南京大学

本科毕业设计

院　　系　软件学院

专　　业　软件工程

题　　目　**预训练对代码补全效果影响的研究**

年　　级　2018　学　　号　181250092

学生姓名　刘子仰

导　　师　葛季栋　职　　称　副教授

提交日期　2022 年 6 月 3 日

南京大学本科生毕业论文（设计、作品）中文摘要

题目：预训练对代码补全效果影响的研究

院系：软件学院

专业：软件工程

本科生姓名：刘子仰

指导教师（姓名、职称）：葛季栋 副教授

摘要：

随着软件开发规模和复杂度的日益上升，对软件开发人员工作效率的需求一再提高，而一个优秀的代码编辑工具便能有效提高程序员开发效率。代码补全是集成开发环境中一个重要的功能，它能提供下一个可能的关键字或词语供程序员选择以此来提升软件开发效率。随着近些年来深度学习的成功，基于循环神经网络的语言模型被运用在源代码建模上，而在这些模型中，代码被转换为一个单词、序列或者抽象语法树中的节点，方便模型计算。通过代码片段模型计算出可能出现的下个单词并根据上下文选出最高几率的单词。进一步的，这些模型可以根据以前输入的代码片段来学习单词嵌入层，可用于后续的任务中。

本代码补全模型基于预训练模型 BERT[1]，同时使用了 CugLM[2]中的预训练任务进一步提升准确性。有别于传统的 SLM 模型或 CNN、RNN 架构，采用 Transformer 构成模型，在代码补全任务中表现远远优于其他模型。在这次毕业设计中本人完成了数据集获取、数据预处理、编写半监督学习模型主要架构、设计实验并验证预训练对代码补全的影响等工作。

关键词：代码补全；预训练；BERT；Transformer

南京大学本科生毕业论文（设计、作品）英文摘要

DESIGN: Effect of Pre-training on Code Completion

DEPARTMENT: School of Software Engineering

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Zi-Yang Liu

MENTOR: Professor Ji-Dong Ge

ABSTRACT:

According to the increasing scale and complexity of software development, the demand for the work efficiency is getting strict while an excellent code editor can effectively improve the development efficiency of software engineer. Code Completion is an important function of Integrated Development Environment, which can offer the next possible token for software engineers to choose, so as to promote the efficiency of software development. In recent years, the language model based on recurrent neural network is applied to source code modeling. In code completion task, the code snippet is transformed into a word, sequence or abstract syntax tree node to be the input of these models. Next, the model calculates the probability of the next token and selects the token with the highest probability according to the context. Further, these models can learn the word embedding layer according to the previously input code snippet, which can be used in downstream tasks.

The code completion model in this paper is based on the pre training model Bert, and uses the pre training task in CugLM to further improve the accuracy. BERT is different from the traditional SLM model or CNN and RNN architecture. It uses Transformer to form the model, which has better performance than other models in code completion task. In this graduation project, I have completed the data preparation, data pre-processing, main architecture based BERT coding, design and completion of experiments and so on.

Keywords: Code Completion; Pre-training; BERT; Transformer

目　录

插图清单

表格清单

第一章　前言

1.1　研究背景及意义

代码补全是目前集成开发环境中不可或缺的功能，可以有效帮助程序员提高开发效率，同时避免程序员产生简单的开发错误如错字。以目前的集成开发环境来说，大多数静态语言如 Java、C++ 等能在编译时确定数据类型和代码运行顺序的语言可以有效的实施代码补全；相反对于动态语言如 Javascript、Python 等在运行前能随意改动代码架构和运行顺序的语言，集成开发环境无法有效预测下个单词，而这也是本次论文中需要实验的部分。

现有的代码补全工具大多都是基于统计语言模型。统计语言模型就是表示语言基本单位的分布函数，描述了该语言的基于统计生成的规则。一般来说，任何一个自然语言的单词量都不是一个模型能够涵盖的，所以将该语言拆分为最小单位单词来计算，而该分布函数就是每个单词出现概率的乘积。由于统计语言模型的理论，该计算概率只考虑该单词和上文的概率，故无法准确对动态语言代码补全，需要有更加周全、考虑上下文的模型来支撑代码补全——即 BERT[1]，一种抛弃传统网络使用 Transformer[3]结构的模型，也是本次论文的理论支撑。

BERT 采用预训练和微调来适应多种不同的下游任务，与目前主流的代码补全模型都不相同。本次论文目的便是探讨预训练对代码补全的影响，对比使用预训练和微调的模型如 BERT 与没有使用预训练，直接对模型进行目标训练任务的模型有何影响以及背后原因，在本次论文中皆有相关实验和探讨。

1.2　代码补全使用模型现状与比较

vanilla LSTM[4-5]：2005 年由 Alex Graves 提出的双向长短期记忆神经网络 BLSTM (Bidirectional Long-Short Term Memory)，是一种基于 RNN 的循环神经网络，通过在细胞状态上的添加、去除和更新来达到保留记忆，是目前主流使用的 LSTM。

Pointer Mixture Network[6]：2015 年 Vinyals 等提出的 Pointer Networks，打破传统 seq2seq 中无法适应输入序列长度变化的问题；于 2018 年由 Jian Li 等人提出的将 Pointer Networks 运用在代码补全上，有别于传统的 Pointer Network，Jian Li 将一个标准的 RNN 和 Pointer Network 混合构成 Pointer Mixture Network，其中 RNN 部分 (attention LSTM) 能借由预先设置好的语料库来进行单词预测；对于 Pointer 组件根据学习到的权重指向最有可能的上文。最终在连接两个输出前会被转换器来调整规格，而后模型能通过学习选择出要挑选哪个预测结果。

Byte Pair Encoding based Neural Language Model (BPE NLM)[7]：1994 年 BPE 数据压缩概念提出，能有效的将大型语料库压缩以减少计算机负载和提高计算效率，通常会对 BPE 压缩设置目标，达成目标则停止压缩迭代达到最佳性能。在 2020 年由 Rafael 提出的一种大型开源词典的自然语言模型，利用 BPE 算法确保小规格的单词并能成功预测语料库外的单词。

Transformer-XL[8]：2019 年由 Zihang Dai 等提出的对于 Transformer 的变形；针对 Transformer 可能产生的三个问题: 上下文碎片、推断速度慢和长期依赖缺失。其中针对前两个问题，Transformer-XL 提供片段递归机制；针对长期依赖缺失则采用相对位置编码而非绝对位置编码。片段递归机制是指在训练过程中，Transformer-XL 的上一个片段状态会被缓存下来并在计算当前段时重复使用上个时间片的状态。同理由于 Trannsformer-XL 相比 Transformer 直接重复使用上个时间片的状态而不是从头计算，简化推理过程的结果是非常可观的。对于相对位置编码则有别于绝对位置编码直接将位置编码加入到输入层，它是把位置编码加入到了 slef-attention 内部，这也使得预测结果更加准确。

Code Understanding and Generation pre-trained Language Model (CugLM)[2]：是一个多任务预训练学习模型,在预训练过程中同时训练三个模型或任务：Masked bidirectional Language Modeling(MLM), Next Code segment Predicting(NCP) 和 Unidirectional Language Modeling(ULM)；其中 MLM 针对标识符进行预测，屏蔽代码中的标识符，训练目标为预测出正确位置上所屏蔽的标识符；NCP 目标是了解代码片段间的关系，为此 NCP 预训练出单词间的前后关系；ULM 是单向语言模型，只对该位置之前 (左) 的上文进行编码，类似于 SLM 中的 N-grams 方法。预训练后对代码补全任务进行微调 (直接应用预训练模型并通过微调预训练模型的参数使模型适应后续任务)，首先预测出单词类型，再预测单词。该模型

对 Java 和 TypeScript 程序代码进行预训练并相较于 BPE NLM 和 Pointer Mixture Network 有着更加优异的预测结果。CugLM 的主要贡献有 (1) 目前效果达到最高水平，基于 Transformer 架构的预训练语言模型并使用在代码补全 (2) 考量标识符类型预测并取得些许成功，这在当前的代码补全领域中是完全领先的 (3) 对比最新语言模型，在代码补全领域中取得最佳的实验结果

1.3 论文的主要工作与组织架构

本文大致分为五章：

第一章前言，主要包括当前代码补全的研究背景和主流模型，介绍模型使用理论和模型架构，并描述本次论文的主要工作。

第二章背景知识和相关技术概述，描述本次论文中使用模型牵涉到的理论知识，对 BERT 底层 Transformer 架构详细介绍，对 Transformer 使用的 Self-Attention 机制做简单描述；介绍首个使用预训练在 NLP 任务上的 GPT，并介绍半监督学习架构的意义与预训练微调任务。最后介绍 BERT 整个模型架构及流程，预训练任务目标和微调训练任务目标的阐述，及 BERT 的输入输出和训练目的做简要说明。

第三章模型设计与实现，详细介绍 CugLM 模型架构及核心代码，预训练和微调任务代码和相关说明，及本人在本次论文中的工作从数据预处理到模型编写及所有代码的设计与实现。

第四章实验，首先介绍实验环境和数据获取，后介绍两个实验，包括实验参数和实验结果，最后根据实验结果综合理论知识给出相应探讨。

第五章总结与展望，对本次研究工作进行探讨与总结，同时对本文未来优化部分进行阐述。

第二章　背景知识和相关技术概述

此部分包含撰写论文所需的知识储备工作，包括涉及的相关知识点及使用到的理论，后续将直接引用不再详细说明。

2.1　Transformer

2.1.1　Transformer Encoder

Transformer 由两个组件组成，Encoder 和 Decoder，其中 BERT 模型使用到 **Transformer** 模型中的 **Encoder** 部分，下图2.1左是 Encoder 组成结构，包含两个子层（Multi-Head Attention 和 Feed-Forward Networks），分别对两个子层使用到的机制和计算过程进行介绍，以便理解本项目使用到的代码架构。另外还能发现残差网络[9]的 Short-cut 机制在 Transformer 中的运用，避免了梯度消失的问题，同时使用归一化防止梯度爆炸，实现良好的权重更新。

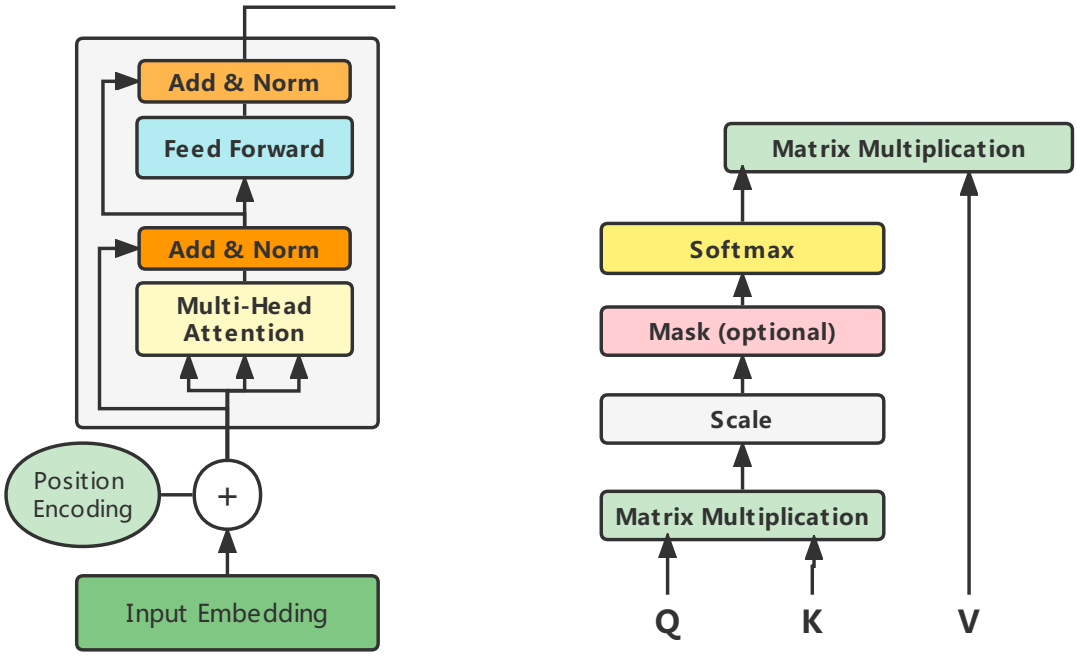

图 2.1　Transformer Encoder(左) 和 Scaled Dot-Product Attention(右)

2.1.2 Transformer Decoder

在 2.2 中介绍的 GPT 使用 Transformer Decoder 的变形。Transformer Decoder 相较于 Encoder 多了一层 Masked Multi-Head Attention 和最终的线性层和 $Softmax$ 层。其中 Masked Multi-Head Attention 层屏蔽的是未来时刻的信息，即可以理解为计算当前解码内容和已经解码内容 (前面解码器的输出) 之间的关系。而第二层 Multi-Head Attention(也可以称为 cross-attention) 是计算当前解码内容和编码得出的特征向量之间的关系。需要注意的是，两者在构造上并无区别，由于 Multi-Head Attention 输入的差异导致两者名称不同。下图 2.2是 Transformer Decoder 结构，可以看出编码输出作为解码器中第二层 Multi-Head Attention 层的输入，而解码器中第一层 Masked Multi-Head Attention 只计算当前和前面解码器的内容。

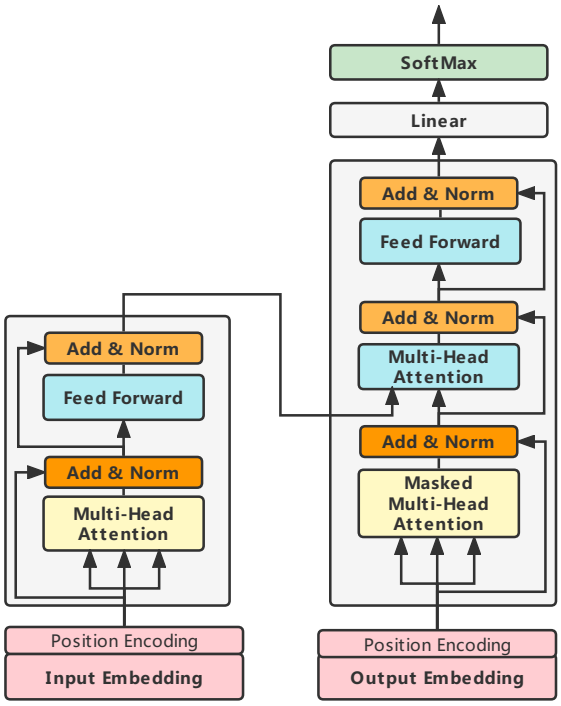

图 2.2 Transformer Encoder(左) 和 Transformer Decoder(右)

2.1.3 Scaled Dot-Product Attention(Self-Attention)

Multi-Head Attention 由多个 **Self-Attention** 组成，首先需要理解 **Self-Attention** 机制。每个单词 *token* 先转化为嵌入向量 x，同时存在三个权值矩阵 W_q，W_k, W_v

用作生成三个特殊的向量 q, k, v ，即：

$$q = x \times W_q, k = x \times W_k, v = x \times W_v$$

然后计算每个 *token* 的 *score* ，代表这个 *token* 的 q 向量和 k 向量的相似度，用作后续推测相似的结果。*score* 计算公式为：

$$score = q \cdot k$$

score 计算完后需要收缩，因为 Transformer 采用的 Scaled Dot-Product Attention 使用点乘进行相似度计算，当 d_k 量级大时容易产生巨大的 *score*，故对 *score* 归一化：

$$score = score/\sqrt{d_k}$$

句子中每个单词 *token* 都会产生一个 *score* ，将这些 *score* 作为 $Softmax$ 函数输入，得到概率分布（权重）后对每个输入向量 v 评分，最后将句子中所有单词评分相加得到最终结果 z：

$$v = Softmax(score) \times v$$

$$z = \sum v$$

2.1.4 Multi-Head Attention

根据实验结果[3]发现，相较于单个 Attention，多个 Attention 线性执行后效率更高，在实验结果中使用 8-head，根据 Attention Head 数量计算 d_k 和相应参数（q, k, v），根据以下公式改变模型参数：

$$d_k = d_v = d_{model}/head$$

将 Attention 输出 z 连接后与全连接层 W 相乘得出最终输出 Z，而不同 Attention 中使用不同的权值矩阵，下图为 **Multi-Head Attention** 示意图。

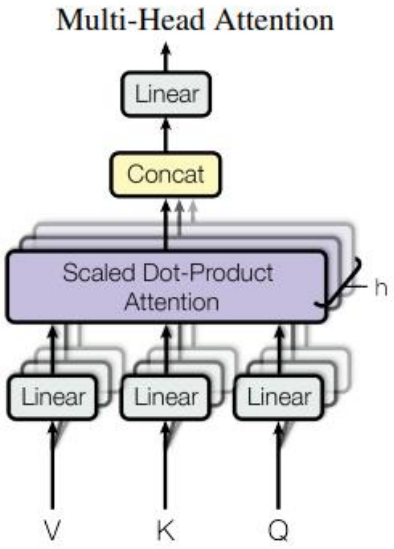

图 2.3 Multi-Head Attention

2.1.5 Position-wise Feed-Forward Network

在 **Multi-Head Attention** 子层后还添加了一层 **Feed-Forward** 层，包含两个线性变换，中间使用 $ReLU$ 激活函数。$Feed-Forward$ 是一个两层的全连接层，若是模型维度 $d_{model} = 512$，则 $Feed-Forward$ 中间隐藏层单元有 $d_{ff} = 2048$ 个。

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

2.2 GPT

GPT[10],(Generative Pre-Training)，是一个半监督学习的自然语言处理模型，包含两个步骤：无监督学习的预训练和监督学习的微调。由于目前 NLP 中能用的数据大多都是未标注的语料，无法有效的进行监督学习，所以 GPT 使用预训从大量预料中学习共性[11]使得模型具备预料的隐含特征。再使用少量标注数据进行监督学习，能快速帮助模型适应不同的任务。

2.2.1 GPT Architecture

如图 2.4左所示，GPT 使用 12 层单向 Transformer Decoder 的变种结构。由于没有 Encoder 输出，所以去除了 Decoder 第二个子层 cross-attention，只保留了

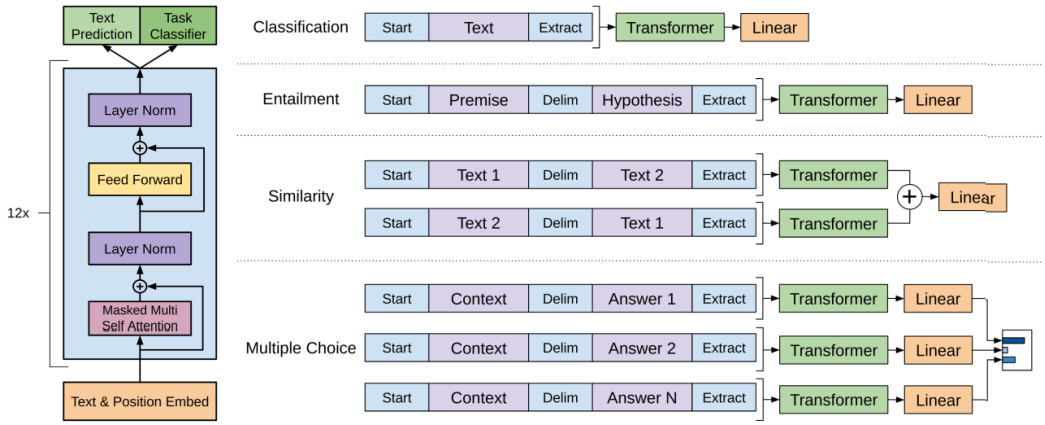

图 2.4 Transformer 架构和训练目标 (左) 为了适应不同下游任务的输入数据转换 (右)

Masked Multi-Head Attention。GPT 基本模型使用 12 个注意力头，768 个隐藏节点和 Feed-Foward 中间层的 3072 个隐藏节点，最大序列长度同样设置为 512。而图 2.4右表示不同下游特定任务微调时的不同输入处理方法，其中文本分类直接在预训练模型上微调即可。文本相似度任务则分别将两个序列输入，通过模型输出两个序列的特征向量，再加入线性层；关于文本蕴含任务，将前提和假设序列拼接，中间通过一个特殊符号 $ 来表示分隔符；关于问答任务，给定上下文文本 c 和问题 q，以及一组问题的答案 a_k，将上下文和问题答案拼接成一个序列 $[c\ q\ \$\ a_k]$ 如图所示，将多个序列输入预训练模型后经过 $Softmax$ 层得到概率分布。

2.3 BERT

BERT,Bidirectional Encoder Representations from Transformers, 是一个预训练的语言表征模型，与其余语言表征模型不同，BERT 可以在缺乏人工标注数据集的情况下训练深层双向的表征。同时 BERT 可适应多种下游任务包括智能问答和词语分类等 11 种不同的 NLP 任务，进而通过对模型的微调来完成对不同下游任务的支持。

BERT 包含了两个训练目标[12]：**MLM**(Masked Language Model) 和 **NSP**(Next Sentence Prediction)，MLM 任务通过预测屏蔽词来包含文本上下文的表征，而非

传统 SLM(Statistical Language Model)[13]只计算单向从左到右的表征。除此之外，NSP 任务用作预测句子与句子间的联系，表示句子间的表征。

2.3.1 BERT Architecture

BERT 基本就是一个特殊的 Transformer Encoder，里面包括多个子层 (cross-attention 和 Feed-Forward)，根据 BERT 模型量级不同有不同数量的子层 ($BERT_{BASE}$12 个;$BERT_{LARGE}$24 个)，通过堆叠形成 BERT 模型。

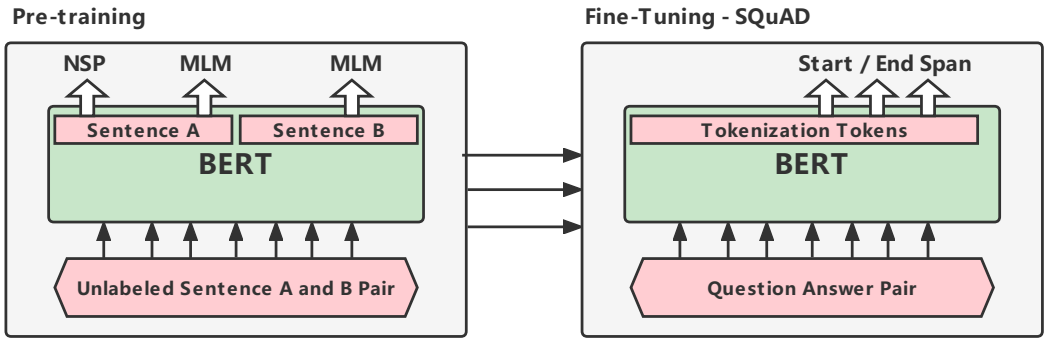

图 2.5 BERT 预训练与微调整体架构

BERT 分为两步骤: 预训练和模型微调，使用未标注的数据进行预训练后，使用标注数据作为微调模型的输入。根据图 2.5所示，不同下游任务 BERT 预训练模型几乎相同，仅仅在微调上实现有些许差异，这也是 BERT 能很好适应多种下游任务的因素。

2.3.2 BERT Input Representation

输入表征包含三个嵌入层[12, 14]：单词、段落和位置。首先对于单词，使用 WordPiece[1]将输入单词对应为 id 作为单词嵌入层。同时对句子添加特殊标注：[CLS] 表示每个输入序列头;[SEP] 表示句子间的分隔符，在后续 MLM 和 NSP 训练目标中会使用到。而对于段落嵌入则根据嵌入 [SEP] 规则同时生成，最后再直观根据单词位置生成位置嵌入层，最终如图 2.6所示。

[1]Google’s neural machine translation system: Bridging the gap between human and machine translation

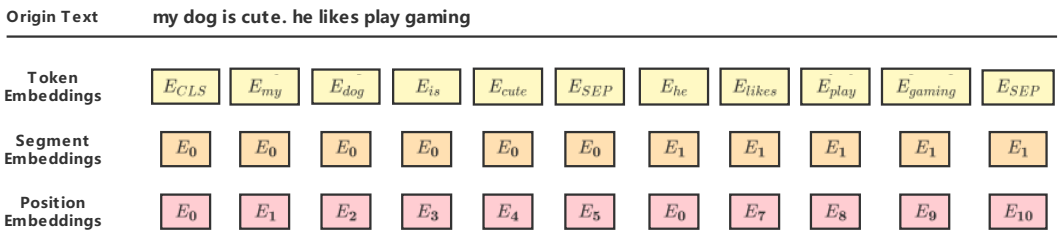

图 2.6 BERT 输入表示层

2.3.3 BERT Pre-training Task

为了支持多种不同的下游任务，BERT 在预训练时设置两个训练任务 MLM 和 NSP，其中 MLM 是 BERT 能够突破传统单向语言模型的关键；NSP 为了支持如智能问答或自然语言推理等任务，诸如此类任务都是基于句子间的联系做出的推断，而 NSP 可以有效的学习句子联系的特征。

2.3.4 Masked LM (MLM)

简单来说，MLM 是在输入序列中挑选几个单词屏蔽，通过训练预测出屏蔽前的正确单词，由于这并非如 SLM 使用到文本的单向关系而是根据屏蔽词上下文进行预测，代表着预测结果肯定比单向预测更加准确。在微调中并没有使用到屏蔽词，也代表着预训练模型与微调间存在误差，为了减轻这个误差，BERT 有其屏蔽词规则：挑选 15% 单词进行屏蔽，其中 (1)80% 替换为 [MASK] (2)10% 保持为原单词 (3)10% 挑选随机单词替换，防止预训练模型结果偏向屏蔽词，下表2.1为不同屏蔽词规则实验结果。观察 MNLI[15](自然语言推断,Multi-Genre Natural Language Inference) 和 NER(命名实体识别,Named Entity Recognition) 任务的微调结果可以得出策略 1 在蕴含关系推断中表现最好。

	Masking Rates			**Dev Set Results**	
	Mask	Origin	Random	MNLI:Fine-tune	NER:Fine-tune
Strategy 1	80%	10%	10%	84.2	95.4
Strategy 2	100%	0%	0%	84.3	94.9
Strategy 3	80%	0%	20%	84.1	95.2
Strategy 4	80%	20%	0%	84.4	95.2
Strategy 5	0%	20%	80%	83.7	94.8
Strategy 6	0%	0%	100%	83.6	94.9

表 2.1 不同屏蔽策略下的结果

2.3.5 Next Sentence Prediction (NSP)

用作提供所有下游任务中需要的句子间关系特征，而特殊标注 [CLS] 用作表示句子头部。在 NSP 任务中，挑选一对句子 (A,B) 训练，其中 50% 在原文本中 B 是 A 的下个句子，而 50% B 不是 A 的下一个句子。

2.4 本章小结

本章主要介绍论文使用到的技术、模型及相关理论；预训练模型的训练目标和训练任务；Transformer 架构和内部模块使用的算法和理论等。

第三章　模型设计与实现

3.1　模型设计

本次使用模型基于 BERT，采用预训练和微调两步骤的半监督自然语言模型。其中预训练任务选择 CugLM 的三个任务：Masked bidirectional Language Model (**MLM**)，Next Code segment Predicting(**NCP**) 和 Unidirectional Language Model(**ULM**)。以下是选用这三个作为预训练任务的原因：NCP 能帮助模型找出句子间的共性，ULM 能够找出单词与单词上文之间的共性，而 MLM 则是进一步考虑单词、单词类型与上下文，找出更加高层隐含的联系。

模型相较目前代码补全方法 BPENLM[7]和 Pointer Mixture Network[6]有着显著进步。对比当前 SLM 将大量软件仓库作为语料库，本模型不使用静态嵌入层作为输入，而是考虑代码上下文而做出预测，同时预测类型，对于模糊词汇和同名不同义的单词也能有效区分，提高准确性。

3.1.1　模型架构

如图3.1所示，由 L 层 Transformer Layer 组成，同时根据不同预训练目标 Transformer Layer 分为双向和单向，MLM 和 NCP 为双向；ULM 为单向。与 BERT 相同，代码分为预训练和微调，其中预训练 MLM、NCP 和 ULM 三个任务, 微调部分则包含 UMLM 和 ULM 两个任务。

3.1.2　预训练流程

a) Masked bidirectional Language Modeling: 基于 CugLM MLM 任务实现，仅在屏蔽词上有所约束，CugLM 的 MLM 任务只屏蔽拥有类型标注的标识符，因为在代码补全中认为仅仅预测标识符是最有效益的。

b) Next Code segment Predicting: 基于 CugLM NSP 任务实现，相较于 BERT 是依照标点符号进行分割，CugLM 是依照代码行分割段落。无论是 NSP 或者

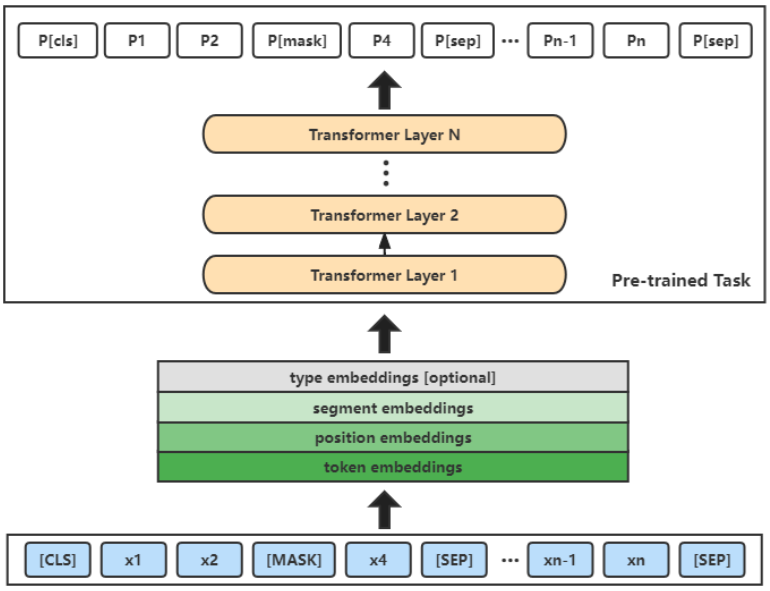

图 3.1 模型简要架构

NCP 的目的都是为了获得句子间或代码段落间的联系特征，在代码补全任务中是非常需要学习代码行间的关系，因此 NCP 不可或缺。同样 50% 是真正隔行，50% 随机行进行预训练。

c) Unidirectional Language Modeling: 为了支持更多自然语言任务，添加单向从左至右的预测任务。在输入 Transformer Encoder 堆叠层前，对输入层进行屏蔽，如图 3.2中 Self-Attention Masks 所示，最后根据 $Softmax$ 后取概率分布最高的词当作预测结果。

3.1.3 微调流程

a) Unidirectional Masked Language Modeling: 与预训练中 MLM 任务不同，UMLM 只基于屏蔽词的上文进行预测。同样设置 Self-Attention Mask 屏蔽输入，使得屏蔽词只能依靠上文进行推断。在 UMLM 中所有带有类型标注的标识符都会被屏蔽，而非取部分输入。与 BERT 不同，在预测部分分为两步骤，通过预测屏蔽词类型，进而预测屏蔽词，能有效提升预测准确率。

b) Unidirectional Language Modeling: 与预训练中 ULM 任务相同，故不再赘述。

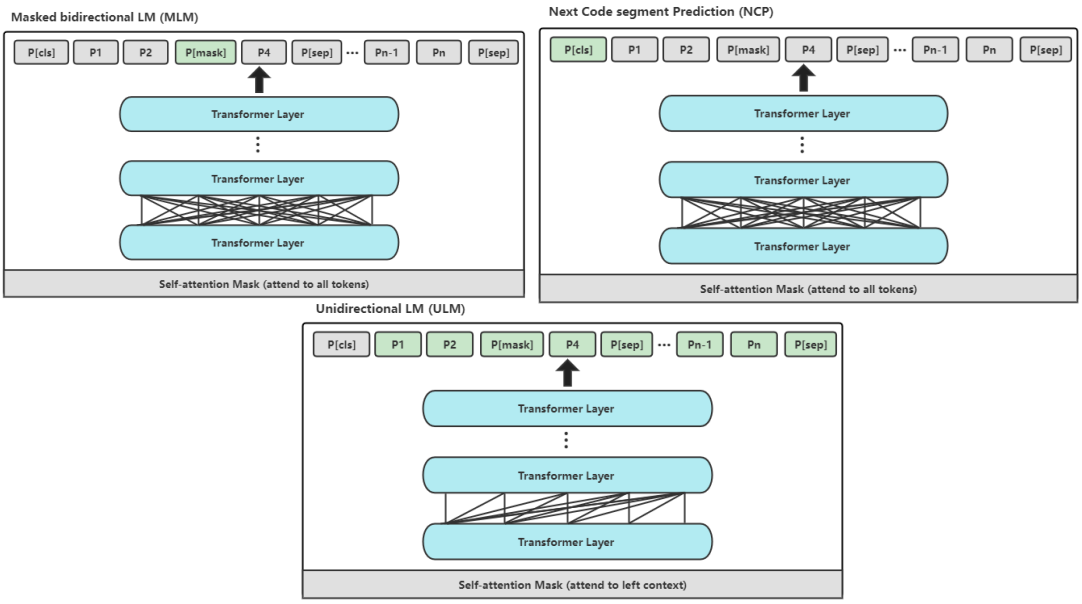

图 3.2 预训练任务

Input = [CLS] public void setTextDirection (int textDirection) {
[SEP] this . mTextDirection = textDirection ; }
Label = 1

Input = [CLS] public void setTextDirection (int textDirection) {
[SEP] this . request = request ;
Label = 0

图 3.3 NCP 任务输入例子，标签 1 表示为真实隔句。

3.2 预处理代码设计与实现

3.2.1 数据获取实现

根据预下载仓库列表将远程仓库下载到本地进行数据处理，故编写一个脚本进行批量下载。仓库列表每行表示为一个仓库，格式为 user_repository，故将其转换为 git clone 形式并指定本地下载目录，区分训练集和评估集。由于 Git 指令经常产生如连接超时等错误，故在代码中需要捕获并处理这些异常。

数据获取脚本代码如下3.5，首先规定训练集和评估集比例，生成 git clone 指令并使用 os 库执行批量执行命令，同时捕获异常。再根据下载完的项目生成预处理所需要的文件 (projects.json 和 dict.json)。前者保存所有下载项目，后者保

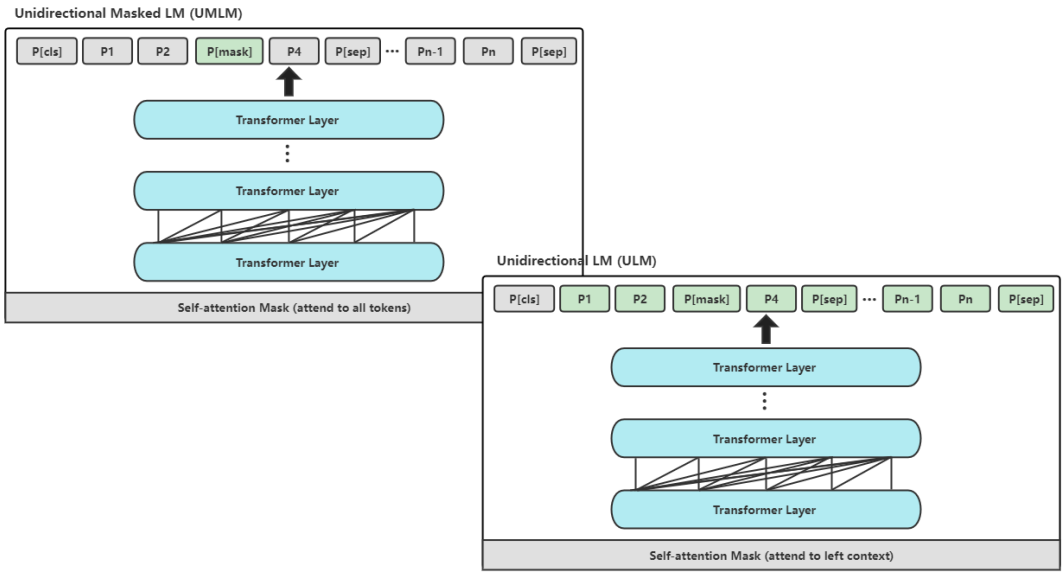

图 3.4 微调训练任务

存下载项目中所有的 JAVA 文件，用于后续预处理时只对 JAVA 文件分词。

3.2.2 预处理设计

如图3.6数据预处理工作主要分为两部分：**生成预处理文件**和**生成模型样例文件**。首先，由于 CugLM 模型输入需要符合其二进制输入文件格式，故需要按照流程采取较为复杂的数据预处理流程。使用 Java 第三方库，按照代码行对初始代码分词，获得分词类型后保存为两个文件 **token_corpus** 和 **type_corpus**。同时根据上述两个文件，生成分词词典和类型词典，用作后续模型内的映射索引：**vocab_token** 和 **vocab_type**。

在生成样例文件部分，首先根据不同句子对生成样例，样例内包含特征序列。同时在生成样例时，进行屏蔽词计算和插入特殊标注词 ([CLS],[SEP]..etc)，模型最终根据样例文件进行计算。

3.2.3 预处理实现

根据数据获取时产生的所有项目列表和项目中所有的 JAVA 文件列表进行遍历，并对每个 JAVA 文件分行并使用分词工具[1]分词并产生分词后的中间文件，格式如下图3.7。

[1]javalang:http://github.com/c2nes/javalang

```python
def download(repo: str):
    try:
        eval_projects, train_projects = balance_train_eval_projects(repo)
    except FileNotFoundError as f:
        exit(f)

    train_clone_url_list, train_url_list = transfer2url(train_projects)
    eval_clone_url_list, eval_url_list = transfer2url(eval_projects)
    try:
        for url in train_clone_url_list:
            project_name = url.split('/')[-1][:-4]
            os.system('git clone ' + url + ' ../train/' + project_name)
        for url in eval_clone_url_list:
            project_name = url.split('/')[-1][:-4]
            os.system('git clone ' + url + ' ../eval/' + project_name)
    except BaseException as e:
        print(e)
        exit()

    train_url_list = get_downloaded_repos(0)
    eval_url_list = get_downloaded_repos(1)

    # data preparation for create_data_corpus.py
    create_input_projects(train_url_list, True, False)
    create_input_projects(eval_url_list, False, True)
    create_project_dict(train_url_list, True, False)
    create_project_dict(eval_url_list, False, True)
```

图 3.5 数据获取核心代码

同时除了分词的中间文件，还生成分词类型中间文件，用以生成输入类型张量。有了这两个分词文件便可根据这两个文件生成单词库，用作映射单词索引，同时为了防止计算内存不够和去除噪声，对单词库词频进行限制，将单词库大小控制在 50000。使用正则表达式对单词出现次数排序，取高频的 50000 个剩余的舍弃，最终生成单词库文件 vocab_token.txt。

完成单词库和分词文件后，生成符合 BERT 模型输入实例，每个实例都包含一个句子对，即模型输入时处理的最小单位。先是读取分词文件中每一个文件作为处理单位 (在分词文件中不同文件中间有空行) 并生成 all_documents 和 all_type_documents 两个列表包含所有文件行，再去除文件中空行，并将两个列表按照种子随机打乱排序，这里两个列表是一样的随机排序，保证在后续映射不偏差。然后根据不同文件生成实例，注意每个文件包含多个实例 (一个文件应当包含 n 个实例，n 表示该文件中行数)。

接下来的生成实例算法有些复杂，就根据概括图3.8细化讲解。首先遍历文件的所有行，如果指定行 $Line_i$ 长度 L_i 小于目标长度 L_{target} (默认为 512，但是有 10% 几率当作小于 512 随机数的长度) 则直接舍弃不生成该行实例；如果行长度 L_i 大于目标长度 L_{target} 则进行后续计算。由于需要一个句子对生成实例

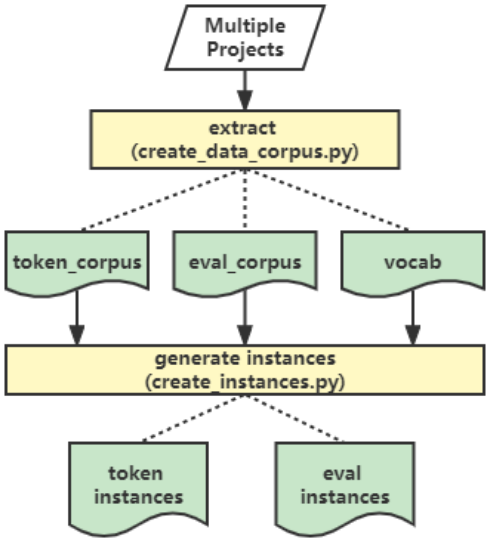

图 3.6 预处理工作概述

```
["package", "android", ".", "app", ";"]
["import", "droidsafe", ".", "runtime", ".", "*", ";"]
["import", "droidsafe", ".", "helpers", ".", "*", ";"]
["import", "droidsafe", ".", "annotations", ".", "*", ";"]
["import", "java", ".", "util", ".", "List", ";"]
["import", "android", ".", "content", ".", "ActivityNotFoundException", ";"]
["import", "android", ".", "content", ".", "ComponentName", ";"]
["import", "android", ".", "content", ".", "ContentResolver", ";"]
["import", "android", ".", "content", ".", "Context", ";"]
["import", "android", ".", "content", ".", "DialogInterface", ";"]
["import", "android", ".", "content", ".", "Intent", ";"]
["import", "android", ".", "content", ".", "pm", ".", "ResolveInfo", ";"]
["import", "android", ".", "database", ".", "Cursor", ";"]
["import", "android", ".", "graphics", ".", "Rect", ";"]
["import", "android", ".", "net", ".", "Uri", ";"]
["import", "android", ".", "os", ".", "Bundle", ";"]
["import", "android", ".", "os", ".", "Handler", ";"]
["import", "android", ".", "os", ".", "RemoteException", ";"]
["import", "android", ".", "os", ".", "ServiceManager", ";"]
["import", "android", ".", "text", ".", "TextUtils", ";"]
["import", "android", ".", "util", ".", "Log", ";"]
["import", "android", ".", "view", ".", "KeyEvent", ";"]
["public", "class", "SearchManager", "implements", "DialogInterface", ".", "OnDismissListener", ",", "DialogInterfa
["@", "DSGeneratedField", "(", "tool_name", "=", "\"Doppelganger\"", ",", "tool_version", "=", "\"2.0\"", ",", "ger
["private", "static", "final", "boolean", "DBG", "=", "false", ";"]
```

图 3.7 分词中间文件

(NCP 任务需要)，故要挑选其余行 $Line_j$ 与指定行 $Line_i$ 组成句子对。按照随机行几率 (50%)，若是随机行则随机在同个文件中挑选一个行作为下一行，故句子对可以表示为：50%(L_i,L_j),1<=j<=n、50%(L_i,L_{i+1})。

选取完句子对后，插入和替换特殊标注，如在句首插入 [CLS]，在两个句子间插入 [SEP] 和将不在单词库中的单词替换为 [UNK]，最后会得到两个列表分别表示句子的单词索引序列和类型索引序列。将这两个序列根据屏蔽规则进行屏蔽，替换 [MASK] 表示该单词屏蔽和返回屏蔽位置列表，最终将屏蔽前和屏蔽后的所有序列初始化实例，即该实例中包含句子对所有表示序列。

下图3.9为特殊标注词插入和替换部分代码，由于单词列表有词频限制，出

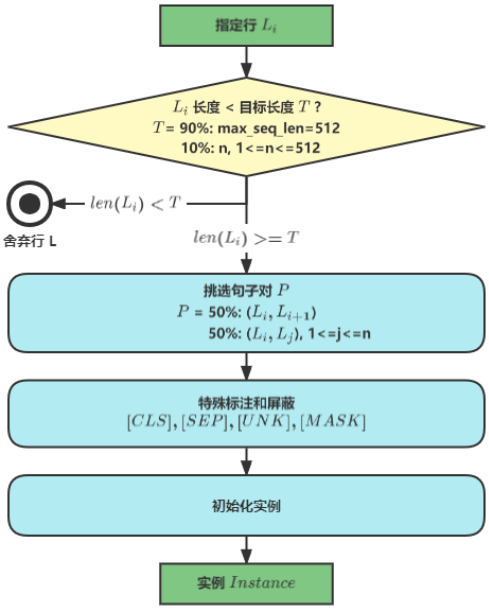

图 3.8 实例生成代码流程

现次数为 1 或低频的单词不在单词列表中，故会有单词没有索引的情况，此时该单词替换为 [UNK]。

```
tokens = []
types = []
segment_ids = []
tokens.append("[CLS]")
types.append("[CLS]")
segment_ids.append(0)

for token in tokens_a:
    if token in vocabs:
        tokens.append(token)
    else:
        tokens.append('[UNK]')
    segment_ids.append(0)
for t in types_a:
    types.append(t)

tokens.append("[SEP]")
types.append("[SEP]")
```

图 3.9 特殊标注词处理代码

由每个文件中的每行生成的实例列表堆叠而成并写入到实例文件中，作为模型输入。值得注意的是，由于后续实验需要，将训练集划分为 10%-90%，故在此以文件个数作为最小单位划分 10%-90% 的数据集并生成对应实例文件，若要更为准确应该要以句子对个数作为依据。

形成的实例列表再根据重复因子 (dupe_factor) 决定需要重复几次数据，默认重复 5 次。在写入实例文件步骤中，为了减少实例文件大小，将实例中的序列映射为单词列表中的索引，同时也是为了后续方便计算。为了压缩，将该实例转为二进制字符串写入，最终生成二进制实例文件。

3.2.4 模型类设计与实现

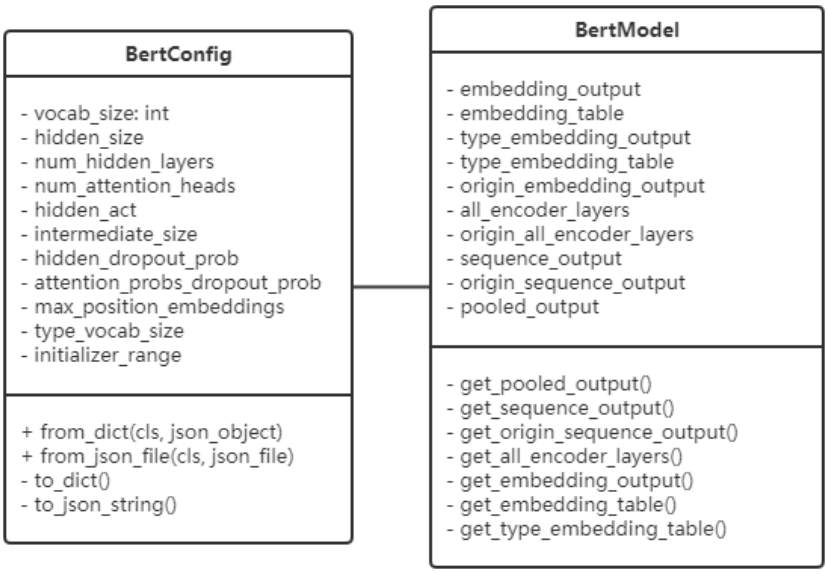

图 3.10 预训练配置类和模型类图

配置类 BertConfig 接受 JSON 数据输入，成员变量对应配置文件 bert_config。模型类中首先设计嵌入层，包含单词嵌入层 embedding_table 和类型嵌入层 type-_embedding_table。每个隐藏节点都参与计算，所以需要扩建维度来完成嵌入层，最终单词嵌入层和类型嵌入层维度为 [50000,512]，其中 50000 是单词数量上限，512 是隐藏节点个数。设计完嵌入层后仍需要设计输入序列张量 embedding_output，此张量表示分词后对应单词嵌入的 ID 序列，维度为 [2,512,512]，其中 2 是设置的 batch 大小；第一个 512 是序列长度上限，超出 512 单词则舍弃不计算；第二个 512 同样是隐藏节点个数，表示每个隐藏节点都参与计算。

输入序列张量是模型用于计算的主要数据，需要同时包含类型和位置的隐含特征，故通过张量相加达成隐含特征目的，如下图3.11代码。将类型序列张量 token_token_embeddings 和位置序列张量 position_embeddings 表示为符合输入序列张量维度的张量后，张量相加表示隐含属性。

```
if use_token_type:
  if token_type_ids is None:
    raise ValueError("`token_type_ids` must be specified if"
                     "`use_token_type` is True.")
  token_type_table = tf.get_variable(
      name=token_type_embedding_name,
      shape=[token_type_vocab_size, width],
      initializer=create_initializer(initializer_range))
  flat_token_type_ids = tf.reshape(token_type_ids, [-1])
  one_hot_ids = tf.one_hot(flat_token_type_ids, depth=token_type_vocab_size)
  token_type_embeddings = tf.matmul(one_hot_ids, token_type_table)
  token_type_embeddings = tf.reshape(token_type_embeddings,
                                     [batch_size, seq_length, width])
  output += token_type_embeddings

if use_position_embeddings:
  assert_op = tf.assert_less_equal(seq_length, max_position_embeddings)
  with tf.control_dependencies([assert_op]):
    full_position_embeddings = tf.get_variable(
        name=position_embedding_name,
        shape=[max_position_embeddings, width],
        initializer=create_initializer(initializer_range))
    position_embeddings = tf.slice(full_position_embeddings, [0, 0],
                                   [seq_length, -1])
    num_dims = len(output.shape.as_list())
    position_broadcast_shape = []
    for _ in range(num_dims - 2):
      position_broadcast_shape.append(1)
    position_broadcast_shape.extend([seq_length, width])
    position_embeddings = tf.reshape(position_embeddings,
                                     position_broadcast_shape)
    output += position_embeddings
```

