The amazing world of Neural Language Generation

Challenges for deploying Natural Language Generation models in production
Many different types of models have been developed over the long history of NLG. In this chapter, we will focus on the production challenges associated with deploying the recent wave of models based on large-scale pretrained Transformer models.
Scope of this part

● What has already been covered in previous chapters:
  ○ Weaknesses and limits of languages models, in particular:
    ■ coherency, consistency, accuracy, repetitions, hallucinations, fairness, bias

  We won’t focus on these topics again here

● What we will cover in this chapter
  ○ Practical considerations impeding easy deployment of LM-based NLG systems:
    ■ Speed, memory and energy consumption
    ■ Hands-on
The cost of deploying large-scale models

- Setting apart the cost of training, **deploying** a large-scale transformer in production can bear a significant cost.

Based on what we know, it would be safe to say the hardware costs of running GPT-3 would be between $100,000 and $150,000 without factoring in other costs (electricity, cooling, backup, etc.).

Alternatively, if run in the cloud, GPT-3 would require something like Amazon’s p3dn.24xlarge instance, which comes packed with 8xTesla V100 (32 GB), 768 GB RAM, and 96 CPU cores, and costs $10-30/hour depending on your plan. That would put the yearly cost of running the model at a minimum of $87,000.

https://bdtechtalks.com/2020/09/21/gpt-3-economy-business-model/
https://twitter.com/matthausk/status/1301474259915755521
Strategies to optimize speed/memory/consumption

Various aspects can be investigated:

- High-level optimizations:
  - Efficient model architectures
  - Efficient decoding algorithms

- Implementation level optimizations
  - Framework for computation
  - Efficient model deployment

And the interactions between the above topics :)
High-level optimizations

More efficient model architectures:

- In pre-training optimizations
  - Controlling the number of parameters
    - Less heads and layers
  - More efficient computations

- Post pre-training optimizations
  - Distillation
  - Pruning

Images by Wannapik and Aneeque Ahmed
High-level optimizations – During pre-training

Controlling the number of parameters:

- We can also prepare to reduce the number of layers
  [Reducing Transformer Depth on Demand with Structured Dropout](#)
  (Fan et al. 2019)

- Or heads
  [Voita et al. 2019](#),
  [Michel et al. 2019](#)
High-level optimizations – Di

Controlling memory/computation ratio:

**Encoder-decoder models**

*One Write-Head is All You Need*

(Shazeer, 2019)

Notations: \( n = \) decoded seq, \( m = \) encoded seq, \( b = \) batch, \( k = \) keys dim, \( d = \) inner dim

The total number of arithmetic operations is \( \Theta(bnd^2) \). (Since the complexity of each of the `tf.einsum` operations above is \( O(bnd^2) \) given the simplifying assumptions.

The total size of memory to be accessed is equal to the sum of the sizes of all the tensors involved: \( O(bnd + bhn^2 + d^2) \). The first term is due to \( X, M, Q, K, V, O \) and \( Y \), the second term due to the logits and weights, and the third term due to the projection tensors \( P_q, P_k, P_v \) and \( P_o \).

Dividing the two, we find that the ratio of memory access to arithmetic operations is \( O(\frac{1}{k} + \frac{1}{bn}) \). This low ratio is necessary for good performance on modern GPU/TPU hardware, where the computational capacity can be two orders of magnitude higher than the memory bandwidth.
High-level optimizations – During pre-training

Transformers are largely memory-bound

“Over a third (37%) of the runtime in a BERT training iteration is spent in memory-bound operators: While tensor contractions account for over 99% of the flop performed, they are only 61% of the runtime.”

*Data Movement Is All You Need: A Case Study on Optimizing Transformers (Ivanov et al. 2020)*

Controlling memory/computation trade-off: Encoder-decoder models

Across $n$ calls, the total number of arithmetic operations is again $\Theta(bnd^2)$. Across $n$ calls, the total amount of memory access is $\Theta(bn^2d + nd^2)$, the first term due to $K$ and $V$ and the second term due to $P_q$, $P_k$, $P_v$ and $P_o$.

Dividing the memory by the computations, we find that the ratio of memory access to arithmetic operations is $\Theta\left(\frac{n}{d} + \frac{1}{b}\right)$. When $n \approx d$ or $b \approx 1$, the ratio is close to 1, causing memory bandwidth to be a major performance bottleneck on modern computing hardware. In order to make incremental generation efficient, we must reduce both of these terms to be $\ll 1$. The $\frac{1}{b}$ term is the easier one - we can just use a larger batch size, memory size permitting.
High-level optimizations – During pre-training

More efficient computations: Fusing operations

https://microsoft.github.io/onnxruntime/docs/resources/graph-optimizations.html

Transformer-based networks trigger many invocations of CUDA kernels adding a lot of cost for transferring data to/from global memory & overhead from kernel launching
High-level optimizations

Multi-Query self-attention in the decoder

One Write-Head is All You Need (Shazeer, 2019)

<table>
<thead>
<tr>
<th>Attention Type</th>
<th>$d_k, d_o$</th>
<th>$d_{ff}$</th>
<th>ln(PPL) (dev)</th>
<th>BLEU (dev)</th>
<th>BLEU (test) beam 1 / 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi-head</td>
<td>8</td>
<td>128</td>
<td>4096</td>
<td>1.424</td>
<td>26.7</td>
</tr>
<tr>
<td>multi-query</td>
<td>8</td>
<td>128</td>
<td>5440</td>
<td>1.439</td>
<td>26.5</td>
</tr>
<tr>
<td>multi-head local</td>
<td>8</td>
<td>128</td>
<td>4096</td>
<td>1.427</td>
<td>26.6</td>
</tr>
<tr>
<td>multi-query local</td>
<td>8</td>
<td>128</td>
<td>5440</td>
<td>1.437</td>
<td>26.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attention Type</th>
<th>Training</th>
<th>Inference</th>
<th>Beam-4 Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>enc. + dec.</td>
<td>enc. + dec.</td>
<td>enc. + dec.</td>
</tr>
<tr>
<td>multi-head</td>
<td>13.2</td>
<td>1.7 + 46</td>
<td>2.0 + 203</td>
</tr>
<tr>
<td>multi-query</td>
<td>13.0</td>
<td>1.5 + 3.8</td>
<td>1.6 + 32</td>
</tr>
<tr>
<td>multi-head local</td>
<td>13.2</td>
<td>1.7 + 23</td>
<td>1.9 + 47</td>
</tr>
<tr>
<td>multi-query local</td>
<td>13.0</td>
<td>1.5 + 3.3</td>
<td>1.6 + 16</td>
</tr>
</tbody>
</table>

Dividing the memory by the computations, we find that the ratio of memory access to arithmetic operations is $\Theta(\frac{1}{d} + \frac{n}{dn} + \frac{1}{b})$. We have reduced the offensive $\frac{n}{d}$ by a factor of $h$. Theoretically, given large batch size $b$, this should dramatically improve performance of incremental generation. In our experimental section, we will show that the performance gains are real and that model quality remains high.
### High-level optimizations – During pre-training

Controlling the computational complexity:

- **Efficient Transformers: A Survey** (Tay et al. 2020)

<table>
<thead>
<tr>
<th>Model / Paper</th>
<th>Complexity</th>
<th>Decode</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Compressed† (Liu et al., 2018)</td>
<td>$\mathcal{O}(n^2)$</td>
<td>✓</td>
<td>FP+M</td>
</tr>
<tr>
<td>Image Transformer† (Parmar et al., 2018)</td>
<td>$\mathcal{O}(n.m)$</td>
<td>✓</td>
<td>FP</td>
</tr>
<tr>
<td>Set Transformer† (Lee et al., 2019)</td>
<td>$\mathcal{O}(nk)$</td>
<td>x</td>
<td>M</td>
</tr>
<tr>
<td>Transformer-XL† (Dai et al., 2019)</td>
<td>$\mathcal{O}(n^2)$</td>
<td>✓</td>
<td>RC</td>
</tr>
<tr>
<td>Sparse Transformer (Child et al., 2019)</td>
<td>$\mathcal{O}(n_v + n_i)$</td>
<td>✓</td>
<td>FP</td>
</tr>
<tr>
<td>Reformer† (Kitaev et al., 2020)</td>
<td>$\mathcal{O}(n \log n)$</td>
<td>✓</td>
<td>LP</td>
</tr>
<tr>
<td>Routing Transformer (Roy et al., 2020)</td>
<td>$\mathcal{O}(n \log n)$</td>
<td>✓</td>
<td>LP</td>
</tr>
<tr>
<td>Axial Transformer (Ho et al., 2019)</td>
<td>$\mathcal{O}(n_v/n_i)$</td>
<td>✓</td>
<td>FP</td>
</tr>
<tr>
<td>Compressive Transformer† (Rae et al., 2020)</td>
<td>$\mathcal{O}(n^2)$</td>
<td>✓</td>
<td>RC</td>
</tr>
<tr>
<td>Sinkhorn Transformer† (Tay et al., 2020b)</td>
<td>$\mathcal{O}(b^2)$</td>
<td>✓</td>
<td>LP</td>
</tr>
<tr>
<td>Longformer (Beltagy et al., 2020)</td>
<td>$\mathcal{O}(n(k + m))$</td>
<td>✓</td>
<td>FP+M</td>
</tr>
<tr>
<td>ETC (Ainslie et al., 2020)</td>
<td>$\mathcal{O}(n^2 g + n n_g)$</td>
<td>x</td>
<td>FP+M</td>
</tr>
<tr>
<td>Synthesizer (Tay et al., 2020a)</td>
<td>$\mathcal{O}(n^2)$</td>
<td>✓</td>
<td>LR+LP</td>
</tr>
<tr>
<td>Performer (Choromanski et al., 2020)</td>
<td>$\mathcal{O}(n)$</td>
<td>✓</td>
<td>KR</td>
</tr>
<tr>
<td>Linformer (Wang et al., 2020b)</td>
<td>$\mathcal{O}(n)$</td>
<td>x</td>
<td>LR</td>
</tr>
<tr>
<td>Linear Transformers† (Katharopoulos et al., 2020)</td>
<td>$\mathcal{O}(n)$</td>
<td>✓</td>
<td>KR</td>
</tr>
<tr>
<td>Big Bird (Zaheer et al., 2020)</td>
<td>$\mathcal{O}(n)$</td>
<td>x</td>
<td>FP+M</td>
</tr>
</tbody>
</table>

---

**Figure 2: Taxonomy of Efficient Transformer Architectures.**
High-level optimizations – During pre-training

More efficient computations for pre-training:

- **Invertible operations:**
  - Softmax/LayerNorm are invertible: backward pass is independent of the inputs (Rota Bulò et al. 2018)

- **Gradient checkpointing**
  - [https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9](https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9)

- **Fully invertible networks**
  - Reformer (Kitaev et al. 2020)
High-level optimizations – During pre-training

Preparing for later-optimization:

- Quantization-pruning aware training

Training with Quantization Noise for Extreme Model Compression (Fan et al., 2020)

Figure 1: **Quant-Noise** trains models to be resilient to inference-time quantization by mimicking the effect of the quantization method during training time. This allows for extreme compression rates without much loss in accuracy on a variety of tasks and benchmarks.
High-level optimizations – After training

- Two types of Transformers models are typically used in NLG
  - **Decoder** only (GPT, CTRL, MegatronLM…)
  - **Encoder-Decoder** (BART, T5, Pegasus…)

![Diagram of GPT2 Block and Encoder-Decoder](image)
High-level optimizations – After training

- Optimizing **Encoders:**
  - Knowledge-Distillation
  - Quantization (FP16 or INT8)
  - Pruning

---

**Figure 3:** Accuracy versus queries per second with various optimizations on CPU.

**Table:** Optimization methods added and their impact on performance and cost.

| Optimization methods added                                      | Time (sec) | Cumulative speed-up | Speed-up | Accuracy | USD for 100 M queries |
|================================================================|------------|---------------------|----------|----------|-----------------------|
| Baseline (PyTorch out-of-the-box, 12L, 768)                    | 734.35     | 1.00x               | -        | 74.01    | $4,223                |
| + dynamic sequence length                                      | 209.29     | 3.51x               | 3.51x    | 74.01    | $1,204                |
| + knowledge distillation (4L, 312)                            | 22.5       | 32.64x              | 9.30x    | 74.04    | $129                  |
| + 8-bit quantization + graph optimization                      | 9.97       | 73.66x              | 2.26x    | 73.43    | $57                   |
| + multi-instance inference                                     | 5.68       | 129.29x             | 1.76x    | 73.43    | $33                   |
| + structured pruning                                           |            |                     |          |          |                       |
| 25% heads and 25% hidden states pruned                         | 4.11       | 178.67x             | 1.38x    | 73.36    | $24                   |
| 33% heads and 50% hidden states pruned                         | 3.14       | 233.87x             | 1.81x    | 72.81    | $18                   |

**Young Jin Kim et al. 2020** – FastFormers: Highly Efficient Transformer Models for Natural Language Understanding
High-level optimizations – After training

