Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Задание 3

Классификация текстов

В этом задании вам предстоит попробовать несколько методов, используемых в задаче классификации, а также понять насколько хорошо модель понимает смысл слов и какие слова в примере влияют на результат.

```
import pandas as pd
import numpy as np
import torch

from torchtext.legacy import datasets

from torchtext.legacy.data import Field, LabelField
from torchtext.legacy.data import BucketIterator

from torchtext.vocab import Vectors, GloVe

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import random
from tqdm.autonotebook import tqdm
```

В этом задании мы будем использовать библиотеку torchtext. Она довольна проста в использовании и поможет нам сконцентрироваться на задаче, а не на написании Dataloader-а.

```
TEXT = Field(sequential=True, lower=True, include_lengths=True)  # Поле текста
LABEL = LabelField(dtype=torch.float)  # Поле метки
```

```
SEED = 1234

np.random.seed(SEED)
torch.manual_seed(SEED)
torch.random.manual_seed(SEED)
torch.cuda.random.manual_seed_all(SEED)
torch.backends.cudnn.deterministic = True
```

```
def calc_f1_score(model, iter, output_calc_func):
    model.eval()

    tp = 0.0
    tn = 0.0
    fp = 0.0
    fn = 0.0
    for batch in iter:
        with torch.no_grad():
            outputs = output_calc_func(model, batch)
            outputs = (torch.sigmoid(outputs).cpu() > 0.5).int().squeeze(1)
            labels = batch.label.cpu().int()

            tp += (labels * outputs).sum()
            tn += ((1 - labels) * (1 - outputs)).sum()
            fp += ((1 - labels) * outputs).sum()
            fn += (labels * (1 - outputs)).sum()

    epsilon = 1e-7
    precision = tp / (tp + fp + epsilon)
    recall = tp / (tp + fn + epsilon)

    f1_score = 2 * (precision * recall) / (precision + recall + epsilon)
    f1_score = f1_score.item()

    return f1_score
```

Напишем функции для обучения моделей

```
def train_rnn(model, loss_func, train_iter, val_iter, max_epochs, patience, max_grad_norm=2):
    min_loss = np.inf

    cur_patience = 0

    for epoch in range(1, max_epochs + 1):
        train_loss = 0.0
        model.train()
        pbar = tqdm(enumerate(train_iter), total=len(train_iter), leave=False)
        pbar.set_description(f"Epoch {epoch}")
        for it, batch in pbar:
            opt.zero_grad()

            outputs = model(batch.text[0], batch.text[1].cpu())
            loss = loss_func(outputs, batch.label.unsqueeze(1))
            loss.backward()
            if max_grad_norm is not None:
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
            opt.step()

            train_loss += loss.cpu().detach()
```

```
        train_loss /= len(train_iter)
        val_loss = 0.0
        model.eval()
        pbar = tqdm(enumerate(val_iter), total=len(val_iter), leave=False)
        pbar.set_description(f"Epoch {epoch}")
        for it, batch in pbar:
            with torch.no_grad():
                outputs = model(batch.text[0], batch.text[1].cpu())
                loss = loss_func(outputs, batch.label.unsqueeze(1)).cpu()

                val_loss += loss

        val_loss /= len(val_iter)
        if val_loss < min_loss:
            min_loss = val_loss
            best_model = model.state_dict()
        else:
            cur_patience += 1
            if cur_patience == patience:
                cur_patience = 0
                break

        print('Epoch: {}, Training Loss: {}, Validation Loss: {}'.format(epoch, train_loss, val_loss))
    model.load_state_dict(best_model)

def freeze_embeddings(model, req_grad=False):
    for c_p in model.embedding.parameters():
        c_p.requires_grad = req_grad

def train_cnn(model, loss_func, train_iter, val_iter, max_epochs, patience, max_grad_norm=2, num_freeze_epochs=0):
    min_loss = np.inf

    cur_patience = 0

    freeze_embeddings(model)

    for epoch in range(1, max_epochs + 1):
        train_loss = 0.0
        model.train()
        pbar = tqdm(enumerate(train_iter), total=len(train_iter), leave=False)
        pbar.set_description(f"Epoch {epoch}")

        if epoch > num_freeze_epochs:
            freeze_embeddings(model, True)

        for it, batch in pbar:
            opt.zero_grad()

            outputs = model(batch.text)
            loss = loss_func(outputs, batch.label.unsqueeze(1))
            loss.backward()
            if max_grad_norm is not None:
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
            opt.step()

            train_loss += loss.cpu().detach()

        train_loss /= len(train_iter)
        val_loss = 0.0
        model.eval()
        pbar = tqdm(enumerate(val_iter), total=len(val_iter), leave=False)
        pbar.set_description(f"Epoch {epoch}")
        for it, batch in pbar:
            with torch.no_grad():
                outputs = model(batch.text)
                loss = loss_func(outputs, batch.label.unsqueeze(1)).cpu()

                val_loss += loss

        val_loss /= len(val_iter)
        if val_loss < min_loss:
            min_loss = val_loss
            best_model = model.state_dict()
        else:
            cur_patience += 1
            if cur_patience == patience:
                cur_patience = 0
                break

        print('Epoch: {}, Training Loss: {}, Validation Loss: {}'.format(epoch, train_loss, val_loss))
    model.load_state_dict(best_model)
```

Датасет на котором мы будем проводить эксперементы это комментарии к фильмам из сайта IMDB.

```
train, test = datasets.IMDB.splits(TEXT, LABEL)  # загрузим датасет
train, valid = train.split(random_state=random.seed(SEED))  # разобьем на части
```

```
TEXT.build_vocab(train)
LABEL.build_vocab(train)
```

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

train_iter, valid_iter, test_iter = BucketIterator.splits(
    (train, valid, test),
    batch_size = 64,
    sort_within_batch = True,
    device = device)
```

RNN

Для начала попробуем использовать рекурентные нейронные сети. На семинаре вы познакомились с GRU, вы можете также попробовать LSTM. Можно использовать для классификации как hidden_state, так и output последнего токена.

```
class RNNBaseline(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)

        self.rnn = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=

        self.dropout = nn.Dropout(dropout)

        self.fc = nn.Linear(2 * hidden_dim, output_dim)


    def forward(self, text, text_lengths):

        #text = [sent len, batch size]

        embedded = self.embedding(text)
```

```
        #embedded = [sent len, batch size, emb dim]

        #pack sequence
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

        # cell arg for LSTM, remove for GRU
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        #unpack sequence
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)

        #output = [sent len, batch size, hid dim * num directions]
        #output over padding tokens are zero tensors

        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]

        #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
        #and apply dropout

        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), 1)
        hidden = self.dropout(hidden)

        #hidden = [batch size, hid dim * num directions] or [batch_size, hid dim * num directions]

        return self.fc(hidden)
```

Поиграйтесь с гиперпараметрами

```
vocab_size = len(TEXT.vocab)
emb_dim = 300
hidden_dim = 256
output_dim = 1
n_layers = 2
bidirectional = True
dropout = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

rnn_model = RNNBaseline(
    vocab_size=vocab_size,
    embedding_dim=emb_dim,
    hidden_dim=hidden_dim,
    output_dim=output_dim,
    n_layers=n_layers,
    bidirectional=bidirectional,
    dropout=dropout,
    pad_idx=PAD_IDX
).to(device)
```

```
opt = torch.optim.Adam(rnn_model.parameters(), lr=1e-4)
loss_func = nn.BCEWithLogitsLoss()
```

Обучите сетку! Используйте любые вам удобные инструменты, Catalyst, PyTorch Lightning или свои велосипеды.

```
max_epochs = 50
patience=15

train_rnn(rnn_model, loss_func, train_iter, valid_iter, max_epochs, patience)
```

```
Epoch: 1, Training Loss: 0.6742545366287231, Validation Loss: 0.5824978351593018
Epoch: 2, Training Loss: 0.5109237432479858, Validation Loss: 0.5463767051696777
Epoch: 3, Training Loss: 0.4130497872829437, Validation Loss: 0.4198527932167053
Epoch: 4, Training Loss: 0.3480762541294098, Validation Loss: 0.42421337962150574
Epoch: 5, Training Loss: 0.2857218384742737, Validation Loss: 0.418121874332428
Epoch: 6, Training Loss: 0.24492843449115753, Validation Loss: 0.4250085651874542
Epoch: 7, Training Loss: 0.19758240878582, Validation Loss: 0.4353417754173279
Epoch: 8, Training Loss: 0.15108461678028107, Validation Loss: 0.46530306339263916
Epoch: 9, Training Loss: 0.12880533933639526, Validation Loss: 0.5643097758293152
Epoch: 10, Training Loss: 0.09997963905334473, Validation Loss: 0.6360642313957214
Epoch: 11, Training Loss: 0.0769193023443222, Validation Loss: 0.5590224266052246
Epoch: 12, Training Loss: 0.06668514758348465, Validation Loss: 0.6920998692512512
Epoch: 13, Training Loss: 0.048860158771276474, Validation Loss: 0.6697909832000732
Epoch: 14, Training Loss: 0.041297007352113724, Validation Loss: 0.7189866900444031
Epoch: 15, Training Loss: 0.03422073647379875, Validation Loss: 0.7576189041137695
Epoch: 16, Training Loss: 0.03396064043045044, Validation Loss: 0.8378239274024963
Epoch: 17, Training Loss: 0.022049326449632645, Validation Loss: 0.8449418544769287
Epoch: 18, Training Loss: 0.015442287549376488, Validation Loss: 0.8026885390281677
```

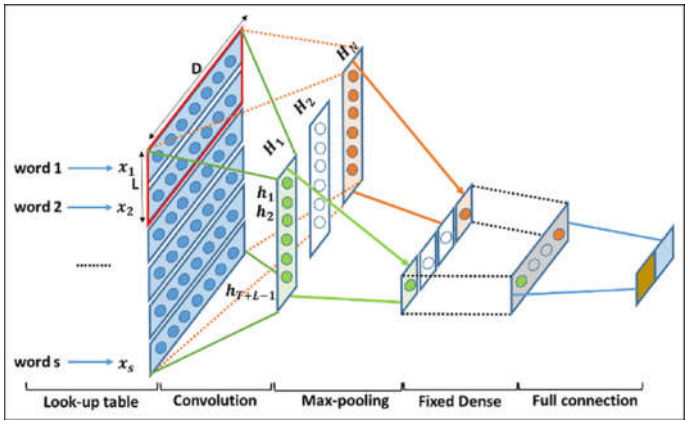
Посчитайте f1-score вашего классификатора на тестовом датасете.

Ответ:

```
calc_f1_score(rnn_model, test_iter, lambda model, batch: model(batch.text[0], batch.text[1].cpu()))
```

```
0.8374814987182617
```

CNN



Для классификации текстов также часто используют сверточные нейронные сети. Идея в том, что как правило сентимент содержат словосочетания из двух-трех слов, например "очень хороший фильм" или "невероятная скука". Проходясь сверткой по этим словам мы получим какой-то большой скор и выхватим его с помощью MaxPool. Далее идет обычная полносвязная сетка. Важный момент: свертки применяются не последовательно, а параллельно. Давайте попробуем!

```
TEXT = Field(sequential=True, lower=True, batch_first=True)  # batch_first тк мы используем conv
LABEL = LabelField(batch_first=True, dtype=torch.float)
```

```
train, tst = datasets.IMDB.splits(TEXT, LABEL)
trn, vld = train.split(random_state=random.seed(SEED))

TEXT.build_vocab(trn)
LABEL.build_vocab(trn)

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

```
train_iter, val_iter, test_iter = BucketIterator.splits(
        (trn, vld, tst),
        batch_sizes=(128, 256, 256),
        sort=False,
        sort_key= lambda x: len(x.src),
        sort_within_batch=False,
        device=device,
        repeat=False,
)
```

