2 Практическая часть

In [78]:

```
from collections import defaultdict, Counter
import copy
import os
import re
import string
import time


import pandas as pd
import numpy as np
import scipy
import matplotlib.pyplot as plt


from tqdm.notebook import trange, tqdm


import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

from filimdb_evaluation.score import load_dataset_fast
```

2.1 Загрузка датасета.

1. Составьте таблицу, в которой указано число токенов, уникальных токенов, предложений для каждой из трех частей датасета.

In [4]:

```
folder_name = './filimdb_evaluation/PTB/'
filenames = ['train', 'valid', 'test']
```

In [5]:

```
def get_file_info(filename):
    global folder_name
    tokens_cnt = defaultdict(int)
    cnt_lines = 0
    with open(folder_name + f"ptb.{filename}.txt", 'r') as inp:
        for line in inp:
            cnt_lines += 1
            for token in line.strip().split():
                tokens_cnt[token] += 1
    total_tokens = sum(tokens_cnt.values())
    unique_tokens = len(tokens_cnt.keys())
    return filename, total_tokens, unique_tokens, cnt_lines, tokens_cnt

data = {
    'file':[],
    'token_cnt':[],
    'unique_tokens':[],
    'sentences_cnt': []
}

file_dicts = []

for f in filenames:
    f_data = get_file_info(f)
```

```
    data['file'].append(f_data[0])
    data['token_cnt'].append(f_data[1])
    data['unique_tokens'].append(f_data[2])
    data['sentences_cnt'].append(f_data[3])
    file_dicts.append(f_data[4])

all_tokens = defaultdict(int)
for d in file_dicts:
    for k, v in d.items():
        all_tokens[k] += v

data['file'].append('all files')
data['token_cnt'].append(sum(data['token_cnt']))
data['unique_tokens'].append(len(all_tokens.keys()))
data['sentences_cnt'].append(sum(data['sentences_cnt']))

df = pd.DataFrame(data=data)
df
```

Out[5]:

	file	token_cnt	unique_tokens	sentences_cnt
0	train	887521	9999	42068
1	valid	70390	6021	3370
2	test	78669	6048	3761
3	all files	1036580	9999	49199

2. Приведите 10 самых частотных и 10 самых редких токенов с их частотами.

(тут видимо для всех файлов**)**

In [6]:

```
tokens_cnt = defaultdict(int)
for f in filenames:
     with open(folder_name + f"ptb.{f}.txt", 'r') as inp:
        for line in inp:
            for token in line.strip().split():
                tokens_cnt[token] += 1
cnt_list = list(tokens_cnt.items())
cnt_list.sort(key=lambda x: x[1])
most_frequent_data = {'word':[], 'cnt':[]}
for w, c in cnt_list[-10:][::-1]:
    most_frequent_data['word'].append(w)
    most_frequent_data['cnt'].append(c)
least_frequent_data = {'word':[], 'cnt':[]}
for w, c in cnt_list[:10]:
    least_frequent_data['word'].append(w)
    least_frequent_data['cnt'].append(c)
```

In [7]:

```
pd.DataFrame(data=most_frequent_data)
```

Out[7]:

	word	cnt
0	the	59421
1	<unk>	53299
2	N	37607
3	of	28427
4	to	27430
5	a	24755

	word	cnt
6	in	21032
7	and	20404
8	's	11555
9	for	10436

In [8]:

```
pd.DataFrame(data=least_frequent_data)
```

Out[8]:

	word	cnt
0	buffet	5
1	lancaster	5
2	barnett	5
3	rewrite	5
4	downgrading	5
5	backgrounds	5
6	stanza	5
7	vessel	5
8	unstable	5
9	peat	5

3. Какие специальные токены уже есть в выборке, что они означают?

Вроде как, токены выглядят как текст в треугольных кавычках. Поищем такие фрагменты.

In [9]:

```
spec_tokens = set()
for f in filenames:
    with open(folder_name + f"ptb.{f}.txt", 'r') as inp:
        for line in inp:
            cur_spec = set(re.findall(r'<[a-z]*>', line))
            spec_tokens = spec_tokens.union(cur_spec)
print(spec_tokens)
```

```
{'<unk>'}
```

Этим токеном заменяются слова, невошедшие в **10000** самых популярных в корпусе.

Также есть специальные токены вида:

1. **N -** все отдельно стоящие числа заменяются на этот токен.
2. **$ -** на этот токен заменяются все знаки валют.

2.2 Генерацей батчей.

Тут написана версия разбиения на батчи для для слов в обычном виде, чтобы проще было проверить правильность построения. Сильно ниже будет версия генератора для уже приведенных к индексам слов в предложении.

In [10]:

```
def print_batch(ind, X_b, Y_b):
```

```
    print(f"Batch # {ind}")
    for i in range(len(X_b)):
            print(X_b[i], ' ', Y_b[i])
```

In [11]:

```
def batch_generator_text(data_path, batch_size, num_steps, debug=False):
    eos_token = '<eos>'
    L_tokens = []
    with open(data_path, 'r', encoding='utf-8') as inp:
        for line in inp:
            line_tokens = list(map(str.lower, line.strip().split()))
            L_tokens.extend(line_tokens + [eos_token])

    L_shifted = L_tokens[1:]
    L_tokens = L_tokens[:-1]
    print(len(L_tokens), len(L_shifted))
    slice_len = len(L_tokens) // batch_size
    X_lists = [L_tokens[i * slice_len : (i + 1) * slice_len] for i in range(batch_size)]
    Y_lists = [L_shifted[i * slice_len : (i + 1) * slice_len] for i in range(batch_size)
]

    total_batchs = slice_len // num_steps
    for i in range(total_batchs):
        X_batch = []
        Y_batch = []
        for lst in X_lists:
            X_batch.append(lst[i * num_steps : (i + 1) * num_steps])
        for lst in Y_lists:
            Y_batch.append(lst[i * num_steps : (i + 1) * num_steps])
        if debug and i < 3:
            print_batch(i, X_batch, Y_batch)
#          yield torch.tensor(X_batch, requires_grad=False), torch.tensor(Y_batch, requir
es_grad=False)

res = batch_generator_text(folder_name + "ptb.train.txt", batch_size = 2, num_steps = 3,
debug=True)
```

```
929588 929588
Batch # 0
['aer', 'banknote', 'berlitz']   ['banknote', 'berlitz', 'calloway']
['guarantee', 'the', 'government']   ['the', 'government', 'can']
Batch # 1
['calloway', 'centrust', 'cluett']   ['centrust', 'cluett', 'fromstein']
['can', 'ensure', 'the']   ['ensure', 'the', 'same']
Batch # 2
['fromstein', 'gitano', 'guterman']   ['gitano', 'guterman', 'hydro-quebec']
['same', 'flow', 'of']   ['flow', 'of', 'resources']
```

На файле из первых трех строчек **train** датасета функция создаёт батчи похожие на правду.

2.3 Реализация **LSTM LM.**

2.3.1 Класс **LSTMCell**

Для реализации **LSTM** ячейки будем отталкиваться от реализации обычной **RNN** ячейки из семинара.

In [12]:

```
class LSTMCell(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        '''
        Args:
            input_size: Size of token embedding
            hidden_size: Size of hidden state of LSTM cell
        '''
        super(LSTMCell, self).__init__()
        self.input_size = input_size
```

```
        self.hidden_size = hidden_size

        # Creating matrices whose weights will be trained
        # Token embedding (input of this cell) will be multiplied by this matrix
        self.U_input = torch.nn.Parameter(torch.Tensor(input_size, 4 * hidden_size))
        self.BU_input = torch.nn.Parameter(torch.Tensor(4 * hidden_size))

        # Creating matrices whose weights will be trained
        # Hidden state from previous step will be multipied by this matrix
        # Zero hidden state at the initial step
        self.W_hidden = torch.nn.Parameter(torch.Tensor(hidden_size, 4 * hidden_size))
        self.BW_hidden = torch.nn.Parameter(torch.Tensor(4 * hidden_size))

        # Weights initialization
        self.reset_parameters()

    def forward(self, inp: torch.Tensor, cell_state: torch.Tensor, hidden_state: torch.T
ensor) -> (torch.Tensor, torch.Tensor):
        '''
        Performes forward pass of the recurrent cell
        Args:
            inp: Output from Embedding layer at the current timestep
                Tensor shape is (batch_size, emb_size)
            cell_state: Output cell_state from previous recurrent step or zero state
                Tensor shape is (batch_size, hidden_size)
            hidden_state: Output hidden_state from previous recurrent step or zero state
                Tensor shape is (batch_size, hidden_size)
        Returns:
            Output from LSTM cell
        '''
        hidden_mult = hidden_state @ self.W_hidden + self.BW_hidden
        input_mult  = inp @ self.U_input + self.BU_input
        matr_sum = input_mult + hidden_mult

        f, i, c_new, o, = matr_sum.chunk(chunks=4, dim=1)
        f = torch.sigmoid(f)
        i = torch.sigmoid(i)
        c_new = torch.tanh(c_new)
        o = torch.sigmoid(o)

        cell_state_new = cell_state * f + i * c_new
        hidden_state_new = o * torch.tanh(cell_state_new)

        return cell_state_new, hidden_state_new

    def reset_parameters(self):
        '''
        Weights initialization
        '''
        stdv = 1.0 / np.sqrt(self.hidden_size)
        for weight in self.parameters():
            torch.nn.init.uniform_(weight, -stdv, stdv)
```

8 матриц и векторов смещений заменили на **2** каждого вида.
Всё перемножили и сложили по формулам, применили функции активация к каждой из **4** частей большой матрицы.
Дальше осталось просто всё правильно поэлементно перемножить и получить новые состояния ячейки и скрытое состояние.

2.3.2 Класс LSTMLayer

In [13]:

```
class LSTMLayer(torch.nn.Module):
    def __init__(self, emb_size, hidden_size):
        super(LSTMLayer, self).__init__()
        self.input_size = emb_size
        self.hidden_size = hidden_size
        self.LSTMCell = LSTMCell(emb_size, hidden_size)
```

```
    def forward(self, X_batch, initial_states):
        cell_state, hidden_state = initial_states
        outputs = []
        for timestamp in range(X_batch.shape[0]):
            cell_state, hidden_state = self.LSTMCell(X_batch[timestamp], cell_state, hid
den_state)
            outputs.append(hidden_state)
        return torch.stack(outputs), (cell_state, hidden_state)
```

2.3.3 Класс LSTM

In [14]:

```
class LSTM(torch.nn.Module):
    def __init__(self, emb_size, hidden_size, num_layers, dropout_rate):
        super(LSTM, self).__init__()
        self.input_size = emb_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.dropout_rate = dropout_rate

        self.layers = []
        for i in range(num_layers):
            self.layers.append(torch.nn.Dropout(p=self.dropout_rate))
            if i == 0:
                self.layers.append(LSTMLayer(emb_size, hidden_size))
            else:
                self.layers.append(LSTMLayer(hidden_size, hidden_size))

        self.layers.append(torch.nn.Dropout(p=self.dropout_rate))
        self.layers = torch.nn.ModuleList(self.layers)

    def forward(self, X_batch, initial_states):
        for ind, layer in enumerate(self.layers):
            if ind % 2 == 1:
                X_batch, states = layer(X_batch, initial_states)
            else:
                X_batch = layer(X_batch)
        return X_batch, states
```

2.3.4 Класс PTBLM

In [15]:

```
class PTBLM(torch.nn.Module):
    def __init__(self, num_layers, emb_size, hidden_size, vocab_size, dropout_rate, weig
ht_init=0.1, tie_emb=True, adaptive=False):
        super(PTBLM, self).__init__()
        self.num_layers = num_layers
        self.input_size = emb_size
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.dropout_rate = dropout_rate
        self.weight_max = weight_init
        self.tie = tie_emb
        self.adaptive = adaptive


        self.embedding = torch.nn.Embedding(num_embeddings=vocab_size, embedding_dim=emb
_size)
        self.LSTM = LSTM(emb_size, hidden_size, num_layers, dropout_rate)
        self.decoder = torch.nn.Linear(in_features=hidden_size, out_features=vocab_size)
        self.tie_b =  torch.nn.Parameter(torch.zeros(vocab_size))

        self.adaptive_sm = torch.nn.AdaptiveLogSoftmaxWithLoss(self.hidden_size, self.vo
cab_size, cutoffs=[500, 2000, 10000])
```

```
        self.sentiment_decoder = torch.nn.Linear(in_features=self.hidden_size, out_featu
res=2)

        self.init_weights()


    def forward(self, model_input, initial_states, target=None):
        embs = self.embedding(model_input).transpose(0, 1).contiguous()

        outputs, states = self.LSTM(embs, initial_states)

        if self.adaptive:
            outputs = outputs.transpose(0, 1).contiguous()
            out, loss = self.adaptive_sm(outputs.view(-1, self.hidden_size), target.view
(-1))
            return out, loss, states

        # print(outputs.shape)
        if self.tie:
            ns, bs = outputs.shape[0], outputs.shape[1]
            outputs = outputs.view(-1, self.hidden_size)
            logits = outputs.mm(self.embedding.weight.t()) + self.tie_b
            logits = logits.view(ns, bs, self.vocab_size)
        else:
            logits = self.decoder(outputs)

        logits = logits.transpose(0, 1).contiguous()

        return logits, states

    def forward_classify(self, batch_texts, text_lenghts, initial_states):
        """
            model_input: batch of indexed tests
            text_lenghts: lenght of examples in batch
            initial_states: states for lstm layers
        """
        embs = self.embedding(batch_texts).transpose(0, 1).contiguous()

        outputs, states = self.LSTM(embs, initial_states)
        # outputs.shape = (max_len, bs, hidden_size)

        max_len, bs = outputs.shape[0], outputs.shape[1]
        outputs = outputs.transpose(0, 1).contiguous()
        # outputs.shape = (bs, max_len, hidden_size)

        # Getting last non pad output vector
        outputs = outputs[np.arange(outputs.shape[0]), text_lenghts]
        #outputs.shape = (bs, hidden_size)

#          print(outputs.shape)

        logits = self.sentiment_decoder(outputs)

        return logits, states

    def init_weights(self):
        self.embedding.weight.data.uniform_(-self.weight_max, self.weight_max)
        self.decoder.weight.data.uniform_(-self.weight_max, self.weight_max)
        torch.nn.init.uniform_(self.tie_b, -self.weight_max, self.weight_max)

    def init_hidden(self, batch_size, device):
        return torch.zeros(batch_size, self.hidden_size).to(device), torch.zeros(batch_s
ize, self.hidden_size).to(device)
```

2.4 Обучение языковой модели.

Ниже функции для подготовки **ptb** датасета и словарей.

In [16]:

```
START_TOKEN =  '<start>'
EOS_TOKEN =  '<eos>'
```

In [17]:

```
def _read_words(path):
    with open(path, 'r') as inp:
        names = inp.read().lower().split()
        return names
print(_read_words(folder_name + 'small.txt'))
```

```
['pierre', '<unk>', 'n', 'years', 'old', 'will', 'join', 'the', 'board', 'as', 'a', 'none
xecutive', 'director', 'nov.', 'n', 'mr.', '<unk>', 'is', 'chairman', 'of', '<unk>', 'n.v
.', 'the', 'dutch', 'publishing', 'group', 'rudolph', '<unk>', 'n', 'years', 'old', 'and'
, 'former', 'chairman', 'of', 'consolidated', 'gold', 'fields', 'plc', 'was', 'named', 'a
', 'nonexecutive', 'director', 'of', 'this', 'british', 'industrial', 'conglomerate']
```

In [18]:

```
def _read_sentences(path):
    with open(path, 'r') as inp:
        sentences = inp.read().lower().split('\n')
    sentences = [[START_TOKEN] + sent.split() for sent in sentences]
    return sentences

sents = _read_sentences(folder_name + 'small.txt')
for sent in sents:
    print(sent)
```

```
['<start>', 'pierre', '<unk>', 'n', 'years', 'old', 'will', 'join', 'the', 'board', 'as',
'a', 'nonexecutive', 'director', 'nov.', 'n']
['<start>', 'mr.', '<unk>', 'is', 'chairman', 'of', '<unk>', 'n.v.', 'the', 'dutch', 'pub
lishing', 'group']
['<start>', 'rudolph', '<unk>', 'n', 'years', 'old', 'and', 'former', 'chairman', 'of', '
consolidated', 'gold', 'fields', 'plc', 'was', 'named', 'a', 'nonexecutive', 'director',
'of', 'this', 'british', 'industrial', 'conglomerate']
```

In [19]:

```
def _build_vocab(path):
    data = _read_words(path)
    special_tokens = [START_TOKEN, EOS_TOKEN]
    data += special_tokens

    counter = Counter(data)
    sorted_words = sorted(counter.items(), key=lambda x: -x[1])

    words = [w for w, _ in sorted_words]
    word_to_id = dict(zip(words, range(len(words))))
    id_to_word = {v: k for k, v in word_to_id.items()}

    return word_to_id, id_to_word

word_to_id, id_to_word = _build_vocab(folder_name + 'small.txt')
print('Vocab size = ', len(word_to_id))
print(list(word_to_id.items()))
```

```
Vocab size =  37
[('<unk>', 0), ('n', 1), ('of', 2), ('years', 3), ('old', 4), ('the', 5), ('a', 6), ('non
executive', 7), ('director', 8), ('chairman', 9), ('pierre', 10), ('will', 11), ('join',
12), ('board', 13), ('as', 14), ('nov.', 15), ('mr.', 16), ('is', 17), ('n.v.', 18), ('du
tch', 19), ('publishing', 20), ('group', 21), ('rudolph', 22), ('and', 23), ('former', 24
), ('consolidated', 25), ('gold', 26), ('fields', 27), ('plc', 28), ('was', 29), ('named'
, 30), ('this', 31), ('british', 32), ('industrial', 33), ('conglomerate', 34), ('<start>
', 35), ('<eos>', 36)]
```

In [20]:

```
def _sentences_to_word_ids(word_to_id, texts = None, path=None):
    if path is not None:
        sentences = _read_sentences(path)
```

```
    elif texts is not None:
        sentences = texts
    return [[word_to_id[word] for word in sent] for sent in sentences]

word_to_id, id_to_word = _build_vocab(folder_name + 'small.txt')
res = _sentences_to_word_ids(word_to_id, path = folder_name + 'small.txt',)
for sent in res:
    print(sent)
```

```
[35, 10, 0, 1, 3, 4, 11, 12, 5, 13, 14, 6, 7, 8, 15, 1]
[35, 16, 0, 17, 9, 2, 0, 18, 5, 19, 20, 21]
[35, 22, 0, 1, 3, 4, 23, 24, 9, 2, 25, 26, 27, 28, 29, 30, 6, 7, 8, 2, 31, 32, 33, 34]
```

In [21]:

```
def ptb_raw_data(data_path, debug=False):
    train_path = os.path.join(data_path, 'ptb.train.txt')
    dev_path = os.path.join(data_path, 'ptb.valid.txt')
    test_path = os.path.join(data_path, 'ptb.test.txt')

    word_to_id, id_to_word = _build_vocab(train_path)
    train_data = _sentences_to_word_ids(word_to_id, path=train_path)
    dev_data = _sentences_to_word_ids(word_to_id, path=dev_path)
    test_data = _sentences_to_word_ids(word_to_id, path=test_path)

    return train_data, dev_data, test_data, word_to_id, id_to_word

train_data, dev_data, test_data, word_to_ind, ind_to_word = ptb_raw_data(folder_name)
print('Vocab size = ', len(word_to_ind))
for sent in train_data[:5]:
    print(sent)
```

```
Vocab size =  10001
[9999, 9969, 9970, 9971, 9972, 9973, 9974, 9975, 9976, 9977, 9978, 9979, 9980, 9981, 9982
, 9983, 9984, 9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992]
[9999, 8568, 1, 2, 71, 392, 32, 2115, 0, 145, 18, 5, 8569, 274, 406, 2]
[9999, 22, 1, 12, 140, 3, 1, 5277, 0, 3054, 1580, 95]
[9999, 7231, 1, 2, 71, 392, 7, 336, 140, 3, 2467, 656, 2157, 948, 23, 520, 5, 8569, 274,
3, 38, 302, 436, 3660]
[9999, 5, 940, 3, 3142, 494, 261, 4, 136, 5881, 4218, 5882, 29, 985, 5, 239, 754, 3, 1012
, 2764, 210, 5, 95, 3, 426, 4059, 4, 13, 44, 54, 2, 71, 194, 1232, 219]
```

In [22]:

```
def batch_generator_inds(data, word_to_id, batch_size, num_steps, debug=False):
    L_tokens = []
    for sentence in data:
        L_tokens.extend(sentence + [word_to_id[EOS_TOKEN]])
    L_shifted = L_tokens[1:]
    L_tokens = L_tokens[:-1]

    slice_len = len(L_tokens) // batch_size
    X_lists = [L_tokens[i * slice_len : (i + 1) * slice_len] for i in range(batch_size)]
    Y_lists = [L_shifted[i * slice_len : (i + 1) * slice_len] for i in range(batch_size)
]
    # print(len(X_lists))

    total_batchs = slice_len // num_steps
    for i in range(total_batchs):
        X_batch = []
        Y_batch = []
        for lst in X_lists:
            X_batch.append(lst[i * num_steps : (i + 1) * num_steps])
        for lst in Y_lists:
            Y_batch.append(lst[i * num_steps : (i + 1) * num_steps])
        if debug:
            print_batch(i, X_batch, Y_batch)
        else:
            if X_batch:
                yield torch.tensor(X_batch, requires_grad=False), torch.tensor(Y_batch,
requires_grad=False)
```

```
train_data, dev_data, test_data, word_to_ind, ind_to_word = ptb_raw_data(folder_name, de
bug=True)
res = batch_generator_inds(train_data, word_to_ind, batch_size = 2, num_steps = 3)
next(res)
```

Out[22]:

```
(tensor([[9999, 9969, 9970],
         [   4, 2818,  507]]),
 tensor([[9969, 9970, 9971],
         [2818,  507,   35]]))
```

Теперь перейдем к функциям для обучения сети.

In [23]:

```
def update_lr(optimizer, lr):
    for g in optimizer.param_groups:
        g['lr'] = lr

def run_epoch(
    lr,
    model,
    data,
    word_to_id,
    loss_fn,
    batch_size,
    num_steps,
    optimizer = None,
    clip_value = None,
    device = None
) -> float:
    '''
    Performs one training epoch or inference epoch
    Args:
        lr: Learning rate for this epoch
        model: Language model object
        data: Data that will be passed through the language model
        char_to_id: Mapping of each character into its index in the vocabulary
        loss_fn: Torch loss function
        optimizer: Torch optimizer
        device: Input tensors should be sent to this device
    Returns:
        Perplexity
    '''

    total_loss, total_examples = 0.0, 0
    generator = batch_generator_inds(data, word_to_id=word_to_id, batch_size=batch_size,
num_steps=num_steps)

    initial_state = model.init_hidden(batch_size=batch_size, device=device)
    for step, (X, Y) in enumerate(generator):
        X = X.to(device)
        Y = Y.to(device)

        if model.adaptive:
            out, loss, new_state = model(X, initial_state, target=Y)
        else:
            logits, new_state = model(X, initial_state)
        initial_state = (new_state[0].detach(), new_state[1].detach())

        if model.adaptive:
            total_examples += out.shape[0]
            total_loss += loss.item() * out.shape[0]
        else:
            loss = loss_fn(logits.view((-1, model.vocab_size)), Y.view(-1))
            total_examples += loss.size(0)
            total_loss += loss.sum().item()
            loss = loss.mean()
```

```
        # Gradients computation
        if optimizer is not None:
            loss.backward()

#             print("-------------------------------------------")
#             print("CHECK GRADS")
#             for p in list(filter(lambda p: p.grad is not None, model.parameters())):
#                 print(p.grad.data.norm(2).item())
#             print("-------------------------------------------")


            # We have a new learning rate value at every step, so it needs to be updated
            update_lr(optimizer, lr)

            # Gradient clipping by predefined norm value - usually 5.0
            if clip_value is not None:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip_value)

            # Applying gradients - one gradient descent step
            optimizer.step()
            optimizer.zero_grad()

    return np.exp(total_loss / total_examples)
```

In [24]:

```
base_config = {
        'batch_size': 64, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 256,
        'hidden_size': 256, 'vocab_size': -1,
        'dropout_rate': 0.2, 'num_epochs': 13,
        'learning_rate': 0.01, 'lr_decay' : 0.9,
        'epoch_decay' : 6, 'tied_embs': False,
        'weight_init':0.1, 'grad_clipping' : None,
        'optimizer' : 'Adam'
}
```

In [25]:

```
raw_data = ptb_raw_data(folder_name)
train_data, dev_data, test_data, word_to_id, id_to_word = raw_data
base_config['vocab_size'] = len(word_to_id)
base_config['vocab_size']
```

Out[25]:

```
10001
```

наконец-то функция для тестировочного обучения

In [26]:

```
def train_on_config(cur_config, train_data, dev_data, test_data):
    model = PTBLM(num_layers=cur_config['num_layers'], emb_size=cur_config['emb_size'],
              hidden_size=cur_config['hidden_size'], vocab_size=cur_config['vocab_size']
,
              dropout_rate=cur_config['dropout_rate'], weight_init=cur_config['weight_in
it'],
              tie_emb=cur_config['tied_embs'],
             )
    print(model)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("Training on device: ", device)
    model.to(device)
    loss_fn = torch.nn.CrossEntropyLoss(reduction='none')
    if cur_config['optimizer'] == 'Adam':
        optimizer = torch.optim.Adam(model.parameters(), lr=cur_config['learning_rate'])
    elif cur_config['optimizer'] == 'SGD':
        optimizer = torch.optim.SGD(model.parameters(), lr=cur_config['learning_rate'])
    else:
        optimizer = torch.optim.SGD(model.parameters(), lr=cur_config['learning_rate'],
```

```
momentum=0.8)
    plot_data = [[], []]
    for i in trange(cur_config['num_epochs']):
        lr_decay = cur_config['lr_decay'] ** max(i + 1 - cur_config['epoch_decay'], 0.0)
        if cur_config['lr_decay'] > 1:
            lr_decay = 1 / lr_decay
        decayed_lr = cur_config['learning_rate'] * lr_decay

        model.train()
        train_perplexity = run_epoch(decayed_lr, model, train_data,
                                     word_to_id, loss_fn,
                                     cur_config['batch_size'], cur_config['num_steps'],
                                     optimizer=optimizer,
                                     clip_value=cur_config['grad_clipping'],
                                     device=device)

        model.eval()

        # Disabling gradient calculation.
        # It will reduce memory consumption for computations
        # The result of every computation will have requires_grad=False,
        with torch.no_grad():
            dev_perplexity = run_epoch(decayed_lr, model, dev_data,
                                       word_to_id, loss_fn, cur_config['batch_size'], c
ur_config['num_steps'],
                                       device=device)

        plot_data[0].append(train_perplexity)
        plot_data[1].append(dev_perplexity)
        print(f'Epoch: {i+1}. Learning rate: {decayed_lr:.3f}. '
              f'Train Perplexity: {train_perplexity:.3f}. '
              f'Dev Perplexity: {dev_perplexity:.3f}. '
             )

    model.eval()
    with torch.no_grad():
        test_perplexity = run_epoch(
            decayed_lr, model, test_data,
            word_to_id, loss_fn, cur_config['batch_size'], cur_config['num_steps'],
            device=device)
        print(f"Test Perplexity: {test_perplexity:.3f}")

    return model, plot_data
```

In [27]:

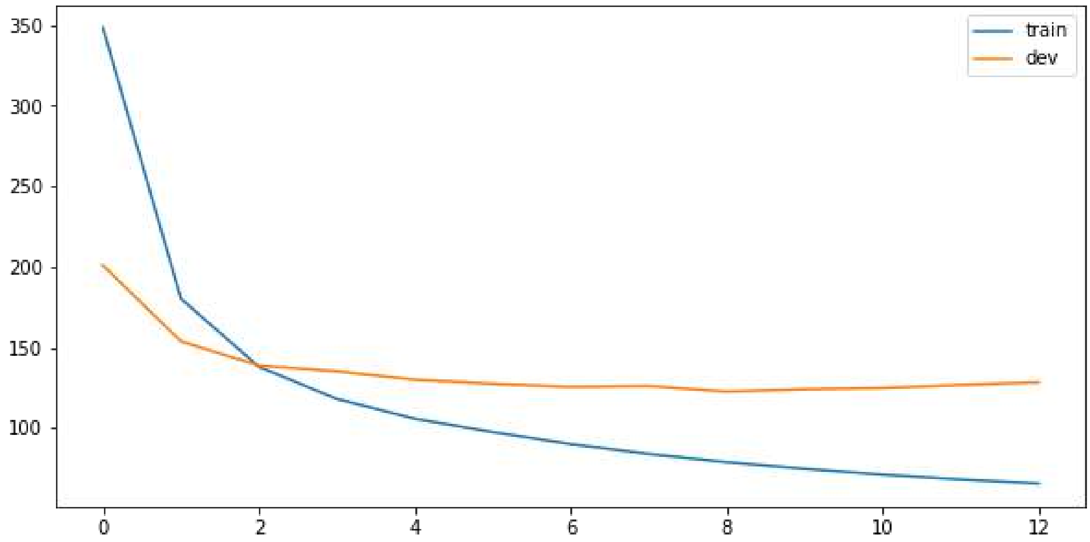
```
base_model, base_train_data = train_on_config(base_config, train_data, dev_data, test_dat
a)
```

```
PTBLM(
  (embedding): Embedding(10001, 256)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.2, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.2, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.2, inplace=False)
    )
  )
  (decoder): Linear(in_features=256, out_features=10001, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=256, out_features=503, bias=False)
    (tail): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=256, out_features=64, bias=False)
        (1): Linear(in_features=64, out_features=1500, bias=False)
      )
```

```
        (1): Sequential(
          (0): Linear(in_features=256, out_features=16, bias=False)
          (1): Linear(in_features=16, out_features=8000, bias=False)
        )
        (2): Sequential(
          (0): Linear(in_features=256, out_features=4, bias=False)
          (1): Linear(in_features=4, out_features=1, bias=False)
        )
      )
    )
    (sentiment_decoder): Linear(in_features=256, out_features=2, bias=True)
)
Training on device:  cuda:0


Epoch: 1. Learning rate: 0.010. Train Perplexity: 348.177. Dev Perplexity: 200.924.
Epoch: 2. Learning rate: 0.010. Train Perplexity: 180.191. Dev Perplexity: 153.685.
Epoch: 3. Learning rate: 0.010. Train Perplexity: 137.892. Dev Perplexity: 138.730.
Epoch: 4. Learning rate: 0.010. Train Perplexity: 117.969. Dev Perplexity: 135.068.
Epoch: 5. Learning rate: 0.010. Train Perplexity: 105.617. Dev Perplexity: 129.934.
Epoch: 6. Learning rate: 0.010. Train Perplexity: 97.422. Dev Perplexity: 127.332.
Epoch: 7. Learning rate: 0.009. Train Perplexity: 89.902. Dev Perplexity: 125.433.
Epoch: 8. Learning rate: 0.008. Train Perplexity: 83.761. Dev Perplexity: 126.054.
Epoch: 9. Learning rate: 0.007. Train Perplexity: 78.708. Dev Perplexity: 122.514.
Epoch: 10. Learning rate: 0.007. Train Perplexity: 74.587. Dev Perplexity: 124.048.
Epoch: 11. Learning rate: 0.006. Train Perplexity: 70.950. Dev Perplexity: 124.801.
Epoch: 12. Learning rate: 0.005. Train Perplexity: 67.971. Dev Perplexity: 126.628.
Epoch: 13. Learning rate: 0.005. Train Perplexity: 65.380. Dev Perplexity: 128.310.
Test Perplexity: 116.980
```

In [28]:

```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)
ax.plot(base_train_data[0], label='train')
ax.plot(base_train_data[1], label='dev')
ax.legend()
plt.show()
```



Сеть обученная выше наиболее похожа на самую маленькую, представленную в статье **Zaremba.**
Отметим, что среднюю сеть из статьи не получается обучать на **Adam,** потому что-то там происходит что-то неадекватное с градиентами и лоссом.
Поэтому возвращаем **SGD** и пробуем обучить среднюю сеть.

In [46]:

```
medium_config = {
        'batch_size': 20, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 650,
        'hidden_size': 650, 'vocab_size': 10001,
        'dropout_rate': 0.5, 'num_epochs': 39,
        'learning_rate': 1.0, 'lr_decay' : 0.8,
```

```
          'epoch_decay' : 10, 'tied_embs': False,
          'weight_init': 0.05, 'grad_clipping' : 5,
          'optimizer' : 'SGD'
           }
```

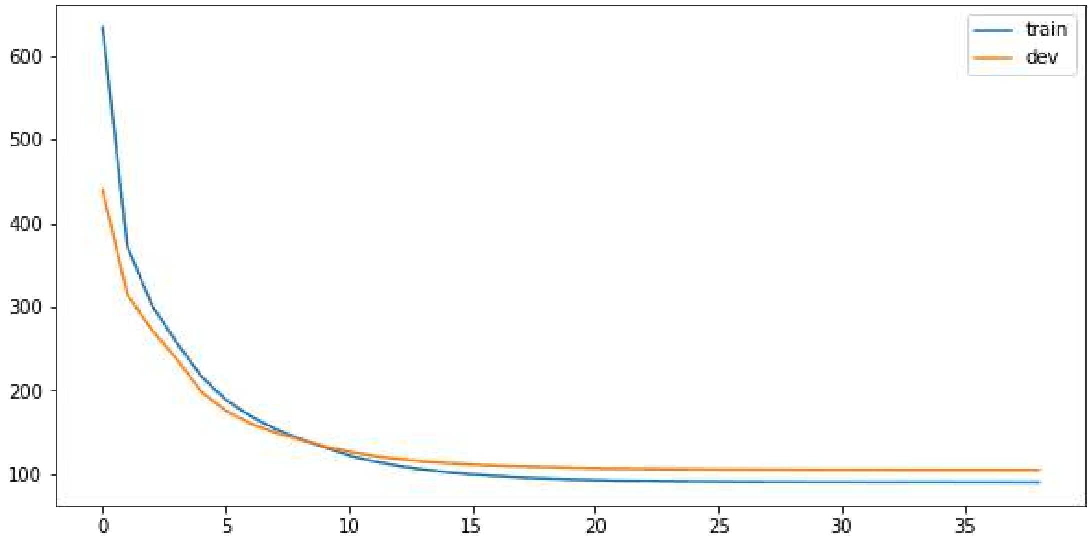
In [47]:

```
medium_model, medium_train_data = train_on_config(medium_config)
```

```
PTBLM(
  (embedding): Embedding(10001, 650)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.5, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.5, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (decoder): Linear(in_features=650, out_features=10001, bias=True)
)
Training on device:  cuda:0


Epoch: 1. Learning rate: 1.000. Train Perplexity: 633.883. Dev Perplexity: 439.388.
Epoch: 2. Learning rate: 1.000. Train Perplexity: 372.538. Dev Perplexity: 314.932.
Epoch: 3. Learning rate: 1.000. Train Perplexity: 302.003. Dev Perplexity: 272.168.
Epoch: 4. Learning rate: 1.000. Train Perplexity: 257.440. Dev Perplexity: 237.747.
Epoch: 5. Learning rate: 1.000. Train Perplexity: 217.215. Dev Perplexity: 198.726.
Epoch: 6. Learning rate: 1.000. Train Perplexity: 188.994. Dev Perplexity: 175.633.
Epoch: 7. Learning rate: 1.000. Train Perplexity: 169.267. Dev Perplexity: 160.579.
Epoch: 8. Learning rate: 1.000. Train Perplexity: 154.120. Dev Perplexity: 149.544.
Epoch: 9. Learning rate: 1.000. Train Perplexity: 142.161. Dev Perplexity: 141.131.
Epoch: 10. Learning rate: 1.000. Train Perplexity: 132.170. Dev Perplexity: 133.325.
Epoch: 11. Learning rate: 0.800. Train Perplexity: 122.400. Dev Perplexity: 126.713.
Epoch: 12. Learning rate: 0.640. Train Perplexity: 115.275. Dev Perplexity: 121.686.
Epoch: 13. Learning rate: 0.512. Train Perplexity: 109.875. Dev Perplexity: 118.061.
Epoch: 14. Learning rate: 0.410. Train Perplexity: 105.643. Dev Perplexity: 115.229.
Epoch: 15. Learning rate: 0.328. Train Perplexity: 102.198. Dev Perplexity: 113.077.
Epoch: 16. Learning rate: 0.262. Train Perplexity: 99.618. Dev Perplexity: 111.438.
Epoch: 17. Learning rate: 0.210. Train Perplexity: 97.634. Dev Perplexity: 110.168.
Epoch: 18. Learning rate: 0.168. Train Perplexity: 95.863. Dev Perplexity: 109.082.
Epoch: 19. Learning rate: 0.134. Train Perplexity: 94.615. Dev Perplexity: 108.523.
Epoch: 20. Learning rate: 0.107. Train Perplexity: 93.585. Dev Perplexity: 107.471.
Epoch: 21. Learning rate: 0.086. Train Perplexity: 92.775. Dev Perplexity: 106.965.
Epoch: 22. Learning rate: 0.069. Train Perplexity: 92.068. Dev Perplexity: 106.601.
Epoch: 23. Learning rate: 0.055. Train Perplexity: 91.824. Dev Perplexity: 106.286.
Epoch: 24. Learning rate: 0.044. Train Perplexity: 91.313. Dev Perplexity: 105.863.
Epoch: 25. Learning rate: 0.035. Train Perplexity: 90.958. Dev Perplexity: 105.679.
Epoch: 26. Learning rate: 0.028. Train Perplexity: 90.784. Dev Perplexity: 105.366.
Epoch: 27. Learning rate: 0.023. Train Perplexity: 90.522. Dev Perplexity: 105.333.
Epoch: 28. Learning rate: 0.018. Train Perplexity: 90.465. Dev Perplexity: 105.152.
Epoch: 29. Learning rate: 0.014. Train Perplexity: 90.470. Dev Perplexity: 105.077.
Epoch: 30. Learning rate: 0.012. Train Perplexity: 90.292. Dev Perplexity: 104.991.
Epoch: 31. Learning rate: 0.009. Train Perplexity: 90.285. Dev Perplexity: 104.979.
Epoch: 32. Learning rate: 0.007. Train Perplexity: 90.162. Dev Perplexity: 104.928.
Epoch: 33. Learning rate: 0.006. Train Perplexity: 90.107. Dev Perplexity: 104.856.
Epoch: 34. Learning rate: 0.005. Train Perplexity: 90.213. Dev Perplexity: 104.814.
Epoch: 35. Learning rate: 0.004. Train Perplexity: 90.230. Dev Perplexity: 104.788.
Epoch: 36. Learning rate: 0.003. Train Perplexity: 90.049. Dev Perplexity: 104.765.
Epoch: 37. Learning rate: 0.002. Train Perplexity: 90.114. Dev Perplexity: 104.751.
Epoch: 38. Learning rate: 0.002. Train Perplexity: 90.063. Dev Perplexity: 104.742.
Epoch: 39. Learning rate: 0.002. Train Perplexity: 90.045. Dev Perplexity: 104.732.
Test Perplexity: 101.630
```

In [48]:

```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)
ax.plot(medium_train_data[0], label='train')
ax.plot(medium_train_data[1], label='dev')
ax.legend()
plt.show()
```



Не очень понятно почему, но модель описанная в статье не показывает тех результатов, которые должна. Я думаю, это из-за слишком сильного и резкого уменьшения **lr**, потому что сдвинув порог начального уменьшения, удалось улучшить результаты.
Можно конечно ещё поиграться с рейтом и порогом, но попробуем теперь **SGD with momentum**.
(Можно было бы попробовать другие размеры слоёв и т.д., но мне не даёт покоя то, что я не могу выбить нормальную перплексию для средней модели.)

In [54]:

```
medium_momentum_config = {
        'batch_size': 20, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 650,
        'hidden_size': 650, 'vocab_size': 10001,
        'dropout_rate': 0.5, 'num_epochs': 39,
        'learning_rate': 1.0, 'lr_decay' : 0.9,
        'epoch_decay' : 10, 'tied_embs': False,
        'weight_init': 0.05, 'grad_clipping' : 5,
        'optimizer' : 'Momentum'
         }
```

In [55]:

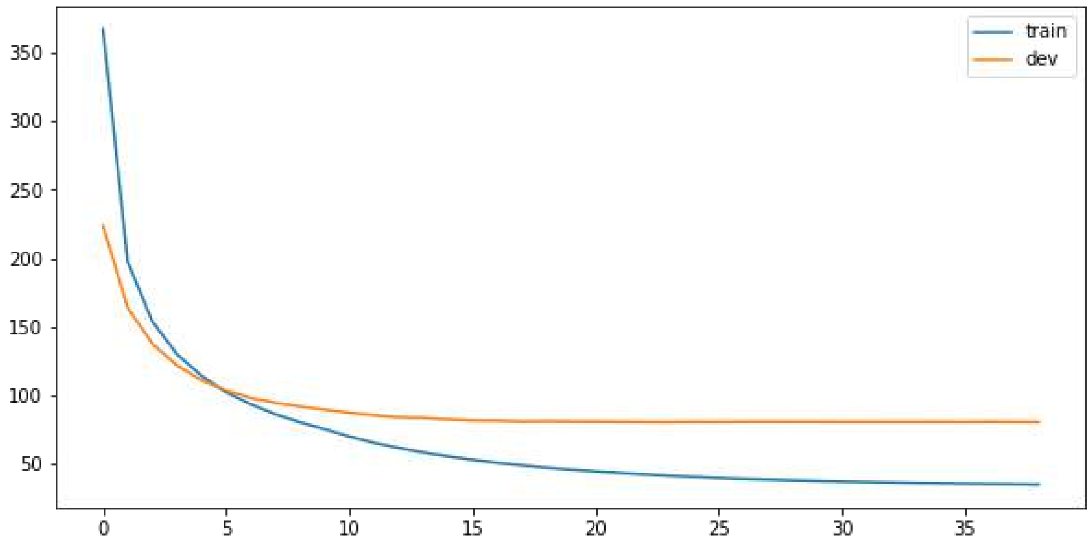
```
best_medium_model, mediumMom_train_data = train_on_config(medium_momentum_config)
```

```
PTBLM(
  (embedding): Embedding(10001, 650)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.5, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.5, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (decoder): Linear(in_features=650, out_features=10001, bias=True)
)
Training on device:  cuda:0


Epoch: 1. Learning rate: 1.000. Train Perplexity: 367.003. Dev Perplexity: 223.769.
```

```
Epoch: 2. Learning rate: 1.000. Train Perplexity: 197.540. Dev Perplexity: 163.970.
Epoch: 3. Learning rate: 1.000. Train Perplexity: 154.021. Dev Perplexity: 137.690.
Epoch: 4. Learning rate: 1.000. Train Perplexity: 130.067. Dev Perplexity: 121.907.
Epoch: 5. Learning rate: 1.000. Train Perplexity: 114.202. Dev Perplexity: 110.973.
Epoch: 6. Learning rate: 1.000. Train Perplexity: 102.295. Dev Perplexity: 103.803.
Epoch: 7. Learning rate: 1.000. Train Perplexity: 93.508. Dev Perplexity: 98.044.
Epoch: 8. Learning rate: 1.000. Train Perplexity: 86.204. Dev Perplexity: 94.622.
Epoch: 9. Learning rate: 1.000. Train Perplexity: 80.383. Dev Perplexity: 91.859.
Epoch: 10. Learning rate: 1.000. Train Perplexity: 75.276. Dev Perplexity: 89.566.
Epoch: 11. Learning rate: 0.900. Train Perplexity: 69.964. Dev Perplexity: 87.278.
Epoch: 12. Learning rate: 0.810. Train Perplexity: 65.419. Dev Perplexity: 85.503.
Epoch: 13. Learning rate: 0.729. Train Perplexity: 61.632. Dev Perplexity: 83.899.
Epoch: 14. Learning rate: 0.656. Train Perplexity: 58.359. Dev Perplexity: 83.604.
Epoch: 15. Learning rate: 0.590. Train Perplexity: 55.586. Dev Perplexity: 82.462.
Epoch: 16. Learning rate: 0.531. Train Perplexity: 53.023. Dev Perplexity: 81.735.
Epoch: 17. Learning rate: 0.478. Train Perplexity: 50.855. Dev Perplexity: 81.508.
Epoch: 18. Learning rate: 0.430. Train Perplexity: 49.059. Dev Perplexity: 80.952.
Epoch: 19. Learning rate: 0.387. Train Perplexity: 47.325. Dev Perplexity: 81.158.
Epoch: 20. Learning rate: 0.349. Train Perplexity: 45.859. Dev Perplexity: 80.947.
Epoch: 21. Learning rate: 0.314. Train Perplexity: 44.498. Dev Perplexity: 80.893.
Epoch: 22. Learning rate: 0.282. Train Perplexity: 43.353. Dev Perplexity: 80.685.
Epoch: 23. Learning rate: 0.254. Train Perplexity: 42.322. Dev Perplexity: 80.579.
Epoch: 24. Learning rate: 0.229. Train Perplexity: 41.382. Dev Perplexity: 80.534.
Epoch: 25. Learning rate: 0.206. Train Perplexity: 40.556. Dev Perplexity: 80.690.
Epoch: 26. Learning rate: 0.185. Train Perplexity: 39.776. Dev Perplexity: 80.567.
Epoch: 27. Learning rate: 0.167. Train Perplexity: 39.082. Dev Perplexity: 80.787.
Epoch: 28. Learning rate: 0.150. Train Perplexity: 38.580. Dev Perplexity: 80.820.
Epoch: 29. Learning rate: 0.135. Train Perplexity: 37.997. Dev Perplexity: 80.766.
Epoch: 30. Learning rate: 0.122. Train Perplexity: 37.516. Dev Perplexity: 80.734.
Epoch: 31. Learning rate: 0.109. Train Perplexity: 37.100. Dev Perplexity: 80.655.
Epoch: 32. Learning rate: 0.098. Train Perplexity: 36.713. Dev Perplexity: 80.680.
Epoch: 33. Learning rate: 0.089. Train Perplexity: 36.376. Dev Perplexity: 80.634.
Epoch: 34. Learning rate: 0.080. Train Perplexity: 36.013. Dev Perplexity: 80.638.
Epoch: 35. Learning rate: 0.072. Train Perplexity: 35.808. Dev Perplexity: 80.687.
Epoch: 36. Learning rate: 0.065. Train Perplexity: 35.499. Dev Perplexity: 80.541.
Epoch: 37. Learning rate: 0.058. Train Perplexity: 35.307. Dev Perplexity: 80.758.
Epoch: 38. Learning rate: 0.052. Train Perplexity: 35.107. Dev Perplexity: 80.575.
Epoch: 39. Learning rate: 0.047. Train Perplexity: 34.896. Dev Perplexity: 80.565.
Test Perplexity: 77.659
```

In [56]:

```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)
ax.plot(mediumMom_train_data[0], label='train')
ax.plot(mediumMom_train_data[1], label='dev')
ax.legend()
plt.show()
```



Ну и как видно, всё это было не зря. Удалось выбить достойные цифры для средней модели.
Т.к. время поджимает, пока было принято решение оставить попытки выбивать лучшие результаты и пробовать большие модели. Но если останется время, то обязательно попробуем.

Результаты из лидерборда:

Средняя модель из статьи **Zaremba(**обучалась **1** час на **tesla P4)**

In [1]:

```
medium_config = { 'batch_size': 20, 'num_steps': 35,
            'num_layers': 2, 'emb_size': 650,
            'hidden_size': 650, 'vocab_size': -1,
            'dropout_rate': 0.5, 'num_epochs': 25,
            'learning_rate': 1.0, 'lr_decay' : 0.9,
            'epoch_decay' : 10, 'weight_init': 0.05,
            'grad_clipping':5,
            'optimizer':'Momentum'
         }
```

```
train 42.00
valid 94.91
test 91.41
```

Большая модель из статьи **Zaremba,** но с меньшим количеством эпох**(**обучалась **4** часа на **tesla P4)**

In [ ]:

```
big_config = { 'batch_size': 20, 'num_steps': 35,
            'num_layers': 2, 'emb_size': 1500,
            'hidden_size': 1500, 'vocab_size': -1,
            'dropout_rate': 0.65, 'num_epochs': 40,
            'learning_rate': 1.0, 'lr_decay' : 1.15,
            'epoch_decay' : 14, 'weight_init': 0.04,
            'grad_clipping': 10,
            'optimizer':'Momentum'
         }
```

```
train 43.26
valid 92.252
test 88.636
```

Добавив обычный **Momentum SGD(**не **Nesterov)** можно неплохо улучшить результат средней модели и скорее всего большой модели**.**

2.5 Генерация предложений

In [29]:

```
def sentence_sampling(model, word_to_id, id_to_word, cnt = 1, temperature = 1.0, max_len
= 20):

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    sentences = []
    for _ in range(cnt):
#         print(f"generating sentence #{_}")
        sent = []
        prev_word_idx = word_to_id[START_TOKEN]
        model.eval()
        with torch.no_grad():
            hidden_states = model.init_hidden(batch_size=1, device=device)
            for i in range(max_len):
                X = torch.tensor([[prev_word_idx]], dtype=torch.int64, device=device)
```

```
                logits, new_states = model(X, hidden_states)
                hidden_states = (new_states[0].detach(), new_states[1].detach())
                softmax = torch.nn.functional.softmax(logits, -1).cpu().numpy()[0, 0]
                if temperature != 1.0:
                    softmax = np.float_power(softmax, 1.0 / temperature)
                    softmax /= softmax.sum()
                prev_word_idx = np.random.choice(list(range(len(softmax))), p=softmax)
                sent.append(id_to_word[int(prev_word_idx)])

        sentences.append(' '.join(sent).split(EOS_TOKEN)[0])
    return sentences
```

Вопрос. Приведите по 10 примеров сгенерированных предложений при разных температурах.

In [30]:

```
generated_sents_1 = sentence_sampling(base_model, word_to_id, id_to_word, cnt=10, temper
ature=1)
```

In [31]:

```
for sent in generated_sents_1:
    print(sent)
```

```
chairman steven <unk> the bill for the irs directors age <unk> more <unk> owners ' antibo
dy
an investment banker chairman alan <unk> vice president and chief operating officer in th
e other trade committee oct. n to
tariff devices in red leaders get theatrical swedish emergency service but the conference
coup force decided to stay with a
consumer installment profits were n't expected
mr. hahn & co. said it would n't disclose it has been strong a n n increase by the result
s
macmillan bates also want to cut down the statutory loan for the <unk> machines and the w
hitbread spirits channel it
of the trust food industries scheduled to be considered the tax code in july
for the summer with sales $ n million
that a real system does he said
possible investigation of the women <unk> indicating a record to azt throughout adjusting
periods and opposition into who could violate
```

In [32]:

```
generated_sents_2 = sentence_sampling(base_model, word_to_id, id_to_word, cnt=10, temper
ature=0.5)
```

In [33]:

```
for sent in generated_sents_2:
    print(sent)
```

```
the <unk> <unk> of the n <unk> is the <unk> <unk> of <unk> and the <unk> <unk> of <unk> <
unk>
the buyer are n't interested in the <unk> of <unk>
the <unk> <unk> <unk> and <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <un
k> a <unk> <unk> <unk>
the agreement with the bill is n't likely to <unk> the <unk> <unk> <unk> <unk> <unk> <unk
> <unk> the <unk>
the n n <unk> and the <unk> <unk> the <unk> <unk> and <unk> are <unk>
<unk> the <unk> <unk> of <unk>
<unk> the <unk> <unk> <unk> <unk> in n n <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> the
<unk> and the <unk> wine and political <unk> <unk> <unk>
the <unk> of <unk> the <unk> family has been <unk> to <unk> the <unk> <unk> <unk> the <un
k> <unk> in
the company is <unk> a <unk> of <unk>
```

In [34]:

```
generated_sents_3 = sentence_sampling(base_model, word_to_id, id_to_word, cnt=10, temper
```

```
ature=10)
```

In [35]:

```
for sent in generated_sents_3:
    print(sent)
```

```
diesel anyone jersey he resource civil question purchase horrible barbara mile david acti
on pitches emigration nationally crossland participant newsweek sang
hudson predicts withdrawals parcel enter turf orange abortion classic dominates expensive
interesting consequences inventor revolving indonesia espn mci pile described
consistently cineplex pulled governments ball creatures overly s.c. leveraged confirmed p
ools master retire described contributions ventures result resolve improvement associates
larger charts united quantum family participate wayne replacement misleading sierra eaton
comsat engelken joseph electric miami shareholder toronto short-term shift
highly neatly liberty jets state interviewed ehrlich manic capel owen debt builds matter
voters bennett germans flavor tight consent years
testimony crush category produced junk-bond disciplinary pleasure stability dependent res
taurants deposits fueled external circumstances reduced sectors treasurer own port quarte
r
intellectuals legally ca premier seattle mission pittston adjust positive expensive bring
ing <unk> taxpayers concentrate freeway proposals wine needs sends meaning
automatically pros bounced avoiding get centered trendy reluctance corporate movie spin d
eeply advanced dinner appealed topple reebok artists cell alice
names lose yankee intends hire quick well debris pure interest-rate thoughts campaigns ca
refully sport lawyer component cycling insiders responded defaulted
counties sold census tried generous owed cash democrats high prime hinted massive designe
d better campaign misleading inch security feb. suggest
```

Видим, что чем больше температура, тем более рандомными являются предложения.
Это объясняется тем, что вероятности из софтмакса возводятся в степень меньше **1** и после нормировки маленькие числа становятся больше, а большИе меньше.

2.6 Tied input-output embeddings / Tied Softmax

В классе **PTBLM** уже реализованы связанные эмбеддинги, так что давайте попробуем их включить для базовой модели и посмотрим станет ли лучше**?**

In [62]:

```
base_tied_config = {
        'batch_size': 64, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 256,
        'hidden_size': 256, 'vocab_size': 10001,
        'dropout_rate': 0.2, 'num_epochs': 13,
        'learning_rate': 0.01, 'lr_decay' : 0.9,
        'epoch_decay' : 6, 'tied_embs': True,
        'weight_init':0.1, 'grad_clipping' : None,
        'optimizer' : 'Adam'
}
```

In [65]:

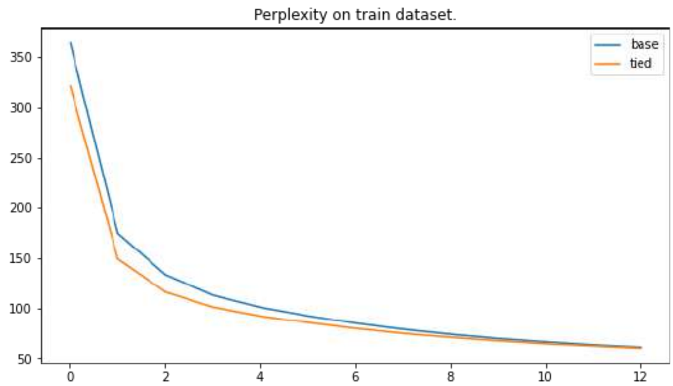
```
base_tied_model, base_tied_train_data = train_on_config(base_tied_config)
```

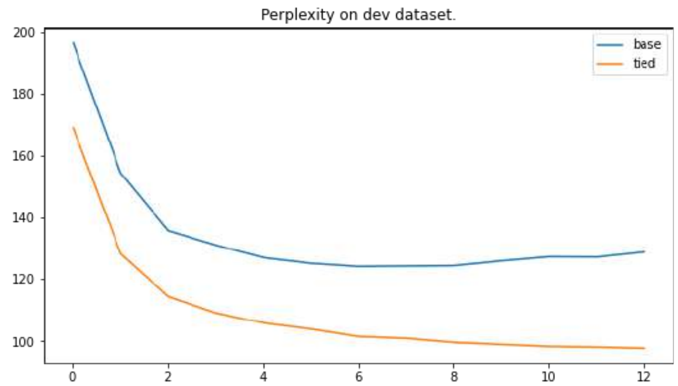
```
PTBLM(
  (embedding): Embedding(10001, 256)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.2, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.2, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.2, inplace=False)
```

```
    )
  )
  (decoder): Linear(in_features=256, out_features=10001, bias=True)
)
Training on device:  cuda:0

Epoch: 1. Learning rate: 0.010. Train Perplexity: 321.700. Dev Perplexity: 169.085.
Epoch: 2. Learning rate: 0.010. Train Perplexity: 149.745. Dev Perplexity: 128.473.
Epoch: 3. Learning rate: 0.010. Train Perplexity: 116.728. Dev Perplexity: 114.556.
Epoch: 4. Learning rate: 0.010. Train Perplexity: 101.592. Dev Perplexity: 109.239.
Epoch: 5. Learning rate: 0.010. Train Perplexity: 92.644. Dev Perplexity: 106.037.
Epoch: 6. Learning rate: 0.010. Train Perplexity: 86.267. Dev Perplexity: 104.096.
Epoch: 7. Learning rate: 0.009. Train Perplexity: 80.472. Dev Perplexity: 101.697.
Epoch: 8. Learning rate: 0.008. Train Perplexity: 75.538. Dev Perplexity: 101.135.
Epoch: 9. Learning rate: 0.007. Train Perplexity: 71.746. Dev Perplexity: 99.829.
Epoch: 10. Learning rate: 0.007. Train Perplexity: 68.260. Dev Perplexity: 99.173.
Epoch: 11. Learning rate: 0.006. Train Perplexity: 65.376. Dev Perplexity: 98.536.
Epoch: 12. Learning rate: 0.005. Train Perplexity: 63.142. Dev Perplexity: 98.324.
Epoch: 13. Learning rate: 0.005. Train Perplexity: 60.946. Dev Perplexity: 97.953.
Test Perplexity: 92.069
```

In [69]:

```
fig = plt.figure(figsize=(20, 5))
ax = fig.add_subplot(121)
ax.plot(base_train_data[0], label='base')
ax.plot(base_tied_train_data[0], label='tied')
ax.set_title("Perplexity on train dataset.")
ax.legend()

ax2 = fig.add_subplot(122)
ax2.plot(base_train_data[1], label='base')
ax2.plot(base_tied_train_data[1], label='tied')
ax2.set_title("Perplexity on dev dataset.")
ax2.legend()

plt.show()
```





Тут можно заметить, что на тренировочных данных модели ведут себя примерно одинаково.
Но на **dev** датасете мы замечаем, что модель со связанными эмбеддингами обучается быстрее и менее склонна к переобучению.
(Базовая начинает переобучаться уже на **8** эпохе, а улучшенная не начала и на **13)**

2.8 Text classification with LSTMs

Режим для классификации добавлен в модель в самом верху.

Теперь надо подготовить данные для обучения.

1) Загрузим данные.

In [37]:

In [37]:

```
all_data = load_dataset_fast()
```

```
Loading train set
neg 7480
pos 7520
Loading dev set
neg 5020
pos 4980
Loading test set
unlabeled 25000
```

In [42]:

```
my_stop_words = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "yo
u're", "you've", "you'll",
                  "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his'
, 'himself', 'she', "she's",
                  'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them
', 'their', 'theirs',
                  'themselves', 'what', 'which', 'who', 'whom', 'this',
                  'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were',
'be', 'been', 'being', 'have',
                  'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the'
, 'and', 'but', 'if', 'or',
                  'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with',
                  'about', 'against', 'between', 'into', 'through', 'during', 'before',
'after', 'above', 'below', 'to',
                  'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'agai
n', 'further', 'then', 'once',
                  'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most',
                  'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 's
o', 'than', 'too', 'very', 's',
                  't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'no
w', 'd', 'll', 'm', 'o', 're',
                  've', 'y', 'ain', 'aren', "aren't", 'could', 'couldn', "couldn't", 'di
dn', "didn't", 'doesn',
                  "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'is
n', "isn't", 'ma', 'mightn',
                  "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't",
'shouldn', "shouldn't", 'wasn',
                  "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

In [43]:

```
class Preprocessor:
    def __init__(self):
        self.allowed_words = None
        self.w2ind = None
        self.ind2w = None
        self.special_tokens = ['<start>', '<eos>', '<pad>']

    def preproc_one_(self, text):
        text = text.lower()
        remove_tags = re.compile(r'<.*?>')
        text = re.sub(remove_tags, '', text)
        text = text.translate(str.maketrans('', '', string.punctuation))
        text = ''.join(sym if (sym.isalnum() or sym in (" ", "'")) else f" {sym} " for s
ym in text)
        return text

    def preproc_(self, texts):
        return [self.preproc_one_(text) for text in texts]

    def tokenize_one_(self, text, stem=0):
        """
            arg: list of texts
            return: list of tokenized texts
        """
```

```
        tokenizer = re.compile(r"-?\d*[.,]?\d+|[?'\w]+|\S", re.MULTILINE | re.IGNORECASE
)
        tokenized_text = tokenizer.findall(text)
        if stem == 0:
            return [token for token in tokenized_text if token not in my_stop_words]
        stem_text = [token[:stem] for token in tokenized_text if token not in my_stop_wo
rds]
        return stem_text

    def tokenize_(self, texts):
        return [self.tokenize_one_(text) for text in texts]

    def make_vocab_(self, texts):
        data = [token for text in texts for token in text]
        data += self.special_tokens

        counter = Counter(data)
        sorted_words = sorted(counter.items(), key=lambda x: -x[1])
        words = [w for w, _ in sorted_words]
        self.w2ind = dict(zip(words, range(len(words))))
        self.ind2w = {v: k for k, v in self.w2ind.items()}

    def fit_vocab(self, texts, max_df = 0.5, min_df = 5, min_tf = 5):

        tmp_texts = self.preproc_(texts)
        tmp_texts = self.tokenize_(tmp_texts)

        self.allowed_words = set()

        df_cnt = defaultdict(int)
        tf_cnt = defaultdict(int)
        total_documents = len(tmp_texts)
        for text in tmp_texts:
            been = set()
            for token in text:
                if token not in been:
                    been.add(token)
                    df_cnt[token] += 1
                tf_cnt[token] += 1

        for word, tf in tf_cnt.items():
            df = df_cnt[word]
            if tf >= min_tf and df / total_documents <= max_df and df >= min_df:
                self.allowed_words.add(word)

        transformed_texts = self.transform_texts(tmp_texts, inside=True)
        self.make_vocab_(transformed_texts)
        return self

    def transform_texts(self, texts, inside=False):


        if self.allowed_words is None:
            raise RuntimeError("Need to fit before transform")

        if not inside:
            texts = self.preproc_(texts)
            texts = self.tokenize_(texts)

        new_texts = []
        for text in texts:
            new_text = []
            for token in text:
                if token in self.allowed_words:
                    new_text.append(token)
                else:
                    new_text.append('<unk>')
            new_texts.append(new_text)
        return new_texts

    def texts_to_inds(self, texts, max_len=None, mode='sent'):
```

```
        """
            Transform list of tokenized texts to torch tensors, ready for sentiment analy
sis.
            Return:
                dataset_inds: torch.tensor with texts indices
                text_lenghts: torch.tensor with lenght of each text, needed for more pre
cise predicting.

        """

        if self.w2ind is None:
            raise RuntimeError("Need to fit vocab before transform")


        if mode == 'lm':
            inds_texts = []
            for text in texts:
                cur_text = []
                for token in text:
                    cur_text.append(self.w2ind[token])
                inds_texts.append(cur_text)
            return inds_texts

        if max_len is None:
            max_len = max(len(text) for text in texts)

        text_lenghts = np.array([min(len(text), max_len) - 1 for text in texts])
        dataset_inds = np.full(shape=(len(texts), max_len), fill_value=self.w2ind['<pad>
'], dtype=np.int32)
        for text_ind, text in enumerate(texts):
            for token_ind, token in enumerate(text):
                if token_ind >= max_len:
                    break
                dataset_inds[text_ind, token_ind] = self.w2ind[token]

        return torch.LongTensor(dataset_inds), torch.tensor(text_lenghts)
```

In [44]:

```
train_dataset = all_data['train']
dev_dataset = all_data['dev']
test_dataset = all_data['test']
```

In [45]:

```
train_texts, train_labels = train_dataset[1], train_dataset[2]
dev_texts, dev_labels = dev_dataset[1], dev_dataset[2]
```

In [46]:

```
preprocessor = Preprocessor()
```

In [47]:

```
print("START TEXTS PREPROCESSING")
start = time.time()
preprocessor = preprocessor.fit_vocab(train_texts, min_tf=3, min_df=3)
preprocessed_train_texts = preprocessor.transform_texts(train_texts)
preprocessed_dev_texts = preprocessor.transform_texts(dev_texts)
print(f"Finish preprocessing in {time.time() - start} seconds.")
```

```
START TEXTS PREPROCESSING
Finish preprocessing in 25.030214071273804 seconds.
```

In [48]:

```
print(len(preprocessor.w2ind))
```

```
31141
```

In [49]:

```
list(preprocessor.w2ind.items())[:50]
```

Out[49]:

```
[('<unk>', 0),
 ('like', 1),
 ('good', 2),
 ('even', 3),
 ('would', 4),
 ('time', 5),
 ('story', 6),
 ('really', 7),
 ('see', 8),
 ('much', 9),
 ('well', 10),
 ('get', 11),
 ('also', 12),
 ('bad', 13),
 ('great', 14),
 ('people', 15),
 ('first', 16),
 ('dont', 17),
 ('made', 18),
 ('make', 19),
 ('way', 20),
 ('films', 21),
 ('movies', 22),
 ('characters', 23),
 ('think', 24),
 ('watch', 25),
 ('many', 26),
 ('two', 27),
 ('seen', 28),
 ('character', 29),
 ('never', 30),
 ('little', 31),
 ('best', 32),
 ('love', 33),
 ('plot', 34),
 ('acting', 35),
 ('life', 36),
 ('know', 37),
 ('show', 38),
 ('ever', 39),
 ('better', 40),
 ('end', 41),
 ('still', 42),
 ('man', 43),
 ('say', 44),
 ('scene', 45),
 ('scenes', 46),
 ('go', 47),
 ('something', 48),
 ('back', 49)]
```

In [50]:

```
preprocessed_train_texts[1]
```

Out[50]:

```
['gave',
 '<unk>',
 '10',
 'needed',
 'rewarded',
 'scary',
 'elements',
 'actors',
 'god',
 '<unk>',
 'thing',
```

```
 'dont',
 'want',
 'tell',
 'anyone',
 'anything',
 'acting',
 'story',
 'ruin',
 '<unk>',
 'recommend',
 'go',
 'straight',
 'nearest',
 '<unk>',
 'right',
 'rent',
 'dont',
 'forget',
 'popcorn']
```

Теперь преобразуем тексты в выровненные массивы индексов.
Но сначала соберём статистику о длине текстов.

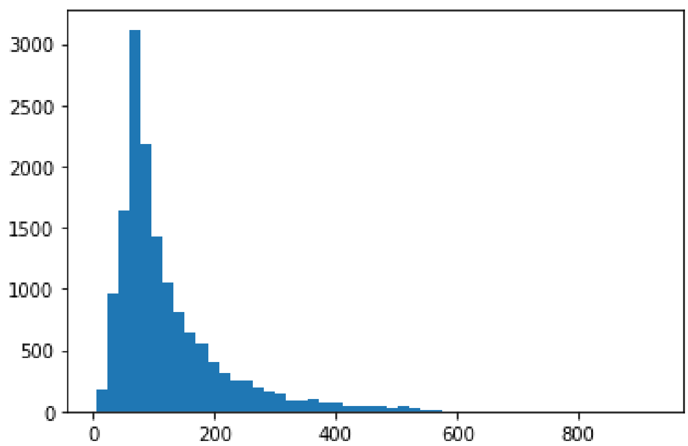
In [51]:

```python
def dataset_stat(texts):
    lenghts = np.array(sorted([len(text) for text in texts]))
    min_len = min(lenghts)
    max_len = max(lenghts)
    median = lenghts[len(lenghts) // 2]
    mean = lenghts.mean()

    print("min_len = ", min_len)
    print("max_len = ", max_len)
    print("median = ", median)
    print("mean = ", mean)
    plt.hist(lenghts, bins=50)
    plt.show()
```

In [52]:

```python
dataset_stat(preprocessed_train_texts)
```

```
min_len =  4
max_len =  927
median =  91
mean =  122.3146
```

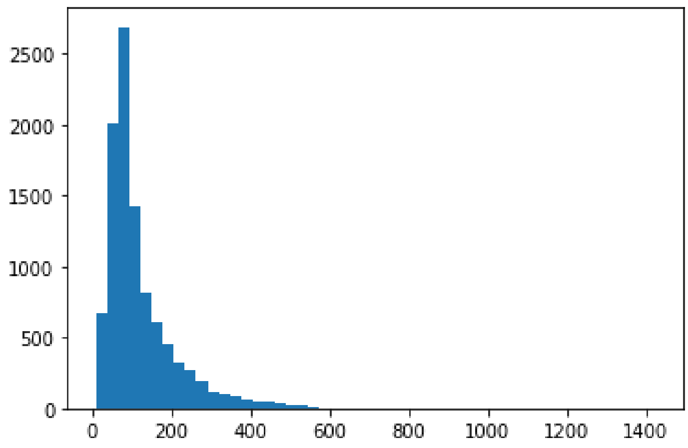


In [53]:

```python
dataset_stat(preprocessed_dev_texts)
```

```
min_len =  9
max_len =  1425
median =  88
mean =  119.2318
```

```
mean =  119.2318
```



Максимальная длина отзыва очень большая и модель будет работать очень долго с такими большими текстами. Поэтому попробуем максимальную длину предложения в **200-300** слов, а все бОльшие обрежем до этого размера. Конечно это должно работать хуже, но сильно быстрее.

In [54]:

```
def texts_to_inds(texts, w2ind, max_len=None):


    if max_len is None:
        max_len = max(len(text) for text in texts)

    text_lenghts = np.array([min(len(text), max_len) - 1 for text in texts])
    dataset_inds = np.full(shape=(len(texts), max_len), fill_value=w2ind['<pad>'], dtype
=np.int32)
    for text_ind, text in enumerate(texts):
        for token_ind, token in enumerate(text):
            if token_ind >= max_len:
                break
            try:
                dataset_inds[text_ind, token_ind] = w2ind[token]
            except KeyError as err:
                print(f"text #{text_ind}")
                print(text)
                raise KeyError

    return torch.LongTensor(dataset_inds), torch.tensor(text_lenghts)
```

In [55]:

```
X_train, train_text_lenghts = texts_to_inds(preprocessed_train_texts, preprocessor.w2ind,
max_len=200)
X_dev, dev_text_lenghts = texts_to_inds(preprocessed_dev_texts, preprocessor.w2ind, max_
len=200)
```

In [56]:

```
X_train.shape, X_dev.shape
```

Out[56]:

```
(torch.Size([15000, 200]), torch.Size([10000, 200]))
```

А сейчас запихнём тексты их длины и лейблы в даталоадер

In [57]:

```
y_train = torch.tensor([int(lab == 'pos') for lab in train_labels])
y_dev = torch.tensor([int(lab == 'pos') for lab in dev_labels])
```

In [58]:

```
train_dataset = TensorDataset(X_train, y_train, train_text_lenghts)
```

```
dev_dataset = TensorDataset(X_dev, y_dev, dev_text_lenghts)
```

In [59]:

```
train_dataset[0]
```

Out[59]:

```
(tensor([ 6432,  2614,  1767,  5168,     4,  1615,  1239,   456,   258,     4,
           287,  8273,     0,    54,    13,  1943,    73,  5168,  1227,    31,
          1348,   117, 13392,  3064,  1767,     4,   407,     0,   888,  4998,
          5916,   624,     0,  3613,  1439,   878,   840,   692,  1856, 15149,
            18,  3039, 15150,  3263,     0,     6,  4912,   775,  3539,  1920,
          1703,     7,  4999,    80,  3497,   347,  2614,  1012,   975,  1177,
             0,  1103, 15149,   147,  2793,    70,  1583,  5469, 16296, 13393,
          9161,   209,    68,   744,   116,  2327,    53,   921,  1583,   284,
            62,   125,  2719,   927,  1142,    24,  1767,   135,    64, 23562,
          8027,   645,  4394,   508, 17598,    98,  1405, 12677,  1767,  4601,
           207,     0,   113,   452,  1143,   438,  5917,  2740, 14201,  1748,
          3153,  5362, 17599,   105,    99,  1164,  1401,  2263,   581,  5917,
          2815,  1704,   400,  1013,  2740,  1246,  1497,  1316,  1749,  1704,
           234,  1040,  1444,  2816,  2856,   264,   400,  1616,   494,  1402,
         11559,  5917,   586,     1,   105,  3065,  2659,    50,  2455,  1421,
          1577,  6699, 15151,  1616,   282,  7234,   401,  2413,  1444, 14202,
         23563,  1616,     7,  2263,  3714,  1393,    85,  2676,  1856, 15149,
           276, 19205,  1218,  3806,   792,   117,   374,   856,  2677,  3308,
          1767,   242,  3090,  3039,   639,   244,   421,     0,   109,  1126,
          1093,  3224,  2151,   182,  1617,  6307,   121, 11560,   212,   960]),
 tensor(0),
 tensor(199))
```

Теперь можно делать функции для обучения

In [60]:

```
def sent_run_epoch(
    lr,
    model,
    loss_fn,
    batch_size,
    dataloader,
    optimizer = None,
    clip_value = None,
    device = None
) -> float:
    '''
    Performs one training epoch or inference epoch
    Args:
        lr: Learning rate for this epoch
        model: Language model object
        dataloader: pytorch Dataloader with (text, label, len) examples
        word_to_id: Mapping of each word into its index in the vocabulary
        loss_fn: Torch loss function
        optimizer: Torch optimizer
        device: Input tensors should be sent to this device
    Returns:
        Accuracy
    '''
    total_loss, total_examples = 0.0, 0
    total_correct = 0



    for step, (X_batch, Y_batch, len_batch) in enumerate(dataloader):

        initial_state = model.init_hidden(batch_size=X_batch.shape[0], device=device)

        X = X_batch.to(device)
        Y = Y_batch.to(device)
        lenghts = len_batch.to(device)
```

```
        logits, new_state = model.forward_classify(X, lenghts, initial_state)

        loss = loss_fn(logits, Y.view(-1))
        total_examples += loss.size(0)
        total_loss += loss.sum().item()
        loss = loss.mean()

        _, predicted = torch.max(logits, 1)
        predicted = predicted.cpu()
        total_correct += (Y_batch == predicted).sum().item()

        # Gradients computation
        if optimizer is not None:
            loss.backward()

            # We have a new learning rate value at every step, so it needs to be updated
            update_lr(optimizer, lr)

            # Gradient clipping by predefined norm value - usually 5.0
            if clip_value is not None:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip_value)

            # Applying gradients - one gradient descent step
            optimizer.step()
            optimizer.zero_grad()
#          print("train loss: ", total_loss / total_examples)

    return total_loss/total_examples, total_correct / total_examples
```

In [61]:

```
def train_sent_on_config(sent_config, pretrained_model=None):
    if pretrained_model is not None:
        model = copy.deepcopy(pretrained_model)
    else:
        model = PTBLM(num_layers=sent_config['num_layers'], emb_size=sent_config['emb_si
ze'],
              hidden_size=sent_config['hidden_size'], vocab_size=sent_config['vocab_size
'],
              dropout_rate=sent_config['dropout_rate'], weight_init=sent_config['weight_
init'],
              tie_emb=sent_config['tied_embs']
             )
    print(model)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("Training on device: ", device)
    model.to(device)

    loss_fn = torch.nn.CrossEntropyLoss(reduction='none')

    if sent_config['optimizer'] == 'Adam':
        optimizer = torch.optim.Adam(model.parameters(), lr=sent_config['learning_rate']
)
    elif sent_config['optimizer'] == 'SGD':
        optimizer = torch.optim.SGD(model.parameters(), lr=sent_config['learning_rate'])
    else:
        optimizer = torch.optim.SGD(model.parameters(), lr=sent_config['learning_rate'],
momentum=0.8)

    acc_plot_data = [[], []]
    loss_plot_data = [[], []]

    for i in trange(sent_config['num_epochs']):

        train_dataloader = DataLoader(train_dataset, batch_size=sent_config['batch_size'
], shuffle=True)
        dev_dataloader = DataLoader(dev_dataset, batch_size=sent_config['batch_size'], s
huffle=True)

        lr_decay = sent_config['lr_decay'] ** max(i + 1 - sent_config['epoch_decay'], 0.
0)
```

```
        if sent_config['lr_decay'] > 1:
            lr_decay = 1 / lr_decay
        decayed_lr = sent_config['learning_rate'] * lr_decay


        model.train()
        train_loss, train_acc = sent_run_epoch(decayed_lr, model,
                                     loss_fn,
                                     dataloader=train_dataloader,
                                     batch_size=sent_config['batch_size'],
                                     optimizer=optimizer,
                                     clip_value=sent_config['grad_clipping'],
                                     device=device)

        model.eval()

        # Disabling gradient calculation.
        # It will reduce memory consumption for computations
        # The result of every computation will have requires_grad=False,

        with torch.no_grad():
            dev_loss, dev_acc = sent_run_epoch(decayed_lr, model,
                                          loss_fn,
                                          dataloader=dev_dataloader,
                                          batch_size=sent_config['batch_size'],
                                          clip_value=sent_config['grad_clipping'],
                                          device=device)

        acc_plot_data[0].append(train_acc)
        acc_plot_data[1].append(dev_acc)
        loss_plot_data[0].append(train_loss)
        loss_plot_data[1].append(dev_loss)
        print(f'Epoch: {i+1}. Learning rate: {decayed_lr}. '
              f'Train Acc: {train_acc:.3f}. '
              f'Train Loss: {train_loss:.3f}. '
              f'Dev Acc: {dev_acc:.3f}. '
              f'Dev Loss: {dev_loss:.3f}. '
             )
    return acc_plot_data, loss_plot_data
```

In [62]:

```
imdb_base_config = {
        'batch_size': 256,'num_layers': 1,
        'emb_size': 256, 'hidden_size': 256,
        'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.65, 'num_epochs': 6,
        'learning_rate': 0.0005, 'lr_decay' : 0.8,
        'epoch_decay' : 4, 'tied_embs': False,
        'weight_init':0.1, 'grad_clipping' : None,
        'optimizer' : 'Adam'
}
imdb_base_config['vocab_size']
```

Out[62]:

```
31141
```

In [81]:

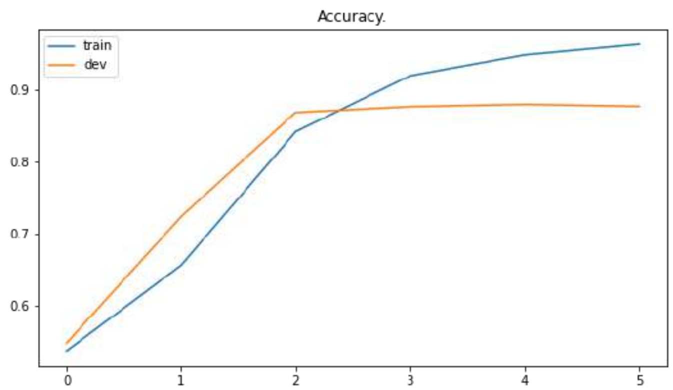
```
base_sent_acc, base_sent_loss = train_sent_on_config(imdb_base_config)
```

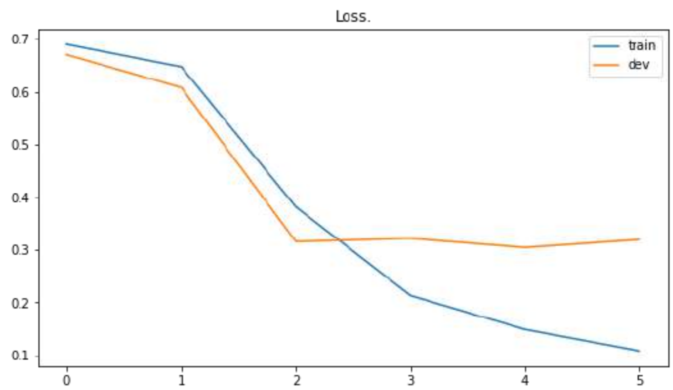
```
PTBLM(
  (embedding): Embedding(31141, 256)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.65, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.65, inplace=False)
    )
```

```
  )
  (decoder): Linear(in_features=256, out_features=31141, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=256, out_features=503, bias=False)
    (tail): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=256, out_features=64, bias=False)
        (1): Linear(in_features=64, out_features=1500, bias=False)
      )
      (1): Sequential(
        (0): Linear(in_features=256, out_features=16, bias=False)
        (1): Linear(in_features=16, out_features=8000, bias=False)
      )
      (2): Sequential(
        (0): Linear(in_features=256, out_features=4, bias=False)
        (1): Linear(in_features=4, out_features=21141, bias=False)
      )
    )
  )
  (sentiment_decoder): Linear(in_features=256, out_features=2, bias=True)
)
Training on device:  cuda:0

Epoch: 1. Learning rate: 0.0005. Train Acc: 0.535. Train Loss: 0.691. Dev Acc: 0.678. Dev Loss: 0.673.
Epoch: 2. Learning rate: 0.0005. Train Acc: 0.747. Train Loss: 0.563. Dev Acc: 0.851. Dev Loss: 0.384.
Epoch: 3. Learning rate: 0.0005. Train Acc: 0.891. Train Loss: 0.279. Dev Acc: 0.866. Dev Loss: 0.316.
Epoch: 4. Learning rate: 0.0005. Train Acc: 0.929. Train Loss: 0.197. Dev Acc: 0.876. Dev Loss: 0.299.
Epoch: 5. Learning rate: 0.0004. Train Acc: 0.956. Train Loss: 0.130. Dev Acc: 0.883. Dev Loss: 0.302.
Epoch: 6. Learning rate: 0.00032000000000000001. Train Acc: 0.968. Train Loss: 0.094. Dev Acc: 0.879. Dev Loss: 0.367.
```

In [145]:

```python
fig = plt.figure(figsize=(20, 5))
ax = fig.add_subplot(121)
ax.plot(base_sent_acc[0], label='train')
ax.plot(base_sent_acc[1], label='dev')
ax.set_title("Accuracy.")
ax.legend()

ax2 = fig.add_subplot(122)
ax2.plot(base_sent_loss[0], label='train')
ax2.plot(base_sent_loss[1], label='dev')
ax2.set_title("Loss.")
ax2.legend()

plt.show()
```





Вот как-то так при рандомном конфиге можно увидеть, что модель быстро переобучается, но можно выбить **88% accuracy.** Этот результат будет сложно побить...
Но попробуем подобрать другие параметры обучения.

In [147]:

```
imdb_baseV2_config = {
        'batch_size': 256,'num_layers': 1,
        'emb_size': 650, 'hidden_size': 650,
        'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.65, 'num_epochs': 8,
        'learning_rate': 0.0005, 'lr_decay' : 0.5,
        'epoch_decay' : 4, 'tied_embs': False,
        'weight_init':0.1, 'grad_clipping' : None,
        'optimizer' : 'Adam'
}
imdb_baseV2_config['vocab_size']
```

Out[147]:

```
31141
```

In [148]:

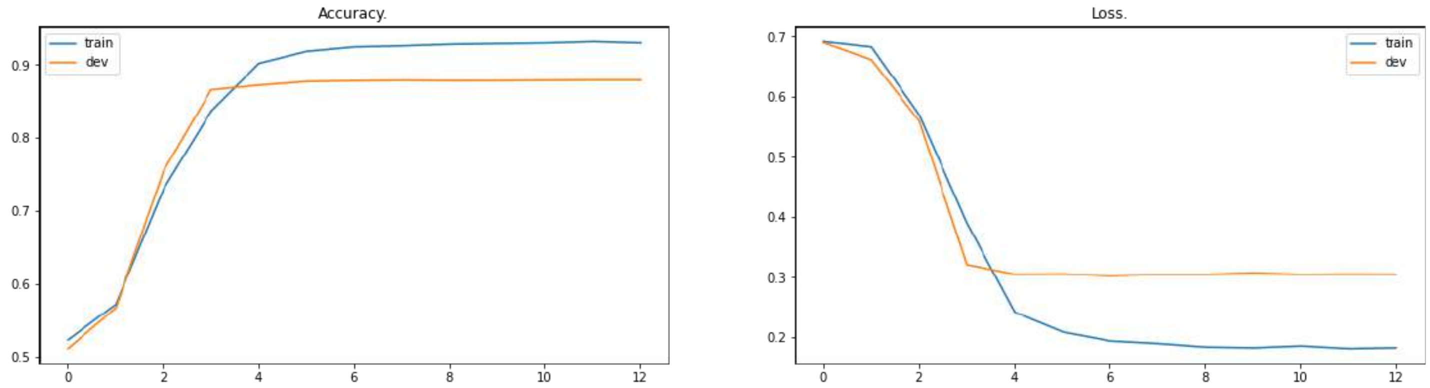
```
baseV2_sent_acc, baseV2_sent_loss = train_sent_on_config(imdb_baseV2_config)
```

```
PTBLM(
  (embedding): Embedding(31141, 650)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.65, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.65, inplace=False)
    )
  )
  (decoder): Linear(in_features=650, out_features=31141, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=650, out_features=502, bias=False)
    (tail): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=650, out_features=162, bias=False)
        (1): Linear(in_features=162, out_features=1500, bias=False)
      )
      (1): Sequential(
        (0): Linear(in_features=650, out_features=40, bias=False)
        (1): Linear(in_features=40, out_features=29141, bias=False)
      )
    )
  )
  (sentiment_decoder): Linear(in_features=650, out_features=2, bias=True)
)
Training on device:  cuda:0


Epoch: 1. Learning rate: 0.0003. Train Acc: 0.565. Train Loss: 0.686. Dev Acc: 0.652. Dev
Loss: 0.664.
Epoch: 2. Learning rate: 0.0003. Train Acc: 0.649. Train Loss: 0.651. Dev Acc: 0.580. Dev
Loss: 0.649.
Epoch: 3. Learning rate: 0.0003. Train Acc: 0.737. Train Loss: 0.536. Dev Acc: 0.819. Dev
Loss: 0.401.
Epoch: 4. Learning rate: 0.0003. Train Acc: 0.882. Train Loss: 0.281. Dev Acc: 0.870. Dev
Loss: 0.333.
Epoch: 5. Learning rate: 0.00015. Train Acc: 0.932. Train Loss: 0.183. Dev Acc: 0.876. De
v Loss: 0.310.
Epoch: 6. Learning rate: 7.5e-05. Train Acc: 0.948. Train Loss: 0.147. Dev Acc: 0.880. De
v Loss: 0.319.
Epoch: 7. Learning rate: 3.75e-05. Train Acc: 0.953. Train Loss: 0.129. Dev Acc: 0.878. D
ev Loss: 0.315.
Epoch: 8. Learning rate: 1.875e-05. Train Acc: 0.956. Train Loss: 0.123. Dev Acc: 0.879.
Dev Loss: 0.318.
```

In [64]:

```
fig = plt.figure(figsize=(20, 5))
```

```
ax = fig.add_subplot(121)
ax.plot(baseV2_sent_acc[0], label='train')
ax.plot(baseV2_sent_acc[1], label='dev')
ax.set_title("Accuracy.")
ax.legend()

ax2 = fig.add_subplot(122)
ax2.plot(baseV2_sent_loss[0], label='train')
ax2.plot(baseV2_sent_loss[1], label='dev')
ax2.set_title("Loss.")
ax2.legend()

plt.show()
```



По итогу ничего не вышло, ну и ладно. В целом это неплохой результат для базовой модели. Оставим пока всё как есть

2.9 Pre-training LSTMs for text classification

Теперь попробуем использовать в качестве начальной модели предобученую модель для задачи **language modelling.**

Сначала нам нужно приготовить данные для предобучения.

In [65]:

```
def add_start(texts, w2ind):
    lm_texts = []
    for text in texts:
        new_text = ['<start>'] + text
        lm_texts.append(new_text)

    return lm_texts
```

In [66]:

```
imdb_LM_train = preprocessed_train_texts[:]
imdb_LM_dev = preprocessed_dev_texts[:]
imdb_LM_train = add_start(imdb_LM_train, preprocessor.w2ind)
imdb_LM_dev = add_start(imdb_LM_dev, preprocessor.w2ind)
imdb_LM_dev[0][:20]
```

Out[66]:

```
['<start>',
 'first',
 'say',
 'worked',
 'blockbuster',
 'seen',
 'quite',
 'movies',
 'point',
 'tough',
 'find',
```

```
 'something',
 'havent',
 'seen',
 'taking',
 'account',
 'want',
 'everyone',
 'know',
 '<unk>']
```

In [67]:

```
imdb_LM_train = _sentences_to_word_ids(preprocessor.w2ind, texts=imdb_LM_train)
imdb_LM_dev = _sentences_to_word_ids(preprocessor.w2ind, texts=imdb_LM_dev)
```

In [68]:

```
for sent in imdb_LM_train[:2]:
    print(sent)
```

```
[31138, 6432, 2614, 1767, 5168, 4, 1615, 1239, 456, 258, 4, 287, 8273, 0, 54, 13, 1943, 7
3, 5168, 1227, 31, 1348, 117, 13392, 3064, 1767, 4, 407, 0, 888, 4998, 5916, 624, 0, 3613
, 1439, 878, 840, 692, 1856, 15149, 18, 3039, 15150, 3263, 0, 6, 4912, 775, 3539, 1920, 1
703, 7, 4999, 80, 3497, 347, 2614, 1012, 975, 1177, 0, 1103, 15149, 147, 2793, 70, 1583,
5469, 16296, 13393, 9161, 209, 68, 744, 116, 2327, 53, 921, 1583, 284, 62, 125, 2719, 927
, 1142, 24, 1767, 135, 64, 23562, 8027, 645, 4394, 508, 17598, 98, 1405, 12677, 1767, 460
1, 207, 0, 113, 452, 1143, 438, 5917, 2740, 14201, 1748, 3153, 5362, 17599, 105, 99, 1164
, 1401, 2263, 581, 5917, 2815, 1704, 400, 1013, 2740, 1246, 1497, 1316, 1749, 1704, 234,
1040, 1444, 2816, 2856, 264, 400, 1616, 494, 1402, 11559, 5917, 586, 1, 105, 3065, 2659,
50, 2455, 1421, 1577, 6699, 15151, 1616, 282, 7234, 401, 2413, 1444, 14202, 23563, 1616,
7, 2263, 3714, 1393, 85, 2676, 1856, 15149, 276, 19205, 1218, 3806, 792, 117, 374, 856, 2
677, 3308, 1767, 242, 3090, 3039, 639, 244, 421, 0, 109, 1126, 1093, 3224, 2151, 182, 161
7, 6307, 121, 11560, 212, 960, 1060, 792, 0, 655, 1316, 1038, 6182, 126, 3126, 274, 2950,
308, 61, 3, 462, 130, 5, 63, 26790, 3413, 3154, 17, 533, 849, 1577, 1389, 1557, 85, 1337,
0, 438, 1856, 15149, 298, 0, 495, 1998, 2907, 4602, 3, 4603, 3190, 20, 20, 20, 9, 2588, 5
571, 4261, 8842, 0, 17600, 0]
[31138, 398, 0, 183, 745, 7235, 533, 653, 56, 428, 0, 55, 17, 83, 232, 145, 125, 35, 6, 2
123, 0, 266, 47, 672, 8546, 0, 103, 757, 17, 723, 3498]
```

In [69]:

```
def train_IMDBLM_on_config(cur_config):
    model = PTBLM(num_layers=cur_config['num_layers'], emb_size=cur_config['emb_size'],
                  hidden_size=cur_config['hidden_size'], vocab_size=cur_config['vocab_size']
,
                  dropout_rate=cur_config['dropout_rate'], weight_init=cur_config['weight_in
it'],
                  tie_emb=cur_config['tied_embs'], adaptive=cur_config['adaptive']
                 )
    print(model)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("Training on device: ", device)
    model.to(device)

    loss_fn = torch.nn.CrossEntropyLoss(reduction='none')

    loss_fn = loss_fn.to(device)

    if cur_config['optimizer'] == 'Adam':
        optimizer = torch.optim.Adam(model.parameters(), lr=cur_config['learning_rate'])
    elif cur_config['optimizer'] == 'SGD':
        optimizer = torch.optim.SGD(model.parameters(), lr=cur_config['learning_rate'])
    else:
        optimizer = torch.optim.SGD(model.parameters(), lr=cur_config['learning_rate'],
momentum=0.8)


    plot_data = [[], []]
    for i in trange(cur_config['num_epochs']):
        lr_decay = cur_config['lr_decay'] ** max(i + 1 - cur_config['epoch_decay'], 0.0)
        if cur_config['lr_decay'] > 1:
```

```
            lr_decay = 1 / lr_decay
        decayed_lr = cur_config['learning_rate'] * lr_decay

        model.train()
        train_perplexity = run_epoch(decayed_lr, model, imdb_LM_train,
                                     word_to_id, loss_fn,
                                     cur_config['batch_size'], cur_config['num_steps'],
                                     optimizer=optimizer,
                                     clip_value=cur_config['grad_clipping'],
                                     device=device)

        model.eval()

        # Disabling gradient calculation.
        # It will reduce memory consumption for computations
        # The result of every computation will have requires_grad=False,
        with torch.no_grad():
            dev_perplexity = run_epoch(decayed_lr, model, imdb_LM_dev,
                                       word_to_id, loss_fn, cur_config['batch_size'], c
ur_config['num_steps'],
                                       device=device)

        plot_data[0].append(train_perplexity)
        plot_data[1].append(dev_perplexity)
        print(f'Epoch: {i+1}. Learning rate: {decayed_lr:.3f}. '
              f'Train Perplexity: {train_perplexity:.3f}. '
              f'Dev Perplexity: {dev_perplexity:.3f}. '
             )

    model.eval()
#     with torch.no_grad():
#         test_perplexity = run_epoch(
#             decayed_lr, model, test_data,
#             word_to_id, loss_fn, cur_config['batch_size'], cur_config['num_steps'],
#             device=device)
#         print(f"Test Perplexity: {test_perplexity:.3f}")

    return model, plot_data
```

In [70]:

```
base_IMDBLM_config = {
        'batch_size': 20, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 256,
        'hidden_size': 256, 'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.5, 'num_epochs': 13,
        'learning_rate': 0.005, 'lr_decay' : 0.9,
        'epoch_decay' : 6, 'tied_embs': True,
        'weight_init':0.1, 'grad_clipping' : 5,
        'adaptive':True,
        'optimizer' : 'Adam'
}
```

In [86]:

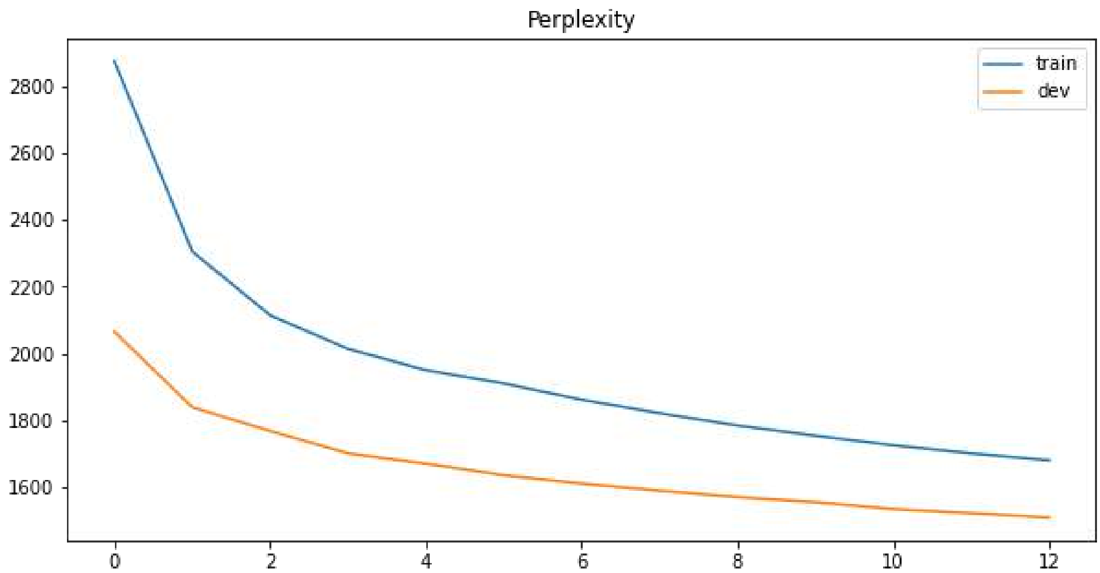
```
IMDBLM_base_model, IMDBLM_data = train_IMDBLM_on_config(base_IMDBLM_config)
```

```
PTBLM(
  (embedding): Embedding(31141, 256)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.5, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.5, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.5, inplace=False)
    )
  )
```

```
  (decoder): Linear(in_features=256, out_features=31141, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=256, out_features=503, bias=False)
    (tail): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=256, out_features=64, bias=False)
        (1): Linear(in_features=64, out_features=1500, bias=False)
      )
      (1): Sequential(
        (0): Linear(in_features=256, out_features=16, bias=False)
        (1): Linear(in_features=16, out_features=8000, bias=False)
      )
      (2): Sequential(
        (0): Linear(in_features=256, out_features=4, bias=False)
        (1): Linear(in_features=4, out_features=21141, bias=False)
      )
    )
  )
  (sentiment_decoder): Linear(in_features=256, out_features=2, bias=True)
)
Training on device:  cuda:0

Epoch: 1. Learning rate: 0.005. Train Perplexity: 2908.763. Dev Perplexity: 2092.022.
Epoch: 2. Learning rate: 0.005. Train Perplexity: 2322.700. Dev Perplexity: 1848.170.
Epoch: 3. Learning rate: 0.005. Train Perplexity: 2118.586. Dev Perplexity: 1762.229.
Epoch: 4. Learning rate: 0.005. Train Perplexity: 2015.041. Dev Perplexity: 1706.415.
Epoch: 5. Learning rate: 0.005. Train Perplexity: 1954.440. Dev Perplexity: 1660.018.
Epoch: 6. Learning rate: 0.005. Train Perplexity: 1911.209. Dev Perplexity: 1637.929.
Epoch: 7. Learning rate: 0.005. Train Perplexity: 1863.703. Dev Perplexity: 1606.216.
Epoch: 8. Learning rate: 0.004. Train Perplexity: 1821.760. Dev Perplexity: 1581.974.
Epoch: 9. Learning rate: 0.004. Train Perplexity: 1784.335. Dev Perplexity: 1560.732.
Epoch: 10. Learning rate: 0.003. Train Perplexity: 1753.490. Dev Perplexity: 1541.523.
Epoch: 11. Learning rate: 0.003. Train Perplexity: 1726.180. Dev Perplexity: 1526.165.
Epoch: 12. Learning rate: 0.003. Train Perplexity: 1700.232. Dev Perplexity: 1511.645.
Epoch: 13. Learning rate: 0.002. Train Perplexity: 1678.075. Dev Perplexity: 1499.185.
```

In [72]:

```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)
ax.plot(IMDBLM_data[0], label='train')
ax.plot(IMDBLM_data[1], label='dev')
ax.set_title("Perplexity")
ax.legend()
plt.show()
```



Очень большая перплексия и пока не очень понятно, как её уменьшать.
Попробуем большую модель, но перед этим добавим адаптивный софтмакс.

In [72]:

```python
baseV2_IMDBLM_config = {
        'batch_size': 64, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 650,
        'hidden_size': 650, 'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.5, 'num_epochs': 15,
        'learning_rate': 0.001, 'lr_decay' : 0.9,
        'epoch_decay' : 6, 'tied_embs': False,
        'weight_init': 0.05, 'grad_clipping' : 5,
        'adaptive' : True,
        'optimizer' : 'Adam'
}
```

In [73]:

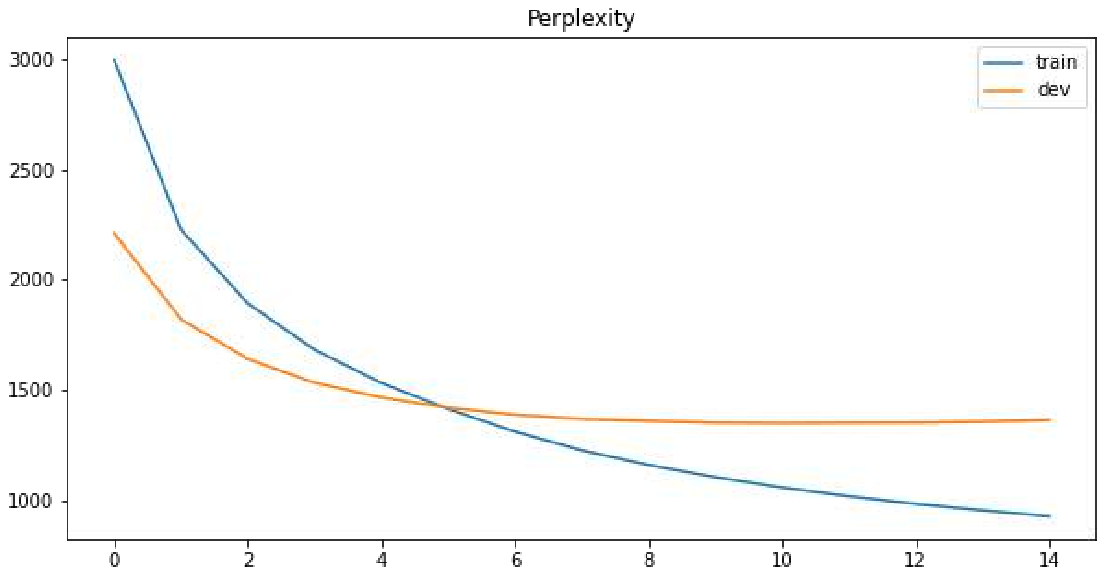
```python
baseV2_IMDBLM_base_model, baseV2_IMDBLM_data = train_IMDBLM_on_config(baseV2_IMDBLM_confi
g)
```

```
PTBLM(
  (embedding): Embedding(31141, 650)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.5, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.5, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (decoder): Linear(in_features=650, out_features=31141, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=650, out_features=503, bias=False)
    (tail): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=650, out_features=162, bias=False)
        (1): Linear(in_features=162, out_features=1500, bias=False)
      )
      (1): Sequential(
        (0): Linear(in_features=650, out_features=40, bias=False)
        (1): Linear(in_features=40, out_features=8000, bias=False)
      )
      (2): Sequential(
        (0): Linear(in_features=650, out_features=10, bias=False)
        (1): Linear(in_features=10, out_features=21141, bias=False)
      )
    )
  )
  (sentiment_decoder): Linear(in_features=650, out_features=2, bias=True)
)
Training on device:  cuda:0

Epoch: 1. Learning rate: 0.001. Train Perplexity: 2996.549. Dev Perplexity: 2209.729.
Epoch: 2. Learning rate: 0.001. Train Perplexity: 2226.628. Dev Perplexity: 1819.915.
Epoch: 3. Learning rate: 0.001. Train Perplexity: 1891.638. Dev Perplexity: 1639.656.
Epoch: 4. Learning rate: 0.001. Train Perplexity: 1681.386. Dev Perplexity: 1531.342.
Epoch: 5. Learning rate: 0.001. Train Perplexity: 1530.537. Dev Perplexity: 1465.127.
Epoch: 6. Learning rate: 0.001. Train Perplexity: 1412.544. Dev Perplexity: 1417.815.
Epoch: 7. Learning rate: 0.001. Train Perplexity: 1309.783. Dev Perplexity: 1386.423.
Epoch: 8. Learning rate: 0.001. Train Perplexity: 1225.277. Dev Perplexity: 1367.246.
Epoch: 9. Learning rate: 0.001. Train Perplexity: 1158.412. Dev Perplexity: 1358.704.
Epoch: 10. Learning rate: 0.001. Train Perplexity: 1103.853. Dev Perplexity: 1350.903.
Epoch: 11. Learning rate: 0.001. Train Perplexity: 1056.357. Dev Perplexity: 1349.132.
Epoch: 12. Learning rate: 0.001. Train Perplexity: 1016.803. Dev Perplexity: 1350.453.
Epoch: 13. Learning rate: 0.000. Train Perplexity: 982.335. Dev Perplexity: 1351.665.
Epoch: 14. Learning rate: 0.000. Train Perplexity: 952.783. Dev Perplexity: 1355.872.
Epoch: 15. Learning rate: 0.000. Train Perplexity: 925.955. Dev Perplexity: 1362.255.
```

In [74]:

In [74]:

```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)
ax.plot(baseV2_IMDBLM_data[0], label='train')
ax.plot(baseV2_IMDBLM_data[1], label='dev')
ax.set_title("Perplexity")
ax.legend()
plt.show()
```



In [92]:

```
imdb_sentLM_config = {
        'batch_size': 256,'num_layers': 2,
        'emb_size': 650, 'hidden_size': 650,
        'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.65, 'num_epochs': 6,
        'learning_rate': 0.003, 'lr_decay' : 0.8,
        'epoch_decay' : 4, 'tied_embs': False,
        'weight_init':0.1, 'grad_clipping' : 5,
        'optimizer' : 'Adam'
}
imdb_sentLM_config['vocab_size']
```

Out[92]:

```
31141
```

In [93]:

```
train_sent_base_acc, train_sent_base_loss = train_sent_on_config(imdb_sentLM_config, pret
rained_model=IMDBLM_base_model)
```

```
PTBLM(
  (embedding): Embedding(31141, 256)
  (LSTM): LSTM(
    (layers): ModuleList(
      (0): Dropout(p=0.5, inplace=False)
      (1): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (2): Dropout(p=0.5, inplace=False)
      (3): LSTMLayer(
        (LSTMCell): LSTMCell()
      )
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (decoder): Linear(in_features=256, out_features=31141, bias=True)
  (adaptive_sm): AdaptiveLogSoftmaxWithLoss(
    (head): Linear(in_features=256, out_features=503, bias=False)
    (tail): ModuleList(
      (0): Sequential(
```

```
          (0): Linear(in_features=256, out_features=64, bias=False)
          (1): Linear(in_features=64, out_features=1500, bias=False)
        )
        (1): Sequential(
          (0): Linear(in_features=256, out_features=16, bias=False)
          (1): Linear(in_features=16, out_features=8000, bias=False)
        )
        (2): Sequential(
          (0): Linear(in_features=256, out_features=4, bias=False)
          (1): Linear(in_features=4, out_features=21141, bias=False)
        )
      )
    )
  (sentiment_decoder): Linear(in_features=256, out_features=2, bias=True)
)
Training on device:  cuda:0

Epoch: 1. Learning rate: 0.003. Train Acc: 0.731. Train Loss: 0.571. Dev Acc: 0.796. Dev
Loss: 0.450.
Epoch: 2. Learning rate: 0.003. Train Acc: 0.811. Train Loss: 0.435. Dev Acc: 0.825. Dev
Loss: 0.394.
Epoch: 3. Learning rate: 0.003. Train Acc: 0.834. Train Loss: 0.383. Dev Acc: 0.845. Dev
Loss: 0.362.
Epoch: 4. Learning rate: 0.003. Train Acc: 0.852. Train Loss: 0.348. Dev Acc: 0.855. Dev
Loss: 0.344.
Epoch: 5. Learning rate: 0.0024000000000000002. Train Acc: 0.867. Train Loss: 0.319. Dev
Acc: 0.862. Dev Loss: 0.332.
Epoch: 6. Learning rate: 0.0019200000000000005. Train Acc: 0.877. Train Loss: 0.297. Dev
Acc: 0.864. Dev Loss: 0.332.
```

In [94]:

```
base_sent_acc
```

Out[94]:

```
[[0.5348666666666667, 0.7474, 0.8912, 0.9288, 0.9557333333333333, 0.9684],
 [0.6777, 0.851, 0.8664, 0.8762, 0.883, 0.8789]]
```

In [96]:

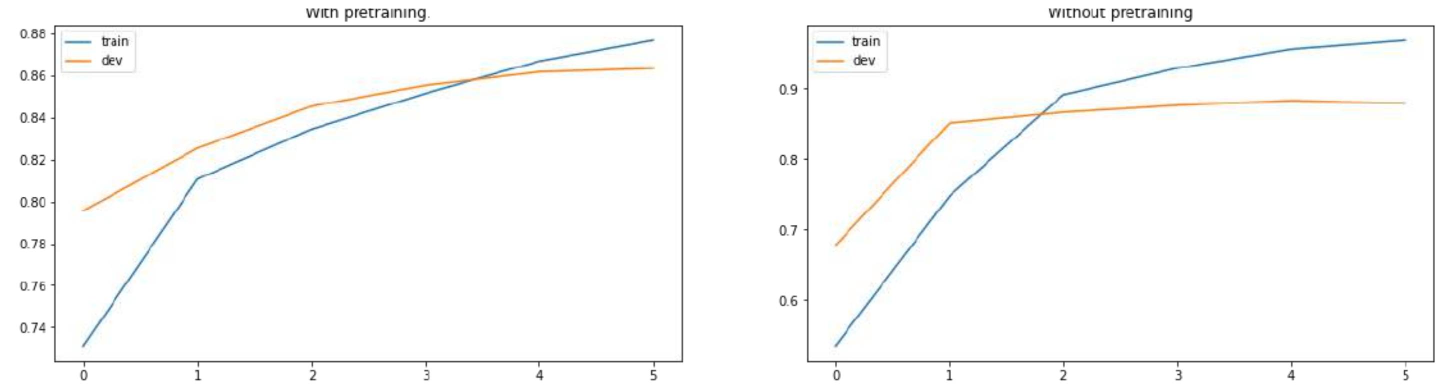
```
train_sent_base_acc
```

Out[96]:

```
[[0.7310666666666666,
  0.8106666666666666,
  0.8344666666666667,
  0.8515333333333334,
  0.8669333333333333,
  0.8768666666666667],
 [0.7956, 0.8254, 0.8454, 0.8554, 0.8618, 0.8635]]
```

In [100]:

```
fig = plt.figure(figsize=(20, 5))
ax = fig.add_subplot(121)
ax.plot(train_sent_base_acc[0], label='train')
ax.plot(train_sent_base_acc[1], label='dev')
ax.set_title("Accuracy With pretraining.")
ax.legend()



ax2 = fig.add_subplot(122)
ax2.plot(base_sent_acc[0], label='train')
ax2.plot(base_sent_acc[1], label='dev')
ax2.set_title("Accuracy Without pretraining")
ax2.legend()

plt.show()
```



Можно заметить, что модель с предобучением менее склонна к переобучению и получает более высокие результаты в начале обучения.

Лучшие модели

По результатам экспериментов с тестирующим скриптом получилось следующее:

Лучший конфиг для **LSTM** сети без предобучения:

In [63]:

```
imdb_baseV2_config = {
        'batch_size': 256,'num_layers': 1,
        'emb_size': 650, 'hidden_size': 650,
        'vocab_size': len(preprocessor.w2ind),
        'dropout_rate': 0.65, 'num_epochs': 8,
        'learning_rate': 0.0005, 'lr_decay' : 0.5,
        'epoch_decay' : 4, 'tied_embs': False,
        'weight_init':0.1, 'grad_clipping' : None,
        'optimizer' : 'Adam'
}
imdb_baseV2_config['vocab_size']
```

Out[63]:

```
31141
```

Лучше контролируется переобучение, чем на двухслойной модели и достигает **88** процентов точности.

Лучший конфиг для предобучаемой сети:

In [ ]:

```
# Для модели и предобучения на IMDB
lm_config = {
        'batch_size': 256, 'num_steps': 35,
        'num_layers': 2, 'emb_size': 650,
        'hidden_size': 650, 'vocab_size': -1,
        'dropout_rate': 0.65, 'num_epochs': 10,
        'learning_rate': 0.005, 'lr_decay' : 0.8,
        'epoch_decay' : 8, 'tied_embs': False,
        'weight_init': 0.05, 'grad_clipping' : 5,
        'optimizer' : 'Adam',
        'adaptive' : True
         }
```

В целом обучить языковую модель с хорошей перплексией на **IMDB** оказалось тяжело, минимальная получалась в районе **~1300** при словаре на **60000** слов. У этого конфига **~1600** на **60000** слов.

In [ ]:

```
# Для обучения определения тональностей на IMDB
sent_config = {
```

```
         'batch_size' : 256,
         'vocab_size': -1,
         'dropout_rate' : 0.65, 'num_epochs' : 5,
         'learning_rate': 0.0005, 'lr_decay' : 0.5,
         'epoch_decay' : 3, 'grad_clipping' : 5,
         'optimizer' : 'Adam'
         }
```

Но предобучение действительно даёт свои плоды, модель меньше переобучается и стартует первую с большей точности. **(~82%)** Если подкрутить параметры **(**или добавить валидационный датасет **XD),** то точно можно выбить **90+.**

Этот конфиг получает следующие результаты в лидерборде:

```
train: 95.97
dev: 89.58
dev-b: 76.65
```

In [ ]:

```

```