Министерство науки и высшего образования РФ

Федеральное государственное автономное образовательное учреждение высшего образования

«Казанский (Приволжский) Федеральный Университет»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА ТЕОРЕТИЧЕСКОЙ КИБЕРНЕТИКИ

Направление: 01.03.02. Прикладная математика и информатика

Профиль: Прикладная математика и информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Реализация нейронных сетей с применением технологии CUDA

Работа завершена:

Студент гр. 09-811 ИВМиИТ

«__» _____ 2022 г. _____ М. В. Царьков

Работа допущена к защите:

Научный руководитель,

ст. преподаватель КПМиИТ ИВМиИТ

«__» _____ 2022 г. _____ Д. Х. Гиниятова

Заведующий кафедрой:

д.ф.м.н., профессор

«__» _____ 2022 г. _____ Ф. М. Аблаев

Казань 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задачи распознавания образов за последние годы получили большую популярность и стали перспективными для дальнейшего исследования и поиска оптимальных решений. Распознать объект – значит обозначить, какой образ указан на конкретном изображении. Решение данной задачи сводится к решению задачи классификации при помощи некоторых алгоритмов, которые зависят от предметной области.

Актуальность исследования в области идентификация объектов на изображениях подтверждается тем, что использование систем распознавания является популярной практикой в медицинской и технической диагностиках, контроле качества на производстве, системах видеонаблюдения и многих других сферах деятельности человека. Цифровые изображения составляют большой объём данных в любой промышленной области. Возможность точно, быстро и качественно классифицировать объекты играет важную роль в работе компаний и производств. Множество подобных задач, требующих решения, стимулируют разработку многочисленных приложений в различных областях. Приведём несколько примеров.

Для медицинских исследований алгоритмы компьютерного зрения имеют огромное значение. Самое главное при лечении пациента – правильно поставить диагноз и сделать это в максимально короткий промежуток времени, чтобы дальнейшее лечение было сформировано наиболее качественно. Благодаря алгоритмам классификации изображений появилась возможность исследовать рентгенографические и томографические снимки на предмет различных отклонений. Чувствительность новых методов исследования в сочетании с методами визуализации позволяет проводить более точную диагностику, чем при использовании традиционных исследовательских методов [1].

В промышленности детектирование объектов помогает выполнять классические задачи производства продукции и проверки на соответствие

стандартам качества. Для производства деталей необходимого размера и уменьшения количества брака на производстве используются специальные системы автоматического контроля над процессами.

Во многих компаниях длительное время используют всевозможные цифровые устройства, в том числе сканеры, способные обрабатывать документы, написанные человеком. Возможность преобразовывать рукописный текст в цифровой набор символов при помощи алгоритмов классификации объектов экономит огромное количество времени по сравнению с ручным переносом текста. Одним из первых практических применений методов распознавания текста стало создание систем чтения ZIP-кодов почты США [2], сканирования банковских чеков [3].

Благодаря исследованиям учёных наука предложила множество алгоритмов, позволяющих распознавать образы. Особо популярными являются алгоритмы нейронных сетей, в основе которых лежит схема работы головного мозга. На основе биологического нейрона успешно создана математическая модель и использована при создании искусственной нейронной сети.

Применение эффективных методов и современных компьютерных технологий позволяет решать задачи распознавания образов за приемлемое время. Одной из перспективных технологий является платформа CUDA (Compute Unified Device Architecture) [4], которая позволяет программному обеспечению использовать графические процессоры от компании NVIDIA. Технология CUDA позволяет выполнять множество операций параллельно, что ускоряет определённые части алгоритма и позволяет добиться более быстрого выполнения программы.

Цель работы – выбор и разработка алгоритма классификации объектов на языке программирования python версии 3.10 с использованием технологии параллельного программирования CUDA.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Изучение существующих архитектур и алгоритмов нейронных сетей.
2. Выбор оптимальной архитектуры нейронной сети для задачи распознавания объектов на изображениях рукописных цифр. Поиск данных для обучения нейронной сети.
3. Исследование математических формул, необходимых для реализации нейронной сети.
4. Изучение технологии CUDA. Выбор инструментов и языка программирования для реализации выбранной архитектуры нейронной сети.
5. Сравнение скорости выполнения последовательных и параллельных алгоритмов.
6. Реализация нейронной сети с применением технологии CUDA. Обучение нейронной сети на выбранных данных.
7. Анализ результатов и скорости выполнения алгоритма.
8. Оптимизация времени выполнения кода с помощью современных инструментов профилирования.

Новизна исследования заключается в применении современных методов и технологий параллельного выполнения алгоритмов к решению задач классификации изображений.

Глава 1 Теоретические основы нейронных сетей

1.1 Биологический нейрон

В настоящее время существует множество архитектур нейронных сетей, но все они построены на принципе работы связей в головном мозге.

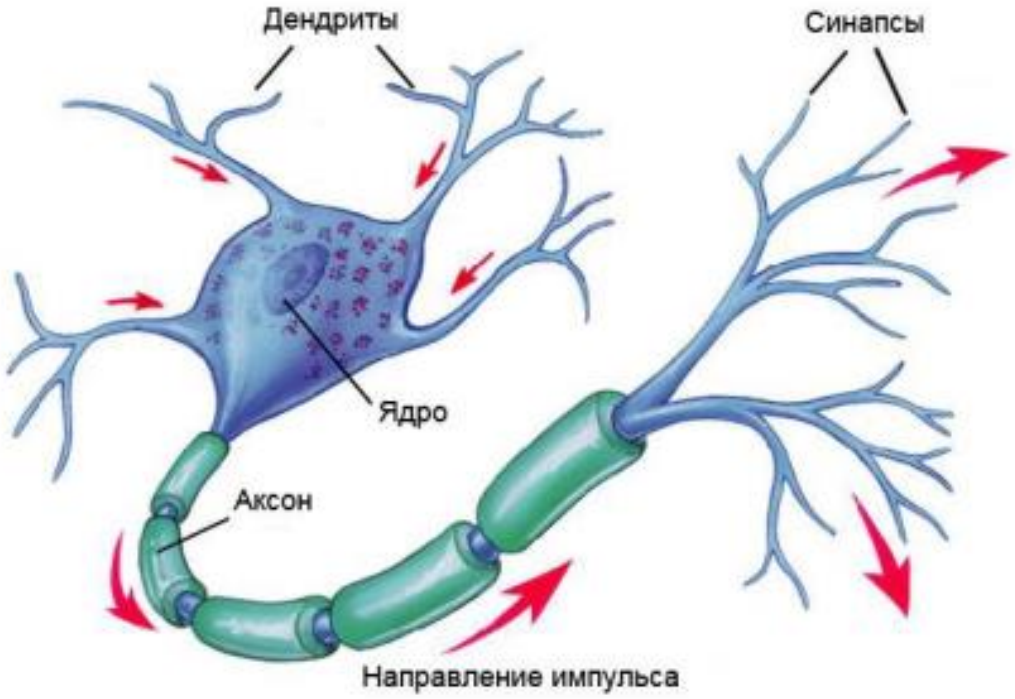

Рисунок 1.1 – Биологическая модель нейрона

Рассмотрим биологическую модель нейрона на рисунке 1.1. Нейрон – это специализированный тип клетки, осуществляющий передачу нервного импульса [5]. Данная модель содержит отростки различных типов, которые необходимы для различных процессов:

- Дендриты передают электрические импульсы в ядро. Их может быть несколько для одного нейрона.

- Аксон передаёт электрический импульс от ядра к следующим клеткам.

- Синапсы распределяют силу заряда после аксона и пропускают заряд дальше с различными электрическими импульсами.

Можно представить изменение заряда при прохождении от дендрита через ядро клетки как умножение исходного заряда на некоторое число из промежутка [0, 1]. Обозначим это число как вес нейрона. Заряды приходят на множество дендритов нейрона, умножаются на свой собственный вес, суммируются и передаются ядру нейрона. Далее нейрон возбуждает свой собственный импульс, который транслируется в аксон. Аксон в свою очередь изменяет выходящий заряд в зависимости от типа клетки и внутреннего строения. Следует заметить, что при развитии мозга человека веса для каждого входящего сигнала могут изменяться, что является основной идеей при обучении нейронных сетей.

1.2 Математическая модель нейрона

На основе рассмотренной биологической модели сформируем математическую модель нейрона.

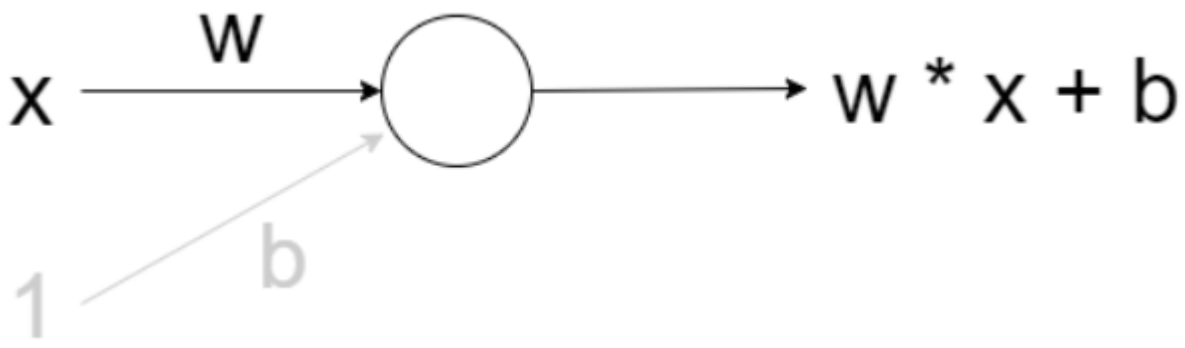

Рисунок 1.2 – простейшая математическая модель нейрона

Самым наивным и достаточным представлением нейрона можно считать модель на рисунке 1.2. Роль дендрита выполняет x и передаёт входящий в нейрон сигнал, умноженный на вес w. Веса w изменяются в процессе обучения и могут обнулиться, поэтому, чтобы значение нейрона не

было равно нулю, используют так называемое смещение b. Получим конечную формулу для одного нейрона:

$$x \cdot w + b \tag{1.1}$$

Рассмотрим следующую модель нейрона, которая содержит несколько входов, как в биологической модели, и функцию активации, заменяющую аксон.

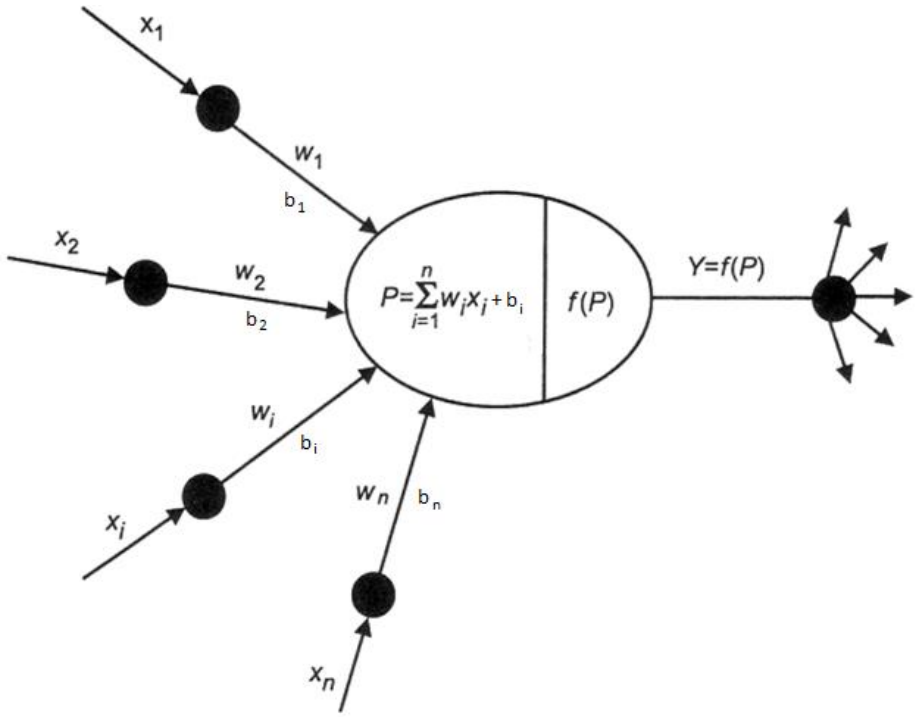

Рисунок 1.3 – сложная математическая модель нейрона с функцией активации

На рисунке 1.3 изображена модель с несколькими входными сигналами, обозначенными как $\vec{x} = (x_1, x_2, \dots, x_n)$, вектор весов $\vec{w} = (w_1, w_2, \dots, w_n)$ и вектор смещения $\vec{b} = (b_1, b_2, \dots, b_n)$. Внутри ядра нейрона находится взвешенная сумма входных сигналов P и передаётся в функцию активации $f(P)$.

$$Y_i = f(\sum_{i=1}^{n} w_i \cdot x_i + b_i) \tag{1.2}$$

Получаем выходной сигнал нейрона Y_i, который отправляется следующим нейронам. Заметим, что нейронная сеть обязательно должна иметь хотя бы одну нелинейную функция активации $f(P)$. Это следует из вывода, что если все функции активации будут линейными, нейронная сеть, состоящая из множества нейронов, будет вести себя как один большой нейрон, что не даёт никакой вариативности.

Согласно определению, линейная функция – это функция $f(x)$, ставящая в соответствие элементу x линейного пространства X действительное число y, называется линейной, если выполняются следующие условия [6]:

1) Для любых $x_1, x_2 \in X$ верно:

$$f(x_1 + x_2) = f(x_1) + f(x_2)$$

2) Для любого $x \in X$ и любого действительного числа $\lambda \in \mathbb{R}$ верно:

$$f(\lambda x) = \lambda f(x)$$

Поэтому, сумма линейных функций является линейной функцией. Следовательно, необходимо использовать нелинейные функции активации для нейрона.

Рассмотрим подробнее стандартные функции активации [7]:

1) Сигмоида.

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad (1.3)$$

Производная.

$$\sigma'(x) = 1 \cdot (1 - x) \quad (1.4)$$

Функция определена на отрезке $(0, 1)$. График функции представлен на рисунке 1.4.

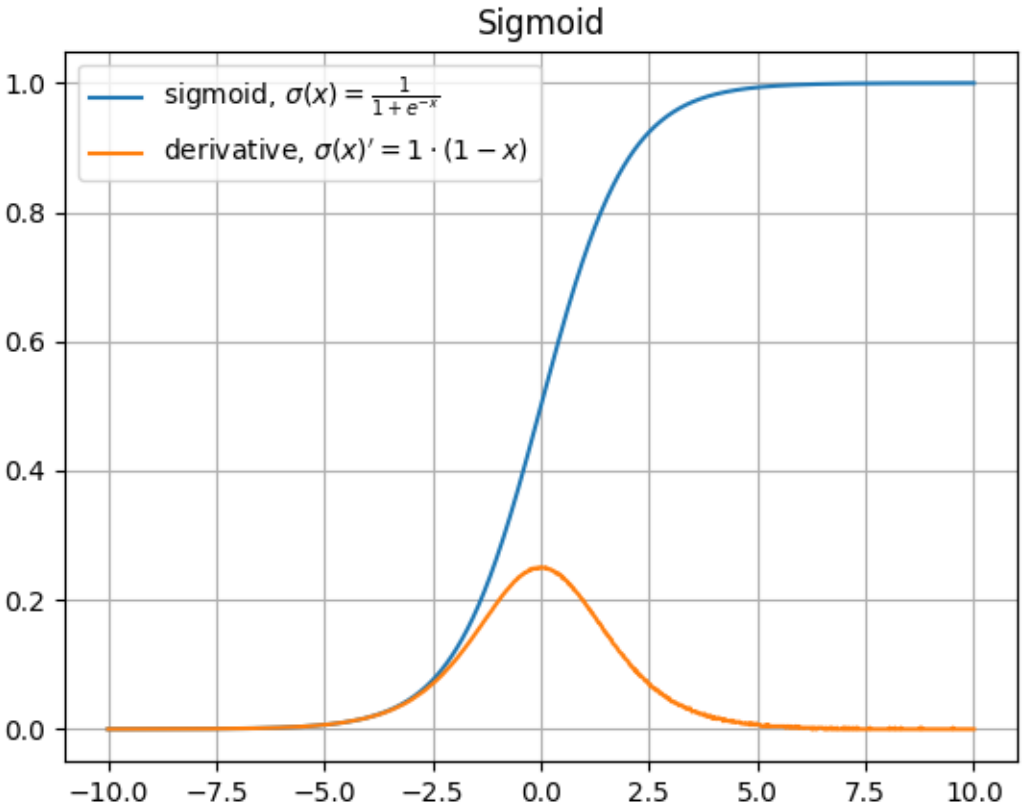

Рисунок 1.4 – график сигмоиды и её производной

Преимущества сигмоиды:

- Функция является нелинейной. Это позволяет использовать многослойные сети.
- Аналоговая активация за счёт того, что функция не бинарная.

Недостатки сигмоиды:

- Приближаясь к концам функции, выходные значения нейронов начинают меньше реагировать на входные значения. Это приводит к уменьшению значения градиента, что в дальнейшем не позволяет нейронной сети успешно продолжать обучение.

2) LeakyReLU.

$$\mathrm{F(x)} = \begin{cases} 0{,}01 \cdot x\,, & \text{если } x < 0 \\ x, & \text{если } 0 \le x \le 1 \\ 1 + 0{,}01 \cdot (x - 1), & \text{если } x > 1 \end{cases} \quad (1.5)$$

Производная.

$$F(x) = \begin{cases} 0{,}01, & \text{если } x < 0, x > 1 \\ 1, & \text{если } 0 \leq x \leq 1 \end{cases} \quad (1.6)$$

Диапазон значений функции $(-\infty, \infty)$. График функции представлен на рисунке 1.5.

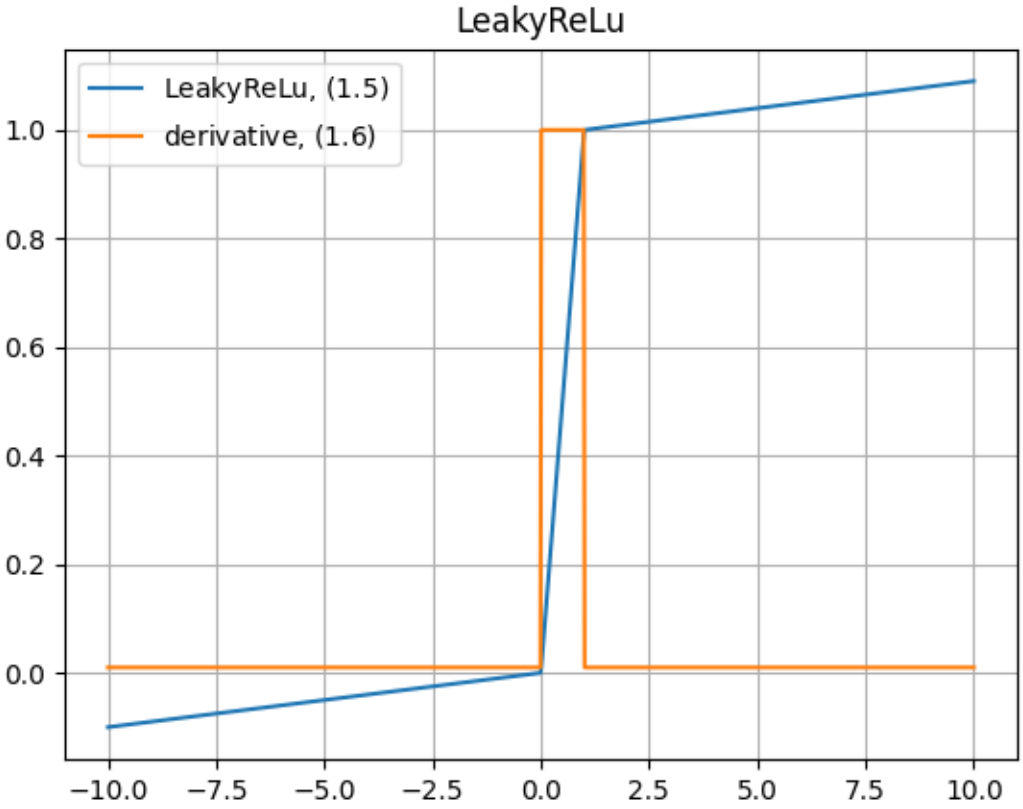

Рисунок 1.5 – график LeakyReLu и её производной

Преимущества LeakyReLu:

- В отличие от классической функции ReLu, модифицированная позволяет нейрону активироваться даже при отрицательном входе. Следовательно, производная не будет нулевой при отрицательных x.
- При отрицательных значениях создаёт наклонную на отрезках $(-\infty, 0)$ и $(1, +\infty)$ с угловым коэффициентом 0.01.

- Функция содержит простые операции, следовательно, скорость её вычисления довольно быстрая.

Недостатки modRelu:

- Угловой коэффициент является гиперпараметром, который необходимо настраивать.
- При решении некоторых классов задач практически нет различий с классической функцией ReLu.

Рассмотренная модель нейрона имеет некоторый уровень интеллекта, но этот уровень достаточно низкий. Модель представляет собой регрессионную модель для N количества независимых входных данных. Однако нейронная сеть состоит из совокупности нейронов, которая может быть представлена в виде слоя - персептрона.

1.3 Однослойный персептрон

Однослойный персептрон [8] представляет собой слой из K количества нейронов с таким же количеством выходов $\overline{Y}$. Количество входов $\overline{X}$ размерности N. Для такого персептрона веса записываются в матрицу $\overline{W}$ размерности (N, K), где K – количество нейронов в слое. Схема однослойного персептрона представлена на рисунке 1.6.

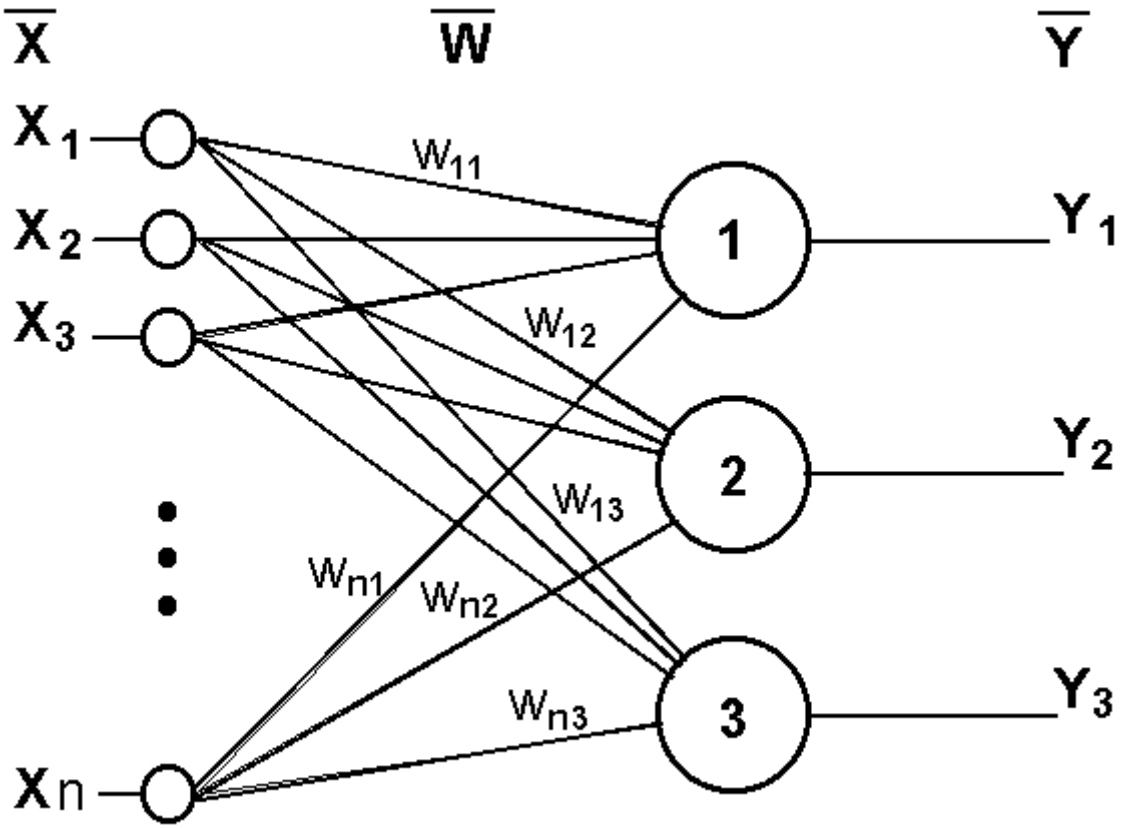

Рисунок 1.6 – однослойный персептрон. K=3.

Выходы нейрона высчитываются по следующим формулам:

$$P_i = \sum_{j=0}^{n} x_j \cdot w_{ji} + b_i \quad (1.7)$$

$$Y_i = f(P_i) \quad (1.8)$$

Аналогично одному нейрону считаем взвешенную сумму каждого нейрона P_i и передаём в функцию активации и получаем выходное значение для каждого нейрона Y_i.

1.4 Многослойный персептрон

Разберём следующую модель многослойного персептрона, состоящего из входного, двух скрытых и одного выходного слоя. Можно заметить, что выходной слой состоит из двух нейронов, следовательно, нейронная сеть данной архитектуры решает задачу бинарной классификации. Для задачи

распознавания рукописных цифр выходных нейронов будет 10 – по количеству цифр.

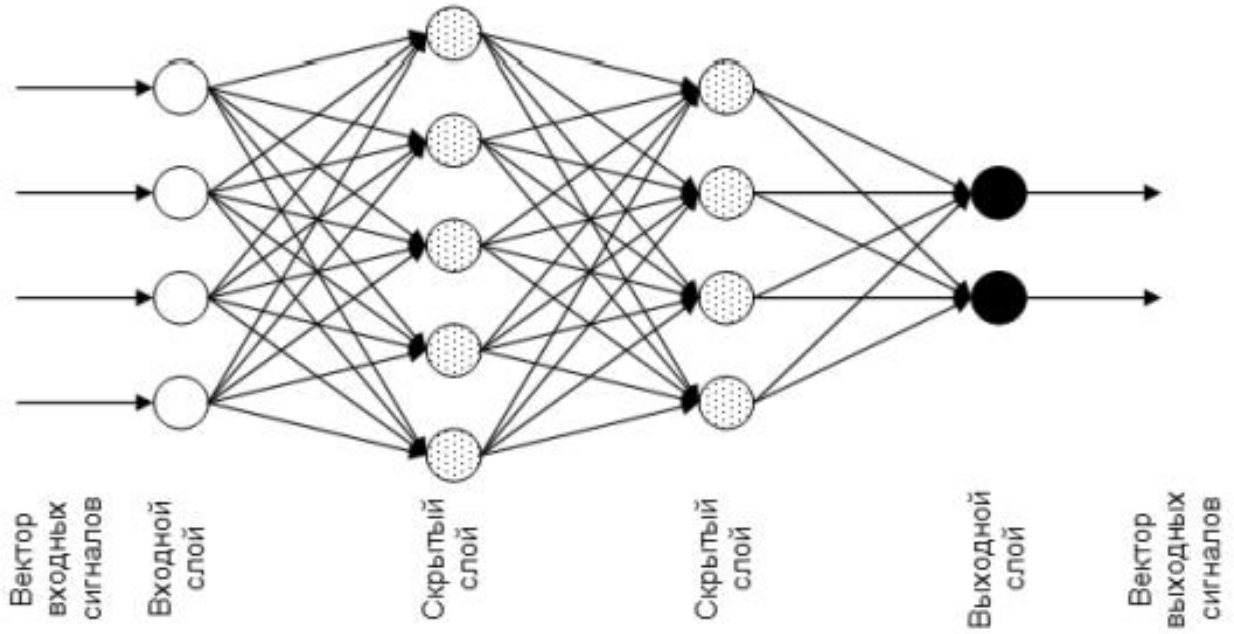

Рисунок 1.7 – многослойный персептрон для задачи бинарной классификации

Работа схемы на рисунке 1.7 представлена в нескольких последовательных подсчётах значений каждого слоя нейронной сети. Входной слой получает вектор данных, посчитывает взвешенную сумму, использует функцию активации и получает значение выходных сигналов. Выходные значения нейронов становятся входными значениями для следующего слоя. Далее повторяем операции с каждым слоем, пока не получим значения на выходном слое.

Чтобы узнать, какой ответ выдала нейронная сеть, необходимо воспользоваться функцией $argmax(Y)$. Получаем номер класса, к которому принадлежат входные данные. Существует практика использования функции $softmax(x_i) = \frac{e^{x_i}}{\sum e^x}$, которая возвращает вектор вероятностей

принадлежности к каждому классу [9]. Это необходимо, когда важно понимать, с какой уверенностью нейронная сеть классифицирует объекты.

1.5 Обучение нейронной сети

Остановимся на выбранной архитектуре нейронной сети и подробнее разберём многослойный персептрон, который ещё называют полносвязной нейронной сетью. Для решения задач распознавания образов это хороший вариант алгоритма. Он способен с большой точностью классифицировать рукописные цифры, а также обучается за довольно короткий промежуток времени. Алгоритм прямого процесса подсчёта выходных сигналов нейронной сети был разобран в пункте 1.4. Для выбранной архитектуры рассмотрим процесс обучения, основанный на алгоритме обратного распространения ошибки [10].

Основная идея состоит том, чтобы выбрать порцию данных для обучения и получить вывод нейронной сети для всей порции. Далее оцениваем величину ошибки нейронной сети при помощи, так называемой loss функции.

Существует несколько функций ошибок, например, CE – cross entropy (1.9), LSE - least squares method (1.10).

$$\text{cross entropy} = -\sum Y^*(\tau) \log Y(\tau) \tag{1.9}$$

$$LSE = \frac{1}{2}\sum_{\tau \in U_{out}} \big(Y(\tau) - Y^*(\tau)\big)^2 \tag{1.10}$$

где $Y^*(\tau)$ – истинное значение нейрона, $Y(\tau)$ – предсказанное значение.

Перекрёстная энтропия используется в задачах, где важна вероятность предсказания. Для задачи распознавания рукописных цифр достаточно выбрать метод наименьших квадратов в качестве функции потерь. С точки зрения производительности операции вычитания и возведения квадрат выполняются быстрее подсчёта логарифма, что также ускоряет обучение.

Цель обучения состоит в минимизации функции ошибок. Входные данные для обучения не изменяются, следовательно, необходимо изменять веса и смешение каждого слоя нейронной сети пропорционально тому, как они влияют на функцию ошибок.

Для выполнения задачи изменения весов можно воспользоваться методом градиентного спуска, который известен из курса математического анализа.

Весовые коэффициенты и смещения следует изменять по следующим формулам [11]:

$$w_{i,j} = w_{i,j} + \alpha \cdot \frac{dE}{dw_{i,j}},\ b_i = b_i + \alpha \cdot \frac{dE}{db_i} \quad (1.11)$$

где E – функции ошибки, α- коэфициент скорости обучения.

Введём обозначения:

X_i - входной вектор, Y_i – выходной вектор, $w_{i,j}^k$- i-ый весовой коэффициент j-го нейрона k-го слоя, b_i^k- смешение i-го нейрона k-го слоя, Y^*- истинное значение i-го нейрона, S_j – взвешенная сумма j-го нейрона.

Выходное значение j-го нейрона k-го слоя вычисляется по формуле:

$$Y_j^k = F(\sum w_{i,j}^k \cdot Y_i^{k-1} - b_j^k)$$

Выходное значение j-го нейрона выходного слоя вычисляется по формуле:

$$Y_j = F(\sum w_{i,j} \cdot Y_i^{n-1} - b_j) \quad (1.12)$$

Функция ошибки из формулы (1.10) равна $E = \frac{1}{2}\sum_j \left(Y_j - Y_J^*\right)^2$, где $\gamma_j = Y_j - Y_j^*$ - ошибка j-го нейрона выходного слоя. Ошибка j-го элемента k-го скрытого слоя:

$$\gamma_j^k = \frac{\partial E}{\partial Y_j^k} = \sum_j \frac{\partial E}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial Y_j^k} = \sum_j \frac{\partial E}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial S_j} \cdot w_{i,j}$$

$$= \sum_j \left(Y_j - Y_j^*\right) \cdot F'(S_j) \cdot w_{i,j} = \sum_j \gamma_j \cdot F'\left(S_j\right) \cdot w_{i,j}$$

Градиенты ошибок равны:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial w_{i,j}} = \gamma_j \cdot F'\left(S_j\right) \cdot Y_j^k$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial b_j} = -\gamma_j \cdot F'\left(S_j\right)$$

$$\frac{\partial E}{\partial w_{i,j}^k} = \sum_j \frac{\partial E}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial Y_j^{k-1}} \cdot \frac{\partial Y_j^{k-1}}{\partial S_j^{k-1}} \cdot \frac{\partial S_j^{k-1}}{\partial w_{i,j}^k} = \gamma_j \cdot F'\left(S_j^k\right) \cdot Y_j^k$$

Получаем формулы весовых коэффициентов и смещение нейронов:

$$w_{i,j}^k = w_{i,j}^k - \alpha\gamma_j^k \cdot F'\left(S_j^k\right) \cdot Y_j^k \quad (1.13)$$

$$b_j^k = b_j^k - \alpha\gamma_j^k \cdot F'\left(S_j^k\right) \quad (1.14)$$

Обобщим алгоритм обучения многослойной нейронной сети [11]:

1) Задаётся шаг обучения α $(0 < \alpha < 1)$ и желаемая среднеквадратичная ошибка сети E_m.
2) Инициализируем случайным образом весовые коэффициенты $w_{i,j}^k$ и пороговые b_j^k значения нейронной сети. Лучшей практикой будет использовать непрерывное равномерное распределение [12] на отрезке $(-\alpha, \alpha)$.
3) Случайным образом подаются данные из обучающей выборки на входной слой нейронов. Хорошей практикой для ускорения вычислений является объединение различных изображений в так называемый batch (массив образов). Далее для каждого массива

вычисляется процедура прямого распространения данных по нейронной сети. Получаем значения Y_j^k для выходных нейронов.

4) Вычисляется ошибка γ_jдля нейронов входного слоя и далее для скрытых слоёв.
5) Изменяются веса и смещения нейронов для каждого слоя сети.
6) Вычисляется полная ошибка нейронной сети E.
7) Если $E > E_m$, то переходим к шагу 3, в противном случае обучение завершается.

Глава 2 Технология CUDA

2.1 Введение в технологию параллельных вычислений на видеокартах

CUDA (Compute Unified Device Architecture) – это аппаратно-программная платформа для параллельных вычислений, которая использует ресурсы графического процессора NVIDIA для неграфических вычислений [4].

Параллельные вычисления на GPU уже давно используются в решении повседневных задач, например, рендеринг изображения на экран или обучение простых нейронных сетей на домашних компьютерах.

Во многих существующих алгоритмах применение технологии CUDA позволило существенно повысить производительность вычислений. Первые графические процессоры с поддержкой CUDA были представлены компанией NVIDIA ещё в 2007 году. Основной особенностью данных процессоров является возможность выполнять большое количество процессов одновременно. Вычислительные возможности каждого из процессов меньше чем у CPU, но количество параллельных операций позволяет нивелировать этот недостаток.

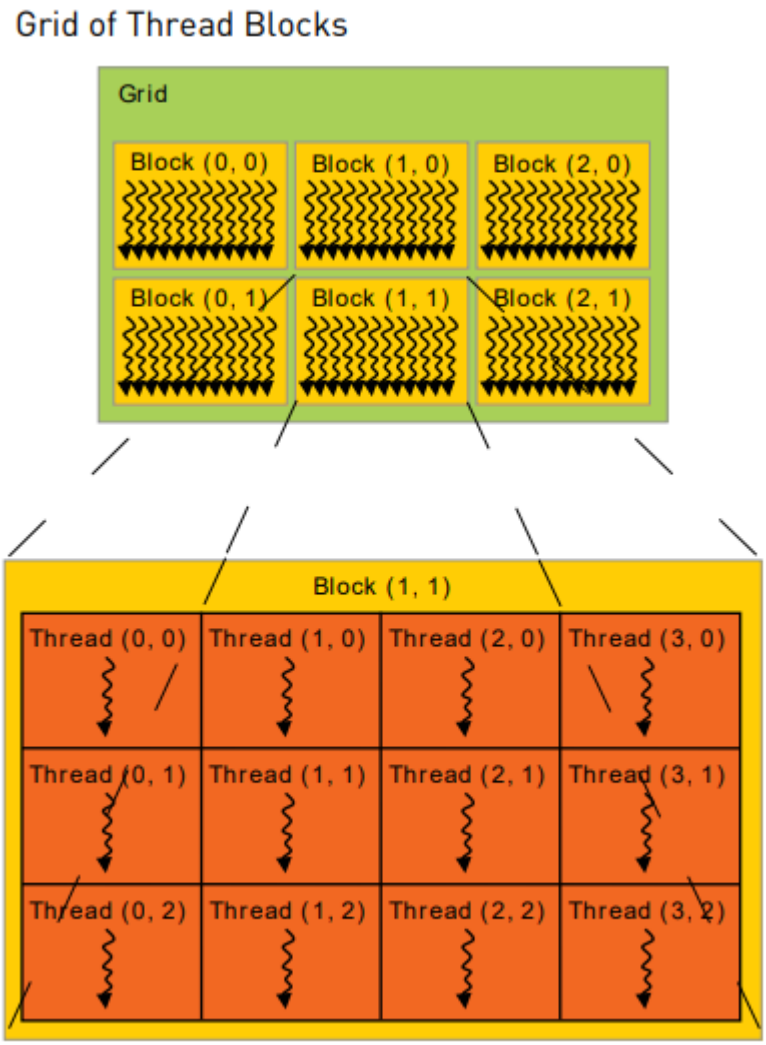

Рисунок 2.1 – организация структуры многопоточности на видеокартах компании NVIDIA [13]

Технология CUDA поддерживает модель SIMD (Single Instruction Multiple Data), которая позволяет выполнять одну инструкцию для всех запускаемых процессов. Особенностью выполнения инструкций является сложная структура графического процессора, которая состоит из сетки, блоков, нитей (рисунок 2.1). Сетка (grid) графического процессора представляет собой одномерный, двухмерный или трехмерный массив блоков (block). Каждый блок – это одномерный, двухмерный или трехмерный массив нитей (thread) [14]. Поскольку одно ядро выполняется на каждой нити, необходимо назначить каждому потоку исполнения

определённый номер. Для решения данной задачи используются зарезервированные переменные blockIdx и threadIdx, которые являются трехмерным целочисленным вектором [14].

Для исполнения инструкций кода вызываемых с CPU на GPU используется префикс для функции __global__. Для вызова функций исключительно внутри GPU используется префикс __device__ [14].

Пример функции сложения двух одномерных векторов:

```
__global__ void add(const int *a, const int *b, int *c) {
     c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Вызов функции осуществляется при помощи специальной конструкции

```
add<<<Block_dim , Thread_dim>>>(d_a, d_b, d_c);
```

где Block_dim – количество блоков для вычислений, Thread_dim – количество нитей в каждом блоке. Задача отслеживания правильного выполнение инструкций потоком с необходимым номером лежит на создателе алгоритма.

Для использования данных на графическом устройстве необходимо выделить для них память специальной функцией:

```
cudaMalloc((void**)&devicePtr, byteSize)
```

где devicePtr - указатель на глобальную память, byteSize – размер выделяемой памяти в байтах.

2.2 Технология Numba

Рассмотрим возможности и способы доступа к вычислениям на GPU через реализацию программного пакета Numba. Данный пакет является

расширением языка программирования python, что позволяет довольно быстро и легко разрабатывать программное обеспечение. Numba – это JIT (Jist In Time) компилятор с открытым исходным кодом, который преобразует python инструкции в быстрый машинный код с использованием технологии LLVM (low level virtual machine) через python пакет llvmlite (рисунок 2.2). Пакет предлагает ряд опций для распараллеливания байт кода python для центральных и графических процессоров с некоторыми изменениями кода, направленными исключительно на оптимизацию выполняемых операций. Производительность компилируемого кода сравнима со скоростью выполнения C/C++ кода [15].

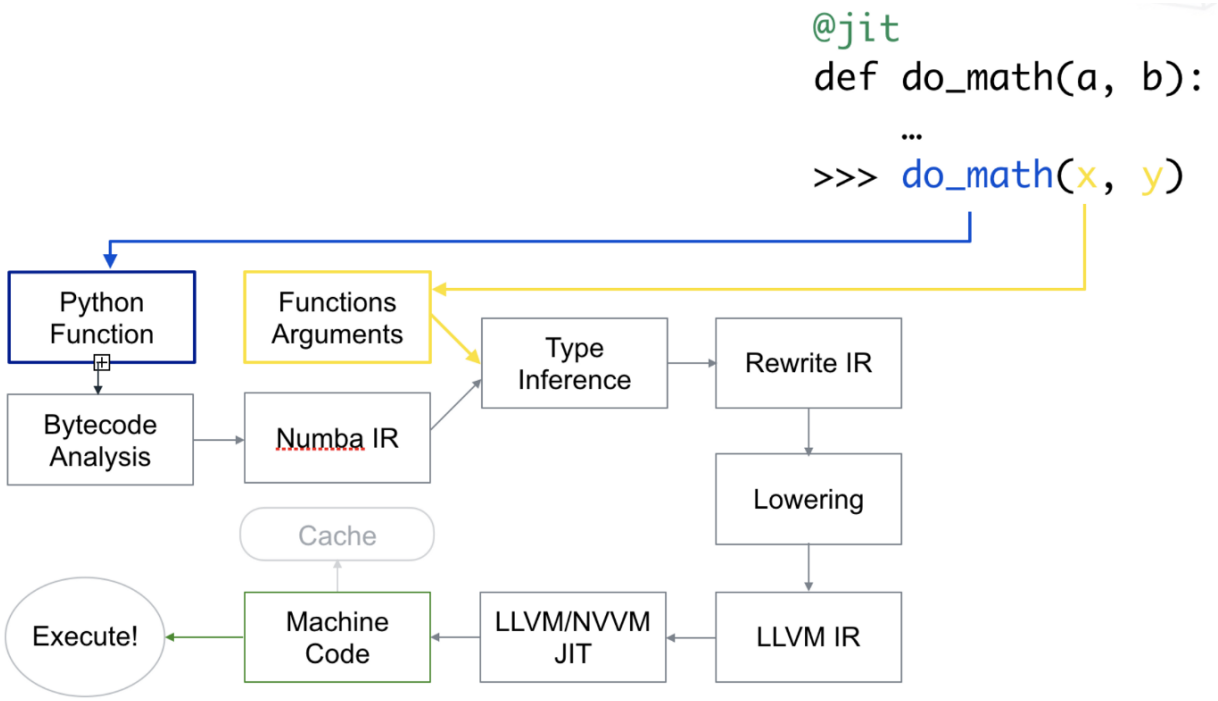

Рисунок 2.2 – структура работы декоратора @jit

Предположим, что есть функция `do_math`, которая оформлена с помощью декоратора Numba `@jit`. Компиляция будет отложена до первого выполнения функции, которая будет кэширована и использована позже при повторном запуске программы. Пакет Numba определит типы аргументов во время вызова и сгенерирует оптимизированный код на основе этой

информации. Следующим шагом будет промежуточное представление байт кода IR (Intermediate Representations) для уже определённых аргументов функции. На последнем этапе низкоуровневая виртуальная машина (LLVM – для центральных процессоров, NVVM – для графических процессоров) компилирует промежуточное представление IR в машинный код и сохраняет его в кэш [16].

2.3 Преимущество технологии CUDA

Области применения вычислений на графических ускорителях постоянно расширяются из-за постоянного технического прогресса. Уже сегодня, многие задачи, которые требовали огромного количества вычислительных ресурсов и несколько дней времени, решаются за считанные минуты с применением технологии CUDA. В среднем при переносе вычислений с CPU на GPU достигается ускорение в 5-30 раз. Для задачи распознавания рукописных цифр существуют операции, которые можно вычислять на видеокартах. Например, матричное сложение и умножение (формула 1.7). Подробно рассмотрим данные операции на CPU и GPU, а также сравним абсолютное и относительное время выполнения операций в зависимости от размера матриц.

Для выполнения на CPU будем использовать python пакет numpy, который использует алгоритмы, написанные на языке программирования C. Так мы не потеряем в производительности при использовании стандартных операций интерпретатора python, который в разы медленнее функций, написанных на C.

Функция умножения матриц для GPU имеет вид:

```
from numba import cuda
@cuda.jit
def matmul(matrix1, matrix2, out):
```

```
i, j = cuda.grid(2)
if i < out.shape[0] and j < out.shape[1]:
    tmp_value = 0.
    for k in range(matrix1.shape[1]):
        tmp_value += matrix1[i, k] * matrix2[k, j]
    out[i, j] = tmp_value
```

Операция `cuda.grid(2)` возвращает абсолютное положение текущего потока во всей сетке. Вычисление первого числа происходит следующим образом: `cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x`. Для измерений y и z аналогично.

Возьмём квадратные матрицы размерностей от 50 до 8000 с шагом 50. Тогда график зависимости времени выполнения операции будет выглядеть следующим образом (рисунок 2.3):

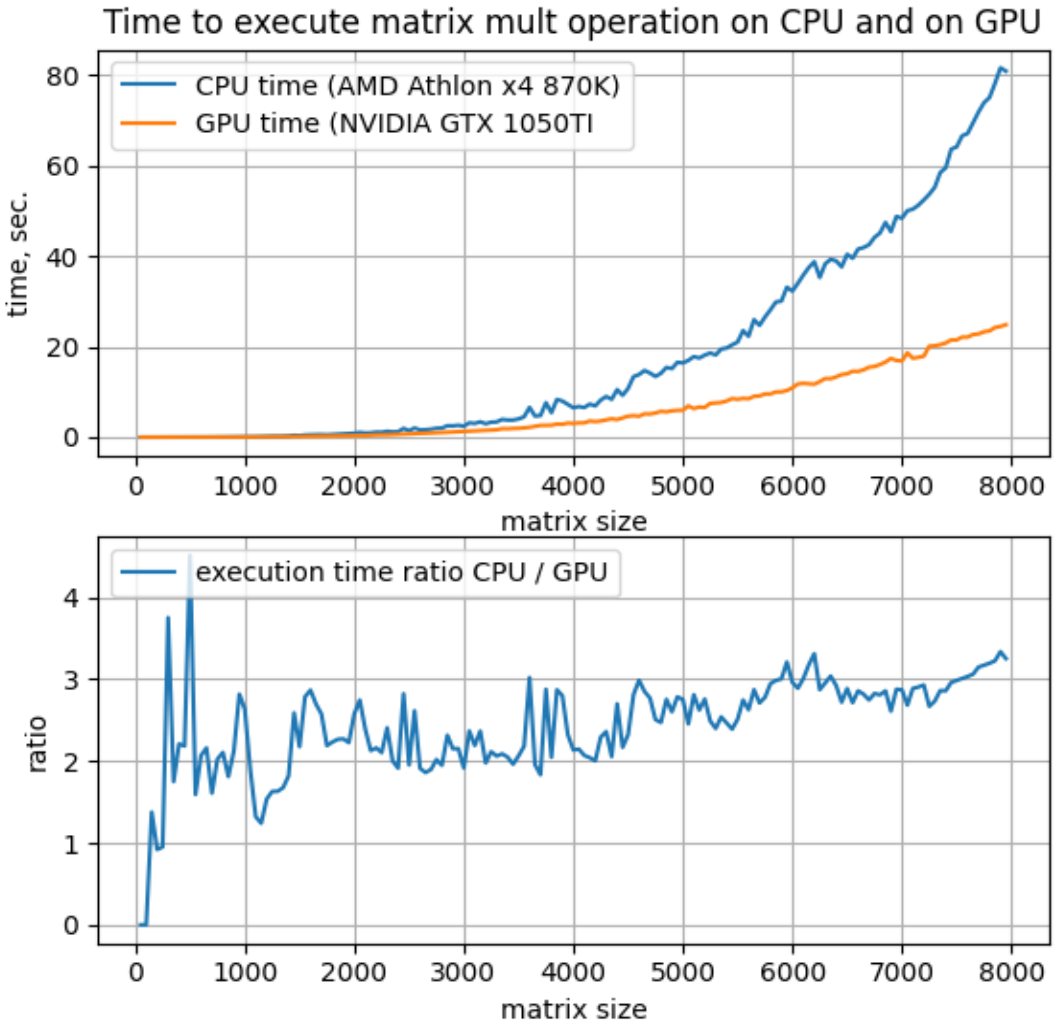

Рисунок 2.3 – график зависимости времени выполнения матричного умножения от размера матрицы

Аналогично выполним анализ выполнения алгоритма суммирования матриц (рисунок 2.4). Функция сложения матриц для GPU имеет вид:

```
from numba import cuda
@cuda.jit
def add(arr1, arr2):
    x, y = cuda.grid(2)
    if x < arr1.shape[0] and y < arr1.shape[1]:
        arr1[x, y] += arr2[x, y]
```

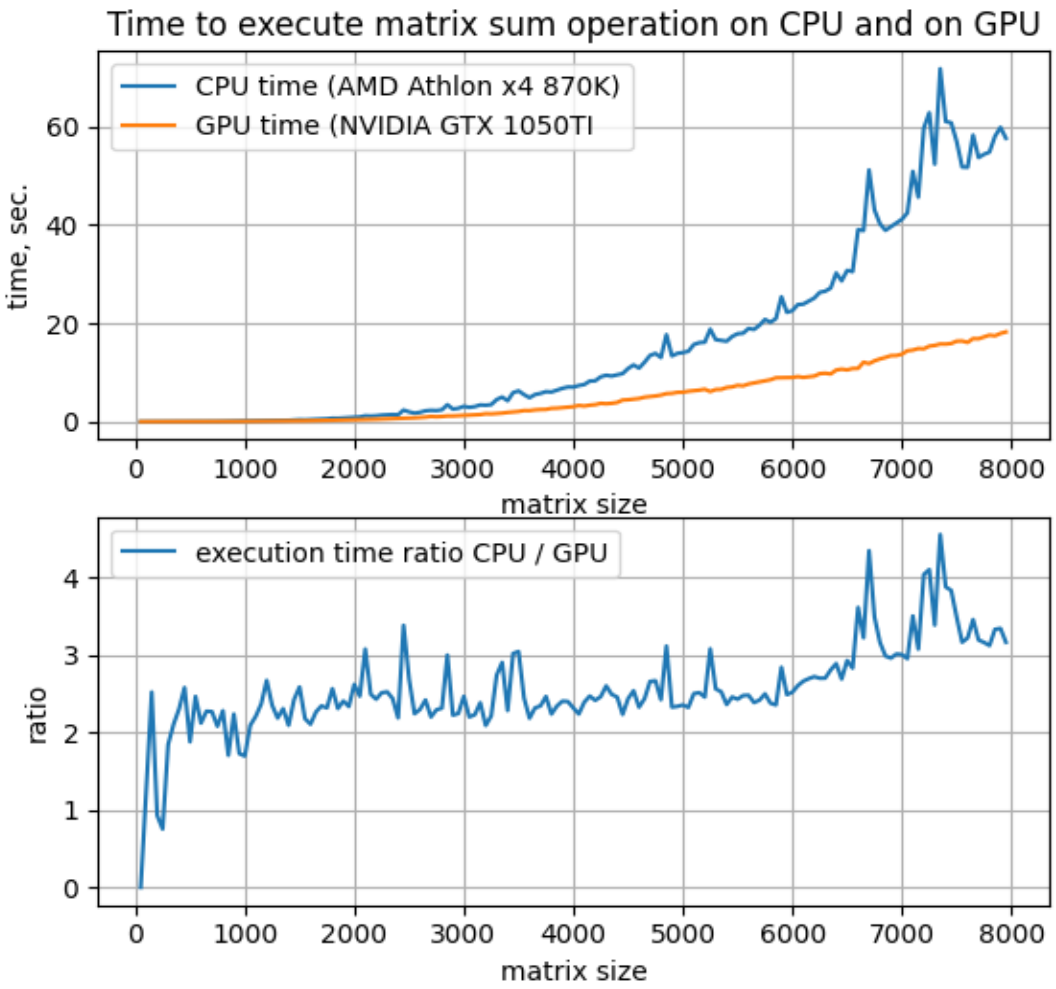

Рисунок 2.4 – график зависимости времени выполнения матричного сложения от размера матрицы

Можем заметить, что скорость выполнения вычислений на GPU значительно лучше, чем на CPU. Отношение времени выполнения при размерности 8000 x 8000 достигает прироста в 3 и более раз для рассмотренных операций. Таким образом, можно сделать вывод, что технология CUDA применима при решении множества математических задач, в которых требуется параллельно обрабатывать большое количество данных.

Глава 2 Реализация нейронной сети для задачи классификации

3.1 Данные для обучения

Необходимо подобрать данные для обучения нейронной сети. Для решения задачи классификации рукописных цифр рассмотрим датасет Mnist [17] с открытого информационного ресурса.

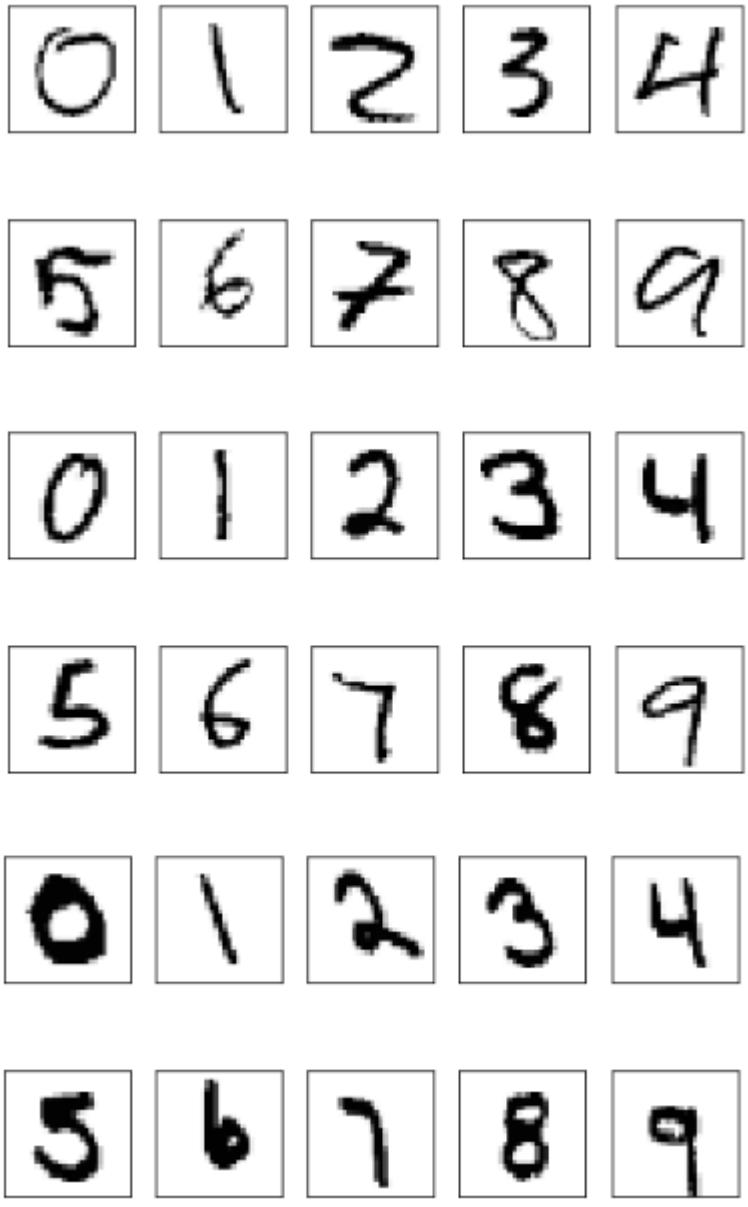

Рисунок 3.1 – визуализация датасета Mnist

На рисунке 3.1 представлена визуализация части данных. Можем заметить, что изображения имеют черно-белый формат и выровнены по центру. Данные представлены в байтовом представлении. Каждый пиксель - байтовое число от 0 до 255, следовательно, для обучения поделим каждый пиксель на максимальное значение 255. Таким образом, значения пикселей находятся на отрезке [0,1]. Размеры изображений составляют 28x28 пикселей. Количество обучающей выборки составляет 60000 изображений, тестовой 10000.

Рисунок 3.2 – визуализация датасета Fashion Mnist

Существует аналогичный набор изображений, который называется Fashion Mnist (рисунок 3.2). Его единственным отличием является то, что на нём изображены элементы одежды вместо цифр. При проверке работоспособности нейронной сети также проверим обучаемость на наборе данных Fashion Mnist.

3.2 Реализация GPU функций

Реализуем необходимые для выполнения на графическом процессоре функции. Для формулы (1.7) необходимо выполнять две операции: умножение и сложение матриц. Данные функции были подробно представлены в главе 2.3. Для вычисления ошибки по формуле (1.10) Реализуем функцию вычитания на GPU:

```
@cuda.jit
def subtract(arr1, arr2, out):
     x, y = cuda.grid(2)
     if x < arr1.shape[0] and y < arr1.shape[1]:
          out[x, y] = arr1[x, y] - arr2[x, y]
```

Распишем реализацию прямого распространения сигнала согласно формуле (1.12). Создадим фабрику для генерации функции с одним аргументом в виде функции активации. Во время инициализации нейронной сети будет компилироваться код с выбранной конфигурацией.

```
def make_feedforward_step(act_f):
         @cuda.jit
         def _feedforward_step(inputs, weights, biases, outputs):
             x, batch = cuda.grid(2)
             if x<outputs.shape[0] and batch<outputs.shape[1]:
                 outputs[x, batch] = matmul_device(weights, inputs)
                 cuda.atomic.add(outputs, (x, batch), biases[x, 0])
```

```
            outputs[x, batch] = act_f(outputs[x, batch])
    return _feedforward_step
```

Для выполнения подсчёта активационных функций на девайсе укажем ключевой параметр в декораторе `@cuda.jit(device=True)`. Функции активации и их производные будут исполняться на GPU.

```
@cuda.jit(device=True)
def sigmoid(x):
    '''sigmoid'''
    return 1 / (1 + exp(-x))

@cuda.jit(device=True)
def dsigmoid(y):
    '''derivative sigmoid'''
    return y * (1 - y)

@cuda.jit(device=True)
def relu(x):
    '''LeakyReLU'''
    return 1. + 0.01 * (x - 1.) if x > 1
          else x * 0.01 if x < 0
          else x

@cuda.jit(device=True)
def drelu(y):
    '''derivative LeakyReLU'''
    return 0.01 if y < 0 or y > 1 else 1
```

Реализуем алгоритм обратного распространения ошибки. Для вычисления градиента напишем фабричную функцию, которая аналогично с функцией прямого распространения принимает производную функции активации в виде параметра. Фабрика возвращает функцию подсчёта

градиента на основе векторов ошибок и гиперпараметра α - коэфициента обучения.

```
def make_gradient(dact_f):
    @cuda.jit
    def _gradient(outputs, err, lr, gradients):
        x, batch = cuda.grid(2)
        if x < outputs.shape[0] and batch < outputs.shape[1]:
            gradients[x, batch] = dact_f(outputs[x, batch]) *
err[x, batch] * lr
    return _gradient
```

3.3 Реализация обучения

Сигнатура создания класса нейронной сети выглядит следующим образом:

```
NeuralNetwork(784, 512, 10, dataset=mnist_dataset,
                            act_f='relu',
                            thread_per_block=(32,32))
```

Первые три параметра отвечают на количество нейронов в каждом слое. Первый и последний слои зафиксированы. Датасет содержит изображения 28x28, что соответствует 784 входам нейронной сети. Количество классов для распознавания соответствует выходному количеству нейронов. Далее идут три именованных параметра, среди которых датасет, функция активации и количество нитей для каждого блока вычислений.

При инициализации нейронной сети создаются соответствующие матрицы весовых коэффициентов и смещений по закону равномерного распределения на отрезке $(0,1)$. Функция обучения имеет сигнатуру:

```
def train(self, epochs, learning_rate, batch_size):.
```

Обучение проходит в несколько эпох, задаваемых значением `epochs`. Значение `learning_rate` – коэффициент обучения нейронной сети. Для

улучшения качества обучения сделаем его не статическим, а динамическим. Зададим формулу изменения коэффициента в зависимости от текущего номера эпохи по формуле (3.1) экспоненциального затухания.

$$lr_{epoch} = learning\ rate \cdot e^{-\left(\frac{epoch}{epochs}\right)} \quad (3.1)$$

За размер батча отвечает параметр `batch_size`, который позволяет значительно ускорить время вычислений. Приведём график времени выполнения одной эпохи в зависимости от размера батча (рисунок 3.3).

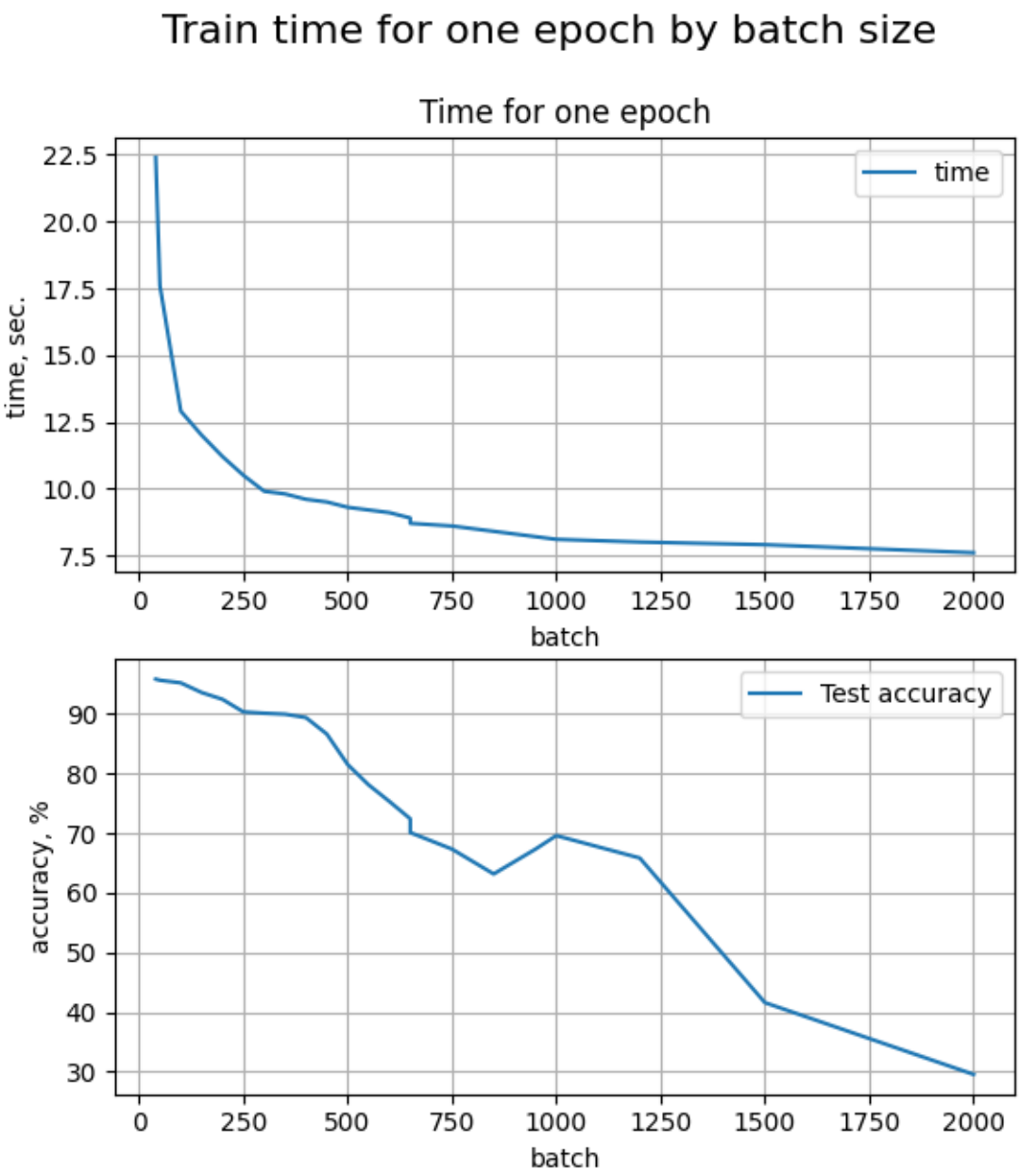

Рисунок 3.3 – график зависимости времени выполнения и точности на тестовой выборке от размера батча

На рисунке 3.3 подробно показана зависимость скорости обучения от размера батча. Самым медленным местом в программе может быть копирование памяти с хоста на девайс и с девайса на хост [14]. Используя наивную реализацию без оптимизации, получаем необходимость синхронно копировать данные сначала в глобальную память, а только после - в память

графического устройства. Чем меньше размер батча, тем больше необходимо провести операций копирования и вызовов математических функций. Этого можно избежать, правильно подобрав размер батча и используя асинхронные вызовы, которые будут разобраны при оптимизации программы.

В реализации используется алгоритм случайной перестановки (random permutation [18]) внутри всей выборки, чтобы случайным образом упорядочить множество данных для обучения. Это необходимо для того, чтобы процесс обучения не зависел от порядка обучающих элементов.

На основании полученных результатов выберем оптимальных размер батча как 100 изображений. Получаем среднюю скорость выполнения одной эпохи 13 секунд при точности на тестовой выборке более 92% после первой эпохи обучения.

Изначальная реализации метода обратного распространения ошибки начинается с переноса данных в глобальную память. `inputs_d` – входные изображения в виде матрицы размерности 784 x размер батча. `targets_d` - ожидаемые выходные данные в формате one hot encode.

```
# 784 x batch
inputs_d = cuda.to_device(inputs)
# 10 x batch
targets_d = cuda.to_device(targets)
```

Далее дважды выполняется функция прямого распространения сигнала `feedforward_step`, получаем выход нейронной сети. Вычисляем ошибки и градиенты для каждого слоя. Изменяем веса и смещения согласно формулам (1.13-1.14).

Текущая реализация позволяет достигать точности 96% на обучающей выборке и 94% на тестовой. Среднее время обучения одной эпохи составляет 195 секунд, что чрезвычайно долго. Следует заглянуть внутрь выполнения алгоритмов и с помощью современных программ профилирования кода

замерить время выполнения каждой операции. Необходимо найти замедляющие вычисления места программы и отредактировать их.

Глава 3 Оптимизация обучения нейронной сети

4.1 Инструменты профилирования

Оптимизация кода играет важную роль в разработке программного обеспечения. Чтобы верно диагностировать место, где можно оптимизировать код, необходимо использовать инструменты профилирования для получения информации о работе приложения [19]. Мониторинг работы программы и использования ресурсов компьютера может осуществляться по следующим параметрам:

1) замер времени исполнения отдельных инструкций
2) общее время выполнения функций, количество вызовов функций
3) максимальное и минимальное время исполнения процедуры
4) связанный граф исполнения функций
5) потребление памяти устройства
6) обращение к программным или аппаратным ресурсам компьютера, например, к файловым дескрипторам или графическим процессорам.

Для профилирования воспользуемся современными инструментами и анализаторами. Выделяют два самых главных типа профайлеров: статистический и динамический.

Принцип работы статистического профайлера состоит в сохранении информации о текущей исполняемой операции, состоянии памяти и других важных данных. Мониторинг данных происходит через заданные промежутки времени, например, каждую миллисекунду. Главным недостатком является возможность выполнения функции или инструкции быстрее, чем за 1 миллисекунду. В таком случае информация о состояниях регистров, локальных переменных и стеке вызовов не может быть сохранена. Вторая проблема связана с тем, что одна функция может быть вызвана

несколько раз подряд. Тогда невозможно определить, сколько вызовов функции было произведено за промежуток профилирования.

Принцип работы динамического профайлера состоит в измерении времени между некоторыми событиями в программе. Каждое событие запоминает последний вызов функции, время выполнения и другую сопутствующую информацию. Основным недостатком можно считать замедление времени выполнения программы. Процесс исполнения может сильно отличатся от условий работы без инструмента профилирования. С помощью данного подхода можно успешно выявить места длительного выполнения кода и оптимизировать их.

4.2 Поиск проблемного кода

После процесса разработки программного продукта появляется необходимость протестировать существующий код. В этом случае многие разработчики программного обеспечения сразу стараются использовать cProfile – один из стандартных модулей python [19]. Запустим процедуру профилирования основного файла программы. Заметим, что время выполнения при профилировании значительно увеличилось. Для более удобной читаемости воспользуемся пакетом gprof2dot для визуализации дерева вызовов. Переходим в корневую папку проекта и вводим следующую команду в консоль:

```
python -m cProfile -o output.pstats nn.py gprof2dot.py -f pstats output.pstats | dot -Tpng -o output.png
```

Вывод работы профилировщика будет представлен в файле output.png в текущем каталоге. Рассмотрим рисунок 4.1 подробнее.

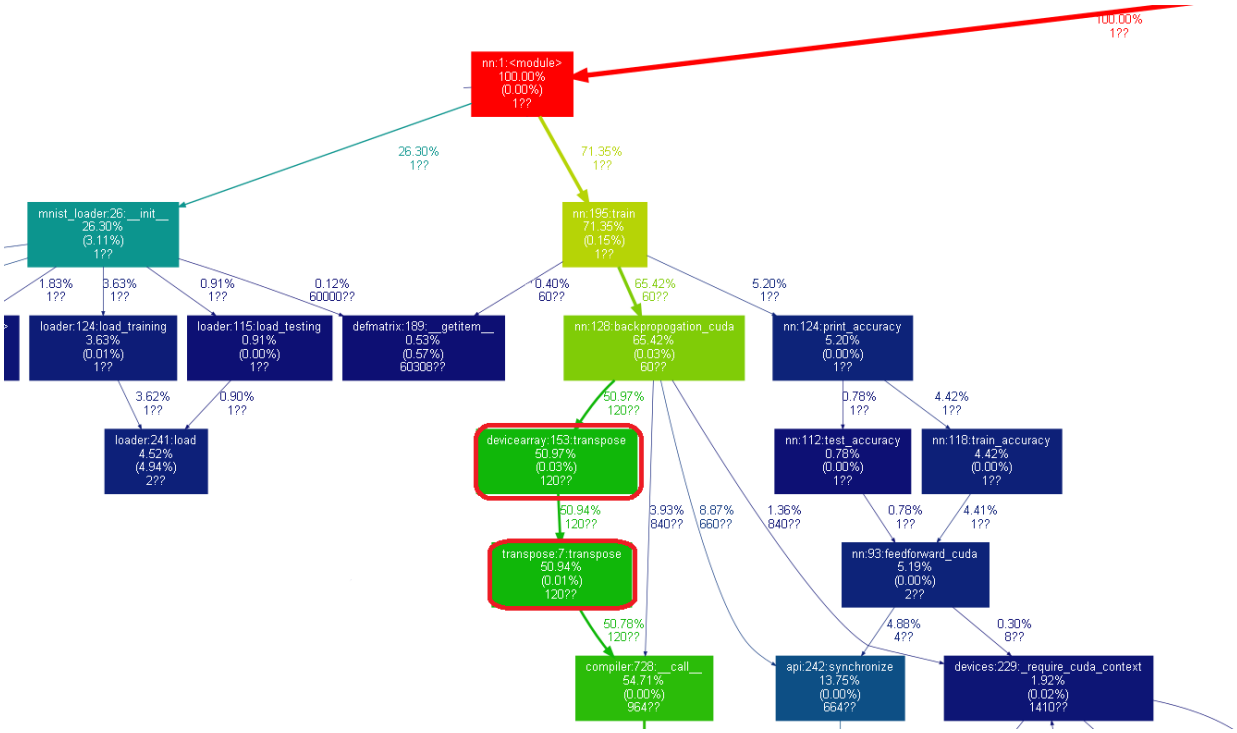

Рисунок 4.1– визуализация результатов профилирования cProfile

Входной точкой в начало выполнения программы является красный прямоугольник. Для тестирования обучение нейронной сети проходило одну эпоху. Заметим, что загрузка данных происходит быстро и выполняется один раз за время выполнения программы. Основная нагрузка по времени идёт на функцию обратного распространения ошибки, а именно на операцию `devicearray:153:transpose`, 50.97% времени вызова из общего времени исполнения функции `nn:128:backpropogation_cuda`.

4.3 Оптимизация проблемного кода

Рассмотрим операции транспонирования, вызовы которых выполняются при обучении. Действительно, в алгоритме были использованы библиотечные вызовы функции транспонирования данных в памяти графического ускорителя. В коде присутствуют два вызова функции

транспонирования. Первый вызов осуществляется во время операции подсчёта ошибки для скрытого слоя нейронов:

```
cu_k.matmul[self.get_grid_dim(hidden_errors_d.shape)](self.weigh
ts[1].transpose(), errors_d, hidden_errors_d)
```

Второй вызов происходит во время подсчёта изменения весовых коэффициентов нейронной сети:

```
cu_k.matmul[self.get_grid_dim(self.weights[1].shape)](gradient_d
, hidden_d.transpose(), delta_w_d)
```

Эти вызовы колоссально замедляют выполнение программы. Заглянем в определение функции и получим данный код:

```
def transpose(self, axes=None):
    return FakeCUDAArray(numpy.transpose(self._ary, axes=axes))
```

При вызове операции транспонирования происходит вызов создания нового массива с переносом данных из памяти GPU в оперативную память. Далее массив обрабатывается при помощи стандартной функции `numpy.transpose(self._ary, axes=axes)` и данные копируются обратно на GPU. Избавимся от вызовов стандартных функций путём создания нового ядра для умножения матрицы на матрицу с учётом того, что одна из них должна быть транспонирована.

```
@cuda.jit
def matmul_T(matrix1_T, matrix2, matrix_out):
    i, j = cuda.grid(2)
    if i < matrix_out.shape[0] and j < matrix_out.shape[1]:
        tmp_value = 0.
        for k in range(matrix1_T.shape[0]):
            tmp_value += matrix1_T[k, i] * matrix2[k, j]
        matrix_out[i, j] = tmp_value
```

Время обучения нейронной сети после изменений в среднем составило 17 секунд для одной эпохи. Поиск и оптимизация кода привели к ускорению

выполнения обучения нейронной сети более чем в 11 раз. Это доказывает необходимость применения инструментов профилирования кода в процессе разработки. Как оказалось, стандартное API библиотеки создаёт множество излишних вызовов, которые отрицательно сказываются на производительности.

4.4 Дополнительные сведения о потоках технологии CUDA

В текущей реализации нейронная сеть обучается, последовательно вызывая функции работы с памятью и ядра графического процессора. Использование нескольких параллельных потоков исполнения потенциально может быть эффективнее, чем последовательное исполнение.

По умолчанию все операции ассоциированы с нулевым потоком исполнения и выполняются последовательно. Графические процессоры с вычислительной мощностью более 2.0 могут выполнять несколько операций параллельно, например, копирование с хоста на девайс, копирование с девайса на хост, выполнение ядра. При извлечении свойств устройства следует проверить свойство `concurrentKernels`, которое сообщает, что графический процессор при наличии ресурсов может выполнять несколько ядер параллельно. Использование асинхронных функций передачи данных с указанием стримов значительно ускоряет выполнение программы [14]. Асинхронные функции передачи данных работают только с закреплённой памятью (pinned memory), которая в свою очередь ускоряет доступ к памяти устройства. Закреплённая память имеет одно важное свойство – она гарантированно не будет выгружена операционной системой. Потоки имеют возможность синхронизации [4].

4.5 Оптимизация потоков выполнения

На этапе оптимизации потоков выполнения кода рассмотрим инструмент Visual Profiler от компании NVIDIA. Данное приложение предоставляет необходимую информацию о вызовах функций и использовании процессорного времени. На рисунке 4.2 рассмотрим визуальное представление времени выполнения обработки одного батча информации. Заметим, что в вычислениях используется один стандартный поток.

Рисунок 4.2 – визуализация времени выполнения GPU функций

Применим знания оптимизации с использованием потоков и асинхронных вызовов. Выделим закреплённую память для обучающих данных с помощью контекстного менеджера python:

```
with
cuda.pinned(self.train_data),cuda.pinned(self.train_targets):
```

Теперь появилась возможность вызывать копирование памяти в память устройства через асинхронные функции. Для оптимизации следует разделить

независимые операции на два потока. Представим потоки операций в таблице 4.1.

Таблица 4.1 – Очереди выполнения команд для потоков

Поток 1	Поток 2
inputs_d = cuda.to_device(inputs)	targets_d = cuda.to_device(targets)
hidden_d = cuda.device_array	inputs_d_T = cuda.to_device(inputs.T)
feedforward_step(inputs_d, self.weights[0], self.biases[0], hidden_d)	
outputs_d = cuda.device_array	
feedforward_step(inputs_d, self.weights[1], self.biases[1], outputs_d)	
errors_d = cuda.device_array	
subtract(targets_d, outputs_d, errors_d)	hidden_errors_d = cuda.device_array
stream1.synchronize()	
	matmul_T(self.weights[1], errors_d, hidden_errors_d)
stream2.synchronize()	
gradient_d = cuda.device_array	
gradient(outputs_d, errors_d, lr, gradient_d)	
delta_b_d = cuda.device_array	
sum_cols(gradient_d, delta_b_d)	
stream1.synchronize()	
add (self.biases[1], delta_b_d)	gradient_d = cuda.device_array
delta_w_d = cuda.device_array	gradient(hidden_d, hidden_errors_d, lr, gradient_d)
matmul(gradient_d, inputs_d_T, delta_w_d)	delta_b_d = cuda.device_array
add(self.weights[0], delta_w_d)	sum_cols(gradient_d, delta_b_d)
	add (self.biases[0], delta_b_d)
stream1.synchronize()	stream2.synchronize()

Сравним время выполнения до оптимизации и после. Выберем конфигурацию в 512 нейронов в скрытом слое, размер батча 1000, активационная функция LeakyReLU. Время выполнения одной эпохи до оптимизации составило 9.7 секунды. Время выполнения одной эпохи после оптимизации составило 6.5 секунды.

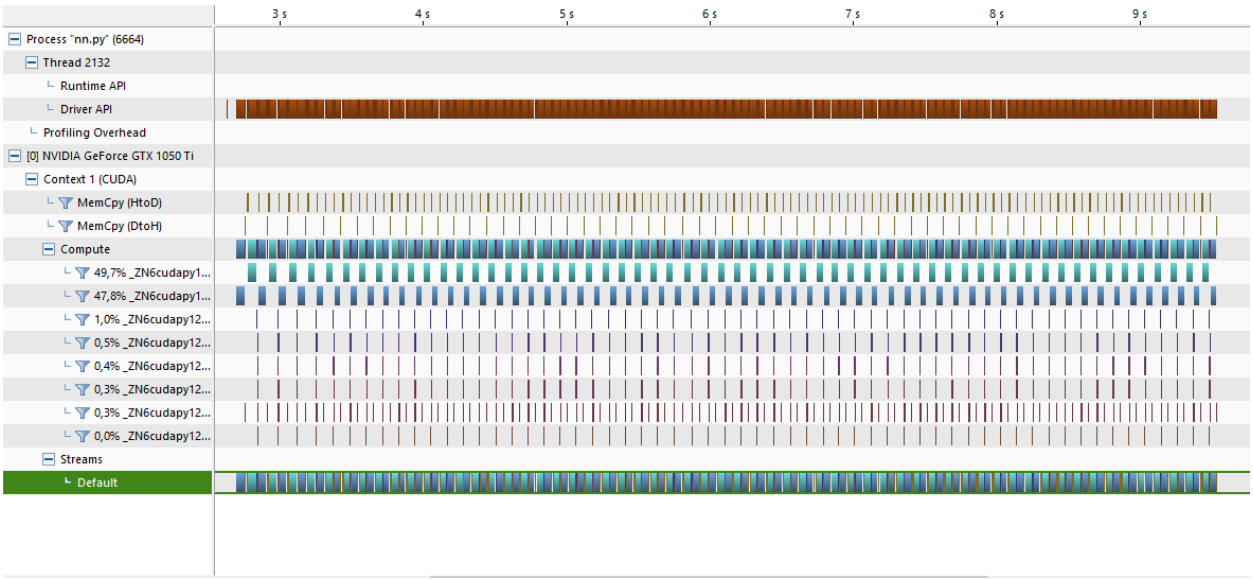

Рисунок 4.3 – визуализация времени выполнения GPU функций до оптимизации с потоками

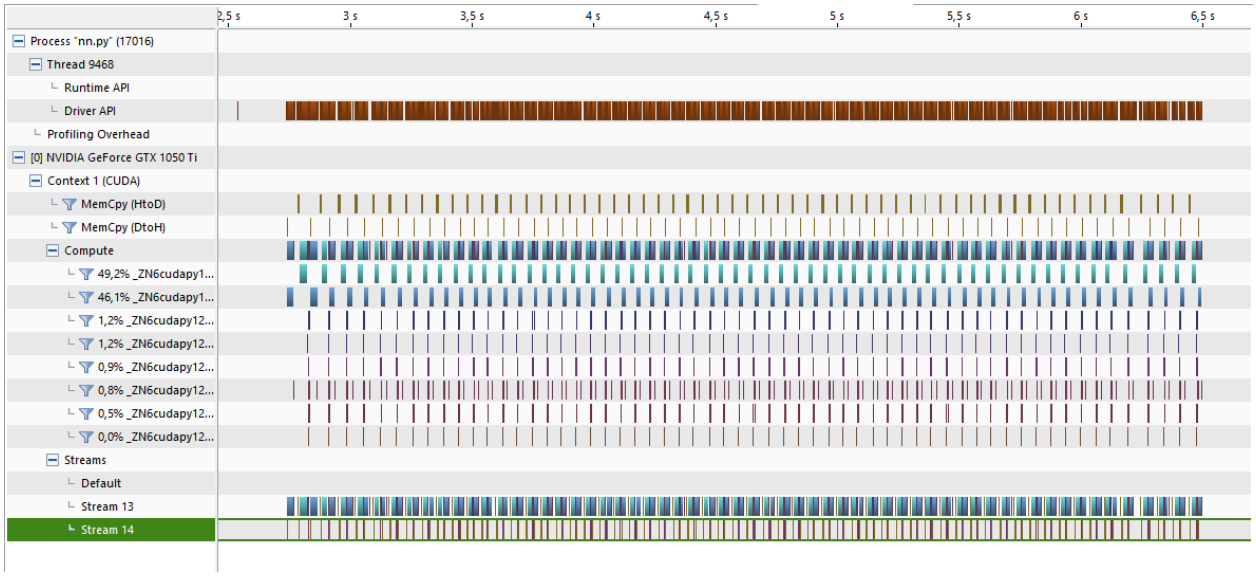

Рисунок 4.4 – визуализация времени выполнения GPU функций после оптимизации с потоками

Изучим информацию на рисунках 4.3, 4.4. На рисунке 4.3 видно, что процессом обучения управлял один поток, в очереди которого все операции выполнялись последовательно. На рисунке 4.4 видно, что после оптимизации, часть операций перешла от потока с номером 13 к потоку с номером 14. Так же оптимизация с использованием закреплённой памяти позволила асинхронно копировать данные, что ускорило время выполнения программы. Время выполнения алгоритма ускорилось более чем на 30%.

Глава 5 Анализ результатов

5.1 Анализ точности обучения нейронной сети

Проведём сравнительный анализ архитектур нейронной сети. Обучим реализованную архитектуру на датасетах Mnist и Fashion Mnist. Сравним точность распознавания при различных параметрах сети, таких как коэффициент обучения и количество нейронов в скрытом слое.

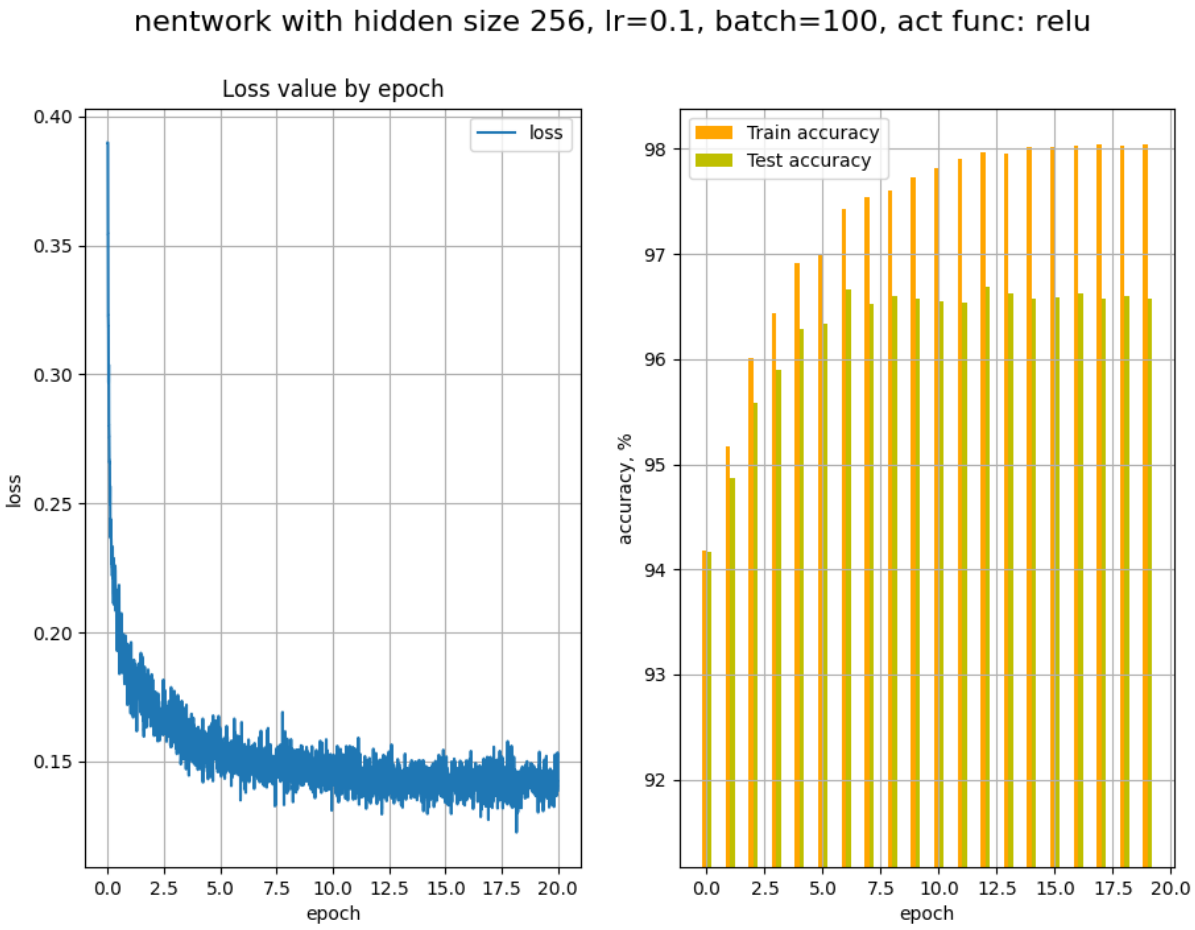

Рисунок 5.1 – функция ошибки и точность обучения сети с 256 скрытыми нейронами, коэффициентом обучения 0.1, размером батча 100, активационной функцией LeakyReLU на датасете Mnist

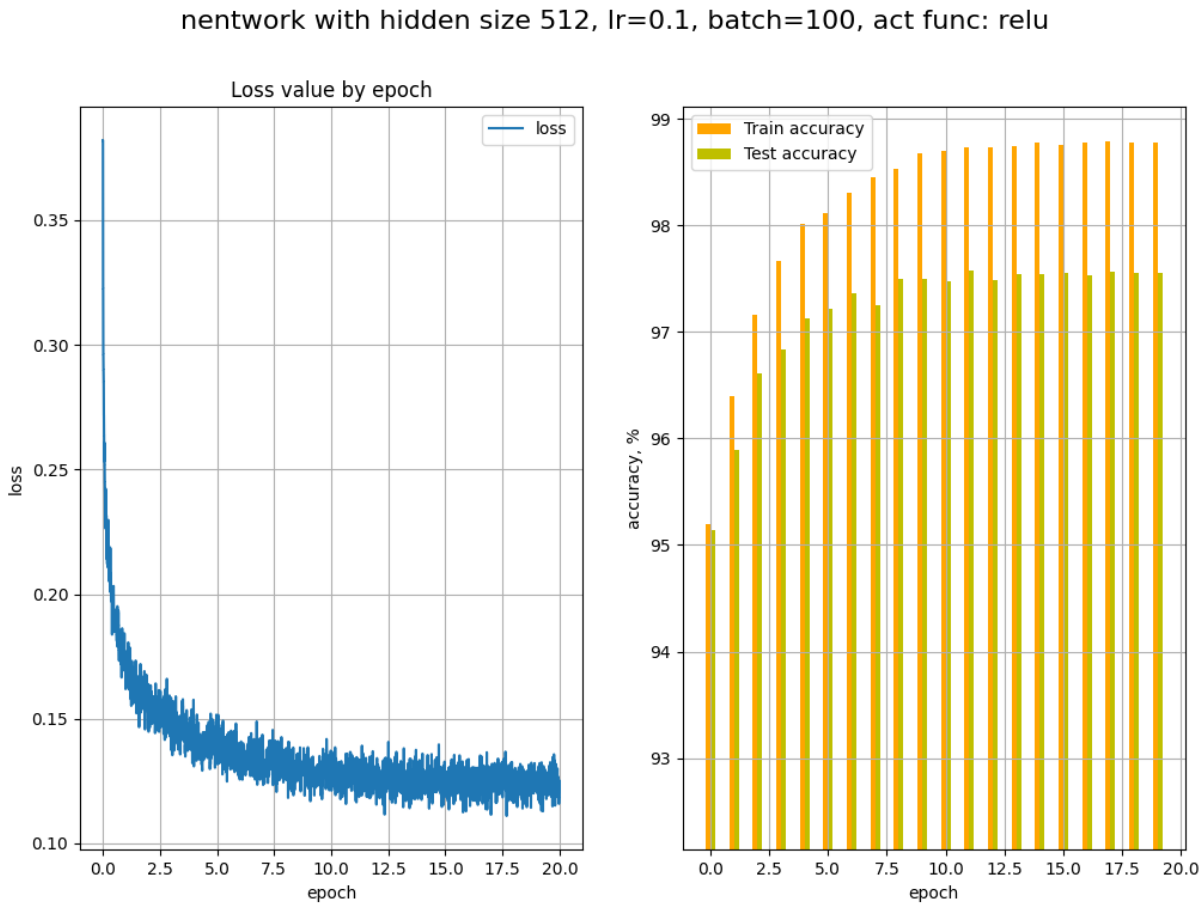

Рисунок 5.2 – функция ошибки и точность обучения сети с 512 скрытыми нейронами, коэффициентом обучения 0.1, размером батча 100, активационной функцией LeakyReLU на датасете Mnist

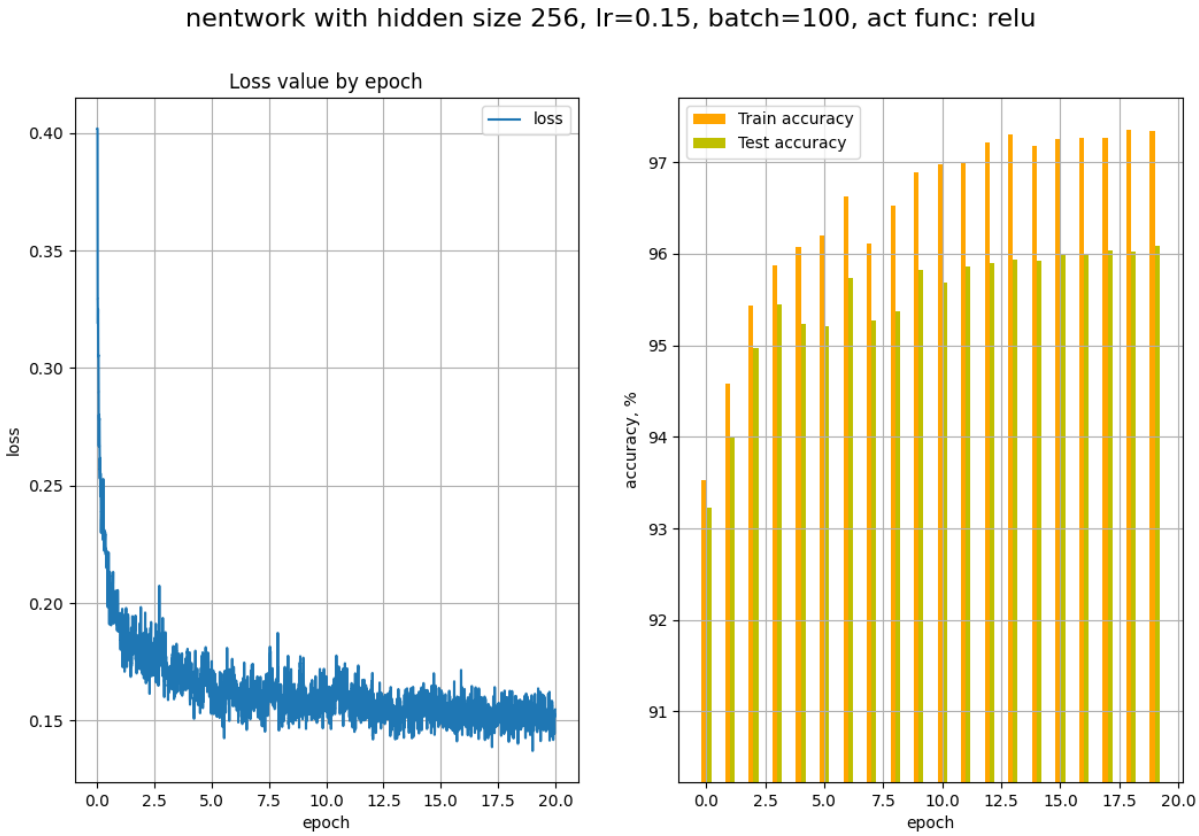

Рисунок 5.3 – функция ошибки и точность обучения сети с 256 скрытыми нейронами, коэффициентом обучения 0.15, размером батча 100, активационной функцией LeakyReLU на датасете Mnist

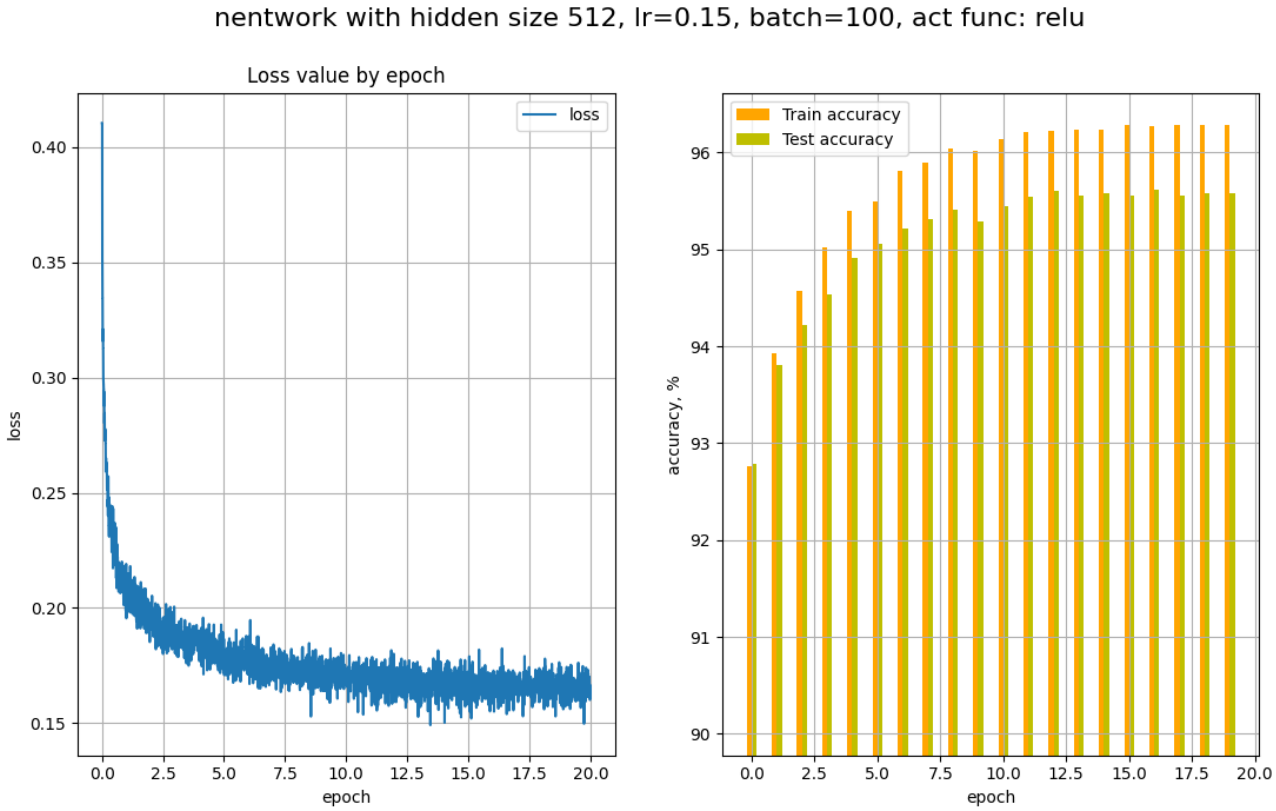

Рисунок 5.4 – функция ошибки и точность обучения сети с 512 скрытыми нейронами, коэффициентом обучения 0.15, размером батча 100, активационной функцией LeakyReLU на датасете Mnist

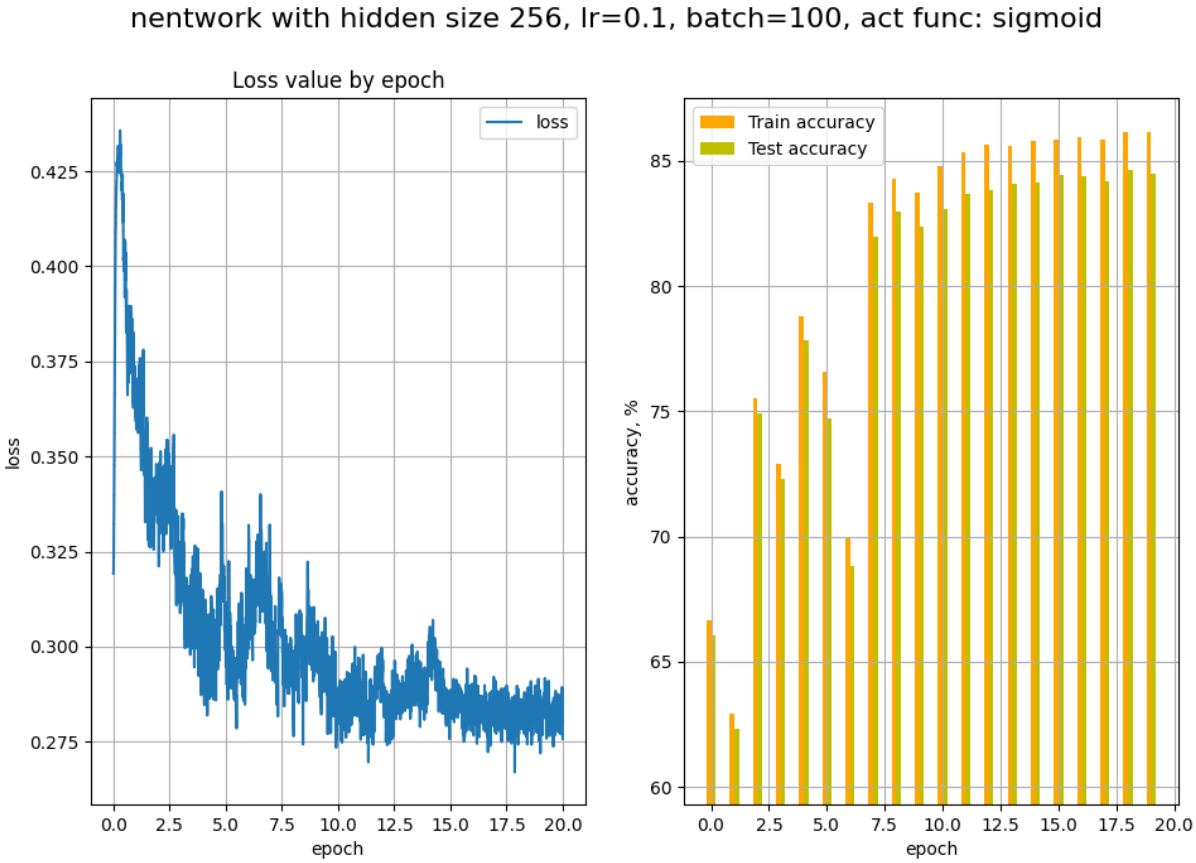

Рисунок 5.5 – функция ошибки и точность обучения сети с 256 скрытыми нейронами, коэффициентом обучения 0.1, размером батча 100, активационной функцией Sigmoid на датасете Fashion Mnist

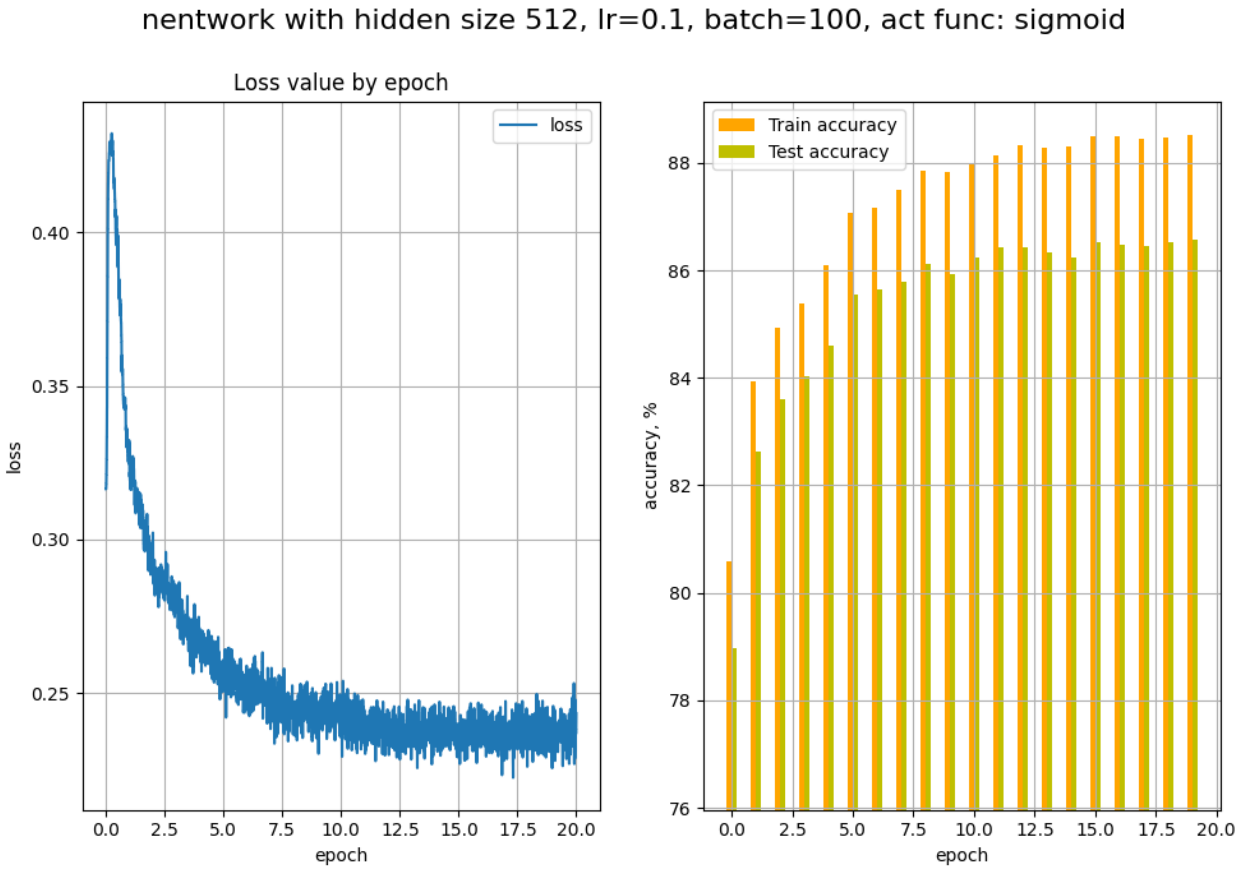

Рисунок 5.6 – функция ошибки и точность обучения сети с 512 скрытыми нейронами, коэффициентом обучения 0.1, размером батча 100, активационной функцией Sigmoid на датасете Fashion Mnist

Сравнительный анализ показал, каким образом параметры нейронной сети влияют на показатели средней ошибки и точности. На рисунках видно, что более высокое значение коэффициента обучения не позволяет функции ошибок быстро уменьшаться. Заметим, что количество нейронов в скрытом слое играет важную роль в точности классификации. Чем больше нейронов в скрытом слое, тем больше правильных и точных ответов у нейронной сети. Выбранная и реализованная архитектура успешно обучилась на датасете Fashion Mnist с хорошей точностью. Это позволяет сделать вывод, что нейронную сеть можно обучать на аналогичных датасетах.

5.2 Анализ скорости обучения нейронной сети

В данной главе рассмотрим прирост производительности за счёт применения технологии параллельного программирования CUDA. Протестируем время обучения аналогичной нейронной сети для классификации рукописных цифр с помощью фреймворка pytorch. Выполним обучение на CPU и GPU. Конфигурация сети состоит из одного скрытого слоя с 512 нейронами, функция активации ReLu, размер батча 100 изображений. Обучение выполняется до момента достижения точности 95% на тестовой выборке. Время обучения на CPU до достижения заданной точности составило 102.38 секунды. Время обучения на GPU составило 26.86 секунд. Время обучения реализованной в рамках задачи нейронной сети составило 32.3 секунды.

Результаты скорости обучения показывают, что вычисления на видеокартах значительно превосходят вычисления на центральном процессоре. Реализованная в работе нейронная сеть превосходит время обучения на CPU, но в то же время хуже, чем реализация обучения на GPU при помощи специальной библиотеки pytorch. Можно сделать вывод, что реализованный алгоритм можно оптимизировать.

Основной идеей реализации алгоритма обучения с применением технологии CUDA было ускорение вычислений относительно CPU. Поставленную задачу получилось выполнить полностью. Предложенная архитектура нейронной сети обучается до определённой точности в несколько раз быстрее обучения на CPU.

ЗАКЛЮЧЕНИЕ

В настоящее время алгоритмы машинного обучения и нейронных сетей находят широкое применение во многих сферах жизни, начиная с медицинских исследований и производства инструментов, заканчивая военной техникой и системами видеонаблюдения. Тем не менее, несмотря на их актуальность, немногие могут полностью разобраться с принципами работы нейронных сетей. Создаются многочисленные архитектуры НС, в которых можно запутаться. Одна из задач, которую решают нейронные сети – задача распознавания образов. Подобный вопрос требует комплексного подхода и применения современных технологий для его решения. Одной из таких технологий является CUDA – универсальная платформа для вычислений на видеокартах. Графические устройства позволяют выполнять большое количество операций параллельно.

В последнее время применение технологии CUDA в алгоритмах нейронных сетей является популярной практикой. Обучение устроено на многомерных операциях с большим количеством данных. Графические ускорители помогают работать с большими данными и позволяют ускорять вычисления в десятки раз, а в некоторых случаях - в сотни раз. Полученное в работе решение задачи распознавания цифр с применением технологии CUDA позволяет использовать его в дальнейших разработках. Новизна исследований заключается в оптимизации алгоритмов выполнения кода на видеокартах при помощи особых конструкций платформы, а также ускорения разработки за счёт использования упрощённого API к графическому ускорителю с помощью python библиотеки numba.

В рамках данной работы были изучены технологии:

1) CUDA (Compute Unified Device Architecture) – технология выполнения кода на графических ускорителях от фирмы NVIDIA.

2) Технология numba, необходимая для упрощённого доступа к API видеокарты.
3) CProfile - технология профилирования python кода.
4) NVIDIA Visual Profiler - технология профилирования кода, исполняющегося на GPU.

Для достижения результата были решены следующие задачи:

1) Изучены существующие алгоритмы нейронных сетей.
2) Для задачи классификации рукописных цифр была подобрана оптимальная архитектура нейронной сети.
3) Было проведено исследование и разбор математических формул, необходимых для реализации нейронной сети.
4) Изучена технология параллельного программирования CUDA. Была выбрана уникальная технология доступа к ресурсам графических ускорителей через библиотеку numba языка python.
5) Проведён сравнительный анализ времени выполнения алгоритмов на GPU и на CPU.
6) Реализована полносвязная нейронная сеть для решения задачи распознавания рукописных чисел. Применена технология CUDA.
7) Проведён анализ результатов обучения нейронной сети. Проведён анализ времени обучения алгоритма.
8) Выполнена оптимизация кода с помощью динамических профилировщиков.

Таблица 1 – Компетенции и освоенные навыки

Компетенция и расшифровка	Освоенные навыки
УК-1 Способен осуществлять поиск, критический анализ и синтез информации, применять системный подход для решения поставленных задач	Был осуществлён поиск необходимых инструментов для реализации задачи. Подобранные библиотеки были проанализированы и успешно применены при решении задачи.
УК-2 Способен определять круг задач в рамках поставленной цели и выбирать оптимальные способы их решения, исходя из действующих правовых норм, имеющихся ресурсов и ограничений	Был определён список задач, необходимых для достижения цели. Был произведён поиск открытых ресурсов для использования информации при решении задач. Действующие нормы и ограничения были соблюдены.
УК-3 Способен осуществлять социальное взаимодействие и реализовывать свою роль в команде	Были исполнены роли разработчика, тестировщика, аналитика, дизайнера и проектировщика программного обеспечения.
УК-4 Способен осуществлять деловую коммуникацию в устной и письменной формах на государственном языке Российской Федерации и иностранном(ых) языке(ах)	Была осуществлена в устной и письменной форме деловая и научная коммуникация с научным руководителем.
УК-5 Способен воспринимать межкультурное разнообразие общества в социально-историческом, этическом и философском контекстах	Был проведен поиск по открытым источникам в международных ресурсах.

УК-6 Способен управлять своим временем, выстраивать и реализовывать траекторию саморазвития на основе принципов образования в течение всей жизни	Был продуман поэтапный план выполнения работ, и задачи были выполнены вовремя.
УК-7 Способен поддерживать должный уровень физической подготовленности для обеспечения полноценной социальной и профессиональной деятельности	Был проведён должный контроль над состоянием физической подготовленности и обеспечен необходимый уровень физической формы во время выполнения выпускной квалификационной работы.
УК-8 Способен создавать и поддерживать безопасные условия жизнедеятельности, в том числе при возникновении чрезвычайных ситуаций	Был проведен инструктаж по технике безопасности на время прохождения учебной практики и выполнения выпускной квалификационной работы.
ОПК-1 Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности	Были исследованы современные архитектуры нейронных сетей и математические аспекты их работы. Проведён глубокий анализ необходимых алгоритмов для успешной реализации поставленной задачи.
ОПК-2 Способен использовать и адаптировать существующие методы и системы программирования для разработки и реализации алгоритмов решения прикладных задач	Были успешно применены современные методы разработки и оптимизации алгоритмов. Были использованы инструменты профилирования кода, оптимизации времени выполнения.

ОПК-3 Способен применять и модифицировать математические модели для решения задач в области профессиональной деятельности	Был проведен анализ возможных решений поставленной задачи, из которых был выбран оптимальный вариант решения задачи, с учетом всех преимуществ и недостатков. Были проведены работы с приведений математических моделей к конкретному варианту решению задачи.
ОПК-4 Способен решать задачи профессиональной деятельности с использованием существующих информационно-коммуникационных технологий и с учетом основных требований информационной безопасности	Был произведен поиск по самой актуальной литературе из открытых источников. Были использованы последние технические решения из области параллельных вычислений и информационных технологий. Были применены лучшие практики для реализации поставленной задачи.
ПК-1 Проверка работоспособности и рефакторинг кода программного обеспечения, интеграция программных модулей и компонент и верификация выпусков программного обеспечения	Была реализована нейронная сеть и успешно обучена на данных из открытых источников. Верификация выходных данных была протестирована. Проект разбит на функциональные и модульные составляющие. Получен опыт рефакторинга и обеспечения работоспособности ПО.

ПК-2 Мониторинг функционирования интеграционного решения в соответствии с трудовым заданием, работа обращениями пользователей по вопросам функционирования интеграционного решения в соответствии с трудовым заданием	В проект были интегрированы средства визуального информирования о работе алгоритма. Проведён анализ функционирования системы, полноты выводимой информации.
ПК-3 Проверка и отладка программного кода, тестирование информационных ресурсов с точки зрения логической целостности (корректность ссылок, работа элементов форм)	Разработанный алгоритм был протестирован на подобранных данных. Проведён контроль качества приложения, анализ корректности работы алгоритма.
ПК-4 Ведение информационных баз данных	Была использована база данных изображений рукописных цифр Mnist и база изображений элементов одежды Fashion Mnist.
ПК-5 Обеспечение функционирования баз данных	Был написан специальный класс для взаимодействия с базами данных. Чтение базы данных было реализовано через байтовый поток.

ПК-6 Педагогическая деятельность по проектированию и реализации общеобразовательных программ	Были изучены последние практики разработки и проектирования программного обеспечения. Были использованы последние инструменты. Полученный опыт можно использовать в педагогической деятельности.
ПК-7 Разработка и документирование программных интерфейсов, разработка процедур сборки модулей и компонент программного обеспечения, разработка процедур развертывания и обновления программного обеспечения	Был разработан программный интерфейс и описана его работа. Была создана инструкция для развёртывания виртуального окружения и запуска программы.
ПК-8 Применять методы и средства сборки модулей и компонент программного обеспечения, разработки процедур для развертывания программного обеспечения, миграции и преобразования данных, создания программных интерфейсов	Была проведена настройка программного окружения с помощью модуля pip. Процедура сборки программного обеспечения осуществлялась с помощью открытой программы Visual Studio Code.
ПК-9 Описание возможной архитектуры развертывания каждого компонента, включая оценку современного состояния предлагаемых архитектур, оценка архитектур с точки зрения надежности правовой поддержки	Были описаны необходимые зависимости и версии пакетов для разработки и запуска программного обеспечения.

ПК-10 Документальное предоставление прослеживаемости требований, согласованности с системными требованиями; приспособленность стандартов и методов проектирования; осуществимость, функционирования и сопровождения; осуществимость программных составных частей	Были изучены и применены современные практики построения приложения, ведения репозитория. Разработка велась с документированием системных требований для успешного запуска приложения.
ПК-11 Техническое сопровождение возможных вариантов архитектуры компонентов, включающее описание вариантов и технико-экономическое обоснование выбранного варианта	Было проведено исследование возможных архитектур разработки компонентов. Выбран оптимальный с технико-экономической точки зрения вариант проекта.
ПК-12 Выполнение работ по созданию (модификации) и сопровождению ИС, автоматизирующих задачи организационного управления и бизнес-процессы	Был продуман план работы по сопровождению информационной системы. Автоматизации работы с компонентами и системами контроля версий. Были продуманы бизнес-процессы.
ПК-13 Создание и сопровождение требований и технических заданий на разработку и модернизацию систем и подсистем малого и среднего масштаба и сложности	Были созданы технические задания на разработку и модернизацию системы. Рассмотрены возможные пути усовершенствования работы приложения.

ПК-14 Способность использовать основы экономических знаний в профессиональной деятельности	Были разобраны возможные варианты применения разработанного приложения в малом и среднем бизнесе.
ПК-15 Способность к коммуникации, восприятию информации, умение логически верно, аргументировано и ясно строить устную и письменную речь на русском языке для решения задач профессиональной коммуникации	Развита способность к восприятию сложной технической документации и информации. Развита устная и письменная речь.
ПК-16 Способность находить организационно-управленческие решения в нестандартных ситуациях и готовность нести за них ответственность	Развита способность находить нестандартные решения и применять их в разработке программного обеспечения.

ПК-17 Знание своих прав и обязанностей как гражданина своей страны, способностью использовать действующее законодательство и другие правовые документы в своей профессиональной деятельности, демонстрировать готовность и стремление к совершенствованию и развитию общества на принципах гуманизма, свободы и демократии	Была разработана система распознавания объектов с целью развития технологий общества и совершенствования инструментов производства. При разработке были использованы свободные и законные источники информации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Pun T. Image Analysis and Computer Vision in Medicine [текст]. / Pun T., Gerig G., Ratib O. // Computerized Medical Imaging and Graphics – 1993. – с. 4-5.
2. Matan O. Reading handwritten digits: A ZIP code recognition system [текст]. / Matan O., Bromley J., Burges C., Denker J., Jackel L., Le Cun Y., Pednault E., Satterfield W., Stenard C., Thompson T. // B: Computer 25.7 1992. – 33 с.
3. Coelho F. Automatic System for the Recognition of Amounts in Handwritten Cheques [текст]. / Coelho F., Batista L. Teixeira Filipe Luis, Cardoso J. // Portugal, Porto: Материалы конференции SIGMAP 2008 (26-29 июля 2008г.). - 2008, с. 320 - 324.
4. Сандерс Д., Кэндрот Д. Технология CUDA в примерах. Введение в программирование графических процессоров. – ДМК Пресс: Москва. – 2013. – 232 с.
5. Ерохин И.С. Введение в функциональную морфологию нервной системы: Методические материалы по биологии. / Ерохин И.С., Скобеева В.А., Чернов Т.А. // Москва, МФТИ – 2018. - с.11-15.
6. Ершов А.В. Лекции по линейной алгебре. / Ершов А.В. // Москва, МФТИ – 2022. – с. 100 – 108.
7. Activation function // Материал из Википедии – свободной энциклопедии [электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Activation_function, свободный (дата обращения: 20.05.2022).
8. Модели нейронных сетей [электронный ресурс]. Режим доступа: https://intuit.ru/studies/courses/6/6/lecture/178?page=5, свободный (дата обращения: 25.05.2022).
9. Заенцев И. В. Нейронные сети: основные модели: учебное пособие. / Заенцев И. В. – Воронеж, 1999. - с 22-33.

10. Buscema M. Back propagation neural networks // National Library of Medicine [электронный библиотека]. – 1998. – Режим доступа: https://pubmed.ncbi.nlm.nih.gov/9516725/ , свободный. (дата обращения: 22.05.2022).
11. Гафаров Ф. М. Исскуственные нейронные сети и их приложения: учебное пособие / Гафаров Ф. М., Галимянов А.Ф. – Казань: Изд-во Казанского ун-та. – 2018. – 121 с.
12. Володин И.Н. Лекции по теории вероятности и математической статистике. – Казань: (Издательство) , 2006. – с. 41-85.
13. CUDA C++ Programming Guide, NVIDIA. – 2022. // [эл. ресурс] – Режим доступа: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf , свободный (дата обращения: 26.05.2022).
14. Тумаков Д.Н Технология программирования CUDA: учебное пособие / Д.Н. Тумаков, Д.Е. Чикрин, А.А. Егорчев, С.В. Голоусов. – Казань: Казанский государственный университет, 2017. – 112 с.
15. Numba // Материал из Википедии – свободной энциклопедии [электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Numba , свободный (дата обращения: 27.05.2022).
16. Антонюк В.А. GPU+Python. Параллельные вычисления в рамках языка Python. / Антонюк В.А. // Москва: Физический факультет МГУ им. М. В. Ломоносова, 2018. – 48 с.
17. Yann LeCun и Corinna Cortes. The MNIST database of handwritten digits. 1998. [Электронный ресурс]. – Режим доступа: http://yann.lecun.com/exdb/mnist , свободный. (дата обращения: 26.05.2022).
18. Random permutation // Материал из Википедии – свободной энциклопедии [Электронный ресурс]. – Режим доступа:

https://en.wikipedia.org/wiki/Random_permutation , свободный (дата обращения: 29.05.2022).

19. Hattem R. Mastering Python. / Rick van Hattem // UK: Birmingham. – 2016, c. 351-367.

ПРИЛОЖЕНИЕ

Программный код загрузчика датасета:

Файл mnist_loader.py:

```
from mnist import MNIST
import numpy as np
class MnistDataSet(IDataset):
    NUMBER_OF_CLASSES = 10

    def __init__(self, ds_path="./mnist_dataset"):
        """Loads dataset of training and testing data"""
        print("Loading MNIST dataset ...", end='')
        # loading data
        self.train_data, self.train_labels =
MNIST(ds_path).load_training()
        self.test_data, self.test_labels =
MNIST(ds_path).load_testing()

        # translate data into range (0,1)
        self.train_data = np.matrix(self.train_data,
dtype=np.float64).T / 255
        self.test_data = np.matrix(self.test_data,
dtype=np.float64).T / 255

        self.train_labels = np.matrix(self.train_labels)
        self.test_labels = np.matrix(self.test_labels)

        # one hot encoding step
        self.train_targets =
np.zeros((MnistDataSet.NUMBER_OF_CLASSES,

self.train_data.shape[1]))
        for i in range(self.train_targets.shape[1]):
            self.train_targets[self.train_labels[0, i], i] = 1
        self.dataset_loaded = True
```

```
        print(" Done")
```

Программный код функций для графического процессора.

Файл cuda_kernels.py:

```
from math import exp, log
from numba import cuda, float64
import numpy as np
"""Cuda kernels for neural network"""

@cuda.jit(device=True)
def matmul_device(matrix1, matrix2):
    i, batch = cuda.grid(2)
    if i < matrix1.shape[0] and batch < matrix2.shape[1]:
        tmp_value = 0.
        for j in range(matrix1.shape[1]):
            tmp_value += matrix1[i, j] * matrix2[j, batch]
        return tmp_value

@cuda.jit
def matmul(matrix1, matrix2, out):
    i, j = cuda.grid(2)
    if i < out.shape[0] and j < out.shape[1]:
        tmp_value = 0.
        for k in range(matrix1.shape[1]):
            tmp_value += matrix1[i, k] * matrix2[k, j]
        out[i, j] = tmp_value

@cuda.jit
def matmul_T(matrix1_T, matrix2, matrix_out):
    i, j = cuda.grid(2)
    if i < matrix_out.shape[0] and j < matrix_out.shape[1]:
        tmp_value = 0.
```

```
        for k in range(matrix1_T.shape[0]):
            tmp_value += matrix1_T[k, i] * matrix2[k, j]
        matrix_out[i, j] = tmp_value


@cuda.jit
def gradient_sm(outputs, err, lr, gradients):
    x, batch = cuda.grid(2)
    if x < outputs.shape[0] and batch < outputs.shape[1]:
        gradients[x, batch] = dsoftmax(outputs[x, batch]) *
err[x, batch] * lr


@cuda.jit(device=True)
def relu(x):
    '''LeakyReLU'''
    return 1. + 0.01 * (x - 1.) if x > 1 else x * 0.01 if x < 0
else x


@cuda.jit(device=True)
def drelu(y):
    '''derivative LeakyReLU'''
    return 0.01 if y < 0 or y > 1 else 0.1


@cuda.jit(device=True)
def sigmoid(x):
    '''sigmoid'''
    return 1 / (1 + exp(-x))


@cuda.jit(device=True)
def dsigmoid(y):
    '''derivative sigmoid'''
    return y * (1 - y)
```

```
@cuda.jit
def feedforward_step(inputs, weights, biases, outputs):
    x, batch = cuda.grid(2)
    if x < outputs.shape[0] and batch < outputs.shape[1]:
        outputs[x, batch] = matmul_device(weights, inputs)
        cuda.atomic.add(outputs, (x, batch), biases[x, 0])
        outputs[x, batch] = sigmoid(outputs[x, batch])

@cuda.jit
def gradient(outputs, err, lr, gradients):
    x, batch = cuda.grid(2)
    if x < outputs.shape[0] and batch < outputs.shape[1]:
        gradients[x, batch] = dsigmoid(outputs[x, batch]) *
err[x, batch] * lr

@cuda.jit
def sum_cols(in_arr, out):
    x = cuda.grid(1)
    if x < in_arr.shape[0]:
        out[x, 0] = 0
        for y in range(in_arr.shape[1]):
            out[x, 0] +=  in_arr[x, y]

@cuda.jit
def subtract(arr1, arr2, out):
    x, y = cuda.grid(2)
    if x < arr1.shape[0] and y < arr1.shape[1]:
        out[x, y] = arr1[x, y] - arr2[x, y]

@cuda.jit
def add(arr1, arr2):
    x, y = cuda.grid(2)
```

```
    if x < arr1.shape[0] and y < arr1.shape[1]:
        arr1[x, y] += arr2[x, y]


def make_feedforward_step(act_f):
    @cuda.jit
    def _feedforward_step(inputs, weights, biases, outputs):
        x, batch = cuda.grid(2)
        if x < outputs.shape[0] and batch < outputs.shape[1]:
            outputs[x, batch] = matmul_device(weights, inputs)
            cuda.atomic.add(outputs, (x, batch), biases[x, 0])
            outputs[x, batch] = act_f(outputs[x, batch])
    return _feedforward_step


def make_gradient(dact_f):
    @cuda.jit
    def _gradient(outputs, err, lr, gradients):
        x, batch = cuda.grid(2)
        if x < outputs.shape[0] and batch < outputs.shape[1]:
            gradients[x, batch] = dact_f(outputs[x, batch]) *
err[x, batch] * lr
    return _gradient
```

Программный код нейронной сети.

Файл nn.py:

```
import warnings
from math import exp, tanh, ceil
from time import time

import numpy as np
from numba import cuda, core
```

```
import cuda_kernels as cu_k
from mnist_loader import *

class NeuralNetwork:

    def __init__(self,
                 in_features, hidden_layer_sizes, out_features,
                 dataset: IDataset,
                 thread_per_block=(8, 8),
                 act_f="sigmoid"):

        # Threads per block for cuda
        self.tpb = thread_per_block
        self.af = act_f
        self.in_features = in_features
        self.hidden_layer_sizes = hidden_layer_sizes
        self.out_features = out_features


        self.__init_dataset(dataset)
        self.__init_act_func(act_f)


        scale1 = np.sqrt(1 / in_features)
        scale2 = np.sqrt(1 / hidden_layer_sizes)
        self.weights = [
            cuda.to_device(np.random.uniform(low=-scale1,
high=scale1, size=(hidden_layer_sizes, in_features))),
            cuda.to_device(np.random.uniform(low=-scale2,
high=scale2, size=(out_features, hidden_layer_sizes)))
            ]

        self.biases = [
```

```
            cuda.to_device(np.random.uniform(low=-scale1,
high=scale1, size=(hidden_layer_sizes, 1))),
            cuda.to_device(np.random.uniform(low=-scale2,
high=scale2, size=(out_features, 1)))
            ]


    def __init_act_func(self, act_f):
        act_list = {'sigmoid': {
                        'act':cu_k.sigmoid, 'dact':cu_k.dsigmoid
                            },
                    'relu': {
                        'act':cu_k.relu, 'dact':cu_k.drelu
                        }
                    }
        assert (act_f in act_list)


        cu_k.feedforward_step =
cu_k.make_feedforward_step(act_list[act_f]['act'])
        cu_k.gradient =
cu_k.make_gradient(act_list[act_f]['dact'])
        return


    def __init_dataset(self, dataset: IDataset):


        # Each column of data is a single array of input
        # (e.g. a single image from MNIST)
        self.train_data = dataset.train_data
        self.test_data = dataset.test_data


        # Labels for corresponding data
        self.train_labels = dataset.train_labels
        self.test_labels = dataset.test_labels


        # Desired output for corresponding data based on
self.train_labels
```

```
        self.train_targets = dataset.train_targets

        self.dataset_loaded = False

        return

    def get_grid_dim(self, out_shape, stream=0):
        """Returns cuda grid dimensions needed for kernel
execution"""
        return (ceil(out_shape[0] / self.tpb[0]),
                ceil(out_shape[1] / self.tpb[1])), self.tpb,
stream

    def feedforward_cuda(self, inputs):
        """Returns outputs of NN for each input"""

        inputs_d = cuda.to_device(inputs)
        outputs_d = cuda.device_array((self.weights[0].shape[0],
                                       inputs.shape[1]))

cu_k.feedforward_step[self.get_grid_dim(outputs_d.shape)](inputs
_d, self.weights[0], self.biases[0], outputs_d)
        cuda.synchronize()

        inputs_d = outputs_d
        outputs_d = cuda.device_array((self.weights[1].shape[0],
                                       inputs.shape[1]))

cu_k.feedforward_step[self.get_grid_dim(outputs_d.shape)](inputs
_d, self.weights[1], self.biases[1], outputs_d)
        cuda.synchronize()

        return outputs_d.copy_to_host()
```

```
    def test_accuracy(self):
        """Returns categorical accuracy of NN on
self.test_data"""

        output = self.feedforward_cuda(self.test_data)
        return (output.argmax(axis=0) == self.test_labels[0,
:]).sum() / self.test_data.shape[1] * 100

    def train_accuracy(self):
        """Returns categorical accuracy of NN on
self.train_data"""

        output = self.feedforward_cuda(self.train_data)
        return (output.argmax(axis=0) == self.train_labels[0,
:]).sum() / self.train_data.shape[1] * 100

    def print_accuracy(self):
        train_acc = self.train_accuracy()
        test_acc = self.test_accuracy()

        print(f"Train accuracy: {round(train_acc, 2)}%")
        print(f"Test accuracy: {round(test_acc, 2)}%")

        return train_acc, test_acc

    def backpropogation_cuda(self, inputs, targets, lr, batch,
                             stream1, stream2):
        """
        Implements batch gradient descent.
        Each column of targets is desired output of NN for each
input
        """
        # copy data to device
        inputs_d = cuda.to_device(inputs, stream=stream1) # 784
x batch
```

```
        targets_d = cuda.to_device(targets, stream=stream2) # 10
x batch

        inputs_d_T = cuda.to_device(inputs.T, stream=stream2)


        hidden_d = cuda.device_array((self.hidden_layer_sizes,
batch), stream=stream1) # hidden_layer_sizes x batch


        # forward from IN layer to HIDDEN layer

        # IN --> HIDDEN

        cu_k.feedforward_step[self.get_grid_dim(hidden_d.shape,
stream1)](inputs_d, self.weights[0], self.biases[0], hidden_d)


        # set output from HIDDEN layer to input for OUT layer

        inputs_d = hidden_d

        outputs_d = cuda.device_array((self.out_features,
batch), stream=stream1)


        # forward from HIDDEN layer to OUT layer

        # HIDDEN --> OUT

        cu_k.feedforward_step[self.get_grid_dim(outputs_d.shape,
stream1)](inputs_d, self.weights[1], self.biases[1], outputs_d)


        # create device array tensor

        # 10 x batch

        errors_d = cuda.device_array(targets.shape,
stream=stream1)


        # create tensor for HIDDEN layer derivative

        hidden_errors_d =
cuda.device_array((self.hidden_layer_sizes, batch),
stream=stream2)


        # == > backward step / weights update step < ==


        # compute error

        cu_k.subtract[self.get_grid_dim(errors_d.shape,
stream1)](targets_d, outputs_d, errors_d)
```

```
        stream1.synchronize()


        # errors_h = errors_d


        # self.weights[1].T

        # compute HIDDER error

        cu_k.matmul_T[self.get_grid_dim(hidden_errors_d.shape,
stream2)](self.weights[1], errors_d, hidden_errors_d)

        stream2.synchronize()


        # gradient for HIDDEN

        gradient_d = cuda.device_array(outputs_d.shape,
stream=stream1)

        # compute gradient for HIDDEN layer

        cu_k.gradient[self.get_grid_dim(gradient_d.shape,
stream1)](outputs_d, errors_d, lr, gradient_d)




        # delta for HIDDEN layer bias

        delta_b_d = cuda.device_array(self.biases[1].shape,
stream=stream1)

        # compute delta for HIDDEN layer bias by gradient

        cu_k.sum_cols[ceil(self.biases[1].shape[0] / 4), 4,
stream1](gradient_d, delta_b_d)

        stream1.synchronize()


        # update HIDDEN layer bias by delta

        cu_k.add[self.get_grid_dim(self.biases[1].shape,
stream1)](self.biases[1], delta_b_d)


        # delta for HIDDEN layer weights

        delta_w_d = cuda.device_array((gradient_d.shape[0],
hidden_d.shape[0]), stream=stream2)

        # hidden_d.T

        # compute delta for HIDDEN layer weights by gradient
```

```
        cu_k.matmul_T[self.get_grid_dim(self.weights[1].shape,
stream2)](gradient_d, hidden_d, delta_w_d)


        # update HIDDEN layer weights by delta

        cu_k.add[self.get_grid_dim(self.weights[1].shape,
stream2)](self.weights[1], delta_w_d)


        # gradient for IN layer

        gradient_d = cuda.device_array(hidden_d.shape,
stream=stream2)

        # compute gradient for IN layer

        cu_k.gradient[self.get_grid_dim(gradient_d.shape,
stream2)](hidden_d, hidden_errors_d, lr, gradient_d)


        # delta for IN layer bias

        delta_b_d = cuda.device_array(self.biases[0].shape,
stream=stream2)

        # compute delta for IN layer bias by gradient

        cu_k.sum_cols[ceil(self.biases[0].shape[0] / 4), 4,
stream2](gradient_d, delta_b_d)


        # update IN layer bias by delta

        cu_k.add[self.get_grid_dim(self.biases[0].shape,
stream2)](self.biases[0], delta_b_d)


        # delta for IN layer weights

        delta_w_d = cuda.device_array((gradient_d.shape[0],
inputs.shape[0]), stream=stream1)

        # W * x.T


        # inputs_d_T copied on start

        # delta for IN layer weights

        cu_k.matmul[self.get_grid_dim(self.weights[0].shape,
stream1)](gradient_d, inputs_d_T, delta_w_d)


        # update IN layer weights by delta
```

```
        cu_k.add[self.get_grid_dim(self.weights[0].shape,
stream1)](self.weights[0], delta_w_d)


        errors_h = errors_d.copy_to_host(stream=stream2)
        stream1.synchronize()
        stream2.synchronize()


        return errors_h


    def train(self, epochs: int, learning_rate: float,
batch_size: int):
        """
        Train neural network on a dataset, divided into batches.
        Weights are corrected after each batch
        """
        stream1 = cuda.stream()
        stream2 = cuda.stream()


        lr = learning_rate


        with cuda.pinned(self.train_data),
cuda.pinned(self.train_targets):
            for r in range(epochs):


                epoch_loss = .0
                start_time = time()


                # exponential attenuation
                lr = lr * exp(-r / epochs / 2)


                # random permutation all training dataset
                perm =
np.random.permutation(self.train_targets.shape[1])
                batch_count = ceil(perm.size / batch_size)
                for i in range(batch_count):
```

```
                    pr_bar = ('#' * ((i + 1) * 20 //
batch_count)).ljust(20, ' ')

                    # batch from permutation
                    batch_perm = perm[i * batch_size: (i + 1) *
batch_size]

                    # profiling part
                    with cuda.profiling():
                        loss =
self.backpropogation_cuda(self.train_data[:, batch_perm],
self.train_targets[:, batch_perm],
                                                          lr,
batch_size,
                                                          stream1,
stream2)

                    # time for one batch
                    dur = round(time() - start_time, 1)

                    loss = np.mean(np.abs(loss))
                    epoch_loss += loss

                    # update progress bar
                    print(f"\rEpoch: {r + 1} / {epochs}
[{pr_bar}] {dur}s, loss: {loss:.5f}", end='')

                print()
                print(f'learning rate: {lr:.5f}')
                self.print_accuracy()
                print(f'mean epoch loss:
{epoch_loss/batch_count:.5f}')
                print()
```

```
if __name__ == '__main__':
    # Silence Numba warnings about low occupancy of GPU
    warnings.simplefilter('ignore',

category=core.errors.NumbaPerformanceWarning)

    mnist_dataset = MnistDataSet()
    # fashion_dataset = FashionDataSet()

    dc = NeuralNetwork(784, 512, 10, dataset=mnist_dataset,
                                     act_f='relu',
# relu sigmoid
                                     thread_per_block=(8,8))

    dc.train(20, 0.1, 200)
```

Файл зависимостей для виртуального окружения requirements.txt:

```
cycler==0.11.0
fonttools==4.33.3
gprof2dot==2021.2.21
kiwisolver==1.4.2
llvmlite==0.38.1
matplotlib==3.5.2
numba==0.55.2
numpy==1.22.4
packaging==21.3
pandas==1.4.2
Pillow==9.1.1
pyparsing==3.0.9
python-dateutil==2.8.2
python-mnist==0.7
pytz==2022.1
six==1.16.0
```