图 3.11 输入序列张量蕴含类型张量和位置张量

输入序列张量处理完后，需要设计 Transformer 层，并将输入序列张量作为模型输入到 Transformer Encoder 层，而 Transformer Encoder 层则表示为 all-_encoder_layers。Transformer 最后一层输出表示为 sequence_output，用于计算池化层。此处池化层根据 BERT 模型并非采用最大化池化层，而是默认序列首个单词蕴含隐藏状态，所以使每个序列首个单词代表池化后的值，而此池化层在代码中表示为 pooled_output，池化层代码如下3.12：

```
with tf.variable_scope("pooler"):
  first_token_tensor = tf.squeeze(self.sequence_output[:, 0:1, :], axis=1)
  self.pooled_output = tf.layers.dense(
      first_token_tensor,
      config.hidden_size,
      activation=tf.tanh,
      kernel_initializer=create_initializer(config.initializer_range))
```

图 3.12 池化层算法：序列首个单词代表池化后的值

3.2.5 Transformer 层设计与实现

设计模型内部类时要注重数据维度，有时要缩放数据或将数据平坦用于计算，有时要还原数据初始维度，故首先设计调整数据维度的函数。在计算中，只牵涉到 n-D 与 2-D(n>2) 维度间的转换，设计一个从高维到 2 维的函数 reshape_to_matrix() 和一个还原数据初始维度的函数 reshape_from_matrix()，用于在模型转换数据维度中使用，代码在图3.13。在高维度转变为低维度时，将数据平坦并保留最后一个维度，如输入序列张量 [batch, seq, hidden] 转变为 [batch * seq, hidden]。

```
def reshape_to_matrix(input_tensor):
  """Reshapes a >= rank 2 tensor to a rank 2 tensor (i.e., a matrix)."""
  ndims = input_tensor.shape.ndims
  if ndims < 2:
    raise ValueError("Input tensor must have at least rank 2. Shape = %s" %
                     (input_tensor.shape))
  if ndims == 2:
    return input_tensor

  width = input_tensor.shape[-1]
  output_tensor = tf.reshape(input_tensor, [-1, width])
  return output_tensor

def reshape_from_matrix(output_tensor, orig_shape_list):
  """Reshapes a rank 2 tensor back to its original rank >= 2 tensor."""
  if len(orig_shape_list) == 2:
    return output_tensor

  output_shape = get_shape_list(output_tensor)

  orig_dims = orig_shape_list[0:-1]
  width = output_shape[-1]

  return tf.reshape(output_tensor, orig_dims + [width])
```

图 3.13 两个维度变换函数

Transformer 层函数按照图2.1左的 Encoder 结构设计。第一层输入是维度变换后的输入序列张量 T_i(shape=[batch * seq, hidden])，调用 attention_layer 函数[2]后得到 Multi-Head Attention 模块输出 T_{after}(shape=[batch * seq, hidden])，先经过一层全连接层保存数据特征，然后 Dropout 防止过拟合，再将 $T_i + T_{after}$ 相加进行标准化。标准化后的结果需要再通过 Feed-Foward 层，包含两层全连接层，第一层映射个数为 2048，第二层输出映射回 512 隐藏节点；第一层先执行激活函数 GeLU，第二层线性变换。同样 Feed-Foward 输出需要经过 Dropout 和标准化后才得到 Transformer 层的输出，同时这层输出也作为下一层输入。

[2]此函数来自 https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py

```
attention_head = attention_layer(
    from_tensor=layer_input,
    to_tensor=layer_input,
    attention_mask=attention_mask,
    num_attention_heads=num_attention_heads,
    size_per_head=attention_head_size,
    attention_probs_dropout_prob=attention_probs_dropout_prob,
    initializer_range=initializer_range,
    do_return_2d_tensor=True,
    batch_size=batch_size,
    from_seq_length=seq_length,
    to_seq_length=seq_length)
attention_heads.append(attention_head)

with tf.variable_scope("output"):
  attention_output = tf.layers.dense(
      attention_output,
      hidden_size,
      kernel_initializer=create_initializer(initializer_range))
  attention_output = dropout(attention_output, hidden_dropout_prob)
  attention_output = layer_norm(attention_output + layer_input)
```

图 3.14 Multi-Head Attention 模块核心代码

3.3 预训练代码设计与实现

3.3.1 模型配置

下图 3.15是模型配置文件，使用激活函数 GeLU，每层 Dropout 概率是 0.1，有 512 个隐藏节点构成 6 个隐藏层，拥有 6 个注意力头。设置词典上限为 50000，且最长序列为 512，超过 512 长度则丢弃计算。

```
{
  "attention_probs_dropout_prob": 0.1,
  "directionality": "bidi",
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 512,
  "initializer_range": 0.02,
  "intermediate_size": 2048,
  "max_position_embeddings": 512,
  "num_attention_heads": 8,
  "num_hidden_layers": 6,
  "pooler_fc_size": 512,
  "pooler_num_attention_heads": 6,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 64,
  "pooler_type": "first_token_transform",
  "type_vocab_size": 2,
  "vocab_size": 50000,
  "vocab_type_size": 50000
}
```

图 3.15 模型配置文件

3.3.2 预训练主要架构

预训练代码分为**加载数据、配置运行时环境、定义模型架构、数据匹配和运行模型**。部分代码参考 BERT 模型，加入了类型嵌入层支持类型预测。使用 `Tensorflow 1.14` 机器学习库中的 `tensorflow.estimator.Estimator` 训练和预估模型。

加载数据

加载配置文件 **bert_config** 生成 `BertConfig` 实例和预处理生成的词典 **vocab_token**。其中词典内容为了方便转换而选择保存为 JSON 格式，包含标识符和所有符号，且按照词频进行排序，如图3.16所示，可以看到出现频率最高的是符号”.”。

```
[".", "(", ")", ";", ",", "=", "{", "}", "public", "import", "return", "int", "if",
"@", "String", "new", "<", ">", "final", "+", "static", "void", "null", "private",
"[", "]", "0", "this", "org", "com", "i", "==", "Override", "java", "1", "class",
"boolean", "!=", ":", "util", "android", "else", "-", "false", "throws", "true",
"tool_name", "tool_version", "generated_on", "\"Doppelganger\"", "\"2.0\"",
"google", "get", "e", "List", "?", "package", "for", "append", "super", "&&", "value",
"protected", "length", "Object", "add", "common", "!", "apache", "extends", "*",
"internal", "2", "result", "assertEquals", "case", "size", "++", "throw", "T",
"DSGenerator", "hash_original_method", "hash_generated_method", "Integer", "env",
"long", "R", "api", "data", "name", "context", "espertech", "esper", "core", "try",
"double", "equals", "eclipse", "model", "Test", "catch", "||", "->", "io", "client",
"IOException", "toString", "float", "break", "id", "Exception", "index",
"DSGeneratedField", "hash_original_field", "hash_generated_field", "x", "put",
"ArrayList", "protobuf", "3", "c", "type", "view", "byte", "Map", "DSCat",
"instanceof", "b", "hyracks", "implements", "out", "s", "v", "compiere", "event",
"View", "eu", "esig", "europa", "dss", "lang", "key", "che", "System", "Builder", "a",
"net", "y", "File", "ctx", "os", "assertThat", "4", "log", "getName", "&", "r", "p",
"buf", "\"\"", "junit", "assertTrue", "t", "input", "node", "/", "text", "start",
"javax", "list", "of", "Context", "Log", "j", "elki", "A", "5", "content",
"DSComment", "asterix", "base", "println", "DSSafe", "info"]
```

图 3.16 词频排序后的词典

配置运行时环境

包含使用 GPU/TPU、是否使用分布式策略运行代码、日志配置等等。由于本次论文只使用单 GPU 运行，故不使用分布式策略运行模型，而使用 `tf.distribute.OneDeviceStrategy()` 替代运行策略。其他配置如每 8 步记录日志，每 1000 步设置保存点和设置模型保存路径。

```
if FLAGS.use_tpu and FLAGS.tpu_name:
    tpu_cluster_resolver = tf.contrib.cluster_resolver.TPUClusterResolver(
        FLAGS.tpu_name, zone=FLAGS.tpu_zone, project=FLAGS.gcp_project)

is_per_host = tf.contrib.tpu.InputPipelineConfig.PER_HOST_V2

dist_strategy = tf.distribute.OneDeviceStrategy("/device:GPU:0")

log_every_n_steps = 8
run_config = RunConfig(
    train_distribute=dist_strategy,
    eval_distribute=dist_strategy,
    log_step_count_steps=log_every_n_steps,
    model_dir=FLAGS.output_dir,
    save_checkpoints_steps=FLAGS.save_checkpoints_steps)
```

图 3.17 运行时环境代码

定义模型架构

使用 `tensorflow.estimator.Estimator` 运行模型，模型架构则定义在 `model_fn` 函数中，完全根据官方 API 文档[3]进行编写。`model_fn` 输入参数为特征和标签映射过后的数据，而这些数据向量作为三个预训练任务的输入，最终计算结果返回。三个训练任务中 MLM 和 NCP 参考 BERT 代码，ULM 则将输入向量通过注意力掩码后使得输入变为单向，对应代码如3.18。

```
model_fn = model_fn_builder(
    bert_config=bert_config,
    init_checkpoint=FLAGS.init_checkpoint,
    learning_rate=FLAGS.learning_rate,
    num_train_steps=FLAGS.num_train_steps,
    num_warmup_steps=FLAGS.num_warmup_steps,
    use_tpu=FLAGS.use_tpu,
    use_one_hot_embeddings=FLAGS.use_tpu,
    word2id=token_word2id)

estimator = Estimator(
    model_fn=model_fn,
    config=run_config, )
```

图 3.18 定义模型架构代码

数据匹配

`Estimator` 规定训练时调用输入匹配函数 `input_fn` 用以匹配数据。首先定义特征，其次将数据随机打乱防止模型拟合，最后将数据与特征映射为一连串向量支持模型计算。

[3]https://tensorflow.google.cn/versions/r1.15/api_docs/python/tf/estimator/Estimator

运行模型

调用 `Estimator.train(input_fn)` 训练模型。

3.3.3 预训练模型函数设计与实现

模型函数 model_fn 设计的意义是为了传递优化器和参数给 Estimator，而模型函数在官方文档中也有明确定义返回类型 EstimatorSpec，格式如图3.19，故围绕这个类型进行设计。首先需要计算总体损失，而总体损失为所有训练任务损失相加，其中预训练含有三个训练任务，所以先设计获取三个训练任务损失的函数。

```
tf.estimator.EstimatorSpec(
    mode, predictions=None, loss=None, train_op=None, eval_metric_ops=None,
    export_outputs=None, training_chief_hooks=None, training_hooks=None,
    scaffold=None, evaluation_hooks=None, prediction_hooks=None
)
```

图 3.19 类型 EstimatorSpec 参数格式

对于 MLM 的损失计算，相关代码如下 3.20。先将模型输出通过由激活函数

```
def get_masked_lm_output(bert_config, input_tensor, output_weights,
output_type_weights, positions,
                         label_ids, masked_type_ids, label_weights):
    input_tensor = gather_indexes(input_tensor, positions)
    with tf.variable_scope("transform"):
        input_tensor = tf.layers.dense(
            input_tensor,
            units=bert_config.hidden_size,
            activation=modeling.get_activation(bert_config.hidden_act),
            kernel_initializer=modeling.create_initializer(
                bert_config.initializer_range))
        input_tensor = modeling.layer_norm(input_tensor)

    with tf.variable_scope("cls/predictions"):
    …
    type_numerator = tf.reduce_sum(type_label_weights * type_per_example_loss)
    type_denominator = tf.reduce_sum(type_label_weights) + 1e-5
    type_loss = type_numerator / type_denominator

    with tf.variable_scope("cls/predictions/addtype"):
    …
    numerator = tf.reduce_sum(label_weights * per_example_loss)
    denominator = tf.reduce_sum(label_weights) + 1e-5
    loss = numerator / denominator

return (loss, type_loss, per_example_loss, log_probs)
```

图 3.20 计算 MLM 训练任务损失

GeLU 创建的全连接层，再分别计算类型预测损失和单词预测损失。损失计算方

法使用适合多分类任务的**交叉熵**计算，公式为 -YLog(Y_{pred})。Y 张量表示正确用 one-hot 向量表示，大小为 50000 表示所有分类；Y_{pred} 表示预测结果，取对数后与 Y 相乘得到交叉熵损失。另外，根据 BERT 理论模型得知，MLM 训练任务需包含两步骤，先预测类型，而后将类型与单词向量合并预测单词，能获得比独立预测单词与类型更好的训练效果。

对于 ULM 的损失计算，将模型输出 (这是不屏蔽计算后的输出) 通过激活函数组成的全连接层后，对嵌入层矩阵相乘后做 Softmax 获得分布概率。同样使用交叉熵[16]计算损失。结果 Y 同样拥有 50000 个分类，故 one-hot 向量长度也相关代码如下 3.21。

```
def get_lm_output(bert_config, input_tensor, output_weights, label_ids,
label_weights):
    input_tensor = tf.reshape(input_tensor, [-1, bert_config.hidden_size])
        with tf.variable_scope("transform"):
            input_tensor = tf.layers.dense(
                input_tensor,
                units=bert_config.hidden_size,
                activation=modeling.get_activation(bert_config.hidden_act),
                kernel_initializer=modeling.create_initializer(
                    bert_config.initializer_range))
            input_tensor = modeling.layer_norm(input_tensor)
        output_bias = tf.get_variable(
            "output_bias",
            shape=[bert_config.vocab_size],
            initializer=tf.zeros_initializer())
        logits = tf.matmul(input_tensor, output_weights, transpose_b=True)
        logits = tf.nn.bias_add(logits, output_bias)
        log_probs = tf.nn.log_softmax(logits, axis=-1)

        label_ids = tf.reshape(label_ids, [-1])
        label_weights = tf.reshape(tf.cast(label_weights, tf.float32), [-1])

        one_hot_labels = tf.one_hot(
            label_ids, depth=bert_config.vocab_size, dtype=tf.float32)

        per_example_loss = -tf.reduce_sum(log_probs * one_hot_labels, axis=[-1])
        numerator = tf.reduce_sum(label_weights * per_example_loss)
        denominator = tf.reduce_sum(label_weights) + 1e-5
        loss = numerator / denominator

    return (loss, per_example_loss, log_probs)
```

图 3.21　计算 ULM 训练任务损失

对于 NCP 的损失计算则简单许多，one-hot 数据类型深度为 2，表示标签只有 2 个分类，即是下个句子和不是下个句子。同样是将 Softmax 后的结果映射到多分类区间，取对数后与 one-hot 向量相乘获得 NCP 任务损失，相关代码如下 3.22。

回到模型函数，获得计算训练任务损失后，判断当前模型处于训练还是评估阶段返回不同数据。根据图 3.19 得知，预训练阶段只需要返回损失和训练优

```
def get_next_sentence_output(bert_config, input_tensor, labels):
    with tf.variable_scope("cls/seq_relationship"):
        output_weights = tf.get_variable(
            "output_weights",
            shape=[2, bert_config.hidden_size],

initializer=modeling.create_initializer(bert_config.initializer_range))
        output_bias = tf.get_variable(
            "output_bias", shape=[2], initializer=tf.zeros_initializer())

        logits = tf.matmul(input_tensor, output_weights, transpose_b=True)
        logits = tf.nn.bias_add(logits, output_bias)
        log_probs = tf.nn.log_softmax(logits, axis=-1)
        labels = tf.reshape(labels, [-1])
        one_hot_labels = tf.one_hot(labels, depth=2, dtype=tf.float32)
        per_example_loss = -tf.reduce_sum(one_hot_labels * log_probs, axis=-1)
        loss = tf.reduce_mean(per_example_loss)
    return (loss, per_example_loss, log_probs)
```

图 3.22 计算 NCP 训练任务损失

化器种类；而评估阶段仍需要对损失和概率分布进一步处理成不同训练任务的正确率和损失。

```
masked_lm_accuracy = tf.metrics.accuracy(
    labels=masked_lm_ids,
    predictions=masked_lm_predictions,
    weights=masked_lm_weights)
masked_lm_mean_loss = tf.metrics.mean(
    values=masked_lm_example_loss, weights=masked_lm_weights)

next_sentence_accuracy = tf.metrics.accuracy(
    labels=next_sentence_labels, predictions=next_sentence_predictions)
next_sentence_mean_loss = tf.metrics.mean(
    values=next_sentence_example_loss)

lm_accuracy = tf.metrics.accuracy(
    labels=lm_target_ids,
    predictions=lm_predictions,
    weights=lm_weights)
lm_mean_loss = tf.metrics.mean(
    values=lm_example_loss, weights=lm_weights)

return {
    "masked_lm_accuracy": masked_lm_accuracy,
    "masked_lm_loss": masked_lm_mean_loss,
    "next_sentence_accuracy": next_sentence_accuracy,
    "next_sentence_loss": next_sentence_mean_loss,
    "lm_accuracy": lm_accuracy,
    "lm_loss": lm_mean_loss,
}
```

图 3.23 评估阶段计算正确率和损失

在计算正确率部分，将预测的序列和正确的序列对比，使用 `tf.metrics.accuracy`[4]函数返回各个训练任务的正确率；在计算损失部分则是计算需要考虑权重序列，使用 `tf.metrics.mean`[5]计算加权平均值。在代码 3.23 中分别计算三个训练目标的正确率以及加权后的损失，最终作为返回参数 eval_metric_ops。

[4]https://tensorflow.google.cn/versions/r1.15/api_docs/python/tf/metrics/accuracy

[5]https://tensorflow.google.cn/versions/r1.15/api_docs/python/tf/metrics/mean

3.3.4 预训练数据匹配函数设计与实现

数据匹配函数的意义是作为小批量提供训练输入数据的函数，根据官方文档设计需要返回包含 (特征，标签) 的元组，利于在后续模型中辨别特征与标签间的维度。

```
output_classes = {dict: 10}
{'input_ids': <class 'tensorflow.python.framework.ops.Tensor'>,
 'input_mask': <class 'tensorflow.python.framework.ops.Tensor'>,
 'input_type_ids': <class 'tensorflow.python.framework.ops.Tensor'>,
 'lm_input_ids': <class 'tensorflow.python.framework.ops.Tensor'>,
 'lm_target_ids': <class 'tensorflow.python.framework.ops.Tensor'>,
 'masked_lm_ids': <class 'tensorflow.python.framework.ops.Tensor'>,
 'masked_lm_positions': <class 'tensorflow.python.framework.ops.Tensor'>,
 'masked_lm_weights': <class 'tensorflow.python.framework.ops.Tensor'>,
 'next_sentence_labels': <class 'tensorflow.python.framework.ops.Tensor'>,
 'segment_ids': <class 'tensorflow.python.framework.ops.Tensor'>}

output_shapes = {dict: 10}
{'input_ids': TensorShape([Dimension(1), Dimension(512)]),
 'input_mask': TensorShape([Dimension(1), Dimension(512)]),
 'input_type_ids': TensorShape([Dimension(1), Dimension(30)]),
 'lm_input_ids': TensorShape([Dimension(1), Dimension(512)]),
 'lm_target_ids': TensorShape([Dimension(1), Dimension(512)]),
 'masked_lm_ids': TensorShape([Dimension(1), Dimension(30)]),
 'masked_lm_positions': TensorShape([Dimension(1), Dimension(30)]),
 'masked_lm_weights': TensorShape([Dimension(1), Dimension(30)]),
 'next_sentence_labels': TensorShape([Dimension(1), Dimension(1)]),
 'segment_ids': TensorShape([Dimension(1), Dimension(512)])}

output_types = {dict: 10}
{'input_ids': tf.int32, 'input_mask': tf.int32, 'input_type_ids': tf.int32, 'lm_input_ids': tf.int32,
'lm_target_ids': tf.int32, 'masked_lm_ids': tf.int32, 'masked_lm_positions': tf.int32, 'masked_lm_weights':
tf.float32, 'next_sentence_labels': tf.int32, 'segment_ids': tf.int32}
```

图 3.24 特征张量的类、维度和类型

在设计深度学习模型架构时首先要注意的就是维度的变换，故先介绍所有特征及其维度大小：`input_ids` 为添加屏蔽词后的索引张量，维度 [1, 512]，1 为一批处理几个数据，512 为最长序列长度；`lm_input_ids` 为初始的没有屏蔽词的索引张量，维度 [1,512]；`input_mask` 为是否屏蔽计算张量，值为 0 表示不计算，维度 [1,512]；`segment_ids` 为段落张量，值为 0 表示第一段，值为 1 表示第二段，维度 [1,512]；`masked_lm_positions` 表示屏蔽词位置，维度 [1,30]，由于设置每个序列中最多产生 30 个屏蔽词，故特征序列长度设置 30；`masked_lm_ids` 表示屏蔽词原本代表的索引，维度 [1,30]；`input_type_ids` 为屏蔽词原本代表的单词类别索引，由于只有屏蔽词计算进行类别预测，故只有屏蔽词类别索引而不用保存整个序列类别索引，维度 [1,30]；`next_sentence_labels` 表示是否为下一句，值为 1 表示真实隔句，维度 [1,1]。所有特征及维度如图3.24:

3.4 微调代码设计与实现

本次代码参考 BERT 模型，其预训练部分代码与微调代码相差不大，故微调代码设计与实现部分介绍两者不同之处，其余基础架构如加载数据及配置运行时环境等可以参考章节3.3.2。在前面章节3.1提到，预训练拥有三个训练任务或说训练目标，分别是 MLM、ULM 和 NCP；微调则只需要训练两个任务 UMLM 和 ULM。故在设计模型函数时不同，去除 NCP 任务，同时双向 MLM 变为单向 MLM，需要在预测时避免预测单词下文参与计算，而损失计算则是两个训练任务损失之和。代码3.25去除掉计算 NCP 任务的损失，最终损失只有 UMLM 和 ULM 参与计算。

```
(masked_lm_loss, masked_lm_type_loss,
 masked_lm_example_loss, masked_lm_log_probs) = get_masked_lm_output(
    bert_config, model.get_sequence_output(), model.get_embedding_table(),
    model.get_type_embedding_table(),
    masked_lm_positions, masked_lm_ids, masked_type_ids, masked_lm_weights)

(lm_loss, lm_example_loss, lm_log_probs) = get_lm_output(
    bert_config, model.get_origin_sequence_output(), model.get_embedding_table(),
input_target_ids,
    lm_weights)

# (next_sentence_loss, next_sentence_example_loss,
#  next_sentence_log_probs) = get_next_sentence_output(
#     bert_config, model.get_pooled_output(), next_sentence_labels)

total_loss = masked_lm_loss + masked_lm_type_loss + lm_loss
```

图 3.25 微调模型函数：计算两个训练任务

3.5 本章小结

本章首先介绍使用模型及其架构，并分析其预训练流程及微调流程。接着主要介绍所有代码设计与实现，包括数据获取、数据预处理、预训练模块及微调模块。

第四章　实验

设置实验主要目的为探究预训练对代码补全任务效果的影响，故设计两个实验方便探究结果。实验 1 直接设计为有无预训练是否对实验结果有明显的影响，即实验组 (有预训练) 和对照组 (无预训练)，其余实验参数保持一致避免误差和添加其他变因。实验 2 则设计为使用不同数据量对预训练效果有何种影响，和是否存在一个临界值使得预训练效果达到最佳。

4.1　实验环境

GPU：GeForce RTX 2080Ti

CPU：Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz ×8

CUDA: 11.0

Python: 3.6.13

Tensorflow: 1.14.0

4.2　数据获取

数据获取来源于论文中[2]使用的数据集共 9,708 个 Github 远程仓库。本次所有实验取其中的 1,385 个项目作为数据集使用，其中按照 10:1 比例分为训练集和评估集，下图 4.2为运行下载脚本后成功结果的截图。

