基于神经网络的拥塞控制算法实验报告

刘帅 2020212267 2020219111班

1、实验目标

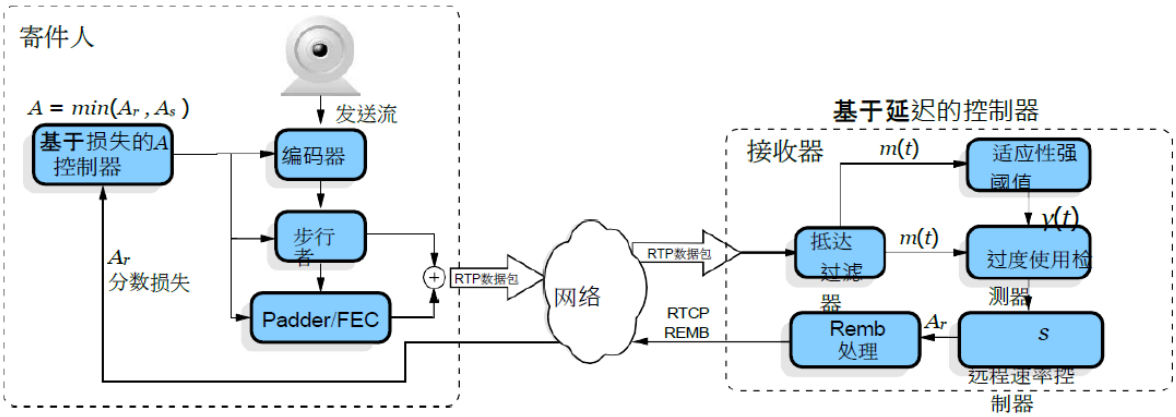
本实验基于仿真器模拟WebRTC传输，利用神经网络方法对实时音视频通信（RTC）进行码率预测。利用GCC所提供的标签进行参数训练，从而获得效果较好的模型。并通过修改训练逻辑和功能，使得模型训练pipeline更加完善，并能够具有较好的实用性和封装性，从而扩展模型训练的适用域。

2、实验背景

2.1.WebRTC和GCC介绍

WebRTC是web端的实时通信解决方案，它可以做到在不借助外部插件的情况下，在浏览器中实现点对点的实时通信。针对视频通信类任务，通常需要低延迟和高带宽的网络需求。而传统的TCP协议由于可靠性和按部就班的交付机制，因此，WebRTC任务的传输层通常基于UDP协议进行。为了保证视频图像质量、连接延迟、无缝通信等因素，产生了GCC算法，其核心思路是利用延迟的梯度来对传输过程中的拥堵情况进行判断，基于卡曼滤波估计单程延迟变化，并将估计值与与自适应阈值相比较，从而动态控制发送速率。

2.2 GCC网络架构：



发送方采用UDP套接字发送RTP数据报，并从接收方接收RTCP反馈报告。该算法共分为两个部分：1、基于延迟的控制器，放置在接收端，计算目标发送的延迟。2、基于丢包的控制器，放置在发送端，计算丢包数量。

2.3基于延迟的控制器

基于延迟的控制器主要包含的到达时间滤波器、过度使用检测器和远程速率控制器三个部分。下面简述三个控制器的功能：

①到达时间滤波器：产生单程延迟梯度的估计值，为此采用卡曼滤波算法，根据测量的单程梯度dm(ti)进行估计

$$d_m(t_i) = (t_i - t_{i-1}) - (T_i - T_{i-1})$$

②过度使用检测器：没收到视频帧，该检测器都会根据阈值对信号进行检测，并对适用情况进行判断

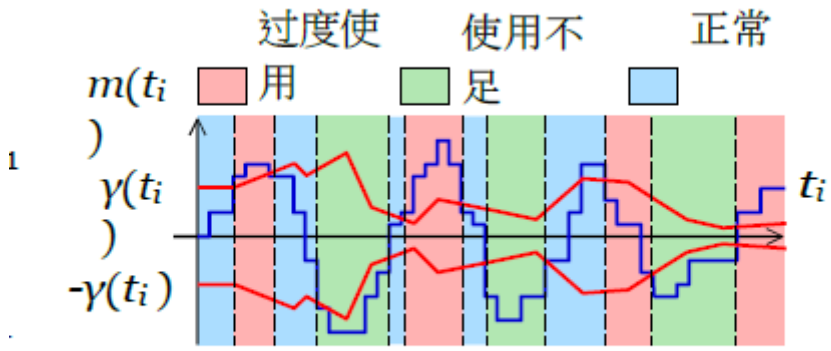
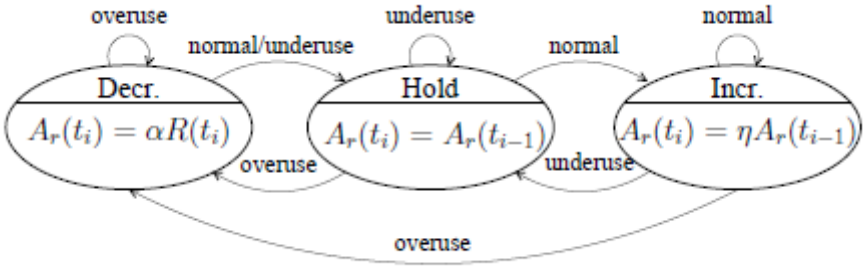


