

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра систем автоматизированного проектирования

ОТЧЕТ
по курсовой работе
по дисциплине «Алгоритмы и структуры данных»
Тема: «Реализация нейронной сети для распознавания цветов»

Студент гр. 3352

Гультяев А.С.

Преподаватель

Пестерев Д.О.

Санкт-Петербург

2024

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Гультияев А.С.

Группа 3352

Тема работы: Реализация нейронной сети для распознавания цветов.

Исходные данные: Python 3.12, библиотеки matplotlib, numpy, os. На вход сети подаются картинка размером 128x128 с некоторым изображением цветка.

Содержание пояснительной записки: «Аннотация», «Введение», «Теоретическая часть», «Базовая архитектура любой сети», «Виды нейронных сетей», «Методы улучшения нейронных сетей», «Архитектура сверточной нейронной сети», «Принцип работы исходной сети», «Принцип обучения сети», «Практическая часть», «Предобработка изображения», «Подготовка и расчет сети», «Обучение сети», «Раздел со всем кодом», «Вывод», «Ссылка на репозиторий github с исходным кодом».

Предполагаемый объем пояснительной записки:

Не менее 30 страниц.

Дата выдачи задания: 29.10.2024

Дата сдачи реферата: .12.2024

Дата защиты реферата: .12.2024

Студент

Гультияев А.С.

Преподаватель

Пестерев Д.О.

Оглавление

Аннотация.....	4
Введение.....	5
1. Теоретическая часть.....	6
1.1. Базовая архитектура любой сети.....	6
1.2. Виды нейронных сетей.....	8
1.3. Методы улучшения работы сетей.....	10
1.4. Архитектура исходной сверточной нейронной сети.....	11
1.5. Принцип работы исходной нейронной сети.....	12
1.6. Принцип обучения сети.....	17
2. Практическая часть.....	22
2.1. Предобработка изображений.....	22
2.2. Подготовка и расчет сети.....	23
2.3. Обучение сети.....	24
3. Раздел со всем кодом.....	30
4. Вывод.....	46
5. Ссылка на репозиторий GitHub с исходным кодом.....	46

Аннотация

В данной работе реализована сверточная нейронная сеть (CNN), предназначенная для распознавания одного из пяти видов цветков на изображениях. Архитектура сети включает два сверточных слоя, два слоя подвыборки (max-pooling), а также два полносвязных слоя. Реализация нейронной сети была выполнена "с нуля" без использования сторонних библиотек машинного обучения, что позволило глубже изучить внутренние процессы обработки данных, такие как свертка, активация и обучение с помощью градиентного спуска. В ходе экспериментов было показано, что предложенная архитектура обладает достаточной точностью для выполнения поставленной задачи.

Summary

This work presents the implementation of a convolutional neural network (CNN) designed to classify images into one of five flower species. The network architecture consists of two convolutional layers, two max-pooling layers, and two fully connected layers. The neural network was implemented from scratch without using external machine learning libraries, which allowed for a deeper understanding of internal data processing mechanisms, such as convolution, activation, and training using gradient descent. Experimental results demonstrate that the proposed architecture achieves sufficient accuracy for the given task.

Введение

Цель: реализация сверточной нейронной сети, которая будет распознавать один из пяти видов цветков.

Задачи:

- 1) Изучить строение сетей.
- 2) Изучить различные виды нейронных сетей.
- 3) Изучить принципы подготовки изображений для подачи в сеть.
- 4) Изучить принципы обучения нейронных сетей для классификации изображений.
- 5) Рассмотреть методы улучшения сети.
- 6) Реализовать собственную нейронную сеть для предсказания классификации изображения.
- 7) Построить графики и модели эффективности обучения.

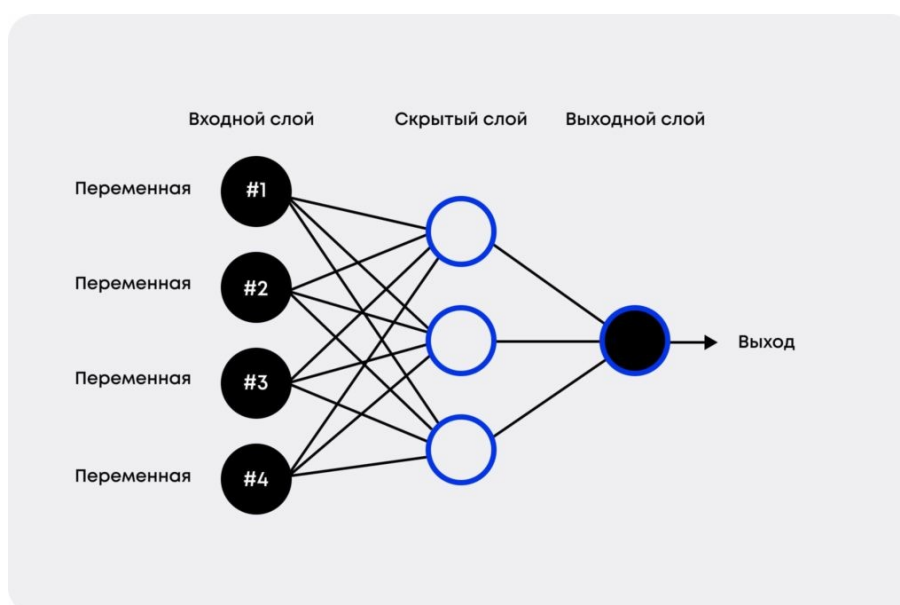
1. Теоретическая часть

1.1. Базовая архитектура любой сети.

Каждая нейронная сеть (примеры различных сетей будут рассмотрены далее), имеет на начальном этапе общий вид — входной слой, какое-то количество скрытых слоев, выходной слой. Данное устройство помогает сети обрабатывать гигантское количество информации и чисел. По своему устройству нейронная сеть чем-то напоминает устройство мозга человека, а именно — наличием нейронов. Однако в отличие от человеческих нейронов, нейроны сети обрабатываются в сотни раз быстрее, поскольку компьютер лучше работает с числами, нежели человек. Компьютеру проще обрабатывать большие массивы данных, нежели человеку. Также человек использует для связи между нейронами химические реакции, которые обрабатываются медленнее, чем компьютер.

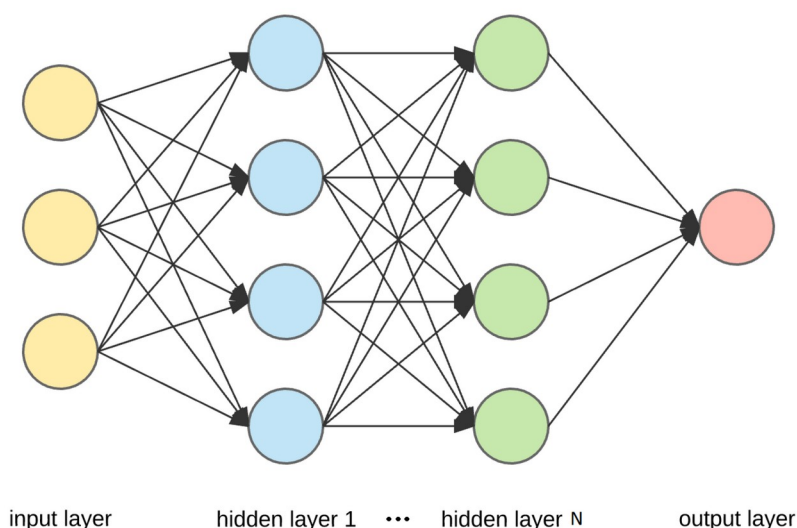
Сами по себе нейроны в сети — некоторые математические функции или абстракции, которые принимают на вход данные (числовые значения), а затем обрабатывает их с использованием весов и функций активации. Пример нейрона приведен на изображении 1. На данном изображении модель нейрона сети — это каждый отдельный кружок на центральном слое.

Изображение 1. Модель нейрона сети.



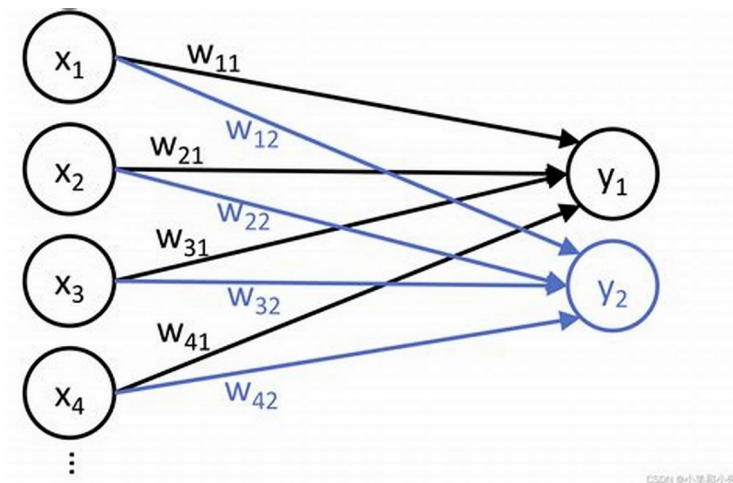
Следующей составляющей нейронной сети будет являться слой сети. Слой сети, будь то скрытый, входной или выходной — тоже условная абстракция, которая содержит в себе как раз таки те самые нейроны. Вообще слои бывают разных видов — полносвязный, не полносвязный, сверточный, dropout, и многие многие другие. Однако большинство базовых сетей имеет обыкновенный полносвязный слой. Структура полносвязного слоя работает так, что каждый нейрон предыдущего слоя соединен с нейроном текущего, а также каждый нейрон текущего слоя соединен с нейроном следующего слоя. Пример полносвязного слоя приведен на изображении 2.

Изображение 2. Пример полносвязных слоев сети.



Поскольку, как было сказано ранее, нейроны — некоторая функция, которая обрабатывается с помощью весов, стоит разобраться, что такое веса. Веса — это связи между каждым нейроном предыдущего слоя с нейроном текущего слоя. Веса имеют числовое значение и помогают работать нейронной сети так же, как и человеческие нейроны, а именно — помогают какие-то нейроны активироваться сильнее, чем другие. Иными словами, веса служат некоторым инструментом, который помогает сети понять, какие нейроны должны быть более активны в том или ином случае. Примеры весов так же приведены на изображении 3. На данном изображении веса обозначены буквой W .

Изображение 3. Пример весов нейронной сети.

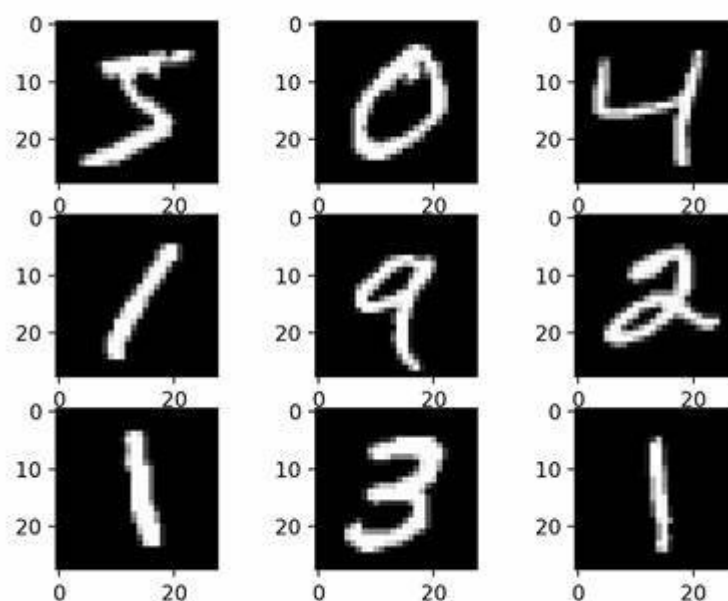


Из данного изображения видно, что каждый нейрон X соединен с каждым нейроном Y некоторыми значениями W_{ij} — весами связи.

1.2. Виды нейронных сетей.

Всего видов нейронных сетей — бесчисленное множество. Например, для распознавания базовой задачи в обучении сетей MNIST (набор рукописных цифр от 0 до 9, размерами 28 на 28 пикселей) используется обыкновенная многослойная нейронная сеть (MLP — многослойный персептрон). Пример устройства такой сети ничем не отличается от базовой. Пример цифр из набора MNIST приведен на изображении 4.

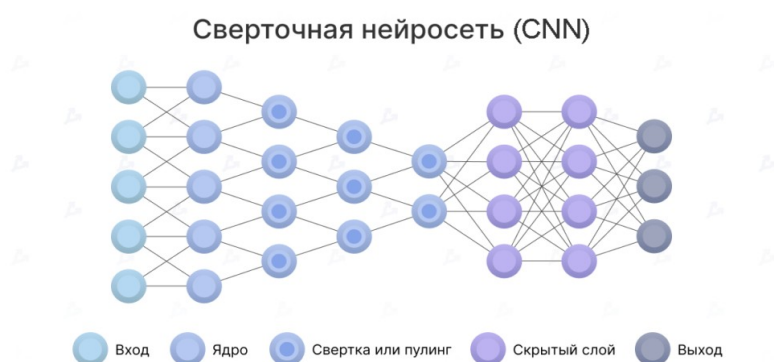
Изображение 4. Пример цифр набора MNIST



Регрессионная нейронная сеть — сеть используемая для сопоставления и анализа многих факторов, а также анализа прошлых значений и их зависимостей для того, что бы выдать какое-то итоговое значение. Например, такие сети используются для предсказания стоимости домов в зависимости от местности, города, положения в городе и стране, курсе валют и много другого. В свою очередь разница между работами данных сетей заключается в том, что регрессионная сеть выдает итоговое число, в то время как многослойная нейронная сеть выдает вероятность того или иного исхода. Структура такой сети зачастую проще, в силу того, что в ней содержится чуть меньше слоев, чем в многослойном персептроне.

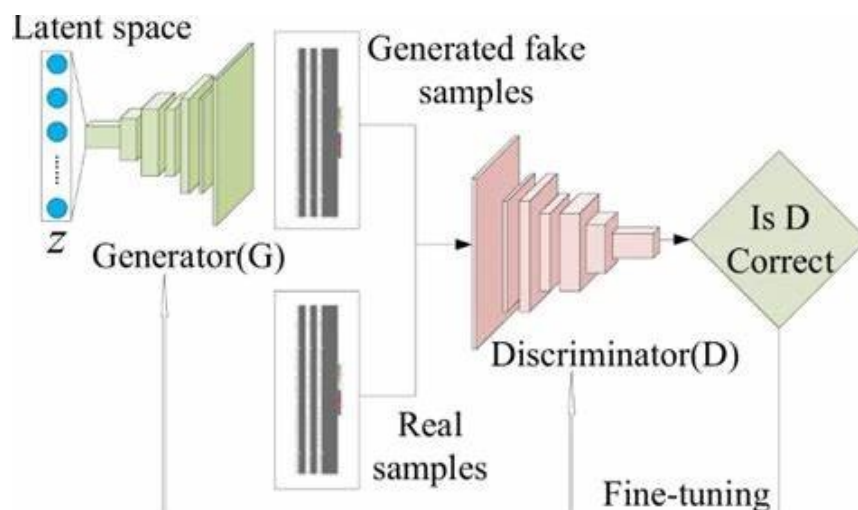
Классификационная нейронная сеть (или сверточная сеть) — сеть, которая используется для классификации изображений. Данная сеть имеет более сложную структуру чем два прошлых вида, вследствие того, что классификационной сети чаще всего подаются цветные изображения, в которых нужно распознать один из множества классов. На примере сети, которая будет строится в следующих пунктах, на вход подается изображение и сеть должна понять, что за цветок на ней изображен. Для распознавания этого сети необходимо выделять некоторые наиболее значимые детали изображения — такие как основание цветка, размеры лепестков, их наличие или отсутствие, а также многие другие факторы. Для этой задачи в классификационной сети используются определенные слои — сверточные (Convolution) и слои выделения наиболее важных параметров (MaxPool). Принципы работы каждого из этих слоев будут рассмотрены далее. Однако пример структуры такой сети приведен на изображении 5.

Изображение 5. Пример классификационной сети.



Помимо MLP, регрессионной и классификационной сетей существуют также генеративные сети, структура, принцип обучения и работы таких сетей зачастую сложнее и имеет иной подход, поэтому в данной работе такие сети не рассматриваются, поскольку они ничем не помогают в освоении материала для построения классификационной нейронной сети. Структура генеративной сети приведена на изображении 6.

Изображение 6. Структура генеративной сети.



Из изображения 6 видно, что сеть как бы состязуется сама с собой, где генерируется, допустим, изображение а сеть должна угадывать, что тут изображено.

1.3. Методы улучшения работы сетей.

Для улучшения работы сети можно использовать множество вариантов. Допустим, если качество распознавания не так важно, как скорость обучения, то можно попытаться уменьшить размерность входного изображения, вследствие чего тензоры уменьшаться, и, следовательно, ускорится их машинная обработка. Однако, такой способ уменьшает качество работы сети, что не есть хорошо. В таком случае можно рассмотреть другие методы, один из них — использование батчей. Батч — некоторый пакет или набор изображений.

Допустим, изначально планировалась обработка каждого изображения отдельно, в таком случае каждое изображение будет проходить через каждый

слой, что займет гораздо больше времени, чем обработка некоторого количества изображений одновременно. Это особенно заметно на трехмерных изображениях для нашей задачи. Поскольку на вход планируются подаваться изображения размерами $128 \times 128 \times 3$, то начальный размер тензора из одного изображения можем изобразить как (1, 128, 128, 3), в то время как батч из 16 изображений будет размерностью (16, 128, 128, 3). И вроде как ничего не поменялось, и кажется, что второй вариант будет обрабатываться в 16 раз дольше. Однако это далеко не так, поскольку существуют такие вещи, как параллельная обработка и векторизация.

Когда сеть обрабатывает сразу несколько изображений (батч), большинство операций (например, матричные умножения) могут быть эффективно выполнены с использованием параллельных вычислений на графических процессорах (GPU) или других ускорителях. Это значительно сокращает время обработки по сравнению с поочередной обработкой каждого изображения.

Однако, при сильно большом размере батча может забиться оперативная память, вследствие чего компьютер аварийно завершит работу и прекратится процесс обучения сети.

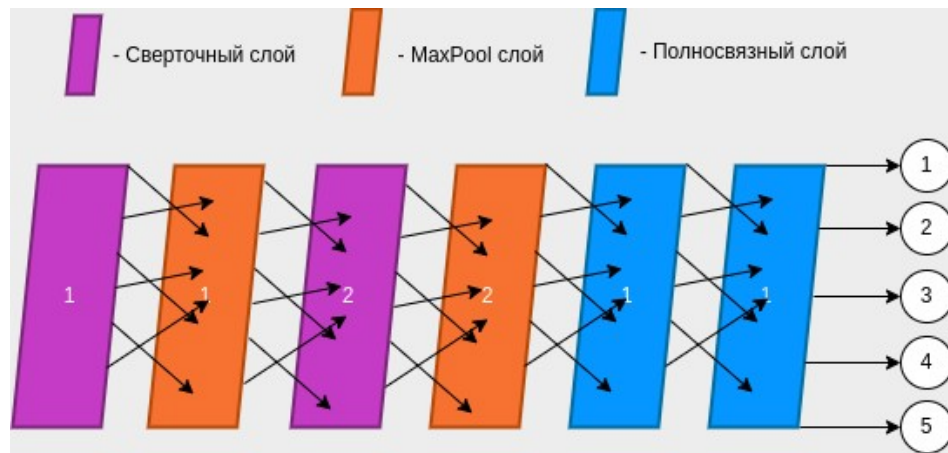
1.4. Архитектура исходной сверточной нейронной сети.

В случае нашей задачи, а именно распознавания и классификации цветов используется именно классификационная сеть (CNN).

Устройство такой сети отличается от MLP или регрессионной сетей. Как было сказано ранее, в CNN используются два дополнительных слоя — сверточный, и слой выделения наиболее важных параметров. Чаще всего эти два слоя следуют друг за другом, поскольку в комбинации они помогают ускорять обучение и сети и обращать внимание только на наиболее важные признаки. В нашем случае это может означать, что сеть просто не обращает внимание на фон изображения или какие-либо побочные объекты рядом, допустим пчел, траву.

В случае нашей сети было принято использовать структуру сети, указанную на изображении 7.

Изображение 7. Архитектура основной сети.



За счет такой архитектуры изображение сначала свертывается, выделяя наиболее важные параметры в сверточном слое, затем оттуда берутся наиболее важные параметры в слое MaxPool и так два раза. После чего значения передаются в полносвязный слой, где точно также как описано обрабатываются веса и само изображение и на выход идет один из пяти классов цветов.

1.5. Принцип работы исходной нейронной сети.

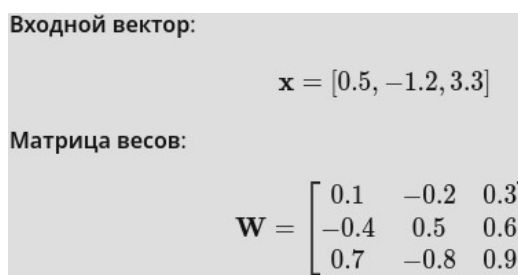
Поскольку архитектура сети уже выбрано, стоит разобрать принцип работы сети, а также принцип ее обучения.

Для начала стоит разбобраться с принципом работы каждого из слоев.

Начнем с полносвязного слоя. Суть полносвязного слоя в том, что он перемножает входные значения на веса, после чего пропускает их через функцию активации. Виды функции активаций будут рассмотрены чуть позже. В общем случае формула для метода работы сети forward выглядит так: $OutPut = x * W + b$, где OutPut — выходной тензор значений, x — входная матрица или скаляр, W — матрица весов, b — смещения. Примером для работы такого слоя можно взять простейшую имитацию сети. Допустим, у нас есть некоторый входной тензор размером (1, 3). Тогда матрица весов будет иметь размерность (3, 3), поскольку имеется три входных признака и три

нейрона. Для тестового примера возьмем смещения равные нулю. Изобразим данные тензоры со случайными значениями (изображение 8)

Изображение 8. Тензоры со случайными значениями.



Входной вектор:

$$\mathbf{x} = [0.5, -1.2, 3.3]$$

Матрица весов:

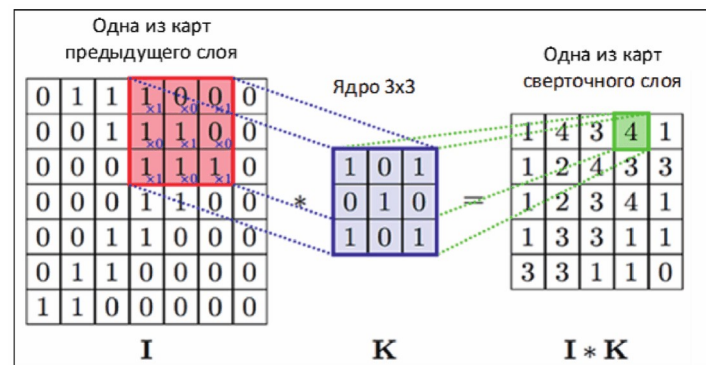
$$\mathbf{W} = \begin{bmatrix} 0.1 & -0.2 & 0.3 \\ -0.4 & 0.5 & 0.6 \\ 0.7 & -0.8 & 0.9 \end{bmatrix}$$

После этого, можем применить исходную формулу и получим выходной тензор: $[2.84, -3.34, 2.4]$. Тогда, если представим, что данная сеть была бы сверточной, то на выходе она бы дала нам третье значение или класс. Поскольку его вероятность выше всего.

Следующим слоем для обозревания будет сверточный. Он является основной отличительной чертой классификационной сети. Суть его работы заключается в том, что он принимает на вход некоторый тензор, после чего создает новый, путем матричного умножения, а именно, тензор фильтров домножается на подтензоры входного тензора. Что бы было чуть понятнее, рассмотрим пример. Допустим, у нас имеется некоторый входной тензор размерностью $(10, 10, 1)$. Тогда, нужно определиться с размером фильтров. Размер фильтра стоит выбирать опираясь на данную формулу: $\text{out} = ((s - \text{size}) / \text{stride}) + 1$, где out — выходная высота или ширина тензора, s — высота или ширина входного тензора, size — размер фильтра (его высота или ширина), stride — шаг, на который будет смещаться подтензор. Таким образом, если у нас имеется тот самый тензор размерами $(10, 10, 1)$, где 10 — высота и ширина, а 1 — количество каналов, то попробуем использовать для этого фильтр размером $(3, 3, 32)$, где 3 — высота и ширина фильтра, 32 — количество условных частиц, по которым фильтр будет обрабатывать и выделять некие черты изображения. Возьмем некоторый малый шаг, что бы не сильно уменьшить изображение, допустим 1. Тогда, итоговый размер выходного тензора будет: $\text{size} = (10 - 3) / 1 + 1 = 8$, получается, что итоговый

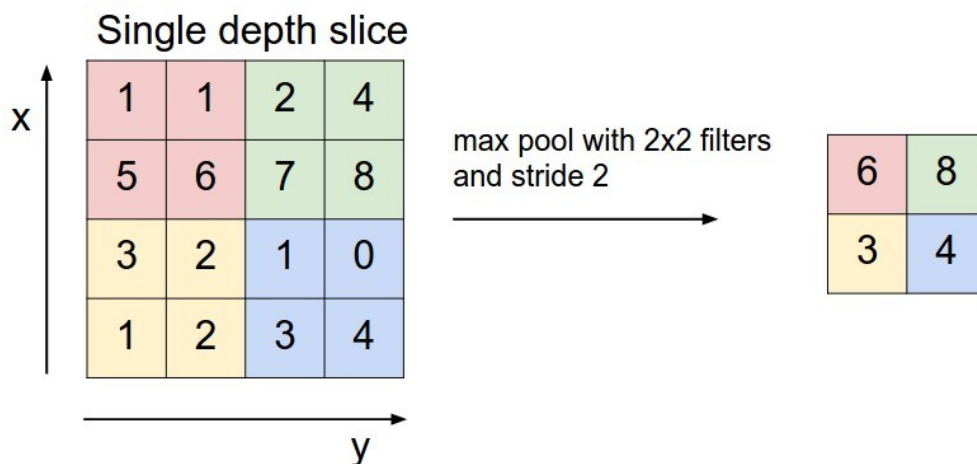
размер (8, 8, 32). Для основных вычислений используется формула: $T_i = \Sigma(r_i * c_i)$, где T_i — это элемент итоговой подматрицы, r_i — элемент по строке в подматрице, c_i — элемент по колонке в фильтре. Пример того, как происходят вычисления приведен на изображении 9. Как видно из данного изображения, каждое значение подматрицы карты предыдущего слоя — то есть некоторого изображения в случае классификации, умножается на каждое значение фильтра того же размера, затем все эти значения суммируются и записываются в соответствующую ячейку выходного тензора. После чего подматрица передвигается на шаг, который в данном случае равен одному.

Изображение 9. Пример вычисления в CL.



Еще один слой, который используется в данной сети — MaxPool. Суть данного слоя, как было написано ранее, заключается в том, что бы выделять наиболее значимые участки изображения, избегая ненужные. Тогда, рассмотрим пример работы такого слоя на изображении 10.

Изображение 10. Пример работы MaxPool слоя.



Из изображения видно, что суть работы заключается в том, что у нас, как и в случае с CL имеется некоторая условная матрица, по которой будет идти работа. То есть, точно так же мы выбираем некоторый регион из входного изображения, при этом размер этой матрицы так же будет: $\text{region} = (\text{size} - \text{pool}) / \text{stride} + 1$, где region — размерность итогового региона, size — входной размер, pool — некоторый параметр (высота или ширина), stride — шаг, на который будет смещаться регион.

После того, как стала понятна работа каждого слоя отдельно, стоит разобраться с таким параметром, как функция активации.

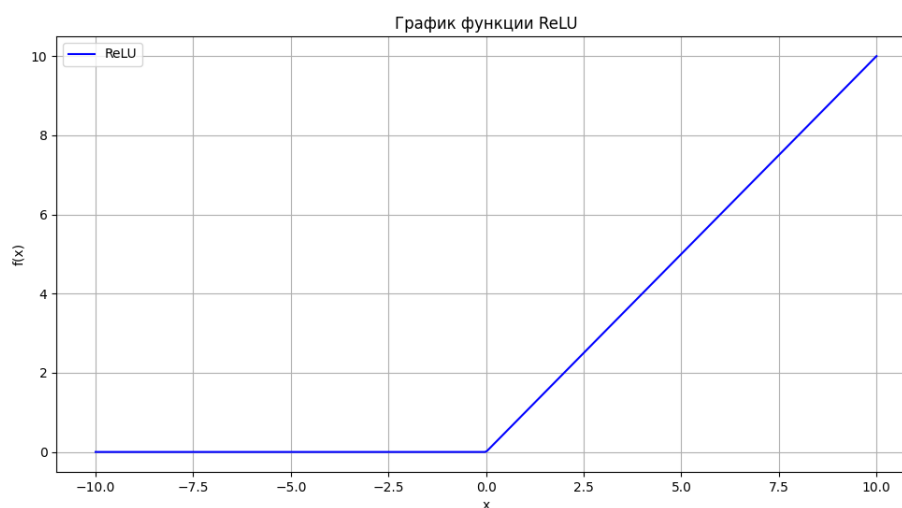
Функции активации — некоторые функции, которые принимают все тот же входной тензор, при этом обрабатывая его и приводя к более удобному для обучения и работы виду. Функций активации есть огромное множество, в данной работе использовались такие функции как softmax и ReLU. Softmax — функция активации, которая принимает значение и возвращает другое значение в пределах от 0 до 1, при этом всем, сумма всех выходных значений сети будет равна 1. Данная функция чаще всего используется для выходных слоев в сверточных сетях, поскольку, так как сумма всех нейронов равна 1, то каждое из значений будет иметь некоторое процентное отношение. В таком случае возвращается вероятность в удобном виде, и сеть выдает то значение, которое имеет наибольшую вероятность. Формула данной функции, а также ее график приведены ниже, на изображении 11. Из изображения приведенного ниже, можно заметить, что для некоторого случайного набора значений из 5 классов образуется 5 различных кривых, при этом, если приглядеться, то видно, что сумма всех пяти классов для одного $p(x)$ относительно оси x как раз будет равняться одному.

Изображение 11. Функция softmax для 5 классов.



Другая функция активации — ReLU, используется для внутренних, скрытых слоев. Данная функция удобна тем, что если значение меньше 0, то функция возвращает ноль, если же значение больше, то функция выдает это же значение. Получается, что преимущество данной функции в том, что она обнуляет ненужные нейроны, оставляя только значимые. Данная функция описывается такой формулой: $F(x) = \max(0, x)$. Так же график такой функции приведен на изображении 12.

Изображение 12. Функция ReLU.



Резумируя все выше сказанное, картинка на входе превращается в тензор значений, после чего проходит в слой свертки, откуда выделяются наиболее важные параметры изображения, допустим лепестки. После этого, в MaxPool слое так же выделяются только наиболее полезные участки изображения. После каждого из слоев используется функция активации, что бы придать

некоторую нелинейность сети. Затем, тензор переходит в полносвязный слой, где перемножается на веса связи между слоями, после чего это повторяется n раз. И в конце сеть получает 5 чисел, обрабатывает их функцией softmax и имеет на выходе 5 значений, где каждое из полученных значений имеет вероятность от 0 до 0.9. И выбирается наибольшее из них, после чего выводится результат предсказания сети.

1.6. Принцип обучения сети.

Поскольку принцип работы самого по себе распознавания того, что изображено на картинке уже разобран, стоит понять, как именно обучается сеть. Для этого необходимо разобрать такие понятия: градиент, градиентный спуск, функция потерь, обратное распространение ошибки, скорость обучения.

Стоит начать с функции потерь, поскольку в алгоритме обучения первая идет она. Функцией потерь, как и в случае с функциями активации, бесчисленное множество, где каждая подходит для своей задачи. В случае решения задачи распознавания изображения чаще всего используется функция кросс-энтропии. Вообще кросс-энтропийная функция выглядит так: $L = -\sum y_i \log(p_i + 1e-7)$, где L — итоговая потеря, y_i — метка класса для i (метка класса — показатель который принимает значение или 0 или 1, при этом 1 означает, что класс верный, 0 — класс неверный), p_i — предсказанные моделью вероятность для i . Сумма в кросс-энтропии выполняется от 1 элемента, до последнего элемента класса. Однако для задачи классификации с использованием батчей больше подходит чуть другая интерпретация, формула такой функции выглядит как-то так: $L = -\sum y_i \log(p_i + 1e-7) / \text{batch_size}$, где L — итоговая потеря, y_i — метка класса для i (что такое метка класса описано выше), p_i — предсказанные моделью вероятность для i , batch_size — размер батча. Данная функция хороша тем, что добавление небольшого числа (например, $1e-7$), чтобы избежать вычисления логарифма от нуля, делает кросс-энтропийную функцию потерь стабильной с точки зрения числовых

ошибок. Это важно, так как в процессе обучения нейронная сеть может предсказывать очень маленькие вероятности (очень близкие к нулю), что может привести к числовым ошибкам без такого добавления. Найдем градиент этой функции:

Для того чтобы вычислить градиент функции потерь по отношению к p_i , мы применяем дифференцирование:

$$\partial L / \partial p_i = \partial / \partial p_i (-\sum y_j \log(p_j))$$

Поскольку $\partial / \partial p_i \log(p_j)$ для каждого j не зависит от других, дифференцируем только соответствующий элемент:

$$\partial L / \partial p_i = -\partial / \partial p_i (y_i \log(p_i))$$

Теперь применяем стандартное правило дифференцирования для $\log(p_i)$:

$$\partial / \partial p_i \log(p_i) = 1/p_i$$

Таким образом, производная по p_i будет:

$$\partial L / \partial p_i = -y_i / p_i$$

Однако y_i — это либо 0, либо 1 (для одного правильного класса $y=[0,1,0]$ и т.д.). Поэтому для тех классов, для которых $y_i=1$, градиент будет:

$$\partial L / \partial p_i = -1/p_i$$

Для остальных классов, где $y_i=0$, градиент будет равен нулю:

$$\partial L / \partial p_i = 0$$

Таким образом, для класса i , для которого $y_i=1$, градиент будет пропорционален разнице между предсказанным значением p_i и истинной меткой, которая равна 1:

$$\partial L / \partial p_i = p_i - 1$$

А для классов с $y_i=0$ градиент равен:

$$\partial L / \partial p_i = p_i$$

Для многоклассовой классификации градиент функции потерь для каждого p_i будет:

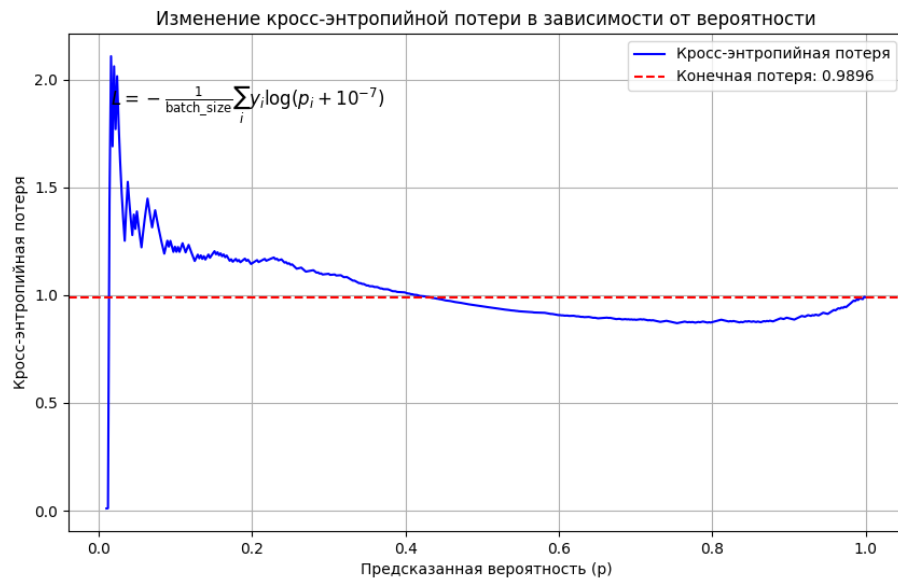
$$\partial L / \partial p_i = p_i - y_i$$

где: p_i — это предсказанная вероятность для класса i , y_i — это истинная метка (0 или 1) для этого класса.

Таким образом итоговое выражение для градиента: $\partial L / \partial p_i = p_i - y_i$.

Пример этой функции приведен на изображении 13.

Изображение 13. Пример работы функции потерь.



Данное изображение строилось из случайных данных, схожих на практические (данные далее указаны для каждого десятого значения в батче): $y = \{0, 1, 0, 1, 0, 0, 0, 0, 0, 1\}$, $p = \{0.01, 0.1092, 0.2084, 0.3076, 0.4068, 0.5060, 0.6052, 0.7044, 0.8036, 0.9028\}$, $\text{loss} = \{0, 2.2146, 0, 1.790, 0, 0, 0, 0, 0, 0.1023\}$. Из данного изображения видно, что потеря в начале обучения сети равна нулю, а затем начинает очень сильно скакать от 0 до 2. Однако со временем данные значения приходят в норму и потеря уменьшается. В данном случае потеря в конце достигла значения 0.9896.

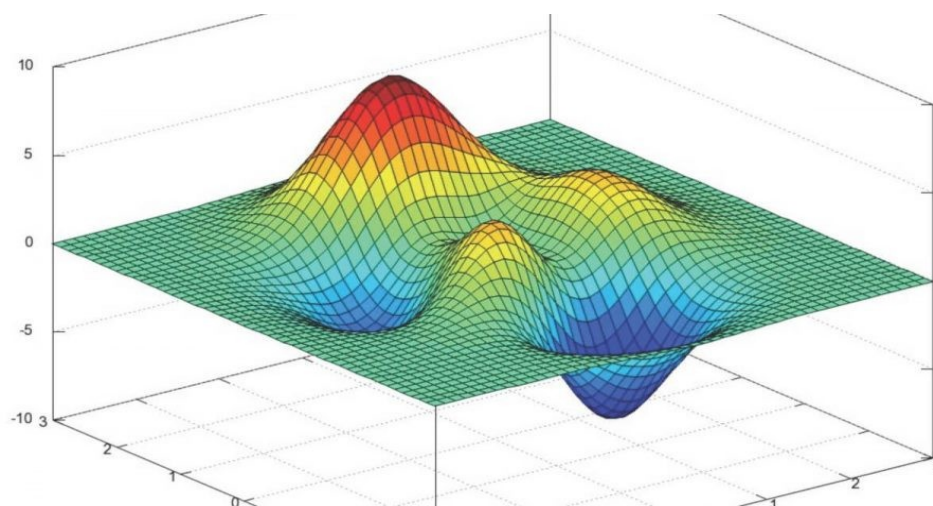
После того, как вычислена функция потерь, она передается обратно в сеть, начиная с конца и двигаясь к входному слою. Такой метод называется *back propagation* или обратное распространение. Для этого вычисляются градиенты весов в полносвязном слое и градиенты фильтров в сверточном слое.

Сами по себе, с математической точки зрения, градиенты высчитываются так: пусть $f(x, y) = 7x^2 + 3y$, тогда ее градиентом $\text{grad}(f(x, y)) = (df/dx, df/dy)$ в некоторой точке $M(x, y)$. Тогда для нашего

примера, градиентом будет: $\text{grad}(f(x,y)) = (14x, 3)$. Подставляя некоторую точку $M(1, 0)$ получаем, что градиент в данной точке будет равен $(14, 3)$. Градиент показывает путь, по которому функции растет с наибольшей скоростью.

Таким образом, градиент находит максимумы функции, однако, поскольку в сетях необходимо уменьшать ошибку, что бы точность росла, необходимо двигаться в другую сторону от градиента, что бы достичь минимума функции. Пример того, как выглядит градиентный спуск для некоторой функции представлен на изображении 14. На данном изображении видно некоторые возвышения и спуски, при этом возвышения показывают локальные максимумы функции, то есть ее максимальные значения, в то же время синие области (спуски) показывают минимальные значения функции или ее локальный минимум. В контексте машинного обучения это означает, что ошибка (по оси Z) уменьшается, а следовательно, увеличивается точность модели.

Изображение 14. Пример градиентного спуска.



Однако, в контексте машинного обучения используется немного иной подход к подсчету градиентов. В нашем случае, будет использоваться улучшенный метод градиентного спуска, а именно Adam. Данный метод использует более сложный подход, но зато выдает лучшие результаты на относительно небольших выборках данных. Его преимущество в том, что он

соединяет в себе градиентный спуск с моментумом и адаптивный шаг обучения. Обновление параметров по Adam выглядит как-то так:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w L(w_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla_w L(w_{t-1})^2$$

$$m'_t = m_t / (1 - \beta_1^t),$$

$$v'_t = v_t / (1 - \beta_2^t)$$

$$w_t = w_{t-1} - \eta * m'_t / (\sqrt{v'_t} + \epsilon)$$

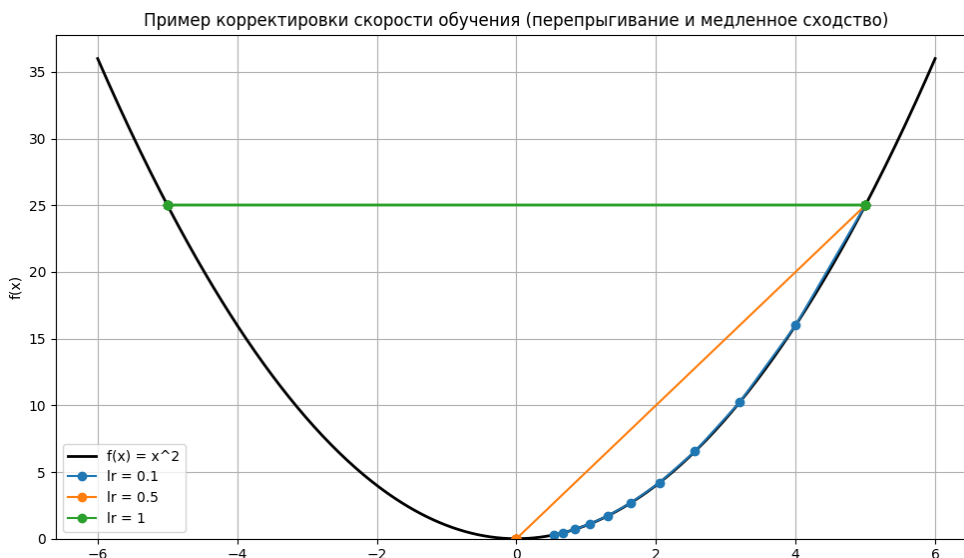
где: m_t — первый момент (экспоненциально взвешенное среднее градиента), v_t — второй момент (экспоненциально взвешенное среднее квадрата градиента), m'_t, v'_t — исправленные значения для устранения смещения на первых шагах, η — шаг обучения, ϵ — небольшая константа для числовой стабильности.

Отличие такого метода от градиентного спуска, как было сказано ранее, в том, что он использует состояния моментов и корректировку направления в текущем времени, другими словами, адаптирует шаги обучения для каждого параметра модели. Из формул выше можно доказать, что такой способ оптимизации будет приближаться к минимуму быстрее, нежели обычный градиентный спуск из-за того, что шаг не фиксированный и зависит от градиентов. Если градиенты слишком большие — Adam увеличивает шаг и быстрее идет к минимуму. При этом чем меньше градиенты — тем меньше шаги делает оптимизатор, что бы не перескочить минимум.

Корректировка весов происходит с помощью нахождения градиентов и использованием такого параметра, как скорость обучения. Скорость обучения — некоторый небольшой параметр, который подсказывает сети, с какой скоростью ей изменять веса. В контексте математики — это то, на какое расстояние от начальной точки мы изменяем положение в пространстве. Очень важно выбрать средний параметр для этого параметра, поскольку сильно большой параметр может мешать достичь минимума из-за того, что постоянно перепрыгивает его. Сильно малый же параметр сильно замедляет обучение сети, а так же имеет возможность застрять в локальном минимуме и

не достичь минимального значения всей функции. Пример этого приведен на изображении 15.

Изображение 15. Пример скорости подхода к локальному минимуму разными значениями скорости обучения.



Такая иллюстрация наглядно позволяет заметить, что если шаг сильно большой, как в случае с зеленым отрезком, то сеть никогда не достигнет локального минимума. Если же шаг подобран сильно небольшим, как показано на синем отрезке. Однако, если же выбрать достаточно хороший шаг для сети, то можно максимально приблизиться к этому самому минимуму, как в примере с оранжевым отрезком. Это показывает, что выбор шага для градиентного спуска очень важен, ведь хорошо подобранный шаг позволяет сети быстрее достичь локального минимума.

Однако, поскольку в текущей работе используется адаптивный метод моментов, то данное строение не столько важно.

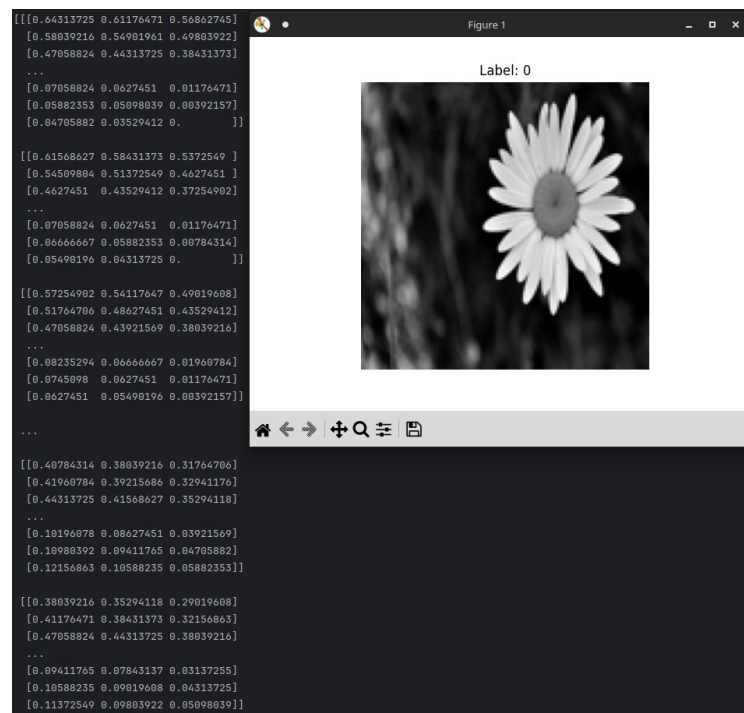
2. Практическая часть.

2.1. Предобработка изображений.

Поскольку компьютер умеет работать только с числами, стоит понять, как преобразовать входное изображение. Так как сеть должна иметь фиксированное количество нейронов, то стоит преобразовать все изображения

к одному общему размеру, чаще всего к некому квадрату, допустим, размерами 64x64 или 128x128. Поскольку известно, что изображение — набор пикселей размерностями от (0, 0, 0) до (255, 255, 255) в RGB формате, где (0, 0, 0) — черный цвет, а (255, 255, 255) — белый, то можно заметить, что если привести изображение к 3-х мерной матрице, где одна ось — высота, вторая — ширина, третья — количество каналов (в случае RGB — 3). Тогда мы получим некоторый тензор, в котором каждое значение варьируется от 0 до 255, однако, такой вид замедляет вычисления в сети, поэтому можно попробовать поделить каждое значение на 255, таким образом значения в тензоре будут варьироваться от 0.0 до 1.0, что упрощает работу компьютеру и практически убирает возможность переполнения в памяти. Пример такого преобразования приведен на изображении 16.

Изображение 16. Пример преобработки изображения.



2.2. Подготовка и расчет сети.

Изображения подготовлены, теперь стоит разделить выборку в размере 90/10, где 90% - это обучающая выборка, остальные 10% - выборка для тестов. Это необходимо, что бы проверить реальную способность сети к

распознаванию, а так же для того, что бы проверить, что сеть реально распознает изображения, а не просто запоминает их.

Изображения готовы, выборки созданы, после чего было принято использовать работу по батчам для ускорения обучения и экономии ресурсов. Поскольку всего имеется 3306 изображений, батч будет составлять 29 изображений, поскольку 3306 без остатка делится на 29.

Размеры для сетей были рассчитаны так же, как описано в теоретической части, итоговые размеры для каждого слоя представлены в таблице 1.

Таблица 1. Размеры для каждого слоя.

Название слоя	Размеры входные	Размеры выходные	Размер фильтра	Шаг
Сверточный 1	-	-	(2, 2, 3, 32)	2
MaxPool 1	-	-	(2, 2)	2
Сверточный 2	-	-	(2, 2, 32, 64)	2
MaxPool 2	-	-	(2, 2)	2
Полносвязный 1	4096	128	-	-
Полносвязный 2	128	5	-	-

Помимо описанных выше параметров использовалась также скорость обучения — 0.01.

Размерности были выбраны таким образом, что бы все сходилось и соблюдалось основное условие матричного умножения.

2.3. Обучение сети.

Обучение одной эпохи (обработка всей выборки изображений 1 раз) занимает в среднем 62 минуты, что довольно таки много. Это значение можно улучшить путем переноса некоторых вычислений на GPU или использование сервера с более мощными составляющими, однако, в учебных целях не требуется сверх-точная модель. Именно поэтому количество эпох было

выбрано так, что бы точность модели была в пределах 60-70%, итоговое количество эпох составило 20.

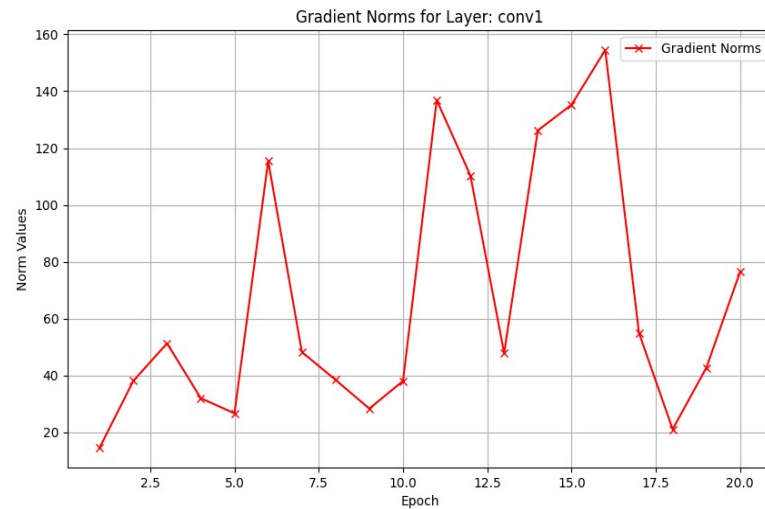
Время обучения, потерю, а также точность модели можно увидеть в таблице 2.

Таблица 2. Параметры каждой из эпох.

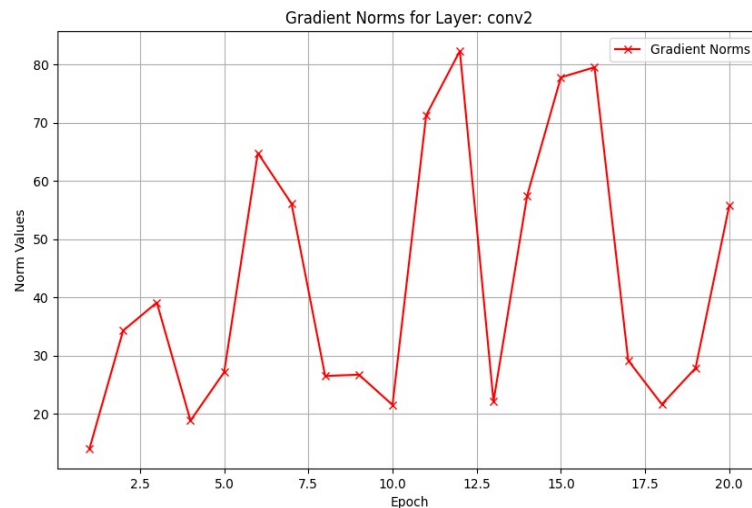
Номер эпохи	Потеря	Точность
1	1.6142	25.01%
2	1.4600	34.19%
3	1.3326	42.80%
4	1.3497	38.54%
5	1.1928	42.14%
6	1.1378	48.24%
7	1.1843	47.76%
8	1.0816	55.30%
9	1.0882	54.96%
10	1.0805	56.09%
11	0.9811	62.04%
12	1.3311	49.94%
13	0.9381	62.66%
14	0.9936	62.06%
15	0.9859	58.62%
16	0.9477	65.99%
17	1.0729	55.47%
18	0.9761	58.49%
19	0.8326	73.17%
20	0.7500	74.65%

Все данные собраны, модель работает, теперь стоит построить графики и убедиться в том, что модель действительно стремится к локальному минимуму, далее, на графиках (изображения 17, 18, 19, 20) и модели (изображение 21) это все показано.

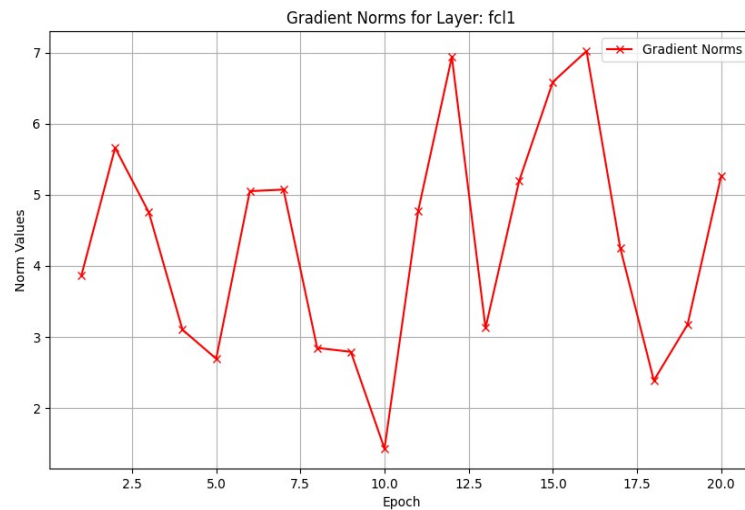
Изображение 17. График обучения сверточного слоя 1.



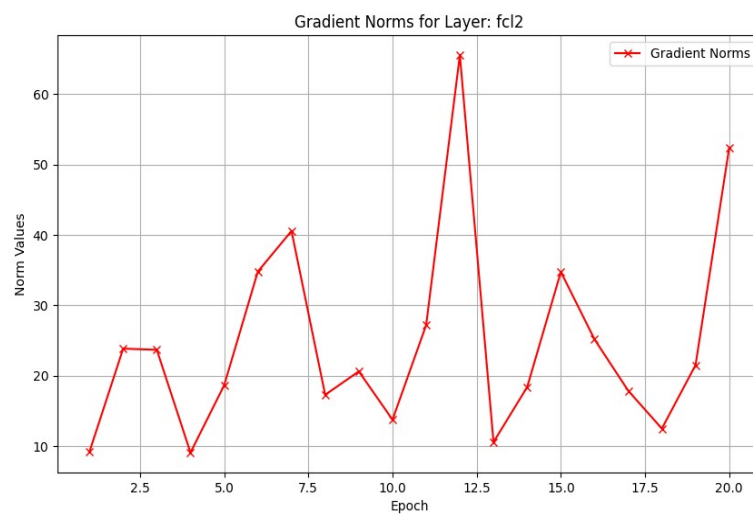
Изображение 18. График обучения сверточного слоя 2.



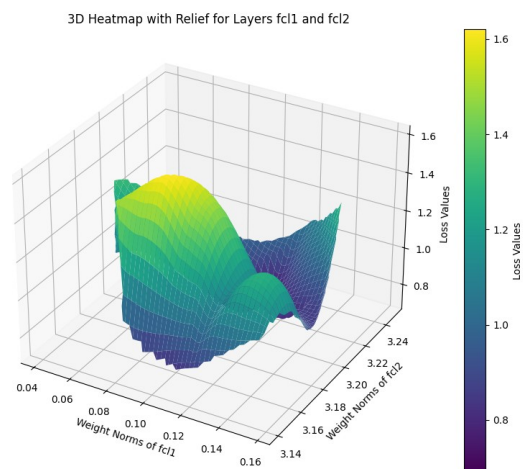
Изображение 19. График обучения полносвязного слоя 1.



Изображение 20. График обучения полносвязного слоя 2.



Изображение 21. Градиентный спуск для полносвязных слоев.



Как можно заметить из изображений 17-20 градиенты слоев скачут, данное явление в машинном обучении может указывать на скачки в точности в предсказаниях и возможности переобучения модели — явление, при котором модель на обучающей выборке показывает хорошие значения, однако на тестовых примерах может показать не очень удовлетворительные значения. Такие явления могут быть вызваны слишком высокой скоростью обучения, использованием батчей, в которых могут быть некоторые шумы или некачественные фотографии и многое другое. В нашем случае данное явление не является критическим, поскольку модель не должна выдавать какие-то хорошие значения точности.

Также из приведенной модели на изображении 21 видно, что значение ошибки зависит от весов двух выходных полносвязных слоев, при этом, в минимуме функции выдается максимальная точность сети. Ошибка модели действительно стремится к нулю по мере изменения весов и фильтров сети в процессе обучения с использованием Adam, как это было указано в теоретической части работы.

После обучения модели стоит проверить ее на тестовой выборке, которую модель еще не видела, результат показал, что точность модели на новых изображениях выдает 78.0220% точности, что очень даже неплохо для 20 эпох при такой выборке значений.

Сами примеры работы такой сети, включая удачные и неудачные приведены на изображениях ниже (изображение 22, 23, 24).

Изображение 22. Правильное предсказание сети 1.



Изображение 23. Неверное предсказание сети.

True Image



Prediction: sunflowers



Изображение 24. Правильное предсказание сети 2.

True Image



Prediction: sunflowers



Код программы

```
import os.path
import random
from PIL import Image
import numpy as np
import time
import matplotlib.pyplot as plt
from tqdm import tqdm

classes = {"daisy": 0, "dandelion": 1, "roses": 2, "sunflowers": 3, "tulips": 4}

def relu(x):
    return np.maximum(0, x)

class AdamOptimizer:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-7):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = None
        self.v = None
        self.t = 0

    def update(self, weights, grads, t):
        """
        Обновляет веса с использованием алгоритма Adam.
        :param weights: текущие веса
        :param grads: градиенты
        :param t: текущий шаг обучения
        :return: обновленные веса
        """
        if self.m is None:
            self.m = np.zeros_like(weights)
            self.v = np.zeros_like(weights)
```

```

        self.t += 1
        self.m = self.beta1 * self.m + (1 - self.beta1) * grads
        self.v = self.beta2 * self.v + (1 - self.beta2) * np.square(grads)

        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)

        weights -= self.learning_rate * m_hat / (np.sqrt(v_hat) + self.epsilon)

    return weights

def get(self):
    return {
        'm': self.m.tolist(),
        'v': self.v.tolist(),
        't': self.t,
        'beta1': self.beta1,
        'beta2': self.beta2
    }

def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def invert_image(image_path):
    image = Image.open(image_path).resize((128, 128))
    image_arr = np.array(image) / 255.0
    return image_arr

def load_data(path):
    learn_data = []
    train_data = []

    for name, label in classes.items():

```

```

folder_path = os.path.join(path, name)
batch_of_files = os.listdir(folder_path)

for_train = len(batch_of_files) // 10

for idx, file in enumerate(batch_of_files):
    if file.endswith(".jpg"):
        image_path = os.path.join(folder_path, file)
        image = Image.open(image_path).resize((128, 128))
        image_arr = np.array(image) / 255.0

        if idx < for_train:
            train_data.append((image_arr, label))
        else:
            learn_data.append((image_arr, label))

random.shuffle(learn_data)
random.shuffle(train_data)

learn_images = np.array([item[0] for item in learn_data])
learn_labels = np.array([item[1] for item in learn_data])

train_images = np.array([item[0] for item in train_data])
train_labels = np.array([item[1] for item in train_data])

return (learn_images, learn_labels), (train_images, train_labels)

```

```

class FullConnectedLayer:
    def __init__(self, input_size, output_size, history_file, activation=relu,
learning_rate=0.01):
        self.input_size = input_size
        self.output_size = output_size
        self.weights = np.random.randn(input_size, output_size) * np.sqrt(2 /
input_size)
        self.biases = np.zeros((1, output_size))
        self.activation = activation
        self.history_file = history_file

```



```

self.learning_rate = learning_rate
self.grads = {}
self.input_data = None
self.d_weights = None
self.d_biases = None
self.d_input = None

self.optimizer = AdamOptimizer(learning_rate=learning_rate)

def calculate_grads(self, d_out, learning_rate=0.01):
    self.d_weights = np.dot(self.input_data.T, d_out)
    self.d_biases = np.sum(d_out, axis=0, keepdims=True)

    self.d_input = np.dot(d_out, self.weights.T)

    count = self.input_data.shape[0]

    self.d_input /= count
    self.d_weights /= count
    self.d_biases /= count

    # Обновляем веса с помощью Adam
    self.weights = self.optimizer.update(self.weights, self.d_weights,
self.optimizer.t)
    self.biases -= self.learning_rate * self.d_biases # Для biases можно
оставить обычное обновление

    self.grads = {
        'input': self.d_input,
        'weights': self.d_weights,
        'biases': self.d_biases
    }

    return self.d_input

def get_weights(self):
    return self.weights

```

```

def forward(self, input_data):
    input_x = input_data.reshape(input_data.shape[0], -1)
    self.input_data = input_x

    z = np.dot(input_x, self.weights) + self.biases

    output = self.activation(z) if self.activation else z # функция активации
    return output

def save_history(self, history_file):
    weights_history = []
    biases_history = []
    grads_history = []

    weights_history.append(self.weights.copy())
    biases_history.append(self.biases.copy())

    grads_history.append({
        'input': self.grads['input'].tolist(),
        'weights': self.grads['weights'].tolist(),
        'biases': self.grads['biases'].tolist()
    })

    np.savez(history_file,
             weights=weights_history,
             biases=biases_history,
             grads=grads_history)

    print(f"Сохранена информация о weights, biases, grads в файл:
{history_file}")

class MaxPoolLayer:
    """Слой для уменьшения изображения и выявления наиболее полезных
    параметров"""
    def __init__(self, pool_size, stride=None):
        self.pool_size = pool_size
        self.stride = stride if stride is not None else pool_size

```

```

self.input_shape = None
self.forward_cache = None

def forward(self, input_data):
    self.input_shape = input_data.shape
    (batch_size, height, width, channels) = self.input_shape
    try:
        pool_height, pool_width = self.pool_size
    except:
        pool_height = self.pool_size
        pool_width = self.pool_size
    stride = self.stride

    out_height = (height - pool_height) // stride + 1
    out_width = (width - pool_width) // stride + 1

    self.forward_cache = input_data
    output = np.zeros((batch_size, out_height, out_width, channels))

    for b in range(batch_size):
        for c in range(channels):
            for i in range(out_height):
                for j in range(out_width):
                    start_i, start_j = i * stride, j * stride
                    end_i, end_j = start_i + pool_height, start_j + pool_width
                    output[b, i, j, c] = np.max(input_data[b, start_i:end_i,
start_j:end_j, c])

    return output

def backward_prop(self, d_out):
    (batch_size, height, width, channels) = self.input_shape
    pool_height, pool_width = self.pool_size
    stride = self.stride
    d_input = np.zeros(self.input_shape)

    h_e = (height - pool_height) // self.stride + 1
    w_e = (width - pool_width) // self.stride + 1

```

```

d_out_reshaped = d_out.reshape(batch_size, h_e, w_e, channels)

for b in range(batch_size):
    for c in range(channels):
        for i in range(h_e):
            for j in range(w_e):
                start_i, start_j = i * stride, j * stride
                end_i, end_j = start_i + pool_height, start_j + pool_width

                # Получаем текущий регион из входных данных
                input_region = self.forward_cache[b, start_i:end_i,
start_j:end_j, c]

                # Маска для элементов, которые были максимальными
                mask = (input_region == np.max(input_region))

                # Градиент распространяется только на максимальные элементы
                d_input[b, start_i:end_i, start_j:end_j, c] += mask *
d_out_reshaped[b, i, j, c]

return d_input

```

```

class ConvolutionLayer: # maybe ready
    """Слой для свертки изображения"""

    def __init__(self, filter_size, num_filters, num_channels=3, stride=2,
learning_rate=0.01, activation=relu):
        self.num_filters = num_filters
        self.num_channels = num_channels
        self.stride = stride
        self.filter_size = filter_size
        self.filters = np.random.randn(filter_size, filter_size, num_channels,
num_filters) * np.sqrt(2 / filter_size)
        self.biases = np.zeros((num_filters,))
        self.learning_rate = learning_rate
        self.activation = activation
        self.optimizer = AdamOptimizer(learning_rate=learning_rate)
        self.image = None
        self.grads = {}

```

```

def forward(self, image):
    count, h, w, channels = image.shape
    self.image = image
    h_out = (h - self.filter_size) // self.stride + 1
    w_out = (w - self.filter_size) // self.stride + 1
    out = np.zeros((count, h_out, w_out, self.num_filters))

    for b in range(count):
        for f in range(self.num_filters):
            for i in range(0, h_out):
                for j in range(0, w_out):
                    h_start, w_start = i * self.stride, j * self.stride
                    h_end, w_end = h_start + self.filter_size, w_start +
self.filter_size

                    region = image[b, h_start:h_end, w_start:w_end, :]
                    out[b, i, j, f] = np.sum(region * self.filters[:, :, :, f]) +
self.biases[f]

    return self.activation(out)

def backward_prop(self, grad_out):
    count, h_image, w_image, c_image = self.image.shape
    _, h_out, w_out, num_filters = grad_out.shape

    d_filters = np.zeros_like(self.filters)
    d_biases = np.zeros_like(self.biases)
    d_input = np.zeros_like(self.image)

    for i in range(count):
        for f in range(num_filters):
            for k1 in range(h_out):
                for k2 in range(w_out):
                    h_start = k1 * self.stride
                    h_end = h_start + self.filter_size
                    w_start = k2 * self.stride
                    w_end = w_start + self.filter_size

                    region = self.image[i, h_start:h_end, w_start:w_end, :]

```

```

        d_filters[:, :, :, f] += region * grad_out[i, k1, k2, f]

        d_input[i, h_start:h_end, w_start:w_end, :] +=
self.filters[:, :, :, f] * grad_out[i, k1, k2, f]

        d_biases[f] += np.sum(grad_out[i, :, :, f])

d_biases /= count
d_filters /= count
d_input /= count

self.grads = {
    'input': d_input,
    'filters': d_filters,
    'biases': d_biases
}

        self.filters = self.optimizer.update(self.filters, d_filters,
self.optimizer.t)
        self.biases -= self.optimizer.learning_rate * d_biases

return d_input

def save_history(self, history_file):
    weights_history = []
    biases_history = []
    grads_history = []

    weights_history.append(self.filters.copy())
    biases_history.append(self.biases.copy())

    grads_history.append({
        'input': self.grads['input'].tolist(),
        'filters': self.grads['filters'].tolist(),
        'biases': self.grads['biases'].tolist()
    })

# Сохраняем данные в файл

```

```

np.savez(history_file,
          weights=weights_history,
          biases=biases_history,
          grads=grads_history)

print(f"Сохранена информация о filters, biases, grads в файл:
{history_file}")

```

```

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
                  filter_size, num_filters,
                  pool_size, stride,
                  learning_rate=0.01, activation=relu):
        self.conv1 = ConvolutionLayer(filter_size, num_filters // 2, 3, stride,
                                       learning_rate)
        self.mpl1 = MaxPoolLayer((pool_size, pool_size), stride)
        self.conv2 = ConvolutionLayer(filter_size, num_filters, num_filters //
                                       2, stride, learning_rate)
        self.mpl2 = MaxPoolLayer((pool_size, pool_size), stride)
        self.fcl = FullConnectedLayer(hidden_size, 128, "history_fcl1.npz",
                                       activation, learning_rate)
        self.fcl2 = FullConnectedLayer(128, output_size, "history_fcl2.npz",
                                       learning_rate=learning_rate)

    def forward(self, x):
        x = self.conv1.forward(x)
        x = self.mpl1.forward(x)
        x = self.conv2.forward(x)
        x = self.mpl2.forward(x)
        x = self.fcl.forward(x)
        x = self.fcl2.forward(x)
        x = softmax(x)
        return x

    def backward_prop(self, d):
        d = self.fcl2.calculate_grads(d)
        d = self.fcl.calculate_grads(d)
        d = self.mpl2.backward_prop(d)
        d = self.conv2.backward_prop(d)

```

```

d = self.mpl1.backward_prop(d)
d = self.conv1.backward_prop(d)
return d

def train(self, images, labels, save, epochs=10, batch_size=29):
    num_samples = images.shape[0]
    losses_e = []
    losses_b = []
    accuracies_b = []
    accuracies_e = []
    weights_b = []
    weights_e = []
    save_dir = "saves"
    query = input("Загрузить уже готовые веса для модели? (y/n) - ")
    if query == "y":
        self.load_model(f"{save_dir}/full_model.npz")

    with open("Accuracy.txt", "w") as f:
        for epoch in range(epochs):
            time_s = time.time()
            total_loss = 0
            correct_predictions = 0
            epoch_weights = []
            epoch_losses = []

            # Перемешивание данных в начале каждой эпохи
            indices = np.arange(num_samples)
            np.random.shuffle(indices)
            images = images[indices]
            labels = labels[indices]

            for batch_start in tqdm(range(0, num_samples, batch_size)):
                acc = []
                loss_b = []
                batch_weights = []

                time_s_b = time.time()
                batch_end = min(batch_start + batch_size, num_samples)
                image_batch = images[batch_start:batch_end]
                label_batch = labels[batch_start:batch_end]

```



```

        predictions = self.forward(image_batch)

        one_hot_labels = np.zeros_like(predictions)
        one_hot_labels[np.arange(label_batch.size), label_batch] = 1

        loss = -np.sum(one_hot_labels * np.log(predictions + 1e-7)) /
batch_size

        total_loss += loss
        loss_b.append(loss)

    now_correct_predictions = np.sum(np.argmax(predictions, axis=1)
== label_batch)

    correct_predictions += now_correct_predictions

    grad_loss = predictions - one_hot_labels

    self.backward_prop(grad_loss)

    batch_weights.append(self.fcl.get_weights())

    time_e_b = time.time()

    weights_b.append(batch_weights)
    losses_b.append(loss_b)

time_e = time.time()

average_loss = total_loss / (num_samples // batch_size)
accuracy = correct_predictions / num_samples

epoch_weights.append(self.fcl.get_weights())
weights_e.append(epoch_weights)
epoch_losses.append(average_loss)
losses_e.append(epoch_losses)

self.save_model(save)

```

```

        self.fcl.save_history(f"{save_dir}/fcl1_history_{epoch + 1}.npz")
        self.fcl2.save_history(f"{save_dir}/fcl2_history_{epoch + 1}.npz")
        self.conv1.save_history(f"{save_dir}/conv1_history_{epoch +
1}.npz")
        self.conv2.save_history(f"{save_dir}/conv2_history_{epoch +
1}.npz")

        print(f"Epoch {epoch + 1}/{epochs} - Loss: {average_loss:.4f},
Accuracy: {accuracy:.4%}\n")
        f.write(f"Time: {time_e - time_s} sec.")
        f.write(f"Epoch {epoch + 1}/{epochs} - Loss: {average_loss:.4f},
Accuracy: {accuracy:.4%}\n")
        f.write(f"Time: {time_e - time_s} sec.")
    print("Learn of model is ready! Enjoy!")

def save_model(self, file_path):
    """
    Сохраняет параметры модели в файл.
    :param file_path: путь к файлу для сохранения
    """
    model_data = {
        "conv1_filters": self.conv1.filters,
        "conv1_biases": self.conv1.biases,

        "mpl1_pool_size": self.mpl1.pool_size,
        "mpl1_stride": self.mpl1.stride,

        "conv2_filters": self.conv2.filters,
        "conv2_biases": self.conv2.biases,

        "mpl2_pool_size": self.mpl2.pool_size,
        "mpl2_stride": self.mpl2.stride,

        "fcl_weights": self.fcl.weights,
        "fcl_biases": self.fcl.biases,

        "fcl2_weights": self.fcl2.weights,
        "fcl2_biases": self.fcl2.biases,
    }

```

```

np.savez(file_path, **model_data)
print(f"Модель сохранена в файл: {file_path}")

def load_model(self, file_path):
    """
    Загружает параметры модели из файла.
    :param file_path: путь к файлу для загрузки
    """
    model_data = np.load(file_path, allow_pickle=True)
    self.conv1.filters = model_data["conv1_filters"]
    self.conv1.biases = model_data["conv1_biases"]
    self.conv1.activation = relu
    self.mpl1.pool_size = model_data["mpl1_pool_size"]
    self.mpl1.stride = model_data["mpl1_stride"]
    self.conv2.filters = model_data["conv2_filters"]
    self.conv2.biases = model_data["conv2_biases"]
    self.conv2.activation = relu
    self.mpl2.pool_size = model_data["mpl2_pool_size"]
    self.mpl2.stride = model_data["mpl2_stride"]
    self.fcl.weights = model_data["fcl_weights"]
    self.fcl.biases = model_data["fcl_biases"]
    self.fcl.activation = relu
    self.fcl2.weights = model_data["fcl2_weights"]
    self.fcl2.biases = model_data["fcl2_biases"]

    print(f"Модель загружена из файла: {file_path}")

def use(self, image_path):
    image = invert_image(image_path)
    image = np.expand_dims(image, axis=0)
    true_image = Image.open(image_path)
    prediction = self.forward(image)
    result = np.argmax(prediction, axis=1)

    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    axes[0].imshow(true_image)
    axes[0].axis('off')
    axes[0].set_title("True Image")

```

```

axes[1].imshow(true_image)
axes[1].axis('off')
axes[1].set_title(f"Prediction: {list(classes.keys())[result[0]]}")

plt.tight_layout()
plt.show()

def test_model(model, test_images, test_labels):
    """
    Функция для тестирования обученной модели на тестовом наборе данных.

    :param model: обученная модель (экземпляр NeuralNetwork)
    :param test_images: изображения для тестирования (матрица данных)
    :param test_labels: метки классов для тестовых изображений
    :return: точность на тестовом наборе данных
    """
    num_samples = test_images.shape[0]
    correct_predictions = 0

    for i in range(num_samples):
        if i % 100 == 0:
            print(f"Testing: {i}/{num_samples} samples processed")

        prediction = model.forward(test_images[i:i+1, :, :, :])

        predicted_class = np.argmax(prediction, axis=1)

        if predicted_class == test_labels[i]:
            correct_predictions += 1

    accuracy = correct_predictions / num_samples
    print(f"Test Accuracy: {accuracy:.4%}")

    return accuracy

```

```

def main():
    # Загрузка данных
    learn, train = load_data("flower_photos")
    learn_images, learn_labels = learn
    train_images, train_labels = train

    real_model = "saves/full_model.npz"

    output_size = len(classes)
    input_size = 128*128
    filter_size = 2
    num_filters = 64
    hidden_size = 4096
    pool_size = 2
    stride = 2

    model = NeuralNetwork(input_size, hidden_size, output_size,
                           filter_size, num_filters, pool_size, stride)

    # model.train(learn_images, learn_labels, real_model)

    model.load_model("saves/full_model.npz")
    # model.use("/home/zamni/PycharmProjects/neuro/flower_photos/tulips/
65347450_53658c63bd_n.jpg")
    test_model(model, train_images, train_labels)

if __name__ == "__main__":
    main()

```

Заключение

В ходе выполнения курсовой работы была разработана и реализована сверточная нейронная сеть для распознавания одного из пяти видов цветков. Сеть состоит из двух сверточных слоев, двух полносвязных слоев и двух слоев максимального пула, что позволило эффективно извлекать признаки из изображений и классифицировать их. Реализация выполнена без использования специализированных библиотек, что позволило лучше понять внутренние процессы работы нейронных сетей, такие как свертка, пулинг, активация и обучение.

Для подготовки данных были выполнены операции предобработки изображений, что включало их нормализацию и преобразование в нужный формат для подачи на вход сети. Процесс обучения включал настройку параметров модели, использование функции потерь и оптимизатора для улучшения точности классификации.

Эксперименты показали, что предложенная модель успешно распознает цветки с достаточно высокой точностью. Несмотря на отсутствие использования сторонних библиотек, удалось реализовать все необходимые компоненты сверточной сети и добиться стабильных результатов.

Для дальнейших улучшений можно рассмотреть внедрение более сложных архитектур, использование различных методов регуляризации и расширение набора данных для повышения обобщающих способностей сети.

Ссылка на репозиторий GitHub

<https://github.com/ZamniProg/neuro>