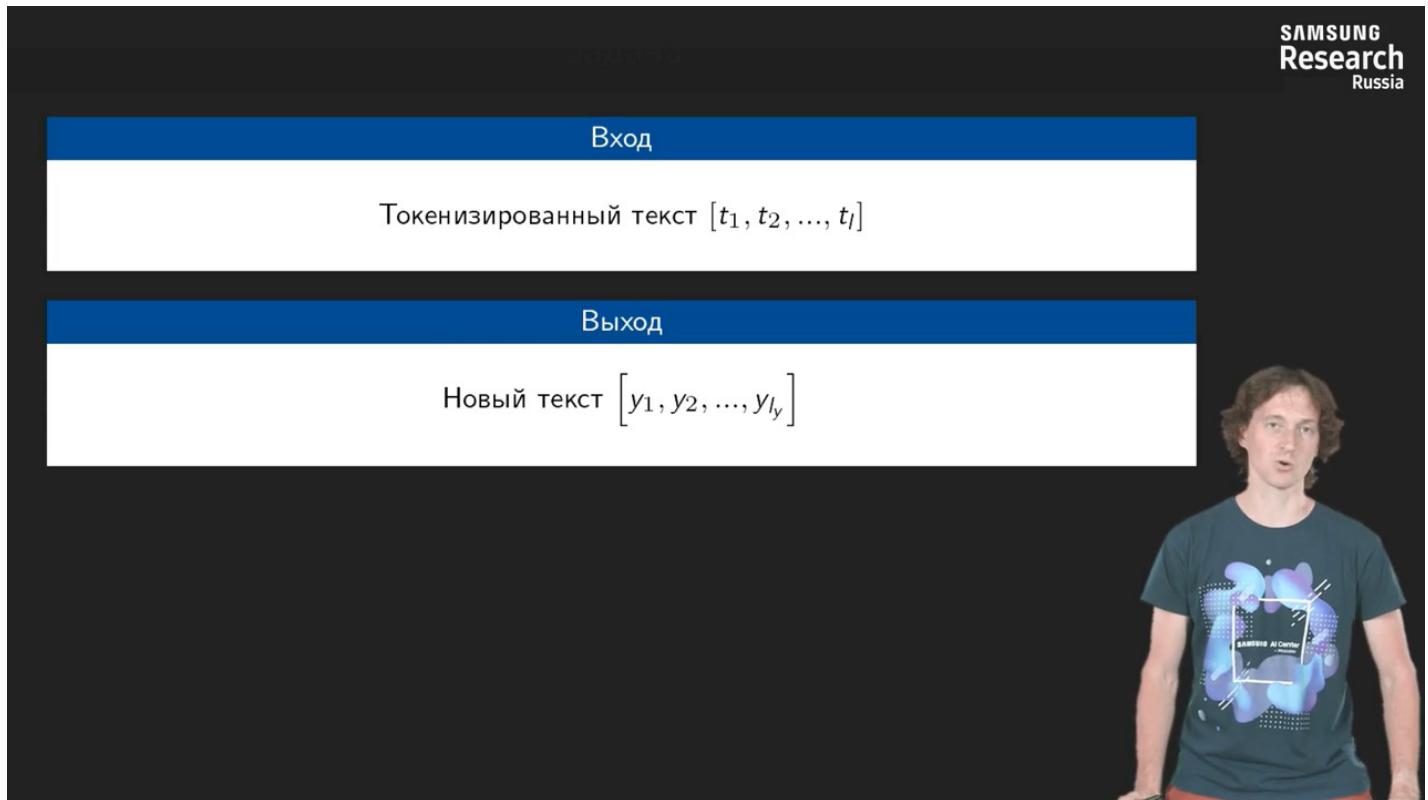


Stepik. Neural networks and NLP. 5. Sequence conversion: 1-to-1 and N-to-M - Part 2

5.4 Преобразование последовательностей (seq2seq)

Всем привет! В этом видео мы поговорим о задаче преобразования последовательностей, а также о некоторых подходах к её решению. К таким задачам, например, относится [машиинный перевод](#). Для обозначения рассматриваемых в этом видео методов, чаще всего, в англоязычной литературе используется "метод seq2seq" (сокращение "sequence to sequence").^[1] На вход нам подают список номеров токенов (результат токенизации исходного текста) и ждут от нас, что мы сгенерируем новый список номеров токенов, по которым можно будет восстановить результирующий текст. Таким образом, задача "sequence to sequence" — это задача генерации одного текста на основе другого текста. При этом, длины исходного и результирующего текстов могут не совпадать. Самое известное приложение "sequence to sequence" — это, конечно же, машинный перевод. Можно использовать подобные архитектуры и для разработки чат-ботов, например, и для генерации аннотаций, то есть скатого описания содержимого более длинного текста. В принципе, такой же подход можно использовать и для анализа структуры предложений — например, для определения частей речи, но с помощью методов [seq2seq](#) это делают (чаще) только для демонстрации возможностей метода. С практической точки зрения, для таких задач есть более простые и надёжные методы (например, [CRF](#)). Рассматриваемые же в настоящем видео методы имеет смысл использовать тогда, когда более специализированную архитектуру подобрать не получается. Обучающая выборка состоит из пар: входной текст - выходной текст. Для борьбы с проблемой неизвестных слов ("out of vocabulary words"), чаще всего, используется токенизация с помощью [byte pair encoding](#) (BPE) или других подобных алгоритмов. Таким образом, seq2seq модели преобразовывают одну последовательность чисел (целых, то есть номеров) в другую последовательность номеров. В каждой seq2seq модели есть две самые главные части — это [энкодер](#) (encoder) и декодер (decoder). Энкодер наблюдает сразу всю входную последовательность и преобразует её в один или несколько векторов. Декодер принимает в качестве входа результат работы энкодера и, на каждом шаге, выдаёт [распределение](#) вероятностей очередного токена при условии всех предыдущих. Распределение первого токена обусловлено только на входную последовательность, а все последующие — уже и на первый токен. Процесс генерации продолжается до тех пор, пока декодер не сгенерирует специальный токен, обозначающий конец последовательности. Такой процесс генерации называется "авторегрессионным".

[1] Sutskever I., Vinyals O., Le Q. V. [Sequence to sequence learning with neural networks](#) //Advances in neural information processing systems. – 2014. – С. 3104-3112.



-
- Приложения**
- SAMSUNG Research Russia
- **Машинный перевод**
"Искусственный интеллект это круто" → "La inteligencia artificial es genial"
 - **Диалоговые системы (генерация реплик)**
"Привет, как дела?" → "Всё отлично, а у тебя?"
 - **Суммаризация**
Длинный текст → Сжатое содержание
 - **Анализ структуры последовательностей**
 - POS-теггинг
"Мама мыла раму" → [NOUN, VERB, NOUN]
 - лучше решать не через seq2seq
 - seq2seq - для генерации, когда более простые (надёжные) методы неприменимы

- Исходный датасет - пары предложений

Мама мыла раму	\rightarrow	Mom washed the window.
Потом мы пошли гулять	\rightarrow	Then we went for a walk.

- Токенизация (BPE, wordpieces)

$x_{i_1,1} x_{i_1,2} x_{i_1,3} \dots$	\rightarrow	$y_{i_1,1} y_{i_1,2} y_{i_1,3} y_{i_1,4} \dots$
$x_{i_2,1} x_{i_2,2} x_{i_2,3} t_{i_2,4} \dots$	\rightarrow	$y_{i_2,1} y_{i_2,2} \dots$



Общий алгоритм и обучение

- Закодировать входную последовательность x_1, x_2, \dots, x_n
- Генерировать выходную последовательность слово за словом
 - $P(y_1 | x_1, x_2, \dots, x_n)$
 - $P(y_2 | x_1, x_2, \dots, x_n, y_1)$
 - $P(y_3 | x_1, x_2, \dots, x_n, y_1, y_2)$
 - и т.д., пока не встретится $\langle \text{EOS} \rangle$



Итак, наша нейросеть задаёт (вот такую вот) вероятностную модель.^[1] θ — это множество всех параметров модели, x — это номера токенов входной последовательности, y — это возможное значение номеров токенов результирующей последовательности. Такие модели настраиваются, в принципе, как обычно — то есть, путём решения оптимизационной задачи, сформулированной через [метод максимального правдоподобия](#). То есть, мы ищем такие

параметры нейросети, при которых вероятность про наблюдать обучающую выборку является наибольшей. В качестве функции правдоподобия для каждого токена используется категориальная [кросс-энтропия](#). Нижний индекс здесь обозначает взятие элемента массива с таким индексом. В самом классическом варианте в [seq2seq](#) использовался рекуррентный [энкодер](#), который сворачивал всю входную последовательность в один вектор. Дальше этот вектор используется, чтобы проинициализировать скрытое состояние декодера, который, опять же, рекуррентный. Декодер, на каждом шаге, предсказывает вещественный вектор размерности, равной размерности словаря. Каждый такой вектор задаёт [распределение](#) вероятностей на всех возможных токенах. В некотором смысле, это — минимальная архитектура, с помощью которой можно одну последовательность преобразовать в другую, при этом длины входной и выходной последовательностей могут не совпадать. Ожидаемо, у этой архитектуры есть пачка недостатков. Во-первых, это немного странно — кодировать последовательность любой длины всегда в вектор постоянной длины. Известно, что скрытое состояние [рекуррентки](#), в большей мере, содержит информацию о самых последних шагах, а информация о первых теряется.^[2,3,4] Это всё приводит к потере информации и ослаблению связей между декодером и энкодером. Это может приводить к тому, что декодер всегда выдаёт примерно одинаковый текст, вне зависимости от того, какой текст был подан на вход. Корень второго недостатка исходит из использования односторонних рекурренток в энкодере. Они на каждом шаге видят контекст только слева от каждого токена. Третий недостаток заключается в самих рекуррентках. Они достаточно плохо распараллеливаются, то есть, чтобы выполнить очередной шаг, надо полностью закончить вычисление предыдущего. Это не позволяет эффективно использовать возможности современных видеокарт^[9] и других вычислителей, которые используются для обучения нейросетей. Бывает, что, после обучения, нейросети с такой архитектурой генерируют примерно одни и те же тексты. Есть несколько гипотез касательно причин низкого разнообразия. Мы о некоторых из них поговорим чуть позже в этом видео. Давайте решать проблемы поочереди. Начнём с узкого места между энкодером и декодером. Давайте теперь будем брать не один лишь последний выход энкодера, а выходы энкодеров со всех шагов. Декодировать будем так же — авторегрессионно. Но теперь на каждом шаге мы строим новый контекстный вектор — c_1, c_2, c_3 , и так далее. Получать такие вектора мы будем с помощью механизма внимания^[5], в котором в качестве запроса выступает скрытое состояние декодера, а в качестве ключей и значений — выходы энкодера со всех шагов. И мы используем всю эту информацию для предсказания очередного слова, а именно — контекстный вектор, полученный на этом шаге, скрытое состояние декодера и предыдущее слово. Что же мы получаем в итоге? Кажется, узкое место между энкодером и декодером стало уже не таким узким. Но остальные проблемы как были — так и остались. Следующий шаг — обогатить контекстуальное представление в энкодере. Если мы используем рекуррентки, то первое, что приходит в голову — это добавить рекуррентку, идущую в обратную сторону.^[6] С одной стороны это, конечно, здорово. Но

только тогда, когда все элементы с предыдущего слоя уже обработаны полностью. Это ограничивает возможности использовать одновременно несколько видеокарт. Если бы вся наша нейросеть состояла только из двунаправленных рекурренток, составленных в несколько слоёв, то мы бы могли использовать только (максимум) две видеокарты. Одну — для расчёта рекурренток, проходящих "вперёд", и одну — для тех, что идут назад. Поэтому авторы этой модели — "Google Neural Machine Translation"^[7,8] — предложили компромисс между мощностью и параллелизмом: они предложили использовать двунаправленные рекуррентки только на первом слое энкодера, остальные же слои проходят по последовательности только в прямом направлении. Это позволяет, например, начать рассчитывать (вот этот) элемент тогда, когда вот этот ещё не вычислен, и, тем самым, более эффективно загружать вычислительные мощности. Показанная на слайде архитектура позаимствована из научной статьи исследователя из Google о применении нейросетей для улучшения [машииного перевода](#). Следуя тому же принципу баланса между выразительностью и параллелизмом, авторы предложили связать с помощью механизма внимания только последний слой энкодера и только первый слой декодера. На всех более высоких слоях декодера используются те же контекстные векторы, которые были получены после первого слоя с помощью механизма внимания. Кстати, здесь и в энкодере, и в декодере используется много слоёв [LSTM](#). Очень важный момент — использовать соединения в обход нелинейности — так называемые "[skip connections](#)". Это существенно облегчает процесс обучения и позволяет сделать архитектуру глубже. Чуть ранее такой же принцип былложен в основу архитектуры ResNet, очень популярной в обработке изображений. Эта модель очень выразительная и мощная. Но, так как в ней используются рекуррентки, которые трудно заставить работать параллельно, разработчикам пришлось искать компромисс. По-прежнему, с проблемой разнообразия, на этом уровне, мы ещё ничего не сделали. То есть, в научной статье, в которой была предложена эта архитектура, и откуда была взята эта картинка, предлагается набор решений, но мы ещё о них не говорили.

[1] [Статистическая теория обучения](#)

[2] [Recurrent Neural Networks \(RNN\) - The Vanishing Gradient Problem](#)

[3] Bengio Y., Simard P., Frasconi P. [Learning long-term dependencies with gradient descent is difficult](#) //IEEE transactions on neural networks. – 1994. – Т. 5. – №. 2. – С. 157-166.

[4] Pascanu R., Mikolov T., Bengio Y. [On the difficulty of training recurrent neural networks](#) //International conference on machine learning. – 2013. – С. 1310-1318.

[5] Luong M. T., Pham H., Manning C. D. Effective approaches to attention-based neural machine translation //[arXiv preprint arXiv:1508.04025](#). – 2015.

[6] Schuster M., Paliwal K. K. [Bidirectional recurrent neural networks](#) //IEEE transactions on Signal Processing. – 1997. – Т. 45. – №. 11. – С. 2673-2681.

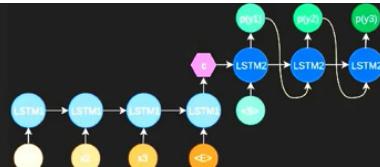
[7] Wu Y. et al. Google's neural machine translation system: Bridging the gap between human and machine translation //[arXiv preprint arXiv:1609.08144](#). – 2016.

