РЕФЕРАТ

Выпускная квалификационная работа бакалавра содержит 44 страницы, 15 рисунков, 2 таблицы, 16 использованных источников, 1 приложение.

НЕЙРОННЫЕ СЕТИ, УМЕНЬШЕНИЕ КОЛИЧЕСТВА ПАРАМЕТРОВ ОБУЧЕНИЯ, СЖАТИЕ НЕЙРОННЫХ СЕТЕЙ, PYTORCH, ЭФФЕКТИВНОЕ ПРЕДСТАВЛЕНИЕ ТЕНЗОРА, ТЕНЗОРНЫЕ ПОЕЗДА.

В данной работе рассматривается способ представления многомерного массива в виде тензорного поезда и его применение для сжатия нейронной сети.

Цель работы – применить формат тензорного поезда для многократного уменьшения количества параметров полносвязного слоя с последующим после этого обучением нейронной сети.

В ходе выполнения работы был реализован полносвязный слой с применением тензорных поездов на фреймворке Pytorch, выполнено обучение нейронных сетей с данными слоями, произведен анализ процесса обучения и эффективности применения тензорных поездов.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В данной работе будут использоваться массивы разных размерностей:

1. Под *вектором* будем понимать одномерный массив. Для обозначения будут использоваться строчные латинские символы, выделенные курсивом: $a\ c\ d$. Элемент вектора будет обозначаться следующим образом: $a(i)$, где i – индекс данного элемента.
2. Под *матрицей* будем понимать двумерный массив. Для обозначения будут использоваться прописные латинские символы, выделенные курсивом: $A\ C\ D$. Элемент матрицы будет обозначаться следующим образом: $\mathrm{A}(i,j)$, где i,j – индексы данного элемента.
3. Под *тензором* будем понимать массив размерности более двух. Для обозначения будут использоваться прописные каллиграфические символы, выделенные курсивом: $\mathcal{A}\ \mathcal{C}\ \mathcal{D}$. Элемент матрицы будет обозначаться следующим образом: $\mathcal{A}(i_1,\dots,i_d)$, где $i_1,\dots,i_d$ – индексы данного элемента тензора размерности d.

ВВЕДЕНИЕ

Нейронные сети – это подмножество методов машинного обучения, которое само по себе заключается в составлении моделей, способных учиться на основе данных и совершенствоваться при помощи полученного опыта. Нейронные сети используются для решения сложных задач, которые требуют аналитических вычислений, схожих с рассуждениями человека. Они стали невероятно популярны и сейчас повсесеместно используются для решения широкого спектра задач, начиная с классификации, регрессии и заканчивая генерацией, детектированием и не только. Они применяются в системах навигации, к примеру, в беспилотных автомобилях или промышленных роботах. Алгоритмы на основе нейросетей защищают информационные системы от атак злоумышленников и помогают выявлять незаконный контент в сети. Таких примеров использования нейронных сетей становится все больше и больше. Однако чем сложнее задача, тем больше и сложнее приходится использовать модель. Сеть 2020 года GPT-3 использует 175 млрд. параметров, при этом наборы данных для ее обучения составили 570 ГБ текстов. Такая мощность позволяет модели генерировать осмысленные тексты, создавать музыку и программный код. Размеры сети действительно внушительны, однако в 2021 команда из Google Research представила сеть Switch Transformer, которая вмещает в себя уже 1,6 трлн параметров. Но и это еще не предел - в Китае была представлена нейросеть Wu Dao 2.0 с 1,75 трлн. параметров. Wu Dao является мультимодальной системой и способна выполнять задачи по обработке естественного языка, генерации текста, распознаванию и созданию изображений. Модель может не только писать эссе, стихи и двустишия на китайском языке, но и генерировать альтернативный текст на основе статического изображения и фотореалистичные изображения на основе описаний на естественном языке. Обучалась такая модель на китайском и английском языках на 4.9 ТБ изображений и текстов. Способности нейросетей впечатляют, однако есть и обратная сторона. Большие объемы занимаемой памяти – одна из самых больших проблем глубинных нейронных сетей. Нейронным сетям нужна память для того, чтобы хранить входные данные, весовые параметры, функции активации, а также как вход распространяется через сеть. При этом результаты вычислений, полученные в процессе обучения, должны храниться в памяти до момента использования их для расчета обратного распространения ошибки. Если предположить, что один параметр обучения вышеупомянутой сети Wu Dao 2.0 занимает 4 байта, то в пересчете на всю сеть потребуется 7 TB RAM памяти только для хранения параметров. Естественно, отдельных GPU, содержащих такое количество памяти не существует. Даже если взять более

простую модель, например GPT-3(175 млрд. параметров), то ее время обучения на одной видеокарте по некоторым оценкам составило бы 355 лет при затратах в 4.6 млн. долларов. Поэтому, единственный способ эффективно решить данную задачу – использовать суперкомпьютеры, которые вмещают сотни, а иногда и тысячи GPU. Например, с этой задачей смогут справиться суперкомпьютеры компании Яндекс. Самый мощный из них, Червоненкис, включает в себя 1592 GPU с общим объемом оперативной памяти 199 TB. Однако стоимость такого суперкомпьютера внушительна, а содержание – достаточно затратное. Кроме того, такие вычислительные мощности доступны далеко не всем компаниям. Все это заставляет задуматься о задаче уменьшения количества параметров.

С другой стороны, это большое число параметров в нейронных сетях представляется в виде тензоров, т. е. многомерных массивов данных. Именно эти параметры участвуют в вычислениях и изменяются каждую итерацию обучения. Существует эффективный способ представления многомерных тензоров – Tensor Train(TT). TT является малопараметрическим представлением, по которому можно быстро восстановить весь тензор целиком. При этом это не просто архивация данных – в этом формате сохраняется возможность работы с любым из параметров. Главное преимущество такого формата в том, что он не страдает от «проклятия размерности» – экспоненциальная зависимость от числа размерностей заменяется линейной. Для сжатия количества параметров нейронных сетей был предложен метод, заключающийся в разложении слоев нейронной сети в тензорный поезд(Tensor-Train Decomposition). Такой формат позволяет компактно хранить тензоры в оперативной памяти компьютера, при этом остаются доступны все необходимые операции линейной алгебры. Этот метод позволяет значительно сократить число параметров, в то время как точность работы модели остается на прежнем уровне.

Проблема уменьшения числа параметров стала актуальна с появлением сетей глубокого обучения. Большое количество работ посвящено данной тематике с полносвязными нейронными сетями[7,8,9]. Понятие тензорный поезд появилось в 2011 году в работе Оселедеца И.В.[1], где он подробно описал принципы построения TT-разложения и основные операциями над тензорами в TT-формате. Применение тензорных поездов к нейронным сетям является лишь одним из многих приложений TT-формата и стало набирать популярность не сразу. В работе[2] успешно внедряют тензорный поезд в нейронную сеть, описав прямой и обратный проход в полносвязной сети. Однако авторы работы акцентировали внимание на самой идее применения тензорных поездов, при этом аспект процесса обучения нейронной сети и его анализа был раскрыт недостаточно. Несмотря на это, тензорные поезда получили распространение и на другие архитектуры

нейронных сетей. В работах были представлены способы использования TT-формата со сверточными слоями[4] и 3DCNN слоями[5], а также с рекуррентными слоями[6].

В данной работе представлен метод представления полносвязных слоев в TT-формате, основанный на специальном способе представления массивов параметров в виде многомерных тензоров. Основной вклад этой работы в развитие данной тематики заключается в следующем:

- Реализован алгоритм применения тензорных поездов для сжатия полносвязных слоев нейронной сети на фреймворке машинного обучения PyTorch;
- Произведено исследование процесса обучения нейросетей, содержащих TT-слои. Выбраны наилучшие настройки оптимизации обучения для нейронных сетей этого типа;
- Разработано оформление модуля для общедоступного применения на PyTorch.

В данной работе представлены: основные принципы работы нейронных сетей, способы обучения и их недостатки, теоретические аспекты разложения тензорных поездов, описание работы полносвязных слоев и алгоритм внедрения тензорных поездов в их работу, архитектуры нейронных сетей и практические результаты обучения, эксперименты, а также анализ самого процесса обучения.

1 Нейронные сети

1.1 Определение

Нейронная сеть – модель машинного обучения, которая представляет собой последовательность соединенных нейронов. Нейрон – вычислительная единица, которая получает на вход некоторые значения, выполняет с ними преобразование и передает далее. В простейшем случае работу нейрона можно представить в виде:

$$z = wx + b, \tag{1}$$

где x – вектор входных признаков,

w – вектор соответствующих им весов,

b – вектор соответствующих смещений.

Выходное значение нейрона определяется функцией активации. Она позволяет добавить нелинейность на выход, без которой линейные преобразования многослойной сети были бы эквивалентны однослойному линейному преобразованию. Наиболее популярной из-за своей простоты и легкости взятия производной является ReLu. Кроме перечисленных преимуществ, ReLu имеет еще одну важную особенность – разреженность активации. Это позволяет снизить количество включаемых нейронов и повысить производительность обучения, по сравнению с другими функциями активации, таких как гиперболический тангенс и сигмоидальной функции.

Выходная функция активации должна интерпретировать значения нейронов в выходное значение. Для задачи классификации хорошо подходит Softmax – на вход подается вектор значений, а на выходе получается вектор той же размерности с диапазоном значений от 0 до 1, которые можно рассматривать как вероятность принадлежности к выбранному классу.

Структурно нейроны можно разделить на слои: входной, скрытые и выходной. Структура нейронной сети показана на рисунке 1.

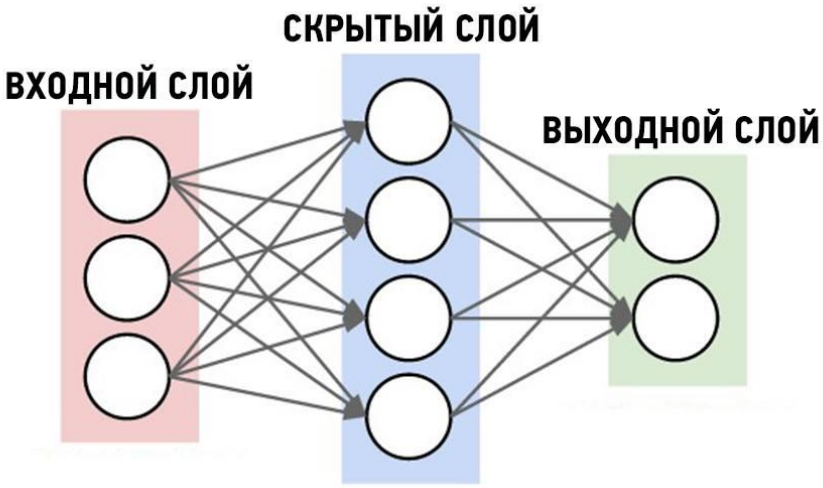

Рисунок 1 - Структура нейронной сети

1.2 Задача классификации

В данной работе рассматриваются нейронные сети для решения задач классификации. Формально постановка задачи выглядит следующим образом:

- X – конечное множество признаков объектов
- Y – конечное множество классов объектов
- Существует целевая зависимость – отображение $y^*: X \to Y$, значения которой существуют только на $X^m = \{(x_1, y_1), \dots, (x_m, y_m)\}$ – обучающей выборке
- Необходимо построить алгоритм $a: X \to Y$, приближающий y^* на любом $x \in X$

1.3 Обучение

После прямого прохода по нейронной сети полученный результат сравнивается с эталонным значением с помощью функции потерь. В задачах классификации, как правило, используется Log loss(логистическая функция потерь). Задача состоит в минимизации этой функции. Это достигается за счет корректировки параметров обучения.

Методы обучения нейронных сетей основываются на градиентном спуске. Идея заключается в том, что антиградиент указывает направление наискорейшего убывания функции в данной точке. Поэтому алгоритм считает частные производные по весам и обновляет их значения, основываясь на значениях градиента функции потерь в данной точке, а так же на параметре скорости обучения. Процесс обновления весов показан на рисунке 2.

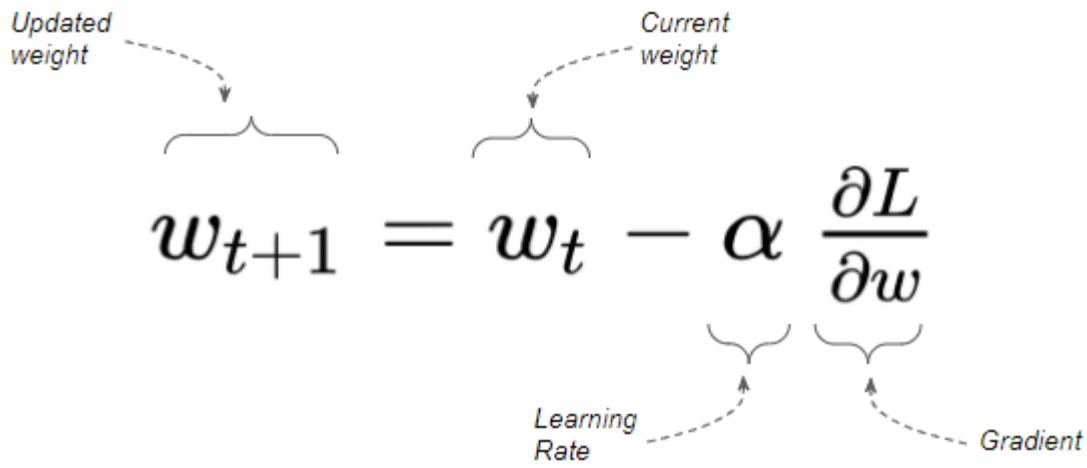

Рисунок 2 - Обновление весов

Однако на практике при обучении возникает множество проблем. Одна из них – локальные минимумы. Помимо глобального минимума на функции потерь может встретиться большое количество локальных минимумов. Попав в такую область, алгоритм часто уже не способен оттуда выбраться, и застревает там, так и не добравшись до глобального минимума. Кроме того, проблему представляют седловые точки. В таких точках функция в направлении одного параметра находится в локальном минимуме, в направлении другого – в локальном максимуме. Область вокруг седловой точки достаточно плоская, что делает градиент близким к нулю. Из-за этого алгоритм не может спускаться дальше и ложно полагает, что минимум найден. Следующая проблема – овраги. Овраги имеют крутой уклон в одном направлении и плавный уклон в другом. Сами по себе овраги часто приводят к минимуму, и оптимально двигаться по ним подобно тому, как вода течет в русле реки. Но алгоритм легко может начать движение, попеременно отталкиваясь от сторон оврага, при этом продвижение «по течению» будет минимальным.

Чтобы решить проблемы обучения и сделать его более эффективным, у алгоритма градиентного спуска есть большое количество методов оптимизации:

- Стохастический градиентный спуск (SGD). В этом случае обучение проходят мини-батчами(mini-batches). Для подсчета градиента и обновления параметров используются не все элементы набора данных, а только элементы мини-батча. Если подавать разные входные данные, то и значения градиента и функции ошибки будут отличаться, это позволяет алгоритму не застревать на определенных участках своего спуска;
- Накопление импульса(Momentum). Недостатком SGD является его полная зависимость от текущего вычисляемого градиента, что вызывает нестабильность

алгоритма. Momentum является аналогом импульса из физики: если в течении нескольких эпох наблюдается тенденция движения в одном направлении, то импульс не даст быстро от него отклониться и сохранит эту тенденцию. Это позволяет алгоритму более быстро продвигаться, когда направления градиентов совпадают, и замедляться, когда их направления меняются;

- Изменение скорости обучения. Во-первых, изменять скорость нужно для всех параметров в зависимости от этапа обучения. Очевидно, что вначале выгодно делать большие шаги, чтобы быстрее достичь цели, но по мере приближения к минимуму шаги стоит делать более аккуратными для достижения более точного результата. Во-вторых, к каждому параметру нужен индивидуальный подход.

Градиенты с разными параметрами довольно сильно отличаются. Одни параметры сильно влияют на градиент, а другие – слабо. Идея заключается в том, что скорость обучения адаптируется к каждому параметру отдельно, для этого используется информация с предыдущих уже посчитанных градиентов. На этом основан метод Adagrad(adaptive gradient). Он хранит для каждого параметра сети сумму квадратов его обновлений. Это позволяет уменьшить влияние часто изменяющихся параметров и повысить значимость редко встречающихся параметров. Обновление весов происходит следующим образом:

$$G_t \mathrel{+}= \left(\frac{\partial L}{\partial w_t}\right)^2 \tag{2}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t+\epsilon}} * \frac{\partial L}{\partial w_t} \tag{3}$$

где ϵ – сглаживающий параметр, который препятствует делению на 0.

Недостаток Adagrad в том, что G_t может увеличиваться сколько угодно, тем самым приводить к слишком маленьким обновлениям. Метод RMSProp решает эту проблему, используя усредненный по истории квадрат градиента.

Отдельного внимания заслуживает оптимизатор Adam(adaptive moment estimation), который сочетает в себе как идею накопления движения, так и изменение скорости обучения для отдельных параметров и объединяет лучшие свойства алгоритмов AdaGrad и RMSProp. Адам, как правило, быстро достигает хороших показателей при обучении. Кроме этого он неприхередлив к гиперпараметрам – не затрачивая время на исследования, можно использовать настройки по умолчанию и получать неплохие результаты.

Все эти методы будут экспериментально применены для обучения TT-слоев.

1.4 Batch normalization

На самом деле, при обучении нейронных сетей с некоторым даже не очень большим количеством скрытых слоев могут возникнуть проблемы. Слабым местом является передача выходных значений предыдущего слоя следующему за ним слою. Распределение входных данных каждого слоя меняется в процессе обучения вместе с изменением параметров предыдущего слоя. Это значительно замедляет обучение. При этом снижение этого негативного воздействия накладывает определенные требования к организации обучения в виде снижения скорости обучения и тщательной инициализации параметров. Кроме того, распределение выходов мини-батча одного нейрона может неудачно лечь на функцию активации, тем самым остановить процесс обучения. Batch normalization – метод, который нормирует входные данные слоя для каждого мини-батча. Принцип работы Batch normalization указан на рисунке 3.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Рисунок 3 - Схема batch normalization

После получения распределения с центром в 0 и дисперсией 1, над элементом ещё выполняется линейное преобразование с двумя регулируемыми параметрами γ и β, что позволяет модели выбирать оптимальные распределения для каждого скрытого слоя и дает возможность убрать батч-нормализацию, если она мешает процессу обучения. В итоге получается слой с $2m$ параметрами, который добавляется между слоями нейронной сети. Это простое преобразования на самом деле дает целый ряд преимуществ:

- стабилизация процесса обучения
- уменьшение зависимости от других слоев и начальной инициализации

- добавление регуляризационного эффекта
- ускорение обучения

1.5 Dropout

Одной из главных проблем глубоких нейронных сетей является переобучение. Этот процесс заключается в том, что модель показывает хорошие результаты только на обучающей выборке, в то время как на валидационной выборке результаты совсем не удовлетворительные. Это можно интерпретировать следующим образом: модель вместо того, чтобы искать общие закономерности для решения задачи, начинает просто подстраиваться и запоминать обучающие примеры. Для борьбы с этой проблемой существует простой метод, который эффективно показал себя на практике – Dropout. Принцип работы данного метода изображен на рисунке 4.

