СОДЕРЖАНИЕ

ГЛАВА 1. ОБРАБОТКА ИЗОБРАЖЕНИЙ

1.1 Изображения

Изображения определяют мир, каждое изображение имеет свою собственную историю, оно содержит много важной информации, которая может быть полезна во многих отношениях. Эта информация может быть получена с помощью техники, известной как *обработка изображений*.

Это основная часть компьютерного зрения, которая играет решающую роль во многих примерах реального мира, таких как робототехника, самоуправляемые автомобили и обнаружение объектов. Обработка изображений позволяет нам преобразовывать и манипулировать тысячами изображений одновременно и извлекать из них полезные сведения. Она имеет широкий спектр применения практически во всех областях.

Как видно из названия, обработка изображений означает обработку изображения, которая может включать в себя множество различных методов, пока мы не достигнем нашей цели.

Конечный результат может быть либо в виде изображения, либо в виде соответствующей характеристики этого изображения. Это может быть использовано для дальнейшего анализа и принятия решений.

Изображение можно представить в виде двумерной функции $F(x, y)$, где x и y – пространственные координаты. Амплитуда F при определенном значении x, y называется интенсивностью изображения в этой точке. Если x, y и значение амплитуды конечны, то мы называем это цифровым изображением. Оно представляет собой массив пикселей, расположенных в столбцах и строках. Пиксели – это элементы изображения, которые содержат информацию об интенсивности и цвете. Изображение также может быть представлено в $3D$, где x, y и z становятся пространственными координатами. Пиксели располагаются в виде матрицы. Такое изображение известно как RGB-*изображение*.

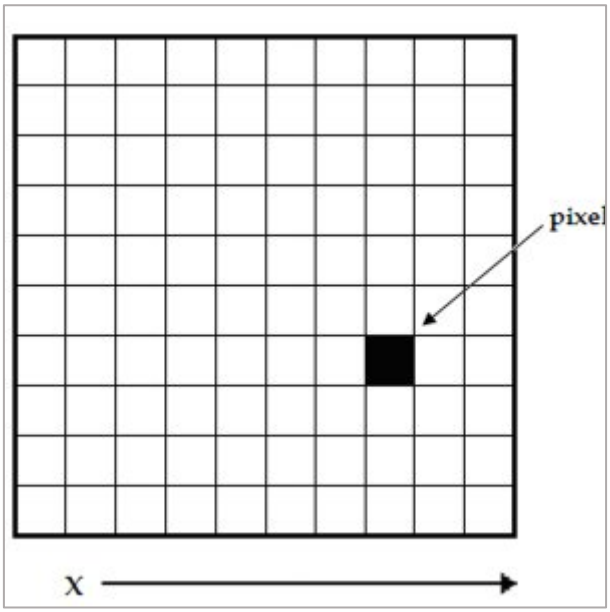

Рисунок 1.1.1 – Двумерный массив пикселей

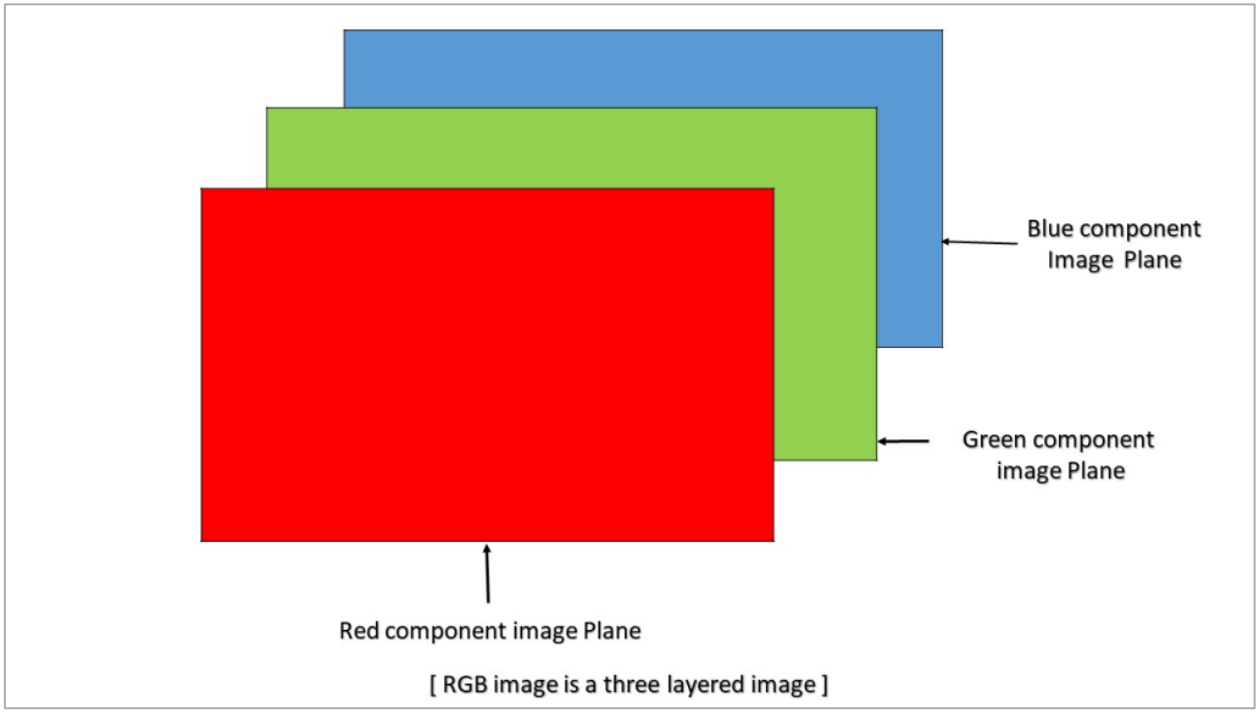

Рисунок 1.1.2 – RGB-изображение

Существуют различные типы изображений:

- RGB-изображение: содержит три слоя двухмерного изображения – красный, зеленый и синий каналы.
- Изображение в оттенках серого: эти изображения содержат оттенки черного и белого цветов и содержат только один канал.

1.2 Классические алгоритмы обработки изображений

1.2.1 Морфологическая обработка изображений

Морфологическая обработка изображений пытается удалить недостатки из бинарных изображений, поскольку бинарные области, полученные простым пороговым выделением, могут быть искажены шумом. Она также помогает сгладить изображение с помощью операций открытия и закрытия.

Морфологические операции могут быть распространены на полутоновые изображения. Они состоят из нелинейных операций, связанных со структурой особенностей изображения. Они зависят не от связанного упорядочивания пикселей, а от их числовых значений. Эта техника анализирует изображение с помощью небольшого шаблона, известного как *структурирующий элемент*, который размещается в различных возможных местах изображения и сравнивается с соответствующими соседними пикселями. Структурирующий элемент – это небольшая матрица со значениями 0 и 1.

Рассмотрим две фундаментальные операции морфологической обработки изображений – дилатацию и эрозию:

- Операция *дилатации* добавляет пиксели к границам объекта на изображении
- Операция *эрозии* удаляет пиксели от границ объекта.

Количество пикселей, удаленных или добавленных к исходному изображению, зависит от размера структурирующего элемента.

Структурирующий элемент – это матрица, состоящая только из 0 и 1, которая может иметь произвольную форму и размер. Она размещается во всех возможных местах изображения и сравнивается с соответствующей окрестностью пикселей.

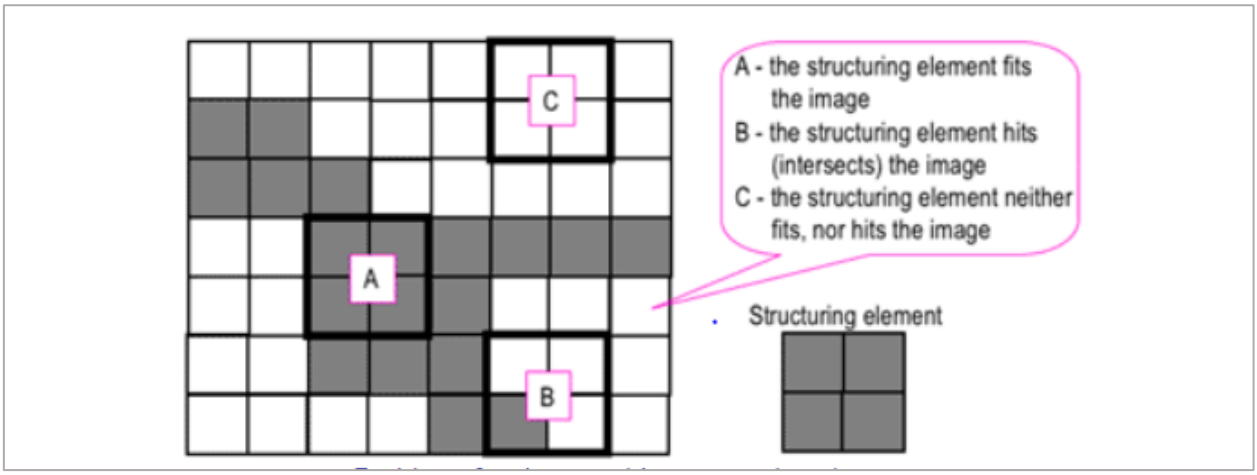

Рисунок 1.2.1 – Квадратное структурирование

Квадратный структурирующий элемент 'A' помещается в объект, который мы хотим выделить, 'B' пересекает объект, а 'C' находится вне объекта.

Шаблон "ноль-один" определяет конфигурацию структурирующего элемента. Она соответствует форме объекта, который мы хотим выбрать. Центр структурирующего элемента определяет обрабатываемый пиксель.

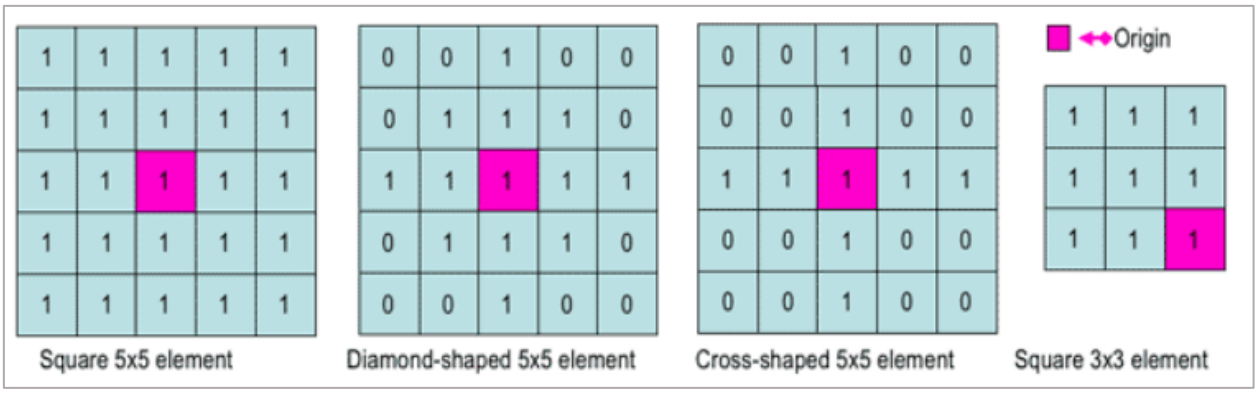

Рисунок 1.2.2 – Квадратное структурирование

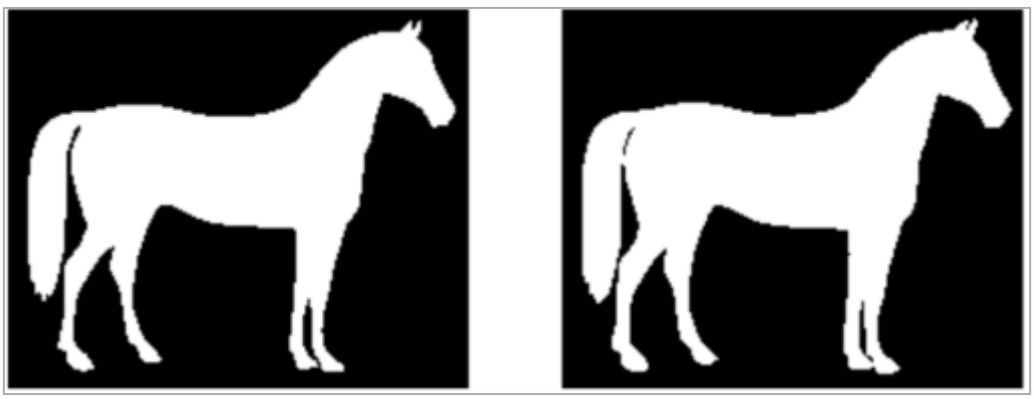

Рисунок 1.2.3 – Дилатация

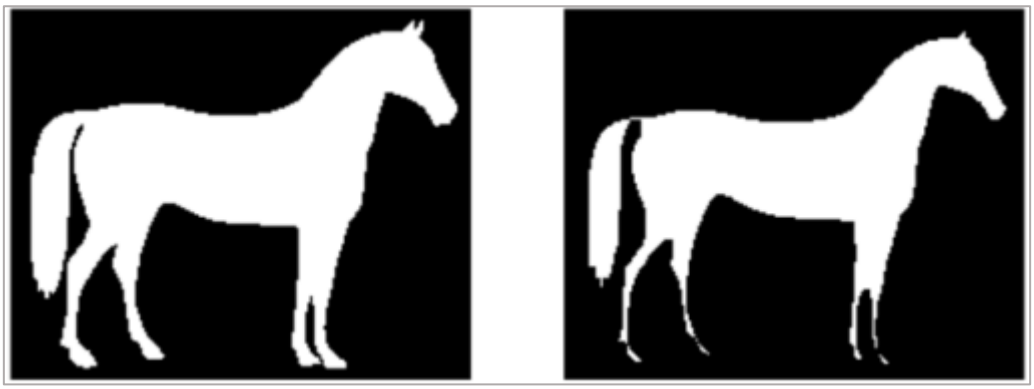

Рисунок 1.2.4 – Эрозия

1.2.2 Гауссова обработка изображений

Размытие по Гауссу, которое также известно как гауссово сглаживание, является результатом размытия *изображения* с помощью *гауссовой* функции.

Оно используется для *уменьшения шума* изображения и *уменьшения деталей*. Визуальный эффект этой техники размытия похож на взгляд на изображение через полупрозрачный экран. Иногда она используется в компьютерном зрении для улучшения изображений в различных масштабах или как техника преобразования данных в глубоком обучении.

Базовая гауссова функция имеет вид:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Рисунок 1.2.5 – Гауссова функция

На практике лучше всего воспользоваться таким свойством гауссова размытия как сепарабельность, разделив процесс на два прохода. В первом проходе используется одномерное ядро для размытия изображения только в горизонтальном или вертикальном направлении. Во время второго прохода то же самое одномерное ядро используется для размытия в остальных направлениях. В результате получается тот же эффект, что и при свертке с

двумерным ядром за один проход. Чтобы понять, что именно гауссовы фильтры делают с изображением, был рассмотрен следующий пример.

Если есть нормально распределенный фильтр, то при его применении к изображению результаты выглядят следующим образом:

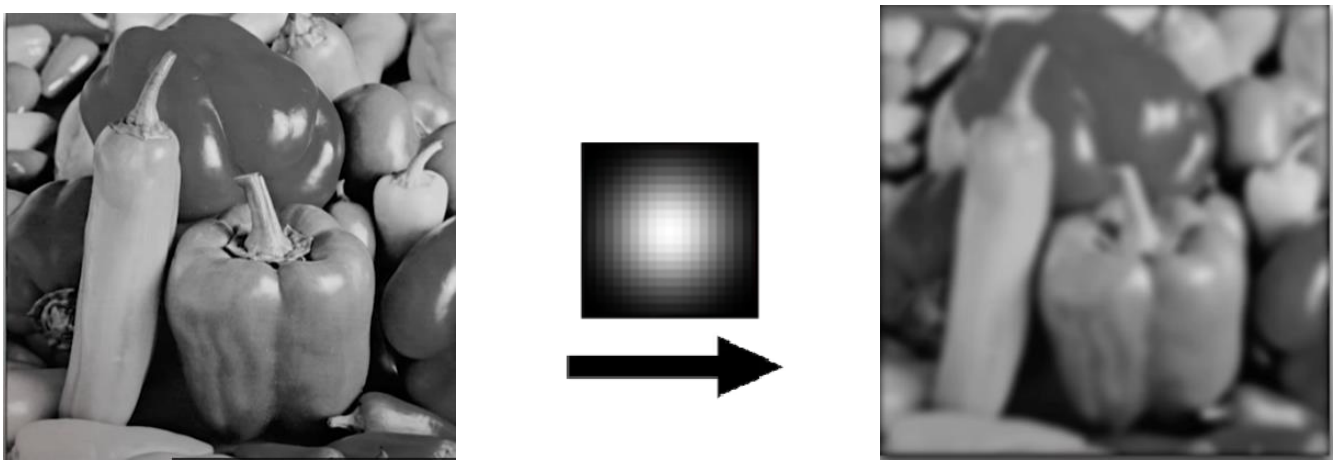

Рисунок 1.2.6 – Слева направо: оригинал, гауссовый фильтр, результат

Видно, что некоторые края имеют немного меньше деталей. Фильтр придает больший вес пикселям в центре, чем пикселям вдали от центра. Гауссовы фильтры являются фильтрами низких частот, то есть ослабляют высокие частоты. Он широко используется при обнаружении границ.

1.2.3 Преобразование Фурье в обработке изображений

Преобразование Фурье разбивает изображение на синусоидальную и косинусоидальную составляющие.

Оно имеет множество применений, таких как реконструкция изображений, сжатие изображений или фильтрация изображений.

Поскольку рассматриваются изображения, будет рассматриваться дискретное преобразование Фурье.

Рассмотрим синусоиду, она состоит из трех составляющих:

- Величина – связана с контрастом

- Пространственная частота – связана с яркостью
- Фаза – связана с информацией о цвете

Изображение в частотной области выглядит следующим образом:

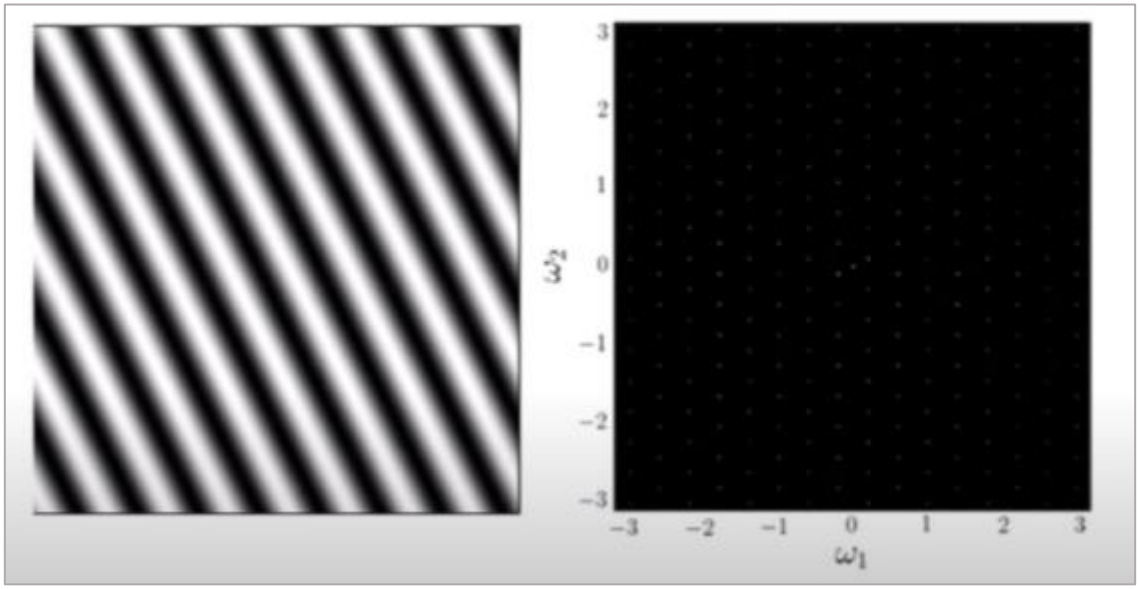

Рисунок 1.2.7 – Изображение в частотной области

Формула для двумерного дискретного преобразования Фурье имеет вид:

$$F(u, v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

В приведенной выше формуле $f(x, y)$ обозначает изображение.

Обратное преобразование Фурье преобразует преобразование обратно в изображение. Формула двумерного обратного дискретного преобразования Фурье имеет вид:

$$f(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v)\, e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

1.2.4 Обнаружение краев в обработке изображений

Определение краев – это метод обработки изображений для нахождения границ объектов на изображениях. Он работает путем обнаружения перепадов яркости.

Это может быть очень полезно для извлечения полезной информации из изображения, поскольку большая часть информации о форме заключена в краях. Классические методы определения краев работают путем обнаружения разрывов в яркости.

Они могут быстро отреагировать, если на изображении обнаружен шум, при обнаружении изменений уровней серого. Края определяются как локальные максимумы градиента.

Наиболее распространенным алгоритмом обнаружения краев является *алгоритм обнаружения краев Собеля*. Оператор обнаружения Собеля использует ядра свертки 3 x 3. Простое ядро Gx и повернутое на 90 градусов ядро Gy. Отдельные измерения производятся путем применения обоих ядер по отдельности к изображению.

$$Gx = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * Image\ matrix$$

Рисунок 1.2.8 – Изображение Gx

$$Gy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * Image\ matrix$$

Рисунок 1.2.9 – Изображение Gy

, где $*$ обозначает двумерную операцию свертки

Результирующий градиент может быть рассчитан как:

$$G = sqrt(Gx^2 + Gy^2)$$

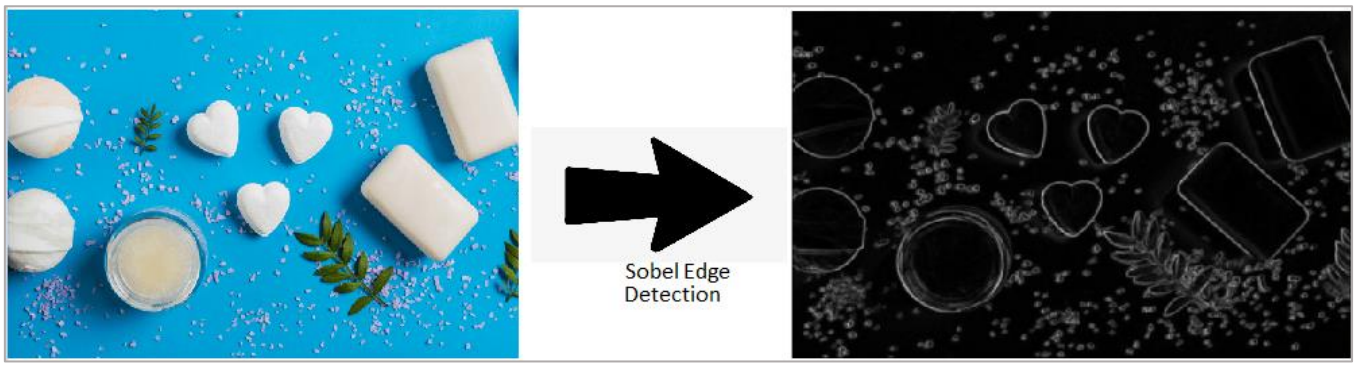

Рисунок 1.2.9 – Определение краев

1.2.5 Вейвлет обработка изображений

Рассмотренное ранее преобразование Фурье ограничено только частотой. Вейвлеты учитывают как время, так и частоту. Это преобразование подходит для нестационарных сигналов.

Известно, что края являются одной из важных частей изображения, при применении традиционных фильтров было замечено, что шум удаляется, но изображение становится размытым. Вейвлет-преобразование разработано таким образом, что мы получаем хорошее частотное разрешение для низкочастотных компонентов. Ниже приведен пример двухмерного вейвлет-преобразования:

Рисунок 1.2.10 – Вейвлет-обработка изображений

1.3Обработка изображений с помощью нейронных сетей

1.3.1 Нейронные сети

Нейронные сети – это многослойные сети, состоящие из нейронов или узлов. Эти нейроны являются основными вычислительными единицами нейронной сети. Они разработаны таким образом, чтобы действовать подобно человеческому мозгу. Они принимают данные, обучаются распознавать закономерности в данных и затем предсказывают результат.

Стандартная нейронная сеть состоит из трех слоев:

- Входной слой
- Скрытый слой
- Выходной слой

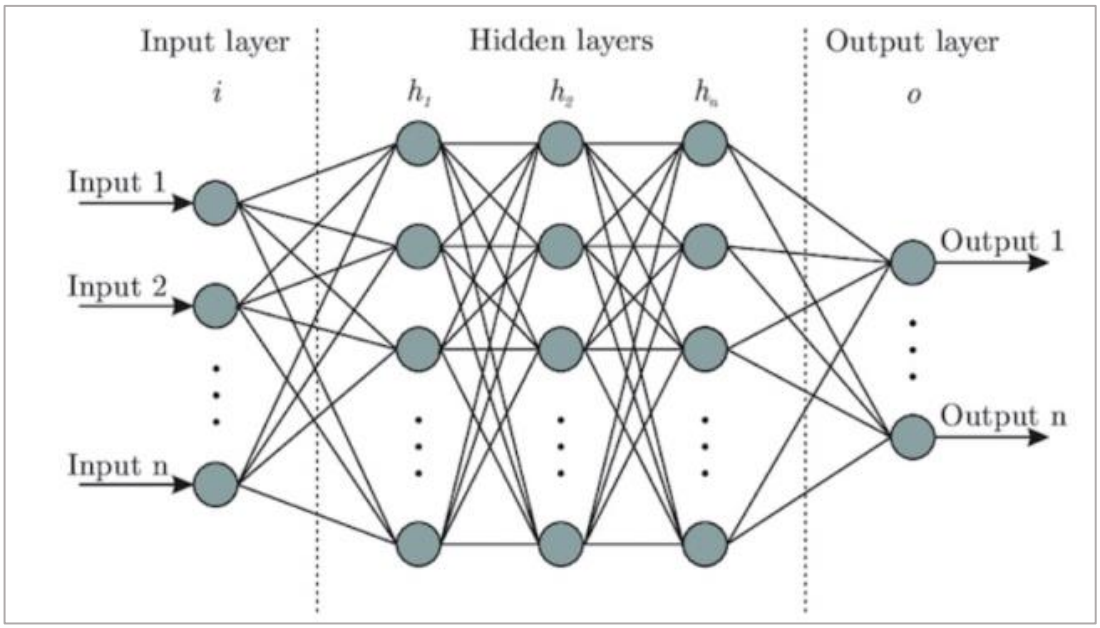

Рисунок 1.3.1 – Стандартная нейронная сеть

Входные слои получают входные данные, выходной слой прогнозирует выходной сигнал, а скрытые слои выполняют большую часть вычислений. Количество скрытых слоев может быть изменено в соответствии с требованиями. В нейронной сети должен быть как минимум один скрытый слой.

Принцип работы нейронной сети:

1. Рассмотрим изображение – каждый пиксель подается на вход каждого нейрона первого слоя, нейроны одного слоя соединены с нейронами следующего слоя через каналы.
2. Каждому из этих каналов присваивается числовое значение, известное как вес.
3. Входы умножаются на соответствующие веса, и эта взвешенная сумма подается на вход скрытых слоев.
4. Выходные данные скрытых слоев проходят через функцию активации, которая определяет, будет ли конкретный нейрон активирован или нет.
5. Активированные нейроны передают данные на следующие скрытые слои. Таким образом, данные распространяются по сети, что известно как прямое распространение.
6. В выходном слое нейрон с наибольшим значением предсказывает выход. Эти выходы являются значениями вероятности.
7. Предсказанный выход сравнивается с фактическим выходом для получения ошибки. Эта информация затем передается обратно через сеть, этот процесс известен как обратное распространение ошибки.
8. На основе этой информации корректируются веса. Этот цикл прямого и обратного распространения выполняется несколько раз на нескольких входах, пока сеть не предскажет выход правильно в большинстве случаев.
9. На этом процесс обучения нейронной сети заканчивается. В некоторых случаях время обучения нейронной сети может оказаться существенным.

На рисунке ниже a_i – набор входных значений, w_i – веса, z – выходное значение, а g – функция активации.

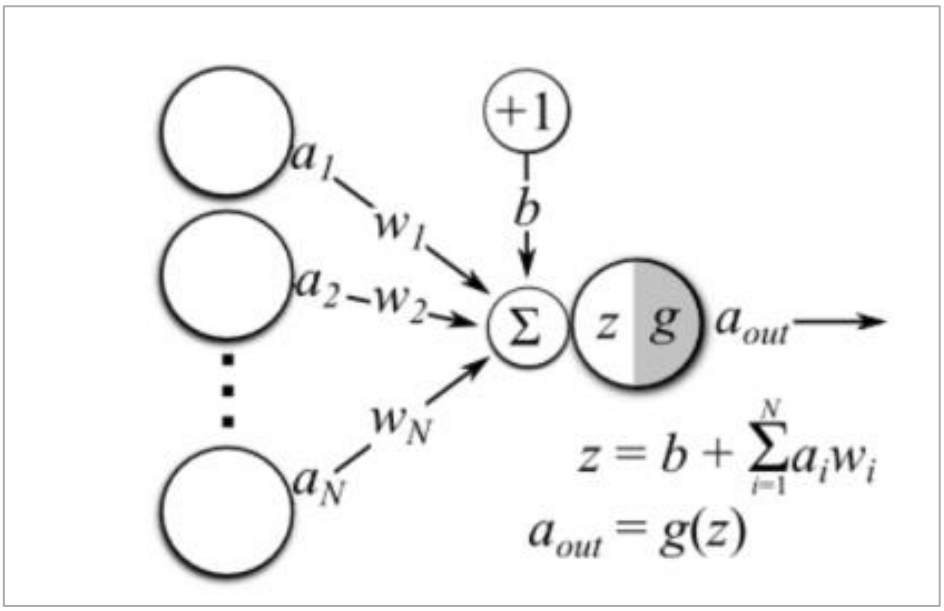

Рисунок 1.3.2 – Операции в одном нейроне

Можно выделить несколько рекомендаций по подготовке данных для обработки изображений.

- Для получения лучших результатов в модель необходимо подавать больше данных.
- Набор данных изображений должен быть высокого качества, чтобы получить более четкую информацию, но для их обработки могут потребоваться более глубокие нейронные сети.
- Во многих случаях RGB-изображения преобразуются в градации серого перед подачей их в нейронную сеть.

1.3.2 Типы нейронных сетей в обработке изображений

Свёрточная нейронная сеть

Свёрточная нейронная сеть имеет три типа слоёв:

- Слой свёртки ($Conv$) – это основной строительный блок CNN, он отвечает за выполнение операции свертки. Элемент, участвующий в выполнении операции свертки в этом слое, называется *ядро/фильтр (матрица)*. Ядро выполняет горизонтальные и вертикальные сдвиги, пока не будет пройдено все изображение.

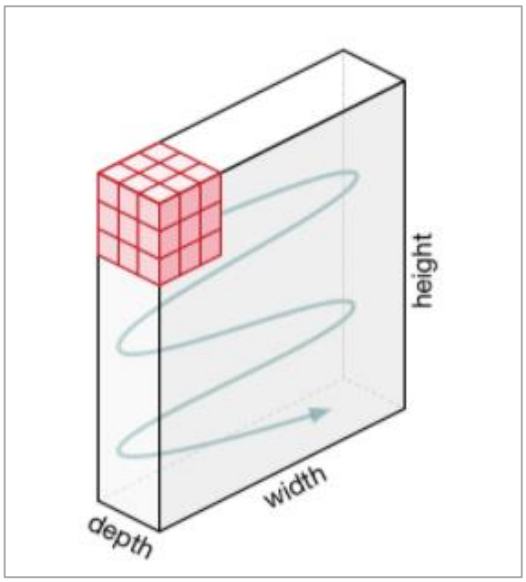

Рисунок 1.3.3 – Движение ядра

o Слой субдискретизации или пулинг ($Pool$) – этот слой отвечает за снижение размерности. Он помогает уменьшить вычислительную мощность, необходимую для обработки данных. Существует два типа пулинга: $MaxPooling$ и $AveragePooling$. Первый возвращает максимальное значение из области, покрытой ядром на изображении. Второй же возвращает среднее значение всех значений в части изображения, покрытой ядром.

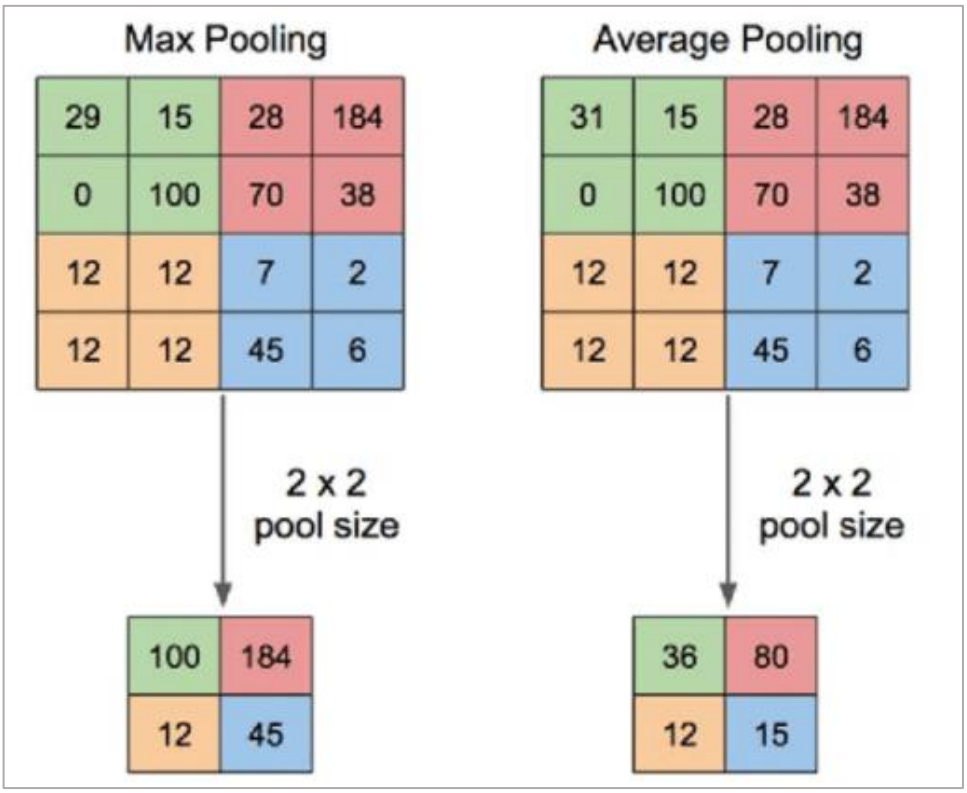

Рисунок 1.3.4 – Операция пулинга

- Полносвязная нейронная сеть (FC) – полносвязная нейронная сеть работает через входной слой $Flatten$, где каждый вход подключен ко всем нейронам. При наличии в сети, данные слои обычно находятся в конце архитектуры CNN.

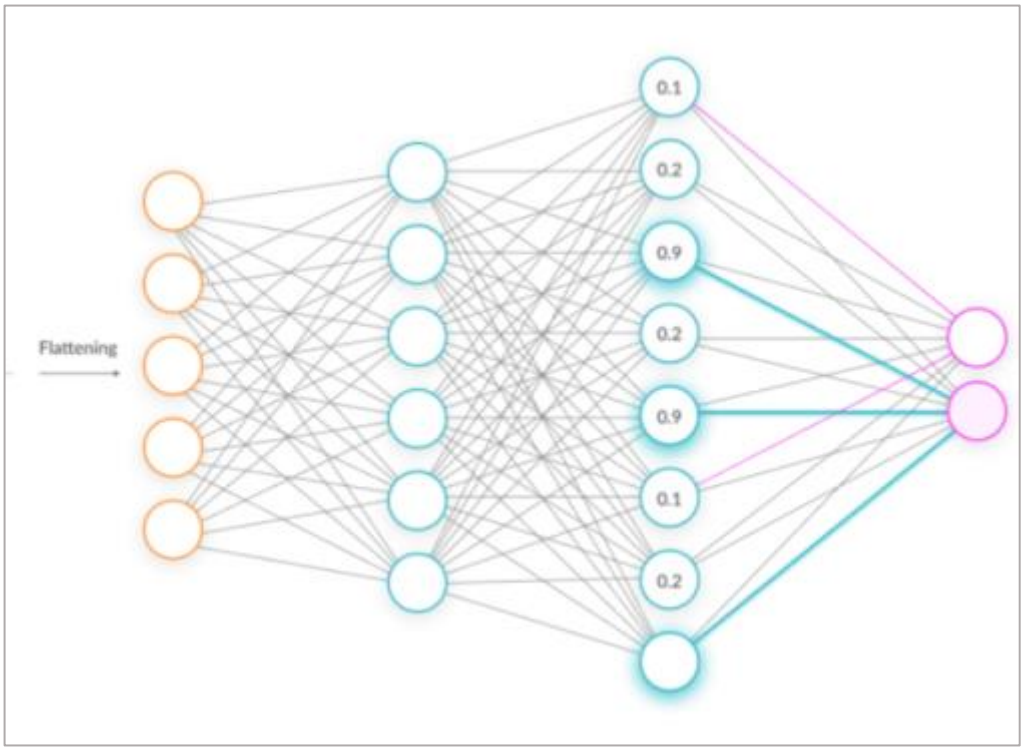

Рисунок 1.3.5 – Полносвязная нейронная сеть

CNN в основном используется для извлечения особенностей из изображения с помощью своих слоев. CNN широко используются для классификации изображений, где каждое входное изображение пропускается через ряд слоев для получения вероятностного значения между 0 и 1.

