## Большие языковые модели

## Что такое большая языковая модель (LLM)

**Большая языковая модель** (БЯМ — калька с англ. large language model, LLM) — это [языковая модель](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C), состоящая из [нейронной сети](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) со множеством параметров (обычно миллиарды весовых коэффициентов и более), обученной на большом количестве неразмеченного текста с использованием [обучения без учителя](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B1%D0%B5%D0%B7_%D1%83%D1%87%D0%B8%D1%82%D0%B5%D0%BB%D1%8F). БЯМ появились примерно в 2018 году и хорошо справляются с широким спектром задач. Это сместило фокус исследований [обработки естественного языка](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B5%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0) с предыдущей парадигмы обучения специализированных [контролируемых](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D1%83%D1%87%D0%B8%D1%82%D0%B5%D0%BB%D0%B5%D0%BC) моделей для конкретных задач.

Bag of words

Идея очень простая.

У нас есть набор текстов 1, 2, 3, 4. Мы берем все слова, которые у нас там есть. Выбрасываем, там какие-нибудь самые популярные, например, артикли, предлоги может быть

## Как устроена большая языковая модель: эмбеддинги

Проще говоря **эмбеддинг** - это способ преобразования чего-то абстрактного, например слов или изображений в набор чисел и векторов. Эти числа не случайны; они стараются отражают суть или семантику нашего исходного объекта.

В NLP, например, эмбеддинги слов используются для того, чтобы компьютер мог понять, что слова «кошка» и «котенок» связаны между собой ближе, чем, скажем, «кошка» и «окошко». Это достигается путем присвоения словам векторов, которые отражают их значение и контекстное использование в языке.

Эмбеддинги **не ограничиваются только словами.** В компьютерном зрении, например, можно использовать их для преобразования изображений в вектора, чтобы машина могла понять и различать изображения.

**Векторные пространства** — это математические структуры, состоящие из векторов. Векторы можно понимать как точки в некотором пространстве, которые обладают направлением и величиной. В эмбеддингах, каждый вектор представляет собой уникальное представление объекта, преобразованное в числовую форму.

**Размерность вектора** определяет, сколько координат используется для описания каждого вектора в пространстве. В эмбеддингах высокая размерность может означать более детализированное представление данных. Векторное пространство для текстовых эмбеддингов может иметь тысячи измерений.

**Расстояние между векторами** в эмбеддингах измеряется с помощью метрик, таких как Евклидово расстояние или косинусное сходство. Метрики позволяют оценить, насколько близко или далеко друг от друга находятся различные объекты в векторном пространстве, что является основой для многих алгоритмов машинного обучения, таких как классификация

Самый интуитивно понятный способ измерения расстояния - это **Евклидово расстояние**. В эмбеддингах расстояние вычисляется как квадратный корень из суммы квадратов разностей соответствующих компонентов двух векторов. Для двух векторов a и b, евклидово расстояние d определяется как:

Евклидово расстояние подходит для измерения абсолютных различий, но может быть менее эффективным в высокоразмерных пространствах из-за проклятия размерности.

**Манхэттенское расстояние** иногда более подходящее для определенных типов данных, это расстояние измеряется как сумма абсолютных разностей их компонентов. Оно определяется как:

Сходство между векторами измеряет степень "похожести" между двумя векторами. В эмбеддингах это используется для определения степени семантической или контекстной близости между элементами.

Одна из наиболее популярных метрик в текстовых эмбеддингах это **косинусное сходство**. Косинусное сходство измеряет косинус угла между двумя векторами. Если угол между векторами мал, косинус приближается к 1, что указывает на высокое сходство. Математически оно выражается как:

где ⋅ обозначает скалярное произведение векторов, а ∥a∥ и ∥b∥ — их нормы.

Многомерные эмбеддинги могут быть сокращены до более низких размерностей с помощью техник [PCA](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%B3%D0%BB%D0%B0%D0%B2%D0%BD%D1%8B%D1%85_%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82) или [t-SNE](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%BE%D1%85%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D0%BE%D1%81%D0%B5%D0%B4%D0%B5%D0%B9_%D1%81_t-%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5%D0%BC).

#### В общем виде, эмбеддинги делятся на:

|  |  |  |
| --- | --- | --- |
| **Категория** | **тип** | **описание** |
| Текстовые эмбеддинги | Word Embeddings | Эти эмбеддинги преобразуют слова в векторы, так что слова с похожим значением имеют похожие векторные представления. Они впервые позволили машинам понять семантику человеческих слов. |
| Текстовые эмбеддинги | Sentence Embeddings | Здесь уже идет дело о целых предложениях. Подобные модели создают векторные представления для целых предложений или даже абзацев, улавливая гораздо более тонкие нюансы языка. |
| Эмбеддинги изображений | CNN | CNN позволяет преобразовать изображения в векторы, которые затем используются для различных задач, например, классификации изображений или даже генерации новых изображений. |
| Эмбеддинги изображений | Autoencoders | Автоэнкодеры могут сжимать изображения в более мелкие, плотные векторные представления, которые затем могут быть использованы для различных целей, включая декомпрессию или даже обнаружение аномалий. |
| Эмбеддинги для других типов данных | Graph Embeddings | Применяются для работы с графовыми структурами (к примеру рекомендательные системы). Это способ представить узлы и связи графа в виде векторов. |
| Эмбеддинги для других типов данных | Sequence Embeddings | Используются для анализа последовательностей, например, во временных рядах или в музыке. |

Word2Vec использует нейронные сети для обучения векторных представлений слов из больших наборов текстовых данных. Существуют две основные архитектуры Word2Vec:

**CBOW**: предсказывает текущее слово на основе контекста (окружающих слов). Например, в предложении "Собака лает на \_\_\_", CBOW попытается угадать недостающее слово (например, "почтальона") на основе окружающих слов.

**Skip-gram**: работает наоборот по сравнению с CBOW. Использует текущее слово для предсказания окружающих его слов в предложении. Например, если взять слово "кошка", модель попытается предсказать слова, которые часто встречаются в окружении слова "кошка", такие как "мышь", "мяукает" и т.д.

Эти модели хороши в выявлении семантических отношений между словами.

#### GloVe

GloVe создает векторы слов, анализируя, как часто пары слов встречаются вместе в большом текстовом корпусе. Идея заключается в том, что смысловые отношения между словами можно выразить в виде векторных различий. Например, векторы для "король" - "мужчина" + "женщина" должны быть близки к вектору для "королева".

GloVe стремится минимизировать разницу между произведением векторов слов и логарифмом их совместной встречаемости, позволяя модели улавливать различные типы отношений между словами, такие как синонимы, антонимы, контекстные отношения и т.д

## Как устроена большая языковая модель: трансформер

Трансформер - архитектура глубоких нейросетей, которые как и RNN, предназначены для обработки текстов на естественном языке, перевода, суммаризации, но не требует обработки текста по порядку. Что открывает прекрасные возможности по распараллеливанию её работы.

Архитектура трансформера состоит из кодировщика и декодировщика. Кодировщик получает на вход [векторизованую последовательность](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2) с позиционной информацией. Декодировщик получает на вход часть этой последовательности и выход кодировщика. Кодировщик и декодировщик состоят из слоев. Слои кодировщика последовательно передают результат следующему слою в качестве его входа. Слои декодировщика последовательно передают результат следующему слою вместе с результатом кодировщика в качестве его входа.

Каждый кодировщик состоит из механизма самовнимания (вход из предыдущего слоя) и [нейронной сети с прямой связью](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C_%D1%81_%D0%BF%D1%80%D1%8F%D0%BC%D0%BE%D0%B9_%D1%81%D0%B2%D1%8F%D0%B7%D1%8C%D1%8E) (вход из механизма самовнимания). Каждый декодировщик состоит из механизма самовнимания (вход из предыдущего слоя), механизма внимания к результатам кодирования (вход из механизма самовнимания и кодировщика) и [нейронной сети с прямой связью](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C_%D1%81_%D0%BF%D1%80%D1%8F%D0%BC%D0%BE%D0%B9_%D1%81%D0%B2%D1%8F%D0%B7%D1%8C%D1%8E) (вход из механизма внимания).

