TCN

Implementation

https://github.com/shivadb/TCN

Original TCN: https://github.com/locuslab/TCN

For building the deployment pipeline, we use MNIST data as our toy dataset

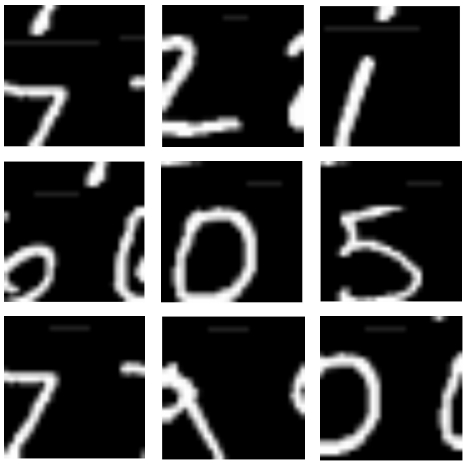
Training

Original Implementation:

- Convert 28x28 image to 1D 784 vector
- Use the final element of the output vector to train the network
- Expects the full 784 vector to perform inference; is not trained to make intermediate predictions

Augmented Implementation:

- Idea is to allow the model to converge to the correct classification as each pixel is provided
- Training images are randomly clipped so the model is trained to classify with missing data
    - Maximum clipped size is set to 1/4 of the model's receptive field. This means the model should be able to converge to the correct classification once it has 3/4 of the data for the current image
    - The clipping is performed at the beginning of the image and the end of another random digit is stitched in place of the missing data to simulate streaming data
- Examples:



Evaluation

Qualitative

Augmented Demo
Original Demo

Quantitative

Original Implementation:

- The original TCN implementation evaluates accuracy by comparing the output produced after processing all 784 pixels
- The paper reports a test accuracy of 99.0% on MNIST data

- When the augmented model is evaluated using this method, we obtain a test accuracy of 93.0%

Sequential Implementation:

- Our model is designed to output a prediction after processing each pixel. Due to the augmentation performed for training, it makes more sense to consider the last n outputs to compute the final prediction
- In the below table we consider the last n outputs before the full image is seen by the model, and take the highest occurring output as the final prediction for the current digit:

n	Accuracy (%)
1	93.0%
50	96.9%
100	97.9%
150	98.2%
200	98.3%

Inference

Original Implementation:

- The original TCN implementation performs inference using the same module as used for training
    - The input is a 784 long vector

Sequential implementation:

- The fast implementation consists of queues for each convolutional layer
- The network only takes the latest pixel as input and uses past computations stored in each queue to compute the next output for each layer

****Stateful modules are not supported by both ONNX and TensorRT. The model is simple enough that the non-cache implementation gives similar performance to the original implementation. Hence, we use the original implementation to build the deployment pipeline**

Inference Optimization

We explore 3 methods for generating an optimized inference engine for deployment

ONNX Runtime

Pipeline:
PyTorch model -> ONNX model -> ONNX Runtime

ONNX Conversion:

- Torch model is converted to ONNX model using torch.onnx.export
    - This is done via tracing or scripting (or both)
    - Tracing is performed by executing the network forward pass using an example input, and colleting all performed operations
    - Scripting converts nn.module objects to torch.jit scripts, which are then converted to ONNX format

ONNX Runtime:

- Microsoft provides a runtime library that can be used to execute ONNX models
- Available python builds:

    - Cpu backend
    - Gpu (CUDA) backend
    - OpenVINO - Intel managed
    - TensorRT (Jetson) - nvidia managed
- Additionally onnxruntime can be built with various execution providers (such as TensorRT)
    - https://onnxruntime.ai/docs/how-to/build/eps.html

Pros	Cons
- ONNX is widely accepted as a common NN representation - Most of the common torch operations are supported in ONNX - Onnxruntime is easy to use	- Torch model is limited to using operations that are supported by ONNX, and by the torch.onnx module - ONNX does not support stateful modules - Python onnxruntime-gpu environment does not give much control over data transfers

TensorRT

In this approach we use the following Python module released by Nvidia: https://github.com/NVIDIA-AI-IOT/torch2trt

Pipelines:

- Method 1: PyTorch model -> ONNX model -> TensorRT engine (using ONNX parser)
- Method 2: PyTorch model -> TensorRT engine (using torch2trt converters)

Conversion via ONNX

- First it uses torch.onnx.export to generate an onnx graph
- Then it uses tensorrt library's OnnxParser to define the network

Pros	Cons
- ONNX to TensorRT is more developed (and the recommended workflow according to Nvidia) compared to direct mapping offered by torch2trt converters	- Torch model is limited to using operations that are supported by ONNX, and by the torch.onnx module - 2 points of failure

Conversion via torch2trt Converters

- In this approach we can skip an ONNX conversion all together
- Uses converters defined in the torch2trt library that directly map PyTorch functions to equivalent TensorRT API functions

Pros	Cons
- One less point of failure - Ability to write custom converters for layers/operations that are not available in the released codebase	- Fewer operations are supported by the current available converters than using the Torch -> ONNX -> TensorRT workflow

TensorRT basic objects and core concepts can be found here

- The main object created is the engine which can be used to run inference

More information about TensorRT

Basic implementation for using a generated engine in C++ is shown in:
Cpp/main.cpp

Pros	Cons
- C++ API allows a lot of control over data transfer and memory allocation - Provides slightly better performance compared to onnxruntime-gpu	- TensorRT must be supported and installed in the target deployment device

Performance Analysis

Python tests are available in:
Performance Tests.ipynb

Following table summarizes the performance results obtained using the MNIST toy model. All tests are performed on Nvidia GeForce RTX 2080Ti

Model	Python Avg Execution Time (ms)	C++ Avg Execution Time (ms)
PyTorch	3.13	-
ONNX via onnxruntime-gpu	1.21	-
TRT via ONNX - full precision	0.63	0.53
TRT via converters - full precison	0.51	0.41
TRT via ONNX - half precision	0.52	0.41
TRT via converters - half precison	0.52	0.41

Efficient Data Transfer

Notes for data transfer: CUDA Memory Transfer

The TRT-via-converters model is used to perform the following experiments:

Experiment	Full Precision Execution Time (ms)	Half Precision Execution Time (ms)
Regular In / Regular Out	0.42	0.44
Regular In / Zerocopy (cudaHostAllocMapped) Out	0.40	0.42
Pinned In / Regular Out	0.41	0.44
Pinned In / Zerocopy (cudaHostAllocMapped) Out	0.43	0.42
Zerocopy (cudaHostAllocWriteCombined) In / Zerocopy (cudaHostAllocMapped) Out	0.39	0.42

TensorRT API offers two methods of executing the model:

- context->execute - This is a synchronous operation
- context->enqueue - This is used to execute the model asynchronously

TensorRT defines the model input and output in the form of a buffer object

- Input:
  - The input indices are defined when the TensorRT engine is generated. The buffer value at each input index should point to an array of the defined input size (for the particular input)
  - In our model we only have 1 input and hence only have 1 input index. The buffers object at this input index should point to a 784-long array of float values.
- Output:
  - Output indices are also similarly defined during the generation of the TRT engine. The buffer object at each output index should point to the location where the corresponding output should be stored by the engine
  - In our model we have 1 output, so our buffer at output index should point to a location where the engine can write a single integer

Output Buffer

To store a sequence of model outputs, we can define a n-long integer array. During every subsequent execution, we can increment a pointer the next location in the array and set the TensorRT buffers object to point to this address with its output index.

