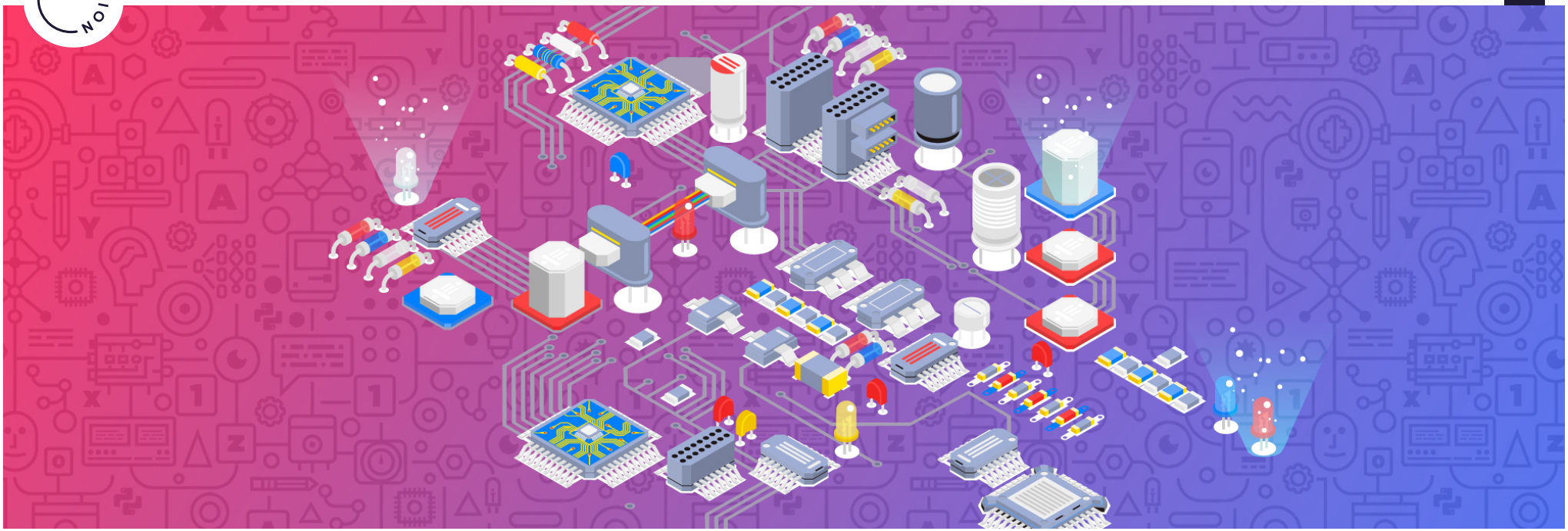


spaCy meets Transformers: Fine-tune BERT, XLNet and GPT-2

August 2, 2019 · by **Matthew Honnibal and Ines Montani**

Huge transformer models like BERT, GPT-2 and XLNet have set a new standard for accuracy on almost every NLP leaderboard. You can now use these models in spaCy, via a new interface library we've developed that connects spaCy to Hugging Face's awesome implementations.

In this post we introduce our new wrapping library, `spacy-transformers`. It features consistent and easy-to-use interfaces to several models, which can extract features to power your NLP pipelines. Support is provided for **fine-tuning the transformer models** via spaCy's standard `nlp.update` training API. The library also calculates an **alignment to spaCy's linguistic tokenization**, so you can relate the transformer features back to actual words, instead of just wordpieces. Transformer-based pipelines won't be perfect for every use-case, but they're not just for research either: even if you're processing text at scale, there are lots of ways your team could make use of these huge but highly accurate models.

Update (October 2019)

The `spacy-transformers` package was previously called `spacy-pytorch-transformers`. Since this blog post was published, Hugging Face have released an updated and renamed `transformers` package that now supports both PyTorch and TensorFlow 2. We have updated our library and this blog post accordingly.

Transformers and transfer-learning

Natural Language Processing (NLP) systems face a problem known as the "knowledge acquisition bottleneck". Deep neural networks have offered a solution, by building **dense representations** that transfer well between tasks. In the last few years, research has shown that linguistic knowledge can be acquired effectively from unlabelled text, so long as the network is large enough to represent the long-tail of rare usage phenomena. In order to continue scaling the size of the network, research has focused particularly on models which can use current GPU and TPU hardware efficiently, and models which support efficient parallel training.

The upshot of this is a new class of architectures that offer a very different set of trade-offs from prior technologies. Transformers use a network architecture that hard-codes fewer assumptions about the importance of word order and local context. This (slightly more) "blank slate" approach is a disadvantage when the model is small or data is limited, but with a big enough model and sufficient examples, transformers are able to reach a much more subtle **understanding of linguistic information**. The transformer architecture **lets bigger models be better**. With previous technologies, if you just made your model bigger (and therefore slower), your accuracy would plateau reasonably quickly, even given sufficient training data. Transformers also let you make better use of expensive GPU and TPU hardware — another benefit that's only relevant for larger models.

Using transformer models "in production"

Even though transformer models have been breaking new accuracy records every month, it's not easy to apply them directly to most practical problems. Usually, if

Even though transformer models have been breaking new accuracy records every month, it's not easy to apply them directly to most practical problems. Usually, if NLP is worth doing at all, it's worth doing quickly: the technologies are usually only well motivated for applications where you need to process a lot of text, or

where you need the answers in real-time. Your project should already have access to a process that has human-level natural language understanding capabilities, at the expense of high run-time cost and high latency: manual annotation. Transformer models provide a new middle-ground: much cheaper and lower latency than manual annotation, but still too slow for most direct applications. Almost as accurate as manual annotation on some problems, but with unpredictable errors and output that's difficult to reason about.

In a recent talk at Google Berlin, Jacob Devlin described how Google are using his BERT architectures internally. The models are **too large to serve in production**, but they can be used to **supervise a smaller production model**. Based on the (fairly vague) marketing copy, AWS might be doing something similar in SageMaker. Another offline use-case for the transformer models is in quality control — this is how S&P Global have been using BERT, as their application context requires strict accuracy guarantees. Another potential use-case is in **monitoring**, as a sort of "health-check" to gauge how a production model is performing on recent data.

Introducing spacy-transformers

Thomas Wolf and the other heroes at Hugging Face have implemented several recent transformer models in an easy-to-use package, `transformers`. This has made it easy to write a wrapping library that lets you use these models in a spaCy pipeline. We've also made use of the `spacy package` command to build pip packages that provide the weights, entry points and all the requirements. This way, you can download and load the transformer-based models with the same workflow as our other model packages:

```
$ pip install spacy-transformers
```

```
$ python -m spacy download en_trf_bertbaseuncased_lg
```

spaCy model package	Pretrained model	Language	Author
`en_trf_bertbaseuncased_lg`	`bert-base-uncased`	English	Google Research
`de_trf_bertbasecased_lg`	`bert-base-german-cased`	German	deepset
`en_trf_xlnetbasecased_lg`	`xlnet-base-cased`	English	CMU/Google Brain

> **What's in the packages?**
> The packages contain configuration settings, the binary weights for the transformer models, and mapping tables used for the wordpiece tokenization.

The transformer pipelines have a `trf_wordpiecer` component that performs the model's wordpiece pre-processing, and a `trf_tok2vec` component, which runs the transformer over the doc, and saves the results into the built-in `doc.tensor` attribute and several extension attributes.

The token vector encoder component of the model sets custom hooks that override the default behavior of spaCy's `.vector` attributes and `.similarity` methods on the `Token`, `Span` and `Doc` objects. By default, these usually refer to the word vectors table. Naturally, in the transformer models we'd rather use the `doc.tensor` attribute, since it holds a much more informative context-sensitive representation.

```
import spacy
import torch

import numpy
from numpy.testing import assert_almost_equal

is_using_gpu = spacy.prefer_gpu()
if is_using_gpu:
    torch.set_default_tensor_type("torch.cuda.FloatTensor")

nlp = spacy.load("en_trf_bertbaseuncased_lg")
doc = nlp("Here is some text to encode.")
assert doc.tensor.shape == (7, 768)  # Always has one row per token
doc._.trf_word_pieces_  # String values of the wordpieces
doc._.trf_word_pieces  # Wordpiece IDs (note: *not* spaCy's hash values!)
doc._.trf_alignment  # Alignment between spaCy tokens and wordpieces
# The raw transformer output has one row per wordpiece.
assert len(doc._.trf_last_hidden_state) == len(doc._.trf_word_pieces)
# To avoid losing information, we calculate the doc.tensor attribute such that
# the sum-pooled vectors match (apart from numeric error)
assert_almost_equal(doc.tensor.sum(axis=0), doc._.trf_last_hidden_state.sum(axis=0), decimal=5)
span = doc[2:4]
# Access the tensor from Span elements (especially helpful for sentences)
assert numpy.array_equal(span.tensor, doc.tensor[2:4])
# .vector and .similarity use the transformer outputs
apple1 = nlp("Apple shares rose on the news.")
apple2 = nlp("Apple sold fewer iPhones this quarter.")
apple3 = nlp("Apple pie is delicious.")
print(apple1[0].similarity(apple2[0]))  # 0.73428553
print(apple1[0].similarity(apple3[0]))  # 0.43365782
```

> **A note about performance**
>
> Transformer architectures are not designed to operate efficiently on CPU, so we recommend you have a GPU available for both training and usage. Internally, the library relies on the DLPack format supported by both PyTorch and Cupy, which allows zero-copy inter-operation between PyTorch and spaCy's machine learning library Thinc.
>
> This should mean that the wrapper introduces negligible overhead, even though it communicates arrays between different libraries. However, the wrapping strategy does have some current drawbacks. The main one is that PyTorch and Cupy each use a different allocation cache, leading to out-of-memory errors on workloads that might have been fine with PyTorch alone. Multiple GPUs are also not currently supported. We are working on both of these issues.

The most important features are the **raw outputs of the transformer**, which can be accessed at `doc._.trf_outputs.last_hidden_state`. This variable gives you a tensor with one row per wordpiece token. The `doc.tensor` attribute gives you one row per spaCy token, which is useful if you're working on token-level tasks such as part-of-speech tagging or spelling correction. We've taken care to **calculate an alignment** between the models' various wordpiece tokenization schemes and spaCy's linguistically-motivated tokenization, with a weighting scheme that ensures that no information is lost.

We've also tried to make sure that preprocessing details such as the boundary tokens are handled correctly for each of the different models. Seemingly small details such as whether the "class" token should be placed at the beginning (as for BERT) or the end (as for XLNet) of the sentence can make a big difference for effective fine-tuning. If the inputs to the transformer don't match how it was pretrained, it will have to rely much more on your small labelled training corpus, leading to lower accuracies. We hope that providing a more **unified interface to the transformer models** will also help researchers, as well as production users. The aligned tokenization should be especially helpful for answering questions like "Do these two transformers pay attention to the same words?". You could also add spaCy's tagger, parser and entity recognizer to the pipeline, which would allow you to ask questions like "Does the attention pattern change when there's a syntactic ambiguity?".

Transfer learning

The main use case for pretrained transformer models is transfer learning. You load in a **large generic model** pretrained on lots of text, and start training on your smaller dataset with **labels specific to your problem**. The `spacy-transformers` package has custom pipeline components that make this especially easy. We provide an example component for text categorization. Development of analogous components for other tasks should be quite straightforward.

