【手搓大模型】从零训练GPT2

前文我们已经实现了完整的GPT-2代码，并用来生成文本，只不过模型在胡言乱语。现在我们开始训练模型，通过在本地电脑上进行极小样本的训练来给模型注入一点点智能。

本文示例中，我们使用了Wikipedia上关于world war II的公开介绍，仅有10000单词。如前文所述，训练样本是不需要任何打标的(pretrain on unlabeled data)，训练过程是自回归过程，targets总是比inputs向右偏移1个token。

Understand training targets

对于训练而言，最重要的是理解训练的目标。

我们直接读取样本，每次batch有2行，context size是4，加载训练样本，如下：

代码块

```
from gpt2_v1 import dataloader_v1

with open("world_war_ii.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
dataloader = dataloader_v1(raw_text,batch_size=2, context_size=4,stride=1)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("shape of input: ",inputs.shape)
print("first step, input: \n", inputs,"\n targets: \n", targets)
```

```
shape of input:  torch.Size([2, 4])
first step, input:
 tensor([[10603,  1810,   314,   393],
        [ 1810,   314,   393,   262]])
 targets:
tensor([[1810,  314,  393,  262],
        [ 314,  393,  262, 3274]])
```

可见，input和targets都是2x4；且targets比inputs偏移1。

让我们反查词表，把token id转换成文本，方便查看，如下：

代码块

```
from gpt2_v1 import tensor_to_text,build_tokenizer
tokenizer = build_tokenizer()
for i in range(inputs.size(0)):
    text = tensor_to_text(inputs[i].unsqueeze(0), tokenizer)
    print(f"Input {i}: {text}")

for i in range(targets.size(0)):
    text = tensor_to_text(targets[i].unsqueeze(0), tokenizer)
    print(f"target {i}: {text}")
```

> Input 0: World War I or
>
> Input 1: War I or the
>
> target 0: War I or the
>
> target 1: I or the First

可见，targets刚好比inputs偏移1个token。

现在，我们把inputs输入模型，进行生成，如下：

代码块

```
with torch.no_grad():
    logits = model(inputs)
probas = torch.softmax(logits, dim=-1)
print("shape of logits: ",logits.shape)
print("shape of probas: ",probas.shape)
```

> shape of logits: torch.Size([2, 4, 50257])
>
> shape of probas: torch.Size([2, 4, 50257])

得到的logits和softmax后的probas输出均是[2, 4, 50257]，而probas代表的是2行4列总共8个token在50257维的词表中对应的概率。

如前所述，让我们从probas的50257个token中，找出最大的值的id，并反查词表，返回对应的token，如下：

代码块

```
output_token_ids = torch.argmax(probas, dim=-1) # Replace probas with logits yield same result
print("shape of output_token_ids: ",output_token_ids.shape)
print("output_token_ids: \n",output_token_ids)

for i in range(output_token_ids.size(0)):
    text = tensor_to_text(output_token_ids[i].unsqueeze(0), tokenizer)
```

```
    print(f"output {i}: {text}")
```

其中argmax是返回值最大对应的下标，所以会把维度直接从50257降低到1。结果如下：

> shape of output_token_ids: torch.Size([2, 4])
>
> output_token_ids:
>
> tensor([[38491, 2448, 36069, 24862],
>
> [36397, 15489, 10460, 18747]])
>
> output 0: constants Per Rebels myriad
>
> output 1: Gathering bay 800array

回想上文，我们的目标是：

> target 0: War I or the
>
> target 1: I or the First

但看起来outputs与targets相去甚远；而我们训练的目标，就是要让output与target尽可能地接近。换言之，我们希望输出的probas矩阵中，target token的概率尽可能大，理想应该是100%。

那让我们先看下，target在输出的probas矩阵中对应的概率：

代码块

```
batch_size, seq_len = targets.shape
target_probas = torch.empty(batch_size, seq_len)

for text_idx in range(batch_size):
    positions = torch.arange(seq_len)
    print("targets: ", targets[text_idx])
    #same as probas[0,[0,1,2,3],[1810,  314,  393,  262]], advanced indexing
    target_probas[text_idx] = probas[text_idx, positions, targets[text_idx]]
    print(f"Text {text_idx + 1} target_probas:", target_probas[text_idx])
```

> targets: tensor([1810, 314, 393, 262])
>
> Text 1 target_probas: tensor([9.9453e-06, 1.9495e-05, 1.4662e-05, 1.8303e-05])
>
> targets: tensor([314, 393, 262, 3274])
>
> Text 2 target_probas: tensor([1.6926e-05, 2.1331e-05, 1.0184e-05, 1.8939e-05])

可见，target的概率在e-5以下，实在是太低了；这也是为什么模型在胡说八道，因为target的概率太低了，output距离target实在太远了，我们的目标是output距离target尽量近。

Cross-Entropy Loss

那到底如何衡量output与target的距离呢？

在上面的例子中，text 1的target词在output probas矩阵中的分布是：

> y_predict:
>
> [9.9453e-06, P1,.....P20256]
>
> [1.9495e-05, P1,.....P20256]
>
> [1.4662e-05, P1,.....P20256]
>
> [1.8303e-05, P1,.....P20256]

