Reducing inference time

Akin Wilson

Value proposition

Currently rely on TF serving framework to deploy models

Advantages

Ease of use. Once a model has been *signed,* required model files just need to be placed into the correct directory of a TFserving image.

Disadvantages

Not the most inference-optimized framework; some latency incurred

No fine-grained control about GPU utilisation and so on.

Currently requiring to prop up second serving node to handle volume for just UK verification service. Likely be the same resource requirements for all other territories.

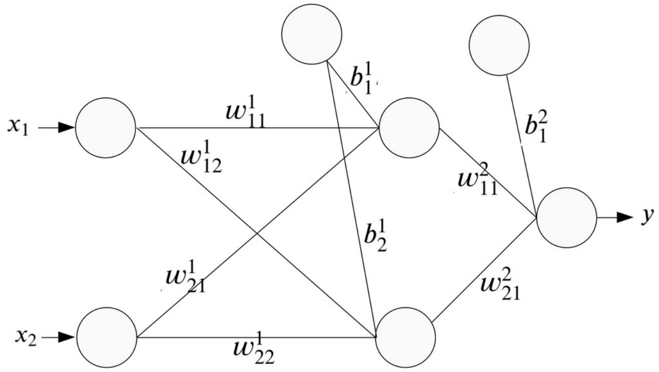
Weights, states and the DL model

A deep learning model is built using a dataset, an architecture and an optimizer

Weights, states and the DL model

A deep learning model is built using a dataset, an architecture and an optimizer

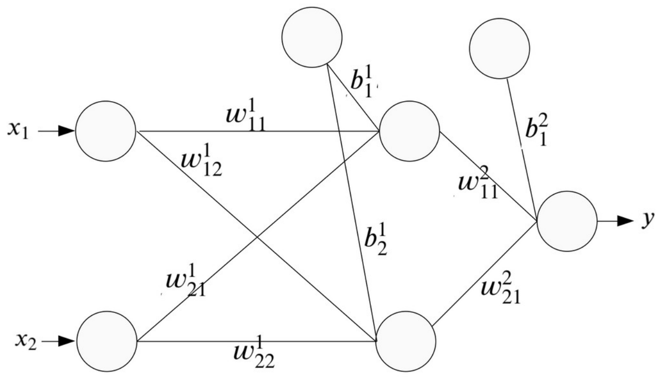
Architecture: Archetypal dense neural network



Weights, states and the DL model

A deep learning model is built using a dataset, an architecture and an optimizer

Architecture: Archetypal dense neural network



Dataset: The XOR gate relations

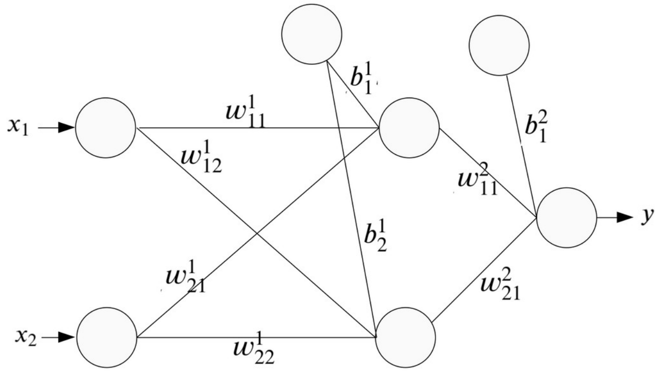
$$x^T = \left[x^1 x^2\right]$$

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, y_{xor} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Weights, states and the DL model

A deep learning model is built using a dataset, an architecture and an optimizer

Architecture: Archetypal dense neural network



Dataset: The XOR gate relations

$$x^T = \left[x^1 x^2\right]$$

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, y_{xor} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

