Nvidia Jetson测试

一、Nvidia jetpack

NVIDIA JetPack SDK 是构建端到端加速 AI 应用的全面解决方案。JetPack 为硬件加速的边缘 AI 开发提供了完整的开发环境。JetPack 支持所有 Jetson 模组和开发者套件。

JetPack 包括带有引导加载程序的 Jetson Linux、Linux 内核、Ubuntu 桌面环境，以及一整套用来为 GPU 计算、多媒体、图形和计算机视觉加速的库。它还包含用于主机和开发者套件的示例、文档和开发者工具，并支持更高级别的 SDK，例如用于流媒体视频分析的**DeepStream**、用于机器人开发的 Isaac 以及用于对话式 AI 的 Riva。

JetPack 5.0.2 包括搭载 Linux 内核 5.10 的 Jetson Linux 35.1 BSP、基于 Ubuntu 20.04 的根文件系统、基于 UEFI 的引导加载程序以及作为可信执行环境的 OP-TEE。JetPack 5.0.2 包括 Jetson 上的新版计算栈，配备了 **CUDA 11.4、TensorRT 8.4.1 和 cuDNN 8.4.1。**

- **CUDA**：CUDA 工具套件为 C 和 C++ 开发者构建 GPU 加速应用提供了全面的开发环境。该工具包中包括一个针对 NVIDIA GPU 的编译器、多个数学库，以及多款用于调试和优化应用性能的工具。
- **cuDNN**：CUDA 深度神经网络库为深度学习框架提供了高性能基元。它可大幅优化标准例程（例如用于前向传播和反向传播的卷积层、池化层、归一化层和激活层）的实施。
- **TensorRT**：TensorRT高性能推理框架依托于 CUDA 而构建，是 NVIDIA 的并行编程模型，支持优化各种深度学习框架的推理过程，让深度学习推理应用实现低延迟和高吞吐量，集成了NVIDIA性能分析工具NVIDIA Nsight™ Systems 与 NVIDIA Deep Learning Profiler (DLProf)。同时提供C++与Python两种API，Python API可以与Numpy、Scipy等科学计算库一同使用，追求极致性能则使用C++。

详情见：https://developer.nvidia.cn/embedded/jetpack。

二、配置VsCode远程开发Jetson

1. jetson设置静态IP

（1）终端输入：

```
sudo vim /etc/network/interfaces
```

（2）注释掉所有信息，输入以下信息，将ip地址固定为192.168.1.102，iface后的wlan0要根据ifconfig中连接的网络类型而定。

```
iface wlan0 inet static                # 设置静态ip
address 192.168.1.102                # ip地址
netmask 255.255.255.0                # 掩码
gateway 192.168.1.1                      # 路由
dns-nameservers 8.8.8.8                #dns
```

（3）重启网络服务

```
sudo systemctl restart networking.service
```

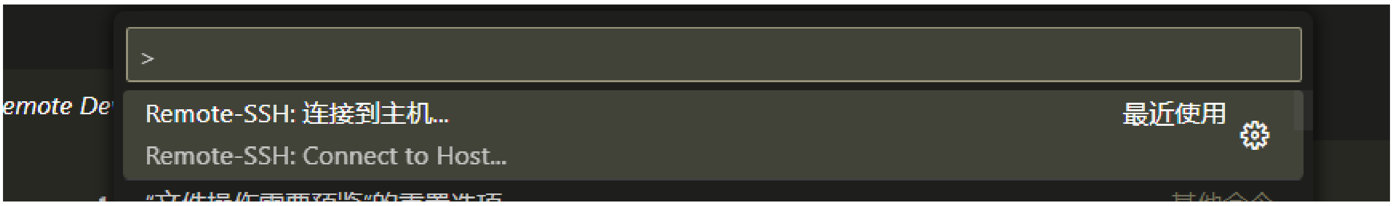
（4）重启机器

```
sudo reboot
```

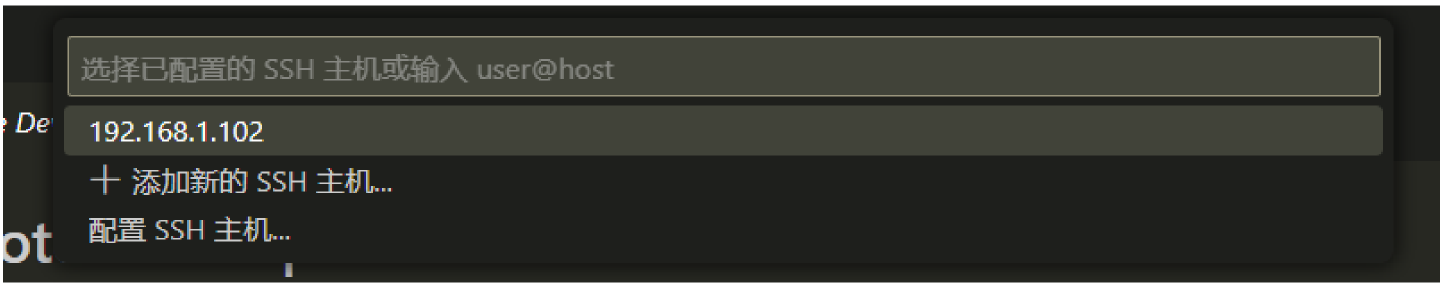
2. 上位机Vscode ssh jetson

（1）安装ssh插件Remote Development、Remote-SSH，直接搜索即可安装。

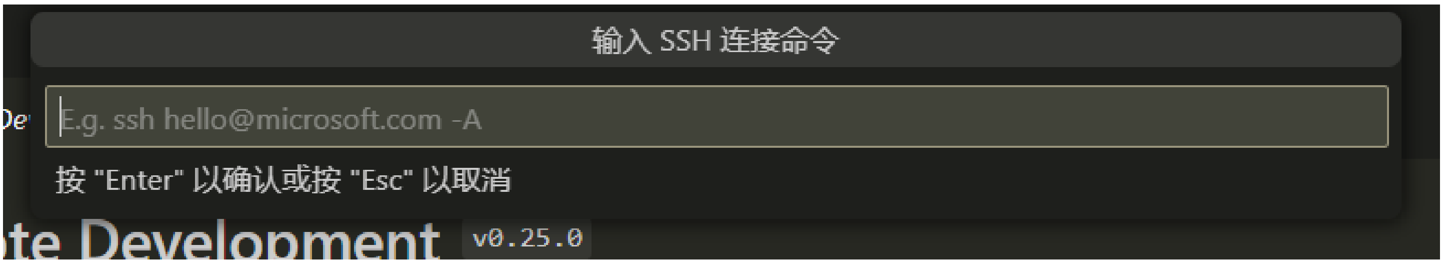
（2）ctrl + shift + P打开命令输入窗口选择Connect to Host



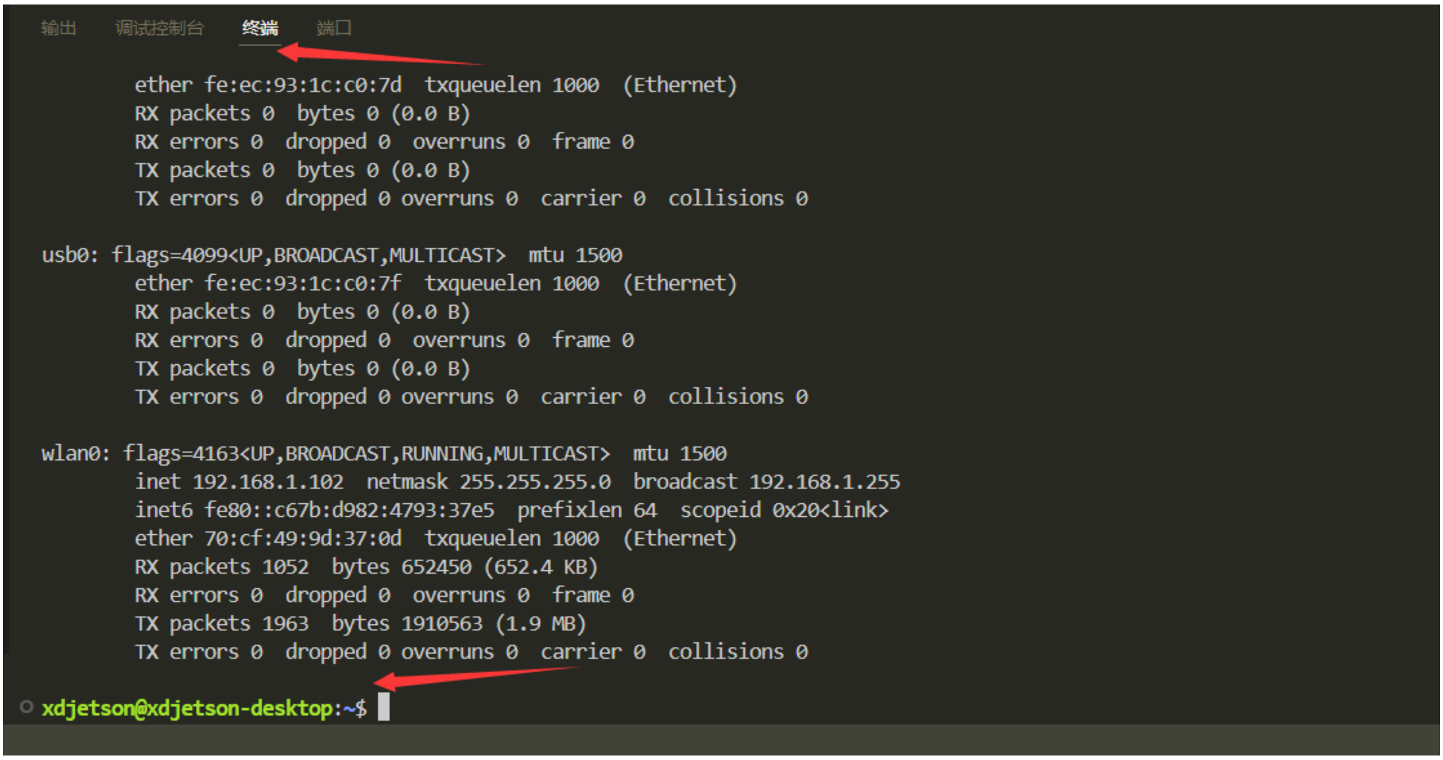
（3）192.168.1.102是已经配置好的，可以之间连接，也可以点添加新的ssh主机。



（4）添加新的SSH主机显示，然后在下面框中输入`ssh xdjetson@192.168.1.102`，然后输入密码123，连接即可。



（5）打开终端查看即可。



三、环境配置

1. 安装Cuda、Cudnn

安装：在刷系统的时候，已经预装了JetPack，所以不用再手动安装，包括了与jetson适配的cuda、cudnn、TensorRT。

cuDNN的头文件在：`/usr/include`，库文件位于：`/usr/lib/aarch64-linux-gnu`，将头文件与库文件复制到cuda目录下：

```
#复制文件到cuda目录下
cd /usr/include && sudo cp cudnn* /usr/local/cuda/include
cd /usr/lib/aarch64-linux-gnu && sudo cp libcudnn* /usr/local/cuda/lib64

#修改文件权限，修改复制完的头文件与库文件的权限，所有用户都可读，可写，可执行：
sudo chmod 777 /usr/local/cuda/include/cudnn.h
sudo chmod 777 /usr/local/cuda/lib64/libcudnn*

#重新软链接，这里的8.4.1和8对应安装的cudnn版本号和首数字
cd /usr/local/cuda/lib64
```

```
sudo ln -sf libcudnn.so.8.4.1 libcudnn.so.8

sudo ln -sf libcudnn_ops_train.so.8.4.1 libcudnn_ops_train.so.8 # 注意版本号
sudo ln -sf libcudnn_ops_infer.so.8.4.1 libcudnn_ops_infer.so.8

sudo ln -sf libcudnn_adv_train.so.8.4.1 libcudnn_adv_train.so.8
sudo ln -sf libcudnn_adv_infer.so.8.4.1 libcudnn_adv_infer.so.8

sudo ln -sf libcudnn_cnn_train.so.8.4.1 libcudnn_cnn_train.so.8
sudo ln -sf libcudnn_cnn_infer.so.8.4.1 libcudnn_cnn_infer.so.8

sudo ldconfig
```