Рассмотрим последовательно шаг за шагом этапы работы кодировщика:

1. На вход поступает последовательность элементов *wi*

, по ней создается последовательность эмбедингов, где каждый *xi* это векторное представление элемента *wi*

.

2. Добавляются позиционные векторы *pi*

: *hi*=*xi*+*pi*, *H*=(*h*1,...,*hn*)

. Это необходимо для того, чтобы отобразить информацию о позиции элемента в исходной последовательности. Основное свойство позиционного кодирования — чем дальше два вектора будут стоять друг от друга в последовательности, тем больше между ними будет расстояние. Более подробное устройство позиционного кодирования будет рассмотрено ниже.

3. Полученный вектор *hi*

подается на вход в блок многомерного самовнимания (англ. *multi-headed self-attention*). *hji*=Attn(*Qjhi*,*KjH*,*VjH*), где обучаемые матрицы: *Q* для запроса, *K* для ключа, *V*

для значения.

4. Затем необходима конкатенация, чтобы вернуться в исходную размерность: *h*′*i*=*MHj*(*hji*)=[*h*1*i*...*hJi*]

5. Добавим сквозные связи (англ. *skip connection*) — по факту просто добавление из входного вектора к выходному (*h*′*i*+*hi*

). После делаем нормализацию слоя (англ. *layer normalization*): *h*′′*i*=LN(*h*′*i*+*hi*;*μ*1,*σ*1)

. У нее два обучаемых параметра, для каждой размерности вектора вычисляется среднее и дисперсия.

6. Теперь добавим преобразование, которое будет обучаемым — полносвязную двухслойную нейронную сеть: *h*′′′*i*=*W*2ReLU(*W*1*h*′′*i*+*b*1)+*b*2

7. Повторим пункт 5 еще раз: добавим сквозную связь и нормализацию слоя: *zi*=LN(*h*′′′*i*+*h*′′*i*;*μ*2,*σ*2)

После, в кодирующем компоненте пункты кодировщика 3--7 повторяются еще несколько раз, преобразовывая друг за другом из контекста контекст. Тем самым мы обогащаем модель и увеличиваем в ней количество параметров.

### Позиционное кодирование

Так как в архитектуре трансформер обработка последовательности заменяется на обработку множества мы теряем информацию о порядке элементов последовательности. Чтобы отобразить информацию о позиции элемента в исходной последовательности мы используем позиционное кодирование.

Позиционное кодирование (англ. *positional encoding*) — позволяет модели получить информацию о порядке элементов в последовательности путем прибавления специальных меток к вектору входных элементов. Позиции элементов *i*

кодируются векторами *pi*, *i*=1,2,...,*n*, так, что чем больше |*i*−*j*|, тем больше ||*pi*−*pj*||, и *n*

не ограничено. Пример такого кодирования:

*p*(*i*,*s*)=⎧⎩⎨⎪⎪⎪⎪⎪⎪sin(*i*⋅10000−2*kdmodel*)cos(*i*⋅10000−2*kdmodel*)если *s*=2*k*если *s*=2*k*+1

### Self-attention

**Self-Attention** — разновидность [механизма внимания](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9C%D0%B5%D1%85%D0%B0%D0%BD%D0%B8%D0%B7%D0%BC_%D0%B2%D0%BD%D0%B8%D0%BC%D0%B0%D0%BD%D0%B8%D1%8F), задачей которой является выявление закономерности между входными данными.

Будем для каждого элемента *xi*

получать обучаемым преобразованием три вектора:

* Запрос (*query*) *qi*=*Qxi*

  Ключ (*key*) *ki*=*Kxi*

  Значение (*value*) *vi*=*Vxi*

Векторы *qi*

и *ki* будем использовать, чтобы посчитать важность элемента *xj* для элемента *xi*. Чтобы понять, насколько для пересчета вектора элемента *xi* важен элемент *xj* мы берем *kj* (вектор ключа элемента *xj*) и умножаем на *qi* (вектор запроса элемента *xi*). Так мы скалярно перемножаем вектор запроса на все векторы ключей, тем самым понимаем, насколько каждый входной элемент нам нужен, чтобы пересчитать вектор элемента *xi*

.

