**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

**Оглавление**

[**Введение 3**](#_Введение)

1. [**Теоретическая часть 4**](#_Теоретическая_часть)
   1. [Бинарное дерево поиска 4](#_Бинарное_дерево_поиска)
   2. [AVL дерево 7](#_AVL_дерево)
   3. [Красно-черное дерево 10](#_Красно-черное_дерево)
2. [**Практическая часть 15**](#_Сортировка_слиянием)
   1. [Зависимость высоты для бинарного дерева поиска 15](#_Зависимость_высоты_для)
   2. [Зависимость высоты для AVL дерева 17](#_Сортировка_выбором_1)
   3. [Зависимость высоты для красно-черного дерево](#_Сортировка_пузырьком_1) 18
   4. [Реализация обхода деревьев 20](#_Реализация_обхода_деревьев)

[**Код** **программы 23**](#_Код_программы)

[Заключение 3](#_Заключение)7

[**Ссылка на**](#_Ссылка_на_репозиторий)[**репозиторий GitHub 3**](#_Ссылка_на_репозиторий)**7**

# 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.

# 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 может использовать параллельные вычисления для обработки большого количества данных.

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.

Изображение 1. Пример нейрона сети.

Тут картинка

Следующим составляющим 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 разные 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.

Тут картинка

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 нейроннной сети так же, как и человеческие нейроны, а именно — помогают какие-то нейронны 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.

Изображение 3. Пример весов нейронной сети.

Тут картинка

1.2. Виды нейронных сетей.

Всего видов нейронных сетей — бесчисленное множество. Например, для распознавания базовой задачи в обучении сетей MNIST (набор рукописых 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 одной цифры из набора MNIST приведен на изображении 4.

Изображение 4. Пример цифры набора MNIST

Тут картинка

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. Пример работы такой сети приведен на изображении 5.

Изображение 5. Пример работы регрессионной нейронной сети.

Тут картинка

Классификационная нейронная сеть (или сверточная сеть) — сеть, которая используется для классификации изображений. Данная сеть имеет более сложную структуру чем два прошлых вида, вследствии того, что класификационной 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 изображение — 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.

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.

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, вследствии 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 128х128х3, то начальный размер тензор из одного изображения можем изобразить как (1, 128, 128, 3), в то время как батч из 16 изображений будет рамзерностью (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, вследствии чего компьютер аварийно завершит работу и прекратится процесс обучения сети.

Также, помимо основных слоев сети, могут использоваться еще и другие виды, такие как BatchNormalization, DropOut.

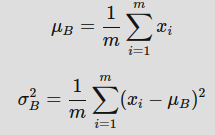
BatchNormalization — слой, который нормализует входы каждого из слоев в рамках каждого батча. То есть, для каждого слоя в сети вычисляется среднее значение и стандартное отклонение на всех входах по каждому признаку, в рамках батча. Данные значения далее используются для того, что бы нормализовать вхдоные данные слоя, то есть вычитается среднее значение и делится на стандартное отклонение. Далее, с помощью полей масштабирования (гамма) и смещения (бета), эти нормализованные данные масштабируются и сдвигаются, что бы сеть могла лучше обучаться. Пример работы показан на изображении 6.

Изображение 7. Пример работы BatchNormalization.

Тут картинка

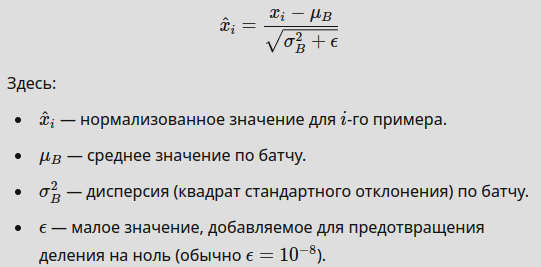
Рассмотрим поподробнее принцип работы BatchNormalization. Допустим, у нас есть набор входных данных x=[x1​,x2​,...,xm​], где m — это размер батча (количество примеров в батче), и xi​ — это входные значения для i-го примера. Тогда с**реднее значение** по батчу вычисляется по формуле в изображении 8.

Изображение 8. Формула для среднего значения и стандартного отклонения по батчу.



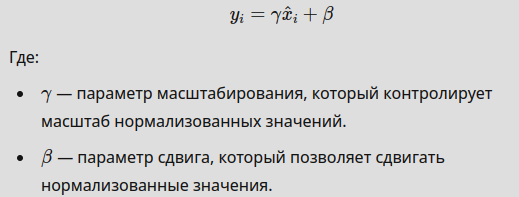
После вычисления среднего и стандартного отклонения данные нормализуются как на изображении 9.

Изображение 9. Формула для нормализации.



Для повышения гибкости нейронная сеть вводит два обучаемых параметра: **масштаб (gamma)** и **сдвиг (beta)**. После нормализации данные масштабируются и сдвигаются с использованием этих параметров, как показано на изображении 10.

Изображение 10. Масштабирование и сдвиг.



DropOut — слой, который отвечает за сброс некоторых случайных из нейронов. Это используется для того, что бы не возникало переобучения сети, при котором вся работа зависит только от малого множества. Это можно представить в виде множеств. Допустим, у нас есть некоторое множество всех нейронов слоя, тогда dropout выделит некоторое небольшое подмножество и обнулит его. Пример приведен на изображении 8.

Изображение 11. Пример DropOut на множестве.

Тут картинка

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 12.

Изображение 12. Архитектура основной сети.

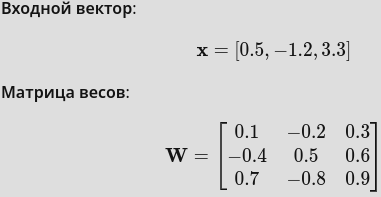
Тут картинка

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. Однако, поскольку в нашем случае есть такой слой, как BatchNormalization, то смещения можно убрать, поскольку в этом слое они уже реализованы. Тогда итоговая формула будет иметь вид: OutPut = x \* W. 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. Изобразим данные тензоры со случайными значениями (изображение 13)

Изображение 13. Тензоры со случайными значениями.



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 доманожается 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: out = ((s — size) / 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: size = (10 — 3) / 1 + 1 = 8, получается, что итоговый размер (8, 8, 32). Для основных вычислений используется формула: Ti = Σ(ri\*ci), где Ti — это элемент итоговой подматрицы, ri — элемент по строке в подматрице, ci — элемент по колоне в фильтре. Пример того, как происходят вычисления приведен на изображении 14.

Изображение 14. Пример вычисления в CL.

Тут картинка

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 15.

Изображение 15. Пример работы MaxPool слоя.

Тут картинка

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: region = (size — pool) / stride + 1, где region — размерность итогового региона, size — входной размер, pool — некоторый параметр (высота или ширина), stride — шаг, на который будет смещаться регион.

Двумя другими слоями будут BatchNormalization и dropout. По поводу них все было сказано и разобрано ранее.

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 нейронов (входных значений) 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 16.

Изображение 16. Функция softmax.

Тут картинка

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, почти так же, как и dropout слой. Данная функция описывается такой формулой: F(x) = max(0, x). Так же график такой функции приведен на изображении 17.

Изображение 17. Функция ReLU.

Тут картинка

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 плоносвязный 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 99.(9) значений. 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 понятие: градиент, градиентный спуск, функция потерь, обратное распространение ошибки, скорость обучения.

Стоит начать с функции потерь, поскольку в алгоритме обучения первая идет она. Функций 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. Формула такой функции выглядит как-то так: L = (x — y)2, где L — итоговая потеря, x — предсказания сети, y — реальные значения. Данная функция хороша тем, что она простая, удобная и понятная всем. Однако, данную функцию можно улучшить, просто используя принцип one hot, по которому вместо дробных чисел от 0.0 до 1.0 используется матрица, состоящая только из 0 и 1, где 1 — это реальное значение, а 0 — нет. Сама функция выглядит так: L = x — y, где L — потеря, x — предсказания сети, y — реальные значения. Однако x и y матрицы, поэтому и происходит матричная разность. Пример этой функции приведен на изображении 18.

Изображение 18. Пример работы функции потерь.

Тут картинка

После того, как вычисленна 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, градиенты гаммы и беты в BatchNormalization слое и градиенты фильтров в сверточном слое.

Сами по себе, с математической точки зрения, градиенты высчитываются так: пусть f(x) = 7x2+3y, тогда ее градиентом grad(f(x))=(df/dx, df/dy) в некоторой точке M(x, y). Тогда для нашего примера, градиентом будет: grad(f(x)) = (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 заданной функции представлен на изображении 19.

Изображение 19. Пример градиентного спуска.

Тут картинка

Однако, в контексте машинного обучения используется немного иной подход к подсчету градиентов. В случае с нейронной сетью используется подход, при котором происходит матричное множение транспонированных входных данных, на градиент прошлого слоя. То есть формулу можно записать так: grad(W) = xT \* grad(O), где xT — транспонированная матрица с исходными значениями, grad(O) — некоторый градиент предыдущего слоя. То же самое происходит и с входными данными для того, что бы передать их следующему слою: grad(I) = grad(O) \* WT, где I — исходные данные, grad(O) — градиент предудщего слоя, WT — транспонированная матрица весов. Такие действия происходят в каждом из слоев сети для того, что бы уменьшать ошибку и корректировать веса связей между слоями.

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 20.

Изображение 20. Пример корректировки скорости обучения.

Тут картинка

Сама же по себе корректировка весов для обучения происходит путем умножения градиента весов на этот самый параметр скорости обучения, то есть итоговая формула имеет вид: W = r \* grad(W), где W — веса слоя, r — скорость обучения, grad(W) — градиенты весов.

Данное строение помогает сети корректировать связи между всеми слоями сети, используя итоговую потерю и двигаясь к началу сети корректируя веса.

**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 64х64 или 128х128. 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 21.

Изображение 21. Пример преобработки изображения.

Тут картинка

2.2. Подготовка и рассчет 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. Так же из улучшений сети использовались BatchMormalization и dropout слои, а так же one hot функция потерь.

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

Таблица 1. Размеры для каждого слоя.

Тут табличку со всеми значениями.

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 3300 секунд, 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 10.

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

Таблица 2. Параметры каждой из эпох.

Тут таблица

Все данные собраны, модель работает, теперь стоит построить графики и убедится в том, что модель действительно стремится к локальному минимуму, далее, на графиках (изображения 22, 23, 24, 25, 26) и модели (изображение 27) это все показано.

Изображение 22. График обучения сверточного слоя 1.

Тут картинка

Изображение 23. График обучения сверточного слоя 2.

Тут картинка

Изображение 24. График обучения сверточного слоя 3.

Тут картинка

Изображение 25. График обучения полносвязного слоя 1.

Тут картинка

Изображение 26. График обучения полносвязного слоя 2.

Тут картинка

Изображение 27. Градиентный спуск для полносвязных слоев.

Тут картинка

Как можно заметить из приведенных выше графических моделей обучения сети, ее ошибка действительно стремится к нулю в зависимости от изменения весов и фильтров сети так, как это было сказано в теоретической части работы.

Сами примеры работы такой сети, включая удачные и неудачные приведены на изображениях ниже (изображение 28, 29, 30).

Изображение 28. Правильное предсказание сети 1.

Тут картинка

Изображение 29. Неверное предсказание сети.

Тут картинка

Изображение 30. Правильное предсказание сети 2.

Тут картинка

# 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

import csv

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)

# Определяем количество данных для тренировочного набора (10%)

for\_train = len(batch\_of\_files) // 10 # Количество файлов, которые пойдут в тренировочные данные

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)) # Меняем размер изображения на 64x64

image\_arr = np.array(image) / 255.0 # Нормализуем пиксели (0-1)

# Добавляем изображение в соответствующий список (train или learn)

if idx < for\_train:

train\_data.append((image\_arr, label))

else:

learn\_data.append((image\_arr, label))

# Перемешиваем данные

random.shuffle(learn\_data)

random.shuffle(train\_data)

# Извлекаем изображения и метки из данных

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])

# Возвращаем данные в виде кортежей: изображения и метки для learn и train

return (learn\_images, learn\_labels), (train\_images, train\_labels)

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

def relu(x):

return np.maximum(0, x)

def softmax(x):

exps = np.exp(x - np.max(x, axis=1, keepdims=True))

return exps / np.sum(exps, axis=1, keepdims=True)

class FlattenLayer:

def \_\_init\_\_(self):

self.input\_shape = None

def forward(self, inputs):

self.input\_shape = inputs.shape

return inputs.reshape(inputs.shape[0], -1) # Преобразование в одномерный вектор

def backward(self, grad\_output):

return grad\_output.reshape(self.input\_shape)

class ReluLayer:

def \_\_init\_\_(self):

self.inputs = None

def forward(self, inputs):

self.inputs = inputs

return np.maximum(0, inputs) # ReLU функция

def backward(self, grad\_output):

grad\_input = grad\_output \* (self.inputs > 0) # Производная ReLU

return grad\_input

class DropoutLayer:

def \_\_init\_\_(self, dropout\_rate=0.5):

self.dropout\_rate = dropout\_rate

self.mask = None

def forward\_prop(self, input\_data, training=True):

if training:

self.mask = (np.random.rand(\*input\_data.shape) > self.dropout\_rate).astype(float)

return input\_data \* self.mask / (1.0 - self.dropout\_rate)

else:

return input\_data

def backward\_prop(self, d\_out):

return d\_out \* self.mask / (1.0 - self.dropout\_rate)

class BatchNormalizationLayer:

def \_\_init\_\_(self, channels, epsilon=1e-5, momentum=0.9):

self.channels = channels

self.epsilon = epsilon

self.momentum = momentum

self.gamma = np.ones((1, channels))

self.beta = np.zeros((1, channels))

self.running\_mean = np.zeros((1, channels))

self.running\_var = np.zeros((1, channels))

self.norm\_inputs = None

def forward(self, inputs, train=True):

if train:

batch\_mean = np.mean(inputs, axis=0, keepdims=True)

batch\_var = np.var(inputs, axis=0, keepdims=True)

self.running\_mean = self.momentum \* self.running\_mean + (1 - self.momentum) \* batch\_mean

self.running\_var = self.momentum \* self.running\_var + (1 - self.momentum) \* batch\_var

self.norm\_inputs = (inputs - batch\_mean) / np.sqrt(batch\_var + self.epsilon)

else:

self.norm\_inputs = (inputs - self.running\_mean) / np.sqrt(self.running\_var + self.epsilon)

return self.norm\_inputs

def backward(self, grad\_out):

batch\_size = grad\_out.shape[0]

grad\_norm = grad\_out \* self.gamma

grad\_var = np.sum(grad\_norm \* (self.norm\_inputs \* -0.5 \* np.power(self.running\_var + self.epsilon, -1.5)),

axis=0)

grad\_mean = np.sum(grad\_norm \* -1 / np.sqrt(self.running\_var + self.epsilon), axis=0) + grad\_var \* np.mean(

-2 \* self.norm\_inputs, axis=0)

grad\_input = grad\_norm / np.sqrt(

self.running\_var + self.epsilon) + grad\_var \* 2 \* self.norm\_inputs / batch\_size + grad\_mean / batch\_size

grad\_gamma = (grad\_out \* self.norm\_inputs).mean(axis=(0, 1, 2), keepdims=False)

grad\_beta = grad\_out.mean(axis=(0, 1, 2), keepdims=False)

self.gamma -= grad\_gamma

self.beta -= grad\_beta

return grad\_input

class FullConnectedLayer:

"""Полносвязный слой для нелинейного преобразования всех предыдущих входных данных"""

def \_\_init\_\_(self, input\_size, output\_size, 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.activation = activation

self.learning\_rate = learning\_rate

self.input\_data = None

self.d\_weights = None

self.d\_biases = None

self.d\_input = None

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)

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

return output

def calculate\_grads(self, d\_out, learning\_rate=0.01):

self.d\_weights = np.dot(self.input\_data.T, d\_out)

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.weights -= self.learning\_rate \* self.d\_weights

return self.d\_input

def save\_history(self, losses, history\_file):

weights\_history = []

losses\_history = []

weights\_history.append(self.weights.copy())

losses\_history.append(losses.copy())

np.savez(history\_file,

weights=weights\_history,

losses=losses\_history)

print(f"Сохранена информация о весах и потерях 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

pool\_height, 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.learning\_rate = learning\_rate

self.activation = activation

self.image = None

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])

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\_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\_filters /= count

d\_input /= count

self.filters -= self.learning\_rate \* d\_filters

return d\_input

def save\_history(self, history\_file):

filters\_history = []

filters\_history.append(self.filters.copy())

np.savez(history\_file,

filters=filters\_history)

print(f"Сохранена информация о фильтрах в файл: {history\_file}")

class NeuralNetwork:

def \_\_init\_\_(self, input\_shape=(14, 14, 128), learning\_rate=0.01, num\_classes=5):

self.conv1 = ConvolutionLayer(filter\_size=3, num\_filters=32, stride=1,

learning\_rate=learning\_rate)

self.bn1 = BatchNormalizationLayer(32)

self.relu1 = ReluLayer()

self.mpl1 = MaxPoolLayer(pool\_size=(2, 2), stride=2)

self.conv2 = ConvolutionLayer(filter\_size=3, num\_filters=64, num\_channels=32, stride=1,

learning\_rate=learning\_rate)

self.bn2 = BatchNormalizationLayer(64)

self.relu2 = ReluLayer()

self.mpl2 = MaxPoolLayer(pool\_size=(2, 2), stride=2)

self.conv3 = ConvolutionLayer(filter\_size=3, num\_filters=128, num\_channels=64, stride=1,

learning\_rate=learning\_rate)

self.bn3 = BatchNormalizationLayer(128)

self.relu3 = ReluLayer()

self.mpl3 = MaxPoolLayer(pool\_size=(2, 2), stride=2)

self.dropout = DropoutLayer(dropout\_rate=0.5)

self.flatten = FlattenLayer()

self.fcl1 = FullConnectedLayer(input\_size=128 \* input\_shape[0] \* input\_shape[1], output\_size=256,

activation=relu, learning\_rate=learning\_rate)

self.fcl2 = FullConnectedLayer(input\_size=256, output\_size=num\_classes, activation=softmax,

learning\_rate=learning\_rate)

def forward(self, x):

x = self.conv1.forward(x)

x = self.bn1.forward(x)

x = self.relu1.forward(x)

x = self.mpl1.forward(x)

x = self.conv2.forward(x)

x = self.bn2.forward(x)

x = self.relu2.forward(x)

x = self.mpl2.forward(x)

x = self.conv3.forward(x)

x = self.bn3.forward(x)

x = self.relu3.forward(x)

x = self.mpl3.forward(x)

x = self.dropout.forward\_prop(x)

x = self.flatten.forward(x)

x = self.fcl1.forward(x)

x = self.fcl2.forward(x)

return x

def backward\_prop(self, d):

grads = {}

d = self.fcl2.calculate\_grads(d)

grads["fcl2"] = d

d = self.fcl1.calculate\_grads(d)

grads["fcl1"] = d

d = self.flatten.backward(d)

d = self.dropout.backward\_prop(d)

d = self.mpl3.backward\_prop(d)

d = self.relu3.backward(d)

d = self.bn3.backward(d)

d = self.conv3.backward\_prop(d)

grads["conv3"] = d

d = self.mpl2.backward\_prop(d)

d = self.relu2.backward(d)

d = self.bn2.backward(d)

d = self.conv2.backward\_prop(d)

grads["conv2"] = d

d = self.mpl1.backward\_prop(d)

d = self.relu1.backward(d)

d = self.bn1.backward(d)

d = self.conv1.backward\_prop(d)

grads["conv1"] = d

return d, grads

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 = []

query = input("Загрузить уже готовые веса для модели? (y/n) - ")

if query == "y":

self.load\_model(save)

with open("Accuracy.txt", "w") as f:

for epoch in range(1, 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]

# Разделение данных на мини-батчи

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]

# Прямое распространение (forward pass)

predictions = self.forward(image\_batch)

# Преобразуем метки в one-hot формат для кросс-энтропии

one\_hot\_labels = np.zeros\_like(predictions)

one\_hot\_labels[np.arange(label\_batch.size), label\_batch] = 1

# Вычисляем функцию ошибки (Cross-Entropy Loss)

loss = -np.sum(one\_hot\_labels \* np.log(predictions + 1e-7)) / batch\_size

total\_loss += loss

loss\_b.append(loss)

# Оценка корректных предсказаний

now\_correct\_predictions = np.sum(np.argmax(predictions, axis=1) == label\_batch)

correct\_predictions += now\_correct\_predictions

# Градиент ошибки

grad\_loss = predictions - one\_hot\_labels # Градиент CrossEntropyLoss

self.backward\_prop(grad\_loss) # Запускаем обратное распространение

# Сохранение весов после обратного распространения

batch\_weights.append(self.fcl1.get\_weights())

time\_e\_b = time.time()

# Сохраняем результаты для текущего батча

weights\_b.append(batch\_weights)

losses\_b.append(loss\_b)

time\_e = time.time()

# Средняя ошибка и точность за эпоху

average\_loss = total\_loss / (num\_samples // batch\_size)

accuracy = correct\_predictions / num\_samples

# Сохранение весов и потерь для текущей эпохи

epoch\_weights.append(self.fcl1.get\_weights())

weights\_e.append(epoch\_weights)

epoch\_losses.append(average\_loss)

losses\_e.append(epoch\_losses)

self.save\_model(save)

self.fcl1.save\_history(average\_loss, f"saves/history\_fcl\_{epoch + 1}.npz")

self.fcl2.save\_history(average\_loss, f"saves/history\_fcl2\_{epoch + 1}.npz")

self.conv1.save\_history(f"saves/history\_conv1\_{epoch + 1}.npz")

self.conv2.save\_history(f"saves/history\_conv2\_{epoch + 1}.npz")

self.conv3.save\_history(f"saves/history\_conv3\_{epoch + 1}.npz")

# Вывод результатов текущей эпохи

print(f"Epoch {epoch + 1}/{epochs} - Loss: {average\_loss:.4f}, Accuracy: {accuracy:.4%}\n"

f"Time: {time\_e - time\_s} sec.")

f.write(f"Epoch {epoch + 1}/{epochs} - Loss: {average\_loss:.4f}, Accuracy: {accuracy:.4%}\n"

f"Time: {time\_e - time\_s} sec.")

f.flush()

# Сохранение в файлы

with open("Batch\_Losses.txt", "w") as f\_b\_loss:

f\_b\_loss.write(str(losses\_b))

with open("Epoch\_Losses.txt", "w") as f\_e\_loss:

f\_e\_loss.write(str(losses\_e))

with open("Batch\_Weights.txt", "w") as f\_b\_weights:

f\_b\_weights.write(str(weights\_b))

with open("Epoch\_Weights.txt", "w") as f\_e\_weights:

f\_e\_weights.write(str(weights\_e))

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,

"conv2\_filters": self.conv2.filters,

"conv3\_filters": self.conv3.filters,

"bn1\_running\_mean": self.bn1.running\_mean,

"bn1\_running\_var": self.bn1.running\_var,

"bn1\_epsilon": self.bn1.epsilon,

"bn2\_running\_mean": self.bn2.running\_mean,

"bn2\_running\_var": self.bn2.running\_var,

"bn2\_epsilon": self.bn2.epsilon,

"bn3\_running\_mean": self.bn3.running\_mean,

"bn3\_running\_var": self.bn3.running\_var,

"bn3\_epsilon": self.bn3.epsilon,

"mpl1\_pool\_size": self.mpl1.pool\_size,

"mpl1\_stride": self.mpl1.stride,

"mpl2\_pool\_size": self.mpl2.pool\_size,

"mpl2\_stride": self.mpl2.stride,

"mpl3\_pool\_size": self.mpl3.pool\_size,

"mpl3\_stride": self.mpl3.stride,

"dropout": self.dropout.dropout\_rate,

"fcl\_weights": self.fcl1.weights,

"fcl2\_weights": self.fcl2.weights,

}

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 + ".npz", allow\_pickle=True)

self.conv1.filters = model\_data["conv1\_filters"]

self.conv1.activation = relu

self.conv2.filters = model\_data["conv2\_filters"]

self.conv2.activation = relu

self.conv3.filters = model\_data["conv3\_filters"]

self.conv3.activation = relu

self.bn1.running\_var = model\_data["bn1\_running\_var"]

self.bn1.running\_mean = model\_data["bn1\_running\_mean"]

self.bn1.epsilon = model\_data["bn1\_epsilon"]

self.bn2.running\_var = model\_data["bn2\_running\_var"]

self.bn2.running\_mean = model\_data["bn2\_running\_mean"]

self.bn2.epsilon = model\_data["bn2\_epsilon"]

self.bn3.running\_var = model\_data["bn3\_running\_var"]

self.bn3.running\_mean = model\_data["bn3\_running\_mean"]

self.bn3.epsilon = model\_data["bn3\_epsilon"]

self.mpl1.pool\_size = model\_data["mpl1\_pool\_size"]

self.mpl1.stride = model\_data["mpl1\_stride"]

self.mpl2.pool\_size = model\_data["mpl2\_pool\_size"]

self.mpl2.stride = model\_data["mpl2\_stride"]

self.mpl3.pool\_size = model\_data["mpl3\_pool\_size"]

self.mpl3.stride = model\_data["mpl3\_stride"]

self.dropout.dropout\_rate = model\_data["dropout"]

self.fcl1.weights = model\_data["fcl\_weights"]

self.fcl1.activation = relu

self.fcl2.weights = model\_data["fcl2\_weights"]

self.fcl2.activation = relu

print(f"Модель загружена из файла: {file\_path}")

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 = "saved\_model"

gray\_image = learn\_images[0]

plt.imshow(gray\_image, cmap='gray')

plt.title(f"Label: {learn\_labels[0]}")

plt.axis('off')

plt.show()

model = NeuralNetwork()

model.train(learn\_images, learn\_labels, real\_model)

test\_accuracy = test\_model(model, train\_images, train\_labels)

print(f"Final test accuracy: {test\_accuracy:.4f}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# epochs: 0

# Заключение

В ходе выполнения курсовой работы были рассмотренны и изучены нейронные сети, их обучение и принцип работы. Были выявлены способы улучшения работоспособности сети, рассмотренны конкретные примеры задач, которые решает та или иная сеть. Так же в данной работе была реализована сеть, которая распознает один из 5 классов цветов. Помимо всего прочего, были построены визуальные модели обучения сети, в особенности градиентный спуск.

# Ссылка на репозиторий GitHub

[https://github.com/ZamniProg/n](https://github.com/ZamniProg/aisd_lab_2)euro