Contextual word representations

Christopher Potts

Stanford Linguistics

CS 224U: Natural language understanding
May 11
Overview

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Associated materials

- Notebook: contextualreps.ipynb
- Smith 2019
- ELMo: Peters et al. 2018; [project site]
- Transformer
  1. Vaswani et al. 2017
  3. Hugging Face transformers: project site
     a. BERT: Devlin et al. 2019; project site
     b. RoBERTa: Liu et al. 2019; project site
     c. ELECTRA: Clark et al. 2019; project site
     d. XLNet: Yang et al. 2019; project site
Word representations and context

1. a. The vase broke.
   b. Dawn broke.
   c. The news broke.
   d. Sandy broke the world record.
   e. Sandy broke the law.
   f. The burgler broke into the house.
   g. The newscaster broke into the movie broadcast.
   h. We broke even.

2. a. flat tire/beer/note/surface
   b. throw a party/fight/ball/fit

3. a. A crane caught a fish.
   b. A crane picked up the steel beam.
   c. I saw a crane.

4. a. Are there typos? I didn’t see any.
   b. Are there bookstores downtown? I didn’t see any.
Model structure and linguistic structure

The Rock rules x 47 x 30 x 34

h_1 h_2 h_3

+ x h_r

x_{47} x_{30} x_{34}

The Rock rules

The rock rules

h_1 h_2 h_3

x_{47} x_{30} x_{34}

The rock rules

The Rock rules

h_1 h_2 h_3

• •• attention
Guiding idea: Attention (from the NLI slides)

classifier \[ y = \text{softmax}(\tilde{h}W + b) \]

attention combo \[ \tilde{h} = \tanh([\kappa; h_C]W_\kappa) \]

context \[ \kappa = \text{mean}(\alpha_1 h_1, \alpha_2 h_2, \alpha_3 h_3) \]

attention weights \[ \alpha = \text{softmax}(\tilde{\alpha}) \]

scores \[ \tilde{\alpha} = \begin{bmatrix} h_C^T h_1 & h_C^T h_2 & h_C^T h_3 \end{bmatrix} \]
Guiding idea: Subword modeling

Max-pooling layers concatenated to form the word representation

Filters of different length, obtained via dense layers processing the input character embeddings and combined via max-pooling:

<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
<th>6</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>
Guiding idea: Word piece tokenization

```python
[1]: from transformers import BertTokenizer
[2]: tokenizer = BertTokenizer.from_pretrained('bert-base-cased')
[3]: tokenizer.tokenize("This isn't too surprising.")
[3]: ['This', 'isn', '''', 't', 'too', 'surprising', '']
[4]: tokenizer.tokenize("Encode me!")
[4]: ['En', '##code', 'me', '!']
[5]: tokenizer.tokenize("Snuffleupagus?")
[5]: ['S', '##nu', '##ffle', '##up', '##agu', '##s', '?']
[6]: tokenizer.vocab_size
[6]: 28996

Sennrich et al. 2016,
https://github.com/google/sentencepiece
Guiding idea: Positional encoding

From ‘The Annotated Transformer’
Current issues and efforts

Figure 1: The computational demands of modern deep learning methods for NLP, measured in Floating Point Operations (FLOPs).

- GloVE: 1 exaflop
- GPT: 190 exaflops
- BERT-Base: 60
- BERT-Large: 90
- RoBERTa: 3 zettaflops
- XLNet: 1 exaflop

Clark et al. 2019
Current issues and efforts

Who said that training GPT-2 or BERT was expensive?

"We use 512 Nvidia V100 GPUs [...] Upon the submission of this paper, training has lasted for three months [...] and perplexity on the development set is still dropping."

https://twitter.com/artetxem/status/1178794889229864962
Current issues and efforts

<table>
<thead>
<tr>
<th>Consumption</th>
<th>CO₂e (lbs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air travel, 1 person, NY↔SF</td>
<td>1984</td>
</tr>
<tr>
<td>Human life, avg, 1 year</td>
<td>11,023</td>
</tr>
<tr>
<td>American life, avg, 1 year</td>
<td>36,156</td>
</tr>
<tr>
<td>Car, avg incl. fuel, 1 lifetime</td>
<td>126,000</td>
</tr>
</tbody>
</table>

**Table 1:** Estimated CO₂ emissions from training common NLP models, compared to familiar consumption.¹

¹Sources: (1) Air travel and per-capita consumption: https://bit.ly/2Hw0xWc; (2) car lifetime: https://bit.ly/2Qbr0w1.
Current issues and efforts

Transformers

All Models and checkpoints

Also check out our list of Community contributors 🖤 and Organizations 🌐.

Search models...

Filter by model tags

- All
- PyTorch
- TensorFlow
- French 🇫🇷
- German 🇩🇪
- Dutch 🇳🇱
- Italian 🇮🇹
- Spanish 🇪🇸
- Swedish 🇸🇪
- Finnish 🇫🇮
- Greek 🇬🇷
- Turkish 🇹🇷
- Arabic ﷼
- Chinese 🇨🇳
- Malay 🇲🇾
- Polish 🇵🇱
- Esperanto
- Multilingual 🌍

https://huggingface.co
Current issues and efforts

Compressing Large-Scale Transformer-Based Models: A Case Study on BERT

Prakhar Ganesh\(^1\), Yao Chen\(^1\), Xin Lou\(^1\), Mohammad Ali Khan\(^1\), Yin Yang\(^2\), Deming Chen\(^3\), Marianne Winslett\(^3\), Hassan Sajjad\(^4,2\) and Preslav Nakov\(^4,2\)

\(^1\)Advanced Digital Sciences Center
\(^2\)Hamad Bin Khalifa University
\(^3\)University of Illinois at Urbana-Champaign
\(^4\)Qatar Computing Research Institute

\{prakhar.g, yao.chen, lou.xin, mohammad.k\}@adsc-create.edu.sg,
\{yyang, hsajjad, pnakov\}@hbku.edu.qa, \{dchen, winslett\}@illinois.edu

Mitchell A. Gordon

All The Ways You Can Compress BERT

Nov 18, 2019

Model compression reduces redundancy in a trained neural network. This is useful, since BERT barely fits on a GPU (BERT-Large does not) and definitely won’t fit on your smart phone. Improved memory and inference speed efficiency can also save costs at scale.

http://mitchgordon.me/
ELMo

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Core model structure

\[ \text{rules} = s_{\text{task}.0} \cdot \text{rules}_0 + s_{\text{task}.1} \cdot \text{rules}_{4,1} + s_{\text{task}.2} \cdot \text{rules}_{4,2} \]
Word embeddings

A final linear projection into the embedding dimensionality, which must be twice the RNN hidden dimensionality.

Highway layers introduce gating information between layers.

A series of convolutional filters with max pooling, concatenated to form the initial representation.
ELMo model releases

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
<th>Hidden size</th>
<th>Output size</th>
<th>Highway layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>13.6M</td>
<td>1024</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>Medium</td>
<td>28.0M</td>
<td>2048</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>Original</td>
<td>93.6M</td>
<td>4096</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>Original (5.5B)</td>
<td>93.6M</td>
<td>4096</td>
<td>512</td>
<td>2</td>
</tr>
</tbody>
</table>

Additional details at [https://allennlp.org/elmo](https://allennlp.org/elmo); the options files reveal additional information about the subword convolutional filters, activation functions, thresholds, and layer dimensions.
Transformers

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Core model structure

The core model structure is represented by a diagram showing the flow of data through different layers. The model consists of input layers, attention layers, norm layers, and output layers.

### Core Model Equations

- **Core Model Structure Diagram**
  - The model takes inputs from various sources and processes them through layers.
  - The outputs are derived through a series of operations involving attention, normalization, and dropout.

- **Mathematical Equations**
  - **Core Output Equation**: \( C_{out} = \frac{c_{fflayer} - \text{mean}(c_{fflayer})}{\text{std}(c_{fflayer}) + \epsilon} \)
  - **Core Feedback Equation**: \( C_{ff} = \text{ReLU}(c_{anorm}W_1 + b_1)W_2 + b_2 \)
  - **Core Normalization Equation**: \( C_{anorm} = \frac{c_{alayer} - \text{mean}(c_{alayer})}{\text{std}(c_{alayer}) + \epsilon} \)
  - **Core Attention Equation**: \( c_{attn} = \text{sum}([\alpha_1 a_{input}, \alpha_2 b_{input}]) \)
    - \( \alpha = \text{softmax}(\tilde{\alpha}) \)
    - \( \tilde{\alpha} = \left[ \frac{c_{input}^T a_{input}}{\sqrt{d_k}}, \frac{c_{input}^T b_{input}}{\sqrt{d_k}} \right] \)
  - **Core Input Equation**: \( C_{input} = x_{34} + p_3 \)
Computing the attention representations

Calculation as previously given

\[ c_{\text{attn}} = \text{sum}(\alpha_1 a_{\text{input}}, \alpha_2 b_{\text{input}}) \]
\[ \alpha = \text{softmax}(\tilde{\alpha}) \]
\[ \tilde{\alpha} = \begin{bmatrix} \frac{c_{\text{input}}^T a_{\text{input}}}{\sqrt{d_k}}, & \frac{c_{\text{input}}^T b_{\text{input}}}{\sqrt{d_k}} \end{bmatrix} \]

Matrix format

\[
\text{softmax} \left( \frac{c_{\text{input}} \begin{bmatrix} a_{\text{input}}^T \\ b_{\text{input}}^T \end{bmatrix}}{\sqrt{d_k}} \right) \begin{bmatrix} a_{\text{input}} \\ b_{\text{input}} \end{bmatrix}
\]
Computing the attention representations

```python
[1]: import numpy as np

[2]: seq_length = 3
d_k = 4

[3]: inputs = np.random.uniform(size=(seq_length, d_k))
inputs

[3]: array([[0.31436922, 0.66969307, 0.270804 , 0.72023504],
          [0.87180132, 0.27637445, 0.43091867, 0.34138704],
          [0.20292054, 0.6345131 , 0.01058343, 0.22846636]])

[4]: a_input = inputs[0]
b_input = inputs[1]
c_input = inputs[2]
```
Computing the attention representations

```python
[5]: def softmax(X):
    z = np.exp(X)
    return (z / z.sum(axis=0)).T

[6]: c_alpha = softmax([
    (c_input.dot(a_input) / np.sqrt(d_k)),
    (c_input.dot(b_input) / np.sqrt(d_k))])

[7]: c_attn = sum([c_alpha[0]*a_input, c_alpha[1]*b_input])
c_attn

[7]: array([[0.57768027, 0.48390338, 0.34643646, 0.54128076]])

[8]: ab = inputs[:-1]

[9]: softmax(c_input.dot(ab.T) / np.sqrt(d_k)).dot(ab)

[9]: array([[0.57768027, 0.48390338, 0.34643646, 0.54128076]])

