ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

О Т Ч Е Т
по учебной практике
на факультете компьютерных наук НИУ ВШЭ

Выполнил:
Студент группы БПМИ208 _____ Брусенин Д.А.
Подпись И.О.Фамилия

20.10.2022
Дата

Руководитель практики Колесниченко Елена Юрьевна
(ФИО руководителя практики)

департамент больших данных и информационного поиска ФКН ВШЭ, доцент
Подразделение ФКН, должность

Дата – 2022 _____ –
Подпись

Москва 2022

Содержание

1 Аннотация

Тут будет аннотация

2 Введение

Графические процессоры способны быстро обрабатывать большие объемы данных, но имеют меньшую точность по сравнению с центральными процессорами, но тем не менее точности, достигаемой с помощью GPU, хватает для решения многих задач, в том числе для большинства задач машинного обучения. В ML очень важна скорость обучения моделей, особенно она критична для громоздких моделей с большим количеством параметров, например, для нейронных сетей, поэтому возникает резонное желание ускорить процесс тренировки моделей за счет распределённого обучения и использования графических процессоров.

3 PipeDream

Главными подходами для распределенного обучения нейронных сетей является data-parallel и model-parallel. Однако эти подходы не лишены недостатков. В статье "PipeDream: Fast and Efficient Pipeline Parallel DNN Training"описаны основные проблемы данных подходов и предложена система PipeDream, комбинирующая эти методы и тем самым оптимизирующая их.

3.1 Data parallelism

Идея Data parallelism заключается в разбиении исходных данных на части, каждая из которых будет использована на отдельной GPU. На каждой GPU находится своя полная копия Нейронной Сети, и веса, получаемые в ходе обучения, синхронизируются между GPU раз в несколько эпох.

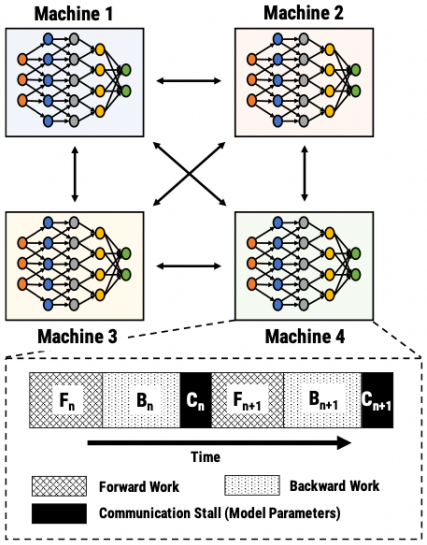


Рис. 1: Пример data-parallel на 4 GPU

Однако главная проблема data-parallel заключена в долгом синхронизировании данных между процессами. Были проведены тесты на различных известных нейронных сетях на трех разных видеокартах NVIDIA GPU – Kepler (K80), Pascal (Titan X) и Volta (V100), и было замерено время, идущее на синхронизацию между процессами:

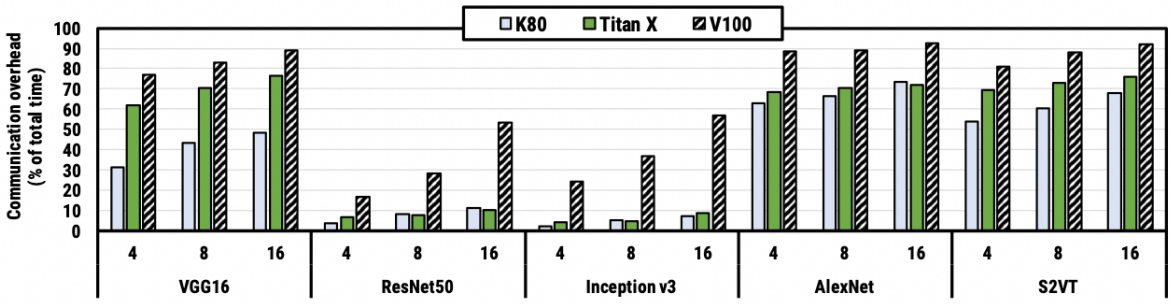


Рис. 2: Время на синхронизацию в % от всего времени обучения DNN

Как видно на данном графике, время, занимаемое на синхронизацию , может достигать до 90% от всего времени обучения DNN.

3.2 Model Parallelism

Идея Model Parallelism заключается в разбиении самой модели на несколько частей, каждая из которых будет вычисляться на отдельной GPU. Данный подход часто используется в Машинном Обучении (ML). Однако для DNN наивный подход model-parallel имеет огромный недостаток. Разбивая Нейронную Сети на части по несколько слоев Нейронной Сети, отдельная часть должна вычисляться только после вычисления предыдущей части:

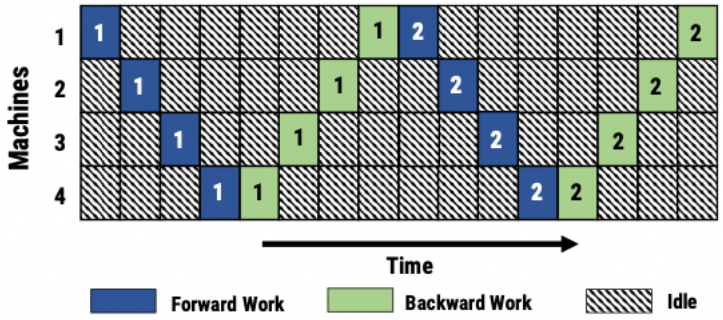


Рис. 3: Процесс наивного model-parallel на 4 GPU.
Числа обозначают номер батча.

Из-за такого подхода наши GPU значительную часть времени находятся в режиме ожидания, и как такового прироста в производительности нет.

3.3 PipeDream

PipeDream использует идею model-parallel, деля DNN на части по несколько слоев Нейронной Сети, но при этом, прямые и обратные проходы для разных батчей будут выполянться параллельно, тем самым минимизируя работу GPU в режиме ожидания.

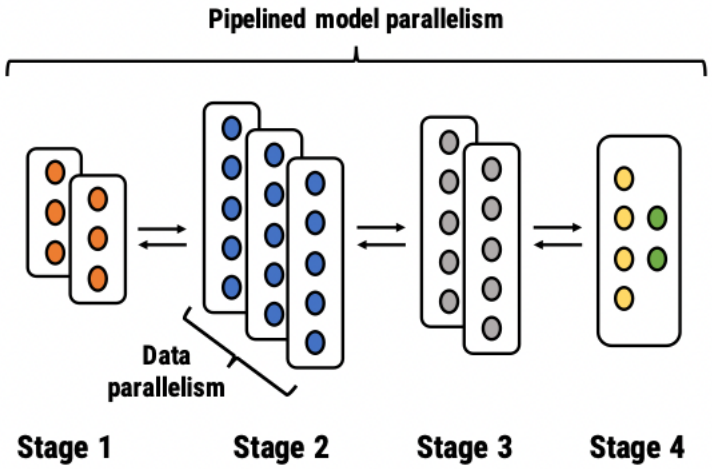


Рис. 4: Процесс обучения PipeDream, объединяющий data-parallel и model-parallel

3.3.1 Распределение слоев между GPU

Для минимизации времени работу GPU в режиме ожидания важно правильно поделить DNN на части. Главная задача при разделении заключается в том, что каждая отдельная компонента PipeDream выполнялась одинаковое количество времени.

Для этого вначале производится замер времени работы DNN, производя вычисления 1000 батчей на одной из GPU. После этого при помощи динамического программирования находится оптимальное разделения на части, каждая из который будет вычисляться одинаковое количество времени (тогда мы можем считать, что каждая часть выполняется ровно одну единицу времени).

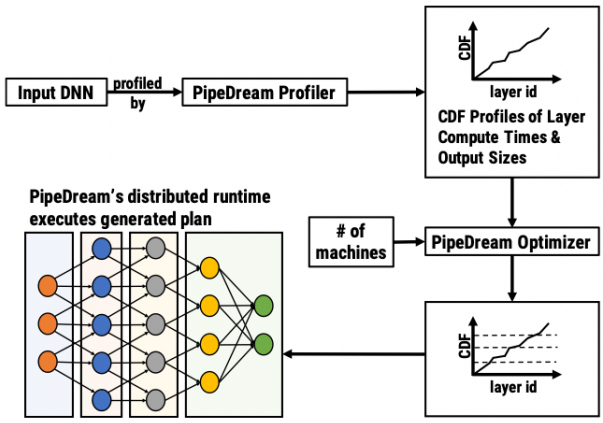


Рис. 5: Автоматизированный процесс деления DNN на части

3.3.2 Последовательность вычисления

В отличие от обычных реализаций pipeline, в которых одновременно вычисления происходят только либо в прямом, либо в обратном порядке, в PipeDream вычисления происходят в обе стороны одновременно. То есть у каждой GPU в каждую единицу времени есть выбор: либо делать прямой проход графа вычислений, либо обратный.

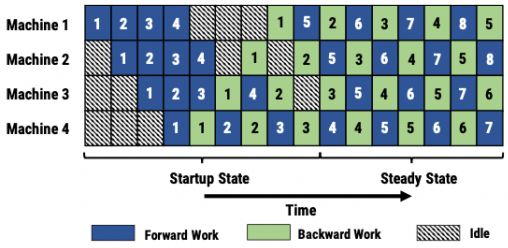


Рис. 6: Автоматизированный процесс деления DNN на части

В начальной стадии испульзнения PipeDream, первая часть получает на вход оптимальное количество батчей, последовательно после вычисления передавая следующей части (а значит и следующей GPU) результат вычисления.

В основной стадии работы PipeDream на одной GPU чередуются прямой и обратный проходы. Такой механизм авторы статьи назвали one-forward-one-backward (1F1B).

3.3.3 Эффективность обучения

Как мы можем видеть из примера из последней иллюстрации: прямой проход для батча №5 для первой GPU проходит после обновления весов после обратного прохода батча №1. При этом обратный проход батча №5 проходит после оборатных проходов батчей №2, №3 и №4. Это может негативно влиять на сходимость.

Weight Stashing: в PipeDream реализована идея сохранения весов для всех батчей, которые в данный момент времени вычисляются в модели. Таким образом, когда батчу нужно пройти прямой проход, то используются последняя версия весов. После этого сохраняется версия весов, которая вспоследствии будет использована при обратном проходе того же батча. Такой метод авторы статьи называют Weight Stashing. Благодаря ему для каждой части модели для каждого батча используются подходящие веса, как для прямого, так и для обратного проходов. Тем самым не будет нарушена сходимость графа вычислений.

3.4 Заключение

PipeDream существенно превосходит метод data-parallel, посколько каждая GPU обменивается только частью параменторв (таким образом синхронизация происходит быстрее):

DNN Model	# Machines (Cluster)	BSP speedup over 1 machine	PipeDream Config	PipeDream speedup over 1 machine	PipeDream speedup over BSP	PipeDream communication reduction over BSP
VGG16	4 (A)	1.47×	2-1-1	3.14×	2.13×	90%
	8 (A)	2.35×	7-1	7.04×	2.99×	95%
	16 (A)	3.28×	9-5-1-1	9.86×	3.00×	91%
	8 (B)	1.36×	7-1	6.98×	5.12×	95%
Inception-v3	8 (A)	7.66×	8	7.66×	1.00×	0%
	8 (B)	4.74×	7-1	6.88×	1.45×	47%
S2VT	4 (A)	1.10×	2-1-1	3.34×	3.01×	95%

Рис. 7: Сравнение PipeDream с data-parallelism (BSP)

По резуьтатам экспериментов для моделей VGG16 и S2VT время, затраченное на синхронизацию, уменьшилось на 90%, при этом в целом модель стала в среднем обучаться в 3 раза быстрее.

4 Tensor Parallelism NLP

Следующий текст отражает [3].

У нас есть классные задачки на NLP, модели GPT-2 и Bert, которые достигли State-of-the-art (выдающихся результатов) в решении каких-то сложных задач. Но есть проблема - они состоят из большого количества параметров и требуют довольно много данных чтобы обучится. В итоге обучить их становится довольно трудно, если использовать только один процессор.

Поэтому люди делают обучение многопоточным. Некоторые популярные способы:

1. Разбивают данные на партии (batch-и), и их отдают модели параллельно на несколько GPU. Затем результаты агрегируются через функцию (например берется среднее градиентов для последующего изменения весов модели)

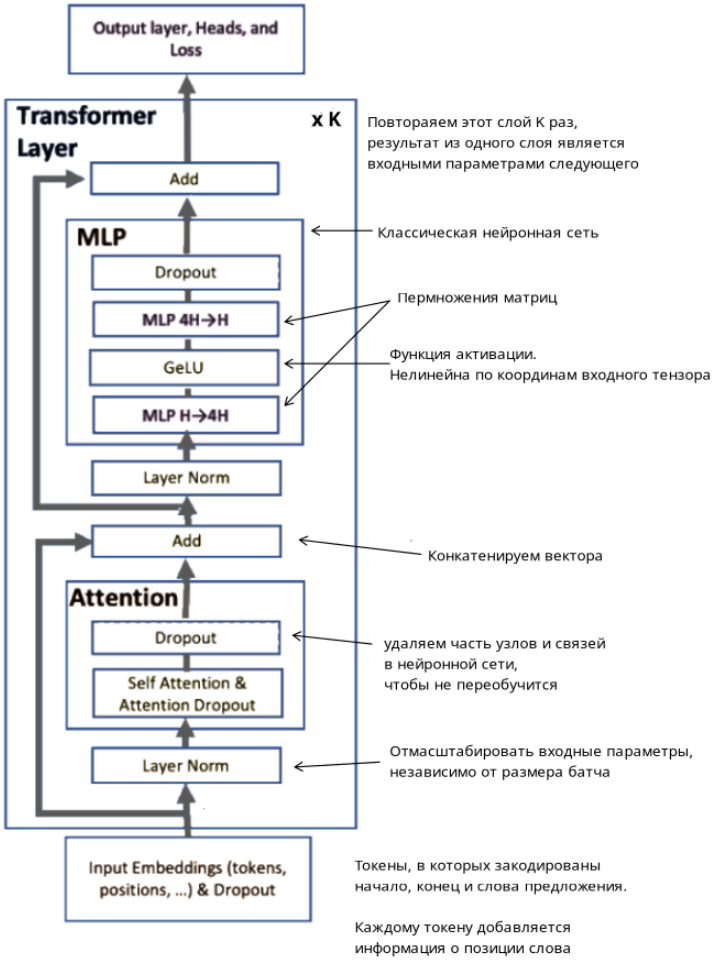
2. Делают модель распределенной. Разбивают модель на набор операций, и разносят их по процессам, а затем синхронизируют. Здесь нужна дополнительная логика поверх используемой модели.

Авторы используют немного другой подход. Одни из самых частых операций - это вычисления с тензорами (например перемножения матриц). Поэтому можно разбить тензор параметров модели на несколько, все полученные составляющие раскидать по разным GPU, каждому отдать одни и те же входные данные, и запустить параллельную обработку. Если в модели есть нелинейный слой, требующий всех данных сразу -

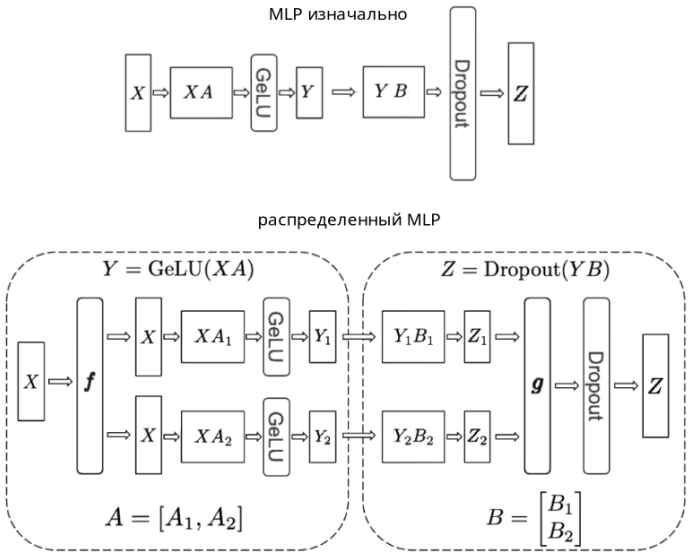
собрать результат с всех процессоров и пропустить через нее. Чем меньше точек синхронизации, тем лучше производительность. А из-за того, что мы разбиваем элементы модели, не меняя особо процесс ее обучения - то изменения нужны минимальные.

Описанные дальше подходы можно применять для любой подходящей модели, но авторы взяли задачи NLP. В основе наиболее успешных моделей лежат трансформеры (Transformers). Его истоки состояли из RNN - что вообще не ложится на параллельность. Но [1] от реккурентости избавляются, что позволяет обрабатывать входные данные и выполнять некоторые слои параллельно. Теперь трансформеры - корень всех моделей, их можно разбить на encoder-a и decoder-a, собирать их последовательно, навешивать дополнительные слои, обучать модели для разных задач и получать удивительные результаты. Bert обучался понимать структуру языка, изначально решая набор базовых задач (как вставить правильное слово в пропуск и сказать про предложения - они связаны или нет). Затем он дообучается для более конкретной задачи. GPT-2 это модель которая умеет по началу предложения генерировать продолжение. Сначала GPT уступало Bert, но разработчики значительно увеличили количество параметров до 1.5 млдр, сделав ее самой большой на тот момент, разнообразили датасет из статей из интернета, и она достигла воистину state-of-the-art, ее даже боялись публиковать полностью. Поэтому чтобы ускорить модели, можно оптимизировать именно трансформеры.

Это модель трансформера, взятая авторами статьи:

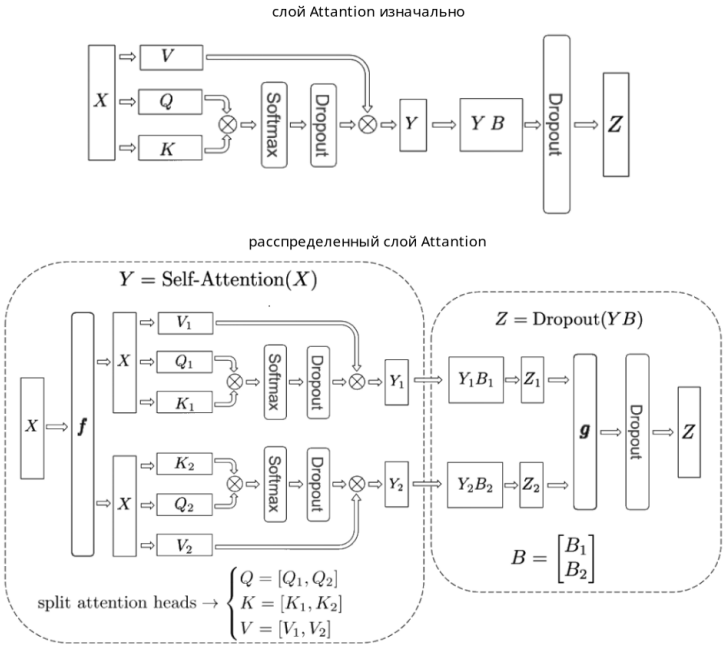


Найдем, где в этой модели используется перемножение матриц. В более простом блоке нашей модели - MLP преобразования будут следующими:



Если посмотреть на них внимательно, то обе схемы почти одинаковые. Только в нижней есть блок f, который в прямом проходе дублирует данные на несколько GPU, и блок g - который собирает данные с нескольких процессоров в одну матрицу. Эти блоки являются точками синхронизации, и их на каждый проход (прямой и обратный для релаксации весов) по 2 для одного Transformer Layer-a.

Аналогично поступаем и для слоя внимания



Получаем распараллеленную модель, добавив всего два простых блока f и g.

Некоторые модели могут отличаться от представленного в статье, но почти любое перемножение матрица можно параллелить, не забывая потом добавлять простой объединяющий слой когда нужно.

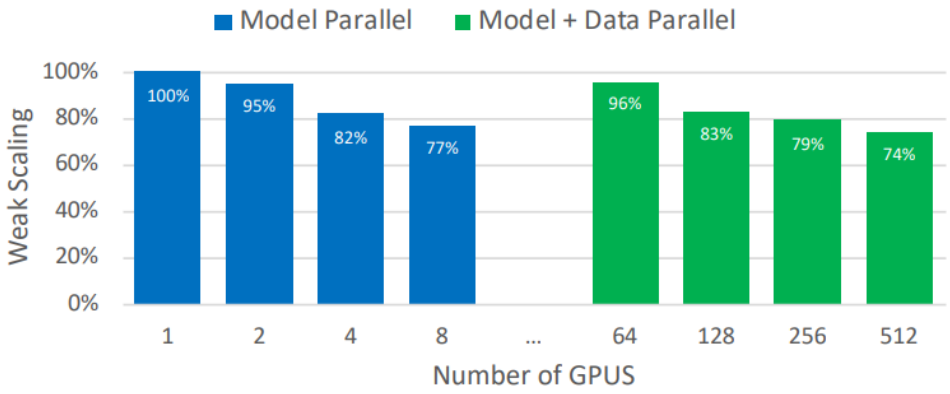
Еще одно улучшение - перед тем, как считать cross entropy, мы должны отправить одному процессору объединенную матрицу. Одной из ее размерностью является размер словаря, что для хороших моделей будет большой. Там будут находится вероятности для данного слова быть в предложении следующим. Так вот, всю эту размерность можно схлопнуть, если считать cross entropy на каждом GPU работающем параллельно, и

отправлять уже меньше данных. Авторы говорят, что это сильно улучшило их время обучения.

Теперь интересно посмотреть на то, как хорошо это работает.

Для обучения моделей был собран единый датасет из существующих (тексты с Википедии, новостей и тд), который весил 174 Гибибайтов.

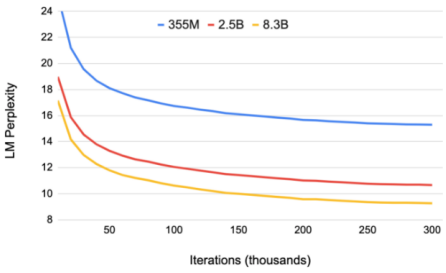
Сначала применили паралелизацию описанную выше для 1-2-48 GPU, и посмотрели на эффективность (на графике синие столбца). Раньше я также говорила про один из способов ускорить обучение - разбить данные на batch-и. Это называется data parallel, и его можно также внедрить в схему. До этого мы обучающие данные копировали полностью на несколько GPU, и на каждом работали с разными весами модели. Теперь можно вместо одной GPU в схеме, поставить 64 идентичных, полностью скопировать внутреннее состояние, а вот входные данные разбить между ними. Получается что так мы сначала делим модель, а затем входные embedding-и. Используемых GPU получаем больше, измеряем эффективность распределенных вычислений на них и рисуем зеленные столбцы.



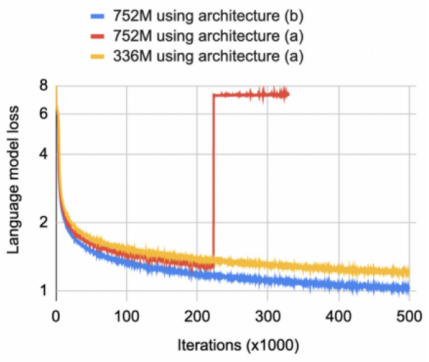
Что вообще на этом графике изображено и про что weak scaling? Для его измерения мы увеличиваем количество GPU и количество входных данных нашего датасета для нашей модели. В идеальном мире, время работы каждого отдельного процессора не должно изменится, ведь он обрабатывает одно и тоже количество данных, и это мы называем 100%. Но в реальном мире у нас есть накладки например в виде времени обмена данными между GPU, и поэтому время работы немного увеличивает по сравнению с идеальным вариантом. Weak Scaling здесь показывает эффективность наших действий с добавлением большего числа GPU по сравнению с идеальным вариантом.

В итоге, авторы статьи достигают 77% и 74% по сравнению с наилучшим вариантом, в котором бы при увеличении размера входных данных время обучения бы не изменилось. Чтобы как-то ощутить сколько это хорошо, модель GPT-2 которая имеет 355 миллиона параметров обучалась одну эпоху за 20.6 часов, а модель с 8.3 миллиардами (в 23 раза больше) - за 50.4 часов, что является самой большой моделью, что когда-либо собирали на момент написания статьи.

Более того, большие модели показывают лучше результат - на графиках видно, что чем больше параметров у модели, тем меньше функции потерь. На втором графике архитектура (a) (классический Bert) от архитектуры (b) (на что мы смотрели в этой статье) отличается количеством вектором, проходящих через слов Layer Norm. Как видно на графике, у классического Bert-а с большим числом параметров начинает переобучение, поэтому авторам статьи пришлось изменить модель.



GPT-2



BERT

Суммируя рассказ, модели нейронных сетей с большим количеством параметров могут работать лучше, но также требуют значительно больше временных ресурсов чтобы обучится. Можно использовать больше процессоров, и внеся довольно мало изменений в работу программ, в итоге получить результаты за более разумное время.

5 TODO

Чем больше датасет - тем больше времени тратится на обучение модели по нему. Распределенный синхронный стохастический градиентный спуск - это одно из возможных решений этой проблемы.

5.1 Описание

Предположим, у нас есть 8 GPU и мы хотим обучать модель, используя SGD на мини-батчах размера n.

В первую очередь, мы увеличиваем размер батчей до kn, чтобы затем распределить его на k воркеров. И здесь мы задаемся первым вопросом - а не ухудшится ли точность модели от увеличения размера батча?

Обычно функция потерь для модели имеет следующий вид:

$$L(w) = \frac{1}{|X|} \sum_{x \in X} l(x, w)$$

Здесь w - это параметры модели, X - обучающая выборка, а l – некоторая функция ошибки, которая в том числе может включать в себя нормализацию.

Тогда один шаг обучения SGD имеет следующий вид:

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in \mathcal{B}} \nabla l(x, w_t)$$

Здесь η – это скорость обучения, гиперпараметр модели, а $\mathcal{B}$ – текущий батч.

Заметим, что если мы увеличим размер батча в k раз, то количество слагаемых в сумме уменьшится в k раз. Отсюда вытекает идея: давайте увеличим скорость обучения в k раз, чтобы скомпенсировать это. И на реальных данных оказывается, что это действительно неплохой подход, который сохраняет точность модели, за исключением одного случая - начала обучения.

В самом начале, когда модель меняется больше всего, предлагается начать со старой скорости обучения, равной η, а затем каждую эпоху увеличивать ее на константу так, чтобы к пятой эпохе она стала равной $k\eta$. А далее сохраним эту скорость обучения.

Теперь, когда мы научились увеличивать размер батча без особых потерь, будем распределять каждый батч между воркерами, а затем синхронизировать полученные результаты между ними.

5.2 Анализ

Обучалась модель ResNet-50 на датасете ImageNet в течение 90 эпох. В результате модель обучилась за 1 час с размером батча 8192.

Для сравнения, на одной видеокарте с размером батча 256 модель обучается около 29 часов.

При этом самая большая ошибка обучения в датасете увеличилась не более, чем на 0.3%.

6 Large Scale Distributed Deep Networks

Начнем изучение темы распределённого обучения моделей со статьи “Large Scale Distributed Deep Networks” [2] , в которой делается большой шаг в сторону распределенного обучения нейронных сетей.

Нейронная сеть состоит из нескольких слоев нейронов и синапсов, связей между нейронами. Каждый нейрон принимает на вход какие-то небольшие данные, производит над ними работу и передает полученный результат по синапсу следующему нейрону. У каждого синапса есть вес, который влияет на то, какой вклад внесет результат вычислений одного нейрона на результат вычислений следующего.

Веса синапсов - это одни из параметров нейронной сети и в задачу обучения сети входит подбор весов синапсов так, чтобы минимизировать ошибку модели. Таким образом, подбор весов нейронной сети сводится к задаче минимизации функции ошибки.

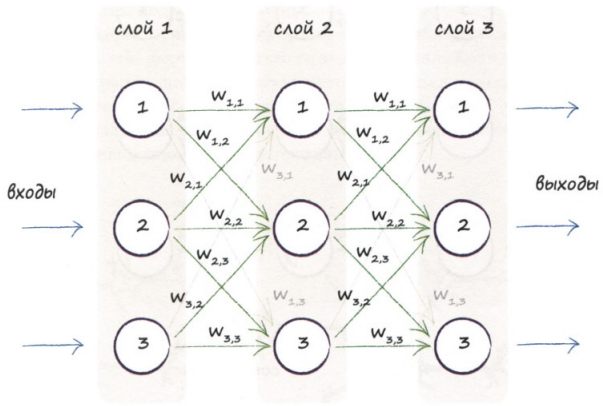


Рис. 8: Визуализация нейронной сети

Основной вклад, который внесла статья в развитие распределённого обучения моделей, заключается в том, что авторы разработали два распределенных метода оптимизации и провели серии экспериментов, показывающих эффективность этих моделей по сравнению с нераспределёнными, работающими на CPU или на соптимизированном GPU.

В статье авторы описывают два распределенных метода оптимизации, которые можно применять для поиска весов нейронной сети: Downpour SGD и Sandblaster L-BFGS. Оба метода были разработаны для фреймворка DistBelief, поддерживающего параллелизм моделей, когда модель делится на части и каждая часть обучается на одном и том же датасете, и параллелизм данных, когда копия модели обучается на разных поднаборах датасета.

6.1 Downpour SGD

Downpour SGD - асинхронная вариация градиентного спуска SGD. В основе распределенности этой модели лежит параллелизм данных. Несколько копий модели запускаются на разных репликах, каждой реплике отведен поднабор параметров, который она будет менять во время обучения. Модели обмениваются обновленными параметрами с помощью центрального сервера параметров, который хранит текущее состояние всех параметров, которые используются на разных репликах.

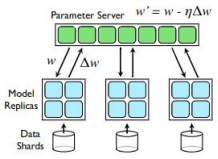


Рис. 9: Визуализация Downpour SGD

Преимуществом Downpour SGD по сравнению с синхронной распределенной вариацией SGD является его устойчивость к отказам машин, поскольку каждая реплика индивидуально выполняет свою работу и по завершении не ждет остальных, чтобы обновить значения параметров. Основной проблемой асинхронного подхода является невозможность поддержания консистентности значений параметров для каждой реплики. Поскольку каждая реплика действует независимо, то нет никакой гарантии, что в любой момент времени к параметрам на разных машинах было применено одинаковое количество обновлений или что обновления были применены в одном и том же порядке.

6.2 Sandblaster L-BFGS

Авторы рассматривают распределенное обучение модели L-BFGS, заключающееся в параллелизме данных.

Ключевые идеи распределенности для алгоритма L-BFGS, которые предлагают авторы статьи:

- распределенное хранение параметров
- выделение процесса координатора, который не имеет доступа к параметрам модели и выступает в роли оркестратора всех реплик, выполняющих обучение

Координатор назначает каждой из реплик модели небольшую часть работы, которую она может выполнить независимо, и каждый раз, когда какая-то реплика заканчивает свою работу, координатор назначает ей новую часть. Такой подход позволяет хорошо использовать быстрые машины и исключает ожидания самой медленной реплики, чтобы продолжить процесс. Результаты работы, а также кэшируемая информация сохраняются локально на каждой реплике и не отсылаются в центральный сервер, что позволяет запускать обучение достаточно больших моделей без оверхеда на отправку и сбор всех параметров.

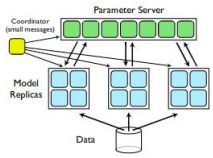


Рис. 10: Визуализация Sandblaster L-BFGS

6.3 Достигнутые результаты и проверка эффективности

Авторы статьи применяют описанные методы оптимизации для двух задач глубинного обучения: распознавание объектов на изображениях и акустическая обработка для распознавания речи. Ниже приведены графики для задачи обработки речи. Авторы сравнивают три оптимизационных модели, которые тренируются распределённо: Downpour SGD, Downpour SGD w Adagrad, Sandblaster L-BFGS с двумя моделями, которые были обучены на центральном процессоре одной реплики (черная линия) и на оптимизированном графическом процессоре (розовая пунктирная линия).

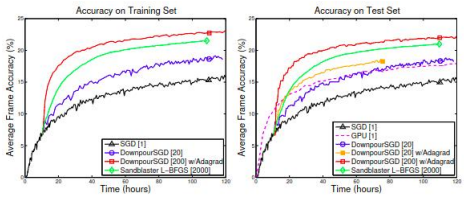


Рис. 11: Зависимость точности модели от времени обучения

Весьма значительным результатом является то, что авторы статьи применили разработанный метод оптимизации к модели с 1.7 миллиардами параметров и получили точность классификации 15% на кросс валидации, что является относительным улучшением более чем на 60% по сравнению с лучшей производительностью в этой задаче классификации.

Список литературы

1. Attention is all you need / Vaswani [и др.] // Advances in neural information processing systems. — 2017. — Т. 30. — URL: https://arxiv.org/pdf/1706.03762.pdf.
2. Large Scale Distributed Deep Networks / J. Dean [и др.] // Advances in Neural Information Processing Systems 25 (NIPS 2012). — NeurIPS Proceedings, 11.2012. — URL: https://papers.nips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html.
3. Megatron-lm: Training multi-billion parameter language models using model parallelism / Shoeybi [и др.] // arXiv preprint arXiv:1909.08053. — 2019. — URL: https://arxiv.org/pdf/1909.08053.pdf.