ЦИФРОВАЯ КУЛЬТУРА

УНИВЕРСИТЕТ ИТМО

Векторная семантика

Высшая школа цифровой культуры

Университет ИТМО

dc@itmo.ru

Оглавление

Введение

В данной лекции речь пойдёт о векторной семантике. Семантикой называется изучение значений слов, а векторная семантика — это раздел NLP, который представляет значения слов в виде векторов.

Это одна из самых базовых и самых важных задач в NLP — прежде всего потому, что для большинства остальных задач слова́ **или токены**, которые содержатся в тексте, нужно представить в векторном формате, причём таким образом, чтобы эти векторы содержали значимую информацию о словах, которые представляют.

Именно для этого и нужна векторная семантика — она пытается создать релевантные векторные **представления (или репрезентации)** слов, чтобы алгоритмы, которые потом **будут использовать** эти **представления**, могли **явно или неявно** извлечь **из них** нужную для себя информацию и успешно решить задачу. Скажем, для задачи классификации можно обучить нейронную сеть - и на вход **ей в качестве признаков** подавать векторы слов в тексте. А если данных недостаточно для обучения нейросети, можно просто сравнивать тексты с образцами **из** каждого класса - и для такого сравнения можно, **например,** представить тексты как усреднённые векторы слов.

Кроме того, векторная семантика позволяет изучать язык сам по себе. Например, благодаря представлению слов в виде векторов можно визуализировать слова в языке - на таком **размещении слов в пространстве** будет видно, какие **из них** обладают близкими значениями, а какие удалены **друг от друга**. Можно отслеживать изменения в употреблениях слов - на иллюстрации показано, как изменилось употребление слова “маска” в 2020 году по сравнению с 2019. Можно находить лишнее слово из данного ряда и делать множество подобных вещей.

Как и во множестве других задач в **обработке естественного языка**, **здесь** доминирует подход, основанный на данных. Значения слов моделируются через закономерности их употребления в корпусе. Мы не пытаемся напрямую объяснить компьютеру, что, скажем, верёвка - это что-то длинное и гибкое; вместо этого мы предполагаем, что все подобные сведения уже содержатся в употреблениях слова “верёвка” в различных контекстах.

Предположение о том, что слова, которые употребляются в схожих контекстах, имеют схожие значения, называется дистрибутивной гипотезой (а соответствующее направление NLP часто называется дистрибутивной семантикой). Лучше всего эта идея

передаётся в знаменитой цитате английского лингвиста Джона Фёрса - основателя лондонской школы лингвистики: “You shall know a word by the company it keeps”, то есть о слове судят по его компании. Можно вспомнить и другую его цитату: о том, что значение слова всегда контекстуально, и нет смысла исследовать значение слова, не занимаясь его контекстами.

Дистрибутивная гипотеза важна для векторной семантики, потому что она позволяет нам использовать контексты слова в качестве приближения его значения. Если слова с похожими контекстами обладают похожими значениями, то мы можем в качестве векторов слов использовать векторы, представляющие контексты слова. И сходство или различие слов будет передаваться через бОльшую или меньшую близость векторов, представляющих их контексты: скажем, у слов “бегемот” и “гиппопотам” будет гораздо больше общих контекстов, чем у слов “бегемот” и “теплоснабжение”.

Итак, мы решили, что схожесть слов - это почти то же самое, что схожесть контекстов. Но как определить схожесть контекстов? Интуитивно кажется, что это должно быть связано с общими вхождениями: слова со схожими контекстами встречаются в одинаковых или похожих текстах. Но тут нужно различать два разных случая схожести контекстов.

Слова, которые часто оказываются рядом друг с другом, можно считать соседями первого порядка - например, такими соседями, скорее всего, окажутся слово “выпил” и слово “чаёк”, потому что у них много совместных вхождений первого порядка. Это синтагматическая, или ассоциативная схожесть.

Слова, которые имеют похожих соседей - это уже соседи второго порядка. Например, слова “Татры” и “Карпаты”, будучи названиями гор, оказываются в приблизительно одинаковых контекстах - у них много совместных вхождений второго порядка. Это парадигматическая схожесть.

И теперь, когда у нас есть подобные определения, мы можем перейти непосредственно к векторной семантике.

Наш план таков. Сначала мы узнаем, как оценивать качество наших векторов слов, **а также какие наборы данных** и инструменты важны для векторной семантики. Затем мы обсудим наиболее простой тип векторов слов - это векторы слов, основанные на прямом подсчёте слов, встречающихся в контекстах. Затем мы научимся получать **другие** векторные представления слов с помощью методов линейной алгебры. **Затем нам придётся разобраться с азами** нейронных сетей и их применения в NLP. Эти знания понадобятся нам в следующем разделе - там мы научимся использовать

нейросети для получения векторов слов и узнаем, как заставить такие нейронные векторы слов отражать не только общий смысл слова, но и особенности его употребления в отдельном тексте.

1. Оценка, датасеты и инструменты

Для того чтобы приступить к работе над векторизацией слов, нам прежде всего понадобятся способы оценки векторных представлений: ведь нам нужно узнать, насколько хорошо **некоторые** векторы слов подходят для наших целей и насколько полноценно они передают свойства слов, которые мы хотим представить. Кроме того, для оценки алгоритмов векторной семантики нужны размеченные данные - так что мы обсудим, какие датасеты доступны и особенно часто применяются в векторной семантике. Мы также узнаем, какими существующими инструментами можно воспользоваться, чтобы векторизовать слова.

Недавно мы выяснили, что векторы слов нужны **в том числе** для двух целей — во-первых, как этап обработки (а именно векторизация) текстовых данных перед их использованием в других алгоритмах и для других задач, а во-вторых — для исследования языка, так сказать, изнутри. Способы оценки векторов слов тоже бывают двух типов — внешние (extrinsic) и внутренние (intrinsic).

Внешняя оценка векторов предполагает использование векторов слов для решения стандартных задач обработки естественного языка — скажем, для классификации текстов. Для этого нужно взять какой-нибудь стандартный алгоритм классификации текстов и подать ему на вход тексты, векторизованные несколькими способами. После этого можно посмотреть, как меняется качество работы алгоритма в зависимости от выбранного способа векторизации. Если с одними векторами алгоритм работает значительно *лучше*, чем с другими, значит, эти векторы лучше передают свойства слов в текстах - или, во всяком случае, лучше подходят для нашей задачи.

Внутренняя оценка подразумевает сравнение свойств векторов слов с человеческими суждениями о том, как устроены слова и какими свойствами они обладают. Например, если мы считаем, что слово “бегемот” больше похоже на “гиппопотам”, чем на слово “теплоснабжение”, то разумно ожидать, что и вектор слова “бегемот” будет **ближе к** вектору слова “гиппопотам”.

Для внутренней оценки существует множество разных подходов. В частности, один из стандартных методов оценки - сравнение близости векторов слов с близостью

слов по мнению экспертов-разметчиков. (Для этого нужен **набор данных**, состоящий из пар слов, каждой из которых приписана какая-то оценка сходства - например, от нуля до десяти. Пример такого датасета вы видите на слайде). Для такого сравнения нужно вычислить коэффициент корреляции Спирмена между человеческими оценками словесного сходства и векторной близостью репрезентаций тех же слов.