Input and Execution Pipeline

1. Input data is read into pinned memory
2. TRT Buffer is created for each image in 3 steps:
   a. CUDA memory allocation
   b. Data Transfer (pinned memory -> gpu)
   c. Concatenate cuda memory pointer with output (mapped zerocopy memory) pointer
3. Each TRT Buffer is added to a readerwriterqueue (https://github.com/cameron314/readerwriterqueue)
4. Each buffer is used in context->enqueue to process outputs

Multithreading

3 Threads

Objects:

- TRTBuffer: contains a void pointer array of size 2 (as required by the TRT engine)
  - Keeps a record of the input and output indices as defined by the TRT engine
  - Instances of this class are used to perform model execution

- PinnedCopyBuffer: contains the address of a pinned memory allocation (of image size)
  - Instances of this class are used to copy data from regular memory to pinned memory
  - The instances are then used to transfer data to cuda memory
  - Additionally, this object also keeps a record of the sample number

Queues:

- Available Pinned Buffers: Indices of available PinnedCopyBuffer objects for copying from regular memory to pinned memory
  - When a buffer index is in this queue, it means the data in the buffer's pinned memory address can be safely overwritten

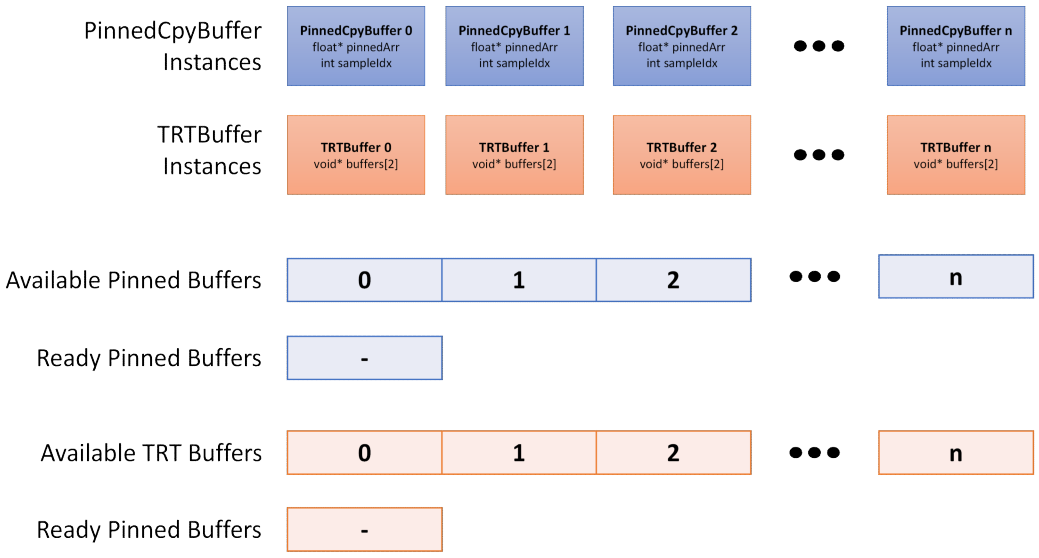
- Ready Pinned Buffers: Indices of PinnedCopyBuffers that contain data to be processed
    - When a buffer index is in this queue, it means there is valid data in the buffer's pinned address that needs to be processed

- Available TRT Buffers: Indices of available TRTBuffer instances for gpu data transfer
    - When a buffer index is in this queue, the data is in the inputIndex void pointer is ready to be overwritten
    - The data in the inputIndex memory location is populated using cudamemcpy
    - The outputIndex address is specified by using the sample number (recorded by PinnedCopyBuffer) as the offset

- Ready TRT Buffers: Indices of TRTBuffer instances with valid data for model execution
    - When a buffer index is in this queue, it is ready to be passed to the TRT engine to be processed

