University of Science and Technology of China

基于机器学习的小文件预取

小组成员：董恒、李楠、连家诚

2018 年 7 月 3 日

目录

1 小组成员介绍

董恒 C、C++、Python编程经验，未做过大型项目。对神经网络有一些了解。

李楠 C、C++编程经验

连家诚 C、C++编程经验，学过单片机开发。

小结 总的来说，我们4个人均不是大佬级别的人物，没有太多的背景支持，所以关于这个课题的东西基本上都需要从零开始。

2 项目背景

本节介绍这个课题的相关背景，包括分布式文件系统概述，HDFS的介绍，其他分布式文件系统介绍。

2.1 分布式文件系统

简介什么是分布式文件系统以及它的特点和历史。

2.1.1 简介

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。分布式文件系统的设计基于客机/服务器模式。一个典型的网络可能包括多个供多用户访问的服务器。另外，对等特性允许一些系统扮演客户机和服务器的双重角色。

2.1.2 设计目标

分布式文件系统在很多方面都希望实现“transparency”这是说，他们希望这些对客户机程序是不可见的，让客户机可以把DFS看成本地系统。在这种情况下，分布式文件系统处理定位文件、传递数据，以及可能提供其他的特性如下：

Access Transparency 客户机并不会意识到文件是分布式的，它们能够像存取本地文件一样存取分布式文件。

Location Transparency 有一个一致的名字空间存在，包含了本地文件和远程文件。一个文件的名字不会提供具体的存储位置。

Concurrency Transparency 所有的客户机看到同样的文件系统状态。这就是说，如果

一个进程修改了一个文件，任何其他的进程，不论它在相同的文件系统上还是远程系统上，都会看到看到修改。
Failure Transparency 如果一个服务器故障，客户机和客户机程序应该正确运行。

Heterogeneity 文件服务应该可以提供给不同的硬件和操作系统。

Scalability 文件系统应该能够在小规模环境（一个或几个机器）和大规模环境（成百上千的机器）中正常运行。

Replication Transparency 为了提供Scalability，文件系统需要在不同的服务器里复制文件，并且客户机不会意识到这一点。

Migration Transparency 文件应该可以自由移动。

2.2 HDFS

以下内容主要来自[1]作为专门讲解HDFS结构的文献，这篇文章讲得比较详细，整理如下。

2.2.1 简介

HDFS被设计为可以运行在商用硬件上，与现存的分布式文件系统有很多的相似之处。但是区别也很明显，比如高容错性，高吞吐率。这一点也让HDFS非常适合大数据存取。HDFS基本实现了以下的几个特点：

硬件故障 HDFS由很多机器构成，这种情况下，硬件故障将成为一种常态，所以系统必须能够检测到故障并且能够快速自动恢复。

流数据存取 在HDFS上运行的应用需要流存取数据集，并且系统更多地被设计成为了批处理而不是交互式使用，所以更多地需要数据的大吞吐率。

大数据集 在HDFS上运行的应用需要大数据集，每一个文件大概是GB到TB大小，因此系统需要提供大的数据带宽。

简单连接模型 基本模型是：write-once-read-many存取模型。一个文件一旦被创建、写入然后关闭之后就不必被改变。这样的假设简单化了模型，并且使高吞吐率成为可能。

移动计算而不是移动数据 让计算更加接近数据会更加高效，特别是当数据量很大时。这种方式减少了网络拥挤并且提高了系统总的吞吐率。一种假设是，移动计算到接近数据的位置比移动数据到应用运行的位置更加合适。因此，HDFS提供了一种让应用移动它们自己到数据存储位置的接口。

可移植性 HDFS设计成很容易从一个平台移动到另一个平台。

2.2.2 NameNode 和 DataNode

HDFS有一个master/slave结构，一个HDFS集群由一个NameNode和几个DataNode组成，其中NameNode是一个master服务器，管理文件系统命名空间以及管理客户机的存取，DataNode管理储存。在内部，一个文件被分割成一个或者几个块，这些块被存储在一些DataNode中。NameNode处理文件系统命名空间的操作，比如说，打开、关闭、重命名文件或者目录，并且NameNode也决定了块到DataNodes的映射关系。DataNode则处理客
户机的读写请求，以及块的创建、删除和复制。下面的图1形象表示了这种结构。

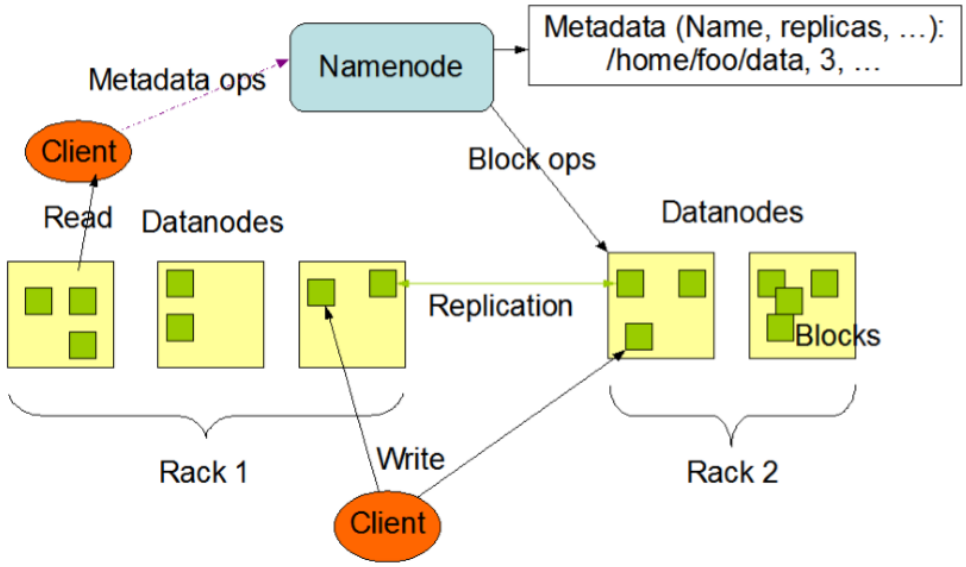


图 1：HDFS Architecture

NameNode和DataNode都是一些能在商用机器上运行的软件。可用Linux由于HDFS使用JAVA实现，因此任何支持JAVA的机器可以运行NameNode和DataNode。

2.2.3 文件系统命名空间

HDFS支持一个传统的文件组织体系，一个用户或者一个应用额可以创建目录然后储存文件。文件系统命名空间和现有的文件系统非常相似，可以创建、移除文件，移动文件到另一个目录，或者重命名一个文件。但是HDFS并未实现用户限额和存取许可，以及硬链接、软链接也不支持。但是HDFS的结构并不排除实现这个功能。NameNode维持文件系统的命名空间，任何对文件系统命名空间的修改都会被NameNode记录。

2.2.4 数据复制

HDFS被设计成能够可信地在一个大集群内存储大容量数据，它把一个文件存储在

连续的几个块中，除了最后一个块外，其他的块都由同样的大小。为了实现容错性，块都会被复制存储。每个块的大小和复制数都可以被设置，在创建的时刻或者在之后。HDFS文件每次只能有一个写入。NameNode处理所有的关于块复制的决定，它会周期地从每DataNode收到一个心跳和一个块报告，心跳代表NameNode正常运行，块报告包含了在一DataNode里面所有的块的列表。

2.2.5 文件系统元数据的存储

HDFS命名空间被存储在NameNode，NameNode使用信息日志（EditLog）来记录每次元数据的改变。比如说，创建一个新的文件会导致NameNode插入一个记录到EditLog，同样如果改变复制数也会插入记录。整个文件系统命名空间，包括块到文件的映射和文件系统的属性，都被存储在本地hostOS的一个文件（Fsimage）上。

NameNode保存了整个文件系统命名空间的图像和文件块映射在内存里。当NameNode启动的时候，它会从硬盘里读取Fsimage和EditLog。DataNode存储HDFS数据在它自己的文件系统里，对于文件信息，DataNode一无所知，它把每个块存储在一个它本地文件系统的单独的一个文件里。在同一个文件里，DataNode不会把所有的文件都放到同一个目录里，它会使用一个探索式的方法来决定最佳的方式，比如每个目录多少文件，以及是否产生子目录。这样做是因为，本地文件系统如果只用一个目录会不高效。当DataNode启动的时候，它会扫描整个本地系统，产生所有块的列表，并且把这些报告给NameNode，这就是上文提到的块报告（Blockreport）

2.2.6 数据组织方式

数据块 HDFS本身设计是为了支持大型文件，适合HDFS的是那些处理大数据集的应用，这些应用只写入一次，读出一次或者多次，并且需要读出有比较高的速度。通常来说，一个块的大小是64M。

Staging 客户端请求创建一个文件并不会直接马上到达NameNode，事实上，最开始时，HDFS客户机会把文件数据缓存到一个暂时的本地目录。应用的写入会被显式地重定向到这个暂时文件，当本地文件的数据积累到超过HDFS块的大小，客户机才会联络NameNode。NameNode会把文件名字插入到文件系统，然后分配一个数据块。NameNode会回复客户端这个DataNode的ID和块地址。然后客户端会把一个块的数据传输到特定的DataNode。当一个文件被关闭的时候，剩下的未传输的数据会被传输到DataNode，然后客户端会高数NameNode文件已经关闭，在这个时候，NameNode会把文件创建操作提交到固定储存。如果NameNode在文件关闭之前垮掉，文件就会丢失。以上方法方法是经过仔细考虑的，如果一个客户端直接向远程端写入，而不经过本地的缓存，网络速度和吞吐率就会被很大程度影响。这个方法也不是首次的，之前的一个分布式文件系统AFS也用了客户端缓存去提高性能。

复制管道 当客户端写入数据到HDFS文件的时候，会首先写入到本地文件，就像上面写到的，现在假设复制数为3，当本地文件积累到满块的时候，客户端会从NameNode里面检索DataNode列表，这个列表包含了一些以后会存储复制的DataNode.第一个DataNode会开始一小部分（4KB）来接收数据，把每个小部分都写入到它自己的本地

目录，然后把这个小部分传送到在这个列表的第二个DataNode，第二个DataNode同样会把这部分传送到第三个DataNode。这样就像一个管道一样在同一时间把数据传送到3个节点。

3 立项依据

以上介绍了基本的DFS以及常见的分布式文件系统，也详细解析了HDFS的组织结构，但是并未涉及到具体的使用和改造过程，作为背景部分，也没有涉及为什么选择DFS，为什么选择HDFS，以及用这个做什么。下面将具体解释。

3.1 分布式文件系统海量小文件存取问题

正如[2]所说，大量小文件在分布式文件系统的存取是普遍的问题。具体而言，由于社交网络（Facebook、微博）和网购（Amazon、淘宝）的流行，有很多种类的海量小文件需要存取，比如说用户的信息、图片等等，这些数据都有它们特有的特征，它们的大小一般是几KB到几MB。比如说，淘宝使用的Taobao File System（TFS），大概管理了28.6亿万的图片，平均大小是17.45KB，这和大文件很不相同。

从上文我们可以看到，DFS是一种基于客户机/服务机架构的存取文件方式，在一个DFS中，所有的文件都会被复制然后存储在一些数据服务机上，并且元数据会被存储在元数据服务器上。一个客户机必须先在元数据服务器上查找一个文件，而不是传统的本地式的通过路径来查找。因此，一个客户机取文件包括两个阶段，首先，询问元数据服务器得到存储相应文件的数据服务器的IP地址，然后，联络数据服务器去获取数据文件。

在一般的DFS中，比如上文提到的GFS、HDFS，这样的分块存储是被设计成处理大文件的（比如HDFS是64M），每一个块都被当作一个普通的文件存储在不同的数据服务器上。对每个文件，至少需要一个元数据对象存储在元数据服务器以及至少一个普通文件存储在数据服务器。当存储在系统中的文件主要是小文件的时候，存储文件的数目会显著增加，这会导致增加很大数目的元数据存储在元数据服务器，并且也会导致低效率的文件获取。因而关于在DFS存储小文件，是一个很热门的话题。

3.2 HDFS 小文件存取问题

在[3]中提到，Hadoop的用户量非常大，有一些很大的客户，比如说，Yahoo、Facebook、Netflix、Amazon会使用Hadoop来做非结构化数据的分析，而HDFS是用来存储大文件的，但是大量的小文件也需要被存储，正如上文提到了，在这方面其他的DFS遇到了不小的问题，同样的，HDFS也存在很大的问题需要解决，本小节先详细探讨问题来源，然后在之后的章节里阐述相关研究，以及我们提出的初步探索方案。

正如[4]中阐述的，当文件远小于HDFS块的大小（默认64M），并且有很多的这样的文件。每一个文件、目录和块，在HDFS中都被表示成一个对象，存在NameNode的内存里，每一个都会占用150bytes。举例来说，如果有10百万的小文件，每一个使用一个块，那么将使用3GB的内存。除此之外，HDFS也不能高效地存取小文件，读取小文件的时候，将会需要很多的查找以及不断地从DataNode到NameNode的跳跃。

总的来说，[3]总结问题如下：

每个块都只能操作一个文件，这会导致产生很多的块，里面的文件远小于块的大小，读取这样的块非常消耗时间。

NameNode为每个块和文件都保存了一个记录，并且存储在它的内存里，大数目的小文件将会需要大量的内存空间。

3.3 相关调研

初期选题阶段，阅读了大量在2017年的相关文献，其名称基本上可以概括为“A Novel Approach to Improve the Performance for Massive Small Files in HDFS”，相关的总结将在后面列出。现在先阐释以前出现的部分优化方案。

3.3.1 Improved HDFS

改进的HDFS结构[5]包括两个部分：

- 客户端部分，整合小文件到一个大文件。
- 数据节点部分，满足缓存源管理

图3表现了这两点。改进的HDFS模型是基于索引的，属于同一个目录的文件会被整合到一个大文件，然后为每一个小文件建立索引，相应地减少了NameNode的负担。缓存机制则提高了小文件的读取效率，缓存管理是在DataNode里面的，当读取一个小文件的时候，在缓存里面的数据首先会被搜索。

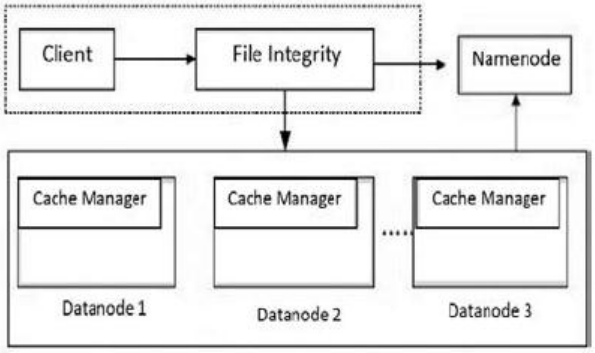


图 2：Improved HDFS Framework

每一个整合的大文件都有一个索引文件，包含了偏移量和每个原小文件的长度。整合的步骤如下：

- 在每个目录里，根据文件的大小排序，然后把一个接一个把小文件写入大文件里面
- 确定所有小文件的总和
- 确定大文件的大小，然后把结果和HDFS块的大小进行比较索引文件是依据偏移量和每个小文件的大小来创建的，为了把这个大文件存储在一个块里，文件大小应

该小于块的大小。如果比它大，就会分割成几个块。

3.3.2 Extended HDFS

这个方法[6]扩展了HDFS并且被命名为Extended HDFS（EHDFS）.EHDFS是基于索引的，当小文件的数目过于庞大，就会使索引文件很大，更新它会很困难。为了提高读取效率，使用了预取机制。EHDFS提供了一个改进的索引机制，以及预取索引信息。它有四个技术起了决定性作用：文件合并（file merging）、文件映射（file mapping）、预取（prefetching）、抽取（extraction）.图3表现了这几点。

File Merging 在文件合并的时候，NameNode只会保存大文件而不是里面所有的小文件，除了文件数据，一个索引表也会被放在每个块的开头，

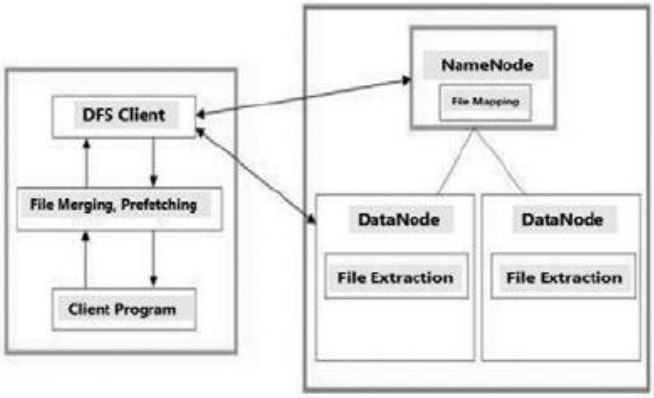


图 3：Extended HDFS Architecture

这张表包含了每个小文件的条目，每个表条目都是一个pair（offset，length），文件合并后的块的结构如图4，DataNode里面块的储存如图5.

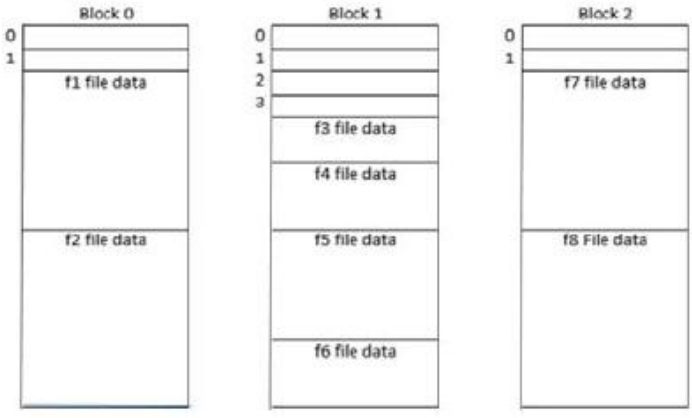


图 4：Block Structure after File Merging

File Mapping 文件映射是由NameNode完成的，文件映射意思是，将小文件的名字映射到包含这个小文件的合并文件上，为了获取所需的小文件的位置，一个包含小文件和合并文件名字的请求被发送到NameNode.对每个合成文件，NameNode都包含了一个称为Constituent File Map的数据结构，小文件名字和包含这个小文件的合成文件的逻

辑块号被映射。NameNode也提供了一个索引条目号在块的开头。Prefetch 文件合并不会提高读取的性能，它只会减少元数据的量，使用预取可以优化这一点，无论何时客户端试图读取一个小文件的时候，在同一个块的其他小文件的元数据都会被预取下来。

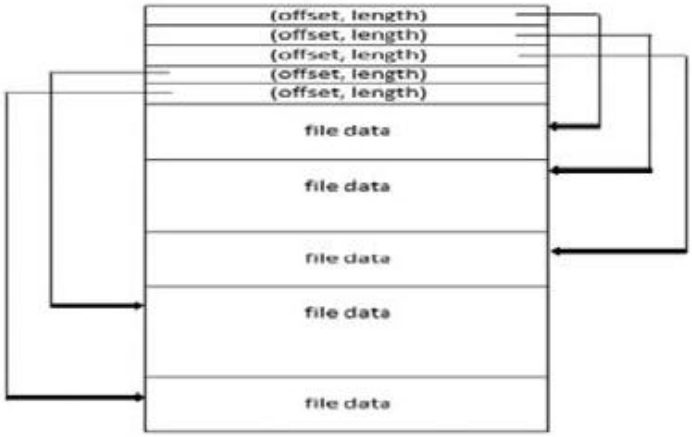


图 5：Structure of an Extended Block

File Extraction 由DataNode执行，将制定文件从一个块中提取出来。当读取一个文件的时候，客户端需要明确小文件的名字和合成文件的名字，用这两个信息去获取：包含这个文件的块、包含这个块的DataNode、在NameNode里面的索引表。

3.3.3 小结

这几种方法在[3]中已经总结，基本上是2012-2014年发表的，从2017年发表的相关论文来看，近期的很多论文都是在早期一些实践基础上的改进，包括算法的改良，以及适用于不同领域的方法。

3.4 新思路

3.4.1 对上文的总结

上文针对DFS做了详细的介绍，并且指出了DFS的存取问题，特别地，选择了一个通用性比较强的HDFS作为主要研究对象，探讨了它相关的问题，以及自从问题出现以来，各种的解决思路。在各路优化策略之后，是否仍有进步的空间？以及这样解决问题的方法是否仅仅适用于HDFS？

在[7]的启发下，我们做了如下的分析

3.4.2 基于预测的文件预取

在上文指出，大部分分布式文件系统有这样的特点，即：写一次读多次，那么去解决读取的问题会有更大的价值以及进步空间。要解决读取的问题，就是要解决读取

速度的问题，类似于使用多级缓存，在DFS中要解决的问题就是如何准确地把文件预取到本地文件系统，最终关于预取的问题，就是关于预测的问题。

结合上文，我们把问题缩小到对小文件的读取预测。当用户需要读取大量小文件的时候，特别是长时间对文件进行读取的时候，可以通过学习读取模式来预测下一步可能的内容，再通过预取，等到用户用到该文件的时候可以更加节省时间。

3.4.3 使用神经网络

当前神经网络作为一个强大的技术在序列预测问题方面，比如说自然语言处理，文本理解。而在DFS预测预取方面使用神经网络目前没有看到任何相关的研究，因而本次研究是首次的。

对此做几点说明

- 普通的深度神经网络主要针对标准化的数据，比如固定长度的特征向量或者图片转化成的RGB张量，对于序列化的数据使用RNN已经有了很多的应用。针对RNN的梯度消失和梯度爆炸问题，学界给出的改进LSTM则解决了这个问题。
- 普通的预测问题可以看成回归问题，但是对于海量小文件的问题，由于数量非常大，而且目标很稀疏，此时传统的回归学习已经非常不适用了。
- 使用神经网络必然会增加时间的开销，这也是我们所担心的一个问题，但是考虑到目前已经有相关的神经网络加速手段，比如FPGA，而我们仅仅只用做一个开端性的工作即可。
- 目前神经网络发展迅速，已经不需要自己从零开始构建，我们打算使用现有的库Keras（TensorFlow作为底层）来搭建。
- 考虑到时间的因素，我们以准确率作为主要的度量。

除此之外，可能会遇到以下的几点问题

- 神经网络准确率优化困难
- 数据集的寻找或者构建
- DFS和神经网络的结合

4 可行性分析

前期的调研报告提出，可以从存储安排优化和文件预存取两方面来改进海量小文件的存取问题。以下主要讨论文件预存取的可行性。

4.1 理论依据

4.1.1 分布式文件系统存取文件的流程

以Hadoop为例，其存取文件的流程如下：

写文件

- client向namenode发送创建文件的请求。
- namenode检测待创建的文件是否存在，如果可以创建，则返回待写入的datanode的地址。
- client向datanode中写入文件，当一个datanode中的block写入完成后，会异步地在其他的datanode中的block进行复制。
- datanode之间、datanode和client、client和namenode之间反馈上传完成的确认信息

读文件
- client向NameNode发送读取文件的请求
- namenode返回待读取文件所在的datanode的存储地址
- client从datanode中读取文件

单从存取文件的流程来看，似乎存取文件的瓶颈是网络带宽，但这其实只是对大文件而言的。对于海量的小文件，存取速度则会受限于硬盘。我们有这样的经验，当在电脑上删除大量的小文件时，速度会变得非常慢，甚至于只有每秒几K。这是因为尽管文件总大小可能不大，但是文件数量过多，磁盘磁头需要不断寻道，然后才对文件进行操作，真正删除文件所用时间并没有占多少。对于分布式文件系统也是一样，需要在短时间内找到大量小文件的地址，性能可想而知。

一个比较粗暴的解决办法就是，提升硬件性能，也就是换一块性能更好的硬盘，比如固态硬盘。但这其实并不会有太大的改善。有人曾经做过测试，向固态硬盘中拷贝100万个大小只有几十字节的小文件，文件总大小是几十兆，但拷贝速度仍然只有每秒几十K。

另一种办法是，提前读取或写入用户（或者应用等等）可能存取的文件，也就是预测用户（或者应用）的行为，进行预存取。这样就避免接触到硬件上的瓶颈，在尽可能不改变DFS架构和硬件配置的情况下提升性能。这正是我们要采取的办法。

4.1.2 几个通过预测提高性能的案例

- CPU分支预测(branch prediction)

CPU通过流水线技术来提高效率。所谓流水线，就是将一条指令的执行分成几个阶段，在同一时刻执行不同指令的不同阶段，以达到提高吞吐率的目的。

但是当执行到一个分支跳转语句的时候，由于不知道下一条指令究竟是什么，流水线不得不阻塞，一直到分支跳转语句执行完，确定下一条指令的地址才能继续。这样就会降低速度。

分支预测就是为了解决这个问题而提出的，它预测分支的结果并立即沿预测方向执行。如果分支预测正确，那么显然就节省了等待分支指令执行的时间，如果预测错误，就会导致流水线的停顿，比等待分支指令执行完产生额外的时间开销，因为需要将已经填入流水线的指令清除。因此，提高分支预测的准确率，可以有效提高CPU的效率。

- 内存预取

WindowsXP：Prefetch，即预取，但这里的预取比较低级，它只是在某个程序运行时，记录下这个程序经常从硬盘读取的文件。当下一次启动一个程序时，系统会到那

些记录中查找有无关于这个程序的记录，如果有，则预先将那些可能需要用到的文件载入内存，然后才载入该程序。可以看出，这里的预取只有在程序启动时才会进行，之前仅仅是记录以下而已。

WindowsVista及以后：Superfetch，这比Prefetch要高级不少，它会记录用户的使用习惯（如经常在某个时刻启动某个程序），然后自动预先（不同于Prefetch的被动预先）将硬盘中的文件载入内存。这个和我们将要采用的解决DFS小文件存取问题的策略很类似。

4.2 技术依据

4.1节中给出了通过预测来提升DFS的海量小文件存取性能的理论可行性，下面主要介绍神经网络以及神经网络如何用于预测。

4.2.1 神经网络简介

人工神经网络（ArtificialNeuralNetworks，ANNs），简称神经网络，顾名思义，它是模仿动物大脑的神经网络的行为特征而提出的一种计算模型，其根本目的在于处理各种信息。

- 基本结构

动物大脑的神经网络是由一个个神经元通过突触相互连接而成的，人工神经网络中自然也有相似的结构。

图6是一个人工神经细胞。$x1,x2,x3,x4,x5$是神经细胞的输入信号，$w1,w2,w3,w4,w5$是相应的输入信号的权重，反映了该信号对神经细胞状态的影响程度，由此可以得到对于该神经细胞的激励值$\Sigma i=15wixi$，然后再通过一个激活函数$f(x)$，将激励值映射成神经细胞的输出值$f(\Sigma i=15wixi)$。
常用的激活函数有sigmoid函数、tanh函数、ReLU（rectifiedlinearunits）函数等等。

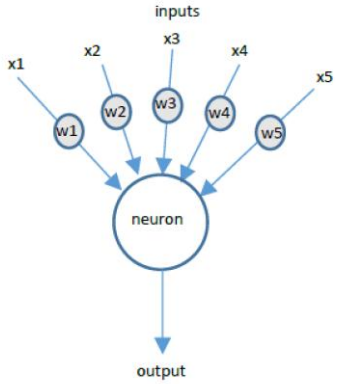


图 6：人工神经细胞

通过多个这样的神经细胞，就可以组成一个神经网络。最简单的一种神经网络是

前馈网络（feed-forward network），它由输入层（inputlayer）、隐藏层（hiddenlayer）和输出层（outputlayer）这三部分组成，其中隐藏层可以有任意多层。每一层的神经细胞只与前一层相连，信息只会向前传递，而同一层的神经细胞没有连接，如图7。

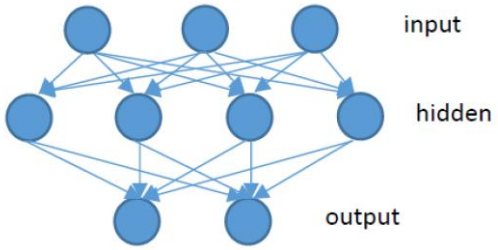


图 7：一个单层前向反馈神经网络

当然还有更复杂的神经网络，如深度神经网络（DeepNeuralNetwork，DNN）、卷积神经网络（ConvolutionalNeural Network，CNN）、递归神经网络（RecursiveNeuralNetwork，RNN）等等。下一节中将详细介绍RNN，这是我们采用的预测模型。

- 训练算法

所谓神经网络的训练，就是通过给定一组训练集（trainingset），根据神经网络的实际输出，不断调整输入信号的权重，使输出结果与期望输出之间的误差达到最小。从数学的角度来讲，神经网络的训练就是求损失函数（lossfunction，可以粗略理解为实际输出与期望输出之差）的最小值。由此可以产生以下几种训练算法：梯度下降算法、牛顿算、共轭梯度法

4.2.2 循环神经网络[8]

递归神经网络（Recursive Neural Network，RNN）可分为结构递归神经网络（Recursive Neural Network，RNN）和时间递归神经网络（Recurrent Neural Network，RNN，即循环神经网络），通俗地讲，结构递归神经网络可理解为空间上的循环，而循环神经网络则是时间上的循环。

我们要根据用户已有的存取文件的行为来预测用户将来可能存取的文件，这些已经存在的行为可以看成一个时间上相关的序列，要根据已有的序列来预测序列的下一个是什么，就好像根据一个句子来预测句子的下一个单词。传统的神经网络对此无能为力，因为一个句子的前后单词不是独立的，预测下一个单词是什么需要用到之前的信息，这就需要神经网络对之前的信息进行记忆，因而用到循环神经网络。这似乎有点像时序逻辑电路和组合逻辑电路的区别。

- 基本结构

图 8 是一个循环神经网络，可以看出，神经细胞与同一层的其他神经细胞及自身都有连接，这种结构使得神经网络可以记忆上一个时刻的输出，并影响到现在的输出。

如果我们将循环神经网络按照时间顺序展开的话，如图 9，就会发现，这其实是一种链式结构，这也暗示了循环神经网络可以用来处理序列相关的问题。

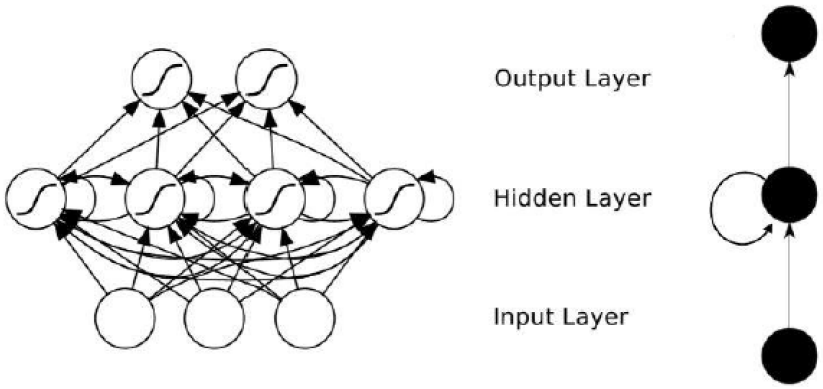


图 8：循环神经网络

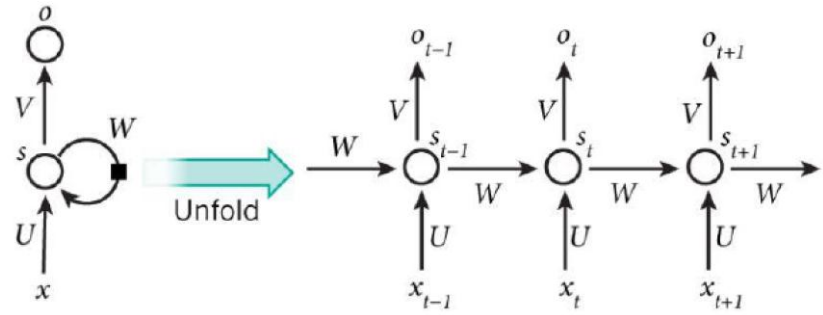


图 9：循环神经网络的展开

- 缺陷

从理论上来讲，循环神经网络可以处理任意长度的序列，但其实还有很多其他依赖因素。 举一个简单的例子：The clouds are in the sky.循环神经网络可以很容易就预测出最后一个单词是 sky，因为它距离预测这个单词需要的信息 clouds 很近。而对于另一个句子：The cat, which already ate a bunch of food, was full.预测单词 was 则会有一定的困难，因为预测这个单词需要的信息 cat 离得太远了。这就是所谓的长期依赖问题（Long-Term Dependencies），当预测位置与预测所需要的相关信息的间隔太远时，循环神经网络会失去学习连接到那么远的信息的能力。

应用领域我们已经提到，循环神经网络擅长处理序列信息，因此具有以下用途：

- 语言模型与文本生成（Language Modelling and Generating Text）：判断一个语句正确的可能性，或者根据已有语句预测下一个单词。
- 机器翻译（Machine Translation）
- 语音识别（Speech Recognition）：根据一段声音信号，判断对应的语句。

4.2.3 长短期记忆网络

- 简介

长短期记忆（Long Short-Term Memory，LSTM）网络由 Hochreiter 和 Schmidhuber

(1997) 提出，它解决了 RNN 的长期依赖问题[9]。

LSTM 是一种特殊的循环神经网络，它只是对 RNN 的隐藏层做了修改。

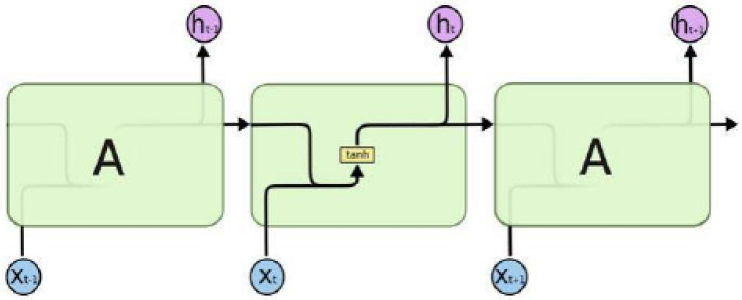


图 10：一般 RNN 的重复模块

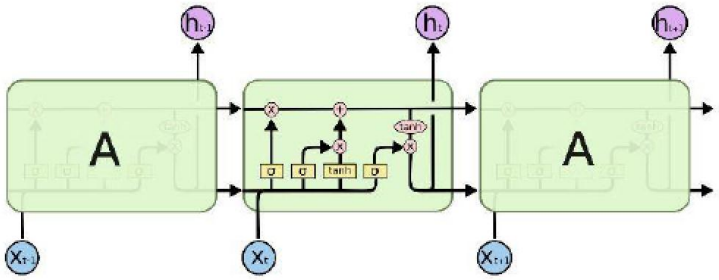


图 11：LSTM 的重复模块

如图 10、11，一般的循环神经网络的（时间上）重复模块只有一个层，而 LSTM 的重复模块则有四个层，且这四个层以一种特殊的方式相互连接。

LSTM 的关键在于其神经细胞状态的流传，如图七。神经细胞的状态就在这样一条时间链上前进，每个时刻决定保留、更新或丢弃细胞状态的某些信息，而这个决定是由三个门来控制的。

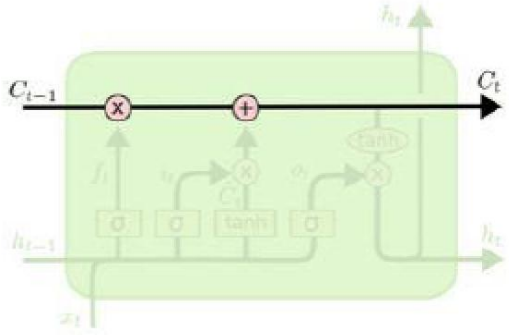


图 12：神经细胞的状态控制

- 确定丢弃哪些信息：通过忘记门层（forget gate layer）完成，该门会读取此时的输入x_t和上一个时刻的输出 h_{t-1},通过$sigmoid$激活函数向细胞状态C_{t-1}中的每个数字输出一个 0 到 1 之间的数值，1 表示保留，0 表示丢弃。
- 确定更新哪些信息：分为两步，首先通过输入门层（input gate layer）（激活函数是$sigmoid$函数）确定更新哪些信息，然后通过一个$tanh$ 层向细胞状态中加入一个新的候选值向量。用数学公式表示的话就是：

$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$, C_t^* =tanh ($W_C \cdot [h_{t-1}, x_t] + b_C$)。

- 更新信息。
- 确定输出信息：分为两步，首先通过一个$sigmoid$层确定输出哪些信息，然后将这些信息经过处理之后输出。

4.3 创新点

利用神经网络预测用户行为，进行文件的预存取来提升 DFS 性能。

当然，通过文件预存取来提升性能的方法已经有很多论文提出过，但不同之处在于，我们采用了神经网络来预测，经过训练之后的神经网络的预测准确率应该是更为可观的。

4.4 总结

种种成功的预测案例表明，通过预测，确实很有可能提高 DFS 海量小文件存取的效率，而不是受限于硬件的瓶颈。而神经网络，主要是 LSTM，给我们提供了预测用户行为的手段，因此，我们认为这个课题的可行性是比较高的。

5 概要设计

5.1 总体设计

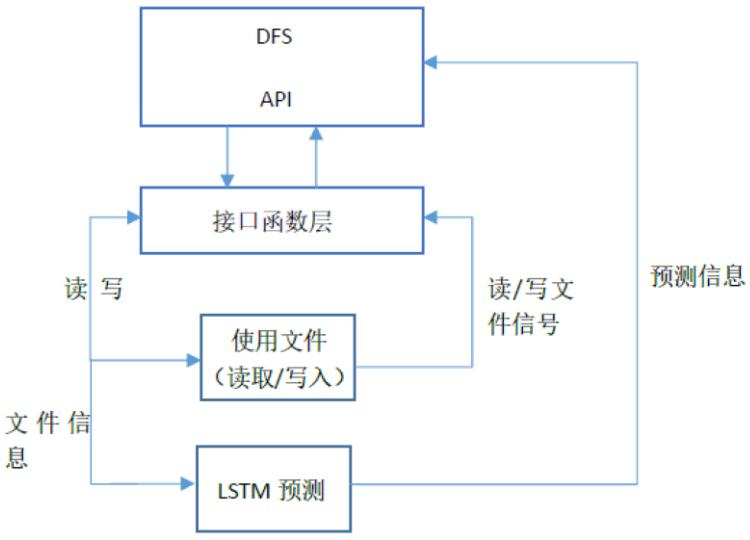


图 13：总体设计框架

5.2 DFS 文件存取信息的获取

文件存取信息可以通过分析 DFS 的日志得到，但这一步存在的问题是，我们没有

那么多的小文件可供存取，训练神经网络的数据集不够大，谈不上海量二字。对此，我们只能采用其他手段来手动构建文件存取信息。

一种办法是，通过机器学习根据一些特征，如文件名称、存取时间、权限等等，产生文件读写日志，对这种方法的细节尚不了解。

5.3 LSTM 的训练和预测

Keras 是由 python 编写的基于 theano/tensorflow 的深度学习框架。由于 Keras 的易用性和高效性，我们决定采用 Keras 来搭建神经网络。 神经网络的训练采用 5.2 节中生成的数据集。

5.4 DFS 预取

神经网络完成预测后，我们要将预测结果传给 DFS，为此，我们需要写一个接口函数来协同各个模块（DFS 和 LSTM）之间的工作。

6 详细设计及预测效果

6.1 预测

为了达到接口的可重用性，使用预测模块单独使用一个部分，并且提供简单的借口给其他模块。预测使用 python，用到的神经网络使用 keras 来搭建。

6.1.1 静态预测

预先训练好模型，直接使用，而且在预测的时候，不对新产生的数据进行训练。但是可以在一个阶段之后可以使用新产生的数据对模型的参数进行更新。这种方式相对减少了负载。本课题中的与DFS交接的部分使用了这个方式。下面主要介绍所用数据集和模型。

- 数据集

目前科研界对于file access patterns的研究非常少，相关的数据集也非常非常少，在查询了很多相关的论文之后，一共看到了如下几种解决办法：

- 使用DFSTrace的，这个界面上提供了相关论文以及使用的工具，但是，这个数据集是在 2 0 年前搜集的，而且使用的解析工具以及不能使用，如果重新构建的话，需要耗费大量时间。
- 使用另一个搜集的数据集，具体论文为《Characteristics of File System Workloads》,这个数据集也是将近 2 0 年前搜集的，而且我没有找到相关的地址，无法使用。

- 使用Traces and Snapshots Public Archive,这个是比较近期搜集的，但是这个是对文件系统的一个Snapshot，每一天都有一个，而且文件非常大，比较不容易解析出文件的变化，同样无法使用。

基于以上，最终选择自行使用代码（见https://github.com/OSH-2018/X-dll/tree/master/LSTM/generate_data.py）构建的数据集。

- 模型及预测效果

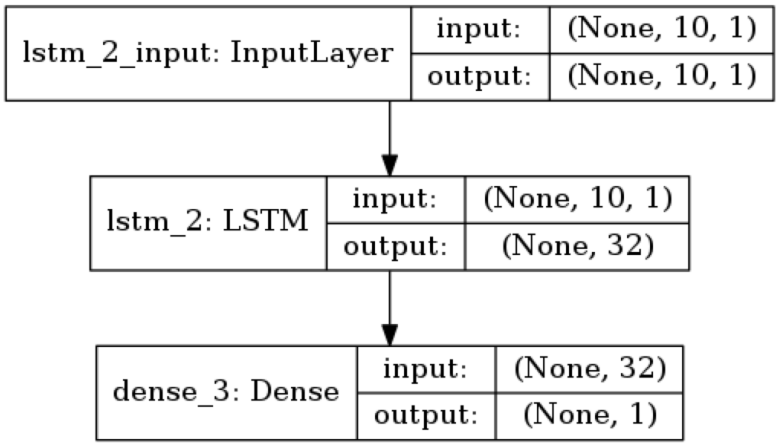
假设文件访问可预测，也就是说，有某种规律性，直观的或者隐藏的，并且提取到的信息足够用于预测。在探索性质的基础上，很容易就想到，所有能够使用的性质都可能蕴含信息，所以首先利用所有的信息。比如：

- name, 如果只采用字母数字编码，有可能针对name有相应的规律，比如有可能都是前子串相同，"data1","data2"等等。但是直接利用这种编码所需要的训练数据和模型体量很大，而且可能提供的信息量很少。
- id, 文件的id不同于PC值，虽然同样可以作为唯一的标识符，但是这种id并不具有位置的意义。
- extension, 文件的后缀。这个信息量非常少，可以直接使用上一个作为下一个预测。
- directory, 目录，或者节点，这是很重要的一个信息，以下详述。
- size, 相同时期文件访问的大小大致相同，使用上一个作为下一个预测，不需要预测，而且所含信息很少。
- protection, owner, operation, 均同上一个。
- created, modified,access, 关于时间的信息，的确包含信息，但是即便预测出来了也很难使用。

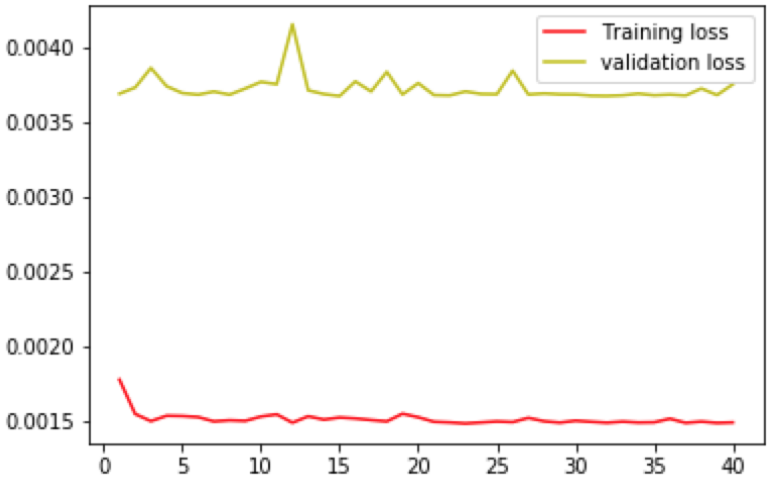
总的来说，最重要的两个可以使用的是id与directory,但是为了探索的完整性，实际上这些信息都使用了。从本质上来说这些内容应当结合起来形成一个统一的预测网络，但是考虑到以下几点，最后还是选择了id和directory：

- 信息的形式不一致，很难标准化
- 为了便于拓展，有可能增加新的信息来源，但是使用统一模型，就需要完全重新构建。
- 不同信息之间可能能够作为相互的补充，这种补充也许可以通过网络的连接发掘出来。但是实际上这种联系很少，而所付出的代价很大。

id使用的模型如下：

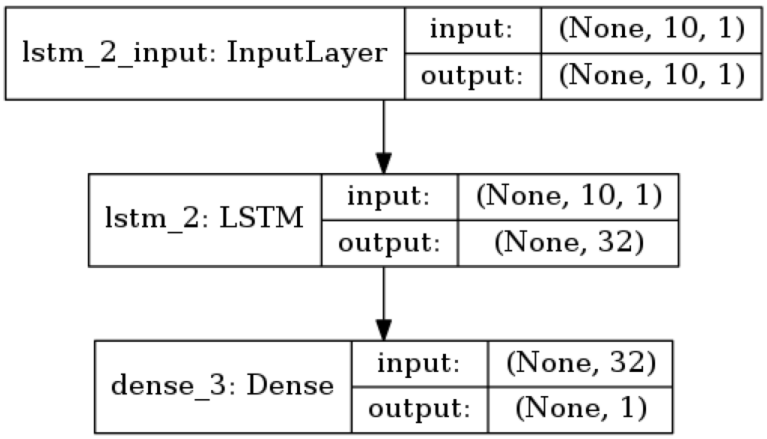


预测效果如下：

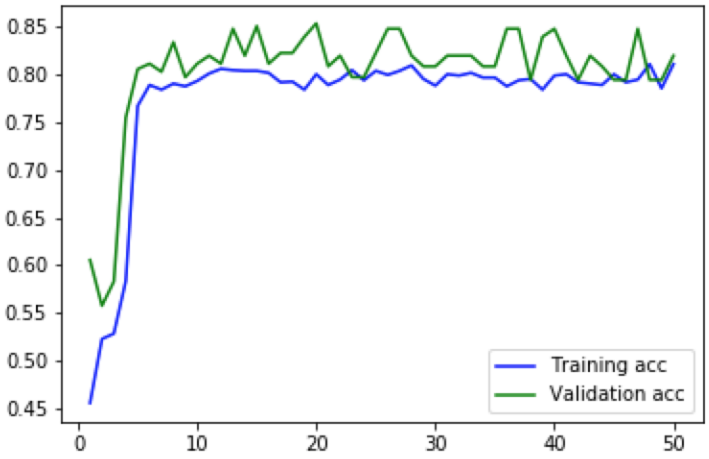


预测的效果可以从validation loss 来看，training loss 与validation loss 严重分离，代表拟合效果不好。效果不好的根本原因是数据量不够而且不是很贴合实际。同时也使用了LSTM, GRU, SimpleRNN，以及相应的bidirectional方式，效果差不多。

directory使用的模型如下：



最终结果如下：



看起来准确率比较高，但也是因为这里构建的数据集比较简单。

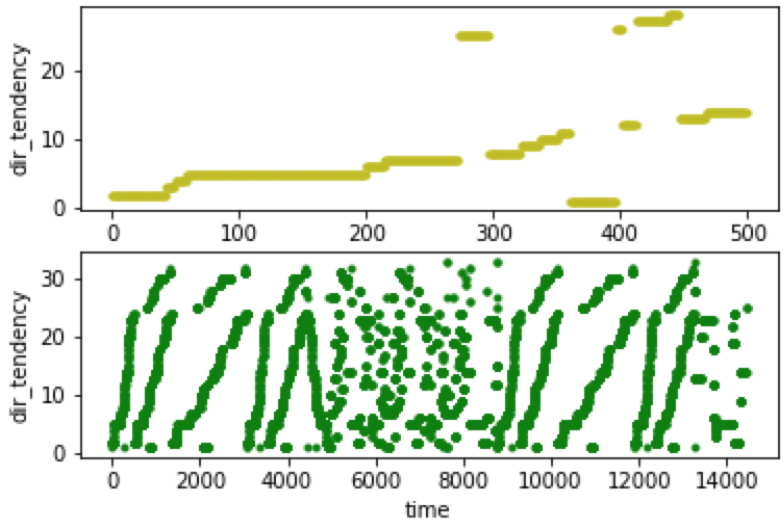
6.1.2 动态预测

- 数据及预处理

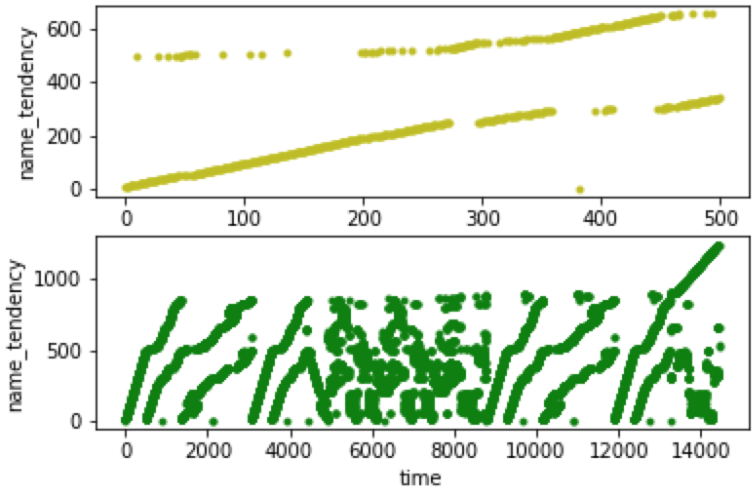
使用了strace跟踪make busybox时候的系统调用

strace -f -F -t -e trace=file -o ./trace make

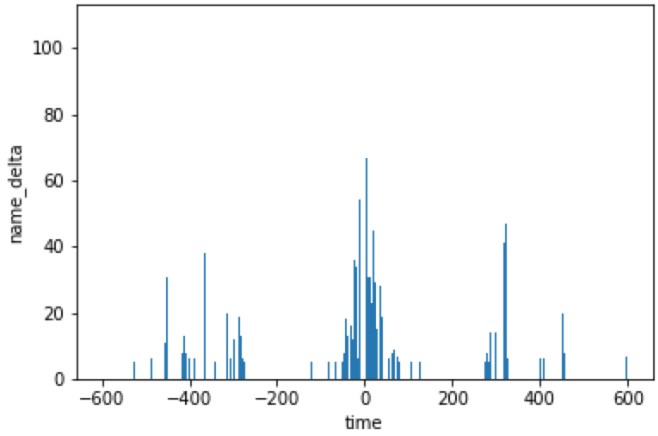
形成的数据大概有217 MB，但是为了简化问题，本次实验只使用busybox 内含的文件作为测试数据，而不使用其他库文件，也不涉及目录的处理，只考虑在二级目录下的文件。按文件出现的顺序对其进行编码，也包括对目录重新编码，得到dir的趋势如下（上面的是局部放大，下面的是整体的）：



id的趋势如下（上面的是局部放大，下面的是整体的）：



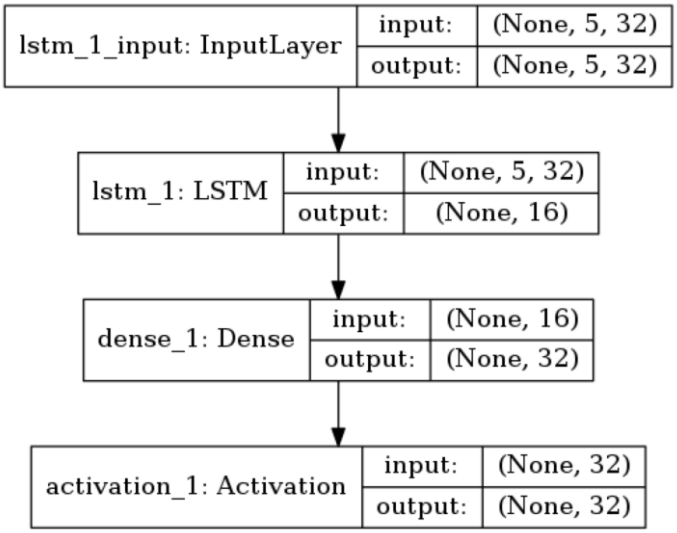
从这里可以看出来，当按照出现的顺序进行编码的时候，有明显的可提取特征，或者趋势。但是直接对文件的名称进行编码，搜索空间太大，无法做到，从而考虑id的差值作为数量，为此，可以先统计一下delta的分布