Отвлечёмся, чтобы рассказать или напомнить, что такое коэффициент Спирмена. Возможно, вы слышали о коэффициентах корреляции -- если даны две переменные, коэффициенты корреляции помогают выяснить, существует ли между ними зависимость.

Например, коэффициент Пирсона (формула которого представлена на слайде) показывает наличие или отсутствие линейной зависимости. Высокий коэффициент корреляции означает, что существует зависимость одной переменной от другой (а высокий по модулю, но отрицательный коэффициент означает обратную зависимость).

В нашем случае наличие зависимости между человеческими и векторными оценками означает, что сравнение слов тем и другим способ**ами** даёт приблизительно один и тот же результат, и, значит, векторные представления хорошо передают близость слов.

Однако показатели вроде коэффициента Пирсона не совсем подходят для сравнения оценок сходства слов. Зависимость между оценками ведь не обязательно должна быть линейной -- главное, чтобы одни и те же пары слов оказались более близкими или более далёкими (**например**, чтобы “гиппопотам” оказался ближе к “бегемоту”, чем к “теплоснабжению”). Поэтому мы применяем коэффициент Спирмена - он показывает не линейную, а **так называемую** ранговую корреляцию.

На слайде представлен пример -- четыре пары слов, каждой из которых присвоена оценка сходства. Эта оценка сходства -- и есть наша переменная. Кроме того, оценке каждой пары слов присвоен свой ранг. Ранг - это порядковый номер данного значения, которое оно получило бы, если бы все значения отсортировали по убыванию, от большего к меньшему. Например, ранг пары “акции - рынок” -- единица, потому что оценка этой пары выше всех остальных, а ранг пары “акции - ягуар” - четвёрка, потому что эти два слова почти никак не связаны друг с другом. Вместо того чтобы оценивать зависимость значений двух переменных, ранговая корреляция сравнивает ранги этих значений.

(Из формулы корреляции Спирмена, которая представлена на слайде, видно, что значение коэффициента Спирмена - это то же самое, что коэффициент Пирсона между

рангами переменных вместо самих переменных. При этом можно использовать упрощённую формулу корреляции Спирмена, которая также представлена на слайде.)

Если ранговая корреляция велика, значит, человеческие и векторные оценки близости, **можно сказать,** “идут в одном порядке”. Даже если значения слегка различаются, “бегемот” всегда будет ближе к “гиппопотаму”, чем к “теплоснабжению”.

Большое значение ранговой корреляции с человеческими оценками означает, что алгоритм векторной семантики, который мы исследуем, хорошо отражает **мнение экспертов о** свойствах слов.

Ещё один способ внутренней оценки - это аналогии. Предполагается, что если взять вектор слова “Москва”, а затем отнять **от него** вектор слова “Россия” и прибавить вектор слова “Франция”, то получившийся вектор должен быть близок к вектору слова “Париж”. Мы как бы решаем словесное уравнение: “Москва относится к России так же, как Париж относится к иксу”. Оказывается, что **некоторые** хорошие векторы слов можно использовать для решения подобных уравнений.

Эти два подхода к внутренней оценке - самые распространённые и стандартные. Но существуют и другие - например, через оценку кластеризации векторов слов.

Для внутренней оценки моделей дистрибутивной семантики существует множество датасетов, которые отличаются языком и постановкой задачи. **Одним из самых старых и известных** из них является WordSim353 - датасет из 353 пар английских слов, каждой из которых разметчики приписали какую-то оценку от 0 (**ноль нужно ставить если слова в паре** совсем непохож**и**) до 10 (одно и то же слово). Например, сходство слов stock “акции” и market “рынок” оценили в 8.08, а “stock” и “jaguar” (ягуар) - в 0.92. Отрывок из WordSim353 **представлен** на слайде. Кроме этого, есть датасет SimLex999, для которого важна именно синонимИя, а не любое сходство контекстов - например, слова “шкаф” и “одежда” считаются очень похожими в первом датасете, но совсем непохожими во втором, потому что шкаф и одежда - это всё-таки совсем не одно и то же. Стоит упомянуть также стэнфордский датасет сходства для редких слов и созданный для русского языка датасет Human Judgement.

Стандартным **набором данных для оценки** словесных аналогий является google analogies test set (фрагмент из него также представлен на слайде). Он разделён на две части - семантическую и синтаксическую. Для аналогий из первой части нужно понимать скорее смысл слова (например, подобрать антонимы), а для второй части - грамматику (например, от слова “делать” по аналогии образовать слово “делающий”). А **набор данных** Deep Semantic Analogies, включающий в себя английский и немецкий

языки, содержит датасеты **для** нескольких задач, в том числе поиск**а** синонимов, антонимов или гиперонимов. **Гиперонимы -- это слова, обозначающие более общие понятия, чем данное слово. Так, для слова “пудель” гиперонимом является слово “пёс”.**

Для русского же языка существуют **наборы данных**, использованные в рамках воркшопа RUSSE, включая датасеты **с указанием** сходства слов, синонимов, гиперонимов, а также датасет ассоциаций.

Для того, чтобы использовать дистрибутивную семантику в своей работе, нужно знакомство с существующими инструментами. Самый распространённый из них называется gensim (“**дженсим**”). Это библиотека на python’е, которая совместима со многими **распространёнными реализациями** модел**ей** дистрибутивной семантики и позволяет совершать все элементарные операции: преобразовать слово в вектор, найти ближайших соседей слова, решить задачу аналогии и так далее. Кроме того, у многих моделей есть свои собственные библиотеки. Например, word2vec, fasttext, glove - это названия моделей дистрибутивной семантики и одновременно имена библиотек в python’е (что не отменяет возможности использовать эти модели через gensim).

Кроме того, часто выкладываются уже обученные векторные модели слов. В частности, такие модели, обученные на корпусах русского языка, можно скачать с сайта https://rusvectores.org/

Отдельно нужно сказать про инструменты для работы с контекстуализированными векторами. Это особенный тип векторов слов, он основан на глубоких нейронных сетях, и такие векторы нельзя использовать через библиотеки вроде gensim - вам понадобятся специализированные библиотеки для глубокого обучения, такие как tensorflow или torch. Для получения контекстуализированных векторов я рекомендую также библиотеку transformers. Она совместима со многими современными языковыми моделями, которые помогают извлекать контекстуализированные векторы слов.

(О том, что такое контекстуализированные векторы, мы ещё поговорим дальше, а языковым моделям будет посвящена отдельная лекция.)

2. Разреженные векторы

Первая группа **методов**, которой мы воспользуемся для векторного моделирования значений слов — это подходы, основанные на прямом подсчете слов,

которые встречаются в **тех или иных** контекстах. **Но что это значит?** При таком подходе каждая компонента вектора, обозначающего слово, передаёт информацию о **так называемых** контекстах этого слова — например, о количестве **совместных упоминаний недалеко от другого слова** или о частотности данного слова в определенном документе. Такие векторы называются разреженными, потому что многие компоненты вектора равны нулю — это **значит**, что слово не имеет общих контекстов со многими словами и входит в состав не всех документов. В этом отличие разреженных векторов от плотных, в которых значения компонент обычно ненулевые. Кроме того, разреженные векторы обычно проще интерпретировать: каждая компонента несёт конкретную информацию о встречаемости слова, тогда как плотные векторы либо содержат более тонкие признаки слов, либо могут интерпретироваться только геометрически — как точки в векторном пространстве, каждая из которых может быть ближе или дальше от другой точки, но отдельная компонента вектора не содержит информации в явном виде.

Ранее мы обсуждали разницу между синтагматической и парадигматической схожестью. При синтагматической схожести одно слово часто появляется рядом с другим -- как слово “выпил” и слово “чаёк”. При парадигматической схожести у слов похожие соседи, как у слов “Татры” и “Карпаты”.

Для того, чтобы передать первый тип **сходства**, используется **представление** “термы-документы”. В рамках этого подхода **близки друг к другу** векторы слов, которые встречаются в одинаковых документах. Для этого можно использовать term-document matrix - с этим понятием мы уже сталкивались в курсе, но я напомню, что это матрица размера n на m, где n - размер словаря, а m - число документов в коллекции. На пересечении n-ной строки и m-того столбца находится число вхождений n-ного слова в m-тый документ. Раньше нас интересовали столбцы этой матрицы - каждый из них является “мешком слов”, то **есть** вектором, который показывает число вхождений в текст различных слов из словаря.Теперь же мы обращаем внимание на её строки: i-той компонентой в строке будет число вхождений в i-тый документ из коллекции, так что два вектора слов будут похожи, если два соответствующих слова появляются примерно в **одних и тех же** документах.

Обратите внимание, что это синтагматическое сходство: два слова считаются похожими, потому что они встречаются в одних и тех же документах, то есть часто оказываются друг с другом в одном контексте.

Однако у такого подхода есть недостатки. Качество такого представления

сильно зависит от коллекции документов — если документов недостаточно много или тематики, к которым принадлежат тексты, распределены неравномерно, векторы слов не будут адекватно представлять их свойства.

Если же коллекция документов достаточно велика, то размерность векторов слов пропорционально увеличится, так как она равна числу документов в коллекции. При этом для слов, которые встречаются во многих документах, векторы не будут очень разреженными, и это затруднит вычисления.

Чтобы решить эту проблему, нужно спуститься от уровня целых документов до уровня коротких последовательностей слов — контекстов. Для этого нам понадобится **так называемая** word-context matrix. Пример такой матрицы представлен на слайде: на пересечении i-той строки и j-того столбца находится число вхождений j-того слова в словаре в контексты i-того слова. Контексты при этом можно определять по-разному: можно, например, рассматривать окно в два слова слева и два слова справа от данного слова.

Важная деталь: word-context matrix -- обычно квадратная матрица. То есть, как правило, вычисляют сколько раз то или иное слово побывало в одном контексте с другим. Строки соответствуют целевым словам -- то есть это те самые векторы, которые мы хотим получить. Столбцы же соответствуют контекстным словам -- их иногда тоже так и называют контекстами. Здесь важно не путаться.

При этом влияние состава коллекции документов на векторы уменьшается: мы учитываем только вхождение слов в непосредственной близости от данного слова, а не текст документа целиком. Размерность вектора больше не зависит от размера коллекции — она равна размеру словаря. При этом почти все компоненты будут равны нулю, что облегчает вычисления.

Кстати, заметьте, что мы перешли от синтагматической схожести к парадигматической: теперь похожими словами считаются слова с похожими контекстами (так сказать, с “одинаковыми соседями”), а не слова, которые в текстах часто бывают вместе (сами являются соседями).

Проблемой такого подхода являются распространенные слова, которые по той или иной причине встречаются почти везде в текстах. Сюда относятся, например, такие слова, как “он”, “этот”, “если” - то есть слова, которые организуют текст и могут попасться в контексте любого слова. Они не добавляют в вектор слова почти никакой информации (соседство со словом вроде “этот” не позволяет отличить **смысл** одно**го** слов**а** от другого), но при этом компоненты в векторе, соответствующие этим словам,

будут иметь очень большие значения. Это вредно для алгоритмов, которые пользуются векторами слов. Они будут некорректно предполагать, что соседство именно с этими словами является важной особенностью конкретного слова и что на эти слова-соседи нужно обращать особое внимание. Это плохо сказывается на качестве работы алгоритмов, потому что общие контексты с такими словами не содержат никакой важной информации.

Для того, чтобы справиться с этой проблемой, используется взвешивание. При взвешивании каждому слову присваивается **особый** коэффициент (вес), который должен быть меньше у популярных слов, бесполезных для векторной семантики, и больше у редких и специфичных слов, которые помогают лучше описать употребление других слов. Например, у слова “этот” такой коэффициент должен быть очень мал - оно может употребляться рядом с любыми словами, а у слова “идемпотентность” коэффициент должен быть большим: слова, с которыми оно употребляется, скорее всего, относятся к математике или информатике.

Первый метод взвешивания, который мы разберём**, точнее, вспомним,** называется ти-эф ай-ди-эф. Это сокращение от “text frequency - inverted document frequency”, то есть “частотность в тексте - обратная частотность в документах”. Как видно из формулы, представленной на слайде, tf-idf - это произведение двух чисел - tf и idf.

tf - это функция от токена и документа, она **возвращает** число вхождений токена в данный документ. Это то самое число, которое мы использовали в word-document matrix.

А idf - это и есть вес или коэффициент, на который мы умножаем число вхождений. idf является функцией, которая принимает на вход токен и коллекцию документов. Эта функция вычисляет дробь, у которой в числителе - размер коллекции, а в знаменателе - число документов, в которых хотя бы один раз встречается данный токен. Иными словами, эта дробь - обратная доля документов, содержащих этот токен.

Например, если у нас 1000 документов, а токен встречается в 700 из них, то доля документов - 70%, а обратная доля - 1 / 0.7, то есть примерно 1.43. На выход эта функция отдаёт логарифм этой дроби. (Как известно, логарифм числа, умноженный на минус единицу, равняется логарифму обратного числа. Так что можно считать, что idf - это минус логарифм доли документов, в которые входит данное слово). Для слов, которые встречаются во многих документах, обратная доля документов с этим словом чуть больше единицы, и логарифм будет положительным, но близким к нулю. Для

редких слов обратная доля будет положительным и большим по модулю числом, и логарифм - тоже. (При этом логарифм растёт медленнее, чем просто обратная доля документов с данным токеном).

На слайде представлен пример вычисления tf-idf для двух слов в некой коллекции. Распространённое слово “этот” при этом получает меньшее значение, чем информативное слово “идемпотентность”, хотя и появляется в документе в шесть раз чаще.

