Deep Learning School

Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Embeddings

Привет! В этом домашнем задании мы с помощью эмбеддингов решим задачу семантической классификации твитов.

Для этого мы воспользуемся предобученными эмбеддингами word2vec.

Для начала скачаем датасет для семантической классификации твитов:

```
!gdown https://drive.google.com/uc?id=1eE1FiUkXkcbw0McId4i7qY-L8hH-_Qph&export=download
!unzip archive.zip
```

Импортируем нужные библиотеки:

```
import math
import random
import string

import numpy as np
import pandas as pd
import seaborn as sns

import torch
import nltk
import gensim
import gensim.downloader as api


random.seed(42)
np.random.seed(42)
torch.random.manual_seed(42)
torch.cuda.random.manual_seed(42)
torch.cuda.random.manual_seed_all(42)

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

data = pd.read_csv("training.1600000.processed.noemoticon.csv", encoding="latin", header=None, names=["emotion", "id", "date"
```

Посмотрим на данные:

```
data.head()
```

Выведем несколько примеров твитов, чтобы понимать, с чем мы имеем дело:

```
examples = data["text"].sample(10)
print("\n".join(examples))
```

Как вилим, тексты твитов очень "грязные". Нужно предобработать датасет, прежде чем строить для него модель классификации.

Чтобы сравнивать различные методы обработки текста/модели/прочее, разделим датасет на dev(для обучения модели) и test(для получения качества модели).

```
indexes = np.arange(data.shape[0])
np.random.shuffle(indexes)
dev_size = math.ceil(data.shape[0] * 0.8)

dev_indexes = indexes[:dev_size]
test_indexes = indexes[dev_size:]

dev_data = data.iloc[dev_indexes]
test_data = data.iloc[test_indexes]

dev_data.reset_index(drop=True, inplace=True)
test_data.reset_index(drop=True, inplace=True)
```

Обработка текста

Токенизируем текст, избавимся от знаков пунктуации и выкинем все слова, состоящие менее чем из 4 букв:

```
tokenizer = nltk.WordPunctTokenizer()
line = tokenizer.tokenize(dev_data["text"][0].lower())
print(" ".join(line))
```

```
filtered_line = [w for w in line if all(c not in string.punctuation for c in w) and len(w) > 3]
print(" ".join(filtered_line))
```

Загрузим предобученную модель эмбеддингов.

Если хотите, можно попробовать другую. Полный список можно найти здесь: https://github.com/RaRe-Technologies/gensim-data.

Данная модель выдает эмбеддинги для **слов**. Строить по эмбеддингам слов эмбеддинги предложений мы будем ниже.

```
word2vec = api.load("word2vec-google-news-300")
```

```
emb_line = [word2vec.get_vector(w) for w in filtered_line if w in word2vec]
print(sum(emb_line).shape)
```

Нормализуем эмбеддинги, прежде чем обучать на них сеть. (наверное, вы помните, что нейронные сети гораздо лучше обучаются на нормализованных данных)

```
mean = np.mean(word2vec.vectors, 0)
std = np.std(word2vec.vectors, 0)
norm_emb_line = [(word2vec.get_vector(w) - mean) / std for w in filtered_line if w in word2vec and len(w) > 3]
print(sum(norm_emb_line).shape)
print([all(norm_emb_line[i] == emb_line[i]) for i in range(len(emb_line))])
```

Сделаем датасет, который будет по запросу возвращать подготовленные данные.

```
from torch.utils.data import Dataset, random_split
```

```
class TwitterDataset(Dataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.models.Word2Vec):
        self.tokenizer = nltk.WordPunctTokenizer()

        self.data = data

        self.feature_column = feature_column
        self.target_column = target_column

        self.word2vec = word2vec

        self.label2num = lambda label: 0 if label == 0 else 1
        self.mean = np.mean(word2vec.vectors, axis=0)
```

```
        self.std = np.std(word2vec.vectors, axis=0)

    def __getitem__(self, item):
        text = self.data[self.feature_column][item]
        label = self.label2num(self.data[self.target_column][item])

        tokens = self.get_tokens_(text)
        embeddings = self.get_embeddings_(tokens)

        return {"feature": embeddings, "target": label}

    def get_tokens_(self, text):
        # Получи все токены из текста и профильтруй их
        tokenizer = nltk.WordPunctTokenizer()
        line = tokenizer.tokenize(text.lower())
        filtered_line = [w for w in line if all(c not in string.punctuation for c in w) and len(w) > 3]
        return filtered_line

    def get_embeddings_(self, tokens):
        emb_line = [self.word2vec.get_vector(w) for w in tokens if w in self.word2vec]
        mean = np.mean(self.word2vec.vectors, 0)
        std = np.std(self.word2vec.vectors, 0)
        norm_emb_line = [(self.word2vec.get_vector(w) - mean) / std for w in tokens if w in self.word2vec and len(w) > 3]
        embeddings = norm_emb_line # Получи эмбеддинги слов и усредни их

        if len(embeddings) == 0:
            embeddings = np.zeros((1, self.word2vec.vector_size))
        else:
            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)

        return embeddings

    def __len__(self):
        return self.data.shape[0]

dev = TwitterDataset(dev_data, "text", "emotion", word2vec)
```

Отлично, мы готовы с помощью эмбеддингов слов превращать твиты в векторы и обучать нейронную сеть.

Превращать твиты в векторы, используя эмбеддинги слов, можно несколькими способами. А именно такими:

Average embedding (2 балла)

Это самый простой вариант, как получить вектор предложения, используя векторные представления слов в предложении. А именно: вектор предложения есть средний вектор всех слов в предлоежнии (которые остались после токенизации и удаления коротких слов, конечно).

```
indexes = np.arange(len(dev))
np.random.shuffle(indexes)
example_indexes = indexes[::1000]

examples = {"features": [np.sum(dev[i]["feature"], axis=0) for i in example_indexes],
            "targets": [dev[i]["target"] for i in example_indexes]}
print(len(examples["features"]))
```

Давайте сделаем визуализацию полученных векторов твитов тренировочного (dev) датасета. Так мы увидим, насколько хорошо твиты с разными target значениями отделяются друг от друга, т.е. насколько хорошо усреднение эмбеддингов слов предложения передает информацию о предложении.

Для визуализации векторов надо получить их проекцию на плоскость. Сделаем это с помощью `PCA`. Если хотите, можете вместо PCA использовать TSNE: так у вас получится более точная проекция на плоскость (а значит, более информативная, т.е. отражающая реальное положение векторов твитов в пространстве). Но TSNE будет работать намного дольше.

```
from sklearn.decomposition import PCA


pca = PCA(n_components=2)
examples["transformed_features"] = pca.fit(examples["features"]) # Обучи PCA на эмбеддингах слов
```

```
import bokeh.models as bm, bokeh.plotting as pl
from bokeh.io import output_notebook
output_notebook()

def draw_vectors(x, y, radius=10, alpha=0.25, color='blue',
                 width=600, height=400, show=True, **kwargs):
    """ draws an interactive plot for data points with auxilirary info on hover """
    data_source = bm.ColumnDataSource({ 'x' : x, 'y' : y, 'color': color, **kwargs })

    fig = pl.figure(active_scroll='wheel_zoom', width=width, height=height)
    fig.scatter('x', 'y', size=radius, color='color', alpha=alpha, source=data_source)

    fig.add_tools(bm.HoverTool(tooltips=[(key, "@" + key) for key in kwargs.keys()]))
    if show: pl.show(fig)
    return fig


draw_vectors(
    examples["transformed_features"][:, 0],
    examples["transformed_features"][:, 1],
    color=[["red", "blue"][t] for t in examples["targets"]]
    )
```

Скорее всего, на визуализации нет четкого разделения твитов между классами. Это значит, что по полученным нами векторам твитов не так-то просто определить, к какому классу твит пренадлежит. Значит, обычный линейный классификатор не очень хорошо справится с задачей. Надо будет делать глубокую (хотя бы два слоя) нейронную сеть.

Подготовим загрузчики данных. Усреднее векторов будем делать в "батчевалке"(`collate_fn`). Она используется для того, чтобы собирать из данных `torch.Tensor` батчи, которые можно отправлять в модель.

```
from torch.utils.data import DataLoader


batch_size = 1024
num_workers = 4

def average_emb(batch):
    features = [np.mean(b["feature"], axis=0) for b in batch]
    targets = [b["target"] for b in batch]

    return {"features": torch.FloatTensor(features), "targets": torch.LongTensor(targets)}


train_size = math.ceil(len(dev) * 0.8)

train, valid = random_split(dev, [train_size, len(dev) - train_size])

train_loader = DataLoader(train, batch_size=batch_size, num_workers=num_workers, shuffle=True, drop_last=True, collate_fn=ave
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_fn=a
```

Определим функции для тренировки и теста модели:

```
from tqdm.notebook import tqdm

# https://pytorch.org/tutorials/beginner/introyt/trainingyt.html
def training(model, optimizer, criterion, train_loader, epoch, device="cpu"):
    pbar = tqdm(train_loader, desc=f"Epoch {e + 1}. Train Loss: {0}")
    model.train()
    for batch in pbar:
        features = batch["features"].to(device)
        targets = batch["targets"].to(device)

        optimizer.zero_grad()

        # Получи предсказания модели
        outputs = model(features)

        loss = criterion(outputs, targets) # Посчитай лосс
        # Обнови параметры модели
        loss.backward()

        optimizer.step()

        pbar.set_description(f"Epoch {e + 1}. Train Loss: {loss:.4}")

# https://discuss.pytorch.org/t/testing-in-loop-as-training/70881
def testing(model, criterion, test_loader, device="cpu"):
    pbar = tqdm(test_loader, desc=f"Test Loss: {0}, Test Acc: {0}")
```

```
    mean_loss = 0
    mean_acc = 0
    model.eval()
    with torch.no_grad():
        for batch in pbar:
            features = batch["features"].to(device)
            targets = batch["targets"].to(device)

            # Получи предсказания модели
            outputs = model(features)

            loss = criterion(outputs, targets) # Посчитай лосс

            # acc = ... # Посчитай точность модели

            _, predicted = torch.max(outputs, 1)
            correct = (predicted == targets).sum().item()
            acc = ((correct / len(targets)) * 100)

            mean_loss += loss.item()
            mean_acc += acc.item()

            pbar.set_description(f"Test Loss: {loss:.4}, Test Acc: {acc:.4}")

    pbar.set_description(f"Test Loss: {mean_loss / len(test_loader):.4}, Test Acc: {mean_acc / len(test_loader):.4}")

    return {"Test Loss": mean_loss / len(test_loader), "Test Acc": mean_acc / len(test_loader)}
```

Создадим модель, оптимизатор и целевую функцию. Вы можете сами выбрать количество слоев в нейронной сети, ваш любимый оптимизатор и целевую функцию.

```
class W2VModel(nn.Module):
    def __init__(self, voc_size, emb_dim):
        super().__init__()
        self.encoder = nn.Embedding(voc_size, emb_dim)
        self.decoder = nn.Linear(emb_dim, voc_size, bias=False)
        self.voc_size = voc_size
        self.emb_dim = emb_dim
        self.init_emb()

    def forward(self, word):
        return self.decoder(self.encoder(word))

    def init_emb(self):
        """
        init the weight as original word2vec do.
        """
        initrange = 0.5 / self.emb_dim
        self.encoder.weight.data.uniform_(-initrange, initrange)
        self.decoder.weight.data.uniform_(0, 0)
```

```
import torch.nn as nn
from torch.optim import Adam
```

```
# Не забудь поиграться с параметрами ;)
vector_size = dev.word2vec.vector_size
num_classes = 2
lr = 1e-2
num_epochs = 1

model = W2VModel(vector_size, 300) # Твоя модель
model = model.cuda()
criterion = torch.nn.CrossEntropyLoss() # Твой лосс
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9) # Твой оптимайзер
```

Наконец, обучим модель и протестируем её.

После каждой эпохи будем проверять качество модели на валидационной части датасета. Если метрика стала лучше, будем сохранять модель. **Подумайте, какая метрика (точность или лосс) будет лучше работать в этой задаче?**

```
best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader, e, device)
    log = testing(model, criterion, valid_loader, device)
    print(log)
    if log["Test Loss"] < best_metric:
```

```
        torch.save(model.state_dict(), "model.pt")
        best_metric = log["Test Loss"]


test_loader = DataLoader(
    TwitterDataset(test_data, "text", "emotion", word2vec),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model.pt", map_location=device))

print(testing(model, criterion, test_loader, device=device))
```

Embeddings for unknown words (8 баллов)

Пока что использовалась не вся информация из текста. Часть информации фильтровалось – если слова не было в словаре эмбеддингов, то мы просто превращали слово в нулевой вектор. Хочется использовать информацию по-максимуму. Поэтому рассмотрим другие способы обработки слов, которых нет в словаре. А именно:

- Для каждого незнакомого слова будем запоминать его контекст(слова слева и справа от этого слова). Эмбеддингом нашего незнакомого слова будет сумма эмбеддингов всех слов из его контекста. (4 балла)
- Для каждого слова текста получим его эмбеддинг из Tfidf с помощью `TfidfVectorizer` из sklearn. Итоговым эмбеддингом для каждого слова будет сумма двух эмбеддингов: предобученного и Tfidf-ного. Для слов, которых нет в словаре предобученных эмбеддингов, результирующий эмбеддинг будет просто полученный из Tfidf. (4 балла)

Реализуйте оба варианта **ниже**. Напишите, какой способ сработал лучше и ваши мысли, почему так получилось.

```
# 1
class TwitterDatasetUnknownTokens(TwitterDataset):
    def get_embeddings_(self, tokens, context_size=4):
        # Получи эмбеддинги слов и усредни их
        embeddings = []
        for i, token in enumerate(tokens):
            try:
                vector = word2vec.get_vector(token)
                embeddings.append(vector)
            except KeyError:
                context = []
                for j in range(max(0, i - context_size), min(i + context_size, len(tokens))):
                    if j == i:
                        continue

                    try:
                        context.append(word2vec.get_vector(tokens[j]))
                    except KeyError:
                        pass
                if context:
                  vector = np.mean(context, axis=0)
                  embeddings.append(vector)

        if len(embeddings) == 0:
            embeddings = np.zeros((1, self.word2vec.vector_size))
        else:
            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)
        return embeddings


dev_context = TwitterDatasetUnknownTokens(dev_data, "text", "emotion", word2vec)
train, valid = random_split(dev_context, [train_size, len(dev_context) - train_size])

train_loader = DataLoader(train, batch_size=batch_size, num_workers=num_workers, shuffle=True, drop_last=True, collate_fn=ave
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_fn=a

model = nn.Sequential(
          nn.Linear(vector_size, 600),
          nn.ReLU(),
          nn.Linear(600, 300),
          nn.ReLU(),
          nn.Linear(300, 100),
          nn.ReLU(),
          nn.Linear(100, num_classes),
        ) # Твоя модель
model = model.cuda()
```

```
criterion = nn.CrossEntropyLoss() # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)# Твой оптимайзер


best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader, e, device)
    log = testing(model, criterion, valid_loader, device)
    print(log)
    if log["Test Loss"] < best_metric:
        torch.save(model.state_dict(), "model.pt")
        best_metric = log["Test Loss"]


test_loader = DataLoader(
    TwitterDatasetUnknownTokens(test_data, "text", "emotion", word2vec),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model.pt", map_location=device))

print(testing(model, criterion, test_loader, device=device))


# 2
from sklearn.feature_extraction.text import TfidfVectorizer


class TwitterDatasetTfidf(TwitterDataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.models.Word2Vec):
        self.tokenizer = nltk.WordPunctTokenizer()

        self.data = data

        self.feature_column = feature_column
        self.target_column = target_column

        self.word2vec = word2vec
        self.tfidf = TfidfVectorizer(max_features=word2vec.vector_size).fit(dev_data['text'])
        self.label2num = lambda label: 0 if label == 0 else 1
        self.mean = np.mean(word2vec.vectors, axis=0)
        self.std = np.std(word2vec.vectors, axis=0)

    def get_embeddings_(self, tokens, context_size=4):
        # Получи эмбеддинги слов и усредни их
        embeddings = []
        if len(tokens) > 0:
          tfidf_array = self.tfidf.transform(tokens).toarray()
        for i, token in enumerate(tokens):
            try:
                vector = word2vec.get_vector(token)
            except KeyError:
                vector = None

            if vector is not None:
                embeddings.append(vector + tfidf_array[i])
            else:
                embeddings.append(tfidf_array[i])

        if len(embeddings) == 0:
            embeddings = np.zeros((1, self.word2vec.vector_size))
        else:
            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)

        return embeddings



dev_context = TwitterDatasetTfidf(dev_data, "text", "emotion", word2vec)
train, valid = random_split(dev_context, [train_size, len(dev_context) - train_size])

train_loader = DataLoader(train, batch_size=batch_size, num_workers=4, shuffle=True, drop_last=True, collate_fn=average_emb)
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=4, shuffle=False, drop_last=False, collate_fn=average_emb

model = nn.Sequential(
          nn.Linear(vector_size, 600),
          nn.ReLU(),
          nn.Linear(600, 300),
          nn.ReLU(),
```

```
            nn.Linear(300, 100),
            nn.ReLU(),
            nn.Linear(100, num_classes),
        ) # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss() # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)# Твой оптимайзер


best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader, e, device)
    log = testing(model, criterion, valid_loader, device)
    print(log)
    if log["Test Loss"] < best_metric:
        torch.save(model.state_dict(), "model.pt")
        best_metric = log["Test Loss"]


test_loader = DataLoader(
    TwitterDatasetTfidf(test_data, "text", "emotion", word2vec),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model.pt", map_location=device))

print(testing(model, criterion, test_loader, device=device))


# первый вариант чуть лучше по сравнению с tf-idf, потому что 2й вариант вектор слова предоставляет больше смысла для модели
```