

Методы ускорения инференса нейронных сетей на примере видеоаналитики

Шалимов Александр, 2023

Кто я и чем занимаюсь

- Инженер машинного обучения в Inventos
- Специализация - компьютерное зрение
- Разработка и поддержка систем видеоаналитики
- Аспирант в ОГУ им. Тургенева
- Сфера научных интересов: обнаружение аномалий в CV



План доклада

1. В каких задачах требуется оптимизация нейронных сетей
2. Наша задача: Обнаружение автомобилей в видеопотоке
3. Пример оптимизации нейронных сетей в видеоаналитике и её экономический эффект
4. Типовые проблемы видеоаналитики
5. Подход к решению задачи: Использование YOLO X
6. Классификация техник оптимизации
7. Выбор метрик для оптимизации
8. Оптимизация нашего решения: Как мы ускорили инференс детектора **в 4 раза!**

В каких задачах требуется оптимизаций нейронных сетей?

Нейронные сети играют ключевую роль в многих задачах.

Тип оптимизации может быть разным:

- **Розничная торговля:** Оптимизация для анализа покупательского поведения на устройстве
- **Обработка VOD контента:** Максимизация пропускной способности для обработки большого объема видео
- **Аналитика автомобильного трафика:** Снижение задержки для моментальной обработки данных от камер
- **Системы безопасности:** Быстрое реагирование и высокая точность

В докладе поговорим об оптимизации инференса



Источник данных с которыми мы работаем

- Камеры наблюдения за дорожным движением
- Видеопоток с высоким разрешением
- Различные ракурсы, погодные условия и условия освещенности



Наши задачи: Аналитика автомобильного трафика

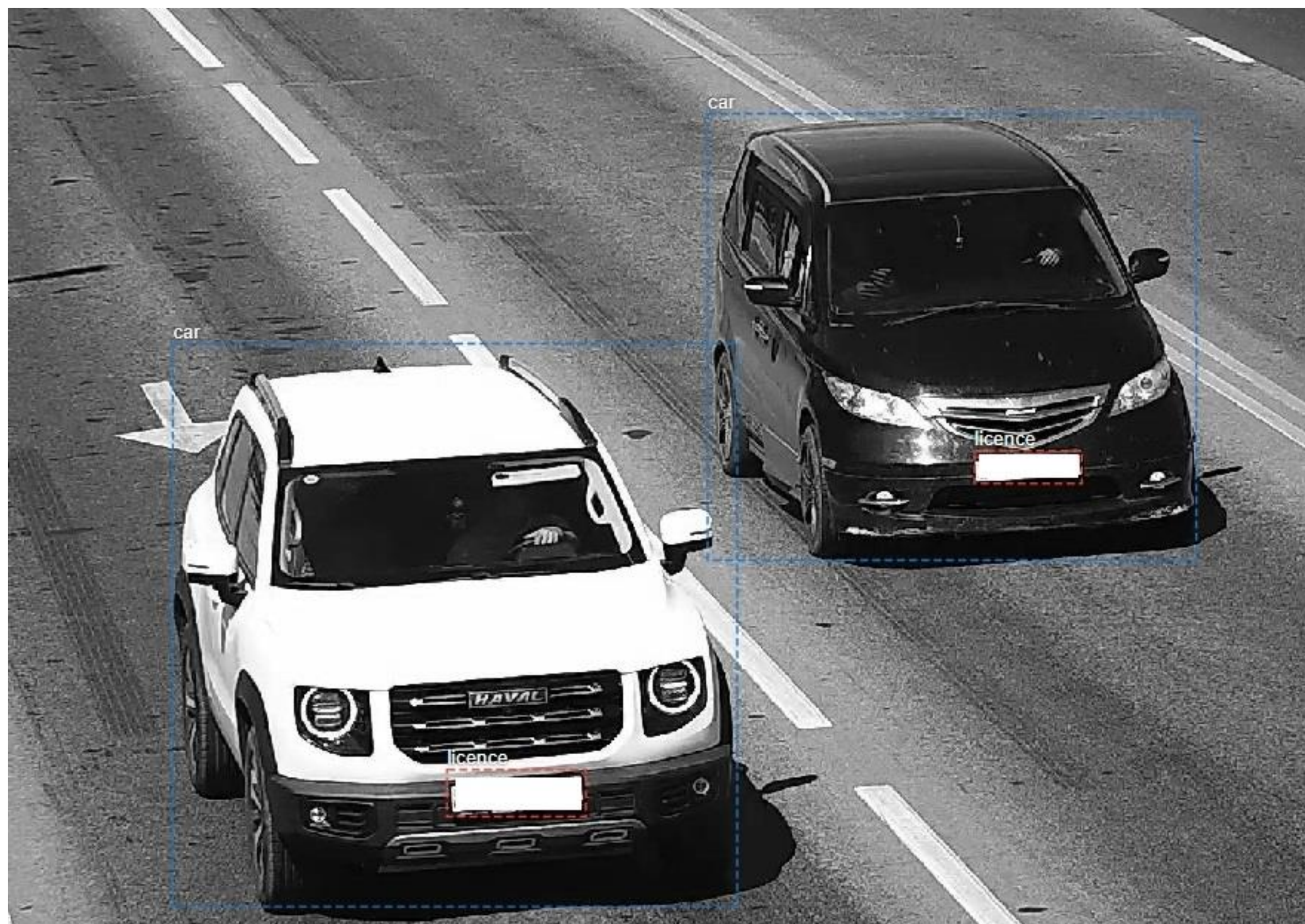
- Обнаружение ТС
- Сопровождение ТС
- Классификация ТС
- Распознавание ГРЗ

Особенности:

- Выполнение на устройстве (ограничены в ресурсах)

Цель оптимизации:

- Требуется ловить нарушителей на большой скорости (не можем скипать кадры)
- Необходимо **снижать latency** - время прогона нейронной сети



Еще пример: нейронные сети для анализа видеоконтента

Для чего используются нейронные сети:

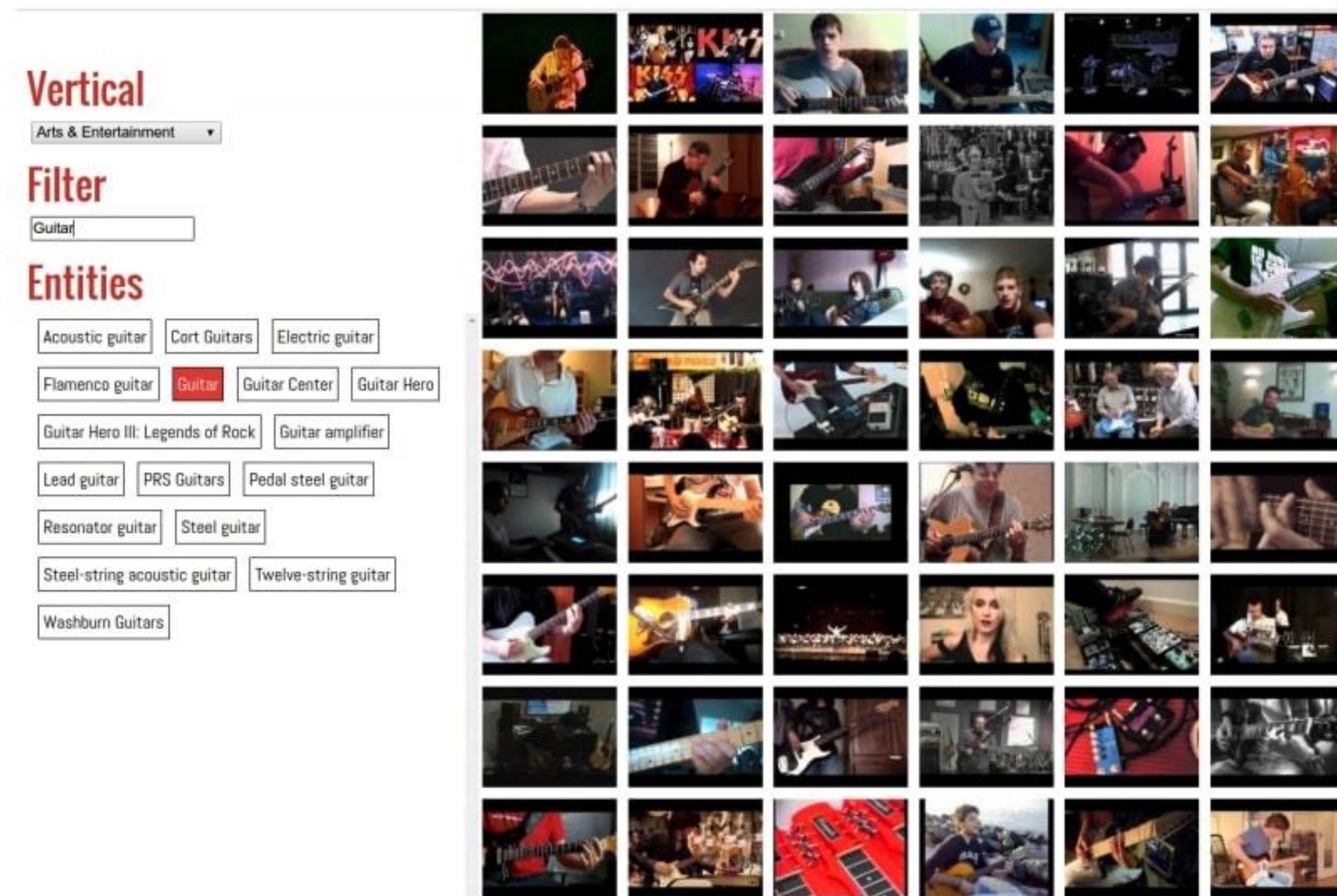
- Позволяют формировать точные тэги к видео.
- Помогают улучшить качество рекомендаций.