Вы можете использовать Conv2d с `in_channels=1, kernel_size=(kernel_sizes[0], emb_dim))` или Conv1d с `in_channels=emb_dim, kernel_size=kernel_size[0]`. Но хорошенько подумайте над shape в обоих случаях.

```
class CNN(nn.Module):
    def __init__(
        self,
        vocab_size,
        emb_dim,
        out_channels,
        kernel_sizes,
        dropout=0.5,
    ):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.conv_0 = nn.Conv1d(in_channels=emb_dim, out_channels=out_channels, kernel_size=kernel_sizes[0])

        self.conv_1 = nn.Conv1d(in_channels=emb_dim, out_channels=out_channels, kernel_size=kernel_sizes[1])

        self.conv_2 = nn.Conv1d(in_channels=emb_dim, out_channels=out_channels, kernel_size=kernel_sizes[2])

        self.fc = nn.Linear(len(kernel_sizes) * out_channels, 1)

        self.dropout = nn.Dropout(dropout)


    def forward(self, text):
        embedded = self.embedding(text)

        embedded = embedded.permute(0, 2, 1)

        conved_0 = F.relu(self.conv_0(embedded))  # may be reshape here
        conved_1 = F.relu(self.conv_1(embedded))  # may be reshape here
        conved_2 = F.relu(self.conv_2(embedded))  # may be reshape here

        pooled_0 = F.max_pool1d(conved_0, conved_0.shape[2]).squeeze(2)
        pooled_1 = F.max_pool1d(conved_1, conved_1.shape[2]).squeeze(2)
        pooled_2 = F.max_pool1d(conved_2, conved_2.shape[2]).squeeze(2)

        cat = self.dropout(torch.cat((pooled_0, pooled_1, pooled_2), dim=1))

        return self.fc(cat)
```

```
kernel_sizes = [3, 4, 5]
vocab_size = len(TEXT.vocab)
out_channels=64
dropout = 0.5
dim = 300

cnn_model = CNN(vocab_size=vocab_size, emb_dim=dim, out_channels=out_channels, kernel_sizes=kernel_sizes, dropout=dropout).to(device)
```

```
opt = torch.optim.Adam(cnn_model.parameters(), lr=1e-4)
loss_func = nn.BCEWithLogitsLoss()
```

Обучите!

```
max_epochs = 50
patience = 15

train_cnn(cnn_model, loss_func, train_iter, val_iter, max_epochs, patience)
```

```
Epoch: 1, Training Loss: 0.7576324343681335, Validation Loss: 0.6517045497894287
Epoch: 2, Training Loss: 0.6899340748786926, Validation Loss: 0.6062830686569214
Epoch: 3, Training Loss: 0.6387899518013, Validation Loss: 0.5613199472427368
Epoch: 4, Training Loss: 0.5910921692848206, Validation Loss: 0.5337907075881958
```

Посчитайте f1-score вашего классификатора.

Ответ:

```
Epoch: 9, Training Loss: 0.4728925824165344, Validation Loss: 0.46500885486602783
```

```
calc_f1_score(cnn_model, test_iter, lambda model, batch: model(batch.text))
```

```
0.8496467471122742
Epoch: 14, Training Loss: 0.410084068775177, Validation Loss: 0.43029505014419556
```

Интерпретируемость

Посмотрим, куда смотрит наша модель. Достаточно запустить код ниже.

```
!pip install -q captum
```

```
from captum.attr import LayerIntegratedGradients, TokenReferenceBase, visualization

PAD_IND = TEXT.vocab.stoi['pad']

token_reference = TokenReferenceBase(reference_token_idx=PAD_IND)
lig = LayerIntegratedGradients(cnn_model, cnn_model.embedding)
```

```
Epoch: 32, Training Loss: 0.26171180605888367, Validation Loss: 0.35848888754844666
```

```
def forward_with_softmax(model, inp):
    logits = model(inp)
    return torch.softmax(logits, 0)[0][1]

def forward_with_sigmoid(model, input):
    return torch.sigmoid(model(input))


# accumalate couple samples in this array for visualization purposes
vis_data_records_ig = []

def interpret_sentence(model, sentence, min_len = 7, label = 0):
    model.eval()
    text = [tok for tok in TEXT.tokenize(sentence)]
    if len(text) < min_len:
        text += ['pad'] * (min_len - len(text))
    indexed = [TEXT.vocab.stoi[t] for t in text]

    model.zero_grad()

    input_indices = torch.tensor(indexed, device=device)
    input_indices = input_indices.unsqueeze(0)

    # input_indices dim: [sequence_length]
    seq_length = min_len

    # predict
    pred = forward_with_sigmoid(model, input_indices).item()
    pred_ind = round(pred)

    # generate reference indices for each sample
    reference_indices = token_reference.generate_reference(seq_length, device=device).unsqueeze(0)

    # compute attributions and approximation delta using layer integrated gradients
    attributions_ig, delta = lig.attribute(input_indices, reference_indices, \
                                           n_steps=5000, return_convergence_delta=True)

    print('pred: ', LABEL.vocab.itos[pred_ind], '(', '%.2f'%pred, ')', ', delta: ', abs(delta))

    add_attributions_to_visualizer(attributions_ig, text, pred, pred_ind, label, delta, vis_data_records_ig)

def add_attributions_to_visualizer(attributions, text, pred, pred_ind, label, delta, vis_data_records):
    attributions = attributions.sum(dim=2).squeeze(0)
    attributions = attributions / torch.norm(attributions)
    attributions = attributions.cpu().detach().numpy()

    # storing couple samples in an array for visualization purposes
    vis_data_records.append(visualization.VisualizationDataRecord(
                            attributions,
                            pred,
                            LABEL.vocab.itos[pred_ind],
                            LABEL.vocab.itos[label],
                            LABEL.vocab.itos[1],
                            attributions.sum(),
                            text,
                            delta))
```

```
interpret_sentence(cnn_model, 'It was a fantastic performance !', label=1)
interpret_sentence(cnn_model, 'Best film ever', label=1)
interpret_sentence(cnn_model, 'Such a great show!', label=1)
interpret_sentence(cnn_model, 'It was a horrible movie', label=0)
interpret_sentence(cnn_model, 'I\'ve never watched something as bad', label=0)
interpret_sentence(cnn_model, 'It is a disgusting movie!', label=0)
```

```
pred:  pos ( 0.99 ) , delta:  tensor([0.0001], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.30 ) , delta:  tensor([7.6005e-06], device='cuda:0', dtype=torch.float64)
pred:  pos ( 0.96 ) , delta:  tensor([1.3040e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.09 ) , delta:  tensor([3.4205e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.12 ) , delta:  tensor([7.6346e-05], device='cuda:0', dtype=torch.float64)
pred:  pos ( 0.69 ) , delta:  tensor([1.7903e-05], device='cuda:0', dtype=torch.float64)
```

Попробуйте добавить свои примеры!

```
print('Visualize attributions based on Integrated Gradients')
visualization.visualize_text(vis_data_records_ig)
```

```
Visualize attributions based on Integrated Gradients
```

Legend: ☐ Negative ☐ Neutral ☐ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
neg	neg (0.09)	pos	-0.45	It was a horrible movie pad pad

Эмбэдинги слов

Вы ведь не забыли, как мы можем применить знания о word2vec и GloVe. Давайте попробуем!

```
TEXT.build_vocab(trn, vectors='glove.840B.300d')
LABEL.build_vocab(trn)

word_embeddings = TEXT.vocab.vectors
```

```
train, tst = datasets.IMDB.splits(TEXT, LABEL)
trn, vld = train.split(random_state=random.seed(SEED))

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

train_iter, val_iter, test_iter = BucketIterator.splits(
        (trn, vld, tst),
        batch_sizes=(128, 256, 256),
        sort=False,
        sort_key= lambda x: len(x.src),
        sort_within_batch=False,
        device=device,
        repeat=False,
)
```

```
kernel_sizes = [3, 4, 5]
vocab_size = len(TEXT.vocab)
out_channels=64
dropout = 0.5
dim = 300

cnn_and_embeddings_model = CNN(vocab_size=vocab_size, emb_dim=dim, out_channels=out_channels, kernel_sizes=kernel_sizes, dropout=dropo

word_embeddings = TEXT.vocab.vectors

prev_shape = cnn_and_embeddings_model.embedding.weight.shape

cnn_and_embeddings_model.embedding.weight.data = torch.clone(word_embeddings)

assert prev_shape == cnn_and_embeddings_model.embedding.weight.shape
cnn_and_embeddings_model.to(device)
```

```
CNN(
  (embedding): Embedding(202268, 300)
  (conv_0): Conv1d(300, 64, kernel_size=(3,), stride=(1,))
  (conv_1): Conv1d(300, 64, kernel_size=(4,), stride=(1,))
  (conv_2): Conv1d(300, 64, kernel_size=(5,), stride=(1,))
  (fc): Linear(in_features=192, out_features=1, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

```
opt = torch.optim.Adam(cnn_and_embeddings_model.parameters(), lr=1e-4)
loss_func = nn.BCEWithLogitsLoss()
```

Вы знаете, что делать.

```
max_epochs = 50
patience = 15

train_cnn(cnn_and_embeddings_model, loss_func, train_iter, val_iter, max_epochs, patience, num_freeze_epochs=0)
```

```
Epoch: 1, Training Loss: 0.6762555241584778, Validation Loss: 0.6392307877540588
Epoch: 2, Training Loss: 0.6033423542976379, Validation Loss: 0.5459020733833313
Epoch: 3, Training Loss: 0.49749210476875305, Validation Loss: 0.4511328339576721
Epoch: 4, Training Loss: 0.42663270235061646, Validation Loss: 0.40656739473342896
Epoch: 5, Training Loss: 0.3887517142167664, Validation Loss: 0.38155120611190796
Epoch: 6, Training Loss: 0.362044245004653, Validation Loss: 0.3653334379196167
Epoch: 7, Training Loss: 0.33915168046951294, Validation Loss: 0.353719709083557
Epoch: 8, Training Loss: 0.31842857599258423, Validation Loss: 0.3433770537376404
Epoch: 9, Training Loss: 0.30003029108047485, Validation Loss: 0.33546820282936096
Epoch: 10, Training Loss: 0.28027257323265076, Validation Loss: 0.3285302519798279
Epoch: 11, Training Loss: 0.26262593269348145, Validation Loss: 0.32217657566070557
Epoch: 12, Training Loss: 0.2462833672761917, Validation Loss: 0.3174632489681244
Epoch: 13, Training Loss: 0.23081234097480774, Validation Loss: 0.31211230158805847
Epoch: 14, Training Loss: 0.2127387821674347, Validation Loss: 0.3088242709636688
Epoch: 15, Training Loss: 0.19626827538013458, Validation Loss: 0.30575281381607056
Epoch: 16, Training Loss: 0.18307611346244812, Validation Loss: 0.3018968999385834
Epoch: 17, Training Loss: 0.16972137987613678, Validation Loss: 0.2993110418319702
Epoch: 18, Training Loss: 0.15460239350795746, Validation Loss: 0.29740700125694275
Epoch: 19, Training Loss: 0.14086130261421204, Validation Loss: 0.2963598966598511
Epoch: 20, Training Loss: 0.12892025709152222, Validation Loss: 0.29563918709754944
Epoch: 21, Training Loss: 0.11790939420461655, Validation Loss: 0.2952145040035248
Epoch: 22, Training Loss: 0.10530146211385727, Validation Loss: 0.2947268486022949
Epoch: 23, Training Loss: 0.09470847994089127, Validation Loss: 0.2952643036842346
Epoch: 24, Training Loss: 0.08577204495668411, Validation Loss: 0.29551824927330017
Epoch: 25, Training Loss: 0.0754757970571518, Validation Loss: 0.29775363206863403
Epoch: 26, Training Loss: 0.06945324689149857, Validation Loss: 0.2990273237228396
Epoch: 27, Training Loss: 0.06202911585569382, Validation Loss: 0.3005805015563965
Epoch: 28, Training Loss: 0.05581586062908172, Validation Loss: 0.3018071055412292
Epoch: 29, Training Loss: 0.04888837784528732, Validation Loss: 0.3048644661903381
Epoch: 30, Training Loss: 0.043837517499923706, Validation Loss: 0.3083876967430115
Epoch: 31, Training Loss: 0.038906726986169815, Validation Loss: 0.3083434104919433
Epoch: 32, Training Loss: 0.035062339156866074, Validation Loss: 0.31097838282585144
Epoch: 33, Training Loss: 0.030737191438674927, Validation Loss: 0.31603652238845825
Epoch: 34, Training Loss: 0.027629932388663292, Validation Loss: 0.31838181614875793
Epoch: 35, Training Loss: 0.025365378707647324, Validation Loss: 0.3211582899093628
Epoch: 36, Training Loss: 0.022064093500375748, Validation Loss: 0.32449501752853394
```

Посчитайте f1-score вашего классификатора.

Ответ:

```
calc_f1_score(cnn_and_embeddings_model, test_iter, lambda model, batch: model(batch.text))
```

```
0.872462272644043
```

Проверим насколько все хорошо!

