

Stepik. Neural networks and NLP. 6. Transfer learning - Part 2

6.2 Семинар: PyTorch-Transformers, или... как мне запустить BERT?

Для данного семинара Вам потребуется ноутбук [task9_bert_sentiment_analysis.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! На этом семинаре мы рассмотрим такую популярную в последнее время [нейронную сеть](#), как [BERT](#). Как вы знаете из лекции, зачастую, при отсутствии более чем одной [GPU](#), даже [файнтюнинг](#) BERT может занять существенное количество времени. Поэтому мы рассмотрим, наверное, самую простую задачу, на решение которой будем файнтюнить BERT: будем [классифицировать предложения](#). А именно, мы решим достаточно простую, но интересную задачу: будем определять эмоциональную окраску твитов с помощью BERT. Для дообучения BERT на такую простую задачу вам не потребуется много вычислительных ресурсов — при желании, можно дообучить сеть даже на [CPU](#). Итак, мы возьмём предобученный BERT и

добавим к нему слой нейронов, чтобы обучить полученную модель для классификации предложений. Возникает резонный вопрос — если, в среднем, файнтонинг BERT занимает много времени и много ресурсов, зачем же нам это нужно, если можно взять, и с нуля обучить какую-нибудь сеть (например, [LSTM](#)) для решения той же самой задачи? Давайте отвечать на этот вопрос постепенно. Во-первых, веса BERT уже содержат много информации о том, как устроен язык. А значит, на файнтонинг модели уйдёт совсем немного времени. Авторы статьи про BERT советуют запускать всего от двух до четырёх эпох, чтобы получить хороший результат. Кроме того, для файнтурина нам нужно достаточно мало данных — небольшого количества данных, зачастую, оказывается достаточно, чтобы получить хорошие результаты. Ну, ещё одна причина немаловажная. Один и тот же BERT (одну и ту же сеть с одними и теми же весами) можно использовать как основу для дообучения для совершенно разных задач, начиная от классификации и заканчивая [вопросно-ответными системами](#) или [машиенным переводом](#), или чем-то ещё. Зачастую это гораздо проще и приятнее, чем разбираться в деталях недавно вышедшей статьи со сложной архитектурой, заточенной под решение конкретной задачи.

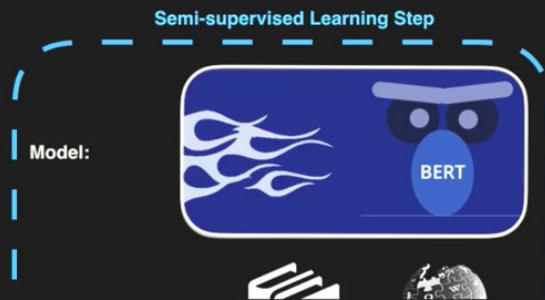
SAMSUNG
Research
Russia

Определение эмоциональной окраски твитов с помощью BERT

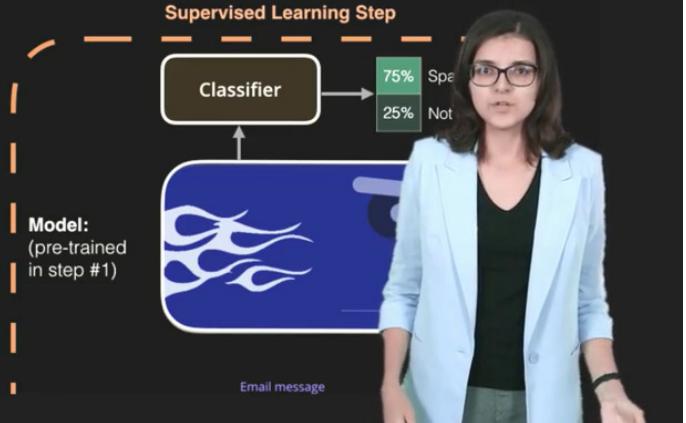
Введение

- 1 - **Semi-supervised** training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.



- 2 - **Supervised** training on a specific task with a labeled dataset.

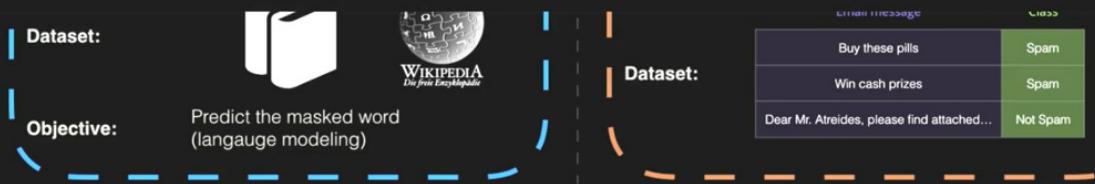


Итак, мы немного обсудили [BERT](#), повторили чуть-чуть материал лекции и теперь давайте начнём кодить. Мы постарались максимально упростить этот урок. Сегодня мы не будем кодить руками обучение BERT, а покажем, как, максимально просто, максимально быстро, найти готовый предобученный BERT и, минимальным количеством кода, [дофайнтонить](#) его для решения вашей задачи. Это значит что мы будем использовать готовые библиотеки. Одна из них называется [pytorch-transformers](#). В ней есть pytorch-интерфейс BERT, а также в этой библиотеке есть удобные обёртки и для других популярных моделей (например, для XL [трансформера](#), или [GPT-2](#), или каких-то других моделей — например, для RoBERTa^[2]). На

сегодняшний день, пожалуй, эта библиотека является самой популярной обёрткой для удобного использования BERT. Таким образом, если вы не являетесь мастером tensorflow и хотите быстро получить хороший результат для вашей прикладной задачи, вам совсем не придётся разбираться с исходным кодом модели на tensorflow, достаточно просто воспользоваться этой замечательной библиотекой. Ещё один факт: обратите внимание, что раньше эта библиотека называлась по-другому: pytorch-pretrained-BERT, и на гитхабе вы можете найти код с именно такими импортами. Стоит помнить, что это — одна и та же библиотека, просто более ранние её версии. Ещё про эту библиотеку хочется отметить, что в ней есть не только удобное выкачивание предобученного BERT, но и специальные модификации для конкретных задач. Например, для [задачи классификации](#) предложений (классификации текстов), которую мы собственно и будем использовать. Итак, для начала, ставим эту библиотеку. У меня она уже установлена, поэтому я не буду запускать этот кусочек кода. Пойдем дальше. Дальше посмотрим внимательно на данные, на которых мы будем дообучать нашу модель. Мы выбрали достаточно интересный датасет с разметкой сентимента русскоязычных твитов.^[1] Подробнее про этот датасет и про его сбор, создание этого датасета, можно почитать в этой статье — по ссылке. [Корпус](#), который мы будем использовать, можно выкачать по ссылке тут. Корпус содержит около 115 тысяч записей с положительной эмоциональной окраской и около 112 000 — с отрицательной эмоциональной окраской. В принципе, не очень много. И этот датасет был размечен вручную. Давайте посмотрим на наши данные. Загружаем наши тексты и смотрим на произвольные 5 записей. Как вы видите, лексика здесь достаточно неформальная, ну собственно, как в твитах. Есть какие-то слова (вот такие, например) забавные, есть смайлики. Есть какая-то очень странная, сильно эмоциональная пунктуация... Ну, и будет интересно посмотреть, как BERT справится вот с такими текстами (достаточно неформальными). Теперь давайте подготовим наши данные так, чтобы их можно было подать в BERT для дообучения. Обратите внимание на специальные токены — "CLS" и "SEP", которые мы будем добавлять в начало и конец наших предложений. Как вы помните, именно такой формат нужен BERT для того, чтобы работать с входными предложениями. Давайте запустим этот кусочек кода. Посмотрим, что всё у нас нормально — все размерности сошлись, и распечатаем пример — ну, например, 1000-ое предложение. Видим какой-то твит — в начале метка "CLS", в конце — метка "SEP". Замечательно. Теперь давайте поделим наши данные на обучающую выборку и холдаут (holdout) — то есть, некоторые данные отложим и не будем их использовать в процессе дообучения, а затем на этой отложенной выборке померим качество нашей дообученной модели. Отлично, делим наши данные в пропорции 70/30%, смотрим, какие данные у нас получились: в обучающей выборке примерно 158 000 записей, а в холдауте 68 000. Отлично.

[1] Корпус коротких текстов Юлии Рубцовой: <http://study.mokoron.com/>

[2] Liu Y. et al. [Roberta: A robustly optimized bert pretraining approach](#) //arXiv preprint arXiv:1907.11692. – 2019.



Установка библиотек

```
In [1]: pip install pytorch-transformers
Collecting pytorch-transformers
  Downloading https://files.pythonhosted.org/packages/a3/b7/d3d18008a67e0b968d1ab93ad44fc05699403fa662f67508b/pytorch_transformers-1.2.0-py3-none-any.whl (176kB)
    100% |██████████| 184kB 10.2MB/s eta 0:00:01
Requirement already satisfied: boto3 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.9.214)
Collecting tqdm (from pytorch-transformers)
  Downloading https://files.pythonhosted.org/packages/e1/c1/bc1dba38b48f4ae3c4428aea669c5e27bd5a7642ff86/tqdm-4.36.1-py2.py3-none-any.whl (52kB)
    100% |██████████| 61kB 18.8MB/s eta 0:00:01
Collecting sacremoses (from pytorch-transformers)
  Downloading https://files.pythonhosted.org/packages/1f/8e/ed5364a06a9ba720fddd9820155cc57300d28f5e642/sacremoses-0.0.35.tar.gz (859kB)
    100% |██████████| 860kB 9.7MB/s eta 0:00:01
Collecting sentencepiece (from pytorch-transformers)
  Downloading https://files.pythonhosted.org/packages/14/3d/efb655a670b98f62ec32d66954e1109f403db3a29/sentencepiece-0.1.83-cp36-cp36m-manylinux1_x86_64.whl (1.0MB)
    100% |██████████| 1.0MB 11.8MB/s eta 0:00:01
Requirement already satisfied: torch>=1.0.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.9.214)
Requirement already satisfied: torchtext>=0.3.1 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.3.1)
Requirement already satisfied: transformers>=2.8.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2.8.0)
Requirement already satisfied: tokenizers>=0.10.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.10.0)
Requirement already satisfied: datasets>=0.12.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.12.0)
Requirement already satisfied: numpy>=1.19.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.19.5)
Requirement already satisfied: sentencepiece>=0.1.83 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.1.83)
Requirement already satisfied: sacremoses>=0.0.35 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.0.35)
Requirement already satisfied: regex>=2019.8.19 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2019.8.19)
Requirement already satisfied: torch>=1.0.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.9.214)
Requirement already satisfied: torchtext>=0.3.1 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.3.1)
Requirement already satisfied: transformers>=2.8.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2.8.0)
Requirement already satisfied: tokenizers>=0.10.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.10.0)
Requirement already satisfied: datasets>=0.12.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.12.0)
Requirement already satisfied: numpy>=1.19.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.19.5)
Requirement already satisfied: sentencepiece>=0.1.83 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.1.83)
Requirement already satisfied: sacremoses>=0.0.35 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.0.35)
Requirement already satisfied: regex>=2019.8.19 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2019.8.19)
Requirement already satisfied: torch>=1.0.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.9.214)
Requirement already satisfied: torchtext>=0.3.1 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.3.1)
Requirement already satisfied: transformers>=2.8.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2.8.0)
Requirement already satisfied: tokenizers>=0.10.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.10.0)
Requirement already satisfied: datasets>=0.12.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.12.0)
Requirement already satisfied: numpy>=1.19.0 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (1.19.5)
Requirement already satisfied: sentencepiece>=0.1.83 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.1.83)
Requirement already satisfied: sacremoses>=0.0.35 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (0.0.35)
Requirement already satisfied: regex>=2019.8.19 in /home/stepic/anaconda3/lib/python3.6/site-packages (from pytorch-transformers) (2019.8.19)
```



```
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /home/stepic/anaconda3/lib/python3.6/site-packages (from requests->pytorch-transformers) (1.22)
Requirement already satisfied: certifi>=2017.4.17 in /home/stepic/anaconda3/lib/python3.6/site-packages (from requests->pytorch-transformers) (2018.11.29)
Requirement already satisfied: docutils<0.16,>=0.10 in /home/stepic/anaconda3/lib/python3.6/site-packages (from boto3->boto3->pytorch-transformers) (0.14)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1; python_version >= "2.7" in /home/stepic/anaconda3/lib/python3.6/site-packages (from boto3->boto3->pytorch-transformers) (2.6.1)
Building wheels for collected packages: sacremoses, regex
  Running setup.py bdist_wheel for sacremoses ... done
  Stored in directory: /home/stepic/.cache/pip/wheels/63/2a/db/63e2909042c634ef551d0d9ac825b2b0b32dede4a6d87ddc94
  Running setup.py bdist_wheel for regex ... done
  Stored in directory: /home/stepic/.cache/pip/wheels/90/04/07/b5010fb816721eb3d6dd64ed5cc8111ca23f97fdab8619b5be
Successfully built sacremoses regex
Installing collected packages: tqdm, joblib, sacremoses, sentencepiece, regex, pytorch-transformers
Successfully installed joblib-0.14.0 pytorch-transformers-1.2.0 regex-2019.8.19 sacremoses-0.0.35 sentencepiece-0.1.83 tqdm-4.36.1
You are using pip version 18.1, however version 19.2.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```



```
In [2]: import torch
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from pytorch_transformers import BertTokenizer, BertConfig
from pytorch_transformers import AdamW, BertForSequenceClassification
from tqdm import tqdm, trange
import pandas as pd
import io
import numpy as np
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

```
In [4]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
import numpy as np
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

```
In [4]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Загрузка данных

Мы выбрали достаточно необычный датасет с разметкой сентимента русскоязычных твитов (подробнее про него в [статье](#)). В корпусе, который мы использовали 114,911 положительных и 111,923 отрицательных записей. Загрузить его можно [тут](#).

```
In [26]: import pandas as pd
```

```
pos_texts = pd.read_csv('positive.csv', encoding='utf8', sep=';', header=None)
neg_texts = pd.read_csv('negative.csv', encoding='utf8', sep=';', header=None)
```

```
In [27]: pos_texts.sample(5)
```

```
Out[27]:
```

	0	1	2	3	4	5	6	7	8	9
28784	409619124911079424	1386495784	freshnichek	девушки, забирайте все по своим ценам!!!...	1	0	0	0	2001	5060
104623	411122578707578880	1386854235	Lily_Gabrielyan	@SolarEclipse57 ПИЧАЛЬНО :Dи А Я ТО ДУМАЛ У Н...	1	0	0	0	8491	352
27496	409591763431473152	1386489260	nastyagrint	Доброе утро:)\\nМама уехала, я сегодня целый де...	1	0	0	0	145	12
76174	410695837728395264	1386752492	rajasimib	гыги, скоро моск взорвётся от загруженой в нег...	1	0	0	0	260	229
76186	410695977922613248	1386752526	pinkpanter_vs	@Gau_Bee у меня дома шерстяная елка - плетена...	1	0	0	0	4887	10

Обратите внимание на специальные токены [CLS] и [SEP], которые мы добавляем в начало и конец предложения.



