

面向内核数据路径的高性能自适应神经网络

张骏雪^{1,2}, 曾朝亮¹, 张弘³, 胡水海^{4*}, 陈凯^{1,2}

摘要

自适应神经网络由于其能在不断变化的环境中保持较高的性能, 已经被用来优化操作系统中的一些内核数据路径功能, 例如拥塞控制、负载均衡、流量调度等。在之前的工作中, 大家的注意力都在如何设计更好的自适应神经网络上面, 但是这篇文章另辟蹊径, 指出现有自适应神经网络部署机制本身会损害自适应神经网络带来的性能增益并提出了新的神经网络部署框架 LiteFlow。LiteFlow 是一种混合式的解决方案, 其核心就是将神经网络的控制路径解耦, 将模型推理-快速路径部署在内核空间, 模型优化-慢速路径部署在用户空间, 而且这篇文章采用 LiteFlow 框架部署神经网络并实现了对部分内核数据路径功能的优化, 包括拥塞控制、流量调度和负载均衡, 与之前的工作相比 LiteFlow 在拥塞控制方面的吞吐量提高了 44.4%, 在流量调度和负载均衡方面, 长流的完成时间分别提高了 33.7% 和 56.7%^[1]。

关键词: 内核数据路径; 自适应神经网络; 部署

1 引言

操作系统内核数据路径是在内核中实现的, 是上层应用程序和下层网络硬件之间的连接通道, 它提供了多种重要的网络功能, 包括拥塞控制 (CC)、数据包的过滤和流调度等功能。近些年, 由于自适应神经网络 (ANN) 拥有适应不断变化的网络环境的能力及其卓越的拟合能力和优于手工优化算法的性能, 因此采用 ANN 对操作系统内核数据路径进行优化已经成为一种趋势。到目前为止, ANN 已用于 CC、数据包的转发、流调度等, 以优化其功能性能, 例如, 让 CC 实现更好的吞吐量, 或让流调度实现更好的流完成时间 (FCT) 等。以 CC 为例, Aurora 是一个三层的 ANN, 可以实现比 BBR 更低的延迟, 同时快速适应不同的网络环境。

尽管前景看好, 但目前存在的神经网络 (NN) 部署机制在很大程度上损害了上述优势。一种方法是在用户空间中部署 NN, 使用目前现有的成熟的工具, 在用户空间部署 NN 很容易, 但 NN 需要经常与内核空间的网络数据路径功能通信。在本文中, 我们发现, 无论如何设置通信间隔, 跨空间通信都会损害数据路径功能的性能, 从而影响自适应神经网络带来的好处。另一种方法是直接在核空间中部署 NN。在内核空间部署 NN, 目前有两种思路。一种思路是在内核空间中部署完整的 NN, 包括模型的推理和优化。然而, 由于两个原因, 这种部署方式会导致不可避免的数据路径函数性能下降: (1) 自适应 NN 的模型优化算法需要消耗大量地计算资源, 与数据路径函数本身的工作逻辑相互干扰 (2) 虽然使用 SIMD/FP 指令等高级 CPU 指令可以实现高精度, 但它也导致了额外的开销, 进一步降低了数据路径函数的性能; 另一种思路是放弃模型的优化, 将 NN 转换为仅用于推理的轻量级 NN, 例如, 通过整数量化优化 NN 或转换为决策树。然而, 这些轻量级 NN 缺乏重要的学习和适应环境的能力, 也会降低数据路径函数的性能^[1]。

如何为数据路径函数设计高性能的神经网络呢? 在我选择的这篇文章里指出现有的 NN 部署方式都是将模型推理和模型优化视为一个整体, 然而, 模型推理需要快速执行, 这更适合于内核空间; 模型优化需要高精度和密集的计算, 这更适合于用户空间。因此, 在这篇文章中提出了一种混合解决方

案 LiteFlow，它将 NN 的控制路径解耦为用于模型推理的内核空间快速路径和用于模型优化的用户空间慢速路径，以便分别在正确的位置执行模型推理和模型优化，事实也证明采用 LiteFlow 框架部署的神经网络，表现的更好。

2 相关工作

2.1 自适应神经网络的用户空间部署方式

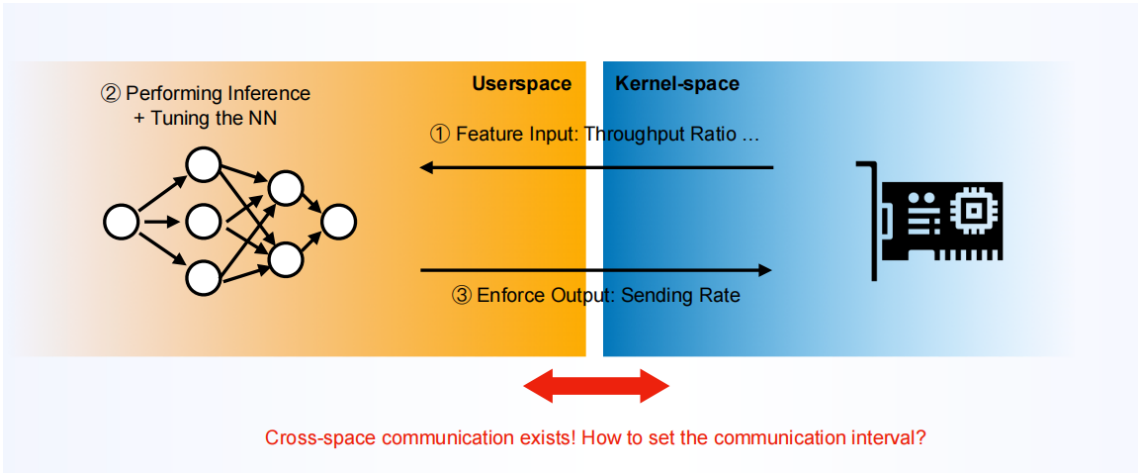


图 1: 用户空间部署神经网络

在用户空间部署自适应神经网络，我们以拥塞控制为例，当网络环境发生变化时，内核数据路径函数需要进行相应的调整，例如上图，为了响应网络拥塞需要改变流的发送速率。首先，内核数据路径函数向用户空间部署的 NN 发送所需的输入数据，即特征输入，图中发送的是吞吐量比率，以计算相应的推断结果，然后这些结果将被发送回内核空间，即流的发送速率。

虽然在用户空间部署实现起来比较简单，但是它带来了不可避免的性能损失。为了能快速响应网络变化，用户空间部署的 NN 需要与内核空间中相关的数据路径功能进行频繁通信，两个空间之间的频繁通信会消耗大量的 CPU 资源。从而分配给内核数据路径函数的 CPU 资源（用来执行数据包/流量处理逻辑）会降低，因此现有的自适应神经网络在支持高并发时无法提供高性能^[1]。

2.2 自适应神经网络的内核空间部署方式

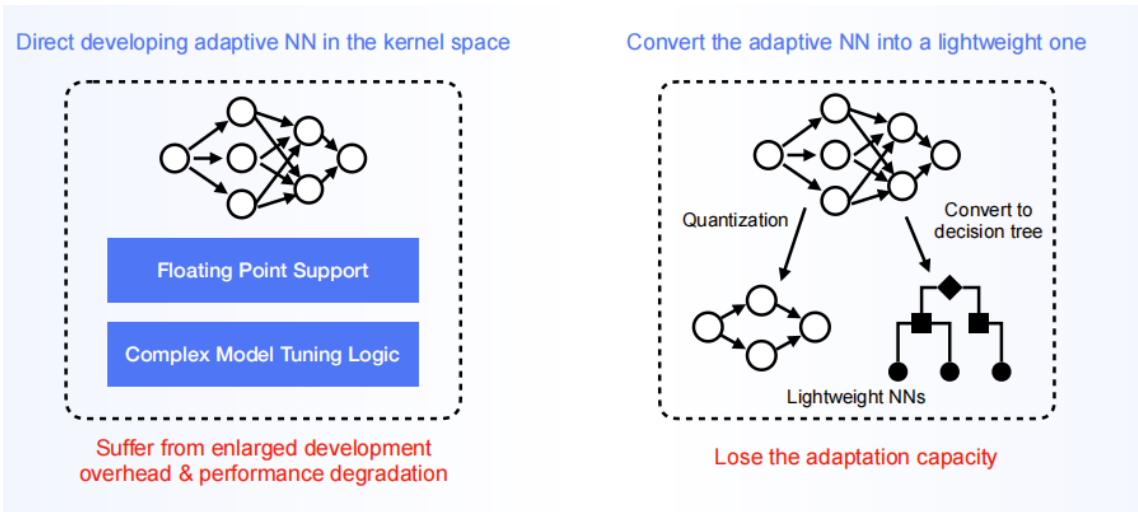


图 2: 内核空间部署神经网络的两种情况

如上图所示，为了消除跨空间通信造成的性能损失，可以在内核空间中部署自适应神经网络，目前有两个方向，但都存在功能性能下降的问题。一个方向是在内核空间中直接实现自适应神经网络，包括神经网络的推理和优化。(1) 这种方法使得 NN 的开发和调试都非常困难，因为我们必须要使用 C 等系统编程语言，并且在内核空间中受到各种限制，例如有限的库支持等。尽管有一些研究工作，如 KMLib，旨在降低内核空间中 NNs 的开发难度，但它们依旧存在不可避免的性能退化问题。(2) 为了实现神经网络自适应，我们必须实现模型优化算法，如随机梯度下降。由于这些算法需要过于复杂的计算，因此直接在内核空间中实现自适应神经网络会降低性能。(3) 在纯整数开发环境中，实现这些算法要么会损失精度，要么会增加使用 SIMD/FP 指令的开销。实验表明，即使使用批量数据，吞吐量也会下降 90%^[1]。

另一个方向是放弃神经网络优化，将神经网络转换为仅用于推理的一次性轻量级神经网络。我们可以执行整数量化以避免使用 SIMD/FP 指令或将这些 NNs 转换为与内核空间兼容的基于 C/C++ 的决策树，因此这些轻量级 NNs 有可能在内核空间中高效执行。然而，这些轻量级神经网络失去了一个重要的特性，学习和适应动态环境。实验表明，如果没有这种特性，数据路径函数会遭受严重的性能损失。因此直接在内核空间部署自适应神经网络会损害性能，原因是实现复杂度、开销大或缺乏对变化环境的适应能力^[1]。

3 混合式解决方案 LiteFlow

3.1 LiteFlow 概述

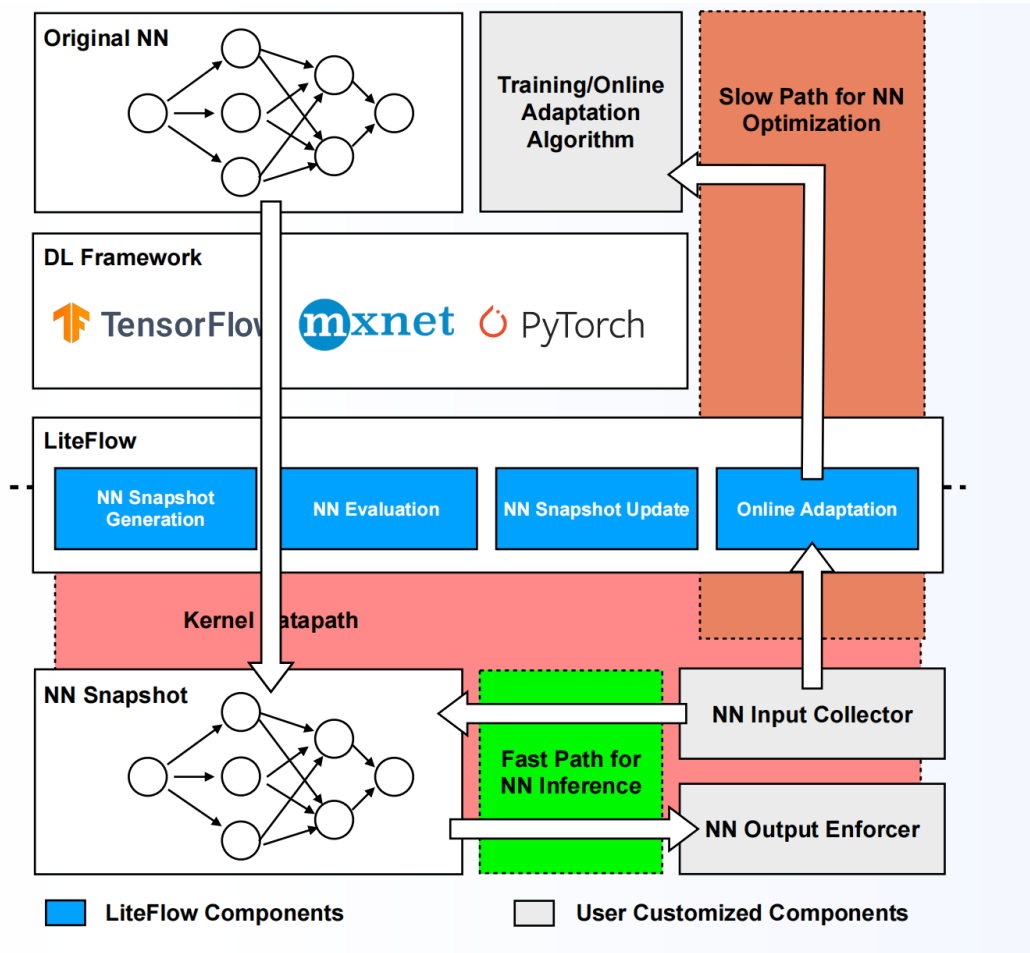


图 3: litelflow 框架结构及工作流程示意图

LiteFlow 是为内核数据路径部署高性能自适应神经网络提出的混合式解决方案, LiteFlow 与上述 2.1 和 2.2 的部署方式的不同之处在于, LiteFlow 并非采用一条控制路径来进行模型推理和模型优化, 而是将其解耦为两条路径, 一条是部署在内核空间的快速路径, 用于模型推理, 另一条是部署在用户空间的慢速路径, 用于模型优化, 这样一来就可以解决无论是用户空间还是内核空间的自适应神经网络部署方案都存在的功能性能损失的固有问题。为什么 LiteFlow 能够解决现有自适应神经网络部署方式存在的功能性能损失的固有问题呢? 可以从以下两个方面考虑 (1) 用于内核数据路径函数的自适应神经网络需要非常频繁的推理才能响应不断变化的网络环境, 而模型的推理又需要与相应的内核空间数据路径函数进行通信, 因此将模型推理部署在内核空间, 可以避免频繁的跨空间通信带来的额外开销 (2) 将模型优化部署在用户空间是因为它能更方便的使用用户空间提供的 API、库和成熟软件带来的高级功能, 例如浮点/多线程的支持, 这可以简化模型优化算法的实现^[1]。LiteFlow 框架的结构以及工作流程图如图 3 所示:

3.2 liteflow 的工作流程

LiteFlow 的具体工作流程如下: (1)Snapshot Generation LiteFlow 首先生成给定神经网络的快照并部署到内核空间, 内核空间的快照仅仅用于推理。同时 LiteFlow 还收集快照的输入和输出数据 (输入数据, 以拥塞控制为例, 输入数据就是快照执行推理时的数据, 比如收集到的一些拥塞控制的信号, 输出数据便是推理得到的预测结果), 以进一步调整 (批处理模式下, 在慢速路径中的) 用户空间的神经网络 (NN) (2)Online Adaptation LiteFlow 使用收集到的数据, 分批量对用户空间部署的神经网络进行优化 (3)NN Evaluation&Update 在每次批训练后 (Online Adaption), LiteFlow 都会从正确性和必要性两个方面评估是否需要更新快照。值得注意的是, LiteFlow 没有明确评估神经网络的性能, 而是依赖于一个共识, 即在线适应可能会在收敛后产生更好的神经网络^[1]。

3.3 liteflow 的核心模块实现

LiteFlow 的实现包括用户空间和内核空间的实现。

3.3.1 用户空间实现

LiteFlow 的用户空间提供了一组 python 接口供用户实现, 它还提供了一个服务来接受实现这些接口的用户定义的 python 类。通过允许用户实现接口, LiteFlow 没有与任何深度学习或强化学习框架紧密耦合 (TensorFlow、Pytorch 都是深度学习框架), 因此 LiteFlow 用户可以使用他们的首选框架来优化神经网络。此外, 通过提供这些标准 API, LiteFlow 可以灵活地支持新的神经网络 (这需要额外实现新的输入收集器和输出执行器)^[1]。具体来说, 主要接口如下:

(1)NN Freezing Interface 该接口用于生成 NN 快照。要实现这个接口, 用户应该保存模型并返回保存模型的路径。

(2)NN Evaluation Interface 该接口用于评估 NN 从正确性和必要性确定是否需要更新。该接口要求 LiteFlow 用户实现两个功能 a. 正确性, 返回给定模型的一个稳定的指标, 这个指标我们可以灵活的选择, 例如训练损失。LiteFlow 会监测该值一段时间来确定在线自适应是否收敛, 即该值在小范围内变化 b. 必要性, 返回保真度损失值。为了评估必要性, 文中引入了保真度损失的定义。用 f 表示用户空间中的 NN, f' 表示内核空间中的 NN 的快照, 保真度损失定义为: $L(x) = |f'(x) - f(x)|$, 其中 x 表示输入

数据。当给定一组输入数据时，计算 NN 快照的保真度损失，当保真度损失超过用户定义的阈值时，即两个 NN 之间差异足够大时，才有必要更新 NN 快照。

(3)NN Online Adaptation Interface 该接口用于用户空间中 NN 的优化。为了实现该接口，LiteFlow 用户必须使用脚本或程序来调整 NN。如上所述，用户可以使用任何深度学习框架 (例如 TensorFlow) 或强化学习工具 (例如 GYM) 来实现模型调优的逻辑。

3.3.2 内核空间实现

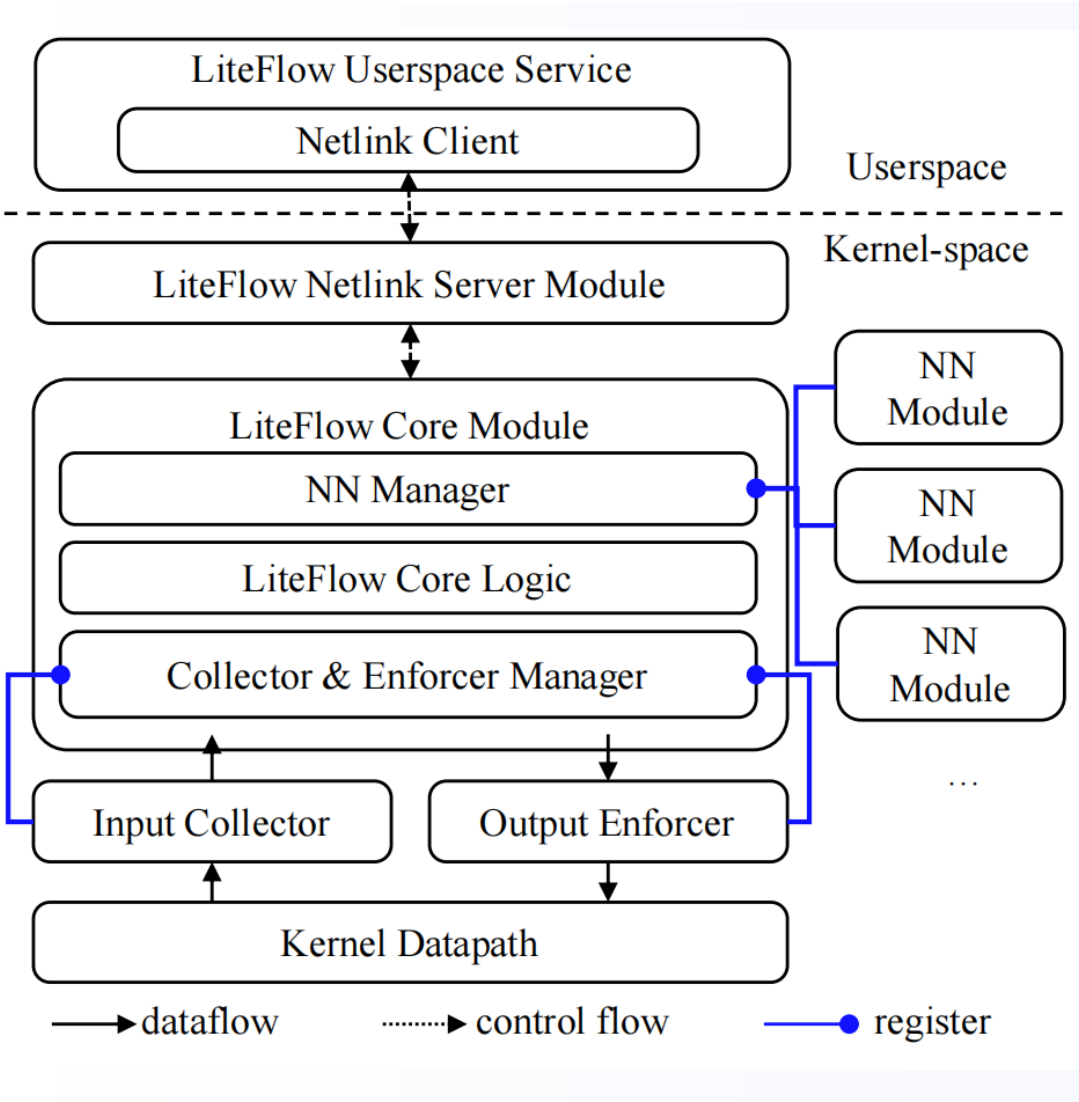


图 4: liteflow 内核空间的主要模块

LiteFlow 的内核空间实现是在 Linux 内核 4.15.0 中实现的。由于 LiteFlow 的设计目标之一是可以支持各种数据路径功能的自适应神经网络，因此文中采用模块化原则来设计 LiteFlow 的内核空间组件^[1]。图 4显示了 LiteFlow 内核空间的主要模块，接下来将介绍这些模块。

(1)LiteFlow Netlink Module: 该模块注册在内核空间中的 netlink 子系统中，用来与 LiteFlow 用户空间的服务通信。有两种类型的消息通过该 netlink 信道传输：a. 用于在线自适应新收集的数据 b. 当给定一组用于必要性评估的输入数据时，NN 快照的输出。

(2)LiteFlow Core Module: 该模块由四个模块组成 a. NN Manager 它实现了一个 NN 管理器，NN 管理器使用一个链表来操作所有已安装的 NN 快照，它提供了 lf_register_model API 来安装新的快照 b. LiteFlow Core Logic 包括 NN 的评估和更新逻辑 c. Collector&Enforcer Manager 用来管理 Input Collector

和 Output Enforcer，以将不同的 NN 与不同的数据路径功能集成 d. LiteFlow Core Module 该模块为其它内核空间模块使用神经网络提供了统一的推理接口 `lf_query_model`。

(3)NN Module: 每个 NN 快照都是一个单独的内核模块。LiteFlow 生成快照的内核空间实现，并调用 GCC 将其编译为一个内核模块。LiteFlow 用户空间服务调用 `insmod` 系统指令来安装模块，在模块的初始化中，我们必须调用 `lf_register_model` 向 LiteFlow Core Module 注册 NN。在注册期间，我们需要告诉 LiteFlow NN 的输入和输出大小。

(4)Input Collector&Output Enforcer: 为了支持各种数据路径功能，LiteFlow 应提供将自适应神经网络与不同数据路径功能集成的灵活性。因此，如果要优化新的数据路径功能，LiteFlow 要求用户在内核数据路径中实现自己的数据收集 (例如，收集 ECN 字节，TCP 状态) 和输出强制逻辑 (例如，基于 NN 的输出，设置拥塞窗口、流优先级等)。用户可以使用 `lf_register_io` 和 `lf_unregister_io` API 动态添加或删除数据收集和输出强制模块。此外，API 将检查这些用户定义模块中 NN 所需的输入和输出大小是否安装的 NN 一致。

4 复现细节

4.1 liteflow 核心代码实现

在 LiteFlow 的复现过程中，引用了作者发布的所有代码，下面将展示部分代码，详细代码参考 <https://github.com/snowzjx/liteflow>

`nn_freeze` 是用户定义的 python 接口，首先 LiteFlow 的用户空间服务调用 `nn_freeze()` 接口获得部署在用户空间的神经网络。

```
def nn_freeze():  
    return "../data/aurora/"
```

然后进一步利用 TensorFlow Lite 对神经网络进行量化，生成要部署在内核空间快速路径中的神经网络快照。

```
def _quant_model_output_path(uuid):  
    return "../build/aurora_int_quan_model_{}.tflite".format(uuid)  
  
def _quant_nn_from_saved_model(path):  
    quant_model_output_path_real = _quant_model_output_path(uuid)  
    lf_quant(path, representative_dataset, quant_model_output_path_real)  
    lf_generate_snapshot(quant_model_output_path_real, uuid, appid_for_cc,  
        ↪ kernel_model_quant_path, False)  
    TEMPLATE_FILE = "Makefile.jinja"  
    _template = template.get_template(TEMPLATE_FILE)  
    code = _template.render(uuid=uuid)  
    OUTPUT_FILE = f"{kernel_model_quant_path}Makefile"  
    with open(OUTPUT_FILE, "w") as output_file:
```



```
output_file.write(code)
return quant_model_output_path_real
```

其次，通过 netlink 从内核空间批量获取数据 (代码中以拥塞控制相关数据为例)，并调用神经网络的 NN Online Adaptation Interface 接口对用户空间部署的神经网络进行优化。

```
# 从内核空间传输数据到用户空间训练
static inline int report_to_user(s64 *nn_input, u32 input_size) {
    struct sk_buff *skb;
    void *msg_head;
    int ret;
    skb = genlmsg_new(NLMSG_GOODSIZE, GFP_ATOMIC);
    if (skb == NULL) {
        printk (KERN_ERR "Cannot allocate skb for LiteFlow TCP netlink ...\n");
        return LF_ERROR;
    }
    msg_head = genlmsg_put(skb, 0, 0, &lf_tcp_gnl_family, GFP_ATOMIC,
↪ LF_TCP_NL_C_REPORT);
    if (msg_head == NULL) {
        printk (KERN_ERR "Cannot allocate msg_head for LiteFlow TCP netlink ...\n");
        return LF_ERROR;
    }
    ret = nla_put(skb, LF_TCP_NL_ATTR_NN_INPUT, input_size * sizeof(s64), nn_input);
    if (ret != 0) {
        printk (KERN_ERR "Cannot put data for LiteFlow TCP netlink, error code: %d
↪ ...\n", ret);
        return LF_ERROR;
    }
    genlmsg_end(skb, msg_head);
    genlmsg_multicast(&lf_tcp_gnl_family, skb, 0, LF_TCP_NL_MC_DEFAULT, GFP_ATOMIC);
    return LF_SUCCS;
}
```

在每个批训练之后，LiteFlow 进一步调用用户空间的 NN Evaluation Interface 接口，以根据正确性和必要性来确定快照是否需要更新。

```
def nn_evaluate():
    new_freezed_nn_path = nn_freeze()
    quant_model_output_path_real=_quant_nn_from_saved_model(new_freezed_nn_path)
    # 内核空间中的神经网络的快照
    prediction1 = _tf_inference(quant_model_output_path_real)
    # 用户空间中的神经网络
```

```

prediction2 = _tf_inference(current_installed_model)
# 必要性
fidelity_loss = _cal_fidelity_loss(prediction1, prediction2)
# 正确性: stability 为 true 时, 即在线自适应收敛, 才能更新快照
return quant_model_output_path_real, True, fidelity_loss

```

采用主备切换方式更新快照, 生成一个新的快照, 部署到内核空间.

```

//register a new NN snapshot to LiteFlow
int
lf_register_model(u8 appid, struct model_container *model)
{
    u8 ret;
    s8 info_model_uuid_0, info_model_uuid_1;
    // Install
    if (appid > MAX_APP) {
        printk(KERN_ERR "Unsupported appid: %u.\n", appid);
        return LF_ERROR;
    }
    if (apps[appid].appid == APP_ID_UNUSE) {
        printk(KERN_ERR "Need to register app before registering model.\n");
        goto error;
    }
    if (apps[appid].input_size != model->input_size) {
        printk(KERN_ERR "Input size of app and model are not consistent.\n");
        goto error;
    }
    if (apps[appid].output_size != model->output_size) {
        printk(KERN_ERR "Output size of app and model are not consistent.\n");
        goto error;
    }
    printk(KERN_INFO "Registering model with uuid: %u to app with appid: %u...\n",
↪ model->uuid, appid);
    ret = init_model(model);
    if (ret == LF_ERROR) {
        goto error;
    }
    if (apps[appid].active_model == 0) {
        if (apps[appid].model_slot_1 != NULL) {
            printk(KERN_INFO "Deleting model with uuid: %u to app with appid: %u...\n",
↪ apps[appid].model_slot_1->uuid, appid);
            destroy_model(apps[appid].model_slot_1);
        }
    }
}

```



```

        apps[appid].model_slot_1 = model;
    } else {
        if (apps[appid].model_slot_0 != NULL) {
            printk(KERN_INFO "Deleting model with uuid: %u to app with appid: %u...\n",
↪ apps[appid].model_slot_0->uuid, appid);
            destroy_model(apps[appid].model_slot_0);
        }
        apps[appid].model_slot_0 = model;
    }
    // Lock
    write_lock(&lf_lock);
    if (apps[appid].active_model == 0) {
        apps[appid].active_model = 1;
    } else {
        apps[appid].active_model = 0;
    }
    write_unlock(&lf_lock);
    printk(KERN_INFO "Model with uuid: %u is registered to app with appid: %u!\n",
↪ model->uuid, appid);

    if (apps[appid].model_slot_0 != NULL) {
        info_model_uuid_0 = apps[appid].model_slot_0->uuid;
    } else {
        info_model_uuid_0 = -1;
    }

    if (apps[appid].model_slot_1 != NULL) {
        info_model_uuid_1 = apps[appid].model_slot_1->uuid;
    } else {
        info_model_uuid_1 = -1;
    }
    printk(KERN_INFO "Current slot 0 is registered with model: %d\n",
↪ info_model_uuid_0);
    printk(KERN_INFO "Current slot 1 is registered with model: %d\n",
↪ info_model_uuid_1);
    printk(KERN_INFO "Current active slot is: %u\n", apps[appid].active_model);
    return LF_SUCCS;

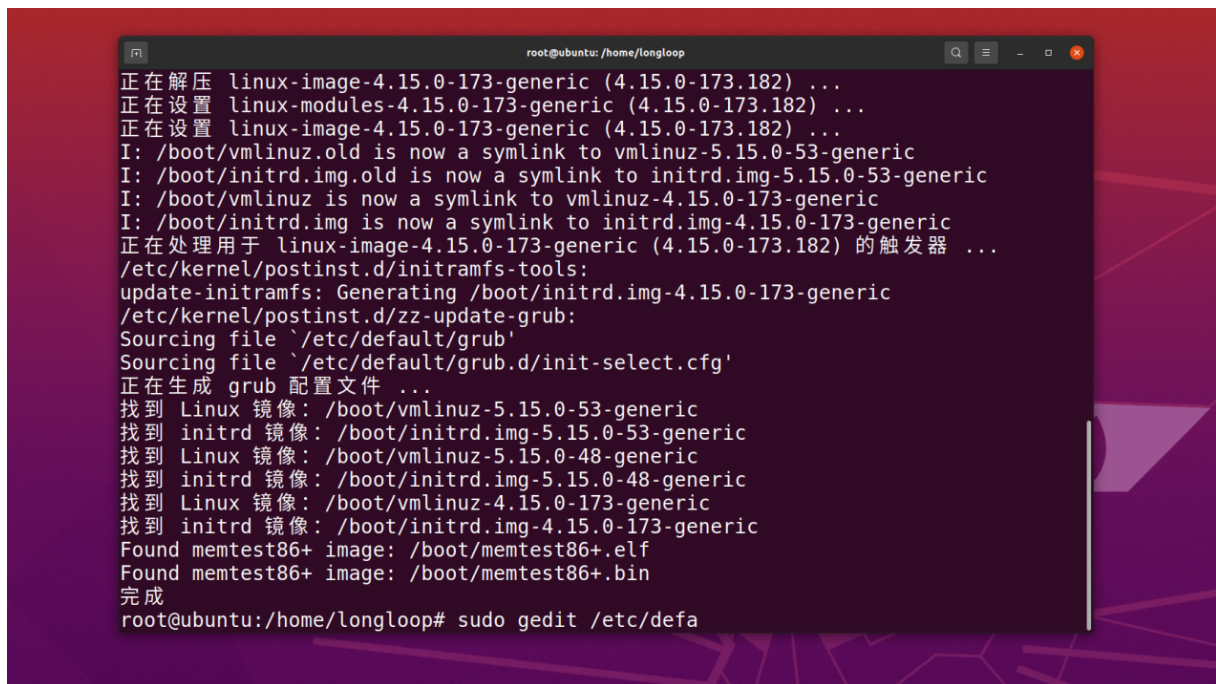
```

4.2 实验环境搭建

实验环境: VMware Workstation 16 Pro、Ubuntu20.04.1, 由于 LiteFlow 是在内核版本为 4.15.0-173-generic 中实现的, 而我的 Ubuntu 里面只有 5.15.0-53-generic 版本的内核, 所以需要安装新的内核, 并安装实验所需要的其它文件。

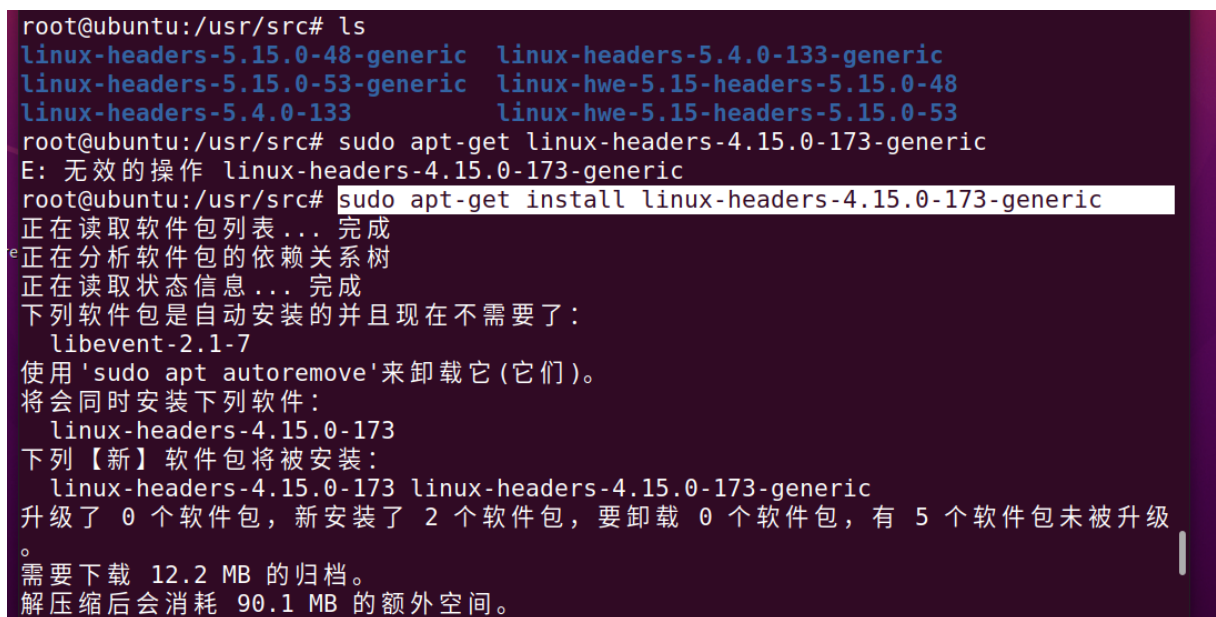
(1) 使用 `sudo apt-get install linux-image-4.15.0-173-generic` 命令, 安装 linux 内核 4.15.0-173-generic,

再使用 `sudo apt-get install linux-headers-4.15.0-173-generic` 安装内核对应的头文件，最后使用 `sudo gedit /etc/default/grub` 命令打开 grub 文件，用 `dpkg --get-selections | grep linux-image` 命令查找所有内核安装、卸载的记录检查内核 4.15.0-173-generic 是否安装成功



```
root@ubuntu: /home/longloop
正在解压 linux-image-4.15.0-173-generic (4.15.0-173.182) ...
正在设置 linux-modules-4.15.0-173-generic (4.15.0-173.182) ...
正在设置 linux-image-4.15.0-173-generic (4.15.0-173.182) ...
I: /boot/vmlinuz.old is now a symlink to vmlinuz-5.15.0-53-generic
I: /boot/initrd.img.old is now a symlink to initrd.img-5.15.0-53-generic
I: /boot/vmlinuz is now a symlink to vmlinuz-4.15.0-173-generic
I: /boot/initrd.img is now a symlink to initrd.img-4.15.0-173-generic
正在处理用于 linux-image-4.15.0-173-generic (4.15.0-173.182) 的触发器 ...
/etc/kernel/postinst.d/initramfs-tools:
update-initramfs: Generating /boot/initrd.img-4.15.0-173-generic
/etc/kernel/postinst.d/zz-update-grub:
Sourcing file '/etc/default/grub'
Sourcing file '/etc/default/grub.d/init-select.cfg'
正在生成 grub 配置文件 ...
找到 Linux 镜像: /boot/vmlinuz-5.15.0-53-generic
找到 initrd 镜像: /boot/initrd.img-5.15.0-53-generic
找到 Linux 镜像: /boot/vmlinuz-5.15.0-48-generic
找到 initrd 镜像: /boot/initrd.img-5.15.0-48-generic
找到 Linux 镜像: /boot/vmlinuz-4.15.0-173-generic
找到 initrd 镜像: /boot/initrd.img-4.15.0-173-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
完成
root@ubuntu: /home/longloop# sudo gedit /etc/defa
```

图 5: 安装内核 4.15.0-173-generic



```
root@ubuntu: /usr/src# ls
linux-headers-5.15.0-48-generic  linux-headers-5.4.0-133-generic
linux-headers-5.15.0-53-generic  linux-hwe-5.15-headers-5.15.0-48
linux-headers-5.4.0-133          linux-hwe-5.15-headers-5.15.0-53
root@ubuntu: /usr/src# sudo apt-get linux-headers-4.15.0-173-generic
E: 无效的操作 linux-headers-4.15.0-173-generic
root@ubuntu: /usr/src# sudo apt-get install linux-headers-4.15.0-173-generic
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了:
  libevent-2.1-7
使用 'sudo apt autoremove' 来卸载它(它们)。
将会同时安装下列软件:
  linux-headers-4.15.0-173
下列【新】软件包将被安装:
  linux-headers-4.15.0-173 linux-headers-4.15.0-173-generic
升级了 0 个软件包，新安装了 2 个软件包，要卸载 0 个软件包，有 5 个软件包未被升级。
需要下载 12.2 MB 的归档。
解压缩后会消耗 90.1 MB 的额外空间。
```

图 6: 安装内核头文件

```
root@ubuntu: /home/longloop
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
完成
root@ubuntu:/home/longloop# sudo gedit /etc/default/grub

(gedit:6559): Tepl-WARNING **: 21:30:09.516: GVfs metadata is not supported. Fall
back to TeplMetadataManager. Either GVfs is not correctly installed or GVfs met
adata are not supported on this platform. In the latter case, you should configu
re Tepl with --disable-gvfs-metadata.
root@ubuntu:/home/longloop# dpkg --get-selections | grep linux-image
linux-image-4.15.0-173-generic          install
linux-image-5.11.0-27-generic          deinstall
linux-image-5.11.0-41-generic          deinstall
linux-image-5.11.0-43-generic          deinstall
linux-image-5.11.0-44-generic          deinstall
linux-image-5.11.0-46-generic          deinstall
linux-image-5.13.0-41-generic          deinstall
linux-image-5.13.0-44-generic          deinstall
linux-image-5.13.0-52-generic          deinstall
linux-image-5.15.0-46-generic          deinstall
linux-image-5.15.0-48-generic          install
linux-image-5.15.0-53-generic          install
linux-image-generic-hwe-20.04         install
root@ubuntu:/home/longloop#
```

图 7: 检查是否安装成功

(2) 安装 Jinja2、ProtoBuf、Tensorflow 等框架。Jinja2 可以有效地将业务逻辑和页面逻辑分开，增强代码的可读性，且使代码更容易维护。ProtoBuf 的全称是 Protocol Buffers，是一种轻便高效的结构化数据存储格式，可以用于结构化数据序列化，很适合做数据存储或 RPC 数据交换格式，它可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式，可以简单理解为是一种跨语言、跨平台的数据传输格式。Tensorflow 是一个用于机器学习的开源框架，可以用来快速的构建神经网络，同时快捷地进行模型的训练、评估与保存。

```
[3]+ 已停止          vi template.py
root@ubuntu:/home/longloop/projects/liteflow/bin# cd ..
root@ubuntu:/home/longloop/projects/liteflow# pip install Jinja2==3.1.2
Collecting Jinja2==3.1.2
  WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None, status=
None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='file
s.pythonhosted.org', port=443): Read timed out. (read timeout=15)")': /packages/bc/
c3/f068337a370801f372f2f8f6bad74a5c140f6fda3d9de154052708dd3c65/Jinja2-3.1.2-py3-no
ne-any.whl
  WARNING: Retrying (Retry(total=3, connect=None, read=None, redirect=None, status=
None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='file
s.pythonhosted.org', port=443): Read timed out. (read timeout=15)")': /packages/bc/
c3/f068337a370801f372f2f8f6bad74a5c140f6fda3d9de154052708dd3c65/Jinja2-3.1.2-py3-no
ne-any.whl
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
    133.1/133.1 kB 446.1 kB/s eta 0:00:00
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.8/dist-pac
kages (from Jinja2==3.1.2) (2.1.1)
Installing collected packages: Jinja2
Successfully installed Jinja2-3.1.2
WARNING: Running pip as the 'root' user can result in broken permissions and confli
cting behaviour with the system package manager. It is recommended to use a venv
environment instead: https://pip.pypa.io/warnings/venv
root@ubuntu:/home/longloop/projects/liteflow#
```

图 8: 安装 jinja2

```
root@ubuntu:/home/longloop/projects/liteflow/script# pip install protobuf==3.19.0
Collecting protobuf==3.19.0
  WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None, status=None)) after connection broken by 'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at 0x7ff7382677f0>: Failed to establish a new connection: [Errno -3] 域名解析暂时失败')': /packages/69/ee/949eb6182636fdc4fa0e2fa02a94d79e21069d46b56d4f251d0ac39b5678/protobuf-3.19.0-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
  WARNING: Retrying (Retry(total=3, connect=None, read=None, redirect=None, status=None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='files.pythonhosted.org', port=443): Read timed out. (read timeout=15)")': /packages/69/ee/949eb6182636fdc4fa0e2fa02a94d79e21069d46b56d4f251d0ac39b5678/protobuf-3.19.0-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
  Downloading protobuf-3.19.0-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.1 MB)
    1.1/1.1 MB 1.8 MB/s eta 0:00:00
Installing collected packages: protobuf
  Attempting uninstall: protobuf
    Found existing installation: protobuf 4.21.9
    Uninstalling protobuf-4.21.9:
      Successfully uninstalled protobuf-4.21.9
  Successfully installed protobuf-3.19.0
  WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv
root@ubuntu:/home/longloop/projects/liteflow/script#
```

图 9: 安装 protobuf

5 以拥塞控制为例的实验结果分析

本部分对实验所得结果进行分析，详细对实验内容进行说明，实验结果进行描述并分析。本部分是以拥塞控制为例，展示 liteflow 的工作流程及结果。LiteFlow Congestion Control Module 模块，该模块作为自定义拥塞控制算法插入到 Linux 内核的网络堆栈中。每次接收到 ACK 时，模块都会收集相关的拥塞信号，如平均吞吐量等，并使用自适应神经网络来预测目标发送速率，为了在数据路径中强制执行发送速率，该模块通过设置 `sk_pacing_rate` 属性来执行流量调节。

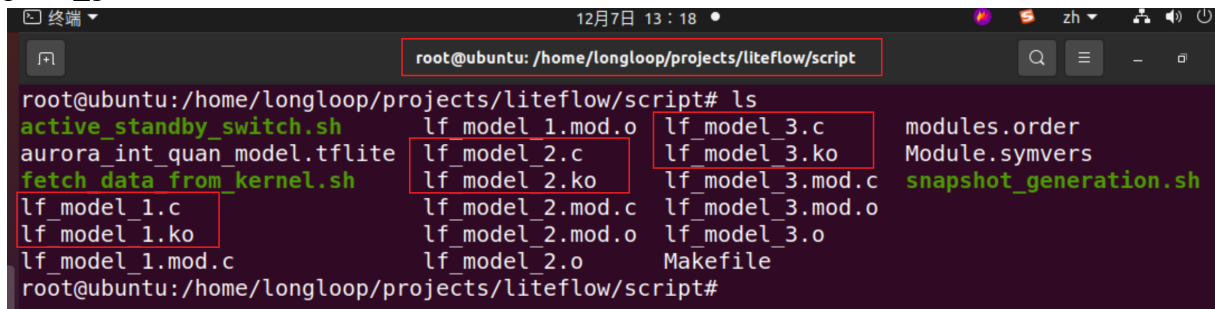
(1) 准备工作进入到 liteflow 路径下，通过命令 `make module_install` 和 `make tcp_kernel_install` 完成 LiteFlow 内核模块和 LiteFlow TCP 拥塞控制模块的编译，之后我们可以通过 `lsmod` 命令查看 `lf_kernel` 和 `lf_tcp_kernel` 是否存在

```
活动 终端 12月7日 12:34 root@ubuntu:/home/longloop/projects/liteflow
sudo sysctl net.ipv4.tcp_congestion_control=lf_tcp_kernel
net.ipv4.tcp_congestion_control = lf_tcp_kernel
root@ubuntu:/home/longloop/projects/liteflow# lsmod
Module              Size  Used by
lf_tcp_kernel        16384  1
lf_kernel            16384  1 lf_tcp_kernel
vmw_vsock_vmci_transport 32768  2
vsock               36864  3 vmw_vsock_vmci_transport
nls_iso8859_1        16384  1
binfmt_misc          20480  1
crc16               16384  0
crt10dif_pclmul      16384  0
crc32_pclmul         16384  0
ghash_clmulni_intel  16384  0
pcbc                 16384  0
vmw_balloon          20480  0
aesni_intel          188416  0
aes_x86_64           20480  1 aesni_intel
crypto_simd           16384  1 aesni_intel
```

