BERT (Bidirectional Encoder Representation from Transformers)

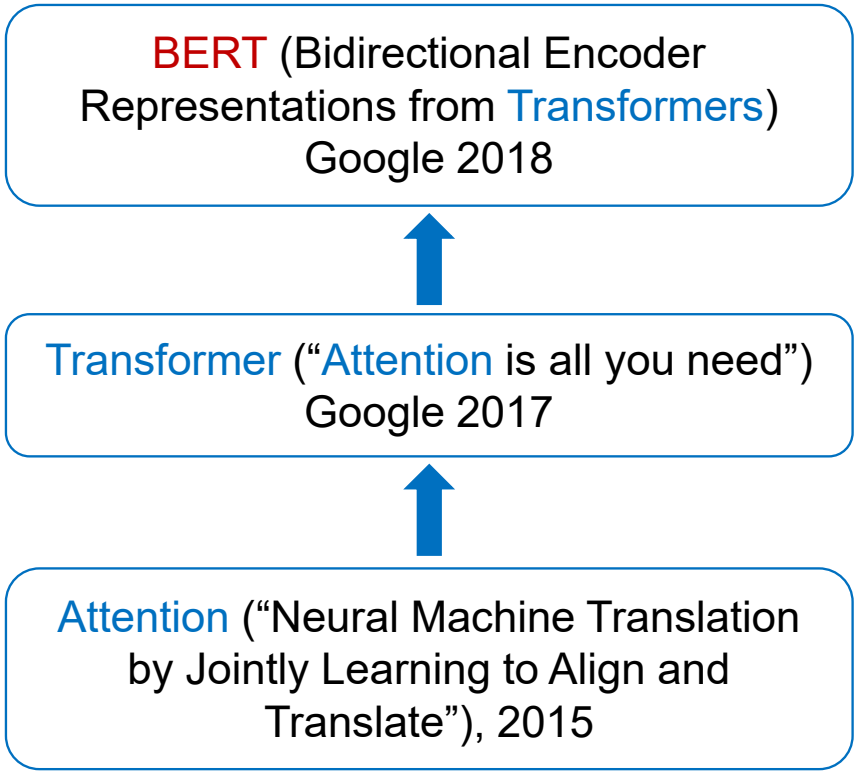
谢海华 助理研究员

北京雁栖湖应用数学研究院

大数据及人工智能

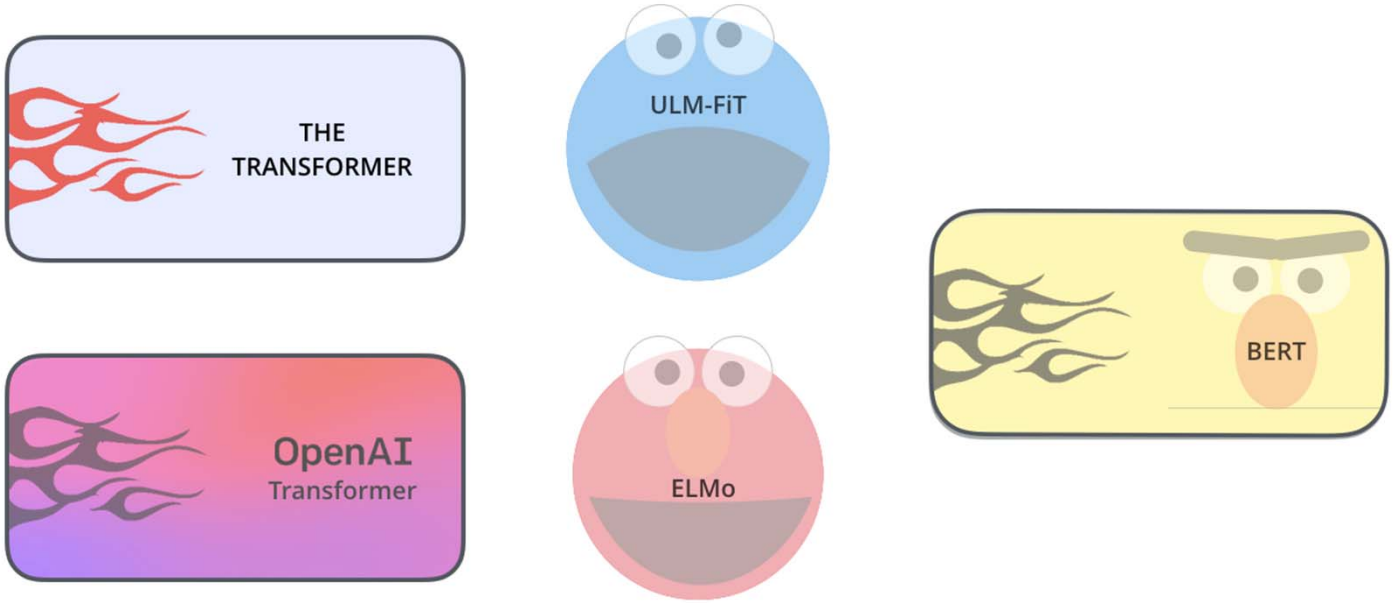
2022.04

The BERT


BERT (Bidirectional Encoder Representations from Transformers) Google 2018
Transformer ("Attention is all you need") Google 2017
Attention ("Neural Machine Translation by Jointly Learning to Align and Translate"), 2015

Brief Introduction to BERT

Our conceptual understanding of how best to represent words and sentences in a way that best captures underlying meanings and relationships is rapidly evolving. Moreover, the NLP community has been putting forward incredibly powerful components that you can freely download and use in your own models and pipelines.



Brief Introduction to BERT

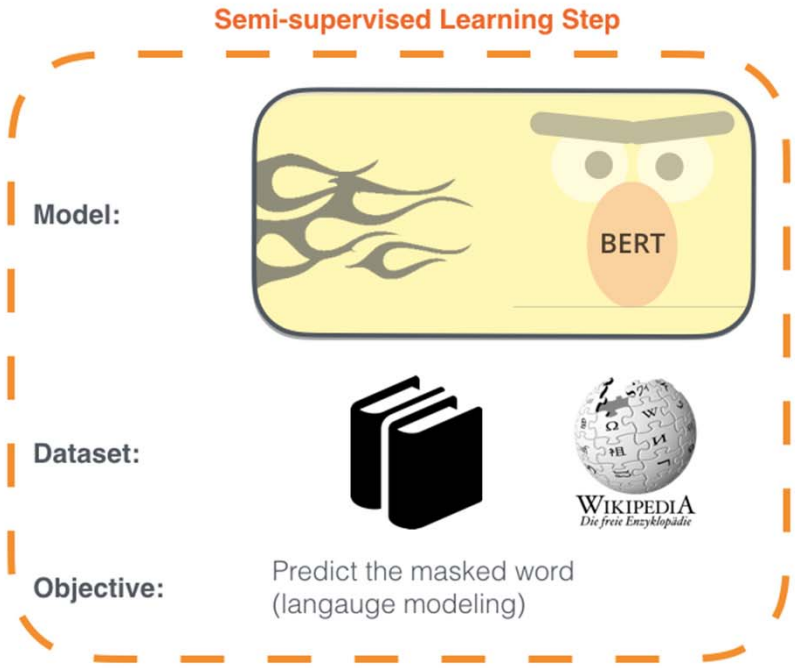
BERT is a model that broke several records for how well models can handle language-based tasks. Soon after the release of the paper describing the model, the team also open-sourced the code of the model, and made available for download versions of the model that were already pre-trained on massive datasets. This is a momentous development since it enables anyone building a machine learning model involving language processing to use this powerhouse as a readily-available component – saving the time, energy, knowledge, and resources that would have gone to training a language-processing model from scratch.

BERT builds on top of a number of clever ideas that have been bubbling up in the NLP community recently – including but not limited to Semi-supervised Sequence Learning, ELMo, ULMFiT, the OpenAI transformer, and the Transformer.

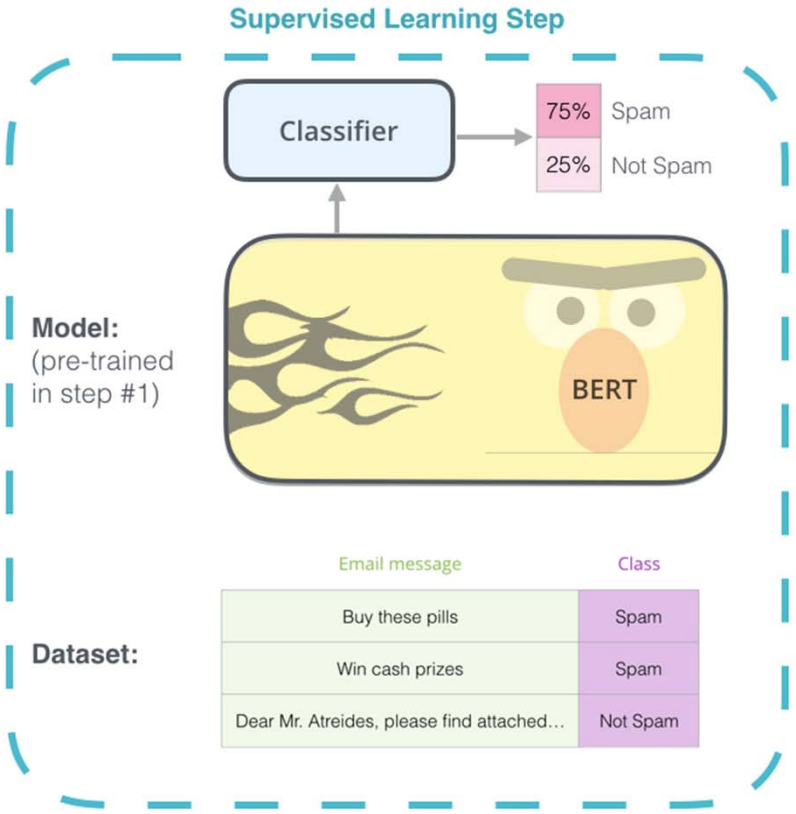
Brief Introduction to BERT

1 - Semi-supervised training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.

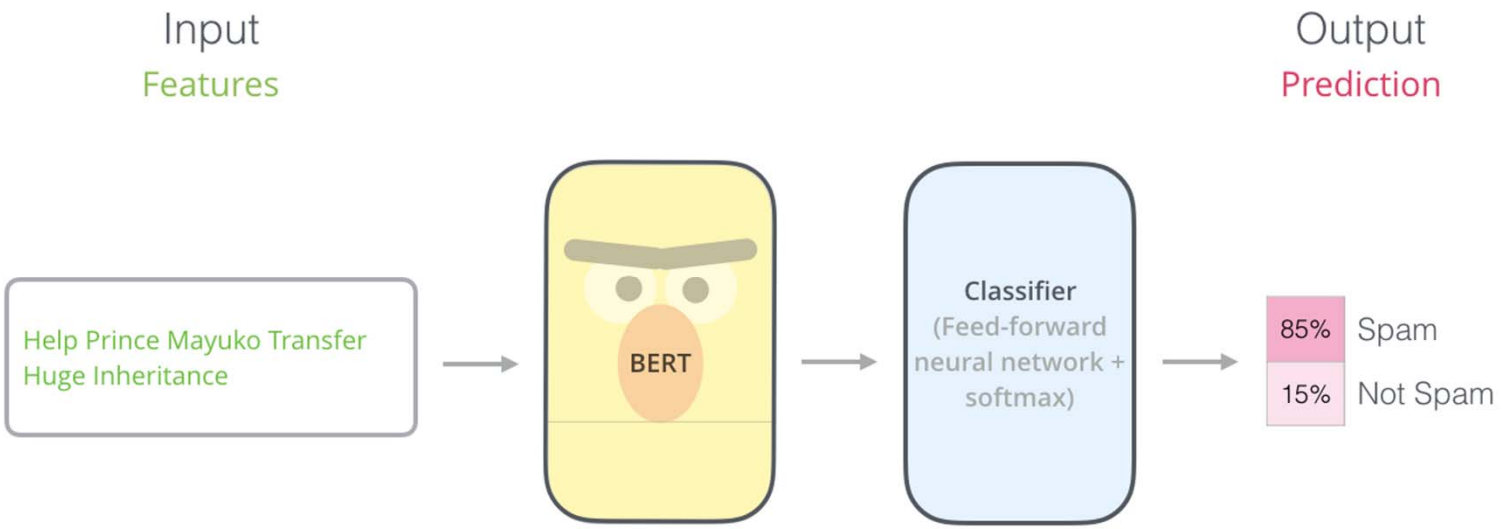


2 - Supervised training on a specific task with a labeled dataset.



Example: Sentence Classification

The most straight-forward way to use BERT is to use it to classify a single piece of text. This model would look like this:



To train such a model, you mainly have to train the classifier, with minimal changes happening to the BERT model during the training phase. This training process is called Fine-Tuning, and has roots in Semi-supervised Sequence Learning and ULMFiT.

Example: Sentence Classification

For this spam classifier example, the labeled dataset would be a list of email messages and a label (“spam” or “not spam” for each message).

Email message	Class
Buy these pills	Spam
Win cash prizes	Spam
Dear Mr. Atreides, please find attached…	Not Spam

Example: Sentence Classification

Other examples for such a use-case include:

- **Sentiment analysis**
  - Input: Movie/Product review. Output: is the review positive or negative?
  - Example dataset: SST
- **Fact-checking**
  - Input: sentence. Output: “Claim” or “Not Claim”
  - More ambitious/futuristic example:
    - Input: Claim sentence. Output: “True” or “False”
  - Full Fact is an organization building automatic fact-checking tools for the benefit of the public. Part of their pipeline is a classifier that reads news articles and detects claims (classifies text as either “claim” or “not claim”) which can later be fact-checked (by humans now, with ML later, hopefully).
  - Video: Sentence embeddings for automated factchecking - Lev Konstantinovskiy.

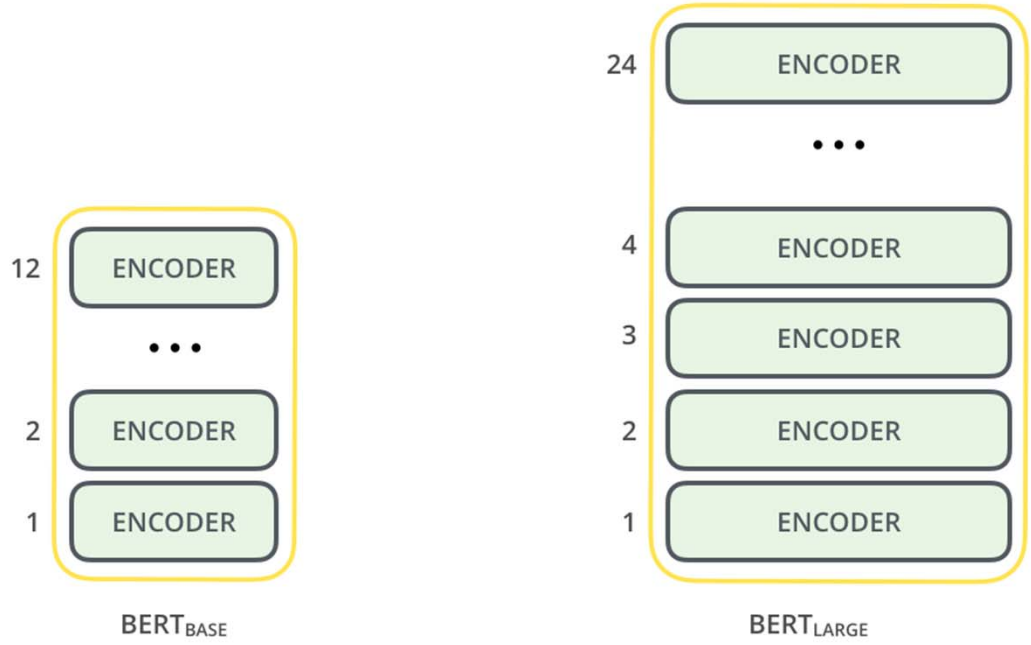
Model Architecture

Two model sizes for BERT:

- BERT BASE – Comparable in size to the OpenAI Transformer in order to compare performance
- BERT LARGE – A ridiculously huge model which achieved the state of the art results

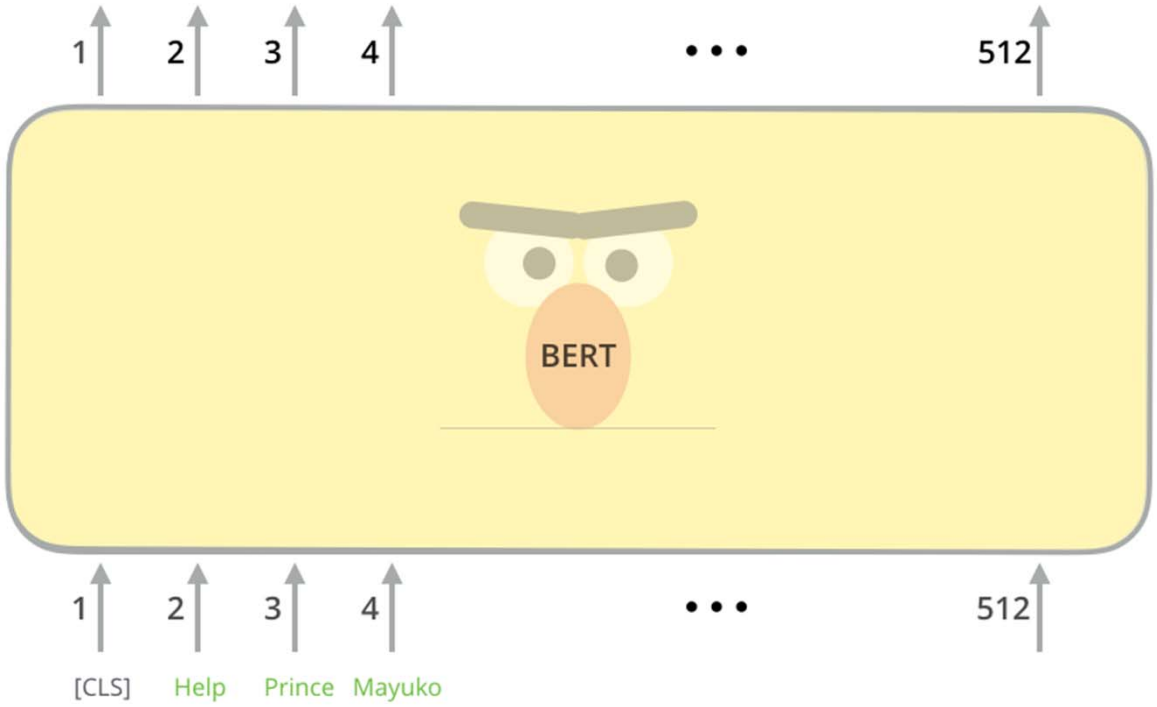
Model Architecture

BERT is basically a trained Transformer Encoder stack. Both BERT model sizes have a large number of encoder layers (which the paper calls Transformer Blocks) – twelve for the Base version, and twenty four for the Large version. These also have larger feedforward-networks (768 and 1024 hidden units respectively), and more attention heads (12 and 16 respectively) than the default configuration in the reference implementation of the Transformer in the initial paper (6 encoder layers, 512 hidden units, and 8 attention heads).



