МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра Информатики и автоматизации научных исследований**

Направление подготовки: «Прикладная информатика»

Профиль подготовки: «Прикладная информатика в области принятия решений»

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

на тему:

**«Автоматизация анализа данных сканирующей электронной микроскопии»**

**Выполнил:** студент группы 3822М1ПИ

Шеин Илья Дмитриевич

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

**Научный руководитель:**

Доцент кафедры ИАНИ, к.т.н.

Липкин Семен Михайлович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

Нижний Новгород  
2024

**Оглавление**

[**Введение.** 3](#_Toc166615345)

[**Глава 1. Сверточные нейронные сети.** 5](#_Toc166615346)

[**1.1.** **Обзор основных блоков сверточной нейронной сети.** 5](#_Toc166615347)

[**1.2.** **Обзор архитектур сверточных нейронных сетей.** 5](#_Toc166615348)

[**Глава 2. Постановка задачи.** 8](#_Toc166615349)

[**2.1.** **Формальная постановка задачи.** 8](#_Toc166615350)

[**2.2.** **Варьируемые параметры.** 9](#_Toc166615351)

[**2.3.** **Критерий оптимальности.** 9](#_Toc166615352)

[**Глава 3. Алгоритмы решения задачи.** 10](#_Toc166615353)

[**3.1.** **Генерация искусственных данных.** 10](#_Toc166615354)

[**3.2.** **Метод на основе детектирования объектов.** 11](#_Toc166615355)

[**3.3.** **Метод end-to-end.** 14](#_Toc166615356)

[**Глава 4. Техническая реализация.** 14](#_Toc166615357)

[**Глава 5. Результаты запусков.** 15](#_Toc166615358)

[**5.1.** **Результаты метода на основе детектирования объектов.** 15](#_Toc166615359)

[**5.2. Результаты метода end-to-end.** 19](#_Toc166615360)

[**5.3. Выводы и предложения по результатам экспериментов.** 21](#_Toc166615361)

[**Выводы.** 21](#_Toc166615362)

[**Список литературы.** 22](#_Toc166615363)

[**Приложение.** 23](#_Toc166615364)

# **Введение.**

В наше время развитие производства представляет собой, как и усовершенствование существующих отраслей, так и появлением новых. В современном индустриальном производстве аддитивные технологии становятся все более значимыми, обеспечивая высокую степень гибкости, точности и экономичности в процессе производства деталей сложной формы. Ключевым сырьем для аддитивных технологий являются порошковые материалы, в частности порошки металлов и сплавов. Они обладают уникальными свойствами и позволяют создавать изделия с высокой прочностью и сложной геометрии. Данные порошки, характеризуются не только химическим и фазовым составом, но также и гранулометрическим составом, который является распределением количества частиц в единице объема по геометрическим размерам. Гранулометрический состав порошков играет критическую роль в процессе аддитивного производства, влияя на качество и свойства изготавливаемых изделий. Так для определения данного состава порошковых материалов используются механические и оптические методы [1]. Механические методы являются более косвенными, из-за этого и менее точные и основаны на применении декантирования, то есть, иными словами, отделение одних порошковых материалов от других. Применение оптических методов, таких как получение микрофотографий материала и статический анализ получаемых изображений, обеспечивает высокую точность и непрерывный контроль качества порошков, что важно для соответствия конечных изделий требуемым техническим характеристикам и стандартам. Но несмотря на высокую точность оптических методов, сейчас данный метод ограничен трудоемкостью обработки микрофотографий из-за недостаточности уровня автоматизации данного процесса. Так, например, ГОСТ 23402-78 [2] предполагает ручную обработку микрофотографий. Для автоматизации данного процесса, можно попытаться использовать современные методы компьютерного анализа изображений. Это может позволить уменьшить затрачиваемое время на обработку каждой фотографии и уменьшить погрешность подсчёта гранулометрического состава.

На практике используются различные методы компьютерного подсчета распределения частиц. Существует два современных метода автоматизированного анализа размера частиц на основе преобразования Хафа (Hough transformation) и модуля ImageJ ParticleSizer [3]. Основным недостатком данных методов является наличие одного или нескольких параметров, подбираемых человеком под каждое конкретное изображение для наибольшей точности работы. Таким образом, недостаточное или неравномерное освещение, а также наложение частиц друг на друга или на различных уровнях, это все приводит к новому подбору параметров, для достижения наилучшего результата.

Также существуют методы, основанные на нейронных сетях, которые позволяют убрать ручную настройку параметров, и они более устойчивы к изменению изображений. Самыми популярными нейронными сетями для решения данной задачи являются сверточные нейронные сети (CNN) [4]. Для их обучения достаточно иметь набор фотографий с размеченными на них объектами. Так мы можем выделить каждую отдельную частицу для входного изображения или классифицировать объекты на то является это группой нескольких объектов или это отдельная частица. Примерами таких архитектур являются RCNN, Fast-RCNN, Mask-RCNN [6][8].

Таким образом, **целью** данной работы является разработка метода автоматического получения гистограммы распределения частиц порошка по размерам.

Из-за сложности получения достаточного для обучения количества изображений в наборе данных, необходимо их сгенерировать самостоятельно по подобию реально существующего материала. Примером исходного порошкового материала для создания синтетического набора данных является частично спеченный диоксид кремния. Для достижения поставленной цели будут решены следующие задачи.

**Задачи:**

1. Генерация набора искусственных данных для подбора параметров нейронных сетей и тестирования.
2. Реализации методов вычисления распределения для синтетического набора данных и проведение экспериментов.
3. Определить оптимальный с точки зрения точности алгоритм метода вычисления распределения частиц порошка.
4. Сформулировать выводы о применимости выбранных методов.

# **Глава 1. Сверточные нейронные сети.**

## **Обзор основных блоков сверточной нейронной сети.**

Рассмотрим принцип сверточных нейронных сетей и архитектуры потенциально используемых моделей.

Сверточные нейронные сети являются алгоритмом глубокого обучения, на вход которого приходит изображение, выделяются определенные аспекты на основании которых получается различать изображения друг от друга. Также сверточные нейронные сети — это математическая конструкция, которая обычно состоит из трех типов слоев (или строительных блоков): свертка, объединение и полносвязные слои [4]. Сверточные слои помогают выделить различные характеристики изображений, такие как края, углы и текстуры. Объединение позволяет уменьшить размерность изображения, сохраняя важные признаки. Это позволяет уменьшить количество параметров в сети и сделать более устойчивым к переобучениям. Полносвязные слои принимают данные из предыдущих слоев и отображают извлеченные объекты в окончательный результат. После этого чаще всего применяется функция softmax для получения вероятностного распределения по классам. Также важным элементом сверточных нейронных сетей является активационная функция, что добавляет нелинейности в модель, тем самым позволяя ей выучить более сложные зависимости в данных.

## **Обзор архитектур сверточных нейронных сетей.**

Архитектура сверточных нейронных сетей включает в себя несколько блоков,

которые были описаны выше, такие как свертки, объединения и полносвязные слои.

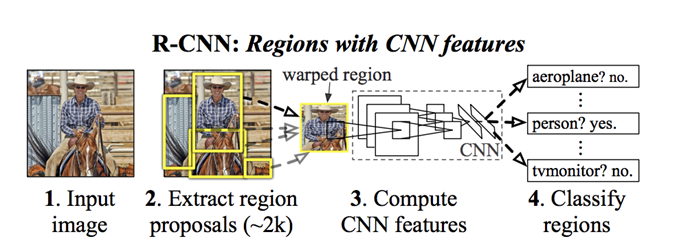
Типичная архитектура состоит из повторяющихся слоев сверток и объединения, после которых идут несколько полносвязных слоев.

Рассмотрим несколько примеров архитектур сверточных нейронных сетей:

* **RCNN, Fast-RCNN, Mask-RCNN**.

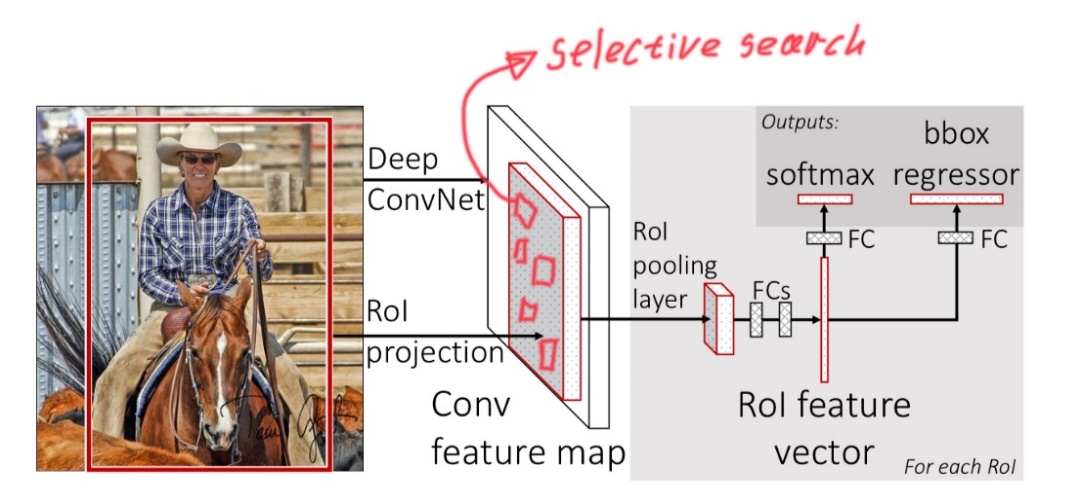
Данные архитектуры нейронных сетей применяются для детектирования и сегментации объектов на изображениях. Рассмотрим каждую из них подробнее.

**RCNN**:

Эта модель сужает поиск возможных положений объекта при помощи алгоритма Region Proposal [7]. Основная идея данного алгоритма, что на вход приходит изображение, а на выходе мы получаем множество прямоугольников, в которых потенциально может находится необходимый объект. И затем применяется CNN к каждой области для извлечения признаков. Основным недостатком модели является медленная скорость работы из-за многократного применения CNN к каждой области. Архитектура R-CNN представлена на рисунке 1.1.