Особенности:

- Выполняется на сервере (много GPU)
- Оцифрованные данные

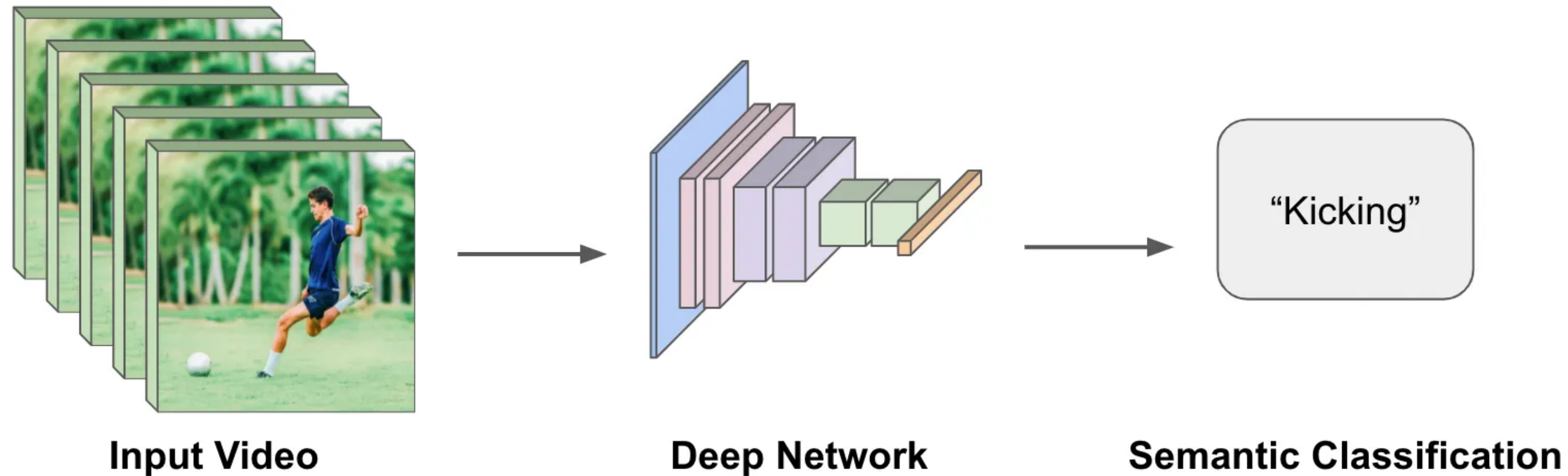
Цель оптимизации:

- Требуется обрабатывать большое количество роликов в сутки
- Необходимо **увеличивать throughput**
- Можно увеличивать и за счет ускорения инференса



Набор данных YouTube-8M

Применение оптимизаций в анализе видеоконтента

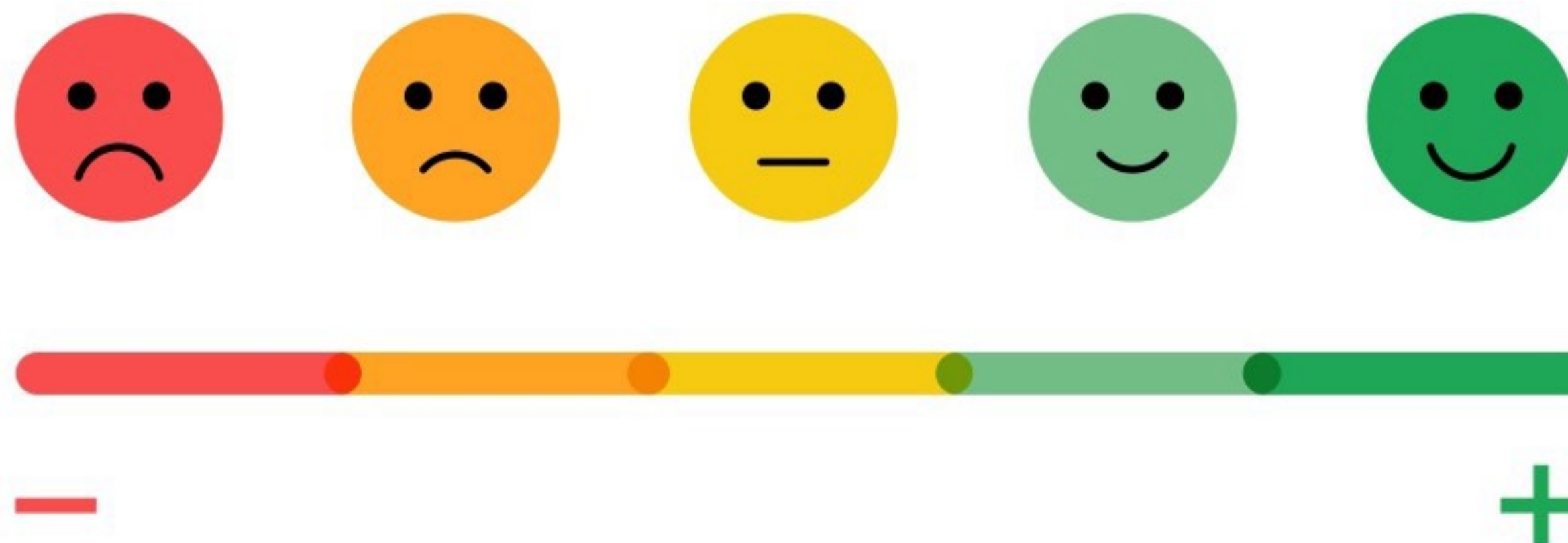


Одна из простых оптимизаций в архитектуре сети - **batching** (пакетная обработка)

Эффект оптимизаций:

- Увеличивает скорость обработки контента.
- Позволяют проводить аналитику в режиме работы, приближенному к реальному.
- Необходимое соблюдать баланс: **Качество** работы нейронной сети и **скорость** ее работы.

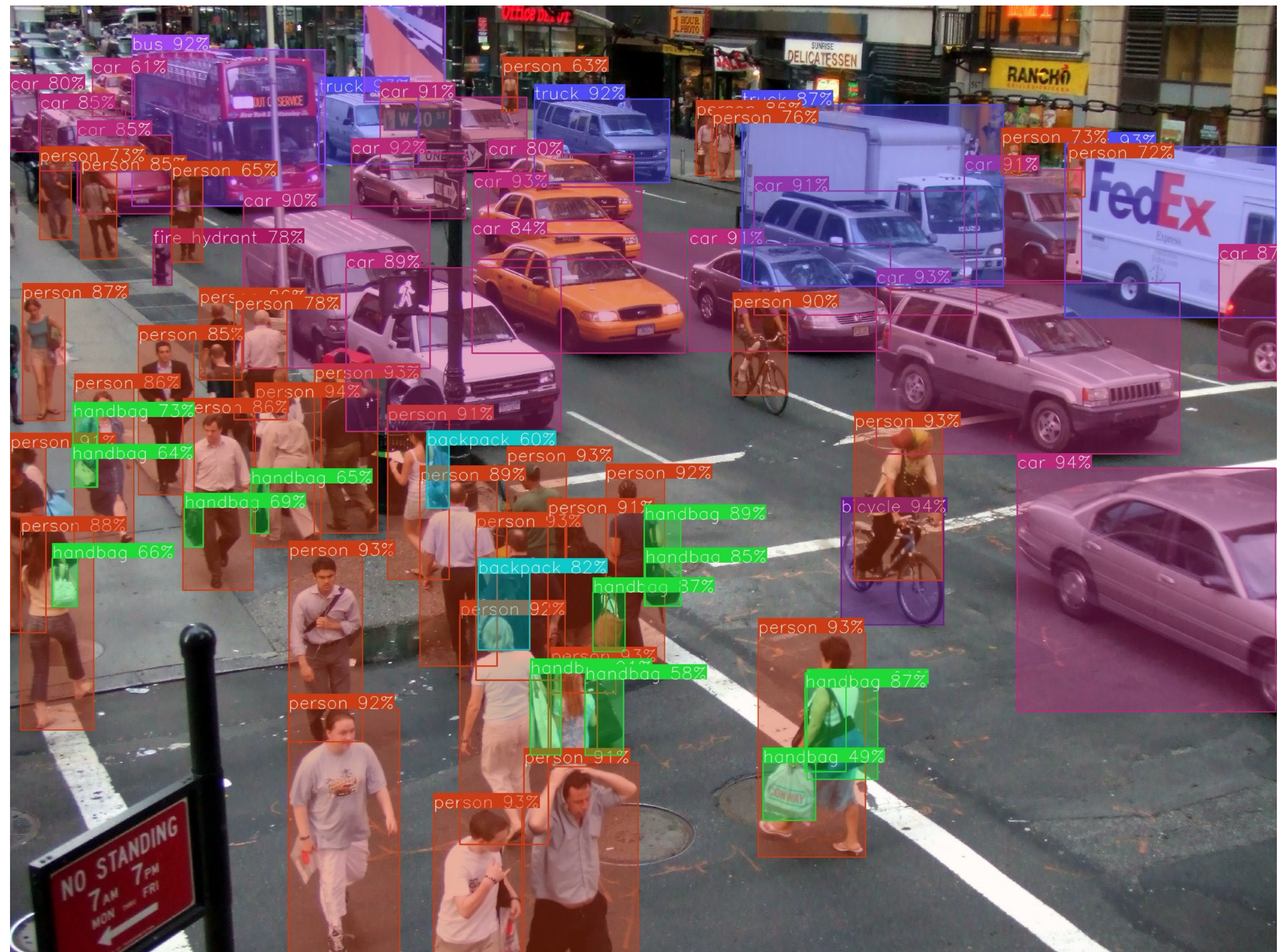
Экономический эффект оптимизации в анализе видеоконтента



- Уменьшается время обработки данных и снижается нагрузка на серверы
- Благодаря незначительному снижению качества сети, пользователи своевременно получают персонализированные рекомендации, что повышает их вовлеченность и удовлетворенность
- Высокая степень удовлетворенности пользователей приводит к увеличению их активности и времени, проведенного на платформе

Типовые проблемы в задачах видеоаналитики

- Работа с данными высокой размерности
- Обработка данных в режиме приближенному к реальному времени
- Различный размер объектов, перекрытия и изменяющиеся фоны
- Изменчивость качества видео



Концепция решения: Свёрточные нейронные сети (CNN)

Рассмотрим оптимизации, которые мы использовали
на примере обнаружения ТС



Применение CNN для задачи классификации ТС

Преимущества:

- Автоматически учатся иерархическим представлениям признаков
- Эффективны для задач обработки изображений и видео
- Устоявшаяся технология, хорошо работают с встраиваемыми системами

Недостатки:

- Слабая способность моделировать глобальный контекст

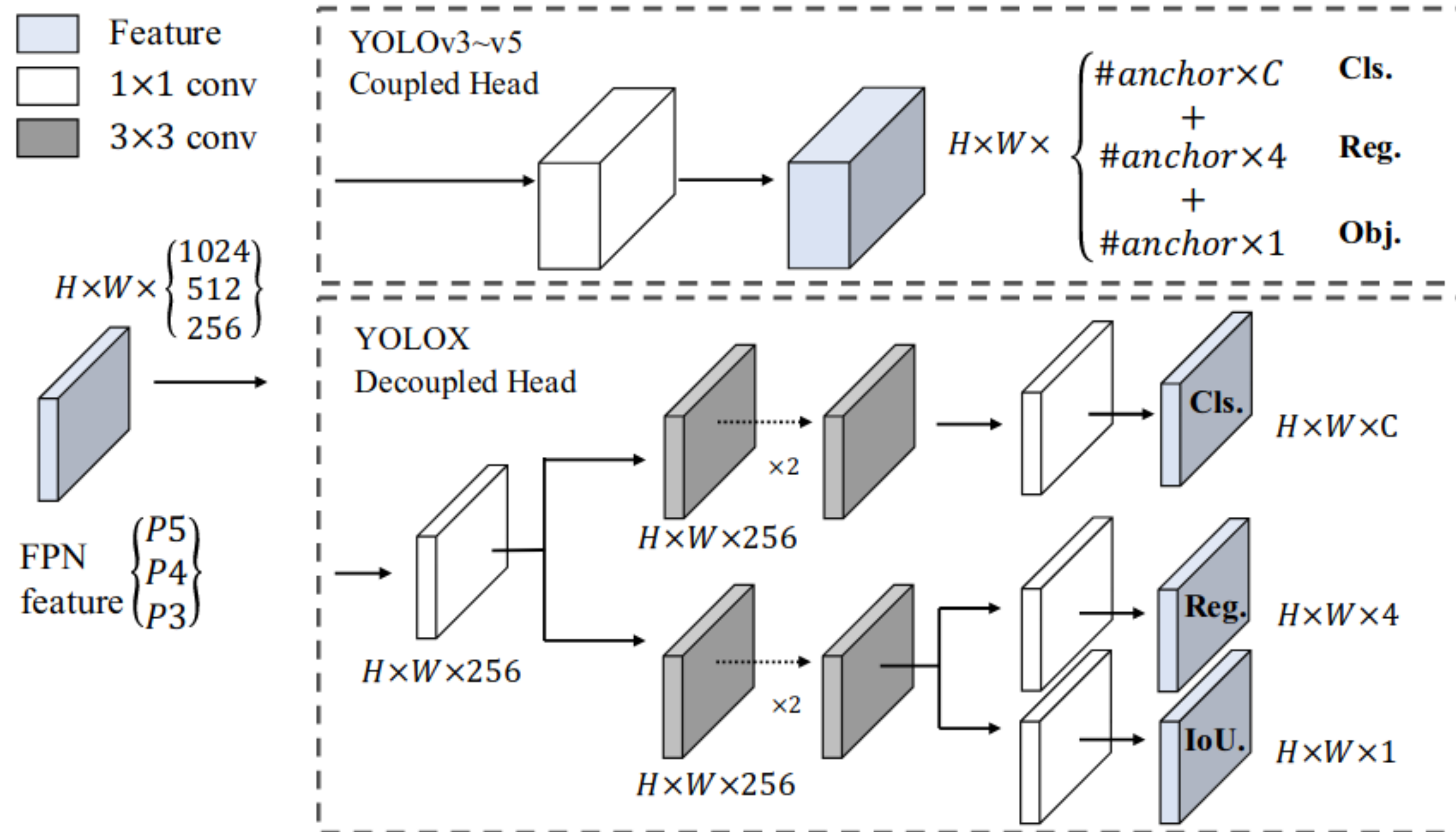
YOLO X: Система обнаружения объектов в реальном времени

Преимущества:

- Делает прогноз на основе всего изображения за одно выполнение сети (One-stage detector)
- Подход без якорей (Anchor-free) позволяет снизить число параметров
- Разделение голов под разные задачи: классификация, локализация, наличие объекта

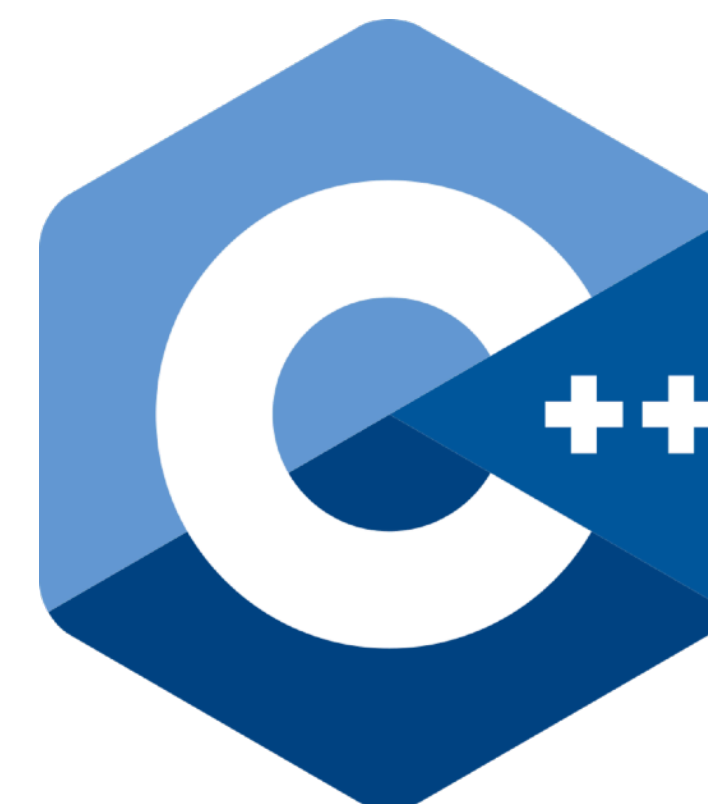
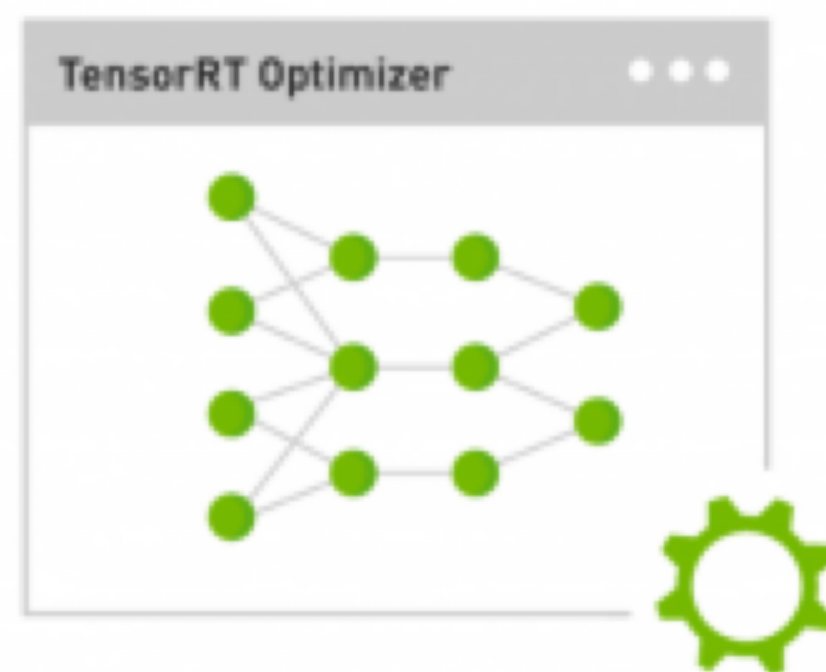
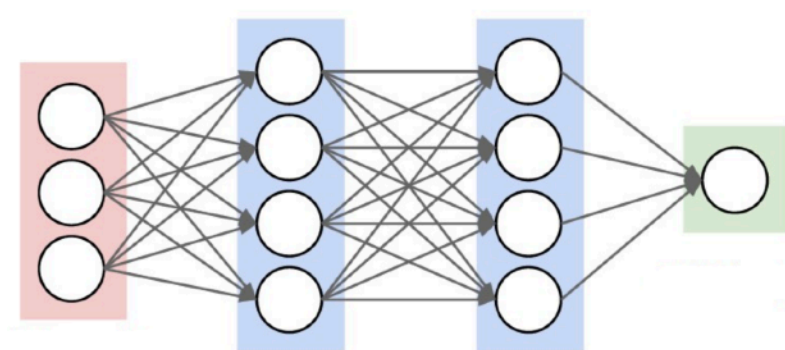
Недостатки:

- Mish активация - сильно замедляет выполнение
- Трудности с экспортом NMS внутри сети и необходимость реализации постпроцессинга



Разделение голов YOLO X в сравнении с объединенным подходом

Классификация подходов к оптимизации скорости одного прогона сети



Архитектура сети

Многозадачные архитектуры
Многомодальные архитектуры
Динамический размер пакета
Легковесные архитектуры
(MobileNet)

