РЕФЕРАТ

Выпускная квалификационная работа содержит 43 страницы, 15 рисунков, 3 таблицы, 7 листингов, 13 использованных источников.

КВАНТИЗАЦИЯ, МАШИННОЕ ОБУЧЕНИЕ, НЕЙРОННЫЕ СЕТИ, ОПТИМИЗАЦИЯ РАБОТЫ НЕЙРОСЕТЕЙ, C++, ЭФФЕКТИВНОЕ ПЕРЕМНОЖЕНИЕ МАТРИЦ, НИЗКОБИТОВЫЕ ВЫЧИСЛЕНИЯ, ОГРАНИЧЕННОСТЬ РЕСУРСОВ, ПРОМЕЖУТОЧНОЕ ГРАФОВОЕ ПРЕДСТАВЛЕНИЕ.

В данной работе реализуется квантизатор для перевода нейронных сетей в сжатый бинарный формат, в котором работа квантизованных слоёв реализовывается при помощи эффективного алгоритма перемножения матриц, основанного на работе побитовых операций.

В теоретической части работы приводится описание и математическое обоснование метода квантизации. Рассматривается и выводится алгоритм эффективного матричного произведения. Обосновывается формат хранения данных с точки зрения архитектуры современных процессоров.

В практической части описывается процесс поддержки нового квантизованного слоя в рамках нейросетевого фреймворка Samsung ONE, включая сериализацию и десериализацию параметров операции, поддержку слоя в промежуточном графовом представлении и создание вычислительного ядра в рамках интерпретатора нейросетевых моделей. Приводится анализ показателей квантизованного слоя в сравнении с исходной обученной глубокой моделью.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Нейронные сети - мощный и современный инструмент для решения огромного множества задач, начиная от предсказания цен на недвижимость, основываясь на имеющихся характеристиках жилья, заканчивая генерацией или обработкой изображений в задачах компьютерного зрения. Нейронные сети сегодня используются повсюду, мы сталкиваемся с результатами их работы буквально каждый день. Однако возможность их применения ограничивается их ресурсоёмкостью: современные модели имеют миллионы настраиваемых параметров, что накладывает свой отпечаток не только на процесс обучения сети, ведь для обучения одной state-of-the-art нейронной модели необходимо иметь дорогостоящую ЭВМ или даже целый кластер, связанный в единую вычислительную систему, потратить огромное количество времени и электричества на каждую из попыток запуска алгоритма обучения при фиксированном наборе гиперпараметров, но и на использование уже обученных сетей в прикладных задачах, особенно в реальном времени, поскольку перемножение вещественных матриц, лежащее в основе работы практически любых нейронных моделей, является крайне затратной операцией, особенно, когда количество перемножаемых элементов велико.

Над процессом оптимизации работы нейронных сетей на сегодняшний день бьются тысячи ученых и программистов по всему миру, ведь возможность исполнять сети на недорогих устройствах, таких как микроконтроллеры и мобильные телефоны, при как можно меньших затратах ресурсов (энергии, оперативной и внешней памяти, времени на расчеты) привлекает практически любую компанию, заинтересованную в извлечении экономической прибыли на продуктах с использованием глубокого обучения. Особенно хорошо такая перспектива вписывается в концепцию интернета вещей (IoT), поскольку соединение нескольких «умных» устройств в сеть, с возможностью взаимодействовать друг с другом, передавая уже не «сырые», а обработанные локально данные, может заметно упростить повседневную жизнь пользователя и улучшить материальное благосостояние того, кто сможет предложить дешевое и эффективное решение для описанной проблемы.

Для того, чтобы снизить объем вычислений и размер занимаемой памяти, поступающие на вход глубоким моделям данные можно обрабатывать классическими математическими алгоритмами, поскольку простейшая обработка может позволить специалисту по проектированию методов машинного обучения уменьшить глубину сетей и количество нейронов в них при наименьшей потере точности модели. Но даже этого становится недостаточно на сегодняшний день, потому что растущий интерес к искусственному интеллекту побуждает придумывать и применять все новые и новые оптимизации и эвристики для повышения эффективности моделей. Среди наиболее известных следует отметить прунниг (подрезка сети, уменьшение количества параметров), дистилляцию (создание сети меньшего размера, обученной подражать исходной) и квантизацию (переход к другому типу данных парметров сети), являющуюся на текущий момент одним из самых интересных и эффективных методов улучшения качества работы нейронной сети.

Целью данной работы является описание и реализация метода низкобитовой квантизации предварительно обученных нейронных сетей вместе с эффективным алгоритмом работы квантизованных слоев в целочисленном представлении на современных архитектурах ЭВМ.

Для осуществления поставленной цели необходимо выполнить следующие задачи:

— Изучить существующие алгоритмы квантизации и проанализировать их преимущества и недостатки.

— Выбрать фреймворк, в рамках которого можно реализовать предлагаемую в этой работе функциональность.

— Подобрать схему хранения и использования квантизованных значений и параметров.

— Добавить в выбранный фреймворк нейронный слой с квантизованными параметрами.

— Реализовать вычислительное ядро для работы слоя.

— Написать инструмент квантизации сетей.

— Проанализировать показатели полученных моделей.

ОСНОВНАЯ ЧАСТЬ

1 Теоретическая часть

1.1 Классическая квантизация

Чтобы ответить на вопрос, что такое квантизация и что она дает, следует рассмотреть классический вариант, который наиболее прост и популярен на текущий момент, но имеет несколько существенных недостатков.

Квантизация — процесс перевода вещественных параметров нейронной сети в целочисленный формат ограниченной длины бит. В классическом варианте, подробно описанном в [2], производится преобразование всех действительных параметров (весов и входных значений) нейронной сети в целочисленное представление с размером в 8 бит (1 байт) на параметр. При такой схеме квантизации каждое целое число $q \in \{0, ..., 255\}$ обозначает число $\overline{f}$ на вещественной оси, определяемое по (1.1).

$$\overline{f} = s(q - zp) \tag{1.1}$$

А оптимальное целое число, к которому квантуется произвольное число $f \in R$ будет определяться по (1.2).

$$q = round\left(\frac{f}{s}\right) + zp \tag{1.2}$$

Как можно заметить, в данных преобразованиях присутствуют константы s и zp, которые являются параметрами квантизации и используются для преобразования из float-pointing представления мощности континуума, к которому принадлежит число f, в ограниченное множество целых 8-битных чисел (integer), представителем которого является число q. Схема такого преобразования проиллюстрирована на рис.1.1, где показано, как для вещественных чисел определяются соответствующие им целые числа, а для квантизованных значений действительные числа, которые они представляют.

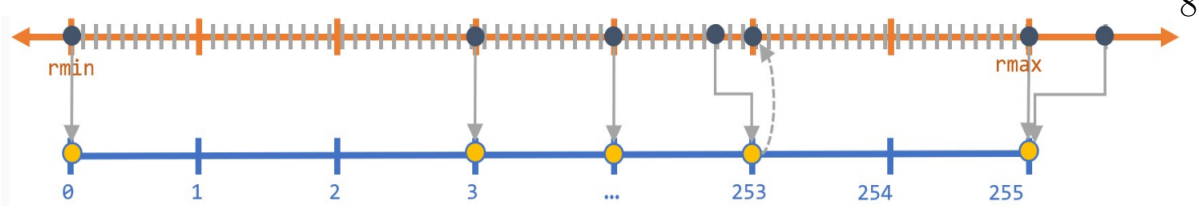


Рис. 1.1 — Схема классической квантизации

Будем называть уровнями квантизации вещественные значения, которые соответствуют всевозможным целочисленным квантизованным значениям q параметра и определяются по формуле (1.1). Они обозначены оранжевыми штрихами на вещественной оси на рис.1.1.

Стоит обратить внимание на то, что в данной схеме 256 уровней квантизации распределены равномерно на отрезке минимального и максимального возможного значения параметра. Он определяется в процессе прогонки репрезентативной выборки реальных данных через исходную сеть. Процесс получения оптимальных параметров квантизации для уже обученной модели при помощи прогонки через неё реальных данных, но без использования известных ответов, под которые обучалась сеть, будет называться в этой работе post-training квантизацией.

Перспективы, которые даёт квантизованное представление параметров:

— Снижается объем вычислений за счет того, что операции сложения и умножения, которые используются в сетях, работают быстрее за счёт целочисленных вычислений и меньшего количества бит на одно число.

— Снижается размер, занимаемый сетью при вычислениях в оперативной памяти.

— Снижается размер, занимаемый сетью при хранении на жестком диске ЭВМ.

— Появляется возможность исполнения моделей на тех устройствах, в которых нет поддержки вещественной арифметики на уровне железа. Такими устройствами являются как микроконтроллеры компании ARM ниже версии Cortex Arm M4 в силу своей низкой стоимости и низкой производительности, так и высокоскоростные нейронные процессоры компа-

ний Huawei и Qualcomm, которые берут на себя исполнение большинства нейронных сетей в современных телефонах и телевизорах.

Однако очевидными являются и недостатки метода. Поскольку мощность поля вещественных чисел выше, чем ограниченного 2^K значениями множества целых чисел, где K — количество бит для кодировки одного параметра, легко догадаться, что данное преобразование будет вести к потерям точности работы обученной сети. Минимизация этой потери - основная задача инструмента, называемого квантизатором нейронных сетей, который выполняет это преобразование, подбирая наиболее оптимальные параметры квантизации zp (zero point — точка, которой соответствует значение 0 в вещественном поле) и s (scale — коэффициент масштабирования).

Равномерность распределения уровней квантизации является слабым местом классического алгоритма, поскольку распределение параметров в сети редко является равномерным, а имеет свою собственную природу, что будет показано далее. Этот факт является ограничителем для использования этого простого алгоритма для квантизации в представления с еще меньшим количеством бит, чем 8, что демонстрируют результаты замеров, полученные в [1], график которых показан на рис.1.2.

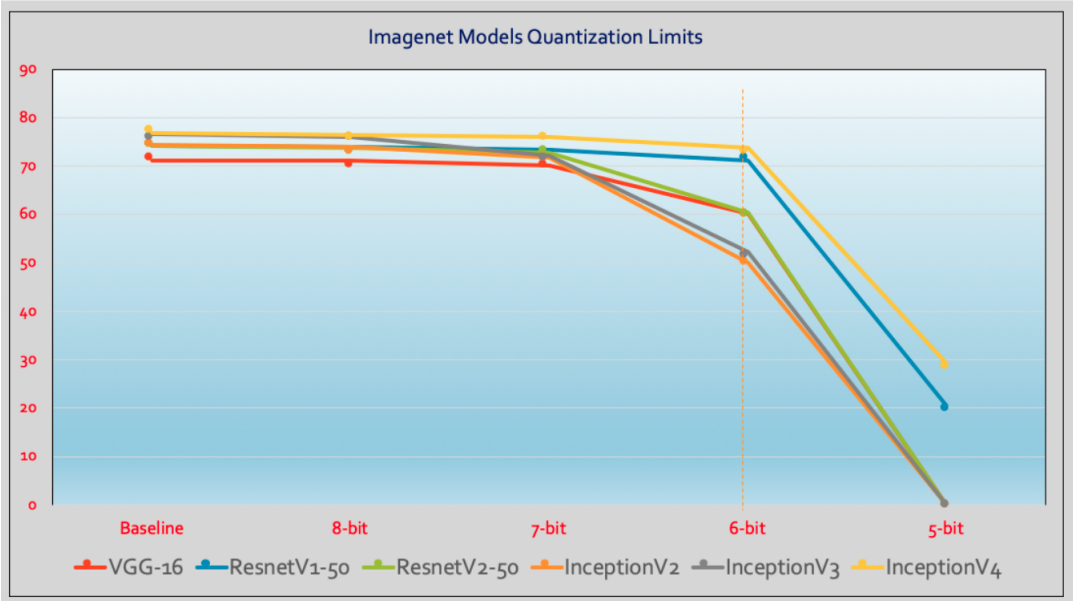


Рис. 1.2 — Точность моделей при уменьшении бит

Этот же факт наталкивает на мысль о том, что для проведения более «экстремальной» квантизации с меньшим количеством бит на параметр, необходимо воспользоваться иными схемами интерпретации целого числа, которые будут лучше подстраивать уровни квантизации новой модели под распределение квантизуемых параметров. Именно о таком методе и будет идти речь в этой работе.

1.2 Обучаемая квантизация

Как уже было сказано выше, в данной работе вводится новый метод для квантизации сетей, который отличается от наиболее популярного и проверенного временем метода с равномерным распределением уровней квантизации. Перед предлагаемым алгоритмом квантизации, также как и перед любым другим, ставятся следующие задачи, которые он должен комплексно решать:

— Определение оптимальных параметров квантизации как можно с меньшим количеством прогоняемых данных, затрачиваемых ресурсов и наибольшей сохраняемой точностью модели.

— Возможность квантизации как можно в меньшие целочисленные размерности весов и активаций нейронной сети с целью уменьшения количества занимаемой памяти.

— Квантизация в такое представление, в котором эффективно производятся операции, тесно связанные с работой нейронных слоев, такие как перемножения матриц и свёртки.

По этой причине был выбран метод обучаемой низкобитовой post-training квантизации, который преобразует веса и активации глубокой модели в формат, позволяющий быстро и эффективно производить перемножения матриц при помощи побитовых операций.

Суть этого метода заключается в том, что предлагается совершенно иной способ представления квантизованного числа в вещественом множестве. Обозначим кортеж $b = (b_0, b_1, ..., b_{K-1}) \in \{0, 1\}^K$ как бинарную запись квантизованного числа в Least Significant Bit first формате. При классической квантизации одно $intK$ (где K — количество бит для хра-

нения одного значения) число q, вычисляемое по формуле (1.2), интерпретируется в действительном множестве как число $\overline{f}$ при помощи 2-ух параметров квантизации s и zp по формуле (1.1).

$$q = \sum_{i=0}^{K-1} b_i 2^i \tag{1.3}$$

В новом методе каждое квантизованное в $intK$ значение с бинарной записью b интерпретируется согласно формуле (1.4), используя $a = (a_0, a_1, ..., a_{K-1})$ - вещественный базис-вектор коэффициентов для каждого из битов бинарного представления целого числа, имеющий длину K.

$$\overline{f} = \sum_{i=0}^{K-1} b_i a_i, \tag{1.4}$$

Иными словами, вещественное число, которое представлено квантизованным числом q вычисляется как скалярное произведение базис-вектора из действительных чисел a с бинарной записью этого квантизованного числа b. Базис-вектор a будет являться настраиваемым параметром для отдельного слоя в процессе post-training квантизации.

Следует оговориться, что из-за того, что веса для каждого нейрона имеют разные распределения (что позволяет извлекать различные признаки из входных данных в рамках одного слоя), следует применять channel-wise квантизацию вместо layer-wise. Это означает наличие своего настраиваемого вещественного базис-вектора у каждого из нейронов слоя, вместо единого вектора для кодирования всех весов одного слоя.

С математической точки зрения это свойство не является обязательным, но в дальнейшем следует считать, что на значения базис-вектора a накладывается строгое отношение частичного порядка: $a0 < a1 < ... < a_{K-1}$. Это свойство дает возможность быстро определять отсортированные уровни квантизации «на лету», что является важным свойством с при реализации.

Стоит отметить, что предлагаемый метод дает возможность интерпретировать биты числа не только в привычном множестве возможных

значений $\{0, 1\}$, но и в $\{-1, 1\}$, что позволяет уровням подстраиваться под отрицательные значения. Именно вариант $\{-1, 1\}$ будет использоваться в дальнейшем в данной работе.

Описанная схема интерпретации числа q на множестве действительных чисел имеет следующее преимущество: уровни квантизации накладываются не равномерно, а могут подстраиваться под конкретное распределение квантизуемого параметра. Например, если веса нейрона имеют стандартное распределение, то уровни квантизации должны иметь большую концентрацию ближе к математическому ожиданию, что позволит точнее представлять величины, которые встречаются в сети наиболее часто. Пример вычисления и распределения уровней квантизации на числовой оси для 3-х битной квантизации показан на рис.1.3.

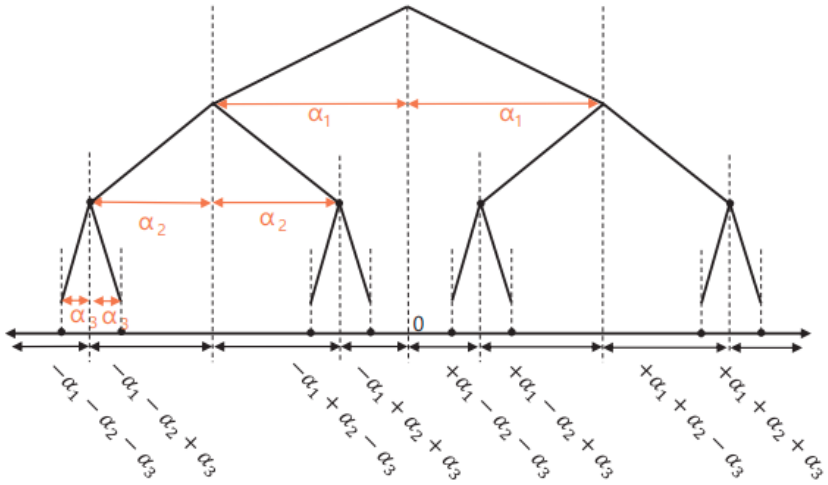


Рис. 1.3 — Распределение уровней квантизации

Если посмотреть на гистограмму весов в каналах слоёв нейронной сети для задач компьютерного зрения ResNet-20, полученное в работе [1], становится понятно, что такой способ представления квантизованных значений точнее описывает генеральное распределение параметров.

