Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 5: Huggingface datasets and tokenization

Topics

- Loading huggingface datasets
- DatasetDict.map
- Opening the black box
- Tokenization
- Outlook on extracting the last hidden states

Huggingface Datasets

Loading datasets

```
>>> from datasets import load_dataset

>>> ds = load_dataset("rotten_tomatoes")
>>> ds
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 1066
    })
})
```

- Format is a `DatasetDict`
- Feels like dictionary of lists
- Easy to transform to many formats
- Powerful methods

Selecting data

```
>>> ds["train"]
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 8530
})
```

```
>>> data["train"][:2]
```

```
{'text': [
    'the rock is [...]',
    'the gorgeously elaborate [...]'
  ],
 'label': [1, 1]
}
```

```
>>> data["train"][0]
```

```
{'text': 'the rock is [...]', 'label': 1}
```

- Output format strongly depends on how you select data
- Takes time to get used to but always sensible
- Despite looking like pure python list, storage is very efficient

Inspecting data

```
>>> ds.num_columns
```

```
{'train': 2, 'validation': 2, 'test': 2}
```

```
>>> ds.num_rows
```

```
{'train': 8530, 'validation': 1066, 'test': 1066}
```

```
>>> ds.column_names
```

```
{'train': ['text', 'label'],
 'validation': ['text', 'label'],
 'test': ['text', 'label']}
```

```
>>> ds.shape
```

```
{'train': (8530, 2), 'validation': (1066, 2), 'test': (106
```

```
>>> ds.unique("label")
```

```
{'test': [1, 0], 'train': [1, 0], 'validation': [1, 0]}
```

- For more attributes check the documentation
- Information is shown per split
- Hints at the fact that there are ways to apply functions to each split

Deleting data

```
>>> ds.cache_files
```

```
{'train': [{'filename': '/home/janos/.cache/huggingface/da
 'validation': [{'filename': '/home/janos/.cache/huggingfa
 'test': [{'filename': '/home/janos/.cache/huggingface/dat
```

- Shows that on my computer the files are in `home/janos/.cache`
- Can simply delete them:
  - In your file explorer
  - In a terminal
  - Using pathlib

Converting to other formats

```
>>> ds.set_format(type="pandas")
>>> ds["train"]
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 8530
})
```

```
>>> type(ds["train"])
```

```
datasets.arrow_dataset.Dataset
```

```
>>> type(ds["train"][:])
```

```
pandas.core.frame.DataFrame
```

```
>>> print(ds["train"][:2])
```

```
                                                text  labe
0  the rock is destined to be the 21st century's ...
1  the gorgeously elaborate continuation of " the...
```

- No effect until rows of data are selected
- Need to select all rows (via `[:]`) to get everything in a dataset
- Many other format available
  - numpy
  - torch
  - tensorflow
  - ...

Task 1

(7 min)

What is `DatasetDict.map`

- A method that applies a function to the data
- Two modes are interesting for us:
  - single observation mode
  - batch mode
- Easy to parallelize
- Difficult part: the function needs a specific interface
- We will go very slowly!

Big picture

```
>>> def f(...):
...     # do stuff
...     return ...

>>> new_ds = ds.map(f, ...)
>>> new_ds.column_names
```

```
{'train': ['text', 'label', 'new_variable'],
 'validation': ['text', 'label', 'new_variable'],
 'test': ['text', 'label', 'new_variable']}
```

- What should `f` take?
- What should `f` return?
- Do we need additional arguments for `map`?

Look at documentation

```
function (callable) — with one of the following signature:

    function(example: Dict[str, Any]) -> Dict[str, Any] if batched=False and with_indices=False
    function(example: Dict[str, Any], indices: int) -> Dict[str, Any] if batched=False and with_indices=True
    function(batch: Dict[str, List]) -> Dict[str, List] if batched=True and with_indices=False
    function(batch: Dict[str, List], indices: List[int]) -> Dict[str, List] if batched=True and with_indices=True

    For advanced usage, the function can also return a pyarrow.Table. Moreover if your function
    returns nothing (None), then map will run your function and return the dataset unchanged.

[...]

with_indices (bool, defaults to False) — Provide example indices to function. Note that in this
case the signature of function should be def function(example, idx): ....

batched (bool, defaults to False) — Provide batch of examples to function.

[...]
```

Or try to find out

```
>>> def fake_f(x):
...     print(x)
...     return x
>>> new_ds = ds.map(fake_f)
```

```
{'text': 'the rock is destined to [...]', 'label': 1}
{'text': 'the gorgeously elaborate [...]', 'label': 1}
...
```

```
>>> new_ds.column_names
```

```
{'train': ['text', 'label'],
 'validation': ['text', 'label'],
 'test': ['text', 'label']}
```

```
>>> ds["train"][0]
```

```
{'text': 'the rock is destined to [...]', 'label': 1}
```

- Even if we would get an error later, the print statement would show how `fake_f` was called
- Each printed line looks like when we select an individual row of data
- The columns of the mapped dataset seem to be the keys of the dictionaries we returned

So how does `map` work?

```python
def f(row):
    row["new_variable"] = row["text"].split(" ")[0]
    return row

f(ds["train"][0])
```

```
{'text': 'the rock is destined [...]',
 'label': 1,
 'new_variable': 'the'}
```

```python
>>> new_ds = ds.map(f)
>>> new_ds.column_names
```

```
{'train': ['text', 'label', 'new_variable'],
 'validation': ['text', 'label', 'new_variable'],
 'test': ['text', 'label', 'new_variable']}
```

- `f` needs to take same dictionary as you get from selecting one row of data
- `f` needs to return a dictionary with all columns you want
- Can test the function before using map!

Reflection

- Do not panic when you meet a complex method
- Do not try out random stuff
- Learn from documentation and experimentation
- Simplify as much as possible until you understand what you are doing
- "*Programming isn't about what you know; it's about what you can figure out.*" (Chris Pine)

Important: Reset format!

```
>>> ds.set_format(type="pandas")
>>> ds.map(f)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[20], line 2
      1 ds.set_format(type="pandas")
----> 2 ds.map(f)
[...]
      1 def f(row):
----> 2     row["new_variable"] = row["text"].split(" ")[0]
      3     return row
[...]
---->

AttributeError: 'Series' object has no attribute 'split'
```

This would have shown (with a nicer error message) from testing on one row:

```
>>> f(ds["train"])[0]
```

Task 2

(10 min)

Task 3

(15 min)

Opening the Black Box

Our curret mental model

NLP models ...

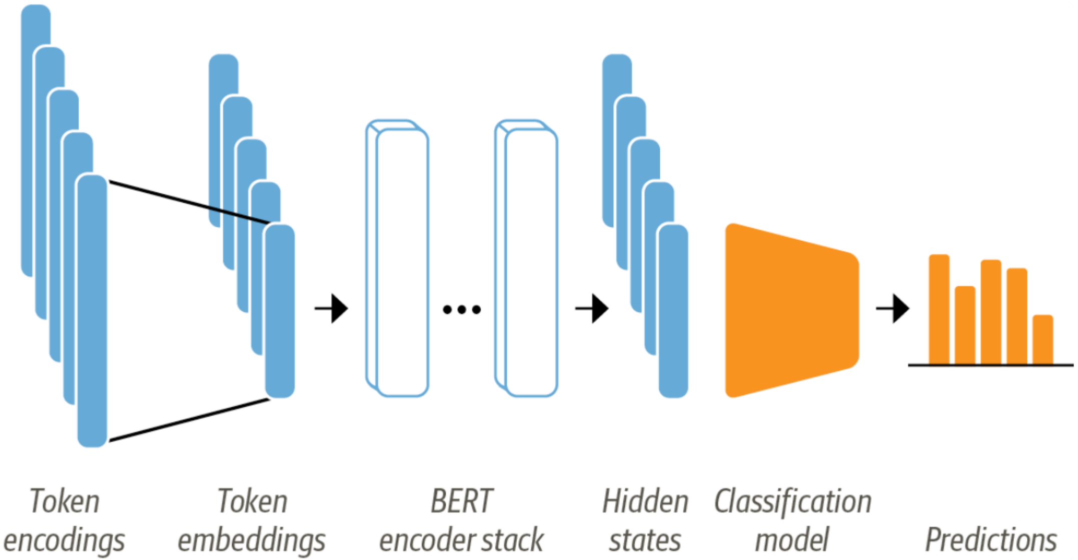
- take text
- return text

Our new mental model

NLP models …

- **take text**
- convert text to integers (**tokenization**)
- convert integers to one-hot vectors
- do a lot of matrix products and other calculations
- produce vectors containing a lot of information (**last hidden states**)
- have a task specific head that uses this information
- **return text**

