

# Stepik. Neural networks and NLP. 4. Language models and text generation - Part 3

---

## 4.6 Семинар: моделирование языка с помощью Transformer

#####

Для данного семинара Вам потребуется ноутбук [task5\\_text\\_transformer.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 `dlnlutils`

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 у нас насыщенный семинар. Мы попробуем охватить темы [моделирования языка](#), механизмов внимания и рассмотрим модную архитектуру [трансформер](#). Проверять работоспособность методов сегодня мы будем с помощью известного произведения Льва Николаевича Толстого "Война и мир". Итак, загрузим обучающую выборку. В нашем случае, обучающая выборка — это просто большой текстовый файл без всякой разметки. При загрузке текста мы читаем весь его в память, а потом нарезаем на кусочки размером в 200 символов. Таким образом мы получим набор небольших фрагментов текста, но, при этом, длина каждого фрагмента будет больше, чем длина отдельного предложения. Современные языковые модели работают с более длинными

последовательностями, поэтому общепринятая схема — это "не выполнять разбиение текста на отдельные предложения перед подачей их в языковую модель". Всего у нас получилось около 8 тысяч фрагментов. На экране вы видите один такой фрагмент. Он содержит фрагменты двух предложений. Как обычно, разобьём все наши данные на обучающую и тестовую выборку в соотношении 70% в обучающую выборку, 30% в тестовую. Современные языковые модели работают, как правило, не с целыми токенами. Они работают с фрагментами слов, или, так называемыми "sub-word units" (по сути, это [N-граммы](#) символов). Поэтому для токенизации мы используем не классический токенизатор с помощью регулярных выражений или каких-то других правил (например, как те которые мы использовали в предыдущих семинарах). Здесь мы используем алгоритм "[byte pair encoding](#)". Напомню, в двух словах, как он работает, в чём его основной принцип. Допустим, у нас есть последовательность символов: "ABCABE", например. Тогда, сначала, этот алгоритм будет искать наиболее частотную биграмму (в данном случае это "AB"), и он заменит её в тексте на какой-то новый символ, который в тексте ранее не встречался. Например, мы получим "XCXE". Такая же операция поиска наиболее частотной биграммы и замены её на новый символ будет повторяться в цикле. Например, на следующем шаге мы заменим биграмму "XC" на какой-нибудь символ "Y" и получим новую последовательность, и так далее. Таким образом мы последовательно скжимаем текст и, в процессе сжатия текста, запоминаем те замены, которые мы сделали. Например, мы можем запомнить, что "AB" было заменено на "X", а "XC" заменено было на "Y". Такой алгоритм позволяет получить нечто среднее между алгоритмами, работающими на уровне отдельных токенов, и на уровне отдельных символов. Когда мы работаем с отдельными символами, у нас алфавит маленький, то есть решать задачу классификации нам попроще, но длина последовательностей растёт и поэтому нам нужно строить модели, которые умеют запоминать далёкие зависимости. Это достаточно сложно. Наоборот, если мы работаем с отдельными токенами, то последовательности у нас гораздо короче. Но, с другой стороны, словарь у нас разрастается очень быстро (нам нужно уметь предсказывать каждый отдельный токен, а токенов очень много — гораздо больше, чем символов). И поэтому, с одной стороны, нам нужно помнить более короткие зависимости, но, зато, задача классификации становится сложной, потому что классов очень много. Алгоритмы, такие как byte pair encoding, позволяют найти золотую середину между этими двумя крайностями. В этом семинаре мы будем использовать реализацию byte pair encoding из библиотеки "[YouTokenToMe](#)". Эта библиотека была разработана ребятами из "ВКонтакте"<sup>[1]</sup> и, на сегодняшний день, является самой быстрой реализацией byte pair encoding. Давайте посмотрим, как работать с этой библиотекой. Основной модуль библиотеки реализован не на python, и поэтому наиболее удобный и быстрый способ скармливания данных в эту библиотеку — это через текстовый файл. Поэтому мы, сначала, сохраним все наши тексты в этот файлик, а затем вызываем функцию обучения. И функция обучения читает данные из текстового файла и складывает обученную модель (то есть, словарь замен) в другой файлик, который мы указали. Самый важный параметр здесь — это размер словаря. Он и позволяет

нам выбрать, что мы хотим — более длинные последовательности, но меньший словарь, или более короткие последовательности, но более крупный словарь, и, чем больше наш словарь, тем больше будет редких классов — это будет создавать нам некоторые сложности при обучении. Когда модель обучена, мы создаём экземпляр класса "bpe" и передаём туда путь к файлу с обученной моделью. Давайте посмотрим на словарь, который наша модель выучила. Мы сказали алгоритму: "Пожалуйста, выдели нам тысячу наиболее характерных N-грамм через byte pair encoding". Вот какие N-граммы нашлись. Во-первых, словарь содержит несколько служебных токенов — это токен "padding", то есть токен, предназначенный для выравнивания длин последовательностей, чтобы их подавать в нейросеть; это токен "unknown" — это когда алгоритм встретил в тексте какую-то N-грамму, которую не видел при обучении; и два токена "beginning of sequence" и "end of sequence". Далее идёт набор юниграмм, то есть, по сути — это все уникальные символы, которые встретились в обучающей выборке. А вот дальше уже идут более сложные конструкции, причём здесь идут вперемешку как биграммы и так и более длинные последовательности. Мы можем видеть здесь как фрагменты слов (какие-то устойчивые под слова, то есть основы слова) можем видеть имена людей без окончания (как, например, "Андрей", "Ростов", и так далее). Также здесь есть и явно слишком специфические последовательности (например, "Пьер"). Скорее всего, это сигнал к тому, что можно сделать словарь поменьше для нашей модели. Но это не так очевидно, нужно смотреть на метрики на отложенной выборке, чтобы выбрать правильный размер словаря. Токенизатор из библиотеки "YouTokenToMe" принимает на вход не отдельный текст, а сразу список текстов (список строк) и на выходе возвращает список списков, каждый вложенный список содержит числа — это номера токенов (номера N-грамм) в словаре. В принципе, как обычно. Давайте посмотрим — а какой длины последовательности после токенизации у нас получились.

Напомню что, когда мы загружали датасет, мы нарезали исходный текст на кусочки длиной 200 символов. В результате токенизации, большая часть фрагментов получила длину от 60 до 140, примерно. Причём наиболее распространённая длина последовательности — мода — около 80, то есть получилось сжать среднюю длину текста чуть более, чем в два раза. Таким образом, с точки зрения длины последовательности, задача уже проще, чем моделирование языка на уровне отдельных символов. Если мы построим гистограмму частот встречаемости токенов, то мы найдём старое [доброе распределение Ципфа](#): у нас очень мало частотных N-грамм (то есть тех N-грамм, которые встретились больше 2000 раз — их, наверное, меньше 20 суммарно), и основное количество N-грамм встретилось порядка нескольких сотен раз. Надо сказать, что очень редких N-грамм (то есть, вот этот — самый левый столбик) — их небольшое количество (около 100, всего лишь), то есть большая часть словаря у нас не является редкими классами. Это хорошая новость — всегда проще решать задачу классификации, когда классы сбалансированы. Когда мы обучали наш токенизатор, мы использовали только обучающую подвыборку всех данных, то есть только 70% текстов. Логично, что в остальных 30% могут встретиться токены, которые не встречались в обучающей выборке. Но мы используем здесь BPE, и поэтому в тестовой выборке, на самом деле, не оказалось токенов, которые бы мы не увидели в том или ином виде в обучающей выборке. То есть — да, может быть, какие-то

длинные N-граммы мы там не нашли, но зато мы смогли эти длинные N-граммы разбить на более мелкие, и, всё равно, все символы у нас так или иначе нашлись в словаре. Таким образом, когда у нас в новом тексте встречаются только неизвестные слова, то BPE просто деградирует до character-level, то есть наша модель просто становится моделью на уровне отдельных символов. Это, конечно, посложнее, но она не перестаёт работать.

[1] <https://github.com/VKCOM/YouTokenToMe>



The image shows a Jupyter Notebook interface with code snippets and text output. The top right corner features the 'SAMSUNG Research Russia' logo. The notebook contains the following code and text:

```
from dlnputils.pipeline import train_eval_loop, init_random_seed
from dlnputils.base import get_params_number

init_random_seed()

Загрузка текстов и разбиение на обучающую и тестовую подвыборки

In [2]: all_chunks = load_war_and_piece_chunks('./datasets/war_and_peace.txt')
len(all_chunks)
Out[2]: 7976

In [3]: print(all_chunks[10])
у нее был грипп, как она говорила (грипп был тогда новое
слово, употреблявшееся только редкими). В записках, разосланных утром с
красным лакеем, было написано без различия во всех:
"Si vous n'avez

In [4]: np.random.shuffle(all_chunks)

TRAIN_SPLIT = int(len(all_chunks) * 0.7)
train_texts = all_chunks[:TRAIN_SPLIT]
test_texts = all_chunks[TRAIN_SPLIT:]

print('Размер обучающей выборки', len(train_texts))
print('Размер валидационной выборки', len(test_texts))

Размер обучающей выборки 5583
Размер валидационной выборки 2393
```



The image shows the same Jupyter Notebook interface as the previous frame, but with more code visible. The top right corner features the 'SAMSUNG Research Russia' logo. The notebook contains the following code:

```
1 import heapq
2 import random
3
4 import numpy as np
5 import torch
6 import torch.nn.functional as F
7 from torch.utils.data import Dataset
8
9 from .nets import ensure_length
0
1
2 def load_war_and_piece_chunks(fname, chunk_size=200):
3     with open(fname, 'r') as fin:
4         full_text = fin.read()
5         return [full_text[start:start + chunk_size] for start
6             in range(0, len(full_text), chunk_size // 2)]
7
8
9 def save_texts_to_file(texts, out_file):
0     with open(out_file, 'w') as outf:
1         outf.write('\n'.join(texts))
2
3
4 class LanguageModelDataset(Dataset):
5     def __init__(self, token_ids, chunk_length=100, pad_value=0):
6         self.token_ids = token_ids
7         self.chunk_length = chunk_length
8         self.pad_value = pad_value
9
0     def __len__(self):
1         return len(self.token_ids)
2
3     def __getitem__(self, item):
```

## Токенизация корпуса с помощью BPE

BPE - Byte Pair Encoding

YouTokenToMe - быстрая реализация BPE

авсаве хсye  $a \rightarrow X$   
уxe  $y \rightarrow y$

```
In [5]: BPE_MODEL_FILENAME = './models/war_and_peace_bpe.yttm'

In [6]: TRAIN_TEXTS_FILENAME = './datasets/war_and_peace_bpe_train.txt'
        save_texts_to_file(train_texts, TRAIN_TEXTS_FILENAME)
        yttm.BPE.train(data=TRAIN_TEXTS_FILENAME, vocab_size=1000, model=BPE_MODEL_FILENAME)

In [7]: tokenizer = yttm.BPE(BPE_MODEL_FILENAME)

In [8]: print(' '.join(tokenizer.vocab()))
        ...

In [9]: print(tokenizer.encode(train_texts[:1]))
        ...

In [10]: train_token_ids = tokenizer.encode(train_texts, bos=True, eos=True)
        test_token_ids = tokenizer.encode(test_texts, bos=True, eos=True)

In [11]: plt.hist([len(sent) for sent in train_token_ids], bins=30)
        plt.title('Распределение длин фрагментов в токенах')
        plt.yscale('log')

In [12]: token_counts = np.bincount([token_id for text in train_token_ids for token_id in text])
        plt.hist(token_counts, bins=100)
```



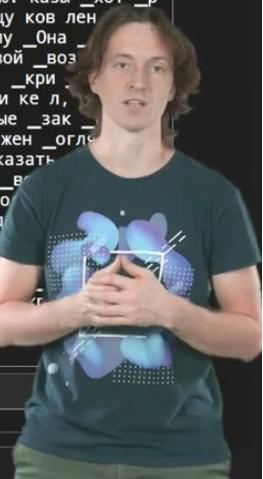
```
save_texts_to_file(train_texts, TRAIN_TEXTS_FILENAME)
yttm.BPE.train(data=TRAIN_TEXTS_FILENAME, vocab_size=1000, model=BPE_MODEL_FILENAME)

In [7]: tokenizer = yttm.BPE(BPE_MODEL_FILENAME)

In [8]: print(' '.join(tokenizer.vocab()))
<PAD> <UNK> <BOS> _ о а е и н т с л р в к , д м у п я г ъ з б . ч э - ж ъ х ю ъ ц а п г ъ и ѿ т щ ъ є ѕ ъ н
ф А в м д с ѡ ? ! К Д Б р М в Р " ) С ( ' ; И Т ё 1 h : Я 2 q f Г ъ ѿ г Ч э О З Е 5 ] [ ј 4 I z L 3 6 8 M A 9 7 У ъ ѿ
V L x X F ѿ ѿ С X J V R D E N S K C R Q O T ` w ѿ H U F G K ѿ W й * & # ѿ Z / _ с _ п _ в _ н т o _ и _ к г о а l r a c t
но _ _ по ен _ д ер ел _ б ро _ не ко во ка _ ч _ м ри _ на ло ть на ли ла _ з _ е _ у _ т ре ва ни ся сь ак _ что ру ет
ко _ б y ми _ ня да _ т o д i хo _ за _ го ем _ г _ он ол ени _ от ки ви ну каз ё, _ э та _ П тиши _ при _ вы _ра му _ Н _ х _ о
в _ вс ле _ А до _ В _ про _мо ля _ как м o _ во казал ры _его ма _об стo _это ль й, _ сказaл ере не _а _до _О _я _к _ко
тo _сво _кня _д м, у, шe _б али по чи _но сти _ни си ча ста ель _из ве лу ала дe _Ан вори _M _под _ка _д
... за ска жи ще ез я, лся _со че лы _зь сть ско ои _ли _хo _ви _ст ень _ру ря енно _Л _так ме ты _р _Р
лю сно ге же _раз _ ( со те ду ку _се ски _с _С _ва вши _все тель бе еп вал _1 _дru _было _И оп _говори
у _сто ели щи ать лько _бу ву _Пьер _пере _Ро дре _ему _Т дно _пре _Андре _а вер ю, би ство ез _са _ф ван
_Он _зна ар сп _она _которы а, _сп _т _был _же _всё _гла чал сь, _бо ды _ми _те ву ѹ. _ду ег жени _голо x,
(сно _князъ _(_сноска шел фи ба _лю _ста мот бы _лиц _бл _вз _на е. _ха кой _Бо ть, _ело _та енны _де _Я в, _е
нц ми, _с _то елов ai _си ало _да _гра _только т, _ела гля _свое _ш _1 _оп _м. тесь ходи _буд _и са _ха _ко
_у гу _"лов _ты шь оис ера _улы тельно _Кня _ее _еще _рас _п _рем _ро вно _ме _Г к, _пи ный нно _ре _-
м _qu _свои я. _дь елове _и, _жал фиц ясь _ым ерь _челове _глаза оло _оста _дени сов дя ап _По _себ _с
_боль су _для _ц _чу ка, _сказала ятъ ался аль е, _ет, _вля _Не рел _ло на, _нул _Ч _ма лась бу _Э _т
оси _него ск нцу _после гра _были _Ростов мы _эти _мне _сол _Андрей _офиц _врем ща _ех _княз _х _бо
и, _гi _он, _нимя ств _Е _мог ранцу _ком тे, _сту вать _дол _н, _л e ят лись _бе _граф _Князъ чень _вер -
ы _сло _лицо _исп _[ _Долохо ва, _чно ему _ульб _сдел _му _ча вор _пред _одно te _f _зи перь _моло
руг _комна _смот нулся ца _еи жет ения _ет _вет _нов ражени _когда ои _чего _стра _чтобы _Денисов _Ни _к
а ки, _зы _ухе _кра _они _ба _хоро ались ком ного _Ва _ни вше дин ав се вст _друго _очень спо _фа
то _вас _су ал, _княж _Ми _Но зна х. _который от _ту _перед _отве том нт _пра об пе _Ky est _од
_ови _теперь _опять _сов ная говори _л _этого _де _Ната _vous _поло _стоя арь зов _гу _que _j _иши вой
ной вство ных ение _дел _жен ем, _нъ _Что _мину стви _спроси _з _дет _офицер ент ский _этот ку, _то, _к
ш _ар _вл _ожи _ку зо ния _Во дол ми. _ла _ch _le _Нико _поч ную _двер _подо _обе _коман сили _или к
_неп жели _вой у. _чем сте му, _ние ют мал _ла _нес _разго _Мо вали is _Вы енер _Миха _взгля _которые
дар _меня он стро _М ѿ кры ее _продол _мал сы _пла вший кон _ничего друг ва. _Ба _Михай _сле ром же
```