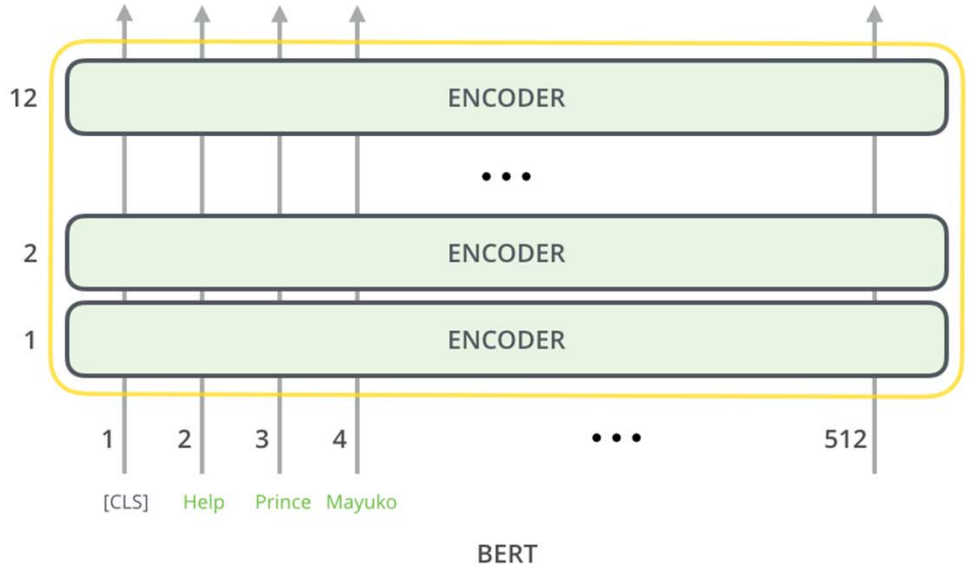
Model Inputs

The first input token is supplied with a special [CLS] token for reasons that will become apparent later on. CLS here stands for Classification.



Model Inputs

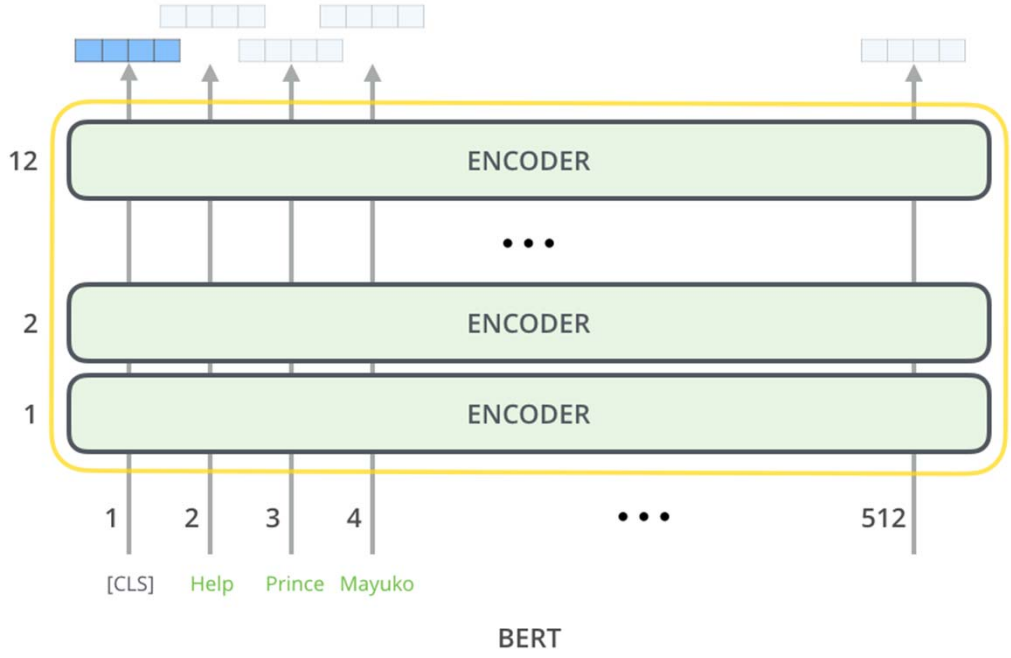
Just like the vanilla encoder of the transformer, BERT takes a sequence of words as input which keep flowing up the stack. Each layer applies self-attention, and passes its results through a feed-forward network, and then hands it off to the next encoder.



In terms of architecture, this has been identical to the Transformer up until this point (aside from size, which are just configurations we can set). It is at the output that we first start seeing how things diverge.

Model Outputs

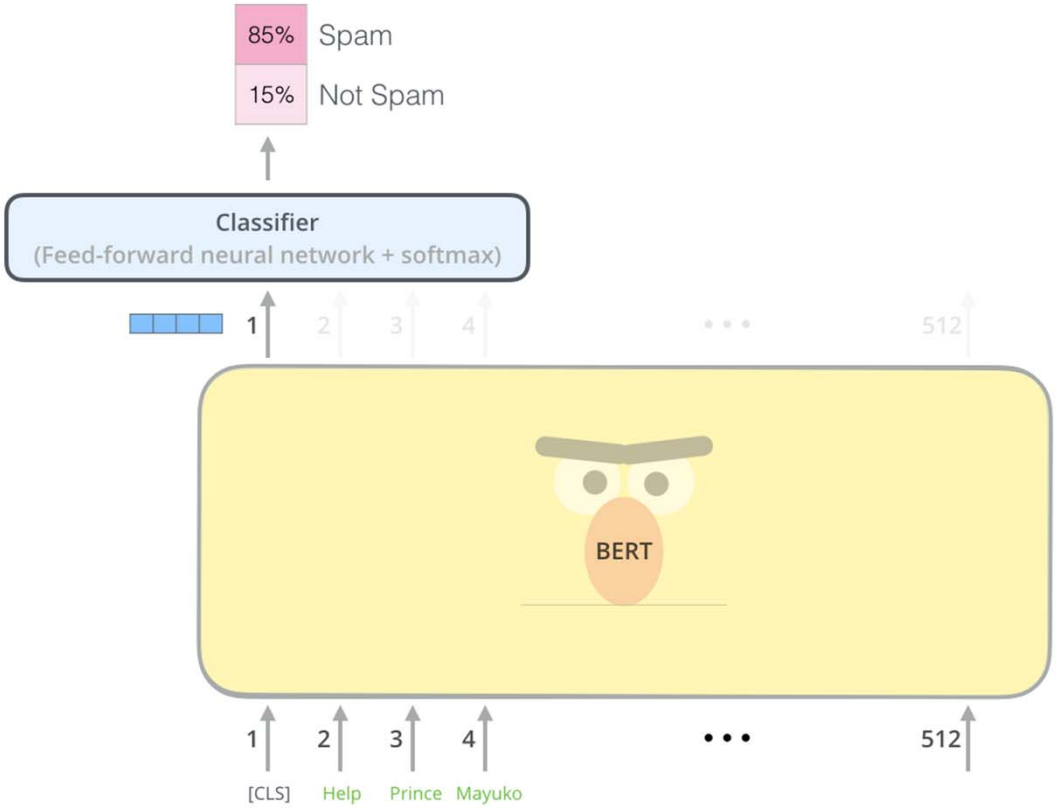
Each position outputs a vector of size *hidden_size* (768 in BERT Base). For the sentence classification example we've looked at above, we focus on the output of only the first position (that we passed the special [CLS] token to).



Model Outputs

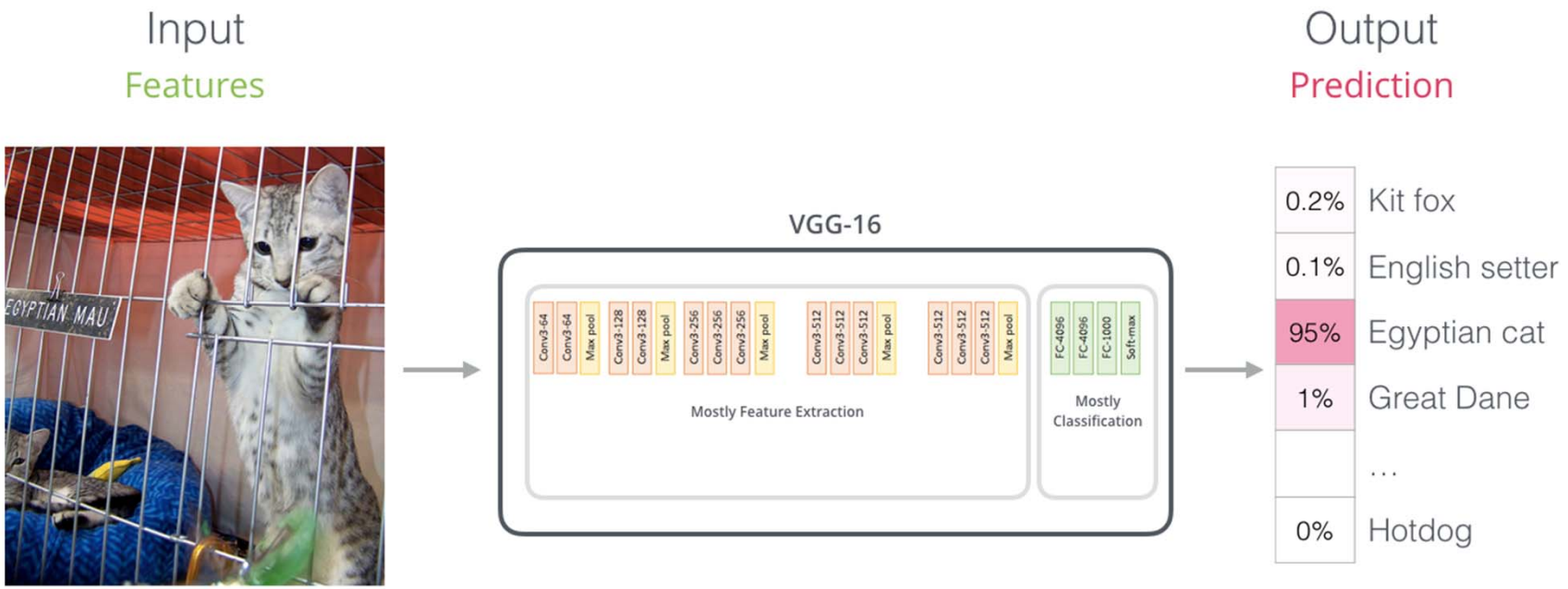
That vector can now be used as the input for a classifier of our choosing. The paper achieves great results by just using a single-layer neural network as the classifier.