The `trf_textcat` component is based on spaCy's built-in `TextCategorizer` and supports using the features assigned by the transformer models, via the `trf_tok2vec` component. This lets you use a model like BERT to predict contextual token representations, and then learn a text categorizer on top as a task-specific "head". The API is the same as any other spaCy pipeline:

Example data

```
TRAIN_DATA = [
    ("text1", {"cats": {"POSITIVE": 1.0, "NEGATIVE": 0.0}})
]
```

Training loop

```
import spacy
from spacy.util import minibatch
import random
import torch

is_using_gpu = spacy.prefer_gpu()
if is_using_gpu:
    torch.set_default_tensor_type("torch.cuda.FloatTensor")

nlp = spacy.load("en_trf_bertbaseuncased_lg")
print(nlp.pipe_names) # ["sentencizer", "trf_wordpiecer", "trf_tok2vec"]
textcat = nlp.create_pipe("trf_textcat", config={"exclusive_classes": True})
for label in ("POSITIVE", "NEGATIVE"):
    textcat.add_label(label)
nlp.add_pipe(textcat)

optimizer = nlp.resume_training()
for i in range(10):
```

```
        random.shuffle(TRAIN_DATA)
        losses = {}
        for batch in minibatch(TRAIN_DATA, size=8):
            texts, cats = zip(*batch)
            nlp.update(texts, cats, sgd=optimizer, losses=losses)
        print(i, losses)
    nlp.to_disk("/bert-textcat")
```

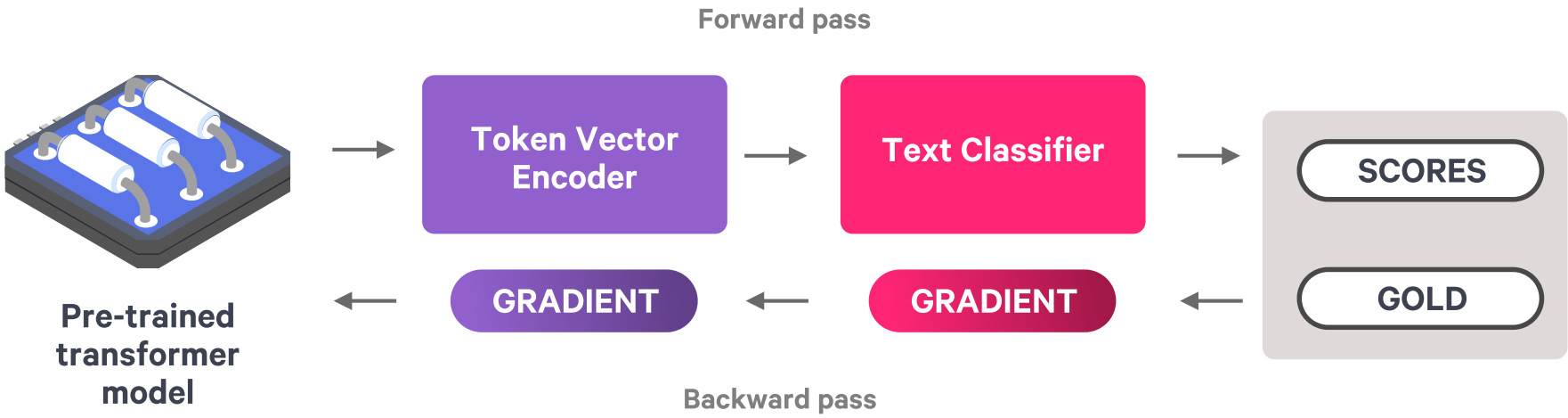
> **About the example**
>
> This example has been reduced to show the most minimal workflow. For a more full-featured training loop, see the `train_textcat.py` example script, which shows how to train a text classifier on the IMDB data.

We're still testing and refining the workflows around this, and there are a number of features we haven't implemented yet. The most important feature we're currently missing is **support for all of the model outputs**: we currently only pass through the last hidden state. The full hidden-state activations and attention matrices should be available shortly. There are also a number of options that we still need to expose in the API, especially the ability to configure **whether updates are propagated** back into the transformer model.

We're especially looking forward to rolling out support for these transformer models in our annotation tool Prodigy. When we designed Prodigy, one of our core assumptions was that **a little supervision could go a long way**. With **good tooling** and a **well-factored annotation scheme**, you don't need to annotate millions of data points – which means you don't need to frame annotation tasks as low-value click-work. Modern transfer learning techniques are bearing this out. Xie et al. (2019) have shown that a transformer models trained on only 1% of the IMDB sentiment analysis data (just a few dozen examples) can exceed the pre-2016 state-of-the-art. While the field has moved far faster than we could have anticipated, this type of tool-assisted workflow is exactly why we designed Prodigy to be scriptable and developer-friendly.

