6 min read · Dec 10, 2022

The pytorch gitlab repo has a torchtext API migration notebook, but it stopped working in torchtext version 0.14.0, and has not been upgraded.

Up to a certain torchtext version, it was still possible to use things like `torchtext.legacy.data.Field`. That is not available anymore.

In this post, we walk through an updated torchtext migration colab notebook that has been tested with torchtext 0.14.0. The outline below is shamelessly borrowed from the original pytorch sources.

Let's start with the first step.

Create a dataset object

The new dataset API returns the train/test dataset split directly without the preprocessing information. Each split is an iterator which yields the raw text and labels line-by-line.

```
>>> from torchtext.datasets import IMDB
>>> train_iter, test_iter = IMDB(split=('train', 'test'))
```

Here, `train_iter` and `test_iter` are `ShardingFilterIterDataPipe` objects, which derives from the `DataPipe` class base. They are iterable-style data pipes. To print out the raw data, you can use a `for` loop:

```
>>> for label, line in train_iter:
>>>     print(f"Label: {label}")
>>>     print(f"Line: '{line}'")
>>>     break
>>>
>>> Label: 1
>>> Line: 'I rented I AM CURIOUS-YELLOW from my video store because of all the co
```

Build the data processing pipeline

This part is inspired from this nice AICore youtube tutorial.

Users can access different kinds of tokenizers directly via `data.get_tokenizer()` function. We will use the `basic_english` tokenizer:

```
>>> from torchtext.data.utils import get_tokenizer
>>> tokenizer = get_tokenizer("basic_english")
>>> tokens = tokenizer("You can now install TorchText using pip!")
>>> tokens
>>> ['you', 'can', 'now', 'install', 'torchtext', 'using', 'pip', '!']
```

Next step is to build a `Vocab` class. Use the argument `min_freq` to set up the cutoff frequency to in the vocabulary. Special tokens, like `<UNK>` and `<PAD>` can be assigned to the special symbols in the constructor of the `Vocab` class.

```
>>> from torchtext.vocab import build_vocab_from_iterator
>>> from torchtext.data.utils import get_tokenizer

>>> tokenizer = get_tokenizer("basic_english")
```

```
>>> def yield_tokens(data_iter):
>>>   for _, text in data_iter:
>>>     yield tokenizer(text)

>>> def get_vocab(train_datapipe):
>>>   vocab = build_vocab_from_iterator(yield_tokens(train_datapipe),
>>>                                     specials=['<UNK>', '<PAD>'],
>>>                                     max_tokens=20000)
>>>   vocab.set_default_index(vocab['<UNK>'])
>>>   return vocab

>>> train_iter = IMDB(split='train')
>>> vocab = get_vocab(train_iter)
```

- The length of the new vocab is `len(vocab)` .
- To get a dictionary of keys-> tokens, call `vocab.get_itos()` , where itos stands for integer-to-string.
- To get the token for a key: `vocab.get_itos()[key]`
- Conversely, to get a key for a token: `vocab['the']`
- Or, more indirectly, get a dictionary of tokens->keys, calling `vocab.get_stoi()['the']}`

Generate the batch behavior

To train a model efficiently, we build an iterator to generate data batch.

We use `torch.utils.data.DataLoader` to generate data batch. We can customize the batch by defining a `collate_batch()` function, and pass it as a `collate_fn` argument to the `DataLoader` constructor. In `collate_batch()` we process the raw text data and add padding to dynamically match the longest sentence in a batch.

```
>>> from torch.utils.data import DataLoader
>>> from torch.nn.utils.rnn import pad_sequence

>>> def collate_batch(batch):
>>>  label_list, text_list = [], []
```

```
>>>  for (_label, _text) in batch:
>>>    label_list.append(label_transform(_label))
>>>    processed_text = torch.tensor(text_transform(_text))
>>>    text_list.append(processed_text)

>>>  return torch.tensor(label_list), pad_sequence(text_list, padding_value=3.0)

>>> train_iter = IMDB(split='train')
>>> train_dataloader = DataLoader(list(train_iter),
>>>                               batch_size=8,
>>>                               shuffle=True,
>>>                               collate_fn=collate_batch)
```

Older versions of torch had a `BucketIterator` class, which could easily group texts with similar lengths together. This helped reduce the minibatch max length and made training more efficient.

In new versions of torch, `BucketIterator` is not available, but the behavior can be implemented as follows:

- We randomly create multiple "pools", each of them of size `batch_size * 100`.
- We sort the samples within the individual pool by length.

In the code below, we implemented a generator that yields batch of indices for which the corresponding batch of data is of similar length.

(This code snippet is inspired from here.)

```
import random
from torch.utils.data import Sampler

train_iter = IMDB(split='train')
train_list = list(train_iter)
batch_size = 8  # A batch size of 8

class BatchSamplerSimilarLength(Sampler):
  def __init__(self, dataset, batch_size, indices=None, shuffle=True):
    self.batch_size = batch_size
    self.shuffle = shuffle
```

```
        # get the indices and length
        self.indices = [(i, len(tokenizer(s[1]))) for i, s in enumerate(dataset)]
        # if indices are passed, then use only the ones passed (for ddp)
        if indices is not None:
           self.indices = torch.tensor(self.indices)[indices].tolist()

    def __iter__(self):
        if self.shuffle:
           random.shuffle(self.indices)

        pooled_indices = []
        # create pool of indices with similar lengths
        for i in range(0, len(self.indices), self.batch_size * 100):
          pooled_indices.extend(sorted(self.indices[i:i + self.batch_size * 100], key=
        self.pooled_indices = [x[0] for x in pooled_indices]

        # yield indices for current batch
        batches = [self.pooled_indices[i:i + self.batch_size] for i in
                   range(0, len(self.pooled_indices), self.batch_size)]

        if self.shuffle:
            random.shuffle(batches)
        for batch in batches:
            yield batch

    def __len__(self):
        return len(self.pooled_indices) // self.batch_size
```

We now create the `DataLoader`. We pass the `batch_sampler` class and the `collate_batch` function as arguments.

```
>>> bucket_dataloader = DataLoader(train_list,
>>>                                batch_sampler=BatchSamplerSimilarLength(
>>>                                      dataset = train_list,
>>>                                      batch_size=batch_size),
>>>                                collate_fn=collate_batch)
>>> print(next(iter(bucket_dataloader)))
```

Iterate through batch to train a model

The batch iterator can be iterated or executed with `next()` method.

```
>>> for idx, (label, text) in enumerate(train_dataloader):
>>>   model(item)
```

How do I use my own data?

Our example code used the built-in `torchtext.datasets.IMDB` dataset. In a real project, you want to implement your own dataset.

An example of how that is done is available in the packed_lstm.ipynb worksheet.

We can download the same IMDB movie review dataset (http://ai.stanford.edu/~amaas/data/sentiment/) for positive-negative sentiment classification as a CSV-formatted file. We'll actually download it from one of Sebastian Raschka's repositories. His class Introduction to Deep Learning is outstanding.

```
$ wget https://github.com/rasbt/python-machine-learning-book-3rd-edition/raw/maste
$ gunzip -f movie_data.csv.gz
```

Check that the dataset looks okay:

```
>>> import pandas as pd
>>>
>>> df = pd.read_csv('movie_data.csv')
>>> df.head()
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0
3	hi for all the people who have seen this wonde...	1
4	I recently bought the DVD, forgetting just how...	0

Now create the same DataPipe:

```
>>> from torchdata.datapipes.iter import IterableWrapper, FileOpener
>>> datapipe = IterableWrapper(["movie_data.csv"])
>>> datapipe = FileOpener(datapipe, mode='b')
>>> datapipe = datapipe.parse_csv(skip_lines=1)

>>> for sample in datapipe:
>>>     print(sample)
>>>     break
>>>
>>> ['In 1974, the teenager Martha Moxley (Maggie Grace) moves to the high-class a
```

Notice that the data samples are a list of `[text, label]` , compared with the `torchtext.datasets.IMDB` dataset, where the order was `[label, text]` .

From this point, you have a `DataPipe` object, and can repeat the earlier steps to create a `DataLoader` .

Pytorch Torchtext Naturallanguageprocessing

Follow

Written by Andrei Radulescu-Banu

28 Followers

Find out more about me at https://bitdribble.github.io/

More from Andrei Radulescu-Banu

 Andrei Radulescu-Banu

Ali Ghodsi's U Waterloo AI Courses

I came across Ali Gholdsi's lectures by chance, and some of the YouTube comments were so positive that I had to watch his lectures:

4 min read · Jan 18, 2022

143 1

Andrei Radulescu-Banu

When & Where — at Startup Boston

Tuesday night's panel at the Startup Boston 2018 conference spoke about the resources available to Boston entrepreneurs trying to bootstrap...

4 min read · Sep 11, 2018

5

Andrei Radulescu-Banu

TechStars Meet & Greet

Below are my notes from the TechStars Boston Meet & Greet at their GSV Labs location near Downtown Crossing. This meet took place Thursday…

6 min read · Sep 13, 2018

10

 Andrei Radulescu-Banu

Elasticsearch and LogZ at Boston DevOpsDays

Today at DevOps Boston — fist day of two-day conference — I got to talk to a number of companies that set up booth. While most companies...

5 min read · Sep 24, 2018

1

See all from Andrei Radulescu-Banu

Recommended from Medium

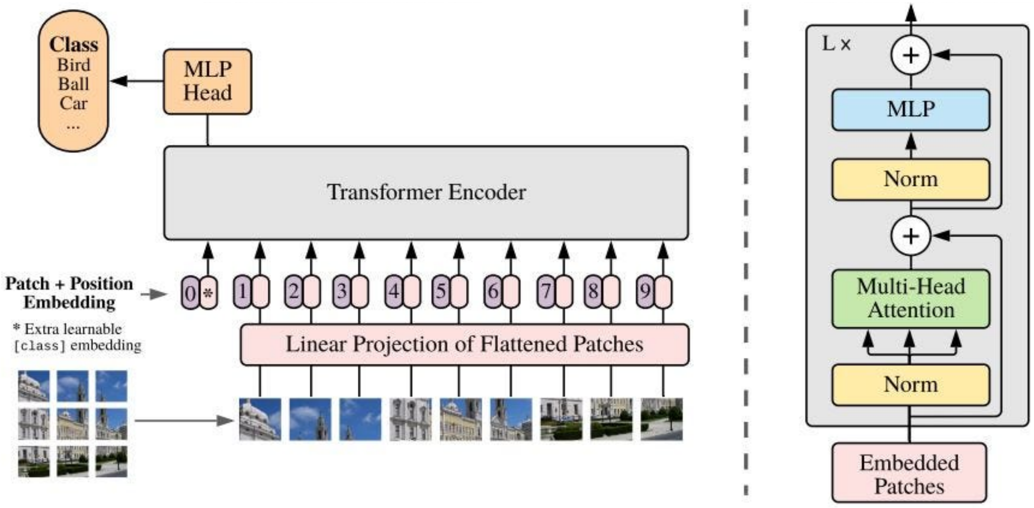


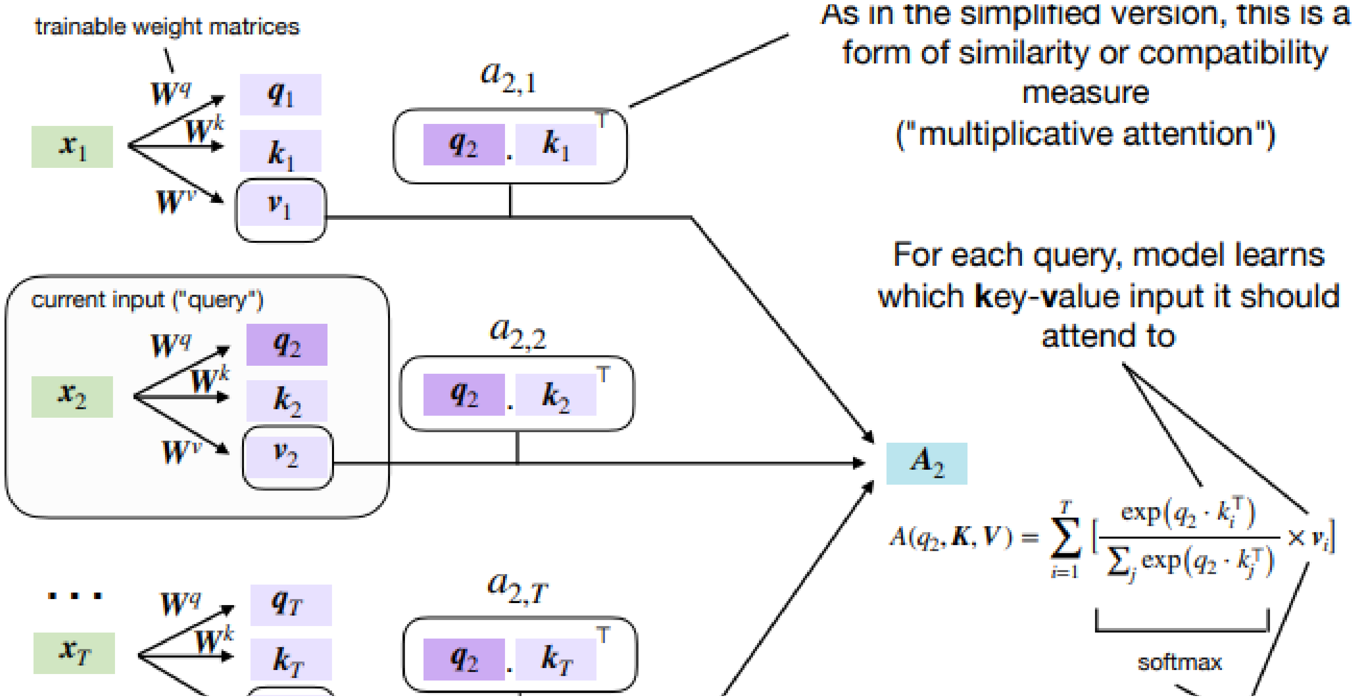
Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by

f Fahim Rustamy, PhD

Vision Transformers vs. Convolutional Neural Networks

This blog post is inspired by the paper titled AN IMAGE IS WORTH 16×16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE from google's...

7 min read · Jun 4



Zain ul Abideen

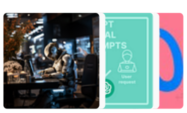
Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26