If you have more labels (for example if you're an email service that tags emails with "spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons that then pass through softmax.



Parallels with Convolutional Nets

For those with a background in computer vision, this vector hand-off should be reminiscent of what happens between the convolution part of a network like VGGNet and the fully-connected classification portion at the end of the network.



A New Age of Embedding

Word Embedding Recap

For words to be processed by machine learning models, they need some form of numeric representation that models can use in their calculation. Word2Vec showed that we can use a vector (a list of numbers) to properly represent words in a way that captures *semantic* or meaning-related relationships (e.g. the ability to tell if words are similar, or opposites, or that a pair of words like “Stockholm” and “Sweden” have the same relationship between them as “Cairo” and “Egypt” have between them) as well as syntactic, or grammar-based, relationships (e.g. the relationship between “had” and “has” is the same as that between “was” and “is”).

The field quickly realized it’s a great idea to use embeddings that were pre-trained on vast amounts of text data instead of training them alongside the model on what was frequently a small dataset. So it became possible to download a list of words and their embeddings generated by pre-training with Word2Vec or GloVe.

-0.34	-0.84	0.20	-0.26	-0.12	0.23	1.04	-0.16	0.31	0.06	0.30	0.33	-1.17	-0.30	0.03	0.09	0.35	-0.28	-(

The GloVe word embedding of the word "stick" - a vector of 200 floats (rounded to two decimals). It goes on for two hundred values.

ELMo: Context Matters

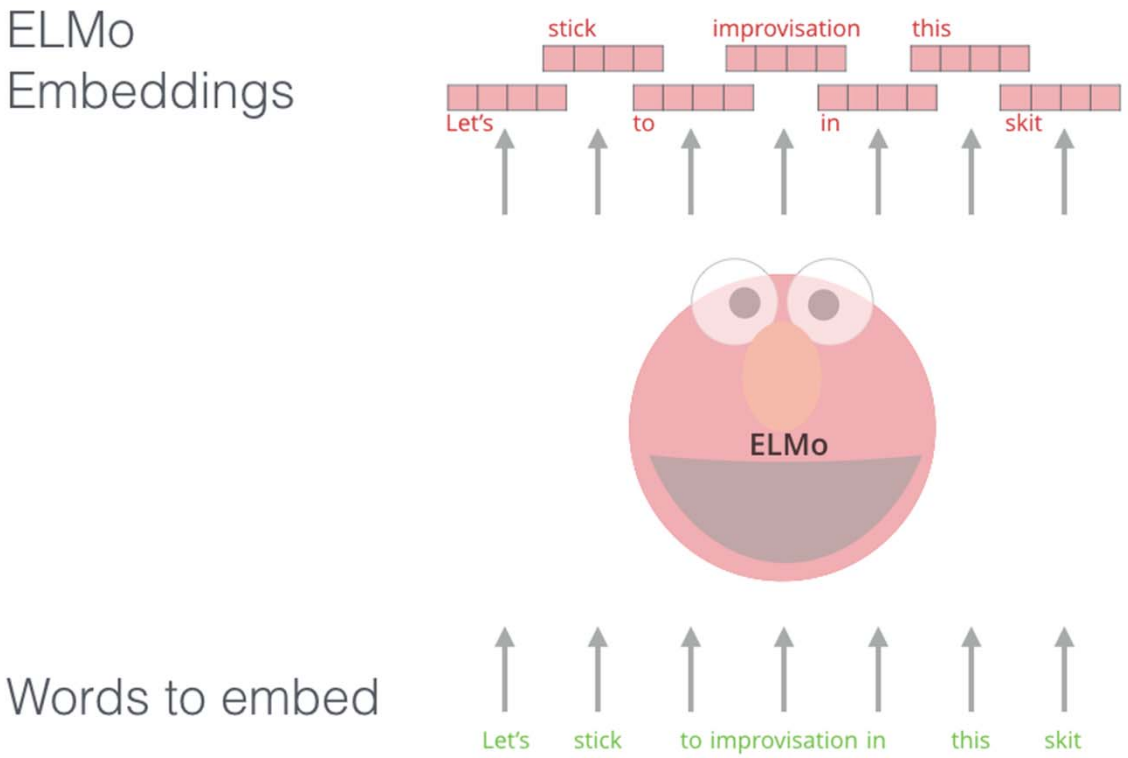
If we're using this GloVe representation, then the word "stick" would be represented by this vector no-matter what the context was. "Wait a minute" said a number of NLP researchers, "*stick*"" has multiple meanings depending on where it's used. Why not give it an embedding based on the context it's used in – to both capture the word meaning in that context as well as other contextual information?". And so, *contextualized* word-embeddings were born.

Contextualized word-embeddings can give words different embeddings based on the meaning they carry in the context of the sentence.

ELMo: Context Matters

Instead of using a fixed embedding for each word, ELMo looks at the entire sentence before assigning each word in it an embedding. It uses a bi-directional LSTM trained on a specific task to be able to create those embeddings.

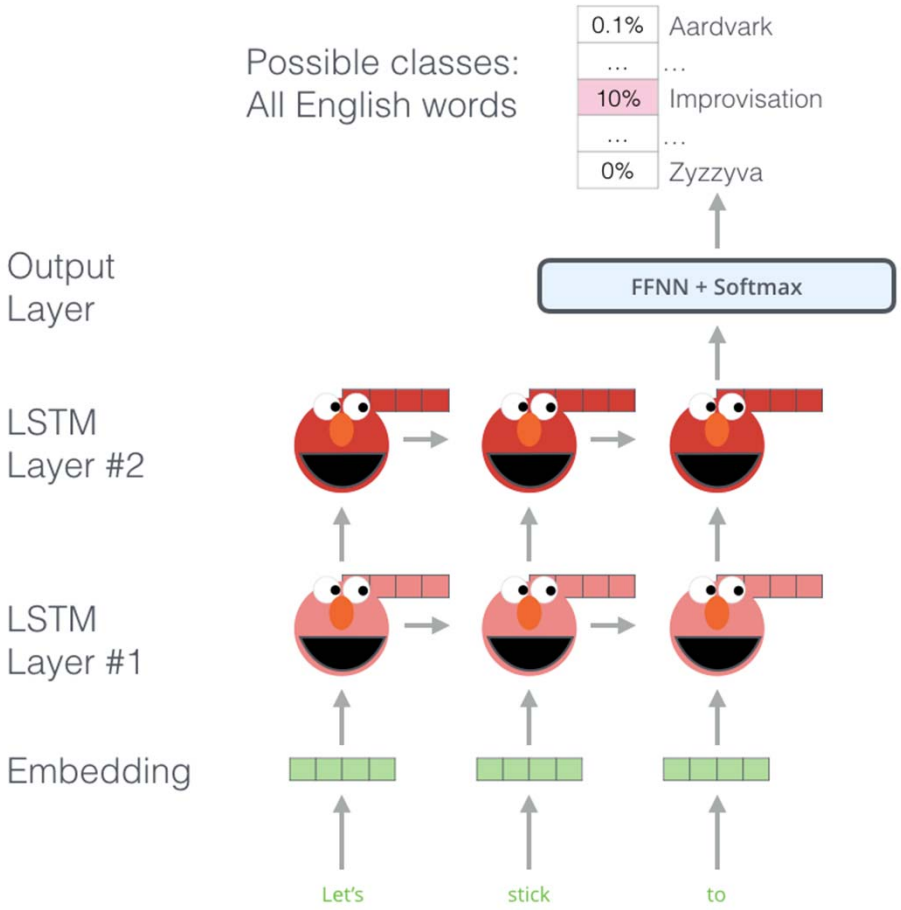
ELMo provided a significant step towards pre-training in the context of NLP. The ELMo LSTM would be trained on a massive dataset in the language of our dataset, and then we can use it as a component in other models that need to handle language.



ELMo: Context Matters

What's ELMo's secret?

ELMo gained its language understanding from being trained to predict the next word in a sequence of words - a task called *Language Modeling*. This is convenient because we have vast amounts of text data that such a model can learn from without needing labels.

