KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 8 урок > Тяжелые языковые модели для высокоточной работы: BERT и трансформеры

> Оглавление

> Представление текста (Tokenization)

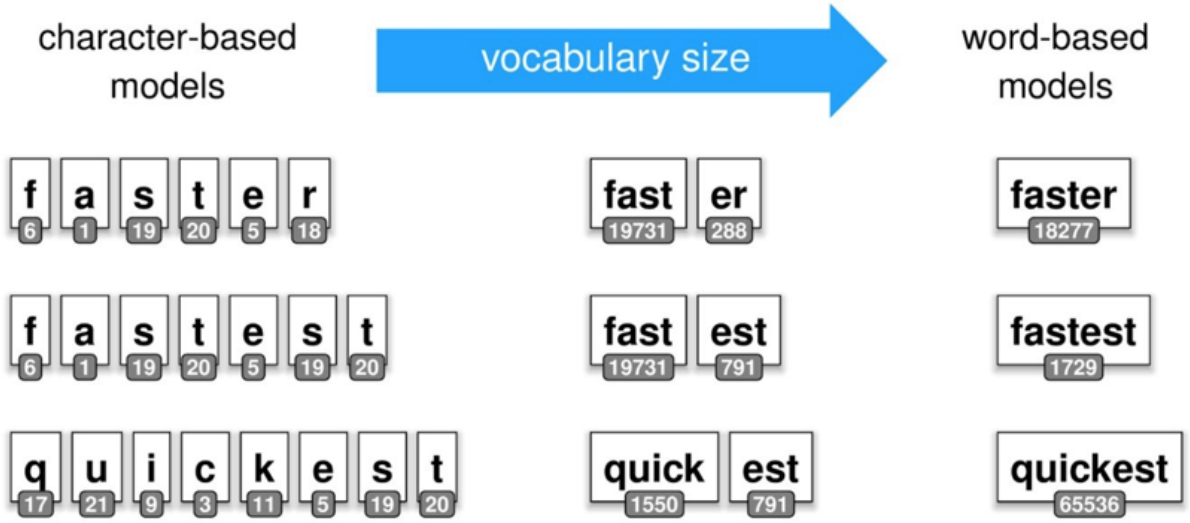
Как вы помните из лекции про эмбеддинги, у Word2Vec есть острая проблема, связанная с отсутствием информации о числе, роде, времени и прочих признаках слов. Частично она решается с помощью FastText за счёт разбиения на N-граммы. Однако если задуматься, то N здесь может быть разным для

разных смысловых частей — сам корень слова может быть длинным, а столь необходимое окончание может характеризоваться всего одной буквой или одним символом. Более того, при матчинге товаров также появляется острая проблема сопоставления по каким-то специфическим наборам символов, означающим характеристики или, может даже, артикулы (какая-нибудь цифробуквенная последовательность из 8 символов нарезается на странные триграммы, каждая из которых не несёт как такового смысла).

Однако во время разбора эмбеддингов мы упустили char-level (символьные) модели. В них создаются эмбеддинги отдельных букв, цифр или знаков, а затем с помощью, например, свёрток агрегируется информация о соседних символах. Такие модели в настоящее время применяются редко, поэтому о них мы и не говорим. На картинке снизу слева изображён пример такой токенизации, т.е. разбиения предложения, или в данном случае слова, на отдельные объекты, которым ставится в соответствие уникальный идентификатор, или номер.

Основное преимуществo character-base токенизации — это размер словаря. В данном случае он минимален — на весь алфавит одного языка будет максимум 40 символов. Их число возрастёт до 80, если мы возьмём верхний и нижний регистры. Если же мы добавим сюда ещё знаки и цифры, то достигнем 100.

В то же время модели, работающие с целыми словами (изображены ниже справа), напротив, имеют сотни тысяч буквенных последовательностей в словаре, т.е. большую часть реальных слов в языке. В идеале хотелось бы найти такой подход, который лежал бы где-то посередине и при этом не скатывался в заранее определённые по размеру N-граммы.



Демонстрация символьных и WordPiece токенизаций

В трансформерах часто используются 2 вида токенизации, которые очень схожи. Это Byte Pair Encoding (BPE) и WordPiece. Мы не будем вдаваться в технические подробности реализации этих методов: обозначим лишь основные идеи, лежащие в их основе.

Для начала создадим базовый словарь, в котором будут представлены все отдельные символы, какие только есть в выборке. Если текст мультиязычный, то в такой словарь будут добавлены несколько алфавитов. Например, даже в моделях для русского языка, очевидно, встречается латиница. Буквы опционально можно приводить к нижнему регистру для сокращения вариативности, в целом это практически не влияет на итоговое качество модели, построенной на таких токенах. Далее необходимо для каждого объекта словаря построить модель совместной встречаемости с другими объектами словаря. На первом этапе будут получаться просто пары символов, т.е. то, какие буквы чаще или реже идут за другими. Соответственно, максимальную по частоте пару можно добавить в словарь, расширив его таким образом. Итеративно повторяя процесс объединения самых частых пар, можно наполнить словарь до предварительно заданного размера. Пример такого разбиения для английского языка можно наблюдать в центре изображения выше: видно, что "fast" является одинаковым корнем и в сравнительной, и в превосходной форме слова. При этом окончание "est", как раз отвечающее за превосходную форму, будет использоваться и в других словах. Его эмбеддинг, если забегать вперёд, будет формировать именно такой смысл у слова, стоящего перед ним. То есть корень несёт общий смысл слова, а окончание и суффикс — уточняют его.

В качестве примера возьмём пару заголовков товаров из онлайн-магазина и применим токенизатор для русского BERT. На самом деле тут два разных токенизатора: один приводит к нижнему регистру, другой нет. Здесь две решётки (##) означают, что токен является продолжением слова и перед ним есть как минимум один символ до пробела. Видно, что какие-то специфические характеристики разрезаются на разные токены, какие-то остаются целыми, но даже цвет товара (чёрный в данном случае) не образует единый токен.

```
tok.tokenize('Монитор Iiyama 21.5" ProLite E2282HS-B1 черный')
```

```
['М',
 '##они',
 '##тор',
 'I',
 '##iya',
 '##ma',
 '21',
 '.',
 '5',
 '"',
 'Pro',
 '##L',
 '##ite',
 'E',
 '##22',
 '##82',
 '##HS',
 '-',
 'B',
 '##1',
 'че',
 '##рный']
```

```
tok.tokenize('Плата монтажная сплош. для Altis 1800x1000мм Leg 047611')
```

```
['плата',
 'монтажн',
 '##ая',
 'спл',
 '##ош',
 '.',
 'для',
 'alt',
 '##is',
 '1800',
 '##x100',
 '##0м',
 '##м',
 'leg',
 '047',
 '##611']
```

Демонстрация токенизации двух предложений двумя разными токенизаторами

Но если обучить токенизатор под свои нужды на собственном корпусе данных (для задачи матчинга это в основном заголовки товаров и их параметры), то можно увидеть абсолютно иную картину (см. ниже):

```
['ноутбук', 'acer', 'aspire', '3', '(', 'a317', '-', '32', '-', 'p3', '##dh', ')', '(', 'intel', 'pentium', 'n5000',
'1100mhz', '/', '17', '.', '3', '"', '/', '1600x900', '/', '4gb', '/', '256gb', 'ssd', '/', 'dvd', 'нет', '/', 'inte
l', 'uhd', 'graphics', '605', '/', 'wi', '-', 'fi', '/', 'bluetooth', '/', 'endless', 'os', ')', 'nx', '.', 'hf', '##
2er', '.', '005', 'черный']
```

Здесь понятно, что означает практически каждый отдельный токен и какую смысловую нагрузку он несёт. Марка процессора, производитель (как "intel" "pentium"), объём памяти в гигабайтах, разрешение экрана и даже наличие bluetooth — всё это преобразуется в отдельные токены, которыми очень удобно оперировать. Кстати, тут очень легко проверить, переобучен токенизатор или нет (не слишком ли большой размер словаря у него). В случае переобучения в словарь начинают добавляться токены, которые как бы “заучивают” тренировочную выборку. Для примера рассмотрим артикул товара `a317`: в случае переобучения он практически весь был бы одним токеном, а это плохо. Целевая ситуация — это когда какие-то коды или артикулы всё еще разбиваются на разные составляющие по 2-3 буквы.

В случае BERT эмбеддинги строятся следующим образом и состоят из 3 компонент, которые складываются между собой:

1. Эмбеддинги токенов(смысловые);

2. Эмбеддинги сегментов подаваемого на вход BERT текста (важно для обучения);
3. Позиционные эмбеддинги.

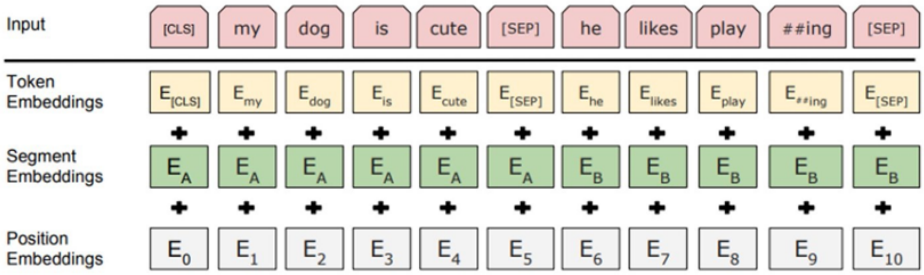


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

Первый эмбеддинг мы уже разобрали — это вектор для токена, который задан такой же матрицей, как и в Word2Vec. По сути механизм аналогичен: это маппинг токена в индекс в матрице, который выражен вектором, и он также обучается, т.е. меняет свои значения. Это бежевый ряд на изображении (token embeddings).

Сразу под ним идёт зеленый ряд, означающий эмбеддинг сегмента. Таких сегментов может быть всего несколько штук (чаще от 2 до 5). Про сегменты поговорим ближе к концу разбора BERT, пока стоит думать о них как о добавочных векторах, которые немного меняют основной эмбеддинг.

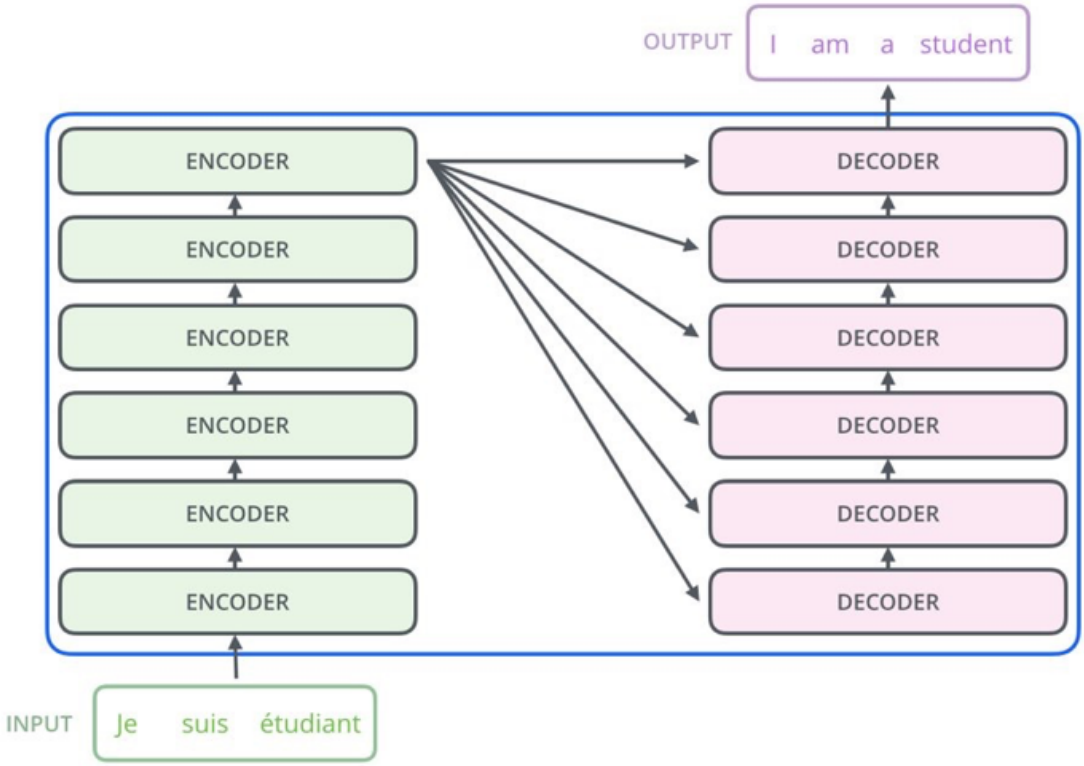
И последний, но не менее важный — позиционный эмбеддинг. Он строго привязан к номеру токена в предложении, и таким образом в модель подаётся информация о последовательности слов. Есть отдельный эмбеддинг для нулевого токена, для первого и т.д. Добавляя их к основному эмбеддингу, получаем обогащение информацией о расположении, что позволяет учитывать изменения смысла от словосочетаний, либо более сложных зависимостей вроде оборотов и даже сложноподчинённых предложений.

Все 3 эмбеддинга одинаковы по размерности вектора, поэтому их легко складывать покомпонентно. В классическом BERT используется размер эмбеддинга `768` (64x12). Эта цифра ещё несколько раз будет появляться в конспекте к этой лекции в качестве пояснения или примера. В верхнем ряду на изображении видно, как токенизируется предложение: здесь почти все токены являются полными словами, прямо как в Word2Vec, однако у одного из

последних слов справа есть окончание "##ing", которое указывает на причастие — в этом заключается несомненный плюс BPE токенизации. Также вы наверняка заметили, что в верхнем розовом ряду есть `[CLS]` токен в начале и `[SEP]` токен в середине между предложениями и в конце. Это специальные токены, которые также входят в словарь. `[CLS]` токен используется как агрегатор информации всего предложения. Он вбирает в себя всё необходимое, и после этого при обучении трансформера именно эмбеддинг этого самого первого токена используется как набор признаков для решения какой-либо задачи, например, классификации. По сути это эмбеддинг всего предложения или даже нескольких предложений разом. А `[SEP]` токен, как несложно понять, это SEParator, или разделитель, который просто отделяет предложения друг от друга, а также указывает на конец входной текстовой последовательности.

> Архитектура Трансформера

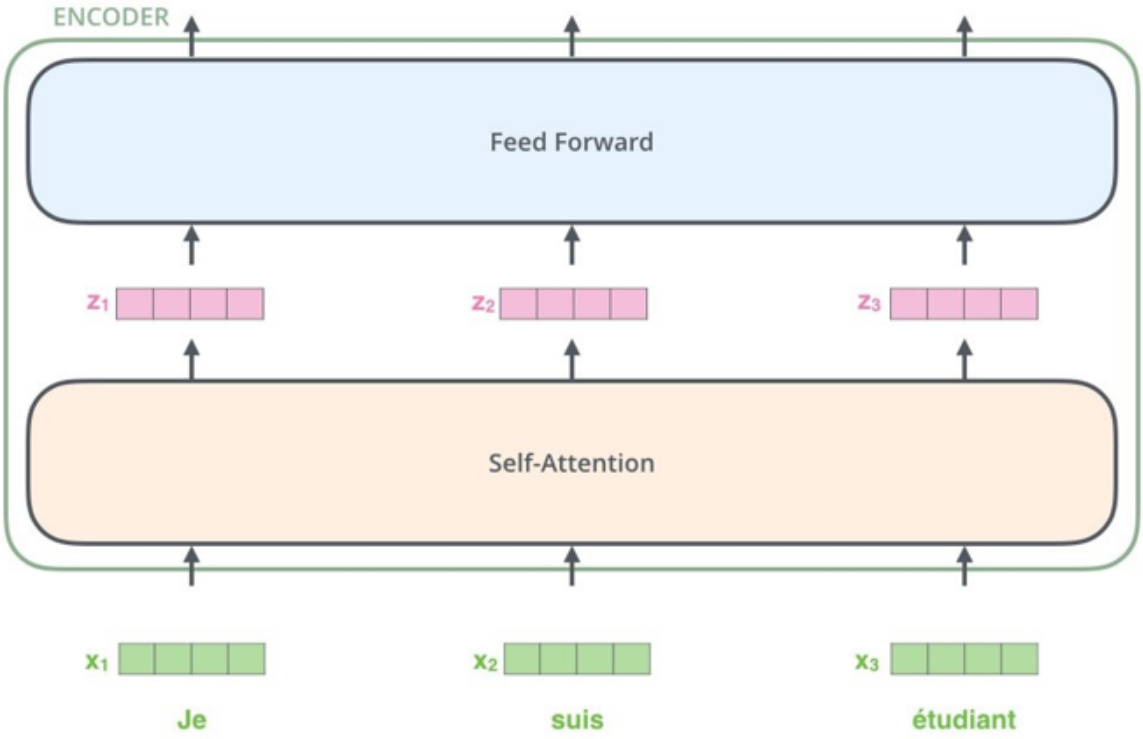
Классические трансформеры состоят из кодировщика (encoder), который отвечает за формирование осмысленного эмбеддинга, пригодного для решения некоторой задачи, и декодировщика (decoder), который как бы расшифровывает эти вектора в нечто осмысленное, например в текст. К примеру, так работают переводчики — сначала текст с одного языка кодируется набором векторов, после чего происходит генерация текста на втором языке, где в качестве “смысла” для генерации выступает закодированное энкодером представление. Именно такой пример представлен ниже.



Пример использования трансформеров для машинного перевода текстов

В самом же BERT, как представителе семейства трансформеров, есть только энкодер — никакого декодинга не происходит (буква E в названии как раз и означает encoder). Таким образом всю правую часть на изображении можно просто закрыть, оставив левую. Вообще BERT означает Bidirectional Encoder Representations from Transformers, т.е. это двунаправленный кодировщик эмбеддингов из трансформера. Здесь есть важное слово — “двунаправленный”. О его смысле мы и будем говорить далее.

Для начала рассмотрим архитектуру блока этого кодировщика. На первом изображении с иллюстрацией трансформера для машинного перевода вы могли заметить, что друг над другом выстроено несколько максимально похожих зелёных блоков. Действительно, каждый блок имеет одинаковый интерфейс на вход и выход. Для последовательности токенов длины N, каждый из которых представлен вектором размерности D, блок кодировщика выдаёт N преобразованных векторов размерности D. Это позволяет объединять блоки, или стакать (stack) их, используя выход одного в качестве входа для другого.



Устройство блока кодировщика (encoder)

Изображение такого блока представлено выше. Под капотом лежат две основные части трансформера. Верхняя называется FeedForward сеть. Не пугайтесь названия, концепция максимально проста — это всего лишь два линейных слоя с нелинейностью между ними, один из которых разжимает вектор, переводя его в пространство высокой размерности, а другой возвращает к необходимому значению, чтобы выход блока соответствовал размерности входа. Для полноты понимания укажем, что в классическом BERT вектор разжимается до эмбеддинга размерности примерно 3000, а затем сжимается до 768. Есть статья, в которой показывается, что эти 3000 ячеек представляют собой что-то вроде базы данных смыслов, и входной эмбеддинг "набирает" из разных строк совокупность таких смыслов, расширяя исходный.

Вторая часть блока кодировщика, которая является основной — это великий Self-Attention, основная механика, благодаря которой трансформеры так хороши, в частности в задаче матчинга.

> Self-Attention

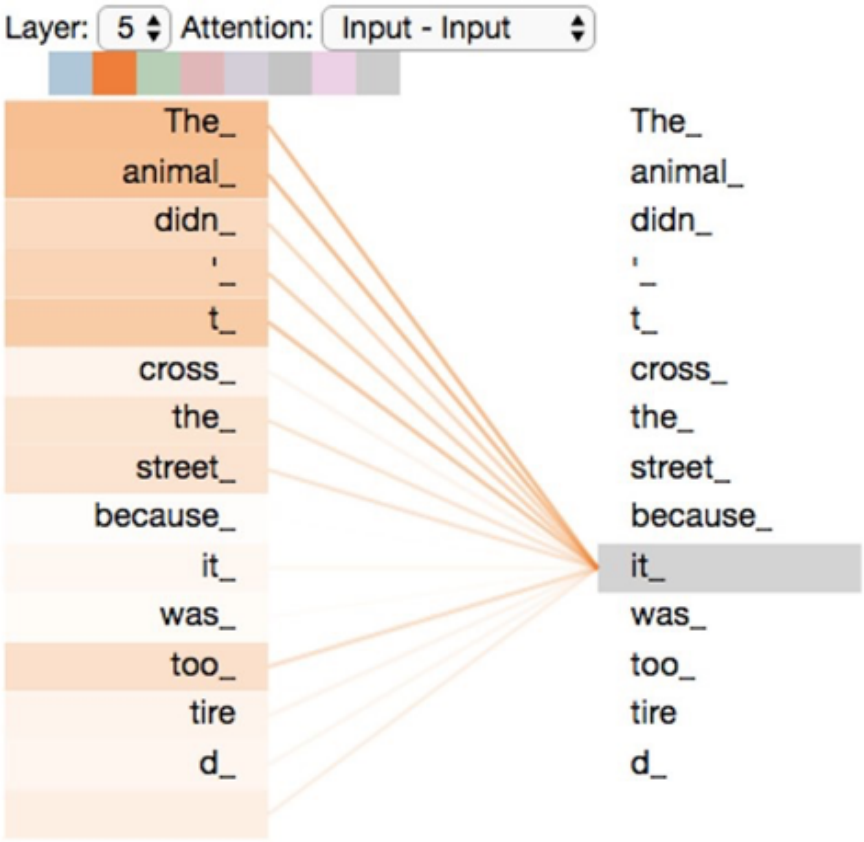
Для начала попытаемся на уровне интуиции понять, что это такое. Ни для кого не секрет, что слова в естественном языке связаны между собой в предложения не случайно. Простой пример: допустим, есть предложение:

"The animal didn’t cross the street because it was too tired".

В переводе на русский язык это означает следующее: "животное не переходило дорогу, потому что оно было очень уставшим". Во второй части предложения есть местоимение *"оно"* (*"it"*), и для человека абсолютно понятно, что идёт указание на животное, а не на дорогу. К сожалению, при переводе эта тонкость теряется, но если изменить "*дорога"* на "*дорожное полотно"*, чтобы род слов "*животное"* и "*полотно"* стал одинаковым, то получится то же самое:

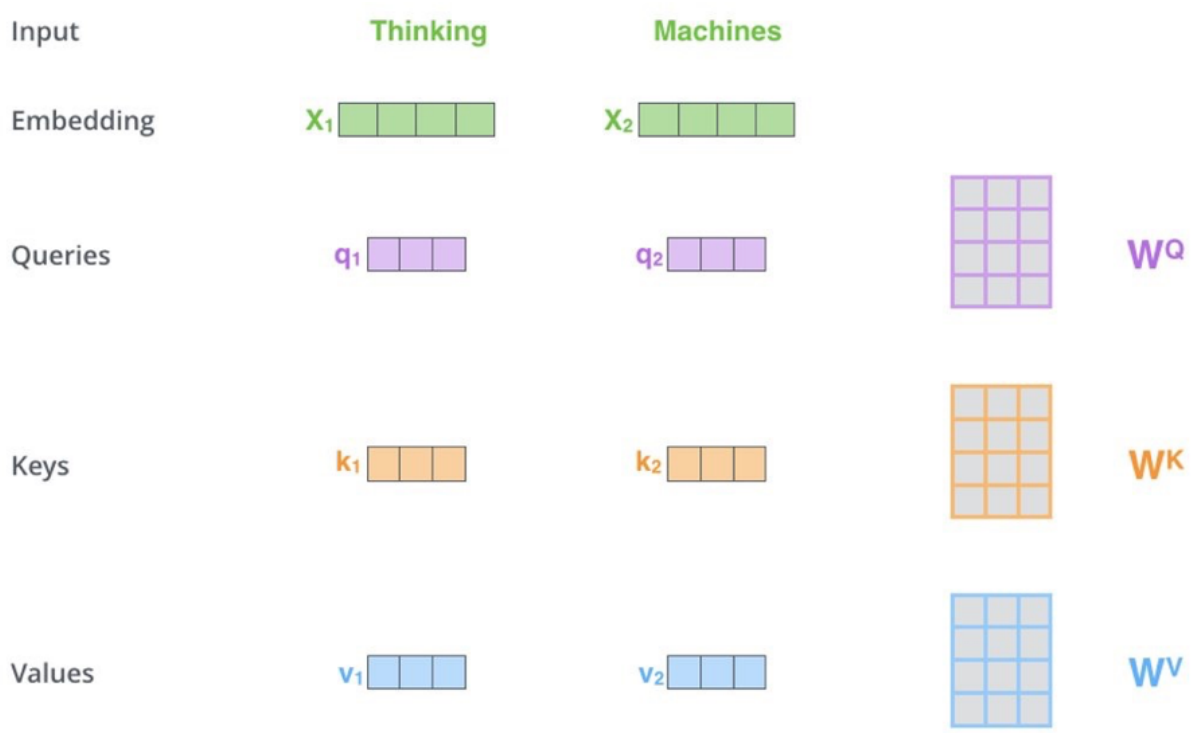
"Животное не переходило дорожное полотно, потому что оно было очень уставшим".

В этом примере "*оно"* — это животное или полотно? Наверняка вы вспомнили аналогичные задачи, которые давали на уроках русского языка в школе. Сейчас установление связи вам даётся легко. Установление именно такой связи и заложено в self-attention механизм, лежащий в основе трансформеров. Он задаёт веса связи от одного слова до других слов в предложении. Причём такие веса рассчитываются от каждого слова к каждому. Чем больше вес, тем больше связь и тем она более выраженная. Пример визуализации весов связей для слова “it” в разобранном предложении можно наблюдать ниже. Здесь видно, что наибольшую окраску, т.е. больший вес, получают верхние слова в предложении, указывающие как раз на животное.



Пример работы Self-Attention в рассмотренном предложении

На вход в Selft-Attention, как вы могли заметить на изображении блока энкодера, поступают N эмбеддингов, характеризующих N токенов. Для примера возьмём два токена — два эмбеддинга, выделенных зелёным цветом в верхней части изображения, представленного ниже. Сам механизм внимания (Attention) состоит из трёх разных обучаемых матриц. Обозначим их Query (W^Q), т.е. запросы, Keys (W^K), т.е. ключи, и Values (W^V), т.е. значения. Они изображены в правой части картинки и выделены разными цветами. Размерность этих матриц такова, что одна из сторон равна входным эмбеддингам.



Формирование эмбеддингов Queries, Keys и Values

Это означает, что входные эмбеддинги можно перемножить с этими матрицами, получив новые вектора. Для каждого эмбеддинга формируется 3 новых, возможно даже другого размера (это зависит от оставшегося размера матриц). Итого для 2 слов получаем 6 эмбеддингов, как представлено на изображении. Они обозначены q_1 и q_2 для эмбеддингов Query, k_1 и k_2 для ключей Keys, v_1 и v_2 для значений Values первого и второго токена соответственно. Если бы у нас было 10 слов, то тогда пришлось бы рассчитывать 30 векторов.

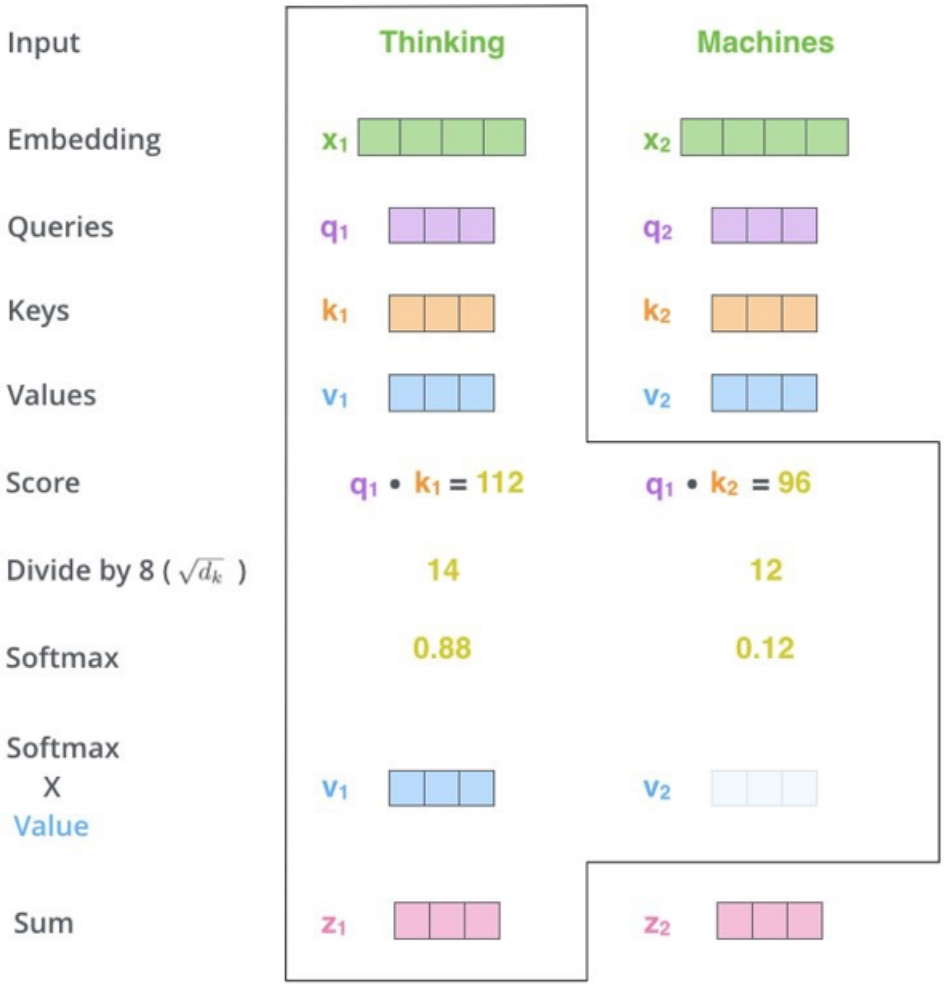


Иллюстрация работы Attention механизма

Далее давайте возьмём первое слово, его Query-эмбеддинг и рассчитаем скалярное произведение со всеми Keys-эмбеддингами. На выходе каждого произведения получается число, на изображении они выделены желтым (`112` и `96`). Будем считать, что эти числа пропорциональны тем самым весам связей в предложении (с каким весом каждое слово влияет на другое), что мы и хотим моделировать. Тут для каждого из N токенов получается N чисел, что порождает квадратную матрицу связей $N \times N$, в которой N^2 элементов. Для второго слова при сравнении со всеми остальными ключами будет использоваться его собственный Query-вектор, т.е. произведения будут $q_2 \cdot k_1$ и $q_2 \cdot k_2$ (меняется только первый множитель). Таким образом можно говорить, что Query формирует запрос от самого слова (токена) к контексту, т.е. то, что хочется видеть рядом с ним и что обычно встречается. А Key в данном случае формирует смысл слова, если оно стоит в контексте к чему-либо. Чем более схожи Key и Query, тем более сильной

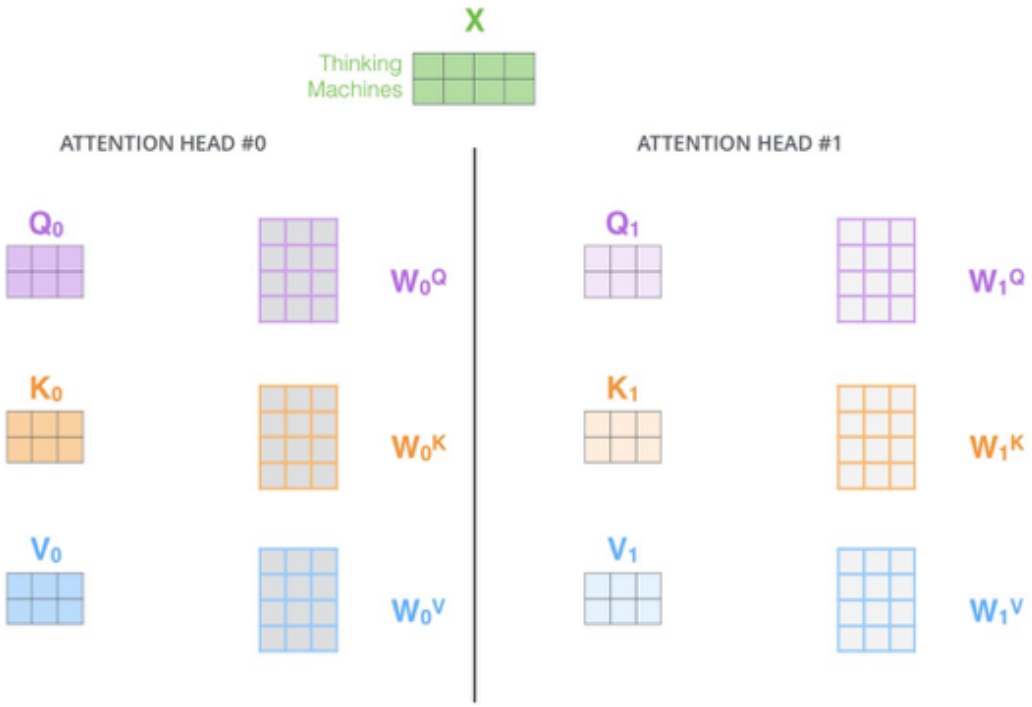
получается связь соответствующих слов, тем больше скалярное произведение и тем выше Score в 6-й строчке.

Далее эти скоры делятся на константу. Не будем вдаваться в детали, откуда она берётся — просто было замечено, что это улучшает сходимость модели (на самом деле она сохраняет размах распределения). К полученным после деления величинам применяется хорошо знакомый Softmax, нормируя величины в рамках каждой строки матрицы Attention таким образом, что их сумма даёт единицу. Это монотонное преобразование, поэтому чем больше значение Score, тем больший вес получается в результате операции. Например, для первого слова получились значения `0.88` и `0.12`, т.е. слово само на себя влияет очень сильно, но примерно на 12% зависит от второго. Далее в игру вступает Value-вектор, про который мы на время забыли и который никак не участвовал в вычислениях до этого момента.

Всё, что осталось сделать, это просуммировать Value-вектора с весами, заданными нормированными скалярными произведениями других векторов. Здесь в нижней части изображения видно, что второй Value-вектор бледнее, так как его вес существенно ниже, и потому в общей сумме он окажет незначительное влияние, но при этом не потеряется полностью. То есть только что полученный эмбеддинг z_1 для первого токена в предложении, обозначенный розовым цветом в самом низу, будет хранить в себе информацию о всех словах из предложения разом. Какие-то повлияют на него больше, какие-то меньше, но, так или иначе, если есть что-либо важное в предложении, то у хорошей модели такая связь получит весомую оценку. Как мы уже говорили, сложность вычисления и хранения такой матрицы квадратично зависит от размерности входной последовательности. Для 2 токенов необходимо всего 4 скалярных произведения, однако для 5 выходит уже 25. Типичный BERT обычно работает с последовательностями длины 512, так что не пугайтесь — "скормить" заголовки товаров представляется возможным. Более того ведётся активная исследовательская работа в области облегчения расчёта Self-Attention. Примеры таких архитектур — LongFormer и ReFormer. Они могут работать с 20000 токенов. Ими обрабатывают если не целые книги, то главы и отдельные статьи, например с Википедии. Более важно то, что если обрабатывать, скажем, два тайтла товаров в матчинге вместо одного и от 50 токенов перейти к 100, то ваши вычисления замедлятся в 4 раза.

Теперь вернёмся к архитектуре трансформера и применим стандартный для машинного обучения метод — если есть один алгоритм, то можно взять

несколько таких же, но с разными весами, и работать с ансамблем. Применительно к Self-Attention это означает, что можно взять не одну тройку матриц W^Q, W^K, W^V, а 2 и даже больше (в BERT таких триплетов 8). Пример показан ниже: для входного эмбеддинга двух токенов будут применяться сразу несколько матриц, и тем самым образуются разные пары ключей и запросов, а также значений. Поскольку все матрицы изначально инициализируются случайно, то в процессе обучения, если так можно выразиться, они сходятся к разным результатам, т.е. по-разному обрабатывают исходные вектора.



