Оглавление

[Актуальность 2](#_Toc198544842)

[Выбор алгоритма Faster R-CNN 2](#_Toc198544843)

[Для решения данной задачи было использовать алгоритм R-CNN алгоритм, а именно самый оптимальный в соотношении точности и быстродействия – Faster R-CNN. 2](#_Toc198544844)

[Постановка задачи 4](#_Toc198544845)

[2.1. Проблемы "нативной" MobileNetV2 в Faster R-CNN 5](#_Toc198544846)

[2.2. Необходимые модификации 5](#_Toc198544847)

[Реализация RPN-модели 7](#_Toc198544848)

[Принцип работы RPN модели 8](#_Toc198544849)

[Подсчет метрик 9](#_Toc198544850)

[Принцип работы IoU (Intersection over Union) — метрики для оценки точности детекции объектов 9](#_Toc198544851)

[Формула IoU 9](#_Toc198544852)

[Применение RoI Poolling и применение пространства признаков 11](#_Toc198544853)

[Принцип работы ROI Pooling 11](#_Toc198544854)

[Архитектура нейронной сети 13](#_Toc198544855)

[Фронтенд модели 15](#_Toc198544856)

[Обучение RPN модели 21](#_Toc198544857)

[Классификатор общий модели (решение задачи классификации) 26](#_Toc198544858)

[Тестирование алгоритма 37](#_Toc198544859)

Актуальность

На сегодняшний день даже на производствах, где вмешательство человека сведено к минимуму, необходимо соблюдение техник безопасности для избегания несчастный случаев, травм и, как следствие, непредвиденных расходов на сотрудников, поэтому возникает потребность в использовании автоматизированных систем обнаружения рабочей экипировки, которые смогут в режиме реального времени отслеживать её наличие.

В качестве принципы обнаружения было решено использовать машинное зрение вместо иных методов, таких как вшиваемых RFID чипов, поскольку это в реализации обойдется гораздо дешевле нежели установка в чипов в экипировку.

Выбор алгоритма Faster R-CNN

Для решения данной задачи было использовать алгоритм R-CNN алгоритм, а именно самый оптимальный в соотношении точности и быстродействия – Faster R-CNN.

**1. Практическая значимость**

* **Безопасность на производстве** – автоматический контроль ношения СИЗ (касок, масок, спецодежды) снижает травматизм и риск штрафов.
* **Автоматизация мониторинга** – замена ручных проверок на компьютерное зрение экономит время и ресурсы.
* **Интеграция с системами видеонаблюдения** – возможность внедрения в реальном времени.

**2. Преимущества Faster R-CNN**

* **Высокая точность** – двухэтапная архитектура (Region Proposal Network + детектор) обеспечивает точное обнаружение даже мелких объектов (например, защитных очков).
* **Устойчивость к сложным сценам** – эффективно работает в условиях:
  + Частичного перекрытия объектов
  + Разного освещения (цеха, склады)
  + Разнообразия СИЗ (цвет, форма, размер)
* **Гибкость** – можно дообучать на новых типах СИЗ (например, для разных отраслей).

**3. Сравнение с другими методами**

На рынке существует 3 самых востребованных модели для решения задач object detection. Нижи приведены и технические достоинства и недостатки:

| **Метод** | **Плюсы** | **Минусы для задачи СИЗ** |
| --- | --- | --- |
| **Faster R-CNN** | Высокая точность, хорошая работа с мелкими объектами | Вычислительно затратный |
| **YOLO** | Быстрая обработка видео | Менее точен для мелких СИЗ (перчатки, наушники) |
| **SSD** | Баланс скорости и точности | Требует больших данных для обучения |

Таблица 1 – сравнение технических плюсов и минусов моделей object detection

Исходя из таблицы, можно сделать вывод, что выбранный алгоритм будет не самым эффективным для решения задачи, но благодаря специфической архитектуры нейронной сети можно достигнуть оптимальных показателей производительности алгоритма, а также возможно интегрировать систему в автономный микроконтроллер (такой как Raspberry Pie).

Постановка задачи

Поскольку для решения данной задачи был выбран алгоритм Faster R-CNN, который подразумевает решение двух задач.

Первая задача, которую необходимо решить это регрессионная, которая будет разделять пиксельное изображение на RoI фрагменты, которые будут постоянного формата, тем самым облегчая задачу регрессии, уменьшив объем данных. Стоит упомянуть, что разделение фрагментов реализовывается с помощью отдельной нейронной сети, отвечающей за процесс энкодинга изображение, чтобы далее можно было с помощью алгоритма RoI Poolling можно сопоставить фрагменты из пиксельного пространства в пространство признаков.

Второй задачей будет классификация вышеупомянутых фрагментов на 8 классов, связанных с PPE объектами (и 1 void-объект), чтобы в дальнейшем можно было получить тензор с классов, соответствующих выделенным рамкам.

Так же необходимо построить алгоритм вычислений метрик и обучения на основе тензорных операций из tensorflow, чтобы можно было

Поскольку сам алгоритм хоть и является самым быстрым из всей линейки R-CNN моделей, но существуют более оптимизированные модели, поэтому для компенсации этого недостатка будет использована видоизмененная архитектура нейронной сети MobileNetV2 как для RPN-модели, так и для классификатора, что в свою очередь обеспечивает достаточно высокое быстродействие, а также возможность полной самостоятельной работы на микроконтроллерах. Хоть это и повлияет негативно на точность, но в связи с использованием достаточно эффективного алгоритма небольшими потерями в точности ради быстродействия можно пренебречь.