测试cuDNN：

```
sudo cp -r /usr/src/cudnn_samples_v8/ ~/
cd ~/cudnn_samples_v8/mnistCUDNN
sudo chmod 777 ~/cudnn_samples_v8
sudo make clean && sudo make
./mnistCUDNN
```

如果报错，那就运行：

```
sudo apt-get install libfreeimage3 libfreeimage-dev
```

如果配置成功 测试完成后会显示：“Test passed!”。
大功告成！！

2. Pytorch安装

Pytorch需要安装英伟达已经编译好的库文件，链接https://forums.developer.nvidia.com/t/pytorch-for-jetson/72048，根据自己Jetpack版本选择torch 2.1.0。

JetPack 5

▼ PyTorch v2.1.0

- JetPack 5.1 (L4T R35.2.1) / JetPack 5.1.1 (L4T R35.3.1) / JetPack 5.1.2 (L4T R35.4.1)
  - Python 3.8 - `torch-2.1.0a0+41361538.nv23.06-cp38-cp38-linux_aarch64.whl` 5.6k

上位机下载好后再ssh传输到jetson上（推荐这样，因为jetson直接下载会因为cpu、网络问题很慢）：

```
scp -r C:\Users\ZhengLJ\Downloads\torch-2.1.0a0+41361538.nv23.06-cp38-cp38-linux_aarch64.whl xdjetson@192.168.1.102:/home/xdjetson #上位机->jetson
```

传输上去之后，先别着急安装。

3. 安装Anaconda

为了防止环境污染，安装一个Anaconda进行环境隔离，镜像源地址https://repo.anaconda.com/archive/，选择最新的就行，但是一定要选择aarch64架构的：

Anaconda3-2024.02-1-Linux-aarch64.sh 798.5M 2024-02-26 14:50:21 28c5bed6fba84f418516e41640c7937514aabd55e929a8f66937c737303c7bba

还是一样，在上位机下载，然后ssh传输到jetson上，直接安装：

```
sh ./Anaconda3-2024.02-1-Linux-aarch64.sh
```

安装好后查看conda版本号

```
conda init
conda --version
```

可能找不到conda命令，那么运行：

```
# 将anaconda的bin目录加入PATH，根据版本不同，也可能是~/anaconda3/bin
echo 'export PATH="~/anaconda3/bin:$PATH"' >> ~/.bashrc
# 更新bashrc以立即生效
source ~/.bashrc
```

然后进行初始化：

```
conda init
```

如果还是不行，就运行一下：

```
source activate
```

前边出现base就ok了。

然后创建虚拟环境

```
conda create -n xavier_nx python=3.8
conda activate xavier_nx
```

如下图就ok了：

```
(xavier_nx) xdjetson@xdjetson-desktop:~$
```

然后在这个conda环境里安装pytorch就行了：

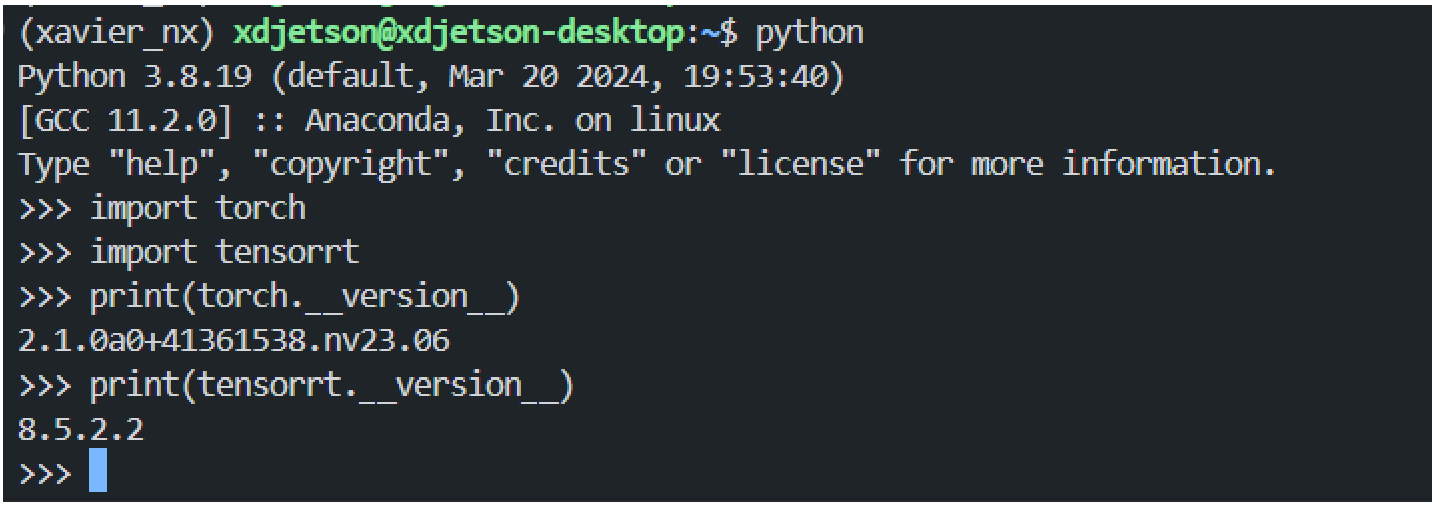
```
pip install torch-2.1.0a0+41361538.nv23.06-cp38-cp38-linux_aarch64.whl
```

4. TensorRT安装

TensorRT已经预装好了，安装在`/usr/lib/python3.8/dist-packages/`，不能被虚拟环境中定位使用，直接把这个目录下的tensorrt包复制到conda环境中：

```
sudo cp /usr/lib/python3.8/dist-packages/tensorrt* /home/xdjetson/anaconda3/envs/xavier_nx/lib/python3.8/site-packages/
```

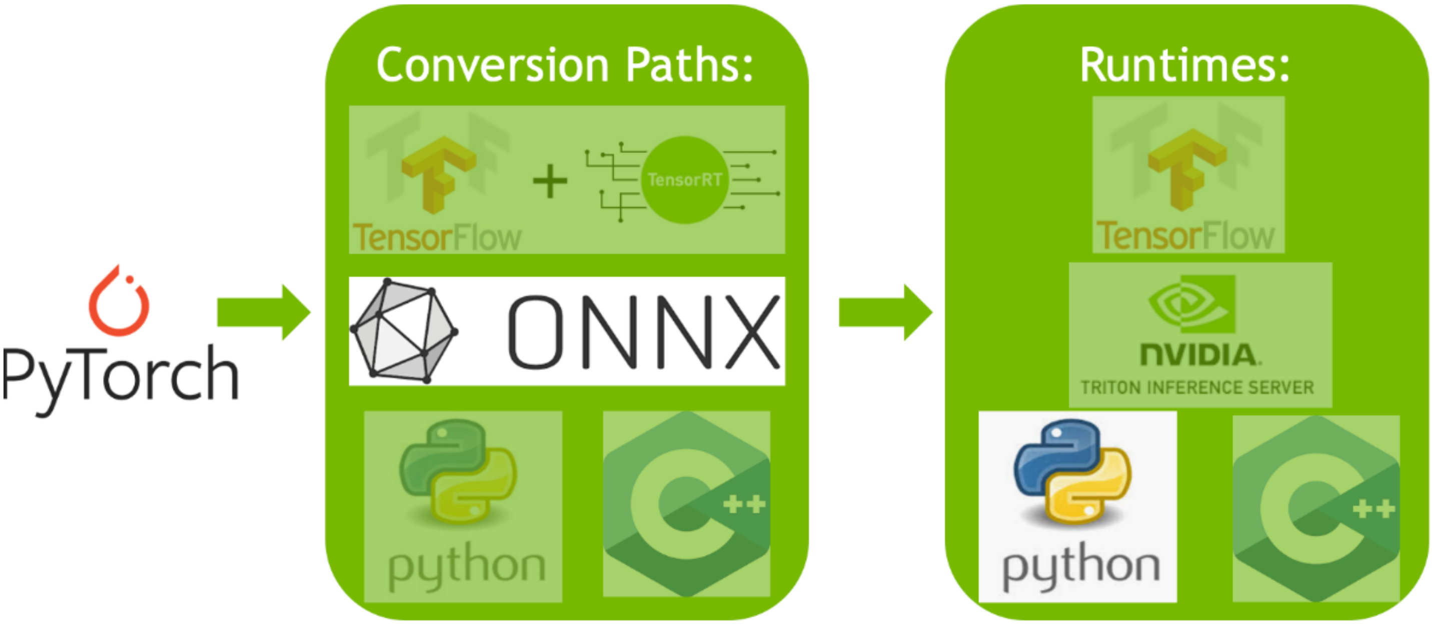
5. 测试环境



```
(xavier_nx) xdjetson@xdjetson-desktop:~$ python
Python 3.8.19 (default, Mar 20 2024, 19:53:40)
[GCC 11.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import tensorrt
>>> print(torch.__version__)
2.1.0a0+41361538.nv23.06
>>> print(tensorrt.__version__)
8.5.2.2
>>>
```

二、Jetson测试

采用下面的流程，PyTorch模型转ONNX，然后使用TensorRT的Python接口。



1. 模型转onnx

在转trt格式之前，需要先转为中间onnx（Open Neural Network Exchange），报错gru_cell算子不支持：

> ONNX：是一种针对机器学习所设计的开放式的文件格式，用于存储训练好的模型。它使得不同的人工智能框架（如Pytorch、MXNet）可以采用相同格式存储模型数据并交互。

```
torch.onnx.errors.UnsupportedOperatorError: Exporting the operator 'aten::_thnn_fused_gru_cell'
to ONNX opset version 11 is not supported. Please feel free to request support or
submit a pull request on PyTorch GitHub: https://github.com/pytorch/pytorch/issues.
```

然后与之前一样选择自定义GRU算子：

```python
class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRUCell, self).__init__()
        stdv = 1.0 / math.sqrt(hidden_size)
        self.weight_ih = nn.Parameter(nn.init.uniform_(torch.Tensor(3 * hidden_size, input_size), -stdv, stdv))
        self.in2hid_w = nn.ParameterList([self.__init(stdv, input_size, hidden_size) for _ in range(3)])
```

```python
        self.hid2hid_w = nn.ParameterList([self.__init(stdv, hidden_size, hidden_size) for _ in range(3)])
        self.in2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])
        self.hid2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])

    @staticmethod
    def __init(stdv, dim1, dim2=None):
        if dim2 is None:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1), -stdv, stdv))  # 按照官方的初始化方法来初始化网络参数
        else:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1, dim2), -stdv, stdv))

    def forward(self, x, hid):
        r = torch.sigmoid(torch.mm(x, self.in2hid_w[0]) + self.in2hid_b[0] + torch.mm(hid, self.hid2hid_w[0]) + self.hid2hid_b[0])
        z = torch.sigmoid(torch.mm(x, self.in2hid_w[1]) + self.in2hid_b[1] + torch.mm(hid, self.hid2hid_w[1]) + self.hid2hid_b[1])
        n = torch.tanh(torch.mm(x, self.in2hid_w[2]) + self.in2hid_b[2] + torch.mul(r, (torch.mm(hid, self.hid2hid_w[2]) + self.hid2hid_b[2])))
        next_hid = torch.mul(-(z - 1), n) + torch.mul(z, hid)

        return next_hid
```