4.3　实验一

主要对比经过预训练和没有经过预训练模型对代码补全效果的影响。

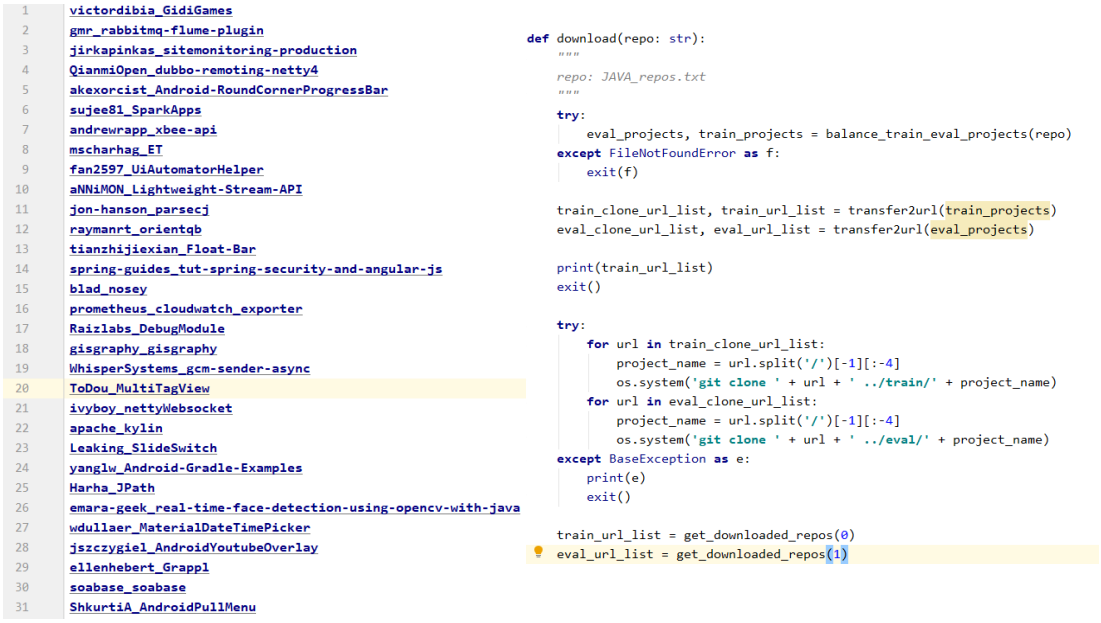

图 4.1 部分仓库列表 (左) 和下载脚本 (右)

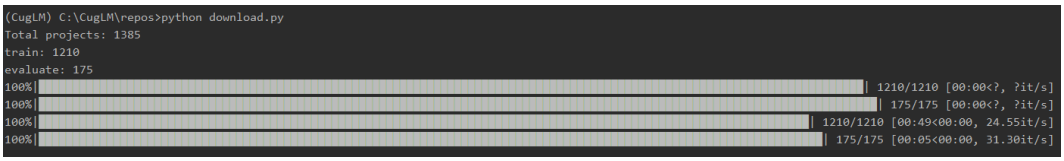

图 4.2 下载脚本成功运行结果截图

4.3.1 实验参数

如下表格 4.1所示，实验使用 6 层 Transformer 作为模型主要架构，拥有 512 个隐藏节点，同时 Attention Head 有别于 BERT 的 8 个，采用了 6 个进行计算。Dropout 每层设置为 0.1 防止数据过拟合，学习率按照论文中使用的 5e-5，词典大小控制在 50,000。

设计两组对照组和实验组，包含 300,000 训练步数和 600,000 训练步数。值得一提的是，组别二中的实验组是采用 600,000 预训练和 300,000 微调训练，而对照组采用 600,000 微调训练，故在表格中呈现为 300k 或 600k。因为预训练过的模型在微调训练步数 300,000 的情况下已经完全收敛，故组别二实验组微调训练步数无需再设置为 600,000。

4.3.2 实验结果

实验组 (EG) 为有预训练；反之对照组 (CG) 没有预训练。下表 4.2为两组实验组和对照组在实验结果方面的展示，因对照组没有预训练故预训练时间为-。

	Group 1	Group 2
Pre-train Train Steps	300,000	600,000
Fine-tune Train Steps	300,000	300k/600k
Activation Function	ReLU	ReLU
Transformer Layer	6	6
Hidden states	512	512
Attention Heads	6	6
Learning Rate	5e-5	5e-5
Dropout Probability	0.1	0.1
Max sequence length	512	512
Vocab Size	50,000	50,000

表 4.1 实验一: 两组对照组与实验组参数配置

	EG1(Group 1)	CG1(Group 1)	EG2(Group 2)	CG2(Group 2)
Pre-train Time	3hr25min	-	6hr6min	-
Fine-tune Time	3hr17min	3hr15min	3hr10min	6hr18min
Evaluate Time	41min	38min	40min	41min
LM accuracy	0.757	0.693	0.783	0.703
Masked LM accuracy	0.283	0.273	0.289	0.274
Loss	7.197	7.684	7.001	7.581
LM Loss	1.835	2.210	1.676	2.013
Masked LM Loss	4.602	4.680	4.560	4.677

表 4.2 实验一: 两组有无预训练的实验结果对比

图 4.3分别为组别一中对照组和实验组的 Loss 曲线图对比，同样都是取微调步骤的 Loss 收敛曲线。橘色曲线为 CG1，采用 300,000 训练步数微调；深红曲线为 EG1，采用 300,000 训练步数预训练和微调。

图 4.4分别为组别二中对照组和实验组的 Loss 曲线图对比，同样都是取微调步骤的 Loss 收敛曲线。橘色曲线为 CG2，采用 600,000 训练步数微调；深蓝色曲线为 EG2，采用 600,000 训练步数预训练和 300,000 微调。

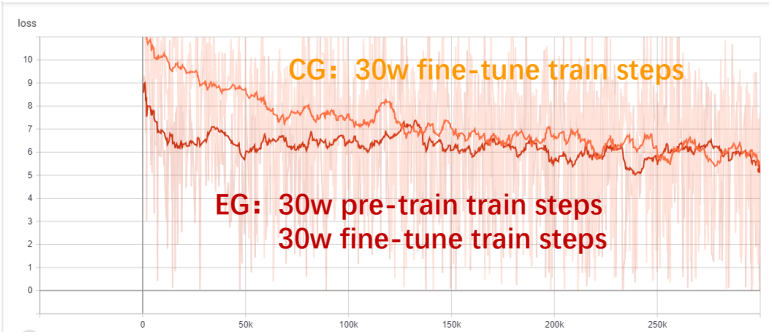

图 4.3 实验一: 组别一微调收敛曲线

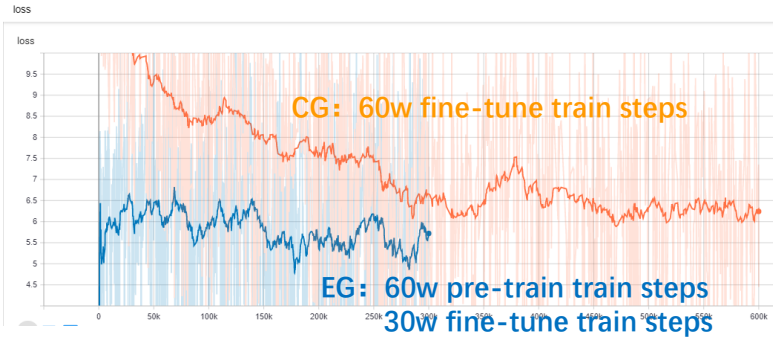

图 4.4 实验一: 组别二微调收敛曲线

4.4 实验二

探讨不同预训练程度对模型结果有何影响，及观察是否存在临界值和逼近收敛临界值的所需时间。设计 10%-100% 的训练集进行预训练，观察 Loss 曲线及其收敛速度。

4.4.1 实验参数

如下图4.5所示，实验同样采用 6 层 Transformer 作为模型主要架构，拥有 512 个隐藏节点。Dropout 每层设置为 0.1 防止数据过拟合，学习率按照论文中使用的 5e-5，词典大小控制在 50,000。不同的是，实验 2 激活函数采用较为稳定的 GeLU，能有效减小误差获得更好的对比效果。

4.4.2 实验结果

实验二中有一个实验组 EG 和九个对照组 CG1-CG9，实验组使用 100% 训练集，而对照组 CG 使用 10%-90% 的训练集用以进一步观察预训练对代码补全的影响，下表为实验结果4.3。

4.5 结果与讨论

根据实验一表格4.2可以得出，无论是组一还是组二，有预训练的模型相比没有预训练的模型效果来的更好，两者在整体模型预测单词正确率相差 6-8%，而在屏蔽词预测准确率来说相差并不大 (1%)，可以直接得出结论——预训练对

Pre-train Train Steps	300,000
Fine-tune Train Steps	300,000
Activation Function	GeLU
Transformers	6
Hidden states	512
Attention Heads	6
Learning Rate	5e-5
Dropout Probability	0.1
Max sequence length	512
Vocab Size	50,000
Optimizer	Adam

图 4.5 实验二模型参数

	Train Dataset	LM accuracy	Masked LM accuracy	Loss
EG	100%	75.7%	28.2%	7.246
CG1	10%	74.6%	28.1%	7.332
CG2	20%	74.9%	28.3%	7.267
CG3	30%	74.9%	28.1%	7.288
CG4	40%	75.0%	28.2%	7.284
CG5	50%	75.1%	28.3%	7.267
CG6	60%	75.1%	28.2%	7.250
CG7	70%	74.9%	28.3%	7.267
CG8	80%	75.0%	28.2%	7.278
CG9	90%	75.1%	28.3%	7.258

表 4.3 实验二: 不同大小数据集是否影响预训练结果

代码补全效果是正面影响，能得出结论——**预训练能提升代码补全任务的准确性**。

我们还能从收敛曲线图 4.3和图 4.4得知，有预训练的模型在 5k-10k 左右微调训练步数时便以达到收敛临界值，表示**有预训练的模型能快速收敛**，这对于越大的模型体现越明显，而这也是预训练的优点。相比之下，没有预训练的模型到训练结束都未能达到有收敛临界值，两者收敛速度相差 80 倍以上。

根据实验二表格4.3可以得出，训练集大小会影响模型正确率，这也侧面显示预训练对代码补全任务有影响。从 CG1 到 CG9 能够看出模型预测正确率是逐步上升，且拥有上限 (75.1%)，这也表示训练集大小对模型准确性是正面影响，**训练集规模越大训练出的模型效果越好，在泛化方面也更加优秀**。关于屏蔽词预测方面，训练集大小并不影响。

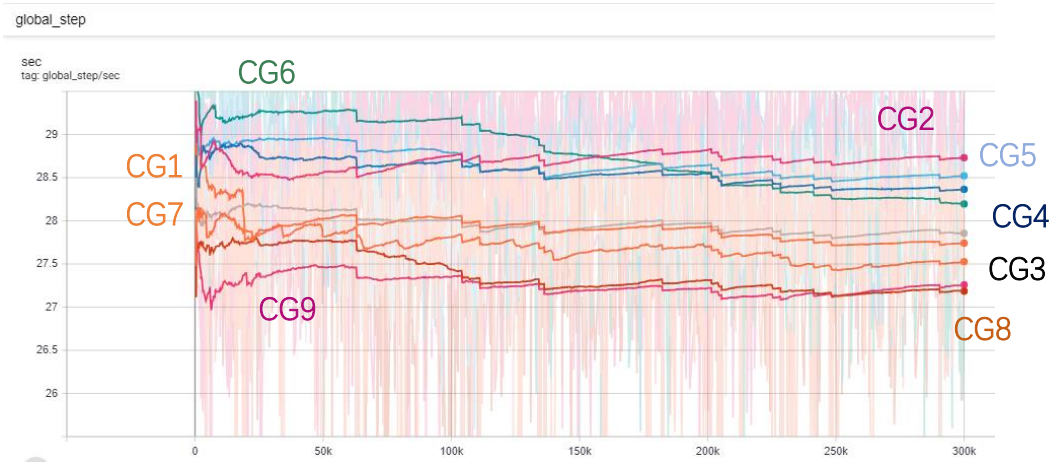

图 4.6 实验 2: 训练时间对比

关于训练时间对比如图 4.6，可以观察出训练集规模越大的模型运行速度越慢，得出训练集大小细微影响训练时间，因此在挑选训练集时也不能追求模型准确率而使用过大的训练集。

