

Stepik. Neural networks and NLP. 3. Basic neural network methods for working with texts

- Part 1

3.1 Общий алгоритм работы с текстами с помощью нейросетей

Всем привет! В этом модуле мы наконец-то перейдём к настоящим, современным нейросетям, интересным и глубоким. Однако перед тем, как начать говорить об основных архитектурах, нам нужно построить широкую картину происходящего, чтобы затем добавлять в эту картину больше деталей. Роль этой картины у нас будет выполнять общий алгоритм работы с текстами с помощью нейросетей. Сначала мы очищаем текст от мусора (разметки, оформления) — того, что считаем нерелевантным в нашей задаче. Затем мы разбиваем текст на базовые элементы. Чаще всего нейросети работают на основе отдельных символов или целых токенов. Результат этого этапа — это список объектов, очищенных и нормализованных. Затем мы пересчитываем все уникальные элементы в [корпусе](#) и сопоставляем каждому элементу некоторый идентификатор. Идентификаторы глобальные и, для всех текстов, один и тот же символ (или токен) будет получать один и тот же идентификатор. Можно перед назначением идентификаторов сортировать элементы по алфавиту или по частоте (как удобнее, это не принципиально). Затем мы получаем отображение из символов или токенов в числа и применяем его к предобработанным текстам. В результате мы получим такие же списки, в которых каждый элемент заменён на его глобальный идентификатор. На этом предобработка текста для нейросетей заканчивается и начинаются сами нейросети.

1. Разбить текст на базовые элементы

"Мама мыла раму."

- Символы - ["м", "а", "м", "а", " ", ...]
- Токены - ["мама", "мыла", "раму", "."]

2. Построить словарь и перенумеровать элементы

- {"м": 1, "а": 2, " ": 3, ...}
- {"мама": 1, "мыла": 2, "раму": 3, ".": 4, ...}

3. Преобразовать текст в список номеров

- Символы - [1, 2, 1, 2, 3, ...]
- Токены - [[1, 2, 3, 4]]



Во всех архитектурах самый первый этап — это отображение номеров элементов в вектора. Для этого в нейросети хранится таблица, в которой каждая строчка соответствует какому-то элементу словаря — соответственно, в ней столько же строчек, сколько в нашем словаре, то есть столько же, сколько уникальных элементов мы насчитали во время предобработки текста. Такая таблица называется "таблицей представлений" или "таблицей [эмбеддингов](#)". Русскоязычные исследователи пока что не пришли к единому мнению — какой русскоязычный термин использовать для обозначения [векторных представлений](#), поэтому мы будем использовать либо "векторное представление", либо "эмбеддинг". Так вот преобразование списка идентификаторов элементов в вектора выполняется путём выбора соответствующих строк из этой таблицы. Таким образом, исходный текст преобразуется в матрицу, в которой количество строк равно длине текста, а количество столбцов — размеру эмбеддинга. Отлично! Теперь у нас есть универсальный математический объект — тензор. Его мы будем крутить-вертеть в нашей нейросети, чтобы получить что-нибудь полезное, то есть решить нашу конечную задачу. Как мы уже говорили, этот тензор может принимать форму матрицы. Следующий шаг — учёт локального контекста, то есть мы хотим привнести в вектор для каждого слова информацию о том, какие слова идут перед ним или после него. Это контекст. В результате такого шага мы получим тензор другой формы — например, количество строк у нас может уменьшиться, а размерность эмбеддинга — увеличиться. Локальный контекст мы поучивали, накрутили нелинейных преобразований, всё круто, пора бы и конечную задачу порешать. А для этого мы хотим учитывать не только локальный контекст, но и глобальный. Часто для решения конечной задачи (например, классификации или поиска) очень удобно получить не матрицу, которая зависит от длины текста, а один вектор фиксированной длины.

Дальше с таким вектором можно делать уже что угодно: классифицировать, конкатенировать с другими, и так далее.

Предобученные эмбеддинги для разных русскоязычных моделей -

<https://rusvectores.org/ru/models/>

Length × EmbeddingSize'. On the right side of the slide, there is a logo for 'SAMSUNG Research Russia'. Below the slide, a man in a dark t-shirt with a blue abstract graphic is standing and speaking, holding a pen. Handwritten blue arrows point from the text in sections 1-3 to the table in section 4."/>

1. Разбить текст на базовые элементы

"Мама мыла раму."

- Символы - ["м", "а", "м", "а", " ", ...]
- Токены - ["мама", "мыла", "раму", "."]

2. Построить словарь и перенумеровать элементы

- {"м": 1, "а": 2, " ": 3, ...}
- {"мама": 1, "мыла": 2, "раму": 3, ".": 4, ...}

3. Преобразовать текст в список номеров

- Символы - [1, 2, 1, 2, 3, ...]
- Токены - [1, 2, 3, 4]

4. Для каждого элемента выбрать вектор из таблицы

мама	0.1	...	0.5
...	...		
раму	0.2	...	0.1

Текст → Matrix_{Length × EmbeddingSize}



5. Преобразовать тензор текста

Текст → Matrix_{Length × EmbeddingSize} → Matrix_{NewLength × NewEmbSize}

Цель - получение информации о локальном контексте.



5. Преобразовать тензор текста

Текст \rightarrow $Matrix_{Length \times EmbeddingSize} \rightarrow Matrix_{NewLength \times NewEmbSize}$

Цель - получение информации о локальном контексте.

6. Агрегировать тензор для принятия решения

Цель - получение объекта фиксированной размерности для решения конечной задачи, учит глобального контекста.

$Matrix_{NewLength \times NewEmbSize} \rightarrow Vector_{ResultSize}$



Из комментариев:

Вопрос:

Прямо в тексте ничего не сказано про то, как определяется размер эмбеддинга. Более того, на 0:23 говорится, что количество строк матрицы представлений равно количеству уникальных (!) элементом (токенов, символов) в тексте, а на 1:04 - количество строк равно длине текста(!).

Ответ (Сергей Устьянцев):

размер эмбеддинга либо является гиперпараметром, который мы подбираем во время обучения сети, либо он уже задан, если мы пользуемся готовыми предобученными эмбеддингами. Размер матрицы эмбеддингов равен $VocabSize \times EmbSize$, где $VocabSize$ - размер словаря, $EmbSize$ - размер эмбеддинга. Когда мы составляем тензор для нашего текста, мы из матрицы эмбеддингов берём вектора для всех слов и получаем тензор размером $LenText \times EmbSize$, где $LenText$ - размер текста.

PS [вот здесь](#) есть предобученные эмбеддинги для разных русскоязычных моделей, можно посмотреть типичные размеры эмбеддингов и словарей.

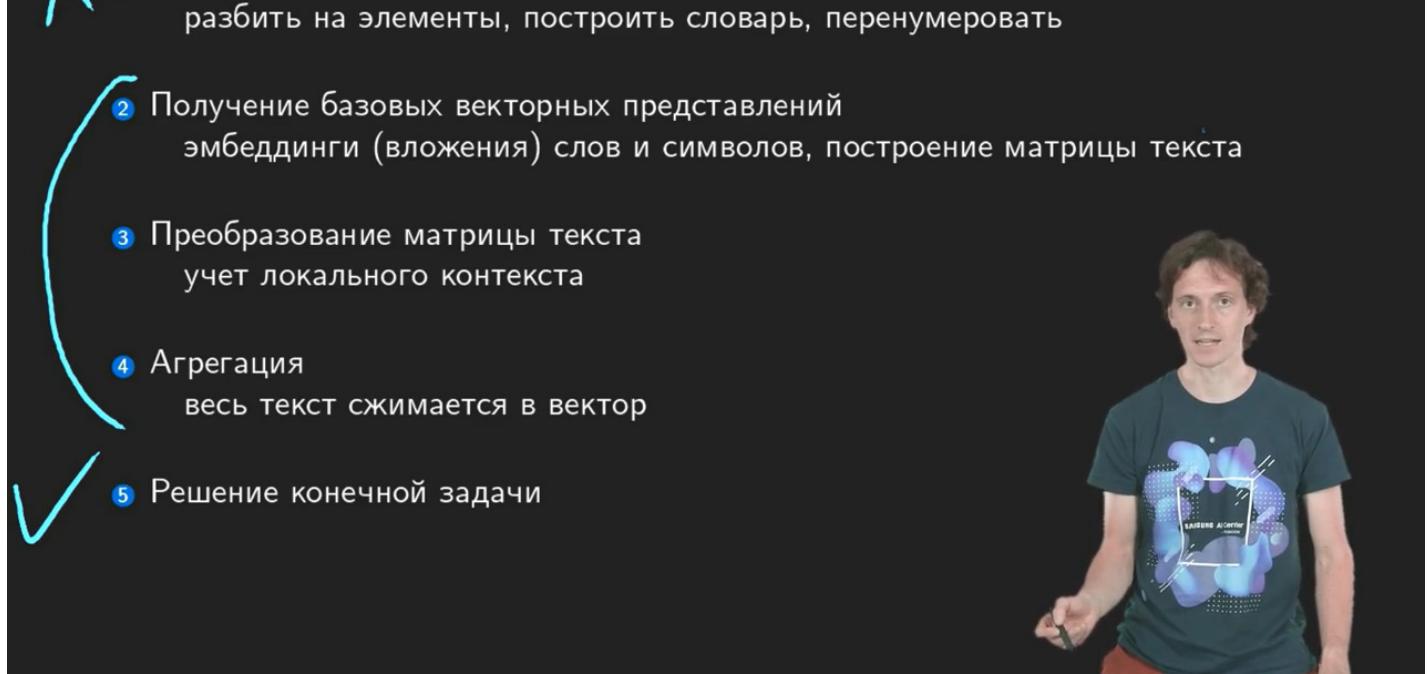
Повторение — мать учения, давайте ещё раз посмотрим на нашу картину без лишних деталей. Сначала мы готовим текст, в результате получаем списки номеров элементов. Затем, каждый элемент преобразуем в вектор. Вектора для всех элементов конкатенируем в таблицу. Затем крутим-вертим эту таблицу, чтобы учесть контексты. Затем, агрегируем, и решаем конечную задачу. При этом шаги 3 и 4 могут повторяться в произвольном порядке, так, чтобы решать

конечную задачу лучше. В этом и заключается искусство работы с нейросетями, а именно — построение правильной архитектуры.

- 1 Подготовка текста
разбить на элементы, построить словарь, перенумеровать
- 2 Получение базовых векторных представлений
эмбеддинги (вложения) слов и символов, построение матрицы текста
- 3 Преобразование матрицы текста
учет локального контекста
- 4 Агрегация
весь текст сжимается в вектор
- 5 Решение конечной задачи



Подготовкой текста мы уже, в принципе, занимались на предыдущем семинаре, больше не будем останавливаться на этом этапе. А также мы уже немного потрогали одну конечную задачу — классификацию на примере длинных текстов и [линейных моделей](#). Мы к ней ещё вернёмся на семинарах. Мы больше всего будем говорить о вот этих трёх типах операций: векторизации, учёте локального контекста и агрегации. Поехали!

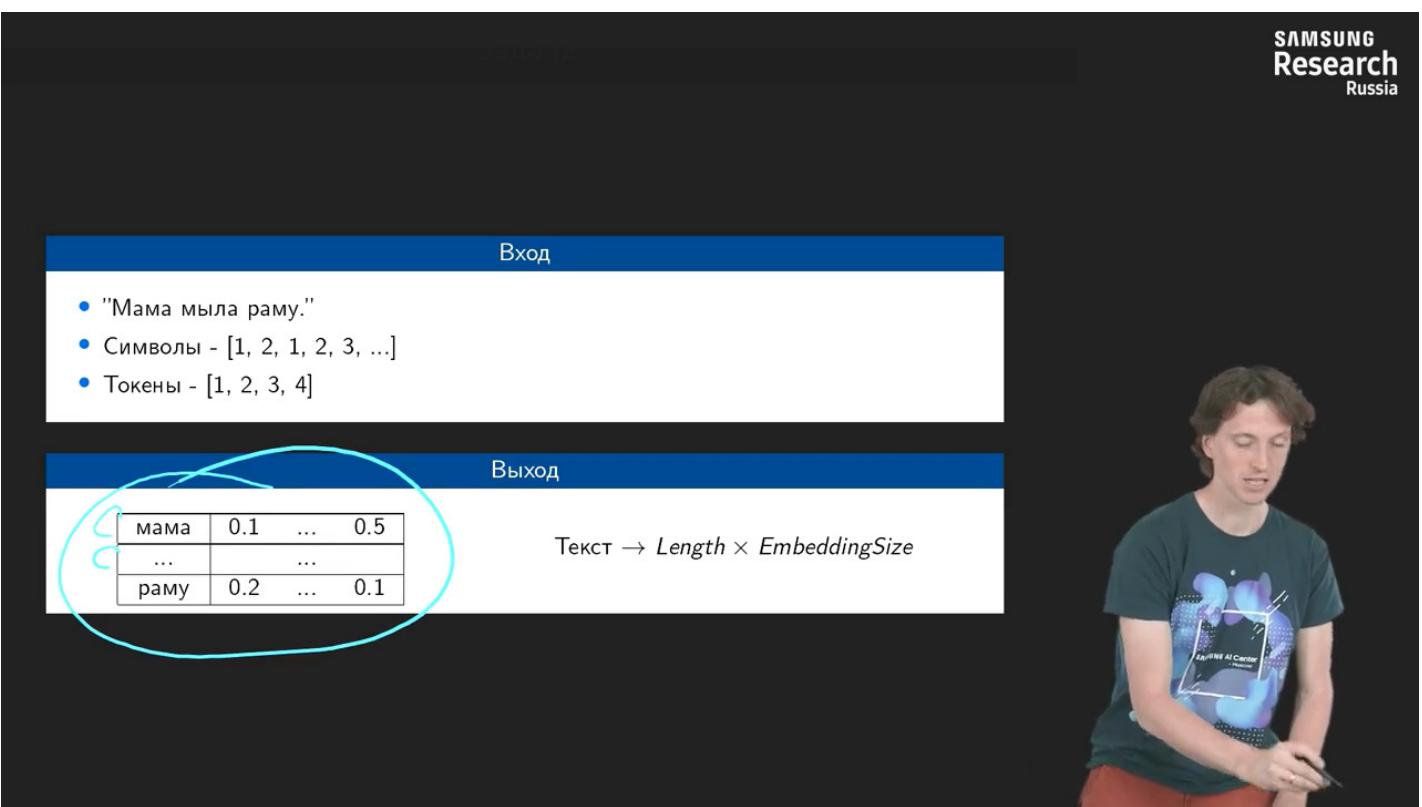


3.2 Дистрибутивная семантика и векторные представления слов

Всем привет! Давайте поговорим о первой операции в нейросетях: векторизации. Существует множество способов получить векторы для элементов текста (символов или токенов), но в этой лекции мы сфокусируемся на [дистрибутивно-семантических моделях](#). Напомню, что текст сначала разбивается на базовые элементы — символы или токены, они нумеруются, заменяются на идентификаторы, а затем по этим идентификаторам мы вытаскиваем из таблицы [эмбеддингов](#) нужные вектора. В этой лекции мы рассмотрим, какие способы построения такой таблицы существуют. Раз уж мы работаем с нейросетями (дифференцируемыми моделями, которые настраиваются [градиентным спуском](#)), то самый естественный вариант — это считать таблицу эмбеддингов параметрами нейросети и обучать вместе с остальной сетью. В этом случае мы будем уверены, что эмбеддинги будут подобраны наилучшим образом для решения конечной задачи. Так как этот способ использует [обучение с учителем](#), у него есть существенный недостаток: нам нужна очень большая размеченная обучающая выборка, в которой будет достаточно разнообразная лексика, чтобы настроить эмбеддинги всех слов и чтобы модель в итоге хорошо работала на новых данных. Помните [распределение Ципфа](#)? У нас очень много редких слов и нам нужно набрать статистику даже для них, если мы хотим чтобы модель хорошо обобщалась. Обучение с учителем — это наиболее стабильная и практическая постановка задачи машинного обучения. Надо стремиться всегда все задачи сводить к обучению с учителем. Для этого нам нужно найти разметку, то есть

некоторую целевую переменную, которую мы будем предсказывать. Оказывается, в текстах такая целевая переменная существует: это следующее слово. Например, если вы увидите фрагмент фразы "шла Саша по...", какое следующее слово вы будете ожидать? Скорее всего, многие из вас будут ожидать слово "шоссе". Задача предсказания следующего слова по известному началу фразы называется "[моделированием языка](#)". Эта тема сейчас очень бурно развивается, придумываются эффективные архитектуры, прогресс впечатляет. Есть и более экономичная постановка задачи моделирования языка. В этом случае, входом нашей модели является не целая фраза, а одно лишь слово — обозначим его "x". Целевых переменных сразу несколько — это вероятности употребления всех остальных слов рядом со словом "x". Такие модели относятся к отрасли дистрибутивной семантики. О них и пойдёт речь в этой лекции. Из названия отрасли "дистрибутивная семантика" мы можем понять, что она занимается изучением смыслов слов. В центре этой области науки лежит дистрибутивная гипотеза, которая заключается в том, что смысл слова, в значительной мере, описывается тем контекстом, в котором слово обычно употребляется.^[1] Контекст можно понимать по-разному. Чаще всего, это просто соседние слова — соседние в предложении, или соседние в целом документе. Ключевой объект, характеризующий контекст — это матрица совместной встречаемости слов. Элементы этой матрицы — это счётчики, то есть натуральные числа, обозначающие количество раз, когда два слова встретились в рамках одного контекста. Что мы можем сказать про эту матрицу? Во-первых, размер словаря для достаточно большого [корпуса](#) текстов имеет порядок сотен тысяч. Матрица встречаемости — квадратная и симметричная, то есть в ней — порядка 10 в десятой степени элементов. Это очень много. Хорошая новость в том, что матрица очень разреженная, то есть почти все её элементы нулевые и их можно хотя бы не хранить. Как мы знаем, есть небольшое количество слов (союзов, предлогов общеупотребимой лексики), которые встречаются намного чаще, чем все остальные слова. Значение матрицы в строках и столбцах соответствующих таким словам, существенно выше, чем в остальных. Это создаёт некоторый дисбаланс. Итак, матрица гигантская, распределение элементов сильно скошенное. Как же с этим всем работать?

