Квантование нейронных сетей

Александр Крапухин, Дмитрий Крапухин, Андрей Стоцкий, Влад Шахуро

При обучении нейросетей используются числа с плавающей точкой (обычно формата FP32). Такие числа обладают широким диапазоном значений и высокой точностью (маленькое расстояние между двумя соседними числами), позволяя эффективно обучать сложные нейросетевые модели.

Однако при инференсе FP32-модели работают медленно, а также требуют много памяти и электроэнергии. Кроме того, необходима поддержка вычислений с плавающей точкой со стороны устройства. Это затрудняет использование нейросетей в приложениях, которые должны работать в режиме реального времени на маломощных устройствах – смартфонах, AR/VR-гарнитурах, беспилотных автомобилях, дронах и носимой электронике.

Один из наиболее эффективных способов оптимизации нейросетей для инференса – квантование, когда вместо чисел с плавающей точкой используются целые числа.

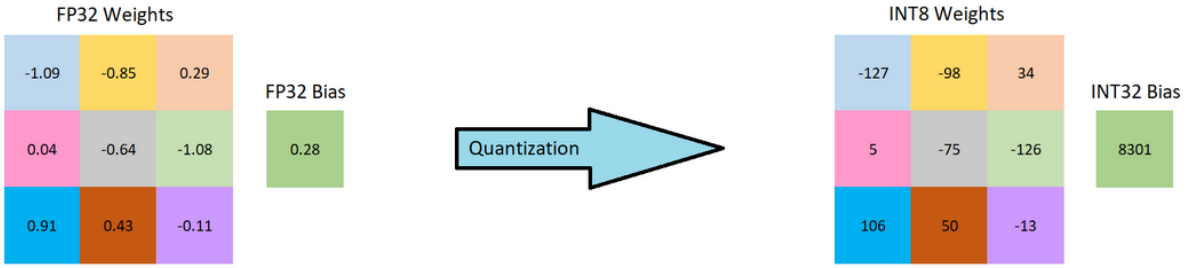

Рис. 1: Квантование весов и сдвигов

Преимущества квантования:

- Уменьшение размера весов модели в 4 раза.
- Ускорение работы нейросети. Некоторые процессоры быстрее обрабатывают 8-битные данные, а также снижается нагрузка на оперативную память и более эффективно используется кэш.
- Уменьшение энергопотребления за счет сокращения количества вычислений и обращений к памяти.
- Возможность запуска на устройствах, не поддерживающих вычисления с плавающей точкой (DSP-процессоры, нейропроцессоры, микроконтроллеры, ПЛИСы).

Недостатки:

- Снижение точности модели. Квантованная модель является аппроксимацией FP32-модели, и ее итоговая точность снижается.

Преимущества оказываются существеннее, чем снижение точности (обычно незначительное). Поэтому квантование широко применяется на практике, а в Pytorch и Tensorflow есть встроенные модули квантования.

В этом задании вам предстоит:

- Вручную (при помощи NumPy) проквантовать простую сверточную модель классификации изображений.
- Выполнить квантование, используя фреймворки Tensorflow и Pytorch.
- Сравнить точность квантованных моделей с исходной FP32-моделью.

Ручное квантование позволит лучше понимать, как происходит процесс квантования, и как квантованная модель работает при инференсе. При ручном квантовании будет использована схема квантования TensorFlow Lite, она проще и нагляднее Pytorch-схемы (Pytorch-квантование находится в статусе беты).

Квантованная модель будет обладать следующими свойствами:

- Веса будут переведены в формат INT8, сдвиги (biases) в формат INT32.
- Модель будет принимать на вход и отдавать на выход данные формата INT8.
- При инференсе все вычисления будут выполняться с целыми числами.
- Активации каждого слоя будут INT8-числами, при этом в процессе их вычисления может использоваться большее количество бит для повышения точности и предотвращения переполнений.

Критерии оценки

Максимальная оценка за задание — 5 баллов.

Файлы должны располагаться следующим образом:

```
├── solution.py
├── run.py
├── fp32_model.py
├── manual_quantization.py
├── tensorflow_quantization.py
├── pytorch_quantization.py
├── comparison.py
└── tests/
    ├── 00_unittest_compute_quantization_params_input/
    ├── 01_unittest_quantize_input/
    └── ...
```

Для решения задания нужно заполнить в файле `solution.py` фрагменты кода, помеченные `# Your code goes here`. В проверяющую систему нужно загружать только файл `solution.py`.

1 Обучение FP32-модели

Сначала обучим простую сверточную модель для классификации цифр на датасете MNIST.

`$ python fp32_model.py`

Скрипт должен создать файл `fp32_model.pt`.