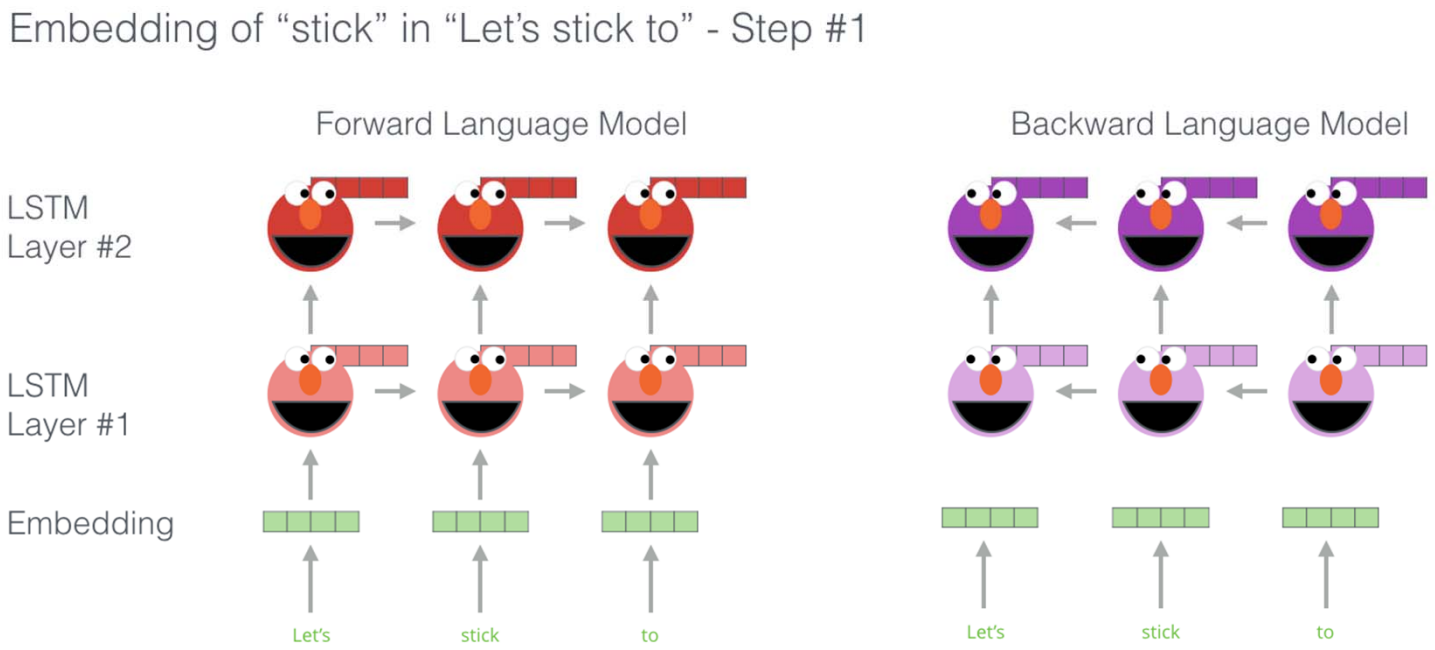


A step in the pre-training process of ELMo: Given "Let's stick to" as input, predict the next most likely word – a *language modeling* task. When trained on a large dataset, the model starts to pick up on language patterns. It's unlikely it'll accurately guess the next word in this example. More realistically, after a word such as "hang", it will assign a higher probability to a word like "out" (to spell "hang out") than to "camera".

ELMo: Context Matters

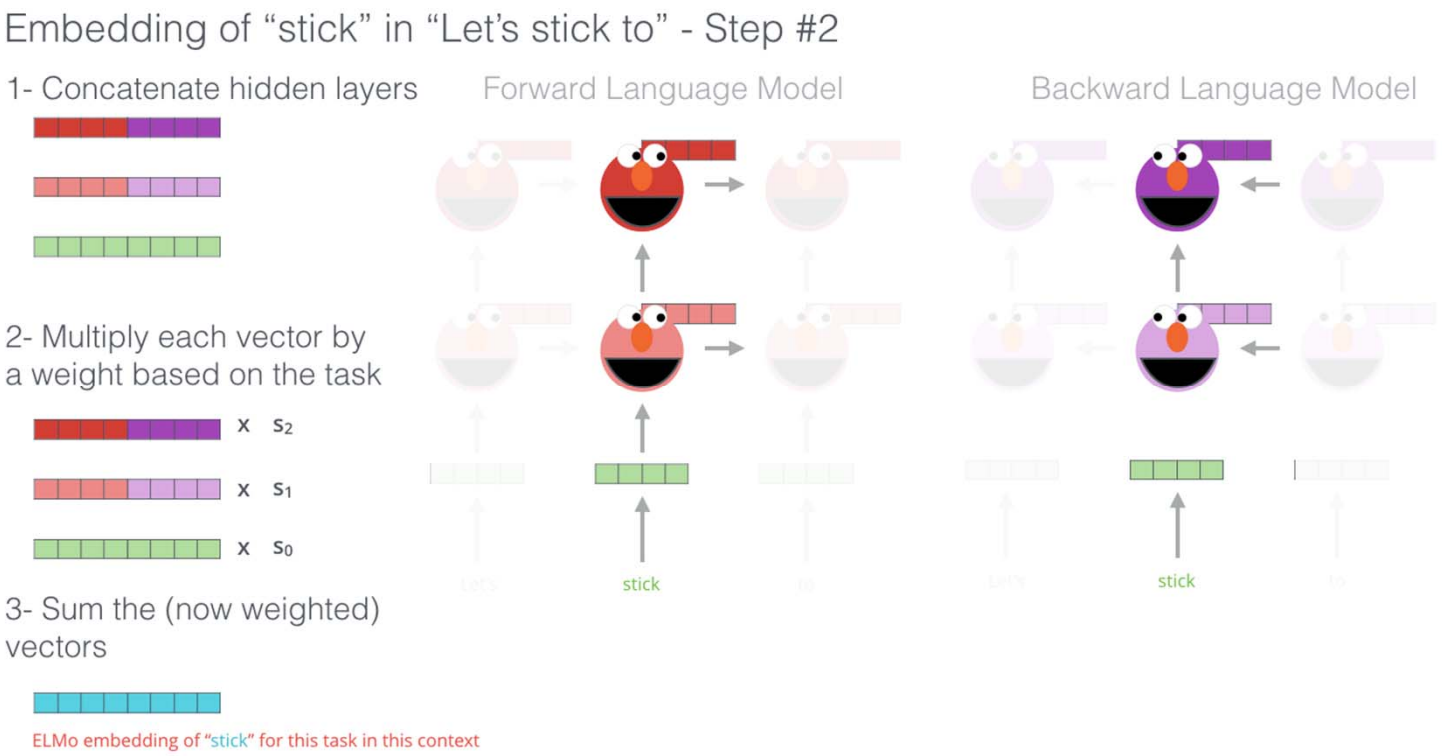
We can see the hidden state of each unrolled-LSTM step peaking out from behind ELMo's head. Those come in handy in the embedding process after this pre-training is done.

ELMo actually goes a step further and trains a bi-directional LSTM – so that its language model doesn't only have a sense of the next word, but also the previous word.



ELMo: Context Matters

ELMo comes up with the contextualized embedding through grouping together the hidden states (and initial embedding) in a certain way (concatenation followed by weighted summation).



ULM-FiT: Nailing down Transfer Learning in NLP

ULM-FiT introduced methods to effectively utilize a lot of what the model learns during pre-training – more than just embeddings, and more than contextualized embeddings. ULM-FiT introduced a language model and a process to effectively fine-tune that language model for various tasks.

NLP finally had a way to do transfer learning probably as well as Computer Vision could.

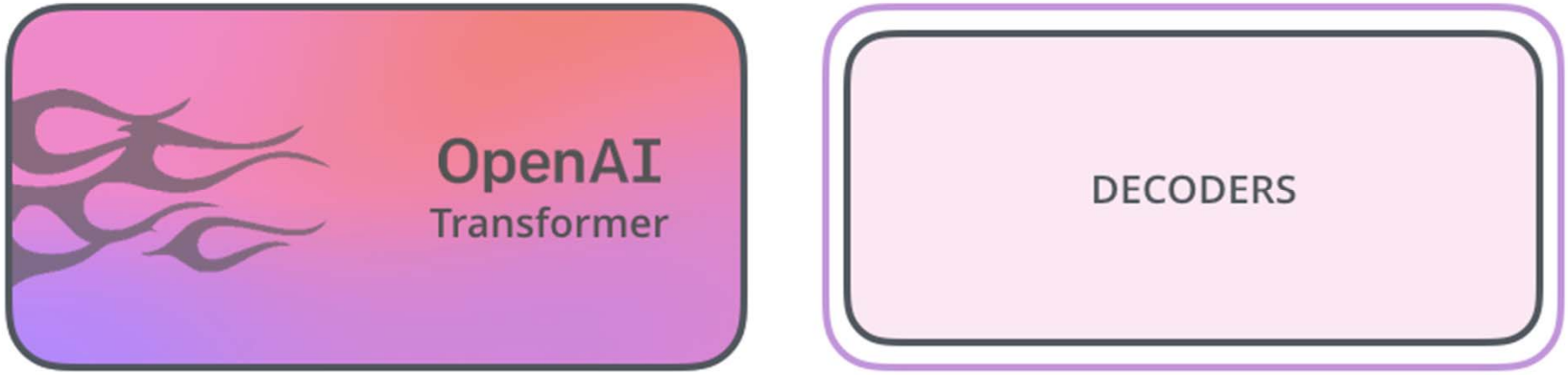
The Transformer: Going beyond LSTMs

The release of the Transformer paper and code, and the results it achieved on tasks such as machine translation started to make some in the field think of them as a replacement to LSTMs. This was compounded by the fact that Transformers deal with long-term dependencies better than LSTMs.

The Encoder-Decoder structure of the transformer made it perfect for machine translation. But how would you use it for sentence classification? How would you use it to pre-train a language model that can be fine-tuned for other tasks (*downstream* tasks is what the field calls those supervised-learning tasks that utilize a pre-trained model or component).

