Sentiment Analysis with Pytorch — Part 1 — Data Preprocessing

Gal Hever · Follow
8 min read · Apr 8, 2020

Introduction

This tutorial is the first in a serie of blog-posts that will take you through sentiment analysis task with Pytorch. Each step of the code will be explained with an example for people that are doing it for the first time. The full code of this tutorial is available here.

If you wish to continue to the next parts in the serie:

Sentiment Analysis with Pytorch — Part 2— Linear Model

Sentiment Analysis with Pytorch — Part 3— CNN Model

Sentiment Analysis with Pytorch — Part 4 — LSTM\BiLSTM Model

Sentiment Analysis with Pytorch — Part 5—MLP Model

First, let's start with the basic question: "What does it mean Sentiment Analysis?"

Sentiment Analysis definition

Sentiment analysis is a task in Natural Language Processing (NLP) that it's purpose is to classify sentences into one of several categories that refer to sentence's expression for a certain topic, such as: positive, negative, natural. For example: "This guitar is the best that I have ever seen" will be classified as 'Positive'.

In this tutorial we will focus on sentiment analysis of social media content. The data consist of 12K comments in Hebrew that people wrote on Facebook pages of political figures. We will use the data of the Neural Sentiment Analyzer for Modern Hebrew article. Please download the data from Github and you can also go over the Keras version of what we are going to do now. There are two kinds of data files, one is Token-Based and the other is Morpheme-Based. In this serie we will create a few Neural Network models while we will be able to choose how to treat the data: Word-Based or Char-

Based and for each will be chosen the data type: Token-Based or Morpheme-Based.

Let's Code!

So now that we understood the concept, let's start coding:) If it's your first time coding with Pytorch so please take a few seconds to install and import Torch and TorchText packages.

```
pip install torch
pip install torchtext

import torch
from torchtext import data
```

Data Preprocessing

In this tutorial I will use TorchText package to deal with the data. There are a few ways to process the data, but this package makes this procedure much easier. Now we will go over the next steps with TorchText:

1. Preprocessing and tokenization
2. Building vocabulary
3. Loading Embedding Vectors
4. Padding the text
5. Batching the data

The data is separated into two columns while the first column represents the sentence in Hebrew and the second column represents the label. This is a multiclass task with 3 categories: 0-Positive, 1-Negative, 2-Natural.

Note: your labels are supposed to start also from zero, otherwise you will get an error.

0	0	איש יקר שלנו
1	0	כל הכבוד והמון בהצלחה
2	0	...תל חי , רובי . בכל העצב הזה היית קרן אור של ת
3	0	. נקי כפיים ובר לבב בהצלחה לך ולנו
4	0	. רובי חג שמח והצלחה בבחירות לנשיאות
5	0	...הנשיא לא נותן יד . דיי להתלהמות אפשר להביע דיע
6	0	...כבוד הנשיא אין מתאים ממך בעולם מלהיות נשיא מדי
7	0	...כל הכבוד לך אדוני הנשיא כאיש ימין דווקא , אני
8	1	...אדוני הנשיא סליחה על הבוטות תפסיק לזיין את השכ
9	0	בהצלחה אדוני הנשיא
10	1	ואני עוד חשבתי שהגיע נשיא נורמלי

What is a Field object?

First, we will define an object of class type 'Field' that will store information about the way of preprocessing the data. There are two kind of columns in Sentiment Analysis task: Field and Label.

1. Field is used to specify how to preprocess each data column in our dataset.
2. LabelField is used only to define the label in classification tasks.

In the code below I created two Field objects, one named ‘Text’ and the other ‘Label’ (you can choose your own name for your convenient).

```
char_based = True
if char_based:
    tokenizer = lambda s: list(s) # char-based
else:
    tokenizer = lambda s: s.split() # word-based

Text = data.Field(preprocessing=cleanup_text, tokenize=tokenizer,
batch_first=True, include_lengths=True,
fix_length=max_document_length)

Label = data.Field(sequential=False, use_vocab=False, pad_token=None,
unk_token=None)
```

Parameters of Field object:

- **Preprocessing**: If you want to do some manipulation on the data column before numericalizing so you can replace this attribute with a custom preprocessor (see cleanup_text function belows).
- **Tokenize**: Tokenization specifies the way of breaking up the sentence into small pieces that are called tokens (divides the sentence into a list of words or chars). You can also use known NLP open source libraries that supply tokenizers like ‘spacy’ (segmenting text into words, punctuations marks etc).
- **Fix_length**: In default, TorchText will dynamically pad each sequence to the longest sequence in each batch, but if `Fix_length` variable will be set to some number, it will pad all the sequences in all the batches to have the same length.
- **Include_lengths**: If we didn’t use the `Fix_length` option, we will have to set this variable to TRUE that TorchText will know how to pad each

sequence in each batch.

- **Batch_first**: There are some layers in NN that expect the batch dimension in the input to be first as CNN layers whereas RNN layers that expect it to be second. TorchText can return the data already permuted using the `batch_first=TRUE`.
- **Sequential**: Sequential data is any kind of data where the order matters. In our case Text field will be sequential compared to Label field that will be set to FALSE.
- **Use_vocab**: This variable will be set to TRUE for the column that is responsible for building the vocabulary of unique words.
- **Lower**: Converts the text to lowercase.
- **Pad_token**: Will be set to NONE when padding is unnecessary.
- **UNK_token**: Will be set to NONE when we don't want to use "out of vocabulary" (OOV) tokens for the specific Field object.

You can continue reading about all the variables of Field object here.

```
import re

def cleanup_text(texts):
    cleaned_text = []
    for text in texts:
        # remove punctuation
        text = re.sub('[^a-zA-Z0-9]', ' ', text)
        # remove multiple spaces
        text = re.sub(r' +', ' ', text)
        # remove newline
        text = re.sub(r'\n', ' ', text)
        cleaned_text.append(text)
    return cleaned_text
```

Note: Make sure to remove all the '\n' character before saving the CSV as TorchText have trouble handling '\n' character.

What are "Out of Vocabulary" words?

A text classification model is trained on fixed vocabulary size, all the words that weren't included in the vocabulary are known as "Out of Vocabulary" (OOV) words. In order to handle the OOV, pytorch supports a feature that replaces the rare words in our training data with unknown token <UNK>.

Now we will connect between the Field objects we created to the columns in the dataset. In our dataset we don't have a title to the columns so we can just call it "text" and "labels" (note that the order of the fields names is supposed to be in the same order as your dataset). If there is a column that you don't want to use at all you can set it columns to NONE, as: `('column_name', None)`.

```
fields = [('text', Text), ('labels', Label)]
```

Now, we can use it to import the data by TabularDataset. TabularDataset is one of many TorchText data structures that specifically deals with tabular datasets like CSV and TSV. We will need to define the data folder path, the names of the files and the format of the data file (CSV, TSV). In the `fields` variable we will define the Fields objects that we just created before and `skip_header=TRUE` refers to the first row in the dataset. In our case we don't have a header so we will set this variable to False (by defining it the first row will be considered).

```
train_data, test_data = data.TabularDataset.splits(
    path='C:/Users/Gal/PycharmProjects/Sentiment_Analyzer/data',
    train='train.tsv',
    test='test.tsv',
    format='tsv',
    fields=fields,
    skip_header=False
)
```

TabularDataset wraps all the columns (text and labels) into a single object, that looks like that:

```
print(vars(train_data[0]))
```

```
{'text': ['צער', 'עוד', 'תדעו', 'שלא', 'המשפחה', 'עם', 'בוכה', 'אני', '.....', 'כואב', 'ממש'], 'labels': ['0']}
```

We can print just the sentence or the label of the first row using our Filed object name (in our case we called it ‘text’ and ‘labels’)

```
train_data[0].text
train_data[0].labels
```

For validation set creation we will split feature as follow:

```
train_data, valid_data = train_data.split(split_ratio=0.8,
random_state=random.seed(seed))
```

Build the Vocabulary

In the next step we will create the vocabulary by mapping all the unique words (in our case we set it to 5000 by `max_size` variable) in the train_data to an index. If you will also use word embedding it will map the index to the corresponding word embedding right after.

```
Text.build_vocab(train_data, valid_data, max_size=5000)
Label.build_vocab(train_data)

vocab_size = len(Text.vocab)
```

Note: You can also use 'min_freq' variable to ignore all the words in the vocabulary which have a lower frequency than specified and map them to unknown token.

