Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский государственный социальный университет»

Факультет политических и социальных технологий
Направление подготовки – 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Выпускная квалификационная работа

Тема: «Применение фрактального анализа для оптимизации нейронных сетей»

Обучающийся _____ Котыга Майя Максимовна
подпись

Дата «03» июня 2024г.

Руководитель _____ Романова Елена Юрьевна, к.п.н., доцент
подпись *(Ф.И.О., ученая степень, ученое звание)*

ВКР допущена к защите «03» июня 2024г.

Декан факультета политических и социальных технологий, кандидат педагогических наук, доцент _____ Пивнева С.В.
подпись

Москва, 2024

Оглавление

Введение

Оптимизация в информатике позволяет улучшить производительность программ, сокращая время выполнения и потребление ресурсов, что, в свою очередь, положительно сказывается на функциональности программного обеспечения и эффективности работы компьютерных систем. Разработка оптимизированных программных модулей является актуальной задачей как для индустрии, так и для исследовательских проектов, поскольку позволяет повысить конкурентоспособность продуктов и улучшить пользовательский опыт.

Тема оптимизации также актуальна в области наук о данных. По состоянию на 2022 год общий объем собранных и сохраненных данных составил около 97 зеттабайт [20]. Это свидетельствует о значительном объеме информации, который необходимо эффективно обрабатывать. В этом контексте большие данные играют ключевую роль, и их обработка требует оптимальных методов, включая применение нейронных сетей.

Нейронная сеть — математическая модель, имитирующая сеть нервных клеток мозга (рисунок 1). С точки зрения математики, это сложная дифференцируемая функция (можно представить в виде суперпозиции простых функций), задающая отображение из экзогенного признакового пространства в эндогенное. Все параметры при этом настраиваются одновременно и взаимосвязанно [26].

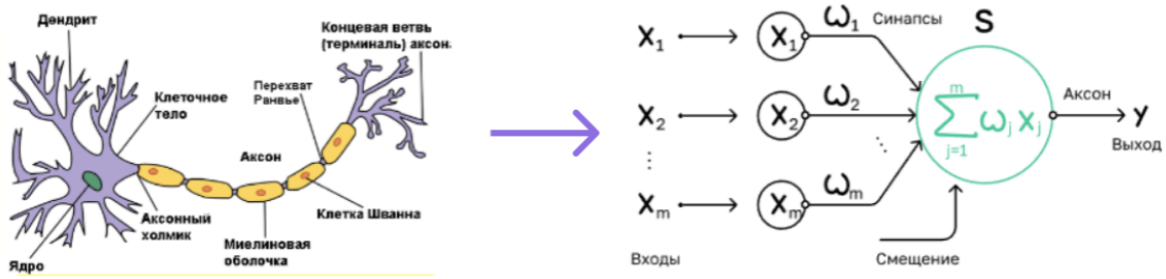

Рисунок 1. Аналогия естественной модели мозга и модели искусственной нейронной сети

У нейронной сети есть следующие образующие блоки:

- Линейный слой — линейное преобразование входных данных: $x \mapsto xW + b, W \in \mathbb{R}^{d \times k}, x \in \mathbb{R}^{d}, b \in \mathbb{R}^{k}$.
- Функция активации — нелинейное преобразование, поэлементно применяющееся к входным данным предыдущего слоя. Благодаря функциям активации порождаются более информативные признаковые описания.

В машинном обучении существует гипотеза лотерейного билета, формулировка которой звучит так: случайно инициализированная плотная нейронная сеть содержит в себе подсеть, способную достичь того же качества на тестовых данных, что и исходная сеть за то же или меньшее число итераций, что и исходная плотная сеть.

Для исследования данной проблемы предлагается использовать метод фрактального анализа. Фрактальный анализ (фрактальный прунинг) предполагает создание модели на основе исходной с последующим сравнением результатов с «материнской» моделью. Критерием оптимальности в данном случае будет отставание не более, чем на 5% от исходной модели по заданной метрике.

Актуальность выбранной темы подтверждается работой в области фрактального анализа структуры обучения нейронных сетей [18], в которой существует аналогия между процессом генерации фракталов и обучением нейронных сетей. Оба явления связаны с многократным применением функции к ее собственным результатам. Из наблюдений исследователей ясно, что тепловая карта поиска гиперпараметров для нейронной сети имеет фрактальную структуру (рисунок 2). Синие области соответствуют гиперпараметрам, при которых обучение сходится; на красных же сходимости не наблюдается. Данное открытие может привести к глубокому пониманию взаимосвязи между асимметричной геометрической структурой фракталов и динамикой в нейронных сетях. Это может способствовать улучшению методов оптимизации гиперпараметров.

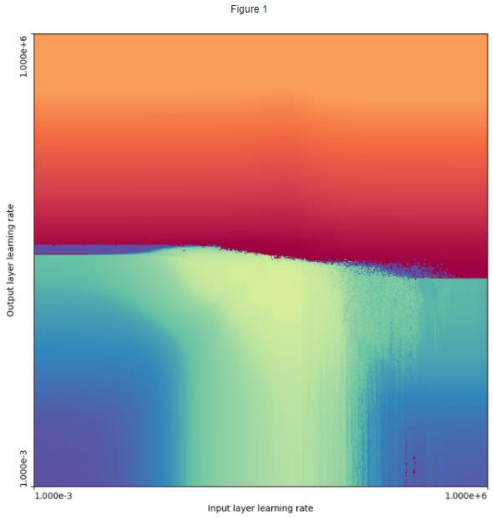

Рисунок 2. Тепловая карта поиска эффективных гиперпараметров

Фрактальный анализ представляет собой метод оценки фрактальных характеристик данных. Нейронную сеть можно представить в виде графа, где каждый нейрон соответствует вершине. Применение фрактального сжатия позволяет выявить "лотерейные билеты" в нейронной сети. Выбор этого инструмента обусловлен постановкой задачи и идеей самоподобия между исходной и найденной нейросетью.

Фрактальная математика, формализованная Бенуа Мандельбротом, является относительно новым разделом математики. Несмотря на свою молодость она уже привнесла значительные открытия. Например, фрактальный анализ помог открыть тайну возникновения турбулентности, явления, которое характеризуется образованием многочисленных нелинейных фрактальных волн и линейных волн различных размеров при движении жидкости или газа. Кроме того, фракталы применяются для анализа временных рядов.

Объект исследования — методы и алгоритмы фрактального сжатия для прунинга нейронных сетей.

Предмет исследования — программный код для нахождения «лотерейных билетов» методом фрактального анализа.

Цель выпускной квалификационной работы (ВКР) — разработка алгоритма для нахождения «лотерейных билетов» методом фрактального анализа, который будет предоставлять возможность оптимального

использования ресурсов как по памяти, так и по времени, при этом погрешность ответов допустима не более 5% относительно исходной модели.

Для достижения поставленной цели необходимо решить следующие задачи:

1. собрать теоретическую базу для разрешения гипотезы лотерейного билета;

2. проанализировать эти методы;

3. реализовать алгоритм фрактального сжатия;

4. имплементировать данный алгоритм в прунер;

5. провести бенчмаркинг.

Методы исследования: анализ математической базы; анализ предпосылок и проблем нахождения «лотерейных билетов»; реализация соответствующих алгоритмов фрактального прореживания и бенчмаркинг. Практическая значимость полученных результатов состоит в том, что они могут быть использованы для оптимизации нейронных сетей.

Структура работы: ВКР состоит из введения, трех глав, заключения, списка литературы и приложения.

В первой главе изложена теоретическая основа искусственного интеллекта, алгоритма обучения нейронных сетей, а также составляющая техник поиска «лотерейного билета».

Во второй главе реализован алгоритм фрактального прунинга для плотной нейронной сети.

В третьей главе освещены другие методы оптимизации, а также приведена оценка эффективности и бенчмаркинг.

В заключении приведены выводы по результатам проделанной работы.

Глава 1. Теоретические основы гипотезы лотерейного билета

1.1 История искусственного интеллекта

Цель данной главы — осветить ключевые исторические события, связанные с развитием искусственного интеллекта, исключая широко известные факты: от использования абака до появления первой алгоритмической программы.

В 1935 году Аланом Тьюрингом была представлена концепция абстрактной вычислительной машины [30], основанной на идее безграничной памяти и сканера, который перемещается по ней, считывая и записывая символы. Работа сканера регулируется набором инструкций, хранящимся в памяти. Такая модель, известная как хранимая программа Тьюринга, предполагает автономную работу машины. Это послужило базовой концепцией и заложило архитектурную идею искусственного интеллекта [2]. Также Тьюрингом был предложен тест, благодаря которому можно распознать, является собеседник человеком или нет.

Модель Мак-Каллока-Питтса [23] появилась в 1943 году и сразу стала точкой отсчета в развитии искусственных нейронных сетей. На первых этапах исследователи выделяли два основных подхода: монотипический и генотипический. При монотипическом подходе параметры нейронов задаются на начальном этапе с фиксированной топологией сети; а при генотипическом параметры нейронов определены лишь частично.

При использовании генотипического подхода, инициализируется не одна определенная сеть, а задается совокупность возможных. Основное отличие генотипического подхода заключается в наличии механизма обучения, который позволяет выбирать конкретные параметры сети [11].

Данные нейрофизиологии подкрепляли идею передачи информации в мозге с использованием коротких импульсов — спайков. Это привело Мак-Каллока и Питтса к отождествлению человеческого мозга с цифровой машиной, оперирующей двоичными сигналами.

Исследования не включали в себя методы обучения нейронных сетей. Первооткрыватели лишь разработали математическую модель нейронов и продемонстрировали их потенциал. Ученые не предприняли формальных усилий для доказательства того, что их модель является тьюринг-полной, так как в то время это считалось очевидным. Но проблема обучения нейронных сетей активно обсуждалась в академических кругах, и первые шаги в этом направлении были предприняты учениками Мак-Каллока.

В свою очередь, нейронную сеть можно использовать и без обучения, основываясь на определенных правилах. Однако возможность самообучения, характерная для человеческого мозга, привела к созданию машин, обладающих подобным свойством.

В 1948 году ученики Мак-Каллока (Анатоль Рапопорт и Альфонсо Шимбел из Чикагского университета) представили концепцию статистически организованных сетей [17]. Они рассматривали параметры нейронов и их компонентов как случайные величины с определенными вероятностными распределениями. На основе этого было выведено общее уравнение для оценки вероятности активации нейрона.

Дональд Хебб в 1950 году впервые опубликовал исследование, в котором был сформулирован известный принцип: "Если аксон клетки А часто и близко связывается с клеткой В, то это приводит к изменениям в их росте или метаболизме, улучшая способность клетки А вызывать возбуждение в клетке В" [10].

Идеи Хебба для обучения нейронных сетей послужили основой при разработке практических алгоритмов. А последователи Хебба уже начали обучать нейронные сети по его принципу [6], но их работы не получили должного признания со стороны научного сообщества.

В ходе дальнейших исследований, проводимых М. Мински с его соавторами, был разработан метод под названием стохастический нейронный аналоговый калькулятор с подкреплением [12]. Он включал создание "электронной крысы" для навигации в электронном лабиринте с целью

нахождения выхода. Модель крысы состояла из случайного соединения сорока искусственных нейронов, построенных на базе нескольких электронных ламп и двигателя. Такое устройство стало первой самообучающейся системой.

Позже Фрэнк Розенблатт собрал и систематизировал различные знания в области архитектур и алгоритмов обучения [16]. Исследования он проводил в Авиационной лаборатории Корнеллского университета. В начале 1957 года лаборатория опубликовала работу "Перцептрон: воспринимающий и распознающий автомат", где основное внимание уделялось вероятностному подходу, а не детерминистскому, как это было ранее.

Данный перцептрон был представлен системой, которая включает в себя три слоя нейронов: сенсорный (S), ассоциативный (A) и реагирующий (R). Обучаемые веса присутствуют только между ассоциативным и реагирующим слоями нейронов (A → R) и корректируются в ходе обучения. Слой синаптических связей S → A также имеет веса, которые не меняются в процессе обучения и могут быть настроены вручную. Значения этих весов бинарные и могут быть либо "1", либо "-1", что соответствует возбуждению и торможению синапсов. Иногда существовали синапсы абсолютного торможения, веса которых могли быть установлены на минус бесконечность, что, вероятно, отражает применяемый ранее подход Мак-Каллока и Питтса.

В 1959 году исследователь в области искусственного интеллекта и изобретатель первой самообучающейся компьютерной программы игры в шашки, Артур Самуэль, ввел термин "машинное обучение" в научный контекст [3].

Машинное обучение представляет собой методы в области искусственного интеллекта, нацеленные на разработку алгоритмов и статистических моделей, способных улучшать производительность компьютерных систем путем использования опыта при решении конкретных задач. Алгоритмы машинного обучения нацелены на распознание закономерностей в больших объемах данных. Благодаря информации об их внутреннем устройстве создаются статистические методы для прогнозирования, обнаружения аномалий и

построения автоматических ансамблей для решения типовых задач, но с другим распределением. Основной принцип машинного обучения заключается в том, чтобы дать компьютерам возможность извлекать знания из данных без явного программирования. Для этого компьютерные системы обучаются на основе данных с примерами задач и правильными ответами. После процесса обучения модель может быть использована для прогнозирования или принятия решений на основе новых данных.

Несмотря на то, что глубокие нейронные сети были представлены еще в 1980-х годах [24], значительное развитие и распространение глубинного обучения началось только в XXI веке. Предполагалось, что добавление большего количества слоев в нейронную сеть может повысить ее эффективность, так как это позволит создавать более сложные представления. Однако ключевое значение имеет метод обучения сети. Для глубинного обучения применялись алгоритмы, используемые для обучения обычных искусственных нейронных сетей, такие как метод обратного распространения ошибки. Но этот метод оказался эффективным только для обучения последних слоев сети, что замедляло процесс и не позволяло эффективно функционировать скрытым слоям глубокой нейронной сети.

Три группы ученых в 2006 году смогли преодолеть трудности, связанные с обучением нейронных сетей. Джеффри Хинтон предложил использовать машину Больцмана для предварительного обучения сетей, обучая каждый слой отдельно. Ян ЛеКун предложил сверточную нейронную сеть для классификации изображений, состоящую из сверточных и пулинговых слоев. Иошуа Бенджио разработал каскадный автоэнкодер, который позволял эффективно использовать все слои в глубокой нейронной сети.

Следующим важным этапом стали трансформеры — мощные модели глубокого обучения, которые были разработаны с целью эффективного решения задач обработки естественного языка. Их история началась в 2017 году, когда была представлена архитектура Transformer [19] исследователями из Google.

Данная модель основана на механизме внимания, который позволяет ей эффективно обрабатывать последовательности данных.

Основным преимуществом трансформеров является возможность параллельной обработки входных данных, что значительно ускоряет процесс обучения и повышает качество прогнозирования. Архитектура трансформеров состоит из нескольких слоев кодировщика и декодировщика, каждый из которых содержит множество механизмов внимания.

С течением времени трансформеры стали широко используемыми моделями для решения различных задач в области естественного языка, таких как машинный перевод, распознавание речи, сжатие текста. Благодаря своей эффективности и гибкости они стали одними из лучших инструментов для работы с естественным языком, превзойдя многие традиционные методы обработки текста. Впоследствии данная архитектура получила распространение и в других доменах, став state-of-the-art во многих задачах компьютерного зрения.

История развития областей искусственного интеллекта изображена на диаграмме (рисунок 3).

Рисунок 3. Диаграмма Венна областей искусственного интеллекта

1.2 Описание алгоритма обучения нейронных сетей

Цель данной главы заключается в изложении ключевых аспектов обучения нейронных сетей. В данной работе особое внимание уделяется обучению с учителем, а именно задачам классификации. Дальнейшие выкладки будут проводиться с учетом данной предпосылки и дополнительно рассмотрены конкретные алгоритмы и функции, которые используются в ходе реализации алгоритма фрактального прунинга.

Для дальнейшей работы необходимо определить функцию потерь, функцию активации, оптимизатор и метрику.

Функция потерь (или функция ошибки) — это математическая функция, которая измеряет разницу между предсказанными значениями модели и фактическими значениями данных. Она используется в машинном обучении для оптимизации параметров модели путем минимизации этой разницы.

Основой для оптимизации любой функции является производная, которая отражает наибольший рост функции. Обучение сводится к поиску более минимального из всех локальных минимумов, но многое зависит от начальной инициализации (рисунок 4).

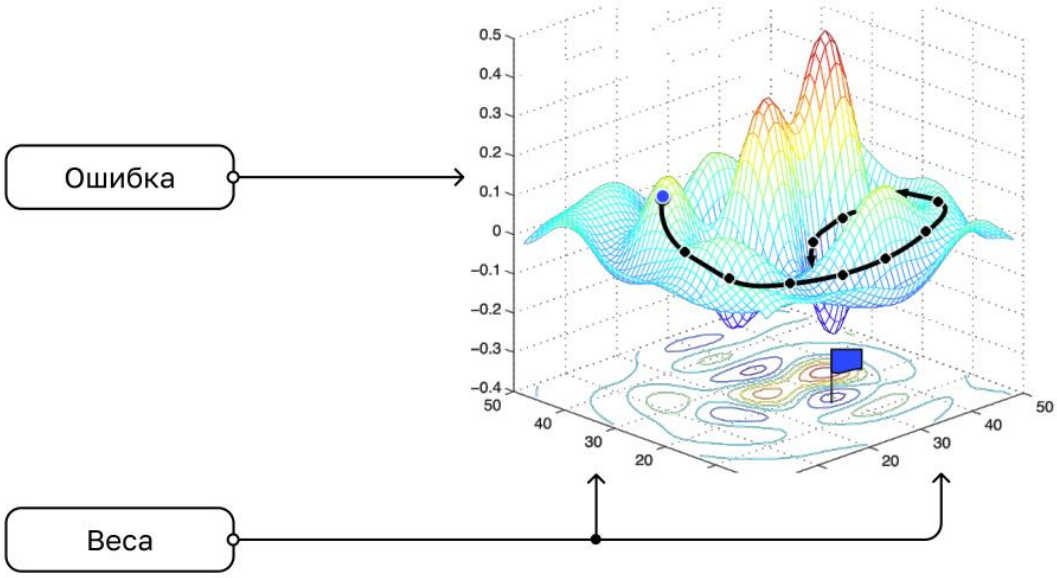

Рисунок 4. Демонстрация нахождения локального минимума для трехмерного случая

В данной работе предлагается использовать кросс-энтропию [4] в качестве функции потерь. Для набора данных с N экземплярами потери кросс-энтропия рассчитывается следующим образом:

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}(y_{i,j}\cdot\log(p_{i,j})),$$ где:

- C — количество классов;
- $y_{i,j}$ — истинные метки класса j для экземпляра i;
- $p_{i,j}$ — предсказанные метки класса j для экземпляра i.

Для достижения минимума применяется оптимизатор. В алгоритме будет использоваться Adam как наиболее популярный.

Adam (сокращение от Adaptive Moment Estimation) представляет собой распространенный алгоритм оптимизации [1], применяемый для моделей глубокого обучения. Ниже приведен алгоритм работы Adam.

Инициализация: Adam определяет два параметра: m и v, которые представляют собой векторы, хранящие экспоненциально убывающее среднее прошлых градиентов и квадраты градиентов. Эти параметры изначально устанавливаются равными нулю.

Вычисление градиентов: на каждой итерации обучения вычисляются градиенты функции потерь по отношению к параметрам модели с использованием методов, таких как обратное распространение ошибки.

Обновление параметров: Adam определяет адаптивную скорость обучения на текущем шаге для каждого параметра, используя экспоненциальные средние первого и второго моментов градиентов (m и v). Параметры обновляются по формуле:

$$\theta_{t+1}=\theta_t-\frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t,$$ где:

- θ_t — параметры на временном шаге t;
- η — скорость обучения;
- ϵ — небольшая константа, добавляемая в знаменатель для числовой стабильности;

- $\widehat{m}_t$ и $\hat{v}_t$ — оценки первого и второго моментов градиента с поправками на смещение.

Повторение: шаги 2 и 3 повторяются до сходимости или определенного числа итераций.

Адаптивные скорость обучения и момент позволяют алгоритму быстро и эффективно сходиться, что делает его популярным выбором при работе с различными задачами глубокого обучения.

Метрика — это оценка качества выхода модели. В данном случае выбрана точность (accuracy). Это отношение правильно угаданных ответов ко всем элементам выборки.

Необходимо отметить, как происходит обучение нейронной сети. Она работает с большим объемом данных. Из-за чего может возникнуть ряд трудностей: например, не хватит памяти графического ускорителя, чтобы в нее поместить все данные, или нейросеть будет долго учиться. Для оптимизации данного процесса существует пакетная обработка данных (батч). У каждого пакета сушествует единый размер (batch_size). Необходимо определить количество этих пакетов, и сколько раз (эпох) их нужно показывать нейросети.

Например, мы разбили данные по батчам, где в каждой эпохе по 4 таких пакета (рисунок 5), затем случайным образом выбираем 4 батча.

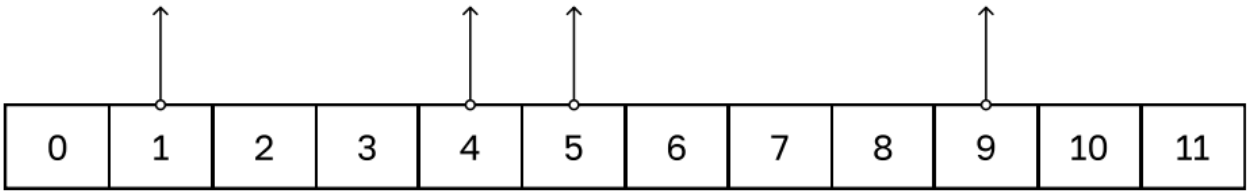

Рисунок 5. Первая эпоха выбора батчей

На следующей эпохе видно, что вероятность выбора четырех совершенно новых батчей меньше по сравнению с прошлым разом (рисунок 6). Необходимо итеративно пройтись по данным, пока не кончатся эпохи. Эти блоки не удаляются навсегда. Лишь решается вопрос с нехваткой данных для качественного обучения. Важно балансировать с тем, сколько раз пакет попадает в выборку для обучения нейросети во избежание ее переобучения.

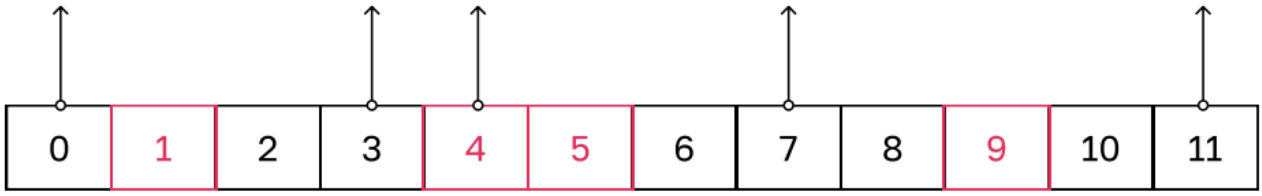

Рисунок 6. Вторая эпоха выбора батчей

Обучение нейронной сети состоит из трех основных этапов:

1. прямой ход: вычисление предсказаний и подсчет ошибки;
2. обратный ход: подсчет градиента ошибки;
3. обновление весов.

Осталось определить функцию активации. Она регулирует выходное значение нейрона на основе результата взвешенной суммы входных сигналов и порогового значения.

В данной работе используется функция активации ReLU (Rectified Linear Unit) [29], определяемая как $max(0, x)$. Функция ReLU возвращает значение x, если оно положительное, и 0 — в противном случае (рисунок 7). Хотя ReLU линейна в первом квадранте, она обладает нелинейными свойствами и может быть использована для аппроксимации любой функции. Применение ReLU позволяет строить слои нейронных сетей. Область значений ReLU находится в интервале $[0, +\infty)$, что может вызвать "взрыв" активаций.

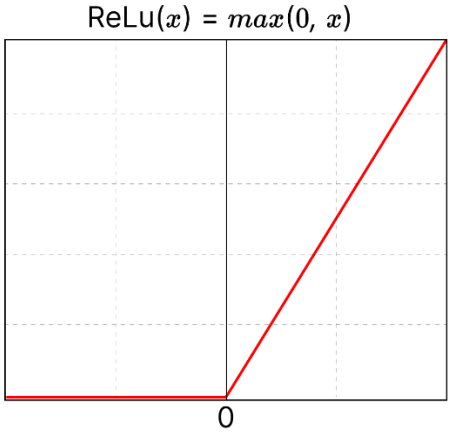

Рисунок 7. График функции активации ReLU

1.3 Гипотеза лотерейного билета и ее проблематика

Благодаря способности обнаруживать сложные связи в данных глубокие нейронные сети достигают значительных успехов в различных областях исследований. Однако высокое качество работы нейронных сетей часто требует значительных вычислительных ресурсов. В связи с этим много исследований нацелено на разработку методов сжатия нейронных сетей, которые позволяют уменьшить их размеры без существенной потери качества. Такие методы включают в себя прунинг, квантизацию, матричное и тензорное разложение, тензорные поезда, дистилляцию знаний.

На практике применяется широко распространенная методология, основанная на использовании предварительно обученных нейронных сетей и методов прунинга, опирающихся на критерии значимости весов, с последующим дообучением разреженной модели или исключением весов. Существуют также подходы к обучению разреженной модели "с нуля", при которых модель постоянно поддерживается в состоянии разреженности с изменениями в выборе уменьшенных весов в течение времени.

Эмпирические данные свидетельствуют о том, что размер модели является важным фактором, так как небольшие нейронные сети часто не способны достичь такого же уровня качества, как и их более крупные аналоги. Данное явление объясняется процессом оптимизации: более масштабные модели обладают большей гибкостью и потенциалом для оптимальной настройки параметров под конкретные задачи.

Разреженная нейронная сеть обладает более низким количеством параметров по сравнению с плотной, но сохраняет достаточное качество выхода. Поэтому в структуре сети теоретически существует подсеть, способная достичь тех же результатов.

В ходе экспериментов исследователи впервые обнаружили указанное явление при тренировке нескольких полностью связанных сверточных моделей на наборах данных MNIST и CIFAR10 [8]. Они иллюстрировали наличие

подсетей, способных обучаться с одинаковой эффективностью, а иногда и лучше, чем исходная плотная сеть при одинаковых параметрах процесса обучения (рисунок 8). Этот феномен можно сравнить с ситуацией, при которой эти подсети были бы проинициализированы случайным образом, будто выиграна инициализация весов в лотерее.

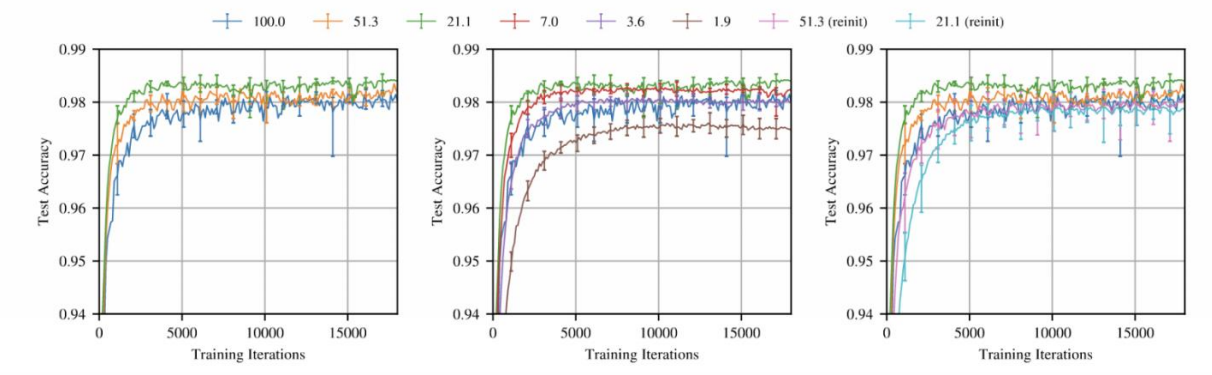

Рисунок 8. Качество на тестовой выборке для исходной модели
и лотерейных билетов с разными степенями прореженности

Гипотеза: в случайно инициализированной плотной нейронной сети существует подсеть, способная достичь того же качества тестовых данных, что и исходная сеть; за то же или меньшее количество итераций, что и исходная плотная сеть.

Авторы статьи предлагают применять итеративный метод уменьшения размера модели [7], начиная с плотно инициализированной сети. Нейронная сеть обучается до достижения сходимости. Далее заданная часть наименьших весов удаляется, что порождает бинарную маску $M^{(1)}$. Эта маска затем фиксируется, и веса восстанавливаются до своей начальной инициализации. Процесс обучения повторяется для обрезанной сети. Удаление еще одной части весов позволяет получить маску $M^{(2)}$, после чего веса реинициализируются. Этот цикл продолжается в течение ряда итераций до достижения требуемого уровня разреженности модели (рисунок 9).

Существует несколько ключевых аспектов, которые следует учитывать.

1. Маска связана с исходной инициализацией весов. После прунинга, если ненулевые веса случайным образом инициализировать, то разреженная сеть может потерять свои характеристики.
2. При одинаковых параметрах обучения разреженные сети сходятся еще быстрее, чем исходные плотные сети (зеленая кривая, соответствующая ~20% ненулевых весов, достигает наивысшего качества). Этот эффект объясняется тем, что такие сети менее склонны к переобучению. Однако данное свойство характерно только для конкретного выбора маски, что маловероятно при случайном отборе. Поэтому данный эффект аналогичен лотерее, где выигрыш возможен, но вероятность мала.
3. Для более сложных сетей, чем LeNet, аккуратная настройка learning rate становится критическим этапом, требующим точной настройки и введения warmup (постепенного увеличения скорости обучения с 0 до максимального значения).
4. Несмотря на то, что генерация "лотерейного билета" за одну итерацию прореживания выполнима, итеративная процедура приводит к созданию более оптимальных масок (синие кривые превосходят зеленые).

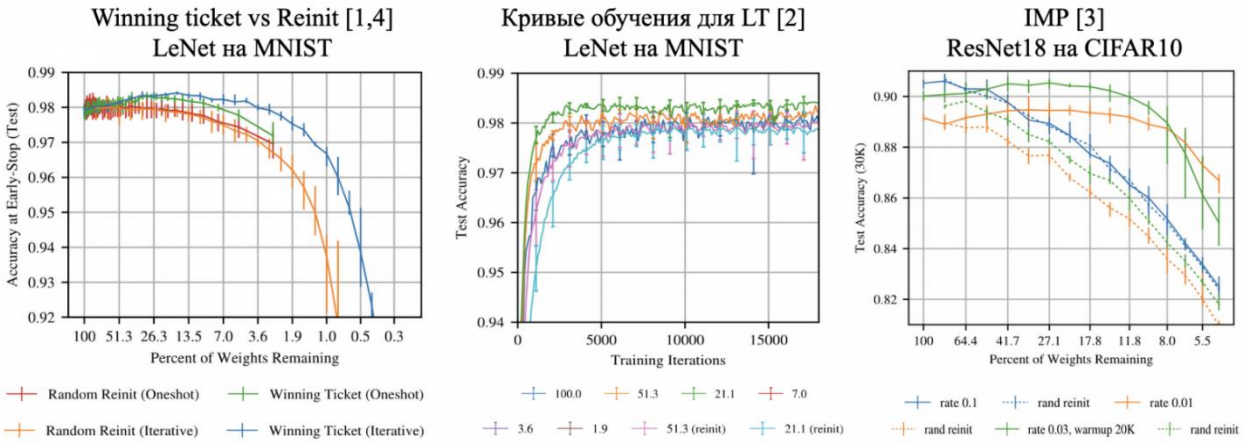

Рисунок 9. Иллюстрации к тезисам [1-4] Из статьи [8]

Основной вывод заключается в возможности обнаружения в нейронной сети подсети, которая способна достичь не только сопоставимого качества с

более крупной сетью, но и быть обученной до этого уровня. Однако поиск такой подсети представляет собой ресурсоемкий и затратный процесс, требующий вычислительных ресурсов в несколько раз больше, чем обучение исходной плотной сети.

В рамках данной работы предполагается применение сети LeNet [22] и набора данных MNIST для сравнительного анализа результатов.

Разработанная в 1994 году сеть LeNet стала одним из первых успешных примеров сверточных нейронных сетей, заложивших основы для развития глубокого обучения.

Использование обучаемых сверточных фильтров обеспечило эффективное извлечение схожих объектов из различных областей изображения при помощи небольшого количества параметров. Ранее ни видеокарты, ни центральные процессоры не могли ускорить процесс обучения из-за их низкой производительности. Основной преимущественной чертой такой архитектуры была возможность сохранения параметров и результатов вычислений, в отличие от использования отдельных пикселей как независимых входных данных для глубокой многослойной нейронной сети. В структуре LeNet первый слой не использует пиксели, так как изображения обычно имеют сильную пространственную корреляцию. Использование пикселей в качестве отдельных входных данных не учитывало бы эти корреляционные свойства.

Сеть глубокого обучения, включающая трехслойную структуру, состоящую из сверточных слоев, слоев пулинга и нелинейных слоев, стала значимым компонентом в области глубокого обучения, применяемого для обработки изображений.

Данная архитектура основывается на методах свертки для извлечения пространственных особенностей; подвыборке с использованием пространственного усреднения объектов; нелинейности, включающей гиперболические тангенсоидальные или сигмоидальные функции.

Набор данных MNIST [5] является широко используемым в машинном и глубоком обучении для задач классификации рукописных цифр. Он содержит

изображения размером 32x32 пикселей, каждое из которых представляет собой рукописную цифру от 0 до 9. Всего в наборе данных MNIST содержится 60 000 изображений для обучения и 10 000 изображений для тестирования. Набор данных MNIST является стандартным и широко применяемым для проверки и сравнения различных алгоритмов машинного обучения и нейронных сетей.

Глава 2. Разработка алгоритма фрактального прунинга

2.1 Формализация фрактального сжатия. Моделирование на графах

Для понимания работы алгоритма фрактального прунинга рассматривается сжимающее отображение.

Предположим, имеется (E, d) — полное метрическое пространство, а $f: E \to E$. Тогда f называют сжимающим отображением [28], если:

$$\exists s, 0 < s < 1 \to \forall x, y \in E, \mathrm{d}\big(f(x), f(y)\big) \leq s\,\mathrm{d}(x, y),$$

а s называют сжимающим коэффициентом.

Приведу формулировки, необходимые для работы данного алгоритма.

1. Теорема Банаха о неподвижной точке: f имеет уникальную неподвижную точку x_0.
2. Теорема коллажа: если $d\big(x, f(x)\big) < \xi$, то $d(x, x_0) < \frac{\xi}{1-s}$. Иными словами, если найдется такое сжатие f, что $f(x)$ близко к x, то неподвижная точка f тоже находится близко к x.
3. Лемма: если f_i являются сжимающими отображениями, то и f тоже сжимающее отображение.

Зададим множество матриц смежности графа нейронов и расстояние. E — это множество матриц с h строками и w столбцами, а также коэффициент сжатия s. Определим расстояние, полученное из нормы Фробениуса:

$$d(x, y) = \left(\sum_{i=1}^{h} \sum_{j=1}^{w} \big(x_{ij} - y_{ij}\big)^2\right)^{\frac{1}{2}}.$$

Допустим, $x \in E$ — это сжимаемая матрица. Дважды разделим ее на блоки:

- Сначала происходит разделение на конечные или интервальные блоки $M_i,\ i = \overline{1, m}$. Эти блоки образуют покрывающее исходную матрицу разбиение.
- Затем матрица разделяется на блоки источников или доменов $D_i,\ i = \overline{1, d}$. Эти блоки могут быть неравномерно разделены и не обязательно охватывают все изображение.

Фрактальное сжатие будет реализовано на языке программирования Python. На первом этапе примером послужит граф как упрощенная модель нейронной сети.

В теории случайных графов заложена концепция вероятностных распределений.

Одним из распространенных методов генерации случайных графов является модель Эрдёша-Реньи [21]. Граф $G(n, M)$ случайным образом выбирается из всех возможных графов с n вершинами и M ребрами путем случайного соединения узлов. Каждое ребро включается в граф с вероятностью p независимо от предыдущего выбора. Все графы с n узлами и M ребрами имеют одинаковую вероятность $p^M(1-p)^{C_2^n - M}$.

Параметр p в данной модели представляет собой вероятность возникновения связи между вершинами графа. Данная модель не предусматривает формирование петель, т.е. отсутствует возможность соединения вершины с самой собой.

Предложенная модель наилучшим образом соответствует критерию, который предполагает, что нейронная сеть должна быть случайной и плотной (допустимые значения параметра $p \geq 0.5$).

Например, так выглядит изоморфизм графа одной из реализаций (рисунок 10).

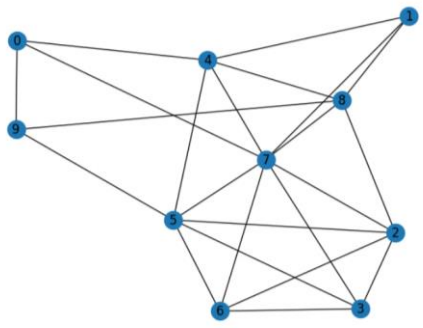

Рисунок 10. Случайная реализация аналога плотной нейронной сети

В данной работе определено следующее сжимающее отображение:

$f_i(x) = s \times rotate_\alpha\left(flip_d\left(reduce(x)\right)\right) + b$, в котором

- $rotate_{\alpha}$ — функция перехода к доменам;
- $flip_{d}$ и $reduce$ — аффинное преобразование;
- s отвечает за контрастность;
- b изменяет яркость.

После процесса сжатия будут получены несколько вариантов, лучшим из которых станет тот, у которого энтропия максимальна. Тем самым решится проблема, возникающая при использовании очень плотных и полносвязных нейронных сетей, в которых множественные возвращаемые отображения имеют низкое разнообразие.

Следует отметить, что в терминах нейронных сетей удаление нейрона подразумевает обнуление его веса. Если нейрон покидает порог активации, то его вес остается без изменений, в противном случае он обнуляется.

В данной ВКР ключевым инструментом будет библиотека PyTorch [14], которая принадлежит к числу наиболее распространенных и мощных библиотек для реализации нейронных сетей. Этот фреймворк обладает рядом преимуществ.

1. Динамичная природа вычислительного графа библиотеки PyTorch облегчает создание сложных моделей с изменяющейся структурой, что приводит к более гибким и удобным процессам разработки и отладки моделей.
2. В PyTorch доступен широкий ассортимент готовых реализаций различных слоев. Необходимо лишь собрать эти слои, подобно конструктору, не задумываясь о деталях внутреннего устройства.
3. Фреймворк PyTorch предоставляет средства оптимизации и параллелизации вычислений, что значительно повышает эффективность использования вычислительных ресурсов и ускоряет процесс обучения моделей.
4. Хорошо документированный код PyTorch облегчает понимание и работу с функциональностью библиотеки.

2.2 Описание базисных блоков архитектуры нейронной сети LeNet

В работе будет использоваться модификация архитектуры LeNet (см. Приложение 4), состоящая из двух сверток, двух пулингов и трех полносвязных слоев (рисунок 11).

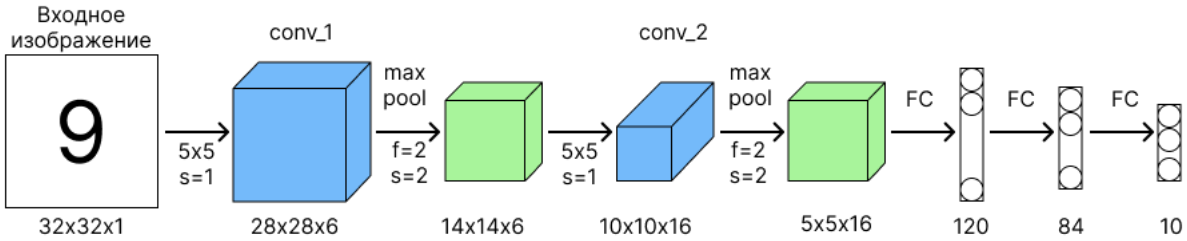

Рисунок 11. Архитектура LeNet

Работа ведется с изображениями входного формата, состоящего из упорядоченных пикселей, каждый из которых формируется из трех "каналов": красного, зеленого и синего цветов [27]. Значения интенсивности находятся в диапазоне от 0 до 255 для умещения в 8-битный формат.

Ниже рассмотрены все структурные блоки архитектуры. Первым блоком станут свертки, главной проблемой которых является необходимость обеспечения инвариантности к смещению в представлении обучающих данных.

Положение представителя класса на изображении может значительно варьироваться, и, следовательно, сложно установить его точное место на изображении, в котором модель дает наилучшие результаты в распознавании классов. Для достижения точных предсказаний целесообразно произвести смещение изображения на все возможные координаты, при этом пустоты заполняются нулями (рисунок 12).

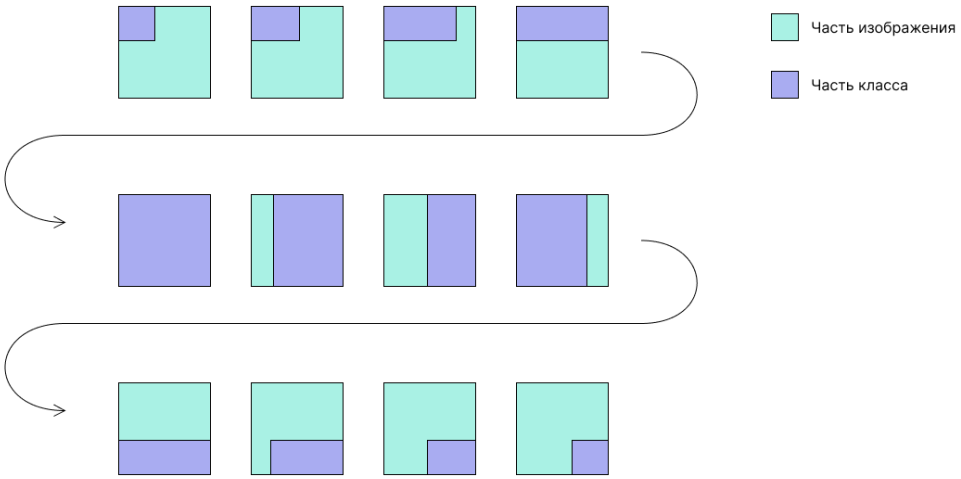

Рисунок 12. Демонстрация последовательного смещения изображения

Далее для каждого смещения будут сделаны прогнозы вероятности наличия цифры на изображении. После этого полученные прогнозы могут быть агрегированы различными способами: например, вычислением среднего или максимального значения (pooling).

Упрощенная модель состоит из ансамбля линейных операторов. Каждое смещенное изображение разворачивается в вектор и умножается на один и тот же вектор весов для всех сдвигов. Это позволяет получить линейный оператор, известный как свертка.

Ядро свертки представляет собой веса, упорядоченные в тензоре. Окно свертки — это область изображения, которую обрабатывает свертка. В двумерных свертках окно перемещается по двум измерениям изображения, учитывая все цветовые каналы (рисунок 13).

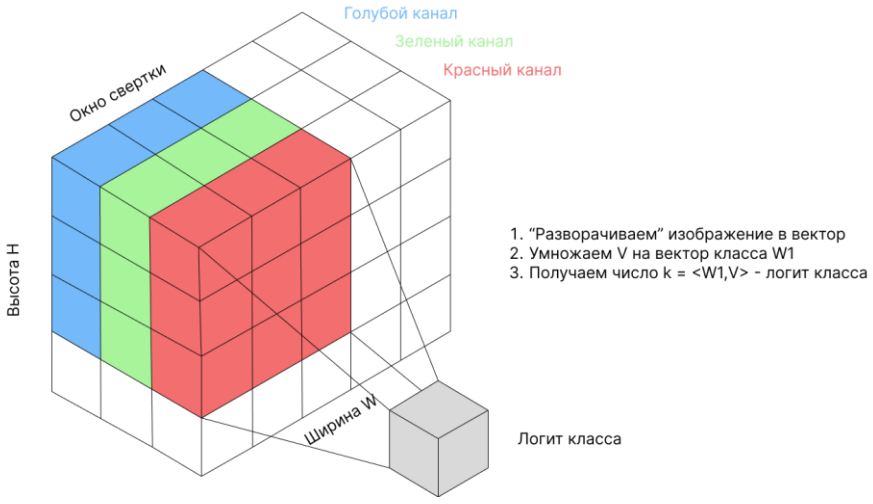

Рисунок 13. Демонстрация работы свертки

Но данный подход создает новую проблему: предсказание для определенного окна не учитывает контекст за его пределами. Это означает, что приемлемые результаты могут быть получены только том в случае, если изучаемый объект имеет характеристики, соответствующие окну свертки, или если объекты заметно отличаются по текстуре [27].

Область изображения, на которую направлена свертка, называется рецептивным полем. Для увеличения его размера (без увеличения размера ядра свертки) применяется техника стекинга слоев. Она позволяет улучшить представление рецептивного поля. Нейросеть включает в себя один слой, выполняющий классификацию. Можно начать с создания C_1 различных сверток с фильтрами размером $k \times k$ на первом этапе. Результаты каждой свертки организовываются в виде нового изображения. Далее эти результаты объединяются в трехмерный тензор размером $H \times W \times C_1$, так называемую, карту признаков. Применение нелинейной функции к полученному тензору и использование K новых сверток позволит получить предсказания для каждого пикселя. Таким образом, размер рецептивного поля для конечных нейронов увеличивается с $k \times k$ до $(2k-1) \times (2k-1)$. Повторение данной операции позволит конечным нейронам обрабатывать почти всю необходимую информацию для эффективного прогнозирования. Такой подход уменьшает количество параметров и снижает вычислительную сложность по сравнению с использованием одной большой полносвязной сети [27].

При рассмотрении первой свертки $(X \rightarrow L_1)$, размер рецептивного поля соответствует размеру окна, равному 3. Рассмотрим вторую свертку $(L_1 \rightarrow L_2)$. В вычислениях этой свертки участвуют пиксели из квадрата 3x3, при этом каждый из них был получен с использованием предыдущей свертки $(X \rightarrow L_1)$. Таким образом, рецептивное поле композиции сверток $(X \rightarrow L_1 \rightarrow L_2)$ является объединением рецептивных полей свертки $(X \rightarrow L_1)$ для всех пикселей из окна свертки $(L_1 \rightarrow L_2)$, образуя новое рецептивное поле размером 5x5. Аналогичные рассуждения могут быть продолжены и для всех последующих сверток [27].

Ниже представлено формальное определение свертки (рисунок 14).

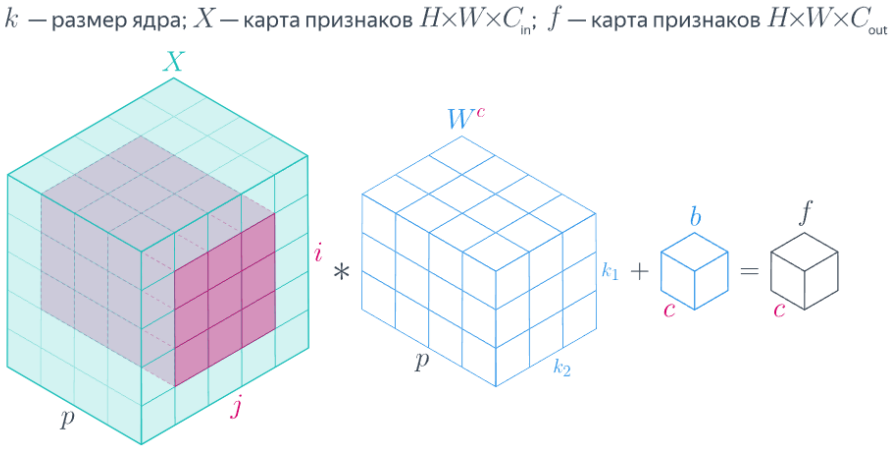

Рисунок 14. Формальное определение свертки

$$f_{ijc} = \sum_{p=1}^{C_{in}} \sum_{k_1=-\lfloor \frac{k}{2} \rfloor}^{\frac{k}{2}} \sum_{k_2=-\lfloor \frac{k}{2} \rfloor}^{\frac{k}{2}} W^{c}_{\lfloor \frac{k}{2} \rfloor+1+k_1, \lfloor \frac{k}{2} \rfloor+1+k_2} p^{X_{i+k_1, j+k_2, p}+b_c}$$

Параметры сверточного слоя включают [27]:

- Число признаков, определяющее количество фильтров в слое нейронной сети.
- Размер фильтров, который определяется высотой и шириной тензора фильтров и обычно является нечетным числом. Часто используются фильтры размером 3x3 или 5x5.
- Шаг свертки, указывающий на количество пикселей, на которое фильтр смещается при применении к входному изображению. При шаге 1 фильтр сдвигается на один пиксель за один шаг, при шаге 2 — на два пикселя. Увеличение значения шага приводит к уменьшению размеров карты признаков на выходе слоя.
- Дополнение нулями представляет собой количество пикселей, добавляемых с каждой стороны изображения перед применением фильтров. Это предотвращает уменьшение изображения до размера фильтра, так как фильтр применяется только в тех областях, где под каждым значением фильтра имеется значение в исходном изображении.

В сверточном слое нейронной сети обучаются только параметры фильтров и смещения. Благодаря тому, что фильтр, применяемый к изображению, остается неизменным в процессе его сканирования, количество обучаемых параметров значительно меньше по сравнению с полносвязными слоями сети.

Рассмотрим прохождение градиентов через сверточный слой, используя пример одномерной свертки с одним входным каналом, ядром длиной 3 и с дополнением нулей по бокам. Эту операцию можно представить в виде матричного умножения:

$$(x_1, \cdots x_d) \times (w_{-1}, w_0, w_1)$$

Обозначим последнюю матрицу через $\widehat{W}$, а ядро свертки через W. Градиент по весам равен:

$$\nabla_{x_0} \mathcal{L} = \nabla_{x_0 * w} \mathcal{L} \widehat{W}^T$$

Матрица $\widehat{W}^T$ тоже соответствует свертке, только:

- с симметричным исходному ядром (w_{-1}, w_0, w_1);
- с дополнением вектора $\nabla_{x_0 * w}$ нулями (это как раз соответствует неполным столбцам: можно считать, что «выходящие» за границы матрицы и отсутствующие в ней элементы умножаются на нули).

Для создания новых карт признаков на изображении в последовательном сверточном слое используются C сверток. Чем больше таких карт, тем лучше для обучения модели различным закономерностям. Однако при работе с высокоразрешенными изображениями это может привести к избыточному количеству параметров. Один из способов решения этой проблемы — применение нескольких сверток с C_1 каналами; затем уменьшение размера карты признаков вдвое; и одновременное увеличение количества сверток вдвое.

Необходимо также рассмотреть способы снижения разрешения изображения. Один из простых методов заключается в выборе всех пикселей с нечетными индексами. Хотя этот подход и рабочий, но он может привести к нежелательной потере информации. Существует несколько альтернативных способов: например, можно использовать среднее или максимальное значение в

окне 2x2 с шагом 2 по карте признаков. Эксперименты показали, что выбор максимального значения часто приводит к хорошим результатам и широко применяется в различных архитектурах. Следует отметить, что максимальное значение вычисляется для каждого канала независимо.

Один из плюсов такого метода заключается в расширении рецептивного поля. Это позволяет увеличить его размер в два раза. Процесс уменьшения разрешения с помощью выбора максимального значения в окне называется max pooling; а с использованием среднего значения — average pooling.

2.3 Описание работы фрактального прунинга

Современные методы глубокого обучения опираются на избыточно параметризованные модели, которые сложно внедрить в бизнес-процессы [13]. В свою очередь, биологические нейронные сети известны своим эффективным разреженным соединением. Идентификация оптимальных техник сжатия моделей путем сокращения количества параметров в них важна для уменьшения использования памяти, энергии и аппаратных ресурсов без ущерба для точности. Это, в свою очередь, позволяет развертывать легковесные модели на устройствах, обеспечивать конфиденциальность с помощью частных вычислений на устройстве. В исследованиях прунинг используется для изучения различий в динамике обучения между избыточно параметризованными и недостаточно параметризованными сетями для изучения роли удачных разреженных подсетей и инициализаций ("лотерейных билетов") как техники поиска деструктивной архитектуры нейросети.

Рассмотрим простую реализацию прунинга (см. Приложение 1). Для этого определим простую архитектуру, состоящую только из двух линейных слоев.

Определим функцию прунинга на весах, где зададим тип атрибута, который будет забираться у модели. Тут же инициализируем маску, по которой будут логически отбираться элементы матрицы весов по порогу, например, 20% от максимального значения. Как правило, берется значение перцентиля. Далее

отбираются те индексы в матрице весов, которые меньше порога; другие же обращаются в 0.

В библиотеке PyTorch есть внутренняя реализация данного метода. Но для исследовательских проектов, где нужна более сложная логика формирования маски или стратегии прунинга, следует использовать собственные программные модули, которые закрывают потребность.

Разберем программный код фрактального сжатия (см. Приложение 2).

В функцию reduce на вход поступают два параметра: матрица весов и фактор сжатия; результатом выхода будет двумерный массив. Принцип работы заключается в том, что создается новая матрица result заданного размера, в которую затем записываются средние значения элементов исходной матрицы matrix, усредненные по блокам размера factor x factor. Для этого используются два цикла для перебора всех блоков и вычисления среднего значения в каждом из них.

Выходом функции rotate будет матрица, повернутая на определенный угол. На вход поступает матрица и угол поворота. Операция производится соответствующей функцией пакета numpy.

Для того, чтобы поменять направление матрицы, используется функция flip, которая принимает эти два параметра на вход.

При объединении данных функций получается функция apply_transformation. Она применяет преобразование к входной матрице путем смены направления; поворота на угол; регулировки контрастности и яркости.

Функция find_contrast_and_brightness2 определяет коэффициенты регулировки контрастности и яркости, которые приходят ей на вход на основе соотношения между исходными и целевыми массивами с использованием метода наименьших квадратов.

Функция generate_all_transformed_blocks генерирует все возможные преобразованные блоки из входной матрицы на основе заданных размеров источника и назначения; размера шага и возможных вариантов преобразования. Здесь происходит преобразование блоков матрицы matrix размера source_size x

source_size в новые блоки размера destination_size x destination_size с использованием заданного масштабного коэффициента factor. Затем для каждой возможной комбинации параметров direction и angle из списка candidates применяется преобразование блока S (полученного из исходной матрицы) и сохраняется в виде кортежа (k, l, direction, angle, transformed_block) в списке transformed_blocks. Таким образом, на выходе получается список преобразованных блоков вместе с их местоположением и параметрами преобразования.

Следующая функция — это функция compress, сжимающая входную матрицу через поиск наилучшего преобразования для каждого блока. Здесь происходит процесс поиска оптимального преобразования размера destination_size x destination_size в матрице matrix. Сначала создается список transformed_blocks, в котором хранятся все возможные преобразованные блоки матрицы matrix с использованием функции generate_all_transformed_blocks. Затем создается двумерный список transformations, в котором определяется оптимальное преобразование. Для каждого блока входной матрицы D функция проходится по всем преобразованным блокам S из transformed_blocks. Для каждой пары D и S находятся такие параметры контраста и яркости, которые минимизируют сумму квадратов разностей между D и S. Найденные параметры сохраняются в transformations[i][j]. Таким образом, после завершения циклов в transformations будет храниться оптимальное преобразование для каждого блока входной матрицы matrix.

Последней функцией является функция decompress. Она распаковывает сжатую матрицу с помощью итеративной реконструкции. На каждой итерации для каждого блока в матрице transformations вычисляется преобразование и добавляется в результирующую матрицу cur_matrix. Для вычисления преобразования каждого блока используются параметры (k, l, flip, angle, contrast, brightness) из списка transformations. Для этой матрицы из iterations выделется блок размером source_size x source_size, который уменьшается в factor раз; и применяются параметры преобразования для получения нового блока D. После

прохода по всем блокам исходной матрицы на каждой итерации результат записывается в список iterations и обнуляется cur_matrix. Такой цикл продолжается nb_iter итераций. Далее формируется итоговая матрица размером height x width из блоков, полученных на каждой итерации.

Рассмотрим работу такого алгоритма на матрице смежности (см. Приложение 6). Модель Эрдёша-Реньи сгенерировала следующий граф (рисунок 15).

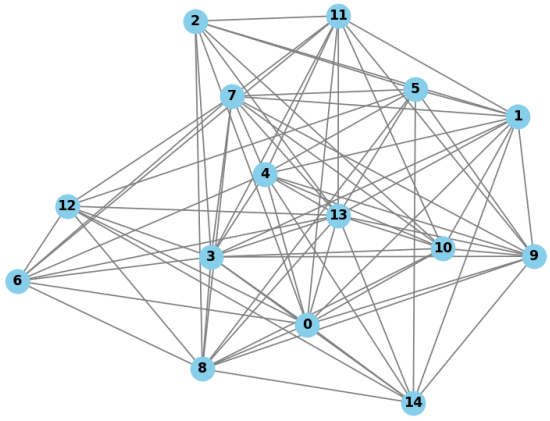

Рисунок 15. Случайный граф с 15 вершинами

После фрактального сжатия получаются такие 3 маски (рисунок 16).

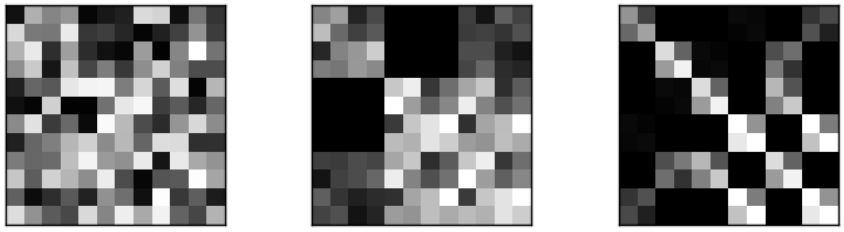

Рисунок 16. Результат фрактального сжатия

Видно, что создаются домены. Противоречий математической теории не наблюдается.

Дополнительно можно создать функцию, которая бы выдавала матрицу с максимальной энтропией. В науках о данных этот показатель часто бывает применим (см. Приложение 3).

Дальше рассмотрим функции, связанные с прунингом (см. Приложение 5).

Начнем с функции print_nonzeros. Она вычисляет и выводит статистику, связанную с разреженностью модели машинного обучения. Конкретно для

каждого параметра в модели код рассчитывает количество ненулевых (nonzero) элементов; общее количество параметров; процент разреженности в параметре; количество обнуленных параметров (total_params - nz_count); размер тензора (shape). Затем код выводит общее количество ненулевых элементов (alive); количество обнуленных элементов (pruned); общее количество элементов (total); коэффициент сжатия (Compression rate); процент обнуленных элементов.

Далее по пунктам представим функцию обучения нейронной сети (train):

1. В начале определяется значение EPS как 1e-6.
2. Затем устройство, на котором будет происходить обучение — GPU (если доступен) или CPU.
3. Модель переводится в режим обучения.
4. Далее идет цикл обучения по батчам из загруженных данных.
5. Оптимизатор обнуляется.
6. Входные изображения и классы переносятся на устройство (GPU/CPU).
7. Подается вход на модель, которая возвращает выход.
8. Вычисляется функция потерь между выходом модели и целью.
9. Рассчитываются градиенты функции потерь.
10. Далее идет блок, где происходит "заморозка" обнуления весов модели. Если в имени параметра содержится 'weight', то для этого параметра проверяется условие: если значение меньше EPS, то градиент равняется 0.
11. Градиенты передаются в оптимизатор для обновления весов модели.

Аналогично проделываются этапы для функции тестирования модели (test):

1. Определяет устройство (девайс), на котором будет выполняться операция.
2. Устанавливает модель в режим оценки (evaluation mode) с помощью model.eval().
3. Инициализирует переменные test_loss и correct для хранения значения потерь и количества правильных предсказаний.
4. Запускает цикл через тестовый загрузчик test_loader.

5. Передает данные и метки входного тензора на выбранный девайс.
6. Проходит данные через модель, чтобы получить выходные значения.
7. Вычисляет потери с использованием кросс-энтропии.
8. Вычисляет количество правильных предсказаний.
9. Корректирует общие потери для данного мини-пакета.
10. По завершении цикла вычисляет суммарные потери и точность модели.
11. Возвращает процент правильных предсказаний (accuracy).

Следующая функция инициализации весов (weight_init). Рассмотрим первую эпоху, когда веса модели еще не обновлялись. Логично предположить, что они должны быть. Есть разные стратегии по их наполнению. Можно начать с заполнения нулями. Но эмпирически выявлено, что лучше всего модели обучаются, если инициализировать веса из нормального распределения (рисунок 17).

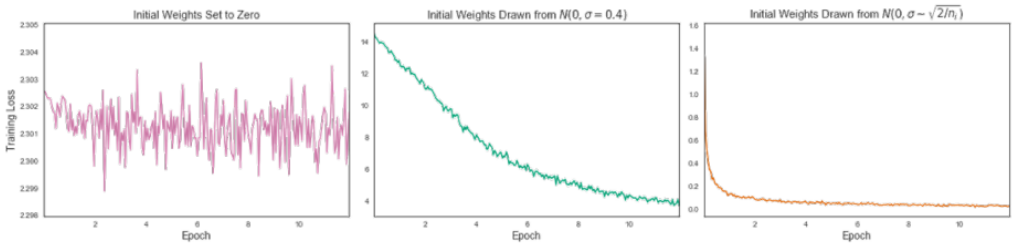

Рисунок 17. Влияние разных стратегий инициализации весов
на дальнейшую сходимость обучения

Современные методы опираются на инициализацию Ксавье [9], которая заключается в случайном выборе значений равномерного распределения на интервале от -1 до 1; после чего веса уменьшаются в $\frac{1}{\sqrt{n}}$ раз, где n — число входных единиц. Такой метод эмпирически показал большую устойчивость к затуханию и взрыву градиентов. Именно так и инициализируются веса модели LeNet в данной реализации.

Далее разберем функцию маскирования:

1. Она сначала инициализирует переменную step с нулевым значением; затем перебирает параметры модели, используя метод named_parameters(). Если имя параметра содержит слово "weight", то step увеличивается на 1.

2. Затем создается список mask с длиной, равной количеству весовых тензоров, найденных на первом шаге.
3. Снова устанавливается step на 0 и повторно перебираются параметры модели. Если имя параметра содержит слово "weight", то для этого весового тензора вычисляется массив Numpy с помощью param.data.cpu().numpy() и создается массив маски из единиц, размером аналогичным форме тензора. Результат сохраняется в соответствующем индексе mask.

Последней функцией стала функция prune_by_percentile, которая прореживает веса по порогу, основанному на перцентиле.

Всего у модели 44 000 параметров. Все они обучаемые (см. Приложение 7). Вес модели меньше мегабайта, что делает ее достаточно портативной.

Рассмотрим результаты обучения небольшой модели. Было принято решение взять десять эпох. Со второй эпохи у модели постепенно уменьшается число ненулевых весов. Такое количество эпох взято из-за склонности модели к переобучению. Итого к последней эпохе прорежется 60% весов (см. Приложение 8).

Взглянем на исторические значения точности на каждой эпохе (рисунок 18).

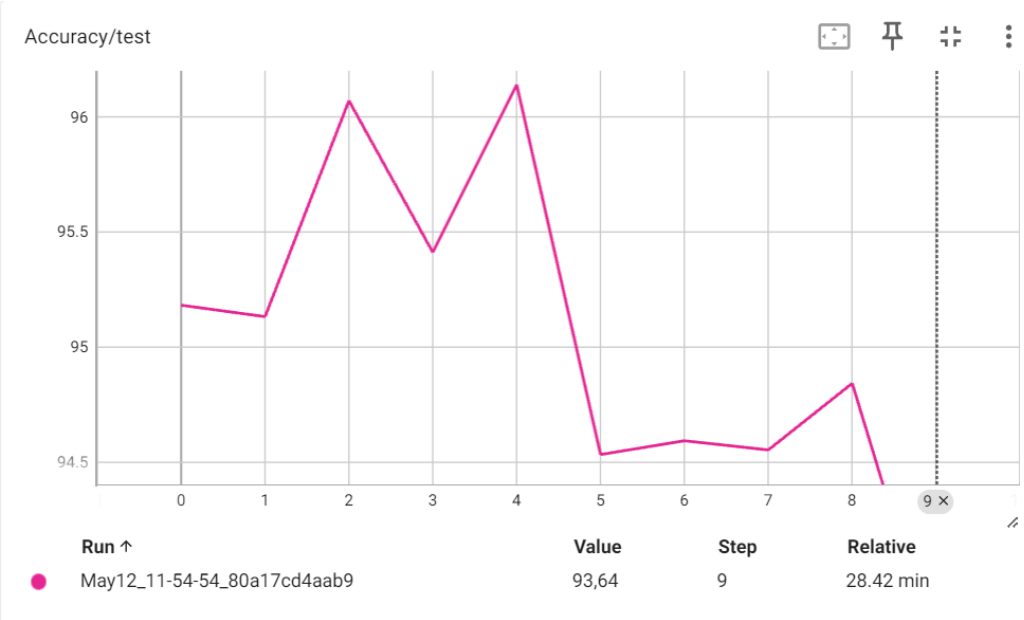

Рисунок 18. Значение точности на каждой эпохе после прореживания на тестовой выборке

Для модели LeNet является приемлемым результатом 98% точности. На итогах прореживания видно, что пик приходится на эпоху под номером 4. Значение точности достигает 96%. Дальнейший прунинг приводит к заметному падению метрики. Обучение заняло около получаса.

На одно предсказание оригинальная модель в среднем тратит несколько десятков миллисекунд (в зависимости от текущей загруженности устройства); прореженная же модель справляется на 15% быстрее.

Глава 3. Обзор и сравнение методов оптимизации нейронных сетей

3.1 Альтернативные решения оптимизации нейронных сетей

Одним из современных методов ускорения нейросетей (без привлечения дополнительных вычислительных мощностей) является квантизация [15].

Квантизация представляет собой метод оптимизации модели нейронной сети, который заключается в преобразовании параметров модели, весов, из формата с плавающей точкой (Float) в форматы целых чисел (Int8, Int12 или UInt8). Реже может использоваться половинная точность (fp-16). Путем применения этого метода происходит ускорение работы модели и сокращение объема используемой памяти; при этом точность модели снижается незначительно. Повышение скорости работы происходит благодаря более быстрой обработке операций сложения и умножения с целыми числами по сравнению с операциями над числами с плавающей точкой. Сокращение размера модели достигается за счет использования целочисленных типов данных (Int8, Int12, UInt8), так как они занимают меньше памяти (1 байт) в сравнении с числовыми типами с плавающей точкой (Float, 4 байта).

Сам процесс квантизации числа $x \in [\gamma_1; \gamma_2]$ в целое число $x_{quant} \in \left[\gamma_{1_q}; \gamma_{2_q}\right]$ можно формализовать следующим образом:

$$x_{quant} = clip(round\left(\frac{1}{s}x + z\right), \gamma_{1_q}, \gamma_{2_q})$$

$$clip(x, a_1, a_2) = \begin{cases} a_1 \; if \; x < \; a_1 \\ x \; if \; x \in [a_1; a_2] \\ a_2 \; if \; x > \; a_2 \end{cases}$$

Параметры s и z являются дополнительными параметрами квантизации и определяются для каждой матрицы весов по следующим формулам:

$$s = \frac{\gamma_2 - \gamma_1}{\gamma_{2_q} - \gamma_{1_q}},$$

$$z = \left(\frac{\gamma_2 \gamma_{1_q} - \gamma_1 \gamma_{2_q}}{\gamma_2 - \gamma_1}\right).$$

Классически параметры γ_1 и γ_2 определяются как минимальное и максимальное значение в развернутом векторе квантуемой матрицы.

Еще пользуется популярностью метод дистилляции знаний (knowledge distillation) в машинном обучении — это метод, при помощи которого обучается "учительская" модель, передающая свои знания "студенческой" модели [25]. Цель этого процесса заключается в передаче знаний из более сложной и мощной модели (учителя) в менее сложную и более легкую модель (студента), чтобы достигались сравнимые или даже лучшие результаты на тестовой выборке.

Процесс дистилляции знаний включает в себя обучение учительской модели на обучающих данных и передачу знаний в виде "мягких меток" или "дистиллированных" знаний студенческой модели вместо использования обычных "жестких меток" (ground truth). Мягкие метки — это вероятности того, что класс объекта принадлежит к определенному классу, который генерируется учительской моделью.

Использование дистилляции знаний может помочь улучшить обобщающую способность студенческой модели; ускорить ее обучение; уменьшить объем тренировочных данных; уменьшить сложность модели; и улучшить ее интерпретируемость. Также это метод может помочь при сжатии моделей для их более эффективного использования на устройствах с ограниченными ресурсами. Дистилляция знаний является важным методом в машинном обучении для передачи знаний более сложных моделей в менее сложные модели для повышения их производительности и эффективности.

По-особому можно подойти к реализации прунинга, например, работать не с весами, а с физическими нейронами в структуре сети путем их выкидывания. Такой метод классически принято называть dropout. Суть его в том, что определяется процент в диапазоне от 20 до 50 нейронов, которые случайным образом больше не будут исходной частью сети.

Существует надстройка над этим методом, называемая Monte Carlo Dropout, где подключается имитационное моделирование. Оно используется для

того, чтобы корректно получать распределение вероятностей от нейронных сетей.

3.2 Бенчмаркинг

Посмотрим на сводную таблицу по критериям (Таблица 1).

Таблица 1. Бенчмаркинг

Критерии	**LeNet**	**Pruning**	**Fractal Pruning**
Время обучения	19 мин	25 мин	29 мин
Инференс на одном изображении	30 мс	23 мс	25 мс
Лучшая достигнутая точность [MNIST]	~98%	~98.5%	~96%
Вес модели	0.22 Мб	0.20 Мб	0.20 Мб

Исходя из результатов экспериментов, представленных в таблице по применению алгоритма fractal pruning к нейронной сети LeNet, можно сделать следующие выводы данной ВКР:

1. Время обучения: алгоритм fractal pruning увеличил время обучения на 10 минут по сравнению с базовой моделью LeNet и на 4 минуты по сравнению с простым прунингом. Это может быть связано с дополнительной сложностью процесса оптимизации и вычислений, необходимых для применения фрактального сжатия матрицы весов.

2. Инференс на одно изображение: алгоритм fractal pruning продемонстрировал незначительное увеличение времени инференса на одно изображение по сравнению с базовой моделью LeNet и простым прунингом. Однако разница во времени незначительная и продолжает оставаться на уровне миллисекунд.

3. Лучшая достигнутая точность: алгоритм fractal pruning продемонстрировал не самую большую точность на датасете MNIST (96%) по сравнению с базовой моделью LeNet и простым прунингом (~98% и ~98.5% соответственно). Это может быть связано с компромиссом между сжатием модели и сохранением ее точности. Несмотря на это данный результат является приемлемым — дельта с материнской моделью менее 5%.

4. Вес модели: алгоритм fractal pruning показал незначительное уменьшение веса модели (до 0,20 Мб), сохраняя при этом такой же вес, как и при простом прунинге. Это говорит об эффективности модели при использовании фрактального сжатия.

Основываясь на полученных результатах, можно сделать вывод, что алгоритм фрактального прореживания является эффективным инструментом для сжатия нейронных сетей с незначительным изменением во времени обучения и инференса; и с небольшим снижением точности модели. Эти выводы важны для дальнейшего исследования и оптимизации методов сжатия нейронных сетей. У данного метода есть потенциал развития.

Заключение

В данной работе были рассмотрены вопросы, связанные с историей искусственного интеллекта; алгоритмов обучения нейронных сетей; их оптимизации; фрактального прунинга; гипотезы лотерейного билета. Было выявлено, что прунинг действительно помогает оптимизировать скорость выхода результата нейронных сетей без серьезных потерь в качестве.

Были изучены стратегии по облегчению моделей; математические основы сжимающих отображений, являющихся базой для фрактального сжатия; алгоритмическая реализация прореживания весов, благодаря которым в сумме получается достигать приемлемых результатов в точности модели.

Указанные в работе методы фрактального анализа позволяют с высокой точностью обучать модели искусственного интеллекта.

Были решены поставленные задачи:

- рассмотрены теоретические основы нейронных сетей и методы их оптимизации;
- представлен краткий обзор истории искусственных нейронных сетей как предпосылок главной задачи данной работы;
- описана математическая база, лежащая как в основе определенной архитектуры, так и в методе фрактального сжатия;
- выполнена алгоритмизация математических основ; создан фрактальный прунер;
- сравнены результаты выходов прореженной модели с материнской.

Практическая значимость данной ВКР заключается в результатах, которые являются значимыми в области оптимизации нейронных сетей. Эта работа может послужить точкой развития для применения фрактальных алгоритмов в других областях искусственного интеллекта и компьютерных наук в целом.

Список используемой литературы

1. Adam // PyTorch: docs. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html (дата обращения: 10.04.2024).
2. Alan Turing and the beginning of AI // Britannica. URL: https://www.britannica.com/technology/artificial-intelligence/Alan-Turing-and-the-beginning-of-AI (дата обращения: 07.03.2024).
3. Arthur Samuel // Schneppat AI. URL: https://schneppat.com/arthur-samuel.html (дата обращения: 17.03.2024).
4. CrossEntropyLoss // PyTorch: docs. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (дата обращения: 02.04.2024).
5. Datasets: mnist // Hugging Face. URL: https://huggingface.co/datasets/mnist (дата обращения: 24.04.2024).
6. Farley B., Clark W. Simulation of self-organizing systems by digital computer // Transactions of the IRE Professional Group on Information Theory. – 1954. – Т. 4. – №. 4. – С. 76-84.
7. Frankle J. et al. Stabilizing the lottery ticket hypothesis // arXiv preprint arXiv:1903.01611. – 2019.
8. Frankle J., Carbin M. The lottery ticket hypothesis: Finding sparse, trainable neural networks // arXiv preprint arXiv:1803.03635. – 2018.
9. Glorot X., Bengio Y. Understanding the difficulty of training deep feedforward neural networks // Proceedings of the thirteenth international conference on artificial intelligence and statistics. – JMLR Workshop and Conference Proceedings, 2010. – С. 249-256.
10. Hebb D. O. The organization of behavior: A neuropsychological theory. – Psychology press, 2005.
11. McCulloch W. S., Pitts W. A logical calculus of the ideas immanent in nervous activity // The bulletin of mathematical biophysics. – 1943. – Т. 5. – С. 115-133.

12. Minksy M., Edmonds D. Stochastic Neural Analogue Reinforcement Calculator // Princeton University. United States. – 1954.
13. Pruning tutorial: PyTorch // URL: https://pytorch.org/tutorials/intermediate/pruning_tutorial.html (дата обращения: 06.06.2024).
14. PyTorch // URL: https://pytorch.org/ (дата обращения: 02.05.2024)
15. Quantization // Hugging Face. URL: https://huggingface.co/docs/optimum/concept_guides/quantization (дата обращения: 12.05.2024).
16. Rosenblatt F. Perceptron simulation experiments // Proceedings of the IRE. – 1960. – Т. 48. – №. 3. – С. 301-309.
17. Shimbel A., Rapoport A. A statistical approach to the theory of the central nervous system // The bulletin of mathematical biophysics. – 1948. – Т. 10. – С. 41-55.
18. Sohl-Dickstein J. The boundary of neural network trainability is fractal // arXiv preprint arXiv:2402.06184. – 2024.
19. Vaswani A. et al. Attention is all you need // Advances in neural information processing systems. – 2017. – Т. 30.
20. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025 // Statista. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (дата обращения: 27.02.2024).
21. Алексей Савватеев: Модели интернета и социальных сетей // Habr. URL: https://habr.com/ru/articles/458454/ (дата обращения: 01.05.2024).
22. Архитектуры нейросетей // Habr. URL: https://habr.com/ru/companies/nix/articles/430524/ (дата обращения: 24.04.2024).
23. Все, что вы хотели знать о перцептронах Розенблатта, но боялись спросить // Habr. URL: https://habr.com/ru/companies/sberdevices/articles/529932/ (дата обращения: 17.03.2024).

24. Глубинное обучение: возможности, перспективы и немного истории // Habr. URL: https://habr.com/ru/companies/cloud_mts/articles/309024/ (дата обращения: 18.03.2024).
25. Дистилляция знаний // Школа анализа данных: Учебник по машинному обучению. URL: https://education.yandex.ru/handbook/ml/article/distillyaciya-znanij (дата обращения: 12.05.2024).
26. Первое знакомство с полносвязными нейросетями // Школа анализа данных: Учебник по машинному обучению. URL: https://education.yandex.ru/handbook/ml/article/pervoe-znakomstvo-s-polnosvyaznymi-nejrosetyami (дата обращения: 29.02.2024).
27. Сверточная нейронная сеть с нуля // Program for you. URL: https://programforyou.ru/poleznoe/convolutional-network-from-scratch-part-zero-introduction (дата обращения: 05.05.2024).
28. Уэлстид С. Фракталы и вейвлеты для сжатия изображений в действии. – 2003.
29. Функции активации нейросети: сигмоида, линейная, ступенчатая, ReLu, than // Neurohive. URL: https://neurohive.io/ru/osnovy-data-science/activation-functions/ (дата обращения: 24.04.2024).
30. Что такое искусственный интеллект? // Neurohive. URL: https://neurohive.io/ru/osnovy-data-science/iskusstvennyj-intellekt-voprosy-i-otvety/ (дата обращения: 05.03.2024).

Приложение

Глоссарий

Бенчмарк (программирование) — эталонный тест производительности компьютерной системы.

Вычислительный граф — иллюстрированная запись какой-либо функции, состоящая из вершин и ребер.

Дистилляция знаний или модельная дистилляция — процесс передачи знаний из большой модели в меньшую. Хотя большие модели обладают более высоким объемом знаний, чем небольшие модели, их потенциал может быть использован не в полной мере.

Изоморфизм графов — биекция между множествами их вершин, сохраняющая отношение смежности.

Квантизация — метод, используемый для уменьшения размера больших нейронных сетей, включая большие языковые модели (LLM), путем изменения точности их весов.

Логит класса — ненормализованные предсказания (или выходные данные) модели.

Матрица смежности графа — квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа.

Отображением (оператором) f множества X в множество Y, или функцией, определенной на множестве X со значениями в множестве Y, называют соответствие, которое каждому элементу $x \in X$ соотносит некоторый однозначно определенный элемент $y \in Y$. Множество X называют областью определения и обозначают D f, элемент $x \in X$ — аргументом функции, а элемент $y \in Y$ — зависимым переменным.

Полное метрическое пространство — метрическое пространство, в котором каждая фундаментальная последовательность сходится (к элементу того же пространства).

Прунинг — сокращение числа синапсов или нейронов для повышения эффективности нейросети, удаления избыточных связей. Прунинг включает в себя как обрезку аксона, так и дендритов.

Свертка — операция, которая применяется к входному сигналу (например, изображению) с использованием фильтра (ядра свертки). Эта операция вычисляет скалярное произведение между фильтром и областью входного сигнала, что позволяет выявить определенные признаки в данных.

Тензорный поезд — алгоритм, который позволяет разлагать тензоры с большой размерностью на несколько тензоров малой размерности (ядер разложения), которые суммируются по общим индексам.

Фрактал — множество, обладающее свойством самоподобия (объект, в точности или приближенно совпадающий с частью себя самого, т.е. целое имеет ту же форму, что и одна или более частей)

Энтропия Шеннона — мера неопределенности, связанная со случайной величиной.

Приложение 1

```
import torch
import torch.nn as nn

def custom_pruning(module, name='weight', amount=0.2):
    weight = getattr(module, name)
    threshold = torch.abs(weight).max() * amount
    weight = torch.where(torch.abs(weight) < threshold, torch.tensor(0., device=weight.device), weight)
    setattr(module, name, nn.Parameter(weight))

class CustomNet(nn.Module):
    def __init__(self):
        super(CustomNet, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
```

```
            x = self.fc1(x)
            x = self.fc2(x)
            return x

    model = CustomNet()

    custom_pruning(model.fc1, name='weight', amount=0.2)
```