- [8] Johnson M. et al. [Google's multilingual neural machine translation system: Enabling zero-shot translation](#) //Transactions of the Association for Computational Linguistics. – 2017. – T. 5. – C. 339-351.
- [9] <https://ru.wikipedia.org/wiki/GPGPU>

Ключевые статьи, упоминаемые в этом видео (от Романа Суворова):

- **Классический Seq2Seq:** Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." *Advances in neural information processing systems*. 2014. <https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>
- **Seq2Seq с механизмом внимания:** Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014). <https://arxiv.org/pdf/1409.0473.pdf>
- **Google Neural Machine Translation:** Wu, Yonghui, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." *arXiv preprint arXiv:1609.08144* (2016). <https://arxiv.org/pdf/1609.08144.pdf>

- Закодировать входную последовательность x_1, x_2, \dots, x_n
- Генерировать выходную последовательность слово за словом
 - $P(y_1 | x_1, x_2, \dots, x_n)$
 - $P(y_2 | x_1, x_2, \dots, x_n, y_1)$
 - $P(y_3 | x_1, x_2, \dots, x_n, y_1, y_2)$
 - и т.д., пока не встретится <EOS>
- Нейросеть задаёт вероятностную модель $P(y_1, y_2, \dots, y_m | x_1, x_2, \dots, x_n; \Theta)$
- Настройка через максимизацию правдоподобия
 $\Theta = \arg \max_{\Theta} \prod_{(x,y)} P(y_1, \dots, y_m | x_1, \dots, x_n; \Theta)$
- Функция правдоподобия - категориальная кросс-энтропия
 $\log P(y_i = y_i^{GT} | x_1, \dots, x_n, y_1, \dots, y_{i-1}) = \log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})_{y_i^{GT}}$



Алгоритм

① Входая последовательность полностью кодируется с помощью RNN (LSTM, GRU)
 $x_1, \dots, x_n \rightarrow c \in \mathbb{R}^D$

② Декодирование - один токен за шаг с помощью RNN (LSTM, GRU)

③ Выход каждого шага - распределение вероятностей токенов

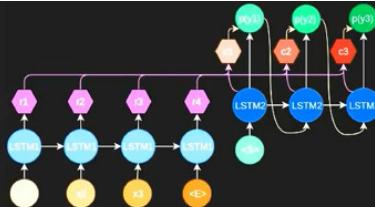
$$P(y_i | x_1, \dots, x_n, y_1, \dots, y_{i-1}) = f(h_i, y_{i-1}, c) = \{p_{i,w}\}_{w \in Vocab}$$

$$0 \leq p_{i,w} \leq 1, \quad \sum_{w \in Vocab} p_{i,w} = 1$$

Анализ

- | | |
|--|---|
| + Простота | - RNN плохо распараллеливается (медленно) |
| - Теряется информация в c | - Низкое разнообразие сгенерированных текстов |
| - Не учитывается правый контекст токенов | |





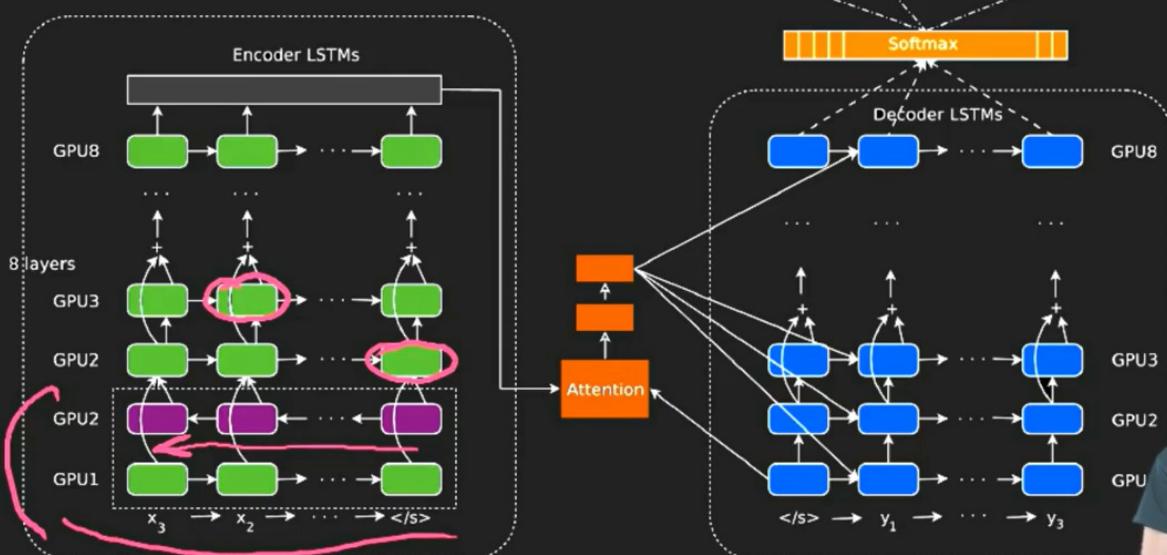
Алгоритм

- ➊ Входная последовательность полностью кодируется с помощью RNN (LSTM, GRU)
 $x_1, \dots, x_n \rightarrow r_1, \dots, r_n \in \mathbb{R}^D$
- ➋ Декодирование - один токен за шаг с помощью RNN (LSTM, GRU)
- ➌ На каждом шаге строится новое представление с помощью механизма внимания
 $c_i = \text{Attention}(r_1, \dots, r_n; h_i)$
- ➍ Выход каждого шага - распределение вероятностей токенов
 $P(y_i|x_1, \dots, x_n, y_1, \dots, y_{i-1}) = f(h_i, y_{i-1}, c_i)$

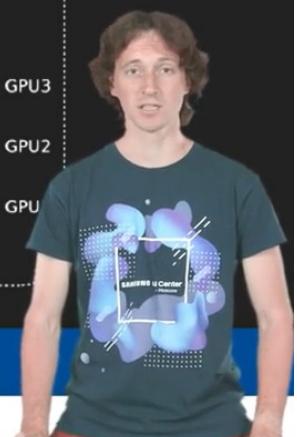


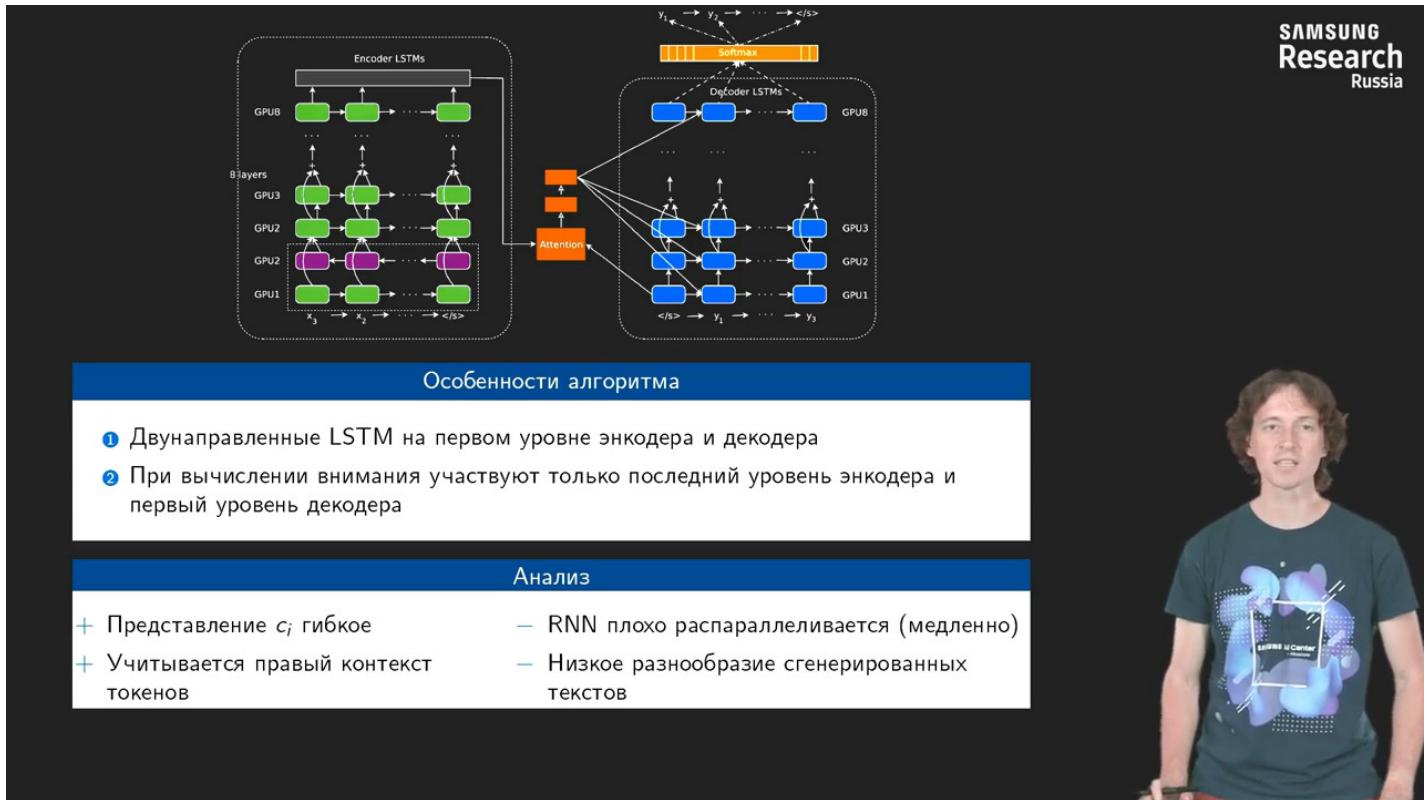
Анализ

- | | |
|--|---|
| + Представление c_i гибкое | - RNN плохо распараллеливается (медленно) |
| - Не учитывается правый контекст токенов | - Низкое разнообразие сгенерированных текстов |



Особенности алгоритма





Особенности алгоритма

- ① Двунаправленные LSTM на первом уровне энкодера и декодера
- ② При вычислении внимания участвуют только последний уровень энкодера и первый уровень декодера

Анализ

- | | |
|---------------------------------------|---|
| + Представление c_i гибкое | - RNN плохо распараллеливается (медленно) |
| + Учитывается правый контекст токенов | - Низкое разнообразие сгенерированных текстов |



Находчивые ребята из фейсбука предложили использовать архитектуру, состоящую целиком из свёрток^[1,2] — свёрточные модули хорошо распараллеливаются, так как зависимости по данным — локальны и укладываются в ширину ядра [свёртки](#). Свёртки абсолютно инвариантны к смещению, к абсолютной позиции относительно начала последовательности. То есть, по умолчанию, они не знают, видят они сейчас слово в начале предложения или в конце предложения. Часто это не очень хорошо. И поэтому, в таких архитектурах используются так называемое "позиционное кодирование", то есть к вектору признаков каждого токена прибавляется какой-то код, который зависит от индекса слова в предложении. Одну возможную схему позиционного кодирования мы рассматривали в лекции про [трансформер](#). Для того, чтобы учесть, с помощью [свёрточных блоков](#), зависимости между словами, находящимися на расстоянии "N", требуется порядка $O(N/K)$ слоёв, где K — это ширина ядра свёртки. При этом, каждый новый слой использует свой набор параметров — они не разделяются между слоями. Это делает обработку длинных зависимостей очень дорогой в смысле использования памяти. Если мы используем прореженные свёртки, то обработка длинных зависимостей становится немного дешевле, но, всё равно — дороже, чем для рекурренток (дороже, в первую очередь, в смысле количества памяти). Такие архитектуры обладают всеми преимуществами вышеописанной архитектуры [Google Machine Translation](#). То есть, они достаточно выразительные, мощные, и на ряде задач они дают лучшее качество, чем [seq2seq](#) модели, основанные на [рекуррентках](#). При этом, в энкодере используется двунаправленный контекст, то есть, для вычисления вектора слова на каком-то уровне, используются как соседи слева, так и соседи справа. [Свёрточные нейросети](#) гораздо

эффективнее с вычислительной точки зрения, то есть, они лучше заполняют вычислительные мощности видеокарт. Однако, как мы уже сказали, длинные зависимости учитывать дорого, а это нужно при обработке текста. И с проблемой разнообразия мы по-прежнему ещё ничего не сделали. Следующее логичное развитие — это использовать трансформер, состоящий целиком из механизмов внутреннего внимания.^[3] Эта архитектура также состоит из двух частей — [энкодера](#) и декодера. В энкодере используется полный механизм внимания, учитывающий все пары элементов последовательности. В декодере у нас ситуация особая — когда мы генерируем очередное слово, мы не видим ничего справа. Мы можем опираться только на часть сгенерированной последовательности слева от текущего слова. Для того, чтобы в режиме обучения игнорировать часть последовательности справа от текущего слова, используется так называемый "[механизм внутреннего внимания](#) с маской". Мaska применяется перед софтмаксами, внутри механизма внимания, и приводит к тому, что некоторые позиции получают нулевой вес и не участвуют в вычислении результирующих векторов. Так же, как и в полносвёрточных архитектурах, здесь необходимо позиционное кодирование для того, чтобы сообщить нейросети информацию о том, где, относительно начала текста, находится каждое входное слово. На практике, часто используется синусоидальный сигнал, который складывается, добавляется к [эмбеддингу](#) токена. Как мы уже говорили в лекции про трансформер и про механизмы внимания, стоимость обработки зависимостей любой длины — константна и не зависит от расстояния между элементами последовательностей, между которыми зависимость существует.

[1] Gehring J. et al. [Convolutional sequence to sequence learning](#) //Proceedings of the 34th International Conference on Machine Learning-Volume 70. – JMLR. org, 2017. – С. 1243-1252.

[2] Gehring J. et al. [A convolutional encoder model for neural machine translation](#) //arXiv preprint arXiv:1611.02344. – 2016.

[3] Vaswani, Ashish, et al. [Attention is all you need](#). Advances in neural information processing systems. 2017.

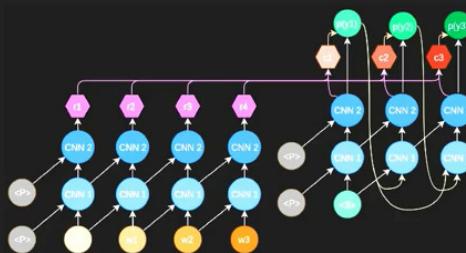
Ключевые статьи, упоминаемые в этом видео (от Романа Суворова):

- **Свёрточный Seq2Seq:** Gehring, Jonas, et al. "Convolutional sequence to sequence learning." *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017. <https://arxiv.org/pdf/1705.03122.pdf>
- **Transformer:** Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems*. 2017. <https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>



Особенности алгоритма

- ➊ Для кодирования и декодирования используются свёртки
- ➋ Позиционное кодирование (эмбеддинги позиций)



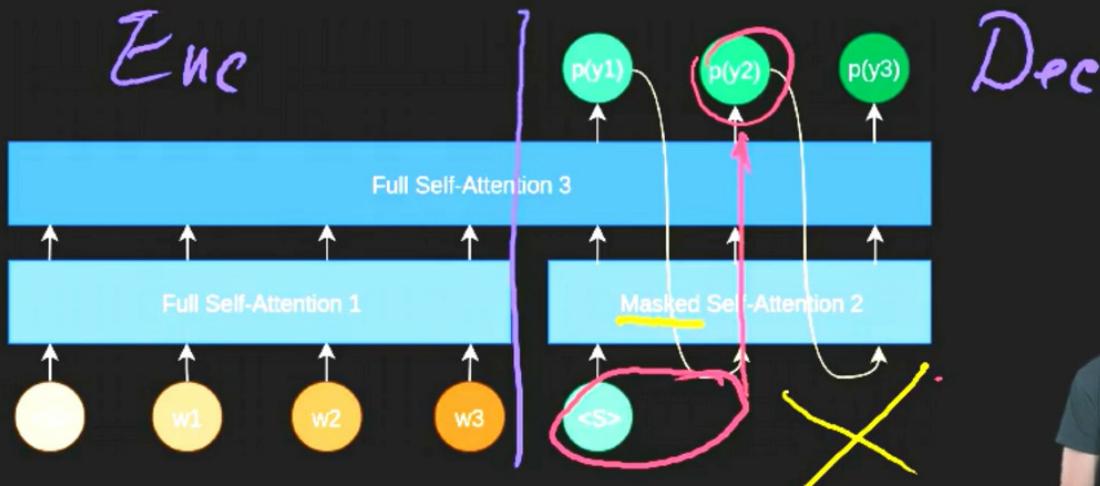
Особенности алгоритма

- ➊ Для кодирования и декодирования используются свёртки
- ➋ Позиционное кодирование (эмбеддинги позиций)
- ➌ Учет зависимостей длины n свёртками с ядром k требует $O(n/k)$ слоёв
- ➍ Прореженные (dilated) свёртки - $O(\log n)$

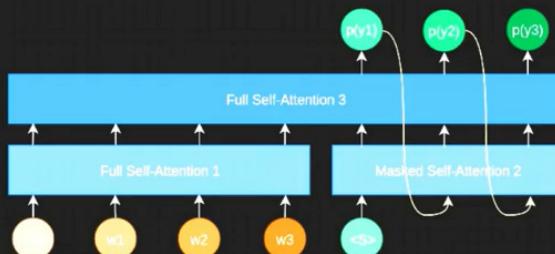


Анализ

- + Представление c_i гибкое
- + Учитывается правый контекст токенов
- + Отлично распараллеливается
- Дорого учитывать длинные зависимости
- Низкое разнообразие сгенерированных текстов



Особенности алгоритма



Особенности алгоритма

- ① Для кодирования и декодирования используются слои с внутренним вниманием (self-attention)
- ② Позиционное кодирование (эмбеддинги позиций или синусоидальный сигнал)
- ③ Учет зависимостей любой длины требует $O(1)$ слоёв



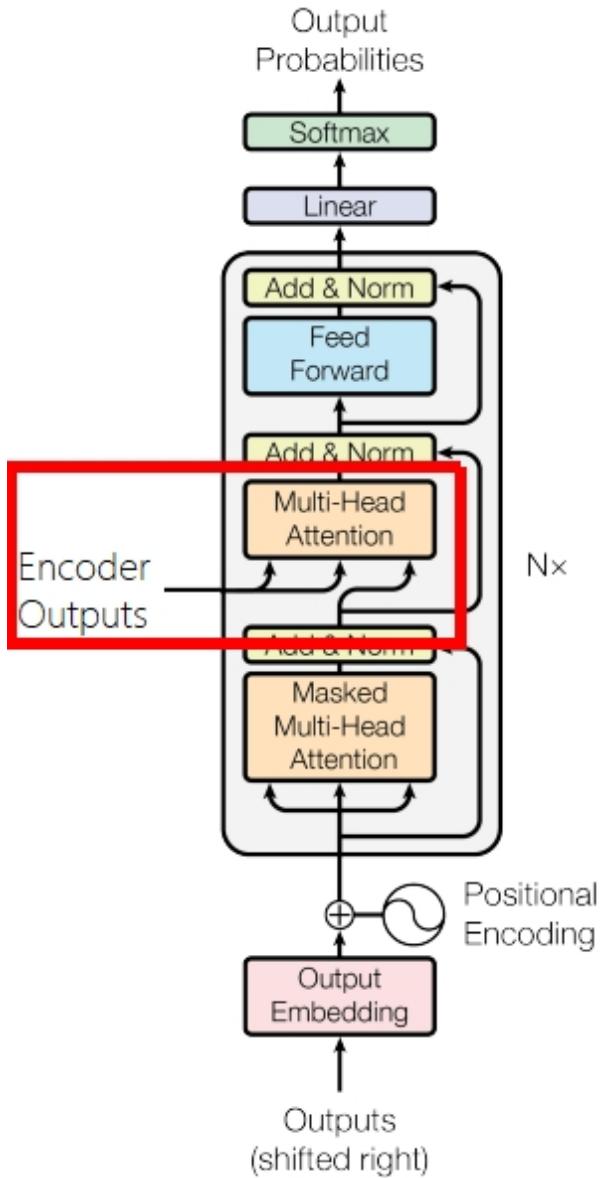
Из комментариев:

Вопрос:

Есть вопрос по визуализации трансформера в конце видео, что это за **общий** блок №3 в трансформер-архитектуре? Такое мы, вроде бы, не проходили, чтоб один и тот же слой и в энкодере и в декодере использовался.

Ответ (Сергея Устянцева):

это не общий блок, это слой encoder-attention из декодера. Ему на входы query и key приходит выход энкодера, а на вход value - нормированный выход со слоя self-attention.



[Трансформер](#), для задач преобразования последовательностей, обладает всеми преимуществами предыдущих архитектур, при этом отлично распараллеливается, учитывается весь контекст каждого слова, зависимости любой длины учитываются за одинаковую цену. Остаётся одна проблема, о которой мы в этой лекции ещё не говорили. Она касается того, как именно мы обучаем всю эту архитектуру и как именно мы декодируем результирующую последовательность. Кстати, несмотря на то, что, в настоящей лекции, я попытался подать материал так, как будто мы постоянно что-то улучшаем, переходим к более совершенной архитектуре, в жизни всё может оказаться не так просто, и разные архитектуры в разных случаях (в разных ролях) могут оказаться лучше или хуже^[1] — например, табличка на слайде показывает сравнение моделей для [машинного перевода](#) между английским и французским языками, причём в моделях использовался [энкодер](#) и декодер разной архитектуры. Transformer

— это трансформер, модель с внутренним вниманием, а RNMT — это [LSTM](#) многослойная. И, как мы видим, самое лучшее качество, пусть и с относительно небольшим отрывом, достигается тогда, когда кодируем исходную последовательность мы с помощью трансформера, а декодируем с помощью рекурренток. Таким образом, в жизни не надо прилипать к какой-то одной архитектуре, всегда имеет смысл попробовать разные варианты. Как же мы декодируем выходную последовательность? Надо вспомнить, что наша модель предсказывает на каждом шаге вероятности для очередного токена. Сначала это условная вероятность первого токена на основе входной последовательности, потом — условная вероятность каждого следующего токена при условии наблюдения входной последовательности и префикса выходной последовательности. Когда мы заканчиваем генерировать выходную последовательность, набор условных распределений складывается в совместное [распределение](#) результирующих токенов при условии всей входной цепочки. Но надо помнить, что это — распределение. То есть мы, пока что, ещё не получили никакую одну конкретную результирующую цепочку. Мы можем из этого распределения получить множество разных цепочек — хороших и не очень. Таким образом задачу генерации текста (декодирования) можно трактовать по-разному. Например, можно просто брать сэмплы, то есть брать реализации вот этой многомерной случайной величины. А можно искать одно наиболее вероятное сочетание токенов. То есть у нас есть противоречие между правдоподобием (когда мы выбираем маленькое число очень хороших вариантов декодирования), и разнообразием (когда мы можем придумать много разных ответов на какой-то вопрос). Например, если наша модель работает как часть чат-бота. Но некоторые из этих декодированных вариантов могут оказаться очень неправдоподобными. Давайте остановимся на втором варианте задачи генерации, когда мы ищем один наиболее правдоподобный набор результирующих токенов. Точное решение этой задачи требует перебора. Эта задача — [NP-полная](#), поэтому её не представляется возможным решить за разумное время. Иногда можно применять полностью жадное решение^[2], то есть на каждом шаге всегда выбирать наиболее правдоподобный токен, идя последовательно слева направо. То есть, сначала, когда мы генерируем первый токен результирующей последовательности, предсказываем такое распределение и выбираем токен — моду этого распределения. Потом мы подставляем этот найденный (наиболее правдоподобный) токен в наш декодер, предсказываем следующее распределение вероятности и снова выбираем его моду, и так далее. Если модель обучилась хорошо, то этого может быть и достаточно. То есть, такой алгоритм может приводить к хорошим вариантам декодированных последовательностей. Однако, полностью жадное решение не позволяет "вернуться и исправить ошибку", то есть, если мы первым токеном сделали неудачный выбор (то есть, например, выбрали конец последовательности сразу), то — всё, мы дальше не идём. На практике же имеет смысл, опять же, найти некий компромисс, то есть баланс между качеством декодирующей последовательности и возможностью, вообще, такую последовательность декодировать. Один алгоритм, который может быть для этого использован — так называемый "[лучевой поиск](#)"

(англоязычный термин — "beam search"). Это эвристический поиск (он, как бы, "умеренно жадный"), он относится к категории "[best first](#) алгоритмов" поиска в графах. Давайте попробуем рассмотреть этот алгоритм более подробно. Для его работы необходима следующая входная информация. Во-первых, нам нужна некоторая функция, которая будет оценивать качество наших последовательностей. Мы можем использовать (как раз) наш декодер для того, чтобы оценивать правдоподобие очередного токена, при условии наблюдения какой-то части последовательности. В результате работы этого алгоритма генерируется последовательность номеров токенов. При этом мы ожидаем, что это сочетание токенов может и не иметь наибольшую из возможных вероятностей генерации (то есть, может не иметь наилучшую оценку в смысле построенной модели), но, на практике, эти последовательности часто получаются достаточно неплохими. Во всяком случае — лучше, чем при декодировании жадным алгоритмом.

[1] Chen, Mia Xu, et al. [The best of both worlds: Combining recent advances in neural machine translation](#). arXiv preprint arXiv:1804.09849 (2018).

[2] Жадное декодирование и декодирование через лучевой поиск (beam-search) рассматривалось [в семинаре про трансформер](#).

Дополнительные комментарии к видео(от Романа Суворова):

- Жадное декодирование и декодирование через лучевой поиск (beam-search) рассматривалось [в семинаре про трансформер](#).

Ключевые статьи, упоминаемые в этом видео (от Романа Суворова):

- **Какой энкодер и декодер лучше?** Chen, Mia Xu, et al. "The best of both worlds: Combining recent advances in neural machine translation." *arXiv preprint arXiv:1804.09849* (2018). <https://arxiv.org/abs/1804.09849>

Энкодер	Декодер	En → Fr BLEU
Transformer	Transformer	40.73 ± 0.19
RNMT+	RNMT+	41.00 ± 0.05
Transformer	RNMT+	<u>41.12 ± 0.16</u>
RNMT+	Transformer	39.92 ± 0.21

LSTM

- Разные архитектуры могут работать лучше в разных ролях

SAMSUNG
Research
Russia

- Предсказываются вероятности для каждого токена
 - $P(y_1|x_1, x_2, \dots, x_n)$
 - $P(y_2|x_1, x_2, \dots, x_n, y_1)$
 - $P(y_3|x_1, x_2, \dots, x_n, y_1, y_2)$
- После обработки последовательности есть оценка распределения токенов в выходной последовательности (правило цепочки)

$$P(y_1, y_2, \dots, y_m|x_1, x_2, \dots, x_n) = \prod_{i=1}^m P(y_i|x_1, x_2, \dots, x_n, y_1, \dots, y_{i-1})$$

SAMSUNG
Research
Russia

- Предсказываются вероятности для каждого токена
 - $P(y_1|x_1, x_2, \dots, x_n)$
 - $P(y_2|x_1, x_2, \dots, x_n, y_1)$
 - $P(y_3|x_1, x_2, \dots, x_n, y_1, y_2)$
- После обработки последовательности есть оценка распределения токенов в выходной последовательности (правило цепочки)

$$P(y_1, y_2, \dots, y_m|x_1, x_2, \dots, x_n) = \prod_{i=1}^m P(y_i|x_1, x_2, \dots, x_n, y_1, \dots, y_{i-1})$$

- Задача генерации
 - $y_1, y_2, \dots, y_m \sim P(y_1, y_2, \dots, y_m|x_1, x_2, \dots, x_n)$
 - $y_1, y_2, \dots, y_m = \arg \max_{y_1, \dots, y_m} P(y_1, y_2, \dots, y_m|x_1, x_2, \dots, x_n)$
- Баланс правдоподобия и разнообразия



Задача декодирования последовательности

$$y_1, y_2, \dots, y_m = \arg \max_{y_1, \dots, y_m} P(y_1, y_2, \dots, y_m | x_1, x_2, \dots, x_n)$$

- Точное решение этой $\arg \max$ задачи требует перебора (NP-полная)
- Полностью жадное решение ($O(m \cdot |\text{Vocab}|)$)

1 $\hat{y}_1 = \arg \max_{y_1} P(y_1 | x_1, x_2, \dots, x_n)$
 2 $\hat{y}_2 = \arg \max_{y_2} P(y_2 | x_1, x_2, \dots, x_n, \hat{y}_1)$
 3 $\hat{y}_3 = \arg \max_{y_3} P(y_3 | x_1, x_2, \dots, x_n, \hat{y}_1, \hat{y}_2)$

- Полностью жадное решение не позволяет вернуться и исправить ошибку



- Полностью жадное решение не позволяет вернуться и исправить ошибку
- Поиск неоптимальных, но достаточно хороших y_1, y_2, \dots, y_m - **beam search**
 - эвристический поиск, "умеренно жадный"



Вход, выход и параметры алгоритма

- Вход - оценка условного распределения $P(y_i | x_1, x_2, \dots, x_n, y_1, \dots, y_{i-1})$
- Выход - последовательность токенов $[\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m]$

У этого алгоритма есть несколько гиперпараметров. Первый (b) — это целое число (как правило, его берут равным примерно 10). Второй гиперпараметр (M) — это максимальная длина решения. Этот параметр ограничивает, как долго мы будем продолжать генерировать последовательности. Первый шаг — инициализация алгоритма. В ходе работы алгоритма мы будем поддерживать два списка. Первый список (это beam) — это множество частичных решений. Это множество состоит из префиксов результирующих текстов. На первом шаге "beam" содержит только b наиболее вероятных первых токенов. То есть это b токенов, с которых вообще может начаться результирующий текст. Второй список, который мы

поддерживаем в процессе работы алгоритма — это список полных решений. То есть "beam" — список частичных решений (список префиксов), и, в ходе работы алгоритма, эти префиксы будут становиться законченными предложениями, и тогда они будут переходить в список результатов. Итак, дальше начинается цикл. Для каждого частичного решения из списка "beam" мы начинаем перебирать все возможные варианты продолжить этот префикс, удлинить его. Таким образом, мы перебираем все возможные токены, каждый токен конкатенируем к концу нашего рассматриваемого частичного решения (таким образом — удлиняем его). Оцениваем правдоподобие полученного нового частичного решения и добавляем это частичное решение в список "beam". Если текущий рассматриваемый вариант продолжения, то есть $у_к_у_k$ равен токену конца последовательности, то мы считаем (вот это) новое частичное решение законченным и помещаем его в список результатов. Дальше, когда мы закончили вот эти два цикла, то есть — когда закончили перебирать все частичные решения из beam и все токены для каждого частичного решения, то мы фильтруем список частичных решений и оставляем в нём только b решений с наилучшей оценкой, то есть b наиболее правдоподобных фрагментов текста. Далее мы повторяем шаги 2 и 3 до тех пор, пока не исчерпаем все возможности перебора (в этом случае у нас список "beam" окажется пустым), а также, пока в списке "beam" есть решения, более короткие чем M . Когда правило останова, описанное на шаге 4 выполнено, мы заканчиваем наш перебор, смотрим список "result" и выбираем из этого списка наилучшее решение, то есть решение с наибольшей оценкой — наиболее правдоподобное предложение. Алгоритм достаточно простой и эффективный. И — да, он не обязательно приводит к оптимальному решению, но он позволяет находить достаточно неплохие варианты декодирования. Но у него есть недостаток — он предпочитает короткие решения. Это не недостаток самого по себе [лучевого поиска](#), это скорее недостаток функции оценки качества частичных решений. Это связано с тем, что, когда мы оцениваем правдоподобие более длинных предложений, мы вынуждены делать больше умножений условных вероятностей. А вероятности — это величины от 0 до 1. То есть, когда мы добавляем очередное умножение к какой-то оценке вероятности, мы не можем увеличить её, она всегда потихонечку уменьшается. Таким образом, более короткие последовательности, как будто бы, более вероятны. Но, с точки зрения здравого смысла, это не вполне так. В качестве решения, обычно, искусственно снижают правдоподобие более коротких решений.

Вход, выход и параметры алгоритма

- Вход - оценка условного распределения $P(y_i|x_1, x_2, \dots, x_n, y_1, \dots, y_{i-1})$
- Выход - последовательность токенов $[\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m]$
- Гиперпараметры b - ширина "луча" (beamsize), M - максимальная длина решения

Алгоритм

- ① Инициализировать список частичных решений (пучок лучей) и список полных решений

$$\text{Beam} = \{[\hat{y}_{i,1}]\}, \quad i = \overline{1, b}, \quad \hat{y}_{i,1} = \arg \max_{y_{i,1}} P(y_1|x_1, x_2, \dots, x_n)$$

$$\text{Result} = \emptyset$$

- ② Для каждого частичного решения $[\hat{y}_{i,1}, \dots, \hat{y}_{i,m_i}] \in \text{Beam}$

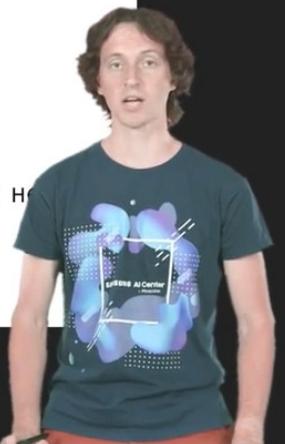
- ① Для каждого токена $y_k \in \text{Vocab}$

- ① Построить продолжение частичного решения $[\hat{y}_{i,1}, \dots, \hat{y}_{i,m_i}, y_k]$
- ② Оценить правдоподобие $P(\hat{y}_{i,1}, \dots, \hat{y}_{i,m_i}, y_k | x_1, x_2, \dots, x_n)$
- ③ Добавить частичное решение в Beam
- ④ Если $y_k = < \text{EOS} >$, считать это решение полным и добавить в Result

- ③ Удалить из Beam все решения, кроме b наиболее правдоподобных

- ④ Повторять шаги 2-3, пока $\text{Beam} \neq \emptyset$ и длина частичных решений в Beam не достигнет M

- ⑤ Выбрать из Result наиболее правдоподобное полное решение



- Простой и эффективный алгоритм, best-first search
- Позволяет найти достаточно хорошие варианты y_1, y_2, \dots, y_m
- Недостаток - предпочитает более короткие решения
- Искусственно занижать правдоподобие более коротких решений $\hat{P}(x) = \sqrt[L(x)]{P(x)}$



Из комментариев:

Вопрос:

А как в beam search мы строим продолжение частичного решения? С помощью тех нейронок, что были показаны ранее (LSTM,CNN, Attention)? И мы получается далее после того как сформировали первые b частичных решений, просто жадно выбираем самый вероятный токен, либо же множим опять на b частичных решений?

