Тверской государственный технический университет

Интеллектуальные информационные системы

Архитектуры НС для работы с последовательностями

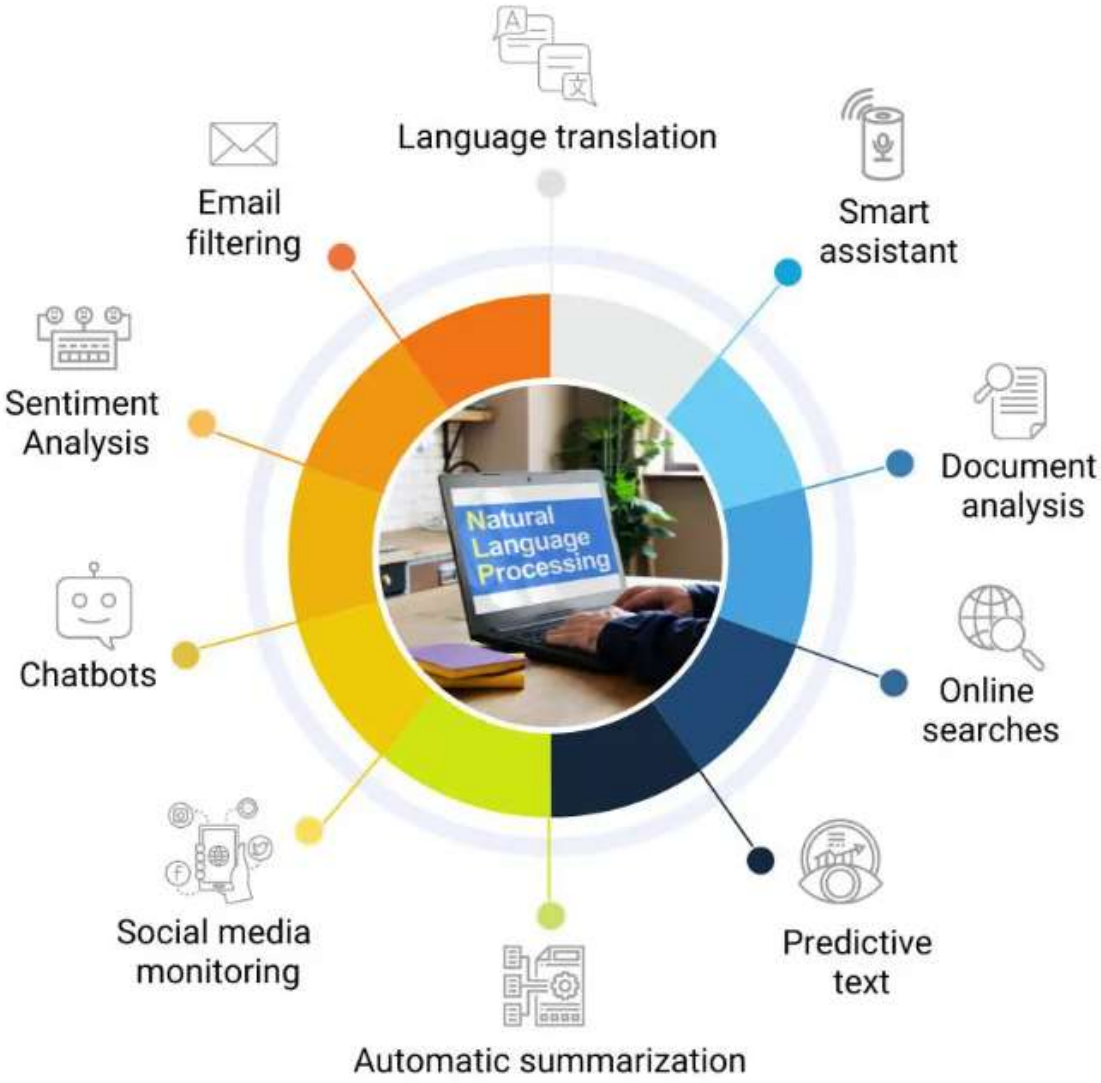
2025 г.

Задачи обработки последовательностей

Natural Language Processing (NLP) – анализ текстов, написанных на естественных языках:

- Классификация текста, анализ тональности документов, поиск релевантных документов по запросу и их ранжирование
- Синтез и распознавание речи
- Машинный перевод
- Генерация текста
- Диалоговые системы и чат-боты
- Суммаризация обращений

Applications of
Natural Language Processing

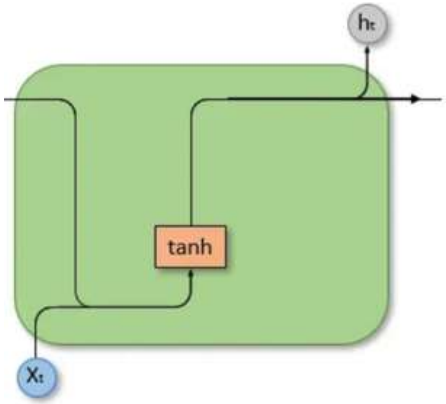


Задачи обработки последовательностей

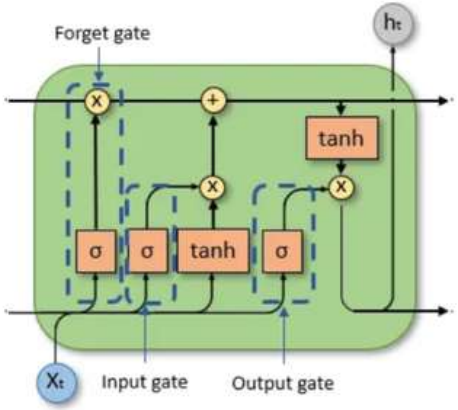
Временные ряды – значения меняющихся во времени признаков, полученные в некоторые моменты времени

- Прогнозирование временных рядов как самостоятельная задача – прогнозирование продаж, спроса, трафика, стоимости и множество других
- Обработка сигналов
- Поиск аномалий
- Интерпретация генома и задачи биоинформатики

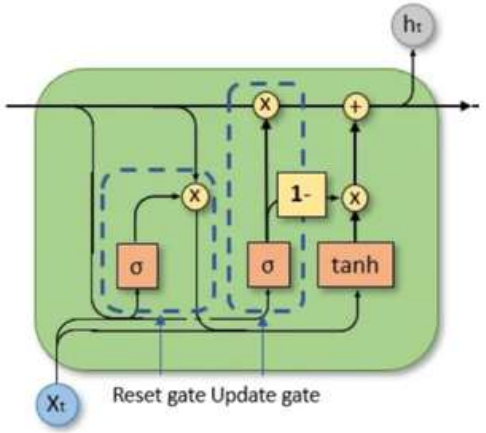
RNN



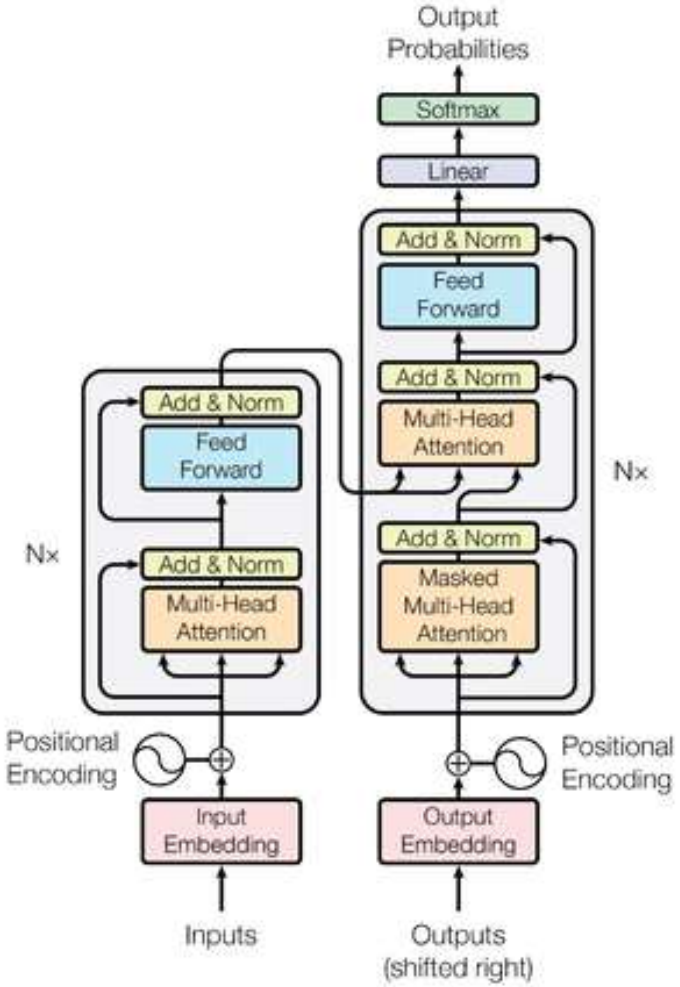
LSTM



GRU

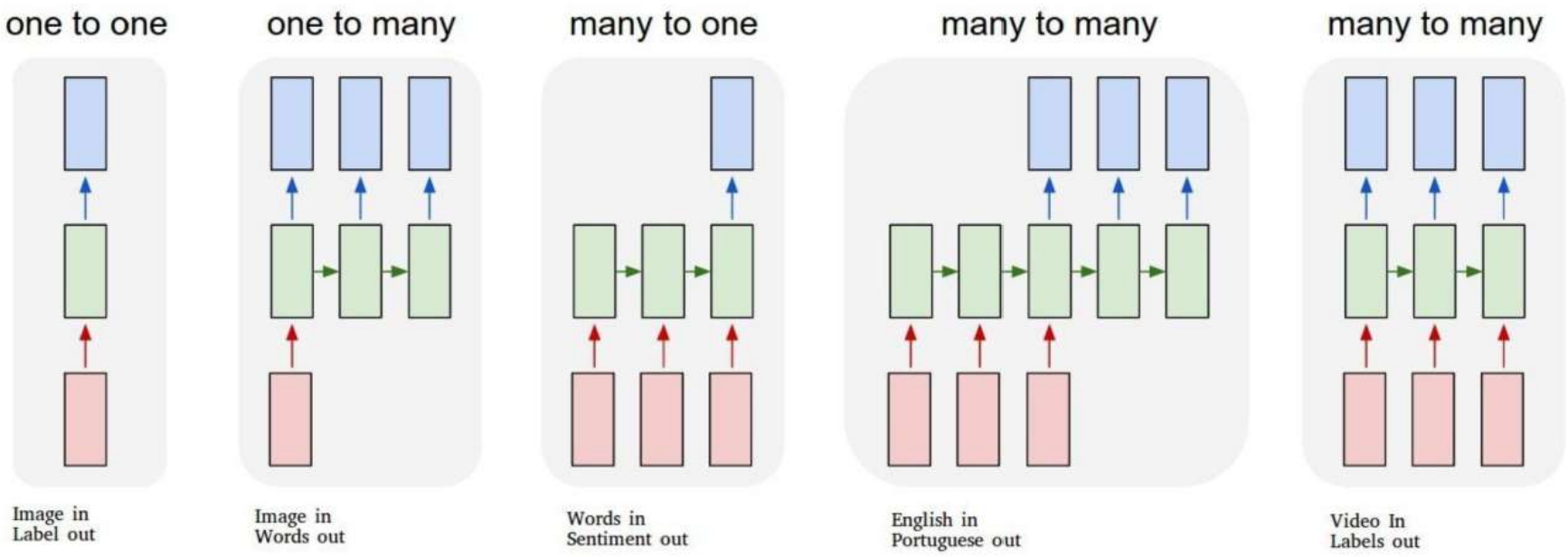


Transformers



Способы работы с последовательностями:

1. One-to-one
2. One to many
3. Many to one
4. Many to many (синхронизированный или нет)



Источник: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Many-to-one

Было сложно, мне понравилось

↓

Позитивный

One-to-many

→ Пингвин в шляпе

Many-to-many, не синхронизированный

I seem to have overfitted

↓

Кажется, я переобучился

Many-to-many, синхронизированный

Будет	сложно,	вам	понравится
↓	↓	↓	↓
глагол	наречие	местоимение	глагол

Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

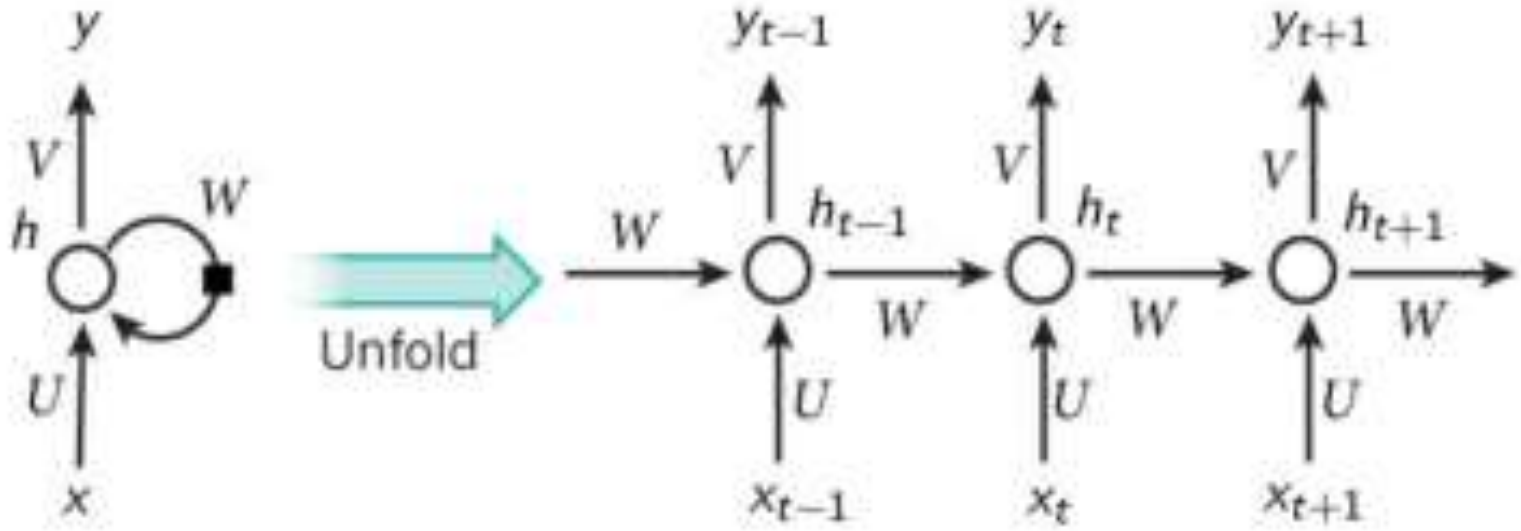
Задача обработки последовательностей

x_t - входной вектор в момент t

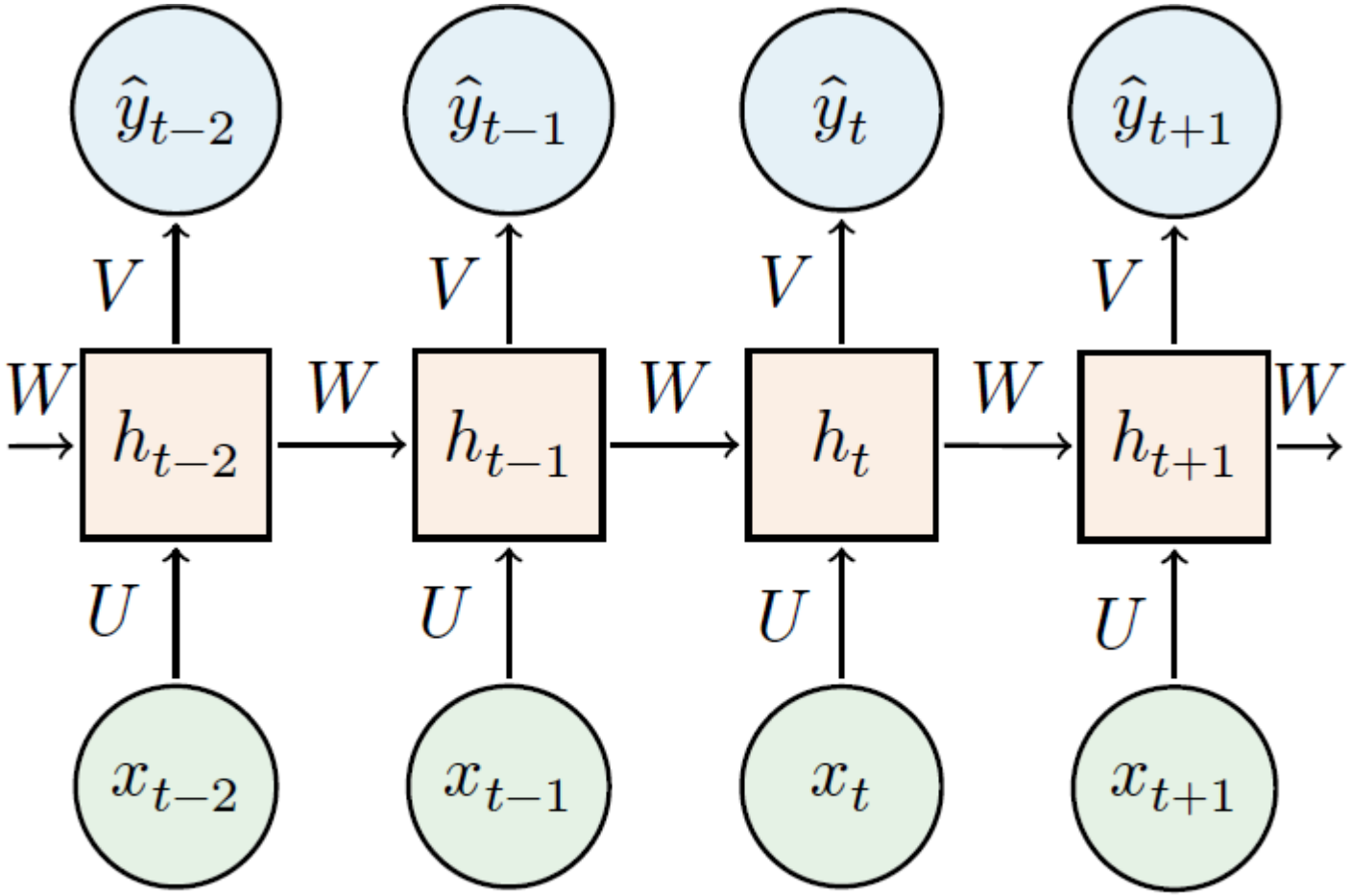
h_t - вектор скрытого состояния в момент t

y_t - выходной вектор

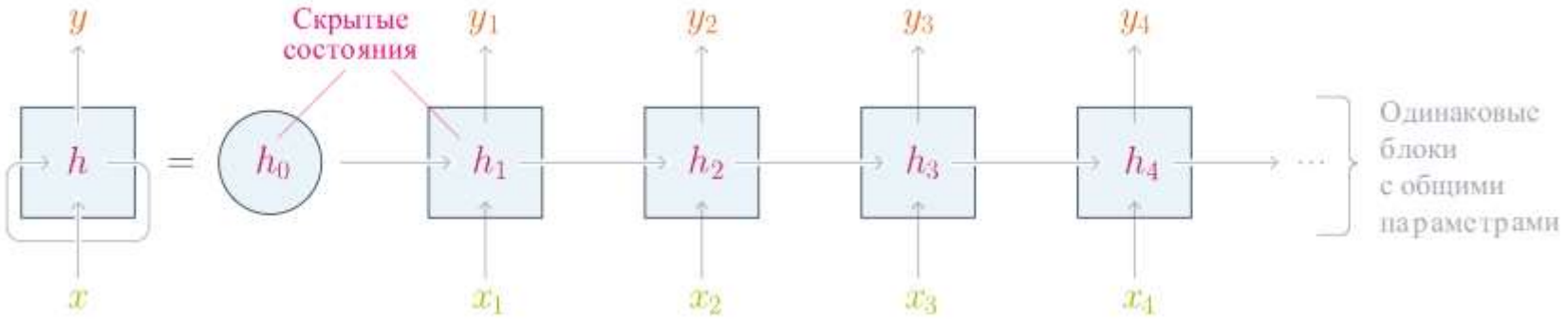
$$h_t = \sigma_h\left(U_{x_t} + W_{h_{t-1}}\right)$$
$$y_t = \sigma_y(V_{h_t})$$



Рекуррентную сеть можно рассматривать, как несколько копий одной и той же сети, каждая из которых передает информацию последующей копии



$$h_t = f_h(Ux_t + Wh_{t-1} + b_h) \qquad \hat{y}_t = f_y(Vh_t + b_y)$$



h_t - вектор скрытого состояния в момент t – «внутренняя память» - для хранения информации о предыдущих элементах последовательности. На каждом дискретном шаге в сеть подаются данные, при этом происходит обновление скрытого состояния:

$h_t = \tanh(h_{t-1}W_1 + x_tW_2)$

По скрытому состоянию предсказывается выходной сигнал:

$$y_t = h_tW_3$$

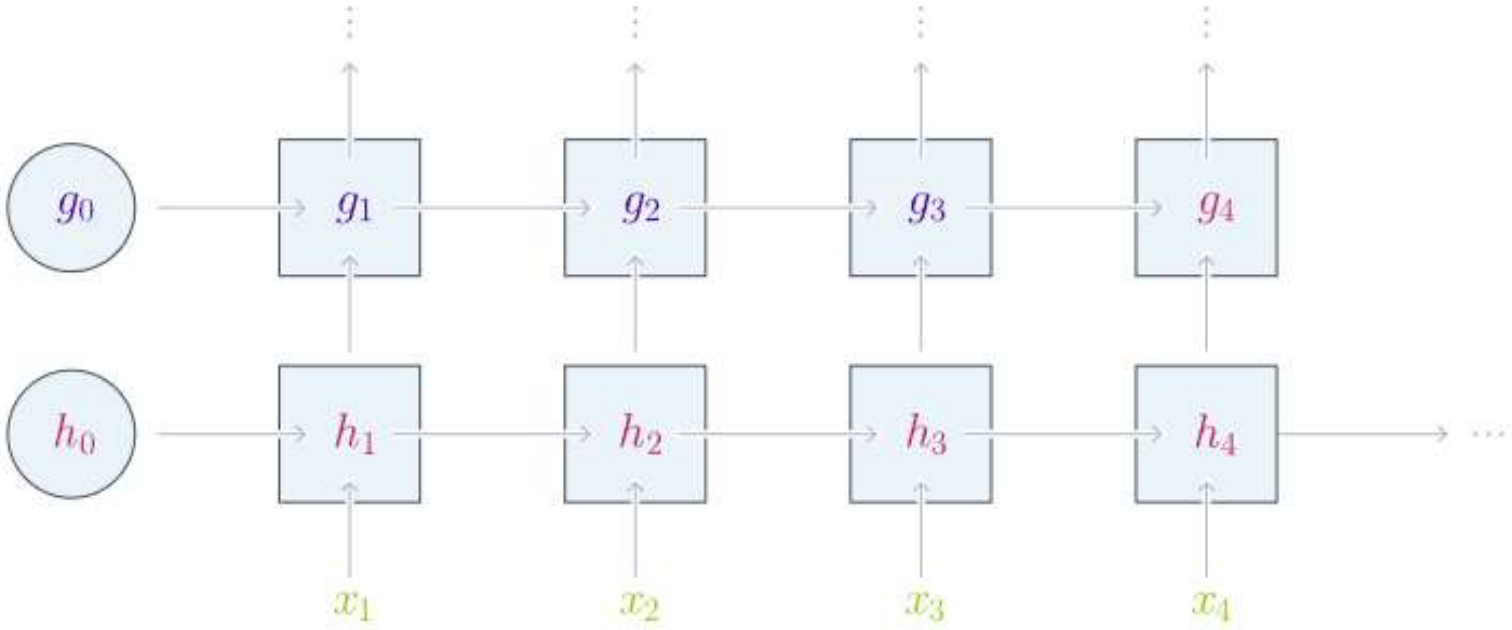
Веса W_i одинаковых на всех итерациях, то есть очередные x_t и h_{t-1} подаются на вход одного и того же слоя, зацикленного на себе

Функция потерь – суммарное отклонение по всем выходным сигналам y_t:

$$\sum_{t=0}^{T} \mathscr{L}_t(U, V, W) \to \min_{U,V,W}$$

$$\mathscr{L}_t(U, V, W) = \mathscr{L}\big(y_t(U, V, W)\big) - \text{потеря от предсказания } y_t$$

Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami



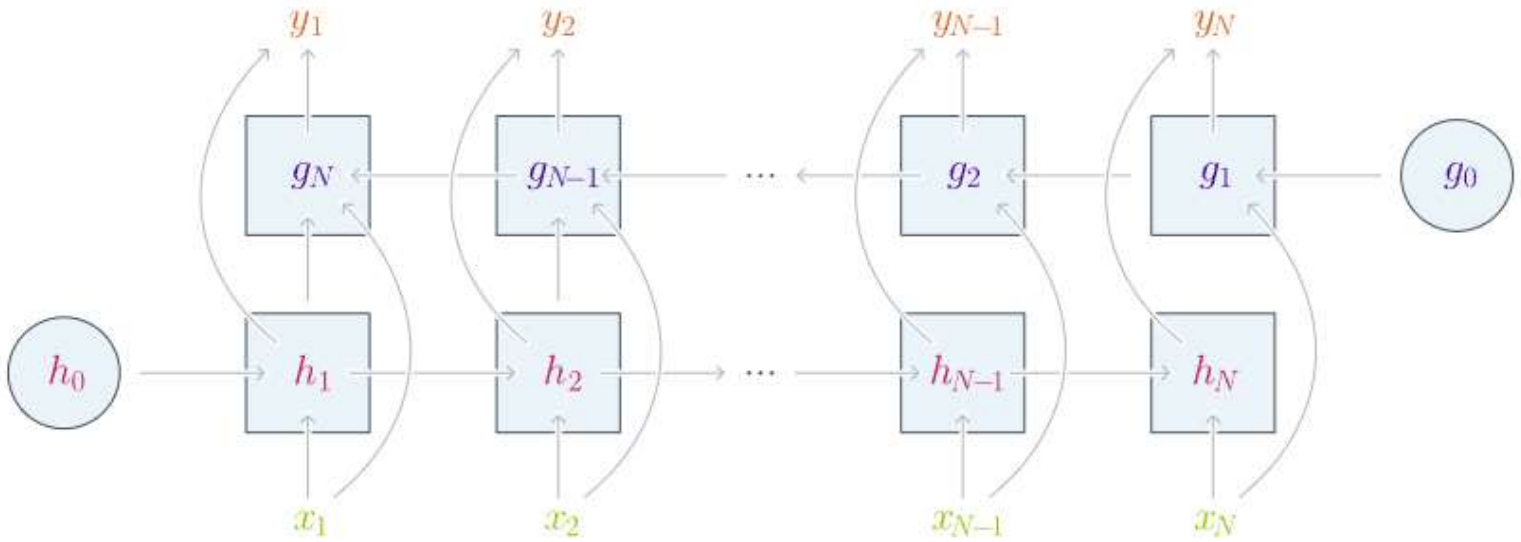
Глубокая RNN — несколько рекуррентных слоев:

- Первый слой RNN (первая сеть) принимает на вход исходную последовательность
- Второй слой RNN – принимает выходы первой сети
- Третий слой RNN – принимает выходы второй сети
- И так далее

Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

Bidirectional RNN

Стандартная RNN учитывает только предыдущий контекст. Но, например, слово в предложении связано не только с предыдущими, но и с последующими словами. Для таких случаев используется двунаправленная рекуррентная сеть (BRNN) – состоящая из прямой (элементы подаются от первого к последнему) и обратной (элементы подаются в обратном порядке) рекуррентных сетей.



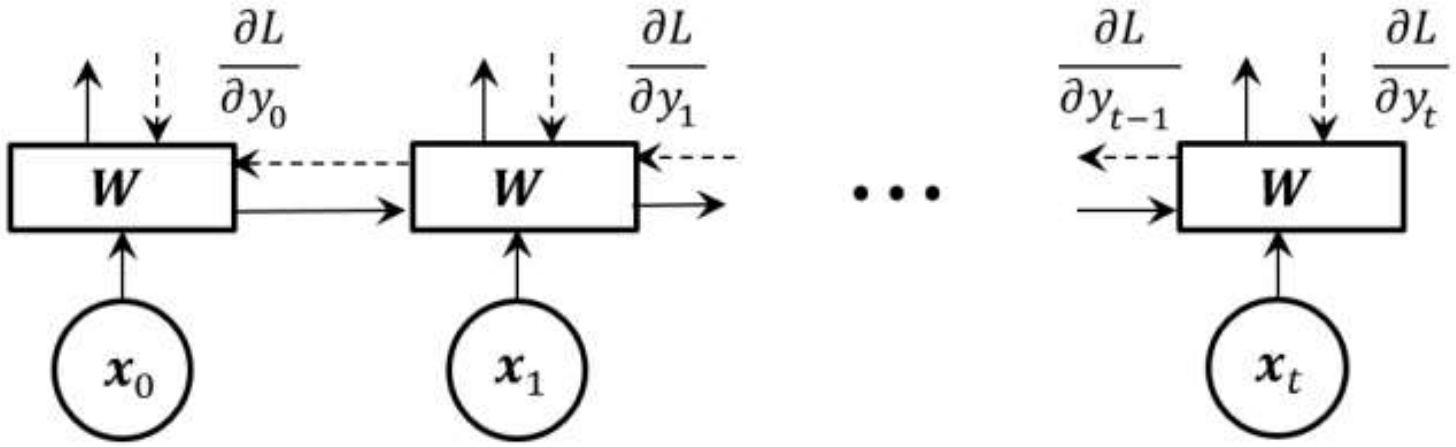
$$y_n = [h_n, g_n]\, w + b$$

Конкатенация

Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

Backpropagation Through Time (BPTT)

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\sum_{k=0}^{t}\left(\prod_{i=k+1}^{t}\frac{\partial h_i}{\partial h_{i-1}}\right)\frac{\partial h_k}{\partial W}$$



Для предотвращения затухания/взрыва градиентов:

частные производные должны стремиться к 1: $\frac{\partial h_i}{\partial h_{i-1}} \to 1$