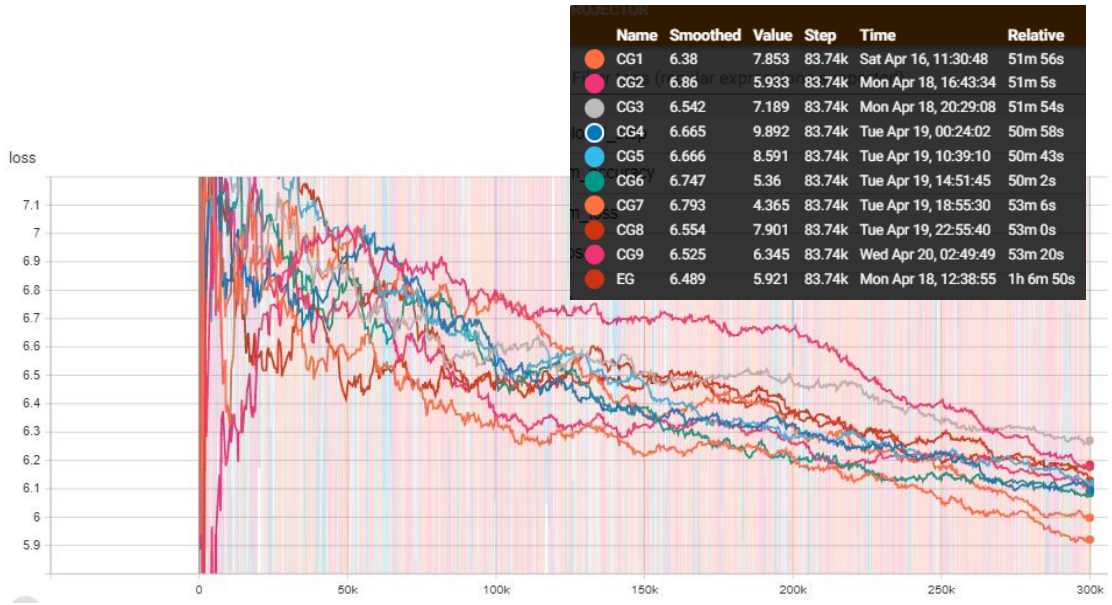

图 4.7 实验 2:Loss 曲线

根据图 4.7得出实验组和对照组的 Loss 收敛曲线，在预定训练步数后都收敛于定值，表示模型正常收敛。再观察不同对照组和实验组的收敛曲线，在暖身时期 (1k)，多数小训练集的对照组 Loss 下降快，曲线抖动幅度大，曲线下降表

示训练集小而模型容易拟合，曲线上升表示 Dropout 步骤使得模型具有泛化能力。训练中期处于各个曲线震荡，而最大的训练集收敛曲线最为平稳，也表示收敛和泛化效果最好。在训练尾部 (300k) 部分能看出训练集越大 Loss 趋近于中值 (6.1)，而根据 9 个对照组最后结果也可以看出，震荡的中间值就是使用 100% 的实验组结果，得出结论——**训练集规模越大泛化效果越好偏差越小**。

第五章　总结与展望

5.1　工作总结

随着软件开发需求日益渐增，要求程序员有着更高的代码开发效率，而协助程序员快速编程的代码补全功能处于重要地位。提升代码补全效率能直接提升编码速度，而预训练便是能有效提高代码补全命中率的方法。预训练泛指使用大量未标注数据或原生数据，提取数据间的共性或隐含特征，使得减轻模型对于特定任务的学习负担。而探讨预训练对代码补全的影响便是本次论文的主要研究方向。

本文第一章主要描述本次研究背景及其意义，介绍代码补全使用模型现状与各自优缺点，及本次论文主要工作组织架构。

第二章描述所需前备知识，包括设计的相关知识点及使用理论。首先介绍打破传统网络结构的 Transformer，其组成模块和使用算法包括自注意力机制、多头注意力机制和全连接前馈网络。介绍半监督学习的自然语言处理模型 GPT，是首个提出半监督概念的模型，也算是 BERT 模型的前身。其模型架构和如何处理特定下游任务输入的方式都影响到后续出现的半监督模型。然后重点介绍 BERT，也是本次论文中参考的模型。与 GPT 不同，BERT 由 12 层 Transformer Encoder 而非 Decoder 堆叠而成。BERT 的输入层表示、数据处理方法与使用到的三个预训练任务目标 MLM、ULM 与 NSP。详细介绍三种任务目标的算法，并在后续设计代码时使用。

介绍完参考模型与相关理论知识后，便开始介绍第三章模型设计与实现。首先介绍选用模型与模型架构包括无监督学习的预训练和监督学习的微调过程，分别描述采用的训练任务目标与在模型中的组成。将工作分为预处理、预训练与微调并分别阐述代码设计与实现。

最后设计实验测试预训练对代码补全任务的影响，设计两个实验分别证明预训练对代码补全有着正面影响和预训练规模对模型的影响与其余预训练相关

结论，在实验部分皆有详细介绍。

5.2 未来展望

主要介绍本次研究中不足与能够进一步优化部分进行讨论：

(1) 在数据集方面并无仔细甄别是否偏向特定项目，如 Android 或 Web 项目开发，导致数据共性偏向特定类型，可能导致泛化效果不优等情况；同时本次研究工作只针对 JAVA 文件进行代码补全，无法支持更多编程语言；训练集规模为论文中使用的 1/4，可能无法获取普遍的共性。

(2) 使用合适的静态分析工具，在类型标注方面使用高层类型 (如 **Keyword**、**Identifier**) 表示而非 JAVA 类 (如 **org.springframework.boot.SpringApplication**) 进行计算，这会导致预测类型时产生较为的模糊结果，根据论文给出的预测类型准确率与本次实验结果中进行对比能得知两者在预测类型正确率上有差距。类型差别如下图 5.1，上面为目前使用的类型标注，下半为论文中使用的细化类型标注。

```
['Modifier','Keyword','Identifier','Separator']
['Modifier','Modifier','Identifier','Identifier','Separator']
['Modifier','Modifier','Keyword','Identifier','Separator','Identifier',
'Identifier','Separator','Separator']
['Keyword','Separator','Identifier','Operator','Null','Separator']
[,'Identifier','Operator','Identifier','Separator','Identifier','Separator',
'Identifier','Separator',
'Identifier','Separator','Separator','Separator','Null','Separator',
'Identifier','Separator','Identifier','Separator','Separator']
['Identifier','Separator','Identifier','Separator','Identifier','Separator',
'Separator']
['Identifier','Separator','Identifier','Separator','Identifier','Separator',
'Identifier','Separator','Separator']
['Identifier','Separator','Identifier','Separator','Separator','Separator',
'Separator']
['Separator']

["_","_","_","_"]
["_","_","android.widget.Toast","_","_"]
,"_","_","_","_","_","_","java.lang.String","_","_","_"]
["_","_","tellh.com.gitclub.common.wrapper.Note.Note.mToast","_","_","_"]
["tellh.com.gitclub.common.wrapper.Note.Note.mToast","_",
"android.widget.Toast","_","android.widget.Toast.makeText","_",
"tellh.com.gitclub.common.AndroidApplication","_",
"tellh.com.gitclub.common.AndroidApplication.getInstance","_","_","_","_","_","_",
"android.widget.Toast","_","android.widget.Toast.LENGTH_SHORT","_","_"]
["tellh.com.gitclub.common.wrapper.Note.Note.mToast","_",
"android.widget.Toast.setText","_","_","_","_"]
["tellh.com.gitclub.common.wrapper.Note.Note.mToast","_",
"android.widget.Toast.setDuration","_","android.widget.Toast","_",
"android.widget.Toast.LENGTH_SHORT","_","_"]
["tellh.com.gitclub.common.wrapper.Note.Note.mToast","_",
"android.widget.Toast.show","_","_","_","_"]
["_"]
```

图 5.1 分词类型标注对比

(3) 运行环境无法匹配论文中使用环境 (使用分布式策略在 8 张显卡上运行或直接在 TPU 运行)，影响运行时间的参数都调为最小才使得模型能够运行，可能导致运行时间成倍增加。

(4) 实验设计能够更好，能挖掘出预训练对下游任务的影响及其背后原因，当前只处于预训练能提升特定任务 (代码补全) 的模型正确率及其收敛曲线，并未深究背后影响的因素。

参考文献

[1] DEVLIN J, CHANG M, LEE K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[C]//BURSTEIN J, DORAN C, SOLORIO T. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, June 2-7, 2019, Volume 1 (Long and Short Papers). Minneapolis, MN, USA: Association for Computational Linguistics, 2019: 4171-4186. DOI: 10.18653/v1/n19-1423.

[2] LIU F, LI G, ZHAO Y, et al. Multi-task Learning based Pre-trained Language Model for Code Completion[C]//35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, September 21-25, 2020. Melbourne, Australia: IEEE, 2020: 473-485. DOI: 10.1145/3324884.3416591.

[3] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is All you Need[C]// GUYON I, von LUXBURG U, BENGIO S, et al. Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017. Long Beach, CA, USA: IEEE Computer Society, 2017: 5998-6008.

[4] HOCHREITER S, SCHMIDHUBER J. Long Short-Term Memory[J]. Neural Comput., 1997, 9(8): 1735-1780. DOI: 10.1162/neco.1997.9.8.1735.

[5] TORTEROLO F, GARBAY C. A Hybrid Agent Model, Mixing Short Term and Long Term Memory Abilities[C]//ASADA M, KITANO H. Lecture Notes in Computer Science: RoboCup-98: Robot Soccer World Cup II: vol. 1604. Paris, France: Springer, 1998: 246-260. DOI: 10.1007/3-540-48422-1_20.

[6] LI J, WANG Y, LYU M R, et al. Code Completion with Neural Attention and Pointer Networks[C]//LANG J. Proceedings of the Twenty-Seventh Interna-

tional Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018. Stockholm, Sweden: ijcai.org, 2018: 4159-4165. DOI: 10.24963/ijcai.2018/578.

[7] KARAMPATSIS R, BABII H, ROBBES R, et al. Big code != big vocabulary: open-vocabulary models for source code[C]//ROTHERMEL G, BAE D. ICSE '20: 42nd International Conference on Software Engineering, 27 June - 19 July, 2020. Seoul, South Korea: ACM, 2020: 1073-1085. DOI: 10.1145/3377811.3380342.

[8] DAI Z, YANG Z, YANG Y, et al. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context[C]//KORHONEN A, TRAUM D R, MÀRQUEZ L. Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, July 28- August 2, 2019, Volume 1: Long Papers. Florence, Italy: Association for Computational Linguistics, 2019: 2978-2988. DOI: 10.18653/v1/p19-1285.

[9] HE K, ZHANG X, REN S, et al. Deep Residual Learning for Image Recognition[C]//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, June 27-30, 2016. Las Vegas, NV, USA: IEEE Computer Society, 2016: 770-778. DOI: 10.1109/CVPR.2016.90.

[10] SUTSKEVER A R K N T S I. Improving Language Understanding by Generative Pre-Training[J]. OpenAI, 2003(09): 22-26.

[11] HAN X, ZHANG Z, DING N, et al. Pre-trained models: Past, present and future[J]. AI Open, 2021, 2: 225-250. DOI: 10.1016/j.aiopen.2021.08.002.

[12] AKBIK A, BLYTHE D, VOLLGRAF R. Contextual String Embeddings for Sequence Labeling[J]., 2018: 1638-1649.

[13] 邢永康, 马少平. 统计语言模型综述[J]. 计算机科学, 2003(09): 22-26.

[14] MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed Representations of Words and Phrases and their Compositionality[C]//BURGES C J C, BOTTOU L, GHAHRAMANI Z, et al. Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013.

Proceedings of a meeting held December 5-8, 2013. Lake Tahoe, Nevada, United States: Association for Computational Linguistics, 2013: 3111-3119.

[15] NANGIA N, WILLIAMS A, LAZARIDOU A, et al. The RepEval 2017 Shared Task: Multi-Genre Natural Language Inference with Sentence Representations[C] //BOWMAN S R, GOLDBERG Y, HILL F, et al. Proceedings of the 2nd Workshop on Evaluating Vector Space Representations for NLP, RepEval@EMNLP 2017, September 8, 2017. Copenhagen, Denmark: Association for Computational Linguistics, 2017: 1-10. DOI: 10.18653/v1/w17-5301.

[16] TANG Q, MA L, ZHAO D, et al. A multi-objective cross-entropy optimization algorithm and its application in high-speed train lateral control[J]. Appl. Soft Comput., 2022, 115: 108151-108152. DOI: 10.1016/j.asoc.2021.108151.

致　　谢

在选择这个毕业设计主题时对机器学习领域可以说是毫无经验，从自行补足机器学习相关基础知识到开始搜集和阅读论文，再到动手设计代码及运行模型可以说是一路坎坷。可一路上遇到许多人许多事帮助我，让我最终完成本次毕业论文撰写。

首先我要感谢葛季栋老师，鉴于老师每周的例行会议，了解每个小组项目进度，并适时督促和给与建议，帮助我们能够按照项目规划，合理的完成毕业设计项目。再者我要感谢一路上给与我帮助的牛长安学长。论文选择、服务器租借和各式数据处理问题都能在与学长讨论后得到解决，同时在项目偏离主题时指导我并与我讨论出如何才能更加贴合主题等等，真的非常感谢！

在生活方面，我也非常感恩辅导员张一品老师。各式招聘信息和申请奖项，担任学生与学校间的桥梁，或者任何学业方面的疑惑和困难老师都能帮助到我，实在感恩。

最后我要感谢我的家人和身边的同学与朋友，是你们陪伴着我大学四年，度过印象深刻且美好的四年，在我情绪低落时拉我一把，与我共同分享喜悦，使我能有勇气去面对各种挑战，谢谢。

在这祝所有人都能积极向上的生活，一生平安！