Fine-tuning pretrained transformer models on your task

Peters et al. (2019) performed a detailed investigation of two transfer learning approaches: fine-tuning (🔥) and feature-extraction (❄️). They found that there are advantages to both approaches, with ❄️ having practical advantages and sometimes out-performing 🔥 in accuracy depending on the task and dataset. The current text classification model uses 🔥, and follows Devlin et al. (2018) in using the vector for the class token to represent the sentence, and passing this vector forward into a softmax layer in order to perform classification. For multi-document sentences, we perform mean pooling on the softmax outputs.



Pipeline components can **communicate gradients back to the transformer** by incrementing the `doc._.trf_d_last_hidden_state` attribute, which is a numpy/cupy array that holds the gradient with respect to the last hidden state of the transformer. To implement a custom component for a new task, you should create a new subclass of `spacy.pipeline.Pipe` that defines the `Model`, `predict`, `set_annotations` and `update` methods. During the `update` method, your component will receive a batch of `Doc` objects that hold the transformer features, and a batch of `GoldParse` objects that contain the gold-standard annotations. Using these inputs, you should update your model and increment the gradient for the last-hidden states on the `doc._.trf_last_hidden_state` variable. Your subclass can handle its model any way you like, but the easiest approach if you're using PyTorch is to use Thinc's PyTorch wrapper class, which will save you from having to implement the to/from bytes/disk serialization methods. We expect to publish a full tutorial with the recommended workflow in future.

Alignment of wordpieces and outputs to linguistic tokens

Transformer models are usually trained on text preprocessed with the "wordpiece" algorithm, which limits the number of distinct token-types the model needs to consider. Wordpiece is convenient for training neural networks, but it doesn't produce segmentations that match up to any linguistic notion of a "word". Most rare words will map to multiple wordpiece tokens, and occasionally the alignment will be many-to-many. Here's an example showing the wordpiece tokens produced by the different pretrained models using a text snippet from the IMDB dataset:

Wordpieces

```
# bert-base-uncased
['[CLS]', 'laced', 'with', 'dreams', '-', 'dripping', 'in', 'reality', ',', 'the', 'american', 'dream', 'reign', '##ites',
# gpt2
['<|endoftext|>', 'L', 'aced', 'Ġwith', 'Ġdreams', 'Ġ-', 'Ġdripping', 'Ġin', 'Ġreality', ',', 'Ġthe', 'ĠAmerican', 'ĠDrea
# xlnet-base-cased
['<cls>', '▁Lac', 'ed', '▁with', '▁dreams', '▁', '-', '▁dripping', '▁in', '▁reality', ',', '▁the', '▁American', '▁Dream',
# xlm-mlm-enfr-1024
['<s>', 'laced</w>', 'with</w>', 'dreams</w>', '-</w>', 'dri', 'pping</w>', 'in</w>', 'reality</w>', ',</w>', 'the</w>',
```

> **Why the weird symbols?**
>
> Wordpiece tokenizers generally record the positions of whitespace, so that sequences of wordpiece tokens can be assembled back into normal strings. The details of how the whitespace is recorded vary, however. The BERT tokenizer inserts `##` into words that don't begin on whitespace, while the GPT-2 tokenizer uses the character `Ġ` to stand in for spaces. Most also perform some unicode and whitespace normalization. Finally, most transformers rely on special control tokens like `[CLS]` that should occur around the string, which are important to support sentence-pair and text-classification tasks.

Wordpieces (cleaned)

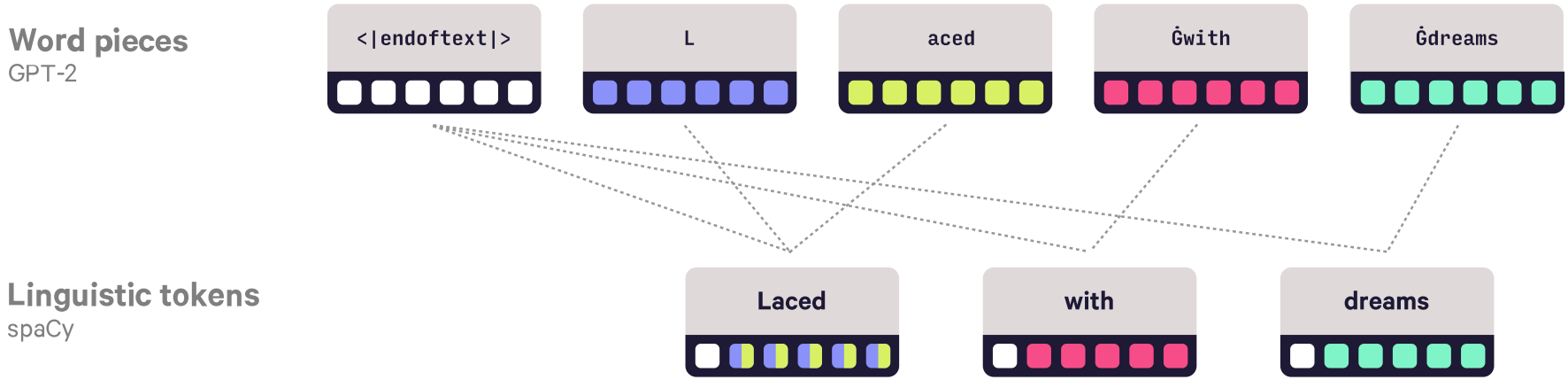
```
# bert-base-uncased
['laced', 'with', 'dreams', '-', 'dripping', 'in', 'reality', ',', 'the', 'american', 'dream', 'reign', 'ites', 'after',
# gpt2
['L', 'aced', 'with', 'dreams', '-', 'dripping', 'in', 'reality', ',', 'the', 'American', 'Dream', 'reign', 'ites', 'afte
# xlnet-base-cased
['Lac', 'ed', 'with', 'dreams', '', '-', 'dripping', 'in', 'reality', ',', 'the', 'American', 'Dream', 'reign', 'ites', 'a
# xlm-mlm-enfr-1024
```

```
['Laced', 'with', 'dreams', '-', 'dri', 'pping', 'in', 'reality', ',', 'the', 'americ', 'an', 'dream', 're', 'ign', 'ites'
```

spaCy tokens

```
['Laced', 'with', 'dreams', '-', 'dripping', 'in', 'reality', ',', 'the', 'American', 'Dream', 'reignites', 'after', '9.1
```

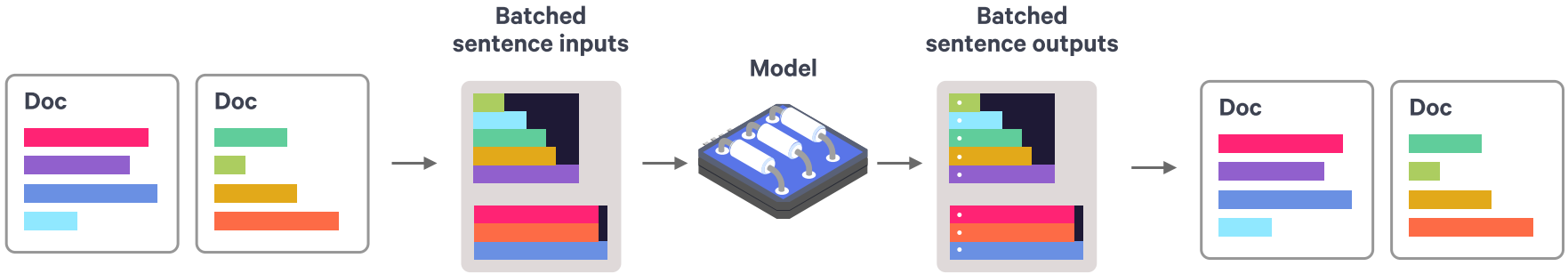
As you can see, the segmentation defined by the wordpieces isn't very close to the linguistic notion of a "word". Many of the segmentations are quite surprising, such as the decision by the GPT-2 classifier to divide "Laced" into two tokens: "L" and "aced". The priority of wordpiece tokenizers is to limit the vocabulary size, as vocabulary size is one of the key challenges facing current neural language models (Yang et al., 2017). While it has undoubtedly proven an effective technique for model training, linguistic tokens provide **much better interpretability and interoperability**. This is especially true in light of the differences between the various wordpiece tokenizations: each model requires its own segmentation, and the next model will undoubtedly require segmentation that is different again.