Multi-head Self-Atention

Обратимся к уже знакомой картинке (см. ниже). Теперь на ней в разных колонках слева разными цветами визуализированы разные “головы” трансформера (на английском их называют heads). Уже знакомая нам оранжевая указывает на действующее лицо, на животное применительно к местоимению *“it”*. А серая, к примеру, указывает на предыдущее слово *"because"*. Возможно, эта часть Self-Attention выполняет функцию объединения в словосочетания, или это аналог свёртки (мы не знаем, но можем полагать). Разные “головы” Attention называют экспертами, у каждого из которых своя задача в рамках вычислений. Все в сумме они позволяют комплексно оценить разные зависимости в тексте. Так как голов несколько, такой блок называется Multi-Head Self-Attention, или

многоголовый. Каждая часть реагирует на свой сигнал, выискивает свой паттерн. Более того, можно обратить внимание на верхний левый угол картинки: там указан пятый слой Layer 5. Как я уже говорил, блоки кодировщиков стакаются друг над другом. В целом было проведено множество исследований по типам связей, которые вылавливает Self-Attention, и можно полагать, что самые нижние блоки (скажем, первые два) опираются в большей степени на локальные признаки, на синтаксические, т.е. конкретное написание. Но чем выше номер блока, т.е. чем выше поднимается эмбеддинг по кодировщику, тем абстрактнее смысл — происходит переход к семантическим признакам, то есть к смыслообразующим.

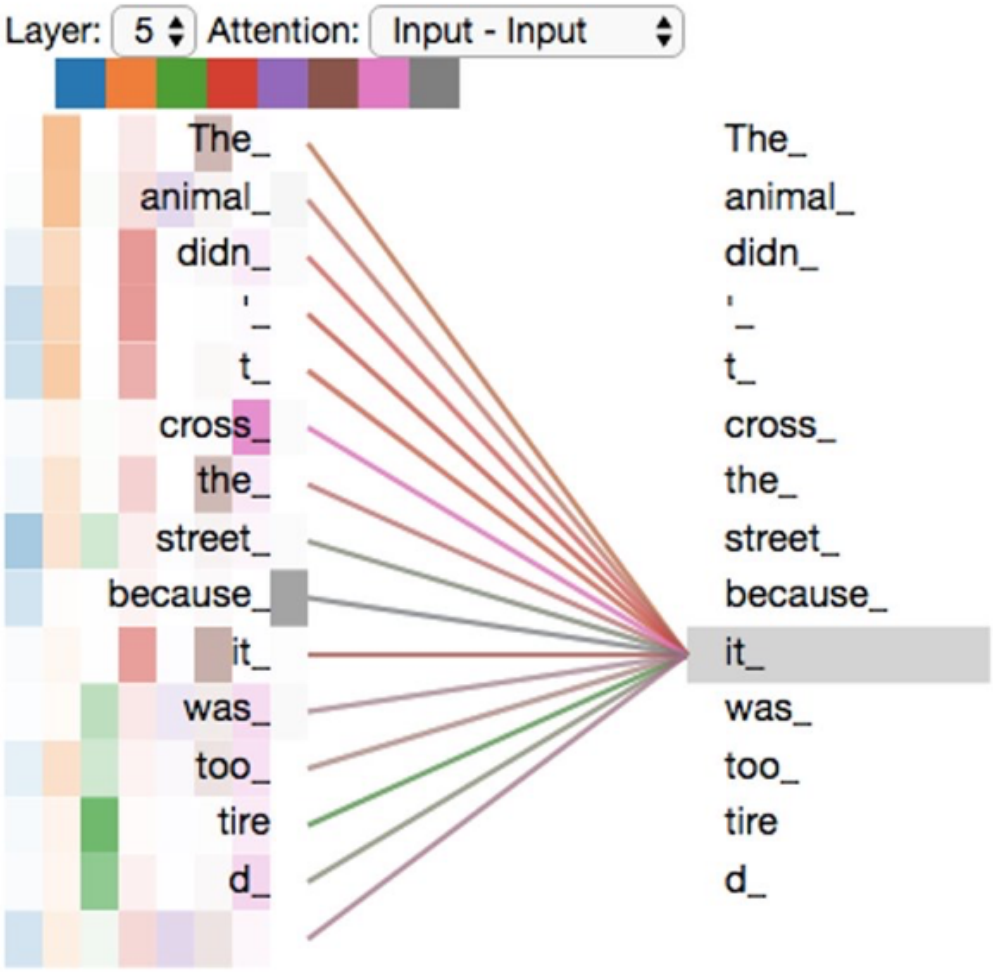


Иллюстрация работы различных "голов" Self-Attention

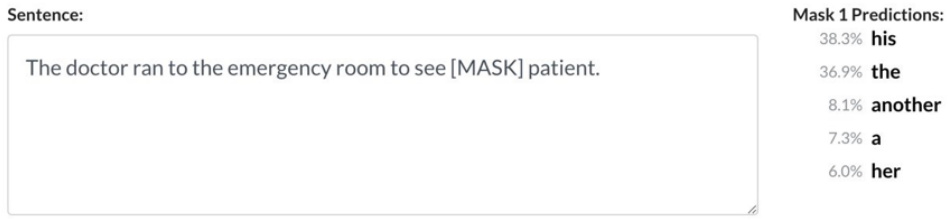
А теперь перейдём к не менее интересной части — процессу тренировки этого монстра.

> Обучение BERT: LML — Language Masked Modeling

Для начала рассмотрим методики Unsupervised предобучения BERT. Существует этап предтренировки модели на общих данных: текстах из интернета, книгах и т.д. Одна из таких задач обучения — Masked Language Modeling (Language Modeling — задача восстановления распределения последовательности слов $P(w_1, \ldots, w_n)$) .В этот момент происходит как бы “выучивание” языка и его основ (то же самое, что и в других моделях эмбеддинга). Тренировка происходит следующим образом *(цитата из оригинальной работы)*:

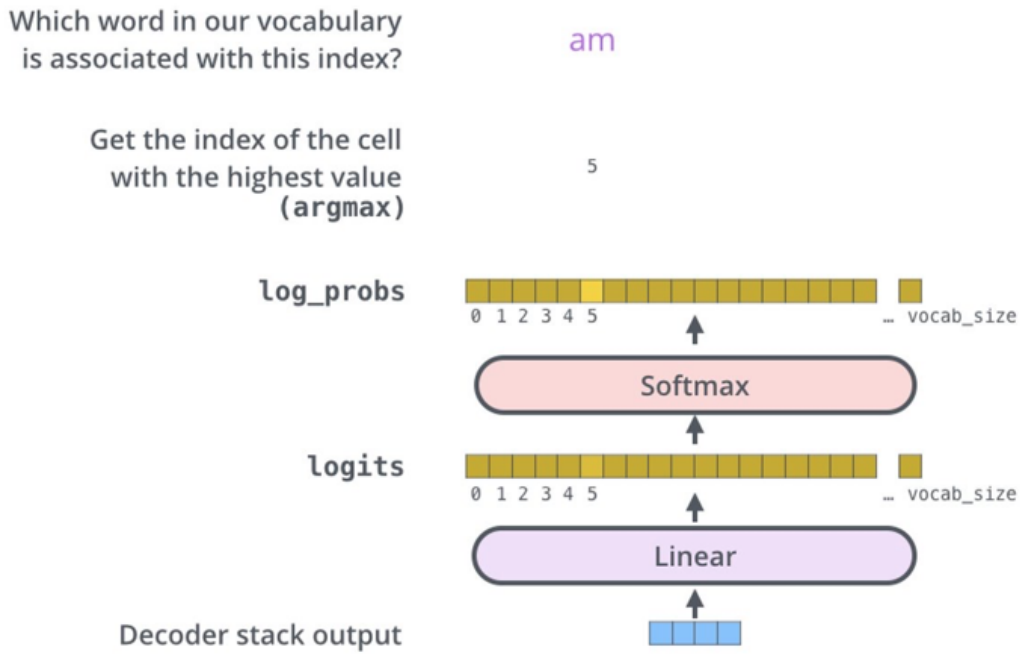
"Training the language model in BERT is done by predicting 15% of the tokens in the input, that were randomly picked. These tokens are pre-processed as follows — 80% are replaced with a `[MASK]` token, 10% with a random word, and 10% use the original word".

То есть случайным образом выбирается 15% токенов. Для 80% из них происходит замена на специальный `[MASK]`-токен, который подобно `[CLS]` и `[SEP]` был добавлен в изначальный словарь, в 10% случаях происходит замена на случайное слово из словаря, а в оставшихся случаях слово не изменяется. Последнее может казаться нелогичным, однако смысл здесь кроется в корректировке выходного распределения по токенам из словаря, чтобы оно было максимально близко к естественному распределению в языке, т.е. это своего рода регуляризация. На иллюстрации ниже представлен пример предложения, в котором есть `[MASK]` токен и справа представлены слова с наибольшей предсказанной вероятностью (по мнению обученной модели).



