Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»**
(МИ ВлГУ)

Факультет Информационных технологий

Кафедра Информационные системы

УТВЕРЖДАЮ

Заведующий кафедрой

_____ Д.Е. Андрианов
(подпись)

«_____» _____ 2021 г.

БАКАЛАВРСКАЯ РАБОТА

Тема Разработка программы имитационного моделирования моделей машинного обучения

МИВУ.09.03.02-05.000 БР

Руководитель

Щаников С.А.
(фамилия, инициалы)

_____
(подпись) (дата)

Студент ИС-117
(группа)

Минеев Р.Р.
(фамилия, инициалы)

_____
(подпись) (дата)

Муром 2021

БЛАНК ЗАДАНИЯ

В рамках данной бакалаврской работы была произведена разработка программы имитационного моделирования моделей машинного обучения. В процессе работы над программой была реализована возможность тестирования любой модели нейронной сети, добавлены алгоритмы имитации аналоговых помех для тестирования работы нейронных сетей, а также повышена производительность работы программы за счет использования метода распараллеливания процессов в целях сокращения времени тестирования.

Табл. 4. Ил. 30. Библ. 11.

As part of this bachelor's work, a program for simulation modeling of machine learning models was developed. In the process of working on the program, the ability to test any model of a neural network was implemented, algorithms for simulating analog noise for testing the operation of neural networks were added, and the performance of the program was increased by using the method of parallelizing processes in order to reduce testing time.

Tab. 4. Fig. 30. Bibl. 11.

Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»**
(МИ ВлГУ)

Факультет Информационных технологий

Кафедра Информационные системы

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Тема: Разработка программы имитационного моделирования моделей машинного обучения

МИВУ.09.03.02-05.000 ПЗ

Муром 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Имитационное моделирование — ключевая технология в работе с ИИ. Оно особенно полезно при создании системы на основе обучения с подкреплением, для которой сложно или невозможно получить входные данные из реального мира.

За последние 5 лет в машинном обучении развивается новый интересный подход - in-memory computing. Применить такой подход стало возможно за счет разработки нового вида оперативной памяти ReRAM, то есть Resitive RAM - памяти, построенной на переменных резисторах. Для оперативной памяти преимущества такой системы заключаются в том, что по скорости она не уступает транзисторной ОЗУ, но при выключении питания вся информация сохраняется. Для машинного обучения применение ReRAM заключается в том, что самую частую операцию взвешивания входного вектора можно выполнять не в цифровом виде на процессоре, как это делается сейчас, а в аналоговом. Каждая ячейка ReRAM выступает в роли делителя напряжения, который изменяет входное напряжение на определенный коэффициент. Она может выполнять умножение весовых коэффициентов в аналоговом виде, а операция умножения является самой частой в машинном обучении. Проблема в таком подходе заключается в том, что аналоговый принцип обработки информации в ReRAM вносит погрешности из-за неизбежного влияния шумов и помех [1]. Этим обусловлена актуальность темы.

Цель данной бакалаврской работы – разработать программу, при помощи которой возможно определить, будут ли на ReRAM готовые модели нейронных сетей работать с приемлемой точностью.

Задачи, поставленные на бакалаврскую работу:

- реализовать возможность тестировать любую модель нейронной сети;

- добавить алгоритмы имитации аналоговых помех для тестирования работы нейронных сетей;

- повысить производительность работы программы за счет использования параллельных вычислений в целях ускорения процесса тестирования;

- сформулировать выводы о работе реализованной программы тестирования.

1 Анализ технического задания

1.1 Для чего необходимо тестировать нейронную сеть.

Создание искусственных нейронных сетей было вдохновлено биологическими аналогами – нейронными сетями живых организмов, которые отлично справляются с решением сложно формализуемых и не формализуемых математически задач, таких как распознавание, классификация, кластеризация, обобщение и т.д. Уоррен Мак-Каллок, который был нейрофизиологом, и Уолтер Питс, который был математиком, предложили собственную модель, описывающую процесс функционирования биологических нейронных сетей в упрощенном виде. Френк Розенблат реализовал эту модель аппаратно в виде первого в мире нейрокомпьютера Марк 1 [2]. Для хранения инструкций использовалась перфорированная лента, а для работы с данными — электромеханические регистры. Это позволяло одновременно пересылать и обрабатывать команды и данные, благодаря чему значительно повышалось общее быстродействие компьютера. Однако такой многообещающий старт развития технологии, в которой компьютер не программируется, а обучается выполнению интеллектуальных задач, был внезапно прерван, после опубликования книги “Персептроны” Марвина Минского в 1970-х годах [3].

Развитие нейрокомпьютеров с тех пор было приостановлено, а теория машинного обучения совершенствовалась лишь в области математического аппарата. В это время активное развитие наблюдалось в области программируемой цифровой техники, результаты, которого мы видим в настоящее время: мобильная электроника, персональные компьютеры, специализированные компьютеры для медицины, военной промышленности и т.п. Вся эта техника функционирует на 2 основных принципах:

- архитектура фон Неймана;

- транзистор, как основной элемент аппаратной реализации.

Совместное использование шины для памяти программ и памяти данных приводит к узкому месту архитектуры фон Неймана, а именно ограничению

пропускной способности между процессором и памятью по сравнению с объемом памяти [4]. Из-за того, что память программ и память данных не могут быть доступны в одно и то же время, пропускная способность канала «процессор-память» и скорость работы памяти существенно ограничивают скорость работы процессора — гораздо сильнее, чем если бы программы и данные хранились в разных местах [5] (рис.1).

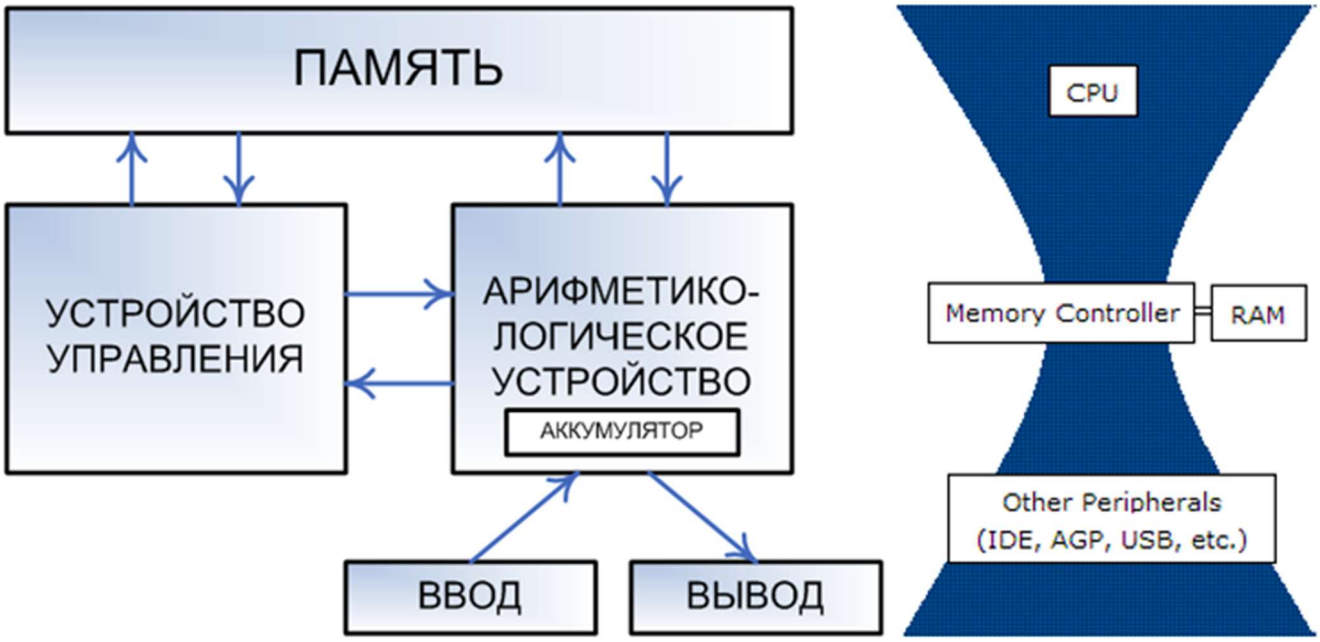


Рисунок 1 – Архитектура Фон Неймана и ее проблема

Применение транзисторов так же имеет проблемы связанные с масштабированием и перегревом. Проблема масштабирования хорошо описывается законом Мура. Закон Мура – эмпирическое наблюдение, изначально сделанное Гордоном Муром, согласно которому (в современной формулировке) количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца (рис.2). В 2007 году Мур заявил, что закон, очевидно, скоро перестанет действовать из-за атомарной природы вещества и ограничения скорости света. Это означает, что в ближайшие годы, для поддержания имеющихся темпов уменьшения размеров транзисторов в процессорах, их размер должен стать меньше размера атома, что физически не возможно. Экспоненциальный рост физических величин в течение длительного

времени невозможен, и постоянно достигаются те или иные пределы. Лишь эволюция транзисторов и технологий их изготовления продлевает действие закона.

Рисунок 2 – Закон Мура

Другая, вытекающая из данного закона проблема – нарастающее энергопотребление и соответственно нагрев цифровых процессоров. За 50 лет энергопотребление выросло на 2 порядка, что влечет за собой не только экономические проблемы, вызванные высокими затратами на обслуживание электросетей, но и экологические, вызванные увеличенным расходом теплоносителей в процессоре (рис.3).

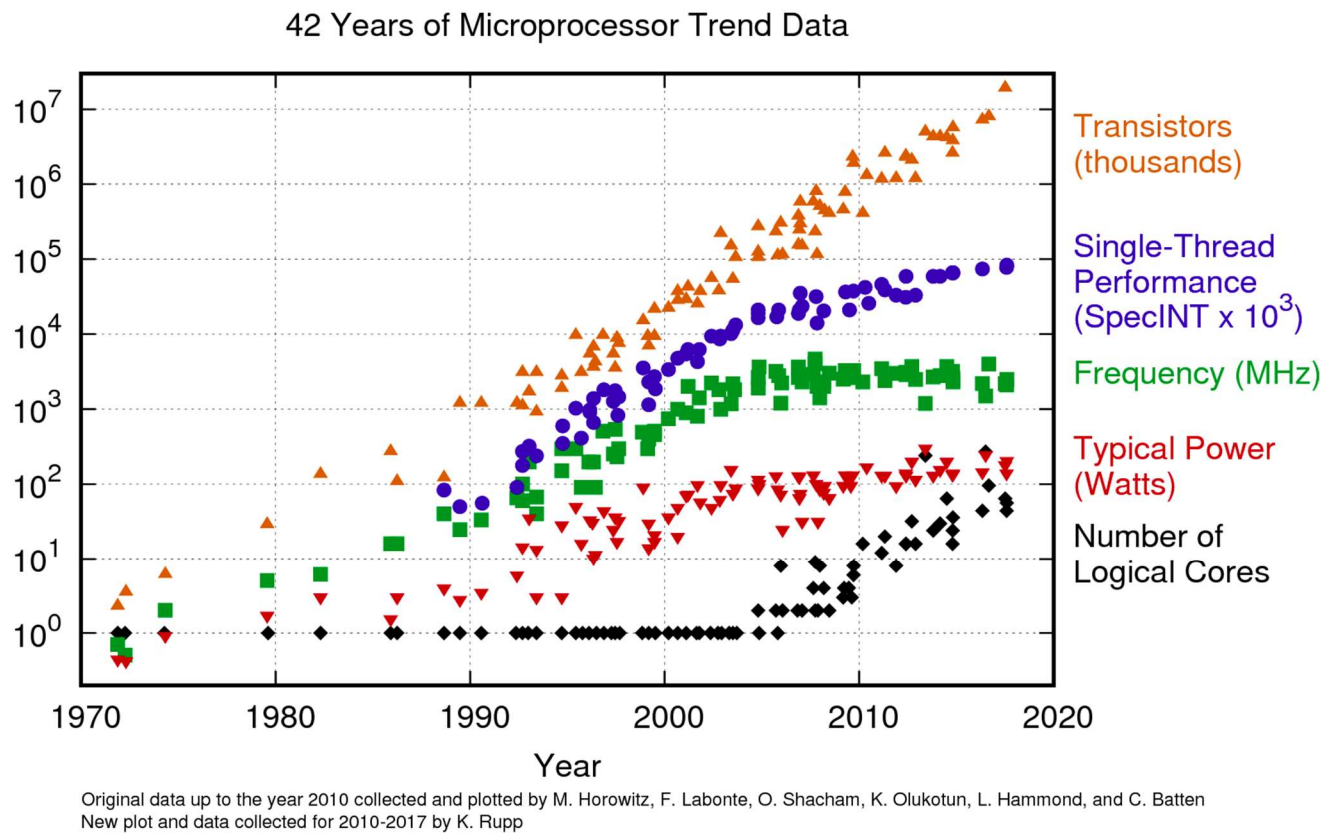


Рисунок 3 – Тренд роста энергопотребления

Альтернативой решения рассмотренных проблем является переход от рассмотренного выше подхода к применению новых парадигм вычисления и новых электронных компонентов. Для этого необходимо сменить архитектуру вычислений и материалы электронных компонентов.

Как было сказано в выше, искусственные нейронные сети были вдохновлены их биологическими аналогами, и по своей природе они являются параллельными средствами обработки информации. Такой параллелизм можно достигнуть за счет применения большого количества простых вычислительных узлов в целом выполняющих одну большую сложную задачу. Поэтому нейросетевая архитектура в данном случае очень подходит на роль альтернативы архитектуре фон Неймана. Для аппаратной реализации искусственных нейронных сетей до недавнего времени не было других аналогов кроме эмуляции их работы на цифровой элементной базе. Однако начиная с 2010-ых годов появилось множество материалов, позволяющих создавать устройства, очень подходящие по своей природе на роль аппаратных нейросетей.

Рассматриваемые материалы – это переменные резисторы на основе тонких пленок различных диэлектриков. Из них делают новые виды оперативной памяти – ReRAM (рис.4). ReRAM очень подходит для аппаратной реализации нейронных сетей, так как отдельная ячейка памяти может выполнять умножение на весовой коэффициент по закону Ома, что значительно увеличит быстродействие и снизит энергопотребление интеллектуальных систем.

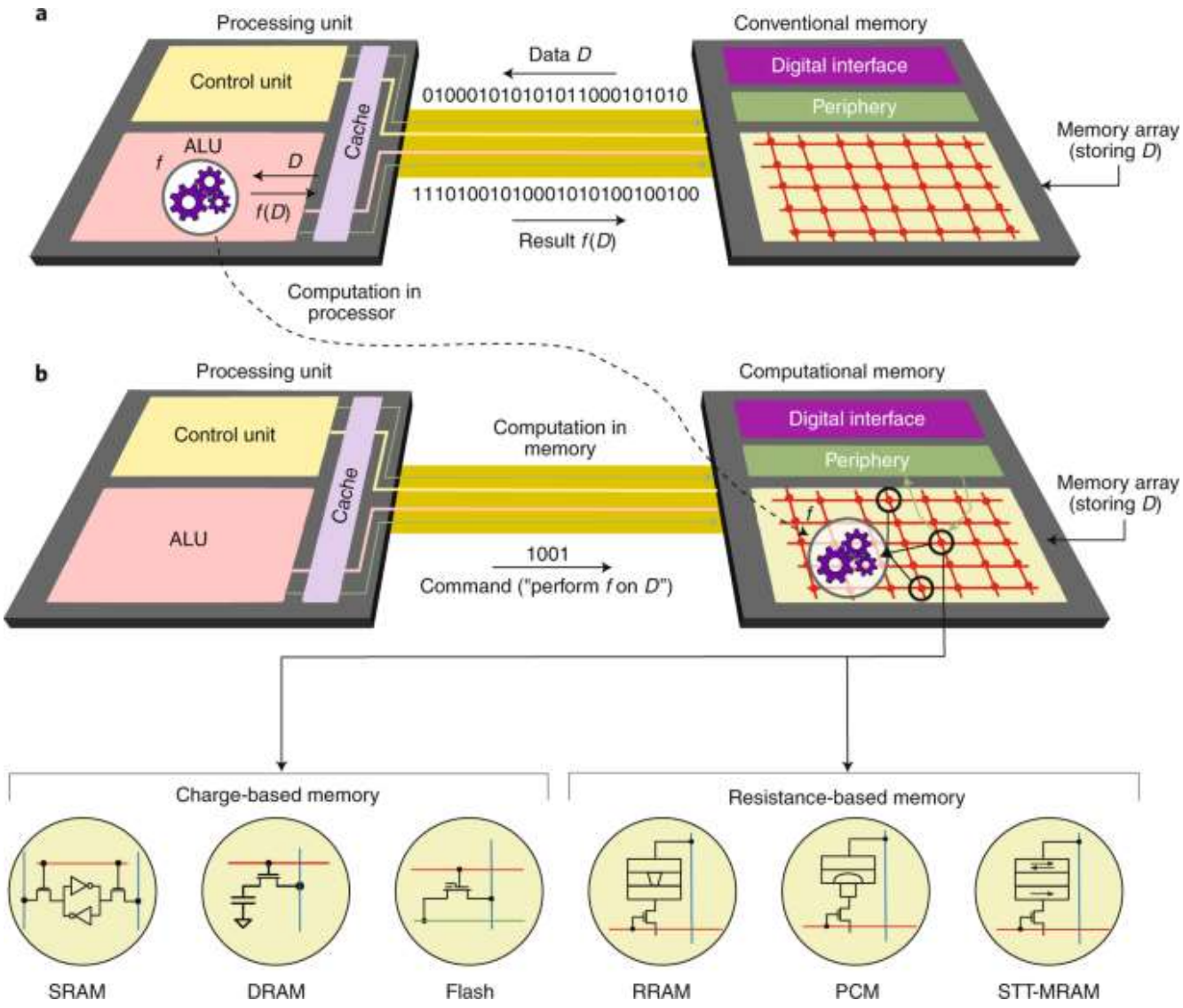


Рисунок 4 – Организация вычислений в ReRAM

Однако достигнутое на этапе компьютерного проектирования номинальное качество работы ИНС снижается в реальных условиях эксплуатации при реализации на ReRAM во многих случаях до полной потери работоспособности из-за аналогового принципа обработки информации. Причиной этого является

неизбежное влияние внутренних и внешних физических и информационных дестабилизирующих факторов, а также производственных и эксплуатационных погрешностей значений параметров элементов и платформы их реализации.

В связи с вышесказанным, актуальными являются исследования, направленные на разработку автоматизированных инженерных методов проектирования высоконадежных ИНС, позволяющих приблизить технические параметры и характеристики ИНС к их потенциально достижимым значениям.

1.2 Обзор методов организации параллельных вычислений

Необходимость разделять вычислительные задачи и выполнять их одновременно (параллельно) возникла задолго до появления первых вычислительных машин.

Существует 3 метода распараллеливания расчетов:

- распараллеливание по задачам (такое распараллеливание актуально для сетевых серверов и других вычислительных систем, выполняющих одновременно несколько функций либо обслуживающих многих пользователей);

- распараллеливание по инструкциям (аппаратно реализовано в современных центральных процессорах общего назначения, поскольку оно эффективно при исполнении программ, интенсивно обменивающихся разнородной информацией с другими программами и с пользователем компьютера);

- распараллеливание по данным [6].

Потоковая обработка данных особенно эффективна для алгоритмов, обладающих следующими свойствами, характерными для задач физического и математического моделирования:

- большая плотность вычислений — велико число арифметических операций, приходящихся на одну операцию ввода-вывода (например, обращение к памяти);

- локальность данных по времени - каждый элемент загружается и обрабатывается за время, малое по отношению к общему времени обработки, после чего он больше не нужен.

Со времени своего появления в начале 1980-х годов, персональные компьютеры развивались в основном как машины для выполнения программ, сложных по внутренней структуре, содержащих большое количество ветвлений, интенсивно взаимодействующих с пользователем, но редко связанных с потоковой обработкой большого количества однотипных данных. Центральные процессоры ПК оптимизировались для решения именно таких задач, поэтому характеризовались следующим:

- большим количеством блоков для управления исполнением программы (кеширование данных, предсказание ветвлений и т.п.) и сравнительно малым количеством блоков для вычислений;

- архитектурой, оптимальной для программ со сложным потоком управления (обработка разнородных команд и данных, организация взаимодействия программ между собой и с пользователем);

- памятью с максимальной скоростью произвольного доступа к данным [7].

Увеличение производительности CPU в основном было связано с увеличением тактовой частоты и размеров высокоскоростной кеш-памяти (память, расположенная прямо на процессоре). Программирование CPU для ресурсоемких научных вычислений подразумевает тщательное структурирование данных и порядка инструкций для эффективного использования всех уровней кеш-памяти.

Ядра современных центральных процессоров являются суперскалярными, поддерживая векторную обработку (расширения SSE и 3DNow!), сами же CPU обычно содержат несколько ядер. Таким образом, в совокупности центральные процессоры могут реализовывать десятки параллельных вычислительных потоков. Однако графические процессоры включают в себя тысячи параллельных «вычислителей». Кроме того, при поточно-параллельных расчетах графические процессоры имеют преимущество благодаря следующим перечисленным особенностям архитектуры:

- память GPU (Graphics Processing Unit) оптимизирована на максимальную пропускную способность (а не на скорость произвольного доступа, как у CPU), что ускоряет загрузку потока данных;

- большая часть транзисторов графического процессора предназначена для вычислений, а не для управления исполнением программы;

- при запросах к памяти, за счет конвейерной обработки данных, не происходит приостановки вычислений.

Однако обработка ветвлений (исполнение операций условного перехода) на GPU менее эффективна, поскольку каждый управляющий блок обслуживает не один, а несколько вычислителей.

Таким образом, производительность одного GPU при хорошо распараллеливаемых вычислениях аналогична кластеру из сотен обычных вычислительных машин, причем графические процессоры сейчас поддерживают практически все операции, используемые в алгоритмах общего назначения:

- математические операции и функции вещественного аргумента;

- организацию циклов;

- операции условного перехода (которые исполняются сравнительно медленно, поскольку в составе GPU блоков управления меньше, чем вычислительных блоков) [7].

В различных источниках информации можно найти много разных определений процессов и потоков. Такой разброс определений обусловлен, в первую очередь, эволюцией операционных систем, которая приводила к изменению понятий о процессах и потоках, во-вторых, различием точек зрения, с которых рассматриваются эти понятия.

С точки зрения пользователя, процесс — экземпляр программы во время выполнения, а потоки — ветви кода, выполняющиеся «параллельно», то есть без предписанного порядка во времени.

С точки зрения операционной системы, процесс — это абстракция, реализованная на уровне операционной системы. Простейшей операционной системе не требуется создание новых процессов, поскольку внутри них работает одна-единственная программа, запускаемая во время включения устройства. В более сложных системах необходимо создавать новые процессы для возможности функционирования нескольких программ.

Процесс — это просто контейнер, в котором находятся ресурсы программы:

- адресное пространство;
- потоки;
- открытые файлы;
- дочерние процессы и т.д.

Также, с точки зрения операционной системы, поток — это абстракция, реализованная на уровне операционной системы. Поток был придуман для контроля выполнения кода программы. Это контейнер, в котором находятся:

- счетчик команд;
- регистры;
- стек.

Поток легче, чем процесс, и создание потока стоит дешевле. Потоки используют адресное пространство процесса, которому они принадлежат, поэтому потоки внутри одного процесса могут обмениваться данными и взаимодействовать с другими потоками.

Поддержка множества потоков внутри одного процесса в рамках работы над разрабатываемой программой необходима. В случае, когда одна программа выполняет множество задач, поддержка потоков в одном процессе позволяет:

- разделить ответственность за разные задачи между разными потоками;
- повысить быстродействие.

Кроме того, часто задачам необходимо обмениваться данными, использовать общие данные или результаты других задач. Такую возможность предоставляют потоки внутри процесса, так как они используют адресное пространство процесса, которому принадлежат. Конечно, можно было бы создать под разные задачи дополнительные процессы, но:

- у процесса будет отдельное адресное пространство и данные, что затруднит взаимодействие частей программы;
- создание и уничтожение процесса дороже, чем создание потока.

Отличие процесса от потока состоит в том, что процесс рассматривается операционной системой, как заявка на все виды ресурсов (память, файлы и пр.),

кроме одного — процессорного времени. Поток — это заявка на процессорное время. Процесс — это всего лишь способ сгруппировать взаимосвязанные данные и выделить область локальной памяти, а потоки — это единицы выполнения, которые выполняются на процессоре.

1.3 Выбор средства

Для разработки программы были выбраны следующие средства:

- язык программирования Python;
- модуль Threading;
- модуль Multiprosessing.

Язык программирования Python был выбран исходя из того, что данный язык часто используется для написания нейросетей.

В случае с разработкой данной системы использование потоков не подойдет, так как Python имеет GIL (Global Interpreter Lock).

GIL – это своеобразная блокировка, позволяющая только одному потоку управлять интерпретатором Python. Это означает, что в любой момент времени будет выполняться только один конкретный поток.

Работа GIL может казаться несущественной для разработчиков, создающих однопоточные программы. Но во многопоточных программах GIL может негативно повлиять на производительность процессоро-зависимых программ.

Модуль Threading впервые был представлен в Python 1.5.2 как продолжение низкоуровневого модуля потоков. Модуль Threading значительно упрощает работу с потоками и позволяет программировать запуск нескольких операций одновременно. Потоки в Python лучше всего работают с операциями ввода/вывода, такими как загрузка ресурсов из интернета или чтение файлов на компьютере.

Модуль Multiprocessing был добавлен в Python версии 2.6 [8]. Изначально он был определен в PEP 371 Джесси Ноллером и Ричардом Одкерком. Модуль Multiprocessing позволяет создавать процессы таким же образом, как при создании потоков при помощи модуля Threading. Использование данного модуля позволяет

обойти GIL и воспользоваться возможностью использования нескольких процессоров на компьютере.

1.4 Требования к разрабатываемой программе

На вход разрабатываемой программе должна поступать модель нейронной сети, а также тестовые значения алгоритма рандомизации данных.

Разрабатываемая программа должна иметь следующие функции:

- функцию загрузки модели из класса программы (а также, описание входных данных тестирования нейронной сети: на чем она обучалась, каким образом загружается тестовая выборка);

- функция определения и вывода временных данных (например, весов нейросетей на всех эпохах обучения);

- функция загрузки полученных данных в исходную модель (таким образом создается новая модель, с новыми данными);

- функция, возвращающая показатели нейросети (точность или расхождение) после проверки модели.

Задачей разрабатываемой программы является повышение скорости расчетов большого количество данных (большого количества тестов, которые будут проводиться для модели нейронной сети). Для этого необходимо равномерно распределить нагрузку по всему процессору.

На выходе пользователь программы должен получать график и возможность сохранения готового ответа, выданного программой. А также необходимо дать пользователю возможность запустить повторное тестирование над той же моделью, если это потребуется, без перезапуска данной программы.

На рисунке 5 представлено различие между последовательной схемой решения (верхняя часть схемы) и способом распределенных вычислений (нижняя часть схемы) [9]. Последовательная схема решения в данном случае не подходит, так как предполагается довольно большой объем обрабатываемых данных, который займет больше времени на обработку из-за низкой эффективности. В данном случае процессор не будет на 100% загружен нашей задачей.

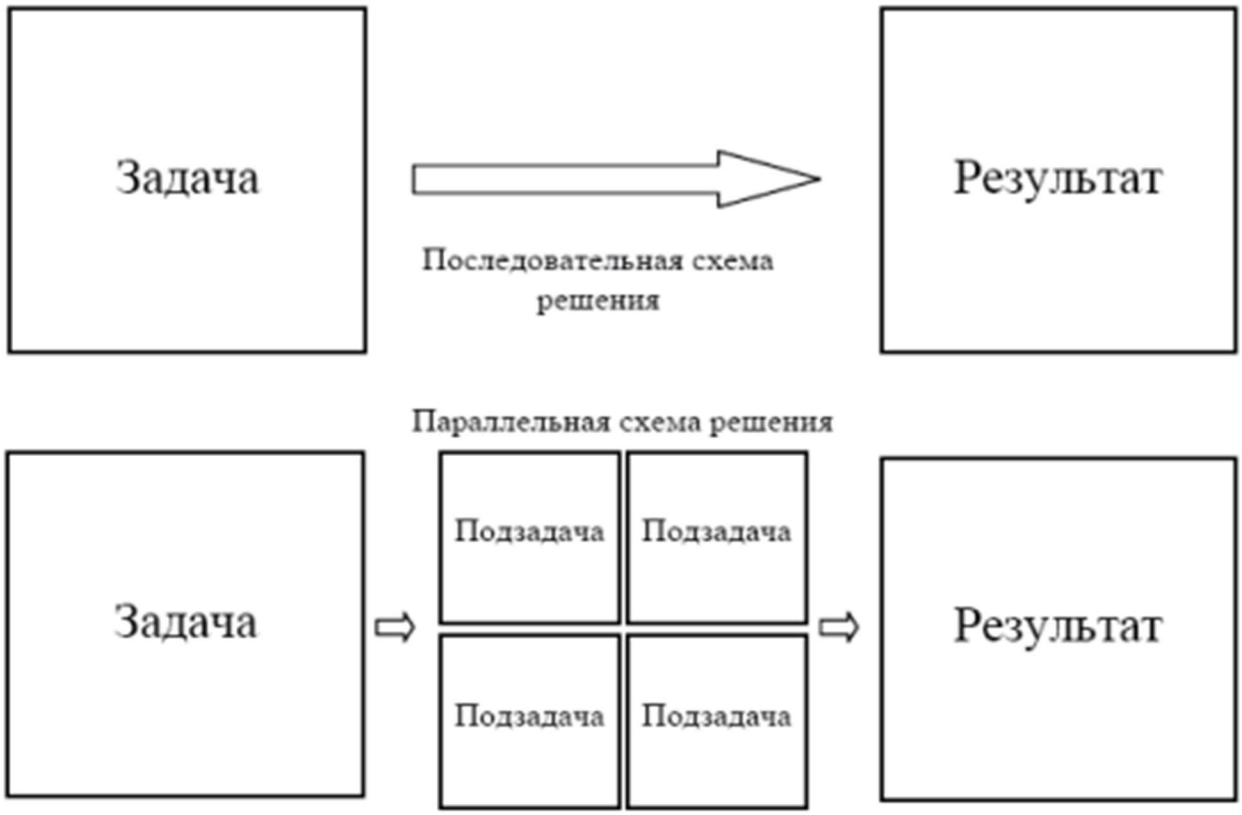


Рисунок 5 - Схема последовательных и распределенных вычислений

Процессор компьютера, на котором производится разработка программы, содержит 12 ядер. В процессе тестирования работы программы будут использоваться все ядра системы. Таким образом, теоретически, производительность программы должна вырасти.

Для генерации случайных значений весов будут использоваться функции нормального распределения и равномерного распределения. Функцию, которая

будет генерировать новые погрешности для весов модели, выбирает пользователь.

Нормальным называется распределение вероятностей, которое для одномерного случая задается функцией Гаусса.

Нормальное распределение случайной величины играет важнейшую роль во многих областях знаний [10]. Случайная величина подчиняется нормальному закону распределения, когда она подвержена влиянию большого числа случайных факторов, что является типичной ситуацией в анализе данных. Поэтому нормальное распределение служит хорошей моделью для многих реальных процессов в мире.

Нормальное распределение зависит от 4-х параметров:

- математическое ожидание - «центр тяжести» распределения;

- дисперсия - степень разброса случайной величины относительно математического ожидания;

- коэффициент асимметрии - параметр формы распределения, определяющий его симметрию относительно математического ожидания.

Типичные формы нормального распределения для различных средних и дисперсии представлены на рисунке 6.

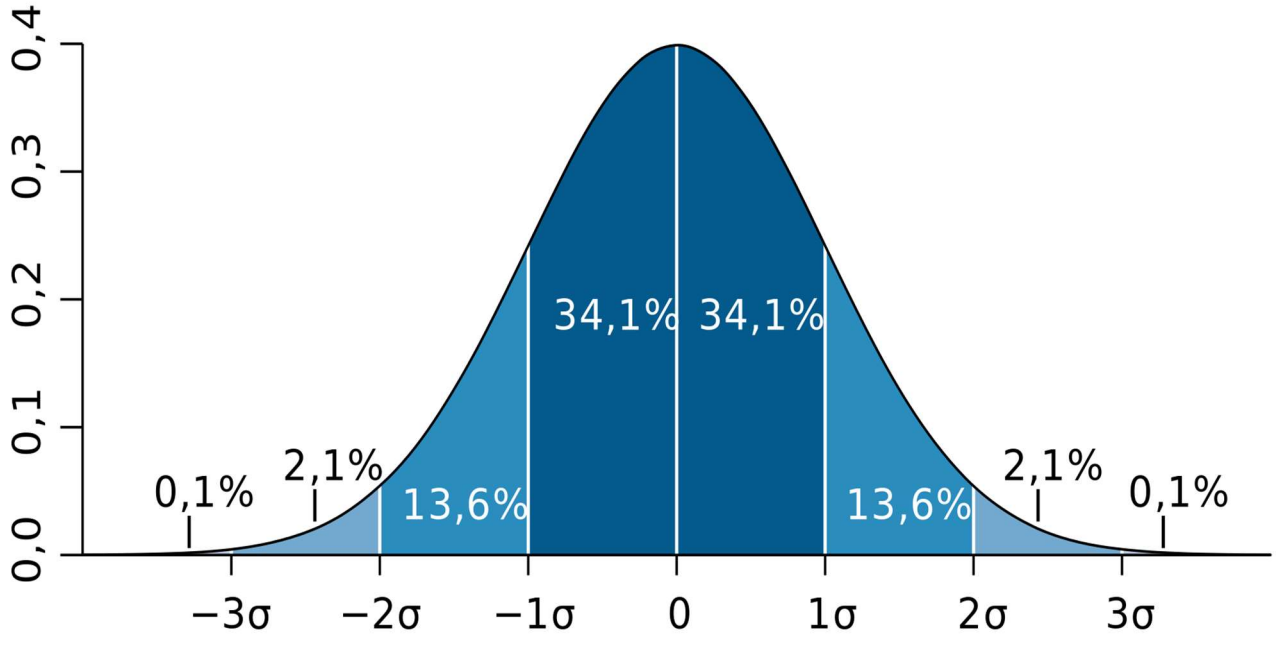


Рисунок 6 - Примеры форм нормального распределения

Формула данной функции – формула (1):

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \quad (1)$$

где μ — математическое ожидание;

σ^2 — дисперсия.

Коэффициент асимметрии определяется по следующей формуле (2).

$$\gamma 1 = \mathrm{E}\left[\left(\frac{x-\mu}{\sigma}\right)^3\right] \quad (2)$$

Если коэффициент асимметрии положителен, то правый «хвост» распределения длиннее левого, и отрицателен в противном случае. Если распределение симметрично относительно математического ожидания, то его коэффициент асимметрии равен нулю.

Коэффициент эксцесса вычисляется по формуле 3:

$$\gamma 2 = \mathrm{E}\left[\left(\frac{x-\mu}{\sigma}\right)^4\right] \quad (3)$$

Равномерным распределением непрерывной случайной величины называется распределение, в котором значения случайной величины с двух сторон ограничены и в границах интервала имеют одинаковую вероятность. Это означает, что в данном интервале плотность вероятности постоянна.

Так как плотность равномерного распределения разрывна в граничных точках отрезка [*a,b*], то функция распределения в этих точках не является дифференцируемой. Существуют также преобразования, позволяющие на основе равномерного распределения получить случайные распределения другого вида.

Таким образом, при равномерном распределении плотность вероятности рассчитывается по формуле (4).

$$f(x)=\begin{cases}\dfrac{1}{b-a}, & x\in[a,b]\\ 0, & x\notin[a,b]\end{cases} \tag{4}$$

Значения $f(x)$ в крайних точках a и b участка $[a,b]$ не указываются, так как вероятность попадания в любую из этих точек для непрерывной случайной величины равна нулю.

Кривая равномерного распределения имеет вид прямоугольника, опирающегося на участок $[a,b]$ (рис.7), в связи с чем равномерное распределение иногда называют «прямоугольным».

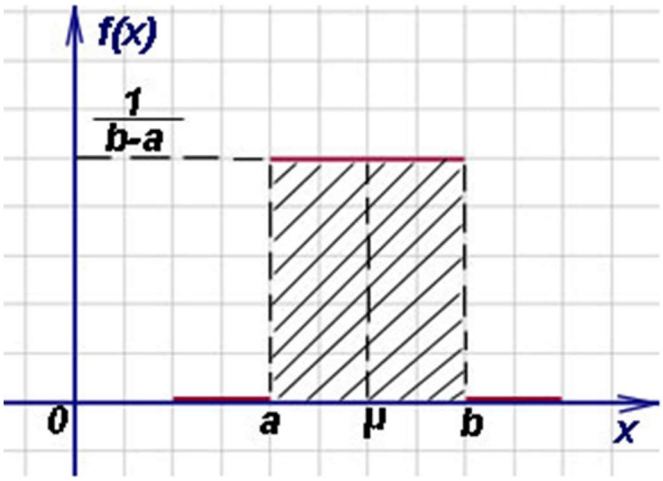


Рисунок 7 - График кривой равномерного распределения

Функция распределения $F(x)$ непрерывной случайной величины при равномерном распределении рассчитывается по формуле (5).

$$F(x)=\begin{cases}0, & x<a\\ \dfrac{x-a}{b-a}, & x\in[a,b]\\ 1, & x>b\end{cases} \tag{5}$$

Характеристиками равномерного распределения являются:

- среднее значение (математическое ожидание) по формуле (6):

$$\mu = \frac{a+b}{2}; \tag{6}$$

- дисперсия по формуле (7):

$$\sigma^2 = \frac{(b-a)^2}{12}; \tag{7}$$

- стандартное отклонение по формуле (8):

$$\sigma = \frac{(b-a)}{12}. \tag{8}$$

Описав алгоритмы, которыми будут обрабатываться веса нейронной сети, имитируя погрешность аналогового сигнала, можно перейти к проектированию общей схемы работы программы (рис.8) [11].

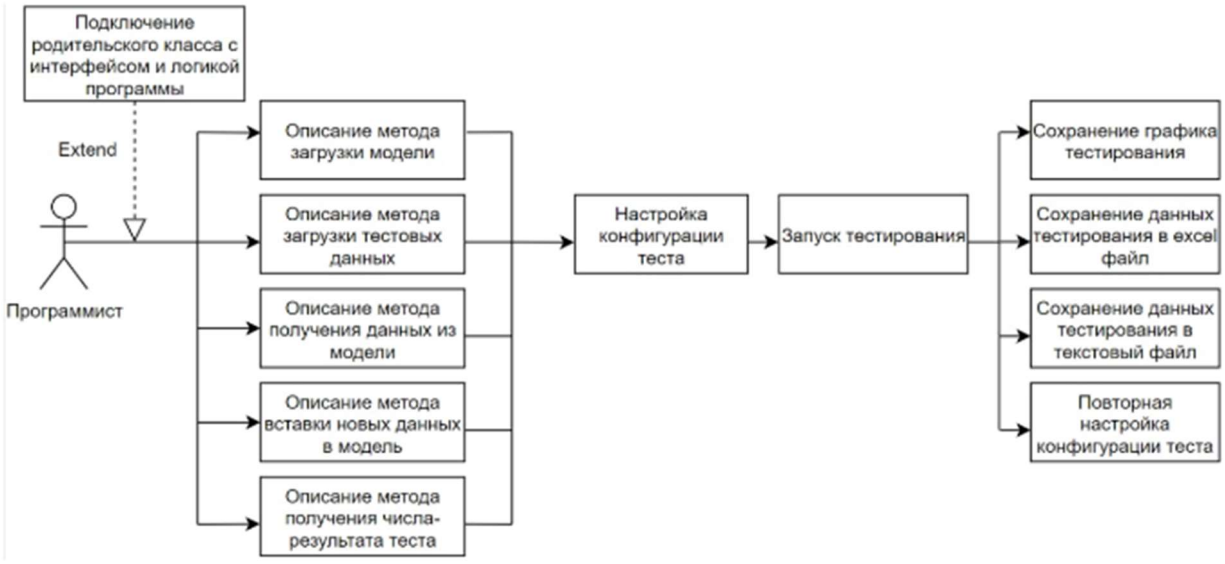


Рисунок 8 - Схема работы программы

Из рисунка 8 видно, что пользователь программы – это программист, перед которым стоит задача написать свой класс, унаследованный от родительского класса, в разрабатываемой программе. Работа с программой начинается с подключения родительского класса с интерфейсом и логикой программы. В данном случае это файл с расширением .py и несколько библиотек, для отрисовки интерфейса и многопроцессорной обработки. Для работы программы пользователю необходимо выполнить следующие действия:

- описывает метод загрузки модели (загрузка обученной модели для последующего тестирования);

- описывает метод загрузки тестовых данных (либо загружает готовые данные (изображения и т.п.), либо описывает код-генератор тестовых данных);

- описывает метод получения данных из модели (данные (веса и пр.), которые будут изменяться в зависимости от выбранного алгоритма рандомизации данных);

- описывает метод загрузки новых данных в модель;

- описывает метод получения числа – результата теста (вывод точности распознавания, расчет ошибки при распознавании или др.).

Далее происходит настройка конфигурации теста, в процессе которой пользователь задает следующие параметры:

- количество ядер, которые будут задействованы в процессе работы программы (по умолчанию выбрано максимальное количество);

- алгоритм обработки данных (нормальное распределение случайной величины, равномерное распределение);

- список значений для алгоритма генерации случайных чисел.

Генерация последнего параметра происходит при помощи 4-х данных:

- начального значения (по умолчанию - 0);

- конечного значения;

- шага;

- количества тестов на каждом шаге (в процессе тестирования полученные данные усредняются для отображения наиболее приближенного).

Далее происходит сам процесс тестирования, после чего на экране

программы отображается график, отражающий результат тестирования.

После этого пользователь может сделать следующее:

- сохранить на устройство полученный в результате тестирования график в виде изображения (.png, .jpg и т.д.);

- сохранить полученные данные тестирования в файл .xlsx или .txt;

- повторно настроить конфигурацию теста.

На рисунке 9 приведена диаграмма последовательности, которая описывает последовательность работы пользователя в программе.

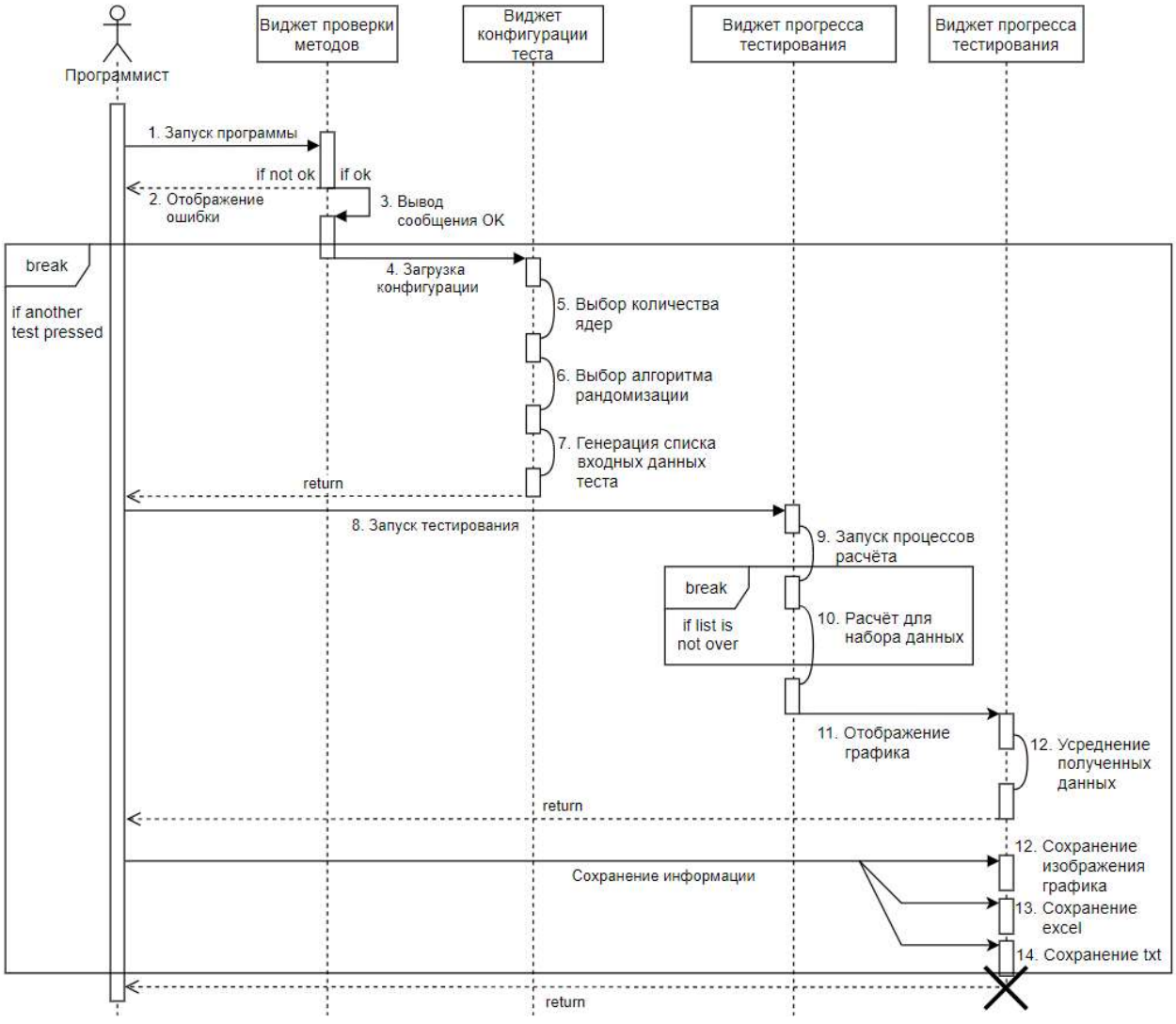


Рисунок 9 - Диаграмма последовательности

На данном этапе процесс проектирования можно считать завершенным. Следующий этап – разработка программы.

3 Разработка программы

В данном разделе пояснительной записки описан процесс разработки программы имитационного моделирования моделей машинного обучения.

Проверка функций, которые необходимо описать пользователю, производится следующим образом:

- метод load_model не должен содержать синтаксических ошибок и должен возвращать объект модели, которую пользователю необходимо протестировать;

- метод load_testdata должен вернуть переменную, которая является списком и хранит в себе элемент, являющийся тестовой выборкой, и элемент, хранящий правильное поведение модели на каждый из элементов тестовой выборки;

- метод get_tested_values содержит описание процесса выгрузки значений весов из модели, которую загрузил пользователь в методе load_model;

- метод set_tested_values загружает новые веса, которые были получены в методе get_tested_values, в модель и вернет ошибку если не было загружено данных, либо метод get_tested_values не вернул ожидаемые функцией set_tested_values переменные;

- методу test_model необходимо взаимодействовать со всеми описанными ранее функциями, поэтому если на одном из них произошла ошибка, то и на данном методе возникнет ошибка.

Код проверки модуля load_model представлен ниже.

```
try:
    if self.name == 'load_model':
        self.neuralconfig.load_model()
        try:
            pickle.dumps(self.neuralconfig.model)
        except TypeError:
            raise PickleProblem
except PickleProblem:
    self.function_checked_signal.emit(self.name, 'ERROR')
    self.error_signal.emit('Your model is not pickleable.\nTry to
```

```
pickle your model before launch the program', self.name)
     except Exception as e:
         self.function_checked_signal.emit(self.name, 'ERROR')
         self.error_signal.emit(str(e), self.name)
     self.return_new_state.emit(self.neuralconfig)
             self.done.emit()
```

Как можно увидеть из данного кода программы, данная функция выведет сообщение об ошибке, если:

- метод load_model не описан (его не существует);

- в коде этого метода обнаружена синтаксическая ошибка;

- модель нельзя представить в байтовом виде.

Модуль Multiprocessing (стандартная библиотека Python) выполняет функцию распределения полученных данных на процессы, поэтому все данные, которые нужны для выполнения функции, должны иметь возможность быть представлены в байтовом виде. Таким образом, если модель, загруженная пользователем, не может быть переведена в байтовый вид, пользователь увидит сообщение об ошибке.

Далее проверяется функция load_testdata. Если метода нет, либо он содержит синтаксическую ошибку, выводится сообщение об ошибке. В данном методе должна быть описана переменная self.testdata. Если она будет описана неправильно, сообщение об ошибке выведет проверка метода test_model.

Следующая на очереди – проверка метода get_tested_values, которая также вернет ошибку при наличии синтаксической ошибки либо отсутствии данного метода, которое повлечет за собой ошибку следующего метода.

После этого происходит проверка метода set_tested_values. Программа уведомит пользователя об ошибке, если метод не сможет подставить в модель данные, которые вернул метод get_tested_values.

Последняя проверка - работа метода test_model. Его задача - подставить модель из метода set_tested_values и данных из метода load_testdata, и вернуть ошибку, если на каком-либо из этих этапов произойдет сбой. А также этот метод

должен вернуть одно число, обозначающее точность модели на заданной выборке.

Если ошибок не обнаружено, происходит загрузка виджета настройки конфигурации тестов. Пользователь выбирает количество используемых ядер и алгоритм распределения данных, а также задает настройки списка значений для генерации ряда случайных чисел: начальное и конечное значения, шаг и число, отражающее количество тестов на каждом шаге.

Код генерации ряда случайных чисел следующий.

```
    def update_config(self):
            numbers = [edit.text() for edit in self.line_edits]
            try:
                start = float(numbers[0])
                stop = float(numbers[1])
                step = float(numbers[2])
                ntes = int(numbers[3])
            except ValueError:
                self.setError('Wrong numbers')
                return
            self.config = np.arange(start, stop, step)
            self.config = np.around(self.config, 6)
            if len(self.config) > 8:
                self.preview.setText('[' + ', '.join(list(map(str,
self.config[:3]))) + ', ..., ' + ', '.join(list(map(str, self.config[-
3:]))) + '] | ' + str(len(self.config)) + ' values')
            else:
                self.preview.setText('[' + ', '.join(list(map(str,
self.config))) + ']')
            self.config_changed.emit()
```

На данном этапе происходит проверка введенных пользователем данных. Возможно возникновение следующих ошибок:

- введены не числа;

- конечное число ряда меньше, чем начальное;

- шаг меньше или равен 0;

- количество тестов меньше 1.

Хранением и выполнением алгоритмов генерации случайных чисел занимается класс GeneratingRandomAlgorithms. Его код представлен ниже.

```
class GeneratingRandomAlgorithms():
    names = ['normal', 'uniform',]

    def generate(self, weight, value):
        if self.algorithm == 'normal':
            return self._random_numpy_normal(weight, value)

        elif self.algorithm == 'uniform':
            return self._random_numpy_uniform(weight, value)

    def _random_numpy_normal(self, weight, sigma):
        return [np.random.normal(array, sigma) for array in
weight]
    def _random_numpy_uniform(self, weight, deviation):
        def generate_number(number):
            return number + np.random.uniform(-deviation,
deviation, size=1)
        return [generate_number(array) for array in weight]
```

В зависимости от выбранного алгоритма (Normal – нормальное распределение, Uniform – равномерно распределение), будет сохранена информация о названии параметра данного алгоритма (sigma, deviation соответственно) и выполнен соответствующий метод (_random_numpy_normal, _random_numpy_uniform соответственно).

Далее пользователь запускает процесс тестирования, что вызывает загрузку виджета прогресса тестирования и старт самого тестирования. Происходит процесс передачи конфигурации данных в класс MultiprocessExecution. Далее вызывается метод multiprocessingCalc, который поделит все входные данные на одинаковые по

размеру списки для каждого процесса и запустит параллельное вычисление для каждого из них и выделит локальную область памяти. Код данного класса представлен ниже.

```
    class MultiprocessExecution():
        def __init__(self, data, neuralconfig, shm_name):
            self.data = data
            self.neuralconfig = neuralconfig
            self.shm_name = shm_name
        def multiprocessingCalc(self):
            values = self.data['config']
            cpu_count = self.data['processors']
            process_step = ceil(len(values) / cpu_count)
            list_in_data = list()
            for i in range(ceil(len(values) / process_step)):
                end = i * process_step + process_step if i + 
process_step <= len(values) else len(values)
                start = i * process_step
                list_in_data.append(values[start:end])
            with 
multiprocessing.get_context("spawn").Pool(cpu_count) as p:
                data = p.map(self._compute_models, list_in_data)
            return data
```

Поступившие данные конфигурации делятся на части, равные количеству ядер, которые были заданы пользователем, с последующим запуском функции _compute_models для каждой из этих частей. Каждый раз, когда функция _compute_models выполняет расчет для новой модели, она прибавляет 1 к общему прогрессу рассчитанных моделей, что в последующем можно использовать для отображения прогресса пользователю. Для этого используется класс ShareableList библиотеки Multiprocessing. Он позволяет иметь доступ к одной ячейке памяти для всех процессов, которые в данный момент выполняются программой и пользоваться этой областью без прерывания основного потока.

Каждый процесс для каждого числа берет следующее число из списка данных, выполняет функцию генерации новых значений модели, тем самым создавая новую модель. Происходит расчет модели на тестовых данных, заданных пользователем, и сохранение пары «значение-результат».

После окончания работы всех процессов все данные собираются в общий список и передаются в виджет отображения результатов тестирования.

При создании данного виджета происходит сбор и расчет данных для построения графика – результата тестирования. Код расчета данных для построения графика следующий.

```
def getXY_values(self):
        XY_data = {}
        for lst in self.data:
            for sigma_r in lst:
                if sigma_r [0] in XY_data:
                   XY_data[sigma_r [0]].append(sigma_r [1])
                else:
                    XY_data[sigma_r [0]] = [sigma_r [1]]
            del(sigma_r)
        for key in XY_data.keys():
            XY_data[key] = np.average(XY_data[key])
        self.data = XY_data
```

В случаях, когда пользователем было указано количество тестов больше одного, при расчете координат точек графика, каждый полученный результат в рамках одного диапазона данных усредняется при помощи операции нахождения среднего арифметического функцией, описанной в библиотеке numpy.

Отображение полученного графика в окне программы реализовано средствами библиотеки Matplotlib.

```
def getGraph(self):
        matplotlib.use('Qt5Agg')
```

```
            sc = MplCanvas(self, width=5, height=5, dpi=100)
            sc.axes.plot(list(self.data.keys()),
list(self.data.values()))
            sc.axes.set_xlabel( \
            self.algorithm.value_name.capitalize()  +  '  (numpy.'  +
self.algorithm.algorithm + ')')
            sc.axes.set_ylabel('Test results')
            sc.axes.set_title(self.name + ' | ' + self.test_time)
            toolbar = NavigationToolbar2QT(sc, self)
            sc.fig.tight_layout()
            lst = toolbar, sc
            return lst
```

В данном коде происходит создание графика. Параллельно оси X будет отображаться название алгоритма, при помощи которого была осуществлена генерация случайных чисел с соответствующим названием параметра этого алгоритма. Название графика отображается вверху графика и представляет собой название модели, заданное пользователем, а также включает в себя дату и время окончания тестирования.

Далее пользователь может либо сохранить график в виде изображения, либо вывести данные графика в текстовом виде или в формате .xlsx, либо продолжить работу с программой с другой конфигурацией теста. Сохранение в формат .xlsx реализовано при помощи библиотеки Openpyxl. Данная библиотека также распространяется свободно и способна сохранять таблицы включая самую последнюю версию Excel. При сохранении в этот формат в файл сохраняются следующие данные:

- название модели, заданное пользователем (если название не задано, по умолчанию модели будет присвоено название «NoName»);

- дата и время окончания тестирования;

- значение, которое вернул метод test_model, на неизмененной модели, загруженной методом load_model, и тестовых данных, загруженных методом load_testdata ранее;

- выбранный пользователем алгоритм распределения данных;

- список данных, полученных в результате выполнения тестирования, в виде пары «значение-результат».

Код данного метода представлен ниже.

```
    ws.cell(row=1, column=1).value = 'Name of model: ' + self._data['name']
    ws.cell(row=2, column=1).value = 'Tested: ' + self._data['time']
    ws.cell(row=3, column=1).value = 'Your model tested value: ' + str(self._data['test_value'])
    ws.cell(row=4, column=1).value = 'Generating random algorithm: numpy.' + self._data['algorithm'].algorithm
    ws.cell(row=6, column=1).value = self._data['algorithm'].value_name.capitalize()
    ws.cell(row=6, column=2).value = 'Tested value'
```

Из кода выше видно, что происходит добавление в файл xlsx, которым является переменная ws. В каждую строку записываются данные из переменной data, которая была посчитана с помощью класса MultiprocessExecution.

Аналогичным образом производится сохранение результатов тестирования в текстовый файл. Для этого используется стандартный метод open языка python, который способен создавать файлы любого расширения и свободно редактировать информацию в них.

Далее пользователю предоставляется возможность выполнить повторное тестирование с другой конфигурацией над той же моделью.

4 Тестирование

4.1 Тест общей работы модуля

Для тестирования работы программы используются методы библиотеки Tensorflow. Была создана модель – классификатор стандартного датасета библиотеки Tensorflow – Fashion-MNIST.

Работа начинается с подключения разработанного модуля и описания дочернего от этого модуля класса. Тест-кейсом будет проверка загрузки ранее созданной модели Fashion-MNIST.

```
class NeuralCrashTest(NeuralCrash):
    def __init__(self):
        super().__init__()
    def load_model(self):
        self.model = load_model('../2 II Creation/fashion_mnist.h5')

    def load_testdata(self):
        (_, _), (testX, testY) = fashion_mnist.load_data()
        testX = testX.reshape(testX.shape[0], 784) / 255
        self.testdata = testX, testY

    def get_tested_values(self):
        return self.model.get_weights()

    def set_tested_values(self, values):
        new_model = self.model
        new_model.set_weights(values)
        return new_model

    def test_model(self, model, data):
        results = model.evaluate(data[0], data[1], batch_size=64)
        return results[-1]
```

В данном коде происходит создание дочернего класса NeuralCrashTest от родительского класса NeuralCrash. В данном классе описаны пять методов, требующиеся для корректной работы программы, и один метод инициализирующий код создания объекта класса NeuralCrash.

После вызова метода run у родительского класса происходит создание интерфейса с проверкой написанных ранее методов.

Попробуем специально исказить написанные методы, чтобы произошло событие отображения ошибок.

```
def load_model(self):
    self.model = load_model('../2 II Creation/fashion_mnist.h5')
```

В вышеприведенном коде попробуем заменить путь к модели на следующий.

```
def load_model(self):
    self.model = load_model('../2 II Creation/fashion_mnist_notModel.h5')
```

Класс проверки корректно обработал ошибку, что отображено на рисунке 10.

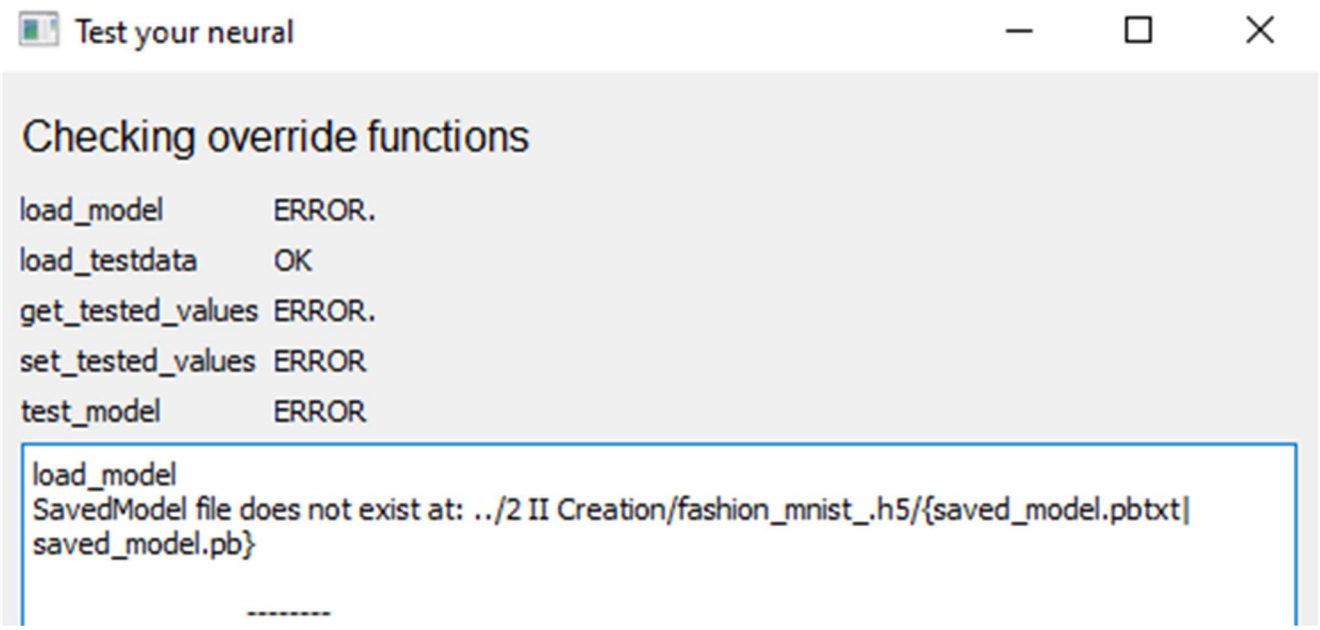


Рисунок 10 - Виджет обработки ошибок

Данное окно (рис.10) показывает, что пользователь совершил ошибку в момент описания метода load_model, что повело за собой появление ошибок и в других методах, которые прямо зависят от load_model (get_tested_values, set_tested_values, test_model).

Далее необходимо проверить еще один важный момент. Модель должна иметь возможность быть приведена в двоичный вид библиотекой pickle. Если данной возможности нет, то данный виджет обработки ошибок также предупредит об этом пользователя. Это отображено на рисунке 11.

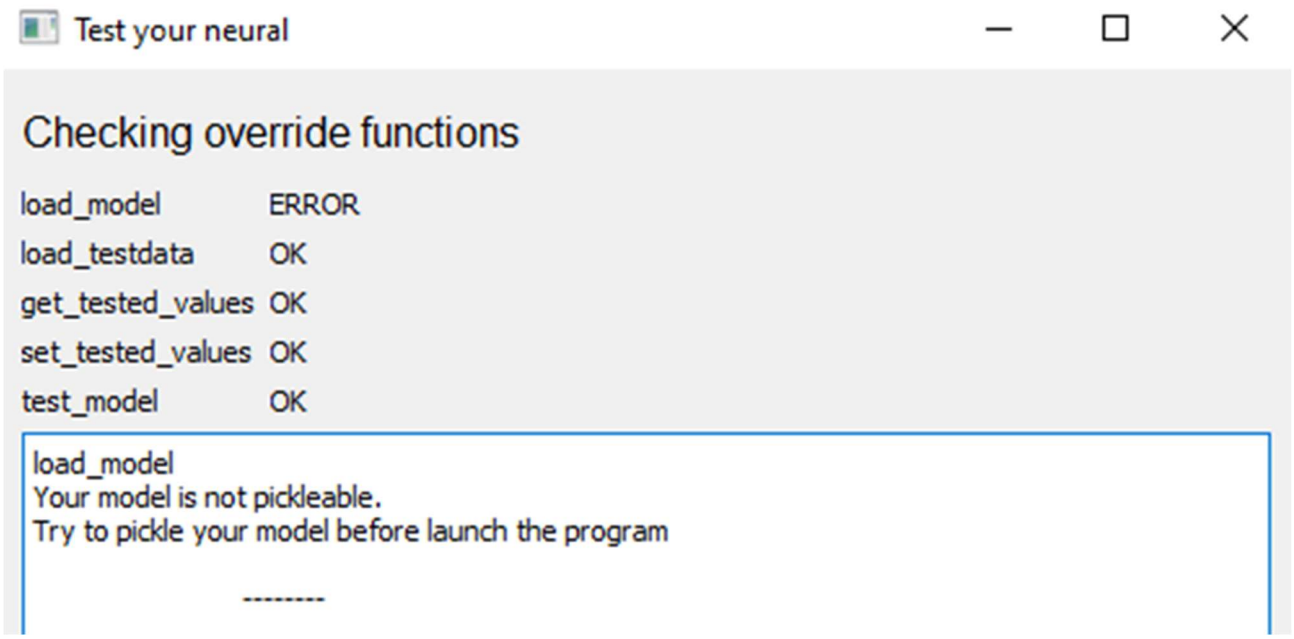


Рисунок 11 - Обработка ошибки метода pickle

В данном случае, чтобы исправить данную ошибку, необходимо для библиотеки tensorflow, модель которого мы пытаемся загрузить, переописать методы deserialize и serialize. Это приведено в коде ниже.

```
def unpack(model, training_config, weights):
  restored_model = deserialize(model)
  if training_config is not None:
    restored_model.compile(
      **saving_utils.compile_args_from_training_config
  restored_model.set_weights(weights)
```

```
    return restored_model
  def make_keras_picklable():
    def __reduce__(self):
      model_metadata = saving_utils.model_metadata(self)
      training_config=model_metadata.get("training_config", None)
      model, weights = serialize(self), self.get_weights()
      return (unpack, (model, training_config, weights))
    cls, cls.__reduce__ = Model, reduce__
```

После вызова данного кода, модель стало возможно использовать в программе. После успешной проверки всех остальных методов происходит отображения виджета настройки конфигурации теста. Его внешний вид показан ниже на рисунке 12.

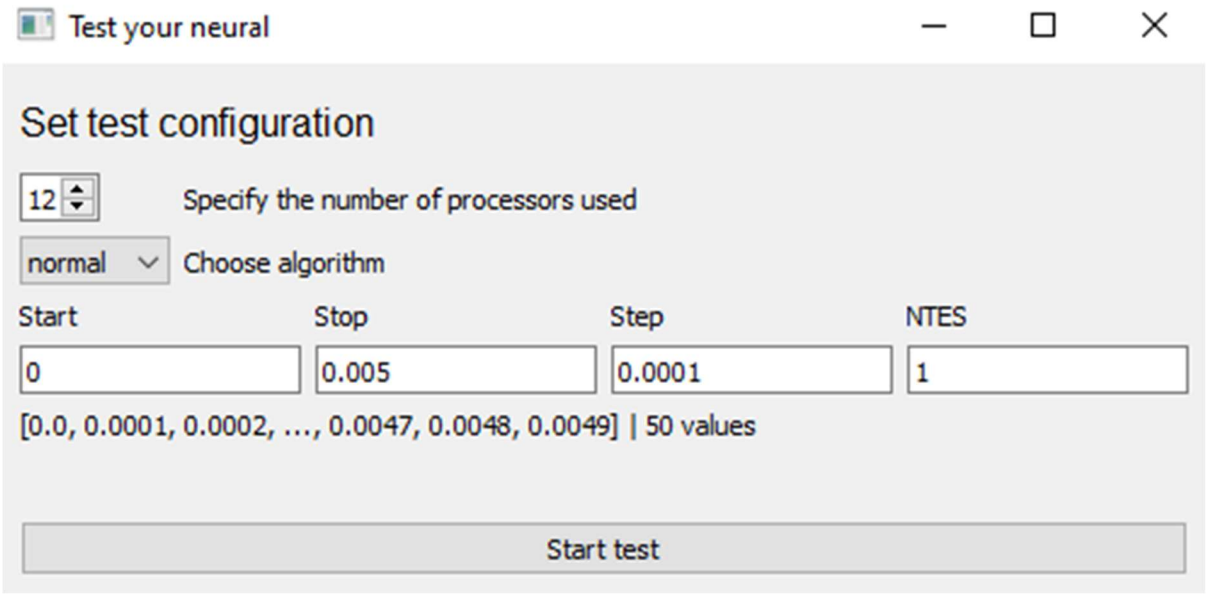


Рисунок 12 - Виджет настройки конфигурации теста

При создании данного виджета (рис.12) во все элементы управления расставляются значения по умолчанию:

- количество ядер процессора равно максимальному значению, определенного системой;

- алгоритм рандомизации данных - normal;

- старт, стоп, шаг и количество тестов - 0, 0.005, 0.0001, 1 соответственно.

Каждое изменение значений списка данных для теста повлечет за собой обработку события update_config, которое проверит возможность создания списка с данными параметрами, а также отобразит ошибку в противном случае.

Примеры ошибок, которые может совершить пользователь при конфигурации списка, представлены на рисунке 13.

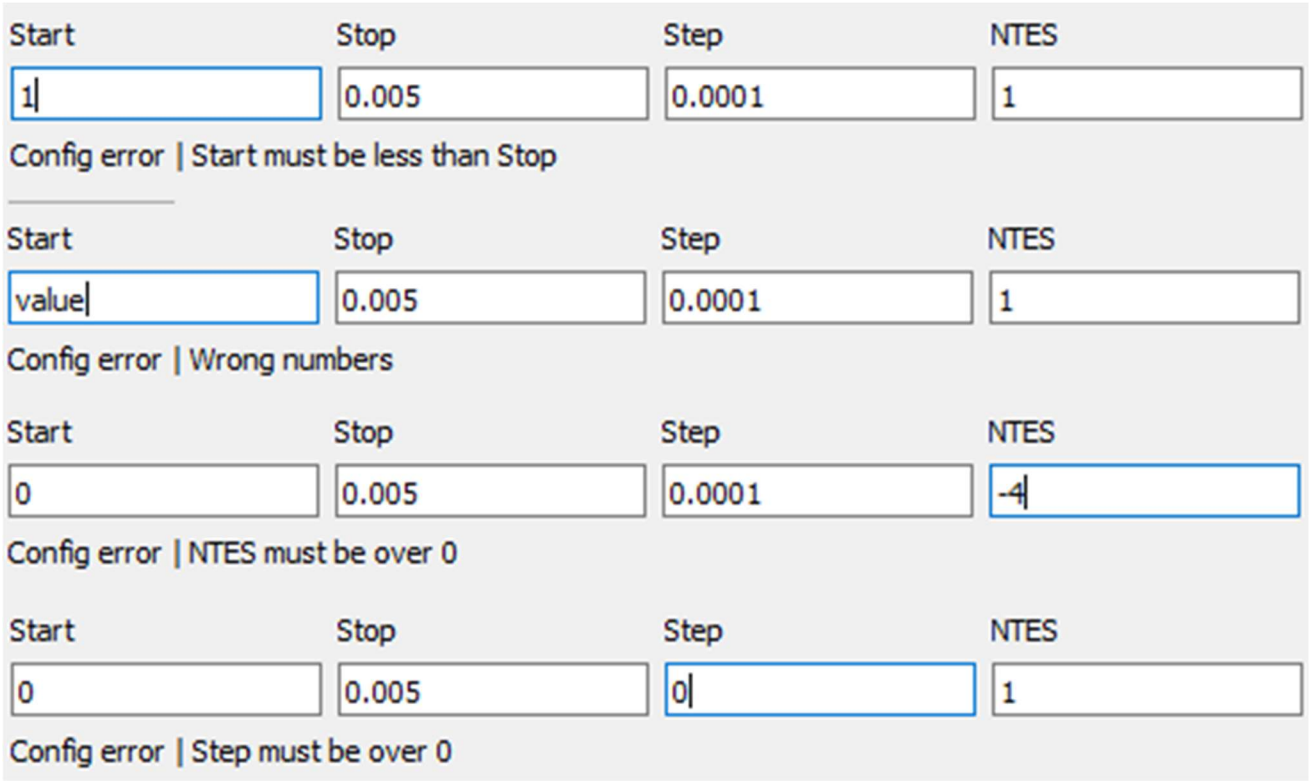


Рисунок 13 - Обработка ошибок при конфигурации списка входных данных теста

Далее происходит процесс запуска теста, где задача разбивается на подзадачи и отправляется на выполнение каждая в свой процесс. Количество таких подзадач пользователь задал в элементе управления для выбора количества ядер процессора.

Пользователю отображается виджет отображения прогресса тестирования (рис.14) с процентным индикатором выполнения общей задачи, а также примерное время до конца тестирования. Это время в начале будет иметь непостоянное значение, так как не будет достаточно данных для проведения точного анализа времени выполнения одной задачи. С каждым запросом на новое отображение прогресса считается новое время выполнения задачи и производится расчет оставшегося времени до конца тестирования.

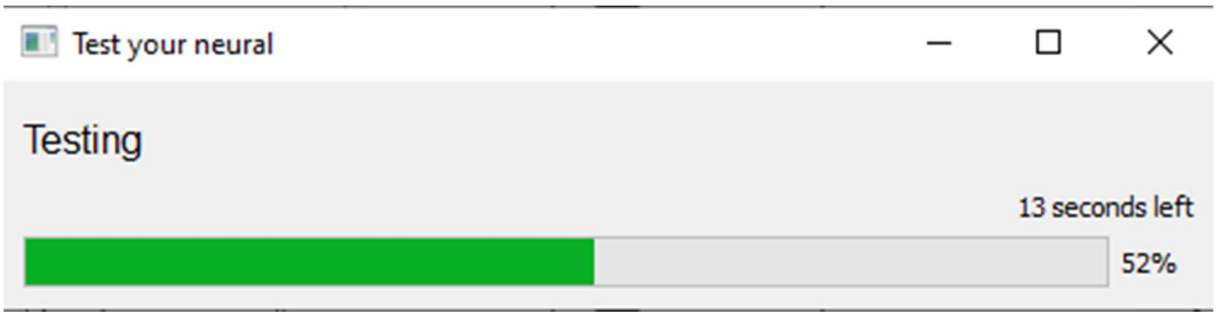


Рисунок 14 - виджет отображения прогресса тестирования

Работу программы на данном этапе можно отследить отладчиком кода (рис.15), либо с помощью диспетчера задач Windows (рис.16).

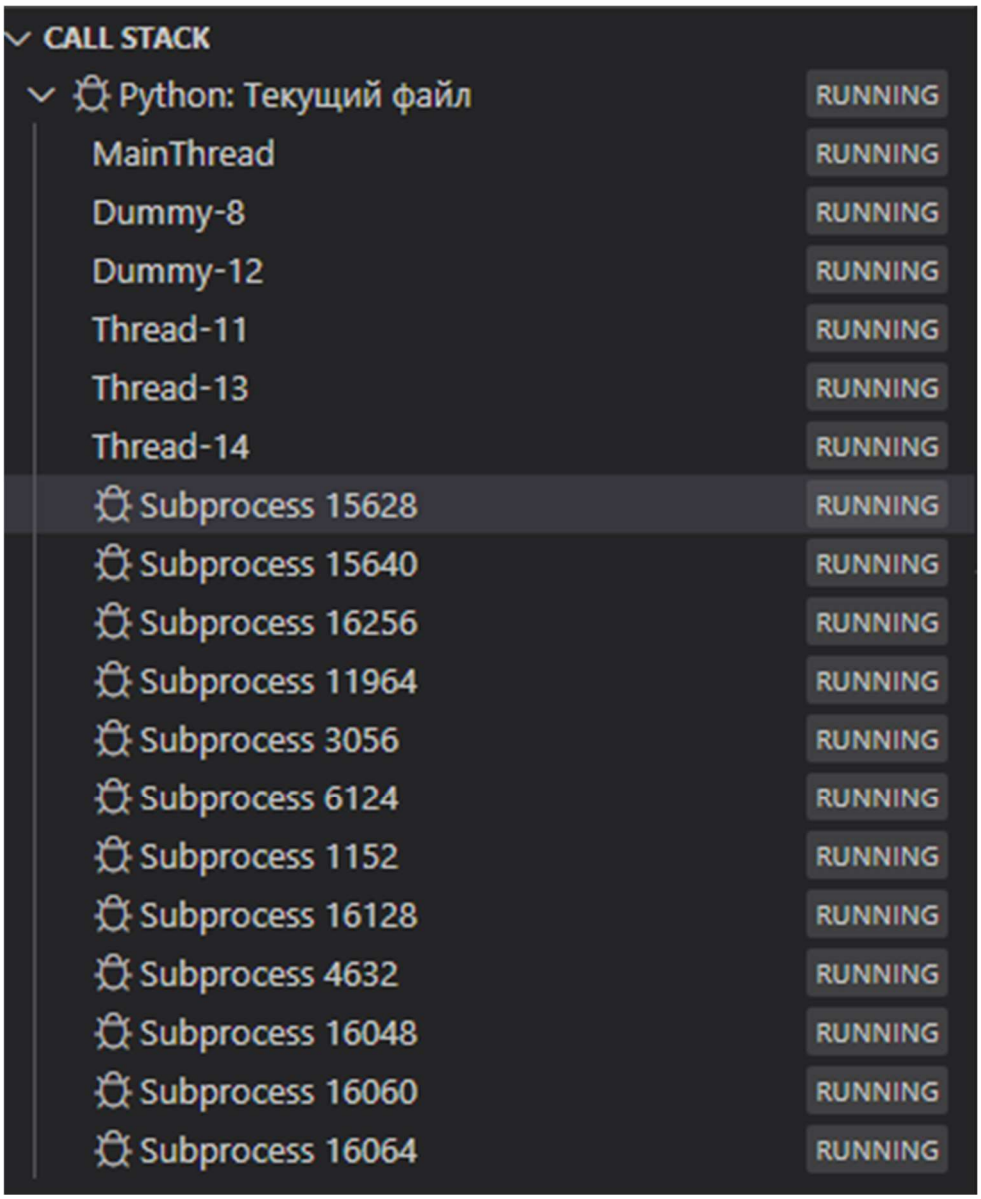


Рисунок 15 - Отображение созданных процессов в отладчике кода VS Code

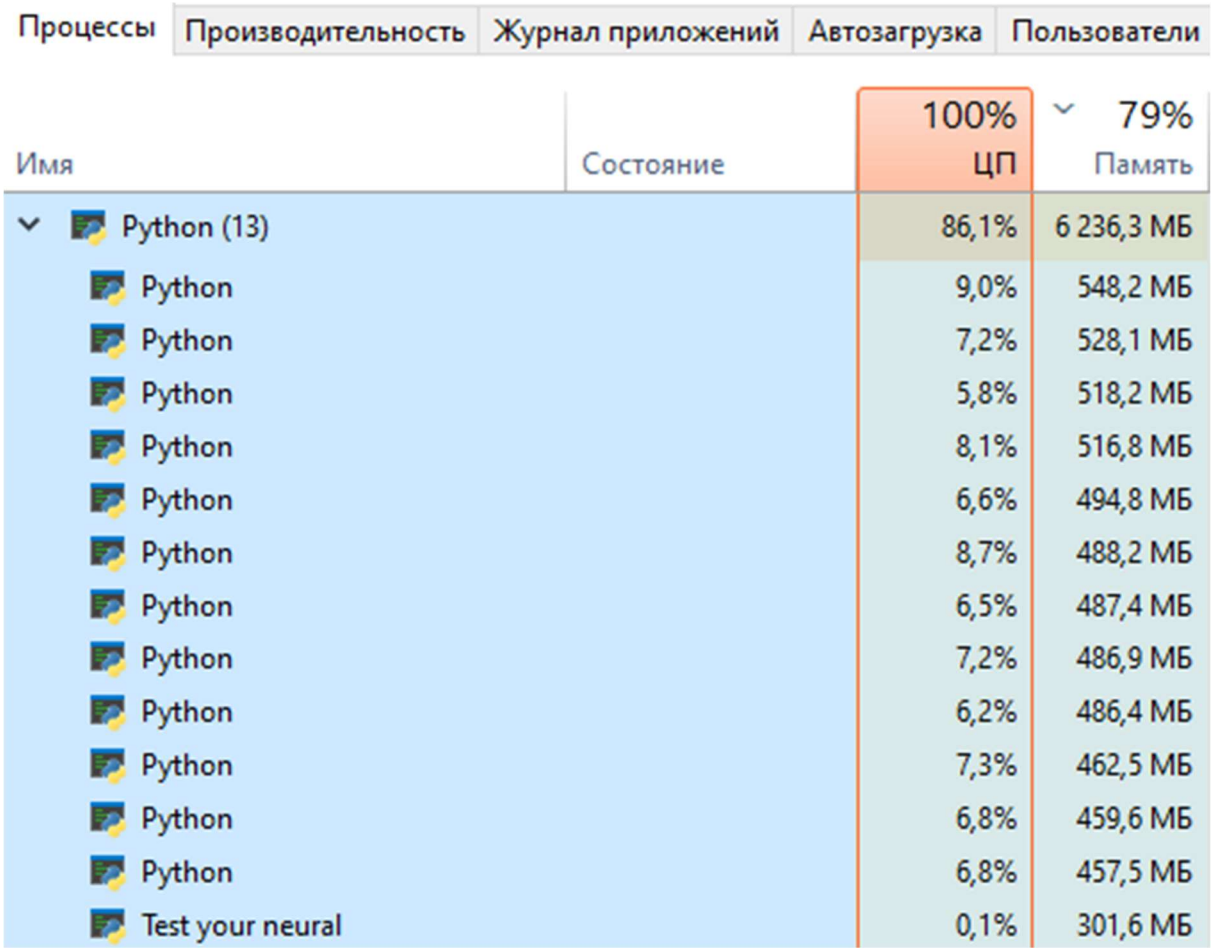


Рисунок 16 - Отображение созданных процессов в диспетчере задач Windows

Из рисунков 15 и 16 можно заметить, что тестирование запустилось на 12-ти ядрах процессора и заняло работу процессора системы на 100%. Каждый из процессов занимает равное количество процессорного пространства и выполняет расчеты параллельно. Из этого можно сделать вывод о том, что программа разделила задачу на подзадачи, количество которых было сконфигурировано пользователем. Скорость тестирования выросла, но также выросла и нагрузка на систему (как на процессор, так и на оперативную память). Для успешного завершения расчетов необходимо подождать некоторое время (в зависимости от конфигурации тестов и размера модели). Это время рассчитывается относительно сложности задачи автоматически.

После успешного тестирования на экране отображается виджет результатов тестирования с графиком результатов и кнопками для сохранения в файлы различных расширений (*.xlsx, *.txt) (рис.17).

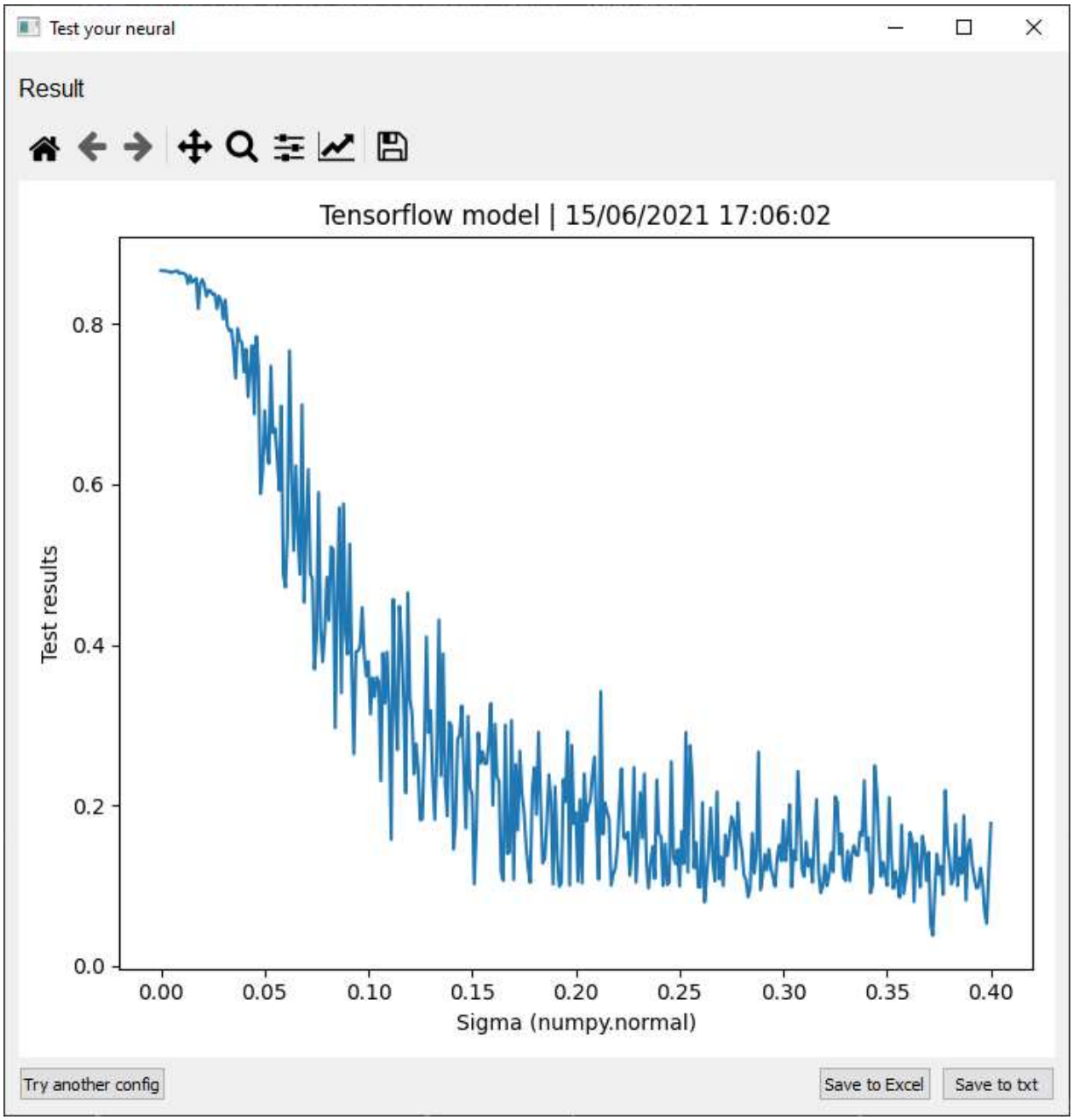


Рисунок 17 - Виджет отображения результатов тестирования

Как видно из рисунка 17, модель с названием «Tensorflow model» была успешно протестирована (рядом верно установлена дата окончания тестирования) с обработкой значений алгоритмом numpy.normal. На графике по оси X отражается значение sigma, а по оси Y – рассчитанные точности созданных во время тестирования моделей.

Необходимо проверить работу и других алгоритмов. Было проведено тестирования работы алгоритма uniform с параметром deviation. Необходимо задать новую конфигурацию теста. Это можно сделать после нажатия на кнопку

Try another config. После нажатия на кнопку снова произойдет создание виджета настройки конфигурации без необходимости перезапускать программу или переописывать методы. Далее необходимо ввести новые параметры тестирования и запустить тест. Результат появится на следующим виджете при завершении тестирования (рис.18).

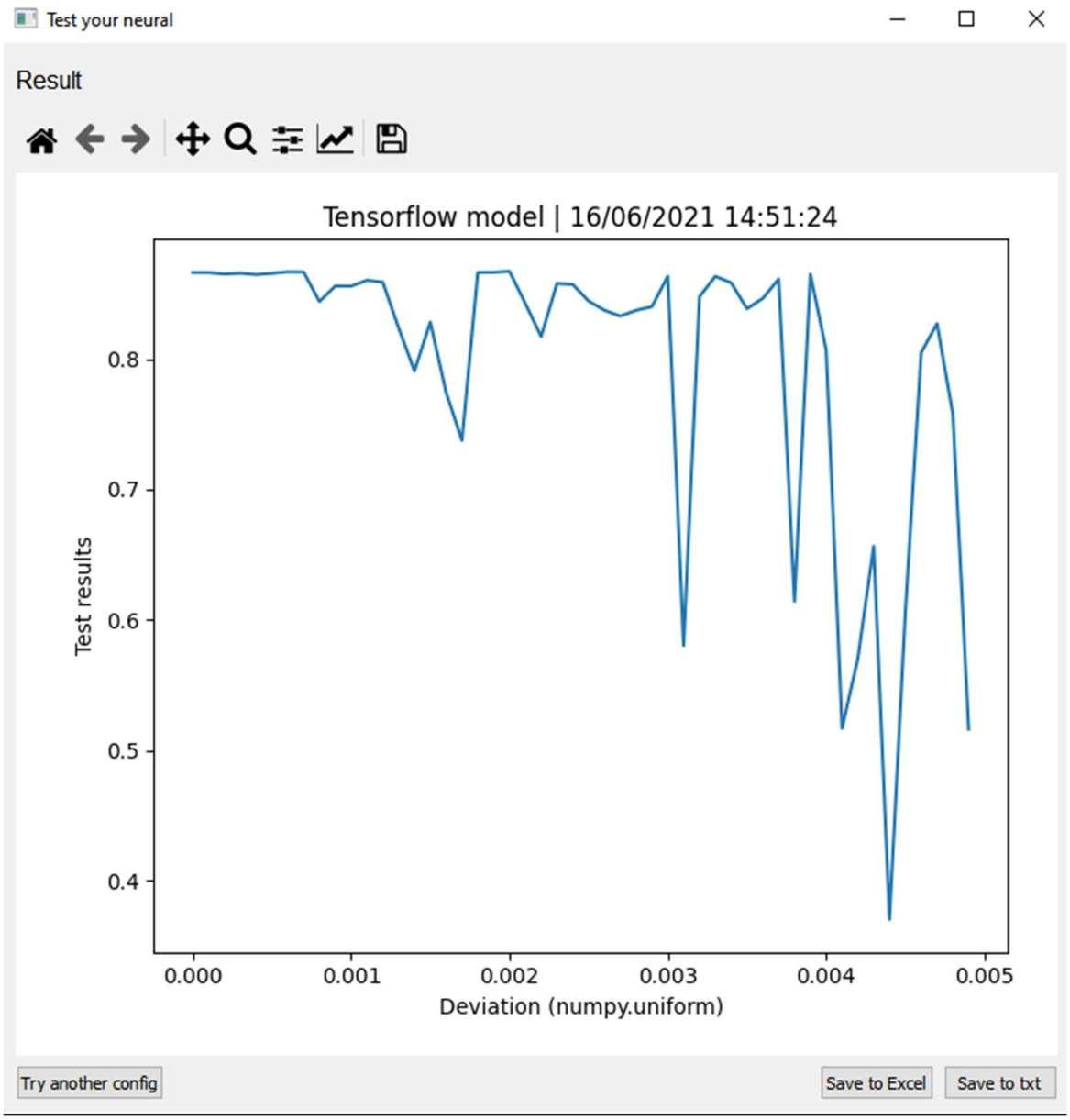


Рисунок 18 - Результат тестирования алгоритма uniform

Далее модель можно сохранить в виде изображения, используя кнопку с изображением дискеты над графиком (рисунок 19 и рисунок 20) или в файлы excel и txt. Результаты сохранения отображены на рисунке 21.

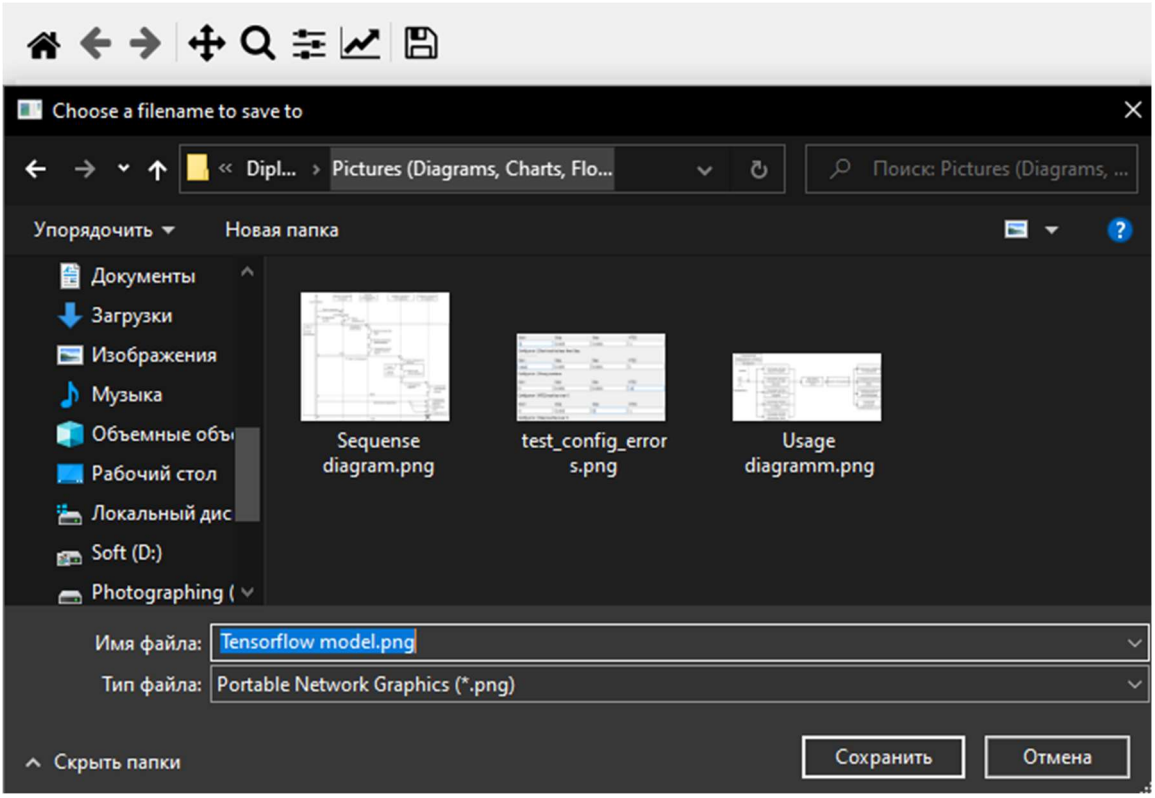


Рисунок 19 - Выбор места для сохранения графика в виде изображения

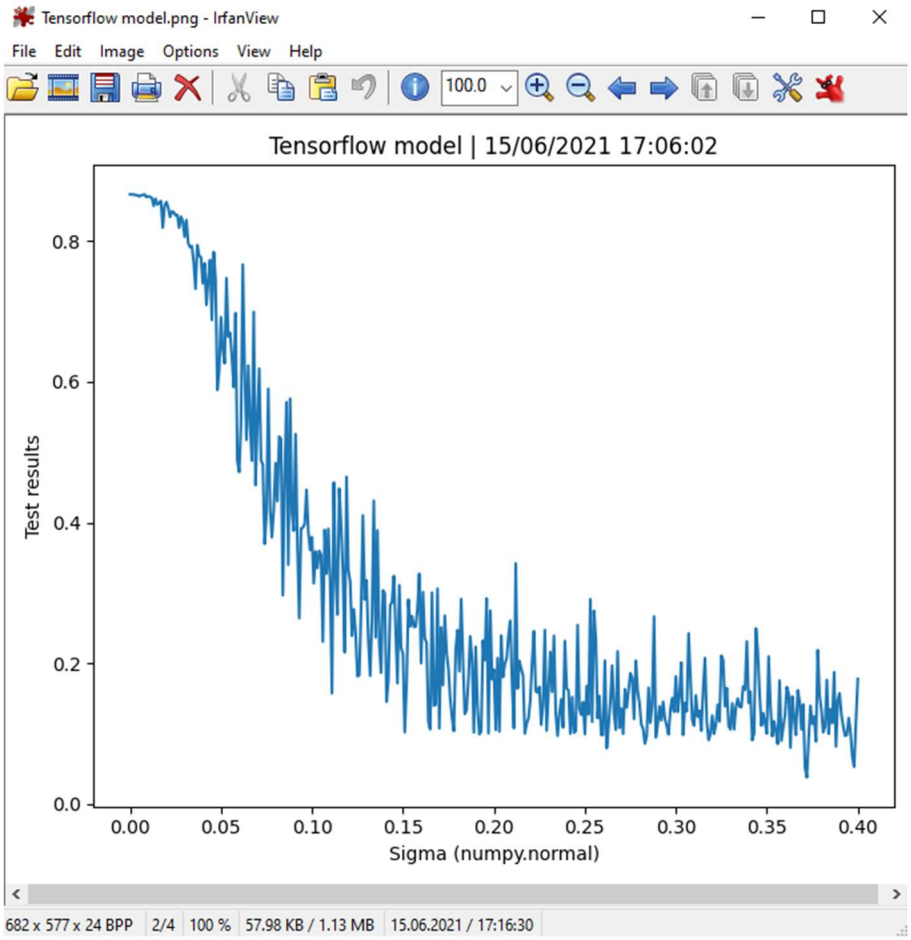
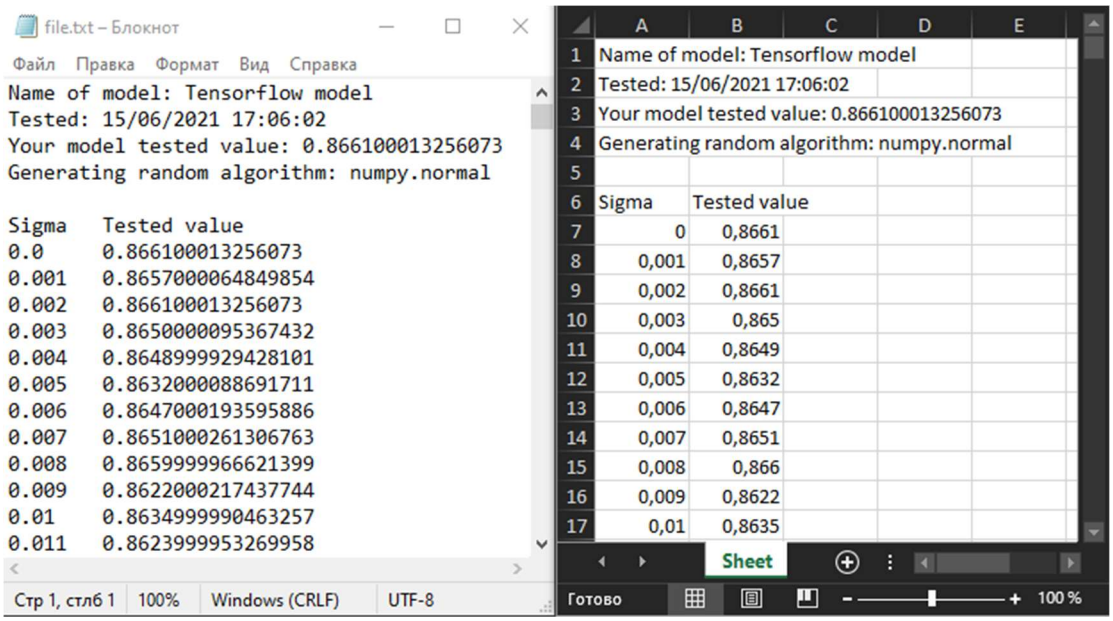


Рисунок 20 - Сохраненный график



а) б)

Рисунок 21 - Сохранение данных графиков: а – сохранение в txt;
б – сохранение в excel

4.2 Анализ производительности в зависимости от входных данных и аппаратных данных устройства

После проверки корректности работы данного модуля необходимо провести тестирование на различных аппаратных конфигурациях системы, а также на нескольких моделях. Для тестирования было выбрано два компьютера, работающих под управлением операционной системы Windows 10.

Компьютер с объемом оперативной памяти 16 Гб и частотой 3200 МГц и процессором AMD Ryzen 5 3600, имеющим тактовую частоту 3,6 ГГц и 6 физических (12 виртуальных) ядер (в дальнейшем Компьютер).

Ноутбук с объемом оперативной памяти 8 Гб и частотой 2400 МГц и процессором Intel® Core(TM) i5-7200U, имеющим тактовую частоту 2,5 ГГц и 4 физических ядра (в дальнейшем Ноутбук).

Будет протестировано две модели нейронной сети:

- модель Fashion_MNIST с двумя слоями: 800 нейронов на входном слое и 10 нейронов - на выходном (в дальнейшем Fashion_mnist);

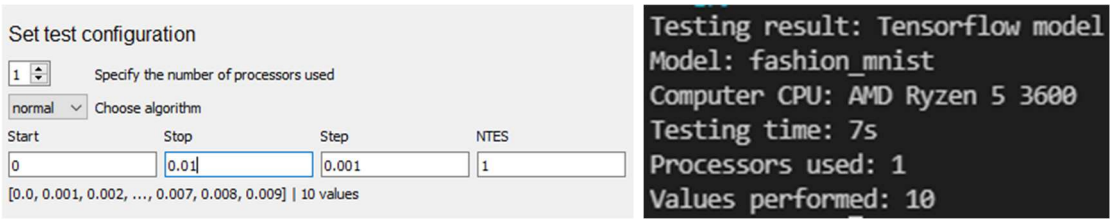
- модель Fashion_MNIST, имеющая 4 слоя: 800 нейронов на входном слое, 2

скрытых слоя, по 1000 нейронов в каждом, и выходной слой, состоящий из 10 нейронов (в дальнейшем Fashion_mnist_big).

Алгоритм будет всегда установлен normal (так как это не повлияет на процесс тестирования). Время выполнения каждого алгоритма различается несущественно и не сильно скажется на результатах тестирования.

4.2.1 Тестирование работы модуля на Компьютере

Первый тест будет задействовать всего одно ядро системы и иметь 10 входных данных для тестирования Fashion_mnist (рис.22).



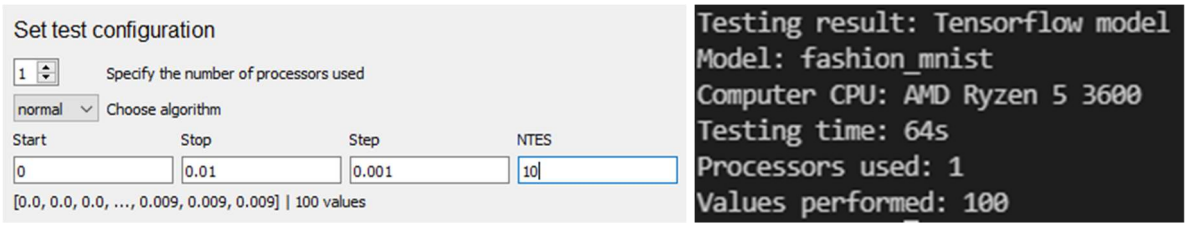
а) б)

Рисунок 22 - Конфигурация теста: а – конфигурация теста;

б – результат тестирования

Из результатов тестирования видно, что Компьютер на одном ядре обработал всю поставленную задачу за 7 секунд.

Второй тест будет задействовать также одно ядро, но количество входных данных вырастет до 100 для модели Fashion_mnist (рис.23).



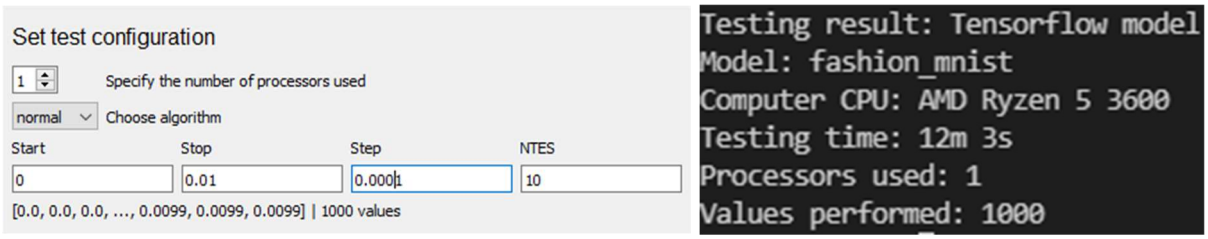
а) б)

Рисунок 23 - Конфигурация теста: а – конфигурация теста;

б – результат тестирования

В результате этого теста время тестирования выросло до 64 секунд.

Третьим тестом будет 1000 значений тестирования на одном ядре и на Fashion_mnist. Для этого снова вызовем виджет настройки конфигурации и зададим новые параметры (рис.24).

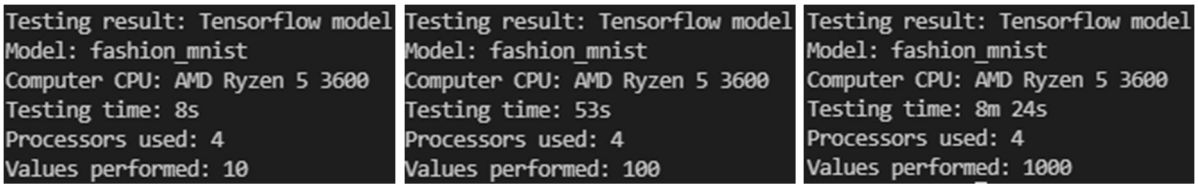


а) б)

Рисунок 24 - Конфигурация теста: а – конфигурация теста;
б – результат тестирования

В результате этого теста время выросло до 12-ти минут.

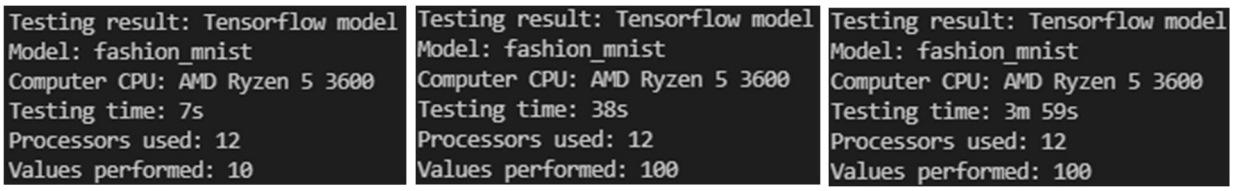
На рисунках 25.а, 25.б и 25.в представлены аналогичные тесты для 4 используемых ядер (33% процессора системы). Это даст небольшой прирост в производительности и не сильно загрузит систему. Задачи будут разделены на 4 одинаковые части и посчитаны параллельно.



а) б) в)

Рисунок 25 - Тестирование Fashion_mnist: а – 10 тестовых данных;
б – 100 тестовых данных; в – 1000 тестовых данных

Далее необходимо проверить работу программы на системе, нагрузив процессор на 100%, чтобы увеличить скорость обработки тестов. На рисунках 26.а, 26.б и 26.в представлены аналогичные тесты для максимального количества используемых ядер в системе Компьютер. В данном случае выросла производительность, что позволило выполнить тестирование быстрее.



а) б) в)

Рисунок 26 - Тестирование Fashion_mnist: а – 10 тестовых данных;
б – 100 тестовых данных; в – 1000 тестовых данных

В результате проведенных тестов можно построить трехмерный график (рис.27) зависимости времени тестирования от количества используемых при тестировании ядер процессора и количества значений, которые нужно будет обработать (таблица 1).

Таблица 1 - Зависимость времени тестирования от ядер и количества операций

Ядра / Значения	1	4	12
10	7	8	7
100	64	53	38
1000	723	504	239

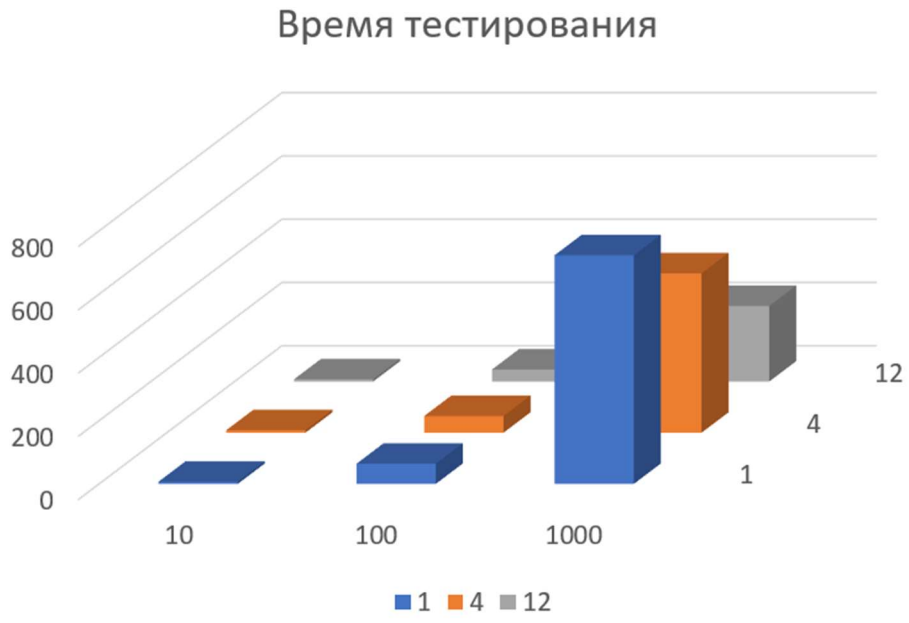


Рисунок 27 - График времени тестирования

Далее будут приведены аналогичные тесты для модели Fashion_mnist_big. Данные тестирования отображены в таблице 2 и на рисунке 28.

Таблица 2 - Зависимость времени тестирования от ядер и количества операций

Ядра / Значения	1	4	12
10	16	13	12
100	102	87	56
1000	2152	1628	890

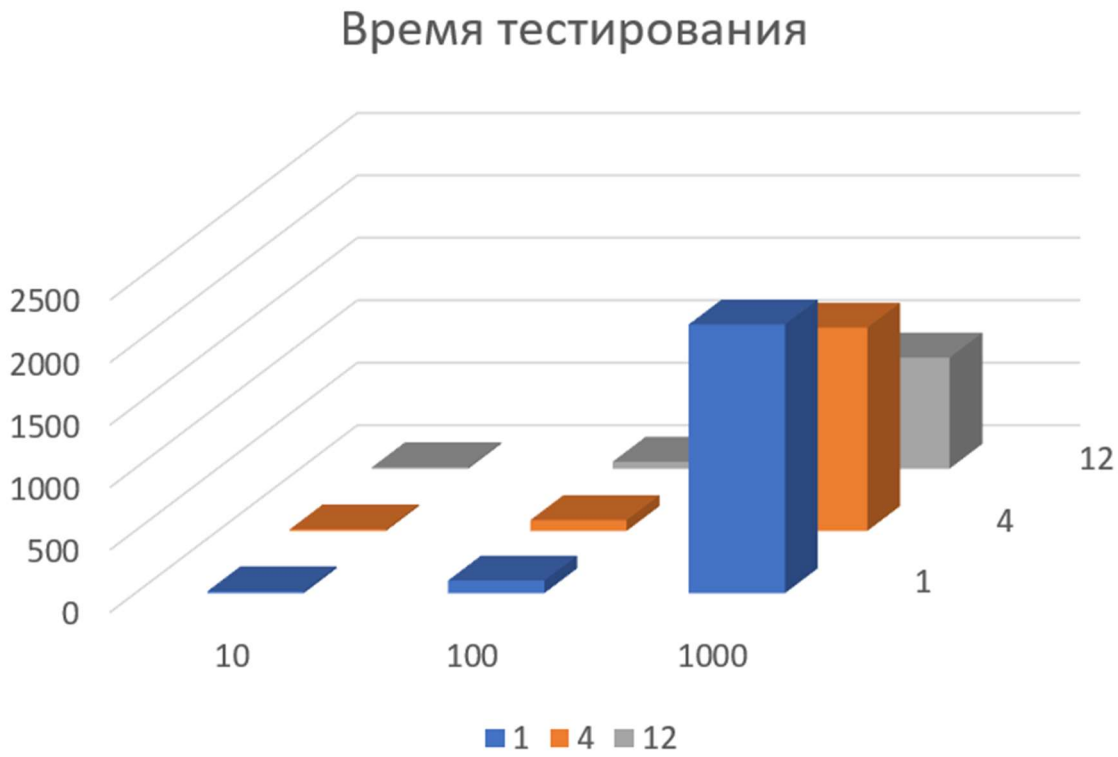


Рисунок 28 - График времени тестирования

Тестирование показало, что при использовании одноядерного решения поставленной задачи требуется в среднем на 150% больше времени, чем при разделении задачи на подзадачи и использовании системы на 100%.

Чтобы понять, насколько важно использовать многоядерную и мощную систему для тестирования нейронных сетей данным модулем, необходимо произвести аналогичные тесты на Ноутбуке – менее производительной системе.

4.2.2 Тестирование работы модуля на Ноутбуке

Тесты будут отличаться от прошлых только системой, на которой эти тесты будут выполняться и количеством используемых ядер (на Ноутбуке их 4). Вначале будет протестирована Fashion_mnist. Результаты представлены в таблице 3 и на рисунке 29.

Таблица 3 - Зависимость времени тестирования от ядер и количества операций

Ядра / Значения	1	2	4
10	18	13	17
100	176	148	113
1000	2042	1593	1142

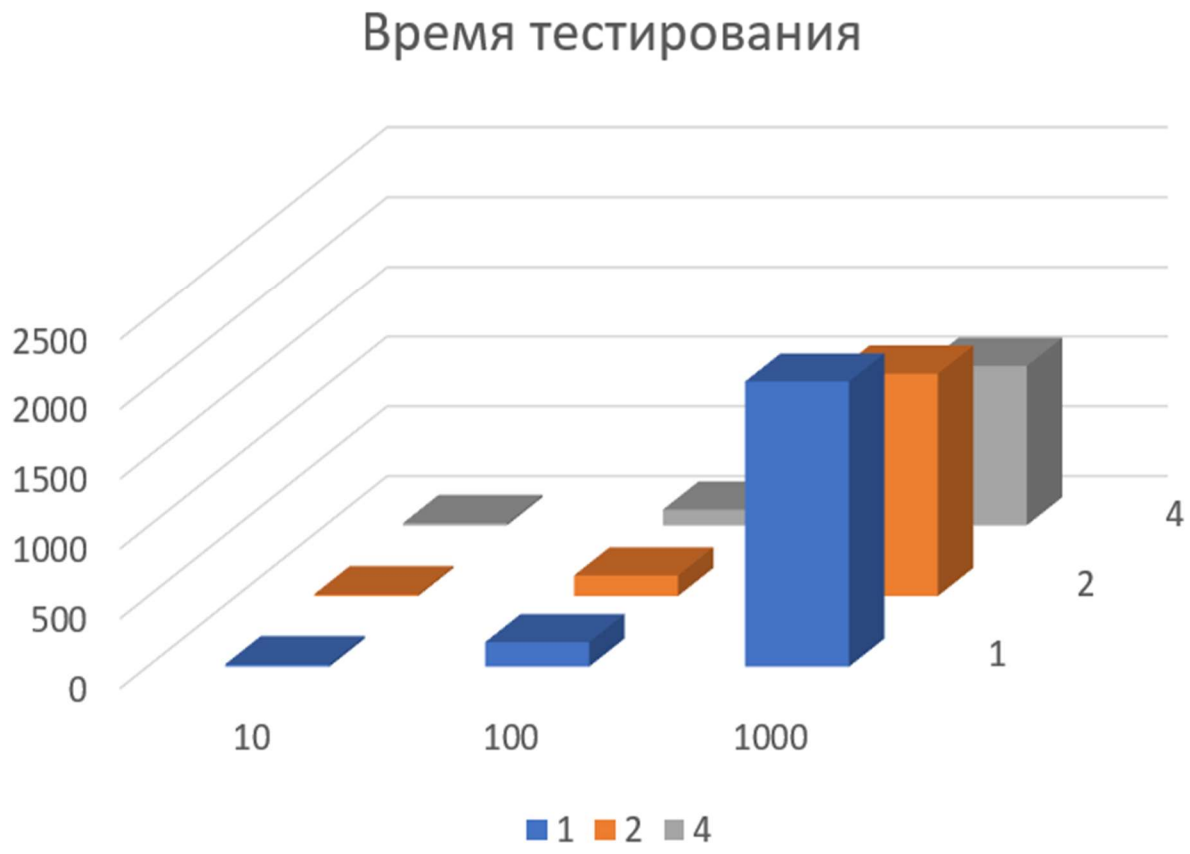


Рисунок 29 - График времени тестирования

Далее будет протестирована модели Fashion_mnist_big. Результаты представлены в таблице 4 и на рисунке 30.

Таблица 4 - Зависимость времени тестирования от ядер и количества операций

Ядра / Значения	1	2	4
10	31	25	24
100	299	231	170
1000	3102	2615	2097

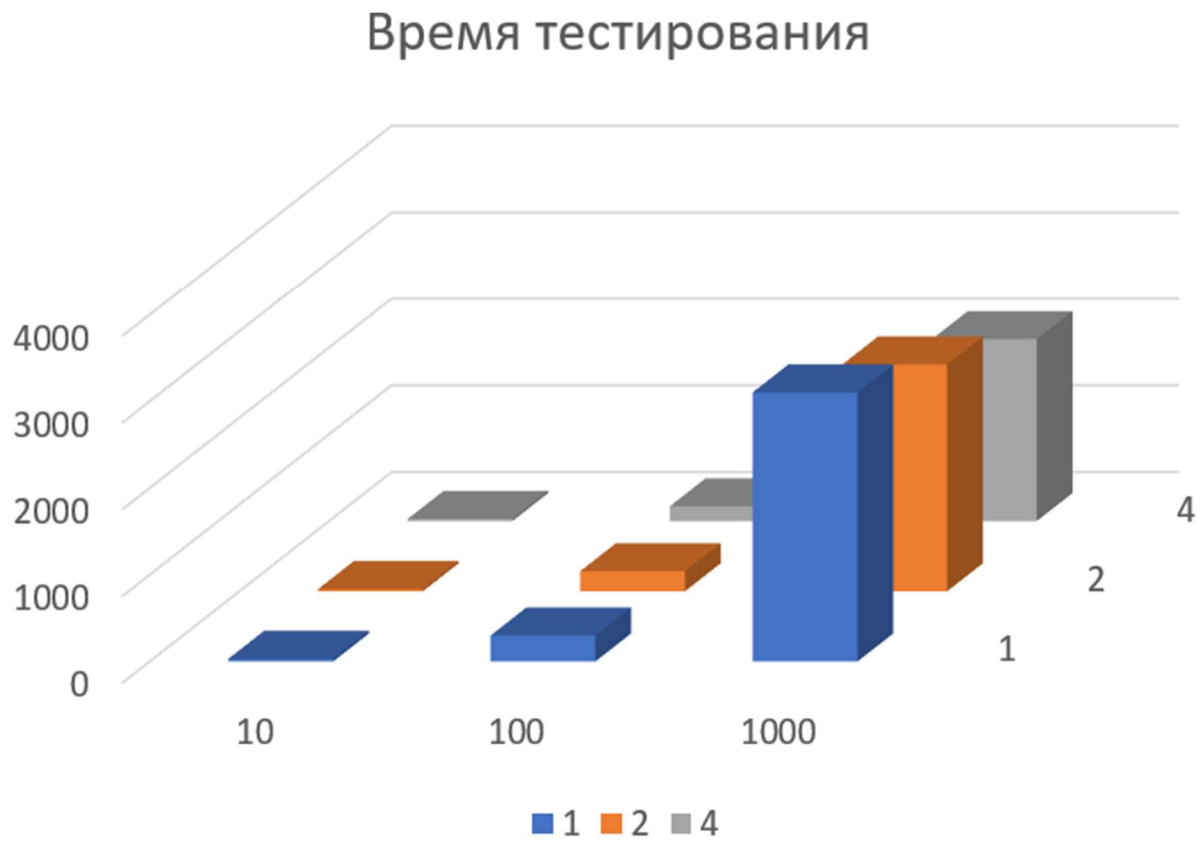


Рисунок 30 - График времени тестирования

4.2.3 Сравнение получившихся результатов

Все тесты проводились на системе, на которой не было запущено никаких других программных средств или утилит, кроме данной программы. Поэтому проведенные тесты можно назвать «чистыми».

Согласно проведенному тестированию, можно сделать вывод о том, что производительность системы сильно влияет на результаты. Чем больше ядер в системе, тем быстрее она может выполнить поставленную задачу, разбив ее на множество подзадач.

От количества тестов зависит будущее время тестирования. Если таких тестов мало (в данном случае проверялось время выполнения 10-ти тестов), то время тестирования практически не изменяется, так как больше времени тратится на процесс разбития задачи на подзадачи. Но при увеличении количества тестов, происходит прирост производительности, если увеличивать количество используемых ядер.

Также производительность данной программы напрямую зависит от системы, на которой производится тестирование. С увеличением аппаратных характеристик, таких как оперативная память (ее объем и частота), процессор (его частота и количество физических и виртуальных ядер), также ускоряется процесс проведения тестов.

Таким образом, было проведено тестирование разработанной программы на двух системах: Компьютере и Ноутбуке. Программа корректно обрабатывает ошибки пользователя при описании дочернего класса программы. Если хотя бы в одном из методов, которые необходимо переописать пользователю, возникнет ошибка, программа корректно её обработает и отобразит соответствующее ошибке сообщение. Также программа разбивает общий список задач на списки подзадач в количестве равному количеству используемых ядер, указанных при конфигурации тестирования, из-за чего сильно увеличивается производительность и уменьшается время тестирования. В конце тестирования пользователю предоставляется возможность выполнить повторный тест с другой конфигурацией тестирования над той же моделью еще раз.

ЗАКЛЮЧЕНИЕ

В данной работе приведено описание процесса разработки программы имитационного моделирования моделей машинного обучения.

Таким образом, в процессе разработки программы были выполнены следующие поставленные на работу задачи:

- реализована возможность тестировать любую модель нейронной сети;

- добавлены алгоритмы имитации аналоговых помех для тестирования работы нейронных сетей (алгоритм равномерного распределения и алгоритм нормального распределения);

- повышена производительность работы программы за счет использования метода распараллеливания процессов в целях сокращения времени тестирования;

- сформулированы выводы о работе реализованной программы тестирования:

а) производительность системы сильно влияет на результаты (чем больше ядер в системе, тем быстрее она может выполнить поставленную задачу, разбив ее на множество подзадач);

б) от количества тестов зависит будущее время тестирования, а также при увеличении количества тестов, происходит прирост производительности, если также увеличивается и количество используемых ядер;

в) производительность данной программы напрямую зависит от системы, на которой производится тестирование (с увеличением аппаратных характеристик, таких как оперативная память (ее объем и частота), процессор (его частота и количество физических и виртуальных ядер), также ускоряется процесс проведения тестов).

Таким образом, задачи, поставленные на выпускную квалификационную работу, можно считать выполненными в полном объеме.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Перепелкин, Е.Е. Вычисления на графических процессорах (GPU) в задачах математической и теоретической физики; Огни - Москва, 2017. - 750 с.

2 Галушкин, А. И. Нейрокомпьютеры. Учебное пособие / А.И. Галушкин. - М.: Альянс, 2014. - 528 с.

3 Смирнов А. Д. Архитектура вычислительных систем: Учебное пособие для вузов. — М.: Наука, 1990. — 320 с.

4 Таненбаум, Э.С. Современные операционные системы: [пер. с англ.]. Питер, 2011. – 1115 с.

5 Каган, Б. М., Каневский, М. М., Цифровые вычислительные машины и системы, 2-е изд., М., 1973. – 347 с.

6 Вьюненко, Л.Ф. Имитационное моделирование: учебник и практикум для академического бакалавриата / Л. Ф. Вьюненко, М. В. Михайлов,; под ред. Л. Ф. Вьюненко. — М. : Издательство Юрайт, 2017. — 283 с.

7 Калачев, А.В. Многоядерные процессоры. Учебное пособие; Интернет-Университет Информационных Технологий (ИНТУИТ) - М., 2017. - 132 с.

8 Multiprocessing — Process-based parallelism [электронный ресурс]. Режим доступа: https://docs.python.org/3/library/multiprocessing.html (дата обращения: 14.04.21).

9 Кобелев, Н.Б. Имитационное моделирование объектов с хаотическими факторами: Учебное пособие / Кобелев Н.Б. - М.: КУРС, НИЦ ИНФРА-М, 2016. - 192 с.

10 Гелиг, А. Х. Введение в математическую теорию обучаемых распознающих систем и нейронных сетей. Учебное пособие / А.Х. Гелиг, А.С. Матвеев. - М.: Издательство СПбГУ, 2014. - 224 с.

11 Тархов, Д.А. Нейросетевые модели и алгоритмы. Справочник / Д.А. Тархов. - М.: Радиотехника, 2014. - 397 с.