浙江大学

本科课程报告

课程名称：网络基础

项目名称：基于 TCP 协议与 Python 中 Socket 模块的人群分布数据传输与远程 CNN 识别模型部署

姓　　名：杨笑波、姚博文、李超

学　　号：3160101465、3160105493、3160104368

学　　院：信息与电子工程学院

系：电子科学与技术

指导教师：王勇

目录

基于 TCP 协议与 Python 中 Socket 模块的人群分布数据传输与远程 CNN 识别模型部署

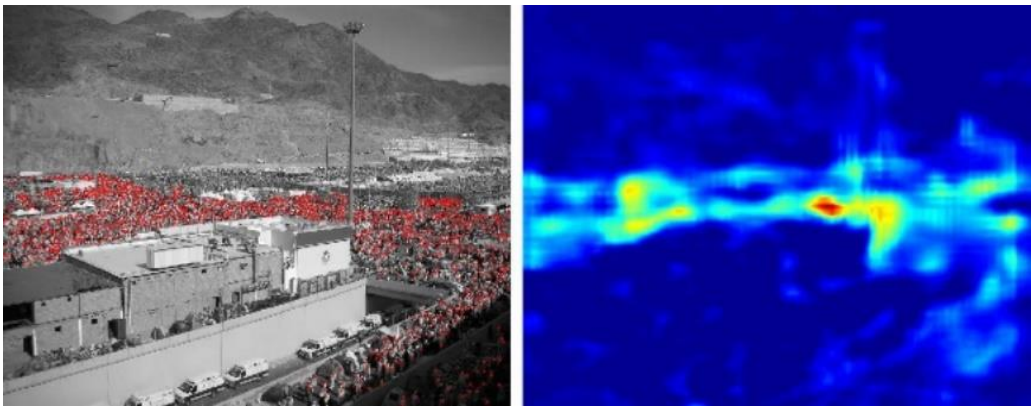
组员：杨笑波、姚博文、李超

摘要：本次 Project 中，我们利用 TCP 协议、通过基于 Python 的 socket 的模块，实现了人群分布数据的传输与远程卷积神经网络的模型部署和识别应用。具体流程与功能包括：将本地笔记本和树莓派设备上的摄像头获取的视频流进行编码后上传至服务器；服务器端利用训练好的基于 Caffe 的 CNN 模型，计算出每帧图片的人群分布密度图；最后，服务器再用类似的方法将结果回传到本地客户端上并显示。

1. 概述

1.1 问题提出

在今年的 SRTP 结题展示中，我们的一名组员遇到了一个问题：他们小组训练一个神经网络模型，可以读入摄像头的视频流，通过神经网络运算，给出视频中人群总数与人群分布的密度图像，效果如下所示。



然而，实时运行该模型需要有安装 Caffe 且带高性能 GPU 的服务器。而结题答辩的时候，各小组只能带笔记本电脑上台展示，不便于实时展示完成效果，便只能播放预先录制好的演示视频。

于是，我们提出，可以利用 TCP 协议、通过基于 Python 的 socket 的模块，将视频流编码传输至服务器端；远程运算完成后再回传至本地。如此将提升整个项目的完成度和易用性。

1.2 程序总体设计

运行在本地端的 client.py 文件能够本地的图片或视频流上传到服务器；

运行在服务器端的 server.py 文件能够通过 CNN 计算出密度图再回传至本地；

其中，server.py 函数调用的 socket_send_recv.py 文件中含有专门用于处理传输相关的函数。

1.3 最终成果简述

最终成果的演示视频包括 caffe_predict.mp4 和 caffe_predict_camera.mp4 文件，可在 res_video 文件夹中找到查看。

1.4 报告结构

本报告包括六大部分，分别为概述、基本原理与概念、程序具体实现原理、遇到的问题与解决方法、结果验证与分析，以及最后的总结心得。第一部分即为此部分，包括问题的提出背景、程序的总体设计、最终成果的简要概述以及本报告的结构概述。第二部分为本项目设计的基本概念与原理，包括 TCP 和 Python 中 Socket 应用相关内容。第三部分分文件详细解释程序的设计、实现与功能。第四部分设计程序设计与应用等过程中碰到的问题与解决方法。第五部分为本项目的结果验证与分析，包括使用 Wireshark 进行抓包验证等讨论。第六部分为组员在完成本次项目之后的感想与收获。

2. 基本概念与原理

2.1 TCP

2.1.1 TCP 原理概述

传输控制协议(Transmission Control Protocol, TCP)是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。在简化的计算机网络 OSI 模型中，它完成第四层传输层所指定的功能，是位于 IP 层之上，应用层之下的中间层。用户数据报协议(UDP)是同一层内另一个重要的传输协议。

TCP 的工作原理简述如下。应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，然后 TCP 把数据流分割成适当长度的报文段。之后 TCP 把结果包传给 IP 层，由它来通过网络将包传送给接收端实体的 TCP 层。TCP 为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的包发回一个相应的确认信息(ACK)；如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据包就被假设为已丢失并进行重传。TCP 用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算校验和。

2.1.2 发展历史与应用

TCP 是一个复杂的但同时又是在发展之中的协议。，发表于 1981 年的 RFC793 中说明的 TCP(TCP-Tahoe)的许多基本操作未作多大改动，是现今 TCP 的基础，尽管在此之后许多重要的改进被提出和实施。RFC1122：《因特网对主机的要求》阐明了许多 TCP 协议的实现要求。RFC2581：《TCP 的拥塞控制》描述了更新后的避免过度拥塞的算法。写于 2001 年的 RFC3168 描述了对明显拥塞的报告，这是一种拥塞避免的信号量机制。在 21 世纪早期，在所有因特网的数据包中，通常有大约 95%的数据包使用了 TCP 协议。

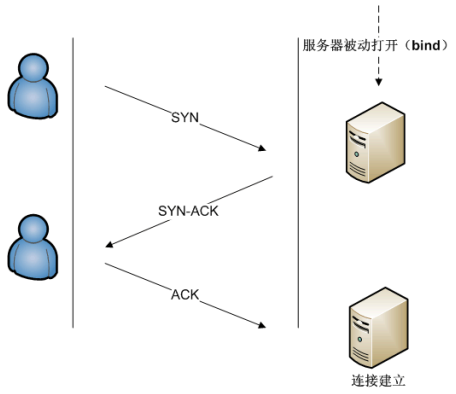
常见的使用 TCP 的应用层有 HTTP/HTTPS(万维网协议)，SMTP/POP3/IMAP(电子邮件协议)以及 FTP(文件传输协议)。不过，一些实时应用并不需要甚至无法忍受 TCP 的可靠传输机制：它们通常允许一些丢包、出错或拥塞，而不是去校正它们。通常不使用 TCP 的应用包括实时流多媒体(如因特网广播)、实时多媒体播放器和游戏、IP 电话(VoIP)等等。此外，在很多情况下，当只需要多路复用应用服务时，用户数据报协议(UDP)可以代替 TCP 为应用

提供服务。

2.1.3 运作方式

1) 连接建立

TCP 用三次握手（或称三路握手，three-way handshake）过程创建一个连接。在连接创建过程中，很多参数要被初始化，例如序号被初始化以保证按序传输和连接的强壮性。



上图表示一个 TCP 连接的建立过程。假定左侧用户运行的是 TCP 客户端程序，而右侧服务器运行的是 TCP 服务器程序。最初两端的 TCP 进程都处于 CLOSED 状态，通常是由一端打开一个套接字(socket)然后监听来自另一方的连接，这就是通常所指的被动打开(passive open)。服务器端被被动打开以后，用户端就能开始创建主动打开(active open)。

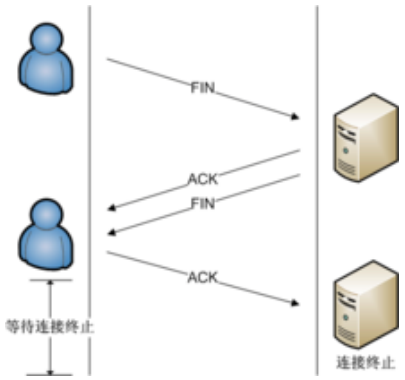
三路握手的具体过程如下：

- TCP 服务器进程先创建传输控制块 TCB，准备接受客户进程的连接请求。然后服务器进程就处于 LISTEN（收听）状态，等待客户的连接请求。如有，即做出响应。
- 客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三次握手的一部分。客户端把这段连接的序号设定为随机数 A；
- 服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机产生的序号 B；
- 最后，客户端再发送一个 ACK。此时包的序号被设定为 A+1，而 ACK 的确认码则为 B+1。当服务端收到这个 ACK 的时候，就完成了三次握手，并进入了连接创建状态。为了防止已失效的连接请求报文段突然又传送到了 B 而产生错误，A 需要于最后再发

送一次确认，避免所渭“已失效的连接请求报文段”建立连接。

2) 连接释放

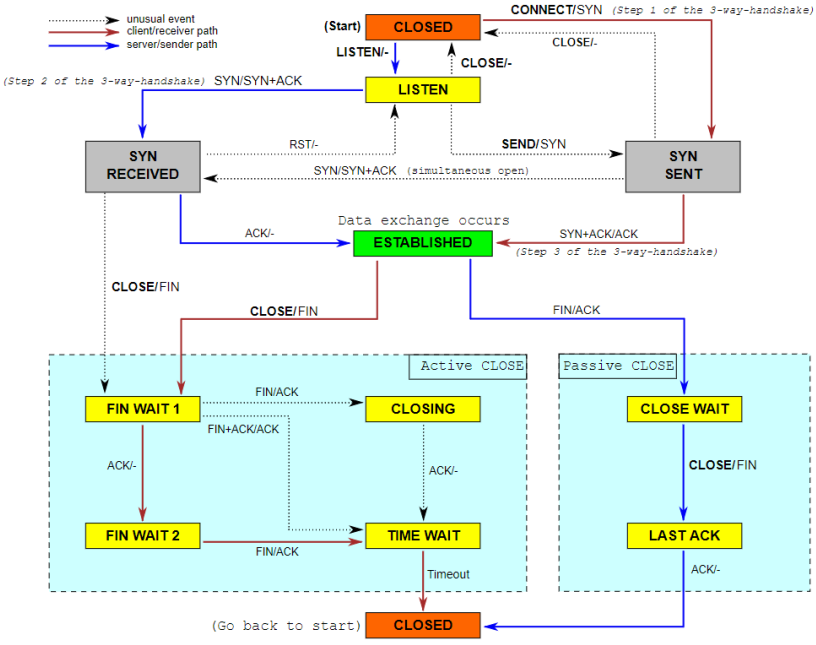
连接终止使用了四路握手过程(或称四次握手，four-way handshake)，在这个过程中连接的每一侧都独立地被终止。



当一个端点要停止它这一侧的连接，就向对侧发送 FIN，对侧回复 ACK 表示确认。因此，拆掉一侧的连接过程需要一对 FIN 和 ACK，分别由两侧端点发出。首先发出 FIN 的一侧，如果给对侧的 FIN 响应了 ACK，那么就会超时等待 2*MSL 时间，然后关闭连接。在这段超时等待时间内，本地的端口不能被新连接使用；避免延时的包的到达与随后的新连接相混淆。连接可以工作在 TCP 半开状态。即一侧关闭了连接，不再发送数据；但另一侧没有关闭连接，仍可以发送数据。已关闭的一侧仍然应接收数据，直至对侧也关闭了连接。

3) 有限状态机

总结以上连接建立与释放过程，可以得到 TCP 的整体工作有限状态机如下图所示。



4) 数据传输

在 TCP 的数据传送状态，很多重要的机制保证了 TCP 的可靠性和强壮性。它们包括：使用序号，对收到的 TCP 报文段进行排序以及检测重复的数据；使用校验和检测报文段的错误，即无错传输；使用确认和计时器来检测和纠正丢包或延时；流量控制；拥塞控制；丢失包的重传。以下将分被对可靠传输、流量控制、拥塞控制等进行介绍。

2.1.4 可靠传输

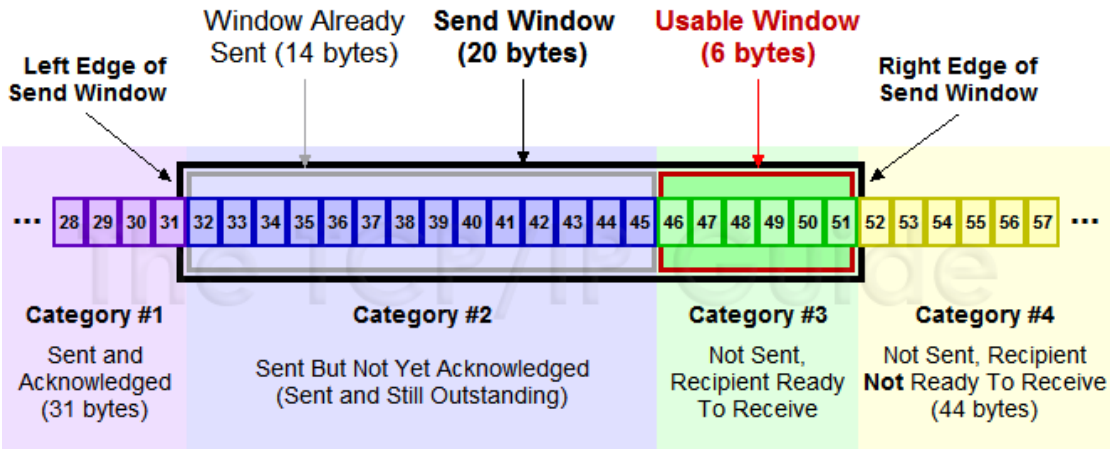
1) 滑动窗口

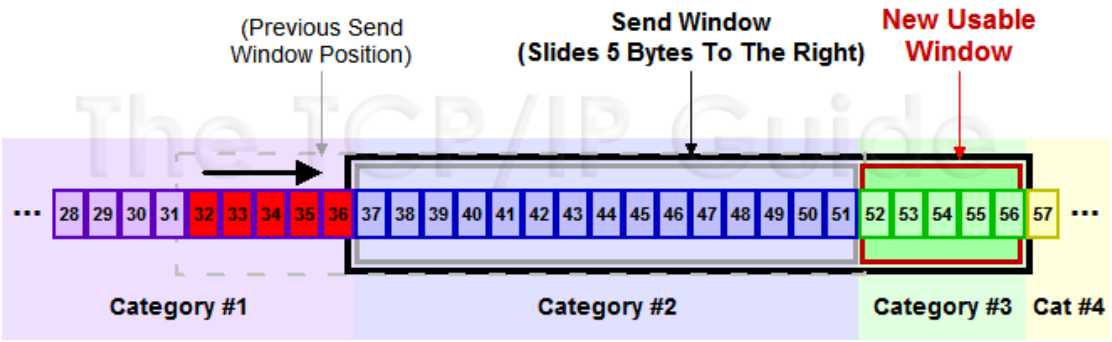
传输数据的每个部分被分配唯一的连续序列号，接收方使用数字并以正确的顺序放置接收到的数据包，丢弃重复的数据包并识别丢失的数据。滑动窗口协议中规定，对于窗口内未经确认的分组需要重传。

滑动窗口协议保证数据包的按序传输。发送窗口包括四个部分：已经发送成功并已经被确认的数据、发送但没有被确认的数据、尽快发送的数据以及未发送的数据。发送窗口可持续地维持一系列未经确认的数据包，因为发送方窗口内的数据包可能在传输过程中丢失或损

坏，所以发送过程必须把发送窗口中的所有数据包保存起来以备重传。发送窗口一旦达到最大值，发送过程就必须停止接收新的数据包，直到有空闲缓存区。

接收窗口的数据有 3 个分类，因为接收端并不需要等待 ACK 所以它没有类似的接收并确认了的分类：接收了数据但是还没有被上层的应用程序接收的数据，于缓存窗口之内；已经接收但是并未回复确认；有空位但是没有接受到的数据。当序列号等于接收窗口下限的数据包到达时，把它提交给应用程序并向发送端发送确认，接收窗口向前移动一位。发送窗口和接收窗口上下限无需相同，大小也无需相同，但接收窗口大小需保持固定，发送窗口大小可随着数据包而改变。





2) 超时重传

TCP 的发送方在规定的时间内没有收到确认就要重传已发送的报文段。对于重新传输时间的选择，TCP 采用了一种自适应算法，它记录一个报文段发出的时间以及收到相应的确认的时间。这两个时间之差就是报文段的往返时间 RTT。TCP 保留了 RTT 的一个加权平均往时间 RTTs(这又称为平滑的往返时间，S 表示 Smoothed。因为进行的是加权平均，因此得

出的结果更加平滑)。每当第一次测量到 RTT 样本时，RTTs 值就取为所测量到的 RTT 样本值。以后每测量到一个新的 RTT 样本，就按下式重新计算一次 RTT：

$$newRTT_s = (1-\alpha) \times (oldRTT_s) + \alpha \times (newRTT)$$

在上式中，0≤α <1。若α很接近于零，表示新的 RTTs 值和旧的 RTTs 值相比变不大，而对新的 RTT 样本影响不大(RTT 值更新较慢)。若选择α接近于 1，则表示新的 RTTs 值受新的 RTT 样本的影响较大(RTT 值更新较快)。RFC2988 推荐的α值为 0.125。用这种方法得出的加权平均往返时间 RTTs 就比测量出的 RTT 值更加平滑。

显然，超时计时器设置的超时重传时间 RTO(Retransmission Time-out)应略大于上面得出的加权平均往返时间 RTTs。RFC2988 建议使用下式计算 RTO：

$$RTO = RTT_s + 4 \times RTT_D$$

而 RTT_D 是 RTT 的偏差的加权平均值，它与 RTTs 和新的 RTT 样本之差有关。RFC2988 建议这样计算 RTT。当第一次测量时，RTT_D 值取为测量到的 RTT 样本值的一半。在以后的测量中，则使用下式计算加权平均的 RTT_D：

$$newRTT_D = (1-\beta) \times (oldRTT_D) + \beta \times |RTTs - newRTT|$$

这里β是个小于 1 的系数，它的推荐值是 0.25。

3) 选择确认

若收到的报文段无差错，只是未按序号，中间还缺少一些序号的数据，可以通过选择确认只传送缺少的数据而不重传已经正确到达接收方的数据。

前后字节不连续的每一个字节块都有两个边界：左边界和右边界。因此用四个指针标记这些边界。RFC2018 规定，如果要使用选择确认 SACK，那么在建立 TCP 连接时，就要在 TCP 首部的选项中加上“允许 SACK”的选项，而双方必须都事先商定好。

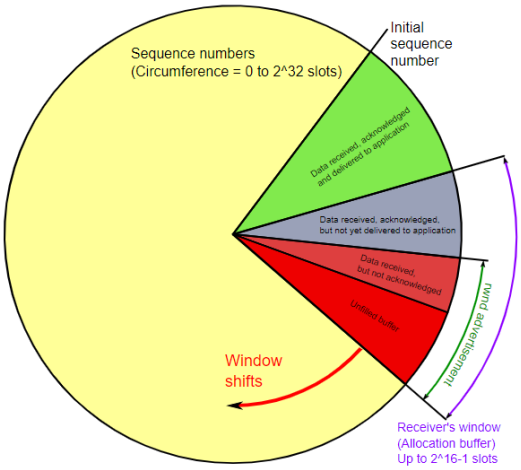
2.1.5 流量控制

流量控制用来避免主机分组发送得过快而使接收方来不及完全收下，一般由接收方通告给发送方进行调控。TCP 使用滑动窗口协议实现流量控制。接收方在“接收窗口”域指出还可接收的字节数量。发送方在没有新的确认包的情况下至多发送“接收窗口”允许的字节数量。接收方可修改“接收窗口”的值。

当接收方宣布接收窗口的值为 0，发送方停止进一步发送数据，开始了“保持定时器”(persist timer)，以避免因随后的修改接收窗口的数据包丢失使连接的双侧进入死锁，发送方

无法发出数据直至收到接收方修改窗口的指示。当“保持定时器”到期时，TCP 发送方尝试恢复发送一个小的 ZWP 包(Zero Window Probe)，期待接收方回复一个带着新的接收窗口大小的确认包。一般 ZWP 包会设置成 3 次，如果 3 次过后还是 0 的话，有的 TCP 实现就会发 RST 把链接断开。

如果接收方以很小的增量来处理到来的数据，它会发布一系列小的接收窗口。这被称作愚蠢窗口综合症，因为它在 TCP 的数据包中发送很少的一些字节，相对于 TCP 包头是很大的开销。解决这个问题，就要避免对小的 window size 做出响应，直到有足够大的 window size 再响应。TCP 包的序号与接收窗口的行为很像时钟，如下所示。



2.1.6 拥塞控制

拥塞控制是发送方根据网络的承载情况控制分组的发送量，以获取高性能又能避免拥塞崩溃(congestion collapse，网络性能下降几个数量级)。这在网络流之间产生近似最大最小公平分配。发送方与接收方根据确认包或者包丢失的情况，以及定时器，估计网络拥塞情况，从而修改数据流的行为，这称为拥塞控制或网络拥塞避免。TCP 的现代实现包含四种相互影响的拥塞控制算法：慢开始、拥塞避免、快速重传、快速恢复。此外，发送方还会采取前述“超时重传”方法。

2.1.7 报文段格式

TCP 虽然是面向字节流的，但 TCP 传送的数据单元却是报文段。一个 TCP 报文段分为首部和数据两部分，而 TCP 的全部功能都体现在它首部中各字段的作用。TCP 报文段的首部格式如下图所示，其中前 20 个字节是固定的，后面有 4n 字节是根据需要而增的选项(n 是

整数)。因此 TCP 首部的最小长度是 20 字节。

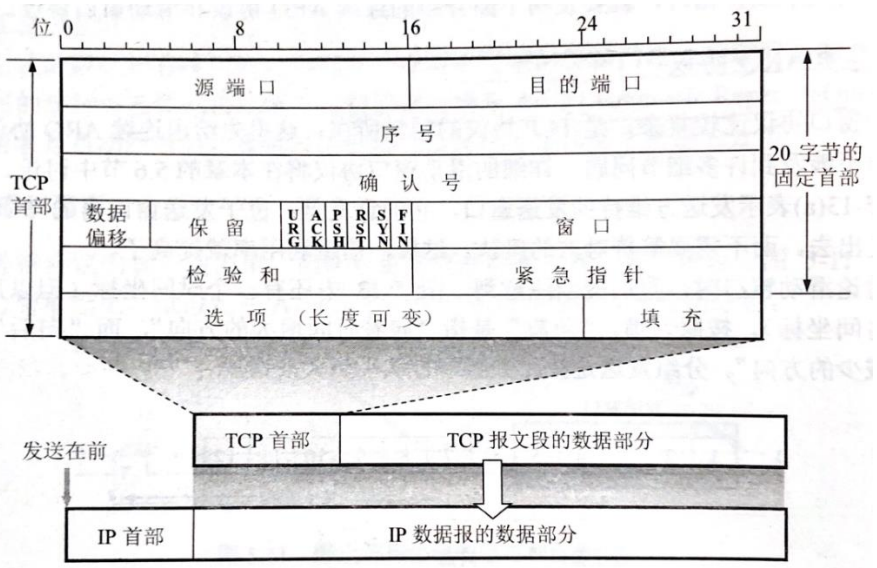


图 5-14 TCP 报文段的首部格式

2.2 Python Socket

2.1.1 概述

Python 提供了两个不同级别的网络服务：低级别的 Socket，提供了标准的 BSD Sockets API，可以访问底层操作系统 Socket 接口的全部方法；高级别的 ScoketServer，提供了服务器中心类，可以简化网络服务器的开发。

2.1.2 Socket 类型

Socket 类型以及相应的功能列出如下：

socket.AF_UNIX：用于同一台机器上的进程通信（既本机通信）。

socket.AF_INET：用于服务器与服务器之间的网络通信。

socket.AF_INET6：基于 IPV6 方式的服务器与服务器之间的网络通信。

socket.SOCK_STREAM：基于 TCP 的流式 socket 通信。

socket.SOCK_DGRAM ：基于 UDP 的数据报式 socket 通信。

socket.SOCK_RAW：原始套接字，普通的套接字无法处理 ICMP、IGMP 等网络报文，而 SOCK_RAW 可以；其次 SOCK_RAW 也可以处理特殊的 IPV4 报文；此外，利用原始套接

字，可以通过 IP_HDRINCL 套接字选项由用户构造 IP 头。

socket.SOCK_SEQPACKET：可靠的连续数据包服务。

2.1.3 Socket 函数

Socket 函数包括服务器端函数、客户端函数以及公共函数等。

1) 常用的服务器端函数与对应功能包括：

socket.bind(address)：将套接字绑定到地址，在 AF_INET 下，以 tuple(host, port)的方式传入，如 socket.bind((host, port))；

socket.listen(backlog)：开始监听 TCP 传入连接，backlog 指定在拒绝链接前，操作系统可以挂起的最大连接数，该值最少为 1，大部分应用程序设为 5 就够用了；

socket.accept()：接受 TCP 链接并返回（conn, address），其中 conn 是新的套接字对象，可以用来接收和发送数据，address 是链接客户端的地址。

2) 常用的客户端函数与对应功能包括：

socket.connect(address)：链接到 address 处的套接字，一般 address 的格式为 tuple(host, port)，如果链接出错，则返回 socket.error 错误。

socket.connect_ex(address)：功能与 socket.connect(address)相同，但成功返回 0，失败返回 errno 的值。

3) 常用公共函数与对应功能包括：

socket.recv(bufsize[, flag])：接受 TCP 套接字的数据，数据以字符串形式返回，buffsize 指定要接受的最大数据量，flag 提供有关消息的其他信息，通常可以忽略。

socket.send(string[, flag])：发送 TCP 数据，将字符串中的数据发送到链接的套接字，返回值是要发送的字节数量，该数量可能小于 string 的字节大小。

socket.sendall(string[, flag])：完整发送 TCP 数据，将字符串中的数据发送到链接的套接字，但在返回之前尝试发送所有数据。成功返回 None，失败则抛出异常。

socket.recvfrom(bufsize[, flag])：接受 UDP 套接字的数据 u，与 recv()类似，但返回值是 tuple(data, address)。其中 data 是包含接受数据的字符串，address 是发送数据的套接字地址。

socket.sendto(string[, flag], address)：发送 UDP 数据，将数据发送到套接字，address 形式为

tuple(ipaddr, port)，指定远程地址发送，返回值是发送的字节数。

socket.close()：关闭套接字。

socket.getpeername()：返回套接字的远程地址，返回值通常是一个 tuple(ipaddr, port)。

socket.getsockname()：返回套接字自己的地址，返回值通常是一个 tuple(ipaddr, port)。

socket.setsockopt(level, optname, value)：设置给定套接字选项的值。

socket.getsockopt(level, optname[, buflen])：返回套接字选项的值。

socket.settimeout(timeout)：设置套接字操作的超时时间，timeout 是一个浮点数，单位是秒，值为 None 则表示永远不会超时。一般超时期应在刚创建套接字时设置，因为他们可能用于连接的操作，如 socket.connect()。

socket.gettimeout()：返回当前超时值，单位是秒，如果没有设置超时则返回 None。

socket.fileno()：返回套接字的文件描述。

socket.setblocking(flag)：如果 flag 为 0，则将套接字设置为非阻塞模式，否则将套接字设置为阻塞模式（默认值）。非阻塞模式下，如果调用 recv()没有发现任何数据，或 send()调用无法立即发送数据，那么将引起 socket.error 异常。

socket.makefile()：创建一个与该套接字相关的文件。

2.1.4 Socket 程序的整体一般结构

1) TCP 服务器

服务器端的 Socket 程序一般化的基础结构如下所示。

```
import socket

HOST = '192.168.1.100'
PORT = 8001

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 创建套接字
s.bind((HOST, PORT)) # 绑定套接字到本地 IP 与端口
s.listen(5)          # 开始监听链接

# 进入循环，不断接受客户端的链接请求
while True:
    conn, addr = s.accept()
    print ('Connected by ', addr)
```

```
    # 接收客户端传来的数据，并且发送给对方发送数据
    while True:
        data = conn.recv(1024)
        print (data)

        conn.send("server received your message.")

conn.close() # 传输完毕后，关闭套接字
```

2) TCP 客户端

客户端的 Socket 程序一般化的基础结构如下所示。

```
import socket
HOST = '192.168.1.100'
PORT = 8001

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 创建套接字并链接至远端地址
s.connect((HOST, PORT))

# 链接后发送数据和接收数据
while True:
    cmd = raw_input("Please input msg:")
    s.send(cmd)
    data = s.recv(1024)
    print (data)

s.close()
```

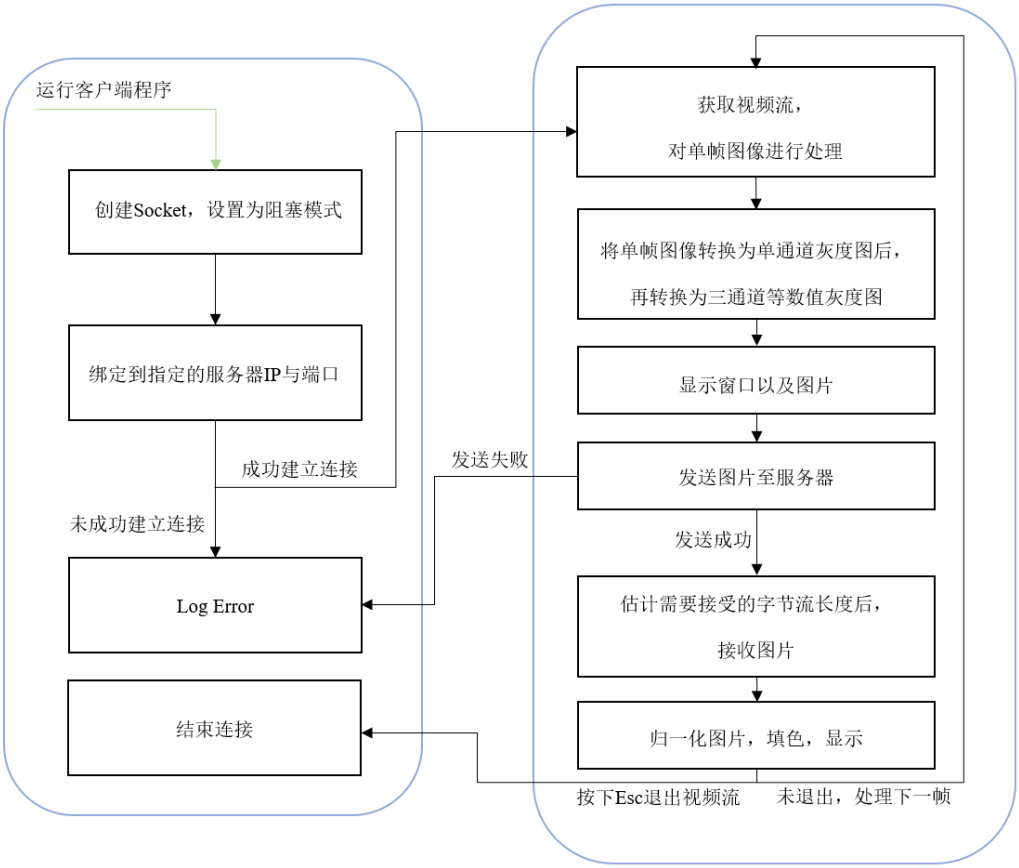
3. 程序实现

3.1 客户端 Python 程序 client.py

客户端的 Python 程序 client.py 实现了通过 Socket 与远程服务器建立 TCP 连接进而传输数据流(包括图片和视频)的功能。现将其整体框图和详细实现步骤以及部分关键代码分别展示如下。

3.1.1 程序框图

程序流程框图如下所示。



3.1.2 详细实现步骤与部分关键代码

1) 客户端初始化：

定义初始化函数，创建套接字(socket)，设定为阻塞模式，并绑定 socket 到指定的服务

器 IP 与端口。

使用日志记录可能出现的错误。

```
def InitClient(addr: str, port: int) :
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # 阻塞模式
    sock.setblocking(True)
    # sock.setsockopt(socket.SOL_SOCKET, socket.TCP_MAXSEG, 1)
    try:
        # 连接服务器
        sock.connect((addr, port))
        # Debug
        logging.info("Client's Peer Name"+str(sock.getpeername()))
        logging.info("Client's Name"+str(sock.getsockname()))
    except Exception as err:
        print(err)
        print("Can't connect to " + addr + ":" + str(port))
        sock.close()
        return None
    else:
        print("Client has connected to " + addr + ":" + str(port))
        return sock
```

2) 数据流传输：

如果成功建立连接，进行数据流传输。定义传输函数，分别对图片和视频流进行相关处理。

对于视频流，进行以下处理：将使用 cv2.VideoCapture 从摄像头中获取的视频流分解为帧；对于单帧图像，使用 cv2.cvtColor 转换为单通道灰度图后，再转换为三通道等数值灰度图(以适应神经网络输入格式)；使用 cv2.namedWindow 显示窗口，使用 cv2.imshow 显示图片。至此单帧图像的预处理完成。

之后，调用 socket_send_recv.py 中的 send_img 函数发送图片。

然后，调用 socket_send_recv.py 中的 recv_img 函数接收带有数据头的密度图；在此之前，先使用其中的 recv_data_len 函数判断需要接收的密度图的字节流的长度。

最后，使用自编的数据预处理库 data_preprocess 中的 MinMaxNormalize 函数将密度图归一化，使用 cv2.applyColorMap 填色，最后显示出来。

```python
def SendStream(addr: str, port: int,src: Union[int, str]):
    sock = InitClient(addr, port)
    if not sock:
        raise RuntimeError("Failed to Establish Socket Connect")

    # 处理视频
    capture = cv2.VideoCapture(src)
    while True:
        ret, frame = capture.read()
        if not ret:
            break
        # 读取图像，转化为三通道灰度图后发送
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frame = cv2.cvtColor(frame, cv2.COLOR_GRAY2BGR)
        if 0 != send_img(sock, frame):
            break
        cv2.namedWindow('Client Cam', 2)
        cv2.imshow('Client Cam', frame)
        # 读入带有数据头（指出密度图字节流的长度）的密度图（对付粘包问题）
        data_len = recv_data_len(sock)
        predict_density = recv_img(sock, data_len, is_density=True)
        # Debug
        print(predict_density)
        # 显示返回的密度图
        density_count = np.sum(predict_density)
        print("Current Count=" + str(density_count))
        show_density = MinMaxNormalize(predict_density, 0, 255).astype (np.uint8)
        cv2.namedWindow('Client Received Density Map', 2)
        show_density = cv2.applyColorMap(show_density, cv2.COLORMAP_JET)
        cv2.imshow('Client Received Density Map', show_density)
        if cv2.waitKey(100) == 27:
            break

    sock.close()
    if src == 0:
        print("Connection End By User")
    else:
        print("Connection End for Video has been Send Out")
    cv2.destroyAllWindows()
```

3.2 服务端 Python 程序 server.py

这里用到了 Python 的 SocketServer 网络服务器框架来简化编程，实现通过 TCP 协议和

客户端通信。最为简单的一种方式是创建 BaseRequestHandler 的继承类，并且实现 handle 方法。将自定义类实例化，当由外部 TCP 连接并接收到信息后，服务器会调用 handle 方法。

服务器在接收图片时采用了预定的数据格式，即数据的前 16 字节表示该段图片的大小，这里调用 recv_data_len 方法，该方法的实现方式在后文中有讲解。在得到图片大小后，就确定了从缓存区读取的数据大小，因此根据这一大小调用 recv_img 方法，读取图像字节流，并恢复为图像，进而可以在服务器端进行处理（处理部分代码在文中的代码段中略去）。

```
class MyTCPHandler(socketserver.BaseRequestHandler):
    # self.request 可以简单认为是 sock
    def handle(self):
        print("Receive an TCP Request From " + str(self.client_address))

        self.request.setblocking(True)
        # Debug
        print("Recv Buffer Size = ", self.request.getsockopt(socket.SOL_SOCKET,
socket.SO_RCVBUF))

        time.sleep(1)
        print("Get An Reuquest From Client")

        # 计数代码，用来计算帧率和两帧之间时间间隔
        last_time = time.time()
        loop_index = time_sum = time_gap = 0
        while True:
            # 收取图像，先收取 data_len，就是图像字节流的长度，再读取图像
            data_len = recv_data_len(self.request)
            if not data_len:
                break
            img = recv_img(self.request, data_len)
            if not isinstance(img, np.ndarray):
                break

            print("Receive an Image")
            print("Size of Img:" + str(img.shape))

            # 计算帧率
            time_gap = time.time() - last_time
            print("Time Beteen Two Img: " + str(time_gap))
            last_time = time.time()
            time_sum += time_gap
            loop_index = (loop_index + 1) % kLoop
```

```
            if not loop_index:
                print("fps: " + str(kLoop / time_sum))
                time_sum = 0
# 省略部分代码
            # Send Back density to client
            if 0 != send_density(self.request, predict_densities[0]):
                break
        print("Connection End At Clinet!")
```

之后的 StartServer 函数功能比较明显，是对以上自定义类的实例化，并且规定了多线程 TCP 相应，即对于每一组 TCP 连接，服务器端都创建一个对应的处理线程。

3.3 socket_send_recv.py 文件

该文件中定义了服务器端和客户端进行数据交流时应用层与运输层之间的数据沟通。根据之前所述的数据传递规则，将每一张图片的数据信息划分为 16 字节的长度表征部分和其余的字节流部分。

recv_data_len 方法实现了对长度表征部分的 16 字节数据的读取，因为预设了数据长度，所以可以直接在缓冲区以 16 字节长度读取。

recv_img 方法定义了读取图片字节流部分。对于读取模式，根据 kRecvAll 参数，有两种供选择：1.发送-确认模式，该种模式下发送端每次发送 kMTU 字节（实验中表示 1024 字节），这种模式下需要接收端反馈有接收信号，以激励发送端的下一次传输。2.流水线模式，该种模式下发送端一次性发出所有数据，但接收端仍然以 1024 字节为单位逐次从缓冲区读取。

send_img 方法用来发送图像，根据预定规则，每张图片的发送都需要以长度表征部分作为开始。之后与 recv_img 方法对应，有两种模式供选择：1.发送-确认模式，这种模式下在一张图片未全部发送时，每次发送 1024 字节，在接收到反馈信息后才开始下一次发送，直到一张图片发送完成。2.直接发送模式，直接将图片字节流完整放入运输层。

```
def recv_data_len(sock):
    recv = sock.recv(kHeadLen)
    if not recv:
        return None
    return int(recv.decode())
```

```
def recv_img(sock, data_len, is_density: bool=False):
    # 如果是密度图，还要收取密度图的长宽
    if is_density:
        density_h = int(sock.recv(kHeadLen).decode())
        density_w = int(sock.recv(kHeadLen).decode())
    string_data = b''
    # 发送-确认模式，发送端每发送 1024 字节就停止等待，直到对方发送确认信号
    if not kRecvAll:
        index = 0
        while index + kMTU <= data_len:
            string_data = string_data + sock.recv(kMTU)
            index += kMTU
            sock.send(str(index).ljust(kHeadLen).encode())
        else:
            string_data = string_data + sock.recv(data_len - index)
    else:
        # 流水线，发送端一次发出所有数据
        while len(string_data) < data_len:
            string_data = string_data + sock.recv(kMTU)

    if not string_data:
        return None

    #debug
    if data_len != len(string_data):
        print("Error May Occurred at recv_img Function!")
        print("Receive An Img with String Length of: " + str(len(string_data)))
        print("However, the DataLen is: " + str(data_len))
        return None

    if is_density:
        # 恢复为密度图像
        data = np.frombuffer(string_data, dtype='f4').reshape(density_h,
density_w)
        return data
    else:
        # 恢复为图像（字节流反序列化+opencv 解码）
        data = np.frombuffer(string_data, dtype=np.uint8).reshape(data_len, 1)
        return cv2.imdecode(data, 2|4)



# 发送图像
```

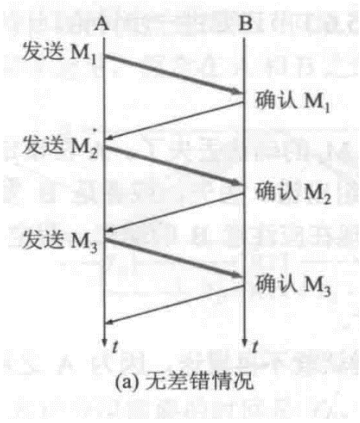
```python
def send_img(sock, img: np.ndarray, is_density: bool=False):
    if is_density:
        string_data = img.tostring()
    else:
        encoded_img = cv2.imencode('.jpg', img, encode_param)[1]
        string_data = encoded_img.tostring()

    # Send Data Length
    data_len = len(string_data)

    sock.send(str(data_len).ljust(kHeadLen).encode())
    # Send the Shape of density map
    if is_density:
        sock.send(str(img.shape[0]).ljust(kHeadLen).encode())
        sock.send(str(img.shape[1]).ljust(kHeadLen).encode())

    #sock.send(string_data)
    index = 0

    if not kSendAll:
        while index + kMTU <= data_len:
            sock.send(string_data[index:index + kMTU])
            index += kMTU
            if not sock.recv(kHeadLen):
                print("Can't Receive Confirm pack when index=" + str(index))
                return None
        else:
            sock.send(string_data[index:data_len])
    else:
        # 直接发送模式
        send_info = sock.send(string_data)
        print("Send Info =", send_info)

    return 0
```

4. 问题排查与解决方法

4.1 出现的问题

如上一节所述，当按照以下参数设置时，程序按照停止等待的方法发送数据包，如下图所示：

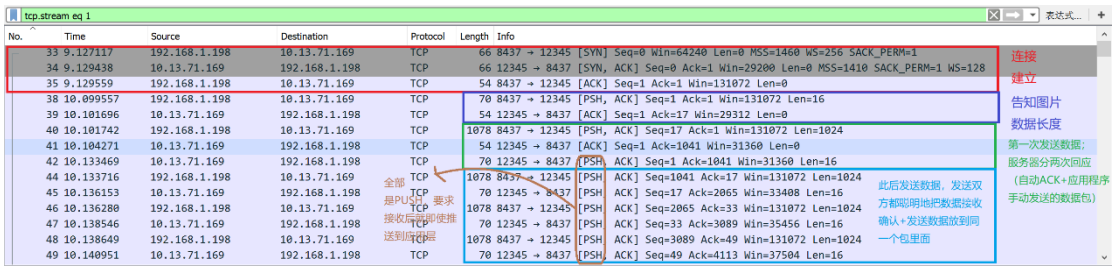
```
kSendAll = False
kRecvAll = False
```



(a) 无差错情况

客户端 A 先发送 1024 字节的数据包给服务器 B，停止等待；服务器 B 收到后确认，发送 16 字节的确认数据包给 A；A 收到后开始下一次发送。

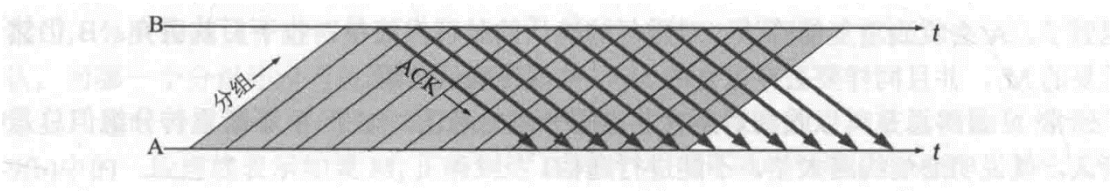
这里可能有一个问题，就是 TCP 接收端在收到数据包后理论上会自动发送一个数据长为 0 的数据包作为确认。那接收端的应用层又手动发送一个确认数据包是否会造成时间浪费呢？答案是不会的，接收端有延迟确认功能，收到数据包后等待一段时间，直到达到等待时间阈值，或者有数据可以捎带，才一起发出，可以用 WireShark 抓包证明，如下所示。



我们知道，这种传输模式效率是十分低下的，其发送时间为：

$$分组数\times(T_D+RTT+T_A)。$$

如果使用如下所示的流水线传输：

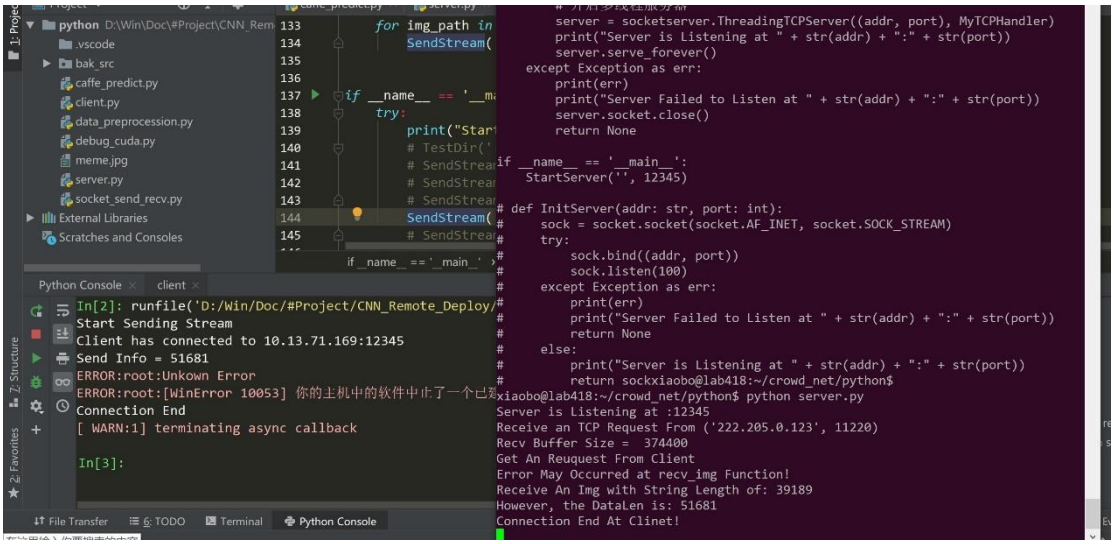


则发送时间为：

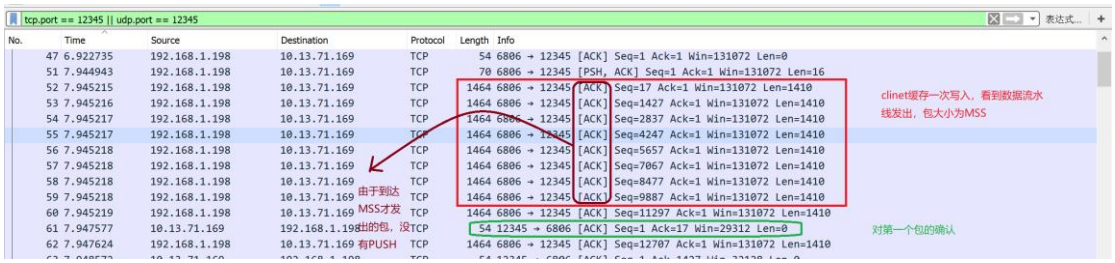
$$分组数\times(T_D)，$$

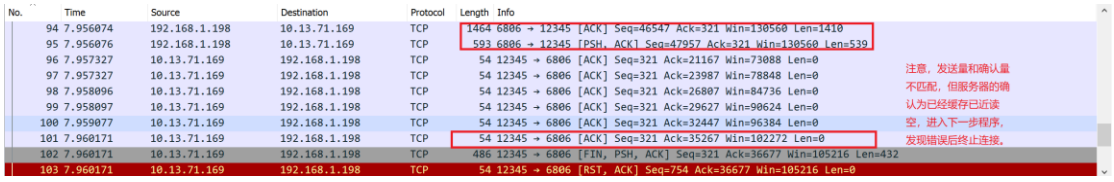
流水下所需时间为停止等待的$\frac{T_D+RTT+T_A}{T_D}$倍。

然而，在尝试使用流水线传输的时候，即一次把所有数据写入缓存，自动发送，却出现了发送数据长度和读取数据长度不等的情况，如下所示。



注意右侧程序提示，收到的数据头，指出数据长度为 51681，然而收到的数据包长度只有 39189。使用 WireShark 进一步分析，结果如下所示。





该问题在初始时迟迟不能解决，故采用了上述停止等待方法传输数据。

4.2 问题排查与解决

接下来开始排查故障原因。以下依序给出几种针对问题原因的假设，并给出对应的分析，最后给出问题的最终解决方法。

4.2.1 假设一与分析

发送方或接收方的缓存容量不足。尤其可能的是发送方的缓存不足，不能一次写入发送数据，多出来的部分被自动抛弃。

然而，这种情况一般不大可能出现，原因是已经手动在发送、接受端设置了阻塞模式，即：

```
sock.setblocking(True)
```

所谓阻塞模式，就是倘若写入缓存的数据量小于预定发送，接受的数据量，程序等待在当地，直到缓存又有空余，可以把剩余数据写入/读取。

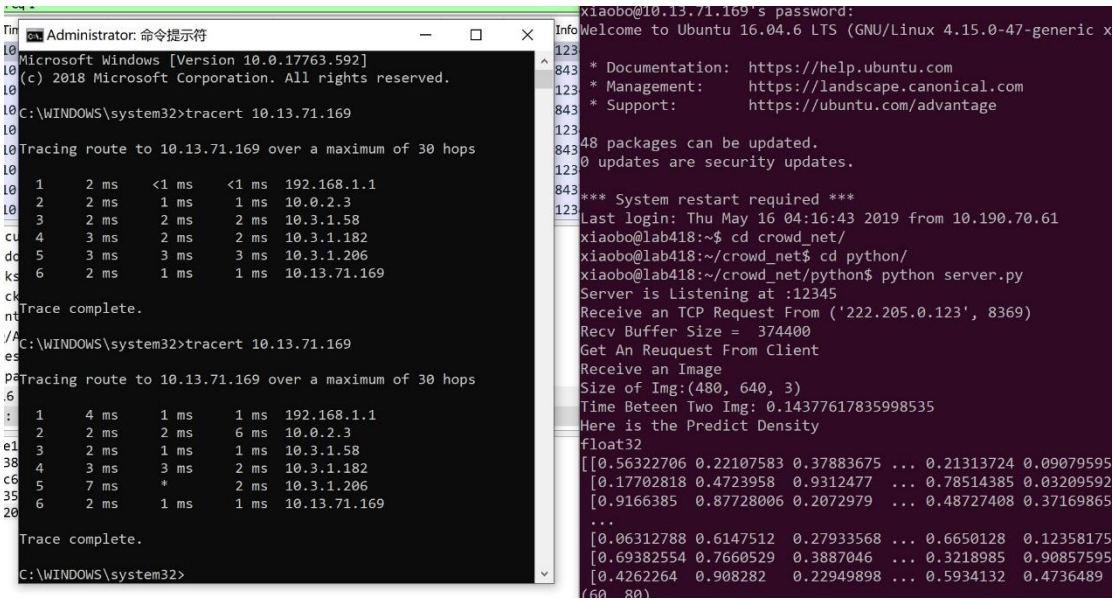
此外，我们还可以利用 getSocketOpt()获取收发两方的缓存大小(上图中已经打印出来)，显示二者缓存均比数据大小大一个数量级，所以应该不是缓存的问题。

4.2.2 假设二与分析

鉴于相当一部分数据没有发送确认接受包，有可能传输过程中发生丢包，尤其是可能的是，接受时丢掉了顺序在后的包(如果丢掉了顺序在前的包，按照快重传规定，发送端会即时补发丢掉的包)。

按照之前 WireShark 结果，所有数据包都被发出，但是只有部分数据包得到确认。那是不是发生丢包了呢？在使用 TCP 协议之前，我用过 UDP 协议传输数据，但有些时候，几乎每帧图像都遇到了丢包问题。

这是可以理解的，注意虽然服务器和主机都在内网，但服务器显示 TCP 连接来自一个外网 IP，如下所示。



这说明发生了 NAT 映射。而 UDP 在 NAT 情况下存在丢包率较高的问题。

此外，也有可能发生在网卡或者驱动缓存不足丢包，防火墙丢包，系统负载过高等等，鉴于前不久服务器的磁盘曾被写坏过一次，尚未完全恢复系统数据，这些可能都不能排除。

至于为什么 TCP 丢包后发送端没有自动超时重传，是因为自编接受程序在读空缓存以后就会判断数据是否长度是否符合数据头中长度，如果不符，就直接判错，中止连接。这也给了我们验证是否存在丢包的方法：可以在接受完成后检测数据长度是否正确，如果接受长度不足，就执行 time.sleep(10),发送端由于没有反馈信号，就会超时重传：

```
if data_len != len(string_data):
```

```
    print("Error May Occurred at recv_img Function!")
    print("Receive An Img with String Length of: " + str(len(string_data)))
    print("However, the DataLen is: " + str(data_len))

    print("Try to sleep until peer resent finished")
    print("Current Index = ", index)
    # 睡眠，强迫超时重传
    time.sleep(10)
    while index + kMTU <= data_len:
        string_data = string_data + sock.recv(kMTU)
        index += kMTU
    else:
        string_data = string_data + sock.recv(data_len - index)
    if data_len != len(string_data):
        print("Error Still!!!")
        print("Receive An Img with String Length of: " + str(len(string_data)))
        print("However, the DataLen is: " + str(data_len))
        return None
```

然而，实验的结果并不符合这一假设：没有超时重传发生，睡眠后所有确认包都正确返回，程序收到的数据长度略有增长，但依然小于发送数据长度。这就需要做出其他假设。

4.2.3 假设三、对应分析以及解决方法

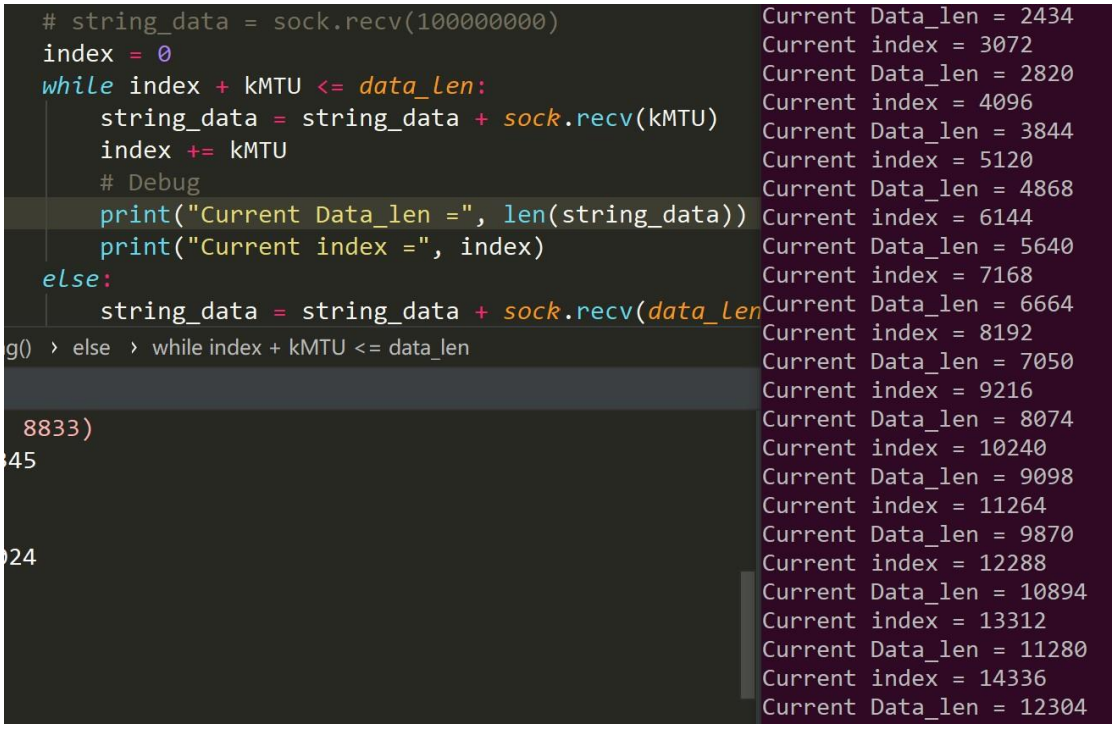
假设 2 中已经证明，如果没有接收端中止，所有数据都正常到达接收缓存并发送确认。那问题只有可能出在应用层对缓存的读取上面。

按照原始程序，应用层程序 1024 字节一次从缓存中读取数据，用变量 index 更新读入数据的长度，到达指定数据长后停止：

```
# 使用 index 跟踪读入了多少数据，确保读入数据量刚好为发送数据量
# 避免出现粘包的时候，读入之后数据包的数据
while index + kMTU <= data_len:
    string_data = string_data + sock.recv(kMTU)
    index += kMTU
```

由于之前已经设置了 recv()为阻塞模式，理论上如果缓存里小于 1024 字节，程序会停下来等待，直到有足够数据供以读取。现在尝试验证：

```
while index + kMTU <= data_len:
    string_data = string_data + sock.recv(kMTU)
    index += kMTU
    # Debug
    # 打印出程序以为的读入数据量 index 和实际累计数据长度
    print("Current Data_len =", len(string_data))
    print("Current index =", index)
else:
    string_data = string_data + sock.recv(data_len - index)
```



结果发现，认为的读入数据量和实际读入数据量并不一致，总是要小一些。略加推断，可以判是每次 recv 的时候，如果缓存内数据没有 1024 字节，就全部读空，也不报错或阻塞，继续执行程序。鉴于数据包在网络中传输总是有较大的时间间隔，所以写入速度更不上服务器读缓存速度，自然出错。

那为什么之前设置了 python socket 模块运行在阻塞模式下，依旧没有进行阻塞读取呢？可能是因为编程时使用了高级抽象方法来开启服务器：

```
# 开启多线程服务器，一次连接一个线程
server = socketserver.ThreadingTCPServer((addr, port), MyTCPHandler)
```

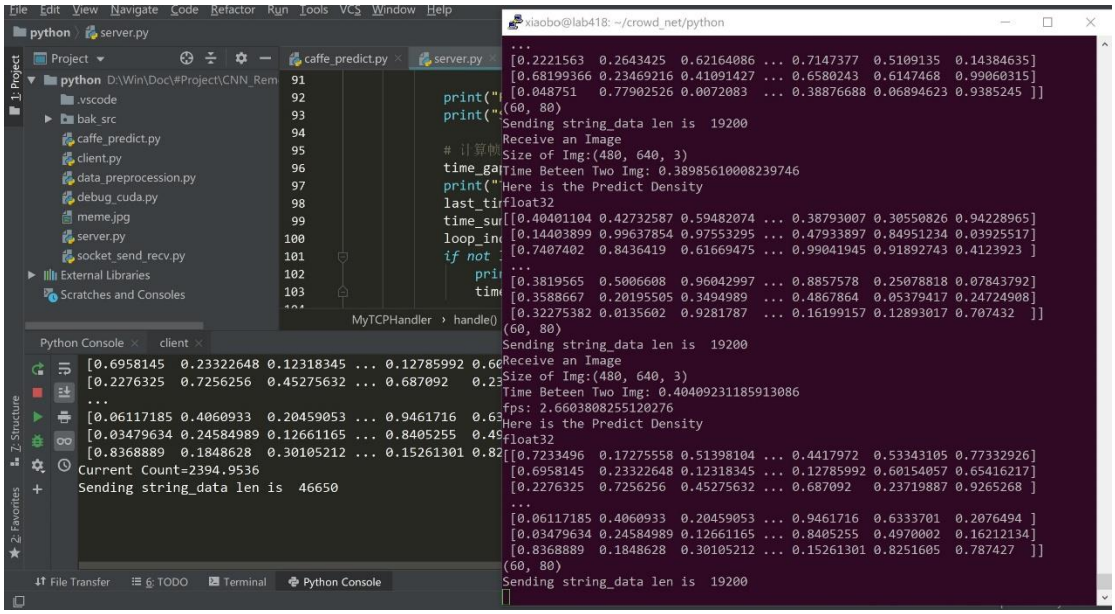
这样做可以让服务器同时服务多个客户端，但可能这种高度抽象的方法，并不像底层方法那么可靠，或者有不为人知的使用注意事项，如果没有通读文档，很难发现出了错误。

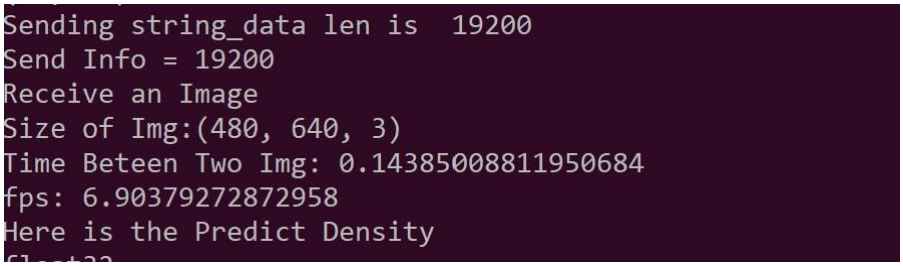
最后，我们可以把接受程序简单的修改为：

```
# 流水线，发送端一次发出所有数据
# 由于实际双方单进程通信，不会有粘包出现，所以不用担心最后一次读取长度大于剩余数据长度
while len(string_data) < data_len:
    string_data = string_data + sock.recv(kMTU)
```

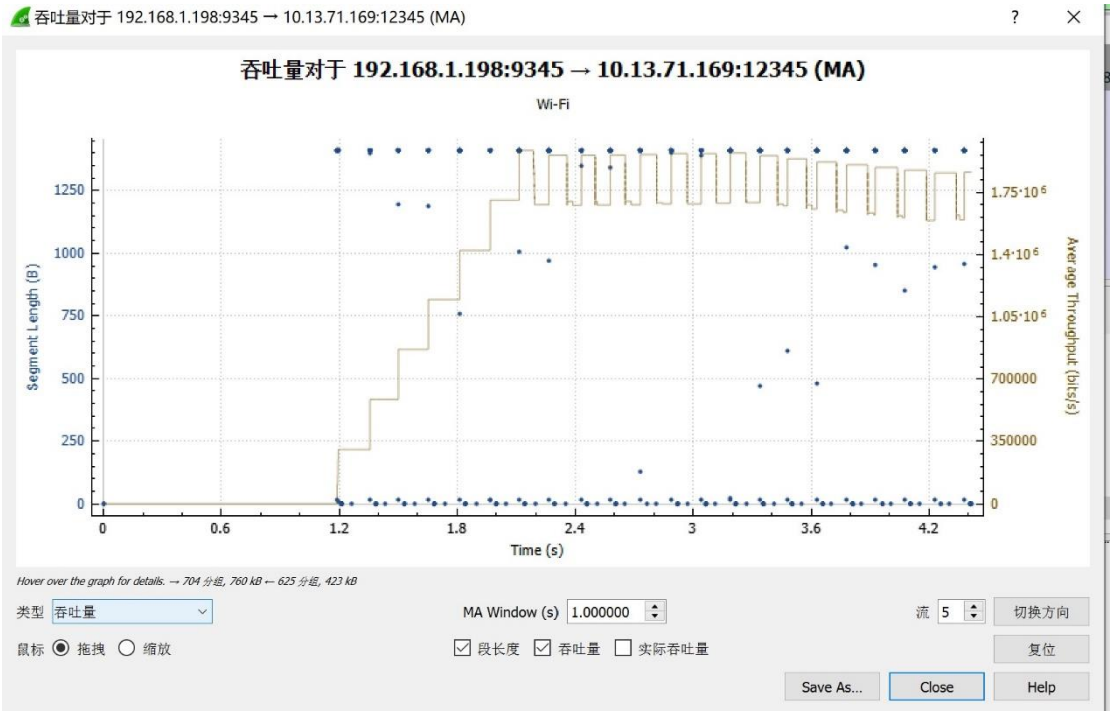
如此即可进行流水线传输。

以下对比停止等待和流水线的帧率(fps)：





发现帧率从 2.6 提高到 7，提高两到三倍。吞吐量有显著改善：



此外，利用之前公式：

$$\frac{\mathrm{fps}_{\text{流水}}}{\mathrm{fps}_{\text{停止等待}}}=\frac{\mathrm{T_D+RTT+T_A}}{\mathrm{T_D}},$$

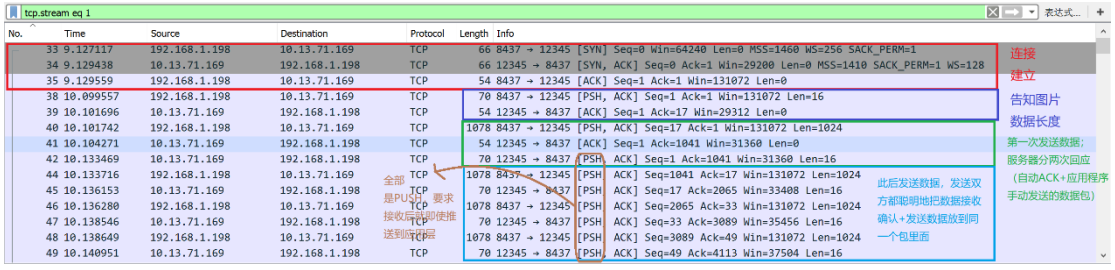
可以估计出发送时间大概和 RTT 时间差不多(这么看来校园网内网速度还挺不错)。

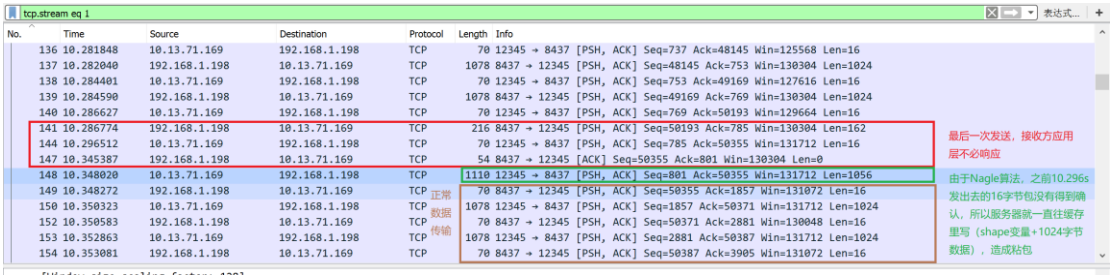
5. 结果展示

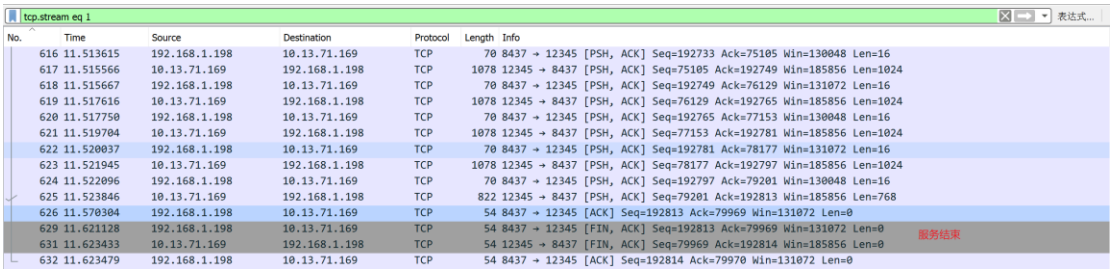
5.1 项目成果

通过以上分析和修改，小组成功实现了两个进程之间的数据传递，能够由笔记本电脑实现图像采集，将图像字节流传递给远程服务器，服务器接到 TCP 连接请求后创建新的线程对该次连接响应，利用训练好的网络对图像进行人群密度识别，再将估计结果返回笔记本电脑。若仍然有图像需要传递，可以保持 TCP 连接继续下一幅图像的识别，若长时间无图像传递，则断开 TCP 连接。下图为一次实验中人口密度识别的结果图：

5.2 利用 WireShark 抓包分析

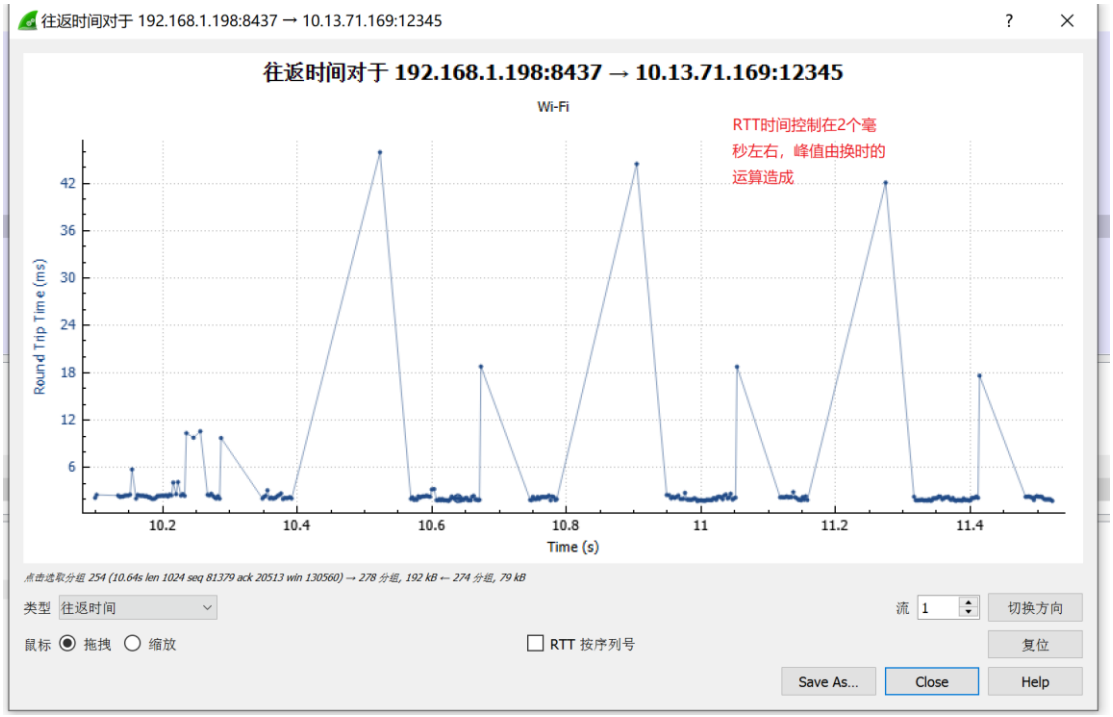






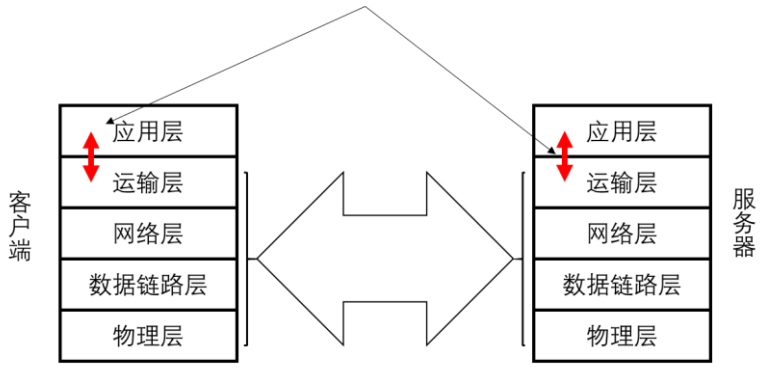
利用 WireShark 抓包软件可以抓取一定时间内系统收发的报文段，并解析首部和数据部分。上图为某次抓包的结果，由上到下按照时间顺序。可以看到，这一段时间内主要进行的是 IP 地址为 10.13.71.169 和 192.168.1.198 的主机之间的通信。其中，绝大部分都有 PSH 标志，表明为推送，其大小为 1024，也符合我们编写程序所规定的报文大小；同时具有 ACK 标志，表明在密切交流的情况下，应答常常和正常的报文传递结合在一起以提高效率。同时，上图中还可以看出 TCP 协议建立连接和释放连接时的三次握手。建立连接的过程也可以从下图中看出：

此外，还可以分析出两个主机之间交流的 RTT：



6. 总结与心得

网络协议的设计十分巧妙，需要综合考虑到各种情况。本次实验我们通过调用 socket 模块，实际上是站在应用层的角度思考问题。运输层及以下三层——网络层、数据链路层和物理层已经封装完成，并且我们采用了 TCP 协议，因此可以认为在两进程之间的线路（客户端的图片获取和服务器图片处理）能够建立有效、准确、可靠的连接，采用的构思模型如下图所示：



我们将主要目光聚焦在进程与运输层（运输线路）之间的数据交流。尽管两进程之间的线路可以认为是完全理想的，为了保证两个进程能够正确解析对方传递的数据，我们在应用层设计了如数据格式、数据传递模式等协议，并且取得了不错的效果。

相对于其他四层，应用层可以说是最为简单的一部分，将底层抽象为 API 接口供上层调用，仅仅需要从软件层面考虑，不会出现诸如线路阻塞、接口匹配等一系列复杂多变的实际问题，而且我们的应用也局限于十分简单的方向，但是我们仍然感到十分困难，难以周全地考虑到各种意外情况。由此可见，TCP/IP 体系结构的设计复杂程度是极高的。向所有为计算机网络的发展做出贡献的前辈们致敬！

此外，之前棘手的服务器、客户端信息传递中遇到的困难，在学习了网络基础课程之后迎刃而解，SRTP 也顺利通过了验收。感谢王勇老师一学期的辛苦教学！