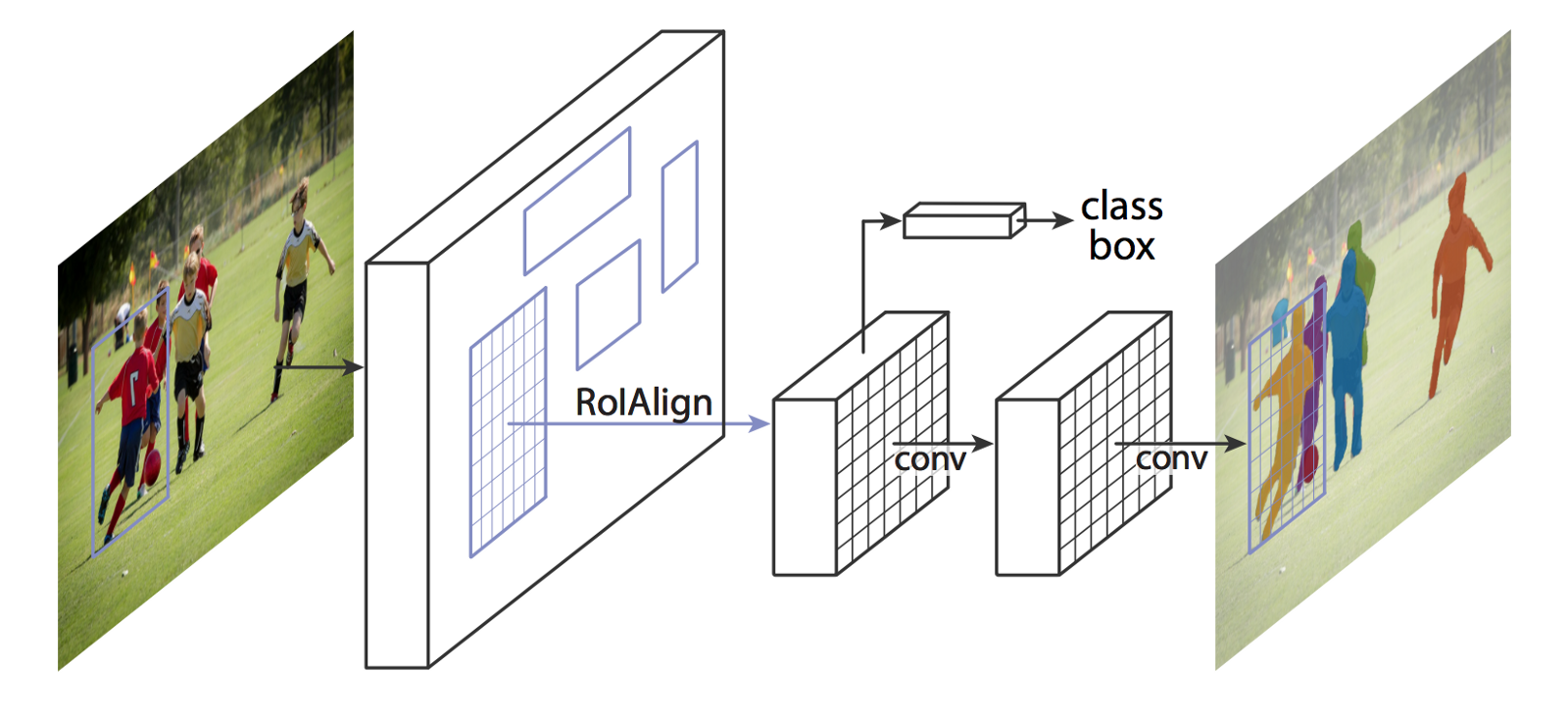
**Рис.1.1** Архитектура RCNN

**Fast-RCNN**:

Данная модель является улучшением обычной модели RCNN для ускорения детекции объектов. Основным отличием от предшественника является использования сверточной нейронной сети для всего изображения сразу, а не для каждой области по отдельности. На выходе мы получаем карты объектов, которые фиксируют различные шаблоны признаков. Далее запускается Selective Search (берутся все найденные прямоугольники и соседние группируются по сходству) и проецируются прямоугольники на карту объектов. Также Fast-RCNN включает в себя Region of Interest Pooling Layer для извлечения фиксированного размера признаков из каждой области [8]. Благодаря чему уменьшается количество вычислений и увеличивается скорость обработки. Архитектура Fast-RCNN представлена на рисунке 1.2.

**Рис.1.2** Архитектура Fast-RCNN

**Mask-RCNN**:

 Данная модель является расширением Fast-RCNN, позволяющее делать сегментацию объектов на изображении. Отличительной чертой является наличие дополнительной ветви в модели, которая генерирует маски для каждого обнаруженного объекта. Маска представляет собой просто прямоугольную матрицу, в которой 1 на некоторой позиции означает принадлежность соответствующего пикселя объекту заданного класса, 0 – что пиксель объекту не принадлежит [9]. Она позволяет точно определять и сегментировать объекты на изображении при этом сохраняя высокую скорость обработки изображений благодаря эффективным оптимизациям. Архитектура Mask-RCNN представлена на рисунке 1.3.

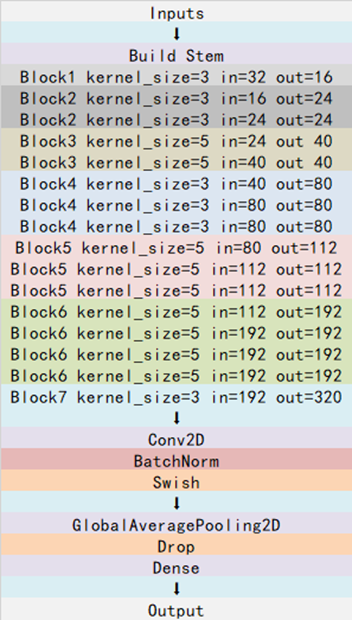
**Рис.1.3** Архитектура Mask-RCNN

* **EfficientNet**:

Модель EfficientNet — это такой класс моделей для классификации изображений, который получен путем масштабирования моделей, а также балансирования между собой глубины и ширины сети, и кроме того разрешения изображения сети [10].

Модель EfficientNet обладает особенностью, которая разработана со ссылкой на другие отличительные нейронные сети. И совмещает в себе следующие характеристики классической нейронной сети: [10]

* Использовать остаточную нейронную сеть, чтобы увеличить глубину нейронной сети, и реализовать извлечение признаков через более глубокую нейронную сеть.
* Если увеличить разрешение входного изображения, то можно заставить сеть учиться и выражать разнообразные признаки, тем самым позволив увеличить точность.



**Рис.1.4** Архитектура EfficientNet

# **Глава 2. Постановка задачи.**

## **Формальная постановка задачи.**

Задачей работы является нахождение распределения размеров частиц по микрофотографии некоторого порошка, сделанной с помощью микроскопа. Рассмотрим более узконаправленную задачу.

Пусть у нас есть фотография, которая является RGB изображением с некоторыми размерами. На данном изображении уже размечены необходимые объекты человеком, в нашем случае это частицы порошка. Разметка представляет собой набор прямоугольников, каждый из которых описывается набором координат (верхнего левого и правого нижнего угла прямоугольника), в каждом из которых содержится одна частица. Всего частиц на изображении будет считать N. На основе одного изображения необходимо найти набор координат прямоугольников, который максимально близкий к исходному набору, то есть для которого выполняется требование того, что отношение пересечения площадей прямоугольников найденных частиц с исходными к объединению этих площадей будет максимальным и для которого расхождение Кульбака-Лейблера [11] к исходному набору будет минимальным.

Расхождение Кульбака-Лейблера [11] дает значения, которые варьируются от 0

для идентичных значений до 1 для совершенно разных распределений вероятности.

**Исходные данные.**

Изображение можно представить в виду матрицы A, элементы которой хранят значения интенсивности каждого цветового канала формата RGB:

, где , (2.1)

, X – ширина изображения

, Y – высота изображения

Частицы описываются множеством прямоугольников, каждый из которых состоит из N пар координат углов и заключает внутри себя частицу.

, (2.2)

,

,

,

## **Варьируемые параметры.**

Варьируемые параметры описываются, как множество значения весов нейронной сети. Пусть вся нейронная сеть содержит всего M весов:

(2.3)

## **Критерий оптимальности.**

Критерием является минимизировать расхождение Кульбака-Лейблера [11] для распределения исходных ограничивающих прямоугольников и найденных.

Для этого найдем распределение размеров частиц и размеров ограничивающих прямоугольников, которые получились после нейронной сети:

– распределение размеров исходных частиц

– распределение размеров ограничивающих прямоугольников в результате работы нейронной сети. Где FCNN – нейронная сеть на вход которой приходит изображение (матрица А).

Тогда критерий будет выглядеть следующим образом:

(2.4)

(2.5)

# **Глава 3. Алгоритмы решения задачи.**

## **Генерация искусственных данных.**

Основной проблемой мешающей применять с легкость нейронные сети семейства

RCNN это отсутствие большого количества размеченных микрофотографий.

Для решения данной проблемы может помочь генерация искусственных данных, близких к реальным. Частицы представляют собой объекты сферической формы. Также было решено учесть тот факт, что на реальных изображениях присутствуют тени, отражения и неровности. Для генерации таких изображений было решено реализовать генератор искусственных данных на основе трехмерных изображений, с помощью Blender.

Алгоритм генерации работает следующим образом:

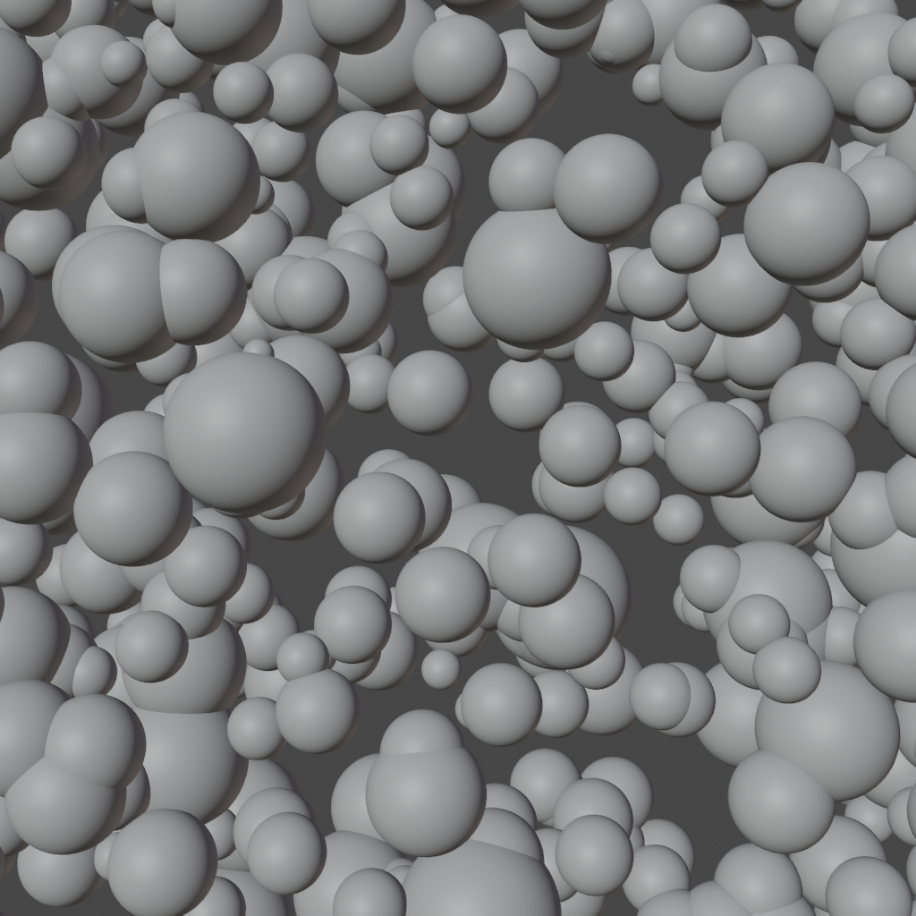
1. Задается количество частиц для текущего изображения (В нашем случае было выбрано число 100). Но так как при генерации, сгенерированные координаты могли выйти за пределы изображения, то количество частиц могло получиться меньше.
2. Случайно генерируются координаты и радиусы.

(3.1)

1. Задается текстура для будущих сфер.
2. Создается пустая сцена.
3. Задаются размеры рендеринга.
4. Задаются лампы для создания более естественных теней и отражений.
5. Устанавливаются координаты камеры (0, 0, 10)
6. Берутся заданные координаты одной частицы, скалируются для более разряженного размещения по формуле:

(3.2)

1. По новым координатам размещается сфера с радиусом r и задается количество сегментов из которых будет состоять сфера (Чем меньше сегментов, тем более неровная получится сфера).
2. Повторять 8-9 пока не будут размещены все сферы.
3. Производится слияние слоев. На фоновый слой накладываются частицы.

** Рис.3.1** Пример сгенерированного изображения

## **Метод на основе детектирования объектов.**

Сверточные нейронные сети являются более актуальными для решения задач

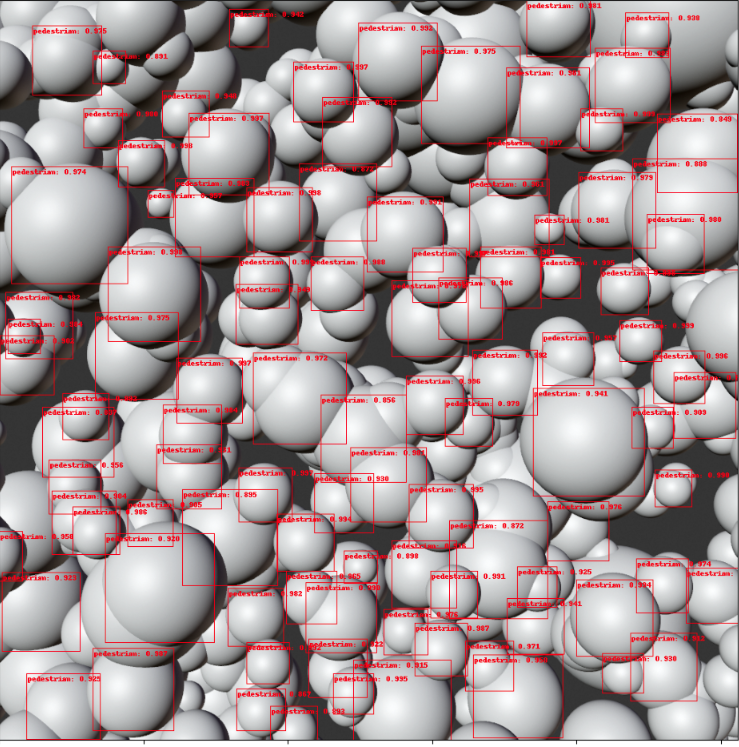
в области компьютерного зрения. В задачах детектирования объектов преимущественно используются сверточные нейронные сети (region based convolution networks, RCNN). Для эксперимента была взята модель Fast-RCNN.

Архитектуру данной нейронной сети можно описать следующим образом:

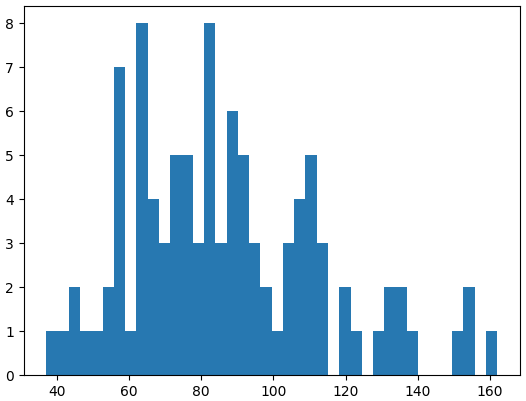
* Нейронная сеть использует предварительно обученную на датасете COCO сверточную нейронную сеть (ResNet 50 [5]) для извлечения признаков их входного изображения. Данные сверточные слои помогают модели выучить различные уровни абстракции объектов на изображении.
* По полученным признакам генерируются области интереса (Region of Interest, RoI) [8], которые могут содержать объекты. Для каждой области используется операция RoI pooling [8], чтобы преобразовать их в фиксированное пространственное распределение для дальнейшей обработки.
* Используя полносвязную нейронную сеть выделенные области на исходном изображении подгоняются под размеры искомых объектов.
* Далее с помощью нескольких полносвязных слоев производится классификация найденных областей по типам объектов.
* После обнаружения объектов применяется метод NMS, который позволяет удалить повторяющиеся обнаруженные объекты с высокой степенью наложения. При этом остается наилучшее предсказание для каждого объекта.

**Описание метода:**

Для каждого изображения на выходе нейронной сети мы получаем набор ограничивающих прямоугольников для объектов на изображении.

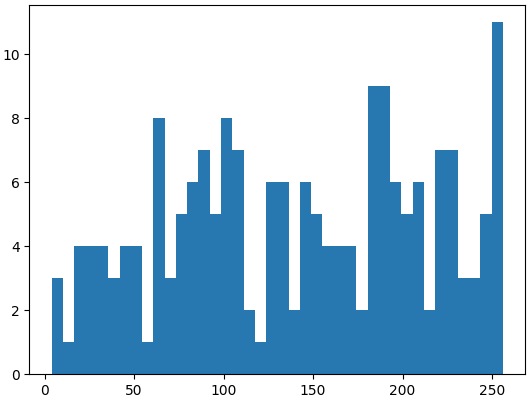


**Рис.3.2** Пример изображения после использования сети Fast-RCNN

Далее по найденным ограничивающим прямоугольникам мы находим распределение по их размерам.

**Рис.3.3** Пример гистограммы размеров ограничивающих прямоугольников на изображении

И соответственно строим гистограмму диаметров для всех частиц, которые есть на изображении.



**Рис.3.4** Пример гистограммы диаметров для всех частиц на изображении

По каждой гистограмме мы генерируем случайную выборку значений, соответствующую значения столбцам по горизонтальной оси с вероятностями, соответствующими вертикальной оси. Далее две полученные выборки сравниваем с помощью расхождения Кульбака-Лейблера [11]. Это нессимитричная мера разницы между двумя распределениями вероятностей, которая рассчитывается по формуле:

, (3.3)

, S – количество элементов в выборке,

Также в качестве дополнительной метрики сравнения используем разность между двумя распределениями:

## **Метод end-to-end.**

Альтернативный рассматриваемый метод предполагает, что мы будем использовать нейронную сеть EfficientNet, которая часто используется для задач классификации [10]. На вход она будет принимать также изображения, каждому из которых будет соответствовать своя гистограмма диаметров частиц. Также добавим еще один дополнительный полносвязный слоя после данной сети, выходами которого будут являться столбцы гистограммы (нормированные распределения).

Основой метрикой данного метода будет функция потерь, которая представляет собой сумму поэлементно вычтенных распределений (из исходного распределения диаметров вычитается полученные распределения на выходах нейронной сети). И то есть в результате, чем ближе к 0 будет данная функция потерь, тем более похожими будут распределения.

# **Глава 4. Техническая реализация.**

В качестве платформы для проведения экспериментов был выбран Google Colab [12].

В качестве языка был выбран Python. Данный язык является скриптовым и позволяет быстро реализовать прототипы и имеет большое количество различных библиотек с готовыми функциями.

Искуственные данные были сгенерированы в два этапа. На первом этапе был написан скрипт на Python для генерации координат сфер и их радиусов, полученные значения которых были записаны в json файл. На втором этапе был использован также скрипт на Python для пакета Blender [13]. Эти два скрипта запускались для каждого изображения отдельно из скрипта написанного на bash. Всего было сгенерировано 10000 изображений. Генерация производилась на персональном компьютере следующей конфигурации: 3.0 Ghz 6-Core Intel Core i5, 16 GB RAM, OS Windows 11.

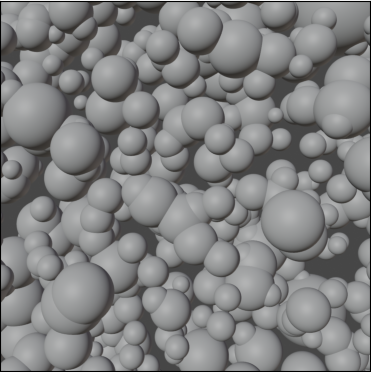
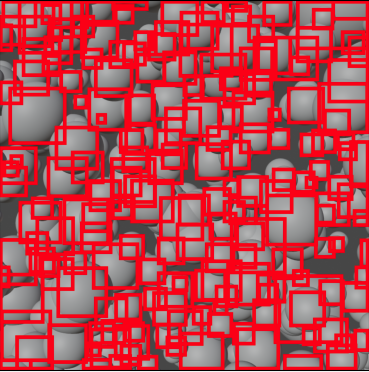
Для реализации первого метода была взята нейронная сеть Fast-RCNN, которая используется для задач детектирования объектов на изображениях. Основой здесь является нейронная сеть ResNet 50 [5], которая была предобучена на датасете COCO. В качестве пакета из которой была взята данная сеть был выбран PyTorch [14]. Для реализации второго метода была взята нейронная сеть EfficientNet, которая используется для классификации объектов на изображении. Данная нейронная сеть была также взята из PyTorch [14].

# **Глава 5. Результаты запусков.**

Для обучения и запусков нейронной сети была использована платформа Google Colab [12], предоставляющая возможность бесплатного использования графических ускорителей с определенными ограничениями или платного использования для выбора более мощных ускорителей, таких, как графический ускоритель NVIDIA A100.

В качестве датасета было сгенерировано 10000 изображений, как говорилось ранее. Из них 8000 изображений использовались в качестве обучающих данных, а 2000 в качестве валидационных.

## **5.1.** **Результаты метода на основе детектирования объектов.**

Для данного метода была дополнительно сделана разметка изображений (для каждого объекта на изображении были добавлены ограничивающие прямоугольники). На рисунке 5.1 и 5.2 приведены примеры изображения и разметки для данного изображения.

**Рис.5.1** Пример изображения из датасета

сета

**Рис.5.2** Пример разметки для изображения

Далее проводим обучение на нейронной сети Fast-RCNN на 10 эпохах с batch\_size = 10 и количеством batch равным 800 на тренировке. И batch\_size = 1 и количеством batch равным 2000. Получаем следующие результаты, представленные в таблице 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер эпохи | Обработка 1 batch на тренировке, сек. | Обработка 1 batch на валидации, сек. | Средняя точность  IoU = 0.5:0.95 | Средняя точность  IoU= 0.5 | Средняя точность  IoU= 0.75 |
| 1 | 3.3514 | 0.3665 | 0.073 | 0.231 | 0.028 |
| 2 | 2.1463 | 0.3729 | 0.265 | 0.469 | 0.280 |
| 3 | 3.2577 | 0.3843 | 0.339 | 0.503 | 0.407 |
| 4 | 3.4378 | 0.4360 | 0.351 | 0.503 | 0.430 |
| 5 | 3.6656 | 0.3561 | 0.358 | 0.503 | 0.441 |
| 6 | 4.6650 | 0.4047 | 0.364 | 0.503 | 0.443 |
| 7 | 5.0417 | 0.4543 | 0.364 | 0.503 | 0.443 |
| 8 | 2.6123 | 0.3453 | 0.368 | 0.513 | 0.454 |
| 9 | 3.0524 | 0.3852 | 0.391 | 0.514 | 0.478 |
| 10 | 3.0874 | 0.4045 | 0.402 | 0.521 | 0.497 |

**Таблица 1.** Результаты работы Fast-RCNN

Пример результата работы Fast-RCNN приведен на рисунке 3.2. Для оценки результатов сначала дадим определение IoU. IoU это отношение пересечения площадей ограничивающих прямоугольников найденных частиц с исходными ограничивающими прямоугольниками к объединению этих площадей. Из результатов таблицы видно, что в наилучшем случае точность работы нейронной сети достигает 52 процента на 2000 изображений. При этом такая точность достигается только при IoU (intersection over union) равным 0.5. При других значениях IoU точность хуже, примерно на 2-12 процентов. Из этого можно сделать вывод, что не все частицы смогли быть обнаружены с помощью нейронной сети, но и также точность самих ограничивающих прямоугольников не совсем совпадает с исходными.

После того, как мы обучили нейронную сеть нам нужно получить распределения диаметров частиц для каждого изображения и распределение размеров найденных ограничивающих прямоугольников из результата работы нейронной сети Fast-RCNN. Примеры таких распределений представлены на рисунках 3.3 и 3.4. В качестве эксперимента рассмотрим три варианта распределения по 10, 20 и 40 значений соответственно.

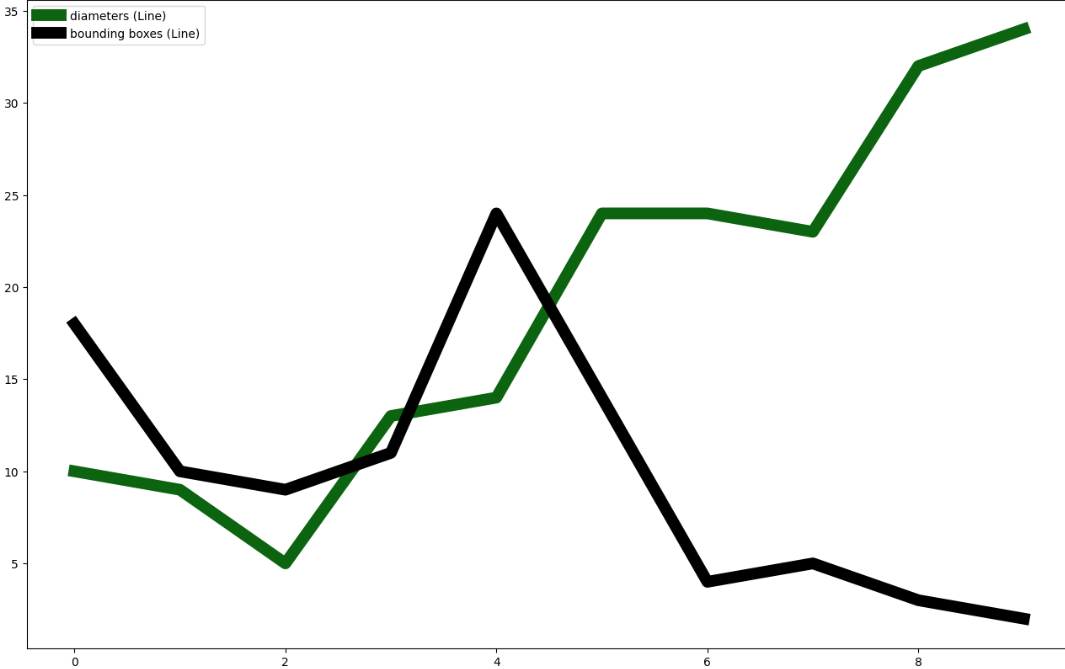
Для каждого изображения посчитаем распределения, нормализуем их и найдем расхождение Кульбака-Лейблера [11] по формулам 2.4 и 2.5. И найдем его среднее значение. Результаты приведены в таблице 2.

|  |  |  |
| --- | --- | --- |
| Количество значений в распределении | Среднее расхождение Кульбака-Лейблера по формуле 2.4 | Среднее расхождение Кульбака-Лейблера по формуле 2.5 |
| 10 | 0.10304726656553434 | 0.10345152973093408 |
| 20 | 0.12436734723547343 | 0.12435264627632841 |
| 40 | 0.13645983483457833 | 0.13689345679033465 |

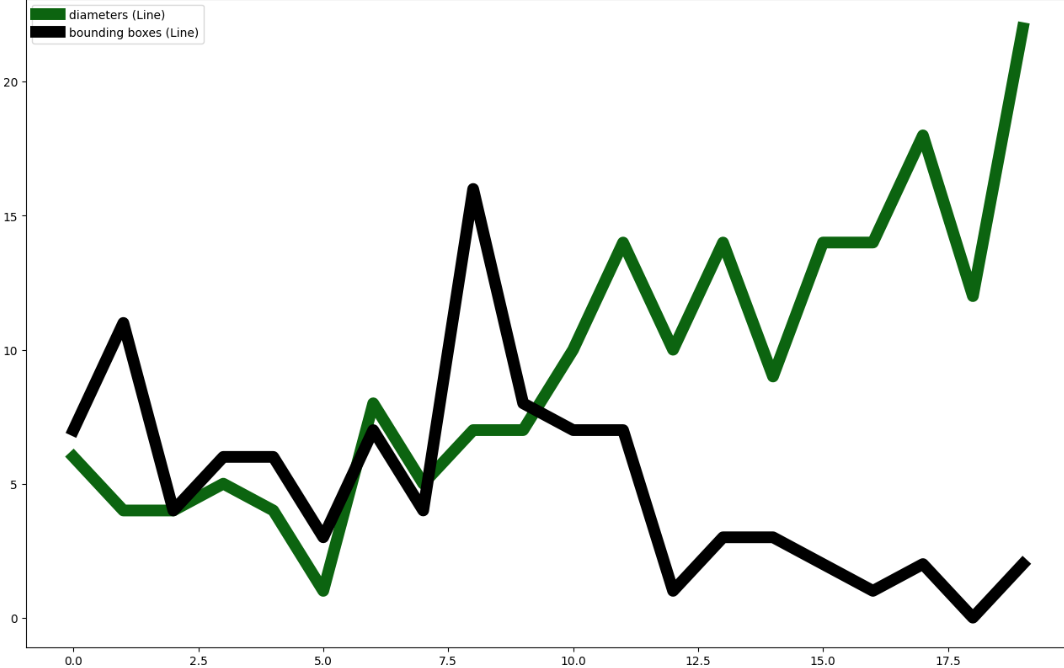
**Таблица 2.** Результаты расчетов расхождения Кульбака-Лейблера

Из результатов таблицы видно, что среднее расхождение Кульбака-Лейблера показывает наилучший результат при 10 значениях в распределении, то есть, когда распределение имеет всего 10 значений. И отличие получается всего на 10 процентов.

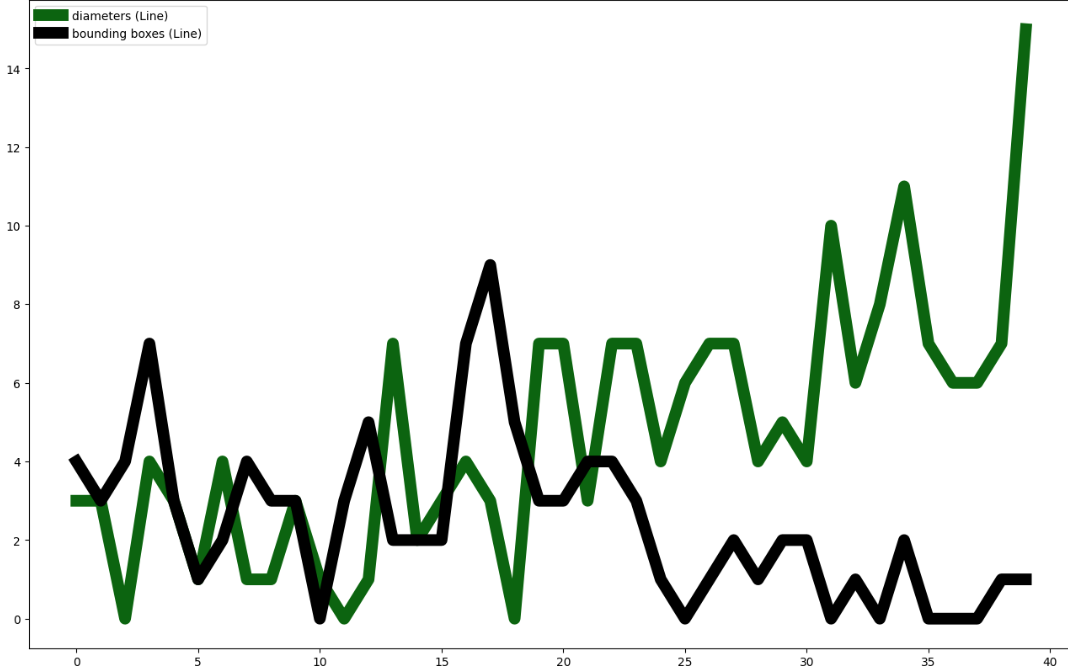
Пример распределений ручной разметки и выхода нейронной сети при 10, 20 и 40 значениях выглядят следующим образом на рисунках 5.3, 5.4, 5.5:



**Рис.5.3** График распределения при 10 значениях



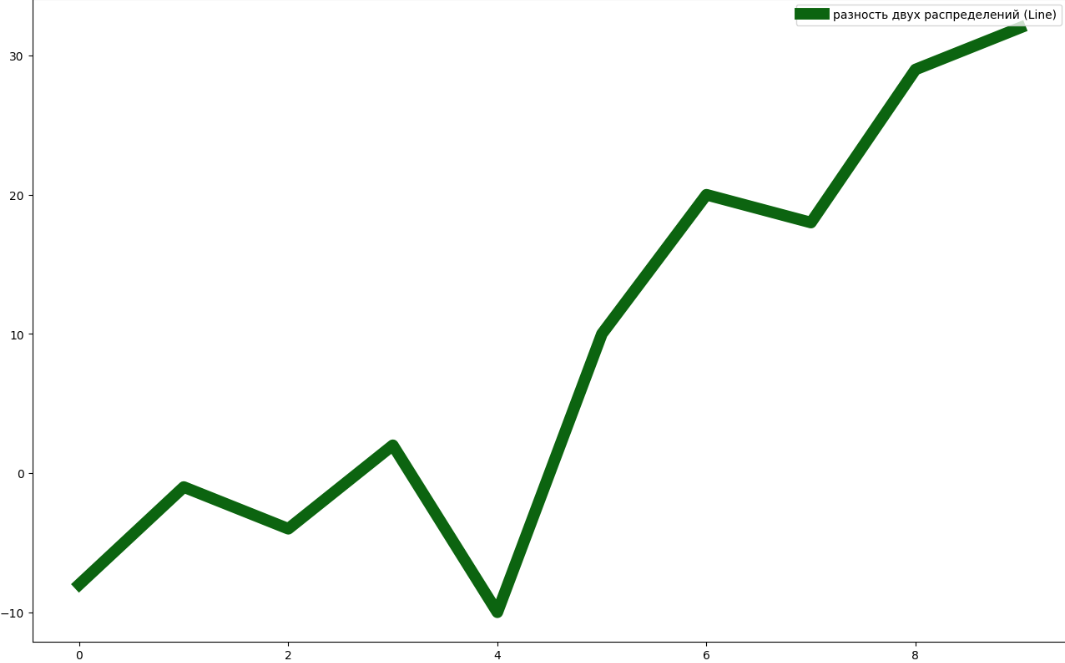
**Рис.5.4** График распределения при 20 значениях



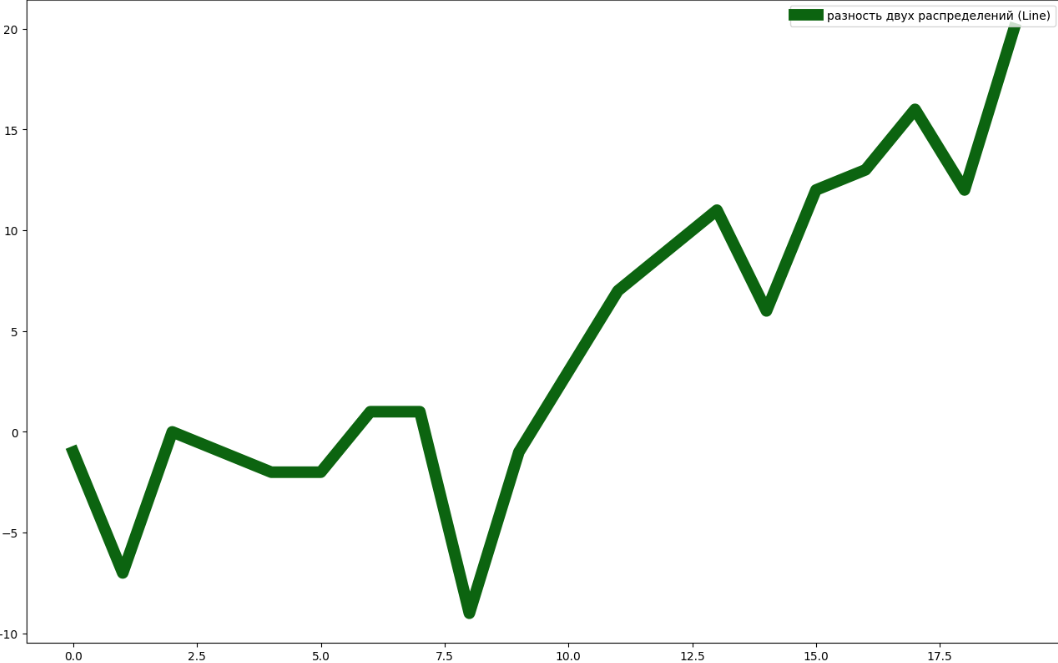
**Рис.5.5** График распределения при 40 значениях

Теперь рассчитаем и выведем метрику, связанную с разностью распределений.

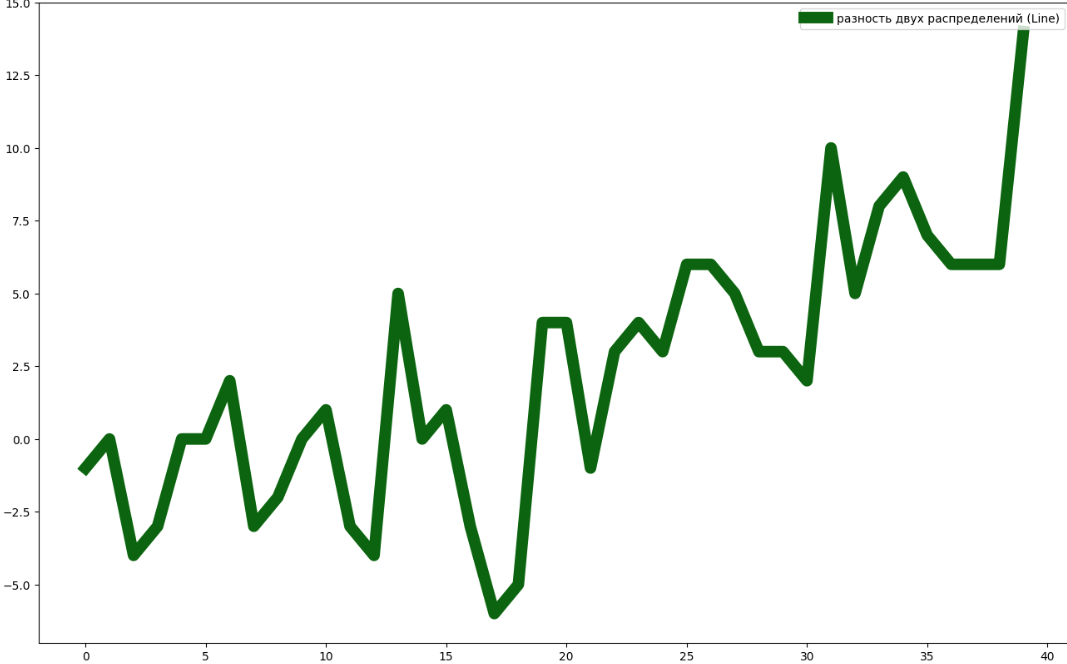
На рисунках 5.6, 5.7 и 5.8 представлены гистограммы разности двух распределений.



**Рис.5.6** График разности двух распределений при 10 значениях



**Рис.5.7** График разности двух распределений при 20 значениях

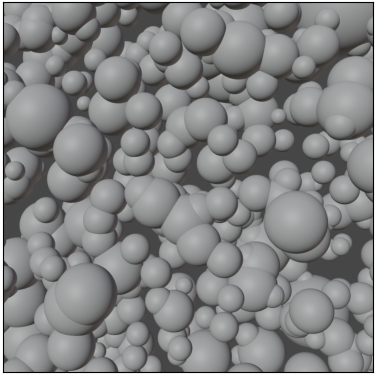
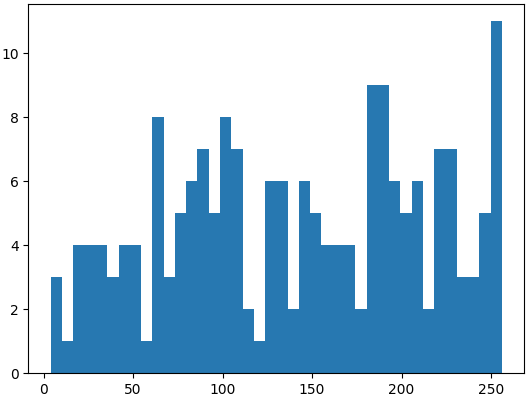


**Рис.5.8** График разности двух распределений при 40 значениях

Как видно из графиков ломанных в редких случая совпадают между собой, но присутствуют интервалы, на которых они накладываются друг на друга. Но данная метрика вычитания одного распределения из другого является грубой, поэтому любое отклонение может привести к сильному расхождению результатов.

## **5.2. Результаты метода end-to-end.**

Для данного метода был взят исходный набор данных и для каждого изображения было рассчитано распределение размеров диаметров частиц. На рисунке 5.3 представлен пример изображения, а на рисунке 5.4 гистограмма распределения.



**Рис.5.4** Пример гистограммы распределения для изображения

**Рис.5.3** Пример изображения

В качестве основной нейронной сети была взята EfficientNet из PyTorch [14]. Также был добавлен еще один полносвязный слой с 40 выходами, так как было решено взять 40 значений для распределения. В качестве функции потерь была взята L1Loss функция [14]. И все было скомпоновано в отдельные классы и методы для удобства запуска.

Обучение проходило в 5 эпох, тренировочный batch\_size = 5 изображениям, валидационный 1. Ниже приведена таблица 3 обучения и валидации (результаты loss функции.)

|  |  |  |
| --- | --- | --- |
| Номер эпохи | Значение loss функции на тренировочных данных | Значение loss функции на валидационных данных |
| 1 | 8.9244 | 0.4542 |
| 2 | 8.4770 | 0.4531 |
| 3 | 7.8683 | 0.4268 |
| 4 | 7.3990 | 0.4113 |
| 5 | 7.1089 | 0.3848 |

**Таблица 3.** Результаты второго метода

Из результатов видно, что полученные распределения в результате работы нейронной сети отличаются примерно на 38 процентов от исходных распределений, что говорит о не самой лучшей работе данного метода.

Также в качестве эксперимента также было проведено обучение и валидации распределения, состоящего из 10 значений. Ниже приведена таблица 4 обучения и валидации результатов.

|  |  |  |
| --- | --- | --- |
| Номер эпохи | Значение loss функции на тренировочных данных | Значение loss функции на валидационных данных |
| 1 | 8.8637 | 0.8002 |
| 2 | 7.4478 | 0.3693 |
| 3 | 7.2310 | 0.3382 |
| 4 | 7.1156 | 0.2985 |
| 5 | 7.0190 | 0.2805 |

**Таблица 4.** Результаты второго метода

Из результатов видно, что при распределении, состоящем из 10 значений, loss функция на влидационных данных показала 28 процентов, то есть отличается от исходного на 28 процентов. Использовать распределение, состоящее из 10 значений более перспективно, так как результаты лучше на 10 процентов по сравнению с предыдущим экспериментом. Но для более лучших результатов нужно увеличить количество итераций обучения.

## **5.3. Выводы и предложения по результатам экспериментов.**

1. Первый метод показал неплохие результаты при использовании метрики с использованием расхождения Кульбака-Лейблера [11] и наилучший результат показал, что исходное распределение и распределение, полученное на выходе нейронной сети, отличаются примерно на 10 процентов. Для улучшения результатов можно увеличить количество стадий для обучения, так как многие частицы на изображении не были найдены в результате работы нейронной сети, подбор других параметров для обучения. Также можно попробовать рассмотреть использование других нейронных сетей с одностадийным проходом для детектирования, таких, как YOLO или SSD.
2. Второй метод является более перспективным, так как мы уходим от ненужных действий, таких как выделение границ каждой частицы, что является минусом первого подхода. Но результаты эксперимента показали не самые высокие результаты. Скорее всего это связано с необученностью до конца нейронной сети. Также минусом данного подхода является долгое обучения, требующего больших ресурсов. Поэтому для улучшения результатов данного подхода можно увеличить количество итераций обучения для данной используемой сети, взять более новую нейронную сеть из класса EfficientNet - EfficientNet-B7 или постараться рассмотреть другую архитектуру нейронной сети, которая сможет показать более лучшие результаты и снизить время обучения.

# **Выводы.**

В рамках магистерской диссертации была рассмотрена задача по автоматизации получения распределения размеров частиц порошка по размерам. Эта задача является актуальной и позволяет определять качество сырья в области аддитивных технологий. Был проведен анализ данной области и реализованы 2 метода решения данной задачи с целью выбора, подходящего для использования.

Цель работы была достигнута и выполнены все поставленные задачи:

1. Был реализован генератор искусственных размеченных изображений, используемых в двух подходах.
2. Было реализована два метода, и они были протестированы на искусственном наборе данных и сделаны выводы и предложения по каждому методу.
3. По итогам сравнения двух подходов было выявлено, что оптимальным является метод end-to-end. Так как позволяет уйти от выделения границ каждой частицы, которые в результате влияли на точность первого метода.

Были реализованы программы на языке Python с использованием дополнительных открытых пакетов, таких, как Blender, PyTorch и другие.

# **Список литературы.**

1. Asante, Samuel & Olawuyi, Babatunde. (2016). Particle size distribution methods as adopted for different materials.
2. ГОСТ 23402-78 Порошки металлические. Микроскопический метод определения размеров частиц
3. Конюхов А.Л., Руководство к использованию программного комплекса ImageJ для обработки изображений: Учебное методическое пособие. – Томск: кафедра ТУ, ТУСУР, 2012. – 105 с.
4. Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do & Kaori Togashi. Convolutional neural networks: an overview and application in radiology, 22 June 2018
5. Vihar Kurama, A Review of Popular Deep Learning Architectures: ResNet, InceptionV3, and SqueezeNet, 2020
6. Ведущие алгоритмы обнаружения объектов – Режим доступа: <https://www.basic.ai/blog-post/leading-object-detection-algorithms-in-2023:-a-comprehensive-overview> – Загл. с экрана
7. Сеть региональных предложений (RPN) при обнаружении объектов – Режим доступа: <https://www.geeksforgeeks.org/region-proposal-network-rpn-in-object-detection/> - Загл. с экрана
8. Girshick, Ross. “Fast r-cnn.” Proceedings of the IEEE International Conference on Computer Vision. 2015.
9. Mask R-CNN: архитектура современной нейронной сети для сегментации объектов на изображениях – Режим доступа: <https://habr.com/ru/articles/421299/> - Загл. с экрана
10. Исследуем архитектуры сверточных нейронных сетей с помощью fast.ai-Режим доступа: https://proglib.io/p/issleduem-arhitektury-svertochnyh-neyronnyh-setey-s-pomoshchyu-fast-ai-2020-12-28 – Загл. с экрана
11. Jonathon Shlens, “Notes on Kullback-Leibler Divergence and Likelihood Theory”, arXiv e-prints, 2014
12. Google Colab – Режим доступа: <https://colab.research.google.com/> - Загл. с экрана
13. Документация к пакету Blender – Режим доступа: <https://docs.blender.org/api/current/info_overview.html> - Загл. с экрана
14. Документация к пакету PyTorch – Режим доступа: <https://pytorch.org/docs/stable/index.html> - Загл. с экрана

# **Приложение.**

**Код генерации искусственного набора данных:**

DIST=lognormal

LOC=0.5

SHAPE=0.7

N=40

CLASS=${DIST}-loc${LOC}-shape${SHAPE}

mkdir -p ${CLASS}

mkdir -p ${CLASS}/particles

PARTICLESPATH=${CLASS}/particles/particles${RENDER\_ID}.json

mkdir -p ${CLASS}/renders

RENDERPATH=${CLASS}/renders/particles${RENDER\_ID}.png

for i in $(seq 1 10000); do

PARTICLESPATH=${CLASS}/particles/particles${i}.json

RENDERPATH=${CLASS}/renders/particles${i}.png

# sample ground truth particle sizes from the given distribution

python generate\_sample.py -n ${N} --textfile ${PARTICLESPATH} --distribution ${DIST} --loc ${LOC} --shape ${SHAPE}

PARTICLESPATH=${PARTICLESPATH} RENDERPATH=${RENDERPATH} blender -b --python blender\_powder.py

done

import sys

import json

import click

import numpy as np

from scipy.stats import weibull\_min

from datetime import datetime

def fit\_weibull\_dist(loc=1, shape=0.1, nsamples=1e5):

loc = np.log(loc)

samples = np.random.lognormal(mean=loc, sigma=shape, size=nsamples)

weibull\_params = weibull\_min.fit(samples)

return weibull\_min(\*weibull\_params)

@click.command()

@click.option('-n', '--samplesize', default=10, type=int,

help='number of particles')

@click.option('-o', '--textfile', default='particles.json', type=click.Path(),

help='json output path')

@click.option('-d', '--distribution', default='lognormal',

type=click.Choice(['lognormal', 'normal', 'weibull\_fit']),

help='distribution family to sample from')

@click.option('-m', '--loc', default=0.1, type=float,

help='distribution scale parameter')

@click.option('-s', '--shape', default=0.5, type=float,

help='distribution shape parameter')

def generate\_sample(samplesize, textfile, distribution, loc, shape):

if distribution == 'lognormal':

loc = np.log(loc)

size = np.random.lognormal(mean=loc, sigma=shape, size=samplesize)

elif distribution == 'normal':

size = np.random.normal(loc=loc, scale=shape, size=samplesize)

elif distribution == 'weibull\_fit':

nsamples = 1e5

dist = fit\_weibull\_dist(loc=loc, shape=shape, nsamples=nsamples)

size = dist.rvs(size=samplesize)

else:

sys.exit('error: choose between normal and lognormal distributions')

xx = np.random.uniform(low=0, high=1, size=samplesize)

yy = np.random.uniform(low=0, high=1, size=samplesize)

zz = np.random.uniform(low=0, high=1, size=samplesize)

particles = []

for s, x, y, z in zip(size, xx, yy, zz):

particles.append({'size': s, 'x': x, 'y': y, 'z': z})

data = {

'distribution': distribution,

'loc': loc,

'shape': shape,

'timestamp': datetime.utcnow().isoformat(),

'particles': particles

}

with open(textfile, 'w') as f:

json.dump(data, f)

if \_\_name\_\_ == '\_\_main\_\_':

generate\_sample()

import bpy

import os

import sys

import json

import numpy as np

os.environ['TEXTUREPATH']="metal\_texture.jpg"

os.environ['PARTICLESPATH']=" lognormal-loc0.3-shape0.7/particies"

os.environ['RENDERPATH']="lognormal-loc0.3-shape0.7/renders"

try:

texture\_path = os.environ['TEXTUREPATH']

except KeyError:

texture\_path = os.path.expanduser('texture\_source.jpg')

print('using default texture path (set TEXTUREPATH to specify a source image)')

try:

particles\_path = os.environ['PARTICLESPATH']

except KeyError:

particles\_path = 'particles.json'

print('using default particles path (set PARTICLESPATH)')

try:

renderpath = os.environ['RENDERPATH']

except KeyError:

renderpath = os.path.expanduser('render.jpg')

print('writing render to default path ./render.jpg (set RENDERPATH)')

bpy.ops.object.delete()

scn = bpy.context.scene

new\_world = bpy.data.worlds.new("New World")

new\_world.use\_sky\_paper = True

scn.world = new\_world

bpy.data.worlds['World'].horizon\_color = (0.012, 0.003, 0.012)

world.horizon\_color = (0.012, 0.003, 0.012)

loc = (3,3,5)

light\_data = bpy.data.lights.new(name="my-light-data", type='POINT')

light\_data.energy = 100

lamp = bpy.data.objects.new(name="my-light", object\_data=light\_data)

bpy.context.collection.objects.link(lamp)

lamp.location = loc

def newlamp(name, lamptype, loc):

bpy.ops.object.add(type='LIGHT', location=loc)

lamp = bpy.context.object

lamp.name = str(name)

lamp.data.name = 'Lamp{}'.format(name)

lamp.data.type = lamptype

return lamp

locs = [(-3,3,5), (-3,-3,5), (3,-3,5)]

for i, loc in enumerate(locs):

newlamp(i, 'POINT', loc)

scene = bpy.data.scenes["Scene"]

scene.render.resolution\_x = 1024

scene.render.resolution\_y = 1024

scene.camera.rotation\_mode = 'XYZ'