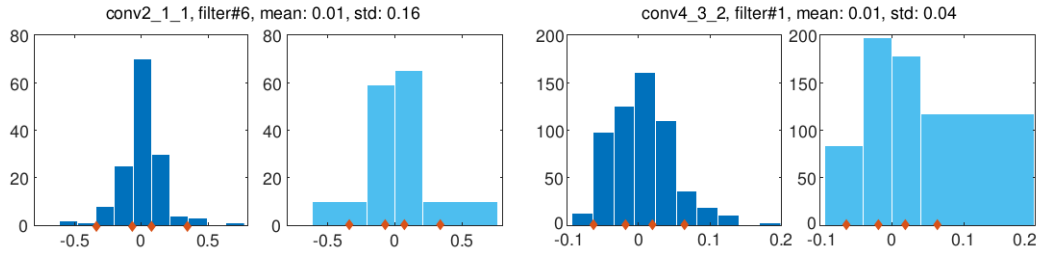


Рис. 1.4 — Распределение весов в ResNet-20

На рис. 1.4. точками оранжевого цвета показано, как можно оптимально разместить уровни при описанном способе кодировки вещественных величин для 2-х битной квантизации весовых параметров. Следует обратить внимание на то, что веса в разных каналах слоя имеют разные распределения, что является обоснованием для выбора channel-wise преобразования вместо layer-wise, о чем было упомянуто выше.

Стоит не забыть упомянуть, что количество уровней квантизации для $intK$ равно 2^K, а сами уровни получаются в результате применения формулы (1.4) ко всевозможным комбинациям бинарных записей из K бит.

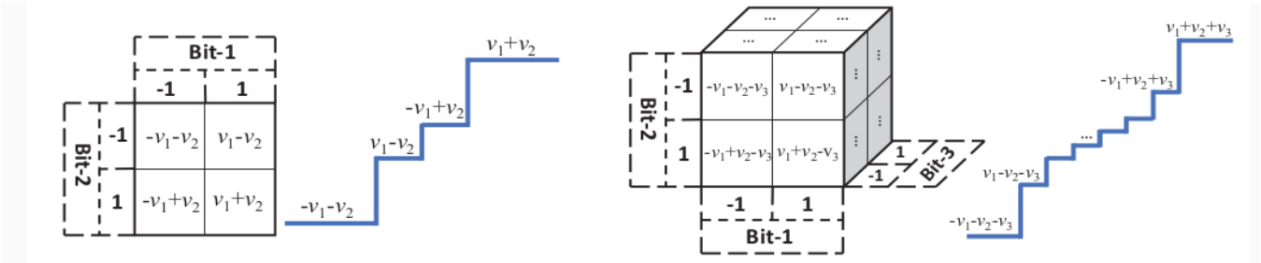


Рис. 1.5 — Уровни квантизации

Возникает вопрос не только, как интерпретировать имеющееся квантизованное число в вещественном представлении, но и каким целочисленным значением оптимальнее всего представить число с действительной оси. Эта часть описываемого алгоритма идентична тому, как аналогичное преобразование производится для классической 8-битной квантизации, однако стоит рассмотреть этот этап поподробнее. Сначала на основе базис-вектора a определяются уровни квантизации $q_l = (a, e_l),\ \forall e_l \in \{-1, 1\}^K$, такие что $q_0 < q_1 < ... < q_{2^K}$, и на основе них рассчитываются

пороги $t_l = \frac{q_l + q_{l-1}}{2}$, с помощью которых определяется к какому уровню следует отнести конкретное число, как показано на рис.1.6.

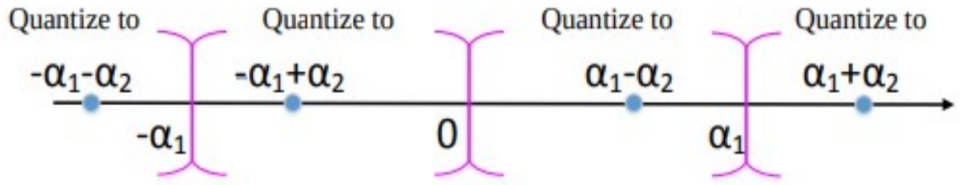


Рис. 1.6 — Определение оптимальной кодировки

После определения уровня квантизации при помощи порогов, следует взять соответствующую бинарную кодировку, которая будет являться оптимальным квантизованным значением для исходного вещественного числа.

Для того, чтобы уровни могли распределяться не только вокруг 0, но и любого другого числа (ведь распределение значений параметра может быть смёщенным), следует добавить свободный коэффициент смещения к обучаемому базис-вектору, что даст дополнительную степень свободы описываемому алгоритму. Однако исследование и решение этой проблемы не затрагивается в данной работе.

У метода есть явный недостаток — большое количество настраиваемых аргументов, равное сумме длин всех базис-векторов. Издержки на хранение таких векторов в памяти достаточно малы по сравнению с количеством весовых значений, поэтому их можно не брать в учет при анализе метода. Но обучение при таком числе параметров квантизации становится трудной задачей, требующей применения различных эвристик, которые позволяют находить оптимумы в вещественных пространствах больших размерностей.

Как было заявлено и будет показано далее в описании работы нового слоя, такой вариант преобразования параметров даёт возможность быстрого матричного перемножения квантизованных весовых и входных значений. Среди огромного разнообразия существующих операций, встречающихся в глубоких моделях на текущий момент, наиболее ярко выделяются полносвязные слои, рекуррентные, свёрточные и субдискретизаци-

онные. Именно на работе этих операций построена любая современная нейронная сеть, поэтому на них следует делать упор при оптимизации работы моделей, в том числе и в квантизации.

Упомянутые субдискретизационные слои являются довольно быстрой операцией, работающей за $O(n)$ от размера входных данных, и издержки на проведение квантизации с учётом потери точности слишком велики по сравнению с выгодой, которую может дать предлагаемое преобразование. Рекуррентные сети, такие как GRU, LSTM, RNN, описанные в [8], [10], [11] соответственно, содержат внутри своей реализации полносвязные нейронные слои. Что касается свёрточных слоев, про которые подробно рассказывается в [6], то операция свёртки в них может быть сведена к перемножению двух матриц (GEMM) при помощи im2col преобразования. Эта техника, описанная в [7], зачастую применяется, поскольку позволяет добиться значительного увеличения скорости работы таких слоёв за счет того, что алгоритм перемножения матриц оптимальнее работает с кешом процессора, благодаря последовательному доступу к памяти. Именно поэтому в данной работе было решено акцентировать внимание на проведение квантизации с классическим полносвязным слоем, поскольку все остальные существующие операции либо содержат такой слой внутри себя, либо могут быть приведены к такому формату, либо издержки и потери превалируют над возможным выигрышем от преобразования, поскольку не содержат в себе ресурсоёмких матричных произведений. Все слои сети, отличные от полносвязных, будут оставаться без изменений в данной работе.

Описанный в [11] классический полносвязный нейронный слой — это набор из нейронов, реализующих в себе алгоритм линейной регрессии вместе с функцией активации, которая призвана добавить нелинейности к выходу нейрона и придать системе большую устойчивость за счет асимптотических ограничений, накладываемых на выход нейрона. Количество таких нейронов задается тем, кто обучает подобный слой. Как нетрудно догадаться, оно равно размерности выходного вектора (output), который выдает слой на основе входного вектора данных (input). Схематично нейрон сети может быть представлен схемой на рис. 1.7.

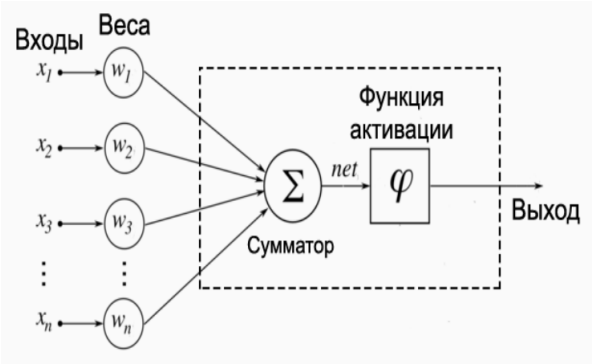


Рис. 1.7 — Нейрон

В то же самое время показанная схема эквивалентна вычислению функции активации σ от суммы между скалярным произведением входов слоя x с вектором весов нейрона w и смещением для i - го нейрона по формуле (1.5).

$$y_i = \sigma((x_i, w_i) + b_i),\ \forall i : 0 < i < m \tag{1.5}$$

Выход такого нейрона — скаляр y_i, называемый выходом нейрона.

Формулу (1.5) можно расширить в более общий случай, когда на вход поступает несколько входных векторов в виде матрицы X и используется целый полносвязный нейронный слой из m описанных выше нейронов, веса которых также могут быть объединены в единую матрицу W, столбцы в которой — весовые вектора для отдельного нейрона. Тогда выходную матрицу $y = (y_0, y_1, ..., y_{m-1})$ можно рассчитать при помощи результата произведения матрицы весов на матрицу входов, к каждой строке которого добавляется вектор смещений $b = (b_0, b_1, ..., b_{m-1})$, состоящий из свободных коэффициентов каждого нейрона слоя, и поэлементно применяется функция активаций σ, что отображено в (1.6).

$$y = \sigma(XW + b) \tag{1.6}$$

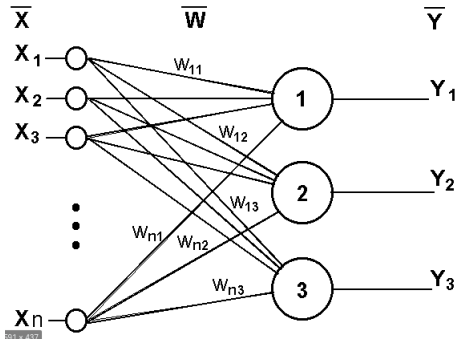


Рис. 1.8 — Полносвязный слой

Основываясь на специфике работы слоя, проиллюстрированного рис. 1.8., можно выделить следующие параметры нейронного слоя, которые необходимо хранить и использовать при расчетах:

— Веса модели в виде матрицы $W \in R^{n \times m}$, где n - размер входных векторов, а m - количество нейронов.

— Вектор из смещений всех нейронов $b \in R^m$.

— Тип функции активации σ, которая применяется к каждому значению выходного вектора.

Поскольку смещения занимают мало места в памяти и сложение происходит за $O(n)$, квантизовать их не имеет смысла. Зато весовые параметры составляют наибольшую часть занимаемого сетью объема, а время на перемножение весов на матрицу входов X является самой ресурсоемкой операцией, оптимизировать которую можно в квантизованных вычислениях, используя побитовую арифметику.

Как было показано выше, поскольку у каждого из нейронов свое распределение весов, то квантизовать следует каждый из его весовых векторов отдельно. При этом в данном методе затрачивается K_w битов на квантизацию каждого веса нейрона. Тогда каждый весовой вектор нейрона может быть представлен бинарной матрицей $B^w \in \{-1, 1\}^{K_w \times n}$, где каждому весу исходного вектора соответствует бинарный столбец, означающий его квантизованное значение. Каждый входной вектор (строка матрицы X) также может быть представлен в бинарном виде $B^x \in \{-1, 1\}^{K_x \times n}$ при помощи отдельных параметров квантизации для входов

слоя. Следует сразу отметить, что количество бит K_w на кодирование весов может не совпадать с количеством K_x, используемым для кодирования входных значений.

В таком случае, результат скалярного произведения входного и весового вектора в квантизованных вычислениях будет расчитываться по формуле (1.7).

$$\sum_{k=1}^{n} x_k w_k \approx \sum_{k=1}^{n} \sum_{i=1}^{K^x} (b_{ik}^x a_i^x) \sum_{j=1}^{K^w} (b_{jk}^w a_j^w) = \sum_{i=1}^{K^x} \sum_{j=1}^{K^w} a_i^x a_j^w (\sum_{k=1}^{n} b_{ik}^x b_{jk}^w), \quad (1.7)$$

где b_{ik}^x и b_{ik}^w - бит, находящийся на i-ой строке в k-ом столбце бинарной матрицы B^x и B^w соответственно.

Обозначим i-ую строку бинарной матрицы B^x и B^w как b_i^x и b_i^w соответственно, а под оператором $\odot$ будем понимать скалярное произведение двух бинарных векторов. В таком случае формулу (1.7) можно переписать в виде (1.8).

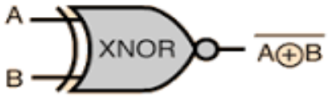
$$\sum_{k=1}^{n} x_k w_k \approx \sum_{i=1}^{K^x} \sum_{j=1}^{K^w} a_i^x a_j^w (b_i^x \odot b_j^w), \quad (1.8)$$

Для обоснования выигрыша в скорости, который дает предложенная формула для расчета, следует подробнее рассмотреть бинарное скалярное произведение.

Скалярное произведение — сумма произведений соответствующих элементов двух векторов. В это же время бинарное множество $\{-1, 1\}$ замкнуто относительно операции умножения, поэтому произведение двух бинарных значений остается бинарным значением, а значит может быть представлено одним битом. Возможны следующие варианты произведения 2-ух битов:

1. $-1(0) \cdot -1(0) = 1(1)$
2. $-1(0) \cdot 1(1) = -1(0)$
3. $1(1) \cdot -1(0) = -1(0)$
4. $1(1) \cdot 1(1) = 1(1)$

Такой схеме соответствует бинарная операция $xnor = \neg xor$, логическая схема которой представлена на рис. 1.9.



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	1

Рис. 1.9 — Xnor

Тогда в результате применения побитового xnor к двум бинарным векторам получается вектор, подсчет скалярного произведения из которого сводится к разности количества единиц в нем и количества нулей (-1 в принятом соглашении). Для подсчета количества единиц в бинарной записи числа в наборах инструкций современных архитектур ЭВМ существуют соответствующаие инструкции, такие как POPCNT в SSE4.2 Intel [12] и VCNT в Arm NEON [13]. Решив простую систему уравнений (1.9), можно получить формулу (1.10).

$$\begin{cases} \odot = pos - neg \\ size = pos + neg, \end{cases} \tag{1.9}$$

$$\odot = 2pos - size \tag{1.10}$$

где pos — количество 1 в строке, neg — количество -1 в строке, $size$ - размер строки в битах.

Следовательно, результат бинарного скалярного произведения может быть посчитан с использованием (1.10) при помощи описанных выше побитовых операций. Формула (1.11) — бинарное скалярное произведение 2-ух строк матриц закодированных весов.

$$b_i^x \odot b_j^w = 2popcnt(\neg(b_i^x \oplus b_j^w)) - size \tag{1.11}$$

Каждая из операций в (1.11) поддержана в современных архитектурах на уровне железа, что делает эту часть скалярного произведения крайне эффективной с вычислительной точки зрения.

Эффективность предложенного метода для расчета выходов нейронов ожидается не только при выполнении нейронных сетей на классических архитекутах процессоров, но и при реализации полносвязного слоя с использованием мемристорных кроссбаров. Реализация бинарного скалярного произведения на схеме с использованием мемристорных элементов приводится в [9]. Поскольку мемристор имеет ограниченное количество уровней проводимости, матричные произведения на кроссбарах необходимо также выполнять в дискретизованном формате, что предполагает использование низкобитовой квантизации. Исследование потери точности при уменьшении количества уровней дискретизации приведено в работе [5], где обосновывается возможность использования кроссбаров для задач глубокого обучения.

Поскольку ЭВМ способны оперировать типами данных ограниченного размера, то при условии хранения строк бинарных матриц в виде массива 32-битных целых чисел uint32, псевдокод быстрого бинарного скалярного произведения в c++20 представлен в листинге 1.1.

Листинг 1.1 — Пример модели с LQFullyConnected

```
int32_t bin_dot(uint32_t* b_a, uint32_t* b_w, uint32_t neurons){
  uint32_t size = (neurons + 31) >> 5; // ceil divide to 32
  int32_t pos = neurons - (size << 5);
  for(uint32_t i = 0; i < size; ++i){
    pos += std::popcount(~(b_a[i] ^ b_w[i])); // popcnt(xnor)
  }
  return (pos << 1) - neurons;
}
```

Опираясь на полученную арифметику оптимального перемножения и на соображения кэш-дружелюбности при вычислениях, принято решение хранить в квантизованных слоях веса в бинаризованном формате в виде 3-х мерного целочисленного тензора с размерами: $(o, k, ceil(n/32))$, где o — размер выходного вектора, k — количество бит для кодирования каждого веса, n — размер входного вектора. Деление с округлением вверх в определении размера 3-го измерения предусматривает случаи,

когда размер входного вектора n не делится на 32 целочисленно. Базис-векторы для весов следует хранить в виде вещественного тензора размера (o, k) для более эффективного доступа к ним во время вычисления аппроксимированного скалярного произведения по выведенной выше формуле, а базис-вектор для кодирования входных значений может быть представлен в виде одномерного вещественного тензора, длина которого совпадает с количеством бит на кодирование одного входного значения.

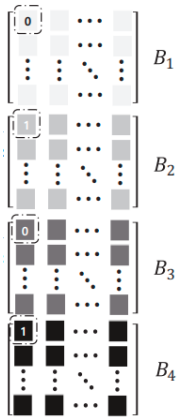


Рис. 1.10 — Бинарные веса для 4 нейронов

Как нетрудно догадаться, на вход слоя поступает матрица в неквантизованном виде, потому что входной вектор слоя напрямую зависит от тех вещественных данных, что приходят на вход всей нейронной сети, что делает невозможным квантизацию входов до начала исполнения модели и вынуждает проводить преобразование в бинарное представление во время работы нейронной сети каждый раз перед вычислением выходов нейронов слоя. Это является одним из существенных недостатков алгоритма, но даже несмотря на это этот казус, алгоритм остается крайне эффективным, поскольку время, которое тратится на квантизацию можно асимптотически оценить как O(n k), поскольку для каждого входного значения определяется набор бит, которые оптимально его кодирует с помощью бинарного поиска по массиву порогов квантизации, что делается за $O(log(2^K)) = O(K)$, после чего квантизованное значение записывается побитово в буффер, на что также требуется $O(K)$ по времени.

Ранее уже было упомянуто, что задача квантизации — как можно сильнее снизить потери от перевода значений из области действительных чисел в целые числа. Для достижения этой цели базис-векторы обучаются таким образом, чтобы минимизировать ошибку между реальным вещественным значением и соответствующим его квантизованному значению действительным числом. Пусть $Q(x)$ - функция-квантизатор, которая возвращает для действительного числа x его оптимальный уровень квантизации. Тогда, если описать решаемую описываемым инструментом проблему более формально, то нам необходимо найти оптимальный квантизатор, который минимизирует математическое ожидание квадратической ошибки для параметра, распределённого с плотностью $p(x)$.

$$Q^* = \arg\min_Q \int_{-\infty}^{\infty} p(x)(Q(x) - x)^2 dx \tag{1.12}$$

Решение этой проблемы аналитически почти всегда невозможно из-за того, что закон распределения генеральной совокупности параметров при квантизации не известен, а поиск решения этой проблемы через линейный поиск по параметрам квантизации может занять слишком много времени. Поэтому в этом методе предлагается использовать эвристический алгоритм QEM(Quantization Error Minimization). Этот алгоритм, описание которого приводится в [1], работает с исходным вещественным вектором входных данных x и итеративно пытается улучшить значение базис-вектора a, основываясь на наилучшем значении этого вектора для промежуточной кодировки B.

Алгоритм QEM:

1. Инициализировать a случайными значениями.
2. Повторять от 0 до T:
    1. Найти оптимальную промежуточную кодировку B^* для x на основе a.
    2. Найти оптимальный $a^* = \arg\min_a \|B^*a - x\|^2$ и обновить a полученным значением a^*.

Поиск оптимальной матрицы в пункте 2.1 определяется процессом квантизации с использованием порогов, который был описан выше. Что касается пункта 2.2, то при справедливости условий Маркова-Гаусса (что на практике, к сожалению, не часто выполняется) оптимальный базис-вектор может быть найден при помощи метода наименьших квадратов по формуле (1.13), что и предлагается в [1].

$$a^* = (BB^T)^{-1}Bx \tag{1.13}$$

Так как операция обращения матрицы является вычислительно емкой и трудно гарантировать, что определитель матрицы ковариаций BB^T не будет равен или близок к 0 в процессе обучения, поэтому в данной работе предлагается использовать итеративный градиентный спуск с L_2 регуляризацией, на каждой итерации которого значения будут обновляться следующим образом по формуле (1.14).

$$a = a - \frac{lr}{n}\sum_{i=1}^{n}[((b_i, a) - x_i)b_i + \lambda a], \tag{1.14}$$

где b_i - i-ый столбец матрицы B, $$ - темп обучения, который задается в качестве гиперпараметра и регулирует скорость сходимости алгоритма к оптимуму. Регуляризация с параметром λ используется чтобы штрафовать большие значения параметров, потому что веса и входы слоёв, как правило, имеют небольшие значения.

Применение данного алгоритма для обучения базис-вектора параметров предлагается в данной работе как основа инструмента для post-training квантизации. Этот метод обучения работает не только в предлагаемом методе работы с весами и активациями слоя. Он может определять оптимальные значения параметров для других методов, например BiQGEMM из [3], в котором предлагается иной способ расчета результата генерального матричного произведения без перевода входных значений в целочисленный вид, но совпадает формат хранения и интерпретация весов слоя.

2 Практическая часть

При выборе базового фреймворка для реализации функциональности, описанной в теоретической части следовало опираться на следующие факторы:

— Открытый и понятный исходный код с лицензией, позволяющей добавление собственного программного кода.

— Наличие собственного формата in-memory представления нейронной сети.

— Наличие формата и инструментов для сериализации/десериализации глубоких моделей.

— Наличие легковесного интерпретатора для нейронных моделей, который можно расширить добавлением собственных вычислительных ядер для интерпретации новых нейронных слоёв.

— Наличие инструментов, которые позволяют переводить глубокие модели во внутренний формат из других популярных форматов, в которых могла быть обучена исходная нейронная сеть.

Среди проектов, соответствующих выделенным критериям, ярко выделяются такие, как Tensorflow от компании Google и ONE от компании Samsung. Выбор был сделан в пользу последнего, поскольку он активно развивается под лицензией Apache 2.0, а его интерпретатор показывает лучшие результаты в проведенных бенчмарках на ЭВМ с жесткими ограничениями в ресурсах за счет лучшей работы с памятью.

Разработка велась в локальном форке исходного проекта на GitHub в отдельной ветке (URL: https://github.com/Bronnikoff/ONE/tree/lq_nets), поскольку это удобный способ управлять добавляемым кодом при определенной независимости от параллельных изменений, вносимых в базовый проект со стороны его разработчиков.

2.1 Поддержка слоя в графовых представлениях

Квантизованный слой имеет отличное от оригинального слоя поведение при вычислениях, количество параметров и формат хранения

данных. Это означает, что для поддержки его работы и для удобства обработки графа при написании квантизатора, необходимо поддержать новую операцию LQFullyConnected во всех имеющихся в ONE графовых представлениях, которые используются для работы с моделью от этапа её считывания из файла до этапа расчета результатов ее работы. Префикс LQ (Learnable Quantization), содержащийся в названии операции, был добавлен к названию чтобы указать на то, что она является результатом перевода полносвязанного (FullyConnected) слоя в низкобитовый формат при помощи обучаемых параметров квантизации.

Такими графовыми представлениями, через которые проходит модель до непосредственного запуска, являются:

— Представление circle для сериализации и десериализации графовых моделей с жёсткого диска.

— In-memory представление luci, которое работает со слоями как с вершинами, соединенными друг с другом посредством указателей.

— Вычислительный граф, который оперирует тензорами, связанными друг с другом вычислительными ядрами операций.

Реализацию было решено начать с описания схемы хранения квантизованного слоя в долгосрочной памяти вместе со схемой его сериализации и десериализации (способа записи структуры данных на жесткий диск и извлечения её из него). Благодаря тому, что принятый в Samsung формат хранения моделей circle основан на известной технологии flatbuffers, этот этап потребовал лишь описания в файле circle-schema.fbs параметров, которые уникальны и присущи только добавляемому слою.

Листинг 2.1 — Структура слоя в схеме

```
table LQFullyConnectedOptions {
        hidden_size: int;
        fused_activation_function:ActivationFunctionType;
}
```

Этими параметрами являются тип функции активации, которая применяется к выходу сумматора каждого нейрона и число hidden_size, равное размеру входного вектора, поскольку его необходимо знать в случа-

ях, когда размер входных данных не делится на цело на 32, для корректного расчета бинарных скалярных произведений при работе операции.

Помимо описания уникальных параметров LQFullyConnected, требуется декларировать наличие этого нового оператора и присвоить ему уникальный номер в схеме формата circle, который был выбран случайным образом.

Остальная работа по сохранению и извлечению модели производится самим flatbuffers. При помощи компилятора flatc, предоставляемого компанией Google, на основе схемы, которая была расширена параметрами нового слоя, генерируется хедер-файл для языка C++. Этот файл schema_generated.h реализовывает алгоритм перевода набора байт из дискового пространства в описанную в C++ структуру хранения этой операции в памяти при работе программы, извлекая в том числе и информацию о связях с соседними операциями в графе модели. Стоит отметить, что flatc поддерживает несколько языков программирования, таких как Java или JS, а не только C++, что позволяет работать с circle не только в C++. Применение этого свойства будет показано далее.

После того, как новая операция была добавлена в circle, следует перейти к поддержке LQFullyConnected в основном промежуточном представлении (Intermideate Representation) под названием luci. В отличие от 2-ух других представлений, оно не создано для решения какой-то конкретной задачи, такой как высокопроизводительный запуск сети или реализация алгоритмов работы с сырыми данными из файла. Поэтому в нём модели представлены в виде классических графов, вершинами которых являются операции, а связи между соседними слоями прописаны явно, через указатели языка C++. Универсальный вид графа делает его удобным для применения различных оптимизаций с подграфами модели. Также из него удобно перевести модель в любой другой формат, который заточен под решение конкретной задачи.

Следует выделить 4 основных части в формате luci:

1. Описание представления каждой операции в формате. (luci/lang)
2. Функциональность для импорта модели из circle. (luci/import)
3. Функциональность для экспорта модели из circle. (luci/export)

4. Алгоритмы оптимизации, применяемые к графу. (luci/pass)

На уровне описания структуры представления LQFullyConnected в luci, отраженной на листинге 2.2, явно объявляется, что новая операция должна иметь 5 входных вершин: входная вершина, из которой поступают входные данные, базис-вектор для квантизации входных данных, набор базис-векторов для квантизации весов слоя, бинаризованные веса и вектор смещений.

Листинг 2.2 — Структура слоя в luci

```
class CircleLQFullyConnected final : public FixedArityNode<5,
    CircleNodeImpl<CircleOpcode::LQ_FULLY_CONNECTED>>,
    public CircleNodeMixin<CircleNodeTrait::FusedActFunc>,
    public CircleNodeMixin<CircleNodeTrait::Bias>
{
public:
  loco::Node *input(void) const { return at(0)->node(); }
  void input(loco::Node *node) { at(0)->node(node); }

  loco::Node *input_scales(void) const { return at(1)->node(); }
  void input_scales(loco::Node *node) { at(1)->node(node); }

  loco::Node *weights_scales(void) const { return at(2)->node(); }
  void weights_scales(loco::Node *node) { at(2)->node(node); }

  loco::Node *weights_binary(void) const { return at(3)->node(); }
  void weights_binary(loco::Node *node) { at(3)->node(node); }

  loco::Node *bias(void) const override { return at(4)->node(); }
  void bias(loco::Node *node) override { at(4)->node(node); }

public:
  int32_t weights_hidden_size(void) const {
    return _weights_hidden_size;
  }
  void weights_hidden_size(int32_t weights_hidden_size)
  {
    _weights_hidden_size = weights_hidden_size;
  }

private:
  int32_t _weights_hidden_size = 0;
};
```

Стоит отметить, что luci явно не требует, чтобы все из входных вершин существовали. Например, bias() может принимать значение nullprt.

Основная функциональность для перевода моделей из luci в circle и обратно уже была реализована внутри проекта профессионалами из Samsung, что потребовало тривиальных изменений в исходном коде для поддержки импорта и экспорта.

Оптимизации для подграфов, содержащих LQFullyConnected в данной работе не рассматриваются, поэтому единственные алгоритмы, которые потребовалось реализовать в luci/pass — это определение типа данных (TypeInferencePass) и алгоритм определения размерности выходов слоя на основе размеров входных вершин (ShapeInferencePass).

Опциональной частью этой работы стала модификация существующего в проекте кода, отвечающего за отображение свойств операции в текстовом формате. Помимо этого, было полезно иметь наглядную визуализацию сетей, в графе которых присутствует добавляемая в этой работе операция. Для этого был модифицирован исходный код другого популярного проекта для визуализации множества различных нейросетевых форматов netron, который работает на языке JS. Как было отмечено ранее, flatc умеет переводить схему не только в C++, но и в JS. Сгенерированная с помощью flatc в JS схема работы с форматом circle была добавлена в этот инструмент, а остальная функциональность для работы с этой схемой уже присутствовала в «исходниках» визуализатора, поскольку он уже поддерживает формат tflite, который также основывается на flatbuffers.

При помощи описания простой нейронной сети с одним входом, одной LQFullyConnected операцией и одним выходом, при использовании имеющегося в проекте ONE инструмента circlechef для перевода описаний моделей в реальные физические модели, был получена первая сеть, визуализация которой netron-ом дала результат на рис. 2.1.

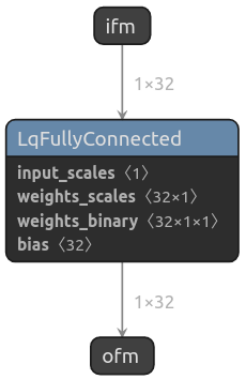


Рис. 2.1 — Пример модели с LQFullyConnected

Возможность анализа моделей посредством визуализатора не входит в данную работу как необходимая ее часть, однако позволяет увеличить удобство работы пользователя с сетями в формате circle и ускорить дальнейший процесс разработки и отладки кода.

Последним из выделенных графовых представлений является форма вычислительного графа, которая спроектирована таким образом, чтобы минимизировать издержки при работе с слоями и достичь максимальной производительности. Вершинами в этом представлении являются тензоры, связанные между собой вычислительными ядрами. Для того, чтобы поддержать операцию на этом уровне, необходимо определить класс для операции, инкапсулирующий в себе разные вычислительные ядра в зависимости от типа поступающих данных и описывающий логику взаимодействия тензоров между собой.

Такой класс помимо ядер и указателей на входные и выходные тензоры содержит метод configure(), который необходимо переопределить для:

— Проверки, что со входными тензорами можно провести запуск ядра. Именно в этом месте накладываются ограничения на параметры квантизованного слоя, описанные и обоснованные в теоретической части этой работы.

— Определения размерности данных, с которыми необходимо оперировать вычислительному ядру, и аллоцирования выходного тензора.

— Создания и инициализация вспомогательных вычислительных структур. Здесь создается объект класса LQBinarizer, инкапсулирующий в себе квантизацию входных данных вместе буфером для его квантизованных значений.

2.2 Вычислительное ядро интерпретатора

Вычислительное ядро интерпретатора вычисляет выходы операции LQFullyConnnected на основе входных данных, находящихся в тензоре input. Поскольку поступающие входные данные являются вещественными, а не квантизованными, перед расчетом выхода отдельного нейрона для одного входного вектора по формуле (1.8) необходимо произвести квантизацию входного вектора, с целью получения закодированного int32 буфера, который будет использоваться при расчетах. В связи с этим следует рассмотреть 2 основные части, которые требуется реализовать для поддержки корректной работы интерпретатора с моделями, содержащими LQFullyConnected:

— Класс LQBinarizer, позволяющий проводить кодирование входных вещественных векторов и предоставляющий доступ к бинарным данным.

— Функция evalFloat, производящая расчет выходов по формуле (1.8), используя LQBinarizer внутри себя.

2.2.1 Кодировщик LQBinarizer

Для квантизации входных значений «на лету» появилась необходимость ввести вспомогательный класс LQBinarizer, объявление которого показано на листинге 2.3.

Листинг 2.3 — Класс кодировщика

```
class LQBinarizer
{
  using Level = std::pair<float, int32_t>;

public:
  LQBinarizer(int32_t data_vec_size, const Tensor *data_scales);

  const int32_t *data() { return data_binary.get(); }
```

```
  void quantize_and_pack(const float *data_vector);

private:
  int32_t bin_search_encoding(float value);

private:
  int32_t data_float_size;
  int32_t data_bin_size;
  int32_t encode_bits;
  std::unique_ptr<int32_t[]> data_binary;

  int32_t levels_count;
  std::unique_ptr<Level[]> quantization_levels;
  std::unique_ptr<float[]> quantization_thresholds;
};
```

В конструкторе класса определяются пороги и уровни квантизации в отсортироанном порядке. Они необходимы для определения оптимальной кодировки для каждого из вещественных чисел. Помимо этого, при создании объекта класса происходит аллокация буфера бинарных данных и определяется чило $data\_bin\_size = ceil(data\_float\_size/32)$, которое равно длине одной строки выделенного буфера. Было решено сделать выделение памяти за один раз в конструкторе, связав её освобождение с концом жизни объекта кодировщика через умные указатели, чтобы снизить издержки на ее перевыделение при расчете квантизованных значений во время работы вычислительного ядра.

Метод quanitize_and_pack тривиально проходит по переданному ему вектору данных, который должен быть фиксированной длины data_float_size, и заполняет в соответствующую строку буфера data_binary биты числа, полученного вызовом приватного метода bin_search_encoding.

Метод bin_search_encoding является модифицированной версией классического бинарного поиска по массиву порогов t. Для любого поступаемого на вход значения $x : t_l < x \leqslant t_{l+1}$, он возвращает число l. После чего, используя число l в качестве индекса в массиве упорядоченных порогов, определяется оптимальная кодировка для числа x.

2.2.2 Расчет выходов слоя

В методе evalFloat по формуле (1.8), реализованной кодом с листинга 2.4, рассчитываются выходы нейронов для каждого из входных векторов.

Листинг 2.4 — Расчет выходов слоя

```
// calculate over output float values
for (int32_t bi = 0; bi < input_encode_bits; ++bi)
{
  // input computation data
  float inp_s = input_scales_data[bi];
  int32_t *inp_bin_line = &input_binary_data[bi * real_size];

  for (int32_t bw = 0; bw < weights_encode_bits; ++bw)
  {
    int32_t w_offset = calcOffset(weights_scales_shape, out_idx, bw);
    float w_s = weight_scales_data[w_offset];
    int32_t *w_bin_line = &weights_binary_data[w_offset * real_size];

    // add to total
    output_total += inp_s * w_s * bin_dot(inp_bin_line, w_bin_line);
  }
}
output_data[calcOffset(out_shape, batch, out_idx)] = output_total;
```

Расчёт бинарного скалярного произведения выполняется лямбда-выражением с листинга 2.5 при помощи побитовых операций и внутренней функции компилятора gcc __builtin_popcount, которая понижается при компиляции с опцией -mpopcnt в соответствующую ассемблерную инструкцию POPCNT из набора команд SSE4.2 для процессоров Intel.

Листинг 2.5 — Бинарное скалярное произведение

```
auto bin_dot = [hidden_size, real_size]
(const int32_t *data_1, const int32_t *data_2) {
  int32_t positives = hidden_size - (real_size << 5); // hs - 32*rs
  for (int32_t i = 0; i < real_size; ++i)
  {
    positives += __builtin_popcount(~(data_1[i] ^ data_2[i]));
  }
  return (positives << 1) - hidden_size; // 2*positives - neurons_count
};
```

Недостатком такой реализации является возможное отсутствие поддержки __builtin_popcount в компиляторах, отличных от gcc. Эта про-

блема должна быть решена с переходом проекта ONE со стандарта C++14 на C++20, в котором присутствует функция std::popcount.

2.3 Квантизатор

Для того, чтобы использовать нейронные сети с квантизованныи полносвязанными нейронными слоями для решения реальных задач, поддержки LQFullyConnected в интерпретаторе и графовых представлениях недостаточно. Данная работа не имеет смысла без инструмента, который переводил бы уже обученные нейронные сети в квантизованный формат с подбором оптимальных параметров и бинарных весов для каждого из FullyConnected слоёв. Поэтому в рамках этой работы вводится и описывается подобный инструмент, получивший название lquantizer. Его работа основана на следующих этапах:

1. Считывание исходной нейронной сети fp_graph и создание ее копии lq_graph - результата работы инструмента, которая будет изменяться в процессе работы lquantizer.
2. Проход по соответствующим вершинам 2-ух сетей с целью:
   — Замены в результирующей сети всех FullyConnected слоев на LQFullyConnected.
   — Определение входных вершин соответствующих слоев обоих сетей для дальнейшего использования при обучении.
3. Обучение параметров квантизации для каждой из вершин LQFullyConnected.
   — Обучение весовых параметров в lq_node на основе соответствующего массива весов из fp_node.
   — Обучение квантизаторов lq_node по входным значениям fp_node с целью минимизации ошибки с входными значениями fp_node.
   — Дообучение квантизаторов lq_node по входным значениям lq_node с целью минимизации ошибки с входными значениями fp_node.

Особый интерес среди указанных выше шагов представляет основной этап программы: обучение параметров.

2.3.1 Поиск оптимальных значений

Для поиска оптимальных параметров квантизации как для весов, так и для входов слоя, был создан класс QEM(Quantization Error Minimization), реализующий в себе итеративное обучение по предложенному в теоретической части алгоритму. В нем используется расширенный класс LQBinarizer из интерпретатора. В этот класс добавлен метод gradient_descent_scales, который принимает массив данных, с которыми требуется минимизировать ошибку, и ищет оптимальные значения для базис-вектора по формуле (1.14) на основе текущей матрицы бинарных кодировок. Стоит напомнить, что LQBinarizer обладает методом quantze_and_pack, который заполняет внутренний буфер кодировками, являющимися оптимальными для текущего базис-вектора. Сам метод QEM итеративно вызывает методы gradient_descent_scales и quantize_and_pack указанное число раз и заботится об упорядоченности базис-вектора.

2.3.2 Обучение весовых параметров

Для того, чтобы обучить весовые параметры каждого слоя, необходимо пройти по соответствующим вершинам lq_graph и fp_graph и для каждого из нейронов в них вызвать QEM::fit() для поиска оптимальных значений параметров квантизации. Поскольку для весов следует хранить не только параметры квантизации, но и бинарные данные, следует закодировать веса из fp_node полученными параметрами квантизации, используя объект класса LQBinarizer, что делает метод QEM::fill_binary().

2.3.3 Обучение и дообучение входных параметров

2.3.3.1 Сохранение входных значений

Поскольку ONE является высокопроизводительным фреймворком для запуска сетей, но не для их обучения, добавить в существующую архитектуру возможность сохранения входных значений для каждого из квантизуемых слоев с целью использования их для поиска оптимального распределения уровней квантизации, стало непростой задачей.

luci_interpreter позволяет реализовать и передать ему соответствующий обработчик Observer, который будет вызываться интерпретатором после завершения каждого из ядер. Обработчик может получать указатель на вершину соответствующей операции в luci графе и на тензор с выходными значениями, полученный в результате исполнения ядра.

Поэтому для того, чтобы корректировать базис-векторы LQFullyConnected вершин, было принято решение реализовать InputSavingObserver, который сохраняет в std::unordered_map выходные значения для каждой операции, которая была поставлена в соответствие в качестве входа на этапе 2 работы lquantizer. Именно этот словарь с значениями будет использоваться для корректировки параметров квантизации.

2.3.3.2 Корректировка базис-векторов

Корректировка параметров квантизации слоев из lq_graph происходит в 2 этапа:

1. Создание и запуск интерпретатора с прикрепленным к нему объектом класса InputSavingObserver.
2. Проход по 2-ум графам, где для каждой операции LQFullyConnected вызывается QEM алгоритм для обновления базис-векторов на основе полученных из словаря «наблюдателя» буфера входных значений.

На первом этапе заполняется репрезентативный массив данных, выборочное распределение которого будет использоваться для корректировки базис-векторов.

Второй этап несколько отличается для этапа обучения и дообучения. Если в первом случае используются входные значения из fp_graph для минимизация ошибки квантизации со значениями из fp_graph, то при дообучении используются входные значения из lq_graph для минимизации ошибки квантизации со значениями из fp_graph, что позволяет подобрать параметры LQFullyConnected для компенсации ошибок, порождаемых предшествующими рассматриваемой вершине графа слоями.

2.4 Анализ результатов

Для оценки того, какие преимущества дает квантизация, следует взять нейронную сеть, содержащую один или несколько полносвязанных нейронных слоев, и преобразовать ее в новый формат для сравнения результатов работы квантизованных сетей.

2.4.1 Получение нейронной сети

2.4.1.1 Обученная модель

Нейронная сеть для сравнения показателей была обучена самостоятельно на датасете MNIST, содержащем в себе рукописные изображения вместе с правильными метками к ним. Поскольку глубокие сети отлично справляются с задачами классификации, особенно на таких датасетах, как выбранный для анализа, в качестве модели была выбрана простая сеть с 2 полносвязными слоями: скрытый с функцией активации Tanh и выходной с Softmax. Для обучения была выбрана библиотека Keras. Полученную архитектуру сети можно увидеть на рис. 2.2.

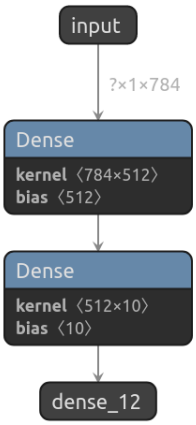


Рис. 2.2 — Keras сеть

2.4.1.2 Circle модель

Поскольку добавленный инструмент работает только с circle моделями, необходимо сконвертировать обученную модель в необходимый фор-

мат. Для этого можно воспользоваться уже имеющимся в ONE инструментом tf2circleV2. На рис. 2.3 показан граф этой модели.

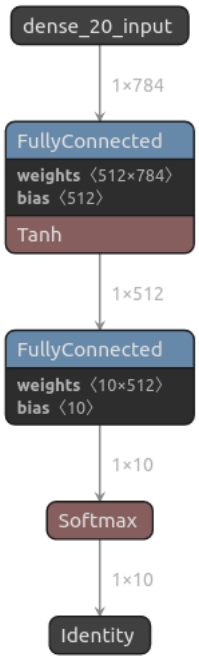


Рис. 2.3 — Circle сеть

2.4.1.3 Квантизованная модель

Для обучения параметров квантизации следует воспользоваться добавленным в этой работе инструментом, однако перед этим необходимо подготовить выборку, которая будет прогоняться через сеть для получения распределения входных параметров. lquantizer позволяет работать с данными в формате hdf5, поэтому перед этим обучающий и валидационный датасеты были преобразованы в этот формат.

Воспользоваться новым инструментом возможно через bash терминал linux при помощи команды с листинга 2.6.

Листинг 2.6 — Пример запуска квантизатора

```
./lquantizer --input_model model.circle  --output_model answer.circle \
             --input_data train.hdf5 --encode_bits 3
```

После некоторого ожидания lquantizer создает новый файл с квантизованной моделью, структура которой отражена на рис. 2.4.