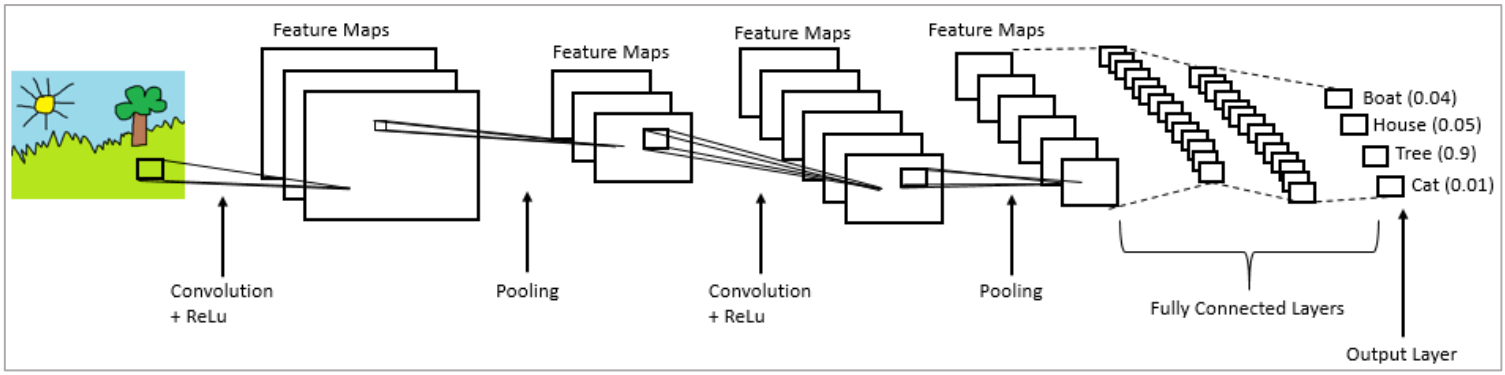

Рисунок 1.3.5 – CNN-обработка изображений

Генеративно-состязательная сеть (GAN)

Генеративные модели используют подход обучения без учителя (есть изображения, но нет меток).

GAN состоят из двух моделей $Generator$ (Генеративная модель) и $Discriminator$ (Дискриминативная модель). Генератор учится создавать поддельные изображения, которые выглядят реалистично, чтобы обмануть дискриминатор, а дискриминатор учится отличать поддельные изображения от настоящих (старается не быть обманутым).

Генератору не разрешается видеть реальные изображения, поэтому на начальном этапе он может давать плохие результаты, в то время как дискриминатору разрешается смотреть на реальные изображения, но они перемешаны с поддельными, созданными генератором, которые он должен классифицировать как настоящие или поддельные.

На вход генератора подается некоторый шум, чтобы он мог каждый раз выдавать разные примеры, а не однотипные изображения. На основе оценок, предсказанных дискриминатором, генератор пытается улучшить свои результаты, через определенное время генератор сможет производить изображения, которые будет сложнее отличить. Дискриминатор также совершенствуется, так как получает от генератора все более реалистичные изображения на каждом раунде.

Популярными типами GAN являются:

- Глубокие свёрточные GAN ($DCGAN$)
- Условные GAN ($cGAN$)
- $StyleGAN$
- $CycleGAN$
- $DiscoGAN$
- $GauGAN$

GAN отлично подходят для генерации и обработки изображений. Некоторые применения GAN включают в себя: состаривание лица, смешивание фотографий, $Super\ Resolution$, восстановление изображения.

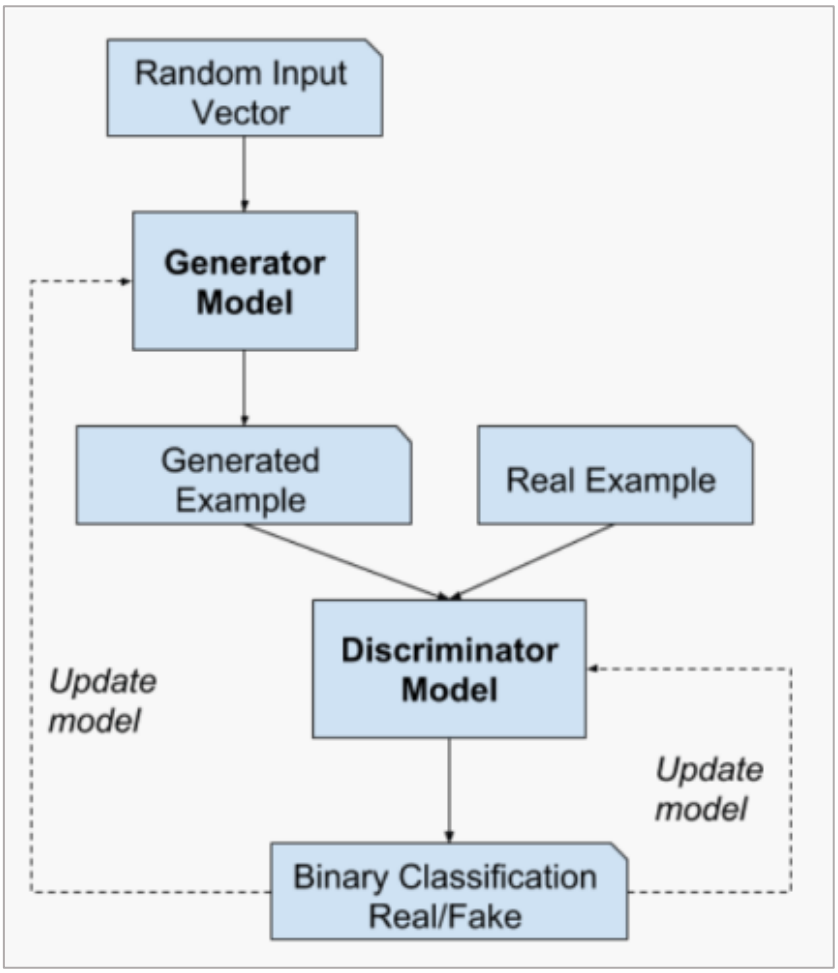

Рисунок 1.3.5 – Принцип работы GAN

ГЛАВА 2. INTEL OPENVINO TOOLKIT

Общее определение *OpenVINO toolkit* можно определить, как набор полезных инструментов, которые объединяют обработку видео, компьютерное зрение, обработку естественного языка, машинное обучение и оптимизацию нейронных сетей в единый пакет.

Выделив самую главную мысль и говоря простым языком можно сказать, что *OpenVINO* – это набор инструментов, который призван максимально эффективно запустить нейронную сеть на вашем железе, то есть выполнить какие-то оптимизации, чтобы все работало максимально быстро на предоставляемых девайсах.

2.1 Процесс разработки

2.1.1 Процесс разработки нейронной сети

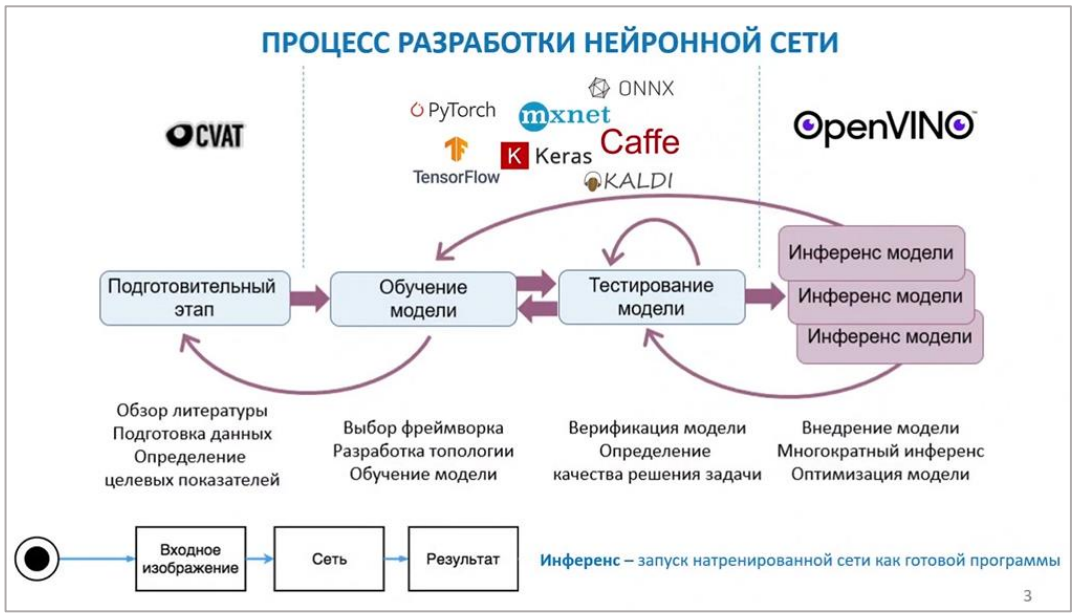

Рисунок 2.1.1.1 – Процесс разработки нейронной сети

Процесс разработки нейронной сети можно разделить на три(четыре) этапа:

1. Подготовительный
   - Чтение литературы на предмет поиска близких для решения задачи архитектур;

- Определение целевых показателей, которые необходимо достичь по точности
- Подготовка данных является самым важным этапом, так как не всегда существует датасет, который подходит под конкретную задачу, иногда его приходится создавать самостоятельно – для создания датасета, относящегося к области компьютерного зрения у $Intel$ существует инструмент $CVAT$ для разметки данных

2. Обучение и тестирование модели
   - Выбор фреймворка
   - Разработка топологии сети / выбор готовой
   - Обучение модели
   - Определить качества решения задачи (этим занимаются фреймворки)
3. Исполнение – инференс модели

Инференс – запуск готовой натренированной сети как готовой программы.

Важно заметить, что *OpenVINO toolkit* не занимается ни обучением, ни тестированием, а направлен только на инференс.

2.1.2 Фреймворки и OpenVINO

Фреймворки можно использовать для инференса, но делать этого не стоит по нескольким причинам:

1. Фреймворки обычно тяжеловесны – содержат большое количество библиотек, большое количество кода, которое направлено как раз на обучение и тестирование моделей
2. Не имеют никаких оптимизаций
3. Фреймворки часто не задействуют весь потенциал аппаратного обеспечения – например во фреймворке *TensorFlow*, чтобы задействовать *GPU* надо устанавливать дополнительные драйвера, то есть во фреймворк они не встроены

В свою очередь *OpenVINO toolkit* имеет следующие плюсы:

1. Использует облегченную среду исполнения, которая отвечает только за инференс модели, то есть исполнение нейронной сети и не привносит за собой какие-либо модули, отвечающие за обучение и тестирование модели
2. *Software* оптимизации – например оптимизация топологии сети
3. Оптимизации под железо Intel
4. Один API для всех целей
5. Model Zoo – можно найти модели обученные Intel и сторонними людьми, но оптимизированные для *OpenVINO toolkit*

2.1.3 Архитектура OpenVINO

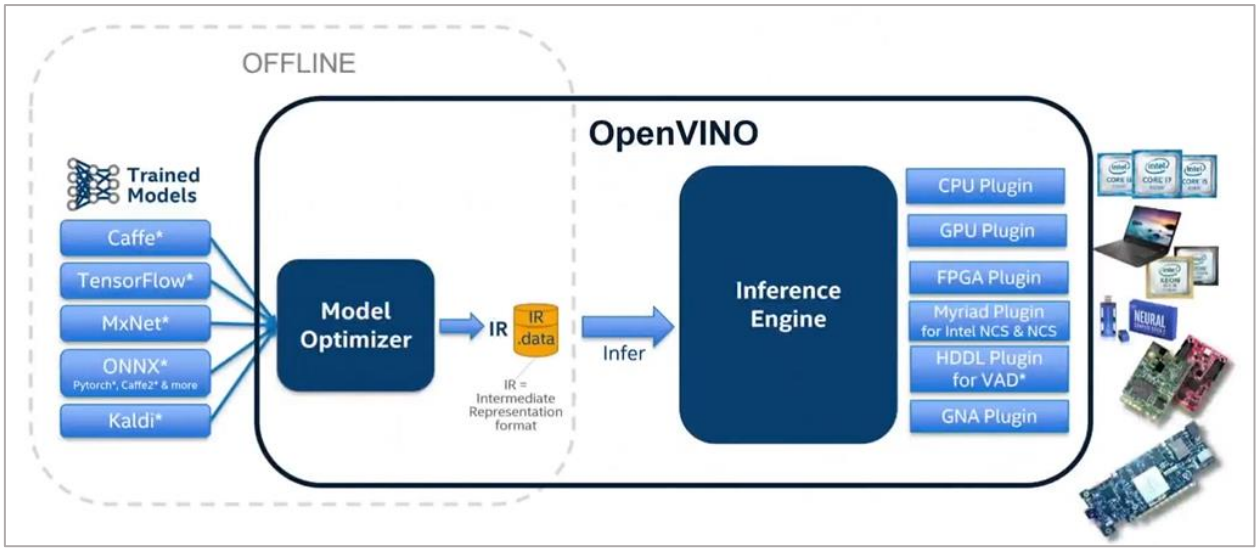

Рисунок 2.1.1.3 – Архитектура OpenVINO

Архитектуру OpenVINO глобально можно разделить на две части:

- Model Optimizer – набор скриптов, написанных на Python, которые позволяют перевести сеть, обученную в каком-то фреймворке в IR-формат.

