NiuTensor框架的python组件实现与应用

姓 名：刘晓倩

学 号：2001775

班 级：计硕2005班

专 业：计算机科学与技术

**1 系统简介**

本文使用了pybind11工具，对NiuTensor框架提供的张量定义及计算功能进行Python组件的接口封装，将NiuTensor框架的调用方式由单纯的C++语言扩展至Python语言，还实现了基于Python语言的语言模型搭建。

2 系统实现

2.1系统开发及训练环境

2.1.1 开发环境

本系统基于Windows 10系统开发，使用 Microsoft Visual Studio 2017 及 PyCharm 2018.1集成开发环境， Python接口封装工具为pybind11，使用的Python编程语言版本为3.6.5，使用的其他工具有文本编辑工具 NotePad++。表2.1详细列出了系统开发的软件需求。

表2.1 系统开发软件需求表

|  |  |  |
| --- | --- | --- |
| **名称** | **版本** | **功能** |
| Windows | 10 | 操作系统 |
| PyCharm | 2018.1 | 集成开发环境 |
| Visual Studio | 2017 | 集成开发环境 |
| Notepad++ | 7.8.4 | 文本编辑工具 |
| Python | 3.6.5 | 编程语言 |
| pybind11 | 2.5.0 | 开发工具 |

**2.1.2 训练环境**

本系统的训练环境为Linux 64位系统，训练过程在Linux服务器上完成。由于在训练过程中系统会占用大量内存和硬盘空间，笔记本电脑的存储空间较小，训练需求无法得到满足，并且随着训练步数和数据量的增长，系统对显存以及硬盘的需求也随之增加。表2.2和2.3列出了系统训练的软硬件需求。

表2.2 系统训练软件需求表

|  |  |  |
| --- | --- | --- |
| **名称** | **版本** | **功能** |
| Linux | CentOS 7.6 | 操作系统 |
| Cmake | 2.8.12.2 | 编译工具 |
| CUDA | 9.2 | 并行计算框架 |

表2.3 系统训练硬件需求表

|  |  |
| --- | --- |
| **硬件资源** | **配置要求** |
| CPU | ≥4核心 |
| 显存 | ≥10GB |
| 硬盘空间 | ≥10GB |
| GPU | GTX Titan X |

**2.2 语言模型的搭建及优化**

**2.2.1 前馈神经网络语言模型搭建及模块优化**

基于NiuTensor深度学习框架实现的FNNLM主要使用了FNNModel类，其他功能通过框架中的接口调用。接下来本文介绍神经网络语言模型FNNModel类的具体实现，图 2.1是整体的类图，不仅给出了FNNModel的实现，也展示了在系统实现时这个类与框架中其他功能类的关联。

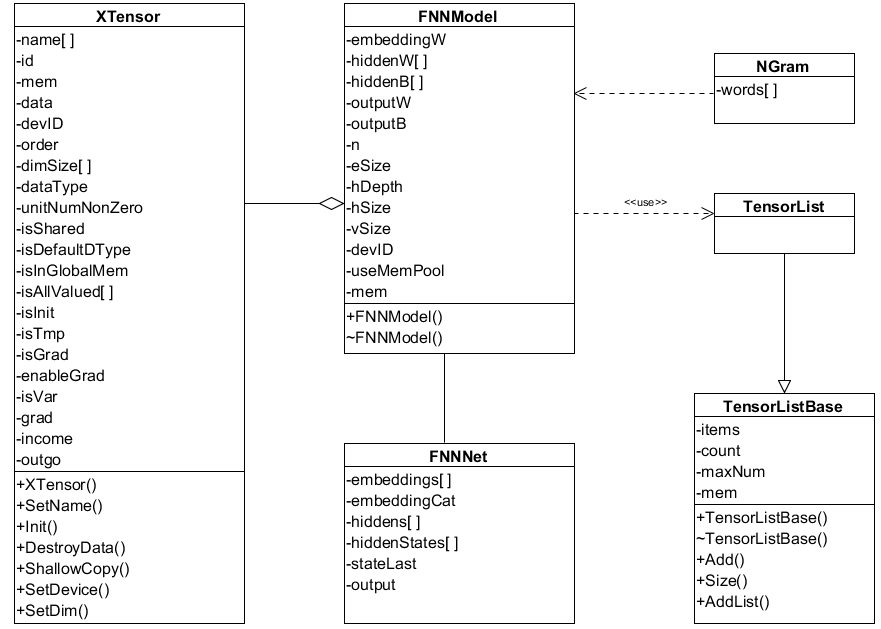


图2.1 FNNModel类图

下面介绍前馈神经网络语言模型的实现流程。首先，对神经网络语言模型中的参数与基本层结构进行初始化。其中，SetVarFlag()函数将张量设定为变量（Variable），在计算过程中会通过反向传播得到这些张量的梯度，然后使用梯度下降法进行参数更新。之后是神经网络的训练过程。在这个语言模型中，单词已经不再是一个孤立的符号串，而是被表示为一个实数向量。这样，可以导致相似的单词会具有相似的分布，进而缓解 n元语言模型的问题——明明意思很相近的两个词但是概率估计的结果差异性却很大。因此，在网络训练时，首先是输入数据的预处理，将原始数据转换成训练时使用的实数向量表示形式。使用LoadNGrams()和MakeWordBatch()两个函数将输入数据处理成词的分布式表示形式。数据处理之后，通过ForwardAutoDiff()函数实现网络前向计算。计算完成之后，使用CrossEntropy()交叉熵函数计算预测结果和标准答案之间的损失。损失计算完毕后，使用Backward()函数自动计算反向过程求得参数的梯度，然后采用梯度下降的方法对参数进行更新。通过不断迭代进行上述的前向计算、反向计算和参数更新过程，模型逐渐达到收敛状态。

上述给出了神经网络语言模型需要实现的功能，本文实现的神经网络语言模型的主要方法如表 2.4所示。

表2.4 神经网络语言模型主要方法表

|  |  |
| --- | --- |
| **函数名称** | **函数功能** |
| LoadArgs() | 加载用户输入的超参数 |
| LoadNGrams() | 输入数据预处理 |
| Init() | 初始化语言模型及参数 |
| Check() | 检查用户输入的超参数是否符合规范 |
| Clear () | 模型每轮训练完毕后需要清理梯度 |
| Train() | 训练模型 |
| Update() | 更新模型的参数 |
| Dump() | 将最终训练好的模型保存到本地 |
| Test() | 使用测试集对训练好的模型进行测试 |
| ForwardAutoDiff() | 自动求导时的前向传播函数 |

最后详细介绍如何进行神经网络预测，也就是概率运算。首先，模型在字的前后添加若干<s>，在词汇表中查到其所在的位置，并将其转换成作为神经网络输入的词向量表示形式。然后进行网络的前向传播过程，最终在输出层得到输出结果，即为词汇表中所有的字在这个字后面的概率。最后计算概率的log值，写入指定的文件中。概率计算的流程图如图2.2所示。

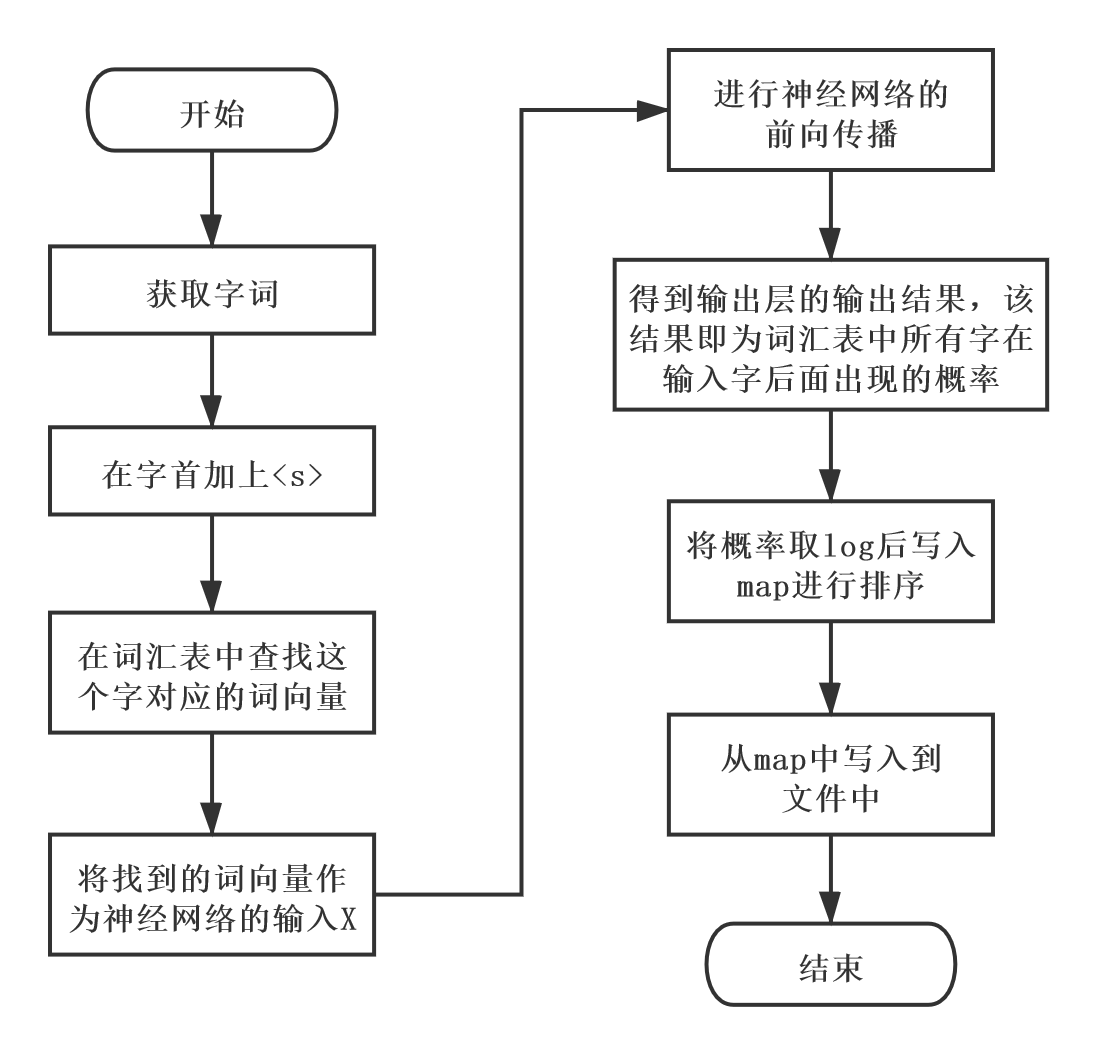


图2.2 神经网络语言模型预测字概率流程

表2.5给出了本文在选择Penn Treebank数据集时，系统训练的超参数设置。

表5.5 Penn TreeBank数据集系统的超参数设置

|  |  |
| --- | --- |
| 超参数名称 | 超参数取值 |
| lrate | 0.01 |
| nepoch | 30 |
| n | 3 |
| esize | 128 |
| vsize | 10001 |
| hdepth | 3 |
| hsize | 16 |
| minmax | 1 |
| shuffle | TRUE |
| autodiff | TRUE |
| batch | 256 |
| wbatch | 256 |

**2.2.2 神经网络语言模型性能优化**

由于前馈神经网络语言模型依赖有限上下文假设的弊端，在对前馈神经网络语言模型进行优化时，考虑引入更大范围的历史信息来捕捉单词间的长距离依赖。本节搭建一个循环神经网络语言模型，通过引入循环单元这种特殊的结构，对任意长度的历史进行建模，在一定程度上对前馈神经网络语言模型性能进行优化。

图2.3是RNNModel类的实现，该类在系统实现时与FNNModel类似，且这个类与框架中其他功能类的关联也是与FNNModel类似的，只是引入循环单元这种特殊的结构，从模型内部进行了优化。

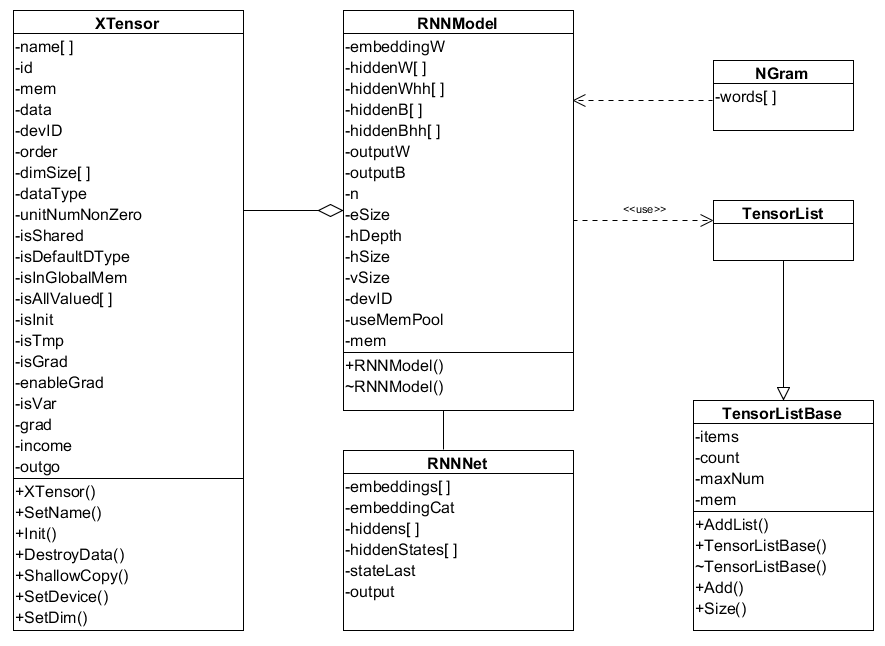


图2.3 RNNModel类图

循环神经网络语言模型的实现流程与前馈模型的实现流程基本一致。首先，将所有输入的单词映射成词的连续空间表示。之后，该模型堆叠了两层循环神经网络每个循环单元，读入前一个时刻循环单元的输出和当前时刻的输入，生成当前时刻循环单元的输出。最后通过 Softmax 层得到每个时刻的预测结果。

2.3 Python接口封装

2.3.1 工程配置

在实现NiuTensor的Python接口封装时，使用了pybind11、numpy/C API与Python/C API，并且这些API都是以头文件的形式提供的。因此封装接口的第一步为修改Visual Studio工程的相关设置，使Visual Studio编译器可以找到这些头文件并完成链接过程。

本节修改的设置为Visual Studio项目的include目录与lib目录，需要在这两部分分别添加pybind11、numpy/C API与Python/C API对应的/include和/libs路径。在解决方案：属性->VC++目录->包含目录/库目录中可以找到这些设置并进行修改。

首先进行编译输出类型的设置。如图2.4所示，需要配置生成动态链接库.dll文件，设置目标文件名为NiuTensor，文件扩展名为.pyd，即编译生成的输出文件是NiuTensor.pyd。

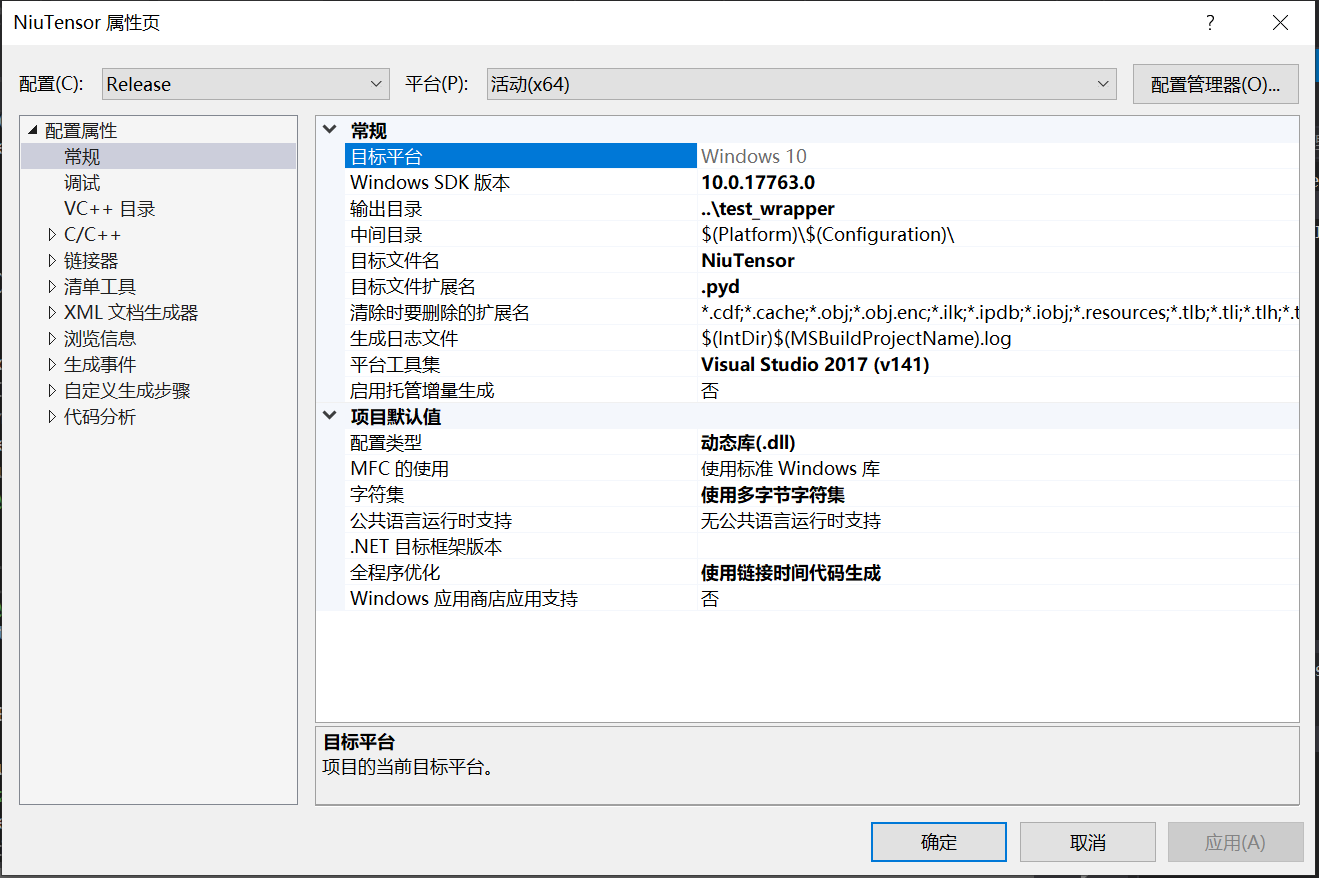


图2.4 编译输出类型设置

之后，如图2.5所示，添加系统实现中使用的pybind11、numpy/C API和Python/ C API几个头文件对应的/include与/libs目录。最后，如图2.6所示，向链接器添加lib。这样就完成了使用pybind11封装NiuTensor的Python接口的工程配置。

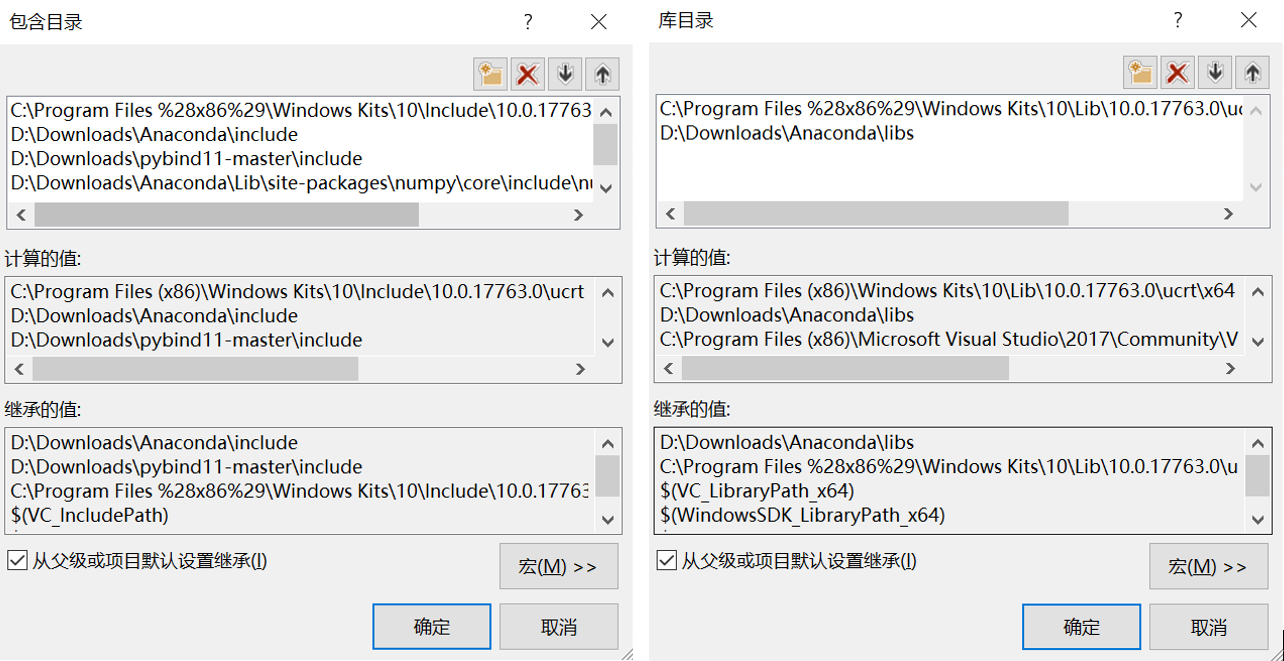


图2.5 添加include路径与lib路径

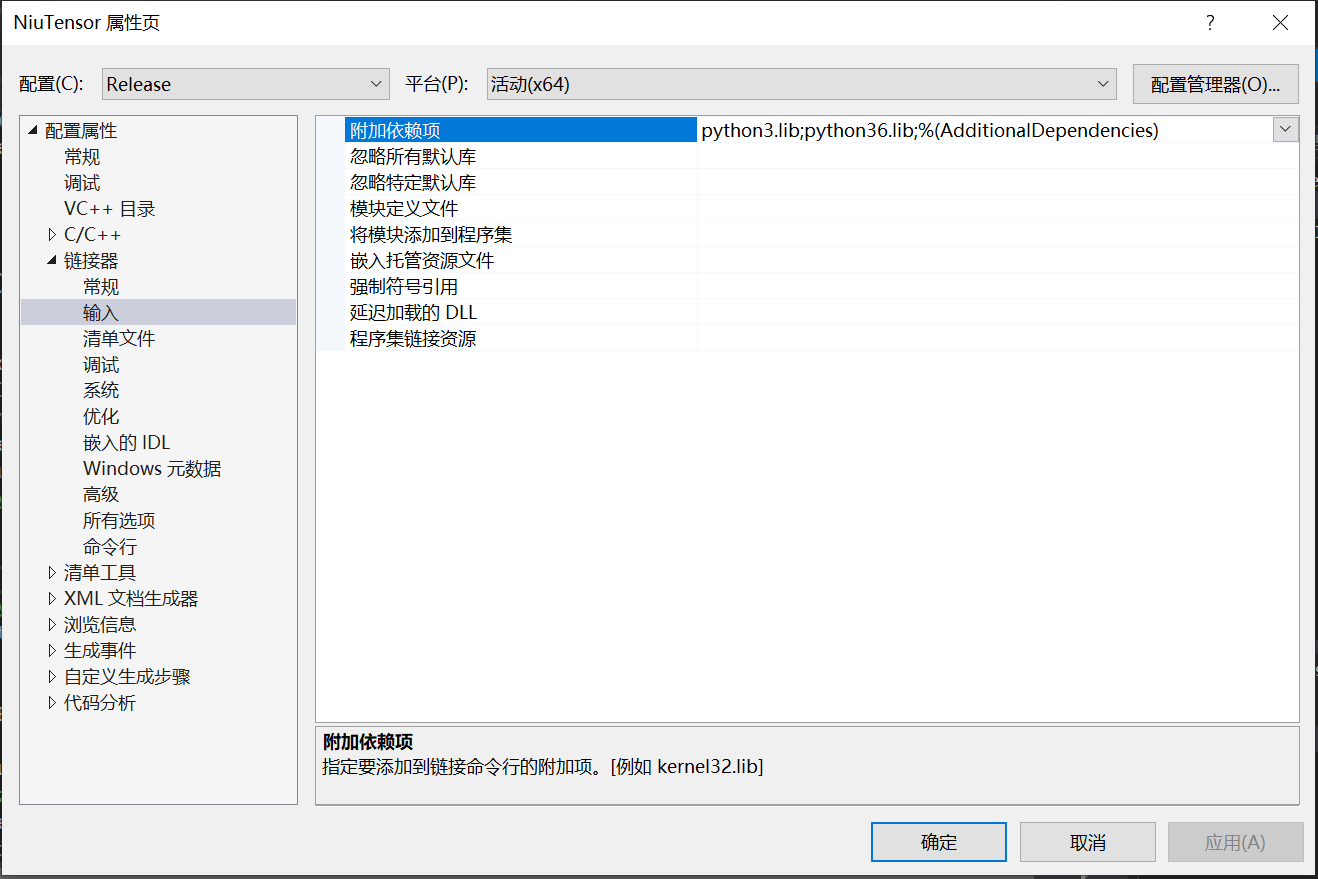


图2.6 链接器添加lib

2.3.2 封装代码具体实现

在封装函数Wrapper的设计中，预期封装好NiuTensor的用法是与PyTorch、TensorFlow等框架用法类似的。例如，想要使用Python的list来构造一个张量，在PyTorch中使用了torch.Tensor()的方法，预期在NiuTensor中可以使用类似的用法。在调用Tensor方法时，产生了数据的传递，这个list类型的data需要被传递到C++层来构造一个张量，返回的数据应该是NiuTensor类型的张量。数据的传递就是Wrapper函数完成的。作为一个媒介，它先获取Python层data的全部信息，再向C++层申请一个张量，将获得的数据信息传递给C++，最后将tensor返回给Python层。

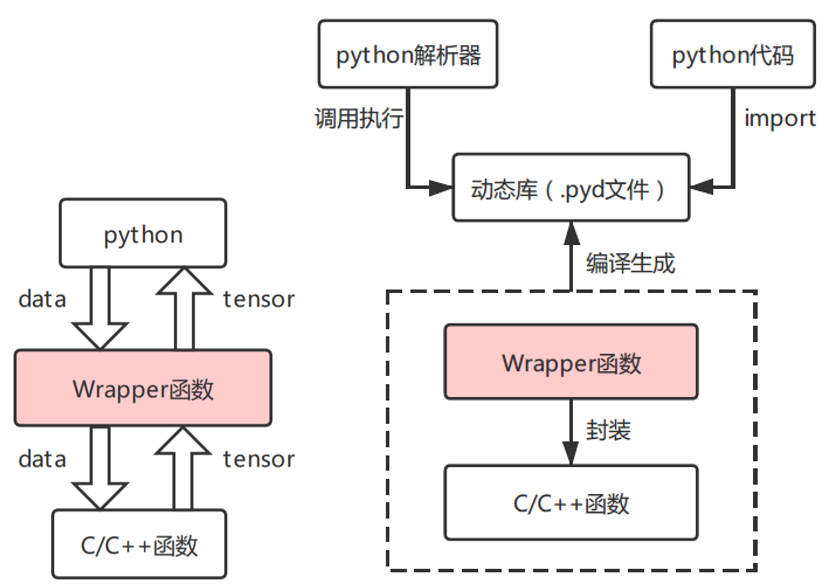


图2.7 封装函数Wrapper的功能

Tensor方法即为使用Python.list来构造张量的方法，下面详细介绍一下Tensor方法的实现。首先是尝试接收Python的object，然后判断所接受的对象是什么数据类型。使用numpy的接口函数PyArray\_Check(obj)函数判断Python层传递的数据类型，然后调用tensor\_from\_numpy函数来处理data。在tensor\_from\_numpy这个函数里使用几个numpy的接口函数计算数组的维度信息、提取数组数据存储的地址。然后，调用NiuTensor框架的NewTensor函数和SetData函数来申请一个新的tensor并用data的信息去定义它，将data的值传入tensor。最后就是封装代码，全部写在 PYBIND11\_MODULE 这个函数中，作用是暴露Tensor方法和Tensor类给Python层。这样就完成了Wrapper函数的编写，只要将这个文件和原来的C++文件一起编译，就可以生成Python调用的模块——NiuTensor.pyd了。

