Deep Learning School

Phystech-School of Applied Mathematics and Informatics MIPT

Assignment 3

Classifying Texts

In this task you will try several methods used in the classification task and understand how well the model understands the meaning of words and which words in the example affect the result

In [ ]:

```
!pip install torchtext
```

In [ ]:

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

# from torchtext import datasets
#from torchtext.data import Field, LabelField
#from torchtext.data import BucketIterator


from torchtext.legacy import datasets

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

from sklearn.metrics import f1_score


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
```

In this task we will use the torchtext library. It is quite easy to use and will help us concentrate on the task at hand and not on writing the Dataloader.

In [ ]:

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

In [ ]:

```
SEED = 1234

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

The dataset on which we will conduct experiments is the comments on movies from the IMDB site.

In [ ]:

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

```
downloading aclImdb_v1.tar.gz
```

```
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:08<00:00, 10.2MB/s]
```

In [ ]:

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

In [ ]:

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

train_iter, valid_iter, test_iter = BucketIterator.splits(
    (train, valid, test),
    batch_size = 32,
    sort_within_batch = True,
    device = device)
```

RNN

First, let's try using recurrent neural networks. In the seminar you got acquainted with GRU, you can also try LSTM. You can use both hidden_state and output of the last token for classification.

In [ ]:

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

        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.n_layers = n_layers
        self.bidirectional = bidirectional
        self.dropout = dropout

        self.num_directions = (2 if self.bidirectional else 1)
```

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

        self.rnn = torch.nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim,
                                num_layers=n_layers, dropout=dropout,
                                    bidirectional=bidirectional) # YOUR CODE GOES HE
RE


        self.fc = nn.Linear(in_features=self.num_directions * self.hidden_dim, out_featu
res=self.output_dim)  # YOUR CODE GOES HERE

        self.sigmoid = nn.Sigmoid()


    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.to('c
pu')) # enforce_sorted = False не пишу, так как тексты - уже отсортированы по убыванию дл
ины (что очень интересно)


        # print('packed_embedded = ', packed_embedded.shape)

        # cell arg for LSTM, remove for GRU
        # packed_output, (hidden, cell) = self.rnn(packed_embedded)
        packed_output, hidden = 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

        if self.num_directions == 2:
            hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), axis=1)
        else:
            hidden = hidden[-1,:,:]

        hidden = torch.nn.Dropout2d(p=dropout)(hidden)

        return self.fc(hidden)
```

In [ ]:

```
TEXT.vocab.freqs
```

In [ ]:

```
vocab_size = len(TEXT.vocab)
emb_dim = 100
hidden_dim = 256
output_dim = 1
n_layers = 2
bidirectional = False
dropout = 0.2
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]
```

```
patience = 3
trashhold = 0.6
```

In [ ]:

```
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
)
```

In [ ]:

```
model = model.to(device)
```

In [ ]:

```
opt = torch.optim.Adam(model.parameters())
loss_func = nn.BCEWithLogitsLoss()

max_epochs = 20
```

Teach the grid! Use whatever tools you are comfortable with,

Catalyst, PyTorch Lightning, or your own bikes.

In [ ]:

```
import numpy as np

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:
        #YOUR CODE GOES HERE
        text = batch.text[0]
        text_lengths = batch.text[1]
        labels = batch.label

        predictions = model(text, text_lengths).squeeze()
        loss = loss_func(predictions, labels)
        train_loss += loss.item()

        opt.zero_grad();
        loss.backward();
        opt.step()

    train_loss /= len(train_iter)
    val_loss = 0.0
    model.eval()
    pbar = tqdm(enumerate(valid_iter), total=len(valid_iter), leave=False)
    pbar.set_description(f"Epoch {epoch}")
    for it, batch in pbar:
        # YOUR CODE GOES HERE
        text = batch.text[0]
        text_lengths = batch.text[1]
        labels = batch.label
        predictions = model(text, text_lengths).squeeze()
```

```
        loss = loss_func(predictions, labels)
        val_loss += loss

    val_loss /= len(valid_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)
```

```
Epoch: 1, Training Loss: 0.6191010311709025, Validation Loss: 0.47462156414985657