... за скла ми ще са, я, лиси чес лы зв сив скло иши \_лю\_ви\_чи си\_ру\_ри сини \_ни\_ти\_ти\_ти дес  
лю сно ге же раз \_с (со теду ку \_се ски \_с \_С\_ва вши \_все тель бе ен вал \_л\_дру \_было \_И\_о говори \_пос гда т  
у \_сто ели ѿи ѿи лико \_бу ви \_Пьер \_пере \_Ро дно \_ему \_Т дно \_пре \_Андре \_а вер ю, бе ство ез \_са \_ф вая \_Росто  
\_Он \_зна ар сп \_она \_которы а, \_сп \_м \_был \_же \_всё \_гла чал съ, бе ды \_ми \_те ву й. \_ду ег жени \_голо х, \_ве \_  
(сно \_князь \_сноска шел фи ба \_лю\_ста мот бы \_лиц \_бол \_вз \_На е. ха \_кой \_бо в, ело \_та енны \_де \_я\_в, ет го,  
иц ми, \_с \_то елов аи \_си ало \_да \_гра \_только т, ела гля \_свое \_ш \_1\_оп м. тсяха ходи \_буд ци са ха \_которо жно  
\_в гу \_-лов \_ты ши ѿи ера \_улы тельно \_Кня\_ее \_еще \_рас \_п рем \_ро вно \_ме \_г\_к, пи ный нно \_ре \_пол \_обра \_и  
\_м \_чу \_свои я. дь елове \_и, жал фиц ясь ным ерь \_челове \_глаза оло \_оста \_Дени сов дя ап \_По \_себ \_слу па \_ле \_жи  
\_боль су \_для \_ц \_чу ка, \_сказала ять аль е, е, вля \_Не рел \_ло на, нул \_Ч\_ма лась бу \_Э \_т ман ей \_была р  
оси \_него ск нцу \_после гра \_были \_Ростов мы \_эти \_мне \_сол \_Андрей \_офиц \_врем ща ех \_княз \_х \_бо \_пер \_говорил  
и, \_г\_о \_он, нима ств \_Е \_мог ранцу \_ком те, стувать \_дол н, ле ят лись \_бе \_граф \_Княз чень \_вер \_Доло ные шо \_м  
ы \_сло \_лицо \_исп \_Г \_Долохо ва, чно ему \_улыб \_сдел \_му \_ча вор \_пред \_одно te \_f\_зи перь \_моло \_2 \_солда \_мен  
руг \_комна \_смот нулся ца еи жет ения \_et \_вет нов ражени \_когда ои \_чего \_стра \_чтобы \_денисов \_Ни ю. казы \_хот \_р  
аки, зи \_уже \_кра \_они \_ба \_хоро алисс ком ного \_Ba \_пи вше дин ав се вст \_друго \_очень спо \_францу ков лен  
то \_vas \_су ал, \_княж \_Ми \_Но зна х. \_который от \_ту \_перед \_отве том нт \_пра об пе \_Ky est \_од \_пу \_Она \_  
ови \_теперь \_опять \_сов ная говори \_L \_этого \_де \_Ната \_vous \_поло \_стоя арь зов \_гу \_que \_j ющи вой \_воз  
ной вство ных ение \_дел \_жен ем, нв \_Что \_мину стви \_спроси \_З дет \_офицер ент ский \_этот ку, \_то, \_кри  
ш \_ар \_вп \_ожи \_ку зо ния \_Во дол ми. \_la \_ch \_le \_Нико \_поч ную \_двер \_лодо \_обе \_коман сили \_или \_ке л,  
\_неп жели \_вой у. \_чем сте му, ние ют мал \_ла \_нес \_разго \_Мо вали is \_Вы енер \_Миха \_взгля \_которые \_зак  
дар \_меня он стро \_M \_ую кре еи \_продол \_мал си \_пла вший кон \_ничего друг ва. \_Ба \_Михай \_сле ром жен \_огло  
и \_дело \_Бори \_ша стоя ров \_быть но, ся, \_зам воль \_Со our \_себе пра ского \_Никола \_Куту кая \_без казат  
ими вала in \_друг на, \_обрати \_Как \_что-то \_Ну, \_чи кий лаго \_князя вл \_их \_Пав \_сер \_своего \_вес \_в  
соб рая \_бра \_че \_всех ниц \_нас \_жиз ais \_свою \_ехал \_Ко \_каз \_само кра \_он. д, про \_тем \_молодо ско  
еge дер \_пе \_команди виши \_останови \_лоша сты \_взя \_генер ра, стно да, \_доро из \_жд может \_да \_сид  
о ѿще \_тре ть. чь \_ну \_чувство \_госу \_Пьера \_мол \_малень \_Наташа кое им \_вдруг it \_I' \_ли, мер \_Баг  
\_непри \_други \_спросил \_будет \_пото и. \_своей \_книга кого ез \_Баграти \_доб \_на \_люб хала \_Ж\_пу \_  
а лось \_Павлов \_Васили \_ок \_видел \_У оль дом \_впере \_начал \_особ \_где кто тер \_вст шь, \_сча \_бле \_  
ме \_про ливо \_вам ерез 'ест вя \_мой \_Ту \_рус \_день \_ши \_ди \_ственno \_g \_дро \_улыба \_дума ку. \_лу \_  
е виц каза стя \_ша \_отвечал \_всегда цо \_сом ch \_Пьер, \_Г\_ско гес шла жу \_более \_выражени \_есть ны,  
я \_сам \_того, \_вели \_спо елы та, \_сла \_ы,



```
In [9]: print(tokenizer.encode(train_texts[:1]))
```

у \_слу\_ лицо \_исп\_ [ \_Долохо ва, чю ему \_ульб\_ \_сдел\_ м\_ча вор \_пред \_одно te \_f зи перь \_моко \_2 \_солда \_мен  
руг \_комна \_смот\_ нулся ца еш жет ения \_et вет нов ражени \_когда о\_чего \_стра \_чтобы \_Денисов \_Ни ю. казы \_хот \_р  
а ки, эы \_уже \_кра \_они \_ба \_хоро ались комого \_Ba \_пи вине дин ав се вст \_друго \_очень спо \_францу ков лен \_буд  
то \_вас \_су ал, \_княж \_Ми \_Но зна х. \_который от \_ту \_перед \_отве том нт \_проба об пе \_Ky est \_од \_пу \_Она \_Анна на  
ови \_теперь \_опять \_сов ная говори \_л \_этого \_де \_Ната vous \_поло \_стоя арь зов \_гу \_que \_j \_ющи вой \_воз \_себя  
ной вство ных ение \_дел \_жен ем, нь \_Что \_мину стви \_спроси \_з дет \_офицер ент ский \_этот ку, \_то, \_кри \_Марь лы  
ш \_ар \_вп \_ожи \_ку зо ния \_Во дол ми. \_la \_ch \_le \_Нико \_поч ную \_двер \_подо \_обе \_коман сили \_или ке л, \_Это те  
\_неп жели \_вой у. \_чем сте мие, ние ит мал \_ла \_нес \_разго \_Мо вали is \_Вы енер \_Миха \_взгля \_которые \_зак \_сы il  
дар \_меня он стро \_М ую кры ее \_продол \_мал сы \_пла вший кон \_ничего друг ва. \_Ба \_Михай \_сле ром жен \_огля \_вид  
и \_дело \_Бори \_ша стоя ров \_быть но, ся, \_зам воль \_Со оиг \_себе пра ского \_Никола \_Куту кая \_без казать \_время н  
ыми вала in \_друг на. \_обрати \_Как \_что-то \_Ну, \_чики \_киага \_казнял вл \_их \_Пав \_сер \_своего \_вес \_вот \_Михайлова  
соб рая \_бра \_че \_всех ниц \_нас \_хиз ais \_свом ехал \_Ко \_каз \_само кра \_он. д, про \_тем \_молодо сколько \_и \_b \_A  
еге дер \_пе \_команди виши \_останови \_лоша сты \_взя \_генера, стно да, \_доро из жд \_может \_да \_сидел \_сторо  
о юще \_тре ть. чь \_ну \_чувство \_госу \_Пьера \_мол \_малень \_Наташа кое им \_вдруг it \_I \_ли, мер \_Багра вого .  
\_непри \_други \_спросил \_будет \_пото и. \_свойкни кого ез \_Баграти \_доб \_ла \_люб хала \_Ж пу \_расс \_ап  
а лось \_Павлов \_Васили \_ок \_видел \_у оль дом \_впере \_начал \_особ \_где кто тер \_вст шь, \_сча \_бле дал \_тог  
м\_про ливо \_вам ерез est вя \_мой \_Ту \_рус \_день \_ши \_ди ственно \_g dro \_ульба \_дума ку. \_лу \_з \_кре  
е виц каза стя шая \_отвечал \_всегда \_ко \_сом ch \_Пьер, \_ско гес шла жу \_более \_выражени \_есть ны, ской \_блага  
я \_сам \_того, \_вели \_спо елы та, \_сла y,



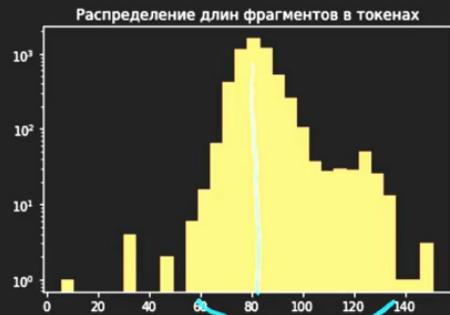
```
In [9]: print(tokenizer.encode(train_texts[:1]))
```

```
[[210, 238, 244, 13, 317, 16, 147, 200, 12, 265, 35, 161, 337, 490, 203, 269, 447, 4, 111, 111, 96, 276, 551, 201, 726, 199, 161, 848, 889, 772, 23, 16, 690, 179, 585, 18, 154, 412, 19, 382, 157, 186, 634, 4, 363, 670, 157, 793, 37, 7, 426, 791, 186, 635, 10, 518, 774, 650, 25, 988, 206, 186, 13, 201, 8, 15, 31, 23, 8, 34, 4441]]
```

```
In [10]: train_token_ids = tokenizer.encode(train_texts, bos=True, eos=True)
test_token_ids = tokenizer.encode(test_texts, bos=True, eos=True)
```

```
In [11]: plt.hist([len(sent) for sent in train_token_ids], bins=30)
plt.title('Распределение длин фрагментов в токенах')
plt.yscale('log');
```

```
In [11]: plt.hist([len(sent) for sent in train_token_ids], bins=30)
plt.title('Распределение длин фрагментов в токенах')
plt.yscale('log');
```

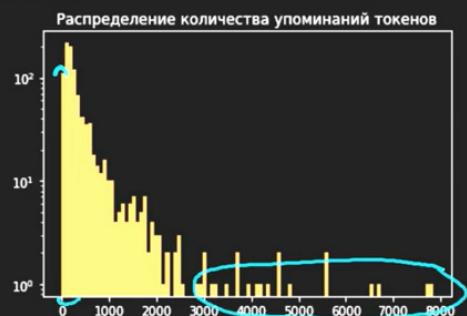


```
In [12]: token_counts = np.bincount([token_id for text in train_token_ids for token_id in text])
plt.hist(token_counts, bins=100)
plt.title('Распределение количества упоминаний токенов')
plt.yscale('log');
```

```
In [13]: unknown_subwords_in_test = sum(1 for text in test_token_ids for token_id in text if token_id == 1)
print('Количество случаев с неизвестными n-граммами символов в валидационной выборке',
      unknown_subwords_in_test)
```



```
In [12]: token_counts = np.bincount([token_id for text in train_token_ids for token_id in text])
plt.hist(token_counts, bins=100)
plt.title('Распределение количества упоминаний токенов')
plt.yscale('log');
```



```
In [13]: unknown_subwords_in_test = sum(1 for text in test_token_ids for token_id in text if token_id == 1)
print('Количество случаев с неизвестными n-граммами символов в валидационной выборке',
      unknown_subwords_in_test)
```



## Подготовка датасетов для PyTorch

```
In [13]: unknown_subwords_in_test = sum(1 for text in test_token_ids for token_id in text if token_id == 1)
print('Количество случаев с неизвестными n-граммами символов в валидационной выборке',
      unknown_subwords_in_test)
Количество случаев с неизвестными n-граммами символов в валидационной выборке 0
```

## Подготовка датасетов для PyTorch

```
In [14]: CHUNK_LENGTH = 80
train_dataset = LanguageModelDataset(train_token_ids,
                                      chunk_length=CHUNK_LENGTH)
test_dataset = LanguageModelDataset(test_token_ids,
                                    chunk_length=CHUNK_LENGTH)
```

```
In [15]: train_dataset[0]
```

...

```
In [16]: tokenizer.decode(list(train_dataset[0]))
```

...

## Общие классы и функции

### Маска зависимостей

```
In [17]: def make_target_dependency_mask(length):
    full_mask = torch.ones(length - length)
```



Из комментариев:

Вопрос:

Когда мы нарезаем текст на последовательности по 200 символов, мы неизбежно режем многие слова пополам. Не лучше ли нарезать по *примерно* 200, к примеру, обрезая на первом пробеле после 200 символов, или такая небольшая потеря информации (что половина обрезанного слова связана со второй половинкой) на практике несущественна?

Ответ (Романа Суворова):

да, нарезка по 200 символов в этом семинаре - это упрощение. На практике лучше резать по целым предложениям или абзацам.

Доп. комментарий (от студента):

не совсем понимаю тогда, в лекции сказали, что общепринято не резать на предложения, что кажется вполне разумно вместе с подходом "не обрезать последнее слово". Однако, с другой стороны, мы же потом всё равно будем "прокручивать" последовательности "потокенно", то есть, фактически, какая нам разница тогда, мы так или иначе обрежим по пол слова. Могу предположить, что первый вариант с "необрезанием" лучше в случае, когда мы получаем представления текста по кускам чтобы усреднить, а второй - когда нужно обучить языковую модель. Хотелось бы комментарий по данному вопросу.

Как обычно, давайте создадим специальную структуру данных — Dataset, который будет готовить обучающие примеры для того, чтобы подавать их в pytorch для обучения. Для этой цели мы написали специальный класс "LanguageModelDataset", который принимает на вход список токенизованных предложений, в которых токены заменены на их номера. А также он

принимает длину фрагмента которую нужно подавать в модель за раз. То есть, это наибольшее количество токенов, которые модель будет видеть за раз. Давайте посмотрим, как этот Dataset работает. Как обычно, наш Dataset реализует два основных метода — это получение длины датасета (то есть, количества примеров) и "get\_item" — это получение одного конкретного примера по его порядковому номеру. Когда мы готовим очередной пример для модели, мы выбираем по индексу фрагмент текста, затем из всего текста мы выбираем какой-то случайный подфрагмент, непрерывный. И делаем из этого фрагмента ещё два кусочка — первый кусочек (здесь это переменная "seed part") — это весь текст, кроме последнего символа. А "target\_part" — это весь текст, кроме первого символа. Таким образом, мы получаем, как бы, два текста — "seed\_part" у нас выполняет роль входа в модель, а "target\_part" — это то, что мы ожидаем на выходе модели. Таким образом, если у нас есть, например, текст "ABCD", тогда "seed-part" — это "ABC", а "target\_part" — это "BCD". Физический смысл у этих последовательностей — следующий. На очередной позиции в "target" стоит токен, который модель должна предсказать, прочитав столько же токенов из входной последовательности. Таким образом, токен "B" она должна уметь предсказать только лишь по токену "A", токен "C" модель должна уметь предсказывать из "AB", а токен "D" она должна уметь предсказывать из всей входной последовательности, то есть "ABC". На экране вы видите один пример, сгенерированный нашим Dataset. Мы видим что здесь токены, как бы, смешены на одну позицию влево всегда, и на очередной позиции вектора "target" у нас стоит токен, который нужно уметь предсказывать после прочтения всех предыдущих токенов. Токенайзер из библиотеки "[YouTokenToMe](#)" умеет и декодировать тексты. То есть, обратно преобразовывать их в текст (то есть в строку). На экране вы видите пример детокенизации, то есть мы взяли тот же самый обучающий пример и прогнали его через детокенизатор. Мы видим, что две последовательности, входная и выходная, сдвинуты относительно друг друга на один токен, на одну [N-грамму](#). Тут вы можете задать вопрос: "Как же так?" Мы подаём на вход в модель весь текст и просим, чтобы она практически этот же текст нам и вернула, сдвинув на один токен влево". Это, кажется, очень простая задача, модель же ничего не выучит... В самом деле, если мы будем просто в лоб подавать текст и на выходе просить его же — действительно, модель ничего не выучит. Нам нужно усложнить ей задачу, нам нужно сделать так, чтобы при предсказании  $i$ -го токена она принципиально не могла учитывать токены, стоящие справа от этой позиции. Для этого мы будем использовать специальную маску. На экране вы видите функцию, которая генерирует такую маску и, собственно, пример этой маски — давайте разберём, какой физический смысл у этой маски. Это так называемая "маска зависимости позиций". Для примера мы сгенерировали эту маску для достаточно короткой последовательности длины 10. Мaska имеет размер 10 на 10, это квадратная матрица, строка соответствует номеру позиции в выходной последовательности, а столбцы соответствуют номерам позиций во входной последовательности, и на пересечении столбца и строки стоит 0, если при предсказании токена на позиции " $i$ " можно учитывать токен на позиции " $j$ ". То есть, если учитывать можно — тогда стоит 0, а если учитывать нельзя — тогда стоит  $-\infty$ . Таким образом, мы "запрещаем" модели смотреть на все токены справа. И, например, для самого первого токена (это первая строчка)

мы можем использовать только самый первый входной токен. Для второго токена в выходной последовательности мы можем использовать уже два токена, а для последнего выходного токена мы можем использовать всю входную последовательность. Если в двух словах — то эти маски подаются в механизм внимания для того, чтобы занулить веса определённых элементов, чтобы мы не учитывали определённые входные позиции при расчёте выходных позиций. Она используется именно в механизмах внимания. Как именно она там используется, мы рассмотрим чуть позже.

## Подготовка датасетов для PyTorch

```
In [14]: CHUNK_LENGTH = 80
train_dataset = LanguageModelDataset(train_token_ids,
                                      chunk_length=CHUNK_LENGTH)
test_dataset = LanguageModelDataset(test_token_ids,
                                      chunk_length=CHUNK_LENGTH)
```

```
In [15]: train_dataset[0]
```

```
Out[15]: (array([ 2, 210, 238, 244, 13, 317, 16, 147, 200, 12, 265, 35, 161,
   337, 490, 203, 269, 447, 4, 111, 111, 96, 27, 415, 148, 176,
   551, 201, 726, 199, 161, 848, 889, 772, 23, 16, 690, 179, 585,
   18, 154, 412, 19, 382, 157, 186, 635, 10, 518, 774, 363, 670,
   157, 793, 37, 7, 426, 791, 186, 635, 10, 518, 774, 650, 25,
   988, 206, 186, 13, 201, 8, 149, 474, 17, 275, 31, 23, 8,
   34, 444]), array([210, 238, 244, 13, 317, 16, 147, 200, 12, 265, 35, 161, 337,
   490, 203, 269, 447, 4, 111, 111, 96, 27, 415, 148, 176, 551,
   201, 726, 199, 161, 848, 889, 772, 23, 16, 690, 179, 585, 18,
   154, 412, 19, 382, 157, 186, 635, 10, 518, 774, 363, 670, 157,
   793, 37, 7, 426, 791, 186, 635, 10, 518, 774, 650, 25, 988,
   206, 186, 13, 201, 8, 149, 474, 17, 275, 31, 23, 8, 34,
   444, 3]))
```

```
In [16]: tokenizer.decode(list(train_dataset[0]))
```

```
Out[16]: ['<BOS> от восторга, с толпою побежал за ним. XXI. На площади куда поехал государь, стояли лицом к лицу с
  воин преображенцев, слева батальон французской гвардии в медвежьих ш',
  'от восторга, с толпою побежал за ним. XXI. На площади куда поехал государь, стояли лицом к лицу справа
  еображенцев, слева батальон французской гвардии в медвежьих ш<EOS>']
```



```
2
3
4 class LanguageModelDataset(Dataset):
5     def __init__(self, token_ids, chunk_length=100, pad_value=0):
6         self.token_ids = token_ids
7         self.chunk_length = chunk_length
8         self.pad_value = pad_value
9
10    def __len__(self):
11        return len(self.token_ids)
12
13    def __getitem__(self, item):
14        text = self.token_ids[item]
15        start_i = random.randint(0, max(0, len(text) - self.chunk_length - 1))
16        chunk = text[start_i : start_i + self.chunk_length + 1]
17
18        seed_part = chunk[:-1]
19        target_part = chunk[1:]
20
21        seed_part = ensure_length(seed_part, self.chunk_length, self.pad_value)
22        target_part = ensure_length(target_part, self.chunk_length, self.pad_value)
23
24        seed_part = np.array(seed_part)
25        target_part = np.array(target_part)
26
27    return seed_part, target_part
28
29
30 class GreedyGenerator:
31     def __init__(self, model, tokenizer, device='cuda', eos_token_id=3):
32         self.model = model
33         self.tokenizer = tokenizer
34         self.device = torch.device(device)
```

$\widehat{a} b c$

$b \underline{c} d$



```
In [14]: CHUNK_LENGTH = 80
train_dataset = LanguageModelDataset(train_token_ids,
                                      chunk_length=CHUNK_LENGTH)
test_dataset = LanguageModelDataset(test_token_ids,
                                      chunk_length=CHUNK_LENGTH)

In [15]: train_dataset[0]
```

```
Out[15]: (array([ 2, 210, 238, 244, 13, 317, 16, 147, 200, 12, 265, 35, 161,
   337, 490, 203, 269, 447, 4, 111, 111, 96, 27, 415, 148, 176,
   551, 201, 726, 199, 161, 848, 889, 772, 23, 16, 690, 179, 585,
   18, 154, 412, 19, 382, 157, 186, 635, 10, 518, 774, 363, 670,
   157, 793, 37, 7, 426, 791, 186, 635, 10, 518, 774, 650, 25,
   988, 206, 186, 13, 201, 8, 149, 474, 17, 275, 31, 23, 8, 34,
   4, 444]), array([210, 238, 244, 13, 317, 16, 147, 200, 12, 265, 35, 161, 337,
   490, 203, 269, 447, 4, 111, 111, 96, 27, 415, 148, 176, 551,
   201, 726, 199, 161, 848, 889, 772, 23, 16, 690, 179, 585, 18,
   154, 412, 19, 382, 157, 186, 635, 10, 518, 774, 363, 670, 157,
   793, 37, 7, 426, 791, 186, 635, 10, 518, 774, 650, 25, 988,
   206, 186, 13, 201, 8, 149, 474, 17, 275, 31, 23, 8, 34,
   444, 3]))
```

```
In [16]: tokenizer.decode(list(train_dataset[0]))
```

```
Out[16]: ['<BOS> от восторга, с толпою побежал за ним. XXI. На площади куда поехал государь, стояли лицом к
  боян преображенцев, слева батальон французской гвардии в медвежьих ш', 'от восторга, с толпою побежал за ним. XXI. На площади куда поехал государь, стояли лицом к лицу спасенные
  изображены, слева батальон французской гвардии в медвежьих ш<EOS>']
```

## Общие классы и функции



### Маска зависимостей

```
In [17]: def make_target_dependency_mask(length):
    full_mask = torch.ones(length, length)
    ignore_mask = torch.tril(full_mask) < 1
    full_mask.masked_fill_(ignore_mask, float('-inf'))
    full_mask.masked_fill_(~ignore_mask, 0)
    return full_mask

make_target_dependency_mask(10)
```

```
Out[17]: tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
   [0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
   [0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
   [0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
   [0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf],
   [0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf],
   [0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf],
   [0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],
   [0., 0., 0., 0., 0., 0., 0., 0., 0., -inf],
   [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

i

10

### Кодирование позиции

```
In [18]: def make_positional_encoding(max_length, embedding_size):
    time = np.pi * torch.arange(0, max_length).float()
    freq_dividers = torch.arange(1, embedding_size // 2 + 1).float()
    inputs = time[:, None] / freq_dividers[None, :]

    result = torch.zeros(max_length, embedding_size)
```



Из комментариев:

Вопрос:

А как работает функция ensure\_length в реализации LanguageModelDataset?

Самоответ:

Вопрос снят, кажется разобрался. Функция просто дополняет длину тренировочного и тестового куска до нужного размера символом <PAD> в случае малой длины, или обрезает их,

если длина слишком большая.

Следующий вспомогательный элемент, который нам нужен — это позиционное кодирование. Как вы помните из лекции, механизм [self-attention](#) — он, в некотором смысле, похож на механизм свёрток, тем, что он инвариантен к позиции элемента в последовательности. Мы можем за одну операцию сравнить каждый элемент последователи с любым другим элементом последовательности. Но кажется, что, когда мы работаем с текстами, особенно с текстами с фиксированным порядком слов, нам важно учитывать позиции токенов. Даже если порядок слов и не фиксированный, то относительные позиции токенов уж точно полезны. Потому что всё-таки это достаточно редкий случай — когда связь между словами идёт через пол-текста. Даже для человека такие связи были бы очень сложными. Короче говоря, нам нужно учитывать относительные позиции токенов. Для того, чтобы закодировать позиции токенов, к [эмбеддингу](#) токена, который мы берём из таблички, будем прибавлять эмбеддинг позиции. Что такое "эмбеддинг позиций"? Это вектор такой же длины, что и эмбеддинг токена, который имеет разное значение для разных позиций. И самый, наверное, интуитивный способ закодировать позицию — это использовать какой-то периодический сигнал. Авторы [трансформера](#) предлагают использовать набор синусоид и косинусоид разной частоты. На экране вы видите график, который изображает сразу несколько таких векторов. Один срез графика (вертикальный) описывает нам эмбеддинг одной позиции. Вы можете видеть, что здесь есть как высокочастотные сигналы (как, например, вот этот), так и низкочастотные (как вот эта горизонтальная прямая, или вот этот голубой график). Таким образом, по изменению сигнала на определённых позициях мы можем определить, как далеко друг от друга два токена находятся. Давайте уже начнём строить какие-нибудь модельки. Для удобства, давайте определим общий класс — "языковая модель". Этот класс будет выполнять некоторые базовые операции вне зависимости от архитектуры нейросети, которая языковую модель будет реализовывать. К таким операциям относится хранение векторов слов, получение кодов позиций, а также предсказание токенов для каждой позиции. Давайте посмотрим на метод "forward" и разберём основные шаги. На вход к нам приходит прямоугольная матрица, в ней количество строк соответствует количеству примеров в батче, а количество столбцов — наибольшей длине последовательности. Зная длину последовательности мы можем сгенерировать маску зависимостей, то есть вот эту вот треугольную матрицу из нулей и минус бесконечности. А также мы можем сгенерировать ещё одну маску, которая нам помечает токены за границей последовательности. То есть, если последовательность короче, чем "max\_in\_len" (то есть — чем наибольшая длина входной последовательности), то она будет "добиваться" нулями до конца, до наибольшей длины, и нам нужно исключить эти нули из рассмотрения, из предсказаний. Для этого мы используем "padding mask". Хорошо, маски построены, теперь нам нужно получить начальное представление токенов для того, чтобы их подать уже в нейросеть. Вектора токенов у нас будут складываться из двух компонент, а именно: эмбеддинг самого токена и эмбеддинг позиции. Эмбеддинги токенов мы берём просто

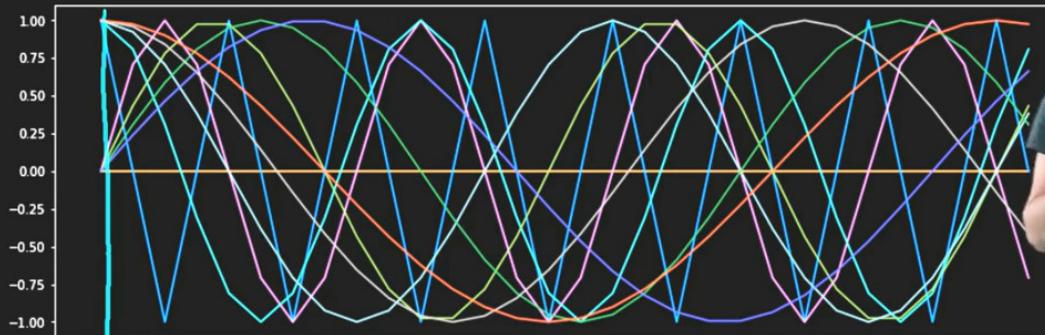
из таблички, для этого мы используем стандартный модуль из pytorch — nn.embedding. Как обычно, мы помечаем, что нулевой токен — это фиктивный токен, то есть "padding". Напомню, что этот модуль, по сути, осуществляет всего лишь выборку строк из матрицы по индексам. И он позволяет обучать эмбеддинги, что называется, "end to end". То есть эмбеддинги будут получать обновления на каждом градиентном шаге. На выходе из эмбеддинг слоя мы уже имеем не двухмерную матрицу, прямоугольную, а трёхмерный [тензор](#). У нас добавилось ещё одно измерение, соответствующее количеству элементов в эмбеддинге. Для того, чтобы получить эмбеддинги позиций, мы используем функцию, которую рассмотрели чуть ранее. Она возвращает нам прямоугольную матрицу размерности ["длина последовательности" на "размер эмбеддинга"]. То есть, в ней нет измерения, соответствующего количеству примеров в батче. Чтобы иметь возможность сложить два тензора эмбеддингов — то есть, эмбеддинги токенов и эмбеддинги позиции, мы добавляем некоторое фиктивное измерение (добавляем единичку). После этой операции тензоры "seed\_embs" и "pos\_codes" будут оба трёхмерными, их можно будет сложить — что мы, собственно, и делаем. Далее к полученным эмбеддингам мы применяем dropout. На самом деле, dropout здесь очень драматично влияет на возможности модели [переобучаться](#). Я предлагаю вам поиграть с силой этого dropout и посмотреть, что будет, если, например, его вообще выключить. А, с другой стороны — насколько сильным это dropout вообще можно сделать, при том, что модель по-прежнему вот как-то учится. Я предлагаю вам ответить на вопрос — какой dropout важнее: этот, или те dropout, которые находятся в основной нейросети (которая собственно, предсказывает токены). Далее мы подаём признаки токенов в некоторую нейросеть, которая здесь у нас лежит в переменной "backbone". Какая это нейросеть — здесь пока здесь не определено, это определяется при создании экземпляра класса "LanguageModel". Кроме эмбеддингов мы передаём туда две маски — маску зависимости и маску padding. Нейросеть "backbone" возвращает нам, также, трёхмерный тензор такой же размерности, что и была на входе, то есть ["количество элементов в батче", "максимальная длина последовательности" и "размер вектора представления"], (то есть рабочая размерность). Размерность модели, то есть величина последнего измерения тензора, не соответствует размеру словаря — это вполне нормально, мы не хотим растянуть ширину модели линейно с ростом количества токенов в словаре (это слишком дорого). Поэтому нам нужен дополнительный слой, который преобразует какой-то вектор в распределение вероятностей токенов в словаре. Для этого мы используем простой линейный слой. Когда вы подаёте на вход линейному слою какой-то многомерный тензор, то линейная проекция применяется к последнему измерению. Выходной тензор у нас представляет [логиты](#) (то есть он представляет не сами вероятности, распределения вероятностей, а логиты). Чтобы получить распределение вероятностей из логитов, нам нужно применить к этому тензору softmax по последнему измерению. Но мы не будем это делать, потому что мы знаем, что после логитов сразу пойдёт [кросс-энтропия](#), а если у нас [софтмакс](#) и кросс энтропия, то можно софтмакс не брать полностью, можно лишние экспоненты и логарифмы сократить, и получить большую вычислительную стабильность. Короче говоря, мы не будем считать софтмакс на выходе модели. Также определим парочку стандартных вспомогательных компонент. Во-

первых, нам нужна функция потерь. В качестве функции потерь мы будем использовать кросс-энтропию, но, перед тем, как подавать данные в функцию расчёта кросс-энтропии, мы просто их вытянем в линию. То есть мы, как бы, забудем, что у нас были отдельно — примеры в батче, и отдельно — токены в предложениях. Мы смешаем все предложения в кучу. Для оценки кросс-энтропии это совершенно не важно. А ещё мы скажем что нужно игнорировать padding. Это сделает фактическое распределение классов сильно менее скошенным и улучшит сходимости. Хотя вы можете выключить это и посмотреть, как это повлияет. А также, другая стандартная утилитка — это расписание изменения длины градиентного шага. Мы говорим, тем самым, что, если в течение двадцати эпох значение функции потерь на валидации не уменьшилось существенно, тогда — уменьшить длину градиентного шага в два раза. Использование такого "расписания" позволяет исключить ошибки, когда вы устанавливаете слишком большой "learning rate" при обучении, то есть если вы поставите слишком большой "learning rate", то модель просто не будет учиться и, в результате, "learning rate" автоматически уменьшится. Да, он уменьшится не сразу (а спустя вот такое вот количество эпох), но, тем не менее, если вы запустили эксперимент на ночь и ушли, то, скорее всего, утром вы получите обученную модель. Давайте предпримем первую попытку обучить языковую модель, используя реализацию трансформера из библиотеки pytorch. Трансформер в pytorch появился, начиная с версии 1.2, то есть, если вы хотите запускать этот семинар, вам нужно обновить библиотеку pytorch до этой версии. Мы будем использовать не весь трансформер, а только его первую часть — трансформер "encoder". Этот вспомогательный класс нам нужен для двух задач. По какой-то причине, стандартная реализация трансформера умеет работать с тензорами, в которых первое измерение соответствует не размеру батча, а длине последовательности, а "batch\_size" стоит на втором месте. Лично мне это кажется не вполне удобным, хотя... это вопрос предпочтений в большей степени. То есть — вопрос удобства: если вам нужно меньше транспонировать тензоры туда-сюда, и, при этом, иметь "batch\_size" на втором месте, тогда вам не нужен этот класс — вы можете сэкономить лишнее транспонирование, reshape, и так далее. Таким образом, этот класс делает, по сути, всего лишь две вещи. Во-первых, он транспонирует тензор перед подачей в трансформер, транспонирует результаты работы трансформера обратно, и возвращает результат не изменённым. И — вторая важная деталь — это инициализация параметров. По умолчанию, в трансформере используется равномерный шум с амплитудой, подбираемой исходя из количества входных признаков. Эта схема инициализации реализовывается в pytorch функцией "[xavier\\_uniform](#)". Таким способом, мы инициализируем все веса, кроме bias, то есть все "матричные" веса. Давайте уже соберём нашего монстра и чему-нибудь обучим.

## Кодирование позиции

```
In [18]: def make_positional_encoding(max_length, embedding_size):
    time = np.pi * torch.arange(0, max_length).float()
    freq_dividers = torch.arange(1, embedding_size // 2 + 1).float()
    inputs = time[:, None] / freq_dividers[None, :]
    result = torch.zeros(max_length, embedding_size)
    result[:, 0::2] = torch.sin(inputs)
    result[:, 1::2] = torch.cos(inputs)
    return result
```

```
In [19]: sample_pos_codes = make_positional_encoding(30, 30)
plt.plot(sample_pos_codes[:, ::3]);
plt.gcf().set_size_inches((15, 5))
```



## Основной класс - языковая модель

```
In [20]: class LanguageModel(nn.Module):
    def __init__(self, vocab_size, embedding_size, backbone, emb_dropout=0.0):
        super().__init__()
        self.embedding_size = embedding_size
        self.embeddings = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
        self.emb_dropout = nn.Dropout(emb_dropout)
        self.backbone = backbone
        self.out = nn.Linear(embedding_size, vocab_size)

    def forward(self, seed_token_ids):
        """
        seed_token_ids - BatchSize x MaxInLen
        """
        batch_size, max_in_length = seed_token_ids.shape

        seed_padding_mask = seed_token_ids == 0
        dependency_mask = make_target_dependency_mask(max_in_length) \
            .to(seed_token_ids.device)

        seed_embs = self.embeddings(seed_token_ids) # BatchSize x MaxInLen x EmbSize
        pos_codes = make_positional_encoding(max_in_length,
                                             self.embedding_size).unsqueeze(0).to(seed_embs.device)
        seed_embs = seed_embs + pos_codes
        seed_embs = self.emb_dropout(seed_embs)

        # BatchSize x TargetLen x EmbSize
        target_features = seed_embs
        target_features = self.backbone(seed_embs,
                                       mask=dependency_mask,
                                       src_key_padding_mask=seed_padding_mask)
        logits = self.out(target_features) # BatchSize x TargetLen x VocabSize
```



### Утилиты для обучения - функция потерь и расписание изменения длины градиентного шага

```
In [21]: def lm_cross_entropy(pred, target):
    """
    pred - BatchSize x TargetLen x VocabSize
    target - BatchSize x TargetLen
    """
    pred_flat = pred.view(-1, pred.shape[-1]) # BatchSize*TargetLen x VocabSize
    target_flat = target.view(-1) # BatchSize*TargetLen
    return F.cross_entropy(pred_flat, target_flat, ignore_index=0)

def lr_scheduler(optimizer):
    return torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                       patience=20,
                                                       factor=0.5,
                                                       verbose=True)
```



### Реализация Transformer из PyTorch 1.2

```
In [22]: class BatchFirstTransformerEncoder(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
        self.impl = nn.TransformerEncoder(*args, **kwargs)
        self.initialize_weights()

    def forward(self, src, *args, **kwargs):
        src = src.transpose(0, 1).contiguous() # MaxInLen x BatchSize x EmbSize
        result = self.impl(src, *args, **kwargs) # TargetLen x BatchSize x EmbSize
        result = result.transpose(0, 1).contiguous() # BatchSize x TargetLen x EmbSize
```

### Реализация Transformer из PyTorch 1.2

```
In [22]: class BatchFirstTransformerEncoder(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
        self.impl = nn.TransformerEncoder(*args, **kwargs)
        self.initialize_weights()

    def forward(self, src, *args, **kwargs):
        ✓ src = src.transpose(0, 1).contiguous() # MaxInLen x BatchSize x EmbSize
        ✓ result = self.impl(src, *args, **kwargs) # TargetLen x BatchSize x EmbSize
        ✓ result = result.transpose(0, 1).contiguous() # BatchSize x TargetLen x EmbSize
        return result

    def initialize_weights(self):
        for param in self.impl.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)
```



```
In [23]: torch_transf_model = LanguageModel(tokenizer.vocab_size(),
                                         256,
                                         BatchFirstTransformerEncoder(
                                             nn.TransformerEncoderLayer(
                                                 d_model=256,
                                                 nhead=16,
                                                 dim_feedforward=512,
                                                 dropout=0.1),
                                             num_layers=3),
                                         emb_dropout=0.1)
print('Количество параметров', get_params_number(torch_transf_model))
...
```

Наша модель реализуется базовым классом "LanguageModel". Мы передаём размер словаря (это 1000, в данном случае), размер эмбеддинга (256 элементов) и он же — это размер модели. А также передаём туда экземпляр backbone нейросети, то есть — это та нейросеть, которая будет производить агрегацию контекстов, будет сравнивать слова с соседними словами и,

таким образом, на вход она будет получать эмбеддинги отдельных токенов, а на выходе у неё уже будут эмбеддинги фраз, предложений... То есть — более крупных конструкций. Мы будем использовать три слоя [self-attention](#) и будем использовать не очень большой dropout. Я предлагаю вам поиграться с этими параметрами, чтобы понять, насколько от них зависит качество. Как мы видим, модель содержит примерно 2 миллиона параметров... Это не очень большая модель (по современным меркам). Давайте попробуем обучить модель. Для этого мы будем использовать нашу стандартную функцию, которую мы использовали весь курс до этого. На что здесь нужно обратить внимание — это то, что мы передаём сюда функцию потерь, "Im cross entropy", то есть — это та функция, которую мы определили, это не просто функция [кросс-энтропии](#) из pytorch. А также здесь стоит большой batch\_size. Если вы будете запускать этот ноутбук на видеокарте с небольшим количеством памяти (меньше 11 гигабайт), то вам нужно будет этот "batch\_size" изменить — так, чтобы ваши данные влезали в видеокарту. Я поставил его таким большим просто потому, что в моём случае так быстрее и всё хорошо учится. А ещё я поставил большое количество эпох здесь. То есть, я ожидаю, что модель будет сходиться не очень быстро. В самом деле, на каждой эпохе мы делаем всего лишь 11 итераций, потому что у нас небольшой датасет (как мы помним, в нём примерно 6000 обучающих фрагментов и 3000 тестовых), и при таком большом размере батча мы делаем маленькое количество обновлений, поэтому нам нужно побольше эпох. Зато каждая эпоха проходит очень быстро. В моём случае, потребовалось примерно 800 эпох, то есть чуть больше чем 8000 градиентных шагов, чтобы модель более-менее сошлась. Сходимость модели мы проверяем по среднему значению функции потерь на отложенной выборке. То есть, спустя примерно 800 тысяч шагов, функция потерь на валидации перестала улучшаться, поэтому мы досрочно остановили обучение.

Хорошая практика — сохранять модели на промежуточных итерациях для того чтобы если, например, вам придётся досрочно остановить обучение или вы захотите усреднить модели (веса моделей с разных итераций) между собой (это, кстати, хорошая практика, она позволяет немного улучшить качество). В данном случае мы сохраняем только последнюю лучшую модель и тут же её загружаем, чтобы поиграться немножко. Как вы помните из лекции, существует два основных подхода к декодированию текста из языковой модели. Это полностью жадный алгоритм декодирования — то есть, когда мы на каждом шаге берём наиболее вероятный токен. И, при этом, наиболее вероятный токен мы выбираем без учёта совместного распределения токенов (то есть — вот на этом шаге модель сказала, что токен "A" самый лучший — мы его и берём вне зависимости от того, какой следующий токен может быть, или предыдущий). Этот алгоритм мы реализовали в нашей библиотеке в классе "GreedyGenerator". Он получает на вход обученную модель и "tokenizer". А затем он будет получать на вход текст и выдавать новый текст. Давайте посмотрим, как это всё работает. Собственно, алгоритм предельно простой — сначала мы токенизуем наше предложение. Затем мы делаем некоторое количество шагов, но не более заданного числа шагов. И, на каждом шаге, мы полностью перевычисляем модель, то есть мы берём все входные токены, которые накопили к данному шагу, заворачиваем их в [тензор](#), копируем на видеокарту и прогоняем через модель, а потом выбираем наиболее вероятный последний токен. Здесь "batch\_size" равен 1, то есть мы

берём в этом "indexer" первый элемент батча (он соответствует нашему предложению) и последний элемент в последовательности. Таким образом, после взятия этих индексов у нас на выходе будет вектор размерности, соответствующей размеру словаря. И мы просто берём токен с наибольшим весом из этого вектора. Как вы помните, наша модель возвращает не распределение вероятностей, а [логиты](#), то есть это какие-то ненормированное числа. Чтобы получить распределение вероятностей, в этом случае, нам нужно ещё [софтмакс](#) применить к этим векторам. Но если мы хотим делать только "arg max", то нам не нужен "softmax", потому что он сглаживает величины, приводит их в диапазон от нуля до единицы, но он их не переупорядочивает, а значит точка максимума не изменится. Вот мы выбираем лучший токен, используя всего лишь предсказание модели именно для этой позиции. Если на очередном шаге мы предсказали конец последовательности, то мы прекращаем генерировать дальше. А если это ещё не конец последовательности, то мы добавляем текущий токен к нашему списку токенов и снова засовываем это всё в модель на следующей итерации. Затем, когда мы сделали достаточное количество шагов, мы просто декодируем список номеров токенов в строку (всё просто). Давайте посмотрим, что же наша модель может генерировать. На вход модели мы подаём какой-то небольшой фрагмент текста. Хочу обратить ваше внимание, что мы подаём не только фрагменты текстов, но даже фрагменты слова. То есть — мы не полностью подали последнее слово, и модель сгенерировала на выходе вот такой фрагмент. То есть она, во-первых, закончила слово, а потом продолжила относительно связным текстом. У неё даже получилось смешать французский и русский язык. Обратите внимание, что данная реализация — она не очень эффективная, потому что она полностью перевычисляет всю модель, хотя, казалось бы, мы можем часть активаций сохранить, потому что когда мы в цикле в "greedy\_generator" выбираем следующий лучший токен, мы всё равно берём все предыдущие токены, прогоняем их через модель, и, активации на большом количестве слоёв — они будут те же самые. То есть, казалось бы, их можно закэшировать и не прогонять одни и те же токены через модель повторно. Просто закэшировав активации. Но реализация этого не очень очевидная, более того, она достаточно сложная, и поэтому в семинаре мы не стали её делать. Также на экране вы видите парочку других примеров, которые были сгенерированы моделью из практически того же текста, но с небольшими изменениями. Мы видим что, несмотря на то, что изменения в исходном тексте небольшие, результирующий текст кардинально меняется.

```
def initialize_weights(self):
    for param in self.impl.parameters():
        if param.dim() > 1:
            nn.init.xavier_uniform_(param)

In [23]: torch_transf_model = LanguageModel(tokenizer.vocab_size(),
                                           256,
                                           BatchFirstTransformerEncoder(
                                               nn.TransformerEncoderLayer(
                                                   d_model=256,
                                                   nhead=16,
                                                   dim_feedforward=512,
                                                   dropout=0.1),
                                               num_layers=3),
                                           emb_dropout=0.1)
print('Количество параметров', get_params_number(torch_transf_model))

Количество параметров 2094312
```



```
In [24]: (best_val_loss,
          best_torch_transf_model) = train_eval_loop(torch_transf_model,
                                                      train_dataset,
                                                      test_dataset,
                                                      lm_cross_entropy,
                                                      lr=2e-3,
                                                      epoch_n=2000,
                                                      batch_size=512,
                                                      device='cuda',
                                                      early_stopping_patience=50,
                                                      max_batches_per_epoch_train=1000,
                                                      max_batches_per_epoch_val=1000,
                                                      lr_scheduler_ctor=lr_scheduler)
```

Количество параметров 2094312

```
In [24]: (best_val_loss,
          best_torch_transf_model) = train_eval_loop(torch_transf_model,
                                                      train_dataset,
                                                      test_dataset,
                                                      lm_cross_entropy,
                                                      lr=2e-3,
                                                      epoch_n=2000,
                                                      batch_size=512,
                                                      device='cuda',
                                                      early_stopping_patience=50,
                                                      max_batches_per_epoch_train=1000,
                                                      max_batches_per_epoch_val=1000,
                                                      lr_scheduler_ctor=lr_scheduler)
```

Эпоха 0  
Эпоха: 11 итераций, 2.32 сек  
Среднее значение функции потерь на обучении 6.400123596191406  
Среднее значение функции потерь на валидации 6.247258567810059  
Новая лучшая модель!

Эпоха 1  
Эпоха: 11 итераций, 2.30 сек  
Среднее значение функции потерь на обучении 6.240467071533203  
Среднее значение функции потерь на валидации 6.256922912597656

Эпоха 2  
Эпоха: 11 итераций, 2.29 сек  
Среднее значение функции потерь на обучении 6.2336131876165215  
Среднее значение функции потерь на валидации 6.182693004608154  
Новая лучшая модель!

Эпоха 3



```
Эпоха: 11 итерации, 2.29 сек
Среднее значение функции потерь на обучении 1.2744716730984775
Среднее значение функции потерь на валидации 2.615639400482178
```

```
Эпоха 828
Эпоха: 11 итераций, 2.29 сек
Среднее значение функции потерь на обучении 1.2744015130129727
Среднее значение функции потерь на валидации 2.618060493469238
```

```
Эпоха 829
Эпоха: 11 итераций, 2.28 сек
Среднее значение функции потерь на обучении 1.2691779028285632
Среднее значение функции потерь на валидации 2.617707681655884
```

```
Эпоха 830
Эпоха: 11 итераций, 2.29 сек
Среднее значение функции потерь на обучении 1.2705340602181174
Среднее значение функции потерь на валидации 2.615696668624878
```

```
Эпоха 831
Эпоха: 11 итераций, 2.29 сек
Среднее значение функции потерь на обучении 1.273527513850819
Среднее значение функции потерь на валидации 2.6124579429626467
Модель не улучшилась за последние 50 эпох, прекращаем обучение
```

```
In [25]: torch.save(best_torch_transf_model.state_dict(), './models/war_and_peace_torch_transf_best.pth')
In [26]: torch_transf_model.load_state_dict(torch.load('./models/war_and_peace_torch_transf_best.pth'))
Out[26]: <All keys matched successfully>
```

## Генерация текста с помощью языковой модели



## Генерация текста с помощью языковой модели

### Жадная генерация

```
In [27]: greedy_generator = GreedyGenerator(torch_transf_model, tokenizer)
In [28]: %%time
print(greedy_generator('сказала княжна, оглядывая Бона'))
...
In [29]: print(greedy_generator('смеялась княжна, оглядывая Наполе'))
...
In [30]: print(greedy_generator('сказала княжна, оглядывая Кутуз'))
...
In [31]: print(greedy_generator('сказал Кутузов, оглядывая Наполеона'))
...
```



### Генерация с помощью лучевого поиска - Beam Search

```
In [32]: beam_generator = BeamGenerator(torch_transf_model, tokenizer)
In [33]: %%time
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполе'
```

```

5     target_part = np.array(target_part)
6
7     return seed_part, target_part
8
9
0 class GreedyGenerator:
1     def __init__(self, model, tokenizer, device='cuda', eos_token_id=3):
2         self.model = model
3         self.tokenizer = tokenizer
4         self.device = torch.device(device)
5         self.model.to(self.device)
6         self.eos_token_id = eos_token_id
7
8     def __call__(self, seed_text, max_steps_n=40):
9         seed_tokens = self.tokenizer.encode([seed_text])[0]
0
1         for _ in range(max_steps_n):
2             in_batch = torch.tensor(seed_tokens).unsqueeze(0).to(self.device)
3             best_next_token = self.model(in_batch)[0, -1].argmax()
4             if best_next_token == self.eos_token_id:
5                 break
6
7             seed_tokens.append(best_next_token)
8
9         return self.tokenizer.decode([seed_tokens])[0]
0
1
2 class BeamGenerator:
3     def __init__(self, model, tokenizer, device='cuda', eos_token_id=3):
4         self.model = model
5         self.tokenizer = tokenizer
6         self.device = torch.device(device)
7         self.model.to(self.device)
8
9

```



## Генерация текста с помощью языковой модели

### Жадная генерация

```

In [27]: greedy_generator = GreedyGenerator(torch_transf_model, tokenizer)

In [28]: %%time
        print(greedy_generator('сказала княжна, оглядывая Бона'))
        сказала княжна, оглядывая Бонапарте. - Ah! chere amie, - сказал он ему, - cousin ne m'a parlez pas de cela
        CPU times: user 3.02 s, sys: 47.5 ms, total: 3.07 s
        Wall time: 1.45 s

In [29]: print(greedy_generator('смеялась княжна, оглядывая Наполе'))
        смеялась княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la véritable bonne so

In [30]: print(greedy_generator('сказала княжна, оглядывая Кутуз'))
        сказала княжна, оглядывая Кутузку и сел портретом и пока ноге робко взглядываясь на нее значение на П
        щеголе

In [31]: print(greedy_generator('сказал Кутузов, оглядывая Наполеона'))
        сказал Кутузов, оглядывая Наполеона и отворяясь, показывавшиеся четыре гостей, около рта. - Кто там?
        о удивл

```



### Генерация с помощью лучевого поиска - Beam Search

Более практичный способ декодирования текста (он даёт более хорошие результаты, более качественные тексты) — это использование [лучевого поиска](#). Упрощённый вариант лучевого поиска мы реализовали в классе "BeamGenerator". У него точно такой же интерфейс, как и у "GreedyGenerator". Давайте посмотрим, как он работает. На экране вы видите код лучевого

поиска. На вход функция принимает исходный текст, а также параметры лучевого поиска. Наибольшее количество шагов — это, по сути, наибольшее количество токенов, которые мы можем добавить к исходной последовательности, это количество лучших гипотез, лучших вариантов декодирования, которое нам нужно вернуть из этой функции. Ширина луча — это количество наилучших промежуточных вариантов, которое мы будем хранить в процессе декодирования. Как и в прошлый раз, начинаем мы с того, что преобразовываем текст в последовательность токенов. Две самые важные переменные, которые мы будем обновлять в процессе генерации: первая — это список промежуточных гипотез (или частичных гипотез). Этот список будет содержать пары (то есть кортежи из двух элементов). На первом месте кортежа будет стоять вес, то есть какая-то оценка правдоподобности этой гипотезы, а на втором месте — собственно, сама гипотеза (в виде списка токенов). На самом деле, эта переменная — это не просто список. Мы будем поддерживать эту переменную в виде очереди с приоритетами, то есть мы будем её пересортировывать после добавления нового элемента каждый раз, чтобы на первом месте стояла гипотеза с наилучшим score (с наилучшей оценкой правдоподобности). Второй важный список — это список готовых гипотез, то есть, когда мы уже либо сделали наибольшее количество шагов, либо мы дошли до конца последовательности, то мы прекращаем генерировать из данной гипотезы и мы перемещаем её в список готовых гипотез. Это список, из которого будет формироваться результат работы этой функции. Для того, чтобы было удобно реализовывать очереди с приоритетами, в python есть пакет "heap queue", то есть это "очередь куча". В нём есть базовые операции для работы с кучей и, соответственно, с очередью с приоритетами. Если вы не знаете, что такое "куча" и "очередь с приоритетами", то мы оставим ссылку на материалы, которые вы можете почитать и освежить свои знания в этой области. Функция "heap pop" возвращает нам элемент с головы кучи, то есть это элемент с наименьшим текущим скором. То есть библиотека "heap queue" реализовывает кучу на минимум. То есть, на вершине кучи лежит наименьший элемент. Эта функция возвращает нам элемент списка "partial hypothesis". Как мы помним, этот список содержит кортежи. На первом месте кортежа стоит score, на втором месте кортежа стоит гипотеза в виде списка токенов. Следующим шагом мы кладём нашу текущую гипотезу в модель и получаем новое предсказание для последнего токена. Здесь мы не можем работать с исходными логитами, нам нужно как-то нормировать. Но сами вероятности от нуля до единицы нам не очень удобны, потому что правдоподобность целой гипотезы будет считаться как произведение вероятностей.  $P(A)*P(B)*P(C)...$  и, таким образом, если у нас хотя бы одна из этих вероятностей достаточно маленькая, то всё произведение устремится к нулю. Это очень неудобно — мы очень быстро выйдем за пределы точности вычислений и эта оценка правдоподобия станет бесполезной. Вместо этого мы будем использовать log-вероятность. Напомню, что, если мы работаем с логарифмированными вероятностями, то у нас произведение заменяется на сумму. Да, эти логарифмы — это большие по модулю отрицательные числа, но, с помощью операции сложения, нам гораздо сложнее выйти за пределы точности, и поэтому, на практике, используют часто именно логарифмированные вероятности. Чтобы получить логарифмированные вероятности мы применяем не "softmax", а

"log softmax". А затем выбираем k токенов с наибольшей log-вероятностью из этого списка. Далее мы итерируемся по списку k лучших вариантов и добавляем новые гипотезы в нашу очередь. Как мы это делаем? Во-первых, мы преобразовываем [тензоры](#) в числа — так, чтобы не тащить за собой объекты pytorch. Это нам экономит память, если бы мы этого не делали, то у нас бы утекала память. Затем нам нужно посчитать новую оценку правдоподобности гипотезы. Оценку правдоподобности гипотезы мы считаем как log-вероятность этой гипотезы, делённую на корень из длины этой гипотезы в токенах. Такую дополнительную нормализацию, то есть деление на корень из длины, обычно используют для того, чтобы в процессе поиска мы не предпочитали слишком короткие гипотезы, потому что понятно — когда мы перемножаем много вероятностей или складываем много отрицательных чисел — score, в принципе, сильно падает, и поэтому более короткие гипотезы априорно оказываются более вероятными. Но мы этого не хотим, поэтому мы делим score на корень из длины. Далее мы обновляем нашу гипотезу, то есть дописываем в неё токены, и кладём эту гипотезу либо в список финальных гипотез (если эта гипотеза уже достаточно длинная или мы на этом шаге вы выбрали токен конца последовательности), либо мы кладём её в нашу очередь, когда мы говорим, что — "ага, начиная с этой гипотезы мы можем продолжить наш поиск". Далее мы обрезаем нашу очередь. То есть мы оставляем в нашей очереди только заданное количество наилучших гипотез. Если бы мы этого не делали, то наш поиск, в принципе, бы выродился в полный перебор и нам бы не хватило никакой памяти для этого. Если "beam\_size" равен единице, то, по сути, "[beam search](#)" откатывается, деградируя до "полностью жадного"<sup>[1]</sup> алгоритма. Поэтому мы должны регулировать значением параметра "beam\_size" то, насколько мы хотим перебирать разные варианты. Чем больше "beam\_size", тем больше времени мы потратим, тем больше памяти мы потратим. Но, скорее всего, мы получим более правдоподобный текст (хотя — не факт, если наша модель [переобучилась](#), то неизвестно мы получим более правдоподобный текст). Такие итерации мы повторяем до тех пор, пока наша очередь не пуста. А именно — она пополняется тогда, когда мы кладём в неё частичные гипотезы, не очень длинные — не законченные фрагменты текста. Она перестаёт пополняться тогда, когда у нас уже накапливаются очень длинные гипотезы или когда мы постепенно выбираем в качестве продолжения токен конца последовательности. Итак, пара операций, которые нам нужно сделать уже после цикла. Мы накопили какой-то список финальных гипотез, нам нужно декорировать их в тексты и выбрать заданное количество наилучших гипотез — всё это возвращается назад. То есть, эта функция возвращает нам список пар (вот она возвращает нам список пар). Первый элемент пары — это score, то есть оценка правдоподобности текста, и второй элемент — это, собственно, сам текст. Необходимо обратить внимание на вот это место — у нас token\_score — это отрицательное число, это логарифм вероятности, и чем это число меньше (то есть, чем оно больше по модулю), тем менее вероятен этот токен. А мы помним, что модуль "heap queue" в python реализует кучу на минимум. Поэтому нам нужно, как бы, накапливать score со знаком "минус" — так, чтобы минимальный score был у самой

правдоподобной гипотезы. Поэтому мы здесь используем минус — мы вычитаем из накопленного score, score текущего токена.

[1] Тема жадного декодирования и лучевого поиска также будет рассматриваться позднее в [лекции про seq2seq](#)



### Генерация с помощью лучевого поиска - Beam Search

```
In [32]: beam_generator = BeamGenerator(torch_transf_model, tokenizer)

In [33]: %%time
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполе',
                                     beamsize=5,
                                     return_hypotheses_n=5)

for score, pred_txt in beam_gen_variants:
    print('*****')
    print(score)
    print(pred_txt)
    print()

...
In [34]: %%time
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполе',
                                     beamsize=20,
                                     return_hypotheses_n=20)

for score, pred_txt in beam_gen_variants:
    print('*****')
    print(score)
    print(pred_txt)
    print()

...
In [35]: %%time
```



```
8     self.eos_token_id = eos_token_id
9
0 def __call__(self, seed_text, max_steps_n=40, return_hypotheses_n=5, beamsize=5):
1     seed_tokens = self.tokenizer.encode([seed_text])[0]
2     initial_length = len(seed_tokens)
3
4     partial_hypotheses = [(0, seed_tokens)]
5     final_hypotheses = []
6
7     while len(partial_hypotheses) > 0:
8         cur_partial_score, cur_partial_hypothesis = heapq.heappop(partial_hypotheses)
9
0         in_batch = torch.tensor(cur_partial_hypothesis).unsqueeze(0).to(self.device)
1         next_tokens_logits = self.model(in_batch)[0, -1]
2         next_tokens_logproba = F.log_softmax(next_tokens_logits)
3         topk_continuations = next_tokens_logproba.topk(beamsize)
4
5         for token_score, token_idx in zip(topk_continuations.values, topk_continuations.indices):
6             token_score = float(token_score)
7             token_idx = int(token_idx)
8
9             old_denorm_score = cur_partial_score * np.sqrt(len(cur_partial_hypothesis))
0             new_score = (old_denorm_score - token_score) / np.sqrt(len(cur_partial_hypothesis) + 1)
1
2             new_hypothesis = cur_partial_hypothesis + [token_idx]
3             new_item = (new_score, new_hypothesis)
4
5             if token_idx == self.eos_token_id or len(new_hypothesis) - initial_length >= max_steps_n:
6                 final_hypotheses.append(new_item)
7             else:
8                 heapq.heappush(partial_hypotheses, new_item)
9
0             if len(partial_hypotheses) > beamsize:
1 ...
2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
8 ...
9 ...
```

$p(a, b, c) = p(a)p(b)p(c)$

$\log p(a, b, c) = \log p(a) + \log p(b) + \dots$

$\log p(a, b, c)$

$\sqrt{\text{len}}$



```
7 while len(partial_hypotheses) > 0:
8     cur_partial_score, cur_partial_hypothesis = heapq.heappop(partial_hypotheses)
9
10    in_batch = torch.tensor(cur_partial_hypothesis).unsqueeze(0).to(self.device)
11    next_tokens_logits = self.model(in_batch)[0, :-1]
12    next_tokens_logproba = F.log_softmax(next_tokens_logits)
13    topk_continuations = next_tokens_logproba.topk(beamsz
14
15    for token_score, token_idx in zip(topk_continuations.values, topk_continuations.indices):
16        token_score = float(token_score)
17        token_idx = int(token_idx)
18
19        old_denorm_score = cur_partial_score * np.sqrt(len(cur_partial_hypothesis))
20        new_score = (old_denorm_score - token_score) / np.sqrt(len(cur_partial_hypothesis) + 1)
21
22        new_hypothesis = cur_partial_hypothesis + [token_idx]
23        new_item = (new_score, new_hypothesis)
24
25        if token_idx == self.eos_token_id or len(new_hypothesis) - initial_length >= max_steps_n:
26            final_hypotheses.append(new_item)
27        else:
28            heapq.heappush(partial_hypotheses, new_item)
29
30    if len(partial_hypotheses) > beamsz:
31        partial_hypotheses = heapq.nsmallest(beamsz, partial_hypotheses)
32        heapq.heapify(partial_hypotheses)
33
34    final_scores, final_token_lists = zip(*final_hypotheses)
35    final_texts = self.tokenizer.decode(list(final_token_lists))
36
37    result = list(zip(final_scores, final_texts))
38    result.sort()
39    result = result[:return_hypotheses_n]
```



Итак, как работает "[beam search](#)" мы посмотрели, давайте теперь обратимся к тому, что же он нагенерировал... Первое, что мы видим — это то, что beam search работает дольше. Раньше нам требовалось полторы секунды, чтобы сгенерировать одно предложение, теперь, чтобы сгенерировать 5 вариантов, нам нужно 8 секунд. Пока что, разница небольшая. Надо заметить, что эта реализация "beam search" тоже не самая эффективная, по той же причине — мы никак не используем кэширование. На экране вы видите 5 лучших вариантов, которые получилось сгенерировать, вы видите score, отсортированные по возрастанию (напомню, что самый лучший score — это самый маленький score, потому, что это "минус сумма log-вероятности"). Как вы видите, все тексты содержат большой общий фрагмент, отличается только концовка. Наиболее вероятная причина этого — это чрезмерная уверенность модели. Таким образом, если мы отклоняемся от единственного, наиболее вероятного варианта декодирования, хотя бы чуть-чуть, то score уже очень сильно ухудшается и у других вариантов нету шансов удержаться внутри луча. В качестве домашнего задания я предлагаю вам побороться с этой чрезмерной уверенностью и повысить разнообразие вариантов генерации. Один из возможных способов борьбы с чрезмерной уверенностью — это сглаживание меток, то есть можно перевесить метки так, чтобы они стали менее контрастными, чтобы [кросс-энтропия](#) не давала очень сильный штраф. Давайте посмотрим, как будут меняться списки сгенерированных предложений в зависимости от настроек [лучевого поиска](#). В первом случае мы использовали ширину луча "5". Если использовать ширину луча "20", то разнообразие чуть-чуть подрастает. Видим, что первые "много" примеров по-прежнему содержат большой общий фрагмент. То есть, они были сгенерированы, на самом деле, из одной гипотезы и отличаются только последней парой

шагов. Но дальше мы видим, что, уже какое-то изменение идёт, хотя оно, конечно, тоже не очень большое. По сути, отличие только в том, что здесь дефис добавляется иногда... Другими словами, с такой моделью пока что не получается разнообразия предложить. Другая простая техника для повышения разнообразия предсказаний — это добавление шума в предсказания модели. Как это делать? Мы берём вектор [логитов](#) для очередного токена и добавляем туда, например [гауссовский шум](#) или, более правильно — шум из [распределения Гумбеля](#). Но эта техника немного не информированная, то есть силу этого шума нужно подбирать руками и мы рискуем получить ерунду. Кроме того, процесс декодирования может стать слишком стохастическим и это приведёт к тому, что запуская его несколько раз, мы будем получать абсолютно не пересекающееся множество вариантов декодирования. Это сделает нашу модель банально непредсказуемой, её нельзя будет вообще никак использовать на практике. Так что с этим способом повышения разнообразия нужно быть аккуратней, хотя, на этапе обучения, его вполне можно использовать посмелее. Обратите внимание, что с шириной луча "20" та же самая работа заняла уже не 8 секунд, а почти 31 секунду. Мы можем пойти ещё дальше и выбрать ширину луча "100". Я предлагаю вам самостоятельно поэкспериментировать с этими параметрами.

In [33]: %time

```
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполеоном',
                                    beamsize=5,
                                    return_hypotheses_n=5)

for score, pred_txt in beam_gen_variants:
    print('****')
    print(score)
    print(pred_txt)
    print()

****  

✓ 0.8626641229794534
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  

****  

✓ 0.867215443138969
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  

****  

✓ 0.9491151991653383
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonn  

****  

0.9805023955786255
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon  

****  

1.1298425040190347
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon  

CPU times: user 16.5 s, sys: 150 ms, total: 16.6 s
Wall time: 8.36 s
```



```
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполеоном',
                                    beamsize=20,
                                    return_hypotheses_n=20)

for score, pred_txt in beam_gen_variants:
    print('****')
    print(score)
    print(pred_txt)
    print()

****  

✓ 0.8626641229794534
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  

****  

✓ 0.867215443138969
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  

****  

✓ 0.9491151991653383
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonn  

****  

0.9805023955786255
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon  

****  

1.1298425040190347
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon  

****  

1.1368209431731167
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon
```



```
****  
1.2564065903940718  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne societe, n  
****  
1.285661493447496  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne societe s  
****  
1.3286466933514898  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne societe t  
****  
1.3625253417089107  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  
****  
1.3803647202669411  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne soci  
****  
1.3964590169917475  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bo  
CPU times: user 1min, sys: 554 ms, total: 1min 1s  
Wall time: 30.8 s
```



```
In [35]: %%time  
  
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполе',  
                                     beamsize=100,  
                                     return_hypotheses_n=20)  
  
for score, pred_txt in beam_gen_variants:
```

```
****  
1.3964590169917475  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bonne societe, d  
CPU times: user 1min, sys: 554 ms, total: 1min 1s  
Wall time: 30.8 s
```



```
In [35]: %%time  
  
beam_gen_variants = beam_generator('сказала княжна, оглядывая Наполе',  
                                     beamsize=100,  
                                     return_hypotheses_n=20)  
  
for score, pred_txt in beam_gen_variants:  
    print('****')  
    print(score)  
    print(pred_txt)  
    print()
```

```
****  
0.8626641229794534  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bo  
****  
0.867215443138969  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable  
****  
0.9491151991653383  
сказала княжна, оглядывая Наполеоном, Анна Павловна собрала у себя вечер. La creme de la veritable bon  
****  
0.9805023955786255  
..
```

Из комментариев:

Вопрос:

Почему мы применяем beam search только для генерации уже обученной моделью? Почему мы не обучаем с beam search?

Ответ (Николая Копырина):

если у вас возникла такая идея, предлагаю описать, как это будет происходить. При обучении нас интересует взять новые значения весов и получить новый лосс, чтобы потом пройтись обратно и опять изменить веса. Почему бы не использовать много лоссов для соседних проходов?.. [Наверное это делается](#), и если переписать лосс-функцию, почему бы и нет. Даже есть статья: [Sequence-to-Sequence Learning as Beam-Search Optimization](#).

Ответ (Алексея Шадрикова):

я так понимаю вы предлагаете создать модель, которая будет выдавать целые предложения. В общем-то в модель можно добавлять разные преобразования. При обучении градиентным спуском главное, чтобы у этих преобразований, т.е. функций существовали производные. Лучевой поиск, в принципе, можно рассматривать как расширение макспулинга, т.е проблема производной решаема.

Получается, что в теории проблем нет, дело за практикой ) Осталось проверить будет ли все это обучаться

Доп. комментарий (задавшего вопрос):

Спасибо. Я подумала, что можно было бы брать количество образцов = ширине луча, пытаться приближать настоящие, и отдалять ненастоящие, как в триплетах. И штрафовать за отсутствие настоящих вообще в луче.

Спасибо большое за статью, мне не попалась

Вопрос:

Один из возможных способов борьбы с чрезмерной уверенностью – это сглаживание меток, то есть можно перевесить метки так, чтобы они стали менее контрастными, чтобы кросс-энтропия не давала очень сильный штраф.

Например, с помощью температуры сэмплирования?

Ответ:

например с помощью [label smoothing](#) )

Доп. комментарий (задавшего вопрос):

проверил, [temperature scaling](#) тоже работает)

They demonstrated that a simple post-processing step, temperature scaling, can reduce ECE and calibrate the network. Temperature scaling consists in multiplying the logits by a scalar before applying the softmax operator. Here, we show that label smoothing also reduces ECE and can be used to calibrate a network without the need for temperature scaling.

(<https://proceedings.neurips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-Paper.pdf>)

P.s. Поправил свое первое сообщение (кросс-энтропия не при чем))

Итак, что мы сделали до настоящего момента? Мы загрузили датасет, мы обучили токенизатор, собственно токенизировали датасет, мы реализовали несколько базовых классов и утилит для того, чтобы выполнить обучение. То есть, это класс LanguageModel, это обёртка для [трансформера](#), которая транспонирует [тензоры](#) и инициализирует веса, это функция потерь, это

расписание для изменения длины градиентного шага. Затем мы обучили модель, используя, в качестве backbone (то есть в качестве основной нейросети) реализацию трансформера из библиотеки pytorch. Затем мы взяли два алгоритма декодирования — это "полностью жадный" и "[beam search](#)", то есть умеренно жадный алгоритм, и проверили, что, в целом, модель как-то учится и что-то генерирует. И это, в принципе, выглядит как какой-то более-менее связный текст. Хорошо, давайте пойдём дальше и теперь заменим реализацию трансформера из библиотеки pytorch на нашу реализацию. Будем строить реализацию трансформера по кирпичикам. Самый базовый кирпичик — это "механизм внимания с несколькими головами" (или "multi-head attention"). Это достаточно универсальная реализация механизма внимания, хотя и несколько упрощённая относительно той реализации, которая входит в библиотеку pytorch. На вход механизму внимания подаётся три главных последовательности. Это последовательность запросов, последовательность ключей и последовательность значений. Каждая из этих последовательностей представляется четырёхмерным тензором. Физический смысл измерений этого тензора следующий: первое — это размер батча (как обычно), второе измерение — это длина последовательности, третье измерение — это количество "голов", то есть, по сути это количество независимых механизмов внимания. "Multihead attention" можно реализовать с помощью нескольких "single head attention", то есть, с помощью нескольких простых механизмов внимания, в которых все вычисления производятся независимо и последовательно. Данная реализация "multihead attention" — она более эффективная, потому что все "головы", то есть другими словами, несколько механизмов внимания, вычисляются параллельно. Эффективнее загружаются видеокарты. Так вот, третье измерение — это количество "голов", и четвёртое измерение — это размер вектора. Если мы говорим про последовательность запросов и последовательность ключей, то у них последнее измерение должно быть одинаковое, потому что эти две группы векторов мы будем сравнивать с помощью [скалярного произведения](#). Отдельно есть ещё тензор значений, он тоже четырёхмерный, но у него допускается другое количество элементов в последнем измерении. Кроме этих трёх основных последовательностей, в механизм внимания передаётся две маски. Первая маска — это "маска паддингов". Это прямоугольный тензор, в котором количество строк соответствует количеству примеров в батче, а количество столбцов соответствует максимальной длине примера. Этот тензор состоит из ноликов и единичек, и единичками помечаются те элементы, которые выходят за границы последовательности (то есть это "паддинги") — это те элементы, которые не нужно учитывать. Вторая маска — это маска зависимости позиций. Она одинакова для всех примеров в батче. Эта маска представляется прямоугольной матрицей. В общем случае — прямоугольной, но в нашем семинаре эту функцию мы будем использовать для реализации механизма "[self-attention](#)", то есть "внутреннее внимание", а там длина последовательности запросов и последовательности ключей одинакова (потому что они вычислены из одной и той же исходной последовательности). Поэтому, в нашем случае, это будет всегда квадратная матрица. Мaska зависимости выглядит примерно вот так, как вы видите на экране. Напомню, что строки соответствуют выходным позициям, столбцы — входным позициям, нолик обозначает, что для

Вычисления [вектора признаков](#) для данной выходной позиции можно использовать данную входную позицию. Если в ячейке стоит  $-\infty$  — это значит, что нельзя использовать данную входную позицию для данной выходной.



### Собственная реализация MultiHeadAttention

```
In [36]: def my_multihead_attention(queries, keys, values,
                               keys_padding_mask, dependency_mask,
                               is_training,
                               weights_dropout):
    """
    queries - BatchSize x ValuesLen x HeadN x KeySize
    keys - BatchSize x KeysLen x HeadN x KeySize
    values - BatchSize x KeysLen x HeadN x ValueSize
    keys_padding_mask - BatchSize x KeysLen
    dependency_mask - ValuesLen x KeysLen
    is_training - bool
    weights_dropout - float

    result - tuple of two:
        - BatchSize x ValuesLen x HeadN x ValueSize - resulting features
        - BatchSize x ValuesLen x KeysLen x HeadN - attention map
    """

    # BatchSize x ValuesLen x KeysLen x HeadN
    relevances = torch.einsum('bvhx,bkhs->bvkh', (queries, keys))

    # замаскировать элементы, выходящие за длины последовательностей ключей
    padding_mask_expanded = keys_padding_mask[:, None, :, None].expand_as(relevances)
    relevances.masked_fill_(padding_mask_expanded, float('-inf'))

    # замаскировать пары <выходная позиция, входная позиция>
    relevances = relevances + dependency_mask[None, :, :, None].expand_as(relevances)

    normed_rels = F.softmax(relevances, dim=2)
    normed_rels = F.dropout(normed_rels, weights_dropout, is_training)
```



### Маска зависимостей

```
In [17]: def make_target_dependency_mask(length):
    full_mask = torch.ones(length, length)
    ignore_mask = torch.tril(full_mask) < 1
    full_mask.masked_fill_(ignore_mask, float('-inf'))
    full_mask.masked_fill_(~ignore_mask, 0)
    return full_mask

make_target_dependency_mask(10)
```

```
Out[17]: tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
                  [0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
                  [0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
                  [0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
                  [0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf],
                  [0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf],
                  [0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf],
                  [0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],
                  [0., 0., 0., 0., 0., 0., 0., 0., 0., -inf],
                  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```



### Кодирование позиции

```
In [18]: def make_positional_encoding(max_length, embedding_size):
    time = np.pi * torch.arange(0, max_length).float()
    freq_dividers = torch.arange(1, embedding_size // 2 + 1).float()
    inputs = time[:, None] / freq_dividers[None, :]

    result = torch.zeros(max_length, embedding_size)
```

Кроме этого, мы хотим иметь возможность включать dropout в механизме внимания. Авторы [трансформера](#) рекомендуют делать так. Для этого мы передаём ещё два аргумента. Первый аргумент — это флаг, которые принимает значение "true", когда мы находимся в режиме обучения — это значит, что dropout нужно включить. Если мы не в режиме обучения, то dropout выключается. И второй параметр — это число от нуля до единицы, это сила dropout. Если эта переменная равна 0, то, по сути, dropout выключен. Если — равно 1, то dropout зануляет практически все элементы входной последовательности. Эта функция возвращает кортеж из двух элементов. То есть, эта функция возвращает сразу два [тензора](#). Первый тензор — это основной результат работы механизма внимания. То есть, для каждого запроса и для каждой "головы" найден новый [вектор признаков](#). Второй тензор возвращать не обязательно, но мы это будем делать для целей визуализации. Это тоже четырёхмерный тензор, который показывает нам оценки значимости или [релевантности](#) каждой позиции из последовательности ключей для каждой позиции из последовательности запросов. Самый первый шаг в механизме внимания — это сравнение. Здесь мы берём последовательность запросов, последовательность ключей, берём все возможные пары выходной позиции и входной позиции и находим сходство посредством [скалярного произведения](#). Для того, чтобы красиво записать вычисления всех нужных нам оценок сходства, мы используем так называемую "[эйнштейновскую запись](#)" (мы приложим ссылку для того, чтобы вы могли поизучать, что это такое). Частое это достаточно удобный способ для описания сложных тензорных произведений. Если в двух словах, то — для того, чтобы описать тензорное произведение в эйнштейновской записи мы задаём обозначения для всех измерений входных тензоров. То есть, "b" — это "batch size", "v" — это "количество запросов" (то есть длина последовательности запросов), "h" — это "количество голов", "s" — это размерность вектора признаков для каждой позиции, для каждой головы. То же самое для второго тензора — это тензор ключей, здесь всё то же самое, кроме второго измерения, потому что, теоретически, количество запросов и количество ключей могут отличаться, хотя в данном семинаре это не так. И, затем, после знака "стрелочки" мы описываем измерения результирующего тензора. Те буквы, которые были слева от стрелочки, но их нету справа от стрелочки, соответствуют измерениям, по которым будет производиться скалярное произведение. Конкретно в этой ситуации у нас уходит буковка "s" — это значит, что мы производим скалярное произведение по последнему измерению и немного переворачиваем результирующий тензор — так, чтобы у нас первое измерение соответствовало количеству примеров в батче, второе — количеству запросов (то есть, длине последовательность запросов), третье — количеству ключей и четвёртое количеству "голов". Получаем такой вот четырёхмерный тензор. Надо сказать, что запросы и ключи мы сравниваем только внутри одной "головы" и только внутри одного примера в батче. Затем мы применяем наши маски. Во-первых, мы применяем маску паддингов. Напомню, что эта маска содержит нолики для значимых токенов и единички для токенов выравнивания (паддингов). Применение этой маски заключается в том, что мы записываем  $-\infty$  во все позиции тензора сходства, соответствующие сравнениям какого-либо запроса с ключами, соответствующими паддингу. Напомню, что, когда мы будем применять [софтмакс](#) для того, чтобы получить

нормированные оценки сходства, позиции, на которых стояло  $-\infty$  получат значение "0". Далее, аналогичным образом, мы применяем маску зависимостей. Как мы помним, в маске зависимостей тоже есть "нолики", есть  $-\infty$ . Когда мы к какому-то конечному числу добавляем  $-\infty$ , то результат становится равным  $-\infty$ . Обратите внимание на вот эту странную запись — мы ставим квадратные скобки, как будто хотим выбрать из тензора какие-то строки или столбцы (то есть, проиндексировать тензор), а потом передаём туда None. Что это за ерунда? Такую запись можно использовать для того, чтобы добавить измерения в тензор. Это примерно то же самое, что вызвать несколько раз метод "unsqueeze" из pytorch. Двоеточие соответствует взятию всех элементов по соответствующему измерению. Таким образом в выделенной строчке мы добавляем два измерения на первую позицию (то есть, как бы, добавляем измерения для размера батча), и на последнюю позицию (то есть, добавляем измерения для количества голов). Далее мы применяем softmax по третьему измерению. То есть по измерению, соответствующему количеству ключей (напомню, что измерения нумеруются с нуля). Таким образом, сумма весов в тензоре "normed rels" для каждой выходной позиции, для каждого элемента батча и для каждой "головы" будет равна 1. Далее мы применяем dropout, причём применяем его не на матрицу признаков, как это обычно делается, а мы применяем его поверх нормированных оценок сходства. Таким образом, мы исключаем зависимости каких-то выходных позиций от входных. Это достаточно хороший способ регуляризации в данном случае. Далее нам нужно взять тензор весов (то есть тензор "normed rels") и тензор значений ("values") и перемножить их и сложить так, чтобы получить новый вектор признаков для каждой выходной позиций. Здесь мы сначала добавляем измерение в исходные тензоры, так, чтобы у них были одинаковые количества измерений, а затем применяем операцию поэлементного перемножения. И в numpy, и в pytorch есть так называемый механизм "broadcasting" — это когда у нас формы тензоров не совпадают, а мы к ним как применяем операцию перемножения или сложения. Pytorch и numpy каким-то образом дополняют размерности этих тензоров, так, чтобы они совпали и операцию можно было выполнить. Одно из базовых правил broadcasting — это когда размерность тензора по какому-то измерению равна единице, мы можем по этому измерению его просто клонировать нужное количество раз. Конечно клонирование, фактически, не выполняется — память дополнительно не выделяется, но, как будто бы, мы клонируем. Таким образом, в результате broadcasting, например, вот это измерение — последнее измерение тензора "normed rels expanded" станет равно "value\_size", то есть количеству признаков для каждого значения. В результате процедуры broadcasting и перемножения мы получаем пятимерный тензор — обратите внимание, что единички (вот эти) были автоматически расширены до соответствующей размерности второго тензора. И, наконец, мы сворачиваем полученный тензор по размерности, соответствующей количеству ключей. Таким образом, получаем четырёхмерный тензор, который для каждого примера в батче, для каждой выходной позиции и для каждой головы содержит некоторый вектор признаков, ну, и возвращаем всё.

## Собственная реализация MultiHeadAttention

```
In [36]: def my_multihead_attention(queries, keys, values,
                               keys_padding_mask, dependency_mask,
                               is_training,
                               weights_dropout):
    """
    queries - BatchSize x ValuesLen x HeadN x KeySize
    keys - BatchSize x KeysLen x HeadN x KeySize
    values - BatchSize x KeysLen x HeadN x ValueSize
    keys_padding_mask - BatchSize x KeysLen
    dependency_mask - ValuesLen x KeysLen
    is_training - bool
    weights_dropout - float

    result - tuple of two:
        - BatchSize x ValuesLen x HeadN x ValueSize - resulting features
        - BatchSize x ValuesLen x KeysLen x HeadN - attention map
    """
    # BatchSize x ValuesLen x KeysLen x HeadN
    relevances = torch.einsum('bvhs,bkhs->bvkh', (queries, keys))

    # замаскировать элементы, выходящие за длины последовательностей ключей
    padding_mask_expanded = keys_padding_mask[:, None, :, None].expand_as(relevances)
    relevances.masked_fill_(padding_mask_expanded, float('-inf'))

    # замаскировать пары <выходная позиция, входная позиция>
    relevances = relevances + dependency_mask[None, :, :, None].expand_as(relevances)

    normed_rels = F.softmax(relevances, dim=2)
    normed_rels = F.dropout(normed_rels, weights_dropout, is_training)
```



```
values - BatchSize x KeysLen x HeadN x ValueSize
keys_padding_mask - BatchSize x KeysLen
dependency_mask - QueriesLen x KeysLen
is_training - bool
weights_dropout - float

result - tuple of two:
    - BatchSize x QueriesLen x HeadN x ValueSize - resulting features
    - BatchSize x QueriesLen x KeysLen x HeadN - attention map
"""

# BatchSize x QueriesLen x KeysLen x HeadN
relevances = torch.einsum('bvhs,bkhs->bvkh', (queries, keys))

# замаскировать элементы, выходящие за длины последовательностей ключей
padding_mask_expanded = keys_padding_mask[:, None, :, None].expand_as(relevances)
relevances.masked_fill_(padding_mask_expanded, float('-inf'))

# замаскировать пары <выходная позиция, входная позиция>
relevances = relevances + dependency_mask[None, :, :, None].expand_as(relevances)

normed_rels = F.softmax(relevances, dim=2)
normed_rels = F.dropout(normed_rels, weights_dropout, is_training)

# BatchSize x ValuesLen x KeysLen x HeadN x 1
normed_rels_expanded = normed_rels.unsqueeze(-1)

# BatchSize x 1 x KeysLen x HeadN x ValueSize
values_expanded = values.unsqueeze(1)

# BatchSize x ValuesLen x KeysLen x HeadN x ValueSize
weighted_values = normed_rels_expanded * values_expanded
result = weighted_values.sum(2) # BatchSize x ValuesLen x HeadN x ValueSize
```



```

queries - BatchSize x QueriesLen x HeadN x KeySize
keys - BatchSize x KeysLen x HeadN x KeySize
values - BatchSize x KeysLen x HeadN x ValueSize
keys_padding_mask - BatchSize x KeysLen
dependency_mask - QueriesLen x KeysLen
is_training - bool
weights_dropout - float

result - tuple of two:
    - BatchSize x QueriesLen x HeadN x ValueSize - resulting features
    - BatchSize x QueriesLen x KeysLen x HeadN - attention map
"""

# BatchSize x QueriesLen x KeysLen x HeadN
relevances = torch.einsum('bvhs,bkhs->bvkh', (queries, keys))

# замаскировать элементы, выходящие за длины последовательностей ключей
padding_mask_expanded = keys_padding_mask[:, None, :, None].expand_as(relevances)
relevances.masked_fill_(padding_mask_expanded, float('-inf'))

# замаскировать пары <выходная позиция, входная позиция>
relevances = relevances + dependency_mask[None, :, :, None].expand_as(relevances)

normed_rels = F.softmax(relevances, dim=2)
normed_rels = F.dropout(normed_rels, weights_dropout, is_training)

# BatchSize x QueriesLen x KeysLen x HeadN x 1
normed_rels_expanded = normed_rels.unsqueeze(-1)

# BatchSize x 1 x KeysLen x HeadN x ValueSize
values_expanded = values.unsqueeze(1)

# BatchSize x QueriesLen x KeysLen x HeadN x ValueSize
weighted_values = normed_rels_expanded * values_expanded

```



Из комментариев:

Примечание:

Для тех из нас, кто (пока) не согласен, что эйнштейновская сумма это так уж удобно и понятно.

Интуиция: эйнштейновская запись суммирует произведения пар элементов вдоль указанного общего измерения ( $s$  в 'bvhs,bkhs') -- это аналогично скалярному произведению для векторов одинаковой длины или матричному произведению для матриц с одной размерностью одинаковой длины. Нужно попробовать перемножить матрицы.

Если кто-то хорошо разобрался и может предложить вариант лучше -- предложите :). Я не знаю, что я делаю :).

```

import torch

b = 2 # BatchSize
v = 3 # QueriesLen (да-да; я не виноват -- так в видео)
h = 4 # HeadN
s = 5 # KeySize
k = 6 # KeysLen
queries = torch.randn(b, v, h, s)
keys = torch.randn(b, k, h, s)

einsum_calculation = torch.einsum('bvhs,bkhs->bvkh', (queries, keys))

# Ставлю общие размерности в queries и keys (b и h) в начало;
# две оставшиеся размерности в каждом тензоре транспонирую,
# чтобы получилось (v, s) x (s, k) (для матричного произведения).

```

```

# Размерность queries: (b, v, h, s) --> (b, h, v, s);
# размерность keys: (b, k, h, s) --> (b, h, s, k);
# размерность plain_calculation в результате матричного произведения: (b, h, v, k).
plain_calculation = queries.permute(0, 2, 1, 3) @ keys.permute(0, 2, 3, 1)

# Транспонирую plain_calculation: (b, h, v, k) --> (b, v, k, h),
# чтобы получить размерности, указанные в einsum.
plain_calculation = plain_calculation.permute(0, 2, 3, 1)

print(torch.all(einsum_calculation == plain_calculation)) # >> tensor(True)

```

Примечание:

В лекции упоминается Эйнштейновская запись. Ссылки на Википедию:

[Эйнштейновская запись](#)

[Einstein notation](#)

Это была реализация механизма внимания с несколькими головами и с раздельными последовательностями запросов, ключей и значений. Теперь нам нужно сделать небольшой шагок сторону "[self attention](#)". Это уже кирпичик, имеющий непосредственное отношение к [трансформеру](#). По сути, "self-attention" — это обычный механизм внимания. Самая главная его особенность заключается в том, что ключи, запросы и значения вычисляются из одной и той же последовательности. Давайте посмотрим, как это всё работает. Метод "forward" — на вход нам дают одну последовательность. Эта последовательность представляет батч текстов, то есть у нас есть какое-то количество примеров в батче, наибольшая длина текста и последнее измерение соответствует вектору признаков, то есть рабочему размеру модели. Что мы делаем дальше? Дальше мы берём три простых линейных слоя. По сути, каждый линейный слой параметризуется квадратной матрицей. В принципе, не обязательно брать именно линейный слой — можно навернуть сюда более сложную архитектуру, но... не нужно, чаще всего. Итак, мы берём три этих линейных слоя и применяем их к исходной последовательности. Таким образом, мы преобразовываем [вектор признаков](#) каждого входного элемента независимо от других элементов. Затем мы немного меняем форму, то есть мы последнее измерение (оно обозначается "model size") разбиваем на два измерения — это количество голов (то есть, по сути, это количество независимых механизмов внимания) и новое количество признаков. Это новое количество признаков равняется, по сути, "model size" делить на количество голов. Как вы видите, запросы, ключи и значения получаются абсолютно одинаково. Отличие заключается только в том, что веса (вот этих) преобразований независимы и они сходятся к разным значениям в ходе обучения. Далее мы берём эти три [тензоры](#), берём маски, которые нам передали свыше на вход, и передаём в функцию, которую мы только что рассмотрели — функцию "my multihead attention". Также мы передаём туда флаг "self train" — этот флаг есть у каждого экземпляра класса "torch.[nn.module](#)". Он автоматически обновляется, в зависимости от

того, находится ли вся модель в режиме обучения, или в режиме предсказания. Как мы помним, функция multihead attention возвращает нам четырёхмерный тензор, мы схлопываем последнее измерение обратно в model size для того, чтобы слои self-attention можно было комбинировать с любыми другими слоями. То есть это удобно — когда форма тензора не меняется после преобразования. А также мы сохраняем карту активации — это позволит нам потом эти карты активации нарисовать и, возможно, получить какие-то знания о том, как работает модель (но — не факт). Важный момент здесь — это вызывать метод "detach". Если мы не будем его вызывать, то у нас будет попросту утекать память, потому что каждый тензор хранит ссылки на тензоры, из которых он был получен, и, таким образом, мы будем хранить ссылки на предыдущие батчи, даже. Поэтому, если мы не хотим дальше использовать какой-то тензор в обучении, не хотим прокидывать через него производную, то всегда лучше сделать "detach". Итак, это был блок "self attention". Вернёмся немножко назад — к конструктору нашего класса "self attention", посмотрим, что же нам нужно, чтобы создать экземпляр этого класса. Нам нужен размер модели (суммарное количество признаков по всем головам, model\_size), нам нужно количество голов и нам нужен коэффициент dropout. Важно, чтобы размер модели ("model\_size") делился нацело на количество голов.

**Self-Attention - это Attention, в котором ключи, значения и запросы вычисляются из элементов одной и той же последовательности**

```
In [37]: class MyMultiheadSelfAttention(nn.Module):
    def __init__(self, model_size, n_heads, dropout=0):
        super().__init__()
        assert model_size % n_heads == 0, 'Размерность модели должна делиться нацело на количество голов'
        self.n_heads = n_heads

        self.queries_proj = nn.Linear(model_size, model_size)
        self.keys_proj = nn.Linear(model_size, model_size)
        self.values_proj = nn.Linear(model_size, model_size)

        self.dropout = dropout

        self.last_attention_map = None

    def forward(self, sequence, padding_mask, dependency_mask):
        """
        sequence - BatchSize x Len x ModelSize
        padding_mask - BatchSize x Len
        dependency_mask - Len x Len

        result - BatchSize x Len x ModelSize
        """
        batch_size, max_len, model_size = sequence.shape

        queries_flat = self.queries_proj(sequence) # BatchSize x Len x ModelSize
        queries = queries_flat.view(batch_size, max_len, self.n_heads, -1)

        keys_flat = self.keys_proj(sequence) # BatchSize x Len x ModelSize
        keys = keys_flat.view(batch_size, max_len, self.n_heads, -1)
```



```
        self.keys_proj = nn.Linear(model_size, model_size)
        self.values_proj = nn.Linear(model_size, model_size)

        self.dropout = dropout

        self.last_attention_map = None

    def forward(self, sequence, padding_mask, dependency_mask):
        """
        sequence - BatchSize x Len x ModelSize
        padding_mask - BatchSize x Len
        dependency_mask - Len x Len

        result - BatchSize x Len x ModelSize
        """
        batch_size, max_len, model_size = sequence.shape

        queries_flat = self.queries_proj(sequence) # BatchSize x Len x ModelSize
        queries = queries_flat.view(batch_size, max_len, self.n_heads, -1)

        keys_flat = self.keys_proj(sequence) # BatchSize x Len x ModelSize
        keys = keys_flat.view(batch_size, max_len, self.n_heads, -1)

        values_flat = self.values_proj(sequence) # BatchSize x Len x ModelSize
        values = values_flat.view(batch_size, max_len, self.n_heads, -1)

        # BatchSize x Len x HeadsN x ValueSize
        result, att_map = my_multihead_attention(queries, keys, values,
                                                padding_mask, dependency_mask,
                                                self.training, self.dropout)
        result_flat = result.view(batch_size, max_len, model_size)

        self.last_attention_map = att_map.detach()

        return result_flat
```



```
self.keys_proj = nn.Linear(model_size, model_size)
self.values_proj = nn.Linear(model_size, model_size)

self.dropout = dropout

self.last_attention_map = None

def forward(self, sequence, padding_mask, dependency_mask):
    """
    sequence - BatchSize x Len x ModelSize
    padding_mask - BatchSize x Len
    dependency_mask - Len x Len

    result - BatchSize x Len x ModelSize
    """
    batch_size, max_len, model_size = sequence.shape

    queries_flat = self.queries_proj(sequence) # BatchSize x Len x ModelSize
    queries = queries_flat.view(batch_size, max_len, self.n_heads, -1)

    keys_flat = self.keys_proj(sequence) # BatchSize x Len x ModelSize
    keys = keys_flat.view(batch_size, max_len, self.n_heads, -1)

    values_flat = self.values_proj(sequence) # BatchSize x Len x ModelSize
    values = values_flat.view(batch_size, max_len, self.n_heads, -1)

    # BatchSize x Len x HeadsN x ValueSize
    result, att_map = my_multihead_attention(queries, keys, values,
                                              padding_mask, dependency_mask,
                                              self.training, self.dropout)
    result_flat = result.view(batch_size, max_len, model_size)

    self.last_attention_map = att_map.detach()

    return result_flat
```



OK, два базовых кирпичика у нас уже есть — это механизм внимания с несколькими головами и механизм "[self-attention](#)" (то есть, механизм внутреннего внимания). Теперь мы можем собрать слой "[трансформер](#)". Давайте сразу перейдём к методу forward. Интерфейс у метода forward — такой же, как и у self attention, то есть он принимает на вход входную последовательность — это трёхмерный [тензор](#) (батч на длину на размер модели), это маска паддингов (прямоугольная матрица) и маска зависимости позиции (это квадратная матрица). Первым делом мы здесь выполняем агрегацию контекста. То есть мы хотим понять смысл каждого токена в контексте всей входной последовательности. Для этого мы вызываем механизм "self attention". Затем, к признакам, которые мы получили из механизма внимания, мы применяем dropout и складываем с исходными признаками последовательности. То есть мы здесь получаем, как бы, "skip connection", или это вам может напоминать блок [ResNet](#). Skip connection очень хорошо помогают учить глубокие нейросети. "Skip connection" или "вычисления без нелинейностей" можно применять в абсолютно любых архитектурах, в рекуррентках, свёрточных нейросетях, вот — в трансформере они тоже применяются. И затем к полученным признакам мы применяем "[Layer norm](#)". Вообще, в обработке последовательностей, такие способы нормализации как "Batch norm" не очень удобно применять, потому что они накапливают статистики и непонятно, как вообще статистики считать, когда у нас количество элементов последовательности вообще отличается — может меняться от батча к батчу. Наиболее часто используемый способ нормализации в обработке текстов — это "Layer norm", его используют и в [рекуррентках](#), и в трансформере. Хорошо, теперь переменная "sequence" у нас содержит признаки с учётом контекста. Механизм

внимания хорошо учитывает контекст, но, как нелинейность, он не очень мощный, поэтому давайте применим некоторую нейросеть (независимо) к признакам каждого токена. В классическом трансформере для этого используется двухслойный перцептрон (с двумя линейными слоями, функцией активации ReLU и с dropout). Применять эту нелинейность мы также будем через ResNet-блок, то есть мы суммируем вход нелинейности и выход нелинейности. И, также, второй раз применяем "Layer norm". Таким образом, один слой трансформера состоит из двух residual слоёв (то есть двух блоков со skip connection) и первый блок содержит self attention (то есть, агрегирует контекст), а второй блок преобразовывает признаки каждого элемента последовательности независимо от других элементов последовательности. То есть, больше играет роль нелинейности и обогащает модель. Мы сделали уже три базовых кирпичика — это механизм внимания с несколькими головами, это self attention и это один слой трансформера. Теперь нам нужно собрать "encoder трансформер". По сути, это просто стопка из нескольких слоёв трансформера, который мы только что описали. Для того чтобы создать энкодер для трансформера, мы должны взять несколько экземпляров класса "my transformer encoding layer" (это класс, который мы только что описали) и слепить из них последовательность. Кроме того, мы должны здесь проинициализировать веса всех этих слоёв. Эти слои применяются последовательно, каждый принимает на вход результат работы предыдущего. Кроме того, мы передаём в каждый слой одни и те же маски — это маска паддингов и маска зависимостей. Названия переменных здесь выбраны так, чтобы они соответствовали названиям параметров в реализации pytorch, для того, чтобы мы могли просто взять экземпляр этого класса и подставить его вместо стандартного трансформера. Ну что ж, создаём экземпляр языковой модели уже с нашей реализацией трансформера, передаём туда примерно те же самые параметры. Здесь можно видеть, что количество параметров в нашей реализации немножко меньше, чем в стандартной. В принципе мы не задавались целью скопировать реализацию один в один. Вы можете открыть исходники pytorch и сами посмотреть — там всё достаточно понятно, хотя реализация механизма внимания с несколькими головами там очень общая и достаточно громоздкая. В этом семинаре мы сделали более простую реализацию, отказавшись от некоторой гибкости, в угоду понятности и простоте. Обучаем нашу модель... Вы можете заметить, что наша реализация медленнее, стандартная реализация проходит эпоху за 2 с копейками секунды, а здесь требуется 6. Наиболее вероятная причина замедления — это то, что мы использовали перемножение тензоров с помощью энштейновского суммирования, то есть помощью функции "torch einsum". Эта функция очень удобная, она гибкая, но она помедленнее. Ничего удивительного в этом нет — бесплатных завтраков не бывает<sup>[1]</sup>. Мы бы тоже могли обойтись без этой функции, но нам бы пришлось сделать несколько дополнительных транспонирований, в результате мы бы получили гораздо менее читаемый код. Нашей модели также требуется порядка 8-9 тысяч градиентных шагов для того, чтобы сойтись. Ну, и давайте проверим, вообще, работает ли наша модель, генерирует ли она что-то осмысленное... В целом — да. Заметьте, что из того же начального предложения эта модель

выдаёт другое предложение, в отличие от первой модели, которую мы обучили в этом семинаре. На самом деле, если мы запустим обучение ещё раз, оно сойдётся к другому минимуму и эта модель тоже будет генерить что-то другое. Помните, из функции, которая реализует механизм внимания с несколькими головами, мы возвращали ещё и карты внимания (то есть тензор нормированных релевантностей — то, что получается после [софтмакса](#) и после dropout). Мы это делали не случайно, а для того, чтобы иметь возможность нарисовать эти карты активации. Я не буду подробно останавливаться на алгоритме, который используется для отрисовки этих графиков. Здесь мы видим несколько графиков, каждая вот такая вот строка соответствует слою трансформера (то есть это самый первый слой трансформера, дальше идёт второй слой трансформера...), а каждый столбец соответствует голове (то есть, это — первые головы, это — вторые головы для каждого слоя). Можно больше, у нас в модели всего 16 голов, но, просто — не влезло бы. По строкам здесь отложены запросы, и каждая позиция помечена токеном во входной последовательности, который стоял на этой позиции. По столбцам отложены ключи. Здесь у нас запросы (queries), а здесь у нас ключи (keys). И, чем ярче клеточка на пересечении строки и столбца, тем более значим этот ключ для этого запроса (по мнению модели). Мы видим, что на первом слое карты активации очень контрастные, очень разреженные. То есть, для каждой выходной позиции близок только один ключ. Интересно обратить внимание вот на эти строки. Эти стройки соответствуют токенам, то есть [N-граммам](#) из имени Бонапарта. Интересно, что для почти всех N-грамм из имени, наиболее значимым ключом является первая N-грамма этого имени (то есть у нас есть такие связи). То есть для вычисления признаков (вот, например, для этой позиции) очень важны признаки начала имени. Это и есть учёт контекста. У второй головы карта активации совершенно другая. Здесь, кажется, в вычислении признаков для всех токенов после запятой самый важный исходный токен — это сама "запятая", то есть мы, как бы, отделяем деепричастный оборот здесь. Конечно, это всего лишь — мои попытки натянуть какую-то рационализацию на эти карты активации, неочевидно, что вообще эти карты активации для человека будут понятны или полезны. Но иногда на них просто интересно посмотреть. На втором слое трансформера карты активации уже более сложенные, уже нету каких-то отдельных, выделенных, наиболее значимых позиций, уже каждая позиция обращает внимание, в принципе, на все предыдущие позиции. Обратите внимание, что верхняя половина этих карт — тёмная. Это именно то, чего мы хотели добиться, применяя маску зависимостей. То есть здесь в маске зависимости на этих позициях стоит  $-\infty$ .

[1] [https://en.wikipedia.org/wiki/No\\_free\\_lunch\\_theorem](https://en.wikipedia.org/wiki/No_free_lunch_theorem)

**Один слой трансформера - Self-Attention, Feed-Forward, skip-connections, LayerNorm**

```
In [38]: class MyTransformerEncoderLayer(nn.Module):
    def __init__(self, model_size, n_heads, dim_feedforward, dropout):
        super().__init__()
        self.self_attention = MyMultiheadSelfAttention(model_size,
                                                       n_heads,
                                                       dropout=dropout)
        self.first_dropout = nn.Dropout(dropout)
        self.first_norm = nn.LayerNorm(model_size)
        self.feedforward = nn.Sequential(
            nn.Linear(model_size, dim_feedforward),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(dim_feedforward, model_size),
            nn.Dropout(dropout)
        )
        self.second_norm = nn.LayerNorm(model_size)
    def forward(self, sequence, padding_mask, dependency_mask):
        att_features = self.self_attention(sequence, padding_mask, dependency_mask)
        sequence = sequence + self.first_dropout(att_features)
        sequence = self.first_norm(sequence)
        sequence = sequence + self.feedforward(sequence)
        sequence = self.second_norm(sequence)
        return sequence
```



## Энкодер Трансформера - стопка из нескольких слоёв

**Энкодер Трансформера - стопка из нескольких слоёв**

```
In [39]: class MyTransformerEncoder(nn.Module):
    def __init__(self, n_layers, **layer_kwargs):
        super().__init__()
        self.layers = nn.ModuleList([
            MyTransformerEncoderLayer(**layer_kwargs)
            for _ in range(n_layers)
        ])
        self.initialize_weights()

    def forward(self, sequence, mask, src_key_padding_mask):
        for layer in self.layers:
            sequence = layer(sequence, src_key_padding_mask, mask)
        return sequence

    def initialize_weights(self):
        for param in self.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)
```



## Попробуем обучить языковую модель с нашим Трансформером

## Попробуем обучить языковую модель с нашим Трансформером

```
In [40]: my_transf_model = LanguageModel(tokenizer.vocab_size(),
                                       256,
                                       MyTransformerEncoder(
                                           n_layers=3,
                                           model_size=256,
                                           n_heads=16,
                                           dim_feedforward=512,
                                           dropout=0.1),
                                       emb_dropout=0.1)
print('Количество параметров', get_params_number(my_transf_model))
Количество параметров 1896936
```

```
In [41]: (best_val_loss,
          best_my_transf_model) = train_eval_loop(my_transf_model,
                                                train_dataset,
                                                test_dataset,
                                                lm_cross_entropy,
                                                lr=2e-3,
                                                epoch_n=2000,
                                                batch_size=512,
                                                device='cuda',
                                                early_stopping_patience=50,
                                                max_batches_per_epoch_train=1000,
                                                max_batches_per_epoch_val=1000,
                                                lr_scheduler_ctor=lr_scheduler)
```

```
In [42]: torch.save(best_my_transf_model.state_dict(), './models/war_and_peace_my_transf_best.pth')
```



```
Количество параметров 1896936
In [41]: (best_val_loss,
          best_my_transf_model) = train_eval_loop(my_transf_model,
                                                train_dataset,
                                                test_dataset,
                                                lm_cross_entropy,
                                                lr=2e-3,
                                                epoch_n=2000,
                                                batch_size=512,
                                                device='cuda',
                                                early_stopping_patience=50,
                                                max_batches_per_epoch_train=1000,
                                                max_batches_per_epoch_val=1000,
                                                lr_scheduler_ctor=lr_scheduler)
```

Эпоха 0  
Эпоха: 11 итераций, 6.02 сек  
Среднее значение функции потерь на обучении 6.404186725616455  
Среднее значение функции потерь на валидации 6.248231697082519  
Новая лучшая модель!

*torch.einsum*

Эпоха 1  
Эпоха: 11 итераций, 6.01 сек  
Среднее значение функции потерь на обучении 6.2512031901966445  
Среднее значение функции потерь на валидации 6.239421558380127  
Новая лучшая модель!

Эпоха 2  
Эпоха: 11 итераций, 6.01 сек  
Среднее значение функции потерь на обучении 6.241438735615123  
Среднее значение функции потерь на валидации 6.232439422607422  
Новая лучшая модель!

Эпоха 3



Среднее значение функции потерь на обучении 1.6361751773140647  
Среднее значение функции потерь на валидации 2.7253012657165527

Эпоха 917  
Эпоха: 11 итераций, 5.99 сек  
Среднее значение функции потерь на обучении 1.630537835034457  
Среднее значение функции потерь на валидации 2.724262523651123  
Модель не улучшилась за последние 50 эпох, прекращаем обучение

```
In [42]: torch.save(best_my_transf_model.state_dict(), './models/war_and_peace_my_transf_best.pth')
```

```
In [43]: my_transf_model.load_state_dict(torch.load('./models/war_and_peace_my_transf_best.pth'))
```

...

## Наша реализация - жадная генерация

```
In [44]: my_greedy_generator = GreedyGenerator(my_transf_model, tokenizer)
```

```
In [45]: my_greedy_generator('сказала княжна, оглядывая Андрея')
```

```
Out[45]: "сказала княжна, оглядывая Андрея. - Ах, как! Господа, - сказала она с Лизой, - не г'афия, - сказала  
езы что-то злоб"
```

## Визуализация карт внимания

```
In [46]: def plot_attention_maps(model, input_string, tokenizer, device='cuda', max_heads=2, figsize=(16, 10)):  
    device = torch.device(device)
```

```
    token_ids = tokenizer.encode([input_string])[0]
```

## Визуализация карт внимания

```
In [46]: def plot_attention_maps(model, input_string, tokenizer, device='cuda', max_heads=2, figsize=(16, 10)):  
    device = torch.device(device)
```

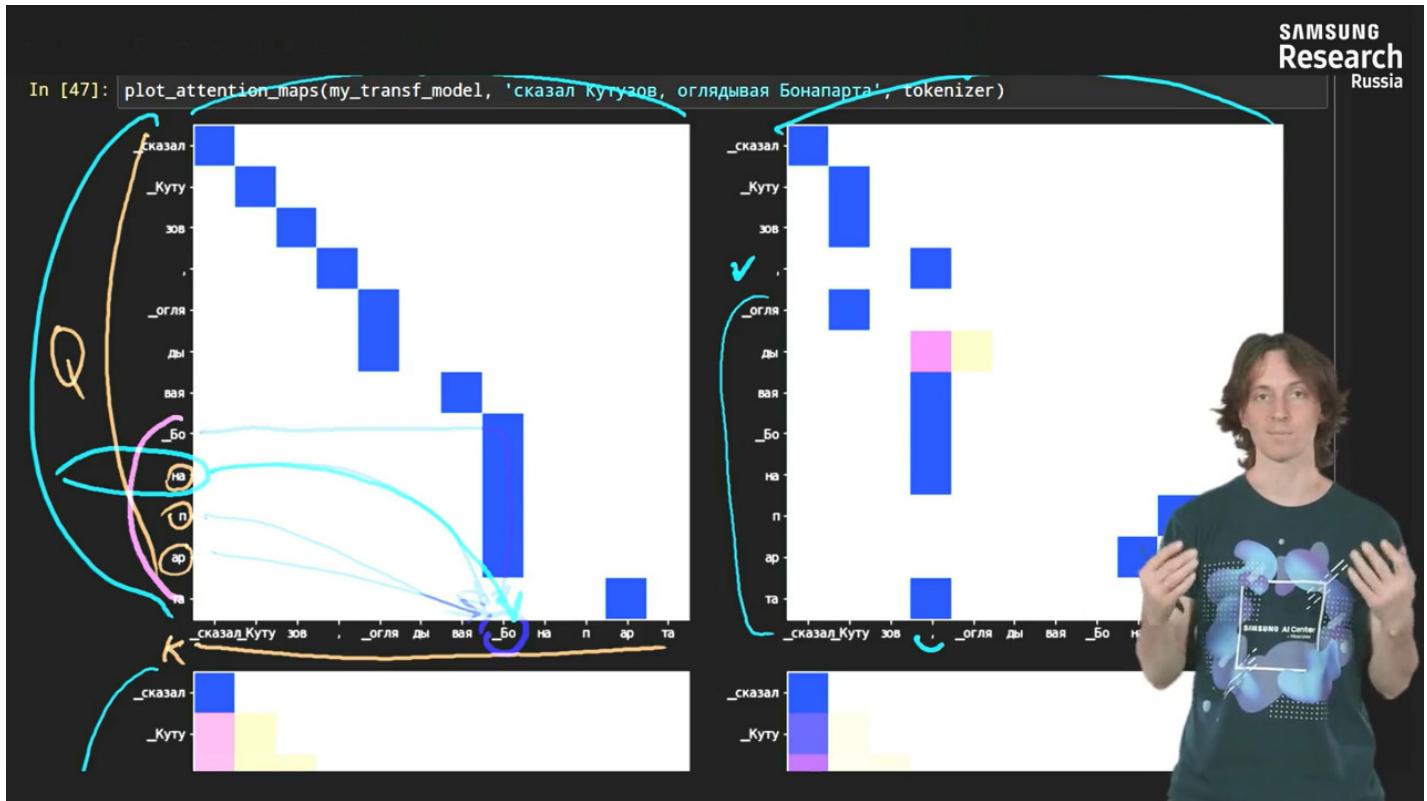
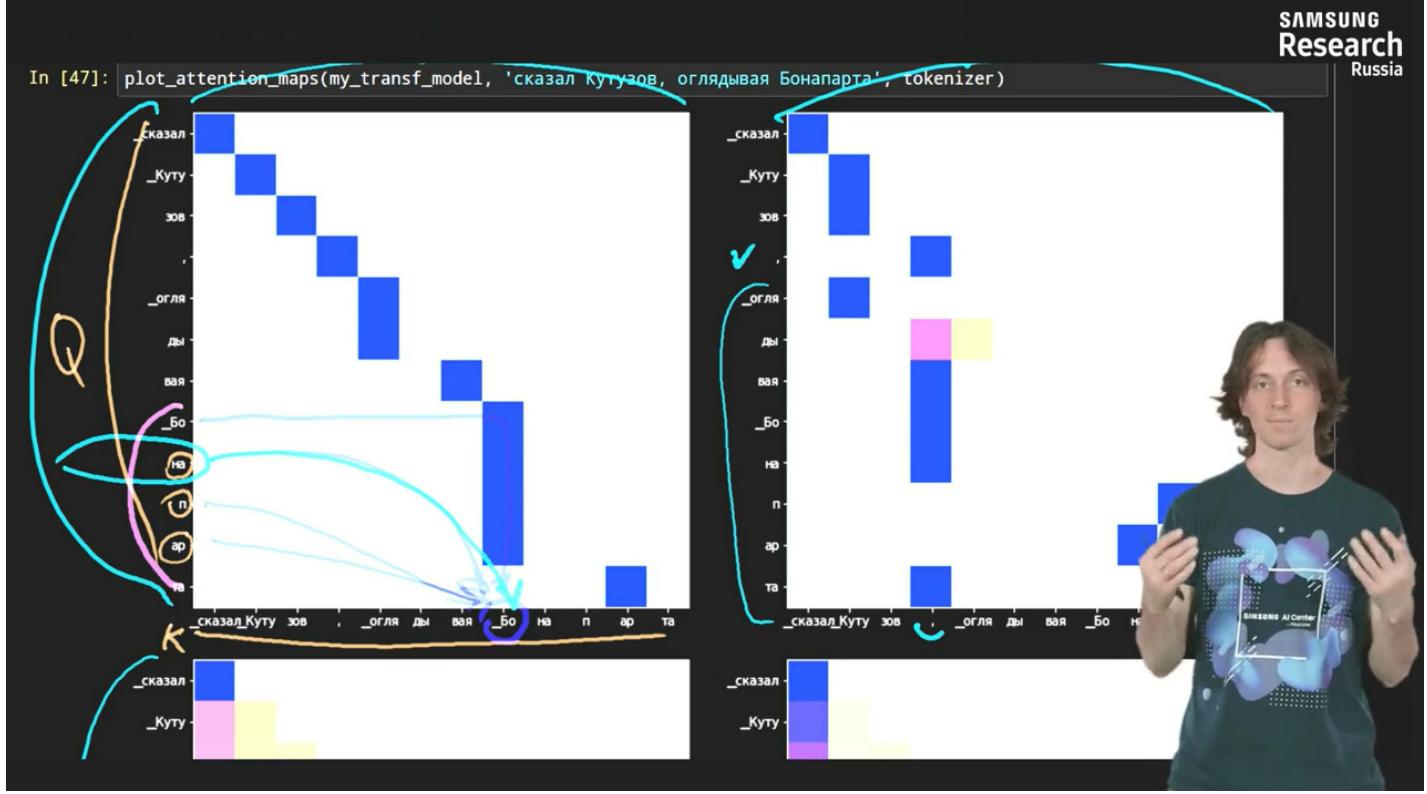
```
    token_ids = tokenizer.encode([input_string])[0]
```

```
    token_strs = [tokenizer.id_to_subword(i) for i in token_ids]  
    in_len = len(token_ids)  
    ticks = np.arange(0, in_len)
```

```
    model.to(device)  
    model.eval()
```

```
    in_batch = torch.tensor(token_ids).unsqueeze(0).to(device)  
    model(in_batch)
```

```
    for module in model.modules():  
        if isinstance(module, MyMultiheadSelfAttention):  
            cur_last_attention_map = module.last_attention_map[0].cpu().numpy()  
            n_heads = cur_last_attention_map.shape[-1]  
            n_heads_to_vis = min(n_heads, max_heads)  
  
            fig, axes = plt.subplots(1, n_heads_to_vis)  
            fig.set_size_inches(figsize)  
            for head_i in range(n_heads_to_vis):  
                ax = axes[head_i]  
                ax.imshow(cur_last_attention_map[..., head_i])  
  
                ax.set_yticks(ticks)  
                ax.set_ylim(bottom=in_len - 0.5, top=-0.5)  
                ax.set_yticklabels(token_strs)  
  
                ax.set_xticks(ticks)
```



Из комментариев:

Вопрос:

А почему мы передаем маски во все слои? Разве не нужно маскировать только первый?

Ответ (Алексея Шадрикова):

если мы не знаем информацию о будущем на этапе предсказания, то и в обучении нигде не должны ее использовать и основываться только на предыдущих токенах/признаках.

Ну, что ж, это был большой семинар, который был посвящён сразу нескольким темам — а именно, [моделированию языка](#), [byte pair encoding](#) (то есть какая-то токенизация универсальная, современная, которая позволяет выбирать между длиной последовательностей и размером словаря). Мы попробовали применить такую токенизацию путём использования библиотеки "[YouTokenToMe](#)". Затем мы собрали и обучили языковую модель, используя реализацию [трансформера](#) из стандартной библиотеки pytorch, мы попробовали погенерировать тексты с помощью "[полностью жадного" алгоритма](#) и с помощью "[лучевого поиска](#)". А затем мы рассмотрели — а как же можно руками, самостоятельно, не используя готовой библиотеки, реализовать механизм внимания — используя только базовые операции из pytorch. И, таким образом, мы собрали свой энкодер для трансформера, реализовав механизм внимания с несколькими головами, реализовав "[self attention](#)", реализовав отдельный слой трансформера и собрав это всё в encoder. А также мы обучили нашу реализацию и увидели, что она, в принципе, работает примерно так же, как и стандартная реализация — значит, что, кажется, мы не ошиблись. А ещё, напоследок, мы заглянули внутрь обученной модели и посмотрели, как между собой связываются входные и выходные позиции на разных уровнях и для разных голов. Спасибо за внимание!