Threads

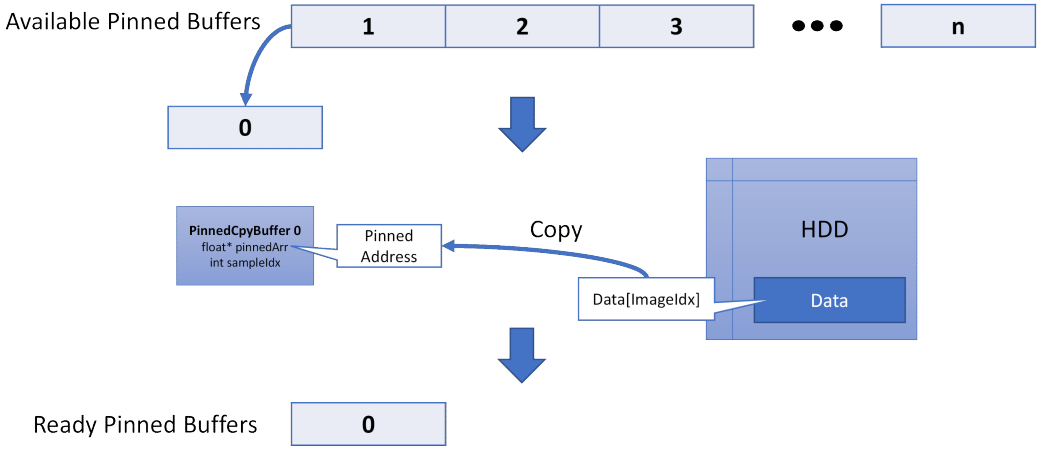
Initialize



Initialize sample number: ImageIdx = 0

HDD-to-Pinned

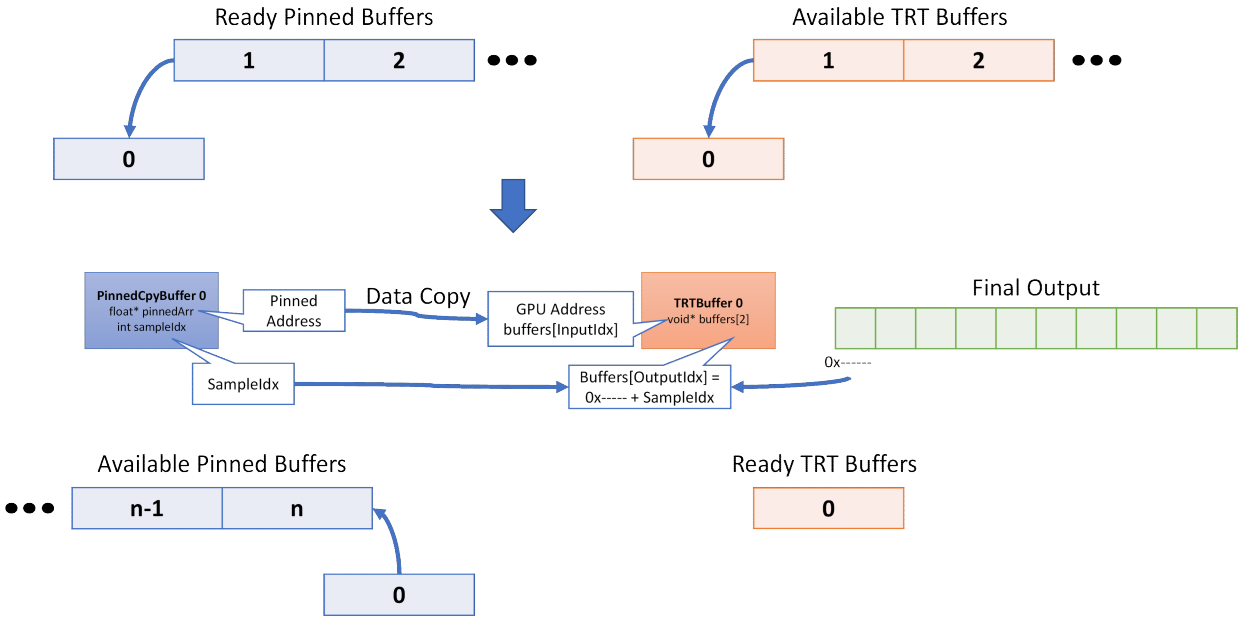
1. Dequeue from Available Pinned Buffers queue:
2. Copy data at ImageIdx in HDD data to pinned memory location from corresponding buffer object
3. Enqueue buffer to Ready Pinned Buffers Queue