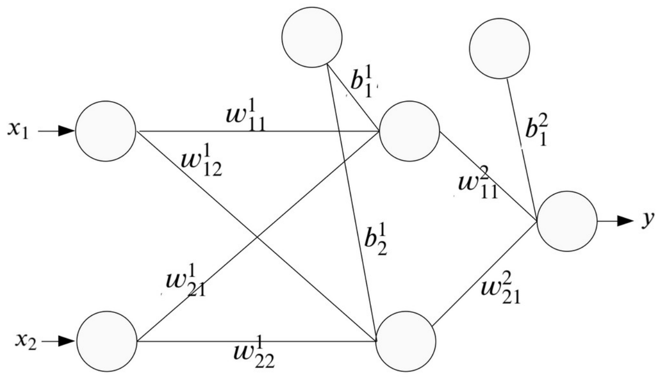
Optimizer: The Adaptive moment estimator

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t}$$
$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t}$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta\frac{\hat{m}_w}{\sqrt{\hat{v}_w}+\epsilon}$$

Weights, states and the DL model

A deep learning model is built using a dataset, an architecture and an optimizer

Architecture: Archetypal dense neural network



Dataset: The XOR gate relations

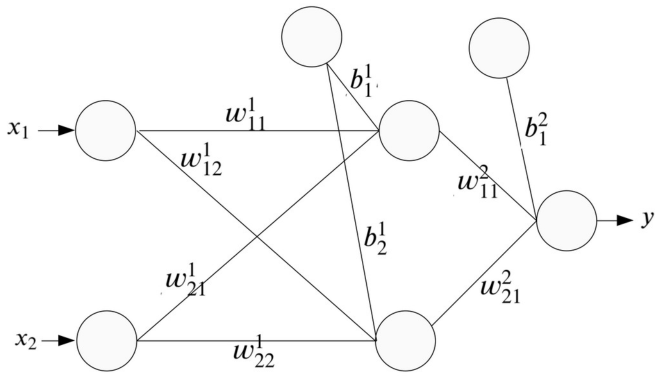
$$x^T = [x^1 x^2]$$

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, y_{xor} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

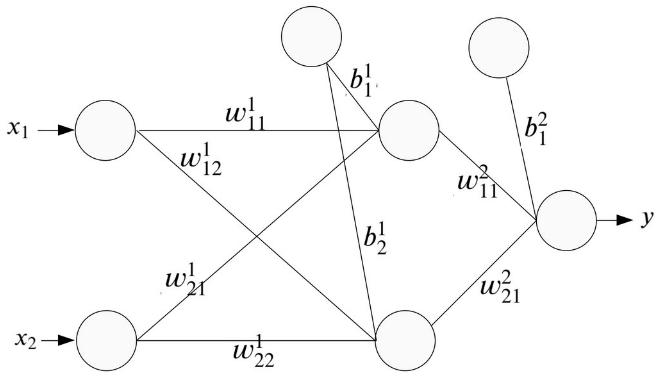
Optimizer: The Adaptive moment estimator

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t}$$
$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t}$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

The weights at a closer look



The weights at a closer look



$$w^1 = \begin{bmatrix} w^1_{11} & w^1_{12} \\ w^1_{21} & w^1_{22} \end{bmatrix} b^1 = \begin{bmatrix} b^1_{11} \\ b^1_{21} \end{bmatrix} x = \begin{bmatrix} x^1 \\ x^2 \end{bmatrix}$$