scene.camera.rotation\_euler = (0.0, 0.0, 0.0)

scene.camera.location.x = 0

scene.camera.location.y = 0

scene.camera.location.z = 10

bpy.context.scene.render.engine = 'CYCLES'

mat = bpy.data.materials.new('thematerial')

mat.use\_nodes = True

texImage = mat.node\_tree.nodes.new('ShaderNodeTexImage')

texImage.image = bpy.data.images.load(texture\_path)

with open(particles\_path, 'r') as f:

dataset = json.load(f)

print('distribution: {}'.format(dataset['distribution']))

print('mean: {}'.format(dataset['loc']))

print('sigma: {}'.format(dataset['shape']))

print('timestamp: {}'.format(dataset['timestamp']))

scale = np.array([11.0, 11.0, 2.0])

for particle in dataset['particles']:

size = particle['size']

loc = np.array([particle['x'], particle['y'], particle['z']])

x, y, z = scale \* loc - scale/2

a1, a2, a3 = np.pi / 2 \* np.random.random(3)

import bpy\_extras

from mathutils import Vector

bpy.ops.mesh.primitive\_uv\_sphere\_add(

segments=64,

ring\_count=32,

radius=size,

location=(x,y,z))

render\_scale = scene.render.resolution\_percentage / 100

render\_size = (

int(scene.render.resolution\_x \* render\_scale),

int(scene.render.resolution\_y \* render\_scale),

)

coord\_3d = Vector((x, y, z))

coords\_2d = bpy\_extras.object\_utils.world\_to\_camera\_view(scene, scene.camera, coord\_3d)

bpy.ops.object.mode\_set(mode='EDIT')

bpy.ops.uv.smart\_project()

bpy.ops.object.mode\_set(mode='OBJECT')

bpy.context.object.data.materials.append(mat)

bpy.ops.object.shade\_smooth()

bpy.context.scene.render.engine = 'BLENDER\_WORKBENCH'

bpy.context.scene.render.filepath = renderpath

bpy.ops.render.render(write\_still=True)

**Код метода с использованием Fast-RCNN:**

import os

import torch

from torchvision.io import read\_image

from torchvision.ops.boxes import masks\_to\_boxes

from torchvision import tv\_tensors

from torchvision.transforms.v2 import functional as F

import torchvision

from torchvision.models.detection.faster\_rcnn import FastRCNNPredictor

model = torchvision.models.detection.fasterrcnn\_resnet50\_fpn(pretrained=True)

num\_classes = 2 # 1 class (person) + background

in\_features = model.roi\_heads.box\_predictor.cls\_score.in\_features

model.roi\_heads.box\_predictor = FastRCNNPredictor(in\_features, num\_classes)

import os

import torch

from torchvision.io import read\_image, ImageReadMode

from torchvision.ops.boxes import masks\_to\_boxes

from torchvision import tv\_tensors

from torchvision.transforms.v2 import functional as F

import json

import math

import numpy as np

class PennFudanDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, root, transforms):

self.root = root

self.transforms = transforms

self.imgs = list(sorted(os.listdir(os.path.join("/content/drive/MyDrive/Dataset/renders/", root))))

self.masks = list(sorted(os.listdir(os.path.join("/content/drive/MyDrive/Dataset/particles/", root))))

def \_\_getitem\_\_(self, idx):

img\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/", self.root, self.imgs[idx])

mask\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/", self.root, self.imgs[idx])

img = read\_image(img\_path, mode=ImageReadMode.RGB).float()

annotations\_file = open(os.path.join("/content/drive/MyDrive/Dataset/particles/", self.root, self.masks[idx]))

annotations = json.load(annotations\_file)

img\_filename = annotations['distribution']

points = annotations['particles']

size = []

tn = []

scale = np.array([11.0, 11.0, 2.0])

import bpy

import bpy\_extras

from mathutils import Vector

i=6

for el in points:

size.append(el['size'])

x = (scale[0] \* (el['x']) - scale[0]/2)

y = (scale[1] \* (el['y']) - scale[1]/2)

z = (scale[2] \* (el['z']) - scale[2]/2)

if i == 6:

scene = bpy.data.scenes["Scene"]

scene.render.resolution\_x = 1024

scene.render.resolution\_y = 1024

scene.camera.rotation\_mode = 'XYZ'

scene.camera.rotation\_euler = (0.0, 0.0, 0.0)

scene.camera.location.x = 0

scene.camera.location.y = 0

scene.camera.location.z = 10

bpy.context.scene.render.engine = 'CYCLES'

bpy.ops.mesh.primitive\_uv\_sphere\_add(

radius=points[i]['size'],

location=(x,y,z))

i += 1

coord\_3d = Vector((x, y, z))

coords\_2d = bpy\_extras.object\_utils.world\_to\_camera\_view(scene, scene.camera, coord\_3d)

x = (coords\_2d.x) \* (1024)

y = (1 - coords\_2d.y)\* (1024)

if x + (el['size']) \*128 <= 30 or x - (el['size']) \*128 >= 1000 or y + (el['size']) \*128 <=30 or y - (el['size']) \*128 >=1000:

continue

relsize=((el['size']) \*128 + z)

x1 = 0 if x - relsize < 0 else x - relsize

y1 = 0 if y - relsize < 0 else y - relsize

x2 = 1024 if x + relsize > 1024 else x + relsize

y2 = 1024 if y + relsize > 1024 else y + relsize

flag = False

repeats = []

for el2 in tn:

xA = max(x1, el2[0])

yA = max(y1, el2[1])

xB = min(x2, el2[2])

yB = min(y2, el2[3])

interArea = max(0, xB - xA + 1) \* max(0, yB - yA + 1)

boxAArea = (x2 - x1 + 1) \* (y2 - y1 + 1)

boxBArea = (el2[2] - el2[0] + 1) \* (el2[3] - el2[1] + 1)

inter1 = (interArea/boxAArea)\*100

inter2 = (interArea/boxBArea)\*100

if inter1 > 50 and el2[4]>el['z']:

flag = True

break

if inter2 > 50 and el2[4] < el['z']:

repeats.append(el2)

if not flag:

for element in repeats:

tn.remove(element)

tn.append([x1, y1, x2, y2, el['z'], el['size']])

repeats.clear()

num\_objs = len(tn)

tn2 = [elem[0:4] for elem in tn ]

boxes = torch.tensor(tn2, dtype=torch.float)

labels = torch.ones((num\_objs,), dtype=torch.int64)

image\_id = idx

area = (boxes[:, 3] - boxes[:, 1]) \* (boxes[:, 2] - boxes[:, 0])

iscrowd = torch.zeros((num\_objs,), dtype=torch.int64)

img = tv\_tensors.Image(img)

target = {}

target["boxes"] = tv\_tensors.BoundingBoxes(boxes, format="XYXY", canvas\_size=F.get\_size(img))

#target["masks"] = tv\_tensors.Mask(mask)

target["labels"] = labels

target["image\_id"] = image\_id

target["area"] = area

target["iscrowd"] = iscrowd

if self.transforms is not None:

img, target = self.transforms(img, target)

return img, target

def \_\_len\_\_(self):

return len(self.imgs)

from engine import train\_one\_epoch, evaluate

device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')

num\_classes = 2

dataset = PennFudanDataset('train', None)

dataset\_test = PennFudanDataset('test', None)

data\_loader = torch.utils.data.DataLoader(

dataset,

batch\_size=20,

shuffle=True,

num\_workers=8,

collate\_fn=utils.collate\_fn

)

data\_loader\_test = torch.utils.data.DataLoader(

dataset\_test,

batch\_size=1,

shuffle=False,

num\_workers=8,

collate\_fn=utils.collate\_fn

)

model.to(device)

params = [p for p in model.parameters() if p.requires\_grad]

optimizer = torch.optim.SGD(

params,

lr=0.0001,

momentum=0.9,

weight\_decay=0.05

)

lr\_scheduler = torch.optim.lr\_scheduler.StepLR(

optimizer,

step\_size=3,

gamma=0.1

)

num\_epochs = 10

for epoch in range(num\_epochs):

train\_one\_epoch(model, optimizer, data\_loader, device, epoch, print\_freq=10)

lr\_scheduler.step()

evaluate(model, data\_loader\_test, device=device)

torch.save(model.state\_dict(), '/content/drive/MyDrive/Dataset/model1.pth')

torch.save(model, '/content/drive/MyDrive/Dataset/model2.pth')

!pip install bpy

import matplotlib.pyplot as plt

from torchvision.utils import draw\_bounding\_boxes, draw\_segmentation\_masks

from torchvision.io import read\_image, ImageReadMode

from torchvision.ops.boxes import masks\_to\_boxes

from torchvision import tv\_tensors

images\_train\_path = list(sorted(os.listdir("/content/drive/MyDrive/Dataset/renders/test")))

images\_train\_path = images\_train\_path[:1500]

all\_pred\_boxes = []

for img in images\_train\_path:

img\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/test", img)

print(img\_path)

image = read\_image(img\_path, mode=ImageReadMode.RGB).float()

img= tv\_tensors.Image(image)

model.eval()

with torch.no\_grad():

img = img.to(device)

predictions = model([img])

pred = predictions[0]

pred\_boxes = pred["boxes"].long()

pred\_boxes\_cpu=pred\_boxes.cpu()

pred\_max = []

for el in pred\_boxes\_cpu:

pred\_max.append(max(el[2]-el[0], el[3]-el[1]))

all\_pred\_boxes.append(pred\_max)

import torch

import numpy as np

import matplotlib.pyplot as plt

import json

import torchvision.transforms.functional as F

plt.rcParams["savefig.bbox"] = 'tight'

from torchvision.utils import draw\_bounding\_boxes

from torchvision.utils import make\_grid

from torchvision.io import read\_image

from torchvision.io import read\_image, ImageReadMode

images\_train\_path = list(sorted(os.listdir("/content/drive/MyDrive/Dataset/renders/test")))

images\_train\_path = images\_train\_path[:1500]

all\_diameters = []

coy = 1

for img\_path1 in images\_train\_path:

img\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/test", img\_path1)

ch = "."

ch\_index = img\_path1.find(ch)

annot = img\_path1[:ch\_index]

annot\_path = f"/content/drive/MyDrive/Dataset/particles/test/{annot}.json"

print(annot\_path)

img = read\_image(img\_path, mode=ImageReadMode.RGB)

annotations\_file = open(annot\_path)

annotations = json.load(annotations\_file)

img\_filename = annotations['distribution']

points = annotations['particles']

size = []

tn = []

coy+=1

diameters = []

scale = np.array([11.0, 11.0, 2.0])

import bpy

import bpy\_extras

from mathutils import Vector

cout = 0

i=6

j=0

siz=0

for el in points:

size.append(el['size'])

x = (scale[0] \* (el['x']) - scale[0]/2)

y = (scale[1] \* (el['y']) - scale[1]/2)

z = (scale[2] \* (el['z']) - scale[2]/2)

if i == 6:

scene = bpy.data.scenes["Scene"]

scene.render.resolution\_x = 1024

scene.render.resolution\_y = 1024

scene.camera.rotation\_mode = 'XYZ'

scene.camera.rotation\_euler = (0.0, 0.0, 0.0)

scene.camera.location.x = 0

scene.camera.location.y = 0

scene.camera.location.z = 10

bpy.context.scene.render.engine = 'CYCLES'

bpy.ops.mesh.primitive\_uv\_sphere\_add(

radius=points[i]['size'],

location=(x,y,z))

i += 1

coord\_3d = Vector((x, y, z))

coords\_2d = bpy\_extras.object\_utils.world\_to\_camera\_view(scene, scene.camera, coord\_3d)

x = (coords\_2d.x) \* (1024)

y = (1 - coords\_2d.y)\* (1024)

if x + (el['size']) \*128 <= 30 or x - (el['size']) \*128 >= 1000 or y + (el['size']) \*128 <=30 or y - (el['size']) \*128 >=1000:

continue

relsize=((el['size']) \*128 + z)

x1 = 0 if x - relsize < 0 else x - relsize

y1 = 0 if y - relsize < 0 else y - relsize

x2 = 1024 if x + relsize > 1024 else x + relsize

y2 = 1024 if y + relsize > 1024 else y + relsize

flag = False

repeats = []

repeats2 = []

for el2 in tn:

xA = max(x1, el2[0])

yA = max(y1, el2[1])

xB = min(x2, el2[2])

yB = min(y2, el2[3])

interArea = max(0, xB - xA + 1) \* max(0, yB - yA + 1)

boxAArea = (x2 - x1 + 1) \* (y2 - y1 + 1)

boxBArea = (el2[2] - el2[0] + 1) \* (el2[3] - el2[1] + 1)

inter1 = (interArea/boxAArea)\*100

inter2 = (interArea/boxBArea)\*100

if inter1 > 50 and el2[4]>el['z']:

flag = True

break

if inter2 > 50 and el2[4] < el['z']:

repeats.append(el2)

if not flag:

for element in repeats:

tn.remove(element)

tn.append([x1, y1, x2, y2, el['z'], el['size']])

repeats.clear()

num\_objs = len(size)

tn2 = [elem[0:4] for elem in tn ]

diameters = [(elem[5]\*128)\*2 for elem in tn]

all\_diameters.append(diameters)

import matplotlib.pyplot as plt

import numpy as np

random\_val = []

random\_val2 = []

for diameters in all\_diameters:

(hst, bin\_edges) = np.histogram(diameters, bins=10)

random\_values = np.random.choice(bin\_edges[:-1], size=500, p=hst/sum(hst))

random\_val.append(random\_values)

for pred\_max in all\_pred\_boxes:

(hst2, bin\_edges2) = np.histogram(pred\_max, bins=10)

random\_values2 = np.random.choice(bin\_edges2[:-1], size=500, p=hst2/sum(hst2))

random\_val2.append(random\_values2)

kl\_div1 = 0

kl\_div2 = 0

for i in range(len(random\_val)):

hist2 = np.abs(random\_val2[i]).astype('float64')

diameters2 = np.abs(random\_val[i]).astype('float64')

hist2 /= np.sum(hist2)

diameters2 /= np.sum(diameters2)

kl\_divergence = np.sum(diameters2 \* np.log(diameters2 / hist2))

kl\_div1 += kl\_divergence

hist2 = np.abs(random\_val2[i]).astype('float64')

diameters2 = np.abs(random\_val[i]).astype('float64')

hist2 /= np.sum(hist2)

diameters2 /= np.sum(diameters2)

kl\_divergence = np.sum(hist2 \* np.log(hist2 / diameters2))

kl\_div2 += kl\_divergence

print(kl\_div1/len(random\_val))

print(kl\_div2/len(random\_val))

import seaborn as sns

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(16, 10))

sns.lineplot(x=np.arange(len(hst)), y=hst-hst2, label='разность двух распределений (Line)', color='darkgreen', ax=ax,linewidth = 10)

plt.title('Probability Density Functions')

plt.xlabel('Index')

plt.ylabel('Value')

plt.legend()

plt.show()

**Код метода с использованием EfficientNet:**

!pip install efficientnet-pytorch

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision.transforms as transforms

from efficientnet\_pytorch import EfficientNet

from torch.utils.data import DataLoader

class HistogramValDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, root):#images, histograms):

self.root = root

self.imgs = list(sorted(os.listdir(os.path.join("/content/drive/MyDrive/Dataset/renders/", root))))[:200]

self.masks =list(sorted(os.listdir(os.path.join("/content/drive/MyDrive/Dataset/particles/", root))))[:200]

def \_\_len\_\_(self):

return len(self.imgs)

def \_\_getitem\_\_(self, idx):

img\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/", self.root, self.imgs[idx])

mask\_path = os.path.join("/content/drive/MyDrive/Dataset/renders/", self.root, self.imgs[idx])

img = read\_image(img\_path, mode=ImageReadMode.RGB).float()

annotations\_file = open(os.path.join("/content/drive/MyDrive/Dataset/particles/", self.root, self.masks[idx]))

annotations = json.load(annotations\_file)

img\_filename = annotations['distribution']

points = annotations['particles']

size = []

tn = []

diameters = []

scale = np.array([11.0, 11.0, 2.0])

import bpy

import bpy\_extras

from mathutils import Vector

cout = 0

i=6

j=0

siz=0

for el in points:

size.append(el['size'])

x = (scale[0] \* (el['x']) - scale[0]/2)

y = (scale[1] \* (el['y']) - scale[1]/2)

z = (scale[2] \* (el['z']) - scale[2]/2)

if i == 6:

scene = bpy.data.scenes["Scene"]

scene.render.resolution\_x = 1024

scene.render.resolution\_y = 1024

scene.camera.rotation\_mode = 'XYZ'

scene.camera.rotation\_euler = (0.0, 0.0, 0.0)

scene.camera.location.x = 0

scene.camera.location.y = 0

scene.camera.location.z = 10

bpy.context.scene.render.engine = 'CYCLES'

bpy.ops.mesh.primitive\_uv\_sphere\_add(

radius=points[i]['size'],

location=(x,y,z))

i += 1

coord\_3d = Vector((x, y, z))

coords\_2d = bpy\_extras.object\_utils.world\_to\_camera\_view(scene, scene.camera, coord\_3d)

x = (coords\_2d.x) \* (1024)

y = (1 - coords\_2d.y)\* (1024)

if x + (el['size']) \*128 <= 30 or x - (el['size']) \*128 >= 1000 or y + (el['size']) \*128 <=30 or y - (el['size']) \*128 >=1000:

continue

relsize=((el['size']) \*128 + z)

x1 = 0 if x - relsize < 0 else x - relsize

y1 = 0 if y - relsize < 0 else y - relsize

x2 = 1024 if x + relsize > 1024 else x + relsize

y2 = 1024 if y + relsize > 1024 else y + relsize

flag = False

repeats = []

repeats2 = []

for el2 in tn:

xA = max(x1, el2[0])

yA = max(y1, el2[1])

xB = min(x2, el2[2])

yB = min(y2, el2[3])

interArea = max(0, xB - xA + 1) \* max(0, yB - yA + 1)

boxAArea = (x2 - x1 + 1) \* (y2 - y1 + 1)

boxBArea = (el2[2] - el2[0] + 1) \* (el2[3] - el2[1] + 1)

inter1 = (interArea/boxAArea)\*100

inter2 = (interArea/boxBArea)\*100

if inter1 > 50 and el2[4]>el['z']:

flag = True

break

if inter2 > 50 and el2[4] < el['z']:

repeats.append(el2)

if not flag:

for element in repeats:

tn.remove(element)

tn.append([x1, y1, x2, y2, el['z'], el['size']])

repeats.clear()

num\_objs = len(size)

tn2 = [elem[0:4] for elem in tn ]

diameters = [(elem[5]\*128 + elem[4])\*2 for elem in tn]

(hst, bin\_edges) = np.histogram(diameters, bins=10)

hst\_norm = hst/np.sum(hst)

img = tv\_tensors.Image(img)

return img, hst\_norm

class HistogramClassifier(nn.Module):

def \_\_init\_\_(self, num\_classes):

super(HistogramClassifier, self).\_\_init\_\_()

self.backbone = EfficientNet.from\_pretrained('efficientnet-b0')

self.fc = nn.Linear(1280, num\_classes)

def forward(self, x):

x = self.backbone.extract\_features(x)

x = nn.functional.adaptive\_avg\_pool2d(x, 1).reshape(x.shape[0], -1)

x = self.fc(x)

return x

def train(model, dataloader, criterion, optimizer):

model.train()

running\_loss = 0.0

for i, (images, histograms) in enumerate(dataloader):

images = images.to(torch.device('cuda:0'))

histograms = histograms.to(torch.device('cuda:0'))

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, histograms)

loss.backward()

optimizer.step()

running\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

return running\_loss/len(dataloader)

def validate(model, dataloader, criterion):

model.eval()

running\_loss = 0.0

with torch.no\_grad():

for i, (images, histograms) in enumerate(dataloader):

images = images.to(torch.device('cuda:0'))

histograms = histograms.to(torch.device('cuda:0'))

outputs = model(images)

loss = criterion(outputs, histograms)

running\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

return running\_loss/len(dataloader)

!pip install bpy

dataset\_train = HistogramDataset('train')

dataset\_val = HistogramValDataset('test')

train\_loader = DataLoader(dataset\_train, batch\_size=5, shuffle=True)

val\_loader = DataLoader(dataset\_val, batch\_size=1)

model = HistogramClassifier(num\_classes=40)

device = torch.device('cuda:0')

model = model.to(device)

criterion = nn.L1Loss(reduction='sum')

optimizer = optim.Adam(model.parameters(), lr=0.0001)

for epoch in range(5):

train\_loss = train(model, train\_loader, criterion, optimizer)

val\_loss = validate(model, val\_loader, criterion)

print(f'Epoch {epoch+1}: train loss = {train\_loss:.4f}, val loss={val\_loss:.4f}')

torch.save(model.state\_dict(), '/content/drive/MyDrive/Dataset/model\_method2.pth')

torch.save(model, '/content/drive/MyDrive/Dataset/model\_method2.pth')