Выполнение сети

Квантизация сети
Оптимизация графа
выполнения
Прунинг
Дистиляция

Код

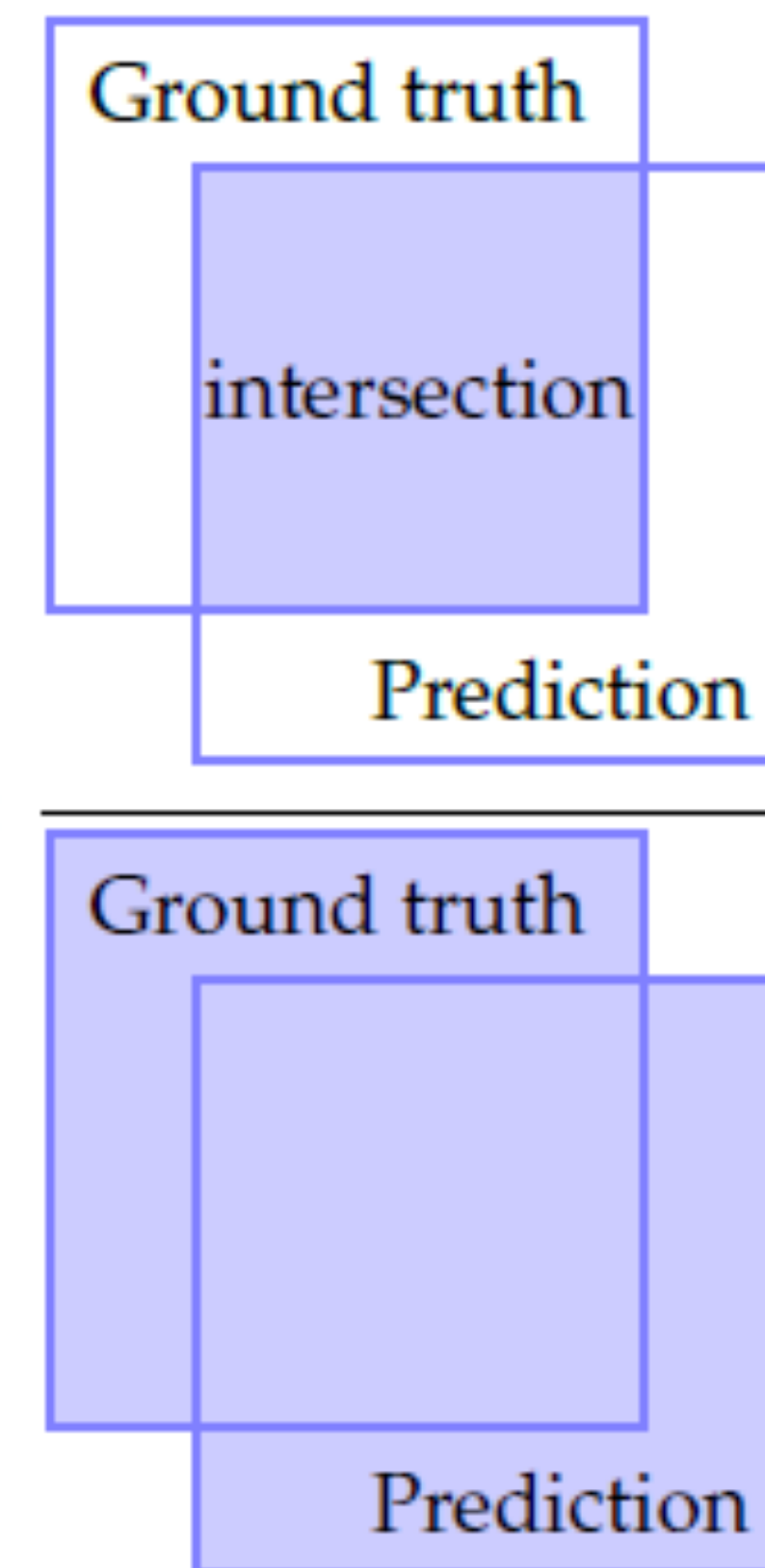
Контроль выделения видеопамяти
Препроцессинг и постпроцессинг
на GPU
C++ TensorRT API
C++ TensorRT Plugins
Triton Inference Server

Выбор метрик для оптимизации: IoU








Для детектора YOLO X рассматривались:

- пересечение над объединением (IoU)
- точность (precision) для порога IoU
- полнота (recall) для порога IoU

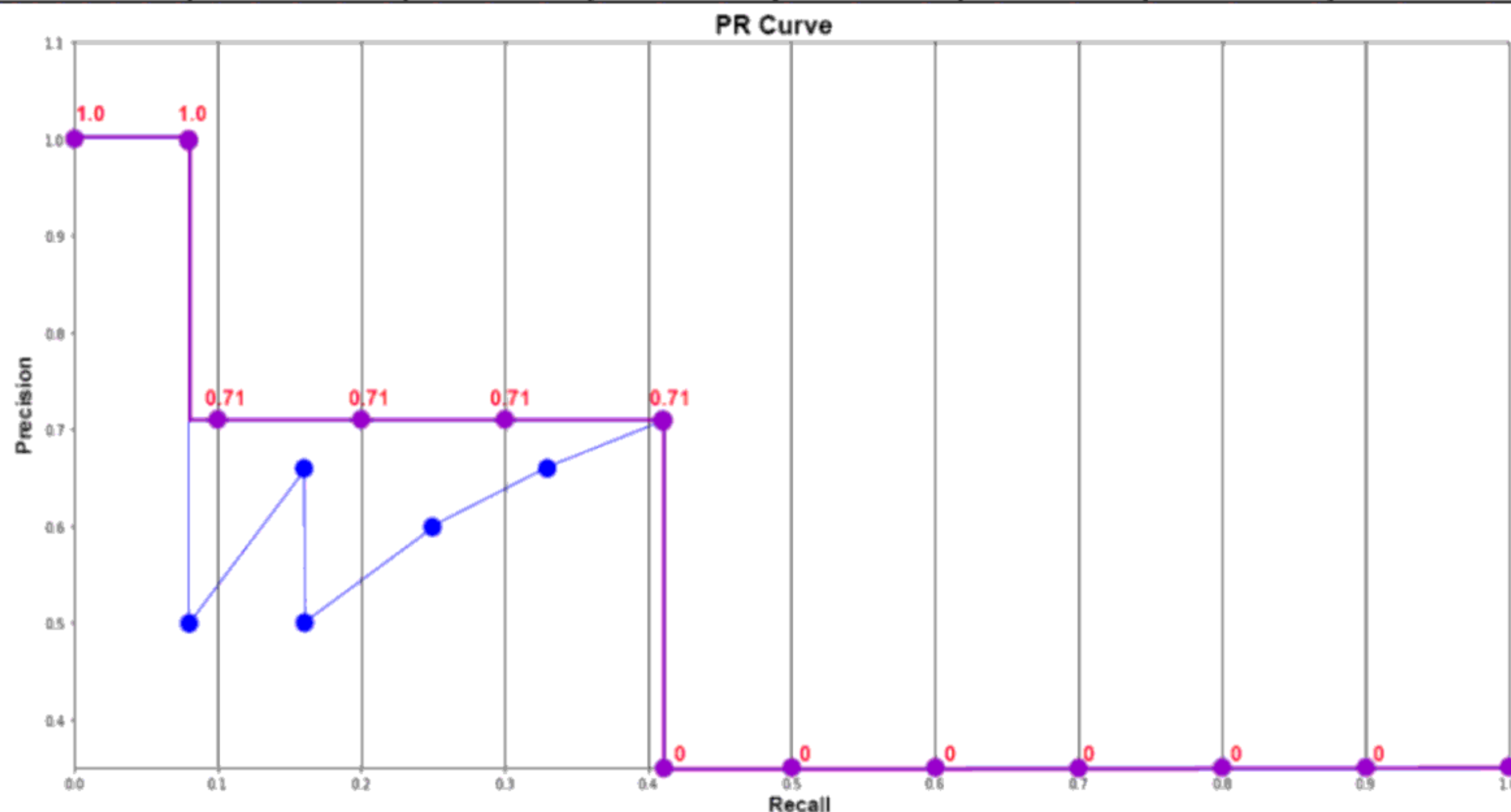
$$IoU = \frac{\textit{area of overlap}}{\textit{area of union}} =$$



Выбор метрик для оптимизации: AP

Detections							
Precision	1	0.5	0.66	0.5	0.6	0.66	0.71
Recall	0.08	0.08	0.16	0.16	0.25	0.33	0.41

Average Precision (AP) - площадь под precision-recall кривой



$$AP_{\text{dog}} = \frac{1}{13} \left(\sum_{i=1}^{13} P_i \right) = \frac{1}{13} (2 \cdot 1 + 4 \cdot 0.71 + 7 \cdot 0)$$

$$= \frac{1}{13} (2 + 2.84 + 0) = \frac{1}{13} \cdot 4.84 = 0.372 = 37.2 \%$$

Выбор метрик для оптимизации: mAP

Объединяющая их метрика: усреднение AP по всем классам - **mAP** для различных IoU.

Average Precision (AP):

AP % AP at IoU=.50:.05:.95 (primary challenge metric)
AP^{IoU=.50} % AP at IoU=.50 (PASCAL VOC metric)
AP^{IoU=.75} % AP at IoU=.75 (strict metric)

AP Across Scales:

AP^{small} % AP for small objects: area < 32²
AP^{medium} % AP for medium objects: 32² < area < 96²
AP^{large} % AP for large objects: area > 96²

Method	Backbone	Size	FPS (V100)	AP (%)	AP ₅₀	AP ₇₅	AP _S	AP _M	AP _L
YOLOX-DarkNet53	Darknet-53	640	90.1	47.4	67.3	52.1	27.5	51.5	60.9
YOLOX-M	Modified CSP v5	640	81.3	46.4	65.4	50.6	26.3	51.0	59.9
YOLOX-L	Modified CSP v5	640	69.0	50.0	68.5	54.5	29.8	54.5	64.4
YOLOX-X	Modified CSP v5	640	57.8	51.2	69.6	55.7	31.2	56.1	66.1

Цель оптимизации - получить **25 FPS** при незначительном ухудшении метрики **mAP**