По сути эта задача очень похожа на разобранную в Word2Vec — по контексту (всему предложению) нужно угадать, какое слово пропущено, или замаскировано в данном случае. Помните, мы говорили про Bidirectional, или двунаправленность модели? Это обусловлено тем, что для

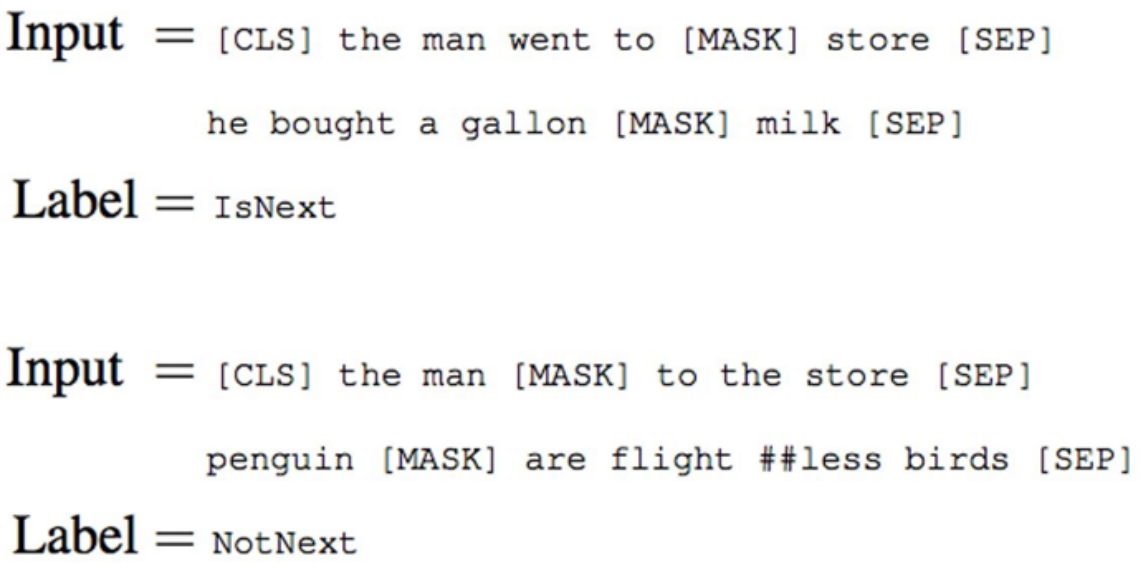
определения пропущенного слова используется полный контекст слова как слева, так и справа (важность такого подхода была рассмотрена в лекции про эмбеддинги). Так как Self-Attention считает связи от каждого слова к каждому, то на замаскированный токен влияют все остальные слова в предложении. Это гораздо лучше, чем некоторое узкое скользящее окно фиксированной ширины. Про понятие “распределение по словарю” мы также уже говорили: для замаскированного токена с помощью линейного слоя, выходная размерность которого равняется размеру словаря, т.е. измеряется десятками тысяч, предсказываются логиты, которые переводятся в вероятности. Затем берётся индекс максимальной величины и сопоставляется с соответствующим словом из словаря.



Такая задача определения замаскированных слов называется Masked Language Modelling (MLM). Но это не единственная задача, на которой учится BERT — тут возможно выучить лишь признаки контекста, но тогда получится, что объединение информации из нескольких предложений может стать проблемой.

> Обучение BERT: NSP — Next Sentence Prediction

Существует вторая задача предтренировки — Next Sentence Prediction, или предсказание следующего предложения. Здесь BERT выступает в качестве простого бинарного классификатора. На вход подаётся два предложения, разделённых сепаратором, и необходимо понять, идут эти два предложения в тексте подряд или же нет. Неоспоримым плюсом является наличие огромного количества данных для обучения модели и совершенно очевидно, откуда брать положительные примеры — в любом тексте есть как минимум пара предложений, которые идут подряд. А примеры для нулевого класса получить ещё проще: можно выбрать любое другое предложение, даже не обязательно из этого текста, любую другую книгу, статью или комментарий из интернета.



Пример данных для обучения

На изображении выше приведён пример разметки в такой задаче. Если мужчина идёт в магазин, а затем покупает молоко, то это очень похоже на естественное изложение истории, но вот если после первого факта говорится о пингвинах и птицах, то тут же что-то явно не так.

В начале лекции мы обсуждали кодирование входной информации для BERT, а конкретно segment tokens как одно из слагаемых в исходном эмбеддинге. С их помощью мы кодируем разные предложения, более явно указывая, какая часть текста относится к первому предложению, а какая — ко второму. Сначала идёт последовательность A, и для неё используется эмбеддинг первого предложения. Затем идёт второе предложение, и для него используется уже другой эмбеддинг B.

Это позволяет модели более явно различать составляющие входных значений, указывая на два разных отрывка текста. И это же подводит нас к переводу BERT в режим решения задачи матчинга или ранжировния.

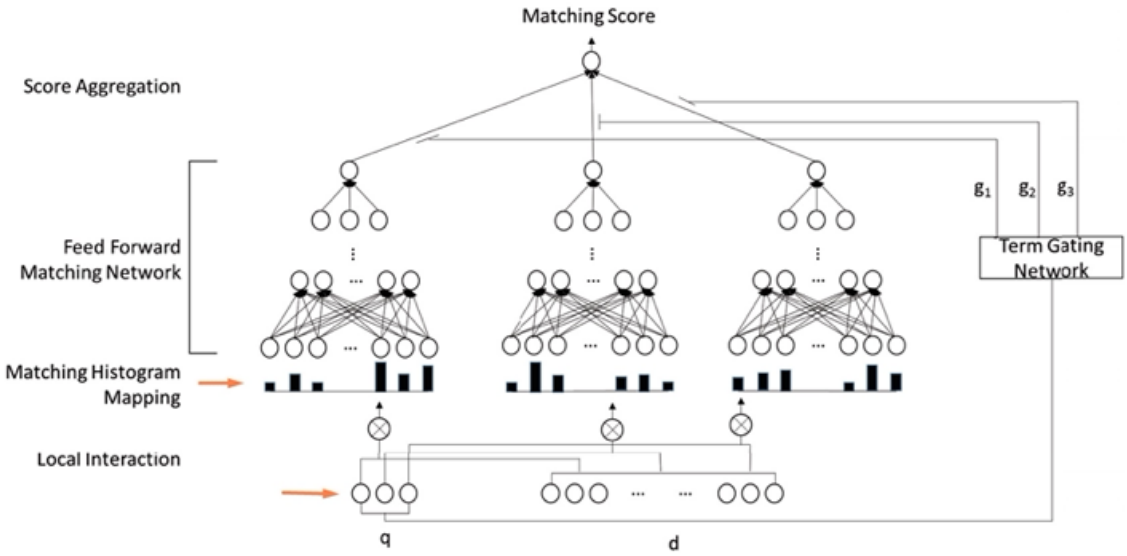
Запрос может выступать в роли первого предложения, и среди прочего кодироваться отдельным сегментом, а документ, или же заголовок товара и его параметры, как второй сегмент. Это даёт BERT понять, что с чем необходимо сравнить. Таким образом, мы имеем возможность за один раз закодировать и учесть информацию сразу от двух сущностей. И если за раз скормить и запрос, и документ-кандидат, то на каждом уровне кодировщика (encoder) будет происходить связывание и обобщение двух объектов, что существенно повышает качество. Говоря очень грубо, если в одном заголовке товара и в другом есть цвет, то BERT их сравнит и накинет условные баллы релевантности или схожести. Если сходится модель (товара), то ещё лучше, а уж если нечто похожее на артикул — так и вообще можно не сомневаться. И в режиме бинарного классификатора BERT очень успешно решает задачу матчинга товаров. Однако стоит понимать, что такой подход остаётся бинарным классификатором со всеми присущими ему проблемами. Скорее всего, бинаризация предсказаний в задаче матчинга будет осуществляться по константному порогу. Грубо говоря, если BERT (или модель следующего уровня, условный бустинг, в который подаются признаки картинок, метаинформация и прочее) предсказывает значение выше `0.93`, то это считается матчем, а если ниже, то пара не формируется. Но такой порог может сильно скакать от выборки к выборке и даже от одной категории товаров к другой. С этой, а также другими проблемами может помочь калибровка, про которую на одном из вебинаров рассказывал Валерий Бабушкин (посмотреть вебинар можно по ссылке). Возвращаясь к BERT, из опыта его использования можно сказать, что даже малого количества данных хватит для того, чтобы доучить (fine tune) модель, чтобы она показывала результат на голову выше, чем, к примеру, DSSM.

Вернёмся к Segment-токенам. В дополнение к двум заголовкам товаров, или в случае ранжирования — запросу и документу, можно добавить какую-либо метаинформацию, выделив её в отдельный сегмент. К примеру, можно учитывать геолокацию пользователя при поиске в интернете, подавать категорию модели товара, добавлять параметры, которых нет в заголовках, но которые могут быть важны. Эти части входных данных можно отдельно

шифровать дополнительными сегментами, помечая, что относится к запросу, а что — к документу.

> CEDR: Contexualized Embeddings for Document Ranking

Теперь, когда мы разобрались, что представляет собой BERT, как он работает и почему так хорош для матчинга, давайте перейдём к усложнению, а именно к его интеграции в уже изученные пайплайны. Напомним, что даже просто доученный BERT, напрямую предсказывая релевантность для задачи ранжирования или вероятность схожести товаров для матчинга, работает в целом неплохо. Но можно сделать и лучше!



