浙江工业大学

文本分析与挖掘实验报告

(2021 级)

期中调研: 预训练语言模型 BERT 的发展历史和动态

学生姓名：温家伟
学生学号：202103151422
学科专业：大数据分析 2101 班
所在学院：理学院
提交日期：2023 年 12 月 18 日

目录

1 发展历史

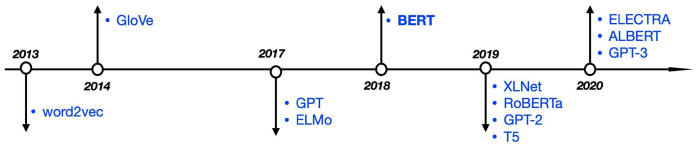

图 1: 语言模型的发展历程

1.1 One-Hot

将单词通过词表映射到矩阵（向量）空间，词表的长度是矩阵（向量）的一个维度。

比如：motel 和 hotel 两个单词，在词表下的 one-hot 表征如下（注意，这里的词表大小 vocab-size = 15，为方便表征，通常的词表大小 > 10K）：

图 2: One-Hot 示意图

1.2 Word2Vec

这应该是当前大火的 Pretrain Model 的雏形，将无标注的语料根据训练目标的不同，通过深度学习方式进行训练，然后取其中的一层或多层综合作为一个单词的 embedding（表征）。Word2Vec 有两种训练目标，分别是根据上下文预测单词（CBOW）和根据单词推测上下文（Skip-Gram）。Word2Vec 是一种静态获取单词语义编码的方法。

论文的另一个出彩之处还在于后面使用霍夫曼树的方式进行 decoder，能够根据词的频率高低快速预测（解码）词语。

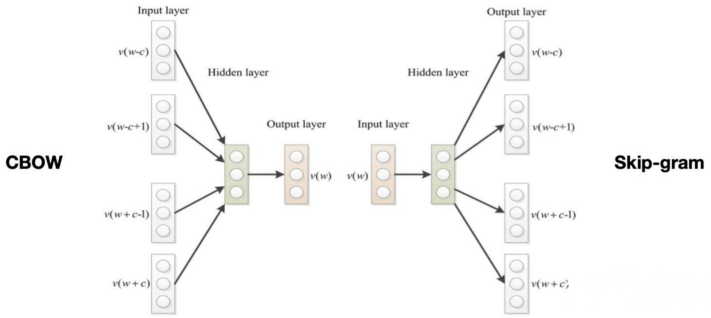

图 3: Word2Vec

1.3 Glove & Cove

Glove 算是对标于 Word2Vec，只不过训练方法和训练目标不一样了。主要特点是：

- 使用了上下文的信息来进行 word embedding，但仍是静态统计的。
- 训练目标，期望两个词的 word vectors 的点乘和两个单词的共线概率一致。

Glove 之后有 Cove，可能很多人没听过它，它属于一种小浪花级别的发展，用于当时很火的机器翻译（Machine Translation），其实就是通过将两个对标的不同语言的句子，如 Sent-DE，Sent-EN 通过双向 LSTM 能够相互映射和预测，思路和方法与 Glove 类似，但是预料是对标的翻译预料。相当于当前的 multi-LM，但是略有不同，主要针对的是机器翻译任务。

注意，前面提到的这些模型都是不能 Fine-Tune 的，此时的 Embedding 真的只是 embedding，用于下游任务的表征和特征表示。

1.4 ELMo

ELMo，Embeddings from Language Models (NAACL 2018 Best Paper)。离 BERT 很近了，ELMo 使用的任务也是 LM-Language Model 的任务，也就是已知上文，预测下一个单词的任务，只不过使用的架构是 BiLSTM，这点和 BERT 稍有区别，也从侧面体现出了 Transformer 结构的强大。当然，ELMo 使用的训练预料数量也远远低于 BERT，这也是很多当前的预训练模型使用的“大力出奇迹”训练法。

ELMo 可以算是第一个动态语义表征的预训练语言模型，也因此获得了 NAACL 2018 的 Best Paper。

1.5 GPT

GPT，Generative Pre-Training，可以说是开启使用 PLM + Fine-Tune 模式完成 NLP 任务的先驱，也是奠定者（PLM，Pretrain Language Model）。GPT 的贡献度是高于 BERT 的，首先它提出的 PLM + Fine-Tune 模式；其次，使用了 Transformer 的模型结构；第三点，便是开启了大数据 PLM 训练时代，BERT 的很多点都是借鉴 GPT 的。

既然作为 BERT 之前最辉煌的模型，我们也放一张比对图，看看 ELMo，GPT 和 BERT 的差异。首先 GPT 模型使用的框架已经是当时最火的 Transformer 框架了，然而模型任务还是使用的语言模型的任务，并且是单向的。当然，并非说是语言模型任务劣于完形填空任务（MLM，mask language model，cloze task），而是在语义表征上 MLM 更能获得很好的语义表征，擅长 NLU 类的下游任务。而 LM 任务则更适合语义推理，所以更擅长 NLG 类的下游任务。展开来说（可以略过啊，后面是个人理解），OpenAI 青睐于 LM 任务，所以后面的 GPT-n 模型均擅长 NLG 任务，GPT-3 尤为之最。但在当前通用的 NLP 数据集上，NLU 任务多于 NLG 任务，所以在表现上，BERT 的表现更抢人眼球，毕竟数据指标说话。当然，BERT 的很多细节处理也确实优于 GPT，这也是它能刷新榜单的原因。

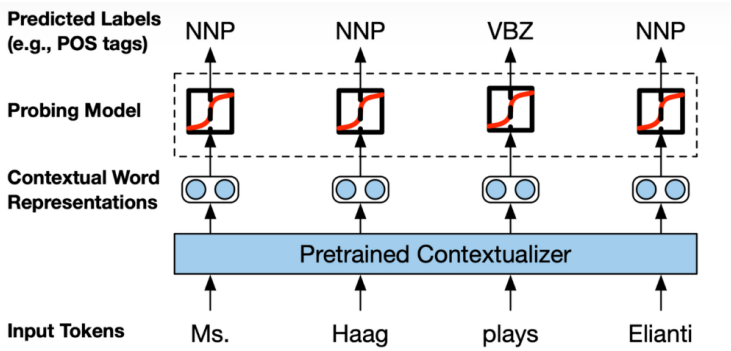

图 4: 对比图

- 单向双向：ELMo 和 BERT 使用双向，GPT 使用单向；

- Transformer：GPT 和 BERT 使用 Transformer 框架，ELMo 使用 LSTM；
- 任务：ELMo 和 GPT 使用语言模型（LM）任务，BERT 使用完形填空。

1.6 BERT

BERT 出场了，BERT，Bidirectional Encoder Representations from Transformers (NAACL 2019 Best Paper) 。具体的介绍就不说细节了，因为本文主要是 Summary Note 模式的，不做细节展开。说一下 BERT 的几个特点。

- MLM（Mask Language Model）+ NSP （Next Sentence Prediction），两个任务；
- 大数据集，800M (BooksCorpus) + 2500M (Wikipedia) ，GPT 只用了前者的 800M；
- 大数据集，800M (BooksCorpus) + 2500M (Wikipedia) ，GPT 只用了前者的 800M；

大数据集，800M (BooksCorpus) + 2500M (Wikipedia) ，GPT 只用了前者的 800M；刷新 NLP 各大榜单，同时，开启新时代，暴力训练的时代！一直说暴力训练，大家可能对这个词没有太多的理解，对于 2500M 训练数据没啥概念，对于 330M 参数量（BERT-Large）没啥具体认知。2500M 文本数据，以一本百万字的小说为例，其存储大约是 2M 左右，打印成册，大约有 10 本西瓜书那么厚，2500M，可以是塞满一栋屋子的书籍的字。那么 330M 参数量又如何理解呢，还记得我们解方程的时候学习的一元一次方程，然后到 2 元一次方程组，要两个方程一起联立来解，就这一个变化，我们那时候都得学一个多月。330M 参数，就是 330M 元方程组，因为激活函数的原因，还不是一次的，所以这个训练过程其实是 330M 非线性方程组的求解过程。可以细节的感受一下。

2 发展动态

2.1 Bert 概述

Bert 全称是“Bidirectional Encoder Representations from Transformers”，BERT 是一种预训练语言模型（pre-trained language model, PLM）。Google 团队在 2018 年发表文章《BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding》提出了 Bert 预训练语言模型，可以说 Bert 的出现轰动了整个 NLP 领域，自然语言处理领域开始进入一个新的阶段。

Bert 和 ELMO，GPT 都一样是两阶段的任务（预训练 + 微调）：

- 预训练阶段 (pre-training)：模型将使用大量的无标签数据训练。
- 微调阶段 (fine-tuning)：BERT 模型将用预训练模型初始化所有参数，这些参数将针对于下游任务，比如文本分类，序列标注任务等，微调阶段需要使用有标签的数据进行模型训练，不同的下游任务可以训练出不同的模型，但是每次都会使用同一个预训练模型进行初始化。

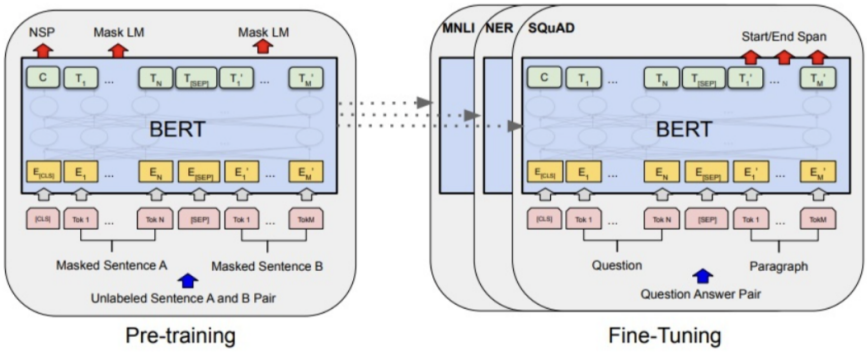

图 5: BERT 的预训练和微调流程

2.2 BERT 的结构

Bert 是基于 Transformer 实现的，主要是 Transformer 的 Encoder 部分，完整架构如下：

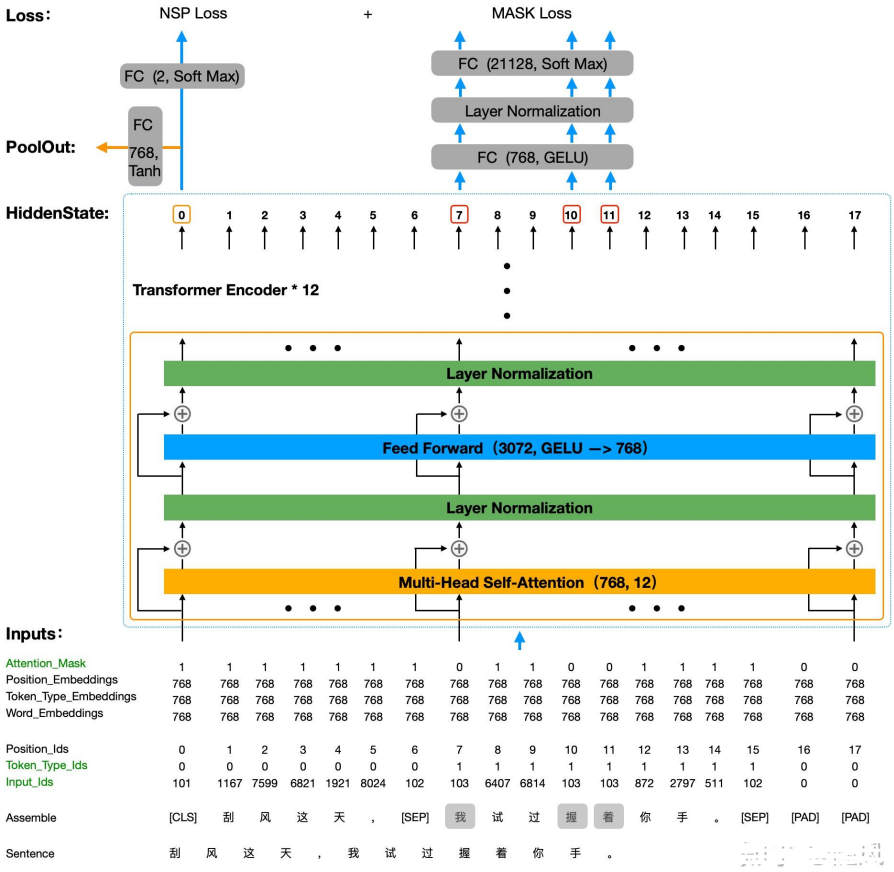

图 6: BERT 架构

论文中提到的 Bert 主要有两种大小，bert-base 和 bert-large 两个 size，base 版一共有 110M 参数，large 版有 340M 的参数，总之 Bert 有上亿的参数量。

- ERT_BASE: L = 12, H = 768, A = 12, Total Parameters = 110M.
- BERT_LARGE: L = 24, H = 1024, A = 16, Total Parameters = 340M.

其中 L：Transformer blocks 层数；H：hidden size；A：the number of self-attention heads。

2.3 Bert 的输入输出形式

Bert 的 Embedding 层由 3 个子层求和得到，分别是词向量层 Token Embedings，句子层 Segment Embeddings 以及位置编码层 Position Embeddings。

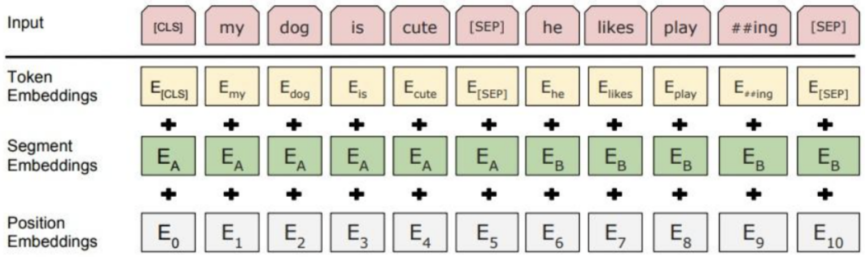

图 7: Bert 的 Embedding 层

- Token Embeddings 字向量：用来表征不同的词，以及特殊的 tokens，第一个单词是 CLS 标志，主要用于之后的分类任务。
- Segment Embeddings 文本向量：用来区别两个句子，来表征这个词是属于哪一个句子，作用于两个句子为输入的分类任务。
- Position Embeddings 位置向量：由于出现在文本不同位置的字/词所携带的语义信息存在差异，对不同位置的字/词分别附加一个不同的向量以作区分，是随机初始化训练出来的结果。

Bert 输出:

主要输出各字对应的融合全文语义信息后的向量表示。

3 代码学习

3.1 Tokenization（BertTokenizer）

```
class BertTokenizer(PreTrainedTokenizer):
    """
    Construct a BERT tokenizer. Based on WordPiece.

```

```
5       This tokenizer inherits from
            :class:`~transformers.PreTrainedTokenizer`
            which contains most of the main methods.
6       Users should refer to this superclass for more
            information regarding those methods.
7       ...
8       """
```

BertTokenizer 是基于 **BasicTokenizer** 和 **WordPieceTokenizer** 的分词器：

- **BasicTokenizer** 负责处理的第一步——按标点、空格等分割句子，并处理是否统一小写，以及清理非法字符。
  - 对于中文字符，通过预处理（加空格）来按字分割；
  - 同时可以通过 **never_split** 指定对某些词不进行分割；
  - 这一步是可选的（默认执行）。
- **WordPieceTokenizer** 在词的基础上,进一步将词分解为子词(**subword**)。
  - **subword** 介于 **char** 和 **word** 之间，既在一定程度保留了词的含义，又能够照顾到英文中单复数、时态导致的词表爆炸和未登录词的 **OOV**（**Out-Of-Vocabulary**）问题，将词根与时态词缀等分割出来，从而减小词表，也降低了训练难度；
  - 例如，**tokenizer** 这个词就可以拆解为“**token**”和“**##izer**”两部分，注意后面一个词的“**##**”表示接在前一个词后面。

BertTokenizer 有以下常用方法：

- **from_pretrained**：从包含词表文件（**vocab.txt**）的目录中初始化一个分词器；
- **tokenize**：将文本（词或者句子）分解为子词列表；
- **convert_tokens_to_ids**：将子词列表转化为子词对应下标的列表；

- **convert_ids_to_tokens** ：与上一个相反；
- **convert_ids_to_tokens** ：与上一个相反；
- **convert_tokens_to_string**：将 **subword** 列表按“**##**”拼接回词或者句子；
- **encode**：对于单个句子输入，分解词并加入特殊词形成“**[CLS], x, [SEP]**”的结构并转换为词表对应下标的列表；对于两个句子输入（多个句子只取前两个），分解词并加入特殊词形成“**[CLS], x1, [SEP], x2, [SEP]**”的结构并转换为下标列表；
- **decode**：可以将 **encode** 方法的输出变为完整句子。

3.2 Model（BertModel）

和 **BERT** 模型有关的代码主要写在**/models/bert/modeling_bert.py**中，这一份代码有一千多行，包含 **BERT** 模型的基本结构和基于它的微调模型等。

下面从 **BERT** 模型本体入手分析：

```
1
2 class BertModel(BertPreTrainedModel):
3     """
4
5     The model can behave as an encoder (with only
          self-attention) as well as a decoder, in which
          case a layer of
6     cross-attention is added between the
          self-attention layers, following the
          architecture described in 'Attention is
7     all you need
          <https://arxiv.org/abs/1706.03762>'__ by
          Ashish Vaswani, Noam Shazeer, Niki Parmar,
          Jakob Uszkoreit,
8     Llion Jones, Aidan N. Gomez, Lukasz Kaiser and
          Illia Polosukhin.
```

```
9
10      To behave as an decoder the model needs to be
            initialized with the :obj:`is_decoder`
            argument of the configuration
11      set to :obj:`True`. To be used in a Seq2Seq
            model, the model needs to initialized with
            both :obj:`is_decoder`
12      argument and :obj:`add_cross_attention` set to
            :obj:`True`; an :obj:`encoder_hidden_states`
            is then expected as an
13      input to the forward pass.
14      """
```

BertModel 主要为 **transformer encoder** 结构，包含三个部分：

- **embeddings**，即 **BertEmbeddings** 类的实体，对应词嵌入；
- **encoder**，即 **BertEncoder** 类的实体；
- **pooler**，即 **BertPooler** 类的实体，这一部分是可选的。

下面将介绍 **BertModel** 的前向传播过程中各个参数的含义以及返回值：

```
1       def forward(
2           self,
3           input_ids=None,
4           attention_mask=None,
5           token_type_ids=None,
6           position_ids=None,
7           head_mask=None,
8           inputs_embeds=None,
9           encoder_hidden_states=None,
10          encoder_attention_mask=None,
11          past_key_values=None,
12          use_cache=None,
13          output_attentions=None,
```

```
14          output_hidden_states=None,
15          return_dict=None,
16      ): ...
```

- **input_ids**：经过 **tokenizer** 分词后的 **subword** 对应的下标列表；
- **attention_mask**：在 **self-attention** 过程中，这一块 **mask** 用于标记 **subword** 所处句子和 **padding** 的区别，将 **padding** 部分填充为 **0**；
- **token_type_ids**：标记 **subword** 当前所处句子（第一句/第二句/**padding**）；
- **position_ids**：标记当前词所在句子的位置下标；
- **head_mask**：用于将某些层的某些注意力计算无效化；
- **inputs_embeds**：如果提供了，那就不需要 **input_ids**，跨过 **embedding lookup** 过程直接作为 **Embedding** 进入 **Encoder** 计算；
- **encoder_hidden_states**：这一部分在 **BertModel** 配置为 **decoder** 时起作用，将执行 **cross-attention** 而不是 **self-attention**；
- **encoder_attention_mask**：同上，在 **cross-attention** 中用于标记 **encoder** 端输入的 **padding**；
- **past_key_values**：这个参数貌似是把预先计算好的 **K-V** 乘积传入，以降低 **cross-attention** 的开销（因为原本这部分是重复计算）；
- **use_cache**：将保存上一个参数并传回，加速 **decoding**；
- **output_attentions**：是否返回中间每层的 **attention** 输出；
- **output_hidden_states**：是否返回中间每层的输出；

- **return_dict**: 是否按键值对的形式（**ModelOutput** 类，也可以当作 **tuple** 用）返回输出，默认为真。

返回部分如下：

```
# BertModel的前向传播返回部分
if not return\_dict:
    return (sequence\_output, pooled\_output) +
        encoder_outputs[1:]

return
    BaseModelOutputWithPoolingAndCrossAttentions(
     last\_hidden\_state=sequence\_output,
     pooler\_output=pooled\_output,
     past\_key\_values=encoder\_outputs.past\_key_values,
     hidden\_states=encoder\_outputs.hidden\_states,
     attentions=encoder\_outputs.attentions,
     cross\_attentions=encoder\_outputs.cross\_attentions,
)
```

可以看出，返回值不但包含了 **encoder** 和 **pooler** 的输出，也包含了其他指定输出的部分（**hidden_states** 和 **attention** 等，这一部分在 **encoder_outputs[1:]**）方便取用：

```
#
    BertEncoder的前向传播返回部分，即上面的encoder_outputs
if not return_dict:
    return tuple(
        for v in [
            hidden_states,
            next_decoder_cache,
            all_hidden_states,
            all_self_attentions,
            all_cross_attentions,
        ]
        if v is not None
```

```
12          )
13      return BaseModelOutputWithPastAndCrossAttentions(
14          last_hidden_state=hidden_states,
15          past_key_values=next_decoder_cache,
16          hidden_states=all_hidden_states,
17          attentions=all_self_attentions,
18          cross_attentions=all_cross_attentions,
19      )
```

3.3 BERT-based Models

基于 **BERT** 的模型都写在**/models/bert/modeling_bert.py** 里面，包括 **BERT** 预训练模型和 **BERT** 分类模型，**UML** 图如下：

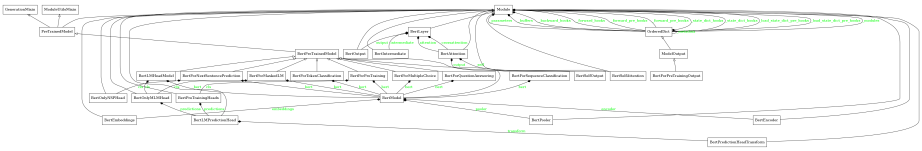

图 8: **BERT** 模型一图流

首先，以下所有的模型都是基于 **BertPreTrainedModel** 这一抽象基类的，而后者则基于一个更大的基类 **PreTrainedModel**。这里我们关注 **BertPreTrainedModel** 的功能：

- 用于初始化模型权重，同时维护继承自 **PreTrainedModel** 的一些标记身份或者加载模型时的类变量。

3.4 Model（BertModel）

3.5 BERT 训练和优化

Pre-Training 预训练阶段，除了众所周知的 **15**

不止 **BERT**，所有 **huggingface** 实现的 **PLM** 的 **word embedding** 和 **masked language model** 的预测权重在初始化过程中都是共享的：

```
1     class PreTrainedModel(nn.Module,
          ModuleUtilsMixin, GenerationMixin):
2     # ...
3     def tie_weights(self):
4         """
5         Tie the weights between the input embeddings
              and the output embeddings.
6
7         If the :obj:`torchscript` flag is set in the
              configuration, can't handle parameter
              sharing so we are cloning
8         the weights instead.
9         """
10        output_embeddings =
              self.get_output_embeddings()
11        if output_embeddings is not None and
              self.config.tie_word_embeddings:
12            self._tie_or_clone_weights(output_embeddings,
                  self.get_input_embeddings())
13
14        if self.config.is_encoder_decoder and
              self.config.tie_encoder_decoder:
15            if hasattr(self, self.base_model_prefix):
16                self = getattr(self,
                      self.base_model_prefix)
17            self._tie_encoder_decoder_weights(self.encoder,
                  self.decoder, self.base_model_prefix)
18    # ...
```

至于为什么，应该是因为 **word_embedding** 和 **prediction** 权重太大了，以 **bert-base** 为例，其尺寸为 **(30522, 768)**，降低训练难度。

3.6 Fine-Tuning

微调也就是下游任务阶段，也有两个值得注意的地方。

首先介绍一下 **BERT** 的优化器：**AdamW**（**AdamWeightDecayOptimizer**）。

这一优化器来自 **ICLR 2017** 的 **Best Paper**：《**Fixing Weight Decay Regularization in Adam**》中提出的一种用于修复 **Adam** 的权重衰减错误的新方法。论文指出，**L2** 正则化和权重衰减在大部分情况下并不等价，只在 **SGD** 优化的情况下是等价的；而大多数框架中对于 **Adam+L2** 正则使用的是权重衰减的方式，两者不能混为一谈。

BERT 的训练中另一个特点在于 **Warmup**，其含义为：

- 在训练初期使用较小的学习率（从 **0** 开始），在一定步数（比如 **1000** 步）内逐渐提高到正常大小（比如上面的 **2e-5**），避免模型过早进入局部最优而过拟合；
- 在训练后期再慢慢将学习率降低到 **0**，避免后期训练还出现较大的参数变化。

在 **Huggingface** 的实现中，可以使用多种 **warmup** 策略：

```
1  TYPE_TO_SCHEDULER_FUNCTION = {
2      SchedulerType.LINEAR:
           get_linear_schedule_with_warmup,
3      SchedulerType.COSINE:
           get_cosine_schedule_with_warmup,
4      SchedulerType.COSINE_WITH_RESTARTS:
           get_cosine_with_hard_restarts_schedule_with_warmup,
5      SchedulerType.POLYNOMIAL:
           get_polynomial_decay_schedule_with_warmup,
6      SchedulerType.CONSTANT: get_constant_schedule,
7      SchedulerType.CONSTANT_WITH_WARMUP:
           get_constant_schedule_with_warmup,
8  }
```

具体而言：

- **CONSTANT**：保持固定学习率不变；

- **CONSTANT_WITH_WARMUP**：在每一个 **step** 中线性调整学习率；
- **LINEAR**：上文提到的两段式调整；
- **COSINE**：和两段式调整类似，只不过采用的是三角函数式的曲线调整；
- **COSINE_WITH_RESTARTS**：训练中将上面 **COSINE** 的调整重复 **n** 次；
- **POLYNOMIAL**：按指数曲线进行两段式调整。