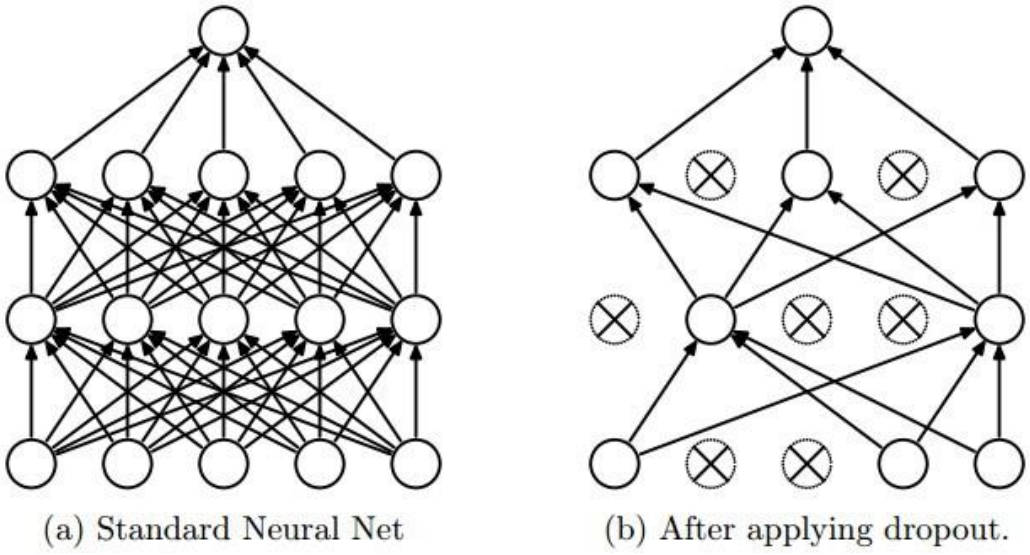

Рисунок 4 - Визуализация Dropout

Во время обучения Dropout исключает случайные нейроны из сети с заданной вероятностью. Исключение нейрона означает, что его выход, независимо от входных параметров, будет равняться нулю. Нейроны, которые были исключены, никак не влияют на процесс обучения, их веса не будут изменены при обратном распространении ошибки. Это позволяет оставшимся нейронам учиться более самостоятельно, без зависимости от исключенных нейронов. Таким образом, Dropout предотвращает совместную адаптацию(co-adaptation) нейронов во время обучения и является очень эффективным методом борьбы с переобучением. С другой стороны, на каждой итерации исключения Dropout равносильны обучению новой сети. Поэтому данный метод позволяет провести аналогию с ансамблевыми методами обучения моделей – после обучения нескольких

различных моделей вывод делается на основе усредненных результатов этих моделей, что в итоге позволяет ансамблевым моделям сглаживать ошибки друг друга. С этой точки зрения Dropout полезен даже когда переобучения модели не происходит.

2. Тензорные поезда

d-мерный тензор $\mathcal{A}$ представлен в формате тензорного поезда если для каждой размерности тензора $k = 1, \dots, d$ и для каждого значения k-ой размерности $j_k = 1, \dots n_k$, которые будем называть модами, существуют матрицы $G_d[j_k]$, такие что любой элемент исходного тензора может быть вычислен по формуле:

$$\mathcal{A}(j_1, \dots j_d) = \; G_1[j_1] * \dots * G_d[j_k] \tag{4}$$

На рисунке 5 иллюстрируется вычисление элемента для 4-х мерного тензора:

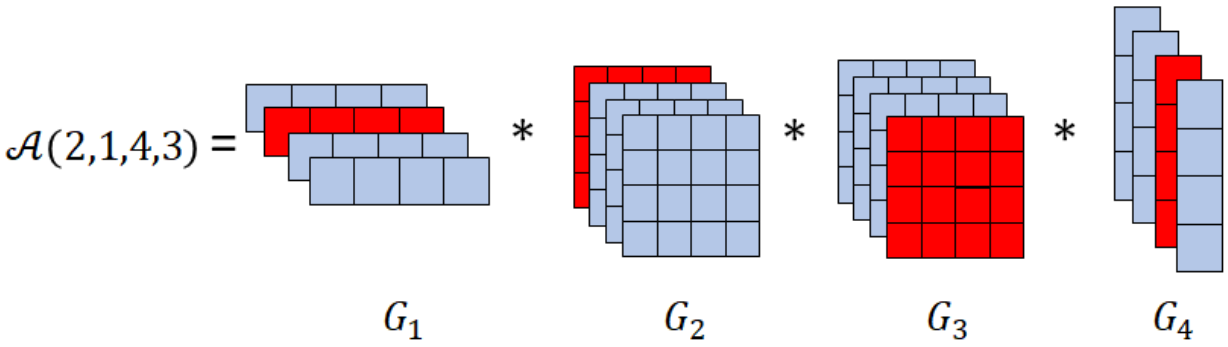

Рисунок 5 - Вычисление элемента с заданными индексами

Все матрицы одной k-ой размерности одинакового размера: $r_{k-1} \times r_k$. При этом $r_1 = r_d = 1$ для того, чтобы итоговое произведение имело размерность 1×1. Значения $\{r_k\}_{k=0}^{d}$ называются TT-рангами TT-разложения, а совокупность матриц одной размерности $\{G_i[j_k]\}_{j_k=1}^{n_k}$ называются TT-ядрами. Фактически, ядра образуют 3-х мерные тензоры.

Если обозначить $n = \max_{k=1\dots d} n_k$ и $r = \max_{k=0\dots d} r_k$, то такое представление включает в себя dnr^2 параметров вместо n^d параметров в обычном представлении. Стоит обратить внимание, что TT-формат решает проблему экспоненциального роста параметров при увеличении размерности тензора, заменяя его на линейный. Это позволяет выгодно работать с тензорами большой размерности. Кроме того, даже если в задаче не подразумевается работы с тензорами большой размерности, их можно искусственно создать, изменяя форму тензора.

Естественно, стоит рассматривать только разложения, имеющие сравнительно небольшие ранги, в противном случае данное разложение не дает никаких преимуществ. Однако не всегда выигрышное с точки зрения количества элементов разложение может существовать. С этим моментом можно справиться следующим образом: теорема

Оселедца(theorem 2.2)[1] позволяет, имея требуемую точность, построить TT-разложение с минимальными рангами. То есть за счет небольшой потери в точности возможно обеспечить снижение рангов до допустимых значений. Кроме того, после небольших модификаций, теорема работает обратным образом: задавая ранги, можно построить разложение с максимально возможной точностью. В данном случае именно такая трактовка представляет больший интерес, так как удобно задавать ранги вручную, вынося их в гиперпараметры слоя.

3. Применение тензорного поезда к полносвязному слою

3.1 Теоретическое решение

В полносвязном слое любой нейрон связан с каждым входным признаком. Работа полносвязного изображена на рисунке 6. Формально слой можно определить следующим образом:

- На вход подается вектор $x \in \mathbb{R}^M$
- Над ним выполняется линейное преобразование вида $y = Wx + b$,
- где W – матрица весов размера $N \times M$, b – вектор смещений размера N
- Результатом преобразования является вектор $y \in N$

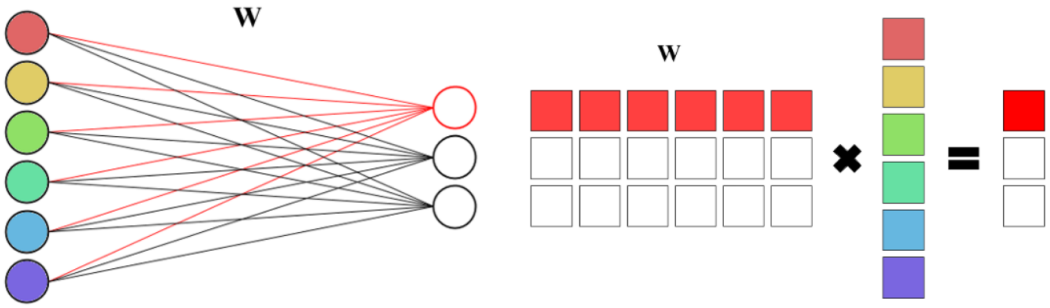

Рисунок 6 - Работа полносвязного слоя

Основную проблему составляет матрица W, содержащая большую часть параметров для обучения слоя. Именно ее представление необходимо оптимизировать, снизив количество параметров. Однако если применить TT-разложение напрямую к матрице, то такое преобразование будет эквивалентно малоранговому представлению матрицы(low-rank matrix format). Поэтому более эффективным решением будет применить специальное разложение для матрицы, предложенное в работе[2]. Оно заключается в следующем: для матрицы $W \in \mathbb{R}^{N \times M}$ раскладываются на множители ее размерности:

$$\prod_{k=1}^{d} n_k = N \ \prod_{k=1}^{d} m_k = M \tag{5}$$

Рассматривается тензор $\mathcal{W}$, имеющий размерности $(m_1 n_1) \times (m_2 n_2) \times \ldots \times (m_d n_d)$
Суммарное количество элементов матрицы W и тензора $\mathcal{W}$ совпадают, поэтому каждому элементу матрицы W ставится элемент тензора $\mathcal{W}$ по правилу:

$$W(i,j) = \mathcal{W}\big((i_1,j_1),(i_2,j_2),\ldots,(i_d,j_d)\big) \tag{6}$$

Наконец, можно представить тензор $\mathcal{W}$ в TT-формате:

$$\mathcal{W}\big((i_1, j_1), (i_2, j_2), \dots, (i_d, j_d)\big) = G_1[i_1, j_1] * G_2[i_2, j_2] * \dots * G_d[i_d, j_d] \quad (7)$$

Представив вектора x, y и b в виде тензоров размерности d

$$x(j) = \mathcal{X}(j_1, j_2, \dots, j_d) \quad y(i) = \mathcal{Y}(i_1, i_2, \dots, i_d) \quad b(i) = \mathcal{B}(i_1, i_2, \dots, i_d) \quad (8)$$

Конечное преобразование слоя выглядит следующим образом:

$$\mathcal{Y}(i_1, i_2, \dots, i_d) = \sum_{j_1 \dots j_d} G_1[i_1, j_1] * \dots * G_d[i_d, j_d] * \mathcal{X}(j_1, j_2, \dots, j_d) + \mathcal{B}(i_1, i_2, \dots, i_d) \quad (9)$$

Выполнение матрично-векторного произведения в данном формате требует вычислительной сложности $O(dr^2 m \max\{M, N\})$ в место $O(MN)$ в обычном представлении.

3.2 Тонкости практической реализации

При обучении глубоких нейронных сетей правильная инициализация весов в слоях может сильно повлиять на время обучения и темп уменьшения функции ошибки. Если задать слишком маленькие веса, то можно столкнуться с проблемой затухающего градиента, при этом обучение станет неэффективным или остановиться совсем. Слишком большие веса тоже плохо влияют на процесс обучения – модель дестабилизируется, функция потерь меняется скачкообразно, это указывает на проблему взрывающегося градиента. Большинство стандартных методов инициализации основано на нормальном распределении с нулевым смещением, а за отклонение обычно берется $1/n$, где n – количество входов в слой. Вместо нормального распределения также можно взять равномерное с граничными условиями $\left(1/n, 1/n\right)$. Однако эти методы слишком обобщены и с ними, как правило, возникают трудности при использовании нелинейных активационных функций, таких как ReLu и Sigmoid. Для решения этой проблемы существует два стандартных метода весовой инициализации. Для слоев, имеющих сигмоиду в качестве функции активации, используются нормальное распределение Ксавье со смещением, равным нулю, а отклонение вычисляется по правилу:

$$\sigma = gain \times \sqrt{\frac{2}{fan\_in + fan\_out}} \quad (10)$$

где $gain$ – коэффициент масштабируемости, fan_in и fan_out – количество входов и выходов данного слоя.

Также можно использовать равномерное распределение Ксавье $U(-a, a)$, где

$$a = gain \times \sqrt{\frac{6}{fan\_in + fan\_out}} \tag{11}$$

Для функции Relu было предложено распределение Кайминга(kaiming initialization). Веса заполняются значениями равномерного распределения $U(-a, a)$, где

$$a = gain \times \sqrt{\frac{3}{fan\_mode}} \tag{12}$$

Именно такую инициализацию весов использует PyTorch по умолчанию. Коэффициент масштабируемости для Relu составляет $\sqrt{2}$. Также PyTorch для инициализации Кайминга *fan_mode* может принимать два значения: *fan_in*, который сохраняет дисперсию весов в прямом проходе, и *fan_out*, при котором дисперсия сохраняется на обратном проходе. Аналогично можно взять инициализацию Кайминга с нормальным распределением, где смещение равно нуля, а отклонение вычисляется следующим образом:

$$\sigma = \frac{gain}{\sqrt{fan\_mode}} \tag{13}$$

Во время создания слоя создается и инициализируется вышеуказанными способами матрица весов, которая перед обучением должна быть представлена в виде TT-представления. Однако после инициализации матрица не несет в себе никакой информации, а в процессе обучения ее элементы будут обновляться. Соответственно, нет никакого смысла переходить на TT-формат, по которому с определенной точностью будет восстанавливаться исходная матрица. Все это делает применение операции TT-разложения нецелесообразной. Более того, процесс вычисления TT-рангов во время разложения требует трудоемких вычислений. В данном случае намного эффективнее создавать слои с весами, уже находящимися в нужном TT-формате, при этом инициализацию весов можно оставить прежней. Все параметры TT-разложения, такие как TT-ранги, TT-моды, необходимо вынести в гиперпараметры модели. Это позволит в полной мере контролировать процесс обучения и его результаты, зависящие от TT-представления.

Такой подход изменит вычислительный граф модели по сравнению с обычным представлением – вместо взятия матрицы весов в вычисления будут последовательно входить TT-ядра, что увеличивает число операций в вычислительном графе.

Преимущество матрицы параметров целиком заключается в том, что вычисления матрично-векторного произведения с легкостью распараллеливается на GPU. Применение TT-представления значительно усложняет задачу, так как вместо доступа к элементам матрицы за $O(1)$, необходимо производить цепочку преобразований с TT-ядрами. Таким образом, несмотря на существенное уменьшение размеров данных, такого же существенного уменьшение по времени работы достигнуть не удастся.

TT-слой задается тремя гиперпараметрами:

- in_modes – факторизация размера входа. Представляет массив np.array, где элементы являются множителями входного размера;
- out_modes – факторизация размера выхода. Представляет массив np.array, где элементы являются множителями выходного размера. Размер массива совпадает с in_modes;
- ranks – TT-ранги, которые представляют размеры ядер в TT-разложении. Представляет массив np.array, размер массива на один больше, чем размер in_modes, первый и последний элемент равны 1.

Прямой проход слоя состоит из произведения входного вектора и весов в тензорном формате с последующим добавлением вектора биаса. Создав логику прямого прохождения слоя, обратный проход реализуется фреймворком PyTotch с помощью автоматического подсчета градиентов.

Обязательной составной частью решения является включение батч-нормализации после предложенных слоев, так как сама структура вычисления слоя предполагает возможность возникновения затухающего градиента. Причем остановка обучения происходит практически сразу и значения параметров почти не отличаются от инициализированных значений. Батч-нормализация позволяет решить эту проблему, однако самой техники есть свои ограничения: большой размер батча и равновероятное попадание элемента в батч для каждого класса.

4. Экспериментальная часть

4.1 Наборы данных

В данной работе эксперименты проводились на двух популярных наборах данных для решения задач классификации: MNIST и CIFAR-10.

Набор данных MNIST представляет собой черно-белые изображения рукописных цифр размером 28×28 пикселей, его визуализацию представляет рисунок 7. Соответственно, количество классов 10 – цифры от 0 до 9. Набор данных содержит 60000 изображений для обучения и 10000 для тестирования. Все цифры нормализованы по размеру и расположены в центре изображений.

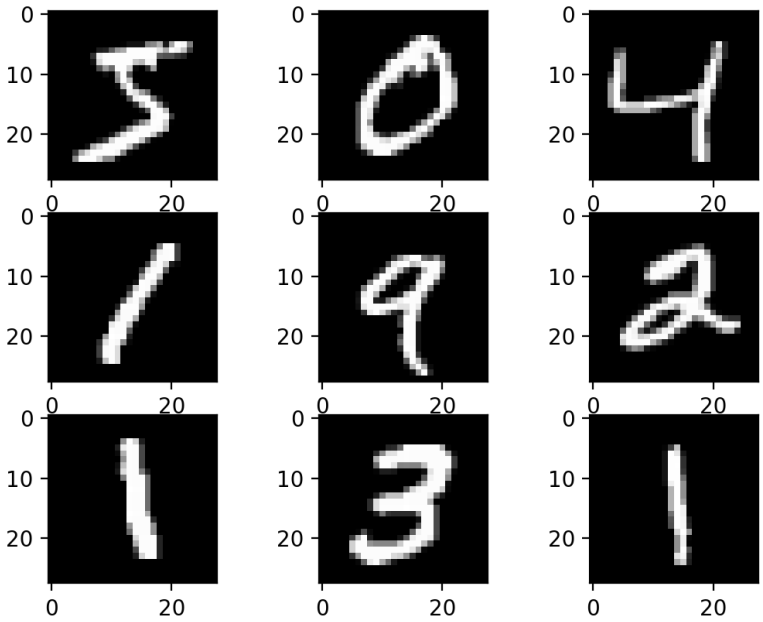

Рисунок 7 - Примеры из MNIST

Набор данных CIFAR-10 состоит из цветных изображений размера 32×32 пикселя, визуализация представлена на рисунке 8. На изображениях находятся один из десяти классов:

- самолет,
- автомобиль,
- птица,
- кошка,
- олень,

- собака,
- лягушка,
- лошадь,
- корабль,
- грузовик.

Набор данных содержит 50000 изображений для обучения и 10000 для тестирования.

