

▼ Дисклеймер!

Вся эта тетрадка переработка материалов семинара взятого вот [ТУТ](#). Так же считаю правильным отметить тот факт, что я пока не очень понял как это все работает. Только в общих чертах. Тем не менее метрику я выбил))

Весь код был выполнен и результаты остались в тетрадке. F1 score = 0.83 или около того. В принципе, если у тебя эта тетрадка при подключении смогла получить в свое распоряжение GPU, то ты можешь перезапустить ее и прогнать. У меня на GPUs весь код выполнялся минут за 40. Тут встроенное скачивание датасета с просторов сети, так что все будет работать нормально (не захотел подключать свой Google Drive).

▼ Первая попытка обучить BERT

Double-click (or enter) to edit

▼ Подготовка данных

Попробуем обучить модель BERT на наших данных и посмотрим какую метрику F1 мы сможем получить.

Мы будем использовать реализацию BERT из библиотеки pytorch-transformers, которая содержит почти все последние архитектуры.

```
! pip install pytorch-transformers
```

```
Collecting pytorch-transformers
  Downloading pytorch-transformers-1.2.0-py3-none-any.whl (176 kB)
    |████████████████████████████████████████| 176 kB 4.1 MB/s
Collecting sentencepiece
  Downloading sentencepiece-0.1.96-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64
    |████████████████████████████████████████| 1.2 MB 43.0 MB/s
Collecting boto3
  Downloading boto3-1.19.5-py3-none-any.whl (131 kB)
    |████████████████████████████████████████| 131 kB 43.8 MB/s
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: torch>=1.0.0 in /usr/local/lib/python3.7/dist-packages (f
Collecting sacremoses
  Downloading sacremoses-0.0.46-py3-none-any.whl (895 kB)
```

```

|████████████████████████████████████████| 895 kB 34.4 MB/s
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from pyt
Requirement already satisfied: regex in /usr/local/lib/python3.7/dist-packages (from pyt
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from pyt
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packag
Collecting jmespath<1.0.0,>=0.7.1
  Downloading jmespath-0.10.0-py2.py3-none-any.whl (24 kB)
Collecting s3transfer<0.6.0,>=0.5.0
  Downloading s3transfer-0.5.0-py3-none-any.whl (79 kB)
|████████████████████████████████████████| 79 kB 7.8 MB/s
Collecting boto3<1.23.0,>=1.22.5
  Downloading boto3-1.22.5-py3-none-any.whl (8.1 MB)
|████████████████████████████████████████| 8.1 MB 39.6 MB/s
Collecting urllib3<1.27,>=1.25.4
  Downloading urllib3-1.26.7-py2.py3-none-any.whl (138 kB)
|████████████████████████████████████████| 138 kB 50.8 MB/s
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/local/lib/python3.7/c
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (f
  Downloading urllib3-1.25.11-py2.py3-none-any.whl (127 kB)
|████████████████████████████████████████| 127 kB 46.8 MB/s
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (from sa
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from sac
Installing collected packages: urllib3, jmespath, boto3, s3transfer, sentencepiece, s
  Attempting uninstall: urllib3
    Found existing installation: urllib3 1.24.3
    Uninstalling urllib3-1.24.3:
      Successfully uninstalled urllib3-1.24.3
ERROR: pip's dependency resolver does not currently take into account all the packages i
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is incompati
Successfully installed boto3-1.19.5 boto3-1.22.5 jmespath-0.10.0 pytorch-transformers

```

Импортируем необходимые библиотеки.

```

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
from sklearn.metrics import f1_score

```

SEED = 42

Хотелось бы использовать GPU процессоры для нашей задачи. Попробуем получить нужные ресурсы.

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

if device == torch.device('cpu'):
    print('Using cpu')
else:
    n_gpu = torch.cuda.device_count()
    print('Using {} GPUs'.format(torch.cuda.get_device_name(0)))

    Using Tesla K80 GPUs
```

Загрузим наши данные. Для модели BERT попробуем обойтись без предобработки текстов.

```
import pandas as pd

df_tweets = pd.read_csv('/content/drive/MyDrive/DropBOX/lemmatized_toxic_tweets.csv')

df_tweets.sample(5)
```

	toxic	lemmas
114493	0	onlinesource small yes title on wikipedia deb...
125133	0	when and where be aaviksoo born the where be ...
73695	0	i have call numerous time here for a hole yea...
140531	0	auc this image or medium upload after may jan...
90124	0	red link thanks for your message mutt

```
df_tweets.shape

(159571, 2)
```

```
df_tweets.dropna(inplace=True)
```

Специальные токены [CLS] и [SEP], которые мы добавляем в начало и конец предложения нужны для того, чтобы наша модель BERT могла правильно распознать задачу и данные, которые мы ей скормим.

- CLS - метка задачи - classification
- SEP - метка разделения текстов - separation

Также нам надо сохранить все наши отметки о токсичности твитов в отдельной переменной labels.

```
sentences = df_tweets['lemmas'].values
```

```
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = df_tweets['toxic'].values
```

Проверим, что у нас совпадают размеры массивов с текстами и метками.

```
assert len(sentences) == len(labels)
```

```
print(sentences[100])
```

```
[CLS] however the moonlite edit note by golden daph be me on optus wake up wikkis so fu
```

Разделим наши данные на обучающую и тестовую выборки.

```
from sklearn.model_selection import train_test_split
```

```
train_sentences, test_sentences, train_gt, test_gt = train_test_split(sentences, labels, test
```

```
print(len(train_gt), len(test_gt))
```

```
127651 31913
```

▼ Inputs

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

```
from torch.nn 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])
```

```
100%|██████████| 231508/231508 [00:00<00:00, 316941.81B/s]
['[CLS]', 'it', 'show', 'in', 'your', 'rev', '##ere', '##ts', 'that', 'you', 'have', 'nc
```

BERT'у 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 - положительную.
- **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.

Паддинг нужен для того, чтобы BERT мог работать с предложениями разной длины.

Выбираем максимально возможную длину предложения (в нашем случае пусть это будет 150). Теперь более длинные предложения будем обрезать до 150 токенов, а для более коротких использовать паддинг. Возьмем готовую функцию `pad_sequences` из библиотеки `keras`.

```
input_ids = [tokenizer.convert_tokens_to_ids(x[:150]) for x in tokenized_texts]
input_ids = pad_sequences(
    input_ids,
    maxlen=150,
    dtype="long",
    truncating="post",
    padding="post"
)
attention_masks = [[float(i>0) for i in seq] for seq in input_ids]
```

Делим данные на train и val:

```
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
)
```

Преобразуем данные в pytorch тензоры:

```
train_inputs = torch.tensor(train_inputs)
train_labels = torch.tensor(train_labels)
train_masks = torch.tensor(train_masks)

validation_inputs = torch.tensor(validation_inputs)
validation_labels = torch.tensor(validation_labels)
validation_masks = torch.tensor(validation_masks)
```

```
train_labels

tensor([1, 1, 0, ..., 0, 0, 0])
```

Воспользуемся классом `DataLoader`. Это поможет нам использовать эффективнее память во время тренировки модели, так как нам не нужно будет загружать в память весь датасет. Данные по батчам будем разбивать произвольно с помощью `RandomSampler`. Если во время тренировки возникнет `Memory Error`, размер батча необходимо будет уменьшить.

```
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_dataloader = DataLoader(
    train_data,
    sampler=RandomSampler(train_data),
    batch_size=40
)

validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_dataloader = DataLoader(
    validation_data,
    sampler=SequentialSampler(validation_data),
    batch_size=40
)
```

▼ Обучение модели

Теперь когда данные подготовлены, надо написать пайплайн обучения модели.

Для начала мы хотим изменить предобученный BERT так, чтобы он выдавал метки для классификации текстов, а затем файнтюнить его на наших данных. Мы возьмем готовую модификацию BERTа для классификации из `pytorch-transformers`. Она интуитивно понятно называется `BertForSequenceClassification`. Это обычный BERT с добавленным линейным слоем для классификации.

Загружаем [BertForSequenceClassification](#):

```
from pytorch_transformers import AdamW, BertForSequenceClassification
```

Теперь подробнее рассмотрим процесс фاین-тюнинга. Как мы помним, первый токен в каждом предложении - это [CLS]. В отличие от скрытого состояния, относящегося к обычному слову (не метке [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 будет вектор со скрытым состоянием, относящийся к метке [CLS]. Дополнительный полносвязный слой, который мы добавили, имеет размер [hidden_state, количество_классов], в нашем случае количество классов равно двум. То есть на выходе мы получим два числа, представляющих классы "положительная эмоциональная окраска" и "отрицательная эмоциональная окраска".

Процесс дообучения достаточно дешев. По факту мы тренируем наш верхний слой и немного меняем веса во всех остальных слоях в процессе, чтобы подстроиться под нашу задачу.

Иногда некоторые слои специально "замораживают" или применяют разные стратегии работы с learning rate, в общем, делают все, чтобы сохранить "хорошие" веса в нижних слоях и ускорить дообучение. В целом, замораживание слоев BERTа обычно не сильно сказывается на итоговом качестве, однако надо помнить о тех случаях, когда данные, использованные для предобучения и дообучения очень разные (разные домены или стиль: академическая и разговорная лексика). В таких случаях лучше тренировать все слои сети, не замораживая ничего.

Загружаем BERT. bert-base-uncased - это версия "base" (в оригинальной статье рассказывается про две модели: "base" vs "large"), где есть только буквы в нижнем регистре ("uncased").

```
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.to(device)
```

```
(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)
)
(10): BertLayer(
  (attention): BertAttention(
    (self): BertSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
```

```

        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
    (output): BertSelfOutput(
        (dense): 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)
    )
    (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)
    )
)
(11): BertLayer(
    (attention): BertAttention(
        (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
        )
        (output): BertSelfOutput(
            (dense): 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)
        )
    )
    (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)
    )
)
)
(pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
)
)

```

Теперь обсудим гиперпараметры для обучения нашей модели. Авторы статьи советуют выбирать learning rate 5e-5, 3e-5, 2e-5, а количество эпох не делать слишком большим, 2-4 вполне достаточно. Мы пойдем еще дальше и попробуем дообучить нашу модель всего за одну эпоху.

```
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)

%%time

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
    b_input_ids = torch.tensor(b_input_ids).to(torch.int64)
    # 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.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()

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
    b_input_ids = torch.tensor(b_input_ids).to(torch.int64)
    # 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.array(label_ids)
    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
))

```





```
print("f1_score: {0:.2%}".format(
    f1_score(valid_labels, valid_preds)
))
```

f1_score: 84.52%

0 500 1000 1500 2000 2500 3000

▼ Оценка качества на отложенной выборке

Процент правильных предсказаний на валидационной выборке: 97.03%

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:

```
tokenized_texts = [tokenizer.tokenize(sent) for sent in test_sentences]
input_ids = [tokenizer.convert_tokens_to_ids(x[:150]) for x in tokenized_texts]
```

```
input_ids = pad_sequences(
    input_ids,
    maxlen=150,
    dtype="long",
    truncating="post",
    padding="post"
)
```

Создаем attention маски и приводим данные в необходимый формат:

```
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
)

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
    b_input_ids = torch.tensor(b_input_ids).to(torch.int64)
    # 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.array(label_ids)
    test_preds.extend(batch_preds)
    test_labels.extend(batch_labels)

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: UserWarning: To copy cc
# Remove the CWD from sys.path while we load stuff.

```

```

acc_score = accuracy_score(test_labels, test_preds)
print('Процент правильных предсказаний на отложенной выборке составил: {:.2%}'.format(
    acc_score
))

```

Процент правильных предсказаний на отложенной выборке составил: 96.90%

```
f1_score(test_labels, test_preds)
```

0.8324306025727828

```
from sklearn.metrics import recall_score, precision_score
```

```
print('1 эпоха: точность (precision) {0:.2%}, полнота (recall) {1:.2%}'.format(
    precision_score(test_labels, test_preds),
    recall_score(test_labels, test_preds)
))
```

1 эпоха: точность (precision) 88.93%, полнота (recall) 78.24%

✓ 0s completed at 2:17 AM