Далее считаем важность элемента *xj*

для кодирования элемента *xi*: *wji*=exp(⟨*qi*,*kj*⟩*d*√)∑*np*=1exp(⟨*qi*,*kp*⟩*d*√), где *d* — размерность векторов *qi* и *kj*, а *n*

— число элементов во входной последовательности.

Таким образом, новое представление элемента *xi*

считаем как взвешенную сумму векторов значения: *zi*=Attn(*Qxi*,*KX*,*VX*)=∑*np*=1*wpivp*, где *X*=(*x*1,*x*2,...,*xn*) — входные векторы. По факту *self-attention* — это *soft-arg-max* с температурой *d*−−√

. Мы перемешиваем все входные векторы, чтобы получить новые векторы всех элементов, где каждый элемент зависит от всех входных элементов.

### Multi-headed self-attention

**Multi-headed self-attention** — улучшенная модификация self-attention.

Слой внимания снабжается множеством «подпространств представлений» (англ. *representation subspaces*). Теперь у нас есть не один, а множество наборов матриц запроса/ключа/значения. Каждый из этих наборов создается случайным образом. Далее после обучения каждый набор используется для отображения входящих векторов в разных подпространствах представлений. Также появляется способность модели фокусироваться на разных аспектах входной информации.

То есть параллельно независимо несколько раз делаем attention. Потом результат каждого attention по элементам конкатенируем, затем сжимаем получившуюся матрицу и получаем для каждого элемента свой вектор той же размерности.

с*j*=Attn(*Qjq*,*KjX*,*VjX*)

, где *j*=1...*J*, *J* — число разных моделей внимания, *X*=(*x*1,*x*2,...,*xn*) — входные векторы, а *W*

— обучаемые матрицы.

## Архитектура трансформера-декодировщика

Архитектура трансформера-декодировщика[[5]](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D1%80" \l "cite_note-5)

На вход декодировщику подается выход кодировщика. Главное отличие архитектуры декодировщика заключается в том, что дополнительно имеется attention к вектору, который получен из последнего блока кодирующего компонента. Компонент декодировщика тоже многослойный и каждому блоку компонента на вход подается вектор именно с последнего блока кодирующего компонента. Разберем по порядку этапы работы декодировщика:

1. Для того, чтобы распараллелить декодировщик и уйти от [рекуррентности](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%80%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B5_%D0%BD%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D1%81%D0%B5%D1%82%D0%B8), но тем не менее генерировать элементы друг за другом, используется прием маскирования данных из будущего. Идея в том, что мы запрещаем себе подглядывать в те элементы, которые еще не сгенерированы с учетом порядка. Когда генерируем элемент под номером *t*

, имеем право смотреть только первые *t*−1 элементов: *ht*=*yt*−1+*pt*; *Ht*=(*h*1,...,*ht*)

2. Далее идет этап многомерного самовнимания: линейная нормализация и multi-headed self-attention. Особенность в том, что в attention ключи и значения применяются не ко всем векторам, а только к тем, значения которых уже синтезировали (*Ht*

): *h*′*t*=LN∘*MHj*∘Attn(*Qjht*,*KjHt*,*VjHt*), где ∘

— композиция.

3. На следующем этапе мы делаем многомерное внимание на кодировку *Z*

, результат работы компонента кодировщика: *h*′′*t*=LN∘*MHj*∘Attn(*Qjht*,*KjZ*,*VjZ*)

4. Линейная полносвязная сеть (по аналогии с кодировщиком): *yt*=LN∘*FNN*(*h*′′*t*)

5. В самом конце мы хотим получить вероятностную порождающую модель для элементов. Результат (индекс слова с наибольшей вероятностью): SoftArgMax(*Wyyt*+*by*)

, где *Wy*, *by* — обучаемые параметры линейного преобразования. Для каждой позиции *t* выходной последовательности мы строим вероятностную модель языка, то есть все элементы из выходного словаря получают значение вероятности. Эти значения как раз получаются из векторов *yt*

из предыдущего пункта, которые мы берем с последнего блока трансформера-декодировщика.

Последний этап выполняется только после того, когда повторились пункты 1--4 для всех декодировщиков. На выходе получаем вероятности классов, по факту для каждой позиции решаем задачу многоклассовой классификации, для того, чтобы понять какие элементы лучше поставить на каждые позиции.