NiuTensor深度学习框架

Python组件设计与实现

姓 名：侯宝玉

学 号：2101728

班 级：计硕2102班

专 业：计算机科学与技术

目录

[1 实践要求 1](#_Toc89203375)

[2 实践内容 1](#_Toc89203376)

[3 相关技术简介 1](#_Toc89203377)

[3.1 NiuTensor深度学习框架 1](#_Toc89203378)

[3.2 Transformer模型结构 2](#_Toc89203379)

[3.3 开发工具 4](#_Toc89203380)

[Anaconda环境 4](#_Toc89203381)

[Visual Studio 2017编译环境 4](#_Toc89203382)

[接口封装工具 5](#_Toc89203383)

[4 系统设计 5](#_Toc89203384)

[4.1 Python接口封装 5](#_Toc89203385)

[4.1.1 NiuTensor数据类型封装 6](#_Toc89203386)

[4.1.2 NiuTensor功能函数封装 9](#_Toc89203387)

[4.1.2 NiuTensor神经网络模型封装 9](#_Toc89203388)

[4.2 Transformer翻译模型搭建 10](#_Toc89203389)

[5 系统实现 11](#_Toc89203390)

[5.1 系统开发及训练环境 11](#_Toc89203391)

[5.1.1 开发环境 11](#_Toc89203392)

[5.1.2 训练环境 11](#_Toc89203393)

[5.2 Python接口封装 12](#_Toc89203394)

[5.2.1 工程配置 12](#_Toc89203395)

[5.2.2 封装代码具体实现 14](#_Toc89203396)

[6 系统测试 15](#_Toc89203397)

[6.1 系统测试环境 15](#_Toc89203398)

[6.2 Transformer翻译模型测试 16](#_Toc89203399)

[7 心得体会 17](#_Toc89203400)

# 1 实践要求

实现Transformer模型，使用预处理好的IWSLT’14 De-En数据集进行训练，输入德文输出对应的英文。

# 2 实践内容

NiuTensor可以支持C/C++调用方式，但是目前还没有完善的Python接口，而深度学习一般使用的都是Python编程语言，Python接口的实现也是未来NiuTensor进行推广完善的一个方向。

本实验以封装NiuTensor框架的张量计算和神经网络函数为主要目的，拓展NiuTensor框架在Python语言端的使用，为研究者提供调用NiuTensor底层计算函数的接口，方便研究者利用NiuTensor接口函数在Python端搭建所需的语言模型，充分利用NiuTensor框架的灵活易用的优势。

本实验将NiuTensor框架实现于Transformer翻译模型中，使用封装好的函数搭建Transformer模型，并利用预处理好的IWSLT’14 De-En数据集进行相应的性能测试。

# 3 相关技术简介

## 3.1 NiuTensor深度学习框架

NiuTensor是东北大学自然语言处理实验室和小牛团队研发的用于搭建语言模型的张量计算框架，提供了完整的张量定义和计算功能，支持CPU以及GPU设备，是一套轻量级的、能够简易灵活使用的面向中文自然语言处理的深度学习开源框架。相较于TensorFlow、PyTorch等应用广泛的框架，NiuTensor凭借代码高度优化、支持C/C++调用、易于修改、简单小巧、具有丰富的张量计算接口等优点占据一席之地。作为一个专用性的平台，NiuTensor主要针对深度学习中的自然语言处理任务，为研究人员提供可选的框架方案。

NiuTensor平台有专业的研发团队持续维护，应用范围不是仅仅局限在自然语言处理领域，在图像语音处理和其他大数据分析等任务中也是可以应用的，这也是NiuTensor未来发展和优化的一大方向。

由于NiuTensor框架是一个崭新的平台，目前NiuTensor的功能不够完善，业界使用者也较少，可应用的领域范围小，使用的平台环境比较局限。NiuTensor框架目前没有完善的Python接口来支持Python语言的使用，这也是本实验主要解决的问题。