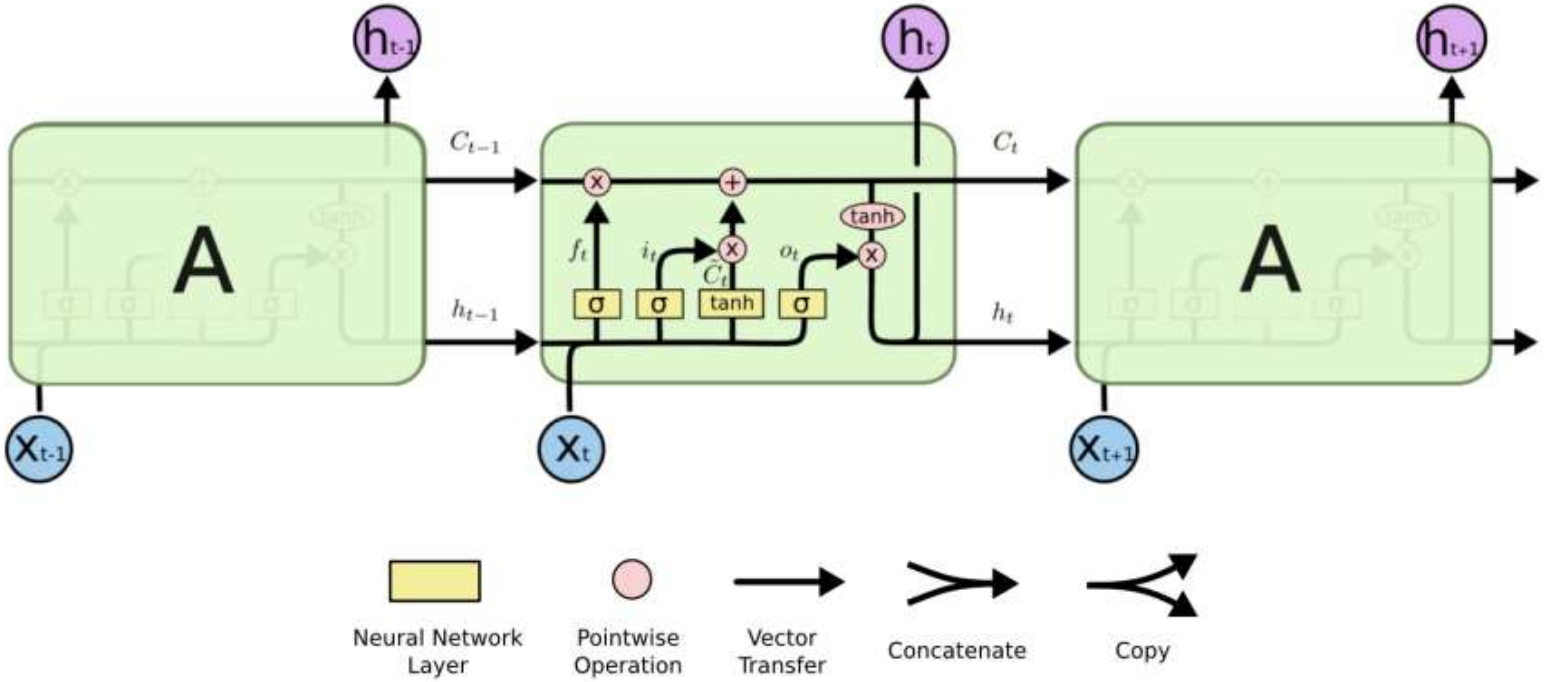
Достигается за счет выбора функций активации

Источник: http://www.machinelearning.ru/wiki/images/7/71/Voron-ML-DeepLearning-slides.pdf

Сети долгой кратковременной памяти
Long Short-term memory (LSTM)

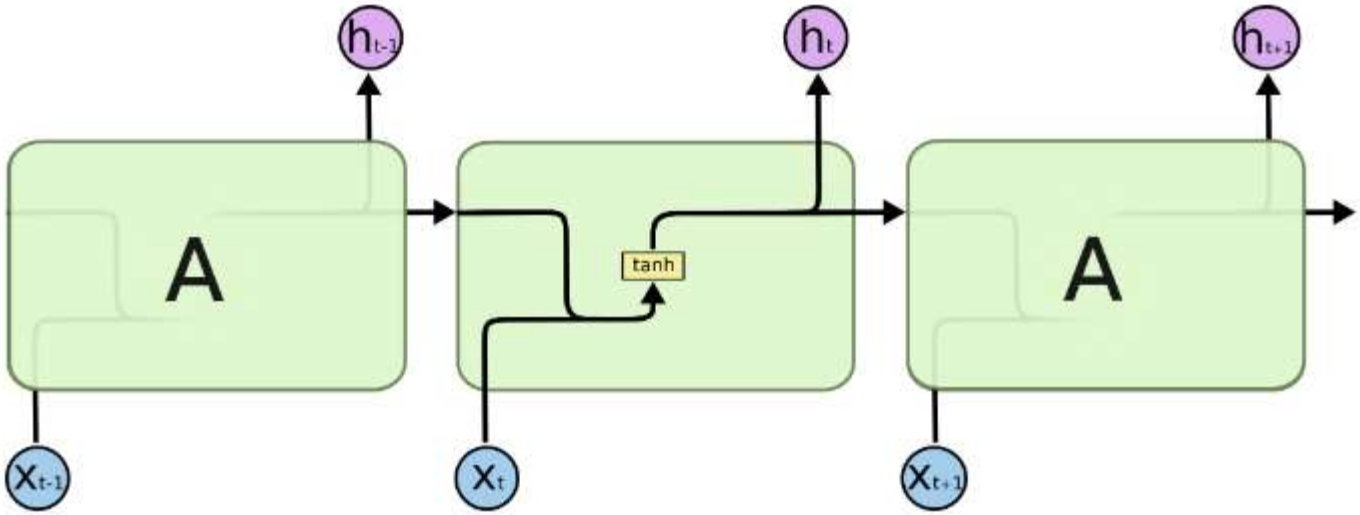
Мотивация LSTM: сеть должна сама долго помнить контекст, какой именно – сеть должна определить сама.

Вводится вектор C_t состояния сети в момент времени t. Разделяются векторы кратковременной и долговременной памяти.

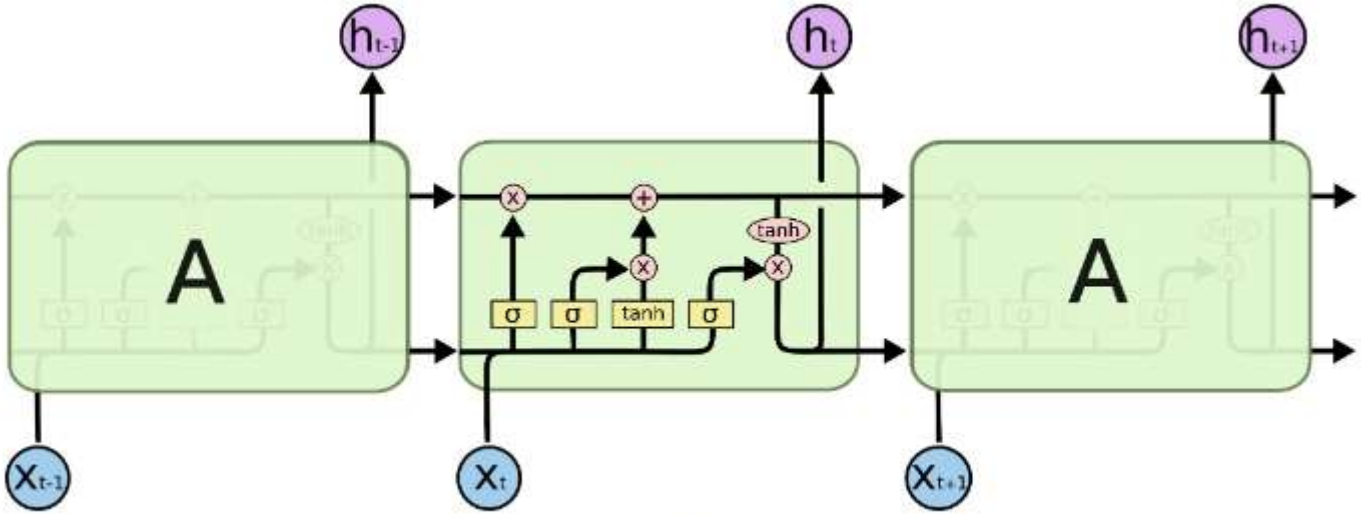


Источник: https://www.bioinf.jku.at/publications/older/2604.pdf

- LSTM частично решает проблему исчезновения и взрыва градиентов в процессе обучения
- Все RNN можно представить в виде цепочки повторяющихся блоков (линейный слой + функция активации). В LSTM повторяющийся блок имеет более сложную структуру, состоящую не из одного, а из четырех слоев. Появляется понятие состояния блока (cell state, c_n)
- Cell state используется в роли внутренней (закрытой) информации LSTM-блока. Скрытое состояние h_t передается наружу, как в следующий блок так и в следующий слой или выход сети.
- LSTM может добавлять или удалять определенную информацию из cell state с помощью специальных механизмов – gates



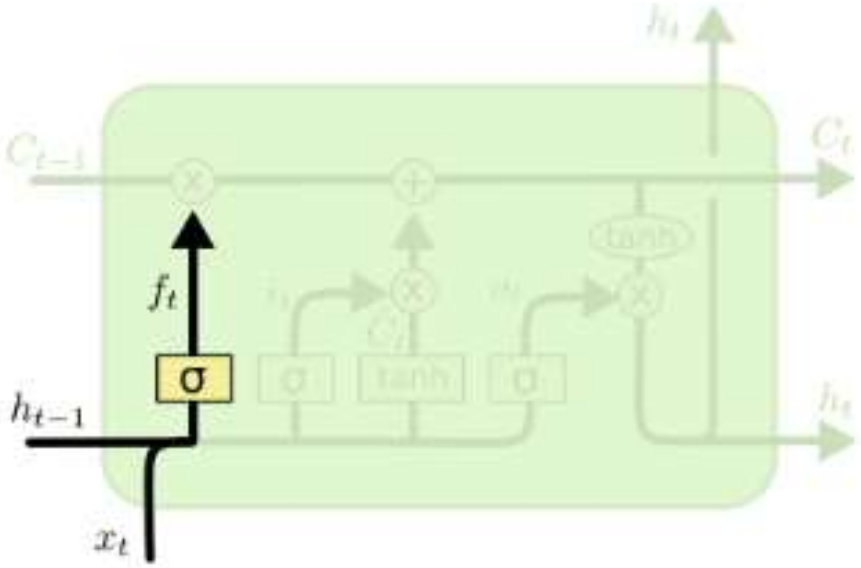
The repeating module in a standard RNN contains a single layer.



The repeating module in an LSTM contains four interacting layers.

Источник: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Forget gate (вентиль забывания) – на основе предыдущего скрытого состояния h_{t-1} и нового входа x_t определить, какую долю информации из c_{n-1} (состояния предыдущего блока) стоит пропустить дальше, а какую забыть

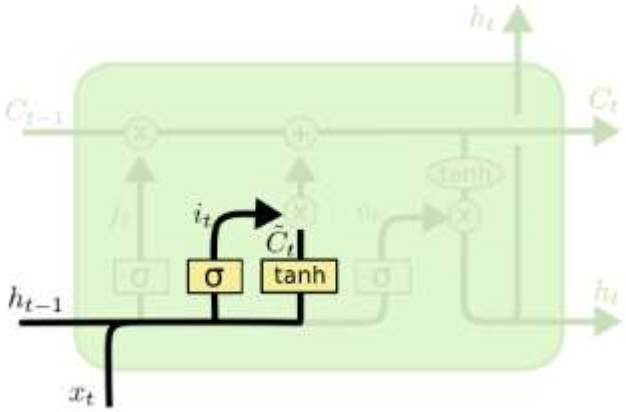


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Следующий шаг – определить что вносится в cell state

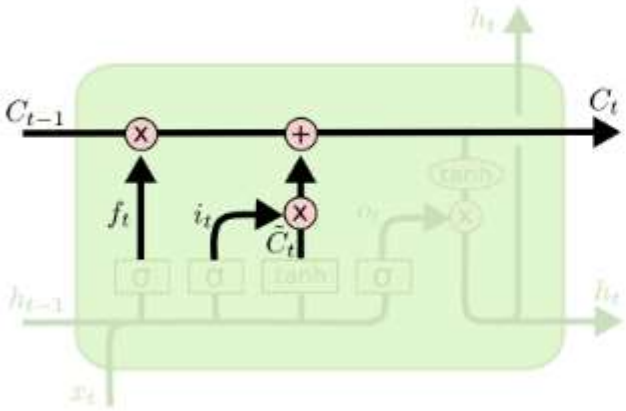
Источник: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Input gate (вентиль входного состояния) – решает, какие слагаемые надо «забыть», а какие внести



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

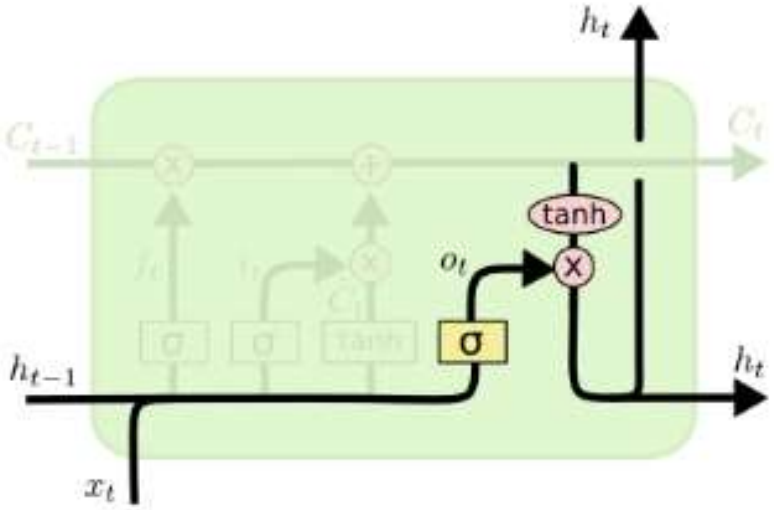
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Источник: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Output gate (вентиль выходного состояния) – отвечает на вопрос о том, сколько информации из cell state следует отдавать на выход из LSTM-блока. Доля передаваемой информации o_t



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

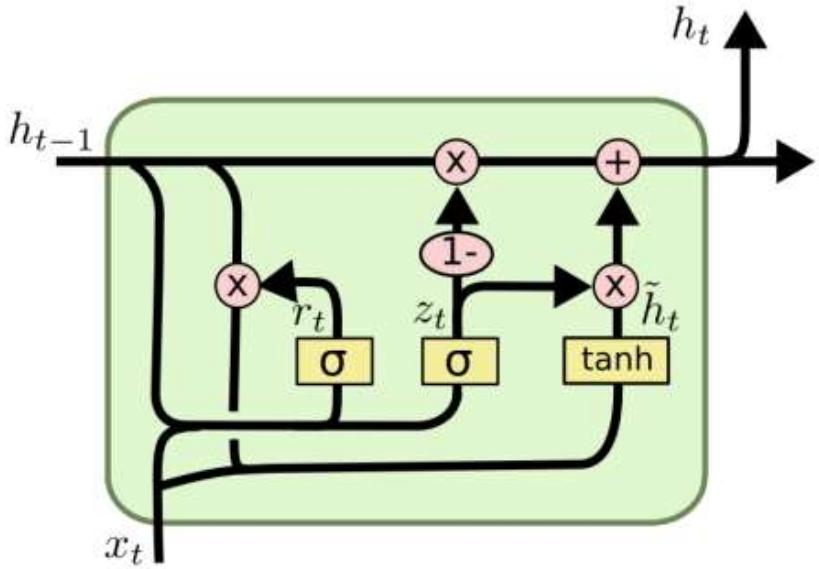
Источник: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Gated Recurrent Unit (GRU)

GRU является логическим упрощением LSTM с сохранением всех достоинств оригинальной архитектуры.

