Отчет по домашнему заданию по KWS

Никита Степанов

21 ноября 2021 г.

Baseline и streaming

Работу над домашним заданием я начал с того, что порефакторил и частично переписал код с семинарского ноутбука. Обучилась модель очень хорошо, качество было равно 1.6×10^{-5}, при том что я обучал модель всего лишь 20 эпох. График обучения можно посмотреть на картинке 1. Я думаю, что такой хороший результат отчасти связан с тем, что мне повезло

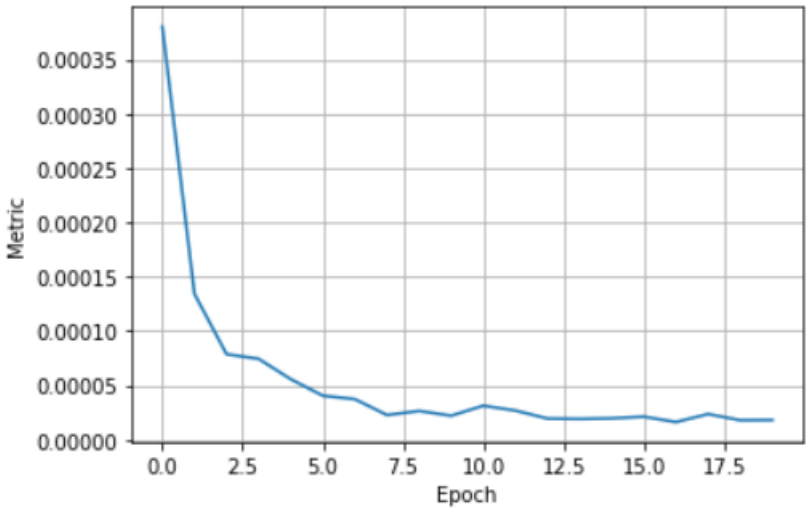

Рис. 1: Качество базовой модели в зависимости от эпохи

с разделением на train и val.

После этого я реализовал streaming CRNN, демонстрация которого находится в файле streaming_demo.ipynb. Я провел там следующий эксперимент: взял 10 аудио без ключевого слова и одно с ключевым, потом склеил все эти аудиозаписи, причем ключевое слово было посередине. После этого я последовательно подавал на вход спектрограмму стриминговой модели и сохранял вероятности, которые она выдает, по ним построил график. Параметр window length, который определяет кол-во векторов, сохраненных для attention, я выставил равным 20. И даже при таком небольшом значении этого параметра модель довольно хорошо справляется со своей задачей, разве что она довольно быстро забывает о ключевом слове, но это и неудивительно. На картинке 2 мы видим, что вероятности согласуются со здравым смыслом. Сначала они равны примерно 0.5, потому что на этом этапе очень мало информации, и модель не уверена в своем предсказании. Примерно в середине мы видим резкий скачок, а потом вероятности снова затухают, потому что модель начинает забывать, что было ранее.

Дистилляция

Когда я начал работать над оптимизацией модели, первое, с чем я решил разобраться — это дистилляция, потому что я уже знал, что это такое, и что это не должно занять много усилий. Я прочитал оригинальную статью, и взял оттуда некоторые параметры, например, температуру выбрал равной 4. Выбор этот я сделал более менее случайно, но все-таки он немного обоснован тем, что в экспериментах на MNIST-е для очень маленьких student-моделей лучше всего работали температуры в диапазоне от 2.5 до 4. Также я взял из статьи параметр α, т.е. коэффициент, на который умножается hard loss. В экспериментах в speech recognition авторы использовали $\alpha = \frac{1}{2}$, и я взял такое же значение. Таким образом

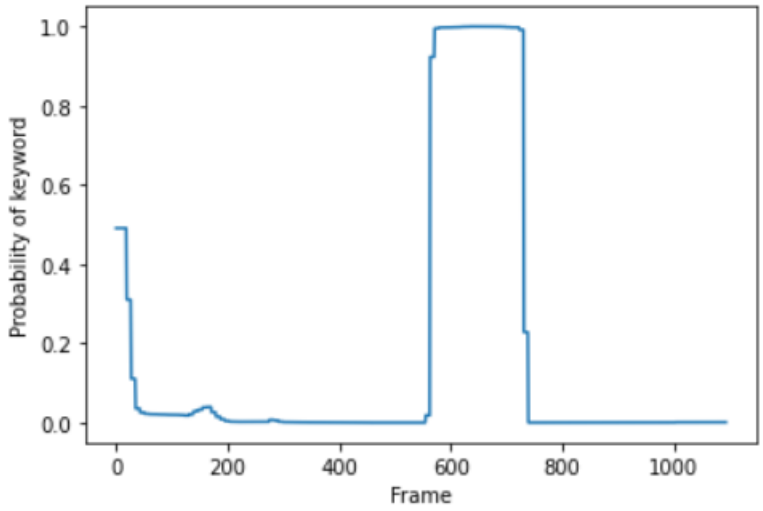

Рис. 2: Вероятности, посчитанные стриминговой моделью

итоговая функция ошибки у меня считается по формуле $\mathcal{L} = \text{softloss} + \frac{1}{2}\,\text{hardloss}$. Больше я эти параметры не подбирал, потому что у меня и так получились достаточно хорошие результаты, но об этом позже.

После этого я начал подбирать параметры для student-модели. Я посмотрел на кол-во параметров в разных слоях изначальной модели и увидел, что большая часть находится в GRU, поэтому я решил в первом эксперименте просто уменьшить hidden size с 64 до 32. Уже это в принципе дало мне достаточно хорошие результаты по памяти: кол-во потребляемой памяти уменьшилось более чем в 3 раза. По скорости работы тоже наблюдалось заметное улучшение, хоть и не такое сильное, примерно в 2 раза. Качество на валидации при этом было достаточно хорошим: всего лишь 2.5×10^{-5}, но обучал модель я на этот раз уже 40 эпох, а график обучения можно посмотреть на картинке 3.

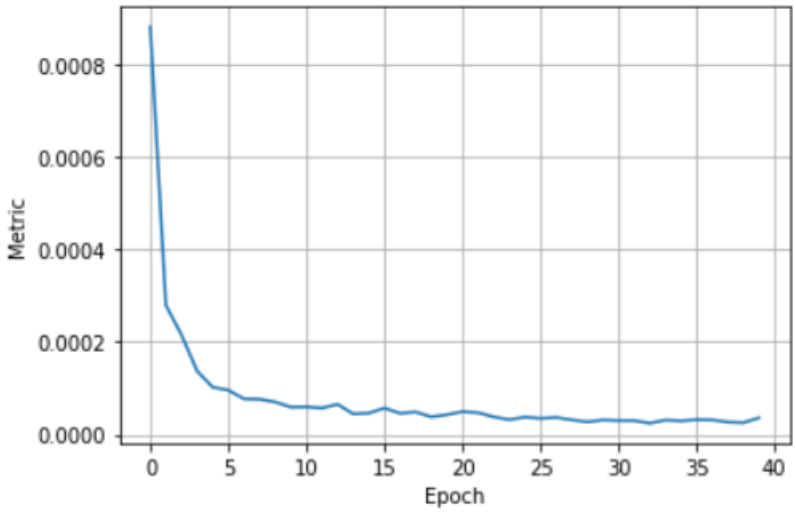

Рис. 3: Обучение student модели с hidden size = 32

Квантизация

После первого эксперимента я решил пока отложить эксперименты с дистилляцией, потому что она уже дала достаточно много. Я подумал, что если получится квантизовать модель в qint8, не сильно потеряв в качестве, то по памяти улучшение будет уже больше, чем в 10 раз, и по скорости тоже должен быть хороший прирост. И у меня действительно получилось хорошо квантизовать, используя dynamic quantization в GRU, attention и в классификаторе, но выигрыша по времени это дало довольно мало. Дело в том, что thop не умеет считать MACs для квантизованных слоев. Локально я замерял время

выполнения с помощью %timeit, и ускорение было не особо сильным, что у бейзлайна, что у дистиллированной модели. Пока что не буду приводить здесь точные значения, сделаю это в конце, когда буду сравнивать все модели.

Дистилляция Part II

Я понял, что так просто ускорить модель в 10 раз не выйдет, и нужно либо писать pruning, либо подбирать параметры для дистилляции. Я выбрал второе, потому что код уже был написан, и я к тому времени провел всего лишь один эксперимент. Во втором эксперименте я просто уменьшил hidden size до 16, качество получилось примерно 4.9×10^{-5}, а график получился как на рисунке 4. Я обучал модель 40 эпох, чтобы сравнение с первой дистиллированной моделью было более репрезентативным. Качество этой модели уже почти впритык к порогу, поэтому может показаться странным, что я после этого продолжил уменьшать параметры student-модели. Но дело в том, что когда я в первый раз провел эксперимент, у меня, видимо из-за random seed-а, качество было получше, и выглядело так, как будто есть еще пространство для улучшения.

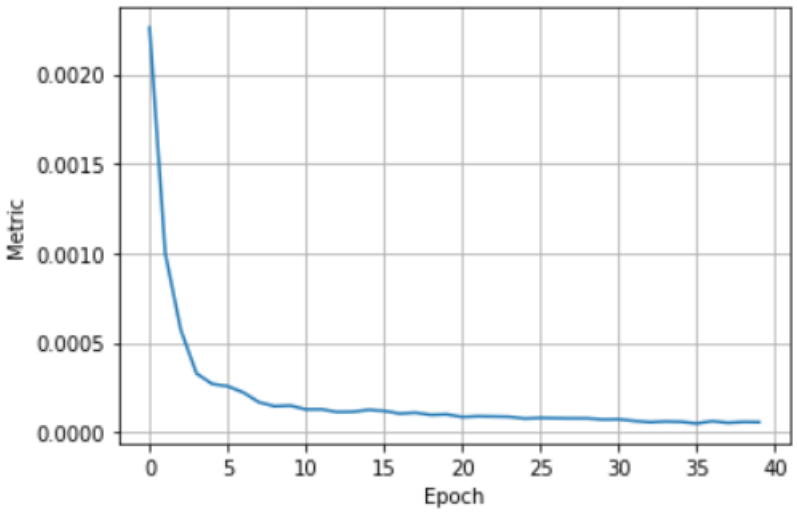

Рис. 4: Обучение student модели с hidden size = 16

Параметры в 3-ем эксперименте я подбирал более аккуратно. Так как квантизация не особо ускорила мои модели, я задался целью получить ускорение в 10 только с помощью дистилляции. И подбирал я параметры следующим образом: вбивал какие-то комбинации параметров и запускал thop.profile на необученной модели. MACs не зависит от того, обучена ли модель, поэтому я мог заранее посмотреть какие комбинации параметров могут дать ускорение в 10 раз, не обучая при этом модель. Попробовав разные комбинации, я остановился на следующих изменениях: hidden size = 16, cnn out channels = 4, kernel size = (10, 20), stride = (5, 10). Модель с такими параметрами имеет как раз примерно в 10 раз меньше MACs, чем исходная, при этом такая конфигурация, на мой взгляд, выглядит довольно здраво и имеет шанс обучиться на приемлемое качество. Дальше я сделал дистилляцию с такими параметрами и обнаружил, что модель действительно обучается на приемлемое качество, как видно на графике 5. Что интересно, в финальной версии ноутбука у этой модели качество равно $\approx 4.7 \times 10^{-5}$, т.е. лучше, чем у предыдущей модели. Думаю, это связано, с тем, что изменение kernel size и stride, возможно, вообще повлияло в лучшую сторону, хоть и параметров стало меньше, а уменьшение cnn out channels тоже, видимо, не особо мешает модели обучаться. Но так или иначе, я с помощью одной лишь дистилляции получил модель, которая более чем в 10 раз ускоряет исходную, и по памяти оптимизирует тоже более чем в 10 раз. Но так как я до этого уже немного поработал с квантизацией и разобрался, как она делается, я решил еще применить и ее ко всем этим моделям. Если кратко резюмировать, то потребление памяти упало очень сильно, время работы чуть-чуть улучшилось и метрики чуть ухудшились. В итоге моя самая маленькая модель ускорила baseline примерно в 10 раз и уменьшила потребление памяти в 38(после квантизации).

Сравнение

Давайте сравним все модели, которые я рассматривал во время выполнения этого домашнего задания. На графике 6 проводится сравнение неквантизованных моделей. Мы видим, что время и память ведут себя похожим образом в зави-

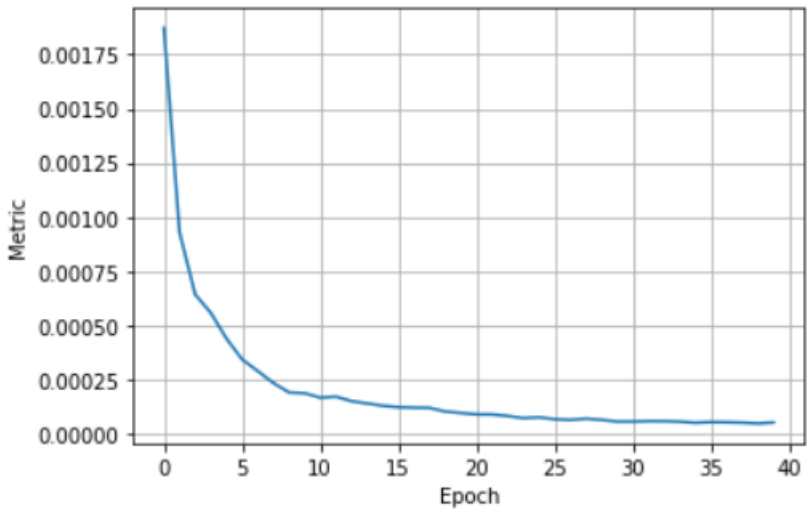

Рис. 5: Обучение student модели с hidden size = 16 и cnn out channels = 4

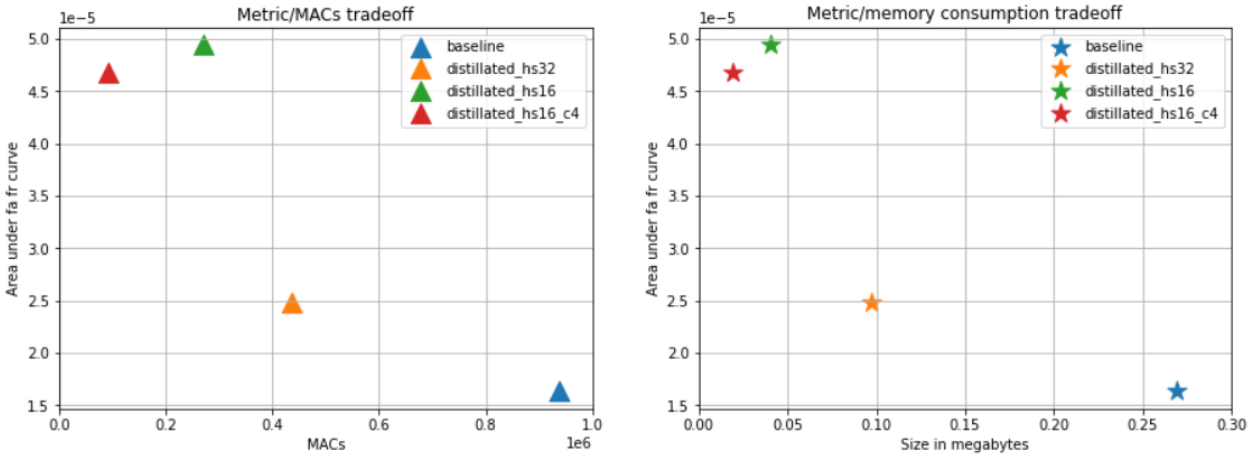

Рис. 6: Сравнение неквантизованных моделей

симости от модели. Из графиков очевидно, что distillated_hs16 показывает себя хуже остальных моделей, потому что distillated_hs16_c4 опережает ее во всем. Если забыть про эту модель, то каждая из 3 оставшихся моделей может быть полезна в тех или иных условиях, в зависимости от ограничений по памяти и времени. Теперь давайте сравним их с квантизованными моделями. Сравнить их по производительности не так то просто, а вот по потребляемой памяти вполне можно. Для квантизованных моделей я считаю размер модели следующим образом: просто у квантизованных слоев делю размер на 4, потому что мы квантизовали из float32 в qint8. На графике 7 видно, что для больших моделей квантизация очень существенно уменьшает потребляемую память, при этом качество ухудшается несильно. Для маленьких моделей уже нельзя сказать, что не имеет смысла использовать неквантизованную модель, потому что качество ухудшается ощутимо. Но давайте все-таки попытаемся сравнить производительность всех этих моделей. Сделаем это так: просто подадим на вход каждой модели тензор размера (1, 40, 100), что соответствует аудиозаписи длиной 1 секунда, и замерим время выполнения с помощью %timeit. Результаты мы видим на графике 8, и они очень неожиданны. Мы видим, что квантизация дает некоторый прирост производительности, хоть и небольшой, но модели очень слабо отличаются между собой. Скорее всего это связано с тем, что на вход подавался очень маленький тензор. Я пытался таким образом приблизить ситуацию, когда во время inference-а нам поступает на вход одна аудиозапись длительностью 1 секунда, как в нашей выборке. Но давайте интереса ради попробуем подать на вход тензор размером побольше. Результат мы видим на картике 9. При таких условиях эксперимента модели между собой отличаются уже посильнее, но такой же разницы, как в MACs, между ними не наблюдается. Возможно, MACs — не очень репрезентативная метрика, если дело идет о вычислении на процессоре. Но мы все же видим, что квантизация дает небольшой прирост производительности.

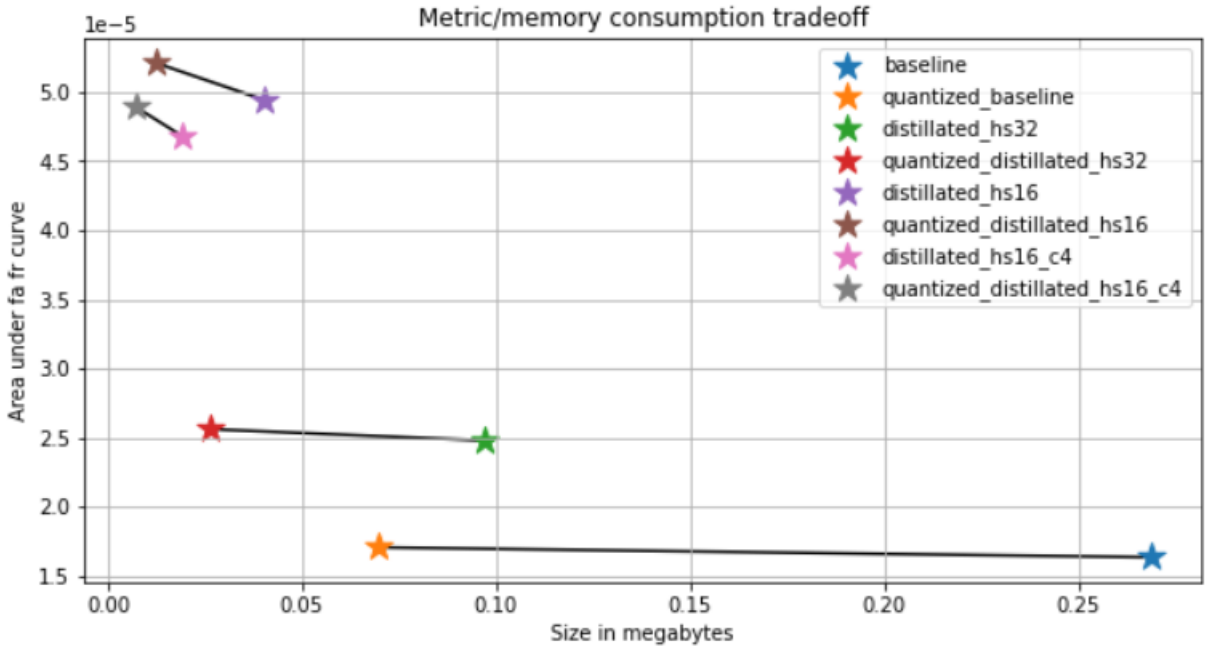

Рис. 7: Сравнение потребления памяти всех моделей

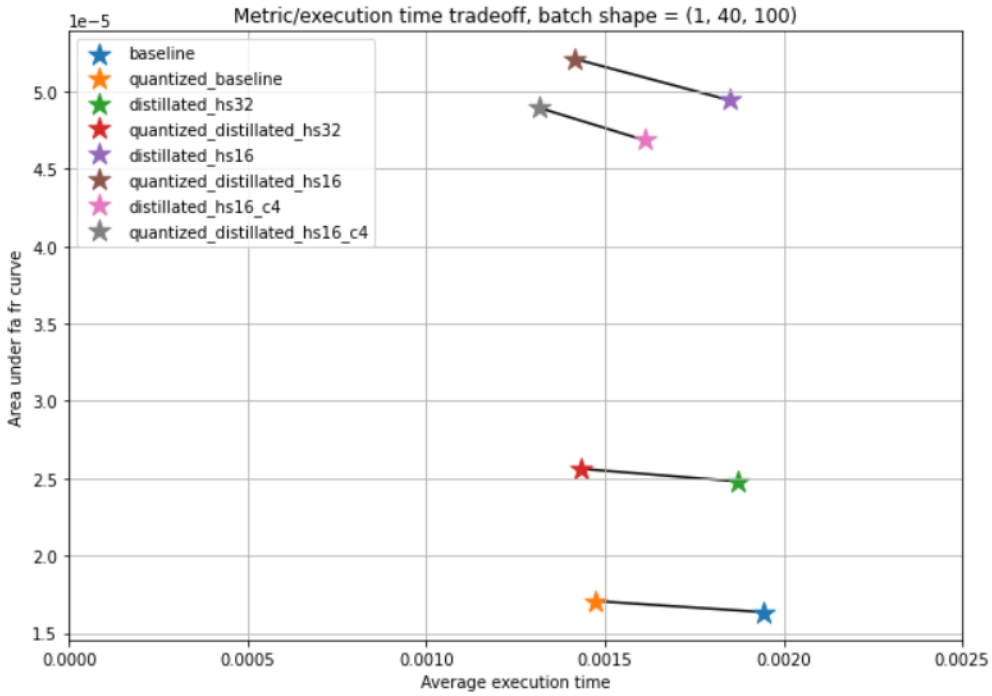

Рис. 8: Сравнение времени работы на тензоре размера (1, 40, 100)

Трудности, с которыми столкнулся

Если так подумать, то часть с ускорением модели мне далась довольно легко. Я тем не менее потратил немало времени на то, чтобы разобраться получше с дистилляцией и квантизацией. Еще я в очередной раз напоролся на то, что в .ipynb ноутбуках очень легко накосячить с воспроизводимостью, потому что можно позапускать ячейки в каком-то порядке, получить хороший результат, забыть, в каком порядке запускались ячейки, а потом выясняется, что если прогнать ноутбук с начала, то результаты не воспроизводятся. После этого домашнего задания я серьезно призадумался о целесообразности их использования, в случае если в них выполняются какие-то тяжелые операции. Еще одной сложностью, связанной с

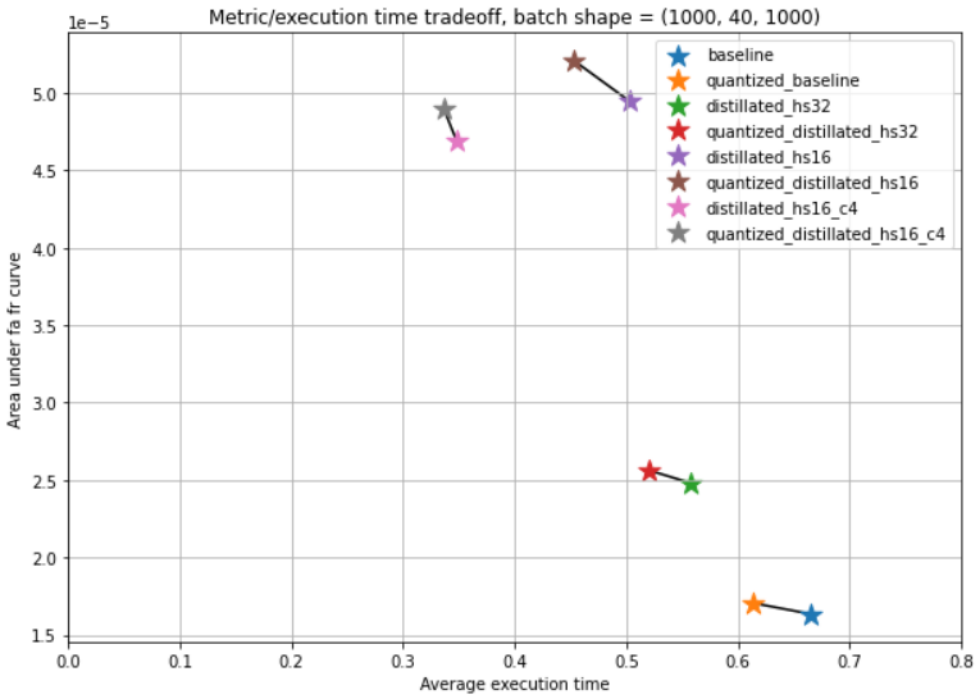

Рис. 9: Сравнение времени работы на тензоре размера (1000, 40, 1000)

ускорением, стало исползование библиотеки thop. Во-первых, там нет никакой поддержки квантизованных слоев. Но это понять можно, потому что, судя по всему, у квантизованных слоев MACs такой же, как и у обычных. Но что меня совсем не порадовало, так это отсутствие нормальной поддержки использования своих слоев. В библиотеке есть способ добавить функцию подсчета MACs для слоя, написанного своими руками, но как им пользоваться нигде не объясняется, и для этого пришлось разбираться в исходниках. Более того, нельзя никак сделать импорт функции, которая считает MACs для какого-нибудь слоя из torch.nn. Например, в слое attention у нас используется softmax, и thop умеет считать для него все необходимое, но сделать импорт нужной функции никак нельзя, и мне пришлось просто ее скопировать из исходников.

В итоге получается, что основная часть работы в моем случае была заключена в реализации стриминга и в переписывании кода. В стриминге сложность заключалась в том, что нужно было разобраться, как там все пересчитывать, и аккуратно все написать, нигде не набагав. Написание template-а и рефакторинг заняли так много времени, пожалуй, потому что я никогда с нуля не писал проекты на pytorch-е, поэтому приходилось многие моменты обдумывать, как все это хорошо и красиво сделать. Но это вообще было необязательно делать, поэтому вряд ли это можно отнести прямо к трудностям, скорее мне самому хотелось потренироваться в подобных вещах.

Если подытожить, то можно сказать, что каких-то больших сложностей я не испытывал во время выполнения домашнего задания, но вся работа вместе взятая все равно заняла довольно много времени. Возможно, у меня все так легко пошло с ускорением модели, потому что сгенерировалось удачное разделение на train и val. Базовая модель обучилась на 1.6×10^{-5} вместо положенных 5×10^{-5} явно неспроста. Возможно, тут помогла магия torch.manual_seed(3407), но на войне все средства хороши :) Еще раз замечу, что код, генерирующий мое разделение, находится в train_val_split.py, и его результат воспроизводится.