We've taken care to calculate the aligned `doc.tensor` representation as faithfully as possible, with priority given to avoiding information loss. To make this work, each row of the tensor (which corresponds to a spaCy token) is set to a **weighted sum of the rows** of the `last_hidden_state` tensor that the token is aligned to, where the weighting is proportional to the number of other spaCy tokens aligned to that row. To include the information from the (often important – see Clark et al., 2019) boundary tokens, we imagine that these are also "aligned" to all of the tokens in the sentence. The implementation of this weighting scheme can be found in the `TransformersTok2Vec.set_annotations` method.

Batching, padding and per-sentence processing

Transformer models have cubic runtime and memory complexity with respect to sequence length. This means that longer texts **need to be divided into sentences** in order to achieve reasonable efficiency. `spacy-transformers` handles this internally, and requires a sentence-boundary detection to be present in the pipeline. We recommend spaCy's built-in sentencizer component. Internally, the transformer model will predict over sentences, and the resulting tensor features will be reconstructed to produce document-level annotations. In order to further improve efficiency and reduce memory requirements, we also perform **length-based subbatching** internally.



The subbatching regroups the batched sentences by sequence length, to minimize the amount of padding required. The default value of 3000 words per batch works reasonably well on a Tesla V100. Many of the pretrained transformer models have a maximum sequence length. If a sentence is longer than the maximum, it is truncated and the affected ending tokens will receive zeroed vectors.

Conclusion

Training large transformer models currently requires **considerable computational resources**. Even if the compute resources were otherwise sitting idle, the energy expenditure alone is considerable. Strubell (2019) calculates that pretraining the BERT base model produces carbon emissions roughly equal to a transatlantic flight. As Sebastian Ruder emphasized well in his keynote at spaCy IRL, it is therefore important that the pretrained weights distributed for these models be **widely reused where possible**, rather than recalculated.

However, reusing pretrained weights effectively can be a delicate operation. Models are typically trained on text that has been uniformly preprocessed, which can make them sensitive to even small differences between training and run-time. The effects of these preprocessing problems can be difficult to predict, and difficult

make them sensitive to even small differences between training and run-time. The effects of these preprocessing problems can be difficult to predict, and difficult to reason about. Hugging Face's `transformers` library has already gone a long way to solving this problem, by making it easy to use the pretrained models and tokenizers with fairly consistent interfaces. However, there are still a number of preprocessing details that need to be done to achieve optimal performance. We hope that our wrapping library will prove useful in this respect.

SPACY-TRANSFORMERS SPACY

About the author

Matthew Honnibal

Matthew is a leading expert in AI technology. He completed his PhD in 2009, and spent a further 5 years publishing research on state-of-the-art NLP systems. He left academia in 2014 to write spaCy and found Explosion.

@honnibal honnibal

About the author

Ines Montani

Ines is a co-founder of Explosion and a core developer of the spaCy NLP library and the Prodigy annotation tool. She has helped set a new standard for user experience in developer tools for AI engineers and researchers.

@_inesmontani ines

Read more

Explosion in 2019: Our Year in Review

sense2vec reloaded: contextually-keyed word vectors

Introducing spaCy v2.2

Introducing spaCy v2.1

Explosion in 2017: Our Year in Review

Introducing custom pipelines and extensions for spaCy v2.0

Navigation

Home
About
Software
Demos
Blog
Legal / Imprint

Our Software

spaCy
Industrial-strength NLP

Prodigy
Radically efficient machine teaching

See more →