Код модели:

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.conv = nn.Conv2d(1, 12, 3)  # 12 filters of size [1, 3, 3]
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(2, 2)
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(2028, 10)

    def forward(self, input):
        x = self.conv(input) # [B,  1, 28, 28] -> [B, 12, 26, 26]
        x = self.relu(x)     # [B, 12, 26, 26]
        x = self.maxpool(x)  # [B, 12, 26, 26] -> [B, 12, 13, 13]
        x = self.flatten(x)  # [B, 12, 13, 13] -> [B, 2028]
        x = self.fc(x)       # [B, 2028] -> [B, 10]
        return x
```

Теперь мы должны проквантовать получившуюся обученную модель.

2 Формула квантования

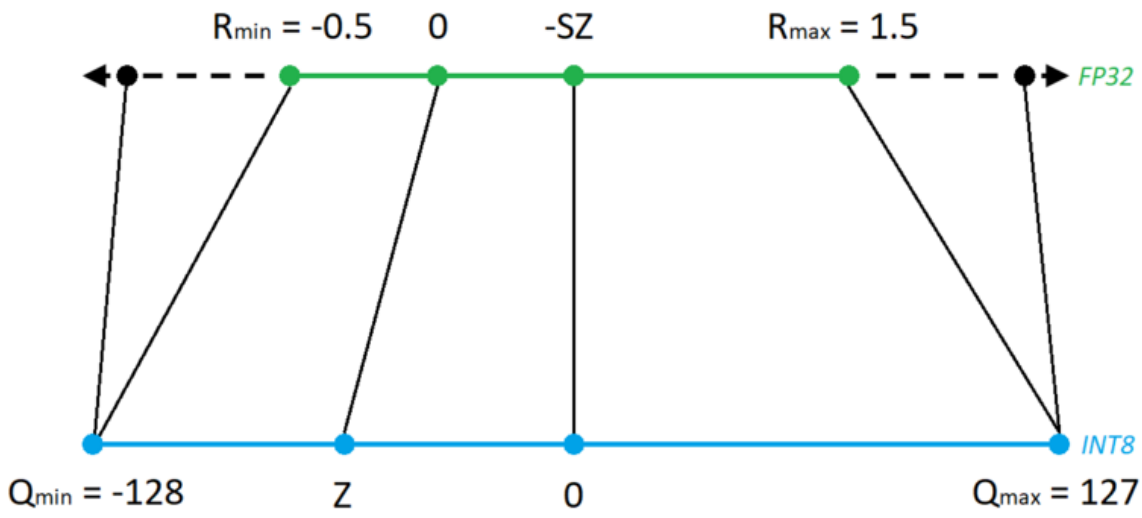

Рис. 2: Отображение FP32-чисел в INT8-числа

Формат FP32 позволяет представить около 4 миллиардов чисел в диапазоне $[-3.4^{38}, 3.4^{38}]$. Для сравнения, формат INT8 позволяет представить всего 255 чисел в диапазоне $[-128, 127]$ (two's complement representation).

Two's complement representation, или Дополнительный код, – самый распространенный способ представления отрицательных чисел в компьютерах.

Однако при квантовании нет необходимости преобразовывать в INT8 весь диапазон $[-3.4^{38}, 3.4^{38}]$, достаточно квантовать ограниченный диапазон $[R_{min}, R_{max}]$. Чтобы преобразовать диапазон FP32-чисел $[R_{min}, R_{max}]$ в диапазон квантованных целых чисел $[Q_{min}, Q_{max}]$, можно использовать линейное преобразование вида $q = \frac{r}{S} + Z$, где r это FP32-число, а q это соответствующее ему квантованное число. При этом крайние точки диапазона $[R_{min}, R_{max}]$ должны отображаться в крайние точки диапазона $[Q_{min}, Q_{max}]$. Составим систему уравнений, чтобы найти S и Z:

$$\begin{cases} Q_{min} = \frac{R_{min}}{S} + Z \\ Q_{max} = \frac{R_{max}}{S} + Z \end{cases}$$

Решив систему уравнений, получим формулы для параметров квантования S (scale) и Z (zero point):

$$S = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}} \tag{1}$$

$$Z = round(\frac{R_{max}Q_{min} - R_{min}Q_{max}}{R_{max} - R_{min}}) \tag{2}$$

В формуле для Z используется округление, так как параметр Z является представлением нуля из диапазона $[R_{min}, R_{max}]$ в диапазоне $[Q_{min}, Q_{max}]$ и должен быть целым числом. Также, целочисленный Z позволяет упростить вычисления, как будет видно далее.

Таким образом, формула квантования принимает следующий вид:

$$q(r) = clip(round(\frac{r}{S} + Z)) \tag{3}$$

где:

q – целое квантованное INT8-число.

r – вещественное FP32-число.

S – scale, соотношение диапазонов.

Z – zero point, представление нуля в диапазоне целых чисел.

$round$ – округление до ближайшего целого числа.

$clip$ – функция, ограничивающая максимальное и минимальное значения. Если вещественное число меньше R_{min}, оно отображается в Q_{min}, а если больше R_{max}, то в Q_{max}.

Также существует **обратная функция квантования** – она используется для преобразования целого числа в вещественное:

$$\widetilde{r} = S(q - Z) \tag{4}$$

Если последовательно выполнить квантование и обратное квантование, то полученное значение не совпадет с исходным ($r \neq \widetilde{r}$) и возникнет ошибка квантования.

Задание 1 (0.5 балла): Напишите функцию `compute_quantization_params`, которая принимает на вход диапазоны $[R_{min}, R_{max}]$, $[Q_{min}, Q_{max}]$ и вычисляет масштаб (S) и нулевую точку (Z). Функция должна возвращать объект класса `QuantizationParameters`. Масштаб S должен иметь тип `np.float64`, нулевая точка Z должна иметь тип `np.int32`.

Рекомендация по всем заданиям – будьте внимательны с типами данных.

Проверьте реализацию с помощью юнит-теста (добавьте скрипту `run.py` права на выполнение с помощью команды `chmod +x run.py`):

```
$ ./run.py unittest compute_quantization_params
```

Имея `QuantizationParameters` (масштаб и нулевую точку), можно перевести числа из диапазона вещественных чисел $[R_{min}, R_{max}]$ в диапазон целых чисел $[Q_{min}, Q_{max}]$, и наоборот.

Задание 2 (0.5 балла): При помощи формул квантования (3) и обратного квантования (4) напишите:

- Функцию `quantize` (0.25 балла). Она принимает тензор с числами типа `np.float64` и параметры квантования, и возвращает тензор такой же формы с числами типа `np.int8`. Числа, находящиеся за пределами диапазона $[R_{min}, R_{max}]$, должны отображаться в крайние точки диапазона $[Q_{min}, Q_{max}]$.
- Функцию `dequantize` (0.25 балла). Обратная операция – функция принимает тензор с целыми `np.int8`-числами и параметры квантования, и вычисляет соответствующий тензор с `np.float64`-числами. Обратите особое внимание на типы при вычитании $q - Z$.

Проверьте функции с помощью юнит-тестов:

```
$ ./run.py unittest quantize
```

```
$ ./run.py unittest dequantize
```

3 Калибровка

Калибровка – процесс определения границ диапазона вещественных чисел $[R_{min}, R_{max}]$ для последующего квантования.

Основные способы определения границ:

- Взять минимальные и максимальные значения – самый простой способ. В большинстве случаев работает хорошо, но при наличии выбросов диапазон $[R_{min}, R_{max}]$ может оказаться излишне большим, ошибка квантования внутри диапазона вырастет, а точность квантованной модели снизится. Именно этот способ применяется в TFLite и будет использован в этом задании.
- Минимизация энтропии. Использовать расстояние Кульбака – Лейблера (KL-расстояние) для минимизации потери информации между оригинальными FP32-значениями и квантованными значениями.
- Использовать процентиль. Например, оставить 99% значений и отбросить 1% самых больших.
- Минимизация среднеквадратичной ошибки (MSE) между FP32-значениями и квантованными значениями.

Веса модели после обучения фиксированы, поэтому диапазоны для них можно определить, просто посчитав их минимумы и максимумы. Но значения активаций зависят от входных данных и могут

существенно отличаться от сэмпла к сэмплу. Поэтому на практике для калибровки на вход FP32-модели подается репрезентативный датасет (обычно достаточно 100-500 изображений из обучающей выборки). В процессе вычислений минимальные и максимальные значения активаций фиксируются и затем используются для определения диапазонов.

Для фиксации минимальных и максимальных значений используются так называемые "наблюдатели" (observers). В SimpleNet они будут использованы в трех местах (см. класс SimpleNetWithObservers):

- На входе. Для определения диапазона квантования входных данных.
- После ReLU-активации сверточного слоя. Свертка, batchnorm (при наличии) и функция активации квантуются как одно целое для оптимизации вычислений (Conv-BN-ReLU fusion).
- После полносвязного слоя.

Задание 3 (0.5 балла): Реализуйте метод `call` для "наблюдателя", который задается классом `MinMaxObserver`. Метод обновляет минимальное и максимальное значения в полях `min` и `max`. Поля должны иметь тип `np.float64`. Класс `SimpleNetWithObservers` является копией класса `SimpleNet` с добавленными наблюдателями.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest minmax_observer
```

4 Квантование сверточного слоя

4.1 Формула свертки

В нашей нейронной сети сверточный слой состоит из фильтров размера 3x3. Поэтому можно представить веса одного фильтра (w) и входные данные (x) в виде векторов, состоящих из 9 элементов. При свертке эти векторы поэлементно перемножаются, складываются, а затем к ним прибавляется bias (b):

$$y = \sum_{i=1}^{9} w^{(i)} x^{(i)} + b$$

В квантованной нейросети это выражение должно вычисляться, используя только целочисленную арифметику. Для этого сначала заменим в формуле каждое вещественное число на соответствующее ему деквантованное число, используя формулу $\widetilde{r} = S(q - Z)$:

$$S_y(q_y - Z_y) = \sum_{i=1}^{9} S_w(q_w^{(i)} - Z_w) S_x(q_x^{(i)} - Z_x) + S_b(q_b - Z_b)$$

Перепишем выражение – вынесем множители $S_w S_x$ из-под знака суммы и раскроем скобки. Получим **формулу свертки**:

$$S_y(q_y - Z_y) = S_w S_x (\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} - Z_w \sum_{i=1}^{9} q_x^{(i)} + \sum_{i=1}^{9} Z_w Z_x) + S_b(q_b - Z_b) \tag{5}$$

Цель – вычислить q_y, int8-значение результата свертки, которое затем будет подаваться на вход следующего слоя. Далее будут рассмотрены методы, позволяющие упростить эту формулу и произвести вычисления, используя только целые числа.

4.2 Квантование весов

4.2.1 Симметричное квантование (Symmetric quantization)

Начнем упрощение формулы свертки с квантования весов. Симметричное квантование – оптимизация при квантовании весов, при которой диапазон значений весов расширяется и становится симметричным относительно нуля. Это приводит к тому, что Z_w становится равен нулю. Например, если минимальное значение среди весов равно -1.25, а максимальное 1.5, то при квантовании минимальное значение снижается до -1.5. Подставим в формулу свертки (5) $Z_w = 0$:

$$S_y(q_y - Z_y) = S_w S_x(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)}) + S_b(q_b - Z_b) \tag{6}$$

4.2.2 32-битный аккумулятор

При вычислении скалярного произведения $\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)}$ перемножаются 8-битные INT8-числа $q_w^{(i)}$ и $q_x^{(i)}$. В результате получаем произведение – 16-битное число. Эти 9 чисел постепенно складываются в аккумуляторе. Аккумулятор 32-битный – чтобы избежать переполнения при суммировании 16-битных произведений.

Таким образом, промежуточные вычисления могут использовать большее количество бит для обеспечения необходимой точности. Но входной тензор каждого слоя, веса и выходной тензор являются 8-битными числами.

4.2.3 Потензорное и поканальное квантование (per-tensor and per-channel quantization)

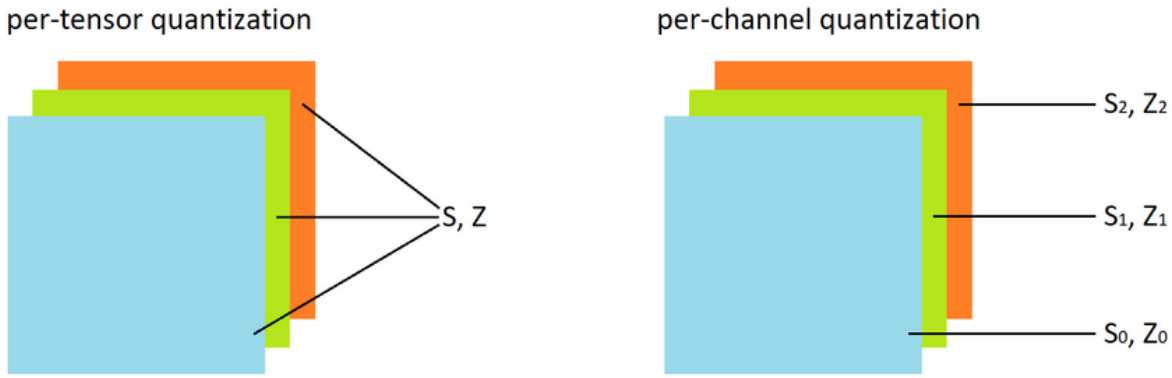

Рис. 3: Потензорное и поканальное квантование

Параметры квантования – масштаб (S) и нулевая точка (Z) могут быть общими для всех каналов или вычисляться отдельно для каждого канала. Например, в нашей модели с 12 фильтрами можно использовать:

- Потензорное квантование – один общий масштаб и одна общая нулевая точка для всех 12 фильтров.
- Поканальное квантование – отдельные масштаб и нулевая точка для каждого фильтра (суммарно получится 12 пар).

В схеме квантования TFLite для весов сверточного слоя используется поканальное симметричное квантование, а для полносвязного слоя потензорное симметричное квантование.

4.2.4 Урезанный диапазон квантования

Еще одна оптимизация, часто применяемая при квантовании весов, заключается в том, чтобы использовать диапазон $[-127, 127]$ вместо $[-128, 127]$. Это позволяет более эффективно использовать особенности архитектуры ARM. Если веса никогда не равны -128, то их умножение с входным значением никогда не равно $-128 \cdot (-128)$ – следовательно, абсолютное значение произведения всегда меньше $2^7 \cdot 2^7 = 2^{14}$. Это позволяет складывать два произведения в локальном int16-аккумуляторе и только затем прибавлять этот 16-битный результат к основному 32-битному аккумулятору. При этом такой урезанный диапазон используется только при квантовании весов, активации могут принимать значение -128.

Задание 4 (1 балл): Реализуйте функции

1. `quantize_weights_per_tensor` - потензорное квантование (0.5 балла). При решении необходимо будет **для всего тензора весов**:
   - Найти минимальное и максимальное значения весов.
   - При помощи функции `compute_quantization_params` найти параметры квантования.
   - Использовать параметры квантования в функции `quantize` для квантования весов.

