



北京大学

人工智慧期末作業

題目： The Annotated
Transformer

姓 名： 干皓丞

学 号： 2101212850

院 系： 信息工程学院

专 业： 计算机应用技术

研究方向： 通信及信息安全技术

导 师： 陳杰 教授

二〇二一 年 十二 月

摘要

如作业目标所示，本作业先进行该作业目标的说明，第二章为该篇论文的心得与 Attention Is All You Need 的文章精读翻译，第三章对 NLP 与 Harvard 等技术文章与进行实作与进行总结，其包含目前网路上技术文章进行总结，第四章则是说明 Transformer 模型的理解与相关工作说明，第五章则是对本期作业的总结。

关键词：Transformer

目录

第一章 作业目标	1
1.1 作业说明	1
1.2 论文作业目标	1
1.3 The Annotated Transformer	1
第二章 论文精读	3
2.1 论文精读资讯	3
2.2 论文心得	3
2.2.1 前言	3
2.2.2 方法	4
2.2.3 结果	4
2.2.4 讨论	5
2.3 论文翻译	5
2.3.1 摘要	5
2.3.2 前言	6
2.3.3 背景	6
2.3.4 模型架构	7
2.3.5 Why Self-Attention	11
2.3.6 训练	12
2.3.7 结果	13
2.3.8 结论	15
第三章 技术文章与讨论	17
3.1 技术文章	17
3.1.1 编程	17
3.1.2 文献说明与综述整理	17
3.1.3 技术文件与实现	18
3.2 程式码与注解说明	18
3.2.1 Harvard NLP	18
3.2.2 Chatbot Transformer	49

3.3 使用技术	49
3.3.1 LaTeX	49
3.3.2 Docker	49
第四章 Transformer.....	53
4.1 Transformer 理解	53
4.1.1 原理简述	53
4.2 Transformer 研究	54
4.2.1 工作追溯	54
4.2.2 Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks.....	57
4.2.3 Deep amortized clustering	59
4.2.4 Non-local Neural Networks	60
4.2.5 Axial-DeepLab: Stand-Alone Axial-Attention for Panoptic Segmentation	62
4.2.6 Taming Transformers for High-Resolution Image Synthesis.....	64
4.2.7 Video Action Transformer Network	65
4.3 Transformer 工作	66
4.3.1 摘要与研究贡献	67
4.3.2 程式码与演示	69
第五章 总结.....	77
5.1 课程回馈	77
参考文献	79

主要符号对照表

x, y, m, n, t	标量, 通常为变量
K, L, D, M, N, T	标量, 通常为超参数
$x \in \mathbb{R}^D$	D 维列向量
(x_1, \dots, x_D)	D 维行向量
$(x_1, \dots, x_D)^T$ or $(x_1; \dots; x_D)^T$	D 维行向量
$x \in \mathbb{R}^{KD}$	(KD) 维的向量
\mathbb{M}_i or $\mathbb{M}_i(x)$	第 i 列为 $\mathbf{1}$ (或者 x), 其余为 $\mathbf{0}$ 的矩阵
$diag(\mathbf{x})$	对角矩阵, 其对角元素为 x
I_N or I	($N \times N$) 的单位阵
$A \in \mathbb{R}^{D_1 \times D_2 \times \dots \times D_K}$	大小为 $D_1 \times D_2 \times \dots \times D_K$ 的张量
$\{x^{(n)}\}_{n=1}^N$	集合
$\{(x^{(n)}, y^{(n)})\}_{n=1}^N$	数据集
$\mathcal{N}(x; \mu, \Sigma)$	变量 x 服从均值为 μ , 方差为 Σ 的高斯分布

① 本符号对照表内容选自邱锡鹏老师的《神经网络与深度学习》^[1]一书。

第一章 作业目标

1.1 作业说明

该作业为人工智慧期末报告，其专案为 kancheng/kan-cs-report-in-2021，程式码则可于 kan-cs-report-in-2021/AI/pytorch-transformer/code 中查询，而文件则可由专案中进行查询。

1.2 论文作业目标

Step 1 : Read the paperAttention is all you needand write a reading report.

Step 2: Read and run Transformer-related code (NLP Harvard).

Recording the operation result and your understanding in the experimental report.

Step 3: You can choose a specific area (No restriction on direction), and apply the transformer to this area.

It is necessary to find a published paper and successfully reproduce the corresponding result in the paper.Recording the related paper, experimental steps, and your results vs results in the paper to the experimental report.

Step 4:

The reading report & source code & the related paper & experimental report will be organized and submitted.

1.3 The Annotated Transformer

Q1: NO GPU available

- A1: a survey about transformer
- Example: <https://arxiv.org/pdf/1809.02165.pdf>

Q2: code plagiarism

- In particular, all code and documentation should be entirely your own work. You may consult with other students about high-level design strategies related to programming assignments, but you may not copy code or use the structure or organization of another student's program.
- If you use any code or functions found from the internet, please tell us the reference link and how do you use it. Direct code copy from the internet would be considered

violation of this policy.

- If we find there are two returned assignments same in large proportional code, both of the assignments would be considered violation of this policy.

第二章 论文精读

此章为 Transformer 模型的 Attention Is All You Need^[2] 与当中阅读论文与心得：

2.1 论文精读资讯

下列论文资讯在 arXiv 为 1706.03762。

Attention Is All You Need
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones,
Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin
<https://arxiv.org/abs/1706.03762>
Comments: 15 pages, 5 figures
Subjects: Computation and Language (cs.CL); Machine Learning (cs.LG)
Cite as: arXiv:1706.03762 [cs.CL] (or arXiv:1706.03762v5 [cs.CL] for this version)

2.2 论文心得

而论文心得必须根据 IMRAD 的方式进行撰写，其流程分别为四部分，分别为前言 (Introduction)、方法 (Methods)、结果 (Results) 和讨论 (Discussion)，最后则是根据原论文架构的范例，该心得则根据 IMRAD 来分析。

2.2.1 前言

该篇研究根据过往研究成果进行研究，将其发展成新的方法与架构，该方法在各个领域上具有显著的效果，研究者发现循环神经网络 (Recurrent neural networks)，特别是 Sepp Hochreiter et al. 的长短期记忆 (long short-term memory) 和 Junyoung Chung et al. 的门控循环 (gated recurrent) 神经网络，已成为序列建模和转导问题，并且是语言建模和机器翻译中的最先进方法。

此后许多研究者们的努力如 Yonghui Wu et al.、Minh-Thang Luong et al.、Rafal Jozefowicz et al. 继续推动循环语言模型和编码器-解码器架构的界限，而循环模型 (Recurrent models) 通常沿输入和输出序列的符号位置因子计算。将位置与计算时间中的步

骤对齐，它们生成一系列隐藏状态 ht ，作为先前隐藏状态 $ht1$ 和位置 t 的输入的函数。此固有的顺序性质排除了训练示例中的并行化，这在更长的序列长度时变得至关重要，因为记忆体限制了跨示例的批处理，近来的工作通过分解技巧 Oleksii Kuchaiev et al. 和条件计算 Noam Shazeer et al. 显著提高了计算效率，同时在后者的情况下也提高了模型性能。然而顺序计算的基本约束仍然存在，而且注意机制已成为各种任务中引入注目的序列建模和转导模型的组成部分，允许对依赖项进行建模，而无需考虑它们在输入或输出序列中的距离 Dzmitry Bahdanau et al. 、 Yoon Kim et al.，然而除了少数情况如 Ankur Parikh et al.，在所有情况下该注意力机制都与循环网络结合使用。在这项工作中，我们提出了名为 Transformer 的一种避免重复的模型架构，而是完全依靠注意力机制来绘制输入和输出之间的全局依赖关系，而 Transformer 允许显著更多的并行化，并且在八个 P100 GPU 上进行了短短 12 小时的训练后，可以在翻译质量方面达到新的水平。

2.2.2 方法

该研究的模型架构，其大多数具有竞争力的神经序列转导模型都具有编码器-解码器结构，在此研究者编码器将符号表示的输入序列 x 映射到连续表示的序列 z 。同时给定 z 解码器，然后生成一个符号的输出序列 y 后，输出一次一个元素。在每一步过程中，模型都是自回归的 Alex Graves et al.，且在生成下一个时，将先前生成的符号作为额外的输入使用。Transformer 遵循这一整体架构，使用堆叠的自注意力和逐点、完全连接的编码器和解码器层。

该章节分别分为编码器和解码器堆栈 (Encoder and Decoder Stacks) 部分，而注意力 (Attention) 的部分则包含缩放点积注意力 (Scaled Dot-Product Attention)、多头注意力 (Multi-Head Attention) 跟注意力在该研究模型中的应用，同时说明该研究的位置前馈网络 (Position-wise Feed-Forward Networks)、位置前馈网络 (Position-wise Feed-Forward Networks)、嵌入和 Softmax (Embeddings and Softmax) 与位置编码 (Positional Encoding)。

2.2.3 结果

在此研究者说明何谓自我注意 (Self-Attention) 与训练机制，研究者将自注意力层的各个方面与通常用于将一个可变长度的符号表示序列 x 映射到另一个等长序列 z ，例如典型序列转导编码器或解码器中的隐藏层。

过程中考虑了三个需求，首先是每层的总计算复杂度，另一个则是可以并行化的计算量，通过所需的最小顺序操作数来衡量，第三个是网络中远程依赖之间的路径长度，而且当中学习远程依赖是许多序列转导任务中的关键挑战，另外影响学习这种依赖性能力的一个关键因素是前向和后向信号必须在网络中穿越的路径长度，当中输入

和输出序列中任意位置组合之间的这些路径越短，而学习远程依赖关系就越容易 Sepp Hochreiter et al.。同时研究者还比较了由不同层类型组成的网络中任意两个输入和输出位置之间的最大路径长度，另外个人注意力不仅清楚地学习执行不同的任务，而且许多人似乎表现出与句子的句法和语义结构相关的行为。

而训练的部分则描述了模型的训练机制并对硬体和优化器 (Optimizer) 与正则化 (Regularization) 进行说明，同时在研究者的训练数据和批处理 (Training Data and Batch-ing)，研究者在包含约 450 万个句子对的标准 WMT 2014 英德数据集上进行了训练，同时对句子使用字节对编码进行编码，使它具有大约 37000 个标记的共享源目标词汇表。对于英法转换，研究者使用了明显更大的 WMT 2014 英法数据集，该数据集由 3600 万个句子组成，并将标记拆分为 32000 个单词词表，而句子对按近似序列长度分批在一起，此外每个训练批次包含一组句子对，其中包含大约 25000 个源标记和 25000 个目标标记。

2.2.4 讨论

根据整个研究的工作，该研究者们提出了 Transformer，这是第一个完全基于注意力的序列转换模型，使用多头自注意力取代了编码器-解码器架构中最常用的循环层。而面对翻译任务，当中 Transformer 的训练速度明显快于基于循环或卷积层的架构。另外在 WMT 2014 English-to-German 和 WMT 2014 English-to-French 翻译任务过程中，研究者达到了最先进的水平。

2.3 论文翻译

在此根据论文 Learning with Privileged Information via Adversarial Discriminative Modality Distillation 的章节结构进行翻译，其论文题目的中文字面意义上为通过对抗性判别模态蒸馏 (Adversarial Discriminative Modality Distillation) 学习特权资讯 (Privileged Information)。该根据原研究论文架构来分配章节，该篇研究的论文程式码可于 GitHub 专案取得 (<https://github.com/pmororio/admd>)。

2.3.1 摘要

主导序列转导模型基于复杂的循环或卷积神经网络，包括编码器和解码器，当中性能最好的模型还通过注意力机制连接编码器和解码器。我们提出了一种新的简单网络架构，即 Transformer，它完全基于注意力机制，完全消除了递归和卷积，在两个机器翻译任务上的实验表明，这些模型在质量上更胜一筹，同时更可并行化并且需要更少的训练时间。我们的模型在 WMT 2014 英德翻译任务上达到了 28.4 BLEU，比现有

的最佳结果（包括集成）提高了 2 BLEU，而在 WMT 2014 英语到法语翻译任务中，我们的模型在 8 个 GPU 上训练 3.5 天后建立了一个新的单模型最先进的 BLEU 分数 41.8，这是最好的训练成本的一小部分文献中的模型。最后我们表明，通过将 Transformer 成功应用于具有大量和有限训练数据的英语选区解析，可以很好地推广到其他任务。

2.3.2 前言

发现循环神经网络 (Recurrent neural networks)，特别是 Sepp Hochreiter et al. 的长短期记忆 (long short-term memory) 和 Junyoung Chung et al. 的门控循环 (gated recurrent) 神经网络，已成为序列建模和转导问题，并且是语言建模和机器翻译中的最先进方法。

此后许多研究者们的努力如 Yonghui Wu et al.、Minh-Thang Luong et al.、Rafal Jozefowicz et al. 继续推动循环语言模型和编码器-解码器架构的界限，而循环模型 (Recurrent models) 通常沿输入和输出序列的符号位置因子计算。将位置与计算时间中的步骤对齐，它们生成一系列隐藏状态 ht ，作为先前隐藏状态 $ht1$ 和位置 t 的输入的函数。此固有的顺序性质排除了训练示例中的并行化，这在更长的序列长度时变得至关重要，因为记忆体限制了跨示例的批处理，近来的工作通过分解技巧 Oleksii Kuchaiev et al. 和条件计算 Noam Shazeer et al. 显著提高了计算效率，同时在后者的情况下也提高了模型性能。然而顺序计算的基本约束仍然存在，而且注意机制已成为各种任务中引人注目的序列建模和转导模型的组成部分，允许对依赖项进行建模，而无需考虑它们在输入或输出序列中的距离 Dzmitry Bahdanau et al.、Yoon Kim et al.，然而除了少数情况如 Ankur Parikh et al.，在所有情况下该注意力机制都与循环网络结合使用。在这项工作中，我们提出了名为 Transformer 的一种避免重复的模型架构，而是完全依靠注意力机制来绘制输入和输出之间的全局依赖关系，而 Transformer 允许显著更多的并行化，并且在八个 P100 GPU 上进行了短短 12 小时的训练后，可以在翻译质量方面达到新的水平。

2.3.3 背景

减少顺序计算的目标也构成了 Lukasz Kaiser et al. 的扩展神经 GPU、Nal Kalchbrenner et al. 的 ByteNet 和 Jonas Gehring et al. 的 ConvS2S 基础，所有这些都使用卷积神经网络作为基本构建块，并行计算所有输入和输出位置。在这些模型中，其关联来自两个任意输入或输出位置的信号所需的操作数量随着位置之间的距离而增加，对于 ConvS2S 是线性增长，对于 ByteNet 是对数增长。这使得学习远距离位置之间的依赖关系变得更加困难 Sepp Hochreiter et al.，在 Transformer 中，这被减少到恒定数量的操作，尽管代价是由于平均注意力加权位置而导致有效分辨率降低，我们用多头注意力来抵消这种影响，如第 3.2 节所述。自注意 (Self-attention) 有时也称为内注意 (intra-attention)，

这是一种将单个序列的不同位置关联起来以计算序列表示的注意机制，而自注意力已成功用于各种任务，包括阅读理解、抽象摘要、文本蕴涵和学习与任务无关的句子表示上 Jianpeng Cheng et al.、Ankur Parikh et al.、Romain Paulus et al.、Zhouhan Lin et al.。而端到端记忆网络基于循环注意机制而不是序列对齐循环，并且已被证明在简单语言问答和语言建模任务上表现良好 Sainbayar Sukhbaatar et al.，然而据我们所知 Transformer 是第一个完全依赖自注意力来计算其输入和输出表示而不使用序列对齐 RNN 或卷积的转换模型。在接下来的部分中，我们将描述 Transformer，激发自我注意并讨论它相对于 Lukasz Kaiser et al.、Nal Kalchbrenner et al. 和 Jonas Gehring et al. 等模型的优势。

2.3.4 模型架构

大多数具有竞争力的神经序列转导模型都具有编码器-解码器结构 Kyunghyun Cho et al.、Dzmitry Bahdanau et al.、Ilya Sutskever et al.。在此编码器将符号表示的输入序列 (x_1, \dots, x_n) 映射到连续表示的序列 $\mathbf{z} = (z_1, \dots, z_n)$ 。给定 \mathbf{z} 解码器然后生成一个符号的输出序列 (y_1, \dots, y_m) ，一次一个元素。在每一步其模型都是自回归的 Alex Graves et al.，在生成下一个时会将先前生成的符号作为额外的输入使用。Transformer 遵循这一整体架构，使用堆叠的自注意力和逐点、完全连接的编码器和解码器层，分别如图 1 的左半部分和右半部分所示。

1. 編碼器和解碼器堆棧 (Encoder and Decoder Stacks)

編碼器 (Encoder): 編碼器由 $N = 6$ 個相同層的堆棧組成，每層有兩個子層，第一個是多頭自注意力機制，第二個是簡單的、位置明智的全連接前饋網絡，我們在兩個子層的每一個周圍都使用了一個殘差連接 Kaiming He et al.，然後是層歸一化 Jimmy Lei Ba et al.。即每個子層的輸出是 $\text{LayerNorm}(x + \text{Sublayer}(x))$ ，其中 $\text{Sublayer}(x)$ 是子層自己實現的函數， $\boxed{\text{E}}$ 了促進這些殘差連接，模型中的所有子層以及嵌入層 $\boxed{\text{E}}$ 生維度 $d_{model} = 512$ 的輸出。

解碼器 (Decoder): 解碼器也由 $N = 6$ 個相同層的堆棧組成。除了每個編碼器層中的兩個子層之外，解碼器還插入了第三個子層，該子層對編碼器堆棧的輸出執行多頭注意。而與編碼器類似，我們在每個子層周圍使用殘差連接，然後進行層歸一化，我們還修改了解碼器堆棧中的自 $\boxed{\text{E}}$ 意子層，以防止位置關注後續位置，這種掩碼與輸出嵌入偏移一個位置的事實相結合，確保位置 i 的預測只能依賴於小於 i 位置的已知輸出。

2. 注意力 (Attention)

注意力函数可以描述为将查询和一组键值对映射到输出，其中查询、键、值和输出都是向量。输出计算为值的加权总和，其中分配给每个值的权重由查询与相应键的兼容性函数计算。

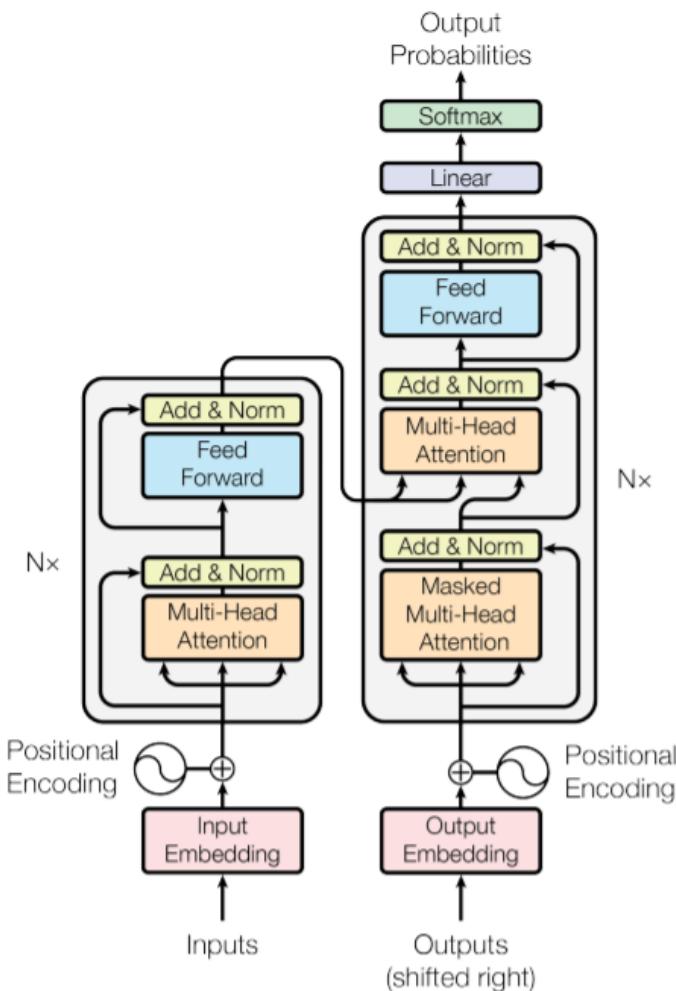


Figure 1: The Transformer - model architecture.

图 2.1 The Transformer - 模型架構 (model architecture)

图 2: (左) 缩放的点积注意力。(右) 多头注意力由多个并行运行的注意力层组成。缩放点积注意力 (Scaled Dot-Product Attention)、多头注意力 (Multi-Head Attention)。

(1) 缩放点积注意力 (Scaled Dot-Product Attention)

我们将我们的特别关注称为“Scaled Dot-Product Attention”(图 2)，输入包括维度 d_k 的查询和键，以及维度 d_v 的值，我们使用所有键计算查询的点积，将每个键除以 $\sqrt{d_k}$ ，并应用 softmax 函数来获得值的权重。在实践中我们同时计算一组查询的注意力函数，将它们打包成一个矩阵 Q ，键和值也一起打包成矩阵 K 和 V ，其键和值也一起打包成矩阵 K 和 V 。其计算的矩阵输出为：

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2.1)$$

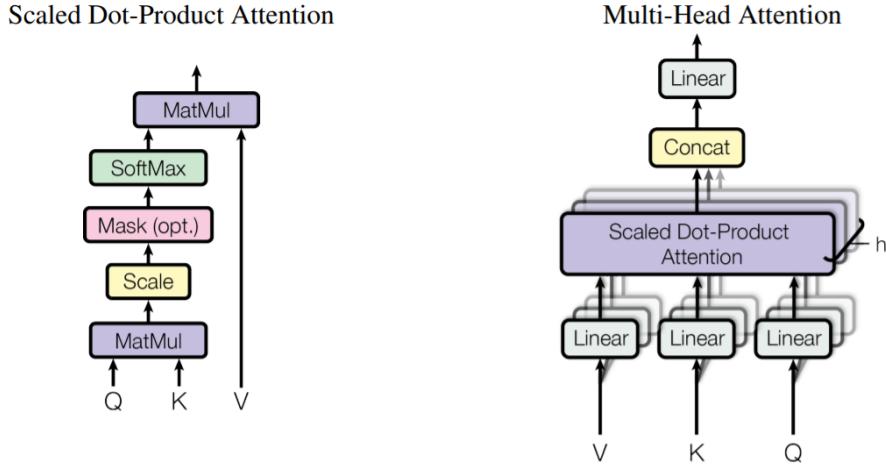


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

图 2.2 缩放的点积注意力和多头注意力

两个最常用的注意力函数是加的注意力 Dzmitry Bahdanau et al. 和点积乘的注意力，除了 $\frac{1}{\sqrt{dk}}$ 的缩放因子之外，点积注意力与我们的方法相同。加法的注意力使用上具有单个隐藏层的前馈网络计算兼容性函数，虽然两者在理论复杂度上相似，但点积注意力在实践中速度更快，空间效率更高，因为它可以使用高度优化的矩阵乘法程序码来实现。虽然对于较小的 d_k 值，这两种机制的表现相似，但加法注意力优于点积注意力，而无需针对较大的 d_k 值进行缩放 Denny Britz et al.，我们怀疑对于较大的 d_k 值，点积的量级会变大，从而将 softmax 函数推入梯度极小的区域^①，为了抵消这种影响，我们通过 $\frac{1}{\sqrt{dk}}$ 缩放点积。

(2) 多头注意力 (Multi-Head Attention)

我们发现，与使用 d_{model} 维度的键、值和查询执行单个注意力函数不同，我们发现将查询、键和值分别线性投影到 d_k 、 d_k 和 d_v 维度的不同学习线性投影 h 次是有益的。然后在查询、键和值的这些投影版本中的每一个上，我们并行执行注意力功能，产生 d_v 维输出值。这些被连接并再次投影，从而产生最终值，如图 2 所示。

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \quad (2.2)$$

其中投影是参数矩阵 $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in$

^① 为了说明点积变大的原因，假设 q 和 k 的分量是均值为 0 且方差为 1 的独立随机变量，那么他们的点积 $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ ，均值为 0，方差为 d_k 。

$\mathbb{R}^{hd_v \times d_{model}}$ 。在这项工作中，我们使用了 $h = 8$ 个平行的注意力层或头部。对于每一个，我们使用 $d_k = d_v = d_{model}/h = 64$ ，由于每个头部的维数减少，总的计算成本与全维的单头注意力相似。

(3) 注意力在我们模型中的应用 (Applications of Attention in our Model)

Transformer 以三种不同的方式使用多头注意力：

- 在“编码器-解码器注意力”层中，查询来自前一个解码器层，记忆键和值来自编码器的输出，这允许解码器中的每个位置都参与输入序列中的所有位置，这模仿了序列到序列模型中典型的编码器-解码器注意力机制，例如 Yonghui Wu et al.、Dzmitry Bahdanau et al.、Jonas Gehring et al.。
- 编码器包含自注意力层，在自注意力层中，所有的键、值和查询都来自同一个地方，在这种情况下是编码器上一层的输出。编码器中的每个位置都可以参与编码器前一层中的所有位置。
- 类似地，解码器中的自注意力层允许解码器中的每个位置关注解码器中直到并包括该位置的所有位置，而我们需要防止解码器中的左向信息流以保留自回归特性。我们通过屏蔽（设置为 $-\infty$ ）softmax 输入中与非法连接相对应的所有值来在缩放点积注意力内部实现这一点。See Figure 2.

3. 位置前馈网络 (Position-wise Feed-Forward Networks)

除了注意力子层之外，我们的编码器和解码器中的每一层都包含一个完全连接的前馈网络，该网络分别且相同地应用于每个位置，这由两个线性变换组成，中间有一个 ReLU 激活。

$$\text{FFN}(x) = \max(0, xW_1 + b_1) W_2 + b_2 \quad (2.3)$$

虽然不同位置的线性变换是相同的，但它们在层与层之间使用不同的参数。另一种描述方式是两个卷积核大小为 1。输入输出维数 $d_{model} = 512$ ，内层维数 $d_{ff} = 2048$ 。

4. 嵌入和 Softmax (Embeddings and Softmax)

其他序列转导模型类似，我们使用学习嵌入将输入标记和输出标记转换为维度 d_{model} 的向量，我们还使用通常学习的线性变换和 softmax 函数将解码器输出转换为预测的下一个标记概率。在我们的模型中，我们在两个嵌入层和 pre-softmax 线性变换之间共享相同的权重矩阵，类似于 Ofir Press et al.，在嵌入层中，我们将这些权重乘以 $\sqrt{d_{model}}$ 。

5. 位置编码 (Positional Encoding)

由于我们的模型不包含递归和卷积，为了让模型利用序列的顺序，我们必须注入一些关于标记在序列中的相对或绝对位置的信息，为此我们将“位置编码”添加到编

码器和解码器堆栈底部的输入嵌入中。位置编码与嵌入具有相同的维度 d_{model} 因此可以将两者相加，位置编码有多种选择，学习和固定 Jonas Gehring et al.。

表 1：不同层类型的最大路径长度、每层复杂度和最小顺序操作数， n 是序列长度、 d 是表示维度、 k 是卷积核大小， r 是受限自注意力中的邻域大小。

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

图 2.3 不同层类型的最大路径长度、每层复杂度和最小顺序操作数

在這項工作中，我們使用不同頻率的正弦和余弦函數：

$$\begin{aligned} PE_{(pos,2i)} &= \sin \left(pos / 10000^{2i/d_{model}} \right) \\ PE_{(pos,2i+1)} &= \cos \left(pos / 10000^{2i/d_{model}} \right) \end{aligned} \quad (2.4)$$

其中 pos 是位置， i 是维度。也就是说，位置编码的每个维度对应一个正弦曲线。波长形成从 2π 到 $10000 \cdot 2\pi$ 的几何级数。我们选择这个函数是因为我们假设它可以让模型很容易地学习通过相对位置来参与，因为对于任何固定的偏移量 k ， PE_{pos+k} 可以表示为 PE_{pos} 的线性函数。我们还尝试使用学习的位置嵌入 Jonas Gehring et al.，发现两个版本产生了几乎相同的结果（见表 3 行 (E)），我们选择了正弦版本，因为它可能允许模型外推到比训练期间遇到的序列长度更长的序列长度。

2.3.5 Why Self-Attention

在本节中，我们将自注意力层的各个方面与通常用于将一个可变长度的符号表示序列 (x_1, \dots, x_n) 映射到另一个等长序列 (z_1, \dots, z_n) ，与 $x_i, z_i \in \mathbb{R}^d$ ，例如典型序列转导编码器或解码器中的隐藏层。为了激发我们使用自注意力，我们考虑了三个需求。一个是每层的总计算复杂度。另一个是可以并行化的计算量，通过所需的最小顺序操作数来衡量。第三个是网络中远程依赖之间的路径长度。学习远程依赖是许多序列转导任务中的关键挑战，影响学习这种依赖性能力的一个关键因素是前向和后向信号必须在网络中穿越的路径长度。输入和输出序列中任意位置组合之间的这些路径越短，学习远程依赖关系就越容易 Sepp Hochreiter et al.，因此我们还比较了由不同层类型组成的

网络中任意两个输入和输出位置之间的最大路径长度。如表 1 中所述，自注意力层将所有位置与恒定数量的顺序执行操作连接起来，而循环层需要 $O(n)$ 顺序操作。在计算复杂度方面，当序列长度 n 小于表示维数 d 时，自注意力层比循环层更快，这是机器翻译中最先进模型使用的句子表示的最常见情况，例如词片 Yonghui Wu et al. 和字节对 Rico Sennrich et al. 表示。为了提高涉及非常长序列的任务的计算性能，可以将自注意力限制为仅考虑输入序列中以相应输出位置为中心的大小为 r 的邻域。这会将最大路径长度增加到 $O(n = r)$ ，我们计划在未来的工作中进一步研究这种方法。内核宽度 $k < n$ 的单个卷积层不会连接所有输入和输出位置对。这样做需要在连续内核的情况下堆叠 $O(n = k)$ 卷积层，或者在扩张卷积的情况下需要 $O(\log k(n))$ Nal Kalchbrenner et al.，增加任意两个位置之间最长路径的长度在网络中，卷积层通常比循环层成本高 k 倍。然而 Francois Chollet 的可分离卷积将复杂度大大降低到 $O(k \cdot n \cdot d + n \cdot d^2)$ 。然而即使 $k = n$ 可分离卷积的复杂度也等于自注意力层和逐点前馈层的组合，我们在我们的模型中采用的方法。作为附带好处，self-attention 可以产生更多可解释的模型。我们从我们的模型中检查注意力分布，并在附录中展示和讨论示例。个人注意力不仅清楚地学习执行不同的任务，而且许多人似乎表现出与句子的句法和语义结构相关的行为。

2.3.6 训练

本节描述了我们模型的训练机制。

1. 训练数据和批处理 (Training Data and Batching)

我们在包含约 450 万个句子对的标准 WMT 2014 英德数据集上进行了训练。句子使用字节对编码 Denny Britz et al. 进行编码，它具有大约 37000 个标记的共享源目标词汇表。对于英法，我们使用了明显更大的 WMT 2014 英法数据集，该数据集由 3600 万个句子组成，并将标记拆分为 32000 个单词词表 Yonghui Wu et al.，句子对按近似序列长度分批在一起。每个训练批次包含一组句子对，其中包含大约 25000 个源标记和 25000 个目标标记。

2. 硬体和时间表 (Hardware and Schedule)

我们在一台配备 8 个 NVIDIA P100 GPU 的机器上训练我们的模型，对于我们使用整篇论文中描述的超参数的基本模型，每个训练步骤大约需要 0.4 秒。我们对基本模型进行了总共 100,000 步或 12 小时的训练，对于我们的大型模型（在表 3 的底线中进行了描述），步进时间为 1.0 秒，大模型训练了 300,000 步（3.5 天）。

3. Optimizer

我们使用了 Adam 优化器 Diederik Kingma et al.，其中 $\beta_1 = 0.9, \beta_2 = 0.98$ and $\epsilon = 10^{-9}$ 。我们根据以下公式在训练过程中改变学习率：

$$\text{lrate} = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (2.5)$$

这对应于在第一个 $warmup_steps$ 训练步骤中线性增加学习率，然后与步数的平方根成反比地减少学习率。我们使用了 $warmup_steps = 4000$ 。

4. 正则化 (Regularization)

我们在训练期间采用三种类型的正则化：

(1) Residual Dropout

我们将 Nitish Srivastava 的 dropout 应用于每个子层的输出，然后再将其添加到子层输入中并进行归一化。此外我们将 dropout 应用于编码器和解码器堆栈中的嵌入和位置编码的总和。对于基本模型，我们使用 $P_{drop} = 0.1$ 的比率。

表 2: Transformer 在 English-German 和 English-to-French newstest2014 测试中取得了比之前最先进的模型更好的 BLEU 分数，而训练成本只是其中的一小部分。

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.8	$2.3 \cdot 10^{19}$	

图 2.4 : Transformer 在 English-German 和 English-to-French newstest2014

(2) 标签平滑 (Label Smoothing)

在训练期间，我们采用了值 $\epsilon_{ls} = 0.1$ Christian Szegedy et al. 的标签平滑，这会降低困惑度，因为模型会变得更加不确定，但会提高准确性和 BLEU 分数。

2.3.7 结果

1. 机器翻译 (Machine Translation)

在 WMT 2014 英德翻译任务中，大转换器模型 (表 2 中的 Transformer (big)) 以超过 2.0 的 BLEU 超过先前报告的最佳模型 (包括集成)，建立了一个至今为止最好的 BLEU 分数为 28.4。该模型的配置列在表 3 的底部。在 8 个 P100 GPU 上训练需要 3.5 天，甚

至我们的基础模型也超过了所有先前发布的模型和集成模型，其训练成本仅为任何竞争模型的一小部分。在 WMT 2014 英法翻译任务中，我们的大模型达到 41.0 的 BLEU 分数，优于之前发布的所有单个模型，低于之前状态的训练成本 1/4。为英语到法语训练的 Transformer (大) 模型使用辍学率 $P_{drop} = 0.1$ ，而不是 0.3。对于基本模型，我们使用了通过平均最后 5 个检查点获得的单个模型，这些检查点以 10 分钟的间隔写入。对于大型模型，我们对最后 20 个检查点进行了平均。我们使用了波束大小为 4 且长度惩罚 $\alpha = 0.6$ Yonghui Wu et al. 的波束搜索，这些超参数是在开发集上进行实验后选择的，我们在推理过程中将最大输出长度设置为输入长度 +50，但在可能的情况下提前终止 Yonghui Wu et al.。表 2 总结了我们的结果，并将我们的翻译质量和训练成本与文献中的其他模型架构进行了比较。我们通过将训练时间、使用的 GPU 数量和每个 GPU 5 的持续单精度浮点容量的估计相乘来估计用于训练模型的浮点运算次数。

2. 型号变化 (Model Variations)

为了评估 Transformer 不同组件的重要性，我们以不同的方式改变了我们的基本模型，测量了开发集 newstest2013 上英德翻译的性能变化，我们使用了前一节中描述的波束搜索，但没有检查点平均。我们在表 3 中展示了这些结果，在表 3 的行 (A) 中，我们改变了注意力头的数量以及注意力键和值维度，保持计算量不变，如第 3.2.2 节所述，虽然单头注意力比最佳设置差 0.9 BLEU，但质量也会随着头数过多而下降。

表 3：Transformer 架构的变化，未列出的值与基本模型的值相同。所有指标都在英语到德语的翻译开发集 newstest2013 上，根据我们的字节对编码，列出的困惑是每个单词的，不应与每个单词的困惑进行比较。

表 4：Transformer 很好地推广到英语选区解析 (结果在 WSJ 的第 23 节) 在表 3 的行 (B) 中，我们观察到减少注意力密钥大小 d_k 会损害模型质量。这表明确定兼容性并不容易，并且比点积更复杂的兼容性函数可能是有益的，我们在行 (C) 和 (D) 中进一步观察到，正如预期的那样，更大的模型更好，并且 dropout 非常有助于避免过度拟合。在行 (E) 中，我们用学习的位置嵌入 Jonas Gehring et al. 替换了我们的正弦位置编码，并观察到与基本模型几乎相同的结果。

3. 英语选区解析 (English Constituency Parsing)

为了评估 Transformer 是否可以推广到其他任务，我们对英语选区解析进行了实验，这项任务提出了具体的挑战：输出受到强大的结构约束，并且明显长于输入。此外 RNN 序列到序列模型无法在小数据机制中获得最先进的结果 Vinyals et al.，我们在 Mitchell P Marcus 的 Penn Treebank 的华尔街日报 (WSJ) 部分训练了一个 $d_{model} = 1024$ 的 4 层转换器，大约有 40K 个训练句子。我们还在半监督环境中对其进行训练，使用更大的高置信度和 BerkleyParser 语料库，其中包含大约 1700 万个句子 Vinyals et al.，我

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)										5.29	24.9	
										5.00	25.5	
										4.91	25.8	
										5.01	25.4	
(B)									16	5.16	25.1	58
									32	5.01	25.4	60
(C)									2	6.11	23.7	36
									4	5.19	25.3	50
									8	4.88	25.5	80
									256	5.75	24.5	28
									1024	4.66	26.0	168
									1024	5.12	25.4	53
									4096	4.75	26.2	90
									0.0	5.77	24.6	
(D)									0.2	4.95	25.5	
									0.0	4.67	25.3	
									0.2	5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16				0.3	300K	4.33	26.4	213

图 2.5 Transformer 架构的变化

们在 WSJ only setting 中使用了 16K 标记的词汇表，在半监督设置中使用了 32K 标记的词汇表。我们只进行了少量实验来选择第 22 节开发集上的 dropout、注意力和残差(第 5.4 节)、学习率和波束大小，所有其他参数与英语到德语基础翻译模型保持不变。在推理过程中，我们将最大输出长度增加到输入长度 +300，我们仅对 WSJ 和半监督设置使用了 21 和 $\alpha = 0.3$ 的波束大小。我们在表 4 中的结果表明，尽管缺乏针对特定任务的调整，但我们的模型表现出奇的好，产生了比除循环神经网络语法 Chris Dyer et al. 以外的所有先前报告的模型更好的结果。与 RNN 序列到序列模型 Vinyals et al. 相比，即使仅在 WSJ 训练集的 40K 句子上进行训练，Transformer 的性能也优于 Berkeley-Parser Slav Petrov et al.。

2.3.8 结论

在这项工作中，我们提出了 Transformer，这是第一个完全基于注意力的序列转换模型，用多头自注意力取代了编码器-解码器架构中最常用的循环层。对于翻译任务，Transformer 的训练速度明显快于基于循环或卷积层的架构。在 WMT 2014 English-to-German 和 WMT 2014 English-to-French 翻译任务中，我们达到了最先进的水平。在前

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)

Parser	Training	WSJ 23 F1
Vinyals & Kaiser et al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser et al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

图 2.6 Transformer 很好地推广到英语选区解析

一个任务中，我们最好的模型甚至优于所有先前报告的集成。我们对基于注意力模型的未来感到兴奋，并计划将它们应用于其他任务。我们计划将 Transformer 扩展到涉及文本以外的输入和输出模式的问题，并研究局部、受限的注意力机制以有效处理大型输入和输出，如图像、音频和影像。减少世代的连续性是我们的另一个研究目标，我们用于训练和评估模型的代码可在 <https://github.com/tensorflow/tensor2tensor> 上找到。

第三章 技术文章与讨论

该作业其专案为 kancheng/kan-cs-report-in-2021，程式码则可于 kan-cs-report-in-2021/AI/pytorch-transformer/code 找到，实验设备为 MacBook Pro (Retina, 15-inch, Mid 2014) 和 Acer Aspire R7。同时参考的技术文章与论文连结皆于该专案的 init.md 文件中条列呈现。该专案根据 Harvard NLP 的 The Annotated Transformer 与名为 fawazsammani/chatbot-transformer 的 GitHub 专案，该专案使用 Cornell Movie Dialog Corpus 资料，进行 Chatbot using Transformers 的实作。作业结果为前者为 transformer-harvard-demo.ipynb，而后者为 transformer-chatbot-demo.ipynb 档案。

在处理过程中大多遇到 Pytorch 套件不相容、版本过旧的问题，这类问题来源大多是 Pytorch 早期开发功能变动所造成，但可以从 Harvard NLP 与 Chatbot using Transformers 的过程中看出整个 Transformer 的设计概念与运作，程式码的版本进行查错与修正后执行，而两个范例的 Epoch 等训练结果，也于该章进行说明。

3.1 技术文章

在此本章将近期所搜集跟阅读的技术文章与资源进行条列与整理后做表如下所示：

3.1.1 编程

主题节录	用途	分类	备注
tensorflow/tensor2tensor ^[3]	技术	编程	TensorFlow
paperswithcode - Attention Is All You Need ^[4]	技术	编程	專案索引
tensorflow/models ^[5]	技术	编程	TensorFlow
graykode/nlp-tutorial ^[6]	技术	编程	NLP
SamLynnEvans/Transformer ^[7]	技术	编程	
huggingface/transformers ^[8]	技术	编程	
harvardnlp/annotated-transformer ^[9]	技术	编程	
fawazsammani/chatbot-transformer ^[10]	技术	编程	

3.1.2 文献说明与综述整理

主题节录	用途	分类	备注
Attention is all you need ^[2]	论文	文献	最早的文献
On the Integration of Self-Attention and Convolution ^[11]	论文	文献	
视觉 Transformer 综述 ^[12]	技术	文献整理	综述
Transformer 最新综述 ^[13]	技术	文献整理	综述
Self-Attention 和 CNN 的优雅集成 ^[14]	技术	文献说明	研究

3.1.3 技术文件与实现

主题节录	用途	分类	备注
搞懂 Transformer 结构 ^[15]	技术	文件	PyTorch
How to code The Transformer in Pytorch ^[16]	技术	文件	PyTorch
The Annotated Transformer ^[17]	技术	文件	harvardnlp
从零实现了 Transformer 模型 ^[18]	技术	文件	
This post is all you need ^[19]	技术	文件	
【機器學習 2021】Transformer (上) ^[20]	技术	影片	
【機器學習 2021】Transformer (下) ^[21]	技术	影片	
Transformer Implementation ^[22]	技术	文件	
Transformer 的 PyTorch 实现 ^[23]	技术	文件	
搞懂 Transformer 结构 ^[24]	技术	文件	
超详细图解 Self-Attention ^[25]	技术	文件	
Transformer ^[26]	技术	文件	
Transformer 详解 ^[27]	技术	文件	

3.2 程式码与注解说明

该小节分为两大部分，其一为 Harvard NLP 的成果，前半段根据本作业已经修正的 Pytorch 版本，后续未成功的部分则根据原本的 Harvard NLP，其二为 Chatbot Transformer 的训练。另外 Harvard NLP 的成果与原本的 Transformer 文献架构可以互相呼应并可以当做该论文的补充说明。

3.2.1 Harvard NLP

虽然"Attention is All You Need" ^①一文中提出的 Transformer 网络结构最近引起了很多人的关注。Transformer 不仅能够明显地提升翻译质量，还为许多 NLP 任务提供了新的结构。虽然原文写得很清楚，但实际上大家普遍反映很难正确地实现 Transformer。

^① <https://arxiv.org/abs/1706.03762>

所以我们为此文章写了篇注解文档，并给出了一行行实现的 Transformer 的代码。本文档删除了原文的一些章节并进行了重新排序，并在整个文章中加入了相应的注解。此外，本文档以 Jupyter notebook 的形式完成，本身就是直接可以运行的代码实现，总共有 400 行库代码，在 4 个 GPU 上每秒可以处理 27,000 个 tokens。

想要运行此工作，首先需要安装 PyTorch^①。这篇文档完整的 notebook 文件及依赖可在 GitHub^② 或 Google Colab^③ 上找到。需要注意的是，此注解文档和代码仅作为研究人员和开发者的入门版教程。这里提供的代码主要依赖 OpenNMT^④ 进行实现，想了解更多关于此模型的其他实现版本可以查看 Tensor2Tensor^⑤ (tensorflow 版本) 和 Sockeye^⑥ (mxnet 版本)

Alexander Rush (@harvardnlp^⑦ or srush@seas.harvard.edu)

0. 准备工作 (Prelims)

```
# http://download.pytorch.org/whl/cu80/torch-0.3.0.post4
# 该版本过于老旧

# http://nlp.seas.harvard.edu/2018/04/03/attention.html
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import math, copy, time
from torch.autograd import Variable
import matplotlib.pyplot as plt
import seaborn
seaborn.set_context(context="talk")
%matplotlib inline
print(torch.__version__)
```

内容目录 (Table of Contents)

- 准备工作 (Prelims)
- 背景 (Background)
- 模型结构 (Model Architecture)
 - Encoder 和 Decoder (Encoder and Decoder Stacks)
 - Encoder

^① <https://pytorch.org/>

^② <https://github.com/harvardnlp/annotated-transformer>

^③ <https://drive.google.com/file/d/1xQXSv6mtAOLXxEMi8RvaW8TW-7bvYBDF/view?usp=sharing>

^④ <https://opennmt.net/>

^⑤ <https://github.com/tensorflow/tensor2tensor>

^⑥ <https://github.com/aws-labs/sockeye>

^⑦ <https://twitter.com/harvardnlp>

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Lukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

图 3.1 Attention is All You Need

- Decoder
- Attention
- Attention 在模型中的应用 (Applications of Attention in our Model)
- Position-wise 前馈网络 (Position-wise Feed-Forward Networks)
- Embedding 和 Softmax (Embeddings and Softmax)
- 位置编码 (Positional Encoding)
- 完整模型 (Full Model)
- 训练 (Training)
 - 批和掩码 (Batches and Masking)
 - 训练循环 (Training Loop)
 - 训练数据和批处理 (Training Data and Batching)
 - 硬件和训练进度 (Hardware and Schedule)

Harvard Transformer

參考 Harvard Transformer 進行修正，此 Pytorch 版本為 1.10.0

```
In [1]: # !pip install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl numpy matplotlib spacy torchtext seaborn

In [2]: # http://nlp.seas.harvard.edu/2018/04/03/attention.html
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import math, copy, time
from torch.autograd import Variable
import matplotlib.pyplot as plt
import seaborn
seaborn.set_context(context="talk")
%matplotlib inline
print(torch.__version__)

1.10.0
```

图 3.2 作業版本 Pytorch 修正

- 优化器 (Optimizer)
- 正则化 (Regularization)
- 标签平滑 (Label Smoothing)
- 第一个例子 (A First Example)
 - 数据生成 (Synthetic Data)
 - 损失计算 (Loss Computation)
 - 贪心解码 (Greedy Decoding)
- 真实示例 (A Real World Example)
 - 数据加载 (Data Loading)
 - 迭代器 (Iterators)
 - 多 GPU 训练 (Multi-GPU Training)
 - 训练系统 (Training the System)
 - 附加组件: BPE, 搜索, 平均 (Additional Components: BPE, Search, Averaging)
- 结果 (Results)
 - 注意力可视化 (Attention Visualization)
- 结论 (Conclusion)

1. 背景 (Background)

减少序列处理任务的计算量是一个很重要的问题，也是 Extended Neural GPU、ByteNet 和 ConvS2S 等网络的动机。上面提到的这些网络都是以 CNN 为基础，并行计算所有输入和输出位置的隐藏表示。

在这些模型中，关联来自两个任意输入或输出位置的信号所需的操作数随位置间的距离增长而增长，例如 ConvS2S 呈线性增长，ByteNet 呈现以对数形式增长，这会使

学习较远距离的两个位置之间的依赖关系变得更加困难。而在 Transformer 中，其操作次数则被减少到了常数级别。

Self-attention 有时候也被称为 Intra-attention，是在单个句子不同位置上做的 Attention，并得到序列的一个表示。它能够很好地应用到很多任务中，包括阅读理解、摘要、文本蕴涵，以及独立于任务的句子表示。端到端的网络一般都是基于循环注意力机制而不是序列对齐循环，并且已经有证据表明在简单语言问答和语言建模任务上表现很好。

据我们所了解，Transformer 是第一个完全依靠 Self-attention 而非使用序列对齐的 RNN 或卷积的方式来计算输入输出表示的转换模型。

2. 模型结构 (Model Architecture)

目前大部分比较热门的神经序列转换模型都有 Encoder-Decoder 结构^①。Encoder 将输入序列 (x_1, \dots, x_n) 映射到一个连续表示序列 $z = (z_1, \dots, z_n)$ 。

对于编码得到的 z ，Decoder 每次解码生成一个符号，直到生成完整的输出序列： (y_1, \dots, y_m) 。对于每一步解码，模型都是自回归的^②，即在生成下一个符号时将先前生成的符号作为附加输入。

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """

    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)
```

^① <https://arxiv.org/abs/1409.0473>

^② <https://arxiv.org/abs/1308.0850>

```
def decode(self, memory, src_mask, tgt, tgt_mask):
    return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

Transformer的整体结构如下图所示,在Encoder和Decoder中都使用了Self-attention, Point-wise 和全连接层。Encoder 和 decoder 的大致结构分别如下图的左半部分和右半部分所示。

3. Encoder 和 Decoder (Encoder and Decoder Stacks)

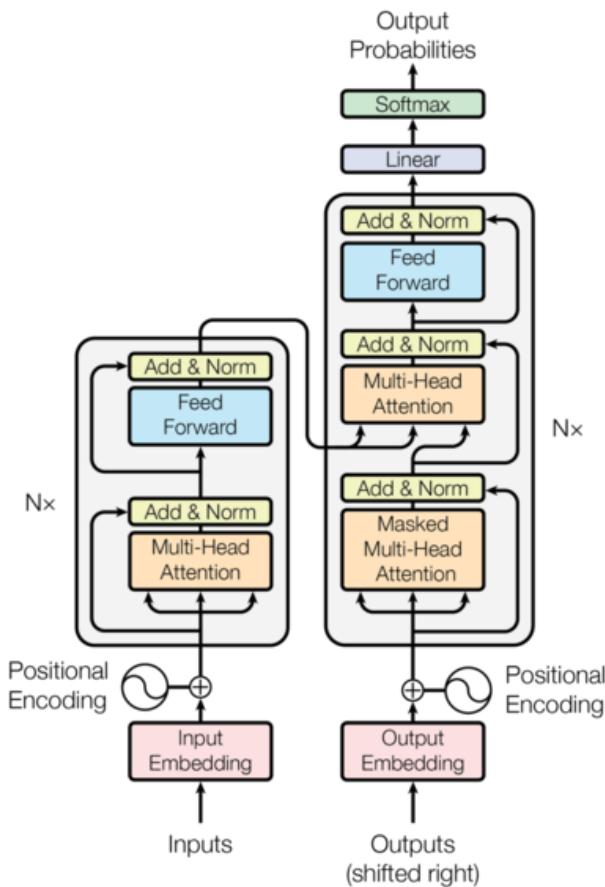


图 3.3 Transformer 的整体结构

4. Encoder

Encoder 由 $N = 6$ 个相同的层组成。

```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

我们在每两个子层之间都使用了残差连接 (Residual Connection)⁽¹⁾ 和归一化 (layer normalization)⁽²⁾。

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

也就是说，每个子层的输出为 $\text{LayerNorm}(x + \text{Sublayer}(x))$ ，其中 $\text{Sublayer}(x)$ 是由子层自动实现的函数。我们在每个子层的输出上使用 Dropout⁽³⁾，然后将其添加到下一子层的输入并进行归一化。

为了能方便地使用这些残差连接，模型中所有的子层和 Embedding 层的输出都设定了相同的维度，即 $d_{model} = 512$ 。

(1) <https://arxiv.org/abs/1512.03385>

(2) <https://arxiv.org/abs/1607.06450>

(3) <https://jmlr.org/papers/v15/srivastava14a.html>

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

每层都有两个子层组成。第一个子层实现了“多头”的 Self-attention，第二个子层则是一个简单的 Position-wise 的全连接前馈网络。

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

5. Decoder

Decoder 也是由 $N = 6$ 个相同层组成。

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
```

```

for layer in self.layers:
    x = layer(x, memory, src_mask, tgt_mask)
return self.norm(x)

```

除了每个编码器层中的两个子层之外，解码器还插入了第三种子层对编码器栈的输出实行“多头”的 Attention。与编码器类似，我们在每个子层两端使用残差连接进行短路，然后进行层的规范化处理。

```

class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)

```

我们还修改解码器中的 Self-attention 子层以防止当前位置 Attend 到后续位置。这种 Masked 的 Attention 是考虑到输出 Embedding 会偏移一个位置，确保了生成位置 i 的预测时，仅依赖小于 i 的位置处的已知输出，相当于把后面不该看到的信息屏蔽掉。

```

def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0

```

下面的 Attention mask 图显示了允许每个目标词（行）查看的位置（列）。在训练期间，当前解码位置的词不能 Attend 到后续位置的词。

```

plt.figure(figsize=(5,5))
plt.imshow(subsequent_mask(20)[0])
None

```

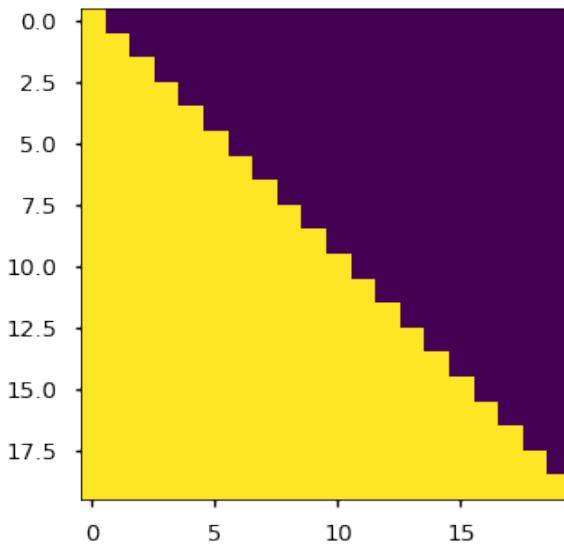


图 3.4 Attention mask 图

6. Attention

Attention 函数可以将 Query 和一组 Key-Value 对映射到输出，其中 Query、Key、Value 和输出都是向量。输出是值的加权和，其中分配给每个 Value 的权重由 Query 与相应 Key 的兼容函数计算。

我们称这种特殊的 Attention 机制为"Scaled Dot-Product Attention"。输入包含维度为 d_k 的 Query 和 Key，以及维度为 d_v 的 Value。我们首先分别计算 Query 与各个 Key 的点积，然后将每个点积除以 $\sqrt{d_k}$ ，最后使用 Softmax 函数来获得 Key 的权重。

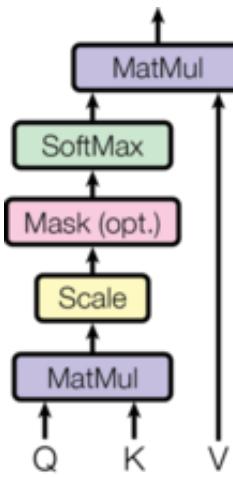


图 3.5 Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (3.1)$$

```

def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
        / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn

```

两种最常用的 Attention 函数是加和 Attention^①和点积（乘积）Attention，我们的算法与点积 Attention 很类似，但是 $\frac{1}{\sqrt{d_k}}$ 的比例因子不同。加和 Attention 使用具有单个隐藏层的前馈网络来计算兼容函数。虽然两种方法理论上的复杂度是相似的，但在实践中，点积 Attention 的运算会更快一些，也更节省空间，因为它可以使用高效的矩阵乘法算法来实现。

虽然对于较小的 d_k ，这两种机制的表现相似，但在不放缩较大的 d_k 时，加和 Attention 要优于点积 Attention^②。我们怀疑，对于较大的 d_k ，点积大幅增大，将 Softmax 函数推向具有极小梯度的区域（为了阐明点积变大的原因，假设 q 和 k 是独立的随机变量，平均值为 0，方差为 1，这样他们的点积为 $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ ，同样是均值 0 为方差为 d_k 。为了抵消这种影响，我们用 $\frac{1}{\sqrt{d_k}}$ 来缩放点积。

“多头”机制能让模型考虑到不同位置的 Attention，另外“多头”Attention 可以在不同的子空间表示不一样的关联关系，使用单个 Head 的 Attention 一般达不到这种效果。

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \\ \text{head}_i &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \quad (3.2)$$

其中参数矩阵为 $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ 。

我们的工作中使用 $h = 8$ 个 Head 并行的 Attention，对每一个 Head 来说有 $d_k = d_v = d_{\text{model}}/h = 64$ 总计算量与完整维度的单个 Head 的 Attention 很相近。

```

class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):

```

^① <https://arxiv.org/abs/1409.0473>

^② <https://arxiv.org/abs/1703.03906>

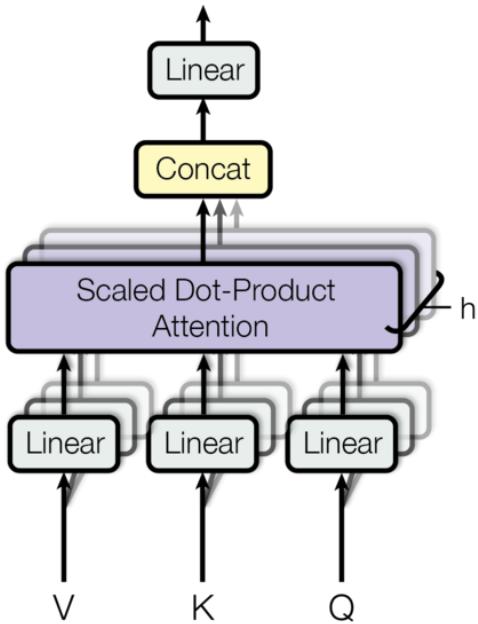


图 3.6 Attention

```

    "Take in model size and number of heads."
super(MultiHeadedAttention, self).__init__()
assert d_model % h == 0
# We assume d_v always equals d_k
self.d_k = d_model // h
self.h = h
self.linears = clones(nn.Linear(d_model, d_model), 4)
self.attn = None
self.dropout = nn.Dropout(p=dropout)

def forward(self, query, key, value, mask=None):
    "Implements Figure 2"
    if mask is not None:
        # Same mask applied to all h heads.
        mask = mask.unsqueeze(1)
    nbatches = query.size(0)

    # 1) Do all the linear projections in batch from d_model => h x d_k
    query, key, value = \
        [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
         for l, x in zip(self.linears, (query, key, value))]

    # 2) Apply attention on all the projected vectors in batch.

```

```

x, self.attn = attention(query, key, value, mask=mask,
                         dropout=self.dropout)

# 3) "Concat" using a view and apply a final linear.
x = x.transpose(1, 2).contiguous() \
    .view(nbatches, -1, self.h * self.d_k)
return self.linears[-1](x)

```

7. Attention 在模型中的应用 (Applications of Attention in our Model)

Transformer 中以三种不同的方式使用了“多头”Attention：

1) 在"Encoder-Decoder Attention" 层，Query 来自先前的解码器层，并且 Key 和 Value 来自 Encoder 的输出。Decoder 中的每个位置 Attend 输入序列中的所有位置，这与 Seq2Seq 模型中的经典的 Encoder-Decoder Attention 机制^①一致。

2) Encoder 中的 Self-attention 层。在 Self-attention 层中，所有的 Key、Value 和 Query 都来同一个地方，这里都是来自 Encoder 中前一层的输出。Encoder 中当前层的每个位置都能 Attend 到前一层的所有位置。

3) 类似的，解码器中的 Self-attention 层允许解码器中的每个位置 Attend 当前解码位置和它前面的所有位置。这里需要屏蔽解码器中向左的信息流以保持自回归属性。具体的实现方式是在缩放后的点积 Attention 中，屏蔽（设为负无穷）Softmax 的输入中所有对应着非法连接的 Value。

8. Position-wise 前馈网络 (Position-wise Feed-Forward Networks)

除了 Attention 子层之外，Encoder 和 Decoder 中的每个层都包含一个全连接前馈网络，分别地应用于每个位置。其中包括两个线性变换，然后使用 ReLU 作为激活函数。

$$\text{FFN}(x) = \max(0, xW_1 + b_1) W_2 + b_2 \quad (3.3)$$

虽然线性变换在不同位置上是相同的，但它们在层与层之间使用不同的参数。这其实是相当于使用了两个内核大小为 1 的卷积。这里设置输入和输出的维数为 $d_{model} = 512$ ，内层的维度为 $d_{ff} = 2048$ 。

```

class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

```

^① <https://arxiv.org/abs/1609.08144>

```
def forward(self, x):
    return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

9. Embedding 和 Softmax (Embeddings and Softmax)

与其他序列转换模型类似，我们使用预学习的 Embedding 将输入 Token 序列和输出 Token 序列转化为 d_{model} 维向量。我们还使用常用的预训练的线性变换和 Softmax 函数将解码器输出转换为预测下一个 Token 的概率。在我们的模型中，我们在两个 Embedding 层和 Pre-softmax 线性变换之间共享相同的权重矩阵，类似于^①。在 Embedding 层中，我们将这些权重乘以 $\sqrt{d_{model}}$ 。

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

10. 位置编码 (Positional Encoding)

由于我们的模型不包含递归和卷积结构，为了使模型能够有效利用序列的顺序特征，我们需要加入序列中各个 Token 间相对位置或 Token 在序列中绝对位置的信息。在这里，我们将位置编码添加到编码器和解码器栈底部的输入 Embedding。由于位置编码与 Embedding 具有相同的维度 d_{model} ，因此两者可以直接相加。其实这里还有许多位置编码可供选择，其中包括可更新的和固定不变的^②。

在此项工作中，我们使用不同频率的正弦和余弦函数：

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\text{pos}/10000^{2i/d_{model}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\text{pos}/10000^{2i/d_{model}}\right) \end{aligned} \quad (3.4)$$

其中 pos 是位置， i 是维度。也就是说，位置编码的每个维度都对应于一个正弦曲线，其波长形成从 2π 到 $10000 \cdot 2\pi$ 的等比级数。我们之所以选择了这个函数，是因为我们假设它能让模型很容易学会 Attend 相对位置，因为对于任何固定的偏移量 k ， PE_{pos+k} 可以表示为 PE_{pos} 的线性函数。

^① <https://arxiv.org/abs/1608.05859>

^② <https://arxiv.org/pdf/1705.03122.pdf>

此外，在编码器和解码器堆栈中，我们在 Embedding 与位置编码的加和上都使用了 Dropout 机制。在基本模型上，我们使用 $PE_{drop} = 0.1$ 的比率。

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                          requires_grad=False)
        return self.dropout(x)
```

如下所示，位置编码将根据位置添加正弦曲线。曲线的频率和偏移对于每个维度是不同的。

```
plt.figure(figsize=(15, 5))
pe = PositionalEncoding(20, 0)
y = pe.forward(Variable(torch.zeros(1, 100, 20)))
plt.plot(np.arange(100), y[0, :, 4:8].data.numpy())
plt.legend(["dim %d"%p for p in [4,5,6,7]])
None
```

11. 完整模型 (Full Model)

下面定义了连接完整模型并设置超参的函数。

```
def make_model(src_vocab, tgt_vocab, N=6,
               d_model=512, d_ff=2048, h=8, dropout=0.1):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
```

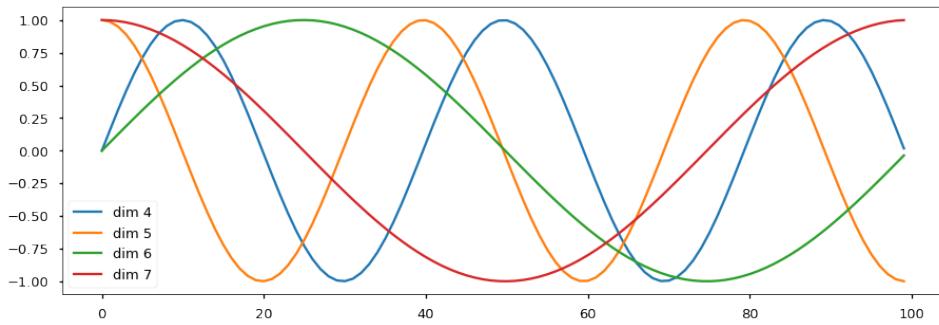


图 3.7 曲线的频率和偏移

```

position = PositionalEncoding(d_model, dropout)
model = EncoderDecoder(
    Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
    Decoder(DecoderLayer(d_model, c(attn), c(attn),
        c(ff), dropout), N),
    nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
    nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
    Generator(d_model, tgt_vocab))

# This was important from their code.
# Initialize parameters with Glorot / fan_avg.
for p in model.parameters():
    if p.dim() > 1:
        #nn.init.xavier_uniform(p)
        nn.init.xavier_uniform_(p)
return model

```

```

# Small example model.
tmp_model = make_model(10, 10, 2)
None

```

12. 训练 (Training)

本节介绍模型的训练方法。

快速穿插介绍训练标准编码器解码器模型需要的一些工具。首先我们定义一个包含源和目标句子的批训练对象用于训练，同时构造掩码。

13. 批和掩码 (Batches and Masking)

```

class Batch:
    "Object for holding a batch of data with mask during training."
    def __init__(self, src, trg=None, pad=0):
        self.src = src

```

```

self.src_mask = (src != pad).unsqueeze(-2)
if trg is not None:
    self.trg = trg[:, :-1]
    self.trg_y = trg[:, 1:]
    self.trg_mask = \
        self.make_std_mask(self.trg, pad)
    self.ntokens = (self.trg_y != pad).data.sum()

@staticmethod
def make_std_mask(tgt, pad):
    "Create a mask to hide padding and future words."
    tgt_mask = (tgt != pad).unsqueeze(-2)
    tgt_mask = tgt_mask & Variable(
        subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
    return tgt_mask

```

接下来，我们创建一个通用的训练和得分函数来跟踪损失。我们传入一个通用的损失计算函数，它也处理参数更新。

14. 训练循环 (Training Loop)

```

def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y, batch.ntokens)
        total_loss += loss
        total_tokens += batch.ntokens
        tokens += batch.ntokens
        if i % 50 == 1:
            elapsed = time.time() - start
            print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %
                  (i, loss / batch.ntokens, tokens / elapsed))
            start = time.time()
            tokens = 0
    return total_loss / total_tokens

```

15. 训练数据和批处理 (Training Data and Batching)

我们使用标准 WMT 2014 英语-德语数据集进行了训练，该数据集包含大约 450 万个句子对。使用字节对的编码方法对句子进行编码，该编码具有大约 37000 个词的共享源-目标词汇表。对于英语-法语，我们使用了 WMT 2014 英语-法语数据集，该数据集由 36M 个句子组成，并将词分成 32000 个词片 (Word-piece) 的词汇表。

句子对按照近似的序列长度进行批处理。每个训练批包含一组句子对，包含大约 25000 个源词和 25000 个目标词。

我们将使用 torch text 来创建批次。下面更详细地讨论实现过程。我们在 torchtext 的一个函数中创建批次，确保填充到最大批训练长度的大小不超过阈值（如果我们有 8 个 GPU，则阈值为 25000）。

```
global max_src_in_batch, max_tgt_in_batch
def batch_size_fn(new, count, sofar):
    "Keep augmenting batch and calculate total number of tokens + padding."
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch, len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch, len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

16. 硬件和训练进度 (Hardware and Schedule)

我们在一台配备 8 个 NVIDIA P100 GPU 的机器上训练我们的模型。对于使用本文所述的超参数的基本模型，每个训练单步大约需要 0.4 秒。我们对基础模型进行了总共 100,000 步或 12 小时的训练。对于我们的大型模型，每个训练单步时间为 1.0 秒。大型模型通常需要训练 300,000 步 (3.5 天)。

17. 优化器 (Optimizer)

我们选择 Adam^①作为优化器，其参数为 $\beta_1 = 0.9$, $\beta_2 = 0.98$ 和 $\epsilon = 10^{-9}$. 根据以下公式，我们在训练过程中改变了学习率：

$$\text{lrate} = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (3.5)$$

在预热中随步数线性地增加学习速率，并且此后与步数的反平方根成比例地减小它。我们设置预热步数为 4000。

注意：这部分非常重要，需要这种设置训练模型。

^① <https://arxiv.org/abs/1412.6980>

```

class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
        "Implement 'lrate' above"
        if step is None:
            step = self._step
        return self.factor * \
               (self.model_size ** (-0.5)) * \
               min(step ** (-0.5), step * self.warmup ** (-1.5))

    def get_std_opt(model):
        return NoamOpt(model.src_embed[0].d_model, 2, 4000,
                      torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))

```

当前模型在不同模型大小和超参数的情况下示例。

```

# Three settings of the lrate hyperparameters.
opts = [NoamOpt(512, 1, 4000, None),
        NoamOpt(512, 1, 8000, None),
        NoamOpt(256, 1, 4000, None)]
plt.plot(np.arange(1, 20000), [[opt.rate(i) for opt in opts] for i in range(1, 20000)])
plt.legend(["512:4000", "512:8000", "256:4000"])
None

```

18. 正则化 (Regularization)

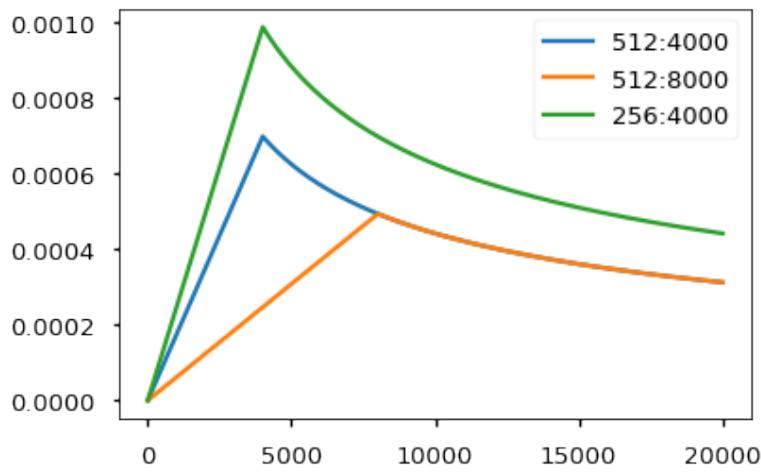


图 3.8 当前模型在不同模型大小和超参数的情况下示例

19. 标签平滑 (Label Smoothing)

在训练期间，我们采用了值 $\epsilon_{ls} = 0.1$ ^① 的标签平滑。这种做法提高了困惑度，因为模型变得更加不确定，但提高了准确性和 BLEU 分数。

我们使用 KL div loss 实现标签平滑。相比使用独热目标分布，我们创建一个分布，其包含正确单词的置信度和整个词汇表中分布的其余平滑项。

```
class LabelSmoothing(nn.Module):
    "Implement label smoothing."
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        #self.criterion = nn.KLDivLoss(size_average=False)reduction='sum'
        self.criterion = nn.KLDivLoss(reduction='sum')
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None

    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
```

① <https://arxiv.org/abs/1512.00567>

```

    self.true_dist = true_dist
    return self.criterion(x, Variable(true_dist, requires_grad=False))

```

在这里，我们可以看到标签平滑的示例。

```

# Example of label smoothing.
crit = LabelSmoothing(5, 0, 0.4)
predict = torch.FloatTensor([[0, 0.2, 0.7, 0.1, 0],
                            [0, 0.2, 0.7, 0.1, 0],
                            [0, 0.2, 0.7, 0.1, 0]])
v = crit(Variable(predict.log()),
          Variable(torch.LongTensor([2, 1, 0])))

# Show the target distributions expected by the system.
plt.imshow(crit.true_dist)
None

```

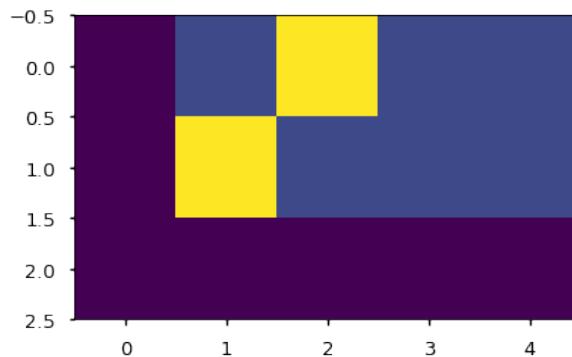


图 3.9 标签平滑的示例

如果对给定的选择非常有信心，标签平滑实际上会开始惩罚模型。

```

crit = LabelSmoothing(5, 0, 0.1)
def loss(x):
    d = x + 3 * 1
    predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 / d],])
    #print(predict)
    #return crit(Variable(predict.log()), Variable(torch.LongTensor([1]))).data[0]
    return crit(Variable(predict.log()), Variable(torch.LongTensor([1]))).item()
plt.plot(np.arange(1, 100), [loss(x) for x in range(1, 100)])
None

```

20. 第一个例子 (A First Example)

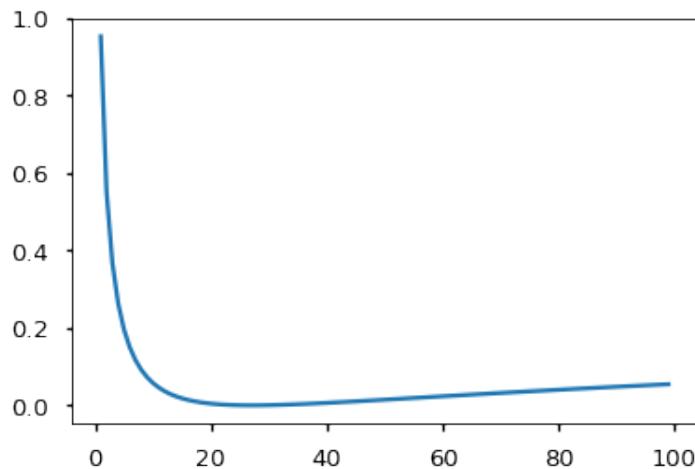


图 3.10 标签平滑实际上会开始惩罚模型

我们可以先尝试一个简单的复制任务。给定来自小词汇表的随机输入符号集，目标是生成那些相同的符号。

21. 数据生成 (Synthetic Data)

```
def data_gen(V, batch, nbatches):
    "Generate random data for a src-tgt copy task."
    for i in range(nbatches):
        data = torch.from_numpy(np.random.randint(1, V, size=(batch, 10)))
        data[:, 0] = 1
        src = Variable(data, requires_grad=False)
        tgt = Variable(data, requires_grad=False)
        yield Batch(src, tgt, 0)
```

22. 损失计算 (Loss Computation)

```
class SimpleLossCompute:
    "A simple loss compute and train function."
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt

    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                              y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
```

```

        self.opt.step()
        self.opt.optimizer.zero_grad()
    #return loss.data[0] * norm
    return loss.item() * norm
#.item()

```

23. 贪心解码 (Greedy Decoding)

```

# Train the simple copy task.

V = 11
criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
model = make_model(V, V, N=2)
model_opt = NoamOpt(model.src_embed[0].d_model, 1, 400,
                     torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))

for epoch in range(10):
    model.train()
    run_epoch(data_gen(V, 30, 20), model, SimpleLossCompute(model.generator, criterion, model_opt))
    model.eval()
    print(run_epoch(data_gen(V, 30, 5), model,
                    SimpleLossCompute(model.generator, criterion, None)))

```

该作业实际输出如下：

```

Epoch Step: 1 Loss: 2.952445 Tokens per Sec: 576.319092
Epoch Step: 1 Loss: 1.998418 Tokens per Sec: 698.345398
tensor(1.9607)
Epoch Step: 1 Loss: 1.944340 Tokens per Sec: 607.740173
Epoch Step: 1 Loss: 1.703401 Tokens per Sec: 697.352722
tensor(1.6663)
Epoch Step: 1 Loss: 1.836476 Tokens per Sec: 608.583740
Epoch Step: 1 Loss: 1.518878 Tokens per Sec: 698.963074
tensor(1.4429)
Epoch Step: 1 Loss: 1.751896 Tokens per Sec: 611.272522
Epoch Step: 1 Loss: 1.445300 Tokens per Sec: 699.340576
tensor(1.4257)
Epoch Step: 1 Loss: 1.457413 Tokens per Sec: 609.735352
Epoch Step: 1 Loss: 0.952619 Tokens per Sec: 700.159241
tensor(0.9635)
Epoch Step: 1 Loss: 1.194762 Tokens per Sec: 603.452271
Epoch Step: 1 Loss: 0.720170 Tokens per Sec: 699.754761
tensor(0.6950)
Epoch Step: 1 Loss: 1.114729 Tokens per Sec: 610.193848

```

```

Epoch Step: 1 Loss: 0.482503 Tokens per Sec: 694.832153
tensor(0.4766)
Epoch Step: 1 Loss: 0.910544 Tokens per Sec: 605.345947
Epoch Step: 1 Loss: 0.245647 Tokens per Sec: 699.077209
tensor(0.3241)
Epoch Step: 1 Loss: 0.579662 Tokens per Sec: 606.469055
Epoch Step: 1 Loss: 0.210620 Tokens per Sec: 678.093811
tensor(0.2678)
Epoch Step: 1 Loss: 0.556862 Tokens per Sec: 575.055847
Epoch Step: 1 Loss: 0.189768 Tokens per Sec: 690.105469
tensor(0.2050)

```

为简单起见，此代码使用贪心解码来预测翻译。

```

def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask,
                           Variable(ys),
                           Variable(subsequent_mask(ys.size(1))
                                   .type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
        #next_word = next_word.data[0]
        next_word = next_word.item()
        ys = torch.cat([ys,
                       torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys

model.eval()
src = Variable(torch.LongTensor([[1,2,3,4,5,6,7,8,9,10]]))
src_mask = Variable(torch.ones(1, 1, 10))
print(greedy_decode(model, src, src_mask, max_len=10, start_symbol=1))

```

24. 真实示例 (A Real World Example)

现在我们通过 IWSLT 德语-英语翻译任务介绍一个真实示例。该任务比上文提及的 WMT 任务小得多，但它说明了整个系统。我们还展示了如何使用多个 GPU 处理加速其训练。

本作业至第 24 节真实示例 (A Real World Example) 后，因为套件与本版本错误，选择用注解说明。

```
#!pip install torchtext spacy
#!python -m spacy download en
#!python -m spacy download de
```

25. 数据加载 (Data Loading)

我们将使用 torchtext 和 spacy 加载数据集以进行词语切分。

```
# For data loading.
from torchtext import data, datasets

if True:
    import spacy
    spacy_de = spacy.load('de')
    spacy_en = spacy.load('en')

    def tokenize_de(text):
        return [tok.text for tok in spacy_de.tokenizer(text)]

    def tokenize_en(text):
        return [tok.text for tok in spacy_en.tokenizer(text)]

    BOS_WORD = '<s>'
    EOS_WORD = '</s>'
    BLANK_WORD = "<blank>"
    SRC = data.Field(tokenize=tokenize_de, pad_token=BLANK_WORD)
    TGT = data.Field(tokenize=tokenize_en, init_token = BOS_WORD,
                      eos_token = EOS_WORD, pad_token=BLANK_WORD)

    MAX_LEN = 100
    train, val, test = datasets.IWSLT.splits(
        exts=('.de', '.en'), fields=(SRC, TGT),
        filter_pred=lambda x: len(vars(x)['src']) <= MAX_LEN and
        len(vars(x)['trg']) <= MAX_LEN)
    MIN_FREQ = 2
    SRC.build_vocab(train.src, min_freq=MIN_FREQ)
    TGT.build_vocab(train.trg, min_freq=MIN_FREQ)
```

批训练对于速度来说很重要。我们希望批次分割非常均匀并且填充最少。要做到这一点，我们必须修改 torchtext 默认的批处理函数。这部分代码修补其默认批处理函数，以确保我们搜索足够多的句子以构建紧密批处理。

26. 迭代器 (Iterators)

```

class MyIterator(data.Iterator):
    def create_batches(self):
        if self.train:
            def pool(d, random_shuffler):
                for p in data.batch(d, self.batch_size * 100):
                    p_batch = data.batch(
                        sorted(p, key=self.sort_key),
                        self.batch_size, self.batch_size_fn)
                    for b in random_shuffler(list(p_batch)):
                        yield b
            self.batches = pool(self.data(), self.random_shuffler)

        else:
            self.batches = []
            for b in data.batch(self.data(), self.batch_size,
                                self.batch_size_fn):
                self.batches.append(sorted(b, key=self.sort_key))

    def rebatch(pad_idx, batch):
        "Fix order in torchtext to match ours"
        src, trg = batch.src.transpose(0, 1), batch.trg.transpose(0, 1)
        return Batch(src, trg, pad_idx)

```

27. 多 GPU 训练 (Multi-GPU Training)

最后为了真正地快速训练，我们将使用多个 GPU。这部分代码实现了多 GPU 字生成。它不是 Transformer 特有的，所以我不会详细介绍。其思想是将训练时的单词生成分成块，以便在许多不同的 GPU 上并行处理。我们使用 PyTorch 并行原语来做到这一点：

- 复制 - 将模块拆分到不同的 GPU 上
- 分散 - 将批次拆分到不同的 GPU 上
- 并行应用 - 在不同 GPU 上将模块应用于批处理
- 聚集 - 将分散的数据聚集到一个 GPU 上
- nn.DataParallel - 一个特殊的模块包装器，在评估之前调用它们。

```

# Skip if not interested in multigpu.
class MultiGPULossCompute:
    "A multi-gpu loss compute and train function."
    def __init__(self, generator, criterion, devices, opt=None, chunk_size=5):
        # Send out to different gpus.
        self.generator = generator

```

```

self.criterion = nn.parallel.replicate(criterion,
                                       devices=devices)
self.opt = opt
self.devices = devices
self.chunk_size = chunk_size

def __call__(self, out, targets, normalize):
    total = 0.0
    generator = nn.parallel.replicate(self.generator,
                                       devices=self.devices)
    out_scatter = nn.parallel.scatter(out,
                                      target_gpus=self.devices)
    out_grad = [[] for _ in out_scatter]
    targets = nn.parallel.scatter(targets,
                                  target_gpus=self.devices)

    # Divide generating into chunks.
    chunk_size = self.chunk_size
    for i in range(0, out_scatter[0].size(1), chunk_size):
        # Predict distributions
        out_column = [[Variable(o[:, i:i+chunk_size].data,
                               requires_grad=self.opt is not None)]
                      for o in out_scatter]
        gen = nn.parallel.parallel_apply(generator, out_column)

        # Compute loss.
        y = [(g.contiguous().view(-1, g.size(-1)),
              t[:, i:i+chunk_size].contiguous().view(-1))
              for g, t in zip(gen, targets)]
        loss = nn.parallel.parallel_apply(self.criterion, y)

        # Sum and normalize loss
        l = nn.parallel.gather(loss,
                               target_device=self.devices[0])
        l = l.sum()[0] / normalize
        total += l.data[0]

        # Backprop loss to output of transformer
        if self.opt is not None:
            l.backward()
            for j, l in enumerate(loss):
                out_grad[j].append(out_column[j][0].grad.data.clone())

    # Backprop all loss through transformer.

```

```

if self.opt is not None:
    out_grad = [Variable(torch.cat(og, dim=1)) for og in out_grad]
    o1 = out
    o2 = nn.parallel.gather(out_grad,
                           target_device=self.devices[0])
    o1.backward(gradient=o2)
    self.opt.step()
    self.opt.optimizer.zero_grad()
return total * normalize

```

现在我们创建模型，损失函数，优化器，数据迭代器和并行化。

```

# GPUs to use
devices = [0, 1, 2, 3]

if True:
    pad_idx = TGT.vocab.stoi["<blank>"]
    model = make_model(len(SRC.vocab), len(TGT.vocab), N=6)
    model.cuda()
    criterion = LabelSmoothing(size=len(TGT.vocab), padding_idx=pad_idx, smoothing=0.1)
    criterion.cuda()
    BATCH_SIZE = 12000
    train_iter = MyIterator(train, batch_size=BATCH_SIZE, device=0,
                           repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                           batch_size_fn=batch_size_fn, train=True)
    valid_iter = MyIterator(val, batch_size=BATCH_SIZE, device=0,
                           repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                           batch_size_fn=batch_size_fn, train=False)
    model_par = nn.DataParallel(model, device_ids=devices)
None

```

现在我们训练模型。我将稍微使用预热步骤，但其他一切都使用默认参数。在具有 4 个 Tesla V100 GPU 的 AWS p3.8xlarge 机器上，每秒运行约 27,000 个词，批训练大小大小为 12,000。

28. 训练系统 (Training the System)

```

#!wget https://s3.amazonaws.com/opennmt-models/iwslt.pt

if False:
    model_opt = NoamOpt(model.src_embed[0].d_model, 1, 2000,
                        torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
    for epoch in range(10):
        model_par.train()

```

```

        run_epoch((rebatch(pad_idx, b) for b in train_iter),
                   model_par,
                   MultiGPULossCompute(model.generator, criterion,
                                         devices=devices, opt=model_opt))

    model_par.eval()
    loss = run_epoch((rebatch(pad_idx, b) for b in valid_iter),
                      model_par,
                      MultiGPULossCompute(model.generator, criterion,
                                         devices=devices, opt=None))
    print(loss)
else:
    model = torch.load("iwslt.pt")

```

一旦训练完成，我们可以解码模型以产生一组翻译。在这里，我们只需翻译验证集中的第一个句子。此数据集非常小，因此使用贪婪搜索的翻译相当准确。

```

for i, batch in enumerate(valid_iter):
    src = batch.src.transpose(0, 1)[:1]
    src_mask = (src != SRC.vocab.stoi["<blank>"]).unsqueeze(-2)
    out = greedy_decode(model, src, src_mask,
                         max_len=60, start_symbol=TGT.vocab.stoi["<s>"])
    print("Translation:", end="\t")
    for i in range(1, out.size(1)):
        sym = TGT.vocab.itos[out[0, i]]
        if sym == "</s>": break
        print(sym, end=" ")
    print()
    print("Target:", end="\t")
    for i in range(1, batch.trg.size(0)):
        sym = TGT.vocab.itos[batch.trg.data[i, 0]]
        if sym == "</s>": break
        print(sym, end=" ")
    print()
break

```

29. 附加组件: BPE, 搜索, 平均 (Additional Components: BPE, Search, Averaging)

所以这主要涵盖了 Transformer 模型本身。有四个方面我们没有明确涵盖。我们还实现了所有这些附加功能 OpenNMT-py.

1) 字节对编码/字片 (Word-piece): 我们可以使用库来首先将数据预处理为子字单元。参见 Rico Sennrich 的 subword-nmt 实现。这些模型将训练数据转换为如下所示:

Die Protokoll datei kann heimlich per E - Mail oder FTP an einen bestimmte n Empfänger gesendet werden .

2) 共享嵌入：当使用具有共享词汇表的 BPE 时，我们可以在源/目标/生成器之间共享相同的权重向量，详细见。要将其添加到模型，只需执行以下操作：

```
if False:
    model.src_embed[0].lut.weight = model.tgt_embeddings[0].lut.weight
    model.generator.lut.weight = model.tgt_embed[0].lut.weight
```

3) 集束搜索：这里展开说有点太复杂了。PyTorch 版本的实现可以参考 OpenNMT-py。4) 模型平均：这篇文章平均最后 k 个检查点以创建一个集合效果。如果我们有一堆模型，我们可以在事后这样做：

```
def average(model, models):
    "Average models into model"
    for ps in zip(*[m.params() for m in [model] + models]):
        p[0].copy_(torch.sum(*ps[1:]) / len(ps[1:]))
```

30. 结果 (Results)

在 WMT 2014 英语-德语翻译任务中，大型 Transformer 模型（表中的 Transformer（大））优于先前报告的最佳模型（包括集成的模型）超过 2.0 BLEU，建立了一个新的最先进 BLEU 得分为 28.4。该模型的配置列于表 3 的底部。在 8 个 P100 GPU 的机器上，训练需要需要 3.5 天。甚至我们的基础模型也超过了之前发布的所有模型和集成，而且只占培训成本的一小部分。

在 WMT 2014 英语-法语翻译任务中，我们的大型模型获得了 41.0 的 BLEU 分数，优于以前发布的所有单一模型，不到以前最先进技术培训成本的 1/4 模型。使用英语到法语训练的 Transformer（大）模型使用 dropout 概率 $P_{dorp} = 0.1$ ，而不是 0.3。

```
!wget https://s3.amazonaws.com/opennmt-models/en-de-model.pt
```

```
model, SRC, TGT = torch.load("en-de-model.pt")
```

```
model.eval()
sent = "The log file can be sent secret ly with email or FTP to a specified receiver".split()
src = torch.LongTensor([[SRC.stoi[w] for w in sent]])
src = Variable(src)
src_mask = (src != SRC.stoi["<blank>"]).unsqueeze(-2)
out = greedy_decode(model, src, src_mask,
                     max_len=60, start_symbol=TGT.stoi["<s>"])
print("Translation:", end="\t")
trans = "<s> "
```

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.8	$2.3 \cdot 10^{19}$	

图 3.11 WMT 2014 英语-德语翻译任

```

for i in range(1, out.size(1)):
    sym = TGT.itos[out[0, i]]
    if sym == "</s>": break
    trans += sym + " "
print(trans)

```

31. 注意力可视化 (Attention Visualization)

即使使用贪婪的解码器，翻译看起来也不错。我们可以进一步想象它，看看每一层注意力发生了什么。

```

tgt_sent = trans.split()
def draw(data, x, y, ax):
    seaborn.heatmap(data,
                    xticklabels=x, square=True, yticklabels=y, vmin=0.0, vmax=1.0,
                    cbar=False, ax=ax)

for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Encoder Layer", layer+1)
    for h in range(4):
        draw(model.encoder.layers[layer].self_attn.attn[0, h].data,
              sent, sent if h == 0 else [], ax=axs[h])
    plt.show()

for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Decoder Self Layer", layer+1)
    for h in range(4):

```

```

draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(tgt_sent)],
     tgt_sent, tgt_sent if h ==0 else [], ax=axs[h])
plt.show()
print("Decoder Src Layer", layer+1)
fig, axs = plt.subplots(1,4, figsize=(20, 10))
for h in range(4):
    draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(sent)],
         sent, tgt_sent if h ==0 else [], ax=axs[h])
plt.show()

```

32. 结论 (Conclusion)

```

@inproceedings{opennmt,
author      = {Guillaume Klein and
                 Yoon Kim and
                 Yuntian Deng and
                 Jean Senellart and
                 Alexander M. Rush},
title       = {OpenNMT: Open-Source Toolkit for Neural Machine Translation},
booktitle   = {Proc. ACL},
year        = {2017},
url         = {https://doi.org/10.18653/v1/P17-4012},
doi         = {10.18653/v1/P17-4012}
}

```

3.2.2 Chatbot Transformer

该范例为 fawazsammani/chatbot-transformer 的 GitHub 专案，该专案使用 Cornell Movie Dialog Corpus 资料，进行 Chatbot using Transformers 的实作，其训练结果如下。

3.3 使用技术

在此次作业有试验过不同的技术实践，比如虚拟化容器的演练。此次作业有尝试演练 Docker 与 LaTeX 来改善研究工作流程。

3.3.1 LaTeX

LaTex 为 Overleaf 的模板，其模板贡献则是根据 iofu728-pkuthss 的版本进行作业。

3.3.2 Docker

在此使用 Anaconda 的 Docker 容器，并用共同挂载目录进行流程。

1. 在终端窗口中，运行此命令以显示可用图像列表：

```
In [32]:  
for epoch in range(epochs):  
    train(train_loader, transformer, criterion, epoch)  
  
    state = {'epoch': epoch, 'transformer': transformer, 'transformer_optimizer': transformer_optimizer}  
    # torch.save(state, 'checkpoint_{} + str(epoch) + '.pth.tar')  
    torch.save(state, 'checkpoint.pth.tar')  
  
Epoch [0][0/2217] Loss: 8.673  
Epoch [0][100/2217] Loss: 7.910  
Epoch [0][200/2217] Loss: 7.188  
Epoch [0][300/2217] Loss: 6.647  
Epoch [0][400/2217] Loss: 6.294  
Epoch [0][500/2217] Loss: 6.048  
Epoch [0][600/2217] Loss: 5.862  
Epoch [0][700/2217] Loss: 5.717  
Epoch [0][800/2217] Loss: 5.599  
Epoch [0][900/2217] Loss: 5.504  
Epoch [0][1000/2217] Loss: 5.422  
Epoch [0][1100/2217] Loss: 5.351  
Epoch [0][1200/2217] Loss: 5.289  
Epoch [0][1300/2217] Loss: 5.236  
Epoch [0][1400/2217] Loss: 5.190  
Epoch [0][1500/2217] Loss: 5.147  
Epoch [0][1600/2217] Loss: 5.110  
Epoch [0][1700/2217] Loss: 5.076  
Epoch [0][1800/2217] Loss: 5.044  
Epoch [0][1900/2217] Loss: 5.016  
Epoch [0][2000/2217] Loss: 4.990  
Epoch [0][2100/2217] Loss: 4.966  
Epoch [0][2200/2217] Loss: 4.944
```

图 3.12 Chatbot Transformer 训练

```
docker search continuumio
```

2. 拉取所需的图像

```
docker pull continuumio/miniconda3
```

3. 使用图像创建容器

```
docker run -t -i continuumio/miniconda3 /bin/bash
```

4. 若想在本地挂载目录则用 -v , 假设本机使用 Windows 在目录 docker-save , 而 /share 为 Docker 的目录。

```
# docker run -it -v D:\docker-save:/share continuumio/miniconda3 /bin/bash  
# docker run -it -v [本地端目录]:[Docker 目录] [Docker 映像的名称] /bin/bash
```

5. 安裝和启动 Jupyter Notebook

请在浏览器中打开 <http://localhost:8888>

```
docker run -i -t -p 8888:8888 continuumio/miniconda3 /bin/bash -c "\  
conda install jupyter -y --quiet && \  
jupyter notebook --no-browser --port=8888"
```

```
mkdir -p /opt/notebooks && \
jupyter notebook \
--notebook-dir=/opt/notebooks --ip='*' --port=8888 \
--no-browser --allow-root"
```

6. 根据前述设定好路径，则合理的配置如下。

```
docker run -i -t -v D:\docker-save:/opt/notebooks -p 8888:8888 continuumio/miniconda3 /bin/bash -c "\$(
conda install jupyter -y --quiet && \
mkdir -p /opt/notebooks && \
jupyter notebook \
--notebook-dir=/opt/notebooks --ip='*' --port=8888 \
--no-browser --allow-root")"
```

7. 管理容器

docker ps 看所有映像档，而 -a 列出包含未运行的

```
docker ps
docker ps -a
```


第四章 Transformer

本章目标在于说明 Transformer 的概念，同时说明在其 Transformer 的文献上的工作进行说明与复现工作。本作业其专案为 kancheng/kan-cs-report-in-2021，程式码则可于 kan-cs-report-in-2021/AI/pytorch-transformer 中找到，实验设备为 MacBook Pro (Retina, 15-inch, Mid 2014) 和 Acer Aspire R7 与 Google Colab。

4.1 Transformer 理解

此节目标在于说明 Transformer 概念理解，根据原本的技术文件与论文的内容进行说明和理解进行整理。其内容包含了原本论文的该概念描述，同时也对实际原理进行说明。

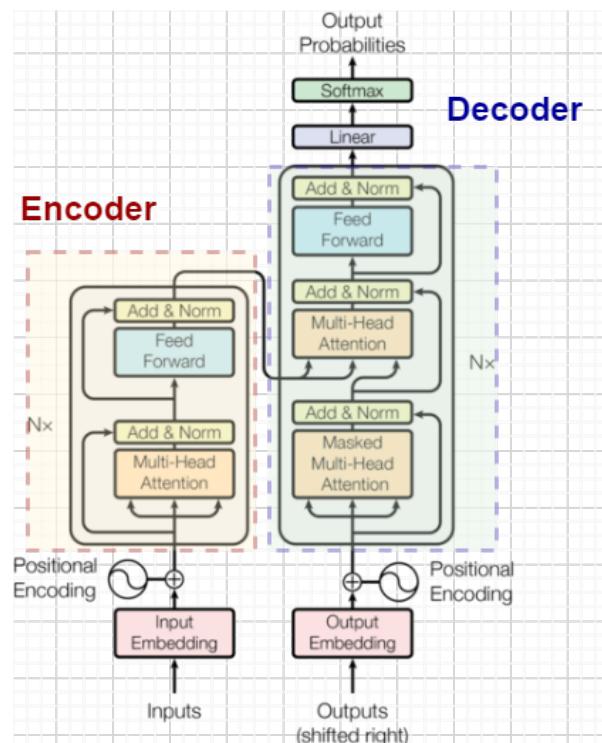


图 4.1 直观架构

4.1.1 原理简述

单纯从 Google 於 2017 所發表的 Attention is All You Need 中的 Transformer，其架构直观来看可以分为 Encoder 和 Decoder 两大部分。就与过往的传统 RNN、CNN 模型

不同在于，工作是由 Attention 机制来进行实现，而且由于 Encoder 端是并行计算，具有训练时间缩短的优势。

在此简述其模型的优势，从过往的研究来看，传统 seq2seq 最大的问题在于该模型会将 Encoder 端的所有信息压缩到一个固定长度的向量里面，同时将其作为 Decoder 端首个隐藏状态的输入，来预测 Decoder 端第一个单词 token 的隐藏状态。当输入序列比较长的时候，此方法很会损失 Encoder 端那方的很多信息，而且将大量的把该固定向量送入 Decoder 端，Decoder 端那方很有可能不能够关注到其想要我们想要关注的信息。同时模型计算不可并行，其计算隐层状态 h_t 依赖于 h_{t-1} 以及状态 t 时刻的输入，因此需要耗费大量时间。而 Transformer 优点则是在于 Transformer 架构完全依赖于 Attention 机制，解决了输入输出的长期依赖问题，同时拥有并行计算的能力，大大减少了计算资源的消耗。self-attention 模块则会让源序列和目标序列首先“自关联”起来，在此情况下源序列和目标序列自身的 embedding 表示所蕴含的信息会更加丰富，而且后续的 FFN 层也增强了模型的表达能力。而 Muti-Head Attention 模块使得 Encoder 端拥有并行计算的能力

4.2 Transformer 研究

本节目标示根据近期研究进行追溯，该节先是对 Transformers in vision: A survey^[28] 进行概括，同时找出六篇研究进行工作调研与总结并进行整理，所以该节使用文献如下条列：

- Transformers in Vision: A Survey
- Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks
- Deep amortized clustering
- Non-local Neural Networks
- Axial-DeepLab: Stand-Alone Axial-Attention for Panoptic Segmentation
- Taming Transformers for High-Resolution Image Synthesis
- Video Action Transformer Network

4.2.1 工作追溯

Transformers in vision: A survey 该篇为视觉中的 Transformer 和自注意力综述涵盖了 transformer 在视觉领域的广泛应用，包括如图像分类，目标检测，动作识别和分割等流行的识别任务，生成模型，如视觉问题解答和视觉推理等多模式任务，又如活动识别，视频预测等视频处理，再如图像超分辨率和彩色化的 low-level 视觉和点云分类

和分割等 3D 分析。

该篇研究先说明 Transformer 模型与研究的重要性，并说明 Transformer 在自然语言任务中获得惊人结果，并引起了计算机视觉领域的兴趣跟注意得原因，研究它们在计算机视觉问题中的应用，Transformer 与类似如长短期记忆 (LSTM) 等的循环网络相比，Transformers 的显着优势之一是能够对输入序列元素之间的长期依赖关系进行建模，并支持序列的并行处理。Transformer 与卷积网络不同，Transformer 的设计需要最小的归纳偏差，自然适合作为集合函数。此外 Transformers 的简单设计允许使用类似的处理块处理多种模态（例如，图像、视频、文本和语音），并展示了对超大容量网络和庞大数数据集的出色可扩展性。这些优势使使用 Transformer 网络的许多视觉任务取得了令人兴奋的进展。本次调查旨在全面概述计算机视觉学科中的 Transformer 模型。研究者首先介绍 Transformers 成功背后的基本概念，即自我注意、大规模预训练和双向特征编码。然后，我们介绍了转换器在视觉中的广泛应用，包括如图像分类、对象检测、动作识别和分割等流行的识别任务领域，生成建模领域，又如视觉问答、视觉推理和视觉基础等多模态任务，再如活动识别、视频预测的视频处理领域，或者是图像超分辨率、图像增强和着色的 low-level vision 领域和点云分类和分割的 3D 分析。研究者比较了流行技术在架构设计和实验价值方面的各自优势和局限性。最后该研究对开放的研究方向和未来可能的工作进行了分析，此研究希望这项努力将进一步激发社区的兴趣，以解决当前在计算机视觉中应用变压器模型所面临的挑战。最后将该篇研究的整理用简略的条例表示如下：

1. 用于图像识别的 Transformer

- Non-local Neural Networks
- Criss-cross Attention
- Stand-alone Self-Attention
- Local Relation Networks
- Attention Augmented Convolutional Networks
- Vectorized Self-Attention
- Vision Transformer
- Data-efficient Image Transformers

2. 用于目标检测的 Transformer

- DETR
- Deformable - DETR

3. 用于分割的 Transformer

- Axial-attention for Panoptic Segmentation

4. 用于图像生成的 Transformer

- Image GPT
- Image Transformer
- High-resolution Image Synthesis
- SceneFormer

5. 用于 low-level 视觉的 Transformer

- Transformers for super-resolution
- Transformers for Image Enhancement Tasks
- Colorization Transformer

6. 用于多模态任务的 Transformer

- ViLBERT: Vision and Language BERT
- LXMERT
- VisualBERT
- VL-BERT
- Unicoder-VL
- UNITER
- Oscar: Object-Semantics Aligned Pre-training
- Vokenization
- Vision-and-Language Navigation

7. 用于视频理解的 Transformer

- VideoBERT: Joint Video and Language Modeling
- Parameter Efficient Multi-modal Transformers
- Video Action Transformer
- Skeleton-based Action Recognition

8. 用于 Low-shot 学习的 Transformer

- Cross-transformer
- FEAT: Few-shot Embedding Adaptation

9. 用于聚类的 Transformer

- Set Transformers

10. 用于 3D 分析的 Transformer

- Point Transformer
- Point-cloud Transformer
- Pose and Mesh Reconstruction

Transformers in Vision: A Survey

Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir,
Fahad Shahbaz Khan, and Mubarak Shah

Abstract—Astounding results from Transformer models on natural language tasks have intrigued the vision community to study their application to computer vision problems. Among their salient benefits, Transformers enable modeling long dependencies between input sequence elements and support parallel processing of sequence as compared to recurrent networks e.g., Long short-term memory (LSTM). Different from convolutional networks, Transformers require minimal inductive biases for their design and are naturally suited as set-functions. Furthermore, the straightforward design of Transformers allows processing multiple modalities (e.g., images, videos, text and speech) using similar processing blocks and demonstrates excellent scalability to very large capacity networks and huge datasets. These strengths have led to exciting progress on a number of vision tasks using Transformer networks. This survey aims to provide a comprehensive overview of the Transformer models in the computer vision discipline. We start with an introduction to fundamental concepts behind the success of Transformers i.e., self-attention, large-scale pre-training, and bidirectional feature encoding. We then cover extensive applications of transformers in vision including popular recognition tasks (e.g., image classification, object detection, action recognition, and segmentation), generative modeling, multi-modal tasks (e.g., visual-question answering, visual reasoning, and visual grounding), video processing (e.g., activity recognition, video forecasting), low-level vision (e.g., image super-resolution, image enhancement, and colorization) and 3D analysis (e.g., point cloud classification and segmentation). We compare the respective advantages and limitations of popular techniques both in terms of architectural design and their experimental value. Finally, we provide an analysis on open research directions and possible future works. We hope this effort will ignite further interest in the community to solve current challenges towards the application of transformer models in computer vision.

Index Terms—Self-attention, transformers, bidirectional encoders, deep neural networks, convolutional networks, self-supervision.

图 4.2 Transformers in vision: A survey

4.2.2 Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks

其原因在于许多机器学习任务，例如多实例学习、3D 形状识别和小样本图像分类，都是在实例集上定义的，而由于此类问题的解决方案不依赖于集合元素的顺序，用于解决这些问题的模型应该是置换不变，所以该研究提出了一个基于注意力的神经网络模块，即 Set Transformer，专门设计用于对输入集中元素之间的交互进行建模。提出的模型由一个编码器和一个译码器组成，两者都依赖于注意力机制，而为了降低计算复杂度，研究者引入了一种注意力机制，其灵感来自于从稀疏高斯过程文献中引入点方法。它将自注意力的计算时间从集合中元素的数量的二次减少到线性。该研究表明的模型在理论上具有吸引力，并且在一系列任务上对其进行了评估，与最近的集合结构数据方法相比，表现出更高的性能。

该研究说明学习表征已被证明是深度学习及其许多成功案例的基本问题，而深度学习解决的大多数问题都是基于实例的，并采用将固定维输入张量映像到其相应目标值的形式 (Krizhevsky et al., 2012; Graves et al., 2013)，而对于某些应用程序，研究者认为会需要处理集合结构化数据。再此研究者说明多实例学习 (Dietterich et al., 1997; Maron & Lozano-Perez , 1998) 是这种集合输入问题的一个例子，其中给定一组实例作为输入，相应的目标是整套。其他问题，例如 3D 形状识别 (Wu et al., 2015; Shi et al., 2015; Su et al., 2015; Charles et al., 2017) 、序列排序 (Vinyals et al., 2016) 以及各种集合操作 (Muandet et al., 2012; Oliva et al., 2013; Edwards & Storkey, 2017; Zaheer et al., 2017) 也可以被视为集合输入问题，同时许多元学习 (Thrun & Pratt, 1998; Schmidhuber, 1987) 问题使用不同但相关的任务进行学习，也可以视为 setinput 任务，其中输入集对应于单个任务的训练数据集。

例如少样本图像分类 (Finn et al., 2017; Snell et al., 2017; Lee & Choi, 2018) 通过使用一组支持图像构建分类器来运行，而该分类器使用查询图像进行评估，集合输入问题的模型应该满足两个关键要求。首先第一个要求在于，它应该是排列不变的——模型的输出在输入集中元素的任何排列下都不应该改变，其次此类模型应该能够处理任何大小的输入集。虽然这些要求源于集合的定义，但它们在基于神经网络的模型中并不容易满足：经典的前馈神经网络违反了这两个要求，并且 RNN 对输入顺序很敏感，而近来 Edwards & Storkey (2017) and Zaheer et al. (2017) 提出了满足这两个标准的神经网络架构，我们称之为集合池方法，在此模型中在集合中的每个元素首先被独立地馈送到接受固定大小输入的前馈神经网络。然后使用池化操作 (均值 (mean), 总和 (sum), 最大值 (max) or 类似 (similar)). 聚合得到的特征空间嵌入，而通过对聚合嵌入的进一步非线性处理获得最终输出。这种非常简单的架构满足上述两个要求，更重要的是它被证明是任何集合函数的通用逼近器 (Zaheer et al., 2017)。由于此类特性可以根据黑盒方式学习输入集与其目标输出之间的复杂映像，就像前馈或循环神经网络一样，尽管这种集合池方法在理论上很有吸引力，但研究者仍不清楚我们是否可以仅使用基于实例的特征提取器和简单的池操作来很好地近似复杂的映射。由于集合中的每个元素都在集合池操作中独立处理，因此必须丢弃一些有关元素之间交互的信息，这可能会使一些问题不必要地难以解决。考虑摊销聚类的问题，研究者会想学习从输入点集到该集内点簇中心的参数映射，即使面对于二维空间中的玩具数据集，这也不是一个容易的问题。

主要的困难在于参数映像必须在对解释模式进行建模时将每个点分配给其相应的集群，这样得到的集群就不会试图解释输入集的重迭子集。由于这种先天的困难，聚类通常通过迭代算法来解决，这些算法细化随机初始化的聚类直到收敛。尽管具有集合极化操作的神经网络可以通过学习量化空间来近似这种摊销映射，但一个关键的缺点是这种量化不能依赖于集合的内容。这限制了解决方案的质量，也可能使此类模型的优化更加困难；研究者在该研究名为 Experiments 的第 5 节，让我们知道经验表明，此类池化架构存在欠拟合的问题。该研究提出了一种称为 Set Transformer 的新型集输入深度神经网络架构 (cf. Transformer, (Vaswani et al., 2017))。而 Set Transformer 的新颖之处在于三个重要的设计选择：

- 研究使用自注意力机制来处理输入集中的每个元素，这允许我们的方法自然地编码集中元素之间的成对或高阶交互。
- 研究提出了一种将完全自注意力 (e.g. the Transformer) 的 $O(n^2)$ 计算时间减少到 $O(nm)$ 的方法，其中 m 是固定的超参数，允许该研究的方法扩展到大型输入集。
- 研究使用自注意力机制来聚合特征，这在问题需要多个相互依赖的输出时特别

有用，例如元聚类问题 (the problem of meta-clustering)，其中每个聚类中心的含义在很大程度上取决于其相对位置到其他集群。

该研究将 Set Transformer 应用于几个集合输入问题，并凭经验证明了这些设计选择的重要性和有效性，并表明我们可以为大多数任务实现最先进的性能。

4.2.3 Deep amortized clustering

研究者们提出了一种深度摊销聚类 (DAC; deep amortized clustering)，这是一种神经架构，它学习使用一些前向传播有效地对数据集进行聚类，其 DAC 隐式地学习是什么构成了一个集群，如何将数据点分组到集群中，以及如何计算数据集中的集群数量。而 DAC 是使用标记数据集进行元学习的，这一过程不同于传统的聚类算法，而传统聚类算法通常需要手工指定的关于聚类形状和结构的先验知识。研究者凭经验表明，在合成数据和图像数据上，DAC 可以有效且准确地对来自用于生成训练数据集的相同分布的新数据集进行聚类。

而从图可知，该研究的模型每次迭代识别一个集群（顶部），使其能够找到任意数量的集群（底部），其聚类是无监督机器学习中的一项基本任务，用于将相似的数据点分组到多个集群中。除了在许多下游任务中的有用性之外，聚类是可视化和理解数据集底层结构的重要工具，也是认知科学中的分类模型，而大多数聚类算法有两个基本组成部分——如何定义集群以及如何将数据点分配给这些集群。

前者通常使用度量来测量数据点之间的距离，或使用描述集群形状的生成模型来定义，而后者如何将数据点分配给集群，然后通常会迭代优化 w.r.t. 基于集群定义导出的目标函数。请注意聚类定义是用户定义的，是用户对聚类过程的先验知识的反映，不同的定义导致不同的聚类，然而实践中使用的集群定义通常非常简单，例如 k-means 中的集群是根据到质心的 2 距离定义的，而高斯是混合模型中集群的常用生成模型。近来深度学习的进步促进了以黑盒方式逼近复杂函数，本研究中与聚类问题相关的一个特殊应用是摊销推理 (Gershman & Goodman, 2014; Stuhlmüller et al., 2013)，其中训练神经网络以预测潜在变量的状态模型或概率程序。而在学习集输入神经网络的背景下 (Zaheer et al., 2017), Lee et al. (2019) 表明可以分摊高斯混合 (MOG;a Mixture of Gaussians) 的迭代聚类过程，而 Pakman et al. (2019) 证明可以训练神经网络将数据点顺序分配给集群。此两种方法都可以解释为使用神经网络对给定数据集的集群分配和参数进行摊销推理，而在此要注意的部分在于，一旦将神经网络用于摊销聚类，研究者们就可以利用它们的灵活性来使用更复杂的方法来定义聚类。此外，摊销网络可以使用生成的数据集进行训练，其中地面实况聚类是已知的。这也可以解释为隐式学习训练数据集底层的集群定义，这样摊销推理（大约）会产生适当的集群。

从某种意义上来看，这与神经营过程 (Garnelo et al., 2018b;a) 有着相似的哲学，后者从多个数据集进行元学习 (meta-learns) 以学习函数的先验，回到该研究来看，研究者以这些先前的工作为基础，并提出了深度摊销聚类 (DAC;Deep Amortized Clustering)，该研究与之前的工作一样，DAC 中的摊销网络使用生成的数据集进行训练，其中地面实况聚类是已知的，像 Lee et al. (2019)，DAC 使用 Set Transformer，但与 Lee et al. (2019) 因为它按顺序生成集群，这可以根据数据集的复杂性生成不同数量的集群。此研究的方法还扩展了 Lee et al. (2019) 从 MOG 到更复杂的集群定义问题，可以说这些问题更难手动指定，更容易从数据中进行元学习，而其工作也不同于 Pakman et al.(2019)，因为我们的网络并行处理数据点，而 Pakman et al. (2019) 按顺序处理它们，这可以说是可扩展性较低，并且限制了对较小数据集的适用性。

该研究的组织如下。研究者们首先在该研究的名为 A PRIMER ON SET TRANSFORMER AND AMORTIZED CLUSTERING 的第 2 节中，去描述在整篇论文中使用的置换不变集变换器模块。同时在名为 DEEP AMORTIZED CLUSTERING 的第 3 节中，此研究描述了研究者们如何实现一次识别一个集群的核心思想，并描述了研究者的聚类框架 DAC 在复杂数据集上解决 DAC 有几个挑战，同时按难度顺序构建此论文。

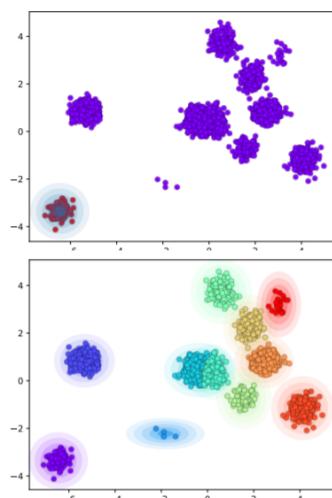


Figure 1: Our model identifies one cluster per iteration (top), allowing it to find any number of clusters (bottom).

图 4.3 Deep amortized clustering

4.2.4 Non-local Neural Networks

此研究指出卷积和循环操作都是一次处理一个局部邻域的构建块。在该研究中，研究者们将非本地操作呈现为用于捕获远程依赖项的通用构建块系列，而受计算器视觉

中经典的非局部均值方法的启发，研究者的非局部操作将某个位置的响应计算为所有位置特征的加权和，而此构建块可以插入到许多计算器视觉架构中。同时在视频分类任务上，即使没有任何繁杂的东西，研究者的非本地模型也可以在 Kinetics 和 Charades 数据集上与当前的竞争者竞争或胜出，另外在静态图像识别中，研究者们的非局部模型改进了 COCO 任务套件上的对象检测/分割和姿态估计，可捕获远程依赖关系在深度神经网络中至关重要。对于如在语音 (speech)、语言 (language) 的循环操作 (recurrent operations) 与序列数据 (sequential data) 是远程依赖建模的主要解决方案，而对于图像数据来说，长距离依赖是由卷积运算的深堆栈形成的大感受野建模。另外卷积运算和循环运算都在空间或时间上处理局部邻域，因此只有在重复应用这些操作时才能捕获远程依赖关系，通过数据逐步传播信号。在此重复本地操作有几个限制，首先它在计算上效率低下，其次它会导致需要仔细解决的优化困难。

最后这些挑战使得多跳依赖建模变得困难，比如当需要在远距离位置之间来回传递消息的时候。在该研究中，研究者将非本地操作作为一种高效、简单且通用的组件，用于捕获深度神经网络的远程依赖关系，当中所提出的非局部操作是计算器视觉中经典非局部平均操作的推广。直观地说，非局部操作将某个位置的响应计算为输入特征图中所有位置的特征的加权。此组位置可以是空间、时间或时空，这意味着我们的操作适用于图像、序列和视频问题。使用非本地操作有几个优点：(a) 与循环和卷积运算的渐进行为相反，非局部运算通过计算任意两个位置之间的相互作用直接捕获远程依赖关系，而不管它们的位置距离如何；(b) 正如我们在实验中所展示的，即使只有几层，非本地操作也是有效的，并且可以达到最佳效果；(c) 最后，研究非本地操作保持了可变的输入大小，并且可以很容易地与其他操作，例如研究者将使用的卷积结合起来，同时研究展示了非本地操作在视频分类应用中的有效性，而在视频中远距离的空间和时间像素之间会发生远程交互。

作为该研究的基本单元的单个非本地块可以前馈方式直接捕获这些时空依赖性，有了一些非局部块，研究者认为非局部神经网络且包含包括膨胀变体的架构对于视频分模拟 2D 和 3D 卷积网络更为准确。此外，非局部神经网络比 3D 卷积神经网络在计算上更经济，同时在 Kinetics 和 Charades 数据集上介绍了综合消融研究。仅使用 RGB 而没有任何类似光流、多尺度测试复杂的东西，本研究的方法在两个数据集上取得的结果与最新的比赛获胜者相当或更好。为了证明非局部操作的普遍性，研究进一步介绍了 COCO 数据集上的对象检测/分割和姿态估计实验，而在强大的 Mask R-CNN 基线之上，研究者的非局部块可以以很小的额外计算成本提高所有三个任务的准确性，连同视频中的证据，这些图像实验表明非局部操作通常是有用的，并且可以成为设计深度神经网络的基本构建块。从图中我们可以看到网络中的一个时空非局部操作训练用

于 Kinetics 中的视频分类，位置 x_i 的响应由所有位置 x_j 的特征的加权平均值计算，而此处仅显示最高权重的特征，在这个由我们的模型计算的示例中，请注意它如何将第一帧中的球与最后两帧中的球相关联。

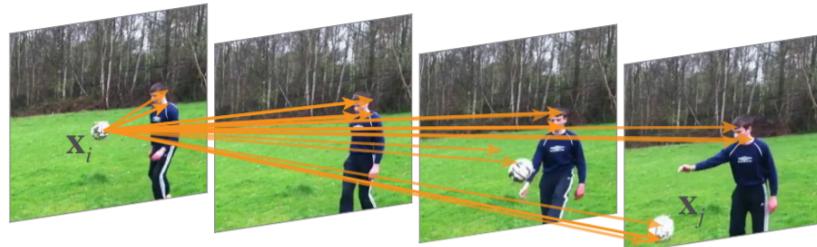


Figure 1. A spacetime ***non-local*** operation in our network trained for video classification in Kinetics. A position \mathbf{x}_i 's response is computed by the weighted average of the features of *all* positions \mathbf{x}_j (only the highest weighted ones are shown here). In this example computed by our model, note how it relates the ball in the first frame to the ball in the last two frames. More examples are in Figure 3.

图 4.4 Non-local Neural Networks

4.2.5 Axial-DeepLab: Stand-Alone Axial-Attention for Panoptic Segmentation

卷积利用局部性来提高效率，代价是丢失了远程上下文，已采用自注意力来增强具有非本地交互作用的 CNN，最近的工作证明可以通过将注意力限制在局部区域来堆栈自注意力层以获得完全注意力网络，该研究试图通过将 2D 自注意力分解为两个一维自注意力来消除此约束，在此降低了计算复杂度，并允许在更大甚至全局区域内执行注意力。同时研究者还提出了一种位置敏感的自注意力设计，将两者结合产生研究者想要的位置敏感轴向注意层，这是一种新颖的构建块，可以堆栈以形成用于图像分类和密集预测的轴向注意模型。该研究证明了研究者的模型在四个大规模数据集上的有效性，特别在于其模型优于 ImageNet 上所有现有的独立自注意力模型，而研究者的 Axial-DeepLab 在 COCO 测试开发上比自下而上的最新技术提高了 2.8% 的 PQ，先前的最新技术是通过我们的小变体实现的，该变体的参数效率为 3.8 倍，计算效率为 27 倍，Axial-DeepLab 还在 Mapillary Vistas 和 Cityscapes 上取得了最先进的结果。

卷积是计算器视觉的核心构建块，早期算法使用卷积滤波器来模糊图像、提取边缘或检测特征。前者与全连接模型相比，由于其效率和泛化能力，它在现代神经网络中得到了大量利用。卷积的成功主要来自平移等方差性和局部性的两个特性，平移等方差虽然不精确，但与成像的性质非常吻合，因此可以将模型推广到不同的位置或不同尺寸的图像。另一方面，局部性减少了参数计数和 M-Adds，然而它使建模远程关系

具有挑战性，另外大量文献讨论了在卷积神经网络 (CNN) 中对远程交互进行建模的方法，而一些采用多孔卷积、更大的内核或图像金字塔，要么是手工设计的，又或者是通过算法搜索，另一行作品采用注意力机制，注意显示了它在语言建模、语音识别和神经字幕中对远程交互进行建模的能力。此后注意力已扩展到视觉，显著提升了图像分类、对象检测、语义分割、视频分类和对抗性防御。这些工作通过非局部或远程注意力模块丰富了 CNN，近来有研究已经提出将注意力层堆栈为没有任何空间卷积的独立模型并显示出有希望的结果，同时朴素的注意力在计算上是昂贵的，尤其是在大输入上。提出的将局部约束应用于注意力，可以降低成本并能够构建完全注意力模型。同时局部约束限制了模型感受野，这对于分割等任务至关重要，尤其是在高分辨率输入上。在此项工作中，我们建议采用轴向注意力，它不仅可以进行高效计算，而且可以恢复独立注意力模型中的感受。

核心思想是将 2D 注意力分解为沿高度和宽度轴顺序的两个 1D 注意力，其效率使我们能够参与大区域并构建模型来学习远程甚至全局交互。此外，大多数先前的注意力模块不利用位置信息，这会降低如多尺度的形状或对象的注意力在建模与位置相关的交互中的能力。最近的作品会将位置术语引入注意力，但是以一种与上下文无关的方式。在该研究中，研究者将位置项增加为上下文相关，使研究的注意力对位置敏感，并具有边际成本。研究展示该研究的轴向注意力模型在 ImageNet 上进行分类的有效性，以及在三个数据集（COCO、Mapillary Vistas 和 Cityscapes）上进行全景分割、实例分割、和语义分割。特别是，在 ImageNet 上，研究者通过用其位置敏感的轴向注意层替换所有残差块中的 3×3 卷积来构建 Axial-ResNet，并通过采用轴向注意层进一步使其获得完全注意。因此该研究的 Axial-ResNet 在 ImageNet 上的独立注意力模型中取得了最先进的结果。而对于分割任务，研究者通过替换 Panoptic-DeepLab 中的主干将 Axial-ResNet 转换为 Axial-DeepLab。

在 COCO 上，研究者的 Axial-DeepLab 在测试开发集上以 2.8% 的 PQ 优于当前自下而上的最新技术 Panoptic-DeepLab，而研究者还在 Mapillary Vistas 和 Cityscapes 上展示了最先进的分割结果。

总而言之，研究者的贡献有四方面：

- 所提出的方法是首次尝试构建具有大或全局感受野的独立注意力模型。
- 该研究提出了位置敏感的注意力层，它可以更好地利用位置信息而不会增加太多的计算成本。
- 该研究表明轴向注意力运作良好，不仅作为图像分类的独立模型，而且作为全景分割、实例分割和分割分割的支柱
- 该研究的 Axial-DeepLab 在 COCO 上比自下而上的最新技术显著改进，实现了

与两阶段方法相当的性能。

最后此研究还在 Mapillary Vistas 和 Cityscapes 上超越了之前最先进的方法。

4.2.6 Taming Transformers for High-Resolution Image Synthesis

研究首先指出在学习序列数据上的远程交互，transformers 继续在各种任务上显示出最先进的结果，而前者与 CNN 相比，它们不包含优先考虑局部交互的归纳偏差。这使它们具有表现力，但对于如高分辨率图像的长序列，在其计算上也不可行。

本研究展示了如何将 CNN 的归纳偏置的有效性与 Transformer 的表达能力相结合，使它们能够建模并从而合成高分辨率图像，大致分为 (i) 使用 CNN 来学习图像成分的上下文丰富的词汇，并反过来 (ii) 利用转换器在高分辨率图像中有效地对其进行组成进行建模。

该研究的方法很容易应用于条件合成任务，其中如对像类等非空间信息和如分割等空间信息都可以控制生成的图像，特别在于此研究展示了带有转换器的百万像素图像的语义引导合成的第一个结果，并在类条件 ImageNet 上的自回归模型中获得了最先进的技术。

Transformer 正在兴起——它们现在是语言任务的事实上的标准架构，并且越来越多地适用于其他领域，例如音频和视觉。前者与主要的视觉架构卷积神经网络 (CNN) 相比，transformer 架构不包含有关交互位置的内置归纳先验，因此可以自由地学习其输入之间的复杂关系，然而此种普遍性也意味着它必须学习所有关系，而 CNN 旨在利用有关图像内强局部相关性的先验知识。因此 transformer 的表现力的增加伴随着计算成本的二次增加，因为所有的成对交互都被考虑在内，同时最先进的变压器模型所产生的能量和时间要求为将它们缩放为具有数百万像素的高分辨率图像带来了根本问题。

变压器倾向于学习卷积结构的观察结果因此提出了一个问题：

每次训练视觉模型时，研究者考虑到是否必须从头开始重新学习我们所知道的关于图像的局部结构和规律性的所有内容，或者研究者是否可以有效地编码归纳图像偏差，同时仍然保留转换器的灵活性的部分，该研究假设低级图像结构可以通过局部连接很好地描述，即卷积架构，而这种结构假设在更高的语义级别上不再有效。此外 CNN 不仅表现出强烈的局部性偏差，而且通过在所有位置上使用共享权重，还表现出对空间不变性的偏差，但如果需要更全面地了解输入，这会使它们无效。

研究者获得有效且富有表现力的模型的关键见解是，卷积和变换器架构一起可以仿真我们视觉世界的组成性质，该研究使用卷积方法有效地学习上下文丰富的视觉部分的码本，然后学习它们的全局组合模型。而这些组合中的远程交互需要一个富有表现力的转换器架构来仿真其组成视觉部分的分布。此外，研究者利用对抗性方法来确

保局部部分的字典捕获感知上重要的局部结构，以减轻使用转换器架构对低级统计数据进行建模的需要。让 Transformer 专注于它们独特的优势——对远程关系建模——使它们能够生成如图中所示的高分辨率图像，这是以前无法实现的壮举。其图中表明此研究的方法使 Transformer 能够合成像这样的高分辨率图像，其中包含 1280x460 像素，同时该研究的公式通过调节有关所需对像类别或空间布局的信息来控制生成的图像。最后实验表明，该研究的方法通过优于以前基于卷积架构的基于码本的最新方法，保留了转换器的优势。

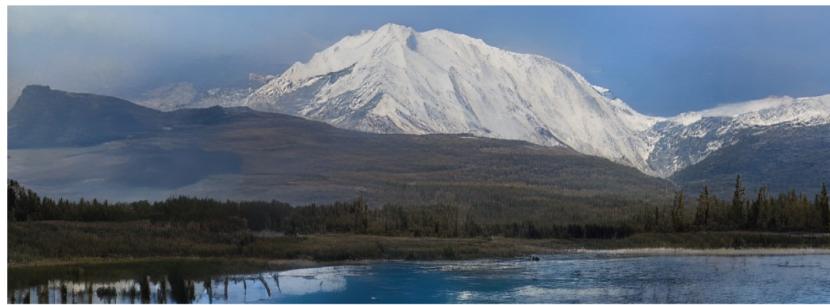


Figure 1. Our approach enables transformers to synthesize high-resolution images like this one, which contains 1280x460 pixels.

图 4.5 Taming Transformers for High-Resolution Image Synthesis

4.2.7 Video Action Transformer Network

本研究引入了 Action Transformer 模型，用于识别和定位视频剪辑中的人类动作，该研究重新利用了 Transformer 风格的架构，从研究中试图对其行为进行分类的人周围的时空上下文中聚合特征。研究表明，通过使用高分辨率、特定于个人、与类别无关的查询，该模型自发地学习跟踪个人并从他人的行为中获取语义上下文。此外，它的注意力机制学习强调手和脸，这对于区分动作通常是非常重要的——除了框和类别标签之外，所有这些都没有明确的监督。该研究在 Atomic Visual Actions (AVA) 数据集上训练和测试研究本身的 Action Transformer 网络，仅使用原始 RGB 帧作为输入，其性能明显优于最新技术。在此研究中，研究者的目标是定位和识别视频剪辑中的人类行为，人类行为仍然如此难以识别的一个原因是，推断一个人的行为通常需要了解周围的人和物体。比如识别一个人是否在“听某人说话”取决于场景中是否存在另一个人在说些什么。同样要识别一个人是“指着一个物体”，还是“拿着一个物体”，还是“握手”；所有这些都需要对人及其周围环境的有生命和无生命元素进行共同推理。

所以这不仅限于给定时间点的上下文而已，从识别“看人”的动作，在被看人走出画面后，需要随着时间的推移进行推理，以了解我们感兴趣的人实际上是在看人，而不仅仅是凝视远方。因此，该研究寻求一种在确定感兴趣的人的行为时可以确定和利用此类上下文信息（其他人、其他对象）的模型，Vaswani et al. 的 Transformer 架构是

一个合适的模型，因为它使用自注意力明确地为其表示构建上下文支持。在此与传统的循环模型相比，这种架构在序列建模任务方面取得了巨大成功。同时另一个问题是在于，如何为人类行为识别建立一个类似的模型？其研究者的的答案是一个新的视频动作识别网络 Action Transformer，它使用修改后的 Transformer 架构作为“头部”来对感兴趣的人的动作进行分类。它汇集了另外两个想法：(i) 一个时空 I3D 模型，该模型在以前的视频动作识别方法中取得了成功，这项已提供了基本特征；(ii) 区域提议网络 (RPN)，此项提供了一种用于定位执行动作的人的采样机制。I3D 特征和 RPN 一起生成查询，该查询是 Transformer 头的输入，该查询聚合来自周围视频中其他人和对象的上下文信息。研究者除了详细描述了这种架构也表明，经过训练的网络能够学习跟踪个人并根据视频中其他人的行为将他们的行为置于情境中。此外，转换器关注手部和面部区域，研究者已经知道它们在区分动作时具有一些最相关的特征。

所有这些都是在没有明确监督的情况下获得，而是在动作分类期间学习，并且在 Atomic Visual Actions (AVA) 数据集上训练和测试研究的模型，而对于这种上下文推理，这是一个有趣且合适的测试平台，它需要在视频中半密集地检测多人，并识别多个基本动作。许多这些动作通常不能仅从人物边界框确定，而是需要推断与其他人和物体的关系，前者与之前的作品不同，研究者的模型无需明确的对象检测即可学会这样做。研究者在 AVA 数据集上创造了新记录，将性能从 17.4% 提高到 25.0% mAP。该网络仅使用原始 RGB 帧，但其性能优于之前的所有工作，包括使用额外光流和声音输入的大型集成。在提交时，该研究的方法是 ActivityNet 排行榜上表现最好的方法然而研究者也注意到在 25% mAP 时，这个问题，甚至这个数据集，还远未解决。因此，此研究最后严格分析了模型的失败案例，该研究描述了一些常见的故障模式，并分析了按语义和空间卷标分解的性能。

同时研究者发现许多训练集相对较大的类仍然难以识别，并调查此类尾部案例以标记未来工作的潜在途径。而从图中可以知道，操作中的 Action Transformer，研究者提出的多头/层 Action Transformer 架构学习关注感兴趣的人的相关区域及其上下文（其他人、对象），以识别他们正在做的动作。每个头部计算一个剪辑嵌入，用于关注不同的部分，如面部、手和其他人，以识别感兴趣的人正在“牵手”和“看着一个人”。

4.3 Transformer 工作

该章目标在于为了呼应前述报告而对其进行复现的工作，此报告根据 Paris Dauphine University 和 Facebook AI 于 ECCV 2020 所发表的 End-to-End Object Detection with Transformers^[29] 进行测试。而测试的程式码则可于本专案的 code 目录中找到，其档案名称为 `detr_demo.ipynb`。

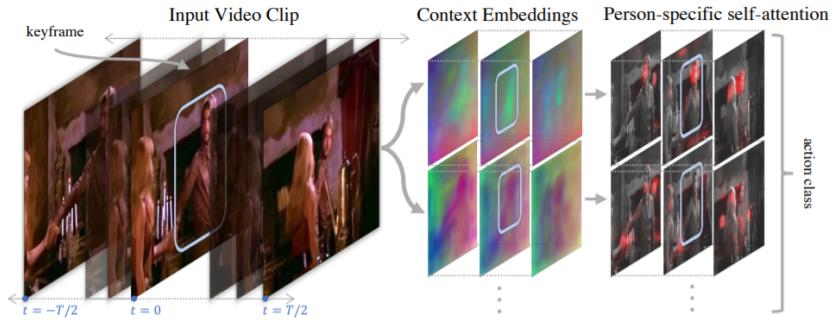


Figure 1: Action Transformer in action. Our proposed multi-head/layer Action Transformer architecture learns to attend to relevant regions of the person of interest and their context (other people, objects) to recognize the actions they are doing. Each head computes a clip embedding, which is used to focus on different parts like the face, hands and the other people to recognize that the person of interest is ‘holding hands’ and ‘watching a person’.

图 4.6 Video Action Transformer Network

4.3.1 摘要与研究贡献

研究者提出了一种将对象检测视为直接集预测问题的新方法，該方法简化了检测管道，且有效地消除了对许多手工设计组件的需求，例如非最大抑制程序或锚生成，这些组件明确地编码了我们关于任务的先验知识。在此新框架的主要成分称为 DETec-tion TRansformer 或 DETR，其原理是基于集合的全局损失，通过二部匹配强制进行唯一预测，以及转换器编码器-解码器架构。其流程是给定一组固定的学习对象查询集，DETR 会推理对象和全局图像上下文之间的关系，以直接并行输出最终的预测集。本研究的成果与许多其他现代探测器不同，該研究的新模型在概念上很简单，不需要专门的库。而 DETR 在具有挑战性的 COCO 对象检测数据集上展示了与完善且高度优化的 Faster RCNN 基线相当的准确性和运行时性能。同时 DETR 可以很容易地推广到以统一的方式产生全景分割。

研究者先是明对象检测的目标是为每个感兴趣的对象预测一组边界框和类别标签，而现代检测器 (Modern detectors) 通过在大量建议、锚点或窗口中心上定义代理回归和分类问题，以间接方式解决此集合预测任务。這些檢測器的性能受到后处理步骤的显着影响，基本上會以折叠近似重复的预测锚集的设计以及将目标框進行分配。而为了简化这些流程，我们提出了一种直接集预测方法来绕过代理任务。此類端到端的理念在复杂的结构化预测任务在如机器翻译或语音识别等方面取得了重大进展，但尚未在对象检测領域上取得重大进展與成功，而之前的尝试要么添加其他形式的先验知识，或者没有被证明在具有挑战性的基准上具有强大的基线竞争力。而本研究的目標

End-to-End Object Detection with Transformers

Nicolas Carion^{1,2[0000-0002-2308-9680]}, Francisco Massa^{2[000-0003-0697-6664]},
Gabriel Synnaeve^{2[0000-0003-1715-3356]}, Nicolas Usunier^{2[0000-0002-9324-1457]},
Alexander Kirillov^{2[0000-0003-3169-3199]}, and Sergey
Zagoruyko^{2[0000-0001-9684-5240]}

¹ Paris Dauphine University

² Facebook AI

{alcinos, fmassa, gab, usunier, akirillov, szagoruyko}@fb.com

Abstract. We present a new method that views object detection as a direct set prediction problem. Our approach streamlines the detection pipeline, effectively removing the need for many hand-designed components like a non-maximum suppression procedure or anchor generation that explicitly encode our prior knowledge about the task. The main ingredients of the new framework, called DETR, are a set-based global loss that forces unique predictions via bipartite matching, and a transformer encoder-decoder architecture. Given a fixed small set of learned object queries, DETR reasons about the relations of the objects and the global image context to directly output the final set of predictions in parallel. The new model is conceptually simple and does not require a specialized library, unlike many other modern detectors. DETR demonstrates accuracy and run-time performance on par with the well-established and highly-optimized Faster R-CNN baseline on the challenging COCO object detection dataset. Moreover, DETR can be easily generalized to produce panoptic segmentation in a unified manner. We show that it significantly outperforms competitive baselines. Training code and pretrained models are available at <https://github.com/facebookresearch/detr>.

图 4.7 End-to-End Object Detection with Transformers

就是想要弥补这一個差距，其研究者通过将对象检测视为直接集预测问题来简化训练管道。他們采用基于转换器的编码器-解码器架构，这是一种流行的序列预测架构。

转换器的自注意力机制显式地对序列中元素之间的所有成对交互进行建模，使这些架构特别适用于集合预测的特定约束，例如删除重复预测。該研究的检测变换器 DETR 會一次预测所有对象，并使用一组损失函数进行端到端训练，该函数在预测对象和真实对象之间执行二分匹配。而所謂的 DETR 會通过将常见的 CNN 与变压器架构相结合，直接预测（并行）最终检测集，在模型進行训练期间，二分匹配使用真实值框唯一地進行分配预测，而没有匹配的预测应产生“无对象”（）类的预测。DETR 通过删除多个手工设计的编码先验知识的组件来简化检测管道，如空间锚点或非最大抑制。而 DETR 与大多数现有检测方法不同，DETR 本身不需要任何自定义层，因此可以在包含标准 ResNet 和 Transformer 类的任何框架中轻松重现。該模型与之前关于直接集预测的大多数工作相比，DETR 的主要特征是二部匹配损失和变换器与非自回归的并行解码的结合。跟過往的研究相較之下，其工作侧重于使用 RNN 进行自回归解码。而研究者的匹配损失函数唯一地将预测分配给真实对象，并且对预测对象的排列

是不变的，因此該研究可以并行发出它们。

最後研究團隊在最流行的对象检测数据集 COCO 上评估 DETR，將 DETR 與非常有竞争力的 Faster R-CNN 基线相比，Faster RCNN 经历了多次设计迭代，其性能自最初发布以来得到了极大的提升。研究的实验表明，其研究的新模型实现了可比的性能，DETR 在大型对象上表现出明显更好的性能，这一结果可能是由转换器的非本地计算实现的。然而 DETR 在小物体上的性能较低，研究者预计未来的工作将以与 FPN 为 Faster R-CNN 所做的开发相同的方式改进这方面，同时 DETR 的训练设置与标准物体检测器有多种不同。而新模型需要超长的训练计划，并受益于变压器中的辅助解码损失，同时該研究也徹底探索了哪些组件对展示的性能是至关重要部分，而且 DETR 的设计理念很容易扩展到更复杂的任务。= 在研究者的实验中表明在预训练的 DETR 之上训练的简单分割头优于全景分割的竞争基线，这是一项近期具有挑战性的像素级识别任务。

4.3.2 程式码与演示

在此根据专案的论文测试的程式码进行呈现，而该档案则可于本专案的 code 目录中找到，其档案名称为 detr_demo.ipynb。而在此使用 DETR 进行对象检测，研究者展示了名为 DETR 检测变换器的演示并展示了如何定义模型、加载预训练权重以及可视化边界框和类别预测。

1. 套件与环境

```
from PIL import Image
import requests
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import torch
from torch import nn
from torchvision.models import resnet50
import torchvision.transforms as T
torch.set_grad_enabled(False);
```

2. DETR 实现

```
class DETRdemo(nn.Module):
    """
    Demo DETR implementation.

    Demo implementation of DETR in minimal number of lines, with the
```

```

following differences wrt DETR in the paper:
* learned positional encoding (instead of sine)
* positional encoding is passed at input (instead of attention)
* fc bbox predictor (instead of MLP)

The model achieves ~40 AP on COCO val5k and runs at ~28 FPS on Tesla V100.
Only batch size 1 supported.

"""

def __init__(self, num_classes, hidden_dim=256, nheads=8,
             num_encoder_layers=6, num_decoder_layers=6):
    super().__init__()

    # create ResNet-50 backbone
    self.backbone = resnet50()
    del self.backbone.fc

    # create conversion layer
    self.conv = nn.Conv2d(2048, hidden_dim, 1)

    # create a default PyTorch transformer
    self.transformer = nn.Transformer(
        hidden_dim, nheads, num_encoder_layers, num_decoder_layers)

    # prediction heads, one extra class for predicting non-empty slots
    # note that in baseline DETR linear_bbox layer is 3-layer MLP
    self.linear_class = nn.Linear(hidden_dim, num_classes + 1)
    self.linear_bbox = nn.Linear(hidden_dim, 4)

    # output positional encodings (object queries)
    self.query_pos = nn.Parameter(torch.rand(100, hidden_dim))

    # spatial positional encodings
    # note that in baseline DETR we use sine positional encodings
    self.row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
    self.col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))

def forward(self, inputs):
    # propagate inputs through ResNet-50 up to avg-pool layer
    x = self.backbone.conv1(inputs)
    x = self.backbone.bn1(x)
    x = self.backbone.relu(x)
    x = self.backbone.maxpool(x)

    x = self.backbone.layer1(x)
    x = self.backbone.layer2(x)

```

```

x = self.backbone.layer3(x)
x = self.backbone.layer4(x)

# convert from 2048 to 256 feature planes for the transformer
h = self.conv(x)

# construct positional encodings
H, W = h.shape[-2:]
pos = torch.cat([
    self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
    self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),
], dim=-1).flatten(0, 1).unsqueeze(1)

# propagate through the transformer
h = self.transformer(pos + 0.1 * h.flatten(2).permute(2, 0, 1),
                     self.query_pos.unsqueeze(1)).transpose(0, 1)

# finally project transformer outputs to class labels and bounding boxes
return {'pred_logits': self.linear_class(h),
        'pred_boxes': self.linear_bbox(h).sigmoid()}

```

由于 Transformer 的表示能力，DETR 架构非常简单。有两个主要组成部分：

1. 一个卷积主干——我们在这个演示中使用 ResNet-50
2. 一个 Transformer - 我们使用默认的 PyTorch nn.Transformer

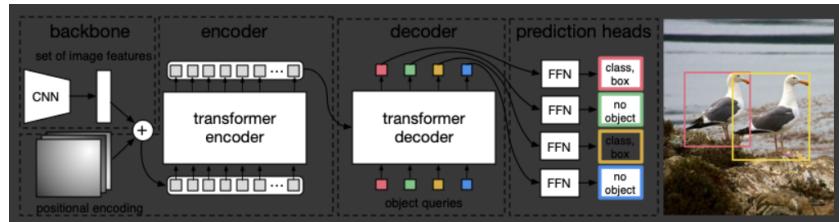


图 4.8 DETR 架构

让我们用 80 个 COCO 输出类 + 1 0 “无对象” 类构建模型并加载预训练的权重。权重以半精度保存，以节省带宽而不影响模型精度。

3. 载入训练

```

detr = DETRdemo(num_classes=91)
state_dict = torch.hub.load_state_dict_from_url(
    url='https://dl.fbaipublicfiles.com/detr/detr_demo-da2a99e9.pth',
    map_location='cpu', check_hash=True)
detr.load_state_dict(state_dict)
detr.eval();

```

4. 使用 DETR 计算预测

加载的预训练 DETR 模型已经在 80 个 COCO 类上进行了训练，类索引从 1 到 90（这就是我们在模型构建中考虑 91 个类的原因）。在以下单元格中，我们定义了从类索引到名称的映射。

```
# COCO classes
CLASSES = [
    'N/A', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A',
    'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
    'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack',
    'umbrella', 'N/A', 'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
    'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove',
    'skateboard', 'surfboard', 'tennis racket', 'bottle', 'N/A', 'wine glass',
    'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
    'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
    'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table', 'N/A',
    'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
    'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
    'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
    'toothbrush'
]

# colors for visualization
COLORS = [[0.000, 0.447, 0.741], [0.850, 0.325, 0.098], [0.929, 0.694, 0.125],
          [0.494, 0.184, 0.556], [0.466, 0.674, 0.188], [0.301, 0.745, 0.933]]
```

DETR 使用标准的 ImageNet 归一化，并以 $[x_{\text{center}}, y_{\text{center}}, w, h]$ 格式输出相对图像坐标中的框，其中 $[x_{\text{center}}, y_{\text{center}}]$ 是边界框的预测中心， w, h 是其宽度和高度。因为坐标是相对于图像尺寸的，并且位于 $[0, 1]$ 之间，我们将预测转换为绝对图像坐标和 $[x_0, y_0, x_1, y_1]$ 格式以用于可视化目的。

```
# standard PyTorch mean-std input image normalization
transform = T.Compose([
    T.Resize(800),
    T.ToTensor(),
    T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# for output bounding box post-processing
def box_cxcywh_to_xyxy(x):
    x_c, y_c, w, h = x.unbind(1)
```

```

b = [(x_c - 0.5 * w), (y_c - 0.5 * h),
      (x_c + 0.5 * w), (y_c + 0.5 * h)]
return torch.stack(b, dim=1)

def rescale_bboxes(out_bbox, size):
    img_w, img_h = size
    b = box_cxcywh_to_xyxy(out_bbox)
    b = b * torch.tensor([img_w, img_h, img_w, img_h], dtype=torch.float32)
    return b

```

把所有东西放在一个检测函数中：

```

def detect(im, model, transform):
    # mean-std normalize the input image (batch-size: 1)
    img = transform(im).unsqueeze(0)

    # demo model only support by default images with aspect ratio between 0.5 and 2
    # if you want to use images with an aspect ratio outside this range
    # rescale your image so that the maximum size is at most 1333 for best results
    assert img.shape[-2] <= 1600 and img.shape[-1] <= 1600, 'demo model'

    # propagate through the model
    outputs = model(img)

    # keep only predictions with 0.7+ confidence
    probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
    keep = probas.max(-1).values > 0.7

    # convert boxes from [0; 1] to image scales
    bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)
    return probas[keep], bboxes_scaled

```

5. 使用 DETR

在此使用本作业专案的图进行测试，从网路上任意抓得仓鼠、考拉、柯基犬与范例的猫。

6. 结果输出

在此可以观察到加入的仓鼠并没有被辨识出来，同时研究中所用的猫能够被正确辨识之外，柯基犬被辨识成狗，而考拉则被辨识成熊。

```

def plot_results(pil_img, prob, boxes):
    plt.figure(figsize=(16,10))
    plt.imshow(pil_img)

```



(a) 測試 - 仓鼠

(b) 測試 - 考拉

(c) 測試 - 柯基犬

图 4.9 最终自有测试资料集

```
ax = plt.gca()
for p, (xmin, ymin, xmax, ymax), c in zip(prob, boxes.tolist(), COLORS * 100):
    ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                               fill=False, color=c, linewidth=3))
    cl = p.argmax()
    text = f'{CLASSES[cl]}: {p[cl]:0.2f}'
    ax.text(xmin, ymin, text, fontsize=15,
            bbox=dict(facecolor='yellow', alpha=0.5))
plt.axis('off')
plt.show()

plot_results(im1, scores1, boxes1)
plot_results(im2, scores2, boxes2)
plot_results(im3, scores3, boxes3)
plot_results(im4, scores4, boxes4)
```

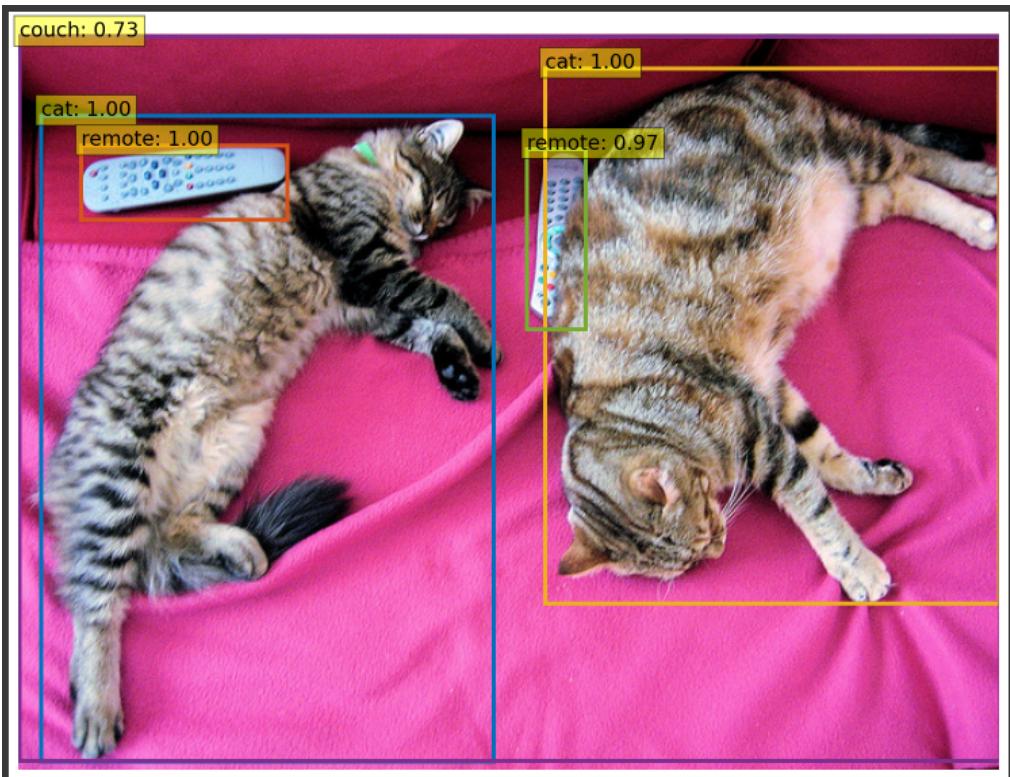


图 4.10 测试 - 猫

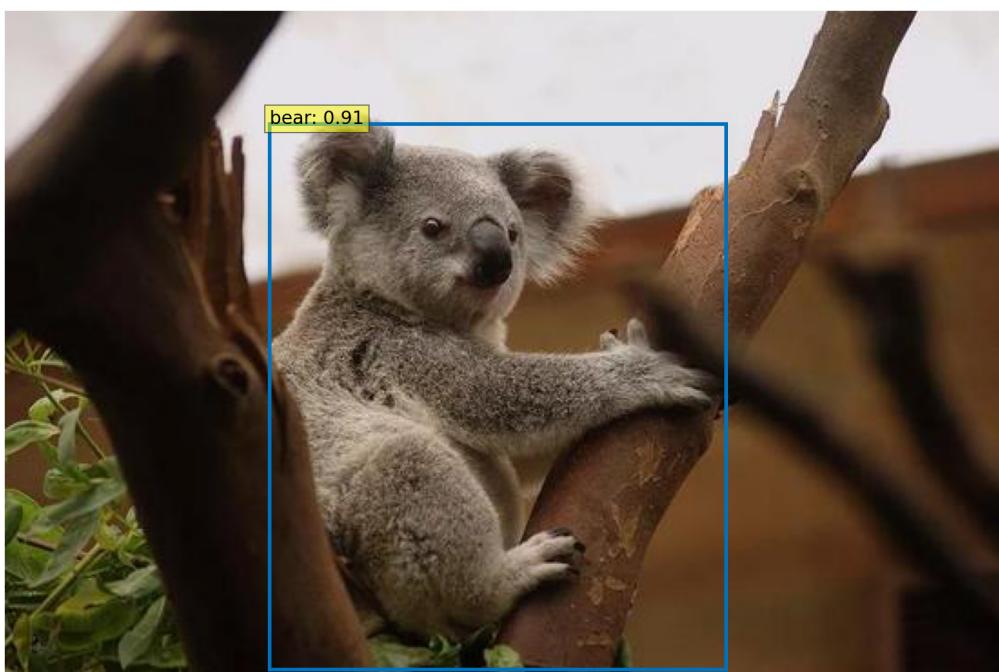


图 4.11 测试 - 考拉

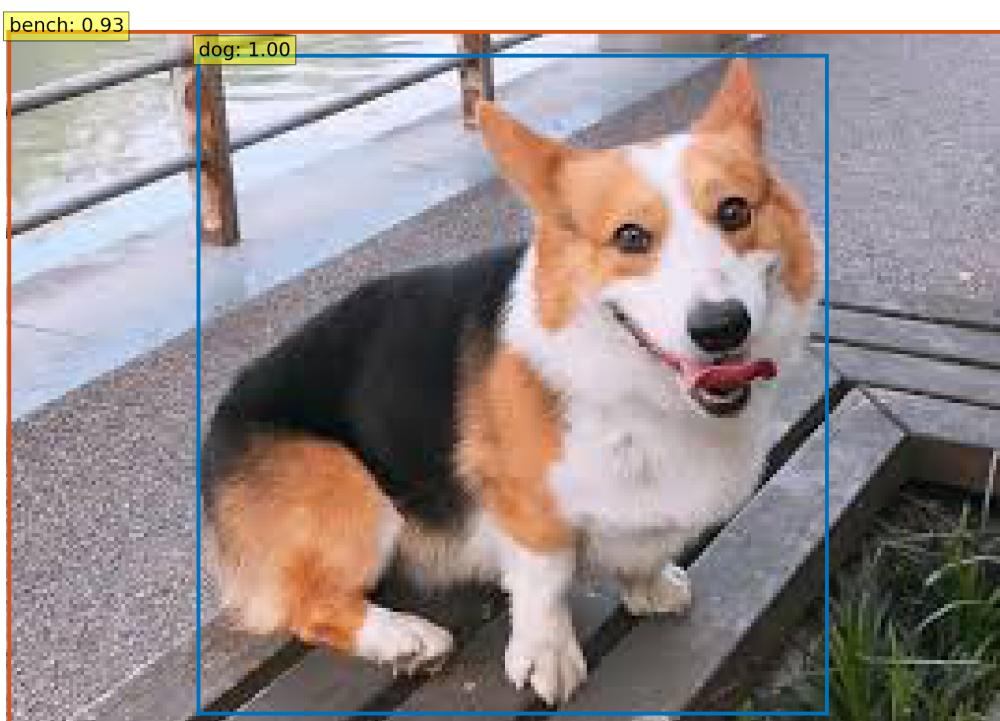


图 4.12 测试 - 柯基

第五章 总结

此报告根据调查与程序代码两方向做各自的呈现，此作业尽量力求兼顾两者，第一章说明作业目标、第二章撰写论文心得与论文全文翻译与精读、第三章处理哈佛大学自然语言处理实验室的文件心得含程序代码改版测试、全文翻译精读，并简述说明过程使用 LaTeX 与 Docker 来改善自身目前的研究工作流程跟说明为了完成作业所阅读的众多技术文件与网络资源，而第四章则分为三大部分，第一部分为 Transformer 理解、第二部分为论文文献综述还有调研准备工作包含阅读的研究，第三部分则为自己所选择复现地论文。最后才是此章回馈与总结。

5.1 课程回馈

Feedback (5pt)

Time you spend for this assignment, i.e., how many hours? (1pt)

Comments for this course? (1pt)

Comments for this assignment? (1pt)

Suggestion for the lectures of next year? (2pt)

1. Time you spend for this assignment, i.e., how many hours?

第二章的论文与第三章的技术文件所涉略的技术文件与资源收集，从 9 月始开学作业标准公布后就慢慢进行搜集，在 10 月时抽出各周周末慢慢查技术文件等资源去了解 Transformer 的运作机制、原理、主流前沿研究的发展，在 10 月时就已经完成了第二章论文精读的草稿并做出笔记，而到了 11 月开始查 Transformer 的程式码，并开始尝试哈佛实验室技术文件的问题，最后于 11 底完成第一版本的作业。同时本作业的 LaTeX 北京大学模板与 Docker 使用，甚至其他技术部分则是根据科技论文写作、计算机视觉、机器学习、数位媒体技术等不同课程作业探索而来的集大成，这些成果也是于这学期不同的周末所完成。最后则是第四章的三大部分，其进度于 12 月开工，但由于身体健康不佳的缘故，导致必须作息回归正常，整体的进度才缓下来，最后只能把握不同的时间进行准备。

总而言之，该作业横跨整个学期不同时段跟周末，具体花了多少小时很难进行计算，但在完成这份作业的路上学生写得很尽兴，而且也利用这个接触了一些自己平常不会去了解的有趣技术，若时间够长学生还可以写的更完善、更完美，当然这份作业会因为研究的需求，后续工作会继续进行下去。

2. Comments for this course?

我其实写得很开心，但如果可以，我会希望能有课堂录影之类的手段，让有机会重复复习，因为自己的基础跟能力有限，往往都是课后慢慢阅读，甚至要读好一段时间才能理解。很多时候看到同在教室的同济们的优秀表现，让学生羡慕不已。

另一个希望推广组成团队，全班同学一起进行共笔等，所谓的共笔是台湾医学院学生们的文化，一群人一起集思广益变强，学生相信一群人一起写协作写笔记爬文讨论的感觉必然不一样。

3. Comments for this assignment?

希望下次作业细节条件能够更早确定。作业条件的变化会影响学生排时间的时间表，而时间表的改动则会影响学生的时间安排紧凑程度，从而导致作息状况。

4. Suggestion for the lectures of next year?

下次可以建议课程模组化，分为必须掌握的基础跟老师自由发挥部分，基础部分预先录制好，这样老师即便面对跟不上学生也没关系，当然这必须要有规划自由发挥的部分，这样学生就必须来上课，确保出席率。若配合建议的共笔或许会更有成效。

参考文献

- [1] 邱锡鹏. 神经网络与深度学习[M/OL]. 北京: 机械工业出版社, 2020. <https://nndl.github.io/>.
- [2] VASWANI A, SHAZER N, PARMAR N, et al. Attention is all you need[C]//Advances in neural information processing systems. [S.l. : s.n.], 2017: 5998-6008.
- [3] TensorFlow. GitHub - tensorflow/tensor2tensor[EB/OL]. <https://github.com/tensorflow/tensor2tensor>.
- [4] Paperswithcode. Paperswithcode - Attention Is All You Need[EB/OL]. <https://paperswithcode.com/paper/attention-is-all-you-need>.
- [5] TensorFlow. GitHub - tensorflow/models[EB/OL]. <https://github.com/tensorflow/models/tree/master/official/nlp/transformer>.
- [6] Graykode. GitHub - graykode/nlp-tutorial[EB/OL]. <https://github.com/graykode/nlp-tutorial>.
- [7] Graykode. GitHub - SamLynnEvans/Transformer[EB/OL]. <https://github.com/SamLynnEvans/Transformer>.
- [8] Huggingface. GitHub - huggingface/transformers[EB/OL]. <https://github.com/huggingface/transformers>.
- [9] Huggingface. GitHub - harvardnlp/annotated-transformer[EB/OL]. <https://github.com/harvardnlp/annotated-transformer>.
- [10] Fawazsammani. GitHub - fawazsammani/chatbot-transformer[EB/OL]. <https://github.com/fawazsammani/chatbot-transformer>.
- [11] PAN X, GE C, LU R, et al. On the Integration of Self-Attention and Convolution[J]. ArXiv preprint arXiv:2111.14556, 2021.
- [12] CVer 计算机视觉. ZhiHu - 又一篇视觉 Transformer 综述来了! [EB/OL]. <https://zhuanlan.zhihu.com/p/341995737>.
- [13] CUI J. Transformer 最新综述! [EB/OL]. <https://jishuin.proginn.com/p/763bfbd5f55d>.
- [14] 极市平台. Self-Attention 和 CNN 的优雅集成! 清华大学等提出 ACmix, 性能速度全面提升! [EB/OL]. <https://mp.weixin.qq.com/s/0LAYmXsGjxBwCm5roXF0tQ>.
- [15] 赵正宇 哈 S. ZhiHu - 搞懂 Transformer 结构, 看这篇 PyTorch 实现就够了[EB/OL]. <https://zhuanlan.zhihu.com/p/339207092>.
- [16] LYNN-EVANS S. How to code The Transformer in Pytorch[EB/OL]. <https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec>.
- [17] KLEIN G, KIM Y, DENG Y, et al. OpenNMT: Open-Source Toolkit for Neural Machine Translation[C/OL]//Proc. ACL. [S.l. : s.n.], 2017. <https://doi.org/10.18653/v1/P17-4012>. DOI: 10.18653/v1/P17-4012.
- [18] ZhiHu. ZhiHu - 熬了一晚上, 我从零实现了 Transformer 模型, 把代码讲给你听[EB/OL]. [http://zhuanlan.zhihu.com/p/411311520](https://zhuanlan.zhihu.com/p/411311520).

- [19] ZhiHu. ZhiHu - This post is all you need (层层剥开 Transformer) [EB/OL]. <https://zhuanlan.zhihu.com/p/420820453>.
- [20] LEE H Y. 【機器學習 2021】Transformer (上)[EB/OL]. <https://www.youtube.com/watch?v=n9TlOhRjYoc>.
- [21] LEE H Y. 【機器學習 2021】Transformer (下)[EB/OL]. <https://www.youtube.com/watch?v=N6aRv06iv2g>.
- [22] SAINI D. Transformer Implementation (Attention all you Need)[EB/OL]. <https://medium.com/Analytics-vidhya/bert-implementation-multi-head-attention-4a10142636fe>.
- [23] NEWER D. Transformer 的 PyTorch 实现[EB/OL]. <https://wmathor.com/index.php/archives/1455/>.
- [24] 忆臻. ZhiHu - 搞懂 Transformer 结构，看这篇 PyTorch 实现就够了（上）[EB/OL]. <https://zhuanlan.zhihu.com/p/48731949>.
- [25] ZhiHu. ZhiHu - 超详细图解 Self-Attention[EB/OL]. <https://zhuanlan.zhihu.com/p/410776234>.
- [26] ZhiHu. ZhiHu - Transformer - Attention is all you need[EB/OL]. <https://zhuanlan.zhihu.com/p/311156298>.
- [27] NEWER D. Transformer 详解[EB/OL]. <https://wmathor.com/index.php/archives/1438/>.
- [28] KHAN S, NASEER M, HAYAT M, et al. Transformers in vision: A survey[J]. ArXiv preprint arXiv:2101.01169, 2021.
- [29] CARION N, MASSA F, SYNNAEVE G, et al. End-to-end object detection with transformers[C]// European Conference on Computer Vision. [S.l. : s.n.], 2020: 213-229.