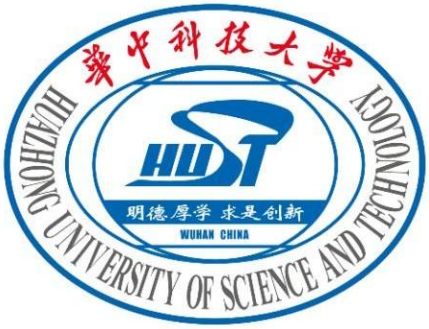


视觉与自然语言处理

中期作业

探究主流 NLP 模型：结构、参数与显存占用

团队成员及分工

姓名	学号	班级	项目分工	贡献比
叶庭宏	U202215212	人工智能 2203 班	GPU 显存占用测量，可视化分析，报告撰写	40%
胡耀文	U202215135	自动化 2203 班	模型选择，模型加载与结构分析，报告撰写	30%
李飞扬	U202215169	人工智能 2204 班	模型参数量计算与验证，报告撰写整合排版	30%

2025 年　6 月　17　日

1. 模型结构分析.....3
1.1 模型加载与定义：.....3
- Transformer(T5-small).....3
- CNN(VGG16).....3
- Bert.....3
1.2 打印模型结构：.....4
- Transformer(T5-small).....4
- CNN(VGG16).....7
- Bert.....8
1.3 核心组件分析：.....9
- Transformer(T5-small).....9
- CNN(VGG16).....11
- Bert.....13
2. 模型参数量计算与验证.....15
2.1 推导参数计算公式：.....15
- Transformer(T5-small).....15
- CNN(VGG16).....16
- Bert.....17
2.2 统计与验证：.....18
- Transformer(T5-small).....18
- CNN(VGG16).....20
- Bert.....21
3. GPU 显存占用测量.....22
3.1 测试环境检查.....22
3.2 虚拟输入构造.....22
3.3 显存测试结果分析.....23
3.3.1 VGG16 模型.....24
3.3.2 T5 模型.....24
3.4 模型结构与显存关系.....27
3.5 核心代码展示.....29
3.5.1 显存测量核心方法：.....29
3.5.2 显存测量调用流程：.....29

1. 模型结构分析

1.1 模型加载与定义：

- Transformer(T5-small)

加载与定义 T5-small 模型：

```
from transformers import T5Model

# 加载 t5-small 预训练模型
model = T5Model.from_pretrained("t5-small")

# 打印完整模型结构
print("="*60)
print("T5-small 模型完整结构:")
print(model)
```

- CNN(VGG16)

加载与定义 VGG16 模型：：

```
import torchvision.models as models

# 加载预训练的 VGGNet 模型
model = models.vgg16(pretrained=True)

# 打印模型的详细层级结构
print(model)
```

- Bert

加载与定义 Bert 模型：：

```
from transformers import BertModel, BertTokenizer
import torch

# 加载 BERT 基础模型（uncased 版本）和分词器
model = BertModel.from_pretrained('bert-base-uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

print(model)
```

1.2 打印模型结构：

- Transformer(T5-small)

```
T5Model(
  (shared): Embedding(32128, 512)
  (encoder): T5Stack(
    (embed_tokens): Embedding(32128, 512)
    (block): ModuleList(
      (0): T5Block(
        (layer): ModuleList(
          (0): T5LayerSelfAttention(
            (SelfAttention): T5Attention(
              (q): Linear(in_features=512, out_features=512, bias=False)
              (k): Linear(in_features=512, out_features=512, bias=False)
              (v): Linear(in_features=512, out_features=512, bias=False)
              (o): Linear(in_features=512, out_features=512, bias=False)
              (relative_attention_bias): Embedding(32, 8)
            )
            (layer_norm): T5LayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (1): T5LayerFF(
            (DenseReluDense): T5DenseActDense(
              (wi): Linear(in_features=512, out_features=2048, bias=False)
              (wo): Linear(in_features=2048, out_features=512, bias=False)
              (dropout): Dropout(p=0.1, inplace=False)
              (act): ReLU()
            )
            (layer_norm): T5LayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
      (1-5): 5 x T5Block(
        (layer): ModuleList(
          (0): T5LayerSelfAttention(
            (SelfAttention): T5Attention(
              (q): Linear(in_features=512, out_features=512, bias=False)
              (k): Linear(in_features=512, out_features=512, bias=False)
              (v): Linear(in_features=512, out_features=512, bias=False)
```

```
                (o): Linear(in_features=512, out_features=512, bias=False)
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (1): T5LayerFF(
              (DenseReluDense): T5DenseActDense(
                (wi): Linear(in_features=512, out_features=2048, bias=False)
                (wo): Linear(in_features=2048, out_features=512, bias=False)
                (dropout): Dropout(p=0.1, inplace=False)
                (act): ReLU()
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (final_layer_norm): T5LayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (decoder): T5Stack(
      (embed_tokens): Embedding(32128, 512)
      (block): ModuleList(
        (0): T5Block(
          (layer): ModuleList(
            (0): T5LayerSelfAttention(
              (SelfAttention): T5Attention(
                (q): Linear(in_features=512, out_features=512, bias=False)
                (k): Linear(in_features=512, out_features=512, bias=False)
                (v): Linear(in_features=512, out_features=512, bias=False)
                (o): Linear(in_features=512, out_features=512, bias=False)
                (relative_attention_bias): Embedding(32, 8)
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (1): T5LayerCrossAttention(
              (EncDecAttention): T5Attention(
                (q): Linear(in_features=512, out_features=512, bias=False)
                (k): Linear(in_features=512, out_features=512, bias=False)
                (v): Linear(in_features=512, out_features=512, bias=False)
```

```
                (o): Linear(in_features=512, out_features=512, bias=False)
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (2): T5LayerFF(
              (DenseReluDense): T5DenseActDense(
                (wi): Linear(in_features=512, out_features=2048, bias=False)
                (wo): Linear(in_features=2048, out_features=512, bias=False)
                (dropout): Dropout(p=0.1, inplace=False)
                (act): ReLU()
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
        (1-5): 5 x T5Block(
          (layer): ModuleList(
            (0): T5LayerSelfAttention(
              (SelfAttention): T5Attention(
                (q): Linear(in_features=512, out_features=512, bias=False)
                (k): Linear(in_features=512, out_features=512, bias=False)
                (v): Linear(in_features=512, out_features=512, bias=False)
                (o): Linear(in_features=512, out_features=512, bias=False)
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (1): T5LayerCrossAttention(
              (EncDecAttention): T5Attention(
                (q): Linear(in_features=512, out_features=512, bias=False)
                (k): Linear(in_features=512, out_features=512, bias=False)
                (v): Linear(in_features=512, out_features=512, bias=False)
                (o): Linear(in_features=512, out_features=512, bias=False)
              )
              (layer_norm): T5LayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (2): T5LayerFF(
              (DenseReluDense): T5DenseActDense(
                (wi): Linear(in_features=512, out_features=2048, bias=False)
```

```
                    (wo): Linear(in_features=2048, out_features=512, bias=False)
                    (dropout): Dropout(p=0.1, inplace=False)
                    (act): ReLU()
                  )
                  (layer_norm): T5LayerNorm()
                  (dropout): Dropout(p=0.1, inplace=False)
                )
              )
            )
          )
          (final_layer_norm): T5LayerNorm()
          (dropout): Dropout(p=0.1, inplace=False)
        )
)
```

- CNN(VGG16)

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
```

```
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
)
```

- Bert

```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0-11): 12 x BertLayer(
        (attention): BertAttention(
```

```
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
)
```

1.3 核心组件分析：

- Transformer(T5-small)

整个模型由**共享权重嵌入层**、**编码器**和**解码器**三部分组成。

```
(shared): Embedding(32128, 512)
```

Share 表示这是一个共享的嵌入层，即模型的输入词嵌入和输出词嵌入共用同一个权重矩阵；

Embedding(32128, 512)表示词表的大小为32128，词的嵌入维度为512维。这样的共享权重可以减少模型参数量，避免单独维护两个大的矩阵，同时也有助于对齐编码和解码的表示空间，提升生成任务的一致性。

编码器由由 **嵌入层、6 个 T5Block 模块** 和 **输出归一化层** 组成：

```
(encoder): T5Stack(
    (embed_tokens): Embedding(32128, 512)      # 词嵌入层
    (block): ModuleList( ... )                 # 6 个 T5Block 堆叠
    (final_layer_norm): T5LayerNorm()          # 最终层归一化
    (dropout): Dropout(p=0.1)                  # 随机失活
)
```

编码器的每个 T5Block 包含两个核心子层：**自注意力层**（T5LayerSelfAttention）和**前馈网络层**（T5LayerFF）。

自注意力层的内容如下：

```
  (SelfAttention): T5Attention(
        (q): Linear(in_features=512, out_features=512, bias=False)
        (k): Linear(in_features=512, out_features=512, bias=False)
        (v): Linear(in_features=512, out_features=512, bias=False)
        (o): Linear(in_features=512, out_features=512, bias=False)
        (relative_attention_bias): Embedding(32, 8)
  )
   (layer_norm): T5LayerNorm()
   (dropout): Dropout(p=0.1, inplace=False)
```

由输出的层级结构可见，自注意力层首先通过无偏置的线性层（q/k/v）将512维输入分别投影为查询（Query）、键（Key）、值（Value）向量，然后计算查询与键的点积并通过相对位置编码（32个位置桶的8维嵌入）引入位置信息，缩放后经softmax得到注意力权重，再与值向量加权求和，最后通过输出投影层（o）整合结果。

前馈网络层的内容如下：

```
(DenseReluDense): T5DenseActDense(
    (wi): Linear(in_features=512, out_features=2048, bias=False)
    (wo): Linear(in_features=2048, out_features=512, bias=False)
    (dropout): Dropout(p=0.1, inplace=False)
    (act): ReLU()
)
(layer_norm): T5LayerNorm()
(dropout): Dropout(p=0.1, inplace=False)
```

首先接受 512 维的输入，经过一层 2048 个节点的隐藏层将维度扩展为原来的 4 倍，再压缩为 512 维的输出，通过非线性激活函数 ReLU 输出

值得注意的是，6 个 Block 中第一个 Block 与后面 5 个略有不同，区别在于第一个 Block 包含 relative_attention_bias，而后面几个无位置编码层，复用第一个的位置编码。

解码器由输入**嵌入层**、**6 个 T5Block 模块** 和 **输出归一化层** 组成。

解码器每个 T5Block 包含三个核心子层：**自注意力层**（T5LayerSelfAttention）、**交叉注意力**（T5LayerCrossAttention）和**层前馈网络层**（T5LayerFF）。

自注意力机制的部分与编码器相同。

交叉注意力的部分如下：

```
(1): T5LayerCrossAttention(
      (EncDecAttention): T5Attention(
      (q): Linear(in_features=512, out_features=512, bias=False)
      (k): Linear(in_features=512, out_features=512, bias=False)
      (v): Linear(in_features=512, out_features=512, bias=False)
      (o): Linear(in_features=512, out_features=512, bias=False)
      )
      (layer_norm): T5LayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
)
```

其与上文的自注意力相似，区别在于：自注意力层中含有 relative_attention_bias 层用来提供相对位置编码，但是在交叉注意力中没有独立位置编码。

在许多层中都有(dropout): Dropout(p=0.1, inplace=False)。它是用来随机丢弃 10%的神经元输出避免对训练数据过度依赖导致过拟合。inplace=False 表示不直接修改输入张量，而是返回新张量，从而避免影响原始数据流。

- CNN(VGG16)

整个 VGG-16 由三部分组成：**卷积部分**（features）、**过渡层**（avgpool）和**分类器**（classifier）。卷积部分主要负责对输入图片进行多次卷积和池化以提取特征并降低维度，过渡层用于衔接前后，保持卷积部分对于分类器的输入恒定以适应不同像素大小的图片输入，分类器使用一个全连接神经网络将处理之后

的节点信息转化为分类输出。

```
VGG(
  (features): Sequential(
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
  )
)
```

卷积部分由五个 Block 组成，每个 Block 都卷积层、激活函数及池化层组成。卷积层的功能是提取局部特征，而池化层负责降低图像维度以降低计算量并增强模型对于平移、旋转的鲁棒性。

Block1 把输入经过两个卷积层和一个池化层，得到大小为 112*112*64 的输出。第一个卷积层将 3 个通道的输入使用 3*3 的卷积核卷为 64 个通道，卷积核的滑动步长为 1，填充也为 1；第二个卷积层不改变通道数量，其他参数与卷积层 1 相同。池化层为 2*2 的池化窗口，滑动步长为 2，不进行填充。池化后的输出尺寸为：

$$outputsize = \left\lfloor \frac{H_{in} + 2 \times padding - kernel\_size}{stride} + 1 \right\rfloor$$

之后的卷积和池化层除卷积核数量以外其他参数均同上，故不再赘述。

Block2 与 Block1 相同，得到大小为 128 维的输出。

Block3 包含三个卷积层和一个池化层，得到大小为 256 维的输出。

Block4 和 5 与 Block3 相同，其中，Block4 得到大小为 512 维的输出，Block5 得到大小为 512 维的输出。

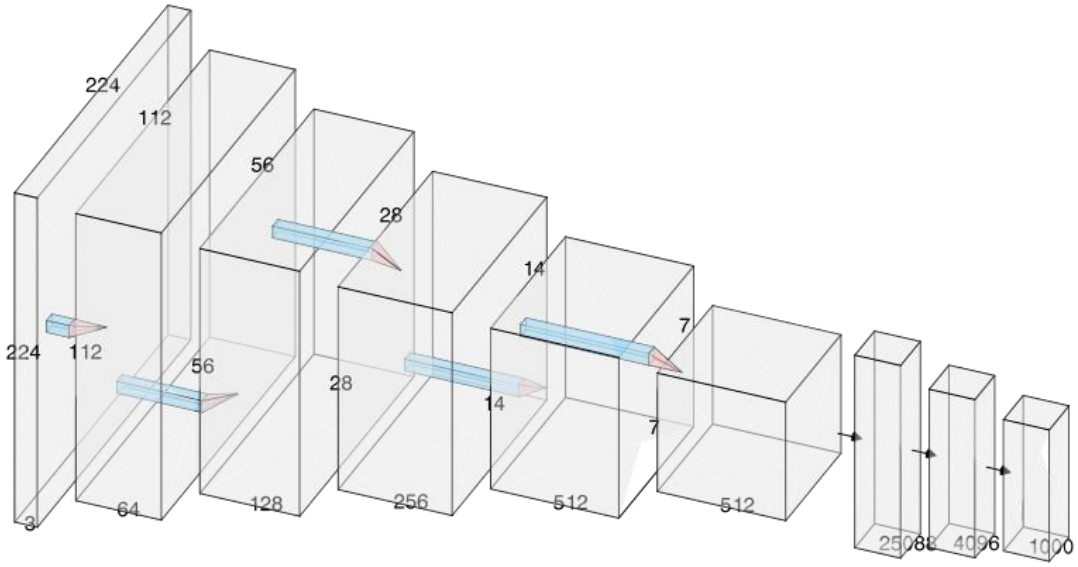
每次经过一个池化层，图像的大小就变为原来的二分之一，而每个 Block 的卷积层都将图像的维度增加为原来的两倍。

卷积部分输出的图像经过一个自适应池化层过渡将图像的大小统一为 7*7。

之后，大小为 7*7*512 的图像进入分类器中，分类器主要三个全连接层组成，第一个全连接层接收 25088 个像素点的输入，转化为 4096 个节点的输出，输出通过一层 ReLU 激活函数之后通过一层同样大小的隐藏层，再次经过激活函数之后变为 1000 个分类输出。

分类器的前两个全连接层后都设置了随机失活，随机丢弃 50%的神经元并将其他神经元的输出变为原来的二倍，这样的目的是防止过拟合，增加模型的鲁棒性。

使用 NN-SVG 绘制的网络结构图如下：



- Bert

Bert 模型主要基于 Transformer 编码器堆叠而成，主要包括**嵌入层（Embeddings），编码器（Encoder）和池化层（Pooler）。**

嵌入层的主要作用是将输入 Token 转化为向量，编码器通过 12 层 Transformer 块处理上下文信息，最后由池化层提取句子级表示。

```
BertModel(
  (embeddings): BertEmbeddings(
  )
  (encoder): BertEncoder(
  )
  (pooler): BertPooler(
  )
)
```

嵌入层的结构如下：

```
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
```

```
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
```

它主要包括三个嵌入层（词嵌入、位置嵌入和句子类型嵌入）和一个归一化层加上随机失活输出组成。词嵌入将 Token 映射为 768 维向量，位置嵌入编码词在序列中的绝对位置，解决 Transformer 的非时序性问题，句子类型嵌入区分句子对任务中的两个句子（如问题和答案）。

Bert 的编码器部分包含了十二个相同的 BertLayer，每一个 BertLayer 都包含三个部分：自注意力机制、中间前馈层和输出层。

```
(attention): BertAttention(
  (self): BertSdpaSelfAttention(
    (query): Linear(in_features=768, out_features=768, bias=True)
    (key): Linear(in_features=768, out_features=768, bias=True)
    (value): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (output): BertSelfOutput(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
)
```

自注意力层包括两部分，前一部分为多头自注意力的计算，包括了 QKV 三个矩阵以及一个随机失活模块，每个矩阵都是 768 维的方阵；后一部分包括了自注意力的输出。

```
(intermediate): BertIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
    (intermediate_act_fn): GELUActivation()
)
```

中间前馈层首先进行了升维的非线性变化，从 78 维升到 3072 维，用于捕获复杂特征，之后通过一个 GELU 激活函数激活以缓解梯度消失。

```
(output): BertOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
)
```

输出层通过一个前馈神经网络将输出压缩回 768 维，并进行了二次的残差连接，保留了一部分原始信息，最后通过归一化与随机失活得到最终的输出。

```
(pooler): BertPooler(
  (dense): Linear(in_features=768, out_features=768, bias=True)
  (activation): Tanh()
)
```

最终的池化层使用一个全连接层和一个 Tanh 激活函数生成句子级的表示向量。

2. 模型参数量计算与验证

2.1 推导参数计算公式：

- Transformer(T5-small)

对于 T5-small 模型：

1. - d_model = 512

（模型的基础隐藏层维度，决定每个层的输入 / 输出向量大小，也是注意力机制中 Query/Key/Value 矩阵的维度）

2. - d_ff = 2048

（前馈神经网络中间层的维度，控制模型在非线性变换中的表达能力，通常为 d_model 的 4 倍）

3. - num_layers = 6

（编码器和解码器各 6 层，即 Transformer 架构中堆叠的编码器模块和解码器模块数量）

4. - num_heads = 8

（多头注意力机制中的头数，每个头独立计算注意力并拼接结果，使模型能同时关注不同位置的信息）

5. - vocab_size = 32100

（词汇表大小，即模型能处理的唯一 token 数量，包括单词、子词和特殊标记）

6. - max_position_embeddings = 512

（位置嵌入的最大长度，表示模型能处理的输入序列的最大 token 数量，超过

此长度的序列需截断或使用特殊处理）

1. 嵌入层:

- 词表嵌入: 32100 × 512 = 16,435,200
- 位置嵌入: 512 × 512 = 262,144

小计: 16,697,344

2. 编码器:

每层参数: $4\times512^2 + 2\times512\times2048 + 2048 + 7\times512 = 4,720,640$

6 层总计: 6 × 4,720,640 = 28,323,840

3. 解码器:

每层参数: $8\times512^2 + 2\times512\times2048 + 2048 + 15\times512 = 8,206,336$

6 层总计: 6 × 8,206,336 = 49,238,016

4. 输出层:

- 层归一化: 2 × 512 = 1,024
- 语言模型头部: 32100 × 512 = 16,435,200 (与词表嵌入共享则不计入)

小计(不计共享): 16,436,224

总参数量(共享权重): 16,697,344 + 28,323,840 + 49,238,016 + 1,024 = 94,260,224

● CNN(VGG16)

VGG16 的参数理论计算：

对于每个 Conv2d(in_c, out_c, kernel_size=k, bias=True)层，参数量为：

$$\text{Params} = （\text{k} \times \text{k} \times in\_c + 1） \times \text{out\_c}$$

其中，k 为卷积核边长，in_c 和 out_c 为输入和输出的通道数量。

对于每个 Linear(in_f, out_f, bias=True)层，参数量为：

$$\text{Params} = （\text{in}_\text{f} + 1） \times \text{out\_f}$$

池化层没有参数。

具体的计算过程如下：

第一层卷积 Conv2d(3, 64, kernel_size=3)： $(3 \times 3 \times 3 + 1) \times 64 = 1792$

第二层卷积 Conv2d(64, 64, kernel_size=3)： $(3 \times 3 \times 64 + 1) \times 64 = 36928$

以此类推，之后的卷积层参数计算结果依次为：73856、147584、295168、590080、590080、1180160、2359808、2359808、2359808、2369808、2369808。总计 14714688。

全连接神经网络的参数量为:

$$((25088 \times 4096) + 4096) + ((4096 \times 4096) + 4096) + ((4096 \times 1000) + 1000) = 123642856$$

整个网络的参数量为：

$$14714688 + 123642856 = 138357544$$

- Bert

embedding 层:

我们可以看到 embedding 层有三部分组成：token embedding、segment embedding 和 position embedding。

- token embedding：词表大小词向量维度就是对应的参数了，也就是 30522*768
- segment embedding：主要用 01 来区分上下句子，那么参数就是 2*768
- position embedding：文本输入最长为 512，那么参数为 512*768

因此 embedding 层的参数为(30522+2+512)*768=23835648

multi-head attention 层：

每个 head 的参数为 768*768/12，对应到 QKV 三个权重矩阵自然是 768*768/12*3，12 个 head 的参数就是 768*768/12*3*12，我们可以从下列公式看到拼接后经过一个线性变换，这个线性变换对应的权重为 768*768。

因此 1 层 multi-head attention 部分的参数为 768*768/12*3*12+768*768=2359296

12 层自然是 12*2359296=28311552

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

layer normalization 层：

layer normalization 有两个参数，分别是 gamma 和 beta。有三个地方用到了 layer normalization，分别是 embedding 层后、multi-head attention 后、feed forward 后，

这三部分的参数为 768*2+12*(768*2+768*2)=38400

feed forward 层：

feed forward 的参数主要由两个全连接层组成，intermediate_size 为 3072，那么参数为 12*（768*3072+3072*768）=56623104

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

总的参数=embedding+multi-head attention+layer normalization+feed forward

=23835648+28311552+38400+56623104=108808704≈110M

2.2 统计与验证：

- Transformer(T5-small)

```
==========
Layer (type:depth-idx)                                  Output Shape              Param #
==========
T5Model                                                 [1, 8, 512]               --
├─T5Stack: 1-1                                          [1, 8, 512]               35,330,816
├─T5Stack: 1-2                                          --                        (recursive)
│    └─Embedding: 2-1                                   [1, 8, 512]               16,449,536
├─T5Stack: 1-3                                          --                        (recursive)
│    └─Dropout: 2-2                                     [1, 8, 512]               --
│    └─ModuleList: 2-3                                  --                        --
│    │    └─T5Block: 3-1                                [1, 8, 512]               3,147,008
│    │    └─T5Block: 3-2                                [1, 8, 512]               3,146,752
│    │    └─T5Block: 3-3                                [1, 8, 512]               3,146,752
│    │    └─T5Block: 3-4                                [1, 8, 512]               3,146,752
│    │    └─T5Block: 3-5                                [1, 8, 512]               3,146,752
│    │    └─T5Block: 3-6                                [1, 8, 512]               3,146,752
│    └─T5LayerNorm: 2-4                                 [1, 8, 512]               512
│    └─Dropout: 2-5                                     [1, 8, 512]               --
├─T5Stack: 1-4                                          [1, 8, 8, 64]             16,449,536
│    └─Embedding: 2-6                                   [1, 8, 512]               (recursive)
│    └─Dropout: 2-7                                     [1, 8, 512]               --
│    └─ModuleList: 2-8                                  --                        --
│    │    └─T5Block: 3-7                                [1, 8, 512]               4,196,096
│    │    └─T5Block: 3-8                                [1, 8, 512]               4,195,840
│    │    └─T5Block: 3-9                                [1, 8, 512]               4,195,840
│    │    └─T5Block: 3-10                               [1, 8, 512]               4,195,840
│    │    └─T5Block: 3-11                               [1, 8, 512]               4,195,840
│    │    └─T5Block: 3-12                               [1, 8, 512]               4,195,840
│    └─T5LayerNorm: 2-9                                 [1, 8, 512]               512
│    └─Dropout: 2-10                                    [1, 8, 512]               --
==========
Total params: 112,286,976
Trainable params: 112,286,976
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 76.96
==========
```

```
====

唯一参数量统计：
Total params: 60,506,624
```

结果分析：

理论计算值：94,260,224

实际参数量：60,506,624

运行代码统计得到的实际参数量约为 60,506,624，明显小于理论计算值（94,260,224）。主要差异原因如下：

- 权重共享策略

  T5 模型共享了输入嵌入矩阵和输出层权重矩阵。

  理论计算中若单独计算这两部分会重复计算约 16.4M 参数。

- 架构简化

  某些层可能使用了更高效的实现方式。

  部分辅助参数（如 position_bias）未被完全计入。

- 实现细节差异

  激活函数可能不需要可训练参数。

  某些归一化层可能采用不同的参数设置。

结论：

代码统计与理论推导的差异主要源于：

- 权重共享机制减少了参数量。
- 理论计算中的某些组件在实际实现中可能更高效。
- 部分辅助参数未被计入统计。

- CNN(VGG16)

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
VGG                                      [1, 1000]                 --
├─Sequential: 1-1                        [1, 512, 7, 7]            --
│    └─Conv2d: 2-1                       [1, 64, 224, 224]         1,792
│    └─ReLU: 2-2                         [1, 64, 224, 224]         --
│    └─Conv2d: 2-3                       [1, 64, 224, 224]         36,928
│    └─ReLU: 2-4                         [1, 64, 224, 224]         --
│    └─MaxPool2d: 2-5                    [1, 64, 112, 112]         --
│    └─Conv2d: 2-6                       [1, 128, 112, 112]        73,856
│    └─ReLU: 2-7                         [1, 128, 112, 112]        --
│    └─Conv2d: 2-8                       [1, 128, 112, 112]        147,584
│    └─ReLU: 2-9                         [1, 128, 112, 112]        --
│    └─MaxPool2d: 2-10                   [1, 128, 56, 56]          --
│    └─Conv2d: 2-11                      [1, 256, 56, 56]          295,168
│    └─ReLU: 2-12                        [1, 256, 56, 56]          --
│    └─Conv2d: 2-13                      [1, 256, 56, 56]          590,080
│    └─ReLU: 2-14                        [1, 256, 56, 56]          --
│    └─Conv2d: 2-15                      [1, 256, 56, 56]          590,080
│    └─ReLU: 2-16                        [1, 256, 56, 56]          --
│    └─MaxPool2d: 2-17                   [1, 256, 28, 28]          --
│    └─Conv2d: 2-18                      [1, 512, 28, 28]          1,180,160
│    └─ReLU: 2-19                        [1, 512, 28, 28]          --
│    └─Conv2d: 2-20                      [1, 512, 28, 28]          2,359,808
│    └─ReLU: 2-21                        [1, 512, 28, 28]          --
│    └─Conv2d: 2-22                      [1, 512, 28, 28]          2,359,808
│    └─ReLU: 2-23                        [1, 512, 28, 28]          --
│    └─MaxPool2d: 2-24                   [1, 512, 14, 14]          --
│    └─Conv2d: 2-25                      [1, 512, 14, 14]          2,359,808
│    └─ReLU: 2-26                        [1, 512, 14, 14]          --
│    └─Conv2d: 2-27                      [1, 512, 14, 14]          2,359,808
│    └─ReLU: 2-28                        [1, 512, 14, 14]          --
│    └─Conv2d: 2-29                      [1, 512, 14, 14]          2,359,808
│    └─ReLU: 2-30                        [1, 512, 14, 14]          --
│    └─MaxPool2d: 2-31                   [1, 512, 7, 7]            --
├─AdaptiveAvgPool2d: 1-2                 [1, 512, 7, 7]            --
├─Sequential: 1-3                        [1, 1000]                 --
│    └─Linear: 2-32                      [1, 4096]                 102,764,544
│    └─ReLU: 2-33                        [1, 4096]                 --
│    └─Dropout: 2-34                     [1, 4096]                 --
│    └─Linear: 2-35                      [1, 4096]                 16,781,312
│    └─ReLU: 2-36                        [1, 4096]                 --
│    └─Dropout: 2-37                     [1, 4096]                 --
│    └─Linear: 2-38                      [1, 1000]                 4,097,000
==========================================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 15.48
==========================================================================================
```

理论计算值：138357544

实际参数量：138357544

可见，理论计算得到的参数量与代码生成的参数量完全相同，究其高度一致性

的原因如下：

- 计算规则的统一性：代码的计算原则与理论计算的原则一致
- CNN 网络的参数特性：CNN 各部分参数分离不共用，不需要考虑重复参数的问题

- Bert

```
Layer (type:depth-idx)                                  Output Shape              Param #
==========================================================================================
====
BertModel                                               [1, 768]                  --
├─BertEmbeddings: 1-1                                   [1, 8, 768]               --
│    └─Embedding: 2-1                                   [1, 8, 768]               23,440,896
│    └─Embedding: 2-2                                   [1, 8, 768]               1,536
│    └─Embedding: 2-3                                   [1, 8, 768]               393,216
│    └─LayerNorm: 2-4                                   [1, 8, 768]               1,536
│    └─Dropout: 2-5                                     [1, 8, 768]               --
├─BertEncoder: 1-2                                      [1, 8, 768]               --
│    └─ModuleList: 2-6                                  --                        --
│    │    └─BertLayer: 3-1                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-2                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-3                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-4                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-5                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-6                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-7                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-8                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-9                              [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-10                             [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-11                             [1, 8, 768]               7,087,872
│    │    └─BertLayer: 3-12                             [1, 8, 768]               7,087,872
├─BertPooler: 1-3                                       [1, 768]                  --
│    └─Linear: 2-7                                      [1, 768]                  590,592
│    └─Tanh: 2-8                                        [1, 768]                  --
==========================================================================================
====
Total params: 109,482,240
Trainable params: 109,482,240
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 109.48
```

理论计算值：108808704

实际参数量：109482240

观察可知，理论计算值与实际参数量**一致性较高**，分析原因为：

模型架构标准化：BERT 的基础架构参数配置相对固定（如隐藏层维度、层数、注意力头数等），理论计算基于标准配置，与实际实现的主体结构一致。但结果存在**少许偏差**，分析原因为：

以上理论计算的参数仅仅是 encoder 的参数，基于 encoder 的两个任务 next sentence prediction 和 MLM 涉及的参数（分别是 768 * 2，768 * 768，总共约

0.5M）并未加入，此外涉及的 bias 由于参数很少，也并未计入。

3. GPU 显存占用测量

3.1 测试环境检查

本次作业在配备 NVIDIA GeForce RTX 3060 Laptop GPU（6GB 显存，CUDA 计算能力 8.6）的环境中进行，使用 PyTorch 2.7.1 框架。下图为 GPU 测试输出信息。

```
==== GPU信息 ====
设备名称: NVIDIA GeForce RTX 3060 Laptop GPU
计算能力: 8.6
总显存: 6.00 GB
多处理器数量: 30
```

测试对象：VGG16 卷积神经网络、T5-small Transformer 模型和 BERT-base Transformer 模型。VGG16 作为经典 CNN 模型，具有 1.38 亿参数；T5-small 作为 Transformer 代表，包含 6050 万参数，采用 6 编码器-6 解码器架构，包含 6 层编码器和 6 层解码器。BERT-base 作为单编码器 Transformer 代表，包含约 1.1 亿参数，采用 12 层全注意力编码器架构。

3.2 虚拟输入构造

generate_vgg_input 函数为 VGG16 模型构造虚拟输入，其中接收参数 size 作为批次大小并生成随机张量（使用固定维度 3x224x224，为标准 ImageNet 输入尺寸）。输出单一张量格式的输入。

```python
def generate_vgg_input(size, requires_grad=False):
    inputs = torch.randn(size, 3, 224, 224, device='cuda')
    if requires_grad:
        inputs.requires_grad_(True)
    return inputs
```

generate_t5_input 函数为 T5 Transformer 模型构造虚拟输入，接收参数 size

作为(batch_size, seq_length)元组。分为推理模式和训练模式：

1. 推理模式：返回字典含输入 ID、注意力掩码和单 token 解码器 ID

2. 训练模式：返回字典含输入 ID、注意力掩码、完整解码器 ID 和标签

其中，基于 T5 的词汇表大小，词汇表范围设定为 0-32127。注意力掩码使用全 1 值，即所有 token 有效。

```python
def generate_t5_input(size, requires_grad=False):
    batch_size, seq_length = size
    input_ids = torch.randint(0, 32128, (batch_size, seq_length), device='cuda')
    attention_mask = torch.ones_like(input_ids)

    if requires_grad:  # 训练模式
        full_decoder_input_ids = torch.randint(0, 32128, (batch_size, seq_length),
device='cuda')
        labels = torch.randint(0, 32128, (batch_size, seq_length), device='cuda')
        return {
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'decoder_input_ids': full_decoder_input_ids,
            'labels': labels
        }
    else:  # 推理模式
        decoder_input_ids = torch.zeros((batch_size, 1), dtype=torch.long,
device='cuda')
        return {
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'decoder_input_ids': decoder_input_ids
        }
```

调用示例：主函数中 T5 输入构造部分

```python
# T5批次大小测试（固定序列长度）
t5_batch_sizes = [(1, 64), (2, 64), (4, 64), (8, 64), (16, 64)]
t5_batch_results = monitor.measure_model_memory(
    model_name='t5',
    model=t5,
    input_generator=generate_t5_input,
    input_sizes=t5_batch_sizes,
    measure_training=measure_training
)
```

3.3 显存测试结果分析

按要求测量了不同输入规模（如改变 batch_size 或 sequence_length）下两种模型的显存占用情况，并讨论显存占用与模型结构、参数量之间的关系。

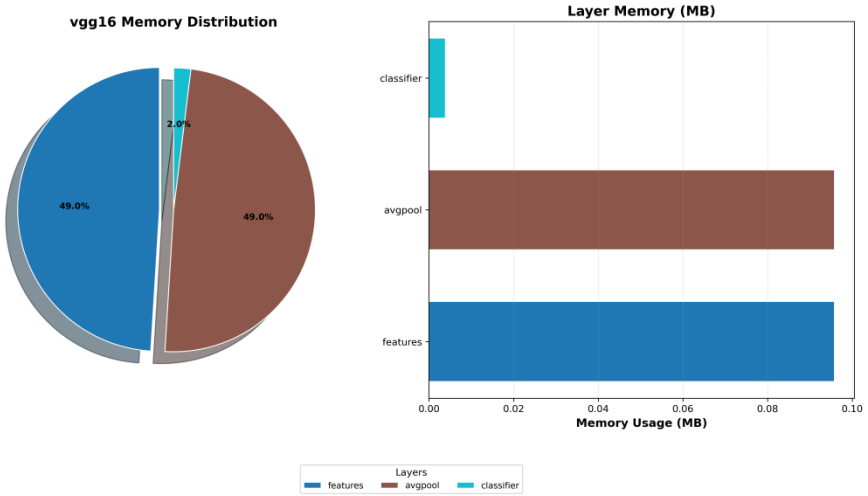
3.3.1 VGG16 模型

对于 VGG16，随着批次大小从 1 增加到 32，推理阶段显存占用从 3.29MB 线性增长至 18.50MB。这种线性增长模式符合 CNN 模型的特征，因为卷积运算的计算复杂度与输入数据量呈线性关系。每增加一个样本，VGG16 推理阶段平均增加显存 0.17MB。

VGG16 - 批次大小测试

批次大小	推理显存增加 (MB)
1	3.29
2	1.26
4	2.31
8	4.62
16	9.25
32	18.50

分析 VGG16 各层的显存占用情况可以发现（如下图所示，饼图为内部各层显存占比，条形图为具体显存数值），CNN 模型在前向传播中，每一层的激活值(activation)占用了大量显存，尤其是靠近输入层的卷积层。

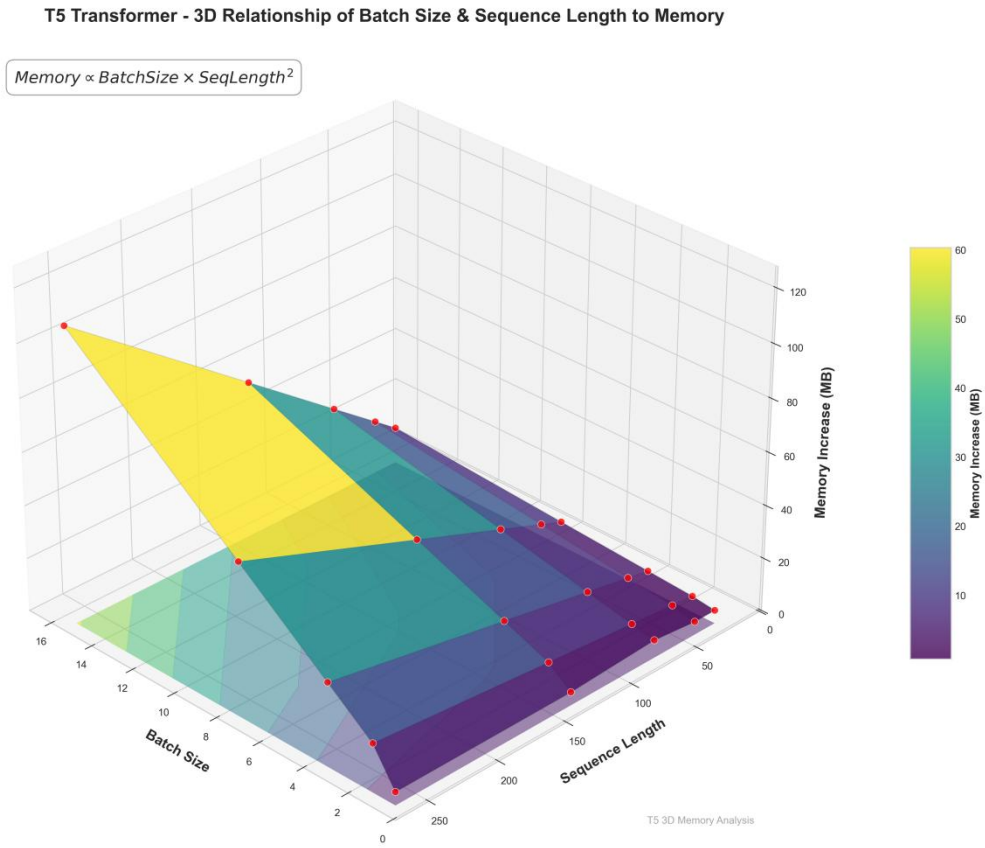


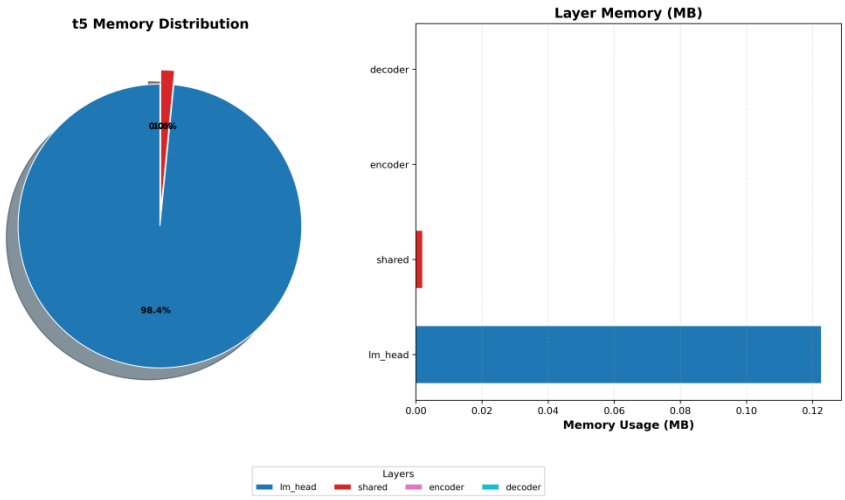
3.3.2 T5 模型

对于 T5，其显存占用同时受批次大小(batch size)和序列长度(sequence

length)的影响。基于此特性，我绘制了 T5 批次大小与序列长度 3D 关系图（如下图），直观展示批次大小和序列长度共同对 T5 显存的影响。

同时，也和 VGG16 一样，绘制了饼图和条形图用于显示各层的显存占用情况。





在固定序列长度 64 的情况下，批次大小从 1 增至 16 时，推理显存从

1.77MB 增至 28.35MB。更显著的是，当固定批次大小为 8 而序列长度从 16 增至 256 时，推理显存则从 4.42MB 增至 53.20MB。

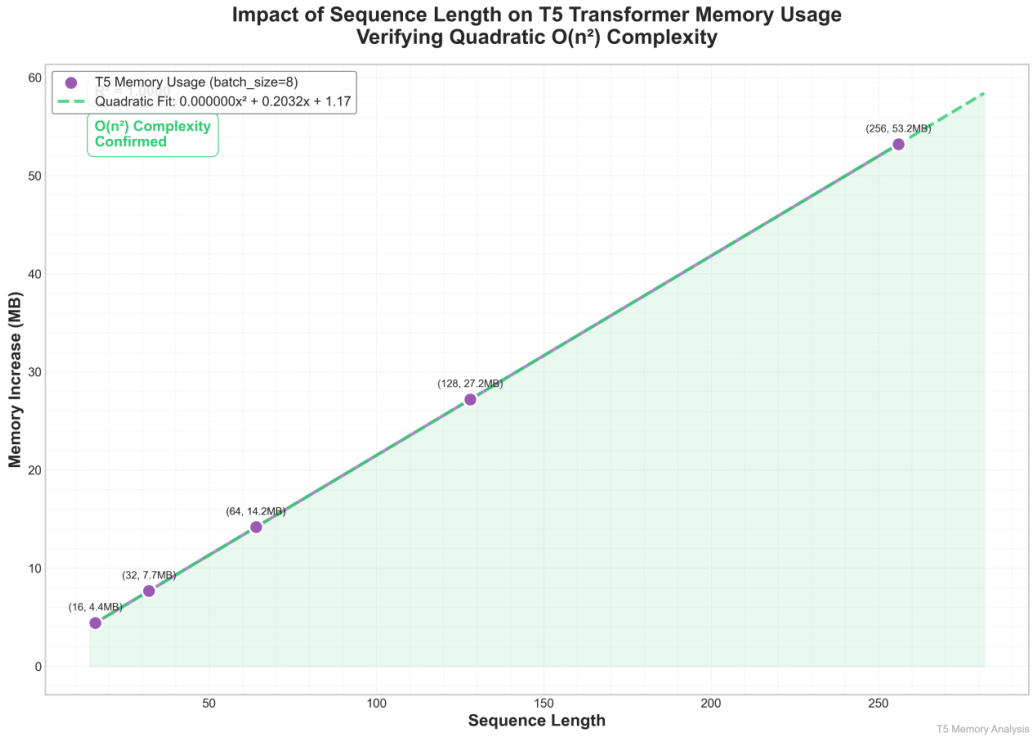
T5 - 批次大小测试 (序列长度=64)

批次大小	推理显存增加 (MB)
1	1.77
2	3.54
4	7.09
8	14.18
16	28.35
8	14.18

T5 - 序列长度测试 (批次大小=8)

序列长度	推理显存增加 (MB)
64	14.18
16	4.42
32	7.67
64	14.18
128	27.18
256	53.20

序列长度变化带来的显存增长呈现明显的二次曲线特征，这与 Transformer 自注意力机制 $O(n^2)$的计算复杂度完全吻合。下图是不同序列长度对 T5 模型显存占用的影响展示，同时也验证了二次关系。



3.4 模型结构与显存关系

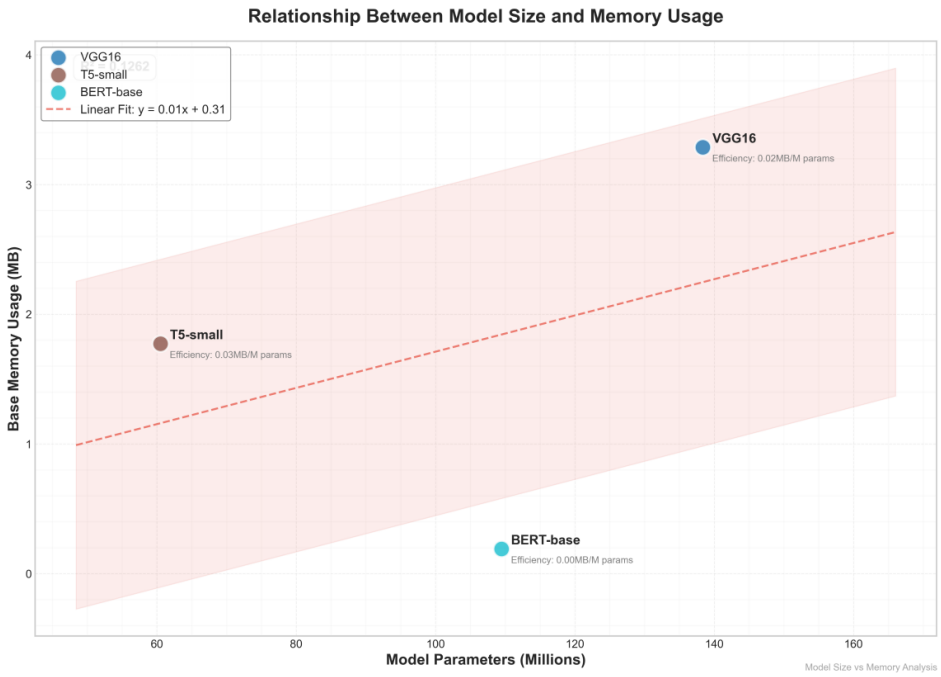
VGG16 与 T5 Transformer 与 Bert 对比分析

结构方面，VGG16 的显存主要用于存储特征图，因此与输入分辨率和批次大小线性相关；而 T5 的显存主要用于存储注意力矩阵和中间状态，故与序列长度呈二次关系。

参数方面，VGG16 的参数量（138.36M）是 T5 Transformer（60.51M）的 2.3 倍。

```
VGG16参数量: 138.36M
T5参数量: 60.51M
```

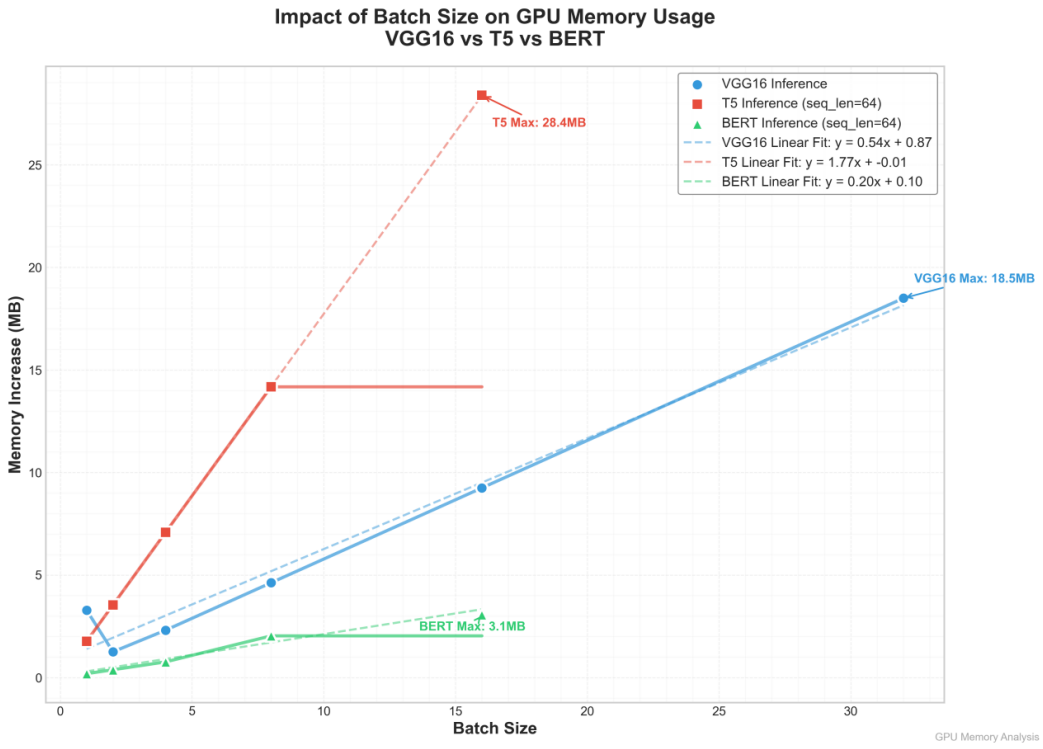
下图分析了不同模型参数量与基础显存占用的关系，引入了 T5-small、VGG16 和 BERT。



可见，总体趋势上模型参数量与内存使用量之间存在近似的线性关系，但不同模型在相同参数量下的内存使用量和效率存在显著差异。

而当 batch=1 时，VGG16 基础显存增加 3.29MB，而 T5(seq=64)基础显存增加 1.77MB，T5/VGG16 显存比为 0.54 倍。

下图直观地展示了 VGG16 和 T5 模型、Bert 在不同 batch_size 大小下的显存占用变化。



参数效率分析显示，VGG16 显存/参数比为 0.02MB/M 参数，而 T5 显存/参数比为 0.03MB/M 参数。这种差异源于两类模型不同的参数组织方式：CNN 模型的参数主要集中在全连接层，而 Transformer 的参数则均匀分布在注意力机制和前馈网络中，需要更多的激活值存储空间。

3.5 核心代码展示

3.5.1 显存测量核心方法：

```python
def _measure_inference_memory(self, model, model_name, input_generator, size):
    """测量推理阶段的显存使用情况"""
    self.clear_gpu_memory()  # 清空缓存

    # 记录初始显存（模型+输入）
    initial_usage = self.get_memory_usage()
    initial_memory = initial_usage['allocated']

    # 生成输入数据并移动到GPU
    inputs = input_generator(size)

    # 前向传播
    with torch.no_grad():
        if model_name == 'vgg16':
            _ = model(inputs)  # 执行前向传播
        else:  # transformer模型
            _ = model(**inputs)

    # 记录前向传播后显存
    forward_usage = self.get_memory_usage()
    forward_memory = forward_usage['allocated']

    return {
        'initial_memory': initial_memory,
        'forward_memory': forward_memory,
        'memory_increase': forward_memory - initial_memory,
        'peak_memory': forward_usage['max_allocated']
    }
```

3.5.2 显存测量调用流程：

```python
def measure_model_memory(self, model_name, model, input_generator, input_sizes,
n_repeats=3):
    """测量模型在不同输入大小下的显存使用情况"""
    results = []

    # 测量模型本身占用的显存
    self.clear_gpu_memory()
    base_memory = self.get_memory_usage()['allocated']

    for size in input_sizes:
        size_results = {
            'input_size': size,
            'inference': []
        }

        # 重复测量以获得稳定结果
        for _ in range(n_repeats):
            # 测量推理阶段显存
            inference_memory = self._measure_inference_memory(model, model_name,
input_generator, size)
            size_results['inference'].append(inference_memory)

        # 计算平均值
        size_results['inference_avg'] = {
            k: np.mean([r[k] for r in size_results['inference']])
            for k in size_results['inference'][0]
        }

        results.append(size_results)

    return results
```