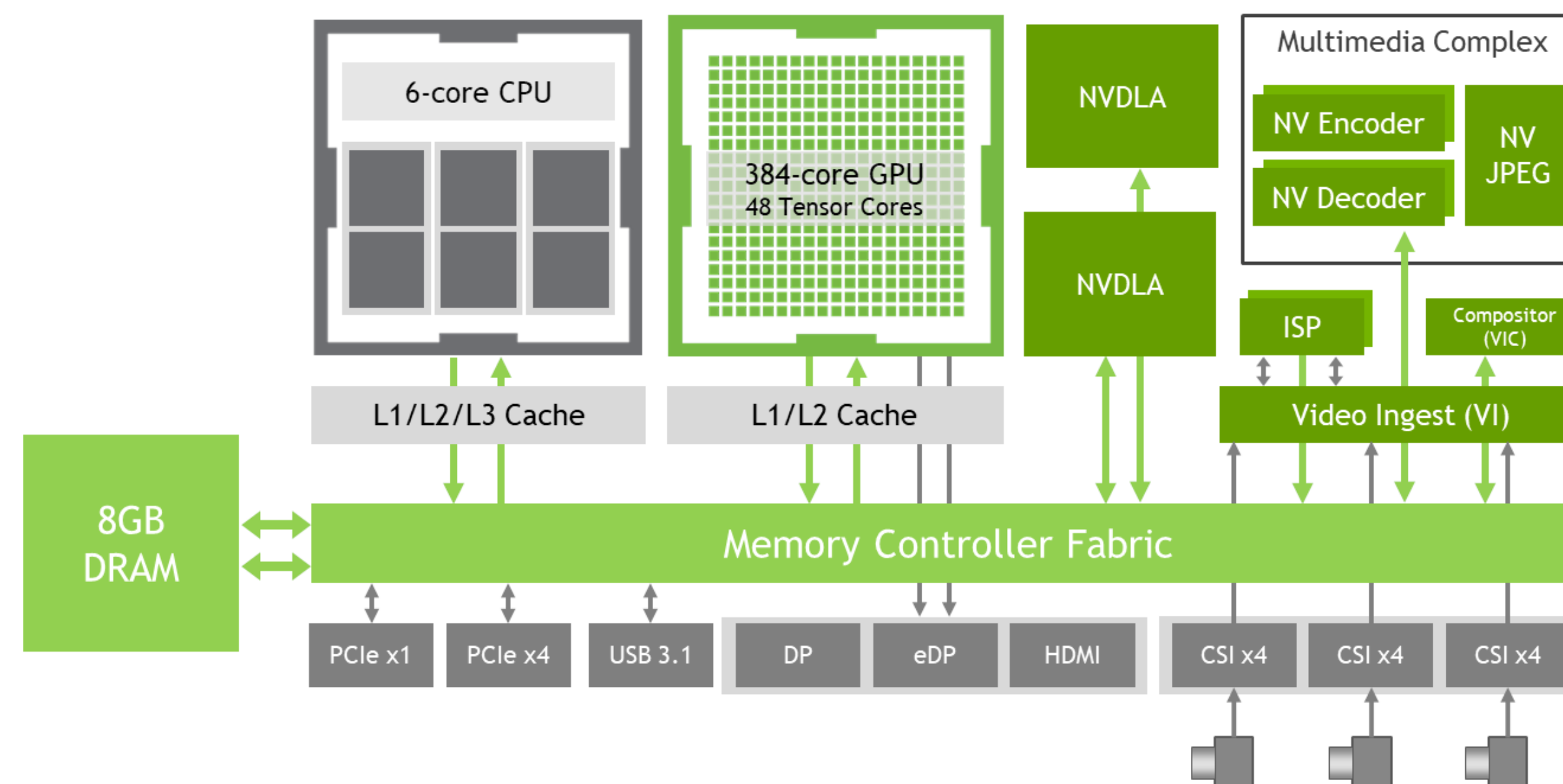
Выбор устройства исполнения: Jetson NX

Требования:

- Встраиваемая система
- Достаточная производительность в задачах машинного обучения
- Низкое тепловыделение

Решение: Nvidia Jetson Xavier NX

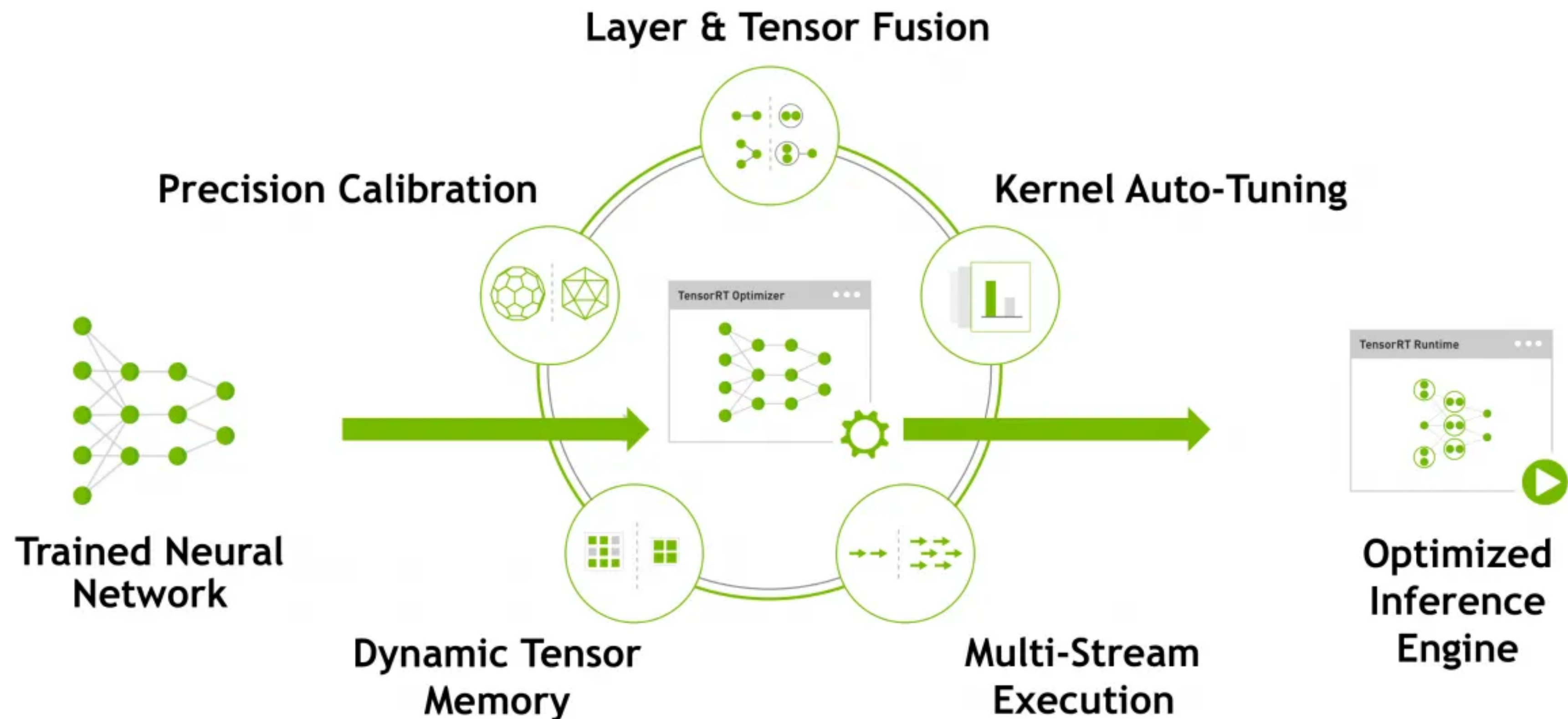
- GPU: 21 TOPS (INT 8), (384 ядра **CUDA** и 48 тензорных ядра) + 2 ускорителя глубокого обучения NVDLA
- CPU: Шестиядерный 64-разрядный процессор
- Память: 8 ГБ объединенной памяти (**unified memory**)
- Энергоэффективность: Потребление энергии 10-15W



Структурная схема компонентов Jetson Xavier NX

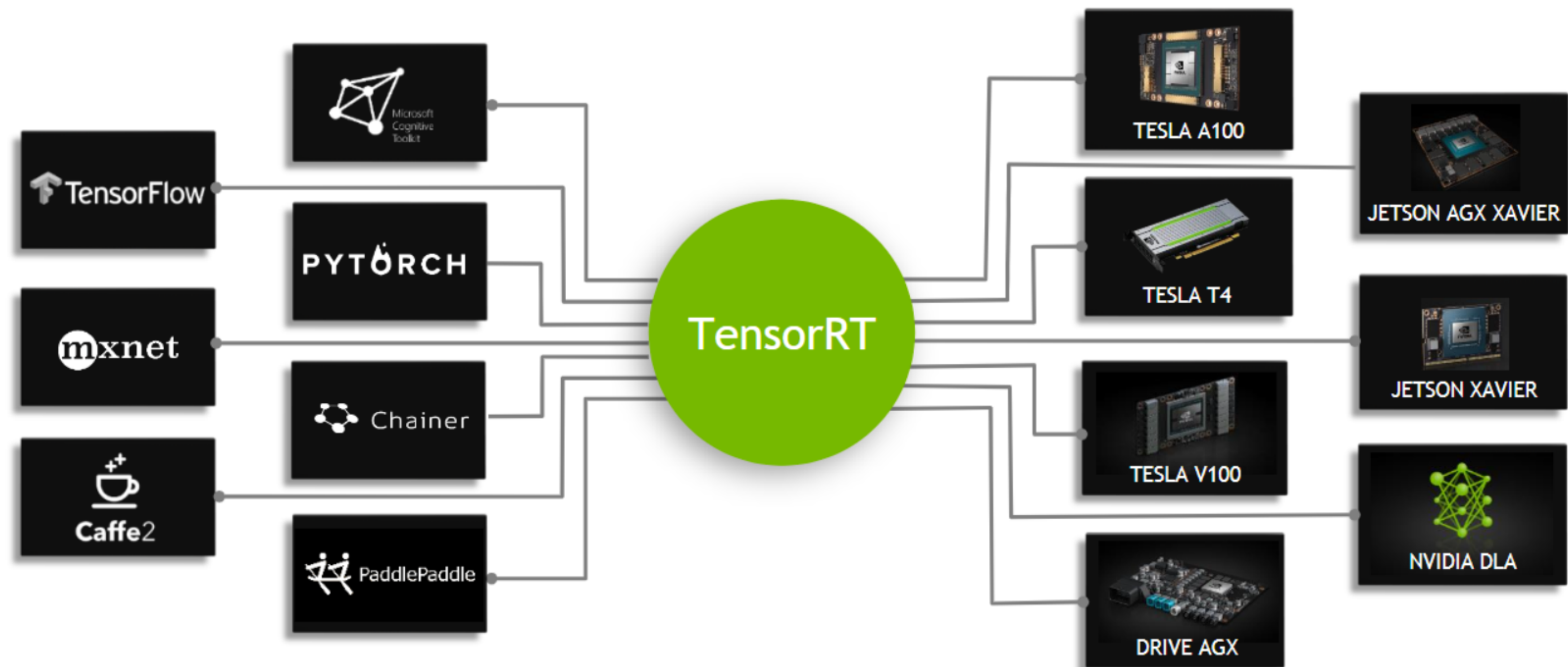
Первоначальная реализация: TensorRT + Python

TensorRT: Оптимизатор и среда выполнения, обеспечивающие **низкую задержку** и **высокую пропускную способность** для приложений глубокого обучения.



Первоначальная реализация: TensorRT + Python

- TensorRT трансформирует и оптимизирует обученные различными фреймворками модели в **inference engine**.
- Движок может быть сформирован для различных устройств Nvidia.
- Для каждого конечного устройства будет сформирован свой движок.



