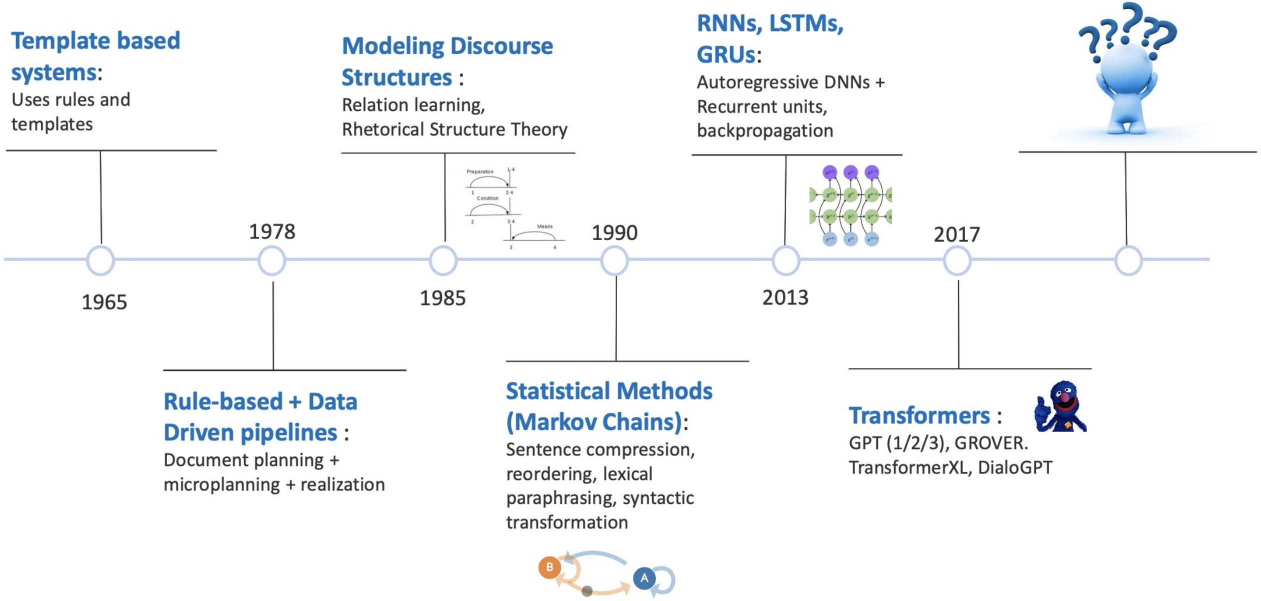
The amazing world of Neural Language Generation

Challenges for deploying Natural Language Generation models in production

Challenges of deploying NLG models in production



Many different type of models have been developed over the long history of NLG.
In this chapter we will focus on the production challenges associated to 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.

Matthäus Krzykowski
@matthausk

Fascinating insight into early GPT-3 economics experiments. Not surprised about the relatively high pricing. I was hearing that even in limited private Beta they have monthly costs in significant 8 digits, despite MSFT Azure backing/discounts.

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

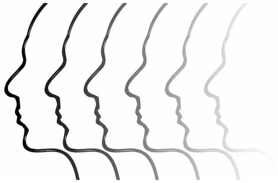
- Implementation level optimizations
  - Framework for computation
  - Efficient model deployment

And the interactions between the above topics :)

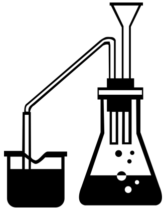
High-level optimizations

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

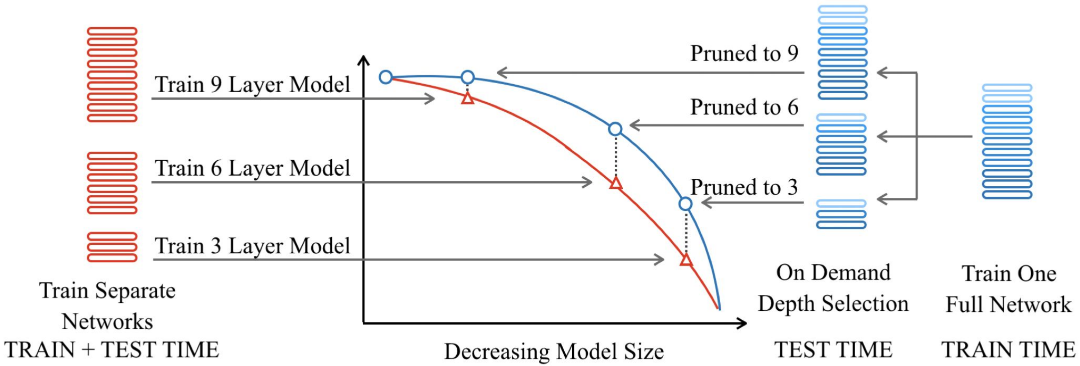


Images by Wannapik and Aneeque Ahmed

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 – D

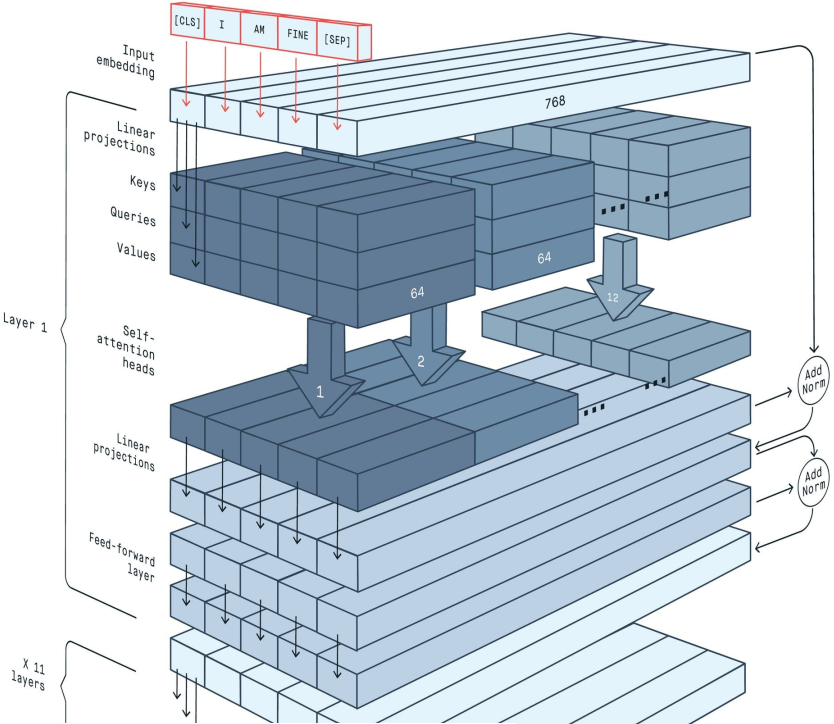
Controlling memory/computation ratio:

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.

PELTARION

peltarion.com

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 $\Theta(\frac{n}{d} + \frac{1}{b})$. 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.

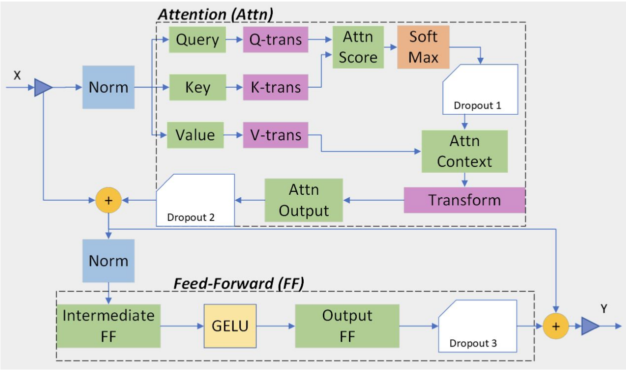
High-level optimizations – During pre-training

More efficient computations: Fusing operations

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

https://www.deepspeed.ai/news/2020/05/27/fastest-bert-training.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



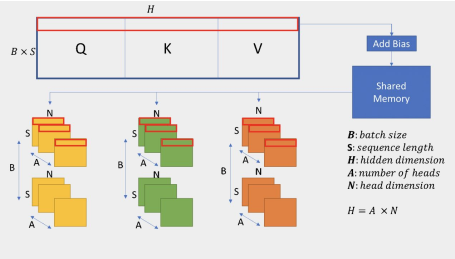


Figure 3: QKV's GEMM and transform Kernel-Fusion

Extended Graph Optimizations

These optimizations include complex node fusions. They are run after graph partitioning and are only applied to the nodes assigned to the CPU or CUDA execution provider. Available extended graph optimizations are as follows:

Optimization	Execution Provider	Comment
GEMM Activation Fusion	cpu	
Matmul Add Fusion	cpu	
Conv Activation Fusion	cpu	
GELU Fusion	cpu or cuda	
Layer Normalization Fusion	cpu or cuda	
BERT Embedding Layer Fusion	cpu or cuda	Fuse BERT embedding layer, layer normalization and attention mask length
Attention Fusion	cpu or cuda	Attention mask has approximation in cuda execution provider
Skip Layer Normalization Fusion	cpu or cuda	Fuse bias of fully connected layer, skip connection and layer normalization
Bias GELU Fusion	cpu or cuda	Fuse bias of fully connected layer and GELU activation
GELU Approximation	cuda	Erf is approximated by a formula using tanh function

