РЕФЕРАТ

Выпускная квалификационная работа бакалавра содержит 41 страницу, 39 рисунков, 18 использованных источников, 1 приложение.

НЕЙРОННЫЕ СЕТИ, ДИФФЕРЕНЦИАЛЬНЫЕ УРАВНЕНИЯ, ОСТАТОЧНЫЕ СЕТИ, СРАВНЕНИЕ СЕТЕЙ, PYTHON, PYTORCH, СВЕРТОЧНЫЕ СЕТИ, ГИБРИДНЫЙ ПОДХОД

Выпускная квалификационная работа посвящена сравнению характеристик сетей на основе дифференциальных уравнений с остаточными сетями и сетями сочетающими в себе эти подходы.

В теоретической части рассматриваются общие подходы к классификации изображений с помощью сверточных сетей, описывается устройство таких типов сетей, как ResNet и ODENet.

В практической части рассматривается реализация сетей ResNet, ODENet, гибридного подхода и сравнение их параметров.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На сегодняшний день нейронные сети являются одним из передовых ответвлений науки и позволяют решать многие задачи, например, распознавание образов, классификация и кластеризация. Подобные задачи решаются нейронными сетями практически в каждой части нашей жизни, в том числе в беспилотных автомобилях, распознавателях лиц, интерполяции различных научных наблюдений, рекомендательных системах и даже в браузерах.

Причём в отличие от более старых методов подобные системы вполне способны работать и в реальном времени, что прибавляет им практичности. К сожалению, многие классические подходы обучения подобных сетей довольно быстро устаревают, что заставляет авторов искать более эффективные методы обучения, для либо же ускорения моделей или их усложнения, благодаря использованию новых базовых блоков моделей с упрощенной структурой, но с эффективностью на уровне предшественников.

Сегодня существует множество различных подходов, для оптимизации сетей, в том числе улучшения работы с памятью и увеличение гибкости работы сети. Кроме того, известно, что многие оптимизации можно осуществить с помощью математический методов и одним из таких подходов является использование слоёв, рассчитывающих следующее состояние сети с помощью дифференциальных уравнений, которые и позволяют реализовать описанные выше оптимизации.

Целью работы является проведение анализа производительности и прочих параметров нейронных сетей на основе дифференциальных уравнений, а также рассматривание их эффективности в задаче классификации изображений датасета CIFAR-10, по сравнению с архитектурой ResNet и комбинированном подходе.

Для достижения цели необходимо решить следующие задачи:

1) Рассмотреть существующие подходы реализации сетей на основе дифференциальных уравнений (ODENet) для классификации изображений;
2) Изучить подход ResNet;
3) Выбрать фреймворк, который позволит реализовать необходимый функционал при обучении и тестировании;
4) Определить структуру сети для ODENet, ResNet и гибридного подхода;
5) Реализовать экспериментальные сети с использованием выбранного фреймворка;
6) Провести тесты производительности, памяти и точности полученных сетей;
7) Сделать выводы по полученным данным.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Решение задачи классификации изображений

1.1.1 Сверточные сети

В настоящее время для задач классификации картинок применяют в основном сверточные нейронные сети как наиболее эффективные. Основная идея таких сетей заключается в переходе от конкретных признаков объекта на изображении к его более абстрактным деталям.

Слой нейронной сети - это набор операций применяемых к его входному сигналу.

Прямой проход в нейронной сети - это процесс прохождения входного сигнала через один или несколько слоев нейронной сети, до получения на выходе конкретного результата работы сети.

Обратный проход в нейронной сети - это процесс прохождения по сети в обратную сторону, с изменение параметров модели: весов и смещений слоев.

Метод обратного распространения ошибки - алгоритм вычисления градиента при распространении сигнала от выхода сети ко входу (при обратном проходе).

Канал изображения - это массив значений, соответствующих одному пикселю и определяющий его свойства.

Ядро свертки - это матрица чисел, называемых весами, которые подстраиваются под поиск определенных частей на изображении. Ядра сверток применяются в фильтрах, которые являются набором таких ядер (или одним ядром).

Фильтр перемещают по изображению и определяют есть ли определенная деталь объекта на данной части изображения. Чтобы получить ответ на данный вопрос применяется операция свертки, представляющая собой ничто иное как сумму произведений элементов фильтра и матрицы входных сигналов.

Дополнительно применяется дополнительный обучаемый вес в качестве сдвига(bias). Вовремя операции свертки изображения естественным образом уменьшаются и пиксели находящиеся по краям изображения не участвуют в большом количестве, поэтому для сохранения размера изображения и увеличения "области покрытия" в сверточных слоях изображение дополняется до входных размеров(padding), например добавлением незначащих пикселей по краям изображения. Кроме того в сверточных слоях применяют операцию сдвига(stride), что позволяет пропустить некоторое количество шагов при перемещении ядра и уменьшить входное изображение. Альтернативой страйду является использование пулингового(subsampling) слоя, который по определенному алгоритму сворачивает соседние пиксели в один, уменьшая размер входного изображения. Рассмотрим пример классического устройства сверточной сети для классификации изображений на рисунке 1.1:

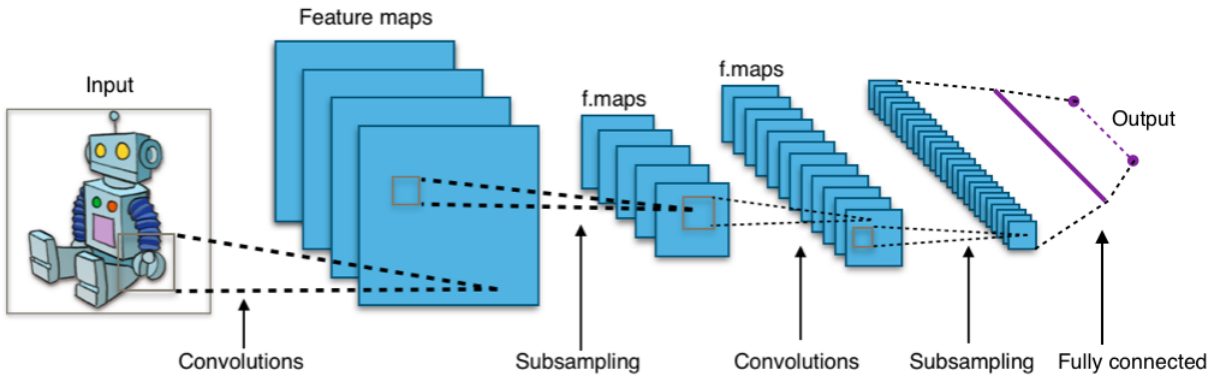

Рисунок 1.1 — Пример сверточной сети

Как видно из изображения, кроме уже упомянутых слоёв присутствует полносвязный линейный слой, в который передается развернутая матрица предыдущего сверточного слоя, таким образом на выходе линейного слоя получается сигналы полученные от различных нейронов сети, соответствующие каждому классу.

Функция потерь - функция позволяющая оценить разницу между полученным результатом от сети и правильным ответом.

Для задачи многоклассовой классификации в качестве функции потерь используют мультиклассовую кросс-энтропию. В таком случае каждому классу присваивается свой уникальный номер и минимизируется разница между фактическим значением вероятности принадлежности объекта к определенному классу и предсказанной моделью вероятностью. Формульно такая функция потерь представляется как сумма потерь по каждому классу:

$$crossentropyloss = -\sum_{c=1}^{n} y_{o,c} ln(p_{o,c}) \tag{1.1}$$

где n - число классов,

c - номер класса,

o - текущий объект классификации,

y - истинная вероятность принадлежности объекта о, классу c,

p - предсказанная вероятность принадлежности o классу c.

Если рассматривать как работает такая функция потерь для конкретного объекта можно посмотреть на следующий график на рисунке 1.2:

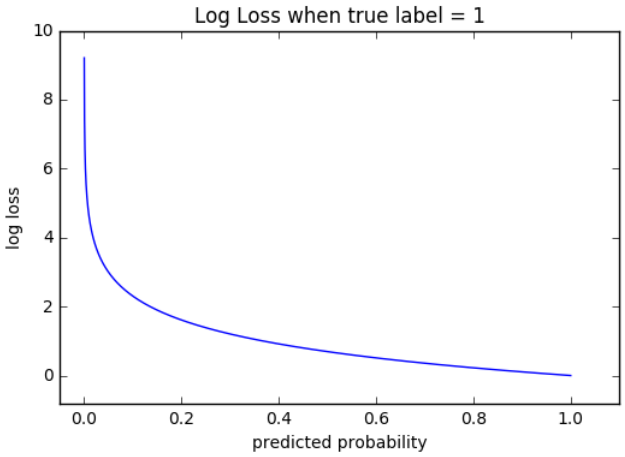

Рисунок 1.2 — Кросс-энтропия

На оси y расположены возможные значения ошибки для текущего объекта, на оси x - предсказанная вероятность его принадлежности определенному классу, как можно заметить при малых значениях вероятности функция потерь логарифмически возрастает для нашего объекта, соответ-

ственно модель не поощряет неточные предсказания. Кроме того такая функция потерь сразу включает в себя возможность вывода вероятности принадлежности объекта к определенному классу, делая нашу сеть менее абстрактной.