- **Optimizing Decoder:** compressing through distillation
  - Shrink and Fine-Tune
  - Pseudo-labels
  - Knowledge Distillation

<table>
<thead>
<tr>
<th>Teacher</th>
<th>Size</th>
<th>Data</th>
<th>Teacher Score</th>
<th>SFT Score</th>
<th>Cost</th>
<th>KD Score</th>
<th>Cost</th>
<th>Pseudo Score</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>BART 12-3</td>
<td>XSUM</td>
<td>22.29</td>
<td>21.08</td>
<td>2.5</td>
<td><strong>21.63</strong></td>
<td>6</td>
<td>21.38</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Pegasus 16-4</td>
<td>XSUM</td>
<td>24.56</td>
<td>22.64</td>
<td>13</td>
<td>21.92</td>
<td>22</td>
<td><strong>23.18</strong></td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>BART 12-6</td>
<td>CNN</td>
<td>21.06</td>
<td><strong>21.21</strong></td>
<td>2</td>
<td>20.95</td>
<td>14</td>
<td>19.93</td>
<td>19.5</td>
<td></td>
</tr>
<tr>
<td>Pegasus 16-4</td>
<td>CNN</td>
<td>21.37</td>
<td><strong>21.29</strong></td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>20.1</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Marian 6-3</td>
<td>EN-RO</td>
<td>27.69</td>
<td>25.91</td>
<td>4</td>
<td>24.96</td>
<td>4</td>
<td><strong>26.85</strong></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>mBART 12-3</td>
<td>EN-RO</td>
<td>26.457</td>
<td>25.6083</td>
<td>16</td>
<td>25.87</td>
<td>24</td>
<td><strong>26.09</strong></td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

**Teacher** | **Student** | **MM Params** | **Time (MS)** | **Speedup** | **Rouge-2** | **Rouge-L** |
--- | --- | --- | --- | --- | --- | --- |
BART 12-1 | 222 | 743 | 2.35 | 17.98 | 33.31 |
BART 12-3 | 255 | 905 | 1.93 | **22.40** | 37.30 |
BART 6-6 | 230 | 1179 | 1.48 | 21.17 | 36.21 |
BART 9-6 | 268 | 1184 | 1.47 | 22.08 | 37.24 |
BART 12-6 | 306 | 1221 | 1.43 | 22.32 | **37.39** |
Baseline (12-12) | 406 | 1743 | 1.00 | 22.29 | 37.20 |

Pegasus 16-4 | 369 | 2038 | 2.40 | 23.18 | 38.13 |
Pegasus 16-8 | 435 | 2515 | 1.94 | 23.25 | 38.03 |
Baseline (16-16) | 570 | 4890 | 24.46 | **39.15** |

**BertABS** | **Baseline (6-6)** | 110 | 1120 | 16.50 | 31.27 |

Table 5: Main results. Score is Rouge-2 for the 2 summarization datasets (first 4 rows), and BLEU for the bottom two rows. Cost measures the GPU hours required to run the approach end to end, which, in the case of Pseudo-labeling, requires running beam search on the full training set. The highest scoring distillation technique is in bold.

Table 6: Best XSUM results across all methods. Each sub-table is sorted fastest to slowest by inference time. dBART-12-3 and dBPEGASUS-16-4 are trained on Pegasus pseudo-labels. dBART-12-6, dBART-6-6, and dBART-9-6 are trained with KD. dBPEGASUS-16-8 and dBART-12-1 are trained with SFT. For the BART experiments where the encoder is smaller than 12 layers, we do not freeze it during training.

S Shleifer, A M. Rush 2020 – Pre-trained Summarization Distillation
High-level optimizations – Decoding

- Optimizing **Decoding algorithm:**
  - Caching
    - Caching encoder
  - Caching decoding
    Transformer-XL
    *(Dai et al. 2019)*
High-level optimizations – Decoding

- Optimizing Decoding: Decoding algs

![Diagram of Greedy Search, Beam Search, Sampling, and Top-K Sampling](image)
Implementation level optimizations – Decoding

- Optimizing **Decoding**:  
  - Greedy-search - Sampling  
    - Compile decoding loop  
      - JAX, ONNX, TF  
  - Beam-search  
    - **A Streaming Approach For Efficient Batched Beam Search** (Yang et al. 2020)
Implementation level optimizations

- **Low level optimizations**: Controlling parallelism:
  - PyTorch/TensorFlow typically use all available CPU cores
  - Transformer ops not always big enough to fully utilize parallelism of many cores
  - Overheads of parallelizing can overshadow actual gains
  - => Control level of parallelism

