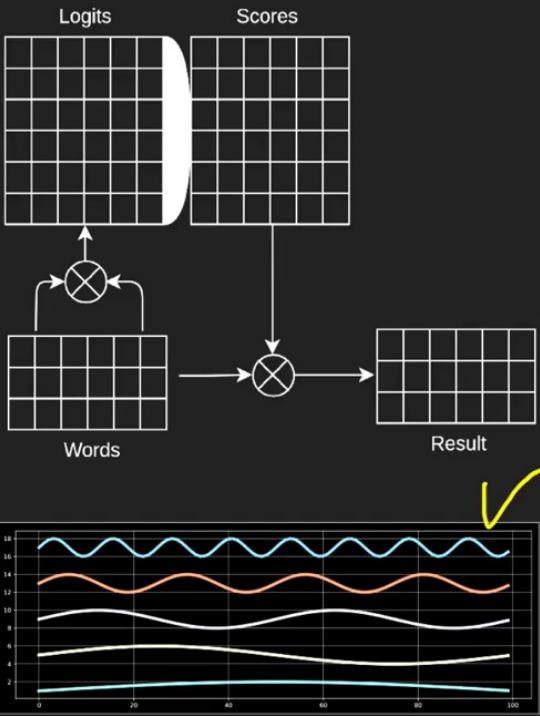
[2] Transformer Architecture: The Positional Encoding

([https://kazemnejad.com/blog/transformer\\_architecture\\_positional\\_encoding/](https://kazemnejad.com/blog/transformer_architecture_positional_encoding/))



- Multi-Head Self-Attention – учет глобального контекста
- Свёрточные слои (position-wise dense layer) - нелинейные преобразования
- Связи в обход нелинейностей (skip connections) - как в ResNet
- Transformer Encoder - несколько таких слоёв



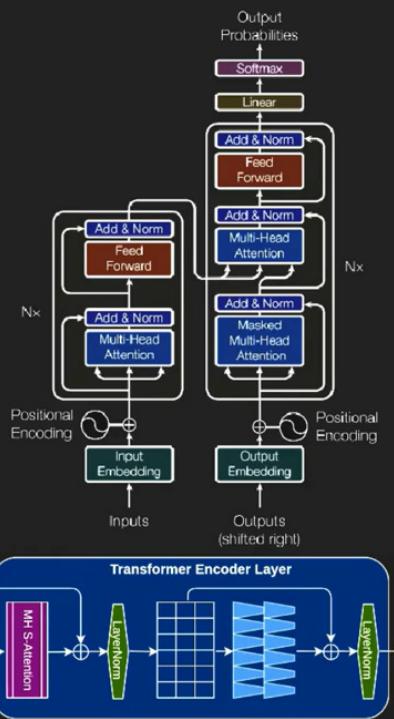


- Механизм внимания ничего не знает про порядок слов!
- Заложить информацию в признаки!
- Кодирование позиции

$$W(t_i) = Emb(t_i) + PosCode(i)$$

$$PosCode(i) \in \mathbb{R}^{EmbSize}$$

- Синусоидальный сигнал
- Обучаемые коды



- Задача машинного перевода
- Энкодер и декодер
- В декодере используются маски
- Улучшение качества и снижение времени обучения



Из комментариев:

Вопрос:

Я не совсем понял данный момент: "В принципе, коды позиций можно хранить в таблице эмбеддингов — так же, как и эмбеддинги токенов. Но в ряде случаев это неудобно — например, если мы обучались на примерах размера не более 10, то у нас просто нет в этой таблице эмбеддингов вектора для одиннадцатой позиции и тексты такой длины мы уже не

можем обрабатывать. Ну... и не проблема, давайте тогда просто сгенерируем какой-нибудь периодический сигнал. Авторы, например, предлагают использовать синусоиды и косинусоиды разных частот — это чем-то напоминает базис Фурье. На практике эти два подхода — обучаемые коды и периодически сигнал — работают примерно одинаково, но сигнал гораздо удобнее использовать"

1. На размерах не более 10. Размер чего?