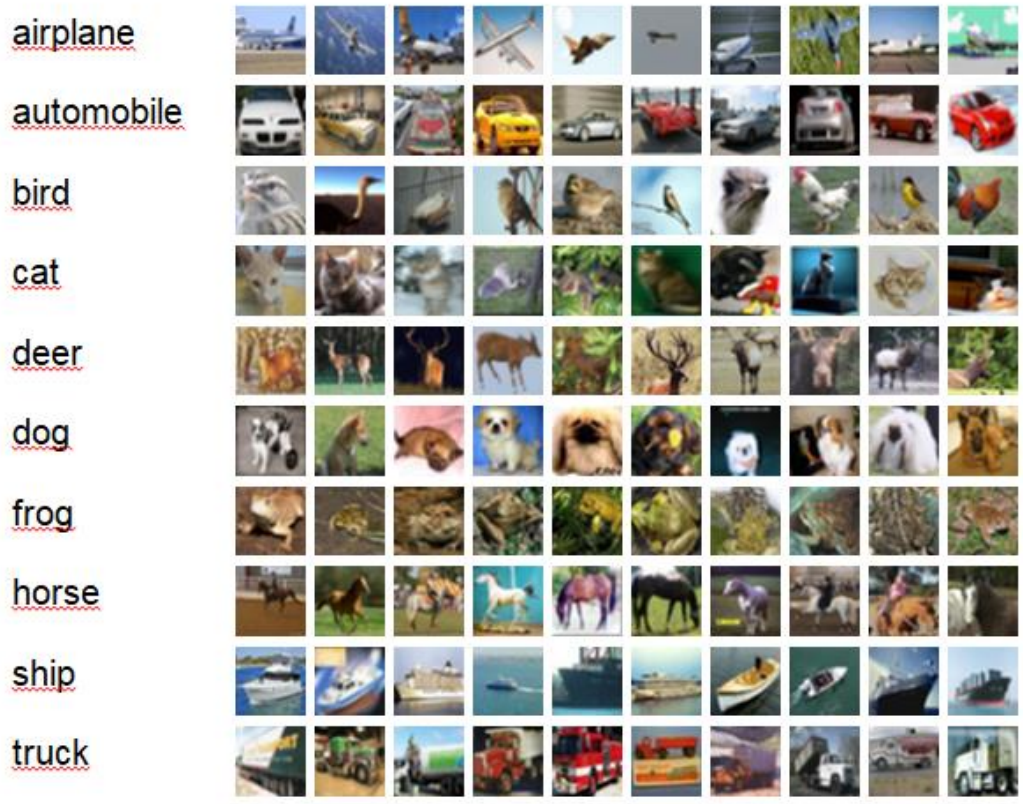

Рисунок 8 - Примеры из CIFAR-10

4.2 Эксперименты MNIST

Рисунок 9 показывает базовую архитектуру нейронной сети, которая была взята для экспериментов.

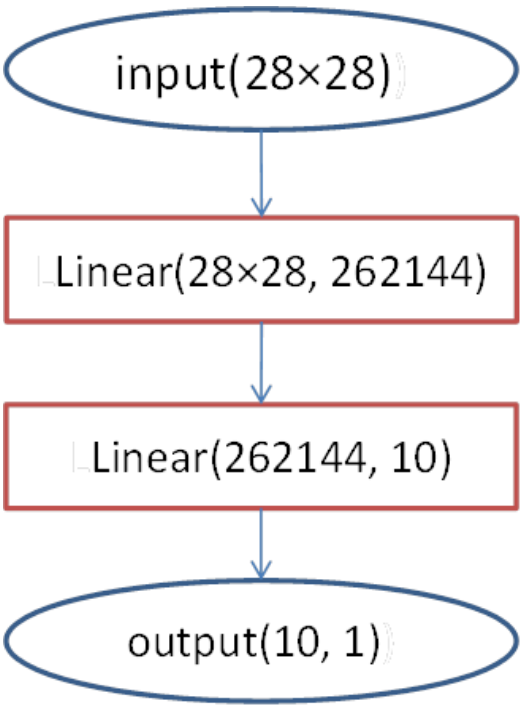

Рисунок 9 - Архитектура нейронной сети для MNIST

Модель содержит два полносвязных слоя. На рисунке 10 показаны процессы обучения обычной полносвязной модели и эквивалентной модели с TT-слоями:

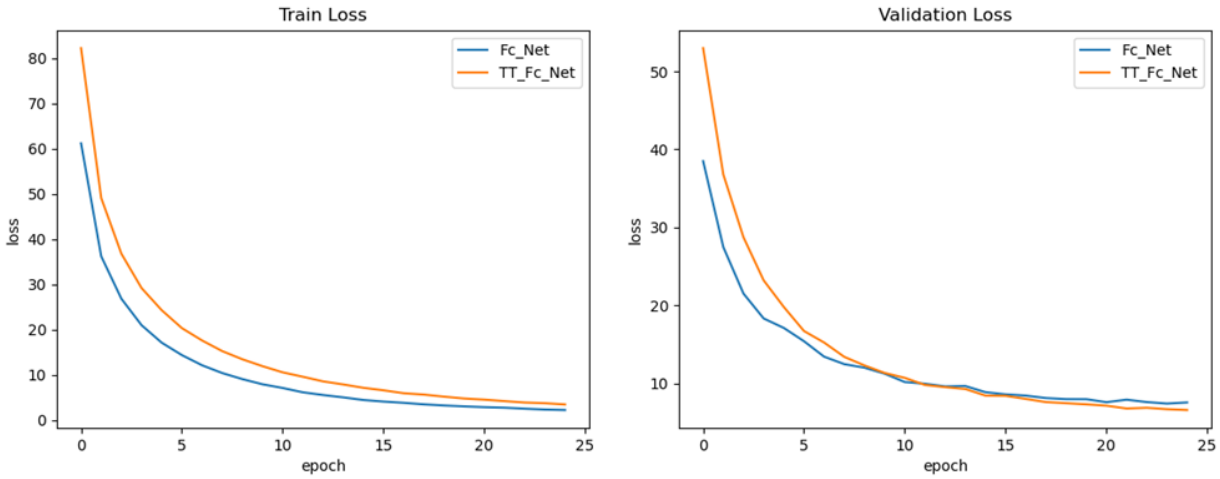

Рисунок 10 - Сравнение обучений поллносвязных и TT-полновязных сетей

Как видно из графиков, применение TT-формата на сам процесс обучения никак не повлияло. Функция ошибки у двух моделей уменьшается аналогично, как на обучающей, так и на валидационной выборке.

Обучение на данном наборе данных происходило обучение методом стохастического градиентного спуска с моментами (коэффициент момента равен 0.9) и со

скоростью обучения, равной 0.001. Остановка осуществлялась по прохождении 25 эпох обучения.

Таблица 1 - Результаты обучения на MNIST

Модель	Число обучающихся параметров	Значение функции ошибки	Метрика accuracy	Время
FC-FC	208928798	7.5404	98.38	18m 45s
TT_FC-TT_FC	**791294**	6.5720	98.59	17m 16s
TT_FC-FC	3411518	**6.5004**	**98.71**	**12m 58s**
FC-FC-FC	11638814	7.5564	98.34	**4m 29s**
TT_FC-TT_FC-TT_FC	**30574**	8.3797	98.21	5m 56s
TT_FC-TT_FC-FC	50622	**6.7396**	**98.53**	5m 20s

В таблице 1 указаны результаты обучения моделей на MNIST. Первая модель из обычных полносвязных слоев, очевидно слишком сложная для решения данной задачи и содержит слишком много параметров. В этом случае уменьшение параметров на три порядка с помощью TT-полносвязных слоев еще и дает выигрыш по точности предсказания модели. Также хорошем решением является оставить последний слой модели полносвязным – он содержит всего 10 выходов и сравнительно небольшой по сравнению с первым слоем, но вместе с тем именно по выходу этого слоя совершается предсказание модели, поэтому на этом этапе желательно по максимуму сохранять точность. Если сравнивать модели по времени обучения, то можно отметить, что TT-слои, несмотря на меньшее количество параметров, значительно проигрывают обычным полносвязным слоям, за исключением одного случая, когда полносвязный слой становиться настолько большим, что матрично-векторное произведение не может быть эффективно распараллелено на GPU. В этом случае многократная разность в количестве параметров дает преимущество TT-слоям. Все предположения подтверждаются и для моделей из трех слоев.

В целом, по итогам обучения TT-слои обеспечили многократное уменьшение количества параметров(более чем в 250 раз), при этом итоговая точность предсказания снизилась незначительно(на десятые процента).

4.3 Эксперименты CIFAR-10

Считается, что полносвязные модели не подходят для решения задачи классификации изображений в связи со следующими недостатками:

- Для каждого нейрона первого слоя количество весов должно равняться количеству пикселей входного изображения. Поэтому большой размер входного изображения быстро приводит к чрезмерным размерам сети;
- Сеть не обрабатывает информацию о соседних по расположению участках изображения. Это не позволяет определять объект, если на изображениях он встречается разного масштаба или смещен в одну из сторон.

Действительно, применяя все техники, описанные методы оптимизации, финальная точность, которую удалось получить полносвязной сетью на CIFAR-10, составила 59.2%, что однозначно нельзя назвать качественной классификацией.

Для решения данной задачи была взята модель с пятью TT-полносвязными слоями со следующей архитектурой, изображенной на рисунке 11.

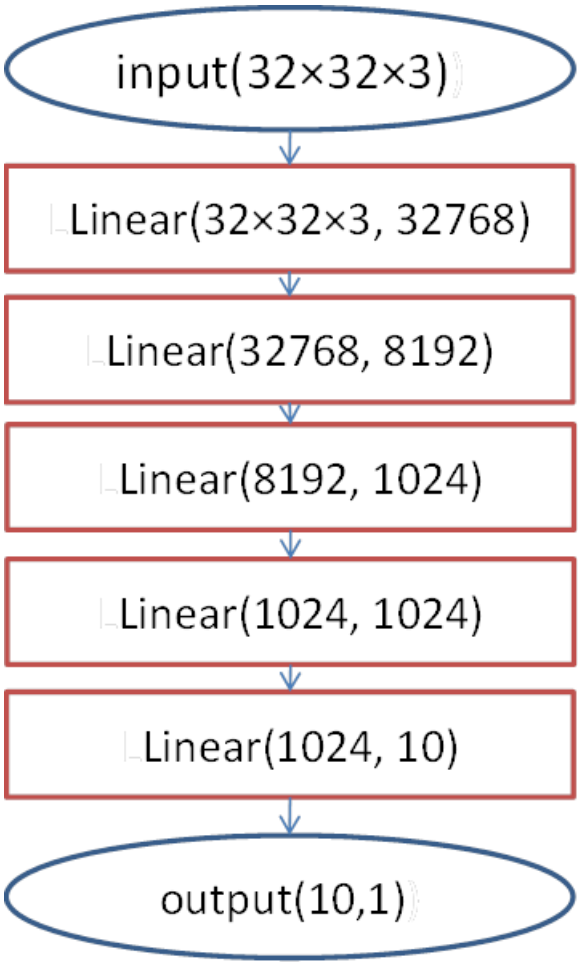

Рисунок 11 - Архитектура нейронной сети для CIFAR-10

Благодаря TT-формату модель содержит 146814 параметров обучения. Начальные эксперименты состояли в обучении моделей методом стохастического градиентного спуска с разными скоростями обучения и ее снижением каждые 10 эпох, результаты которого представлены на рисунке 12. Обучение останавливалось по прохождении 30 эпох.

