Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №6**

**Дисциплина: Алгоритмы цифровой обработки мультимедиа**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_К.В.Стасюк

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.А. Крамаренко

**Лабораторная работа №6**

**Введение в нейронные сети.**

В рамках данной лабораторной работы будет рассматриваться решение

классификации изображений. Классификация - это процесс разделения объектов или данных на категории или классы на основе их характеристик или свойств. Она является одной из фундаментальных задач в машинном обучении и обработке данных. В задаче классификации алгоритм или модель обучается определять, к какому классу или категории принадлежит каждый входной объект. Нейронные сети широко используются для задач классификации из-за их способности извлекать сложные признаки и паттерны из данных. Для обучения нейронных сетей для задач классификации используются различные архитектуры, такие как сверточные нейронные сети (CNN) для обработки изображений, рекуррентные нейронные сети (RNN) для обработки последовательных данных (таких как текст и звук), и комбинации этих архитектур.

**Задание 1.** Построить многослойный персептрон средствами библиотеки Keras языка Python (или соответствующей библиотеки на другом языке), позволяющий распознавать цифры. Для обучения и тестирования использовать базу MNIST http://yann.lecun.com/exdb/mnist/.Возможно использовать любую функции активации, любую структуру сети и любой алгоритм обучения.

Многослойный персептрон (MLP) - это вид искусственной нейронной сети (ИНС) с множеством слоев нейронов (также известных как скрытые слои) между входным и выходным слоями. MLP - это одна из основных архитектур глубокого обучения, и она обычно используется для задач классификации и регрессии.MLP состоит как минимум из трех слоев: входного слоя, одного или нескольких скрытых слоев и выходного слоя. Слои нейронов в скрытых слоях могут иметь разное количество нейронов.Все нейроны в одном слое связаны с каждым нейроном в следующем слое. Это означает, что каждый нейрон в скрытом слое принимает входные сигналы от всех нейронов в предыдущем слое.Каждый нейрон в MLP имеет функцию активации, которая вводит нелинейность в модель и позволяет MLP моделировать сложные нелинейные зависимости в данных. Популярные функции активации включают сигмоиду, гиперболический тангенс (tanh) и ReLU (Rectified Linear Unit). MLP обучается с использованием алгоритмов обратного распространения ошибки (backpropagation), которые настраивают веса нейронов с целью минимизации функции потерь. Веса обновляются на основе градиента функции потерь, вычисленного для каждого нейрона. MLP может применяться к разнообразным задачам, включая классификацию, регрессию и другие. В задачах классификации, выходной слой обычно использует функцию активации Softmax, которая преобразует выходы сети в вероятности для различных классов. Если MLP имеет несколько скрытых слоев, его иногда называют "глубоким" многослойным персептроном. Глубокие многослойные персептроны стали основой для архитектур глубокого обучения, таких как сверточные нейронные сети (CNN) и рекуррентные нейронные сети (RNN).

Первым делом я загружаю данные MNIST, которые представляют из себя изображения и соответствующие им метки(цифры) для обучения и тестирования модели.

После чего необходимо провести предобработку тестовых и тренировочных данных.

Изначально изображения в наборе данных MNIST представлены в трехмерной форме (60000, 28, 28), где 60000 - количество изображений, 28 - высота изображения и 28 - ширина изображения. Я добавляю четвертое измерение к данным, чтобы сделать их совместимыми с ожидаемым входным форматом для нейронной сети: (60000, 28, 28, 1). Последняя размерность (1) обозначает количество цветовых каналов, в данном случае, у нас один канал, потому что изображения черно-белые.

После решейпинга изображений их значения пикселей нормализуются. Обычно пиксели изображений имеют значения от 0 до 255 (для каждого канала цвета). Нормализация заключается в преобразовании этих значений в диапазон от 0 до 1. Это важно для обучения нейронных сетей, так как она помогает стабилизировать обучение и ускорить сходимость алгоритма оптимизации.

В задачах классификации, особенно при использовании нейронных сетей, метки классов обычно представляются в формате one-hot encoding. Это означает, что каждая метка преобразуется в вектор нулей и одной единицы, где единица находится в позиции, соответствующей классу объекта. Функция **to\_categorical** из библиотеки Keras преобразует цифровые метки классов в формат one-hot encoding. Это необходимо для правильной интерпретации меток классов нейронной сетью во время обучения.

Далее можно приступить к созданию модели. Сначала создаем экземпляр модели нейронной сети, и она инициализируется как последовательная модель (**Sequential**). Данные будут передаваться от входного слоя к выходному слою последовательно. Используя эту модель, можно легко добавлять слои и удалять их, если это необходимо. Это делает ее идеальной для начинающих и для простых моделей. Однако, Sequential модель ограничена в том смысле, что она подходит только для создания линейных моделей, в которых данные проходят через слои последовательно без разветвлений или объединений.

Добавим слои в нашу модель. Первый слой модели - **"Flatten"** используется для преобразования входных данных (изображений) из формы (28, 28, 1) в одномерный вектор (плоский вектор). В данном случае, входные изображения имеют размерность 28x28x1 (где 1 - количество цветовых каналов), и этот слой выполняет решейпинг, чтобы преобразовать их в вектор длиной 28\*28=784 элемента.

После слоя Flatten добавляется полносвязный **(Dense)** слой с 128 нейронами (узлами) и функцией активации ReLU (Rectified Linear Unit). Этот слой выполняет линейные преобразования данных и активацию ReLU, чтобы внести нелинейность в модель. Полносвязный слой представляет собой слой, в котором каждый нейрон соединен с каждым нейроном предыдущего и следующего слоев (если они существуют) .В этом слое будет 128 внутренних переменных (весов), которые будут обучаться в процессе обучения сети. Для каждого нейрона в полносвязном слое есть веса, которые связаны с каждым элементом входного вектора (784 веса для каждого из 128 нейронов). Каждый вес определяет важность соответствующего пикселя изображения для этого нейрона(инициализируется случайными значениями). В процессе обучения нейронной сети веса в полносвязных слоях корректируются, чтобы минимизировать ошибку между предсказанными значениями и истинными метками в обучающем наборе данных. Это делается с помощью алгоритмов оптимизации, таких как градиентный спуск.Полносвязный слой выполняет линейную комбинацию весов и входных данных для каждого нейрона. Это можно представить как скалярное произведение вектора весов и входного вектора.К результатам линейных комбинаций добавляется функция активации ReLU (Rectified Linear Unit). Она определяется как f(x) = max(0, x), где x - входной сигнал, а f(x) - выходной сигнал.Функция ReLU применяется к выходу каждого нейрона и заменяет отрицательные значения на нули, оставляя положительные значения без изменений. Это способствует внесению нелинейности в модель, что позволяет сети обучаться сложным зависимостям в данных. Важно, что функция ReLU помогает решать проблему затухающего градиента, что облегчает обучение глубоких нейронных сетей.Таким образом, каждый нейрон в этом слое вычисляет нелинейную функцию от взвешенной суммы пикселей изображения, и каждый нейрон в слое работает независимо от других.В результате работы этого слоя получается набор признаков, которые выступают как абстрактные представления входного изображения. Эти признаки передаются дальше в следующие слои сети для выполнения более сложных операций.

Следующий слой - полносвязный слой с 64 нейронами и активацией ReLU. Этот слой уменьшает количество нейронов в сравнении с предыдущим слоем. Уменьшение числа нейронов в слое делает модель менее сложной и снижает количество параметров модели, что уменьшает вычислительную нагрузку при обучении и предсказании. А такжеуменьшение сложности модели может помочь в предотвращении переобучения.

Последний полносвязный слой- состоит из 10 нейронов, что соответствует 10 классам цифр (от 0 до 9). Для задачи классификации, в частности классификации рукописных цифр, обычно используется активация softmax, которая преобразует выходы сети в вероятности принадлежности каждого класса.Мы получаем 64 значения сумм после прошлого слоя, далее в каждый из 10 нейронов поступает 64 суммы и перемножается на значения весов, после процесса обучения мы получим значения сумм для каждого из 10 классов. После чего применяется функция softmax- линейная комбинация подвергается экспоненциальной функции.После применения экспоненты, каждая из 10 линейных комбинаций теперь представлена как экспонент, и для каждой из них вычисляется вероятность путем деления на сумму всех экспонент.Класс с наибольшей вероятностью считается предсказанным классом для данного входного изображения.

После чего можно обучать модель. **model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])** В этой строке указываются три основных параметра:

* + **optimizer='adam'**: Оптимизатор Adam используется для настройки весов сети в процессе обучения. Это один из популярных оптимизаторов для глубокого обучения. Он учитывает как скорость изменения параметров модели, так и прошлые изменения. Это помогает более стабильно обучать нейронные сети, устраняя шум и ускоряя процесс обучения. Adam также автоматически настраивает скорость обучения для каждого параметра, что делает его подходящим для разнообразных задач обучения нейронных сетей. Adam вычисляет скользящие средние градиентов(среднее значение градиентов с течением времени,это помогает определить, насколько быстро нейронная сеть обучается на основе изменений в градиентах) и квадратов градиентов(позволяет оценить изменчивость градиентов), а затем использует эти оценки для коррекции весов с учетом динамики градиентов.
  + **loss='categorical\_crossentropy'**: Это функция потерь, которая используется для оценки ошибки между предсказанными значениями и истинными метками. В данном случае, используется кросс-энтропия, которая обычно применяется в задачах классификации с несколькими классами.Задача модели-максимально уменьшить это значение.
  + **metrics=['accuracy']**: Это метрика, которая будет отслеживаться в процессе обучения для оценки производительности модели. Здесь используется точность (accuracy), которая измеряет долю правильно классифицированных примеров.

После обучения модели можно перейти к тестированию. **history = model.fit(train\_images, train\_labels, epochs=epochs, batch\_size=64, validation\_split=0.2)** используется для обучения нейронной сети и записи информации об истории обучения. **batch\_size** - определяет, сколько образцов данных будет использоваться для обновления параметров сети в каждой итерации. Мини-пакеты (batches) позволяют обучать модель более эффективно и управляемо.**validation\_split** - это параметр, который указывает, какую часть обучающих данных следует использовать для валидации (проверки производительности модели) во время обучения. В данном случае, 20% обучающих данных будут отложены для валидации.При выполнении **model.fit()**, нейронная сеть начинает проходить через обучающие данные в течение заданного числа эпох, итеративно обновляя веса и смещения, чтобы минимизировать функцию потерь. Вся информация об обучении, такая как значение функции потерь и точность на каждой эпохе, сохраняется в переменной **history**.

После чего я визуализировала работу модели. **model.predict(test\_images)** - это операция, в которой нейронная сеть принимает тестовые изображения и вычисляет предсказания для каждого изображения. Предсказания обычно представляют собой вероятности принадлежности объекта к разным классам (в данном случае, классам цифр).Далее выбираем максимальное значение вероятности, его индекс-наш ответ.Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание Рисунок 1 – Обучение модели на 5 эпохах.

Изображение выглядит как снимок экрана, диаграмма

Автоматически созданное описание

Рисунок 2 – Визуализация предсказаний модели.

Начало формы

**Задание 2** Исследовать нейронную сеть, построенную в задании 1 Подобрать количество эпох обучения, необходимое для выбранного вами предела корректной работы на тестовой базе. Сравнить несколько значений количества эпох по параметрам скорость обучения, скорость работы сети, процент корректной работы на тестовой базе.

В данном задании были сделаны те же самые действия, что и в 1 задаче, только для каждого значения эпох была создана и обучена своя модель.После чего был произведен сравнительный анализ.

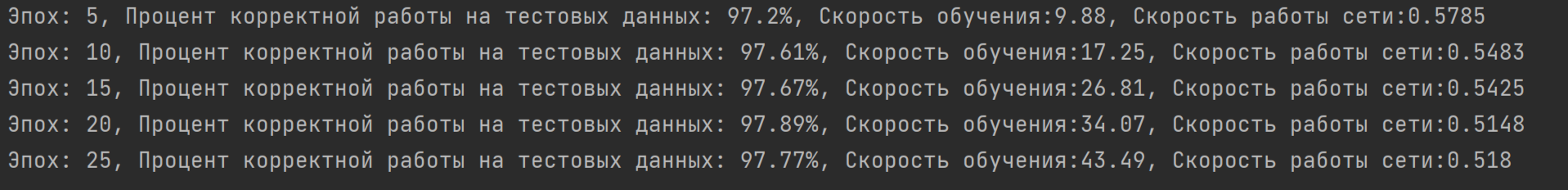


Рисунок 3 – Сравнительный анализ эпох.

Можно заметить, что наивысшая точность и скорость работы сети получена на 20 эпохе. Для удобства обученные модели для каждого значения эпох были сохранены и позже загружен.

**Задание 3** Построить сверточную нейронную сеть,решающую ту же задачу классификации. Рассмотреть несколько архитектур,подобрать оптимальные параметры. Сравнить результаты с предыдущей архитектурой.

Сверточные нейронные сети (CNN) - это класс нейронных сетей, специально разработанный для анализа изображений и видео. Они используют слои для извлечения признаков, такие как края и текстуры, и позволяют сети автоматически обучаться распознавать объекты на изображениях. CNN успешно применяются в компьютерном зрении, включая задачи классификации и обнаружения объектов. Их архитектуры включают сверточные слои, слои подвыборки и полносвязные слои, а также функции активации. CNN могут быть глубокими и способны извлекать сложные признаки из данных, делая их мощными инструментами в задачах анализа изображений.

В самом начале я также загрузила данные MNIST и сделала предобработку.После чего я выбрала 4 разных значения параметров для сравнительного анализа и выбора лучших значений. Были выбраны параметры:

1)Скорость обучения- это один из гиперпараметров, который определяет, насколько сильно модель будет корректировать веса своих параметров в процессе обучения. Она определяет размер шага, с которым модель движется в пространстве параметров при каждой итерации обучения.

2)Количество эпох

3)Размер пакета. Меньший размер пакета обычно означает более частые обновления весов модели, что может привести к более быстрому обучению. Однако слишком маленький размер пакета может вызвать шум в градиентах и нестабильность в обучении.Больший размер пакета требует больше памяти для хранения промежуточных данных во время обучения.

* + 4)Оптимизатор. Я рассматривала 2 оптимизатора.**Adam (Адаптивный метод оценки моментов.**Adam поддерживает адаптивную скорость обучения для каждого параметра, что означает, что скорость обучения адаптируется во время обучения. Это помогает модели сходиться быстрее и надежнее.Также он использует скользящие средние градиентов параметров, что обеспечивает сглаживание. Это помогает уменьшить шум и нестабильность в обновлениях параметров.**SGD (Стохастический градиентный спуск).**SGD - это фундаментальный алгоритм оптимизации, на котором основано множество других оптимизаторов.Он обновляет параметры модели в направлении отрицательного градиента функции потерь относительно этих параметров."Стохастический" в его названии означает, что он случайным образом выбирает подмножество (мини-пакет) тренировочных примеров для вычисления каждого обновления градиента, что вводит некоторый шум и помогает оптимизатору выйти из локальных минимумов.В то время как SGD прост и часто работает довольно хорошо, он может потребовать более ручной настройки скорости обучения и может сходиться медленнее по сравнению с адаптивными оптимизаторами, такими как Adam.

Также было рассмотрено 3 вида архитектур:

1. Разберем каждый слой и его назначение.

* **Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1))**: Этот слой является сверточным слоем. Он имеет 32 фильтра которые представляют собой небольшие матрицы (ядра свертки) размером 3x3 пикселя. Каждый из 32 фильтров будет скользить по входному изображению для выделения различных признаков. Для каждого из элементов матрицы имеем веса, которые умножаются на соответствующий пиксель и суммируются, тем самым мы получаем признак, а при прохождении всего изображения-карту признаков.Функция активации ReLU (Rectified Linear Unit) применяется к каждому элементу результирующего тензора.
* **MaxPooling2D((2, 2))**: После сверточного слоя идет слой подвыборки. Для каждой карты признаков, окно размером **(2, 2)** скользит по ней.В каждом окне находим максимальное значение.Эти максимальные значения формируют новую карту признаков с уменьшенными размерами. Это помогает уменьшить вычислительную сложность и уменьшить количество параметров модели.
* **Flatten()**: Этот слой выполняет преобразование данных из двумерного формата в одномерный вектор. Это необходимо для последующих полносвязных слоев.
* **Dense(64, activation='relu')**: Это полносвязный слой с 64 нейронами и функцией активации ReLU.
* **Dense(10, activation='softmax')**: Последний полносвязный слой, состоящий из 10 нейронов, соответствующих 10 классам цифр (от 0 до 9). Функция активации Softmax используется для преобразования выходных значений в вероятности принадлежности к каждому классу.

Эта архитектура CNN представляет собой стандартную конфигурацию для решения задачи классификации изображений, и она настроена на работу с набором данных MNIST, который содержит черно-белые изображения цифр.

2)Отличия этой архитектуры от 1 в том, Архитектура 2 имеет больше фильтров (64 против 32) в своем первом сверточном слое, что может помочь изучить более высокоуровневые признаки изображения.Также первый полносвязный слой в Архитектуре 2 содержит больше нейронов (128 против 64), что может увеличить ее способность к изучению сложных зависимостей. Более глубокие и "широкие" сети (с большим количеством фильтров и нейронов) могут иметь больший потенциал для изучения сложных паттернов в данных, но они также требуют больше вычислительных ресурсов и времени на обучение.

3)Архитектура 3 имеет дополнительный полносвязный слой с 64 нейронами. Это может увеличить ее способность к изучению сложных зависимостей.

Далее происходит обучение каждой из архитектур для каждого набора параметров. Аналогично первому заданию модель создается, обучается и тестируется.

Изображение выглядит как снимок экрана, диаграмма

Автоматически созданное описание Рисунок 4 – Визуализация работы CNN.

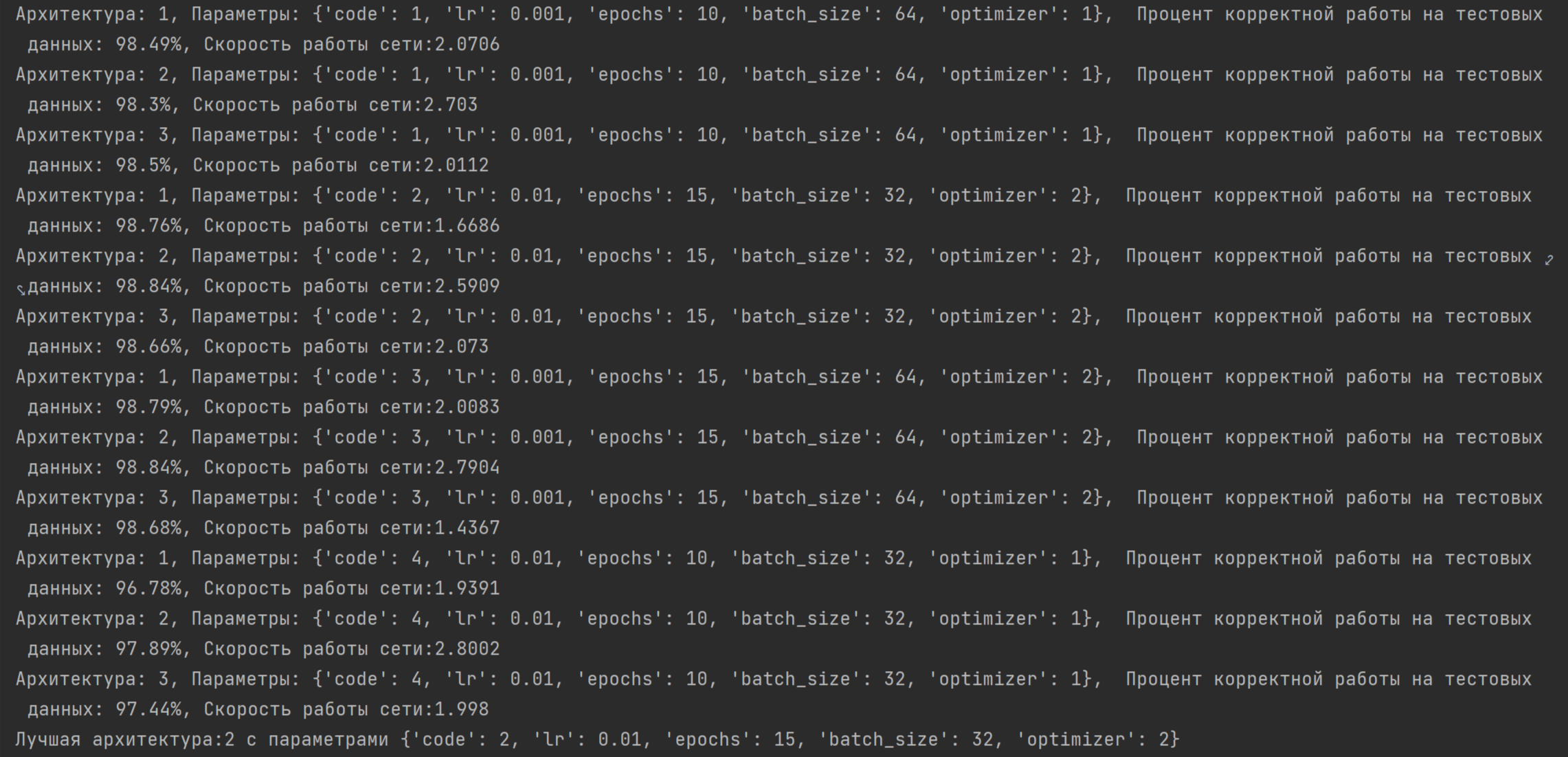


Рисунок 5 – Сравнительный анализ CNN.

Можно заметить, что лучший результат по точности и времени работы показала Архитектура 2 с параметрами скорость обучения=0.01,эпохи=15,размер пакета=32,оптимизатор-SGD, но имея достаточно большое значение времени обучения.

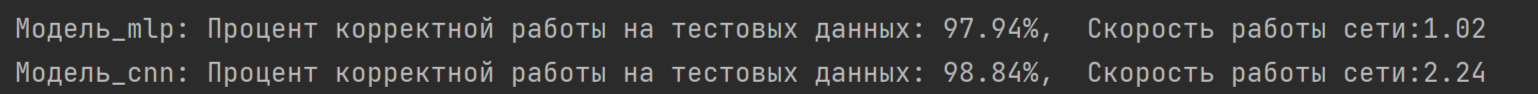


Рисунок 6 – Сравнительный анализ CNN и MLP.

Можно заметить, что CNN показывает лучшие результаты в точности, а MLP в скорости.

**Листинг программы**

**Python**

mlp\_for\_mnist\_digit.py

import numpy as np  
from keras.datasets import mnist  
from keras.utils import to\_categorical  
import matplotlib.pyplot as plt  
import time  
from keras.models import load\_model  
from keras.models import Sequential  
from keras.layers import Dense, Flatten  
  
  
# Загрузим данные MNIST для обучения модели(train-изображения и метки для обучения,test-изображения и метки для тестирования)  
(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()  
  
# Предобработка данных  
train\_images = train\_images.reshape((60000, 28, 28, 1)) # Добавляем размерность (1)-количество цветовых каналов(изображения черно-белые)  
test\_images = test\_images.reshape((10000, 28, 28, 1))  
  
  
# Нормализуем данные и переводим метки классов в формат one-hot encoding  
train\_images = train\_images.astype('float32') / 255 # Нормализуем значения пикселей до диапазона [0, 1]  
test\_images = test\_images.astype('float32') / 255  
train\_labels = to\_categorical(train\_labels) # метка преобразуется в вектор нулей и одной единицы, где единица находится в позиции, соответствующей классу объекта  
test\_labels = to\_categorical(test\_labels)  
  
# Используем разные значения количества эпох  
epochs\_list = [5,10,15,20,25]  
  
results = []  
# для процесса обучения модели  
for epochs in epochs\_list:  
 model = Sequential() #Последовательная модель  
 #1 слой модели, используется для преобразования входных данных из формы (28, 28, 1) в одномерный вектор 28\*28=784 элемента.  
 model.add(Flatten(input\_shape=(28, 28, 1)))  
 #полносвязный (Dense) слой с 128 нейронами и функцией активации ReLU,выполняет линейные преобразования данных и активацию ReLU, чтобы внести нелинейность в модель.  
 model.add(Dense(128, activation='relu'))  
 #уменьшает количество нейронов в сравнении с предыдущим слоем  
 model.add(Dense(64, activation='relu'))  
 #Последний полносвязный слой состоит из 10 нейронов, что соответствует 10 классам цифр,  
 # используется активация softmax, которая преобразует выходы сети в вероятности принадлежности каждого класса  
 model.add(Dense(10, activation='softmax'))  
  
 #компиляция модели  
 model.compile(optimizer='adam', #Оптимизатор Adam используется для настройки весов сети в процессе обучения  
 loss='categorical\_crossentropy', #функция потерь, которая используется для оценки ошибки между предсказанными значениями и истинными метками  
 metrics=['accuracy']) #точность-метрика, которая будет отслеживаться в процессе обучения для оценки производительности модели.  
  
 start\_time = time.time()  
 # Обучаем модель и сохраняем историю обучения  
 history = model.fit(train\_images,  
 train\_labels,  
 epochs=epochs,  
 batch\_size=64,  
 validation\_split=0.2)  
 end\_time = time.time()  
 training\_time = end\_time - start\_time  
  
 results.append((epochs, round(training\_time, 4)))  
  
 # Сохранение модели  
 # model.save(f'models/mlp\_model{epochs}.keras')  
  
# Выводим значения в прцессе обучения  
for epochs,training\_time in results:  
 print(f"Эпох: {epochs} "  
 f"Скорость обучения:{training\_time}")  
  
results=[]  
#вывод значений для сохраненных моделей  
for epochs in epochs\_list:  
 model = load\_model(f'models/mlp\_model{epochs}.keras')  
  
 # Оцениваем производительность на тестовых данных  
 \_,test\_accuracy = model.evaluate(test\_images, test\_labels)  
  
 start\_time = time.time() # Засекаем начальное время обучения  
 # Предсказания модели на тестовых данных  
 predictions = model.predict(test\_images)  
 end\_time = time.time() # Засекаем конечное время обучения  
 work\_time=end\_time-start\_time  
  
 results.append((epochs, round(test\_accuracy\*100,2),round(work\_time,4)))  
  
 # Выбираем случайное изображение для визуализации  
 index = np.random.randint(0, len(test\_images))  
  
 # Отображаем выбранное изображение  
 plt.imshow(test\_images[index].reshape(28, 28), cmap='gray')  
 plt.title(f'Предсказание модели: {np.argmax(predictions[index])}')  
 plt.show()  
  
for epochs, accuracy,work\_time in results:  
 print(f"Эпох: {epochs}, Процент корректной работы на тестовых данных: {accuracy}%, "  
 f" Скорость работы сети:{work\_time}")

CNN.py

from keras.datasets import mnist  
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense  
from keras.models import Sequential  
from keras.optimizers import Adam, SGD  
from keras.utils import to\_categorical  
import time  
import numpy as np  
import matplotlib.pyplot as plt  
from keras.models import load\_model  
  
# Подготовка данных  
(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()  
  
#Добавляем размерность (1)-количество цветовых каналов  
#Нормализуем значения пикселей до диапазона [0, 1]  
train\_images = train\_images.reshape((60000, 28, 28, 1)).astype('float32') / 255  
test\_images = test\_images.reshape((10000, 28, 28, 1)).astype('float32') / 255  
# метка преобразуется в вектор нулей и одной единицы,  
# где единица находится в позиции, соответствующей классу объекта  
train\_labels = to\_categorical(train\_labels)  
test\_labels = to\_categorical(test\_labels)  
  
# Список параметров для экспериментов  
params\_to\_try = [  
#скорость обучения,количество эпох,размер пакета  
 {'code': 1,'lr': 0.001, 'epochs': 10, 'batch\_size': 64,'optimizer': 1},  
 {'code': 2,'lr': 0.01, 'epochs': 15, 'batch\_size': 32,'optimizer': 2},  
 {'code': 3,'lr': 0.001, 'epochs': 15, 'batch\_size': 64,'optimizer': 2},  
 {'code': 4,'lr': 0.01, 'epochs': 10, 'batch\_size': 32,'optimizer': 1},  
]  
  
# Список архитектур для экспериментов  
architectures = [  
 # Архитектура 1  
 [  
 #сверточный слой(32 слоя,в каждом слое матрица 3x3 ходит по изображению-на выходе 32 матрицы признаков)  
 Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),  
 #матрица 2x2 ходит по каждой из 32 выходных матриц признаков и выбирает макисмальный=уменьшает)  
 MaxPooling2D((2, 2)),  
 #преобразует матрицу в вектор  
 Flatten(),  
 #находит 64 значения нелинейного скалярного произведения  
 Dense(64, activation='relu'),  
 #итоговые вероятности  
 Dense(10, activation='softmax')  
 ],  
 # Архитектура 2  
 [  
 Conv2D(64, (3, 3), activation='relu', input\_shape=(28, 28, 1)),  
 MaxPooling2D((2, 2)),  
 Flatten(),  
 Dense(128, activation='relu'),  
 Dense(10, activation='softmax')  
 ],  
 # Архитектура 3  
 [  
 Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),  
 MaxPooling2D((2, 2)),  
 Flatten(),  
 Dense(64, activation='relu'),  
 Dense(64, activation='relu'),  
 Dense(10, activation='softmax')  
 ],  
]  
  
# results = []  
# for params in params\_to\_try:  
# arch\_num=0  
# for architecture in architectures:  
# arch\_num += 1  
# #полносвязная модель  
# model = Sequential(architecture)  
# #выбираем оптимизатор  
# if(params['optimizer']==1):  
# optimizer = Adam(params['lr'])  
# else:  
# optimizer = SGD(learning\_rate=params['lr'])  
# #обучаем  
# model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])  
# #тренируем  
# start\_time = time.time()  
# model.fit(train\_images, train\_labels, epochs=params['epochs'], batch\_size=params['batch\_size'], validation\_split=0.2)  
# end\_time = time.time()  
# training\_time = end\_time - start\_time  
#  
# results.append((arch\_num,params, round(training\_time, 2)))  
#  
# # Сохранение модели  
# model.save(f'models/cnn\_model{arch\_num}{params["code"]}.keras')  
#  
# for ind, params,training\_time in results:  
# print(f"Архитектура: {ind}, Параметры: {params}, "  
# f"Скорость обучения:{training\_time}")  
  
  
#Для сохраненных моделей  
results = []  
best\_model = 0  
best\_params = None  
best\_accuracy = 0  
  
for params in params\_to\_try:  
 arch\_num=0  
 for architecture in architectures:  
 arch\_num+=1  
 #загрузка модели  
 model=load\_model(f'models/cnn\_model{arch\_num}{params["code"]}.keras')  
 #тренируем  
 \_,test\_accuracy = model.evaluate(test\_images, test\_labels)  
  
 start\_time = time.time() # Засекаем начальное время обучения  
 # Предсказания модели на тестовых данных  
 predictions = model.predict(test\_images)  
 end\_time = time.time() # Засекаем конечное время обучения  
 work\_time = end\_time - start\_time  
  
 # Выбираем случайное изображение для визуализации  
 index = np.random.randint(0, len(test\_images))  
  
 # Отображаем выбранное изображение  
 plt.imshow(test\_images[index].reshape(28, 28), cmap='gray')  
 plt.title(f'Предсказание модели: {np.argmax(predictions[index])}')  
 plt.show()  
  
 results.append((arch\_num,params, round(test\_accuracy \* 100, 2), round(work\_time, 4)))  
  
 if test\_accuracy > best\_accuracy:  
 best\_accuracy = test\_accuracy  
 best\_model = arch\_num  
 best\_params=params  
  
for ind, params,accuracy,work\_time in results:  
 print(f"Архитектура: {ind}, Параметры: {params}, "  
 f" Процент корректной работы на тестовых данных: {accuracy}%, "  
 f"Скорость работы сети:{work\_time}")  
  
print(f"Лучшая архитектура:{best\_model} с параметрами {best\_params}")

cnn\_vs\_mlp.py

from keras.models import load\_model  
from keras.utils import to\_categorical  
from keras.datasets import mnist  
import time  
  
# Подготовка данных  
(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()  
test\_images = test\_images.reshape((10000, 28, 28, 1))  
test\_images = test\_images.astype('float32') / 255  
test\_labels = to\_categorical(test\_labels)  
  
  
#загрузка моделей  
model\_mlp = load\_model(f'models/mlp\_model25.keras')  
model\_cnn = load\_model(f'models/cnn\_model22.keras')  
  
# Оцениваем производительность на тестовых данных  
\_,test\_accuracy\_mlp = model\_mlp.evaluate(test\_images, test\_labels)  
\_,test\_accuracy\_cnn = model\_cnn.evaluate(test\_images, test\_labels)  
  
start\_time = time.time() # Засекаем начальное время обучения  
# Предсказания модели на тестовых данных  
predictions\_mlp = model\_mlp.predict(test\_images)  
end\_time = time.time() # Засекаем конечное время обучения  
work\_time\_mlp=end\_time-start\_time  
  
start\_time = time.time() # Засекаем начальное время обучения  
# Предсказания модели на тестовых данных  
predictions\_cnn = model\_cnn.predict(test\_images)  
end\_time = time.time() # Засекаем конечное время обучения  
work\_time\_cnn=end\_time-start\_time  
  
print(f"Модель\_mlp: Процент корректной работы на тестовых данных: {round(test\_accuracy\_mlp\*100,2)}%, "  
 f" Скорость работы сети:{round(work\_time\_mlp,2)}")  
  
print(f"Модель\_cnn: Процент корректной работы на тестовых данных: {round(test\_accuracy\_cnn\*100,2)}%, "  
 f" Скорость работы сети:{round(work\_time\_cnn,2)}")