OpenAI Transformer: Pre-training a Transformer Decoder for Language Modeling

It turns out we don't need an entire Transformer to adopt transfer learning and a fine-tunable language model for NLP tasks. We can do with just the decoder of the transformer. The decoder is a good choice because it's a natural choice for language modeling (predicting the next word) since it's built to mask future tokens – a valuable feature when it's generating a translation word by word.



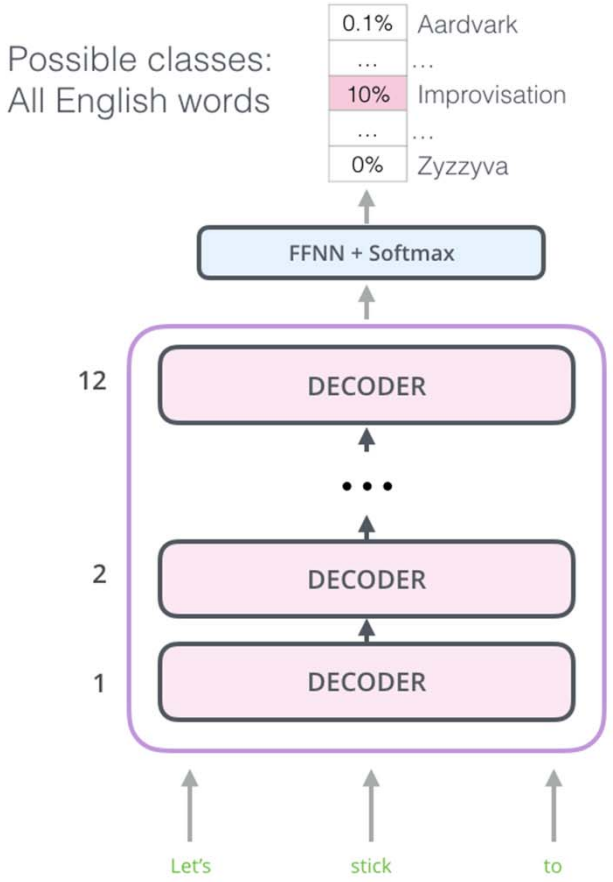
The OpenAI Transformer is made up of the decoder stack from the Transformer

OpenAI Transformer: Pre-training a Transformer Decoder for Language Modeling

The model stacked twelve decoder layers. Since there is no encoder in this set up, these decoder layers would not have the encoder-decoder attention sublayer that vanilla transformer decoder layers have. It would still have the self-attention layer, however (masked so it doesn't peak at future tokens).

With this structure, we can proceed to train the model on the same language modeling task: predict the next word using massive (unlabeled) datasets. Just, throw the text of 7,000 books at it and have it learn! Books are great for this sort of task since it allows the model to learn to associate related information even if they're separated by a lot of text – something you don't get for example, when you're training with tweets, or articles..

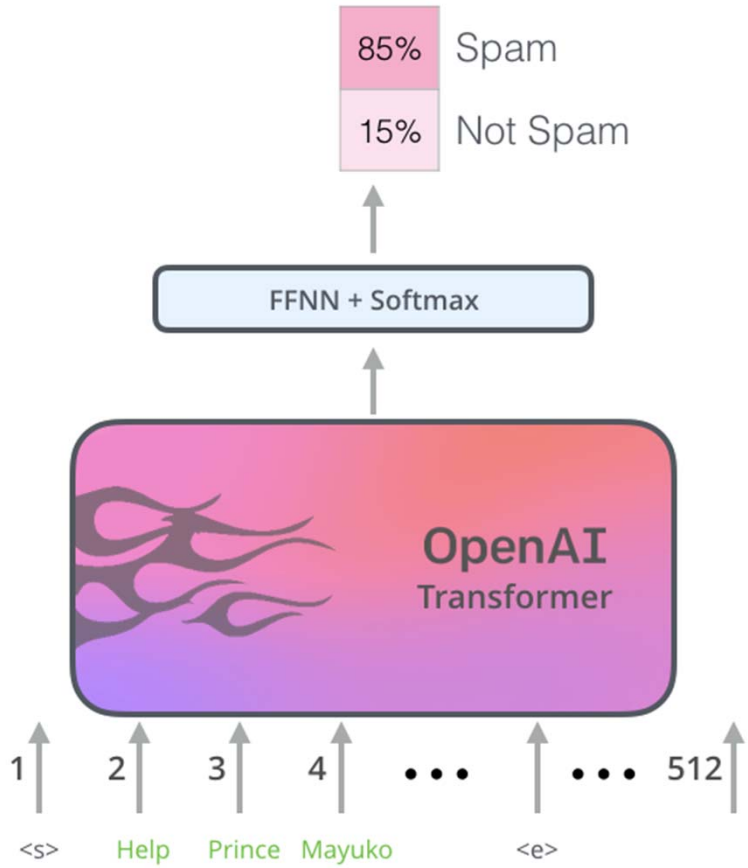
The OpenAI Transformer is now ready to be trained to predict the next word on a dataset made up of 7,000 books.



Transfer Learning to Downstream Tasks

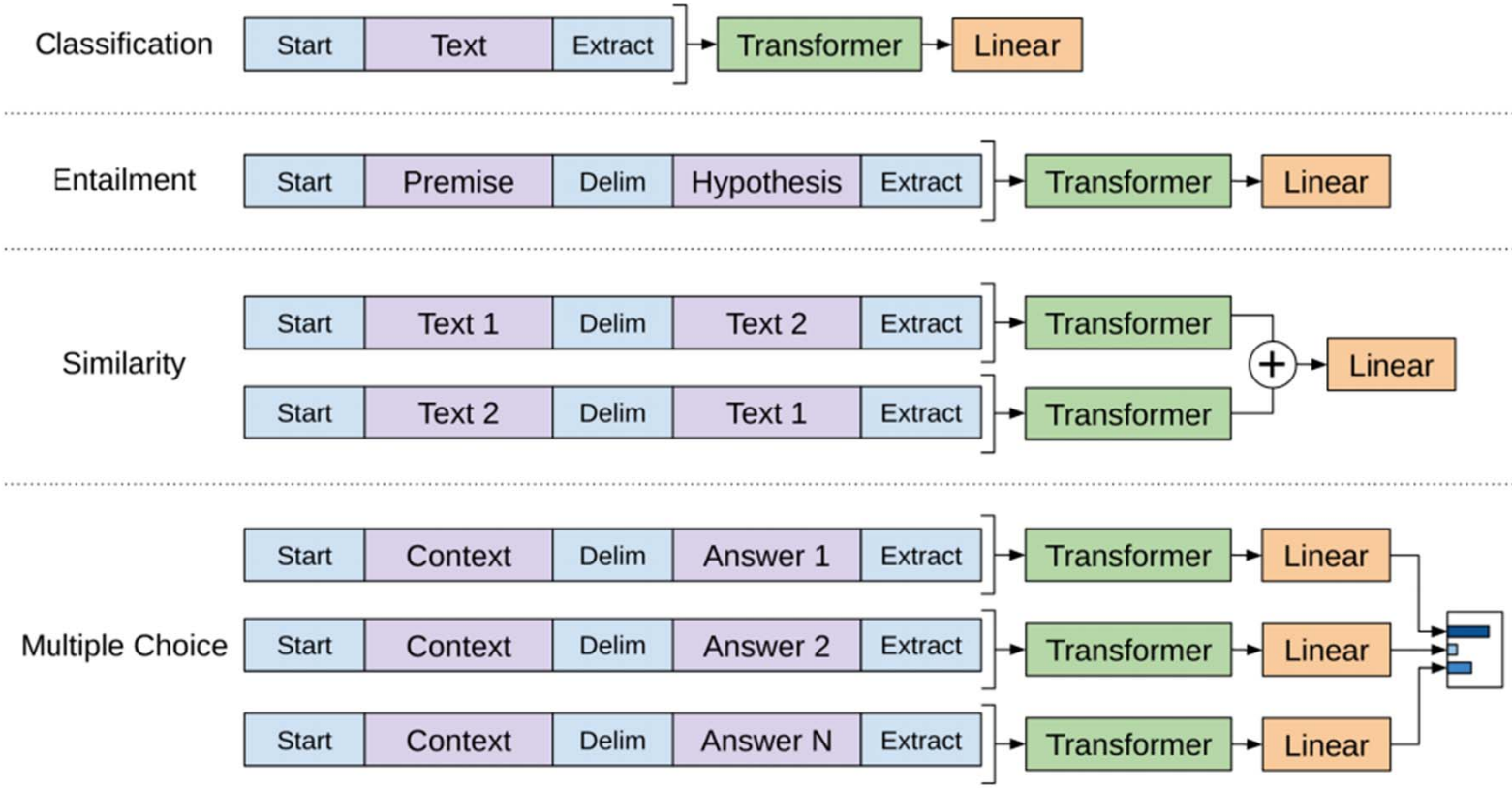
Now that the OpenAI transformer is pre-trained and its layers have been tuned to reasonably handle language, we can start using it for downstream tasks. Let's first look at sentence classification (classify an email message as "spam" or "not spam"):

How to use a pre-trained OpenAI transformer to do sentence classification



Transfer Learning to Downstream Tasks

The OpenAI paper outlines a number of input transformations to handle the inputs for different types of tasks. The following image from the paper shows the structures of the models and input transformations to carry out different tasks.



