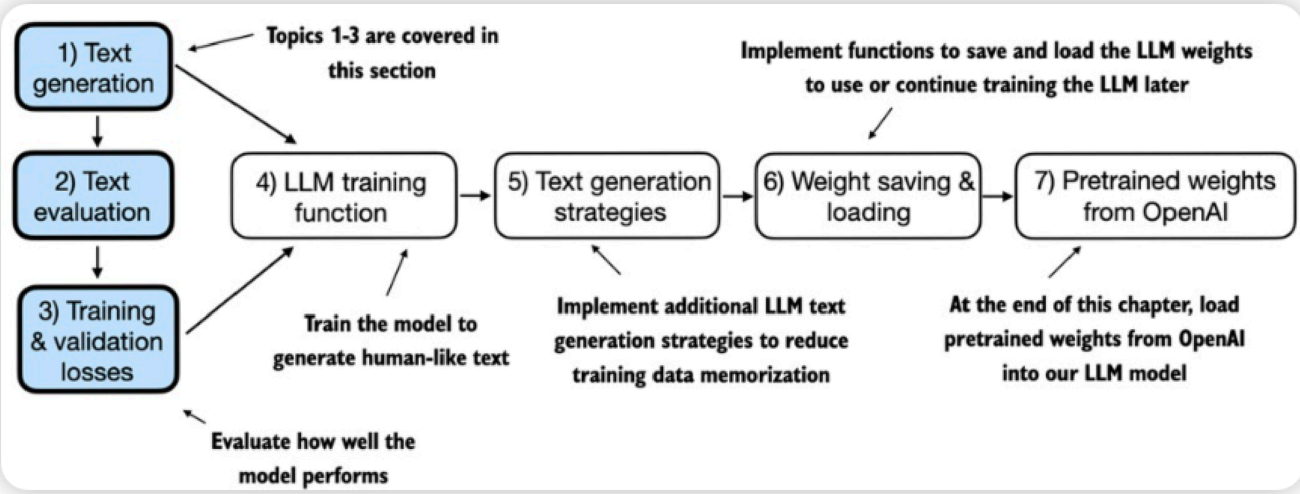
预训练的目标：在于从大量未标注的数据中提取出有用的特征，并形成一个具备较强泛化能力的基础模型。



承接上一章的文本生成内容，然后实现在预训练阶段进行模型评估。

5 - 1 文本生成

代码 - generate_text_simple

```
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # 当前上下文的索引数组 idx，要生成的新token的最大数量 max_new_tokens
    for _ in range(max_new_tokens):
        # 使用最后上下文大小的索引数组
        idx_cond = idx[:, -context_size:]

        # 调用模型
        with torch.no_grad():
            logits = model(idx_cond)
        # 将 (batch, n_token, vocab_size) 转换为 (batch, vocab_size)
        logits = logits[:, -1, :]
        # 获取具有最高logits值的词汇表条目的索引
        idx_next = torch.argmax(logits, dim=-1, keepdim=True) # 形状为 (batch, 1)
        # 将采样的索引追加到运行序列
        idx = torch.cat((idx, idx_next), dim=1) # 形状为 (batch,
```

```
n_tokens+1)
    return idx
```

5 - 2 - 文本评估

交叉熵损失函数

作用

衡量预测概率分布与真实分布的差异，分类问题中常用的损失函数。

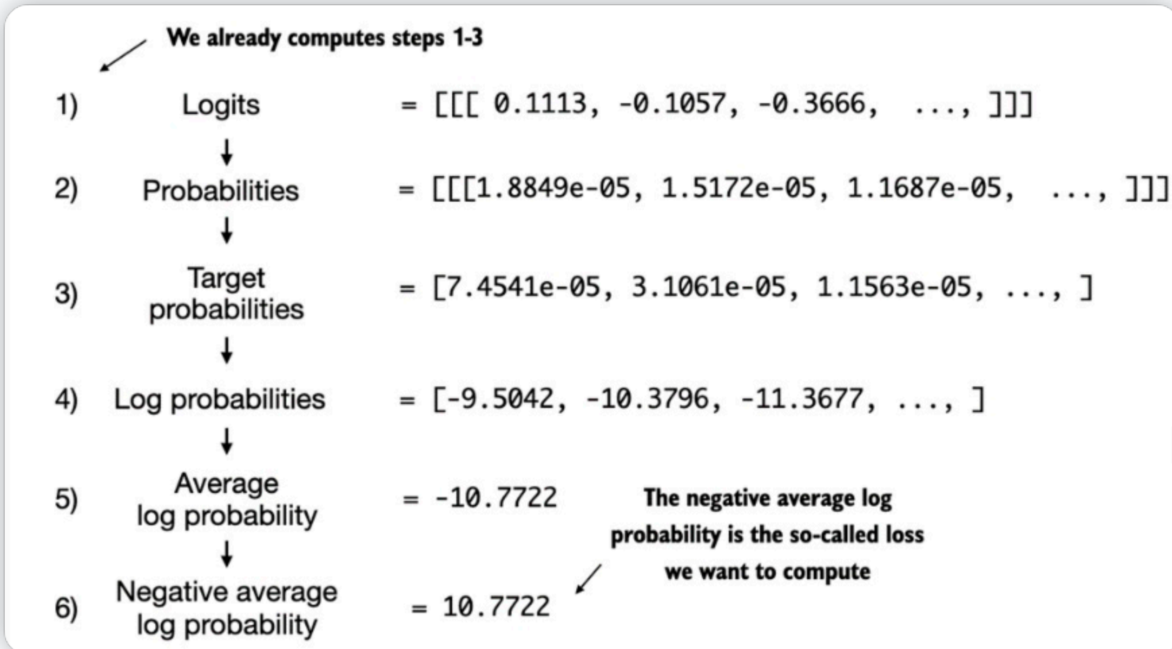
- 公式：

$$H(p,q) = -\sum_{i=1}^{n} p(x_i)log(q(x_i))$$

计算步骤

exp - 以 inputs: `["every effort moves"], ["I really like"]` 共 `2` 个 batch，每个批次 `3` 个 token 为例。（词汇表共 `50257` 个单词）

> 相关概念：logits - 矫正值，probabilitites - 概率，target probabilities - 目标概率，log probabilitites - 对数概率，average log probability - 平均对数概率，

> negative average log probability - 负平均对数概率，交叉熵损失
>
> 
>

1. 进行一次 traning loop， 计算每个输入token的 logits 矫正值（维度： `[2, 3, 50257]` ， 下同。即每个词汇为对应输出的矫正值）
2. 使用 Softmax函数 将 logits 转换为 probabilitites （ `[2, 3, 50257]` ）
3. 对每个输入 token 取最大 probabilitites 作为 target probabilities （ `[6]` ）
4. 对 target probabilities 取对数，得 log probabilitites （ `[6]` ）
5. log probabilitites 求和得 average log probability (`[1]`)
6. 取负得到 negative average log probability (`[1]`)

代码复现

```
# 数据
inputs = torch.tensor([[16833, 3626, 6100], # ["every effort
moves",
                       [40, 1107, 588]]) # "I really like"]
targets = torch.tensor([[3626, 6100, 345 ], # [" effort moves you",
                        [588, 428, 11311]]) # " really like
chocolate"]
```

- 手动实现

```
# 步骤1: 获得 logits
with torch.no_grad():
    logits = model(inputs)

# 步骤2:
probas = torch.softmax(logits, dim=-1) # 词表中每个 token 的概率
print(probas.shape)

# 步骤3: 对每个输入 token 取最大 probabilitites 作为 target
probabilities
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
target_probas_1 = probas[0, [0, 1, 2], targets[text_idx]]
target_probas_2 = probas[1, [0, 1, 2], targets[text_idx]]

# 步骤4: 对概率分数应用对数函数
log_probas = torch.log(torch.cat((target_probas_1,
target_probas_2)))
print(log_probas)

# 步骤5: 计算平均值将对数概率合并为一个分数
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)

# 步骤6: 取得负数
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

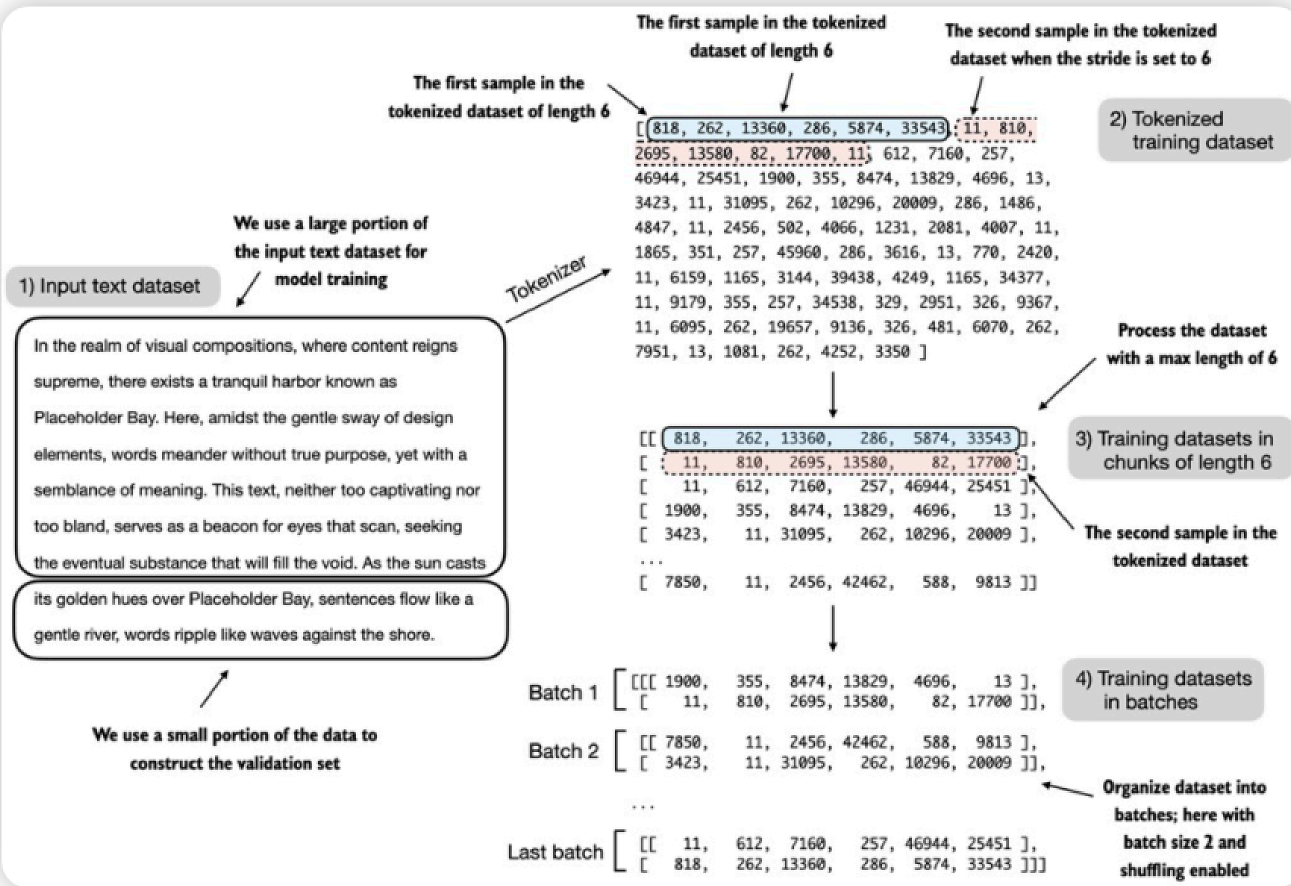
- 使用封装的cross entropy

> 等价与上面的六步

```
# 步骤1: 获得 logits
with torch.no_grad():
```

```python
    logits = model(inputs)

# 铺平
logits_flat = logits.flatten(0, 1) # [6, 50257]
targets_flat = targets.flatten() # [6]

loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

5 - 3 - 训练和验证损失



1. 先将输入文本分割为训练集和验证集；
2. 对文本进行 token 化处理，并将token化后的文本划分为用户指定长度块；
3. 打乱各行顺序，并划分batch；
4. 进行模型评估。（下面的代码实现了模型评估）

代码 - 计算损失函数

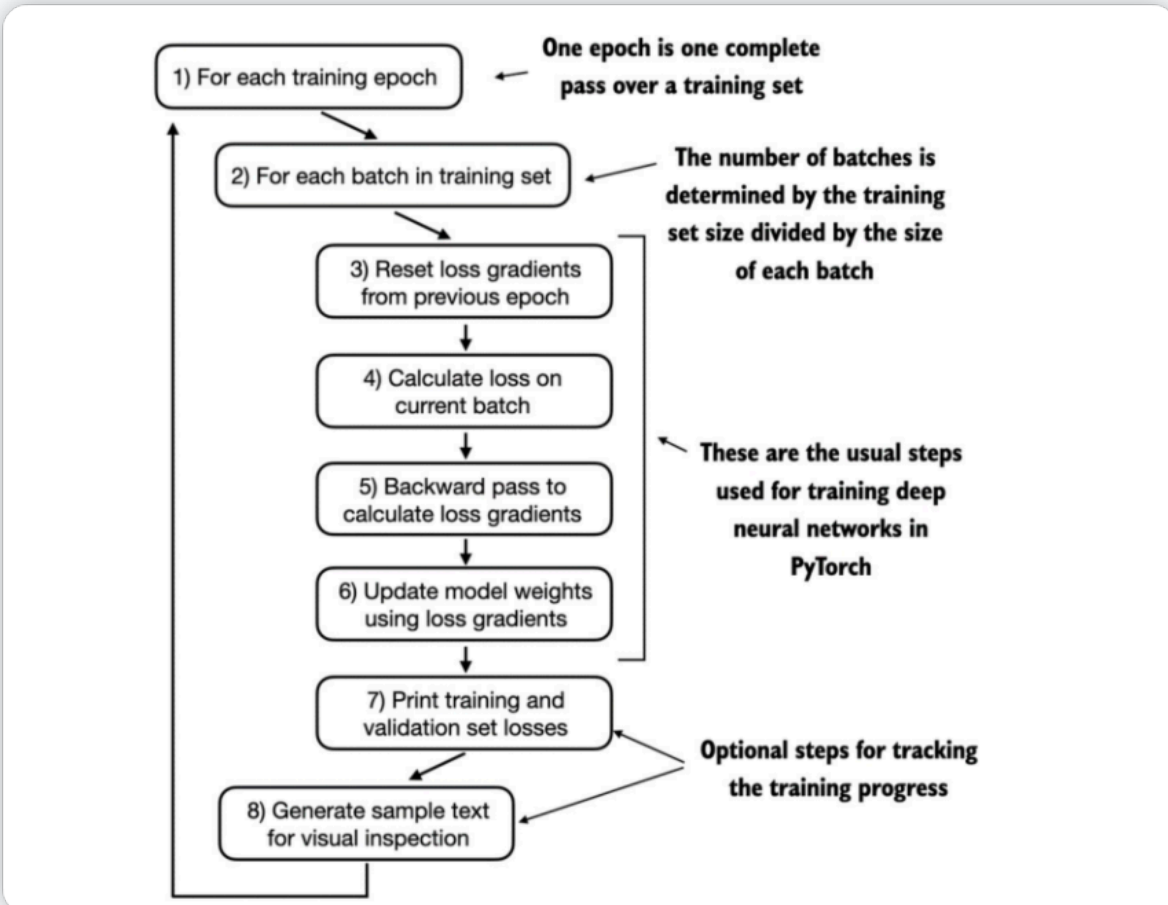
> `flatten()` 参考 [Function > ^b354a3]()

```python
import torch

# 计算单批次的损失函数
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch, target_batch = input_batch.to(device),
target_batch.to(device) #A
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1),
        target_batch.flatten()
    )
    return loss

# 计算损失函数
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch,
model, device)
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

实现了基本的模型评估技术以计算训练集和验证集的损失。接下来，我们将进入了解训练函数，并对 LLM 进行预训练。

5 - 4 - 模型训练

exp - 典型Pytorch神经网络训练工作流程

> 从迭代每个时期开始，处理批处理，重置和计算梯度，更新权重，监控步骤
>
> 
>

代码 - train_model_simple

```
def train_model_simple(model, train_loader, val_loader, optimizer,
device, num_epochs,
                       eval_freq, eval_iter, start_context):
    # eval_iter : 每隔多少个训练步骤（global_step）对模型进行一次评估
    # eval_iter : 每次评估时要运行的批次数量
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1
    for epoch in range(num_epochs):
        model.train()
```

```
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() # 在每个优化步骤前清除梯度
            loss = calc_loss_batch(input_batch, target_batch,
model, device)
            loss.backward() # 反向传播，计算梯度
            optimizer.step() # 更新模型参数
            tokens_seen += input_batch.numel() # 更新已处理的token数量
            global_step += 1
            if global_step % eval_freq == 0:
                # 评估模型在训练集和验证集上的损失
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device,
eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}):
"f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")
        generate_and_print_sample(
            model, train_loader.dataset.tokenizer, device,
start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

5 - 5 - 增强随机性的编码策略

温度缩放 Temperatrue scaling

> 将概率选择过程添加到一下token生成任务的技术。

- 定义：对 logit 矫正值 除以大于0的数。
- 作用：温度大于1，导致 token 分布更均匀；温度小于1会导致分布更靠谱。
- 对比
    - 贪婪解码（greedy decoding)：

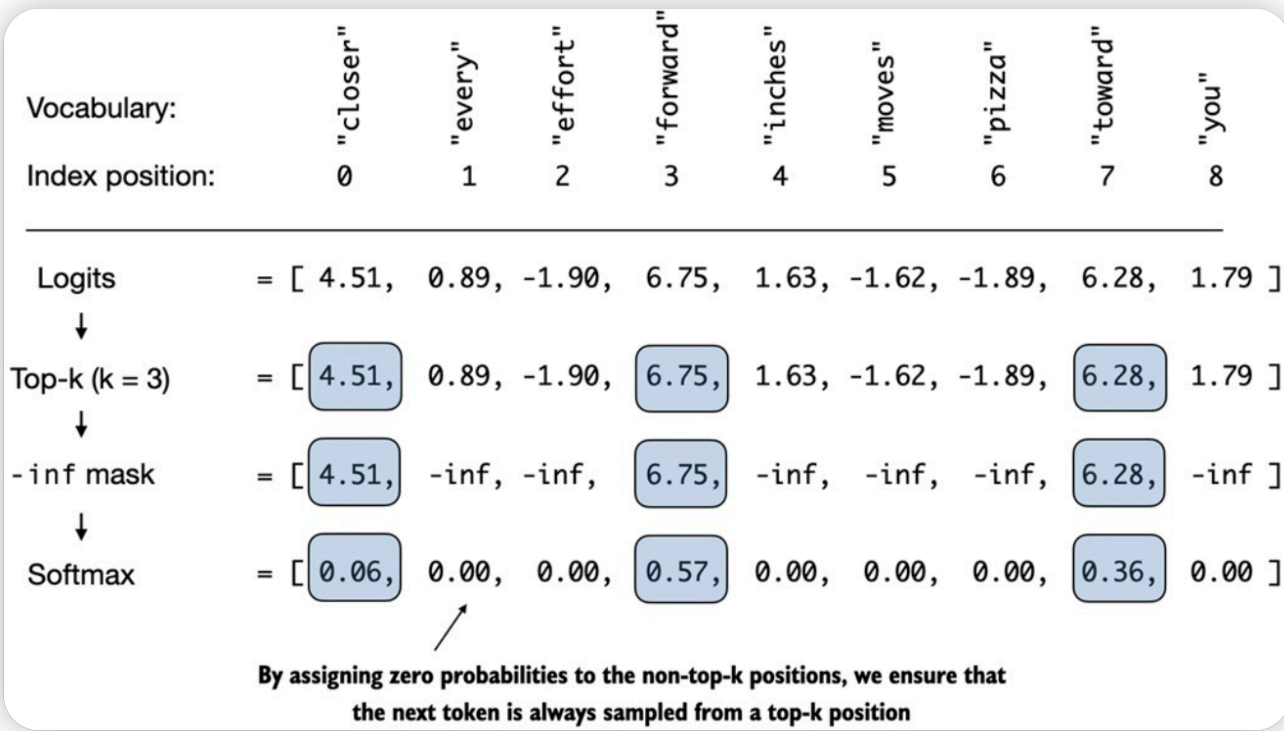
- 普通：之前使用，使用 `torch.argmax` 抽取概率最高的令牌作为下一令牌。
- 概率采样：将 `torch.argmax` 替换成从概率分布中采样的函数（比如 `torch.multinomial`）。

- 缺点：会导致语法错误或者完全无意义的输出。

Top - k 采样

与概率采样和温度放缩结合时，可以优化文本生成结果。

exp - top-3采样



> 图示步骤：
>
> 1. 选择Top - 3 个logits
> 2. 将未被选择的 logits 替换为负无穷大（-inf）

> 3. 计算softmax值时非top-k的token的概率分数为0，其余概率总和为1。

```
# 1.选择top - 3个 logits
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)

# 2.将其余logits设置为0
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float("-inf")),
    other=next_token_logits
)

# 3.使用softmax将 -inf处理为0
topk_probas = torch.softmax(new_logits, dim=0)
```

用温度缩放和Top-k来优化生成函数

```
def generate(model, idx, max_new_tokens, context_size, temperature,
top_k=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
        if top_k is not None:
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )
        if temperature > 0.0:
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
```

```
            idx_next = torch.multinomial(probs, num_samples=1)
        else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

总结