Neural Language Models and BERT

CS 485, Fall 2023
Applications of Natural Language Processing
https://people.cs.umass.edu/~brenocon/cs485_f23/

Brendan O'Connor
College of Information and Computer Sciences
University of Massachusetts Amherst

including slides from Mohit Iyyer and Emma Strubell
Language modeling as representation learning

- Train a skip-gram LM ->
  get useful \textit{word embeddings}

- Today: train a (funny) LM ->
  get useful \textit{token embeddings}
Why contextual embeddings?
BERT

- “Bidirectional… Transformers”
  - Transformer: a specific neural net architecture for token sequences, that uses attention and token embeddings
  - Bidirectional: The core model is a masked LM, predicting missing word(s) from rest of words in sentence

- Usage
  - 1. "Pretrain": train it as a masked language model on a large corpus
    - It learns to infer useful contextual word embeddings per token
  - 2. "Fine-tune": apply it for your desired supervised learning task.
    (Further update the parameters to do well at your task.)

- BERT (+ variants) are incredibly successful
  - ... and it learns useful linguistic structure by itself!
    Rogers et al., 2020
  - ... and there are easy to use implementations:
    https://huggingface.co/docs/transformers/index
Transformers: Self-attention
The self-supervision task used to train BERT is the masked language-modeling or cloze task, where one is given a text in which some of the original words have been replaced with a special mask symbol. The goal is to predict, for each masked word, the identities of the masked-out input words.

Fig. 3. A high-level illustration of BERT. Words in the input sequence are randomly masked out and then all words are embedded as vectors in \( \mathbb{R}^d \). A Transformer network applies multiple layers of multiheaded attention to the representations. The final representations are used to predict the identities of the masked-out input words.
Attention for Masked LM
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

The two most commonly used attention functions are additive attention \[2\], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ \[3\]. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension \( d_k \), and values of dimension \( d_v \). We compute the dot products of the query with all keys, divide each by \( \sqrt{d_k} \), and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix \( Q \). The keys and values are also packed together into matrices \( K \) and \( V \). We compute the matrix of outputs as:

\[
\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V
\]

The two most commonly used attention functions are additive attention \(^2\) and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of \( \frac{1}{\sqrt{d_k}} \). Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of \( d_k \) the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of \( d_k \).\(^3\) We suspect that for large values of \( d_k \), the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.\(^4\) To counteract this effect, we scale the dot products by \( \frac{1}{\sqrt{d_k}} \).

3.2.2 Multi-Head Attention

Instead of performing a single attention function with \( d_{\text{model}} \)-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values \( h \) times with different, learned linear projections to \( d_k \), \( d_k \) and \( d_v \) dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding \( d_v \)-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

\(^4\) To illustrate why the dot products get large, assume that the components of \( q \) and \( k \) are independent random variables with mean 0 and variance 1. Then their dot product, \( q \cdot k = \sum_{i=1}^{d_k} q_i k_i \), has mean 0 and variance \( d_k \).
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

The two most commonly used attention functions are additive attention \[^2\] and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ \[^3\]. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients \[^4\]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{model}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

\[^4\] To illustrate why the dot products get large, assume that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance $d_k$.
Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

### 3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $p_{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

The two most commonly used attention functions are additive attention \[2\], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{p_{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ \[3\]. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients \[4\]. To counteract this effect, we scale the dot products by $\frac{1}{p_{d_k}}$.

### 3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

\[4\] To illustrate why the dot products get large, assume that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance $d_k$. (Vaswani et al. 2017, original notation, slide: Emma Strubell)
3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\frac{1}{\sqrt{d_k}}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$, and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

4 To illustrate why the dot products get large, assume that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance $d_k$. [Vaswani et al. 2017 original notation, slide: Emma Strubell]
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\frac{1}{\sqrt{d_k}}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$
\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V
$$

The two most commonly used attention functions are additive attention, and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $p d_k$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{p d_k}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{p d_k}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

4 [Vaswani et al. 2017 original notation, slide: Emma Strubell]
Multi-head self-attention

[Vaswani et al. 2017 original notation]
Multi-head self-attention

[Vaswani et al. 2017 original notation]
Multi-head self-attention

[Original notation by Vaswani et al. 2017]
Multi-head self-attention

Layer 1

Layer p

Layer J

Multi-head self-attention + feed forward

Nobel committee awards Strickland who advanced optics

[Vaswani et al. 2017 original notation]
Multi-head self-attention

[Layer 1]

[Layer p]

[Layer J]

Multi-head self-attention + feed forward

Nobel committee awards Strickland who advanced optics

[Vaswani et al. 2017 original notation]
These are residual connections
Positional encoding

Positional Encoding

EMBEDDING WITH TIME SIGNAL

POSITIONAL ENCODING

EMBEDDINGS

INPUT

Je

suis

étudiant
Hacks to get it to work:
Optimizer

We used the Adam optimizer (cite) with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula: \( lrate = \alpha_{\text{model}} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \) This corresponds to increasing the learning rate linearly for the first \( warmup\_steps \) training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used \( warmup\_steps = 4000 \).

Note: This part is very important. Need to train with this setup of the model.
Label Smoothing

During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ (cite). This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

We implement label smoothing using the KL div loss. Instead of using a one-hot target distribution, we create a distribution that has confidence of the correct word and the rest of the smoothing mass distributed throughout the vocabulary.

I went to class and took ___

cats TV notes took sofa

0 0 1 0 0 0

0.025 0.025 0.9 0.025 0.025 0.025

with label smoothing
Byte pair encoding (BPE)

- Deal with rare words / large vocabulary by using subword tokenization
  - Initial analysis step iteratively merges frequent character n-grams to form the vocabulary
  - Confusing name comes from data compression literature - not actually about bytes for us

<table>
<thead>
<tr>
<th>system</th>
<th>sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>health research institutes</td>
</tr>
<tr>
<td>reference</td>
<td>Gesundheitsforschungsinstitute</td>
</tr>
<tr>
<td>WDict</td>
<td>Forschungsinstitute</td>
</tr>
<tr>
<td>C2-50k</td>
<td>Forschungsinst</td>
</tr>
<tr>
<td>BPE-60k</td>
<td>Gesundheitsforschungsinsti</td>
</tr>
<tr>
<td>BPE-J90k</td>
<td>Gesundheitsforschungsinstitude</td>
</tr>
<tr>
<td>source</td>
<td>asinine situation</td>
</tr>
<tr>
<td>reference</td>
<td>dumme Situation</td>
</tr>
<tr>
<td>WDict</td>
<td>asinine situation $\rightarrow$ UNK $\rightarrow$ asinine</td>
</tr>
<tr>
<td>C2-50k</td>
<td>as</td>
</tr>
<tr>
<td>BPE-60k</td>
<td>as</td>
</tr>
<tr>
<td>BPE-J90K</td>
<td>as</td>
</tr>
</tbody>
</table>

Sennrich et al., ACL 2016
What does BERT learn?

3.2 Semantic Knowledge

To date, more studies have been devoted to BERT's knowledge of syntactic rather than semantic phenomena. However, we do have evidence from an MLM probing study that BERT has some knowledge of semantic roles (Ettinger, 2019). BERT even displays some preference for the incorrect fillers for semantic roles that are semantically related to the correct ones, as opposed to those that are unrelated (e.g., "to tip a chef" is better than "to tip a robin", but worse than "to tip a waiter").

Tenney et al. (2019b) showed that BERT encodes information about entity types, relations, semantic roles, and proto-roles, since this information can be detected with probing classifiers. BERT struggles with representations of numbers. Addition and number decoding tasks showed that BERT does not form good representations for floating point numbers and fails to generalize away from the training data (Wallace et al., 2019b). A part of the problem is BERT's wordpiece tokenization, since numbers of similar values can be divided up into substantially different word chunks.

Out-of-the-box BERT is surprisingly brittle to named entity replacements: For example, replacing names in the coreference task changes 85% of predictions (Balasubramanian et al., 2020). This suggests that the model does not actually form a generic idea of named entities, although its F1 scores on NER probing tasks are high (Tenney et al., 2019a). Broscheit (2019) finds that fine-tuning BERT on Wikipedia entity linking "teaches" it additional entity knowledge, which would suggest that it did not absorb all the relevant entity information during pre-training on Wikipedia.

Figure 2: BERT world knowledge (Petroni et al., 2019).

3.3 World Knowledge

The bulk of evidence about commonsense knowledge captured in BERT comes from practitioners using it to extract such knowledge. One direct probing study of BERT reports that BERT struggles with pragmatic inference and role-based event knowledge (Ettinger, 2019). BERT also struggles with abstract attributes of objects, as well as visual and perceptual properties that are likely to be assumed rather than mentioned (Da and Kasai, 2019).

The MLM component of BERT is easy to adapt for knowledge induction by filling in the blanks (e.g., "Cats like to chase [MASK]"). Petroni et al. (2019) showed that, for some relation types, vanilla BERT is competitive with methods relying on knowledge bases (Figure 2), and Roberts et al. (2020) show the same for open-domain QA using the T5 model (Raffel et al., 2019). Davison et al. (2019) suggest that it generalizes better to unseen data. In order to retrieve BERT's knowledge, we need good template sentences, and there is work on their automatic extraction and augmentation (Bouraoui et al., 2019; Jiang et al., 2019b).

However, BERT cannot reason based on its world knowledge. Forbes et al. (2019) show that BERT can "guess" the affordances and properties of many objects, but cannot reason about the relationship between properties and affordances. For example, it "knows" that people can walk into houses, and that houses are big, but it cannot infer that houses are bigger than people. Zhou et al. (2020) and Richardson and Sabharwal (2019) also show that the performance drops with the number of necessary inference steps. Some of BERT's world knowledge success comes from learning stereotypical associations (Poerner et al., 2019), for example, a person with an Italian-sounding name is predicted to be Italian, even when it is incorrect.

[Roberts et al., 2020]
What does BERT learn?

3. What Knowledge Does BERT Have?

Anumber of studies have looked at the knowledge encoded in BERT weights. The popular approaches include fill-in-the-gap probes of MLM, analysis of self-attention weights, and probing classifiers with different BERT representations as inputs.

3.1 Syntactic Knowledge

Lin et al. (2019) showed that BERT representations are hierarchical rather than linear, that is, there is something akin to syntactic tree structure in addition to the word order information. Tenney et al. (2019b) and Liu et al. (2019a) also showed that BERT embeddings encode information about parts of speech, syntactic chunks, and roles. Enough syntactic information seems to be captured in the token embeddings themselves to recover syntactic trees (Vilares et al., 2020; Kim et al., 2020; Rosa and Mareš, 2019), although probing classifiers could not recover the labels of distant parent nodes in the syntactic tree (Liu et al., 2019a). Warstadt and Bowman (2020) report evidence of hierarchical structure in three out of four probing tasks.

As far as how syntax is represented, it seems that syntactic structure is not directly encoded in self-attention weights. Htut et al. (2019) were unable to extract full parse trees from BERT heads even with the gold annotations for the root. Jawahar et al. (2019) include a brief illustration of a dependency tree extracted directly from self-attention weights, but provide no quantitative evaluation. However, syntactic information can be recovered from BERT token representations. Hewitt and Manning (2019) were able to learn transformation matrices that successfully recovered syntactic dependencies in PennTreebank data from BERT's token embeddings (see also Manning et al., 2020). Jawahar et al. (2019) experimented with transformations of the [CLS] token using Tensor Product Decomposition Networks (McCoy et al., 2019a), concluding that dependency trees are the best match among five decomposition schemes (although the reported MSE differences are very small). Miaschi and Dell'Orletta (2020) perform a range of syntactic probing experiments with concatenated token representations as input.

Note that all these approaches look for the evidence of gold-standard linguistic structures, and add some amount of extra knowledge to the probe. Most recently, Wu et al. (2020) proposed a parameter-free approach based on measuring the impact that one word has on predicting another word within a sequence in the MLM task (Figure 1). They concluded that BERT “naturally” learns some syntactic information, although it is not very similar to linguistic annotated resources.

The fill-in-the-gap probes of MLM showed that BERT takes subject-predicate agreement into account when performing the cloze task (Goldberg, 2019; van Schijndel et al., 2019), even for meaningless sentences and sentences with distractor clauses between the subject and the verb (Goldberg, 2019). A study of negative polarity items (NPIs) by Warstadt et al. (2019) showed that BERT is better able to detect the presence of NPIs (e.g., “ever”) and the words that allow their use (e.g., “whether”) than scope violations.

The above claims of syntactic knowledge are belied by the evidence that BERT does not “understand” negation and is insensitive to malformed input. In particular, its predictions were not altered even with shuffled word order.

---

Figure 1: Parameter-free probe for syntactic knowledge: words sharing syntactic subtrees have larger impact on each other in the MLM prediction (Wu et al., 2020).

---

[Rogers et al., 2020]
What does BERT learn?

**Fig. 6.** Some BERT attention heads that appear sensitive to linguistic phenomena, despite not being explicitly trained on linguistic annotations. In the example attention maps, the darkness of a line indicates the size of the attention weight. All attention to/from red words is colored red; these words are chosen to highlight certain of the attention heads’ behaviors. [CLS] (classification) and [SEP] (separator) are special tokens BERT adds to the input during preprocessing. Attention heads are numbered by their layer and index in BERT. Reprinted with permission from ref. 59, which is licensed under CC BY 4.0.
Using BERT

• You get
  • Per-token embeddings
  • Multiple layers at each
  • Embedding for per-sentence “[CLS]” symbol
• Use as input for tasks. Two learning approaches
  • “Frozen”: use them as input features
  • Fine-tuning: backprop through the actual BERT model itself (better)
Using BERT for classif.
• Many pretrained BERT or BERT-like models are available (especially for English and other high-resource languages…)

• Check out HuggingFaces’ examples

  • https://huggingface.co/transformers/examples.html