caffe中的信号处理机制

训练中断了该怎么办?

我们在使用caffe框架训练的时候会遇到下面几种情况导致模型训练中断，（这几种是我想到可能会常遇到的问题，当然导致模型中断训练有很多的原因，这里就不做一一介绍了）。

1、我们给终端发送了一个ctrl-c信号。

2、电脑由于某种不可抗力原因导致宕机。

3、我们给终端 发送了一个ctrl-z信号。

如果遇到这种问题我们该怎么办？可能有些同学会重新启动程序来从头训练过，不得不说这是一种最简单且最暴力的解决方法，但是如果我们已经迭代了好几万次了，那么从头训练过是不是成本太高了。其实对于一些简单的中断情况，caffe已经为我们考虑了。caffe通过捕捉信号和snapshot机制已经较好的解决了这一问题，关于信号捕捉和snapshot机制是本节的重点。那在讲正式内容之前我们先来看一下上述三个问题该如何解决，我们以mnist数据的集的Lenet训练为例。

第一个和第三个问题

首先我们得确认我们保存的快照的位置，这个位置在lenet_solver.prototxt中的snapshot_prefix这个位置进行设置，可以是绝对路径也可以是相对路径。但是记住这个路径一定要存在，caffe并不会帮你去创建这么一个路径，除非自己改动caffe源码的SnapshotFilename函数（src/caffe/solver.cpp），对于创建路径我们可以利用boost::filesystem中的库函数进行创建，这里就不列出来了。

```
snapshot_prefix: "mnist/lenet"     //这个文件夹就是我们存snapshot的位置
# solver mode: CPU or GPU
solver_mode: GPU
type: "Adam"
```

确认了snapshot保存位置之后我们运行训练程序然后向中断发送一个ctrl-c信号，那么整个训练程序将会被中断，最新的snapshot将会被保存到mnist/lenet的目录下，我们可以看到其中的snapshot文件有：

```
yupefieng@yupefieng:~/Documents/tutorial/caffe/mnist/lenet$ ls -l
总用量 10120
-rw-rw-r-- 1 yupefieng yupefieng 1725006 5月  12 22:53 lenet_solver_iter_5000.caffemodel
-rw-rw-r-- 1 yupefieng yupefieng 3448896 5月  12 22:53 lenet_solver_iter_5000.solverstate
-rw-rw-r-- 1 yupefieng yupefieng 1725006 5月  12 22:54 lenet_solver_iter_6000.caffemodel
-rw-rw-r-- 1 yupefieng yupefieng 3448896 5月  12 22:54 lenet_solver_iter_6000.solverstate
```

可以看到，有两组文件，其中5000编号的这一组是程序每迭代5000次自动保存的，而6000编号这一组是收到了ctrl-c发送的信号保存的。那我们重启训练的时候可以将6000这一组编号的训练状态参数和模型权重参数加载到训练网络当中，我们可以具体看一下如何加载。我们需要对examples/mnist/train_lenet.sh进行修改，修改为：

```
#!/usr/bin/env sh
set -e
./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt\
    --snapshot=mnist/lenet/lenet_solver_iter_6000.solverstate    //添加我们snapshot的文件路径
```

那下一次运行整个程序就会从之前停止的6000次那边开始训练。这样就解决了训练中断这个问题。

第二个问题

为什么介绍这么一个命令呢？因为这个命令很有用。想象下面的几种场景：

1、你是用ssh客户端连接到你的服务器进行训练的，当你写完代码debug代码之后启动训练程序进行训练，你估计整个训练过程可能要5个小时，在这训练的五个小时的时间内你可以去外面放松一下（比如去酒吧聊会天等等），在和朋友推杯换盏的时候突然接到老板的微信问你的模型训练的怎么样了？你回答说快训练好了，等会就可以有结果了，这个时候你飞奔回实验室打开电脑一看，（我的天）实验室刚刚断网了一会，你连接服务器的网络终端中断了，而且断网的时间就在你出去之后的几分钟，所以也就是说你啥也没有训练。因为网络终端一断就会导致你的服务器上面对应的训练进程也会中断。

2、或者你在服务器上开启了一个终端进行训练，但是在你不在的时候，其他人因为一些失误将你的终端给关掉了，这就会导致你的训练进程中断了，而他们却没有对其进行重启训练。

3、或者你在训练一个较大型网络的时候，你的同学想用服务器来做一个模型测试（可能就只需要半个小时），由于你的模型太大了，只能关闭你的程序释放GPU资源来供他人使用。

应该还有一些其他的情景，我就不列举了。这些问题都可以通过合理使用ctrl-z来完成。我们在终端输入ctrl-z的时候并没有终端进程，而只是停止进程，这个时候进程占有的资源（CPU、内存、GPU等）会被释放。那我们可以简单看一下和ctrl-z相关的命令。

```
CTRL+Z    //挂起进程并放入后台
jobs      //显示当前暂停的进程
bg %N     //使第N个任务在后台运行(%前有空格)
fg %N     //使第N个任务在前台运行
```

通过上述几个命令操作就可以解决上面提到的问题，大家可以自行去尝试一下。另外关于这些命令的底层实现可以参考CSAPP的第八章异常控制流这一章节。

caffe的信号管理类(SignalHandler)

caffe框架对SIGINT信号和SIGHUP信号进行信号捕捉动作，对于SIGINT信号caffe会做一个snapshot，保存当前的训练参数，然后整个训练过程就会终止了。对于SIGHUP信号，caffe收到该信号只会做一个snapshot，当并不会退出整个训练过程。那接下来我们从源码的角度来看一下caffe是如何对这两个信号进行捕捉的吧。

首先看一下SignalHandler的头文件。

(include/caffe/util/signal_handler.h)

```
class SignalHandler {
 public:
  // Constructor. Specify what action to take when a signal is received.
  SignalHandler(SolverAction::Enum SIGINT_action,          //收到SIGINT信号对应的动作
                SolverAction::Enum SIGHUP_action);        //收到SIGHUP信号对应的动作
  ~SignalHandler();
  ActionCallback GetActionFunction();                      //接口供外界来调用
 private:
  SolverAction::Enum CheckForSignals() const;
  SolverAction::Enum SIGINT_action_;
  SolverAction::Enum SIGHUP_action_;
};
```

首先对于SignalHandler这个类来说，其主要目的是用来捕捉SIGINT和SIGHUP信号，并且将捕捉到的信号相对应的意图保存成SolverActio::Enum类型的数据，那程序如何得知程序接收到了信号呢？答案是通过轮询机制，我们可以看到SignalHandler类中有一个public函数GetActionFunction可以返回一个函数指针，而其他程序段调用该函数指针可以获知捕捉到的信号情况。我们先来看一下SolverAction::Enum数据类型和ActionCallback函数指针类型吧。

(include/caffe/solver.hpp)

```
namespace SolverAction {
enum Enum {
    NONE = 0,  // Take no special action.
    STOP = 1,  // Stop training. snapshot_after_train controls whether a
    // snapshot is created.
    SNAPSHOT = 2  // Take a snapshot, and keep training.
};
}
```

对于SolverAction有三种动作，

第一种是NONE，表示接收到信号后啥也不干。

第二种是STOP，程序会保存当前iteration的snapshot，然后终止进程。

第三种是SNAPSHOT，程序会保存当前iterator的snapshot，然后继续训练，不会终止进程。

(include/caffe/solver.hpp)

```
typedef boost::function<SolverAction::Enum()> ActionCallback;
```

这条语句定义了一个函数指针类型，其所对应的函数的传入参数为void，但是该函数返回的值为SolverAction::Enum类型的数据。记住一般function和bind是对应起来的，后面我们会在Signalhandler.cpp文件中看到bind函数的用法。

注意：C++11也引入了function+bind的机制，这个机制非常强悍，有的时候可以替代C++的虚函数的用法。我会单独开辟一节来具体分析一下function+bind的使用机制。

volatile关键字的用法

这个关键字是从线程安全的问题中引出来的，我们在使用多线程访问公共资源的时候需要对其加锁，但是对于一些编译器来说加锁可能并不能完全解决问题。因为有些编译器会针对并发代码做过度的优化从而导致其执行并发程序的时候出错，我们举一个简单的例子。

```
x=0;
Thread1             Thread2
lock();             lock();
x++;                x++;
unlock();           unlock();
```

这个代码一看应该没啥问题，因为我们使用了锁对公共资源x进行了保护，但是如果编译器为了提高x的访问速度，将x放到了某个寄存器内，那么我们知道不同线程的寄存器是各自独立的，因此Thread1先获得锁，则程序执行可能会呈现如下的情况：

1、[Thread1]读取x的值到某个寄存器R[1] (R[1] = 0)。

2、[Trhead1] R[1]++。（由于之后还要可能访问x，所以Thread1暂时不把R[1]写回x）。

3、[Thread2]读取x的值到某个寄存器R[2] (R[2]=0)。

4、[Thread2] R[2]++(R[2] = 1)。

5、[Thread2]将R[2]写回x（x = 1）。

6、[Thread1]将R[1]写回x（x = 1）。

所以这个问题可以总结为：**编译器为了提高速度将一个变量缓存到寄存器内而不写回**，所以针对这一问题我们可以使用volatile关键字来解决。

(src/caffe/util/signal_handler.cpp)

```
static volatile sig_atomic_t got_sigint = false;
static volatile sig_atomic_t got_sighup = false;
```

注意我们还在变量前面添加了sig_atomic_t关键字，这个关键字的作用是该数据类型为原子类型的数据，我们给got_sigint赋值将在一个指令周期内完成。这么一看是不是用了双重保护呢?

信号捕捉函数handle_signal(int signal)

(src/caffe/util/signal_handler.cpp)

```
void handle_signal(int signal) {
    switch (signal) {
    case SIGHUP:
      got_sighup = true;                //捕捉到SIGHUP信号
      break;
    case SIGINT:                        //捕捉到SIGINT信号
      got_sigint = true;
      break;
    }
  }
```

注册信号函数HookupHandler()

我们可以看下面的代码，信号注册函数是使用了sigaction函数，该函数提供更深层次的信号捕捉机制。比如它设定了信号捕捉结束之后将会重启原来被信号中断的系统调用，同时当信号捕捉函数进行的时候该程序将会对其他信号屏蔽，不会被其他信号给影响了。

(src/caffe/util/signal_handler.cpp)

```
void HookupHandler() {
    if (already_hooked_up) {
      LOG(FATAL) << "Tried to hookup signal handlers more than once.";
    }
    already_hooked_up = true;
    struct sigaction sa;
    // Setup the handler
    sa.sa_handler = &handle_signal;
    // Restart the system call, if at all possible
    sa.sa_flags = SA_RESTART;    //重启系统调用。
    // Block every signal during the handler
    sigfillset(&sa.sa_mask);     //在处理信号捕捉函数的时候屏蔽其他的信号
    // Intercept SIGHUP and SIGINT
    if (sigaction(SIGHUP, &sa, NULL) == -1) {
      LOG(FATAL) << "Cannot install SIGHUP handler.";
    }
    if (sigaction(SIGINT, &sa, NULL) == -1) {
      LOG(FATAL) << "Cannot install SIGINT handler.";
    }
  }
```

信号注销函数UnhookHandler()

该函数是信号捕捉函数注销函数，它与上面代码最大的区别就是所对应的信号信号捕捉函数不同，在UnhookHandler函数中其捕捉函数是默认函数SIG_DFL。表示默认捕捉函数，那就是原来该干嘛干嘛。

(src/caffe/util/signal_handler.cpp)

```
void UnhookHandler() {
    if (already_hooked_up) {
      struct sigaction sa;
      // Setup the sighup handler
      sa.sa_handler = SIG_DFL;          //按照默认处理
      // Restart the system call, if at all possible
      sa.sa_flags = SA_RESTART;
      // Block every signal during the handler
      sigfillset(&sa.sa_mask);
      // Intercept SIGHUP and SIGINT
      if (sigaction(SIGHUP, &sa, NULL) == -1) {
        LOG(FATAL) << "Cannot uninstall SIGHUP handler.";
      }
      if (sigaction(SIGINT, &sa, NULL) == -1) {
        LOG(FATAL) << "Cannot uninstall SIGINT handler.";
      }
```

```
      already_hooked_up = false;
    }
  }
```

检测信号函数CheckForSignals()const

前面我们可以看到在信号捕捉函数中，如果我们捕捉到了SIGINT信号，则将got_sigint设置为true，那么我们的检测信号函数一旦检测到捕捉到了SIGINT信号，它会往外界发送一个SIGINT_action动作。这样外界就可以采取相应的动作了。那下面的问题是如何让外界来获取信号的变化情况呢。

(src/caffe/util/signal_handler.cpp)

```
SolverAction::Enum SignalHandler::CheckForSignals() const {
  if (GotSIGHUP()) {
    return SIGHUP_action_;
  }
  if (GotSIGINT()) {
    return SIGINT_action_;
  }
  return SolverAction::NONE;
}
```

