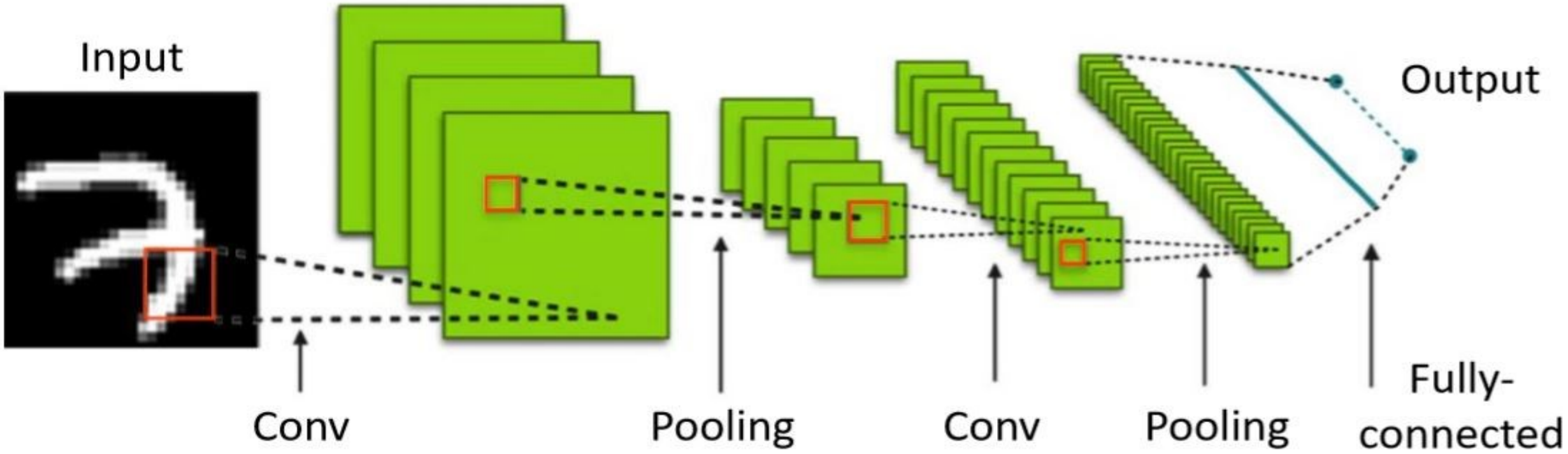
Improve inference on edge devices using TensorRT and TFLite

- Ashwin Phadke

Who am I?

- Normal human being (likes Pikachu, why not?).
- Programming since 5+ years (contiguous arrays , ah!).
- Experience in deep learning and computer vision of more than 2+ years.
- Worked at Cynapto - a upcoming leading tech startup.
- Consulting funded startups in the field of artificial intelligence.
- Electronics and Telecomm engineer (Boy, was it a rocky ride).

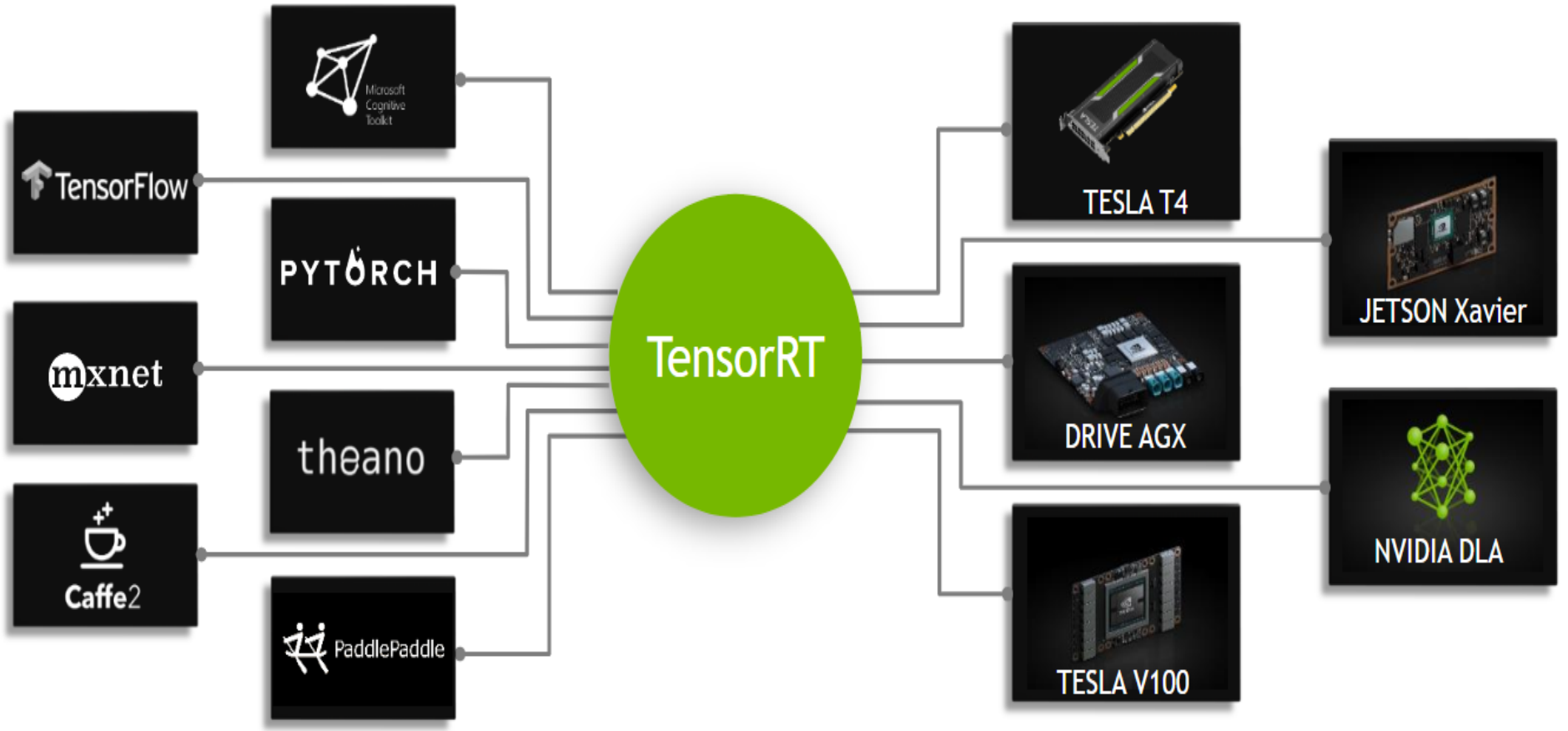
Convolutional Neural Network



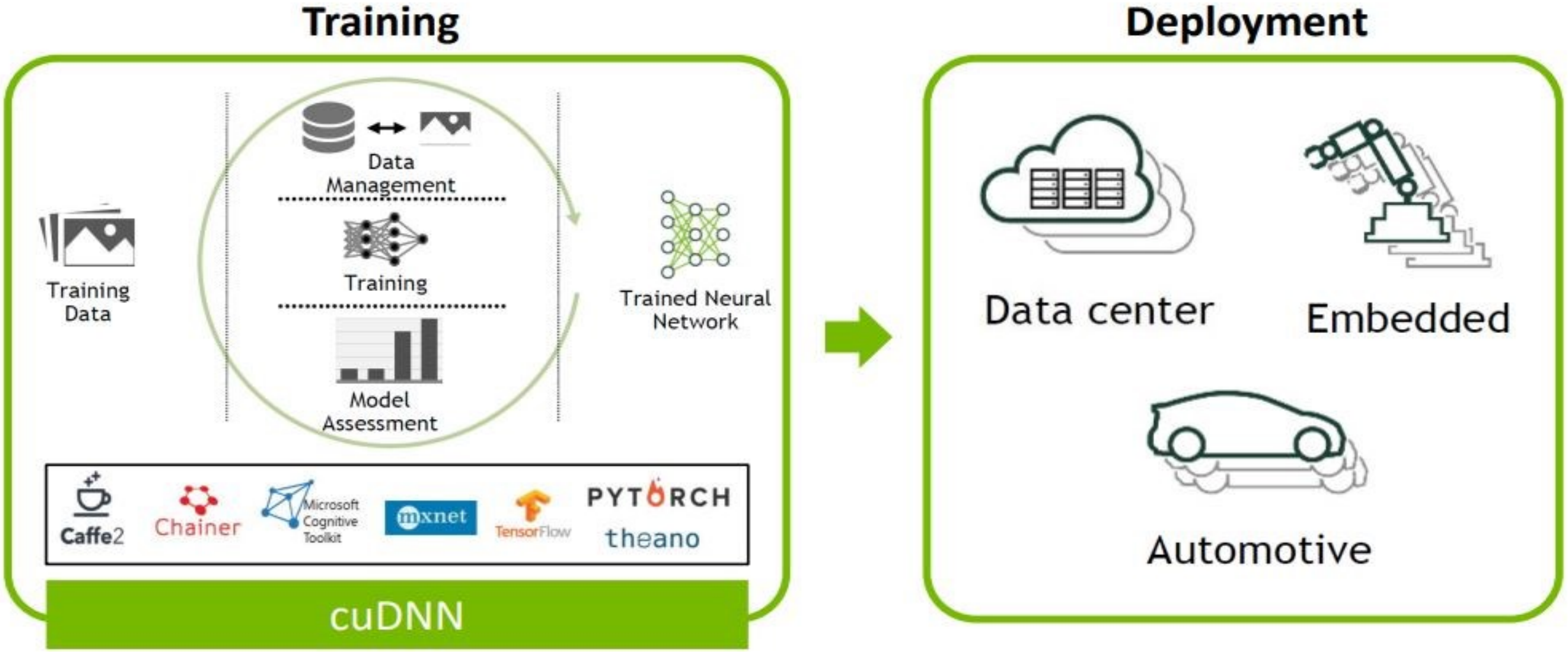
TensorRT

- Released around early 2017.
- Tensor flow tweaked version for inference optimizations.
- Works on embedded and production platforms.
- Provides acceleration on devices like Jetson nano, TX2, Tesla GPUs and more.
- Optimizations upto FP16 and INT8.
- Provides 8x increase in performance when accurately implemented.

TensorRT Ecosystem

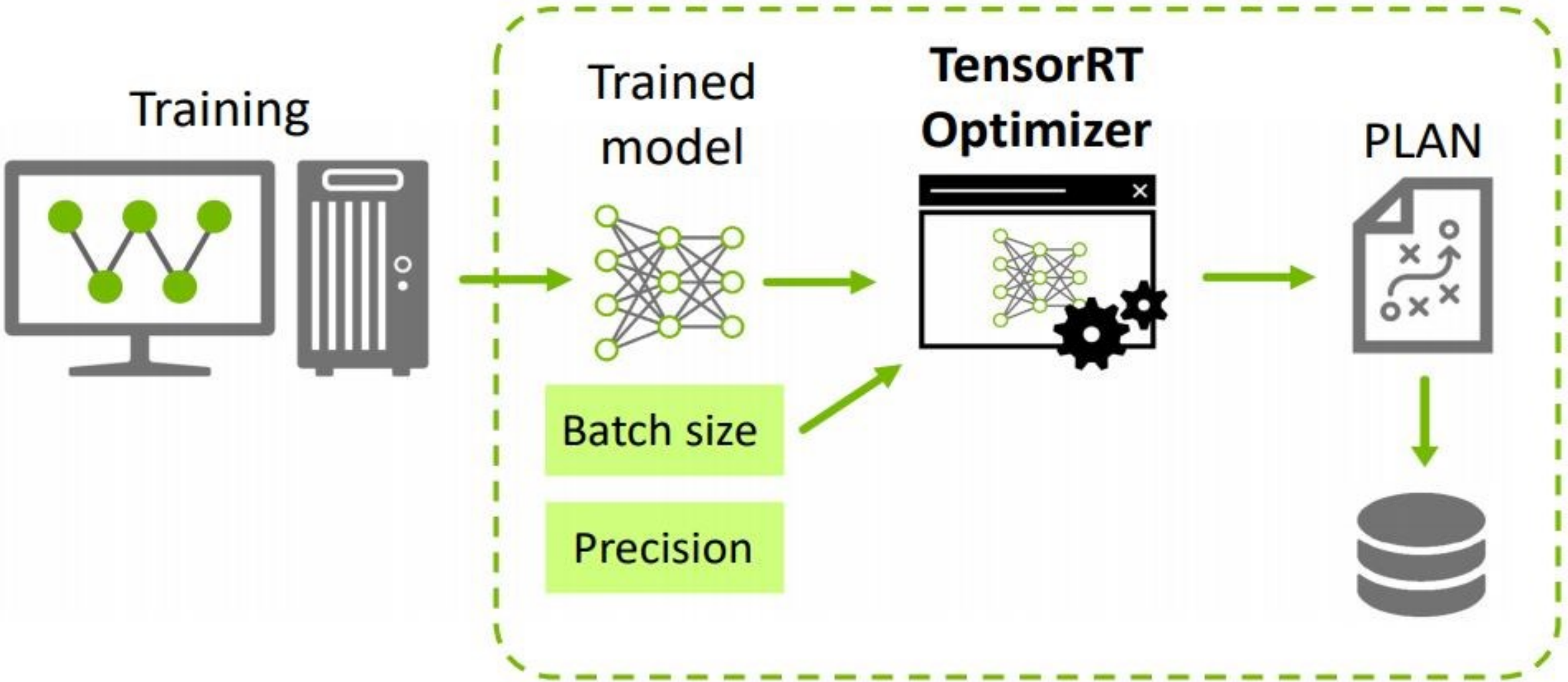


DL app development process



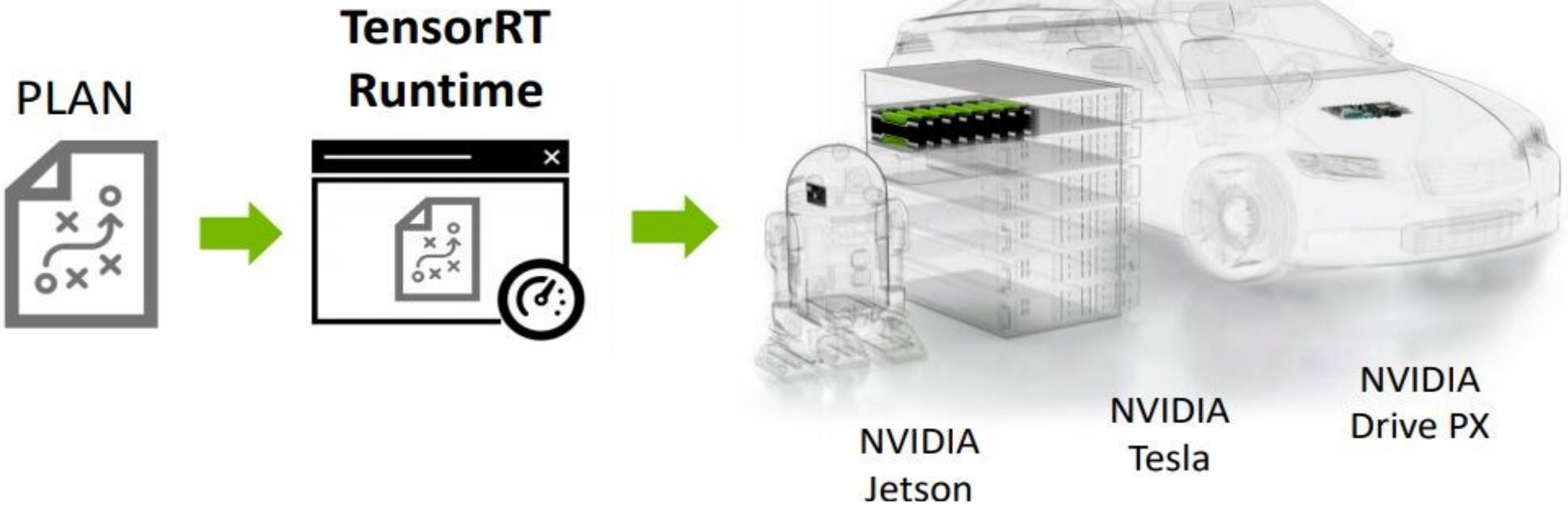
Credits : Dmitry Orbochenko, NVidia, July 2018

Step 1: Optimization



Credits : Dmitry Orbochenko, NVidia, July 2018

Step 2: Inference



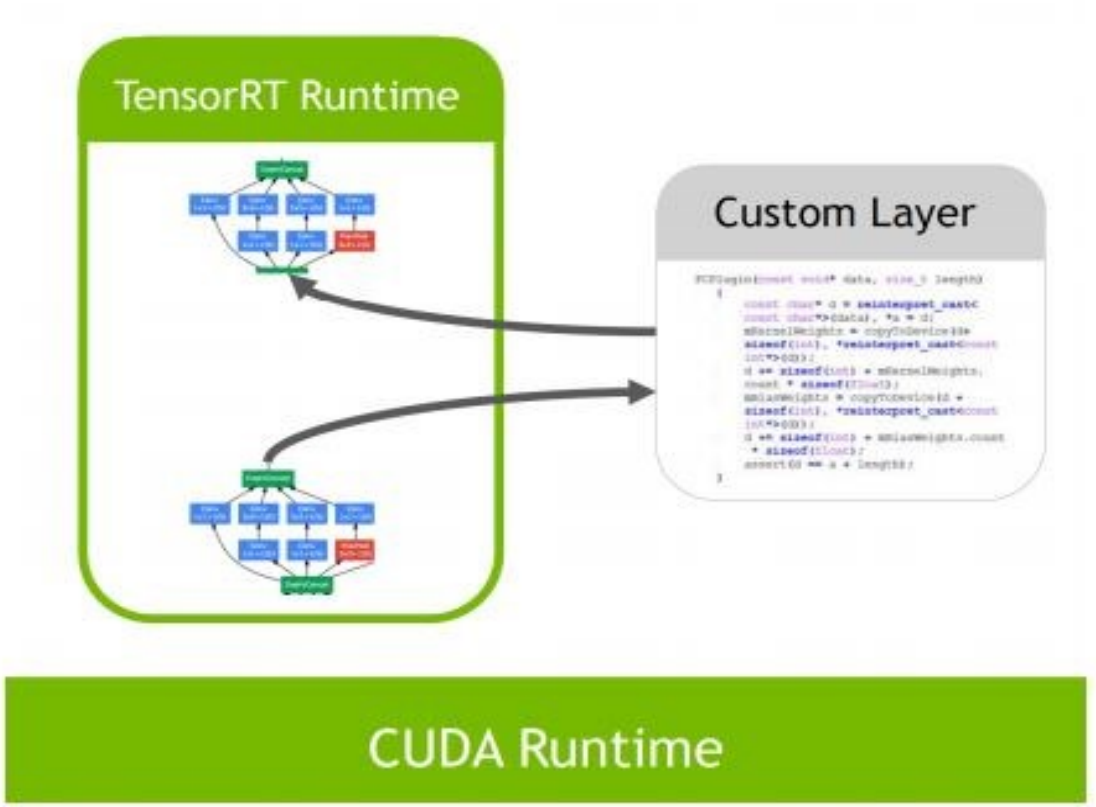
Credits : Dmitry Orbochenko, NVidia, July 2018

Supported layers

Built-in layers (v 4.0)

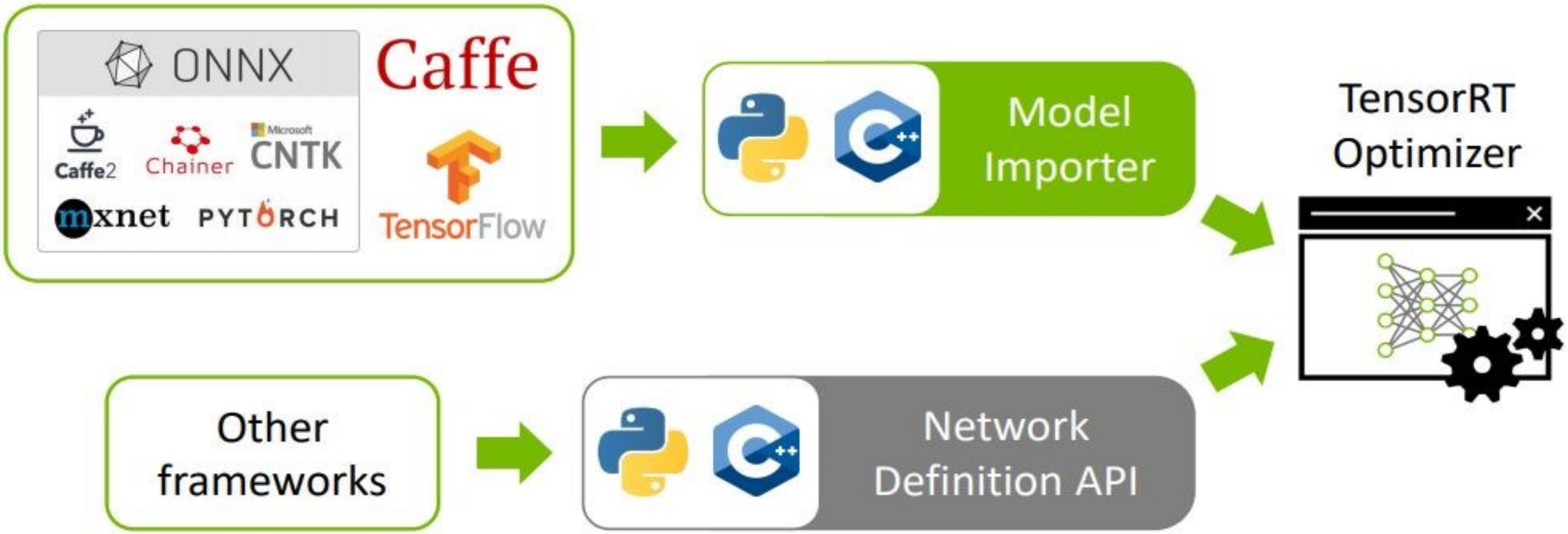
- Convolution
- Deconvolution
- Fully-connected
- Pooling, LRN
- Activation: ReLU, tanh, sigmoid
- SoftMax, Ragged SoftMax
- RNN: LSTM, GRU, RNNv2
- Misc: scaling, element-wise, concatenation, flatten, padding, shuffle, squeeze, matmult, const, gather, reduce, topK
- Unary: exp, log, sqrt, recip, abs, neg

Custom layers



Credits : Dmitry Orbochenko, NVidia, July 2018

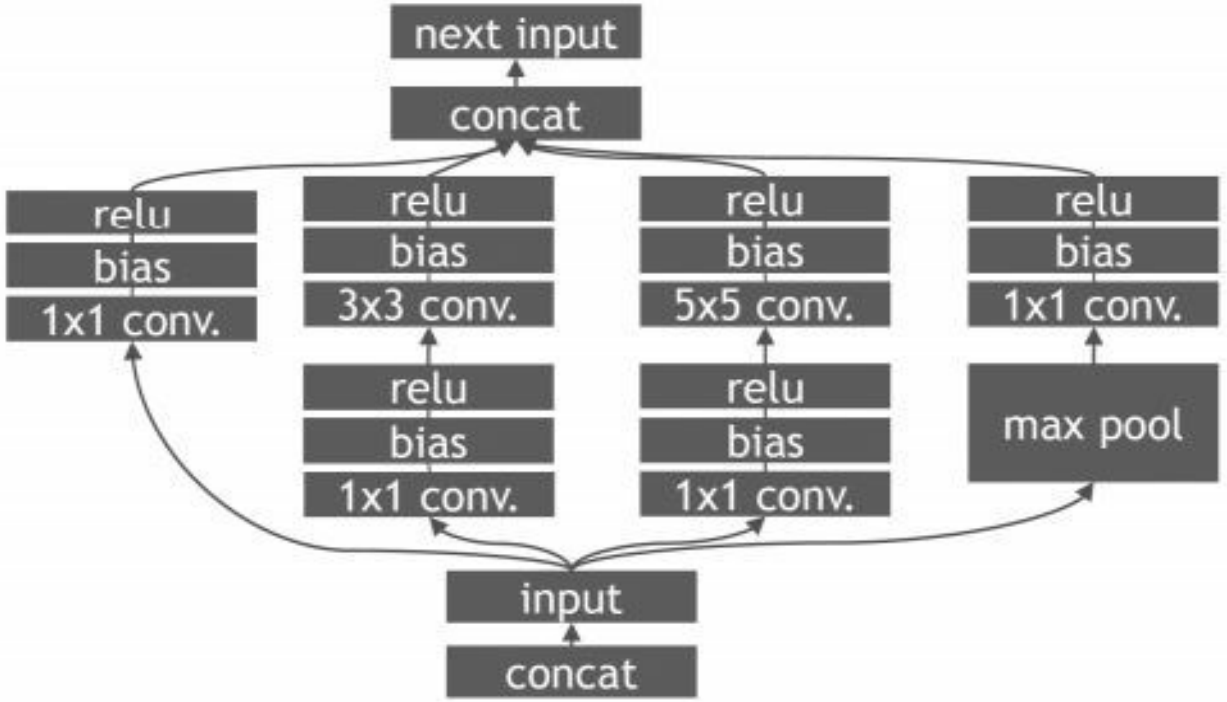
Model import



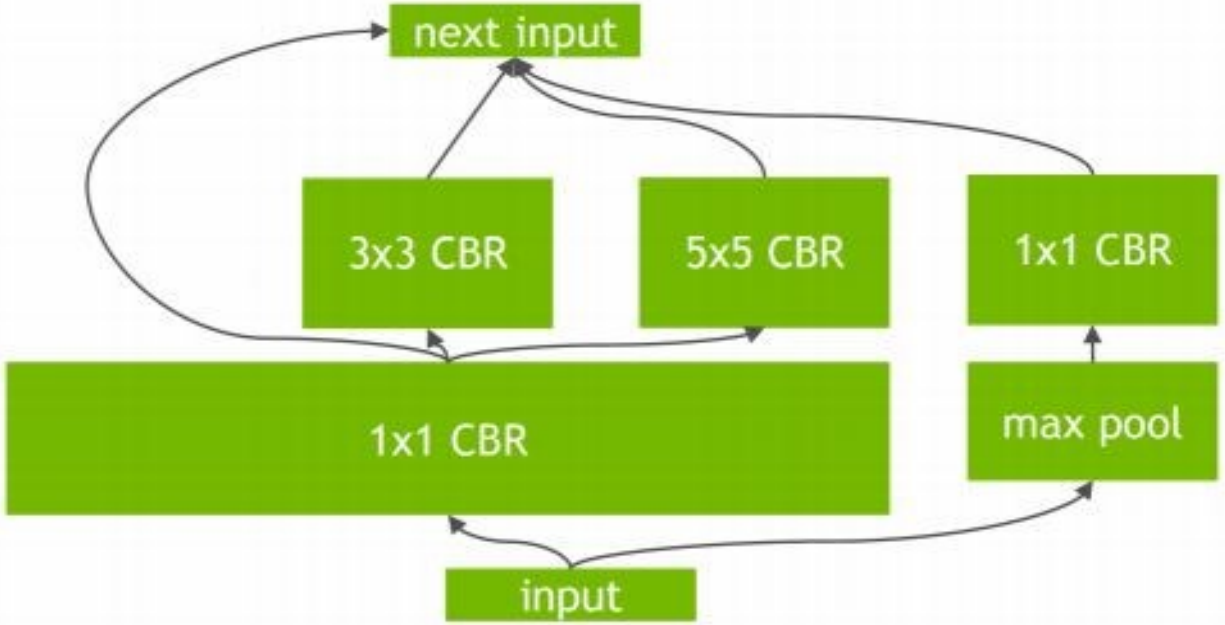
Credits : Dmitry Orbochenko, NVidia, July 2018

Layer and tensor fusion

Unoptimized net



After TensorRT optimization



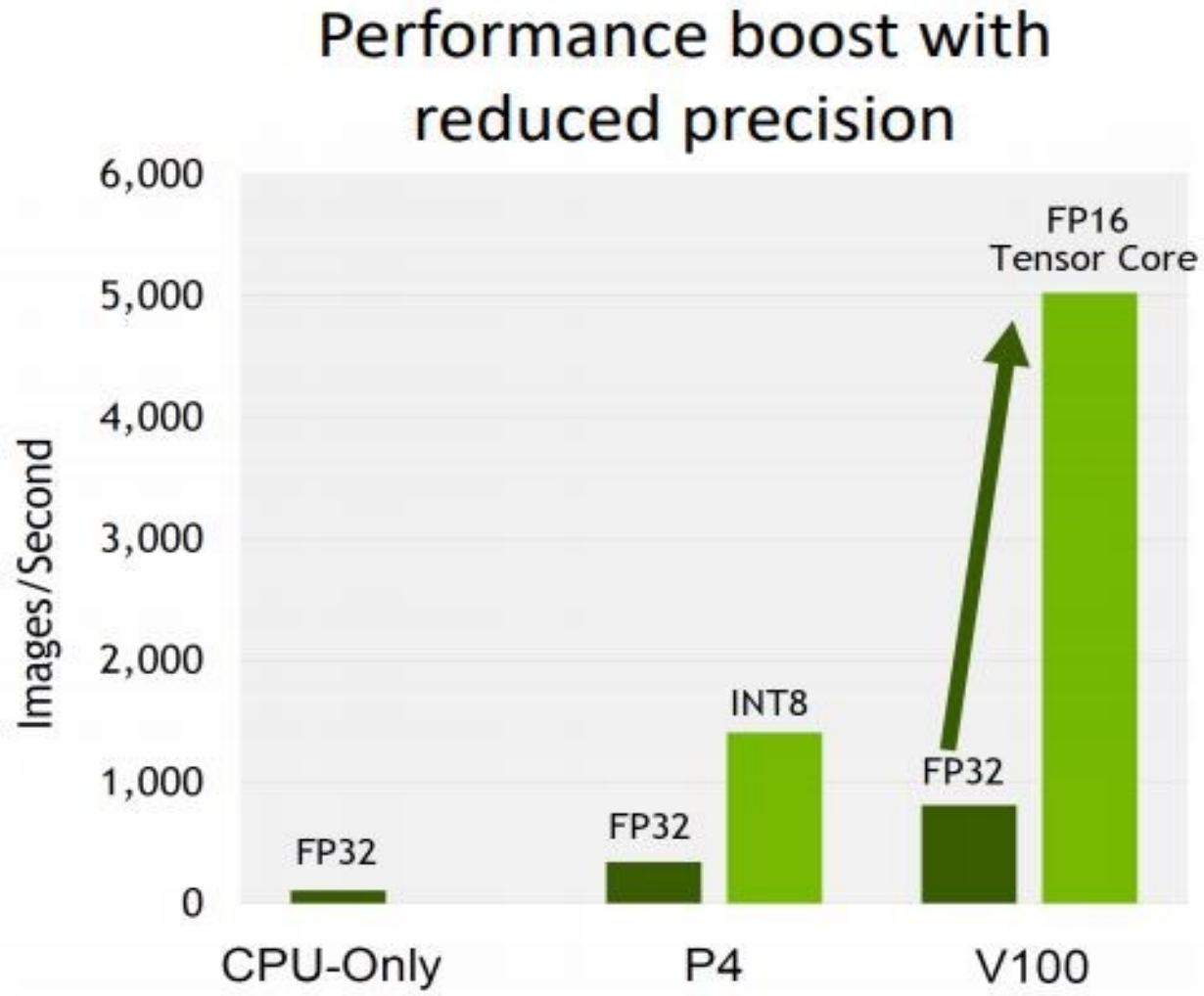
- Vertical + horizontal fusion
- Elimination of concat layers

Credits : Dmitry Orbochenko, NVidia, July 2018

Precision calibration

Precision	Range
FP32	-3.4×10^{38} ~ $+3.4 \times 10^{38}$
FP16	-65504 ~ +65504
INT8	-128 ~ +127

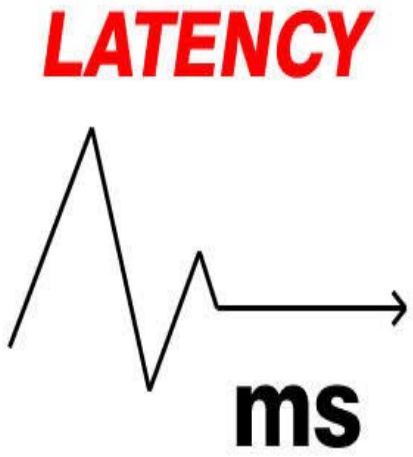
Automatic weights calibration is performed, if **INT8** is used.



Credits : Dmitry Orbochenko, NVidia, July 2018

Factors deciding performance.

Throughput	Efficiency	Latency	Accuracy	Memory Usage
- Inferences per second - Samples per second	- Performance per watt - Throughput per unit-power	- Time to execute an inference. - Measured In milliseconds.	- Delivering the correct answer. - Top-5 or Top-1 predictions in case of classifications	- Host+Device memory for inference. - Important in multi-network, multi-camera configurations



Performance

- What will likely happen:

- I am the neural network:

What really makes it fast though?

Function

The build phase performs the following optimizations on the layer graph:

- Elimination of layers whose outputs are not used
- Elimination of operations which are equivalent to no-op
- Fusion of convolution, bias and ReLU operations
- Aggregation of operations with sufficiently similar parameters and the same source tensor (for example, the 1x1 convolutions in GoogleNet v5’s inception module)
- Merging of concatenation layers by directing layer outputs to the correct eventual destination.

Usages and code

- Python API

```
import tensorrt as trt
```

```
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
```

- C++ API

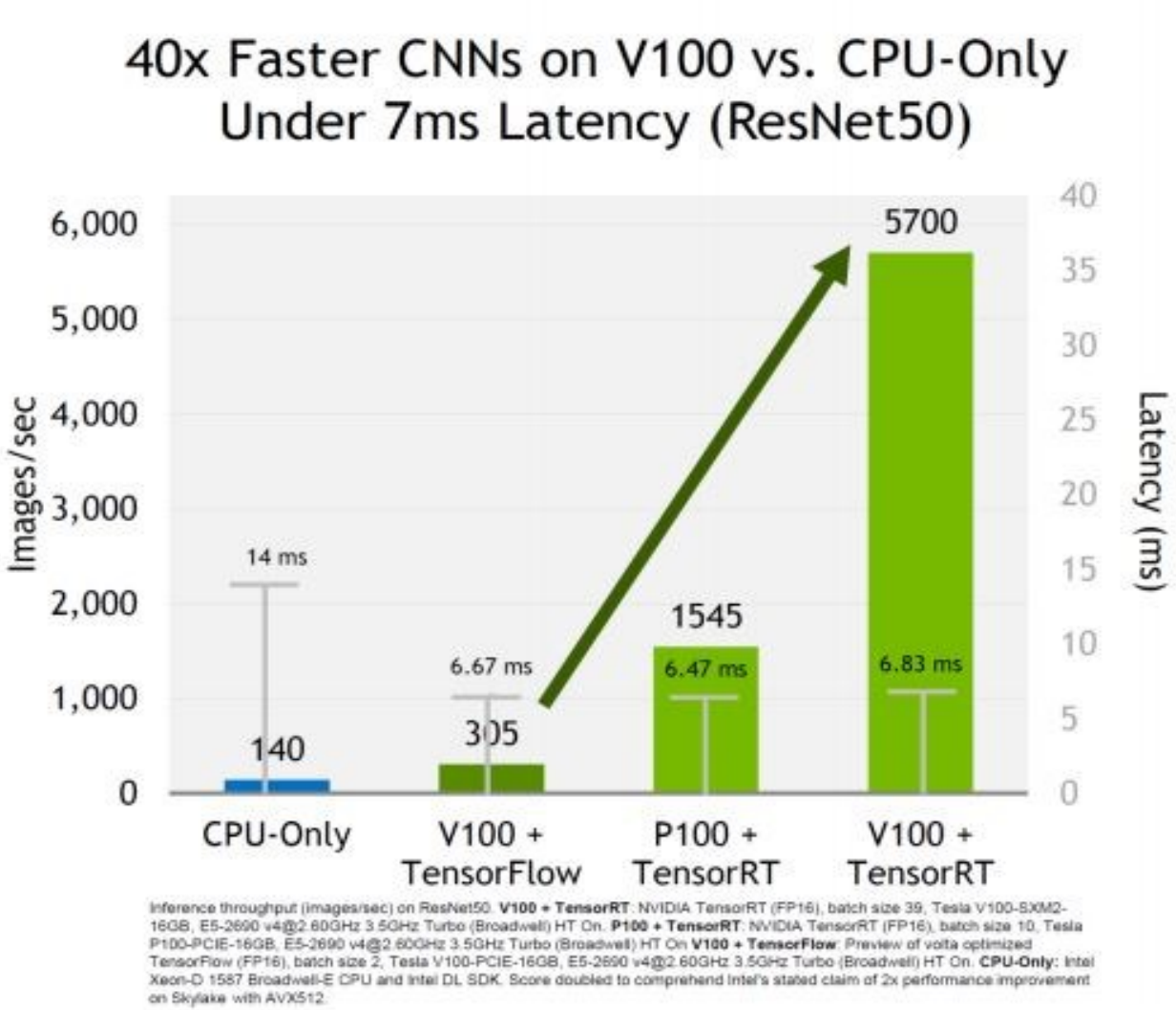
```
class Logger : public ILogger
{
    void log(Severity severity, const char* msg) override
    {
        // suppress info-level messages
        if (severity != Severity::kINFO)
            std::cout << msg << std::endl;
    }
} gLogger;
```

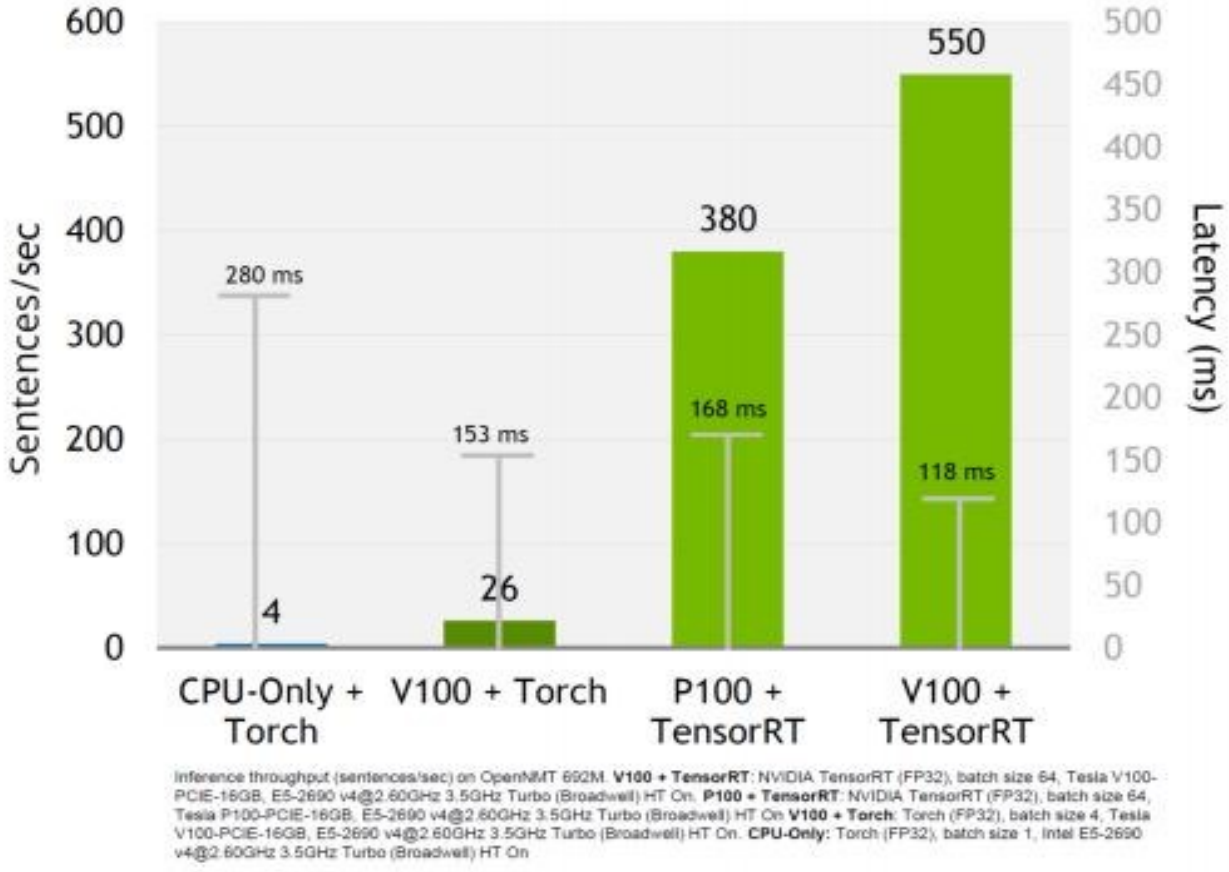
Writing Network definitions.

- Python API

```
with trt.Builder(TRT_LOGGER) as builder, builder.create_network() as network:
        # Configure the network layers based on the weights provided. In this case, the weights are imported from a pytorch model.
        # Add an input layer. The name is a string, dtype is a TensorRT dtype, and the shape can be provided as either a list or tup
        input_tensor = network.add_input(name=INPUT_NAME, dtype=trt.float32, shape=INPUT_SHAPE)

        # Add a convolution layer
        conv1_w = weights['conv1.weight'].numpy()
        conv1_b = weights['conv1.bias'].numpy()
        conv1 = network.add_convolution(input=input_tensor, num_output_maps=20, kernel_shape=(5, 5), kernel=conv1_w, bias=conv1_b)
        conv1.stride = (1, 1)

        pool1 = network.add_pooling(input=conv1.get_output(0), type=trt.PoolingType.MAX, window_size=(2, 2))
        pool1.stride = (2, 2)
        conv2_w = weights['conv2.weight'].numpy()
        conv2_b = weights['conv2.bias'].numpy()
        conv2 = network.add_convolution(pool1.get_output(0), 50, (5, 5), conv2_w, conv2_b)
        conv2.stride = (1, 1)

        pool2 = network.add_pooling(conv2.get_output(0), trt.PoolingType.MAX, (2, 2))
        pool2.stride = (2, 2)

        fc1_w = weights['fc1.weight'].numpy()
        fc1_b = weights['fc1.bias'].numpy()
        fc1 = network.add_fully_connected(input=pool2.get_output(0), num_outputs=500, kernel=fc1_w, bias=fc1_b)

        relu1 = network.add_activation(fc1.get_output(0), trt.ActivationType.RELU)

        fc2_w = weights['fc2.weight'].numpy()
        fc2_b = weights['fc2.bias'].numpy()
        fc2 = network.add_fully_connected(relu1.get_output(0), OUTPUT_SIZE, fc2_w, fc2_b)

        fc2.get_output(0).name =OUTPUT_NAME
        network.mark_output(fc2.get_output(0))
```

TensorRT Performance



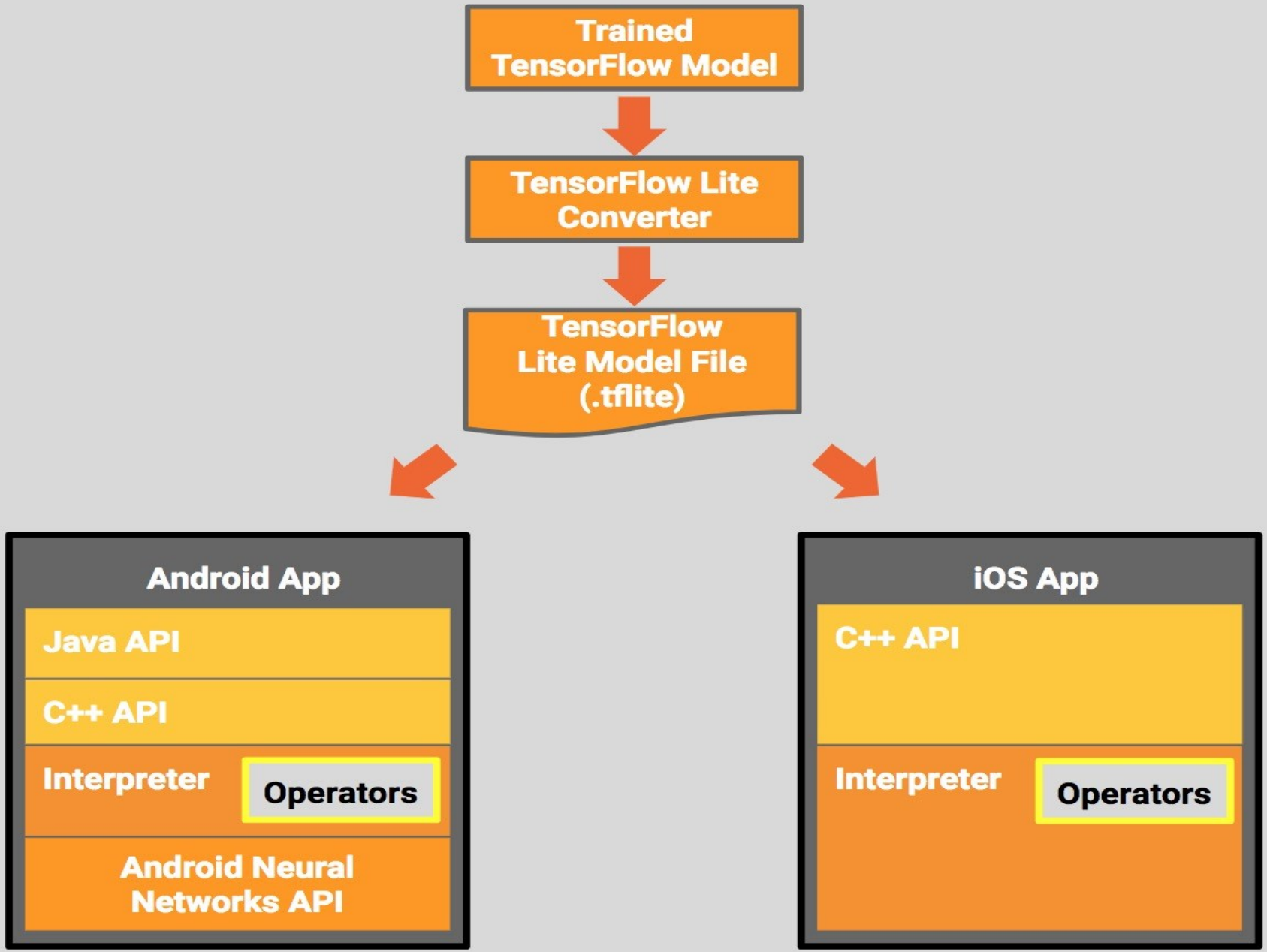


Credits : Dmitry Orbochenko, NVidia, July 2018

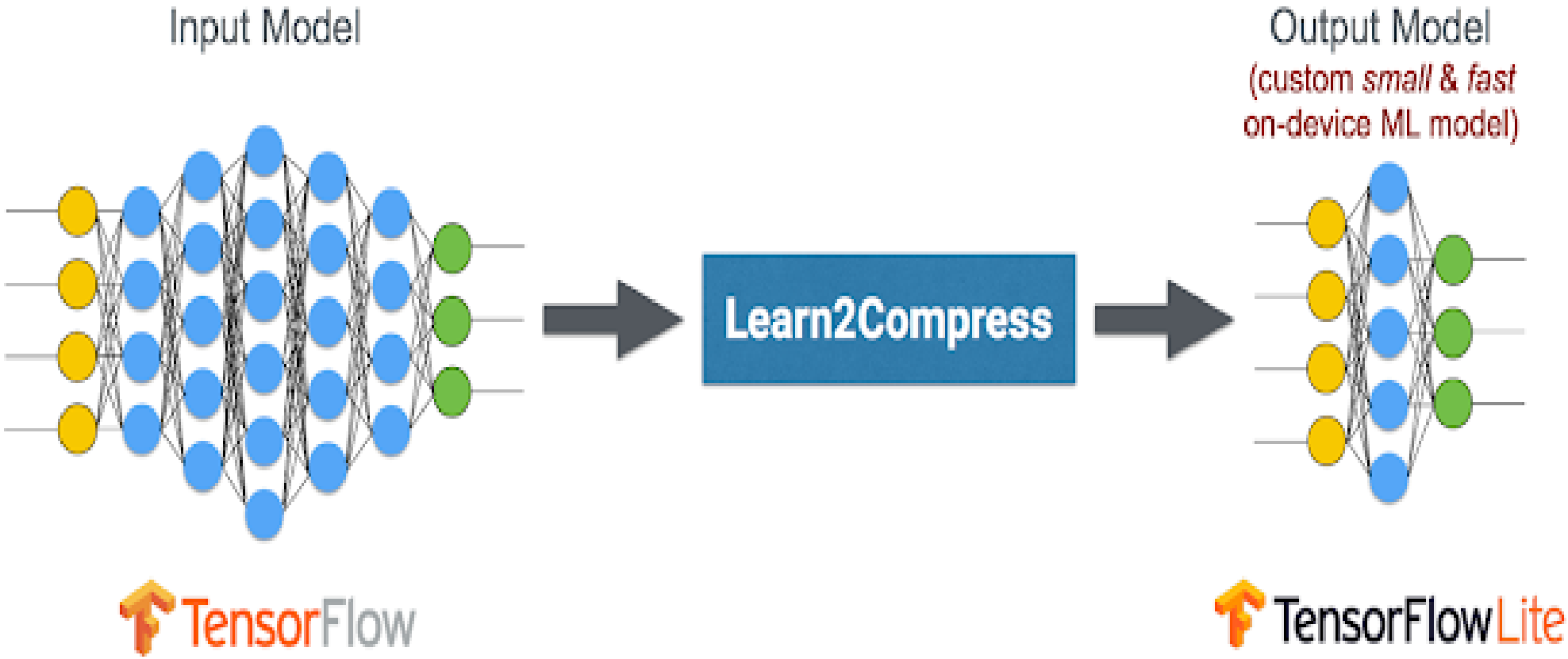
Tensorflow Lite(TFLite)

- Version 0.5 initial release in early 2016.
- Released for mobile, web and embedded devices.
- Tensorflow tweaked for model optimizations.
- Less binary size for the model.
- Works on a large ecosystem of devices and operating systems.
- Range of TFLite specific devices compatible with Raspberry Pi, USB accelerator, edge TPU.

Architecture



TFLite Ecosystem



Usages and code.

- Convert existing tensorflow SavedModel :

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_model)
```

- Quantized tflite model – reducing precision:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_quantized_model)
```

Inference

- Python API

```python
import numpy as np
import tensorflow as tf

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="converted_model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test model on random input data.
input_shape = input_details[0]['shape']
input_data = np.array(np.random.random_sample(input_shape), dtype=np.float32)
interpreter.set_tensor(input_details[0]['index'], input_data)

interpreter.invoke()

# The function `get_tensor()` returns a copy of the tensor data.
# Use `tensor()` in order to get a pointer to the tensor.
output_data = interpreter.get_tensor(output_details[0]['index'])
print(output_data)
```

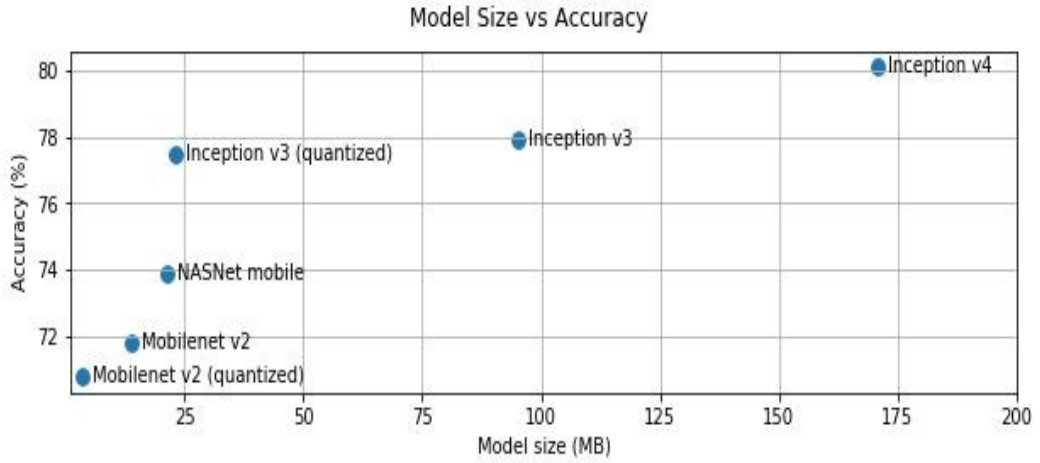
- C++ API

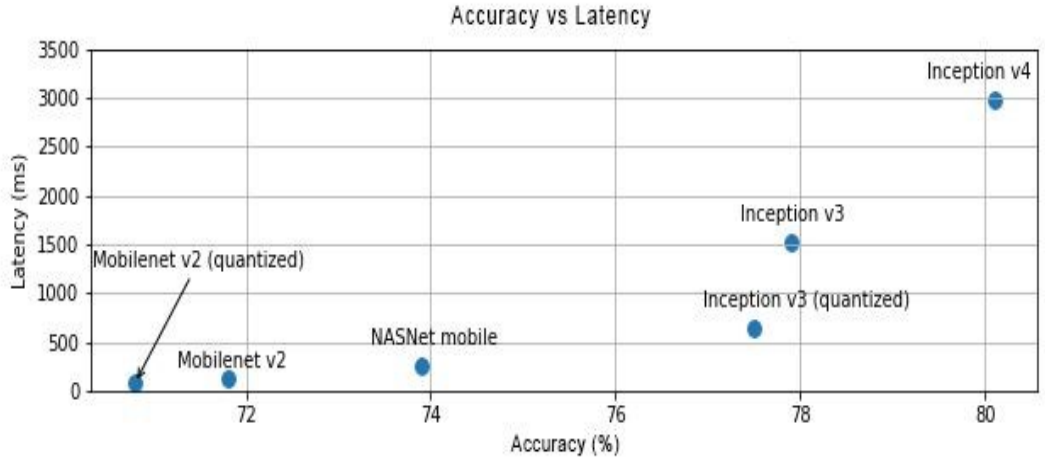
```cpp
// Load the model
std::unique_ptr<tflite::FlatBufferModel> model =
    tflite::FlatBufferModel::BuildFromFile(filename);

// Build the interpreter
tflite::ops::builtin::BuiltinOpResolver resolver;
std::unique_ptr<tflite::Interpreter> interpreter;
tflite::InterpreterBuilder(*model, resolver)(&interpreter);

// Resize input tensors, if desired.
interpreter->AllocateTensors();

float* input = interpreter->typed_input_tensor<float>(0);
// Fill `input`.

interpreter->Invoke();

float* output = interpreter->typed_output_tensor<float>(0);
```

Is it really fast though?

Performance

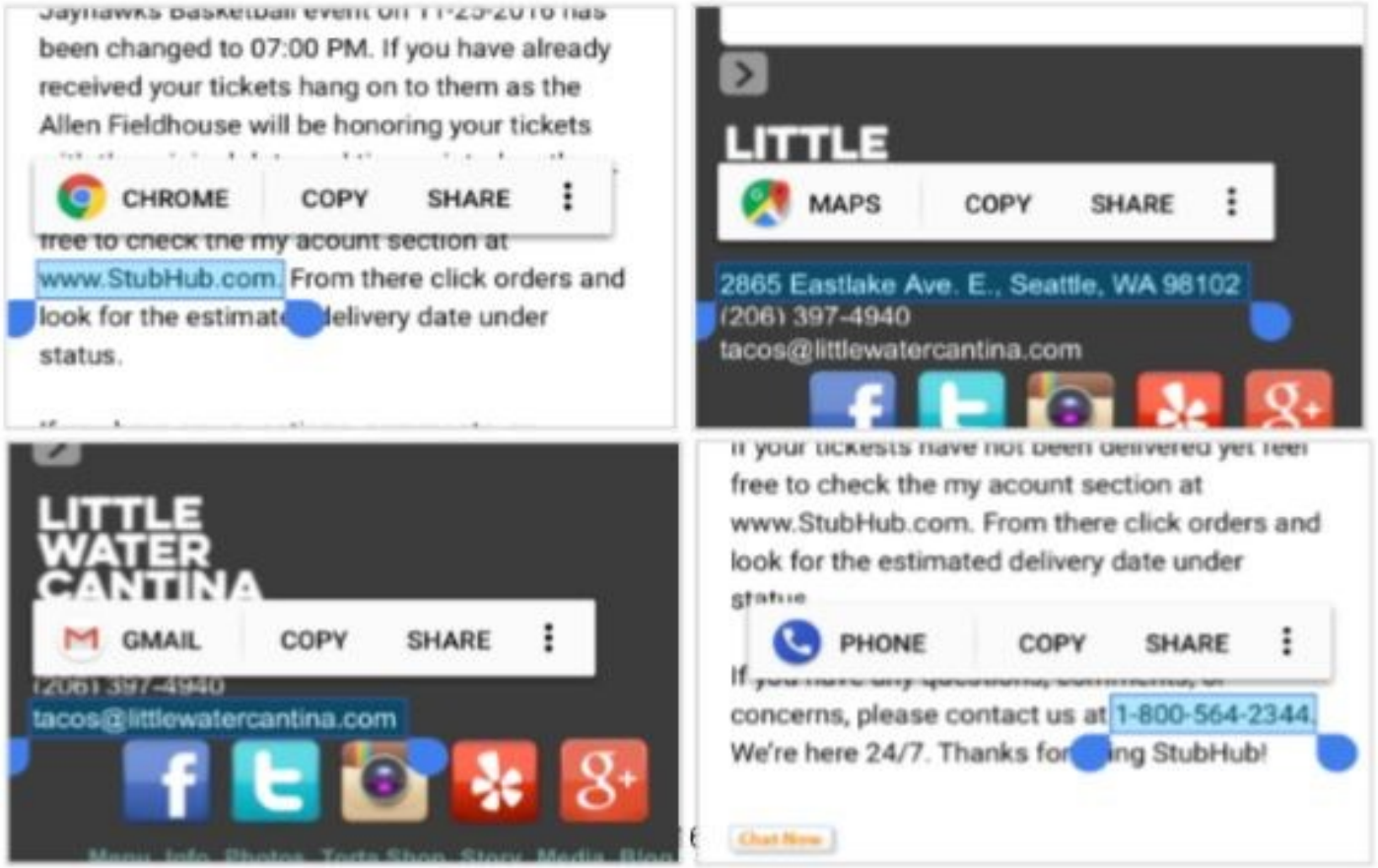
- Post training quantization.
- Quantization aware training.
- GPU Delegates.





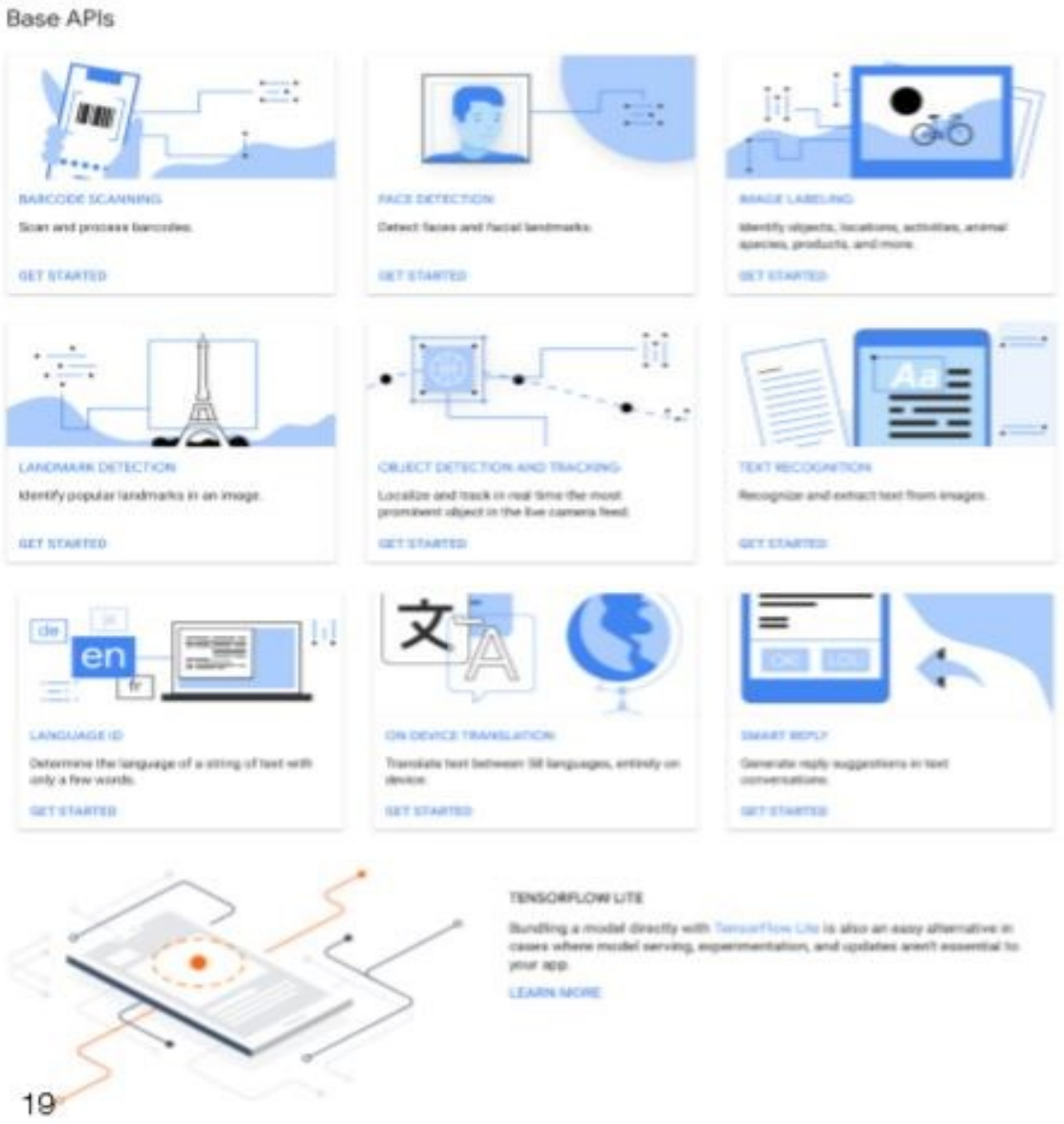
TF Lite in Android Pie

- There are '`libtflite.so`'s in /system/lib and /system/lib64
- https://source.android.com/devices/tech/display/textclassifier



ML Kit

- https://developers.google.com/ml-kit/, part of FireBase
- Originally, only custom models are TFLite
- Now, as far as I can tell, vision parts are using TFLite also



https://developers.google.com/ml-kit/

In a jiffy

- TensorRT and/or Tensorflow Lite can be your solution to :
  - Training your model in a optimized manner.
  - Deploy your optimized model.
  - Inference at an increased speed of upto 8x faster.
  - Minimize hardware resource usages.
  - Reduce latency if model is on cloud.

Thank you