2. При чём тут 11 позиция? Или имеется слово на 11 позиции и для неё нет вектора эмбеддинга? такого ведь не может быть или это про вектор контекста.. ничего не понимаю

3. Давате просто сгенерируем сигнал... во что сгенерировать, куда это вставить?) Совсем не понял идею с сигналом

Можете пожалуйста прояснить эту ситуацию? может как то по другому перефразировать.

Спасибо большое!

Ответ (от студента):

здесь речь о том, что для того чтобы учитывать взаимное расположение токенов, приходится искусственно создавать эмбеддинги позиций, которые добавляются к эмбеддингам токенов. В данном примере 10 и 11 это длины последовательностей в токенах. Если просто кодировать номер токена в последовательности, то если мы обучались на последовательностях до 10 токенов, то на тесте мы уже не сможем обработать последовательности в 11 токенов и более. Поэтому на практике используют другой подход, который и описывается в лекции.

Используется суперпозиция нескольких синусоид с разной частотой и начальной фазой. Тогда для каждого токена можно сгенерировать свой вектор позиции. Размерность векторов позиции такая же, как у векторов эмбеддингов токенов. Затем эмбеддинги токенов и эмбеддинги позиций просто складывают, так получаются окончательные эмбеддинги токенов с учётом их позиций. [Вот здесь](#) это описывается довольно подробно.

Мы поговорили про [механизм внутреннего внимания](#) или self attention. Это краеугольный камень, вокруг которого сейчас строится много хороших решений сложных задач. Он так хорош, например, потому, что позволяет учитывать глобальные зависимости (очень далёкие), связывать слова, расположенные очень далеко друг от друга. А также процесс его обучения очень хорошо распараллеливается. В нём также можно динамически управлять зависимостями между элементами входных данных за счёт масок в механизме внимания. Ещё мы вкратце поговорили про архитектуру "[трансформер](#)", построенную практически целиком из механизма внимания.<sup>[1]</sup> И надо помнить, что мы говорили о самых базовых ключевых принципах, кроме них есть ещё очень много интересных и важных нюансов, которые лучше изучать, читая исходные коды работающих моделей.

[1] The Illustrated Transformer (<http://jalammar.github.io/illustrated-transformer/>)

- Механизм внутреннего внимания (self-attention)
- Глобальные зависимости
- Эффективные параллельные вычисления
- Управление зависимостями - маски
- Архитектура Transformer
- Базовые принципы, нюансы - в конкретных реализациях



Из комментариев:

Вопрос:

В чем отличие нелинейного преобразования от линейного?

Ответ (Романа Суворова):

примеры линейного преобразования - умножение матрицы на матрицу, матрицы на вектор, сложение векторов. Примеры нелинейного преобразования - возведение в степень, ReLU (да и вообще, все функции активации), тригонометрические функции.

Преобразование  $f$  является линейным, если для него выполняются равенства  $f(a + b) = f(a) + f(b)$  и  $f(a * b) = a * f(b)$

Зачем нужны нелинейности?

Допустим, у нас есть какой-то вектор признаков  $x$  и мы применяем к нему один слой нейросети:  $y = Ax + b$ . Если мы хотим сделать модель более выразительной, мы можем попробовать добавить ещё один слой:  $y_2 = W y + q$ . Тут  $A$  и  $W$  - матрицы весов, а  $b$  и  $q$  - вектора. Однако так как умножение на матрицу и прибавление вектора - операции линейные, то мы можем найти такие новые матрицу  $R$  и вектор  $p$ , что  $y_2 = R x + p$ . Другими словами, применяя несколько линейных преобразований одно поверх другого, мы не усиливаем итоговое преобразование.

Чтобы эту проблему решить, между линейными преобразованиями добавляются нелинейные преобразования -  $y_2 = \text{Layer2}(\text{ReLU}(\text{Layer1}(x)))$  - такое преобразование уже нельзя представить одним линейным слоем.

Более формальное определение линейных преобразований лучше почитать в литературе, можно начать с [вики](#).

У меня получилось ответить на Ваш вопрос?