CS 6770 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

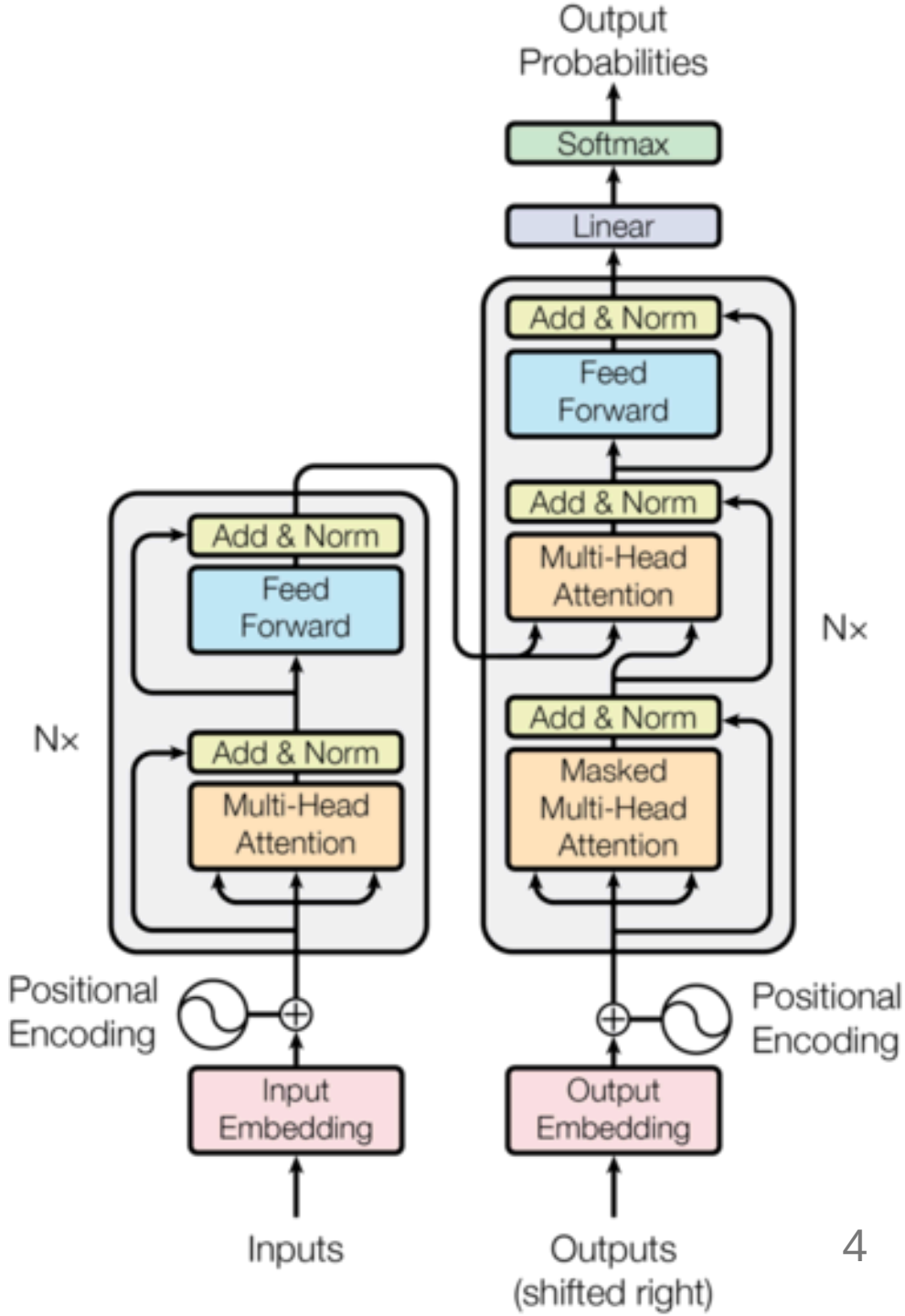
Based on the Annotated Transformer from the Harvard NLP group.

RNN LMs vs. Transformer-based LMs

The structrual difference

Transformer Module Overview

Goal: explain every connection in this figure



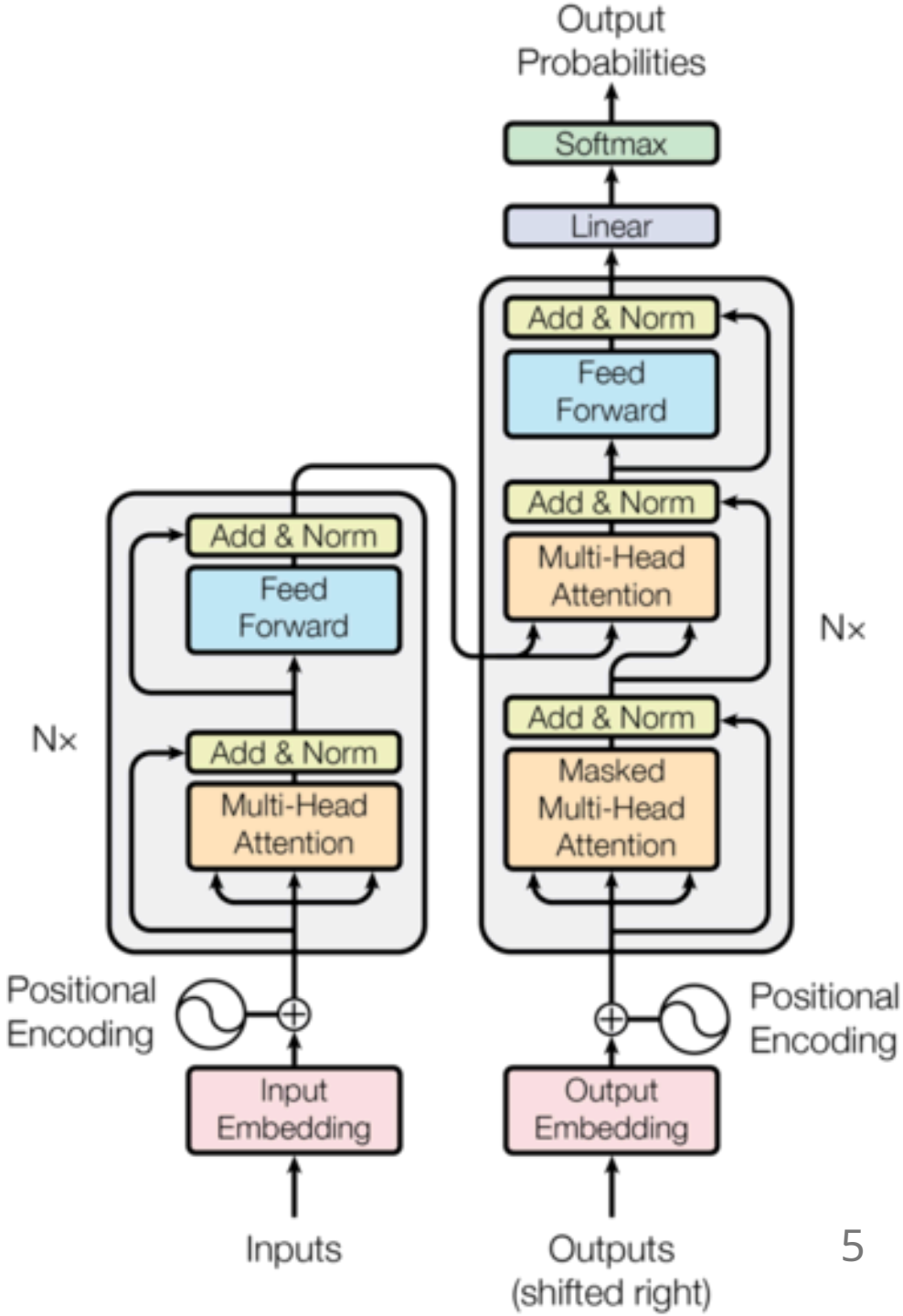
Building Blocks

Two sub-layers

- Multi-head attention layer
- Feed-forward layer

Two additional building blocks

- Layer normalization
- Residual connection



Single-head Self-attention

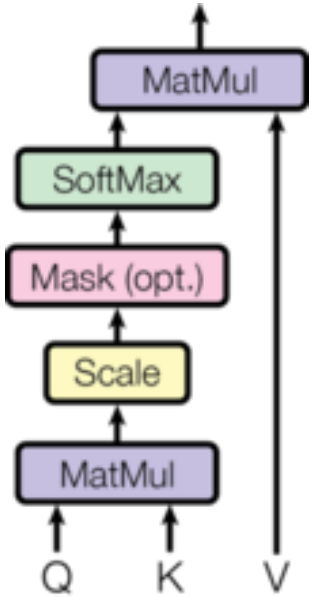
or just self-attention

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

- 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 $\text{Attention}(Q, K, V)$

```
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

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

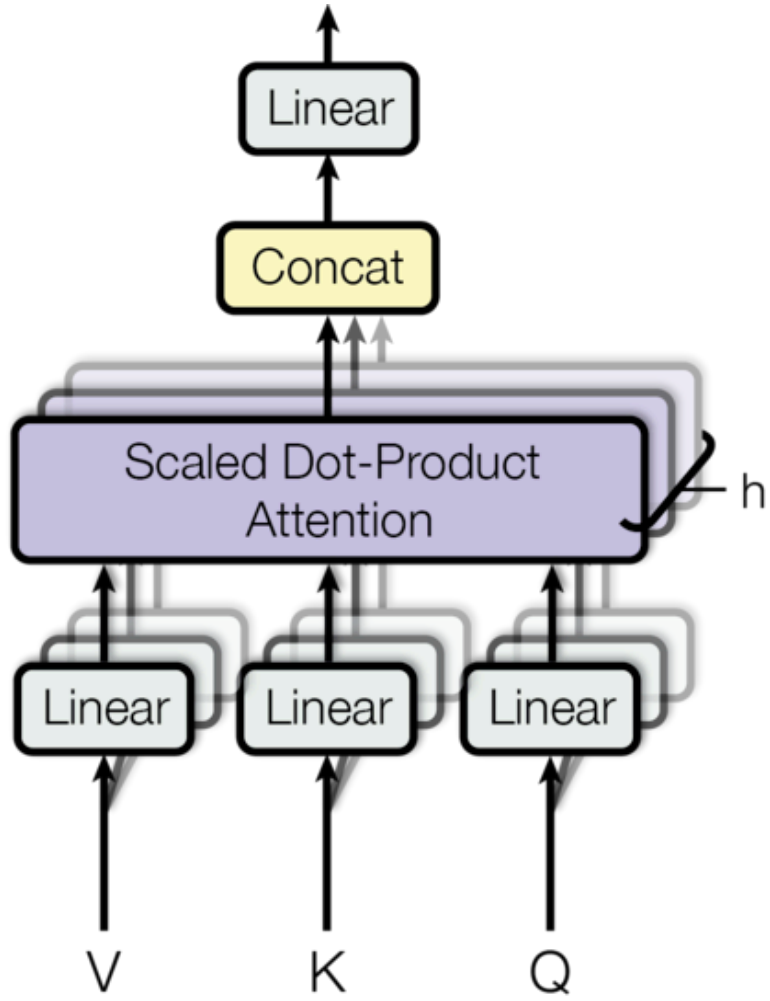
Parameters

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

Illustration

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

Visualization



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

```
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

Normalize each sample individually

```
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
```

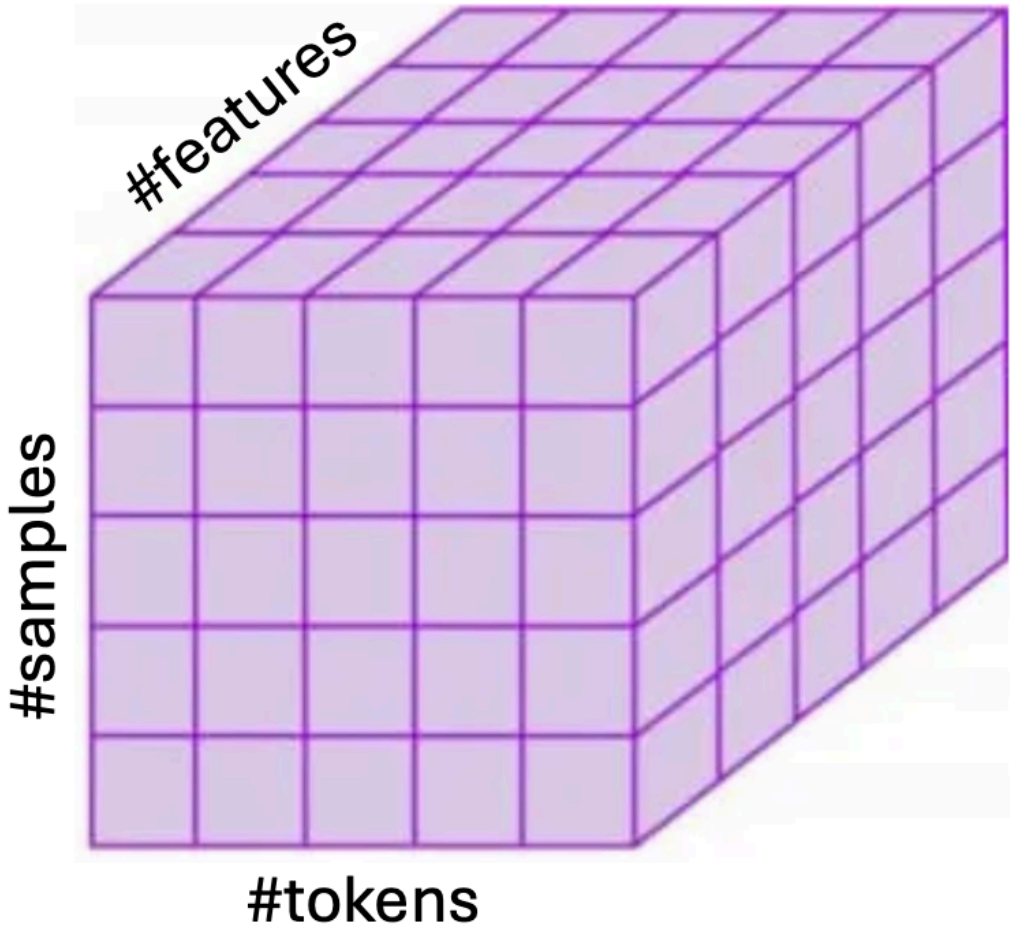
(Ba et al., 2016)

Layer Normalization vs. Batch Normalization

Conceptually,

- Batch Normalization normalizes **per-feature** across the batch,
- Layer Normalization normalizes **per-token** across the features.

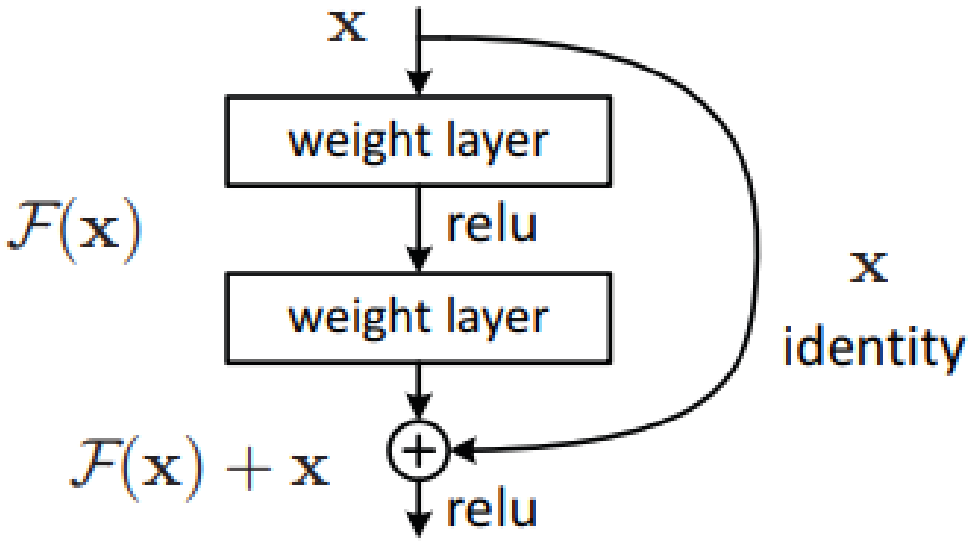
(Actually, LLMs does not use batch normalization.)



Residual Connection

Residual connection from prior work

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



(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

```
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

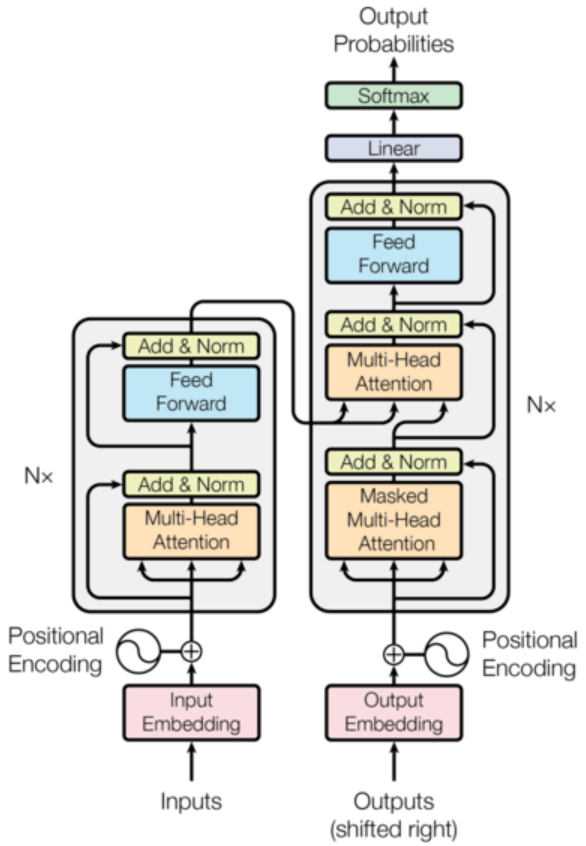
```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"

    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),
    )
```

Positional Encoding: Requirements

A good positional encoding should

- provide unique representation for each position
- have a consistent way to express the number of steps between two input vectors
- be able to scale to longer sequences

Positional Encoding: An Implementation

The implementation suggested in (Vaswani et al., 2017)

$$r_{ni} = \begin{cases} \sin(\frac{n}{L^{i/D}}) & \text{if } i \text{ is even} \\ \cos(\frac{n}{L^{(i-1)/D}}) & \text{if } i \text{ is odd} \end{cases}$$

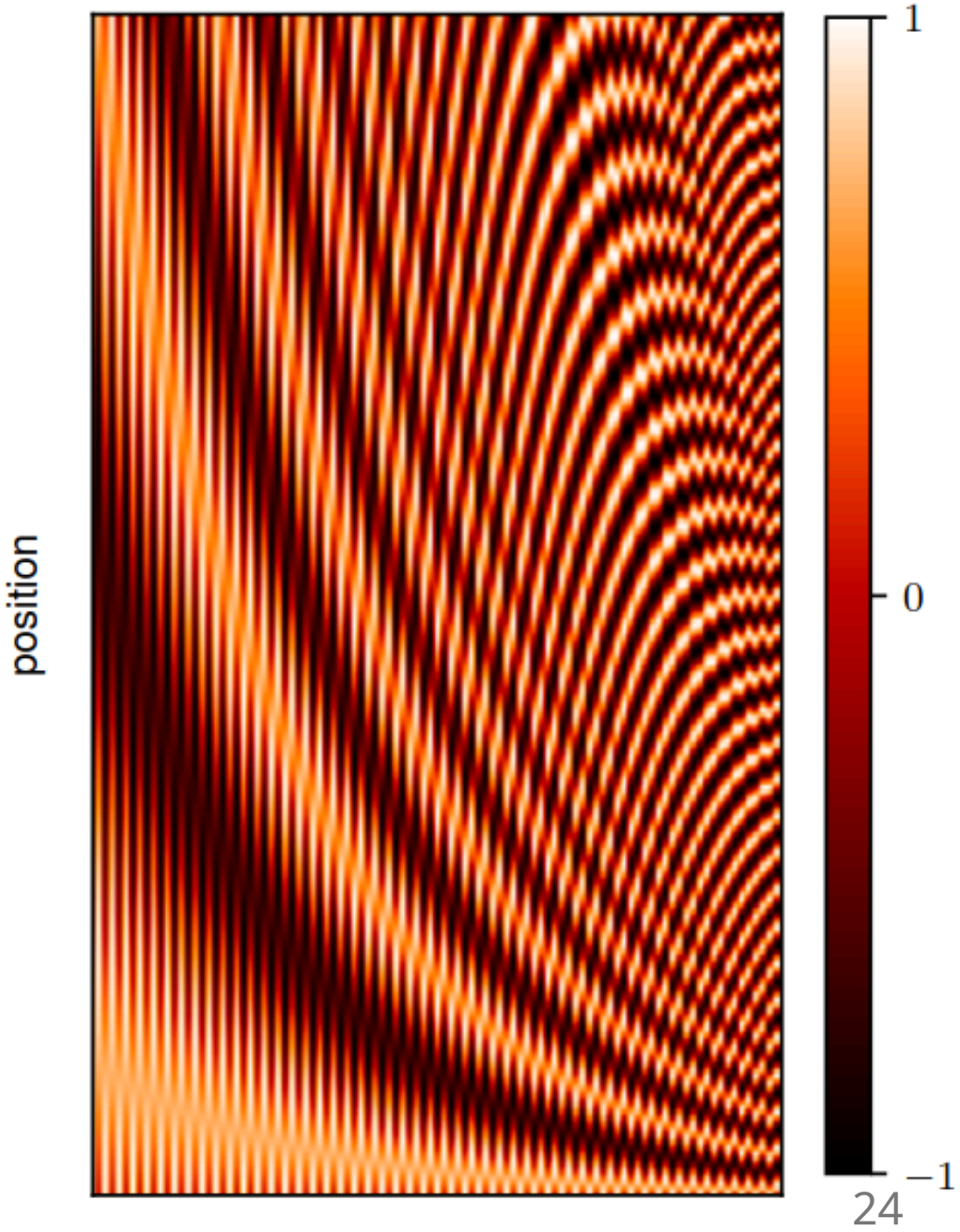
where D is the dimension of embeddings, n is the position index, and i is the embedding dimension index.

Positional Encoding: A Visualization

With

- $D = 100$
- $L = 30$

The first $N = 200$ positional embeddings



Transformer Visualization

Transformer Explainer

What Else?

- Tokenization
- Word embeddings
- Positional embeddings

Section II

BERT

(Devlin et al., 2018)

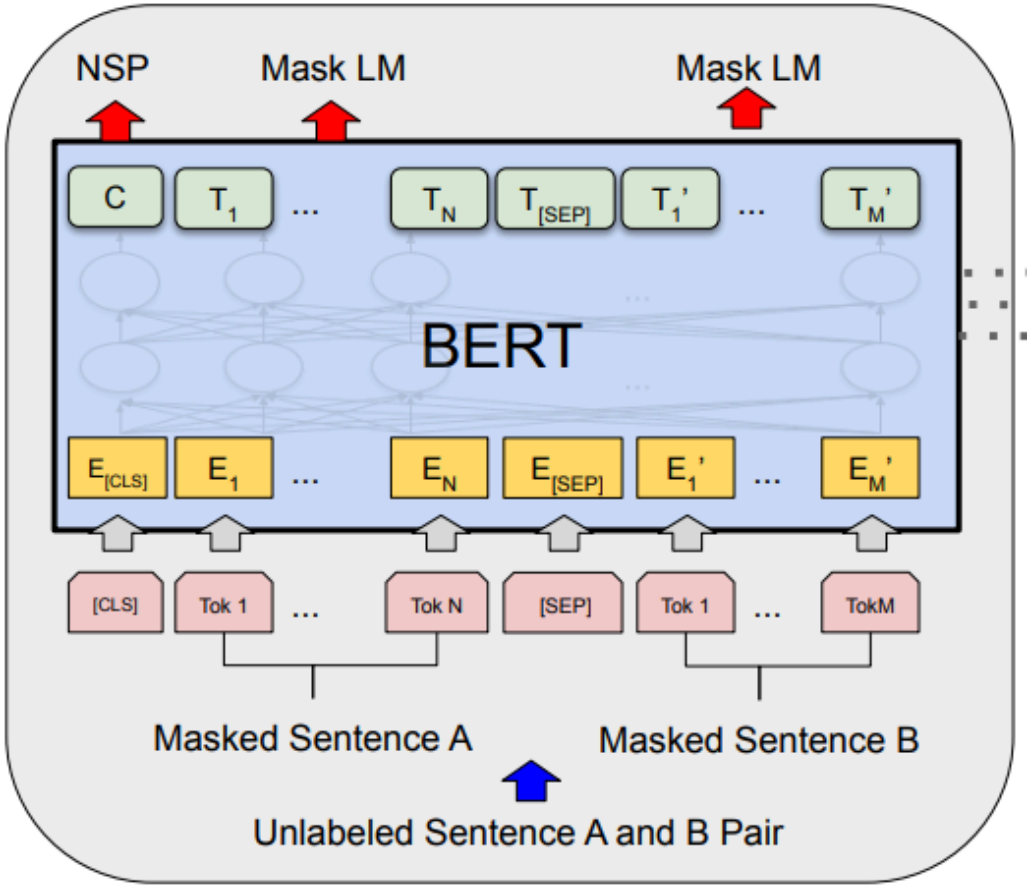
Pre-training

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: Overview

WordPiece is a subword tokenization algorithm. For example

Input

```
Unfriendly dog behavior
```

Output:

```
['Un', '##friend', '##ly', 'dog', 'behav', '##ior']
```

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

(Wu et al., 2016)

Wordpiece Tokenization: Method

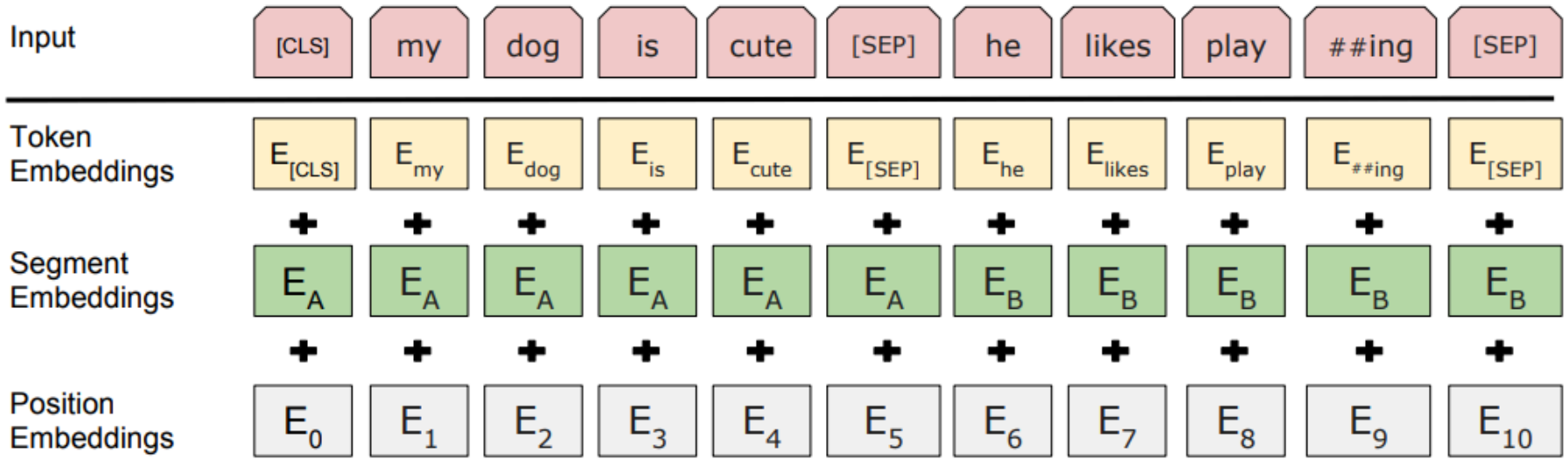
- Training
  - Initialization: The vocabulary starts with every individual character in the corpus.
  - Iteration: The algorithm iteratively merges two subtokens based on its score:

$$\text{score} = \frac{\text{frequency}(\text{subtoken}_i\text{subtoken}_j)}{\text{frequency}(\text{subtoken}_i)\text{frequency}(\text{subtoken}_j)}$$

- Tokenization
  - Use a greedy, longest-match-first algorithm

BERT Vocab

Input Representation



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

Masked Language Model

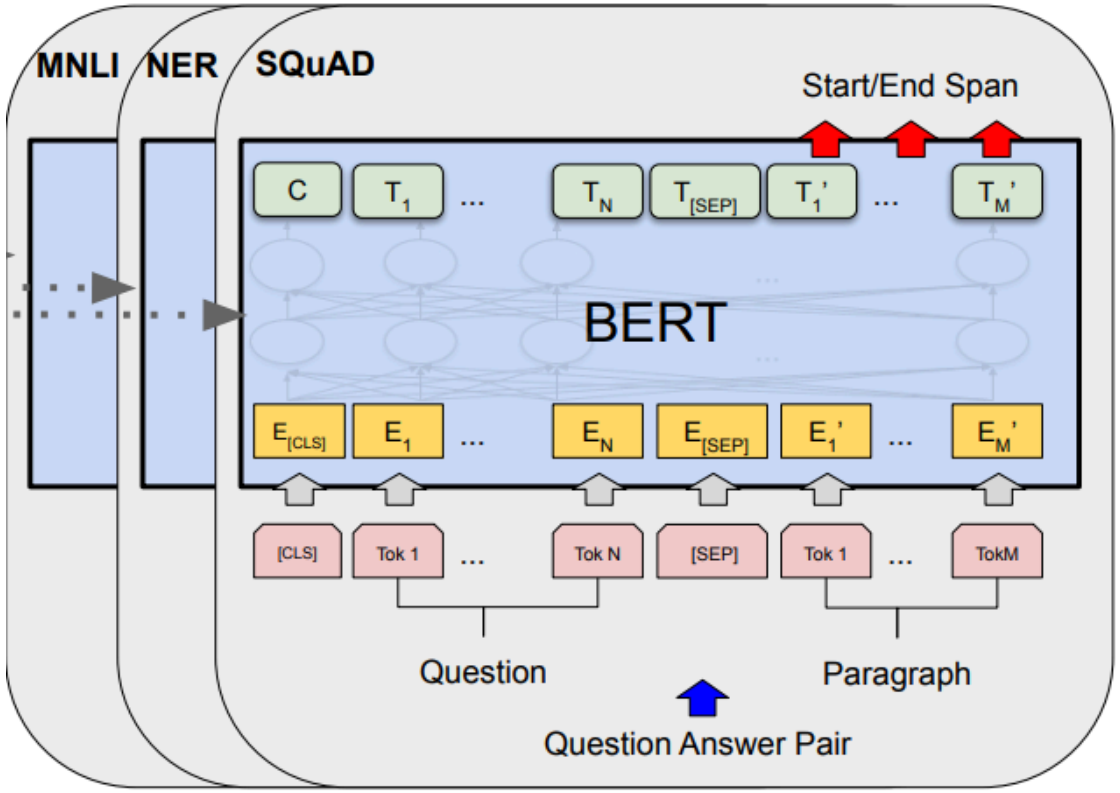
During pre-training, randomly mask some words in the text and ask the LM to predict them

```
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

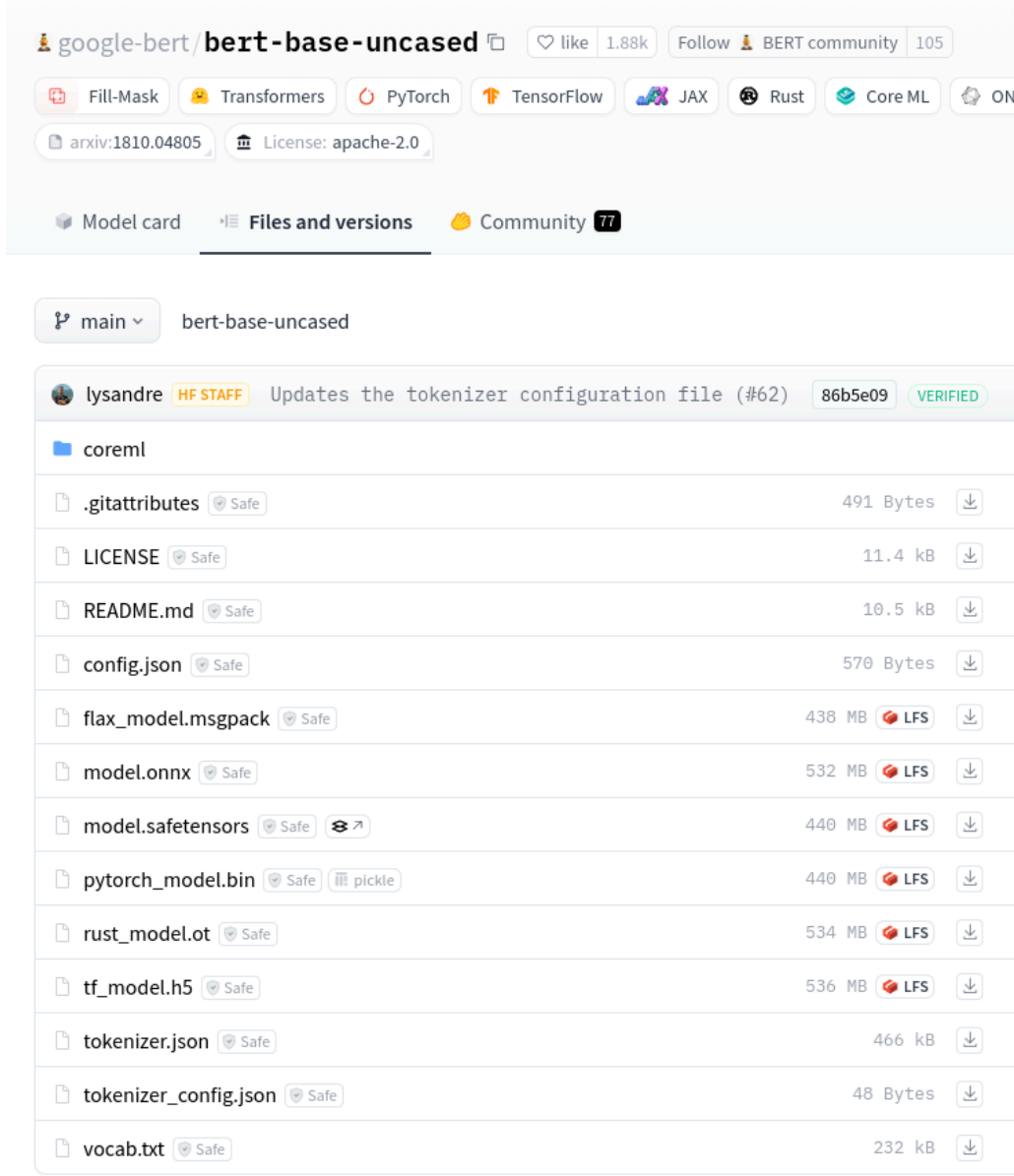


For more information, please refer to this Hugging Face page

Some Implementation Details

More about model configuration and tokenization.

A simple demo



Section III

The GPT Family

The GPT Family

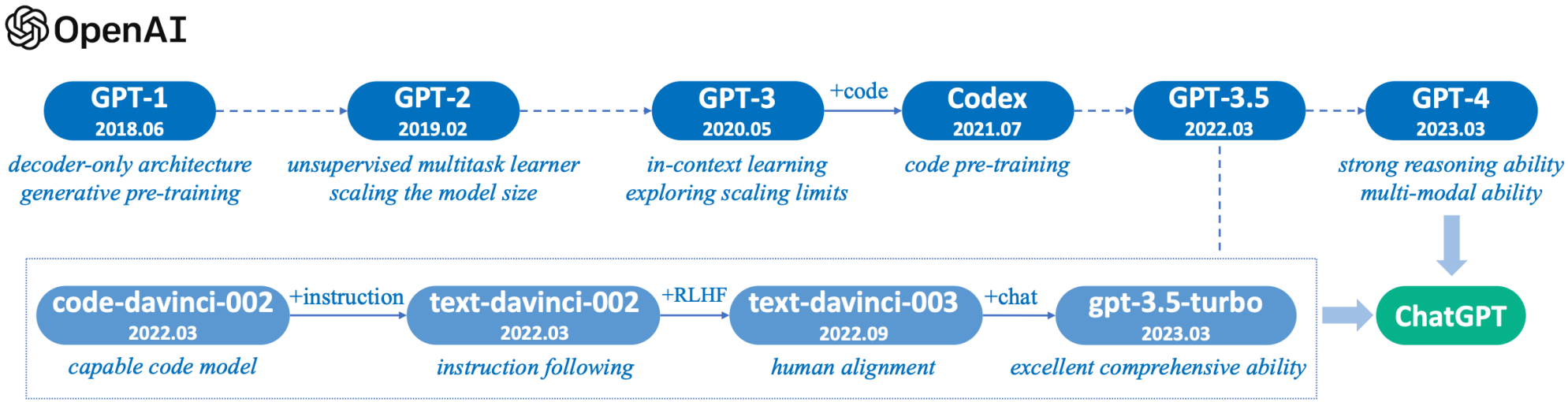


Fig. 3: A brief illustration for the technical evolution of GPT-series models. We plot this figure mainly based on the papers, blog articles and official APIs from OpenAI. Here, *solid lines* denote that there exists an explicit evidence (*e.g.*, the official statement that a new model is developed based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

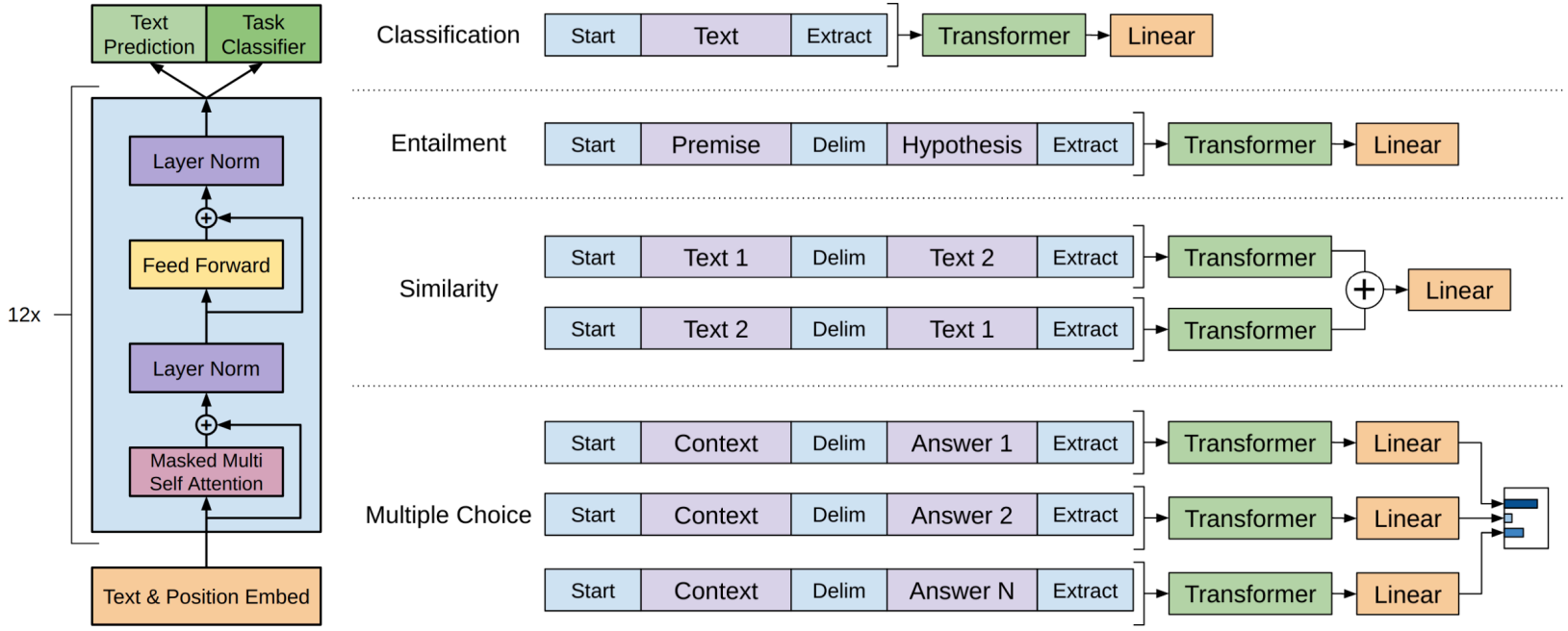


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

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

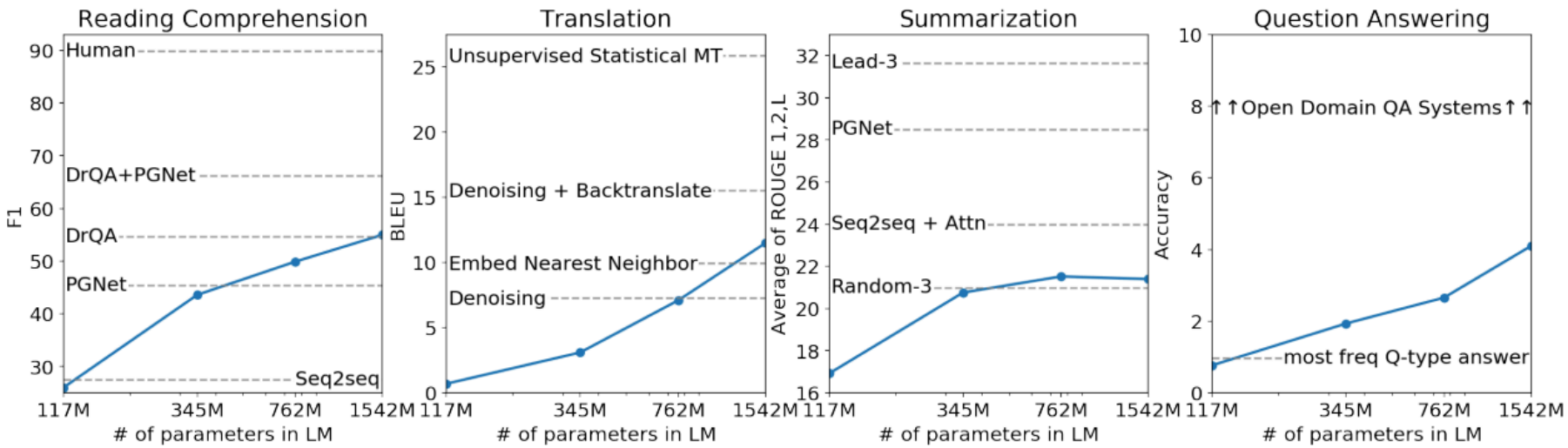
$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y | x^1, \ldots, x^m)$$

- 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: `I love coffee and like tea.`

- Create the character sequence
  - `I Ġ l o v e Ġ c o f f e e Ġ a n d Ġ l i k e Ġ t e a .`

(Sennrich et al., 2015)

Byte Pair Encoding (II)

Based on the character sequence

`I Ġ l o v e Ġ c o f f e e Ġ a n d Ġ l i k e Ġ t e a .`

- Initial vocab

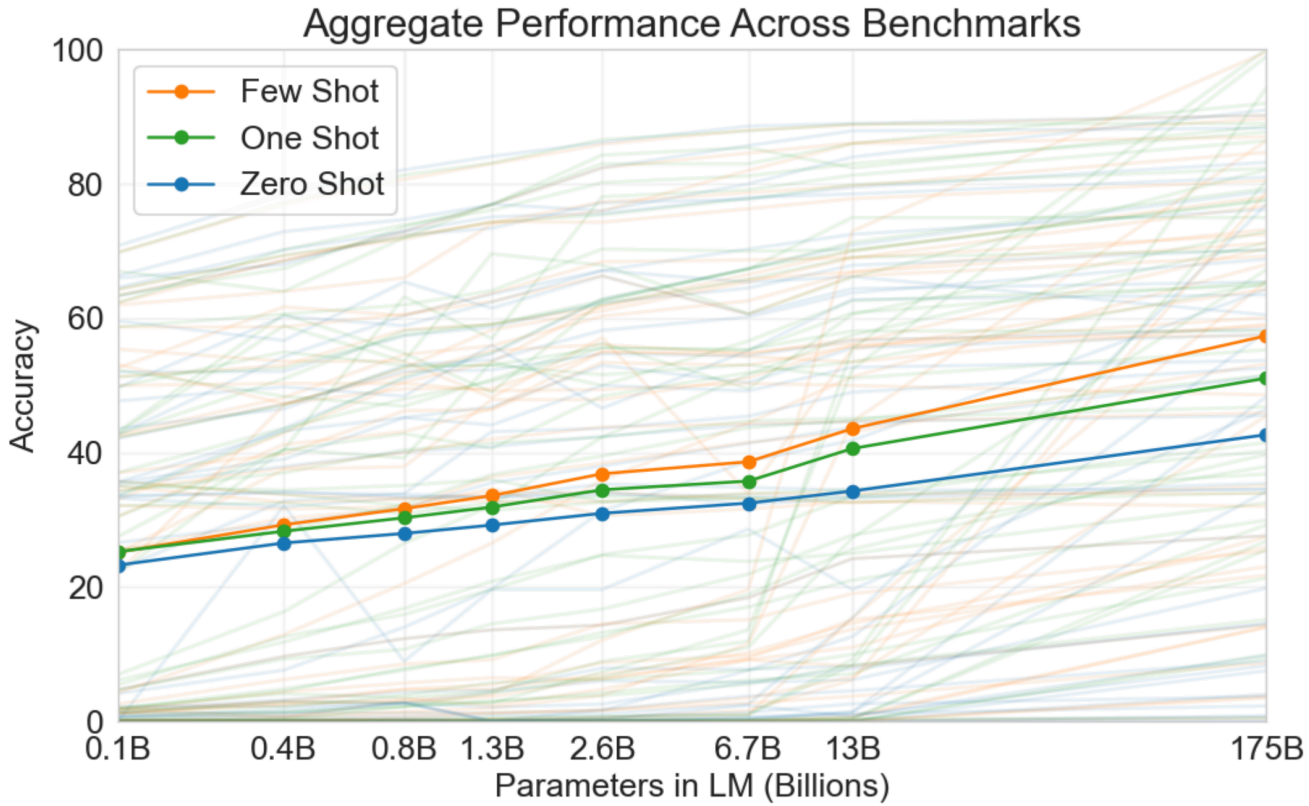
  `I Ġ l o v e c o f e a n d t`
- First step of merge operation: `Ġ` and `l` as `Ġl`
- The vocab after the first merge operation:

  `I Ġ l o v e c o f e a n d t Ġl`

Link

GPT-3: LMs as Few-shot Learners

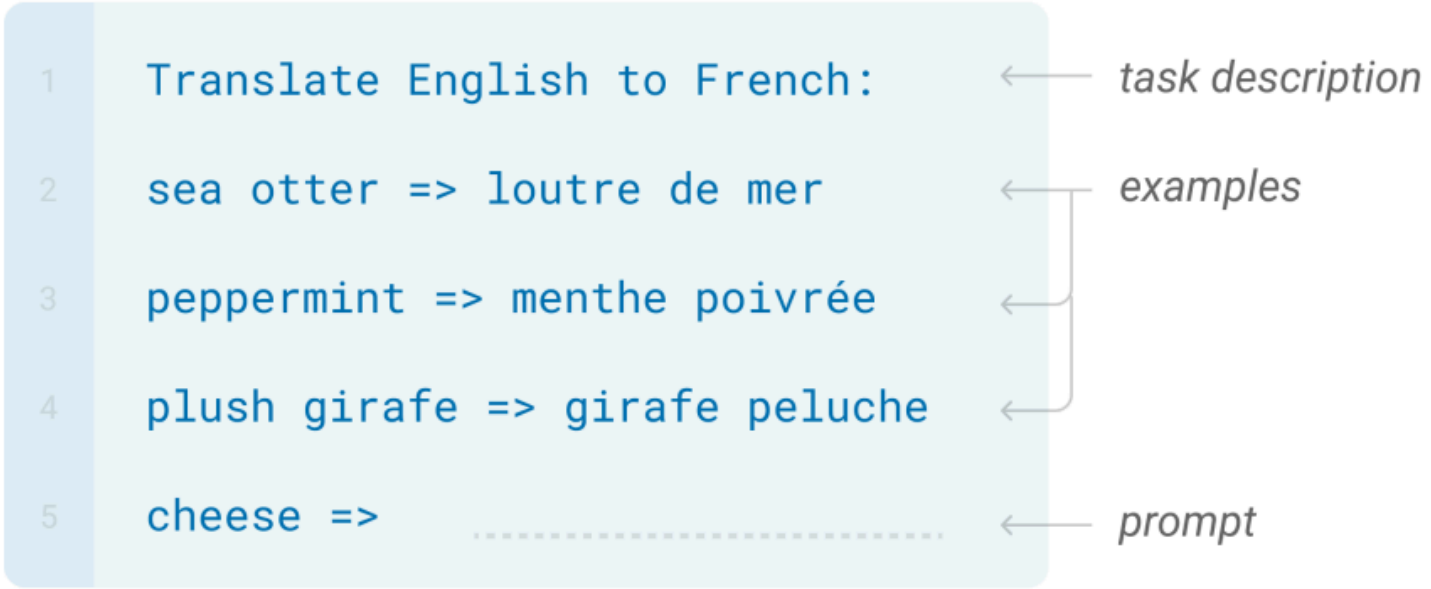
The performance of GPT-3 on few-shot in-context learning



Larger models produce better performance

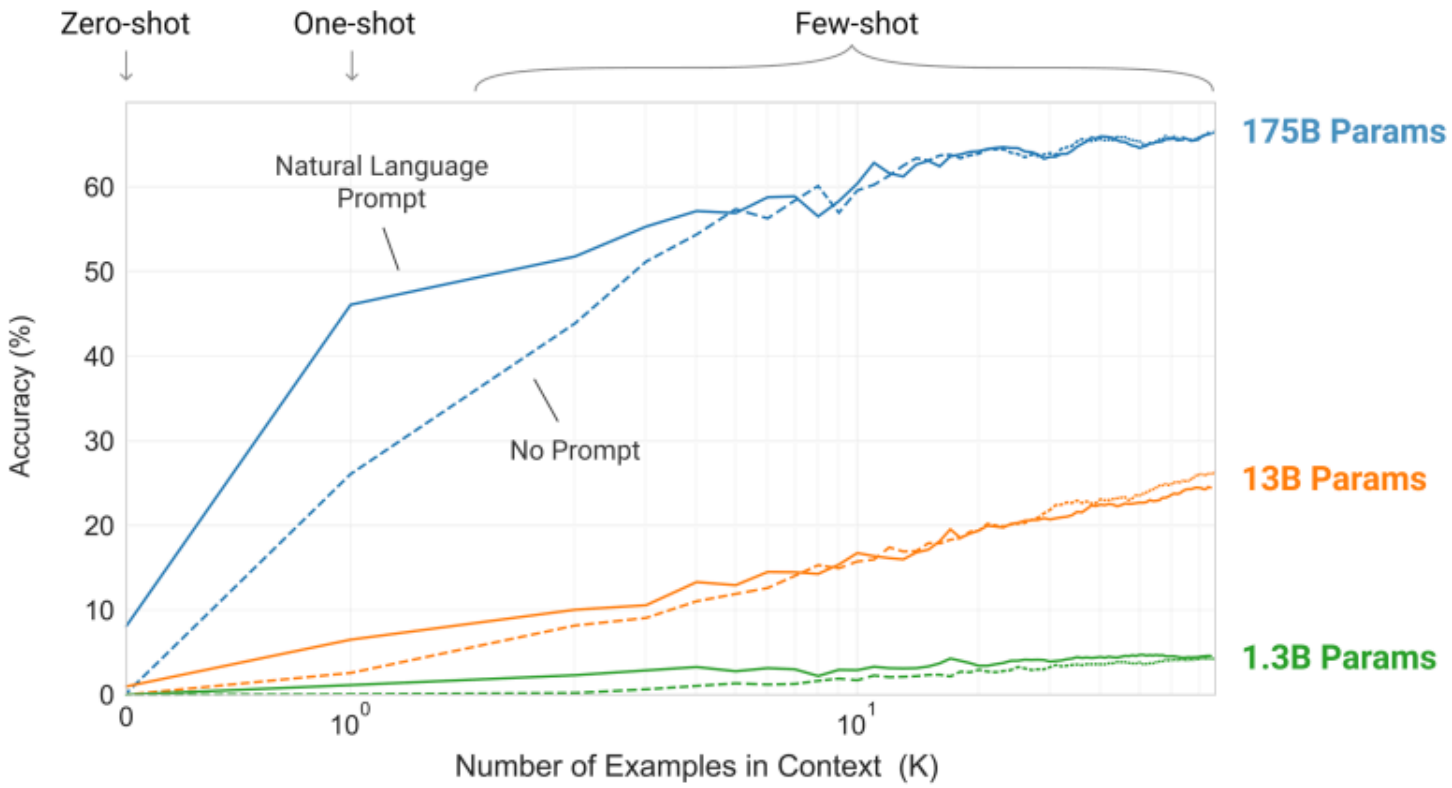
GPT-3: In-Context Learning

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



Thank You!