[10]: # If we allow every input to attend to itself:
    softmax(inputs.dot(inputs.T) / np.sqrt(d_k)).dot(inputs)

[10]: array([[0.4614388 , 0.53204444, 0.2451212 , 0.45136127],
    [0.50173123, 0.50618272, 0.26184404, 0.43678288],
    [0.45493467, 0.5332328 , 0.23643403, 0.4388242 ]])
```
Multi-headed attention

\[
c_{\text{attn}}^3 = \text{sum}\left( \left[ \alpha_1(a_{\text{input}} W_3^V), \alpha_2(b_{\text{input}} W_3^V) \right] \right)
\]

\[
\alpha = \text{softmax}(\tilde{\alpha})
\]

\[
\tilde{\alpha} = \frac{(c_{\text{input}} W_3^Q)^T (a_{\text{input}} W_3^K)}{\sqrt{d_k}}, \quad \frac{(c_{\text{input}} W_3^Q)^T (b_{\text{input}} W_3^K)}{\sqrt{d_k}}
\]
Repeated transformer blocks

Repeated 6 times with $c_{out}$ serving as $c_{input}$ at each successive layer.

Includes multi-headed attention in each block.
The architecture diagram

Each decoder state self-attends with all of its fellow decoder states and with all the encoder states.

The left side is repeated for every state in the encoder.

The right side is repeated for every decoder state, with outputs for each state that has them (all of them for dialogue and machine translation, only the final one for NLI).

In the decoder, self-attention is limited to preceding words.

Figure 1: The Transformer - model architecture.
BERT

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Core model structure

Transformer blocks

[CLS] in a in b in c in S in

x47 p0 $s_A$

x47 p1 $s_A$

x30 p2 $s_A$

x34 p3 $s_A$

x1 p4 $s_A$

[CLS] 0 SentA The 1 SentA Rock 2 SentA rules 3 SentA [SEG] 4 SentA

CLS out a out b out c out S out
Masked Language Modeling (MLM)

Transformer blocks

masking: none
Masked Language Modeling (MLM)
Masked Language Modeling (MLM)

Transformer blocks

masking: random word
MLM loss function

For Transformer parameters $H_{\theta}$ and sequence $\mathbf{x} = [x_1, \ldots, x_T]$ with masked version $\hat{\mathbf{x}}$:

$$\max_{\theta} \sum_{t=1}^{T} m_t \log \frac{\exp \left( (e(x_t))^T H_{\theta}(\hat{\mathbf{x}})_t \right)}{\sum_{x' \in \mathcal{V}} \exp \left( (e(x'))^T H_{\theta}(\hat{\mathbf{x}})_t \right)}$$

where $\mathcal{V}$ is the vocabulary, $x_t$ is the actual token at step $t$, $m_t = 1$ if token $t$ was masked, else 0, and $e(x)$ is the embedding for $x$. 
Binary sentence prediction pretraining

Positive: Actual sentence sequences

- [CLS] the man went to [MASK] store [SEP]
- he bought a gallon [MASK] milk [SEP]
- Label: IsNext

Negative: Randomly chosen second sentence

- [CLS] the man went to [MASK] store [SEP]
- penguin [MASK] are flight # # less birds [SEP]
- Label: NotNext
Transfer learning and fine-tuning

Transformer blocks

your task labels

your task params

CLSout

aout

bout

cout

Sout

CLSin

alin

bin

cin

Sin

x47

p0

SA

x47

p1

SA

x30

p2

SA

x34

p3

SA

x1

p4

SA

[CLS] 0  SentA

The  1  SentA

Rock  2  SentA

rules  3  SentA

[SEG] 4  SentA

your task labels

The

rules

SentA

sentA

SentA

SentA

x47

p0

x47

p1

x30

p2

x34

p3

x1

p4
Tokenization and the BERT embedding space

```python
[1]: from transformers import BertTokenizer

[2]: tokenizer = BertTokenizer.from_pretrained('bert-base-cased')

[3]: tokenizer.tokenize("This isn't too surprising.")

[3]: ['This', 'isn', '', 't', 'too', 'surprising', ']

[4]: tokenizer.tokenize("Encode me!")

[4]: ['En', '#code', 'me', '!']

[5]: tokenizer.tokenize("Snuffleupagus?")

[5]: ['S', '#nu', '#ffle', '#up', '#agu', '#s', '?']