Epoch: 2, Training Loss: 0.3493758476793875, Validation Loss: 0.33488476276397705


Epoch: 3, Training Loss: 0.1651708846486138, Validation Loss: 0.4366738498210907


Epoch: 4, Training Loss: 0.06053116925527262, Validation Loss: 0.5876861214637756
```

Out[ ]:

```
<All keys matched successfully>
```

In [ ]:

```
best_model
```

Calculate the f1-score of your classifier on a test dataset.

Answer: 0.8476774895137487 - for 1-directional 0.7754515979620195 - for 2-directional

In [ ]:

```
from sklearn.metrics import f1_score
```

In [ ]:

```
def get_predicted_labels(trashhold, Model, text, text_lengths):
    prediction = nn.Sigmoid()(Model(text, text_lengths)).squeeze()
    return (prediction > trashhold).long()
```

In [ ]:

```
model.eval()
pbar = tqdm(enumerate(test_iter), total=len(test_iter), leave=False)
full_true_labels = []
full_labels = []

for it, batch in pbar:
    text = batch.text[0]
    text_lengths = batch.text[1]
    labels = batch.label
    full_true_labels += labels.tolist()
    predictions = get_predicted_labels(trashhold, model, text, text_lengths)
    full_labels += predictions.tolist()
```

In [ ]:

```
f1_score(full_true_labels, full_labels)
```

```
f1_score(full_true_labels, full_labels)
```

Out[ ]:

```
0.8476774895137487
```

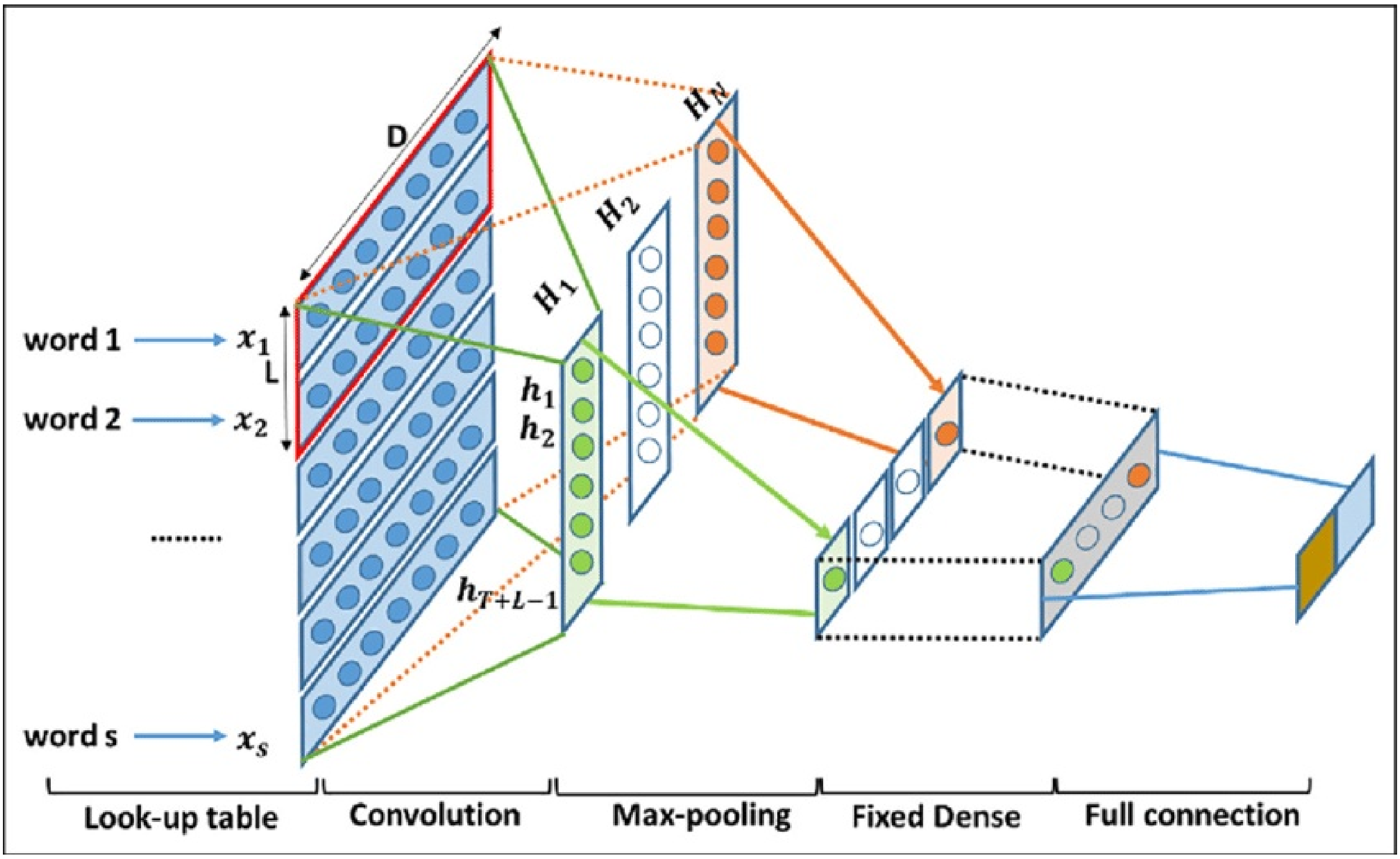
In [ ]:

```
f1_score(full_true_labels, full_labels)
```

Out[ ]:

```
0.7754515979620195
```

CNN



Convergent neural networks are also often used to classify texts. The idea is that sentiment usually contains word combinations of two or three words, such as "very good movie" or "incredible boredom. By convolving these words, we'll get some big score and grab it with MaxPool. Next comes the usual full-link grid. Important point: convolutions are applied in parallel, not sequentially. Let's give it a try!

In [ ]:

```
TEXT = Field(sequential=True, lower=True, batch_first=True)
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"
```

In [ ]:

```
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,
)
```

In [ ]:

```
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_s
ize=kernel_sizes[0])  # YOUR CODE GOES HERE

        self.conv_1 = nn.Conv1d(in_channels=emb_dim, out_channels=out_channels, kernel_s
ize=kernel_sizes[1])  # YOUR CODE GOES HERE

        self.conv_2 = nn.Conv1d(in_channels=emb_dim, out_channels=out_channels, kernel_s
ize=kernel_sizes[2])  # YOUR CODE GOES HERE

        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)  # may be reshape here (batch_size=1, len,
channels)

        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

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

Out[ ]:

```
CNN(
  (embedding): Embedding(201699, 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)
)
```

)

In [ ]:

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

model = CNN(vocab_size=vocab_size, emb_dim=dim, out_channels=out_channels,
            kernel_sizes=kernel_sizes, dropout=dropout)
```

In [ ]:

```
model.to(device)
```

Out[ ]:

```
CNN(
  (embedding): Embedding(201699, 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)
)
```

In [ ]:

```
opt = torch.optim.Adam(model.parameters())
loss_func = nn.BCEWithLogitsLoss()
```

In [ ]:

```
max_epochs = 30
```

In [ ]:

```
import numpy as np

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:
        #YOUR CODE GOES HERE
        text = batch.text
        labels = batch.label

        predictions = model(text).squeeze()
        loss = loss_func(predictions,batch.label)
        train_loss += loss.item()

        opt.zero_grad()
        loss.backward()
        opt.step()

    train_loss /= len(train_iter)
    val_loss = 0.0
    model.eval()
    pbar = tqdm(enumerate(valid_iter), total=len(valid_iter), leave=False)
    pbar.set_description(f"Epoch {epoch}")
    for it, batch in pbar:
        # YOUR CODE GOES HERE
        text = batch.text[0].permute(1, 0)
```

```
        text_lengths = batch.text[1]
        labels = batch.label

        predictions = model(text).squeeze()
        loss = loss_func(predictions, batch.label)
        val_loss += loss.item()

    val_loss /= len(valid_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)
```

```
Epoch: 1, Training Loss: 0.6577484022526845, Validation Loss: 0.4861943632998365

Epoch: 2, Training Loss: 0.50270312067366, Validation Loss: 0.43321259789010313

Epoch: 3, Training Loss: 0.43607614292715585, Validation Loss: 0.3965200604910546

Epoch: 4, Training Loss: 0.37478363231150774, Validation Loss: 0.377902693761156

Epoch: 5, Training Loss: 0.32260232717886456, Validation Loss: 0.351920526141816

Epoch: 6, Training Loss: 0.2542978553441319, Validation Loss: 0.3426128013336912

Epoch: 7, Training Loss: 0.19984156176121565, Validation Loss: 0.34108469036031275

Epoch: 8, Training Loss: 0.14387191027185342, Validation Loss: 0.35172091342033224

Epoch: 9, Training Loss: 0.09879345101487898, Validation Loss: 0.36325071997782016
```

Out[ ]:

```
<All keys matched successfully>
```

Count the f1-score of your classifier.

Answer: 0.8160759383436386

In [ ]:

```
def get_predicted_labels(trashhold, Model, text):
    prediction = nn.Sigmoid()(Model(text)).squeeze()
    return (prediction > trashhold).long()
```

In [ ]:

```
model.eval()
pbar = tqdm(enumerate(test_iter), total=len(test_iter), leave=False)
full_true_labels = []
full_labels = []
```

```python
for it, batch in pbar:
    text = batch.text
    labels = batch.label
    full_true_labels += labels.tolist()
    predictions = get_predicted_labels(trashhold, model, text)
    full_labels += predictions.tolist()
```

In [ ]:

```python
f1_score(full_true_labels, full_labels)
```

Out[ ]:

```
0.8160759383436386
```

Interpretability

Let's see where our model looks. Just run the code below..

In [ ]:

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

```
     |████████████████████████████████| 1.4 MB 2.8 MB/s
```

In [ ]:

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

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

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

In [ ]:

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

def forward_with_sigmoid(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(input_indices).item()
    pred_ind = round(pred)

    # generate reference indices for each sample
```

```
    reference_indices = token_reference.generate_reference(seq_length, device=device).un
squeeze(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))
```

In [ ]:

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

```
pred:  pos ( 0.90 ) , delta:  tensor([0.0002], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.14 ) , delta:  tensor([1.3985e-06], device='cuda:0', dtype=torch.float64)
pred:  pos ( 0.96 ) , delta:  tensor([4.6455e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.39 ) , delta:  tensor([5.1680e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.25 ) , delta:  tensor([0.0001], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.03 ) , delta:  tensor([2.9328e-05], device='cuda:0', dtype=torch.float64)
```

In [ ]:

```
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 (0.90)	pos	1.67	It was a fantastic performance ! pad
pos	neg (0.14)	pos	1.38	Best film ever pad pad pad pad
pos	pos (0.96)	pos	1.36	Such a great show! pad pad pad
neg	neg (0.39)	pos	1.88	It was a horrible movie pad pad
neg	neg (0.25)	pos	0.34	I've never watched something as bad pad
neg	neg (0.03)	pos	-0.20	It is a disgusting movie! pad pad

Out[ ]:

Legend: Negative Neutral Positive

Legend: ■ Negative □ Neutral ■ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
pos	pos (0.90)	pos	1.67	It was a fantastic performance ! pad
pos	neg (0.14)	pos	1.38	Best film ever pad pad pad pad
pos	pos (0.96)	pos	1.36	Such a great show! pad pad pad
neg	neg (0.39)	pos	1.88	It was a horrible movie pad pad
neg	neg (0.25)	pos	0.34	I've never watched something as bad pad
neg	neg (0.03)	pos	-0.20	It is a disgusting movie! pad pad

Word Embeddings

You haven't forgotten how we can apply knowledge about word2vec and GloVe. Let's give it a try!

In [ ]:

```
TEXT.build_vocab(trn, vectors='glove.6B.300d')# YOUR CODE GOES HERE
LABEL.build_vocab(trn)

word_embeddings = TEXT.vocab.vectors

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

```
.vector_cache/glove.6B.zip: 862MB [03:11, 4.50MB/s]
100%|█████████▉| 399999/400000 [00:51<00:00, 7828.51it/s]
```

In [ ]:

```
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,
)
```

In [ ]:

```
model = CNN(vocab_size=vocab_size, emb_dim=dim, out_channels=64,
            kernel_sizes=kernel_sizes, dropout=dropout)

word_embeddings = TEXT.vocab.vectors

prev_shape = model.embedding.weight.shape

# model.embedding.weight = word_embeddings
model.embedding.weight.data.copy_(word_embeddings)

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

opt = torch.optim.Adam(model.parameters())
loss_func = nn.BCEWithLogitsLoss()
```

In [ ]:

```python
import numpy as np

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:
        #YOUR CODE GOES HERE
        text = batch.text
        labels = batch.label
        predictions = model(text).squeeze()

        loss = loss_func(predictions, labels)
        train_loss += loss.item()

        opt.zero_grad()
        loss.backward()
        opt.step()


    train_loss /= len(train_iter)
    val_loss = 0.0
    model.eval()
    pbar = tqdm(enumerate(valid_iter), total=len(valid_iter), leave=False)
    pbar.set_description(f"Epoch {epoch}")
    for it, batch in pbar:
        # YOUR CODE GOES HERE
        text = batch.text[0].permute(1, 0)
        text_lengths = batch.text[1]
        labels = batch.label
        predictions = model(text).squeeze()
        loss = loss_func(predictions, batch.label)
        val_loss += loss.item()

    val_loss /= len(valid_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)
```

```
Epoch: 1, Training Loss: 0.2618396611761873, Validation Loss: 0.30894329234006557


Epoch: 2, Training Loss: 0.12871238181408304, Validation Loss: 0.2984120455194027


Epoch: 3, Training Loss: 0.05146971217145885, Validation Loss: 0.3270345648552509


Epoch: 4, Training Loss: 0.021074199855300416, Validation Loss: 0.35639029375891734
```

Out[ ]:

```
<All keys matched successfully>
```

Count the f1-score of your classifier. Answer: 0.8595054656931951

Count the f1-score of your classifier. Answer: 0.8595054656931951

In [ ]:

```
model.eval()
pbar = tqdm(enumerate(test_iter), total=len(test_iter), leave=False)
full_true_labels = []
full_labels = []

for it, batch in pbar:
    text = batch.text
    labels = batch.label
    full_true_labels += labels.tolist()
    predictions = get_predicted_labels(trashhold, model, text)
    full_labels += predictions.tolist()
```

In [ ]:

```
f1_score(full_true_labels, full_labels)
```

Out[ ]:

```
0.8595054656931951
```

In [ ]:

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

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

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

```
pred:  pos ( 0.90 ) , delta:  tensor([8.7221e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.00 ) , delta:  tensor([2.6858e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.26 ) , delta:  tensor([0.0002], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.00 ) , delta:  tensor([4.3905e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.33 ) , delta:  tensor([3.2961e-05], device='cuda:0', dtype=torch.float64)
pred:  neg ( 0.00 ) , delta:  tensor([0.0001], device='cuda:0', dtype=torch.float64)
```

In [ ]:

```
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 (0.90)	pos	2.00	It was a fantastic performance ! pad
pos	neg (0.00)	pos	0.54	Best film ever pad pad pad pad
pos	neg (0.26)	pos	1.75	Such a great show! pad pad pad
neg	neg (0.00)	pos	0.00	It was a horrible movie pad pad
neg	neg (0.33)	pos	1.67	I've never watched something as bad pad
neg	neg (0.00)	pos	-0.21	It is a disgusting movie! pad pad

Out[ ]:

Legend: ■ Negative □ Neutral ■ Positive

Legend: Negative Neutral Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
pos	pos (0.90)	pos	2.00	It was a fantastic performance ! pad
pos	neg (0.00)	pos	0.54	Best film ever pad pad pad pad
pos	neg (0.26)	pos	1.75	Such a great show! pad pad pad
neg	neg (0.00)	pos	0.00	It was a horrible movie pad pad
neg	neg (0.33)	pos	1.67	I've never watched something as bad pad
neg	neg (0.00)	pos	-0.21	It is a disgusting movie! pad pad

```
In [ ]:
```