Deep Relevance Matching Model (DRMM)

BERT для каждого токена выдаёт эмбеддинг, который отражает его смысл в заданном контексте. По сути это всё еще векторизующая модель, но гораздо более продвинутая. Очень легко представить, как такой трансформер внедрить в уже изученную модель DRMM (см. иллюстрацию выше, на которой дорисованы две стрелочки, указывающие места, в которых есть модификации).

Нижняя стрелочка указывает на первое важное изменение — теперь сопоставление схожести слов запроса и документа формируется уже из новых эмбеддингов, а не примитивных вроде Word2Vec. На данном этапе просто вычисляется распределение косинусных схожестей векторов, а затем

происходит деление на бины и создание гистограммы. Перед обработкой последующими слоями нейронной сети происходит добавление того самого `[CLS]` токена, который в BERT подаётся самым первым. В данном случае он несёт в себе весь контекст не каждого отдельного слова, а сразу пары запрос-документ. Также он позволяет дообучать сам трансформер, так как не участвует в нарезке на бины, а добавляется после. Следовательно модель получается дифференцируемой, градиенты текут, BERT обучается. Под “добавляется” имеется в виду простая конкатенация, или соединение векторов. Допустим, происходит распределение по 10 бакетам всех косинусных схожестей (cosine similarity). Это даёт 10 признаков, и к ним прикрепляется ещё 768 признаков от `[CLS]` токена (в расчёт добавляется контекст). Значит, теперь в верхнюю часть сети, Feed Forward Matching Network, подаётся 778 признаков. Данная операция производится, как и в классическом DRMM, с каждым словом запроса, после чего происходит взвешивание от Term Gating Network, которая определяет вес токена в сумме согласно его эмбеддингу, и на выходе получается суммарное значение релевантности. С точки зрения написания кода это очень простая и изящная интеграция.

Похожим образом происходит и добавление в KNRM, которая вместо гистограмм строит распределения, обработанные ядрами. К результату работы ядер также присоединяется эмбеддинг всего предложения, `[CLS]` токен. Всё это было предложено в рамках подхода CEDR. Однако если имеется много данных, даже офлайн процесс матчинга (без требований к работе в режиме настоящего времени с минимальной задержкой на ответ) работает очень долго, вы ограничены в ресурсах или вам всё-таки необходимо работать в реальном времени — BERT может стать проблемой.

> YATI: Yet Another Transformer (with Improvements)

Как уже было сказано, интеграция BERT очень сильно замедляет работу матчинг-архитектур, что делает их использование практически невозможным в реальном времени. Для решения этой проблемы инженерами Яндекса было предложено сделать следующий трюк (см. ниже).

Архитектура YATI

Можно распилить первые несколько слоёв кодировщика (encoder), тем самым снизив для них квадратичную сложность расчётов до суммы квадратов длин запроса и документа. В Яндексе решили распилить первые 9 слоёв. Они по отдельности обрабатывают поисковый запрос, как бы формируя высокоуровневое представление о нём, и проделывают то же самое для целого набора документов, для индекса всего интернета, формируя базу для поиска. Обратите внимание, что поскольку теперь эмбеддинг первых 9 слоёв документа никак не связан с запросом, то представляется возможным посчитать вектора для документов в офлайн-режиме (заранее), и не тратить время на вычисления, пока пользователь ждёт реакции от поисковика. При поступлении нового запроса его эмбеддинг формируется уже честным прогоном первых слоёв, однако это уже в несколько раз быстрее, чем расчёт в паре с документом. Далее некоторым образом формируется набор кандидатов из всех документов, и уже их объединённые векторные представления обрабатываются общей шапкой из трёх слоёв, в которой работает честный Self-Attention от каждого токена к каждому, что позволяет перенять все плюсы этого механизма без учёта низкоуровневых взаимодействий. В конечном итоге такая

модель всё еще предсказывает одно число — целевую релевантность, по которой можно упорядочить документы в поисковой выдаче.
В статье на Хабре указано, что внедрение BERT обогнало по качеству практически всю работу, проделанную за 10 лет в области поиска, генерации разных признаков, эвристик и т.д. Иначе говоря, можно выкинуть наработки, оставив только трансформер, и лишь немного потерять в качестве.

Но что делать в ситуации, если на обучение так или иначе ещё можно найти ресурсы, но применять саму модель точно не на чем, так как не хватит видеокарт и процессорных ядер? Паниковать не стоит, так как есть определённые способы, с помощью которых можно облегчить модель, сохранив улучшения.

Первый подход — это дистилляция. К примеру, вы обучили стандартный BERT на 12 слоёв, оперирующий эмбеддингами размера 768. Можно перенести знания из него в BERT из 4 или 6 слоёв, попутно сократив эмбеддинги в 4 или даже 6 раз и потеряв при этом несколько процентов качества. Такая модель работает гораздо шустрее. Примеры работ, на которые стоит обратить внимание, будут приложены к уроку. Ключевые слова: DistilBERT и TinyBERT.

Если у вас много серверов с процессорами, то можно оптимизировать работу BERT для них с помощью приёма квантизации — он может дать ускорение в $4-6$ раз, что существенно. О квантизации трансформеров написано много, даже в библиотеке PyTorch есть туториал и готовые инструменты.

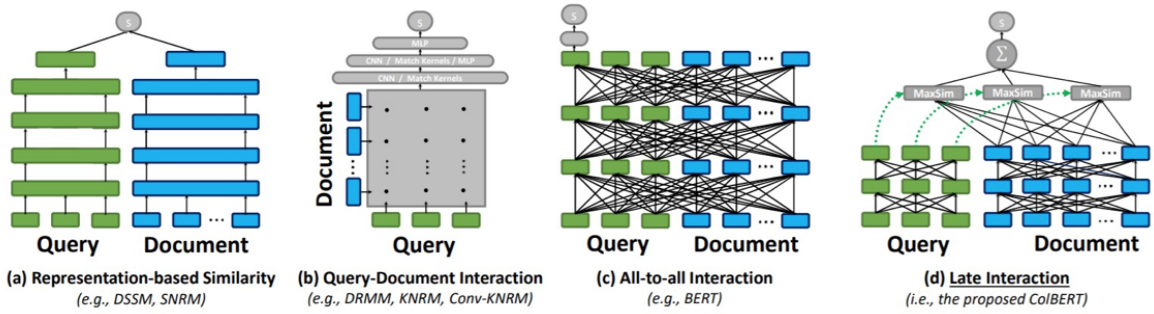
И последнее, что хочется упомянуть, это псевдо-лейблинг (pseudo-labeling), суть которого состоит в увеличении выборки, на которой обучаются модели попроще. Допустим, у вас есть 50 миллионов пар запрос-документ, но размечено лишь 300 тысяч, которые раньше использовались для тренировки DSSM. На этих парах вы можете обучить BERT, сравнив качество с DSSM, а после для оставшихся 49 миллионов в офлайн-режиме сделать предсказание релевантности, т.е. искусственно создать разметку (проставить псвевдо-лейблы). И уже на таком большом пулле данных обучить DSSM или другой легковесный алгоритм. На практике такой трюк используется достаточно часто: это хороший способ улучшить метрики уже существующих моделей, и главное — не придётся тратиться на их интеграцию, ведь нужно просто поменять веса на те, что получены с обучением на доразмеченных данных. Про этот подход написано в самом посте от команды Яндекса. Резюмируя, можно сказать, что из одного и того же набора данных можно с помощью более тяжёлой модели вытащить больше информации и сигнала,

которые затем передаются простой модели. Финальное качество выходит лучше, чем просто обучение на том же самом датасете. Таким образом, можно сказать, что происходит более качественное обучение того же самого алгоритма.

Но вернёмся к архитектурным трюкам. В подходе от Яндекса всё еще есть относительно тяжелая операция расчёта общей шапки с Self-Attention. Можно ли от неё избавиться?

> ColBERT

Оказывается, что можно. Решение предоставил Facebook. Модель называется ColBERT. В качестве замены общей части соединения векторов запроса и ответа предлагается применить в целом знакомую схему расчёта схожестей, что проиллюстрировано ниже (вариант d).



Сравнение различных подходов расч`та схожестей (similarity score)

Запрос и документ по отдельности векторизуются, производя на выходе некоторый набор эмбеддингов. Далее для определения релевантности для каждого слова запроса выбирается максимально схожий по скалярному произведению или косинусной мере токен документа. После этого все схожести суммируются, порождая оценку релевантности. Метод очень напоминает KNRM и DRMM с их Matching Matrix. Но что более важно — при таком подходе BERT начинает порождать вектора, которые можно просто сравнивать между собой. Поэтому процесс поиска кандидатов для ранжирования крайне прост: все документы предварительно векторизуются, а затем эти эмбеддинги кладутся в уже знакомый по лекции про поиск ближайших соседей индекс FAISS. Теперь для каждого слова запроса достаточно получить некоторое количество документов, содержащих максимально близкие по эмбеддингам слова, и уже на них запускать реранжирование. Опять же, это возможно потому, что при обучении

модели на итоговую релевантность влияют именно оценки схожести. Такая модель, как теперь понятно, может работать end-2-end, от начала и до конца пайплайна ранжирования или матчинга.