[6]: tokenizer.vocab_size

[6]: 28996
```
Initial BERT model releases

**Base**
- Transformer layers: 12
- Hidden representations: 768 dimensions
- Attention heads: 12
- Total parameters: 110M

**Large**
- Transformer layers: 24
- Hidden representations: 1024 dimensions
- Attention heads: 16
- Total parameters: 340M

Limited to sequences of 512 tokens due to dimensionality of the positional embeddings.

Many new releases at the [project site](#) and on [Hugging Face](#).
Efforts to make BERT smaller
Efforts to make BERT smaller

All The Ways You Can Compress BERT

Nov 18, 2019

Model compression reduces redundancy in a trained neural network. This is useful, since BERT barely fits on a GPU (BERT-Large does not) and definitely won’t fit on your smart phone. Improved memory and inference speed efficiency can also save costs at scale.
Efforts to make BERT smaller

All The Ways You Can Compress BERT

Nov 18, 2019

Model compression reduces redundancy in a trained neural network. This is useful, since BERT barely fits on a GPU (BERT-Large does not) and definitely won’t fit on your smartphone. Improved memory and inference speed efficiency can also save costs at scale.

Particularly relevant to this lecture:

- Sanh et al. (2019): DistilBERT
- Michel et al. (2019): Fewer attention heads
- Lan et al. (2019): ALBERT
Known limitations with BERT

1. Devlin et al. (2019:§5): admirably detailed but still partial ablation studies and optimization studies.

2. Devlin et al. (2019): “The first [downside] is that we are creating a mismatch between pre-training and fine-tuning, since the [MASK] token is never seen during fine-tuning.”

3. Devlin et al. (2019): “The second downside of using an MLM is that only 15% of tokens are predicted in each batch”

4. Yang et al. (2019): “BERT assumes the predicted tokens are independent of each other given the unmasked tokens, which is oversimplified as high-order, long-range dependency is prevalent in natural language”
RoBERTa

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Addressing the known limitations with BERT

1. Devlin et al. (2019:§5): admirably detailed but still partial ablation studies and optimization studies.

2. Devlin et al. (2019): “The first [downside] is that we are creating a mismatch between pre-training and fine-tuning, since the [MASK] token is never seen during fine-tuning.”

3. Devlin et al. (2019): “The second downside of using an MLM is that only 15% of tokens are predicted in each batch.”

4. Yang et al. (2019): “BERT assumes the predicted tokens are independent of each other given the unmasked tokens, which is oversimplified as high-order, long-range dependency is prevalent in natural language.”
## Robustly optimized BERT approach

<table>
<thead>
<tr>
<th>BERT</th>
<th>RoBERTa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static masking/substitution</td>
<td>Dynamic masking/substitution</td>
</tr>
<tr>
<td>Inputs are two concatenated document segments</td>
<td>Inputs are sentence sequences that may span document boundaries</td>
</tr>
<tr>
<td>Next Sentence Prediction (NSP)</td>
<td>No NSP</td>
</tr>
<tr>
<td>Training batches of 256 examples</td>
<td>Training batches of 2,000 examples</td>
</tr>
<tr>
<td>Word-piece tokenization</td>
<td>Character-level byte-pair encoding</td>
</tr>
<tr>
<td>Pretraining on BooksCorpus and English Wikipedia</td>
<td>Pretraining on BooksCorpus, CC-News, OpenWebText, and Stories</td>
</tr>
<tr>
<td>Train for 1M steps</td>
<td>Train for up to 500K steps</td>
</tr>
<tr>
<td>Train on short sequences first</td>
<td>Train only on full-length sequences</td>
</tr>
</tbody>
</table>

Additional differences in the optimizer and data presentation (sec 3.1).
RoBERTa results informing final system design

<table>
<thead>
<tr>
<th>Masking</th>
<th>SQuAD 2.0</th>
<th>MNLI-m</th>
<th>SST-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>76.3</td>
<td>84.3</td>
<td>92.8</td>
</tr>
<tr>
<td>static</td>
<td>78.3</td>
<td>84.3</td>
<td>92.5</td>
</tr>
<tr>
<td>dynamic</td>
<td>78.7</td>
<td>84.0</td>
<td>92.9</td>
</tr>
</tbody>
</table>

Table 1: Comparison between static and dynamic masking for BERT\textsubscript{BASE}. We report F1 for SQuAD and accuracy for MNLI-m and SST-2. Reported results are medians over 5 random initializations (seeds). Reference results are from Yang et al. (2019).
RoBERTa results informing final system design

<table>
<thead>
<tr>
<th>Model</th>
<th>SQuAD 1.1/2.0</th>
<th>MNLI-m</th>
<th>SST-2</th>
<th>RACE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Our reimplementation (with NSP loss):</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGMENT-PAIR</td>
<td>90.4/78.7</td>
<td>84.0</td>
<td>92.9</td>
<td>64.2</td>
</tr>
<tr>
<td>SENTENCE-PAIR</td>
<td>88.7/76.2</td>
<td>82.9</td>
<td>92.1</td>
<td>63.0</td>
</tr>
<tr>
<td><strong>Our reimplementation (without NSP loss):</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FULL-SENTENCES</td>
<td>90.4/79.1</td>
<td>84.7</td>
<td>92.5</td>
<td>64.8</td>
</tr>
<tr>
<td>DOC-SENTENCES</td>
<td>90.6/79.7</td>
<td>84.7</td>
<td>92.7</td>
<td>65.6</td>
</tr>
<tr>
<td>BERT&lt;sub&gt;BASE&lt;/sub&gt;</td>
<td>88.5/76.3</td>
<td>84.3</td>
<td>92.8</td>
<td>64.3</td>
</tr>
<tr>
<td>XLNet&lt;sub&gt;BASE&lt;/sub&gt; (K = 7)</td>
<td>–/81.3</td>
<td>85.8</td>
<td>92.7</td>
<td>66.1</td>
</tr>
<tr>
<td>XLNet&lt;sub&gt;BASE&lt;/sub&gt; (K = 6)</td>
<td>–/81.0</td>
<td>85.6</td>
<td>93.4</td>
<td>66.7</td>
</tr>
</tbody>
</table>

Table 2: Development set results for base models pretrained over BOOKCORPUS and WIKIPEDIA. All models are trained for 1M steps with a batch size of 256 sequences. We report F1 for SQuAD and accuracy for MNLI-m, SST-2 and RACE. Reported results are medians over five random initializations (seeds). Results for BERT<sub>BASE</sub> and XLNet<sub>BASE</sub> are from Yang et al. (2019).
RoBERTa results informing final system design

<table>
<thead>
<tr>
<th>bsz</th>
<th>steps</th>
<th>lr</th>
<th>ppl</th>
<th>MNLI-m</th>
<th>SST-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1M</td>
<td>1e-4</td>
<td>3.99</td>
<td>84.7</td>
<td>92.7</td>
</tr>
<tr>
<td>2K</td>
<td>125K</td>
<td>7e-4</td>
<td><strong>3.68</strong></td>
<td><strong>85.2</strong></td>
<td><strong>92.9</strong></td>
</tr>
<tr>
<td>8K</td>
<td>31K</td>
<td>1e-3</td>
<td>3.77</td>
<td>84.6</td>
<td>92.8</td>
</tr>
</tbody>
</table>

Table 3: Perplexity on held-out training data (ppl) and development set accuracy for base models trained over BOOKCORPUS and WIKIPEDIA with varying batch sizes (bsz). We tune the learning rate (lr) for each setting. Models make the same number of passes over the data (epochs) and have the same computational cost.
RoBERTa results informing final system design

Table 4: Development set results for RoBERTa as we pretrain over more data (16GB $\rightarrow$ 160GB of text) and pretrain for longer (100K $\rightarrow$ 300K $\rightarrow$ 500K steps). Each row accumulates improvements from the rows above. RoBERTa matches the architecture and training objective of BERT$_{\text{LARGE}}$. Results for BERT$_{\text{LARGE}}$ and XLNet$_{\text{LARGE}}$ are from Devlin et al. (2019) and Yang et al. (2019), respectively. Complete results on all GLUE tasks can be found in the Appendix.
A Primer in BERTology: What we know about how BERT works

Anna Rogers, Olga Kovaleva, Anna Rumshisky
Department of Computer Science, University of Massachusetts Lowell
Lowell, MA 01854
{arothers, okovalev, arum}@cs.uml.edu
ELECTRA

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Addressing the known limitations with BERT

1. Devlin et al. (2019:§5): admirably detailed but still partial ablation studies and optimization studies.

2. Devlin et al. (2019): “The first [downside] is that we are creating a mismatch between pre-training and fine-tuning, since the [MASK] token is never seen during fine-tuning.”

3. Devlin et al. (2019): “The second downside of using an MLM is that only 15% of tokens are predicted in each batch”

4. Yang et al. (2019): “BERT assumes the predicted tokens are independent of each other given the unmasked tokens, which is oversimplified as high-order, long-range dependency is prevalent in natural language”
Core model structure (Clark et al. 2019)

Random sample of \( \approx 15\% \) of tokens masked

Masked tokens replaced proportional to Generator probabilities

Loss: Generator + \( \lambda \) ELECTRA

Generator (typically a small MLM; paper uses the BERT loss)

Discriminator (ELECTRA)

- the \( \rightarrow \) [MASK] \( \rightarrow \) original
- chef \( \rightarrow \) chef \( \rightarrow \) original
- cooked \( \rightarrow \) [MASK] \( \rightarrow \) replaced
- the \( \rightarrow \) the \( \rightarrow \) original
- meal \( \rightarrow \) meal \( \rightarrow \) original

x masked

x corrupt
Generator/Discriminator relationships

Where Generator and Discriminator are the same size, they can share Transformer parameters, and more sharing is better. However, the best results come from having a Generator that is small compared to the Discriminator:

![Diagram showing GLUE scores for different generator/discriminator sizes (number of hidden units).](https://i.imgur.com/39L39.png)

Clark et al. 2019, Figure 3
Efficiency

Figure 3: Left: GLUE scores for different generator/discriminator sizes (number of hidden units). Interestingly, having a generator smaller than the discriminator improves results. Right: Comparison of different training algorithms. As our focus is on efficiency, the x-axis shows FLOPs rather than train steps (e.g., ELECTRA is trained for fewer steps than BERT because it includes the generator).

Smaller Generators

If the generator and discriminator are the same size, training ELECTRA would take around twice as much compute per step as training only with masked language modeling. We suggest using a smaller generator to reduce this factor. Specifically, we make models smaller by decreasing the layer sizes while keeping the other hyperparameters constant. We also explore using an extremely simple "unigram" generator that samples fake tokens according to their frequency in the train corpus. GLUE scores for differently-sized generators and discriminators are shown in the left of Figure 3. All models are trained for 500k steps, which puts the smaller generators at a disadvantage in terms of compute because they require less compute per training step. Nevertheless, we find that models work best with generators 1/4-1/2 the size of the discriminator. We speculate that having too strong of a generator may pose a too-challenging task for the discriminator, preventing it from learning as effectively. In particular, the discriminator may have to use many of its parameters modeling the generator rather than the actual data distribution. Further experiments in this paper use the best generator size found for the given discriminator size.

Training Algorithms

Lastly, we explore other training algorithms for ELECTRA, although these did not end up improving results. The proposed training objective jointly trains the generator and discriminator. We experiment with instead using the following two-stage training procedure:

1. Train only the generator with $L_{MLM}$ for $n$ steps.
2. Initialize the weights of the discriminator with the weights of the generator. Then train the discriminator with $L_{Disc}$ for $n$ steps, keeping the generator’s weights frozen.

Note that the weight initialization in this procedure requires having the same size for the generator and discriminator. We found that without the weight initialization the discriminator would sometimes fail to learn at all beyond the majority class, perhaps because the generator started so far ahead of the discriminator. Joint training on the other hand naturally provides a curriculum for the discriminator where the generator starts off weak but gets better throughout training. We also explored training the generator adversarially as in a GAN, using reinforcement learning to accommodate the discrete operations of sampling from the generator. See Appendix F for details.

Results are shown in the right of Figure 3. During two-stage training, downstream task performance notably improves after the switch from the generative to the discriminative objective, but does not end up outscoring joint training. Although still outperforming BERT, we found adversarial training to underperform maximum-likelihood training. Further analysis suggests the gap is caused by two
**ELECTRA efficiency analyses**

**Full ELECTRA**

- **Generator** (typically a small MLM; paper uses the BERT loss)
  - The $\rightarrow$ [MASK]$\rightarrow$
  - Chef $\rightarrow$ Chef $\rightarrow$
  - Cooked $\rightarrow$ [MASK]$\rightarrow$
  - The $\rightarrow$ The $\rightarrow$
  - Meal $\rightarrow$ Meal $\rightarrow$

- **Discriminator (ELECTRA)**
  - The $\rightarrow$ Original
  - Chef $\rightarrow$ Original
  - Ate $\rightarrow$ Replaced
  - The $\rightarrow$ Original
  - Meal $\rightarrow$ Original

$\times$ masked $\times$ corrupt
ELECTRA efficiency analyses

ELECTRA 15%

Generator (typically a small MLM; paper uses the BERT loss)

the → [MASK] →
chef → chef →
cooked → [MASK] →
the → the →
meal → meal →

Discriminator (ELECTRA)

the →
chef →
athe →
meal →

original
replaced
ELECTRA efficiency analyses

Replace MLM