1.2 Подход ResNet

ResNet является сокращением от названия Residual Network или же остаточной сети, с появлением все более глубоких сетей появилась проблема уменьшения качества предсказаний после определенного уровня глубины сети (по-другому количества слоев в сети) из-за затухания градиента, возникающем из-за того что множество функций активаций в больших сетях со временем уменьшают большое пространство входов и как следствие большие изменения входных параметров практически не влияет на выход, что ведет к значительному уменьшению параметров в градиенте и как следствие невозможности дальнейшего обучения. В связи с этим компанией Microsoft была предложена концепция остаточных слоев с применением соединений быстрого доступа, которые пропускают один или несколько слоев. Если описывать принцип работы более подробно, то остаточный блок слоев изменяет цель обучения набора слоев с идеальных весов и смещений $H(x)$, на обучение выхода остаточного слоя $F(x) = H(x) - x$, иными словами теперь мы пытаемся аппроксимировать не всю функцию $H(x)$, а только её остаток. Пример остаточного блока можно увидеть на рисунке 1.3:

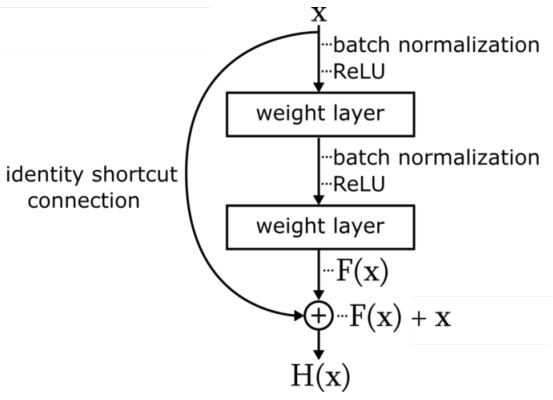

Рисунок 1.3 — Пример остаточного блока

Таким образом, затухание градиента начало происходить позже, что позволило продолжить наращивание глубины сетей и получать лучшие результаты в точности работы сверточных сетей.

1.3 Подход ODENet

В 2018 году был предложен новый подход к решению задачи оптимизации слоев в глубоких нейронных сетях с использованием дифференциальных уравнений, с помощью т.н. нейронных дифференциальных уравнений. Новый подход основывается на применении теории динамических систем для описания выходов множества скрытых слоев и соответственно позволяет получать значения скрытых слоев решая дифференциальные уравнения в такой системе. Так как, нейронные дифференциальные уравнения являются наследником сетей ResNet, рассмотрим их в аналогии с ними.

1.3.1 Аналогия с ResNet

Рассмотрим устройство сети состоящей из нескольких остаточных блоков на рисунке 1.4:

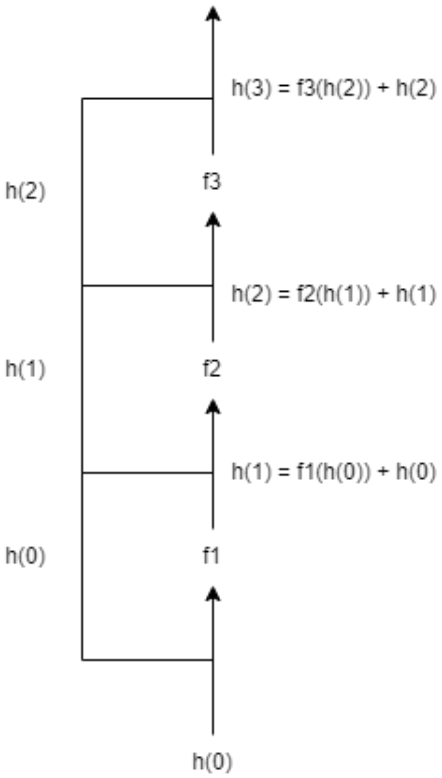

Рисунок 1.4 — Пример ResNet

В данной схеме $h(0)$ - вход нашей сети, который проходит через цепочку остаточных блоков с некоторым набором функций $f1, f2, f3$, являющиеся некоторыми функциями-слоями(т.е. выходом этих функций является вход, которые проходит через множество слоев скрытых в ней), соответственно после прохождения первого блока получаем его выход $h(1) = f1(h(0)) + h(0)$ и так далее со всеми остальными блоками. Теперь рассмотрим устройство сети состоящей из нескольких блоков новой структуры на рисунке 1.5:

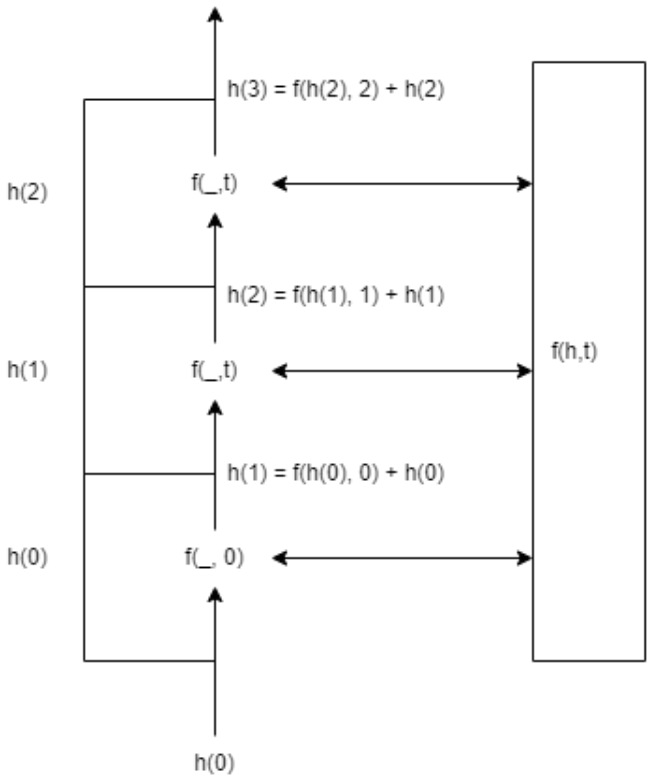

Рисунок 1.5 — Пример NeuralODE

Нетрудно заметить, очевидное сходство с предыдущей схемой, но теперь множество функций-слоев $f1, f2, f3$ описывается функцией-слоем $f(h,t)$, с дополнительным параметром t, который отвечает за номер вызова функции слоя $f(h,t)$ и понадобится нам для дальнейшего описания. Как можно заметить, выходы блоков рассчитываются аналогично, но с другим видом функции f. Прохождение множества блоков ResNet можно описать функцией вида $F(f_1, f_2, f_3, h, \theta)$, где f_1, f_2, f_3 - функции-слои, h - вход се-

ти, θ - параметры слоев, с другой стороны прохождение ODENet можно представить как $F(f, h, \theta)$, где f - общая функция-слой, зависящая дополнительно t и хранящая себе в себе все три условные функции-слои, как в предыдущем случае при различном t, h - входы сети в различные блоки ODENet, θ - параметры f. При этом, так как мы не знаем начальные значения $h(1), h(2)$ и т.д., и знаем только начальное значение входа $h(0)$, можно преобразовать функцию ODENet в следующий вид:

$$F(f, h(0), t_{max}, \theta) \tag{1.2}$$

где f - множество функций-слоев,

$h(0)$ - изначальный вход сети,

t_{max} - количество блоков в сети(или её глубина),

θ - параметры функций-слоев.

1.3.2 Теоретические достоинства ODENet

1) Из-за того что, нам не нужно хранить промежуточные частные производные при обратном проходе в сети растет её эффективность по памяти, так как часть частных производных мы вычисляем на ходу через те же дифференциальные уравнения;
2) Нейронные дифференциальные уравнения позволяют настраивать баланс между точностью вычислений и времени работы сети, благодаря возможности настройки точности решателей дифференциальных уравнений.

Теперь рассмотрим устройство сетей на основе дифференциальных уравнений более подробно.

1.3.3 Прямой проход

Для удобства, будем рассматривать h, как множество выходов блоков-слоев ODENet, в некоторых случаях будем называть их состояниями системы. Для того чтобы перейти из $h(0)$ в $h(1)$ и так далее, рассмотрим

динамику множества скрытых слоев следующего вида:

$$\frac{dh(t)}{dt} = f(h(t),t,\theta_t) \tag{1.3}$$

где $f(h(t),t,\theta)$ - скрытый слой с параметрами t и θ, причем θ – распределенные параметры скрытого слоя, а именно веса и смещения.

Для получения значения выхода скрытого слоя $h(t)$, необходимо вычислить следующий интеграл:

$$h(t_1) = \int_{t_0}^{t_1} f(h(t), t, \theta)dt + h(t_0) \tag{1.4}$$

Приняв за $h(t_0) = x$, за начальное условие, причем x - вход скрытого слоя в начальный момент времени t_0 и $h(t_1) = y$, где y - выход последнего слоя или же значение интеграла от t_0 до t_1 плюс предыдущее значение $h(t_0)$, таким образом мы смогли описать динамику изменения выходов скрытых слоев, сведя прямой проход к нахождению множества значений $h(t)$.

Нетрудно заметить, что динамика множества скрытых слоев сводится к нахождению решения уравнений вида:

$$\frac{dh(t)}{dt} = f(h(t),t) \tag{1.5}$$

Соответственно, имея начальные условия, представленные выше, мы можем рассчитать его решение, с помощью любого численного метода для решения обыкновенных дифференциальных уравнений, в данном случае рассмотрим метод Эйлера:

Пусть дана задача Коши следующего вида $\frac{dh(t)}{dt} = f(h(t),t)$ при $h(t_0) = h_0$, при этом f определена на некоторой области действительных чисел $\mathbb{R}^2$.

Само решение будем искать на интервале $[t_0, t_n]$, введем на нем узлы $t_0 < t_1 < ... < t_n$, обозначим приближенное решение в узлах t_i за $h(i)$, тогда

решение в i-ом узле определяется по следующей формуле:

$$h(i) = h(i-1) + (t_i - t_{i-1}) f(h(i-1), t(i-1)), i = 1,2,3,...,n \tag{1.6}$$

1.3.4 Обратный проход с динамикой скрытых слоев

Для вычисления градиента на обратном проходе в сети, также был предложено использовать метод сопряженных уравнений, рассмотрим принцип его действия на процессе обучения модели.

Обсудим процесс минимизации функции потерь:

$$L(h(t_1)) = L(Solver(h(t_0), t_0, t_1, \theta, f)) \tag{1.7}$$

Чтобы минимизировать L, нужно рассчитать частные производные по всем его параметрами: $h(t_0), t_0, t_1, \theta$. Для этого необходимо понять, как L зависит от состояния $h(t)$ в каждый момент времени, для этого вводится сопряженное состояние $a(t)$:

$$a(t) = -\frac{\partial L}{\partial h(t)} \tag{1.8}$$

Его динамика аналог дифференцирования сложной функции (см. B.1 в [1]):

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f(h(t), t, \theta)}{\partial h} \tag{1.9}$$

Из аналога дифференцирования сложной функции через решение в обратную сторону во времени получим зависимость от начального состояния $h(t_0)$:

$$\frac{\partial L}{\partial h(t_0)} = \int_{t_1}^{t_0} a(t)\frac{\partial f(h(t), t, \theta)}{\partial h} dt \tag{1.10}$$

Для подсчета частных производных от t и θ, нужно считать их частью состояния, которое мы будем называть аугментированным(f_{aug}), его динамика задаётся следующим образом:

$$f_{aug} = (\frac{dh}{dt} = f(h, \theta, t), \frac{d\theta}{dt} = 0, \frac{dt}{dt} = 1) \tag{1.11}$$

Найдем сопряженное состояние к аугментированному:

$$a_{aug} = (a_h(t) = \frac{\partial L}{\partial h}, a_\theta(t) = \frac{\partial L}{\partial \theta}, a_t(t) = \frac{\partial L}{\partial t}) \tag{1.12}$$

Градиент аугментированной динамики:

$$\frac{\partial f_{aug_h}}{\partial h} = \frac{\partial f}{\partial h} \tag{1.13}$$

$$\frac{\partial f_{aug_\theta}}{\partial \theta} = \frac{\partial f}{\partial \theta} \tag{1.14}$$

$$\frac{\partial f_{aug_t}}{\partial t} = \frac{\partial f}{\partial t} \tag{1.15}$$

Получим три новых дифференциальных уравнения для сопряженного аугментированного состояния:

$$\frac{da_{aug_h}}{dt} = -a\frac{\partial f}{\partial h} \tag{1.16}$$

$$\frac{da_{aug_\theta}}{dt} = -a\frac{\partial f}{\partial \theta} \tag{1.17}$$

$$\frac{da_{aug_t}}{dt} = -a\frac{\partial f}{\partial t} \tag{1.18}$$

Получаем необходимые частные производные из решения этих ОДУ:

$$\frac{\partial L}{\partial h(t_0)} = \int_{t_1}^{t_0} a(t)\frac{\partial f(h(t), t, \theta)}{\partial h}dt \tag{1.19}$$

$$\frac{\partial L}{\partial \theta} = \int_{t_1}^{t_0} a(t)\frac{\partial f(h(t), t, \theta)}{\partial \theta}dt \tag{1.20}$$

$$\frac{\partial L}{\partial t_0} = \int_{t_1}^{t_0} a(t)\frac{\partial f(h(t), t, \theta)}{\partial t}dt \tag{1.21}$$

$$\frac{\partial L}{\partial t_1} = -a(t)\frac{\partial f(h(t), t, \theta)}{\partial t} \tag{1.22}$$

Таким образом были получены частные производные по всем параметрам функции потерь, что теперь позволяет нам исходя из них обновлять веса сети для её улучшения, как в классической сверточной сети, но с помощью вычисления градиентов "на ходу" без сохранения промежуточных вычислений.

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Программное обеспечение

2.1.1 Python

В качестве языка для разработки был выбран Python, из-за простоты в использовании и наличии большого количества библиотек для разработки нейронных сетей различных конфигураций. Для разработки программы был использована версия языка Python 3.6.

2.1.2 Используемые фреймворки и библиотеки

Фреймворк в машинном обучении позволяет с помощью различных уже готовых инструментов и программных компонентов разрабатывать свои собственные решения и реализации различных алгоритмов и нейросетевых моделей. В качестве фреймворка в данном проекте был использован PyTorch (версия 1.10), позволяющий реализовывать свои нейронные сети, обрабатывать данные и тестировать свои модели.

В качестве дополнительных библиотек был использован matplotlib для построения графиков, tqdm для отображения промежуточных результатов обучения и библиотека time для замеров времени работы различных конфигураций сетей, а также математические библиотеки math и numpy для различных вычислений.

2.1.3 Google Colab

Так как обучение нейронных сетей является ресурсоемкой задачей для компьютера, было использовано облачное решение от компании Google под названием Google Colab, данный сервис предоставляет среду разработки с применением языка программирования Python и возможностью подключать все необходимые библиотеки для машинного обучения, а также использовать серверные мощности для обучения и тестирования своих моделей в режиме онлайн. К сожалению время работы на сервере ограничено 6-ю часами в день на видеокарте Nvidia Tesla T4, что в свою очередь огра-

ничило возможные размеры и сложность применяемых моделей, но этого вполне достаточно для проведения базовых экспериментов и исследования работоспособности подходов.

2.2 Описание входных данных

2.2.1 Описание датасета

Под тренировочной выборке при обучении сверточной нейронной сети подразумевается данные используемые в процессе обучения сети с применением метода обратного распространения ошибки.

В качестве тестовой выборки используется меньшая часть датасета, не обладающая общими изображениями с тренировочной выборкой.

На данный момент в свободном доступе существует множество датасетов состоящих из изображений различного качества, но мною был выбран датасет CIFAR-10.

Он состоит из 60000 изображений по 6000 изображений на класс, 50000 из которых отводится на тренировочную выборку и 10000 на тестовую выборку.

Датасет представлен исключительно цветными картинками размером 32 на 32 пикселя и состоит из 10 классов: самолеты, автомобили, птицы, коты, олени, собаки, лягушки, лошади, корабли, грузовики. Классы не пересекаются. Примеры изображений можно увидеть на рисунке 2.1:

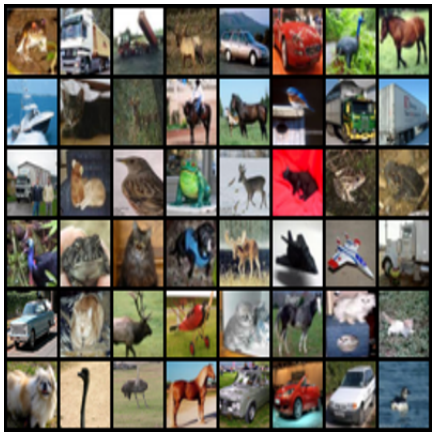

Рисунок 2.1 — Пример изображений

2.2.2 Преобразования над датасетом

Для расширения тренировочной выборки и увеличения возможностей модели к генерализации(обобщении данных) применяются различные аугментации над исходными изображениями.

Есть множество различных техник для аугментации данных но конкретно с нашим датасетом мы воспользуемся тремя в силу достаточной размерности наших тренировочных данных и небольшого качества изображений.

Так как нам нужно расширить только тренировочную выборку единственной общей аугментацией для тренировочной и тестовой выборки будет нормализация изображений.

Нормализация изображения - это приведение диапазона значений его каналов к единому промежутку для упрощения работы с ними для сети.

В PyTorch нормализация изображения происходит по следующей формуле:

$$output[i] = (input[i]) - mean[i])/std[i], i = 0,1,2,...,n \tag{2.1}$$

где $output[i]$ - значение i-ого выходного канала изображения,

$input[i]$ - значение i-ого входного канала изображения,

$mean[i]$ - среднее значение i-ого канала изображения,

$std[i]$ - стандартное отклонение i-ого канала изображения.

Для нашей сети за значение среднего и стандартного отклонения для каждого из трех каналов было выбрано число 0.5, как теоретически упрощающее для модели обработку изображений. Для тренировочной выборки были применены кроме нормализации две другие аугментации: переворот изображения по горизонтали и случайное кадрирование. Примеры аугментированных изображений из тренировочной и тестовой выборки можно увидеть на рисунках 2.2 и 2.3 соответственно:

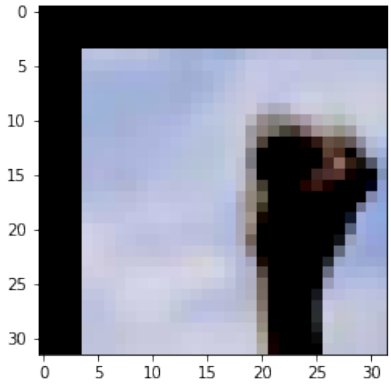

Рисунок 2.2 — Пример изображения тренировочной выборки

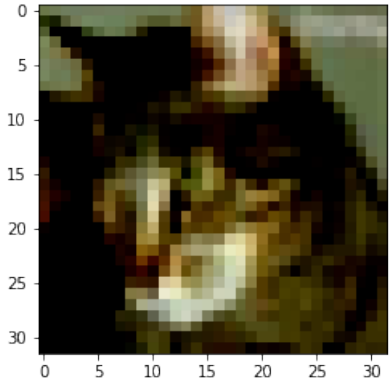

Рисунок 2.3 — Пример изображения тестовой выборки

Использованные параметры кадрирования: при кадрировании помимо стандартных параметров и размера изображения 32 на 32, к изображениям дополнительно добавляются по 4 пикселя пустого пространства сбоку изображения и сверху.

Использованные параметры переворота: единственный параметр отвечающий за вероятность переворота изображения p равен 0.5, т.е. картинки переворачиваются с вероятностью 50%.

2.3 Архитектура сетей

2.4 Реализация

Из-за того что фреймворк PyTorch предоставляет библиотеку torch, не возникло необходимости реализовывать метрики качества и функции потерь, так как они уже были реализованы в библиотеке.

Рассмотрим фрагменты кода отвечающие за тренировку и тестирование на рисунках 2.4 и 2.5:

```
def train(epoch, model, opt):
    losses = []
    loss_func = nn.CrossEntropyLoss()
    for id, (images, labels) in tqdm(enumerate(loader_tr),
        total=len(loader_tr)):
        if torch.cuda.is_available():
            images = images.cuda()
            labels = labels.cuda()
        opt.zero_grad()
        outs = model(images)
        cur_losses = loss_func(outs, labels)
        cur_losses.backward()
        opt.step()
        losses += [cur_losses.item()]
    print('Train current loss: {:.2f}'.format(np.mean(losses)))
    return np.mean(losses)
```

Рисунок 2.4 — Тренировка

```
def test(model):
    acc = 0.0
    counter = 0.0
    model.eval()
    loss_func = nn.CrossEntropyLoss()
    with torch.no_grad():
        for id, (images, labels) in tqdm(enumerate(loader_te),
                total=len(loader_te)):
            if torch.cuda.is_available():
                images = images.cuda()
                labels = labels.cuda()
            outs = model(images)
            tmp = torch.sum(torch.max(outs, dim=1)[1] == labels).item()
            acc = acc + tmp
            counter = counter + images.shape[0]
    acc = (acc / counter) * 100.
    print('Test current accuracy: {:.2f}'.format(acc))
    return acc
```

Рисунок 2.5 — Тестирование

Как можно заметить в тестовую функцию, подается только модель, являющаяся подклассом *torch.nn.Module*, что делает наши модели универ-

сальными при использовании совместно с компонентами библиотеки *torch*. Рассмотрим важные компоненты данных функций подробнее.

2.4.1 Функция потерь

В качестве функции потерь, как уже упоминалось в теоретической части была использована кросс-энтропия, её мы используем для оценки того насколько качественно учится наша модель на тренировочной выборке, в реализации её использование описывается в трех местах на рисунке 2.6:

```
loss_func = nn.CrossEntropyLoss()
cur_losses = loss_func(outs, labels)
cur_losses.backward()
```

Рисунок 2.6 — Кросс-энтропия

В первой строке инициализируем функцию потерь, во второй считаем её на единичном батче, в третьей обратным проходом оптимизируем параметры сети.

2.4.2 Метрика качества

Для задач классификации существует множество метрик качества, но в данном случае нам подойдет одна из самых распространенных, а именно точность(accuracy), её значение можно вычислить по следующей формуле:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

где TP - истинно верные предсказания модели для всех классов,

TN - истинно ложные предсказания модели для всех классов,

FP - ложно верные предсказания модели для всех классов,

FN - ложно неверные предсказания модели для всех классов.

В нашей реализации, данная метрика используется для оценки качества на тестовой выборке и вычисляется с помощью расчета её значения для каждого отдельного батча, и вычисления позднее среднего значения метрики для всей эпохи. Листинг метрики изображен на рисунке 2.7:

```
tmp = torch.sum(torch.max(outs, dim=1)[1] == labels).item()
acc = acc + tmp
counter = counter + images.shape[0]
acc = (acc / counter) * 100.
```

Рисунок 2.7 — Метрика качества

В первой строке считаем значение за батч, через одну среднее за все батчи. Размер батча может варьироваться, но в данной работе рассматривается только его вариации размером в 200 элементов, чтобы утилизировать доступную память на GPU.

2.4.3 Оптимизатор

Одним из важных параметров любой модели является подбор оптимизатора, позволяющего эффективно менять параметры сети для достижения лучшего результата, PyTorch, существует много различных оптимизаторов, но наиболее универсальным ,как показывает статья [16], является оптимизатор Adam, так, даже с дефолтной конфигурацией оптимизаторов в torch, были получены хорошие результаты. При обучении использовались две настройки скорости обучения 0.01 и стандартная 0.001 соответственно.

2.4.4 Функции динамики и общее устройство сетей

Если рассматривать конкретную реализацию, то стоит отметить что все классы сетей для использования базовых слоев должны наследоваться от nn.Module. В качестве инициализации в такой класс передается множество используемых слоев динамики (остаточный слой и/или слой динамики), количество каналов на выходе первого слоя и число повторений в остаточных блоках в случае остаточной сети.

Устройство слоя динамики и остаточного блока определяются через отдельный класс в виде маленьких подсетей. Рассмотрим каждый из них в отдельности.

В случае остаточной сети на вход классу подается количество каналов на вход и выход, при инициализации определяется единственный тип

свертки определяемый отдельной функцией следующего вида, как на рисунке 2.8:

```
def conv_nxn(in_sizes, out_sizes):
    return nn.Conv2d(in_sizes, out_sizes, 3, 1, 1)
```

Рисунок 2.8 — Базовая свертка

Кроме того определяется функция нормализации и функция активации, в конечном итоге, с методом прямого прохода класс остаточного блока приобретает следующий вид, как на рисунке 2.9:

```
class res_block(nn.Module):
    def __init__(self, out_sizes):
        super().__init__()
        self.conv_out = conv_nxn(out_sizes, out_sizes)
        self.norm = norm(out_sizes)
        self.relu = nn.ReLU(inplace = True)

    def forward(self, h):
        enter = h
        h = self.relu(self.norm(self.conv_out(h)))
        h = self.relu(self.norm(self.conv_out(h)))
        h = self.relu(h + enter)
        return h
```

Рисунок 2.9 — Класс остаточного блока

При этом функция нормализации norm дополнительно определяется через сокращение, как на рисунке 2.10:

```
def norm(out_sizes):
    return nn.BatchNorm2d(out_sizes)
```

Рисунок 2.10 — norm

Как можно заметить на выход подается сумма входов сети и выхода прошедшего через свертки, как и должно быть в базовых остаточных блоках в подобных сетях.

Аналогично определяется и слой динамики для сети на основе нейронных дифференциальных уравнений, как на рисунке 2.11:

```
class DynamicsBlock(MYF):
    def __init__(self, out_sizes):
        super().__init__()
        self.conv_out = conv_nxn(out_sizes + 1, out_sizes)
        self.norm = norm(out_sizes)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, h, times):
        h = cat_time(h, times)
        h = self.relu(self.norm(self.conv_out(h)))
        h = cat_time(h, times)
        h = self.relu(self.norm(self.conv_out(h)))
        return h
```

Рисунок 2.11 — Динамика

Но сам класс изначально наследуется от класс подсчета аугментированной динамики устройство которого мы рассмотрим позднее. Кроме того, на вход такому слою подаются сигналы сети в виде состояния и момент времени t. Дополнительно определная функции конкатенации cat_time, для возможности дальнейших вычислений данным подходом, как на рисунке 2.12:

```
def cat_time(ts, times):
    bias, tmp, we, hid = ts.shape
    return concat((ts, times.expand(bias, 1, we,
    hid)), dim=1)
```

Рисунок 2.12 — Конкатенация времени

2.4.5 ResNet

В качестве базовой модели будем использовать схему, похожую на модель представленную в статье [2].

С помощью PyTorch была реализована упрощенная схема такой сети, для удобства и скорости работы с датасетом CIFAR-10. Рассмотрим используемую схему на рисунке 2.13:

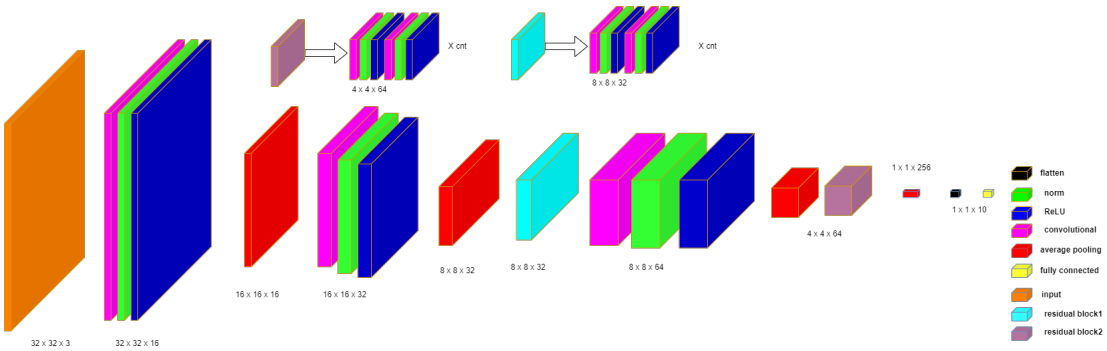

Рисунок 2.13 — Схема ResNet

Рассмотрим устройство класса реализующего данную схему на рисунке 2.14:

```
class ResidualNetwork(nn.Module):
    def __init__(self, res_block, base_channels, repeat_cnt):
        super().__init__()
        self.to_16 = conv_nxn(3, 16)
        self.to_32 = conv_nxn(16, 32)
        self.to_64 = conv_nxn(32, 64)
        self.relu = nn.ReLU(inplace=True)
        self.norm1 = norm(base_channels)
        self.norm2 = norm(base_channels * 2)
        self.norm3 = norm(base_channels * 4)
        self.avg_pool2 = nn.AvgPool2d(2)
        self.res_block1 = self.create_block(res_block, 32, repeat_cnt)
        self.res_block2 = self.create_block(res_block, 64, repeat_cnt)
        self.linear = nn.Linear(base_channels * 16, 10)
    def create_block(self, block, out_sizes, cnt):
      layers = []
      for i in range (0, cnt):
        layers.append(block(out_sizes, out_sizes))
      return nn.Sequential(*layers)
    def forward(self, inout):
        inout = self.to_16(inout)
        inout = self.norm1(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.to_32(inout)
        inout = self.norm2(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.res_block1(inout)
        inout = self.to_64(inout)
        inout = self.norm3(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.res_block2(inout)
        inout = self.avg_pool2(inout)
        inout = inout.view(inout.size(0), -1)
        out = self.linear(inout)
        return out
```

Рисунок 2.14 — Класс ResNet

На вход классу при инициализации подается устройство остаточного блока, количество входных каналов для промежуточных вычислений и число повторений остаточных блоков. В качестве двух методов реализованы прямой проход и create_block, отвечающий за объединение остаточных блоков для их последовательного применения к данным с помощью nn.Sequential из библиотеки torch. Данная реализация сети состоит из нескольких блоков вида свертка->нормализация->задание нелинейности с помощью ReLu. Такие блоки увеличивают количество каналов изображения, чтобы не потерять данные при дальнейшем его уменьшении с помощью усреднения по пикселям (avg_pool2). Между блоками сверток используются их остаточные виды res_block1 и res_block2, для "деталей" изображения разных размеров. После прохождения всех этих слоев данные линеризуются и подаются в линейный слой (nn.Linear), где и производится классификация.

2.4.6 ODENet

Схема сети на основе дифференциальных уравнений практически аналогична предыдущей, за тем исключением, что вместо остаточных блоков используются блоки на основе дифференциальных уравнений ode1 и ode2. Рассмотрим её на рисунке 2.15:

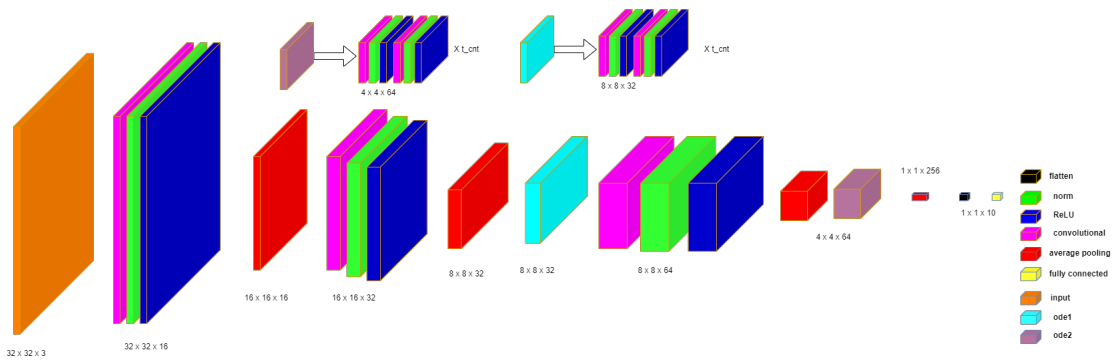

Рисунок 2.15 — Схема ODENet

Рассмотрим само устройство сети реализованной с помощью модулей PyTorch на рисунке 2.16:

```
class ODEClassifier(nn.Module):
  def __init__(self, ode, base_channels):
        super().__init__()
        self.to_16 = conv_nxn(3, 16)
        self.to_32 = conv_nxn(16, 32)
        self.to_64 = conv_nxn(32, 64)
        self.relu = nn.ReLU(inplace=True)
        self.norm1 = norm(base_channels)
        self.norm2 = norm(base_channels * 2)
        self.norm3 = norm(base_channels * 4)
        self.avg_pool2 = nn.AvgPool2d(2)
        self.ode1 = ode[0]
        self.ode2 = ode[1]
        self.linear = nn.Linear(base_channels * 16, 10)
  def forward(self, inout):
        inout = self.to_16(inout)
        inout = self.norm1(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.to_32(inout)
        inout = self.norm2(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.ode1.fwd_bcwd(inout)
        inout = self.to_64(inout)
        inout = self.norm3(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.ode2.fwd_bcwd(inout)
        inout = self.avg_pool2(inout)
        inout = inout.view(inout.size(0), -1)
        out = self.linear(inout)
        return out
```

Рисунок 2.16 — Класс ODENet

Как можно заметить при инициализации мы падаем на вход параметр ode, который описывает слой на основе дифференциальных уравнений и определяется блоком динамики представленным выше и точностью решения дифференциальных уравнений применяемых при прямом и обратном проходе через такой блок, на рисунке 2.17:

```
ode2 = HelperClass(DynamicsBlock(64), 0.1)
```

Рисунок 2.17 — Пример определения слоя ode

Рассмотрим устройство HelperClass, определяющий внутри себя прямой и обратный проход по сети с применением нейронных дифференциальных уравнений, на рисунке 2.18

```
class HelperClass(nn.Module):
    def __init__(self, dynamics_f, ode_accuracy):
        super().__init__()
        self.dynamics_f = dynamics_f
        self.ode_accuracy = ode_accuracy
    def fwd_bcwd(self, h0, times = [0., 1.]):
        times = Tensor(times)
        times = times.to(h0)
        h = Adjoint.apply(h0, times, self.dynamics_f.flat_to_one(),
        self.dynamics_f, self.ode_accuracy)
        return h[-1]
```

Рисунок 2.18 — HelperClass

Как можно заметить в классе определен метод fwd_bcwd, получающий на вход начальное состояние в виде сигналов предыдущего слоя и множество моментов времени t, в виде тензора, сама же функции внутри себя применяет прямой и обратный проход с помощью функции apply из torch, сам же прямой и обратный проход дополнительно объявлены в классе Adjoint, рассмотрим его устройство поэтапно, код представлен на рисунке 2.19:

```
class Adjoint(torch.autograd.Function):
    @staticmethod
    def forward(ctx, h_begin, times, parameters, dynamics_f,
    ode_accuracy):
        max_time = times.size(0)
        bias, *shapes_h = h_begin.size()
        with torch.no_grad():
            h = torch.zeros(max_time, bias, *shapes_h).to(h_begin)
            h[0] = h_begin
            for cur_t in range(max_time - 1):
                h_begin = euler_solve(h_begin, times[cur_t],
                times[cur_t+1],
                dynamics_f, ode_accuracy)
                h[cur_t+1] = h_begin
        ctx.dynamics_f = dynamics_f
        ctx.ode_accuracy = ode_accuracy
        ctx.save_for_backward(times, h.clone(), parameters)
        return h
```

Рисунок 2.19 — Прямой проход

Для организации своего прямого и обратного прохода и его совместного использования с компонентами сетей на основе PyTorch и применения методов расчета градиентов torch, данный класс наследует torch.autogard.Function. В прямом проходе как упомниналось в теоретической части рассчитываются выходы сети через переходы по состояниям h.

Рассмотрим обратный проход данного класса с методом обратного распространения ошибки поэтапно, код представлен на рисунке 2.20:

```
def backward(ctx, dLdh):
    ode_accuracy = ctx.ode_accuracy
    dynamics_f = ctx.dynamics_f
    times, h, params = ctx.saved_tensors
    max_time, bias, *shapes_h = h.size()
    parameters_cnt = params.size(0)
    dimensions = np.prod(shapes_h)
    def aug_dynamics(aug_h_iter, times_iter):
        adf_d_hidden = torch.zeros(bias, *shapes_h)
        adf_d_parameters = torch.zeros(bias, parameters_cnt)
        adfd_times = torch.zeros(bias, 1)
        h_iter, grad_outs = aug_h_iter[:, :dimensions],
        aug_h_iter[:, dimensions:2*dimensions]
        h_iter = h_iter.reshape(bias, *shapes_h)
        grad_outs = grad_outs.reshape(bias, *shapes_h)
        with torch.set_grad_enabled(True):
            times_iter = times_iter.requires_grad_(True)
            h_iter = h_iter.requires_grad_(True)
            dadt = dynamics_f.compute_f_adf(h_iter, times_iter,
            g_outs = grad_outs)
            function_out, adf_d_hidden, adfd_times,
            adf_d_parameters = dadt
            adf_d_hidden = adf_d_hidden.to(h_iter)
            adf_d_parameters = adf_d_parameters.to(h_iter)
            adfd_times = adfd_times.to(h_iter)
        function_out = function_out.reshape(bias, dimensions)
        adf_d_hidden = adf_d_hidden.reshape(bias, dimensions)
        return concat((function_out, -adf_d_hidden,
        -adf_d_parameters, -adfd_times),dim=1)
```

Рисунок 2.20 — 1 этап

На вход обратному проходу подается градиент от функции потерь, на начальном этапе в локальные переменные сохраняются параметры и состояния из прямого прохода и определяется функция аугментированный динамики (aug_dynamics).

В дополнительном классе объявляется функция для подсчета аугментированной динамики (compute_f_adf), а также функция конкатенации и выравнивания параметров функции динамики (flat_to_one), код представлен на рисунке 2.21:

```
class MYF(nn.Module):
  def compute_f_adf(self, h, times, g_outs):
        req_o = self.forward(h, times)
        b_s = h.shape[0]
        tmp = g_outs
        unchangable = tuple(self.parameters())
        adf_d_hidden, adfd_times, *adf_d_parameters =
        torch.autograd.grad((req_o,), (h, times)
        + unchangable, grad_outputs=(tmp))
        adf_d_parameters = concat([parameter_gradients.flatten()
        for parameter_gradients in adf_d_parameters])
        adf_d_parameters = adf_d_parameters[None, :]
        adf_d_parameters = adf_d_parameters.expand(b_s, -1) / b_s
        adfd_times = adfd_times.expand(b_s, 1) / b_s
        return req_o, adf_d_hidden, adfd_times, adf_d_parameters
  def flat_to_one(self):
        shapes = []
        flatten = []
        for i in self.parameters():
            shapes.append(i.size())
            flatten.append(i.flatten())
        tmp = concat(flatten)
        return tmp
```

Рисунок 2.21 — Класс MYF

На 2-ом этапе инициализируем нулями переменные adjoint_h, adjoint_p, adjoint_times, в которых в дальнейшем будут хранится градиен-

ты по состояниям, параметрам и времени соответственно. Далее в цикле назад во времени вычисляем прямые градиенты от состояний dLdh_iter(уже автоматически посчитано), dLdtimes_iter и поправим ими сопряженные состояния adjoint_h и adjoint_times[cur_t], код представлен на рисунке 2.22:

```
        adjoint_h = torch.zeros(bias, dimensions).to(dLdh)
        adjoint_p = torch.zeros(bias, parameters_cnt).to(dLdh)
        adjoint_times = torch.zeros(max_time, bias, 1).to(dLdh)
        for cur_t in range(max_time - 1, 0, -1):
            h_iter = h[cur_t]
            times_iter = times[cur_t]
            f_iter = dynamics_f(h_iter,
            times_iter).reshape(bias, dimensions)
            dLdh_iter = dLdh[cur_t]
            dLdtimes_iter = matrix_product(transp(dLdh_iter[:,
            :, None],1, 2), f_iter[:, :, None])[:, 0]
            adjoint_h += dLdh_iter
            adjoint_times[cur_t] = adjoint_times[cur_t]
            - dLdtimes_iter
            augmented_h = concat((h_iter.reshape(bias,
            dimensions), adjoint_h,
            torch.zeros(bias, parameters_cnt).to(h),
            adjoint_times[cur_t]), dim=-1)
            augmented_solution = euler_solve(augmented_h, times_iter
            , times[cur_t-1], aug_dynamics, ode_accuracy)
            adjoint_h[:] = augmented_solution[:,
            dimensions:2*dimensions]
            adjoint_p[:] += augmented_solution[:,
            2*dimensions:2*dimensions + parameters_cnt]
            adjoint_times[cur_t-1] = augmented_solution[:,
            2*dimensions + parameters_cnt:]
            del augmented_h, augmented_solution
        dLdh_0 = dLdh[0]
        dLdtimes_0 = matrix_product(transp(dLdh_0[:, :, None],
        1, 2),f_iter[:, :, None])[:, 0]
        adjoint_h += dLdh_0
        adjoint_times[0] = adjoint_times[0] - dLdtimes_0
    return adjoint_h.reshape(bias, *shapes_h), adjoint_times,
    adjoint_p, None, None
```

Рисунок 2.22 — 2 этап

Далее сложим аугментированные переменные в один тензор, код представлен на рисунке 2.23:

```
augmented_h = concat((h_iter.reshape(bias, dimensions), adjoint_h,
torch.zeros(bias, parameters_cnt).to(h),
adjoint_times[cur_t]), dim=-1)
```

Рисунок 2.23 — Аугментированные переменные в тензоре

Далее находим решение аугментированной системы назад во времени с помощью метода Эйлера, как показано на рисунке 2.24:

```
augmented_solution = euler_solve(augmented_h, times_iter,
times[cur_t-1], aug_dynamics, ode_accuracy)
```

Рисунок 2.24 — Решение системы назад во времени

При этом сам метод Эйлера в нашей реализации выглядит следующим образом на рисунке 2.25:

```
def euler_solve(h, times, times_last, f, acc_param = 0.05):
    steps = ceiling(abs(times_last - times) / acc_param)
    time_step = (times_last - times)/steps
    for i in range(steps):
        h = h + time_step * f(h, times)
        times = times + time_step
    return h
```

Рисунок 2.25 — Метод Эйлера

Запишем полученные градиенты, код представлен на рисунке 2.26:

```
adjoint_h[:] = augmented_solution[:,
dimensions:2*dimensions]
adjoint_p[:] += augmented_solution[:,
2*dimensions:2*dimensions + parameters_cnt]
adjoint_times[cur_t-1] = augmented_solution[:,
2*dimensions + parameters_cnt:]
```

Рисунок 2.26 — Полученные градиенты

Подправим сопряженное состояние в начальный момент времени прямыми градиентами от функции потерь, код представлен на рисунке 2.27:

```
dLdh_0 = dLdh[0]
dLdtimes_0 = matrix_product(transp(dLdh_0[:, :, None],
1, 2), f_iter[:, :, None])[:, 0]
adjoint_h += dLdh_0
adjoint_times[0] = adjoint_times[0] - dLdtimes_0
```

Рисунок 2.27 — Правим сопряженное состояние

Возвращаем полученные градиенты функции ошибки по h в начальный момент времени, моменты времени t и параметров слоя ode, код представлен на рисунке 2.28:

```
return adjoint_h.reshape(bias, *shapes_h), adjoint_times,
adjoint_p, None, None
```

Рисунок 2.28 — Итоговые частные производные

2.4.7 ResidualODE

Гибридная схема с остаточными слоями и ode слоями является попыткой достичь баланса между эффективностью по памяти и по времени. Рассмотрим её схему на рисунке 2.29:

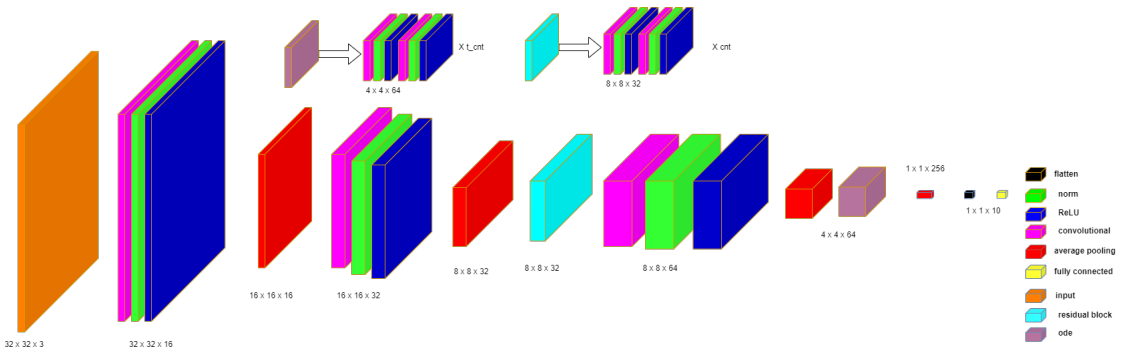

Рисунок 2.29 — Схема гибридного подхода

Класс такой схемы выглядит аналогично предыдущим, но с применением, как методов остаточных сетей, так и блока динамики из нейронных дифференциальных уравнений, как на рисунке 2.30:

```
class ResidualODE(nn.Module):
  def __init__(self, res_block, ode, base_channels, repeat_cnt):
        super().__init__()
        self.to_16 = conv_nxn(3, 16)
        self.to_32 = conv_nxn(16, 32)
        self.to_64 = conv_nxn(32, 64)
        self.relu = nn.ReLU(inplace=True)
        self.norm1 = norm(base_channels)
        self.norm2 = norm(base_channels * 2)
        self.norm3 = norm(base_channels * 4)
        self.avg_pool2 = nn.AvgPool2d(2)
        self.res_block1 = self.create_block(res_block, 32, repeat_cnt)
        self.ode = ode
        self.linear = nn.Linear(base_channels * 16, 10)
  def create_block(self, block, out_sizes, cnt):
      layers = []
      for i in range (0, cnt):
        layers.append(block(out_sizes, out_sizes))
      return nn.Sequential(*layers)
  def forward(self, inout):
        inout = self.to_16(inout)
        inout = self.norm1(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.to_32(inout)
        inout = self.norm2(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.res_block1(inout)
        inout = self.to_64(inout)
        inout = self.norm3(inout)
        inout = self.relu(inout)
        inout = self.avg_pool2(inout)
        inout = self.ode.fwd_bcwd(inout)
        inout = self.avg_pool2(inout)
        inout = inout.view(inout.size(0), -1)
        out = self.linear(inout)
        return out
```

Рисунок 2.30 — Гибридная схема

Таким образом, мы рассмотрели все использованные и реализованные схемы нейронных сетей, и теперь можем перейти к непосредственной проверке подходов.

2.5 Результаты

2.5.1 Сравнение параметров сетей

Рассмотрим полученные результаты тестирования на рисунке 2.31, а также графики точности в процессе тестирования, тестирование проводилось после каждой эпохи обучения на тестовой выборке, рассматриваются только лучшие из таблицы, выделены цветом на рисунке 2.32:

Сеть	Максимальная точность	Время обучения (все эпохи)	Время теста(последняя эпоха)	Используемая память
ODENet(точность 0.5, lr = 0.01)	74.44	1899	3.109251738	296144
ODENet(точность 0.25, lr = 0.01)	77	1967	3.352573395	296144
ODENet(точность 0.1, lr = 0.01)	75.86	2536	3.572696209	296144
ResNet(1 блок , lr = 0.01)	76.94	1041	2.818629265	292688
ResNet(2 блока, lr = 0.01)	64.38	1074	3.321099997	478944
ResNet(3 блока, lr = 0.01)	64.09	1106	3.471029282	665200
ResidualODE(точность 0.25, 1 блок, lr = 0.01)	76.9	1711	2.894787073	294992
ResidualODE(точность 0.25, 2 блока, lr = 0.01)	74.98	1793	3.053413868	332504
ResidualODE(точность 0.1, 1 блок, lr = 0.01)	77.28	2143	3.349468946	294992
ResidualODE(точность 0.1, 2 блока, lr = 0.01)	68.95	2167	3.509723186	332504
ResidualODE(точность 0.5, 1 блок, lr = 0.01)	78.8	1574	2.834675789	294992
ResidualODE(точность 0.5, 2 блока, lr = 0.01)	66.66	1597	2.982625008	332504
ResidualODE(точность 0.5, 3 блока, lr = 0.01)	67.83	1607	3.721503258	370016
ODENet(точность 0.5, lr = 0.001)	79.89	1926	2.478101015	296144
ODENet(точность 0.25, lr = 0.001)	73.24	1999	2.919076443	296144
ODENet(точность 0.1, lr = 0.001)	67.78	2543	3.331475496	296144
ResNet(1 блок , lr = 0.001)	79.65	1028	2.398873568	292688
ResNet(2 блока, lr = 0.001)	79.8	1159	2.479676485	478944
ResNet(3 блока, lr = 0.001)	68.55	1162	2.686522484	665200
ResidualODE(точность 0.25, 1 блок, lr = 0.001)	77.66	1609	2.510575056	294992
ResidualODE(точность 0.25, 2 блока, lr = 0.001	67.86	1679	2.701941729	332504
ResidualODE(точность 0.1, 1 блок, lr = 0.001)	79.36	2109	2.625131845	294992
ResidualODE(точность 0.1, 2 блока, lr = 0.001)	71.64	2285	2.829180002	332504
ResidualODE(точность 0.5, 1 блок, lr = 0.001)	78.37	1588	2.503554106	294992
ResidualODE(точность 0.5, 2 блока, lr = 0.001)	70.32	1599	2.583010435	332504
ResidualODE(точность 0.5, 3 блока, lr = 0.001)	74.26	1658	2.725016832	370016

Рисунок 2.31 — Результаты обучения 50 эпох

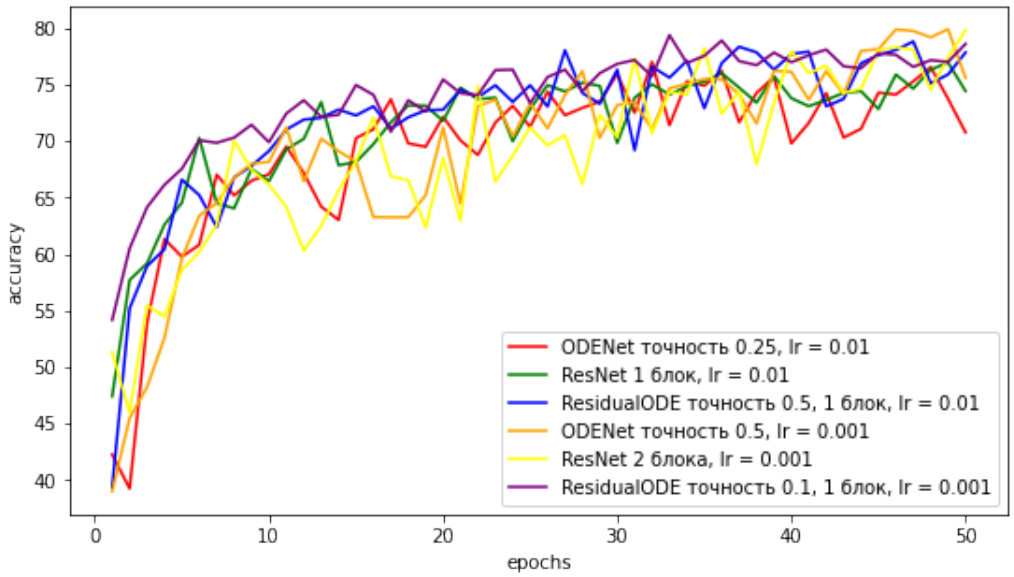

Рисунок 2.32 — Результаты тестирования график

Как можно заметить графики после за 50 эпох не сильно увеличиваются в точности после отметки в 40 эпох, вероятнее всего было достигнуто некоторое подобие плато обучения при данных настройках, при этом сети при обучении со скоростью обучения равной 0.01 справились хуже чем модели с аналогичным параметром равном 0.001 вероятнее всего это связано с "проскакиванием"локальных минимумов, хотя ошибка на тренировочной выборке всех моделей продолжает падать, как можно заметить на графике ниже по рисунку 2.33:

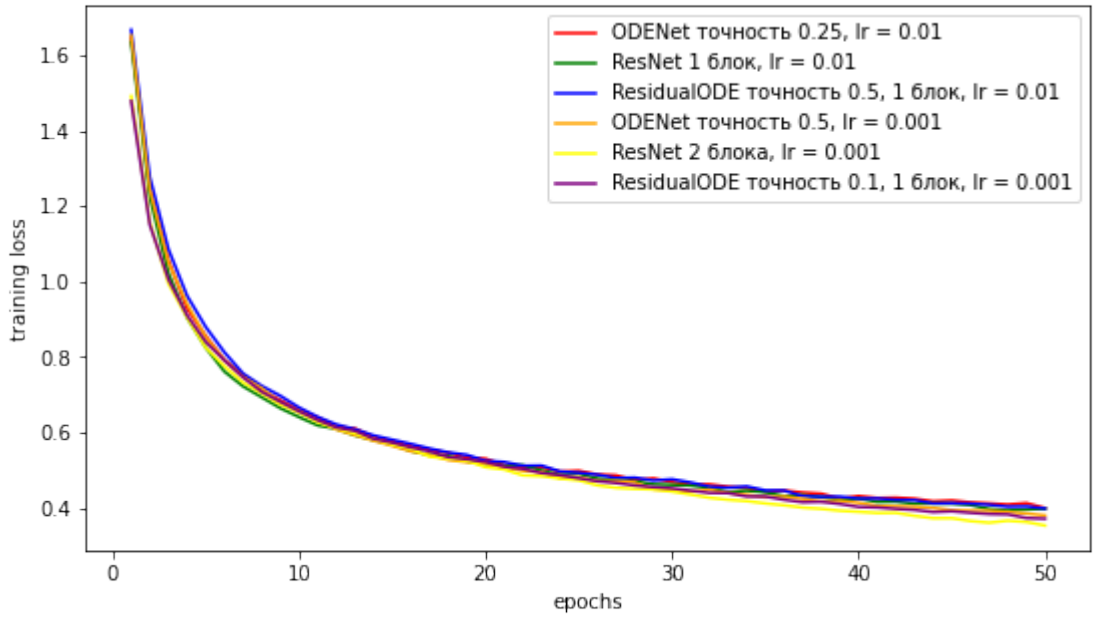

Рисунок 2.33 — Результаты тренировки график

В большинстве случаев $lr = 0.001$ больше подошел для нашей задачи, что наглядно видно по графикам, хотя гибридная схема хорошо показала себя в обоих случаях (ResidualODE). При этом, несмотря на общую схожесть результатов по точности, схемы на основе дифференциальных уравнений, как и ожидалось оказались более эффективны по памяти, примерно на тот же порядок, как они отстали от времени выполнения от остаточных сетей, гибридная схема стала некоторым компромиссом в данном подходе, что и ожидалось при описанных ранее подходах. Таким образом, сети с блоками на основе дифференциальных уравнений могут вполне подойти в задачах с ограничениями на используемую память.

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе бакалавра были получены следующие результаты:

1) Реализованы нейронные сети на основе дифференциальных уравнений и остаточных блоках;
2) С применением полученной реализации решена задача классификации изображений датасета CIFAR-10;
3) Проведен сравнительный анализ полученных реализаций.

Рассмотренные методы с применением дифференциальных уравнений действительно показывают свою эффективность по памяти, хотя и отстают по времени от сетей на основе остаточных блоков, точность рассмотренных сетей держится примерно на одном уровне, что говорит о работоспособности всех предложенных подходов. В дальнейшем следует рассмотреть различные алгоритмы для оптимизации сетей на основе дифференциальных уравнений, чтобы сократить разницу во времени и сохранить эффективность по памяти, провести большое количество тестов для прочих возможных параметров в используемых сетях.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud. (2019). Neural Ordinary Differential Equations.arXiv:1806.07366
2. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. (2015). Deep Residual Learning for Image Recognition.arXiv:1512.03385
3. PyTorch documentation. URL: https://pytorch.org/docs/stable (дата обращения 10.03.22).
4. Jonty Sinai. (2019). Understanding Neural ODE's.URL: https://clck.ru/h3c7d
5. Kashiwa. (2022). Implement ResNet with PyTorch.URL: https://towardsdev.com/implement-resnet-with-pytorch-a9fb40a77448
6. Арнольд, В. И. Обыкновенные дифференциальные уравнения / В. И. Арнольд. — Изд. 2-е. — Москва : МЦНМО, 2020. — ISBN 978-5-4439-3254-5.— С. 35—200
7. Jake Krajewski. (2020). PyTorch Layer Dimensions: Get your layers to work every time (the complete guide). URL: https://clck.ru/h3c7o
8. Loss Functions ML Gloassary docs. — URL: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html (дата обращения 10.03.22).
9. Margherita Grandini, Enrico Bagli, Giorgio Visani. (2020). Metrics for Multi-Class Classification: an Overview. arXiv:2008.05756
10. Kashiwa. (2022). Implement ResNet with PyTorch. URL: https://towardsdev.com/implement-resnet-with-pytorch-a9fb40a77448
11. Практики реализации нейронных сетей. - URL: https://usnd.to/tpru (дата обращения 10.03.22).
12. Обучение нейронных сетей. Обратное распространение ошибки. - URL: https://bit.ly/395MkR9 (дата обращения 10.03.22).
13. Гудфеллоу, Я. Глубокое обучение / Я. Гудфеллоу, Й. Бенджио, А. Курвилль. - 2017 : ДМК, 2018.

14. Многослойная нейронная сеть (Multilayer neural net). - URL: https://wiki.loginom.ru/articles/multilayer-neural-net.html (дата обращения 10.03.22).
15. Николенко, С. Глубокое обучение. Погружение в мир нейронных сетей / С. Николенко, А. Кадурин, Е. Архангельская. — 2020 : Питер СПб, 2020. — ISBN 978-5-4461-1537-2.— С. 20—100.
16. Sebastian Ruder. (2016). An overview of gradient descent optimization algorithms. URL: https://ruder.io/optimizing-gradient-descent/
17. Eldad Haber, Lars Ruthotto. (2017). Stable Architectures for Deep Neural Networks.arXiv:1705.03341
18. Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, Elliot Holtham. (2017). Reversible Architectures for Arbitrarily Deep Residual Neural Networks.arXiv:1709.03698

ПРИЛОЖЕНИЕ А

Ссылка на хранилище с кодом: `https://github.com/IllCher/diplomacode/blob/main/code.py`

По ссылке можно перейти отсканировав код на рисунке А.1:

Рисунок А.1 — QR-код