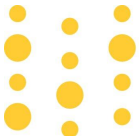




Course Study Group with a bent

Session 2

Hosted by  W&B

Agenda

This week we'll be **looking at section 2**.

Our goal this week is to **understand how Hugging Face's high-level pipeline works under the covers**, including how to preprocess your inputs with a tokenizer, run your inputs through a model, and finally postprocess the outputs the same aforementioned tokenizer.

We'll also **take an initial look at Blurr, and in particular how it handles the tokenization and models in a fastai world**.

Resources

Study Group registration page: <https://wandb.me/fastai-hf>

Study Group discord: <https://discord.gg/DsnRxSyt>

fastai:

1. The **fastai course** (<https://course.fast.ai/>)
2. The **Walk w/ fastai** course (<https://walkwithfastai.com/>)
3. The **FastBook** (available for purchase or free online via Jupyter notebooks)
4. The **FastBook reading/study group** form W&B (<http://wandb.me/fastbook>)

fastai + Hugging Face libraries:

1. AdaptNLP (<https://novetta.github.io/adaptnlp/>)
2. FastHugs (<https://github.com/morganmcg1/fasthugs>)
3. Blurr (<https://ohmeow.github.io/blurr/>)

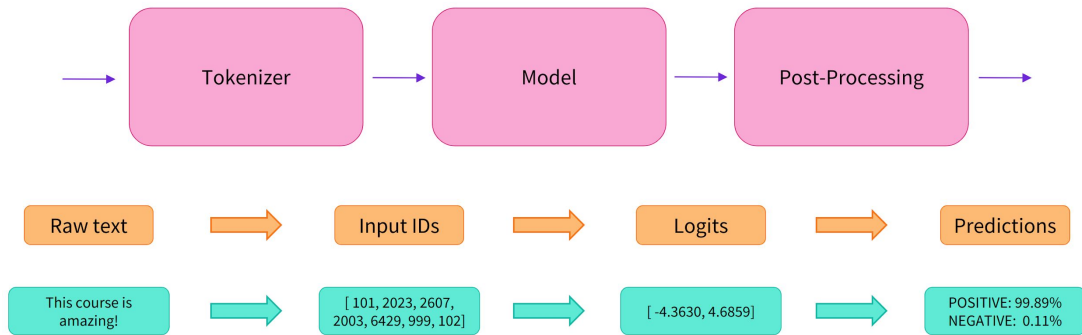
ML/Data Science in general:

1. The **Chai Time Data Science** podcast (<http://youtube.com/c/chaitimedatascience>)
2. **Weights & Biases** (<https://wandb.ai/>)

2. Using Transformers: **Behind the pipeline**

Step 1: Tokenize inputs

- Splits the input into words, subwords, or symbols called ***tokens***
- Maps each token to an integer
- Adds additional inputs useful to the model



Important: “All this preprocessing needs to be done in exactly the same way as when the model was pretrained.”

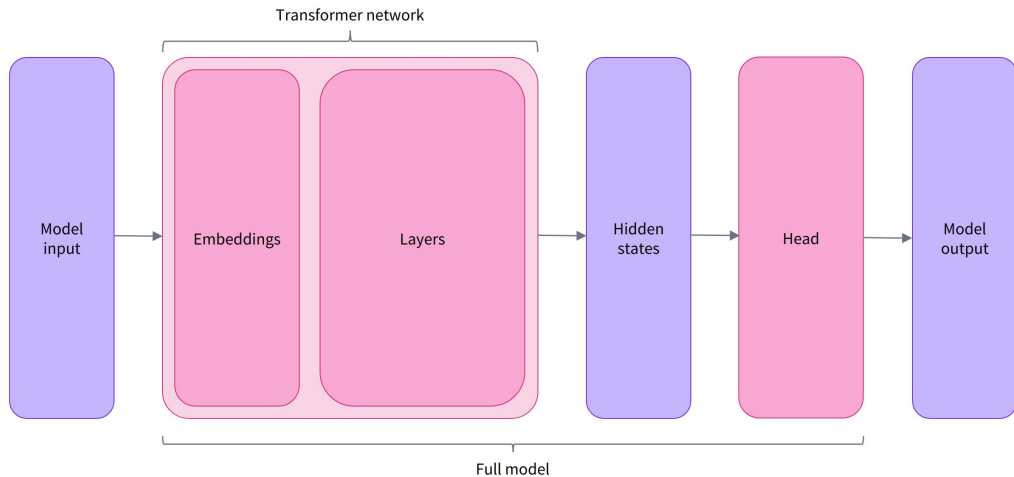
To do this we build a tokenizer using the checkpoint name of our model

2. Using Transformers: Behind the pipeline

Step 2: Run inputs through model

```
model = AutoModel.from_pretrained(checkpoint)
```

“This architecture contains only the base Transformer module: given some inputs, *it outputs what we’ll call **hidden states**, also known as **features***. For each model input, we’ll retrieve *a **high-dimensional vector representing the contextual understanding of that input*** by the Transformer model.”



These features/hidden states are usually passed to a task specific part of the model called **the head**; each task will have a different head associated with it.

2. Using Transformers: **Behind the pipeline**

Step 3: Postprocess outputs

Important: “... the outputs of 🙌
*Transformers models behave like
namedtuples or dictionaries.*”

Important: All Transformers models
outputs the logits “as the loss
function for training will generally
fuse the last activation function,
such as Softmax, with the actual
loss function, such as cross
entropy)

```
from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(**inputs)
print(outputs.logits.shape)
# torch.Size([2,2])

print(outputs.logits)
# tensor([[ -1.5607,  1.6123], [ 4.1692, -3.3464]]),
# grad_fn=<AddmmBackward>)

predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
# tensor([[4.0195e-02, 9.5980e-01],
#         [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)

model.config.id2label

# {0: 'NEGATIVE', 1: 'POSITIVE'}
```

2. Using Transformers: **Models**

The **AutoModel** class is a simple wrapper that “can automatically guess the appropriate model architecture for your checkpoint, and then instantiates a model with this architecture.”

Use **from_pretrained** ... *“This model is now initialized with all the weights of the checkpoint. It can be used directly for inference on the tasks it was trained on, and it can also be fine-tuned on a new task.”*

Use **save_pretrained**(<directory on your computer>) to save your model. This method creates two files:

1. **config.json**: “the attributes necessary to build the model architectures ... some metadata, such as where the checkpoint originated and what Transformers version you were using when you last saved the checkpoint.”
2. **pytorch_model.bin**: “the state dictionary; it contains all your model’s weights.”

2. Using Transformers: **Tokenizers**

Tokenizers translate text into numerical data that can be passed to a model. There are three core algorithms.

2. Using Transformers: **Tokenizers**

Word-based Tokenizers

Split on spaces

Let's	do	tokenization!
-------	----	---------------

Split on punctuation

Let	's	do	tokenization	!
-----	----	----	--------------	---

Benefits:

1. Easy to set up
2. Few rules (e.g., split on space)
3. Often yields decent results

Negatives:

1. Large vocabularies
2. Loss of meaning across related words
3. Produces lots of [UNK] tokens

2. Using Transformers: **Tokenizers**

Character-based Tokenizers

L	e	t	'	s	d	o	t	o	k	e	n	i	z	a	t	i	o	n	!
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Benefits:

1. Smaller vocabulary
2. Fewer OOV (out of vocabulary)/unknown values

Negatives:

1. Less meaningful embeddings
2. More tokens required to represent a sequence

2. Using Transformers: **Tokenizers**

Subword Tokenizers

Let's</w>	do</w>	token	ization</w>	!</w>
-----------	--------	-------	-------------	-------

https://huggingface.co/transformers/tokenizer_summary.html#subword-tokenization

Benefits:

1. Provides a lot of semantic meaning
2. Small vocab with close to no unknown values
3. Works well for many different languages

Negatives:



2. Using Transformers: Tokenizers

How it works

Step 1: Raw text -> Tokens

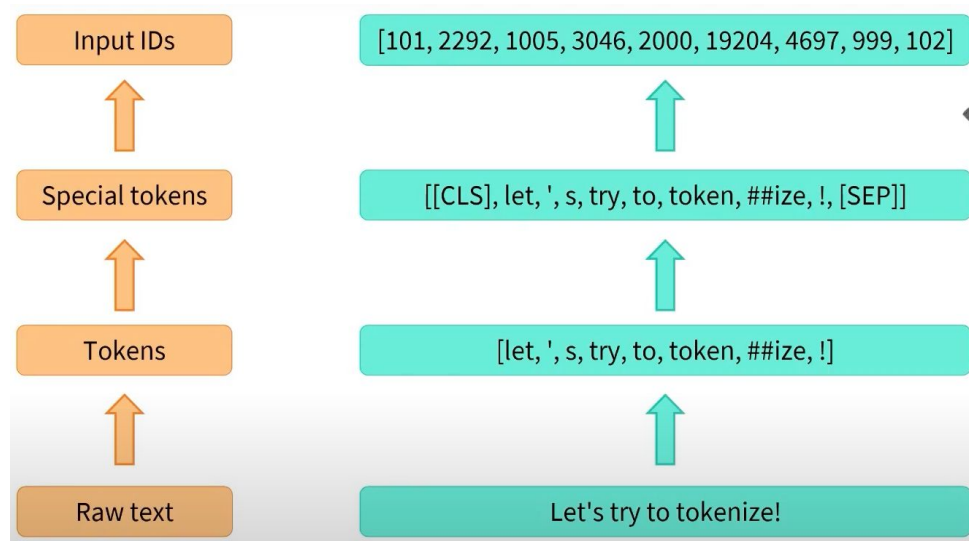
```
sequence = "Using a Transformer network is simple"  
tokens = tokenizer.tokenize(sequence)  
# ['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']
```

Step 2: Tokens -> Input IDs

```
ids = tokenizer.convert_tokens_to_ids(tokens)  
# [7993, 170, 11303, 1200, 2443, 1110, 3014]
```

Step 3: Input IDs -> Raw text

```
decoded_string = tokenizer.decode([7993, 170,  
11303, 1200, 2443, 1110, 3014])  
# 'Using a Transformer network is simple'
```



There are multiple rules that can govern the tokenization process, which is why we need to instantiate the tokenizer using the name of the model, to make sure we use the same rules that were used when the model was pretrained.

2. Using Transformers: **Handling multiple sequences**

When training, we'll be passing out inputs in mini-batches at a time ... so how do we handle cases where the input lengths are different and how do we ensure the self-attention layers only pay "attention" to the inputs?

2. Using Transformers: **Handling multiple sequences**

Padding

... “makes sure all our sentences have the same length by adding a special word called the ***padding token*** to the sentences with fewer values.”

Will pad the sequences up to the maximum sequence length in the batch

```
model_inputs = tokenizer(sequences, padding="True|longest")
```

Will pad the sequences up to the model max length (512 for BERT or DistilBERT)

```
model_inputs = tokenizer(sequences, padding="max_length")
```

Will pad the sequences up to the specified max length

```
model_inputs = tokenizer(sequences, padding="max_length", max_length=8)
```

2. Using Transformers: **Handling multiple sequences**

Truncation

... used to ensure the lengths of your sequences correspond to what the model you are using can handle (and/or your GPU)

```
# Will truncate the sequences that are longer than the model max length (512 for BERT or DistilBERT)  
model_inputs = tokenizer(sequences, truncation=True)
```

```
# Will truncate the sequences that are longer than the specified max length  
model_inputs = tokenizer(sequences, max_length=8, truncation=True)
```

2. Using Transformers: **Handling multiple sequences**

Attention Mask

“... the key feature of Transformer models is ***attention layers that contextualize each token***. These will take into account the padding tokens since they attend to all of the tokens of a sequence. ***To get the same result when passing individual sentences of different lengths through the model or when passing a batch with the same sentences and padding applied, we need to tell those attention layers to ignore the padding tokens.*** This is done by using an ***attention mask***.”

“***Attention masks*** are tensors with the exact same shape as the input IDs tensor, filled with 0s and 1s: 1s indicate the corresponding tokens should be attended to, and 0s indicate the corresponding tokens should not be attended to (i.e., they should be ignored by the attention layers of the model).”

2. Using Transformers: **Handling multiple sequences**

Usually, you'll want to go with:

- padding=True # pad to lognest sequence in the batch
- truncation=True # truncate to "max_length" if set or to the max. length allowed by the model
- max_length=None|int # max length allowed by model -or- a specific length

See [Everything you always wanted to know about padding and truncation](#) for all the possible combinations you can consider using.

Homework

1. **Watch** the official course videos from week 2
2. **Blog** about something you've learned from week 1 & 2. See if you can include some actual code using pure Hugging Face and/or one of the fastai integration libraries.
3. Re-read the “***Attention is all you need***” paper and optionally pick an architecture your curious about, the the paper, and give it a read (use discord to ask any questions about it)
4. Get read for some **competitions to be announced next week!**