76174	410695837728395264	1386752492	rajasimib	гыги, скоро моск взорвётся от загруженой в нег...	1	0	0	0	260	229	205	0
76186	410695977922613248	1386752526	pinkpanter_vs	@Gau_Bee у меня дома шерстяная елка - плетена...	1	0	0	0	4887	108	108	1

Обратите внимание на специальные токены [CLS] и [SEP], которые мы добавляем в начало и конец предложения.

```
In [28]: sentences = np.concatenate([pos_texts[3].values, neg_texts[3].values])

sentences = "[CLS] " + sentence + "[SEP]" for sentence in sentences]
labels = [[1] for _ in range(pos_texts.shape[0])] + [[0] for _ in range(neg_texts.shape[0])]
```

```
In [29]: assert len(sentences) == len(labels) == pos_texts.shape[0] + neg_texts.shape[0]
```

```
In [30]: print(sentences[1000])
```

[CLS] Дим, ты помогаешь мне, я тебе, все взаимно, все правильно) [SEP]

```
In [31]: from sklearn.model_selection import train_test_split

train_sentences, test_sentences, train_gt, test_gt = train_test_split(sentences, labels, test_size=0.2)
```

```
In [32]: print(len(train_gt), len(test_gt))
```

158783 68051



Inputs

Теперь импортируем токенизатор для БЕРТА, который превратит наши тексты в набор токенов, соответствующих тем, что во

Из комментариев:

Комментарий:

А подскажите пожалуйста, в видео используется "bert-base-uncased" судя по описанию с transformers, это не multilingual моделька, а чисто англоязычная. Почему она хорошо работает на задаче классификации русских текстов?

Ответ:

попробуйте multilingual. Она ещё лучше.

Вопрос:

А если мы классифицируем текст, а не одно предложение. Мы метки [CLS] и [SEP] ставим в начале и конце текста, или в начале и конце каждого предложения в тексте?

Ответ (Романа Суворова):

насколько я знаю, [CLS] всегда должна стоять на первой позиции, а [SEP] - после последовательности (если мы классифицируем один текст, а не пару). Логика в том, что изначально BERT учится делать сразу несколько задач и модель понимает, что ей надо делать, именно по первому токену (она связывает эмбеддинг этого токена и позицию). К тому же, BERT учится не на предложениях, а на произвольных фрагментах текста не длиннее 512 токенов (уже после wordpiece токенизации). Цитата из статьи:

Throughout this work, a “sentence” can be an arbitrary span of contiguous text, rather than an actual linguistic sentence.

To generate each training input sequence, we sample two spans of text from the corpus, which we refer to as “sentences” even though they are typically much longer than single sentences (but can be shorter also). ... They are sampled such that the combined length is \leq 512 tokens.

To speed up pretraining in our experiments, we pre-train the model with sequence length of 128 for 90% of the steps. Then, we train the rest 10% of the steps of sequence of 512 to learn the positional embeddings.

Переходим к следующему шагу. Теперь импортируем токенизатор для [BERT](#), который превратит наши тексты в набор токенов, соответствующих тем, что встречались в словаре предобученной модели. Давайте импортируем из [pytorch-transformers](#) "bert_tokenizer" и затем загрузим модель, которая называется "bert_base_uncased". Это означает, что мы используем токенизатор для модели BERT Base — это та самая модель, которая меньше, чем BERT Large, и содержит внутри 12 [self-attention](#) модулей. "Uncased" означает, что все слова в словаре этого токенайзера написаны в нижнем регистре (то есть, все слова написаны с маленькой буквы). Давайте запустим этот кусочек кода и посмотрим, на какие токены наш токенайзер разобьёт, ну, например, первое предложение из наших данных. А пока этот кусочек кода выполняется, давайте более подробно посмотрим, в каком формате нужно предоставить BERT наши данные. BERT требует специальный формат входных данных. Нам нужно предоставить четыре сущности. Первая называется `input_ids` — это последовательность чисел, которая отождествляет каждый токен с его номером в словаре. Вторая сущность называется `labels` — это вектор из нулей и единиц. В нашем случае, нули и обозначают негативную эмоциональную окраску, а единицы — положительную эмоциональную окраску. А ещё есть две сущности, два параметра, не обязательных. Они называются `segment_mask` и `attention_mask`. Давайте посмотрим на них чуть подробнее. `segment_mask` — это последовательность нулей и единиц, которая показывает, состоит ли наш входной текст из двух предложений или из одного. Соответственно, в случае одного предложения, мы просто подаём вектор из одних нулей. Зачем это нужно? Как вы

помните, BERT предобучается на двух задачах. Первая задача — это предсказание маскированных слов в тексте, а вторая задача — определение, является ли одно предложение продолжением другого предложения. И, соответственно, для того, чтобы дать BERT информацию о том, есть ли у нас два предложения в нашем входе, либо одно, нам нужен этот параметр `segment_mask`. Следующий параметр ([attention mask](#)) — это также последовательность нулей и единиц, где единицы обозначают токены предложения, а нули — паддинг. Паддинг нужен для того, чтобы BERT мог работать с предложениями разной длины. Ну, например, в нашем случае, мы можем сказать, что максимальная длина предложения у нас будет равна 100, можно выбрать какую-то другую длину, это не сильно важно. И теперь более длинные предложения мы будем обрезать до 100 токенов, а для более коротких предложений использовать паддинг. Для того, чтобы добавить паддинг нашим предложениям, мы будем использовать готовую функцию из keras под названием `pad_sequences`. Собственно, вот здесь написан код, который добавляет паддинг к нашим предложениям. Тем временем, у нас отработал код по токенизации наших предложений и, как мы видим, первое предложение из нашей обучающей выборки разбилось на токены, почти все токены здесь состоят из одной буквы, но встречаются иногда и двухбуквенные токены, иногда встречаются символы пунктуации. Хорошо, теперь нам нужно поделить наши данные на тренировку и валидацию. Как вы помните, отложенную выборку мы уже отложили в переменную `test_data` или `test_sentences`, и теперь давайте поделим оставшийся данные на обучение и валидацию. Делаем мы это точно так же, с помощью функции `train_test_split`, делим в пропорции 90% оставшихся данных — на обучение, 10% — на валидацию. Дальше мы превращаем наши данные в pytorch [тензоры](#), и давайте посмотрим на формат, в котором у нас лежат наши лэйблы. Обратите внимание на формат лейблов в наших данных. Мы подаём не просто `list()` из нулей и единиц, а мы подаём `list(list())`. Это нужно для того, чтобы поддерживать также возможность работы с задачами, где мы каждому объекту присваиваем несколько классов. Например, если бы мы хотели присваивать некоторые метки (лейблы) тем для наших предложений (ну, или скорее для наших текстов), то мы хотели бы уметь присваивать несколько тем одному тексту. В нашем же более простом случае, у каждого предложения есть только одна метка — положительная или отрицательная эмоциональная окраска. Но, тем не менее, данные мы подаём в таком формате. Теперь, что осталось? Нужно создать итераторы с помощью [DataLoader](#). Данные по батчам мы будем разбивать произвольно с помощью [RandomSampler](#) (здесь). Разбитие данных на батчи позволит нам более эффективно использовать память во время обучения, поскольку нам не придётся загружать весь датасет в память. И, также, обратите внимание на размер батча. Здесь мы пишем число "32", но, если вы обнаружите, что во время тренировки у вас появляется, "memory error" — значит, можно попробовать уменьшить размер батча (все как обычно). Отлично, на этом процесс предобработки данных завершён. Давайте переходить к обучению модели.

```
In [32]: print(len(train_gt), len(test_gt))  
158783 68051
```

Inputs

Теперь импортируем токенизатор для БЕРТа, который превратит наши тексты в набор токенов, соответствующих тем, что встречаются в словаре предобученной модели.

```
In [25]: from pytorch_transformers import BertTokenizer, BertConfig  
  
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)  
  
tokenized_texts = [tokenizer.tokenize(sent) for sent in train_sentences]  
print (tokenized_texts[0])  
  
['[CLS]', 'н', '#e', '#p', '##n', '##и', '##с', '##ы', '##в', '##а', '##т', '##ъ', '##с', '#', '#', '##а', '##', '##е', '##м', 'в', '2', '-', 'x', 'м', '##е', '##с', '##т', '##а', '##х', 'о', '#', '##ов', '##п', '##е', '##м', '##е', '##н', '##н', '##о', ' ', ' ', 'э', '##т', '##о', 'с', '##т', '#', '#', '#н', '##о', 'к', '##а', '##к', 'т', '##о', ')', '[SEP]']
```

БЕРТу нужно предоставить специальный формат входных данных.

- **input_ids**: последовательность чисел, отождествляющих каждый токен с его номером в словаре.
- **labels**: вектор из нулей и единиц. В нашем случае нули обозначают негативную эмоциональную окраску, единицы - позитивную.
- **segment mask**: (необязательно) последовательность нулей и единиц, которая показывает, состоит ли входной текст из одного или двух предложений. Для случая одного предложения получится вектор из одних нулей. Для двух: нулей и единиц.
- **attention mask**: (необязательно) последовательность нулей и единиц, где единицы обозначают токены предложения, нули - паддинг.



- **segment mask**: (необязательно) последовательность нулей и единиц, которая показывает, состоит ли входной текст из одного или двух предложений. Для случая одного предложения получится вектор из одних нулей. Для двух: нулей и единиц.
- **attention mask**: (необязательно) последовательность нулей и единиц, где единицы обозначают токены предложения, нули - паддинг.

```
In [14]: input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]  
input_ids = pad_sequences(  
    input_ids,  
    maxlen=100,  
    dtype="long",  
    truncating="post",  
    padding="post"  
)  
attention_masks = [[float(i>0) for i in seq] for seq in input_ids]
```

Делим данные на тренировку и валидацию:

```
In [15]: train_inputs, validation_inputs, train_labels, validation_labels = train_test_split(  
    input_ids, train_gt,  
    random_state=42,  
    test_size=0.1  
)  
  
train_masks, validation_masks, _, _ = train_test_split(  
    attention_masks,  
    input_ids,  
    random_state=42,  
    test_size=0.1  
)
```

```
In [16]: train_inputs = torch.tensor(train_inputs)  
train_labels = torch.tensor(train_labels)  
train_masks = torch.tensor(train_masks)
```



```
        input_ids, train_gt,
        random_state=42,
        test_size=0.1
    )

    train_masks, validation_masks, _, _ = train_test_split(
        attention_masks,
        input_ids,
        random_state=42,
        test_size=0.1
)
```

```
In [16]: train_inputs = torch.tensor(train_inputs)
train_labels = torch.tensor(train_labels)
train_masks = torch.tensor(train_masks)
```

```
In [17]: validation_inputs = torch.tensor(validation_inputs)
validation_labels = torch.tensor(validation_labels)
validation_masks = torch.tensor(validation_masks)
```

```
In [18]: train_labels
Out[18]: tensor([[1],
[1],
[1],
...,
[1],
[0],
[1]])
```

Теперь создаем итераторы с помощью DataLoader. Данные по батчам будем разбивать произвольно с помощью RandomSampler.



Теперь создаем итераторы с помощью DataLoader. Данные по батчам будем разбивать произвольно с помощью RandomSampler.

```
In [19]: train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_dataloader = DataLoader(
    train_data,
    sampler=RandomSampler(train_data),
    batch_size=32
)
```

```
In [20]: validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_dataloader = DataLoader(
    validation_data,
    sampler=SequentialSampler(validation_data),
    batch_size=32
)
```

Обучение модели

Загружаем `BertForSequenceClassification`:

```
In [21]: from pytorch_transformers import AdamW, BertForSequenceClassification
```

Аналогичные модели есть и для других задач. Все они построены на основе одной и той же архитектуры и различаются только

```
In [22]: from pytorch_transformers import BertForQuestionAnswering, BertForTokenClassification
```

Загружаем BERT ("bert-base-uncased"):



Из комментариев:

Вопрос:

Насколько правильно использовать bert-base-uncased для русского языка?

Ведь эта модель обучалась на английский текстах. Видимо именно поэтому токенизатор разбивает текст в примере на отдельные буквы.

Ответ:

насколько я понимаю, там используется модель, которая обучалась на 104 языках.

Доп. комментраий задающего вопрос:

судя по [доке](#), та что на 100+ языках, должна называться bert-base-multilingual-uncased

Ответ:

подозреваю, что это *BERT-Base, Multilingual Uncased (Orig, not recommended): 102 languages, 12-layer, 768-hidden, 12-heads, 110M parameters*

Доп. комментраий задающего вопрос:

согласен, разбиение каждого токена на отдельные символы это по сути фолбэк, самый неудачный вариант. Об этом ранее говорилось в этом курсе.

Вопрос:

Если я правильно понял. То BERT токеном являются символы а не слова или там что-то типа Byte Pair Encoding?

Ответ (Романа Суворова):

в BERT используется аналог BPE - wordpieces ([\[1\]](#), [\[2\]](#), [\[3\]](#), [\[4\]](#)). Этот алгоритм на этапе обучения выявляет устойчивые n-граммы символов, а при предсказании разбивает текст на наиболее подходящую последовательность этих n-грамм. Каждую n-грамму можно представить как числом (её номером в словаре), так и последовательностью символов (как в этом видео).

Вопрос:

Что означают двойные решетки в обозначении токенов? В некоторых они есть, в некоторых нет:

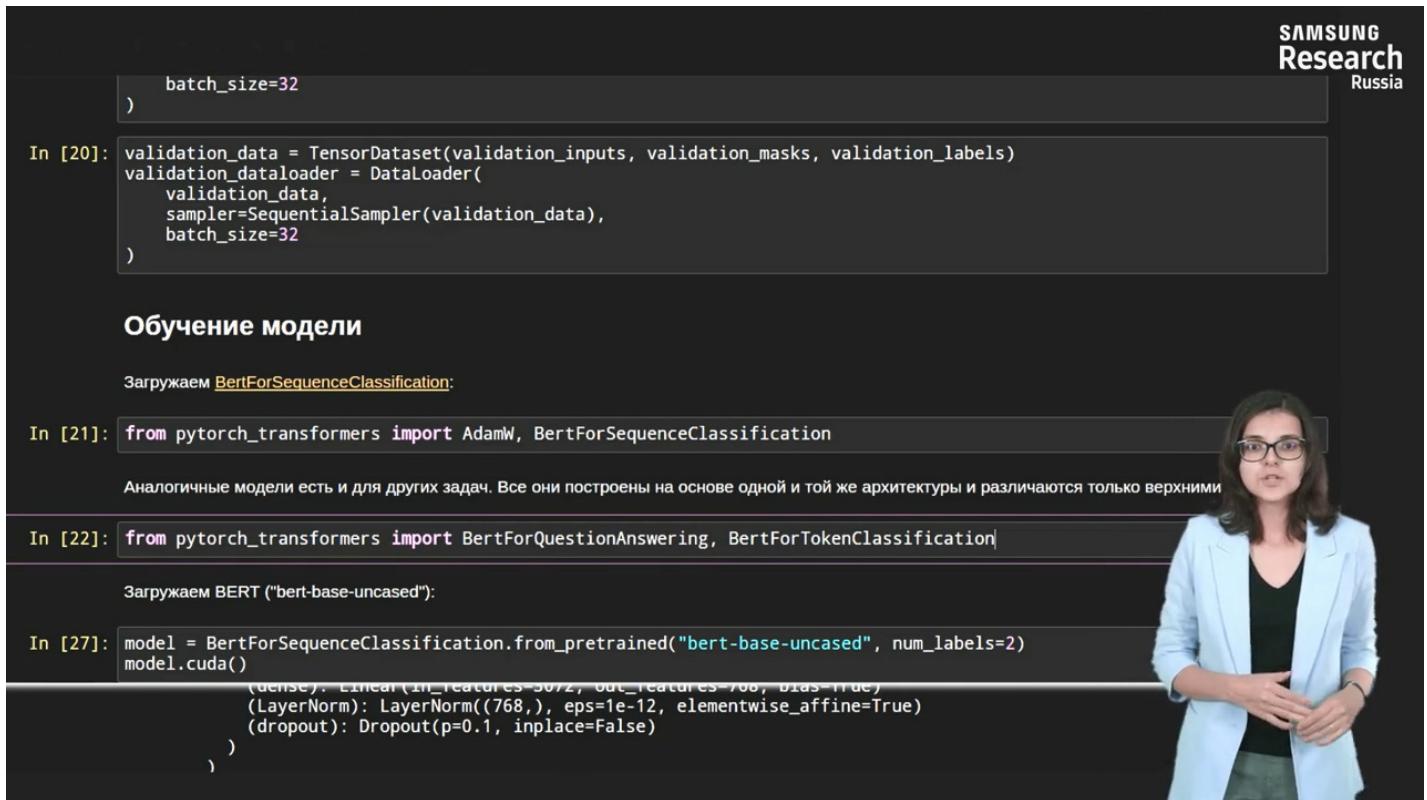
'в', '##c', '##e', 'з', '##a'

Ответ :

с решетками это так сказать "суффиксы" и "окончания", т.е. часто встречающиеся дополнения к другим токенам

Для начала мы хотим изменить предобученный [BERT](#) так, чтобы он выдавал метки для классификации. А затем — [дофайнтюнить](#) полученную сеть на наших данных. Мы берём готовую модификацию BERT для классификации из [pytorch-transformers](#), она называется "BertForSequenceClassification". Импортируем её. Это обычный BERT с добавленным одним линейным слоем для классификации. Аналогичные модели есть и для решения других задач, например, есть "BertForQuestionAnswering", есть "BertForTokenClassification". Все эти модели построены на основе одной и той же архитектуры и различаются только верхними слоями. Теперь давайте чуть-чуть подробнее рассмотрим сам процесс файнтюнинга. Как мы помним, первый токен в каждом предложении в наших данных — это метка "CLS". Скрытое состояние, относящееся к этой метке, должно содержать в себе агрегированное представление всего предложения, которое дальше будет использоваться для классификации. Таким образом, когда мы скормили предложение в процессе обучения сети, выходом должен быть вектор со

скрытым состоянием, относящийся к метке "CLS". Дополнительный полносвязный слой, который мы добавили, имеет размер ["hidden state", "количество классов"] — это двухмерный вектор. В нашем случае, количество классов равно "2", то есть, на выходе мы получим два числа, представляющих классы "положительная эмоциональная окраска", "отрицательная эмоциональная окраска". Сам процесс дообучения достаточно дешёв. По факту, мы тренируем наш верхний слой и немного меняем веса во всех остальных слоях. Иногда некоторые слои специально замораживают или применяют разные стратегии работы с learning rate. В общем делают всё, чтобы сохранить хорошие веса в нижних слоях и ускорить процесс дообучения. В целом, замораживание слоёв BERT обычно не сильно сказывается на итоговом качестве, однако стоит помнить о тех случаях, когда домен для предобучения и дообучения был разным. Например, когда мы предобучили нашу сеть на каких-то официальных текстах (на правовых актах или на научных статьях), а дообучаем её на, например, твитах, на неформальной лексике. В таких случаях лучше тренировать все слои сети, не замораживая ничего.



The screenshot shows a Jupyter Notebook interface with a dark theme. In the top right corner, there is a logo for 'SAMSUNG Research Russia'. The notebook contains the following code:

```
batch_size=32
)
In [20]: validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_dataloader = DataLoader(
    validation_data,
    sampler=SequentialSampler(validation_data),
    batch_size=32
)
```

Обучение модели

Загружаем [BertForSequenceClassification](#):

```
In [21]: from pytorch_transformers import AdamW, BertForSequenceClassification
```

Аналогичные модели есть и для других задач. Все они построены на основе одной и той же архитектуры и различаются только верхними

```
In [22]: from pytorch_transformers import BertForQuestionAnswering, BertForTokenClassification
```

Загружаем BERT ("bert-base-uncased"):

```
In [27]: model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.cuda()
        (cls): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
```

A woman with glasses and a blue blazer is standing on the right side of the screen, speaking.

Из комментариев:

Вопрос:

И еще вопрос про последнее высказывание, что лучше не замораживать БЕРТ, если новые данные. А не лучше ли сначала заморозить все слои, дообучить последний новый слой, а потом уже разморозить оставшие и дообучать уже все вместе? Если я не ошибаюсь, если сразу все тренировать на новом сете, градиенты будут высокие (из-за инициализации верхнего слоя) и проходя через backprop они испортят все веса более начальных слоев. Или же я ошибаюсь?

Ответ:

градиенты будут высокие только если дообучать с высоким learning rate. Но дообучают обычно с низким lr и это приведёт к долгому и неоптимальному обучению последнего слоя. Так что вы

правы, лучше сначала обучить голову с высокой скоростью обучения, а потом разморозить тело берта и дообучать всё с маленькой скоростью.

Вопрос:

А можно как-то с помощью этой библиотеки не делать файнтюнинг на каком.-то новом задании, а скорее дообучить Masked LM и NSP на своем датасете? Буду благодарна за совет :)

Ответ:

в библиотеке есть BertForMaskedLM. Для NSP, видимо, придётся дописать свою голову. Но в transformers это не так уж и сложно.

Отлично, теперь давайте обсудим гиперпараметры для обучения нашей модели. Авторы статьи про [BERT](#) советуют выбирать learning rate из следующего списка: $5*10^{-5}$, $3*10^{-5}$, $2*10^{-5}$. А количество эпох делать не слишком большим (2 или 4 будет достаточно). Мы же пробуем дообучать нашу сеть за одну эпоху, а в качестве learning rate давайте выберем $2*10^{-5}$. Вы можете попробовать другие параметры и убедиться, что это не слишком сильно повлияет на итоговое качество вашей модели. Теперь давайте обсуждать, что происходит в коде с циклом обучения. Цикл в нашем случае весьма номинален, поскольку он состоит из всего одной итерации. Вот наш код для дообучения модели. Что же мы делаем? В процессе обучения, сначала мы переводим нашу модель в train mode, затем распаковываем входные данные — распаковываем вектор с индексами токенов и метки классов. И также помещаем наши данные на [GPU](#) с помощью .to(device). Затем не забываем очистить градиенты с прошлого шага с помощью [zero_grad\(\)](#). Делаем forward pass, считаем loss, затем делаем backward pass, считаем градиенты, и дальше — стандартно — делаем [optimizer.step\(\)](#). Кроме того, в процессе обучения мы будем рисовать график и считать лосс. В конце каждой эпохи (в нашем случае — в конце единственной эпохи) посчитаем лосс на нашей обучающей выборке.

```
(LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
)
(intermediate): BertIntermediate(
  (dense): Linear(in_features=768, out_features=3072, bias=True)
)
(output): BertOutput(
  (dense): Linear(in_features=3072, out_features=768, bias=True)
  (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
)
(2): BertLayer(
```

```
In [28]: param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'gamma', 'beta']
optimizer_grouped_parameters = [
    {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.01},
    {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.0}
]
optimizer = AdamW(optimizer_grouped_parameters, lr=2e-5)
```

Теперь обсудим, что происходит в коде с циклом обучения. "Цикл" в данном случае весьма номинален: он состоит из всего одног

```
In [29]: from IPython.display import clear_output
# Будем сохранять loss во время обучения
```

```
In [29]: from IPython.display import clear_output
```

```
# Будем сохранять loss во время обучения
# и рисовать график в режиме реального времени
train_loss_set = []
train_loss = 0

# Обучение
# Переводим модель в training mode
model.train()

for step, batch in enumerate(train_dataloader):
    # добавляем батч для вычисления на GPU
    batch = tuple(t.to(device) for t in batch)
    # Распаковываем данные из dataloader
    b_input_ids, b_input_mask, b_labels = batch

    # если не сделать .zero_grad(), градиенты будут накапливаться
    optimizer.zero_grad()

    # Forward pass
    loss = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask, labels=b_labels)

    train_loss_set.append(loss[0].item())

    # Backward pass
    loss[0].backward()

    # Обновляем параметры и делаем шаг используя посчитанные градиенты
    optimizer.step()
```

```
batch = tuple(t.to(device) for t in batch)
# Распаковываем данные из dataloader
b_input_ids, b_input_mask, b_labels = batch

# если не сделать .zero_grad(), градиенты будут накапливаться
optimizer.zero_grad()

# Forward pass
loss = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask, labels=b_labels)

train_loss_set.append(loss[0].item())

# Backward pass
loss[0].backward()

# Обновляем параметры и делаем шаг используя посчитанные градиенты
optimizer.step()

# Обновляем loss
train_loss += loss[0].item()

# Рисуем график
clear_output(True)
plt.plot(train_loss_set)
plt.title("Training loss")
plt.xlabel("Batch")
plt.ylabel("Loss")
plt.show()

print("Loss на обучающей выборке: {:.5f}".format(train_loss / len(train_dataloader)))

# Валидация
# Переводим модель в evaluation mode
model.eval()
```



Из комментариев:

Вопрос:

AdamW что это?

Можем ли мы использовать обычный Adam из torch.optim.Adam, как было в предыдущей задаче про seq-to-seq?

Ответ (Романа Суворова):

если лосс падает, то можем :) Попробуйте - иногда бывает, что для какой-то архитектуры один оптимизатор проще настроить, чем другой. Обычно я пользуюсь Adam, но мне несколько раз сильно помогало заменить его на RMSprop. Кстати, [AdamW](#) входит в стандартный PyTorch >= 1.2.0

Ответ (Анастасии Яниной):

AdamW - Adam с применением Weight Decay.

Что такое **Weight Decay**? При каждом обновлении веса давайте кроме движения в сторону антиградиента еще и будем вычитать маленький кусочек веса, умноженный на некоторый гиперпараметр. Например, формула стохастического градиентного спуска с применением Weight Decay будет выглядеть так:

$$w = w - lr * \frac{\partial L}{\partial w} - lr * wd * w$$

Здесь lr - learning rate, wd - weight decay, гиперпараметр.

Если Вы внимательно посмотрите на эту формулу, то заметите, что в случае стохастического градиентного спуска Weight Decay эквивалентен применению L2-регуляризации к loss-функции:

$$L_{new} = L + \frac{wd}{2} \|w\|^2$$

$$\frac{\partial(L_{new})}{\partial w} = \frac{\partial L}{\partial w} + wd \cdot w$$

$$w = w - lr \cdot \frac{\partial L_{new}}{\partial w} = w - lr * \frac{\partial L}{\partial w} - lr * wd * w$$

Однако, L2-регуляризация (меняем loss-функцию) и Weight Decay (не меняем loss-функцию, меняем только способ обновления весов) работают одинаково только в простом случае стохастического градиентного спуска, в случае адаптивных оптимизаторов, например, Адама, эти два подхода различаются (причем эмпирически было показано, что часто Weight Decay работает лучше). Вся эта история породила некоторую путаницу в терминологии, и во многих библиотеках Адам реализовали именно с применением L2-регуляризации, ошибочно называя такой подход Weight Decay. Авторы статьи [Decoupled Weight Decay](#).

[Regularization](#) решили разграничить Adam+L2 и Adam+Weight Decay, назвав последнее AdamW.

Вопрос:

Самой интересной ячейкой, не проговоренной в лекции, показался код в ячейке с объявлением оптимизатора (0:35 в видео). Там выбираются определенные слои, для которых будет проходить обучение с разными затуханиями (весов?). Что там делается конкретно и зачем это нужно? По названиям переменных похоже на исключение банч-нормализации из логики затухания

Ответ (Сергея Устянцева):

тут другая идея. У модели есть параметры, которые мы и обучаем. Эти параметры бывают 'weights', 'bias', 'beta', 'gama'... В PyTorch есть возможность указывать оптимизатору как обучать разные группы параметров, можно даже разные параметры изменять с разной скоростью обучения.

Здесь мы разбиваем параметры на группы, для которых мы будем делать weight decay и для которых не будем. Про weight decay смотрите коментарий от Анастасии ниже.

Давайте посмотрим на график, который получился. Перезапускать этот код я не буду, при обучении на одной [GPU](#) у меня модель училась примерно минут 15-20. В принципе, на [CPU](#) она будет учиться часа 3-4, возможно, чуть дольше. По графику видно, что лосс, с первых батчей, достаточно сильно просел, потом сильно не менялся, но достаточно сильно флюктуировал. Отлично, теперь давайте посмотрим, какое качество мы можем получить на наших валидационных данных после одной эпохи. Вот наш код для валидации. Сначала переводим нашу модель в evaluation mode. Также, как в процессе обучения, распаковываем наш батч, загружаем данные на GPU, скармливаем данные сети (то есть делаем forward pass) и считаем лосс на валидационных данных (мониторим прогресс). Также — обратите внимание — мы используем `torch.no_grad`, при этом модель не будет считать и хранить градиенты, это ускорит

процесс предсказания меток для наших валидационных данных. И давайте посмотрим, что у нас получилось, какой score у нас получился на валидационных данных. Процент правильных предсказаний на валидационной выборке составил 97.87%. Кажется, вполне неплохо всего для одной эпохи дообучения и 15-20 минут обучения на одной GPU. Теперь давайте же оценим качество нашей модели на отложенной выборке. Может быть, наша модель [переобучилась](#)? Давайте проверим. Для этого нам нужно сделать точно такую же предобработку наших тестовых данных, как мы делали для обучающей и валидационной выборки. По сути, мы просто копируем два кусочка кода из предыдущих частей ноутбука. Делаем токенизацию тем же самым токенайзером, готовим переменные `input_ids`, добавляем паддинг, делаем [attention](#)-маски. Отлично, всё то же самое. И, затем, запускаем код, который посчитает нам [accuracy](#) на наших тестовых данных. Отлично, и что же получается? Процент правильных предсказаний на отложенной выборке составил 98% (если точнее, 98.12%). Очень похоже на то, что получилось на валидационной выборке. Чуть точнее — у нас оказалось всего около 1300 неправильных предсказаний из 68 тысяч. Для такого недолгого и простого процесса дообучения это очень и очень здорово.

```

plt.ylabel("Loss")
plt.show()

print("Loss на обучающей выборке: {:.5f}".format(train_loss / len(train_dataloader)))

# Валидация
# Переводим модель в evaluation mode
model.eval()

valid_preds, valid_labels = [], []

for batch in validation_dataloader:
    # добавляем батч для вычисления на GPU
    batch = tuple(t.to(device) for t in batch)

    # Распаковываем данные из dataloader
    b_input_ids, b_input_mask, b_labels = batch

    # При использовании .no_grad() модель не будет считать и хранить градиенты.
    # Это ускорит процесс предсказания меток для валидационных данных.
    with torch.no_grad():
        logits = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask)

    # Перемещаем logits и метки классов на CPU для дальнейшей работы
    logits = logits[0].detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()

    batch_preds = np.argmax(logits, axis=1)
    batch_labels = np.concatenate(label_ids)
    valid_preds.extend(batch_preds)
    valid_labels.extend(batch_labels)

print("Процент правильных предсказаний на валидационной выборке: {:.2f}%".format(
)
)

```

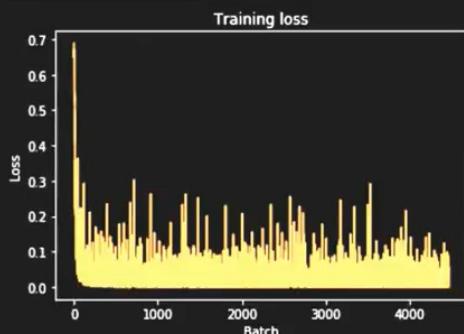


```

batch_preds = np.concatenate(batch_labels)
valid_preds.extend(batch_preds)
valid_labels.extend(batch_labels)

print("Процент правильных предсказаний на валидационной выборке: {:.2f}%".format(
    accuracy_score(valid_labels, valid_preds) * 100
))

```



Loss на обучающей выборке: 0.03665

```

-----
ValueError                                Traceback (most recent call last)
<ipython-input-29-33fe463caa45> in <module>
    74
    75 print("Процент правильных предсказаний на валидационной выборке: {:.2f}%%: {}".format(
--> 76     accuracy_score(valid_labels, valid_preds) * 100
    77 ))

```



```
In [32]: print("Процент правильных предсказаний на валидационной выборке: {:.2f}%".format(
    accuracy_score(valid_labels, valid_preds) * 100
))
Процент правильных предсказаний на валидационной выборке: 97.87%
```

Оценка качества на отложенной выборке

Качество на валидационной выборке оказалось очень хорошим. Не переобучилась ли наша модель?

Делаем точно такую же предобработку для тестовых данных, как и в начале ноутбука делали для обучающих данных:

```
In [33]: tokenized_texts = [tokenizer.tokenize(sent) for sent in test_sentences]
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]

input_ids = pad_sequences(
    input_ids,
    maxlen=100,
    dtype="long",
    truncating="post",
    padding="post"
)
```

Создаем attention маски и приводим данные в необходимый формат:

```
In [34]: attention_masks = [[float(i>0) for i in seq] for seq in input_ids]
prediction_inputs = torch.tensor(input_ids)
```



Создаем attention маски и приводим данные в необходимый формат:

```
In [34]: attention_masks = [[float(i>0) for i in seq] for seq in input_ids]

prediction_inputs = torch.tensor(input_ids)
prediction_masks = torch.tensor(attention_masks)
prediction_labels = torch.tensor(test_gt)

prediction_data = TensorDataset(
    prediction_inputs,
    prediction_masks,
    prediction_labels
)

prediction_dataloader = DataLoader(
    prediction_data,
    sampler=SequentialSampler(prediction_data),
    batch_size=32
)
```

```
In [35]: model.eval()
test_preds, test_labels = [], []

for batch in prediction_dataloader:
    # добавляем батч для вычисления на GPU
    batch = tuple(t.to(device) for t in batch)

    # Распаковываем данные из dataloader
    b_input_ids, b_input_mask, b_labels = batch

    # При использовании .no_grad() модель не будет считать и хранить градиенты.
```



```
b_input_ids, b_input_mask, b_labels = batch

# При использовании .no_grad() модель не будет считать и хранить градиенты.
# Это ускорит процесс предсказания меток для тестовых данных.
with torch.no_grad():
    logits = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask)

# Перемещаем logits и метки классов на CPU для дальнейшей работы
logits = logits[0].detach().cpu().numpy()
label_ids = b_labels.to('cpu').numpy()

# Сохраняем предсказанные классы и ground truth
batch_preds = np.argmax(logits, axis=1)
batch_labels = np.concatenate(label_ids)
test_preds.extend(batch_preds)
test_labels.extend(batch_labels)

In [42]: acc_score = accuracy_score(test_labels, test_preds)
print('Процент правильных предсказаний на отложенной выборке составил: {:.2f}%'.format(
    acc_score*100
))
Процент правильных предсказаний на отложенной выборке составил: 98.12%
```



```
In [43]: print('Неправильных предсказаний: {0}/{1}'.format(
    sum(test_labels != test_preds),
    len(test_labels)
))
Неправильных предсказаний: 1282/68051
```



Итак, мы показали, что предобученный BERT может быстро (всего за одну эпоху) давать хорошее качество при решении задачи анализа эмоциональной окраски текстов. Обратите внимание, что мы не тюнили параметры и использовали сравнительно небольшой

Из комментариев:

Вопрос:

Конкретно в данном примере - зачем мы разделяли набор данных на валидационную и тестовую выборку. Обычно смысл валидационной выборки это подбор гиперпараметров, или поиск наилучшей эпохи, если accuracy в процессе обучения скакает или модель переобучается в процессе переобучения. В данном коде мы нигде не делали оценку в процессе обучения качества на валидационной выборке. У нас одна эпоха и гиперпараметры мы тоже не меняли...

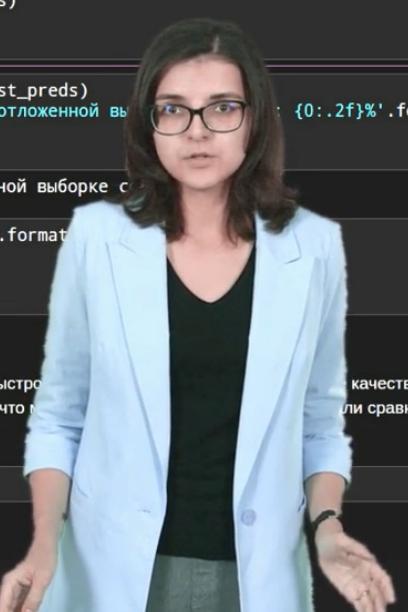
Правильно ли я понимаю что в данном примере мы могли отказаться от разбиения на обучающую, валидационную и тестовую? А в случае двух и более эпох, если бы мы при этом пытались выбрать наилучшую модель из всех эпох, тогда уже валидационная была бы нужна.

Ответ (Романа Суворова):

на практике лучше всегда иметь отложенную выборку. Исключениями могут быть ситуации, в которых Вы точно уверены, что переобучения не произойдет (например, когда делается достаточно малое количество итераций и ранее, делая отложенную выборку, Вы уже убедились, что ничего страшного не случится, если её убрать)

Мы показали, что предобученный [BERT](#) может быстро (всего за одну эпоху) давать хорошее качество при решении задачи анализа эмоциональной окраски текстов. Кроме того, обратите внимание, что мы не тюнили параметры и использовали сравнительно маленький размеченный [корпус](#), чтобы получить [accuracy](#) больше 98%. Тем не менее, если не делать дообучение под конкретную задачу вовсе, то получить хорошее качество вряд ли выйдет.

Кроме того, на этом семинаре мы познакомились с библиотекой [pytorch-transformers](#), которая позволяет использовать готовые обёртки над моделями, специально созданными для решения той или иной задачи. Использовать BERT при решении повседневных [NLP](#) задач совсем не трудно. Не нужно даже вручную скачивать веса модели, искать их где-то в интернете — библиотека абсолютно всё сделает за вас. Отбросив необходимость небольшой предобработки текстов, сложность применения предобученного BERT с использованием библиотеки pytorch-transformers оказывается не сильно больше, чем — ну, например, импортировать лог-регрессию из sk-learn, и примените её, а качество итоговое получается гораздо выше. Вы можете использовать предобученный BERT, [GPT-2](#) или какие-то другие сети для решения других задач — не только классификации, но и чего-то более сложного, например, для решения задачи вопросно-ответного поиска, или, может быть, [машинного перевода](#) или выделения именованных сущностей. Единственное, что вам нужно будет сделать — это импортировать другую модель из pytorch-transformers и подготовить ваши данные для обучения в чуть-чуть другом формате. Успеха в дальнейшей работе с pytorch-transformers!



SAMSUNG
Research
Russia

```
logits = logits[0].detach().cpu().numpy()
label_ids = b_labels.to('cpu').numpy()

# Сохраняем предсказанные классы и ground truth
batch_preds = np.argmax(logits, axis=1)
batch_labels = np.concatenate(label_ids)
test_preds.extend(batch_preds)
test_labels.extend(batch_labels)

In [42]: acc_score = accuracy_score(test_labels, test_preds)
print('Процент правильных предсказаний на отложенной выборке: {:.2f}%'.format(
    acc_score*100
))
Процент правильных предсказаний на отложенной выборке с

In [43]: print('Неправильных предсказаний: {0}/{1}'.format(
    sum(test_labels != test_preds),
    len(test_labels)
))
Неправильных предсказаний: 1282/68051

Итак, мы показали, что предобученный БЕРТ может быстро определить эмоциональной окраски текстов. Обратите внимание, что для этого требуется очень большой корпус, чтобы получить accuracy больше 98%.
```

In []:

Из комментариев:

Вопрос:

Вот бы пример кода с заморозкой слоев.

Ответ (Романа Суворова):

конкретно по PyTorch-Transformers Анастасия ответит лучше меня, но в общем в PyTorch есть несколько способов заморозить слои (по сути, защитить их от изменения оптимизатором).

Допустим у нас есть следующая модель:

```
class OurModel(nn.Module):
    def __init__(self):
```

```
super().__init__()  
self.pretrained_model = load_super_pretrained_model()  
self.output_layer = nn.Linear(...)
```

Способ 1 (предпочтительный). Не передавать в оптимизатор те параметры, которые надо заморозить

Обычно мы создаём оптимизатор для такой модели как

```
model = OurModel()  
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Если мы не хотим учить часть нашей модели (например, предобученные слои self.pretrained_model), тогда можно сделать так:

```
model = OurModel()  
optimizer = torch.optim.Adam(model.output_layer.parameters(), lr=1e-3)
```

Это подход "по умолчанию".

Способ 2. Грязный хак: спрятать параметры от самой модели

Когда мы вызываем model.parameters(), модель делает проход по своим полям и либо собирает экземпляры класса nn.Parameter, либо рекурсивно вызывает .parameters() (если поле имеет тип nn.Module). Если модель видит поле какого-то другого типа, она его игнорирует.

Поэтому можно обернуть вложенную модель, которую не надо учить, например, в список.

```
class OurModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.pretrained_model = [load_super_pretrained_model()]  
        self.output_layer = nn.Linear(...)
```

В этом случае в оптимизатор не попадут параметры self.pretrained_model

```
model = OurModel()  
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Этим можно пользоваться осторожно, но надо помнить, что в разработке явное лучше, чем скрытое.

Способ 3. Гибкий вариант: занулять градиенты параметров руками

После того, как мы вызвали loss.backward(), у всех объектов-экземпляров класса nn.Parameter появится атрибут .grad, это тензор, имеющий ту же форму, что и сам тензор параметров, и содержащий значения производной функции потерь по этой переменной.

Тогда можно руками пройтись по нужным параметрам и занулить у них grad. При этом надо именно обнулить значения тензора, а не перезаписать его нулём (обратите внимание на квадратные скобки):

```
loss.backward()  
for param in model.pretrained_model.parameters():
```

```
param.grad[:] = 0 # или param.grad.zero_()
```

Такой вариант наиболее удобен, когда надо на разных шагах оптимизации нужно оптимизировать разные переменные. Это достаточно экзотическая ситуация и в рамках этого курса это не пригодится, лучше пользоваться первым способом.

Из комментариев к практическому заданию (<https://stepik.org/lesson/262253/step/10?unit=243136>) :

Вопрос:

1. Первая проблема в IMDB предложения получаются больше 512 Токенов. Приходится обрезать
2. На видеокарте не помещается, batch_size приходится уменьшать
3. Во время обучения loss идет в разнос 😊

Ответ (Романа Суворова):

> больше 512 Токенов. Приходится обрезать

В целом это нормальная ситуация. Более того, BERT обучался не на предложениях, а на произвольных фрагментах текста (которые могли включать несколько предложений). В качестве решения можно предложить нарезать несколько обучающих примеров из одного предложения, так чтобы не терять "хвосты" насовсем - это можно делать, например, скользящим окном.

> На видеокарте не помещается, batch_size приходится уменьшать

Что ж поделаешь, бывает :) С одной стороны, это может быть и неплохо. В некоторых задачах лучше увеличить длину последовательности, подаваемой за раз и уменьшить batch_size (эффективное количество токенов в батче может остаться прежним), а в других задачах - наоборот, лучше увеличить batch_size и пожертвовать длиной текста - тут надо мерять качество на отложенной выборке.

> 3. Во время обучения loss идет в разнос :)

В целом график функции потерь выглядит нормально. Попробуйте добавить сглаживание - тренд должен прослеживаться лучше.

Доп. комментарий (студента):

и еще gradient accumulation для небольших batch_size используют

Доп ответ (Романа Суворова):

да, хорошая идея. Большой батч сглаживает градиенты и помогает сделать процесс обучения стабильнее в ряде случаев. Если большой батч не влезает - можно сделать виртуальный батч (он же gradient accumulation), для этого надо прокрутить несколько батчей, на каждом вызывая loss.backward(retain_graph=True), и только после прокручивания достаточного количества батчей вызвать optimizer.step() - тогда градиенты от нескольких батчей просуммируются.

И ещё важно не забывать менять learning rate при изменении размера батча.

От себя:

<https://huggingface.co/models> - множество различных моделей, датасетов и т.п.

6.3 Семинар: BERT для вопросно-ответного поиска

Для данного семинара Вам потребуется ноутбук [task10_bert_squad.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 `dlnlputils`

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! На этом семинаре мы продолжим работать с [BERT](#) и применим его для решения более сложной задачи — а именно, мы научим BERT отвечать на вопросы ([question-answering](#)). Стандартный датасет для тренировки QA моделей (или question-answering моделей) называется SQuAD^[2] и расшифровывается как Stanford Question Answering [Dataset](#). Он включает в себя вопросы и ответы на них по достаточно разнообразной тематике. Вопросы формировались на основе статей или отрывков из статьи из Википедии. Ответ на каждый вопрос представляет собой сегмент текста или промежуток из соответствующего отрывка. В новой версии датасета (в 2.0 версии) даже возможны вопросы без ответа. Имеется в виду, что есть вопросы, для ответа на которые недостаточно информации в предложенном фрагменте текста. Таким образом, алгоритм составления этого датасета был следующим. Ассесор читал фрагмент текста из Википедии (как правило, всего несколько абзацев), затем формулировал вопрос по прочитанному и фиксировал правильный ответ. На большинство вопросов присутствует

несколько вариантов ответа. Но, как правило, это всего лишь переформулировки одного и того же, по смыслу, варианта ответа. Например, на вопрос "когда началась эпоха Возрождения в Италии" можно ответить "14 век", "в 14 веке", "в начале 14 века" или как-то ещё, использовав слова "14" и "век". Учитывая специфику нашего датасета, на вход нейросети мы будем подавать не только сам вопрос, но и соответствующий фрагмент текста, параграф текста. А в качестве выхода нейросети будем ожидать две позиции в тексте: начало ответа и конец ответа. Итак, начнём работать с кодом ноутбука. Для начала, скачайте датасет (вот отсюда). Для выполнения семинара вам понадобится два файла. Один — с обучающей выборкой (train.json), и с выборкой, на которой мы будем тестировать наш алгоритм (он называется dev_version2.json). Так же как и в предыдущем семинаре, мы будем использовать библиотеку [pytorch-transformers](#). Мы, опять же, не будем писать самостоятельно сеть для решения задачи понимания текста и ответа на вопросы по нему, а научимся работать со скриптами из pytorch-transformers. Итак, давайте сначала склонируем (clone) репозитории и положим путь до папки "examples" в переменную path_to_examples. У меня репозиторий уже склонирован, поэтому давайте выполним вот эту ячейку кода. И, дальше, импортируем функции модели, которые могут нам понадобиться. Также объявим переменную device, здесь есть два варианта — [cuda](#) или [cpu](#) — всё, как обычно. Дальше, так же как и в прошлом семинаре, загружаем токенизатор для BERT и готовую модификацию BERT для [вопросно-ответных систем](#) под названием "bert for question answering" из библиотеки pytorch-transformers. Выполняем вот этот кусочек кода. Загрузка модели может занять какое-то время. Отлично, модель скачалась. Теперь у вас есть два варианта развития событий — простой и чуть-чуть более сложный. Вы можете попробовать дообучить модель на датасете SQuAD^[2] самостоятельно или же загрузить уже предобученные веса сети и перейти сразу к блоку "оценка качества работы модели".^[1] Второй вариант имеет смысл, когда у вас нет под рукой свободной видеокарты. Подгрузить веса нейросети можно с помощью функции "load_state_dict", как вы уже помните. И при подгрузке весов в случае отсутствия [GPU](#) не забудьте указать параметр "map_location" (вот как в этой строке кода). Но давайте всё-таки отклонимся от простого пути и пойдём дообучать модель.

[1] <https://stepik.org/lesson/268748/step/7?unit=249768>

[2] <https://rajpurkar.github.io/SQuAD-explorer/>

BERT для вопросно-ответных систем

Скачайте датасет (SQuAD) [отсюда](#). Для выполнения семинара Вам понадобятся файлы `train-v2.0.json` и `dev-v2.0.json`.

Склонируйте репозиторий <https://github.com/huggingface/transformers> и положите путь до папки `examples` в переменную `PATH_TO_EXAMPLES`.

In [7]: `# !git clone https://github.com/huggingface/transformers`

```
Cloning into 'transformers'...
remote: Enumerating objects: 71, done.
remote: Counting objects: 100% (71/71), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 12251 (delta 38), reused 40 (delta 25), pack-reused 12180
Receiving objects: 100% (12251/12251), 6.44 MiB | 2.68 MiB/s, done.
Resolving deltas: 100% (8970/8970), done.
Checking connectivity... done.
```

In [27]: `PATH_TO_EXAMPLES = 'transformers/examples/'`
`import sys`
`sys.path.append(PATH_TO_EXAMPLES)`

Теперь импортируем функции и модели, которые могут нам понадобиться:

In [32]: `import torch`
`import tqdm`
`import json`

`from utils_squad import (read_squad_examples, convert_examples_to_features,`



In [34]: `PATH_TO_EXAMPLES = 'transformers/examples/'`
`import sys`
`sys.path.append(PATH_TO_EXAMPLES)`

Теперь импортируем функции и модели, которые могут нам понадобиться:

In [35]: `import torch`
`import tqdm`
`import json`

`from utils_squad import (read_squad_examples, convert_examples_to_features,`
 `RawResult, write_predictions,`
 `RawResultExtended, write_predictions_extended)`

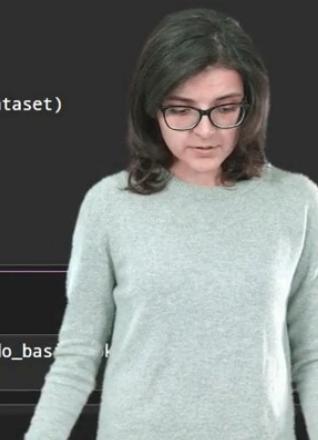
`from run_squad import train, load_and_cache_examples`

`from torch.utils.data import (DataLoader, RandomSampler, SequentialSampler, TensorDataset)`

`from transformers import (WEIGHTS_NAME, BertConfig, XLNetConfig, XLMConfig,`
 `BertForQuestionAnswering, BertTokenizer)`

`from utils_squad_evaluate import EVAL_OPTS, main as evaluate_on_squad`

`device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`



Загружаем токенизатор для БЕРТА и BertForQuestionAnswering из pytorch-transformers:

In [29]: `tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True, do_basic_tokenize=False)`
`model = BertForQuestionAnswering.from_pretrained('bert-base-uncased')`

```
какую текстенную, write_predictions_extendaed)

from run_squad import train, load_and_cache_examples

from torch.utils.data import DataLoader, RandomSampler, SequentialSampler, TensorDataset

from transformers import (WEIGHTS_NAME, BertConfig, XLNetConfig, XLMConfig,
                          BertForQuestionAnswering, BertTokenizer)

from utils_squad_evaluate import EVAL_OPTS, main as evaluate_on_squad

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Загружаем токенизатор для БЕРТа и BertForQuestionAnswering из pytorch-transformers:

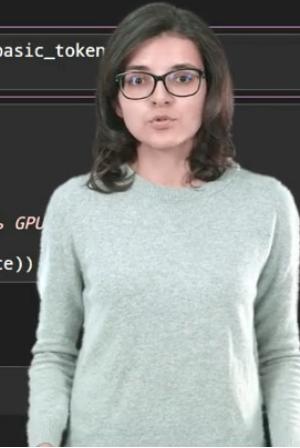
```
In [36]: tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True, do_basic_tokenize=True)
model = BertForQuestionAnswering.from_pretrained('bert-base-uncased')
```

```
In [30]: # если Вы не хотите запускать файн-тюнинг, пропустите блок "Дообучение",
# подгрузите веса уже дообученной модели и переходите к блоку "Оценка качества"

if torch.cuda.is_available():
    model.cuda()
    model.load_state_dict(torch.load('models/bert_squad_5epochs.pt')) # если у вас есть GPU
else:
    model.load_state_dict(torch.load('models/bert_squad_5epochs.pt', map_location=device))
```

Дообучение

```
In [62]: # !pip install dataclasses
from dataclasses import dataclass
```



Из комментариев:

Вопрос:

А какие могут быть ground truth ответы на вопросы, для которых нет ответов?

Ответ (Романа Суворова):

чуть позже Анастасия сможет ответить точнее, но я тоже попытаюсь :) SQuAD - задача поиска в заданном тексте фрагментов, которые могут быть ответами на заданный же вопрос. При этом алгоритм, выполняющий поиск ответов, не имеет возможности оценить достоверность этого ответа (для этого нужно привлекать большой контекст, базу знаний и т.п.). Авторы датасета при разметке тоже руководствовались своей базой знаний, которая есть у них в голове (типа common knowledge). Частично роль такой базы знаний выполняет предобученная нейросеть (например, BERT), но как и в любой задаче машинного обучения с учителем, модель будет настолько хорошая, насколько хороший датасет.

Доп. ответ (Романа Суворова):

вроде бы в этом датасете есть несколько обучающих примеров, подразумевающих пустой ответ (то есть ответ на вопрос в тексте не содержится), но прямо сейчас не помню.

Доп. ответ (Анастасии Яниной):

да, все верно! Имеется в виду, что ответ на вопрос не содержится в тексте

Итак, рассмотрим процесс дообучения модели. Сразу отмечу, что аналогичный результат можно получить, просто запустив скрипт `run_squad.py`^[1] из папки "examples", но мы попробуем разобрать этот скрипт чуть более подробно и осознать, за что отвечают все входные параметры из этого скрипта. А этих входных параметров достаточно много. В класс

"train_opts" я выписала параметры, на которые стоит обратить внимание перед запуском модели. Давайте пройдёмся по ним. Обратите внимание, что с помощью декоратора [DataClass](#) можно сделать код вашего класса с перечислением параметров чуть более красивым. Но теперь давайте всё-таки посмотрим на все наши параметры. Первый параметр называется "train_file" — сюда мы кладём путь до обучающей выборки, в переменную "predict_file", соответственно, кладём путь до датасета, на котором мы будем тестироваться и оценивать качество работы модели. Ещё из интересных параметров — у нас есть параметр "model_type", здесь нужно указать, с какой моделью мы будем работать (в нашем случае, это [BERT](#), но также можно было указать, например, [XLNet](#) на DistilBERT). А в параметр "model_name_or_path" мы кладём либо путь до предобученной модели, если таковая у вас есть, либо же название модели из переменной "all_models". Эту переменную "all_models" я определила после этого класса — вот она, давайте посмотрим, что в ней содержится. В этой переменной есть все варианты моделей, с которыми мы можем работать с помощью данного кода (с помощью кодов библиотеки [pytorch-transformers](#)). Если говорить про BERT, то у нас есть достаточно вариантов — есть варианты моделей BERT Base и BERT Large, можно использовать "uncased" или "cased" варианты, можно брать multilingual-модели, либо же какие-то специально заточенные под конкретный язык (например, BERT для китайского языка). Мы же будем использовать самый простой вариант "bert-base-uncased". Давайте продолжим наше путешествие по параметрам, которые мы будем использовать для обучения модели. Ещё из интересных параметров стоит отметить "version_2_with_negative" — этот параметр отвечает за то, будем ли мы при обучении учитывать вопросы, на которые ответа в датасете нет. Это — те самые вопросы, которые я упоминала в начале семинара. Также мы можем указать максимальную длину параграфа, который мы подаём в нашу нейросеть (в нашем случае это будет 384, имеется виду 384 токена, а не 384 символа). Также можно указать страйд — параметр "doc_stride" отвечает за то, сколько страйда мы будем использовать при делении длинного документа на чанки (chunks), на небольшие кусочки. Кроме того, можно указать максимальное количество токенов в вопросе — за это отвечает параметр "max_query_length" (в нашем случае, это 128) и, также, можно указать максимальную длину ответа — это параметр "max_answer_length". Кроме того, давайте посмотрим на параметры, которые нам нужны непосредственно для обучения нейросети. А именно, мы указываем "learning_rate", указываем "weight_decay" (в нашем случае, мы его не используем, он равен нулю). Можем указать эпсилон для оптимизатора [Adam](#), также мы, конечно, указываем, сколько эпох мы будем тренировать нашу модель. В нашем случае это всего 5 эпох. Отмечу, что на тренировку 5 эпох для нашей модели у меня ушло около 7 часов на одной [GPU](#). Кроме того, мы указываем, сколько у нас будет warmup-шагов (в нашем случае мы вовсе не будем использовать warm-up), ещё из интересных параметров стоит отметить вот эти две вещи — "logging_steps" и "save_steps". Мы говорим, через сколько шагов мы хотим логировать (log) прогресс нашей модели и сохранять чекпоинты (checkpoints). В нашем случае — будем сохранять чекпоинты каждые 5 000 шагов. Также, если у нас выставлен параметр "evaluate_during_training", через каждые 5 000 шагов наша модель будет оценивать качество своей работы на датасете из

`predict_file` (на датасете, который лежит вот по этому пути). И также наша модель будет выводить нам список метрик, которые удалось достичь на текущий момент тренировки. Кроме того, мы можем указать — печатать ли `warnings`, можем указать, что мы не хотим использовать [CUDA](#) (даже если у нас доступна GPU). Кроме того, можем задать размеры батча для тренировки и для `evaluation`.

[1] https://github.com/huggingface/transformers/blob/master/examples/run_squad.py



SAMSUNG Research Russia

```
In [62]: # !pip install dataclasses
from dataclasses import dataclass

@dataclass
class TRAIN_OPTS:
    train_file : str = 'train-v2.0.json'      # SQuAD json-файл для обучения
    predict_file : str = 'dev-v2.0.json'        # SQuAD json-файл для тестирования
    model_type : str = 'bert'                  # тип модели (может быть 'bert', 'xlnet', 'xlm', 'distilbert')
    model_name_or_path : str = 'bert-base-uncased' # путь до предобученной модели или название модели из ALL_MODELS
    output_dir : str = '/tmp' # путь до директории, где будут храниться чекпоинты и предсказания модели
    device : str = 'cuda' # cuda или cpu
    n_gpu : int = 1 # количество gpus для обучения
    cache_dir : str = '' # где хранить предобученные модели, загруженные с s3

    # Если true, то в датасет будут включены вопросы, на которые нет ответов.
    version_2_with_negative : bool = True
    # Если (null_score - best_non_null) больше, чем порог, предсказывать null.
    null_score_diff_threshold : float = 0.0
    # Максимальная длина входной последовательности после WordPiece токенизации. Sequences
    # Последовательности длиннее будут укорочены, для более коротких последовательностей будет и
    max_seq_length : int = 384
    # Сколько stride использовать при делении длинного документа на чанки
    doc_stride : int = 128
    # Максимальное количество токенов в вопросе. Более длинные вопросы будут укорочены до
    max_query_length : int = 128 #

    do_train : bool = True
    do_eval : bool = True

    # Запускать ли evaluation на каждом logging_step
    evaluate_during_training : bool = True
    # Должно быть True, если Вы используете uncased модели
```



SAMSUNG Research Russia

```
do_eval : bool = True

# Запускать ли evaluation на каждом logging_step
evaluate_during_training : bool = True
# Должно быть True, если Вы используете uncased модели
do_lower_case : bool = True #

per_gpu_train_batch_size : int = 8 # размер батча для обучения
per_gpu_eval_batch_size : int = 8 # размер батча для eval
learning_rate : float = 5e-5 # learning rate
gradient_accumulation_steps : int = 1 # количество шагов, которые нужно сделать перед backward/update pass
weight_decay : float = 0.0 # weight decay
adam_epsilon : float = 1e-8 # эпсилон для Adam
max_grad_norm : float = 1.0 # максимальная норма градиента
num_train_epochs : float = 5.0 # количество эпох на обучение
max_steps : int = -1 # общее количество шагов на обучение (override num_train_epochs)
warmup_steps : int = 0 # шаги разогрева
n_best_size : int = 5 # количество ответов, которые надо сгенерировать для записи в nbest_predictions.json
max_answer_length : int = 30 # максимально возможная длина ответа
verbose_logging : bool = True # печатать или нет warnings, относящиеся к обработке данных
logging_steps : int = 5000 # логировать каждые X шагов
save_steps : int = 5000 # сохранять чекпоинт каждые X шагов

# Evaluate all checkpoints starting with the same prefix as model_name ending and ending
eval_all_checkpoints : bool = True
no_cuda : bool = False # не использовать CUDA
overwrite_output_dir : bool = True # переписывать ли содержимое директории с выходными файлами
overwrite_cache : bool = True # переписывать ли закешированные данные для обучения и eval
seed : int = 42 # random seed
local_rank : int = -1 # local rank для распределенного обучения на GPU
fp16 : bool = False # использовать ли 16-bit (mixed) precision (через NVIDIA apex) вместо float32
# Apex AMP optimization level: ['00', '01', '02', and '03'].
# Подробнее тут: https://nvidia.github.io/apex/amp.html
fp16_opt_level : str = '01'
```

```
# Подробнее тут: https://nvidia.github.io/apex/amp.html
fp16_opt_level : str = '01'

In [38]: ALL_MODELS = sum((tuple(conf.pretrained_config_archive_map.keys()) \
                         for conf in (BertConfig, XLNetConfig, XLMConfig)), ())
ALL_MODELS

Out[38]: ('bert-base-uncased',
 'bert-large-uncased',
 'bert-base-cased',
 'bert-large-cased',
 'bert-base-multilingual-uncased',
 'bert-base-multilingual-cased',
 'bert-base-chinese',
 'bert-base-german-cased',
 'bert-large-uncased-whole-word-masking',
 'bert-large-cased-whole-word-masking',
 'bert-large-uncased-whole-word-masking-finetuned-squad',
 'bert-large-cased-whole-word-masking-finetuned-squad',
 'bert-base-cased-finetuned-mrpc',
 'bert-base-german-dbmdz-cased',
 'bert-base-german-dbmdz-uncased',
 'xlnet-base-cased',
 'xlnet-large-cased',
 'xlm-mlm-en-2048',
 'xlm-mlm-ende-1024',
 'xlm-mlm-enfr-1024',
 'xlm-mlm-enro-1024',
 'xlm-mlm-tlm-xnli15-1024',
 'xlm-mlm-xnli15-1024',
 'xlm-clm-enfr-1024',
 'xlm-clm-ende-1024',
 'xlm-mlm-17-1280',
 'xlm-mlm-100-1280')
```



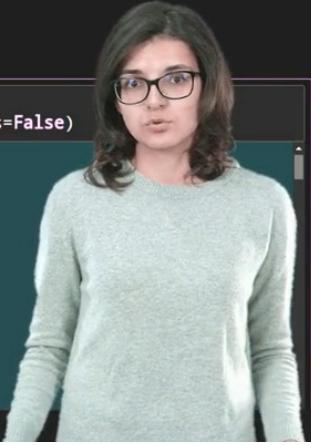
Итак, мы рассмотрели основные параметры, которые нам нужны для тренировки модели, теперь можем подгрузить наши данные и запустить тренировку. Давайте загрузим наш датасет. Обратите внимание на параметр "evaluate" в этой строчке кода. Если здесь будет написано "evaluate = True", то никакого обучения происходит не будет. Запустится evaluation на наших данных, а веса модели меняться никак не будут. После того, как мы загрузили наш датасет (это достаточно долгий процесс...), мы можем запустить обучение нашей модели (с помощью вот этой строчки кода). Полное обучение всех пяти эпох, как я уже говорила, заняло около 7 часов, поэтому запускать эту ячейку мы не будем. Отмету лишь, что через каждые 5 000 шагов, как и обсуждалось ранее, у нас происходит evaluation модели. В нашем случае, это примерно через каждые 30 процентов работы алгоритма. То есть, на 5 000 итерации из 16500 (примерно) мы запускаем evaluation. Evaluation тоже занимает некоторое время, но зато в конце мы можем посмотреть на метрики, которые у нас получились. Например, после первых 5 000 итерации мы видим, что наша модель дала абсолютно правильный ответ в 57% случаев, f-мера составила 69.16 (примерно). В принципе, довольно неплохо. Но, тем не менее, давайте "дотренируем" нашу модель дальше (точнее — посмотрим, как она тренировалась). И после того как наша модель обучилась мы можем сохранить веса нашей модели (например, в файл "bert_squad[1]_final_5epochs"). А потом, если захотим, мы сможем подгрузить веса нашей модели с помощью вот этой строчки кода, которую мы уже видели чуть ранее.

[1] <https://rajpurkar.github.io/SQuAD-explorer/>

```
'bert-large-uncased-whole-word-masking-finetuned-squad',
'bert-large-cased-whole-word-masking-finetuned-squad',
'bert-base-cased-finetuned-mrpc',
'bert-base-german-dbmdz-cased',
'bert-base-german-dbmdz-uncased',
'xlnet-base-cased',
'xlnet-large-cased',
'xlm-mlm-en-2048',
'xlm-mlm-ende-1024',
'xlm-mlm-enfr-1024',
'xlm-mlm-enro-1024',
'xlm-mlm-tlm-xnli15-1024',
'xlm-mlm-xnli15-1024',
'xlm-clm-enfr-1024',
'xlm-clm-ende-1024',
'xlm-mlm-17-1280',
'xlm-mlm-100-1280')
```

In [63]: `args = TRAIN_OPTS()
train_dataset = load_and_cache_examples(args, tokenizer, evaluate=False, output_examples=False)`

```
0%|       | 167/130319 [00:01<23:16, 93.17it/s]  
0%|       | 177/130319 [00:01<25:43, 84.30it/s]  
0%|       | 186/130319 [00:02<31:14, 69.42it/s]  
0%|       | 194/130319 [00:02<30:20, 71.48it/s]  
0%|       | 203/130319 [00:02<28:31, 76.01it/s]  
0%|       | 216/130319 [00:02<25:09, 86.20it/s]  
0%|       | 233/130319 [00:02<21:36, 100.35it/s]
```



```
100%|██████████| 130256/130319 [17:41<00:00, 124.90it/s]  
100%|██████████| 130269/130319 [17:42<00:00, 123.51it/s]  
100%|██████████| 130282/130319 [17:42<00:00, 124.62it/s]  
100%|██████████| 130295/130319 [17:42<00:00, 123.96it/s]  
100%|██████████| 130319/130319 [17:42<00:00, 122.66it/s]
```

In [64]: `train(args, train_dataset, model, tokenizer)`

Evaluating: 100%|██████████| 1529/1529 [02:08<00:00, 11.89it/s]

```
{
  "exact": 66.51225469552767,
  "f1": 69.15696383081102,
  "total": 11873,
  "HasAns_exact": 57.45614035087719,
  "HasAns_f1": 62.75314297625138,
  "HasAns_total": 5928,
  "NoAns_exact": 75.54247266610597,
  "NoAns_f1": 75.54247266610597,
  "NoAns_total": 5945,
  "best_exact": 66.82388612819001,
  "best_exact_thresh": -0.3868217468261719,
  "best_f1": 69.25620960293222,
  "best_f1_thresh": -0.3868217468261719
}
```



Сохраняем веса дообученной модели на диск, чтобы в следующий раз не обучать модель заново.

```
Iteration: 100%|██████████| 16490/16493 [1:28:16<00:00, 3.64it/s]

Iteration: 100%|██████████| 16491/16493 [1:28:16<00:00, 3.63it/s]

Iteration: 100%|██████████| 16492/16493 [1:28:17<00:00, 3.63it/s]

Iteration: 100%|██████████| 16493/16493 [1:28:17<00:00, 3.11it/s]

Epoch: 100%|██████████| 1/1 [1:28:17<00:00, 5297.46s/it]
```

Out[64]: (16493, 1.205482169009105)

Сохраняем веса дообученной модели на диск, чтобы в следующий раз не обучать модель заново.

In [66]: `torch.save(model.state_dict(), 'models/bert_squad_final_5epoch.pt')`

Подгрузить веса модели можно так:

In [90]: `model.load_state_dict(torch.load('models/bert_squad_5epochs.pt'))`

Out[90]: <All keys matched successfully>

Оценка качества работы модели

In [14]: `PATH_TO_DEV_SQUAD = 'dev-v2.0.json'`
`PATH_TO_SMALL_DEV_SQUAD = 'small_dev-v2.0.json'`



Поздравляю! Наша модель обучена. И хотя, в процессе обучения, после каждой из 5 000 итераций, модель производила evaluation на dev-датасете и выводила нам метрики, давайте всё-таки сделаем evaluation ещё раз. И — самое главное — посмотрим не на метрики, а вручную — на вопросы и ответы, сгенерированные [BERT](#). Чтобы не просматривать слишком много вопросов вручную, давайте отделим маленький кусочек, состоящий из всего 208 вопросов, и оценим качество работы модели на нём. Также наши метрики будут считаться быстрее на этом маленьком кусочке, чем на всём dev-датасете. Отделим наш маленький датасет с помощью следующего кода, считываем наш dev.json файл, отделяем маленький кусочек (208 вопросов) и записываем его в файл small_dev.json. Теперь объявим несколько констант, которые нам будут нужны для evaluation. Определяем максимальную длину параграфа (384 — так же как и для обучения), также определяем максимальную длину вопроса, максимальную длину ответа. А теперь, с помощью функции "read_squad[1]_examples()", мы загрузим наши 208 вопросов и будем дальше с ними работать (загружаем наши вопросы с помощью вот этого кода). Здесь мы выставляем "is_training = False", потому что мы собираемся только оценить качество работы нашей модели и ничего обучать мы здесь не будем. И дальше, с помощью функции "convert_examples_to_features()" превращаем загруженные данные в фичи. Здесь мы ставим параметр "is_training" в значении False, так же как и для функции read_examples, и используем тот же самый токенайзер, что и при обучении (вот этот токенайзер). Также передаём наши примеры, загруженные с помощью "read_squad[1]_examples()". После этого вытаскиваем из наших фичей "input_ids", "input_mask",

"segment_ids", "cls_index" и "p_mask". Что же значит все эти переменные? Формат, на самом деле, очень похож на тот, что мы обсуждали в прошлом семинаре, когда работали с BERT для классификации предложений. Давайте посмотрим чуть более подробно. "input_ids" — это просто последовательность чисел, отождествляющих каждый токен с его номером в словаре (также, как и в прошлом семинаре). "input_mask" — это последовательность из нулей и единиц, где единицы обозначают токены предложения, а нули — паддинг. Похоже на "attention_mask" из ноутбука про классификацию с помощью BERT на прошлом семинаре. Переменная "p_mask" означает следующее — здесь мы будем маскировать единицами токены, которых не может быть в ответе, а нулями — все токены, которые могут встретиться в нашем ответе. Ну, и "cls_index" — это индекс нашего классификационного токена. Дальше мы передаём все эти переменные в [тензорDataset](#) (вот здесь) и, дальше, создаём из него [DataLoader](#) (вот таким образом). Здесь мы передаём sampler и определяем размер батча. В нашем случае, размер батча будет равен 8.

[1] <https://rajpurkar.github.io/SQuAD-explorer/>

```
In [66]: torch.save(model.state_dict(), 'models/bert_squad_final_5epochs.pt')
```

Подгрузить веса модели можно так:

```
In [90]: model.load_state_dict(torch.load('models/bert_squad_5epochs.pt'))
```

```
Out[90]: <All keys matched successfully>
```

Оценка качества работы модели

```
In [39]: PATH_TO_DEV_SQUAD = 'dev-v2.0.json'
PATH_TO_SMALL_DEV_SQUAD = 'small_dev-v2.0.json'

with open(PATH_TO_DEV_SQUAD, 'r') as iofile:
    full_sample = json.load(iofile)

small_sample = {
    'version': full_sample['version'],
    'data': full_sample['data'][:1]
}

with open(PATH_TO_SMALL_DEV_SQUAD, 'w') as iofile:
    json.dump(small_sample, iofile)
```

```
In [15]: max_seq_length = 384
outside_pos = max_seq_length + 10
doc_stride = 128
max_query_length = 64
max_answer_length = 30
```

```
In [16]: examples = read_squad_examples()
```

```
max_query_length = 64
max_answer_length = 30
```

```
In [16]: examples = read_squad_examples(
    input_file=PATH_TO_SMALL_DEV_SQUAD,
    is_training=False,
    version_2_with_negative=True)

features = convert_examples_to_features(
    examples=examples,
    tokenizer=tokenizer,
    max_seq_length=max_seq_length,
    doc_stride=doc_stride,
    max_query_length=max_query_length,
    is_training=False,
    cls_token_segment_id=0,
    pad_token_segment_id=0,
    cls_token_at_end=False
)

input_ids = torch.tensor([f.input_ids for f in features], dtype=torch.long)
input_mask = torch.tensor([f.input_mask for f in features], dtype=torch.long)
segment_ids = torch.tensor([f.segment_ids for f in features], dtype=torch.long)
cls_index = torch.tensor([f.cls_index for f in features], dtype=torch.long)
p_mask = torch.tensor([f.p_mask for f in features], dtype=torch.float)

example_index = torch.arange(input_ids.size(0), dtype=torch.long)
dataset = TensorDataset(input_ids, input_mask, segment_ids, example_index, cls_index, p_mask)
```

100% |██████████| 208/208 [00:01<00:00, 136.52it/s]

```
In [17]: eval_sampler = SequentialSampler(dataset)
eval_dataloader = DataLoader(dataset, sampler=eval_sampler, batch_size=8)
```

Теперь давайте запустим evaluation модели на нашем маленьком кусочке dev-датасета. Evaluation может занять достаточно большое время. В нашем случае это будет не больше пары минут, и, пока, давайте рассмотрим код, который у нас есть для evaluation модели. Здесь мы используем наш "eval_dataloader", мы разбиваем его на батчи. В цикле проходимся по нашим

батчам (размер каждого батча равен 8). Дальше задаём, что мы не хотим считать градиент на каждом шаге, поскольку у нас происходит процесс evaluation, а не обучения модели. Мы распаковываем наш батч и передаём полученные данные в нашу модель, чтобы получить предсказание модели. И дальше складываем полученные результаты в list под названием "all_results". Давайте посмотрим на то, как выглядит наш лист all_results. Например, на его нулевой элемент. Но, сначала, нужно немножко подождать, пока наша модель доэвалюйтится. Отлично, эта ячейка кода выполнилась — смотрим на то что у нас содержится в листе all_results. На самом деле, выход не очень интерпретируемый. У нас есть некоторые айдишники, у нас есть start_logits, есть end_logits, и в целом, по вот такому вот списку достаточно сложно что-то понять. Поэтому давайте, с помощью функции "write_predictions()", которая уже есть в готовом виде в библиотеке [pytorch-transformers](#), сформируем человекочитаемый ответ. Для начала, нам нужно задать несколько констант. Давайте зададим параметры "n_best_size" равным 5 — это значит, что на каждый вопрос мы будем генерировать по 5 самых лучших ответов. Также нам нужно задать пути до файла с нашими предсказаниями, в файл под названием "output_1_best_file" мы будем писать айдишник нашего вопроса и один наиболее вероятный ответ на этот вопрос. Этот файл нам нужен для того, чтобы посчитать метрики, потому что для подсчёта метрик нам нужно не 5 возможных вариантов ответа, а всего один наиболее вероятный. Для того чтобы просмотреть глазами наши варианты ответа, мы всё-таки хотим видеть не один вариант, в котором наиболее уверена наша модель, а несколько вариантов. Поэтому будем записывать N лучших вариантов (в нашем случае, 5 лучших вариантов) в файл "output_n_best_file". И, кроме того, давайте поставим, что параметр "version_2_with_negative" у нас будет равен True. Генерируем предсказания с помощью функции "write_predictions()". Файл с предсказанием будет выглядеть следующим образом. У нас есть ordered dict, в котором содержится ID вопроса и наиболее вероятный вариант ответа на него. И, как раз, вся эта информация лежит в файле "output_1_best_file". Теперь давайте посчитаем метрики на нашем датасете. Мы будем использовать "eval_opts" — по структуре оно очень похоже на класс "train_opts", который мы использовали, только поля чуть-чуть другие. И, кроме того, с помощью функции и "evaluate_on_squad^[1]()" мы можем посчитать метрики, используя наши evaluate options. Давайте посчитаем их! Получается, что абсолютно правильный ответ, с точностью до символа, был получен в 78% случаев — это достаточно неплохо. [f-мера](#) 68, а если говорить про вопросы, на которые нет ответа, то [BERT](#) смог выдать правильный ответ, то есть угадать, что на этот вопрос нет ответа, в 54% случаев.

[1] <https://rajpurkar.github.io/SQuAD-explorer/>

100% | 208/208 [00:01<00:00, 121.06it/s]

In [41]: eval_sampler = SequentialSampler(dataset)
eval_dataloader = DataLoader(dataset, sampler=eval_sampler, batch_size=8)

In [*]: def to_list(tensor):
 return tensor.detach().cpu().tolist()

all_results = []
for idx, batch in enumerate(tqdm.tqdm_notebook(eval_dataloader, desc="Evaluating")):
 model.eval()
 batch = tuple(t.to(device) for t in batch)
 with torch.no_grad():
 inputs = {'input_ids': batch[0],
 'attention_mask': batch[1]
 }
 inputs['token_type_ids'] = batch[2]
 example_indices = batch[3]
 outputs = model(**inputs)

 for i, example_index in enumerate(example_indices):
 eval_feature = features[example_index.item()]
 unique_id = int(eval_feature.unique_id)
 result = RawResult(unique_id=unique_id,
 start_logits=to_list(outputs[0][i]),
 end_logits=to_list(outputs[1][i]))
 all_results.append(result)

Evaluating: 52% | 14/27 [01:08<01:03, 4.91s/it]

In [19]: n_best_size = 5
do_lower_case = True

```
unique_id = int(eval_feature.unique_id)
result = RawResult(unique_id=unique_id,
                   start_logits=to_list(outputs[0][i]),
                   end_logits=to_list(outputs[1][i]))
all_results.append(result)
```

Evaluating: 100% | 27/27 [02:12<00:00, 4.92s/it]

In [43]: all_results[0]

Out[43]: RawResult(unique_id=1000000000, start_logits=[-6.092804431915283, -11.206320762634277, -11.824389457702637, -11.866362571716309, -11.848936080932617, -11.488204002380371, -11.480500221252441, -11.728873252868652, -11.1748552322387, -8.258770942687988, -8.198786735534668, -9.782806396484375, -10.650155067443848, -11.243077278137207, -11.384310722351074, -10.723287582397461, -11.626534461975098, -11.893603324890137, -11.418963432312012, -11.74912111.0591268533942871, -11.421287536621094, -11.024538040161133, -11.484709739685059, -11.66255283355947814941, -11.40916919708252, -10.677251815795898, -10.957788467407227, -11.21056842803955, -11.58044715773, -11.211819648742676, -11.1763334274292, -11.837714195251465, -9.805883407592773, -10.4651947021484389355, -11.843772888183594, -11.082221031188965, -11.627579689025879, -11.31775951385498, -11.5712860810279846191, -10.878663063049316, -5.569657802581787, -8.86194133758545, -7.836868762969971, -8.562983512878418, 7.984959125518799, -4.779574871063232, -9.943062782287598, -11.6543560028076177324, -10.994288444519043, -9.627310752868652, -10.33826732635498, -10.584446907043457, -10.9258220558166504, -11.262892723083496, -11.257826805114746, -10.933144569396973, -10.9534435272273, -11.371384620666504, -10.561057090759277, -9.075515747070312, -11.937963485717773, -9.5883476257324, -5.500494003295898, -11.809952735900879, -7.737244129180908, -11.93971157073974, -12.083854675292969, -12.0913724899292, -11.275609016418457, -11.354109764099121, -11.545376965332, -11.641295433044434, -12.106725692749023, -11.474525451660156, -11.6566028594970711.757279396057129, -11.691466331481934, -12.249774932861328, -11.869754791259766, -10.619814038086, -11.327420234680176, -11.553250312805176, -9.758345603942871, -11.5175924301147411.1554542728271484, -10.563084602355957, -11.110987663269043, -11.95317554473877, -10.93698980137, -11.281010627746582, -11.737060546875, -11.571770668029785, -11.625266075134277]

In [19]: n_best_size = 5

```
[...], -11.368694305419922, -11.36880970012207, -11.35778993286133, -11.348855972290039, -11.378762245178223, -11.37571907043457, -11.358270645141602, -11.36280632019043, -11.364044189453125, -11.361371994018555, -11.347851753234863])]
```

```
In [19]: n_best_size = 5
do_lower_case = True
output_prediction_file = 'output_1best_file'
output_nbest_file = 'output_nbest_file'
output_na_prob_file = 'output_na_prob_file'
verbose_logging = True
version_2_with_negative = True
null_score_diff_threshold = 0.0
```

```
In [44]: # Генерируем файл с n лучшими ответами 'output_nbest_file'
write_predictions(examples, features, all_results, n_best_size,
                  max_answer_length, do_lower_case, output_prediction_file,
                  output_nbest_file, output_na_prob_file, verbose_logging,
                  version_2_with_negative, null_score_diff_threshold)
```

```
Out[44]: OrderedDict([( ('56ddde6b9a695914005b9628', 'France'),
    ('56ddde6b9a695914005b9629', 'the 10th and 11th centuries'),
    ('56ddde6b9a695914005b962a', 'Denmark, Iceland and Norway'),
    ('56ddde6b9a695914005b962b', 'Rollo'),
    ('56ddde6b9a695914005b962c', '10th'),
    ('5ad39d53604f3c001a3fe8d1', ''),
    ('5ad39d53604f3c001a3fe8d2', 'The Normans'),
    ('5ad39d53604f3c001a3fe8d3', 'The Normans'),
    ('5ad39d53604f3c001a3fe8d4', '10th century'),
    ('56ddf4066d3e219004dad5f', 'William the Conqueror'),
    ('56ddf4066d3e219004dad60', 'Richard I of Normandy'),
    ('56ddf4066d3e219004dad61', 'Catholic orthodoxy'),
    ('5ad3a266604f3c001a3fea27',
     'political, cultural and military impact'),
    ('5ad3a266604f3c001a3fea28', '')])
```



```
( ('56ddde6b9a695914005b9628', 'The Normans'),
    ('5ad39d53604f3c001a3fe8d2', 'The Normans'),
    ('5ad39d53604f3c001a3fe8d3', '10th century'),
    ('56ddf4066d3e219004dad5f', 'William the Conqueror'),
    ('56ddf4066d3e219004dad60', 'Richard I of Normandy'),
    ('56ddf4066d3e219004dad61', 'Catholic orthodoxy'),
    ('5ad3a266604f3c001a3fea27',
     'political, cultural and military impact'),
    ('5ad3a266604f3c001a3fea28', ''),
    ('5ad3a266604f3c001a3fea29', ''),
    ('5ad3a266604f3c001a3fea2a', 'Richard I'),
    ('5ad3a266604f3c001a3fea2b', ''),
    ('56dde0379a695914005b9636', 'Normant'),
    ('56dde0379a695914005b9637', '9th century'))
```

```
In [23]: # Считаем метрики используя официальный SQuAD script
evaluate_options = EVAL_OPTS(data_file=PATH_TO_SMALL_DEV_SQUAD,
                             pred_file=output_prediction_file,
                             na_prob_file=output_na_prob_file)
results = evaluate_on_squad(evaluate_options)

{
    "exact": 65.38461538461539,
    "f1": 68.14217032967032,
    "total": 208,
    "HasAns_exact": 78.125,
    "HasAns_f1": 84.0997023809524,
    "HasAns_total": 96,
    "NoAns_exact": 54.464285714285715,
    "NoAns_f1": 54.464285714285715,
    "NoAns_total": 112,
    "best_exact": 69.71153846153847,
    "best_exact_thresh": -19.84104323387146,
    "best_f1": 72.10851648351647,
    "best_f1_thresh": -10.105807900428772
```



Отлично! Мы смогли получить достаточно высокие значения метрик, но всё-таки давайте посмотрим глазами на вопросы и предсказанные [BERT](#) ответы, а также на каноничные ответы из нашего датасета. Давайте прочитаем файл с названием "output_n_best_file". Дальше сформируем словарь, в котором хранятся ID вопроса, а также текст вопроса, варианты ответа

из нашего датасета и параграф, на котором основывался вопрос. И распечатаем вопросы-ответы BERT и ответы из нашего датасета. Для простоты понимания, давайте параграфы пока печатать не будем, (но, в принципе, можно раскомментировать вот этот кусочек кода и, также, смотреть на параграф, на котором основывался вопрос). Как вы видите, BERT с достаточно хорошей уверенностью (со стопроцентной уверенностью) отвечает правильно на первый вопрос. Похожая история происходит со вторым вопросом, BERT отвечает правильно с точностью до артикла, при этом, уверенность 71%. На третий вопрос мы также получаем правильный ответ с уверенностью 100%, и похожая ситуация происходит с остальными вопросами. Если говорить про вопросы, на которые в датасете ответа нет, то BERT достаточно неплохо предсказывает отсутствие ответа на вопрос, то есть BERT, примерно в 50% случаев, научился понимать, что информации, содержащаяся в параграфе, недостаточно для ответа на этот вопрос. Всего здесь 208 вопросов — вы можете более подробно изучить ответы BERT на них. Но мы, давайте, перейдём к следующему простому заданию.

```

        pred_file=output_prediction_file,
        na_prob_file=output_na_prob_file)
results = evaluate_on_squad(evaluate_options)

{
    "exact": 65.38461538461539,
    "f1": 68.14217032967032,
    "total": 208,
    "HasAns_exact": 78.125,
    "HasAns_f1": 84.0997023809524,
    "HasAns_total": 96,
    "NoAns_exact": 54.464285714285715,
    "NoAns_f1": 54.464285714285715,
    "NoAns_total": 112,
    "best_exact": 69.71153846153847,
    "best_exact_thresh": -19.84104323387146,
    "best_f1": 72.10851648351647,
    "best_f1_thresh": -10.105807900428772
}

```

Посмотрим глазами на вопросы и предсказанные БЕРТом ответы:

```

In [24]: with open('output_nb3est_file', 'r') as iofile:
           predicted_answers = json.load(iofile)

In [25]: questions = {}
for paragraph in small_sample['data'][0]['paragraphs']:
    for question in paragraph['qas']:
        questions[question['id']] = {
            'question': question['question'],
            'answers': question['answers'],
            'paragraph': paragraph['context']
        }

```



```

        paragraph + paragraph['context']
    }

In [48]: for q_num, (key, data) in enumerate(predicted_answers.items()):
    gt = '' if len(questions[key]['answers']) == 0 else questions[key]['answers'][0]['text']
    print('Вопрос {0}:'.format(q_num+1), questions[key]['question'])
    print('-----')
    print('Ground truth:', gt)
    print('-----')
    print('Ответы, предсказанные БЕРТом:')
    preds = ['{0} '.format(ans_num + 1) + answer['text'] + \
             ' (уверенность {0:.2f}%)'.format(answer['probability']*100) + \
             '\n' for ans_num, answer in enumerate(data)]
    print('\n'.join(preds))
#    print('-----')
#    print('Парраграф:', questions[key]['paragraph'])
    print('\n\n')

```

Вопрос 1: In what country is Normandy located?

Ground truth: France

Ответы, предсказанные БЕРТом:
1) France (уверенность 100.00%)
2) France. (уверенность 0.00%)
3) France. They were descended from Norse ("Norman" comes from "Norseman") raiders and pirates and Norway (уверенность 0.00%)
4) Normandy, a region in France (уверенность 0.00%)
5) France. They were descended from Norse ("Norman" comes from "Norseman") raiders and pirates and Norway (уверенность 0.00%)
6) (уверенность 0.00%)

Вопрос 2: When were the Normans in Normandy?



б) (уверенность 0.00%)

Вопрос 2: When were the Normans in Normandy?

Ground truth: 10th and 11th centuries

Ответы, предсказанные БЕРТом:

- 1) the 10th and 11th centuries (уверенность 71.27%)
- 2) 10th and 11th centuries (уверенность 18.23%)
- 3) in the 10th and 11th centuries (уверенность 10.51%)
- 4) the 10th (уверенность 0.00%)
- 5) the 10th and 11th (уверенность 0.00%)
- 6) (уверенность 0.00%)



Вопрос 3: From which countries did the Norse originate?

Ground truth: Denmark, Iceland and Norway

Ответы, предсказанные БЕРТом:

- 1) Denmark, Iceland and Norway (уверенность 100.00%)
- 2) Denmark, Iceland (уверенность 0.00%)
- 3) Denmark (уверенность 0.00%)
- 4) Norway (уверенность 0.00%)
- 5) Iceland and Norway (уверенность 0.00%)
- 6) (уверенность 0.00%)

Вопрос 4: Who was the Norse leader?

Ground truth: Rollo

Вопрос 4: Who was the Norse leader?

Ground truth: Rollo

Ответы, предсказанные БЕРТом:

- 1) Rollo (уверенность 99.98%)
- 2) Rollo, (уверенность 0.02%)
- 3) leader Rollo (уверенность 0.00%)
- 4) under their leader Rollo (уверенность 0.00%)
- 5) their leader Rollo (уверенность 0.00%)
- 6) (уверенность 0.00%)

Вопрос 5: What century did the Normans first gain their separate identity?

Ground truth: 10th century

Ответы, предсказанные БЕРТом:

- 1) 10th (уверенность 99.10%)
- 2) 10th century (уверенность 0.77%)
- 3) the 10th (уверенность 0.11%)
- 4) the first half of the 10th (уверенность 0.02%)
- 5) first half of the 10th (уверенность 0.00%)
- 6) (уверенность 0.00%)



Вопрос 6: Who gave their name to Normandy in the 1000's and 1100's

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) (уверенность 100.00%)
- 2) the Normans (уверенность 0.00%)

- 3) the 10th (уверенность 0.01%)
4) the first half of the 10th (уверенность 0.02%)
5) first half of the 10th (уверенность 0.00%)
6) (уверенность 0.00%)

Вопрос 6: Who gave their name to Normandy in the 1000's and 1100's

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) (уверенность 100.00%)
- 2) The Normans (уверенность 0.00%)
- 3) Normans (уверенность 0.00%)
- 4) s (уверенность 0.00%)
- 5) The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) (уверенность 0.00%)



SAMSUNG
Research
Russia

Вопрос 7: What is France a region of?

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) The Normans (уверенность 40.81%)
- 2) Normandy (уверенность 37.83%)
- 3) Normans (уверенность 10.83%)
- 4) (уверенность 6.20%)
- 5) Normandy, a region in France. (уверенность 4.33%)

Вопрос 8: Who did King Charles III swear fealty to?

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) The Normans (уверенность 52.38%)
- 2) King Charles III of West Francia. (уверенность 28.48%)
- 3) King Charles III of West Francia (уверенность 11.81%)
- 4) Normans (уверенность 4.20%)
- 5) West Francia. (уверенность 3.14%)
- 6) (уверенность 0.00%)



Вопрос 9: When did the Frankish identity emerge?

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) 10th century (уверенность 55.42%)
- 2) first half of the 10th century (уверенность 30.60%)

Вопрос 9: When did the Frankish identity emerge?

Ground truth:

Ответы, предсказанные БЕРТом:

- 1) 10th century (уверенность 55.42%)
- 2) first half of the 10th century (уверенность 30.60%)
- 3) 10th century, (уверенность 7.65%)
- 4) first half of the 10th century, (уверенность 4.22%)
- 5) the first half of the 10th century (уверенность 2.11%)
- 6) (уверенность 0.00%)

Вопрос 10: Who was the duke in the battle of Hastings?

Ground truth: William the Conqueror

Ответы, предсказанные БЕРТом:

- 1) William the Conqueror (уверенность 100.00%)
- 2) Conqueror (уверенность 0.00%)
- 3) duke, William the Conqueror (уверенность 0.00%)
- 4) William (уверенность 0.00%)
- 5) the Conqueror (уверенность 0.00%)
- 6) (уверенность 0.00%)

Вопрос 11: Who ruled the duchy of Normandy

Ground truth: Richard I



Вопрос 11: Who ruled the duchy of Normandy

Ground truth: Richard I

Ответы, предсказанные БЕРТом:

- 1) Richard I of Normandy (уверенность 95.59%)
- 2) Richard I (уверенность 4.37%)
- 3) under Richard I of Normandy (уверенность 0.03%)
- 4) Normandy, which they formed by treaty with the French crown, was a great fief of medieval France, and under Richard I of Normandy (уверенность 0.00%)
- 5) under Richard I (уверенность 0.00%)
- 6) (уверенность 0.00%)

Вопрос 12: What religion were the Normans

Ground truth: Catholic

Ответы, предсказанные БЕРТом:

- 1) Catholic orthodoxy (уверенность 99.96%)
- 2) Christian piety, becoming exponents of the Catholic orthodoxy (уверенность 0.02%)
- 3) the Catholic orthodoxy (уверенность 0.01%)
- 4) orthodoxy (уверенность 0.01%)
- 5) Catholic (уверенность 0.00%)
- 6) (уверенность 0.00%)



Вопрос 13: What type of major impact did the Norman dynasty have on modern Europe?

Ground truth:

Ответы, предсказанные БЕРТом:

Для того чтобы получить такие достаточно хорошие результаты, мы тренировали [BERT](#) в целых пять эпох. Что же будет, если мы потренируем BERT чуть меньше количество времени? Как вы помните, тренировать 5 эпох BERT заняло достаточно много времени. Давайте попробуем потренировать всего одну эпоху и посмотрим, что произойдет, насколько метрики станут ниже. Мы не будем тренировать нашу модель прямо сейчас, а просто подгрузим веса модели

после тренировки одной эпохи (файлик называется "bert_squad^[1]_1epoch"). Подгружаем веса и ещё раз делаем evaluation. Посмотрим на метрики. Как вы видите, метрики стали чуть ниже, но не сильно ниже. На вопросы с ответом модель даёт правильный ответ в 76% случаев, сравнивая с 78% после 5 эпох, а [f-мера](#) стала равна 71. Кроме того, на вопросы без ответа — на вопросы, на которые нельзя ответить, учитывая данный параграф текста, модель научилась отвечать (точнее... не отвечать) даже лучше: 61% случаев, сравнивая с 54% после пяти эпох. Получается, что если вам нужен сравнительно быстрый результат, можно немного пожертвовать качеством работы модели и обучать модель не 7 часов, а примерно полтора часа.

[1] <https://rajpurkar.github.io/SQuAD-explorer/>

Загружаем токенизатор для БЕРТа и BertForQuestionAnswering из pytorch-transformers:

```
In [36]: tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True, do_basic_tokenize=True)
model = BertForQuestionAnswering.from_pretrained('bert-base-uncased')
```

```
In [50]: # если Вы не хотите запускать файн-тюнинг, пропустите блок "Дообучение",
# подгрузите веса уже дообученной модели и переходите к блоку "Оценка качества"

if torch.cuda.is_available():
    model.cuda()
    model.load_state_dict(torch.load('models/bert_squad_1epoch.pt')) # если у вас есть GPU
else:
    model.load_state_dict(torch.load('models/bert_squad_1epoch.pt', map_location=device)) # если GPU нет
```

Дообучение

```
In [62]: # !pip install dataclasses
from dataclasses import dataclass

@dataclass
class TRAIN_OPTS:
    train_file : str = 'train-v2.0.json'      # SQuAD json-файл для обучения
    predict_file : str = 'dev-v2.0.json'        # SQuAD json-файл для тестирования
    model_type : str = 'bert'                  # тип модели (может быть 'bert', 'xlnet', 'xlm')
    model_name_or_path : str = 'bert-base-uncased' # путь до предобученной модели или название
    output_dir : str = '/tmp' # путь до директории, где будут храниться чекпоинты и предсказания
    device : str = 'cuda' # cuda или cpu
    n_gpu : int = 1 # количество gpu для обучения
    cache_dir : str = '' # где хранить предобученные модели, загруженные с s3

    # Если true, то в датасет будут включены вопросы, на которые нет ответов.
    version_2_with_negative : bool = True
```



```
# Если true, то в датасет будут включены вопросы, на которые нет ответов.
version_2_with_negative : bool = True
# Если (null_score - best_non_null) больше, чем порог, предсказывать null.
null_score_diff_threshold : float = 0.0
# Максимальная длина входной последовательности после WordPiece токенизации. Sequences
# Последовательности длиннее будут укорочены, для более коротких последовательностей будет использован паддинг
max_seq_length : int = 384
# Сколько stride использовать при делении длинного документа на чанки
doc_stride : int = 128
# Максимальное количество токенов в вопросе. Более длинные вопросы будут укорочены до этой длины
max_query_length : int = 128 #

do_train : bool = True
do_eval : bool = True

# Запускать ли evaluation на каждом logging_step
evaluate_during_training : bool = True
# Должно быть True, если Вы используете uncased модели
do_lower_case : bool = True #

per_gpu_train_batch_size : int = 8 # размер батча для обучения
per_gpu_eval_batch_size : int = 8 # размер батча для eval
learning_rate : float = 5e-5 # learning rate
gradient_accumulation_steps : int = 1 # количество шагов, которые нужно сделать перед батчом
weight_decay : float = 0.0 # weight decay
adam_epsilon : float = 1e-8 # эпсилон для Adam
max_grad_norm : float = 1.0 # максимальная норма градиента
num_train_epochs : float = 5.0 # количество эпох на обучение
max_steps : int = -1 # общее количество шагов на обучение (override num_train_epochs)
warmup_steps : int = 0 # warmup
n_best_size : int = 5 # количество ответов, которые надо сгенерировать для записи в файл
max_answer_length : int = 30 # максимально возможная длина ответа
verbose_logging : bool = True # печатать или нет warnings, относящиеся к обработке ошибочных
```



```
('56dddf4066d3e219004dad61', 'Catholic orthodoxy'),
('5ad3a266604f3c001a3fea27', 'political, cultural and military'),
('5ad3a266604f3c001a3fea28', ''),
('5ad3a266604f3c001a3fea29', ''),
('5ad3a266604f3c001a3fea2a', 'Richard I'),
('5ad3a266604f3c001a3fea2b', ''),
('56dde0379a695914005b9636', 'Normant'),
('56dde0379a695914005b9637', '9th century'),
('5ad3ab70604f3c001a3feb89', '')
```

```
In [57]: # Считаем метрики используя официальный SQuAD script
evaluate_options = EVAL_OPTS(data_file=PATH_TO_SMALL_DEV_SQUAD,
                             pred_file=output_prediction_file,
                             na_prob_file=output_na_prob_file)
results = evaluate_on_squad(evaluate_options)

{
    "exact": 68.26923076923077,
    "f1": 71.22710622710623,
    "total": 208,
    "HasAns_exact": 76.04166666666667,
    "HasAns_f1": 82.45039682539682,
    "HasAns_total": 96,
    "NoAns_exact": 61.607142857142854,
    "NoAns_f1": 61.607142857142854,
    "NoAns_total": 112,
    "best_exact": 72.59615384615384,
    "best_exact_thresh": -5.18088960647583,
    "best_f1": 74.75274725274724,
    "best_f1_thresh": -2.038414478302002
}
```

Посмотрим глазами на вопросы и предсказанные БЕРТом ответы:



На этом семинаре мы разобрали, как работает скрипт по дообучению модели `BertForQuestionAnswering` из библиотеки [pytorch-transformers](#). Получить аналогичные результаты, на самом деле, вы могли бы просто склонировав репозитории и запустив скрипт "`run_squad.py`"^[1] (например, вот так — вот таким образом). Однако, если вам вдруг понадобится что-то поменять в скрипте, модифицировать процесс обучения или даже просто обучить модель на другом, чуть менее популярном датасете, вам придётся закапываться в код библиотеки гораздо глубже, чем простой вызов готового скрипта. Этот семинар может послужить отправной точкой в более подробном освоении кода библиотеки и запуске [BERT](#), [XL трансформера](#), [GPT-2](#) или любых других моделей для решения ваших задач. Успехов!

[1] https://github.com/huggingface/transformers/blob/master/examples/run_squad.py.

```
-----  
Ground truth:  
-----  
Ответы, предсказанные БЕРТом:  
1) (уверенность 100.00%)  
2) several monks (уверенность 0.00%)  
3) monks (уверенность 0.00%)  
4) several monks of Saint-Evroul (уверенность 0.00%)  
5) monks of Saint-Evroul (уверенность 0.00%)
```

```
In [ ]: !export SQUAD_DIR=/path/to/SQuAD  
  
python run_squad.py \  
--model_type bert \  
--model_name_or_path bert-base-cased \  
--do_train \  
--do_eval \  
--do_lower_case \  
--train_file $SQUAD_DIR/train-v1.1.json \  
--predict_file $SQUAD_DIR/dev-v1.1.json \  
--per_gpu_train_batch_size 12 \  
--learning_rate 3e-5 \  
--num_train_epochs 2.0 \  
--max_seq_length 384 \  
--doc_stride 128 \  
--output_dir /tmp/debug_squad/
```

```
In [ ]:
```