图3：过度使用检测器的信号

③远程速率控制器：

利用图示状态机，实现路径上缓冲区排队延迟最小化。



2.4基于损失的控制器

WebRTC通过以下公式对发送码率进行估计，其中，As(tk)表示带宽的估计值,fl(tk)表示tk时刻的丢包率

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0.5 f_l(t_k)) & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1})) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases} \quad \text{——} \quad (1)$$

3、仿真和训练

a.流程介绍

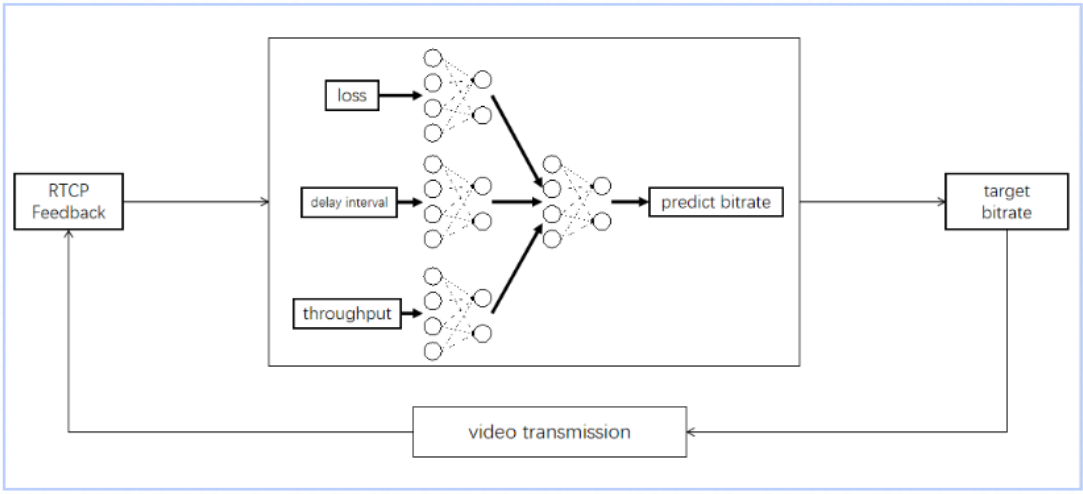
在实验场景中，网络对于loss、delayinterval和throughput三个种类的数据进行预测，输入神经网络预测码率predict_bitrate,仿真器根据预测码率传输并测得下一轮网络数据。具体训练流程在第五部分进行详细介绍。

4、神经网络

4.1介绍

```
ACTOR_VECTOR = np.arange(0.0, 2.02, 0.1)
ACTOR_SLOTS = len(ACTOR_VECTOR)
```

在仿真器中，设置了码率范围为（0，2.02）步长为0.1，即共有21种输出种类。在数据输出中，最后一层利用softmax进行归一化，表示预测对应码率的概率，利用onehot编码对输出标签进行预测。同时，观察神经网络的输入，其两个输入网络指标为input_loss_and_interval和input_throughput，其中，第一个指标前四项表示loss，大小为(n,4)后四项表示delay_interval,大小为(k,4)，第二个输入指标为input_throughput，其大小为(m,4)。如图



因此，我们需要利用多层感知机分别对loss、delay、throughput三个维度进行训练。

4.2具体实现

4.2.1 导入tensorflow及对应的搭建网络框架。

```
import tflearn
import tensorflow as tf
from functools import partial
```

4.2.2 网络定义部分

```
def NN(inputs, inputs2):
    loss = inputs[:, :4]
    delay_interval = inputs[:, 4:]
    throughput = x_ = tflearn.reshape(inputs2, [-1, 10])
    #网络的三个输入，只用考虑其dim=1方向的数据大小，输入大小分别为4,4,10。
    loss=tf.layers.dense(loss,256,partial(tf.nn.leaky_relu, alpha=0.01))
    delay_interval = tf.layers.dense(delay_interval, 256,
activation=partial(tf.nn.leaky_relu, alpha=0.01))
    throughput=tf.layers.dense(throughput,256,partial(tf.nn.leaky_relu,
alpha=0.01))
    #利用tf.layers.dense()搭建全连接层，使第一个隐藏层输出维度统一为256，激活函数选用
leaky_relu代替relu，主要是为了防止训练过程中出现梯度消失的问题。
    loss=tf.layers.dense(loss,512,partial(tf.nn.leaky_relu, alpha=0.01))
    delayinterval=tf.layers.dense(delay_interval,512,partial(tf.nn.leaky_relu,
alpha=0.01))
    throughput=tf.layers.dense(throughput,512,partial(tf.nn.leaky_relu,
alpha=0.01))
    #继续数据升维，使第二个隐藏层输出维度为512
    lay3=tf.concat([loss,delayinterval,throughput],1)
    #在dim=1的方向对网络进行拼接
    dplay3=tf.layers.dropout(lay3,0.5)
    #引入dropout层，对输出神经元进行随机丢失，从而避免数据过拟合。
    lay4=tf.layers.dense(dplay3,64,partial(tf.nn.leaky_relu, alpha=0.01))
```

```
    output=tf.layers.dense(lay4,21,partial(tf.nn.leaky_relu, alpha=0.01))
    #数据降维，使其输出维度变为21。
    output=tf.layers.flatten(output)
    #数据展平
    return output
```

5、训练流程及改进

该部分主要为gcc_train.py代码的讲解

5.1 导入框架

```
from network_simulator import receiver, sender, router, packet #仿真器导入
from network_simulator.frame import Frame
from SendSideCongestionControl import SendSideCongestionController
import _thread
import os
import numpy as np
import warnings
import network
import tensorflow as tf
import matplotlib.pyplot as plt #后续数据可视化
import pylab as pl
warnings.filterwarnings("ignore")
```

5.2 命令行建立

由于训练流程中涉及到模型读取、学习率衰减策略超参数设置、batchsize大小等调整问题，利用命令行可以更好地进行训练，故定义以下超参数。

```
import argparse#命令行
parser=argparse.ArgumentParser()
parser.add_argument('--load_model',action="store_true")#命令行中若输入此命令则进行模型读取，若不输入，则默认为从头训练
parser.add_argument('--UPDATE_SIZE',type=int,default=16)#batchsize大小更新
parser.add_argument('--poly_lr',type=float,default=0.001)#在启停逻辑中，使用poly多项式衰减，其中的超参数需要进行调整。
parser.add_argument('--poly_endlr',type=float,default=0.0001)#poly衰减的参数
parser.add_argument('--use_cycle',action="store_true")#poly衰减的参数
parser.add_argument('--exp_lr',type=float,default=0.002)#指数衰减的参数
args=parser.parse_args()
```

5.3 仿真器数据初始化

此处针对仿真器的部分参数进行初始化，例如缓存大小、窗口大小，读取周期，停止时间等。下方args.UPDATE_SIZE = 16则是命令行中所定义的batchsize大小，可进行调整。码率区间则为上文所介绍的输出标签。

```
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
os.environ["TF_CPP_MIN_LOG_LEVEL"] = '3'
loss_window = 10
router_buffer = 20
start_bitrate_bps = 300 * 1000
stop_time = 60 * 1000 * 200
read_interval = 1000
```

```
default_frame_size = 30 * 1000
S_LEN = 4  # take how many frames in the past
start_arrival_time = 0

# 一次训练中的 batch 宽度
args.UPDATE_SIZE = 16
# 码率区间
ACTOR_VECTOR = np.arange(0.0, 2.02, 0.1)
ACTOR_SLOTS = len(ACTOR_VECTOR)
```

5.4 获取输出结果索引

```
def get_predict_index(logits):  # 根据神经网络的输出得到最终预测结果
    ret = tf.cast(tf.argmax(logits, 1), tf.int32)
    return ret[0]
```

5.5 损失函数计算及精确度计算

此处的改动有二：

①增加flag=0的参数，主要识别目前训练轮次及loss变化，从而决定学习率衰减的策略。

②引入poly多项式衰减，运行逻辑为flag=1，主要用于神经网络训练后期，在epoch较大时使用。考虑到exponential_decay收敛速度较快，因此可能存在过拟合问题，并且在实际训练中会出现loss回升的情况。采用polynomial_decay可有效减少loss上升的情况。

```
def get_evaluate_indicators(global_steps, logits, y_, batch_size,flag=0):
    distance_gradient = tf.zeros(batch_size)
    for i in range(0, batch_size):
        pre_index = tf.cast(tf.argmax(logits[i]), tf.int32)
        dis = pre_index - y_[i]
        one_hot = tf.one_hot(i, batch_size, dtype=np.float32)
        distance_gradient = distance_gradient + tf.cond(tf.greater(pre_index,
y_[i]),
                                                        lambda: tf.cast(
                                                            tf.add(8,
tf.pow(dis, 2)), tf.float32),
                                                        lambda:
tf.cast(tf.pow(dis, 2), tf.float32)) * one_hot
    distance = tf.cast(tf.stop_gradient(distance_gradient), tf.float32)#distance
为一个加权参数，预测结果与目标结果越接近此值越小，越不接近此值越大
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels=y_, logits=logits)#二分类交叉熵
    weight_loss = tf.reduce_mean(distance * loss)
    if(flag==1):
        lr = tf.train.polynomial_decay(
             learning_rate=args.poly_lr, global_step=global_steps,
decay_steps=50,
            end_learning_rate=args.poly_endlr, power=0.5, cycle=args.use_cycle
)
    elif(flag==0):
        lr=tf.train.exponential_decay(args.exp_lr, global_steps, 3, 0.6,
staircase=True)

    train_op = tf.train.AdamOptimizer(learning_rate=lr).minimize(weight_loss)
```

```
    acc_one = 1 - tf.abs(tf.cast(tf.argmax(logits, 1),
                         tf.float32) - tf.cast(y_, tf.float32)) / ACTOR_SLOTS
    acc = tf.reduce_mean(acc_one)
    return weight_loss, train_op, acc, distance
```

5.6 列表建立

建立如下两列表，对于后续训练的loss和accuracy进行记录

```
losslist=[]
acclist=[]
```

5.7 训练执行

每执行一轮，将loss与准确率存入列表。

```
def runable(x_train1, x_train2, y_train, train_op, loss, acc, logits, x1, x2,
y_, distance, global_steps, batch_size, sess,
epoch,losslist=losslist,acclist=acclist,):  # 将数据喂入神经网络
    # 训练和测试数据，可将n_epoch设置更大一些
    train_loss, train_acc,n_batch = 0, 0, 0
    _, err, ac, dist = sess.run([train_op, loss, acc, distance],
                                feed_dict={x1: x_train1, x2: x_train2, y_:
y_train, global_steps: epoch})

    train_loss += err
    train_acc += ac
    n_batch += 1
    losslist.append(train_loss)
    acclist.append(train_acc)
    print("train loss: %f" % (train_loss / n_batch))
    print("train acc: %f" % (train_acc / n_batch))
```

5.8 主函数部分

5.8.1初始化部分

该部分主要执行仿真器初始化过程

```
if __name__ == '__main__':
    print("start")
    # ------------     仿真器初始化     ------------
    packet.Packet.set_max_packet_size(12000)  # bit
    Frame.set_fix_frame_size(False)
    Frame.set_default_frame_size(default_frame_size)  # bit
    sender = sender.Sender(start_bitrate_bps)
    receiver = receiver.Receiver(buffer_size=loss_window)
    router = router.Router(1800000, sender=sender,
                           receiver=receiver, buffer_size=router_buffer)
    sender.set_receiver(router)
    router.base_dir = './trace_data2/temp/mats1/'
    router.mat = './trace_data2/after760h.mat'
    router.read_interval = read_interval          # 限制带宽
    router.stop_time = stop_time    # 设置仿真器传输时间
```

```
    router.set_fix_bitrate(False)
    congestion_controller = SendSideCongestionController()
    congestion_controller.SetStartBitrate(start_bitrate_bps)
    target_send_rate = start_bitrate_bps
    rate, lbrates, dbrates, delay, loss1, delay_diff, bandwidth = [], [], [],
[], [], [], []
    ts_delta, t_delta, trendline, mt, threashold = [], [], [], [], []
    send_time_last_state = 0
    frame_time_windows = []
    feedbackPacket = router.start(target_send_rate)
    # ----------------------------------------

    simulate_round = 0  # 仿真器模拟的轮数
    epoch = 0  # 神经网络训练的轮数
    predict_index = 5  # 初始预测码率在ACTOR_VECTOR的下标

    # loss & delay interval, 更新神经网络第一项输入
    obs_loss_and_interval_batch = np.zeros([args.UPDATE_SIZE, 8])
    # throughput, 更新神经网络第二项输入
    obs_throughput_batch = np.zeros([args.UPDATE_SIZE, 10])
    # 更新神经网络的标签
    gcc_label_batch = np.zeros([args.UPDATE_SIZE])

    # 仿真器一轮模拟的 loss 和 delay interval 结果
    # 一个 obs_loss_and_interval_batch 包含 UPDATE_SIZE 行 input_loss_and_interval
    input_loss_and_interval = np.zeros([1, 8])
    # 仿真器一轮模拟的 throughput 结果
    # 一个 obs_throughput_batch 包含 UPDATE_SIZE 行 input_throughput
    input_throughput = np.zeros([1, 10])
```

5.8.2 对模型存储对象进行初始化，并且定义no_optim代表记录优化器不再继续更新的轮数，定义train_epoch_best_loss表示目前收到的最好loss（最小的loss），首先将其值设置为一个比较大的值，从而保证不会记录到错误的loss，此时的模型进行存储。

```
no_optim = 0
train_epoch_best_loss = 1000000
# 模型saver初始化
saver = tf.train.Saver()
flag=0#标注是否需要缩小学习率
```

5.8.3 模型读取

若对于网络进行训练，则命令行中输入load_model，进行模型读取，将网络权重读入。

```
with tf.Session() as sess:
    if(args.load_model==True):#模型读取
        new_saver = tf.train.import_meta_graph("./mymodel/bestmodel.meta")
        new_saver.restore(sess, "./mymodel/bestmodel")
        print('finish loading model!')
        graph = tf.get_default_graph()
        x_ = graph.get_tensor_by_name("x_ckpt:0")
        y_ = graph.get_tensor_by_name("y_:0")
```

5.8.4 原始数据读入

```
# input_loss_and_interval 以及 obs_batch 置入 x1
x_loss_and_interval = tf.placeholder(
    tf.float32, shape=[None, 8], name='x_ckpt')
# input_throughput以及 obs_throughput_batch 置入 x2
x_throughput = tf.placeholder(
    tf.float32, shape=[None, 10], name='x_ckpt')
# 仿真器内部gcc产生的标签置入y_gcc_label
y_gcc_label = tf.placeholder(
    tf.int32, shape=[None, ], name='y_')  # 标签喂入y_
# 设置神经网络的输出
logits = network.NN(x_loss_and_interval, x_throughput)
# 代表全局步数，比如在多少步该进行什么操作，现在神经网络训练到多少轮等等，类似于一个钟表。
global_steps = tf.placeholder(tf.int32, shape=[], name='global_steps')
loss, train_op, acc, distance =
get_evaluate_indicators(global_steps=global_steps, logits=logits,
y_=y_gcc_label,batch_size=len(gcc_label_batch),flag=flag)
# 神经网络最终预测值的下标
index = get_predict_index(logits)  # 神经网络最终的输出预测值
sess.run((tf.global_variables_initializer()))  # 初始化
```

5.8.5仿真器指标获取

```
while(True):
    # ------------     仿真器指标获取     ------------
    bandwidth.append(feedbackPacket.average_bandwidth)
    target_bitrate, _, _, _, _, _, _ =
congestion_controller.OnRTCPFeedbackPacket(
        feedbackPacket)
    loss1 = feedbackPacket.loss
    send_time = feedbackPacket.send_time_ms
    arrival_time = feedbackPacket.arrival_time_ms
    payload_size = feedbackPacket.payload_size
    average_bandwidth = float(
        feedbackPacket.average_bandwidth) / 1000000.0
    delay_send = []
    delay_arrival = []
    delay = []
    delay_interval = []
    arrival_time_last_state = arrival_time[-1]
    delay_interval.append(
        (send_time[0] - send_time_last_state) - (arrival_time[0] -
arrival_time_last_state))
    for i in range(1, S_LEN):
        delay_send.append(send_time[i] - send_time[i-1])
        delay_arrival.append(arrival_time[i] - arrival_time[i-1])
        delay_interval.append(
            delay_arrival[i-1] - delay_send[i-1])  # delay_interval
    send_time_last_state = send_time[-1]
    arrival_time_last_state = arrival_time[-1]
    for i in range(S_LEN):
        delay.append(arrival_time[i] - send_time[i])
    intervals = arrival_time[-1] - start_arrival_time
    throughput = np.sum(payload_size) / intervals / 1000
    start_arrival_time = arrival_time[-1]
```

```
    # ----------------------------------------

    # target_bitrate是仿真器内部gcc计算出来的目标码率，将其映射在ACTOR_VECTOR上
    # target_bitrate_index是映射结果的下标
    target_bitrate_index = int(
        target_bitrate / 100000 + 0.5)  # 将目标码率映射到相应的下表
    if (target_bitrate_index >= ACTOR_SLOTS):
        target_bitrate_index = ACTOR_SLOTS - 1  # 下标过大则映射为最大值

    # loss 和 delay_interval 填入input
    input_loss_and_interval[0][:4] = loss1[:4]
    input_loss_and_interval[0][4:] = delay_interval[:4]

    # 构造input_throughput
    if (simulate_round == 0):
        # 第一轮时将10行都设置为同一个throughput
        for i in range(0, 10):
            input_throughput[0][i] = throughput
    else:
        # 每行上移，将最后一行设置为新的throughput模拟结果
        for i in range(0, 9):
            input_throughput[0][i] = input_throughput[0][i + 1]
        input_throughput[0][9] = throughput

    # 在 batch 的相应位置填入 input 数据，按照 unpdate_size 取模填充 batch
    obs_loss_and_interval_batch[simulate_round %
                                args.UPDATE_SIZE] = input_loss_and_interval[0]
    gcc_label_batch[simulate_round %
                    args.UPDATE_SIZE] = target_bitrate_index
    obs_throughput_batch[simulate_round %
                         args.UPDATE_SIZE] = input_throughput[0]
```

5.8.6 启停逻辑

此处维护5.8.2中提到的参数，逻辑为：如果此时读到loss列表中的最后一个loss（新添加进来的loss）大于训练过程中最好的loss，则说明此次训练的结果是不理想的（loss出现回升），因此会记录到没有优化的次数，否则我们称这次loss是理想的（最小的），并存入bestloss。对于no_optim进行累计，若这个累计值大于20，说明经过了很多轮效果都没有提升，可能出现了轻微过拟合的情况，此时停止网络训练，记录当前模型。若此累计值大于10，也是说明数据产生了过拟合，但是这种过拟合可能是由于optimizer选择不当导致的，因此设置flag为1，便回到了5.5当中的逻辑，此时lr的衰减从指数衰减转化为多项式衰减，从而缩小训练的步长。

```
if (simulate_round + 1) % args.UPDATE_SIZE == 0:
    runable(x_train1=obs_loss_and_interval_batch, x_train2=obs_throughput_batch,
y_train=gcc_label_batch, train_op=train_op,
            loss=loss,
            acc=acc, logits=logits, x1=x_loss_and_interval, x2=x_throughput,
y_=y_gcc_label, distance=distance, global_steps=global_steps,
            batch_size=32, sess=sess, epoch=epoch)
    epoch += 1
    print("epoch=",epoch)
    if(losslist[len(losslist)-1]>=train_epoch_best_loss):
        no_optim+=1
    else:
        no_optim=0
```

```
            train_epoch_best_loss=losslist[len(losslist)-1]
        if no_optim>20:
            print('early stop at %d epoch'%epoch)
            saver.save(sess, "./mymodel/" + "bestmodel")
            break
        if no_optim>10:
            flag=1
        print(no_optim)
```

5.8.7 得到预测结果

```
# 得到神经网络的预测下标
predict_index = sess.run(
    index, feed_dict={x_loss_and_interval: input_loss_and_interval,
x_throughput: input_throughput})  # 得到神经网络的预测下标
if predict_index == 0:
    predict_index += 1
predict_bitrate = int(
    round(ACTOR_VECTOR[int(predict_index)] * 1e6))# 得到最终预测码率

# 用预测码率进行仿真器的视频传输，同时得到feedbackPacket，进行下一轮的更新。
feedbackPacket = router.start(predict_bitrate)
simulate_round += 1
```

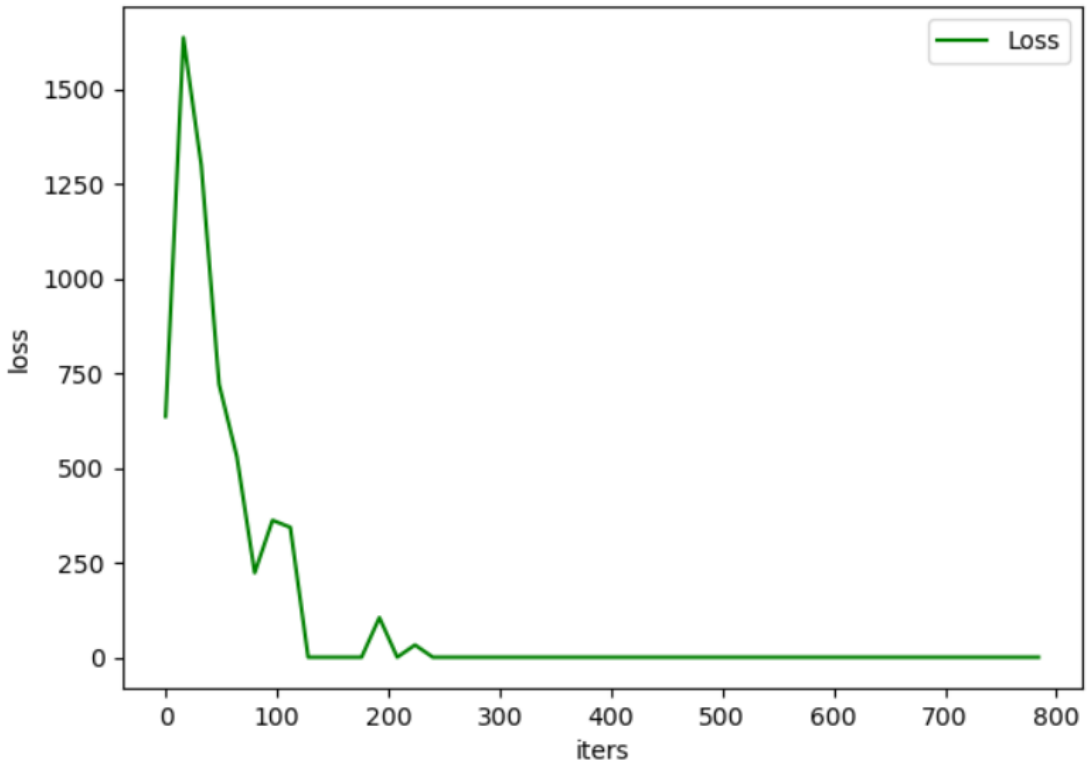
5.8.8 数据可视化

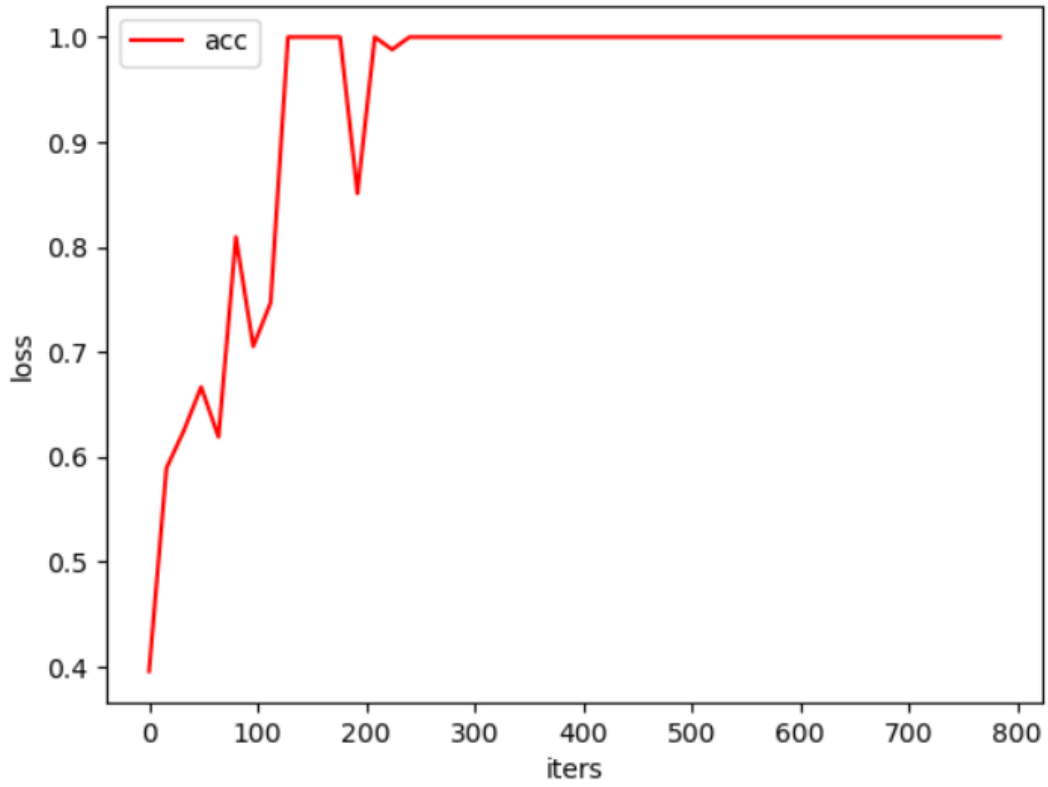
```
iter=[]
for i in range(len(losslist)):
    iter.append(i*args.UPDATE_SIZE)
fig = plt.figure(figsize=(7, 5))  # figsize是图片的大小`
ax1 = fig.add_subplot(1, 1, 1)  # ax1是子图的名字`

pl.plot(iter, losslist, 'g-', label=u'Loss')
pl.legend()
pl.xlabel(u'iters')
pl.ylabel(u'loss')
plt.show()
pl.plot(iter, acclist, 'r-', label=u'acc')
pl.legend()
pl.xlabel(u'iters')
pl.ylabel(u'acc')
plt.show()
```

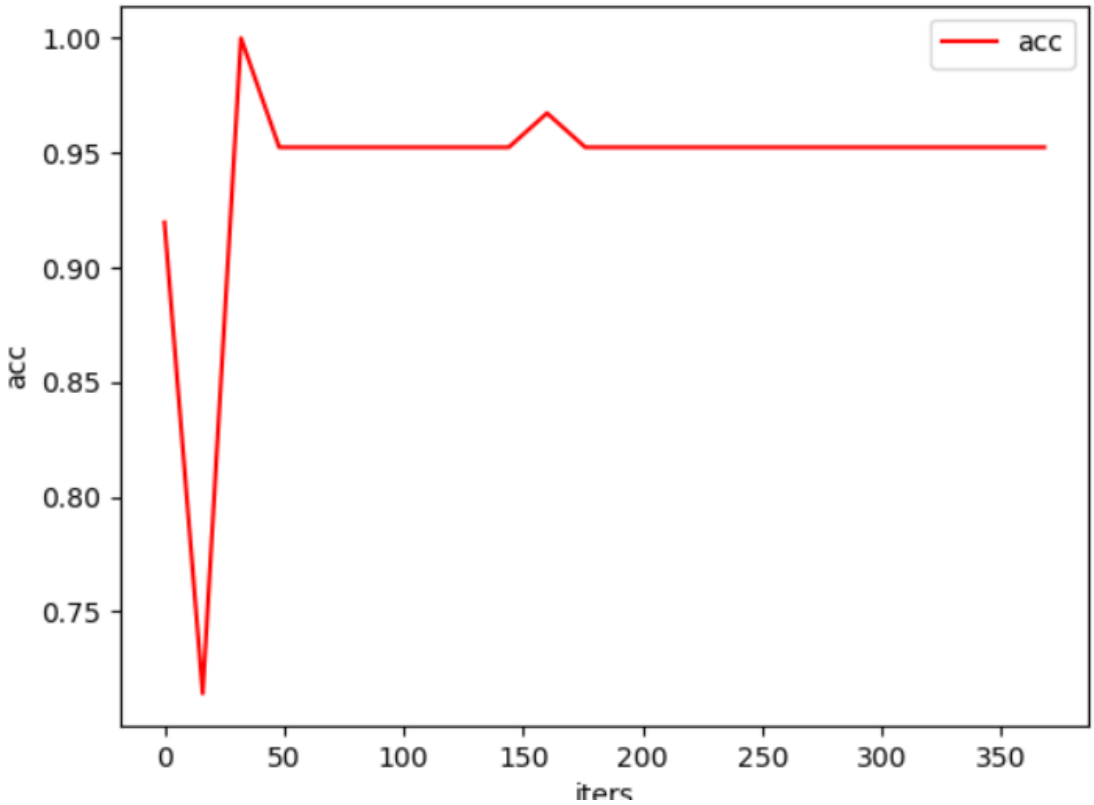
6、实验结果

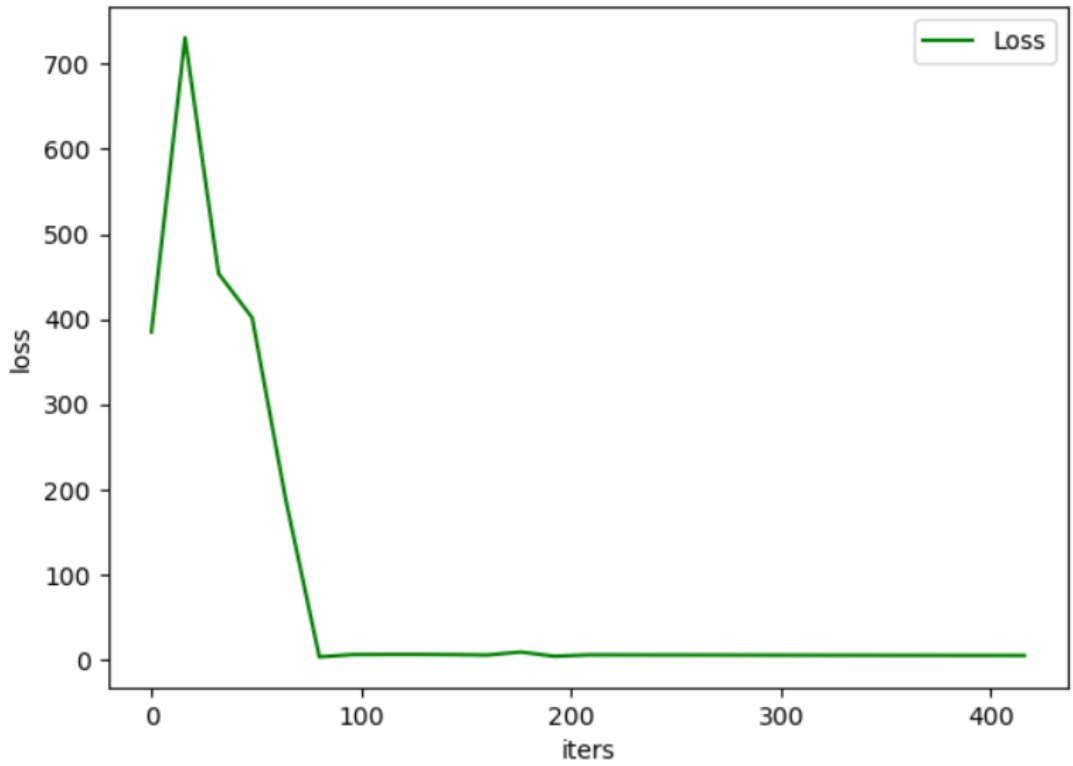
6.1模型未进行读取，从头训练，可见，此时的acc从0.4，经过800个轮回后准确率提升至96%左右





6.2 对于模型参数进行了读取，展示结果如下





7、实验总结

本次计算机网络课程设计让我对于数据的“尽力而为传输”有了更为深刻的了解，对于webrtc、gcc算法以及卡曼滤波算法都有了一定程度的认识，同时，本次实验结合了深度学习的相关知识，锻炼了我对于网络搭建、训练流程的设计、修改、优化的流程。在本次课程设计实验之前，我怎么也想不到计算机网络能够与人工智能进行结合，通过更加智能化的方法实现拥塞控制和数据传输，让我体会到了计算机网络和人工智能相关领域研究者们的智慧，同时使我对计算机网络和人工智能结合的领域产生了浓厚兴趣。

也在此感谢周安福老师和助教学长们的辛勤付出，拓宽了我的眼界，这都将是我今后学习的一段宝贵的财富。