[1] You shall know a word by the company it keeps ([Фёрс, Джон Руперт](#) 1957)



- Обучать end-to-end в составе нейросети для конечной задачи классификация, извлечение информации, машинный перевод
- Обучать end-to-end в составе нейросети для моделирования языка предсказание следующего слова по последовательности слов
- Обучать без нейросети в составе дистрибутивно-семантической модели предсказание типичного контекста употребления слова



Дистрибутивная гипотеза

You shall know a word by the company it keeps (Firth, J. R. 1957:11)

- Слово характеризуется своим типичным контекстом
- Контекст - соседние слова
- Совместная встречаемость слов

	мама	любить	молоток	забивать
мама	-	10^5	10^2	10
любить	10^5	-	10	10^3
молоток	10^2	10	-	10^6
забивать	10	10^3	10^6	-



	мама	любить	молоток	забивать
мама	-	10^5	10^2	10
любить	10^5	-	10	10^3
молоток	10^2	10	-	10^6
забивать	10	10^3	10^6	-

- Размер словаря $|Vocabulary| \sim 10^5..10^6$
- Матрица совместной встречаемости - $|Vocabulary| \times |Vocabulary|$
- Разреженность матрицы 70 – 95%
- Значения матрицы для частотных слов всегда выше



Из комментариев:

Вопрос:

Надо стремиться всегда все задачи сводить к обучению с учителем

Скорее наоборот: надо стремиться сводить задачи к обучению без учителя, ибо с учителем слишком дорого.

Ответ (Романа Суворова):

к сожалению обучение без учителя пока что не работает достаточно хорошо и надёжно, чтобы заменить обучение с учителем в прикладных проектах. Если Вы занимаетесь исследованиями, то несомненно, обучение без учителя представляет наибольший интерес. Если же Вы делаете продукт, то скорее всего Вам больше подойдёт менее рисковый путь - через разметку и обучение с учителем. На помощь тут могут прийти краудсорсинг (например, Яндекс.Толока) и transfer learning.

Многие модели дистрибутивной семантики работают по следующему алгоритму: сначала такая матрица всё-таки строится. Благодаря её разреженности, мы можем, в определённых случаях, даже уместить её в память, однако есть итерационные методы, которые не требуют хранения этой матрицы в памяти целиком в течение всего процесса обучения. Затем мы применяем сглаживание, чтобы значения для частотных слов не выбивались так сильно, а затем сжимаем эту матрицу, потому что она слишком большая — мы не хотим с ней работать. Сжать матрицу, чаще всего, означает — разложить её на произведение двух матриц меньшего ранга или факторизовать. Ранг полученных матриц будет не выше вот этой внутренней размерности. Если мы возьмём эти две матрицы и перемножим, мы должны получить что-то, похожее на исходную матрицу. Да, мы потеряли информацию (как же без этого), но, во-первых, потеряем не очень много, а во-вторых приобретём крайне практичный инструмент. Далее, в качестве таблица [эмбеддингов](#) мы можем использовать только одну из полученных матриц, а вторую, например, выкинуть. Сглаживание нам нужно, чтобы привести распределение элементов матрицы X к менее склоненному виду и уменьшить диапазон значений. В результате сглаживания уменьшается негативное влияние частотных слов. Пример сглаживающей функции — поэлементное логарифмирование. В результате, от абсолютных значений счётчиков мы переходим к их порядкам. Другой способ перевесить элементы матрицы — посчитать [точечную взаимную информацию](#) (мы уже говорили о ней в лекции про [TF-IDF](#)). Как же найти эти матрицы меньшей размерности? Так как все числа в матрице "x" — неотрицательные (это просто счётчики), мы можем применить методы из области неотрицательного [матричного разложения](#). Мы не будем подробно останавливаться на этих методах в лекции. Вместо этого мы поговорим побольше о подходах, которое по духу ближе к нейросетям. Некоторые такие подходы основаны на методах безусловной оптимизации, когда мы подбираем значения наших матриц так, чтобы ошибка восстановления была минимальна. Например, мы можем использовать среднеквадратичную ошибку восстановления. Некоторые популярные подходы, которые также близки по духу к нейросетям, основаны на моделировании распределения вероятности встретить некоторое слово в контексте другого слова. Как правило, они также опираются на безусловную оптимизацию и [метод наибольшего правдоподобия](#). Классика, [сингулярное разложение](#) — позволяет представить любую матрицу в виде произведения трёх матриц: двух ортогональных и одной диагональной. При этом, фактически, первая из этих матриц будет соответствовать таблице эмбеддингов, которую мы ищем. Существуют и другие

методы — например, основанные на вероятностных моделях со скрытыми переменными. В этой лекции мы подробнее остановимся на вот этих двух подходах: регрессии и классификации, которые работают через градиентный спуск.

Общий алгоритм

SAMSUNG
Research
Russia

- Построить матрицу совместной встречаемости размера

$$X_{|\text{Vocabulary}| \times |\text{Vocabulary}|}$$

- Сгладить веса в матрице
- Построить аппроксимацию этой матрицы

$$X = W_{|\text{Vocabulary}| \times \text{EmbSize}} \cdot D_{\text{EmbSize} \times |\text{Vocabulary}|}$$



Общий алгоритм

SAMSUNG
Research
Russia

- Построить матрицу совместной встречаемости размера

$$X_{|\text{Vocabulary}| \times |\text{Vocabulary}|}$$

- Сгладить веса в матрице
- Построить аппроксимацию этой матрицы

$$X = W_{|\text{Vocabulary}| \times \text{EmbSize}} \cdot D_{\text{EmbSize} \times |\text{Vocabulary}|}$$



Сглаживание

- Счетчики могут отличаться на несколько порядков
- Частотные слова вносят намного больший вклад
- Логарифмирование $\log(1 + x_{ij})$
- Точечная взаимная информация PPMI

$$X = W_{|V| \times Z} \cdot D_{Z \times |V|}$$

- Неотрицательное матричное разложение (Non-negative matrix factorization)
- Регрессия

$$\sum_{ij} (Smooth(x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Классификация (word2vec, fastText)

$$\sum_{CurWord} \sum_{CtxWord} P(CtxWord | CurWord; W, D) \rightarrow \max_{W,D}$$

- Сингулярное разложение (SVD)

$$Smooth(X_{|V| \times |V|}) = U_{|V| \times Z} \cdot \Sigma_{Z \times Z} \cdot V_{Z \times |V|} \quad W = U \quad D = \Sigma \cdot V$$



$$X = W_{|V| \times Z} \cdot D_{Z \times |V|}$$

- Неотрицательное матричное разложение (Non-negative matrix factorization)
- Регрессия

$$\sum_{ij} (Smooth(x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Классификация (word2vec, fastText)

$$\sum_{CurWord} \sum_{CtxWord} P(CtxWord | CurWord; W, D) \rightarrow \max_{W,D}$$

- Сингулярное разложение (SVD)

$$Smooth(X_{|V| \times |V|}) = U_{|V| \times Z} \cdot \Sigma_{Z \times Z} \cdot V_{Z \times |V|} \quad W = U \quad D = \Sigma \cdot V$$

- Вероятностные модели со скрытыми переменными, латентное размещение Дирихле (LDA)



Из комментариев:

Подробнее об NMF можно узнать в [курсе по вычислительной линейной алгебре](#) от [fast.ai](#) (Lecture 2. Topic Modeling with NMF and SVD).

Модель [GloVe](#), которая расшифровывается как "глобальные вектора", была предложена в 2014 году ребятами из Стэнфорда.^[1] Во-первых, они считают что правильнее опираться на глобальный контекст, то есть учитывать все случаи употребления двух слов в рамках целого документа, а не только в рамках предложения или какого-то узкого окна. Модель, по сути, решает задачу регрессии с квадратичным функционалом качества. Каждое слагаемое в этом функционале — по сути, ошибка восстановления частоты совместной встречаемости двух слов: i -го и j -го. У модели есть пара особенностей. Во-первых, авторы предлагают сглаживать счётчики с помощью логарифма, во-вторых они добавляют веса к слагаемым, чтобы увеличить значимость специальной лексики, то есть слов с меньшей частотой. Для этого они предлагают такую функцию — графически она выглядит следующим образом: она растёт почти линейно до некоторого значения, которое соответствует наибольшей частоте, которая нас вообще интересует, а после этого она абсолютно горизонтальна. Максимум её — единица. У неё есть два параметра: "x_max" — это наибольшая частота, которую мы считаем нормальной, и альфа, которая отвечает за кривизну вот этой части. Чем меньше альфа, тем сильнее поднимается график — тем значимее становятся редкие сочетания.

[1] Pennington, Jeffrey & Socher, Richard & Manning, Christoper. (2014). Glove: Global Vectors for Word Representation. EMNLP. 14. 1532-1543. 10.3115/v1/D14-1162.

- Глобальный контекст слов - матрица совместной встречаемости в документе

$$X = W_{|V| \times Z} \cdot D_{Z \times |V|}$$

- Приближение сглаженной матрицы

$$\sum_{ij} (Smooth(x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Взвешивание слов

$$\sum_{ij} f(x_{ij}) (\ln(1 + x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Весовая функция

$$f(x) = \begin{cases} (x/x_{max})^\alpha & x \leq x_{max}, \\ 1 & otherwise \end{cases}$$



¹Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014.

- Глобальный контекст слов - матрица совместной встречаемости в документе

$$X = W_{|V| \times Z} \cdot D_{Z \times |V|}$$

- Приближение сглаженной матрицы

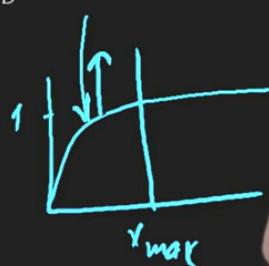
$$\sum_{ij} (Smooth(x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Взвешивание слов

$$\sum_{ij} f(x_{ij}) (\ln(1 + x_{ij}) - w_{i,:} \cdot d_{:,j})^2 \rightarrow \min_{W,D}$$

- Весовая функция

$$f(x) = \begin{cases} (x/x_{max})^\alpha & x \leq x_{max}, \\ 1 & otherwise \end{cases}$$



¹Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014.

Из комментариев:

Вопрос:

Получается, что если пара слов встречается редко (меньше x_max), то модель может себе позволить ошибаться на ней сильнее, поскольку функция $f(x < x_{max}) < 1$. То есть взвешивание помогает сделать акцент на значимом количестве совместных вхождений

во-вторых они добавляют веса к слагаемым, чтобы увеличить значимость специальной лексики, то есть слов с меньшей частотой

на частоту же отдельных слов весовая функция не влияет, поэтому вывод про специальную лексику непонятен

в оригинальной статье также указывают, что добавили весовую функцию для уменьшения шума:

The idea of factorizing the log of the co-occurrence matrix is closely related to LSA and we will use the resulting model as a baseline in our experiments. A main drawback to this model is that it weighs all co-occurrences equally, even those that happen rarely or never. Such rare cooccurrences are noisy and carry less information than the more frequent ones – yet even just the zero entries account for 75–95% of the data in X , depending on the vocabulary size and corpus. We propose a new weighted least squares regression model that addresses these problems.

Ответ (Романа Суворова):

спасибо за комментарий! Это можно понимать двояко. При обучении модель (не только GloVe, а вообще) реагирует не столько на абсолютные значения штрафов, сколько на соотношение штрафов за разные виды ошибок. За счёт x_{\max} (который в статье равен 100) уравниваются все пары слов, которые встретились вместе хотя бы в 100 документах. Таким образом, чтобы у слова появились слагаемые с весом 1, нужно, чтобы оно само встретилось хотя бы в 100 документах.

Примечание из практической задачи:

Зачем нужно сглаживание частот через логарифмирование? Чтобы сделать распределение менее контрастным и сжать диапазон значений, т.к.:

Линейные модели со среднеквадратичным функционалом ошибки не очень подходят для предсказания переменных, имеющих сильно скошенное распределение с большим разбросом.

Следующая модель — [word2vec](#). Она была предложена Томашем Миколавом в 2013 году^[1] и привела к настоящему взрыву интереса к дистрибутивной семантике. В основе подхода — моделирование условного распределения вероятностей соседних слов. Важная особенность и отличие от предыдущей модели — в том что, word2vec работает с локальным контекстом, то есть с окном небольшой длины. Например если ширина равна трём, то мы будем идти по тексту и поочереди выбирать вот такие окна, то есть мы поочереди будем каждое слово ставить в центр окна, и рассматривать его контекст. А также, на каждом шаге, для каждого окна, мы обновляем параметры модели, чтобы повысить правдоподобие того, что мы сейчас наблюдаем. Томаш предложил два варианта модели. Первый называется Skip Gram — он моделирует распределение соседей при условии центрального слова. Второй вариант — наоборот, моделирует распределение центрального слова при условии известных соседей. В

модели для каждого слова хранятся и настраиваются два вектора. Первый (мы будем называть его центральным) мы будем использовать, когда слово находится в центре окна. Второй — когда слово является контекстом, то есть — не в центре. Параметры этой модели настраиваются [градиентным спуском](#). По сути, процесс обучения word2vec идентичен обучению обычной нейросети, когда подаются обучающие примеры (в данном случае окна) один за другим, и после наблюдений небольшой пачки примеров веса модели обновляются.

- [1] Mikolov T. et al. Efficient estimation of word representations in vector space //arXiv preprint arXiv:1301.3781. – 2013. (<https://arxiv.org/abs/1301.3781>)

- Идея - моделируем распределение вероятностей соседних слов
- Локальный контекст - окно небольшой ширины



- Skip Gram

$$\sum_{CenterW_i} P(CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2} | CenterW_i; W, D) \rightarrow \max_{W, D}$$

- CBOW, continuous bag of words

$$\sum_{CenterW_i} P(CenterW_i | CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2}; W, D) \rightarrow \max_{W, D}$$

- Для каждого слова есть два вектора



- Идея - моделируем распределение вероятностей соседних слов

- Локальный контекст - окно небольшой ширины



- Skip Gram

$$\sum_{CenterW_i} P(CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2} | CenterW_i; W, D) \rightarrow \max_{W, D}$$

- CBOW, continuous bag of words

$$\sum_{CenterW_i} P(CenterW_i | CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2}; W, D) \rightarrow \max_{W, D}$$

- Для каждого слова есть два вектора

- "центральный" вектор $w_i \in \mathbb{R}^Z$
- "контекстный" вектор $d_i \in \mathbb{R}^Z$

- Максимизация правдоподобия градиентным спуском

- Аналогично обучению нейросети



Из комментариев:

Примечание от слушателя:

Небольшое замечание по лекции: слова "окно" и "ширина окна" может использоваться для различных понятий:

а) последовательность нескольких идущих подряд слов в тексте и размер этой последовательности,(если мы эти последовательности где-то фиксируем, то они носят

название n-грамм),

6) последовательность слов, находящихся от центрального слова на расстоянии, не превышающем определенное значение, и собственно это максимальное расстояние (окно слева -- центральное слово -- окно справа)
(окно слева + окно справа = контекст)

Уточнение по пункту 6:

Обратите внимание, что "размер окна" в этом курсе равен количеству положительных примеров, которые извлекаются для каждого центрального слова (для каждого положения окна), плюс один. Другими словами, центральное слово стоит в центре окна, а размер окна - расстояние между крайними словами, входящими в окно. В других материалах и библиотеках под размером окна может пониматься "радиус", то есть половина фактической ширины окна. Будьте внимательны!

Вопрос:

В лекции сказано, что word2vec порождает два набора векторов. А какой из них использовать как эмбеддинги слов? Или, возможно, их надо усреднить?

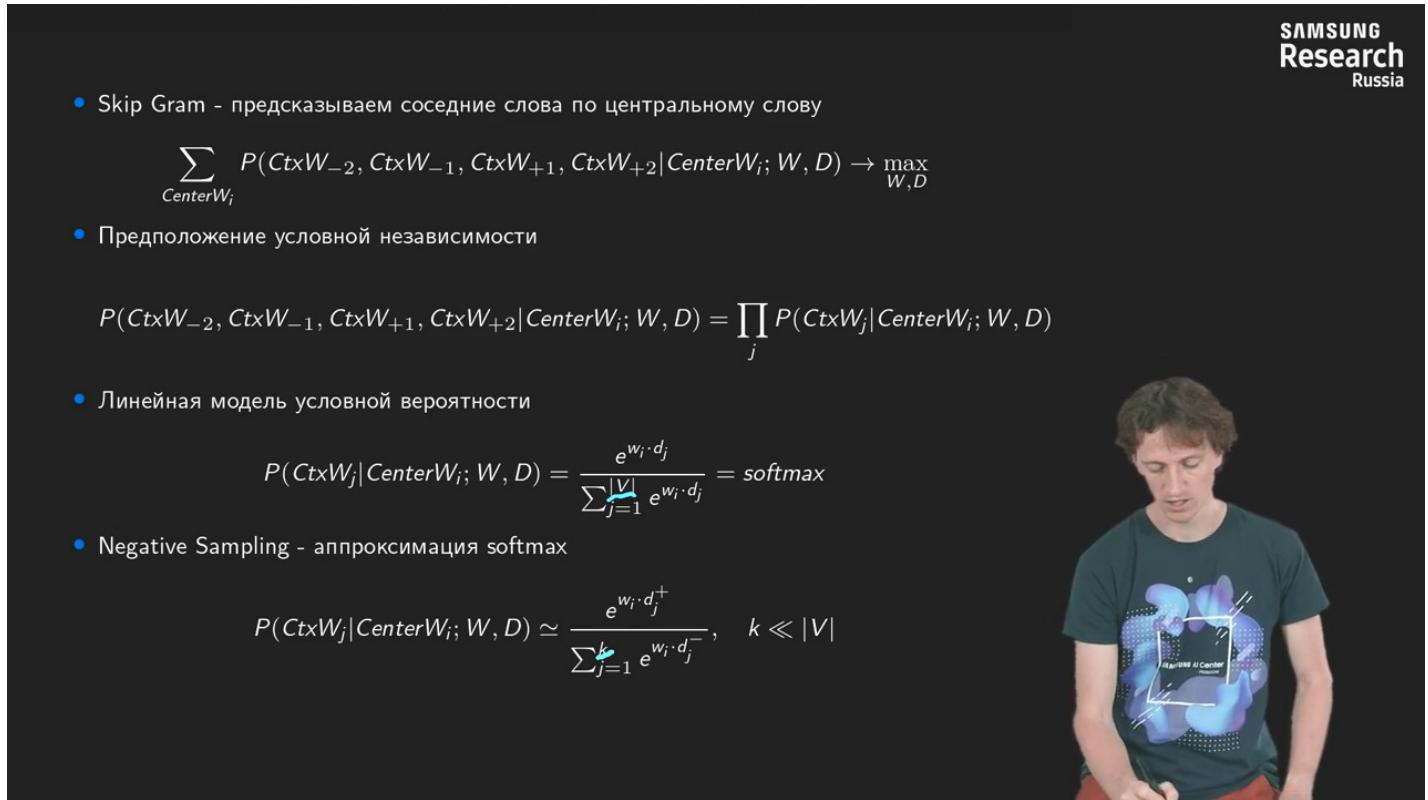
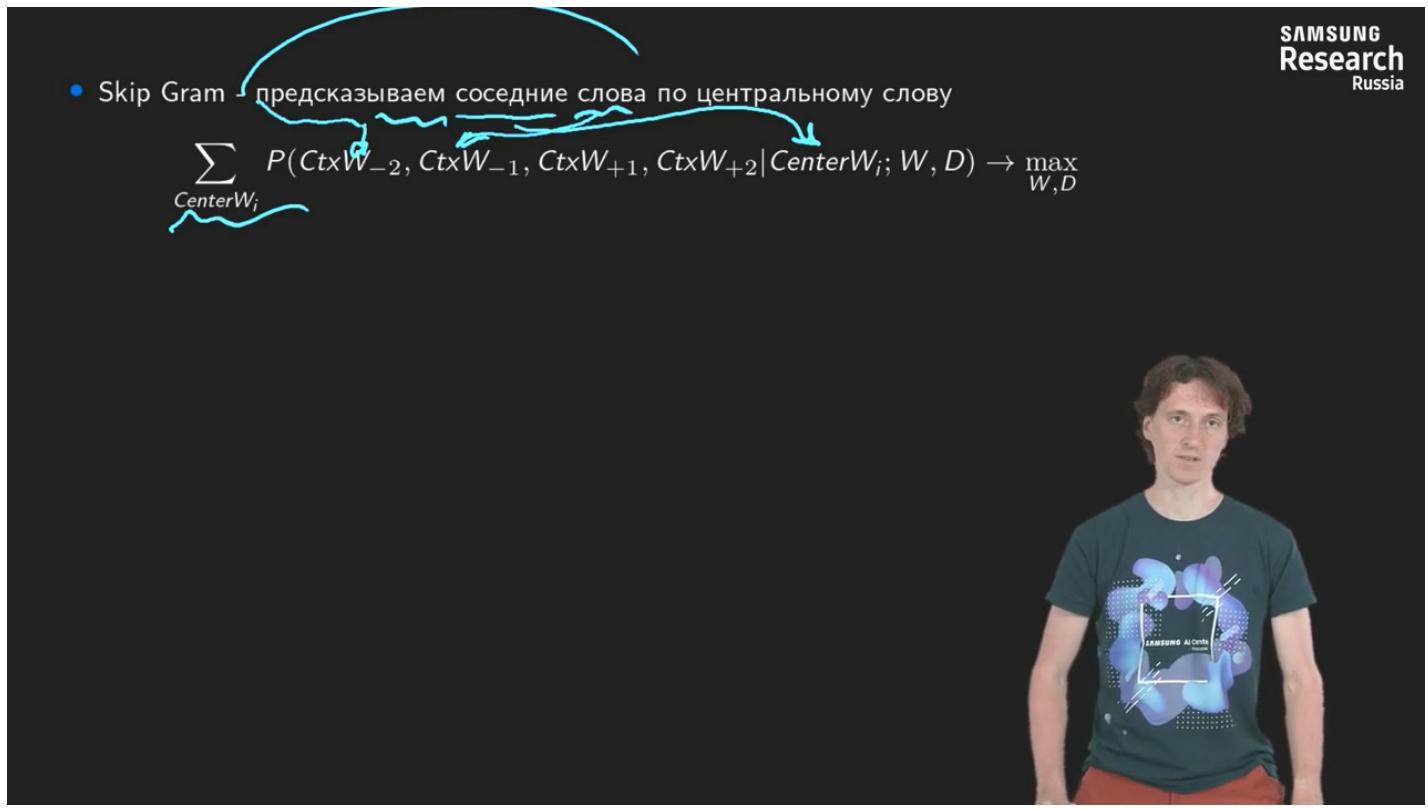
Ответ (от слушателя):

странны, что не было ответа, наверное, не заметили (или ответ дальше в курсе): в принципе, возможны любые варианты. Популярны следующие: вектора центральных слов и полусумма векторов.

Давайте немного подробнее остановимся на варианте Skip Gram. Мы моделируем соседние слова в окне при условии известного центрального слова. В отличие от предыдущей модели, вот эта суммы идёт не по всем уникальным словам, а по всем возможным окнам в [корпусе](#), то есть по всем словоупотреблениям — каждое словоупотребление становится центром окна, мы берём контекст в этом окне, оцениваем его правдоподобие и обновляем веса. Чтобы было удобнее работать с таким распределением, мы предполагаем, что соседние слова условно независимы друг от друга, когда мы уже проанализировали центральное слово. Тогда наше распределение факторизуется, то есть его можно представить в виде произведения более простых распределений. Дальше мы моделируем такие распределения по-отдельности, независимо друг от друга. Моделируем мы их с помощью старого доброго друга — [софтмакса](#). Внутри софтмакса мы используем [скалярное произведение](#) вектора текущего центрального слова с векторами остальных слов. Всё вроде бы хорошо, но в знаменателе у нас стоит сумма по всему словарю, а градиентные шаги нам нужно делать для каждого окна. Это очень дорого с вычислительной точки зрения. Томаш не растерялся и предложил аппроксимировать честный софтмакс более дешёвыми вариантами — один такой способ называется "[отрицательным сэмплированием](#)" (или "negative sampling"). Идея в том, что сумму в знаменателе мы считаем не по всему словарю, а по небольшому числу случайно выбранных слов. Эти слова мы выбираем каждый раз заново. Проблема снижения вычислительной

сложности софтмакса является очень насущной, поэтому были предложены и другие варианты — например, иерархический софтмакс^[1]. Но, пока что, мы не будем их рассматривать.

[1] Hierarchical softmax and negative sampling: short notes worth telling



Из комментариев:

Вопрос:

Почему необходимо две матрицы и следовательно два вектора для каждого слова? Почему нельзя обойтись одним для контекста и центрального слова?

Ответ (от слушателя):

такая схема не будет работать, поскольку вероятность следующего слова (B) находится в контекстном окружении центрального слова (A) прямо пропорциональна значению скалярного произведения векторов $u_A \cdot v_B$, а если и будет равно v , то наибольшая вероятность будет в случае $A == B$, что, очевидно, неправильно (за редким исключением, типа *Canis lupus lupus*). Схема с двумя матрицами позволяет получать для близких слов (синонимов, антонимов, словоформ) близко расположенные вектора, но при этом $u_A \cdot v_B$ уже имеет небольшое значение (а вероятность совместной встречаемости этих слов в одном окне минимальна)

Спустя несколько лет модель [word2vec](#) получила вполне естественное развитие. Важной особенностью [естественных языков](#) является то, что слова могут принимать различные формы, при этом не меняя смысла. Ну, а мы же хотим чтобы наши вектора описывали именно смысл. Другими, словами мы хотим получить инвариантность к словоизменению. Один способ получить такую инвариантность — это нормализовать текст перед обучением модели — например, с помощью стэмминга или лемматизации. Однако первое ненадёжно, а второе сложно — не для всех языков есть хорошие морфологические анализаторы и лемматизаторы. Томаш — парень находчивый. Он и в этот раз не растерялся, а сказал: "Давайте будем учить не на целых словах, а на [N-граммах](#) символов". N-граммы символов встречаются чаще, чем целые слова, при этом, какие-то N-GRAMМЫ могут совпадать с корнями, а какие-то — с окончаниями. Он назвал такую модель: [FastText](#). Работает она, в целом, так же, как и word2vec, но с одним важным отличием. Вот, допустим, мы выбрали какое-то окно и дальше, для центрального слова, мы начинаем извлекать все возможные N-граммы. И делать градиентные шаги, такие же, как мы делали в word2vec, но для каждой N-граммы. При этом мы работаем только с достаточно частотными N-граммами, а редкие игнорируем. Это позволяет использовать FastText для получения векторов слов, которых не было в обучающей выборке. Мы просто берём все возможные N-граммы, которые встречаются в слове, для которого нам нужно получить вектор, берём N-GRAMМЫ, которые мы видели в обучающей выборке, берём их вектора и усредняем. Оказывается, что у векторов слов, полученных способами, которые мы сейчас рассмотрели, есть полезные и забавные свойства. Например, мы можем измерять сходство смыслов слов через [скалярное произведение](#) их векторов. Кстати на сайте "[rusvectores.org](#)" выложено много обученных моделей для русского языка, а также на этом сайте можно поисследовать возможности этих моделей в интерактивном режиме. Например, поискать слова, наиболее похожие на слово-запрос. Забавно, что среди похожих слов для запроса "язык", нет ничего связанного с биологией, только лингвистические термины. Это связано с тем, что word2vec для каждого слова хранит только один вектор, и это может

привести к тому, что некоторые смыслы слова потеряются, а вектор будет описывать только наиболее частотный смысл. А ещё можно выбрать несколько слов и посмотреть графически, как они соотносятся друг с другом. Например, видно, что слово "молоток" ближе к глаголу "забивать", чем к глаголу "любить", например. А глагол "пить" находится недалеко от объектов действия "чай", "кофе". Другое свойство заключается в том, что мы можем складывать и вычитать вектора слов, при этом, как бы, переходя по семантическим связям. Например, если мы вычтем из вектора для слова "женщина" вектор для слова "мужчина", а потом прибавим вектор для слова "дядя" и попробуем поискать ближайшие вектора к полученному, то найдем слова "тётя". Это работает не для всех отношений не для всех слов, но для некоторых частотных — работает.

- Слова могут принимать разные формы - cat/cats, мыло/мыла/мыть
- Инвариантность к форме на уровне слов - нормализация (стемминг, лемматизация)
- Слова -> n-граммы
информация -> { инфо, нфор, форм, ... ация }
- fastText = Word2Vec на уровне целых слов и n-грамм,
если они достаточно частотные

усл
:
:



- Поиск похожих по смыслу слов

$$\text{Similarity}(i, j) = w_i \cdot w_j$$

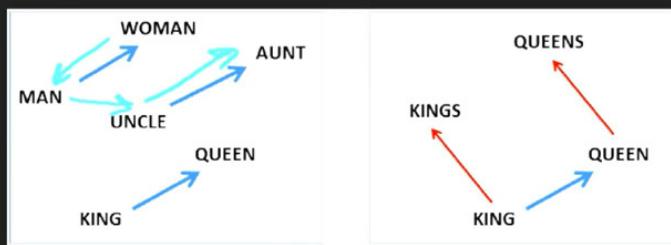
- Предобученные вектора для русского языка и интерактивные примеры:
<https://rusvectores.org/>

язык - грамматика, языке, латынь,
наречие, идиома, синтаксис, лексика ...

чай - кофе, чаять, пить, водка, чашка,
варенье, обед, пиво ...



- Линейная "арифметика смыслов"



Из комментариев:

Вопрос:

Добрый день. А где можно подробнее узнать почему скалярное произведение как схожесть смысла между словами более свойственно для модели fastText? Скалярное произведение это ведь произведение норм векторов на косинус угла между ними? Я провела небольшой эксперимент и оценила среднее значение нормы вектора слова для FastText и обычному W2V

обученному на вики (выборка была в 4000 слов, оценила бутстррапом в ресампл 50 на 100000 итераций), оказалось, что нормы слов отличаются друг от друга совсем незначительно как внутри FastText, так и для W2V, тогда получается, что скалярное произведение почти пропорционально косинусу угла, в таком случае, это говорит о том, что модель FastText в целом более точно располагает вектора слов по смыслу? Либо я что-то упускаю более фундаментальное?

Ответ (от слушателя):

так и есть. Именно поэтому очень часто используют косинус угла между векторами как меру сходства. Но как вы это связали с тем, что модель fastText более точно располагает вектора в пространстве по сравнению с word2vec?

Из последующего практического задания:

Общий алгоритм обучения FastText Skip Gram Negative Sampling выглядит следующим образом:

1. Очистить и токенизировать обучающую коллекцию документов
2. Построить словарь - подсчитать частоты всех целых токенов и N-грамм заданной длины (например, от 3 до 6 символов). При построении словаря раз в заданное число шагов прореживать словарь - удалить из словаря токены, набравшие с предыдущего прореживания меньше всего употреблений (или меньше заданного порога).
3. Проход по корпусу скользящим окном заданной ширины, для каждой позиции окна выполнять шаги 4-7.
4. Для текущего словоупотребления в центре окна выделить его N-граммы, содержащиеся в словаре (то есть только достаточно частотные N-граммы)
5. Вычислить вектор центрального токена, усреднив вектора целого токена (если он есть в словаре) и всех N-грамм, выделенных на шаге 4.
6. Выбрать случайным образом отрицательные слова (сделать negative sampling).
7. Обновить следующие вектора так, чтобы улучшить оценку правдоподобия:
 1. N-грамм, участвовавших в получении вектора центрального токена,
 2. контекстные вектора всех токенов в окне, кроме центрального,
 3. контекстные вектора отрицательных слов.
8. Повторять шаги 3-7 заданное число раз или до сходимости.

Внимание! FastText учитывает само центральное слово как n-грамму, только если оно достаточно частотное.

Из комментариев (к практическому заданию)

Вопрос:

Что значит эта формулировка: Выбрать случайным образом отрицательные слова (сделать negative sampling).

Ответ (Романа Суворова):

у алгоритма есть гиперпараметр - количество отрицательных примеров на каждый положительный. Один положительный пример - это пара <центральное слово и его n-граммы,

слово из контекста>. Отрицательный пример - это пара <центральное слово и его n-граммы, какое-то любое слово>. Отрицательные слова можно выбирать из общего словаря, который был построен на шаге 2. Это можно делать равномерно (`np.random.randint(0, len(vocab))`), а можно с учётом частоты слов в словаре (чтобы чаще выбирать редкие слова).

В этой лекции мы поговорили о векторизации текстов. Она может выполняться на уровне символов, целых токенов или N-грамм символов. Больше внимания мы уделили способам получения векторов слов, поговорили о некоторых методах дистрибутивной семантики, рассмотрели три популярные модели: [GloVe](#), [word2vec](#) и [FastText](#), а также поговорили о том, что можно делать с векторами слов даже без нейросетей — например, искать близайшие слова или делать пусть грубую, но какую-то арифметику смыслов.

SAMSUNG
Research
Russia

- Задача векторизации текстов - на уровне символов, токенов, n-грамм
- Способы получения векторов слов
- Дистрибутивная семантика - выделение "смысла" слова через анализ его типичного контекста
- Способы построения дистрибутивно-семантических моделей - через подсчёт, матричную факторизацию, вероятностные модели
- GloVe, Word2Vec - модели на уровне слов
- FastText - модель на уровне n-грамм
- Поиск похожих по смыслу слов



3.3 Семинар: рецепты еды и Word2Vec на PyTorch

#####

Предисловие из комментария (Романа Суворова):

Для прохождения данного семинара Вам потребуется ноутбук [task2_word_embeddings.ipynb](#).

Чтобы запустить ноутбук с семинара на своем компьютере:

1) Склонируйте [репозиторий курса](#):

```
git clone https://github.com/Samsung-IT-Academy/stepik-dl-nlp.git
```

2) В терминале выполните команду:

```
pip install -r requirements.txt
```

3) Запустите ноутбук:

```
ipython notebook
```

Чтобы запустить ноутбук на [Google Colab](#):

1) Скачайте ноутбук (вкладка Github, затем прописываете адрес репозитория):

The screenshot shows the Google Colab interface. At the top, there's a navigation bar with tabs: Examples, Recent, Google Drive, GitHub, and Upload. The GitHub tab is highlighted. Below the navigation bar is a search bar with placeholder text "Enter a GitHub URL or search by organization or user". To the right of the search bar is a checkbox labeled "Include private repos" and a magnifying glass icon. Underneath the search bar, the URL "https://github.com/Samsung-IT-Academy/stepik-dl-nlp" is entered. Below the URL, there are two dropdown menus: "Repository" set to "Samsung-ITAcademy/stepik-dl-nlp" and "Branch" set to "master". At the bottom of the search area, there's a "Path" input field, a file icon, and a refresh icon. The main content area is currently empty, showing a "NEW PYTHON 3 NOTEBOOK" button and a "CANCEL" button.

2) Запустите ноутбук.

3) Не забудьте выполнить команду `git clone` из первой (закомментированной) ячейки, чтобы выкачать на colab библиотеку `dlnputils`

Ноутбуки также работают и на Kaggle (следуйте комментариям в ячейках ноутбука).

Ссылка на репозиторий со всеми материалами курса и инструкцией по

запуску: <https://github.com/Samsung-IT-Academy/stepik-dl-nlp>

```
#####
```

Всем привет! Сегодня мы реализуем своими собственными руками и обучим одну из самых простых и популярных моделей дистрибутивной семантики [word2vec](#)! Реализовывать модель будем с помощью pytorch. Итак, сначала нам нужно импортировать библиотеки, которые мы будем использовать, к ним относится numpy, matplotlib для визуализации, pytorch, а также наша небольшая библиотечка, которую мы написали специально для этого курса. Самый первый шаг — это загрузить датасет. В этом семинаре мы будем работать с англоязычным датасетом, составленным из рецептов. Нам нужны и из него только сами тексты — разметка нам не нужна. В этой ячейке мы также выполняем разбиение всего датасета на обучающую часть и на

валидационную часть. Для этого мы сначала перемешиваем все предложения, а затем просто берём первые семьдесят процентов как обучение, и остальные тридцать как валидацию. Итого: у нас получается чуть больше 120 тысяч предложений для обучения — достаточно неплохо. Здесь вы можете видеть примеры предложений: первые 10 предложений. Следующий шаг после загрузки датасета — это токенизация, то есть разбиение на базовые лексические элементы. Токенизацию мы выполняем с помощью регулярных выражений. Мы уже использовали этот же алгоритм токенизации в первом семинаре про логистическую регрессию и классификацию новостных текстов. На экране вы видите те же самые 10 предложений, только после токенизации. В результате токенизации мы отбрасываем все очень короткие токены — предположительно, они не несут основного смысла. Далее, используя только обучающую подвыборку, мы строим словарь. При построении словаря мы хотим учитывать только значимые слова. Для этого мы отбрасываем слишком редкие слова, для которых мы просто не можем накопить статистику и не можем выучить их смысл — если слово встречается только один-два раза в [корпусе](#), мы всё равно не сможем для него выучить какой-то "вектор смысла". А также мы отбрасываем самые часто встречающиеся токены — предположительно, это союзы, знаки препинания, цифры... Важный момент, про который нужно упомянуть — это добавление фиктивного токена в словарь: "токена выравнивания", так называемого. Он получает номер "ноль" и он будет использоваться для того, чтобы у нас появилась возможность объединить предложения разной длины в прямоугольный тензор. Это — часто применяемый трюк при обработке текстов с помощью нейросетей. Как мы видим, всего в нашем датасете примерно две тысячи уникальных токенов. Нейросети и компьютеры не умеют работать с текстами, как это делает человек — они работают только с числами. Нам нужно применить наш построенный словарь для того, чтобы отобразить токены (как фрагменты текста) в числа (в номера этих токенов в словаре). На экране вы видите те же самые 10 предложений, но вместо токенов здесь — их номера. Давайте посмотрим, какой длины предложения в нашем датасете встречаются. Видим, что большая часть предложений укладывается в 20 токенов. И, наконец, мы создаём специальные объекты — "датасеты" (Dataset). Они будут использоваться для того, чтобы, непосредственно, подавать данные — подавать фрагменты обучающей выборки в модель в процессе обучения. Мы хотим использовать pytorch и обучать модель, возможно, на видеокарте (особенно, если наша модель большая), а значит, нам нужно предусмотреть всё для эффективной пакетной обработки. Модель должна уметь обрабатывать сразу много предложений разной длины — "за раз". Но, с другой стороны, модели могут работать только с [тензорами](#), а тензор — это такая "прямоугольная" конструкция, она не может иметь "неровный край". Нам нужно сделать, так чтобы все предложения имели одинаковую длину. Самое простое — это выровнять длину этих предложений с помощью фиктивных слов. Именно для этого мы вводили в словарь слово "" с нулевым идентификатором. Для того, чтобы подготавливать данные в нужном нам видео, мы используем объект PaddedSequenceDataset. Этот объект описан в нашей маленькой библиотеке, которую мы сделали специально для этого курса. Он умеет делать только две простые вещи: во-первых он может говорить свою длину (то есть, сколько предложений в нём есть, сколько текстов), а также он умеет возвращать

предложение по номеру. При этом, если предложение короче некоторой заданной длины, то он добавляет нули в конец этого предложения. Или же, наоборот, если предложение длиннее установленного порога, то он его обрезает, то есть берёт только префикс предложения. Эта функция возвращает пары — а именно "текст" и "какая-то метка", которую по этому тексту нужно предсказывать. В данном семинаре мы не будем использовать метки — чуть позже я объясню, почему. Тем не менее, для того, чтобы этот объект можно было использовать в разных задачах (например, в задачах классификации), мы возвращаем и метки тоже. На экране вы видите один обучающий пример — так, как его подготовил наш Dataset. Мы видим, что это предложение состоит только из двух значимых токенов. Все остальные позиции заполняются нулями. С помощью гистограммы, которую мы построили чуть раньше, мы имели возможность выбрать оптимальную длину предложения так, чтобы и лишних вычислений не делать слишком много, и, при этом, уметь обрабатывать практически все предложения из нашего датасета. Для нашего датасета, как мне кажется, неплохо подходит порог "20". Я скажу ещё пару слов о том, зачем нам нужны эти паддинги (добавлять нули, и так далее). Видеокарты (это основные вычислители, которые сейчас используются для обучения нейросетей) умеют делать хорошо достаточно простые операции, и, при этом, они умеют делать эти простые операции параллельно. Допустим, если нам нужно перемножить две матрицы, то в видеокарте запускается четыре тысячи потоков, каждый поток из этих четырёх тысяч, по сути, делает только 1-2 перемножения, и всё — и возвращает результат. При этом мы получаем очень большое ускорение. Но проблема в том, что данные исходные у нас не лежат на видеокарте, они лежат у нас во внешней памяти на жёстком диске. Сначала мы их загружаем с жёсткого диска в оперативную память, как-то предобрабатываем (например — так, как мы это сейчас обсудили), и только потом копируем в память видеокарты. Эта процедура копирования занимает значительное время. Это гораздо дороже, чем скопировать память внутри видеокарты или внутри оперативной памяти. И поэтому мы хотим минимизировать количество переносов данных из оперативной памяти в память видеокарты, и наоборот. Для этого нам нужно объединять наши обучающие примеры в батчи (в группы), и мы описываем нашу модель, наш процесс обучения, таким образом, чтобы он поддерживал обработку нескольких обучающих примеров за раз. Естественно, чем больше данных за раз мы загрузили в видеокарту, тем более эффективно мы можем использовать возможности параллельных вычислений. Все потоки будут хорошо загружены.

Word2Vec

```
In [ ]: %load_ext autoreload
%autoreload 2

import random
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import torch
from torch import nn
from torch.nn import functional as F

import dlnlputils
from dlnlputils.data import tokenize_corpus, build_vocabulary, texts_to_token_ids, \
    PaddedSequenceDataset, Embeddings
from dlnlputils.pipeline import train_eval_loop, predict_with_model, init_random_seed
from dlnlputils.visualization import plot_vectors

init_random_seed()
```

Загрузка данных и подготовка корпуса

```
In [ ]: full_dataset = list(pd.read_csv('./datasets/nyt-ingredients-snapshot-2015.csv')['input'].dropna())
random.shuffle(full_dataset)

TRAIN_VAL_SPLIT = int(len(full_dataset) * 0.7)
```



Загрузка данных и подготовка корпуса

```
In [ ]: full_dataset = list(pd.read_csv('./datasets/nyt-ingredients-snapshot-2015.csv')['input'].dropna())
random.shuffle(full_dataset)

TRAIN_VAL_SPLIT = int(len(full_dataset) * 0.7)
train_source = full_dataset[:TRAIN_VAL_SPLIT]
test_source = full_dataset[TRAIN_VAL_SPLIT:]
print("Обучающая выборка", len(train_source))
print("Тестовая выборка", len(test_source))
print()
print('\n'.join(train_source[:10]))
```

```
In [ ]: # токенизируем
train_tokenized = tokenize_corpus(train_source)
test_tokenized = tokenize_corpus(test_source)
print('\n'.join(' '.join(sent) for sent in train_tokenized[:10]))
```

```
In [ ]: # строим словарь
vocabulary, word_doc_freq = build_vocabulary(train_tokenized, max_doc_freq=0.9, min_count=5, pad_word=True)
print("Размер словаря", len(vocabulary))
print(list(vocabulary.items())[:10])
```

```
In [ ]: # отображаем в номера токенов
train_token_ids = texts_to_token_ids(train_tokenized, vocabulary)
test_token_ids = texts_to_token_ids(test_tokenized, vocabulary)

print('\n'.join(' '.join(str(t) for t in sent)
                for sent in train_token_ids[:10]))
```



Загрузка данных и подготовка корпуса

```
In [2]: full_dataset = list(pd.read_csv('./datasets/nyt-ingredients-snapshot-2015.csv')['input'].dropna())
random.shuffle(full_dataset)

TRAIN_VAL_SPLIT = int(len(full_dataset) * 0.7)
train_source = full_dataset[:TRAIN_VAL_SPLIT]
test_source = full_dataset[TRAIN_VAL_SPLIT:]
print("Обучающая выборка", len(train_source))
print("Тестовая выборка", len(test_source))
print()
print('\n'.join(train_source[:10]))

Обучающая выборка 125344
Тестовая выборка 53719

1/4 cup sour cream
10 ounces swordfish, red snapper or other firm-fleshed fish
1 tablespoon minced basil leaves
Handful fresh parsley, finely minced
4 ounces lard or butter, plus more for brushing tops
4 to 5 green cardamom pods
1 stick ( 1/4 pound) unsalted butter, softened
1/4 teaspoon red pepper flakes, preferably Turkish or Aleppo (see note), more to taste
1 tablespoon fresh lemon juice
1/4 cup scallions, thinly sliced
```



```
In [ ]: # токенизируем
train_tokenized = tokenize_corpus(train_source)
test_tokenized = tokenize_corpus(test_source)
print('\n'.join(' '.join(sent) for sent in train_tokenized[:10]))
```

```
import collections
import re

import numpy as np

TOKEN_RE = re.compile(r'[\w\d]+')

def tokenize_text_simple_regex(txt, min_token_size=4):
    txt = txt.lower()
    all_tokens = TOKEN_RE.findall(txt)
    return [token for token in all_tokens if len(token) >= min_token_size]

def character_tokenize(txt):
    return list(txt)

def tokenize_corpus(texts, tokenizer=tokenize_text_simple_regex, **tokenizer_kwargs):
    return [tokenizer(text, **tokenizer_kwargs) for text in texts]

def add_fake_token(word2id, token='<PAD>'):
    word2id_new = {token: i + 1 for token, i in word2id.items()}
    word2id_new[token] = 0
    return word2id_new

def texts_to_token_ids(tokenized_texts, word2id):
    return [[word2id[token] for token in text if token in word2id]
            for text in tokenized_texts]
```



Обучающая выборка 125344
Тестовая выборка 53719

```
1/4 cup sour cream
10 ounces swordfish, red snapper or other firm-fleshed fish
1 tablespoon minced basil leaves
Handful fresh parsley, finely minced
4 ounces lard or butter, plus more for brushing tops
4 to 5 green cardamom pods
1 stick ( 1/4 pound) unsalted butter, softened
1/4 teaspoon red pepper flakes, preferably Turkish or Aleppo (see note), more to taste
1 tablespoon fresh lemon juice
1/4 cup scallions, thinly sliced
```

```
In [3]: # токенизируем
train_tokenized = tokenize_corpus(train_source)
test_tokenized = tokenize_corpus(test_source)
print('\n'.join(' '.join(sent) for sent in train_tokenized[:10]))
```

```
sour cream
ounces swordfish snapper other firm fleshed fish
tablespoon minced basil leaves
handful fresh parsley finely minced
ounces lard butter plus more brushing tops
green cardamom pods
stick pound unsalted butter softened
teaspoon pepper flakes preferably turkish aleppo note more taste
tablespoon fresh lemon juice
scallions thinly sliced
```

```
In [ ]: # строим словарь
vocabulary, word_doc_freq = build_vocabulary(train_tokenized, max_doc_freq=0.9, min_count=5, pad_word='<PAD>')
print("Размер словаря", len(vocabulary))
```



```
stick pound unsalted butter softened
teaspoon pepper flakes preferably turkish aleppo note more taste
tablespoon fresh lemon juice
scallions thinly sliced
```

```
In [4]: # строим словарь
vocabulary, word_doc_freq = build_vocabulary(train_tokenized, max_doc_freq=0.9, min_count=5, pad_word='<PAD>')
print("Размер словаря", len(vocabulary))
print(list(vocabulary.items())[:10])
```

```
Размер словаря 2267
[('<PAD>', 0), ('tablespoons', 1), ('teaspoon', 2), ('chopped', 3), ('salt', 4), ('pepper', 5), ('cups', 6), ('ground', 7), ('fresh', 8), ('tablespoon', 9)]
```

```
In [5]: # отображаем в номера токенов
train_token_ids = texts_to_token_ids(train_tokenized, vocabulary)
test_token_ids = texts_to_token_ids(test_tokenized, vocabulary)

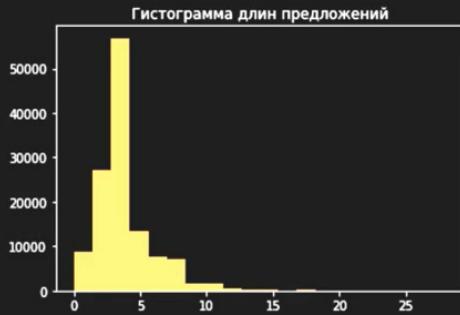
print('\n'.join(' '.join(str(t) for t in sent)
               for sent in train_token_ids[:10]))
```

```
222 52
22 878 574 127 246 707 181
9 19 88 33
517 8 43 15 19
22 586 20 45 47 649 648
59 329 535
200 12 50 20 266
2 5 140 78 1208 735 153 47 10
9 8 31 25
98 65 27
```

```
In [ ]: plt.hist([len(s) for s in train_token_ids], bins=20);
```



```
In [6]: plt.hist([len(s) for s in train_token_ids], bins=20);
plt.title('Гистограмма длин предложений');
```



```
In [7]: MAX_SENTENCE_LEN = 20
train_dataset = PaddedSequenceDataset(train_token_ids,
                                       np.zeros(len(train_token_ids)),
                                       out_len=MAX_SENTENCE_LEN)
test_dataset = PaddedSequenceDataset(test_token_ids,
                                      np.zeros(len(test_token_ids)),
                                      out_len=MAX_SENTENCE_LEN)
print(train_dataset[0])
(tensor([222, 52, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], tensor(0))
```

Алгоритм обучения - Skip Gram Negative Sampling

```
import torch
from torch.utils.data import Dataset

class PaddedSequenceDataset(Dataset):
    def __init__(self, texts, targets, out_len=100, pad_value=0):
        self.texts = texts
        self.targets = targets
        self.out_len = out_len
        self.pad_value = pad_value

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, item):
        txt = self.texts[item]
        if len(txt) < self.out_len:
            txt = list(txt) + [self.pad_value] * (self.out_len - len(txt))
        else:
            txt = txt[:self.out_len]

        txt = torch.tensor(txt, dtype=torch.long)
        target = torch.tensor(self.targets[item], dtype=torch.long)

        return txt, target
```

Из комментариев:

Вопрос:

Для нашего датасета, как мне кажется, неплохо подходит порог "20", а по гистограмме больше на 12 похоже. Что будет, если 20 заменить на 12? Обобщенно?

Ответ (Романа Суворова):

20 - потому что почти все предложения короче. Такую гистограмму я строю всегда, когда приступаю к новому датасету и пытаюсь понять, какой длины последовательности делать. Конкретно в этой задаче это не очень большое значение имеет.

Обобщённая логика для выбора этого значения следующая: мы хотим выбрать такое наименьшее число, чтобы подавляющее большинство предложений из датасета оказалось не длиннее этого числа. Чем меньше это число - тем меньше памяти батч будет занимать и тем больше он будет заполнен значимыми элементами (меньше придётся паддинг использовать), тогда появится возможность увеличить размер батча (это часто приводит к более стабильному процессу обучения). Но с другой стороны, если мы выберем это число совсем маленьким, мы просто не будем решать поставленную задачу. Наоборот, если мы выберем это число как максимальную длину предложения в датасете, то 99% времени нам придётся забивать половину входного тензора нулями (паддить), так как длинных предложений существенно меньше, оно того чаще всего не стоит (вычислений существенно больше, чтобы получить всего лишь пару дополнительных примеров).

Вопрос:

А также мы отбрасываем самые часто встречающиеся токены – предположительно, это союзы, знаки препинания, цифры...

Стоит ли отбрасывать такие токены? Сами по себе они, конечно, смысла не несут, но кажется, что создают тот локальный контекст, благодаря которому учится word2vec: употребление терминов с одними и теми же союзами, вводными словами и цифрами, вероятно, говорит об их похожести. Также хотелось бы услышать мнение по поводу стемминга и лемматизации. Применять ли эти методы при сборе выборки для обучения word2vec? кажется, что окончания и форма слова также в разных контекстах разные

Ответ (Романа Суворова):

Сначала второй вопрос. Если Вы учите Word2Vec или GloVe, то лучше делать POS-теггинг и лемматизацию, это достаточно сильно влияет на вектора омонимичных слов (мыла - мыть или мыло?). Все статические Word2Vec модели, выложенные на <https://rusvectores.org/ru/models/>, обучены с POS-теггингом и лемматизацией. Тогда в словаре модели не будет записи "мыла", но будут две "мыть_VERB" и "мыло_NOUN". Если Вы учите FastText, то определять части речи и лемматизировать не нужно, это не поможет сильно.

По поводу частотных токенов - чаще всего их выбрасывают, потому что они настолько частотные, что встречаются почти со всеми словами и поэтому слабо характеризуют смысл. К тому же в этих моделях в рамках скользящего окна мы по сути имеем "мешок слов", то есть потеряна информация о порядке слов - а союзы и знаки препинания теряют смысл в отрыве от информации об их позиции (после какого именно слова они шли).

Однако есть специализированные модели, где пунктуацию и союзы учитывают, а потом используют эти эмбеддинги для анализа дискурса. Попробую найти эту статью.

Да, забыл сказать, что с частотными словами связана ещё так называемая hubness problem, <https://arxiv.org/abs/1605.02276>, которая приводит к тому, что маргинальная частота слова начинает сильнее влиять на сходство слова с другими словами, чем совместное их распределение

С подготовкой данных — всё, переходим к описанию модели. Давайте я напомню основную идею, лежащую в основе модели SkipGram.^[1] В модели SkipGram для каждого слова у нас есть два вектора — первый вектор используется, когда слово находится в центре скользящего окна, и второй вектор используется, когда это слово описывает контекст. Для того, чтобы хранить эти вектора, мы создаём две матрицы. Каждая матрица имеет размер по количеству слов в словаре и по количеству элементов в векторе. То есть — размер словаря на размер нашего эмбеддинга. Настраивать значения этих матриц, то есть обучать модель, мы будем по методу максимального правдоподобия. Изначальная интуиция описывается вот этой формулой — то есть мы моделируем условное распределение соседних слов в некотором окне (небольшом, как правило это "5") при условии того, что мы проанализировали центральное слово и у нас есть какие-то параметры модели. Это распределение в общем виде посчитать очень сложно. Мы предполагаем, что оно раскладывается на произведения более простых распределений. Каждое такое "более простое" распределение нам говорит, насколько вероятно можно встретить какое-то контекстное слово рядом с центральным словом. То есть, для слов, которые типично встречаются рядом, наша модель должна возвращать вероятность, близкую к единице, а для слов, которые никогда не встречаются рядом — вероятность, близкую к нулю. В этом распределении у нас есть две случайные величины. Обе они — категориальные (то есть, дискретные и принимающие значение из некоторого фиксированного конечного набора). Общепринятый способ моделировать категориальные распределения — это софтмакс. Софтмакс — это функция такого вида, которой на вход подаётся вектор вещественных чисел из произвольного диапазона, и она переводит этот вектор в вектор, описывающий распределение вероятности категориальной величины. В контексте word2vec мы подаём в софтмакс оценки сходства слов. Сходство слов мы будем моделировать как скалярное произведение векторов этих слов. Причём мы будем, для центрального слова, брать векторы из первой матрицы (из матрицы w), а для контекстного слова мы будем брать вектор из второй матрицы (матрицы d). Всё, вроде бы, хорошо, но в этой формуле есть проблемы. Проблема заключается в том, что нам необходимо брать сумму в знаменателе по очень большому числу слагаемых. Это очень вычислительно неэффективно. Авторы модели предлагают заменить сумму по всему словарю суммой по небольшому количеству случайно выбранных слов и, каждый раз, когда нам нужно посчитать аппроксимацию к этому софтмаксу, мы выбираем новые случайные слова — это, как раз, называется "negative sampling". То есть, SkipGram (это "предсказываем соседние слова по центральному слову") — это вот эта часть формул, а

"negative sampling" — это вот эта часть формул. Короче говоря, задача обучения word2vec сводится к обучению классификатора, который, имея два идентификатора слова (два номера токена) предсказывает — могут они встретиться вместе, или не могут. Всё, на самом деле — проще, чем может показаться, когда вы посмотрите на эти формулы. Давайте перейдём к программированию нашей модели и процессу её обучения. Сначала определим вспомогательную функцию, которая нам будет полезна. Наша модель принимает на вход целое предложение и должна оценивать вероятности встречаемости двух слов внутри небольшого окна. Другими словами, нам нужно игнорировать факты совместной встречаемости слов в одном предложении, но — далеко друг от друга, за пределами окна. Для того, чтобы игнорировать такие случаи, введём функцию, которая создаёт маску. В данном случае, маска — это квадратная матрица, сторона матрицы равна длине предложения, с которым мы работаем, все элементы в этой матрице — нулевые, за исключением двух полосок вдоль главной диагонали. Эти полоски заполнены единичками, и ширина этих полосок равна половине ширины окна. Таким образом мы задаём множество пар позиций токенов в предложении, для которых мы учитываем факты совместной встречаемости. Другими словами, строки и столбцы этой матрицы соответствуют некоторым позициям токенов в предложении, и значение этой матрицы задаёт — учитываем ли мы тот факт, что два токена, стоящие на этих позициях, встречается в одном предложении — учитываем ли мы этот факт как положительный пример при обучении, или игнорируем этот факт. Например, для позиций 10 и 2 мы игнорируем, потому что эти две позиции стоят далеко. А если позиции отличаются всего лишь на единичку — например 3 и 2, то мы уже учитываем, потому что тогда эти 2 токена всегда входят в наше окно. Эта матрица нам потребуется чуть позже для того, чтобы реализовать эффективную пакетную обработку на видеокарте. Итак, начнём описывать нашу модель. Как вы помните, в модели есть две прямоугольные матрицы размерности "количество слов в словаре" на "размер эмбеддинга". В pytorch есть модуль, который реализует функции выборки из этой таблицы по номеру токена — этот модуль называется `embedding`, и он лежит в пакете `nn`. Мы создаём два экземпляра этого класса для того, чтобы хранить таблицу центральных векторов и контекстных векторов. Очень важный момент, который не всегда описывается в руководствах и статьях про обучение эмбеддингов: очень важно проинициализировать эти таблицы. По умолчанию, `nn.embedding` заполняет значения в своей матрице нормальным шумом. Для обучения word2vec эта инициализация совершенно не подходит. Правильный способ инициализации весов для обучения word2vec — это равномерный шум, причём диапазон значений этого шума связан с размерностью эмбеддингов. Такая инициализация нужна для того, чтобы длина вектора, начальная, примерно равнялась единице. Также, как мы помним, у нас есть фиктивное слово — фиктивный токен "pad". Мы его используем для того, чтобы дополнять предложения до фиксированной длины. Мы говорим слою `nn embedding`, что токен с индексом "0" — фиктивный и что для него не нужно учить вектор, для него вектор всегда будет нулевым. Он не будет меняться в процессе обучения. Проделываем всё то же самое для матрицы контекстных

векторов. И создаём, один раз, матрицу масок с двумя полосами ненулевых элементов — как мы чуть раньше сказали. Это — то, что касается инициализации.

[1] Word2vec Made Easy <https://towardsdatascience.com/word2vec-made-easy-139a31a4b8ae>

Алгоритм обучения - Skip Gram Negative Sampling

Skip Gram - предсказываем соседние слова по центральному слову

Negative Sampling - аппроксимация softmax

$$\sum_{CenterW_i} P(CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2}|CenterW_i; W, D) \rightarrow \max_{W, D}$$

$$P(CtxW_{-2}, CtxW_{-1}, CtxW_{+1}, CtxW_{+2}|CenterW_i; W, D) = \prod_j P(CtxW_j|CenterW_i; W, D)$$

$$P(CtxW_j|CenterW_i; W, D) = \frac{e^{w_i \cdot d_j}}{\sum_{j=1}^{|V|} e^{w_i \cdot d_j}} = \text{softmax} \simeq \frac{e^{w_i \cdot d_j^+}}{\sum_{j=1}^{|V|} e^{w_i \cdot d_j}}, \quad k \ll |V|$$


```
In [ ]: def make_diag_mask(size, radius):
    """Квадратная матрица размера Size x Size с двумя полосами ширины radius вдоль главной диагонали"""
    idxs = torch.arange(size)
    abs_idx_diff = (idxs.unsqueeze(0) - idxs.unsqueeze(1)).abs()
    mask = ((abs_idx_diff <= radius) & (abs_idx_diff > 0)).float()
    return mask

make_diag_mask(10, 3)

In [ ]: class SkipGramNegativeSamplingTrainer(nn.Module):
    def __init__(self, vocab_size, emb_size, sentence_len, radius=5, negative_samples_n=5):
        super().__init__()
        self.vocab_size = vocab_size
```

In []: def make_diag_mask(size, radius):
 """Квадратная матрица размера Size x Size с двумя полосами ширины radius вдоль главной диагонали"""
 idxs = torch.arange(size)
 abs_idx_diff = (idxs.unsqueeze(0) - idxs.unsqueeze(1)).abs()
 mask = ((abs_idx_diff <= radius) & (abs_idx_diff > 0)).float()
 return mask

make_diag_mask(10, 3)

In []: class SkipGramNegativeSamplingTrainer(nn.Module):
 def __init__(self, vocab_size, emb_size, sentence_len, radius=5, negative_samples_n=5):
 super().__init__()
 self.vocab_size = vocab_size
 self.negative_samples_n = negative_samples_n

 self.center_emb = nn.Embedding(self.vocab_size, emb_size, padding_idx=0)
 self.center_emb.weight.data.uniform_(-1.0 / emb_size, 1.0 / emb_size)
 self.center_emb.weight.data[0] = 0

 self.context_emb = nn.Embedding(self.vocab_size, emb_size, padding_idx=0)
 self.context_emb.weight.data.uniform_(-1.0 / emb_size, 1.0 / emb_size)
 self.context_emb.weight.data[0] = 0

 self.positive_sim_mask = make_diag_mask(sentence_len, radius)

 def forward(self, sentences):
 """sentences - Batch x MaxSentLength - идентификаторы токенов"""
 batch_size = sentences.shape[0]
 center_embeddings = self.center_emb(sentences) # Batch x MaxSentLength x EmbSize

 # оценить сходство с настоящими соседними словами
 positive_context_embs = self.context_emb(sentences).permute(0, 2, 1) # Batch x EmbSize x MaxSentLength
 positive_sims = torch.bmm(center_embeddings, positive_context_embs) # Batch x MaxSentLength
 positive_probs = torch.sigmoid(positive_sims)


```

    return mask

make_diag_mask(10, 3)

Out[8]: tensor([[0., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
               [1., 0., 1., 1., 0., 0., 0., 0., 0., 0.],
               [1., 1., ①, 1., 1., 0., 0., 0., 0., 0.],
               [1., 1., 1., 0., 1., 1., 0., 0., 0., 0.],
               [0., 1., 1., 0., 1., 1., 1., 0., 0., 0.],
               [0., 0., 1., 1., 0., 1., 1., 1., 0., 0.],
               [0., 0., 0., 1., 1., 0., 1., 1., 1., 0.],
               [0., 0., 0., 0., 1., 1., 0., 1., 1., 1.],
               [0., 0., 0., 0., 0., 1., 1., 1., 0., 1.],
               [0., 0., 0., 0., 0., 0., 1., 1., 1., 0.]])

```



```

In [ ]: class SkipGramNegativeSamplingTrainer(nn.Module):
    def __init__(self, vocab_size, emb_size, sentence_len, radius=5, negative_samples_n=5):
        super().__init__()
        self.vocab_size = vocab_size
        self.negative_samples_n = negative_samples_n

        self.center_emb = nn.Embedding(self.vocab_size, emb_size, padding_idx=0)
        self.center_emb.weight.data.uniform_(-1.0 / emb_size, 1.0 / emb_size)
        self.center_emb.weight.data[0] = 0

        self.context_emb = nn.Embedding(self.vocab_size, emb_size, padding_idx=0)
        self.context_emb.weight.data.uniform_(-1.0 / emb_size, 1.0 / emb_size)
        self.context_emb.weight.data[0] = 0

        self.positive_sim_mask = make_diag_mask(sentence_len, radius)

    def forward(self, sentences):
        """sentences - Batch x MaxSentLength - идентификаторы токенов"""
        batch_size = sentences.shape[0]

```



Из комментариев:

Вопрос:

"Для обучения word2vec эта инициализация совершенно не подходит. Правильный способ инициализации весов для обучения word2vec — это равномерный шум, причём диапазон значений этого шума связан с размерностью эмбеддингов. Такая инициализация нужна для того, чтобы длина вектора, начальная, примерно равнялась единице."

Почему, почему, почему???

Ответ (Романа Суворова):

а вот :) Если честно, я не знаю правильного ответа, но есть следующая гипотеза:

Семплы из нормального распределения в многомерном пространстве располагаются равномерно по относительно тонкой сфере.

Семплы из равномерного распределения в многомерном пространстве заполняют кубик.

Теперь представим, что в процессе обучения все слова тянут и отталкивают друг друга. Если мы усредним все силы, действующие на некоторое слово, и получим что-то близкое к 0, то обучение вообще не будет происходить, вектора слов не будут двигаться. Такая ситуация может происходить, когда у нас очень симметричное заполнение пространства. Есть версия, что при нормальной инициализации как раз это и получается. При равномерной инициализации есть как минимум угловые вершины, которые вносят неоднородность во всю систему и она начинает шевелиться и в результате учится.

Такая вот физическая аналогия, не уверен что она правильная.

Вопрос:

Можете объяснить, почему при инициализации весов равномерно на отрезке [-1/emb_size, 1/emb_size] длина вектора будет в среднем равна единице?

Ответ (Алексея Шадрикова):

Вы правы, единицы не получится, скорее делить надо на корень, но и в этом случае будет "примерно".

Суть не в этом, главное все же не диапазон, а как было сказано - использование равномерного распределения вместо нормального

След. comment:

Понял, спасибо) Еще было предположение, что вектор единичной нормы помогает при обучении, то есть решает проблему затухания/взрыва градиента

Ответ (Алексея Шадрикова):

как я понимаю, обычно главное, чтоб был не ноль, из него тяжело выбраться. Но, как оказалось, распределение тоже влияет. И многое другое почему-то влияет. Вообще интерпретируемость результатов - это очень большой вопрос в машинном обучении. Нередко сначала получают какое-то работающее решение, а потом под него подводят теорию.

Из практического задания:

Обратите внимание, что "размер окна" в этом курсе равен количеству положительных примеров, которые извлекаются для каждого центрального слова (для каждого положения окна), плюс один. Другими словами, центральное слово стоит в центре окна, а размер окна - расстояние между крайними словами, входящими в окно. В других материалах и библиотеках (в том числе у авторов Word2Vec) под размером окна может пониматься "радиус", то есть половина фактической ширины окна. Будьте внимательны!

Из комментариев к практическому заданию:

Вопрос:

Возник вопрос по поводу размера окна. Если совместное распределение рассматривается, как произведение независимых величин, насколько имеет смысл брать размер окна больше 3. Поясните это пожалуйста.

Ответ (Романа Суворова):

брать окна больше 3 определённо имеет смысл, потому что слова, отстоящие друг от друга, скажем на 5 слов, могут быть по прежнему связаны и могут дополнять определение "смысла" друг друга. Однако с ростом ширины окна будет меняться то, что вектора слов будет записываться. В крайнем случае можно сделать окно шириной с весь документ и тогда мы получим тематическую модель по аналогии с [LSA](#) и [LDA](#) (тематические модели в курсе не рассматриваются). В этом случае вектора слов будут описывать тематику документов, в которых они используются. На практике чаще всего окно делается шириной около 10-15 токенов.

Кроме того, обратите внимание, что здесь имеется в виду условная независимость (то есть $P(a,b|c)=P(a|c)P(b|c)P(a,b|c)$, а не безусловная (то есть $P(a,b)=P(a)P(b)P(a,b)$ **не** выполняется). Здесь под a и b понимаются контекстные слова, под c - центральное слово.

Перейдём к прямому проходу по этой модели. Традиционно, метод forward используется для того, чтобы получить предсказание модели для какого-то обучающего примера — например, получить метку класса. Но у нас здесь не вполне обычная модель, мы здесь используем метод forward для того, чтобы вычислить функцию потерь. Таким образом, наш класс модели, который мы сейчас описываем, на самом деле — это не модель, это алгоритм обучения, и мы назвали этот класс соответствующим образом — "trainer". Итак, на вход на каждой итерации нам дают пачку предложений — прямоугольный тензор размерности ["размер батча" на "максимальную длину предложения"]. Этот тензор имеет целочисленный тип. На первом шаге, для каждого токена из батча, мы получаем центральный эмбеддинг, то есть делаем выборку из таблицы центральных [эмбеддингов](#). В результате мы получаем тензор вот такой размерности (трёхмерный [тензор](#)): ["размер батча" на "максимально длину предложения", и здесь у нас добавляется ещё "размер эмбеддинга"] в конец. Как вы помните, процесс обучения [word2vec](#) — это процесс обучения классификатора, который предсказывает — могут ли два слова встретиться в рамках какого-то небольшого окна, или не могут. Соответственно, нам нужны как положительные примеры, так и отрицательные примеры. Сначала разберёмся с положительными примерами. Для этого мы берём всё те же токены — все токены, которые у нас есть в батче, и делаем для них выборку из таблицы контекстных токенов. То есть, если сначала мы делали выборку из таблицы центральных токенов, теперь мы делаем выборку из таблицы контекстных токенов. И сразу же транспонируем её — то есть на этом шаге мы получаем тензор размерности ["батч" на "размер эмбеддинга" на "максимальную длину

предложения"]. А затем мы вызываем функцию "[torch.bmm](#)", которая делает матричное произведение для каждой пары матриц в батче, то есть она принимает на вход два батча матриц прямоугольных, и выполняет матричное произведение для каждой пары. В результате мы получаем трёхмерный тензор размерности ["количество примеров в батче" на "максимальную длину предложения" на "максимальную длину предложения"]. Физический смысл этого тензора — это оценка сходства, оценка семантической близости, для каждой пары токенов, встретившихся внутри предложения. Далее мы преобразовываем оценки сходства в вероятности с помощью сигмоиды. Переменная `positive props` содержит оценки вероятностей того, что два слова встретятся вместе, и такие оценки вычислены для всех возможных пар слов для каждого предложения из батча. Но надо помнить, что мы хотим учитывать не все совместные встречаемости, а только совместные встречаемости внутри небольшого окна. Здесь-то нам и приходит на помощь матрица маски, которую мы построили чуть раньше. Так как мы решаем задачу классификации, то мы используем соответствующую функцию потерь — в данном случае бинарную [кросс-энтропию](#), и здесь — важный момент — мы домножаем тензор вероятностей, который содержит вероятность для всех возможных пар токенов из наших предложений, на маску. Таким образом мы зануляем оценки вероятностей для всех пар токенов, которые не лежат внутри окна. И далее мы говорим, что настоящая метка для этих позиций — это единичка. То есть, для этих пар слов и этих позиций, модель должна предсказывать большую вероятность совместной встречаемости. Итак, с положительными примерами разобрались, давайте перейдём к отрицательным. В названии алгоритма есть слова "[negative sampling](#)", то есть сэмплирование отрицательных слов. Это достаточно буквальное название, то есть в качестве отрицательных слов мы выбираем случайные слова. В данном примере мы выбираем слова равномерно. В более продвинутых реализациях в качестве отрицательных слов используются более редкие слова, то есть сэмплирование уже идёт не равномерно, а с учётом частоты встречаемости каждого слова в обучающей выборке. Но, для простоты, мы опустим эти детали здесь. Итак, мы используем метод "`torch.randint`", который возвращает случайные целые числа в заданном диапазоне, для того, чтобы получить номера токенов, которые мы будем использовать в качестве отрицательных. Далее мы делаем выборку из таблицы контекстных векторов для полученных отрицательных токенов и сразу же её транспонируем. Таким образом, мы получаем трёхмерный тензор размерности ["количество предложений в батче" на "размер эмбеддинга" и на "количество отрицательных слов"], которые мы сэмплируем для каждого обучающего примера. Далее мы оцениваем семантическое сходство слов из настоящих предложений и случайно выбранных отрицательных слов. Делаем это так же — [скалярным произведением](#). Далее мы также используем кросс-энтропию, для того чтобы сходство настоящих слов и отрицательных слов было поменьше, то есть мы передаём нули как таргеты. Кстати, здесь я использую функцию `binary cross entropy with logits`, она на вход принимает не оценки вероятностей, а оценки сходства, то есть не нормализованные значения. По сути, вот эти два способа использовать кросс-энтропию — они примерно одинаковы, но второй способ более стабильный с численной

точки зрения. Ну и, наконец, мы просто возвращаем сумму функции потерь для положительных примеров и для отрицательных.



SAMSUNG Research Russia

```
def forward(self, sentences):
    """sentences - Batch x MaxSentLength - идентификаторы токенов"""
    batch_size = sentences.shape[0]
    center_embeddings = self.center_emb(sentences) # Batch x MaxSentLength x EmbSize

    # оценить сходство с настоящими соседними словами
    positive_context_embs = self.context_emb(sentences).permute(0, 2, 1) # Batch x EmbSize x MaxSentLength
    positive_sims = torch.bmm(center_embeddings, positive_context_embs) # Batch x MaxSentLength x MaxSentLength
    positive_probs = torch.sigmoid(positive_sims)

    # увеличить оценку вероятности встретить эти пары слов вместе
    positive_mask = self.positive_sim_mask.to(positive_sims.device)
    positive_loss = F.binary_cross_entropy(positive_probs * positive_mask,
                                           positive_mask.expand_as(positive_probs))

    # выбрать случайные "отрицательные" слова
    negative_words = torch.randint(1, self.vocab_size,
                                    size=(batch_size, self.negative_samples_n),
                                    device=sentences.device) # Batch x NegSamplesN
    negative_context_embs = self.context_emb(negative_words).permute(0, 2, 1) # Batch x EmbSize x NegSamplesN
    negative_sims = torch.bmm(center_embeddings, negative_context_embs) # Batch x MaxSentLength x NegSamplesN

    # уменьшить оценку вероятность встретить эти пары слов вместе
    negative_loss = F.binary_cross_entropy_with_logits(negative_sims,
                                                       negative_sims.new_zeros(negative_sims.size()))

    return positive_loss + negative_loss

def no_loss(pred, target):
    """Фиктивная функция потерь - когда модель сама считает функцию потерь"""
    return pred
```

Из комментариев:

Вопрос:

Вопрос в формуле выше указывается что у нас soft max, а алгоритме указана сигмойда, это не ошибка?

Ответ:

в [оригинальной статье](#) (стр.3) авторы используют логсигмоиду, мотивируя это тем, что снижения качества векторов при таком упрощении не происходит:

While NCE can be shown to approximately maximize the log probability of the softmax, the Skipgram model is only concerned with learning high-quality vector representations, so we are free to simplify NCE as long as the vector representations retain their quality

Также отмечу, что $(\text{sigmoid} + \text{binary_cross_entropy}) == (\text{binary_cross_entropy_with_logits})$

Вопрос:

1) Можно ли было не делать отдельно позитивные и негативные примеры, а просто посчитать $\text{binary_cross_entropy}(\text{positive_probs}, \text{positive_mask.expand_as}(\text{positive_probs}))$? То есть соседние слова - это всё ещё наши позитивные примеры, а негативные - слова из предложения, которые не попали в окно. Я понимаю, что данный в лекции вариант подходит лучше, но теоретически это работало бы?

2) Как вообще по этой функции модель понимает, как и что ей оптимизировать во время выполнения backprop? Хотелось бы подробнее об этом

Ответ (романа Суворова):

1) В целом можно попробовать, но кажется, с этим подходом есть пара проблем. Если мы будем использовать в качестве негативных слов только слова из предложения, то у нас будет мало отрицательных примеров как по количеству, так и по разнообразию. К тому же не факт, что мы хотим так жестко проводить грань между положительными и отрицательными примерами (всё-таки, когда мы семплируем отрицательные из словаря, эта грань сильно размыта). Скорее всего такое обучение приведёт к каким-то совсем другим эмбеддингам, более грамматическим, (потому что даже когда слово встречается рядом с центральным, но не прямо вплотную, оно будет считаться отрицательным). Попробуйте и расскажите, очень интересно!

2) Каждый тензор в pytorch хранит ссылку на функцию и исходные аргументы, как он был получен. Когда Вы вызываете `loss.backward()`, выполняется обход графа от лосса к листьям (тензорам параметров и входных данных) и постепенно вычисляются частные производные. Эта информация съедает достаточно много памяти, поэтому на практике стоит использовать `torch.no_grad` везде, где Вам не нужно пропускать градиенты по графу назад.

Доп. вопрос:

А как во время этого обхода и изменения весов определяется, где у нас тензоры-параметры (входы), а где тензоры-веса (настраиваемые параметры)? Чтобы не начинать изменять исходные данные, например

Ответ (Иван Ш):

за это отвечает параметр `requires_grad` у тензоров, по умолчанию он `False` и вектор считается константой, поэтому `X` и `u` не изменяются при обратном распространении ошибки, при создании экземпляра класса `nn.Embedding` у весов отслеживание градиента уже включено.

Мой комментарий (листинг с моими развернутыми комментариями):

```
class SkipGramNegativeSamplingTrainer(nn.Module):
    ...
    ...
    ...
    def forward(self, sentences):
        """sentences - Batch x MaxSentLength - идентификаторы токенов"""
        batch_size = sentences.shape[0]

        #получает на вход LongTensor с idx (т.е. индексами токенов), возвращает тензор + 1 измерения
        #в котором индексы заменены на соответствующие им embedding'и (это центральные слова)
        #Итого(для batch=1): мы получаем тензор предложения фиксированной длины, где каждое слово
        #заменено на embedding из центральных слов, все отсутствующие слова (нет в словаре или
        #закончилось реальное предложение), заменяются на embedding из 0
        center_embeddings = self.center_emb(sentences) # Batch x MaxSentLength x EmbSize

        ### оценить сходство с настоящими соседними словами

        #получает на вход LongTensor с idx (т.е. индексами токенов), возвращает тензор + 1 измерения
        #в котором индексы заменены на соответствующие им embedding'и, (это контекстные слова)
        #дополнительно транспонируем для целей последующего тензорного (матричного) умножения
```

```

#Итого(для batch=1): мы получаем тензор предложения фиксированной длины, где каждое слово
#заменено на embedding из контекстных слов, все отсутствующие слова (нет в словаре или
#закончилось реальное предложение), заменяются на embedding из 0
positive_context_embs = self.context_emb(sentences).permute(0, 2, 1) # Batch x EmbSize x
MaxSentLength

#перемножение тензоров, по сути, скалярное произведение эмбеддингов,
#Важно отметить, что изначально я предполагал, что эта операция равносильна нахождению косинусных
расстояний,
#т.к. на основе анализа итоговых эмбеддингов, сделал неверный вывод, что длина каждого из
векторов уже здесь = 1
#(т.е. они сразу нормализуются в пределах каждого embedding (например внутри класса
torch.nn.Embedding),
#но это не так, нормализация происходит уже после полного обучения модели, через передачу весов в
конструктор
#созданного вручную класса Embedding)
#Итого(для batch=1): мы получаем матрицу MaxSentLength x MaxSentLength, скалярных произведений,
#между векторами каждого центрального слова и каждого контекстного слова (значения [-inf; inf])
positive_sims = torch.bmm(center_embeddings, positive_context_embs) # Batch x MaxSentLength x
MaxSentLength

#преобразуем в "условные вероятности" через взятие сигмоиды, т.е. получаем как бы
#"условные вероятности" встретить пары слов вместе, по факту для каждой пары, скалярное
произведение,
#обернутое в сигмоиду и как следствие в диапазон значений (0; 1)
positive_probs = torch.sigmoid(positive_sims)

### увеличить оценку вероятности встретить эти пары слов вместе

#переводим тензор self.positive_sim_mask на тот же девайс, на котором positive_sims
positive_mask = self.positive_sim_mask.to(positive_sims.device)

#.expand_as - Expand this tensor to the same size as other.
#self.expand_as(other) is equivalent to self.expand(other.size())
#positive_probs * positive_mask - мы оставляем только позиции пересечения центральных слов в
контекстными,
#все остальные позиции зануляются
#подсчитываем бинарную кросс энтропию вычисленных "условных вероятностей" (сигмоид) и целевых = 1
для всех
#пересечений центральных и контекстных слов, все остальные позиции в обоих матрицах = 0
#Примечание: т.к. по умолчанию BCELoss в реализации torch высчитывает итоговое значение как
'mean',
#a не 'sum' из всех полученных, то количество 0 так же влияет на итоговое значение, имеет ли это
какой
#то эффект, и измениться ли что то, если выставить reduction='sum', не очевидно и нужно проверять
на практике
#Примечание: для всех позиций, которые занулены, их эмбеддинги соответствуют эмбеддинг-вектору с
idx=0, для
#для которого мы при создании мы указали паддинг nn.Embedding(..., padding_idx=0), это означает,
что эти веса
#фиксированы, и не подлежат изменению через градиентных шаг
#

```

```

#Итого: важно понимать, что если бы оптимизировали только данную loss функцию, без отрицательных
примеров,
#которые идут ниже, то, все сводилось бы к тому, что минимальное значение loss было бы, если бы
мы все
    #вектора (и центральных слов и контекстных) устремили бы в бесконечность, в одном направлении
    #например всем
        #их весам присвоили бы значение inf или любые подобные варианты)
        positive_loss = F.binary_cross_entropy(positive_probs * positive_mask,
                                                positive_mask.expand_as(positive_probs))

    ### выбрать случайные "отрицательные" слова
    negative_words = torch.randint(1, self.vocab_size,
                                    size=(batch_size, self.negative_samples_n),
                                    device=sentences.device) # Batch x NegSamplesN
    negative_context_embs = self.context_emb(negative_words).permute(0, 2, 1) # Batch x EmbSize x
    NegSamplesN
    negative_sims = torch.bmm(center_embeddings, negative_context_embs) # Batch x MaxSentLength x
    NegSamplesN

    ### уменьшить оценку вероятность встретить эти пары слов вместе
    #Важно отметить, что BCEWithLogitsLoss равносильна последовательному применению Sigmoid ->
BCELoss
    #но в реализации torch она является более численно стабильной, чем раздельное применение
    #Итого: здесь все целевые (target) значения = 0, и если бы мы минимизировали только эту loss
функцию, то минимальное
    #ее значение было бы, если бы мы устремили все вектора центральных слов в бесконечность одного
направления,
    #а вектора контекстных слов в бесконечность противоположного направления
    negative_loss = F.binary_cross_entropy_with_logits(negative_sims,
                                                        negative_sims.new_zeros(negative_sims.shape))
    return positive_loss + negative_loss

```

Давайте уже наконец перейдём к самому обучению. Сначала мы создаём экземпляр класса `trainer`, то есть — того класса, который мы сейчас описывали, и передаём туда размер нашего словаря, размер [эмбеддингов](#), которые мы строим, максимальную длину предложения, а также описываем ширину окна, в рамках которого нам нужно учитывать совместные встречаемости слов. И для каждого предложения мы будем сэмплировать 25 отрицательных слов. Обучать модель мы будем с помощью функции `"train_eval_loop"`, она реализует стандартный цикл для обучения нейросети, когда мы берём батч примеров из датасета, подаём на вход нейросети, вычисляем функцию ошибки, делаем градиентный шаг, и так далее — повторяем несколько таких эпох. И на каждой эпохе мы делаем проход как по обучающей подвыборке, так и по тестовой. Правда, для этого семинара мы внесли в этот цикл пару изменений — основное изменение заключается в том, что мы будем менять скорость обучения в процессе обучения. Логика такая, что когда, спустя несколько итераций, значение функции потерь перестаёт уменьшаться, мы считаем, что мы достигли некоторого предела, используя градиентные шаги такой длины, и говорим, что дальше будем делать более короткие шаги. Это позволяет нам

спускаться в более узкие локальные минимумы, в которое мы бы не могли зайти, используя длинные шаги. Итак поехали обучать!



SAMSUNG Research Russia

```
negative_sims.new_zeros(negative_sims.shape))

return positive_loss + negative_loss

def no_loss(pred, target):
    """Фиктивная функция потерь - когда модель сама считает функцию потерь"""
    return pred
```

Обучение

```
In [ ]: trainer = SkipGramNegativeSamplingTrainer(len(vocabulary), 100, MAX_SENTENCE_LEN,
                                                 radius=5, negative_samples_n=25)

In [ ]: best_val_loss, best_model = train_eval_loop(trainer,
                                                    train_dataset,
                                                    test_dataset,
                                                    no_loss,
                                                    lr=1e-2,
                                                    epoch_n=10,
                                                    batch_size=8,
                                                    device='cpu',
                                                    early_stopping_patience=10,
                                                    max_batches_per_epoch_train=2000,
                                                    max_batches_per_epoch_val=len(test_dataset),
                                                    lr_scheduler_ctor=lambda optim: torch.optim.lr_scheduler.LambdaLR(optim, lambda epoch: 0.95 ** epoch))

In [ ]: torch.save(trainer.state_dict(), 'models/sgns.pth')

In [ ]: trainer.load_state_dict(torch.load('models/sgns.pth'))
```



SAMSUNG Research Russia

```
negative_sims.new_zeros(negative_sims.shape))

return positive_loss + negative_loss

def no_loss(pred, target):
    """Фиктивная функция потерь - когда модель сама считает функцию потерь"""
    return pred
```

Обучение

```
In [10]: trainer = SkipGramNegativeSamplingTrainer(len(vocabulary), 100, MAX_SENTENCE_LEN,
                                                 radius=5, negative_samples_n=25)

In [ ]: eval_loop(trainer,
                  train_dataset,
                  test_dataset,
                  no_loss,
                  lr=1e-2,
                  epoch_n=10,
                  batch_size=8,
                  device='cpu',
                  early_stopping_patience=10,
                  max_batches_per_epoch_train=2000,
                  max_batches_per_epoch_val=len(test_dataset),
                  lr_scheduler_ctor=lambda optim: torch.optim.lr_scheduler.ReduceLROnPlateau(optim, patience=10))

In [ ]: torch.save(trainer.state_dict(), 'models/sgns.pth')

In [ ]: trainer.load_state_dict(torch.load('models/sgns.pth'))
```

Из комментариев:

Вопрос:

полное описание лямбда-функции потерялось за спиной. сложно ее работу понять

Ответ:

для всех, кому нужна ссылка: <https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate>

Вопрос:

Хочу задать вопрос. Мне не понятно как происходит обучение. В процессе обучения нам нужно взять градиент функции потерь по D и W. Но функции потерь как таковой у нас ведь нет. Мы ведь используем no_loss. Что мы оптимизируем? Также не понятно на каком этапе происходит изменение D и W в соответствии с градиентом.

Ответ (Николая Копырина):

Спасибо за вопрос. Я пересмотрел предыдущий и следующий шаг, и могу пояснить, что такое no_loss: здесь нас не интересует прогноз модели, а интересуют только её внутренние веса, которые она научится сопоставлять каждому слову в словаре. Их мы потом и будем вытаскивать из нейросети и пользоваться ими как новым словарём... Поэтому метод forward возвращает не традиционное "предсказание для данного объекта", а просто качество работы модели на текущем этапе обучения (видимо, это упрощает интерфейсы). А функция no_loss транслирует нам "предсказание", но уже в качестве лосса. Может быть, вы спрашивали о другом? Скажите пожалуйста, насколько теперь всё понятно. Может быть, я тоже не всё понял.



Доп. комментарий:

спасибо за ответ! Вы правы, меня интересует немного другое. А именно: как модифицируются значения матриц W и D? Откуда torch знает, что W и D нужно модифицировать после каждой эпохи?

Доп. комментарий:

Нашел ответ на свой вопрос в исходниках pytorch.

Оказывается, все атрибуты объектов класса nn.Module, являющиеся наследниками Parameter доступны в методе nn.Module#parameters(). Более того, в этом методе доступны все parameters() других атрибутов-модулей. Достигается это посредством магического метода __setattr__. Звучит немного запутано, но на самом деле все просто. Ниже приведу код с примером.

```
class ModuleA(nn.Module):
    def __init__(self):
        super().__init__()
        self.parameter_a = nn.Parameter(torch.tensor([5.0]))

class ModuleB(nn.Module):
    def __init__(self):
        super().__init__()
        self.module_a = ModuleA()
        self.parameter_b = nn.Parameter(torch.tensor([5.0]))

module_b = ModuleB()

for name, param in module_b.named_parameters(recurse=True):
```

```
print(name, param.size())

# Вывод программы:

>> parameter_b torch.Size([1])
>> module_a.parameter_a torch.Size([1])
```

А в библиотеке nlpdlutils есть функция train_eval_loop. Так вот, эта функция использует тот самый метод parameters() для инициализации оптимизатора:

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=l2_reg_alpha)
```

Ну а дальше все очень просто. Вызывается метод, считающий градиенты для функции потерь по всем тензорам, в нее входящим:

```
loss.backward()
```

И вызов

```
optimizer.step()
```

модифицирует model.parameters().

Ответ (Николая Копырина):

спасибо что разобрались и отлично вё расписали для остальных! Я был готов ответить, что "вызов *backward* наделяет переменные модели градиентами, а вызов *step* применяет градиенты к вектору параметров", но боюсь это было бы поверхностно.

Вопрос:

а как мы выбираем размер эмбеддинга ?

Ответ (Николая Копырина):

Можно лишь сказать, что это эвристический выбор. Лучше всего – по аналогии с другими работами... Ведь это как количество нейронов в сети: если взять недостаточно – потеряете в выразительности, если взять слишком много – будет очень долгий процесс обучения и, может быть, сеть станет словно база данных по обучающим примерами (переобучение).

Прошло немного времени, наша модель обучилась — мы видим, что даже уже за одну эпоху функция потерь достигла приемлемого значения и в последующих итерациях она менялась слабо. Наше дополнение для уменьшения скорости обучения тоже сработало. На всякий случай, сохраним нашу модель. Давайте теперь достанем вектора слов из обученной модели и посмотрим — что же мы там "научили". Для того, чтобы было удобно экспериментировать с обученными [эмбеддингами](#) слов, мы сделали небольшой класс, который на вход принимает питчуру-массив двумерный — прямоугольную матрицу, количество строк в которой совпадает с количеством слов в словаре, а количество столбцов с размерностью эмбеддинга. А также, этот

класс принимает словарь, который отображает токены в текстовом виде в номера токенов. Важный момент — для того, чтобы было удобнее искать похожие слова, мы нормируем вектора — каждый вектор нормируется на его евклидову норму (на его длину). Этот класс поддерживает несколько методов — например, поиск ближайших слов, решение семантической пропорции — то есть задача аналогии, а также получение списка векторов для слов. Все эти функции, так или иначе, завязаны на самое главное — это вычисление близости векторов. Этот алгоритм реализуется в методе `most similar by vector`. Метод принимает на вход вектор и возвращает список пар. В каждой паре первый элемент — это токен в строковом виде, второй — это оценка сходства с вектором-запросом. Мы здесь сначала находим сходство данного вектора со всеми вообще векторами в нашей таблице, а затем выбираем заданное количество наиболее близких. Итак, давайте поэкспериментируем с полученными эмбеддингами, посмотрим, "что же там научилось". Давайте найдём слова, самые близкие по смыслу, к слову "сыр". Мы видим, что список похожих слов содержит, в первую очередь, сорта сыра — чеддер, пармезан, рикотта, и так далее. В принципе, выглядит неплохо! Давайте ещё какой-нибудь пример посмотрим — например, "курица". Мы видим, что здесь "утка", "индейка", "грудка" и другие виды мяса. Давайте также попробуем решить смысловую пропорцию, то есть найти слова которые относятся к сыру так же, как "какао" относится к "пирожному". Список похожих слов получился не очень осмысленным, это вполне понятно, потому что предложения короткие, и "сыр", допустим, с "вином", редко встречается в одном предложении — хотя бы поэтому. Из такого небольшого [корпуса](#), как наш, мы могли и не выучить нужные закономерности. То есть, размер корпуса очень важен при обучении дистрибутивно-семантических моделей. Давайте теперь получим вектора для сразу нескольких слов и набросаем их на плоскость. Наши вектора имеют размерность "100". Вектора размерности "100" непосредственно нельзя визуализировать — мы можем визуализировать только вектора размерности не больше трёх, но более удобно использовать вообще "2", чтобы всё на плоскости было. У нас в библиотеке есть специально небольшая функция для этого. Что же у нас тут выучилось? На этом графике мы видим, что — допустим, слова, соответствующие сортам винограда ближе к друг другу, чем к другим словам, а также мясные и рыбные изделия тоже объединились в группу. Центральная группа соответствует тоже каким-то напиткам, куда попало и "вино", и "какао", и "кофе". Алгоритм обучения [word2vec](#) не детерминированный, потому что мы случайно инициализируем таблицы эмбеддингов, делаем случайные шаги по случайно выбранным группам примеров, мы используем случайное сэмплирование отрицательных слов, и поэтому, если вы несколько раз запустите обучение, то вы получите разные рисунки и немного отличающиеся списки "похожих" слов.

Среднее значение функции потерь на валидации 0.8705821942058146
Новая лучшая модель!

Эпоха 7
Эпоха: 2001 итераций, 22.73 сек
Среднее значение функции потерь на обучении 0.8693270162306447
Среднее значение функции потерь на валидации 0.8696883594554836
Новая лучшая модель!

Эпоха 8
Эпоха: 2001 итераций, 22.89 сек
Среднее значение функции потерь на обучении 0.8676051439969674
Среднее значение функции потерь на валидации 0.869179342259956
Новая лучшая модель!

Эпоха 9
Эпоха: 2001 итераций, 22.73 сек
Среднее значение функции потерь на обучении 0.8663681497697768
Среднее значение функции потерь на валидации 0.8689625071307571
Новая лучшая модель!

In [12]: `torch.save(trainer.state_dict(), 'models/sgns.pth')`

In [13]: `trainer.load_state_dict(torch.load('models/sgns.pth'))`

Исследуем характеристики полученных векторов

In []: `embeddings = Embeddings(trainer.center_emb.weight.detach().cpu().numpy(), vocabulary)`

To In []: `embeddings.most_similar('cheese')`



In [13]: `trainer.load_state_dict(torch.load('models/sgns.pth'))`

Исследуем характеристики полученных векторов

In []: `embeddings = Embeddings(trainer.center_emb.weight.detach().cpu().numpy(), vocabulary)`

In []: `embeddings.most_similar('cheese')`

In []: `embeddings.analogy('cake', 'cacao', 'cheese')`

In []: `test_words = ['salad', 'fish', 'salmon', 'sauvignon', 'beef', 'pork', 'steak', 'beer', 'cake', 'coffee', 'chicken', 'potato']
test_vectors = embeddings.get_vectors(*test_words)
print(test_vectors.shape)`

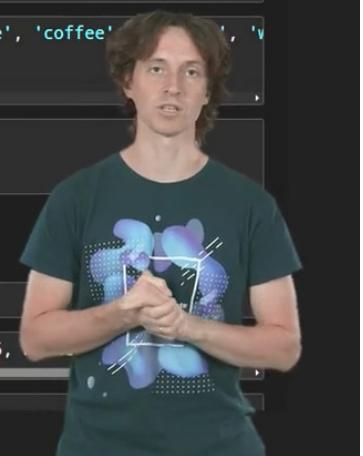
In []: `fig, ax = plt.subplots()
fig.set_size_inches(10, 10)
plot_vectors(test_vectors, test_words, how='svd', ax=ax)`

Обучение Word2Vec с помощью Gensim

In []: `import gensim`

In []: `word2vec = gensim.models.Word2Vec(sentences=train_tokenized, size=100, window=5, min_count=5,`

To In []: `word2vec.wv.most_similar('cheese')`



```
import numpy as np

class Embeddings:
    def __init__(self, embeddings, word2id):
        self.embeddings = embeddings
        self.embeddings /= (np.linalg.norm(self.embeddings, ord=2, axis=-1, keepdims=True) + 1e-4)
        self.word2id = word2id
        self.id2word = {i: w for w, i in word2id.items()}

    def most_similar(self, word, topk=10):
        return self.most_similar_by_vector(self.get_vector(word), topk=topk)

    def analogy(self, a1, b1, a2, topk=10):
        a1_v = self.get_vector(a1)
        b1_v = self.get_vector(b1)
        a2_v = self.get_vector(a2)
        query = b1_v - a1_v + a2_v
        return self.most_similar_by_vector(query, topk=topk)

    def most_similar_by_vector(self, query_vector, topk=10):
        similarities = (self.embeddings * query_vector).sum(-1)
        best_indices = np.argpartition(-similarities, topk, axis=0)[:topk]
        result = [(self.id2word[i], similarities[i]) for i in best_indices]
        result.sort(key=lambda pair: -pair[1])
        return result

    def get_vector(self, word):
        if word not in self.word2id:
            raise ValueError('Невозвестное слово "{}".format(word)')
        return self.embeddings[self.word2id[word]]

    def get_vectors(self, *words):
        return [self.get_vector(w) for w in words]
```



```
In [13]: trainer.load_state_dict(torch.load('models/sgns.pth'))
```

Исследуем характеристики полученных векторов

```
In [14]: embeddings = Embeddings(trainer.center_emb.weight.detach().cpu().numpy(), vocabulary)
```

```
In [15]: embeddings.most_similar('cheese')
```

```
Out[15]: [('cheese', 0.99999353),  
          ('cheddar', 0.5970855),  
          ('parmesan', 0.55999684),  
          ('ricotta', 0.5533215),  
          ('pecorino', 0.5461085),  
          ('gruyere', 0.52810967),  
          ('goat', 0.5276993),  
          ('gruyère', 0.5167546),  
          ('monterey', 0.50285184),  
          ('grated', 0.49667817)]
```

```
In [ ]: embeddings.analogy('cake', 'cacao', 'cheese')
```

```
In [ ]: fig, ax = plt.subplots()
fig.set_size_inches((10, 10))
plot_vectors(test_vectors, test_words, how='svd', ax=ax)
```



```
('ricotta', 0.655511),
('mozzarella', 0.6642516),
('cheddar', 0.6342944),
('emmenthal', 0.6199282),
('reggiano', 0.618102),
('pecorino', 0.61713946),
('asiago', 0.6088553),
('fontina', 0.6078637)]
```

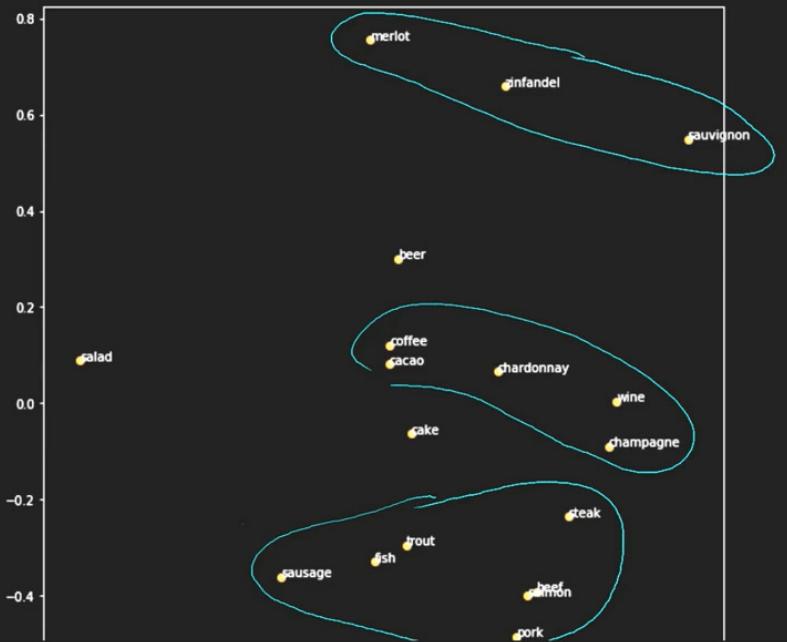
```
In [21]: test_words = ['salad', 'fish', 'salmon', 'sauvignon', 'beef', 'pork', 'steak', 'beer', 'cake', 'coffee', 'sausage', 'trout']
test_vectors = embeddings.get_vectors(*test_words)
print(test_vectors.shape)
```

(18, 100)

```
In [22]: fig, ax = plt.subplots()
fig.set_size_inches(10, 10)
plot_vectors(test_vectors, test_words, how='svd', ax=ax)
```



```
fig.set_size_inches(10, 10)
plot_vectors(test_vectors, test_words, how='svd', ax=ax)
```



До настоящего момента мы пользовались только собственной реализацией [word2vec](#). Однако хорошие люди постарались за нас и уже сделали просто супер библиотеку, в которой реализовано много разных видов [эмбеддингов](#), не только word2vec, и не только [negative sampling](#) — библиотека называется [gensim](#). Давайте применим её к тому же самому [корпусу](#)

про рецепты и посмотрим, что она выучила. Найдём слова, похожие по смыслу на слово "сыр". Видим, что список также содержит, в основном, сорта сыра, но уже в другом порядке, и оценки сходства тоже отличаются от тех, которые мы получили с помощью нашей модели. Давайте попробуем ввести другие слова-запросы. Список слов, похожих на слово "курица", тоже отличается — например, здесь есть не только виды мяса, но и способы приготовления. Давайте также нарисуем вектора слов, полученных с помощью gensim, на плоскости. Мы видим, что слова на плоскости расположены вообще по-другому — не так, как это было в случае с нашей моделью, но, по-прежнему, выделяются кластера. Например, есть большой кластер с напитками, и да — здесь есть не только "вино", но — "пиво", "какао" и "салат". Но, в целом, кластер — про напитки и какие-то ещё продукты, которые связаны с напитками — например пирожные. Второй кластер, скорее, про мясные и рыбные продукты — "рыба", "лосось", "сосиски", "стейки", и так далее. Казалось бы — здорово, авторы gensim реализовали за нас хорошие алгоритмы, эти алгоритмы быстрые и ими удобно пользоваться и они работают... Казалось бы, что ещё надо? Но обучение — оно тоже требует ресурсов. И, самое главное — обучение требует сбора большой обучающей выборки и очистки этой выборки, это очень трудоёмкий процесс. Хорошая новость заключается в том, что в интернете уже есть много обученных моделей — таблиц эмбеддингов, они получены не только с помощью word2vec, но и с помощью других алгоритмов — [GloVe](#), [FastText](#), и так далее, и авторы gensim любезно встроили в свою библиотеку функции для того, чтобы можно было удобно скачивать готовые эмбеддинги и просто их использовать, ничего у себя не обучая. Давайте попробуем загрузить модельку. Загрузка модельки может потребовать некоторого времени и (осторожно) — она достаточно большая, больше полутора гигабайт. Также на экране есть список предобученных эмбеддингов, которые можно вот так просто скачать с помощью gensim. Число в названии эмбеддинга соответствует размерности вектора.

<https://rusvectores.org/ru/> - эмбеддинги для русского языка

<https://wikipedia2vec.github.io/wikipedia2vec/pretrained/> - претренированные эмбеддинги для разных языков

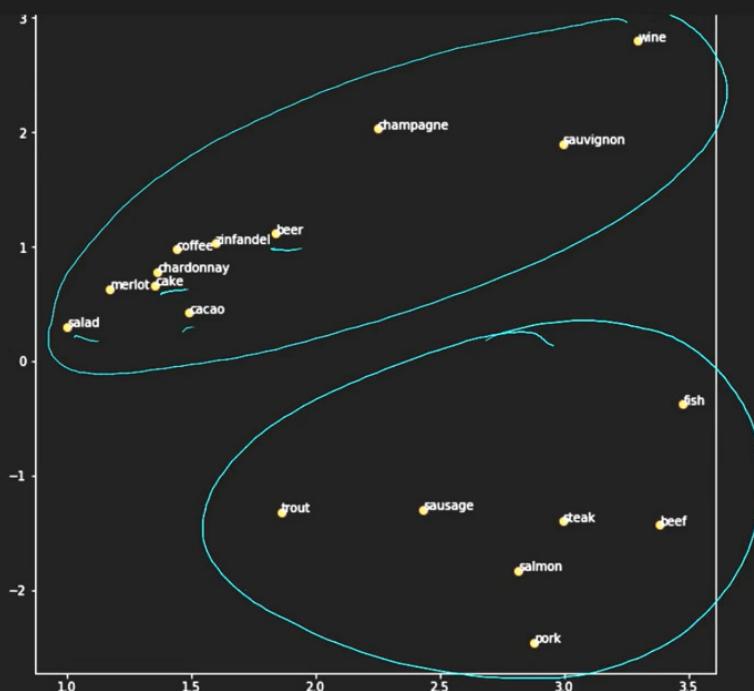
<https://wikipedia2vec.github.io/demo/> - очень интересная 3D визуализация эмбеддингов (соседей)

Обучение Word2Vec с помощью Gensim

```
In [23]: import gensim
In [24]: word2vec = gensim.models.Word2Vec(sentences=train_tokenized, size=100,
                                         window=5, min_count=5, workers=4,
                                         sg=1, iter=10)
In [25]: word2vec.wv.most_similar('cheese')
Out[25]: [('salata', 0.8039901256561279),
           ('gruyere', 0.764767050743103),
           ('pecorino', 0.7603853940963745),
           ('monterey', 0.7370823621749878),
           ('ricotta', 0.7339966297149658),
           ('feta', 0.7327274680137634),
           ('fontina', 0.7314853668212891),
           ('cheddar', 0.7268410325050354),
           ('queso', 0.7254167795181274),
           ('gouda', 0.7254153490066528)]
```

```
In [ ]: gensim_words = [w for w in test_words if w in word2vec.wv.vocab]
gensim_vectors = np.stack([word2vec.wv[w] for w in gensim_words])
In [ ]: fig, ax = plt.subplots()
fig.set_size_inches(10, 10)
plot_vectors(gensim_vectors, test_words, how='svd', ax=ax)
```

Загрузка предобученного Word2Vec



Загрузка предобученного Word2Vec

Источники готовых векторов:

<https://rusvectores.org/ru/> - для русского языка

<https://wikipedia2vec.github.io/wikipedia2vec/pretrained/> - много разных языков

```
In [29]: import gensim.downloader as api  
  
In [*]: available_models = api.info()['models'].keys()  
print('\n'.join(available_models))
```

```
fasttext-wiki-news-subwords-300  
conceptnet-numberbatch-17-06-300  
word2vec-ruscorpora-300  
word2vec-google-news-300  
glove-wiki-gigaword-50  
glove-wiki-gigaword-100  
glove-wiki-gigaword-200  
glove-wiki-gigaword-300  
glove-twitter-25  
glove-twitter-50  
glove-twitter-100  
glove-twitter-200  
_testing_word2vec-matrix-synopsis
```

```
In [*]: pretrained = api.load('word2vec-google-news-300') # > 1.5 GB!  
  
In [ ]: pretrained.wv.most_similar('olive')
```



Из комментариев:

В word2vec заменила параметры на **vector_size** (Formerly: *size*) и **epochs** (Formerly: *iter*)

Ну и ещё некоторые вещи, вот полезная ссылочка по изменениям в 4 версии

библиотеки <https://github.com/RaRe-Technologies/gensim/wiki/Migrating-from-Gensim-3.x-to-4>

Модель загрузилась, это потребовало некоторого времени, давайте теперь поищем похожие слова. Итак, слова-запрос "курица". Видим, что список похожих слов содержит либо блюда из курицы, либо виды мяса. Аналогичный список для запроса "сыр" — опять же, здесь "чеддер" и другие сорта сыра. Кстати, эта модель была обучена не только на отдельных словах, но и на парах слов — на bigramмах. Для того, чтобы такую модель обучить, надо заранее предобработать корпус. А именно, нужно объединить часто идущие подряд слова в "фиктивные слова", которые соответствуют словосочетаниям. Давайте попробуем решить смысловую пропорцию — то есть, решить задачу аналогии. Как вы помните, модель, обученная на нашем маленьком корпусе, не справилась с этой задачей. А сможет ли большая модель это сделать? Ну что ж... О, кажется получается. В этой ячейке мы спросили у модели, какое слово относится к "королеве" так же как слово "мужчина" относится к слову "король". Ну и, вполне ожидаемо, мы получили такие слова как "женщина", "девушка", и так далее. Давайте теперь визуализируем эмбеддинги из большой модели. Опять же, слова расположены совсем по-другому но явно выделяется пара кластеров, более-менее осмысленных. Ну, и давайте подытожим. В этом семинаре мы реализовали руками, с помощью pytorch, модель SkipGram с сэмплированием отрицательных слов, обучили её на корпусе рецептов и немножко поиграли с

этой моделью — то есть поискали похожие слова, порисовали расположение слов на плоскости и выяснили, что, в целом, кластеризация достаточно разумная, то есть сходство слов модель выучила неплохо, но семантические отношения модель не выучила, скорее всего ввиду того, что мы обучали её на отдельных коротких предложениях. Часто для обучения больших моделей тексты не разбивают на предложения, а рассматривают весь текст как единую последовательность токенов. Также мы немножко коснулись библиотеки [gensim](#), а именно — обучили модель a SkipGram с отрицательным сэмплированием с помощью генсима, а также мы загрузили предобученные веса и тоже немножко их поисследовали. Причём мы увидели, что большая модель неплохо аппроксимирует не только близость слов, но и какие-то семантические отношения, то есть уже как-то решает смысловые пропорции.

```
glove-wiki-gigaword-300
glove-twitter-25
glove-twitter-50
glove-twitter-100
glove-twitter-200
__testing_word2vec-matrix-synopsis
```

```
In [31]: pretrained = api.load('word2vec-google-news-300') # > 1.5 GB!
```

```
In [44]: pretrained.most_similar('cheese')
```

```
Out[44]: [('cheeses', 0.7789000272750854),
 ('cheddar', 0.7627596855163574),
 ('goat_cheese', 0.7297402620315552),
 ('Cheese', 0.7286962270736694),
 ('cheddar_cheese', 0.7255136966705322),
 ('Cheddar_cheese', 0.6943709254264832),
 ('mozzarella', 0.6805710196495056),
 ('cheddar_cheeses', 0.6694672703742981),
 ('Camembert', 0.6623162031173706),
 ('gruyere', 0.6615148782730103)]
```

```
In [ ]: pretrained.most_similar(positive=['man', 'queen'], negative=['king'])
```

```
In [ ]: pretrained_words = [w for w in test_words if w in pretrained.vocab]
pretrained_vectors = np.stack([pretrained[w] for w in pretrained_words])
```

```
In [ ]: fig, ax = plt.subplots()
fig.set_size_inches((10, 10))
plot_vectors(pretrained_vectors, test_words, how='svd', ax=ax)
```



```
Out[44]: [('cheeses', 0.7789000272750854),
 ('cheddar', 0.7627596855163574),
 ('goat_cheese', 0.7297402620315552),
 ('Cheese', 0.7286962270736694),
 ('cheddar_cheese', 0.7255136966705322),
 ('Cheddar_cheese', 0.6943709254264832),
 ('mozzarella', 0.6805710196495056),
 ('cheddar_cheeses', 0.6694672703742981),
 ('Camembert', 0.6623162031173706),
 ('gruyere', 0.6615148782730103)]
```

```
In [45]: pretrained.most_similar(positive=['man', 'queen'], negative=['king'])
```

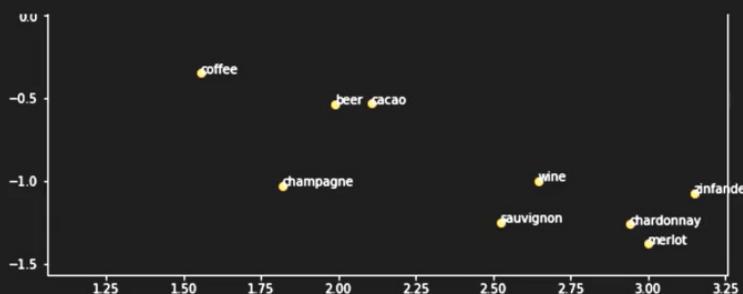
```
Out[45]: [('woman', 0.7609435319900513),
 ('girl', 0.6139993667602539),
 ('teenage_girl', 0.6040961742401123),
 ('teenager', 0.5825759172439575),
 ('lady', 0.5752554535865784),
 ('boy', 0.5077577829360962),
 ('policewoman', 0.5066847205162048),
 ('schoolgirl', 0.5052095651626587),
 ('blonde', 0.48696190118789673),
 ('person', 0.48637545108795166)]
```

```
In [ ]: pretrained_words = [w for w in test_words if w in pretrained.vocab]
pretrained_vectors = np.stack([pretrained[w] for w in pretrained_words])
```

```
In [ ]: fig, ax = plt.subplots()
fig.set_size_inches((10, 10))
plot_vectors(pretrained_vectors, test_words, how='svd', ax=ax)
```



```
fig.set_size_inches((10, 10))  
plot_vectors(pretrained_vectors, test_words, how='svd', ax=ax)
```



Заключение

- Реализовали Skip Gram Negative Sampling на PyTorch
- Обучили на корпусе рецептов
 - Сходство слов модель выучила неплохо
 - Для аналогий мало данных
- Обучили SGNS с помощью библиотеки Gensim
- Загрузили веса Word2Vec, полученные с помощью большого корпуса (GoogleNews)
 - Списки похожих слов отличаются!
 - Аналогии работают

In []:

