从零实现大语言模型

Lec4：模型的训练与自回归解码

Penghao Kuang

SIST, ShanghaiTech
ACM, DataTech, Geekpie

2025.11.9

Overview

PART1
模型训练：宏观的系统架构

模型训练的过程概览

- 确定迭代优化总步数。对于每步迭代：
  - 从语料库中采样输入文本与标准输出
  - 利用语言模型前向传播
  - 利用前向传播输出与标准输出，计算当前损失函数
  - 根据损失函数，计算各参数的梯度并进行梯度裁剪
  - 根据当前步数进行学习率调度
  - 根据梯度、学习率等参数使用优化器进行参数更新
- 中途建议输出实时状态，如当前损失函数值等

需要实现的相关模块

- 输入输出 ⇒ 数据采样模块
- 模型输出与标准输出比较 ⇒ 交叉熵损失模块
- 参数梯度 ⇒ 梯度裁剪模块
- 参数优化 ⇒ 学习率调度模块、AdamW 优化器模块
- 检查点模块、传参预解码模块

PART2
数据采样模块

超大语料库的内存问题

回顾 BPE 板块：

- 训练词表与合并列表后……
- 将语料库编码成一个整数列表……
- 若语料库极为巨大，普通读写操作会将整个列表全部读入内存
- 是否存在一种解决方法，能够只将需要读写的特定部分内容从磁盘读入内存？

⇒numpy.memmap：一种能与磁盘空间直接读写交互的列表。其余方面与普通列表高度相似。

⇒ 实现一个 `class Memmap_Manager`，包含 `def save_as_memmap` 与、`def load_by_range`，即面向磁盘的读写操作

1.def save_as_memmap

- 使用 BPE_Tokenizer 的 encode_iterable，每次只读入单个整数编码（节省内存）。
- 设置列表 buffer 不断读入整数编码。
- buffer 到达一定大小后（如 50W 个编码）作为一个整块写入磁盘。
- buffer 清空，继续读入与整块写入，直至语料库读完。
- 中途可计数该语料库的整数编码总数

1.def save_as_memmap

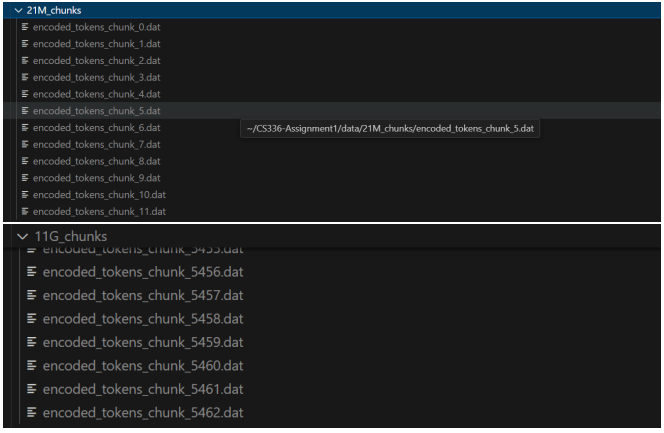
```python
def save_as_memmap(self):
    tokenizer=BPE_Tokenizer.from_files(self.vocab_path,self.merge_path,self.special_tokens)
    buffer=[]
    chunk_num=0
    length=0
    with open(self.corpus_path) as f:
        encoder=tokenizer.encode_iterable(f)
        for id in encoder:
            length+=1
            buffer.append(id)
            if len(buffer)>=self.chunk_size:
                self.save_by_chunks(buffer,self.chunk_size,chunk_num)
                chunk_num+=1
                buffer=[]
        if len(buffer)>0:
            self.save_by_chunks(buffer,len(buffer),chunk_num)
            buffer=[]
    print(f"length of corpus in tokens:{length}")
```

1.def save_as_memmap

- 约定数组类型，以确定读写的数据解析方式
- 约定数组规模 (shape)，以确定磁盘开辟空间大小
- 同普通数组一样操作，仍然存储在内存
- 显式 flush 操作后写入磁盘，先前内存清空

```python
def save_by_chunks(self,token_ids,buffer_len,chunk_num):
    fname="/home/kuangph/CS336-Assignment1/data/"+self.corpus_size+f"_chunks/encoded_tokens_chunk_{chunk_num}.dat"
    dtype=np.int32
    shape=(buffer_len,)
    memmap_arr = np.memmap(fname, dtype=dtype, mode="w+", shape=shape)
    memmap_arr[:] = token_ids[:]
    memmap_arr.flush()
```

1.def save_as_memmap



试图将如此大规模的列表加载到内存中是灾难性的！

2.def load_by_range

需求：需要读取编码列表中 [start_idx,end_idx) 的所有元素

- 计算：该区间覆盖了哪些切分块？
- 只从磁盘中加载相应块上的相应区间到内存中。
- 操作上除了声明 np.memmap 类型，其他与普通列表操作一致。

2.def load_by_range

`memmap_arr[idx_in_start:idx_in_end]`：只将此区间内容从磁盘加载到内存

```python
def load_by_range(self,start_idx,end_idx):
    chunk_size=self.chunk_size
    start_chunk=start_idx//chunk_size
    end_chunk=end_idx//chunk_size
    idx_in_start=start_idx%chunk_size
    idx_in_end=end_idx%chunk_size

    token_ids=[]
    for chunk in range(start_chunk,end_chunk+1):
        fname=f"/home/kuangph/CS336-Assignment1/data/"+self.corpus_size+f"_chunks/encoded_tokens_chunk_{chunk}.dat"
        dtype=np.int32
        memmap_arr=np.memmap(fname,dtype=dtype,mode="r")
        if start_chunk==end_chunk:
            token_ids.extend(memmap_arr[idx_in_start:idx_in_end])
        else:
            if chunk==start_chunk:
                token_ids.extend(memmap_arr[idx_in_start:])
            elif chunk>start_chunk and chunk<end_chunk:
                token_ids.extend(memmap_arr[:])
            else:
                token_ids.extend(memmap_arr[:idx_in_end])
    return token_ids
```

3. 正式采样操作：class Batch_By_Memmap

Problem1: 如何采样输入数据？

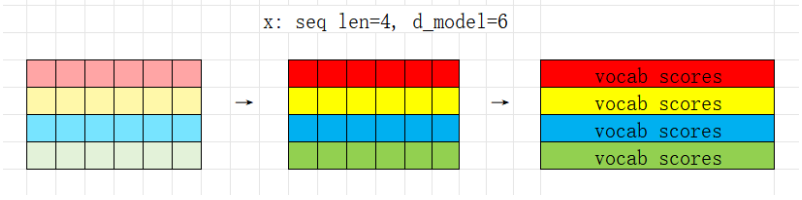
假设语料库长度为 10，单个采样序列长度为 4：

- 密集采样：1234；2345；3456；……
  - 数据量过大。若设置总训练步数不多（即采样次数不多），很有可能无法覆盖语料库后半段的数据。
- 接续采样：1234；5678；……
  - 部分数据缺失。如：无法获得 3456 作为训练输入数据。
- 折中方案：随机采样
  - 随机取起始点（保证不大于 7），从起始点开始连续采样 4 个字符。
  - 一般同时采样 batch size 个序列，则同时随机取 batch size 个起始点各自采样。

3. 正式采样操作：class Batch_By_Memmap

Problem2: 标准输出是什么？

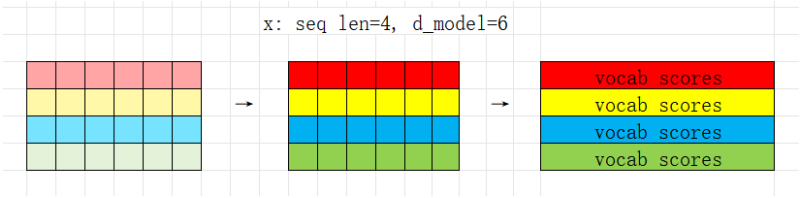
- 矩阵操作的本质是什么？
- 矩阵只是向量的机械拼接 ⇒ **关注单个词向量就等于关注整个矩阵！**
- 输入张量 x[*,seq_len,d_model] 经过模型，实际相当于 seq_len 个向量并行经过了同等规则的变换：
  - RMSNorm
  - 对前序 token 吸收注意力信息
  - FFN 升降维操作



3. 正式采样操作：class Batch_By_Memmap

Problem2: 标准输出是什么？

- 每个向量最终线性投影至 vocab_size 规模权重，以此推断下一位置生成什么词
- 所以标准输出应当为语料库中当前位置的下一词！
- 即：如果输入序列位置为 1234，则标准输出为 2345。



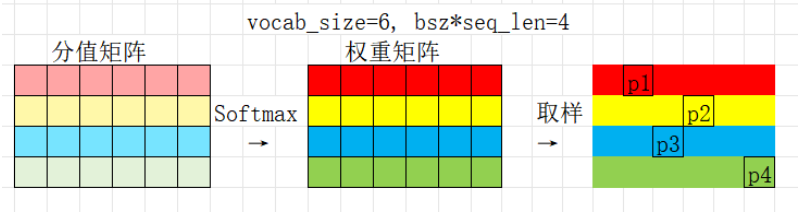
3. 正式采样操作：class Batch_By_Memmap

```python
class Batch_By_Memmap:
    def __init__(self,memmep_manager:Memmap_Manager):
        self.memmap_manager=memmep_manager

    def get_batch(self,bsz,seq_len,dataset_length,device=None):
        max_start_idx=dataset_length-seq_len
        start_indices=np.random.randint(0,max_start_idx,bsz)

        x=np.array([self.memmap_manager.load_by_range(i,i+seq_len) for i in start_indices],dtype=np.int64)
        y=np.array([self.memmap_manager.load_by_range(i+1,i+seq_len+1) for i in start_indices],dtype=np.int64)

        x=torch.tensor(x,dtype=torch.long,device=device)
        y=torch.tensor(y,dtype=torch.long,device=device)
        return (x,y)
```

PART3
训练辅助模块

1.class Cross_Entropy_Calculator

- Transformer 输出规模为 [bsz,seq_len,vocab_size]。即：共有 (bsz*seq_len) 个输出预测，每个预测都是“下一位置各词的可能性分值” s_i $(1 \leq i \leq (bsz * seq\_len))$，下同
- 对 (bsz*seq_len) 个可能性做 Softmax 归一化，得到可能性权重分布 p_i
- 对所有正确取样位置的可能性权重取-log 得 l_j，求均值即为交叉熵损失
- 即：$Loss = Mean\{-logp_j\} = Mean\{-logSoftmax(s_j)\}$



1.class Cross_Entropy_Calculator

数值隐患：如果 p_j 过小被近似为 0？
（如 token1 的 6 词权重分数为 100，-1,1,5,2,1000）
$\Rightarrow$ 计算结果为：$log0 = NAN!$
公式推导：

$$logSoftmax(s_j) = log\frac{exp(s_j)}{\sum expss_k}$$
$$= log\frac{exp(s_j - s_{max})}{\sum exp(s_k - s_{max})}$$
$$= (s_j - s_{max}) - log(\sum exp(s_k - s_{max}))$$

1.class Cross_Entropy_Calculator

```python
class Log_Softmax():
    def __init__(self,dim:int=-1):
        self.dim=dim

    def forward(self,x:torch.Tensor)->torch.Tensor:
        x_max=torch.max(x,dim=self.dim,keepdim=True).values
        x=x-x_max
        x_exp=torch.exp(x)
        x_exp_sum=torch.sum(x_exp,dim=self.dim,keepdim=True)
        return x-torch.log(x_exp_sum)
```

1.class Cross_Entropy_Calculator

```python
class Cross_Entropy_Calculator:
    def __init__(self):
        self.log_softmax=Log_Softmax(dim=-1)

    def forward(self,inputs:torch.Tensor,targets:torch.Tensor)->torch.Tensor:
        inputs=inputs.reshape(-1,inputs.shape[-1])
        inputs=-self.log_softmax.forward(inputs)
        targets=targets.reshape(-1)

        selected=inputs[torch.arange(inputs.shape[0]),targets]
        loss=torch.mean(selected,dim=0)
        return loss
```

2.class Gradient_Clipper

假设一个模型的所有参数来源为 5 个 Linear_Transform(5D to 5D)，则该模型共有 5 个参数张量（均为 5*5 规模）
假设一轮反向传播过后，5 个参数张量的梯度均为：

```
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

令参数张量为 p_1, p_2, p_3, p_4, p_5，则我们认为总 L2 范数为：$\sqrt{\sum \|p_i\|_2} = 11.18$

2.class Gradient_Clipper

- 梯度过大往往会导致优化方向偏离
- 我们需要控制参数梯度的总 L2 范数大小 g
- 假设我们可接受的上限为 max_norm=0.01：
- 所有参数必须乘缩放系数 $\frac{max\_norm}{g+\epsilon}$，以保证梯度总 L2 范数大小合理
- 裁剪后各参数的梯度为：

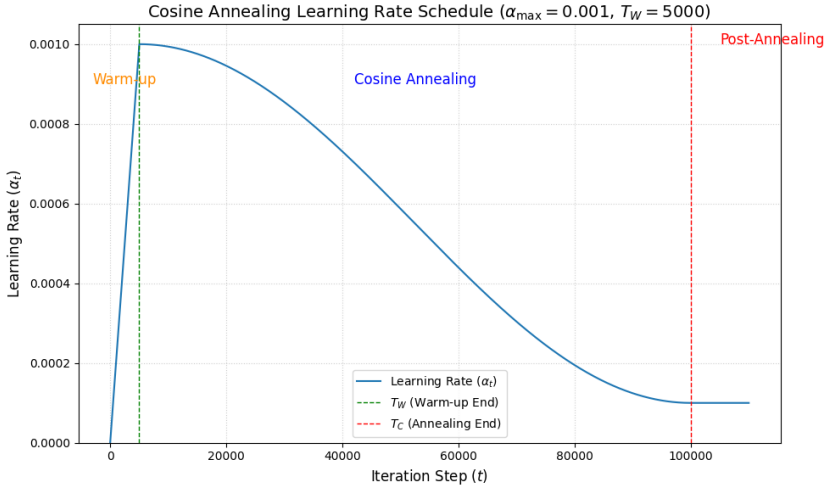
```
tensor([[0.0009, 0.0009, 0.0009, 0.0009, 0.0009],
        [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],
        [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],
        [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],
        [0.0009, 0.0009, 0.0009, 0.0009, 0.0009]])
```

2.class Gradient_Clipper

```python
class Gradient_Clipper:
    def __init__(self,max_norm:float):
        self.max_norm=max_norm
    def clip(self,parameters):
        total_norm=torch.sqrt(sum(p.grad.data.norm(2)**2 for p in parameters if p.grad is not None))
        for p in parameters:
            if p.grad is None:
                continue
            if total_norm<=self.max_norm:
                continue
            clip_factor=self.max_norm/(total_norm+1e-6)
            p.grad.data=p.grad.data*clip_factor
```

3.class Learning_Rate_Scheduler

在模型训练的全过程中，学习率并非一成不变：



3.class Learning_Rate_Scheduler

```python
class Learning_Rate_Scheduler:
    def __init__(self):
        pass
    def get_lr(self,step,lr_max,lr_min,Tw,Tc)->float:
        if step<Tw:
            lr=lr_max*step/Tw
        elif step>Tc:
            lr=lr_min
        else:
            lr=lr_min+0.5*(1+math.cos(math.pi*(step-Tw)/(Tc-Tw) ) )*(lr_max-lr_min)
        return lr
```

PART4
AdamW 优化器

1.torch.optim.Optimizer 基类

任何自定义优化器的实现都继承自 `torch.optim.Optimizer` 基类
Optimizer 基类提供对于模型所有参数的两级管理：

- `self.param_groups`：对参数分组。每组参数可设置不同的学习率等特征。（第一级管理：参数分组，组内共享某些状态）
- 每个 `param_groups` 中至少有默认键值对：“params”，对应一个参数列表。此外还可自定义“lr”等键值对。
- 对于参数列表中的每个参数，也可设置自身的各种状态。如迭代步数“step”。（第二级管理：参数自身更精细的状态）

1.torch.optim.Optimizer 基类

```python
class AdamW_Optimizer(torch.optim.Optimizer):
    def __init__(self,parameters,lr:float,weight_decay:float,betas,eps:float):
        param_groups=[
            {
                "params":parameters,
                "lr":lr
            }
        ]
        super(AdamW_Optimizer,self).__init__(param_groups,{})
        self.weight_decay=weight_decay
        self.beta1=betas[0]
        self.beta2=betas[1]
        self.eps=eps

        for group in self.param_groups:
            for p in group["params"]:
                self.state[p]={
                    "m":torch.zeros_like(p.data),
                    "v":torch.zeros_like(p.data),
                    "step": torch.tensor(0.0, device=p.device)
                }
```

2.class AdamW_Optimizer

Algorithm 1 AdamW Optimizer

init(θ) (Initialize learnable parameters)
$m \leftarrow 0$ (Initial value of the first moment vector; same shape as θ)
$v \leftarrow 0$ (Initial value of the second moment vector; same shape as θ)
for $t = 1, \ldots, T$ **do**
  Sample batch of data B_t
  $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (Compute the gradient of the loss at the current time step)
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (Update the first moment estimate)
  $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (Update the second moment estimate)
  $\alpha_t \leftarrow \alpha \frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ (Compute adjusted α for iteration t)
  $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$ (Update the parameters)
  $\theta \leftarrow \theta - \alpha\lambda\theta$ (Apply weight decay)
end for

```python
def step(self):
    for group in self.param_groups:
        for p in group['params']:
            if p.grad is None:
                continue

            grad=p.grad.data
            state=self.state[p]
            m,v,step=state["m"],state["v"],state["step"]

            if not isinstance(step,torch.Tensor):
                step=torch.tensor(float(step),device=p.device)
                state["step"]=step

            current_lr=group.get("lr")

            m=m*self.beta1+(1-self.beta1)*grad
            v=v*self.beta2+(1-self.beta2)*(grad**2)
            step=step+1

            alpha_t=current_lr*(math.sqrt(1-self.beta2**step)/(1-self.beta1**step))
            p.data=p.data-alpha_t*(m/(torch.sqrt(v)+self.eps))-current_lr*self.weight_decay*p.data

            self.state[p]["m"]=m
            self.state[p]["v"]=v
            self.state[p]["step"]=step
```

PART5
检查点

1. 如何获取所有参数？

- torch 中自带 state_dict() 功能，可将该 torch 模块的所有参数以字典形式存储

```
lm=Transformer_LM(d_model=512,
                  num_heads=8,
                  d_ff=2048,
                  vocab_size=10000,
                  num_layers=2,
                  max_seq_length=128,
                  theta=100000,
                  dtype=torch.float32,
                  device="cpu")
states=lm.state_dict()
for state_key in states:
    print(state_key,states[state_key].shape)
```

```
transformer_blocks.0.RMSNorm_Attn.g torch.Size([512])
transformer_blocks.0.RMSNorm_FF.g torch.Size([512])
transformer_blocks.0.Multihead_Attn.q_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.0.Multihead_Attn.k_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.0.Multihead_Attn.v_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.0.Multihead_Attn.o_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.0.Multihead_Attn.rope.cos_values torch.Size([1, 128, 32])
transformer_blocks.0.Multihead_Attn.rope.sin_values torch.Size([1, 128, 32])
transformer_blocks.0.Feed_Forward.linear_w1.linear_matrix torch.Size([512, 2048])
transformer_blocks.0.Feed_Forward.linear_w3.linear_matrix torch.Size([512, 2048])
transformer_blocks.0.Feed_Forward.linear_w2.linear_matrix torch.Size([2048, 512])
transformer_blocks.1.RMSNorm_Attn.g torch.Size([512])
transformer_blocks.1.RMSNorm_FF.g torch.Size([512])
transformer_blocks.1.Multihead_Attn.q_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.1.Multihead_Attn.k_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.1.Multihead_Attn.v_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.1.Multihead_Attn.o_proj.linear_matrix torch.Size([512, 512])
transformer_blocks.1.Multihead_Attn.rope.cos_values torch.Size([1, 128, 32])
transformer_blocks.1.Multihead_Attn.rope.sin_values torch.Size([1, 128, 32])
transformer_blocks.1.Feed_Forward.linear_w1.linear_matrix torch.Size([512, 2048])
transformer_blocks.1.Feed_Forward.linear_w3.linear_matrix torch.Size([512, 2048])
transformer_blocks.1.Feed_Forward.linear_w2.linear_matrix torch.Size([2048, 512])
final_norm.g torch.Size([512])
final_layer.linear_matrix torch.Size([512, 10000])
```

2. 如何加载所有参数？

- torch 中自带 load_state_dict() 功能，可将该 torch 模块的所有参数的值从一个 state_dict 字典当中加载。前提是二者能完全匹配！

3.class Checkpoint_Manager

- 需要存储的对象：模型参数、优化器参数、当前迭代步数

```python
def save(self,model,optimizer,iteration,save_path):
    import os
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    state_model=model.state_dict()
    state_optimizer=optimizer.state_dict()
    checkpoint={
        "model":state_model,
        "optimizer":state_optimizer,
        "iteration":iteration
    }
    torch.save(checkpoint,save_path)
```

3.class Checkpoint_Manager

- 需要加载的对象：模型参数、优化器参数、当前迭代步数

```
def load(self,src_path,model,optimizer=None):
    checkpoint=torch.load(src_path)
    state_model=checkpoint["model"]
    if optimizer is not None:
        print(f"optimizer is not none")
        state_optimizer=checkpoint["optimizer"]
    iteration=checkpoint["iteration"]

    model.load_state_dict(state_model)
    if optimizer is not None:
        optimizer.load_state_dict(state_optimizer)
    return iteration
```

PART6
训练脚本的启动与参数解码

到此为止……

我们梳理出了：

- 模型各模块的架构与传参
- 训练各模块的实现与传参
- 关于传参……

我们如何规定这些参数的具体值，并有效传给各个模块？

1. 训练脚本的启动

- 我们可以直接在训练脚本中为各参数赋值
- 当训练脚本非常复杂，各功能模块将变得含混不清
- 我们习惯专门用一个 bash 脚本承载需要的参数值，并启动训练脚本的运行

1. 训练脚本的启动

```
#!/bin/bash
python /home/kuangph/CS336-Assignment1/cs336_basics/run_clm.py \
    --d_model 512 \
    --num_heads 8 \
    --d_ff 1344 \
    --vocab_size 32000 \
    --num_layers 8\
    --max_seq_length 256 \
    --seq_length 256 \
    --batch_size 48 \
    --theta 100000 \
    --device cuda \
    --num_epochs 1.5 \
    --lr 1e-4 \
    --lr_min 1e-5 \
    --warmup_ratio 0.05 \
    --warmfix_ratio 0.9 \
    --chunk_size 500000 \
    --vocab_path /home/kuangph/CS336-Assignment1/data/vocab_32000.txt \
    --merges_path /home/kuangph/CS336-Assignment1/data/merges_32000.txt \
    --special_tokens "<|endoftext|>" \
    --corpus_size "5M" \
    --log_interval 100 \
    --save_interval 500 \
    --weight_decay 0.01 \
    --betas 0.9 0.95 \
    --eps 1e-8 \
    --max_norm 1.0
```

- 相当于在终端输入了一个完整的命令行（python 运行及附带的参数）
- 对于运行的 python 程序（训练脚本），需要有将附带参数解码为实际值的模块

2. 参数解码

```python
def parse_bash_args():
    parser=argparse.ArgumentParser()

    parser.add_argument("--d_model",type=int,default=512)
    parser.add_argument("--num_heads",type=int,default=8)
    parser.add_argument("--d_ff",type=int,default=1344)
    parser.add_argument("--vocab_size",type=int,default=32000)
    parser.add_argument("--num_layers",type=int,default=6)
    parser.add_argument("--max_seq_length",type=int,default=512)
    parser.add_argument("--seq_length",type=int,default=256)
    parser.add_argument("--batch_size",type=int,default=32)
    parser.add_argument("--theta",type=int,default=100000)
    parser.add_argument("--device",type=str,default="cuda")

    parser.add_argument("--num_epochs",type=float,default=10)
    parser.add_argument("--lr",type=float,default=1e-4)
    parser.add_argument("--lr_min",type=float,default=1e-5)
    parser.add_argument("--warmup_ratio",type=float,default=0.1)
    parser.add_argument("--warmfix_ratio",type=float,default=0.9)
```

```python
parser.add_argument("--chunk_size",type=int,default=500000)
parser.add_argument("--vocab_path",type=str,default="data/vocab_32000.txt")
parser.add_argument("--merges_path",type=str,default="data/merges_32000.txt")
parser.add_argument("--special_tokens", type=str, nargs="*", default=["<|endoftext|>"])

parser.add_argument("--corpus_size",type=str)

parser.add_argument("--log_interval",type=int)
parser.add_argument("--save_interval",type=int)

parser.add_argument("--weight_decay",type=float,default=0.01)
parser.add_argument("--betas",type=float, nargs="*", default=(0.9,0.95))
parser.add_argument("--eps",type=float,default=1e-8)

parser.add_argument("--max_norm",type=float,default=1.0)

args=parser.parse_args()
return args
```

2. 参数解码

```
args=parse_bash_args()

d_model=args.d_model
num_heads=args.num_heads
d_ff=args.d_ff
vocab_size=args.vocab_size
num_layers=args.num_layers
max_seq_length=args.max_seq_length
seq_length=args.seq_length
batch_size=args.batch_size
theta=args.theta
dtype=torch.float32
device=args.device
```

PART7
完整训练脚本的编写

模型训练的过程概览

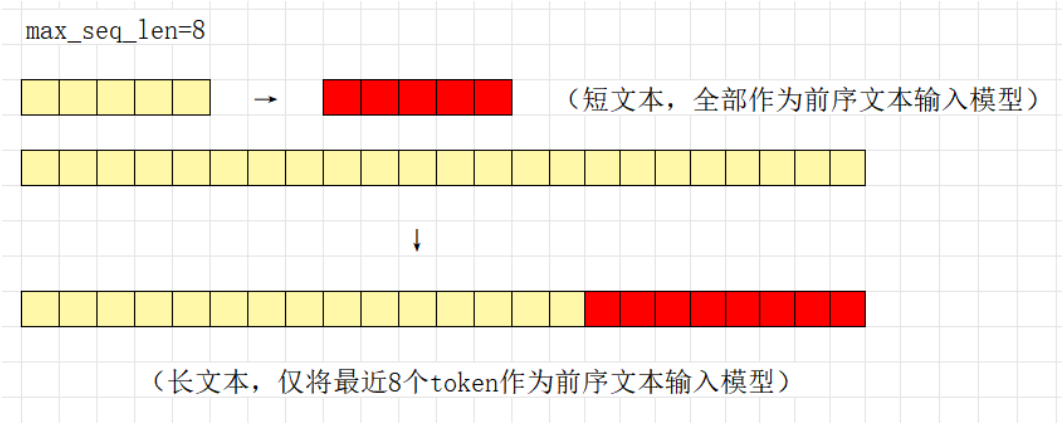
- 接收参数，建立各模型模块与训练模块
- 确定迭代优化总步数，开始逐步迭代。
- 对于每步迭代：
  - 从语料库中采样输入文本与标准输出
  - 利用语言模型前向传播
  - 利用前向传播输出与标准输出，计算当前损失函数
  - 根据损失函数，计算各参数的梯度并进行梯度裁剪
  - 根据当前步数进行学习率调度
  - 根据梯度、学习率等参数使用优化器进行参数更新
- 中途建议输出实时状态，如当前损失函数值、当前文本预测情况等

代码讲解与实操演示

PART8
自回归解码

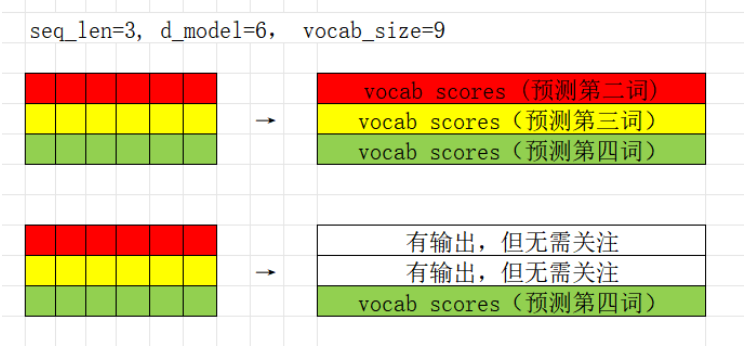
输入文本的确定

训练过程中，往往采样固定长度的文本输入模型。但解码文本总长度可能无法预知。



输出值的取样

训练过程中，我们需要获知所有位置的输出以计算损失值；
解码过程中，我们只需关注“下一词是什么”！

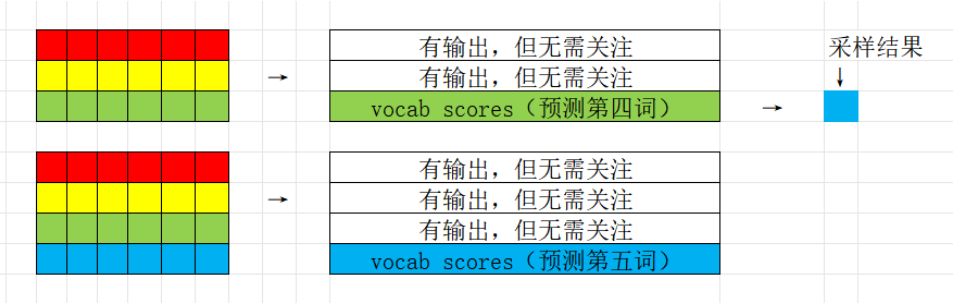


从 vocab scores 到 token 采样

接收到 vocab scores 后：

- 重复惩罚：最近刚出现的 token 分数削减，以避免重复输出同一个词
- 温度采样：将所有 scores 同时缩放相同倍数，使得 softmax 差距更大/更小
- Softmax：将 scores 转化成概率分布
- 按照分布的概率随机采样，作为下一个 token

从 vocab scores 到 token 采样



代码讲解与实操演示

谢谢！