图 10: 模块的安装

(2) Snapshot Generation 本次演示使用的是保存在 `../data/` 路径下的 Aurora 神经网络。LiteFlow 需要对其进行量化，即生成快照 (也是一个内核模块) 部署到内核空间。切换路径至 `../liteflow/script`，执

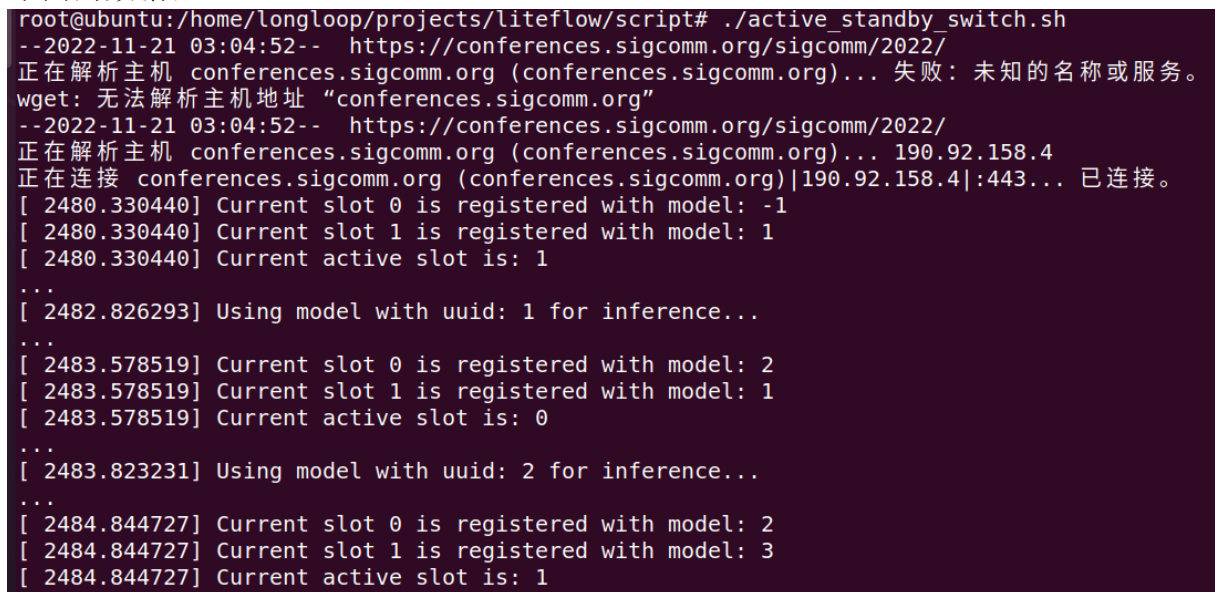
行./snapshot_generation.sh 命令，会生成 Aurora 神经网络的三个相同快照，仅仅用于测试



```
root@ubuntu: /home/longloop/projects/liteflow/script
root@ubuntu:/home/longloop/projects/liteflow/script# ls
active_standby_switch.sh  lf_model_1.mod.o  lf_model_3.c  modules.order
aurora_int_quan_model.tflite  lf_model_2.c  lf_model_3.ko  Module.symvers
fetch_data_from_kernel.sh  lf_model_2.ko  lf_model_3.mod.c  snapshot_generation.sh
lf_model_1.c  lf_model_2.mod.c  lf_model_3.mod.o
lf_model_1.ko  lf_model_2.mod.o  lf_model_3.o
lf_model_1.mod.c  lf_model_2.o  Makefile
root@ubuntu:/home/longloop/projects/liteflow/script#
```

图 11: 生成快照

(3)Snapshot Update 在每次批训练之后，LiteFlow 都会从正确性和必要性来评估是否需要更新快照。切换路径至../liteflow/script，执行命令./active_standby_switch.sh，我们可以看到是不同的神经网络来服务不同的数据流



```
root@ubuntu:/home/longloop/projects/liteflow/script# ./active_standby_switch.sh
--2022-11-21 03:04:52-- https://conferences.sigcomm.org/sigcomm/2022/
正在解析主机 conferences.sigcomm.org (conferences.sigcomm.org)... 失败：未知的名称或服务。
wget: 无法解析主机地址 "conferences.sigcomm.org"
--2022-11-21 03:04:52-- https://conferences.sigcomm.org/sigcomm/2022/
正在解析主机 conferences.sigcomm.org (conferences.sigcomm.org)... 190.92.158.4
正在连接 conferences.sigcomm.org (conferences.sigcomm.org)|190.92.158.4|:443... 已连接。
[ 2480.330440] Current slot 0 is registered with model: -1
[ 2480.330440] Current slot 1 is registered with model: 1
[ 2480.330440] Current active slot is: 1
...
[ 2482.826293] Using model with uuid: 1 for inference...
...
[ 2483.578519] Current slot 0 is registered with model: 2
[ 2483.578519] Current slot 1 is registered with model: 1
[ 2483.578519] Current active slot is: 0
...
[ 2483.823231] Using model with uuid: 2 for inference...
...
[ 2484.844727] Current slot 0 is registered with model: 2
[ 2484.844727] Current slot 1 is registered with model: 3
[ 2484.844727] Current active slot is: 1
```

图 12: 快照更新

6 总结与展望

通过复现这篇文章，将我的关注点从高性能神经网络本身的开发转移到了怎么部署神经网络获得更高的性能上面。文章给出了用 LiteFlow 部署的神经网络优化了三个数据路径模型，并且获得了更好的性能。但是 LiteFlow 仍然面临着一些问题，例如，如果我们要用 LiteFlow 去优化新数据路径功能，就需要自己定义，数据收集器和输出执行器，这需要广泛的领域知识并涉及复杂的内核空间编程，导致使用 LiteFlow 的学习门槛过高，如果 LiteFlow 的应用领域扩大，那么在这个方面，我们可以尝试开发一些框架出来，让用户更方便的使用 LiteFlow。

参考文献

- [1] Junxue Zhang, chaoliang Zeng, hong Zhang, et al. LiteFlow: towards high-performance adaptive neural networks for kernel datapath[J]. Proceedings of the ACM SIGCOMM 2022 Conference, 2022, 14: 414-427.