使用torch.onnx.export转为onnx保存为onnx模型，然后加载模型运行**获得onnx模型**，加载运行onnx模型验证精度是否对齐，结果相同：

gpu推理：

```
common infer: (tensor([[ 0.1159, -0.2020]], device='cuda:0', grad_fn=<AddmmBackward0>), tensor([[-1.7703e-01, -2.6950e-01,  1.2665e+00,  1.7609e-01,  1.0000e-02,
         -8.7089e-01, -1.0123e-01, -2.4782e-01,  2.2785e-01,  4.9184e-02,
          6.1418e-01, -5.4979e-01,  1.7475e-01,  8.5624e-02,  3.0154e-01,
         -3.5055e-01, -4.3665e-01, -1.9050e-01,  1.1630e-01, -1.7518e+00,
         -2.1990e-01,  1.0386e-02, -7.2350e-02,  8.1068e-01, -6.3366e-01,
         -6.3710e-02, -6.6545e-04,  5.7642e-01,  1.3725e+00, -2.9656e-02,
         -1.9846e-01, -2.9213e-01,  1.0719e+00,  5.0153e-02, -6.1796e-02,
         -1.8582e-01,  1.2388e+00, -1.0379e-01,  1.0131e+00,  1.0398e-01,
          3.0087e-01,  8.9566e-01,  6.9024e-01,  4.4005e-01,  4.2488e-01,
         -9.4859e-02, -1.0063e-01,  4.9384e-01, -1.4480e+00, -1.2953e-01,
          1.1670e+00,  7.6220e-01, -6.6640e-02,  7.1227e-01,  1.2147e+00,
          1.5363e-01, -3.3052e-01,  7.5002e-02, -7.3689e-01, -2.3123e-01,
          3.9607e-02,  3.9505e-01,  3.6766e-01, -4.3133e-02,  5.6527e-01,
         -2.7537e-01, -7.5519e-01,  2.3891e-01, -3.7081e-01,  1.0241e+00,
          7.6724e-02, -2.3332e-01,  2.1274e-01, -4.2640e-01, -9.4588e-02,
         -1.1159e+00,  1.1392e-01, -1.2194e-01,  2.0320e-01,  3.1223e-01,
         -3.7281e-01,  9.2662e-01, -3.3289e-01,  1.0160e-01,  1.3714e+00,
         -2.9547e-01,  1.3917e-02,  1.6203e-01, -2.4266e-01,  9.0286e-01,
```

onnx：

```
onnx infer: [array([[ 0.11590122, -0.20197046]], dtype=float32), array([[-1.77031666e-01, -2.69504547e-01,  1.26649904e+00,
         1.76085293e-01,  1.00002587e-02, -8.70890796e-01,
        -1.01226568e-01, -2.47823015e-01,  2.27849141e-01,
         4.91842330e-02,  6.14178777e-01, -5.49787879e-01,
         1.74747333e-01,  8.56243819e-02,  3.01541537e-01,
        -3.50549221e-01, -4.36654299e-01, -1.90502837e-01,
         1.16303161e-01, -1.75179088e+00, -2.19895855e-01,
         1.03857145e-02, -7.23501593e-02,  8.10684443e-01,
        -6.33658409e-01, -6.37103841e-02, -6.65394124e-04,
         5.76416671e-01,  1.37254500e+00, -2.96557993e-02,
        -1.98457941e-01, -2.92127848e-01,  1.07186091e+00,
         5.01531884e-02, -6.17957972e-02, -1.85821712e-01,
         1.23882496e+00, -1.03786021e-01,  1.01314926e+00,
         1.03976205e-01,  3.00866723e-01,  8.95660877e-01,
         6.90238357e-01,  4.40046728e-01,  4.24878687e-01,
        -9.48586613e-02, -1.00628942e-01,  4.93841350e-01,
        -1.44795418e+00, -1.29528448e-01,  1.16695976e+00,
         7.62204111e-01, -6.66402876e-02,  7.12266445e-01,
         1.21474481e+00,  1.53625056e-01, -3.30520183e-01,
         7.50017911e-02, -7.36888647e-01, -2.31230155e-01,
```

代码：

```python
import torch
import torch.nn as nn

import time

import math

import onnx
import onnxruntime

import numpy as np

class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRUCell, self).__init__()
        stdv = 1.0 / math.sqrt(hidden_size)
        self.weight_ih = nn.Parameter(nn.init.uniform_(torch.Tensor(3 * hidden_size, input_size), -stdv, stdv))
        self.in2hid_w = nn.ParameterList([self.__init(stdv, input_size, hidden_size) for _ in range(3)])
        self.hid2hid_w = nn.ParameterList([self.__init(stdv, hidden_size, hidden_size) for _ in range(3)])
        self.in2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])
        self.hid2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])

    @staticmethod
    def __init(stdv, dim1, dim2=None):
```

```python
        if dim2 is None:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1), -stdv, stdv))  # 按照官方的初始化方法来初始化网络参数
        else:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1, dim2), -stdv, stdv))

    def forward(self, x, hid):
        r = torch.sigmoid(torch.mm(x, self.in2hid_w[0]) + self.in2hid_b[0] + torch.mm(hid, self.hid2hid_w[0]) + self.hid2hid_b[0])
        z = torch.sigmoid(torch.mm(x, self.in2hid_w[1]) + self.in2hid_b[1] + torch.mm(hid, self.hid2hid_w[1]) + self.hid2hid_b[1])
        n = torch.tanh(torch.mm(x, self.in2hid_w[2]) + self.in2hid_b[2] + torch.mul(r, (torch.mm(hid, self.hid2hid_w[2]) + self.hid2hid_b[2])))
        next_hid = torch.mul(-(z - 1), n) + torch.mul(z, hid)

        return next_hid
class Agents(nn.Module):
    def __init__(self, state_dim, action_dim, n_layers=3, hidden_size=256):
        super(Agents, self).__init__()
        self._n_layers = n_layers
        self._hidden_size = hidden_size

        layers = [nn.Linear(state_dim, self._hidden_size), nn.ReLU()]
        for l in range(self._n_layers - 1):
            layers += [nn.Linear(self._hidden_size, self._hidden_size), nn.ReLU()]
        self.enc = nn.Sequential(*layers)
        self.rnn = GRUCell(self._hidden_size, self._hidden_size)
        self.init_rnn_wb()
        self.f_out = nn.Linear(self._hidden_size, action_dim)

    def init_rnn_wb(self):
        """load initial weights and bias from official torch.nn.grucell"""
        net = torch.nn.GRUCell(self._hidden_size, self._hidden_size)
        p = self.rnn.state_dict()
        p['in2hid_w.0'] = net.state_dict()['weight_ih'][0:self._hidden_size, :].transpose(0, 1)
        p['in2hid_w.1'] = net.state_dict()['weight_ih'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
        p['in2hid_w.2'] = net.state_dict()['weight_ih'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

        p['hid2hid_w.0'] = net.state_dict()['weight_hh'][0:self._hidden_size, :].transpose(0, 1)
        p['hid2hid_w.1'] = net.state_dict()['weight_hh'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
```

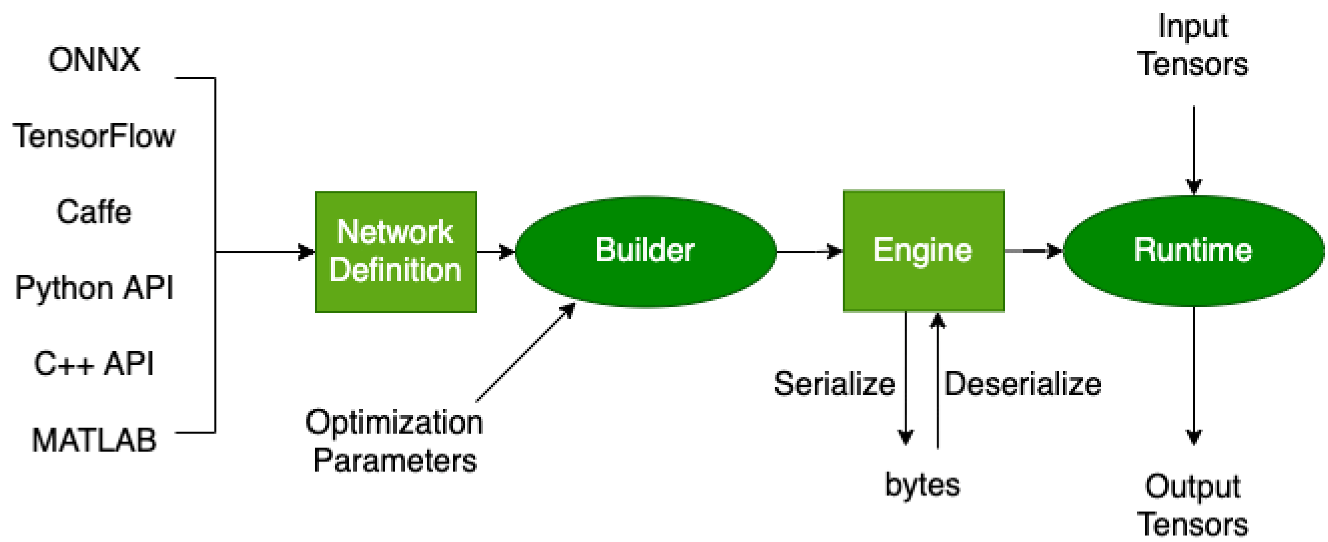
```python
        p['hid2hid_w.2'] = net.state_dict()['weight_hh'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

        p['in2hid_b.0'] = net.state_dict()['bias_ih'][0:self._hidden_size]
        p['in2hid_b.1'] = net.state_dict()['bias_ih'][self._hidden_size:self._hidden_size*2]
        p['in2hid_b.2'] = net.state_dict()['bias_ih'][self._hidden_size*2:self._hidden_size*3]

        p['hid2hid_b.0'] = net.state_dict()['bias_hh'][0:self._hidden_size]
        p['hid2hid_b.1'] = net.state_dict()['bias_hh'][self._hidden_size:self._hidden_size*2]
        p['hid2hid_b.2'] = net.state_dict()['bias_hh'][self._hidden_size*2:self._hidden_size*3]
        self.rnn.load_state_dict(p)

    def forward(self, obs, h):
        x = self.enc(obs)
        h = self.rnn(x, h)
        x = self.f_out(h)

        return x, h

if __name__ == '__main__':
    torch.manual_seed(10)
    torch.cuda.manual_seed(10)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    agents = Agents(state_dim=128, action_dim=5, n_layers=3, hidden_size=256).to(device)
    agents = agents.eval()
    h = torch.randn(1, 256).to(device)
    x = torch.randn(1, 128).to(device)
    print("common infer:", agents(x, h))
    torch.save(agents.state_dict(), 'agent.pth')
    agents.load_state_dict(torch.load('agent.pth'))
    # agents.load_state_dict(torch.load('agent_s.pth'))
    input_tensor = (x, h)
    with torch.no_grad():
        torch.onnx.export(agents,
                          input_tensor,   # model 进行forward的时候的输入,输入的必须是torch.tensor类型
                          'agents.onnx',
                          opset_version=11,
                          input_names=['input_x', 'input_h'],
                          output_names=['output_x', 'output_h'])
    print('onnx model saved successfully...')
    print('begin check onnx model...')
```

```python
    onnx_model = onnx.load('agents.onnx')
    try:
        onnx.checker.check_model(onnx_model)
    except Exception as e:
        print('model incorrect')
        print(e)
    else:
        print('model correct')
    onnx_model = onnxruntime.InferenceSession('agents.onnx')
    output_name = []
    for node in onnx_model.get_outputs():
        output_name.append(node.name)
    print(output_name)
    xx = x.cpu().numpy()
    hh = h.cpu().numpy()
    onnx_outputs = onnx_model.run(output_name, input_feed={'input_x': xx, 'input_h': hh})
    print("onnx infer:", onnx_outputs)
```

2. onnx转trt做推理

TensorRT框架中的几个对象：

- **engine**：TensorRT将模型转换为优化的表示形式，称为**engine**。
- **context**：为**engine**建立推理上下文，所有的推理操作都要在推理上下文下进行。
- **builder**：构建阶段的最高级别接口，用于产生一个**engine**。
- **network**：网络设置。
- **parser**：解析onnx模型的工具。
- **config**：指定一些模型的设置。



整个过程分为如下两个阶段：

（1）Build构建阶段**（离线构建engine）**：

In order to build an engine, you must:

- Create a network definition.
- Specify a configuration for the builder.
- Call the builder to create the engine.

> `NetworkDefinition` 接口：用于定义模型。
>
> `BuilderConfig` 接口：用于指定TensorRT可以如何构建优化模型，比如可以控制**量化**的精度、内存等等。

在官方文档中给的例程中，函数可以直接拿来用，在例程yolov3_onnx的onnx_to_tensorrt.py中，有get_engine的函数，把**network**的设置一改就可以了，可以直接调用该函数从onnx模型构建**trt engine**：

> Tips：jetson输入一定要用float32，不然会报错pycuda._driver.LogicError: cuMemcpyHtoDAsync failed: invalid argument

```
def get_engine(onnx_file_path, engine_file_path=""):
    def build_engine():
        """Takes an ONNX file and creates a TensorRT engine to run inference with"""
        with trt.Builder(TRT_LOGGER) as builder, builder.create_network(
            common.EXPLICIT_BATCH
        ) as network, builder.create_builder_config() as config, trt.OnnxParser(
            network, TRT_LOGGER
        ) as parser, trt.Runtime(
            TRT_LOGGER
        ) as runtime:
            config.max_workspace_size = 1 << 50  # 256MiB
            builder.max_batch_size = 1
            # Parse model file
            if not os.path.exists(onnx_file_path):
                print(
                    "ONNX file {} not found, please run yolov3_to_onnx.py first to generate it.".format(onnx_file_path)
                )
                exit(0)
            print("Loading ONNX file from path {}...".format(onnx_file_path))
            with open(onnx_file_path, "rb") as model:
                print("Beginning ONNX file parsing")
                if not parser.parse(model.read()):
```

```python
                    print("ERROR: Failed to parse the ONNX file.")
                    for error in range(parser.num_errors):
                        print(parser.get_error(error))
                    return None
            # The actual yolov3.onnx is generated with batch size 64. Reshape input to batch size 1
            network.get_input(0).shape = [1, 128]
            network.get_input(1).shape = [1, 256]
            # print('network:', network.get_input(0))
            print("Completed parsing of ONNX file")
            print("Building an engine from file {}; this may take a while...".format(onnx_file_path))
            plan = builder.build_serialized_network(network, config)
            engine = runtime.deserialize_cuda_engine(plan)
            print("Completed creating Engine")
            with open(engine_file_path, "wb") as f:
                f.write(plan)
            return engine

    if os.path.exists(engine_file_path):
        # If a serialized engine exists, use it instead of building an engine.
        print("Reading engine from file {}".format(engine_file_path))
        with open(engine_file_path, "rb") as f, trt.Runtime(TRT_LOGGER) as runtime:
            return runtime.deserialize_cuda_engine(f.read())
    else:
        return build_engine()
```

（2）Runtime运行时阶段

TensorRT执行阶段的最高接口 `Runtime`：

When using the runtime, you will typically carry out the following steps:

- Deserialize a plan to create an engine.
- Create an execution context from the engine.

Then, repeatedly:

- Populate input buffers for inference.
- Call `enqueueV3()` on the execution context to run inference.

> `Engine`：代表一个最优化模型，可以通过此接口向查询有关网络输入和输出张量的信息-预期维度、数据类型、数据格式等。
>
> `ExecutionContext`：此接口用于依据engin创建执行上下文，用于推理。

有了**engine**就可以构建推理上下文**context**，在**context**需要先分配cpu和gpu的memory，分配内存的代码也在例程 `common.py` 中给出，可以直接引入这段代码所在的位置 `/usr/src/tensorrt/samples/python` ，然后调用即可，它的代码如下：

```
def allocate_buffers(engine):
    inputs = []
    outputs = []
    bindings = []
    stream = cuda.Stream()
    for binding in engine:
        size = trt.volume(engine.get_binding_shape(binding)) * engine.max_batch_size
        dtype = trt.nptype(engine.get_binding_dtype(binding))
        # Allocate host and device buffers
        host_mem = cuda.pagelocked_empty(size, dtype)
        device_mem = cuda.mem_alloc(host_mem.nbytes)
        # Append the device buffer to device bindings.
        bindings.append(int(device_mem))
        # Append to the appropriate list.
        if engine.binding_is_input(binding):
            inputs.append(HostDeviceMem(host_mem, device_mem))
        else:
            outputs.append(HostDeviceMem(host_mem, device_mem))
    return inputs, outputs, bindings, stream
```

然后做推理就ok了，也是使用 `common.py` 中的函数，都实现好了，没有特殊需求就可以直接调用，它的实现方式如下：

```
def do_inference(context, bindings, inputs, outputs, stream, batch_size=1):
    # Transfer input data to the GPU.
    [cuda.memcpy_htod_async(inp.device, inp.host, stream) for inp in inputs]
    # Run inference.
    context.execute_async(batch_size=batch_size, bindings=bindings, stream_handle=stream.handle)
    # Transfer predictions back from the GPU.
    [cuda.memcpy_dtoh_async(out.host, out.device, stream) for out in outputs]
    # Synchronize the stream
    stream.synchronize()
    # Return only the host outputs.
    return [out.host for out in outputs]
```

3. 结果比较

- Jetson cuda

```
The inference time of the PyTorch model is: 3 ms
common infer: (tensor([[ 0.2059, -0.0958, -0.0129, -0.3443, -0.0290, -0.6213, -0.2600, -0.1156,
         0.3153, -0.1652]], device='cuda:0', grad_fn=<AddmmBackward0>), tensor([[ 1.9928e-01,
```

- Jetson cuda **onnx**

```
The inference time of the onnx model is: 0 ms
onnx infer: [array([[ 0.20587015, -0.09576682, -0.01288297, -0.34427917, -0.02902389,
        -0.6213429 , -0.26000756, -0.11556078,  0.3152666 , -0.16521229]],
      dtype=float32), array([[ 1.99277610e-01,  8.54276597e-01,  6.24616146e-01,
```

- Jetson cuda **TensorRT**

```
The inference time of the model is: 2 ms
```

```
[ 0.20587006 -0.09576679 -0.01288302 -0.34427923 -0.02902385 -0.6213431
 -0.26000753 -0.11556077  0.31526655 -0.1652123 ]
```

- 服务器CUDA

```
common infer: (tensor([[ 0.2059, -0.0958, -0.0129, -0.3443, -0.0290, -0.6213, -0.2600, -0.1156,
         0.3153, -0.1652]], device='cuda:0', grad_fn=<AddmmBackward0>), tensor([[ 1.9928e-01,
```

代码：

```python
import numpy as np
import tensorrt as trt

import os, sys

sys.path.append('/usr/src/tensorrt/samples/python')

import common

import time

TRT_LOGGER = trt.Logger()

def get_engine(onnx_file_path, engine_file_path=""):
    def build_engine():
        """Takes an ONNX file and creates a TensorRT engine to run inference with"""
        with trt.Builder(TRT_LOGGER) as builder, builder.create_network(
            common.EXPLICIT_BATCH
        ) as network, builder.create_builder_config() as config, trt.OnnxParser(
```

```
            network, TRT_LOGGER
        ) as parser, trt.Runtime(
            TRT_LOGGER
        ) as runtime:
            config.max_workspace_size = 1 << 50  # 256MiB
            builder.max_batch_size = 1
            # Parse model file
            if not os.path.exists(onnx_file_path):
                print(
                    "ONNX file {} not found, please run yolov3_to_onnx.py first to generate it.".format(onnx_file_path)
                )
                exit(0)
            print("Loading ONNX file from path {}...".format(onnx_file_path))
            with open(onnx_file_path, "rb") as model:
                print("Beginning ONNX file parsing")
                if not parser.parse(model.read()):
                    print("ERROR: Failed to parse the ONNX file.")
                    for error in range(parser.num_errors):
                        print(parser.get_error(error))
                    return None
            # The actual yolov3.onnx is generated with batch size 64. Reshape input to batch size 1
            network.get_input(0).shape = [1, 128]
            network.get_input(1).shape = [1, 256]
            # print('network:', network.get_input(0))
            print("Completed parsing of ONNX file")
            print("Building an engine from file {}; this may take a while...".format(onnx_file_path))
            plan = builder.build_serialized_network(network, config)
            engine = runtime.deserialize_cuda_engine(plan)
            print("Completed creating Engine")
            with open(engine_file_path, "wb") as f:
                f.write(plan)
            return engine

    if os.path.exists(engine_file_path):
        # If a serialized engine exists, use it instead of building an engine.
        print("Reading engine from file {}".format(engine_file_path))
        with open(engine_file_path, "rb") as f, trt.Runtime(TRT_LOGGER) as runtime:
            return runtime.deserialize_cuda_engine(f.read())
    else:
        return build_engine()

def rand_Data(len):
    data = np.ones((1, 128)).astype(np.float32)
```

```python

    return data

if __name__ == '__main__':
    onnx_file_path = 'agents.onnx'
    engine_file_path = "agents.trt"
    with get_engine(onnx_file_path, engine_file_path) as engine, engine.create_execution_context() as context:
        inputs, outputs, bindings, stream = common.allocate_buffers(engine)
        # print(inputs[0].host)
        # print(type(inputs[0].host))
        inputs[0].host = np.ones((1, 128)).astype(np.float32)
        inputs[1].host = np.ones((1, 256)).astype(np.float32)
        # print(inputs[0].host)
        # print(type(inputs[0].host))
        # Do inference
        start_time = time.time()
        trt_outputs = common.do_inference(
            context,
            bindings=bindings,
            inputs=inputs,
            outputs=outputs,
            stream=stream,
        )
        end_time = time.time()
        inference_time = int((end_time - start_time) * 1000)
        print("The inference time of the model is:", inference_time, "ms")
        print(outputs[0].host)
        print(outputs[1].host)
    # print(type(inputs))
    # print(len(inputs))
    # print(len(inputs[0]))
```