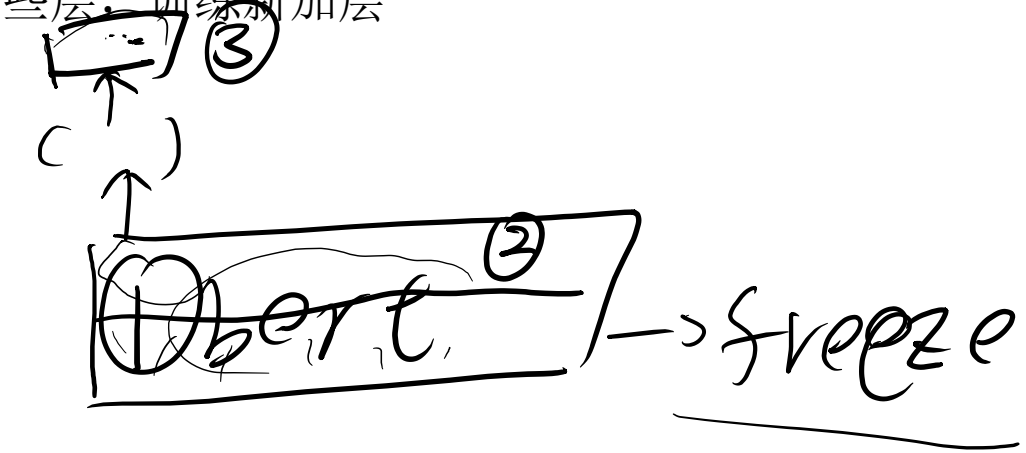
大模型微调

调参：
Batchsize=10
6*6b*10=360G 显存

Adam

Bert
T5
Bart

传统大模型微调：
1.微调所有参数
2.调整部分参数（冻结部分层）
3.对模型尾部加一些层，训练新加层



扫码了解更多

```python
model = AutoModel.from_pretrained(
    "E:\code\chatglm\chatglm2", load_in_8bit=False, trust_remote_code=True, device_map="auto"
).cuda()#.to(torch.float32)
```

```
|===============================+======================+======================|
|   0  NVIDIA RTX A6000    WDDM | 00000000:01:00.0  On |                  Off |
| 30%   44C    P5    30W / 300W |  13232MiB / 49140MiB |     13%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

```python
model = AutoModel.from_pretrained(
    "E:\code\chatglm\chatglm2", load_in_8bit=False, trust_remote_code=True, device_map="auto"
).cuda().to(torch.float32)
```

```
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A6000    WDDM | 00000000:01:00.0  On |                  Off |
| 30%   46C    P8    26W / 300W |  29119MiB / 49140MiB |     23%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

待训练的参数，一定要用float32
如果用float16 可能会出现 nan inf

直接微调模型参数，使用场景：

1.数据量大 token数量>=可调参数

2.机器算力足 显存大小决定能否训练 显卡flops决定训练时间

3.场景非常垂直

显存里面到底存了哪些数据：

- 模型：6b个float32 浮点数

Adam梯度下降法：

- 存前向结果
- 存之前步骤的梯度结果

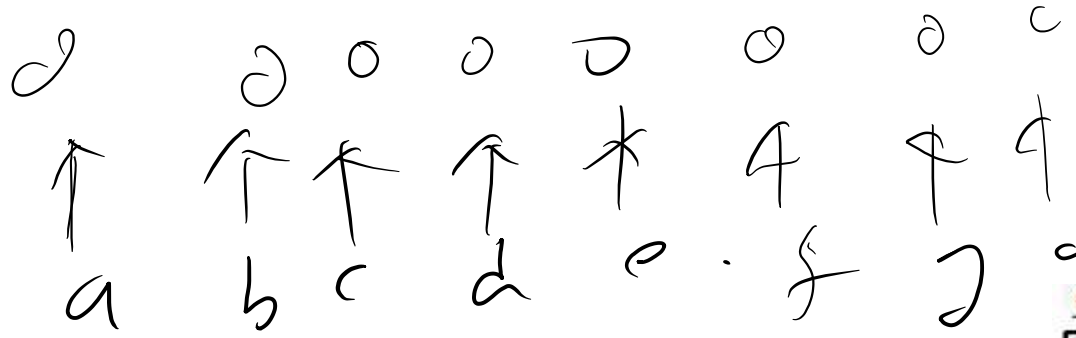
$n \times 6 \times 6b$

因为显存的限制：

模型可训练参数不能太多

Batchsize不能太大

Max_length 不能太大



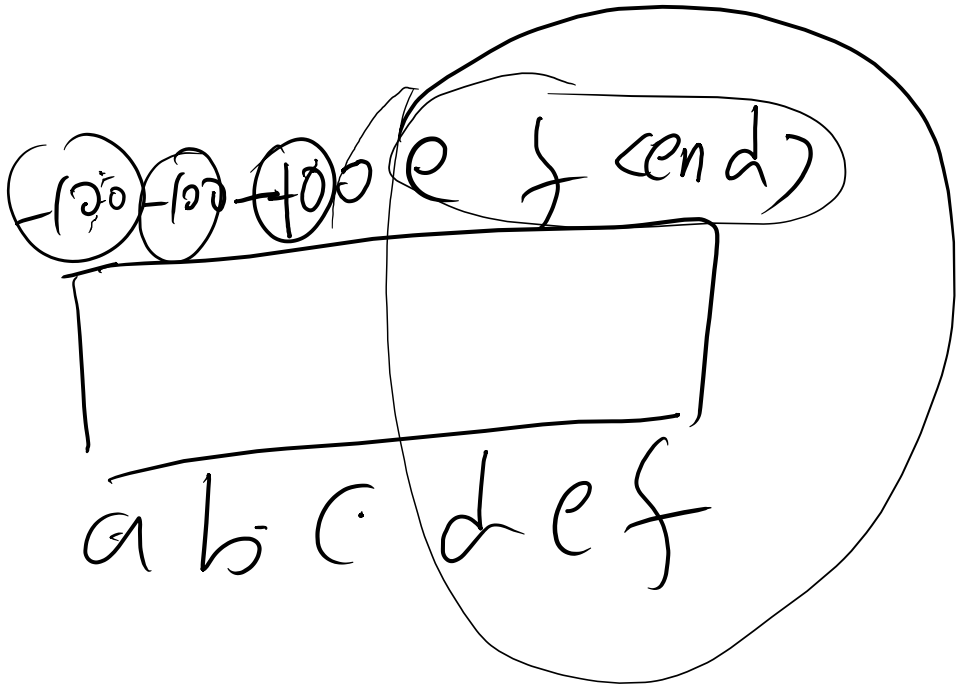
分享经验：
Maxlength：一般会统计平均长度（去除异常值） 平均长度*1.5
确定好训练的参数
Batchsize理论上来说，越大越好

扫码了解更多

bpe

Bpe分词：常见组合的字符串拼接在一起
Sentencepiece：Google出现分词工具
它可以实现bpe
把所有的文本都转成utf-8
所以有可能：
1。几个汉字对应一个token
2.2个token对应一个汉字

max length



模型本质是个语言模型
Gpt为例

Q A

Q: abcd

A: efgh

训练

扫码了解更多

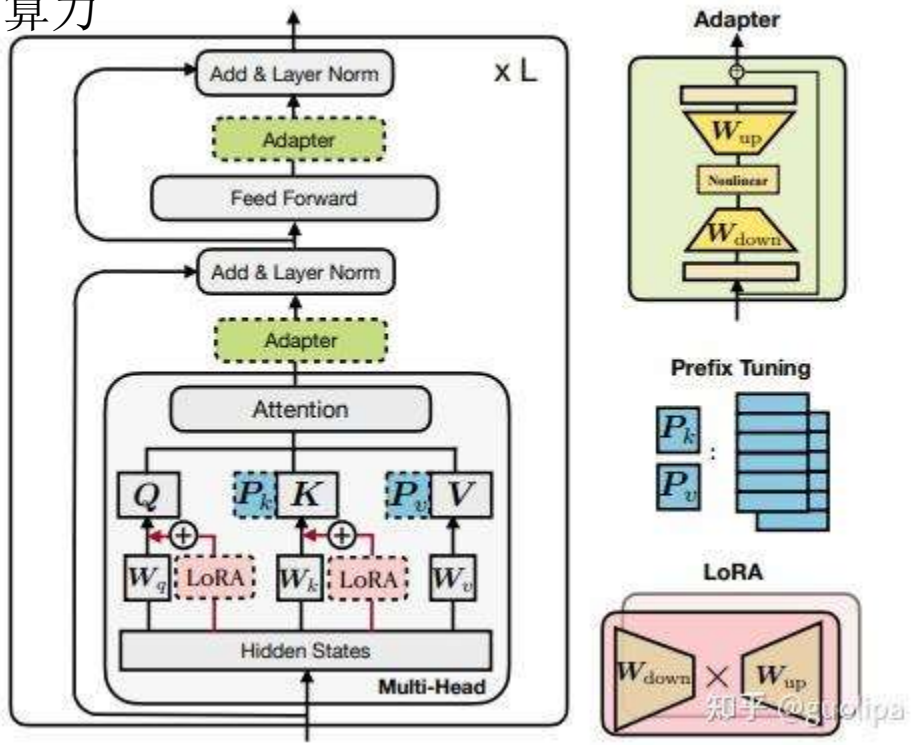
大模型调参难点，为什么很少直接微调？
1.参数多，显存不容易放下
2.参数多，需要对应更大数据
3.参数多，不容易收敛
4.参数多，调参时间过长

做大模型：
一半技术 一半运气

参数高效微调方法（Parameter-Efficient Fine-Tuning， PEFT）

- Prefix-Tuning / Prompt-Tuning：在模型的输入或隐层添加 k 个额外可训练的前缀 tokens（这些前缀是连续的伪 tokens，不对应真实的 tokens），只训练这些前缀参数；
- Adapter-Tuning：将较小的神经网络层或模块插入预训练模型的每一层，这些新插入的神经模块称为 adapter（适配器），下游任务微调时也只训练这些适配器参数；
- LoRA：通过学习小参数的低秩矩阵来近似模型权重矩阵 W的参数更新，训练时只优化低秩矩阵参数。

节约内存 节省算力



3大方法（外挂）
1.方法是大模型通用的
2.但是大模型是各自搞的
3.导致实现方法的代码不通用，需要特殊开发，或者找对应的

扫码了解更多

问题：今天能否打篮球

Prompt+问题：今天天晴，温度25度。今天能否打篮球

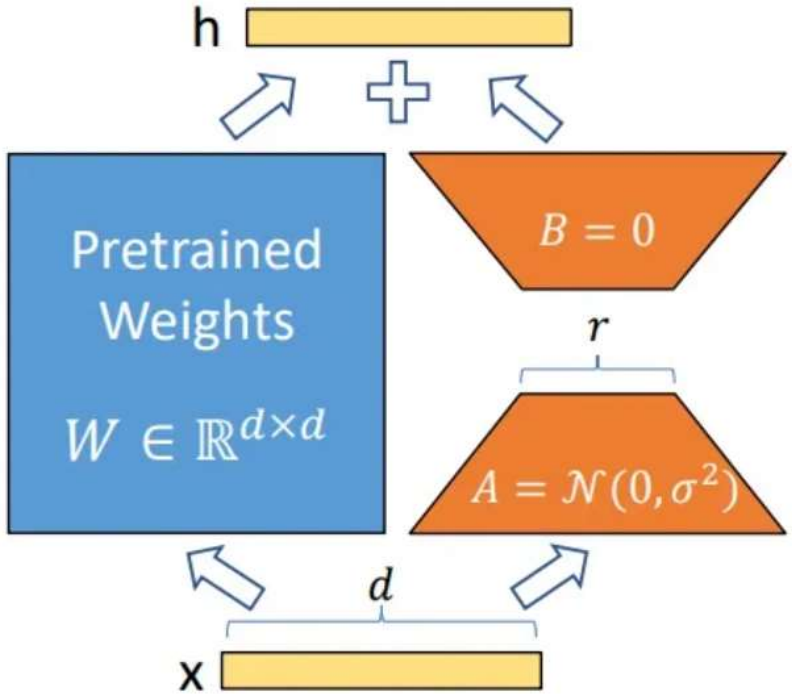
Prefix+问题：a b c d e今天能否打篮球 答案：今天可以打篮球

- Adapter Tuning 增加了模型层数，引入了额外的推理延迟
- Prefix-Tuning 难于训练，且预留给 Prompt 的序列挤占了下游任务的输入序列空间，影响模型性能
- P-tuning v2 很容易导致旧知识遗忘，微调之后的模型，在之前的问题上表现明显变差

扫码了解更多

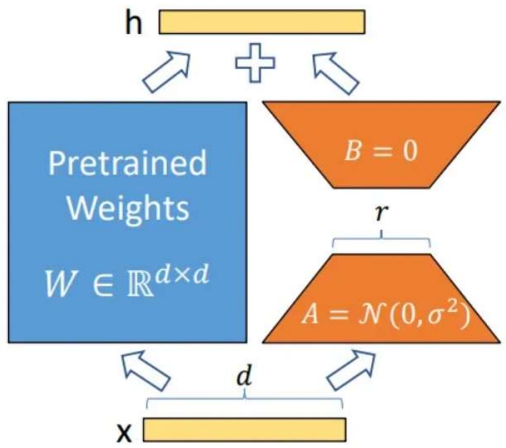
大模型中的LoRA

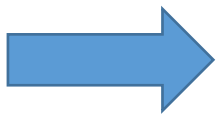
$$h = W_0 x + \Delta W x = W_0 x + BAx$$

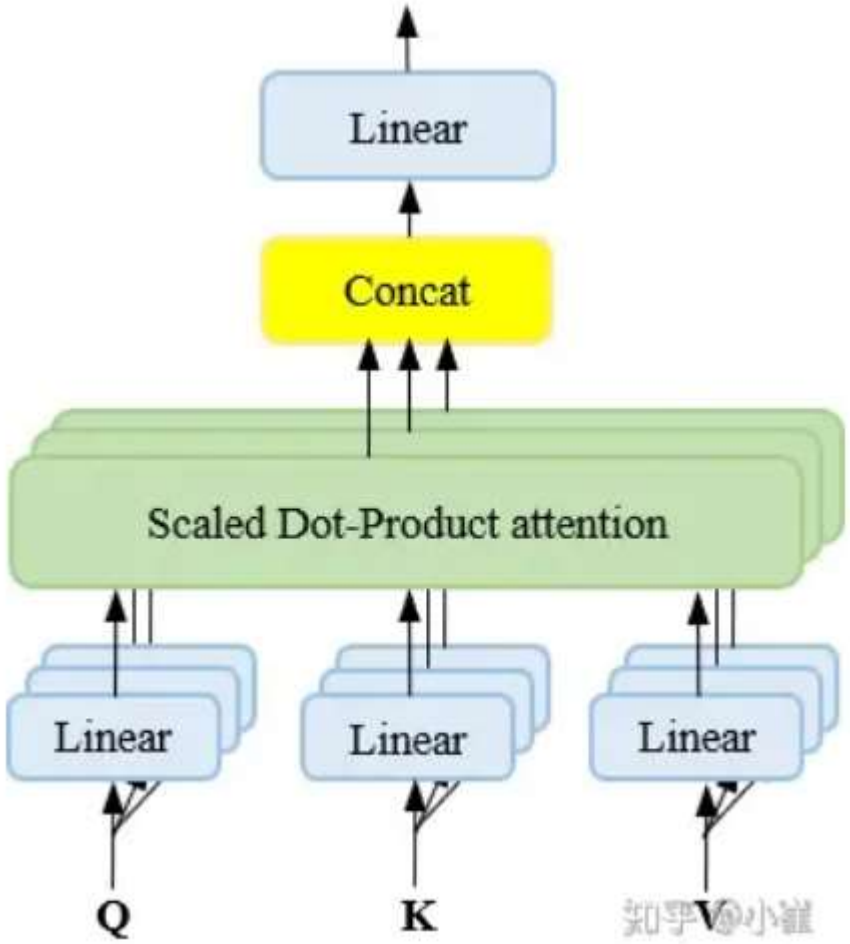


好处：节约内存

大模型中的LoRA







知乎@小崔

扫码了解更多

大模型中的LoRA

$$W = W_0 + UV, \qquad U \in \mathbb{R}^{m\times r}, V \in \mathbb{R}^{r\times n}$$

$$\frac{\partial \mathcal{L}}{\partial U} = \frac{\partial \mathcal{L}}{\partial W} V^\top, \quad \frac{\partial \mathcal{L}}{\partial V} = U^\top \frac{\partial \mathcal{L}}{\partial W}$$

$$U_{t+1} = U_t - \eta \frac{\partial \mathcal{L}}{\partial W_t} V_t^\top, \quad V_{t+1} = V_t - \eta U_t^\top \frac{\partial \mathcal{L}}{\partial W_t}$$

$$W_{t+1} = W_0 + U_{t+1}V_{t+1} = W_t + (U_{t+1}V_{t+1} - U_tV_t)$$

扫码了解更多

大模型中的LoRA

计算量并没有减少，但速度确更快，原因

1、只更新了部分参数：比如LoRA原论文就选择只更新Self Attention的参数，实际使用时我们还可以选择只更新部分层的参数；
2、减少了通信时间：由于更新的参数量变少了，所以（尤其是多卡训练时）要传输的数据量也变少了，从而减少了传输时间；
3、采用了各种低精度加速技术，如FP16、FP8或者INT8量化等。

大模型中的LoRA

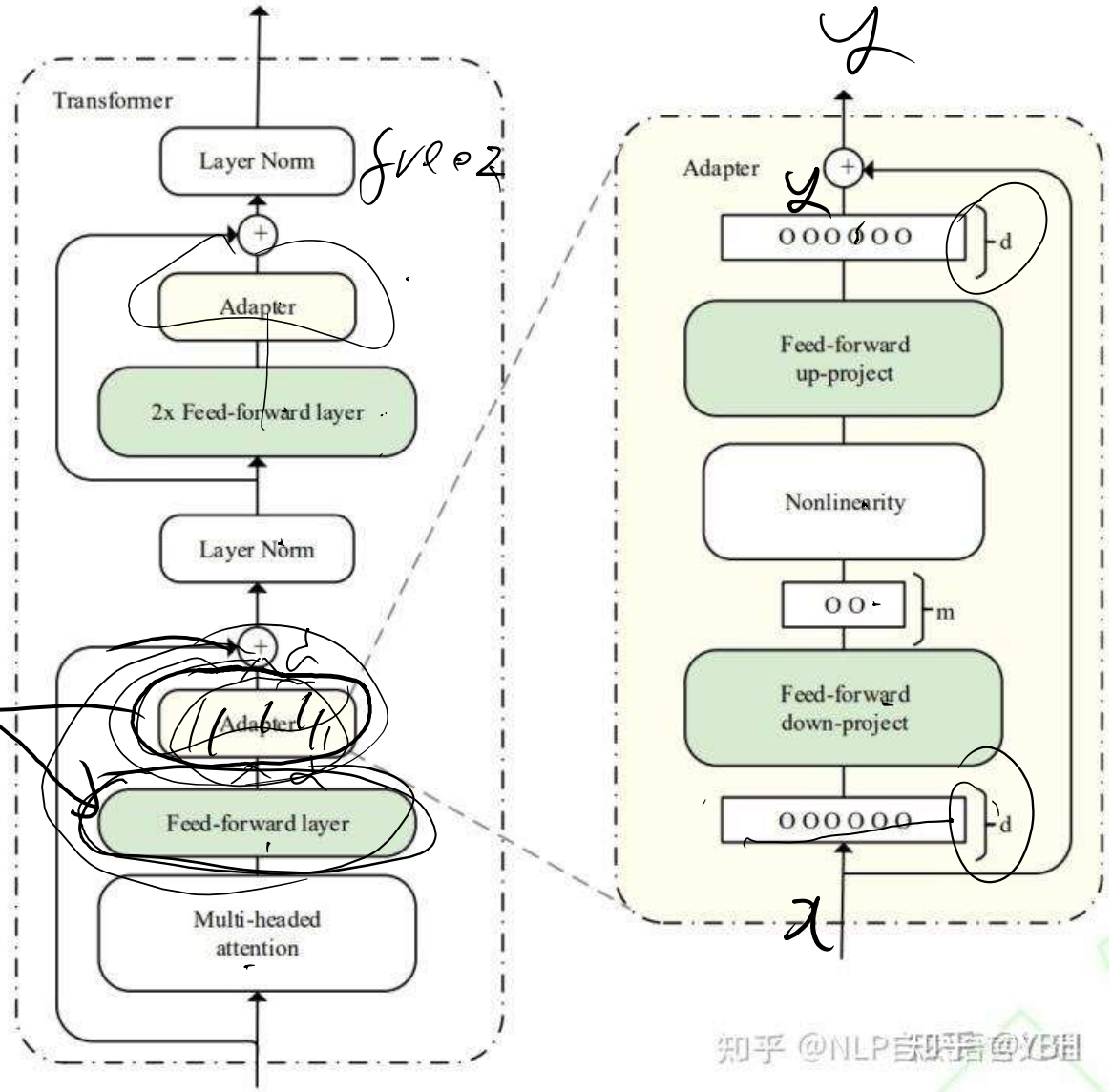
Rank确定方法

度量模型和垂直领域的差异性

大模型中的Adapter

```
rres=4096, out_features=4608, bias=True)
.0, inplace=False)
out_features=4096, bias=False)
)
s=4096, out_features=27392, bias=False)
s=13696, out_features=4096, bias=False)


ut_features=65024, bias=False)
```



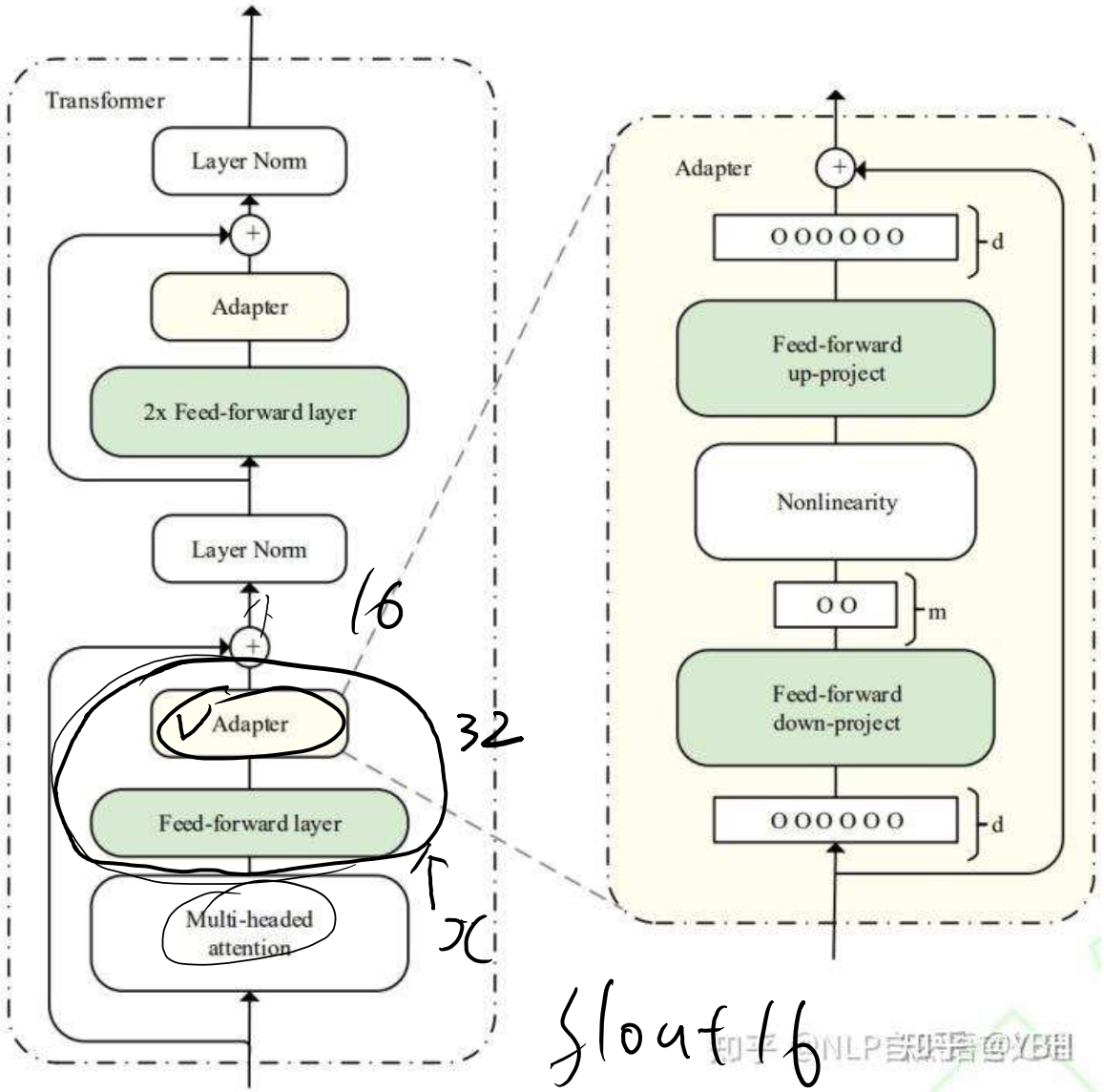
Chatglm没有adapter
Peft没有adapter
但是好处是
我有
我自己写代码实现了

resnet

$y = y + x$

$y = 0.1y + x$

知乎 @NLP



16

32

float16

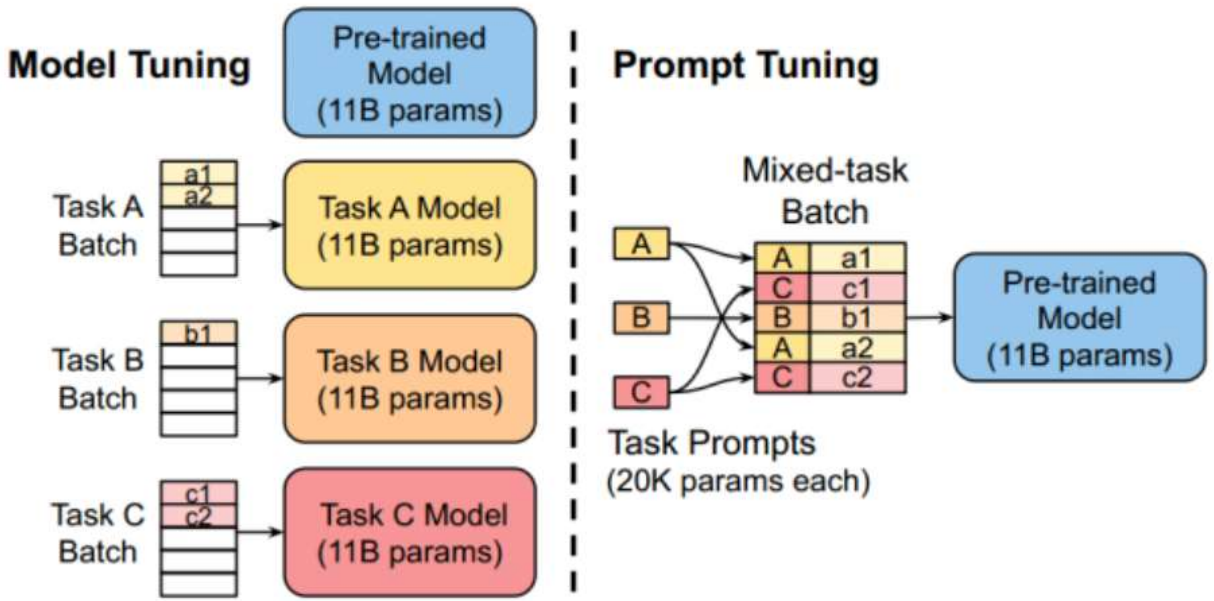
float32

扫码了解更多

大模型中的prompt Tuning

Prompt-Tuning（软提示/连续提示）

1. 可看做是Prefix-Tuning的简化版本，只在输入层加入prompt tokens，并不需要加入MLP进行调整
2. 提出 Prompt Ensembling 方法来集成预训练语言模型的多种 prompts
3. 在只额外对增加的3.6%参数规模（相比原来预训练模型的参数量）的情况下取得和Full-finetuning接近的效果
4. **作用阶段：**第一层transformer block的Attention注意力计算

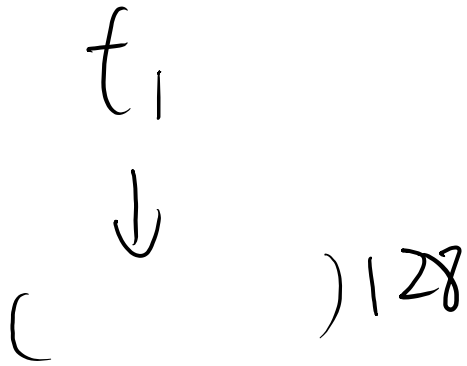


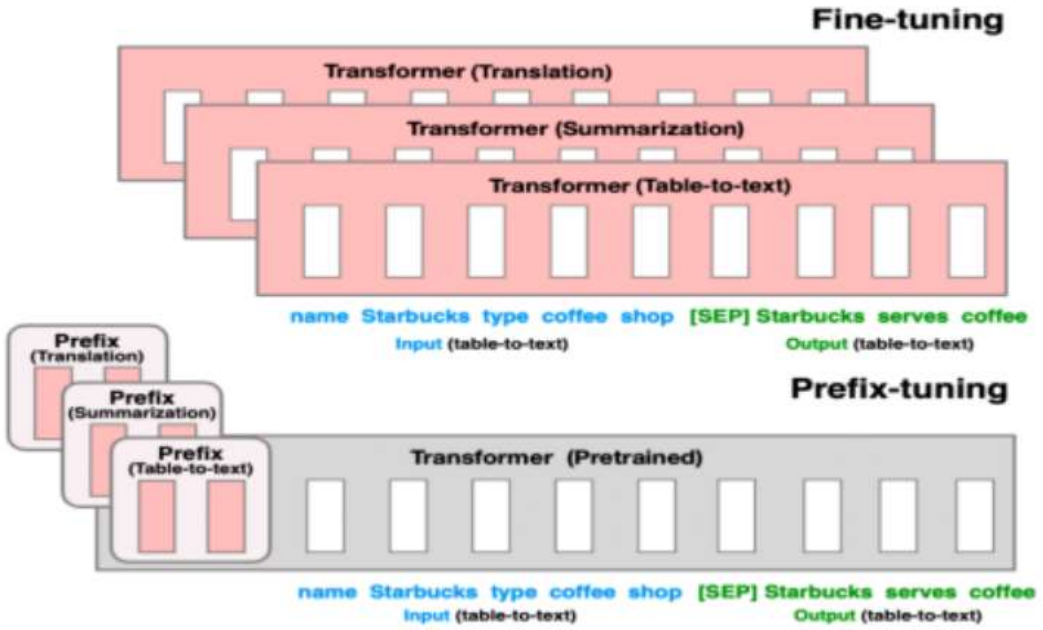
扫码了解更多

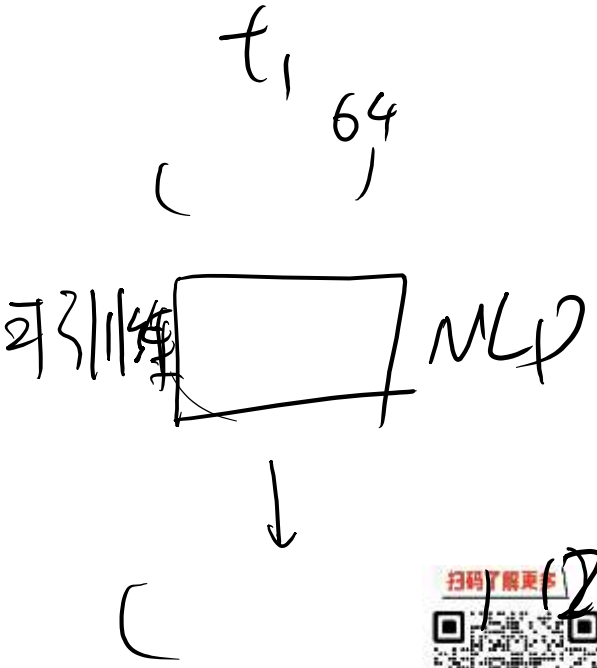
大模型中的Prefix Tuning

Prefix-Tuning（软提示/连续提示）

1. 在每一层的token之前构造一段任务相关的tokens作为Prefix，训练时只更新Prefix部分的参数，而Transformer中的其他部分参数固定
2. 一个Prefix模块就是一个可学习的id到embedding映射表，可以在多层分别添加Prefix模块
3. 为了防止直接更新Prefix的参数导致训练不稳定的情况，在Prefix层前面加了MLP结构
4. **作用阶段：**所有transformer block的Attention注意力计算



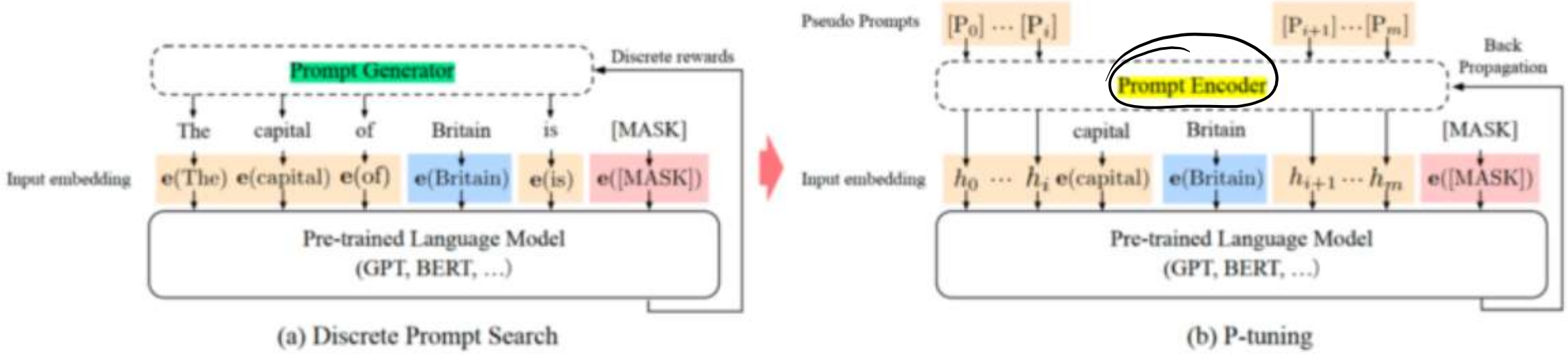




大模型中的P-Tuning

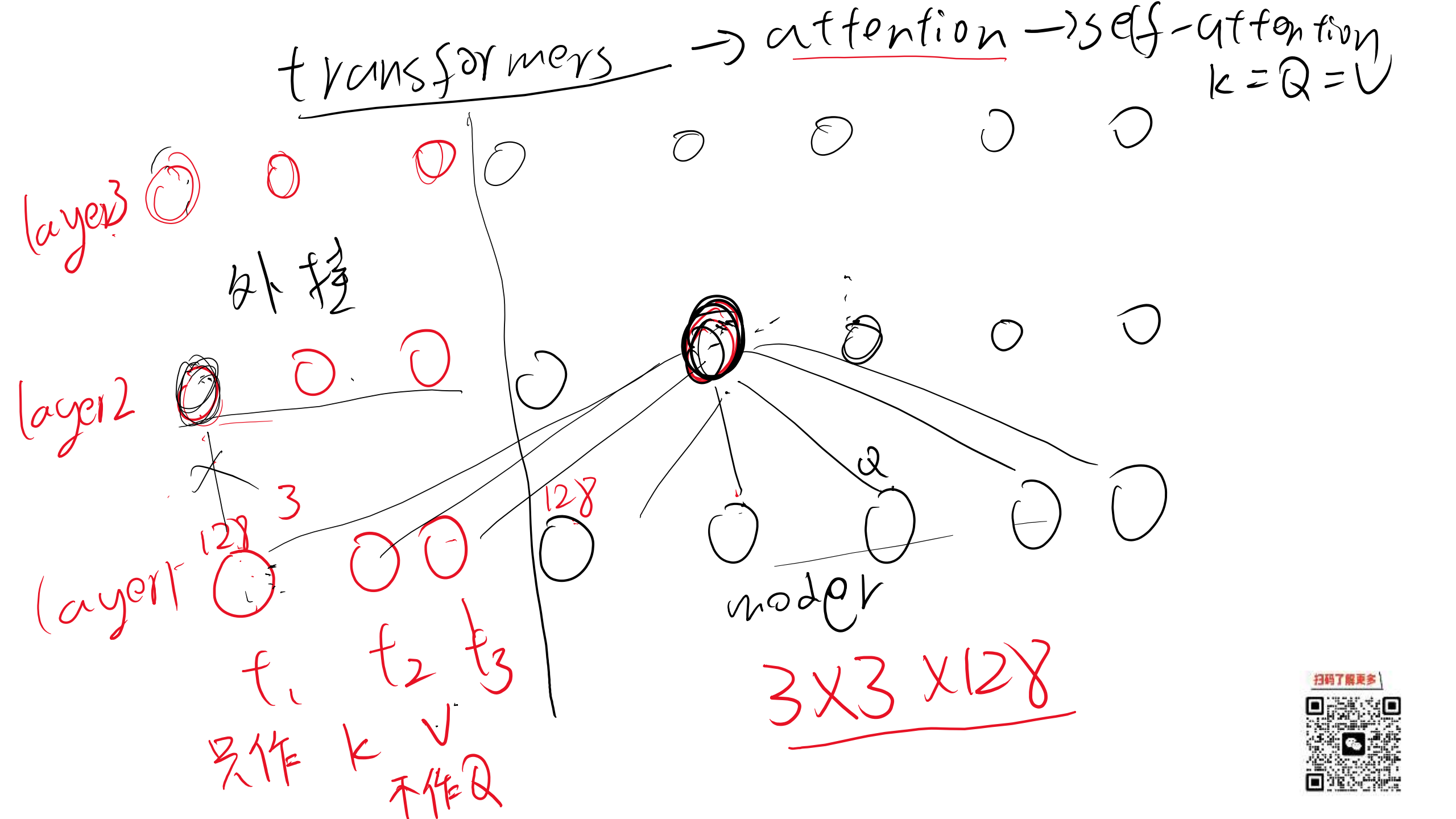
P-Tuning（软提示/连续提示）

1. P-Tuning只是在输入的时候加入Embedding，并通过LSTM+MLP对prompt embedding序列进行编码
2. 根据人工设计模板动态确定prompt token的添加位置，可以放在开始，也可以放在中间
3. **作用阶段：**第一层transformer block的Attention注意力计算



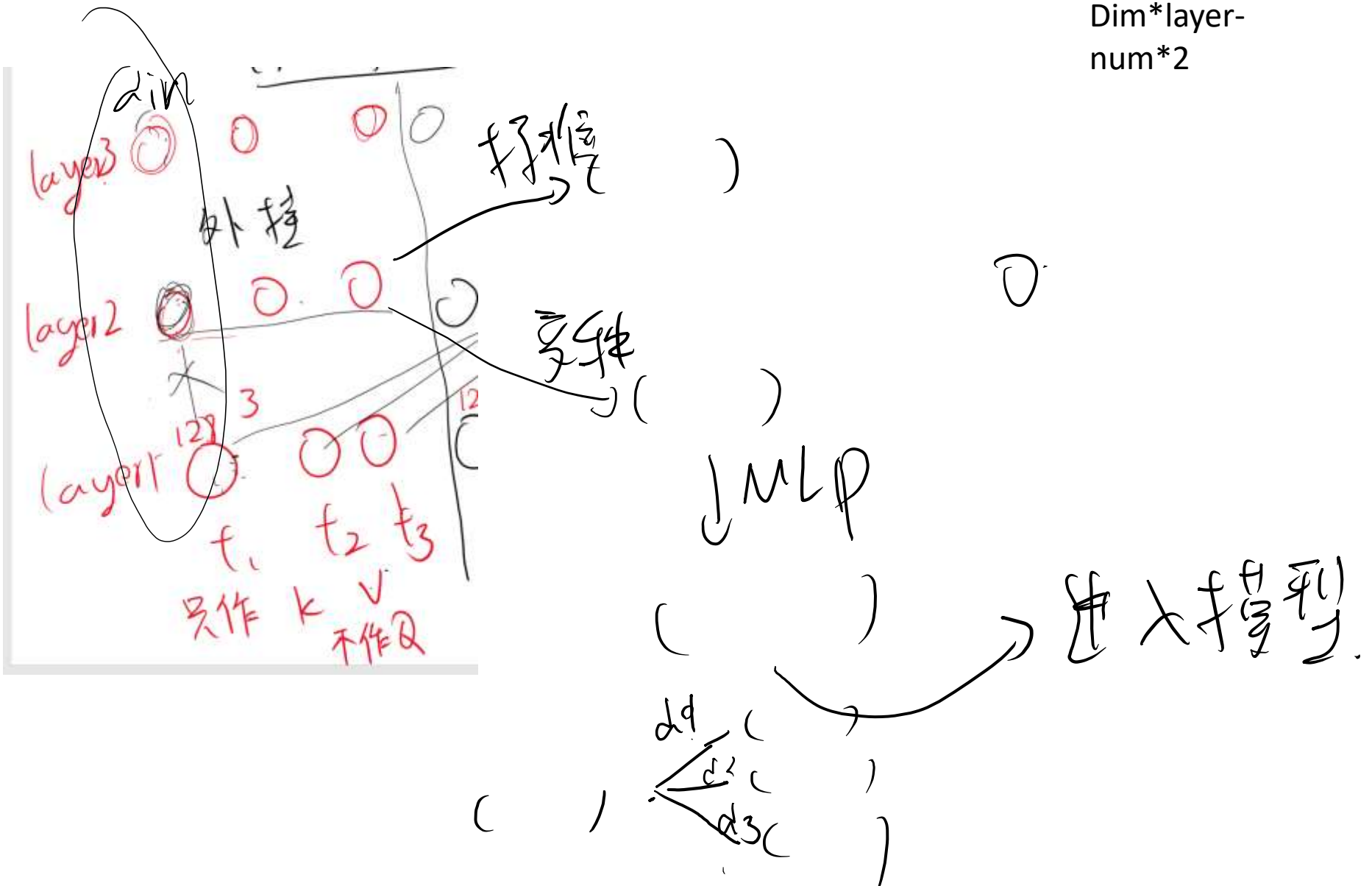
(a) Discrete Prompt Search

(b) P-tuning


transformers → attention → self-attention
k=Q=V
layer3
layer2
layer1
外接
3
128
128
model
q
t_1 t_2 t_3
只作 k v
不作Q
3X3 X128
扫码了解更多

课间休息

Dim*layer-num*2

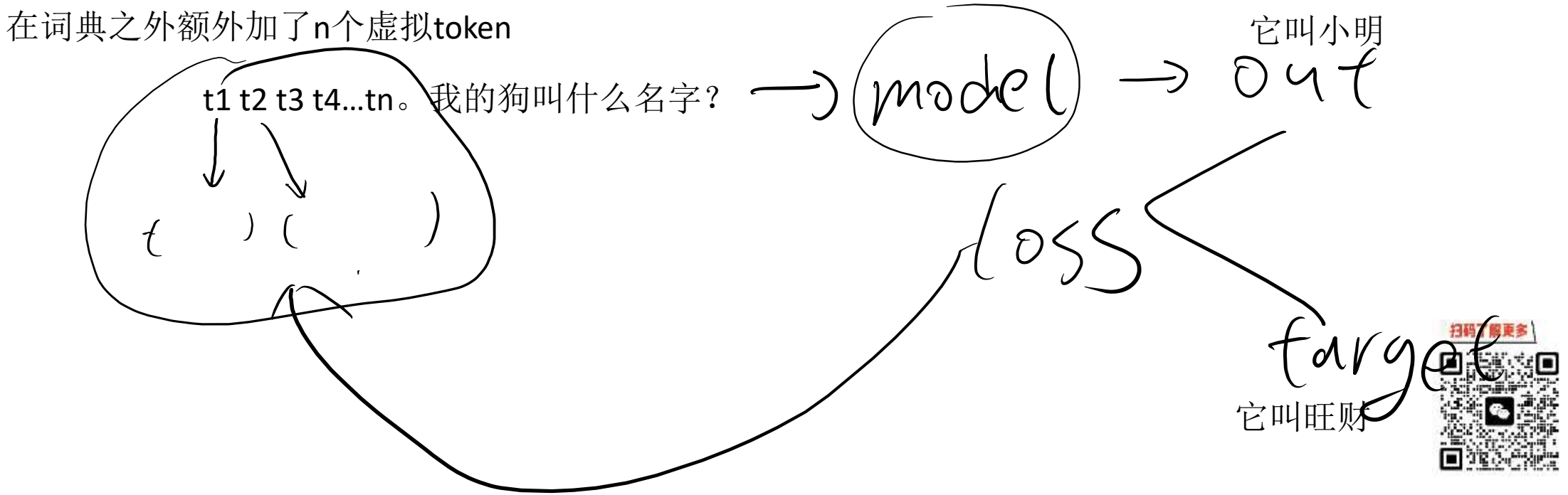


我养了只狗，它叫旺财。我的狗叫什么名字？

通过输入改变输出

模型本身的参数不变

在词典之外额外加了n个虚拟token



安卓充电器

一般来说：

大模型的prefixtuing的微调 都是用 transformers提供的peft库

- Prefix-Tuning / Prompt-Tuning
- LoRA

苹果充电器

但是清华的chatglm系列，采用了特殊的结构，和自由的命名规范

导致

可以用lora，但是不能用prefix-tuning

庆幸的是 清华自己实现了prefix-tuning

不巧的是，代码写的稀烂

好的是，我把代码梳理了，该删的都删了

扫码了解更多

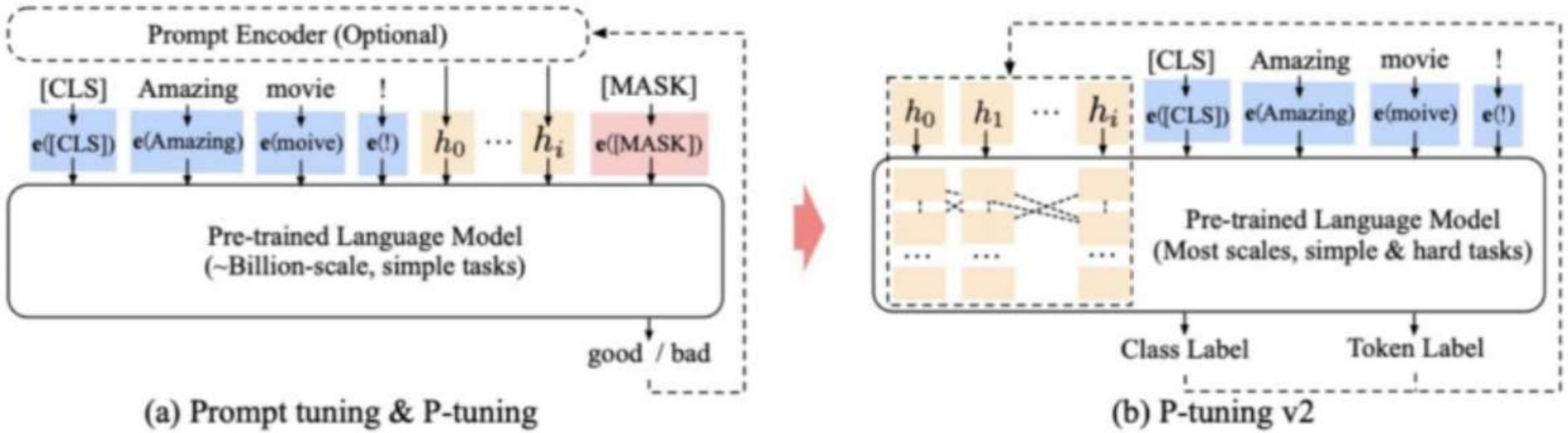
大模型中的P-Tuning

```
--pre_seq_len 128 ^
```

128：表示每层的虚拟token个数是128个

P-Tuning V2（软提示/连续提示）

1. 可看做是Prefix-Tuning的优化版本。在模型的每一层都添加连续的 prompts
2. P-Tuning v2在不同规模和任务中都可与微调效果相媲美
3. 移除重参数化的编码器（如：Prefix Tuning中的MLP、P-Tuning中的LSTM）、针对不同任务采用不同的提示长度、引入多任务学习等
4. **作用阶段：**所有transformer block的Attention注意力计算



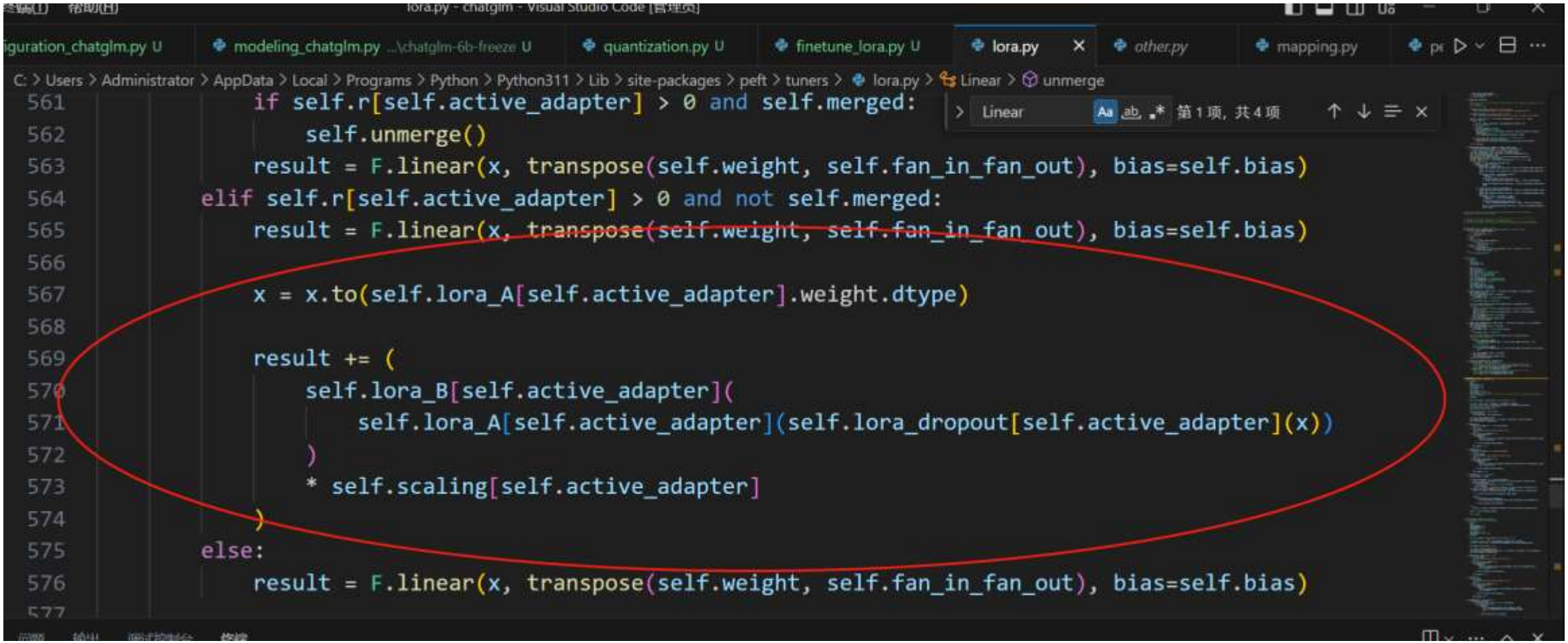
扫码了解更多

实验将五种方法进行对比，包括：Fine-Tuning (全量微调)、Bias-only or BitFit（只训练偏置向量）、Prefix-embedding tuning (PreEmbed，上文介绍的 Prefix Tuning 方法，只优化 embedding 层的激活)、Prefix-layer tuning (PreLayer，Prefix Tuning 方法，优化模型所有层的激活)、Adapter tuning（不同的 Adapter 方法：$\text{Adapter}^{\text{H}}$[10]、$\text{Adapter}^{\text{L}}$[11]、$\text{Adapter}^{\text{P}}$[12]、$\text{Adapter}^{\text{L}}$、$\text{Adapter}^{\text{D}}$[13]）

实验结果以 LoRA 在 GPT-3 175B 上的验证分析为例。如下表所示，**LoRA 在三个数据集上都能匹配或超过微调基准，证明了 LoRA 方法的有效性。**

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

扫码了解更多
知乎 @guolipa


lora.py - chatglm - Visual Studio Code [管理员]
figuration_chatglm.py U
modeling_chatglm.py ...\chatglm-6b-freeze U
quantization.py U
finetune_lora.py U
lora.py
other.py
mapping.py
C: > Users > Administrator > AppData > Local > Programs > Python > Python311 > Lib > site-packages > peft > tuners > lora.py > Linear > unmerge
Linear
第 1 项，共 4 项
561 if self.r[self.active_adapter] > 0 and self.merged:
562 self.unmerge()
563 result = F.linear(x, transpose(self.weight, self.fan_in_fan_out), bias=self.bias)
564 elif self.r[self.active_adapter] > 0 and not self.merged:
565 result = F.linear(x, transpose(self.weight, self.fan_in_fan_out), bias=self.bias)
566
567 x = x.to(self.lora_A[self.active_adapter].weight.dtype)
568
569 result += (
570 self.lora_B[self.active_adapter](
571 self.lora_A[self.active_adapter](self.lora_dropout[self.active_adapter](x))
572)
573 * self.scaling[self.active_adapter]
574)
575 else:
576 result = F.linear(x, transpose(self.weight, self.fan_in_fan_out), bias=self.bias)

扫码了解更多