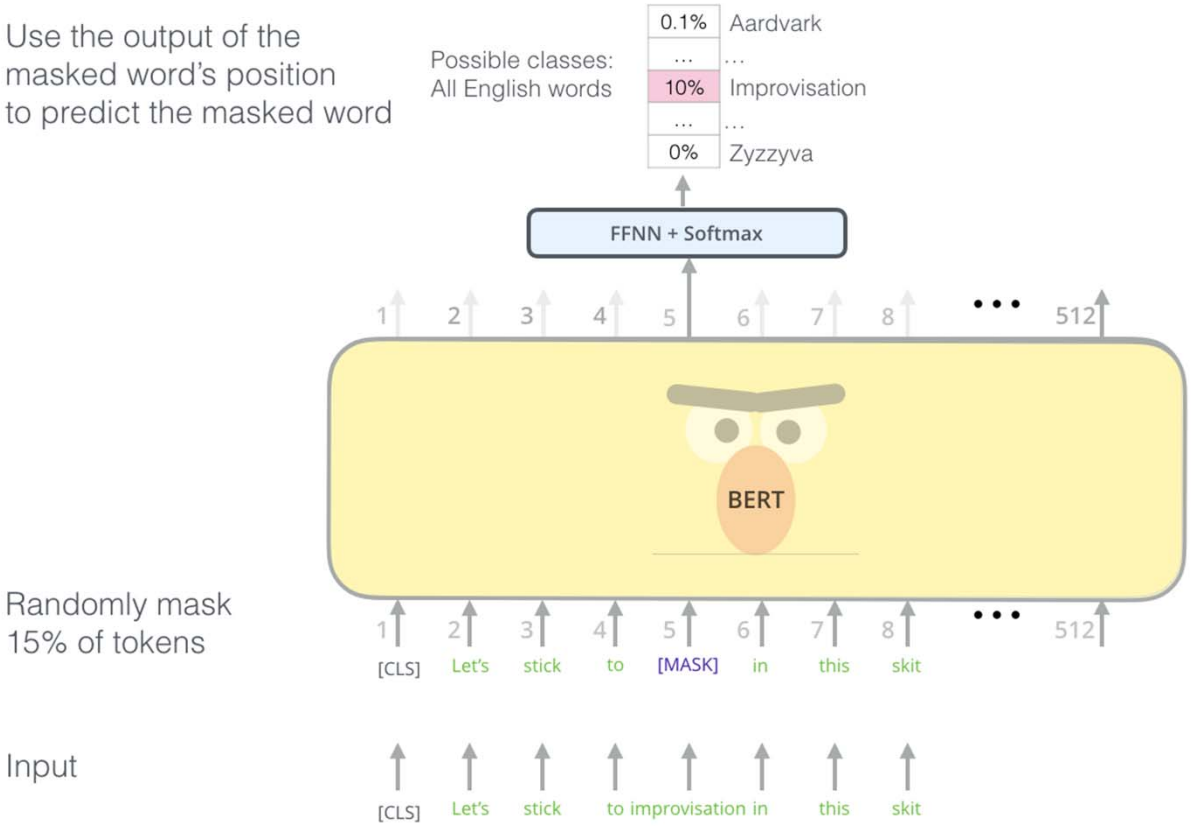
BERT: From Decoders to Encoders

The openAI transformer gave us a fine-tunable pre-trained model based on the Transformer. But something went missing in this transition from LSTMs to Transformers. ELMo's language model was bi-directional, but the openAI transformer only trains a forward language model. Could we build a transformer-based model whose language model looks both forward and backwards (in the technical jargon – "is conditioned on both left and right context")?

Masked Language Model

“We’ll use transformer encoders”, said BERT.
“This is madness”, replied Ernie, “Everybody knows bidirectional conditioning would allow each word to indirectly see itself in a multi-layered context.”
“We’ll use masks”, said BERT confidently.

BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word.



Masked Language Model

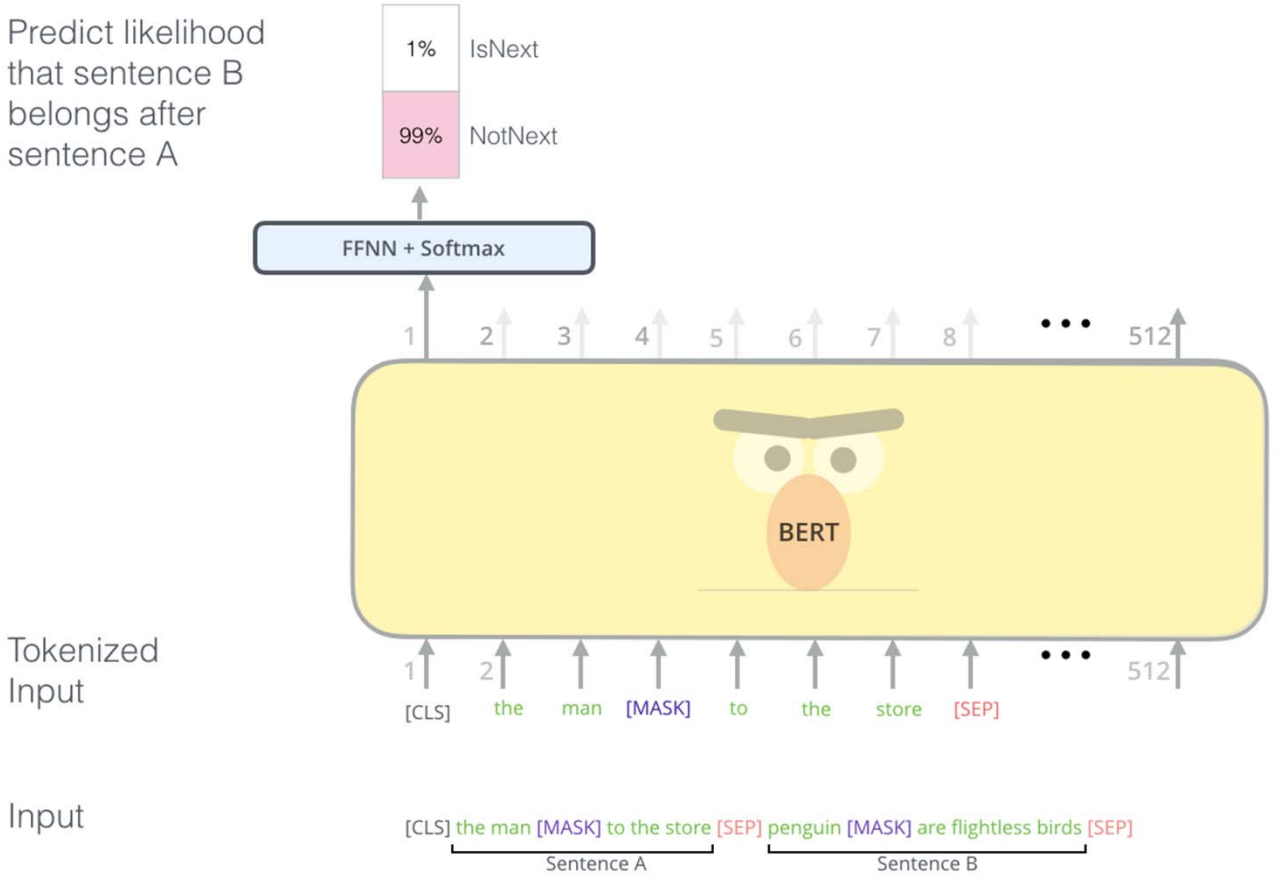
Finding the right task to train a Transformer stack of encoders is a complex hurdle that BERT resolves by adopting a “masked language model” concept from earlier literature (where it’s called a Cloze task).

Beyond masking 15% of the input, BERT also mixes things a bit in order to improve how the model later fine-tunes. Sometimes it randomly replaces a word with another word and asks the model to predict the correct word in that position.

Two-sentence Tasks

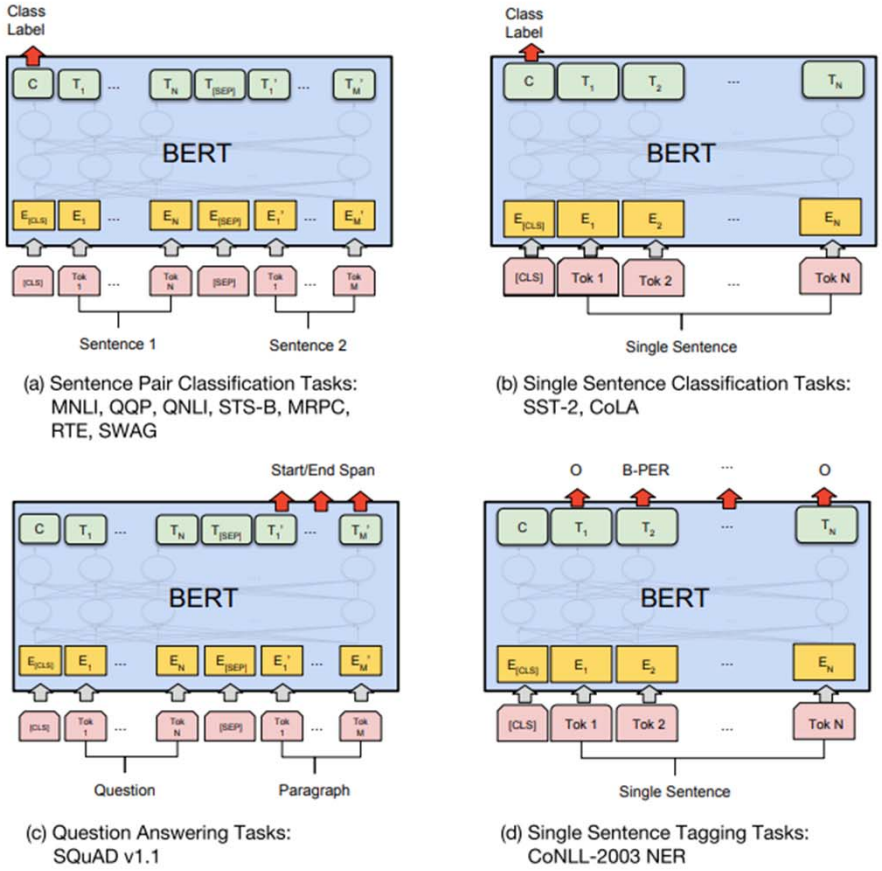
If you look back up at the input transformations the OpenAI transformer does to handle different tasks, you'll notice that some tasks require the model to say something intelligent about two sentences (e.g. are they simply paraphrased versions of each other? Given a wikipedia entry as input, and a question regarding that entry as another input, can we answer that question?).
To make BERT better at handling relationships between multiple sentences, the pre-training process includes an additional task: Given two sentences (A and B), is B likely to be the sentence that follows A, or not?

The second task BERT is pre-trained on is a two-sentence classification task. The tokenization is oversimplified in this graphic as BERT actually uses WordPieces as tokens rather than words --- so some words are broken down into smaller chunks.



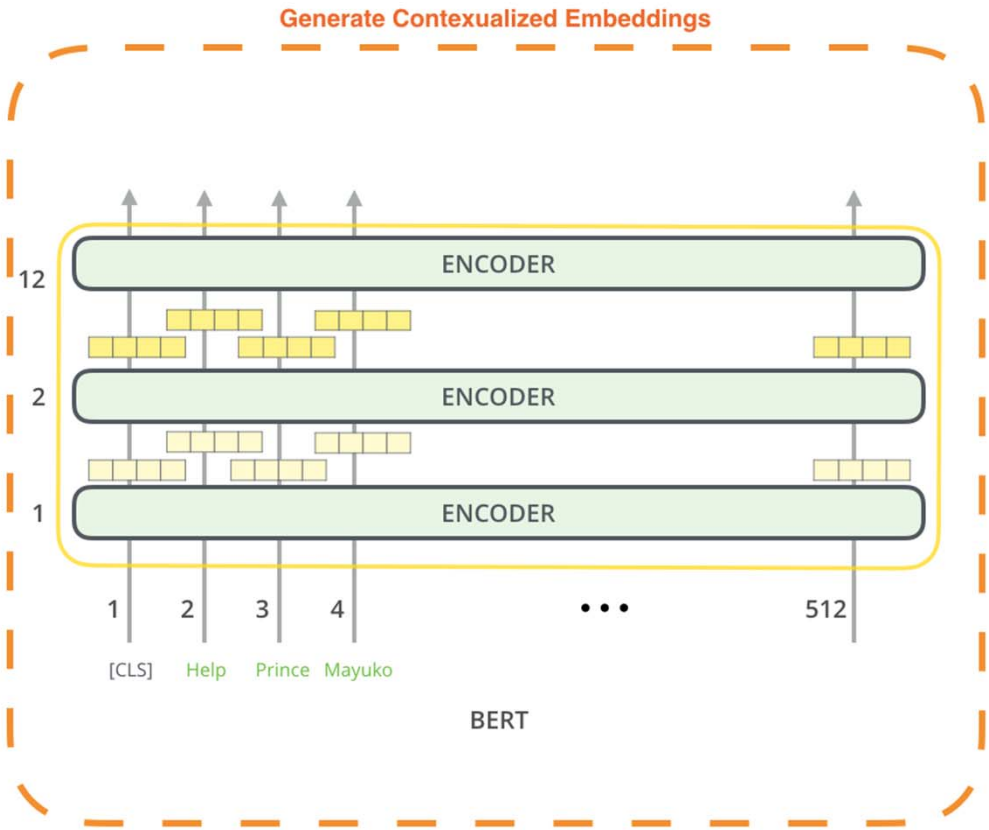
Task specific-Models

The BERT paper shows a number of ways to use BERT for different tasks.

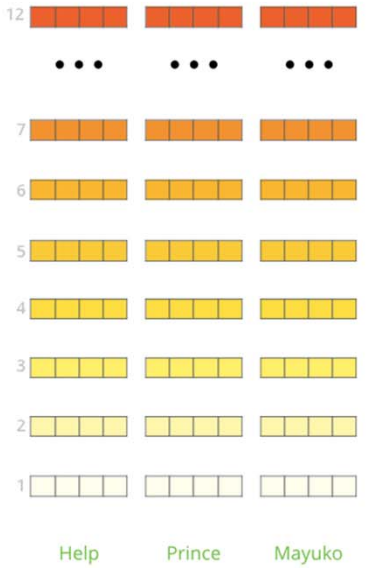


BERT for feature extraction

The fine-tuning approach isn't the only way to use BERT. Just like ELMo, you can use the pre-trained BERT to create contextualized word embeddings. Then you can feed these embeddings to your existing model – a process the paper shows yield results not far behind fine-tuning BERT on a task such as named-entity recognition.



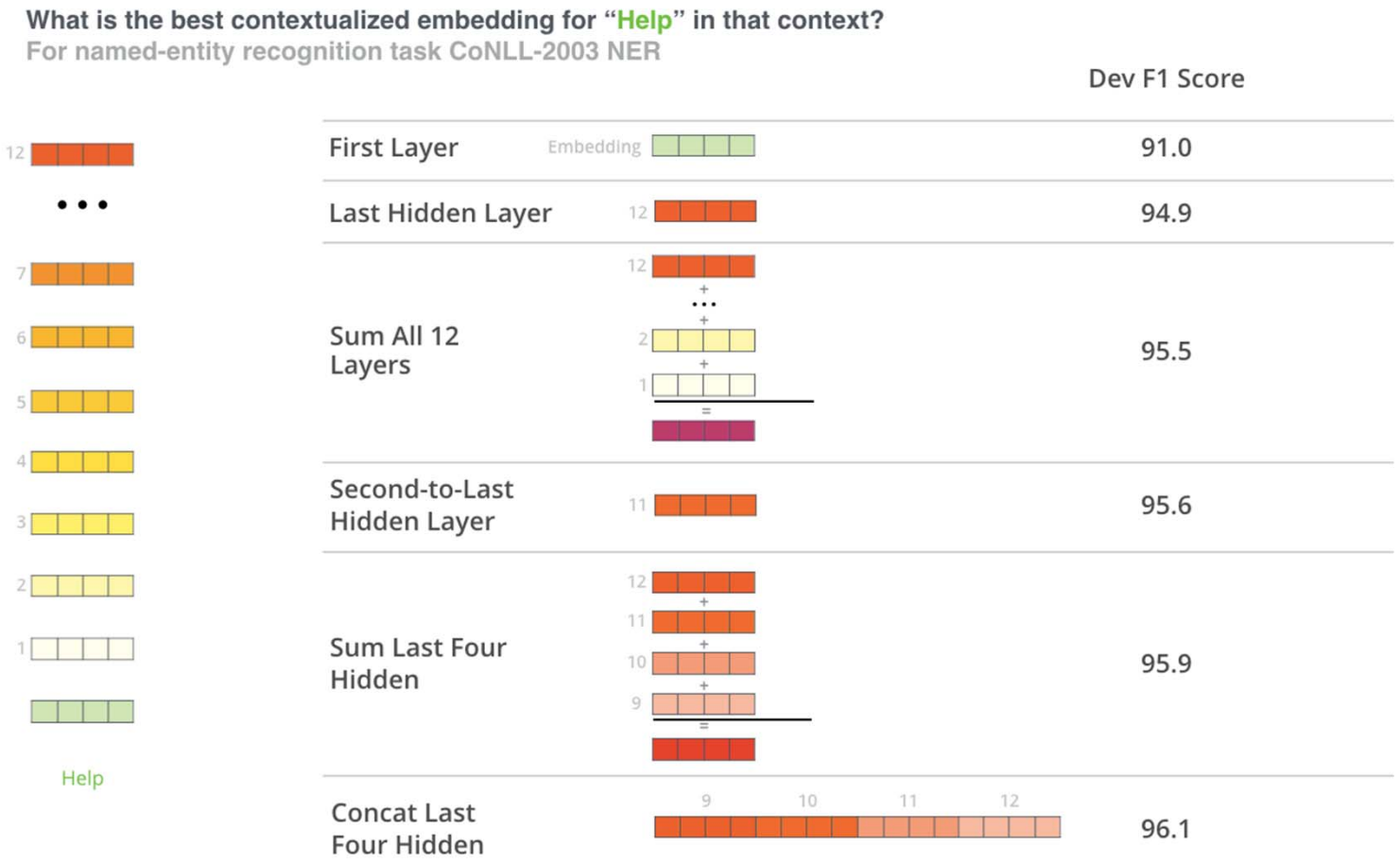
The output of each encoder layer along each token's path can be used as a feature representing that token.



But which one should we use?

Model Architecture

Which vector works best as a contextualized embedding? I would think it depends on the task. The paper examines six choices (Compared to the fine-tuned model which achieved a score of 96.4):



Take BERT out for a spin

Look at the code in the BERT repo:

- The model is constructed in modeling.py (class BertModel) and is pretty much identical to a vanilla Transformer encoder.
- run_classifier.py is an example of the fine-tuning process. It also constructs the classification layer for the supervised model. If you want to construct your own classifier, check out the create_model() method in that file.
- Several pre-trained models are available for download. These span BERT Base and BERT Large, as well as languages such as English, Chinese, and a multi-lingual model covering 102 languages trained on wikipedia.
- BERT doesn't look at words as tokens. Rather, it looks at WordPieces. tokenization.py is the tokenizer that would turns your words into wordPieces appropriate for BERT.

You can also check out the PyTorch implementation of BERT. The AllenNLP library uses this implementation to allow using BERT embeddings with any model.

问题及讨论

Q&A