Ответ (Алексея Шадрикова):

да, продолжение строим с помощью нейронок. Жадный алгоритм на каждом шаге выбирает только лучшее решение, а лучевой поиск после каждого шага оставляет b наиболее значимых продолжений. Т.е. это некий компромисс между жадным поиском и полным перебором.

Итак, давайте теперь скажем пару слов о проблеме низкого разнообразия. Низкое разнообразие — это когда в разных ситуациях генерируется один и тот же текст. Есть множество гипотез касательно того, почему это может происходить. В этой лекции мы остановимся на двух вариантах. Первый — это слабая обусловленность на вход, то есть связь между энкодером и декодером — слабая. Если используется механизм внимания, то считается, что эта проблема решается (более-менее). Вторая гипотеза связана с чрезмерной уверенностью декодера (английский термин: "over-confidence").^[1] Что значит, что модель очень уверена? Это значит что, когда она предсказывает очередное распределение вероятностей, вероятности токенов либо близки к нулю, либо близки к единице, промежуточные значения редки. Это значит, что распределение, получаемое на очередном

шаге, очень контрастно. Это приводит к тому, что когда в ходе [лучевого поиска](#) мы пытаемся расширить очередное частичное решение с помощью какого-то токена, и этот токен получает очень низкое правдоподобие (по мнению модели), то мы вынуждены отбросить всё это частичное решение, потому что его оценка качества домножается на число близкое к нулю и очень сильно падает. Один возможный способ борьбы с чрезмерной уверенностью — это замена функции потерь. Изначально мы использовали категориальную [кросс-энтропию](#). Её можно записать в виде суммы, каждый элемент суммы — это произведение индикаторной функции (которая равна единице тогда, когда k совпадает с номером истинного токена, который должен получиться на этом шаге согласно обучающей выборке, и нулю во всех остальных случаях) и логарифма вероятности. А вероятность оценивается с помощью нашей нейросети, то есть это некоторое $P(x)$. Первый подход — это "label smoothing"^[2], то есть мы заменяем эту жёсткую индикаторную функцию, которая может принимать только значения 0 или 1, на мягкую индикаторную функцию. Авторы этого подхода предлагают использовать функцию следующего вида. Для "правильного" токена предлагается использовать вес не "единичка", а $1-\beta$ ($\beta=0.1$, например), а для всех остальных токенов — брать не нулевой вес, а априорную вероятность встретить этот токен в тексте. То есть, относительную частоту этого токена в обучающей выборке. Таким образом мы, как бы, сообщаем модели информацию (априорную) об обучающей выборке. Мы говорим, какие слова априорно должны встречаться с большей вероятностью, чаще, а какие — реже. И мы не даём ей стать чрезмерно уверенной, то есть выдавать либо единицу, либо ноль. Похожего эффекта можно добиться и другим из способов — например, добавив к изначальной функции потерь (к кросс-энтропии) вот такую добавку (называется "энтропийная регуляризация"). Энтропия максимальна, когда распределение равномерное, а минимальна — когда распределение вырождено.

[1] [Deep neural networks are easily fooled: High confidence predictions for unrecognizable images](#)

[2] Müller R., Kornblith S., Hinton G. E. [When does label smoothing help?](#) //Advances in Neural Information Processing Systems. – 2019. – С. 4696-4705.

- Низкое разнообразие - в разных ситуациях генерируется один и тот же текст
- Возможные причины:
 - Слабая обусловленность на вход (генератор работает как языковая модель)
 - Чрезмерная уверенность декодера (overconfidence) - распределение $P(y_i|x_1, x_2, \dots, x_n, y_1, \dots, y_{i-1})$ очень контрастное
- Борьба с чрезмерной уверенностью - замена функции потерь



- Борьба с чрезмерной уверенностью - замена функции потерь
- Исходная функция потерь для токена y_i

$$CE(y_i, y_i^{GT}) = -\log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})_{y_i^{GT}} = \\ = - \sum_{k=1}^{|Vocab|} [k = y_i^{GT}] \log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})$$

0 1

- Label smoothing - заменить исходные "hard" метки на "soft" метки

$$SmoothedCE(y_i, y_i^{GT}) = - \sum_{k=1}^{|Vocab|} W(k, y_i^{GT}) \log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})$$

$$W(k, y^{GT}) = \begin{cases} 1 - \beta & k = y^{GT} \\ \beta \cdot P(y^{GT}) & k \neq y^{GT} \end{cases}$$

$\beta = 0.1$



- Борьба с чрезмерной уверенностью - замена функции потерь
- Исходная функция потерь для токена y_i

$$\begin{aligned} CE(y_i, y_i^{GT}) &= -\log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})_{y_i^{GT}} = \\ &= -\sum_{k=1}^{|Vocab|} [k = y_i^{GT}] \log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1}) \end{aligned}$$

- Label smoothing - заменить исходные "hard" метки на "soft" метки

$$SmoothedCE(y_i, y_i^{GT}) = -\sum_{k=1}^{|Vocab|} W(k, y_i^{GT}) \log NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})$$

$$W(k, y^{GT}) = \begin{cases} 1 - \beta & k = y^{GT} \\ \beta \cdot P(y^{GT}) & k \neq y^{GT} \end{cases}$$

- Штраф за контрастность (entropy regularization)

$$\begin{aligned} Loss(y_i, y_i^{GT}) &= CE(y_i, y_i^{GT}) - \beta H(NNet(x_1, \dots, x_n, y_1, \dots, y_{i-1})_{y_i^{GT}}) \\ H(P) &= -\sum_{i=1}^K p_i \log p_i \end{aligned}$$



В этом видео мы поговорили о задаче преобразования последовательностей. В англоязычной литературе такие задачи обычно называют термином [seq2seq](#). Это очень популярная задача, которая, в первую очередь, находит применение в машинном переводе в, диалоговых системах, в суммаризации (упрощённом пересказе) и многих других. В общем виде, архитектура состоит из [энкодера](#) и декодера. Однако, чтобы информация от энкодера лучше передавалась в декодер и вся схема лучше обучалась, часто между энкодером и декодером добавляют механизм внимания. В разное время были предложены разные архитектуры для энкодера и декодера, основанные на [рекуррентных нейросетях](#), на свёрточных, а также на механизме [внутреннего внимания](#) (то есть, [трансформер](#)). Для декодирования последовательностей часто используется алгоритм [лучевого поиска](#), который позволяет найти баланс между полностью жадным решением и полным перебором. Для того, чтобы модель генерировала разнообразные ответы, необходимо ограничивать её уверенность. Для этого используются разнообразные регуляризаторы или альтернативные функции потерь.

- Seq2seq - задача преобразования последовательностей
- Приложения - машинный перевод, чат-боты, суммаризация
- Общий алгоритм - энкодер, декодер
- Механизм внимания
- Разные архитектуры энкодеров и декодеров - RNN, CNN, Transformer
- Декодирование последовательностей - лучевой поиск (beam search)
- Борьба с низким разнообразием - регуляризация, альтернативные функции потерь



5.5 Семинар: генерация кода со Stack Overflow

Для данного семинара Вам потребуется ноутбук [task8_generate_stackoverflow_code.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):

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 на семинаре мы научим [нейронную сеть](#) отвечать на вопросы про языки программирования. Да! Сегодня мы научим нашу нейронную сеть отвечать на вопросы, которые люди часто задают на [StackOverflow](#). И теперь, когда у вас появится вопрос про программирование на python, вы сможете не писать этот вопрос на StackOverflow и ждать несколько дней, пока вы получите правильный ответ от другого программиста, а, вместо этого, спросить что-то у вашей нейронной сети. Ну, и надеяться, что нейронная сеть ответит верно. Итак, что мы сегодня будем делать? Мы обучим sequence to sequence модель, которая, по последовательности токенов одного вида (а именно — последовательности слов естественного языка, из которых состоит вопрос), генерирует последовательность токенов другого вида — а именно, название команд, имена переменных пунктуация... которую мы используем при написании кода. Для простоты мы будем использовать примеры только из одного языка — привычного нам python.

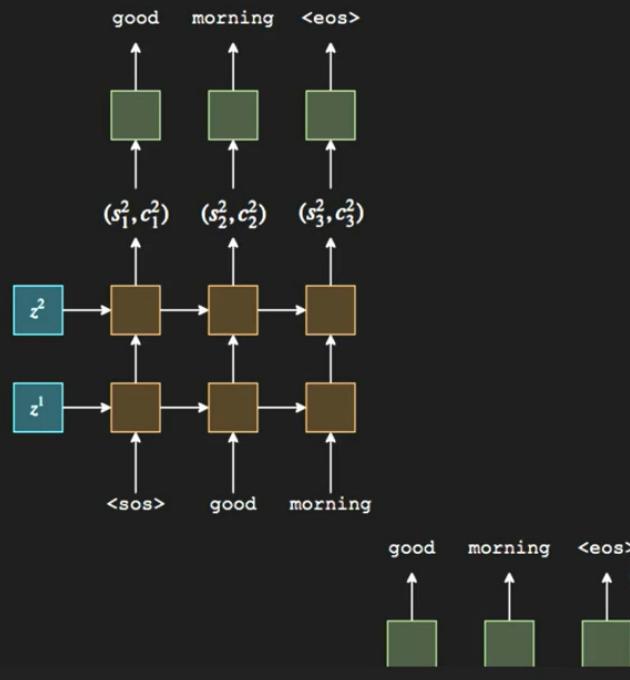
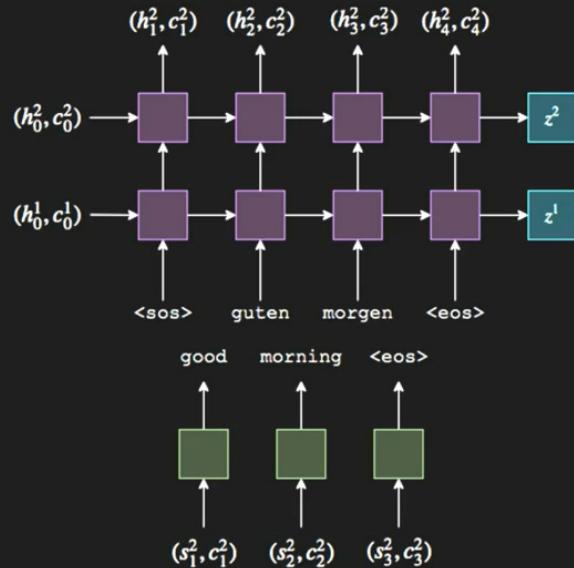
Самые популярные sequence to sequence модели — это модели вида "encoder-decoder", которые разбирались в лекции. Они используют [RNN](#) для того, чтобы закодировать входную последовательность в некоторый вектор. Этот вектор — представление входного предложения в некотором заранее зафиксированном формате. Затем этот вектор декодируется второй RNN ([декодером](#)), который учится предсказывать выходную последовательность, генерируя последовательно токен за токеном. Давайте немного вспомним лекцию и освежим в памяти простой пример с [машиным переводом](#). Перевод вопроса со [StackOverflow](#) на язык python будет проходить по концептуально такой же схеме, как и перевод с немецкого на английский, если мы говорим про использование [seq2seq](#) моделей. На картинке вы можете видеть пример перевода с помощью seq2seq модели. На вход мы подаём фразу "guten morgen" ("доброго утра") — фразу на немецком языке. В энкодер она подается слово за словом. Кроме того, в начало предложения мы добавляем тэг "start of sequence" (коротко — SOS), а в конец мы добавляем токен "end of sequence" (EOS). В каждый момент времени, входом энкодера является текущий токен — назовём его "x", а также некоторое скрытое состояние — назовём его "h" (от слова "hidden"). Причём, в каждый момент времени мы подаём скрытое состояние с предыдущего шага. Выходом будет являться новое скрытое состояние. Скрытое состояние содержит в себе информацию обо всём предложении, которую сеть видела к текущему моменту. Нулевое скрытое состояние можно инициализировать нулями или использовать, например, равномерное [распределение](#). Как только последнее слово было передано в RNN, будем использовать последнее скрытое состояние как вектор, содержащий в себе информацию обо всём предложении. Имея такой вектор, можно начинать декодировать его — генерировать выходную последовательность с помощью декодера. Давайте посмотрим на картинку — в каждый момент времени в мы подаём текущее слово и скрытое состояние с

предыдущего шага. При этом, нулевое скрытое состояние декодера равно последнему скрытому состоянию энкодера. И энкодер и декодер мы можем представить как функции от x и h , то есть от текущего входного слова и скрытого состояния. В декодере на каждом шаге нам нужно предсказывать следующее слово. Для этого будем на каждом шаге пропускать текущее скрытое состояние через [линейный слой](#) и предсказывать следующее слово. После того как мы сгенерирували всю выходную последовательность, мы можем сравнить её с переводом из нашей обучающей выборки. Затем посчитаем [функцию потерь](#) и обновим веса сети, проделав в backward-шаг и посчитав градиент функции потерь. Так — кажется, мы обсудили весь алгоритм. Остаётся только упомянуть о некоторых фишках, которые позволяют сделать процесс обучения декодера чуть более простым и понятным. Например, давайте вспомним что такое "teacher forcing".^[1,2] Это достаточно простая идея. Давайте в качестве некоторых токенов выходной последовательности иногда использовать "[ground truth](#)" из нашего датасета, а иногда — слово предсказанное нашим декодером. Таким образом, наша сеть периодически будет получать некоторую дополнительную информацию из нашей обучающей выборки. Ещё один небольшой трюк — это подход, который касается длины генерируемой последовательности. При генерации выходной последовательности не обязательно ждать, пока модель сгенерирует end-of-sequence токен. Можно, вместо этого, прекратить генерацию, когда мы выдали достаточное количество слов. Например, когда длина выходной последовательности стала примерно равна длине входной последовательности. Это позволит нам избежать слишком долгого обучения, либо генерирования излишнего количества символов в конце нашей последовательности.

[1] Williams R. J., Zipser D. [A learning algorithm for continually running fully recurrent neural networks](#) //Neural computation. – 1989. – Т. 1. – №. 2. – С. 270-280.

[2] Lamb A. M. et al. [Professor forcing: A new algorithm for training recurrent networks](#) //Advances In Neural Information Processing Systems. – 2016. – С. 4601-4609.

Генерация кода по вопросам со StackOverflow с помощью Seq2Seq



Из практического задания:

Что такое teacher forcing?

- метод для обучения RNN, который в некотором (заранее зафиксированном) проценте случаев использует в качестве входа ground truth с предыдущего шага, а не предсказанное сетью значение

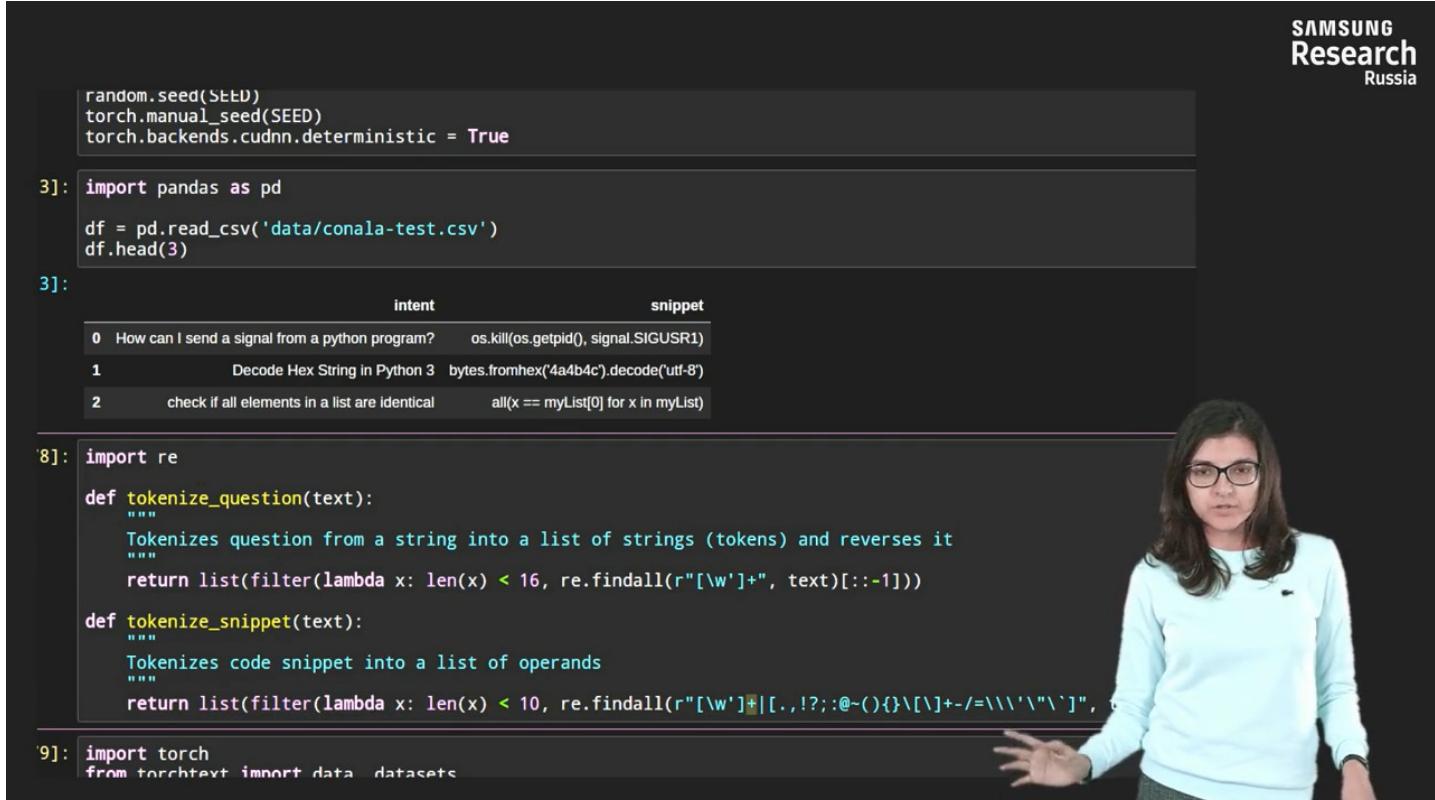
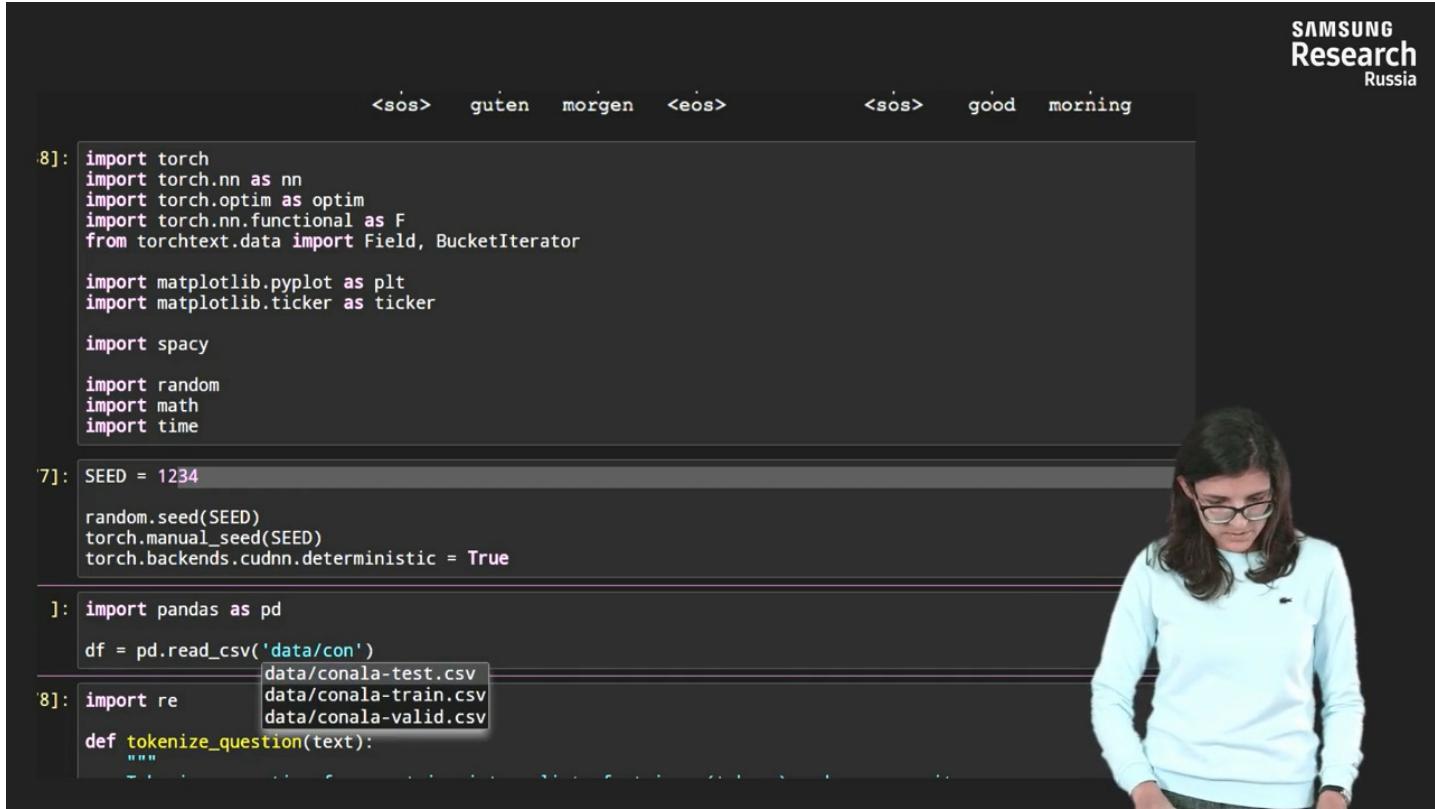
Из комментариев:

Хорошая статья по этой теме - <https://towardsdatascience.com/what-is-teacher-forcing-3da6217fed1c>

Отлично! Теперь давайте закодируем этот алгоритм с помощью библиотек pytorch и "[torchtext](#)". Обучать сеть мы будем не для стандартной (и немного надоевшей) задачи [машинного перевода](#), а для чуть более сложной задачи ответов на вопросы со [StackOverflow](#). Итак, сначала импортируем все библиотеки, которые нам могут понадобиться (это "torch", "torchtext" и некоторые другие библиотеки). После этого задаём "random.seed" для воспроизводимости результатов (вот здесь). И дальше нам нужно посмотреть на формат данных и понять, каким образом мы будем их считывать. Давайте с помощью "[pandas](#)" откроем наши данные и посмотрим, что у нас в них есть. Прочитаем [CSV](#) с помощью библиотеки pandas. Он лежит в папке data. Давайте откроем файл с тестовой выборкой и посмотрим на первые три строки из неё. Мы видим, что в нашем датасете есть два поля: intent и [snippet](#). Intent — это, собственно, вопрос со StackOverflow, а snippet — это кусочек кода, который отвечает на этот вопрос. Теперь нам нужно написать токенизаторы, которые помогут нам поделить на токены вопросы со StackOverflow и кусочек кода. Функция "tokenize_question" токенизирует наш вопрос — делаем это с помощью простой [регулярки](#). Кроме того, мы отрезаем слишком длинные слова, которые, скорее всего, являются названиями веб-сайтов, либо слишком длинными названиями каких-то текстовых полей, или что-то подобное... Кроме того, мы будем подавать наш вопрос в обратном порядке в нашу сеть. То есть, будем начинать с последнего слова и заканчивать первым. Функция "tokenize_snippet" токенизирует кусочек кода. Работает она примерно так же, с помощью чуть другой регулярки, где мы перечисляем знаки пунктуации, которые нас могут интересовать. Отлично, что дальше? Теперь напишем обработчик входных данных с помощью библиотеки "torchtext". Torchtext позволяет сделать это достаточно быстро и с минимальным числом строк кода. [Field](#) позволяет нам определить, как должны обрабатываться данные. Напишем два разных обработчика для вопросов на [естественном языке](#) и для кода. Intent, то есть текстовый вопрос — это исходная последовательность, назовём её "src" (source), а код назовём "trg" (target). Field позволяет задать параметр tokenize, куда мы можем передать наши функции-токенизаторы отдельно для вопроса или для кода. В нашем примере, Field также добавит два дополнительных токена — это "end of sequence" и "start of sequence". Все токены будут приведены к нижнему регистру, а также, для вопроса, мы будем учитывать длину этого вопроса. Таким образом, "field" будет возвращать нам пары, а именно — вопрос и его длина. В дальнейшем это нам понадобится для обучения [seq2seq](#) модели. Далее разделим наши данные на обучающую, валидационную и тестовую выборки. Это уже сделано за нас, наша выборка поделена на 3 CSV-файла, и всё, что нам нужно — это, всего лишь, скачать эти данные с помощью, например, "data.[TabularDataset](#)". Данные мы собрали из следующего источника — он называется [CoNaLa](#)-corpus, это объединённый проект лаборатории из Carnegie Mellon и лаборатории со смешным названием STRUDEL. Этот датасет достаточно маленький, включает в себя всего 2.5 тысячи примеров. Мы поделили их

следующим образом: 2000 на обучение, примерно 350 на валидацию и 500 на тест. Кроме того, данные с похожей тематикой можно найти, загуглив следующее сочетание слов: "StackOverflow question code [dataset](#)". Это датасет, намайненный автоматически со StackOverflow, с помощью "Bi-View Hierarchical Neural Network". Статья про "bi view hierarchical neural network" была представлена в 2018 году^[1], её авторы — сотрудники университетов Огайо, Вашингтона и компании Fujitsu, и их датасет включает в себя гораздо больше данных (там есть около 150 тысяч пар вопрос и сниппет на python и около 120 тысяч пар вопросов и ответов на SQL). Для обучения мы оставили только пары из первого датасета, так как он был собран вручную, гораздо менее шумный и позволяет обучить seq2seq модели с большим качеством. Второй датасет необходимо предварительно чистить, чтобы получить гораздо более хорошее качество. Вы можете попробовать выкачать этот датасет и обучить модель с использованием большего количества данных и посмотреть на результат — посмотреть на метрики, которые вас получатся.

[1] Yao Z. et al. [Staqc: A systematically mined question-code dataset from stack overflow](#) //Proceedings of the 2018 World Wide Web Conference. – 2018. – С. 1693-1703.



```

9]: import torch
from torchtext import data, datasets

SRC = data.Field(
    tokenize = tokenize_question,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True,
    include_lengths = True
)

TRG = data.Field(
    tokenize = tokenize_snippet,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True
)

fields = {
    'intent': ('src', SRC),
    'snippet': ('trg', TRG)
}

train_data, valid_data, test_data = data.TabularDataset.splits(
    path = 'data',
    train = 'conala-train.csv',
    validation = 'conala-valid.csv',
    test = 'conala-test.csv',
    format = 'csv',
    fields = fields
)

```

0]: SRC.build_vocab([train_data.src, valid_data.src, test_data.src], max_size=25000, min_freq=3)



Из комментариев:

Комментарий:

В первой ячейке надо заменить

```

from torchtext.datasets import TranslationDataset, Multi30k
from torchtext.data import Field, BucketIterator

```

на:

```

from torchtext.legacy.datasets import TranslationDataset, Multi30k
from torchtext.legacy.data import Field, BucketIterator

```

И в 4-й ячейке добавить legacy в import:

```

import torch
from torchtext.legacy import data, datasets

SRC = data.Field(...)

```

Вопрос:

Мы решили токенизировать предложение в обратном порядке

```

8]: import re

def tokenize_question(text):
    """
        Tokenizes question from a string into a list of strings (tokens) and reverses it
    """
    return list(filter(lambda x: len(x) < 16, re.findall(r"[\w']+", text)[::-1]))

```

Но вопрос а зачем? Далее мы использовали двунаправленный GRU (LSTM) - а он идет в двух направлениях. Поэтому смысл первоначальной токенизации в обратном порядке.

Далее потом когда мы предиктили(произвольное предложение) - мы вопрос не переворачивали. А когда я попробовал вопрос все-таки перевернуть как мы и обучали - то предикт остался ровно такой же.

Ответ (Анастасии Яниной):

подробное объяснение, почему имеет смысл подавать токены source (но не target) предложения в обратном порядке, приводится в статье "["Sequence to Sequence Learning with Neural Networks"](#)". Приведу цитату оттуда:

We found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier. While we do not have a complete explanation to this phenomenon, we believe that it is caused by the introduction of many short term dependencies to the dataset.

Интуитивное объяснение примерно такое:

Пусть у нас есть source-предложение из трех токенов $\langle a, b, c \rangle$ и target-предложение $\langle a', b', c' \rangle$ (a' - соответствующий перевод токена a и т.д.)

Теперь конкатенируем токены source в прямом порядке с токенами target в прямом порядке (получаем $\langle a \ b \ c \ a' \ b' \ c' \rangle$), каждый токен из source далек от соответствующего ему токена из target (от a до a' расстояние 3, от b до b' тоже 3 и т.д.).

Если мы поменяем порядок токенов из source с прямого на обратный, то получим $\langle c \ b \ a \ a' \ b' \ c' \rangle$, то есть для токенов a и b расстояние до соответствующего перевода стало меньше. А значит, для начальных токенов из source-предложения LSTM будет проще увидеть взаимосвязь с соответствующими токенами из target. В свою очередь правильный перевод начальных токенов может повлиять на правильность перевода следующих токенов в предложении.

Таким образом, сокращение расстояний от первых токенов source-предложения до соответствующих им переводов в target-предложении помогает улучшить качество работы рекуррентной сети за счет появления большего количества "краткосрочных" зависимостей. И хотя LSTM и GRU и так борются с проблемой затухания градиентов в RNN, описанная выше техника помогает еще лучше решать проблему "забывания" при работе с долгосрочными зависимостями.

Доп. комментарий (задающего вопрос):

Да спасибо. Для одностороннего LSTM это вполне разумно. Но у нас двунаправленный(nn.GRU(emb_dim, enc_hid_dim, **bidirectional = True**)) . И у нас и так слой проходит $a \rightarrow b \rightarrow c$ и одновременно ним $c \rightarrow b \rightarrow a$. Тут получается от перестановки слагаемых сумма не меняется.

Ответ (Анастасии Яниной):

да, в случае bidirectional LSTM и правда можно не менять порядок токенов. В этой задаче хотелось продемонстрировать, что подход с изменением порядка токенов существует, поэтому

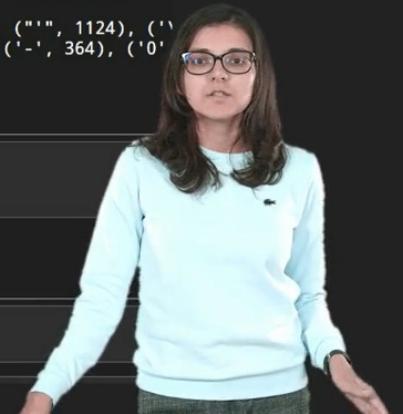
и добавили его сюда. Попробуйте сделать **bidirectional = False** и поучить сетку с прямым и обратным порядком подачи токенов.

