Team Members:

MUKKAVALLI BHARAT CHANDRA - ES19BTECH11016
SAMAR GARG - CS19BTECH11033

OPTIMISING INFERENCE TIME OF MODELS USING **TensorRT**

Introduction

TensorRT is a library developed by NVIDIA to optimize the inference time on NVIDIA's GPUs. TensorRT provides a fast, modular, robust, reliable inference engine that can support the inference needs within the deployment architecture.

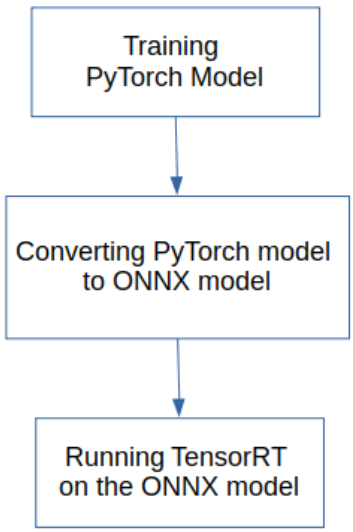
In this project, TensorRT has been used to optimize the inference time on a PyTorch model.

Requirements

In this project, we have used the Jetson Nano device to run TensorRT. It is important to note that TensorRT can be run only on NVIDIA GPU devices. We have used the TensorRT 8.0 version. Other important packages that were used in this project are PyTorch 1.8.0, PyCUDA 2021.1.

Pipeline

The following pipeline has been followed to optimize inference time on a PyTorch Model:



The first step is to train the PyTorch model.
After training the PyTorch model it needs to be converted to the ONNX model. This is important because ONNX makes it easy to access hardware-level optimizations that decrease the inference time. The final step is to parse the ONNX file and run the TensorRT engine.

Model

The below is the description of the model that has been used in the project:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 146, 146]           4,864
         MaxPool2d-2           [-1, 64, 73, 73]               0
       BatchNorm2d-3           [-1, 64, 73, 73]             128
           Dropout-4           [-1, 64, 73, 73]               0
            Conv2d-5          [-1, 128, 71, 71]          73,856
         MaxPool2d-6          [-1, 128, 35, 35]               0
       BatchNorm2d-7          [-1, 128, 35, 35]             256
           Dropout-8          [-1, 128, 35, 35]               0
            Conv2d-9          [-1, 256, 35, 35]          33,024
        MaxPool2d-10          [-1, 256, 17, 17]               0
      BatchNorm2d-11          [-1, 256, 17, 17]             512
           Conv2d-12          [-1, 256, 17, 17]          65,792
        MaxPool2d-13            [-1, 256, 8, 8]               0
      BatchNorm2d-14            [-1, 256, 8, 8]             512
          Dropout-15            [-1, 256, 8, 8]               0
           Linear-16                  [-1, 512]       8,389,120
      BatchNorm1d-17                  [-1, 512]           1,024
          Dropout-18                  [-1, 512]               0
           Linear-19                    [-1, 2]           1,026
================================================================
Total params: 8,570,114
Trainable params: 8,570,114
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.26
Forward/backward pass size (MB): 31.20
Params size (MB): 32.69
Estimated Total Size (MB): 64.15
----------------------------------------------------------------
```

The model consists of 4 Convolutional layers, each Convolutional layer is followed by a MaxPool layer, a BatchNormalisation layer, and a Dropout layer. The Convolutional layers are followed by two Linear layers. The final Linear layer

provides the output of the model. The input to the model is an RGB image of dimensions (150, 150).

The model performs a classification task. The model classifies an image as Occupied and Empty, based on the objects present in the image.

Work-Flow

In this project, we have been provided with the trained PyTorch model. The PyTorch model is stored in the “parkingpytorchmodel.pt” file.

The next step was to convert the PyTorch model to the ONNX model. This can be done using the “torch.onnx.export()” method present in the PyTorch library. This is done in the “pytorch_to_onnx.py” file. To check if the conversion to the ONNX model is successful, the “netron” tool can be used to visualize the neural network present in the model file. The onnx model is stored in the “parkingpytorchmodel.onnx” file.

After getting the .onnx file, the next step is to generate the .trt file. The “trtexec” tool has been used to build the TensorRT engine and run inference. The trtexec tool has options to specify the mode used to build the TensorRT engine. We have used the “fp16” mode to build the engine. The fp16 mode is useful since it reduces the memory, i.e, the size of the tensors is reduced to half and the time taken for inference will also decrease because the computations on the GPU are sped up. The TensorRT engine can be built by running the “get_tensorrt.py” file. The resulting TensorRT engine is stored in the “parkingpytorchmodel.trt” file. It is important to note that the path specified for “trtexec” in the code file is specific to the system.

The next step is to run inference using the TensorRT engine. First, the TensorRT engine has to be read, since it is used to allocate buffers for the input data, output data, bindings (basically the device memory), and the stream (this stream is used to send the inputs to the device and receive the output from the device). The input test images after preprocessing, are transferred to the GPU where the inference is performed and the prediction is returned back from the GPU. The inference can be run by running the file “tensorrt_inference.py”.

Inference and Results

The inference is performed on batches of 91 images each.

The average time taken per batch for inference with TensorRT and without TensorRT are as follows: