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

[Аннотация русскоязычная]

[Аннотация англоязычная]

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»**

(МИВлГУ)

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 3](#_Toc74573318)

[1 Анализ технического задания 4](#_Toc74573319)

[1.1 Зачем нейросетям краш-тест 4](#_Toc74573320)

[1.2 Обзор методов организации параллельных вычислений 4](#_Toc74573321)

[1.3 Выбор средства 8](#_Toc74573322)

[1.4 Требования к разрабатываемой программе 9](#_Toc74573323)

[2 Проектирование 11](#_Toc74573324)

[3 Разработка 13](#_Toc74573325)

[4 Тестирование 13](#_Toc74573326)

[ЗАКЛЮЧЕНИЕ 14](#_Toc74573327)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 15](#_Toc74573328)

# ВВЕДЕНИЕ

Значимость работы, как она может улучшить мир

Цель данной бакалаврской работы – разработать программу имитационного моделирования моделей машинного обучения при условии изменения данных внутри нейросети.

Задачи, поставленные на бакалаврскую работу:

- произвести обзор предметной области;

- проанализировать методы ускорения процесса обработки нейросетей;

- …

- проанализировать изменение данных внутри нейросетей при различных отклонениях входных параметров и сделать соответствующие выводы.

# 1 Анализ технического задания

# 1.1 Зачем нейросетям краш-тест

Потом блок от преподавателя.

# 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.

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). Во многих современных приложениях обработки сигналов она достигает 50:1, причем со сложностью алгоритмов увеличивается;

- локальность данных по времени - каждый элемент загружается и обрабатывается за время, малое по отношению к общему времени обработки, после чего он больше не нужен. В результате в памяти потокового процессора для каждого «вычислителя» можно хранить только данные, необходимые для обработки одного элемента, в отличие от центральных процессоров с моделью произвольно зависимых данных.

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.

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 особенностям архитектуры:

- память GPU 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:

- распространенные математические операции и функции вещественного аргумента. В рамках SM 4.0 поддерживаются целые числа и логические операции, а в SM 4.1 и CUDA — также и вещественные числа двойной (64‑битной) точности;

- организацию циклов. В SM 3 длина циклов ограничена 255 итерациями, в SM 4 длина циклов не ограничена;

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

В различных источниках информации можно найти много разных определений процессов и потоков. Такой разброс определений обусловлен, во-первых, 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. Процесс был придуман для организации всех данных, необходимых для работы программы. 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:

- счётчик 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 множества потоков внутри одного процесса 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 ресурсы, а потоки — это единицы выполнения, которые выполняются на процессоре.

# 1.3 Выбор средства

Для разработки программы были выбраны следующие средства:

1) язык программирования Python;

2) модуль Threading;

3) 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.

GIL (Global Interpreter Lock) – 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 отсутствие GIL может негативно сказываться на производительности 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. 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.

В процессе тестирования разработанной программы будет произведено сравнение результатов работы программы при использовании обоих модулей, а также будут сделаны выводы по полученным результатам.

# 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.

На выходе пользователь программы должен получать график и возможность сохранения готового ответа, выданного программой (например, в виде графа).

Добавить картинок

# 2 Проектирование

UML-diagrams

Как разделяю на задачи multiprocessing

Сделал, столкнулся, победил, как, чем? (столкнулся с проблемой отображения прогресса всех дочерних процессов, решил с использованием sharedmemory)

Я ускорил процесс – главная задача

На рисунке … представлено различие между последовательной схемой решения (верхняя часть схемы) и способом распределенных вычислений (нижняя часть схемы). Последовательная схема решения в данном случае не подходит, так как предполагается довольно большой объем обрабатываемых данных.

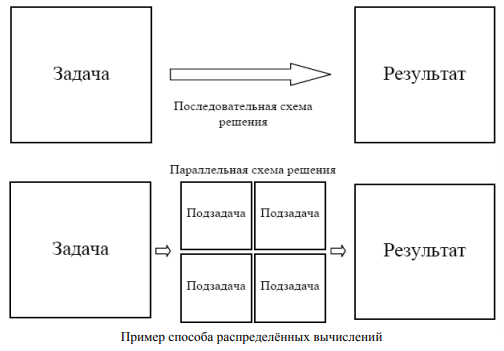


Рисунок … - Схема последовательных и распределенных вычислений

Процессор компьютера, на котором производится разработка программы, содержит 12 ядер. В процессе тестирования нейросетью будет задействовано 11 ядер, так как 12-е ядро будет занято процессом выполнения программы в общем. Таким образом, теоретически, производительность программы должна вырасти.

Графики функций для GeneratingRandomAlgorithms

Для генерации случайных значений весов будут использоваться функции нормального распределения, равномерного распределения и распределения Пуассона. Функцию, с которой будет работать модель, выбирает пользователь.

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

Нормальное распределение играет важнейшую роль во многих областях знаний. 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.

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;

- коэффициент эксцесса - параметр распределения, задающий «остроту» пика распределения.

Типичные формы нормального распределения для различных средних и дисперсии представлены на рисунке …:

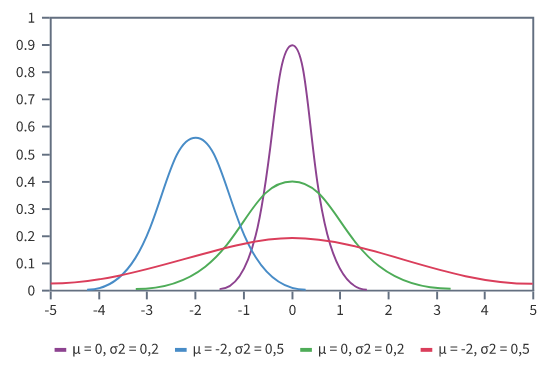


Рисунок … - Примеры форм нормального распределения

Формула данной функции – формула 1:

(1)

где *μ* — математическое ожидание, *σ*2 — дисперсия.

Коэффициент асимметрии определяется следующим образом (формула 2):

(2)

где *E* - знак усреднения. 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:

(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.

Таким образом, при равномерном распределении плотность вероятности имеет вид (формула 4):

(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*) (рисунок …), в связи с чем равномерное распределение иногда называют «прямоугольным».

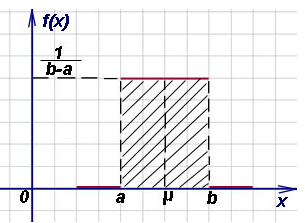


Рисунок … - График кривой равномерного распределения

Функция распределения *F*(*x*) непрерывной случайной величины при равномерном распределении имеет вид (формула 5):

(5)

Характеристики равномерного распределения:

- среднее значение (математическое ожидание) (формула 6):

; (6)

- дисперсия (формула 7):

; (7)

- стандартное отклонение (формула 8):

; (8)

- равномерное распределение не имеет моды.

(тут добавить про пуассоновское распределение)

На рисунке … представлена общая схема работы программы:

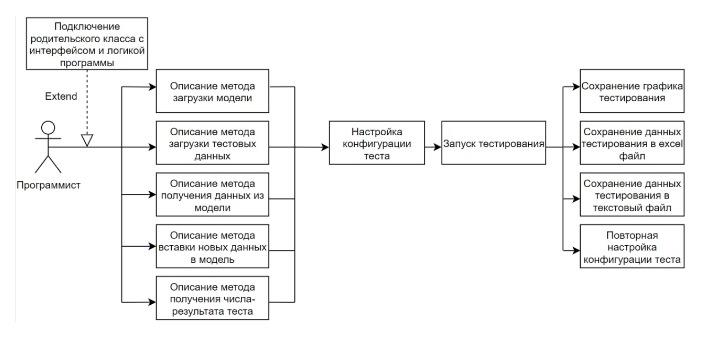


Рисунок … - общая схема работы программы

Отсюда 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. Далее пользователь выполняет следующие действия:

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

- описывает метод загрузки тестовых данных (либо загружает готовые данные (изображения, сигналы и т.д.), 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 (данная характеристика ограничена числом ядер, имеющихся в системе, по умолчанию выбрано максимальное количество);

- алгоритм обработки данных (нормальное распределение, равномерное распределение или распределение Пуассона);

- 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).

Далее происходит сам процесс тестирования, после чего на экране программы отображается график, отражающий результат тестирования.

После этого пользователь может сделать следующее:

- сохранить полученный в результате тестирования график в виде изображения;

- сохранить полученные данные тестирования в файл .xlsx;

- сохранить полученные данные тестирования в текстовый файл;

- повторно настроить конфигурацию теста.

# 3 Разработка

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

Проверка функций, описанных пользователем, производится следующим образом:

try:

if self.name == 'load\_model':

self.neuralconfig.load\_model()

try:

pickle.dumps(self.neuralconfig.model)

except TypeError:

raise PickleProblem

elif self.name == 'load\_testdata':

self.neuralconfig.load\_testdata()

elif self.name == 'get\_tested\_values':

self.neuralconfig.\_basedata = self.neuralconfig.get\_tested\_values()

if type(self.neuralconfig.\_basedata) is not list:

raise NumpyProblem

for array in self.neuralconfig.\_basedata:

if type(array) is not np.ndarray:

raise NumpyProblem

elif self.name == 'set\_tested\_values':

self.neuralconfig.set\_tested\_values(self.neuralconfig.\_basedata)

elif self.name == 'test\_model':

self.neuralconfig.\_test\_value = self.neuralconfig.test\_model(self.neuralconfig.model, self.neuralconfig.testdata)

self.function\_checked\_signal.emit(self.name, 'OK')

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 NumpyProblem:

self.function\_checked\_signal.emit(self.name, 'ERROR')

self.error\_signal.emit('This method must returns a list of numpy arrays (numpy.ndarray)', 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()

Для каждого метода, описанного пользователем, код данной функции запускается последовательно. Первоначально проверяется метод load\_model, который загружает модель в класс.

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.

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.

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

try:

1 / step

if step < 0:

raise ZeroDivisionError

except ZeroDivisionError:

self.setError('Step must be over 0')

return

if start >= stop:

self.setError('Start must be less than Stop')

return

self.line\_edits[3].setText(str(ntes))

if ntes < 1:

self.setError('NTES must be over 0')

return

if (stop - start) / step \* ntes > 100000:

self.setError('Too large amount of data')

return

self.config = np.arange(start, stop, step)

self.config = np.around(self.config, 6)

if ntes > 1:

self.config = np.repeat(self.config, ntes)

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 выглядит следующим образом:

class GeneratingRandomAlgorithms():

names = [

'normal',

'uniform',

'poisson',

]

def \_\_init\_\_(self, algorithm='normal'):

if not self.set\_algorithm(algorithm):

self.algorithm = 'normal'

self.value\_name = 'sigma'

def set\_algorithm(self, algorithm):

if algorithm in self.names:

self.algorithm = algorithm

if algorithm == 'normal':

self.value\_name = 'sigma'

elif algorithm == 'uniform':

self.value\_name = 'deviation'

elif algorithm == 'poisson':

self.value\_name = 'lambda'

return True

else:

return False

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)

elif self.algorithm == 'poisson':

return self.\_random\_numpy\_poisson(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]

def \_random\_numpy\_poisson(self, weight, \_lambda):

def generate\_number(number):

return number + np.random.poisson(\_lambda)

return [generate\_number(array) for array in weight]

В зависимости от выбранного алгоритма (Normal, Uniform, Poisson), будет сохранена информация о названии параметра данного алгоритма (sigma, deviation, lambda соответственно) и выполнен соответствующий метод (\_random\_numpy\_normal, \_random\_numpy\_uniform, \_random\_numpy\_poisson соответственно).

Далее пользователь запускает процесс тестирования, что вызывает загрузку виджета прогресса тестирования и старт самого тестирования. Происходит процесс передачи конфигурации данных в класс MultiprocessExecution. 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 \_create\_new\_model(self, value):

new\_weights = self.data['algorithm'].generate(self.neuralconfig.\_basedata, value)

return self.neuralconfig.set\_tested\_values(new\_weights)

def \_compute\_models(self, values):

data = list()

shm = shared\_memory.ShareableList(name=self.shm\_name)

for value in values:

model = self.\_create\_new\_model(value)

data.append((value, self.neuralconfig.test\_model(model, self.neuralconfig.testdata)))

shm[0] += 1

return data

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.

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\_result in lst:

if sigma\_result[0] in XY\_data:

XY\_data[sigma\_result[0]].append(sigma\_result[1])

else:

XY\_data[sigma\_result[0]] = [sigma\_result[1]]

del(sigma\_result)

del(lst)

for key in XY\_data.keys():

# counting Y

XY\_data[key] = np.average(XY\_data[key])

self.data = XY\_data

В случаях, когда пользователем было указано количество тестов больше 1, при расчете координат точек графика, каждый полученный результат в рамках одного диапазона данных усредняется при помощи операции нахождения среднего арифметического.

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()

return toolbar, sc

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. Это реализовано при помощи следующего кода:

def save(self):

if self.\_save\_type == 'excel':

self.\_save\_to\_excel()

elif self.\_save\_type == 'txt':

self.\_save\_to\_txt()

def \_save\_to\_excel(self):

wb = openpyxl.Workbook()

ws = wb.worksheets[0]

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'

for row, key in enumerate(list(self.\_data['data'].keys())):

ws.cell(row=row + 7, column=1).value = key

ws.cell(row=row + 7, column=2).value = self.\_data['data'][key]

filepath = QFileDialog.getSaveFileName(caption='Save your test result to excel file',

directory=f'./',

filter='Excel file (\*.xlsx)')

wb.save(filepath[0])

def \_save\_to\_txt(self):

filepath = QFileDialog.getSaveFileName(caption='Save your test result to txt file',

directory=f'./',

filter='Text file (\*.txt)')

with open(filepath[0], 'w') as f:

f.write('Name of model: ' + self.\_data['name'] + '\n')

f.write('Tested: ' + self.\_data['time'] + '\n')

f.write('Your model tested value: ' + str(self.\_data['test\_value']) + '\n')

f.write('Generating random algorithm: numpy.' + self.\_data['algorithm'].algorithm + '\n\n')

f.write(f'{self.\_data["algorithm"].value\_name.capitalize()}\tTested value\n')

for key, value in self.\_data['data'].items():

f.write(f'{key}\t{value}\n')

Сохранение в формат .xlsx реализовано при помощи библиотеки Openpyxl. 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;

- выбранный пользователем алгоритм распределения данных;

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

Аналогичным образом производится сохранение результатов тестирования в текстовый файл.

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

# 4 Тестирование

Benchmark

Статистика прироста

Разные модели

Проверка на разных компах (ноут комп)

Графики роста производительности от увеличения количества ядер

Маленькие модели и большие модели – разные показатели.

# ЗАКЛЮЧЕНИЕ

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