Tips and tricks

Хрыльченко Кирилл

Математические методы анализа текстов 2020

October 6, 2020

Подготовка данных

1. Приводим текст в нижний регистр. Бывают кейсы, когда переводить не надо (определение токсичных комментариев)
2. Удаляем:
   - вредные символы — пунктуацию, всевозможные галочки и стрелочки. Используем библиотеку **re**, сложную регулярку можно составить используя **re.compile**;
   - стопслова, например, с помощью библиотеки **nltk**,
   - *редкие слова*,
   - *исправление опечаток*,
   - короткие слова.
3. **Hot tip:** посмотрите глазами на слова, которые не нашлись в w2v!
4. Итог: DCG@1000 $= 0.499$.

О списках

Питоновский **list** — динамический массив, НЕ N-связный список.

- элементы списка находятся в одной области памяти, последовательно
- индексация, добавление и удаление из конца списка — $O(1)$
- проверка на вхождение: *a in mylist* — $O(n)$!
- *mylist[i:j]* — копирует список, т.е. $O(n)$ в худшем случае
- *mylist1 + mylist2* — копирует оба списка в новый, т.е. $O(m+n)$!
- динамический массив — для списка выделена память с запасом (*table doubling*)
- **nltk.stopwords** — список
- список + [элемент] — копирует

Подготовка текста

Модели работают с индексами, а не со строками. Нужно нумеровать слова! Создаём словарь по имеющейся коллекции текста. Для этого:

- Считаем частоты слов с помощью **Counter**[1], убираем редкие слова
- сохраняем список со словами **vocab** — почему не нужен словарь **id2vocab**?
- создаем словарь **vocab2id**, в котором по слову можно найти его индекс (номер в списке)

Про словари:

- добавление, удаление, проверка на вхождение за $O(1)$

Переводим предложение в индексы как:

- *[vocab2id.get(word, UNK) for word in doc if word in vocab2id]*

[1] двойные циклы — *Counter(word for doc in corpus for word in doc)*

Подготовка текста

Трансформация батча[2]:

- Был список документов: ["А и Б сидели на трубе", "А в каком случае P = NP?", . . .]
- Теперь есть список списков индексов слов: [[1 5 2 3 19 4], [1 8 6 100 . . .], [. . .], . . .]

Модели нужен тензор — все списки должны быть одной длины. Для этого используем:

- **паддинг** — дополняем справа все объекты **индексом паддинга** до фиксированной длины **max length**.
- **nn.EmbeddingBag** — склеиваем все индексы в один большой список, запоминая **сдвиги**[3] в другом списке

[2] набор входных данных для модели, передаваемый "одним пакетом"

[3] offsets

Bucket sequencing

Как подбирать одну **max length** для всех батчей? Нет. Трата скорости и памяти.

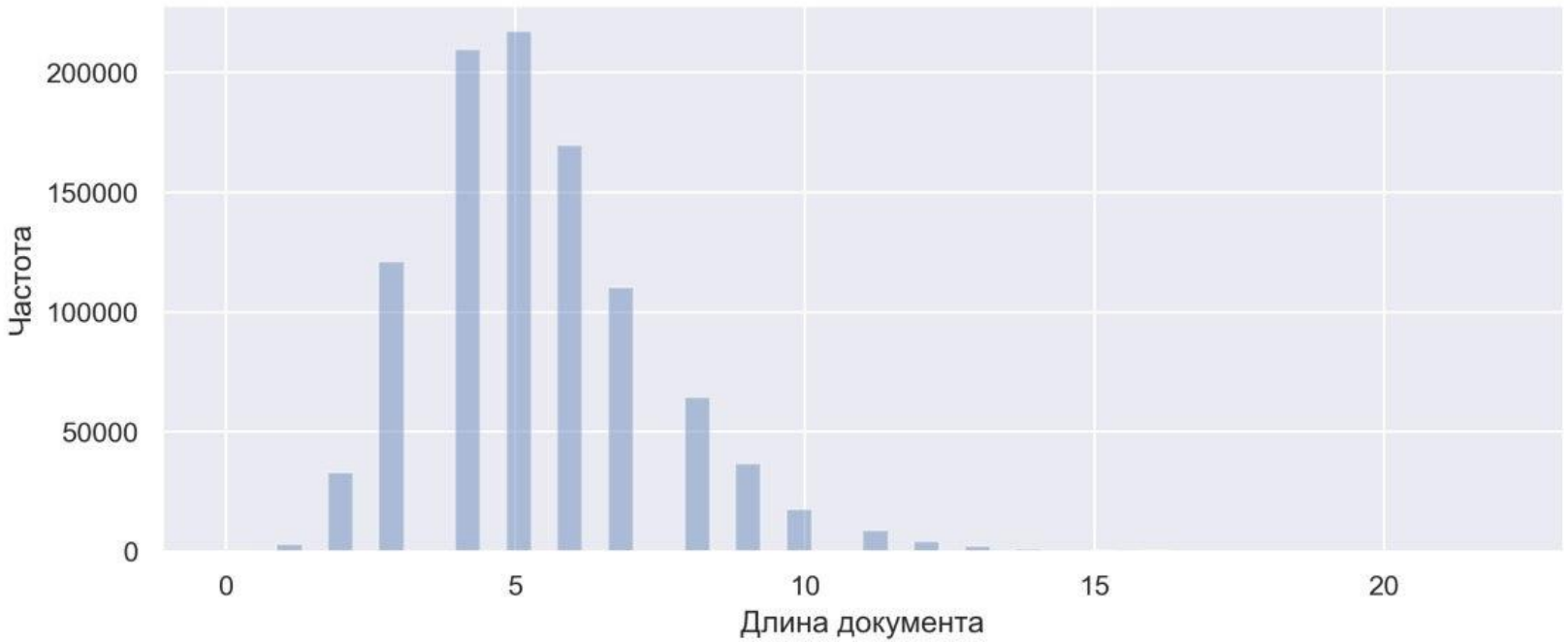

Figure: Гистограмма распределения длин документов

Bucket sequencing

Будем для каждого батча подбирать свою длину **padding**'а. Одинаковые длины разных батчей нам не нужны. Один батч — один тензор.

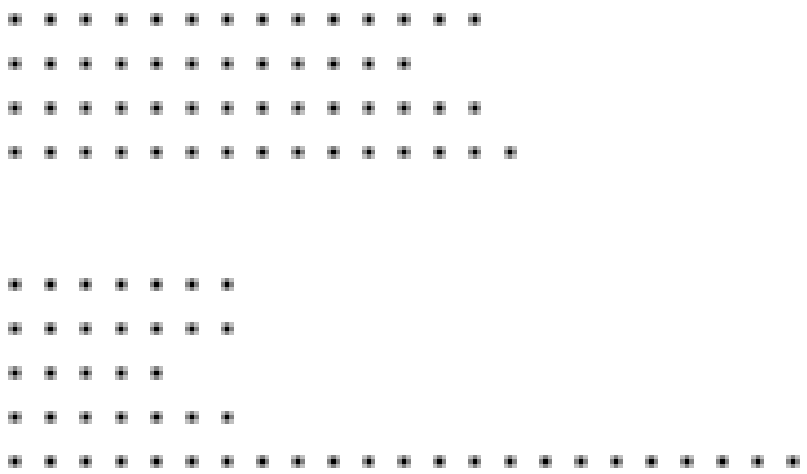

Figure: Два батча.

Лучше обрезать по квантилю: выберем такую длину, что 95% документов в батче будут короче. Сортируйте данные по длине на инференсе.

Преобразование триграмм

```python
class TrigramTokenizer:

    def __init__(self, words):

        # составляем множество всевозможных триграмм, составляющих слова из words
        # делаем маппинг триграм на индексы
        pass

    @staticmethod
    def get_trigrams(word):

        # возвращает список триграмм слова
        pass

    def __call__(self, word):

        # возвращает список индексов триграмм
        pass
```

Для **get trigrams** удобно использовать генератор.

Dataset

Dataset умеет выдавать пронумерованные элементы — $ds[i]$.

```
from torch.utils.data import Dataset

class TrainTrigramDataset(Dataset):

    def __init__(self, vocab, embeddings, tri_tokenizer):

        # формируем выборку для обучения
        # для всех слов из vocab ЗАРАНЕЕ считаем списки индексов
        pass

    def __len__(self):

        # возвращает количество объектов в выборке
        pass

    def __getitem__(self, idx):

        # возвращает список индексов триграмм для idx-го слова в выборке,
        # а также соответствующий эмбеддинг
        pass
```

DataLoader

DataLoader помогает формировать **батч** из выборки:

$$[dataset[i] \text{ for } i \text{ in } ids],$$

где *ids* — номера документов в датасете. Фактически, это итератор по батчам.

Он устроен чуть сложнее:

- Умеет работать параллельно — *num workers*. Полезно при тяжелом **getitem**,
- Может каждую эпоху перемешивать выборку[4] — *shuffle=True*[5],
- Может игнорировать последний батч, если он меньше, чем предыдущие
- *pin memory* ускоряет перенос объектов с CPU на GPU

[4] для более интересных махинаций смотрите **Sampler**'ы

[5] не забывайте выключать эту опцию на валидации

Dataset + DataLoader = Success

Фундамент любого пайплайна:

- Создаем датасет, из которого можно быстро доставать нужные нам объекты
- Создаем даталоадер, который эти объекты достаёт
- Ловим объекты у даталоадера

```
dataset = MyDataset(some_data)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

for obj, target in dataloader:

    # в цикле получили очередной батч,
    # делаем с ним все, что душе угодно
    pass
```

Важно: цикл проходит каждый объект единожды. Это одна **эпоха**.

collate fn

Внутри даталоадера собирается список объектов:
[dataset[i] for i in ids], затем передается специальной функции *collate fn*.

```
def default_collate_fn(batch):
    # пусть у нас объекты уже имеют одинаковую структуру (размерность)
    # принимает на вход [(obj1, target1), (obj2, target2), ...]
    objects, targets = zip(*batch)
    objects = torch.tensor(objects, dtype=torch.long)
    targets = torch.tensor(targets, dtype=torch.float)
    return objects, targets
```

В нашем случае (для триграмм):

- targets — эмбеддинги из w2v
- objects — списки индексов триграмм

Чтобы дальше использовать **nn.EmbeddingBag**, нужно склеить списки индексов в один, а также запомнить сдвиги. Для bucket sequencing используйте **torch.nn.utils.rnn.pad sequence** :)

Модель

Модель — та самая нейронная сеть!

- принимает на вход: тензор (сразу весь батч)
- выдает: предсказания для всего батча

```
class Model(nn.Module):

    def __init__(self, num_embeddings, embedding_dim):

        super().__init__()
        # параметры модели. Например, слои сети
        self.embeddings = nn.EmbeddingBag(num_embeddings, embedding_dim)
        pass

    def forward(self, x):

        # вся цепочка преобразований от исходного тензора до предсказаний
        pass
```

Для триграмм — принимает два параметра. Тензор индексов триграмм и тензор сдвигов!

Оптимизация

Функционалы ошибки:

- Для триграмм — nn.MSELoss,
- Для классификации — nn.BCELoss[6] и nn.BCEWithLogits[7]. Второй лучше!
- кастомный функционал ошибки — всего лишь нужен скаляр на выходе!

Оптимизатор — torch.optim.Adam:

- передаём генератор с параметрами модели,
- learning rate,
- weight decay — l_2 регуляризация.

[6] есть сигмоида в конце модели

[7] сигмоида входит в лосс

Пайплайн

Датасет и даталоадер, модель, функционал, оптимизатор — всё, что нужно для простого пайплайна!

```
criterion = nn.MSELoss()
model = Model()
optimizer = Adam(model.parameters())

model.train() # обсудим дальше

for batch in dataloader:

    objects, targets = batch
    predictions = model(objects) # скармливаем батч модели
    loss = criterion(predictions, targets) # считаем ошибку

    loss.backward() # считаем градиенты
    optimizer.step() # обновляем параметры модели
    optimizer.zero_grad() # убираем посчитанные градиенты
```

Регуляризация

Модель быстро переобучается. Что делать:

- Dropout между линейными слоями
- SpatialDropoput — зануляем измерения в исходных эмбеддингах слов или триграмм
- ограничиваем max norm в слое эмбеддингов
- batch, instance, layer нормализация
- l_2 регуляризация определенных (или всех) слоёв сети
- gradient clipping[8] — **torch.nn.utils.clip grad norm**
- фриз слоёв — например, предобученных эмбеддингов
- большой batch size
- маленький learning rate
- gradual unfreezing
- энтропия как функционал

[8] против взрыва градиентов

model.train vs model.eval:

- model.train — работает дропаут и считаются статистики по батч нормализации
- model.eval — отключается дропаут (детерминированный выход в сети), статистики по батч норме больше не считаются, а используются

Для BCEWithLogits и BCELoss (бинарной кросс-энтропии) — таргеты должны быть torch.float! Можно подавать вероятности (приближать другое распределение), смягчать метки, чтобы делать модель более "неуверенной".

Работа с GPU:

- всё либо на GPU, либо на CPU
- model.cuda(), затем для каждого батча obj.cuda(), target.cuda()
- лучше использовать переменную **device** и делать model.to(device) и obj.to(device)

- Есть дистрибутивный режим работы сразу с несколькими видеокартами — nn.DataParallel
- Для ускорения можно использовать mixed precision — например, библиотеку **apex**.
- broadcasting: **torch.einsum**[9] позволяет делать сложные операции с тензорами чуть быстрее
- Рандом в Python **очень медленный**. Используйте библиотеку **fastrand**, генерируйте случайные числа наперёд, используйте внутренние генераторы случайных чисел pytorch'а.
- Используйте **tensorboard**, чтобы отслеживать обучение нейронной сети.
- Когда не обучаете модель (на валидации, инференсе) — используйте контекстный менеджер **with torch.no grad**!
- не повторяйте одни и те же операции — например, векторизацию текста

[9] аналогично np.einsum

Третья часть в дз

- забудьте про Dataset и DataLoader, напишите функцию, сразу создающую целый батч
- необязательно использовать ВСЕ данные
- одна эпоха при не слишком неоптимальном написании должна занимать не более двух минут