Bert for classification



Source: Natural Language Processing with transformers, Fig 2-3

Why do the last hidden states contain useful information?

- The functions applied to the one-hot vectors have many free parameters
- Those parameter are trained with optimization algorithms
- The objective function is the performance in a pre-training task
  - Predict a masked **token**
  - Predict the next **token**
- Thus, last hidden states are implicitly defined as those vectors that make it easiest to perform the pre-traning task

For the impatient

- We will look at all of this in full detail
- Starting bottom up would not let you see the big picture

Tokenization

What is tokenization

Goal:

- Convert raw text to a list of integers with an invertible mapping
- Encode as much structure as possible

Constraints:

- Number of tokens should not be too large
- Should work for unseen text

Word tokenization

- Number all words in a comprehensive dictionary of your language
- Translate words to ints by looking up the number
- Map unknown words to a single token

Pros

- Preserves word structure
- Simple
- Not too many words

Cons

- Variations like "huuuuuge"
- Typos like "laern"
- Morphology like "learned"
- Incompleteness

Character tokenization

- Number all letters and punctuation characters
- Translate characters to ints by looking up the number

Pros

- Very simply
- Tiny vocabulary size
- No unknown words

Cons

- Loses entire word structure
- Tokenized texts are very long

Task 4

(10 min)

Subword tokenization

- Used in all practical applications
- Middle ground between word and character tokenization
  - Frequent words get their own token
  - Other tokens represent parts of words (##ly, ##ed)
  - Rest is encoded by characters
- You can choose vocabulary size
- No unknown words
- Several algorithms exist

Example: Byte-pair encoding

- Start with a vocabulary containing only characters and end-of-word symbol
- Use a corpus of text to find the most frequent adjacent characters
  - e.g. if a and b often occur together, add the subword "ab" to vocabulary
- Replace instances of the character pair in the corpus with the new subword
- Repeat until you reach desired vocabulary size
- Example: Start with {"a", "b", ..., " "}, end with {"a", "b", ..., "apple", "app##", ...}

You have seen this before

```
ner_tagger = pipeline("ner", aggregation_strategy="simple")
outputs = ner_tagger(text)
pd.DataFrame.from_records(outputs)
```

	entity_group	score	word	start	end
0	ORG	0.879010	Amazon	5	11
1	MISC	0.990859	Optimus Prime	36	49
2	LOC	0.999755	Germany	90	97
3	MISC	0.556568	Mega	208	212
4	PER	0.590256	##tron	212	216
5	ORG	0.669693	Decept	253	259
6	MISC	0.498349	##icons	259	264
7	MISC	0.775361	Megatron	350	358
8	MISC	0.987854	Optimus Prime	367	380
9	PER	0.812096	Bumblebee	502	511

Warnings

- When working with a pre-trained model you have to use the exact same tokenizer as was used to train the model
- There are many subtle ways to get it wrong
  - Encode characters in a different order
  - Use similar tokenizer trained on other corpus
  - ...
- Most of the time, huggingface makes it hard to get it wrong

Pre-trained encoders

```
>>> from transformers import AutoTokenizer
>>> model_name = "distilbert-base-uncased"
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
>>> example_tokens = tokenizer.encode("Hello World")
>>> example_tokens
```

```
[101, 7592, 2088, 102]
```

```
>>> for token in example_tokens:
...     print(token, tokenizer.decode(token))
```

```
101 [CLS]
7592 hello
2088 world
102 [SEP]
```

- There will also be `` `AutoModel` `` and if you use the same `` `model_name` `` in `` `AutoModel` `` and `` `AutoTokenizer` `` everything will match
- `` `[CLS]` `` and `` `[SEP]` `` were added automatically, you can ignore them for now

Some properties

```
tokenizer.vocab_size
```

```
30522
```

```
>>> tokenizer.special_tokens_map
```

```
{'unk_token': '[UNK]',
 'sep_token': '[SEP]',
 'pad_token': '[PAD]',
 'cls_token': '[CLS]',
 'mask_token': '[MASK]'}
```

```
>>> tokenizer.model_max_length
```

```
512
```

- Vocab size shows that this must be a subword tokenizer
- Special tokens are related to the pre-training task and other logistics
- No need to understand them for now

Tokenizers do more than encoding

```
>>> tokenizer(["Hello World"])
```

```
{'input_ids': [101, 7592, 2088, 102],
'attention_mask': [1, 1, 1, 1]}
```

```
tokenizer(
    ["Hello", "Hello World"],
    padding=True,
    truncation=True,
)
```

```
{'input_ids': [
    [101, 7592, 102, 0],
    [101, 7592, 2088, 102]
  ],
 'attention_mask': [
    [1, 1, 1, 0],
    [1, 1, 1, 1]
]}
```

- Padding will be needed to convert nested lists of tokens into arrays
- Truncation is needed if some input text is too long for the model

Task 5

15 min

Pytorch tensors

What is Pytorch

- Library for implementing deep learning models
- Various levels of abstraction
  - Build models spe specifying layers
  - Implement components from scratch using tensors
- Used by OpenAI for all recent models
- Industry standard
- Not as beautiful as JAX

What are Tensors

```
import torch
import numpy as np
>>> a = np.array([1,2,3])
>>> x = torch.from_mumpy(a)
>>> x
tensor([1, 2, 3])
```

```
>>> x.to("cpu")
>>> x.device
device(type='cpu')
```

- Data structure similar to numpy arrays
- Runs on GPU and other hardware accelerators
- Can be created from lists and numpy arrays
- Can be converted to numpy array

Task 6

(5 min)

A few differences to numpy

```
>>> x = torch.tensor([1,2,3])
>>> x[0]
tensor(1, )
```

```
>>> a = torch.arange(4).reshape(2,2)
>>> b = torch.linspace(0.1,0.5,4).reshape(2,2)
>>> a.matmul(b)
---------------------------------------------------------
RuntimeError                Traceback (most recent call la
Cell In[95], line 1
----> 1 a.matmul(b)

RuntimeError: expected scalar type Long but found Float
```

```
>>> a.to(torch.float).matmul(b)
tensor([[0.3667, 0.5000],
        [1.3000, 1.9667]])
```

- Indexing returns single elements as tensors
- `` `@` `` and `` `.matmul` `` for matrix multiplication
- `` `.dot` `` for 1d tensors
- For matrix product, tensor dtypes need to match!

Differentiation with Pytorch

```
>>> x = torch.tensor([1.0], requires_grad=True)
>>> y = torch.tensor([2.0], requires_grad=True)
>>> a = torch.tensor([0.5])
>>> z = a*x**2 + y**2
>>> z.requires_grad
True
```

```
>>> z.backward()
>>> x.grad
tensor([1.])
```

```
>>> y.grad
tensor([4.])
```

```
>>> with torch.no_grad():
...     z = a*x**2 + y**2
```

```
>>> z.requires_grad
False
```

- `` `requires_grad` `` triggers tracking the computational graph
- `` `.backwards` `` does the actual differentiation
- gradients are stored as part of tensors
- `` `torch.no_grad()` `` disables gradient calculation, e.g. during inference

Task 7

(5 min)

Outlook

What we will have to do

- Tokenize the entire emotions dataset using `` `DatasetDict.map` ``
- Write a `` `map` `` compatible function to extract last hidden states
    - process inputs
    - evaluate model
    - convert output to numpy
    - do some post processing
- Create arrays we can use in sklearn

Next week

- Class starts at 12:15
- First half:
  - feature extraction
  - classification with the extracted features
  - comparison against a pre-trained classification model
- Second half: Guest lecture on entrepreneurship