第六章　遮蔽掩码大模型

Bert语言模型的输入输出介绍

DNA modernBERT的预训练微调实现

作业布置 & 参考资料

- 作业：
  - 1. 实践作业：使用Transformers库训练DNA-ModernBERT模型，实现遮蔽掩码策略（随机遮蔽50% token，其中80%用MASK）。微调模型完成增强子分类任务，并报告测试集准确率。
  - 2. 思考题：解释为何在DNA序列处理中需要调整分词器的词汇表（如添加N碱基或特殊符号）。
- 参考资料：
  - 1. Hugging Face Transformers文档：
  - 2. 论文《BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding》（BERT原始论文）

基本原理

在掩码语言建模中，训练数据的构造包括以下步骤：

1. 随机选择句子中的一些单词，替换为特殊的 `[MASK]` 标记。
2. 将这些修改后的句子输入模型。
3. 模型基于未遮蔽的上下文预测被遮蔽的单词。

例如，假设原始句子为：

```
"The quick brown fox jumps over the lazy dog."
```

在构造训练数据时，可以随机遮蔽某些单词：

```
"The quick brown [MASK] jumps over the lazy [MASK]."
```

模型需要通过上下文信息预测 `"fox"` 和 `"dog"`。

具体实现_分词器

- 目标
  - 将每一个 单词，或者字母 编码为数字

编号	内容
0	[UNK]
1	[CLS]
2	[SEP]
3	[PAD]
4	[MASK]
5	apple
6	I
7	banana
8	monkey
9	like
10	,
11	!
12	eat
13	likes

[cls] I like eat apple, monkey likes banana! [sep]

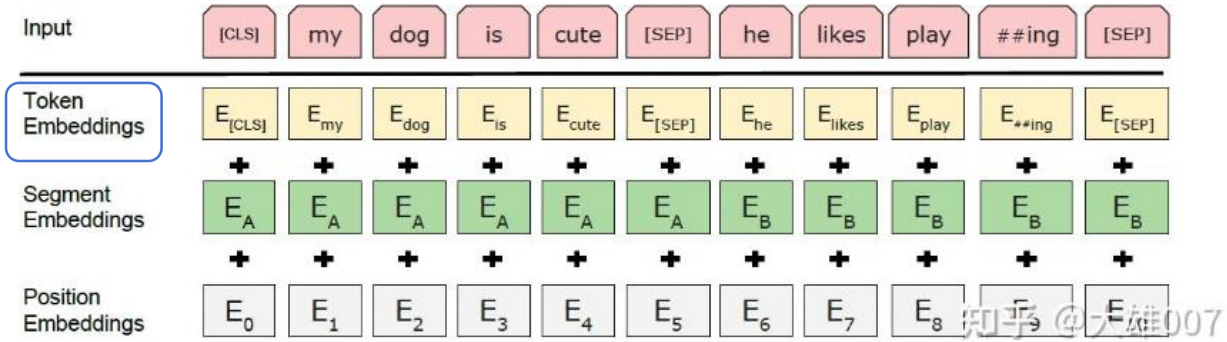
[1, 6, 9, 12, 5, 10, 8, 13, 7, 11, 2]

如果我指定长度是15，那么对于长度不足15的样本，可以在后面补上[pad]

[1, 6, 9, 12, 5, 10, 8, 13, 7, 11, 2, 3, 3, 3, 3]

每一个数字就是一个**token**

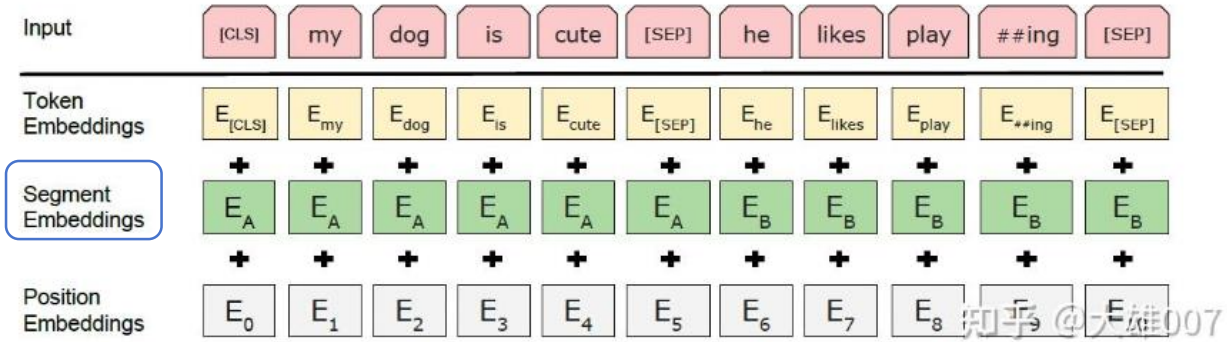
具体实现_对token进行embedding



这一个embedding矩阵是可以学习的，梯度更新的时候，对每一个token的d维编码会更新

编号	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
0	0.55	0.78	0.99	0.84	0.51	0.55	0.11	0.23
1	0.83	0.84	0.01	0.50	0.67	0.31	0.51	0.98
2	0.91	0.33	0.58	0.15	0.99	0.75	0.53	0.79
3	0.62	0.42	0.86	0.23	0.14	0.83	0.36	0.65
4	0.48	0.99	0.53	0.61	0.98	0.42	0.79	0.91
5	0.46	0.27	0.81	0.65	0.63	0.59	0.41	0.32
6	0.31	0.89	0.75	0.99	0.63	0.66	0.10	0.61
7	0.66	0.28	0.37	0.50	0.71	0.99	0.92	0.93
8	0.10	0.88	0.68	0.73	0.75	0.18	0.62	0.95
9	0.74	0.90	0.03	0.97	0.48	0.47	0.60	0.99
10	0.12	0.39	0.24	0.18	0.17	0.39	0.28	0.68
11	0.89	0.40	0.94	0.88	0.75	0.94	0.05	0.70
12	0.30	0.77	0.65	0.03	0.99	0.93	0.26	0.55
13	0.15	0.94	0.71	0.83	0.78	0.73	0.53	0.12

具体实现_对token进行embedding



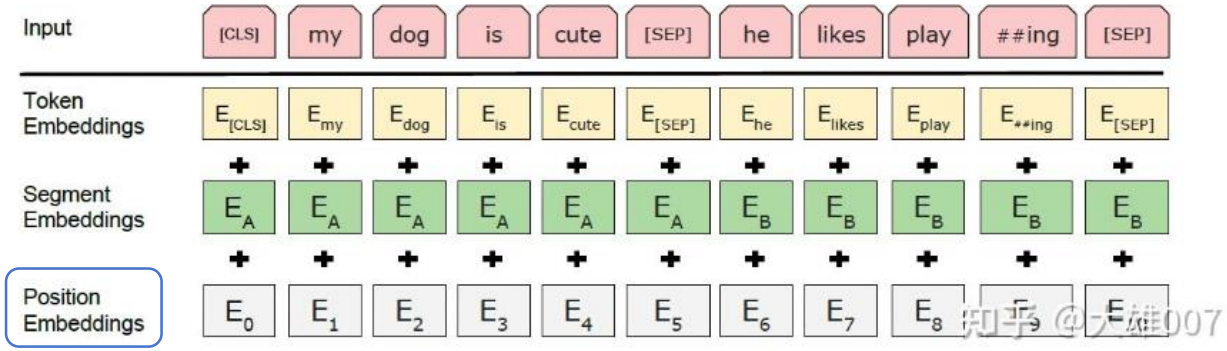
segment embedding 用于区分一个样本里面的两句话

$E_A = 0$

$E_B = 1$

DNA语言模型一般就只有一句话，所以编码为0

具体实现_对token进行embedding



位置编码
让模型能够区分每一个 token的位置

具体实现_模型的输入输出

[cls] I like eat apple, monkey likes banana! [sep]

[1, 6, 9, 12, 5, 10, 8, 13, 7, 11, 2]

如果我指定长度是15，那么对于长度不足15的样本，可以在后面补上[pad]

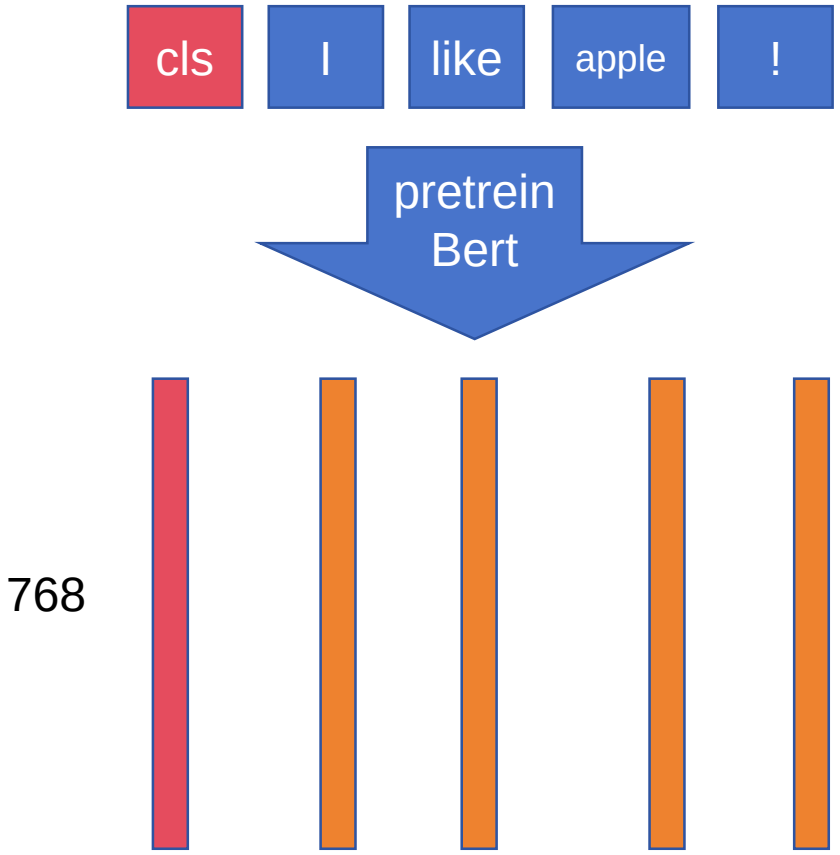
[1, 6, 9, 12, 5, 10, 8, 13, 7, 11, 2, 3, 3, 3, 3]

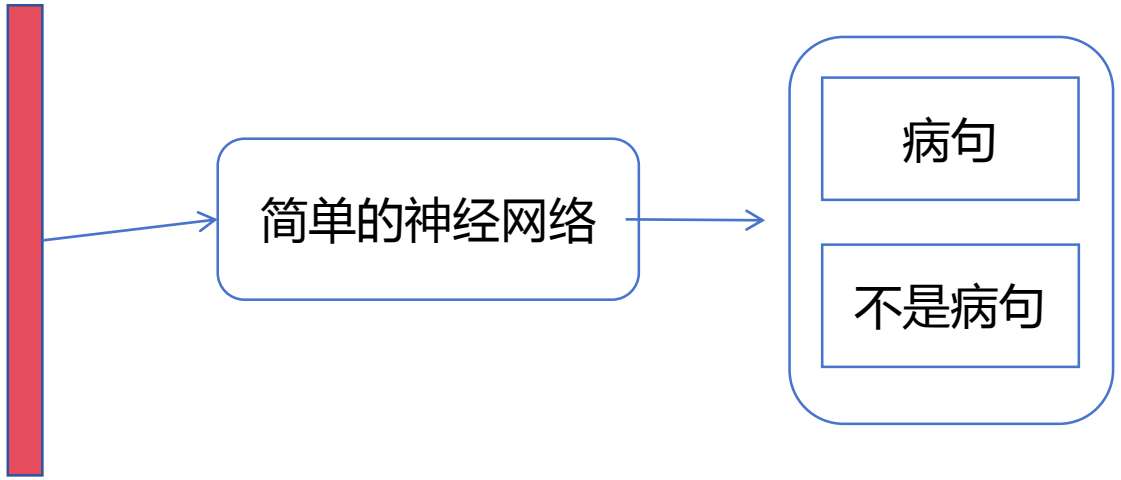
- 训练模型的输入
  - 输入
    - input_idx [1, 6, 4, 12, 5, 4, 8, 13, 7, 11, 2, 3, 3, 3, 3]
    - attention_mask [1,1,1,1,1,1,1,1,1,1,1,0,0,0,0]
  - 输出
    - 对每一个token的 768维的编码
    - [batch_size, seq_len, 768]
    - 这一个编码综合考虑了 序列的上下文信息，其中cls token的编码代表整条序列的信息，每一个token的编码代表对应token的信息
    - 就是说 模型输出的 长度为 768的向量是 整合了这条序列上下文交互的结果

label [-100, -100, 9, -100, -100, 10, -100, -100, -100, -100, -100, -100, -100, -100, -100]

模型用每一个token的 768维向量，解码出这个token到底是什么。然后用label 和 模型预测的计算交叉熵损失

关于模型的微调





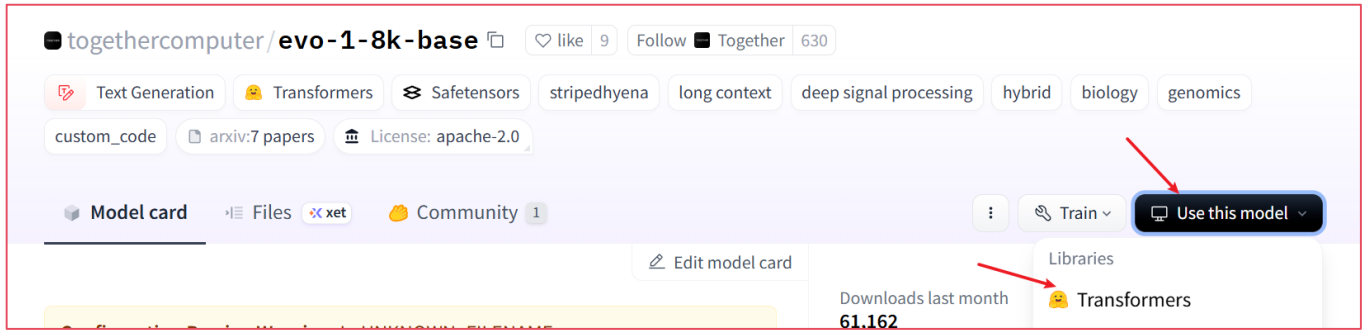
Transformers库介绍和使用示例

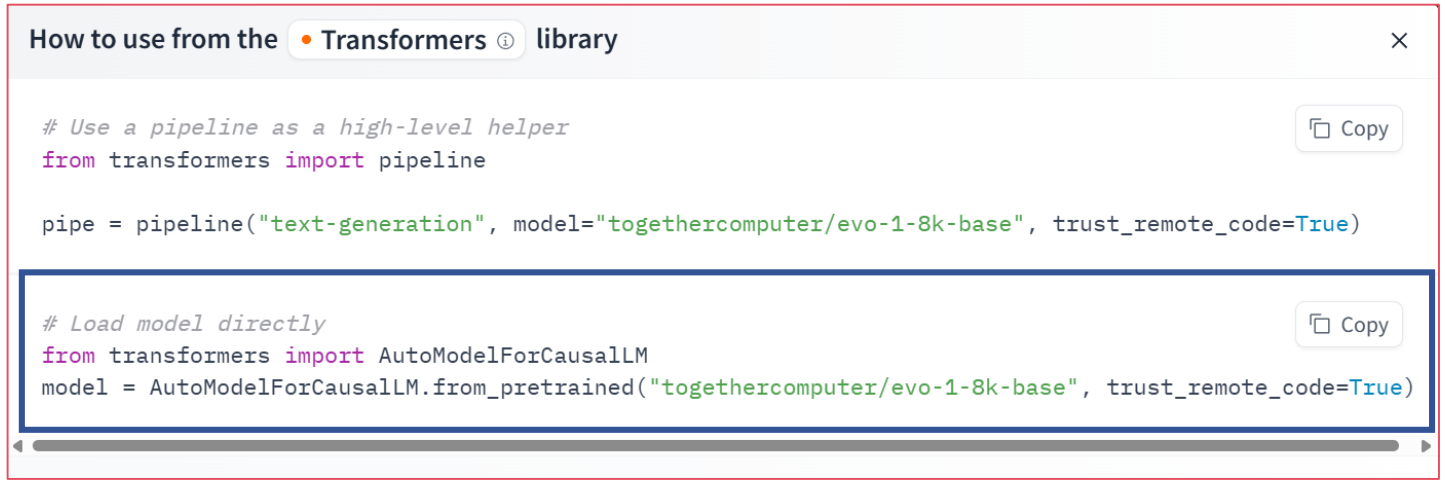
Transformers 库的基本介绍

- 来自 Hugging Face
- 一个很方便的包，很方便地帮我们完成以下事情
  - 根据标准化的配置文件(标准化的意思是有**特定的参数**)，帮我们自动化地定义模型很分词器
  - 通过一行代码，帮我们自动下载别人的模型结构 和 权重，然后帮我们载入预训练好的模型(**会存在科学上网问题**)
  - 少数几行代码(官方定义的接口)，完成模型的训练(预训练+微调)

- 集成程度高，直接载入别人预训练好的模型来用很简单，但是自己用transformers的接口训练微调模型难度大一些，主要表现为，需要通过不断查文档知道每一个类有什么用，输入规范是什么，输出规范是什么。

使用Hugging Face 载入别人的预训练模型





在国外的话，这两行代码就能够很方便地帮我们完成以下工作
自动下载模型权重和分词器到~/.cache/huggingface/hub/ 这一个默认路径

然后导入模型的时候，就会默认去这个路径看看有无下载使用该模型需要的文件，包括分词器，模型权重，模型配置

```
(base) [zyd fyp-X10DAi /home/zyd] WORK 0
# ls ~/.cache/huggingface/hub/
datasets--HuggingFace-CN-community--Diffusion-book-cn      models--Rostlab--prot_bert_bfd                 models--togethercomputer--evo-1-8k-base
datasets--HuggingFace-CN-community--Diffusion-book-cn.tar  models--Rostlab--prot_t5_xl_half_uniref50-enc  version.txt
evo-1-8k-base                                              models--togethercomputer--evo-1-131k-base
```

transformers自动pull下来的示例

main ▾ evo-1-8k-base

Zymrael Update README.md a9be7b6 VERIFIED

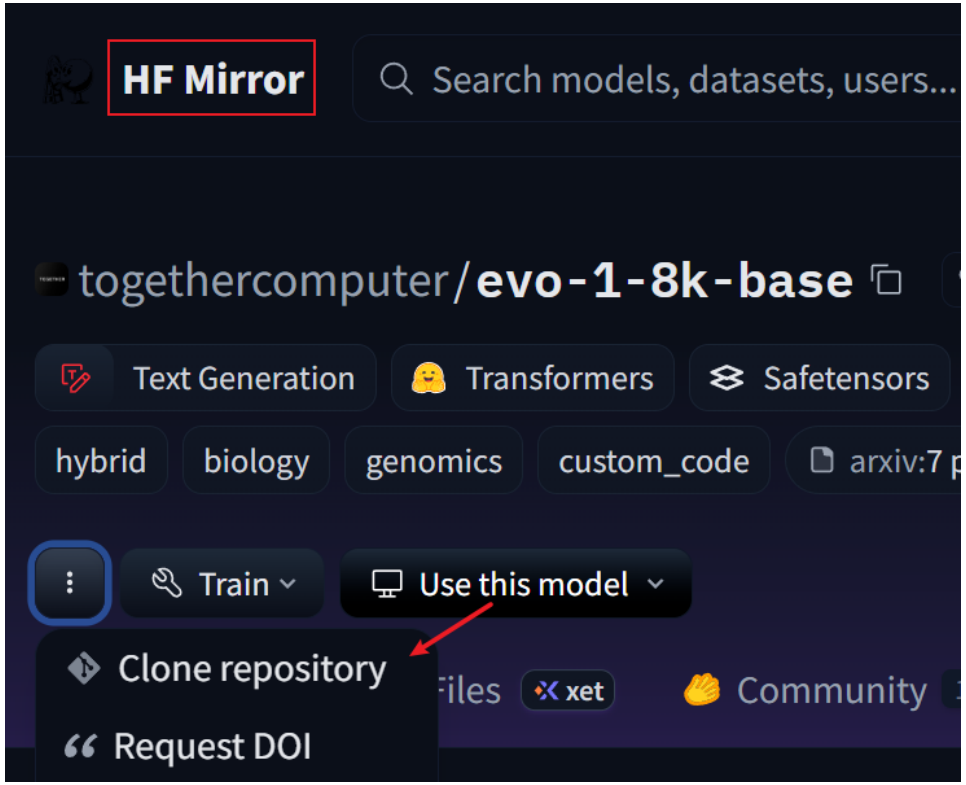
.gitattributes	Safe	1.52 kB
README.md	Safe	5.27 kB
config.json	Safe	1.86 kB
generation_config.json	Safe	69 Bytes
model-00001-of-00003.safetensors	Safe	4.98 GB xet
model-00002-of-00003.safetensors	Safe	4.93 GB xet
model-00003-of-00003.safetensors	Safe	3 GB xet
model.safetensors.index.json	Safe	34.9 kB
pytorch_model.pt	Safe pickle	16.8 GB xet
special_tokens_map.json		3 Bytes
tokenizer_config.json	Safe	299 Bytes

```
(base) [zyd fyp-X10DAi /home/zyd/.cache/huggingface/hub] WORK 0
# tree -alh ./models--togethercomputer--evo-1-8k-base/
./models--togethercomputer--evo-1-8k-base/
├── [4.0K]  blobs
│   ├── [1.1G]  4a08792f22697584c4b0c6cd1729902bc993ad7396b76f5caf6d7cc2b32ab882.incomplete
│   ├── [ 34K]  b08da632da9a41606e63792f414afcd7153488cf
│   └── [1.8K]  f5c2e2cb7bedd60b2d77672093bab04906281c0a
├── [4.0K]  .no_exist
│   └── [4.0K]  a9be7b66485080893399ade87c7d34f81ad3e249
│       └── [   0]  model.safetensors
├── [4.0K]  refs
│   └── [  40]  main
└── [4.0K]  snapshots
    └── [4.0K]  a9be7b66485080893399ade87c7d34f81ad3e249
        ├── [  34]  config.json -> ../../../evo-1-8k-base/config.json
        ├── [  45]  generation_config.json -> ../../../evo-1-8k-base/generation_config.json
        ├── [  55]  model-00001-of-00003.safetensors -> ../../../evo-1-8k-base/model-00001-of-00003.safetensors
        ├── [  55]  model-00002-of-00003.safetensors -> ../../../evo-1-8k-base/model-00002-of-00003.safetensors
        ├── [  55]  model-00003-of-00003.safetensors -> ../../../evo-1-8k-base/model-00003-of-00003.safetensors
        ├── [  51]  model.safetensors.index.json -> ../../../evo-1-8k-base/model.safetensors.index.json
        ├── [  39]  pytorch_model.pt -> ../../../evo-1-8k-base/pytorch_model.pt
        ├── [  32]  README.md -> ../../../evo-1-8k-base/README.md
        ├── [  46]  special_tokens_map.json -> ../../../evo-1-8k-base/special_tokens_map.json
        └── [  44]  tokenizer_config.json -> ../../../evo-1-8k-base/tokenizer_config.json
```

如果不解决科学上网问题，那么就会因为下载不下来直接报错

transformers的科学上网问题简单解决方案

- 使用 hf-mirror.com 镜像，git clone 到本地，然后通过本地绝对路径导入




```
git lfs install
git clone https://hf-mirror.com/togethercomputer/evo-1-8k-base
```

假如 我 git clone 到 /data12T/evo-1-8k-base目录

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained("/data12T/evo-1-8k-base", trust_remote_code=True)
```

```
./evo-1-8k-base/
├── [1.8K]  config.json
├── [  69]  generation_config.json
├── [1.5K]  .gitattributes
├── [4.6G]  model-00001-of-00003.safetensors
├── [4.6G]  model-00002-of-00003.safetensors
├── [2.8G]  model-00003-of-00003.safetensors
├── [ 34K]  model.safetensors.index.json
├── [ 16G]  pytorch_model.pt
├── [5.1K]  README.md
├── [   3]  special_tokens_map.json
└── [ 299]  tokenizer_config.json
```

DNA-ModernBERT的预训练实现

如何使用Transformers训练DNA-ModernBERT

- 目标
  - 使用**DNA Bert2 的分词器 和 预训练数据集**，基于transformers 训练一个ModernBERT 模型，然后使用增强子数据集进行二分类微调
- 数据集
  - /data12T/data_zyd/deeplearning_lab/data/dna_modern_bert/pretrain_data/dev.txt
  - txt文本文件，每一行是 1000 bp的基因组序列，一共有 100,000 条序列 (从DNABert2训练数据集中分出来的)

如何使用Transformers训练DNA-ModernBERT

- 分词器
  - 直接使用DNABert2 的BPE编码的分词器

```
token_path = "../data/dna_modern_bert/tokenizer_config"
tokenizer = AutoTokenizer.from_pretrained(token_path,use_fast_tokenizer=True,use_fast=True)
```

```
./tokenizer_config/
├── tokenizer_config.json
└── tokenizer.json
```

tokenizer_config.json

```
{"tokenizer_class": "PreTrainedTokenizerFast", "unk_token": "[UNK]", "cls_token": "[CLS]", "sep_token": "[SEP]", "pad_token": "[PAD]", "mask_token": "[MASK]"}
```

tokenizer.json

```
"model": {
  "vocab": {
    "[UNK]": 0,
    "[CLS]": 1,
    "[SEP]": 2,
    "[PAD]": 3,
    "[MASK]": 4,
    "A": 5,
    "C": 6,
    "G": 7,
    "T": 8,
    "AA": 9,
    "TT": 10,
    "TG": 11,
    "CA": 12,
    "CC": 13,
    "TA": 14,
    "GG": 15,
    "TC": 16,
    "GA": 17,
    "AAA": 18,
    "GC": 19,
    "TAA": 20,
    "TTTT": 21,
    "TCA": 22,
    "TGA": 23,
    "TTA": 24,
    "GAA": 25,
    "TCC": 26,
    "CAA": 27,
    "CTG": 28,
```

包括5个special token在内，一共有 4096个 token

关于分词器的使用

```python

token_path = "../data/dna_modern_bert/tokenizer_config"
tokenizer = AutoTokenizer.from_pretrained(token_path,
                                          use_fast_tokenizer=True,
                                          use_fast=True)

seq = "AAATTTCCCGGGATCTATC"

seq_tokenizer = tokenizer(
    seq,
    padding="max_length",
    truncation=True,
    max_length=10,
    return_special_tokens_mask=True
)

for key in seq_tokenizer.keys():
    print(f"{key}: {seq_tokenizer[key]}")

```

✓ 0.0s

```
input_ids: [1, 193, 1911, 1426, 16, 2, 3, 3, 3, 3]
token_type_ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
attention_mask: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
special_tokens_mask: [1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

```json
{
  "vocab": {
    "[UNK]": 0,
    "[CLS]": 1,
    "[SEP]": 2,
    "[PAD]": 3,
    "[MASK]": 4,
    "AAATT": 193,
    "TCCCGG": 1911,
    "GATCTA": 1426,
    "TC": 16
  }
}
```

第几句话

如何进行遮蔽掩码

遮蔽掩码前的

```
input_ids: [1, 193, 1911, 1426, 16, 2, 3, 3, 3, 3]
token_type_ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
attention_mask: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
special_tokens_mask: [1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

```python
# 这个会随机对 非special_token 进行遮蔽掩码
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True, # 是否进行MLM任务
    mlm_probability = 0.5,
    mask_replace_prob = 0.8,
    random_replace_prob = 0.1
)

mask_tokenizer = data_collator([seq_tokenizer])
for key in mask_tokenizer.keys():
    print(f"{key}: {mask_tokenizer[key]}")
```

✓ 0.0s

```
input_ids: tensor([[   1,    4, 1911, 1426,    4,    2,    3,    3,    3,    3]])
token_type_ids: tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
attention_mask: tensor([[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]])
labels: tensor([[-100,  193, -100, -100,   16, -100, -100, -100, -100, -100]])
```

随机遮蔽50%，对于该遮蔽的50%，80%使用【MASK】token遮蔽，10%使用其他词元替代遮蔽，剩下的就是不替换

如何进行遮蔽掩码

遮蔽掩码前的

```
input_ids: [1, 193, 1911, 1426, 16, 2, 3, 3, 3, 3]
token_type_ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
attention_mask: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
special_tokens_mask: [1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

```python
# 这个会随机对 非special_token 进行遮蔽掩码
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True, # 是否进行MLM任务
    mlm_probability = 0.5,
    mask_replace_prob = 0.1,
    random_replace_prob = 0.9
)

mask_tokenizer = data_collator([seq_tokenizer])
for key in mask_tokenizer.keys():
    print(f"{key}: {mask_tokenizer[key]}")
```

✓ 0.0s

```
input_ids: tensor([[   1,  151,    4, 1426,   16,    2,    3,    3,    3,    3]])
token_type_ids: tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
attention_mask: tensor([[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]])
labels: tensor([[-100,  193, 1911, -100, -100, -100, -100, -100, -100, -100]])
```

比较

随机遮蔽50%，对于该遮蔽的50%，10%使用【MASK】token遮蔽，90%使用其他词元替代遮蔽

关于Dataset对象的定义

```python
data_files = {
    "train":"../data/dna_modern_bert/pretrain_data/dev.txt",
    "validation":"../data/dna_modern_bert/pretrain_data/dev.txt"
}

data_cache_dir = "./dna_modern_bert_cache"
# 构建的数据集缓存目录，方便下次快速加载
# 如果你多次运行代码加载同一个数据集，datasets 库会检查指定的缓存目录
# 如果数据集已经存在，则直接从缓存加载，而不会重新下载，本地的也不需要重新计算。这可以节省时间和网络带宽。

raw_datasets = load_dataset(
    path = "text", # 说明输入文件是纯文本，按照纯文本的方式进行处理
    data_files=data_files,
    cache_dir=data_cache_dir # 一次加载后，会保存缓存文件，后面加载数据就会快很多
)

raw_datasets
```

✓ 0.2s

```
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 100000
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 100000
    })
})
```

```python
raw_datasets["train"][0]
```

✓ 0.0s

```
{'text': 'GGAGGTCCTTGTCCTGATTTTGTTCAAGGCTCCGGGTGGGCCATTCAATGCTTCTGATCTC
```

对Dataset对象进行 分词(tokenize)

- 本质上就是对 Dataset中的每一条序列进行 分词

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 100000
    })
    validation: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 100000
    })
})
```

```python
preprocessing_num_workers = 12
max_seq_length = 250

# 用于将 序列 转化为 token的进程数目
def tokenize_function(dataset_items):
    text_column_name = "text"
    padding = "max_length"
    # Remove empty lines
    dataset_items[text_column_name] = [
        line for line in dataset_items[text_column_name] if len(line) > 0 and not line.isspace()
    ]
    return tokenizer(
        dataset_items[text_column_name],
        padding=padding,
        truncation=True,
        max_length=max_seq_length,
        return_special_tokens_mask=True
    )

tokenized_datasets = raw_datasets.map(
                    tokenize_function,
                    batched=True,
                    num_proc=preprocessing_num_workers,
                    remove_columns=["text"],
                    # raw_datasets中的训练集和验证集都会有 text 这一列，代表原始的序列信息
                    # 在使用 tokenize_function 对他们进行编码以后，直接将原始的序列信息去除，节约内存
                    load_from_cache_file= True,
                    # 这个load_from_cache_file 不知道有什么用，重新载入 tokenized_datasets 这个数据集还是很慢
                    desc="Running tokenizer on dataset line_by_line",
                )
tokenized_datasets
```

```
1  tokenized_datasets["train"][0]
✓ 0.0s
{'input_ids': [1, 165, 130, 90, 703, 31, 303, 78, 15, 2],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 'special_tokens_mask': [1, 0, 0, 0, 0, 0, 0, 0, 0, 1]}
```

为了便于学习理解，我展示的是 max_length为10的结果。超过10的直接阶段，第一个token必须是【cls】，最后一个token必须是【sep】

max_length设置为 250比较合适，因为DNA Bert2的tokenizer编码后的长度会减少，设置最长为 250，能够确保后面的【padding】不会有太多

max_length设置不合理的示例

```python

token_path = "../data/dna_modern_bert/tokenizer_config"
tokenizer = AutoTokenizer.from_pretrained(token_path,
                                          use_fast_tokenizer=True,use_fast=True)

seq = "AAATTTCCCGGGATCTATC"

seq_tokenizer = tokenizer(
        seq,
        padding="max_length",
        truncation=True,
        max_length=40,
        return_special_tokens_mask=True
    )

for key in seq_tokenizer.keys():
    print(f"{key}: {seq_tokenizer[key]}")

```

✓ 0.0s

pad太多了，有效信息太少

```
input_ids: [1, 193, 1911, 1426, 16, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
token_type_ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
attention_mask: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
special_tokens_mask: [1, 0, 0, 0, 0, 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, 1, 1, 1, 1, 1, 1]
```

对训练过程进行评估的验证集数据设置

```python
train_dataset = tokenized_datasets["train"]
eval_dataset = tokenized_datasets["validation"]
print(f"过滤前的验证集样本数目 {len(eval_dataset)}")
"""
注意以下对验证集数据进行处理的代码
如果 验证集的序列很多，那么就会导致，模型在计算验证集的指标的时候，需要耗费很多时间，
因此要对验证集评估的序列数目进行限制
"""
max_eval_samples = 256
# 为了避免 eval_dataset的数据集太大，导致评估的时候计算时间太长，
# 设置 max_eval_samples 参数，将评估的时候样本数目进行限制
max_eval_samples = min(len(eval_dataset),max_eval_samples)
eval_dataset = eval_dataset.select(range(max_eval_samples))
print(f"过滤后的验证集样本数目 {len(eval_dataset)}")
```

✓ 0.0s

```
过滤前的验证集样本数目 100000
过滤后的验证集样本数目 256
```

因为这是一个教学示例，用的GPU显存也比较小，在训练过程中，对所有的验证集序列进行一个评估，会导致验证评估消耗太多的时间，因此对验证集进行采样(这里是简单取出前面256个样本)

这样子大家就能比较快速地运行完示例，看到效果

对模型的定义--ModernBERT

- ModertBERT 在 Transformers == 4.51.3 中有标准化实现接口，我们只需要通过config.json配置文件，就能够实现ModernBERT模型的定义

```
{
  "vocab": {
    "[UNK]": 0,
    "[CLS]": 1,
    "[SEP]": 2,
    "[PAD]": 3,
    "[MASK]": 4
  }
}
```

```
{
  "_name_or_path": "ModernBERT-base",
  "architectures": [
    "ModernBertForMaskedLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "classifier_activation": "gelu",
  "classifier_bias": false,
  "classifier_dropout": 0.0,
  "classifier_pooling": "mean",
  "cls_token_id": 1,
  "decoder_bias": true,
  "deterministic_flash_attn": false,
  "embedding_dropout": 0.0,
  "eos_token_id": 2,
  "global_attn_every_n_layers": 3,
  "global_rope_theta": 160000.0,
  "gradient_checkpointing": false,
  "hidden_activation": "gelu",
  "hidden_size": 768,
  "initializer_cutoff_factor": 2.0,
  "initializer_range": 0.02,
  "intermediate_size": 1152,
  "layer_norm_eps": 1e-05,
  "local_attention": 128,
  "local_rope_theta": 10000.0,
  "max_position_embeddings": 8192,
  "mlp_bias": false,
  "mlp_dropout": 0.0,
  "model_type": "modernbert",
  "norm_bias": false,
  "norm_eps": 1e-05,
  "num_attention_heads": 12,
  "num_hidden_layers": 8,
  "pad_token_id": 3,
  "position_embedding_type": "absolute",
  "sep_token_id": 2,
  "tie_word_embeddings": true,
  "torch_dtype": "float32",
  "transformers_version": "4.47.0.dev0",
  "vocab_size": 4096
}
```

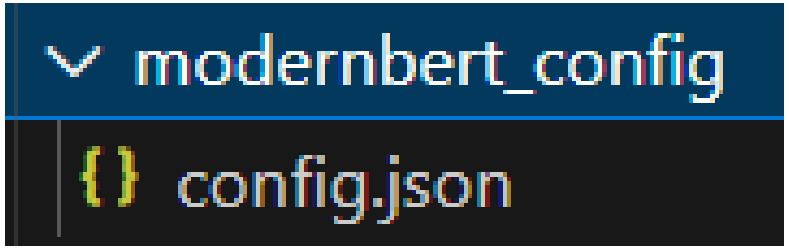
主要关注红色框框中的参数设置

模型调小，5G显存就能预训练

ModernBERT的定义

```
config = AutoConfig.from_pretrained("../data/dna_modern_bert/modernbert_config")
model = AutoModelForMaskedLM.from_config(config)
model
```

该目录只有 config.json这个文件



print(model)
就可以打印出 定义好的模型的结构，方便我们动手推每一层的输入输出维度，定位网络设计问题

```
ModernBertForMaskedLM(
  (model): ModernBertModel(
    (embeddings): ModernBertEmbeddings(
      (tok_embeddings): Embedding(4096, 768, padding_idx=3)
      (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (drop): Dropout(p=0.0, inplace=False)
    )
    (layers): ModuleList(
      (0): ModernBertEncoderLayer(
        (attn_norm): Identity()
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
      (1-7): 7 x ModernBertEncoderLayer(
        (attn_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
    )
    (final_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (head): ModernBertPredictionHead(
    (dense): Linear(in_features=768, out_features=768, bias=False)
    (act): GELUActivation()
    (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (decoder): Linear(in_features=768, out_features=4096, bias=True)
)
```

Trainer对象的定义

- 有了Dataset，有了Model，就可以用transformers的Trainer对象开始预训练
- 前面的鸢尾花分类器，snp预测模型都是在**一个epoch以后就评估一次**的
- 然而，大模型的预训练数据很大，大语言模型预训练可能也就训练1个epoch，因此训练过程的验证集评估要在step水平进行评估，比如**每每10k个step就用验证集评估一次**
- 该简单示例只训练了 50个step，每每10个step验证集评估一次，然后保存记录模型

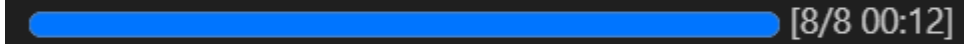
```
dict_training_args = dict(
    output_dir = "./DNA_modern_bert_pretrain/",
    learning_rate = 1e-4,
    max_steps = 50,
    per_device_train_batch_size = 32,
    per_device_eval_batch_size = 32,
    save_total_limit = 2,
    eval_strategy = "steps",
    save_strategy = "steps",
    eval_steps = 10, save_steps = 10,
    logging_steps = 10,
    overwrite_output_dir = True,
    bf16 = True,
    gradient_accumulation_steps = 1,
    dataloader_num_workers = 4,
    remove_unused_columns = False,
    lr_scheduler_type = 'cosine',
    warmup_ratio = 0.05,
    log_level = "info"
)
training_args = TrainingArguments(
    **dict_training_args
)

trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset ,
        eval_dataset=eval_dataset ,
        processing_class=tokenizer,
        data_collator=data_collator,
        compute_metrics=compute_metrics,
        preprocess_logits_for_metrics=preprocess_logits_for_metrics
        # preprocess_logits_for_metrics 必须写一下，不然评估的时候很容易爆显存
    )
```

预训练之前的验证集评估

```
# 训练开始之前，验证集评估一下
trainer.evaluate()
```

```
***** Running Evaluation *****
  Num examples = 256
  Batch size = 32
```




```
{'eval_loss': 8.431022644042969,
 'eval_model_preparation_time': 0.0013,
 'eval_accuracy': 0.00019615535504119262,
 'eval_runtime': 5.2643,
 'eval_samples_per_second': 48.63,
 'eval_steps_per_second': 1.52}
```

开始预训练

```python
# 模型训练和结果保存
train_result = trainer.train()
trainer.save_model()
metrics = train_result.metrics
trainer.log_metrics("train", metrics)
trainer.save_metrics("train", metrics)
trainer.save_state()
```

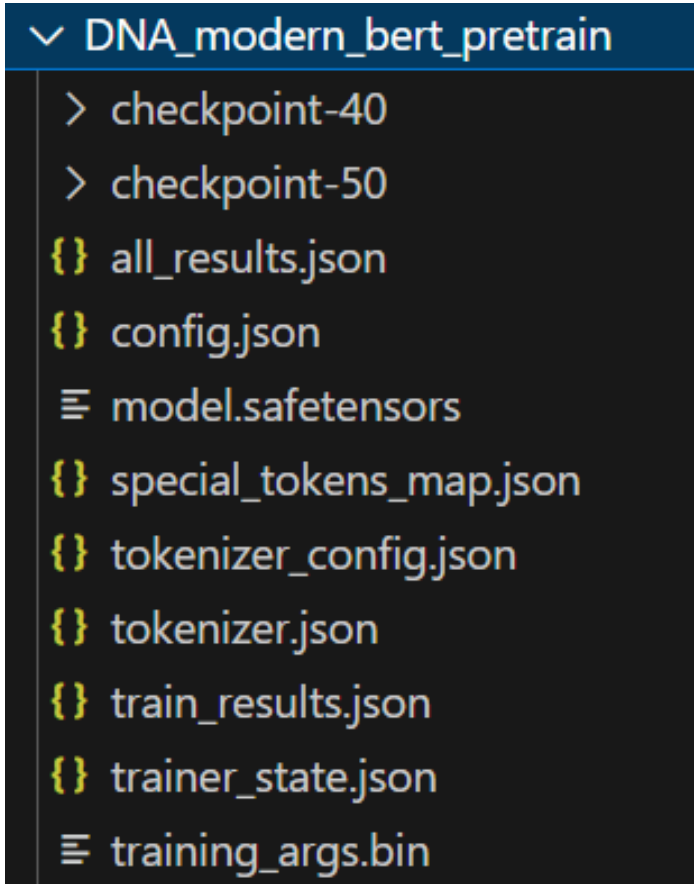
```
***** Running training *****
  Num examples = 100,000
  Num Epochs = 1
  Instantaneous batch size per device = 32
  Total train batch size (w. parallel, distributed & accumulation) = 32
  Gradient Accumulation steps = 1
  Total optimization steps = 50
  Number of trainable parameters = 43,861,504
```

[50/50 00:29, Epoch 0/1]

Step	Training Loss	Validation Loss	Model Preparation Time	Accuracy
10	8.104700	7.747279	0.001300	0.018140
20	7.547900	7.367887	0.001300	0.029506
30	7.267200	7.189133	0.001300	0.050231
40	7.136900	7.116362	0.001300	0.052867
50	7.102400	7.115716	0.001300	0.050860

预训练输出结果

模型结构，模型权重，分词器配置，训练过程的指标



使用预训练模型进行embedding

- 关于模型的载入，**必须使用 AutoModel 类**，而不是 AutoModelForMaskedLM 类
- 见下一页模型结构分析

```
ModernBertForMaskedLM(
  (model): ModernBertModel(
    (embeddings): ModernBertEmbeddings(
      (tok_embeddings): Embedding(4096, 768, padding_idx=3)
      (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (drop): Dropout(p=0.0, inplace=False)
    )
    (layers): ModuleList(
      (0): ModernBertEncoderLayer(
        (attn_norm): Identity()
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
      (1-7): 7 x ModernBertEncoderLayer(
        (attn_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
    )
    (final_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (head): ModernBertPredictionHead(
    (dense): Linear(in_features=768, out_features=768, bias=False)
    (act): GELUActivation()
    (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (decoder): Linear(in_features=768, out_features=4096, bias=True)
)
```

```
ModernBertModel(
  (embeddings): ModernBertEmbeddings(
    (tok_embeddings): Embedding(4096, 768, padding_idx=3)
    (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (drop): Dropout(p=0.0, inplace=False)
  )
  (layers): ModuleList(
    (0): ModernBertEncoderLayer(
      (attn_norm): Identity()
      (attn): ModernBertAttention(
        ...
      )
      (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (mlp): ModernBertMLP(
        ...
      )
    )
    (1-7): 7 x ModernBertEncoderLayer(
      (attn_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (attn): ModernBertAttention(
        ...
      )
      (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (mlp): ModernBertMLP(
        ...
      )
    )
  )
  (final_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
```

左边是AutoModelForMaskedLM
右边是AutoModel
AutoModelForMaskedLM 多了head 和 decoder
而我们只需要 final_norm 的 768维度的embedding

使用预训练模型进行embedding

```python
# 预训练模型的目录
pretrain_model_path = "./DNA_modern_bert_pretrain"
model = AutoModel.from_pretrained(pretrain_model_path)
# 必须是AutoModel，不能是AutoModelForMaskedLM，不然输出的维度是4096，而不是768
tokenizer = AutoTokenizer.from_pretrained(pretrain_model_path,use_fast_tokenizer=True,use_fast=True)
```

```python
seq1 = "AAATTTCCCGGGTTTT"
seq2 = "AAATTTCCCGGGTTTT"
max_seq_length = 20
seq_tokenized = tokenizer(
    [seq1,seq2],
    padding="max_length",
    truncation=True,
    max_length=max_seq_length,
    return_special_tokens_mask=True,
    return_tensors = "pt"
    )
seq_tokenized.keys(),seq_tokenized
```

seq_tokenized的格式

【cls】标签

```
(
 dict_keys(['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask']),

{'input_ids': tensor(
   [[  1, 193, 1911,  87,  2,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3],
    [  1, 193, 1911,  87,  2,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3]]),
  'token_type_ids': tensor(
   [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]),
  'attention_mask': tensor(
   [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]),
  'special_tokens_mask': tensor(
   [[1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}

)
```

使用预训练模型进行embedding

seq_tokenized[“input_ids”] 【batch_size, seq_len=20】
seq_tokenized[“attention_mask”] 【batch_size, seq_len=20】

```
output_logits = model(input_ids = seq_tokenized["input_ids"],
                      attention_mask = seq_tokenized["attention_mask"])
output_logits["last_hidden_state"].shape
```

```
torch.Size([2, 20, 768])
```

```
cls_logits = output_logits["last_hidden_state"][:,0,:]
cls_logits.shape
```

```
torch.Size([2, 768])
```

取出 代表这一整条序列信息的 [cls]标签的embedding
【batch_size, embdding_dim】

DNA-ModernBERT微调增强子分类器

微调数据集介绍

- /data12T/data_zyd/deeplearning_lab/data/dna_modern_bert/finetune_data_enhancer_classifier
- tsv表格 两列，分别是seq和label
  - seq 是200bp 的序列
  - label是 0或者1，1说明是增强子

seq	label
CATTCGATCAAACGCGTA	0
TCTGCCAACAGCTGTCAG	0
GGAAAAAGTTCGCGCCAA	0
GAATCCCACAAAATGACCT	1
TGAGAAGTACCACAAAGG	0
AAGTTGGCAGGCACAGCT	1
CACAGACTGGCTTCTATAC	0
TGTAGATAACCAGCTGTG	0
GTTTTCCCGCCACAACCGT	0
AAACCGCACGCGAGTAAG	1
CAGAGTCCCAAGGTCTGAG	1
TAACCAGATCAAGGAAAA	1
CGCTTATTTGGTTATCCAC	0
TTAGAGCAGGGGTGCACA	1
TACGGTAGGGTGGAAACC	0
ATAACTGTCCAAAGTCCAA	1
GCAGCGGGTCAGATTAGA	0
ACTCCCATTTCCATAGGTG	1
GGCTGTTATCTTTTCCCTCT	0
CCACCAACTTATCATATCT	0
CCCAAAAGCGTCGAATGTA	0
CACGCGAGGGTTAGTACC	1
ACCACATGCTAGGCGGGA	0
ATCATTAACCATGATTCTT	1
TAGATGAGGCTTAGCGCT	1

使用前面预训练模型进行微调

传入 刚刚**预训练模型的输出目录** ，用于生成文本分类模型和分词器

```
# 预训练模型的目录
pretrain_model_path = "./DNA_modern_bert_pretrain"
model = AutoModelForSequenceClassification.from_pretrained(pretrain_model_path,num_labels=2)
# num_labels 就是指定分类器输出的特征数目

token_path = "./DNA_modern_bert_pretrain"
tokenizer = AutoTokenizer.from_pretrained(token_path,use_fast_tokenizer=True,use_fast=True)
```

因为是增强子序列分类问题，所以是二分类，num_labels设置为2

Dataset的定义

```python
class Enhancer_classifier_dataset(Dataset):
    def __init__(self,tsvfile,tokenizer,max_seq_length):
        super().__init__()
        self.tokenizer = tokenizer
        self.max_seq_length = max_seq_length
        self.df = pd.read_csv(tsvfile,sep="\t")
    def __len__(self):
        return self.df.shape[0]
    def get_seq_tokenized(self,seq):
        seq_tokenized = tokenizer(
            seq,
            padding = "max_length",
            truncation=True,
            max_length=self.max_seq_length,
            return_special_tokens_mask=True,
            return_tensors = "pt"
        )
        return {"input_ids":seq_tokenized['input_ids'].squeeze(),"attention_mask":seq_tokenized
        ['attention_mask'].squeeze()}
        # 对于一条序列的处理，tokenizer输出的维数是【1，max_seq_length】，需要squeeze()处理变成
        【max_seq_length】，不然训练的时候会报错
    def __getitem__(self, index):
        # 最开始的序列，是不包括其他标签的
        seq = self.df['seq'].values[index]
        label = self.df['label'].values[index].item()
        seq_tokenized = self.get_seq_tokenized(seq)
        seq_tokenized["label"] = label
        return seq_tokenized
        # 返回的字典 键名必须是 input_ids, attention_mask, label
```

Dataset的定义

```
train_dataset = Enhancer_classifier_dataset(tsvfile = "../data/dna_modern_bert/finetune_data_enhancer_classifier/train.tsv",
                                            tokenizer = tokenizer,max_seq_length = 50)
eval_dataset = Enhancer_classifier_dataset(tsvfile = "../data/dna_modern_bert/finetune_data_enhancer_classifier/valid.tsv",
                                           tokenizer = tokenizer,max_seq_length = 50)
test_dataset = Enhancer_classifier_dataset(tsvfile = "../data/dna_modern_bert/finetune_data_enhancer_classifier/test.tsv",
                                           tokenizer = tokenizer,max_seq_length = 50)
```

```
1  test_dataset[0]
```

```
{'input_ids': tensor([   1, 3698,  103,   40, 1051,   36,  166,   16,  402,  115,   32,  123,
        2475,   72,   35, 1221,  105,   77,  118,  708,  591,   13,  183,   93,
         107,  118,   35,  144,  137, 1601,   16,  239,  531, 2880, 1808, 1382,
         403,  332,   54,  303,   30,  864,   96,   42,   79, 1049,    2,    3,
           3,    3]),
 'attention_mask': 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
        0, 0]),
 'label': 0}
```

为什么设置max_seq_length为50?

增强子序列的数据集长度是 200bp，200/4 = 50，这样子pad就不会很多

关于模型结构和参数冻结

这里输出了
ModernBERT模型的embedding
维度[batch_size, seq_length ,768]

多了两个全连接层，也只有这两层的参数需要学习

注意: 传入head这一个块的，只有cls的embedding, 维度是[batch_size, 768]

```python
# 冻结预训练模型的参数
for param in model.model.parameters():
    param.requires_grad = False

```

```
ModernBertForSequenceClassification(
  (model): ModernBertModel(
    (embeddings): ModernBertEmbeddings(
      (tok_embeddings): Embedding(4096, 768, padding_idx=3)
      ...
    )
    (layers): ModuleList(
      (0): ModernBertEncoderLayer(
        (attn_norm): Identity()
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
      (1-7): 7 x ModernBertEncoderLayer(
        (attn_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): ModernBertAttention(
          ...
        )
        (mlp_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): ModernBertMLP(
          ...
        )
      )
    )
    (final_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (head): ModernBertPredictionHead(
    (dense): Linear(in_features=768, out_features=768, bias=False)
    (act): GELUActivation()
    (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (drop): Dropout(p=0.0, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

模型微调

1

```python
dict_training_args = dict(
    output_dir = "./DNA_modern_bert_fine_tune_Enhancer_classification/",
    learning_rate = 0.001,
    num_train_epochs = 5,
    per_device_train_batch_size = 2048,
    per_device_eval_batch_size = 1024,
    save_total_limit = 2,
    logging_steps = 10,
    overwrite_output_dir = True,
    eval_strategy = "epoch",
    save_strategy = "epoch",
    bf16 = True,
    gradient_accumulation_steps = 1,
    dataloader_num_workers = 4,
    remove_unused_columns = False,
    log_level = "info"
)
training_args = TrainingArguments(
    **dict_training_args
)

trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset ,
        eval_dataset=eval_dataset ,
        processing_class=tokenizer,
        compute_metrics=compute_metrics,
        preprocess_logits_for_metrics=preprocess_logits_for_metrics
        # preprocess_logits_for_metrics 必须写一下，不然评估的时候很容易爆显存
    )
```

2

```python
# 训练之前评估一次
trainer.evaluate()
```

```
{'eval_loss': 0.9198629856109619,
 'eval_model_preparation_time': 0.0018,
 'eval_accuracy': 0.39863130881094955,
 'eval_runtime': 7.0002,
 'eval_samples_per_second': 333.992,
 'eval_steps_per_second': 0.429}
```

3

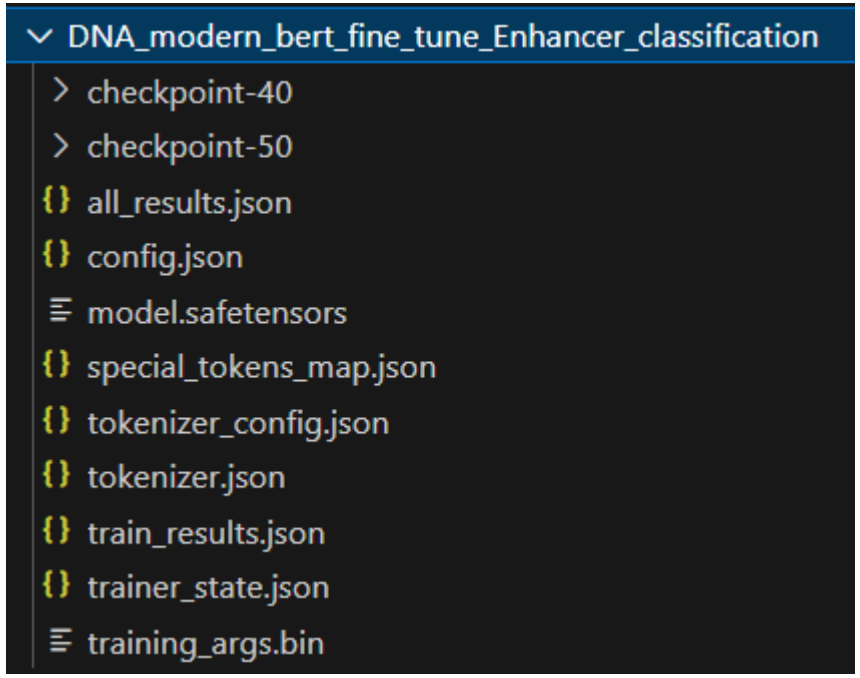
```python
# 模型训练和结果保存
train_result = trainer.train()
trainer.save_model()  # Saves the tokenizer too for easy upload
metrics = train_result.metrics
trainer.log_metrics("train", metrics)
trainer.save_metrics("train", metrics)
trainer.save_state()
```

```
***** Running training *****
  Num examples = 18,705
  Num Epochs = 5
  Instantaneous batch size per device = 2,048
  Total train batch size (w. parallel, distributed & accumulation) = 2,048
  Gradient Accumulation steps = 1
  Total optimization steps = 50
  Number of trainable parameters = 592,130
```

[50/50 01:18, Epoch 5/5]

Epoch	Training Loss	Validation Loss	Model Preparation Time	Accuracy
1	0.881800	0.298940	0.001400	0.883234
2	0.225200	0.209684	0.001400	0.917451
3	0.190000	0.182324	0.001400	0.926861
4	0.165900	0.176989	0.001400	0.928144
5	0.155900	0.173750	0.001400	0.932849

微调结果输出目录




```
# 预训练模型的目录
pretrain_model_path = "/data12T/data_zyd/deeplearning_lab/zyd/DNA_modern_bert_fine_tune_Enhancer_classification"
model = AutoModelForSequenceClassification.from_pretrained(pretrain_model_path,num_labels=2)
```

通过AutoModel，传入 **微调输出目录**，就能够载入模型

使用微调后的模型进行推理

```python
from transformers import AutoModelForSequenceClassification, AutoTokenizer
# 微调后的模型的目录
fine_tune_model_path = "./DNA_modern_bert_fine_tune_Enhancer_classification"
model = AutoModelForSequenceClassification.from_pretrained(fine_tune_model_path)
tokenizer = AutoTokenizer.from_pretrained(fine_tune_model_path,use_fast_tokenizer=True,use_fast=True)

seq1 =
"CCTCTCTGGGTGCTCGCGTCGCCCGAATCGCTGTGTTCAGCACGTGCAACAGGCGGGTAGGAGGAAGCTAGTTACGCGCGGATCAGGACCGCATTCGATCAAACC
TCGTAATCGCTCGGGAGGGGAGTTACAAGGAATGATCCCGCTGGGGCATCTGCAAGGGTGTGATAACAAAATGTGCTAAGT"
# 0 非增强子
seq2 =
"GTACCCGGGGCACTTGATAATCATTAACCAGGCTTAAACGCCTCGTGCCCATGCAACCAACCTGGTTAATCATTAACCAAGGTAGAGTATGAATCCCACAAAATC
GGTATAGGTGCGTTGTTGCCAAGTCTTGAGGCATGCTGGAACAGGTACTTTATGATACACTTTCCCGCATGGATTCTGGTC"
# 1 增强子
seq3 = "ATCG"*25
# 随便写的序列
max_seq_length = 50
seq_tokenized = tokenizer(
    [seq1,seq2,seq3],
    padding="max_length",
    truncation=True,
    max_length=max_seq_length,
    return_special_tokens_mask=True,
    return_tensors = "pt"
    )

logits = model(input_ids = seq_tokenized["input_ids"], attention_mask = seq_tokenized["attention_mask"])
```

logits的输出维度 【batch_size, 2】
最后的分类结果

```python
logits["logits"].argmax(-1)
```

```
tensor([0, 1, 0])
```

完全正确

总结

- 1. 了解到了遮蔽掩码模型的训练原理，输入输出都有什么，是什么格式(维度)，如何进行微调
- 2. 只需要将预训练数据集稍微改一下，处理一下分词器tokenizer，就能够完成遮蔽掩码大语言模型的训练
- 3. Transformers这个包帮我们定义了遮蔽掩码训练的接口实现和模型微调的接口实现，我们弄清楚每一个接口的意义，输入和输出格式，就能够很方便地通过这些接口完成语言模型的预训练和微调
- 4. Transformers的集成程度很高，平时如果我们直接用别人的预训练模型和微调模型的话，那么就是两行代码的事情。如果我们需要自定义地预训练和微调，就需要细读Transformers的文档，搞清楚每一个对象，每一个接口都有啥用，都怎么用，输入是什么，输出是什么，不然的话，Transformers是很难用于预训练和微调的。