Задачи и алгоритмы обработки текстов

NLP (Natural Language Processing)

Что такое NLP?

- Набор техник\подходов для решения задач, в которых присутствует текст, как (один из) видов данных

Может быть вещью в себе:

- Машинный перевод

Или частью большей системы:

- Поиск страницы релевантной запросу

Примеры задач обработки текста

- Анализ тональности высказываний
  - например, отзывы на товары
- Классификация сообщений пользователя
  - “Хочу записаться на стрижку”, “спасибо, не надо”
- Поиск релевантных документов
- Синтез текстов
- Машинный перевод
  - Например, с английского на французский
- Определение авторства произведения
- Экстракция сущностей из текста
  - Найти\опознать все имена собственные и адреса в тексте
- И многое многое другое ...

Отличия задач обработки текста

- Работа с последовательностями переменной длины
  - В отличие от матриц (компьютерное зрение) или фиксированных векторов (табличные данные)
- Данные существенно дискретны
  - К изображению кошки (матрице) можно добавить 0.01 и получить изображение кошки, с текстом так не сделать
- Неочевидно как перевести данные в числовой вид
- Разнородность данных:
  - Омонимы, синонимы
  - Опечатки
  - Словообразование (“превед”, “квантификация” и т.д.)
  - Разное написание:
    - “Привет”, “ПРИВЕТ!”, “ЗДАРОВА!!!!”, “ЗДАРОВА!!!!!!!!!!!!!1111)))”
  - и многое другое ...

Как представить текст в виде набора признаков

Варианты:

- Модель умеет работать с последовательностями
  - Рассмотрим в следующей лекции
- Модель не умеет работать с последовательностями
  - Рассмотрим сейчас

Как представить текст в виде набора признаков

Классический подход

- “Добро пожаловать в Нью-Йорк!”
- Токенизация
  - [“Добро”, “пожаловать”, “в”, “Нью-Йорк!”]
- Нормализация
  - [“добро”, “пожаловать”, “в”, “нью-йорк”]
- Перевод в числовой вид
  - [5, 1, 3, 7]
- Сведение к вектору
  - (0, 1, 0, 1, 1, 0, 1, 0, 0, ….0)

Токенизация

Разделение текста на слова (токены)

Сложности:

- [“Нью-Йорк”] вместо [“Нью”, “Йорк”]
- [“+7 (555) 512 2565”] или [TEL_NUMBER] вместо [“7”, “555”, “5122565”]
- [“П Р И В Е Т!”] вместо [“П”, “Р”, “И”, “В”, “Е”, “Т!”]
- [“20.11.2012”] вместо [“20”, “11”, “2012”]
- [“感谢”, ...] вместо [“谢谢你翻译这句话”]
  - В некоторых языках слова не разделяют пробелами
    - [“привет”, “как”, “дела”] вместо [“приветкакдела”] - хотя это касается и языков где так не делают

Токенизация

Обычно решают с помощью набора регулярных выражений и других эвристик (происходящих из лингвистики)

- Проще простого
  - Делим по пробелам\знакам препинания
- Просто
  - Делим с помощью нескольких регулярных выражений
- Посложнее
  - Делим с помощью специально описанных грамматик
    - Правила для городов
    - Телефонных номеров
    - Дат и других сущностей
    - Обработка знаков препинания
    - … и многое другое :-)

Токенизация

Обычно решают с помощью набора регулярных выражений\грамматик и других эвристик

- Проще простого
  - “Добро пожаловать в Нью-Йорк!”.split()
- Просто
  - nltk.tokenize.regexp
- Посложнее
  - используем NLTK/Spacy/другие специфичные токенизаторы

Важно: нет токенизатора, который обрабатывает все случаи

Сведение к вектору

Классический подход - “мешок слов” aka Bag of Words

- Нумеруем каждое слово

1. Абак
2. Абакан
3. Бурда
4. Волынка
5. …

9999999. Яхта

Мешок слов

Переводим список токенов в список чисел

- [“привет”, “как”, “дела”] -> [1678, 4390, 9054]
- `indices = [dictionary.get(word, UNK) for word in tokens]`

Создаем вектор - мешок слов:

```
vector = [0] * len(dictionary)
for k in indices:
  vector[k]+=1
```

(0, 0, 0, ….., 1, 0, 0, ….., 1, 0, 0, …, 1, 0, 0, 0 …)

Мешок слов

Пример: анализ тональности

- Применим логистическую регрессию к мешку слов
- `sigmoid((weights * vector).sum())`

Пример:

- “Это плохой фильм” -> (0, 0, …, 1, …., 1, …., 1, …, 0)
- `0.0001 * “это” – 1.5 * “плохой” + 0.0001 * “фильм” = -1.5 + 0.0002`
- “Это хороший фильм”
- `0.0001 * “это” + 1.2 * “хороший” + 0.0001 * “фильм” = 1.2002`
- Но что если “Это не плохой фильм”?

Мешок слов

Характеристики мешка слов:

- Не учитывает порядок слов
- Разряженный (много нулей) - можно использовать sparse представления и большие словари
- Каждое слово - новое измерение
- Сильно зависит от препроцессинга (“Привет” и “привет” это два разных слова или одно?)
- Чувствителен к опечаткам
- Не учитывает синонимов/омонимов

Стоп слова

Полезны ли слова, которые встречаются почти везде:

- “в”, “на”, “это”, “что”, “то”, …

Полезны ли слова, которые встречаются один раз среди всех примеров?

При использовании классического подхода (мешок слов) такие слова обычно удаляют

tf-idf

TF(w) = term frequency

- Сколько раз слово w встретилось в документе

DF(w)

- Сколько документов содержит слово w

N - количество документов

IDF(w)

- log (N / DF(w) + 1)

tf-idf = `TF(w) * IDF(w)`

tf-idf. Пример: поиск документа

Документы:

1. In information retrieval, **tf–idf** or **TFIDF**, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a **word** is to a document in a collection or corpus.[1] It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. The **tf–idf** value increases proportionally to the number of times a **word** appears in the document and is offset by the number of documents in the corpus that contain the **word**, which helps to adjust for the fact that some **words** appear more frequently in general. **tf–idf** is one of the most popular term-weighting schemes today; 83% of text-based recommender systems in digital libraries use **tf-idf**.[2]
2. The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its **words**, disregarding grammar and even **word** order but keeping multiplicity. The bag-of-words model has also been used for computer vision.[1] The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each **word** is used as a feature for training a classifier[2]. An early reference to "bag of **words**" in a linguistic context can be found in Zellig Harris's 1954 article on Distributional Structure.[3]

tf(tf-idf, 1) = 5 tf(word, 1) = 4 df(tf-idf) = 1 idf(tf-idf) = 1.1

tf(tf-idf, 2) = 0 tf(word, 2) = 4 df(word) =2 idf(word)= 0.69

tf-idf(tf-idf, 1) = 5 * 1.1 = 5.5 tf-idf(word, 1) = 4 * 0.69 = 2.76

tf-idf(tf-idf, 2) = 0 * 1.1 = 0 tf-idf(word, 2) = 4 * 0.69 = 2.76

Зачем нужен tf-idf

Документы:

1. tf_idf(..., 5, 4, ...) = (..., 5.5, 2.76, ...)
2. tf_idf(..., 0, 4, ...) = (..., 0, 2.76, …)

Идея tf-idf:

- Увеличить веса специфичных слов (встречающихся в малом числе документов)
- Уменьшить вес слов встречающихся во многих документах

Применение tf-idf к мешку слов помогает в разных задачах

Стемминг

Нахождение “корня”слова с помощью эвристических правил и алгоритмов.

Пример: суффиксный стеммер

(.*)ed -> \1

(.*)ely -> \1

(.*)ing -> \1

supervised, supervisely, supervising -> supervis

Лемматизация

Приведение слова к словарной (нормальной) форме.

Используются морфологические анализаторы (определяется часть речи, делается морфологический разбор)

Работают медленнее чем алгоритмы стемминга, дают более корректные результаты с точки зрения лингвистики