High-level optimizations

Multi-Query self-attention in the decoder

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

Attention Type	h	d_k, d_v	d_{ff}	ln(PPL) (dev)	BLEU (dev)	BLEU (test) beam 1 / 4
multi-head	8	128	4096	**1.424**	**26.7**	27.7 / 28.4
multi-query	8	128	5440	1.439	26.5	27.5 / **28.5**
multi-head local	8	128	4096	1.427	26.6	27.5 / 28.3
multi-query local	8	128	5440	1.437	26.5	27.6 / 28.2

Attention Type	Training	Inference enc. + dec.	Beam-4 Search enc. + dec.
multi-head	13.2	1.7 + 46	2.0 + 203
multi-query	**13.0**	1.5 + 3.8	1.6 + 32
multi-head local	13.2	1.7 + 23	1.9 + 47
multi-query local	**13.0**	**1.5 + 3.3**	**1.6 + 16**

```python
def MultiquerySelfAttentionIncremental(
    x, prev_K, prev_V, P_q, P_k, P_v, P_o):
  """Multi-query Self-Attention (one step).
  Args:
    x: a tensor with shape [b, d]
    prev_K: tensor with shape [b, m, k]
    prev_V: tensor with shape [b, m, v]
    P_q: a tensor with shape [h, d, k]
    P_k: a tensor with shape [d, k]
    P_v: a tensor with shape [d, v]
    P_o: a tensor with shape [h, d, v]
  Returns:
    y: a tensor with shape [b, d]
    new_K: tensor with shape [b, m+1, k]
    new_V: tensor with shape [b, m+1, v]
  """
   q = tf.einsum("bd,hdk->bhk", x, P_q)
  K = tf.concat(
     [prev_K, tf.expand_dims(tf.einsum("bd,dk->bk", M, P_k), axis=2)],
     axis=2)
  V = tf.concat(
     [prev_V, tf.expand_dims(tf.einsum("bd,dv->bv", M, P_v), axis=2)],
     axis=2)
  logits = tf.einsum("bhk,bmk->bhm", q, K)
  weights = tf.softmax(logits)
  o = tf.einsum("bhm,bmv->bhv", weights, V)
  y = tf.einsum("bhv,hdv->bd", O, P_o)
  return y, K, V
```

Dividing the memory by the computations, we find that the ratio of memory access to arithmetic operations is $\Theta(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b})$. 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)

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

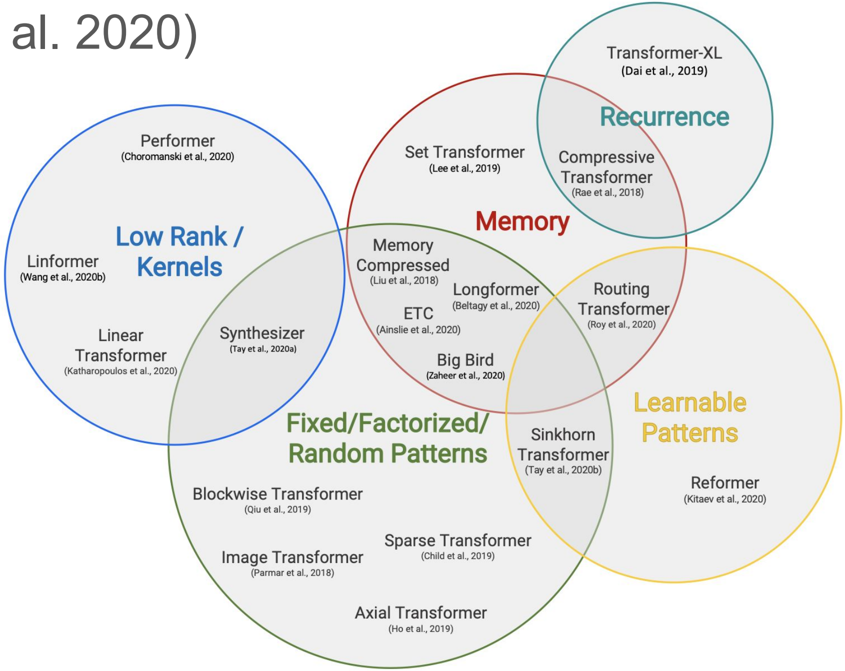
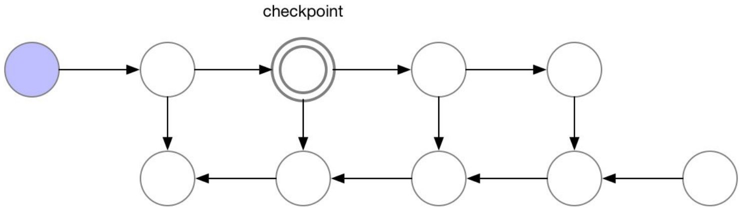


Figure 2: Taxonomy of Efficient Transformer Architectures.

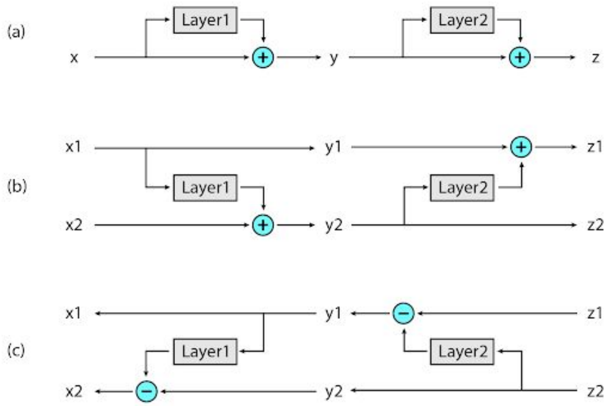
High-level optimizations – During pre-training

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

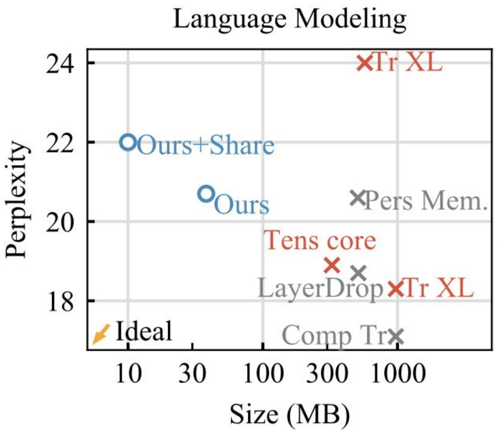


- Fully invertible networks
  - Reformer (Kitaev et al. 2020)

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)



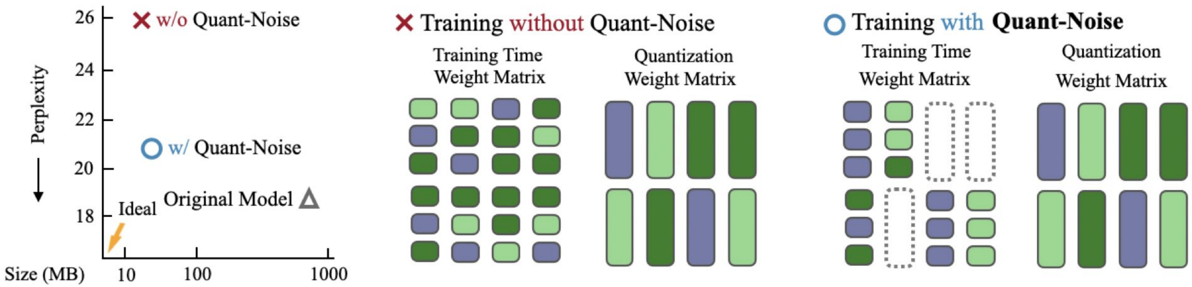
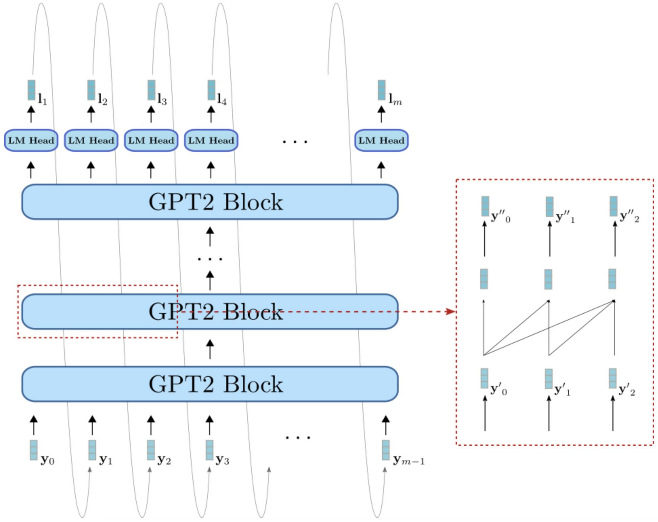


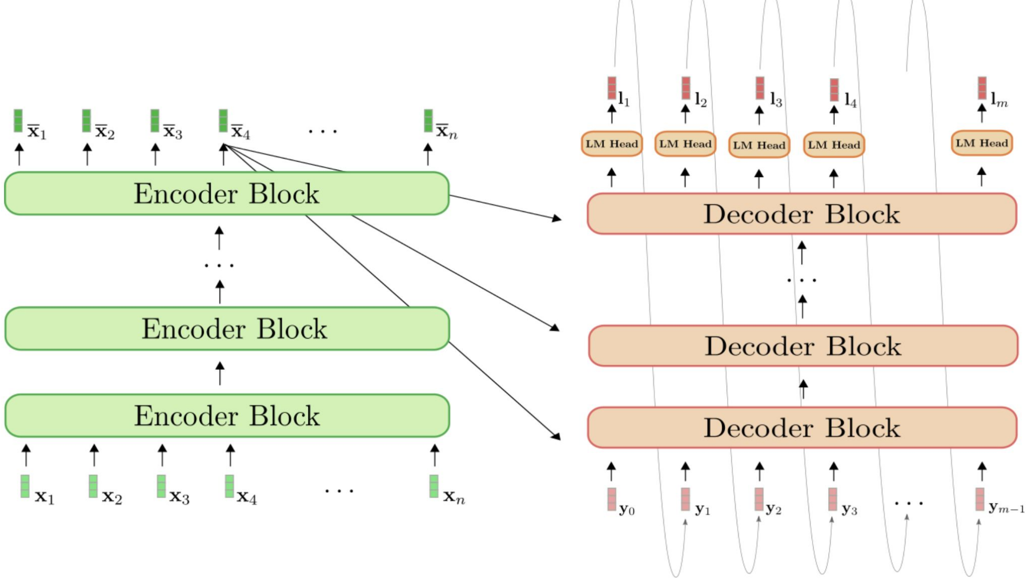
Figure 1: **Quant-Noise** trains models to be resilient to inference-time quantization by mimicking the effect of the quantization method during training time. This allows for extreme compression rates without much loss in accuracy on a variety of tasks and benchmarks.

	Language modeling		
	Comp.	Size	PPL
Unquantized models			
Original model	× 1	942	18.3
+ Sharing	× 1.8	510	18.7
+ Pruning	× 3.7	255	22.5
Quantized models			
iPQ	× 24.8	38	25.2
+ Quant-Noise	× 24.8	38	20.7
+ Sharing	× 49.5	19	22.0
+ Pruning	× 94.2	10	24.7

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…)





High-level optimizations – After training

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

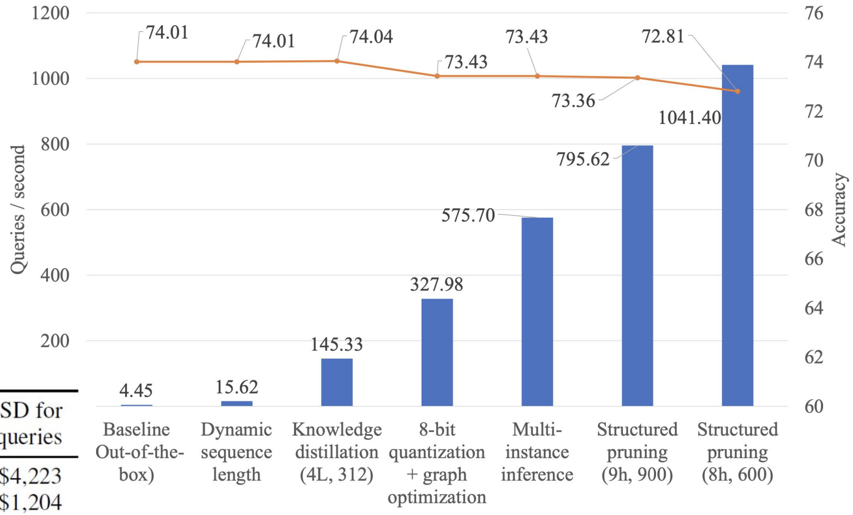


Figure 3: Accuracy versus queries per second with various optimizations on CPU.

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

Teacher	Student	MM Params	Time (MS)	Speedup	Rouge-2	Rouge-L
BART	12-1	222	743	2.35	17.98	33.31
	12-3	255	905	1.93	**22.40**	37.30
	6-6	230	1179	1.48	21.17	36.21
	9-6	268	1184	1.47	22.08	37.24
	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
	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 6: Best XSUM results across all methods. Each sub-table is sorted fastest to slowest by inference time. `dBART-12-3` and `dPegasus-16-4` are trained on Pegasus pseudo-labels. `dBART-12-6`, `dBART-6-6`, and `dBART-9-6` are trained with KD. `dPegasus-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.

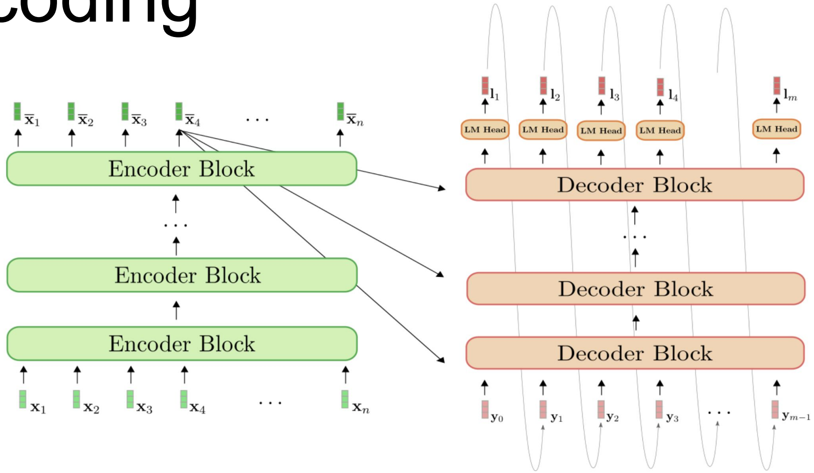
Teacher	Size	Data	Teacher	SFT		KD		Pseudo	
			Score	Score	Cost	Score	Cost	Score	Cost
BART †	12-3	XSUM	22.29	21.08	2.5	**21.63**	6	21.38	15
Pegasus	16-4	XSUM	24.56	22.64	13	21.92	22	**23.18**	34
BART	12-6	CNN	21.06	**21.21**	2	20.95	14	19.93	19.5
Pegasus	16-4	CNN	21.37	**21.29**	31	-	-	20.1	48
Marian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

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.

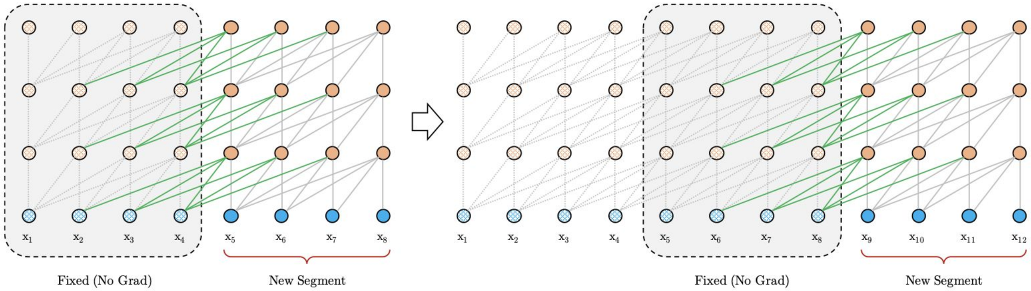
S Shleifer, A M. Rush 2020 – Pre-trained Summarization Distillation

High-level optimizations – Decoding

- Optimizing **Decoding algorithm:**
  - **Caching**
    - Caching encoder

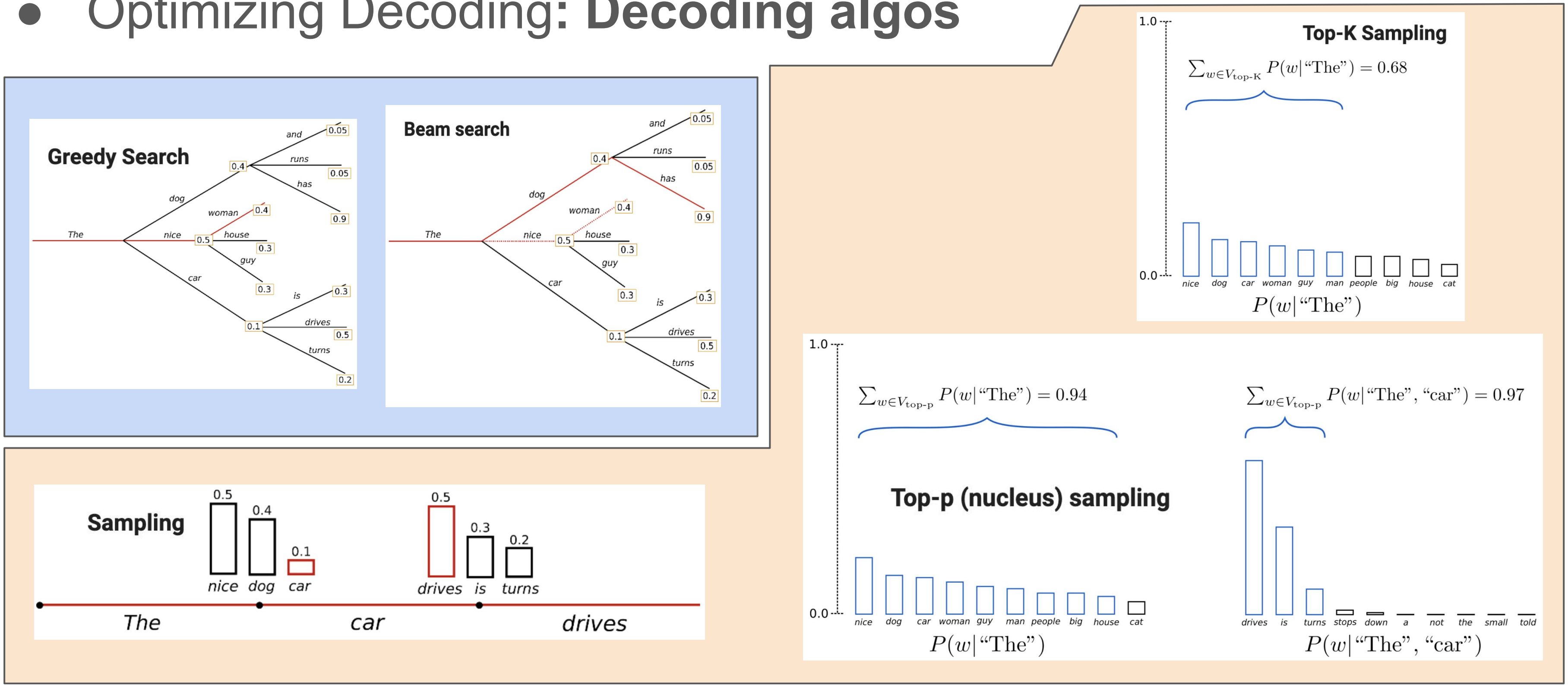


    - Caching decoding
      Transformer-XL ([Dai et al. 2019](#))



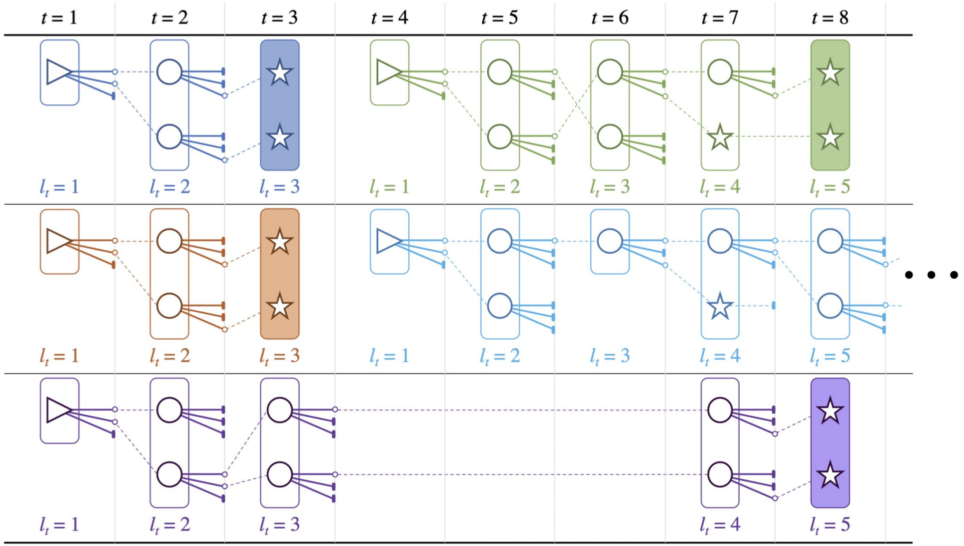
High-level optimizations – Decoding

- Optimizing Decoding: **Decoding algos**



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

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

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

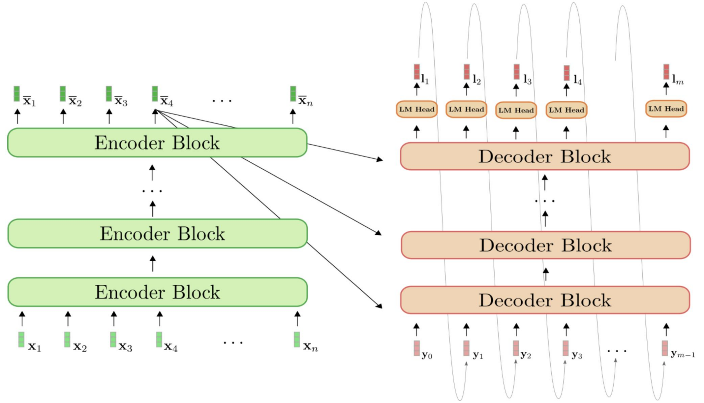
Young Jin Kim et al. 2020

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**

	Gigaword			CNN/Dailymail			BBC XSum		
	R-1	R-2	R-L	R-1	R-2	R-L	R-1	R-2	R-L
Lead	–	–	–	39.60	17.70	36.20	16.30	1.61	11.95
PtGen	–	–	–	39.53	17.28	36.38	29.70	9.21	23.24
ConvS2S	35.88	17.48	33.29	–	–	–	31.89	11.54	25.75
MMN	–	–	–	–	–	–	32.00	12.10	26.00
Bottom-Up	–	–	–	41.22	18.68	38.34	–	–	–
MASS	*38.73*	19.71	*35.96*	–	–	–	–	–	–
TransLM	–	–	–	39.65	17.74	36.85	–	–	–
UniLM	–	–	–	*43.47*	*20.30*	*40.63*	–	–	–
Initialized with the base checkpoint (12 layers)									
RND2RND	36.94	18.71	34.45	35.77	14.00	32.96	30.90	10.23	24.24
BERT2RND	37.71	19.26	35.26	38.74	17.76	35.95	38.42	15.83	30.80
RND2BERT	37.01	18.91	34.51	36.65	15.55	33.97	32.44	11.52	25.65
BERT2BERT	38.01	19.68	35.58	39.02	17.84	36.29	37.53	15.24	30.05
BERTSHARE	38.13	***19.81***	**35.62**	39.09	18.10	36.33	38.52	16.12	31.13
ROBERTASHARE	**38.21**	19.70	35.44	**40.10**	**18.95**	**37.39**	**39.87**	**17.50**	**32.37**
GPT	36.04	18.44	33.67	37.26	15.83	34.47	22.21	4.89	16.69
RND2GPT	36.21	18.39	33.83	32.08	8.81	29.03	28.48	8.77	22.30
BERT2GPT	36.77	18.23	34.24	25.20	4.96	22.99	27.79	8.37	21.91
ROBERTA2GPT	37.94	19.21	35.42	36.35	14.72	33.79	19.91	5.20	15.88
Initialized with the large checkpoint (24 layers)									
BERTSHARE	38.35	19.80	35.66	39.83	17.69	37.01	38.93	16.35	31.52
ROBERTASHARE	**38.62**	19.78	**35.94**	**40.31**	18.91	**37.62**	***41.45***	***18.79***	***33.90***

	total	embed.	init.	random
RND2RND	221M	23M	0	221M
BERT2RND	221M	23M	109M	112M
RND2BERT	221M	23M	109M	26M
BERT2BERT	221M	23M	195M	26M
BERTSHARE	136M	23M	109M	26M
ROBERTASHARE	152M	39M	125M	26M
GPT	125M	39M	125M	0
RND2GPT	238M	39M	125M	114M
BERT2GPT	260M	62M	234M	26M
ROBERTA2GPT	276M	78M	250M	26M

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.

Load the CNN/DailyMail dataset

- Companion Notebook by [Patrick van Platen](#) is here: https://colab.research.google.com/drive/1WIk2bxglElfZewOHboPFNj8H44_VAyKE

```
!pip install datasets==1.0.2
import datasets

train_data = datasets.load_dataset("cnn_dailymail", "3.0.0", split="train")
```

```
from pprint import pprint
print(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

{'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 le
            'because he wants to. "While I believe I have the authority to carry out this military action without specific congressional autho
            'used -- and not by whom," U.N. spokesman Martin Nesirky told reporters on Saturday. But who used the weapons in the reported toxi
            'evidence they collected. "It needs time to be able to analyze the information and the samples," Nesirky said. He noted that Ban ha
            'would not be open-ended or include U.S. ground forces, he said. Syria\'s alleged use of chemical weapons earlier this month "is ar
            'before his Rose Garden speech. "The two leaders agreed that the international community must deliver a resolute message to the As
            'Boehner, Majority Leader Eric Cantor, Majority Whip Kevin McCarthy and Conference Chair Cathy McMorris Rodgers issued a statement
            'theory. "The main reason Obama is turning to the Congress:  the military operation did not get enough support either in the world
            'military levels. Syria\'s prime minister appeared unfazed by the saber-rattling. "The Syrian Army\'s status is on maximum readine
            'that it used chemical weapons in the August 21 attack, saying that jihadists fighting with the rebels used them in an effort to tu
 'highlights': 'Syrian official: Obama climbed to the top of the tree, "doesn\'t know how to get down"\nObama sends a letter to the heads of th
 'id': '0001d1afc246a7964130f43ae940af6bc6c57f01'}
```

Prepare the dataset (tokenize it)

```
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
encoder_max_length=512
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

```
[32] 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.

```
bert2bert = EncoderDecoderModel.from_encoder_decoder_pretrained("bert-base-uncased", "bert-base-uncased")
```

Downloading: 100% 433/433 [00:02<00:00, 190B/s]

Downloading: 100% 440M/440M [00:07<00:00, 60.5MB/s]

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertLMHeadModel: ['cls.seq_relationship.weight', 'cls
- 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"])["rouge2"].mid

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

```python
# 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.

```
[ ] # 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
```

```
[ ] # 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)
/usr/local/lib/python3.6/dist-packages/torch/optim/lr_scheduler.py:123: UserWarning: Detected call of `lr_scheduler.step()` before `optimizer.s
  "https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate", UserWarning)
```

[16/16 08:29, Epoch 2/2]

Step	Training Loss	Validation Loss	Rouge2 Precision	Rouge2 Recall	Rouge2 Fmeasure
4	10.000679	10.383123	0.000000	0.000000	0.000000
8	8.465530	8.130023	0.004300	0.004800	0.004500
12	7.704124	7.786637	0.005100	0.003900	0.004400
16	7.525826	7.755284	0.000000	0.000000	0.000000

```
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"])
```

100% 1/1 [01:56<00:00, 116.48s/ba]

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.