   Используйте **симметричное квантование с урезанным диапазоном** $[-127, 127]$. На вход функция принимает тензор с весами формата `np.float64`, на выход отдает тензор такой же формы с квантованными весами типа `np.int8` и параметры квантования (`QuantizationParameters`).
2. `quantize_weights_per_channel` - поканальное квантование (0.5 балла). При решении необходимо будет **для каждого фильтра**:
   - Найти минимальные и максимальные значения весов.
   - При помощи функции `compute_quantization_params` найти параметры квантования.
   - Использовать параметры квантования в функции `quantize` для квантования весов.

   Используйте **симметричное квантование с урезанным диапазоном** $[-127, 127]$. На вход функция `quantize_weights_per_channel` принимает тензор с весами формата `np.float64`, на выход отдает тензор такой же формы с квантованными весами (`np.array`) типа `np.int8` и параметры квантования **для каждого фильтра** в виде списка (`List[QuantizationParameters]`).

Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest quantize_weights_per_tensor
```

```
$ ./run.py unittest quantize_weights_per_channel
```

4.3 Квантование сдвигов (bias quantization)

Вернемся к формуле (6). В аккумуляторе хранится результат вычисления $(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)})$ в виде INT32-числа. Нам нужно умножить его на $S_w S_x$ и прибавить bias $S_b(q_b - Z_b)$. Чтобы упростить вычисления, можно принять масштаб S_b равным масштабу аккумулятора $S_w S_x$, а Z_b принять равным нулю.

Кроме того, имеет смысл преобразовывать bias в INT32-число, так как его в любом случае пришлось бы конвертировать в INT32-формат для суммирования с INT32-аккумулятором. Также 32-битный сдвиг позволяет увеличить точность квантованной нейросети. При этом сдвигов гораздо меньше, чем весов, поэтому 32-битные сдвиги не оказывают существенного влияния на общий размер параметров, сокращение по памяти остается почти 4-х кратным.

Тогда формула принимает следующий вид:

$$S_y(q_y - Z_y) = S_w S_x(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b) \tag{7}$$

Значение $-Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b$ не зависит от входных данных, все числа в нем известны до запуска модели. Соответственно, можно заранее вычислить его один раз и затем инициализировать 32-битный аккумулятор этим числом (вместо нуля), немного уменьшив количество требуемых при инференсе вычислений.

Задание 5 (0.5 балла): Реализуйте функцию `quantize_bias` для квантования сдвигов. При решении необходимо воспользоваться функцией квантования (3) из п.2.

$$q(r) = clip(round(\frac{r}{S} + Z))$$

На вход функция `quantize_bias` принимает сдвиг в формате `np.float64`, масштаб весов и масштаб входных данных. На выходе отдает квантованный сдвиг в формате `np.int32`. На практике 32 бит обычно более чем достаточно для квантования сдвига, поэтому при выполнении задания вы можете игнорировать опасность того, что bias может выйти за пределы INT32 диапазона $[-2147483648, 2147483647]$ при квантовании.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest quantize_bias
```

4.4 Масштабирование аккумулятора

Вернемся к вычислению результата свертки q_y:

$$q_y = \frac{S_w S_x}{S_y}(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b) + Z_y$$

В аккумуляторе сейчас находится целочисленный результат (INT32) выражения в скобках ($\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b$). Этот результат нужно умножить на вещественное число $\frac{S_w S_x}{S_y}$, используя только целые числа. Для этого будем использовать арифметику с фиксированной точкой (Fixed-point arithmetic). Нужно представить $\frac{S_w S_x}{S_y}$ в виде числа с фиксированной точкой и умножить его на значение аккумулятора.

4.4.1 Числа с фиксированной точкой

В числах с фиксированной точкой есть целая часть (биты до точки) и дробная часть (биты после точки). При переводе из бинарного представления в десятичное биты слева от точки будут умножаться на неотрицательные степени двойки, а биты справа – на отрицательные.

Примеры 8-битных чисел с фиксированной точкой:

$$0.110\,0000 = 1 \cdot (2^{-1}) + 1 \cdot (2^{-2}) = 0.75$$
$$01.10\,0000 = 1 \cdot (2^{0}) + 1 \cdot (2^{-1}) = 1.5$$

С точки зрения битовых значений эти два числа одинаковые: оба равны 0110 0000 (96 в десятичной системе). Однако меняя положение точки, мы можем по-разному интерпретировать эти битовые последовательности.

Предположим, что мы хотим умножить числа 0.75 и 1.5, используя 8-битный умножитель (8x8-bit multiplier). Подаем на вход умножителя два одинаковых 8-битных числа 0110 0000 (96), на выходе получаем 16-битное число 0010 0100 0000 0000 (9216). Фиксированная точка у результата находится после 3 бита (число бит, кодирующих целую часть произведения, равно сумме "целых" бит множителей: 1 + 2 в нашем случае). Тогда 16-битное произведение интерпретируется следующим образом:

$$001.0\,0100\,0000\,0000 = 1 \cdot (2^0) + 1 \cdot (2^{-3}) = 1 + 0.125 = 1.125$$

4.4.2 Представление множителя в виде числа с фиксированной точкой

Обозначим $M = \frac{S_w S_x}{S_y}$ и перепишем формулу квантованного результата свертки q_y:

$$q_y = M(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b) + Z_y \tag{8}$$

Вычислим произведение M на значение аккумулятора $(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b)$. Для этого аппроксимируем M следующим образом:

$$M \approx 2^{-n} M_0$$

где:

n – целое число.

M_0 – 32-битное неотрицательное число с фиксированной точкой, которая находится после первого бита. Остальные биты (31) отведены под дробную часть. При этом первый бит справа от точки всегда равен 1, то есть M_0 кодирует число в диапазоне $[0.5, 1)$.

Такое разделение M на два множителя позволяет получить максимальную точность, так как M_0 всегда находится в одном диапазоне $[0.5, 1)$ и имеет 31 бит точности, а умножение на 2^{-n} можно произвести простым битовым сдвигом.

4.4.3 Алгоритм умножения

Рассмотрим алгоритм умножения аккумулятора на множитель M на примере:

- В аккумуляторе хранится число 909.
- Множитель M равен числу 0.039062500014.

Будем считать, что в компьютере используется 32-битный умножитель и 32-битный сумматор (32x32-bit multiplier, 32-bit adder).

Алгоритм умножения:

1. Аппроксимируем M:

   Нужно найти такое n, при котором M_0 окажется в диапазоне $[0.5, 1)$. В нашем случае $n = 4$, тогда $M_0 = 0.625$:

   $$M \approx 2^{-n} M_0 = 2^{-4} \cdot 0.625 = 0.0390625$$

   M_0 кодируется 32-битным числом, в котором 1 бит используется для целой части и 31 бит для дробной:

   $$0.101\,0000\,0000\,0000\,0000\,0000\,0000\,0000 = 1 \cdot (2^{-1}) + 1 \cdot (2^{-3}) = 0.5 + 0.125 = 0.625$$

При этом в памяти компьютера число хранится как целое:

$$0101\,0000\,0000\,0000\,0000\,0000\,0000\,0000_2 = 1342177280_{10}$$

M_0 может принимать значения

$$\text{от } 0100\,0000\,0000\,0000\,0000\,0000\,0000\,0000$$
$$\text{до } 0111\,1111\,1111\,1111\,1111\,1111\,1111\,1111$$

(от 1073741824 до 2147483647 в десятичной системе).

2. Умножаем значение аккумулятора на M_0. На вход 32-битного умножителя (32x32-bit multiplier) подаются два 32-битных числа – значение аккумулятора и M_0 (без указания положения точки)

$$0000\,0000\,0000\,0000\,0000\,0011\,1000\,1101 \text{ и } 0101\,0000\,0000\,0000\,0000\,0000\,0000\,0000$$

Умножитель возвращает 64-битное произведение:

$$0000\,0000\,0000\,0000\,0000\,0001\,0001\,1100\,0001\,0000\,0000\,0000\,0000\,0000\,0000\,0000$$

В этом числе 33 бита кодируют целую часть (32 бита от аккумулятора + 1 бит от M_0). Ставим точку после 33-го бита:

$$0000\,0000\,0000\,0000\,0000\,0001\,0001\,1100.001\,0000\,0000\,0000\,0000\,0000\,0000\,0000 =$$

$$= 1 \cdot (2^9) + 1 \cdot (2^5) + 1 \cdot (2^4) + 1 \cdot (2^3) + 1 \cdot (2^{-3}) = 512 + 32 + 16 + 8 + 0.125 = 568.125$$

Мы произвели умножение $909 \times 0.625 = 568.125$

3. Умножаем полученное число на 2^{-4}. Такое умножение выполняется простым сдвигом точки влево (в нашем случае на $n = 4$ позиции):

$$0000\,0000\,0000\,0000\,0000\,0001\,0001\,1.100\,0001\,0000\,0000\,0000\,0000\,0000\,0000\,0000 =$$

$$= 1 \cdot (2^5) + 1 \cdot (2^1) + 1 \cdot (2^0) + 1 \cdot (2^{-1}) + 1 \cdot (2^{-7}) = 32 + 2 + 1 + 0.5 + 0.0078125 = 35.5078125$$

Теперь целая часть кодируется 29 битами (было 33 бита и мы сдвинули точку на 4 бита влево)

4. Отбрасываем дробную часть. Результат должен быть целым числом, поэтому нас интересуют 29 бит (кодируют целую часть) и один бит после точки для округления. Запоминаем, что первый бит после точки равен 1 и сдвигаем результат на 35 позиций вправо, отбрасывая все биты после точки:

$$0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0010\,0011 =$$

$$= 1 \cdot (2^5) + 1 \cdot (2^1) + 1 \cdot (2^0) = 32 + 2 + 1 = 35$$

5. Конвертируем из 64 бит в 32 бита (так как у нас 32-битный сумматор).

$$0000\,0000\,0000\,0000\,0000\,0000\,0010\,0011$$

6. Округляем и получаем итоговый результат. Так как первый бит после точки был равен 1 – добавляем единицу к числу и получаем:

$$0000\,0000\,0000\,0000\,0000\,0000\,0010\,0100 = 36$$

7. Сравним полученный результат с обычным умножением $909 \times 0.039062500014 = 35.5078 \approx 36$. Результаты совпадают с точностью до целого.

Задание 6 (1 балл): Напишите функцию `quantize_multiplier`. Функция принимает на вход вещественное число (множитель M). Вам нужно реализовать пункт 1 **Алгоритма умножения**:

- Получить значение n, при котором M_0 окажется в диапазоне $[0.5, 1)$ (как в пункте 1: $M \approx 2^{-n} M_0$). Можно воспользоваться циклом или вывести математически.
- Представить получившееся значение M_0 в виде десятичного числа с 32 битами (для этого можно умножить M_0 на 2^{31} и округлить до целого числа). В примере в пункте 1 вам нужно было бы вернуть число 1342177280.
- Вернуть n и полученное десятичное значение M_0 (оба числа в формате `np.int32`).

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest quantize_multiplier
```

В Numpy при умножении двух чисел типа `np.int8` по умолчанию возвращается результат такого же типа `np.int8`, что приводит к неверному результату умножения:

```
a = np.int8(127)
b = np.int8(126)
wrong_res = a * b # RuntimeWarning: overflow encountered in byte_scalars
print(wrong_res) # -126
print(type(wrong_res)) # <class 'numpy.int8'>
print(np.binary_repr(wrong_res, 8)) # 1000 0010
```

В результате получится неправильный ответ `-126` (в бинарном виде `1000 0010`). Ошибка возникает из-за того, что в качестве результата берутся младшие 8 бит из 16.

Для правильного ответа используйте функцию `np.multiply` с указанием `dtype`:

```
res = np.multiply(a, b, dtype=np.int16)
print(res) # 16002
print(type(res)) # <class 'numpy.int16'>
print(np.binary_repr(res, 16)) # 0011 1110 1000 0010
```

Результат будет правильным: `16002` (в бинарном виде `0011 1110 1000 0010`).

Задание 7 (1 балл): Напишите функцию `multiply_by_quantized_multiplier`. Функция принимает на вход значение аккумулятора, а также n и M_0 из предыдущего задания. Вам нужно реализовать пункты 2-6 **Алгоритма умножения**:

- Умножить аккумулятор на M_0 и получить 64-битное произведение. Для умножения двух `np.int32` чисел нужно указать `dtype=np.int64` в функции `np.multiply`.
- Определить позицию фиксированной точки, учитывая умножение на 2^{-n}
- Запомнить значение первого бита после точки для округления.
- Отбросить дробную часть.
- Привести результат произведения из формата `np.int64` в формат `np.int32` (для упрощения можно исходить из предположения, что результат всегда умещается в INT32).
- При необходимости произвести округление, прибавив единицу.
- Вернуть 32-битный результат произведения аккумулятора на $2^{-n} M_0$.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest multiply_by_quantized_multiplier
```

4.5 Итоговый результат свертки

Вернемся к формуле квантованного результата свертки (8) (п 4.4.2):

$$q_y = M(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b) + Z_y$$

Мы вычислили произведение M на значение аккумулятора: $M(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b)$.

Осталось выполнить две операции:

- Прибавить нулевую точку Z_y.
- Конвертировать получившееся INT32-число в INT8-число (выход слоя должен иметь тип INT8). При этом, числа меньше -128 преобразуются в -128, а числа больше 127 в 127.

Итоговая формула принимает вид:

$$q_y = clip(M(\sum_{i=1}^{9} q_w^{(i)} q_x^{(i)} - Z_x \sum_{i=1}^{9} q_w^{(i)} + q_b) + Z_y)$$

Класс `QuantizedConvReLU` в `manual_quantization.py` объединяет в себе все функции для квантования сверточного слоя.

5 Квантование полносвязного слоя

Функции, написанные для квантования сверточного слоя, применяются и для квантования полносвязанного слоя. Единственное отличие – использование потензорного квантования (per-tensor symmetric quantization) вместо поканального. См. класс `QuantizedLinear` в `manual_quantization.py`.

6 Квантованная модель SimpleNet

Все написанные для квантования классы и функции объединены в классе `SimpleNetQuantized` в `manual_quantization.py`. Функция `call` принимает INT8-тензор, поочередно вызывает квантованные слои и возвращает INT8-результат.