## 3.2 Transformer模型结构

Transformer模型的结构由编码器和解码器组成。编码是将输入序列转换为连续的序列表示，解码过程是将前面的序列ｚ生成对应的输出序列。其中，Transformer解码器由N=6个完全相同的层堆叠而成，每一层都有三个子层，即multi-head self-attention机制、简单的位置全连接的前馈网络和对解码器堆栈的输出执行multi-head attention。对每个子层采用残差连接、层标准化。采用mask机制，防止位置关注到后面的位置。Transformer编码器是由N=6个相同层堆叠而成，但每一层只有两个子层，即multi-head self-attention机制和简单的位置完全连接的前馈网络。在每个编码器中的每个子层后都有残差连接和层正则化的操作。其模型结构如图3.1所示，编码器为左半部分，解码器为右半部分。

图示

描述已自动生成

图3.1 Transformer结构图

位置向量（positional encoding）编码输入序列中每个词的顺序，并对输入的词向量从０开始依次标记。位置向量利用正弦函数和余弦函数，将位置索引为pos的词向量映射成维的位置向量。如式（1）、式（2）所示，pos为位置索引，ｉ为索引对应的向量值。

（1）

（2）

多头注意力机制（multi-headattrntion）是模型的主要模块。注意力机制对源语言元素的Value值进行加权求和，Query和Key被用来计算对应Value的权重系数，Query、Key、Value经过一个线性变换后，被输入到ScaledDot-Product Attention，做h次训练，多头之间参数不能共享，Ｑ、Ｋ、Ｖ进行线性变换的参数不同[14]。将训练后的结果拼接，进行线性变换后，得到的值是多头注意力的结果，如图3.2所示。

图示

描述已自动生成

图3.2 多头注意力机制结构

RNN模型不能并行计算，Transformer模型突破这个限制，并且相比CNN模型，Transformer计算两个位置之间的关联需要的操作次数并不随距离而增长。但是Transformer的缺点也比较明显，它的计算量太大，位置信息利用不明显，无法捕获长距离的信息。另外，多头注意力本身带来了大量的模型，这可能使训练中出现问题，而部署模型、导入庞大数量的参数也需要较高的资源支持。

## 3.3 开发工具

### Anaconda环境

Anaconda是一个开源的Python包、环境管理器，有着conda、Python等180多个科学包及其依赖项，包括Conda、Python以及许多安装好的工具包，比如NumPy（科学运算包）、SciPy（科学计算包）、Matplotlib（绘图工具包）、Scikit-learn（经典机器学习工具包）、Pandas（数据处理和分析工具包）。

### Visual Studio 2017编译环境

Microsoft Visual Studio是美国微软公司的开发工具包系列产品，可用于开发Windows、Web和其他平台的应用，由源代码编辑器、设计器、编译器和调试器组成。本实验使用的是Visual Studio 2017版本，该版本相较之前的版本有着性能和工作效率的提高，可使用Azure开发云应用，有着快速编码、轻松调试和诊断、时常测试的优势，支持Windows应用开发、移动应用开发、跨平台开发、游戏开发、AI开发等高级功能。

### 接口封装工具

pybind11是一个轻量级的可将C++代码暴露给Python调用的C++库。主要用于在已有的C++代码上做拓展。pybind11的语法与Boost.Python类似，但却没有Boost.Python所需的模板技巧以及不必要的对旧版编译器的支持，因而pybind11比Boost.Python更加简洁高效。

pybind11工具包有着兼容性强、可在C++ 中使用lambda表达式、在Python中使用捕获的变量、对函数进行向量化加速、大量使用移动特性、支持Python的切片语法、只需要包含头文件、生成的二进制文件的体积更小、C++中的类型容易进行序列化和反序列化等特点。

# 4 系统设计

## 4.1 Python接口封装

Python封装的主要流程为分析代码、数据类型封装、函数功能封装、语言模型搭建、语言模型测试、语言模型调优。接口封装设计方法如图4.1所示。

图示

描述已自动生成

图4.1 接口封装设计图

Window使用pybind11封装NiuTensor的流程方法如下：

1）利用C++编译工具加载NiuTensor源码，或是编写C++代码；

2）引入pybind11工具，编写Python的入口模块函数，利用pybind11语法封装C++代码的数据类型和功能函数；

3）利用C++编译工具将NiuTensor和封装代码编译为.pyd库文件，在Python端可以直接import定义的模块名。

Linux使用pybind11封装NiuTensor的流程方法如下：

1）利用Cmake命令将NiuTensor生成lib动态链接库；

2）编写pybind11封装代码，引入pybind11的库文件，编写Python的入口模块函数，利用pybind11语法封装C++代码的数据类型和功能函数。

3）利用gcc编译器，引入NiuTensor和pybind11工具包，将封装作用的.cpp文件编译为.so的动态库文件，在Python端可以直接import定义的模块名。

### 4.1.1 NiuTensor数据类型封装

（1）XTensor数据结构

XTensor是NiuTensor定义的张量，是运算的基本单位。表4.1展示了XTensor的成员变量。表4.2展示了XTensor类的部分成员函数的介绍。

表4.1 XTensor成员变量介绍

| 成员变量 | 功能 |
| --- | --- |
| void \* data | 存储张量数据的数组 |
| int devID | 设备ID，>=0表示GPU，-1表示CPU |
| int order | 张量维度 |
| TENSOR\_DATA\_TYPE dataType | 数据元素的数据类型 |
| int unitNum | 数据元素个数 |
| int unitSize | 数据元素大小 |
| float denseRatio | 稠密度，0到1之间的实数 |
| int dimSize[MAX\_TENSOR\_DIM\_NUM] | 维度数组 |
| bool isSparse | 是否为稠密张量 |

表4.2 XTensor成员函数介绍

| 函数 | 功能 | 参数 |
| --- | --- | --- |
| void SetDim(int \* myDimSize) | 设置张量每一维的大小 | myDimSize：张量维度数组 |
| int GetDim(const int dim) | 获取指定维度的大小 | dim：张量维度 |
| int GetSize() const | 用于获取张量元素个数 | N/A |
| void SetZeroAll(Xstream \* stream) | 设置张量所有元素为0 | stream：多线程流 |
| DTYPE Get2D(int ni, int mi) const | 获取二位张量指定行列的单元值 | ni：指定行  mi：指定列 |
| void SetData(  const void \* d, int num, int beg) | 用给定数组给张量赋值 | d：数组地址  num：数组大小  beg：赋值时从张量的开始位置 |
| void Reshape(const int order, const int \* myDimSize) | 重新调整张量维度 | order：张量的维度  myDimSize：张量维度数组 |
| bool Set2D(DTYPE value, int ni, int mi) | 设置二位张量指定行列的单元值 | value：指定单元值  ni：指定行  mi：指定列 |
| bool Add2D(DTYPE value, int ni, int mi) | 增加二位张量指定行列的单元值 | value：指定单元值  ni：指定行  mi：指定列 |
| bool Resize(const int myOrder,  const int \* myDimSize,  const TENSOR\_DATA\_TYPE myDataType,  const float myDenseRatio) | 将张量设置为指定大小 | myOrder：张量维度  myDimSize：张量维度数组  myDataType：张量的数据类型  myDenseRatio：稠密度 |
| bool Resize(const XTensor \* myTensor) | 将张量设置为指定张量的大小 | myTensor：参考张量 |

（2）XNet数据结构

Xnet是基于XTensor的网络结构，用于保存搭建网络的张量节点、训练过程的梯度、输出节点、输入节点等信息，如表4.3所示。

表4.3 XNet成员变量介绍

| 成员变量 | 功能 |
| --- | --- |
| unsigned int id | 网络的ID序号 |
| TensorList nodes | 网络节点 |
| TensorList gradNodes | 网络梯度张量 |
| TensorList outputs | 网络输出节点 |
| TensorList inputs | 网络输入节点 |
| bool isGradEfficient | 指示网络是否只保留参数张量的梯度 |