Приложение 2

```
import matplotlib.pyplot as plt
from scipy import ndimage
import numpy as np
import math

def reduce(matrix: np.array, factor: int) -> np.array:
    '''
    This function takes a matrix (2D numpy array) and reduces its size by a
specified factor.

    Arguments:
    matrix: 2D numpy array - The input matrix
    factor: int - The factor by which the matrix should be reduced

    Returns:
    np.array - The reduced size matrix
    '''
    result = np.zeros((matrix.shape[0] // factor, matrix.shape[1] //
factor))
    for i in range(result.shape[0]):
        for j in range(result.shape[1]):
            result[i,j] =
np.mean(matrix[i*factor:(i+1)*factor,j*factor:(j+1)*factor])
    return result

def rotate(matrix: np.array, angle: float) -> np.array:
    '''
    This function rotates the input matrix by a specified angle without
reshaping.

    Arguments:
    matrix: np.array - The input matrix
```

```
    angle: float - The angle by which the matrix should be rotated

    Returns:
    np.array - The rotated matrix
    '''
    return ndimage.rotate(matrix, angle, reshape=False)

def flip(matrix: np.array, direction: int) -> np.array:
    '''
    This function flips the input matrix along a specified direction.

    Arguments:
    matrix: np.array - The input matrix
    direction: int - The direction in which the matrix should be flipped

    Returns:
    np.array - The flipped matrix
    '''
    return matrix[::direction,:]

def apply_transformation(matrix: np.array, direction: int, angle: float,
contrast: float = 1.0, brightness: float = 0.0) -> np.array:
    '''
    This function applies a transformation to the input matrix by flipping
along a direction, rotating by an angle,
    and adjusting contrast and brightness.

    Arguments:
    matrix: np.array - The input matrix
    direction: int - The direction for flipping the matrix
    angle: float - The angle for rotating the matrix
    contrast: float - The factor for contrast adjustment (default is 1.0)
    brightness: float - The factor for brightness adjustment (default is
0.0)

    Returns:
    np.array - The transformed matrix
    '''
    return contrast*rotate(flip(matrix, direction), angle) + brightness
```

```
def find_contrast_and_brightness2(D: np.array, S: np.array) -> tuple:
    '''
    This function finds the contrast and brightness adjustment factors based
on the relationship between
    the source (S) and the destination (D) arrays using least squares
method.

    Arguments:
    D: np.array - The destination array
    S: np.array - The source array

    Returns:
    tuple - The contrast factor and brightness factor as a tuple
    '''
    A = np.concatenate((np.ones((S.size, 1)), np.reshape(S, (S.size, 1))),
axis=1)
    b = np.reshape(D, (D.size,))
    x, _, _, _ = np.linalg.lstsq(A, b)
    return x[1], x[0]

def generate_all_transformed_blocks(matrix: np.array, source_size: int,
destination_size: int, step: int, candidates: list) -> list:
    '''
    This function generates all possible transformed blocks from the input
matrix based on the specified source and destination sizes,
    step size, and transformation candidates.

    Arguments:
    matrix: np.array - The input matrix
    source_size: int - The size of the source block
    destination_size: int - The size of the destination block
    step: int - The step size for block extraction
    candidates: list - List of transformation candidates (directions and
angles)

    Returns:
    list - List of tuples containing transformation information and
transformed blocks
    '''
    factor = source_size // destination_size
```

```
    transformed_blocks = []
    for k in range((matrix.shape[0] - source_size) // step + 1):
        for l in range((matrix.shape[1] - source_size) // step + 1):
            # Extract the source block and reduce it to the shape of a
destination block
            S =
reduce(matrix[k*step:k*step+source_size,l*step:l*step+source_size], factor)
            # Generate all possible transformed blocks
            for direction, angle in candidates:
                transformed_blocks.append((k, l, direction, angle,
apply_transformation(S, direction, angle)))
    return transformed_blocks

def compress(matrix: np.array, source_size: int, destination_size: int,
step: int, candidates: list) -> list:
    '''
    This function compresses the input matrix by finding the best
transformations for each block.

    Arguments:
    matrix: np.array - The input matrix
    source_size: int - The size of the source block
    destination_size: int - The size of the destination block
    step: int - The step size for block extraction
    candidates: list - List of transformation candidates (directions and
angles)

    Returns:
    list - List of lists containing the best transformation information for
each block
    '''
    transformations = []
    transformed_blocks = generate_all_transformed_blocks(matrix,
source_size, destination_size, step, candidates)
    i_count = matrix.shape[0] // destination_size
    j_count = matrix.shape[1] // destination_size
    for i in range(i_count):
        transformations.append([])
        for j in range(j_count):
            transformations[i].append(None)
```

```
            min_d = float('inf')
            D =
matrix[i*destination_size:(i+1)*destination_size,j*destination_size:(j+1)*de
stination_size]
            for k, l, direction, angle, S in transformed_blocks:
                contrast, brightness = find_contrast_and_brightness2(D, S)
                S = contrast*S + brightness
                d = np.sum(np.square(D - S))
                if d < min_d:
                    min_d = d
                    transformations[i][j] = (k, l, direction, angle,
contrast, brightness)
    return transformations

def decompress(transformations: list, source_size: int, destination_size:
int, step: int, nb_iter: int = 8) -> list:
    '''
    This function decompresses the compressed matrix using iterative
reconstruction.

    Arguments:
    transformations: list - List of lists containing transformation
information for each block
    source_size: int - The size of the source block
    destination_size: int - The size of the destination block
    step: int - The step size for block extraction
    nb_iter: int - Number of iterations for reconstruction (default is 8)

    Returns:
    list - List of reconstructed matrices at each iteration
    '''
    factor = source_size // destination_size
    height = len(transformations) * destination_size
    width = len(transformations[0]) * destination_size
    iterations = [np.random.randint(0, 256, (height, width))]
    cur_matrix = np.zeros((height, width))
    for i_iter in range(nb_iter):
        for i in range(len(transformations)):
            for j in range(len(transformations[i])):
```

```
                k, l, flip, angle, contrast, brightness =
transformations[i][j]
                S = reduce(iterations[-
1][k*step:k*step+source_size,l*step:l*step+source_size], factor)
                D = apply_transformation(S, flip, angle, contrast,
brightness)
                cur_matrix[i*destination_size:(i+1)*destination_size,j*desti
nation_size:(j+1)*destination_size] = D
        iterations.append(cur_matrix)
        cur_matrix = np.zeros((height, width))
    return iterations
```

Приложение 3

```
def shannon_entropy(matrix: np.array) -> float:
    '''
    This function calculates the Shannon entropy of the input matrix.

    Arguments:
    matrix: np.array - The input matrix

    Returns:
    float - The Shannon entropy value
    '''
    flattened_matrix = matrix.flatten()
    _, counts = np.unique(flattened_matrix, return_counts=True)
    probabilities = counts / len(flattened_matrix)
    entropy = -np.sum(probabilities * np.log2(probabilities))
    return entropy
```

Приложение 4

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()

        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Приложение 5

```
def print_nonzeros(model: LeNet) -> float:
    '''
    This function prints the number of nonzeros parameters in the given
model and calculates the compression rate.

    Arguments:
    model: LeNet - The input model

    Returns:
    float - The percentage of nonzeros parameters in the model
    '''
    nonzero = total = 0
    for name, p in model.named_parameters():
        tensor = p.data.cpu().numpy()
        nz_count = np.count_nonzero(tensor)
        total_params = np.prod(tensor.shape)
        nonzero += nz_count
        total += total_params
        print(f'{name:20} | nonzeros = {nz_count:7} / {total_params:7} ({100
* nz_count / total_params:6.2f}%) | total_pruned = {total_params - nz_count
:7} | shape = {tensor.shape}')
    print(f'alive: {nonzero}, pruned : {total - nonzero}, total: {total},
Compression rate : {total/nonzero:10.2f}x  ({100 * (total-nonzero) /
total:6.2f}% pruned)')
    return (round((nonzero/total)*100,1))

def train(model, train_loader, optimizer, criterion):
    '''
```

```
    This function trains the model using the given train loader, optimizer,
and criterion.
    It also applies weight freezing for pruned weights.

    Arguments:
    model: nn.Module - The input neural network model
    train_loader: torch.utils.data.DataLoader - The train data loader
    optimizer: torch.optim - The optimizer for training
    criterion: torch.nn - The loss function

    Returns:
    float - Training loss
    '''
    EPS = 1e-6
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.train()
    for batch_idx, (imgs, targets) in enumerate(train_loader):
        optimizer.zero_grad()
        imgs, targets = imgs.to(device), targets.to(device)
        output = model(imgs)
        train_loss = criterion(output, targets)
        train_loss.backward()

        # Freezing Pruned weights by making their gradients Zero
        for name, p in model.named_parameters():
            if 'weight' in name:
                tensor = p.data.cpu().numpy()
                grad_tensor = p.grad.data.cpu().numpy()
                grad_tensor = np.where(tensor < EPS, 0, grad_tensor)
                p.grad.data = torch.from_numpy(grad_tensor).to(device)
        optimizer.step()
    return train_loss.item()

def test(model, test_loader, criterion):
    '''
    This function tests the model using the given test loader and criterion.

    Arguments:
    model: nn.Module - The input neural network model
    test_loader: torch.utils.data.DataLoader - The test data loader
```

```
    criterion: torch.nn - The loss function

    Returns:
    float - Test accuracy
    '''
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target,
reduction='sum').item()  # sum up batch loss
            pred = output.data.max(1, keepdim=True)[1]  # get the index of
the max log-probability
            correct += pred.eq(target.data.view_as(pred)).sum().item()
        test_loss /= len(test_loader.dataset)
        accuracy = 100. * correct / len(test_loader.dataset)
    return accuracy

def prune_by_percentile(percent,**kwargs):
    '''
    This function prunes the model parameters based on the given percentile
value.

    Arguments:
    percent: float - The percentile value for pruning
    **kwargs: additional keyword arguments

    Returns:
    None
    '''
        global step
        global mask
        global model

        step = 0
        for name, param in model.named_parameters():
```

```
            if 'weight' in name:
                tensor = param.data.cpu().numpy()
                if 'conv' in name:
                  tensor = param.data.cpu().numpy()
                  alive = tensor[np.nonzero(tensor)] # flattened array of
nonzero values
                  percentile_value = np.percentile(abs(alive), percent)
                if 'fc' in name:
                  matr = reduce(tensor, 1)
                  transformations = compress(matr, tensor.shape[0]//3,
tensor.shape[0]//3, tensor.shape[0]//2, candidates)
                  iterations =
decompress(transformations,tensor.shape[0]//3, tensor.shape[0]//3,
tensor.shape[0]//2)
                  entropies = [shannon_entropy(matrix) for matrix in
iterations]
                  iteration = iterations[entropies.index(max(entropies))]
                  alive = tensor[np.nonzero(iteration)] # flattened array of
nonzero values
                  percentile_value = min(alive)
                weight_dev = param.device
                new_mask = np.where(abs(tensor) < percentile_value, 0,
mask[step])
                param.data = torch.from_numpy(tensor *
new_mask).to(weight_dev)
                mask[step] = new_mask
                step += 1
        step = 0
def make_mask(model):
    '''
    This function creates a mask for the model parameters.

    Arguments:
    model: nn.Module - The input neural network model

    Returns:
    None
    '''
    global step
    global mask
```

```
    step = 0
    for name, param in model.named_parameters():
        if 'weight' in name:
            step = step + 1
    mask = [None]* step
    step = 0
    for name, param in model.named_parameters():
        if 'weight' in name:
            tensor = param.data.cpu().numpy()
            mask[step] = np.ones_like(tensor)
            step = step + 1
    step = 0
def weight_init(m):
    '''
    Usage:
        model = Model()
        model.apply(weight_init)
    '''
    if isinstance(m, nn.Conv2d):
        init.xavier_normal_(m.weight.data)
        if m.bias is not None:
            init.normal_(m.bias.data)

    elif isinstance(m, nn.Linear):
        init.xavier_normal_(m.weight.data)
        init.normal_(m.bias.data)
```

Приложение 6

```
import networkx as nx
import matplotlib.pyplot as plt
import random
from fractal_compress import *

seed_value = 42
random.seed(seed_value)

G = nx.erdos_renyi_graph(15, 0.7)
adj_matrix = nx.to_numpy_array(G)
nx.draw(G, with_labels=True, node_color='skyblue', edge_color='gray',
font_weight='bold', font_size=10)
plt.show()
```

```
print(adj_matrix.sum())

orientations = [1, -1]
angles = [0, 90, 180, 270]
candidates = [[orientation, angle] for orientation in orientations for angle
in angles]

adj_matrix = reduce(adj_matrix, 1)

transformations = compress(adj_matrix, 8, 4, 8, candidates)
iterations = decompress(transformations, 8, 4, 8)
plot_iterations(iterations, adj_matrix)
plt.show()
```

Приложение 7

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 6, 24, 24]             156
         MaxPool2d-2           [-1, 6, 12, 12]               0
            Conv2d-3           [-1, 16, 8, 8]            2,416
         MaxPool2d-4           [-1, 16, 4, 4]               0
            Linear-5                 [-1, 120]          30,840
            Linear-6                  [-1, 84]          10,164
            Linear-7                  [-1, 10]             850
================================================================
Total params: 44,426
Trainable params: 44,426
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.04
Params size (MB): 0.16
Estimated Total Size (MB): 0.20
----------------------------------------------------------------
```

Приложение 8

```
--- Pruning Level [10/10]: ---
conv1.weight         | nonzeros =      57 /     150 ( 38.00%) | total_pruned
=      93 | shape = (6, 1, 5, 5)
```

```
conv1.bias           | nonzeros =       6 /       6 (100.00%) | total_pruned
=       0 | shape = (6,)
conv2.weight         | nonzeros =     932 /    2400 ( 38.83%) | total_pruned
=    1468 | shape = (16, 6, 5, 5)
conv2.bias           | nonzeros =      16 /      16 (100.00%) | total_pruned
=       0 | shape = (16,)
fc1.weight           | nonzeros =   11946 /   30720 ( 38.89%) | total_pruned
=   18774 | shape = (120, 256)
fc1.bias             | nonzeros =     120 /     120 (100.00%) | total_pruned
=       0 | shape = (120,)
fc2.weight           | nonzeros =    3912 /   10080 ( 38.81%) | total_pruned
=    6168 | shape = (84, 120)
fc2.bias             | nonzeros =      84 /      84 (100.00%) | total_pruned
=       0 | shape = (84,)
fc3.weight           | nonzeros =     326 /     840 ( 38.81%) | total_pruned
=     514 | shape = (10, 84)
fc3.bias             | nonzeros =      10 /      10 (100.00%) | total_pruned
=       0 | shape = (10,)
alive: 17409, pruned : 27017, total: 44426, Compression rate :       2.55x
( 60.81% pruned)
Train Epoch: 10/10 Loss: 0.190113 Accuracy: 93.64% Best Accuracy: 96.14%:
100%|██████████| 10/10 [03:07<00:00, 18.79s/it]
```