- the → dog
- chef → chef
- cooked → run
- the → the
- meal → meal

Generator (typically a small MLM; paper uses the BERT loss)

- the
- chef
- run
- the
- meal

Discriminator (ELECTRA)

- original
- original
- replaced
- original
- original

Replace MLM

- x
- x^masked
- x^corrupt
ELECTRA efficiency analyses

All-tokens MLM

Generator (typically a small MLM; paper uses the BERT loss)

- the → dog
- chef → chef
- cooked → run
- the → the
- meal → meal

x masked

Discriminator (ELECTRA)

- the → original
- chef → original
- ate → replaced
- the → original
- meal → original

x corrupted
ELECTRA efficiency analyses

<table>
<thead>
<tr>
<th>Model</th>
<th>GLUE score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELECTRA</td>
<td>85.0</td>
</tr>
<tr>
<td>All-tokens MLM</td>
<td>84.3</td>
</tr>
<tr>
<td>Replace MLM</td>
<td>82.4</td>
</tr>
<tr>
<td>ELECTRA 15%</td>
<td>82.4</td>
</tr>
<tr>
<td>BERT</td>
<td>82.2</td>
</tr>
</tbody>
</table>
ELECTRA model releases

Available from the project site:

<table>
<thead>
<tr>
<th>Model</th>
<th>Layers</th>
<th>Hidden Size</th>
<th>Params</th>
<th>GLUE test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>12</td>
<td>256</td>
<td>14M</td>
<td>77.4</td>
</tr>
<tr>
<td>Base</td>
<td>12</td>
<td>768</td>
<td>110M</td>
<td>82.7</td>
</tr>
<tr>
<td>Large</td>
<td>24</td>
<td>1024</td>
<td>335M</td>
<td>85.2</td>
</tr>
</tbody>
</table>

‘Small’ is the model designed to be “quickly trained on a single GPU”.
XLNet

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Addressing the known limitations with BERT

1. Devlin et al. (2019:§5): admirably detailed but still partial ablation studies and optimization studies.

2. Devlin et al. (2019): “The first [downside] is that we are creating a mismatch between pre-training and fine-tuning, since the [MASK] token is never seen during fine-tuning.”

3. Devlin et al. (2019): “The second downside of using an MLM is that only 15% of tokens are predicted in each batch”

4. Yang et al. (2019): “BERT assumes the predicted tokens are independent of each other given the unmasked tokens, which is oversimplified as high-order, long-range dependency is prevalent in natural language”
Transformer dimensions (almost) independent

The order of the positions doesn’t matter except for the positional encodings at the bottom.
Conditional language modeling

\[ \alpha \exp \left( x_{30}^T T \left( h_2 \right) \right) \]
Comparison with BERT

\[ \alpha \exp \left( x_{34}^T c_{\text{out}} \right) \]

Transformer blocks
The two objective functions

For vocabulary $\mathcal{V}$, sequence $\mathbf{x} = [x_1, \ldots, x_T]$, and word-level embedding $e$:

**Language model**

$$
\max_{\theta} \sum_{t=1}^{T} \log \frac{\exp (e(x_t)^\top h_\theta(x_{1:t-1}))}{\sum_{x' \in \mathcal{V}} \exp (e(x')^\top h_\theta(x_{1:t-1}))}
$$

for RNN parameters $h_\theta$.

**BERT**

$$
\max_{\theta} \sum_{t=1}^{T} m_t \log \frac{\exp (e(x_t)^\top H_\theta(\hat{x}_t))}{\sum_{x' \in \mathcal{V}} \exp (e(x')^\top H_\theta(\hat{x}_t))}
$$

for Transformer parameters $H_\theta$, with $m_t = 1$ if token $t$ was masked, else 0.
Permutation orders

Yang et al. 2019:§2.2
Permutation orders

Yang et al. 2019: §2.2
**XLNet permutation orders**

Figure 4: Illustration of the permutation language modeling objective for predicting $x_3$ given the same input sequence $x$ but with different factorization orders.

Yang et al. 2019:§A.7
Lack of sensitivity to the target position

\[
\max_\theta \sum_{t=1}^{T} \log \frac{\exp(e(x_t)^T h_\theta(x_{1:t-1}))}{\sum_{x' \in \mathcal{V}} \exp(e(x')^T h_\theta(x_{1:t-1}))}
\]

Yang et al. 2019:§2.2, A.1
Two-stream attention: order $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$

Content stream

Joint View of the Content Stream
(Factorization order: $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$)
Two-stream attention: order 3 → 2 → 4 → 1
Two-stream attention: order $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$

Query stream

Joint View of the Query Stream
(Factorization order: $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$)
Two-stream attention: order 3 → 2 → 4 → 1

Query stream

Position-3 View

Position-2 View

Position-4 View

Position-1 View
Two-stream attention: order 3 → 2 → 4 → 1

Content stream

Query stream

Yang et al. 2019:§2.2, A.7
XLNet model releases

From https://github.com/zihangdai/xlnet:

<table>
<thead>
<tr>
<th>Model</th>
<th>Layers</th>
<th>Hidden Size</th>
<th>Heads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large, Cased</td>
<td>24</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>Base, Cased</td>
<td>12</td>
<td>768</td>
<td>12</td>
</tr>
</tbody>
</table>

See also https://huggingface.co/models?search=xlnet
Conditional dependencies

For sampled permutation order [is, a, city, New, York] and prediction targets \{New, York\}:

\[
\mathcal{J}_{\text{BERT}} = \log p(\text{New} | \text{is a city}) + \log p(\text{York} | \text{is a city}),
\]
\[
\mathcal{J}_{\text{XLNet}} = \log p(\text{New} | \text{is a city}) + \log p(\text{York} | \text{New, is a city}).
\]

Yang et al. 2019:§2.6
contextualreps.ipynb

1. Overview: Resources and guiding insights
2. ELMo: Embeddings from Language Models
3. Transformers
4. BERT: Bidirectional Encoder Representations from Transformers
5. RoBERTa: Robustly optimized BERT approach
6. ELECTRA: Efficiently Learning an Encoder that Classifies Token Replacements Accurately
7. XLNet
8. contextualreps.ipynb: Easy ways to bring ELMo and BERT into your project
Guiding idea

1. Your existing architecture can benefit from contextual representations.

2. `contextualreps.ipynb` shows you how to bring in ELMo and BERT representations:
   - Simple featurization
   - Fine-tuning

3. By extending existing PyTorch modules for this course, you can create *customized* fine-tuning models with just a few lines of code.

4. (This is possible only because of the amazing work that the Hugging Face and AllenNLP groups have done.)!
Standard RNN dataset preparation

<table>
<thead>
<tr>
<th>Examples</th>
<th>[a, b, a]</th>
<th>[b, c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indices</td>
<td>[1, 2, 1]</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>Vectors</td>
<td>$[-0.42 \ 0.10 \ 0.12], [-0.16 \ -0.21 \ 0.29], [-0.42 \ 0.10 \ 0.12]$</td>
<td>$[-0.16 \ -0.21 \ 0.29], [-0.26 \ 0.31 \ 0.37]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
RNN contextual representation inputs

Examples

\[[a, b, a]\]
\[[b, c]\]

Vectors

\[
\begin{bmatrix}
-0.41 & -0.08 & 0.27 \\
0.17 & -0.22 & 0.78 \\
-0.46 & 0.24 & 0.12 \\
-0.02 & -0.56 & 0.11 \\
-0.45 & 0.43 & 0.32
\end{bmatrix}
\]
Code snippet: ELMo RNN inputs