2.3.3 基于Python的语言模型搭建

使用封装好的NiuTensor框架，在Python中使用import NiuTensor语句就可以导入NiuTensor模块，通过调用各个接口函数，就可以实现语言模型的搭建。首先，要对训练数据进行加载和处理，将原始的训练语料转换成对应的词向量表示形式。之后，将输入的超参数加载到模型中，根据超参数的设置对模型进行检查，确认超参数的设置合法后对模型进行初始化。当当前的训练轮数小于设置的迭代次数时，对模型进行训练。其中，模型的梯度清零、神经网络的前向传播、损失计算已经参数更新这些操作都是在训练模型函数中实现的。当模型的训练结束时，使用最终的模型对语料库中的测试集进行测试，并将最终模型保存。基于NiuTensor的语言模型搭建（Python语言实现）的流程图如2.8所示。

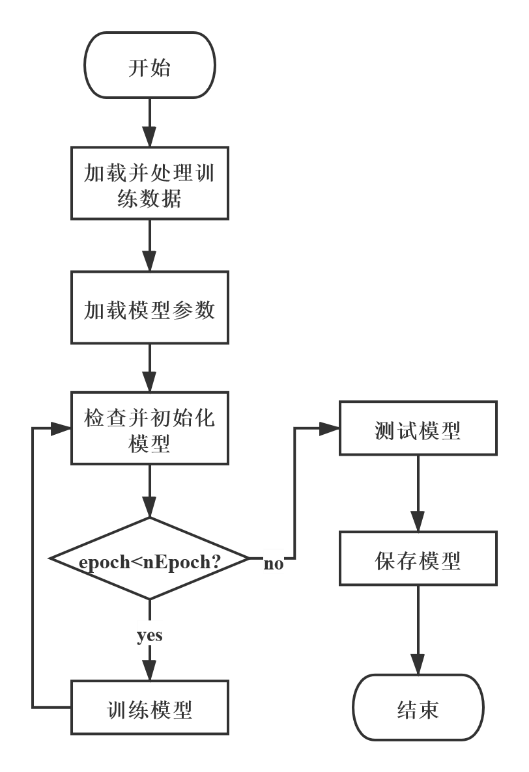


图2.8 Python语言搭建神经网络语言模型流程图

**3 系统测试**

**3.1 FNNLM性能测试**

表3.1给出了前馈神经网络语言模型在PTB数据集上的表现，在这个数据集上进行测试的目的是验证模型搭建的正确性。

表 3.1 fnnlm在ptb数据集训练的测试结果

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **序号** | **nepoch** | **n** | **hDepth** | **hSize** | **CPU/GPU** | **traintime** | **trainppl** | **testppl** |
| 1 | 50 | 3 | 3 | 32 | GPU | 1042.3s | 113.712 | 128.460 |
| 2 | **30** | 3 | 3 | 32 | GPU | **615.7s** | 242.313 | 261.102 |
| 3 | 50 | **4** | 3 | 32 | GPU | 1228.1s | 139.216 | 136.271 |
| 4 | 50 | 3 | **4** | 32 | GPU | 1217.3s | 125.712 | 128.300 |
| 5 | 50 | 3 | 3 | **16** | GPU | 1126.2s | 241.337 | 261.810 |
| 6 | 50 | 3 | 3 | 32 | **CPU** | **51955.0s** | 112.834 | 132.562 |

模型1是设置的基线模型，是本文通过多次实验测试获得的在PTB数据集上最好性能的模型。模型2~模型6都是在其基础上，使用单一变量的控制方法调整参数训练得到的。对比模型1与模型2，可以看到迭代轮数减小会降低模型的训练时间，但会产生欠拟合现象，模型无法获得较好的性能；对比模型1与模型3、4，可以发现3元语言模型和4元语言模型以及网络深度在这个数据集上对模型性能上的影响不太大；对比模型1与模型5，可以发现隐藏层神经元个数较少会降低模型的性能；对比模型1与模型6，GPU对模型训练的加速是毋庸置疑的，GPU训练将模型的训练速度提高了近50倍，并且模型性能也是相似的。

目前，PTB数据集上主流的语言模型ppl值在40~80之间，而本文搭建的基础模型最佳性能只在110左右。这也证明了一个简单的前馈神经网络语言模型无法获得令人满意的性能，对模型的优化工作是非常必要的。

3.2 RNNLM性能测试

表3.2给出了前馈神经网络语言模型的基线模型和循环神经网络语言模型在PTB数据集上的表现，这个测试是为了验证循环神经网络语言模型对前馈神经网络语言模型性能改善而设置。

表 3.2 rnnlm在ptb数据集训练的测试结果

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **序号** | **nepoch** | **n** | **hDepth** | **hSize** | **CPU/GPU** | **traintime** | **trainppl** | **testppl** |
| 1 | 50 | 3 | 3 | 32 | GPU | 1042.3s | 113.712 | 128.460 |
| 2 | 30 | 3 | 3 | 32 | GPU | 731.5s | 64.557 | 68.520 |
| 3 | 50 | 3 | 3 | 32 | GPU | 1226.0s | 52.355 | 53.060 |

模型1是前馈神经网络语言模型的基线模型，模型2和模型2都是在在保证其余超参数设置相同的情况下训练的循环神经网络语言模型。对比模型1与模型2，可以看到即使是迭代轮数较小，模型还未训练到最佳性能，循环神经网络语言模型已经可以取得非常好的性能了，模型3更是验证了这一结论。通过实验，可以得到循环神经网络语言模型能对前馈神经网络语言模型进行优化的结论，并且该模型能取得令人满意的效果。

3.3 WikiText-2数据集测试结果

图3.1与表3.3都是循环神经网络语言模型在WikiText-2数据集上的训练结果。

图3.1 RNNLM在WikiText-2数据集上的训练结果

通过图3.1可以清晰地看出随着参数取值变化，ppl值的变化情况。除了循环单元数hDepth、训练轮数nEpoch、隐藏层神经元个数hSize外，其余超参数均保持一致。通过测试，可以发现随着隐藏层神经元个数的增加，模型的性能总体呈现逐渐变好的趋势。而循环单元个数的增加并不会提高模型的准确性，证明模型已经很好拟合。

表 3.3 rnnlm在WikiText-2数据集训练的测试结果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **序号** | **hDepth** | **nepoch** | **hSize** | **traintime** | **trainppl** | **testppl** |
| 1 | 3 | 30 | 16 | 3710.7s | 198.601 | 160.951 |
| 2 | 3 | 50 | 16 | 6175.2s | 201.932 | 149.482 |
| 3 | 4 | 30 | 16 | 4028.3s | 214.02 | 290.390 |
| 4 | 3 | 30 | 24 | 3739.1s | 82.258 | 74.350 |
| 5 | 3 | 50 | 24 | 6267.1s | 65.445 | 59.421 |
| 6 | 4 | 30 | 24 | 4093.0s | 172.141 | 140.680 |
| 7 | 3 | 30 | 32 | 3775.8s | 62.289 | 53.000 |
| 8 | 3 | 50 | 32 | 6364.6s | 41.651 | 40.142 |
| 9 | 4 | 30 | 32 | 5507.6s | 59.351 | 57.170 |

将搭建的前馈神经网络语言模型和循环神经网络语言模型性能进行对比测试，验证了循环模型的性能更加优良。对于循环神经网络语言模型，不断调整超参数的设置，验证模型的最佳性能是合理的。

**4 总结及展望**

**4.1 总结**

（1）首先搭建了语言模型任务中经典的前馈神经网络语言模型，在对其进行模块优化后搭建了循环神经网络语言模型以实现性能优化任务，在训练时对两个模型的训练效果进行了对比，最终选择了性能更加优秀的循环神经网络语言模型进行系统的搭建。

（2）尝试了SWIG、Cython和pybind11三个接口封装工具，在对比后选择了最适合系统开发的pybind11进行NiuTensor框架的Python组件的接口封装，并成功将其运用在使用Python语言进行语言模型搭建的任务中。

（3）使用了两个语言模型的经典数据集：Penn Treebank数据集和WikiText-2数据集对语言模型进行训练和测试，将训练结果与当前发表过的论文中采用的语言模型得到的预测性能进行对比，实验结果验证了模型搭建的正确性。

**4.2 展望**

本文实现的工作仍有不足之处，有待改进和完善的内容为：

（1）本文搭建的模型是语言模型中较为基础的两个模型，对于循环网络语言模型，在简单的循环单元结构的基础上，也有很多改进工作，如LSTM、GRU等。未来将对这些变种模型进行深入研究。

（2）在对NiuTensor进行封装时，只考虑了搭建前馈神经网络语言模型涉及到的代码功能模块，而并未对其他功能进行封装。未来将继续扩充NiuTensor框架的封装，将函数涵盖更加全面，以达到可以在Python中自由使用NiuTensor框架搭建多种模型。

最后，本文是针对NiuTensor深度学习框架的一次成功的运用，基于该框架搭建的神经网络语言模型可以取得良好的性能。同时，对NiuTensor框架的Python接口封装是一种从0到1的突破，这种接口封装的开发工作打破了基于C++语言调用NiuTensor框架的局限性，为NiuTensor的推广使用带来了新的方向。

**5 致谢**

很高兴可以顺利完成本次机器翻译课程的实验，我深知这一切都离不开肖桐老师的谆谆教诲，肖老师亦师亦友，严谨的学术态度幽默的授课风格有机地融为了一体，为我们营造了生动活泼的良好学习氛围。

特别感谢实验室的张裕浩学长、胡驰学长和单伟乔学长，他们不遗余力地给我提供了很多帮助，解决了我在实验室遇到的困难。经历了本次的实验，在今后的科研道路上，我也会一直秉持一份敬仰之情，不断求索。