  IR (Intermediate representation) – специальный формат, разработанный Intel (промежуточное представление), с которым дальше работает Inference Engine и все инструменты, которые есть в OpenVINO toolkit.
- Inference engine (дословно механизм инференса) – отвечает за как раз таки исполнение сетей на каком-либо железе (подгружаются плагины, которые отвечают за каждый девайс и затем сеть исполняется)

2.2 Model Optimizer

2.2.1 Инструментарий

Model Optimizer – это кроссплатформенный инструментарий командной строки, который упрощает переход между средой обучения и средой развертывания, осуществляет статический анализ модели и настраивает модели глубокого обучения для оптимального исполнения на целевых устройствах.

Процесс исполнения Model Optimizer предполагает, что у вас есть сетевая модель, обученная с применением поддерживаемых фреймворков глубокого обучения: Caffe, TensorFlow, Kaldi, MXNet или преобразованная в формат ONNX. Model Optimizer создает промежуточное представление (IR) сети, которое может быть выведено с помощью Inference Engine. То есть MO получает на вход модель, а на выходе выдает файл в формате IR, что представлено на рисунке 2.2.1.1.

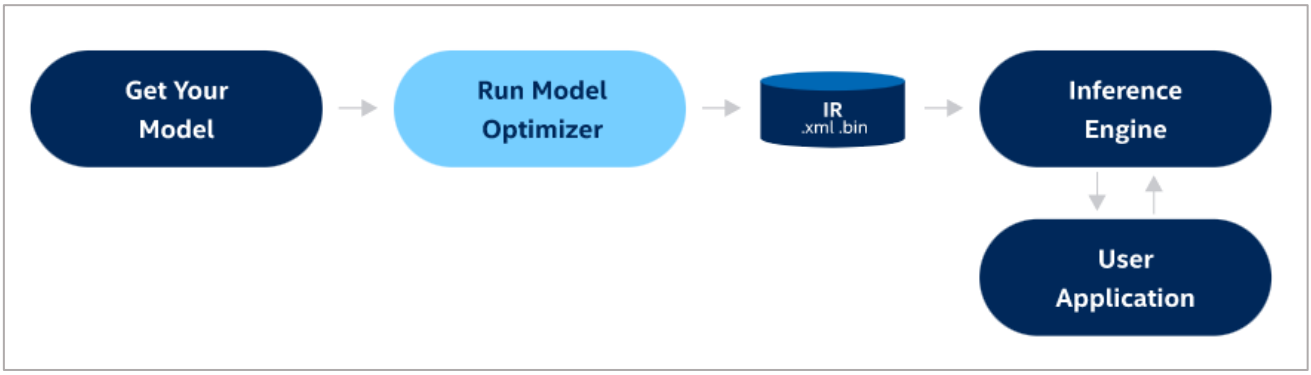

Рисунок 2.2.1.1 – Развертывание модели глубокого обучения

IR состоит из двух файлов:

- Xml, в котором находится описание топологии сети (какие слои, какого типа, как они соединены)
- Bin файл, где в бинарном формате описаны веса модели

Однако есть еще один способ загрузки модели в Inference Engine – минуя Model Optimizer – это сделать с помощью API, то есть каким-то программным способом. Если говорить в общем, то именно поэтому сейчас в Model Optimizer становится все меньше оптимизаций, потому что если модель будет

подгружаться с помощью API, то эти оптимизации потеряются, так как Model Optimizer не задействуется, поэтому все оптимизации сейчас переходят в Inference Engine и другие сторонние инструменты и сейчас Model Optimizer, не смотря на свое название становится больше конвертером, чем оптимизатором.

Model Optimizer выполняет предварительную обработку модели. Можно оптимизировать этот шаг и улучшить время первого вывода, для этого воспользоваться следующими рекомендациями:

1. Параметры среднего и масштаба изображения

Необходимо использовать параметры среднего и масштаба входного изображения (*scale* и *mean_values*) в Model Optimizer, если требуется предварительная обработка. Это позволит инструменту вставить предварительную обработку в IR, чтобы ускорить ее с помощью Inference Engine.

2. Входы RGB и входы BGR

Если сеть предполагает RGB-входы, Model Optimizer может поменять каналы в первой свертке с помощью опции командной строки *reverse_input_channels*, так что не нужно конвертировать входы в RGB каждый раз, когда вы получаете изображение BGR, например, из OpenCV.

3. Больший размер батча

Такие устройства, как GPU, работают лучше при большем размере пакета. Хотя есть возможность установить размер батча и во время выполнения с помощью Inference Engine *ShapeInference feature*.

4. Результирующая точность IR

Результирующая точность IR, например, FP16 или FP32, напрямую влияет на производительность. Поскольку CPU теперь поддерживает FP16 (при этом внутреннее масштабирование все равно происходит до FP32) и поскольку это наилучшая точность для GPU, вы можете захотеть всегда преобразовывать модели в FP16.

Модель в памяти можно представить в виде направленного графа, где узлы – это операции, а ребра соответствуют передаче данных от операции производителя (узел) к операции потребителя (узел).

Model Optimizer использует экземпляр класса mo.graph.Graph.Graph для представления графа вычислений в памяти во время преобразования модели. Этот класс наследуется от класса networkx.MultiDiGraph стандартной библиотеки Python networkx и предоставляет множество удобных методов для обхода и модификации графа

Model Optimizer хранит всю необходимую информацию об операции в атрибутах узла. Он использует класс mo.graph.graph.Node, который является оберткой поверх словаря атрибутов узла networkx и предоставляет множество удобных методов для работы с узлом.

Операция может иметь несколько входов и выходов. Например, операция Split имеет два входа: данные для разбиения и ось для разбиения, и переменное количество выходов в зависимости от значения атрибута num_splits. Каждый входной сигнал операции передается на определенный входной порт операции. Операция производит выходные данные из выходного порта. Входные и выходные порты нумеруются от 0 независимо друг от друга. Model Optimizer использует классы mo.graph.port.Port и mo.graph.connection.Connection, которые являются полезной абстракцией для выполнения таких модификаций графа, как соединение/пересоединение узлов и обход графа.

Не существует специального класса, соответствующего ребру, поэтому для получения доступа к атрибутам ребра при необходимости требуется низкоуровневое манипулирование графом. Между тем, большинство манипуляций с соединениями узлов должно выполняться с помощью классов mo.graph.connection.Connection и mo.graph.port.Port. Таким образом, низкоуровневые манипуляции с графами чреваты ошибками и настоятельно не рекомендуются.

Конвейер преобразования модели можно представить в виде следующей диаграммы:

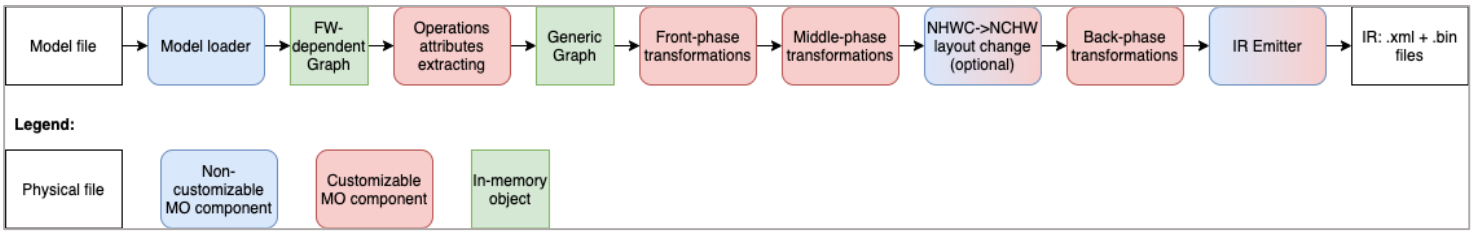

Рисунок 2.2.1.2 – Конвейер преобразования модели

2.2.2 Оптимизации

Model Optimizer не только преобразует модель в формат IR, но и выполняет ряд оптимизаций. Например, отдельные примитивы, такие как линейные операции (BatchNorm и ScaleShift), автоматически соединяются в свертки. Как правило, эти слои не должны выражаться в результирующем IR:

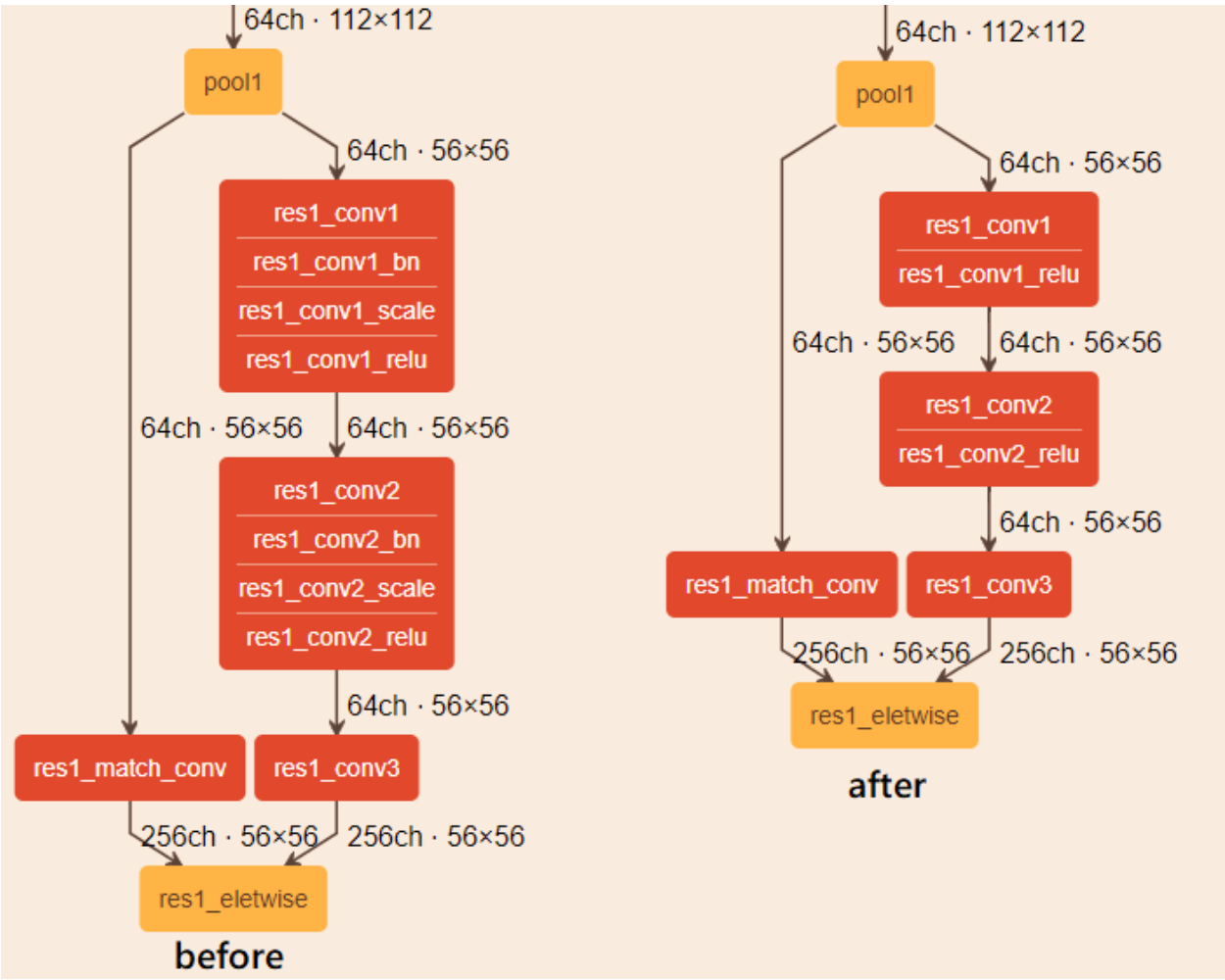

Рисунок 2.2.2.1 – Оптимизации

На рисунке выше изображена топология Resnet269. Слева – первоначальная модель, а справа – модель после преобразования, которую делает Model Optimizer, притом слои BatchNorm и ScaleShift слиты с весами свертки, а не представляют собой единичные слои.

Но Model Optimizer иногда не может выполнить слияние. Например, нелинейные операции (активации) между свертками и линейными операциями могут помешать данной оптимизации. Когда производительность является приоритетом, можно видоизменить (и, может быть, переобучить) топологию.

Важно обратить внимание на то, что активация (слой relu) не затрагивается оптимизатором модели, и, хотя она также может быть объединена в свертку, это скорее специфическая для устройства оптимизация, которая покрывается Inference Engine во время загрузки модели.

Пользовательские операции

Также присутствуют пользовательские операции. Пользовательские операции – это операции, которые не включены в перечень известных операций. Если модель включает какую-нибудь операцию, что не входит в перечень известных операций, Model Optimizer не сможет сформировать IR для данной модели.

Model Optimizer дает механизм расширений для поддержки новых операций и реализации пользовательских модификаций модели для генерации оптимизированного IR.

Для поддержки пользовательских операций нужно реализовать как минимум два типа расширений оптимизатора моделей:

1. Класс Operation для новой операции – данный класс сохраняет информацию об операции, ее атрибутах, функции вывода формы, атрибутах для сохранения в IR и некоторых иных внутренне используемых атрибутах.
2. Экстрактор атрибутов операции. Экстрактор нужен для разбора особого для фреймворка представления операции и использует соответствующий

класс операции для замены атрибутов узлов графа нужными атрибутами операции.

Загрузчик моделей

Данный каталог содержит скрипты, автоматизирующие некоторые задачи, связанные с моделями, на основе конфигурационных файлов в каталогах моделей:

- Model Downloader: downloader.py загружает файлы моделей из онлайн-источников и, при необходимости, исправляет их, чтобы сделать более пригодными для использования в Model Optimizer
- Конвертер моделей: converter.py конвертирует модели, которые не находятся в формате Inference Engine IR, в этот формат с помощью Model Optimizer.
- Model Quantizer: quantizer.py квантует модели полной точности в формате IR в версии с низкой точностью с помощью Post-Training Optimization Toolkit.
- Model Information Dumper: info_dumper.py печатает информацию о моделях в стабильном машиночитаемом формате.

2.3 Intermediate Representation OpenVINO

2.3.1 Описание формата

Инструментарий OpenVINO представляет свой собственный формат представления графов и свой набор операций. Граф представляется в виде двух файлов: XML-файла и бинарного файла. Данное представление принято называть Intermediate Representation или IR (промежуточное представление).

XML-файл описывает топологию сети, используя тег *layer* для узла операции и тег *edge* для соединения потока данных. Любая операция имеет фиксированное количество атрибутов, которые определяют логику операции,

используемой для текущего узла. Например, операция Convolution имеет следующие атрибуты: dilation, stride, pads_begin и pads_end.

В XML-файле нет больших константных значений, таких как веса свертки. Взамен этого он ссылается на часть сопроводительного бинарного файла, который хранит такие значения в формате bin.

2.3.2 Набор операций

Операции в наборе операций OpenVINO избираются на основе возможностей поддерживаемых фреймворков глубокого обучения и аппаратных возможностей устройства вывода. Категории операций представлены ниже:

- Стандартные слои глубокого обучения, такие как Convolution, MaxPool, MatMul (FullyConnected)
- Разные функции активации, например, ReLU, Tanh, PReLU.
- Общие арифметические тензорные операции, такие как сложение, вычитание, умножение.
- Операции сравнения, которые сравнивают два числовых тензора и создают булевы тензоры, например, Less, Equeal, Greater.
- Логические операции, которые работают с булевыми тензорами, например, And, Xor, Not.
- Операции перемещения данных, которые работают с частями тензоров: Concat, Split, StridedSlice, Select.
- Специализированные операции для моделей определенного типа: DetectionOutput, RegionYolo, PriorBox.

2.4 Inference Engine

2.4.1 API Inference Engine

Inference Engine - это набор библиотек C++ с привязкой к C и Python, предоставляющих общий API для создания решений для инференса на вашей

платформе. API Inference Engine используется для чтения промежуточного представления (IR), ONNX и выполнения модели на целевых девайсах. API Inference Engine позволяет:

- Прочитать модель из файлов (IR)
- Загрузить модель в плагин, работающий с конкретным устройством
- Отправить данные для обработки
- Получить результаты обработки

Inference Engine использует архитектуру плагинов. Плагин Inference Engine – это программный компонент, который содержит полную реализацию для выполнения инференса на определенном аппаратном устройстве Intel: CPU, GPU, VPU и т.д. Главная идея – это единый API для разных устройств, выпускаемых Intel, позволяющий запускать нейронные сети, оставляя при этом возможность на каждом устройстве выполнить какую-либо тонкую настройку, то есть передавать какие-то специфические параметры при запуске сети, если это необходимо.

Приведенная далее схема иллюстрирует стандартный рабочий процесс развертывания обученной модели:

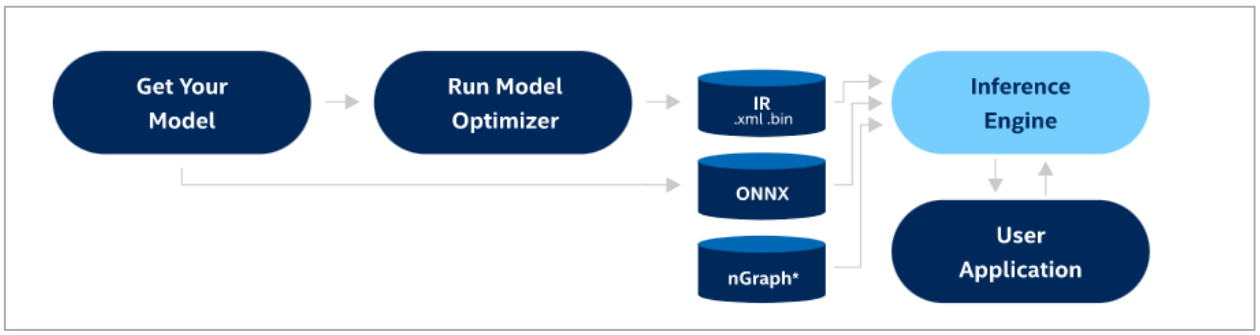

Рисунок 2.4.1.1 – Процесс развертывания обученной модели

nGraph - это внутреннее представление графов в инструментарии OpenVINO, которое используется для построения модели из исходного кода.

Интеграция Inference Engine

Следующая схема иллюстрирует стандартный рабочий процесс Inference Engine Python API:

Рисунок 2.4.1.2 – Стандартный рабочий процесс Inference Engine API

Использование API Inference Engine

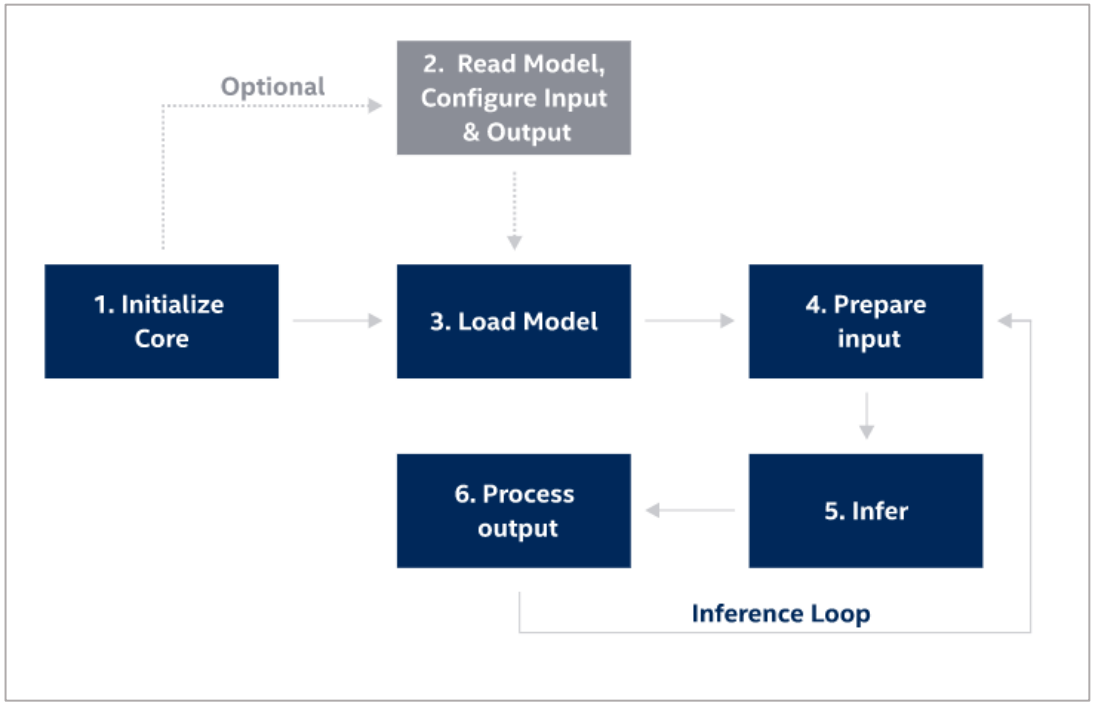

Рисунок 2.4.1.3 – Конвейер инференса

В этом разделе приведены пошаговые инструкции по реализации типичного конвейера выводов с помощью Inference Engine API:

1. Создание Inference Engine Core для управления доступными устройствами и чтения сетевых объектов

```
ie = IECore()
```

2. (Необязательно). Считывание модели. Настройка входов и выходов модели
3. Загрузка модели в устройство

```
exec_net = ie.load_network(network= "model.xml", device_name="CPU")
```

4. Подготовка входного слоя

   Например, конвертация изображения в формат NCWH с типом FP32.

```
input_data = np.expand_dims(np.transpose(image, (2, 0, 1)),
0).astype(np.float32)
```

5. Начало инференса

```
result = exec_net.infer({input_name: input_data})
```

6. Обработка результатов инференса

```
output = result[output_name]
```

2.4.2 Программный стек Inference Engine

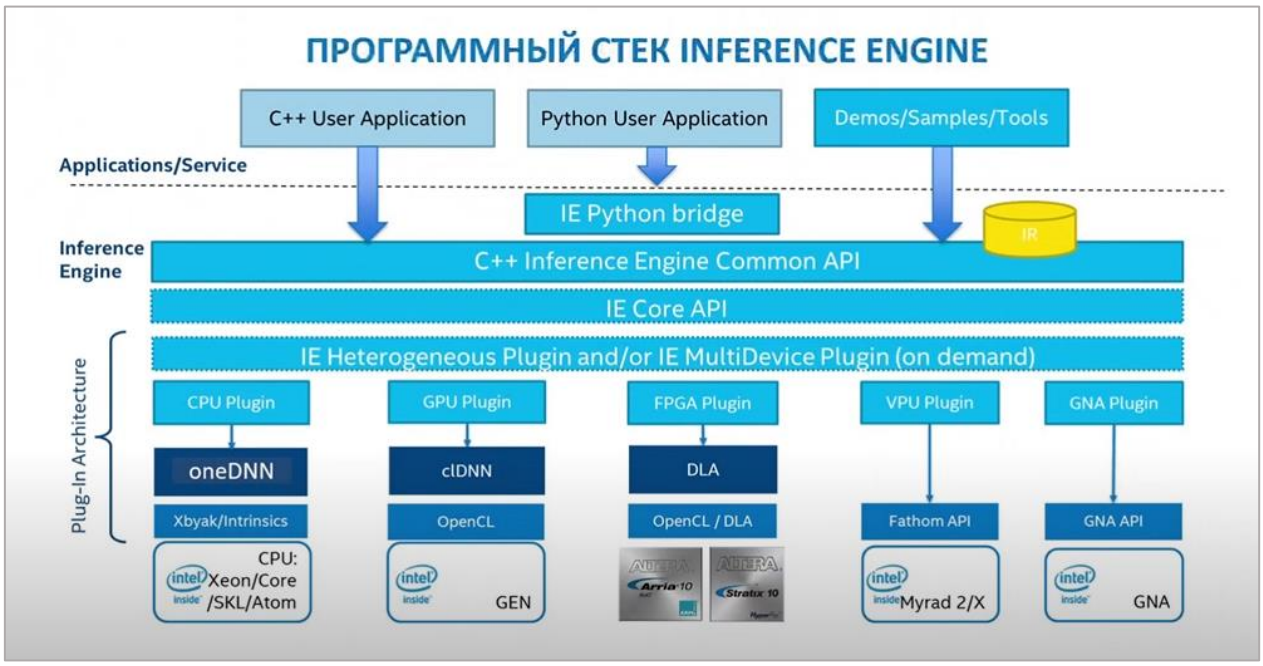

Рисунок 2.4.2.1 – Программный стек IE

Основная идея в том, что приложение пользователя, написанное на Python/C++ через некий API, позволяет включить в работу один из плагинов устройства, на котором запускается сеть.

С одной стороны, в Inference Engine есть интерфейс, которым пользуются пользователи для загрузки своего кода, с другой стороны есть интерфейс, который позволяет добавить поддержку любого устройства.

Каждый из плагинов, который относится к каждому конкретному устройству может иметь свои низкоуровневые библиотеки.

В общем Inference Engine имеет множество разноуровневых задач: есть какие-то общие задачи, общие оптимизации, которые не зависят от устройства, есть низкоуровневые оптимизации под конкретное устройство и есть совсем низкоуровневые оптимизации, уже перед включением железа в работу (например, примитивы из OpenCL или средства парализации, относящиеся к конкретному процессору). За счет всех оптимизаций Inference Engine как раз и достигает высокой производительности.

2.4.3 Примитивы памяти Inference Engine

- *Blobs*

InferenceEngine::Blob – это основной класс, определенный для работы с памятью. С помощью данного класса вы можете читать и записывать в память, получать информацию о структуре памяти и т.д.

Верным способом создания объектов Blob с определенной компоновкой является использование конструкторов с InferenceEngine::TensorDesc.

- *Layouts*

InferenceEngine::TensorDesc - это специальный класс, предоставляющий описание формата компоновки.

Этот класс позволяет создавать планарные раскладки, используя стандартные форматы (такие как InferenceEngine::Layout::NCDHW, InferenceEngine::Layout::NCHW, InferenceEngine::Layout::NC, InferenceEngine::Layout::C и т.д.), а также непланарные раскладки, используя InferenceEngine::BlockingDesc.

Для создания сложной компоновки следует использовать InferenceEngine::BlockingDesc, который позволяет определить блокированную память со смещениями и страйдами.

2.4.4 Низкоточный 8-битный целочисленный инференс

Для вычисления 8-разрядных целых чисел модель должна быть квантована. Можно использовать квантованную модель из OpenVIN Toolkit Intel's Pre-Trained Models или квантовать модель самостоятельно. Для квантования можно использовать Post-Training Optimization Tool.

Процесс квантования добавляет слои FakeQuantize в активации и веса для большинства слоев.

Когда плагину OpenVINO передается квантованный IR, плагин автоматически распознает его как квантованную модель и выполняет 8-битный вывод. Если передается квантованная модель другому плагину, который не поддерживает 8-битный вывод, но поддерживает все операции из модели, модель будет выведена с точностью, которую поддерживает этот плагин.

Во время инференса квантованная модель загружается в подключаемый модуль. Плагин использует элемент Low Precision Transformation для обновления модели, чтобы вывести ее с низкой точностью:

1. Обновление слоев FakeQuantize, для того чтобы иметь квантованные выходные тензоры в диапазоне низкой точности, и добавление слоев деквантизации для компенсации обновления. Слои деквантизации проталкиваются через как можно большее количество слоев, чтобы иметь больше слоев с низкой точностью. После чего большинство слоев имеют квантованные входные тензоры в диапазоне низкой точности и могут быть выведены с низкой точностью. По-хорошему, слои деквантизации должны быть соединены в последующем слое FakeQuantize.
2. Веса квантуются и хранятся в слоях Constant.

2.4.5 Оптимизация развертывания

Для оптимизации результатов производительности на этапе выполнения можно использовать:

- Предварительную обработку
- Режим пропускной способности
- Асинхронное API
- Снижение точности инференса
- Оптимизацию устройств
- Комбинацию устройств

2.4.6 Режим пропускной способности

Одним из методов увеличения эффективности вычислений является обработка батчей, которая объединяет множество входных изображений для достижения оптимальной пропускной способности. Внутренние ресурсы выполнения разделяются/распределяются по потокам выполнения. Применение данной функции позволяет получить значительно более высокую производительность для сетей, которые изначально плохо масштабируются с количеством потоков (например, легкие топологии).

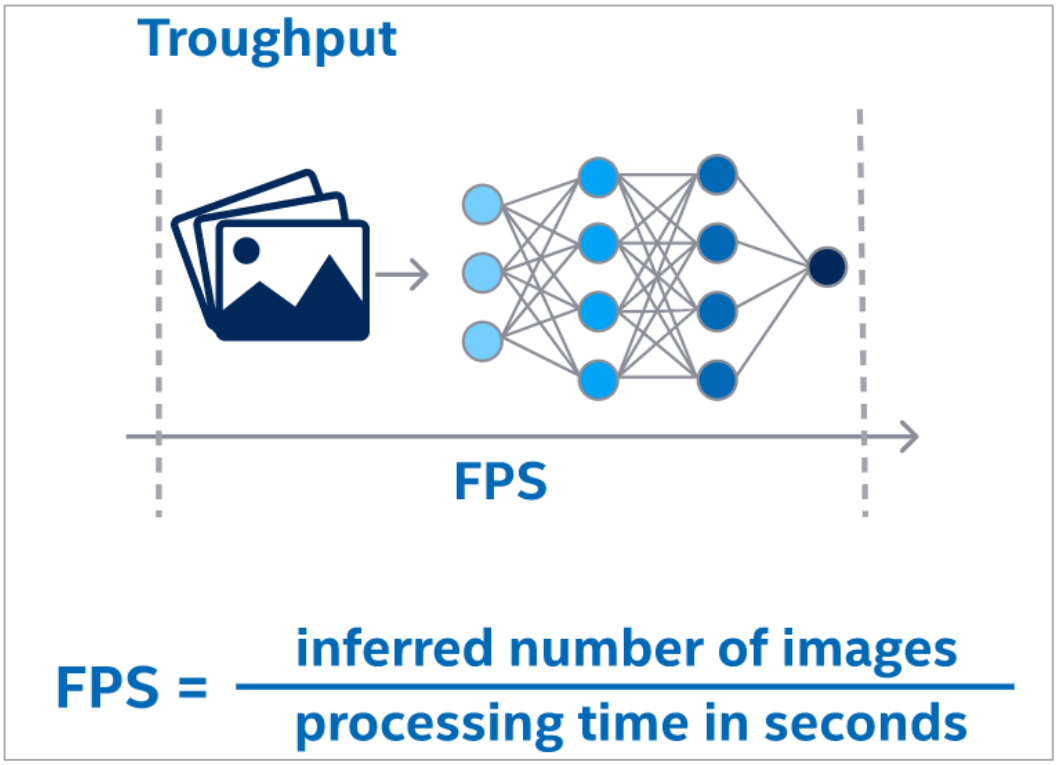

Рисунок 2.4.6.1 – Пропускная способность

Режим пропускной способности ослабляет требование насыщения процессора за счет использования большого пакета: параллельный запуск нескольких независимых запросов на инференс зачастую дает значительно лучшую производительность, чем использование только батча. Это позволяет облегчить логику приложения, так как нет необходимости объединять несколько входов в батч для достижения хорошей производительности процессора. Вместо этого можно держать отдельный запрос на инференс для каждой камеры или другого источника входных данных и обрабатывать запросы параллельно, используя асинхронное API.

2.4.7 Асинхронное API Inference Engine

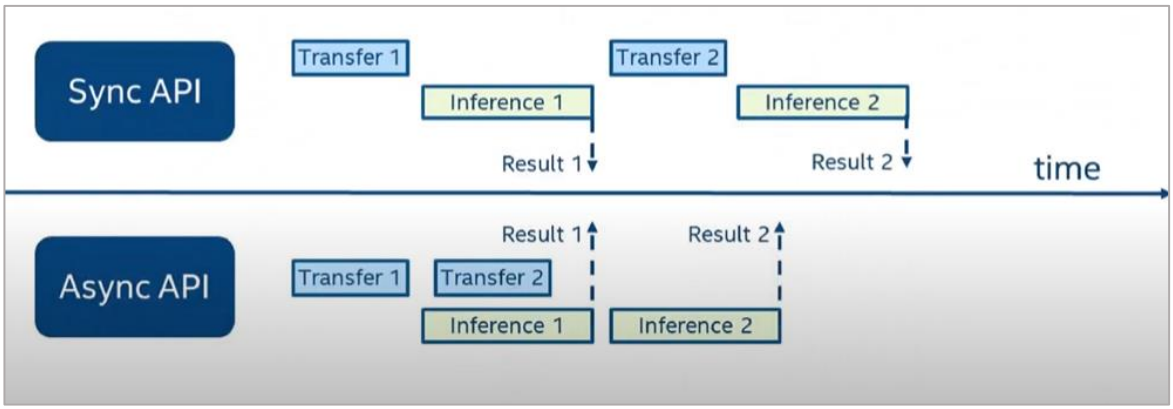

Рисунок 2.4.7.1 – Синхронный и асинхронный режимы

Асинхронное API Inference Engine позволяет увеличить общую частоту кадров приложения. Пока ускоритель занимается инференсом, приложение может продолжать работу на хосте, а не ждать завершения инференса.

В приведенном примере инференс применяется к результатам декодирования видео. Таким образом, можно вести два параллельных запроса выполнения, и пока обрабатывается текущий, происходит захват входного кадра для следующего.

- В синхронном режиме обычно, кадр захватывается с помощью OpenCV, а затем сразу же обрабатывается, то есть выполнение блокируется до исполнения предыдущего запроса:

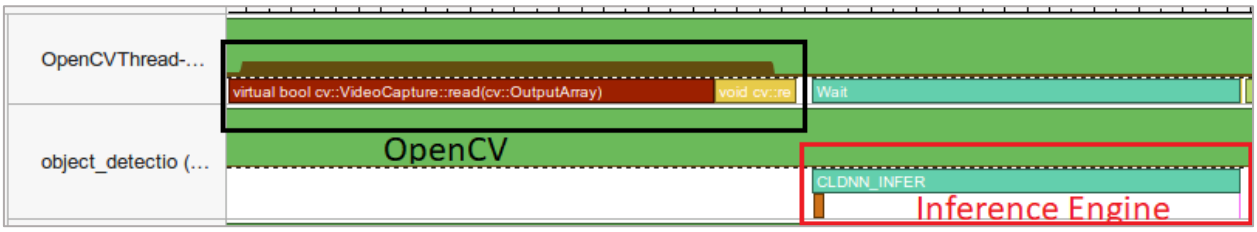

Рисунок 2.4.7.2 – Синхронный режим

- В асинхронном режиме следующий запрос заполняется в основном потоке, в то время как текущий запрос обрабатывается, то есть накидываются запросы на обработку и не дожидаясь выполнения этого запроса вызывается следующий, таким образом они и накапливаются. Окончание отслеживается с помощью механизма callback:

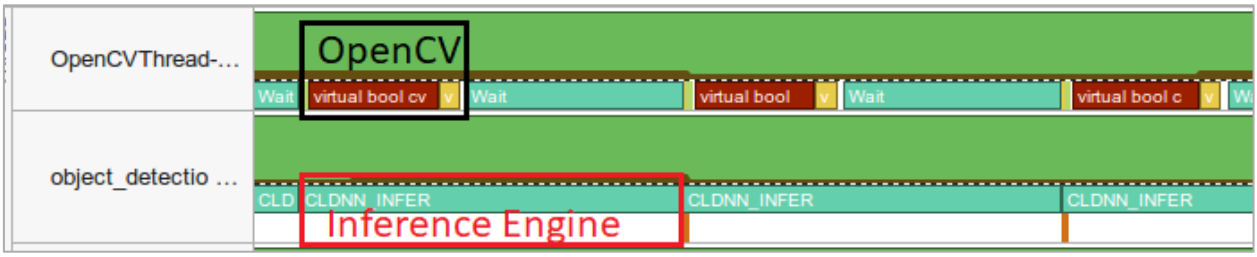

Рисунок 2.4.7.3 – Асинхронный режим

2.4.8 Автоматическое снижение точности инференса

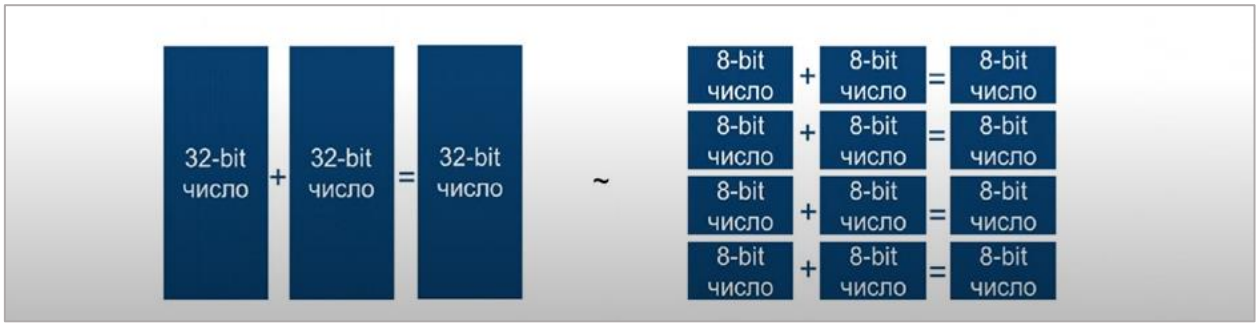

Рисунок 2.4.8.1 – Использование вычислений с меньшей разрядностью

Точность инференса прямо влияет на производительность. Использование вычислений с меньшей разрядностью – позволяет, например, при небольшой потере точности выполнить гораздо больше операций за такт.

Model Optimizer может создавать IR с различной точностью. Например, IR FP16 изначально предназначен для устройств VPU и GPU, тогда как, например, для CPU IR FP16 обычно масштабируется до обычного FP32

автоматически при загрузке. Но доступны дополнительные опции точности инференса для определенного устройства, например, 8-битное целое или bfloat16. Для мультидевайсного плагина, который поддерживает автоматический инференс на нескольких устройствах параллельно, вы можете использовать FP16 IR (FP32 не требуется).