Xnet包含的主要成员函数介绍如表4.4所示。

表4.4 XNet成员函数介绍

| 函数 | 功能 | 参数 |
| --- | --- | --- |
| void Clear() | 清空网络结构 | N/A |
| void Backward(XTensor &root) | 反向传播获得梯度 | root：反向传播首节点 |
| void BackwardNode(XTensor \* node, bool isEfficent = false); | 对给定节点进行反向计算 | node：指定节点  isEfficent：指示是否以有效的方式计算反向传播 |
| void Traverse(XTensor &root) | 通过深度优先搜索遍历网络并找到拓扑顺序（Tarjan算法） | root：遍历网络首节点 |
| void Dump(FILE \* file) | 存储网络信息 | file：信息存储的目标文件 |
| void SetGradEfficientFlag(bool flag = true) | 设置梯度有效标志 | flag：梯度是否有效标志 |
| void MakeEfficientNet() | 为每个节点生成梯度有效标志 | N/A |
| void ClearGrad(XTensor \* node) | 清空未用节点的梯度信息 | node：想清除梯度信息的节点 |
| void ShowNetwork(FILE \* file, XTensor \* node) | 显示网络拓扑信息 | file：储存网络信息的文件  node：网络节点 |

### 4.1.2 NiuTensor功能函数封装

NiuTensor的数据结构的封装是最为基础的环节，当基本单元被封装完毕后，对基本单元的操作和运算同样非常重要。本实验从NiuTensor的代数运算、张量存取、数学运算、数据移动、规约操作、形状转换、排序操作、激活函数八大模块对NiuTensor功能函数封装。

### 4.1.2 NiuTensor神经网络模型封装

Transformer翻译模型的封装涉及Model模型类、Trainer训练类、Translator翻译类、AttEncoder编码器类、AttDecoder解码器类等数据结构的封装，之后本实验利用封装的接口搭建Transformer翻译模型实例时，将编码器和解码器的每个过程函数在Python端重新搭建，组装成新的编码器和解码器类，供研究者灵活使用。表4.5介绍了Model模型类的主要成员变量。

表4.5 Model模型成员变量介绍

| 成员变量 | 功能 |
| --- | --- |
| int devID | 设备id |
| AttEncoder\* encoder | 编码器 |
| AttDecoder\* decoder | 解码器 |
| Output\* outputLayer | 输出层 |
| bool isLM | 是否为语言模型 |
| bool isMT | 是否是机器训练 |
| bool useFP16 | 模型是否以FP16数据类型运行 |
| int nhead | 注意力模型的头数 |
| int shareAllEmbeddings | 是否将编码器embeddings传递给解码器 |
| int shareDecInputOutputWeight | 是否将解码器embeddings传递给输出层 |

Transformer模型封装的功能函数除去各个类主要的成员函数外，主要分为位置向量编码、编码器、解码器、输出层、其他功能函数五大模块。位置向量编码主要包含batchLoader的初始化函数和取小批次操作。编码器和解码器模块主要包含自注意力机制、残差链接、层正则化、全连接、编码-解码注意力机制等函数。输出层主要设计输出层的激活函数。其他功能函数主要涉及创建索引张量IndexToOnehot、计算交叉熵损失函数CrossEntropy、张量元素求和ReduceSumAllValue、模型更新函数Update等等。

下面对编码器以及解码器的封装涉及到的主要函数做一下介绍，如表4.6所示。

表4.6 编码器与解码器封装函数介绍

| 函数 | 功能 | 参数 |
| --- | --- | --- |
| XTensor Sum(const XTensor &a, const XTensor &b, bool inplace, DTYPE beta) | 用作残差连接部分 | a：输入张量  b：输入张量  inplace：指示结果是否放在张量中  beta：比例因子 |
| XTensor Attention::Make(XTensor& k, XTensor& q, XTensor& v, XTensor\* mask, bool isTraining, Cache\* cache, int attType) | 注意力机制函数 | k、q、v：keys、queries、values  mask：mask张量  isTraining：模型是否用来训练  cache：解码器的cache  attType：注意力类型 |
| XTensor LayerNorm(XTensor& input, LN& ln, bool prenorm, bool before, bool after) | 层正则化函数 | input：输入张量  ln：网络的正则化层  prenorm：是否使用prenorm  before：是否在注意力之前使用  after：是否在注意力之后使用 |
| XTensor FNN::Make(XTensor& input, bool isTraining) | 全连接 | input：输入张量  isTraining：是否是训练模型 |

## 4.2 Transformer翻译模型搭建

本实验利用Python封装好的数据类型、功能函数、神经网络模型搭建Transformer翻译模型，在Python端搭建出编码器和解码器，通过组装小模块接口函数，让搭建后的实例更利于使用者理解结构并修改结构的操作。

# 5 系统实现

## 5.1 系统开发及训练环境

### 5.1.1 开发环境

本实验基于Window10操作系统，利用Microsoft Visual Studio 2017工具和Anaconda 4.8.3集成开发环境，接口封装工具为pybind11，Python语言的版本为3.7.6，使用的文本编辑工具为Notepad++。

表5.1 开发环境介绍

| 名称 | 版本 | 功能 |
| --- | --- | --- |
| Windows | 10 | 操作系统 |
| Visual Studio | 2017 | 集成开发环境 |
| Anaconda | 4.8.3 | 集成开发环境 |
| pybind11 | 2.6.0 | 接口封装工具 |
| Python | 3.7.6 | 编程语言 |
| Notepad++ | 7.9.5 | 文本编辑工具 |

### 5.1.2 训练环境

本实验主要于Window操作系统中编写开发程序，在测试成功后会配置Linux的CPU和GPU环境进行测试，保证本实验可适用多个环境。测试成功后会训练数据以此来验证程序的正确性，训练需要的环境为Linux的CPU和GPU、Window的CPU，由于训练过程数据庞大，对存储空间的要求较大，具体要求如表5.2和5.3所示。

表5.2 训练环境软件配置介绍

| 名称 | 版本 | 功能 |
| --- | --- | --- |
| Linux | CentOS 7.6 | 操作系统 |
| Cmake | 9.3.0 | 编译工具 |
| CUDA | 10.0 | 并行计算框架 |

表5.3 开发环境硬件配置介绍

| 硬件资源 | 配置要求 |
| --- | --- |
| CPU | 核心 |
| 显存 |  |
| 硬盘空间 |  |
| GPU | GTX Titan X |

## 5.2 Python接口封装

### 5.2.1 工程配置

在Window操作系统封装NiuTensor框架时需要引入pybind11封装工具以及Python语言API，并且需要在Release模式下生成动态链接库，这就需要配置Visual Studio的实验属性。

首先对输出的编译文件类型进行设置。设置目标文件名，该名称要与pybind11封装的入口模块名相同，这是pybind11语法所要求的。目标文件拓展名设为.pyd，配置类型设置为动态库（.dll），这意味着编译输出文件为Python动态链接库，Python语言可以import动态链接库。如图5.1所示。

之后需要修改VC++目录的包含目录和库目录，将pybind11工具包、Python语言API、numpy库的头文件写入包含目录中，并在库目录中将Anaconda的libs库引入。如图5.2所示。

最后需要在链接器的输入的附加依赖项中添加python3.lib和python37.lib，此处需要根据实际安装的Python版本填写。以上步骤就完成了Visual Studio对于NiuTensor的Python接口封装的工程配置。如图5.3所示。

图形用户界面, 文本, 应用程序

描述已自动生成

图5.1 配置类型设置

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图5.2 添加include和lib路径

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图5.3 链接器引入Python的lib库

### 5.2.2 封装代码具体实现

封装函数的作用是为Python端和C++端搭建数据交互的桥梁，使得双方可以相互分享使用对方的数据和调用对方的函数，并尽可能减少中间交互的代价。在本实验对NiuTensor框架的接口封装中，利用pybind11封装工具编写的接口函数提供了Python端使用C++端数据和函数的接口，同时也提供了C++端使用Python端数据的接口。整个接口函数与Python、C++的关系可如下图所示。接口函数与C++代码一起编译为.pyd（window）或者.so（Linux）动态链接库文件，Python文件可以import动态链接库文件，在Python解析器的作用下使用接口函数封装的数据结构和功能函数。

以XTensor数据结构的封装为例，接口函数处需要将XTensor该类及其想暴露给Python端进行直接调用的成员变量、功能函数命名封装。此时Python端可以利用封装的构造函数创建一个XTensor对象，并且直接显性使用XTensor被封装的成员变量和成员函数。但是单凭XTensor封装的构造函数进行创建太过局限性，为了能够使用Python对象的数据创建并赋值XTensor，本实验在C++端编写create\_from\_python函数，将Python端的List列表或者Numpy数组传入该函数，通过PySequence\_Check判断是否为连续序列，获取List/Numpy数据的维度dim以及每一位的大小shape，将List/Numpy数据转换为pybind11端的array\_t类型后，利用它的data函数获得数据的地址，将该地址以及数据维度、数据维度数组、数据类型、设备ID传入本实验编写修改的XTensor构造函数，创建直接使用Python端对象数据内存的张量，并将该张量的地址返回给Python端，不仅减少内存拷贝的浪费，也减少了中间返回张量的产生。封装函数的功能图如图5.4所示。

图示

描述已自动生成

图5.4 封装函数功能图

XTensor的封装测试成功后，其他数据结构的封装都是类似进行，C++端的函数参数可能存在Python端没有的类型比如char\*等，此时需要编写一个转换函数，获取Python端的str对象将其转换为char\*，再调用C++端原先的函数。

# 6 系统测试

## 6.1 系统测试环境

表6.1 系统测试环境

| 名称 | 版本 | 功能 |
| --- | --- | --- |
| Windows | 10 | 操作系统 |
| Visual Studio | 2017 | 集成开发环境 |
| pybind11 | 2.6.0 | 接口封装工具 |
| Python | 3.7.6 | 编程语言 |
| Linux | CentOS 7.6 | 操作系统 |
| Cmake | 9.3.0 | 编译工具 |
| CUDA | 10.0 | 并行计算框架 |

## 6.2 Transformer翻译模型测试

本实验将通过对Transformer翻译模型的测试，证明Python接口封装的正确性。将会从Window的CPU、Linux的CPU和GPU三个运行环境下对原神经网络模型和Python端自行搭建的神经网络模型进行测试，通过比较trainppl和testppl的数值，验证接口封装得是否正确。其中nepoch表示训练的轮数，hDepth表示隐藏层个数，hSize表示隐藏层的大小，traintime表示训练时间，loss表示损失值，ppl表示混淆度，表示语言模型在未见过的的字符串S上的表现，数值越小越好，bleu值表示翻译结果的评估，数值越大越好。

(1)Window-CPU环境的训练过程测试：

表6.2 Transformer翻译模型测试（Window -cpu）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| model | traintime | epoch | loss | trainppl | lr | testloss | testppl |
| Transformer-C++ | 185289.0s | 10 | 8.240 | 302.325 | 1.26e-03 | 7.004 | 128.355 |
| Transformer-python | 97156.9s | 10 | 8.211 | 296.334 | 1.27e-03 | 6.969 | 125.260 |

Transformer翻译模型在Window-CPU环境下的测试结果如上表所示，Python端和C++端的loss与ppl数据大致相同，由此可以得出Transformer翻译模型在Window-CPU环境下接口测试是正确合理的。