TorchText creates a dictionary of all the unique words and arranges them in a decreasing order in accordance to their frequency. Next, TorchText assigns a unique integer to each word and keeps this mapping in Text.vocab.stoi (string to index) and a reverse mapping in Text.vocab.itos (index to string).

Note: When building the vocabulary you will notice that you will have 2 words more than what you defined, this is because TorchText adds to your vocabulary <pad> and <unk> tokens as well.

Loading an Embedding Layer

If you want to use word vectors, TorchText can load the embedding layer easily by mentioning the name of the pretrained word vector (e.g. charngram.100d, glove.6B.200d, fasttext.en.300d, etc.). Another way, if you have already downloaded the word vectors, then you can specify the folder path as is written below.

```
from torchtext import vocab

embeddings = vocab.Vectors('glove.840B.300d.txt',
'./data/glove_embedding/')

Text.build_vocab(train_data, valid_data, max_size=5000,
vectors=embeddings)
Label.build_vocab(train_data)
```

Iterator Creator

TorchText provides BucketIterator that groups sequences together. It returns a Batch object that contains the data of one batch while the text and labels can be accessed via column names.

If you didn't set `Fix_length` attribute in the Field object above to a particular length, TorchText will batch sequences of similar lengths together to the maximum length in each batch. This action is more efficient because it reduces the amount of padding required within our batch and will save lots of matrices calculations that are unnecessary (note that in our case I didn't use this function).

Finally, you will need to define for the iterator a few variables. First, set your batch size and the device type. The function `torch.cuda.is_available()` checks and returns a Boolean TRUE if a GPU is available. Second, a `sort_key` function that determines how to sort the data in the validation and test set. For example, by setting `sort_key` to lambda `x:len(x.text)`, TorchText will sort the samples by their lengths. When the parameter `sort_within_batch` is set to True, TorchText performs the data in each batch in a descending order following the `sort_key` attribute. This setting is required while using the `pack_padded_sequence` that we will see in the next parts of the serie.

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

```
train_iterator, valid_iterator, test_iterator =
data.BucketIterator.splits((train_data, valid_data, test_data),
        batch_size = 50,
        sort_key = lambda x: len(x.text), sort_within_batch = True,
        device = device)
```

End Notes

In this section we went over the data preparation by TorchText before entering to the model. The next step is building the model and training it. If you wish to continue to the next step it will be explained in details in the next blog post: Sentiment Analysis with Pytorch — Part 2 — Linear Model.

You can find the full code for this tutorial on Github.

References

[https://www.aclweb.org/anthology/C18-1190.pdf]

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

✦ · 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

✦ · 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

✦ · 15 min rea

View list

Written by Gal Hever

Follow

108 Followers

Data Scientist

More from Gal Hever

Gal Hever

Getting Started with NVIDIA NeMo ASR

NVIDIA NeMo — Quick Start Guide

Gal Hever

Sentiment Analysis with Pytorch — Part 4 — LSTM\BiLSTM Model

Introduction

3 min read · Apr 20, 2021

89

8 min read · Apr 11, 2020

74 2

Gal Hever

Coreference Resolution Models

A Review of the Latest Models

12 min read · Aug 10, 2020

17

Gal Hever

Sentiment Analysis with Pytorch — Part 2 — Linear Model

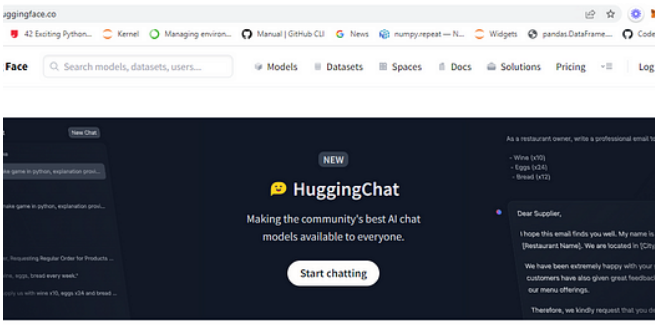
Introduction

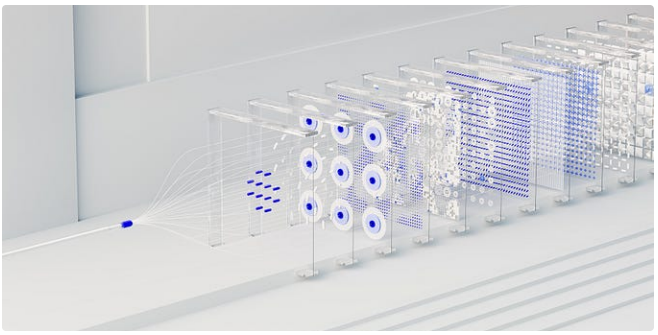
7 min read · Apr 8, 2020

7

See all from Gal Hever

Recommended from Medium





Bright Eshun

Sentiment Analysis (Part 1): Finetuning DistilBert for Text...

I. Introduction

9 min read · May 8

2

Lefteris Charteros

Building a Sentiment Analysis Classifier using PyTorch Lightning

The purpose of this tutorial is to build a complete machine learning system that will...

17 min read · Jul 22

Lists

Staff Picks

465 stories · 317 saves

Stories to Help You Level-Up at Work

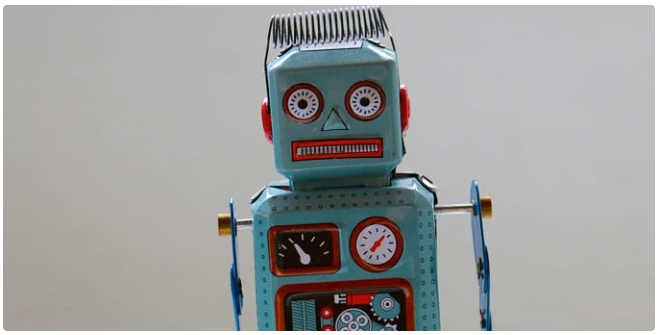
19 stories · 235 saves

Self-Improvement 101

20 stories · 643 saves

Productivity 101

20 stories · 597 saves



sandeep pandey

Text Classification using Custom Data and PyTorch

Text classification is a fundamental natural language processing (NLP) task that involve...

5 min read · May 18

1

AR

BERT for Sequence Classification from Scratch—Code and Theory

Coding BERT for sequence classification from scratch serves as an exercise to better...

16 min read · Aug 3

3 2

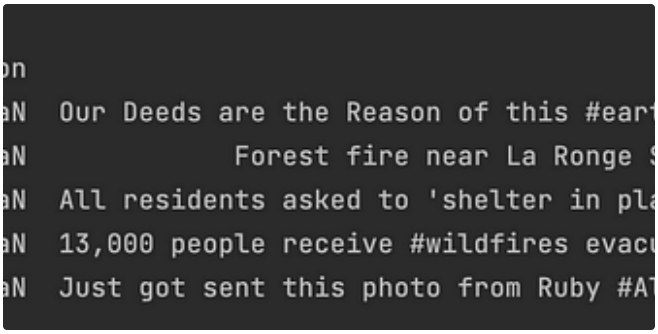
Alidu Abubakari in AI Science

Taking Sentiment Analysis to the Next Level with Huggingface's...

Introduction

17 min read · May 31

104 1



Hatice Şeyma Koç

Fasttext & Doc2Vec for Text Classification

Hello everyone,

5 min read · Jul 7

11

See more recommendations