В самой статье очень подробно описан поиск соседей на больших корпусах документов и многое говорится про оптимизацию и распараллеливание расчётов. Если это близко к вашей прикладной задаче, тогда статья очень рекомендуется к прочтению: в ней рассказано и про бакетирование данных, и про разбиение на индексы, и про параметры для FAISS, и про замеры скорости и качества работы в зависимости от параметров поиска ближайших соседей. И работает это лучше простого поиска по векторам Word2Vec или DSSM, потому что каждый токен здесь контекстно обогащён: он содержит информацию не только о самом слове, но и обо всём, что есть вокруг него. То есть разные слова в одинаковом значении будут порождать очень похожие вектора, что существенно улучшает качество информационного поиска. Также на изображении выше представлено отличие такой модели от уже изученных нами.

Слева на картинке представлены Representation-focused модели, которые целые предложения сжимают в единый вектор, и релевантность оценивается близостью их эмбеддингов. Такие модели быстры на инференсе, т.е. при применении, но потенциально теряют много информации.

Далее идут Interaction-focused модели, которые оценивают все связи между словами и, как следствие, работают дольше. Сам BERT в некотором роде является представителем такой модели, он изображен третьим слева на рисунке. Эта модель, как мы уже обсудили, формирует связи слов каждый-к-каждому, и здесь учитываются даже взаимодействия слов в рамках самого запроса и документа по отдельности, чего не происходило в ранее разобранных моделях.

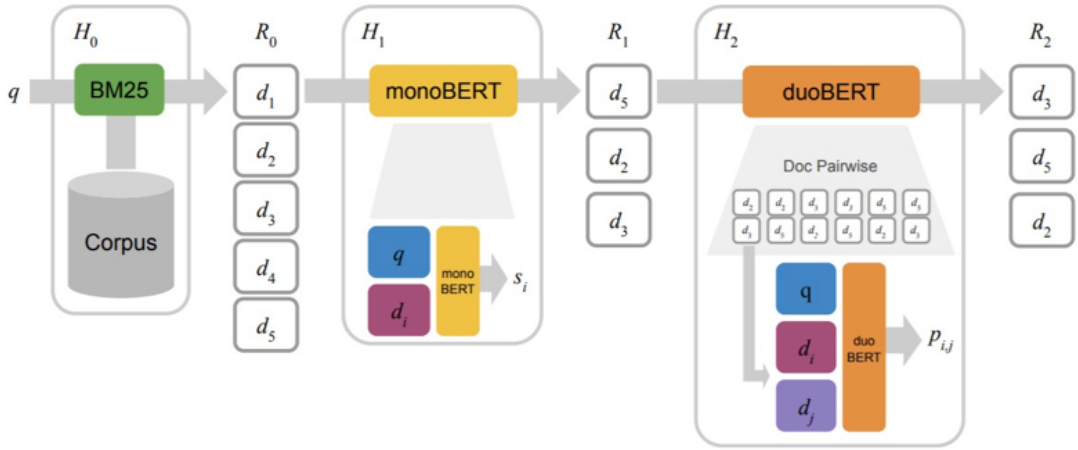
Итак, давайте двигаться дальше. Что делать, если у вас, наоборот, много ресурсов и хочется чего-то более тяжёлого и продвинутого?

> Multi-stage Document Ranking with BERT

Если вам хочется чего-то тяжеловесного, тогда можно использовать сразу несколько BERT моделей на разных этапах!

Ранее была рассмотрена возможность обучать модель на парах запрос-документ, а информацию о ранжировании разных документов для одного запроса вносить с помощью, например, Pairwise функции потерь, которая оценивает, какой из двух объектов более релевантен запросу. Можно эту функцию делегировать самому BERT, обучая его уже на триплетах на задачу бинарной классификации, отвечающей на вопрос: "Верно ли, что i-й документ более релевантен запросу q, чем j-й?"

Очевидная проблема такого подхода на инференсе заключается в том, что чем больше документов отобрано для последнего этапа реранжирования, тем больше получится пар, и зависимость будет снова квадратичная. Для 10 документов нужно обработать и затем корректно упорядочить на основе попарной информации всего 90 пар, но уже для 20 документов это число вырастет до 380, т.е. более чем в 4 раза! Однако это очень интересный подход, который нельзя не упомянуть. Возможно, он кому-нибудь пригодится. В теории это применимо, когда ожидается не так много кандидатов на выдачу — скажем, до 4 или, может, до 10.



Схематичная иллюстрация подхода

В конце необходимо показать следующую картинку, демонстрирующую пример работы системы в социальной сети LinkedIn. На ней видно, как по разному обрабатываются разнородные входные данные.

По сути, BERT и любая другая модель выступает здесь в качестве экстрактора фичей, которые затем объединяются. Такую модель можно назвать мультимодальной, ведь она может работать с любой информацией, комбинируя и текст, и картинки, и даже звук. Ничего заумного в объединении признаков нет. В целом в ранжировании работает стандартная парадигма

глубокого обучения — скиньте всё в модель, а она разберётся при достаточном количестве данных. Даже простая конкатенация работает "на ура".

Авторы опубликовали свой фреймворк и назвали его DeText.

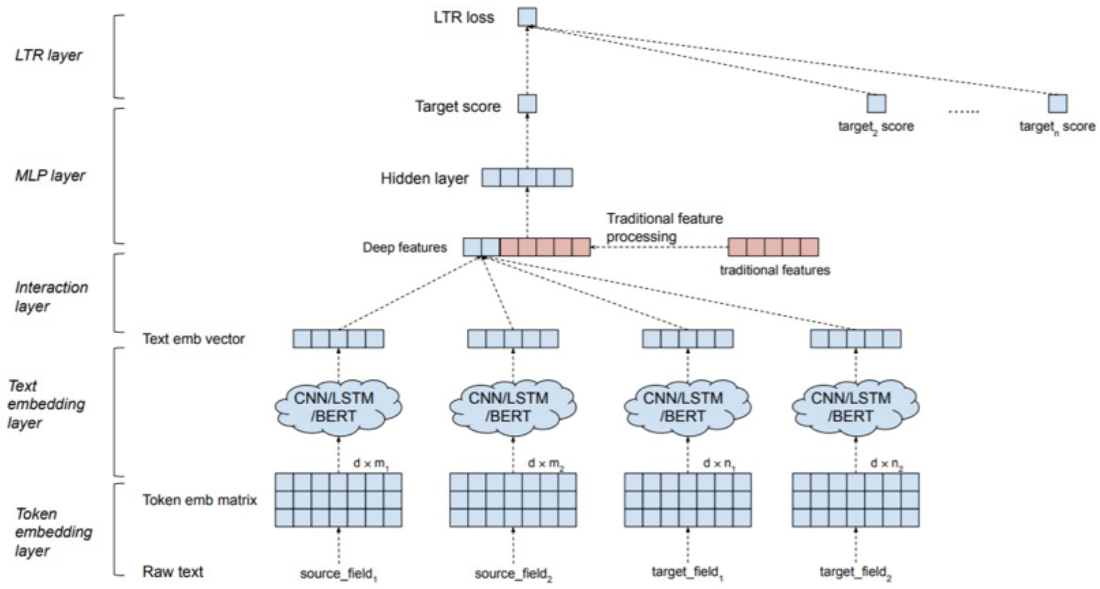


Схема мультимодальной модели, используемой в LinkedIn

> Резюме

Два ключевых момента, которые стоит помнить про BERT, поскольку они существенно влияют как минимум на задачу матчинга:

- Умная токенизация, например BPE — Byte Pair Encoding;
- Механизм Self-Attention, заставляющий модель обращать внимание на каждое отдельное слово в тексте.

Такая модель изначально тренируется в режиме предсказания замаскированных слов, а также на задачу предсказания связи двух предложений. Этот подход, с точки зрения бинарной классификации, очень близок к задаче матчинга. Даже применяя модель в лоб, можно добиться хороших результатов.

Но можно не ограничиваться таким наивным решением и переиспользовать эмбеддинги в других архитектурах, оперирующих векторами. Более того, существуют подходы для ускорения применения модели абсолютно разными способами: от распиливания слоёв модели на две ветви, для документов и запросов (YATI), до избавления от общей части как

таковой и даже до разметки текста с помощью уже обученной модели (ColBERT).

В заключение необходимо отметить, что BERT может казаться чем-то громоздким и ненужным и что в ваших условиях его обучить нереально, даже для простого псевдолейблинга. Однако, к примеру, сервис AWS продаёт сервер с 4 очень мощными видеокартами Tesla V100 за 12-15 долларов в час, а существующие оптимизированные методы обучения, развившиеся за последние 3 года, позволяют обучить BERT с нуля за 4 суток. В результате стоимость очень долгого единоразового процесса предтренировки составляет менее двух тысяч долларов, что в рамках компании, скорее всего, несущественно. А дообучить модель на малом датасете всегда можно локально, используя две или даже одну видеокарту. Более того вам не нужно учить модель с нуля, если вы работаете не со специфичными текстами вроде названий товаров. Если это новостные топики или, например, вопросы-ответы общего характера, то и вам подойдёт и базовая предобученная модель. Несколько лабораторий и компаний из России выложили RuBERT, который скорее всего можно рассматривать в качестве бейзлайна — его нужно всего лишь немного дообучить.