По умолчанию плагины включают оптимизацию, которая позволяет использовать более низкую точность, если сохраняется удовлетворительный диапазон точности. Например, для CPU, поддерживающего инструкции AVX512_BF16, модель FP16/FP32 преобразуется в bfloat16 IR для ускорения инференса.

2.5 Поддержка плагинов для устройств

Inference Engine использует архитектуру плагинов. Плагин Inference Engine – это программный компонент, который содержит полную реализацию для вывода на определенном аппаратном устройстве Intel: CPU, GPU, VPU, GNA и т.д. Каждый плагин реализует унифицированный API и предоставляет дополнительные API, специфичные для конкретного оборудования.

Inference Engine предоставляет возможности для инференса моделей глубокого обучения на представленных далее типах устройств с соответствующими плагинами.

2.5.1 Плагины CPU и GPU

Плагин для CPU был разработан для достижения высокой производительности нейронных сетей на CPU с использованием библиотеки Intel Math Kernel Library for Deep Neural Networks. В настоящее время плагин CPU использует Intel Threading Building Blocks для распараллеливания вычислений.

Плагин для GPU использует библиотеку Intel Compute Library for Deep Neural Networks для вывода глубоких нейронных сетей. clDNN – это библиотека производительности с открытым исходным кодом для

приложений Deep Learning, предназначенная для ускорения вывода глубокого обучения на графических процессорах Intel.

Плагины CPU/GPU поддерживают несколько алгоритмов оптимизации графа, таких как слияние или удаление слоев:

- Снижение точности выводов

Плагин CPU использует стандартный подход к оптимизации. Этот подход означает, что выводы делаются с меньшей точностью, если на данной платформе возможно достичь лучшей производительности с приемлемым диапазоном точности.

- Слияние слоев свертки и простых слоев

  Слияние слоя свертки и любого из простых слоев, перечисленных ниже:

  1. Активации: ReLU, ELU, Sigmoid, Clamp.
  2. Глубокие: ScaleShift, PReLU
  3. FakeQuantize
- Объединение слоев Pooling и FakeQuantize
- Объединение слоев FullyConnected и Activation
- Объединение слоев свертки и глубокой свертки, сгруппированных с простыми слоями
- Объединение слоев свертки и суммирования
- Объединение группы сверток

2.5.2 Мультидевайсный плагин

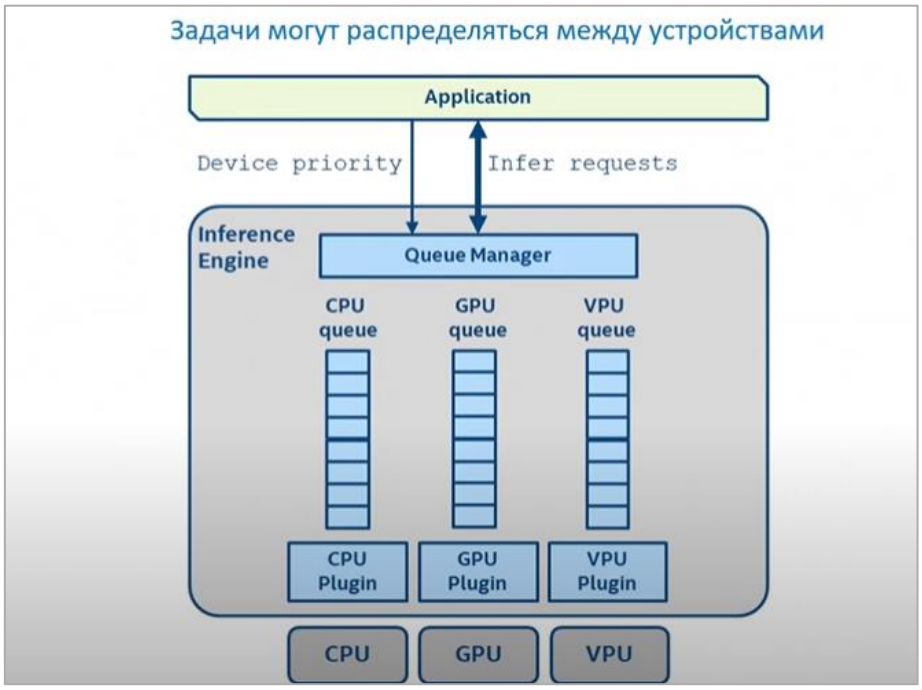

Рисунок 2.5.2.1 – Мультидевайсный режим работы

Плагин Multi-Device автоматически распределяет запросы на инференс по доступным вычислительным устройствам для параллельного выполнения запросов. В отличие от него, гетерогенный плагин, речь о котором пойдет далее, может запускать разные слои на различных устройствах, но не параллельно. Потенциальными преимуществами плагина Multi-Device являются:

1. Увеличение пропускной способности за счет использования нескольких устройств (по сравнению с выполнением на одном устройстве).

2. Более стабильная производительность, так как устройства разделяют нагрузку на выводы (если одно устройство слишком занято, другое может взять на себя больше нагрузки).

При применении плагина Multi-Device логика приложения остается неизменной, поэтому не нужно явно загружать сеть на каждое устройство, создавать и балансировать запросы на инференс и т.д. С точки зрения приложения, это просто еще одно устройство, которое обрабатывает фактический механизм.

Настройка плагина Multi-Device может быть описана тремя основными шагами:

1. Настройка каждого устройства как обычно
2. Загрузка сети в плагин Multi-Device, созданный поверх (приоритетного) списка настроенных устройств. Это единственное изменение, необходимое в приложении.
3. Как и при любом другом вызове ExecutableNetwork (в результате load_network), вы создаете столько запросов, сколько необходимо для насыщения устройств.

2.5.3 Гетерогенный плагин

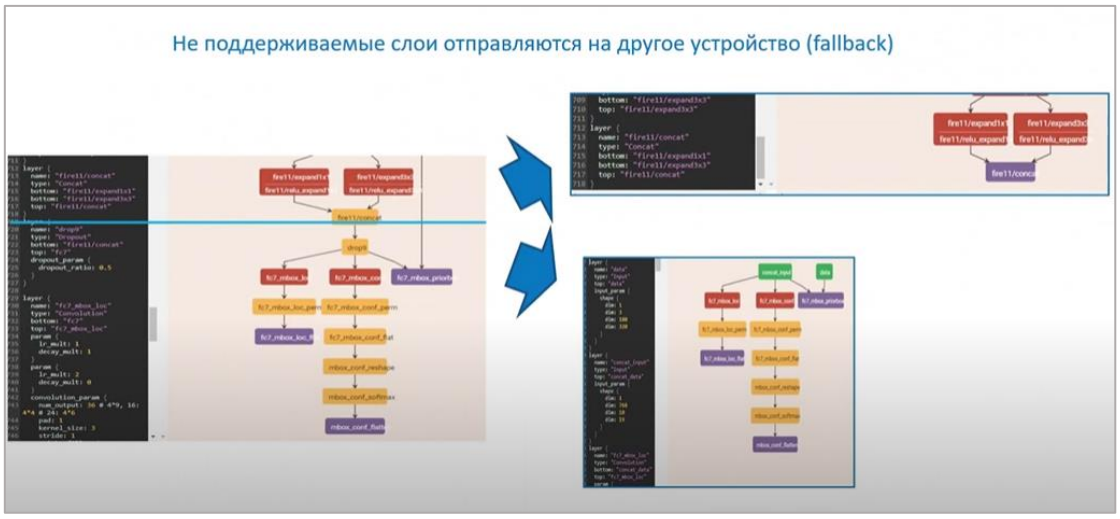

Рисунок 2.5.3.1 – Гетерогенный режим работы

Гетерогенный плагин позволяет производить инференс одной сети на многих устройствах, то есть отправлять не поддерживаемые слои на другое устройство (fallback). Целями инференса сетей в гетерогенном режиме являются:

- Использование мощности ускорителей для обработки особо тяжелых частей сети и выполнять неподдерживаемые уровни на резервных устройствах, таких как CPU.
- Эффективнее утилизировать все доступные аппаратные средства в течение одного инференса.

Инференс через гетерогенный плагин можно разделить на два независимых этапа:

1. Установка аппаратного сродства к слоям
2. Загрузка сети в гетерогенный плагин, разделение сети на составляющие и выполнение их через плагин.

Эти шаги разделены. Настройка сродства может быть выполнена автоматически или в ручном режиме.

Автоматическая политика инициирует "жадное" поведение и назначает все слои, которые могут быть выполнены на определенном устройстве, в соответствии с установленными приоритетами. Автоматическая политика не учитывает особенности плагина, такие как неосуществимость инференса некоторых слоев без других специфических слоев, размещенных до или после этого слоя. Если плагин устройства не поддерживает топологию подграфа, построенную гетерогенным плагином, то необходимо установить сродство вручную.

Отдельные топологии плохо поддерживаются для гетерогенного выполнения на определенных устройствах или вообще не могут быть выполнены в этом режиме.

2.6 Дополнительные утилиты

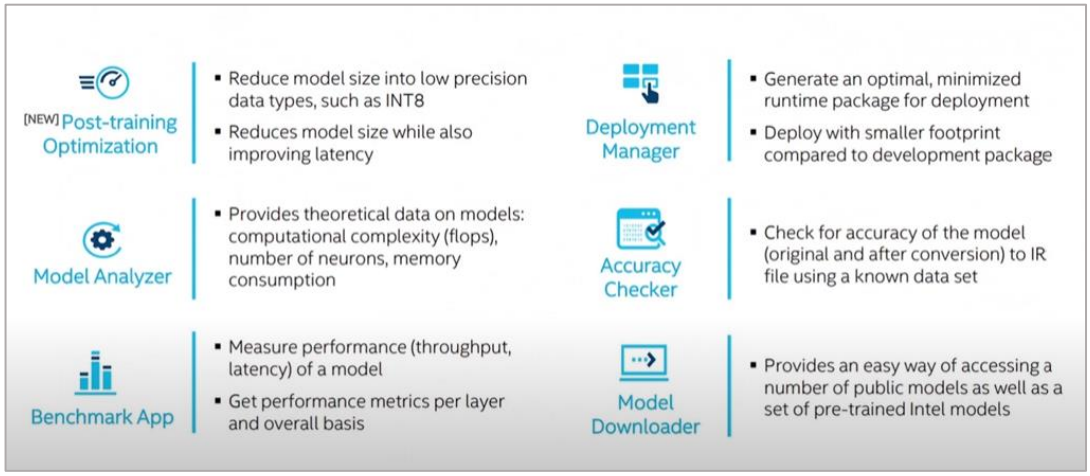

Рисунок 2.6.1 – Дополнительные утилиты

- Post-training Optimization – позволяет оптимизировать нейронную сеть с использованием целочисленной арифметики, то есть например перейти из FP16 в INT8. Проблема в том, что теряется точность при переходе из чисел с плавающей запятой в целые числа. Возникает проблема нахождения scale-фактора, коэффициента преобразования из вещественных чисел в целые. Если взять слишком маленькое, то получится слишком маленький входной диапазон, а если слишком большое, то число может банально не влезть в INT и будет переполнение, что плохо скажется на работе сети. Данная утилита запускается на реальных данных и для каждого слоя сети выясняет диапазон входных данных и таким образом получается подобрать коэффициент преобразования scale.
- Model Analyzer – позволяет узнать теоретический предел производительности, который можно достичь с вашим железом и сравнить с реальным результатом.
- Benchmark App – позволяет с помощью специфичных методологий, разработанных Intel, измерить скорость работы сети.
- Deployment Manager – позволяет минимизировать набор компонентов, который требуется для разработки, то есть позволяет устанавливать не весь пакет OpenVINO, а какие-то его определенные инструменты.
- Accuracy Checker – позволяет измерить точность модели

- Model Downloader – позволяет загружать модели из открытого зоопарка OpenVINO

ЛИТЕРАТУРА

1. https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html
2. https://docs.openvino.ai/latest/openvino_docs_install_guides_installing_openvino_windows.html
3. https://docs.openvino.ai/2021.4/openvino_docs_IE_DG_supported_plugins_MULTI.html?sw_type=switcher-python
4. https://gist.github.com/dkurt/bc567159e5173a6e86c5781badff9ac6
5. https://delta-course.org/docs/IntelCVSchool/IntelCVSchoolOpenVINO.pdf
6. https://www.youtube.com/watch?v=kY9nZbX1DWM&list=PLDKCjIU5YH6jMzcTV5_cxX9aPHsborbXQ
7. https://github.com/opencv/open_model_zoo
8. https://habr.com/ru/company/intel/blog/549634/
9. https://habr.com/ru/company/intel/blog/433772/
10. https://neptune.ai/blog/image-processing-python