```python
[1]: from allennlp.commands.elmo import ElmoEmbedder
from torch_rnn_classifier import TorchRNNClassifier
import os, sst

[2]: s3="https://allennlp.s3.amazonaws.com/models/elmo/2x4096_512_2048cnn_2xhighway/
options_file = s3 + "elmo_2x4096_512_2048cnn_2xhighway_options.json"
weights_file = s3 + "elmo_2x4096_512_2048cnn_2xhighway_weights.hdf5"

[3]: SST_HOME = os.path.join("data", "trees")

[4]: elmo_embedder = ElmoEmbedder(options_file, weights_file)

[5]: def elmo_sentence_phi(tree):
    vecs = elmo_embedder.embed_sentence(tree.leaves())
    return vecs[-1]

[6]: def fit_preactturized_rnn(X, y):
    mod = TorchRNNClassifier(
        vocab=[],
        max_iter=50,
        use_embedding=False)
    mod.fit(X, y)
    return mod

[7]: _ = sst.experiment(
    SST_HOME,
    elmo_sentence_phi,
    fit_preactturized_rnn,
    train_reader=sst.train_reader,
    assess_reader=sst.dev_reader,
    class_func=sst.ternary_class_func,
    vectorize=False)
```
Code snippet: BERT RNN inputs

```python
[1]: import torch  
    from torch_rnn_classifier import TorchRNNClassifier  
    from transformers import BertModel, BertTokenizer  
    import os, sst

[2]: SST_HOME = os.path.join("data", "trees")

[3]: hf_weights_name = 'bert-base-cased'

[4]: hf_tokenizer = BertTokenizer.from_pretrained(hf_weights_name)

[5]: hf_model = BertModel.from_pretrained(hf_weights_name)