TF-IDF хорошо подходит для матриц типа термы-документы и вообще для ситуаций, где документ нужно представить через входящие в него слова. Например, если вам нужно усреднить векторы всех слов в документе, чтобы получить вектор документа, имеет смысл умножить вектор каждого слова на idf этого слова.

Но для матриц типа “термы-термы”, **например**, **“word-context” матриц,** он не подходит, потому что в нём существенным образом используется текст документа и коллекция документов - вычислить TF-IDF на основе набора контекстов из нескольких слов не получится. Взвесить матрицу типа “термы-термы” можно, **например**, с помощью показателя пи эм ай.

PMI расшифровывается как pointwise mutual information, **по**точечная взаимная информация. Формула данного показателя представлена на слайде. Пи-эм-ай равен логарифму дроби, у которой в числителе - вероятность встретить два слова вместе, а в знаменателе -- произведение вероятностей **встретить каждое из них**. Его суть в том -- в том, чтобы выяснить, сколько у данных слов было **бы** общих контекстов, если бы они были распределены независимо, и сравнить это число с настоящим числом общих контекстов. **Чем больше эта величина, тем менее случайным можно считать попадание данных слов в один контекст.**

Скажем, одно слово встречается в контекстах с вероятностью 10%, а другое -- с вероятностью 20%. Тогда при независимом распределении вероятность встретить их вместе равнялась бы 2%. Если же на практике они встречаются вместе в 10% контекстов, то есть в пять раз чаще, то их пи-эм-ай будет равен логарифму тройки.

При этом негативные значения PMI обычно не очень информативны и их рекомендуют заменять на нулевые. Такой показатель называется пи-пи-эм-ай (то есть positive PMI).

3. Плотные векторы

Подходы, которые мы обсуждали до сих пор, сводятся к явному описанию слов через простые признаки, такие как количество **появлений в** контекст**ах**, включающих те или иные слова из словаря. Чтобы получать более качественные векторные представления, нужно научиться извлекать другие, более релевантные признаки слов. Они должны передавать как можно больше значимой информации о свойствах слов как можно меньшими средствами. Для этого используются **так называемые** плотные векторы слов **или плотные векторные представления**. Как правило, такие векторы тоже основаны на информации о совместном употреблении слов. Но у них меньшая размерность в сравнении с разреженными векторами, и компоненты, **как правило,** не равны нулю.

Можно сказать, что Информация, представленная в плотных векторах, сжата. Это значит, что размерность векторов гораздо меньше, чем размер словаря или коллекции документов, а следовательно, в плотных векторах содержатся, **так сказать, закодированы** только самые важные характеристики слова. Ведь у нас нет возможности описывать связь данного слова со всеми словами в словаре или документами в коллекции. Именно это и позволяет плотным векторам **точнее** представлять значения слов.

При этом интерпретируемость таких векторов страдает — ясно понять, почему данное число стоит в данной компоненте вектора, затруднительно. **О** таких вектор**ах** можно думать скорее как **о** геометри**ческих** точк**ах** в **многомерном** ~~векторном~~ пространстве, чем как **о** закодированных значениях заранее заданных признаков слов. Интерпретировать одну точку можно только **путём** сравнени**я** с другими точками, а не саму по себе. Такая интерпретация может, например, включать поиск ближайших соседей данной точки или анализ паттернов, которы**е** такие точки образуют в векторном пространстве.

Мы разберем два способа создания плотных векторов — через матричное разложение и через предсказание контекста слова.

Начнём с матричного разложения. Чаще всего используется способ, называемый “сингулярным разложением”, или singular value decomposition. Представьте себе, что у вас есть **прямоугольная** матрица M. Когда речь идет о дистрибутивной семантике, это может быть матрица термы-документы (в таком случае число строк равно размеру словаря, а число столбцов - числу документов в коллекции), или же матрица

слова-контексты (и в этом частном случае она квадратная, эм равен эн равен размеру словаря). Можно доказать, что любую такую **вообще говоря прямоугольную** матрицу можно представить в виде произведения трех матриц: ортогональной матрицы у, диагональной матрицы (эс) **S** и ортогональной матрицы **V^T** (вэ транспонированное). Что означают эти три матрицы?

Например, мы пытаемся разложить матрицу термы—документы, размера w (дабл-ю) строк на d (дэ) столбцов (кстати, **если специальным образом настроить веса термов в документах,** это **будет называться** латентно-семантическим анализом). В результате получится три матрицы: матрица U (у), размера w (дабл-ю) на f (эф), диагональная матрица S (эс) размера f (эф) на f (эф), и матрица **V^T** (вэ транспонированное) размера **f** (эф) на d (дэ).

Первая матрица - матрица **U** (У) - содержит столько же строк, сколько слов в словаре, и каждая строка является векторным представлением конкретного слова.

Третья матрица - матрица **V^t** (Вэ транспонированное) - содержит столько же столбцов, сколько документов в коллекции. Здесь применяется та же логика: каждый столбец является вектором отдельного документа.

И в представлениях слов, и в представлениях документов содержится **f** (эф) компонент, и каждая компонента в векторе — это отдельный признак в векторном представлении.

Вторая матрица — диагональная матрица **S** (Эс) — нужна для того, чтобы масштабировать эти признаки. Значения на диагонали соответствуют каждому признаку: **обычно считается, что** чем признак важнее, тем больше соответствующее ему значение в диагональной матрице. При умножении матрицы на матрицу **S** (Эс) i-тая компонента в строках (для умножения справа) или столбцах (для умножения слева) умножается на i-тое значение диагональной матрицы. Таким образом, важные и релевантные признаки в векторах слов и документов увеличиваются, а остальные уменьшаются.

Благодаря такому разложению мы в каком-то смысле устанавливаем связь между словами и текстами.

Для того, чтобы ещё сильнее уменьшить размерность (например, если число **R** (ар) велико), мы можем выбрать некое число **k** ка (скажем, сотню) самых важных признаков в векторе и выбросить все остальные. Для **этого** можно воспользоваться нашей диагональной матрицей, а именно отсортировать **её ненулевые значения** и выбрать из них **k** (ка) самых больших. Такой метод называется truncated SVD.

Насколько это хорошая стратегия? Обратите внимание, что если перемножить эти три матрицы, получится матрица прежнего размера, но меньшего ранга — ее ранг будет равняться числу **условно важных отобранных нами признаков**. Теорема Эккарта-Янга утверждает, что эта матрица является наилучшим приближением исходной матрицы среди всех матриц с таким рангом. Так что данный метод эффективно снижает размерность векторов слов и в то же время эффективно сохраняет информацию.

Сингулярное разложение используется в обработке естественных языков для получения векторных представлений слов и документов. **Одно из основных** применени**й SVD** эс ви ди для документов - латентно-семантический анализ, то есть разложение матрицы термы-документы, которое мы сейчас разобрали в качестве примера. Для получения векторов слов можно использовать разложение матрицы “слово-контекст”, например, применяя truncated SVD к PPMI-матрице. Плюсом данного подхода является способность успешно извлекать качественные векторные репрезентации, а использование топ-ка наиболее релевантных признаков позволяет удалять шум из векторных **представлений**. **Недостаток** сингулярного разложения - **большая** вычислительн**ая** сложн**ость**. Поскольку размеры словаря и коллекций документов (а следовательно - и размерности матриц) в обработке естественного языка велики, ~~операция~~ **вычисление** сингулярного разложения может занимать значительное время.

Ещё один из подходов к матричному разложению называется NNMF -- **non-negative matrix factorization**, то есть неотрицательная матричная факторизация. Это **представление неотрицательной матрицы V** (вэ) в виде произведения матриц **W** (дабл-ю) и **H** (эйч). Особенностью данной операции является то, что все три матрицы неотрицательны -- все их элементы больше или равны нулю. Для анализа текстов, **к примеру,** применяют разложение термы-документы: при этом получа**ю**тся матрица термы-признаки и матрица признаки-документы. Логика тут похожа на то, что мы уже обсудили в случае сингулярного разложения -- строки из матрицы термы-признаки используются в качестве векторных представлений слов, **а** столбцы из матрицы признаки-документы -- **в качестве представлений** документов. Для того, чтобы найти такую матрицу, обычно используют численные методы аппроксимации: **значения в матрицах W** дабл-ю и **H** эйч задаются случайно, и затем итеративно исправляются, пока их произведение не станет достаточно близк**им** к исходной матрице **V** вэ.

4. Нейронные сети

Следующий класс методов дистрибутивной семантики - это предиктивные методы. Они относятся к плотным векторам слов, так же как и векторы на основе матричного разложения. Но, в отличие от векторов на основе матричного разложения, они основаны на предсказании контекста слова **в том числе** с помощью нейронных сетей.

Для того, чтобы обсудить основанные на предсказаниях методы дистрибутивной семантики, нам понадобится краткий экскурс в нейронные сети.

Нейронные сети - это собирательное название для множества алгоритмов в машинном обучении.

Как и все алгоритмы машинного обучения, они нужны для того, чтобы решать задачи, чье решение нельзя выписать алгоритмически, напрямую. Вместо этого нейронные сети используют большое количество данных (например, тексты на английском и их переводы на немецкий), чтобы самостоятельно выделить в них закономерности и научиться воспроизводить их (например, переводить тексты с английского на немецкий). Нейронные сети - не единственные алгоритмы в машинном обучении, но они используются чаще других, и им посвящается особенно большое число исследований.

Многослойным нейронным сетям посвящен целый раздел машинного обучения - глубокое обучение.

Нейронные сети настолько важны потому, что способны усваивать и воспроизводить самые разнообразные закономерности в данных **необычайно хорошо**.

С их помощью, например, **можно** добавлять к картинкам подписи на естественном языке, соответствующие содержанию этих картинок; генерировать реалистичные изображения несуществующих вещей или людей, производить тексты, похожие на написанные человеком; играть в шахматы. Все эти задачи требуют хорошего понимания того, как устроены данные, которые поданы на вход нейронной сети, будь то тексты, изображения или шахматные партии.

Нейронные сети называются так потому, что элементарная составляющая их архитектуры называется нейроном (по аналогии с крошечными нервными клетками человека и животных). Нейрон можно представлять в виде машины, у которой несколько входов и один выход (они называются входными и выходными связями).

Каждой входной связи присваивается некое число - так называемый вес. На каждый вход нейрону поступает некое число, которое затем умножается на вес, соответствующий этому входу.

Потом эти произведения суммируются, и сумма подается на вход в нелинейную функцию,

которая называется функцией активации. Нелинейность здесь очень важна - если бы нейроны выражали линейные функции, то и вся нейронная сеть (будучи композицией линейных функций) была бы линейной, и ее не нужно было описывать подобным образом. Нейроны, как правило, **организуют** в слои, где каждый нейрон принимает выходы нейронов предыдущего слоя, а выход нейрона сам передается в следующий слой.

Примером этого является полносвязная сеть, также известная как перцептрон - в ней каждый нейрон предыдущего слоя можно связать с каждым нейроном последующего.

На слайде изображен перцептрон, состоящий из двух слоев, не считая входного. Его работа также выражена формулами, которые приведены на слайде.

На вход он принимает векторы, состоящие из четырех компонент (матрица X1, размерностью n на 4, где n — число примеров) - на изображении этому соответствует тот факт, что во входном слое четыре входа. Затем матрица X1 умножается на матрицу весов скрытого слоя W1 - это матрица размера 4 на 5, потому что в скрытом слое 5 нейронов. Полученное произведение проходит через функцию активации, которая в формуле обозначена буквой сигма - получается “скрытое состояние нейросети”, матрица X2 размером n на 5. Затем таким же образом X2 умножается на веса выходного слоя W2, проходит через нелинейную функцию, и для каждого примера остается одно число - значение активации выходного нейрона. Это и есть предсказание нейросети для данного объекта - матрица X3, размера n на 1.

Как же узнать, какими должны быть весовые коэффициенты между нейронами? В этом и заключается трудность! Когда мы говорим что-то вроде “трехслойная нейронная сеть с десятью нейронами на первом и втором слое и двадцатью нейронами на третьем”, мы описываем не конкретную функцию, а скорее огромное пространство функций, которое задается параметрами - весами. Задача состоит в том, чтобы найти в этом пространстве функцию, которая может решить нужную нам проблему (например, классификацию текстов, машинный перевод и так далее) - или, что эквивалентно, нам нужно найти, какими должны быть связи между нейронами. Такой поиск весов и

называется обучением. Когда веса найдены, то обучение завершено, и нейронную сеть можно использовать в работе.

В общих чертах обучение происходит так. Сначала нейросеть инициализируется случайным образом - из нашего огромного пространства функций мы выбираем одну конкретную точку. Как правило, такая необученная нейросеть справляется с задачей плохо - например, если попросить ее классифицировать тексты, она будет проставлять метки попросту наугад. **Самая важная хитрость --** найти небольшие поправки к весам нейросети, чтобы слегка "**подкорректировать"** ее параметры, в результате чего она будет справляться с задачей немного лучше, чем прежде. Такую процедуру необходимо повторять снова и снова, множество раз, пока не будет соблюден критерий остановки (**например**, качество работы нейросети перестало расти или мы превысили максимальное число **шагов**, или итераций). Если вообразить себе пространство функций, это можно сравнить с путешествием, которое мы начинаем в случайной точке, а затем небольшими шажками приходим к точке, соответствующей лучшей (или просто приемлемой) функции.

Технически это делается так. Качество (а точнее, "некачественность") работы нейросети на некоторых данных (обучающей выборке) выражается функцией по нашему выбору - вне зависимости от того, какую функцию выбрали, она при обучении называется функцией потерь, **невязкой** или лосс-функцией.

На вход она принимает все параметры нейросети, а на выход отдает единственное число - "потери", "лосс" - которое тем выше, чем больше ошибок совершает нейросеть.

Например, для классификации часто используется кросс-энтропия, которая равна нулю при идеально правильном ответе нейросети и тем выше, чем больше разница между ответом нейросети и правильным ответом. Формула кросс-энтропии представлена на слайде.

Функция потерь обязательно должна быть дифференцируемой. **Не будем обсуждать стандартную схему обучения нейронных сетей во всех подробностях, рассмотрим её в обзорном порядке.** Для того, чтобы улучшить качество работы нейросети, мы вычисляем частную производную функции потерь относительно каждого параметра в сети, а затем отнимаем число, пропорциональное значению частной производной (скажем, одну десятую или одну сотую частной производной).

На слайде представлен пример, где лосс принимает на вход n параметров. Тогда, чтобы вычислить изменение в i-том параметре, мы вычисляем частную производную

лосса по этому параметру и отнимаем пропорциональное ей число от прежнего значения параметра.

Возвращаясь к аналогии с путешествием в пространстве функций, это можно описать как прогулку по ландшафту, состоящую из холмов (точек с высоким значением лосса) и низин (точек, где лосс низок), причем целью прогулки является попасть в как можно более низкое / глубокое место. Такую идею называют “ландшафтом функции потерь”, а весь метод обучения называется градиентным спуском, потому что частные производные функции по каждой ее переменной называются градиентом и потому что обучение этим методом можно описать как спуск в ландшафте функции потерь.

Для того, чтобы эффективно вычислять градиент функции потерь для многослойных нейросетей, используется прием, известный как “обратное распространение ошибки”, **backpropagation**. Он основан на так называемом “правиле цепи” из математического анализа - это правило позволяет разложить производную композиции функций в произведение производных отдельных функций.

В нейронных сетях каждый слой применяется к выходу предыдущего слоя, поэтому их можно представить в виде композиций нескольких функций. Правило цепи даёт возможность вычислять частные производные функции потерь по каждому параметру нейросети, при этом используя производные по весам каждого слоя в вычислении весов предыдущего слоя. Для этого сначала вычисляется выход последнего слоя нейросети, который затем сравнивается с правильным ответом (он в данном случае известен, потому что выход последнего слоя - это и есть выход нейросети целиком). После чего с использованием дифференцирования вычисляются частные производные лосса относительно весов последнего слоя. Их можно использовать для корректировки весов слоя, так что значения частных производных можно считать “значением ошибки” весов. С помощью “ошибки” в весах последнего слоя вычисляется значение ошибки в весах предпоследнего слоя и таким образом вычисляются производные для всей нейросети. При этом вычисление идет от последнего слоя до первого, в обратную сторону, поэтому этот метод называется “обратным распространением ошибки”. На слайде представлен пример обратного распространения ошибки для двух входов, которые умножаются и результат умножения передается в функцию активации..

Существует множество нейронных сетей, приспособленных для конкретных задач, таких как обработка изображений, последовательностей и т. п. Те, которые вам **в том числе** понадобятся в NLP - это, **например**, рекуррентные нейронные сети и

трансформеры.

Рекуррентные нейронные сети **были изобретены**, чтобы **использовать для предсказаний** последовательности произвольной длины, что **особенно** актуально для текстов.

Такие нейронные сети обрабатывают последовательности токен за токеном и при этом имеют доступ к своим предыдущим состояниям, что позволяет им воспринимать любое слово на основании того, какие слова ранее встречались в этом тексте. Это возможно благодаря тому, что у рекуррентных сетей есть внутреннее состояние, причем, **в отличие от обычных сетей прямого распространения, которые мы только что бегло рассмотрели,** на каждом шаге они имеют доступ к внутреннему состоянию на предыдущем шаге и могут обновлять его для использования во время следующего шага. Это своеобразный аналог памяти - рекуррентные сети помнят, что воспринимали на предыдущем шаг**е**, и могут сохранять информацию **о текущем для последующих**.

У этого базового принципа множество технических деталей - например, как решить, какую информацию из текста нужно запомнить, а какую можно забыть? Три основные архитектуры рекуррентных нейронных сетей**: так называемые "ванильные"** рекуррентные сети (самый простой вид, **они же -- сети Элмана**), вентильные рекуррентные сети **GRU** (один **так называемый "вентиль" -- это такой особый слой с активацией** -- решает, какая информация из предыдущего состояния поступает в нейросеть, а какая нет) и сети типа "долгая краткосрочная память" **LSTM** (**в них** два вентиля - один для запоминания, другой для забывания информации).

5. Предиктивные методы

Как же используются предиктивные методы для построения векторов слов? Ключевая идея в том, чтобы использовать скрытое состояние нейронных сетей для описания свойств слов. Вы помните, что при работе нейронной сети данные последовательно проходят один слой за другим - результат работы предыдущего слоя подается на вход в следующий слой.

Такие промежуточные результаты и называются скрытым состоянием. Оказывается, что скрытое состояние удобно использовать для векторного представления самых разных объектов, и не только в NLP - такой метод применяется, например, для изображений. Так происходит потому, что нейронная сеть при обучении

стремится извлечь из входных данных информацию, которая лучше всего помогает решить проблему. (Мы говорим “стремится”, хотя нейросеть, конечно, не живая и ни к чему не стремится - просто при обучении нейросети ее веса, как правило, получаются такими, что обладают этим свойством).

Значит, для того, чтобы извлечь хорош**ее векторное представление** слова, нужно подобрать такую **задачу машинного обучения**, для решения которой понадобятся самые важные особенности его значения и употребления. Тут нам и пригодится дистрибутивная гипотеза - утверждение о том, что значение слова сводится к тому, в каких контекстах оно употребляется. Если это так, то для того, чтобы представить значение слова в виде вектора, нам нужно заставить нейронную сеть угадывать, в каком контексте употребляется данное слово или какое слово может быть в данном контексте. Тогда скрытое состояние, которое будет поступать, например, в выходной слой нейросети, будет сформировано таким образом, что из него будет легко угадывать, рядом с какими словами употребляется **рассматриваемое** слово - а значит, это скрытое состояние будет являться хорошей векторной репрезентацией.

Для начала разберем метод (а точнее, группу методов) под названием word2vec, который опубликовала в 2013 году группа исследователей во главе с Томашем Миколовым. Для того, чтобы обучить модель word2vec, нам понадобится нейронная сеть из двух слоев - скрытого и выходного, причем на скрытом слое будет отсутствовать функция активации.

(Обычно так не делается, потому что композиция двух линейных функций является линейной функцией, то есть слой, у которого нет функции активации, можно просто объединить со следующим слоем. Но здесь это нужно потому, что со скрытого слоя мы будем снимать скрытое состояние. **Кроме того, п**оскольку размерность у скрытого слоя намного меньше, чем у входного, такую процедуру можно рассматривать как снижение размерности - как и при матричном разложении.)

Для того, чтобы обучить нашу нейросеть на корпус, сначала понадобится как**о**е-нибудь наивное представление текстов в векторном виде для того, чтобы подавать их в нейронную сеть. Давайте сначала выпишем все слова, для которых мы хотим создать представление по методу word2vec (например, это могут быть все слова, которые встретились в корпусе, или все слова, у которых больше определенного числа вхождений в корпус, и тому подобное). Получится словарь размера k. Потом каждому слову присвоим уникальный номер - проще всего отсортировать **их** по алфавиту и пронумеровать по порядку: например, если у нас 50 000 слов, то номер 1 может

достаться слову “абак”, а номер 50 000 - слову “ящер”.

Наивн**ым** векторн**ым представлением** каждого слова, **которое** можно подавать на вход нейросети при обучении, будет являться так называемый one-hot вектор - для i-того слова в словаре размера k он является k-мерным вектором, в котором i-тая компонента равна единице, а все остальные - нулю. Такой вектор не сообщает ничего интересного о свойствах слова - из него можно извлечь только его порядковый номер. Но это лучше, чем напрямую подавать в нейросеть сам номер слова - когда мы загружаем one-hot vector в нейронную сеть, с i-тым словом ассоциируется i-тый нейрон входного слоя, потому что именно в него поступает ненулевой сигнал.

После этого можно обучать нейросеть. Задача, которую **она** должна будет решать, может быть одного из двух видов.

Первая разновидность такая. Дан контекст, в котором употреблено некое слово, но само слово удалено. Необходимо восстановить исходное слово. Этот метод называется continuous bag of words, “**непрерывный мешок слов**” или сокращенно CBOW. Он действительно напоминает метод “мешка слов”, о котором мы говорили ранее: слово здесь представлено через все слова в отрывке текста, в котором оно употреблено.

Вторая разновидность **в каком-то смысле** противоположная: дано слово, и необходимо угадать, допустим, 2 слова слева и 2 слова справа от него. Этот метод называется skip-gram, то есть словесная n-грамма (последовательность из n слов), в которой одно слово пропущено.

В первом случае все one-hot векторы суммируются и на вход нейросети подается мешок слов. **(Кстати, это то же самое, что извлечь из матрицы весов вектор, соответствующий каждой строке, а затем сложить их. Это одно и то же потому, что умножать матрицу на ван-хот вектор - это все равно что доставать строку из матрицы, а матричное умножение ассоциативно относительно сложения.)**

На выход же нейросеть должна отдать один вектор - в идеале one-hot вектор загаданного слова, на практике - вектор, чья i-тая компонента тем больше, чем вероятнее встретить исходное слово в подобном контексте.

Во втором случае на вход нейросети подается one-hot вектор,

а на выход она отдает несколько векторов с оценками вероятности встретить то или иное слово для каждой позиции в контексте слова - эти векторы похожи на вектор вероятностей, который мы получили на выходе из нейросети в первом случае, но их несколько, а не один. **Мы обучаем так модель для каждой пары слов, в котором**

одно центральное, а второе контекстное.

Для обучения нам понадобится перцептрон, он же полносвязная **нейронная** сеть. В нем будет два слоя - скрытый и выходной. В выходном слое должно быть столько же нейронов, сколько слов в словаре - k. Поэтому нам достаточно выбрать размер скрытого слоя. Этот размер будет совпадать с размером векторной репрезентации слова, которая у нас получится в результате. Это число должно быть значительно меньше размера словаря, чтобы произошло снижение размерности, но при этом не слишком мало, чтобы векторное представление могло передавать значимую информацию. Стандартный выбор - **от десятков до небольшого числа сотен, например**, n=300.

Поскольку на скрытом слое нет функции активации, то работа такого многослойного перцептрона может быть выражена с помощью формулы, которая представлена на слайде: вход перцептрона умножается на две матрицы весов, после чего поступает в функцию активации “софтмакс” (формула софтмакса тоже на слайде).

Здесь матрица входных данных умножается на матрицу весов первого слоя, затем - на матрицу весов второго слоя, потом попадает в функцию активации, которая “нормирует” выход нейронной сети, делая его похожим на распределение вероятностей.

Каким образом будут устроены веса этой нейронной сети после обучения? **П**оследний слой должен быть функцией, которая решает задачу CBOW или skip-gram, получая на вход скрытое состояние, полученное из предыдущего слоя. Если бы выходной слой получал исходный one-hot вектор или мешок слов, он мог бы попросту выучить правильный ответ для каждого слова по отдельности; увидев one-hot вектор с i-тым словом, выходной слой бы просто предсказывал те же слова, которые встретились в обучающей выборке. Но в скрытой репрезентации всего 300 элементов.

Для того, чтобы выходной слой мог успешно предсказывать ответ к проблеме, скрытое состояние должно как можно успешнее вместить в триста компонент вектора всю важную информацию о том, какие контексты употребляются с данным словом.

Теперь, чтобы получить векторное представление i-того слова в словаре, нам достаточно умножить i-тый one-hot вектор на матрицу весов W1 - на выход мы получим скрытое состояние нейросети, которой подали на вход данное слово.

Его мы и можем использовать в качестве векторно**го представления** слова. Кстати, умножение i-того one-hot вектора на матрицу эквивалентно извлечению i-той строки в матрице, поэтому можно считать, что матрица W1 - это уже то, что нам

нужно: в ней хранятся векторные репрезентации каждого слова из словаря, причем i-тая строчка - репрезентация i-того слова.

Почему мы при этом не использовали функцию активации на скрытом слое? Заметьте, что без функции активации мы можем заменить две матрицы W1 и W2 на цельную матрицу W размерности k на k. Такая матрица очень похожа на co-occurrence матрицы, которые мы разбирали ранее. Однако описанный метод позволяет разложить матрицу co-occurence на две матрицы меньших размерностей. В работе Леви и Гольдберга (ссылка представлена на слайде) показано, что skip-gram при определенных параметрах эквивалентен разложению PMI матрицы совместного употребления слов. Таким образом, предиктивные методы дистрибутивной семантики и векторные репрезентации через матричное разложение на самом деле похожи.

Понятно, что у такой нейронной сети очень много параметров. Чтобы ее обучение проходило быстрее, исследователи пользуются различными приемами. **Возможно, самые известные из них** - это hierarchical softmax и negative sampling.

Hierarchical softmax работает следующим образом. Вместо того, чтобы хранить отдельный параметр для каждого слова в выходном слое, мы представляем все слова в словаре в виде бинарного дерева. Это значит, что мы создаем дерево, каждый лист которого представляет одно слово из словаря, а все внутренние вершины (все вершины, кроме листьев) имеют ровно две исходящих ветви. Пример такого дерева представлен на слайде: из каждого узла исходят два новых узла, из тех - еще по два новых узла, и это повторяется, пока дерево не завершится концевыми вершинами, то есть листьями, которым соответствуют слова. Теперь, чтобы указать на конкретное слово из словаря, необязательно называть его номер. Вместо этого можно описать путь, который нужно пройти, чтобы попасть из корневой вершины дерева в нужный нам лист. Представьте, что вы помогаете товарищу добраться до определенного места в городе, описывая дорогу по телефону: “Поверни налево, теперь направо, теперь прямо”. Что-то подобное происходит и здесь, только все описание пути состоит только из слов “налево” или “направо”.

Например, в картинке на слайде слову w1 соответствует следующий путь: “налево, налево, налево”, а слову w2 - следующий: “налево, налево, направо”. Когда мы учим нейронную сеть предсказывать слова по контекстам или контексты по словам, то вместо одной классификации на очень большое количество классов делаем несколько бинарных классификаций, соответствующих инструкциям “налево” или “направо”. На практике это достигается за счёт того, что каждому выходному нейрону мы ставим в

соответствие один из внутренних узлов нашего дерева. Во время обучения мы пытаемся добиться того, чтобы нейрон, соответствующий каждой внутренней вершине, выдавал значение либо как можно больше, либо как можно меньше - в зависимости от того, налево или направо нам нужно повернуть после этой внутренней вершины. А чтобы предсказать какое-нибудь слово, мы сначала пытаемся предсказать, налево или направо нам нужно пойти от корневой вершины, затем - налево или направо от вершины, в которую мы попали, и так далее, пока не дойдем до какого-нибудь листа дерева. Количество внутренних вершин в нашем дереве (а значит, количество выходных нейронов и количество параметров) по-прежнему велико. Но когда мы пытаемся заставить нейросеть выдавать большую или меньшую вероятность того, что такое-то слово встречается в таком-то контексте - мы имеем дело с гораздо меньшим числом внутренних вершин, а значит, с меньшим числом параметров, которые нужно изменять в процессе обучения.

Еще один метод называется negative sampling. При его использовании число параметров в нейронной сети не меняется, **и каждый выходной нейрон по-прежнему соответствует некоторому слову в словаре**, но некоторые из параметров замораживаются во время обучения. Каждую итерацию мы выбираем все выходные нейроны, которые должны отдать на выход единицу, а также - случайным образом - некоторое количество нейронов, которые должны отдать ноль. (Это и есть те самые negative samples - отрицательные примеры, то есть примеры слов, которые не появляются **рядом** с данным словом). Для них мы вычисляем градиент, и именно их параметры мы изменяем при обучении. Все остальные нейроны не изменяются. Для каждой итерации набор нейронов, чьи параметры мы обучаем, разный. Поэтому выходит, что все параметры в нейронной сети были обучены. При этом такая процедура обучения быстрее.

На слайде представлена формула функции, которую мы пытаемся максимизировать для обучения word2vec с помощью negative sampling. Она состоит из двух слагаемых.

Первое слагаемое - предсказание вероятности того, что исходное слово и слово-контекст встречаются вместе.

Второе слагаемое - предсказание вероятности того, что исходное слово встретится в тексте в контексте со случайным словом из словаря. Эта вероятность должна быть как можно меньше, поэтому предсказание вероятности взято со знаком минус.

Кроме того, тут изменена функция активации выходного слоя - вместо софтмакса, формулу которого мы уже видели, тут используется сигмоида. Это связано с тем, что для софтмакса нужен результат работы всех нейронов выходного слоя. **Напомним, их число равно размеру словаря.** Сигмоида **же** работает для каждого нейрона независимо, **что гораздо быстрее с точки зрения вычислительной сложности.**

Можно сказать, что мы заменили одну задачу k-классовых классификаций на k задач бинарной классификации.

Одна из проблем классического алгоритма word2vec в том, что он может моделировать значение лишь тех токенов, которые которые хотя бы один раз встретились в обучающей выборке. С помощью word2vec труднее моделировать значение редких слов, тогда как человеческий мозг с этим справляется: если вы, например, встретите слово “пахицефалозавр”, то сразу догадаетесь, что это название какой-то разновидности динозавров, даже если никогда ~~не~~ раньше не слышали это слово. Кроме того, это проблема для языков, в которых слова могут принимать множество различных форм: если не сделать лемматизацию, в результате которой у всех форм слова будет одинаковый вектор, word2vec’у придется отдельно выучивать значение токенов “делаю”, “делаешь”, “делает” и т. п., а если какая-то из этих форм не встретилась в корпусе, то ее значение нельзя будет смоделировать, хотя носитель сразу сможет осмыслить эту форму.

Одним из подходов к решению этой проблемы является модель векторной семантики под названием fasttext. Она основана на skip-gram с negative sampling, но отличается тем, что вектор целевого слова представлен в виде суммы векторов n-грамм, из которых это слово состоит. Это позволяет решить проблему с редкими словами, о значении которых можно догадаться по их внешнему виду: в слове “пахицефалозавр”, например, есть подстрока “завр”, которая служит важной подсказкой. Это же позволяет решить проблему с формами слов: у них, как правило, одинаковые или похожие основы и разные окончания.

Еще одной распространенной моделью дистрибутивной семантики является Glove. Glove была создана для того, чтобы совместить преимущества использования локальных и глобальных статистик в дистрибутивной семантике. Glove отличается от word2vec тем, что при обучении явным образом использует матрицу слова-контексты токенов в обучающей выборке. При этом в матрице используется совместная встречаемость первого порядка - в матрице считаются контексты, в которых два слова

встречаются вместе. **В разделе про разреженные векторы мы уже рассматривали один способ перейти к сходству второго порядка - для этого мы сравнивали векторы-строки в матрице, и сходство векторов-строк означало сходство второго порядка у слов, которые им соответствуют.**

В glove применяется другой подход. Чтобы вычислить совместную встречаемость второго порядка, в glove используется отношение количества co-occurrence различных слов. Например, чтобы оценить сходство слов “лед” и “пар”, мы можем посчитать их совместную встречаемость с релевантными словами вроде “вода” и нерелевантными словами вроде “мода”. Поскольку у слова “вода” и у слова “пар” примерно совпадает набор релевантных слов, число совместных вхождений будет примерно одинаковым для всех контекстов, а значит, отношение co-occurrence для большинства контекстов будет примерно равно единице. (Пример такого вычисления представлен на слайде). Соответственно, эмбеддинги glove основаны на попытке предсказать отношение числа co-occurrence двух данных слов со словами из словаря.

Недостатком всех этих подходов является то, что значение слова, вообще говоря, может быть разным в различных текстах. Слово может обладать несколькими значениями (как слово “брак” может означать свадьбу, а может - дефект производства), и обычно именно из контекста ясно, какое из них подразумевается в конкретном случае. Кроме того, слово всегда по-разному относится к другим словам в предложении - оно может означать того, кто совершает действие, или того, над кем действие совершается, может употреблятся буквально, а может метафорически. Есть множество подобных тонкостей, но они не учитываются в моделях, которые представляют слово в виде стабильного вектора, не меняющегося от одного текста к другому.

Чтобы учесть все эти тонкости, используются так называемые контекстуализованные векторы - векторы, которые пытаются передать не значение “слова вообще”, в отрыве от контекста, а его смысл в конкретном тексте и то, как оно соотносится с другими словами в нём.

Контекстуализированные векторы получаются из языковых моделей, **которым следует посвятить отдельную лекцию**. Там сохраняется идея о том, что для репрезентации слов нужно снимать активацию со скрытого слоя нейронной сети, когда ей на вход подано это слово.

Языковая модель на основе нейронной сети обрабатывает данный **ей** текст целиком, и ее скрытое состояние на интересующем нас слове используется в качестве

контекстуализированного вектора.