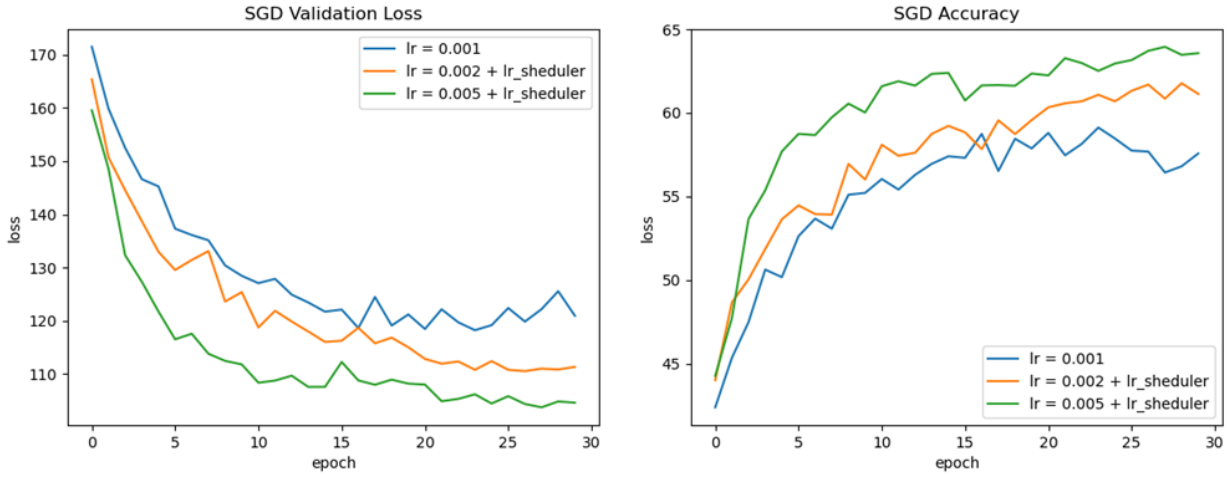

Рисунок 12 - Обучение методом SGD

В итоге удалось достичь точности в 63.95%, что уже лучше обычной полносвязной модели.

Далее были применены другие оптимизаторы: Adam, RMSprop, Adadelta. Сравнение оптимизаторов иллюстрирует рисунок 13.

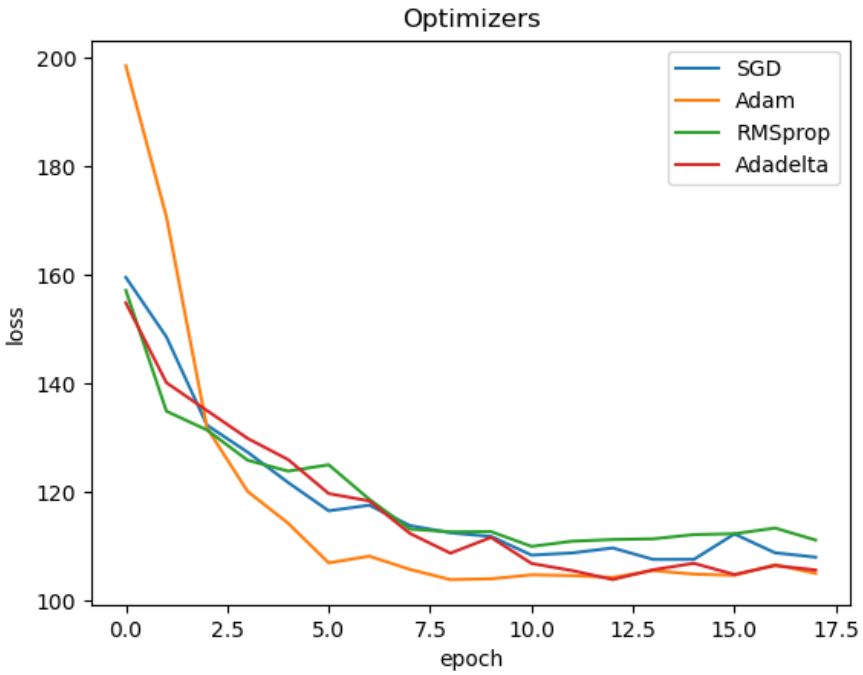

Рисунок 13 - Сравнение оптимизаторов

Лучшие результаты показали Adam и Adadelta. Далее на рисунке 14 будет продемонстрировано влияние гиперпараметров TT-слоя на точность модели.

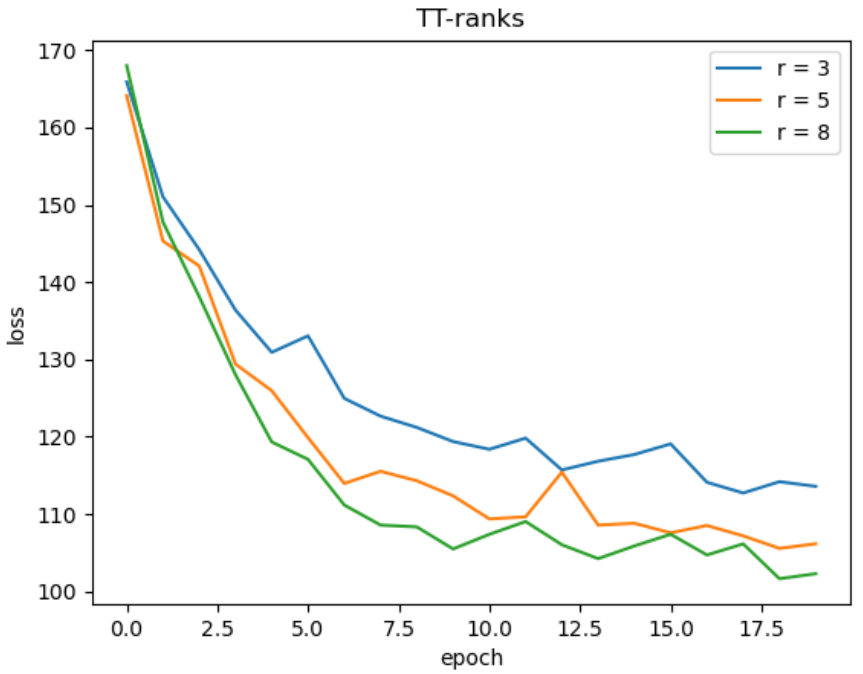

Рисунок 14 - Применение различных TT-рангов

Увеличение TT-рангов увеличивает число обучаемых параметров модели, позволяя достичь большей точности предсказания. Взамен увеличивается размер модели и время обучения. Далее произведено сравнение разных факторизаций входной и выходной размерности TT-слоев на рисунке 15.

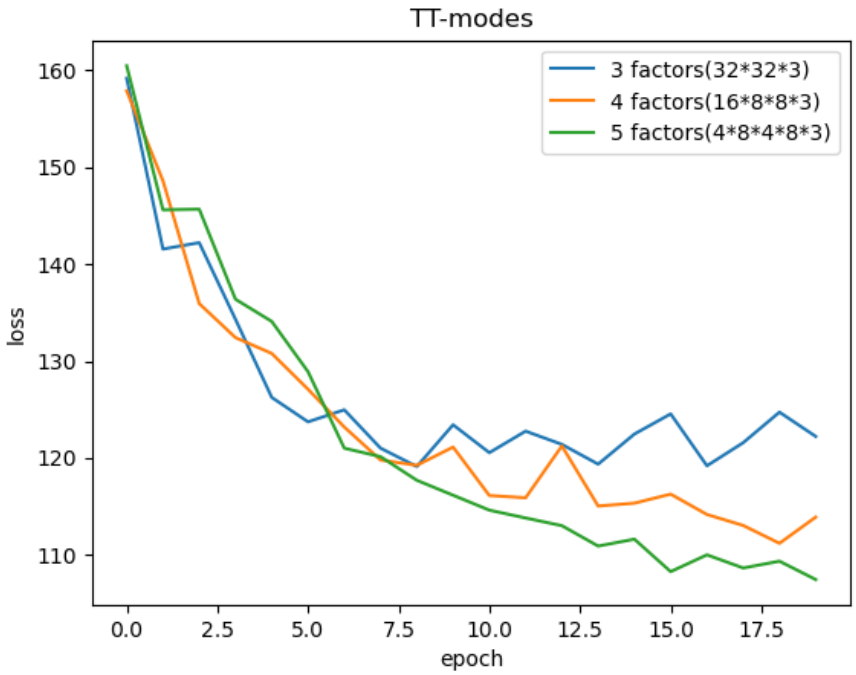

Рисунок 15 - Применение различных TT-мод

Повышение множителей в факторизации «удлиняет» тензорный поезд, последовательно добавляя TT-ядра, что, как выяснилось, увеличивает точность предсказания модели.

Применяя весь полученный опыт и используя все оптимальные методы, за период обучения в 100 эпох удалось достигнуть результата в 72.92%, результаты показаны в таблице 2. Для обучения моделей применялся оптимизатор Adadelta и техники Batch-normalization и Dropout.