Теперь сформируем словарь. Задаём максимально возможный размер слова и минимальную встречаемость слова для того, чтобы попасть в словарь. Для вопроса мы выбираем минимальную встречаемость равную "3", для сниппета с кодом — "5". Эти числа можно варирировать и постараться подобрать такие, чтобы словарь выглядел наиболее адекватно — чтобы в нём не было мусорных слов, и чтобы, при этом, не отрезались слова, которые действительно несут некоторый смысл и помогут нам обучить более адекватную модель. Кроме того, для того, чтобы использовать паддинг, "[torchtext](#)" требует, чтобы все элементы в батче были отсортированы по их длине до применения паддинга, в убывающем порядке. То есть, первая последовательность должна быть самой длинной. Отлично, после создания словарей давайте немного провалидируем наш код. Посмотрим, сколько уникальных токенов в словаре интентов и в словаре сниппетов. Оказывается, что их 754 и 551 (выглядит вполне адекватно для выборки, состоящей из чуть более чем 2000 вопросно-ответных пар). Также мы можем посмотреть на размеры наших обучающей, валидационной и тестовой выборки. Последний этап нашей простой предобработки данных — это создать итераторы. Мы хотим, чтобы итератор возвращал батчи с данными, у которых есть атрибут "src" — это [тензоры](#), кодирующие входные предложения (вопросы на [естественном языке](#)) и атрибут "trg" (target) — это тензоры, кодирующие сниппеты с кодом. Под кодированием здесь понимается простое сопоставление токена его числовому индексу. Это соответствие, собственно, и прописано в слове. Удобно, что torchtext-итераторы умеют автоматически добавлять паддинг (делать все последовательности одной длины). Мы будем использовать "BucketIterator" (здесь), а не обычный итератор, потому что он минимизирует количество паддинга для входных и выходных предложений, что достаточно удобно в нашей задаче. Ну, и также нам нужно определить, на каком девайсе мы будем обсчитывать нашу модель. Если у вас есть [GPU](#) — замечательно, мы можем указать, на какой именно GPU мы хотим обучать (если у вас их больше чем одна). Если же её нет, модель вполне можно обучить на [CPU](#), просто это будет чуть-чуть дольше. Но и для такой маленькой модели на таком маленьком количестве данных CPU будет вполне достаточно, чтобы получить хороший результат. Теперь давайте подробнее рассмотрим код [энкодера](#) и декодера для нашей модели. Итак, энкодер, в нашем случае — это двухслойная [LSTM](#). В статье, которая упоминается в начале семинара, использовалась четырёхслойная сеть, но мы попробуем сэкономить время на обучении сети и обучим двухслойную сетьку. Для многослойной LSTM входная последовательность идёт в первый слой сети, а скрытое состояние первого слоя используется как входная последовательность следующего слоя. Скрытое состояние первого слоя можно представить формулой, зависящей от входных токенов и от предыдущего скрытого состояния. Напомню что, в отличие от [RNN](#), LSTM, кроме того, что берёт на вход предыдущее скрытое состояние и возвращает следующее, ещё и принимает на вход так называемое "cell state". Его обычно обозначают буквой "c". Можно воспринимать его как другой вид скрытого состояния. В

итоге, конечное представление входной последовательности в виде вектора будет конкатенацией скрытого состояния и нашего cell state, которое будем обозначать буквой "с".

SAMSUNG
Research
Russia

```
)  
  
0]: SRC.build_vocab([train_data.src, valid_data.src, test_data.src], max_size=25000, min_freq=3)  
print(SRC.vocab.freqs.most_common(20))  
  
TRG.build_vocab([train_data.trg, valid_data.src, test_data.trg], min_freq=5)  
print(TRG.vocab.freqs.most_common(20))  
  
print(f"Уникальные токены в словаре интентов: {len(SRC.vocab)}")  
print(f"Уникальные токены в словаре синипетов: {len(TRG.vocab)}")  
  
[('a', 1818), ('in', 1359), ('python', 1324), ('to', 1222), ('how', 911), ('of', 884), ('list', 786), ('stri  
5), ('the', 480), ('from', 361), ('with', 328), ('pandas', 273), ('i', 262), ('get', 227), ('dictionary', 21  
nvert', 193), ('do', 181), ('values', 173), ('by', 169), ('into', 164)]  
[(')', 4412), ('(', 4407), ('.', 3266), ('', 2394), ('[', 1425), (']', 1424), ('=', 1177), ('"', 1124), (''  
9), ('in', 796), (':', 729), ('"', 687), ('x', 646), ('for', 581), ('a', 519), ('1', 473), ('-', 364), ('0'  
('/), 318), ('i', 297)]  
Уникальные токены в словаре интентов: 754  
Уникальные токены в словаре синипетов: 551  
  
1]: print(f"Размер обучающей выборки: {len(train_data.examples)}")  
print(f"Размер валидационной выборки: {len(valid_data.examples)}")  
print(f"Размер тестовой выборки: {len(test_data.examples)}")  
  
Размер обучающей выборки: 2000  
Размер валидационной выборки: 379  
Размер тестовой выборки: 500  
  
2]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
torch.cuda.set_device(1)
```



SAMSUNG
Research
Russia

```
Размер тестовой выборки: 500  
  
2]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
torch.cuda.set_device(1)  
  
3]: BATCH_SIZE = 512  
  
train_iterator, valid_iterator, test_iterator = BucketIterator.splits(  
    (train_data, valid_data, test_data),  
    batch_size = BATCH_SIZE,  
    sort_within_batch = True,  
    sort_key = lambda x : len(x.src),  
    device = device)
```

Encoder

```
4]: class Encoder(nn.Module):  
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):  
        super().__init__()  
  
        self.input_dim = input_dim  
        self.emb_dim = emb_dim  
        self.enc_hid_dim = enc_hid_dim  
        self.dec_hid_dim = dec_hid_dim  
        self.dropout = dropout  
  
        self.embedding = nn.Embedding(input_dim, emb_dim)  
        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)  
        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)
```



Из комментариев:

Комментарий:

если кому будет интересно. ну мало ли :) турориал по которому сделан этот семинар. <https://github.com/bentrevett/pytorch-seq2seq/blob/master/1%20->

[%20Sequence%20to%20Sequence%20Learning%20with%20Neural%20Networks.ipynb](#) и в нем есть ссылка на работу <https://arxiv.org/abs/1409.3215> - именно в ней говориться о 4-х слойной LSTM. вот как звучит оригинальная фраза из туториала. "First, the encoder, a 2 layer LSTM. The paper we are implementing uses a 4-layer LSTM, but in the interest of training time we cut this down to 2-layers. The concept of multi-layer RNNs is easy to expand from 2 to 4 layers."

Вопрос:

Вот тут BATCH_SIZE = 512

Но с этим значением colab вылетает и 256 тоже вылетал и с 32 вылетал

CUDA out of memory. Tried to allocate 3.36 GiB (GPU 0; 15.90 GiB total capacity; 11.12 GiB already allocated; 1.97 GiB free; 2.10 GiB cached)

и с 16 вылетал

```
: BATCH_SIZE = 512

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    sort_within_batch = True,
    sort_key = lambda x : len(x.src),
    device = device)
```

на полном датасете где 50000 примеров

Ответ (Романа Суворова):

в [репозитории](#) сейчас значение по умолчанию BATCH_SIZE=2. С этим значением на Колабе работает. Сейчас запустил на колабе с BATCH_SIZE=512 и не упало. На какой ячейке падает? Можно перестартовать ноутбук (Runtime -> Restart runtime) - Pytorch не умеет освобождать память без перезапуска процесса, особенно когда много раз досрочно прерываешь обучение. А ещё на колабе [иногда достаётся уже частично занятая](#) GPU. В этом случае попробуйте прекратить сессию (Runtime -> Manage sessions -> Terminate) и запуститься заново, тогда есть шанс попасть на другую виртуалку. Количество памяти и загрузку GPU можно увидеть, выполнив !nvidia-smi в ячейке ноутбука.

Доп. комментарий (задающего вопрос):

Вылетает в цикле обучения на первой эпохе

```
for epoch in range(N_EPOCHS):
```

Стало работать при BATCH_SIZE = 4

Но это повторюсь на 50 000 наборе. На маленьком не проверял - там смотрел как пример как репозитории.

Ответ (Романа Суворова):

размер датасета по идеи не влияет на потребление памяти. А попробуйте перед выполнением этой строчки в предыдущую ячейку вставить !nvidia-smi - может быть правда Вам достаётся

наполовину занятая видеокарта?

Вопрос:

А зачем мы строим словарь по валидационной и тестовой выборке? Если токена в обучающей выборке нет, но он есть в тестовой. Модель то все равно его не знает и не видела на обучении и в итоге может "отчубучить" какую нибудь "хренъ".

Ответ (Анастасии Яниной):

спасибо за внимательность, поправили ноутбук. Включение в словарь слов из тестовой выборки и правда излишне (если тестовая выборка большая, это еще и сильно замедлит процесс построения словаря). Кроме того, это некоторая "нечестность", не надо использовать тестовые данные в момент построения модели.

Итак, создаём модуль "encoder". Он наследуется от "torch.[nn.module](#)" и [энкодер](#) принимает на вход следующие параметры. Это входная размерность данных — это размерности наших "[one-hot](#)" векторов, которая совпадает с размерностью словаря интентов; это размерность [эмбеддингов](#), размерность слоя с эмбеддингами — например, можно сделать его равным 100 или 200, любому другому числу, которое кажется вам наиболее подходящим. Есть параметр, который называется "encoding hidden dimension" (вот он) — это размерность скрытого состояния; есть то же самое для декодера; и "[dropout](#)" — это количество дропаута, которое мы будем использовать. Здесь мы должны задать число от 0 до 1. Слой эмбеддингов создаётся с помощью "nn.embedding" (вот здесь). Дальше мы используем [GRU](#)-слой, вместо него можно с таким же успехом использовать [LSTM](#). Дропаут будем добавлять между слоями нашей многослойной сети, то есть между скрытыми состояниями, которые идут на вход слою "2". Дальше — давайте рассмотрим метод "forward". В метод "forward" мы передаём входное предложение, которое превращено в dense-вектора с помощью эмбеддинг-слоя, и затем применяем дропаут (вот здесь). После того, как мы передали всю входную последовательность в [RNN](#), она автоматически посчитает скрытое состояние по всей последовательности. Мы не передаём то, чем нужно инициализировать скрытое состояние. По дефолту [тензоры](#) инициализируются нулями, что нас вполне устраивает в этом примере. Сеть нам возвращает две вещи — это "outputs" и скрытый слой. Размерности каждого тензора прописаны в коде в виде комментариев, что упрощает процесс дебага и его осмысление. Ещё один параметр, который нужно упомянуть, называется "bidirectional". Он отвечает за то — двунаправленная или односторонняя сеть используется нашем примере. Кроме того, в нашем примере мы используем [attention](#). В этом модуле мы будем считать веса attention.

```
Encoder:
4]: class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.input_dim = input_dim
        self.emb_dim = emb_dim
        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim
        self.dropout = dropout

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_len):

        #src = [src sent len, batch size]
        #src_len = [src sent len]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src sent len, batch size, emb dim]

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, src_len)

        packed_outputs, hidden = self.rnn(packed_embedded)

        #packed outputs is a packed sequence containing all hidden states
```



```
Encoder:
4]: class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        #embedded = [src sent len, batch size, emb dim]

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, src_len)

        packed_outputs, hidden = self.rnn(packed_embedded)

        #packed_outputs is a packed sequence containing all hidden states
        #hidden is now from the final non-padded element in the batch

        outputs, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs)

        #outputs is now a non-packed sequence, all hidden states obtained
        # when the input is a pad token are all zeros

        #outputs = [sent len, batch size, hid dim * num directions]
        #hidden = [n layers * num directions, batch size, hid dim]

        #hidden is stacked [forward_1, backward_1, forward_2, backward_2, ...]
        #outputs are always from the last layer

        #hidden [-2, :, :] is the last of the forwards RNN
        #hidden [-1, :, :] is the last of the backwards RNN

        #initial decoder hidden is final hidden state of the forwards and backwards
        # encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)))

        #outputs = [sent len, batch size, enc hid dim * 2]
        #hidden = [batch size, dec hid dim]

    return outputs, hidden
```



Из комментариев:

Вопрос:

В видео говорится что dropout будет включаться между слоями LSTM, но в коде этого не видно.

Ответ (Сергея Устянцева):

Дропаут будем добавлять между слоями нашей многослойной сети, то есть между скрытыми состояниями, которые идут на вход слою "2".

Первый слой - nn.embedding, второй - nn.GRU. Вот между ними дропаут и включается.

Доп. комментарий (задающего вопрос):

на самом деле я думаю, что скрипт менялся и уже не соответствует записи лекции.

Первоначально, наверно, там было несколько слоев в GRU, параметр **num_layers = 2**, и в этом случае был **dropout** внутри GRU. А потом **num_layers** сделали = 1, **dropout** убрали. Ведь фраза содержит слово "скрытое состояние", а оно есть только внутри GRU. Но, может все так как Вы написали.

Ответ (Сергея Устьянцева):

Дропаут в рекуррентных сетях - это отдельная тема. Разрывая рекуррентные связи, мы фактически лишаем сеть памяти.

Вопрос/Комментарий:

Как вы, всё-таки, вольно обращаетесь с интерфейсами классов! torch.nn.Module.__init__ имеет единственный аргумент self, а у вас в каждом дочернем классе свой уникальный набор обязательных параметров. Насколько я понял из кода, для этого предназначен метод register_parameter(self, name, param)

Ответ (Романа Суворова):

нет, инициализатор __init__ предназначен ровно для того, для чего мы его используем - для инициализации всех необходимых полей объекта. Добавлять обязательные параметры в инициализатор в дочернем классе абсолютно нормально. Однако Вы правы в том, что менять сигнатуры публичных методов в дочерних классах часто плохая идея (но __init__ - это не обычный метод)

В нашем примере мы будем использовать [attention](#). На коде attention не будем останавливаться подробно, скажем лишь, что здесь мы будем использовать стандартный attention, который посчитает нам веса по нашей входной последовательности. Теперь давайте перейдём к коду [декодера](#). Архитектура декодера аналогична архитектуре энкодера — это [GRU](#) unit, первый слой получает пару с прошлого шага и прогоняет её через нашу сеть вместе с текущим токеном, чтобы предсказать новую пару — таким образом, уравнения декодера будут очень похожи на уравнения энкодера. Затем мы прогоняем наше скрытое состояние с верхнего слоя через [линейный слой](#), чтобы сделать предсказание следующего токена в выходном генерируемом предложении. Входные аргументы класса "decoder" похоже на аргументы класса "encoder", исключения — у нас здесь есть параметр, который называется "output_dimension" (это размер [one-hot](#) векторов, которые подаются на вход декодеру). Это число должно быть равно размеру словаря таргета. Forward-метод декодера принимает батч входных данных, а также скрытое состояние с предыдущего шага. Мы применяем unsqueeze к выходным токенам, чтобы добавить ещё одну размерность. Далее, аналогично энкодеру, применяем [эмбеддинг](#)-слой и дропаут. Дальше применяем attention, и потом батч с токенами передаём в [RNN](#) вместе с "h" и "c" векторами с предыдущего шага. Аналогично энкодеру, мы получаем на выходе

output (это скрытое состояние с верхнего слоя нашей сети), новое скрытое состояние и новое cell состояние, то есть новые вектора "h" и "c". И мы прогоняем output (после того, как избавились от лишней размерности) через линейный слой, чтобы получить предсказание следующего слова в нашей последовательности. И возвращать здесь мы будем, собственно — output, скрытое состояние и вектор "a". Последняя часть нашей имплементации — создаём модуль для [seq2seq](#) модели. Она будет содержать внутри себя энкодер, декодер и уметь обрабатывать входные последовательности (то есть вопросы) и выдавать снппеты в качестве ответов. Входные параметры seq2seq модели — это энкодер, декодер, device, а также токены, с помощью которых мы кодируем паддинг, начало последовательности и конец последовательности. В этой имплементации их можно менять. Нам нужно убедиться, что количество слоёв и размерность скрытых состояний — одинакова для энкодера и декодера. Это не обязательное условие, но в противном случае (в случае разного количества слоёв) нам придётся придумывать некоторые способы это обойти. Например, если в энкодере два слоя, а в декоре один, то можно использовать оба вектора, либо можно их усреднить. "forward"-метод в классе seq2seq берёт на вход вопрос, снппет с кодом и "teacher forcing ratio". На выход он отдаёт нам output, а также веса attention, которые мы посчитали в модуле attention. Теперь мы можем инициализировать модель. Как сказано ранее, входная и выходная размерность определяются размерами словарей. Сделаем так, чтобы количество слоёв у энкодера и декодера, а также размерности векторов скрытых состояний были одинаковыми. Давайте зададим размерность скрытого состояния равную "100", размерность для эмбеддингов — "256". Также зададим [dropout](#) — он будет достаточно большим ("0.8"). Далее мы увидим, что если ставить дропаут достаточно маленьким, то сеть будет плохо учиться и результат будет достаточно некрасивым. Теперь инициализируем веса для обеих моделей. В статье, которую мы пытаемся имплементировать, брали равномерное [распределение](#) от -0.08 до +0.08. Создадим функцию "init_weights" и добавим её в нашу модель с помощью метода apply. Равномерное распределение мы берём из [nn.init.uniform](#) (то есть, берём просто равномерное распределение из pytorch). Также посчитаем количество параметров, которое содержит наша модель. У нас получилось около 1 миллиона параметров — достаточно много, но замечательно, что у нас достаточно высокий процент дропаута, это поможет нам не [переобучиться](#). В качестве оптимизатора будем использовать [Adam](#), в качестве loss-функции — кросс энтропию. Теперь мы можем написать функцию, которая задаст наш цикл обучения. Сначала мы переводим модель в training mode (с помощью model.train). Это включит dropout и батч-нормализацию (если бы она была, но, в нашем случае, мы её не используем). А потом будем итерироваться через батч-итератор. Что мы делаем на каждой итерации? На каждой итерации мы берём входное и выходное предложение из батча, вместо входного предложения мы получаем пару — "входное предложение и его длина" (как мы уже обсуждали ранее). Далее мы делаем [zero_grad](#) — обнуляем градиенты, посчитанные на предыдущем шаге, затем мы передаём source и target в нашу модель и получаем некоторый выход и веса attention, мы считаем градиенты с помощью "loss backward", предварительно посчитав [функцию потерь](#), и дальше клипаем (clip) градиенты, делаем шаг нашим оптимизатором и считаем лосс.

Замечательно, на выход наша функция будет возвращать нормализованный лосс по нашей эпохе. Также у нас есть функция, которую мы будем использовать для оценивания качества модели (для evaluate). Здесь, вначале, мы переводим нашу модель в состояние оценивания качества (это означает "выключить dropout", "выключить батч-нормализацию") и дальше проделываем примерно такие же шаги, как и в функции "train". Кроме того, давайте напишем функцию, которая будет замерять время, потраченное на каждую эпоху, и назовём её "epoch_time".

```
#outputs = [sent len, batch size, enc hid dim * 2]
#hidden = [batch size, dec hid dim]

return outputs, hidden

[285]: class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim

        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim)
        self.v = nn.Parameter(torch.rand(dec_hid_dim))

    def forward(self, hidden, encoder_outputs, mask):

        #hidden = [batch size, dec hid dim]
        #encoder_outputs = [src sent len, batch size, enc hid dim * 2]
        #mask = [batch size, src sent len]

        batch_size = encoder_outputs.shape[1]
        src_len = encoder_outputs.shape[0]

        #repeat encoder hidden state src_len times
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)

        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        #hidden = [batch size, src sent len, dec hid dim]
        #encoder_outputs = [batch size, src sent len, enc hid dim * 2]
```



```
#repeat encoder hidden state src_len times
hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)

encoder_outputs = encoder_outputs.permute(1, 0, 2)

#hidden = [batch size, src sent len, dec hid dim]
#encoder_outputs = [batch size, src sent len, enc hid dim * 2]

energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim = 2)))

#energy = [batch size, src sent len, dec hid dim]

energy = energy.permute(0, 2, 1)

#energy = [batch size, dec hid dim, src sent len]

#v = [dec hid dim]

v = self.v.repeat(batch_size, 1).unsqueeze(1)

#v = [batch size, 1, dec hid dim]

attention = torch.bmm(v, energy).squeeze(1)

#attention = [batch size, src sent len]

attention = attention.masked_fill(mask == 0, -1e10)

return F.softmax(attention, dim = 1)
```



Decoder

Decoder

```
[286]: class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()

        self.emb_dim = emb_dim
        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim
        self.output_dim = output_dim
        self.dropout = dropout
        self.attention = attention

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim)

        self.out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs, mask):

        #input = [batch size]
        #hidden = [batch size, dec hid dim]
        #encoder_outputs = [src sent len, batch size, enc hid dim * 2]
        #mask = [batch size, src sent len]

        input = input.unsqueeze(0)

        #input = [1, batch size]
```



```
def forward(self, input, hidden, encoder_outputs, mask):

    #input = [batch size]
    #hidden = [batch size, dec hid dim]
    #encoder_outputs = [src sent len, batch size, enc hid dim * 2]
    #mask = [batch size, src sent len]

    input = input.unsqueeze(0)

    #input = [1, batch size]

    embedded = self.dropout(self.embedding(input))

    #embedded = [1, batch size, emb dim]
    [
        a = self.attention(hidden, encoder_outputs, mask)

        #a = [batch size, src sent len]
        a = a.unsqueeze(1)

        #a = [batch size, 1, src sent len]
        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        #encoder_outputs = [batch size, src sent len, enc hid dim * 2]
        weighted = torch.bmm(a, encoder_outputs)

        #weighted = [batch size, 1, enc hid dim * 2]
        weighted = weighted.permute(1, 0, 2)
```



```

#weighted = [batch size, 1, enc hid dim * 2]
weighted = weighted.permute(1, 0, 2)
#weighted = [1, batch size, enc hid dim * 2]
rnn_input = torch.cat((embedded, weighted), dim = 2)
#rnn_input = [1, batch size, (enc hid dim * 2) + emb dim]
output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))

#output = [sent len, batch size, dec hid dim * n directions]
#hidden = [n layers * n directions, batch size, dec hid dim]

#sent len, n layers and n directions will always be 1 in this decoder, therefore:
#output = [1, batch size, dec hid dim]
#hidden = [1, batch size, dec hid dim]
#this also means that output == hidden
assert (output == hidden).all()

embedded = embedded.squeeze(0)
output = output.squeeze(0)
weighted = weighted.squeeze(0)

output = self.out(torch.cat((output, weighted, embedded), dim = 1))

#output = [bsz, output dim]

return output, hidden.squeeze(0), a.squeeze(1)

```



```

[287]: class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, pad_idx, sos_idx, eos_idx, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.pad_idx = pad_idx
        self.sos_idx = sos_idx
        self.eos_idx = eos_idx
        self.device = device

    def create_mask(self, src):
        mask = (src != self.pad_idx).permute(1, 0)
        return mask

    def forward(self, src, src_len, trg, teacher_forcing_ratio = 0.5):

        #src = [src sent len, batch size]
        #src_len = [batch size]
        #trg = [trg sent len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use teacher forcing 75% of the time

        if trg is None:
            assert teacher_forcing_ratio == 0, "Must be zero during inference"
            inference = True
            trg = torch.zeros((100, src.shape[1])).long().fill_(self.sos_idx).to(src.device)
        else:
            inference = False

        batch_size = src.shape[1]
        max_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

```



```

else:
    inference = False

batch_size = src.shape[1]
max_len = trg.shape[0]
trg_vocab_size = self.decoder.output_dim

#tensor to store decoder outputs
outputs = torch.zeros(max_len, batch_size, trg_vocab_size).to(self.device)

#tensor to store attention
attentions = torch.zeros(max_len, batch_size, src.shape[0]).to(self.device)

#encoder_outputs is all hidden states of the input sequence, back and forwards
#hidden is the final forward and backward hidden states, passed through a linear layer
encoder_outputs, hidden = self.encoder(src, src_len)

#first input to the decoder is the <sos> tokens
output = trg[0,:]

mask = self.create_mask(src)

#mask = [batch size, src sent len]

for t in range(1, max_len):
    output, hidden, attention = self.decoder(output, hidden, encoder_outputs, mask)
    outputs[t] = output
    attentions[t] = attention
    teacher_force = random.random() < teacher_forcing_ratio
    top1 = output.max(1)[1]
    output = (trg[t] if teacher_force else top1)
    if inference and output.item() == self.eos_idx:
        return outputs[:t], attentions[:t]

```



```

for t in range(1, max_len):
    output, hidden, attention = self.decoder(output, hidden, encoder_outputs, mask)
    outputs[t] = output
    attentions[t] = attention
    teacher_force = random.random() < teacher_forcing_ratio
    top1 = output.max(1)[1]
    output = (trg[t] if teacher_force else top1)
    if inference and output.item() == self.eos_idx:
        return outputs[:t], attentions[:t]

return outputs, attentions

```

[375]:

```

INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 100
DEC_HID_DIM = 100
ENC_DROPOUT = 0.8
DEC_DROPOUT = 0.8
PAD_IDX = SRC.vocab.stoi['<pad>']
SOS_IDX = TRG.vocab.stoi['<sos>']
EOS_IDX = TRG.vocab.stoi['<eos>']

attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT, attn)

model = Seq2Seq(enc, dec, PAD_IDX, SOS_IDX, EOS_IDX, device).to(device)

```



[376]:

```

def init_weights(m):
    for name, param in m.named_parameters():
        if ... in name:

```

```
[375]: INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 100
DEC_HID_DIM = 100
ENC_DROPOUT = 0.8
DEC_DROPOUT = 0.8
PAD_IDX = SRC.vocab.stoi['<pad>']
SOS_IDX = TRG.vocab.stoi['<sos>']
EOS_IDX = TRG.vocab.stoi['<eos>']

attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT, attn)

model = Seq2Seq(enc, dec, PAD_IDX, SOS_IDX, EOS_IDX, device).to(device)
```

```
[376]: def init_weights(m):
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.normal_(param.data, mean=0, std=0.01)
        else:
            nn.init.constant_(param.data, 0)

model.apply(init_weights)

[376]: Seq2Seq(
    (encoder): Encoder(
        (embedding): Embedding(754, 256)
        (rnn): GRU(256, 100, bidirectional=True)
        (fc): Linear(in_features=200, out_features=100, bias=True)
        (dropout): Dropout(p=0.8, inplace=False)
    )
    (decoder): Decoder(
        (attention): Attention(
            (attn): Linear(in_features=300, out_features=100, bias=True)
        )
        (embedding): Embedding(551, 256)
        (rnn): GRU(456, 100)
        (out): Linear(in_features=556, out_features=551, bias=True)
        (dropout): Dropout(p=0.8, inplace=False)
    )
)
```

```
[377]: def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'Модель содержит {count_parameters(model)} параметров')

Модель содержит 1,073,487 параметров
```

Then we define our optimizer and criterion. We have already initialized PAD_IDX when initializing the model, so we don't need to do it again.

```
[378]: optimizer = optim.Adam(model.parameters())

[379]: criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

[380]: def train(model, iterator, optimizer, criterion, clip):
    model.train()
    epoch_loss = 0
    for i, batch in enumerate(iterator):
```



```
(decoder): Decoder(
    (attention): Attention(
        (attn): Linear(in_features=300, out_features=100, bias=True)
    )
    (embedding): Embedding(551, 256)
    (rnn): GRU(456, 100)
    (out): Linear(in_features=556, out_features=551, bias=True)
    (dropout): Dropout(p=0.8, inplace=False)
)
```

```
[377]: def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'Модель содержит {count_parameters(model)} параметров')

Модель содержит 1,073,487 параметров
```

Then we define our optimizer and criterion. We have already initialized PAD_IDX when initializing the model, so we don't need to do it again.

```
[378]: optimizer = optim.Adam(model.parameters())

[379]: criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

[380]: def train(model, iterator, optimizer, criterion, clip):
    model.train()
    epoch_loss = 0
    for i, batch in enumerate(iterator):
```



```

model.train()
epoch_loss = 0
for i, batch in enumerate(iterator):
    src, src_len = batch.src
    trg = batch.trg
    optimizer.zero_grad()
    output, attention = model(src, src_len, trg, 0.4)
    #trg = [trg sent len, batch size]
    #output = [trg sent len, batch size, output dim]
    output = output[1:].view(-1, output.shape[-1])
    trg = trg[1:].view(-1)
    #trg = [(trg sent len - 1) * batch size]
    #output = [(trg sent len - 1) * batch size, output dim]
    loss = criterion(output, trg)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
    optimizer.step()
    epoch_loss += loss.item()
return epoch_loss / len(iterator)

```



```

epoch_loss += loss.item()
return epoch_loss / len(iterator)

```

```

[381]: def evaluate(model, iterator, criterion):
    model.eval()
    epoch_loss = 0
    with torch.no_grad():
        for i, batch in enumerate(iterator):
            src, src_len = batch.src
            trg = batch.trg
            output, attention = model(src, src_len, trg, 0) #turn off teacher forcing
            #trg = [trg sent len, batch size]
            #output = [trg sent len, batch size, output dim]
            output = output[1:].view(-1, output.shape[-1])
            trg = trg[1:].view(-1)
            #trg = [(trg sent len - 1) * batch size]
            #output = [(trg sent len - 1) * batch size, output dim]
            loss = criterion(output, trg)
            epoch_loss += loss.item()

```



```
#output = [trg sent len, batch size, output dim]
output = output[1:].view(-1, output.shape[-1])
trg = trg[1:].view(-1)

#trg = [(trg sent len - 1) * batch size]
#output = [(trg sent len - 1) * batch size, output dim]

loss = criterion(output, trg)
epoch_loss += loss.item()

return epoch_loss / len(iterator)

[352]: def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

[382]: N_EPOCHS = 50
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)
    ...
```



Из комментариев:

Комментарий:

Подача материала оставляет желать лучшего: где "h" и где "c" взяли, что за суммирование размерностей на входах слоев, аттеншн вообще пропустили и т.д. и т.д.

В целом, нет связки слов со строками кода, как это происходит в лекциях Романа Суворова, бегаем по диагонали. Сложно отвыкать от хорошей подачи материала 😞

Ответ (Сергея Устьянцева):

в данном семинаре код большей частью заимствован из [этого ноутбука](#). Возможно, после его внимательного просмотра вопросы у вас отпадут.

Комментарий:

Просмотрев реализацию attention блока, понимаешь, что она как-то не совсем совпадает с той, которая была объяснена в рамках этого курса.

Разве не должно быть что-то такое на t-ом шаге:

1. Получаем s - скрытое состояние декодера с t-1 шага, а также все скрытые состояния h_i , полученные для каждого слова енкодера
2. Вычисляем вектора ключей (K) и значений (V) для всех h_i , вектор запроса (q) для s
3. Тогда веса значимости каждого h_i для s: $w = \text{softmax}(K * q)$
4. Итоговый вектор = $w * V$

В действительности же в реализации, представленной здесь, происходят несколько другие вещи и не совсем понятно как сопоставить её с описанной выше.

Понятно, что в целом можно по разному реализовывать это все, и в целом в коде прослеживается логика, но как-то она не сходится с стандартом, по крайней мере, насколько я

это вижу.

Возможно я где-то что-то не понял и упустил. Если так, буду очень благодарен, если меня поправят.

Но так или иначе, мне кажется, что лучше было бы объяснить, как здесь реализован attention. Лично мне этого не хватило.

Ответ (Алексея Шадрикова):

спасибо. Трансформер вообще не самая простая архитектура и у меня самого к нему много вопросов. Но он работает, а это главное. Что же касается механизма внимания, то это лишь скалярное произведение софтмакса на вектор, т.е. пункт 4 в Вашем списке плюс, отчасти, пункт 3. Это так или иначе присутствует во всех архитектурах, а в остальном "стандарт" нет, есть некоторые работающие практики.

Но да, объяснение, видимо, вышло недостаточно понятным, просим прощения.

Отлично! Наконец, мы можем начать тренировать нашу модель. На каждой итерации будем печатать [перплексию](#) (увидеть разницу в перплексии гораздо проще, чем в [кросс-энтропии](#), поскольку она по порядку больше). На каждой итерации мы будем печатать перплексию, а в конце каждой эпохи будем проверять, получила ли наша модель score лучше, чем во всех предыдущих эпохах. Если это произошло, мы обновим переменную `best_valid_loss` и сохраним параметры лучшей модели с помощью вызова функции `state dict`. Потом, когда мы будем оценивать качество моделей на тестовой выборке, загрузим параметры той модели, которая давала наилучший валидационный score (сделать мы это сможем с помощью вот этого кода). Давайте более внимательно посмотрим на процесс обучения модели. На [GPU](#) модель обучалась очень быстро. Вы можете увидеть, что время на обучение одной эпохи составило меньше одной секунды (это просто замечательно). Если обучать на [CPU](#), то времени это займет достаточно много — на одну эпоху будет уходить примерно 10-20 секунд, в зависимости от конфигурации компьютера, на котором это обсчитывается. Давайте посмотрим, что происходило с лоссом на обучении и валидации. На обучении наш лосс падает, на валидации — тоже. При этом, если мы посмотрим, что происходит в процессе обучения — под конец обучения лосс на валидации практически перестаёт падать и, в принципе, выходит на некоторое плато — начинает флюктуировать. Лосс на трейне всё ещё продолжает падать, но, при этом, тоже начинает немного флюктуировать. Отлично, мы обучили модель, теперь мы можем посмотреть, какой лосс она даёт на тестовой выборке — он примерно равен лоссу на валидации и перплексия на teste — выше, чем перплексия на обучении. В принципе, это достаточно ожидаемый результат. Теперь мы можем написать функцию, которая будет генерировать [кодовый сниппет](#) по нашему вопросу. И наконец — посмотреть, какое же качество получилось у нашей модели. Напишем функцию "translate sentence", которая будет по токенизированному вопросу выдавать ответ на этот вопрос на языке программирования. То есть, выдавать кодовый сниппет, который отвечает на наш вопрос. И, также, напишем функцию, которая будет визуализировать веса [attention](#), то есть, мы сможем посмотреть, какие

слова имеют больший вес относительно каких слов. Давайте возьмём 3 произвольных примера из обучающей, валидационной и тестовой выборки, и посмотрим, что у нас получилось. Мы берём пример из обучающей выборки. Вопрос звучит как "how to convert today daytime string back to datetime object". OK... Ответ должен быть примерно вот таким — то есть мы должны использовать модуль "datetime", вызвать оттуда функцию "strptime". Наша сеть смогла предсказать, что нам нужно использовать функцию daytime.strptime, что достаточно хорошо, но, при этом, у неё в словаре нету вот этих чисел, которые, скорее всего, встретились в нашей выборке один или два раза, и поэтому не попали в наш словарь, и из-за этого наша сеть не может ничего написать в скобках функции — она не имеет в словаре вот этих символов. Достаточно ожидаемый результат. Если мы посмотрим на матрицу attention — мы видим, что слова datetime являются ключевыми в этом вопросе, остальные слова практически никакого веса не вносят. OK, теперь давайте посмотрим на какой-нибудь вопрос из валидационной и тестовой выборки. Из за валидационной мы берём вопрос "python convert a tuple to string" (OK, ответ должен быть примерно таким), Выглядит достаточно странно, но, в валидационной выборке он был именно таким. При этом, эта выборка собиралась руками, так что, наверное, какая-то логика в этом есть. Наша сеть предсказывает вот такой ответ. Он не очень логичный, и он даже синтаксически никак не согласовывается, то есть здесь закрывается круглая скобка, хотя должна бы закрыться квадратная. Так что можно считать, что это не очень хороший пример. Здесь я постаралась показать не только хорошие примеры, которые обучились плюс-минус адекватно, но и любые остальные, чтобы было видно, что, всё-таки, сеть на таком маленьком количестве данных и проучившись небольшое количество эпох обучилась неидеально. Ну и давайте посмотрим на какой-нибудь пример из тестовой выборки, например — вопрос про то, как работать с кодировками [UTF-8](#) (кодировкой, которая помогает нам работать с русскими символами). Ответ должен быть вот таким: сначала сделать "decode", потом "encode". Наша сеть смогла предсказать decode (декодирование), но, при этом, почему-то она здесь поставила слова "soup". Возможно, как-то это коррелирует с библиотекой [beautifulsoup](#) и, как-то, нашей сети показалось, что это здесь будет уместно. Если мы посмотрим на матрицу attention, то и здесь важно было только слово "UTF" — кажется, что если говорить про декодирование то слово "UTF" очень хорошо связано со словом "soup" (видимо, это отсылка к "beautifulsoup") и со словом "decode", с остальными скобками и "unknown token" немного — тоже. В предыдущем примере матрица attention выглядела примерно так же, то есть — слово "string", которое здесь было, видимо, ключевым, по мнению нашей сети было связано достаточно плотно, практически, со всеми остальными символами в нашем примере. Но учитывая, что это не слишком удачный пример, эта матрица не несёт достаточно информации.

```
[352]: def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

[382]: N_EPOCHS = 50
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'conala_model_attention_test.pt')

    print(f'Эпоха: {epoch+1:02} | Время: {epoch_mins}m {epoch_secs}s')
    print(f'Перплексия (обучение): {math.exp(train_loss):7.3f}')
    print(f'Перплексия (валидация): {math.exp(valid_loss):7.3f}')


Перплексия (валидация): 537.779
Эпоха: 02 | Время: 0m 0s
```



```
start_time = time.time()

train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
valid_loss = evaluate(model, valid_iterator, criterion)

end_time = time.time()

epoch_mins, epoch_secs = epoch_time(start_time, end_time)

if valid_loss < best_valid_loss:
    best_valid_loss = valid_loss
    torch.save(model.state_dict(), 'conala_model_attention_test.pt')

print(f'Эпоха: {epoch+1:02} | Время: {epoch_mins}m {epoch_secs}s')
print(f'Перплексия (обучение): {math.exp(train_loss):7.3f}')
print(f'Перплексия (валидация): {math.exp(valid_loss):7.3f}')


Эпоха: 01 | Время: 0m 0s
Перплексия (обучение): 547.415
Перплексия (валидация): 537.779
Эпоха: 02 | Время: 0m 0s
Перплексия (обучение): 522.874
Перплексия (валидация): 485.212
Эпоха: 03 | Время: 0m 0s
Перплексия (обучение): 437.391
Перплексия (валидация): 335.544
Эпоха: 04 | Время: 0m 0s
Перплексия (обучение): 240.795
Перплексия (валидация): 118.726
Эпоха: 05 | Время: 0m 0s
Перплексия (обучение): 87.500
Перплексия (валидация): 66.904
Эпоха: 06 | Время: 0m 0s
```



```
Эпоха: 4 / | Время: 0m 0s
Перплексия (обучение): 27.928
Перплексия (валидация): 43.511
Эпоха: 48 | Время: 0m 0s
Перплексия (обучение): 29.995
Перплексия (валидация): 42.771
Эпоха: 49 | Время: 0m 0s
Перплексия (обучение): 27.796
Перплексия (валидация): 43.022
Эпоха: 50 | Время: 0m 0s
Перплексия (обучение): 28.532
Перплексия (валидация): 43.254
```

```
[383]: model.load_state_dict(torch.load('conala_model_attention_test.pt'))

test_loss = evaluate(model, test_iterator, criterion)

print(f'Перплексия (валидация): {math.exp(test_loss):.3f}')

Перплексия (валидация): 45.399
```

Предсказание кода по вопросу

```
[361]: def translate_sentence(model, sentence):
    model.eval()
    tokenized = tokenize_question(sentence)
    tokenized = ['<sos>'] + [t.lower() for t in tokenized] + ['<eos>']
    numericalized = [SRC.vocab.stoi[t] for t in tokenized]
    sentence_length = torch.LongTensor([len(numericalized)]).to(device)
    tensor = torch.LongTensor(numericalized).unsqueeze(1).to(device)
    translation_tensor_logits, attention = model(tensor, sentence_length, None, 0)
    translation_tensor = torch.argmax(translation_tensor_logits.squeeze(1), 1)
```



```
[361]: def translate_sentence(model, sentence):
    model.eval()
    tokenized = tokenize_question(sentence)
    tokenized = ['<sos>'] + [t.lower() for t in tokenized] + ['<eos>']
    numericalized = [SRC.vocab.stoi[t] for t in tokenized]
    sentence_length = torch.LongTensor([len(numericalized)]).to(device)
    tensor = torch.LongTensor(numericalized).unsqueeze(1).to(device)
    translation_tensor_logits, attention = model(tensor, sentence_length, None, 0)
    translation_tensor = torch.argmax(translation_tensor_logits.squeeze(1), 1)
    translation = [TRG.vocab.itos[t] for t in translation_tensor]
    translation, attention = translation[1:], attention[1:]
    return translation, attention

[362]: def display_attention(candidate, translation, attention):
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)

    attention = attention.squeeze(1).cpu().detach().numpy()

    cax = ax.matshow(attention, cmap='bone')

    ax.tick_params(labelsize=15)
    ax.set_xticklabels([''] + ['<sos>'] + [t.lower() for t in tokenize_question(candidate)] + ['<eo'])
    ax.set_yticklabels([''] + translation)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
    plt.close()
```



```
    rotation=45)
ax.set_yticklabels([''] + translation)

ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

plt.show()
plt.close()
```

```
[363]: example_idx = 2

src = ' '.join(vars(train_data.examples[example_idx])['src'])
trg = ' '.join(vars(train_data.examples[example_idx])['trg'])

print(f'src = {src}')
print(f'trg = {trg}')

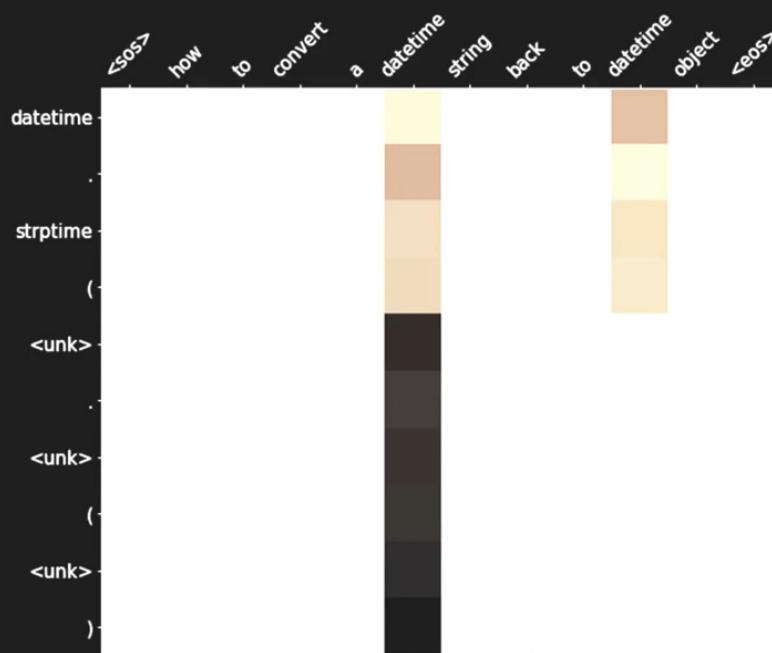
src = object datetime to back string datetime a convert to how
trg = datetime . strftime ( '2010 - 11 - 13 10 : 33 : 54 . 227806' , ' y - m - d h : m : s . f' )
```

```
[364]: translation, attention = translate_sentence(model, src)

print('predicted trg = ', ' '.join(translation))

display_attention(src, translation, attention)

predicted trg =  datetime . strftime ( <unk> . <unk> ( <unk> )
```



```
[371]: example_idx = 8
src = ''.join(vars(valid_data.examples[example_idx])['src'])
trg = ''.join(vars(valid_data.examples[example_idx])['trg'])

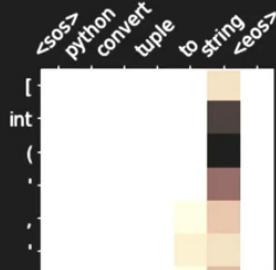
print(f'src = {src}')
print(f'trg = {trg}')

src = string to tuple convert python
trg = " " " " ". join ( 'a' , 'b' , 'c' , 'd' , 'g' , 'x' , 'r' , 'e' ) )
```

```
[372]: translation, attention = translate_sentence(model, src)
print('predicted trg = ', ''.join(translation))

display_attention(src, translation, attention)

predicted trg = [ int ( ' , ' , ' , ' , ' , ' ) for i in s ]
```



```
[373]: example_idx = 4
src = ''.join(vars(test_data.examples[example_idx])['src'])
trg = ''.join(vars(test_data.examples[example_idx])['trg'])

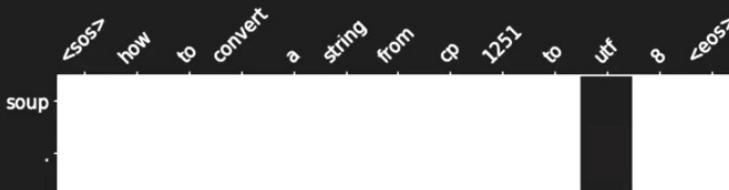
print(f'src = {src}')
print(f'trg = {trg}')

src = 8 utf to 1251 cp from string a convert to how
trg = d . decode ( 'cp1251' ) . encode ( 'utf8' )
```

```
[374]: translation, attention = translate_sentence(model, src)
print('predicted trg = ', ''.join(translation))

display_attention(src, translation, attention)

predicted trg = soup.decode(<unk>)
```



```
print(f'trg = {trg}')
src = 8 utf to 1251 cp from string a convert to how
trg = d . decode ( 'cp1251' ) . encode ( 'utf8' )

[374]: translation, attention = translate_sentence(model, src)
print('predicted trg = ', ''.join(translation))
display_attention(src, translation, attention)
predicted trg =  soup.decode(<unk>)
```



Итак, какой вывод мы можем сделать из этого ноутбука? [seq2seq](#) модели достаточно просто обучать. Можно использовать уже готовые модули из pytorch — например, [GRU](#) юниты или [LSTM](#). Но, при этом, если мы используем недостаточно большое количество данных, либо мы учим модель недостаточно долго, либо мы подобрали неправильно параметры — например, использовали мало дропаута или наша модель [переобучилась](#) — мы не получим какого-то очень крутого результата. При этом, задача, для решения которой мы пытались здесь обучить сеть, достаточно сложная, и генерировать код по словесному описанию проблемы на [естественном языке](#) — это сложная задача, в принципе, даже для человека, не то что для нейронной сети. При этом, наша сеть научилась генерировать, в принципе, какие-то логичные ответы, то есть, зачастую, всплывают ключевые слова, которые действительно относятся к сути проблемы. Если использовать больший объём обучающих данных, а также немного поиграть с параметрами сети — наверняка, мы сможем получить гораздо более красивый результат. Этим вы можете заняться в качестве домашней работы. Но, кроме того, домашней работой будет — написать seq2seq модель, но для гораздо более привычной и простой задачи — перевода предложений с немецкого на английский, которую мы разбирали в начале ноутбука. Удачи с домашней работой!

Из практической задачи:

BLEU (bilingual evaluation understudy) - метрика для оценивания качества машинного перевода, основанная на сравнении перевода, предложенного алгоритмом, и референсного перевода (ground truth). Сравнение производится на основе подсчета n-грамм (n меняется от 1 до некоторого порога, например, 4), которые встретились и в предложенном переводе, и в

референсном (ground truth). После подсчета совстречаемости n-грамм полученная метрика умножается на так называемый brevity penalty - штраф за слишком короткие варианты перевода. Brevity penalty считается как <количество слов в переводе, предложенном алгоритмом> / <количество слов в референсном переводе>.

Формула:

$$BLEU = \text{brevity penalty} \cdot \left(\prod_{i=1}^n \text{precision}_i \right)^{1/n} \cdot 100\%$$

, где brevity penalty = $\min \left(1, \frac{\text{output length}}{\text{reference length}} \right)$

Пример:

SYSTEM A: [Israeli officials] responsibility of [airport] safety
2-GRAM MATCH 1-GRAM MATCH

REFERENCE: Israeli officials are responsible for airport security

SYSTEM B: [airport security] [Israeli officials are responsible]
2-GRAM MATCH 4-GRAM MATCH

Metric	System A	System B
precision (1gram)	3/6	6/6
precision (2gram)	1/5	4/5
precision (3gram)	0/4	2/4
precision (4gram)	0/3	1/3
brevity penalty	6/7	6/7
BLEU	0%	52%