Проквантовать модель, используя написанные функции, можно следующим образом:

```
# load trained fp32-model
fp32_model = SimpleNet()
checkpoint = torch.load("fp32_model.pt", weights_only=True)
fp32_model.load_state_dict(checkpoint)
fp32_model.eval()

# create SimpleNet model with the same weights and with observers
fp32_model_with_observers = SimpleNetWithObservers()
fp32_model_with_observers.load_state_dict(fp32_model.state_dict())
fp32_model_with_observers.eval()

# calibrate
train_dataset = torchvision.datasets.MNIST(
    "mnist",
    train=True,
    download=True,
    transform=torchvision.transforms.ToTensor(),
)
with torch.no_grad():
    for image_idx in range(500):
        fp32_model_with_observers(train_dataset[image_idx][0].unsqueeze(0))

observations = fp32_model_with_observers.get_minmax_observations()

# compute quantization parameters based on observations
qp_input = compute_quantization_params(
    observations["input"].min,
    observations["input"].max,
    -128,
    127,
)
qp_conv_relu = compute_quantization_params(
    observations["conv_relu"].min,
    observations["conv_relu"].max,
    -128,
    127,
)
qp_fc = compute_quantization_params(
    observations["fc"].min,
    observations["fc"].max,
    -128,
    127,
)

# convert fp32-parameters to numpy
parameters = list(fp32_model.parameters())
parameters_np = [p.detach().cpu().numpy() for p in parameters]

# Initialize quantized model by passing fp32 numpy parameters and quantization parameters.
# Quantization will be done during initialization
quantized_model = SimpleNetQuantized(parameters_np, qp_input, qp_conv_relu, qp_fc)
```

7 Встроенные модули квантования в PyTorch и TensorFlow

7.1 Квантование в TensorFlow (опционально)

Все, что мы делали вручную в предыдущих пунктах, можно сделать автоматически в TFLite. Наша квантованная вручную модель и TFLite-модель должны иметь сопоставимую точность.

Если у вас нет установленного TensorFlow, вы можете не запускать команду ниже (это не влияет на оценку). Тогда квантованная TFLite-модель не будет участвовать в итоговом сравнении моделей.

```
$ python tensorflow_quantization.py
```

Скрипт должен создать tflite-файл с квантованной моделью `quantized_model.tflite`

7.2 Квантование в Pytorch (опционально)

Модуль квантования в Pytorch находится в статусе беты. В нем используются разные схемы квантования для архитектур X86 и ARM. При этом модель, квантованная для ARM, при запуске на X86 будет работать некорректно. С другой стороны, в TFLite используется одна схема, а квантованная модель может запускаться на разных архитектурах с практически одинаковой точностью.

Отметим, что схема квантования для X86 в Pytorch отличается от схемы TFLite. Некоторые отличия:

- диапазон $[R_{min}, R_{max}]$ для активаций определяется с помощью минимизации энтропии (в отличие от min/max у TFLite).
- квантованные веса также имеют тип INT8, но при этом диапазон весов равен $[-128, 127]$ (у TFLite диапазон весов $[-127, 127]$).
- для полносвязного слоя используется per-channel symmetric quantization, в отличие от per-tensor symmetric quantization у TFLite (при квантовании сверточного слоя в обоих фреймворках используется per-channel symmetric quantization).
- активации имеют тип UINT8, при этом используется урезанный диапазон $[0, 127]$ (то есть только 7 бит). В TFLite активации имеют тип INT8 и принимают значения в диапазоне $[-128, 127]$

Из-за beta-статуса API квантования в PyTorch, код в файле `pytorch_quantization.py` может потребовать изменений. Если у вас не получается добиться его корректной работы, вы можете не запускать команду ниже (это не влияет на оценку). Тогда квантованная PyTorch-модель не будет участвовать в итоговом сравнении моделей.

```
$ python pytorch_quantization.py
```

Скрипт должен создать файл `quantized_model.pt`.

8 Сравнение моделей

В данном пункте будет произведено сравнение квантованной вручную модели с оригинальной FP32-моделью, а также с моделями, квантованными с помощью фреймворков (опционально). Точность модели, квантованной вручную, не должна существенно падать по сравнению с FP32-моделью. Также, квантованная вручную модель должна иметь практически такую же точность, как модель, квантованная в TFLite, так как мы использовали схему квантования TFLite.

```
$ python comparison.py
```

Скрипт запустит тестирование всех моделей на 1000 изображений из тестовой части датасета MNIST и отобразит точность каждой модели. Если вы хотите получить точность на всем тестовом датасете, укажите NUM_INPUTS = 10000 в `comparison.py`

9 Заключение

В данном задании был рассмотрен широко применяемый на практике вид квантования – post-training static INT8 quantization. В ходе задания вы вручную произвели квантование простой сверточной модели классификации изображений. Такой ручной подход позволяет лучше понимать, как работает квантование внутри DL-фреймворков, а также каким образом квантованная модель работает при инференсе, используя только целые числа.

10 Ссылки

- Practical Quantization in PyTorch
- Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference
- A Survey of Quantization Methods for Efficient Neural Network Inference
- Quantizing deep convolutional networks for efficient inference: A whitepaper
- Integer Quantization For Deep Learning Inference: Principles And Empirical Evaluation