Таблица 2 - Результаты обучения на CIFAR-10

Модель	Число обучающихся параметров	Время	Метрика accuracy
4 × TT_FC – FC	146814	30m 18s	71.67
4 × TT_FC – FC	225278	31m 34s	72.92

В итоге удалось повысить точность примерно на 13% процентов по сравнению с обычной полносвязной моделью.

ЗАКЛЮЧЕНИЕ

В выпускной квалификационной работе удалось показать, что применение TT-формата к полносвязным слоям действительно способно значительно сократить число обучающихся параметров, при этом точность модели остается на прежнем уровне. Кроме того, данный формат намного лучше подходит при работе с изображениями, по сравнению с полносвязными слоями, и позволяет ощутимо повысить качество модели.

В работе экспериментально показана возможность обучения нейронных сетей, состоящих и слоев в TT-формате, вычислены и продемонстрированы основные метрики процесса обучения, а также выделены методы, наиболее подходящие для обучения данных сетей.

Реализация с применением фреймворка PyTorch позволяет любому пользователю добавлять полносвязные слои в TT-формате в свои нейронные сети и производить их обучение.

TT-полносвязный слой имеет ряд существенных преимуществ относительно обычного полносвязного слоя: значительно меньшее число параметров и более высокая стабильность на этапе обучения. Кроме того, данный метод имеет математическое обоснование. Однако некоторые недостатки метода не позволяют ему получить широкое распространение. Во-первых, несмотря на математическую связь тензора с его TT-форматом, после обучения они уже не имеют ничего общего, так как TT-формат изменяет граф вычислений и число параметров, что изменяет саму задачу минимизации функции потерь. Во-вторых, эффективность по памяти TT-формата все-таки не дает ему эффективности по времени, а именно время обучения на сегодняшний день является ключевым фактором в процессе обучения глубоких нейронных сетей.

Стоит отметить, что TT-формат не ограничивается лишь полносвязными слоями и может быть успешно применен и к другим архитектурам нейронной сети, все так же решая задачу уменьшения числа параметров.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Oseledets I. V. (2011). Tensor–Train Decomposition. SIAM J. Scientific Computing. Т. 33, № 5. С. 2295–2317.

2. Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, Dmitry Vetrov. (2015). Tensorizing neural networks. Advances in Neural Information Processing Systems, 2015. С. 442–450.

3. A. Novikov, A. Rodomanov, A. Osokin, and D. Vetrov. (2014). Putting MRFs on a Tensor Train. Proceedings of the International Conference on Machine Learning (ICML), 2014, pp. 811–819.

4. T. Garipov, D. Podoprikhin, A. Novikov, D. Vetrov. (2016). Ultimate tensorization: compressing convolutional and FC layers alike. arXiv:1611.03214

5. Dingheng Wang, Zhao Guangshe, Li Guoqi, Deng Lei, Wu Yang. (2020). Compressing 3DCNNs based on tensor train decomposition. arXiv:1912.03647

6. Tjandra A., Sakti S., Nakamura S. (2017). Compressing recurrent neural network with tensor train. Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2017, pp. 4451–4458.

7. Sainath Tara, Kingsbury Brian, Sindhwani Vikas, Arisoy Ebru, Ramabhadran Bhuvana. (2013). Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets. Proceedings of the 1988 International Conference on. 6655-6659. 10.1109/ICASSP.2013.6638949.

8. Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, Yixin Chen. Compressing Neural Networks with the Hashing Trick. (2015). Proceedings of the 32nd International Conference on Machine Learning, PMLR 37:2285-2294, 2015.

9. Xue J., Li J., Gong Y. (2013). Restructuring of deep neural network acoustic models with singular value decomposition. Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH. 2365-2369.

10. Ioffe Sergey, Szegedy Christian. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Proceedings of the 32nd International Conference on Machine Learning (ICML-15). С. 448–456.

11. Каширина И. Л., Демченко М. В. (2018). Исследование и сравнительный анализ методов оптимизации, используемых при обучении нейронных сетей. Вестник ВГУ. Серия: Системный анализ и информационные технологии, (4), 123-132.

12. Kentan Doshi. (2021). Neural network optimizers made simple - Core algorithms and why they are needed. Towards Data Science.

13. Srivastava Nitish, Hinton Geoffrey, Krizhevsky Alex, Sutskever Ilya, Salakhutdinov Ruslan. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research. 15. 1929-1958.

14. M. Denil, B. Shakibi, L. Dinh, M. Ranzato, N. de Freitas. (2013). Predicting parameters in deep learning. In Advances in Neural Information Processing Systems 26 (NIPS), 2013, pp. 2148–2156.

15. E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In Advances in Neural Information Processing Systems 27 (NIPS), 2014, pp. 1269–1277.

16. Lin Zhouhan, Memisevic Roland, Konda, Kishore. (2015). How far can we go without convolution: Improving fully-connected networks. arXiv:1511.02580.

ПРИЛОЖЕНИЕ

Реализация TT-полносвязного слоя

```
# TT-Liner layer for PyTorch

import numpy as np
import torch
import torch.nn as nn


class tt_LinearV2(nn.Module):

    # initialization:
    ## input:
    ### in_modes - numpy array of input modes
    ### out_modes - numpy array of output modes
    ### ranks - numpy array of tt-ranks, first and last element must be 1
    ### bias - True if necessary to add a vector of bias(deafult True)

    def __init__(self, in_modes, out_modes, ranks, bias=True):
        super().__init__()
        self.bias = bias
        self.in_modes = in_modes
        self.out_modes = out_modes
        self.ranks = ranks
        self.d = in_modes.size
        self.biases =
torch.nn.Parameter(torch.randn(np.prod(out_modes)))
        self.cores =
nn.ParameterList([torch.nn.Parameter(torch.randn(self.in_modes[i],
self.ranks[i],
            self.ranks[i+1], self.out_modes[i])) for i in range(self.d)])
        self.reset_parameters()

    # initialize parametrs of tt-cores by kaiming uniform initialization

    def reset_parameters(self):
        if self.d != 5:
            print('length of the in_modes must be 5, not
{}'.format(self.d))
            exit(1)
        for i in range(self.d):
            torch.nn.init.kaiming_uniform_(self.cores[i], a=np.sqrt(5))
        fan_in, _ =
torch.nn.init._calculate_fan_in_and_fan_out(self.cores[0])
        bound = 1 / np.sqrt(fan_in)
        torch.nn.init.uniform_(self.biases, -bound, bound)

    #forward pass
    ## vector-by-matrix product in tt-format implemenr=ted by einsum
operation

    def forward(self, input):
```

```
        input_rsh = torch.reshape(input, tuple(np.append([-1], self.in_modes)))

        # n index of sample in the batch,
        # a, b, c, d, e - indices corresponding to input modes
        # h, i, j, k, l, m - indices corresponding to tt ranks
        # v, w, x, y, z - indices corresponding to output modes

        out = torch.einsum("nabcde,ahiv,bijw,cjkx,dkly,elmz", input_rsh,
            self.cores[0], self.cores[1], self.cores[2], self.cores[3], self.cores[4])
        out = torch.reshape(out, (-1, np.prod(self.out_modes)))
        if self.bias:
            out = torch.add(out, self.biases)

        return out
```