信号类接口函数

啊，看到了bind函数，这里bind函数就是将信号检测函数CheckForSignals一绑定扔给外界去调用，从而外界就可以获知信号的变化情况，并作出自己的决定了。

(src/caffe/util/signal_handler.cpp)

```
ActionCallback SignalHandler::GetActionFunction() {
  return boost::bind(&SignalHandler::CheckForSignals, this);
}
```

信号类的简单测试

根据前面的介绍我们可以根据自己的需求我们可以在信号类中增加其他的信号，那大家可以自己动手试试看。下面我主要做一下该信号的简单测试，看看如何使用该信号类。

(fish_test/test_signal_handler.cpp)

```
#include <iostream>
#include "caffe/util/signal_handler.h"
#include "caffe/solver.hpp"
using namespace caffe;
int main(){
    caffe::SignalHandler signal_request(caffe::SolverAction::Enum::STOP,
        caffe::SolverAction::Enum::SNAPSHOT);
    caffe::ActionCallback func = signal_request.GetActionFunction();
    while(1){
    //采用轮询机制
```

```
    if (func() == SolverAction::STOP){
        std::cout << "捕捉到SIGINT信号，接下来我们要停止" << std::endl;
        break;
    }
    else if(func() == SolverAction::SNAPSHOT){
        std::cout << "我们捕捉到了SIGHUP信号，我们继续"  << std::endl;
    }
    else{
    }
    }
}
```

信号类和caffe的snapshot机制结合

那我们看一下在caffe中信号类是如何使用的以及snapshot的机制是怎样的?

首先我们在train函数中生成了SignalHandler的一个对象，注意GetRequestedAction(FLAGS_sigint_effect)语句，首先FLAGS_sigint_effect是使用了google的gflags库，具体使用可以看google工具简介之(gflag)。

(tools/caffe.cpp)

```
 caffe::SignalHandler signal_handler(
    GetRequestedAction(FLAGS_sigint_effect),
    GetRequestedAction(FLAGS_sighup_effect));
```

在这里FLAGS_sigint_effect和FLAGS_sighup_effect都已经设定好了，具体看下面代码

(tools/caffe.cpp)

```
DEFINE_string(sigint_effect, "stop",
        "Optional; action to take when a SIGINT signal is received: "
        "snapshot, stop or none.");
DEFINE_string(sighup_effect, "snapshot",
        "Optional; action to take when a SIGHUP signal is received: "
        "snapshot, stop or none.");
```

因此FLAGS_sigint_effect就是“stop”，而FLAGS_sighup_effect的值就是“snapshot”。而GetRequestedAction函数的定义如下：

(tools/caffe.cpp)

```
caffe::SolverAction::Enum GetRequestedAction(
        const std::string& flag_value) {
    if (flag_value == "stop") {
        return caffe::SolverAction::STOP;                //对应的操作是保存snapshot然后终止进程
    }
    if (flag_value == "snapshot") {
        return caffe::SolverAction::SNAPSHOT;              //对应的操作是保存snapshot然后继续
    }
    if (flag_value == "none") {
        return caffe::SolverAction::NONE;
    }
    LOG(FATAL) << "Invalid signal effect \""<< flag_value << "\" was specified";
}
```

ok，前面我们的操作我们已经在程序中实例化了一个信号对象，那么该信号对象就可以接受外界给其发送的信号了。那么程序一旦收到了信号该如何处理呢，这就是后面要和snapshot机制一起结合起来分析的部分。

我们先看src/caffe/sovler.cpp中的Step函数部分，我们可以看到下面一段代码片段：

```
SolverAction::Enum request = GetRequestedAction();      //获取信号信息
        // Save a snapshot if needed.
        if ((param_.snapshot()
                    && iter_ % param_.snapshot() == 0
                    && Caffe::root_solver()) ||
                (request == SolverAction::SNAPSHOT)) {
            Snapshot();             //根据情况做snapshot
        }
        if (SolverAction::STOP == request) {
            requested_early_exit_ = true;       //如果是SIGINT信号则退出训练
            // Break out of training loop.
            break;
        }
```

Step函数每次迭代的时候都会轮询这个信号类接口，并且获知目前的信号状态。我们可以看到如果是状态是SNAPSHOT的话程序会做一次snapshot，然后继续执行。如果是STOP的话，则程序会中断执行，并跳到上一层代码中，并作snapshot保存动作。snapshot保存主要是保存两个东西，第一个是SolverParamete的值，第二个是NetParameter的值，具体的代码可以自己跟着Snapshot函数探索下去，难度不大。