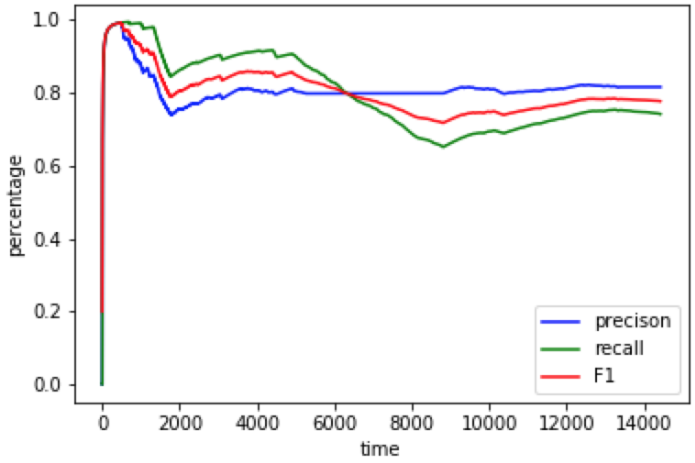
总体的delta序列有14441个，需要编码的delta有966个，但是从图中可以看到，这里的分布比较集中，为了缩小搜索空间，可以只取出现最频繁的。我们取了30个最频繁出现的delta,这30已经囊括了70.5%的delta 序列。

- 模型及预测效果

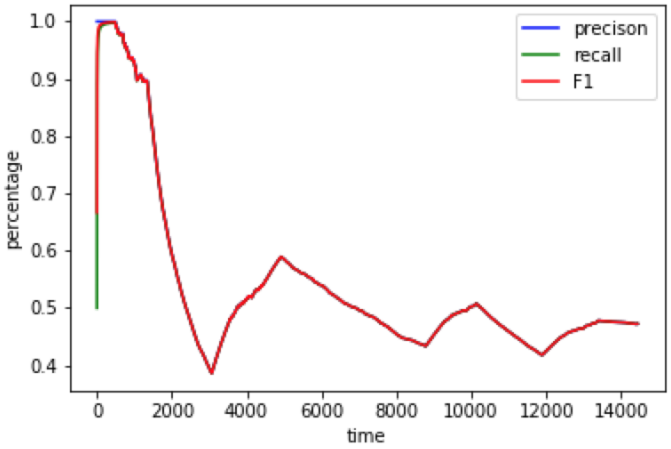
所用模型如下：



由于这是动态预测，所以是逐步拟合和预测的，结果如下：



设计一个baseline，用前一个delta作为本次预测，其结果如下：



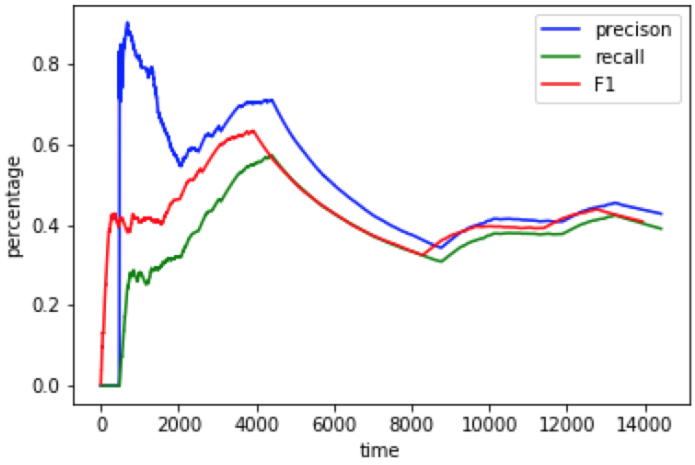
可见使用模型明显好于baseline

- 模块化

  为了将动态预测模块化，建立一个py文件，里面的主要函数即dynamic_predict，使用

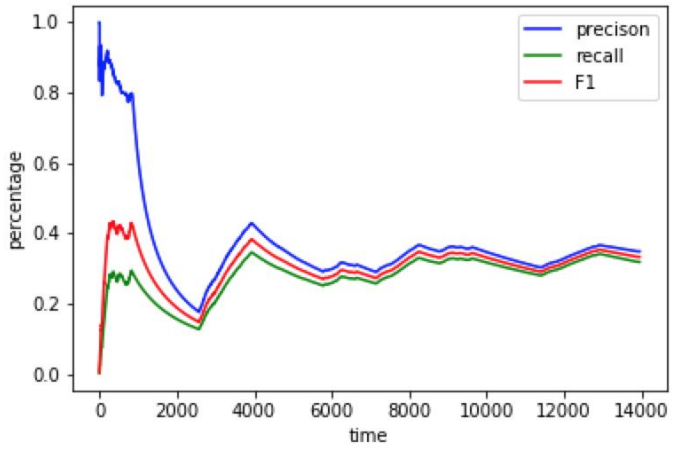
的模型如上文构建。区别在于，在预测开始运行的时候，并不知道出现最频繁的文件，所以需要随着预测的进行不断更新。

使用strace获得的数据测试结果如下：



可以看到，效果大概比之前减半，其根本原因就在于，一开始并不知道最频繁文件。这其实就说明，最频繁文件信息会严重影响预测。

另外使用last successor作为baseline,即使用上一次的访问时候的下一个文件作为预测，得到效果如下：



从此处可以看到，在这种动态情况下，神经网络并不比last successor优秀多少。做以下几点说明

- 最频繁文件信息会严重影响预测预测
- 在没有频繁文件集的时候，神经网络效果与last successor相差不大
- 当第二次使用同样的文件预测时，预测效果将翻倍

6.1.3 其他模型对比

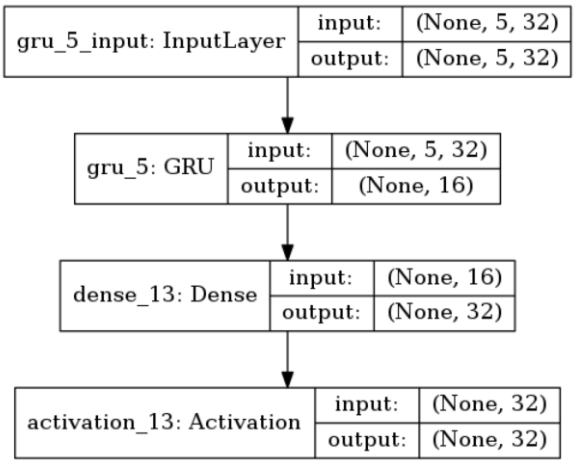
以下只附上一些模型构建图与测试数据图，在此之前做一些说明

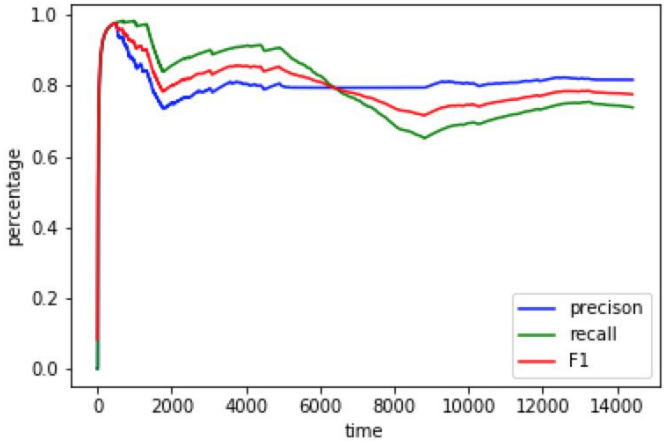
- 使用GRU加dropout=0.2可以达到与LSTM同样的效果
- 使用LSTM或者GRU,加上Bidirectional得到的效果均不如原始的

- 将窗口增大到10,并且使用一维卷积（此处以及以下均包括池化层），或者两层卷积，或者卷积加循环神经网络均效果均不如原始GRU

- GRU

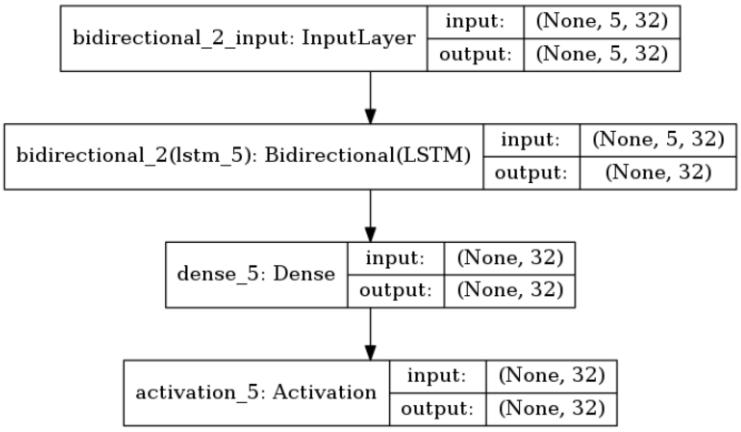
使用 GRU，并且使 dropout=0.2,得到的效果与 LSTM 非常接近。而且 GRU 的运算量小于 LSTM.

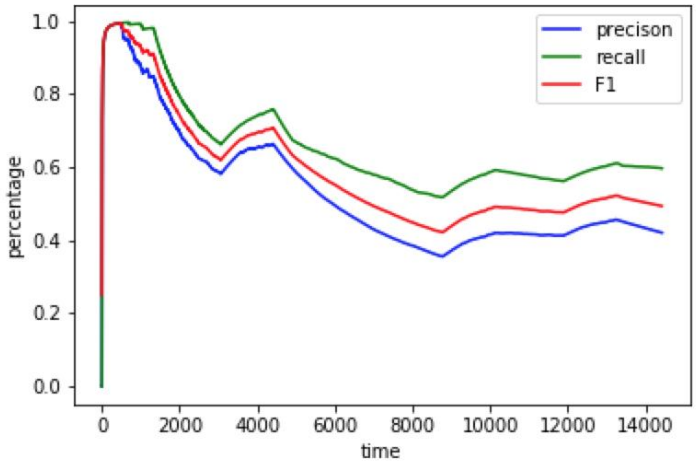




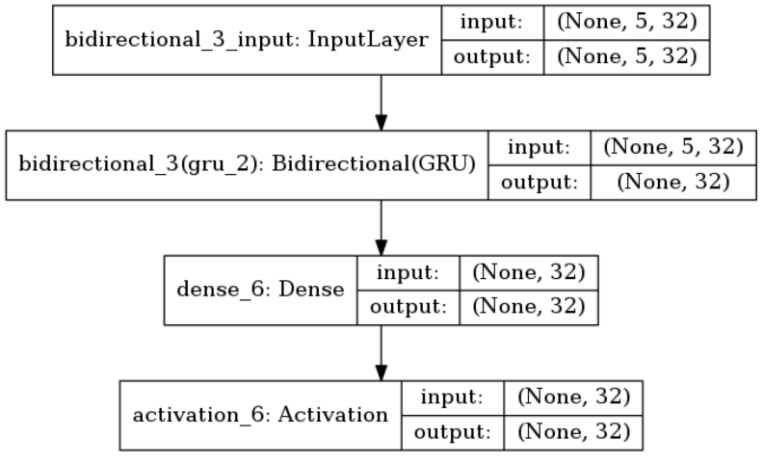
- LSTM+Bidirectional

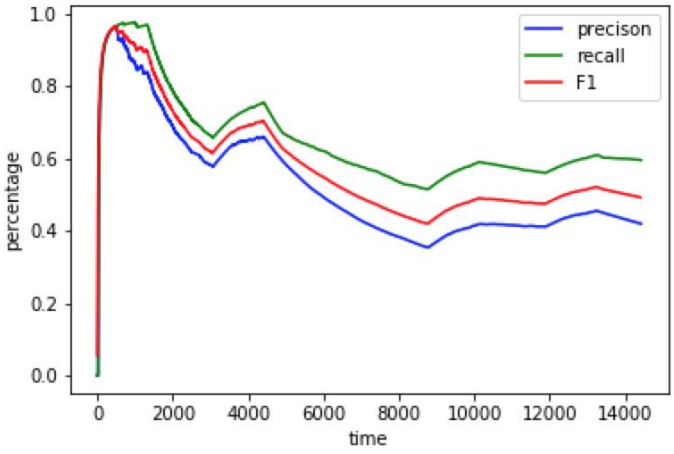
无论对 GRU 还是对 LSTM,使用 bidirectional 之后的效果都不如原本的，这说明，文件 access 序列倾向于正向有序。





- GRU+Bidirectional





6.2 添加模块

6.2.1 概述

图 中的接口函数层即为添加模块。考虑以下两种添加方案：

- 利用 HDFS 提供 java 接口添加模块，HDFS 与用户交互的程序不变(shell)
- 利用其他编程语言(shell，python 等等)，在 HDFS 之外连接预测模块和 HDFS，HDFS 与用户交互的程序为该接口

方案一相当于在 HDFS 内部增加一部分代码，性能可能很好，但比较麻烦；方案二相当于在两个函数（进程）之间再写一个函数，用于参数的传递，但联系的层数变多了，性能可能有所下降。由于时间有限，采用方案二，用 Python 实现。

6.2.2 具体实现

处理过程如下：

- 用户输入命令，如果是上传/下载命令，则先检查HDFS/本地临时文件夹，如果要上传/下载的文件已经存在，则直接移动，否则令HDFS执行下载/上传。
- 在上传/下载完文件后，调用预测模块，根据文件名得到进行预取/预存的文件ID，然后进行预存/预取，保存到HDFS/本地临时文件夹。

此处的文件ID不同于linux下的文件ID，这里的ID是由于预测模块接口的要求而自定义的ID，设置为从1开始。由此，还需要维护一个name2id的字典和一个id2name的字典。具体实现过程请参阅github仓库中的源代码interface.py。

6.2.3 问题及改进方案

目前的实现还存在如下几个问题：

- 随着存取文件的增多，要维护的字典也越来越大，时间开销和空间开销
- 每存取一次文件就进行一次预存取，当一个client连续存取大量文件时，性能会下降

可能的对应解决方案：

- 可以定期清除一部分name和ID
- 将预测和上传/下载单独作为一个/多个进程，设置两个队列来存放要预存取的文件名，在执行用户的上传/下载命令时同时进行预存取(该项已基本实现，详见github仓库中的源代码interface_optimized.py)

6.3 总结

本组最初考虑到 I/O 速度远慢于计算速度，所以想用计算和预取的方式来优化小文件存取。但由于现阶段神经网络的计算速度较慢，对存取速度的优化结果并不让人满意，而文件预取的准确率差强人意。我们认为，在不久的将来，神经网路的计算速度有较大幅度的提高，那时我们的优化结果将仅受优化准确率的限制。从长远角度来看，本组所做的工作还是有一定意义和价值的。

7 参考文献

[1] Dhruba Borthakur : *The Hadoop Distributed File System: Architecture and Design* Hadoop Project Website.

[2] Songling Fu, Ligang He: *Performance Optimization for Managing Massive Numbers of Small Files in Distributed File Systems* IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL.26, NO.12, DECEMBER 2015 3433

[3] Sachin Bendea, Rajashree Shedge: *Dealing with Small Files Problem in Hadoop Distributed File System* Procedia Computer Science 79 (2016) 1001-1012

[4] XiaoJun Liu, Chong Peng, ZhiChao Yu: *Research on the Small Files Problem of Hadoop* International Conference on Education, Management, Commerce and Society (EMCS 2015)

[5] Chen, J., Wang, D., Fu, L., and Zhao, W., *An Improved Small File Processing Method for HDFS*, International Journal of Digital Content Technology an its Applications (JDCTA), Vol.6, pp.296-304, Nov 2012

[6] Gurav, Y.B., Jayakar, K.P., *Efficient way for Handling Small Files using Extended HDFS*, International Journal of Computer Science and Mobile Computing, Vol.3, pp.785-789, June 2014

[7] Milad Hashem Kevin Swersky: *Learning Memory Access Patterns* arXiv:1803.02329v1 [cs.LG] 6 Mar 2018

[8] Simon Haykin:Nerual Networks and Machine Learning, Third Edition

[9] Hochreiter S, Schmidhuber J.: Long short-term memory.[J]. Neural Computation, 1997, 9(8):1735-1780

[10]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[11]Hashemi M, Swersky K, Smith J A, et al. Learning Memory Access Patterns[J]. 2018.

[12]Kroeger T M, Long D D E. Design and Implementation of a Predictive File Prefetching Algorithm[C]// General Track: 2001 Usenix Technical Conference. USENIX Association, 2001:105-118.

[13]Patra P K, Sahu M, Mohapatra S, et al. File access prediction using neural networks[J]. IEEE Transactions on Neural Networks, 2010, 21(6):869-882.

[14]Lipton Z C, Berkowitz J, Elkan C. A Critical Review of Recurrent Neural Networks for Sequence Learning[J]. Computer Science, 2015.

[15]Jozefowicz R, Zaremba W, Sutskever I. An empirical exploration of recurrent network architectures[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2015:2342-2350.

[16]Chung J, Gulcehre C, Cho K H, et al. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling[J]. Eprint Arxiv, 2014.

[17]Greff K, Srivastava R K, Koutník J, et al. LSTM: A Search Space Odyssey[J]. IEEE Transactions on Neural Networks & Learning Systems, 2015, 28(10):2222-2232.

[18]Salehinejad H, Sankar S, Barfett J, et al. Recent Advances in Recurrent Neural Networks[J]. 2017.

[19]Patra P K, Sahu M, Mohapatra S, et al. File access prediction using neural networks[J]. IEEE Transactions on Neural Networks, 2010, 21(6):869-882.
[20]Mummert L, Satyanarayanan M. Long Term Distributed File Reference Tracing: Implementation and Experience[M]. Carnegie Mellon University, 1994.
[21]Kroeger T M, Long D D E. Design and Implementation of a Predictive File Prefetching Algorithm[C]// General Track: 2001 Usenix Technical Conference. USENIX Association, 2001:105-118.