**Яшин Никита Андреевич**

**03.09.2023**

**Отчет по тестовому заданию**

***1. Реализовать модуль генерации данных***

Генерация изображений была разбита на две подзадачи – генерация фигур и генерация изображений. Все необходимые классы и функции описаны в figures.py.

**Для генерации фигур** был написан базовый класс Figure, содержащий общие методы для создания фигуры, отрисовки на изображении и предоставлении параметров фигуры для дальнейшего создания .json. Для разных фигур различается только метод create, который добавляет уникальное поведение при создании конкретного типа фигуры, за исключением фигуры круга, где была необходимость переопределить еще методы, так как круг не является многоугольником.

Построение всех фигур основано на получение параметров x, y, width, height извне, и генерации фигуры под заданные параметры. Так, треугольник генерируется из трех случайных точек с координатами x∈[0,1], y∈[0,1], повороту треугольника на случайный угол и масштабирование с параллельным переносом так, чтобы он соответствовал заданным параметрам.

Ромб и гексагон генерируются в виде шаблона, поворачиваются на случайный угол, а после этого масштабируются под заданные ширину и высоту с учетом того, чтобы ромб остался ромбом, а гексагон – правильным. После этого происходит параллельный перенос в необходимые координаты.

Круг генерируется в центре заданных координат с радиусом, подходящим под заданные границы. После генерации любой фигуры, возможна ситуация, что, чтобы сохранить форму, фигура не была подогнана под заданные ширину и высоту, поэтому данные параметры корректируются на основе созданной фигуры.

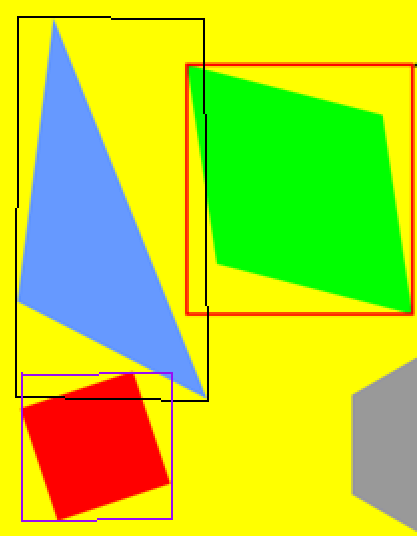
Проблема такого подхода – отсутствие гарантии того, что фигура не будет меньше 25 пикселей по одной из сторон. Однако гарантируется, что ни одна из сторон фигуры не будет более 150. Для исправления необходимо сделать иной принцип генерации фигур, а также, скорее всего, потребуется подход, когда сначала генерируется фигура, а потом высчитываются её характеристики. Данный подход не был выбран из-за принципа создания всего изображения.

**Для генерации изображений** был написан отдельный класс Drawer, который содержит определенные метаданные, необходимые для создания изображений. Их можно редактировать для получения разных эффектов.

Метод генерации изображения данного класса создает пустое изображение, а потом последовательно генерирует несколько прямоугольников с координатами. После этого, он проверяет, можно ли добавить фигуру с таким описывающим прямоугольником на изображение. Если нельзя – генерирует новые параметры, если можно – создает новую фигуру с данными параметрами и добавляет её на изображение.

Одними из метаданных является количество попыток добавить фигуры на изображение, а также вероятность того, что попытка будет пропущена. Данные параметры были подобраны экспериментально – без нескольких попыток добавить случайную фигуру, количество изображений с 5 фигурами пренебрежимо мало, а без пропуска попыток, но с 10 попытками добавить фигуры, подавляющее большинство изображений имеют по 5 фигур.

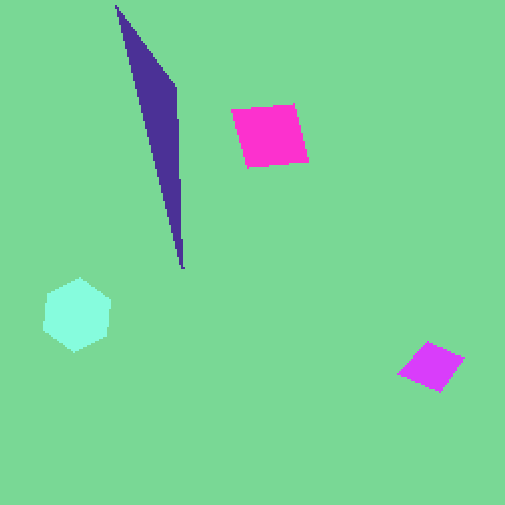
Проблемой данного подхода является то, что не пеерсекаются не сами фигуры, а их описывающие прямоугольники. Так, получить изображение аналогичное примеру не представляется возможным:



Решением данной проблемы может являться применение более сложных алгоритмов детекции коллизии, а также добавление фигур с множественными проверками на коллизию и попытками масштабирования и сдвига по разным осям. Однако данный подхода является более ресурсозатратным и трудоемким, хотя и позволяет применить более сложные алгоритмы создания фигур. Было принято решение использовать первый алгоритм из-за минусов второго.

Кроме этого, не была реализована проверка совместимости цветов изображений и фигур. Так, могут быть сгенерированы фигуры с цветом, идентичным цвету фона. Однако, это может быть использовано как «белый шум» в тестовой выборке.

**Код генерации 100 изображений и json файлов** с описаниями находится в generate100.py, а сами изображения и файлы – в папке ./images. Пример изображения и описания к нему:



***2. Детекция примитивных геометрических фигур на изображении***

Для детекции фигур на изображениях было необходимо использовать нейронную сеть. Для этого необходимо собрать и разметить данные. Генерация изображений и разметки была реализована в виде генерации «на лету». Разметка изображения состоит из массива из 20 значений – 4 значения для 5 фигур. Если на изображении менее 5 фигур, то оставшиеся значения равны нулю.