Используется только состояние h_t, вектор C_t не вводится.
Используется update gate – обобщение input gate и forget gate

Reset gate – решает, какую часть памяти нужно перенести дальше с прошлого шага



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right)$$
$$\tilde{h}_t = \mathrm{th}\left(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h\right)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Источник: https://https://arxiv.org/pdf/1406.1078v3

Update gate – будем забывать только те значения, которые собираемся обновить (forget gate + input gate):

$$z_t = \sigma(h_{t-1}W_1^z + x_t W_2^z + b_z)$$

Reset gate – определяет, какую долю информации из h_{t-1} с прошлого шага надо «сбросить» и инициализировать заново:

$$r_t = \sigma(h_{t-1}W_1^r + x_t W_2^z + b_r)$$

Вычисляем потенциальное обновление для скрытого состояния:

$$\widetilde{h_n} = \tanh((r_t \odot h_{t-1})W_1^h + x_t W_2^h + b_h)$$

Решаем, что из старого забыть, а что из нового добавить:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t}$$

Источник: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

An Empirical Exploration of Recurrent Network Architectures

- LSTM и GRU придуманы эмпирически, неясно являются ли эти архитектуры оптимальными
- В ходе исследования, проведенного Rafal Jozefowicz, Ilya Sutskever (Google Inc.) и Wojciech Zaremba (NY University, Facebook) была проведена оценка более десяти тысяч различных архитектур RNN и LSTM. Были найдены некоторые архитектуры (ячейки), которые превосходят и LSTM и GRU на некоторых, но не на всех задачах

Arch.	5M-tst	10M-v	20M-v	20M-tst
Tanh	4.811	4.729	4.635	4.582 (97.7)
LSTM	4.699	4.511	4.437	4.399 (81.4)
LSTM-f	4.785	4.752	4.658	4.606 (100.8)
LSTM-i	4.755	4.558	4.480	4.444 (85.1)
LSTM-o	4.708	4.496	4.447	4.411 (82.3)
LSTM-b	4.698	4.437	4.423	**4.380 (79.83)**
GRU	4.684	4.554	4.559	4.519 (91.7)
MUT1	4.699	4.605	4.594	4.550 (94.6)
MUT2	4.707	4.539	4.538	4.503 (90.2)
MUT3	4.692	4.523	4.530	4.494 (89.47)

MUT1:

$$\begin{aligned} z &= \text{sigm}(W_{\text{xz}}x_t + b_z) \\ r &= \text{sigm}(W_{\text{xr}}x_t + W_{\text{hr}}h_t + b_r) \\ h_{t+1} &= \tanh(W_{\text{hh}}(r \odot h_t) + \tanh(x_t) + b_{\text{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \text{sigm}(W_{\text{xz}}x_t + W_{\text{hz}}h_t + b_z) \\ r &= \text{sigm}(x_t + W_{\text{hr}}h_t + b_r) \\ h_{t+1} &= \tanh(W_{\text{hh}}(r \odot h_t) + W_{xh}x_t + b_{\text{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \text{sigm}(W_{\text{xz}}x_t + W_{\text{hz}}\tanh(h_t) + b_z) \\ r &= \text{sigm}(W_{\text{xr}}x_t + W_{\text{hr}}h_t + b_r) \\ h_{t+1} &= \tanh(W_{\text{hh}}(r \odot h_t) + W_{xh}x_t + b_{\text{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

Источник: https://proceedings.mlr.press/v37/jozefowicz15.pdf

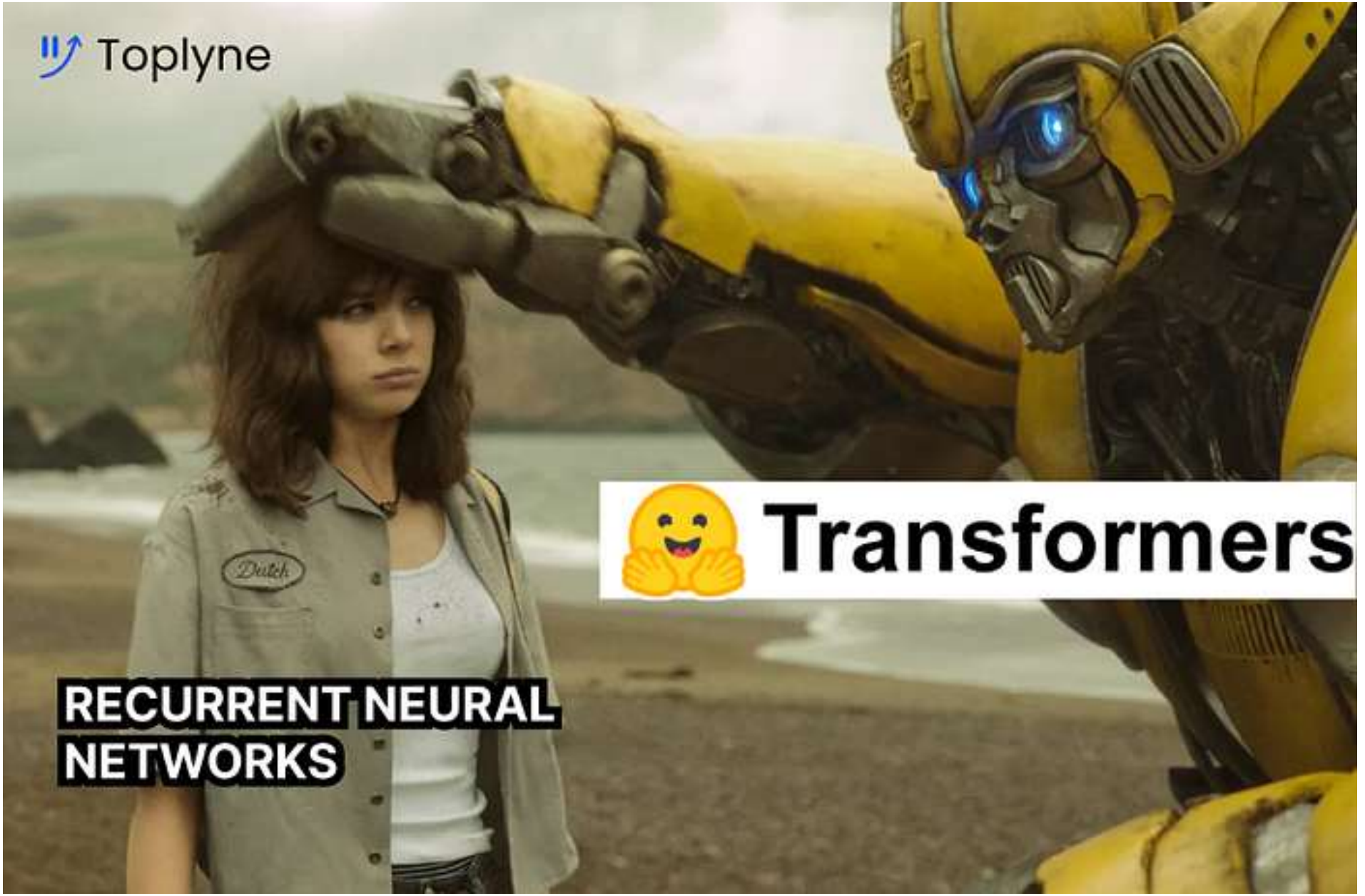
Достоинства RNN архитектур:

- Способны учитывать контекст и последовательность данных
- Способны обрабатывать ввод любой длины
- Эффективны на задачах с временными зависимостями

Недостатки RNN архитектур:

- Вычислительная сложность, взрыв градиента
- Ограничения на долгосрочное запоминание, сложно получить информацию со всей последовательности

Модели внимания и трансформеры



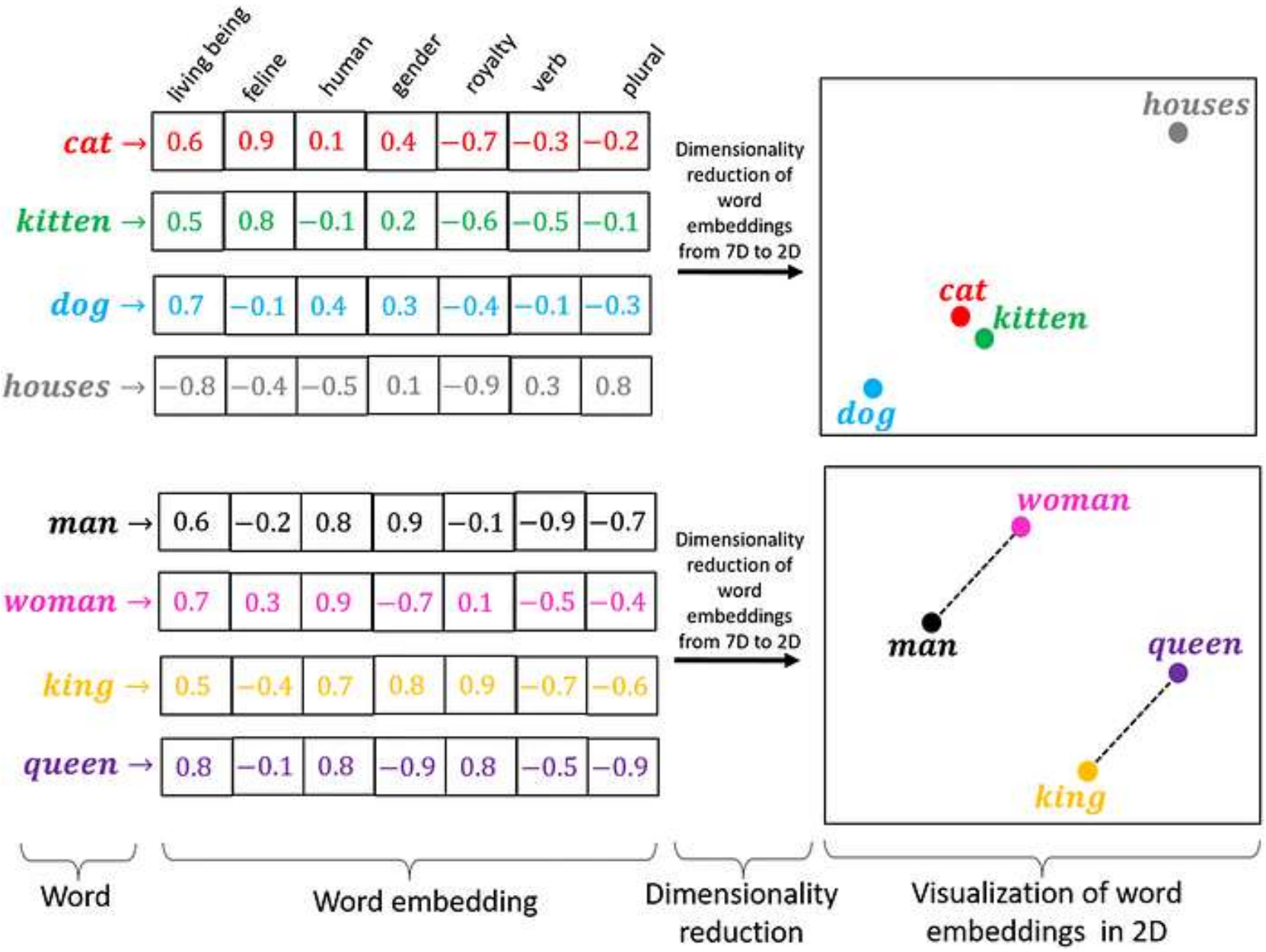
- Текст (документ) – последовательность токенов
- Токенизация – представление текста в виде последовательности
- Токен – последовательность символов (слово или часть слова)

Источник: https://github.com/FUlyankin

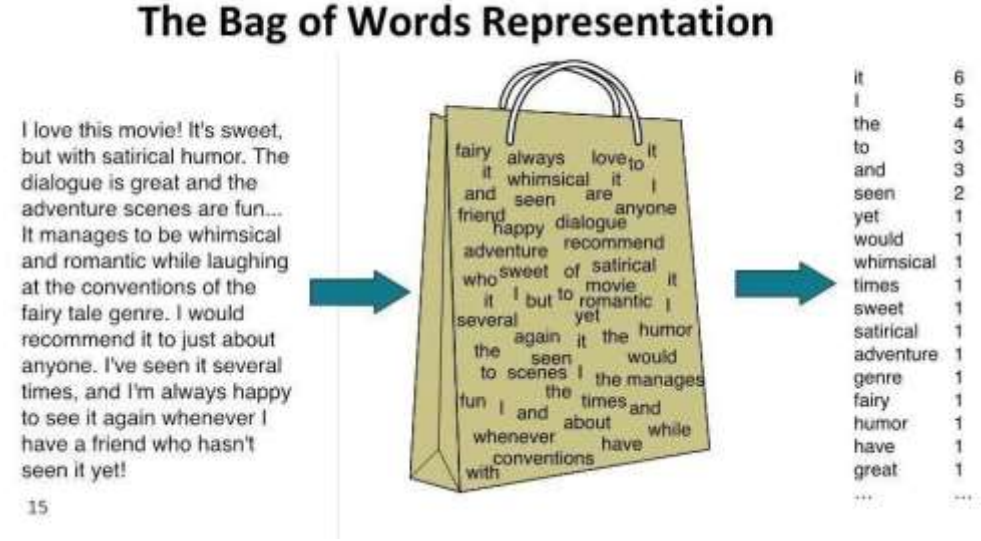
Представление текста

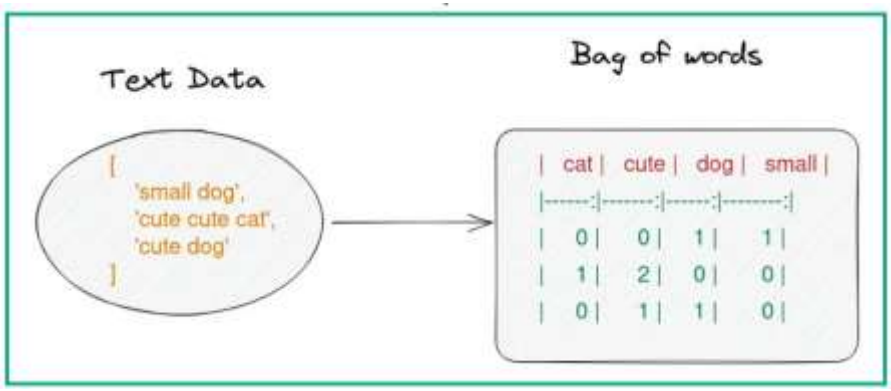
Необходимо кодировать текстовые данные – преобразовать их в вектор, прежде чем подавать на вход нейронной сети. Существует два подхода:

1. Векторизовать текст целиком, превращая его в один вектор
2. Векторизовать отдельные структурные единицы, превращая текст в последовательность векторов (embedding)


living being
feline
human
gender
royalty
verb
plural
cat → 0.6 0.9 0.1 0.4 −0.7 −0.3 −0.2
kitten → 0.5 0.8 −0.1 0.2 −0.6 −0.5 −0.1
dog → 0.7 −0.1 0.4 0.3 −0.4 −0.1 −0.3
houses → −0.8 −0.4 −0.5 0.1 −0.9 0.3 0.8
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
man → 0.6 −0.2 0.8 0.9 −0.1 −0.9 −0.7
woman → 0.7 0.3 0.9 −0.7 0.1 −0.5 −0.4
king → 0.5 −0.4 0.7 0.8 0.9 −0.7 −0.6
queen → 0.8 −0.1 0.8 −0.9 0.8 −0.5 −0.9
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

- *Bag-of-Words (мешок слов)* – текст представляется в виде вектора частот встречаемости каждого токена, кроме элементов заранее заданного списка «стоп-слов», в которые обычно включают самые вездесущие токены: личные местоимения, артикли и так далее
- Представление мешка слов – таблица с числами, в которой столбцы – уникальные слова, а в ячейках находится число вхождений слова в документ. В каждой строке получается набор чисел (вектор), характеризующий состав документа
- Не учитывается порядок слов – два предложения могут называться одинаковыми, если содержат один и тот же набор слов





- *TF-IDF (Term Frequency - Inverse Document Frequency)* – статистическая мера, используемая для оценки важности слова в контексте документа, являющегося частью коллекции документов. Вес некоторого слова пропорционален частоте употребления этого слова в документе и обратно пропорционален частоте употребления слова во всех документа коллекции
- TF (Term Frequency, частота термина) – обозначает, насколько часто определенное слово появляется в данном документе (частота вхождения токена в документ).
- TF измеряет важность слова в контексте отдельного документа

$$TF(t,d) = \frac{n_t}{\sum_k n_k}$$

где n_t – число вхождений токена t в документ,
а в знаменателе стоит общее число слов в данном документе

- IDF (Inverse document frequency) – измеряет, насколько уникально слово является по всей коллекции документов. Слова, которые в появляются большинстве документов, имеют низкое IDF, так как не вносят большой информационной ценности

$$IDF(t, D) = log \frac{|D|}{|\{d_i \in D | t \in d_i\}|}$$

где $\{d_i \in D | t \in d_i\}$ – число документов в текстовой коллекции D, в которых встречается слово t

- Этот множитель штрафует компоненты, отвечающие слишком распространенным токенам и повышает вес специфических для отдельных текстов (и, вероятно, информативных) слов

Мера TF-IDF часто используется в задача анализа текстов и информационного поиска

Преимущества TD-IDF:

- Учет важности слов – учитывает как частоту слова в документе, так и его общую редкость по всей коллекции. Выделяет ключевые слова, которые часто встречаются в данном документе, но не слишком распространены в остальных
- Устранение шума – слова, которые встречаются в большинстве документов - «стоп-слова» - имеют низкий общий вес TF-IDF

Недостатки TF-IDF:

- Отсутствие семантической информации – не учитывает семантические связи между словами
- Чувствительность к длине документа – длинные документы могут иметь более высокие значения TF, даже если ключевые слова встречаются реже

Words embeddings

- Мотивация использовать контекст:

Допустим, что одно и то же слово будет представлено одним и тем же вектором во всех текстах и в любых позициях. Как заключить в векторе его смысл (информацию)?

Использование контекста позволяет понять смысл слова по словам, встречающимся рядом с ним

«Я не успел к началу лекции, потому что опоздал на…»

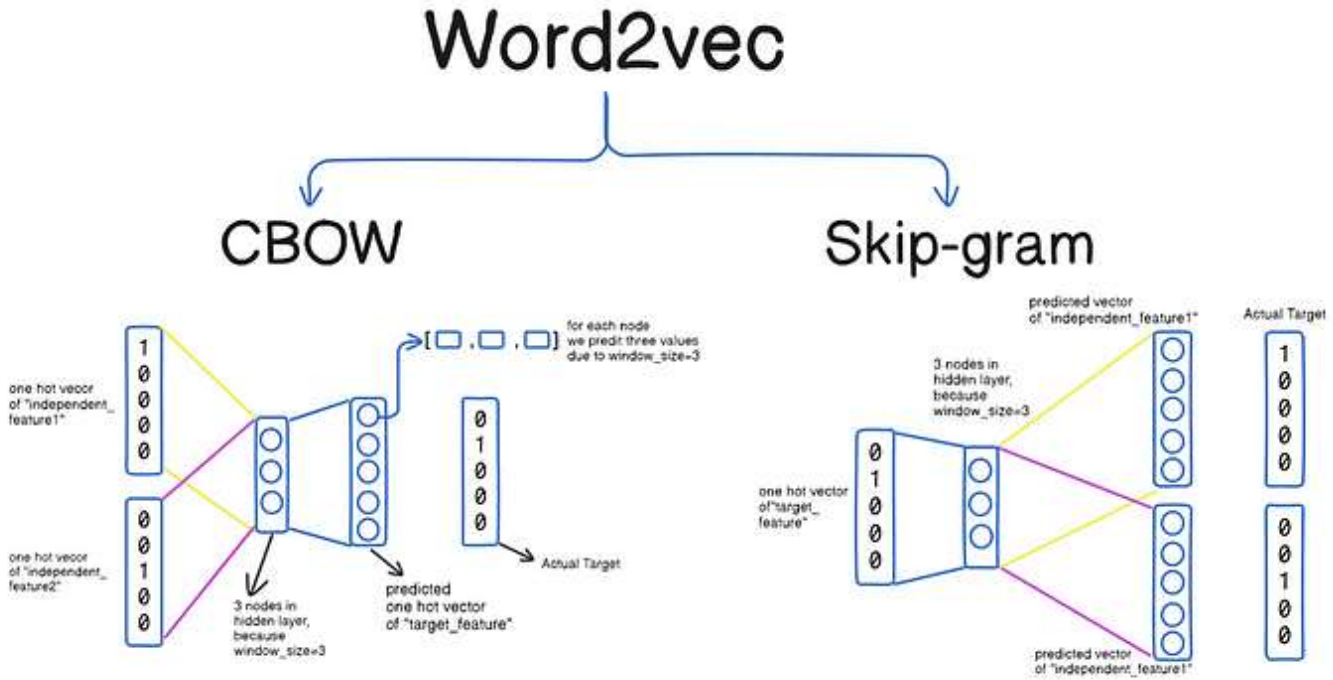
1. *Автобус – 0.6*
2. *Метро – 0.3*
3. *Такси – 0.05*
4. *Велосипед – 0.04*
5. *Самолет – 0.01*
6. *…*

Word2vec

Реализация подхода с использованием контекста – word2vec

Предложен Томашем Миколовым в 2013 году в статье «Efficient Estimation of Word Representations in Vector Space»

Основная идея – упаковать информацию о контексте слов в вектора. Будем обучать вектора, пытаясь предсказать контекст, в котором встречается слово



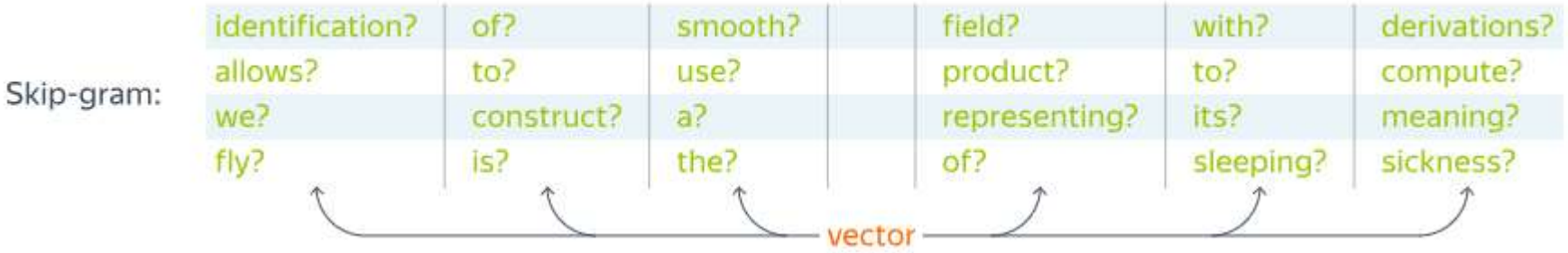
Источник: https://https://arxiv.org/abs/1301.3781

Предложено две стратегии – CBOW и Skip-gram

1. CBOW (continuous bag-of-words) – модель учится предсказывать данное (центральное) слово по контексту. Например, по двум словам перед данным и по двум словам после него.

Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

2. Skip-gram – модель учится предсказывать контекст. Например, каждого из двух соседей слева и справа.



- Для каждого слова w обучается два эмбеддинга: v_u и v_w. Первый используется, когда w является центральным, второе – когда оно рассматривается как часть контекста.

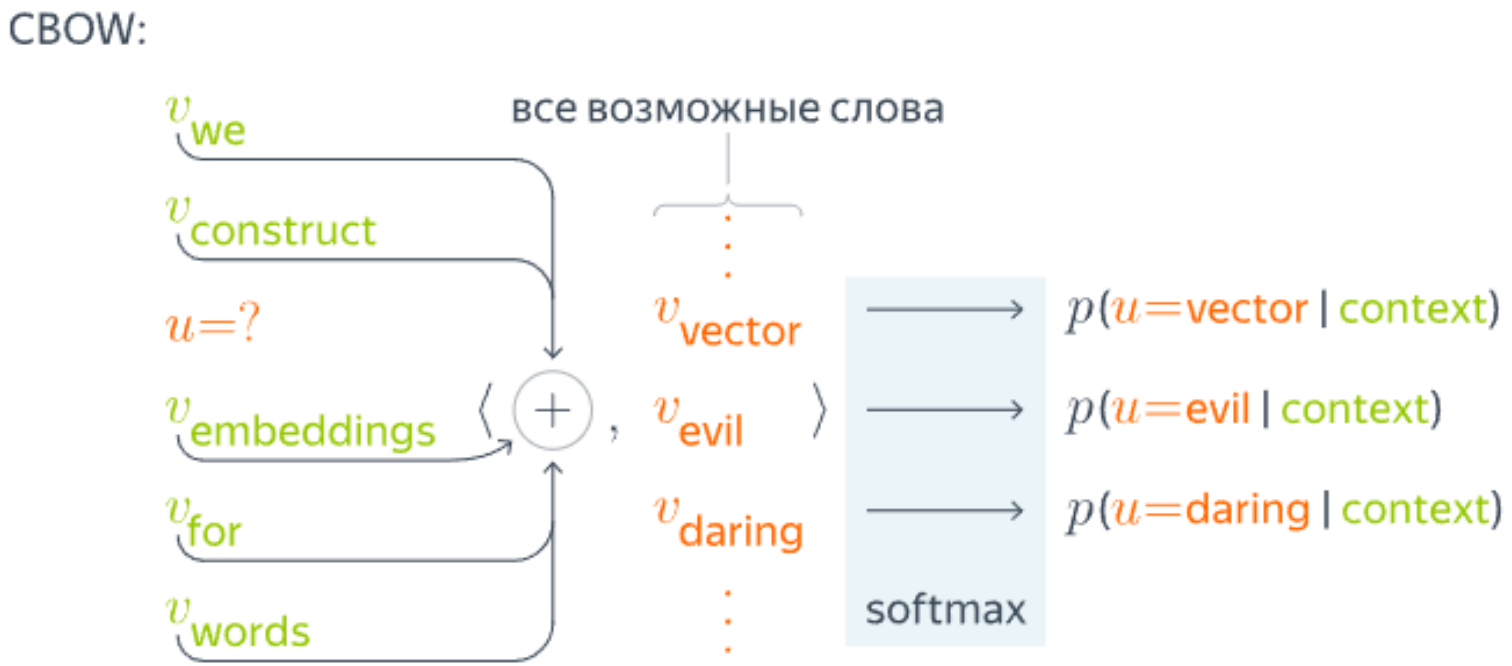
Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

В модели CBOW при фиксированном контексте «*context*» вычисляются логиты

$$logits_u = \langle \sum_{w \in \text{context}} v_w, v_u \rangle$$

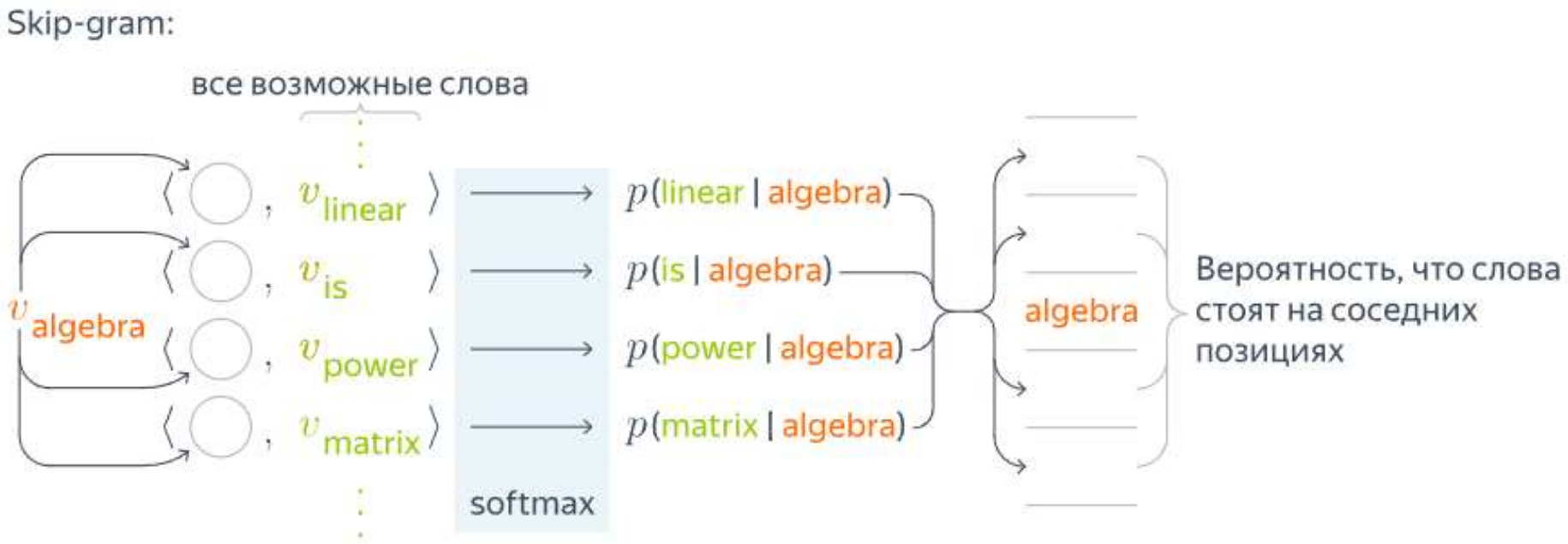
После чего «вероятности» всевозможных слов u быть центральным словом для «*context*» вычисляются как *softmax(logits)*

Модель учится с помощью SGD на кросс-энтропию полученного распределения с истинным распределением центральных слов



Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

В модели Skip-gram по данному центральному слову *u* для каждой позиции контекста «context» независимо предсказывается распределение вероятностей. В качестве функции потерь выступает сумма кросс-энтропий распределений слов контекста с их истинными распределениями



Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

- Примеры. Возьмём несколько слов и посмотрим, как выглядят топ-10 слов, ближайших к ним в пространстве эмбеддингов (обученных на одном из датасетов Quora Questions с помощью word2vec):

1. ***quantum***: electrodynamics, computation, relativity, theory, equations, theoretical, particle, mathematical, mechanics, physics;
2. ***personality***: personalities, traits, character, persona, temperament, demeanor, narcissistic, trait, antisocial, charisma;
3. ***triangle***: triangles, equilateral, isosceles, rectangle, circle, choke, quadrilateral, hypotenuse, bordered, polygon;
4. ***art***: arts, museum, paintings, painting, gallery, sculpture, photography, contemporary, exhibition, artist.