4. ImageIdx++

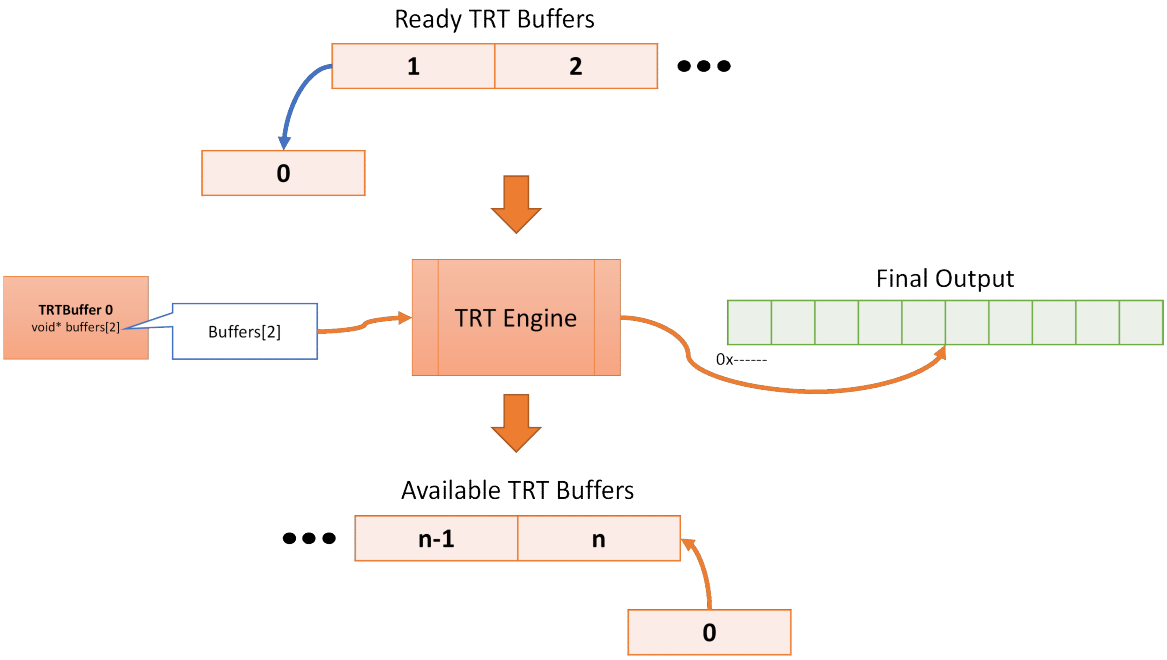
Pinned-to-GPU

1. Dequeue from Ready Pinned Buffer queue
2. Dequeue from Available TRT Buffer queue
3. Copy data from Pinned buffer memory to cuda memory specified by the TRT Buffer
4. Assign the correct final output memory location (using ImageIdx as offset)
5. Once the copy operation is finished:
    a. Enqueue the Pinned Buffer index into the Available Pinned Buffer queue
    b. Enqueue the TRT Buffer index into the Ready TRT Buffer queue



Execute Model

1. Dequeue TRT Buffer index from Ready TRT Buffer queue
2. Enqueue the pointers (input and output) defined in the corresponding TRT Buffer for model execution
3. Once the model execution is complete:
    a. Enqueue the TRT Buffer index into the Available TRT Buffer queue



Performance

HDD to Pinned Mem (# threads)	Pinned Mem to GPU (# threads)	TRT Execution (# threads)	Avg Execution Time (ms)
1	1	1	0.40
1	1	2	0.35
1	2	2	0.37
2	2	2	0.38
4	2	2	0.37
1	1	3	0.43
1	1	4	0.46
8	2	2	0.37

2 Thread Groups

In the 3 thread group approach we saw that increasing the cpu-to-gpu mem transfer threads did not improve performance. In this approach, we define a thread group for copying data from HDD into pinned host memory, and another thread group to copy the input data into GPU memory and execute the model.

HDD to Pinned Mem (# threads)	CUDA Copy and Execution (# threads)	Avg Execution Time (ms)
1	1	0.39
1	2	0.33
2	2	0.33

3	2	0.33
4	2	0.32
6	2	0.32
8	2	0.33
4	3	0.42
8	3	0.40
4	4	0.43
8	4	0.42
16	4	0.42