```
PAD_IND = TEXT.vocab.stoi['pad']

token_reference = TokenReferenceBase(reference_token_idx=PAD_IND)
lig = LayerIntegratedGradients(cnn_and_embeddings_model, cnn_and_embeddings_model.embedding)
vis_data_records_ig = []

interpret_sentence(cnn_and_embeddings_model, 'It was a fantastic performance !', label=1)
interpret_sentence(cnn_and_embeddings_model, 'Best film ever', label=1)
interpret_sentence(cnn_and_embeddings_model, 'Such a great show!', label=1)
interpret_sentence(cnn_and_embeddings_model, 'It was a horrible movie', label=0)
interpret_sentence(cnn_and_embeddings_model, 'I\'ve never watched something as bad', label=0)
interpret_sentence(cnn_and_embeddings_model, 'It is a disgusting movie!', label=0)
```

```
pred:  pos ( 0.98 ) , delta:  tensor([0.0001], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.01 ) , delta:  tensor([2.5854e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.35 ) , delta:  tensor([0.0002], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.00 ) , delta:  tensor([5.4771e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.43 ) , delta:  tensor([0.0002], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.00 ) , delta:  tensor([1.0463e-05], device='cuda:0', dtype=torch.float64)
```

```
print('Visualize attributions based on Integrated Gradients')
visualization.visualize_text(vis_data_records_ig)
```

```
Visualize attributions based on Integrated Gradients
```

Legend: ☐ Negative ☐ Neutral ☐ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
pos	pos (0.98)	pos	1.85	It was a fantastic performance ! pad
pos	neg (0.01)	pos	1.12	Best film ever pad pad pad pad
pos	neg (0.35)	pos	1.57	Such a great show! pad pad pad
neg	neg (0.00)	pos	0.06	It was a horrible movie pad pad
neg	neg (0.43)	pos	1.65	I've never watched something as bad pad
neg	neg (0.00)	pos	0.12	It is a disgusting movie! pad pad

Legend: ☐ Negative ☐ Neutral ☐ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
pos	pos (0.98)	pos	1.85	It was a fantastic performance ! pad
pos	neg (0.01)	pos	1.12	Best film ever pad pad pad pad
pos	neg (0.35)	pos	1.57	Such a great show! pad pad pad
neg	neg (0.00)	pos	0.06	It was a horrible movie pad pad
neg	neg (0.43)	pos	1.65	I've never watched something as bad pad
neg	neg (0.00)	pos	0.12	It is a disgusting movie! pad pad

CNN + Embeddings + Frozen N first epochs

```
cnn_and_frozen_embeddings_model = CNN(vocab_size=vocab_size, emb_dim=dim, out_channels=out_channels, kernel_sizes=kernel_sizes, dropou

word_embeddings = TEXT.vocab.vectors

prev_shape = cnn_and_frozen_embeddings_model.embedding.weight.shape

cnn_and_frozen_embeddings_model.embedding.weight.data = torch.clone(word_embeddings)

assert prev_shape == cnn_and_frozen_embeddings_model.embedding.weight.shape
cnn_and_frozen_embeddings_model.to(device)

opt = torch.optim.Adam(cnn_and_frozen_embeddings_model.parameters(), lr=1e-4)
loss_func = nn.BCEWithLogitsLoss()

max_epochs = 50
patience = 15

train_cnn(cnn_and_frozen_embeddings_model, loss_func, train_iter, val_iter, max_epochs, patience, num_freeze_epochs=20)
```

```
Epoch: 1, Training Loss: 0.6767279505729675, Validation Loss: 0.6421321034431458
Epoch: 2, Training Loss: 0.6141664385795593, Validation Loss: 0.5650098919868469
Epoch: 3, Training Loss: 0.5261306762695312, Validation Loss: 0.47656527161598206
Epoch: 4, Training Loss: 0.4586566686630249, Validation Loss: 0.42748555541038513
Epoch: 5, Training Loss: 0.42121607065200806, Validation Loss: 0.4015980660915375
Epoch: 6, Training Loss: 0.39548027515411377, Validation Loss: 0.385018825531005 86
Epoch: 7, Training Loss: 0.3787156343460083, Validation Loss: 0.37361207604408264
Epoch: 8, Training Loss: 0.36894601583480835, Validation Loss: 0.365096449851989 75
Epoch: 9, Training Loss: 0.35644420981407166, Validation Loss: 0.3579086661338806
Epoch: 10, Training Loss: 0.3496182858943939, Validation Loss: 0.3519825041294098
Epoch: 11, Training Loss: 0.3371344208717346, Validation Loss: 0.3468288481235504
Epoch: 12, Training Loss: 0.3277920186519623, Validation Loss: 0.341876417398452 76
Epoch: 13, Training Loss: 0.32094889879226685, Validation Loss: 0.3384256958961487
Epoch: 14, Training Loss: 0.3118315041065216, Validation Loss: 0.334312379360199
Epoch: 15, Training Loss: 0.3036172688007355, Validation Loss: 0.3307430148124695
Epoch: 16, Training Loss: 0.29895299673080444, Validation Loss: 0.328248858451843 26
Epoch: 17, Training Loss: 0.292708158493042, Validation Loss: 0.3250955045223236
Epoch: 18, Training Loss: 0.28394633531570435, Validation Loss: 0.3225109875202179
Epoch: 19, Training Loss: 0.27842479944229126, Validation Loss: 0.3203069567680359
Epoch: 20, Training Loss: 0.2723568379878998, Validation Loss: 0.318013668060302 73
Epoch: 21, Training Loss: 0.2643948495388031, Validation Loss: 0.313521295785903 93
Epoch: 22, Training Loss: 0.25183463096618652, Validation Loss: 0.309835016727244 75
Epoch: 23, Training Loss: 0.23830914497375488, Validation Loss: 0.307134687900543 2
Epoch: 24, Training Loss: 0.2230561226606369, Validation Loss: 0.303770005702972 4
Epoch: 25, Training Loss: 0.21227459609508514, Validation Loss: 0.300531566143035 9
Epoch: 26, Training Loss: 0.19739606976509094, Validation Loss: 0.297966033220291 14
Epoch: 27, Training Loss: 0.18485459685325623, Validation Loss: 0.295450180768966 7
Epoch: 28, Training Loss: 0.1732412576675415, Validation Loss: 0.293241381645202 64
Epoch: 29, Training Loss: 0.16174115240573883, Validation Loss: 0.291291177272796 63
Epoch: 30, Training Loss: 0.14885404706001282, Validation Loss: 0.289469122886657 7
Epoch: 31, Training Loss: 0.13797134160995483, Validation Loss: 0.288467049598693 85
Epoch: 32, Training Loss: 0.12843380868434906, Validation Loss: 0.286727130413055 4
Epoch: 33, Training Loss: 0.11915233731269836, Validation Loss: 0.287018001079559 3
Epoch: 34, Training Loss: 0.10873080790042877, Validation Loss: 0.285089135169982 9
Epoch: 35, Training Loss: 0.10037156194448471, Validation Loss: 0.284944325685501 1
Epoch: 36, Training Loss: 0.09172287583351135, Validation Loss: 0.284842044115066 53
Epoch: 37, Training Loss: 0.08283637464046478, Validation Loss: 0.286156684160232 54
Epoch: 38, Training Loss: 0.07585844397544861, Validation Loss: 0.284508377313613 9
Epoch: 39, Training Loss: 0.07033748179674149, Validation Loss: 0.287023663520813
Epoch: 40, Training Loss: 0.06372224539518356, Validation Loss: 0.285300850868225 1
Epoch: 41, Training Loss: 0.05621509999036789, Validation Loss: 0.288138747215271
Epoch: 42, Training Loss: 0.05207853019237518, Validation Loss: 0.290058255195617 7
Epoch: 43, Training Loss: 0.04726078733801842, Validation Loss: 0.289147019386291 5
Epoch: 44, Training Loss: 0.04239666834473361, Validation Loss: 0.291218778743171 69
Epoch: 45, Training Loss: 0.037737924605607986, Validation Loss: 0.293079078197479 25
Epoch: 46, Training Loss: 0.034464605152606964, Validation Loss: 0.294528752563839
Epoch: 47, Training Loss: 0.031634438782930374, Validation Loss: 0.298307359218597 4
Epoch: 48, Training Loss: 0.028561849147081375, Validation Loss: 0.298458099365234 4
Epoch: 49, Training Loss: 0.025438071215988636, Validation Loss: 0.301065981388092 04
Epoch: 50, Training Loss: 0.02351159229874611, Validation Loss: 0.3031412959098816
```

```
calc_f1_score(cnn_and_frozen_embeddings_model, test_iter, lambda model, batch: model(batch.text))
```

```
0.8806599974632263
```

Score для всех эксперментов:

- RNN (LSTM) - 0.8374814987182617
- CNN - 0.8496467471122742
- CNN + Glove Embeddings - 0.872462272644043
- CNN + Glove Embeddings (frozen first N epochs) - 0.8806599974632263

Выводы

- CNN работает лучше чем RNN
- Предобученные эмбеддинги помогают улучшить результат
- Замораживание весов эмбеддингов на первые N эпох помогает улучшить результат еще лучше

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.