(2)Linux-CPU环境的训练过程测试：

表6.3 Transformer翻译模型测试（Linux-cpu）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| model | traintime | epoch | loss | trainppl | lr | testloss | testppl |
| Transformer-C++ | 127014.8s | 10 | 8.206 | 295.388 | 1.26e-03 | 6.966 | 125.050 |
| Transformer-python | 68914.1s | 10 | 8.235 | 301.188 | 1.26e-03 | 7.008 | 128.679 |

Transformer翻译模型在Linux-CPU环境下的测试结果如上e表所示，Python端和C++端的loss与ppl数据大致相同，由此可以得出Transformer翻译模型在Linux-CPU环境下接口测试也是正确合理的。

(3)Linux-GPU环境的训练过程测试：

表6.4 Transformer翻译模型测试（Linux-gpu）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| model | traintime | epoch | loss | trainppl | lr | testloss | testppl |
| Transformer-C++ | 5005.2s | 50 | 4.757 | 27.047 | 5.64e-04 | 3.204 | 9.217 |
| Transformer-python | 5036.6s | 50 | 4.745 | 26.816 | 5.64e-04 | 3.180 | 9.066 |

Transformer翻译模型在Linux-GPU环境下的测试结果如上表所示，Python端和C++端的loss与ppl数据大致相同，训练运行时间也几近一致。由此可以得出Transformer翻译模型在Linux-GPU环境下接口测试也是正确合理的。利用Linux-GPU环境下训练的50轮模型进行翻译测试，以测试翻译模块封装的正确性，测试结果为：Python端测试的bleu值为28.45，C++端测试的bleu值为28.68，bleu值相差不大，翻译效果几近一致。

# 7 心得体会

通过本次实验，我的学习能力得到了很大的提高。我对于NiuTensor框架的认识更为深刻，学会了如何对C++的数据类型、功能函数进行Python接口的封装。对于Transformer的学习更加深入，根据Transformer的模型结构设计了NiuTensor的Transformer模型接口函数的划分，利用封装的神经网络模型部件在Python端进行组装。

Python接口封装方面，我学会了使用pybind11工具封装C++代码，并在Linux的CPU、GPU以及Window的CPU三个环境下成功测试了接口的正确性。对于Transformer模型函数的拆分封装过程中，我对于函数拆分的粒度掌握不好，于是参照了Pytorch中搭建的Transformer模型，根据模型的结构和功能进行划分。这也让我意识到学习前辈成果的重要性，也意识到在封装函数的时候就要保证接口函数的普适性。

Trangformer模型搭建方面，我将划分好的函数进行封装，将封装函数在Python端搭建Transformer模型。在模型搭建的过程中，我对于Transformer模型结构的理解更加深入。

本实验仍然需要很多需要提升和改进的地方，主要有以下几个内容：

（1）由于测试时间过长的原因，本实验对于两个神经网络模型接口的测试轮次过少，测试结果可能存在偶然性的影响。接下来本实验会继续进行接口测试，增加训练的轮次数，减少偶然性等因素的影响。

（2）本实验完成了NiuTensor各个模块的函数的封装，虽然保证了接口封装的正确性，但是没有进行系统的测试，这也是未来要测试的一个任务。从多个角度、不同条件、不同环境全面测试接口封装的准确性。

（3）本实验利用Python端接口搭建神经网络模型进行运行时，存在运行时间过长的问题，主要原因在于Python端和C++端数据传递的时间耗费，优化Python端神经网络模型运行时间也是未来要解决的一个问题。

（4）本实验的接口设计是借鉴了Pytorch等深度学习框架的接口设计，但是由于需要适配NiuTensor的底层代码，接口函数的设计不够实用，约束条件过多，相较Pytorch的接口设计还有很大的距离，未来本实验会继续优化接口，提高接口的实用性。