$$x^T = [x^1, x^2]$$

$$h_1 = activation(x^T w^1 + b^1)$$

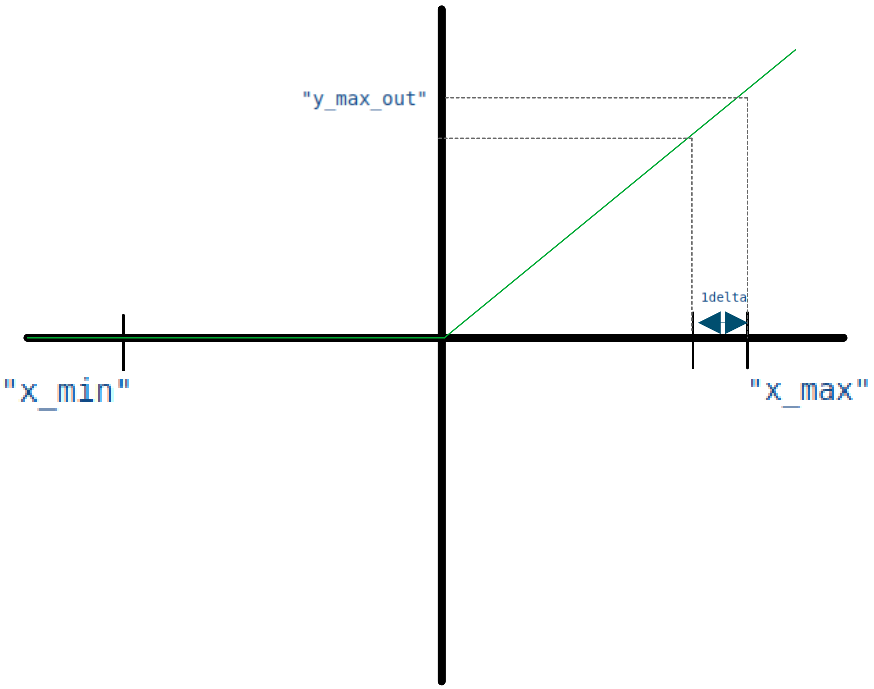
$$h_1 = activation\left(\begin{bmatrix} x^1 w^1_{11} + x^2 w^1_{21} + b^1_{11} \\ x^1 w^1_{12} + x^2 w^1_{22} + b^1_{21} \end{bmatrix} \right)$$

$$w^2 = \begin{bmatrix} w^2_{11} \\ w^2_{21} \end{bmatrix} b^2 = [b^2_1]$$

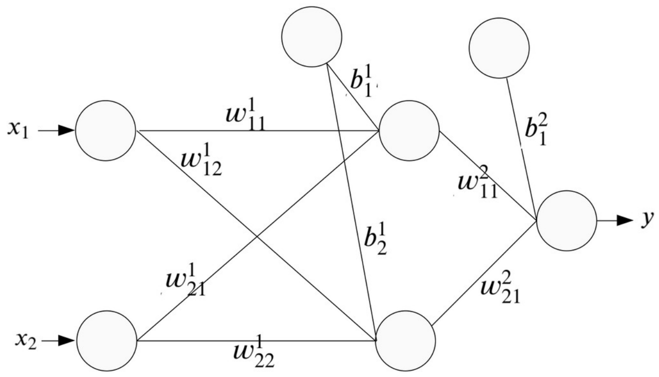
$$y = activation(h_1^T w^2 + b^2)$$

Interlude: quantisation of the activation function

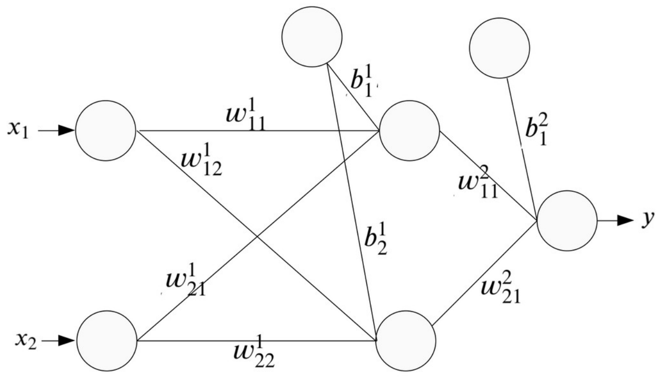
```
{
    "x_min": "y_min_out",
    "x_min + 1delta": "y_min+1_out",
    "x_min + 2delta": "y_min+2_out",
    ...,
    ...,
    ...,
    "x_max - 2delta": "y_min- 2_out",
    "x_max - 1delta": "y_max -1_out",
    "x_max" : "y_max_out"
}
```



