

Механизм внимания

Цель занятия

В результате обучения на этой неделе вы узнаете:

- о механизме внимания — технике, которая лежит в основе наиболее популярных современных подходов;
- особенностях модели глубокого обучения Seq2Seq и как ее используют в задаче машинного обучения;
- Self-Attention — подходе, который позволил отказаться от RNN в задаче машинного обучения;
- Multi-Head Attention — специальном новом слое, который позволяет запускать нескольких параллельных потоков Self-Attention с различными весовыми коэффициентами.

План занятия

1. [Механизм внимания Attention. Обзор](#)
2. [Механизм внимания в математической форме](#)
3. [Механизм внимания Self Attention](#)
4. [Механизм внимания Multi-Head Attention](#)
5. [Реализация механизма внимания на Python](#)

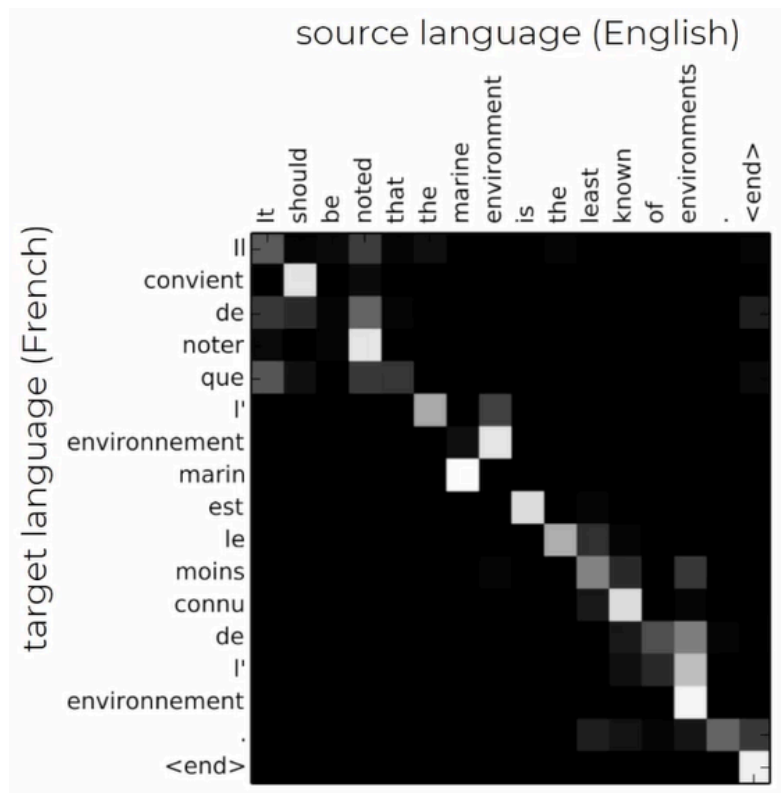
Конспект занятия

1. Механизм внимания Attention. Обзор

Механизм внимания Attention лежит в основе наиболее популярных подходов — архитектуры Трансформер и всех ее производных (BERT, GPT и т. д.). По сути, Трансформер — это несколько блоков Self Attention, которые перемежаются полносвязными слоями.

Word alignment

Word alignment — матрица соответствия слов для двух языков.

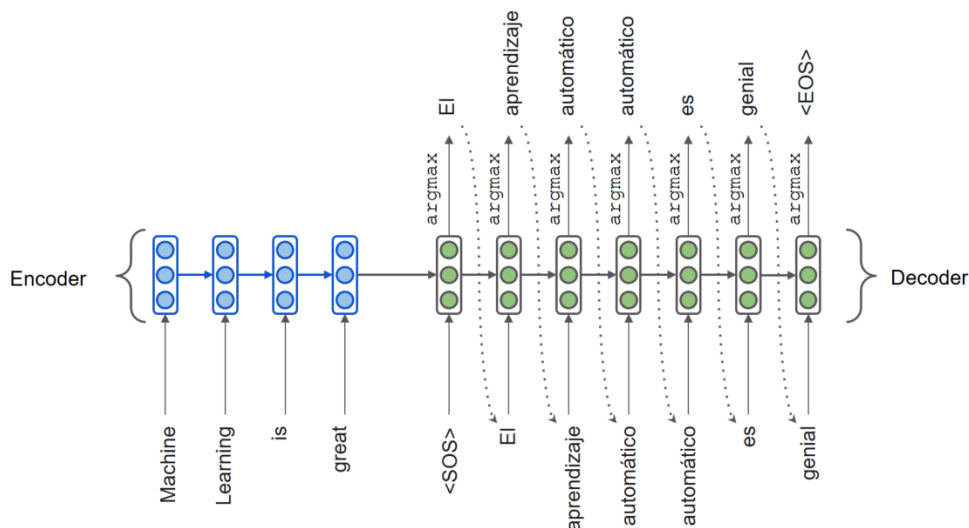


Из такой матрицы видно, какие в какие слова переходят при переводе. Есть отличия от матриц Word alignment из статистического машинного перевода. Приведенную матрицу составляли не люди, она сгенерирована моделью машинного перевода автоматически и самостоятельно на основании правильных векторов. Модели никто не «говорил», какие в какие слова переводить, она сама это поняла.

Данная матрица — визуализация механизма внимания. Механизм внимания позволяет автоматически получать Word alignment Matrix, и это неплохо помогает интерпретации.

Seq2Seq NMT

Рассмотрим классическую структуру Seq2Seq перевода:



Энкодер берет на вход исходное предложение и шаг за шагом его обрабатывает, пока не получит некоторый вектор, который будет кодировать все исходное предложение и сохранять информацию обо всем левом контексте. Именно его мы будем использовать в качестве некоторого эмбединга для всего исходного текста. Затем передадим этот вектор в декодер, где на этапе декодирования будем генерировать целевую фразу.

Заметим, что на этапе обучения Seq2Seq модели мы можем воспользоваться некоторой хитростью — *teacher enforcing*, то есть подачей на вход истинной последовательности вместо сгенерированной самой моделью.

Проведем мысленный эксперимент. Предположим, что мы только начали учить нашу модель. Она пока плохо умеет генерировать текст и ничего не знает, поэтому наверняка на первом шаге сгенерирует неправильное слово, то есть не то, которое должно быть. Если на следующем шаге мы подадим на вход сгенерированное моделью слово, она будет опираться уже на ошибочный input, что приведет к некорректному переводу. Из-за этого модель не будет учиться.

Решить эту проблему достаточно просто. В начале обучения мы можем чаще подавать «истинные» слова на k -й позиции, таким образом постепенно снижать вероятность подачи «истинного» слова по ходу обучения.

На каждом шаге генерации мы, грубо говоря, бросаем монетку: подавать на вход слово, сгенерированное нейронной сетью, или то, которое на самом деле стояло на k -й позиции в исходном тексте. Тем самым мы даем возможность модели избежать собственных ошибок.

Есть некоторая проблема в том, что один вектор энкодера кодирует всю информацию из исходного текста — непонятно, хватит ли вектора для этого фиксированного размера. Логично предположить, что с какой-то длины последовательности качество перевода деградирует. А значит, до декодера будет доходить не вся информация, и он будет генерировать ответ из неполных данных.

Получаем «бутылочное горлышко» в виде выходного вектора энкодера. Чтобы решить эту проблему, нужно научиться подавать декодеру не один маленький вектор, а всю информацию, которая была в энкодере.

Как мы решаем задачу фиксированного представления последовательности:

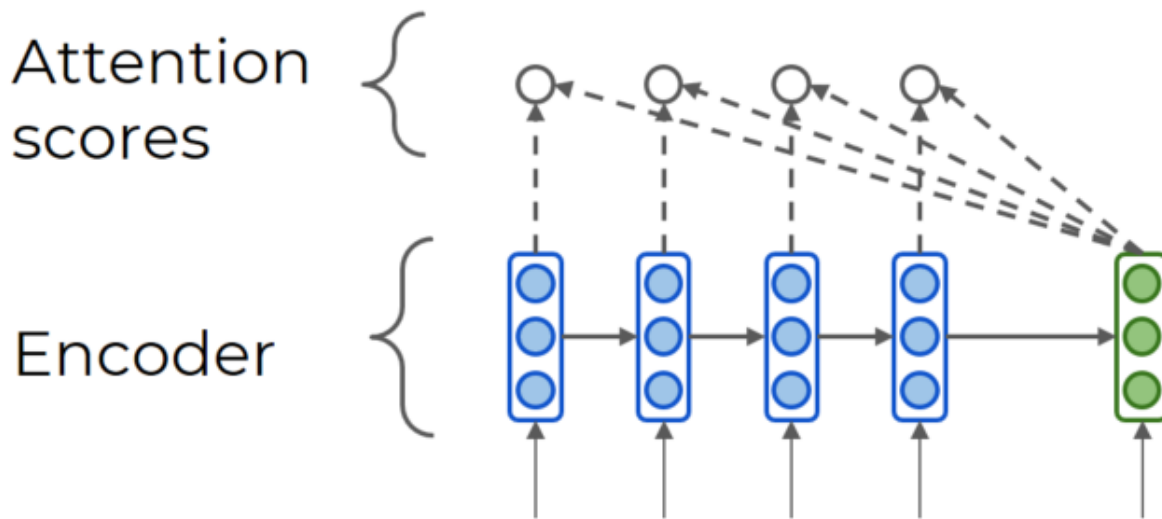
- рекуррентный подход в рекуррентных нейронных сетях, последовательная обработка каждого элемента последовательности;
- глобальный подход использования статистики по всей оси — глобальный пулинг (среднее, максимум, медиана). Глобальная статистика не имеет смысла, поскольку теряется порядок слов.

Почему бы не использовать взвешенное среднее, которое бы обуславливалось на ту позицию, на которую мы сейчас «смотрим»? Если бы взвешенное среднее зависело от состояния декодера, то мы еще на каждом шаге выбирали бы информацию.

Нужно понять на данном шаге, к какому состоянию энкодера ближе всего состояние декодера по смыслу. Можно посчитать какую-то меру близости между двумя векторами, например, евклидово расстояние. Но лучше вспомнить про косинусную меру близости.

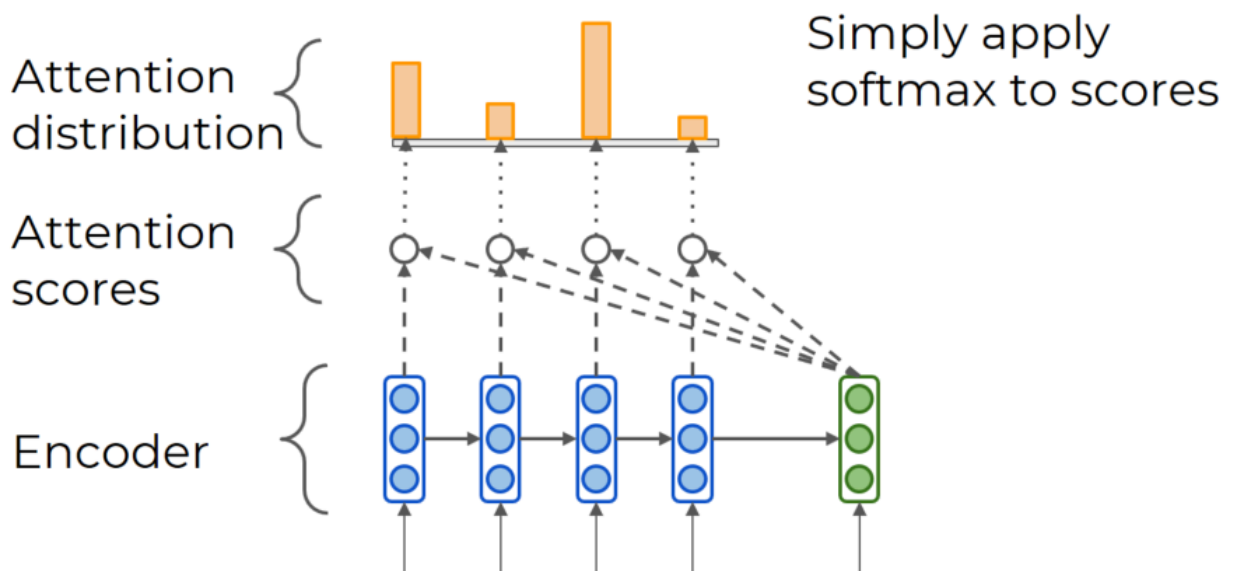
Нормы векторов, как правило, фиксированные, косинусная мера близости вполне подходит. В общем случае подойдет любой функционал, который будет возвращать скаляр двух векторов (состояния энкодера и состояния декодера).

Возьмем состояние декодера и посчитаем его меру близости со всеми состояниями энкодера:



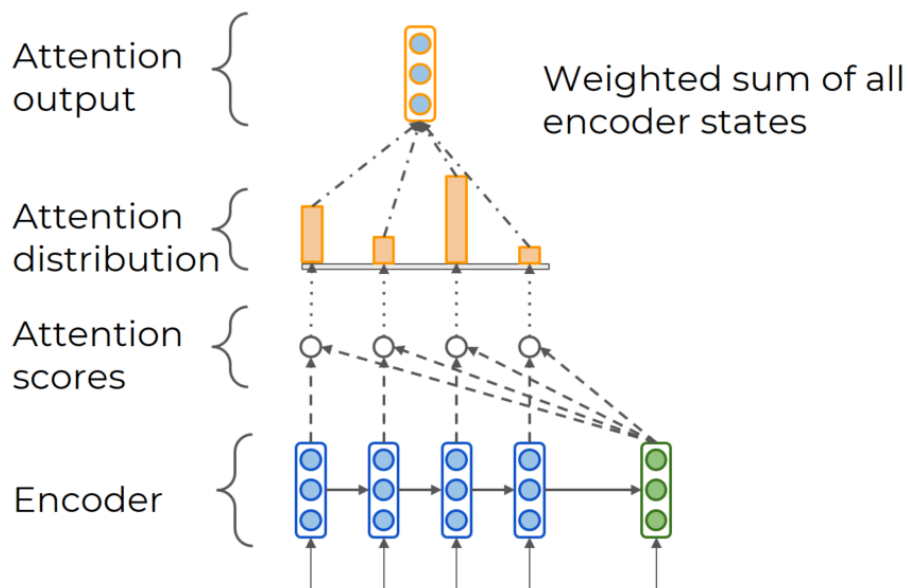
Теперь мы знаем, насколько каждое состояние энкодера важно для данного состояния декодера.

Дальше можно применить Softmax и получить набор вероятностей:



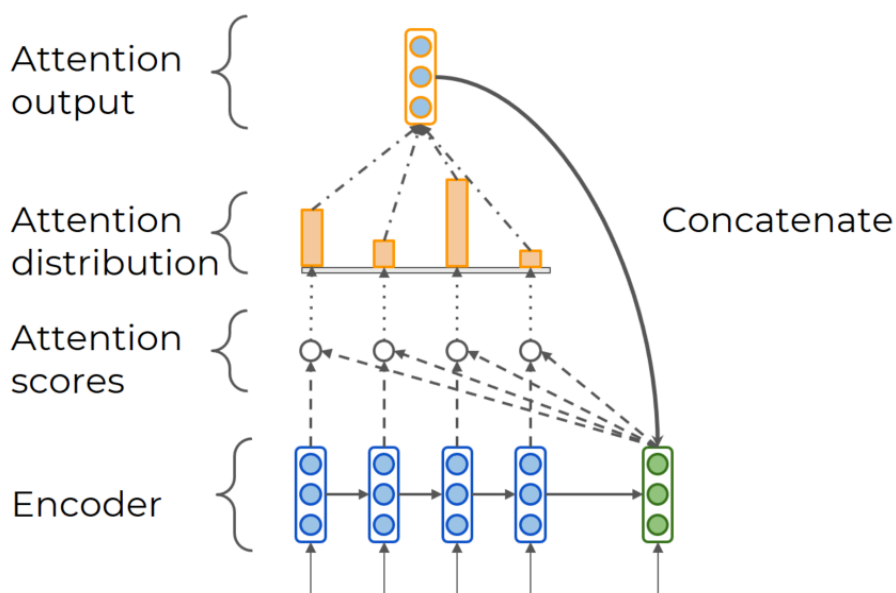
Получаем веса для взвешенного среднего.

С этими весами суммируем все состояния энкодера, получаем вектор Attention:



Вектор Attention включает информацию из всех состояний энкодера, причем веса для каждого состояния энкодера обусловлены на состояния декодера.

Теперь усредняем информацию вдоль оси времени. Способ, как мы это делаем, зависит от того, из какого состояния декодера мы «смотрим».



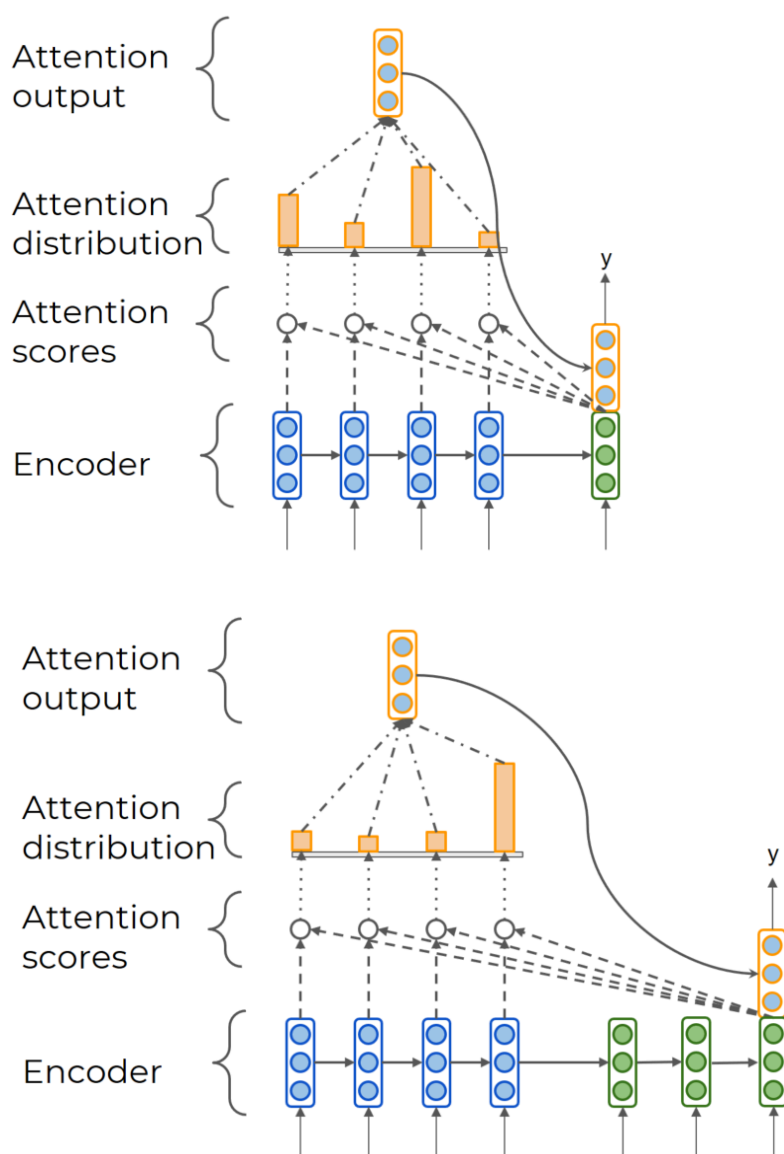
После того, как мы посчитали Attention output (вектор внимания, который включает всю нужную информацию), мы отдаем его декодеру.

Теперь у нас есть:

- текущее представление декодера о собственном контексте в момент времени, когда мы должны сгенерировать следующий шаг (например, обусловлен на свой левый контекст);
- вектор, который говорит, что конкретно из энкодера, то есть из исходной фразы, мы выделим на текущем шаге.

После чего их можно или сконкатенировать, или преобразовать во что-то новое.

Получаем один вектор. Декодер теперь знает все, что было в энкодере.



2. Механизм внимания в математической форме

Повторим еще раз, как работает механизм внимания.

Для каждого состояния декодера мы считаем взвешенное среднее всех состояний энкодера. Во взвешенном среднем веса зависят от состояния декодера. После чего мы получаем одно усредненное значение всех состояний энкодера — усредненный эмбединг. Его отдаем декодеру как некоторому агрегату информации из всех состояний энкодера, но при этом обусловленному на текущую позицию в генерируемом тексте.

Посмотрим на эти операции в математической форме.

Пусть у нас есть все скрытые состояния энкодера $h_1, \dots, h_N \in R^k$ — k -мерные вектора.

Фиксированное состояние декодера $s_t \in R^k$ — k -мерный вектор. Векторы скрытого состояния и фиксированного состояния могут быть разной размерности, ничто нам не мешает задать более сложный способ подсчета их близости (Attention Score). В простейшем случае **Attention Score** считается как скалярное произведение состояния декодера на состояние энкодера:

$$e^t = [s^T h_1, \dots, s^T h_N]$$

Для каждого состояния энкодера мы считаем меру близости состояния энкодера и фиксированного состояния декодера.

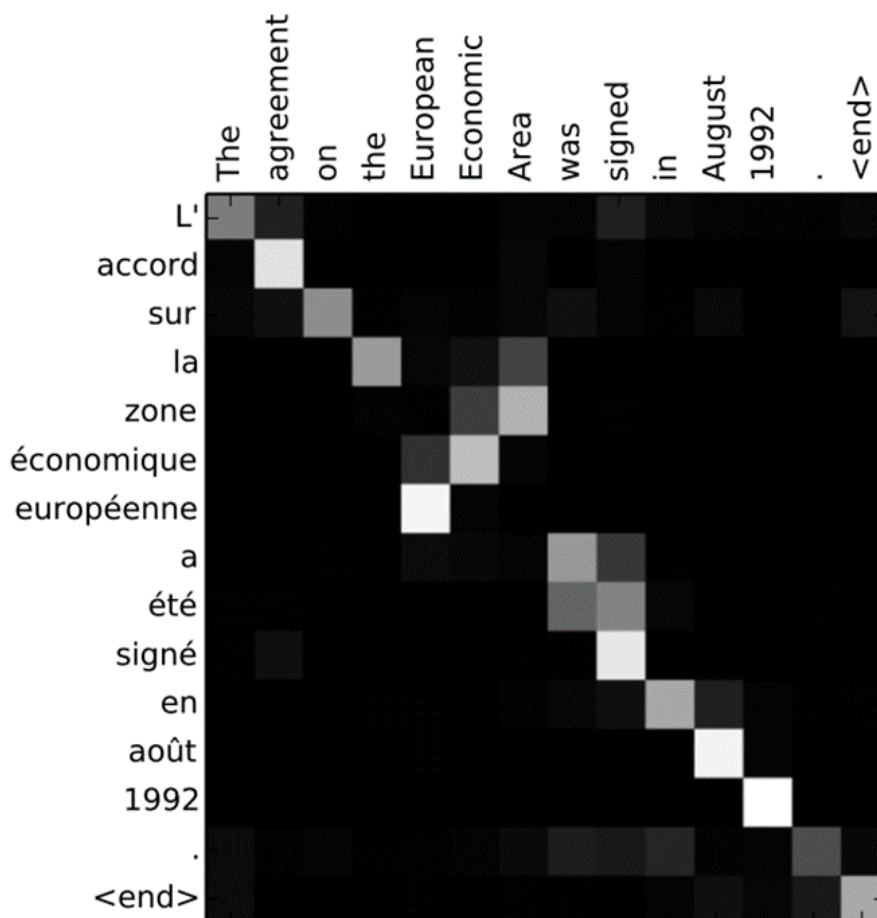
Полученный набор чисел преобразуем в вероятность с помощью **Softmax** — насколько вероятно, что данное состояние энкодера важно для текущего состояния декодера:

$$\alpha_t = \sum_{i=1}^N \alpha_i^t h_i \in R^k, \alpha_t = \text{softmax}(e_t)$$

Тогда **Attention Output** — линейная комбинация состояний энкодера.

Чтобы посчитать механизм внимания, можно использовать разные функционалы.

На матрице ниже представлена визуализация Attention Score:

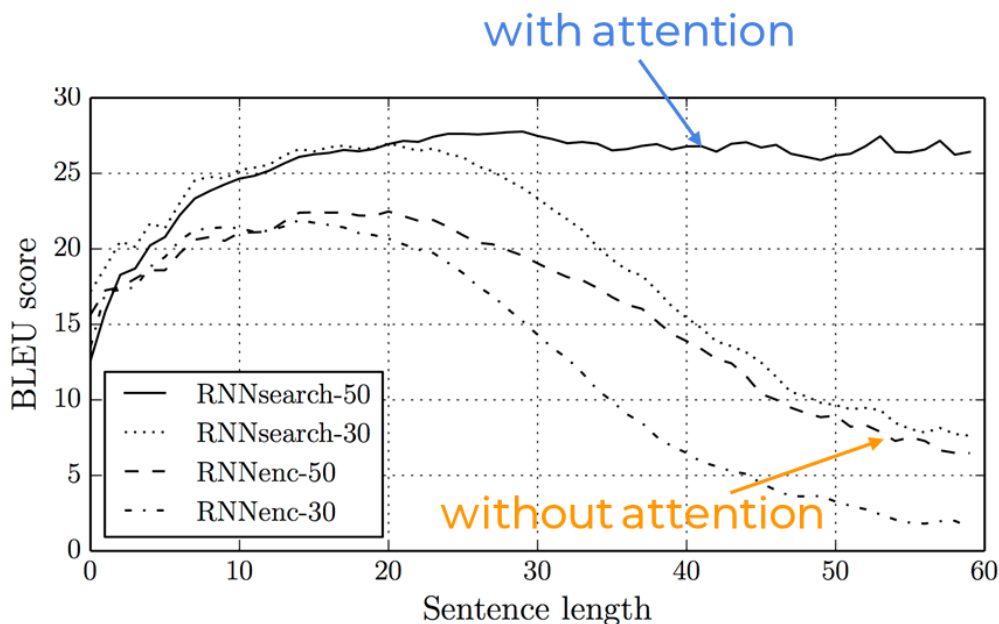


Чем ярче квадрат, тем выше вероятность, что слово является важным для соответствующего слова в декодере.

На каждом шаге нам приходит новый токен, предыдущий контекст. Мы их объединяем и получаем новое скрытое состояние, которое является состоянием энкодера. Оно и рассматривает декодер сквозь призму того, что важно. Теперь на каждом шаге декодера есть доступ ко всем состояниям энкодера, и мы не теряем информацию из-за фиксированной размерности вектора.

По такому матричному изображению Attention Score мы понимаем, каким образом на каждом шаге декодер «думал» и пытался сгенерировать следующий элемент последовательности.

Attention позволяет значительно увеличить длину последовательности текста, с которой можно работать и значительно повысить качество.



При использовании механизма внимания мы получаем:

- бесплатный word alignment;
- с ростом длины последовательности результаты не ухудшаются.

Иллюстрация выше приведена [из статьи 2014 года](#), где описан механизм внимания. По оси y – BLEU, метрика качества, которая позволяет оценивать Precision по n-граммам для сгенерированного текста. Мы видим, что сначала без механизма внимания качество растет быстрее, потому что модель получает более богатый контекст, и она более уверена в том, что генерирует. Потом достигается максимум, и далее качество падает.

При использовании механизма внимания при росте длины последовательности качество не деградирует. Это важно, потому что теперь можно обрабатывать длинные тексты. А чем больше текст, тем больше в нем деталей, тем лучше перевод. Теперь можно уловить стилистику, использовать больший контекст и т. д.

Конечно, у механизма внимания есть своя цена. Матрица Attention Score квадратная, поскольку длина последовательности в исходном и целевом языке примерно одинаковая.

На каждом шаге обработки итоговой последовательности в целевом языке мы должны посчитать Attention Score со всеми состояниями энкодера. Наша последовательность — квадратичная по длине и по сложности. До использования механизма внимания была линейная сложность. Квадратичная сложность — это дорого.

Этим и отличается механизм внимания в вычислительной машине от механизма внимания в головном мозге. Механизм внимания позволяет сфокусироваться человеку на какой-то отдельной области и пустить все вычислительные ресурсы туда. Нам не нужно больше вычислительных ресурсов, чтобы все обрабатывать. В вычислительной машине в механизме внимания фокус на каждом элементе последовательности, и происходит выбор наиболее важного элемента.

Тем не менее механизм внимания значительно повышает качество перевода. Он работает не только в машинном переводе, но в любых Seq2Seq задачах.

Сегодня механизм внимания — стандарт де факто. Практически все языковые модели машинного перевода эксплуатируют механизм внимания.

Для подсчета меры близости или меры важности используется не только скалярное произведение, но и любые функционалы:

- $e_i = s^T h_i \in R$ — классической скалярное произведение. Используется чаще всего, так как быстро вычислительно.
- $e_i = s^T W h_i \in R$, — мультипликативный механизм внимания. Переход в некоторое другое линейное пространство другой размерности.
 - $W \in R^{d_2 \times d_1}$ — матрица весов, матрица линейного преобразования.
- $e_i = v^T \tanh(W_1 h_i + W_2 s) \in R$ — аддитивный механизм внимания, который по факту эксплуатирует небольшую нейронную сеть внутри. Крайне редко встречается, поскольку очень дорогой.
 - $W_1 \in R^{d_3 \times d_1}$, $W_2 \in R^{d_3 \times d_2}$ — матрицы весов, состояния энкодера и декодера, соответственно.
 - $v \in R^{d_3}$ — вектор весов.

Все матрицы W и v — это параметры. Все операции механизмов внимания линейные, дифференцируемые, поэтому все параметры могут быть настроены с помощью градиентного спуска. Это значит, что все механизмы внимания обладают одним общим свойством: позволяют с помощью определенной операции оценить меру близости состояний энкодера и декодера, а потом использовать это, чтобы получить взвешенное среднее для состояния энкодера.

Механизм внимания является одной из важнейших техник для обработки естественного языка, при этом он крайне мало отличается от простого мешка слов.

Обращаем внимание, что в архитектуре модель с механизмом внимания и без него — это две разные модели. У них разные структуры с точки зрения весов, они учатся на разных элементах: с Attention каждое состояние обусловлено на текущее слово, но в окружающем его контексте; без Attention оно будет обусловлено на всю левую последовательность.

3. Механизм внимания Self-Attention

Механизм Self-Attention лег в основу статьи [Attention is all you need, 2017](#), которая перевернула мир машинного перевода NLP и глубокое обучение в целом.

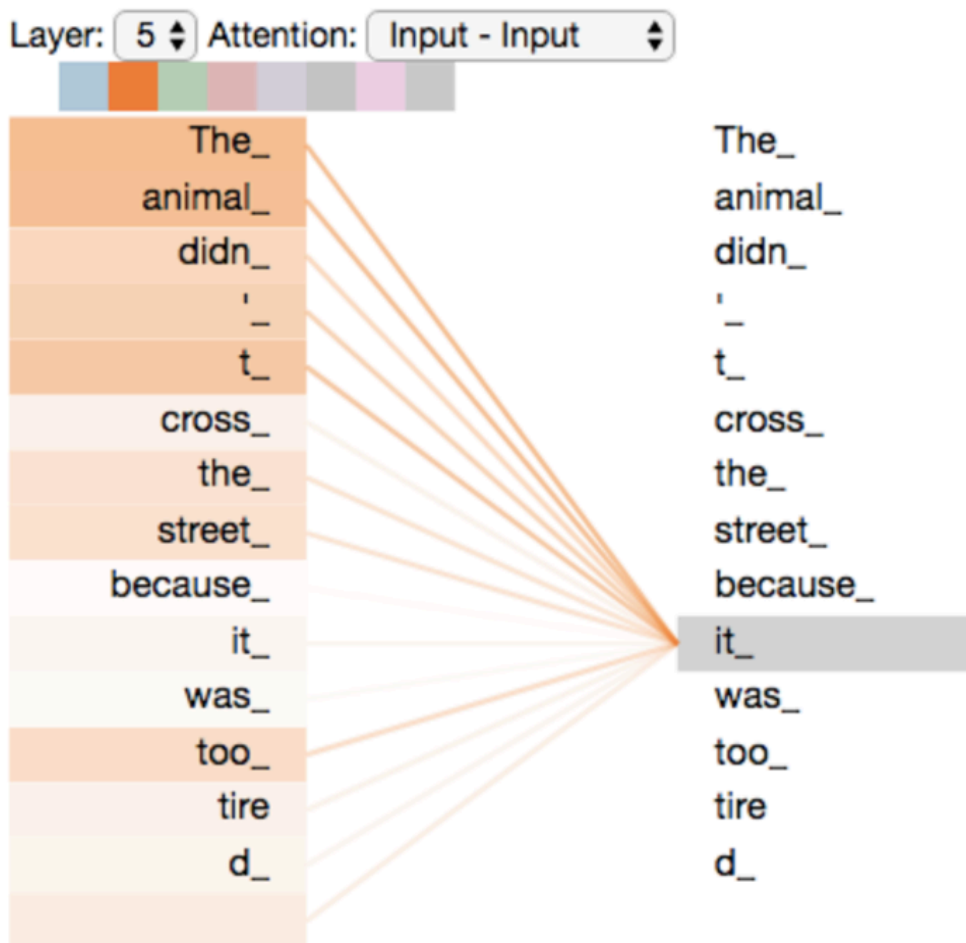
Self-Attention на высоком уровне позволяет достаточно интуитивно отвечать на некоторые вопросы.

Пример. The animal didn't cross the street because it was too tired.

Можем ли мы понять, какие слова к каким относятся в этом примере? Местоимение *it* — это улица или животное? Как донести до модели, что *it* — это «оно», «животное»?

Self-Attention и будет заниматься налаживанием связей между различными элементами заданной последовательности. Причем делать это он будет самостоятельно. Self-Attention — механизм внимания внутри одной последовательности. Мы дадим нашей модели возможность, откуда и куда нужно смотреть, чтобы повысить качество классификации, регрессии и обработки текста в целом.

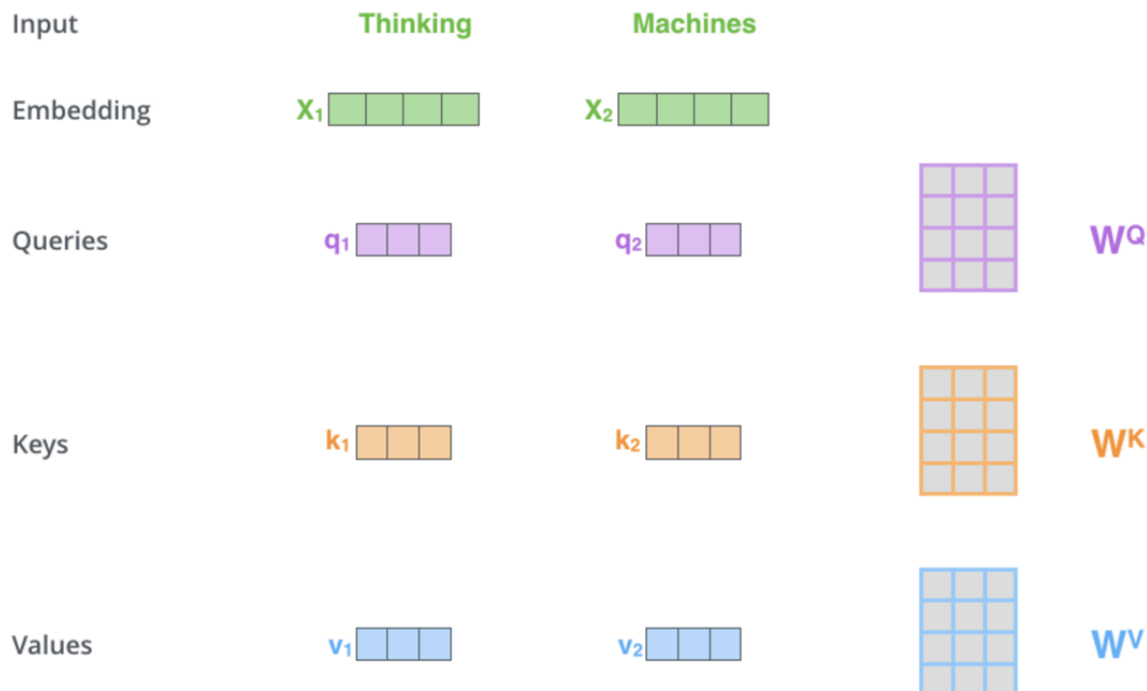
Визуализация Self-Attention для нашего примера:



Мы получаем Attention Scores из предложения в самого себя. Каждое состояние нашего энкодера, каждая позиция в предложении может «посмотреть» на все остальные позиции в предложении и на саму себя.

Для it мы видим, насколько важно каждое слово. И мы видим, что it — это the animal. Это модель поняла самостоятельно. Предложение разбито не только на слова, но и на буквы — более продвинутая процедура токенизации.

Подсчет Attention от элемента последовательности ко всем остальным элементам и к себе самому



Attention был направлен из энкодера в декодер. У Self-Attention нет никакого декодера. К разным частям предложения задаются разные вопросы, и Attention должен быть различным, если он показывает какую-то связь между словами. Поэтому Self-Attention должен быть направленным, а значит, Attention из примера выше от thinking к machines и от machines к thinking должны быть двумя разными числами. Скалярное произведение двух эмбедингов симметрично, направленности не даст.

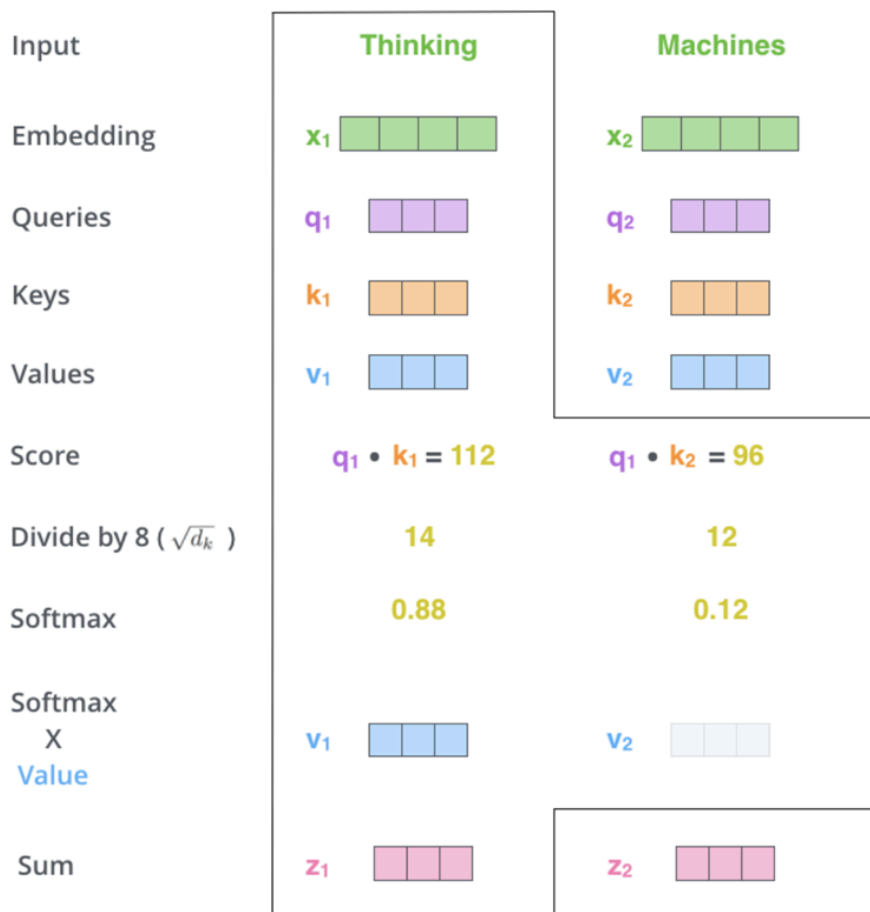
Шаг 1. Возьмем три пространства, каждое из которых отвечает за небольшую функцию:

- Пространство запросов — Queries. Query — то, как мы описываем слово, если мы из него «смотрим» на все остальные. По аналогии, состояние декодера.
- Пространство ключей — Keys — то, как мы описываем слово, если мы на него «смотрим». По аналогии, состояние энкодера.

- Values — то, что мы видим, когда смотрим на слово под определенным ключом. Это те состояния энкодера, которые будем усреднять с помощью Score, который получили из Keys.

У нас есть исходные эмбединги для слов — из примера это X_1 и X_2 . И мы с помощью некоторого линейного преобразования переводим данные эмбединги в пространства Queries, Keys, Values. Мы умножаем на три матрицы весов (на рисунке справа), чтобы попасть в нужное пространство, и получаем по три вектора для каждого из слов. Теперь посчитаем Self-Attention с помощью Query, Key, Value.

Шаг 2. Подсчитаем Score. У нас есть thinking. Мы из него «смотрим» в слово thinking (на самого себя) и в слово machines.



$q_1 \cdot k_1 = 112$ — Attention из слова thinking в себя.

$q_1 \cdot k_2 = 96$ — Attention из слова thinking в machines.

Шаги 3–4. Чтобы понизить дисперсию, делим на $\sqrt{8}$ – корень из размерности.

Получаем взвешенное среднее состояний для каждой из позиций. Применяем Softmax.

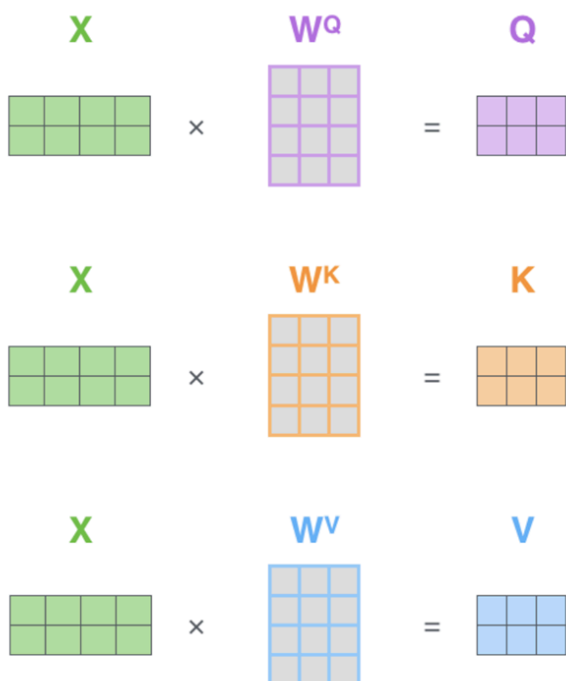
Шаги 5-6. Суммируем v_1 и v_2 с весами 0,88 и 0,12. Получаем вектор z_1 , который объединяет информацию обо всем предложении в одном векторе.

Таким образом, мы посмотрели на все предложение сквозь призму слова thinking. То же самое делаем для слова machines, получим z_2 . Это еще одно представление всего предложения, но сквозь призму второго слова.

Теперь мы можем оценивать все предложение сквозь призму каждой из позиций. Из одной позиции мы имеем представление не только об одном слове, но и обо всем предложении. Из последовательности эмбедингов для слов получаем последовательность эмбедингов для предложения. Получаем несколько различных вариаций для эмбединга всего предложения.

Матричное умножение

Self-Attention неплохо параллелизуется и эффективно считается в отличие от рекуррентных сетей.

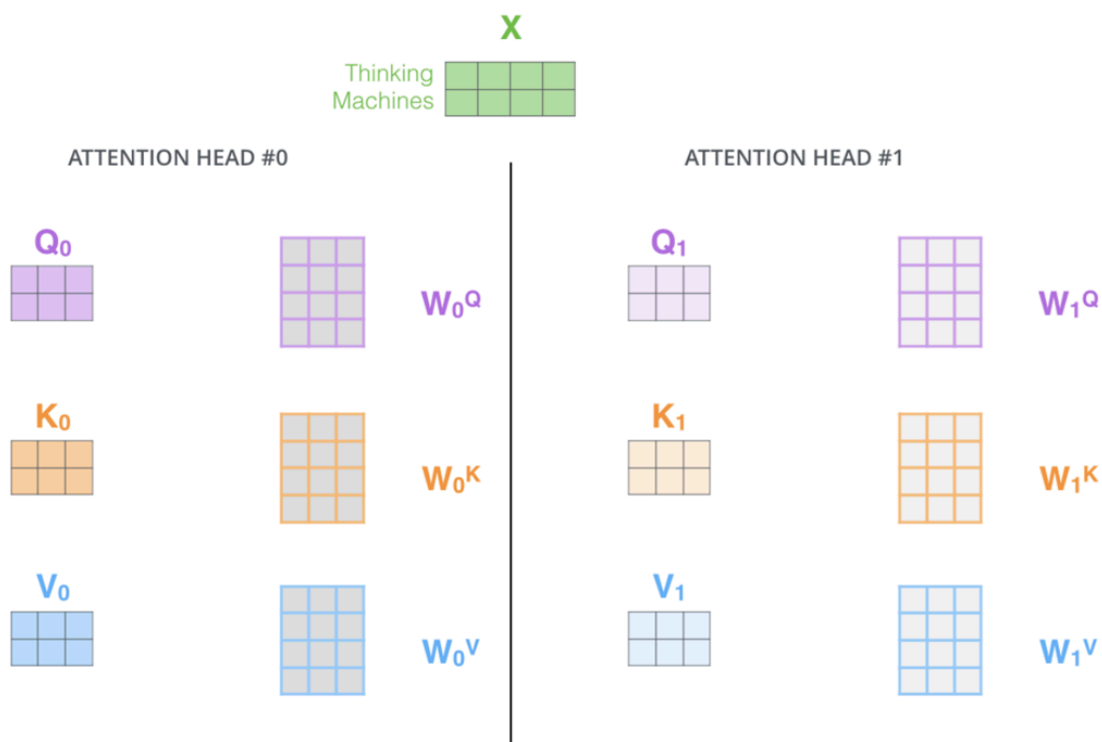


$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \square & \square \\ \square & \square \\ \square & \square \end{matrix} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix} = \begin{matrix} \text{Z} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix}$$

Все матричное умножение — крайне быстро и эффективно с точки зрения оптимизации.

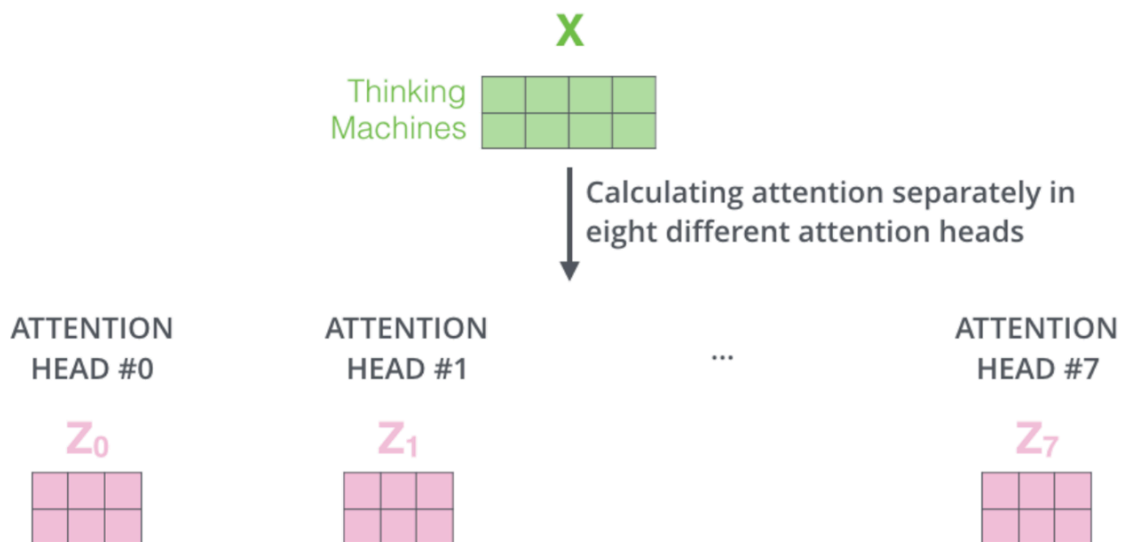
4. Механизм внимания Multi-Head Attention

Мы используем различные матрицы W с разными весами, чтобы получать разные механизмы внимания.



В сверточных сетях операция свертки использует много различных ядер, чтобы выделять зависимости (паттерны, патчи) на исходном изображении.

В предложении между словами также могут быть различные связи — к одному и тому же слову задается несколько вопросов. Мы можем использовать различные **Attention-Heads** (головы Attention), которые будут отвечать за различные смысловые части.



В этом и заключается смысл Self-Attention. Мы позволяем модели не только «посмотреть» на все соседние позиции из каждого слова и понять, на сколько они важны, но еще и взвешенно усреднить их несколькими различными способами.

Все эти операции производятся независимо от длины последовательности, позиции не меняются. Мы преобразуем каждую позицию, обуславливаясь на все остальные позиции, и именно поэтому архитектура называется Трансформер. Для k векторов на входе она даст k векторов на выходе, но это будут обогащенные, более информативные вектора, которые «знают» обо всей последовательности в целом.

Чтобы не было увеличения или перегруза информацией на выходе из-за нескольких голов, все результаты усредняются.

1) Concatenate all the attention heads

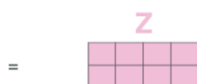


2) Multiply with a weight matrix W^O that was trained jointly with the model

\times



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Несколько голов можно сконкатенировать вдоль оси каналов (оси смысла, признаков) в вектор такого же размера, который был на входе. Для этого достаточно умножить на матрицу подходящего размера. Матрицу точно так же обучаем.

За счет использования множества голов Self-Attention улавливает различные связи в исходном предложении, тем самым позволяет нам извлечь много информации из одной фразы.

1) This is our input sentence*

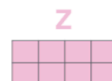
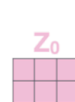
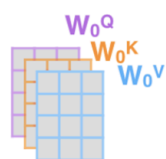
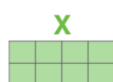
2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

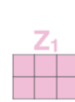
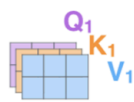
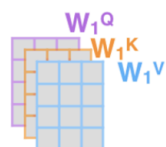
4) Calculate attention using the resulting $Q/K/V$ matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

Thinking Machines



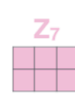
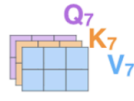
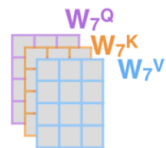
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



...

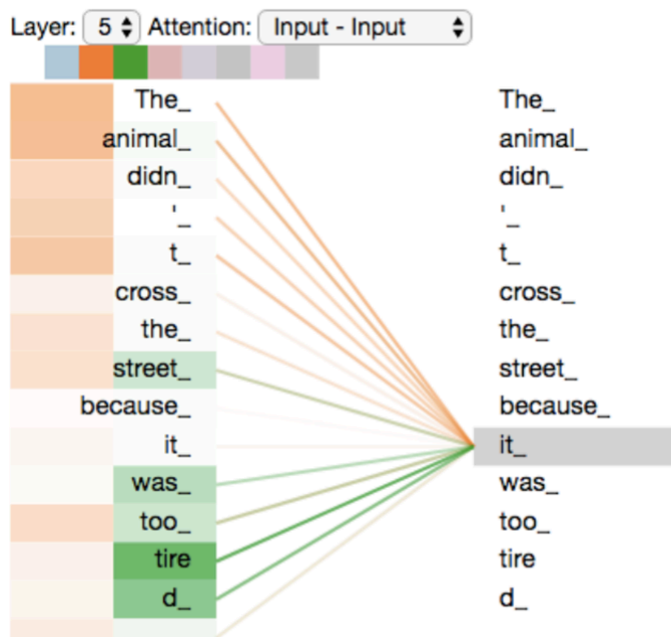
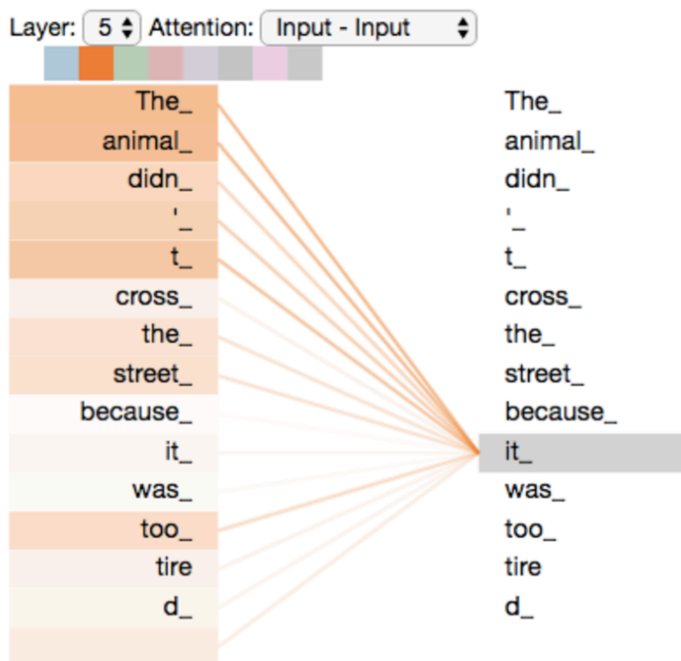
...

...



Подаем фразу без дополнительной разметки, без какого-либо ее перевода. Теперь определяем, как различные позиции в фразе связаны между собой и как именно они влияют друг на друга. Единственное, что придется учить, это матрицы Q, K, V в каждой голове.

Теперь исходную картинку можно рассмотреть целиком:

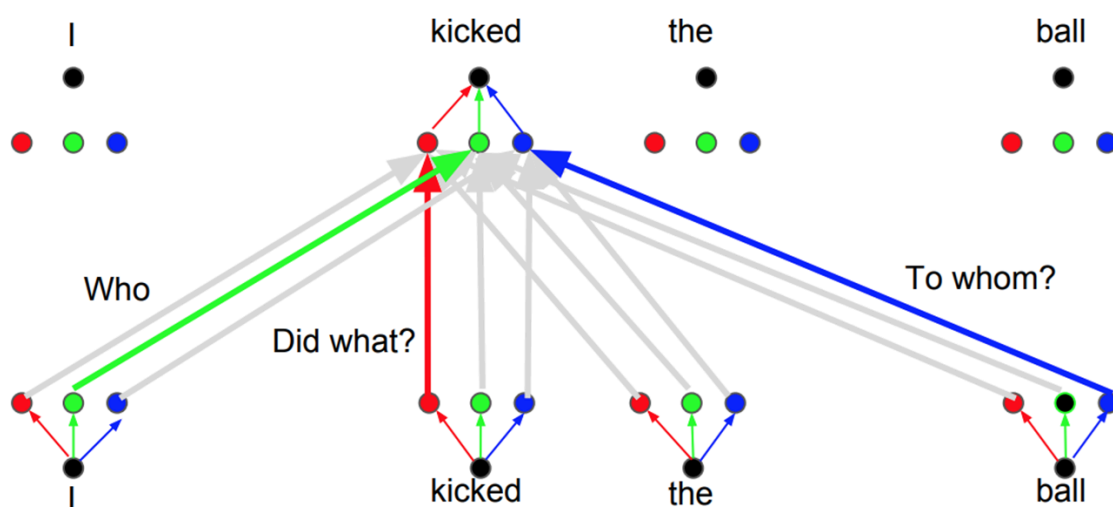


Механизм Self-Attention работает не только с последовательностями, но и на любом структурированном входе. Существуют Visual Transformers. Картинка — упорядоченный набор пикселей, ее можно поделить на патчи и между ними считать Self-Attention.

Точно так же обрабатывается временной ряд (временными окнами), графы (считать эмбединг для каждой вершины графа, и между ними считать Self-Attention).

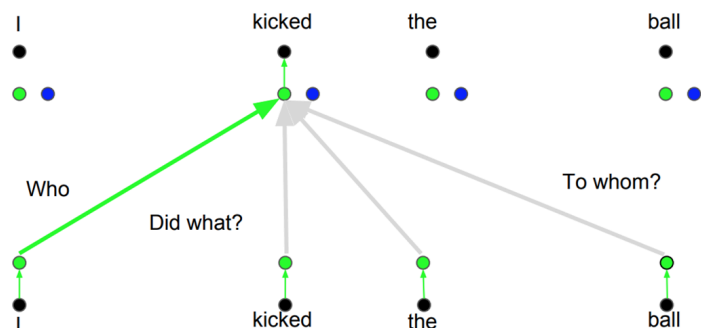
Механизм внимания — простой механизм подсчета близости между элементами входа (сложной упорядоченной структуры данных).

Посмотрим визуально, как работает Multi-Head Attention:

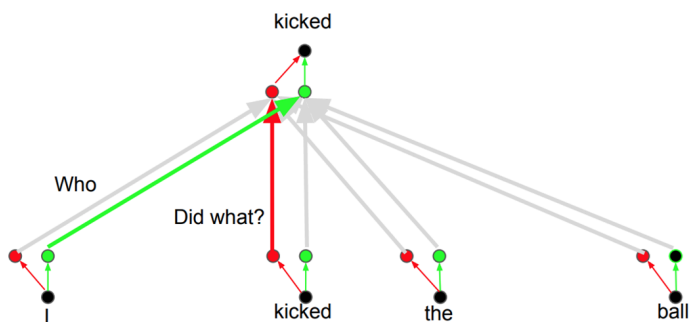


Каждый цвет на картинке отвечает за определенную голову. Здесь 3 головы, и 3 различных вопроса, которые описывают каждый Attention Head. Конечно, это наша интерпретация, что они «описывают» какие-то вопросы. Никаких вопросов они не задают, а просто статистически устанавливают в предложении различные зависимости между словами. Наша речь — достаточно сложно структурированная система, так как в ней есть и грамматика, и семантика, и пр.

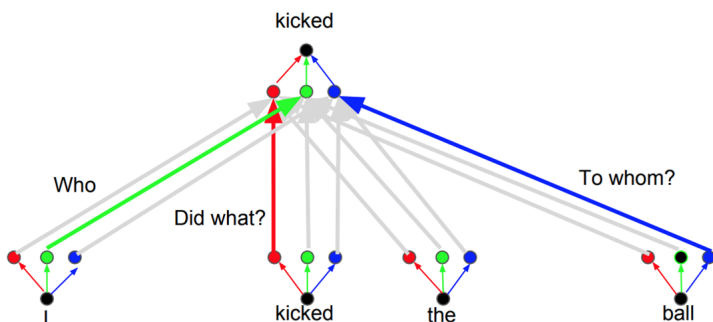
Attention head: Who



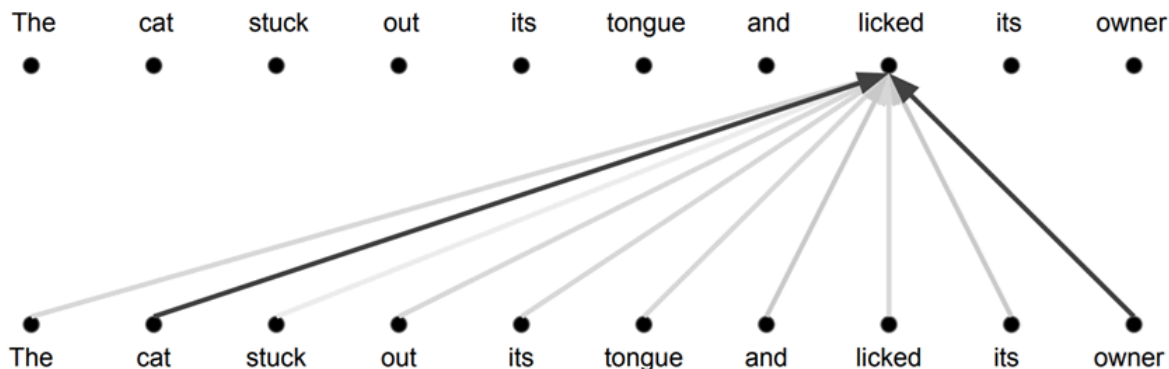
Attention head: Did What?



Attention head: To Whom?

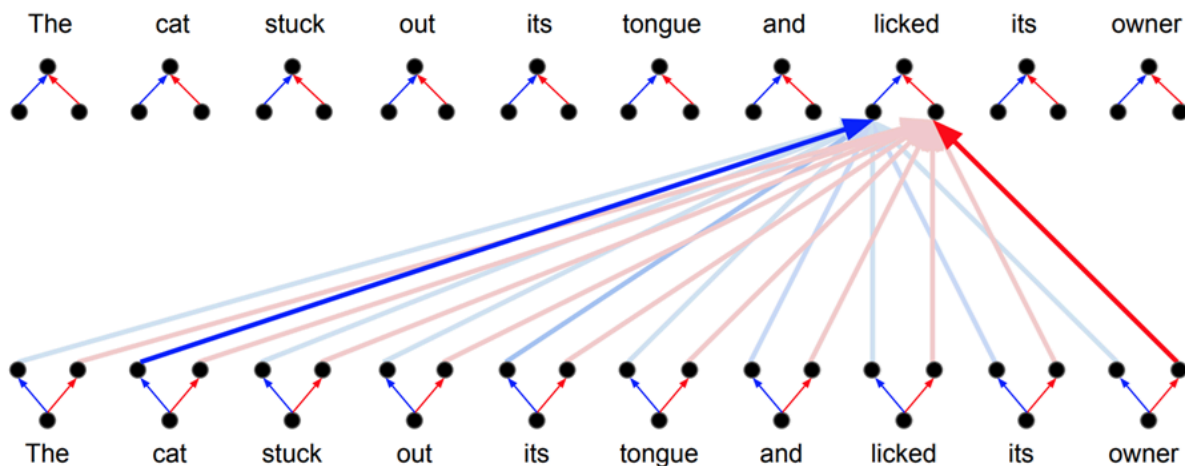


Разница между Attention и Self-Attention достаточно проста. Attention позволяет из состояния декодера посмотреть во все состояния энкодера.



Self-Attention позволяет из всех состояний декодера посмотреть «в себя».

Attention опирается только на одно представление слов, он один раз считает их меру близости. Multi-Head Attention преобразует слова в несколько различных пространств, где на первый план выходят различные их свойства. В этих пространствах считается их значимость, поэтому можно уловить больше различных связей.



Все процессы в Multi-Head Attention хорошо параллелизуются. Все головы независимы друг от друга, поэтому могут считаться параллельно.

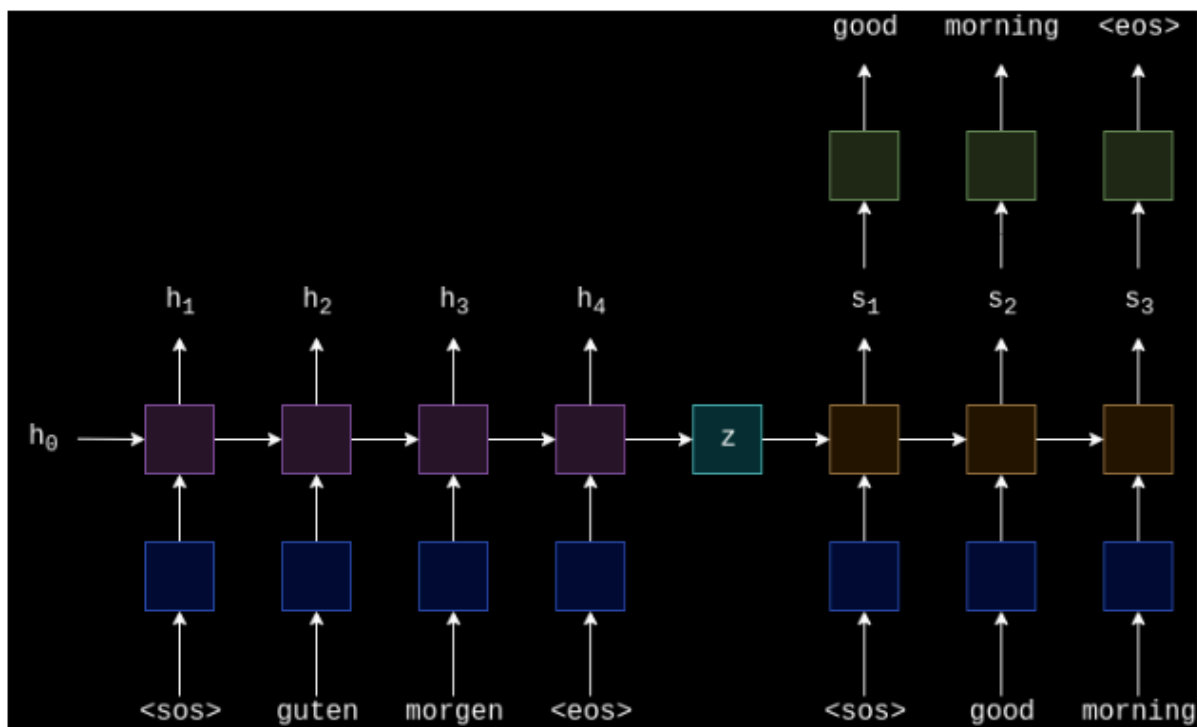
Self-Attention не мешает нам использовать архитектуру (например, GPU) для параллельных вычислений. В отличие от RNN, здесь нет последовательной структуры. В Multi-Head Attention мы сразу считаем отношение всего ко всему.

5. Реализация механизма внимания на Python

В задаче машинного перевода на вход мы подаем некоторое предложение, и на выходе должны получить какое-то предложение, но не обязательно той же самой длины. Следовательно предложение — некоторая последовательность слов. Если слова поменять местами, то смысл может измениться. У нас формируется задача: последовательность на входе и последовательность на выходе. То есть **Sequence to Sequence** или **Seq2Seq**.

Модели, которые решают такие задачи, также называются Seq2Seq моделями. Они состоят из двух частей: энкодера и декодера.

В занятии рассмотрим модель из статьи [Sequence to Sequence Learning with Neural Networks](#).



На примере перевода видим:

- Guten morgen — предложение ввода или источника, которое проходит через слой эмбединга (желтый, служит для сопоставления элементов речи — слова, предложения и т. д. — числовому вектору), а затем вводится в кодировщик (зеленый).
- В начало и конец предложения всегда добавляются токены начала последовательности (<sos> — start of sequence) и конца последовательности (<eos> — end of sequence).
- На каждом временном шаге на вход в RNN кодировщика подается как эмбединг-версия текущего слова $e(x_t)$, которая порождена слоем эмбединга e , так и скрытое состояние из предыдущего временного шага, h_{t-1} . На выход RNN кодировщика подает новое скрытое состояние h_t .

Подготовка данных

Для предобработки используются библиотеки torchtext и nltk.

Загрузим **параллельный корпус текстов** — это корпус Multi30k, который представляет набор данных из ~30 000 параллельных предложений на английском, немецком и французском языках:

```
!pip install torchtext==0.11.0
```

```
!pip install portalocker
```

Переведем с немецкого на английский:

```
from torchtext.datasets import Multi30k

train_iter = Multi30k(split="train")
```

```
# torchtext.datasets.DatasetName yield exhaustible
IterableDataset.

# To fix this we convert our dataset to a list.

train_data = list(train_iter)

print(f"Number of training examples: {len(train_data)}")

print(train_data[0])
```

```
>>> 100%|██████████| 1.21M/1.21M [00:01<00:00, 983kB/s]Number of
training examples: 29000

('Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.\n',
'Two young, White males are outside near many bushes.\n')
```

Реализуем **токенизаторы**. Они превращают строку, которая содержит предложение, в список отдельных токенов, которые составляют эту строку. Например «доброе утро!» становится ["доброе", "утро", "!"].

Каждому токenu сопоставляется индекс, каждому индексу сопоставляется вектор слова — эмбединг слова:

```
from nltk.tokenize import WordPunctTokenizer

tokenizer = WordPunctTokenizer()

print(tokenizer.tokenize("good morning!"))
```

```
>>> ['good', 'morning', '!']
```

Теперь реализуем отдельную функцию для токенизации:

```
def tokenize(sent):  
    return tokenizer.tokenize(sent.rstrip().lower())  
  
src, trg = train_data[0]  
  
print(tokenize(src))  
print(tokenize(trg))
```

```
>>> ['zwei', 'junge', 'weiße', 'männer', 'sind', 'im', 'freien',  
    'in', 'der', 'nähe', 'vieler', 'büsche', '.']  
  
['two', 'young', ',', 'white', 'males', 'are', 'outside', 'near',  
    'many', 'bushes', '.']
```

Построим **словари** (множества уникальных токенов) для исходного и целевого языков. Будем учитывать токены, которые встречаются не реже `min_freq` раз. Словарь связывает каждый уникальный токен с индексом (целым числом). Словари исходного и целевого языков различаются.

```
from collections import Counter  
  
from torchtext.vocab import vocab as Vocab  
  
src_counter = Counter()  
trg_counter = Counter()  
  
for src, trg in train_data:
```

```
src_counter.update(tokenize(src))

trg_counter.update(tokenize(trg))

src_vocab = Vocab(src_counter, min_freq=2)
trg_vocab = Vocab(trg_counter, min_freq=2)
```

Добавим **технические токены**. Токен <UNK> используется для обозначения токенов, которые не присутствовали в словаре, построенном по обучающей выборке:

```
unk_token = "<UNK>"

for vocab in [src_vocab, trg_vocab]:
    if unk_token not in vocab:
        vocab.insert_token(unk_token, index=0)
        vocab.set_default_index(0)
```

Также добавим технические токены начала строки <SOS>, конца строки <EOS> и паддинга <PAD>:

```
sos_token, eos_token, pad_token = "<SOS>", "<EOS>", "<PAD>"
specials = [sos_token, eos_token, pad_token]

for vocab in [src_vocab, trg_vocab]:
    for token in specials:
        if token not in vocab:
```

```
vocab.append_token(token)
```

Оценим полученный размер словаря:

```
print(f"Unique tokens in source (de) vocabulary:
{len(src_vocab)}")

print(f"Unique tokens in target (en) vocabulary:
{len(trg_vocab)}")
```

```
>>> Unique tokens in source (de) vocabulary: 7892
```

```
Unique tokens in target (en) vocabulary: 5903
```

Наконец, реализуем простую функцию encode для приведения последовательности к формату, с которым работают нейронные сети. Она объединяет все предыдущие наработки.

```
def encode(sent, vocab):

    tokenized = [sos_token] + tokenize(sent) + [eos_token]

    return [vocab[tok] for tok in tokenized]

print(encode(src, src_vocab)[::-1])

print(encode(trg, trg_vocab))
```

```
>>> [7890, 13, 180, 6, 457, 33, 1596, 15, 46, 0, 236, 423, 157,
249, 193, 210, 3358, 33, 687, 8, 25, 18, 7889]
```

```
[5900, 18, 28, 14, 284, 89, 18, 1573, 30, 214, 256, 38, 534, 71,
18, 1051, 659, 3, 61, 1549, 89, 1421, 14, 38, 181, 11, 5901]
```

Последний шаг подготовки данных — создание итераторов. С их помощью можно итерационно возвращать пакеты данных, которые будут иметь атрибут `src` (тензоры PyTorch, которые содержат набор оцифрованных исходных предложений) и атрибут `trg` (тензоры PyTorch, которые содержат набор оцифрованных целевых предложений).

Оцифрованные предложения — это просто причудливый способ сказать, что они были преобразованы из последовательности читаемых токенов в последовательность соответствующих индексов с использованием словаря.

```
import torch

from torch.nn.utils.rnn import pad_sequence

from torch.utils.data import DataLoader

def collate_batch(batch):

    src_list, trg_list = [], []

    for src, trg in batch:

        # Encode src and trg sentences, convert them into a
        # tensor

        # and store them in src_list and trg_list respectively.

        src_encoded = encode(src, src_vocab)

        src_list.append(torch.tensor(src_encoded))

        trg_encoded = encode(trg, trg_vocab)

        trg_list.append(torch.tensor(trg_encoded))

    # Pad sequences with pad_sequence function.

    # src_padded = pad_sequence(...)
```

```
# trg_padded = pad_sequence(...)

src_padded = pad_sequence(src_list,
padding_value=src_vocab[pad_token])

trg_padded = pad_sequence(trg_list,
padding_value=trg_vocab[pad_token])

return src_padded, trg_padded

batch_size = 256

train_dataloader = DataLoader(train_data, batch_size,
shuffle=True, collate_fn=collate_batch)

src_batch, trg_batch = next(iter(train_dataloader))

src_batch.shape, trg_batch.shape
```

```
>>> (torch.Size([34, 256]), torch.Size([41, 256]))
```

Теперь генерируем батчи для обработки данных из обучающей выборки. Аналогичным образом обрабатываем валидационную выборку:

```
val_data = list(Multi30k(split="valid"))

val_dataloader = DataLoader(val_data, batch_size,
collate_fn=collate_batch)
```

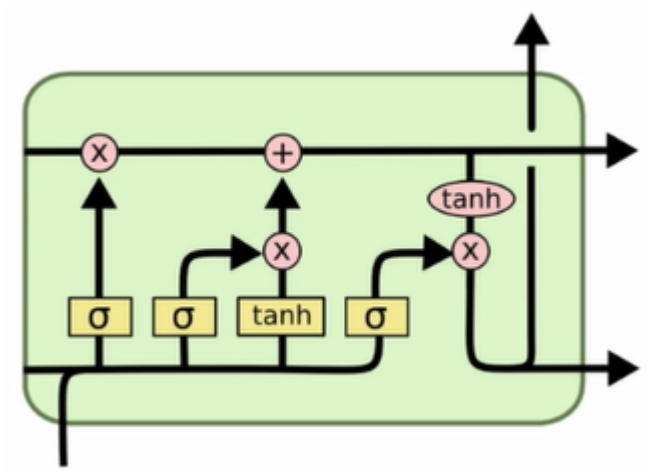
```
>>> 100%|██████████| 46.3k/46.3k [00:00<00:00, 280kB/s]
```

Построение модели Seq2Seq

Модель состоит из двух частей: энкодер и декодер. Затем они будут объединены в одну модель Seq2Seq.

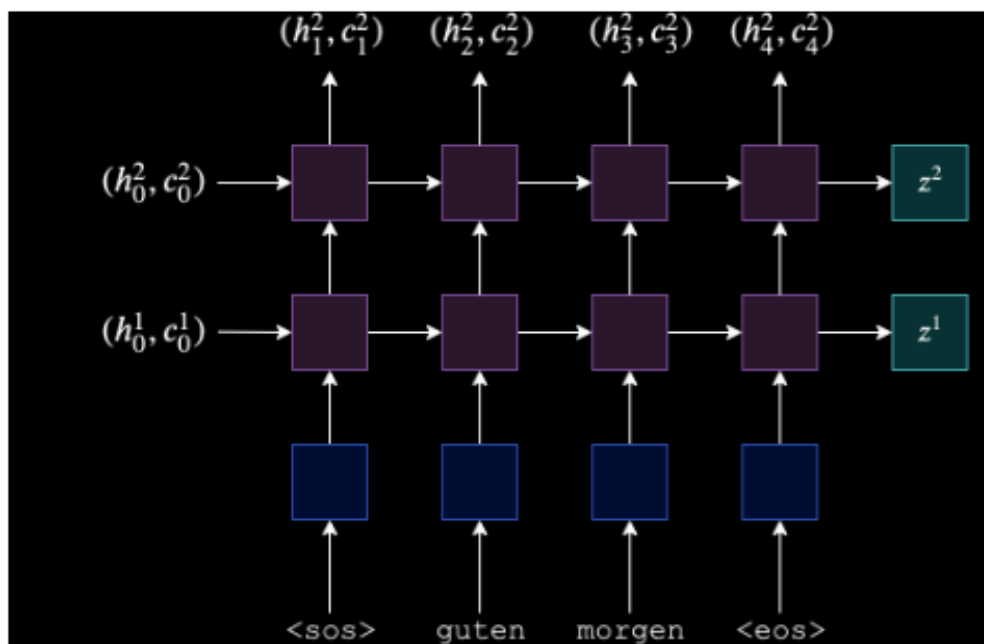
Энкодер — это двухслойная LSTM сеть. В оригинальной статье используется четырехслойная LSTM, но в целях экономии времени она упрощена.

Повторим еще раз, как выглядит LSTM:



LSTM — модификация структуры Vanilla RNN, у которой существует skip-connection. Он пробрасывает данные с предыдущего слоя практически без изменений. Эти данные либо запоминаются и идут дальше, либо не запоминаются.

Визуально энкодер выглядит следующим образом:



Ему будет соответствовать класс `Encoder`. Конструктор этого класса принимает следующие аргументы:

- `input_dim` — размер или размерность one-hot векторов, которые будут вводиться в кодировщик. Они равны размеру входного (исходного) размера словаря.
- `emb_dim` — размерность слоя эмбединга. Этот слой преобразует one-hot векторы в dense векторы с размерами `emb_dim`.
- `hid_dim` — размерность скрытого состояния и состояния ячейки.
- `n_layers` — количество слоев в RNN.
- `dropout` — количественная характеристика дропаута. Это параметр регуляризации для предотвращения переобучения.

Отметим, что аргумент `dropout` для LSTM заключается в том, сколько связей необходимо отключить между уровнями многослойной RNN, то есть между скрытыми состояниями, выводимыми из уровня l , и теми же скрытыми состояниями, используемыми для ввод слоя $l + 1$.

RNN возвращает:

- `outputs` — скрытые состояния верхнего уровня для каждого временного шага;
- `hidden` — окончательное скрытое состояние для каждого слоя, h_T , которые наложены друг на друга;
- `cell` — конечное состояние ячейки для каждого слоя, c_T , которые наложены друг на друга.

Размеры каждого из тензоров оставлены в виде комментариев в коде. В этой реализации `n_directions` всегда будет 1. Обратите внимание, существуют и двунаправленные RNN, они могут иметь `n_directions` равное 2.

```
import torch.nn as nn

class Encoder(nn.Module):

    def __init__(self, n_tokens, emb_dim, hid_dim, n_layers,
```

```
dropout):  
  
    super().__init__()  
  
    self.n_tokens = n_tokens  
    self.hid_dim = hid_dim  
    self.n_layers = n_layers  
  
    # YOUR CODE HERE  
  
    # Define embedding, dropout and LSTM layers.  
    self.embedding = nn.Embedding(n_tokens, emb_dim)  
    self.dropout = nn.Dropout(dropout)  
  
    self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers,  
dropout=dropout)  
  
    def forward(self, src):  
  
        # src has a shape of [seq_len, batch_size]  
  
        # Compute an embedding from src data and apply dropout.  
        # embedded should have a shape of [seq_len, batch_size,  
emb_dim]  
  
        embedded = self.embedding(src)  
        embedded = self.dropout(embedded)  
  
        # Compute the RNN output values.
```

```
# When using LSTM, hidden should be a tuple of two
tensors:

# 1) hidden state

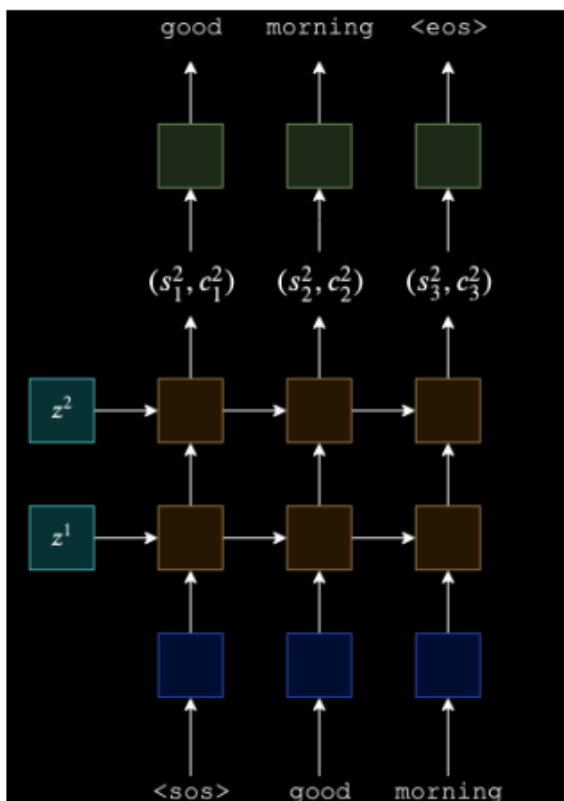
# 2) cell state

# both of shape [n_layers * n_directions, batch_size,
hid_dim]

_, hidden = self.rnn(embedded)

return hidden
```

Декодер также представлен двуслойной LSTM. Иллюстрация декодера:



Основное отличие декодера от энкодера в том, что он принимает начальное скрытое состояние от энкодера, что позволяет передать информацию о рассматриваемом предложении декодеру.

Аргументы и инициализация аналогичны классу Encoder, за исключением `output_dim`, который является размером словаря для выходной или целевой последовательности. Кроме того, добавлен слой Linear, который используется для прогнозирования токена на основе скрытого состояния верхнего уровня.

```
class Decoder(nn.Module):

    def __init__(self, n_tokens, emb_dim, hid_dim, n_layers,
dropout):

        super().__init__()

        self.n_tokens = n_tokens

        self.hid_dim = hid_dim

        self.n_layers = n_layers

        # Define embedding, dropout and LSTM layers.

        # Additionally, Decoder will need a linear layer to
predict next token.

        self.embedding = nn.Embedding(n_tokens, emb_dim)

        self.dropout = nn.Dropout(dropout)

        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers,
dropout=dropout)

        self.out = nn.Linear(hid_dim, n_tokens)

    def forward(self, input, hidden):
```

```
# input has a shape of [batch_size]

# hidden is a tuple of two tensors:

# 1) hidden state

# 2) cell state

# both of shape [n_layers, batch_size, hid_dim]

# (n_directions in the decoder shall always be 1)


# Compute an embedding from input data and apply dropout.

# Remember, that LSTM layer expects input to have a shape
of

# [seq_len, batch_size, emb_dim], which means that we
need

# to somehow introduce the seq_len dimension into our
input tensor.

input = input.unsqueeze(dim=0)

embedded = self.embedding(input)

embedded = self.dropout(embedded)


# Compute the RNN output values.


output, hidden = self.rnn(embedded, hidden)

# output has a shape of [seq_len, batch_size, hid dim]

# Compute logits for the next token probabilities from
RNN output.
```

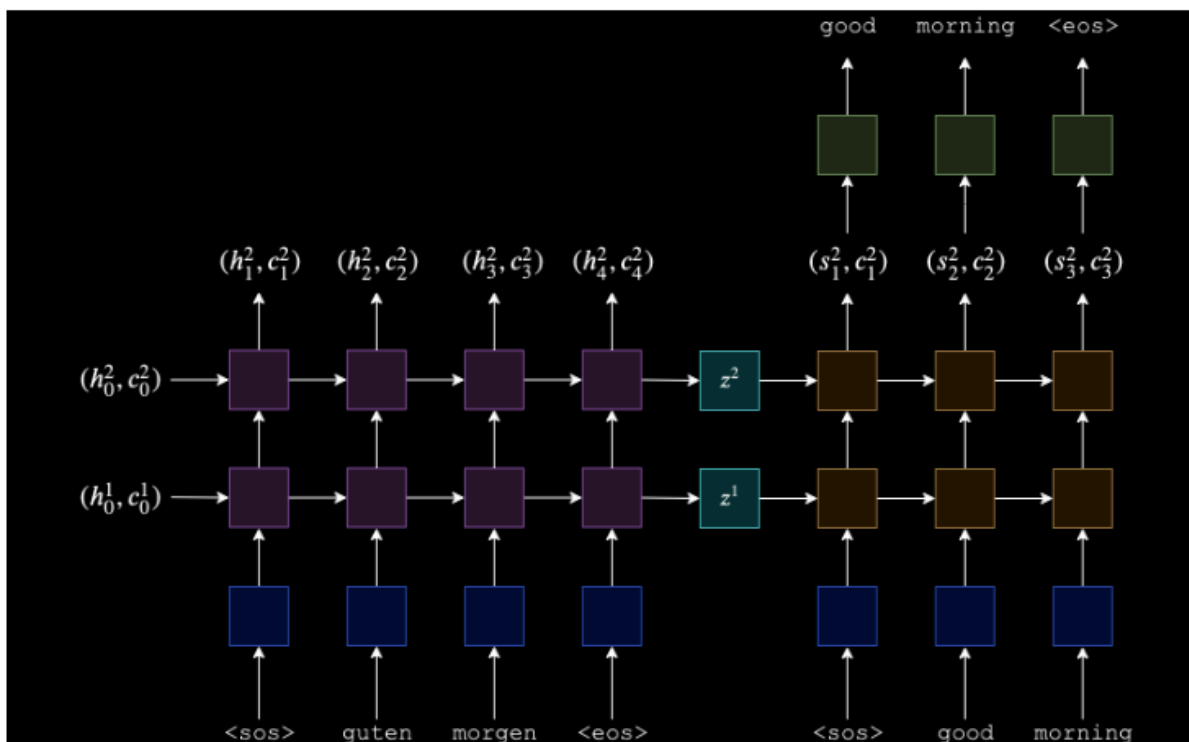
```
pred = self.out(output.squeeze(dim=0))

# should have a shape [batch_size, n_tokens]

return pred, hidden
```

Наконец, энкодер и декодер могут быть объединены в класс Seq2Seq. Данный класс работает сразу с предложениями как в исходном, так и в целевом языке.

Иллюстрация всей модели:



```
import random

class Seq2Seq(nn.Module):
```

```
def __init__(self, encoder, decoder):  
    super().__init__()   
  
    self.encoder = encoder  
  
    self.decoder = decoder  
  
    assert encoder.hid_dim == decoder.hid_dim, "encoder and  
decoder must have same hidden dim"  
  
    assert (  
        encoder.n_layers == decoder.n_layers  
    ), "encoder and decoder must have equal number of layers"  
  
def forward(self, src, trg, teacher_forcing_ratio=0.5):  
    # src has a shape of [src_seq_len, batch_size]  
  
    # trg has a shape of [trg_seq_len, batch_size]  
  
    # teacher_forcing_ratio is probability to use teacher  
forcing, e.g. if  
  
    # teacher_forcing_ratio is 0.75 we use ground-truth inputs  
75% of the time  
  
    batch_size = trg.shape[1]  
  
    trg_len = trg.shape[0]  
  
    trg_vocab_size = self.decoder.n_tokens  
  
    # tensor to store decoder predictions  
  
    preds = []
```

```
# Last hidden state of the encoder is used as
# the initial hidden state of the decoder.

hidden = self.encoder(src)

# First input to the decoder is the token.

input = trg[0, :]

for i in range(1, trg_len):
    pred, hidden = self.decoder(input, hidden)
    preds.append(pred)
    teacher_force = random.random() < teacher_forcing_ratio
    _, top_pred = pred.max(dim=1)
    input = trg[i, :] if teacher_force else top_pred
#формируем input для следующего состояния декодера

return torch.stack(preds)
```

Обучение модели Seq2Seq

Процедура обучения модели имеет мало отличий от предыдущих занятий. Также необходимо определить `torch.device`. Она используется, чтобы указать, на каком устройстве будут происходить вычисления: на GPU или на CPU.

Обратимся к функции `torch.cuda.is_available()`, которая вернет True, если был обнаружен графический процессор:


```
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

enc = Encoder(len(src_vocab), emb_dim=256, hid_dim=512, n_layers=2,
dropout=0.5)

dec = Decoder(len(trg_vocab), emb_dim=256, hid_dim=512, n_layers=2,
dropout=0.5)

model = Seq2Seq(enc, dec).to(device)
```

Оценим количество параметров в модели:

```
def count_parameters(model):

    return sum(p.numel() for p in model.parameters() if
p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable
parameters')
```

```
>>> The model has 13,916,175 trainable parameters
```

В функции потерь необходимо указать, что некоторые токены не стоит учитывать при подсчете ошибки. Для этого укажем `ignore_index` равный `<pad>`:

```
optimizer = torch.optim.Adam(model.parameters())

criterion =
nn.CrossEntropyLoss(ignore_index=trg_vocab[pad_token])
```

Наконец, обучим нашу модель. Не забывайте, что для оценки на отложенной выборке, модель необходимо перевести в режим inference с помощью `model.eval()`. Также

будем использовать `with torch.no_grad()`, чтобы гарантировать, что градиенты не вычисляются на этапе валидации.

```
from torch.nn.utils import clip_grad_norm_

from tqdm.auto import tqdm, trange

n_epochs = 5
clip = 1

for epoch in trange(n_epochs, desc="Epochs"):

    model.train()

    train_loss = 0

    for src, trg in tqdm(train_dataloader, desc="Train",
leave=False):

        # Use model to get prediction and compute loss using
criterion.

        # After you've computed loss, zero gradients, run backprop,
clip

        # gradients and update model with optimizer.

        src, trg = src.to(device), trg.to(device)

        output = model(src, trg)

        output = output.view(-1, output.shape[-1])

        trg = trg[1:].view(-1)

        loss = criterion(output, trg)

        optimizer.zero_grad()
```

```
    loss.backward()

    clip_grad_norm_(model.parameters(), clip)

    optimizer.step()

    train_loss += loss.item()

train_loss /= len(train_dataloader)

print(f"Epoch {epoch} train loss = {train_loss} ")

model.eval()

val_loss = 0

with torch.no_grad():

    for src, trg in tqdm(val_dataloader, desc="Val",
leave=False):

        # Once again compute model prediction and loss, but don't
        # try and update model parameters with it.
        # Just use it for model evaluation.

        src, trg = src.to(device), trg.to(device)

        output = model(src, trg)

        output = output.view(-1, output.shape[-1])

        trg = trg[1:].view(-1)

        loss = criterion(output, trg)
```

```
val_loss += loss.item()

val_loss /= len(val_dataloader)

print(f"Epoch {epoch} val loss = {val_loss} ")
```

Epochs: 100%  5/5 [01:55<00:00, 23.34s/it]

Epoch 0 train loss = 5.21652292786983

Epoch 0 val loss = 4.746980428695679

Epoch 1 train loss = 4.684973206436425

Epoch 1 val loss = 4.383065342903137

Epoch 2 train loss = 4.440019795769139

Epoch 2 val loss = 4.194343090057373

Epoch 3 train loss = 4.243898270422952

Epoch 3 val loss = 3.9665597677230835

Epoch 4 train loss = 4.079531184413977

Epoch 4 val loss = 3.9135063886642456

Домашнее задание. Подумать, какие у данной архитектуры есть «бутылочные горлышки».

Оценим качество перевода визуально. Для этого воспользуемся первыми десятью элементами из валидационной выборки:

```
trg_itos = trg_vocab.get_itos()

model.eval()

max_len = 50

with torch.no_grad():

    for src, trg in val_data[:10]:

        encoded = encode(src, src_vocab)[::-1]

        encoded = torch.tensor(encoded)[:, None].to(device)

        hidden = model.encoder(encoded)

        pred_tokens = [trg_vocab[sos_token]]

        for _ in range(max_len):

            decoder_input =
torch.tensor([pred_tokens[-1]]).to(device)

            pred, hidden = model.decoder(decoder_input, hidden)

            _, pred_token = pred.max(dim=1)

            if pred_token == trg_vocab[eos_token]:

                # Don't add it to prediction for cleaner output.

                break

        pred_tokens.append(pred_token.item())
```

```
print(f"src: '{src.rstrip().lower()}'")  
  
print(f"trg: '{trg.rstrip().lower()}'")  
  
print(f"pred: '{' '.join(trg_itos[i] for i in  
pred_tokens[1:])}'")  
  
print()
```

```
>>> src: 'eine gruppe von männern lädt baumwolle auf einen lastwagen'  
trg: 'a group of men are loading cotton onto a truck'  
pred: 'a group of a woman are a a the street .'
```

```
src: 'ein mann schläft in einem grünen raum auf einem sofa.'  
trg: 'a man sleeping in a green room on a couch.'  
pred: 'a group of a woman in a blue and a a .'
```

```
src: 'ein junge mit kopfhörern sitzt auf den schultern einer frau.'  
trg: 'a boy wearing headphones sits on a woman's shoulders.'  
pred: 'a woman and a woman are a a a a a .'
```

```
src: 'zwei männer bauen eine blaue eisfischerhütte auf einem  
zugefrorenen see auf'  
trg: 'two men setting up a blue ice fishing hut on an iced over lake'  
pred: 'a group of a woman are a a a the street .'
```

```
src: 'ein mann mit beginnender glatze, der eine rote rettungsweste  
trägt, sitzt in einem kleinen boot.'
```

trg: 'a balding man wearing a red life jacket is sitting in a small boat.'

pred: 'a group of a woman in a blue and and a a and a a a a .'

src: 'eine frau in einem rotem mantel, die eine vermutlich aus asien stammende handtasche in einem blauen farbton hält, springt für einen schnappschuss in die luft.'

trg: 'a lady in a red coat, holding a bluish hand bag likely of asian descent, jumping off the ground for a snapshot.'

pred: 'a group of a woman in a a and a a and a a a a a a a a a a .'

src: 'ein brauner hund rennt dem schwarzen hund hinterher.'

trg: 'a brown dog is running after the black dog.'

pred: 'a woman and a woman are a a a .'

src: 'ein kleiner junge mit einem giants-trikot schwingt einen baseballschläger in richtung eines ankommenden balls.'

trg: 'a young boy wearing a giants jersey swings a baseball bat at an incoming pitch.'

pred: 'a group of a woman in a and a and a a and a a a .'

src: 'ein mann telefoniert in einem unaufgeräumten büro'

trg: 'a man in a cluttered office is using the telephone'

pred: 'a woman in a a and a a .'

src: 'eine lächelnde frau mit einem pfirsichfarbenen trägershirt hält ein mountainbike'

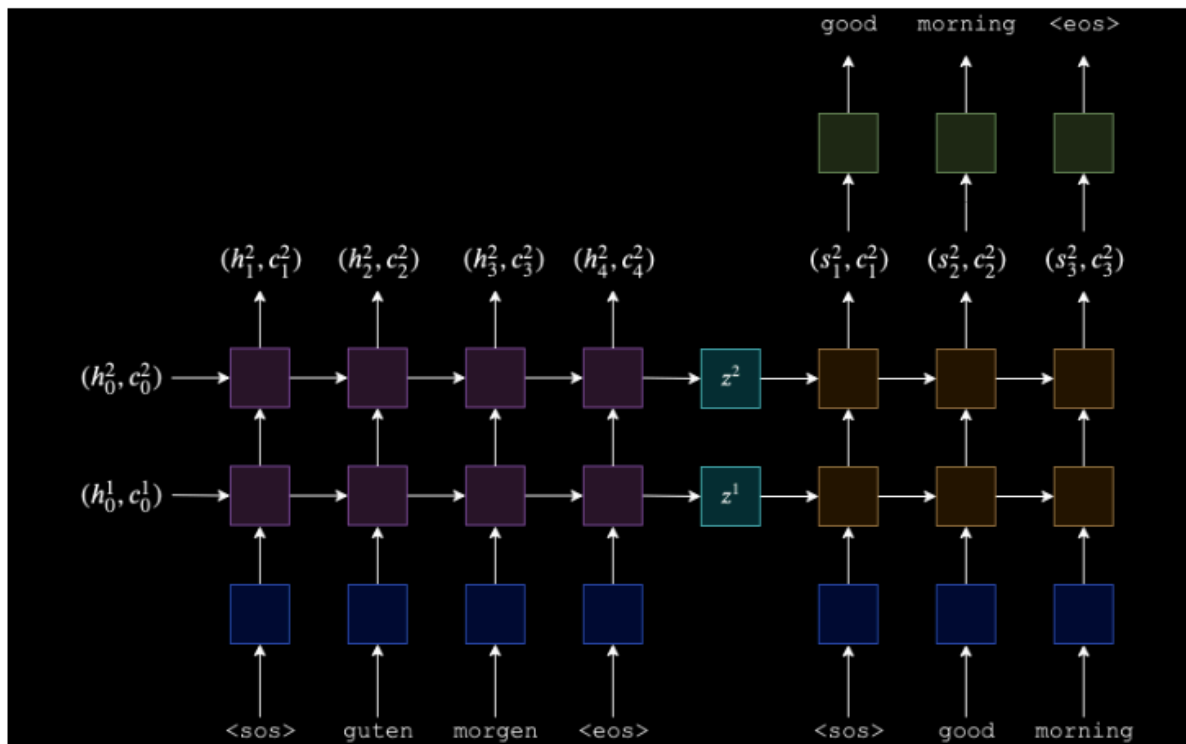
trg: 'a smiling woman in a peach tank top stands holding a mountain bike'

pred: 'a woman and a woman in a blue shirt and a .'

Выводы:

- Архитектура энкодер-декодер — общий подход к работе с данными сложной структуры.
- Рекуррентные сети способны улавливать последовательную структуру данных благодаря своей архитектуре.
- Даже сложные задачи могут быть декомпозированы на несколько простых подзадач.

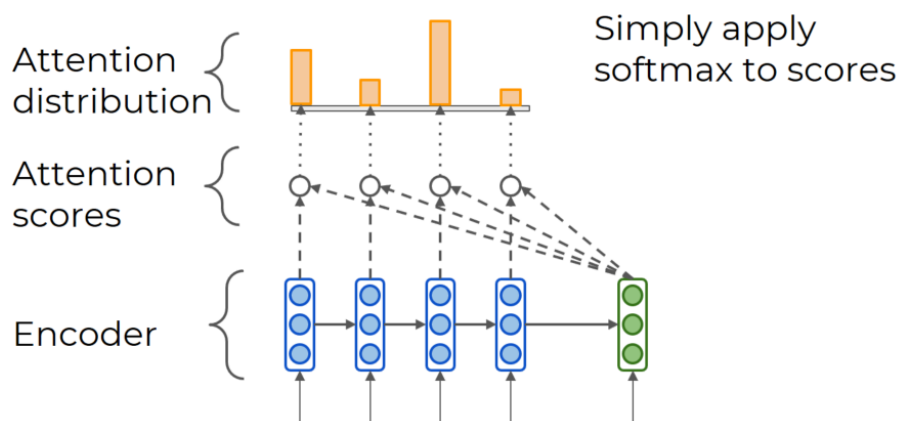
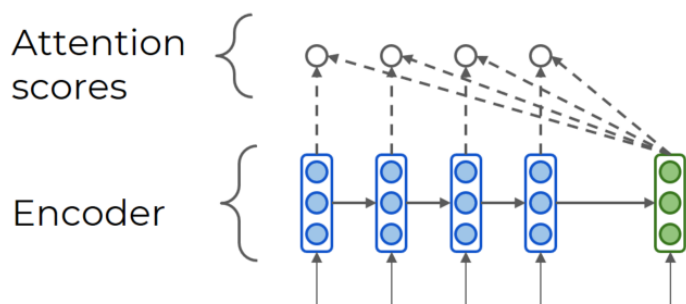
Рассмотрим структуру нашей Seq2Seq модели еще раз:

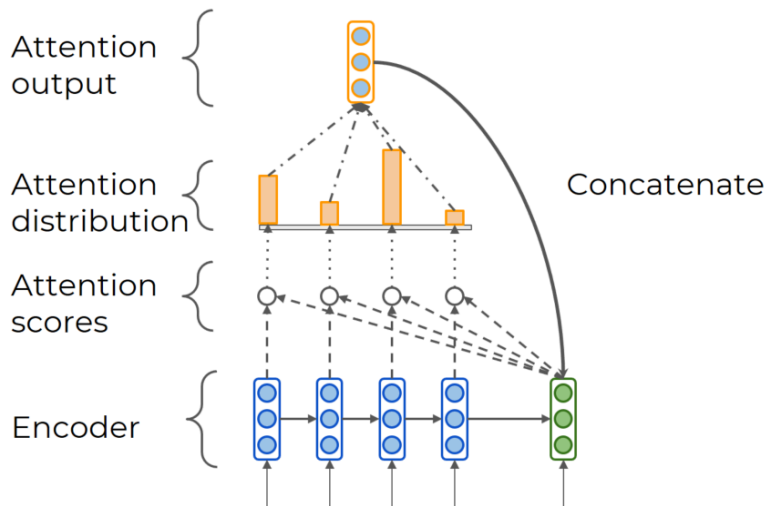
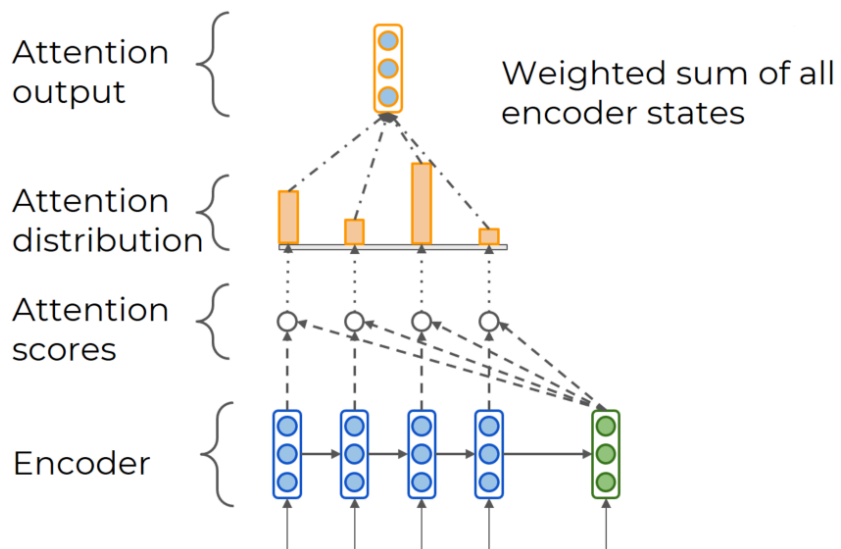


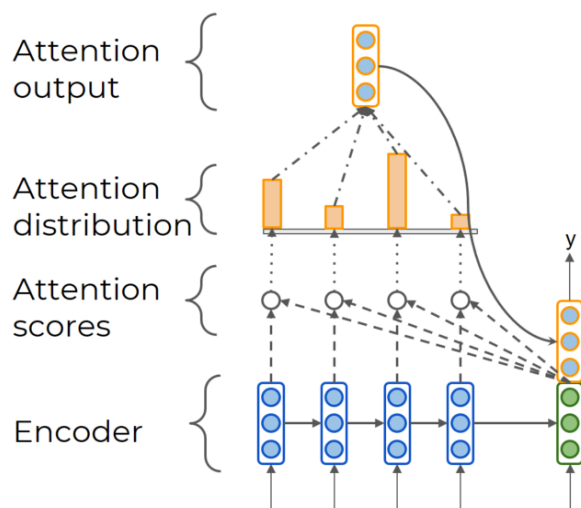
Механизм внимания в нейронных сетях (Attention)

Предложение для перевода может быть длинным, а также иметь причастные и деепричастные обороты. За это отвечает всего один вектор скрытого состояния. Для больших текстов это «бутылочное горлышко» — модель, которая забывает часть слов.

Нам же нужно что-то, что имело бы информацию обо всем предложении полностью.
Для этого появился механизм внимания:







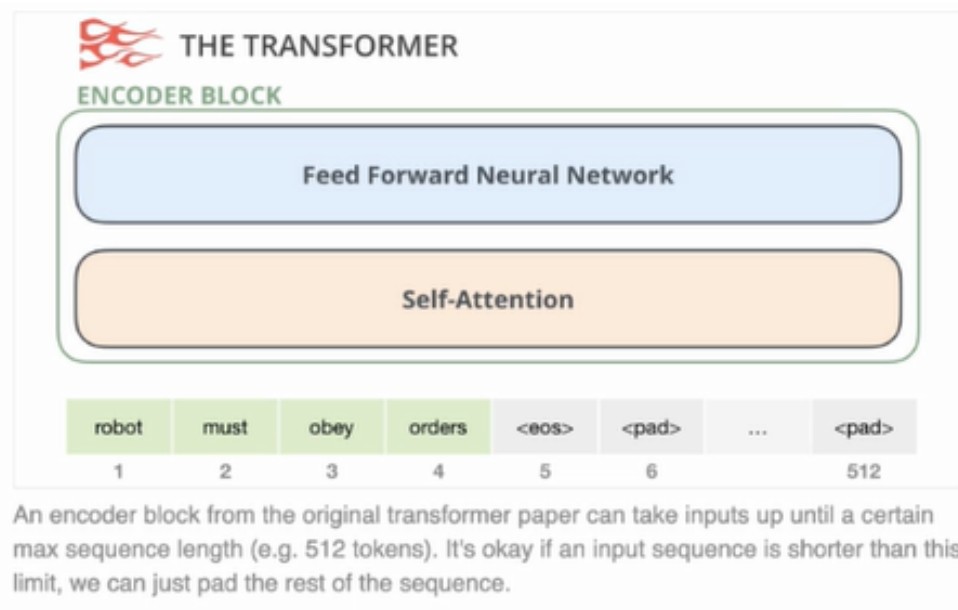
BERT в задаче классификации текстов

Воспользуемся предобученной языковой моделью BERT, чтобы решить уже знакомую нам задачу классификации текстов с использованием датасета SST-2.

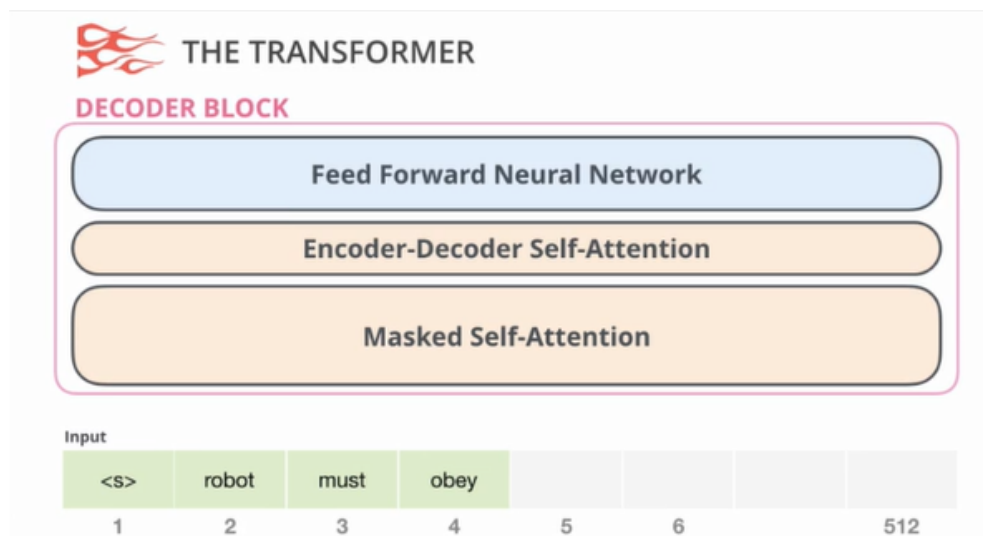


В 2015 году статья Attention is all you need представила архитектуру Трансформера. Трансформер также состоит из части энкодера и части декодера.

Блок энкодера в Трансформере:



Блок декодера:

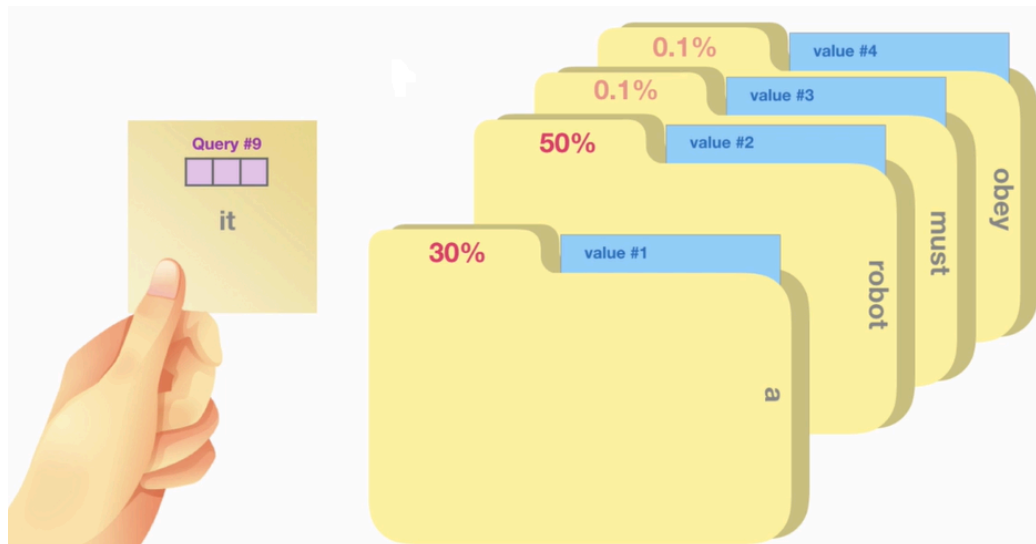


Модели BERT и GPT созданы на основе Трансформера. В BERT несколько энкодеров находятся в стеке. В GPT декодеры находятся в стеке.


















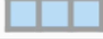

В модели BERT есть механизм Self Attention.

Допустим, мы стоим на некотором токене Query (запрос). Мы хотим узнать, как этот токен соотносится с другими словами в предложении. Мы делаем скалярное

произведение между Query и всеми остальными токенами — Key. Делаем Softmax над этим скалярным произведением и получаем некоторые Attention Scores, некоторые вероятности.



В результате мы «перевзвешиваем» наши токены:

Word	Value vector	Score	Value X Score
<S>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

Посмотрим, как можно применить BERT к задаче классификации предложений.

В BERT очень хорошие эмбединги.

Классифицировать текст можно двумя способами:

1. Взять эмбединги из нашего предложения и поместить их в классификационную модель, например, в логистическую регрессию.
2. Построить над моделью BERT голову (head), сделать линейный слой, отображающий токены класса, и обучить BERT, чтобы он предсказывал вероятность того или иного класса.

Домашнее задание. Попробовать самостоятельно реализовать второй вариант классификации.

Рассмотрим первый способ. Процесс решения задачи можно разделить на два основных этапа:

- DistilBERT обрабатывает текст отзывов и представлен в виде вектора фиксированной размерности (768). DistilBERT — упрощенная версия модели BERT, полученная с помощью техники дистилляции.
- Простая логистическая регрессия использует полученные признаковые описания для решения задачи классификации.

Пример уже знакомых вам данных:

sentence	label
a stirring , funny and finally transporting re imagining of beauty and the beast and 1930s horror films	1
apparently reassembled from the cutting room floor of any given daytime soap	0
they presume their audience won't sit still for a sociology lesson	0
this is a visually stunning rumination on love , memory , history and the war between art and commerce	1
jonathan parker 's bartleby should have been the be all end all of the modern office anomie films	1

Также понадобится библиотека Transformers, которая предоставляет доступ к огромному числу предобученных моделей для работы с текстом:

```
!pip install transformers
```

Подключаем все нужные библиотеки и наш датасет:

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

import torch

import transformers as ppb

import warnings

from sklearn.base import BaseEstimator
from sklearn.metrics import roc_auc_score, roc_curve, accuracy_score

import matplotlib.pyplot as plt

warnings.filterwarnings('ignore')
```

```
df = pd.read_csv(

    'https://github.com/clairett/pytorch-sentiment-classification/raw/master/data/SST2/train.tsv',

    delimiter='\t',

    header=None

)
```

Нам нужно предсказать сентименты — хороший отзыв или плохой.

Для простоты рассмотрим только первые 2000 объектов:

```
batch = df[:2000]
```

Выборку можно считать сбалансированной:

```
batch[1].value_counts()
```

```
>>> 1      1041
```

```
0       959
```

```
Name: 1, dtype: int64
```

Загрузим предобученную модель:

```
# For DistilBERT:

model_class, tokenizer_class, pretrained_weights = (

    ppb.DistilBertModel,

    ppb.DistilBertTokenizer,

    'distilbert-base-uncased'

)

## Want BERT instead of distilBERT? Uncomment the following line:

# model_class, tokenizer_class, pretrained_weights = (ppb.BertModel,
# ppb.BertTokenizer, 'bert-base-uncased')

# Load pretrained model/tokenizer

tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
```



```
model = model_class.from_pretrained(pretrained_weights)
```

Токенайзер должен относиться к модели. Какую модель подключаем, такой токенайзер и закачиваем.

Посмотрим на модель:

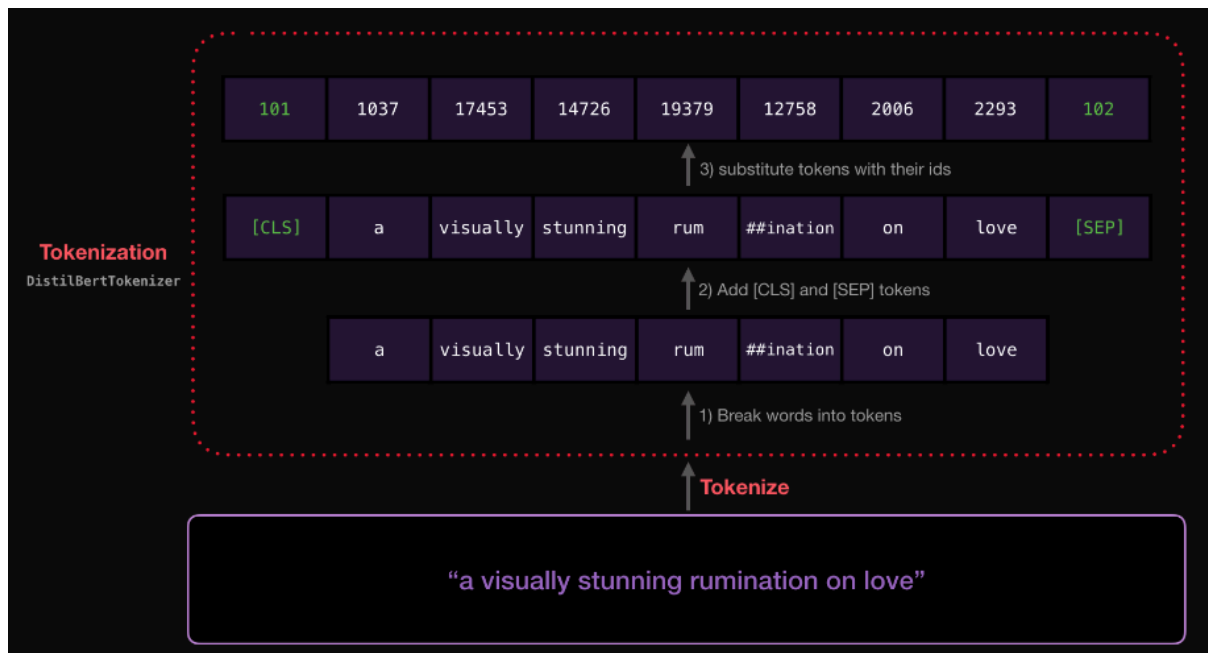
```
model
```

```
DistilBertModel(
  (embeddings): Embeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (transformer): Transformer(
    (layer): ModuleList(
      (0-5): 6 x TransformerBlock(
        (attention): MultiHeadSelfAttention(
          (dropout): Dropout(p=0.1, inplace=False)
          (q_lin): Linear(in_features=768, out_features=768, bias=True)
          (k_lin): Linear(in_features=768, out_features=768, bias=True)
          (v_lin): Linear(in_features=768, out_features=768, bias=True)
          (out_lin): Linear(in_features=768, out_features=768, bias=True)
        )
        (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (ffn): FFN(
          (dropout): Dropout(p=0.1, inplace=False)
          (lin1): Linear(in_features=768, out_features=3072, bias=True)
          (lin2): Linear(in_features=3072, out_features=768, bias=True)
          (activation): GELUActivation()
        )
        (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      )
    )
  )
)
```

Для токенизации обратимся к уже существующему токенизатору. Именно он должен использоваться для токенизации текста, если планируете использовать предобученную модель BERT.

```
tokenized = batch[0].apply((lambda x: tokenizer.encode(x,
```

```
add_special_tokens=True)))
```



С помощью padding приведем все предложения к единой длине. В общем случае стоит делать это для каждого батча отдельно, но в данном случае мы преследуем максимальную простоту кода.

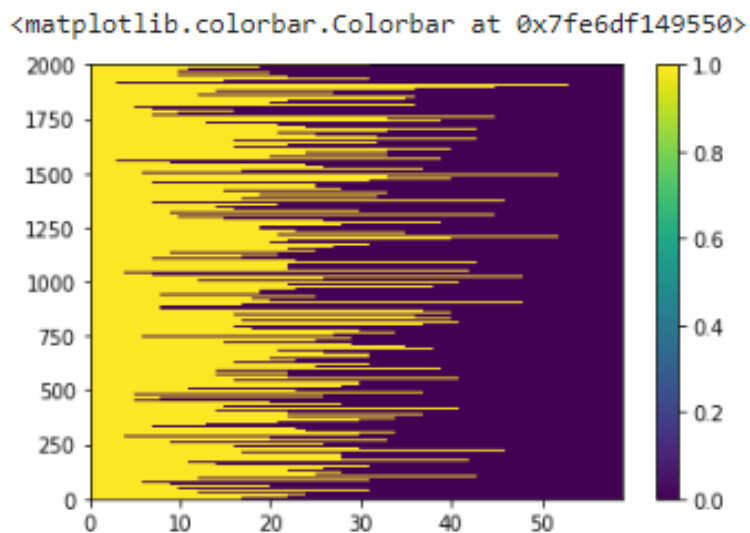
```
max_len = 0

for i in tokenized.values:
    if len(i) > max_len:
        max_len = len(i)

padded = np.array([i + [0]*(max_len-len(i)) for i in
tokenized.values])
```

Переменная padded представляет собой предобработанную версию датасета. Мы видим, что <PAD> токенов впрямь очень много:

```
plt.pcolormesh(padded>0)
plt.colorbar()
```



Также необходимо указать, какие из токенов не стоит учитывать при вычислении Attention. Для этого воспользуемся маской:

```
attention_mask = np.where(padded != 0, 1, 0)
attention_mask.shape
```

```
>>> (2000, 59)
```

Наконец, предобработаем текст с помощью BERT:

```
input_ids = torch.tensor(padded)
attention_mask = torch.tensor(attention_mask)
model.eval()
```

```
with torch.no_grad():

    last_hidden_states = model(input_ids,
                               attention_mask=attention_mask)
```

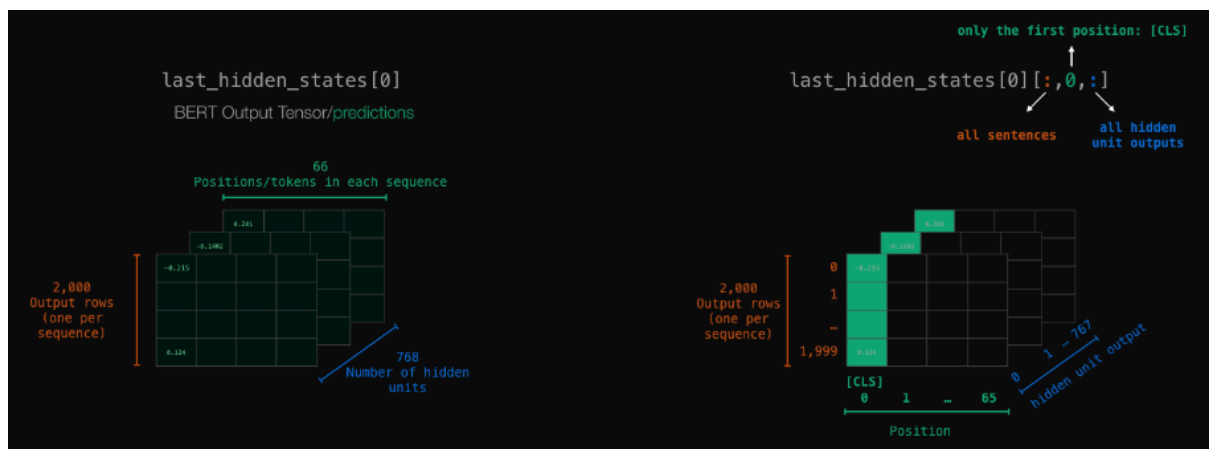
Посмотрим, что из себя представляет последний скрытый слой:

```
last_hidden_states[0].shape
```

```
>>> torch.Size([2000, 59, 768])
```

2000 предложений. Максимальная длина предложения 59. Каждое предложение состоит из токенов, каждый токен — вектор длиной 768.

Нас интересует лишь один вектор для каждого текста. Он стоит на первой позиции и соответствует [CLS] (classification) токenu.



Вытаскивать cls токен будем с помощью кода:

```
last_hidden_states[0][:,0,:]
```

Посмотрим на последний скрытый слой:

```
last_hidden_states
```

Часть его представлена ниже:

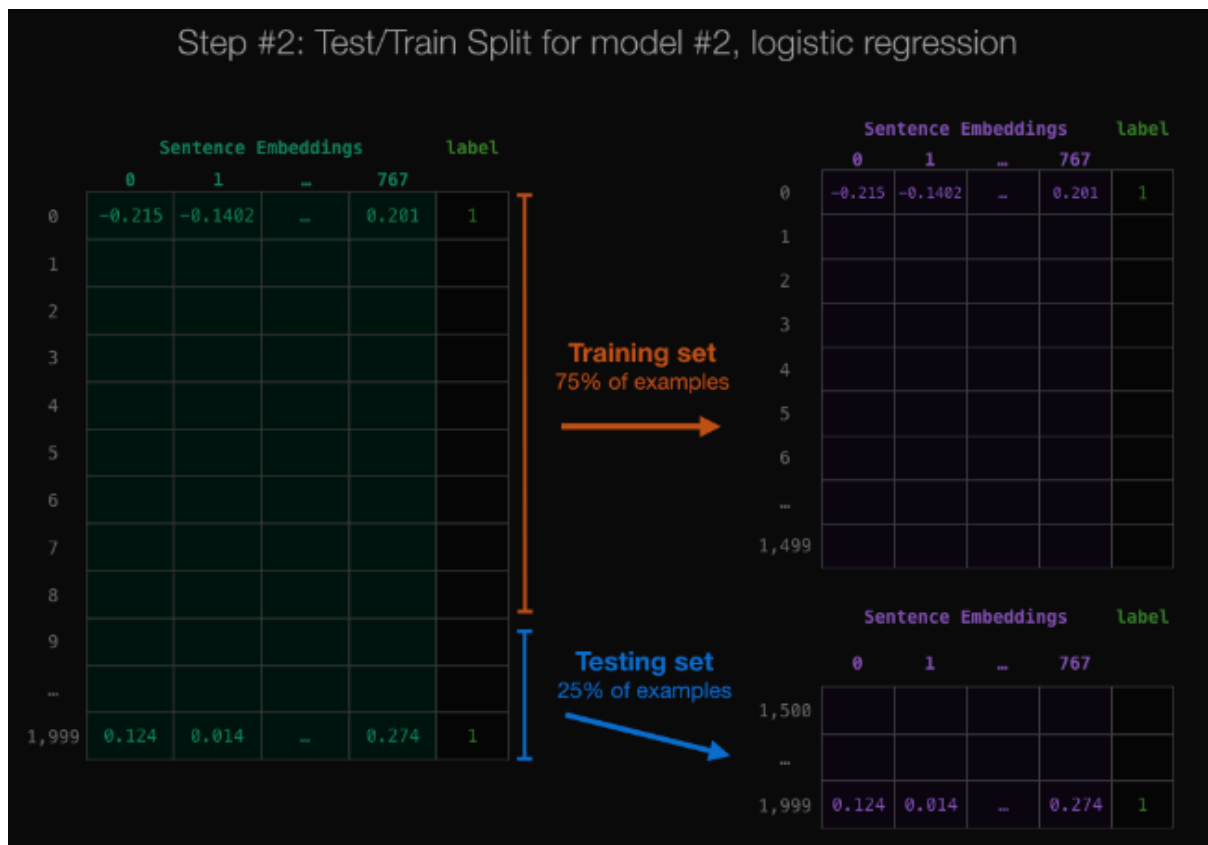
```
BaseModelOutput(last_hidden_state=tensor([[-0.2159, -0.1403,  0.0083, ..., -0.1369,  0.5867,  0.2011],
      [-0.2471,  0.2468,  0.1008, ..., -0.1631,  0.9349, -0.0715],
      [ 0.0558,  0.3573,  0.4140, ..., -0.2430,  0.1770, -0.5080],
      ...,
      [-0.0165,  0.1179,  0.3512, ..., -0.2401,  0.2722, -0.1750],
      [ 0.0961,  0.0667,  0.3147, ..., -0.3277,  0.3556, -0.2135],
      [ 0.0454,  0.0519,  0.3168, ..., -0.2880,  0.1844, -0.1042]],
      ...,
      [[-0.1726, -0.1448,  0.0022, ..., -0.1744,  0.2139,  0.3720],
      [ 0.0022,  0.1684,  0.1269, ..., -0.1888, -0.0195, -0.0283],
      [ 0.0257, -0.2458,  0.0717, ..., -0.4339,  0.1622,  0.0133],
      ...,
      [ 0.0505, -0.0493,  0.0463, ..., -0.0448, -0.0540,  0.3136],
      [-0.2128, -0.1907, -0.0215, ...,  0.0139, -0.2433, -0.0202],
      [-0.1310, -0.1693,  0.1019, ..., -0.0859, -0.1770, -0.0872]],
      ...,
      [[-0.0506,  0.0720, -0.0296, ..., -0.0715,  0.7185,  0.2623],
      [ 0.0536,  0.3136, -0.0598, ...,  0.2676,  0.8668, -0.3380],
      [ 0.3792,  0.2792,  0.0237, ...,  0.0343,  0.4272,  0.1680],
      ...,
      [ 0.2314,  0.2427,  0.1030, ..., -0.0840,  0.2322,  0.0613],
      [ 0.0741,  0.0465,  0.1083, ...,  0.0530,  0.1971,  0.0233],
      [ 0.1042,  0.3139,  0.0768, ..., -0.0301,  0.1052, -0.0780]],
      ...],
```

```
features = last_hidden_states[0][:,0,:].numpy()

labels = batch[1]
```

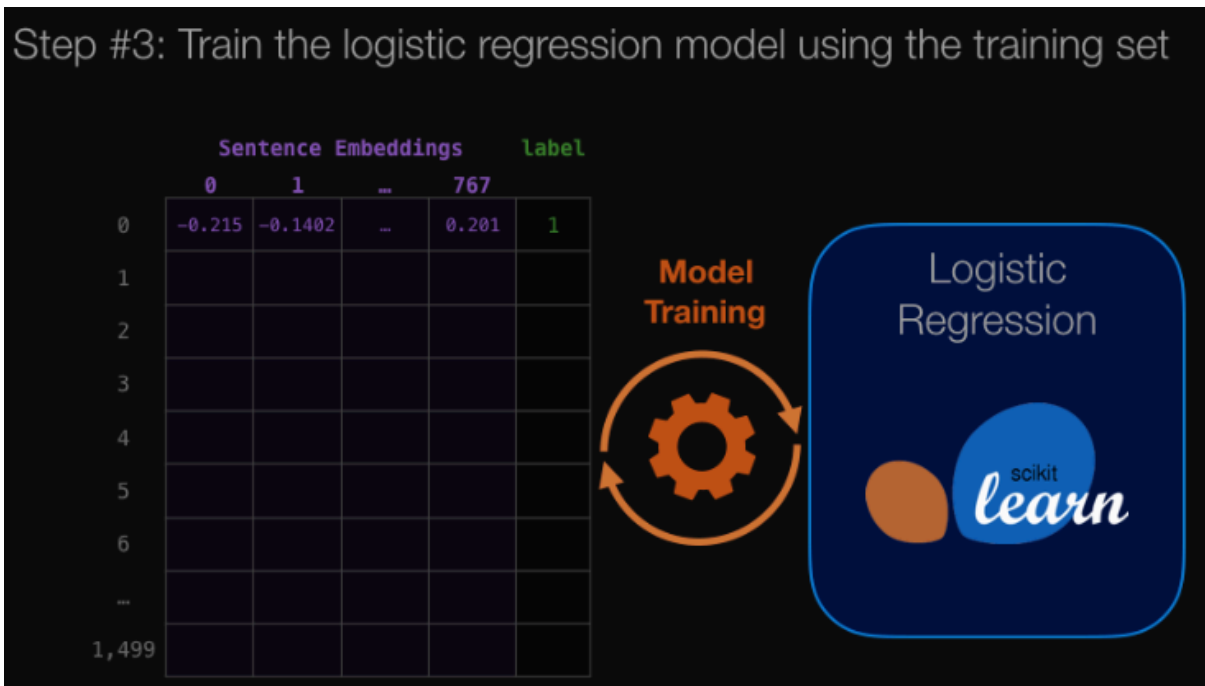
Теперь обучим классическую логистическую регрессию из sklearn. Перед этим разобьем выборку на train и test:

```
train_features, test_features, train_labels, test_labels =
train_test_split(features, labels)
```



```
lr_clf = LogisticRegression()

lr_clf.fit(train_features, train_labels)
```



Оценим качество классификации:

```
lr_clf.score(train_features, train_labels)
```

```
>>> 0.9133333333333333
```

```
lr_clf.score(test_features, test_labels)
```

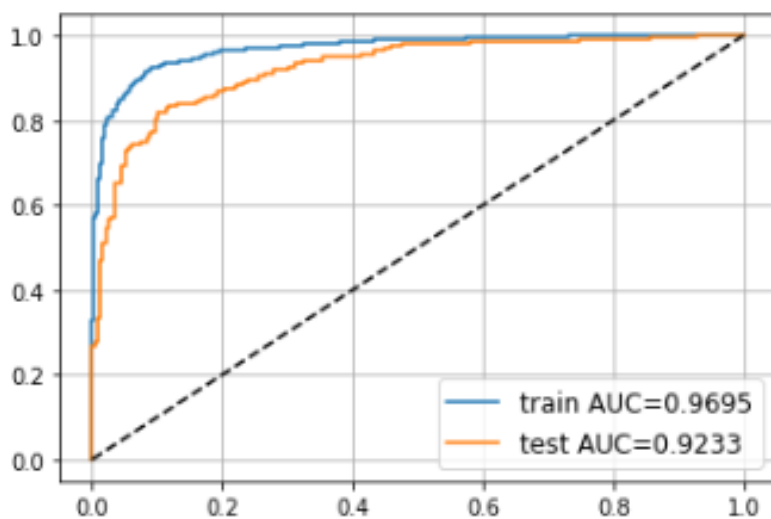
```
>>> 0.85
```

```
def visualize_roc(model, X_train, X_test, y_train, y_test):
    for data_name, X, y, model in [
        ('train', X_train, y_train, model),
        ('test', X_test, y_test, model)
```

```
]:  
  
    if isinstance(model, BaseEstimator):  
        proba = model.predict_proba(X)[:, 1]  
  
    elif isinstance(model, nn.Module):  
        proba = model(X).detach().cpu().numpy()[:, 1]  
  
    else:  
        raise ValueError('Unrecognized model type')  
  
    auc = roc_auc_score(y, proba)  
  
    plt.plot(*roc_curve(y, proba)[:2], label='%s AUC=%.4f' %  
(data_name, auc))  
  
    plt.plot([0, 1], [0, 1], '--', color='black',)  
    plt.legend(fontsize='large')  
    plt.grid()
```

Посмотрим, как выглядит ROC-кривая:

```
visualize_roc(lr_clf, train_features, test_features, train_labels,  
test_labels)
```

Полученное качество достаточно высокое, и модель не показывает явных признаков переобучения. При этом мы использовали лишь подвыборку 2000 объектов (до разбиения), а не весь набор данных.

Выводы

- Механизм внимания (Attention) позволяет оценивать (и выделять) значимость различных элементов последовательностей друг для друга.
- Механизм внимания может быть использован как в задачах обработки текстов, так и при работе с другими данными, которые обладают сложной структурой (изображения, видео, графы и др.).
- В настоящий момент архитектуры, которые основаны на чуть измененном механизме внимания (Self-Attention), показывают state-of-the-art результаты во множестве задач.
- Предобученные модели позволяют использовать полученные в других задачах «знания». Это значительно упрощает решение прикладных задач.
- Многие подходы похожи друг на друга и исходят из похожих (и достаточно простых) предположений.

Дополнительные материалы для самостоятельного изучения

1. [Attention is all you need](#)

2. [Neural Machine Translation by Jointly Learning to Align and Translate](#)
3. [The Illustrated Transformer](#)
4. [Neural Machine Translation by Jointly Learning to Align and Translate](#)
5. [Sequence to Sequence Learning with Neural Networks](#)
6. [Корпус Multi30k](#)
7. [Техника дистилляции](#)