- Размерность эмбеддинга в каждой из архитектур — это гиперпараметр и подбирается эмпирически. В оригинальной статье предлагается взять размерность эмбеддинга 300. Полученные представления центральных слов могут дальше использоваться в качестве эмбеддингов слов, которые сохраняют семантическую связь слов друг с другом.

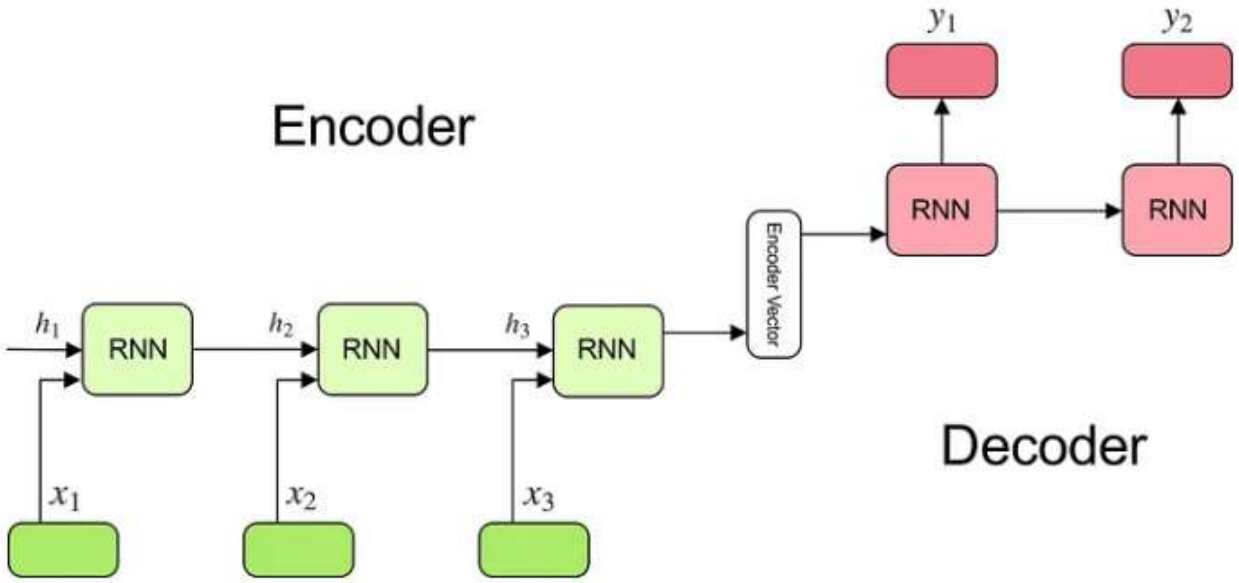
Источник: https://education.yandex.ru/handbook/ml/article/nejroseti-dlya-raboty-s-posledovatelnostyami

Sequence-to-sequence

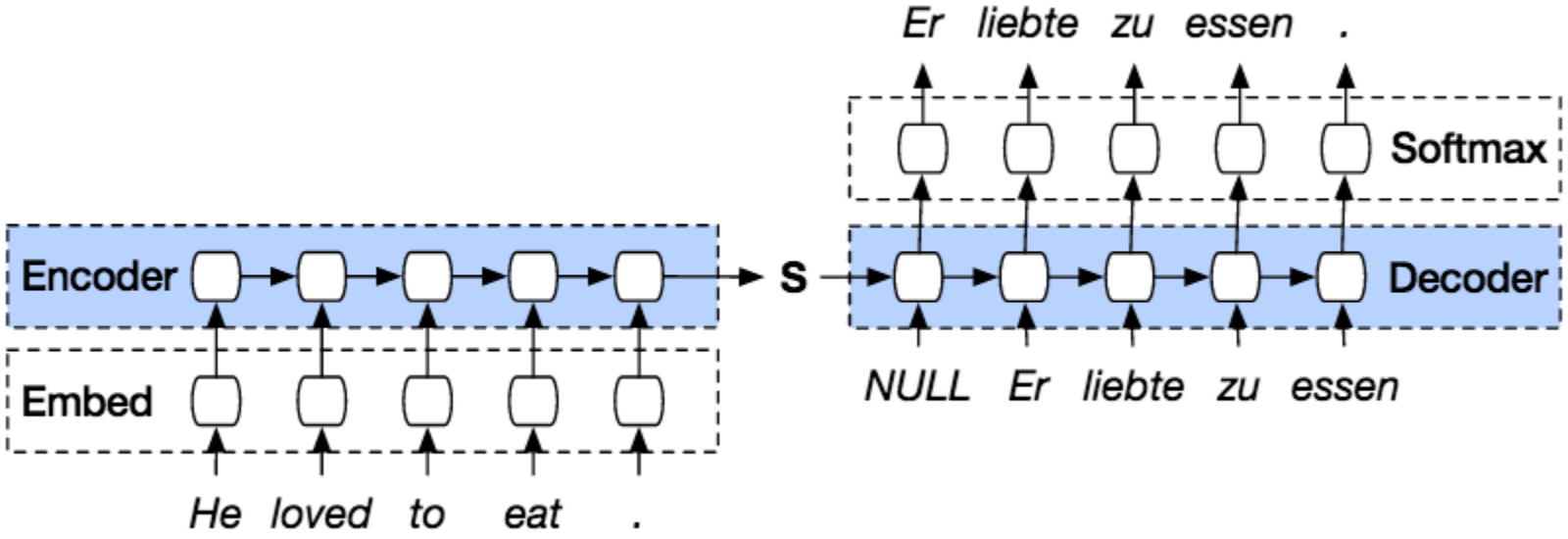
Рекуррентная сеть для синтеза последовательностей (seq2sec) – архитектура, предназначенная для задач, в которых требуется преобразование одной последовательности в другую (машинный перевод, суммаризация текста, генерация описаний и т.д.). Состоит из энкодера и декодера.

Энкодер (encoder) – кодирование информации об исходной последовательности в контекстный вектор (context vector, encoded vector)

Декодер (decoder) – преобразование закодированной энкодером информации в новую последовательность



- Входные текстовые данные преобразуются в числовые представления с помощью эмбеддинга, который преобразует каждый токен во входной последовательности в вектор фиксированной размерности



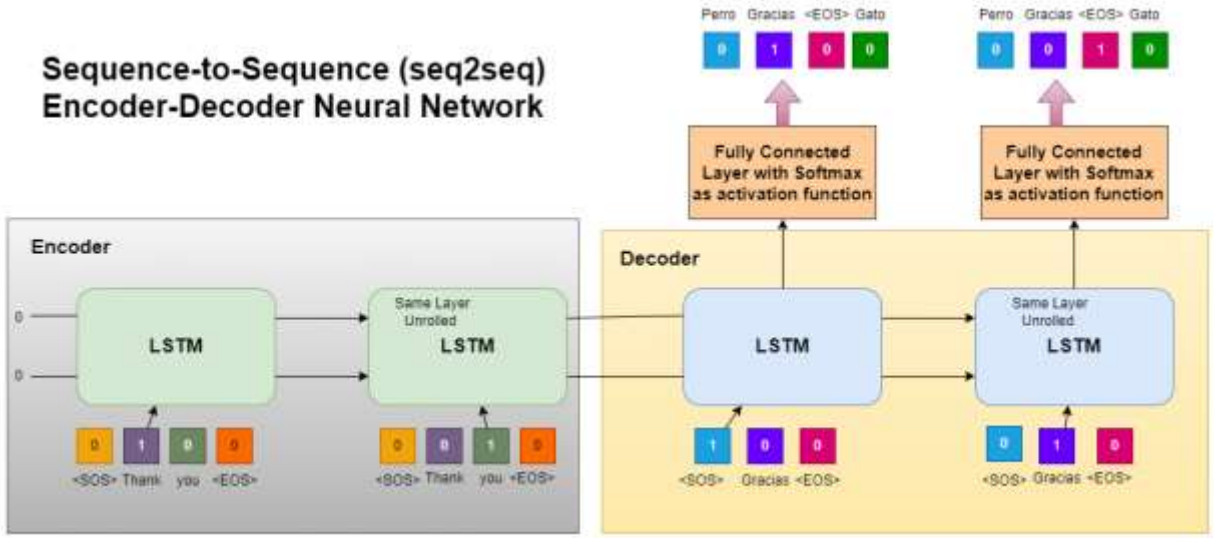
- Encoder – преобразует входную последовательность в контекстный вектор, содержащий обобщённое представление всей входной информации
- Основу энкодера RNN, обычно реализованные с использованием LSTM или GRU. Эти сети обрабатывают входную последовательность пошагово:

1. На каждом шаге RNN принимает эмбеддинговое представление текущего токена и скрытое состояние от предыдущего шага
2. Выход каждого шага включает новое скрытое состояние, которое передаётся на следующий шаг вместе со следующим токеном

- В конце последовательности RNN генерирует контекстный вектор, который является финальным скрытым состоянием. Этот вектор обобщает всю информацию из входной последовательности и передаётся в декодер для дальнейшей генерации выходной последовательности. Контекстный вектор — это своего рода сжатая версия входной последовательности, включающая в себя её смысл
- Для улучшения качества представления входной последовательности часто используются двунаправленные RNN. В этом случае два RNN работают параллельно: один — слева направо, другой — справа налево. Их состояния объединяются на каждом шаге, что позволяет учитывать как предшествующие, так и последующие слова для каждого токена в последовательности

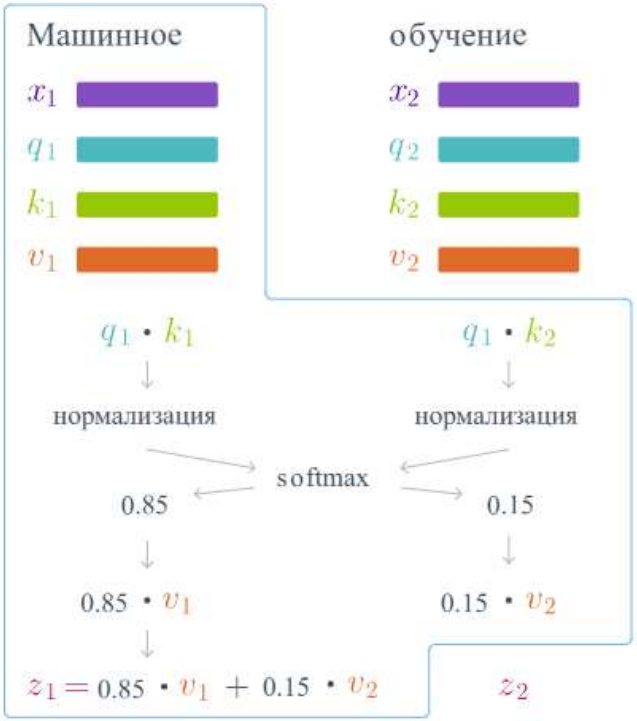
- Decoder – декодер генерирует данные на основе предыдущих предсказаний и контекстного вектора, предоставленного энкодером
- Подобно энкодеру, декодер принимает токены, которые сначала преобразуются в числовые представления с помощью эмбеддинга. На вход подаются не только реальные данные, но и предсказанные токены на предыдущих шагах. Реализован с использованием LSTM или GRU, на каждом шаге принимает:

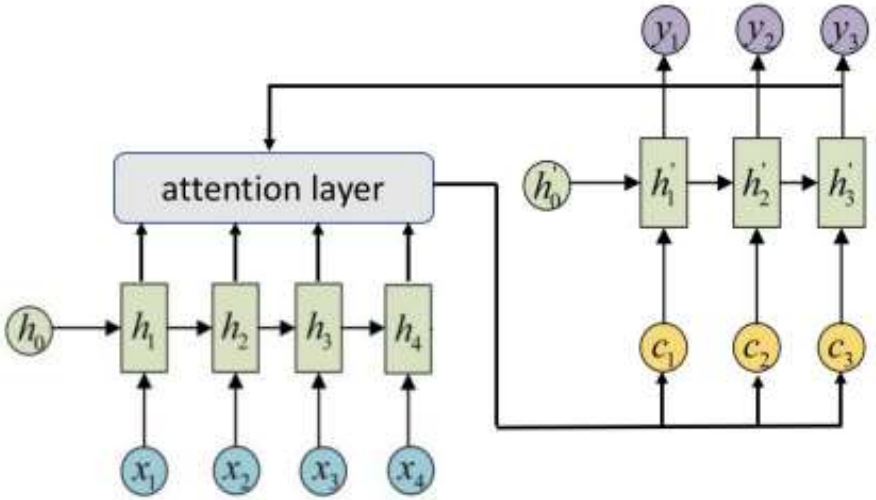
1. Контекстный вектор от энкодера
2. Предыдущий предсказанный токен (начальный токен для первого шага)
3. Скрытое состояние от предыдущего шага декодера. Этот выходной вектор затем преобразуется в вероятности через слой Softmax, который указывает на вероятность каждого возможного токена в выходной последовательности



Механизм внимания (attention)

- Предоставление декодеру информации обо всех токенах исходного предложения на каждом шаге генерации для указания, на какое слово обратить внимание. На каждом шаге декодера считаются attention scores, получаем N значений, указывающих, насколько каждый из токенов с номерами (0 … n) из исходного состояния важен для генерации токена i из выходной последовательности.





Источник: https://arxiv.org/pdf/1409.0473

Attention Is All You Need

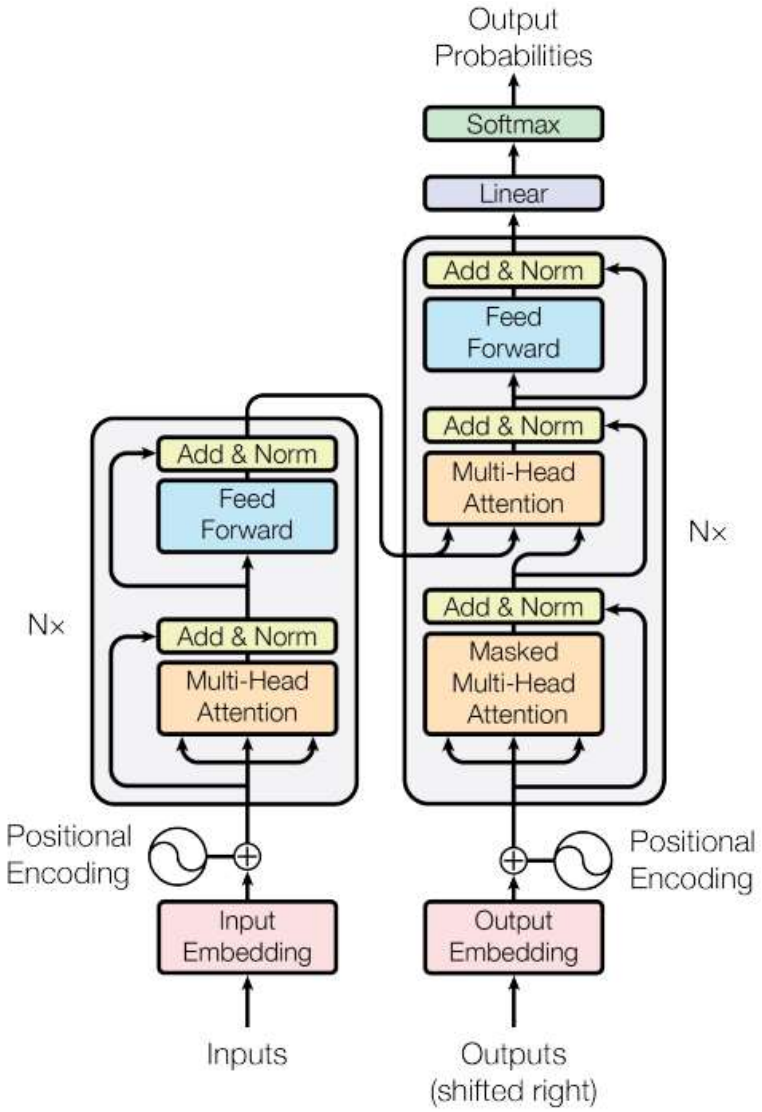


Figure 1: The Transformer - model architecture.

Источник: https://arxiv.org/pdf/1706.03762

Механизм внутреннего внимания (self-attention)

- Используется, чтобы посмотреть на другие слова во входной последовательности во время кодирования конкретного слова
- Из исходного вектора (эмбединга каждого токена) формируется три вектора – Query (запрос), Key (ключ), Value (значение). Они получаются с помощью умножения входного вектора на матрицы W_Q, W_K, W_V, веса которых учатся вместе со всеми остальными параметрами модели с помощью обратного распространения ошибки
- Выделение трех абстракций нужно, чтобы разграничить эмбединги, задающие направление внимания (query, key) и смысловую часть токена (value). Вектор query задает модальность «начальной точки» механизма внутреннего внимания (от какого токена направлено внимание), вектор key – модальность «конечной точки» (к какому токену направлено внимание). Один и тот же токен может выступать как «начальной», так и «конечной» точкой направления внимания – self-attention вычисляется между всеми токенами в выбранном фрагменте текста

Механизм внутреннего внимания (self-attention)

- Каждый токен фиксируется по очереди – он становится query и просчитывается его степень связанности со всеми оставшимися токенами. Для этого поочередно key-вектора всех токенов скалярно умножаются на query-вектор текущего токена. Полученные числа показывают, насколько важны остальные токены при кодировании query токена в конкретной позиции
- Полученные числа нормализуются и пропускаются через Softmax для получения распределения вероятностей. Затем подсчитывается взвешенная сумма value векторов, где в качестве весов используются полученные на предыдущем шаге вероятности. Полученный вектор и будет выходом внутреннего слоя внимания для одного токена
- На практике используются матричные вычисления вместо вычисления значений Q,K,V для отдельных токенов
- Обычно параллельно используется несколько self-attention блоков – «Multi-head self-attention»

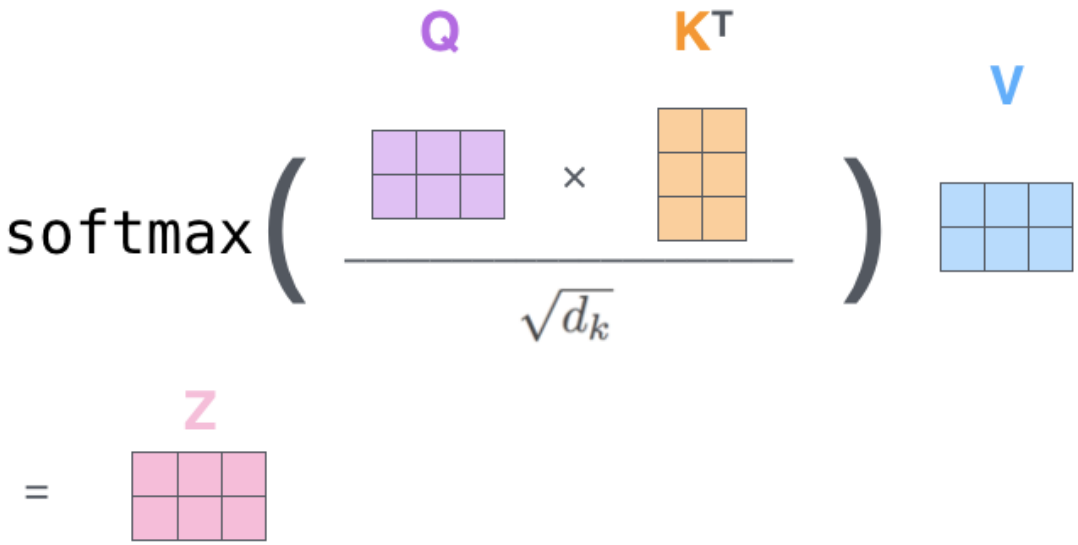
Источник: https://jalammar.github.io/illustrated-transformer/

Механизм внутреннего внимания (self-attention)

$$self\_attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Q, K, V – матрицы запросов, ключей и значений (query, key, values)

$\sqrt{d_k}$ - нормировочная константа – корень из размерности ключей и значений



The self-attention calculation in matrix form

Источник: https://jalammar.github.io/illustrated-transformer/

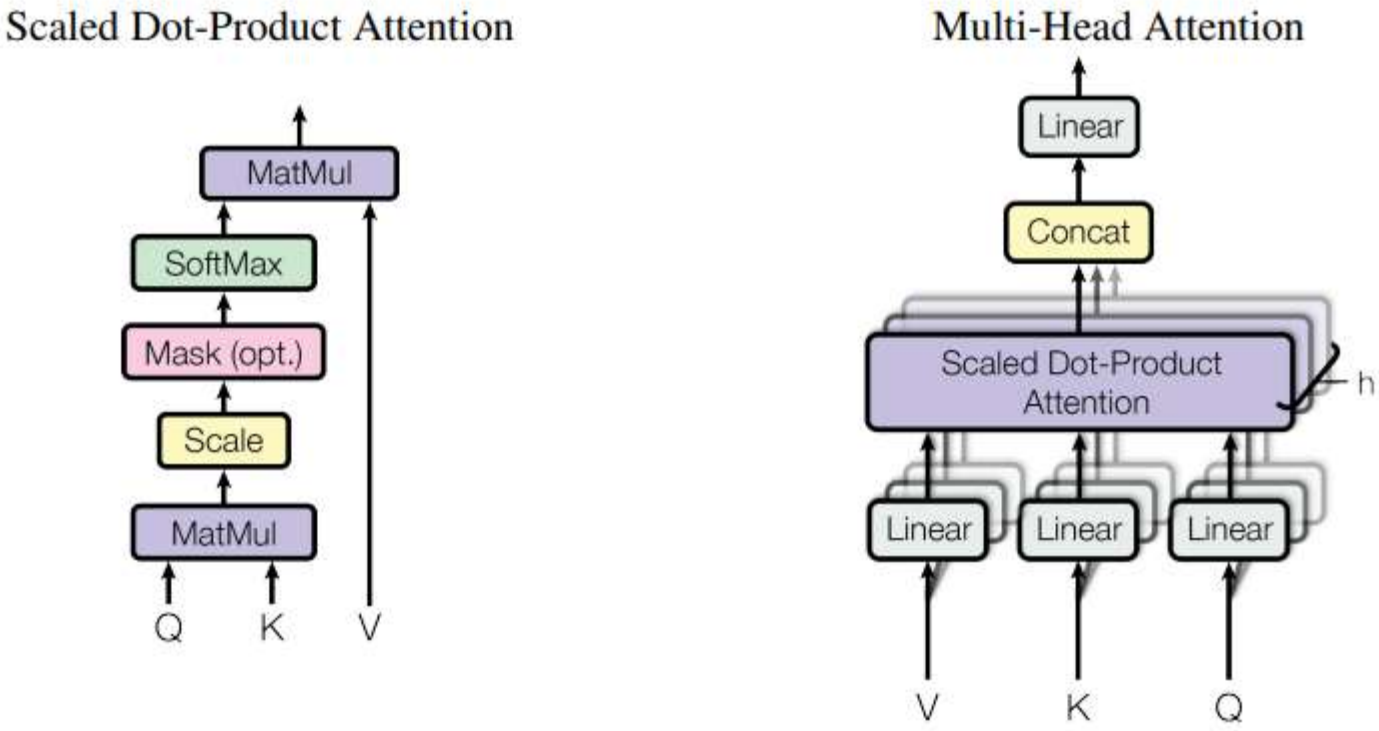


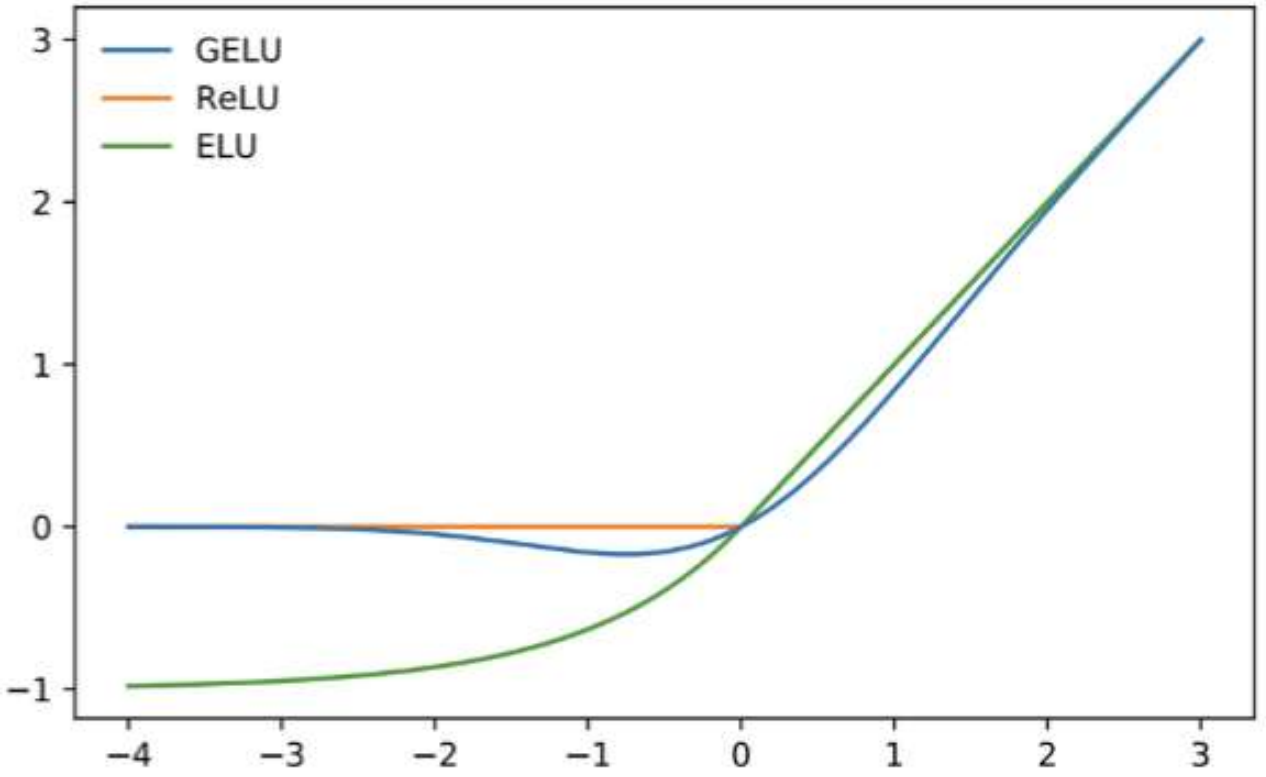
Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Источник: https://arxiv.org/pdf/1706.03762

Полносвязный слой

- Feed-forward network (FFN) – два полносвязных слоя, применяемых независимо к каждому элементу входной последовательности

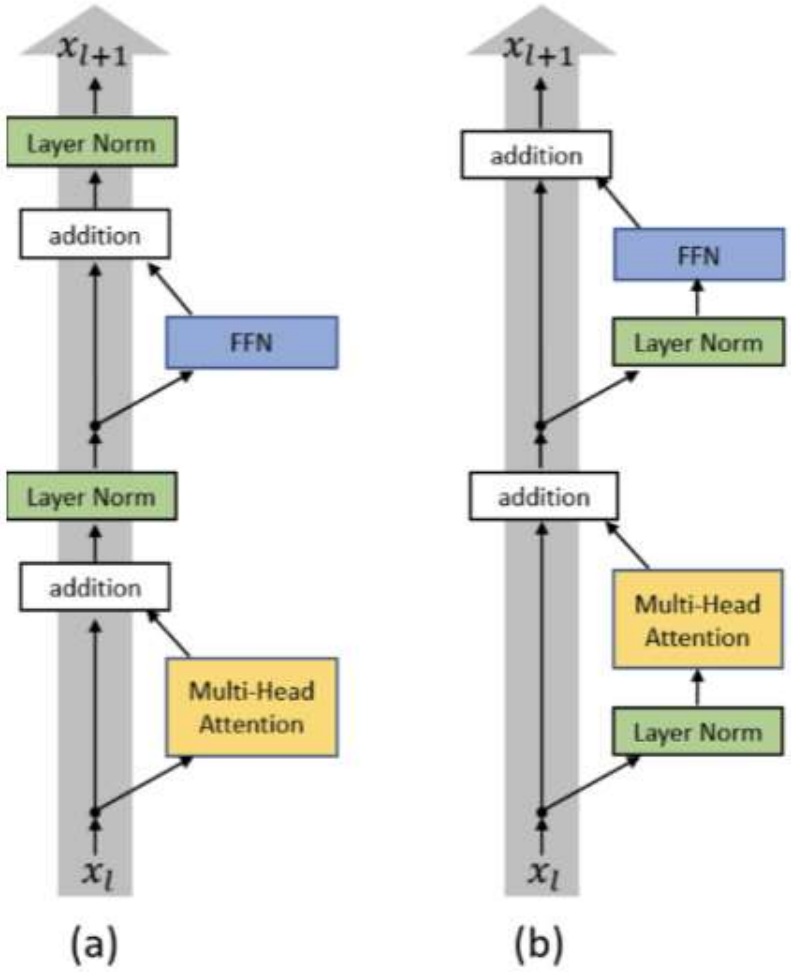
$$FFN(x) = GELU\ (xW_1 + b_1)W_2 + b_2$$



Источник: https://jalammar.github.io/illustrated-transformer/

Нормализация

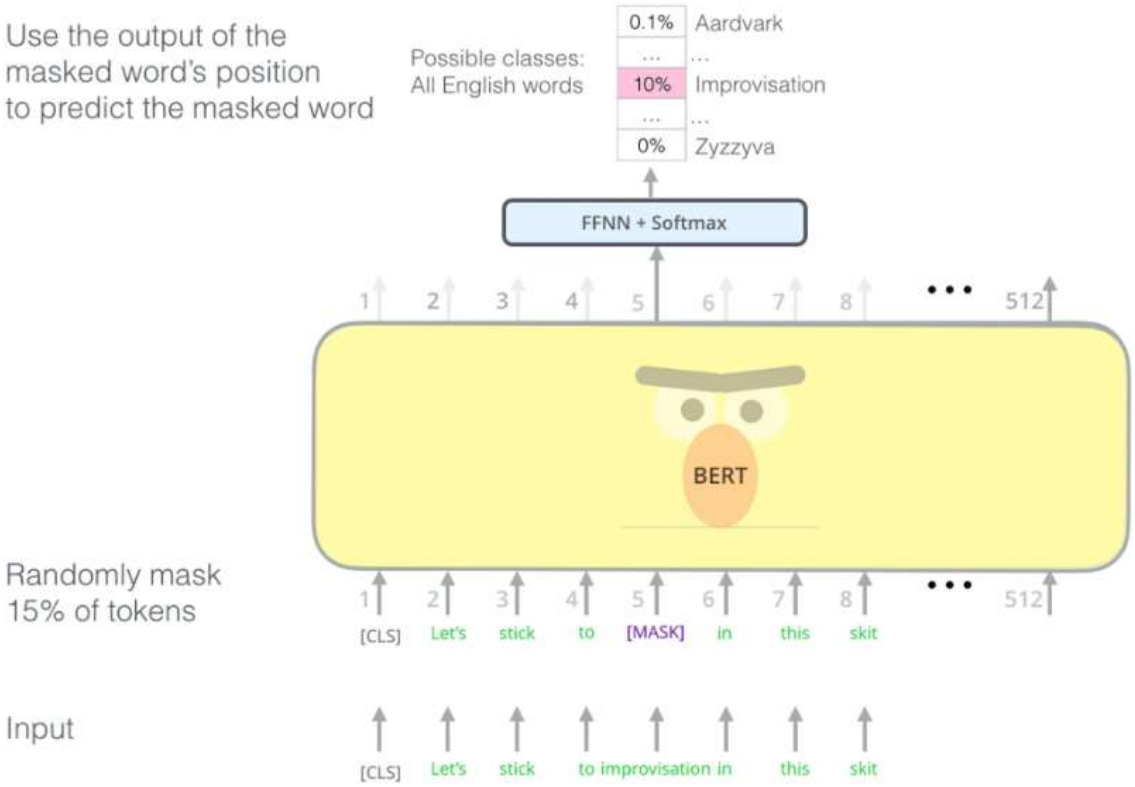
- Layer normalization – нормализация применяется после остаточной связи – PostLN (a) либо ко входу residual-ветки – PreLN (b)



Источник: https://jalammar.github.io/illustrated-transformer/

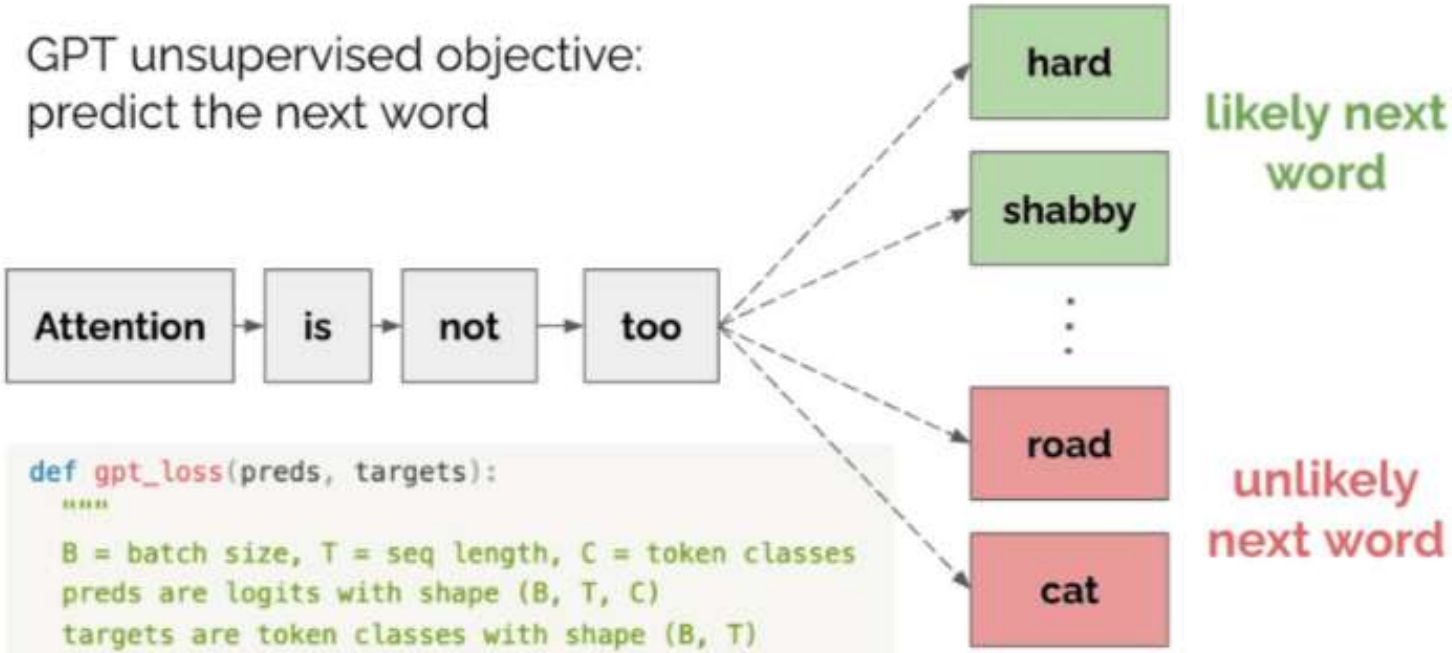
BERT

- Bidirectional Encoder Representations from Transformer – двунаправленный механизм внимания, при обработке входной последовательности все токены могут использовать информацию друг о друге
- BERT не учится генерировать тексты с нуля, первая его задача это предсказание случайно замаскированных слов по оставшимся (masked language modeling), вторая – предсказание по паре текстовых фрагментов, следуют они друг за другом или нет (next sentence prediction)



GPT

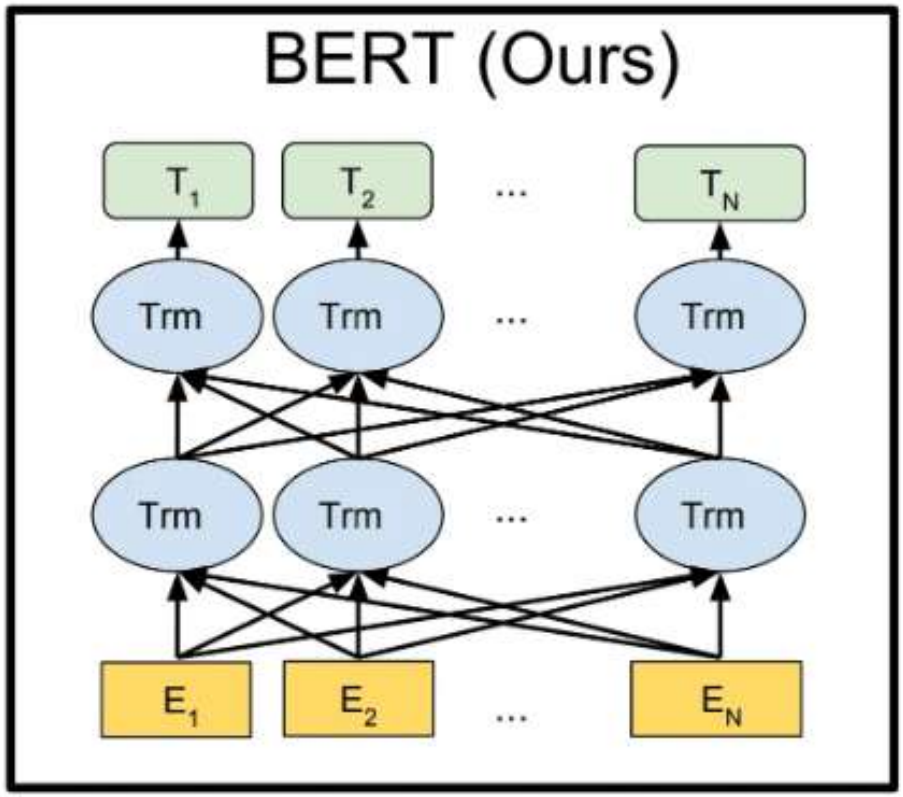
- Generative Pretrained Transformer – языковая модель, реализованная в виде последовательности слоев декодера трансформера
- В качестве задачи при обучении выступает предсказание следующего токена (многоклассовая классификация по словарю)

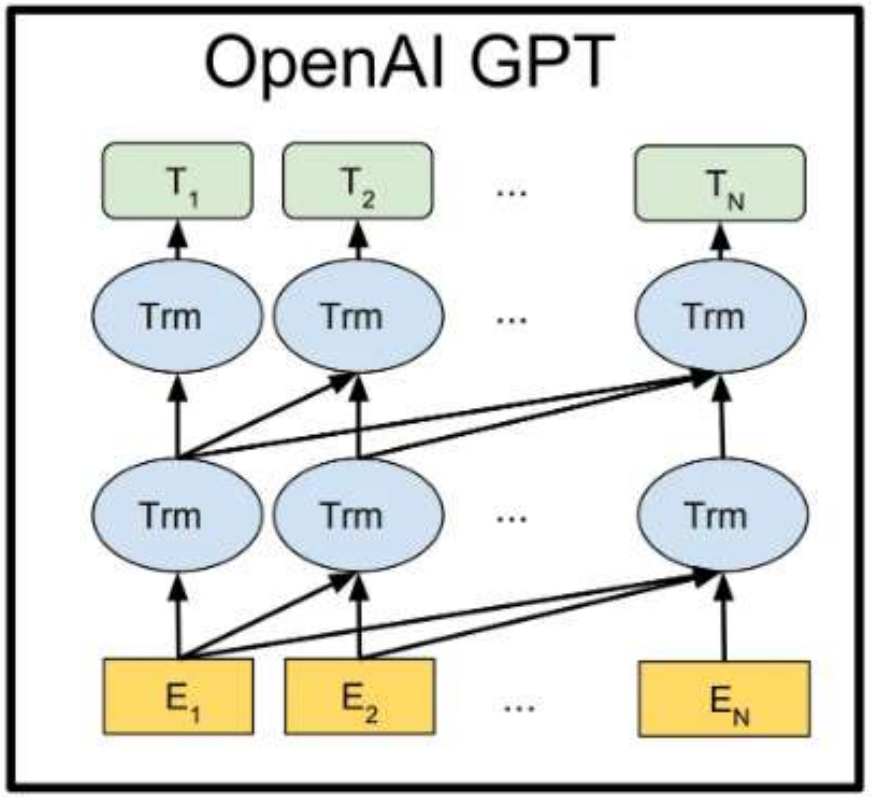



```
def gpt_loss(preds, targets):
  """
  B = batch size, T = seq length, C = token classes
  preds are logits with shape (B, T, C)
  targets are token classes with shape (B, T)
  """
  preds = preds.view(B*T,C)
  targets = targets.view(B*T)
  return F.cross_entropy(preds, targets)
```

BERT vs GPT

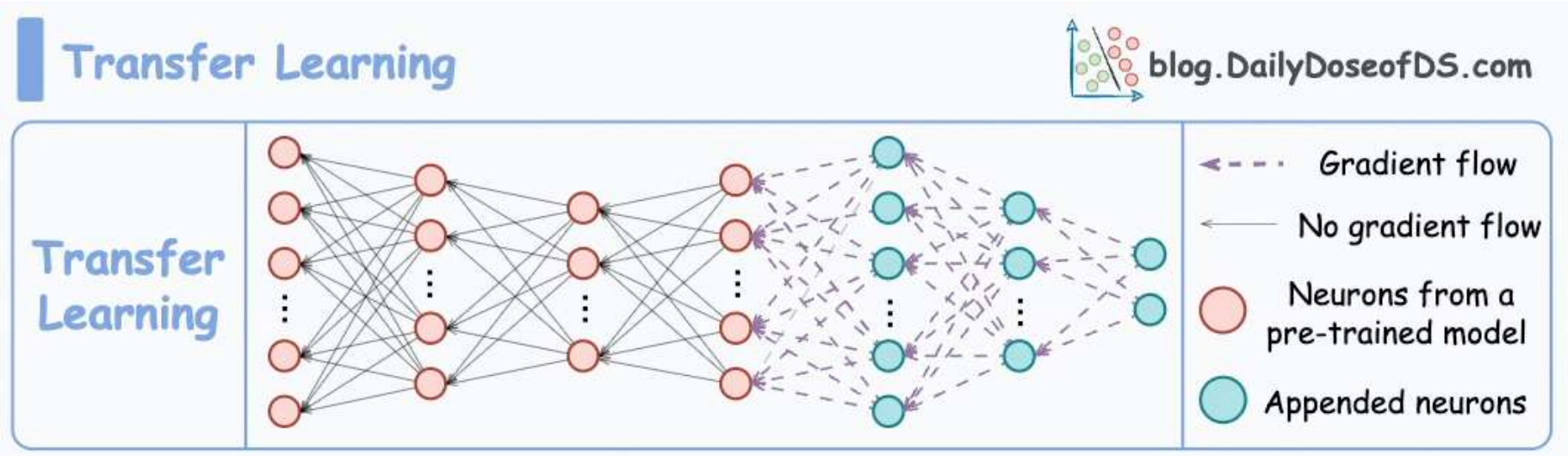
- Ключевое отличие – использование разных видов внимания





Transfer learning

- Перенос обучения (transfer learning) – метод обучения, при котором модель, обученная для одной задачи (связанная задача), переиспользуется для схожей задачи (целевая задача)
- Целевая задача содержит мало данных. Например, классификаця специфических изображений. Связанная содержит много данных, есть обученная модель которая хорошо решает связанную задачу.
- Заменяем от 1 до N последних слоев, обучаем только эти слои (веса в остальных слоях не изменяются)



Fine-tuning

- Fine tuning – метод дополнительного обучения сети, при котором обученная на схожей задаче модель обучается на новом наборе данных. При этом веса не инициализируются случайно, а используются уже оптимизированные в процессе предыдущего обучения веса нейронной сети
- Можно добавить несколько слоев на выходе нейронной сети
- Обучаются все слои