Первоначальная реализация: TensorRT + Python

Нейронная сеть: **YOLOX-S**

```
import torch
from torch2trt import torch2trt

model = model.cuda().eval()
x = torch.ones((1, 3, 640, 640)).cuda()
model_trt = torch2trt(model, [x], max_workspace_size=1 << 32)
torch.save(model_trt.state_dict(), 'model_trt.pth')
```

Пример экспорта в TRT

```
from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load('model_trt.pth'))
```

Пример инференса в TRT

```
x = torch.ones((1, 3, 640, 640)).cuda()
```

```
with torch.no_grad():
    y = model_trt(x)
```

Результат: **8 FPS** на Jetson NX

Повышение производительности нейронных сетей

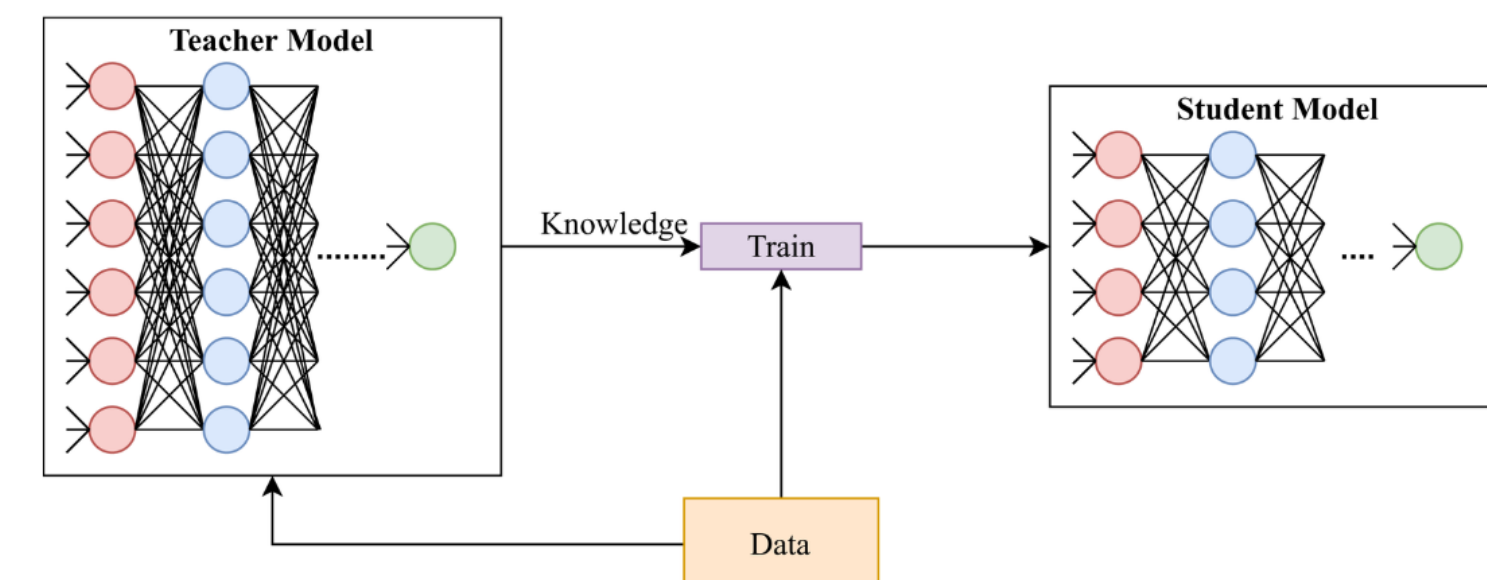
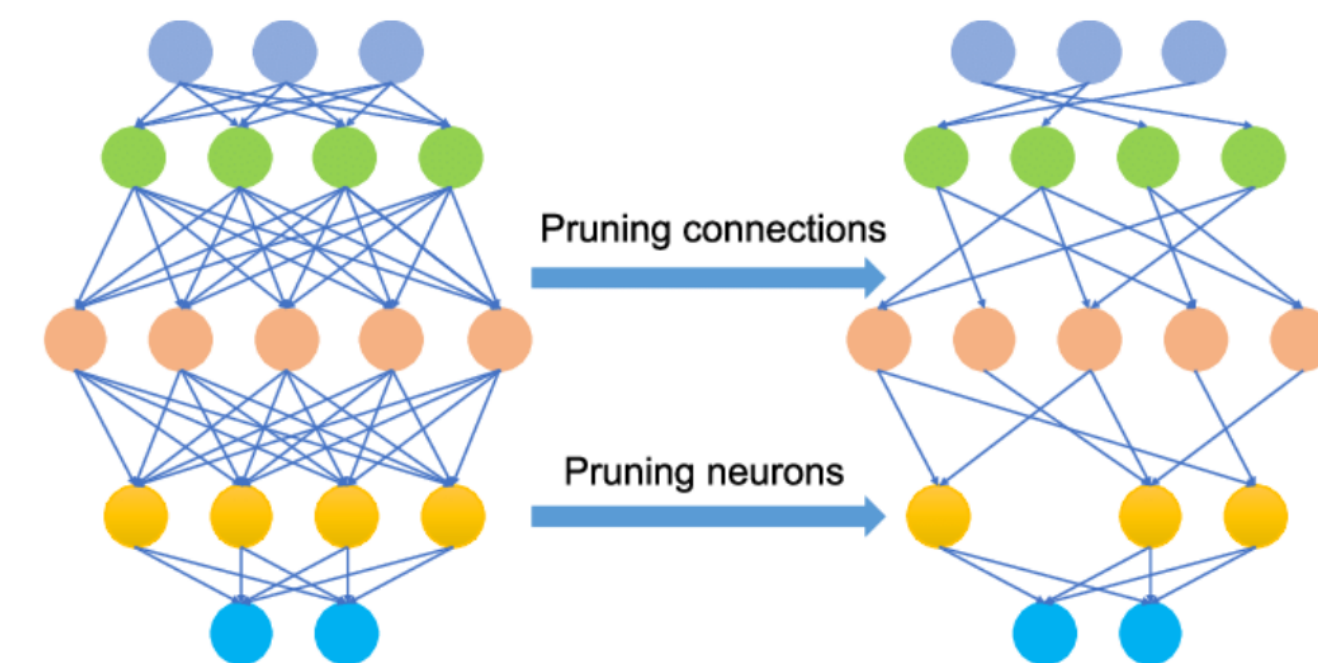
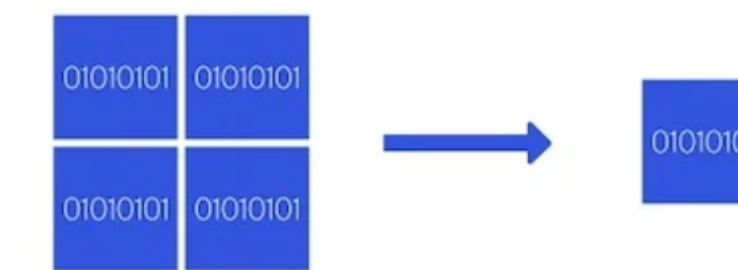
Универсальные методы оптимизации модели:

- **Квантизация** FP16/INT8, Quantization Aware Training
- **Прунинг** - процесс удаления «ненужных» весов модели. Позволяет значительно снизить размер модели, при этом слабо влияет на ускорение инференса.
- **Дистиляция** - передача знаний от одной сложной модели (учитель) к более простой и быстрой (ученик). В задаче с детектором можно использовать данные от **YOLOX-X** для создания дополнительной разметки.

Floating point Integer

3452.3194 → 45

32 bit 8 bit



Квантизация: простой способ ускорить работу сети

32-bit float (FP32)



$$(-1)^0 \times 2^{128-127} \times 1.5707964 = 3.1415927$$

16-bit float (FP16)



$$(-1)^0 \times 2^{128-127} \times 1.571 = 3.141$$

- **Точность (precision)** представления данных - краеугольный камень производительности модели.
- **FP32** обеспечивает высокую точность, но увеличивает размер модели и замедляет вычисления.
- Квантизация до **FP16** - компромисс между точностью и эффективностью, ключевой для встроенных систем.

Первоначальная реализация: TensorRT + Python + FP16

Нейронная сеть: YOLOX-S

```
import torch
from torch2trt import torch2trt

model = model.cuda().eval()
x = torch.ones((1, 3, 640, 640)).cuda()
model_trt = torch2trt(model, [x], max_workspace_size=1 << 32, fp16_mode=True)
torch.save(model_trt.state_dict(), 'model_trt.pth')

from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load('model_trt.pth'))

x = torch.ones((1, 3, 640, 640)).cuda()

with torch.no_grad(), torch.cuda.amp.autocast(enabled=True):
    y = model_trt(x)
```

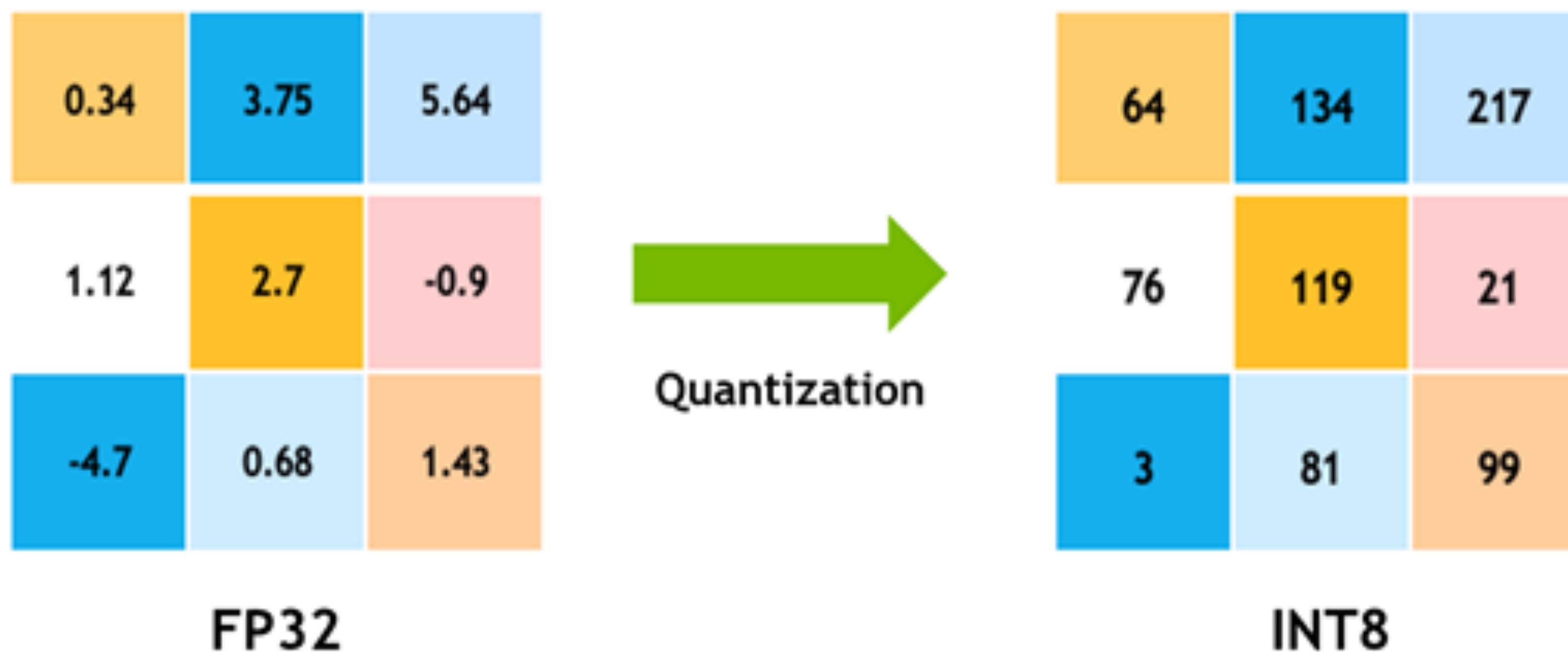
Включаем режим FP16 при
конвертации

Выполняем модель в FP16 режиме

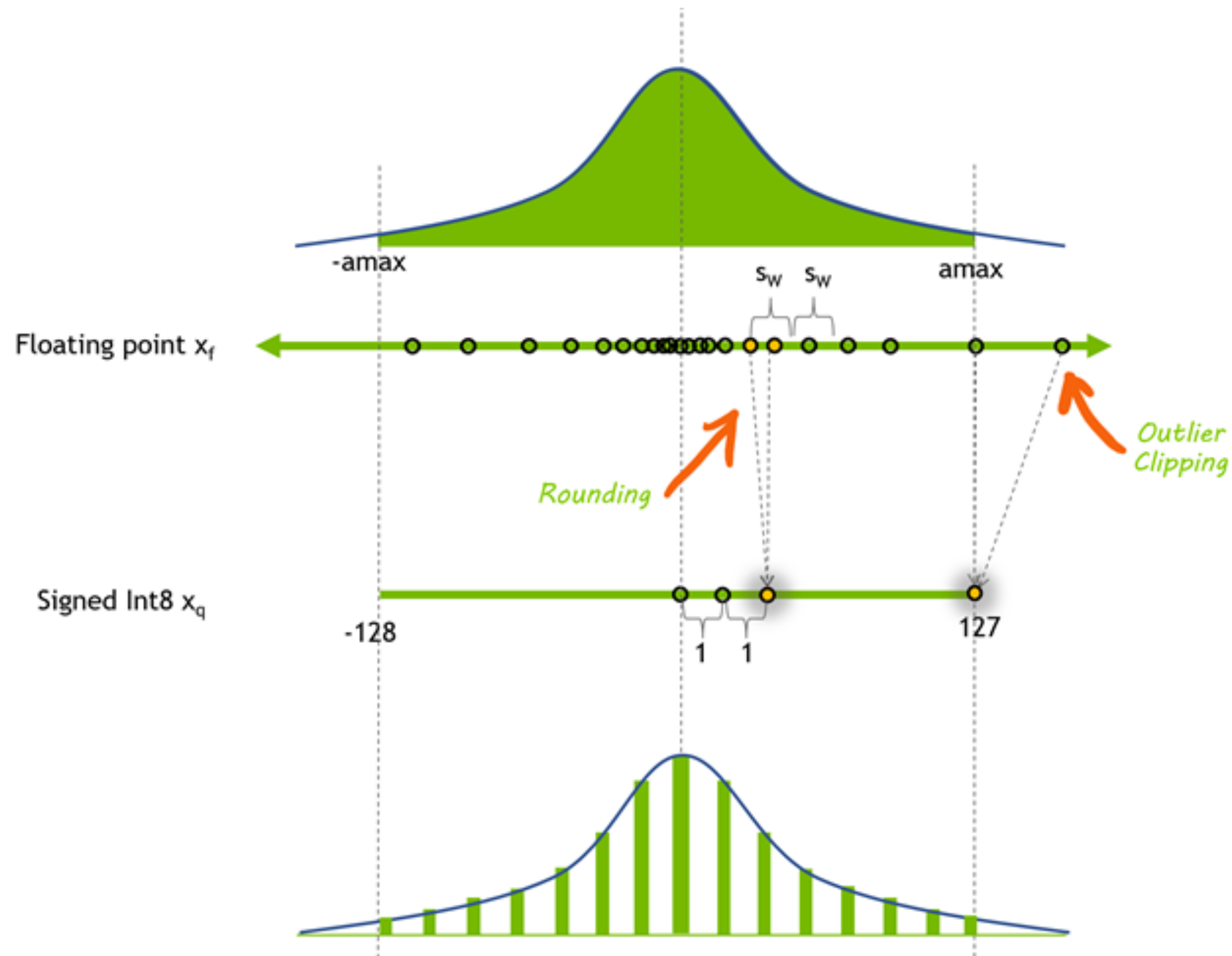
Результат: **16 FPS** на Jetson NX,
но mAP снизилась на **30%**

Квантизация INT8

- Использование **INT8** сокращает вычислительные ресурсы и объем памяти, необходимые для работы модели.
- Перевод чисел с плавающей точкой в целые **уменьшает точность**, что влияет на производительность.
- Для минимизации потерь в точности модели необходим процесс **калибровки**.



Квантизация INT8: Калибровка



- Калибровка нужна чтобы «упаковать» детальные данные о модели (FP32) в более компактный формат (INT8) без больших потерь качества.
- Запускаем модель на известных данных, собираем статистику, как модель реагирует на разнообразные входы.
- Ищем такие параметры сжатия, при которых различие между оригинальной и сжатой моделью минимально.
- В результате получаем **таблицу калибровки**, которая указывает как сжимать каждый слой, чтобы результаты оставались точными.

Квантизация: Наш опыт использования

- INT8 калибровка сильно ускоряет время выполнения, однако метрика mAP **снижается** значительно.

Одним из выходов может быть обучение большей версии модели.

- Самым эффективным для нас стало обучение с учетом квантизации или **Quantization Aware Training**.

Модель «привыкает» к ограничениям квантизации на этапе обучения.

- Его использование дает **16 FPS** на Jetson NX. Снижение mAP на нашем наборе данных с **80%** до **76%**.

Повышение производительности нейронных сетей за счет оптимизации кода

Понимание узких мест в текущей реализации:

- В Python, нет возможности гибко управлять выделяемой памятью.

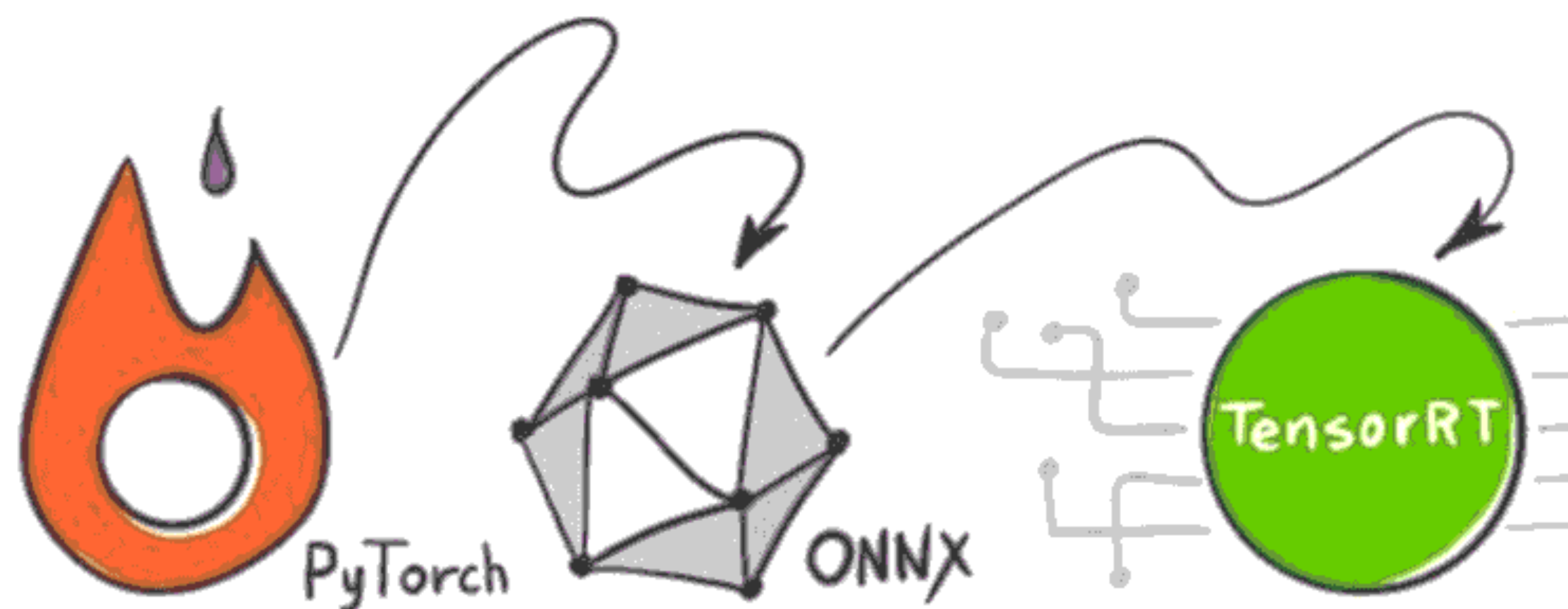
Особенности реализации конкретной архитектуры сети (умножение матриц и функции активации):

- Активация Mish в YOLO представляет композицию функций и ее реализация работает медленно.
- Реализация алгоритма фильтрации bbox (NMS) работает медленно на CPU.