<table>
<thead>
<tr>
<th>Number of inference instances</th>
<th>Time (sec)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (no thread control)</td>
<td>433</td>
<td>1.00x</td>
</tr>
<tr>
<td>1 instance (20 threads/instance)</td>
<td>319</td>
<td>1.36x</td>
</tr>
<tr>
<td>2 instances (10 threads/instance)</td>
<td>243</td>
<td>1.78x</td>
</tr>
<tr>
<td>4 instance (5 threads/instance)</td>
<td>247</td>
<td>1.75x</td>
</tr>
<tr>
<td>5 instance (4 threads/instance)</td>
<td>255</td>
<td>1.70x</td>
</tr>
<tr>
<td>10 instance (2 threads/instance)</td>
<td>300</td>
<td>1.44x</td>
</tr>
<tr>
<td>20 instance (1 thread/instance)</td>
<td>351</td>
<td>1.23x</td>
</tr>
</tbody>
</table>

*Table 2: Speed comparison of different number of inference instances with thread control - time to perform inference on 1,000 ReCoRD validation data samples.*

*Young Jin Kim et al. 2020*
Now a simple hands-on to finish the session :) 

We will reproduce the results of “Leveraging Pre-trained Checkpoints for Sequence Generation Tasks” (TACL 2020) by Sascha Rothe, Shashi Narayan and Aliaksei Severyn from Google and use it in a simple API.

This paper aim to “provide an empirical answer to the following research question: what is the best way to leverage publicly available pre-trained checkpoints for warm-starting sequence generation models?”

E.g. using BERT checkpoint to initialize the encoder for better input understanding and choosing GPT-2 model as the decoder for better text generation?
Now a simple hands-on to finish the session :) 

This paper rigorously experiment with a large number of different settings to combine BERT, GPT and RoBERTa pre-trained checkpoints to initialize a Transformer-based model for:

- sentence-level fusion/splitting
- machine translation
- abstractive summarization

<table>
<thead>
<tr>
<th>Model</th>
<th>Gigaword R-1</th>
<th>Gigaword R-2</th>
<th>Gigaword R-L</th>
<th>CNN/Dailymail R-1</th>
<th>CNN/Dailymail R-2</th>
<th>CNN/Dailymail R-L</th>
<th>BBC XSum R-1</th>
<th>BBC XSum R-2</th>
<th>BBC XSum R-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND2RND</td>
<td>221M</td>
<td>23M</td>
<td>0</td>
<td>221M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERT2RND</td>
<td>221M</td>
<td>23M</td>
<td>109M</td>
<td>112M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RND2BERT</td>
<td>221M</td>
<td>23M</td>
<td>109M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERT2BERT</td>
<td>221M</td>
<td>23M</td>
<td>195M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERTSHARE</td>
<td>136M</td>
<td>23M</td>
<td>109M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROBERTA2BERTSHARE</td>
<td>152M</td>
<td>39M</td>
<td>125M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPT</td>
<td>125M</td>
<td>39M</td>
<td>125M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RND2GPT</td>
<td>238M</td>
<td>39M</td>
<td>125M</td>
<td>114M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERT2GPT</td>
<td>260M</td>
<td>62M</td>
<td>234M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROBERTA2GPT</td>
<td>276M</td>
<td>78M</td>
<td>250M</td>
<td>26M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The number of total trainable parameters, embedding parameters and parameters initialized from the checkpoint vs. randomly. The BERT/GPT-2 embeddings have 23M/39M parameters. The encoder-decoder attention accounts for 26M parameters.

Initial with the base checkpoint (12 layers)

<table>
<thead>
<tr>
<th>Model</th>
<th>Gigaword R-1</th>
<th>Gigaword R-2</th>
<th>Gigaword R-L</th>
<th>CNN/Dailymail R-1</th>
<th>CNN/Dailymail R-2</th>
<th>CNN/Dailymail R-L</th>
<th>BBC XSum R-1</th>
<th>BBC XSum R-2</th>
<th>BBC XSum R-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND2RND</td>
<td>36.94</td>
<td>18.71</td>
<td>34.45</td>
<td>35.77</td>
<td>14.00</td>
<td>32.96</td>
<td>30.90</td>
<td>10.23</td>
<td>24.24</td>
</tr>
<tr>
<td>BERT2RND</td>
<td>37.71</td>
<td>19.26</td>
<td>35.26</td>
<td>38.74</td>
<td>17.76</td>
<td>35.95</td>
<td>38.42</td>
<td>15.83</td>
<td>30.80</td>
</tr>
<tr>
<td>RND2BERT</td>
<td>37.01</td>
<td>18.91</td>
<td>34.51</td>
<td>36.65</td>
<td>15.55</td>
<td>33.97</td>
<td>32.44</td>
<td>11.52</td>
<td>25.65</td>
</tr>
<tr>
<td>BERT2BERT</td>
<td>38.01</td>
<td>19.68</td>
<td>35.58</td>
<td>39.02</td>
<td>17.84</td>
<td>36.29</td>
<td>37.53</td>
<td>15.24</td>
<td>30.05</td>
</tr>
<tr>
<td>ROBERTA2BERTSHARE</td>
<td>38.21</td>
<td>19.70</td>
<td>35.44</td>
<td>40.10</td>
<td>18.95</td>
<td>37.39</td>
<td>39.87</td>
<td>17.50</td>
<td>32.37</td>
</tr>
<tr>
<td>GPT</td>
<td>36.04</td>
<td>18.44</td>
<td>33.67</td>
<td>37.26</td>
<td>15.83</td>
<td>34.47</td>
<td>22.21</td>
<td>4.89</td>
<td>16.69</td>
</tr>
<tr>
<td>RND2GPT</td>
<td>36.21</td>
<td>18.39</td>
<td>33.83</td>
<td>32.08</td>
<td>8.81</td>
<td>29.03</td>
<td>28.48</td>
<td>8.77</td>
<td>22.30</td>
</tr>
<tr>
<td>BERT2GPT</td>
<td>36.77</td>
<td>18.23</td>
<td>34.24</td>
<td>25.20</td>
<td>4.96</td>
<td>22.99</td>
<td>27.79</td>
<td>8.37</td>
<td>21.91</td>
</tr>
<tr>
<td>ROBERTA2GPT</td>
<td>37.94</td>
<td>19.21</td>
<td>35.42</td>
<td>36.35</td>
<td>14.72</td>
<td>33.79</td>
<td>19.91</td>
<td>5.20</td>
<td>15.88</td>
</tr>
</tbody>
</table>

Initial with the large checkpoint (24 layers)

<table>
<thead>
<tr>
<th>Model</th>
<th>Gigaword R-1</th>
<th>Gigaword R-2</th>
<th>Gigaword R-L</th>
<th>CNN/Dailymail R-1</th>
<th>CNN/Dailymail R-2</th>
<th>CNN/Dailymail R-L</th>
<th>BBC XSum R-1</th>
<th>BBC XSum R-2</th>
<th>BBC XSum R-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERTSHARE</td>
<td>38.35</td>
<td>19.80</td>
<td>35.66</td>
<td>39.83</td>
<td>17.69</td>
<td>37.01</td>
<td>38.93</td>
<td>16.35</td>
<td>31.52</td>
</tr>
<tr>
<td>ROBERTA2BERTSHARE</td>
<td>38.62</td>
<td>19.78</td>
<td>35.94</td>
<td>40.31</td>
<td>18.91</td>
<td>37.62</td>
<td>41.45</td>
<td>18.79</td>
<td>33.90</td>
</tr>
</tbody>
</table>
Load the CNN/DailyMail dataset

- Companion Notebook by Patrick van Platen is here: https://colab.research.google.com/drive/1WIk2bxgIElfZewOHboPFNj8H44_VAyKE

```python
!pip install datasets==1.0.2
import datasets
train_data = datasets.load_dataset("cnn_dailymail", "3.0.0", split="train")

from pprint import pprint
pprint(train_data.info.description)
pprint(train_data[0], width=1e3)
```

- CNN/DailyMail non-anonymized summarization dataset.

There are two features:
- article: text of news article, used as the document to be summarized
- highlights: joined text of highlights with <s> and </s> around each highlight, which is the target summary

```python
{'article': 'It\'s official: U.S. President Barack Obama wants lawmakers to weigh in on whether to use military force in Syria. Obama sent a letter to Congress on Saturday, saying the United States cannot accept a "permanent military presence" if it does not have "self-defense". He said that he would not be open-ended or include U.S. ground forces, and that Syria\'s alleged use of chemical weapons earlier this month \"is a threat to our national security.\" He also said \"I believe that the current humanitarian crisis in Syria is a threat to world peace,\" which is taking place in the Middle East. The two leaders agreed that the international community must deliver a resolve message to the Assad regime.

Boehner, Majority Leader Eric Cantor, Majority Whip Kevin McCarthy and Conference Chair Cathy McMorris Rodgers issued a statement saying that \"the military operation did not get enough support either in the world\" and that it used chemical weapons in the August 21 attack, saying that jihadists fighting with the rebels used them in an effort to protect their military levels. Syria\'s prime minister appeared unfazed by the saber-rattling. \"The Syrian Army\"s status is on maximum readiness on an ongoing basis,\" he said. "It is not the first time that our country has faced this type of threat.\"

\"I don\'t know how to get down\"nObama sends a letter to the heads of the id': '0001d1afcc246a7964130f43ae9480af6bc6c57f01'}
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")

# set CLS and SEQ to BOS and EOS token because BERT does not have BOS and EOS by default
tokenizer.bos_token = tokenizer.cls_token
tokenizer.eos_token = tokenizer.sep_token

batch_size=4  # change to 16 for full training
decoder_max_length=128
def prepare_data(batch):
    # tokenize the inputs and labels
    inputs = tokenizer(batch["article"], padding="max_length", truncation=True, max_length=encoder_max_length)
    outputs = tokenizer(batch["highlights"], padding="max_length", truncation=True, max_length=decoder_max_length)

    batch["input_ids"] = inputs.input_ids
    batch["attention_mask"] = inputs.attention_mask
    batch["decoder_input_ids"] = outputs.input_ids
    batch["decoder_attention_mask"] = outputs.attention_mask
    batch["labels"] = outputs.input_ids.copy()

    # because BERT automatically shifts the labels, the labels correspond exactly to `decoder_input_ids`.
    # We have to make sure that the PAD token is ignored
    batch["labels"] = [[-100 if token == tokenizer.pad_token_id else token for token in labels] for labels in batch["labels"]]

    return batch

train_data = train_data.map(prepare_data, batched=True, batch_size=batch_size, remove_columns=["article", "highlights", "id"])
train_data.set_format(type="torch")
Initialize an Encoder-Decoder model from Bert

```python
from transformers import EncoderDecoderModel

In contrast to other model classes in Transformers, the EncoderDecoderModel class has two methods to load pre-trained weights, namely:

1. the "standard" `.from_pretrained(...) method is derived from the general PretrainedModel.from_pretrained(...) method and thus corresponds exactly to the the one of other model classes. The function expects a single model identifier, e.g.
   `.from_pretrained("google/bert2bert_L-24_wmt_de_en") and will load a single .pt checkpoint file into the EncoderDecoderModel class.

2. a special `.from_encoder_decoder_pretrained(...) method, which can be used to warm-start an encoder-decoder model from two model identifiers - one for the encoder and one for the decoder. The first model identifier is thereby used to load the encoder, via AutoModel.from_pretrained(...) (see doc here) and the second model identifier is used to load the decoder via AutoModelForCausalLM (see doc here).

Alright, let’s warm-start our BERT2BERT model. As mentioned earlier we will warm-start both the encoder and decoder with the "bert-base-cased" checkpoint.

```python
t2bert = EncoderDecoderModel.from_encoder_decoder_pretrained("bert-base-uncased", "bert-base-uncased")
```

Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertLMHeadModel: ['cls.seq_relationship.weight', 'cls.bias'] - This IS expected if you are initializing BertLMHeadModel from the checkpoint of a model trained on another task or with another architecture
- This IS NOT expected if you are initializing BertLMHeadModel from the checkpoint of a model that you expect to be exactly identical (initial Some weights of BertLMHeadModel were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['bert.encoder. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Train the Encoder-Decoder model

```python
# load rouge for validation
rouge = datasets.load_metric("rouge")

def compute_metrics(pred):
    labels_ids = pred.label_ids
    pred_ids = pred.predictions

    # all unnecessary tokens are removed
    pred_str = tokenizer.batch_decode(pred_ids, skip_special_tokens=True)
    labels_ids[labels_ids == -100] = tokenizer.pad_token_id
    label_str = tokenizer.batch_decode(labels_ids, skip_special_tokens=True)

    rouge_output = rouge.compute(predictions=pred_str, references=label_str, rouge_types=['rouge2'])

    return {
        "rouge2_precision": round(rouge_output.precision, 4),
        "rouge2_recall": round(rouge_output.recall, 4),
        "rouge2_fmeasure": round(rouge_output.fmeasure, 4),
    }

# set training arguments - these params are not really tuned, feel free to change
training_args = Seq2SeqTrainingArguments(
    output_dir="./",
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    predict_with_generate=True,
    evaluate_during_training=True,
    do_train=True,
    do_eval=True,
    logging_steps=2,  # set to 1000 for full training
    save_steps=16,    # set to 500 for full training
    eval_steps=4,     # set to 8000 for full training
    warmup_steps=1,   # set to 2000 for full training
    max_steps=16,     # delete for full training
    overwrite_output_dir=True,
    save_total_limit=3,
    fp16=True,
)```
Train the Encoder-Decoder model

Because we will use bert2bert for summarization, we set the model's special tokens and active beam_search with sensible parameters.

```python
[ ] # set special tokens
    bert2bert.config.decoder_start_token_id = tokenizer.bos_token_id
    bert2bert.config.eos_token_id = tokenizer.eos_token_id
    bert2bert.config.pad_token_id = tokenizer.pad_token_id

    # sensible parameters for beam search
    bert2bert.config.vocab_size = bert2bert.config.decoder.vocab_size
    bert2bert.config.max_length = 142
    bert2bert.config.min_length = 56
    bert2bert.config.no_repeat_ngram_size = 3
    bert2bert.config.early_stopping = True
    bert2bert.config.length_penalty = 2.0
    bert2bert.config.num_beams = 4
```

```python
[ ] # instantiate trainer
    trainer = Seq2SeqTrainer(
        model=bert2bert,
        args=training_args,
        compute_metrics=compute_metrics,
        train_dataset=train_data,
        eval_dataset=val_data,
    )
    trainer.train()
```

```
/usr/local/lib/python3.6/dist-packages/datasets/arrow_dataset.py:835: UserWarning: The given NumPy array is not writeable, and PyTorch does not return torch.tensor(x, **format_kwargs)


https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

<table>
<thead>
<tr>
<th>Step</th>
<th>Training Loss</th>
<th>Validation Loss</th>
<th>Rouge2 Precision</th>
<th>Rouge2 Recall</th>
<th>Rouge2 FMeasure</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10.000679</td>
<td>10.383123</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>8</td>
<td>8.465530</td>
<td>8.130023</td>
<td>0.004300</td>
<td>0.004800</td>
<td>0.004500</td>
</tr>
<tr>
<td>12</td>
<td>7.704124</td>
<td>7.786637</td>
<td>0.005100</td>
<td>0.003900</td>
<td>0.004400</td>
</tr>
<tr>
<td>16</td>
<td>7.525826</td>
<td>7.755284</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

TrainOutput(global_step=16, training_loss=8.653947830200195)```
Evaluate the Encoder-Decoder model

```python
import datasets
from transformers import BertTokenizer, EncoderDecoderModel

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = EncoderDecoderModel.from_pretrained("./checkpoint-16")
model.to("cuda")

test_data = datasets.load_dataset("cnn_dailymail", "3.0.0", split="test")

# only use 16 training examples for notebook - DELETE LINE FOR FULL TRAINING
# test_data = test_data.select(range(16))

batch_size = 16  # change to 64 for full evaluation

# map data correctly
def generate_summary(batch):
    # Tokenizer will automatically set [BOS] <text> [EOS]
    # cut off at BERT max length 512
    inputs = tokenizer(batch["article"], padding="max_length", truncation=True, max_length=512, return_tensors="pt")
    input_ids = inputs.input_ids.to("cuda")
    attention_mask = inputs.attention_mask.to("cuda")

    outputs = model.generate(input_ids, attention_mask=attention_mask)

    # all special tokens including will be removed
    output_str = tokenizer.batch_decode(outputs, skip_special_tokens=True)

    batch["pred"] = output_str

    return batch

results = test_data.map(generate_summary, batched=True, batch_size=batch_size, remove_columns=["article"], batched=False)
```
Hands-on: summarization with a pretrained Encoder-Decoder model

- Companion Notebook by Patrick van Platen is here: https://colab.research.google.com/drive/1WIk2bxglElfZewOHboPFNj8H44_VAyKE
- The model achieves a ROUGE-2 score of 18.22, which is even a little better than reported in the paper.
- The fully trained BERT2BERT model is uploaded to the HuggingFace model hub: patrickvonplaten/bert2bert_cnn_daily_mail.
- For some summarization examples, the reader can use the online inference widget of the model here.