这里为了方便展示，我们把target_token的位置移到矩阵最前面；省略号后面还有50257-1 = 50256个其他值，但我们并不关心其他值，因为他们不是我们的目标；而这所有的值加起来的和刚好是1（这是softmax函数保证的）。

而我们的目标是什么呢？ 目标是target词的概率应该是1，其他全是0，即：

> y_true:
>
> [1, 0, 0, ... 0]
>
> [1, 0, 0, ... 0]
>
> [1, 0, 0, ... 0]
>
> [1, 0, 0, ... 0]

所以，问题的关键变成，如何衡量预测分布与真实分布之间的差异。

现在我们引入交叉熵的定义：

> For a single classification sample, assume:
>
> - True label (one-hot):
>
> $\mathbf{y} = (y_1, y_2, \ldots, y_C), \quad y_i \in \{0, 1\}$
>
> - Predicted probabilities:
>
> $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_C), \quad \sum_{i=1}^{C} \hat{y}_i = 1$
>
> **Cross Entropy Loss (General Form)**
>
> $$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$
>
> If the true class is K, the formula simplifies to:
>
> $$\mathcal{L} = -\log(\hat{y}_k)$$

看起来略显复杂，其实非常简单。(上述log是以e为底的对数，等价于ln。)

简单解释下：

1）true label是one-hot，真实值概率应该是100%，其他全为0；可见上例y_true;

2）predict probabilities，所有和累加是1；可见上例y_predict;

3）交叉熵的定义是 -y_true*log(y_predict)，然后累加。

依然拿上面例子：

> y_predict: [9.9453e-06, P1,.....P20256]
>
> y_true:[1, 0, 0, ... 0]

手算如下：

-1*log(9.9453e-06)+0*log(P1)+.....+0*log(P20256) = -log(9.9453e-06)

4）也就是可以简化为Loss = -log(y_k) 其中k是真实值的编号。

因此，交叉熵的定义非常之简洁，计算过程是非常之简单。

更严格的定义，以及其背后的数学与信息论原理，这里不再赘述。能用这么简洁的公式发明并定义出来信息的不确定性，是非常了不起的。但对于我们使用者来说，其实是特别简单的。

而这里的交叉熵，其实就是模型训练过程中的损失，我们的目标是不断使得loss尽可能低。完全理想情况下，loss= -log(1) = 0。

Loss Over Batchs

上面我们只计算了单个token的交叉熵，但在训练过程中，其实我们关心的不是单点token的交叉熵，而是一个批次、所有sample、所有预测token的平均值。

定义如下：

$$\mathcal{L}_{\text{batch}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{C} y_i^{(n)} \log\left(\hat{y}_i^{(n)}\right)$$

其实就是不断在sample和batch的维度上取平均值。

我们手动计算下上述targets的交叉熵：

代码块

```
neg_log_probas = torch.log(target_probas) * -1
print("loss matrix: ",neg_log_probas)
loss = torch.mean(neg_log_probas)
print("loss: ",loss)
```

> loss matrix: tensor([[11.5184, 10.8454, 11.1303, 10.9084],
>
> [10.9867, 10.7553, 11.4947, 10.8743]])
>
> loss: tensor(11.0642)

需要注意的是，最终计算出的loss是一个标量，因为我们在不断求平均，最终只得出一个loss。

而pytorch框架已经集成了Cross entropy loss的计算，只需要调用下即可，如下：

代码块

```
print("shape of inputs: ",logits.shape) #(batch_size, seq_len, vocab_size)
print("shape of targets: ",targets.shape)
print("targets: \n",targets) #(batch_size, seq_len)
# inputs must be raw logits (unnormalized scores), NOT probabilities
# inputs shape: (batch_size * seq_len, vocab_size)
# targets shape: (batch_size * seq_len,), containing class indices
loss = torch.nn.functional.cross_entropy(logits.view(-1,logits.size(-1)),
targets.view(-1))
print("loss: ",loss)
```

```
shape of inputs:  torch.Size([2, 4, 50257])
shape of targets:  torch.Size([2, 4])
targets:
 tensor([[1810,  314,  393,  262],
        [ 314,  393,  262, 3274]])
loss:  tensor(11.0642)
```

可见跟我们手算出的，结果是一样的。

Perplexity

在模型训练中，经常用的另一个指标是perplexity困惑度，而其定义非常简单：

$$\text{Perplexity} = e^{\mathcal{L}}$$

即困惑度是交叉熵损失的指数。

只不过是换了一种表达形式，从直觉上理解，困惑度也应该越小越好：

1）理想极端情况下，loss=0, perplexity=1，表示仅有1种选择，模型接近完美。

2）极端最差情况下，loss=inf，perplexity=inf，表示候选有无数种选择，相当于模型随机从词表中挑选，模型接近随机输出。当然，通常情况下，perplexity应该不超过词表大小。

我们计算下上例的perplexity如下：

代码块

```
perplexity = torch.exp(loss)
#Note perplexity is larger than vocab_size, which is expected, since the model
is not trained yet.
```

```
print("perplexity: ",perplexity)
```

> perplexity: tensor(63842.8828)

可见，困惑度其实已经超过了词表大小（因为尚未训练，这是可能出现的），也显示我们的模型确实在随机胡言乱语。

Loss on data sets

而在训练过程中，我们更关心的是大模型在整个训练集和验证集上的损失。

我们处理这篇短文，分割出训练集和验证集，创建出各自的data loader，并计算batch和data loader维度的loss。

首先，简单读取并处理文本，原文有很多空行，如果不去除，大模型就会学习到很多不必要的空格空行；处理如下：

代码块

```
# If you didn't clean the empty lines, LLM may learn to add too many blanks.
def clean_text_remove_empty_lines(text: str) -> str:
    lines = text.splitlines()
    non_empty_lines = [line.strip() for line in lines if line.strip() != ""]
    return "\n".join(non_empty_lines)

with open("world_war_ii.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
cleaned_text = clean_text_remove_empty_lines(raw_text)

print(cleaned_text[:200])
tokens = tokenizer.encode(cleaned_text)
print("Characters: ",len(cleaned_text))
print("Tokens: ",len(tokens))
```

> World War I or the First World War (28 July 1914 – 11 November 1918), also known as the Great War, was a global conflict between two coalitions: the Allies (or Entente) and the Central Powers. Fightin
>
> Characters: 88775
>
> Tokens: 18134

我们把80%的文本作为训练集，其余是测试集，分别创建出data loader：

代码块

```
from gpt2_v1 import GPT_CONFIG_124M
```

```
2
3  # Split text data into training and validation sets
4  train_ratio = 0.8
5  split_idx = int(len(cleaned_text) * train_ratio)
6  train_data, val_data = cleaned_text[:split_idx], cleaned_text[split_idx:]
7  print("Train data: ", len(train_data))
8  print("Val data: ", len(val_data))
9
10 torch.manual_seed(123)
11 train_loader = dataloader_v1(
12     train_data, batch_size=2,
13     context_size=GPT_CONFIG_124M["context_length"],
14     stride=GPT_CONFIG_124M["context_length"],
15     drop_last=True, shuffle=True)
16 val_loader = dataloader_v1(
17     val_data, batch_size=2,
18     context_size=GPT_CONFIG_124M["context_length"],
19     stride=GPT_CONFIG_124M["context_length"],
20     drop_last=False, shuffle=False)
```

> Train data: 71020
>
> Val data: 17755

简单查看并验证下数据维度：

代码块

```
1  print("Train dataloader: ", len(train_loader))
2  train_first_batch = next(iter(train_loader))
3  print(train_first_batch[0].shape, train_first_batch[1].shape)
4  print("Val dataloader: ", len(val_loader))
5  val_first_batch = next(iter(val_loader))
6  print(val_first_batch[0].shape, val_first_batch[1].shape)
```

> Train dataloader: 7
>
> torch.Size([2, 1024]) torch.Size([2, 1024])
>
> Val dataloader: 2
>
> torch.Size([2, 1024]) torch.Size([2, 1024])

可见，全文总共大概18000个token，取前18000*80%=14400作为训练集；而每个batch有2个样本；最大窗口是1024；所以训练集dataloader有14400/2/1024=7个。相应的，验证集仅有2个；所以是极小的数据集，仅供演示目的。

然后，我们分别计算每个batch和整个loader级别的损失，如下：

代码块

```
def loss_batch(inputs, targets, model, device):
    inputs, targets = inputs.to(device), targets.to(device)
    logits = model(inputs)
    loss = torch.nn.functional.cross_entropy(logits.flatten(0, 1), targets.flatten(0))
    return loss


def loss_loader(loader, model, device, num_batches=None):
    if len(loader) == 0:
        return float('nan')

    total_loss = 0.0
    # num_batches no more than len(loader), default to len(loader)
    num_batches = min(num_batches or len(loader), len(loader))

    for i, (inputs, targets) in enumerate(loader):
        if i >= num_batches:
            break
        loss = loss_batch(inputs, targets, model, device)
        total_loss += loss.item()

    return total_loss / num_batches
```

其实上述代码，就是在不断的求loss，然后取平均。

现在，我们可以查看下当前的模型状态，对于测试集和验证集的初始loss，如下：

代码块

```
# MPS may have some issues when training
device = (
    torch.device("cuda") if torch.cuda.is_available()
    else torch.device("mps") if torch.backends.mps.is_available()
    else torch.device("cpu")
)

model.to(device)
torch.manual_seed(123)
with torch.no_grad():
    train_loss = loss_loader(train_loader, model, device)
    val_loss = loss_loader(val_loader, model, device)
print("Train loss: ", train_loss)
print("Val loss: ", val_loss)
```

> Train loss: 11.00249263218471

> Val loss: 10.98751163482666

可见，loss非常高；这符合预期，毕竟我们的模型尚未进行任何训练。

注：device选择上优先GPU cuda，其次是MacBook的mps，如果都没有选择cpu；在演示环节差异不大；有时候mps在训练过程pytorch支持可能有问题，可切回cpu。最关键的是训练过程中要确保model、inputs、target，以及创建出的张量都在同一个device上。否则会有类似报错：all input tensors must be on the same device.

Train

现在让我们用上面的短文训练模型。我们希望进行10个epoch，而每个epoch是指的完整地处理完训练集中的所有样本。

训练过程中，最核心的代码如下：

> `optimizer.zero_grad()`
>
> `loss = loss_batch(input_batch, target_batch, model, device)`
>
> `loss.backward()`
>
> `optimizer.step()`

其中：

1）`optimizer.zero_grad()` 用于在每个批次之初清零梯度。这是因为pytorch的梯度自动累积，而每个批次的训练需要独立的梯度。

2）`loss = loss_batch(input_batch, target_batch, model, device)` 用于计算当前批次的损失值。实际上是前向传播过程，得到模型输出，并计算损失。

3）`loss.backward()` 这行代码的作用是执行反向传播算法。它会根据损失函数，利用链式法则计算每个可训练参数的梯度（也就是偏导数）。计算得到的梯度会被保存在各个参数的 `.grad` 属性中。

4）`optimizer.step()` 这行代码的作用是根据计算得到的梯度，按照特定的优化策略（如 SGD、Adam 等）来更新模型的参数。

总结下就是：梯度清零 -> 前向传播 -> 反向传播 -> 参数更新。

完整的代码如下：

代码块

```
1
2  def train_model_simple(model, train_loader, val_loader, optimizer, device,
   num_epochs,
3                         eval_freq, eval_iter, start_context, tokenizer):
4      train_losses, val_losses, tokens_seen_track = [], [], []
5      tokens_seen, step = 0, 0
6
```

```
    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = loss_batch(input_batch, target_batch, model, device)
            loss.backward()
            optimizer.step()

            tokens_seen += input_batch.numel()
            step += 1

            if step % eval_freq == 0:
                train_loss = loss_loader(train_loader, model, device,
eval_iter)
                val_loss = loss_loader(val_loader, model, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                tokens_seen_track.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {step:06d}): Train loss
{train_loss:.3f}, Val loss {val_loss:.3f}")

        generate_and_print_sample(model, tokenizer, start_context, device)

    return train_losses, val_losses, tokens_seen_track


def generate_and_print_sample(model, tokenizer, start_context, device):
    model.eval()
    with torch.no_grad():
        result = complete_text(start_context, model,20,GPT_CONFIG_124M,device)
        print(result)
    model.train()
```

在上述代码中，我们在每个epoch结束，增加了generate_and_print_sample用于直观地评测模型输出效果；而每个batch相当于一个step；每eval_freq步，我们print输出在训练集和验证集上的损失，以及当前已处理的token总数。

因为我们的短文实在太小，为了演示方便，我们把context_length从1024调小到128；总共训练10个epoch，代码如下：

代码块

```
import copy
from gpt2_v1 import GPT2Model
import time

```

```python
config = copy.deepcopy(GPT_CONFIG_124M)
config["context_length"] = 128

# Set seed for reproducibility
torch.manual_seed(123)
# Initialize model and optimizer
model = GPT2Model(config).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=4e-4, weight_decay=0.1)

# Start timer
start_time = time.time()

# Train the model
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model=model,
    train_loader=train_loader,
    val_loader=val_loader,
    optimizer=optimizer,
    device=device,
    num_epochs=num_epochs,
    eval_freq=10,
    eval_iter=5,
    start_context="at the start of",
    tokenizer=tokenizer
)

# Report execution time
elapsed = (time.time() - start_time) / 60
print(f"Training completed in {elapsed:.2f} minutes.")
```

> Ep 1 (Step 000010): Train loss 8.104, Val loss 8.263
>
> Ep 1 (Step 000020): Train loss 7.535, Val loss 7.935
>
> Ep 1 (Step 000030): Train loss 7.153, Val loss 7.773
>
> Ep 1 (Step 000040): Train loss 6.801, Val loss 7.731
>
> Ep 1 (Step 000050): Train loss 6.626, Val loss 7.619
>
> at the start of the war.
>
> of the war.
>
> and the war, the war, the war, and
>
> Ep 2 (Step 000060): Train loss 6.402, Val loss 7.679
>
> Ep 2 (Step 000070): Train loss 6.217, Val loss 7.721

Ep 2 (Step 000080): Train loss 6.105, Val loss 7.633

Ep 2 (Step 000090): Train loss 6.103, Val loss 7.612

Ep 2 (Step 000100): Train loss 5.734, Val loss 7.622

Ep 2 (Step 000110): Train loss 5.926, Val loss 7.556

at the start of the war.

===

===

===

===

===

===

==== war.

Ep 3 (Step 000120): Train loss 5.407, Val loss 7.646

Ep 3 (Step 000130): Train loss 5.406, Val loss 7.719

Ep 3 (Step 000140): Train loss 5.118, Val loss 7.606

Ep 3 (Step 000150): Train loss 4.908, Val loss 7.610

Ep 3 (Step 000160): Train loss 4.632, Val loss 7.425

at the start of the war.

==== German Army to the war and the war.

==== German troops to the

Ep 4 (Step 000170): Train loss 4.449, Val loss 7.475

Ep 4 (Step 000180): Train loss 3.780, Val loss 7.484

Ep 4 (Step 000190): Train loss 3.948, Val loss 7.506

Ep 4 (Step 000200): Train loss 3.762, Val loss 7.614

Ep 4 (Step 000210): Train loss 3.497, Val loss 7.546

Ep 4 (Step 000220): Train loss 3.456, Val loss 7.564

at the start of the war. The Allies, the war to the war, the German government the war,000,

Ep 5 (Step 000230): Train loss 3.198, Val loss 7.512

Ep 5 (Step 000240): Train loss 3.033, Val loss 7.567

Ep 5 (Step 000250): Train loss 2.296, Val loss 7.575

Ep 5 (Step 000260): Train loss 3.106, Val loss 7.684

Ep 5 (Step 000270): Train loss 2.759, Val loss 7.690

Ep 5 (Step 000280): Train loss 2.314, Val loss 7.581

at the start of the British Army had to Germany.

In the first medical for the end of the war ===

Ep 6 (Step 000290): Train loss 2.134, Val loss 7.646

Ep 6 (Step 000300): Train loss 1.844, Val loss 7.784

Ep 6 (Step 000310): Train loss 1.830, Val loss 7.767

Ep 6 (Step 000320): Train loss 1.437, Val loss 7.774

Ep 6 (Step 000330): Train loss 1.751, Val loss 7.765

at the start of the British and negotiations with the Battle of the British had been called the Battle of the French to,

Ep 7 (Step 000340): Train loss 1.501, Val loss 7.873

Ep 7 (Step 000350): Train loss 1.175, Val loss 7.815

Ep 7 (Step 000360): Train loss 1.029, Val loss 7.923

Ep 7 (Step 000370): Train loss 1.023, Val loss 7.982

Ep 7 (Step 000380): Train loss 1.098, Val loss 8.034

Ep 7 (Step 000390): Train loss 0.628, Val loss 8.024

at the start of the war on the first the German Supreme Army to the French, and the German terms.

====

Ep 8 (Step 000400): Train loss 0.774, Val loss 8.047

Ep 8 (Step 000410): Train loss 0.703, Val loss 8.081

Ep 8 (Step 000420): Train loss 0.442, Val loss 8.087

Ep 8 (Step 000430): Train loss 0.667, Val loss 8.222

Ep 8 (Step 000440): Train loss 0.358, Val loss 8.070

at the start of war ceased under the provisions of the Termination of the Present War (Definition) Act 1918 concerning:

Ep 9 (Step 000450): Train loss 0.439, Val loss 8.133

Ep 9 (Step 000460): Train loss 0.363, Val loss 8.154

> Ep 9 (Step 000470): Train loss 0.271, Val loss 8.249
>
> Ep 9 (Step 000480): Train loss 0.295, Val loss 8.205
>
> Ep 9 (Step 000490): Train loss 0.208, Val loss 8.318
>
> Ep 9 (Step 000500): Train loss 0.234, Val loss 8.252
>
> at the start of these agreements was to isolate France by ensuring the three empires resolved any disputes among themselves. In 1887
>
> Ep 10 (Step 000510): Train loss 0.193, Val loss 8.325
>
> Ep 10 (Step 000520): Train loss 0.168, Val loss 8.322
>
> Ep 10 (Step 000530): Train loss 0.166, Val loss 8.343
>
> Ep 10 (Step 000540): Train loss 0.103, Val loss 8.435
>
> Ep 10 (Step 000550): Train loss 0.169, Val loss 8.382
>
> Ep 10 (Step 000560): Train loss 0.098, Val loss 8.352
>
> at the start of French determination and self-sacrifice.
>
> The Battle of the Somme was an Anglo-French
>
> Training completed in 2.01 minutes.

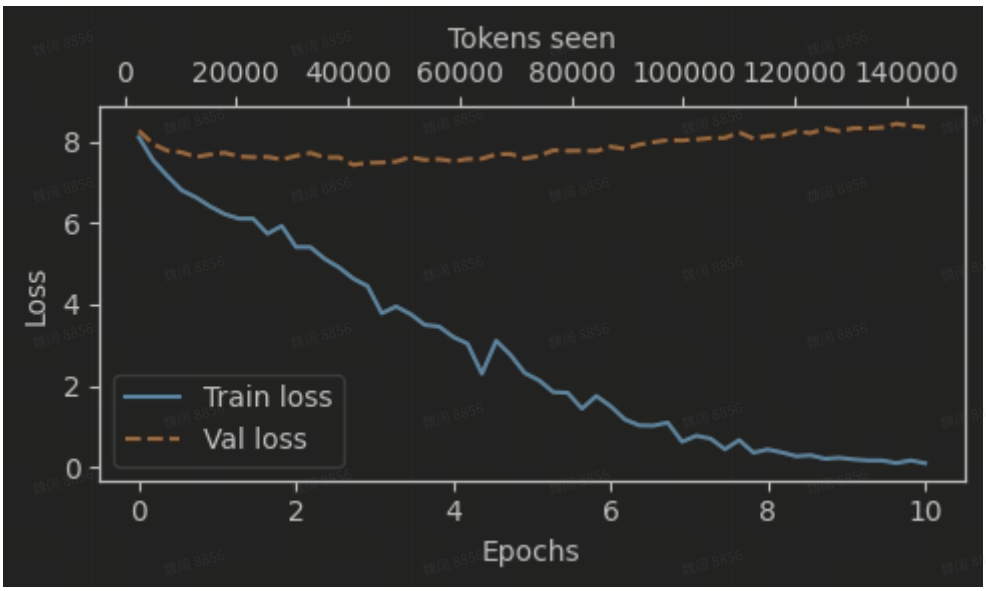
可见，训练在普通MacBook上使用mps在2分钟左右就能完成；测试的文本补全从随机乱码，变得稍微有些合理性；训练集的loss在快速下降；但是验证集的loss几乎没下降。这是明显的过拟合现象，是因为我们的模型本身很复杂，但是所使用的训练样本太过简单。后续，我们会尝试在更大数据集上训练。此处仅做示例用途。

我们也可以可视化输出一下loss和已训练token的变化过程，如下：

代码块

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

def plot_losses(epochs, tokens, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    ax1.plot(epochs, train_losses, label="Train loss")
    ax1.plot(epochs, val_losses, linestyle="--", label="Val loss")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend()
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))

    ax2 = ax1.twiny()
```

```
    ax2.plot(tokens, train_losses, alpha=0)  # Invisible plot for aligning ticks
    ax2.set_xlabel("Tokens seen")

    fig.tight_layout()
    # plt.savefig("loss-plot.pdf")
    plt.show()

# Example usage
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```



Decoding Strategies

Greedy Decoding

> ✏️ **Sample decoding:**
>
> - Greedy decoding: Select the word with the highest probability (argmax) at each step.
> - Sampling decoding: Randomly sample the next word from the probability distribution, for example using torch.multinomial.

我们可以使用训练好的模型补全文本，如下：

代码块

```
model.eval()
result = complete_text("at the start of the", model,15,device="cpu")
print("Output text:\n", result)
```

> Output text:
>
> at the start of the Treaty of Bucharest was formally annulled by the Armistice of

在指定随机数的情况下，我们多次运行相同的start context，结果总是一样的。这是因为我们总是从生成的词表概率表中选择概率最大的那个，也就是如下代码中使用了argmax:

代码块

```
import torch

input_text = "at the start of the"
input_tensor = text_to_tensor(input_text, tokenizer).to("cpu")
print("Input tensor: ", input_tensor)

logits = model(input_tensor)
print("Shape of logits: ", logits.shape)

next_token_logits = logits[:, -1, :]
print("Shape of next_token_logits: ", next_token_logits.shape)
print("next_token_logits: ", next_token_logits)

probas = torch.softmax(next_token_logits, dim=-1)
next_token_id = torch.argmax(probas, dim=-1).item()
print("Next token id: ", next_token_id)

next_token = tokenizer.decode([next_token_id])
print("Next token: ", next_token)
```

> Input tensor: tensor([[2953, 262, 923, 286, 262]], device='mps:0')
>
> Shape of logits: torch.Size([1, 5, 50257])
>
> Shape of next_token_logits: torch.Size([1, 50257])
>
> next_token_logits: tensor([[-2.1345, -0.8003, -6.3171, ..., -6.6543, -5.5982, -6.4263]],
>
> device='mps:0', grad_fn=<SliceBackward0>)
>
> Next token id: 21345
>
> Next token: Treaty

这属于Greedy Decoding，即总是选择概率最大的token。

Sample Decoding

而另一种更随机的decode方式是Sample Decoding，即按照概率分布随机选择token。

而实现的方式也非常简单，只需要把上述代码中的argmax改为multinomial，如下：

```
代码块
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print("Next token id: ", next_token_id)
next_token = tokenizer.decode([next_token_id])
print("Next token: ", next_token)
```

> Next token id: 4141
>
> Next token: French

我们可以运行100次，直观感受下，按照概率采样，next token的生成分布情况，如下：

```
代码块
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item() for _ in range(100)]
    sampled_ids = torch.bincount(torch.tensor(sample), minlength=probas.shape[-1])
    for id, freq in enumerate(sampled_ids):
        if freq > 1:
            print(f"{freq} x {tokenizer.decode([id])}")

print_sampled_tokens(probas)
```

> 3 x end
>
> 2 x war
>
> 2 x policy
>
> 2 x British
>
> 2 x meaning
>
> 22 x French
>
> 2 x direction
>
> 2 x refused
>
> 3 x Empire
>
> 28 x Treaty

可见其中“Treaty”在100次中有28次；而其他词也至少出现了2次以上。

我们也可以采用模拟的方式，去理解decoding strategy的差异。代码如下，我们模拟了补全‘At the start of the’可能的结果词，并按照正态分布生成了概率表：

```python
import torch

#Complete 'At the start of the'
possible_text = "war battle revolution novel experiment day journey movement"
words = possible_text.lower().split()
vocab = {word: idx for idx, word in enumerate(words)}
inverse_vocab = {idx: word for word, idx in vocab.items()}

# Step 2: Generate random logits for each vocab token
vocab_size = len(vocab)
torch.manual_seed(123)
next_token_logits = torch.normal(mean=0.0, std=4.0, size=(vocab_size,))  # increase std to increase randomness

# Convert logits to probabilities
probas = torch.softmax(next_token_logits, dim=0)

# Pick next token by argmax
next_token_id = torch.argmax(probas).item()

# Decode and print the predicted token
print(f"Next generated token: {inverse_vocab[next_token_id]}")
```

> Next generated token: day

如果采用argmax，下一次总是会得到day，因为day的概率最大。

而如果采用multinomial，并运行100次，得到词的分布如下:

代码块

```python
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item() for _ in range(100)]
    sampled_ids = torch.bincount(torch.tensor(sample), minlength=probas.shape[-1])
    for id, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[id]}")

print_sampled_tokens(probas)
```

> 11 x war
>
> 31 x battle
>
> 7 x revolution

> 4 x novel
>
> 0 x experiment
>
> 46 x day
>
> 1 x journey
>
> 0 x movement

可见采用multinomial的sample decoding方法带来了更多随机性。

Top-k Sampling

但multinomial增加随机性带来的后果是，概率很低的词也会出现；而有的时候，我们只想让概率较高的词出现，而想排除掉概率很低的词。

Top-k就是只在概率最高的前K个词中进行采样。

如在上述示例中，我们仅选取top 3的token，如下：

代码块

```
print(next_token_logits)
top_k = 3
top_k_logits, top_k_indices = torch.topk(next_token_logits, k=top_k, dim=-1)
print("top_k_logits: ", top_k_logits)
print("top_k_indices: ", top_k_indices)
```

> tensor([-0.4459, 0.4815, -1.4785, -0.9617, -4.7877, 0.8371, -3.8894, -3.0202])
>
> top_k_logits: tensor([0.8371, 0.4815, -0.4459])
>
> top_k_indices: tensor([5, 1, 0])

我们得到了top 3的原始logits值，最低值是-0.4459；接下来，我们只需要把低于最低值的其他logits遮蔽掉即可，而遮蔽方法也非常简单，只需要填充-inf，后续softmax之后会变为0，如下：

代码块

```
# Mask out logits that are not in the top-k by setting them to -inf
threshold = top_k_logits[-1]
new_logits = torch.where(
    next_token_logits < threshold,
    torch.full_like(next_token_logits, float('-inf')),
    next_token_logits
)

print("new_logits: ", new_logits)
topk_probas = torch.softmax(new_logits, dim=-1)
```

```
print("topk_probas: ", topk_probas)

print_sampled_tokens(topk_probas)
```

```
new_logits: tensor([-0.4459, 0.4815,  -inf,  -inf,  -inf, 0.8371,  -inf,  -inf])
topk_probas: tensor([0.1402, 0.3543, 0.0000, 0.0000, 0.0000, 0.5056, 0.0000, 0.0000])
13 x war
34 x battle
0 x revolution
0 x novel
0 x experiment
53 x day
0 x journey
0 x movement
```

可见，在new_logits中我们只保留了top 3的值；而在生成的概率表中，也仅有top 3；而生成的token也仅有3种。

Temperature

而温度是另一种更常见的控制随机性的方法，而其实现也特别简单，就是对模型输出的原始logits进行scale：scaled_logits = logits / temperature。

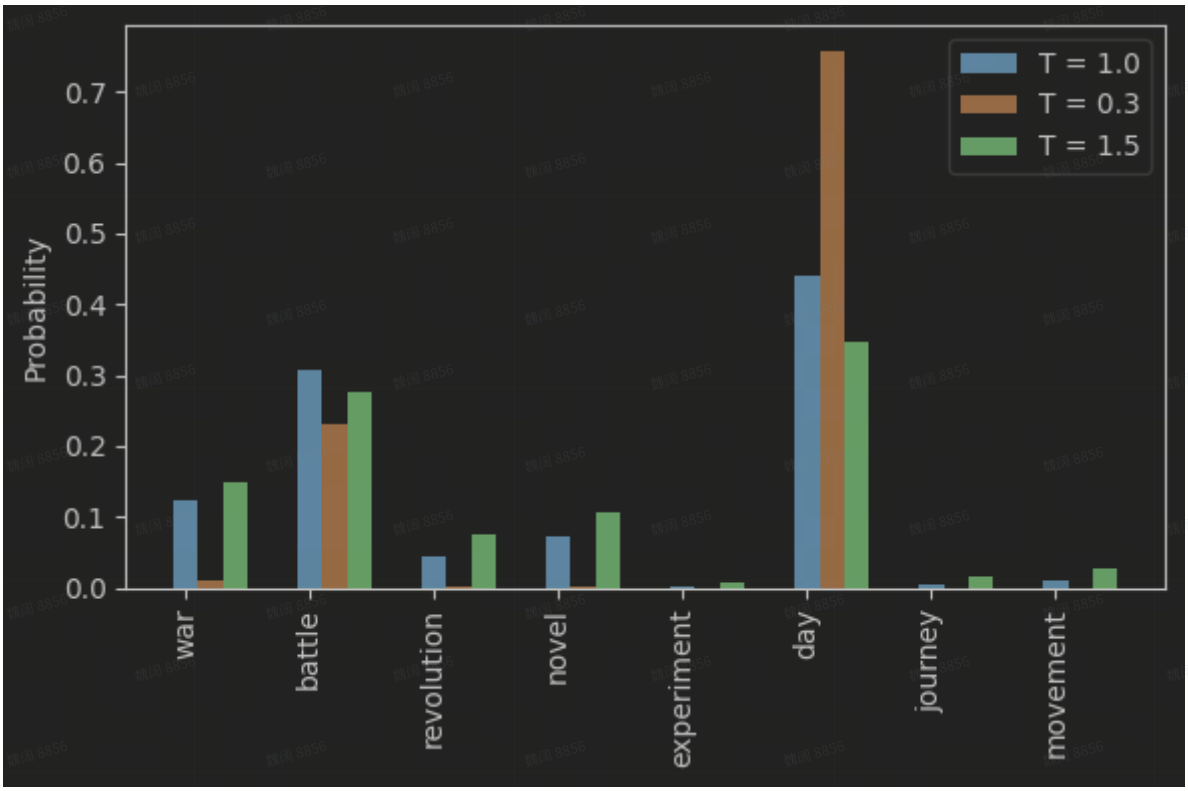
示例代码如下：

代码块

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=-1)

# Temperature values
temperatures = [1.0, 0.3, 1.5]

# Calculate scaled probabilities
scaled_probas = [softmax_with_temperature(next_token_logits, T) for T in
temperatures]
```

我们可以通过画图，直观看下不同温度下的生成token分布：

```python
import torch
import matplotlib.pyplot as plt

# Plotting
x = torch.arange(len(vocab))
bar_width = 0.2

fig, ax = plt.subplots(figsize=(6, 4))

for i, T in enumerate(temperatures):
    ax.bar(x + i * bar_width, scaled_probas[i], width=bar_width, label=f"T = {T}")

ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()

plt.tight_layout()
plt.show()
```



可见，temperature越高，分布越平坦，随机性越大；而temperature越低，分布越尖锐，倾向于集中在概率较大的词，确定性越强，随机性越低。

Generate text with temperature and top_k

我们可以将上述top_k和temperate结合起来，优化生成文本的代码，如下：

代码块

```
def generate_text_simple(model, idx, max_new_tokens, context_size, temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]

        # Get logits from model
        with torch.no_grad():
            logits = model(idx_cond)

        # Take logits for the last time step
        # (batch, n_tokens, vocab_size) -> (batch, vocab_size)
        logits = logits[:, -1, :]

        if top_k is not None:
            top_logits, _ = torch.topk(logits, top_k, dim=-1)  # (batch, top_k)
            threshold = top_logits[:, -1].unsqueeze(-1) # (batch, ) -> (batch, 1)
            logits = torch.where(
                logits < threshold,
                torch.full_like(logits, float('-inf')),
                logits
            )
        if temperature > 0.0:
            logits = logits / temperature

            # Apply softmax to get probabilities
            probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

            # Sample from distribution
            idx_next = torch.multinomial(probas, num_samples=1)  # (batch, 1)
        else:
            # Greedy sampling
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)  # (batch, 1)

        if eos_id is not None and idx_next == eos_id:
            break

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

再次尝试生成文本：

```
torch.manual_seed(123)

token_ids = generate_text_simple(
    model=model.to("cpu"),
    idx=text_to_tensor("at the start of the", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=3,
    temperature=1.4
)

print("Output text:\n", tensor_to_text(token_ids, tokenizer))
```

> Output text:
>
> at the start of the French troops had been unjust. On 3 November 1918 and the German leaders,

在上例中，我们在generate_text_simple中增加了top_k和temperature参数，以更好地控制模型输出的随机性。

简要总结下控制随机性的方法：

1）argmax与multinomial本质上是不改变输入的概率表，只改变采样的方式；argmax是确定性最大值选取；multinational是按概率随机性采样。

2）top-k是过滤掉低概率的值，本质上是对概率表做截断操作。

3）temperate是缩放原始logits，本质上改变了概率表的分布。

Train on larger datasets

我们也可以尝试在更大数据集上训练模型，如可以使用HuggingFace的wikitext，如下：

代码块

```
from datasets import load_dataset
dataset = load_dataset("wikitext", "wikitext-2-raw-v1")
```

> DatasetDict({
>
> test: Dataset({
>
> features: ['text'],
>
> num_rows: 4358
>
> })

```
    train: Dataset({
        features: ['text'],
        num_rows: 36718
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 3760
    })
})
```

具体训练方法类似，有兴趣可以自行尝试。