Использование TensorRT C++ API для оптимизации

Плюсы:

- Обеспечивает более высокую производительность по сравнению с Python.
- Интеграция TensorRT C++ API с существующими проектами и библиотеками.
- Позволяет глубоко контролировать процесс инференса.
- Увеличение скорости препроцессинга, постпроцессинга
- Снижение аллокаций памяти.
- Простой процесс формирования движка для сети из формата ONNX.



Минусы:

- Сложность в освоении TensorRT C++ API.
- Сложность в разработке.
- Сложность в отладке кода.

Повышение производительности: Роль C++ в ускорении инференса

- C++ позволяет нам контролировать, когда и как выделяется и освобождается память.
- Выделяем память для сети при инициализации движка, избегая повторных выделений на каждый батч.
- Выделяем память под входное изображение.
- Препроцессинга на CUDA

Результат: **23 FPS** на Jetson NX

```
// Инициализируем компоненты TensorRT
auto builder = nvinfer1::createInferBuilder();
auto network = builder->createNetworkV2(0);
auto config = builder->createBuilderConfig();

// Загружаем onnx модель
auto parser = nvonnxparser::createParser(*network);
// Формируем engine
constructNetwork(builder, network, config, parser);

// Выделяем память под входы и выходы сети на GPU
size_t inputSize = calculateVolume(
    network->getInput(0)->getDimensions()
) * sizeof(float) * 3;
size_t outputSize = calculateVolume(
    network->getOutput(0)->getDimensions()
) * sizeof(float);
cudaMalloc(&mBuffers[0], inputSize);
cudaMalloc(&mBuffers[1], outputSize);

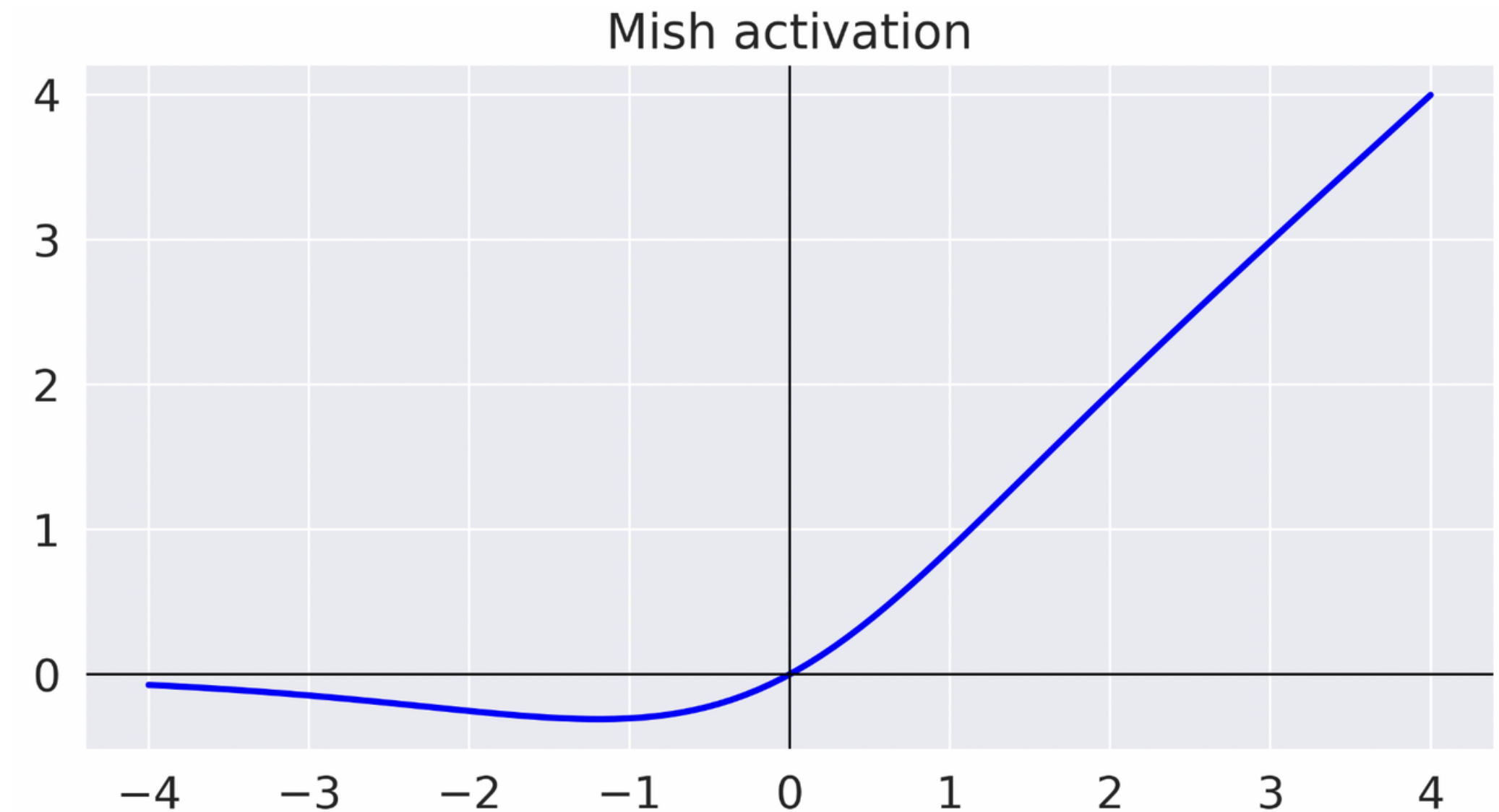
// Выделяем память под входное изображение на GPU
cudaMalloc(&mHostImg, mHeight * mWidth * sizeof(uint8_t));

// Получаем контекст выполнения для движка
mContext = SampleUniquePtr<nvinfer1::IExecutionContext>(
    mEngine->createExecutionContext()
);
```

Ускорение YOLO: Кастомные операции с TensorRT Plugin API

- TensorRT может не поддерживать все операции внутри нейронной сети.
- Для реализации не поддерживаемых операций, можно использовать плагины TensorRT.
- Плагины TensorRT - это пользовательские компоненты, которые расширяют функциональность TensorRT.

Основной проблемой функции активации **Mish**, используемой в YOLO является скорость ее работы. Такой вывод можно сделать из профайлера **TensorRT Perfomance**.



$$f(x) = x \cdot \tanh(\text{softplus}(x))$$

$$\text{softplus}(x) = \log(1 + e^x)$$

Ускорение YOLO: Кастомные операции с TensorRT Plugin API

- Функция **computeMish** на вход принимает массив активаций
- В реализации используется порог в 20 на вычисление функции активации. Он делает ответы более предсказуемыми.

```
// mish.cu

// CUDA функция softplus
__device__ float softplus_kernel(float x, const float threshold = 20.0f) {
    if (x > threshold) {
        return x;
    } else if (x < -threshold) {
        return expf(x);
    }
    return logf(expf(x) + 1.0f);
}

// CUDA функция tanh
__device__ float tanh_kernel(float x) {
    return (2.0f / (1.0f + expf(-2.0f * x)) - 1.0f);
}

// Ядро с Mish
template <typename T>
__global__ void mishKernel(int n, const T* input, T* output, const T MISH_THRESHOLD) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        T x_val = input[idx];
        output[idx] = x_val * tanh_kernel(softplus_kernel(x_val, MISH_THRESHOLD));
    }
}

// Посчитать Mish для входного массива
int computeMish(cudaStream_t stream, int n, const float* input, float* output) {
    constexpr int blockSize = 1024;
    const int gridSize = (n + blockSize - 1) / blockSize;
    mishKernel<float><<<gridSize, blockSize, 0, stream>>>(n, input, output, 20.0f);
    return 0;
}
```

Ускорение YOLO: Кастомные операции с TensorRT Plugin API

- Добавить заглушку с функцией активации при построении сети в torch.
- Экспортировать сеть в onnx.
- Необходимо добавить информацию о вызове плагина для операции **Mish**.
- На этапе построения исполняемого движка все операции в графе выполнения ONNX с названием «Mish» будут выполняться написанной на CUDA реализацией.
- Аналогично использовать уже реализованный плагин **NonMaxSuppression** для **NMS**, написанный на CUDA.

Использование плагинов полезно как для ускорения операций, так и для реализации новых операций, появляющихся с развитием архитектур нейронных сетей.

```
class MishCustomOp(torch.autograd.Function):  
    @staticmethod  
    def symbolic(g, input):  
        return g.op("Mish", input)
```

// Переопределение builtin_op_importers.cpp в TensorRT бэкэнде для ONNX

```
DEFINE_BUILTIN_OP_IMPORTER(Mish)  
{  
    // ...  
    nvinfer1::IPluginV2* plugin = pluginCreator->createPlugin(  
        node.name().c_str(), &fc  
    );  
    nvinfer1::IPluginV2Layer* layer = ctx->network()->addPluginV2(  
        pluginInputs.data(), pluginInputs.size(), *plugin  
    );  
    RETURN_ALL_OUTPUTS(layer);  
}
```

Результат: **30 FPS** на Jetson NX

Результаты оптимизации: до и после

Оптимизация	FPS Jetson NX	mAP@test
Ванильная реализация на Python	8 FPS	80 %
Python and FP16 quantization	16 FPS	72 %
Python and FP16 Quantization Aware Training	16 FPS	76 %
C++ and memory allocation and CUDA preprocessing	23 FPS	76 %
C++ and memory allocation and CUDA preprocessing and custom Mish OP and CUDA NMS postprocessing	30 FPS	76 %

Выводы

- **Производительность улучшена:** FPS детектора вырос с 8 до 30 на Jetson NX.
- **Методы универсальны:** Представленные методы могут быть обобщены и применены в любой области, где используются нейронные сети, чтобы обеспечить быстрый инференс.
- **Роль инженерии:** Представленные подходы демонстрируют, что глубокое понимание работы нейросетей позволяет адаптировать их для разнообразных задач и отраслей - от аналитики автомобильного трафика до систем рекомендаций.
- **Инвестиции в оптимизацию:** Делает ИИ-решения доступными и снижает затраты.

Спасибо за внимание!

Мои контакты

Почта: a.shalimov.work@gmail.com

Телеграм:



@SHALIMOV_AS

Ссылки

Проект **TensorRTx** с реализацией
моделей глубокого обучения с
использованием плагинов:

