AI性能优化方法论

2021/10

内容大纲

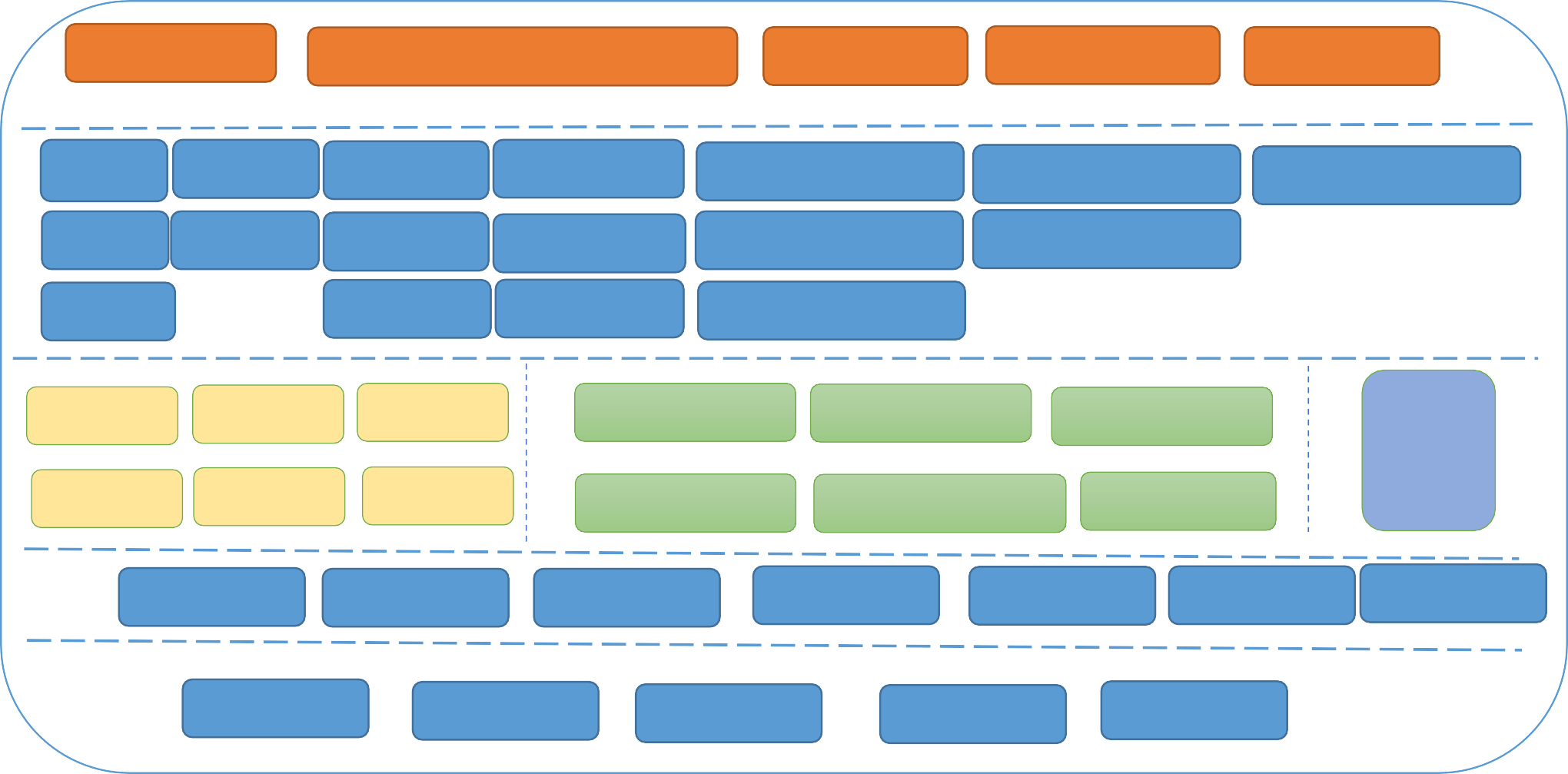
# 简介

## 优化方法论

* + 模型压缩
  + 系统级优化

## 总结

#### AI基本框架

常见应用场景

计算机视觉

自然语言处理、机器翻译

搜索推荐

语音处理 ……

Resnet

VGG

Bert

Albert

WDL

DeepFM

........

常用算法模型

YOLO

MobileNet

Transformer XLNet

DIN

DSSM

Inception

ERNIE

LSTM

DIEN

机器学习框架

Pytorch Caffe

Tensorflow MXNet

PaddlePaddle

Kaldi

OpenVINO TVM

ONNX RUNTIME

Tensor Comprehensions

TensorRT

……

AutoML

系统依赖 oneDNN CUDA cuDNN Compiler OpenMP Eigen ……

硬件平台

CPU GPU ARM

FPGA ……

1

流处理平台

实时输入

预

**模型训练**

特征平台/数据平台

原始数据

（日志、用户历

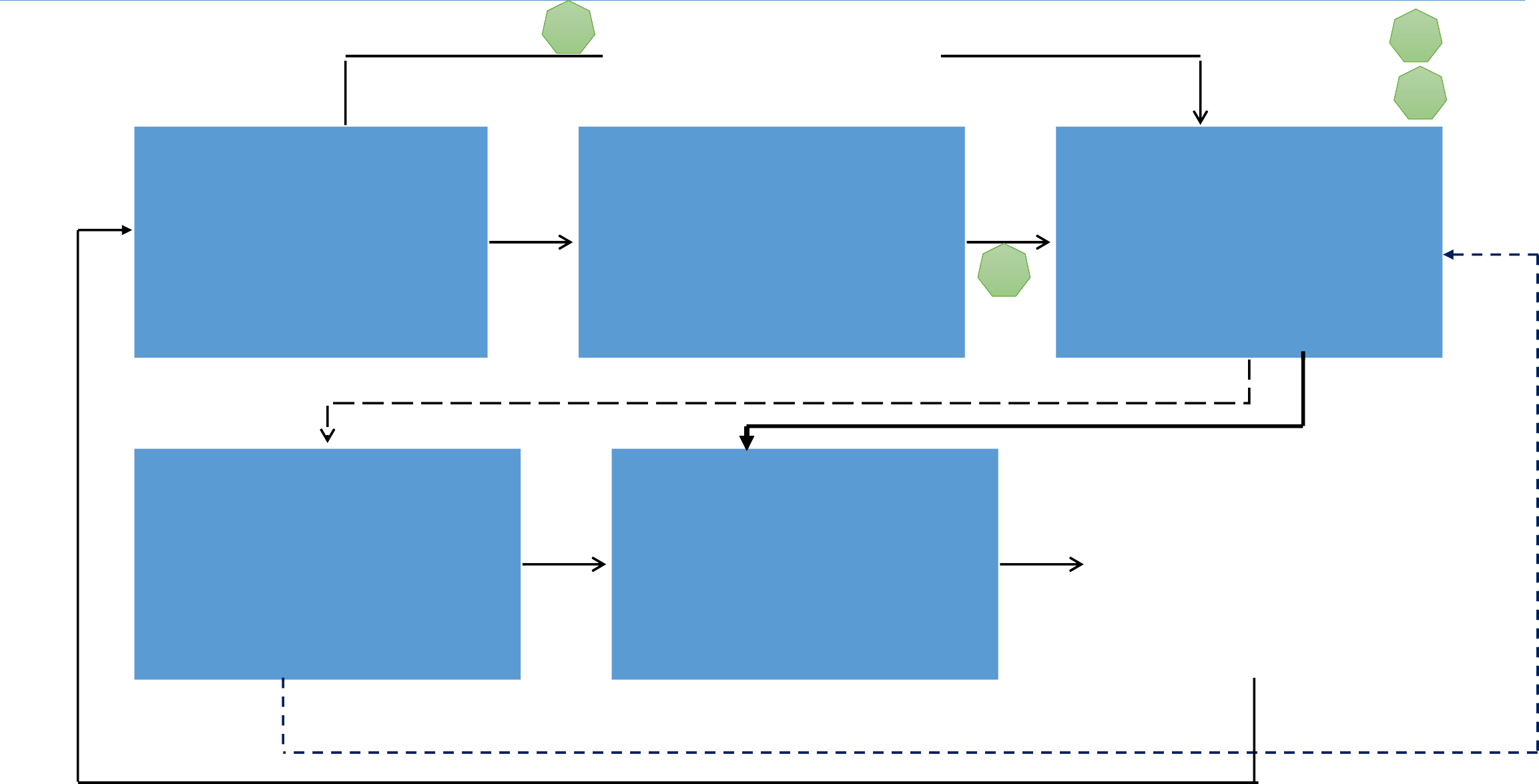
史行为等）

处 输

理 入

2

1. Online training
2. Offline Training

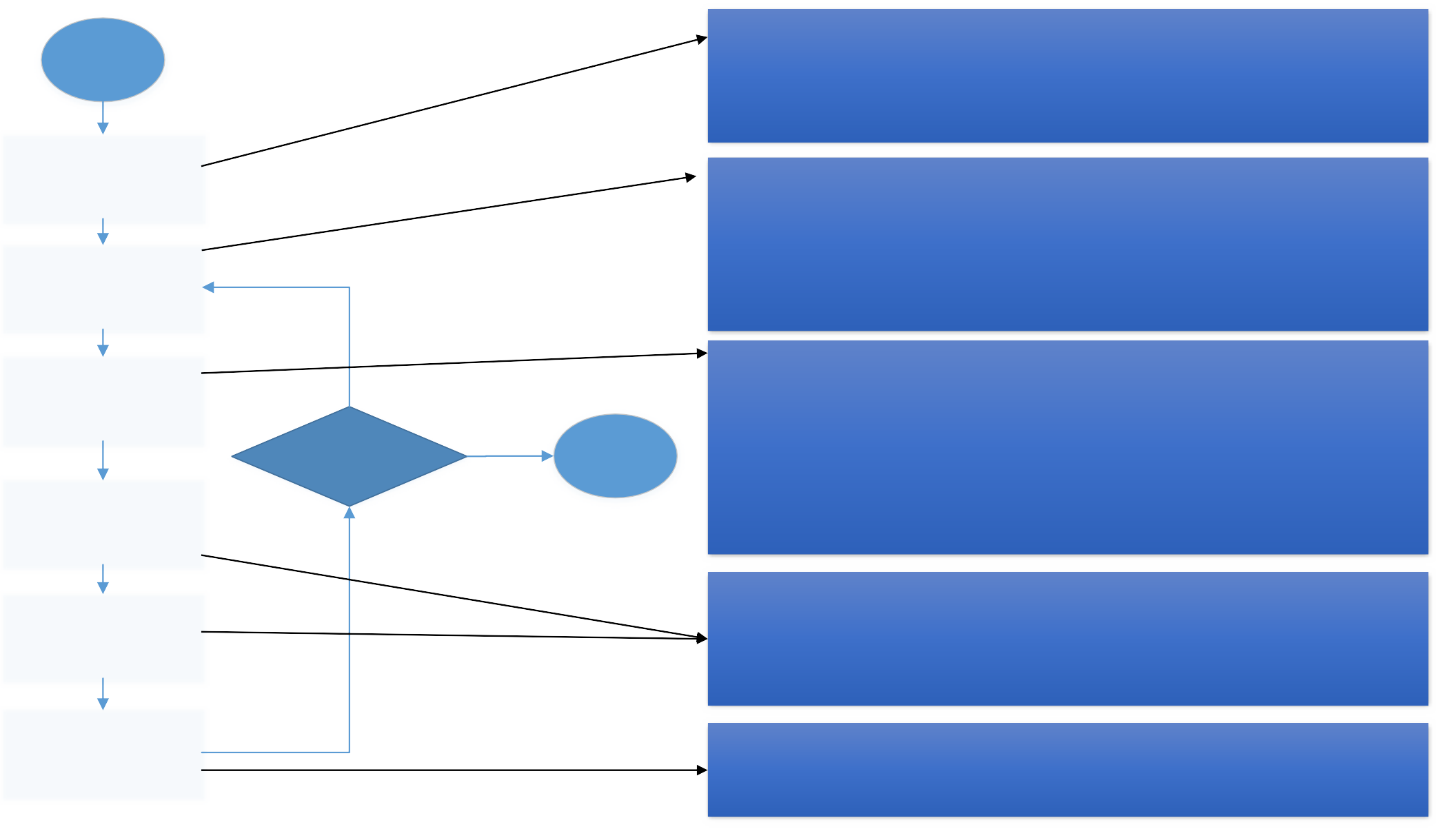


生成 Finetune

模型性能优化

模型部署推理

在线服务





开始

* Workload Info
* Framework
* Training or Influence

确定优化目标

否

性能分析

* Perf
* Emon/Vtune
* Timeline for TF
* Pytorch Profling tool

是

定位性能瓶颈

* Computing bound
* Memory Bound: latency/throughput
* Roofline Analysis
* Pre-process or Post-Process
* Bottleneck Details

达到性能目标 结束

制定优化策略

执行优化措施

* **算法：减小算法复杂度\计算量\访问量**
* **软件：提升硬件使用效率**
* **硬件：升级**

性能测试

* **Throughput**
* **Latency**

内容大纲

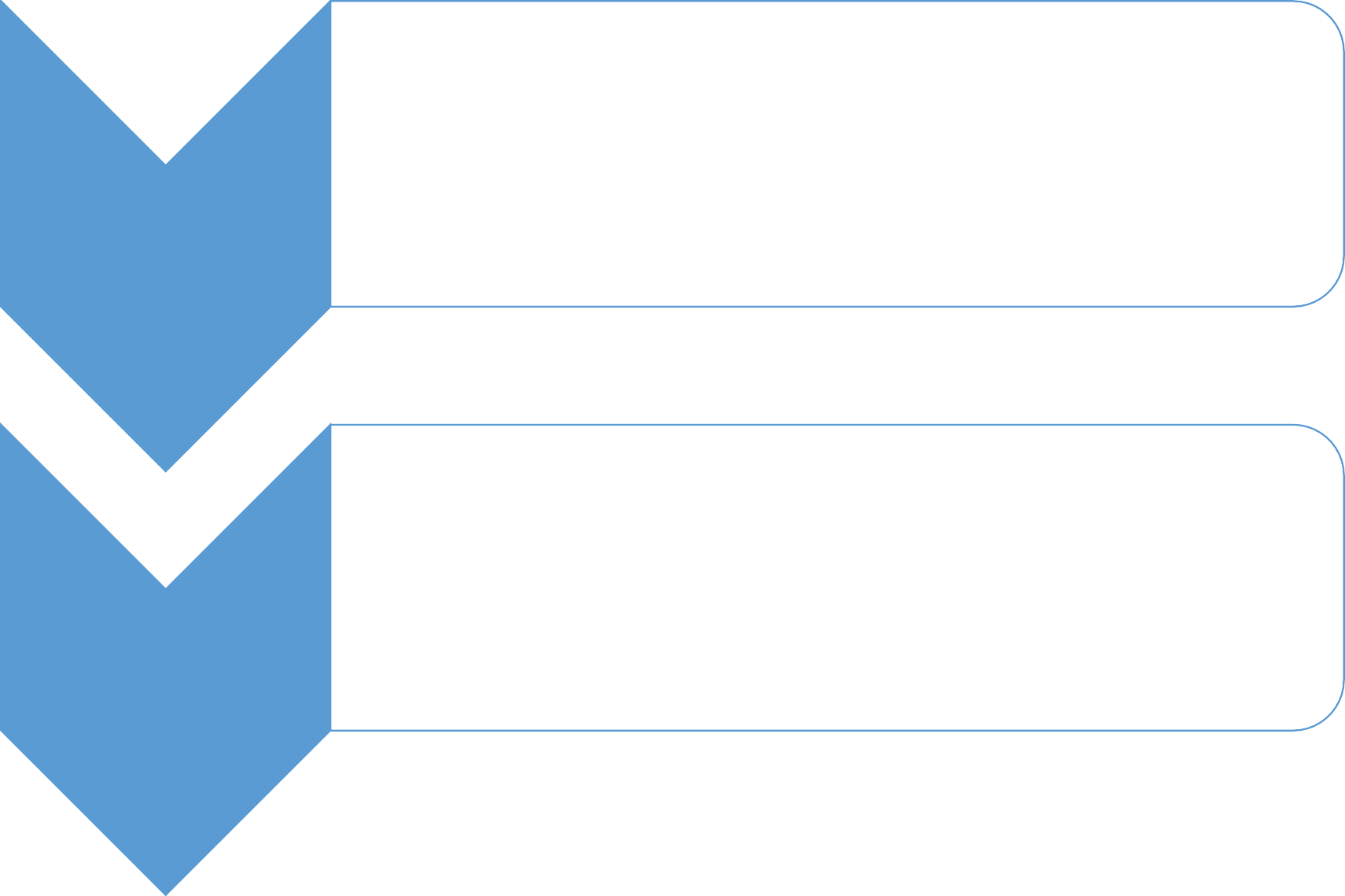
* 简介

# 优化方法论

* + 模型压缩
  + 系统级优化
  + 案例分享

## 总结

##### 单机优化

* 算子优化: 算子重写，混精优化，内存重构
* 图优化：子图融合
* 数学库优化
* 多线程优化：线程调度，负载均衡
* 内存管理优化：实现高效的内存申请和释放

##### 分布式优化

* 通信优化
* 调度优化：负载均衡
* 模型和数据访问优化：多级缓存，混合存储
* 分布式框架优化

模型压缩：降低计算复杂度，减少计算量，部分依赖于硬件平台和框架实现

* 低精度量化：int8 量化校准（依赖硬件平台）
* Bf16，fp16混合精度
* 算法层的量化
* 提升计算速度，减少模型大小，计算量不变
* 依赖于框架和硬件平台

模型量化

* 张量分解（卷积tucker分解）
* SVD分解参数矩阵（针对全连接层matmul）
* 减少总计算量, 灵活，不依赖于框架和平台，压缩效果有限

低秩估计

* 结构化剪枝
  + Channel Pruning
  + Filter Pruning
  + Layer Pruning
  + 减少了计算量
* 非结构化剪枝
  + Weight Pruning（稀

疏化剪枝）

* + 依赖平台性能实现

模型剪枝

* 实现良好的fine tune模块
* 轻量网络设计，使用原来的网络指导训练新网络
* 配合其他网络压缩方法共同发挥作用
* 调参困难，但是普遍可提升模型准确率，不依赖框架和平台

模型蒸馏

* AutoML

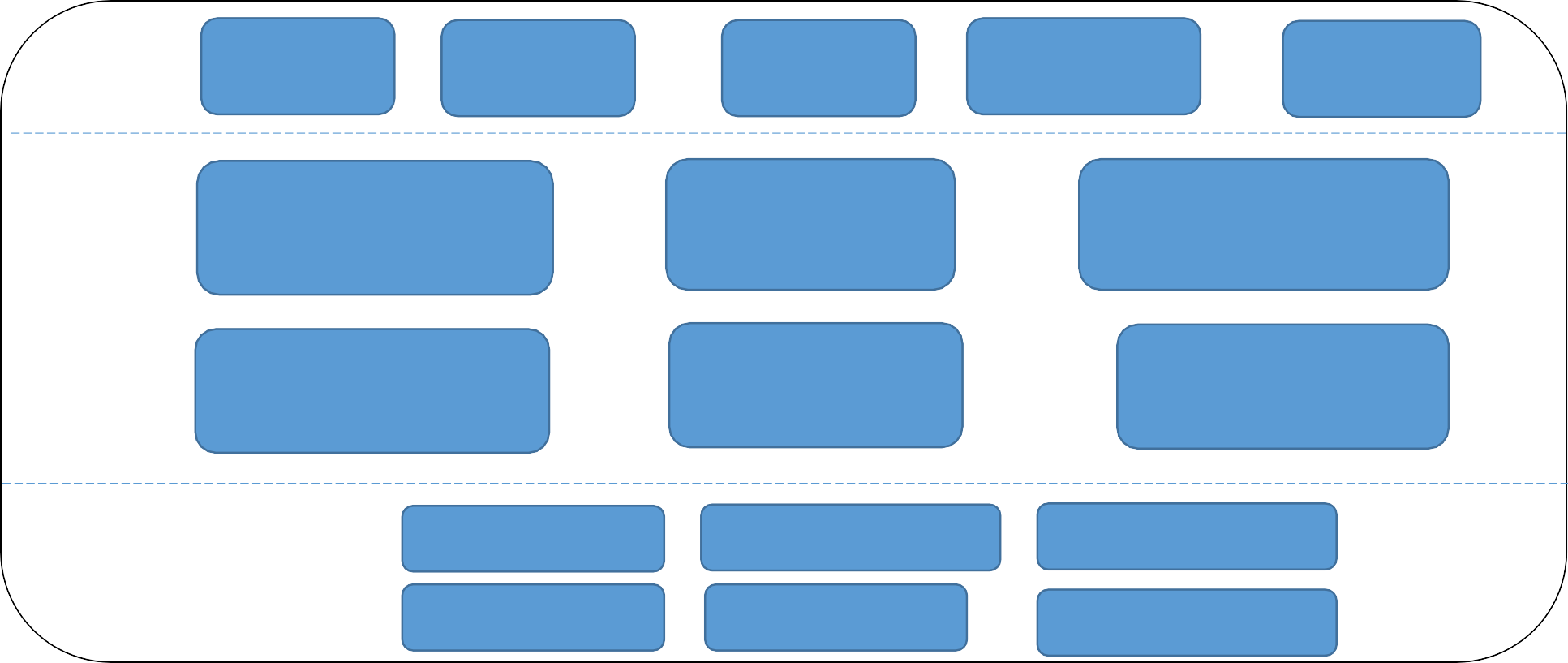
自动压缩

#### AI推理引擎功能模块-系统级优化

**效果有限依赖度高**

**优化重点**

系统级优化：

* + 提升硬件利用率，实现更高的计算访存比，
  + 完全依赖于硬件平台，框架和系统环境

应用层优化

批处理实现

前处理/后处

理优化

Python多线程

Python JIT: numba优化

避免冗余计算

框架层优化

自定义高性能核心算子

Conv/matmul

多框架融合，混合使用

多算法自动匹配使用

Win 卷积

ograd/direct gemm

即时编译技术：

* 整个模型完全使用JIT
* 核心算子使用JIT

自定义算子融合

自定义重写部分算子

系统环境和运行时优化

运行时环境变量设置

高性能数学库和指令集

操作系统配置优化

硬件配置

模型部署方式优化

（Docker/tf serving）

硬件资源分配方式

**模型外的优化**

内容大纲

* 简介

# 优化方法论

###### 模型压缩

* + 系统级优化
  + 案例分享

## 总结

* 对热点算子matmul和conv2D进行int8量化，

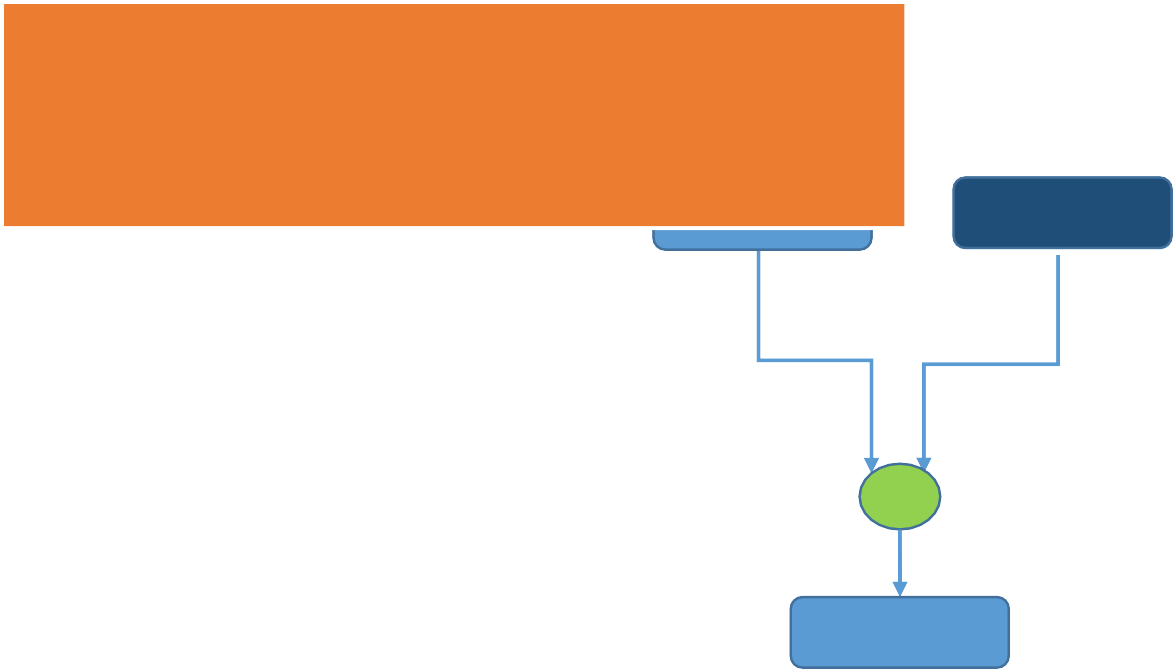
减小模型尺寸和内存带宽，实现模型压缩

* 新增的算子可以通过固化和融合操作去除冗余计算

Quantize

输入数据

min max

QuantizedConv QuantizedMatmul

输入数据

权值参数

矩阵乘法matmul

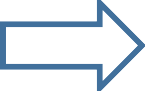
卷积计算conv

输🎧结果

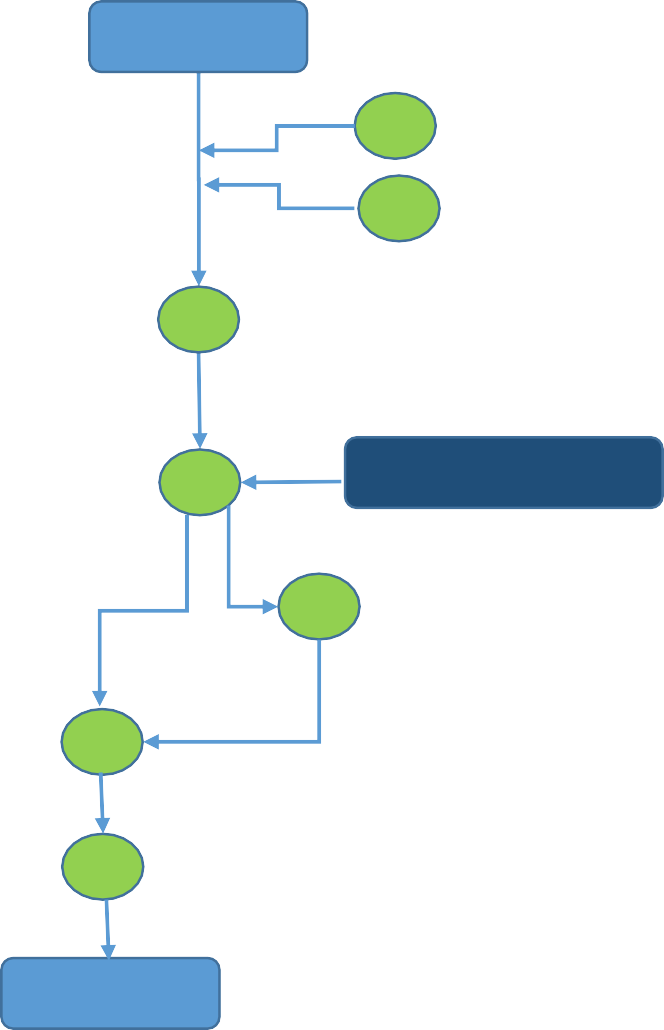
**容易引入额外开销，需要trade-off**

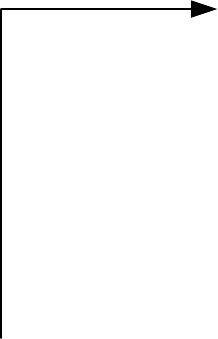
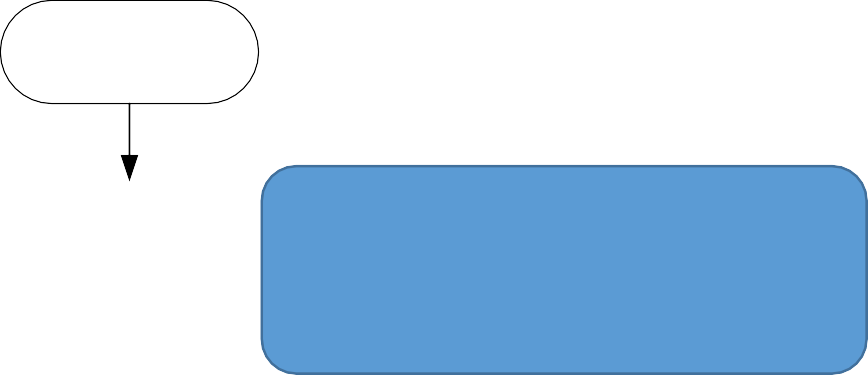
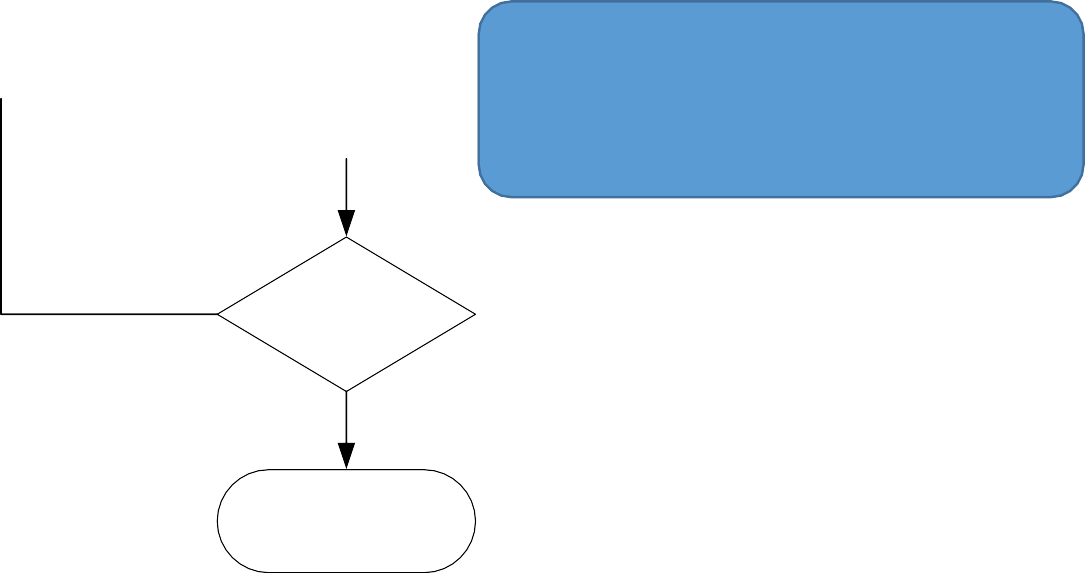
量化权值参数

RequantizationRange

Requantize Dequantize

输🎧结果

图1 float32 计算图 图2 int8量化计算图



读取原始pb文件

* 参考使用tensorflow的图修改工具据具体模型修改工具代码，得到新

的pb文件

* conv/matmul等量化op已支持；特殊的op融合需要修改框架源码自定

义实现量化

* 为了保证量化后的精度，需要使用测试集进行校准；目前主流的校准

方式是基于KL散度

否

是否达到精度

是

结束

量化校准

量化算子实现

修改图生成量化pb

文件

Tensorflow基于CPU平台的int8量化实现：

* 针对不同模型需要对应修改图工具的源码，无法实现完全统一，业界现状也是如此，修改成本高
* Google tensorflow主要量化成果集中在移动端，

tensorflow Lite的解决方案

* 实现量化校准的主流方法：KL散度校准，TensorRT、

Intel LPOT

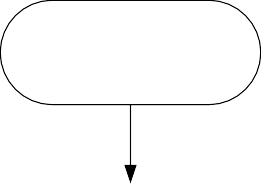
* 自动化选择量化层的高效算法和准则是int8量化的潜在优化点
* 如何选择量化层是难点，排列组合最简单，但是 效率太低，需要建立高 效的量化选择算法来实 现

修改/减少量化层

* GPU fp16量化；CPU bf16 混合精度优化
* 基于算法的量化，例如K-means

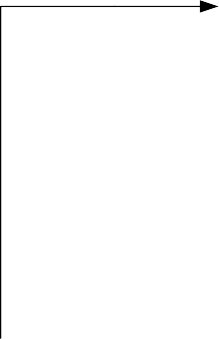
### Intel LPOT



低秩估计方法是基于矩阵的秩去除冗余计算，减少权值参数，对计算进行近似处理，从而降低计算复杂度。主要分为：

读取模型ckpt文件

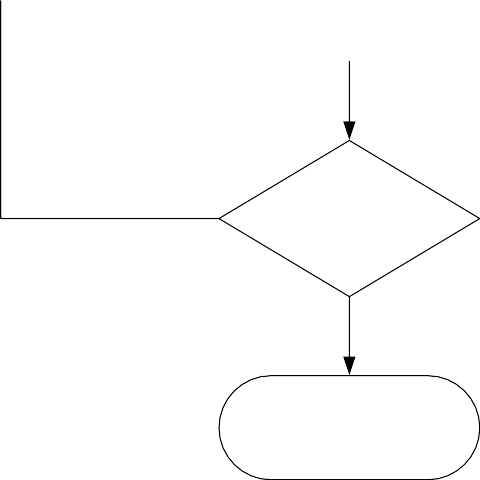
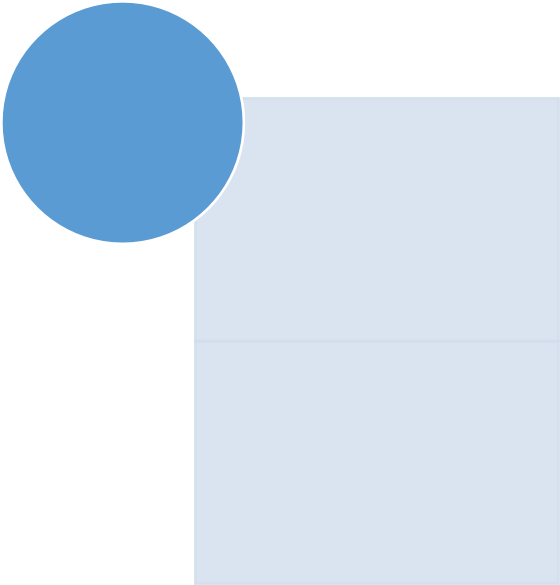
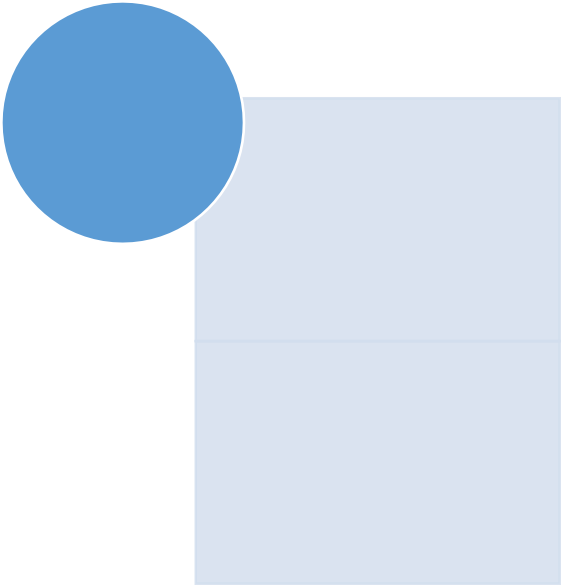
修改模型层

* + 张量分解

Fine tune训练

调整压缩方案

* + 矩阵分解



SVD奇异值矩

阵分解

- 适用于全连接

层优化

-大矩阵乘法分解成小矩阵乘法，

降低计算复杂度

Tucker张量

分解

-适用于卷积层优

化

-将大卷积分解成多个小卷积，降

低计算复杂度

否

是否达到精度

是

结束

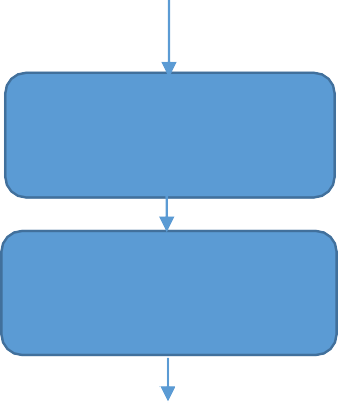
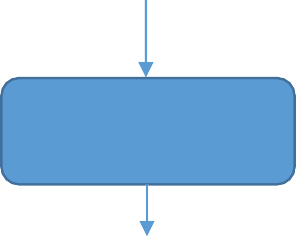
推理测试

优化具体实现案例：

* + 某NLP模型最后一层全连接层分解后，性能提升1.05~1.1x，精度完全不受影响
  + 某超分辨率模型实现tucker张量分解，性能提升约1.9x，精度为

SSIM=99.64%（最大值为1）

* + 还有一些其他分解办法可以考虑，但是效果或许很有限，尤其是矩阵分解，不是主流热门压缩方案

**矩阵分解**

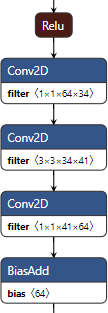
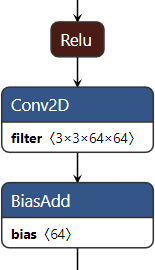
Matmul (m,n)\*(n,k)

Matmul

(m,n)\*(n,r)

Matmul

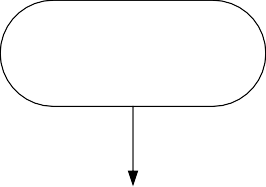
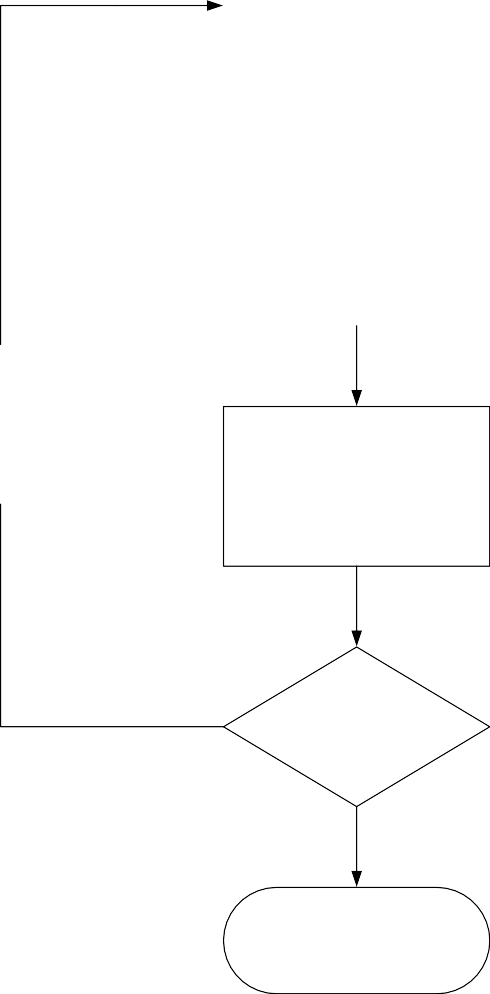
(m,r)\*(r,k)

**Tucker分解**



权值矩阵（n,k）被分解成（n,r）\*(r,k)，1个

matmul分解成2个matmul 卷积被分解成3个小卷积

模型剪枝，核心思想是将权重矩阵中相对“不重要”的权值剔除，然后fine tune进行微调，从而保证精度的前提下，减小网络大小，降低计算复杂度，提升速度。分为：

推理测试

否

是否达到精度

是

结束

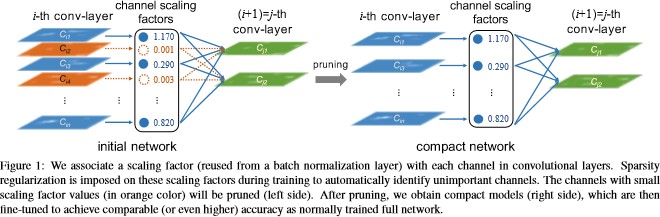
调整剪枝方案

Fine tune训练

读取模型ckpt文件

修改模型层

* 结构化剪枝，主要针对卷积层，其中又分为Channel Pruning和filter Pruning，Layer Pruning，压缩效果可控，不依赖于平台
* 非结构化剪枝，常见的是稀疏化剪枝，粒度更细，适用于卷积层和全连接层，但是运行速度依赖于平台实现，主要是稀疏矩阵乘法

Filter Pruning实现原理：

考虑到batch\_normal层的尺度变化参数scaling factor (gamma)对整个特征图的影响，所以加一个 L1 norm根据尺度变换值的大小来进行剪枝，进而实现对下一层filter中kernel个数的删减，达到压缩网络的目的。如右图所示。

案例分享：

针对超分辨率模型使用该剪枝方案进行剪枝，速度提升3.5x，参数量为原来的1/4，精度为

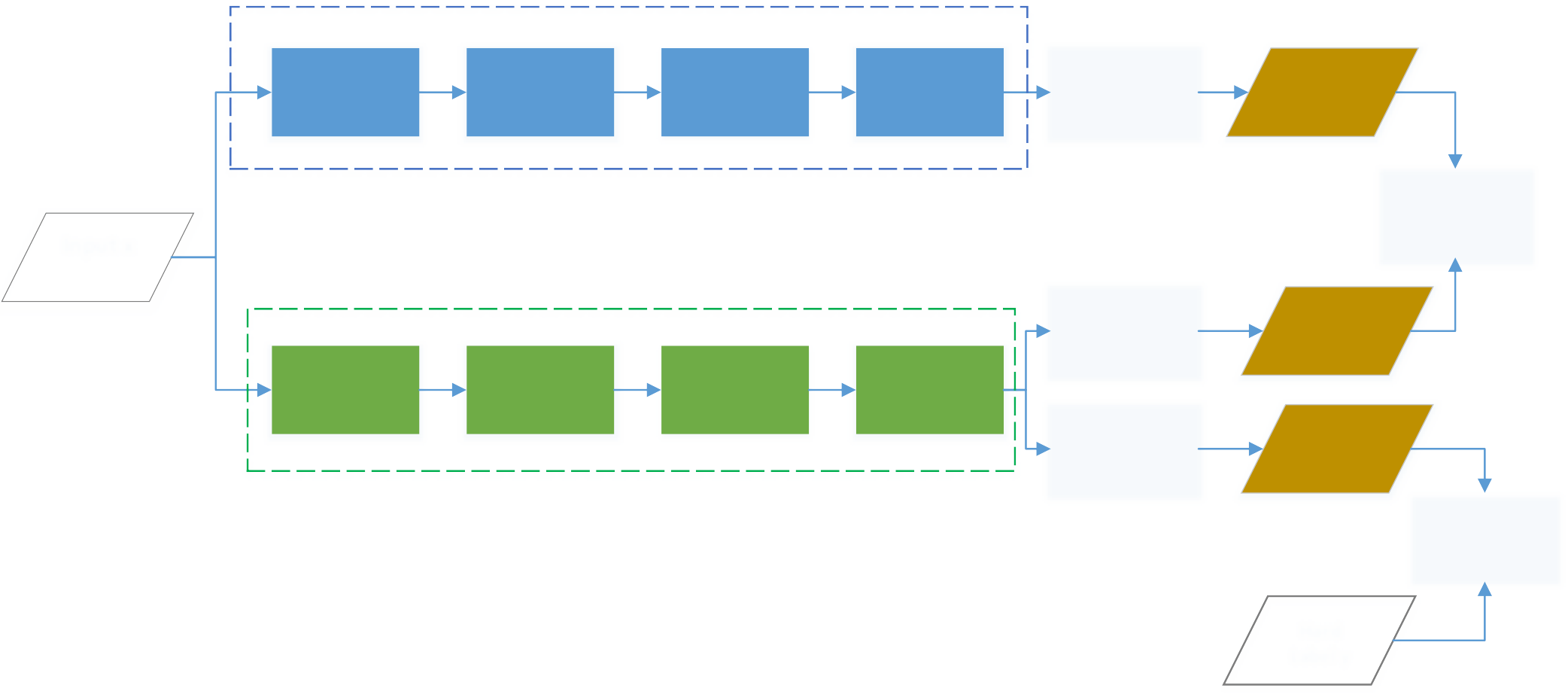
SSIM=0.9962（最大值为1）

参考论文：Learning Efficient Convolutional Networks through Network Slimming <https://arxiv.org/abs/1708.06519>

#### 模型压缩-模型蒸馏

模型蒸馏的一般流程：

### 依赖于重新训练

**Teacher Model**

**⋯⋯⋯**

**Layer 2**

**Layer 1**

**Soft Labels**

**Softmax(T=t)**

**Layer m**

**Input x**

**Softmax(T=t)**

**Distillation Loss**

**Student Model**

**Layer n**

**Loss Fn**

**Soft Prediction s**

**Hard Prediction**

**Softmax(T=1)**

**⋯⋯⋯**

**Layer 2**

**Layer 1**

**Student Loss**

**Loss Fn**

**Hard Label y**

内容大纲

* + 简介

# 优化方法论

* + - 模型压缩

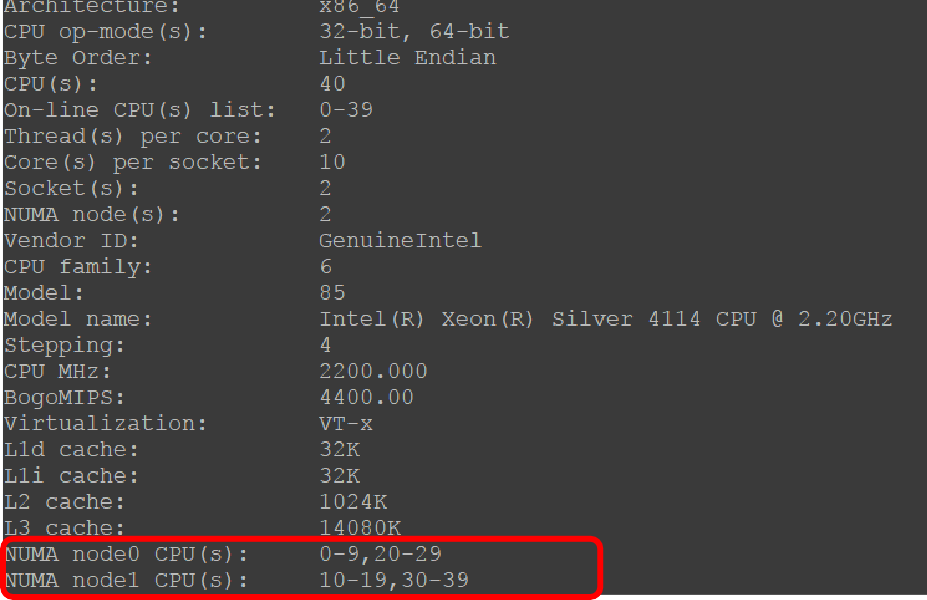
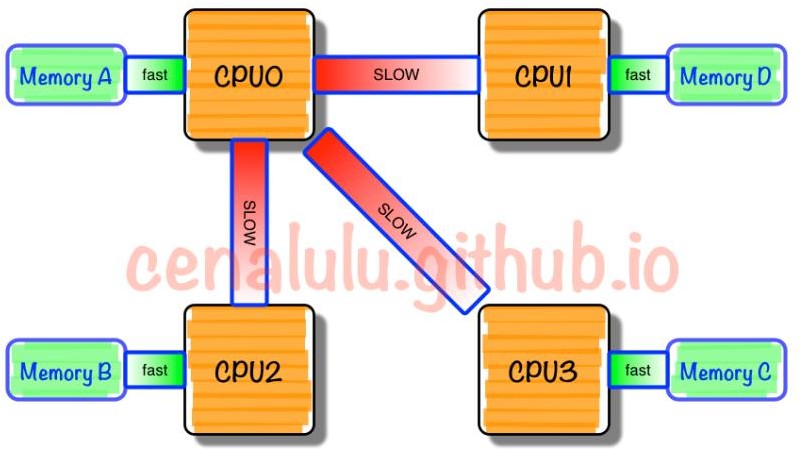
###### 系统级优化

* + - 案例分享

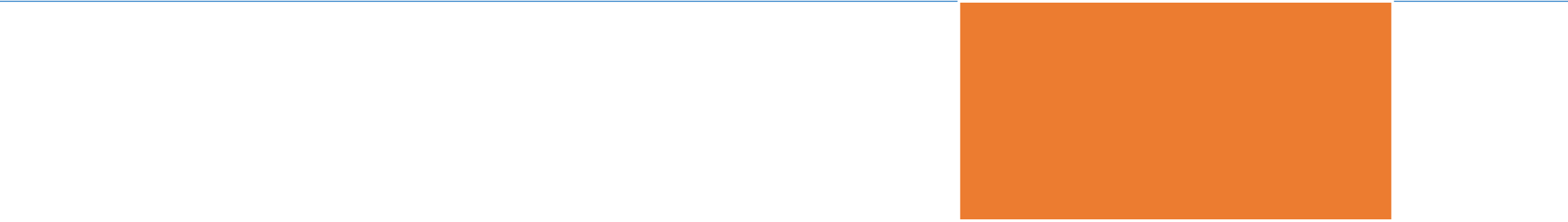
## 总结



Hardware Level: 线上docker环境差异

**Non-uniform memory access** (**NUMA**) is a [computer memory](https://en.wikipedia.org/wiki/Computer_storage) design used in [multiprocessing](https://en.wikipedia.org/wiki/Multiprocessing), where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own [local memory](https://en.wikipedia.org/wiki/Local_memory) faster than non-local memory (memory local to another processor or memory shared between processors).

**lscpu查看本机CPU信息：**

* **2个NUMA node**
* **每个numa node 10个物理核，20个逻辑核，因为超线程开着**
* **（0，20）号逻辑核位于同个物理核，（1，21）依次类推**
*  线上生产环境默认采用CPU时间片轮询的方式，docker实例不会固定在某些CPU核，而是随机跳到空闲的CPU核，性能不好

Hardware Level: 线上docker环境差异(cont.)

Docker环境的性能影响因素：

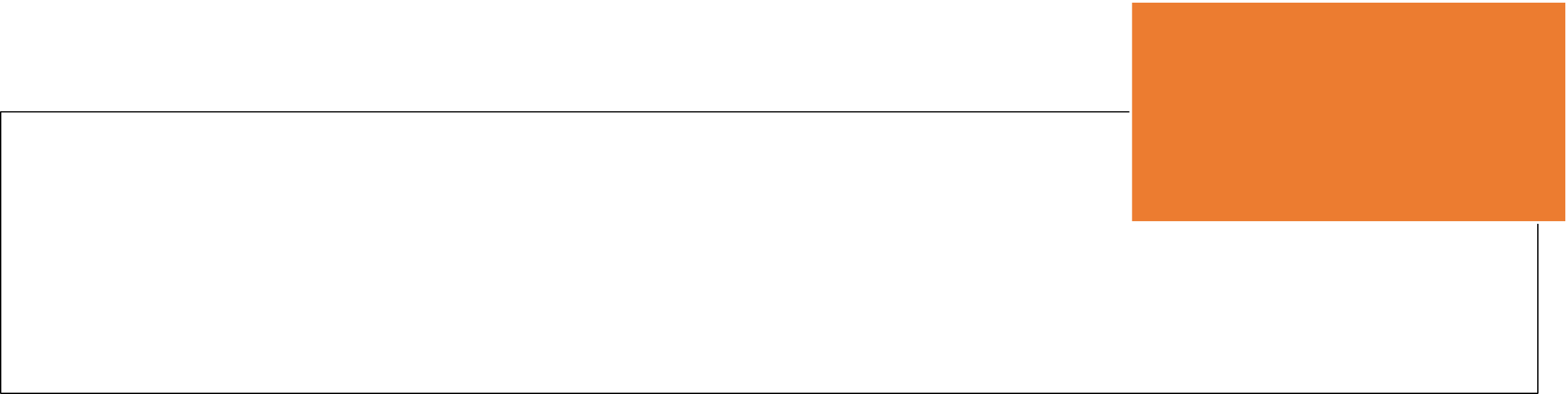
* Docker实例的CPU资源分配方式：

**取决于部署环境和系统资源分配方式**

* 修改为CPUSET独占模式，docker实例固定在几个核心运行，实测性能提升到原来的1.5x
* 即便设置CPUSET模式，绑定方式不同也会导致性能差异，确认docker实例分配到的核心编号，关键点：
  + 超线程打开，所有分配的核心都是逻辑核
  + 登陆到生产环境，查看文件确认分配的逻辑核编号 cat /sys/fs/cgroup/cpuset/cpuset.cpus
  + lscpu查看当前CPU的核心分布，从而判断所分配到的逻辑核如何跟物理核相对应
* 线下在物理机搭建Docker环境，尽量模拟线上生产环境，采用两种不同的方式绑定CPU，实测：方式1/方式2=1.3x,
  + 方式1：docker run --name tf -it --rm --cpuset-cpus=0-7 0-7逻辑核分布在不同物理核
  + *方式2：docker run --name tf -it --rm --cpuset-cpus=0-3,20-23， (0,20),(1,21)(2,22)(3,23)两两共用一个物理核*
* 线上服务器CPU配置如果不统一，支持的硬件指令集不同，对性能影响很大，发布产品申请资源时需要确定所申请的

CPU配置

#### 系统级优化-运行时优化（To be Updated）

* 通过设置环境变量提升性能, 环境变量设置跟运行环境关系密切，对**性能影响很大**，以下设置对当前环境有效，其他环境需要测试调整寻找最佳方案,尤其是线程数设置和KMP\_BLOCKTIME设置
* 编译高性能的MKL库（支持DIRECT\_CALL专门针对小矩阵的优化), 使用不同版本的MKL依赖库
* 案例：对于绝大多数模型，基于CPU平台，这些性能影响因素都有效

**需根据依赖库的要求更新运行时设置**

KMP\_BLOCKTIME=0 #**设置openmp线程同步等待的时间** KMP\_AFFINITY=granularity=fine,warnings,compact,1,0 #**线程亲源性设置** KMP\_SETTINGS=false #**关掉设置打印信息，对性能没影响**

OMP\_NUM\_THREADS=8 #**设置OpenMP线程数，MKL会自动决定使用多少线程，但不超过该上限**

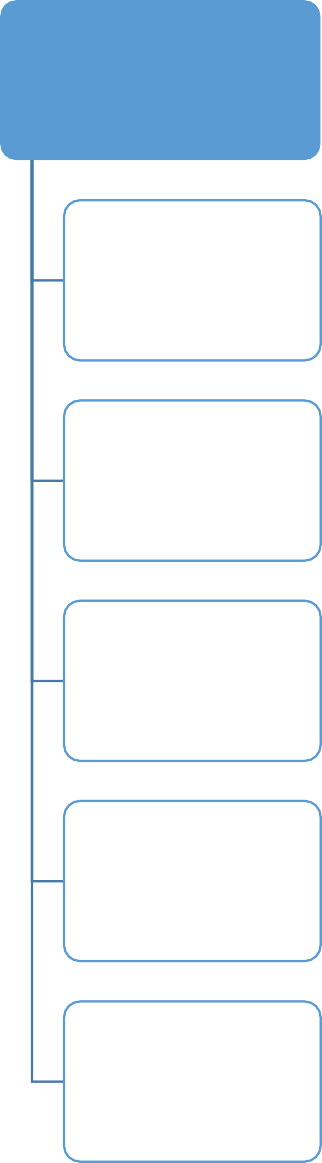
intra\_op\_parallelism\_threads=8 #**设置op内部的线程数**

inter\_op\_parallelism\_threads=1 #**设置op之间的线程数**

MKL\_ENABLE\_INSTRUCTIONS=AVX512 #**指定MKL使用的指令集，默认情况mkl使用当前机器能支持的最高指令集，但未必性能最优**

Notes：本页所有数据都基于线下物理机的模拟docker环境

More info: <https://www.tensorflow.org/guide/performance/overview?hl=zh-cn>



修改框架源码

算子融合

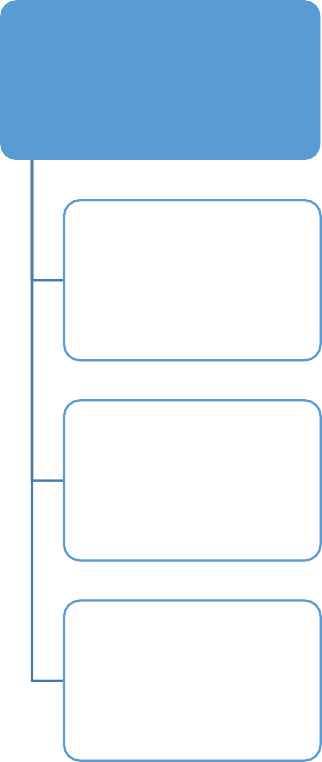
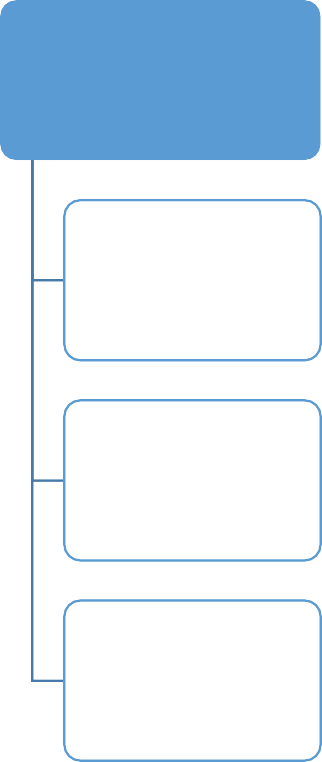
自定义算子重写

自适应混合使用多

种高性能算法

多种数学引擎混合

C++完全重写

框架层优化是主要优化手段，成功案例包含：

使用高性能推

理框架

OpenVINO

TensorRT

ONNX Runtime

Just-in-Time编

译技术

TVM

Intel Ngraph

Tensor

Comprehension

**当前的重点，通过修改框架和软件实现，最大化发挥硬件使用效率**

* **对算子深度优化**
* **提升访存效率**
* **指令集向量化**
* **使用新硬件特征**
  + Bert优化都是基于自定义算子重写/融合和其他优化相结合，运行速度提升明显
  + 某CV模型采用openvino方案，总体提升3~4x
  + OCR文字识别模型采用onnx和其他优化相结合，提升6~7x
  + Winograd和直接卷积混合使用，对于yolov3和超分辨率模型都有效

案例：

批处理

* 收集请求实现batching
* 直接进行batching
* 提升吞吐
* 取决于具体模型

前处理/后处理

* Python多进程
* Python多线程
* JIT优化：Numba
* 不同模块pipeline
* 去除冗余计算

**模型外的优化**

* OCR文本识别应用前后处理耗时非常多，使用python多线程，性能提升明显，总体约7x
* OCR文字检测前处理使用numba优化，性能提升27.68x；使用多进程提升2.97x

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **基准数据** | **使用高性能的数学引擎和运行时优化** | **降低激活函数的计算复杂度** | **向量化指令重写算子和算子融合** | **批处理计算** |
| **延迟（ 时:ms）** | 160 | 143.9 | 126 | 64.6 | 45.6 |
| **相对于基准的加速比** | 1x | 1.11x | 1.27x | 2.48x | 3.51x |

4

3.5

3

2.5

加速比

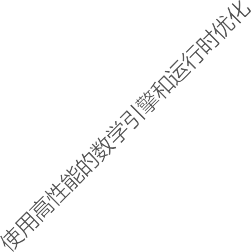
2

1.5

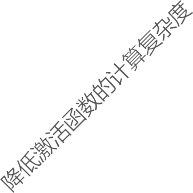
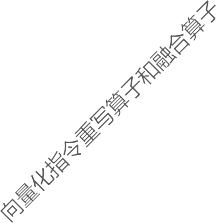
1

0.5

0

Bert性能优化

优化方案

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |  |  |
| 2.48 | | | | | | | | |  |
|  | | | | | | |  |  |  |
| 1 1.11 1.27 | | | | | | |  |  |
|  | | | | | | |  |  |
|  | | | | |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |

3.51



内容大纲

* + 简介

# 优化方法论

* + - 模型压缩
    - 系统级优化
    - **案例分享**

## 总结

#### 模型优化成功案例-某CV模型

美感度模型实验结果

基于inception的CV模型优化，精度损失

0.47%的情况下总体提升**3.42x**，精度损失

1.8%，性能提升**3.69x**；精度无损，性能提升

**3.16x**：

* 使用OpenVINO进行模型转换和推理
* 实现batching
* int8量化处理
* 图片预处理多线程优化

250

200

150

100

50

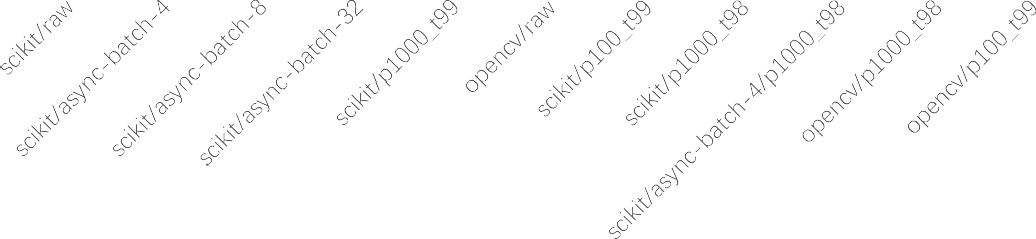
0

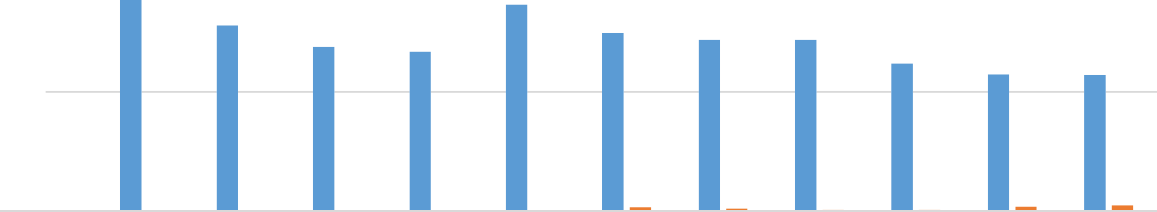
212









time dis score dis

~~88.7 86.7~~

78

69

67

74.8

72

72

62

57.4 57.1

2.4

1.8

0.47

0.47

1.04

1.5

0.3

0

0

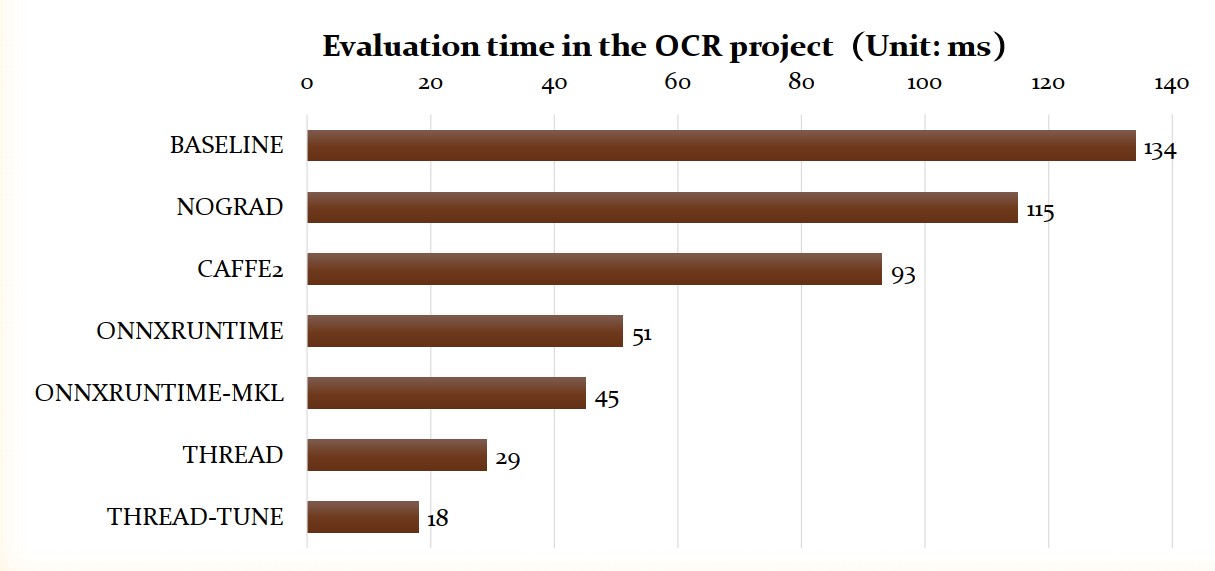
0

0

0

Score dis代表精度损失，总分100，time dis代表运行时间ms

OCR文本识别模块优化之后整体提升**7.44x**，优化方案包含：

* Pytorch代码改写，去除冗余计算调用更高效的接口
* ONNX runtime的选择
* Python多线程处理
* 使用MKL数学库和运行时优化
* 非模型区域，图片处理过程优化

使用了模型压缩和系统优化相结合，性能提升**8.5x**。优化方案包含：

* + 模型剪枝：filter pruning
  + 低秩估计：基于tucker张量分解
  + Winograd卷积算法
  + MKL高性能数学库和运行时优化
  + 模型蒸馏

#### 总结

模型压缩

降低计算复杂度

系统级优化

AI高性能

提升硬件利用率

强大的硬件基础

提供硬件算力

Base model

