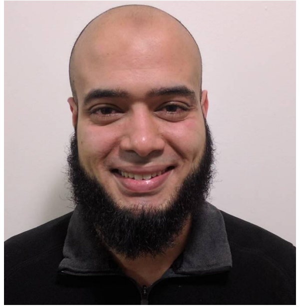
Machine Learning
Boosting Model Inference Speed

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

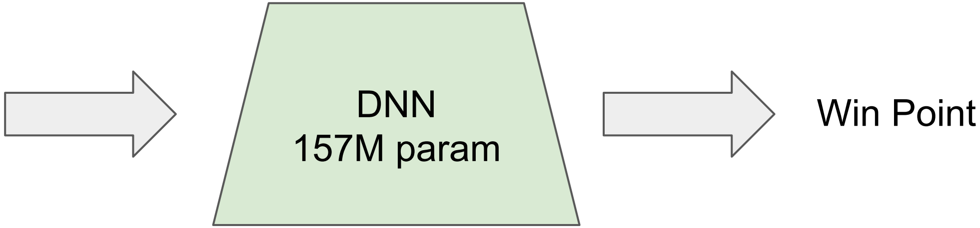


Operations speed

- FLOPS stands for "Floating Point Operations Per Second", a common measure to calculate the number of floating-point calculations
  - GFLOPS (Giga), TFLOPS (trillions), PFLOPS (quadrillions)
- However, still there are many issues with the metric
  - Companies vary on what is an operation (e.g. fusing 2 operations)
  - Apps typically have lower utilization (job’s flops / available flops)
  - Doesn't take into considerations: memory bandwidth, I/O operations / Integer operations
- MLPerf Benchmarks: Nvidia / general

Boosting Model Inference

- Running machine learning models requires **extensive computations and memory**
  - It is cost-saving to have so **fast** models but still with **high** performance
    - Tip: in startups we may skip and validate the idea first (MVP)
  - In real-time apps (e.g. self-driving), you must be extremely fast!
- Imagine we process a live stream for basketball Olympiad
  - Our ML system should detect winning points!

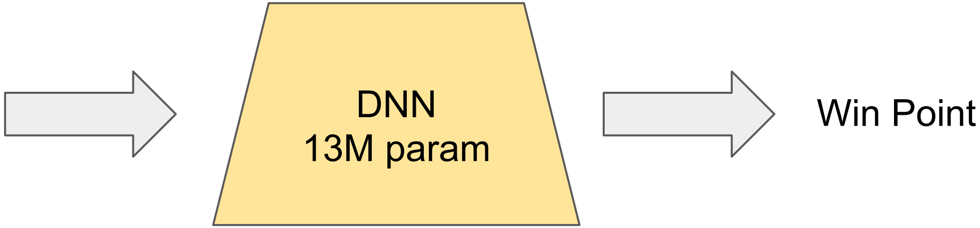


Boosting Model Inference

- There are 2 stages that we do to speed up a model's **inference**!
- 1) Model Compression
  - **First**, think of a simpler network that still has good performance (architecture optimization)
    - Neural Architecture Search (NAS) explores better architectures
    - Convolution **factorization**: 1xN, Nx1, 1x1, depthwise conv (low-rank factorization)
  - **Second**, use Techniques to reduce the model size of the final network
    - Cons: You may **lose** some **accuracy**
- 2) Inference Optimization
  - **Software and hardware** are very specialized to **accelerate** the network inference
- In a real-time app like self-driving, we do all of them!

Model Compression: Simpler Model

- Imagine your first model reads a 3-seconds video at 60 FPS
  - The model requires **157 Million** parameter
- You then realized at 60 FPS, many frames are duplicate
  - then you sampled only 5 frames per second
- You also realized a simpler backbone (like MobileNetV2) works well
  - And with some other tricks, you reduced the network into **13M** parameters!

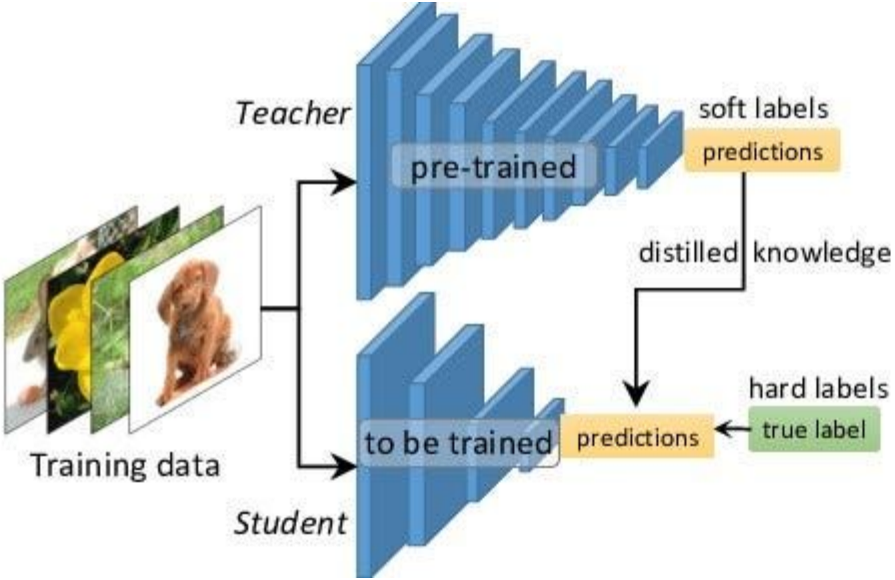


Model Compression: Common Techniques

- Research is so active on approaches to reduce the network size
- Some of these approaches are easily applicable, while others may need more development efforts
- Common techniques
  - **Knowledge Distillation**: a smaller **student** DNN model is trained to transfer the knowledge of a larger **teacher** DNN (nice but requires effort to train the teacher and transfer)
  - **Pruning**: Pruning removes the neurons or connections in the neural network that contribute the least to the model's performance (Filter/Layer Pruning - Sparse/Dynamic Pruning)
    - Also can be applied to some other algorithms
  - **Quantization**: very general/common in industry as easily applicable in many cases
    - E.g. convert network from 32-bit **floats** to 8-bit **integers**.

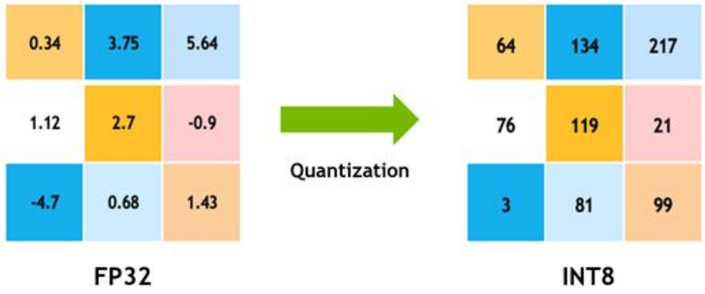
Knowledge Distillation

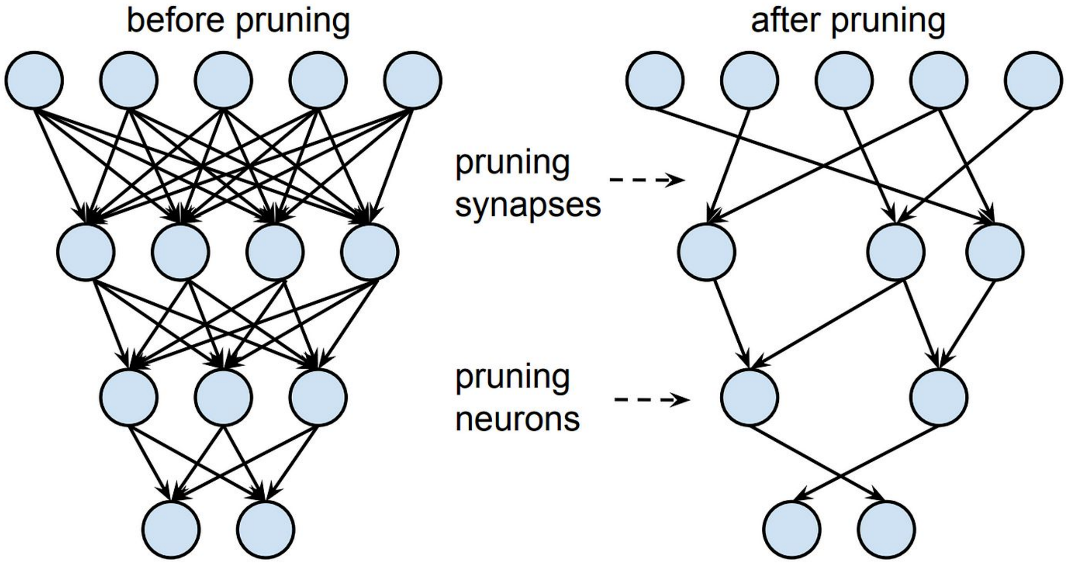
- **Knowledge Distillation**: a smaller **student** DNN model is trained to transfer the knowledge of a larger **teacher** DNN (nice but requires effort to train the teacher and transfer)



Img src

Model Compression: Common Techniques





Img src src

Model Compression: Quantization

- Quantization reduces the precision of the numerical values in the model, converting **32-bit** floats to 16-bit or even **8-bit** integers.
  - Significantly reduce the model size but with a slight degradation in performance.
  - **8-bit** is very common in the industry. Binary quantization is limited
- It is supported by Pytorch and Tensorflow frameworks
  - **Quantization-Aware Training (QAT)**
    - Most common. First, train your normal F32 bit float network
    - Retrain it: Fine-tune it 2-5 epochs with quantization
  - Post-Training Quantization (PTQ)
    - The floating model is quantized after the training (no re-training)
  - Relevant: **Layer fusion**: an optimization technique that combines multiple adjacent layers in a neural network into a single layer

Model Compression: Evaluation

- Given that we have several ways to compress a model, it is good to think in how to criteria to compare
- **Dependency**: Each compression technique exhibits different dependencies on the model's architecture and the target hardware
  - Which technique is DNN agnostic?
  - Which technique is Hardware independent?
- **Speed Gains**
  - Which one of them is faster?
- Tip: nothing prevents us from applying all of them
  - For example, do distillation first to reduce the network
  - Then do pruning to remove useless layers/connections
  - Then Quantize the weights!

Model Compression: Dependency

- **Quantization**
  - Hardware **dependent** (e.g. bits/ram), but **independent** approach for most of DNNs
  - Deeper networks might tolerate aggressive quantization better than shallower ones.
- **Pruning**
  - Different architectures might require specific pruning strategies (**dependent)**
  - **Independent Hardware:** Even with a lot of pruning, CPUs/GPUs are not optimized for **sparse** operations (RAM also degrade performance due to unstructured pruning)
    - Built for dense not sparse operations
  - **Dependent Hardware**: developed to handle sparse computations efficiently
- **Distillation**
  - Student network design is important to learn efficiently / capture the dark knowledge
  - Hardware **independent**. Select a student network that fits with required constraints

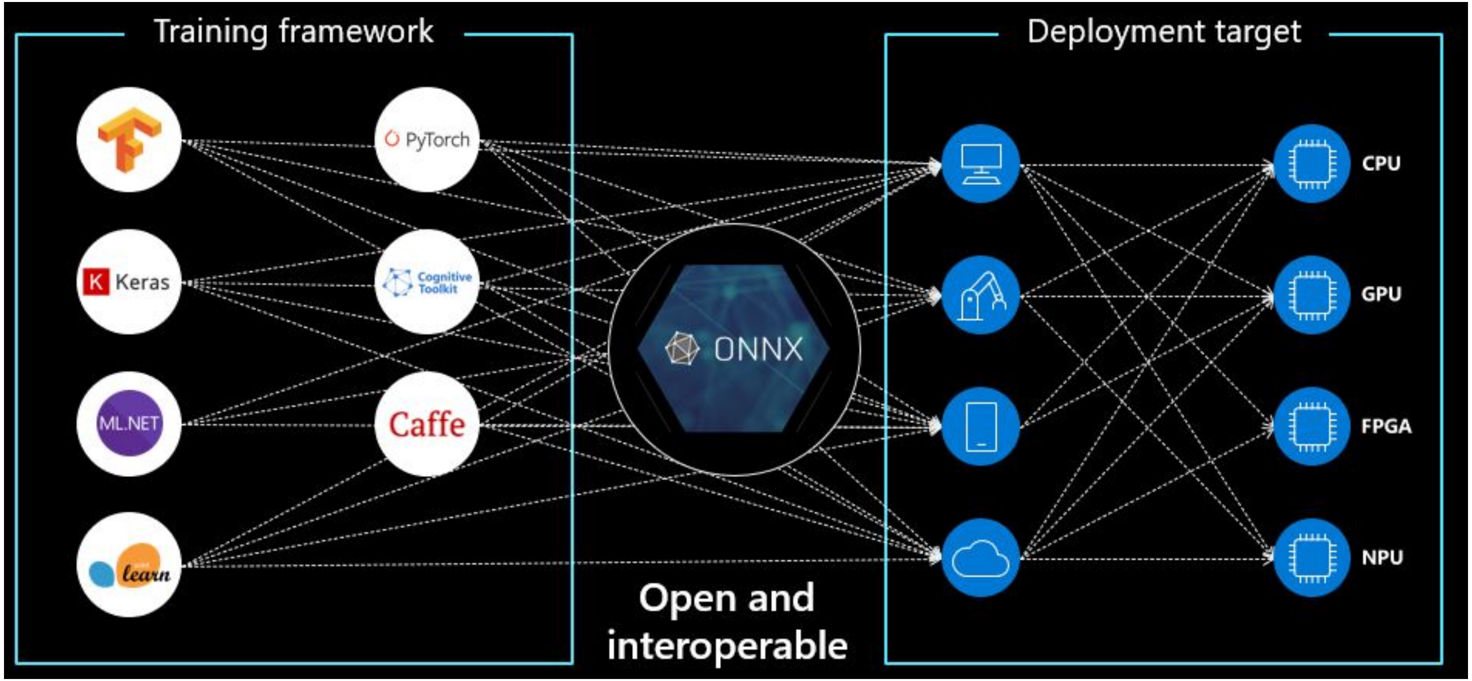
Model Compression: Speed Gains

- **Quantization** can bring 2-4x speed.
  - 8-bits with quantization-aware training is very common choice
  - Lower than will cause performance drop
- **Pruning**
  - The speed gain depends on the 1) sparsity level and the 2) hardware's ability
    - General-purpose hardware might not see a significant speedup
    - Specialized hardware / software may achieve speedups
- **Distillation**
  - The speed gain depends depends on the size of the student models compare to the teacher

Inference Optimization: ONNX Role

- ONNX library convert models from different ML libraries (e.g. PyTorch, TensorFlow) into a unified format (ONNX format) ⇒ **interoperability**
- This way, software/hardware **inference optimization** techniques focus only on optimizing ONNX files toward different deployment targets (e.g., CPU, GPU)
- Challenges
  - Version **Compatibility**
  - Operator Support: Not all layers and operations in every framework are supported
    - We typically change the backbone to use the supported operators
  - Harder to debug the model searching for conversion issues

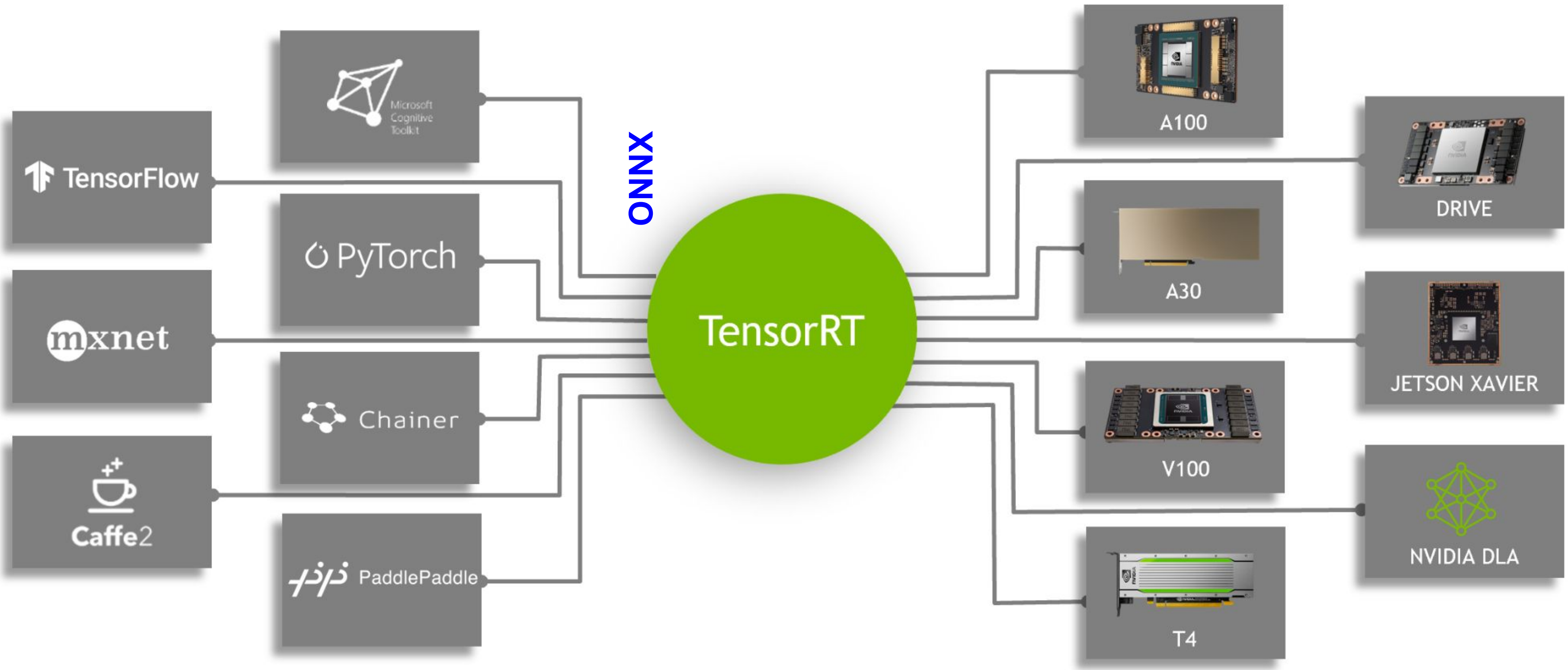
Inference Optimization: ONNX



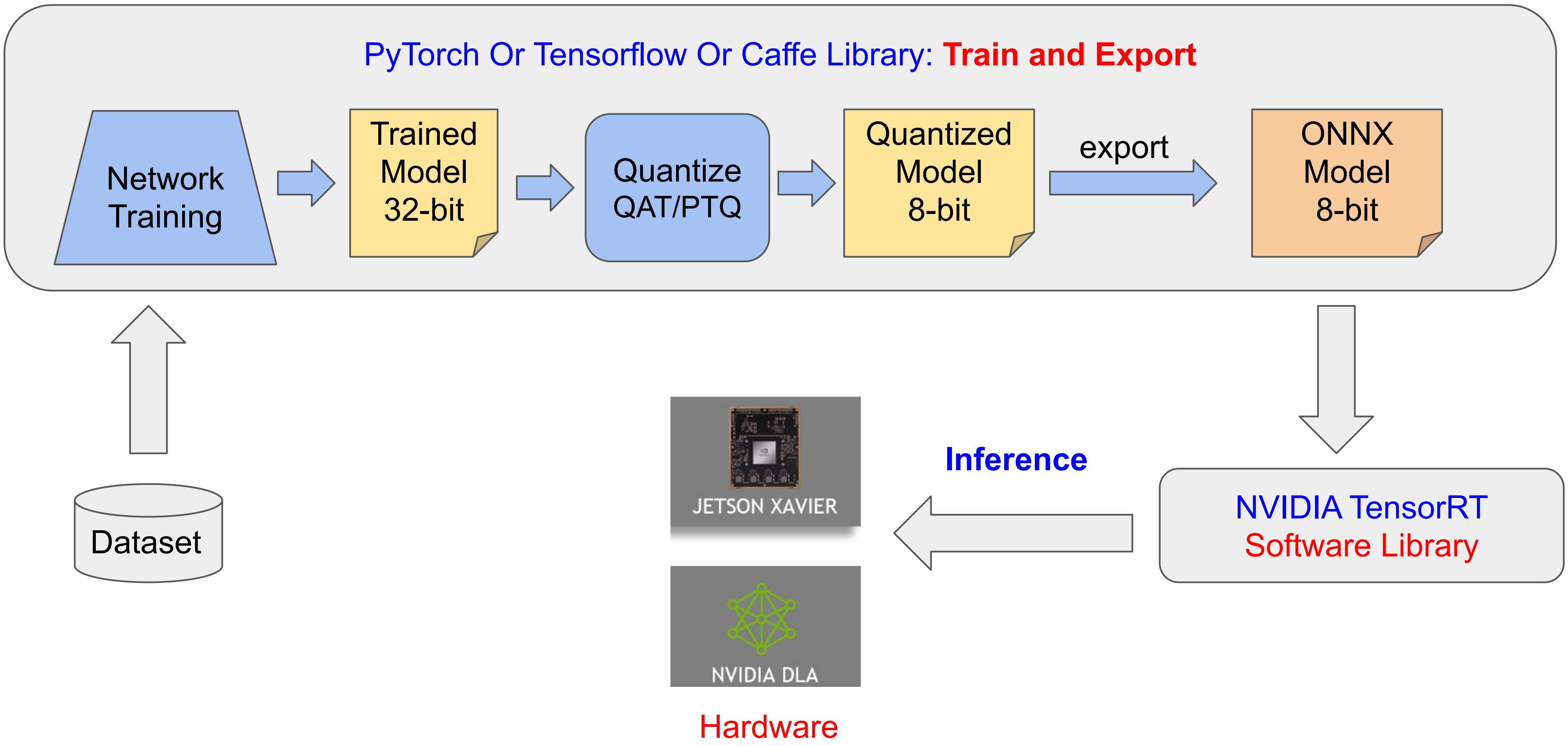
Img src

NVIDIA: Optimized Software and Hardware

- NVIDIA TensorRT is a high-performance deep learning **inference library** for deployment of neural network models. It is designed to work efficiently on **NVIDIA** GPUs and is part of NVIDIA's Deep Learning SDK
    - One major input files are the ONNX files. It optimizes them further
- **[**NVIDIA's**]** **DLA** (Deep Learning Accelerator) refers to **specialized hardware** designed to **accelerate** deep learning tasks, particularly **inference**
    - Primarily used for inference in edge devices like cameras, smartphones, and IoT devices.
    - Optimized for lower latency and **power consumption**
    - Note: GPUs can be used for both **train and inference**

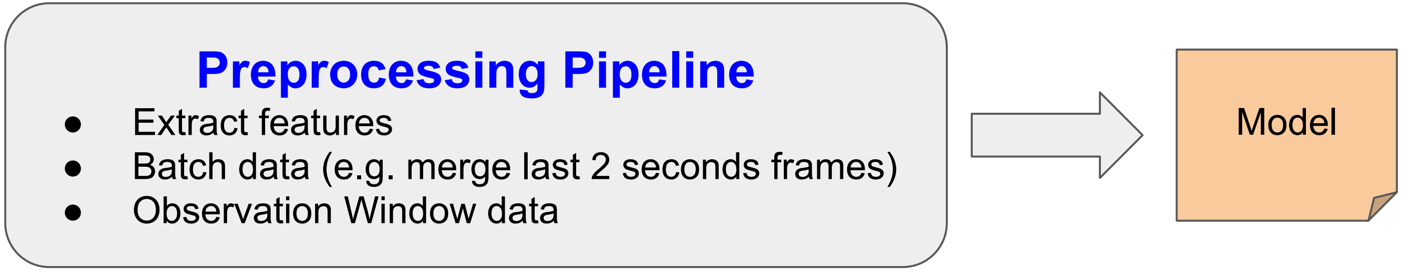


Microsoft Cognitive Toolkit
TensorFlow
PyTorch
mxnet
Chainer
Caffe2
PaddlePaddle
ONNX
TensorRT
A100
DRIVE
A30
JETSON XAVIER
V100
NVIDIA DLA
T4


PyTorch Or Tensorflow Or Caffe Library: Train and Export
Network Training
Trained Model 32-bit
Quantize QAT/PTQ
Quantized Model 8-bit
export
ONNX Model 8-bit
Dataset
JETSON XAVIER
NVIDIA DLA
Inference
NVIDIA TensorRT
Software Library
Hardware

From Python (Development) to C++ (Deployment)

- Nowadays, we typically do DNN **training** using python with frameworks such as PyTorch and Tensorflow
- However, in several applications we run the **inference** on a C++ pipeline to speed up (e.g. in embedded systems, real-time like self-driving)
- One of the common **mistakes** is failing to **sync code** changes between **training python** pipeline and **inference C++** pipeline
  - Wrong Code Logic *(could be 2 teams responsibilities)*
  - Changes in configurations (e.g. temporal window length)



Offline Inference

- Sometimes our model just runs offline, which allows different things
  - For example, google process offline your images and extract information
- We can boost the performance in 2 ways:
  - Model Ensemble (average results of multiple models)
  - Test Time Augmentation (TTA): Average results of multiple argumentations
    - In real inference, we just go with the input as it is

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”