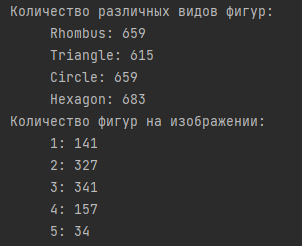
Такой подход позволяет работать с фиксированным размером выхода, так как модели неизвестно количество фигур на изображении, однако сильно влияет на итоговую точность. Вместо этого, одним из более оптимальных подходов было бы использовать несколько нейросетевых моделей в формате ансамбля. Первая модель определяла бы количество фигур на изображении, и передавала бы изображение в одну из пяти нейросетей, обученных специально на заданное количество фигур.

Кроме этого, можно было бы использовать подход «сканирования изображения по фрагментам», с помощью одной нейросети, настроенной на нахождение фигур. Такая модель также могла бы успешно определять тип фигуры. Однако, было бы необходимо решить проблему с масштабированием фигур – так как они могут быть разных размеров.

Оптимальным подходом при решении конкретной задачи было бы использование исключительно алгоритмического подхода – так как все пиксели очень четко распределены по цветам, можно собрать статистику, какой цвет граничит с другими – это будет единственный цвет, при этом принадлежащий фону. Если все же соприкосновение фигур возможно, то цвет, который больше всего граничит с другими – цвет фона. В данном случае исключения пренебрежимо маловероятны.

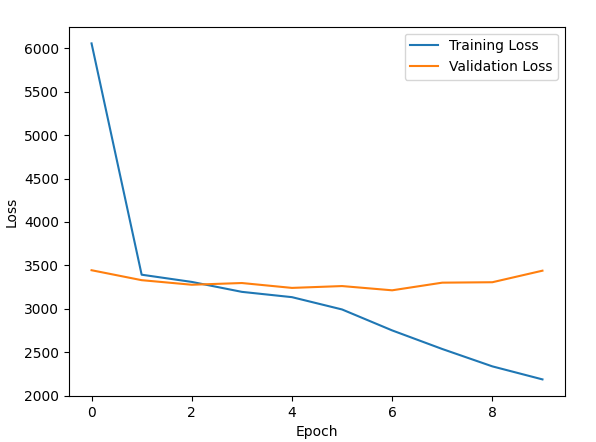
Однако, согласно задаче, было необходимо сделать одну нейронную сеть для детекции фигур. Поэтому, был выбран не самый оптимальный вариант решения данной задачи.

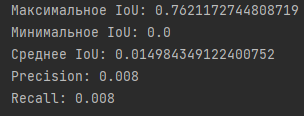
В файле figures\_detection.py была написана функция генерации датасета, а также сбора статистики по данному датасету. Кроме этого, была реализована функция подсчета метрики IoU. Было решено не добавлять аугментации в датасет, потому что применение аугментаций требовало бы комплексного подхода по исправлению разметки данных для измененных изображений. Это могло потребовать большой ручной работы, не позволяя использовать уже готовые функции как tensorflow.keras.preprocessing.image.ImageDataGenerator.random\_transform(). Пример данных о сгенерированном датасете:



В качестве архитектуры была выбрана сверточная модель, так как данная модель является наилучшим выбором при работе с изображениями. Опытным путем была подобрана модель из четырех пар convolution+maxpooling слоев, а также из трех dense слоев после выравнивания до одномерного слоя. Результаты данной нейронной сети являются плохими, так как изначально был выбран неправильный подход, а также потому что данная архитектура не подразумевает работу с динамическим количеством фигур на изображении.

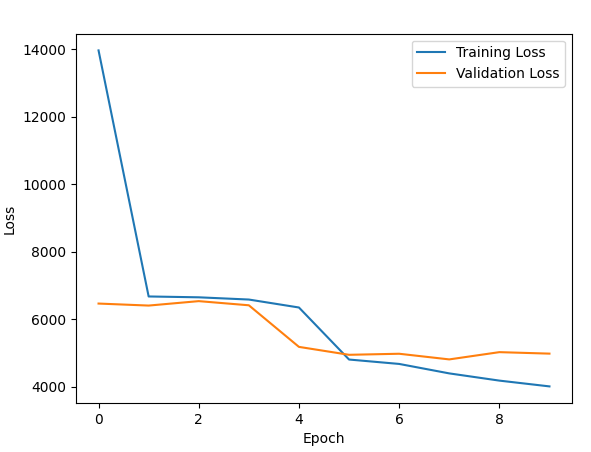
Для обучения генерировался датасет из 1000 изображений, а для проверки – еще 100 изображений. График потерь при обучении на 10 эпохах и значения метрик:

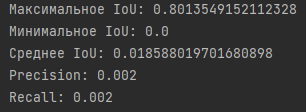




По графику видно, что потери являются большими и после 2-3 эпохи обучение перестает иметь результат. Кроме этого, метрики показывают полную несостоятельность данного подхода при решении текущей проблемы.

Кроме этого, данная модель была проверена на датасете, в котором все изображения имели по 5 фигур:





Среднее значение IoU выросло, однако незначительно. Также, значения precision и recall не изменились значительно. Таким образом, установка фиксированного количества фигур с малой долей вероятности может быть критичной для решения данной проблемы.

***3. Исследования по дообучению нейронной сети***

Для исследования по дообучению были взяты модель и функции из п.2. Также была реализована возможность исключить из генерации изображений фигуру гексагон. Было реализовано сохранение модели в файл и последующая загрузка её оттуда в качестве «точки сохранения», а также цикл, расширяющий датасет, дообучающий модель и проверяющий точности.