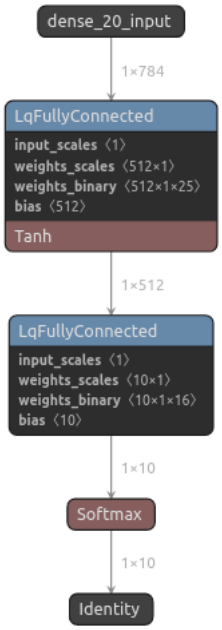


Рис. 2.4 — Квантизованная сеть

2.4.2 Результаты

При сравнении моделей основными метриками являются объем памяти, занимаемый сетью, средняя скорость её работы и точность на валидационной выборке, на которой не обучалась ни одна из них.

2.4.2.1 Оценка занимаемой памяти

Для того, чтобы оценить объем памяти, занимаемый моделью, достаточно посмотреть на вес файла, в котором она хранится. Результаты представлены в таблице 2.1.

Таблица 2.1 — Занимаемая память исходной и квантизованных моделей

-	fp32	int3	int2	int1
Объём	2.1 Mb	218.3 Kb	147.2 Kb	76.2 Kb

2.4.2.2 Оценка качества обучения

Чтобы оценить работу алгоритма обучения, следует рассмотреть насколько удачно обучаются уровни квантизации для параметров нейронных слоёв. Гистограмма значений весов и уровни квантизации для случайного нейрона в первом слое рассматриваемой модели отображены на рис. 2.5.

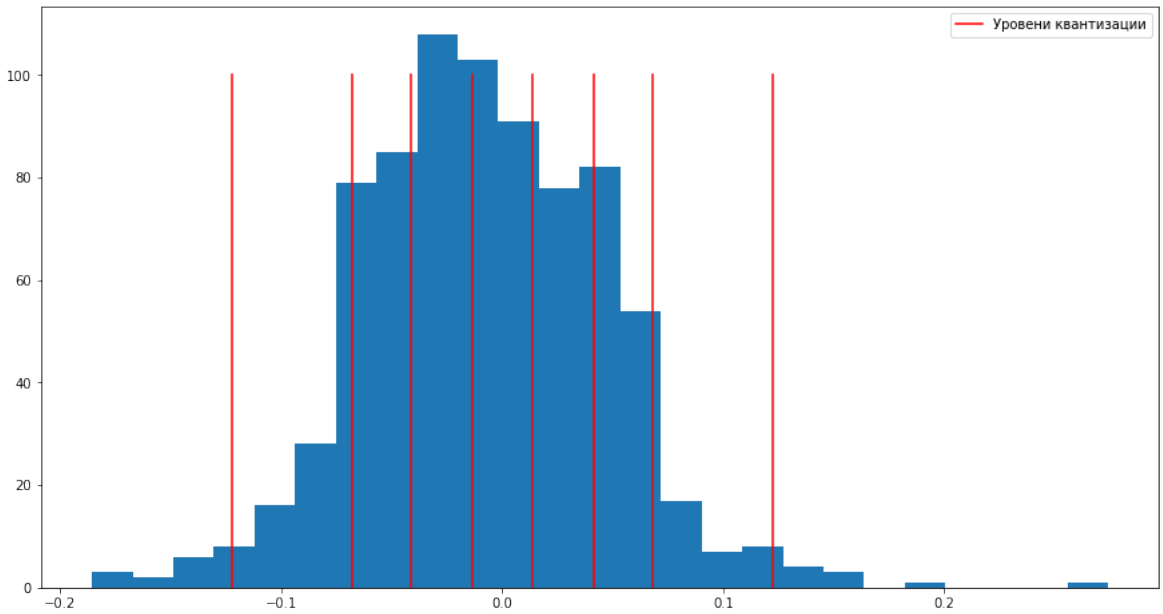


Рис. 2.5 — Распределение весов и уровней квантизации в нейроне

Стоит отметить, что уровни квантизации плотнее расположены к тем значениям весов, которые встречаются в нейроне наиболее часто.

Поскольку датасет, на котором обучалась модель, является сбалансированным, для оценки точности модели можно воспользоваться метрикой Accuracy по формуле (2.1), где $True$ означает количество правильных ответов сети, а $Count$ - количество записей в датасете.

$$Accuracy = \frac{True}{Count} \tag{2.1}$$

Для того, чтобы получать ответы оцениваемых нейронных сетей на валидационной выборке, была написана несложная программа, которая вызывает в себе luci_interpreter на данных из hdf5 файла и замеряет время работы каждого вызова. После чего, при помощи скрипта на языке

Python подсчитывается значение метрики Accuracy. Результаты проведенных замеров приведены в таблице 2.2.

Таблица 2.2 — Показатели точности исходной и квантизованных моделей

-	fp32	int3	int2	int1
Accuracy	97.19 %	95.59 %	88.86 %	75.11 %

2.4.2.3 Оценка производительности

Оценка производительности производится путем сравнения среднего времени работы нейронных сетей на наборе входных данных. Для этого в программе, реализованной для оценки точности, присутствует опция замера времени работы сети для каждого из входных векторов.

Характеристики ЭВМ, на котором производился запуск:

— Операционная система: Ubuntu 18.04.5 LTS

— Количество ОЗУ: 16Гб

— Процессор: Intel Core i7-8550U CPU @ 1.80GHz × 8 (имеется поддержка SSE 4.2)

— Версия компилятора: gcc 7.5.0

Чтобы оценить насколько возрастает скорость работы квантизованных сетей при использовании процессорной инструкции POPCNT из набора SSE 4.2, были проведены замеры при стандартной сборке luci_interpreter и с опцией компилятора -msse4.2. Результаты замеров отображены в таблице 2.3.

Таблица 2.3 — Производительность исходной и квантизованных моделей

-	fp32	int3	int2	int1
x86	**584.829 мкс**	**510.210 мкс**	**256.711 мкс**	**82.0352 мкс**
SSE 4.2	**583.931 мкс**	**152.743 мкс**	**84.7845 мкс**	**44.7829 мкс**

ЗАКЛЮЧЕНИЕ

В результате выполнения ВКР были достигнуты следующие цели:

— Изучены существующие алгоритмы квантизации и проанализированы их преимущества и недостатки.

— Выбран фреймворк, в рамках которого была реализована предлагаемая в этой работе функциональность.

— Подобрана схема хранения и использования квантизованных значений и параметров.

— Добавлен нейронный слой с квантизованными параметрами.

— Реализовано вычислительное ядро для работы слоя.

— Написан инструмент квантизации предобученных сетей.

— Проанализированы показатели полученных моделей.

Реализованное в данной работе программное обеспечение может стать решением проблемы ограниченности ресурсов на микроконтроллерах и прочих недорогих устройствах для тех разработчиков, чьи продукты тесно связаны с нейронными сетями.

Созданный инструмент планируется расширить возможностью квантизации сверточных слоев с соответствующей поддержкой в интерпретаторе и графовых представлениях фреймворка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, Gang Hua. (2018). LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 365-382

2. Raghuraman Krishnamoorthi. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv:1806.08342.

3. Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. (2020). BiqGEMM: matrix multiplication with lookup table for binary-coding-based quantized DNN's. arXiv:2005.09904.

4. Prateeth Nayak, David Zhang, Sek Chai. (2019). Bit Efficient Quantization for Deep Neural Networks arXiv:1910.04877.

5. А. Ю. Морозов, Д. Л. Ревизников, К. К. Абгарян. (2019). Вопросы реализации нейросетевых алгоритмов на мемристорных кроссбарах. Известия высших учебных заведений. Материалы электронной техники. 2019. Т. 22, № 4. С. 272—278.

6. Shujian Yu, Kristoffer Wickstrøm, Robert Jenssen, Jose C. Principe. (2020). Understanding Convolutional Neural Networks with Information Theory: An Initial Exploration. arXiv:1804.06537

7. Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, Mattan Erez. DeLTA: GPU Performance Model for Deep Learning Applications with In-depth Memory System Traffic Analysis. arXiv:1904.01691.

8. Rahul Dey, Fathi M. Salem. (2017). Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks. arXiv:1701.05923.

9. M.S. Tarkov, M.I. Osipov. (2016). The memristor crossbar-based WTA neural network. Bull. Nov. Comp. Center, Comp. Science, 39 (2016), 69–75.

10. Ralf C. Staudemeyer, Eric Rothstein Morris. (2019). Understanding LSTM: a tutorial into Long Short-Term Memory Recurrent Neural Networks. arXiv:1909.09586.

11. Ф.М. Гафаров, А.Ф. Галимьянов. (2018). Искусственные нейронные сети и их приложения. Издательство Казанского университета, 2018.

12. Intel. (2007). Intel SSE4 Programming Reference. 2006-2007 Intel Corporation.

13. Andrey Kamaev. (2012). Arm NEON SIMD. URL: http://www.itlab.unn.ru/file.php?id=731