The weights at a closer look

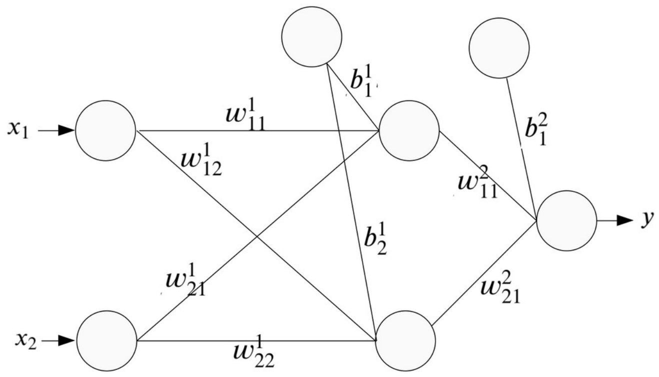


The weights at a closer look



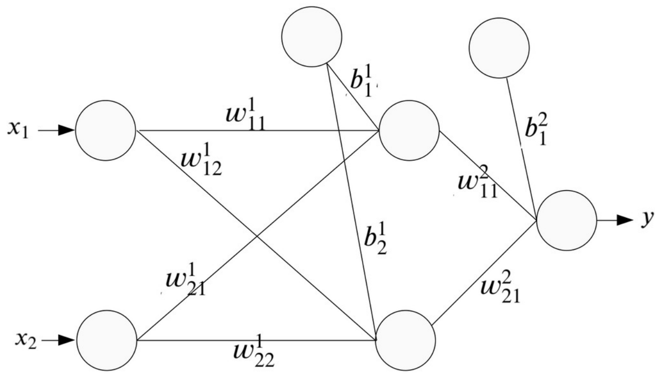

```
{
    "Layer1": {
        "weights": [
            ["w^1_11","w^1_12"],
            ["w^1_21","w^1_22"]
            ],
        "bias": [
                ["b^1_1"],
                ["b^1_2"]
            ],
    "Layer2": {
        "weights": [
            ["w^2_11"],
            ["w^2_21"]
            ],
        "bias": [
                ["b^2_1"],
            ],
    }
}
```

The weights at a closer look



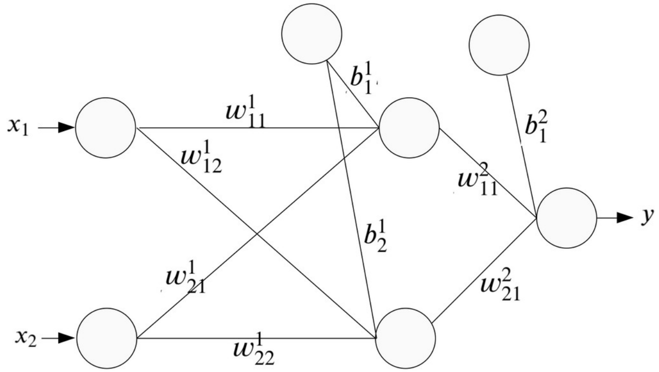

```
[
    {
        "weights": [

            ["w^1_11","w^1_12"],
            ["w^1_21","w^1_22"]
            ],
        "bias": [
            ["b^1_1"],
            ["b^1_2"]]},
    {
        "weights": [
            ["w^2_11"],
            ["w^2_21"]
                    ],
        "bias": [
            ["b^2_1"],
                    ],
    }
]
```

The weights at a closer look




```
[
    [
        ["w^1_11","w^1_12", "b^1_1"],
        ["w^1_21","w^1_22", "b^1_2"]
        ],
        [
            ["w^2_11"],
            ["w^2_21"],
            ["b^2_1"]
    ]
]
```

The weights at a closer look




```
{
    "Layer1": {
        "weights": [
            ["w^1_11","w^1_12"],
            ["w^1_21","w^1_22"]
            ],
        "bias": [
                ["b^1_1"],
                ["b^1_2"]
            ],
    "Layer2": {
        "weights": [
            ["w^2_11"],
            ["w^2_21"]
            ],
        "bias": [
                ["b^2_1"],
            ],
    }
}
```

```
[
    {
        "weights": [

            ["w^1_11","w^1_12"],
            ["w^1_21","w^1_22"]
            ],
        "bias": [
            ["b^1_1"],
            ["b^1_2"]},
    {
        "weights": [
            ["w^2_11"],
            ["w^2_21"]
                    ],
        "bias": [
            ["b^2_1"],
                    ],
    }
]
```

```
[
    [
        ["w^1_11","w^1_12", "b^1_1"],
        ["w^1_21","w^1_22", "b^1_2"]
        ],
        [
            ["w^2_11"],
            ["w^2_21"],
            ["b^2_1"]
    ]
]
```

What is Onnx and the Onnx runtime?

Onnx stands for Open Neural Network Exchange

Founded by Microsoft and Facebook in 2017, the goal was to establish a open standard to save ML and DL models.

The initiative is now supported by IBM, Huawei, Intel, AMD, Arm and Qualcomm.

Onnx today provides post-training quantisation and a suit of other deployment optimization tools

Due to being the standard the industry is converging to, it also allows you, for example, to convert a model to the Tflite format.

Onnx runtime is a part of the Onnx ecosystem.

It provides a means for a model exported via onnx to be deployed and queried

Alleviates the need for framework dependencies. No TF or PT dependencies required

Allows for deployment in low-resource environments, i.e. microcontrollers. The runtime itself is light-weight.

Can use a variety of *execution providers* for the runtime to use.

What is an execution provider?

An **execution provider** is the **hardware and software combination** used to compute the network's outputs given an input.

For example, currently we use the execution provider of **cuda** (the **software**) to compute the matrix multiplications that make up our models. Cuda makes use of the **GPU** to do so (the **hardware**).

When using the **CPU** for executions (the **hardware**) the underlying **software** library being used to perform calculations is called **MLAS**.

Introducing execution provider: TensorRT

Built by those who build the GPUs themselves (Nvidia), TensorRT is an SDK for high-performance deep learning inference, including a deep learning inference optimizer and runtime that delivers low latency and high throughput for inference applications.

The hardware used is still the GPU, but the sotfware is now TensorRT.

How can we combine these?

The nitty gritty: The training pipeline

```
lr_monitor = LearningRateMonitor(logging_interval='epoch')
early_stopping = EarlyStopping(mode="min", monitor='val_loss', patience=25)
checkpoint_callback = ModelCheckpoint(monitor="val_loss",
                                        dirpath=data_path.model_dir,
                                        save_top_k=1,
                                        mode="min",
                                        filename='{epoch}-{val_loss:.2f}-{val_acc:.2f}-{val_ttr:.2f}-{val_ftr:.2f}')

model = ResNet(block=Bottleneck, num_blocks=[8, 8, 36, 3], cfg=cfg)
callbacks = [checkpoint_callback, lr_monitor, early_stopping]

logger = TensorBoardLogger(save_dir=data_path.model_dir, version=1, name="lightning_logs")

trainer = Trainer(accelerator="gpu",
                  devices=3,
                  strategy='dp',
                  logger = logger,
                  default_root_dir=data_path.model_dir,
                  callbacks=callbacks)
trainer.fit(Routine(model, cfg), train_dataloaders=train_loader, val_dataloaders=val_loader)

trainer.test(dataloaders=test_loader)
```

Train, validate and test model; Resnet

```
from wwv.util import OnnxExporter
model = trainer.model.module.module.model
predictor = Predictor(model)
OnnxExporter( model=predictor,
             cfg=cfg,
             output_dir=data_path.model_dir)()
```