[6]: def hugging_face_bert_phi(tree):  
    s = " ".join(tree.leaves())  
    input_ids = hf_tokenizer.encode(s, add_special_tokens=True)  
    X = torch.tensor([input_ids])  
    with torch.no_grad():  
        final_hidden_states, cls_output = hf_model(X)  
        return final_hidden_states.squeeze(0).numpy()

[7]: def fit_prefeaturized_rnn(X, y):  
    mod = TorchRNNClassifier(  
        vocab=[],  
        max_iter=50,  
        use_embedding=False)  
    mod.fit(X, y)  
    return mod

[8]: experiment = sst.experiment(  
    SST_HOME,  
    hugging_face_bert_phi,  
    fit_prefeaturized_rnn,  
    train_reader=sst.train_reader,  
    assess_reader=sst.dev_reader,  
    class_func=sst.ternary_class_func,  
    vectorize=False)  # Pass in the BERT hidden states directly!
```
Code snippet: ELMo fine-tuning with AllenNLP

```python
[1]: from allennlp.modules.elmo import Elmo, batch_to_ids
    import torch
    import torch.nn as nn
    from torch_rnn_classifier import TorchRNClassifier, TorchRNClassifierModel
    import os, sst

[2]: s3="https://allennlp.s3.amazonaws.com/models/elmo/2x4096_512_2048cnn_2xhighway/
    options_file = s3 + "elmo_2x4096_512_2048cnn_2xhighway_options.json"
    weights_file = s3 + "elmo_2x4096_512_2048cnn_2xhighway_weights.hdf5"

[3]: class ElMoRNClassifierModel(TorchRNClassifierModel):
    def __init__(self, options_file, weights_file,
                 hidden_dim, output_dim, bidirectional, device):
        super().__init__(vocab_size=0,
                         embed_dim=1024, # self.elmo.get_output_dim()
                         use_embedding=False, embedding=None,
                         hidden_dim=hidden_dim, output_dim=output_dim,
                         bidirectional=bidirectional, device=device)
        self.options_file = options_file
        self.weights_file = weights_file
        self.elmo = Elmo(
            self.options_file,
            self.weights_file,
            num_output_representations=2,
            dropout=0)

    def forward(self, X, seq_lengths):
        X = X.to(self.device, non_blocking=True)
        result = self.elmo(X)
        X = result['elmo_representations'][-1]
        state = self.rnn_forward(X, seq_lengths, self.rnn)
        logits = self.classifier_layer(state)
        return logits
```
Code snippet: ELMo fine-tuning with AllenNLP

```python
[4]:
class ElmoRNNClassifier(TorchRNNClassifier):
    def __init__(self, options_file, weights_file, *args, **kwargs):
        self.options_file = options_file
        self.weights_file = weights_file
        vocab = []
        super().__init__(
            vocab, *args, use_embedding=False, embedding=None, **kwargs)

    def build_graph(self):
        elmo = ElmoRNNClassifierModel(
            options_file=self.options_file,
            weights_file=self.weights_file,
            hidden_dim=self.hidden_dim,
            output_dim=self.n_classes,
            bidirectional=self.bidirectional,
            device=self.device)
        elmo.train()
        return elmo

    def _prepare_dataset(self, X):
        seq_lengths = [sum([1 for w in ex if w.sum() > 0]) for ex in X]
        return X, torch.tensor(seq_lengths)

@staticmethod
def encode(X):
    return batch_to_ids(X)

[5]:
mod = ElmoRNNClassifier(
    options_file,
    weights_file,
    batch_size=16,
    max_iter=10,  # More iters improves things. How many did the ELMo team do?
    eta=0.0001,
    l2_strength=0.0001)
```
Code: BERT fine-tuning with Hugging Face

```python
[1]: import torch
import torch.nn as nn
from torch_shallow_neural_classifier import TorchShallowNeuralClassifier
from transformers import BertModel, BertTokenizer

class HfBertClassifierModel(nn.Module):
    def __init__(self, n_classes, weights_name='bert-base-cased'):
        super().__init__()
        self.n_classes = n_classes
        self.weights_name = weights_name
        self.bert = BertModel.from_pretrained(self.weights_name)
        self.hidden_dim = self.bert.embeddings.word_embeddings.embedding_dim
        self.W = nn.Linear(self.hidden_dim, self.n_classes)

    def forward(self, X):
        indices = X[:, 0, :]
        indices = indices.long()
        mask = X[:, 1, :]
        (final_hidden_states, cls_output) = self.bert(
            indices, attention_mask=mask)
        return self.W(cls_output)
```
Code: BERT fine-tuning with Hugging Face

```python
[3]: class HfBertClassifier(TorchShallowNeuralClassifier):
    def __init__(self, weights_name, *args, **kwargs):
        self.weights_name = weights_name
        self.tokenizer = BertTokenizer.from_pretrained(self.weights_name)
        super().__init__(*args, **kwargs)

    def define_graph(self):
        bert = HfBertClassifierModel(
            self.n_classes_, weights_name=self.weights_name)
        bert.train()
        return bert

    def encode(self, X, max_length=None):
        data = self.tokenizer.batch_encode_plus(
            X,
            max_length=max_length,
            add_special_tokens=True,
            pad_to_max_length=True,
            return_attention_mask=True)
        indices = data['input_ids']
        mask = data['attention_mask']
        return [[i, m] for i, m in zip(indices, mask)]

[4]: mod = HfBertClassifier(
    'bert-base-cased',
    batch_size=16, # Crucial; large batches will eat up all your memory!
    max_iter=4,
    eta=0.00002)
```
References