Модели и обучение на MNIST

```
#experiments with MNIST

import numpy as np
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import tt_linear
import tt_linearV2

import time
import os
import copy
from prettytable import PrettyTable

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def count_parameters(model):
    table = PrettyTable(["Modules", "Parameters"])
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad: continue
        params = parameter.numel()
        table.add_row([name, params])
```

```
27.         total_params+=params
28.     print(table)
29.     print(f"Total Trainable Params: {total_params}")
30.     return total_params
31.
32. class TT_Net(nn.Module):
33.     def __init__(self):
34.         super().__init__()
35.         self.fc1 = tt_linearV2.tt_LinearV2(np.array([2, 2, 7, 4, 7]),
np.array([8, 16, 16, 16, 8]), np.array([1, 4, 4, 4, 4, 1]))
36.         self.fc2 = tt_linearV2.tt_LinearV2(np.array([8, 16, 16, 16,
8]), np.array([5, 2, 1, 1, 1]), np.array([1, 4, 4, 4, 4, 1]))
37.         self.bn1 = nn.BatchNorm1d(262144)
38.         self.bn2 = nn.BatchNorm1d(10)
39.
40.     def forward(self, x):
41.         x = self.fc1(x)
42.         x = F.relu(self.bn1(x))
43.         x = self.fc2(x)
44.         x = F.relu(self.bn2(x))
45.         return F.log_softmax(x, dim=1)
46.
47. class Usual_Net(nn.Module):
48.     def __init__(self):
49.         super().__init__()
50.         self.bn1 = nn.BatchNorm1d(262144)
51.         self.bn2 = nn.BatchNorm1d(10)
52.         self.fc1 = nn.Linear(28 * 28, 262144)
53.         self.fc2 = nn.Linear(262144, 10)
54.
55.     def forward(self, x):
56.         x = self.fc1(x)
57.         x = F.relu(self.bn1(x))
58.         x = self.fc2(x)
59.         x = F.relu(self.bn2(x))
60.         return F.log_softmax(x, dim=1)
61.
62. def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
63.     since = time.time()
64.     best_model_wts = copy.deepcopy(model.state_dict())
```

```
65.     best_acc = 0.0
66.
67.     lss_train = []
68.     lss_test = []
69.
70.     for epoch in range(num_epochs):
71.         print('Epoch {}/{}'.format(epoch, num_epochs - 1))
72.         print('-' * 10)
73.
74.         for phase in ['train', 'test']:
75.             if phase == 'train':
76.                 model.train()
77.             else:
78.                 model.eval()
79.
80.             running_loss = 0.0
81.             running_corrects = 0
82.
83.             cur_set = trainset if phase == 'train' else testset
84.             for inputs, labels in cur_set:
85.                 inputs = inputs.to(device)
86.                 labels = labels.to(device)
87.                 optimizer.zero_grad()
88.
89.                 with torch.set_grad_enabled(phase == 'train'):
90.                     outputs = model(inputs.reshape(-1, 28*28))
91.                     _, preds = torch.max(outputs, 1)
92.                     loss = criterion(outputs, labels)
93.
94.                     if phase == 'train':
95.                         loss.backward()
96.                         optimizer.step()
97.
98.                 running_loss += loss.item() * inputs.size(0)
99.                 running_corrects += torch.sum(preds == labels.data)
100.
101.                if phase == 'train' and scheduler != None:
102.                    scheduler.step()
103.
104.                epoch_loss = running_loss / len(cur_set)
```

```
105.                 epoch_acc = running_corrects.double() / len(cur_set)
106.
107.                 cur_loss = lss_train if phase == 'train' else lss_test
108.                 cur_loss += [epoch_loss]
109.
110.                 print('{} Loss: {:.4f} Acc: {:.4f}'.format(
111.                     phase, epoch_loss, epoch_acc))
112.
113.                 if phase == 'test' and epoch_acc > best_acc:
114.                     best_acc = epoch_acc
115.                     best_model_wts = copy.deepcopy(model.state_dict())
116.
117.             print()
118.
119.         time_elapsed = time.time() - since
120.         print('Training complete in {:.0f}m {:.0f}s'.format(
121.             time_elapsed // 60, time_elapsed % 60))
122.         print('Best val Acc: {:4f}'.format(best_acc))
123.         #print('train: {}'.format(lss_train))
124.         #print('test: {}'.format(lss_test))
125.
126.         model.load_state_dict(best_model_wts)
127.         return model
128.
129.     if __name__ == '__main__':
130.         train = datasets.MNIST('', train=True, download=True,
131.                                transform=transforms.Compose([
132.                                    transforms.ToTensor()
133.                                ]))
134.         test = datasets.MNIST('', train=False, download=True,
135.                               transform=transforms.Compose([
136.                                   transforms.ToTensor()
137.                               ]))
138.
139.
140.         trainset = torch.utils.data.DataLoader(train, batch_size=100,
shuffle=True)
141.         testset = torch.utils.data.DataLoader(test, batch_size=100,
shuffle=False)
142.
```

```
143.          tt_model = TT_Net()
144.          count_parameters(tt_model)
145.          tt_model.to(device)
146.          criterion = nn.CrossEntropyLoss()
147.          optimizer = optim.SGD(tt_model.parameters(), lr=0.001,
momentum=0.8)
148.          exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer,
step_size=10, gamma=0.1)
149.          tt_model = train_model(tt_model, criterion, optimizer, None,
num_epochs=25)
150.          #for n, p in net.named_parameters():
151.          #   print(n, p)
```

Модели и обучение на CIFAR-10

```
1.  #experiments with CIFAR-10
2.
3.  import numpy as np
4.  import torch
5.  from torch.autograd import Variable
6.  import torch.nn as nn
7.  import torch.nn.functional as F
8.  import torch.optim as optim
9.  from torchvision import datasets, transforms
10. import tt_linear
11. import tt_linearV2
12.
13. import time
14. import os
15. import copy
16. from prettytable import PrettyTable
17. import torch.onnx
18.
19. device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
20.
21. def count_parameters(model):
22.     table = PrettyTable(["Modules", "Parameters"])
23.     total_params = 0
24.     for name, parameter in model.named_parameters():
25.         if not parameter.requires_grad: continue
```

```
        params = parameter.numel()
        table.add_row([name, params])
        total_params+=params
    print(table)
    print(f"Total Trainable Params: {total_params}")
    return total_params

class TT_Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = tt_linearV2.tt_LinearV2(np.array([4, 8, 4, 8, 3]), np.array([8, 8, 8, 8, 8]), np.array([1, 4, 4, 4, 4, 1]))
        self.fc2 = tt_linearV2.tt_LinearV2(np.array([8, 8, 8, 8, 8]), np.array([8, 4, 8, 4, 8]), np.array([1, 4, 4, 4, 4, 1]))
        self.fc3 = tt_linearV2.tt_LinearV2(np.array([8, 4, 8, 4, 8]), np.array([4, 4, 4, 4, 4]), np.array([1, 4, 4, 4, 4, 1]))
        self.fc4 = tt_linearV2.tt_LinearV2(np.array([4, 4, 4, 4, 4]), np.array([4, 4, 4, 4, 4]), np.array([1, 4, 4, 4, 4, 1]))
        self.fc5 = tt_linearV2.tt_LinearV2(np.array([4, 4, 4, 4, 4]), np.array([5, 2, 1, 1, 1]), np.array([1, 4, 4, 4, 4, 1]))
        #self.fc5 = nn.Linear(1024, 10)
        self.bn1 = nn.BatchNorm1d(32768)
        self.bn2 = nn.BatchNorm1d(8192)
        self.bn3 = nn.BatchNorm1d(1024)
        self.bn4 = nn.BatchNorm1d(1024)
        self.bn5 = nn.BatchNorm1d(10)
        self.drop = nn.Dropout(0.2)

    def forward(self, x):
        x = self.drop(F.relu(self.bn1(self.fc1(x))))
        x = self.drop(F.relu(self.bn2(self.fc2(x))))
        x = self.drop(F.relu(self.bn3(self.fc3(x))))
        x = self.drop(F.relu(self.bn4(self.fc4(x))))
        x = F.relu(self.bn5(self.fc5(x)))
        return F.log_softmax(x, dim=1)

class Usual_Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(32*32*3, 4096)
```

```
61.         self.fc2 = nn.Linear(4096, 4096)
62.         self.fc3 = nn.Linear(4096, 1024)
63.         self.fc4 = nn.Linear(1024, 10)
64.         self.bn1 = nn.BatchNorm1d(4096)
65.         self.bn2 = nn.BatchNorm1d(4096)
66.         self.bn3 = nn.BatchNorm1d(1024)
67.         self.bn4 = nn.BatchNorm1d(10)
68.         self.drop = nn.Dropout(0.2)
69.
70.     def forward(self, x):
71.         x = F.relu(self.bn1(self.fc1(x)))
72.         x = F.relu(self.bn2(self.fc2(x)))
73.         x = F.relu(self.bn3(self.fc3(x)))
74.         x = F.relu(self.bn4(self.fc4(x)))
75.         return F.log_softmax(x, dim=1)
76.
77. def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
78.     since = time.time()
79.     best_model_wts = copy.deepcopy(model.state_dict())
80.     best_acc = 0.0
81.
82.     lss_train = []
83.     lss_test = []
84.     acc_test = []
85.
86.     for epoch in range(num_epochs):
87.         print('Epoch {}/{}'.format(epoch, num_epochs - 1))
88.         print('-' * 10)
89.
90.         for phase in ['train', 'test']:
91.             if phase == 'train':
92.                 model.train()
93.             else:
94.                 model.eval()
95.
96.             running_loss = 0.0
97.             running_corrects = 0
98.
99.             cur_set = trainset if phase == 'train' else testset
100.                 for inputs, labels in cur_set:
```

```
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    optimizer.zero_grad()

                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs.reshape(-1, 32*32*3))
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds ==
labels.data)

                if phase == 'train' and scheduler != None:
                    scheduler.step()

                epoch_loss = running_loss / len(cur_set)
                epoch_acc = running_corrects.double() / len(cur_set)

                cur_loss = lss_train if phase == 'train' else lss_test
                cur_loss += [epoch_loss]
                if phase == 'test':
                    acc_test += [epoch_acc.item()]

                print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                    phase, epoch_loss, epoch_acc))

                if phase == 'test' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(model.state_dict())

            print()

        time_elapsed = time.time() - since
        print('Training complete in {:.0f}m {:.0f}s'.format(
            time_elapsed // 60, time_elapsed % 60))
```

```
140.          print('Best val Acc: {:4f}'.format(best_acc))
141.
142.          #print('train: {}'.format(lss_train))
143.          #print('test: {}'.format(lss_test))
144.          #print('acc: {}'.format(acc_test))
145.
146.          model.load_state_dict(best_model_wts)
147.          return model
148.
149.      if __name__ == '__main__':
150.
151.          train = datasets.CIFAR10('./datasets', train=True,
download=False,
152.                                   transform=transforms.Compose([
153.                                       transforms.ToTensor(),
154.                                   ]))
155.          test = datasets.CIFAR10('./datasets', train=False,
download=False,
156.                                   transform=transforms.Compose([
157.                                       transforms.ToTensor()
158.                                   ]))
159.
160.          transf = transforms.Compose([
161.              ])
162.
163.          trainset = torch.utils.data.DataLoader(train, batch_size=100,
shuffle=True)
164.          testset = torch.utils.data.DataLoader(test, batch_size=100,
shuffle=False)
165.
166.          tt_model = TT_Net()
167.          count parameters(tt model)
168.          tt model.to(device)
169.          criterion = nn.CrossEntropyLoss()
170.          optimizer = optim.SGD(tt_model.parameters(), lr=0.005,
momentum=0.9)
171.          #optimizer = optim.RMSprop(tt_model.parameters(), lr=0.01,
momentum=0.0)
172.          #optimizer = optim.Adadelta(tt_model.parameters())
```

```
173.          #optimizer = optim.Adam(tt_model.parameters(), lr=0.004,
betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)
174.          exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer,
step_size=10, gamma=0.6)
175.          tt_model = train_model(tt_model, criterion, optimizer, None,
num_epochs=30)
176.          #for n, p in net.named_parameters():
177.         #   print(n, p)
```