5

Lists

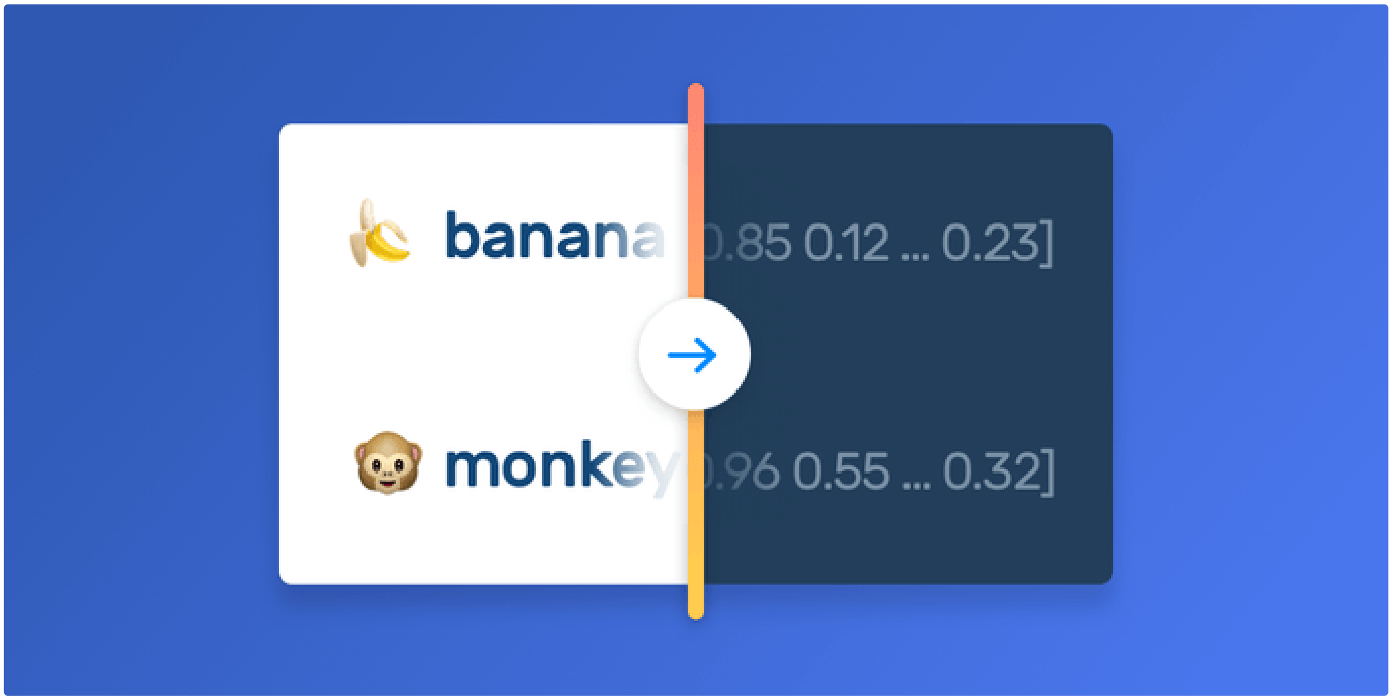


Natural Language Processing

568 stories · 189 saves

Now in AI: Handpicked by Better Programming

266 stories · 118 saves



mayank khulbe in The Good Food Economy

Skipgram implementation from scratch—Pytorch

In recent times, there has been an exponential increase in the use-cases pertaining to Natural Language Processing. With this, word…

12 min read · May 23

23

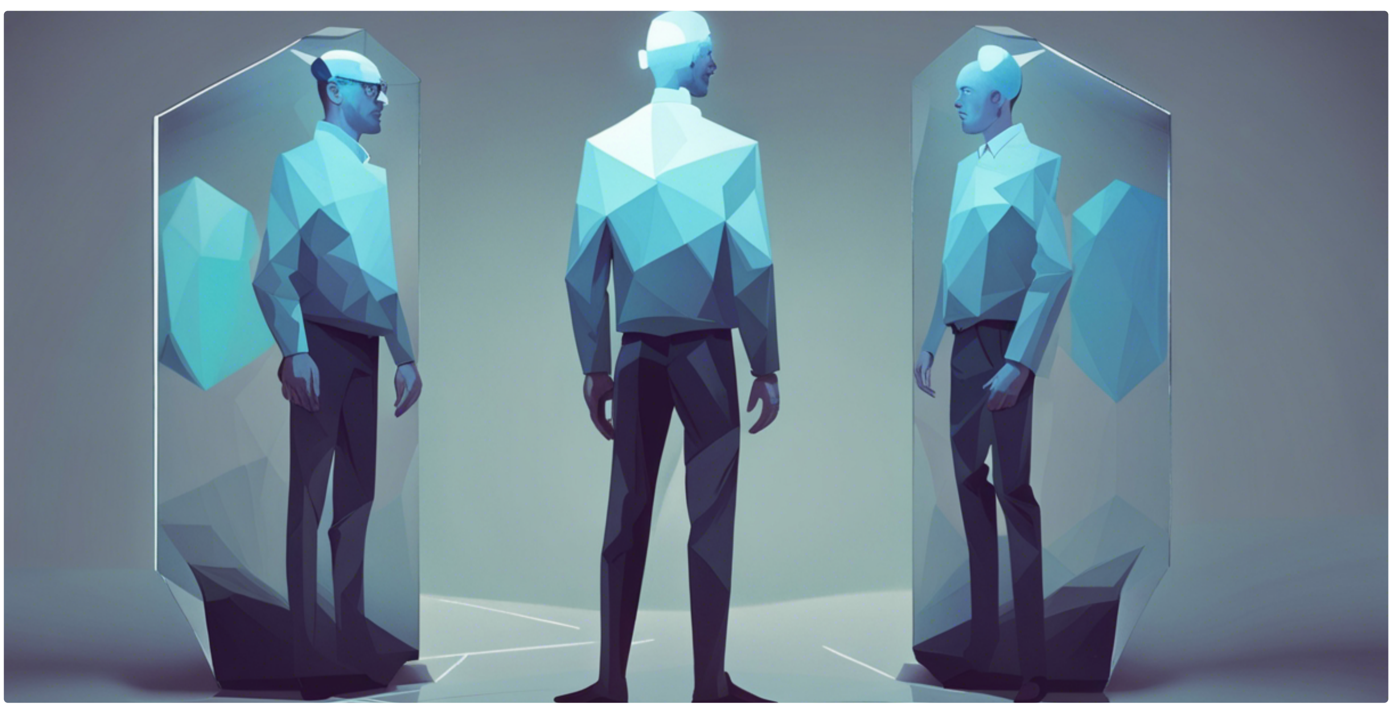
Avi Chawla in Towards Data Science

It’s Time to Say GoodBye to pd.read_csv() and pd.to_csv()

Discussing another major caveat of Pandas

4 min read · May 26, 2022

2.7K 50



Sergei Savvov in Better Programming

Create a Clone of Yourself With a Fine-tuned LLM

Unleash your digital twin

11 min read · Jul 27

2.2K 16

Priyatosh Anand

Handle Long Text Corpus for Bert Model

In this tutorial we will try to understand how to do Sentiment Analysis using FinBERT for the long text corpus greater than 512 tokens.

10 min read · 6 days ago

166 1

See more recommendations