Wrapping model with predictor head and exporting it to onnx

The nitty gritty: Exporting to onnx

```python
class OnnxExporter:

    def __init__(self, model, cfg, output_dir):

        self.cfg = cfg
        self.model_name = cfg.model_name
        self.model = model
        self.output_dir =output_dir
        self.model.eval()
        assert not self.model.training, "Model not in inference mode before exporting to onnx format"
        # Input to the model
        batch_size = 1
        # Get expected input dims from config cfg.processing_output_shape = (40, 241)
        self.x_in = torch.randn(batch_size, 1, 40, 241, device="cpu")

        logger.info(f"Input for model tracing: {self.x_in.shape}")
        self.x_out = self.model(self.x_in)
        logger.info(f"Output given input for model tracing: {self.x_out.shape}")
        self.onnx_model_path=None


    def verify(self):
        model = onnx.load(self.onnx_model_path)
        onnx.checker.check_model(model)


    def to_numpy(self,tensor):
        return tensor.detach().cpu().numpy() if tensor.requires_grad else tensor.cpu().numpy()


    # Export the model
    def __call__(self):
        print("self.output_dir", self.output_dir)
        output_path = self.output_dir + "/model.onnx"
        print("self.output_path", output_path)

        logger.info(f"Onnx model output path: {output_path}")

        model = self.model
        x_dummy =  self.x_in
        torch.onnx.export(model=model,                          # model being run
                          args=x_dummy,                         # model input (or a tuple for multiple inputs)
                          f=output_path,                        # where to save the model (can be a file or file-like object)
                          export_params=True,                   # store the trained parameter weights inside the model file
                          opset_version=15,                     # Only certain operations are available, 17 includes FFTs and IFFTs
                          do_constant_folding=True,             # whether to execute constant folding for optimization
                          input_names = ['input_mfcc', 'dummy_input'],  # the model's input names
                          output_names = ['output_wwp'],        # the model's output names
                          dynamic_axes={'input_mfcc' : {0 : 'batch_size'},  # variable length axes
                                        'output_wwp' : {0 : 'batch_size'}})
        self.onnx_model_path = output_path
        self.verify()
        self.inference_session()
        return self
```

Creating dummy inputs for *tracing* and *scripting.*

Verification function checks output of onnx model and pytorch model match

Export to onnx using *scripting* and *tracing* functionaltiy of torch

Verify the exported model.

The nitty gritty: The deployment environment

```
FROM nvcr.io/nvidia/tensorrt:22.08-py3
LABEL maintainer="Akinola Antony Wilson <akinola.wilson@sky.com>"

# Allow passing in decision threshold and model version during build of serving container.
ARG DECISION_THRESHOLD=0.5
ARG MODEL_VERSION="docker-env-model-version"
ARG EXECUTION_PROVIDER="TensorrtExecutionProvider"
# Setting decision_threshold and model_version and env vars
ENV DECISION_THRESHOLD=${DECISION_THRESHOLD}
ENV MODEL_VERSION=${MODEL_VERSION}
# setting environment variable specifying execution provider, can be: CUDAExecutionProvider, CPUExecutionProvider or TensorrtExecutionProvider

ENV EXECUTION_PROVIDER=${EXECUTION_PROVIDER}
# install utilities
RUN apt-get update && \
    apt-get install --no-install-recommends -y curl
# audio processing dependencies
RUN DEBIAN_FRONTEND=noninteractive TZ=Etc/UTC apt-get -y install tzdata
RUN apt-get install -y libsndfile-dev
# Install python
RUN apt-get install -y python3
RUN apt-get install -y python3-pip
# install protobuf dependencies
RUN apt-get install protobuf-compiler libprotobuf-dev -y


# Installing python dependencies
RUN python3 -m pip --no-cache-dir install --upgrade pip && \
    python3 --version && \
    pip3 --version

# install gpu-enabled torch
RUN pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu113


# check https://github.com/onnx/onnx-tensorrt/issues/354#issuecomment-572279735 --->
RUN git clone --recurse-submodules https://github.com/onnx/onnx-tensorrt.git
WORKDIR /workspace/onnx-tensorrt

RUN mkdir build
WORKDIR /workspace/onnx-tensorrt/build

RUN cmake .. -DTENSORRT_ROOT=/workspace/tensorrt
RUN make -j

RUN export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH

WORKDIR /workspace

COPY ./torch-deploy/requirements.txt .
RUN pip3 --timeout=300 --no-cache-dir install -r requirements.txt

# Copy model files
COPY ./torch-deploy/model /model
RUN pip3 --timeout=300 --no-cache-dir install -r requirements.txt
# Copy app files
COPY ./torch-deploy/app /app


COPY ./torch-deploy/start.sh /app/start.sh
RUN chmod +x /app/start.sh

WORKDIR /app
EXPOSE 80
ENTRYPOINT ["./start.sh"]
```

Get base image: TensorRT using an ubuntu OS

Install onnx-TensorRT dependencies

Clone the onnx-tensorrt repository, build and configure to use TensorRT

The nitty gritty: Providing the executioner

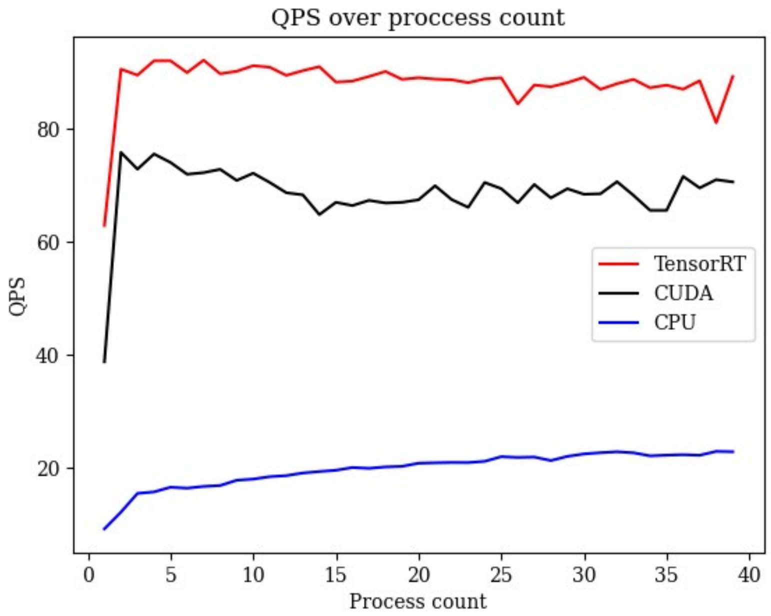
```python
@app.on_event("startup")
async def startup_event():
    """
    Initialize FastAPI and add variables
    """
    logger.info(f"Running envirnoment: {CONFIG['ENV']}")
    logger.info(f"PyTorch using device: {CONFIG['DEVICE']}")

    providers = [os.environ.get("EXECUTION_PROVIDER"),]


    inference_session = onnxruntime.InferenceSession(CONFIG['ONNX_MODEL_PATH'],
                                                     providers=providers)

    # add model and other preprocess tools too app state
    app.package = {
        "torch_model": None,
        "onnx_session": inference_session
    }
```

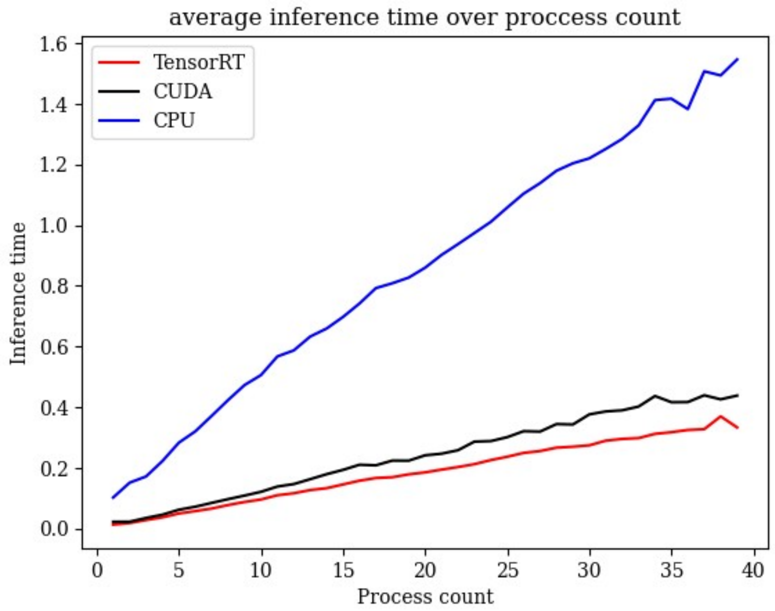
Using micro-service framework FastAPI (like Django or Flask) to allow for querying endpoint

Possible choices of executioners: CPU, GPU or TensorRT
Can add additional inference parameters.

The results: Comparing the execution providers latency under load

The load test pushes the endpoint to its limits, increasing the load and thereby queries per second, until we receive a timeout response.





CPU min latency: 0.1024s
GPU min latency: 0.0224s
TensorRT min latency: 0.0130s