CS 6501 Natural Language Processing

Transformers, BERT, and GPT

Yangfeng Ji

Information and Language Processing Lab
Department of Computer Science
University of Virginia
https://uvanlp.org/

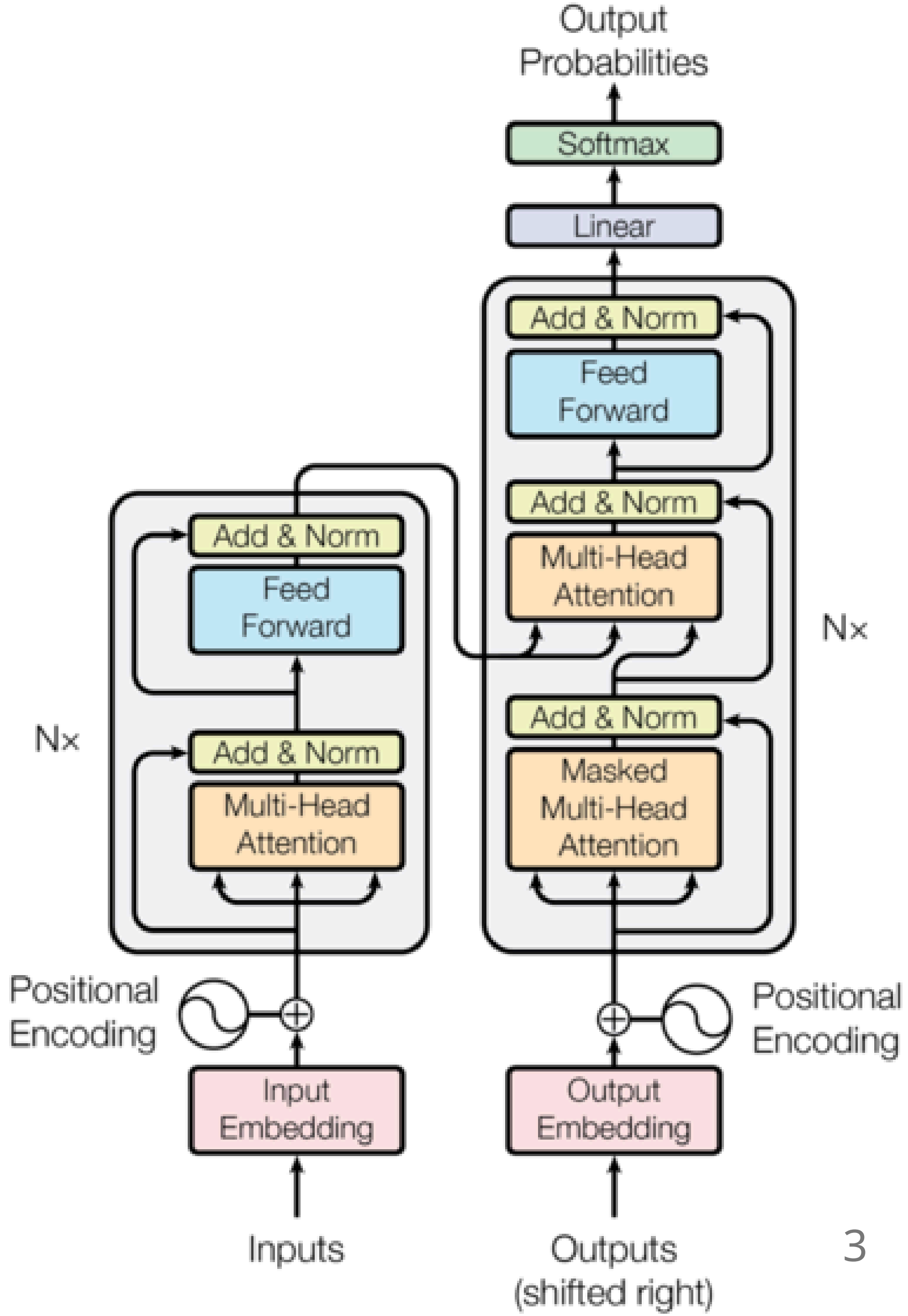
Section I

Transformer in Detail

Based on the Annotated Transformer from the Harvard NLP group.

Overview

Goal: explain every connection in this figure



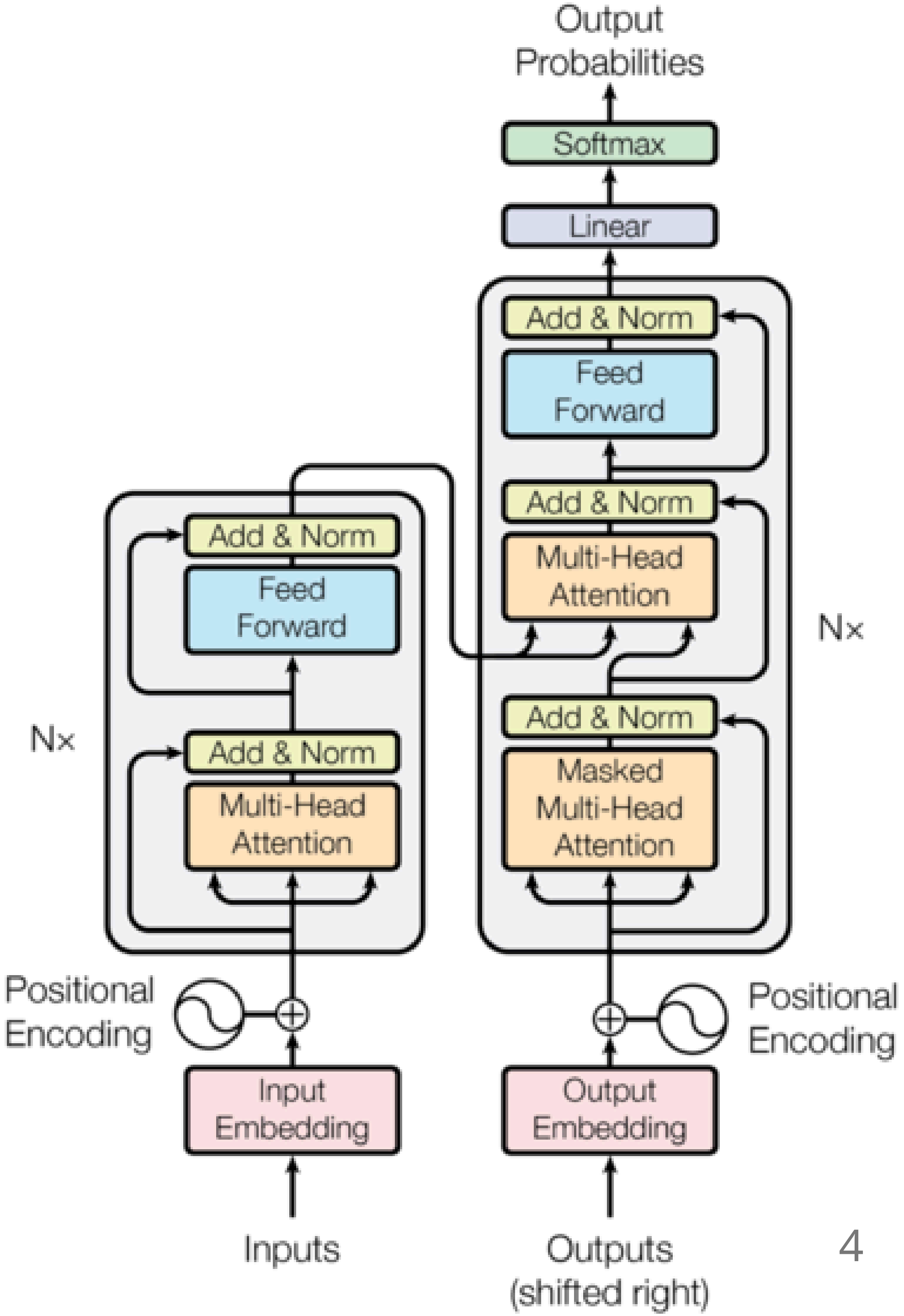
Building Blocks

Two sub-layers

- Multi-head attention layer
- Feed-forward layer

Two additional building blocks

- Layer normalization
- Residual connection



Single-head Self-attention

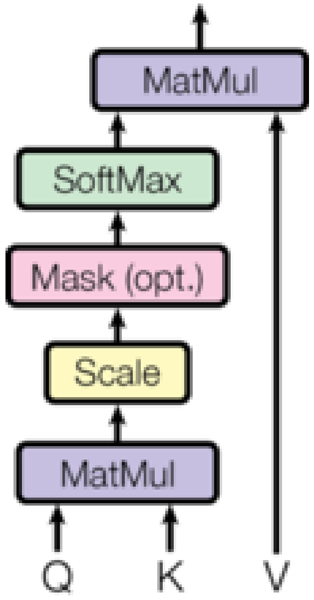
or just self-attention

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

- Q: query matrix
- K: key matrix
- V: value matrix
- d_k: the dimension of query (and key)

Computational Graph

Using query and key to compute the attention weights, and then select the corresponding values



This attention mechanism is also called **Scaled Dot-Product Attention**

Implementation

The implementation of $\mathrm{Attention}(Q, K, V)$

```python
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

No parameter involved so far

Multi-head Attention

With H heads

- For $i = 1, \ldots, H$
  - Compute

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- Concatenate multiple heads as

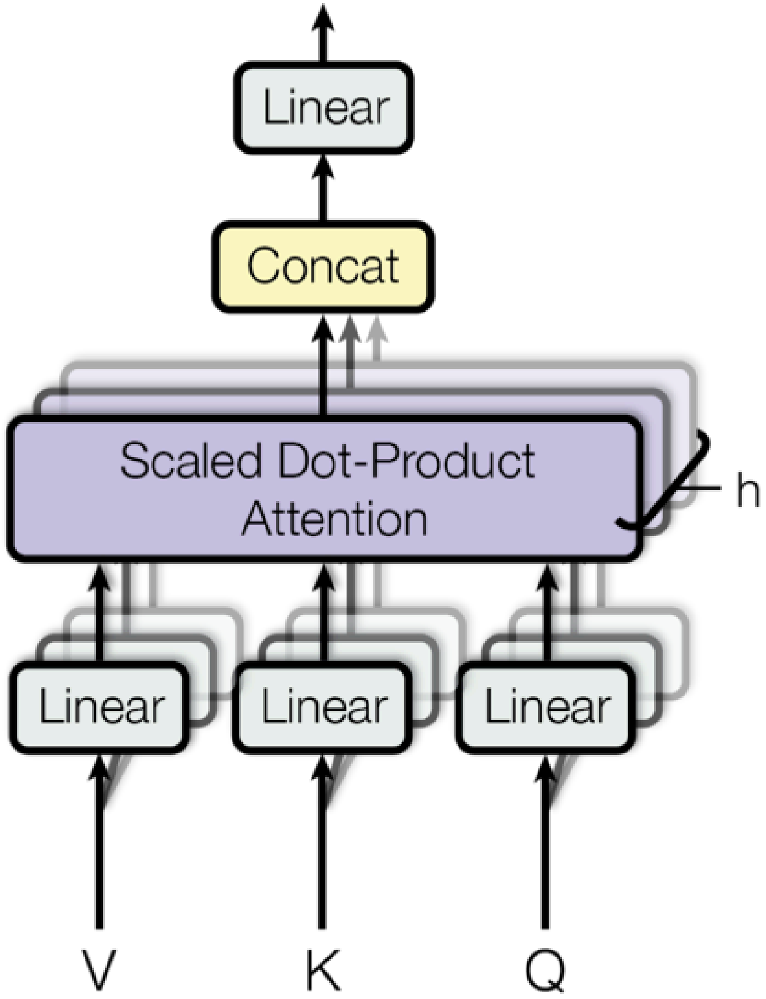
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W_O$$

Parameters

$$\{W_i^Q, W_i^K, W_i^V\}_{i=1}^H, W^O$$

Illustration

The central component is the **Scaled Dot-Product Attention**



Implementation

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
                self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)

    def forward(self, query, key, value, mask=None):
        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [
            lin(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
            for lin, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(
            query, key, value, mask=mask, dropout=self.dropout)
```

`clones()` creates 4 deep copies of `nn.Linear`

Feed-forward Network

Another sub-layer in the Transformer encoder module

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Parameters

$$W_1, b_1, W_2, b_2$$

Implementation

```python
class PositionwiseFeedForward(nn.Module):

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(self.w_1(x).relu()))
```

Layer Normalization

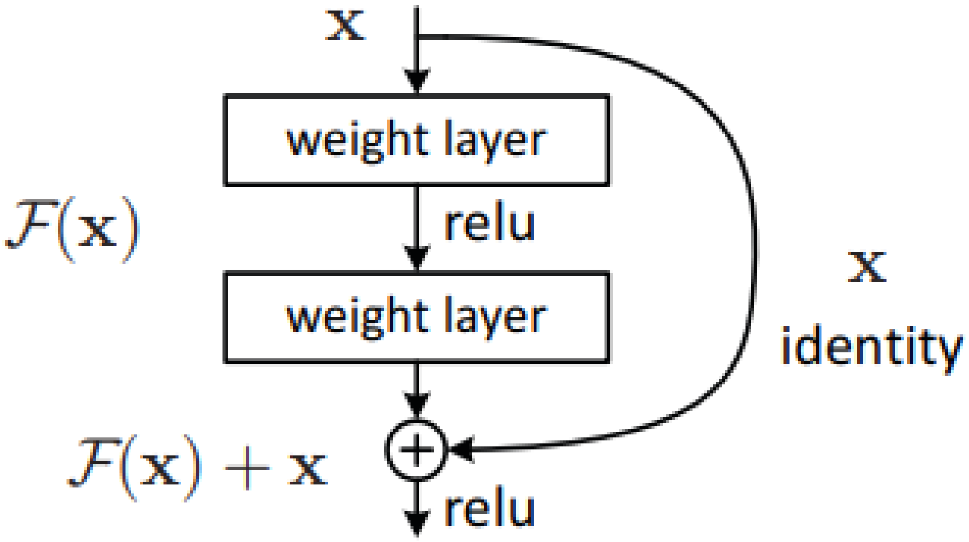
```
class LayerNorm(nn.Module):

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

(Ba et al., 2016)

Residual Connection

Residual connection from prior work

$$x \to \mathcal{F}(x) + x$$



(He et al., 2016)

Implementation

This is applied to both the multi-head attention and the feed-forward modules

```
class SublayerConnection(nn.Module):

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

Encoder Layer

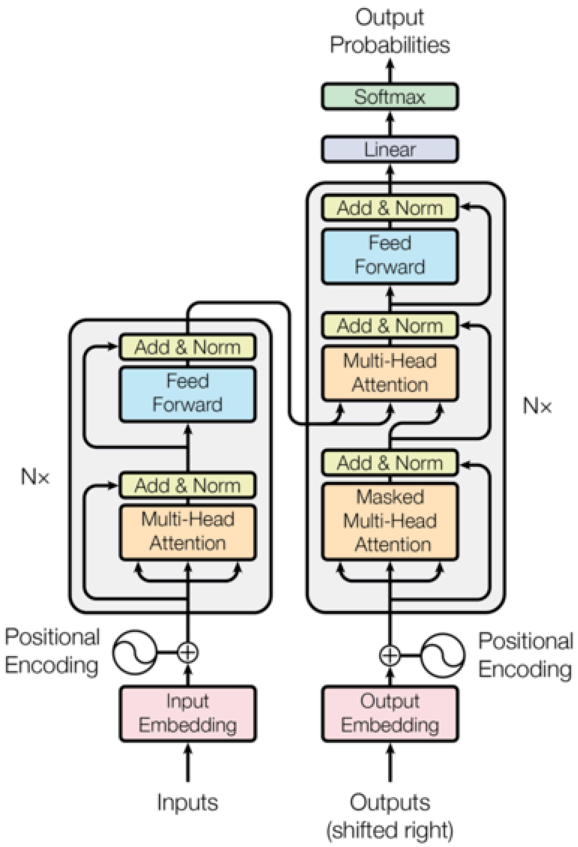
```python
class EncoderLayer(nn.Module):

    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Decoder Layer

```python
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"

    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

Review

The Transformer architecture



Final Model

```
def make_model(
    src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1
):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab),
    )
```

What Else?

- Tokenization
- Word embeddings
- Positional embeddings

Section II

BERT

(Devlin et al., 2018)

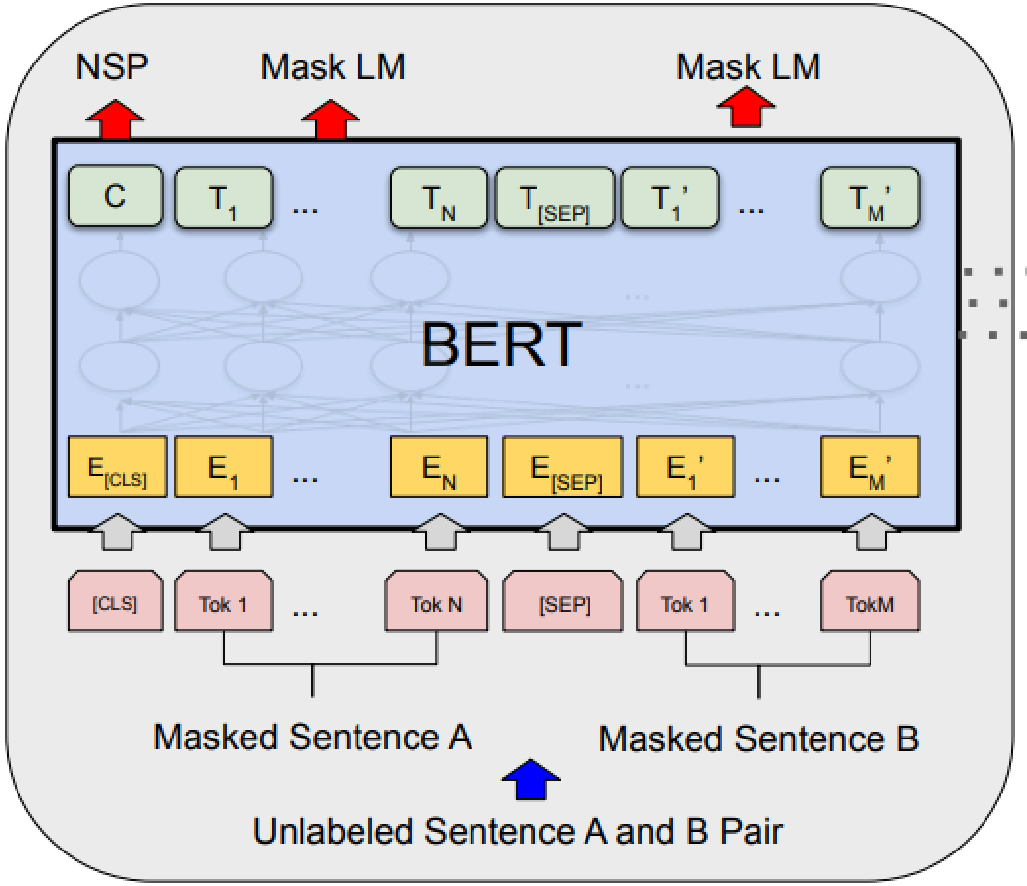
Pre-training

Using the Transformer encoder that we discussed in the previous work

By default, the Transformer will read the context from both sides, unless there is a particularly designed mask

Input pattern

```
[CLS] sentence-A [SEP] sentence-B [SEP]
```



Wordpiece Tokenization

Tokenization example

Input

```
Jet makers feud over seat width
```

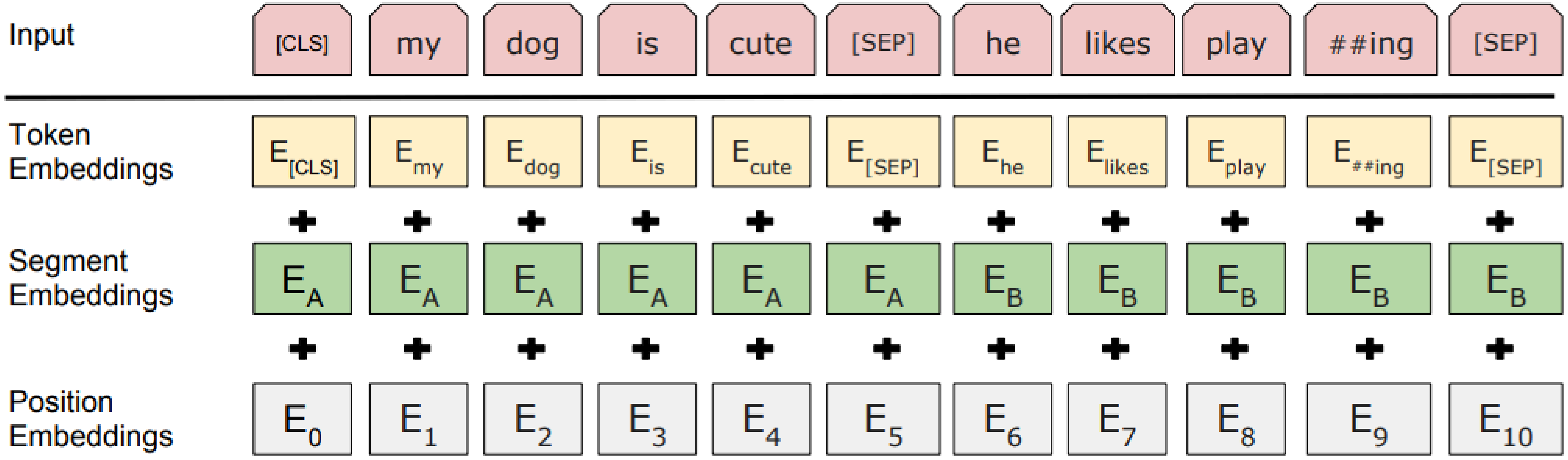
Output:

```
['jet', 'makers', '##s', 'feud', 'over', 'seat', 'width', '.']
```

At decoding time, the model first produces a wordpiece sequence, and then converts them into the corresponding word sequence.

(Wu et al., 2016)

Input Representation



The input embeddings are the sum of the token embeddings, the sementation embeddings, and the position embeddings.

Masked Language Model

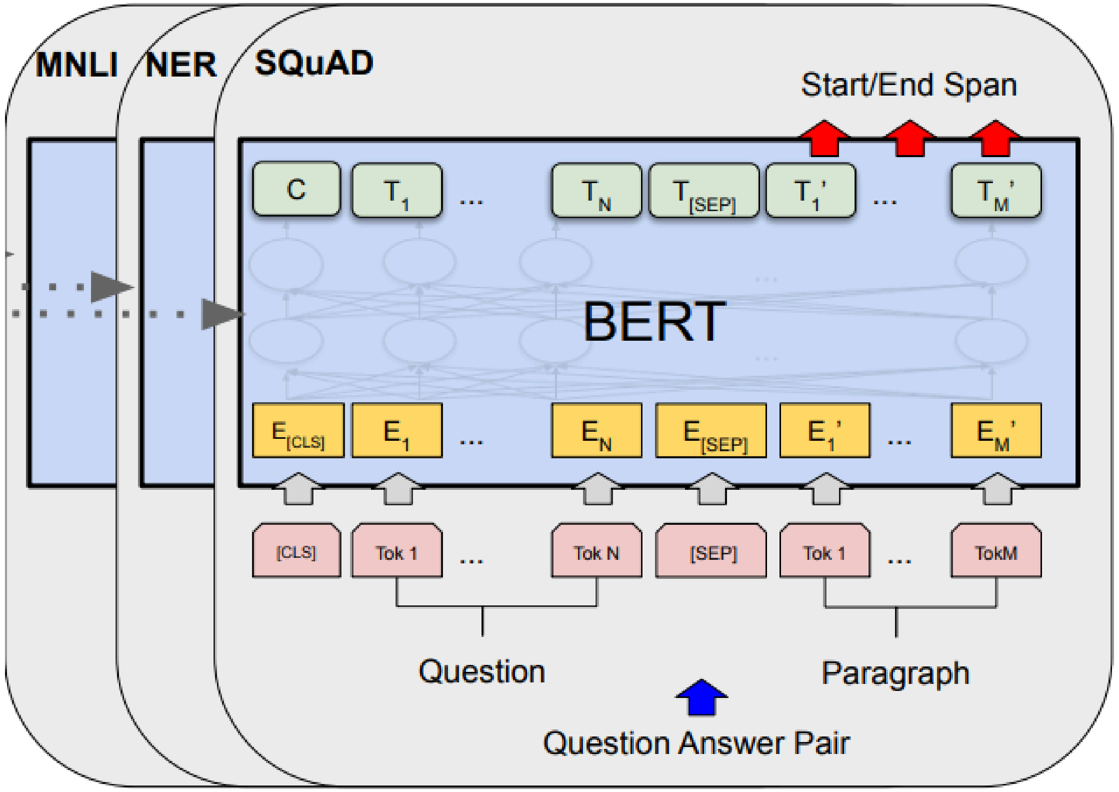
During pre-training, randomly mask some words in the text and ask the LM to predict them

```
I love drinking [MASK] coffee .
```

- 15% of tokens are masked
  - 80% of them are replaced by `[MASK]`
  - 10% of them are replaced by randomly selected tokens
  - 10% of them are left as is

Fine-tuning

To preform different tasks, BERTs are trained with different heads

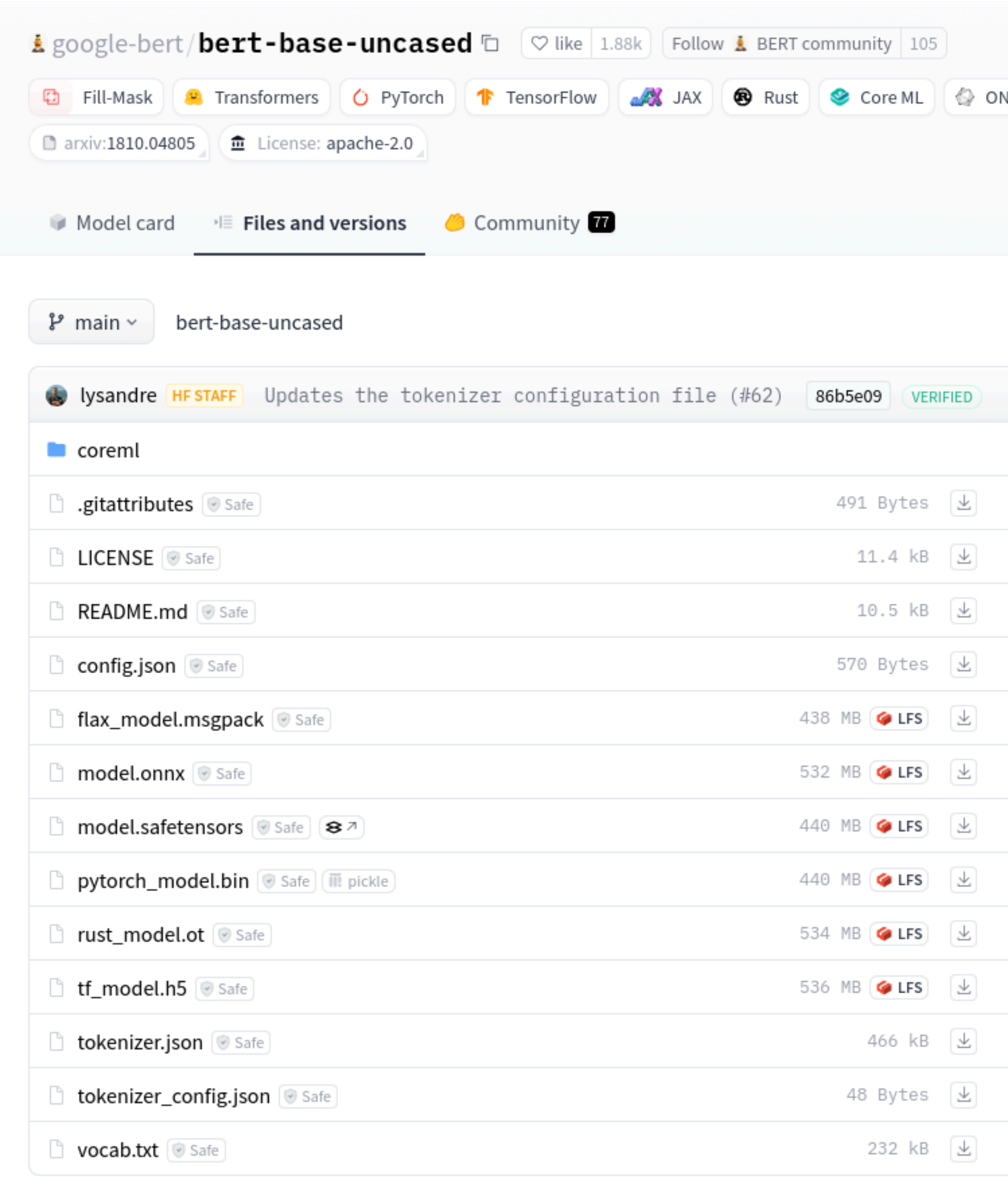


For more information, please refer to this Hugging Face page

Some Implementation Details

More about model configuration and tokenization.

A simple demo



Section III

The GPT Family

The GPT Family

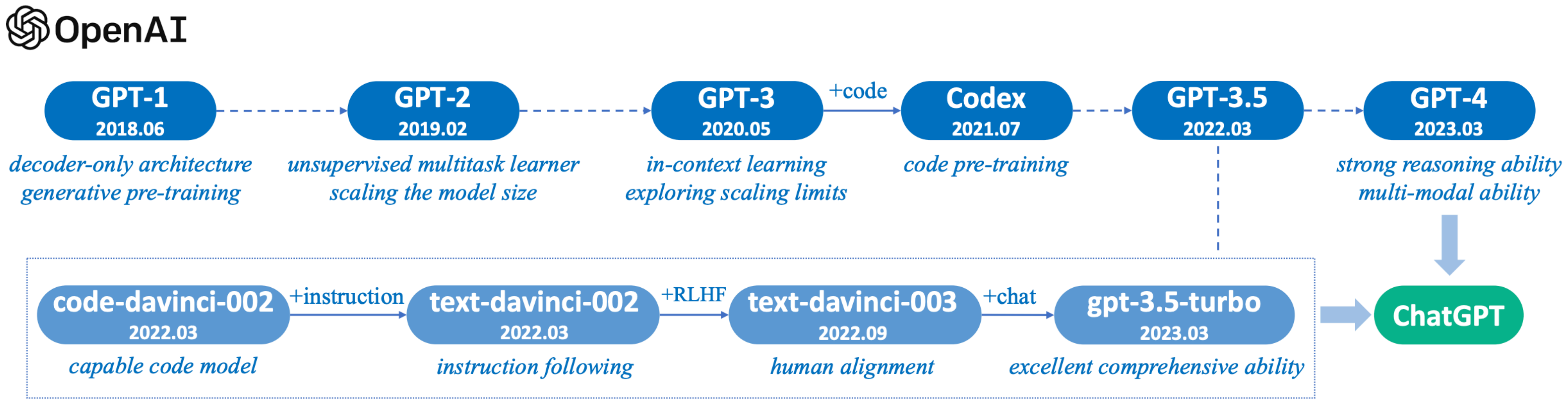


Fig. 3: A brief illustration for the technical evolution of GPT-series models. We plot this figure mainly based on the papers, blog articles and official APIs from OpenAI. Here, *solid lines* denote that there exists an explicit evidence (*e.g.*, the official statement that a new model is developed based on a base model) on the evolution path between two models, while *dashed lines* denote a relatively weaker evolution relation.

GPT-1: Conceptual Idea

One model for multiple tasks

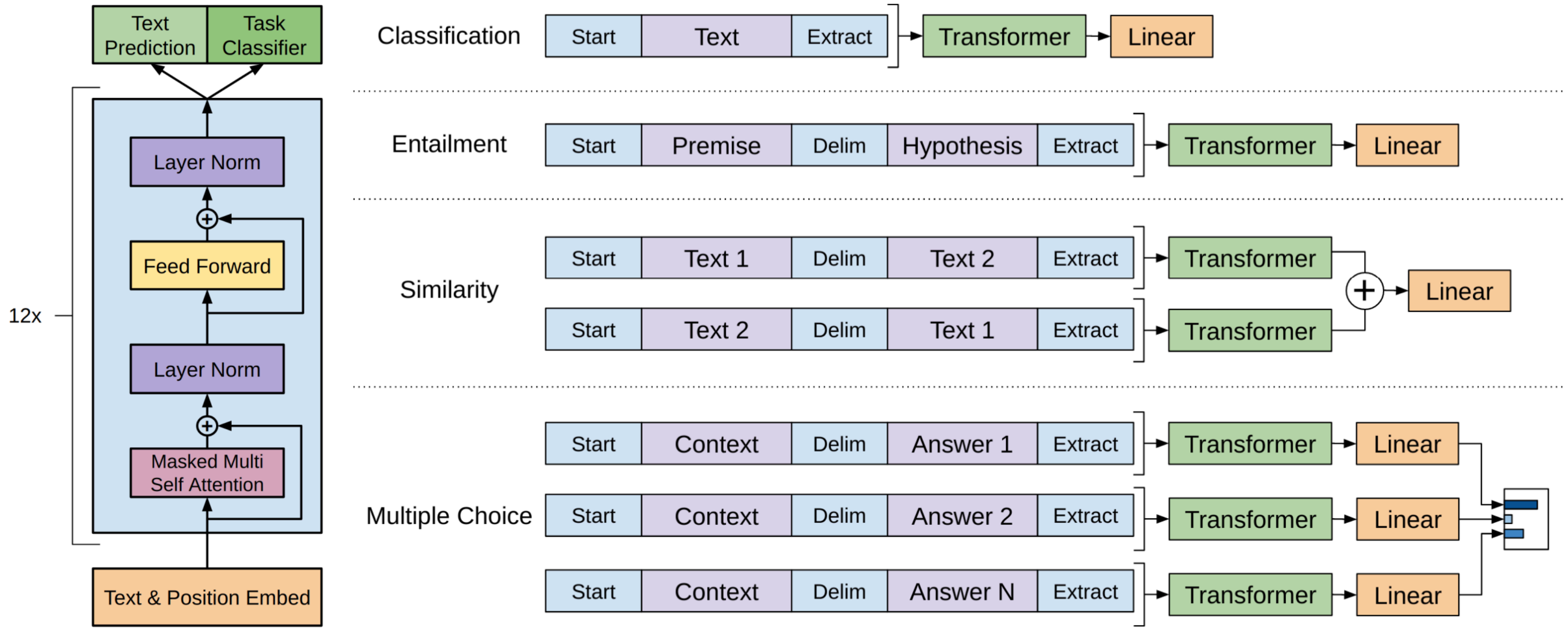


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

12-layer Transformer decoder (Radford et al., 2018)

GPT-1: Training Strategies

- Unsupervised pre-training: given an unsupervised corpus of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-1}, \ldots, u_{i-k})$$

- Supervised fine-tuning: given $\mathcal{C} = \{(x^1, \ldots, x^m, y)\}$ where $x^1, \ldots, x^m$ is the input sequence and y is the label

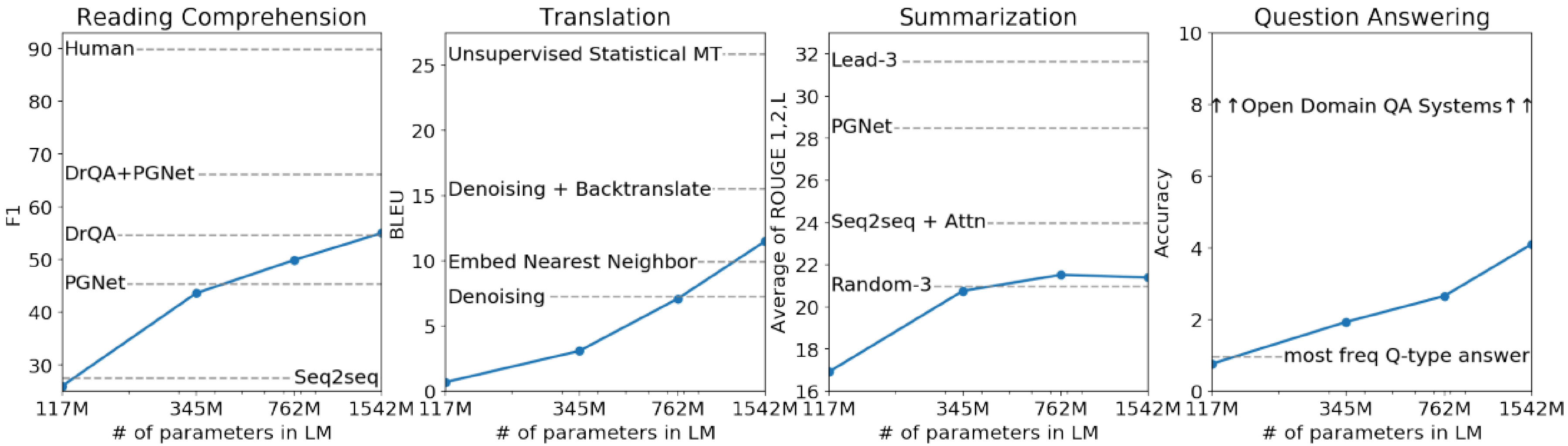
$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y | x^1, \ldots, x^m)$$

- Fine-tuning works better when using L_1 as an aux task

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda L_1(\mathcal{C})$$

GPT-2: Zero-shot Multi-task Learner

Zero-shot task performance of WebText LMs as a function of model size on four NLP tasks



(Radford et al., 2019)

Byte Pair Encoding

Idea: iteratively merge the most frequent character pairs in the sequences

Given the following four words: `low`, `lowest`, `newer`, `wider`

- Create the character sequence
  - `low` → `l o w </w>`
  - `lowest` → `l o w e s t </w>`
  - `newer` → `n e w e r </w>`
  - `wider` → `w i d e r </w>`

(Sennrich et al., 2015)

Byte Pair Encoding (II)

Initial vocab

`l o w e s t n r i d`

- First step of merge operation: `l o` → `lo`
  - `low` → `lo w </w>`
  - `lowest` → `lo w e s t </w>`
  - `newer` → `n e w e r </w>`
  - `wider` → `w i d e r </w>`
- The vocab after the first merge operation: `l o w e s t n r i d lo`

Link

GPT-2: Results (Positive)

	LAMBADA (PPL)	LAMBADA (ACC)	CBT-CN (ACC)	CBT-NE (ACC)	WikiText2 (PPL)	PTB (PPL)	enwik8 (BPB)	text8 (BPC)	WikiText103 (PPL)	1BW (PPL)
SOTA	99.8	59.23	85.7	82.3	39.14	46.54	0.99	1.08	18.3	**21.8**
117M	**35.13**	45.99	**87.65**	**83.4**	**29.41**	65.85	1.16	1.17	37.50	75.20
345M	**15.60**	55.48	**92.35**	**87.1**	**22.76**	47.33	1.01	**1.06**	26.37	55.72
762M	**10.87**	**60.12**	**93.45**	**88.0**	**19.93**	**40.31**	**0.97**	**1.02**	22.05	44.575
1542M	**8.63**	**63.24**	**93.30**	**89.05**	**18.34**	**35.76**	**0.93**	**0.98**	**17.48**	42.16

Table 3. Zero-shot results on many datasets. No training or fine-tuning was performed for any of these results. PTB and WikiText-2 results are from (Gong et al., 2018). CBT results are from (Bajgar et al., 2016). LAMBADA accuracy result is from (Hoang et al., 2018) and LAMBADA perplexity result is from (Grave et al., 2016). Other results are from (Dai et al., 2019).

GPT-2: Results (Negative)

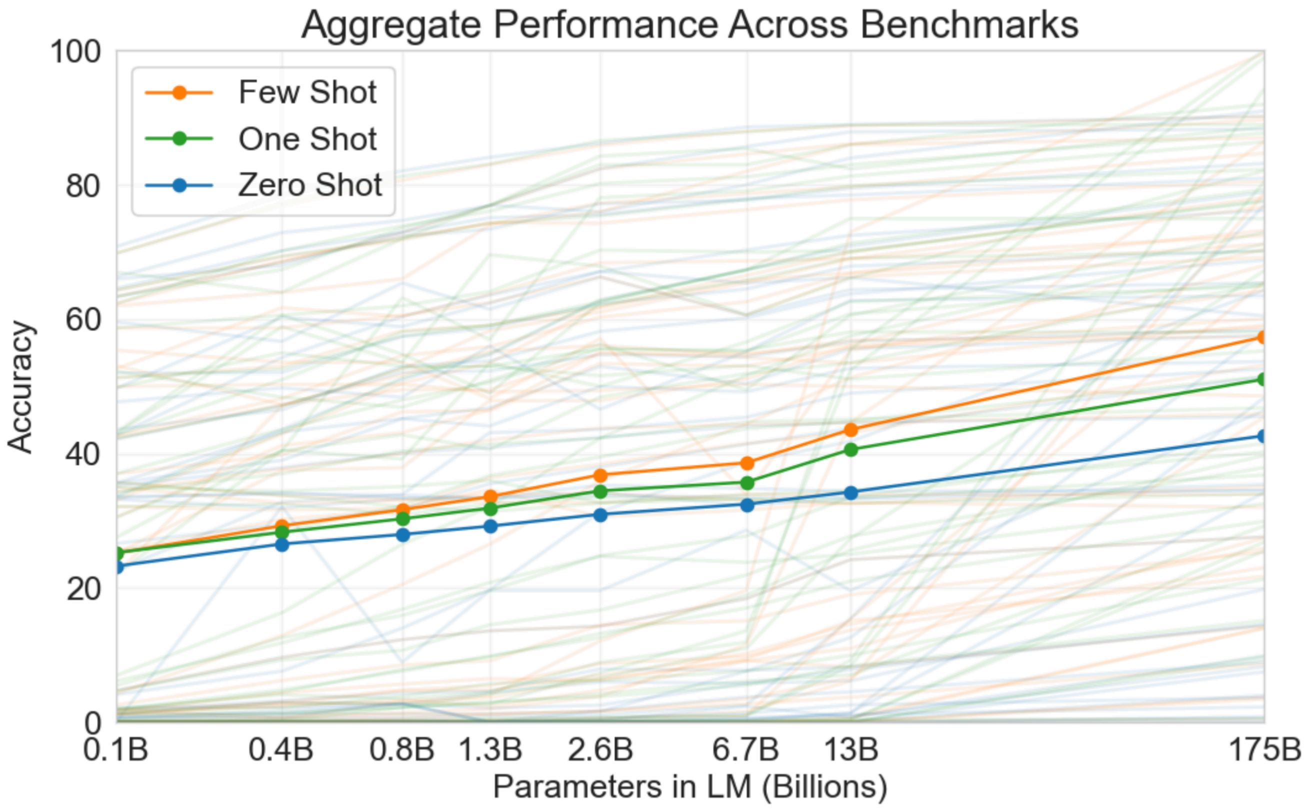
Document summarization is a difficult task

	R-1	R-2	R-L	R-AVG
Bottom-Up Sum	**41.22**	**18.68**	**38.34**	**32.75**
Lede-3	40.38	17.66	36.62	31.55
Seq2Seq + Attn	31.33	11.81	28.83	23.99
GPT-2 `TL;DR:`	29.34	8.27	26.58	21.40
Random-3	28.78	8.63	25.52	20.98
GPT-2 no hint	21.58	4.03	19.47	15.03

Table 4. Summarization performance as measured by ROUGE F1 metrics on the CNN and Daily Mail dataset. Bottom-Up Sum is the SOTA model from (Gehrmann et al., 2018)

GPT-3: LMs as Few-shot Learners

The performance of GPT-3 on few-shot in-context learning



Larger models produce better performance

GPT-3: Specification

The specifications of GPT-3 and some small models compared in (Brown et al., 2021)

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

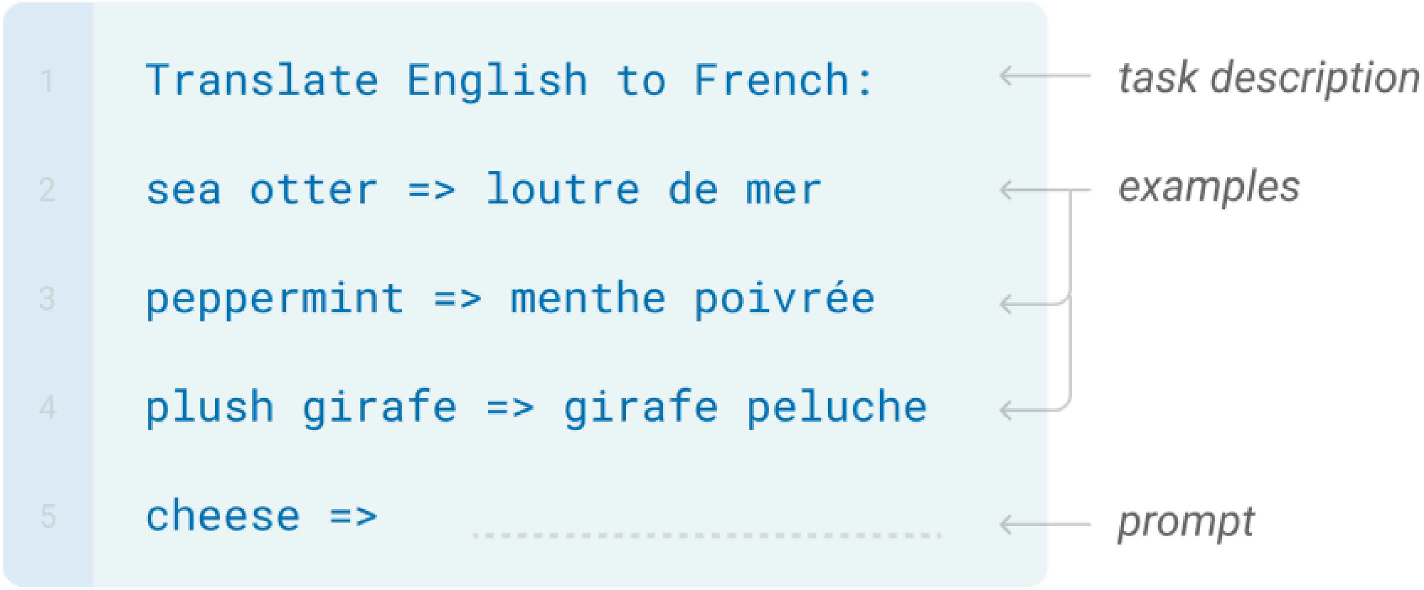
GPT-3: Datasets

The datasets used for GPT-3 pre-training

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

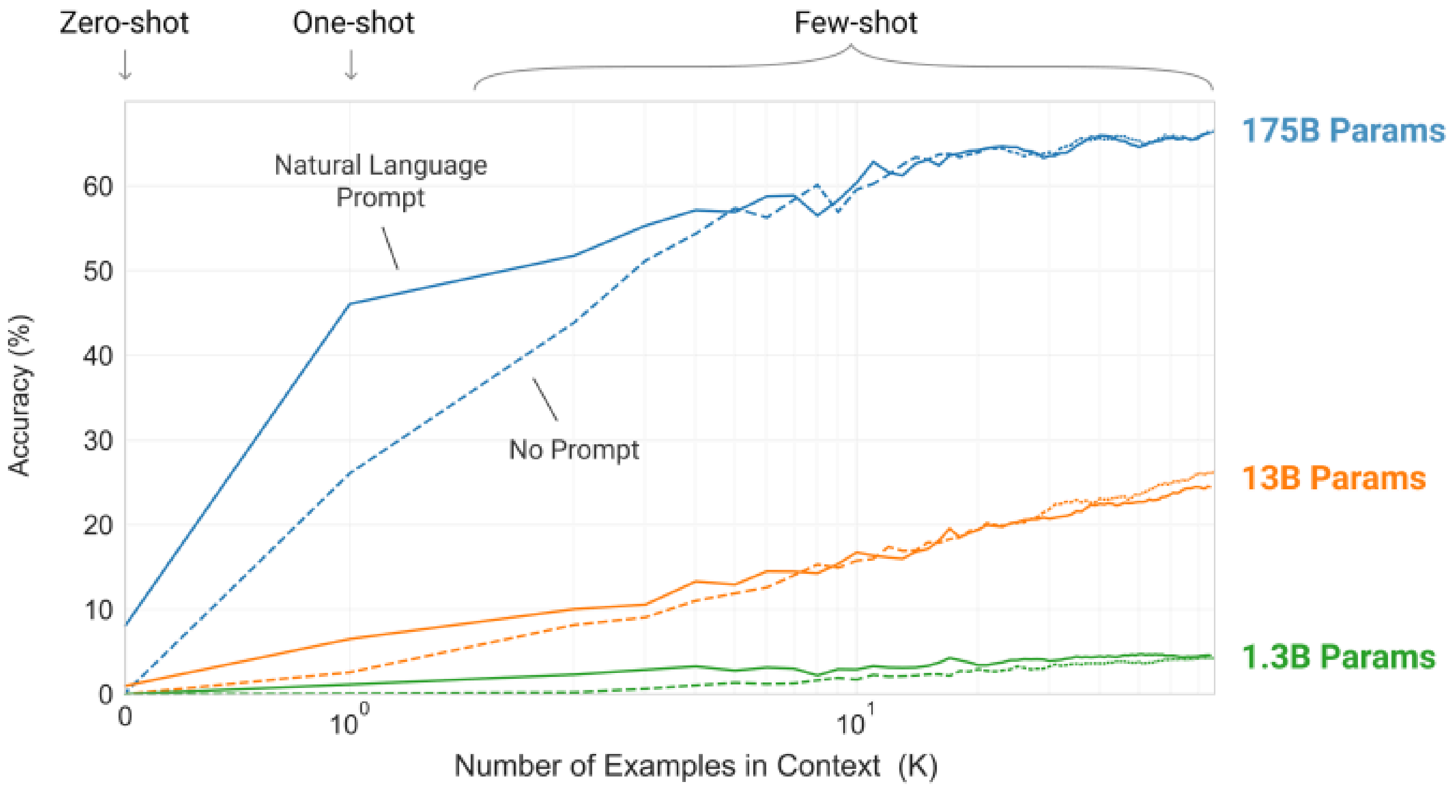
GPT-3: In-Context Learning

Adding a few examples to the input context for demonstration. For example



GPT-3: ICL Performance

In general, in-context learning works with larger models and more examples



GPT-3: Negative Results

Some *negative* results of GPT-3

Setting	ARC (Easy)	ARC (Challenge)	CoQA	DROP
Fine-tuned SOTA	**92.0**[a]	**78.5**[b]	**90.7**[c]	**89.1**[d]
GPT-3 Zero-Shot	68.8	51.4	81.5	23.6
GPT-3 One-Shot	71.2	53.2	84.0	34.3
GPT-3 Few-Shot	70.1	51.5	85.0	36.5

Table 3.3: GPT-3 results on a selection of QA / RC tasks. CoQA and DROP are F1 while ARC reports accuracy. See the appendix for additional experiments. [a][KKS+20] [b][KKS+20] [c][JZC+19] [d][JN20]

- ARC: Question-answering dataset, containing questions from science exams from grade 3 to grade 9
- CoQA: A Conversational Question Answering Challenge
- DROP: A Reading Comprehension Benchmark Requiring Discrete

Thank You!