|  |
| --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ высшего образования |
| **«Национальный исследовательский ядерный университет «МИФИ»**  **(НИЯУ МИФИ)** |

**ОТЧЕТ О ПРОХОЖДЕНИИ**

**УЧЕБНОЙ ПРАКТИКИ (ПОЛУЧЕНИЕ ПЕРВИЧНЫХ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ И НАВЫКОВ)**

**3 семестр**

|  |  |
| --- | --- |
| Тема: Разработка алгоритма квантования весовых коэффициентов нейронной сети | |
| Практикант | **Тоичкин Дмитрий Владимирович** |
| Группа | **М18-502** |
| Направление подготовки | **09.04.01 ИВТ** |
| Место выполнения практики | **НИЯУ МИФИ** |
| Руководитель практики от НИЯУ МИФИ | **Заева Маргарита Анатольевна,**  **к.т.н., доцент НИЯУ МИФИ,** |
| Руководитель практики от предприятия |  |

Москва, 2020 г.

Оглавление

[Введение 3](#_Toc30686581)

[1 Теоретическая часть 5](#_Toc30686582)

[1.1 Модель сверточной нейронной сети 5](#_Toc30686583)

[1.1.1 Сверточный слой 5](#_Toc30686584)

[1.1.2 Полносвязный слой 6](#_Toc30686585)

[1.1.3 Функция активации 7](#_Toc30686586)

[1.2 Описание схемы квантования весовых коэффициентов 8](#_Toc30686587)

[1.3 Вычисление линейной функции над квантованными аргументами 9](#_Toc30686588)

[1.4 Статистическая оценка погрешности линейной функции 10](#_Toc30686589)

[2 Практическая часть 14](#_Toc30686590)

[2.1 Описание программы для проведения эксперимента 14](#_Toc30686591)

[2.2 Анализ результатов эксперимента 15](#_Toc30686592)

[Заключение 18](#_Toc30686593)

[Список использованных источников 19](#_Toc30686594)

[Приложение 1. Исходный код программы эксперимента 20](#_Toc30686595)

План ВКР:

1. Проблема – структура современных нейронных сетей обладают избыточностью как по отношению к длине слова, так и по числу операций.
2. Цель: исследовать возможность уменьшения избыточности сети за счет квантования коэффициентов и усечения слоев
3. Обзор существующих методов. Поиск проблем в методах  
   The Effects of Quantization on Multilayer Neural Networks – представлена оценка квантования коэффициентов с учетом того, что коэффициенты распределены равномерно, функция активации – сигмоида. Мое предложение – рассмотреть случай нормального распределения коэффициентов, рассмотреть активацию ReLU как более популярную в наст. время. Веса – константа + ошибка, вход – сумма норм/експ распр и ошибки.  
   PRUNING CONVOLUTIONAL NEURAL NETWORKS FOR RESOURCE EFFICIENT INFERENCE – описать критерии удаления операций  
   INCREMENTAL NETWORK QUANTIZATION: TOWARDS LOSSLESS CNNS WITH LOW-PRECISION WEIGHTS – описать прием дихотомии, квантования до степени 2  
   DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING – привести пример оптимизируемых уровней квантования, кода Хаффмана для сжатия сети (коэффициенты восьмибитные зашиты в АЛУ, в сети хранятся только индексы)
4. Теор часть по определению уровня ошибки от квантования (доработка практики)
5. Теор часть по разработке критерия удаления лишних операций в сети
6. Теор часть по описанию общей методики оптимизации сети
7. Экспериментальная часть:
   1. Анализ ошибки из-за квантования без дообуч, сравнение с теорией
   2. Анализ ошибки из-за усечения без дообуч, сравнение с теорией
   3. Анализ accuracy drop, сокращение числа FLOPS, IOPS, mem в следствие применения методики

Экспериментальная часть:

* + 1. Кастомная CNN для цифара из практики
    2. Резнет на imagenet, переписанная в квантуемых классах pytorch. Статистика: выходы свертки, активаций
    3. Код для сбора данных в tensorboard
    4. Резнет, переписанная в классах с квантованным обучением
    5. Алгоритм по обнулению слоев согласно критерию

# Введение

Очередным рубежом на пути развития алгоритмов машинного обучения в полный рост стоит задача переноса вычислений в мобильные устройства, снижение зависимости клиентов от мощных удаленных вычислительных серверов.

Согласно данным, предоставленным компаниями Intel, NVidia, разрабатывающими собственные устройства для ускорения работы нейронных сетей, вычисления в формате int8 (integer 8-bit – целое число длиной 8 бит) позволяют увеличить производительность и уменьшить потребление памяти в сравнении с вычислениями в формате float32 (floating point 32-bit, число с плавающей запятой одиночной точности). Основной проблемой вычислений в формате int8 являются снижение точности модели, необходимость дополнительной корректировки модели, алгоритмов. В таблице 1 представлены данные компании Intel, полученные при измерении точности, скорости работы и объема занимаемой памяти квантованных нейронных сетей [1].

В исследовании было выполнено квантование нейронных сетей различных топологий. Практика показывает, что невозможно применить классический алгоритм квантования к нейронной сети и получить «из коробки» хороший результат. Поэтому, с целью сохранения высокой распознавательной способности, параметры нейронной сети лишь частично преобразовывались в формат int8, часть из них оставались в float32. Это позволило добиться заметной оптимизации вычислительных ресурсов при сохранении качества: падение метрики качества не превышало 1%.

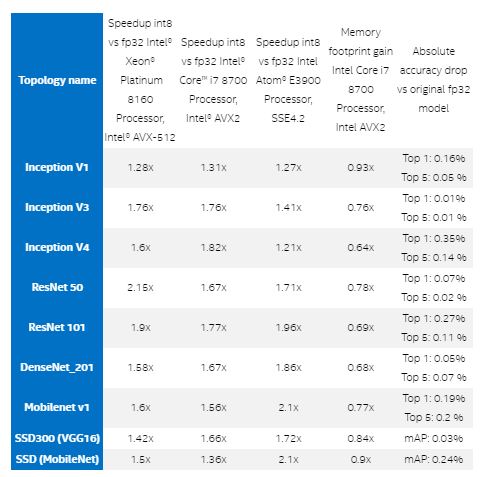


Таблица 1. Результаты тестов квантованных нейронных сетей

Целью данной работы является разработка метода оценки погрешностей, вносимых квантованием коэффициентов и поиск рационального компромисса между сокращением требуемых для нейронной сети вычислительных ресурсов и сохранением ее точности.

# Теоретическая часть

## Модель сверточной нейронной сети

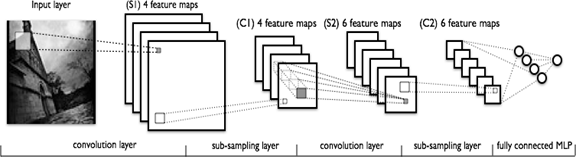
Объектом исследования в данной работе является сверточная нейронная сеть, состоящая из линейных операций свертки, матричного умножения и линейной функции активации. Каждая операция будет рассмотрена отдельно. На рисунке 1 схематично представлена сверточная нейросеть, представляющая собой каскад операций над входным массивом чисел.   


Рисунок 1. Топология сверточной нейронной сети

На рисунке 2 находится визуальное представление работы нейронной сети на примере обработки изображений. Нейронная сеть поэтапно формирует при помощи фильтров отдельные массивы признаков, которые в последствии классифицируются набором перцептронов.

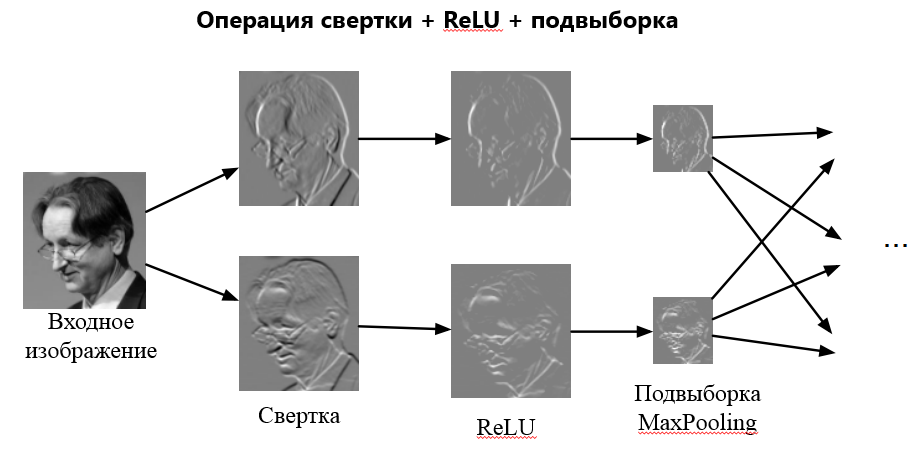
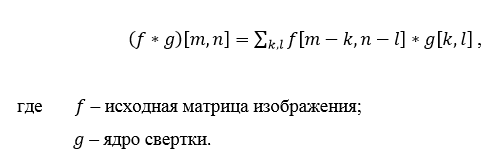


Рисунок 2. Иллюстрация работы сверточной нейронной сети

### Сверточный слой

Первая базовая линейная операция нейронной сети – двумерная свертка. Сверточный слой представляет из себя набор двумерных фильтров, которые применяются к входному n-канальному изображению. Количество каскадов фильтров и количество фильтров в каскаде определяется требованиями к задаче. Исходя из опыта и задачи необходимо находить компромисс между точностью и производительностью.

Сверточный слой работает по следующей формуле:



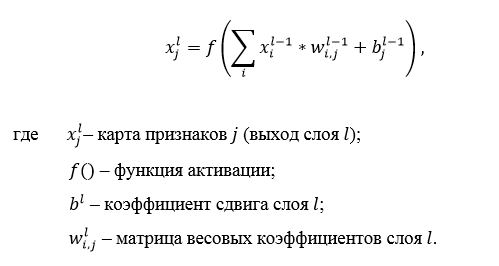
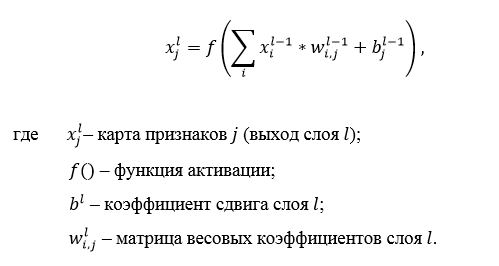
Каждый сверточный слой имеет набор параметров:

1. Число фильтров
2. Размерность фильтра
3. Шаг фильтра
4. Правило работы с отступами.

Число фильтров и их размерность объединяют в тройку чисел в следующем порядке (число фильтров) х (высота фильтра) х (ширина фильтра), например, 8х3х3. Это означает что в сверточном слое присутствуют 8 фильтров размерностью 3х3 и вектор из 8-и коэффициентов смещения. Число каналов выходного изображения равно числу фильтров. Весовыми коэффициентами слоя называется множество всех коэффициентов всех фильтров и всех коэффициентов смещения. Для фильтра 8х3х3 число коэффициентов будет составлять 8\*3\*3 + 8 = 80.

Задание шага определяет децимацию результирующего массива. При шаге равном двум, теряется 50% входных данных, при шаге равном трем – 66%.

### Полносвязный слой

Сверточные слои выделяют признаки в изображении, а полносвязные слои – вычисляют вероятности принадлежности изображения с данным набором признаков к тому или иному множеству. С математической точки зрения, полносвязные слои реализуют операцию матричного произведения со смещением и функцией активации:  
   


Полносвязный слой имеет только один параметр – число нейронов выходного слоя. Весовыми коэффициентами полносвязного слоя называются коэффициенты матрицы, определяющей связи между входными и выходными нейронами, и вектор смещений, соотвествующий числу выходных элементов. Например, если на вход полносвязного поступают 32 нейрона, а на выходе формируются 16 нейронов, то число коэффициентов будет составлять 32\*16 + 16 = 528.

### Функция активации

Функция активации – это нелинейная функция одного аргумента, которая применяется к каждому элементу массива.

Во второй половине 20-го века была доказана Универсальная теорема аппроксимации, утверждающая, что искусственная нейронная сеть прямой связи с одним скрытым слоем может аппроксимировать любую непрерывную функцию многих переменных с любой точностью. Условиями являются: достаточное количество нейронов скрытого слоя, удачный подбор параметров, применение сигмоидальной функции активации.

Заслуживает внимания тот факт, что без применения нелинейной функции активации невозможно аппроксимировать нелинейную функцию. В настоящий момент были сконструированы целые семейства различных функций активации с разнообразными свойствами.

В данной работе в качестве функции активации в скрытых и выходном слоях применяется функция Rectified Linear Unit (ReLU, выпрямитель). График и формула функции представлены на рисунке 3. Эта функция выбрана потому, что, с одной стороны, она обладает линейной составляющей в положительной полуплоскости, что упрощает ее математическую модель, а с другой стороны, не смотря на свою простоту, применение этой функции позволяет аппроксимировать нелинейные зависимости.

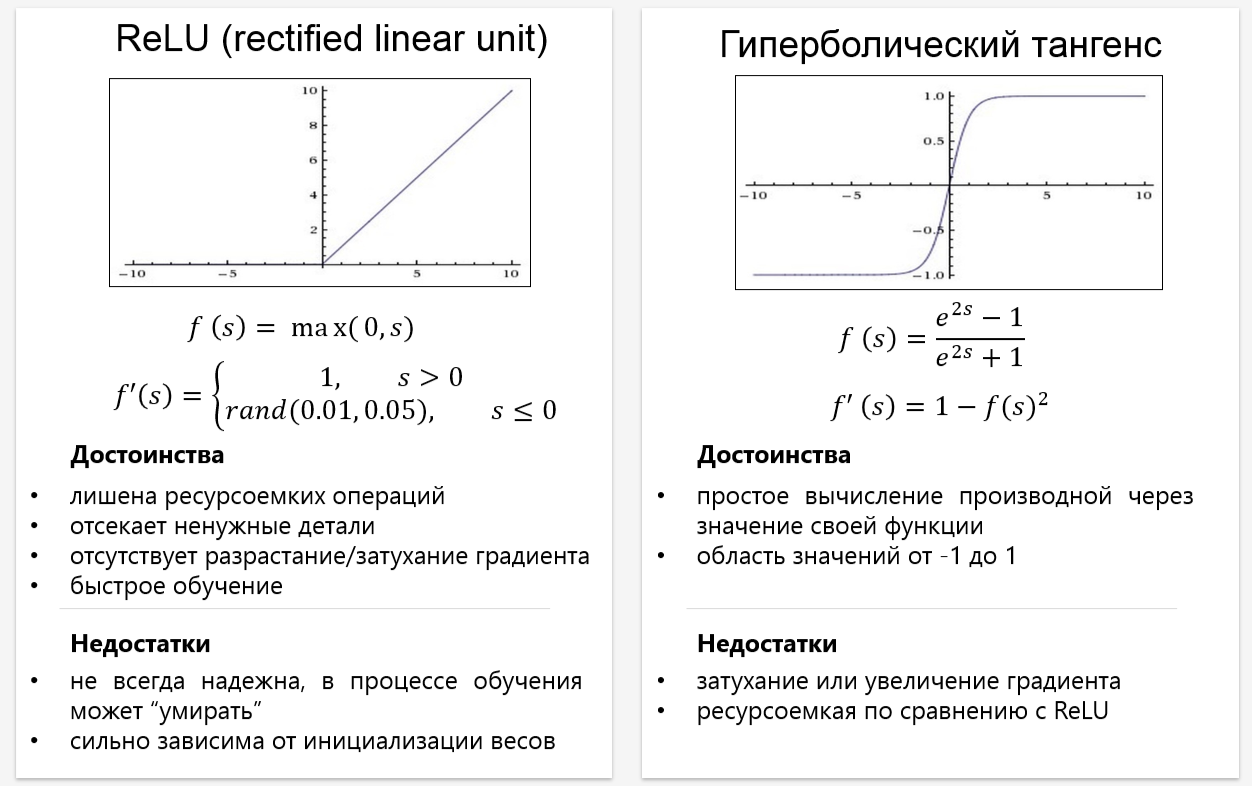


Рисунок 3. График функции активации ReLU

## Описание схемы квантования весовых коэффициентов

При квантовании нейронной сети на данный момент используется равномерные симметричный и ассиметричный методы квантования, состоящие из следующих этапов [2]:

* + - 1. Вычисляется шаг равномерный шаг квантования h для b-разрядных двоичных чисел:

где – максимальное и минимальное значения в массиве чисел

b – разрядность квантованных чисел.

* + - 1. Зная шаг квантования, вычисляется смещение нуля при ассиметричном квантовании:

Где – наибольшее квантованное число (127 для int8)

* + - 1. Квантованное и восстановленное значения вычисляются по следующим формулам:

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |

Схема квантования проиллюстрирована на рисунке 4.

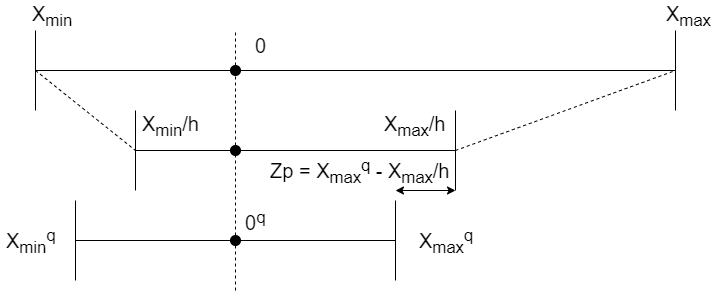


Рисунок 4 Схема квантования массива чисел

Для оценки погрешности вычислений, вносимой в результате квантования, было принято решение использовать показатель отношения «Сигнал/шум квантования» (Signal to Noise Ratio, SQNR):

Где – математическое ожидание квадрата полезного сигнала

– дисперсия шума

В качестве полезного сигнала выступают результаты промежуточных операций свертки и матричного произведения в исходной нейросети, в качестве ошибки – разность результатов промежуточных операций между исходной и квантованной нейросетями.

## Вычисление линейной функции над квантованными аргументами

Базовой операцией в нейронной сети, распознающей изображения, является линейная операция скалярного произведения со смещением. Поэтому целесообразно в первую очередь проанализировать влияние операции квантования на этот вид функций. Найдем точное значение скалярного произведения векторов выразив их через квантованные векторы:

Где – скалярное произведение векторов

– смещение, скаляр

– истинные значения компонент вектора, выраженные через его квантованные значения, согласно формулы (1).

Можно заметить, что точное значение операции скалярного произведения со смещением совпадает с его квантованным вариантом с точностью до постоянного коэффициента и дополнительного смещения . Точное значение скалярного произведения необходимо вновь квантовать для дальнейших вычислений:

|  |  |
| --- | --- |
|  | (3) |

Таким образом, для перехода к операциям над квантованными векторами необходимо предварительно вычислить масштабный коэффициент и дополнительное смещение

## Статистическая оценка погрешности линейной функции

После того, как был рассмотрен алгоритм вычисления скалярного произведения над квантованными числами, необходимо изучить погрешность, возникающую при вычислениях. Как уже было сказано ранее, в качестве меры погрешности вычислений будет использоваться SQNR:

|  |  |
| --- | --- |
|  | (4) |

Вывод статистической оценки SQNR основывается на следующих предположениях:

1. Равномерное квантование с округлением с шагом h вносит равномерный шум :

Где – коэффициент, определяющий размах значений в массиве относительно стандартного отклонения,

b – разрядность квантования.

1. Операция свертки с точки зрения теории вероятностей эквивалентна скалярному произведению реализаций случайных величин, имеющих определенные математическое ожидание и дисперсию
2. Все случайные величины независимы

Получить оценку SQNR для вектора квантованных чисел можно путем простых подстановок:

|  |  |
| --- | --- |
|  | (5) |

Из (5) формулы можно сделать вывод, что нижняя оценка SQNR для массива обратно пропорциональна размаху значений в массиве. С целью увеличения нижней оценки SQNR целесообразно разделить массив на группы и квантовать каждую группу независимо от других, либо перед операцией квантования отфильтровать значения по порогу .

Перейдем непосредственно к выводу соотношения SQNR для операции скалярного произведения. Будем считать, что скалярное произведение двух независимых случайных величин, распределенных по нормальному закону, имеет вид:

Тогда ошибка квантования SQNR:

Оценим , используя определения скалярного произведения и математического ожидания:

|  |  |
| --- | --- |
|  | (6) |

В данном анализе сумма отброшена, т.к. в работе выполнялся поиск нижней границы SQNR.

Найдем дисперсию скалярного произведения случайных векторов, на которые были наложены векторы равномерного шума в результате квантования, используя формулу для дисперсии произведения независимых случайных величин:

Только часть этой дисперсии представляет собой ошибку в следствии квантования, а именно та часть, которая зависит от шага квантования. Вынесем общий множитель, который позволит получить искомое соотношение на следующем шаге:

|  |  |
| --- | --- |
|  | (7) |

В результате повторного квантования результата вносится дополнительный шум:

который нужно сложить с шумом выполнения линейной операции:

Внесем полученные промежуточные формулы в формулу для SQNR (4):

При приведении подобных можно выявить такое соотношение между SQNR входных данных и SQNR результата:

|  |  |
| --- | --- |
|  | (8) |

Важно отметить, что это соотношение приблизительное. Более того, теоретическая оценка погрешности этой формулы тоже затруднительна. Получается, что согласно этой оценке, шум каскада линейных операций есть среднее гармоническое шумов среди всех каскадов, а конечная точность результата зависит, в первую очередь, от наименее точной операции.

Подводя итоги, было установлено, что:

* + - 1. Нижняя оценка SQNR для массива обратно пропорциональна размаху значений в массиве
      2. С целью увеличения показателя SQNR целесообразно квантовать отдельные сверточные фильтры слоя независимо, либо фильтровать все значения, превышающие порог
      3. Шум квантования в каскаде линейных операций распространяется согласно среднему гармоническому, а точность результата будет ниже наименее точной операции.

# Практическая часть

## Описание программы для проведения эксперимента

Для проведения теоретических предположений была использована сверточная нейронная сеть, конфигурация которой представлена в таблице 2. Назначение параметров каждого слоя было указано в первом разделе, поэтому в таблице параметры представлены в сокращенном виде.

Таблица 2. Конфигурация исследуемой нейронной сети

|  |  |  |  |
| --- | --- | --- | --- |
| № | Операция | Параметры | Функция активации |
| 1 | 2D-Свертка | 8x3x3, шаг=2 | max(0,x) |
| 2 | 2D-Свертка | 32x3x3, шаг=1 | max(0,x) |
| 3 | 2D-Свертка | 64x3x3, шаг=2 | max(0,x) |
| 4 | 2D-Свертка | 128x3x3, шаг=1 | max(0,x) |
| 5 | Матричное произведение | 120 | max(0,x) |
| 6 | Матричное произведение | 84 | max(0,x) |
| 7 | Матричное произведение | 10 | - |

Эксперимент состоял из трех этапов: обучения нейронной сети, квантования и сравнительного прогона исходной и квантованной нейросетей.

В качестве математической библиотеки использовалась библиотека pytorch, предоставляющая широкие возможности для конфигурирования структуры и параметров нейронных сетей.

Схема эксперимента представлена на рисунке 5. На первом этапе выполнялось обучение задаче классификации цветных изображений размера 32х32 из набора данных CIFAR-10 по стандартному алгоритму обратного распространения ошибки. Для выполнения этапа (2) к классам библиотеки pytorch, отвечающим за выполнение операций 2D-свертки и матричного произведения, были добавлены функции сбора статистики и вычисления максимального и минимального значений результата для последующего квантования.

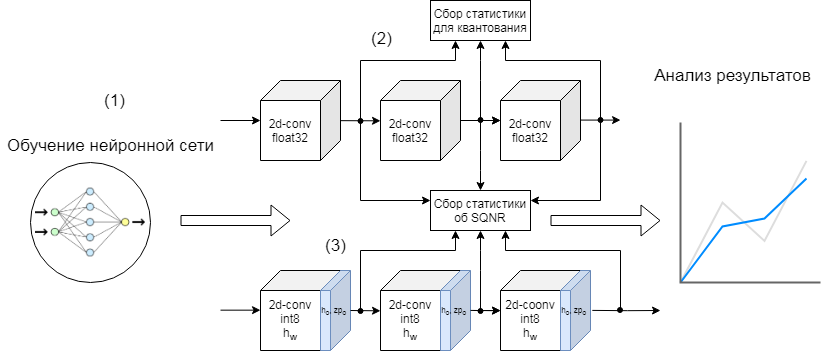


Рисунок 5. Схема вычислительного эксперимента

Для выполнения квантования был добавлен метод, который пересчитывал все весовые коэффициенты нейронной сети согласно формуле (1). В квантованной нейросети результаты всех линейных операций дополнительно квантовались согласно формуле (3):

Для проверки гипотез о зависимости SQNR результата линейной операции от SQNR ее аргументов, тестовые примеры параллельно прогонялись через исходную и квантованную нейросети. Во время прогона сохранялись результаты всех промежуточных вычислений, что позволило покаскадно сравнить работу исходной и квантованной нейросетей.

## Анализ результатов эксперимента

Эксперимент показал, что теоретическая оценка и практическое значение SQNR при квантовании весовых коэффициентов хорошо коррелируют, что видно на рисунке 6. На рисунке видно, что относительный разброс значений в массиве коррелирует с SQNR, а именно, чем выше квадрат относительного разброса, тем ниже SQNR.

Рисунок 6. График зависимости SQNR весовых коэффициентов от квадрата их разброса

Второй показатель, который был измерен во время эксперимента – SQNR квантованной линейной операции. Эксперимент показал, что теоретическое и практическое значения SQNR коррелируют, однако, в отдельных случаях происходят выбросы, совершенно не соответствующие гипотезе. На рисунке 7 представлен график, сравнивающий теоретическое и практическое значения SQNR. Из графика видно, что большинство точек находятся в области прямой y=x. Это говорит о том, что теоретически предсказанное значение SQNR и практическое также коррелируют. Существуют, однако, два примера, когда практический SQNR в 2-3 раза меньше или больше теоретического.

По результатам экспериментов было установлено, что оценка SQNR линейной операции подтверждалось в 71% случаев.

Рисунок 7. Сравнение теоретического и практического SQNR линейной операции

# Заключение

В теоретической части работы был выполнен обзор существующих промышленных технологий оптимизации вычислений в нейронных сетях, исследованы методы квантования нейронных сетей, сделаны оценки влияния квантования на точность работы нейронных сетей, выдвинуты предположения были выдвинуты два предположения о зависимости точности отдельной линейной операции и каскада линейных операций над большими массивами данных.

В практической части работы была разработана схема эксперимента, разработаны программные средства обеспечения эксперимента, проведены вычисления, анализ данных результатов эксперимента, выполнена проверка выдвинутых гипотез.

Первое предположение заключалось в том, что SQNR зависит от разброса значений. Судя по графику экспериментальных данных на рисунке 4, эта гипотеза хорошо коррелирует с данными.

Вторая гипотеза заключалась в том, что SQNR результата скалярного произведение есть среднее гармоническое SQNR его аргументов. Вторая гипотеза подтвердилась лишь в части случаев. Наблюдались несколько выбросов в данных, которые сильно выбивались из общей картины.

На основании этих выводов можно спроектировать более рациональный квантователь нейронных сетей.

# Список использованных источников

|  |  |
| --- | --- |
| [1] | Intel, «Introducing int8 quantization for fast CPU inference using OpenVINO,» 1 01 2019. [Электронный ресурс]. Режим доступа: https://www.intel.ai/introducing-int8-quantization-for-fast-cpu-inference-using-openvino. [Дата обращения: 10 01. 2020]. |
| [2] | Google, «TensorFlow Lite 8-bit quantization specification,» [Электронный ресурс]. Режим доступа:  https://www.tensorflow.org/lite/performance/quantization\_spec. |
| [3] | R. Yates, «Practical Considerations in Fixed-Point FIR Filter Implementations,» [Электронный ресурс]. Режим доступа: http://www.digitalsignallabs.com/fir.pdf. |
| [4] | R. Yates, «Fixed-Point Arithmetic: An Introduction,» [Электронный ресурс]. Режим доступа:  http://www.digitalsignallabs.com/fp.pdf. |
| [5] | В. Гмурман, Теория вероятностей и математическая статистика: учеб.пособие для вузов, Москва: Высшая школа, 2003. |
| [6] | D. L. Darryl, S. T. Sachin и A. V.Sreekanth, «Proceedings of the 33rd International Conference on Machine Learning,» в *Fixed Point Quantization of Deep Convolutional Networks*, New York, 2016. |

# Приложение 1. Исходный код программы эксперимента

# -\*- coding: utf-8 -\*-

"""quantized\_net\_arch\_v1.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/17ILraabRQJuhWMUP5ZV54SQirXlIkLsL

"""

# Commented out IPython magic to ensure Python compatibility.

# %matplotlib inline

import torch

import torchvision

import torchvision.transforms as transforms

from torch.quantization.fake\_quantize import default\_weight\_fake\_quant, FakeQuantize

from torch.quantization.observer import Observer

from copy import deepcopy

import numpy as np

class GroupMeanDispObserver(Observer):

def \_\_init\_\_(self):

self.group\_means = []

self.group\_disps = []

super(GroupMeanDispObserver, self).\_\_init\_\_(torch.qint8)

def forward(self, x\_orig):

x = x\_orig.detach() # avoid keeping autograd tape

x\_orig\_mean = x.mean()

x\_orig.std = x.std()

self.group\_means.append(x.mean())

self.group\_disps.append(x.std()\*\*2)

return x\_orig

def calculate\_qparams(self):

#as far as all tensors have same shape =>

# all measured group means and stds have same weight in

# global std and mean

mean\_disp = np.mean(self.group\_disps)

disp\_mean = np.std(self.group\_means)\*\*2

mean = np.mean(self.group\_means)

std = np.sqrt(mean\_disp + disp\_mean)

return mean, std

class SqnrObserver(Observer):

def \_\_init\_\_(self):

super(SqnrObserver, self).\_\_init\_\_(torch.qint8)

self.x\_true\_observer = GroupMeanDispObserver()

self.noise\_observer = GroupMeanDispObserver()

def forward(self, x\_orig, noise\_orig):

x = x\_orig.detach() # avoid keeping autograd tape

noise = noise\_orig.detach() # avoid keeping autograd tape

self.x\_true\_observer(x)

self.noise\_observer(noise)

return x\_orig

def calculate\_qparams(self):

x\_mean, x\_std = self.x\_true\_observer.calculate\_qparams()

noise\_mean, noise\_std = self.noise\_observer.calculate\_qparams()

sqnr = (x\_mean\*\*2 + x\_std\*\*2)/noise\_std\*\*2

return sqnr

"""The output of torchvision datasets are PILImage images of range [0, 1].

We transform them to Tensors of normalized range [-1, 1].

"""

transform = transforms.Compose(

[

transforms.ToTensor(),

transforms.Lambda(lambda x: x\*127)

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

shuffle=False, num\_workers=2)

classes = ('plane', 'car', 'bird', 'cat',

'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

images.max()

"""Let us show some of the training images, for fun."""

import matplotlib.pyplot as plt

import numpy as np

# functions to show an image

def imshow(img):

img = img / 2 + 0.5 # unnormalize

npimg = img.numpy()

plt.imshow(np.transpose(npimg, (1, 2, 0)))

# get some random training images

dataiter = iter(trainloader)

images, labels = dataiter.next()

# show images

imshow(torchvision.utils.make\_grid(images))

# print labels

print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

images.shape

"""2. Define a Convolution Neural Network

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Copy the neural network from the Neural Networks section before and modify it to

take 3-channel images (instead of 1-channel images as it was defined).

"""

import torch.nn as nn

import torch.nn.functional as F

import torch.nn.quantized as qnn

import torch.quantization

def fake\_quant\_cls():

quantizer = FakeQuantizeSqnr(dtype=torch.qint8, quant\_min=-128, quant\_max=127)

quantizer.zero\_point = 0

return quantizer

class FakeQuantizeSqnr(torch.quantization.FakeQuantize):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(FakeQuantizeSqnr, self).\_\_init\_\_(observer=torch.quantization.MinMaxObserver, \*\*kwargs)

self.sqnr\_observer = SqnrObserver()

def quant(self, X):

#print(self.observer\_enabled,

#self.fake\_quant\_enabled,

#self.qscheme)

if self.observer\_enabled:

self.observer(X.detach())

self.scale, self.zero\_point = self.calculate\_qparams()

if self.fake\_quant\_enabled:

if self.qscheme == torch.per\_channel\_symmetric or self.qscheme == torch.per\_channel\_affine:

#print(self.scale, self.zero\_point)

X = torch.fake\_quantize\_per\_channel\_affine(X, self.scale, self.zero\_point,

self.ch\_axis, self.quant\_min, self.quant\_max)

else:

X\_true = X

#self.zero\_point += self.quant\_min

assert self.quant\_max - torch.round(self.observer.max\_val/self.scale) == self.zero\_point

#print(self.scale, self.zero\_point)

#X = torch.fake\_quantize\_per\_tensor\_affine(X, float(self.scale),

# int(self.zero\_point), self.quant\_min,

# self.quant\_max)

X = torch.round(X/self.scale) + self.zero\_point

self.sqnr\_observer(X\_true, X\_true - (X-self.zero\_point)\*self.scale)

return X

def forward(self, x):

if self.fake\_quant\_enabled:

#print('fake')

x\_true = deepcopy(x.detach())

x\_quant = self.quant(x\_true)

noise = x\_true - x\_quant.detach()

self.sqnr\_observer(x\_true, noise)

return x\_quant

else:

return x

class QConv2d(nn.Conv2d):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(QConv2d, self).\_\_init\_\_(\*args, \*\*kwargs)

self.weight\_scale = 1

self.bias\_scale = 1

self.input\_scale = 1

self.input\_zeropoint = 0

def quantize(self, input\_scale, zero\_point):

weight = self.weight

quantized\_weight, weight\_scale = int8\_quant(weight)

self.weight = nn.Parameter(quantized\_weight)

bias\_scale = input\_scale\*weight\_scale

weight\_sum = self.weight.view(self.weight.shape[0],-1).sum(1)

bias = torch.round(self.bias/bias\_scale) - zero\_point\*weight\_sum

self.bias = nn.Parameter(bias)

self.weight\_scale = weight\_scale

self.bias\_scale = bias\_scale

self.input\_scale = input\_scale

self.input\_zeropoint = zero\_point

return

def compare(self, fp32Lyaer):

weight\_err = torch.abs(self.weight\*self.weight\_scale - fp32Lyaer.weight)

if err.max() > self.weight\_scale:

return False

return True

def forward(self, x):

eff\_scale = self.bias\_scale

return super(QConv2d, self).forward(x) \* eff\_scale

class QLinear(nn.Linear):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(QLinear, self).\_\_init\_\_(\*args, \*\*kwargs)

self.weight\_scale = 1

self.bias\_scale = 1

self.input\_scale = 1

self.input\_zeropoint = 0

def quantize(self, input\_scale, zero\_point):

weight = self.weight

quantized\_weight, weight\_scale = int8\_quant(weight)

self.weight = nn.Parameter(quantized\_weight)

bias\_scale = input\_scale\*weight\_scale

weight\_sum = self.weight.sum(1)

bias = torch.round(self.bias/bias\_scale) - zero\_point\*weight\_sum

self.bias = nn.Parameter(bias)

self.weight\_scale = weight\_scale

self.bias\_scale = bias\_scale

self.input\_scale = input\_scale

self.input\_zeropoint = zero\_point

return

def forward(self, x):

eff\_scale = self.bias\_scale

return super(QLinear, self).forward(x) \* eff\_scale

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.input\_quant = fake\_quant\_cls()

#self.conv1 = nn.Conv2d(3, 8, 3,stride=2)

self.qconv1 = QConv2d(3, 8, 3,stride=2)

self.conv1\_sqnr\_observer = FakeQuantizeSqnr()

self.conv1\_quant = fake\_quant\_cls()

#self.pool = nn.MaxPool2d(2, 1)

#self.conv2 = nn.Conv2d(8, 32, 3)

self.qconv2 = QConv2d(8, 32, 3)

self.conv2\_quant = fake\_quant\_cls()

#self.conv3 = nn.Conv2d(32, 64, 3, stride=2)

self.qconv3 = QConv2d(32, 64, 3, stride=2)

self.conv3\_quant = fake\_quant\_cls()

#self.conv4 = nn.Conv2d(64, 128, 3)

self.qconv4 = QConv2d(64, 128, 3)

self.conv4\_quant = fake\_quant\_cls()

#self.fc1 = nn.Linear(128 \* 3 \* 3, 120)

self.qfc1 = QLinear(128 \* 4 \* 4, 120)

self.fc1\_quant = fake\_quant\_cls()

#self.fc2 = nn.Linear(120, 84)

self.qfc2 = QLinear(120, 84)

self.fc2\_quant = fake\_quant\_cls()

#self.fc3 = nn.Linear(84, 10)

self.qfc3 = QLinear(84, 10)

self.fc3\_quant = fake\_quant\_cls()

self.quantizers = [self.input\_quant,

self.conv1\_quant,self.conv2\_quant,self.conv3\_quant,self.conv4\_quant,

self.fc1\_quant,self.fc2\_quant,self.fc3\_quant]

def disable\_quantizers(self):

#list(map(lambda x: x.disable(), self.quantizers))

for i in range(len(self.quantizers)):

self.quantizers[i] = self.quantizers[i].disable\_fake\_quant()

assert not self.quantizers[i].fake\_quant\_enabled

def forward(self, x):

self.trace\_dict = {}

x = self.input\_quant(x)

x = self.qconv1(x)

self.trace\_dict['qconv1'] = x

x = self.conv1\_quant(x)

if self.conv1\_quant.fake\_quant\_enabled:

x = torch.max(x,self.conv1\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu1'] = (x-self.conv1\_quant.zero\_point)\*self.conv1\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu1'] = x

x = self.qconv2(x)

self.trace\_dict['qconv2'] = x

x = self.conv2\_quant(x)

if self.conv2\_quant.fake\_quant\_enabled:

x = torch.max(x,self.conv2\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu2'] = (x-self.conv2\_quant.zero\_point)\*self.conv2\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu2'] = x

x = self.qconv3(x)

self.trace\_dict['qconv3'] = x

x = self.conv3\_quant(x)

if self.conv3\_quant.fake\_quant\_enabled:

x = torch.max(x,self.conv3\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu3'] = (x-self.conv3\_quant.zero\_point)\*self.conv3\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu3'] = x

x = self.qconv4(x)

self.trace\_dict['qconv4'] = x

x = self.conv4\_quant(x)

if self.conv4\_quant.fake\_quant\_enabled:

x = torch.max(x,self.conv4\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu4'] = (x-self.conv3\_quant.zero\_point)\*self.conv4\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu4'] = x

x = x.view(-1, 128\*4\*4)

x = self.qfc1(x)

self.trace\_dict['fc1'] = x

x = self.fc1\_quant(x)

if self.fc1\_quant.fake\_quant\_enabled:

x = torch.max(x,self.fc1\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu5'] = (x-self.fc1\_quant.zero\_point)\*self.fc1\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu5'] = x

#self.trace\_dict['relu5'] = x

x = self.qfc2(x)

self.trace\_dict['fc2'] = x

x = self.fc2\_quant(x)

if self.fc2\_quant.fake\_quant\_enabled:

x = torch.max(x,self.fc2\_quant.zero\_point[0].to(torch.float))

self.trace\_dict['relu6'] = (x-self.fc2\_quant.zero\_point)\*self.fc2\_quant.scale

else:

x = F.relu(x)

self.trace\_dict['relu6'] = x

x = self.qfc3(x)

self.trace\_dict['fc3'] = x

x = self.fc3\_quant(x)

#self.trace\_dict['relu7'] = x

return x

net2 = Net()

net2.disable\_quantizers()

net2\_quant = deepcopy(net2)

#

net2.load\_state\_dict(buf)

#net2\_quant.load\_state\_dict(buf\_quant)

new\_state\_dict = deepcopy(net2.state\_dict())

for key in new\_state\_dict.keys():

if 'qconv' in key or 'qfc' in key:

true\_name = key[1:]

new\_state\_dict[key] = new\_state\_dict[true\_name]

net2.load\_state\_dict(new\_state\_dict)

def quantize\_layer(input\_quant, layer):

input\_quant.enable\_fake\_quant()

test(net2\_quant, testloader)

input\_scale, input\_zeropoint = input\_quant.calculate\_qparams()

layer.quantize(input\_scale, input\_zeropoint)

test(net2\_quant, testloader)

net2.qconv2.bias/net2\_quant.qconv2.bias\_scale - net2\_quant.qconv2.input\_zeropoint\*net2\_quant.qconv2.weight.view(32,-1).sum(1)

buf\_quant = deepcopy(net2\_quant.state\_dict())

quantize\_layer(net2\_quant.input\_quant, net2\_quant.qconv1)

quantize\_layer(net2\_quant.conv1\_quant, net2\_quant.qconv2)

quantize\_layer(net2\_quant.conv2\_quant, net2\_quant.qconv3)

quantize\_layer(net2\_quant.conv3\_quant, net2\_quant.qconv4)

quantize\_layer(net2\_quant.conv4\_quant, net2\_quant.qfc1)

quantize\_layer(net2\_quant.fc1\_quant, net2\_quant.qfc2)

quantize\_layer(net2\_quant.fc2\_quant, net2\_quant.qfc3)

buf = deepcopy(net2.state\_dict())

def calculate\_qparams(self):

x\_mean, x\_std = self.x\_true\_observer.calculate\_qparams()

noise\_mean, noise\_std = self.noise\_observer.calculate\_qparams()

sqnr = (x\_mean\*\*2 + x\_std\*\*2)/noise\_std\*\*2

return sqnr

for layer in net2\_quant.children():

if 'weight\_scale' in dir(layer):

print(layer, layer.weight\_scale.detach().item())

elif 'sqnr\_observer' in dir(layer):

true\_mean, true\_std = layer.sqnr\_observer.x\_true\_observer.calculate\_qparams()

scale = layer.scale

if scale:

print((true\_mean\*\*2+true\_std\*\*2)/(scale\*\*2/12).item())

class Correlation:

def \_\_init\_\_(self, compute\_func=None):

self.compute\_func = compute\_func

self.true = []

self.pred = []

def compute\_true(self, \*args):

self.true.append(self.compute\_func(\*args))

def compute\_pred(self, \*args):

self.pred.append(self.compute\_func(\*args))

def observe\_true(self, true):

self.true.append(true)

def observe\_pred(self, pred):

self.pred.append(pred)

def observe(self, true, pred):

self.observe\_true(true)

self.observe\_pred(pred)

def mean\_true(self):

return np.mean(self.true)

def mean\_pred(self):

return np.mean(self.pred)

def corrcoef(self):

corrcoef = np.corrcoef(self.true, self.pred)

return corrcoef[0,1]

def least\_squares(self):

x = np.array(self.pred)

y = self.true

A = np.vstack((x, np.ones(len(x)))).T

m, c = np.linalg.lstsq(A, y, rcond=None)[0]

return m, c

from functools import reduce

class StatsAcumulator:

def \_\_init\_\_(self):

self.header\_fmt = '|{:15s}' \* 5 + '|'

self.line\_fmt = '|{:15s}' + '|{:15.3e}' \* 2 + '|'

self.corrline\_fmt = '|{:15s}' + '|{:15.3e}' \* 2 + '|{:15.2f}' + '|{:5.3f}x+{:7.3f}|'

self.reset\_stats()

def reset\_stats(self):

self.ErrorX\_observer = Correlation()

self.ErrorY\_observer = Correlation()

self.EX2\_observer = Correlation()

self.EY2\_observer = Correlation()

self.SqnrX\_observer = Correlation()

self.SqnrY\_observer = Correlation()

self.EConv\_observer = Correlation()

self.DConv\_observer = Correlation()

self.EConv2\_observer = Correlation()

self.ErrConv\_observer = Correlation()

self.SqnrConv\_observer = Correlation()

self.Relu\_observer = Correlation()

self.ErRelu\_observer = Correlation()

self.SqnrRelu\_observer = Correlation()

def per\_layer\_stats(self, X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale):

Y\_scale = Y\_pred.weight\_scale.item()

B\_true = Y.bias.detach().numpy()

B\_pred = Y\_pred.bias.detach().numpy() \* Y\_pred.bias\_scale.item()

Y\_pred = Y\_pred.weight.detach().numpy() \* Y\_scale

Y = Y.weight.detach().numpy()

X = X.detach().numpy()

X\_pred = X\_pred.detach().numpy()

M = reduce(lambda x,y: x\*y, Y.shape[1:], 1)

#X = X.flatten(1)

ErY\_true = (Y-Y\_pred).std()\*\*2

ErY\_pred = Y\_scale\*\*2/12

self.ErrorY\_observer.observe(ErY\_true, ErY\_pred)

ErX\_true = (X-X\_pred).std()\*\*2

ErX\_pred = X\_scale\*\*2/12

self.ErrorX\_observer.observe(ErX\_true, ErX\_pred)

EX2\_true = (X\*\*2).mean()

EX2\_pred = (X\_pred\*\*2).mean()

self.EX2\_observer.observe(EX2\_true, EX2\_pred)

EY2\_true = (Y\*\*2).mean()

EY2\_pred = (Y\_pred\*\*2).mean()

self.EY2\_observer.observe(EY2\_true, EY2\_pred)

SqnrX\_true = EX2\_true/ErX\_true

SqnrX\_pred = SqnrX\_true#EX2\_pred/ErX\_pred

self.SqnrX\_observer.observe(SqnrX\_true, SqnrX\_pred)

SqnrY\_true = EY2\_true/ErY\_true

SqnrY\_pred = EY2\_pred/ErY\_pred

self.SqnrY\_observer.observe(SqnrY\_true, SqnrY\_pred)

EY2 = (Y.mean()\*\*2 + Y.std()\*\*2)

EX2 = (X.mean()\*\*2 + X.std()\*\*2)

ErX = (X\_scale\*\*2/12)

ErY = (Y\_scale\*\*2/12)

result\_mean\_true = qconv\_true.mean()

#print(Y\_pred.ravel(1).shape)

result\_mean\_pred = (Y.reshape(Y.shape[0],-1).sum(1))\*X.mean() + B\_true.mean()

self.EConv\_observer.observe(result\_mean\_true, result\_mean\_pred.mean())

result\_d\_true = qconv\_true.std()\*\*2

mean\_std = result\_mean\_pred.std()\*\*2

std\_mean = (Y\_pred.reshape(Y\_pred.shape[0],-1)\*\*2).sum(1)\*X.std()\*\*2

result\_d\_pred = std\_mean.mean() + mean\_std#((Y.std()\*X.mean())\*\*2 + (Y.mean()\*X.std())\*\*2 + (Y.std()\*X.std())\*\*2)\*M

self.DConv\_observer.observe(result\_d\_true, result\_d\_pred)

result\_e2\_true = result\_mean\_true\*\*2 + result\_d\_true

result\_e2\_pred = result\_mean\_pred.mean()\*\*2 + result\_d\_pred

self.EConv2\_observer.observe(result\_e2\_true, result\_e2\_pred)

ErResult\_true = (qconv\_true - qconv\_pred).std()\*\*2

ErResult\_pred = (EX2\_pred\*ErY\_pred + EY2\_pred\*ErX\_pred + ErX\_pred\*ErY\_pred)\*M

self.ErrConv\_observer.observe(ErResult\_true, ErResult\_pred)

SqnrX = EX2\_pred/(X\_scale\*\*2/12)

SqnrY = EY2\_pred/(Y\_scale\*\*2/12)

SqnrResult\_true = result\_e2\_true / ErResult\_true

SqnrResult\_pred = 1 / (1 / SqnrY\_pred + 1 / SqnrX\_pred)

self.SqnrConv\_observer.observe(SqnrResult\_true, SqnrResult\_pred)

ErRelu = (relu\_true - relu\_pred).std()\*\*2

SqnrRelu\_true = (relu\_true\*\*2).mean() / ErRelu

SqnrRequant = result\_e2\_pred / (relu\_scale\*\*2/12)

SqnrRelu\_pred2 = 1 / (1 / SqnrY\_pred + 1 / SqnrX\_pred + 1 / SqnrRequant)

self.SqnrRelu\_observer.observe(SqnrRelu\_true, SqnrRelu\_pred2)

#print(SqnrRequant.item())

def print\_stats(self):

header\_fmt = self.header\_fmt

line\_fmt = self.line\_fmt

corrline\_fmt = self.corrline\_fmt

print(header\_fmt.format('Переменая', 'Точное значение', 'Оценка', 'Корреляция','kx + b'))

print(line\_fmt.format('Ошибка весов', self.ErrorY\_observer.mean\_true(),self.ErrorY\_observer.mean\_pred()))

print(line\_fmt.format('Ошибка входа', self.ErrorX\_observer.mean\_true(),self.ErrorX\_observer.mean\_pred()))

print(line\_fmt.format('Sqnr входа', self.SqnrX\_observer.mean\_true(),self.SqnrX\_observer.mean\_pred()))

print(line\_fmt.format('Sqnr весов', self.SqnrY\_observer.mean\_true(),self.SqnrY\_observer.mean\_pred()))

print(corrline\_fmt.format('EX2 входа', self.EX2\_observer.mean\_true(),self.EX2\_observer.mean\_pred(),

self.EX2\_observer.corrcoef(),

\*self.EX2\_observer.least\_squares() ))

print(corrline\_fmt.format('EY2 входа', self.EY2\_observer.mean\_true(),self.EY2\_observer.mean\_pred(),

self.EY2\_observer.corrcoef(),

\*self.EY2\_observer.least\_squares() ))

print(corrline\_fmt.format('Знач. выхода', self.EConv\_observer.mean\_true(),self.EConv\_observer.mean\_pred(),

self.EConv\_observer.corrcoef(),

\*self.EConv\_observer.least\_squares() ))

print(corrline\_fmt.format('DX выхода', self.DConv\_observer.mean\_true(),self.DConv\_observer.mean\_pred(),

self.DConv\_observer.corrcoef(),

\*self.DConv\_observer.least\_squares() ))

print(corrline\_fmt.format('EX2 выхода', self.EConv2\_observer.mean\_true(),self.EConv2\_observer.mean\_pred(),

self.EConv2\_observer.corrcoef(),

\*self.EConv2\_observer.least\_squares() ))

print(corrline\_fmt.format('Ошибка выхода', self.ErrConv\_observer.mean\_true(),self.ErrConv\_observer.mean\_pred(),

self.ErrConv\_observer.corrcoef(),

\*self.ErrConv\_observer.least\_squares() ))

print(corrline\_fmt.format('sqnr выхода', self.SqnrConv\_observer.mean\_true(),self.SqnrConv\_observer.mean\_pred(),

self.SqnrConv\_observer.corrcoef(),

\*self.SqnrConv\_observer.least\_squares() ))

print(corrline\_fmt.format('sqnr req', self.SqnrRelu\_observer.mean\_true(),self.SqnrRelu\_observer.mean\_pred(),

self.SqnrRelu\_observer.corrcoef(),

\*self.SqnrRelu\_observer.least\_squares() ))

i = 1

for layer in net2.children():

try:

weight = layer.weight

print(i)

i += 1

print(layer)

span = (weight.max() - weight.min())/ weight.std()

var = weight.mean()\*\*2 / weight.std()\*\*2

print('Размах = {:.2f} стандартных откл., 1/var^2 = {:.2f}'.format(span.item(), var.item()))

except AttributeError:

pass

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = images

X\_scale = net2\_quant.input\_quant.scale.item()

X\_pred = net2\_quant.input\_quant(images)\*X\_scale

Y = net2.qconv1

Y\_pred = net2\_quant.qconv1

qconv\_true = net2.trace\_dict['qconv1'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['qconv1'].detach().numpy()

relu\_true = net2.trace\_dict['relu1'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu1'].detach().numpy()

relu\_scale = net2\_quant.conv1\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu1']

X\_pred = net2\_quant.trace\_dict['relu1']

X\_scale = net2\_quant.conv1\_quant.scale.item()

Y = net2.qconv2

Y\_pred = net2\_quant.qconv2

qconv\_true = net2.trace\_dict['qconv2'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['qconv2'].detach().numpy()

relu\_true = net2.trace\_dict['relu2'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu2'].detach().numpy()

relu\_scale = net2\_quant.conv2\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu2']

X\_pred = net2\_quant.trace\_dict['relu2']

X\_scale = net2\_quant.conv2\_quant.scale.item()

Y = net2.qconv3

Y\_pred = net2\_quant.qconv3

qconv\_true = net2.trace\_dict['qconv3'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['qconv3'].detach().numpy()

relu\_true = net2.trace\_dict['relu3'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu3'].detach().numpy()

relu\_scale = net2\_quant.conv3\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

A = np.array(stats\_acum.SqnrRelu\_observer.true)

B = np.array(stats\_acum.SqnrRelu\_observer.pred)

N = len(A)

EA = A.mean()

DA = A.std()\*\*2

EB = B.mean()

DB = B.std()\*\*2

S2 = (DA\*N + DB\*N) / (2\*N - 2)

std = np.sqrt(S2\*2/N)

t = (EA - EB) / std

print(t)

(EA - EB)

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu3']

X\_pred = net2\_quant.trace\_dict['relu3']

X\_scale = net2\_quant.conv3\_quant.scale.item()

Y = net2.qconv4

Y\_pred = net2\_quant.qconv4

qconv\_true = net2.trace\_dict['qconv4'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['qconv4'].detach().numpy()

relu\_true = net2.trace\_dict['relu4'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu4'].detach().numpy()

relu\_scale = net2\_quant.conv4\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu4']

X\_pred = net2\_quant.trace\_dict['relu4']

X\_scale = net2\_quant.conv4\_quant.scale.item()

Y = net2.qfc1

Y\_pred = net2\_quant.qfc1

qconv\_true = net2.trace\_dict['fc1'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['fc1'].detach().numpy()

relu\_true = net2.trace\_dict['relu5'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu5'].detach().numpy()

relu\_scale = net2\_quant.fc1\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu5']

X\_pred = net2\_quant.trace\_dict['relu5']

X\_scale = net2\_quant.fc1\_quant.scale.item()

Y = net2.qfc2

Y\_pred = net2\_quant.qfc2

qconv\_true = net2.trace\_dict['fc2'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['fc2'].detach().numpy()

relu\_true = net2.trace\_dict['relu6'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['relu6'].detach().numpy()

relu\_scale = net2\_quant.fc2\_quant.scale.item()

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

stats\_acum = StatsAcumulator()

dataiter = iter(trainloader)

for i in range(3000):

images, \_ = dataiter.next()

images = images

net2(images)

net2\_quant(images)

X = net2.trace\_dict['relu6']

X\_pred = net2\_quant.trace\_dict['relu6']

X\_scale = net2\_quant.fc2\_quant.scale.item()

Y = net2.qfc3

Y\_pred = net2\_quant.qfc3

qconv\_true = net2.trace\_dict['fc3'].detach().numpy()

qconv\_pred = net2\_quant.trace\_dict['fc3'].detach().numpy()

relu\_true = net2.trace\_dict['fc3'].detach().numpy()

relu\_pred = net2\_quant.trace\_dict['fc3'].detach().numpy()

relu\_scale = 1

stats\_acum.per\_layer\_stats(X, X\_scale, X\_pred, Y, Y\_pred, qconv\_true, qconv\_pred, relu\_true, relu\_pred, relu\_scale)

stats\_acum.print\_stats()

"""3. Define a Loss function and optimizer

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Let's use a Classification Cross-Entropy loss and SGD with momentum.

"""

import torch.optim as optim

from tqdm import tqdm\_notebook as tqdm

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(net2.parameters(), lr=1e-4)

scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, 2, gamma=0.6, )

"""4. Train the network

^^^^^^^^^^^^^^^^^^^^

This is when things start to get interesting.

We simply have to loop over our data iterator, and feed the inputs to the

network and optimize.

"""

n = 200

quantizer\_enable\_period = 5

quantizer\_count = 0

for epoch in range(15): # loop over the dataset multiple times

#if epoch % quantizer\_enable\_period == 0 and quantizer\_count < len(net.quantizers):

# net.quantizers[quantizer\_count].enable()

# quantizer\_count += 1

running\_loss = 0.0

for q in net2.quantizers:

print(q.fake\_quant\_enabled)

for i, data in tqdm(enumerate(trainloader, 0)):

# get the inputs

inputs, labels = data

# zero the parameter gradients

optimizer.zero\_grad()

# forward + backward + optimize

outputs = net2(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % n == n-1: # print every 2000 mini-batches

print('[%d, %5d] loss: %.3f' %

(epoch + 1, i + 1, running\_loss / n))

running\_loss = 0.0

scheduler.step()

print('Finished Training')

"""5. Test the network on the test data

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

We have trained the network for 2 passes over the training dataset.

But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network

outputs, and checking it against the ground-truth. If the prediction is

correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

"""

def int8\_quant(array):

#low = array.mean() - clip\_coeff\*array.std()

#hi = array.mean() + clip\_coeff\*array.std()

#filt\_clipped = np.clip(array, low, hi)

abs\_max = torch.max( torch.abs(array))

step = 2\*abs\_max / 255

return torch.round(array/step), step

def test(net, testloader):

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (

100 \* correct / total))