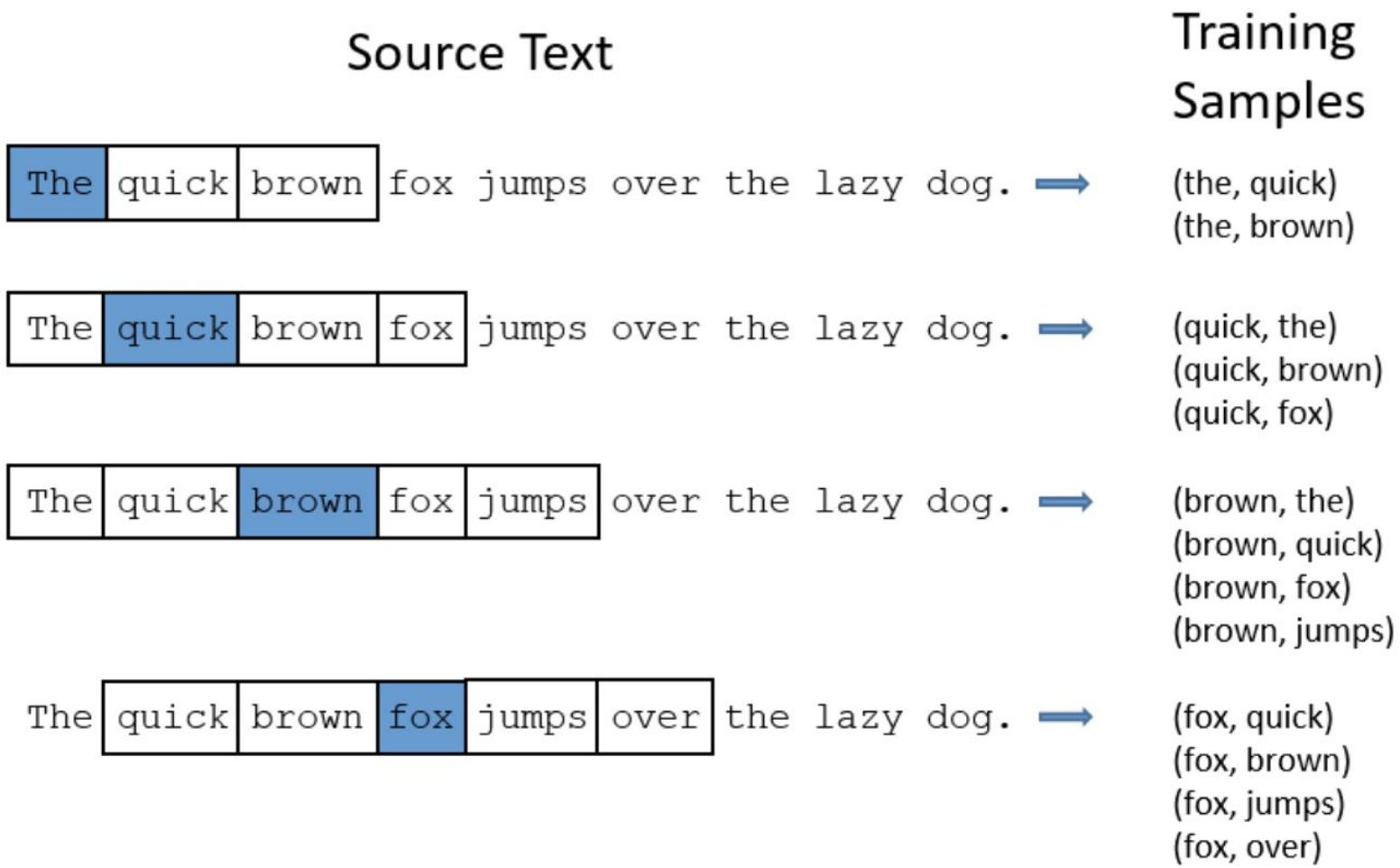
Стемминг и лемматизация - способы “нормализации” слов

Как учитывать связи и порядок слов?

- N-gram - последовательность из n идущих подряд слов в тексте
  - униграммы (n=1), биграммы (n=2), триграммы (n=3)
- K-skip-N-gram - последовательность из n идущих подряд слов в тексте, причём расстояние между соседними должно составлять не более k токенов

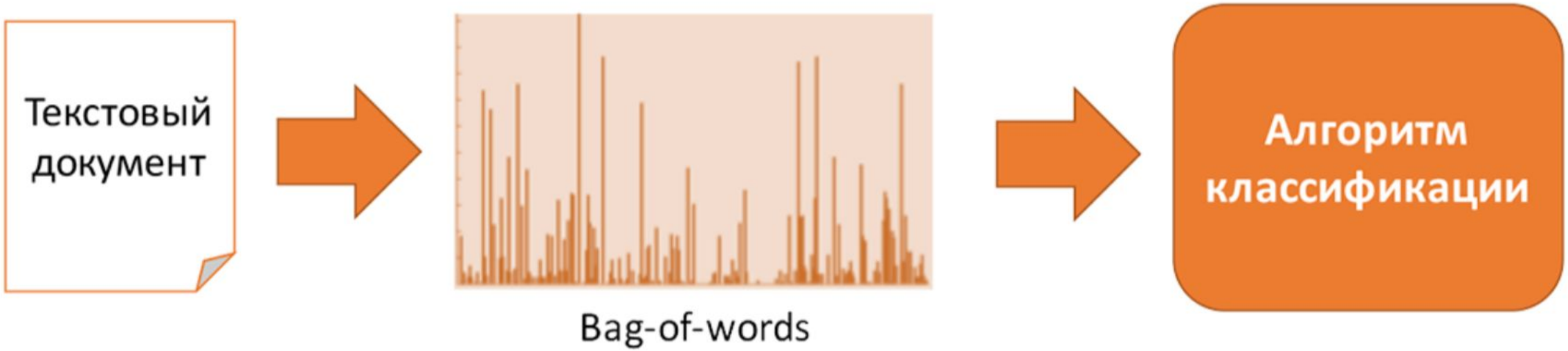
- **Пример: ”Набор подряд идущих токенов”**
- 2-gram: набор подряд, подряд идущих, идущих токенов
- 1-skip-2-gram: набор идущих, подряд токенов

1-skip-2-gram


Source Text
Training Samples
The quick brown fox jumps over the lazy dog.
(the, quick)
(the, brown)
The quick brown fox jumps over the lazy dog.
(quick, the)
(quick, brown)
(quick, fox)
The quick brown fox jumps over the lazy dog.
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)
The quick brown fox jumps over the lazy dog.
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)

Bag of words pipeline

- Препроцессинг текста
- Bag-of-words на словах и N-grammax как векторное представление текста
- Linear models, SVD, Random forest и т.д. для классификации



Word2Vec

- Другой способ векторного представления
- То, что называют ”pretrained word embeddings”
- Статья издана 7 сентября 2013 (Google research)

Word2Vec

Основная идея: дистрибутивная гипотеза

- Слова встречающихся в схожих контекстах имеют близкие значения

Задача:

- предсказать слово по его контексту (соседним словам)

или

- предсказать контекст по слову

Для обучения обычно используют большой корпус текста (википедия, common crawl …)

Word2Vec skip-gram negative sampling

Задача: по слову предсказать его контекст
Обучающая выборка:
“Предсказать по слову его контекст”

```
(“по”, “слову”, 1)
(“его”, “слову”, 1)
(“предсказать”, “слову”, 0)
(“контекст”, “слову”, 0)
(“выборка”, “слову”, 0)
```

Положительные пример: выбираем из текста
Отрицательный пример: ставим в пару случайное слово из текста

Word2Vec skip-gram negative sampling

Задача: по слову предсказать его контекст
Модель:

```
vocab_size = 100_000
emb_size = 200
context_embedding = Linear(vocab_size, emb_size)
word_embedding = Linear(vocab_size, emb_size)
output = Linear(emb_size, 1)

def forward(word, context):
   embd_ctx = context_embedding(context)
   embd_word= word_embedding(word)
   return sigmoid(output(embd_ctx * embd_word))
```

Word2Vec skip-gram negative sampling

Задача: по слову предсказать его контекст
Обучение:

```
for word, context, label in dataset:
   pred = model(word, context)
   loss = binary_cross_entropy(pred, label)
   loss.backward()
   optimizer.step()
   model.zero_grad()
```

Получение векторов:

```
vectors = [word_embedding(word) for word in vocab]
```

Word2Vec

Skip-gram

- Сэмплирование плохих примеров в negative sampling можно базировать на встречаемости слов
- Вместо negative sampling, можно честно считать softmax и минимизировать негативное лог. правдоподобие (неэффективно для больших словарей)
- Можно использовать иерархический softmax

CBOW

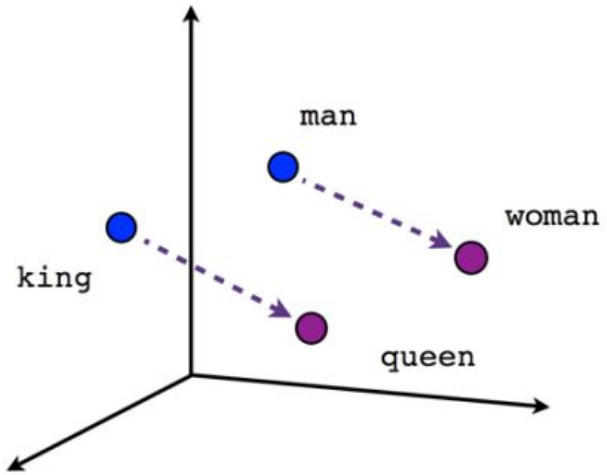
- Вместо предсказания контекста по слову, можно предсказывать слово по контексту

Word2Vec

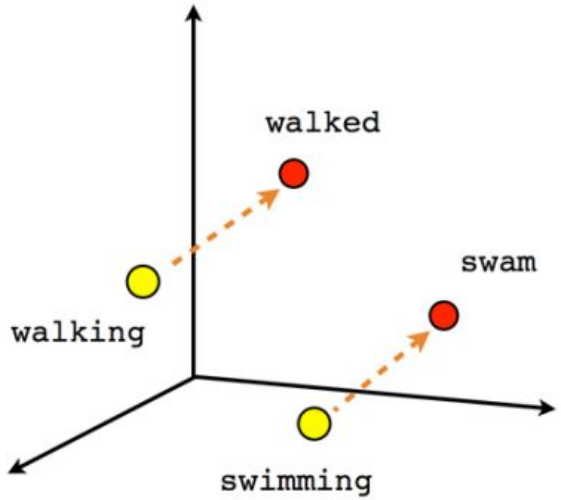
- Получаем “dense” представление слов как векторов небольшой размерности (300-1000)
- Вектора обучают на больших неразмеченных наборах текстов
  - После чего их можно использовать в своих задачах
- Для получения вектора примера берут среднее векторов слов
- Или используют WMD (word mover distance)

Интересное свойство word2vec

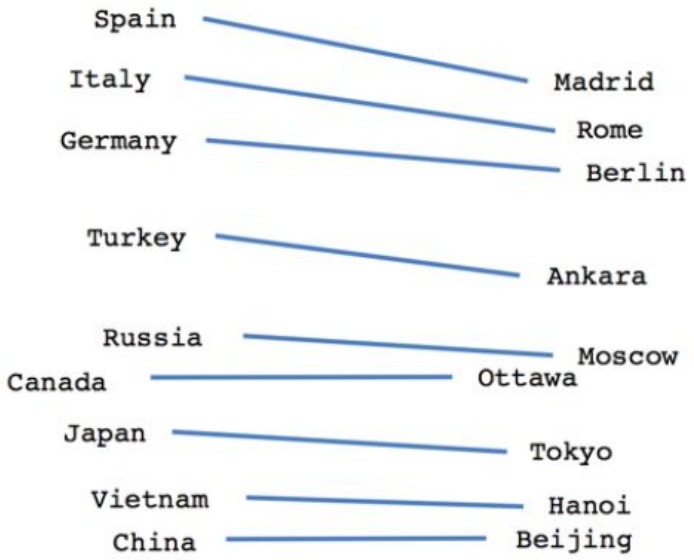
- King - Man + Woman = Queen
- Paris - France + Italy = Rome



Male-Female

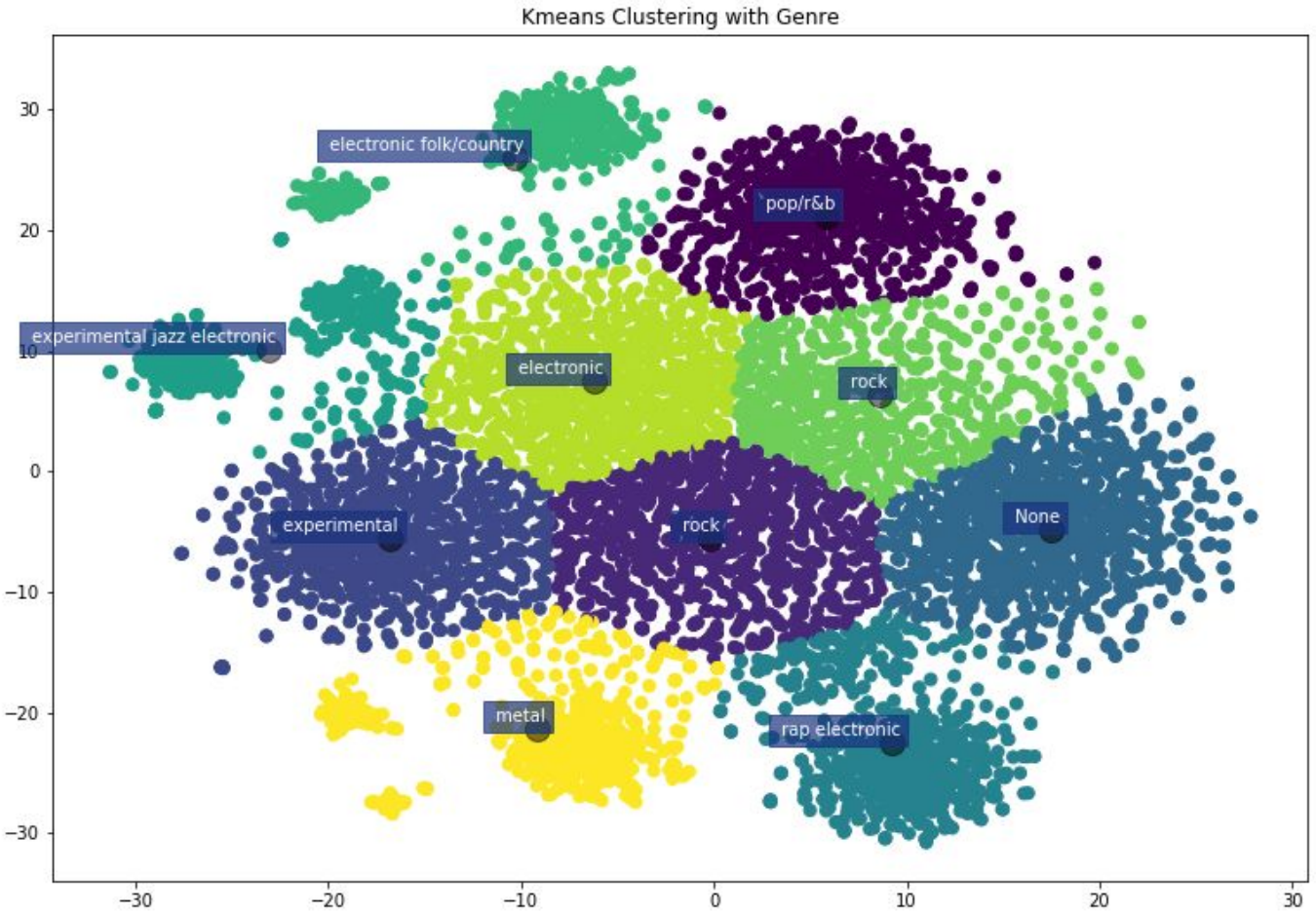


Verb tense

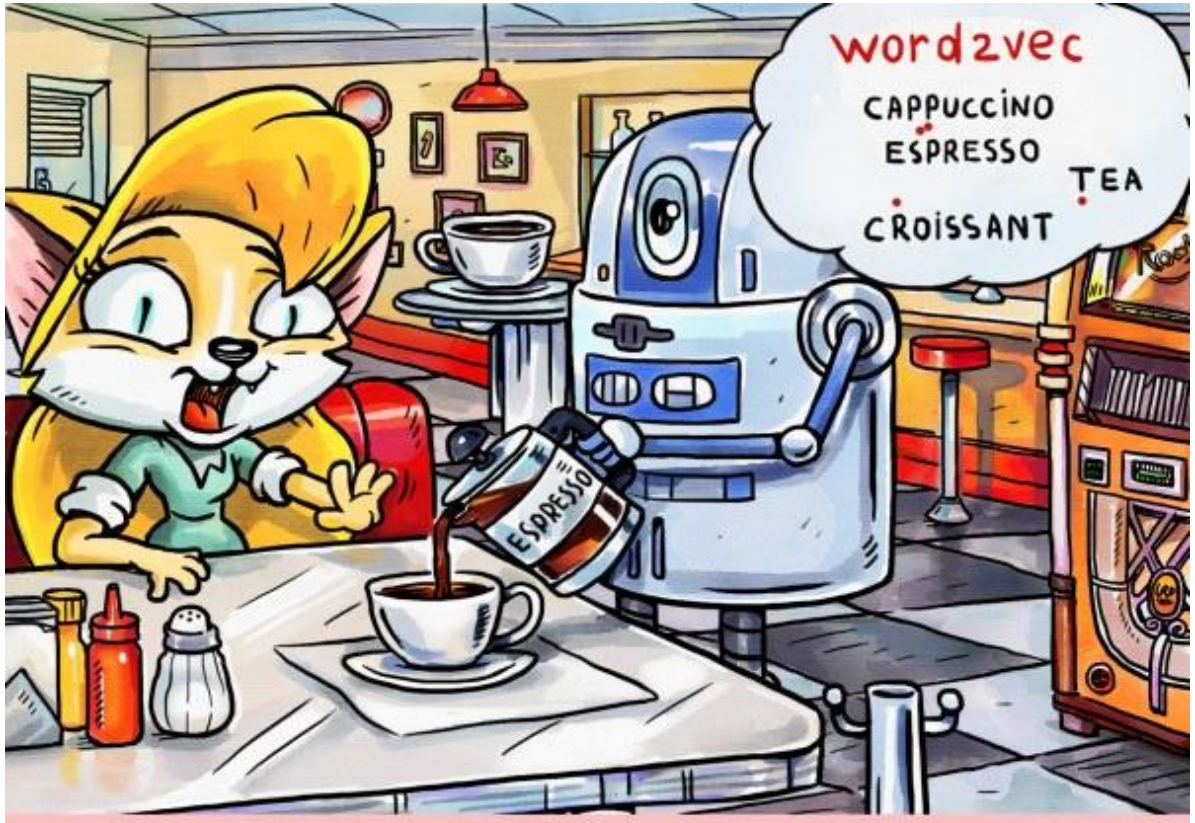


Country-Capital

TSNE на word2vec



Интересное свойство word2vec



- Espresso? But I ordered a cappuccino!
- Don't worry, the cosine distance between them is so small that they are almost the same thing.

Близость векторов слов не означает что слова - синонимы, и тем более что они синонимы в вашей конкретной задаче

Word2Vec как распределенное представление

Пример т.н. “распределенного представления”

- Объект (слово) преобразуется в вектор в некотором пространстве
- Отношения между объектами - похожесть векторов

Другой пример:

- поиск похожих изображений
  - Евклидово\косинусное расстояние между эмбеддингами CNN
  - Методы:
    - Сиамские сети
    - arcface

Расширения word2vec

- Алгоритм word2vec модифицируют, чтобы :
  - Решать проблему омонимов
  - Давать контекстно-зависимые вектора
  - Давать устойчивость к опечаткам и новым словоформам
  - и т.д.
- Советую посмотреть: ELMO
- Сейчас рассмотрим FastText

Идея word2vec

Идея word2vec оказалась полезной не только для слов:

- Для документов
- Для графов
- В рекомендательных системах (последовательности просмотров\покупок)
- ...

FastText

Модификация алгоритма word2vec, устойчивая к опечаткам и новым словоформам

- Обучаются вектора слов и вектора sub-words (3-грамм)
- В случае, если слово есть в словаре используется его вектор
- Иначе вектор слова есть среднее векторов его триграмм

Позволяет обойтись без специального токена UNK для неизвестных слов

В некоторых задачах позволяет избежать добавления модели исправления опечаток

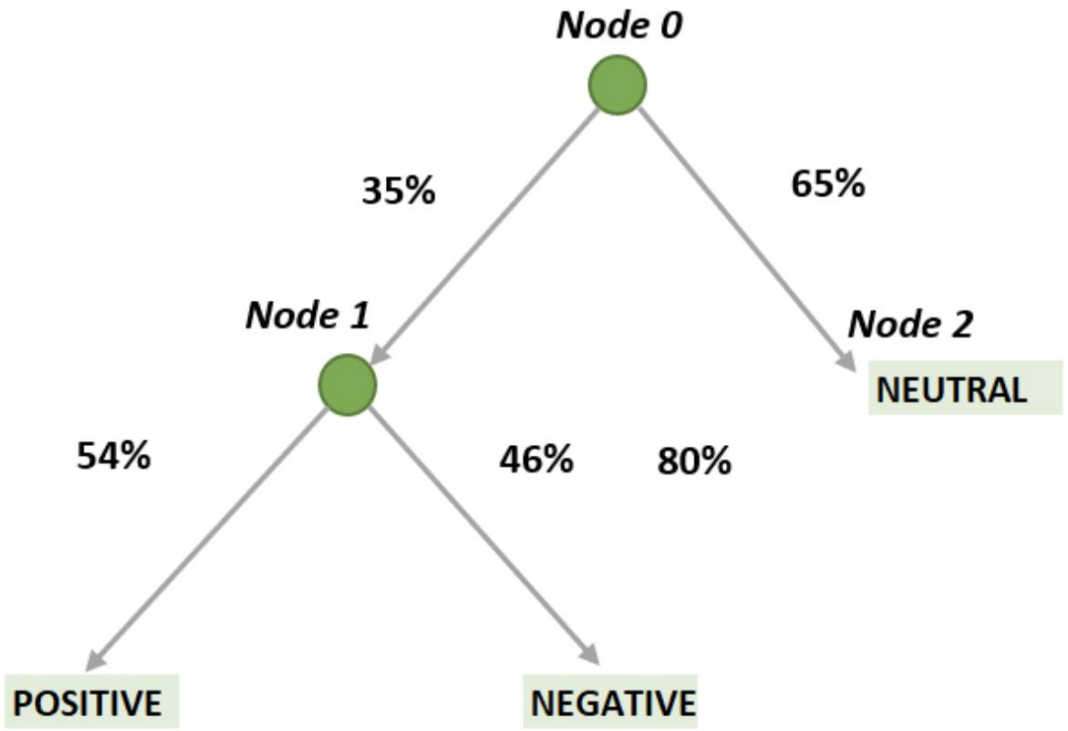
Классификация текстов с FastText

Классический подход, который был до FastText:

- Используем softmax на последнем слое
- Размер последнего слоя раве числу классов

Подход в FastText:

- Используем аналог решающи деревьев для определения класса



Классификация текстов с FastText

- Получаем значительный прирост в скорости обучения и тестирования
- Почти не теряем в точности

	Zhang and LeCun (2015)		Conneau et al. (2016)			fastText
	small char-CNN	big char-CNN	depth=9	depth=17	depth=29	$h = 10$, bigram
AG	1h	3h	24m	37m	51m	1s
Sogou	-	-	25m	41m	56m	7s
DBpedia	2h	5h	27m	44m	1h	2s
Yelp P.	-	-	28m	43m	1h09	3s
Yelp F.	-	-	29m	45m	1h12	4s
Yah. A.	8h	1d	1h	1h33	2h	5s
Amz. F.	2d	5d	2h45	4h20	7h	9s
Amz. P.	2d	5d	2h45	4h25	7h	10s

Table 2: Training time for a single epoch on sentiment analysis datasets compared to char-CNN and VDCNN.

Классификация текстов с FastText

- Получаем значительный прирост в скорости обучения и тестирования
- Почти не теряем в точности

Model	AG	Sogou	DBP	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
BoW (Zhang et al., 2015)	88.8	92.9	96.6	92.2	58.0	68.9	54.6	90.4
ngrams (Zhang et al., 2015)	92.0	97.1	98.6	95.6	56.3	68.5	54.3	92.0
ngrams TFIDF (Zhang et al., 2015)	92.4	97.2	98.7	95.4	54.8	68.5	52.4	91.5
char-CNN (Zhang and LeCun, 2015)	87.2	95.1	98.3	94.7	62.0	71.2	59.5	94.5
char-CRNN (Xiao and Cho, 2016)	91.4	95.2	98.6	94.5	61.8	71.7	59.2	94.1
VDCNN (Conneau et al., 2016)	91.3	96.8	98.7	95.7	64.7	73.4	63.0	95.7
`fastText`, $h = 10$	91.5	93.9	98.1	93.8	60.4	72.0	55.8	91.2
`fastText`, $h = 10$, bigram	92.5	96.8	98.6	95.7	63.9	72.3	60.2	94.6

Table 1: Test accuracy [%] on sentiment datasets. `FastText` has been run with the same parameters for all the datasets. It has 10 hidden units and we evaluate it with and without bigrams. For char-CNN, we show the best reported numbers without data augmentation.

Языковая модель (Language model)

Языковая модель это модель, которая определяет вероятность того, что набор слов может образовывать последовательность с учётом порядка:

$$P(W) = P(w_1, w_2, w_3, \ldots, w_n)$$

Наиболее частое применение - оценка вероятности для следующего слова:

$$P(w_5 | w_1, w_2, w_3, w_4)$$

Языковая модель (Language model)

Наиболее частое применение - оценка вероятности для следующего слова:

$$P(w_5|w_1, w_2, w_3, w_4)$$

Таким образом можно генерировать целые тексты, как “с нуля”, так и уже имея какой-то набор слов в качестве начала.

Языковая модель (Language model)

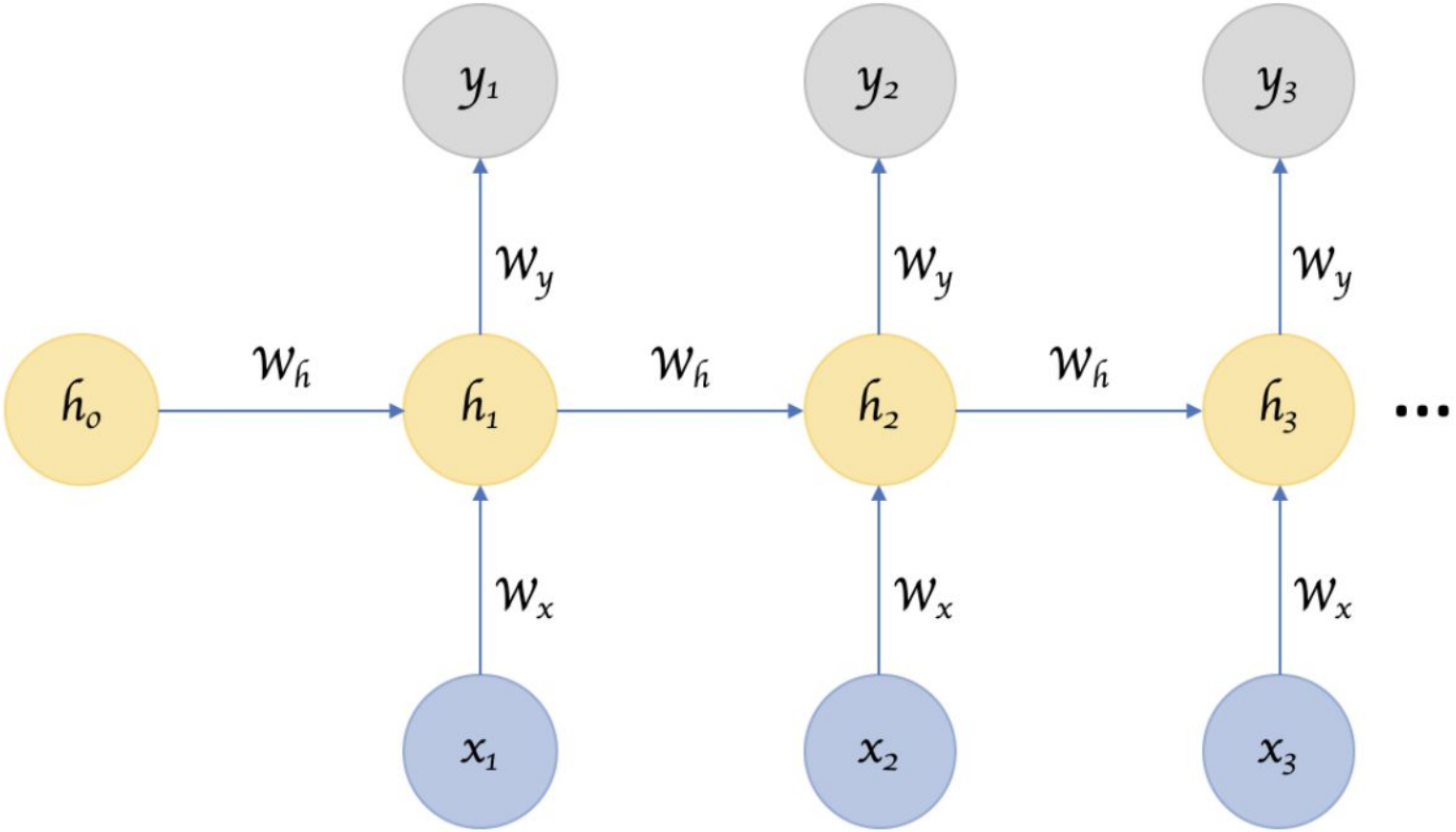
- Как реализовать такую модель?

- Она должна уметь поддерживать в некотором виде текущий контекст, чтобы на его основе предсказывать следующее слово
- При добавлении нового слова контекст должен меняться
- Контекст может быть очень “длинным” (допустим, основываться на 100 последних токенах)
- Не все из последних токенов действительно нужны

- Выход - рекуррентные нейронные сети

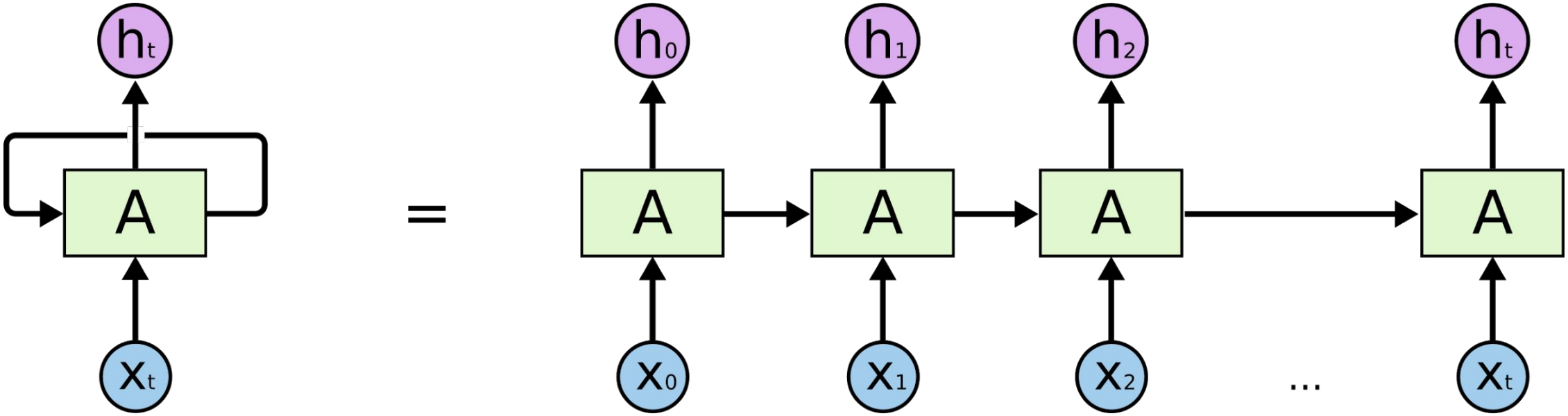
Recurrent Neural Networks (RNN)

- Размер входной последовательност и фиксирован
- Очень сложная структура
- Очень много параметров для обучения
- **Это ещё не RNN**



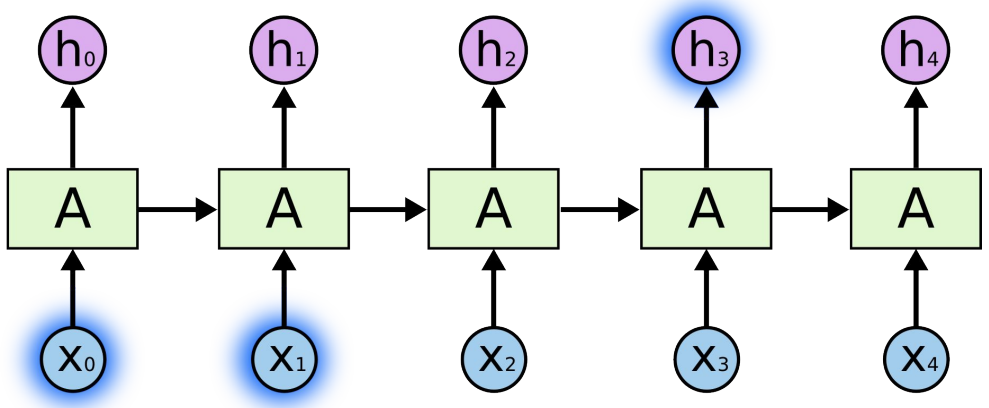
Recurrent Neural Networks (RNN)

- Нет жёстких ограничений на длину входной последовательности
- Выход каждого слоя подаем на вход следующему
- Количество пробросов ограничено (обычно около 80)
- Градиент пробрасывается по всей цепочке (BPTT)

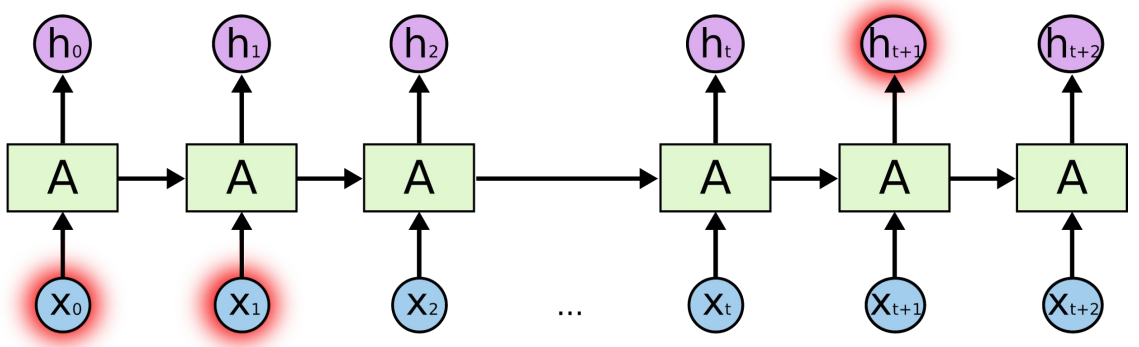


Проблемы RNN

- The clouds are in the *sky*

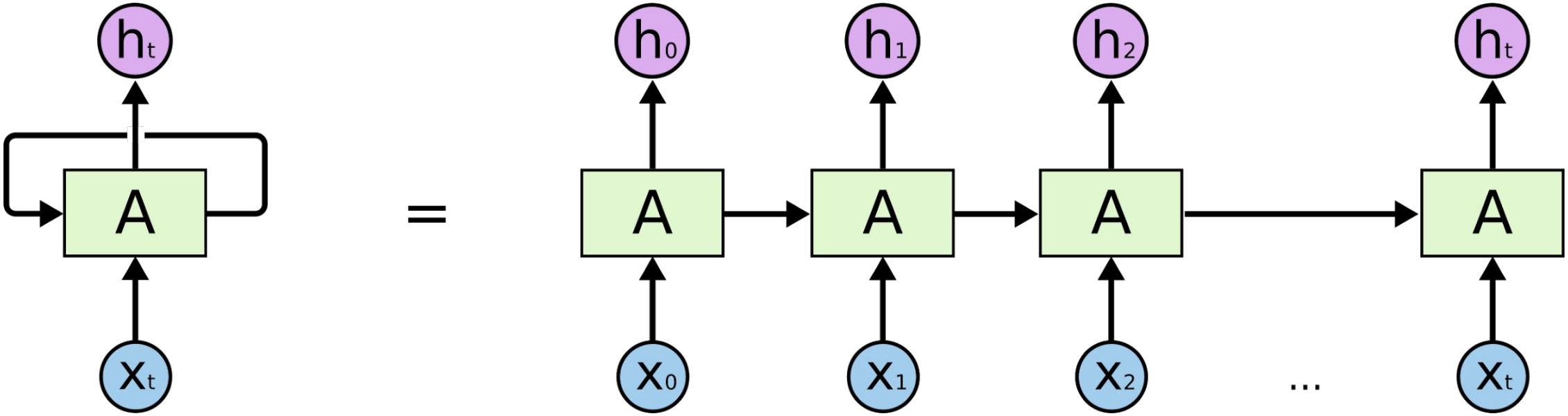


- I grew up in France… I speak fluent *French*.



Проблемы RNN

- На практике глубина проброса должна быть достаточной, чтобы поддерживать смысл текста
- Слишком большая глубина проброса приводит к затуханию градиента, а значит к трудностям при обучении



Проблема “затухания” градиента в RNN

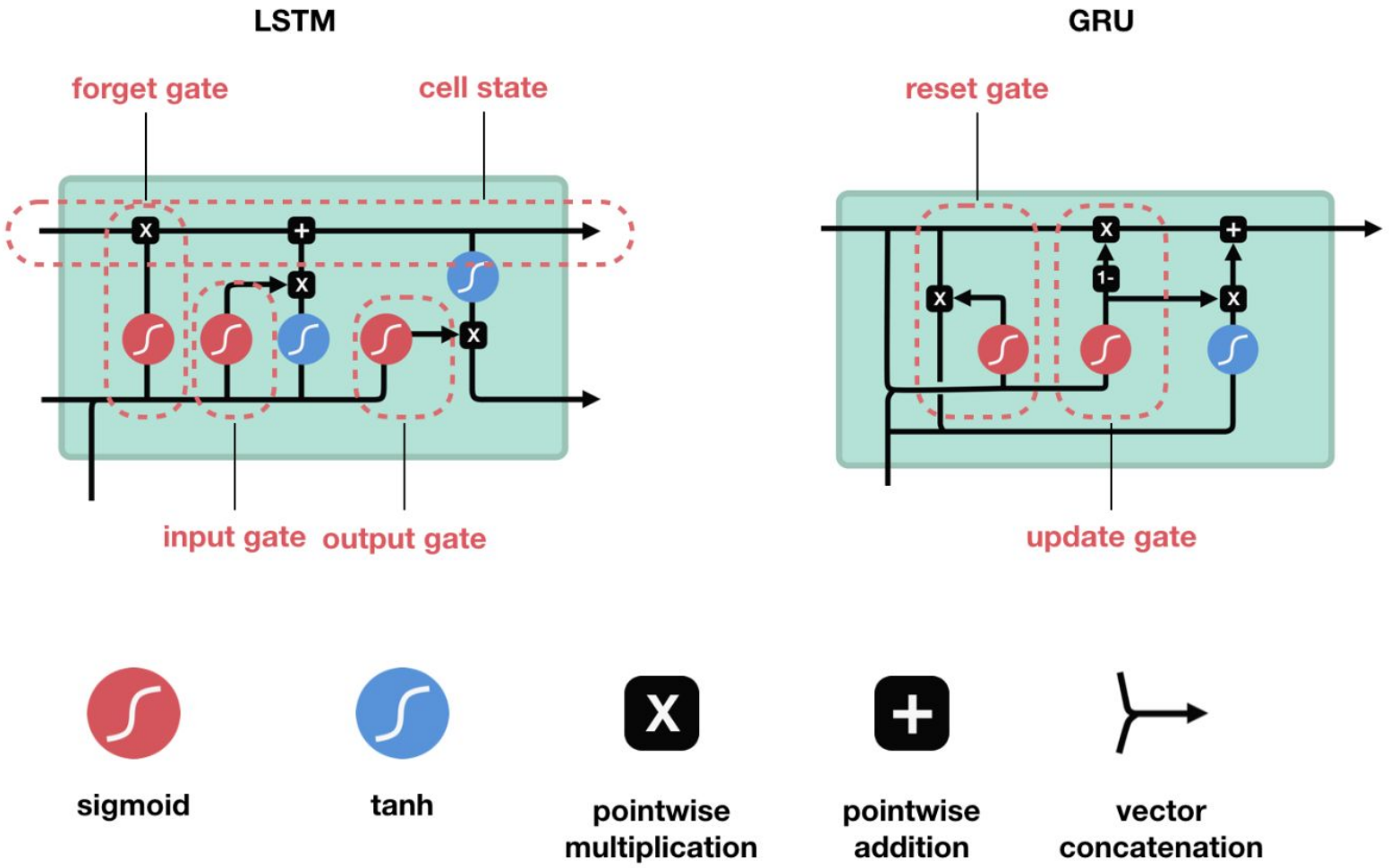
- Последние входные токены в общем случае оказывают большее влияние, чем более “старые”
- При длине последовательности в 80 токенов, первый токен будет оказывать на ответ минимальное воздействие
- А значит градиент для весов на первом шаге будет также минимальным или “размытым”
- То есть сеть и дальше не будет учиться запоминать длинные зависимости
- Это проблема в RNN также называется short-term memory problem (проблема кратковременной памяти)

Проблема “взрывающегося” градиента в RNN

- Аналогично “затуханию”, градиент может “взрываться”
- Проблема решается простым ограничением сверху на величину градиента (gradient norm clipping)

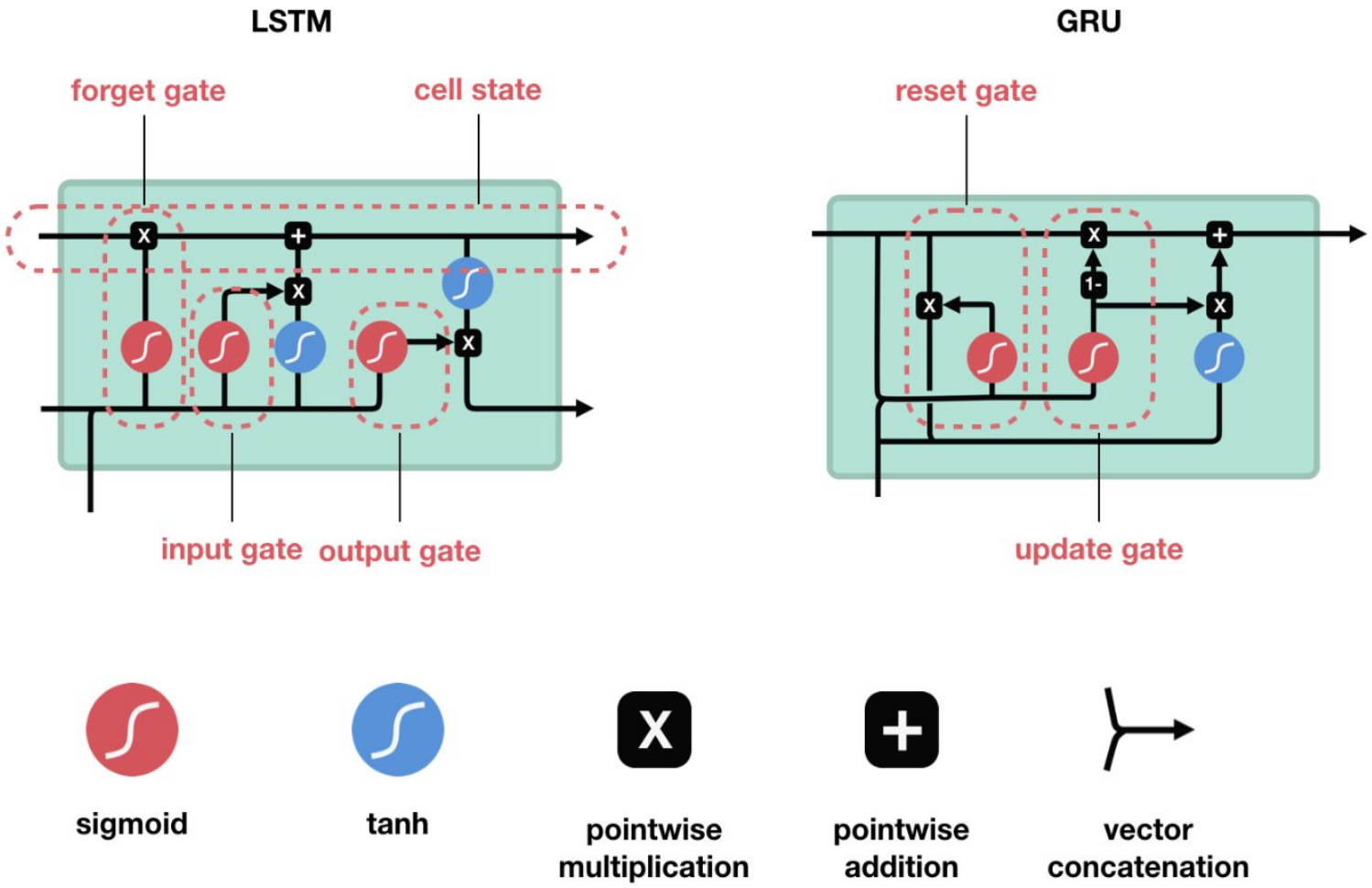
Решение: Gated Recurrent Neural Networks

- LSTM - Long Short-Term Memory
- GRU - Gated Recurrent Units



LSTM и GRU

- Основное нововведение - сигнал “памяти” или forget gate
- Выбирает, какую информацию забыть, а какую запомнить



Как работает forget gate?

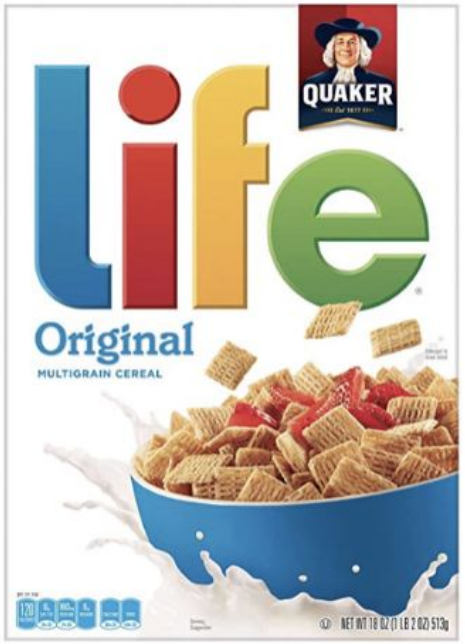
Customers Review 2,491

Thanos

September 2018

Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!



A Box of Cereal

$3.99

Как работает forget gate?

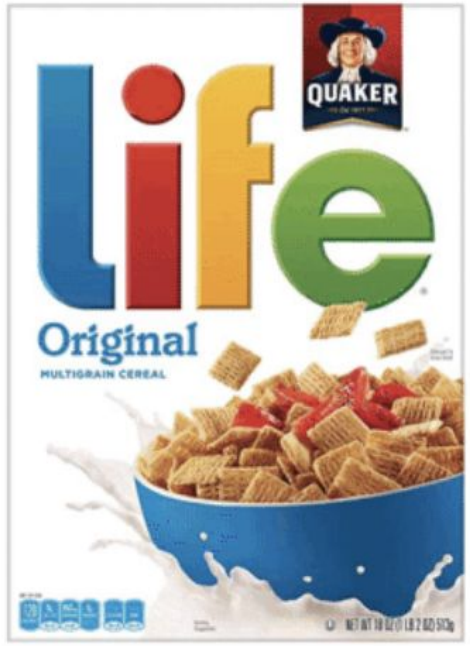
Customers Review 2,491

Thanos

September 2018

Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!



A Box of Cereal

$3.99

Проблемы классического LSTM

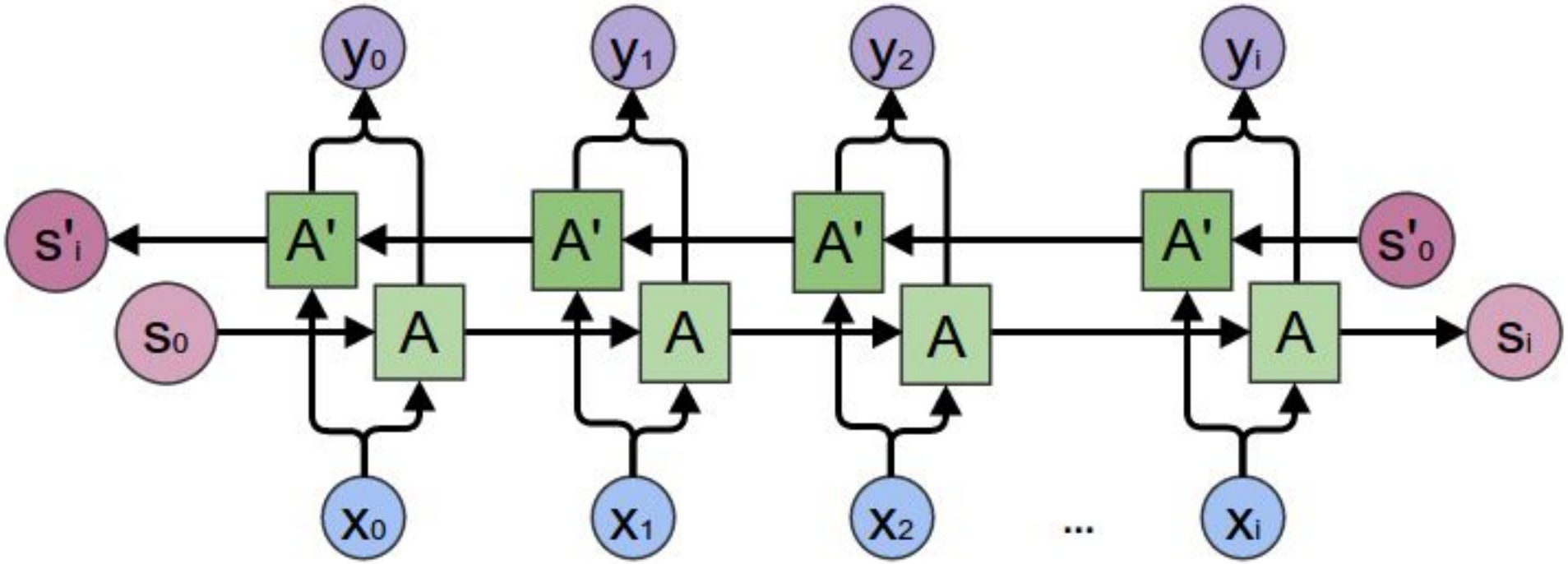
Двустронние зависимости:

I was born in Russia so I’m a native speaker of *Russian*.

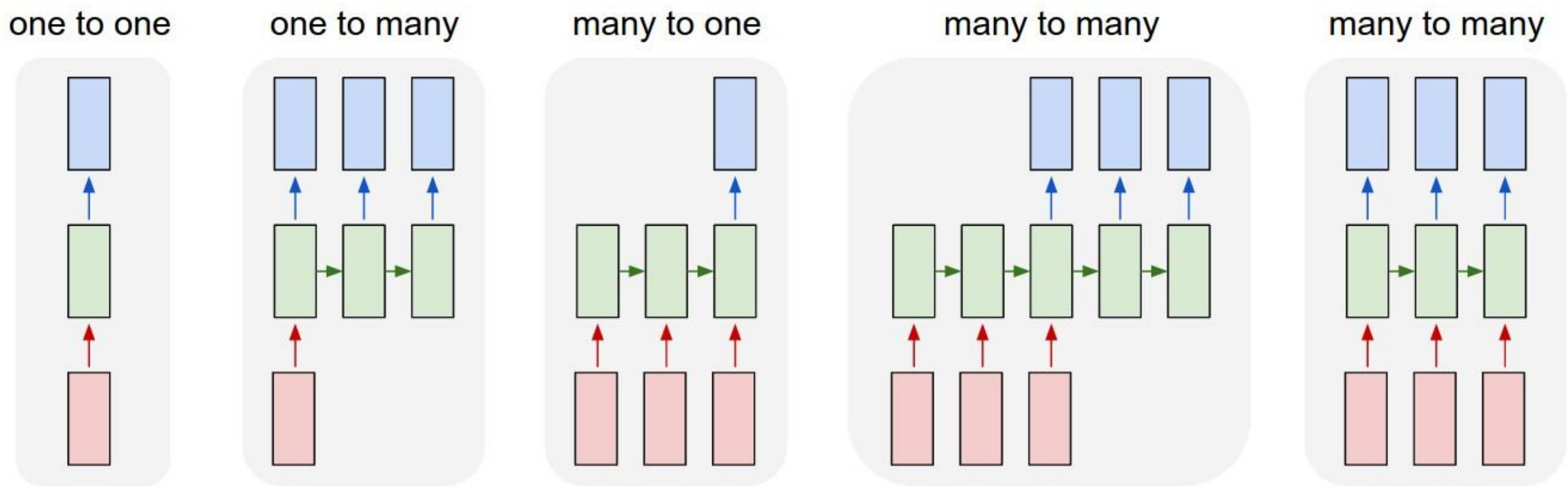
I’m a native speaker of *Russian* because I was born in Russia.

Двусторонний LSTM (BiLSTM)

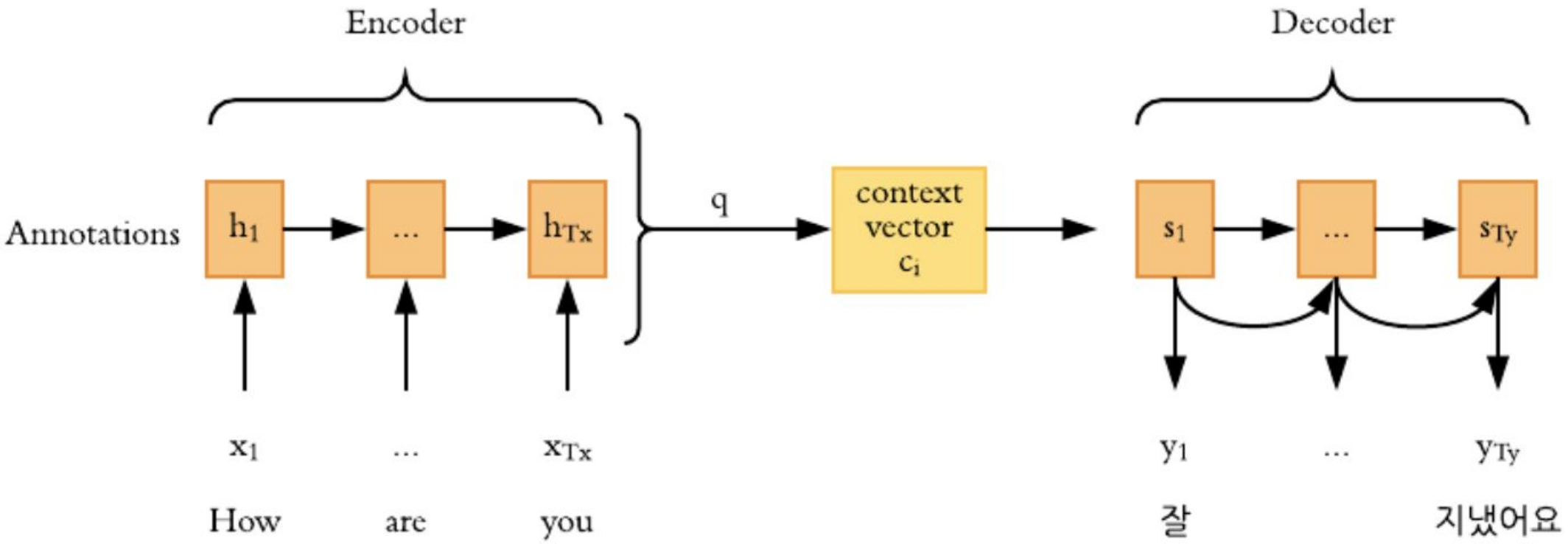
- Учитываем не только предыдущий контекст, но и будущий:



Виды архитектуры рекуррентных сетей

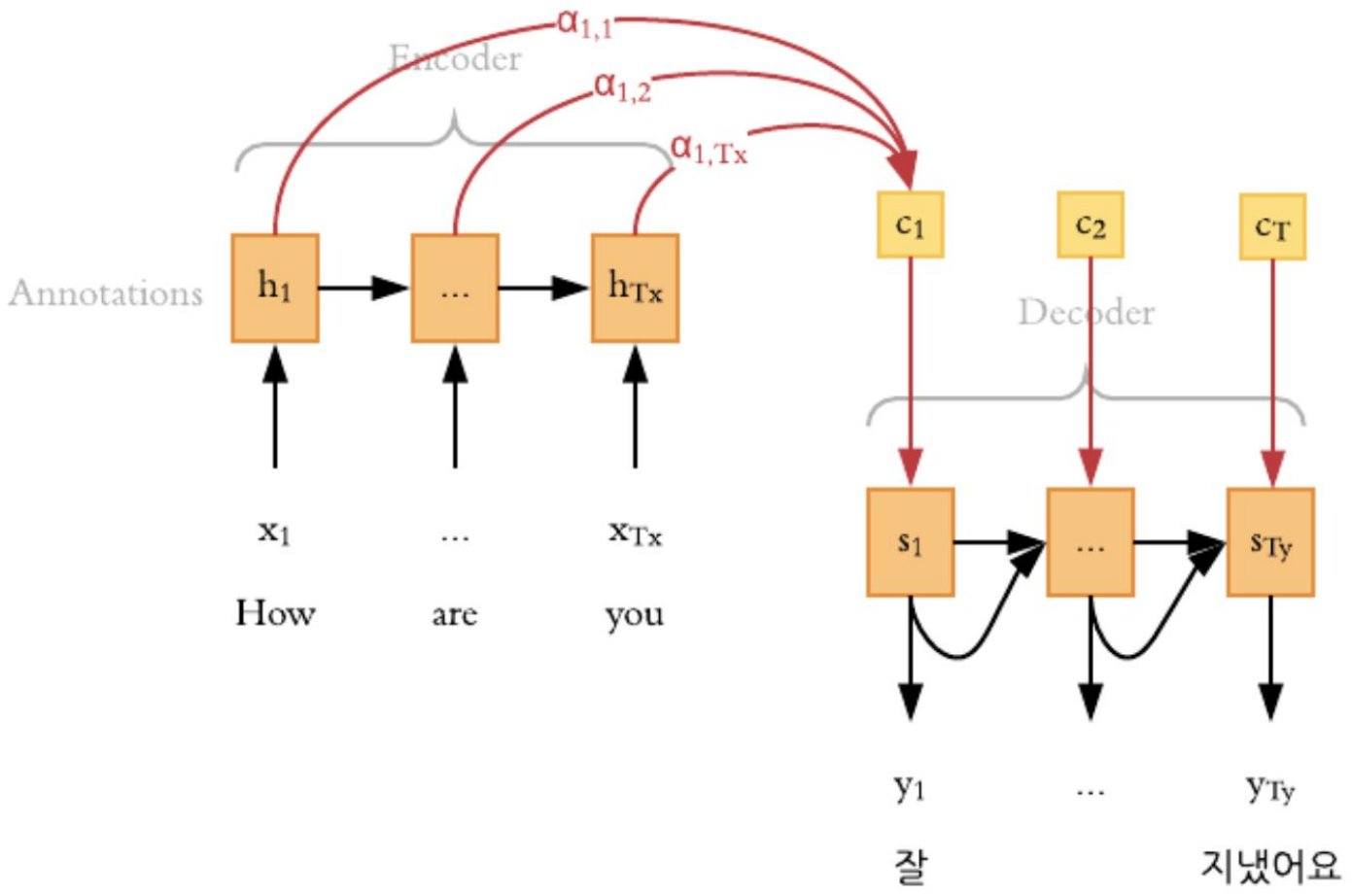


Подход encoder - decoder



Attention

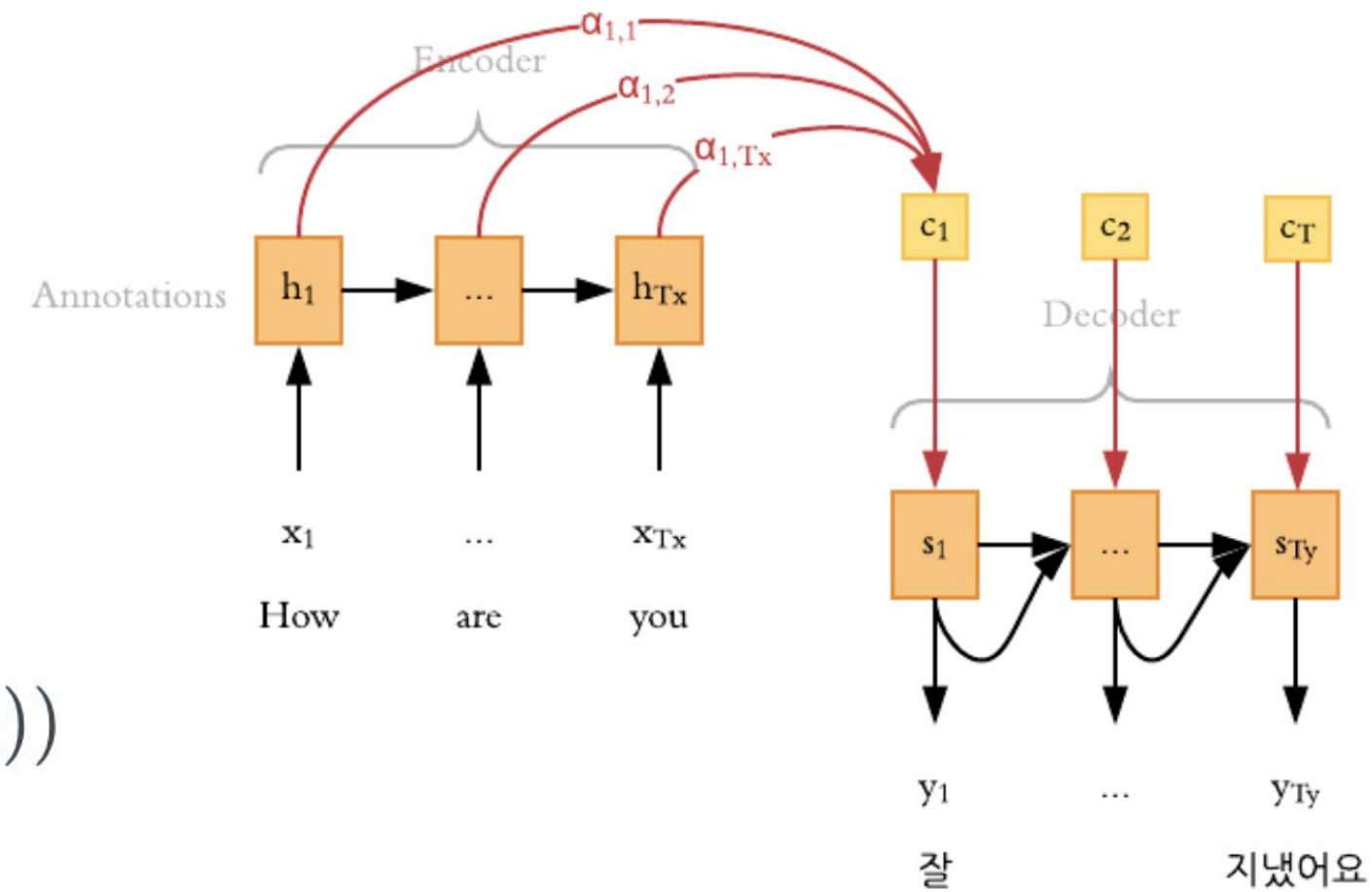
- Каждое выходное слово будет иметь свой вектор, который будет менять важность отдельных компонентов context вектора
- То есть для каждого выходного слова будем обращать внимание (attention) только на некоторые входные слова



Attention

$$\mathbf{c}_i = \sum_j a_{ij}\mathbf{s}_j$$

$$\mathbf{a}_i = \mathrm{softmax}(f_{att}(\mathbf{h}_i, \mathbf{s}_j))$$



Виды attention

- Basic dot-product attention:

$$e_i = q^T k_i \in \mathbb{R}$$

where $d_1 = d_2$

- Multiplicative attention:

$$e_i = q^T W k_i \in \mathbb{R}$$

where $W \in \mathbb{R}^{d_2 x d_1}$ is a weight matrix

- Additive attention:

$$e_i = {W_3}^T \tanh(W_1 k_i + W_2 q) \in \mathbb{R}$$

where $W_1 \in \mathbb{R}^{d_3 x d_1}$, $W_2 \in \mathbb{R}^{d_3 x d_2}$ are weight matrices and $W_3 \in \mathbb{R}^{d_3}$ is a weight vector.

BERT - Bidirectional Encoder Representations from Transformers

- Предложен Google в 2018 году
- Основан на архитектуре языковой модели Transformer, также предложенной Google
- Сразу показал state-of-the-art результаты на большинстве NLP задач
- Помимо прочего позволяет легко делать transfer learning, чего не было раньше
- В данный момент используется практически везде в NLP благодаря простоте применения
- “Свергнут” в феврале 2019 GPT-2 от OpenAI, которая, по словам авторовов, настолько хорошо генерирует тексты, что pretrained модель решили пока не представлять миру

Несколько слов о препроцессинге

- Стемминг/лемматизация, удаление стоп-слов требуются только для классических методов (до word2vec включительно)
- Подходы с использованием RNN требуют добавления различных токенов, вроде начала и конца предложения
- BERT в простейшем случае требует только добавления различным токенов

- **Другими словами, каждый метод требует свой подход к препроцессингу**