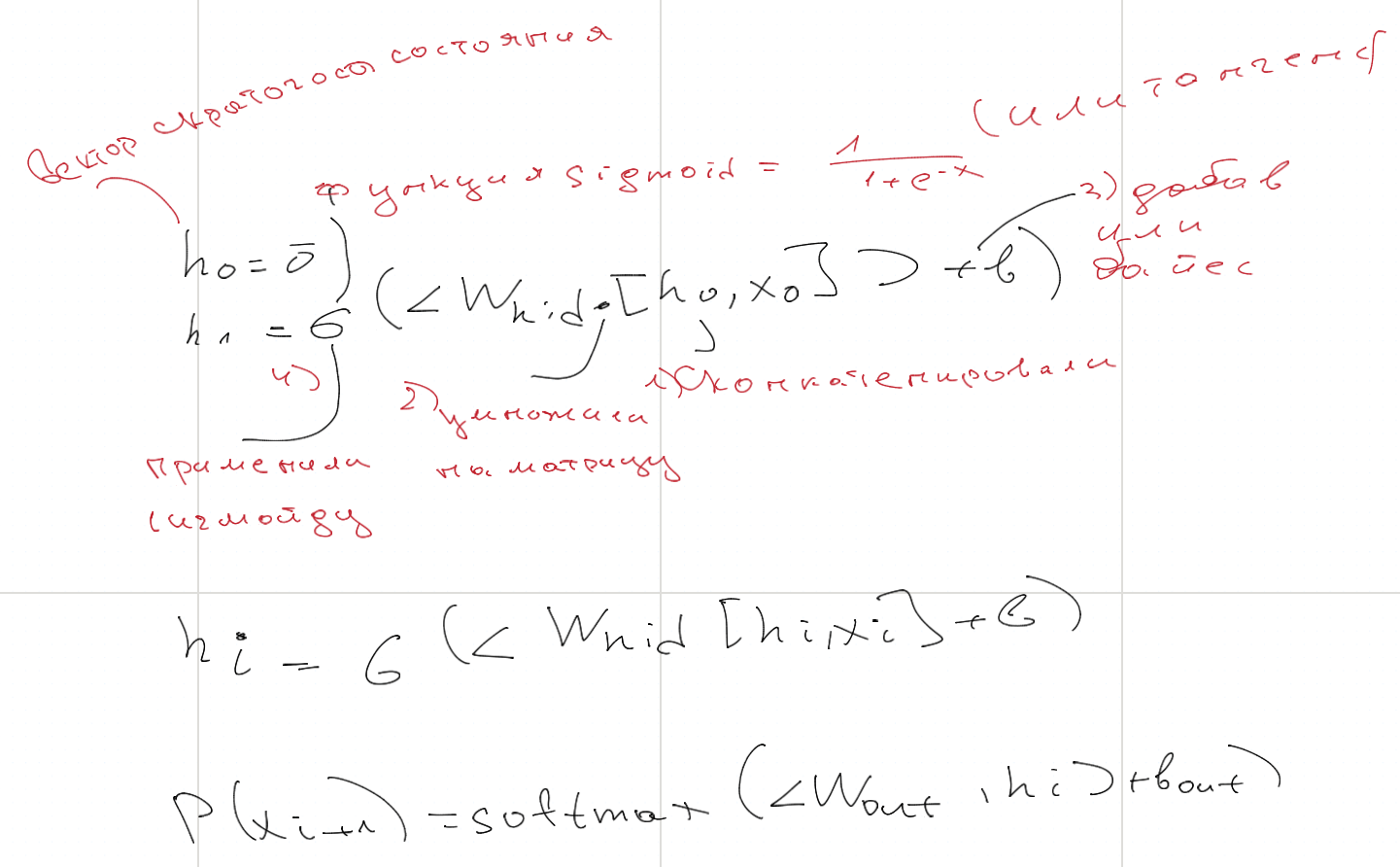
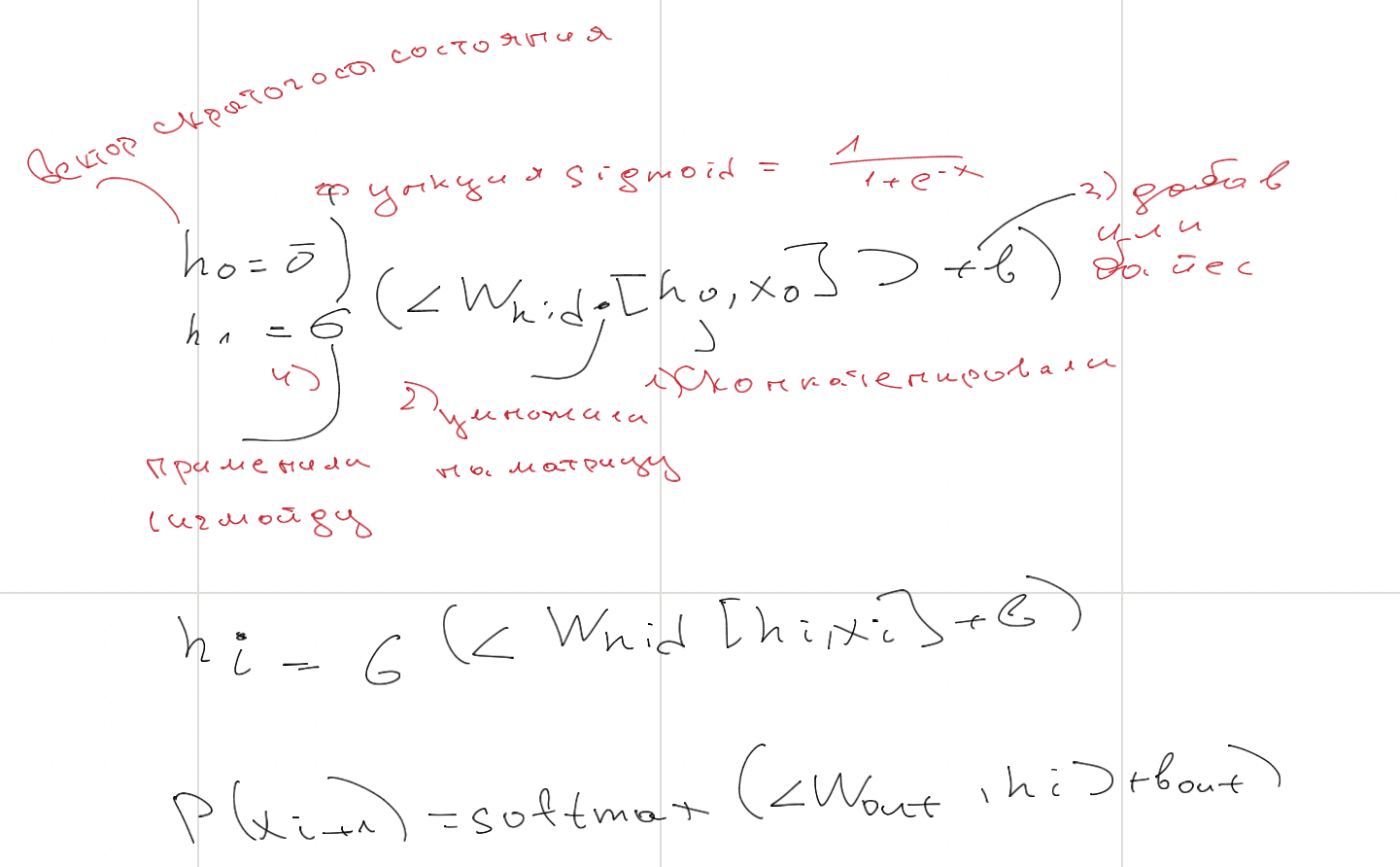
**Введение.**

Cегодня мы хотим рассказать о трансформерах. По теме доклада мы решили разобрать транфсормеры: BERT и GPT, а также их модификации. В течении долгой работы выбора библиотеки и выбора инструментов, было выбрано реализовать работу BERT, используя библиотеку на python pytorch. Главная идея программы - вывести фильмы, которые больше всего подходят под описание.

**Какие нейросети нам нужны?**

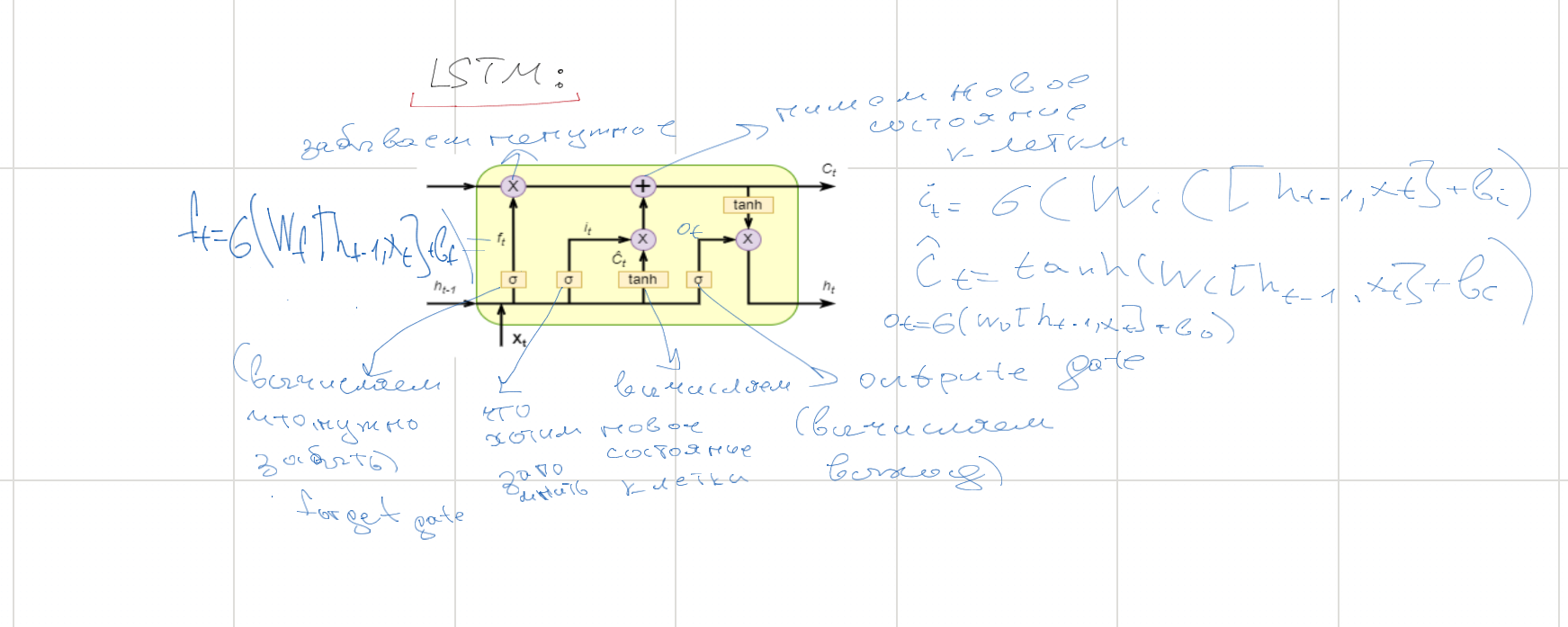
Для того, чтобы понять архитектуру трансформера разберем, какие нейросети применяются в архитектуре, а потом в одной главе соеденим все предыдущие и посмотрим как работает сам трансформер. Самые популярные нейросети LSTM/GRU ( они являются реккуретными), бывают и другие виды, но мы обсудим только эти, так как они показали себя очень хорошо в архитектуре трансформера. Но с начала обсудим главную идею реккурентных нейронных сетей.

РНН(Реккурентная нейронная сеть) - нейросети, которые нужны для работы с данными, которые представляют из себя последовательность некоторых токенов. В данной работе токены — слова. Такие нейросети помогают во многих задачах NLP(определение частей речи, определение спама и тд). Идея нейронной сети состоит в том, что в нейронную сеть прямого распространенния мы добавляем вектор(простейший случай) скрытого состояния, в котором хранится информация за предыдушии шаги. Как считается следующее скрытое состояние показано на рисунке, где xi — входная последовательность. Все обучение состоит в том, чтобы обучить матрицу весов.



Однако в этих нейросетях есть и минусы. Один из самых значительных — это затухания градиента, из-за неё мы плохо запоминаем начальную информацию. Для этого были придуманы разные нейросети, здесь мы разберем LSTM.

Немного про LSTM(Long-short term memory). На схеме ниже преведена схема с описанием. По ней видно, что к классической РНН, где была только функция тангенса добавляется много чего. Самая главная идея — добавить вотора запоминания и забывания. На схеме все подписано, как и все формулы [] - обозначается конкатенация двух векторов.



**Главная идея и функция внимания.**

Начнем с главной идеи механизма внимания в машинном обучении. Например у нас есть текст: 'Надо сдавать лабораторные до срока выполнения". Возьмем второе слово "сдавать", чтобы показать близость между словами учитывая контекст, мы можем токенизировать слова данного предложения, и каждому слову присвоить вектор, например в данном случае используем можно использовать тип кодировки one-hot, далее мы считаем скалярное произведение с другими словами(включая себя же), получается 6-мерный вектор, состоящий из весов, вес - обозначает значимость слов в контексте, для удобства нормализуем эти вектора с использованием функции softmax.

**Позиционные эмбединги.**

Перед тем как ввести формулу само-внимания обсудим что такое позиционные эмбдинги и какие типы бывают. Не будем забывать что запросы это матрицы слов в предложении.

Позиционный эмбединг - вектора, который хранит положение слова в предложении, он необходим, так как без него будет проблема о незнании положения слова в предложении. Почему они нужны? Так как без них мы не будем понимать на каком положении должно стоять каждое слово и без них у нас не будет четкого хорошего ответа.

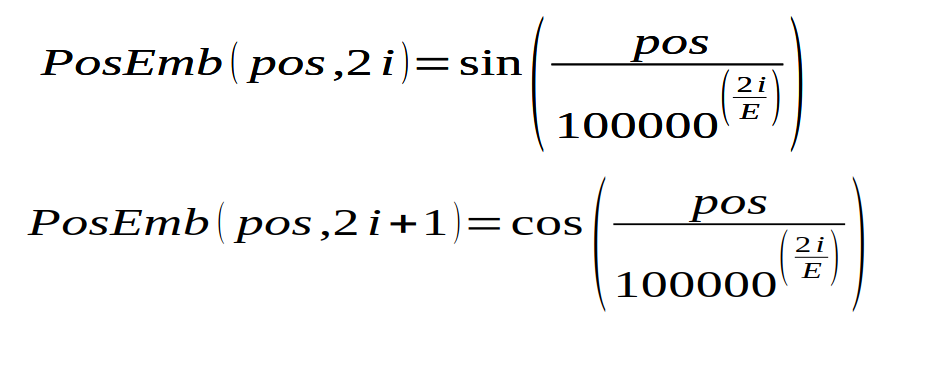
ПЕ разделяются в МХСА на два типа:

1)Абсолютный

2)Относительный

В абсолютном каждый входящий токен на позиции i сопоставляется с тренировочным ебмединг вектором, который показывает строку в матрице R с размером [tokens, dims]. R - тренировочная матрица, инициализированная значениями (0,1).

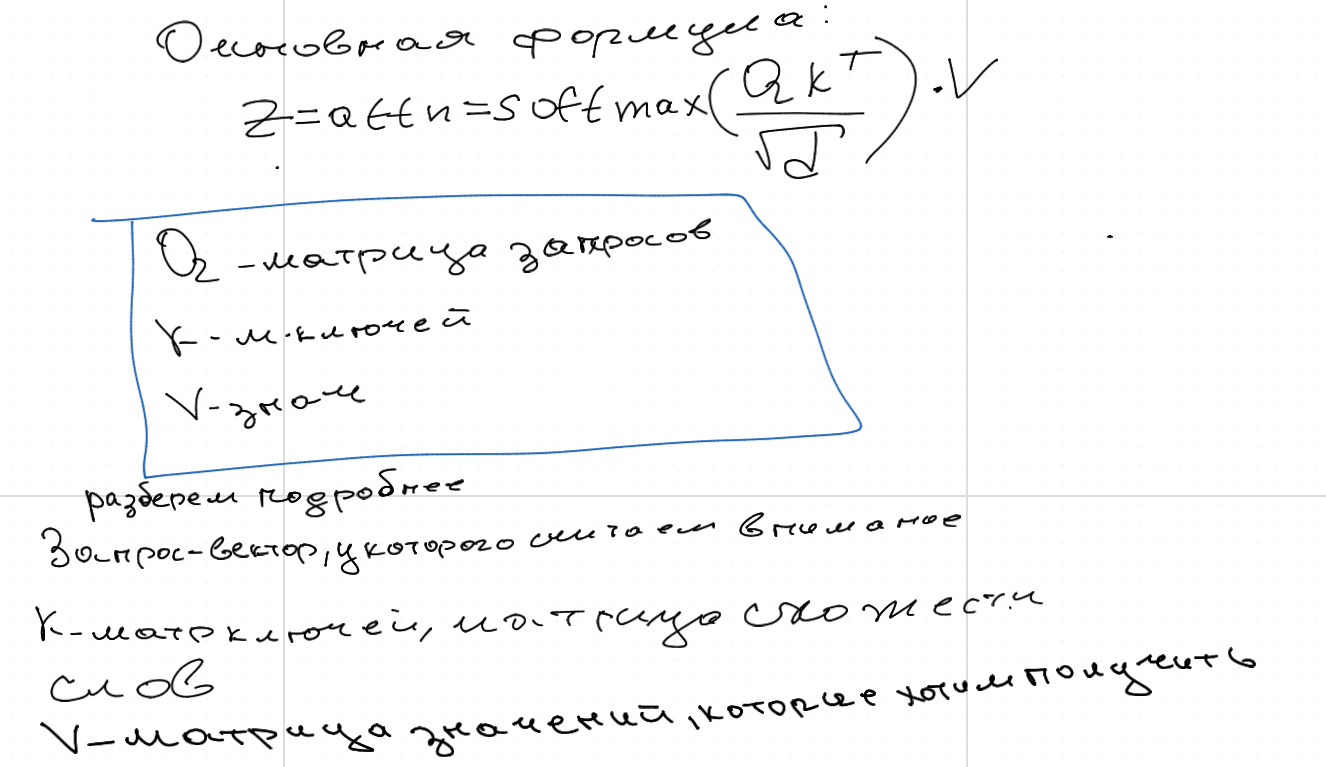
Позиционныеабсолтные эмбединги считаются по даным формулам, где первая для четной компоненты, вторая нечетной. Здесь pos — позиция слова в предложении, i — номер компоненты вектора.



Почему была выбрана функция синуса для четных и косинуса для нечетных? Так как авторы теортически надеялись, что они позволять модели легко обучитья обращать внимания к относительный позициям.

**Функция внимания.**

Теперь, когда был введен термин позициного введения, посмотрим на формулу внимания.



В чем смысл данной формулы? Она помогает вычислять внимание слова к остальным словам. Т.е. у нас есть матрица запросов — матрица векторов у которых считаем внимание, матрица ключей — матрица которая содержит в себе значения, обозначающие отношение данного слова к другим словам, и матрица значений. Цель механизма внимания — найти значения V, которые лучше всего подходят под запрос Q конкретного слова. С самого начала они имеют случайные значения и в процессе обучения заполняются, т. е мы запускаем процесс обучения в обычной сети прямого распространения, например, с методом обратного распространения ошибки, и получаем матрицы Q,K,V, которые далее применяются в функции softmax. Также нельзя забывать, что она решает проблему многозначности слов.

Еще интересный факт, почему именно корень из размерности d? Как приводится в оригиналньной статье: принято делить на корень из размерности E векторов матриц Q и K. Мотивация для этого может быть следующей. Пусть компоненты двух векторов q и k являются независимыми случайными величинами с нулевым средним и единичной дисперсией. Тогда скалярное произведение q⋅k также имеет нулевое среднее и дисперсию равную размерности векторов E, т.е. типичные значения q⋅k находятся в интервале ±√d. Масштабирование переводит их к интервалу ±1.

Также перед рассмотрением многоголовго внимания, надо не забыть сказать про маскированное внимание. Маски в этом типе внимания разделяются на два типа: булева(key\_padding\_mask) и вещественная(attn\_mask). Булева маска нужна для того, чтобы исключать некоторые пары ключей и значений. Для этого создается вектор из значений True, False, где значение True те значения исключаем. Вещественная же маска используется для точечного исключения кГлавная идея и функция внимания.онкретный весов, т. е. Создается матрица [N,M], в которой ячейки обозначаются 0 или минус бесконечности. Минус бесконечность ставится в том случае, если на данноя ячейке нужно исключить вес при помощи сложения.

**Многоголовое внимание.**

Где

Фокусы внимания - головы. h - количество голов внимания. В оригинальной статье было 8 параллельных слоев внимания(или голов). Для каждого из них мы использовали .

По факту входящие матрицы Q,K,V сначала подвергаются линейному преобразованию, а затем разрезаются по вертикали на H голов и веса внимания вычисляются уже независимым образом для каждой головы.

С самого начала для векторов запроса, ключей и значений делают линейное преобразование при помощи трех этих матрицы. При это для каждой головы набор матриц свой. После вычисления функции внимания для каждой головы получается 8 голов внимания их просто соеденяют в одну по последнуму индексу. Её свертка с приводит к финальной матрице.

**Декодеры и енкодеры.**

Далее рассмотрим простейшие структуры енкодера и декодера: Енкодер - это рекуррентная сеть(например LSTM), которая принимает на вход текст, и после работы в финальной ячейке хранит вектор скрытого состояния, который накопил в себе обо всем source-предложении (context vector). Этот вектор поступает как первый вектор скрытого состояние в декодер.

Декодер - тоже реккуретная сеть, у который первый вектор скрытого состояние - вектор из енкодера. Затем на вход первой ячейки декодера подаётся служебный токен (begin of sentence). На выходе этой ячейки сеть обучают выдавать слово-перевод . Для этого выходы ячеек пропускают через линейный слой (fc) с числом нейронов равным числу слов в словаре. Затем softmax-функция (sm), выдаёт "вероятности" слов из которых выбирается номер максимальной (argmax). Вектор полученного слова "кот" передаётся на вход второй ячейки и т.д. пока не получится служебный токен <ЕOS> (end of sentence).

- вектор скрытого состояние на ячейке альфа в енкодере, - в декодере, следующий вектор скрытого состояние считается как его сумма со взвешенным скрытым состоянием энкодера. В след ячейку декодера отправляется не текущее скрытое состояние , а его сумма со взвешенным скрытым состоянием энекодера, где , а (скалярное произведение). Возьмем одно важное уточнение что сумма значенмя омега равно единице.

Теперь применим софтмакс к данному подсчету. (μ=1/√E)

Значит веса внимания к скрытым состояниям декодера определяются следующим образом: **,**

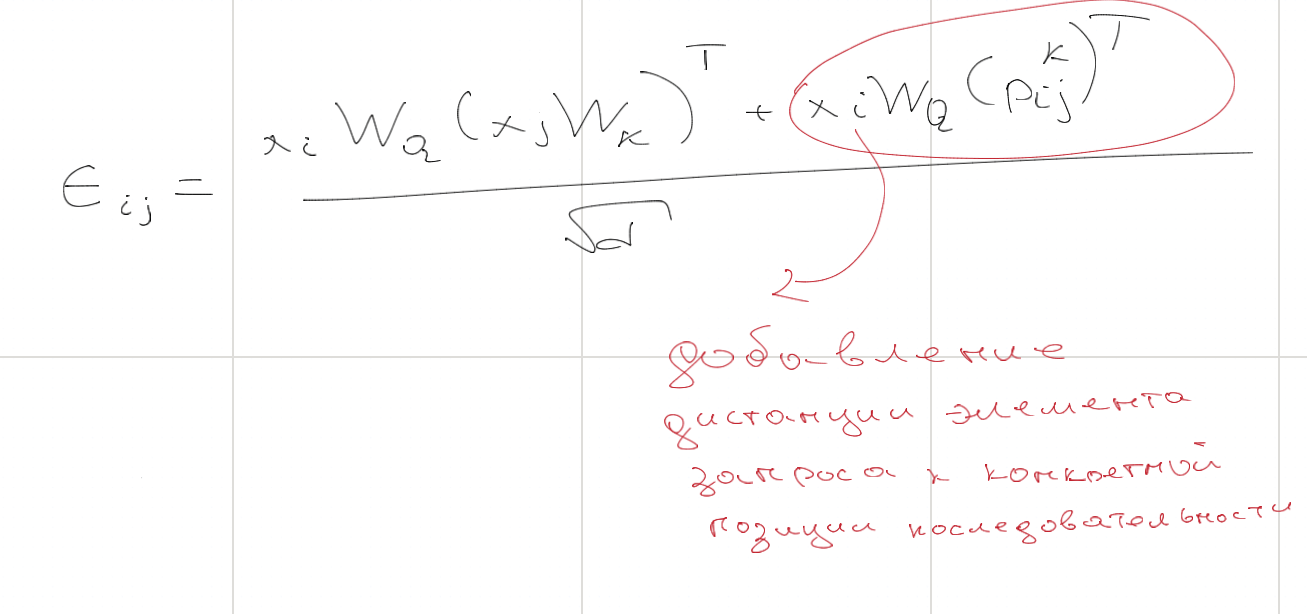
**Трансфомеры.**

Теперь соеденим все наши разобранные темы и обсудим архитектуру трансформера.

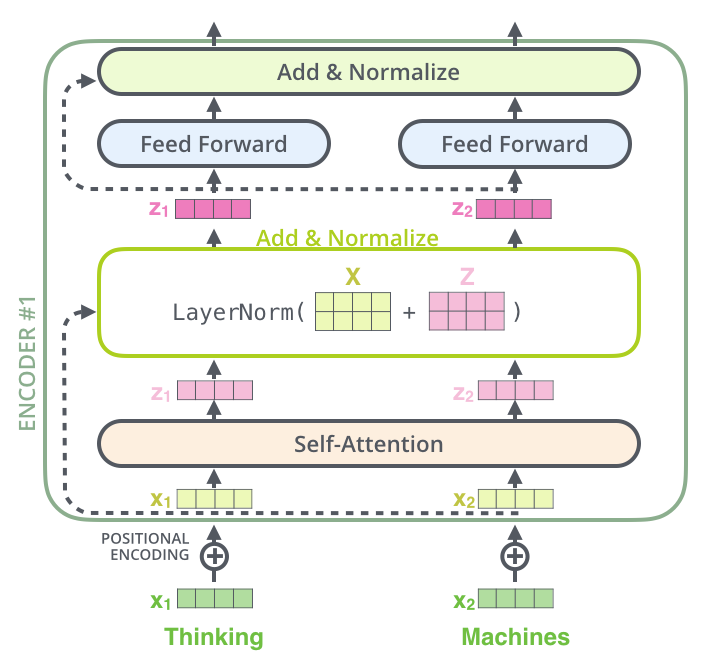
Архитектура трансфомера - это архитектура созданная из енкодера и декодера, только их какое то Н количество штук.

Начнем с начала, каждая такая строка в матрицу была сгенерирована из соответствующего исходного слова посредством серии трансформаций — векторного представления, позиционного кодирования и линейного слоя. Все эти трансформации возможно обучить; это означает, что используемые в этих операциях веса не определены заранее, а изучаются моделью таким образом, чтобы они давали желаемые выходные прогнозы.

Рассмотрим классическую схему енкодера и рассмотрим её подробней с учетом наших знаний. Перед этим возьмем формулу добавляения позиционного эмбединнга к входной последовательности:



На картинке мы можем увидеть классическую схему енкодера трансформера.

****

В енкодер поступают последовательные слова x1,x2, к ним прибавляется позиционный эмбединг, далее этот новый вектор проходит через слой внутренего внимания далее получается новые вектора, которые нормализуются, дальше они проходят через полносвязную сеть прямого распространения, в которой функция активации Relu.

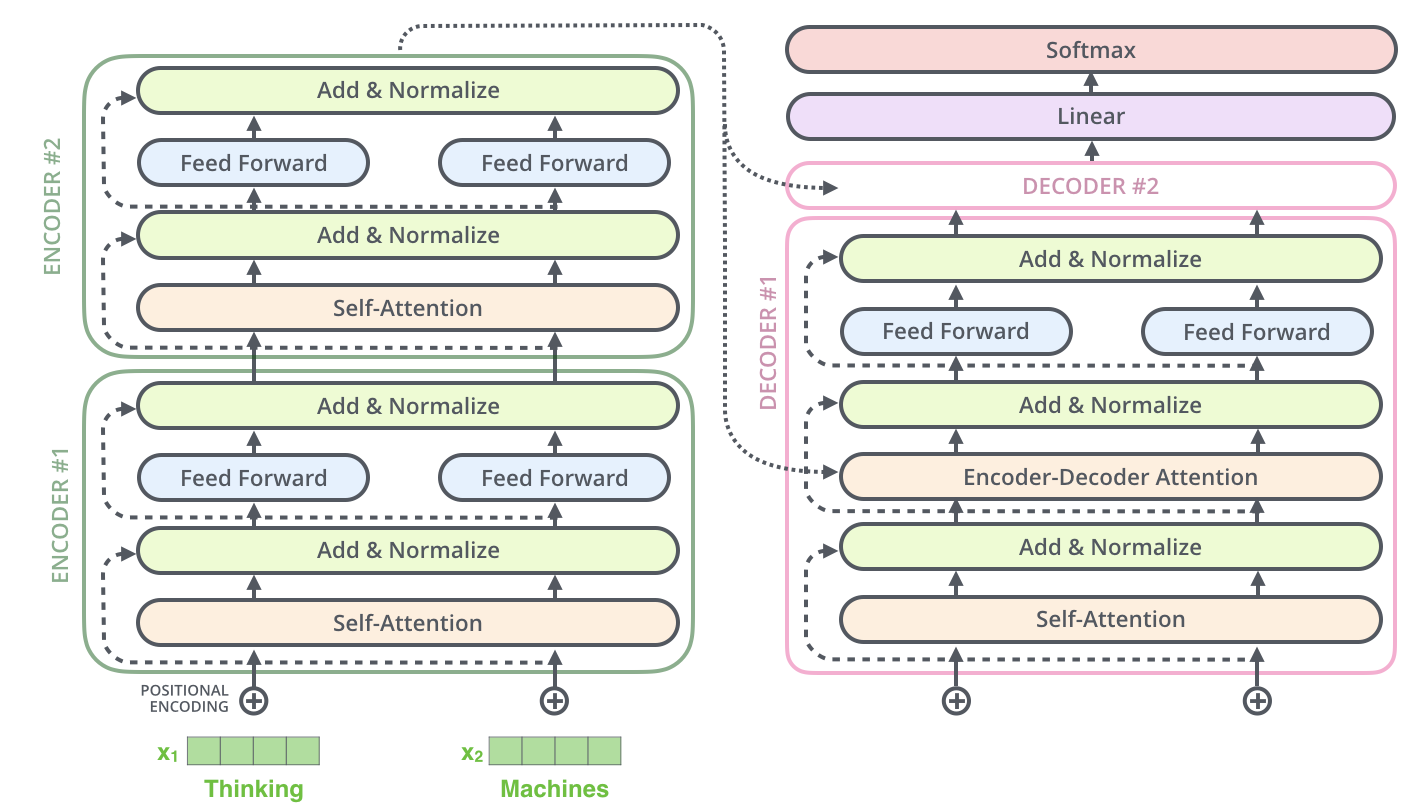
Энкодер начинает обрабатывать входящее предложение. Выход верхнего энкодера затем преобразуется в набор векторов внимания K и V. Они используются всеми декодерами в их «энкодер-декодер» слое внимания, что помогает им фокусироваться на подходящих местах во входящем предложении.

Теперь посмотрим как работает механизм внимания в декодере.

Польная схема архитектуры трансформера.

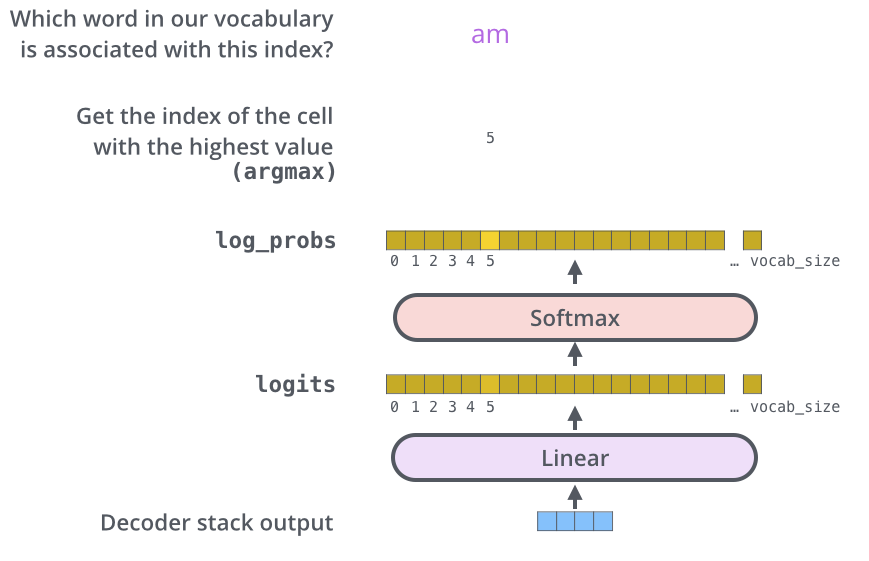
Внимание используется в трансформере в трёх местах:

* Самовнимание в энкодере — исходная последовательность обращает внимание на себя.
* Самовнимание в декодере — целевая последовательность обращает внимание на себя.
* Энкодер-декодер-внимание в декодере — целевая последовательность обращает внимание на исходную последовательность.

В декодере слой внутреннего внимания может фокусироваться только на предыдущих позициях в выходном предложении. Это делается с помощью маскировки всех позиций после текущей (устанавливая их в –inf) перед этапом софтмакс в вычислении внутреннего внимания, вспоминаем про маскировочное внимание, которые упоминалось ранее.

«Энкодер-декодер» слой внимания работает как множественное внимание, кроме того, что он создает матрицу запроса из слоя, который находится ниже, и берет матрицы ключей и значений из выхода стека энкодеров. Т.е. в энкодере-декодере запрос получается из целевого предложения, а ключ/значение — из исходного предложения. Таким образом, он вычисляет релевантность каждого слова в целевом предложении каждому слову в исходном предложении.

**Последние слои.**

Что происходит в этих слоях ?

Линейный слой – это простая полносвязная нейронная сеть, которая переводит вектор, созданный стеком декодеров, в значительно больший вектор, называемый логит вектором (logits vector).

Пусть наша модель знает 10 тысяч уникальных английский слов («выходной словарь» нашей модели), которые она узнала из обучающего корпуса. Это означает, что наш логит вектор будет иметь 10 000 ячеек в ширину – каждая ячейка соответствует коэффициенту одного уникального слова. Таким образом мы интерпретируем выход нашей модели с помощью линейного слоя.  
Слой софтмакс переводит этот показатель в вероятности (положительные числа, сумма которых равна 1). Выбирается ячейка с наиболее высокой вероятностью и на выход данного временного отрезка подается соответствующее слово.

**Время BERT.**

Фундаметально, BERT это стэк закодированных слоев Трансофматора, который состоит из множества «голов» со внутренним вниманием. Для каждого входящего токена в последовательности, в голове каждого вычисляется ключ, значение и векторы запросов, которые нужны для создания взвешенного представления. Все выходные данные всех голов в одном и том же слое соеденяются и проходят через полностью связанный слой. Каждый слой обернут пропускным соединением с последуюзим слоем нормализации.

Входящий образ считается следующим образом: каждое входящее слово это первое токенизированное в фрагмент текста и далее три вложенных слоя (токен, позиция и сегмент) соединяются в созданный вектор с фиксированной длинной. Специальный токен используется для класификации предсказаний и разделения входящий сегментов. Google и huggingFace имеют много вариантов Bert, включая оригинальный и расширенный. Они различаются в количестве „голов“, слоев и размером спрятанного состояния.

Существует две задачи обучения: Masked Language Modeling (MLM) и Next Sentence Prediction (NSP). В MLM 15% токенов статически маскируются, из них 80% заменяются токеном MASK, 10% заменяются на случайный токен, 10% остаются неизменными. Модель предсказывает верный токен, лосс считается только на этих 15% токенов. В NSP модель предсказывает, следует ли 2-й текст за 1-м. Предсказание делается на выходном векторе CLS-токена.

Для ускорения обучения 90% времени длина последовательности была 128 токенов, затем 10% времени модель училась на 512 токенах для получения валидных позиционных эмбеддингов. 16Гб обучающих данных.  
Также нельзя не упомянуть, про то, что в BERT используется wordpiece embeddings, т.е. есть какой то большой словарь из десятка тысяч слов, а в этом словаре слова мы делим на части.  
Берт кодирует слово в контексте текста. Он очень хорош для предсказания слов или даже предложений, но не для генерации текстов.  
 Предварительная тренировка на неразмеченных данных состоит их двух этапов. На первом этапе строиться "маскированная языковая модель" (**Masked LM**). Для этого **15%** входных слов заменяются на служебный токен и сеть учится эти слова восстанавливать (ошибка вычисляется только по маскированным словам). При тонкой настройке токена уже не будет. Чтобы смягчить этот факт, процедура обучения выглядит следующим образом: отбираются **15%** слов текста, **80%** их маскируется, **10%** остаётся неизменными и **10%** заменяются на случайное слово.

Второй этап предварительной тренировки языковой модели учит предсказывать следующее предложение: **Next Sentence Prediction** (**NSP**). Для этого в обучающем корпусе берётся предложение **A** и в **50%** следующее после него предложение **B**. Этот случай помечается классом **IsNext**. В остальных **50%** примерах предложение **B** является случайным, что помечается как **NotNext**. Выход **C** первого служебного токена служит для распознавания одного из этих классов.

Обсудим понимание смыслом BERT.

Большая часть доказательств про понимание здравого смысла BERT приходит из практикующих использующих его для выделения многих знаний. Одно прямо исследовательское изучение BERT докладывает, что BERT имеет проблемы с практическим выводом и понимании главного события. BERT также плохо работает с абстрактными атрибутами объектов, также хорошо как с визуальными и перцептивными свойствами, которые скорее всего будут предпологаться реже чем упоминаться.

Компонент MLM у BERT легко адаптируется для понимания индукционного заполнения в пропуски (например «Кот любит преследовать []). Petroni и другие показали, что для некоторых типов отношений, классический BERT конкурентоспособный с методами основывающимися на понимании основ, также Роберт и команда показали, что то же для открытой области QA используя T5 модель. Дэвисон и другие предпологают, что BERT делает выводы лучше для невиданной информации. Для того, чтобы BERT извлек смысл, нам нужен хорошие шаблонные предложени и их автоматическое увелечение и извлечение.

Однако, BERT не может обосновывать общее понимание. Форбс и его команда показали, что BERT может „предположить“ возможностями и свойства многих объектов, но не может обоснвать причину взаимосвязи межу возможностями и свойствами. Например, он „знает“ , что люди могут входить в дома и что дома большие, но не может заключить что дом больше чем люди. Чжоу, Ричардсон и Сабхарвал также показали, что производительность падает с увеличением числа необходимых заключающих шагов. Некоторое понимание Берта пришло из ассоциации изучения стереотипов, например, человек с именем, звучащим по Итальянски будет предсказан как Итальянец, даже тогда, когда это неверно.

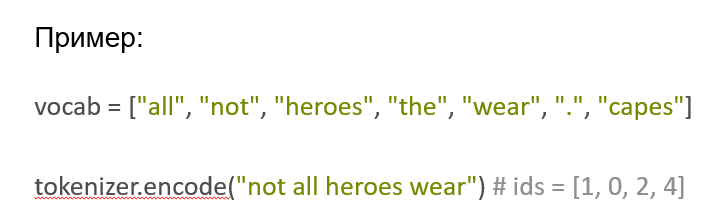
**Что такое gpt.**

GPT – Generative Pre-trained Transformer – тип структуры нейросети, основанный на трансформере. Аббревиатура означает, что GPT: генерирует текст, обучается на данных (это могут быть книги, тексты из интернета, программный код), является трансформером.

**Принцип работы gpt.**

GPT работает с «промптом» - текстом от пользователя с его запросом. GPT дописывает какое-то количество слов к промпту по принципу наибольшей вероятности нахождения таких слов подряд в реальном языке.

Строка (промпт) разбивается на подстроки меньшего размера (например, отдельные слова), называемые токенами.



Токены сопоставляются с целыми числами в соответствии вокабулярия – словаря из данных, на которых модель обучалась.

Результатом является массив i на j, где i - ая строка – номер слова в вводе, j – ый столбец в i - ой строке – вероятность, что данное слово из вокабулярия было бы встречено в языке.

Таким образом, в последней строке массива будет следующее слово для целого промпта.

**Как gpt генерирует текст.**

Мы разобрались, как gpt генерирует следующее слово, а теперь узнаем, как он может генерировать целый текст.

Методы генерации текста:

Авторегрессивный – после генерации каждого слова, в качестве промпта берётся изначальный вместе с дописанными словами, добавляется новое слово и так далее.

Сэмплирование – добавляется случайность при выводе очередного слова. Результат становится более «творческим». Это может быть и плюсом и минусом, в зависимости от задачи и степени случайности, но, в общем и целом, некоторая степень случайности используется почти во всех генеративных моделях.

**Как обучить gpt.**

Обучение происходит при помощи градиентного спуска по функции потерь. Для GPT берётся функция потерь перекрёстной энтропии. GPT может обучаться на абсолютно необработанных данных, то есть их не нужно размечать, выбирать. Поэтому найти много данных (любые тексты) для обучения модели – не является проблемой. Другое дело – вычислительные мощности. При обучении GPT компанией Яндекс было потрачено 626688 часов, по стоимости арендных вычислительных мощностей это стоило бы 109 млн.