MobileNetV2 — это эффективная свёрточная нейронная сеть (CNN), разработанная для мобильных и embedded-устройств. Её ключевые особенности:

Faster R-CNN — двухэтапный детектор:

1. **Region Proposal Network (RPN)** генерирует регионы интереса (RoIs).
2. **Head Network** классифицирует RoIs и уточняет bounding boxes.

### **2.1. Проблемы "нативной" MobileNetV2 в Faster R-CNN**

* **Слабая пространственная детализация**
  + MobileNetV2 сильно уменьшает разрешение (до 7×7), теряя мелкие объекты (например, каски или перчатки).
* **Недостаточная семантическая мощность**
  + Лёгкая архитектура хуже выделяет сложные признаки для классификации СИЗ.
* **Ограниченная receptive field**
  + Depthwise-свёртки менее эффективны для больших объектов.

### **2.2. Необходимые модификации**

Для интеграции MobileNetV2 в Faster R-CNN применяют:

#### ****1. Замена бэкбона (Feature Extractor)****

* **Удаление последних слоёв**
  + Исходная MobileNetV2 заканчивается слоем 7×7×1280 — это слишком грубое представление для детекции.
  + **Решение**: Использовать выход из промежуточных слоёв (например, C4 или C5 из MobileNetV2 с разрешением 14×14).
* **Добавление FPN (Feature Pyramid Network)**
  + FPN комбинирует признаки разных масштабов (например, 28×28, 14×14, 7×7), улучшая детекцию мелких СИЗ.

#### ****2. Адаптация RPN****

* **Anchor Boxes**
  + Подбор anchor-боксов под размеры СИЗ (например, 32×32 для перчаток, 64×64 для касок).
* **Замена свёрток RPN**
  + Стандартные 3×3 свёртки в RPN можно заменить на **разделимые** (depthwise + pointwise) для ускорения.

#### ****3. Оптимизация Head Network****

* **Lightweight Box Head**
  + Замена тяжёлых полносвязных слоёв на **глобальное среднее pooling + 1×1 свёртки**.

Дальнейшая задача сводится к правильному видоизменению данной архитектуры отдельно для каждой задачи.

Реализация RPN-модели

В данной главе рассматривается реализация Region Proposal Network (RPN) — ключевого компонента Faster R-CNN, адаптированного для обнаружения средств индивидуальной защиты (СИЗ) на изображениях. Основная задача RPN — генерация регионов интереса (RoIs), которые затем классифицируются и уточняются основной сетью. Код реализован на TensorFlow и включает:

* Использование MobileNetV2 в качестве бэкбона.
* Гибкую систему обучения с применением категориальной кросс энтропии
* Интеграцию с датасетом в формате YOLO.

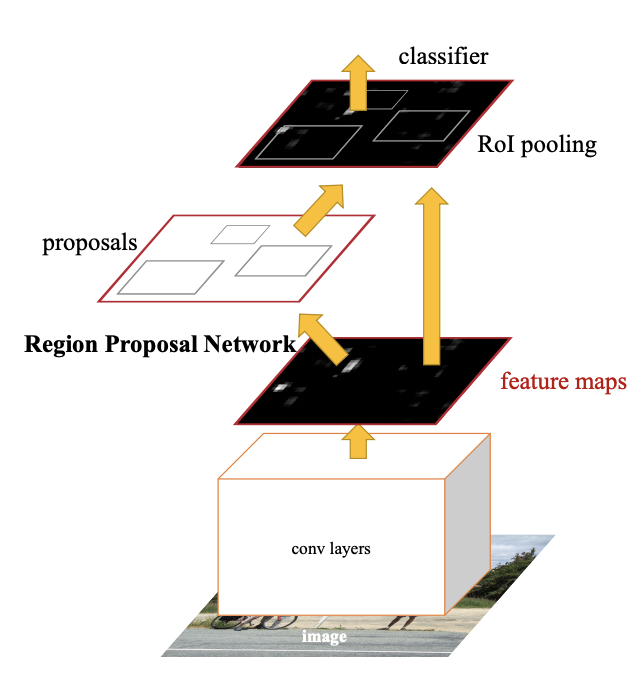


Рисунок 1 – схема принципа работы Faster R-CNN

Принцип работы RPN модели

На вышеуказанном рисунка пока что нас интересует лишь Region Proposal Network, который будет решать задачу регрессии, а именно выделять необходимые объекты в рамки (координаты левого верхнего угла, ширину и высоту прямоугольника), тем самым решая две задачи: регрессия и бинарная классификация, поскольку модель определяет стоит ли подавать данную область в следующую модель или нет.

После чего строится карта признаков с помощью сверточного энкодера, далее, основываясь на рамках предполагаемого объекта с RPN модели и карте признаков энкодера, вырезаются блоки карты для дальнейшей классификации. Данный метод видоизменения формата рамки интересов называется roi poolling.

Подсчет метрик

По мимо метода обработки данных необходимо определить метод вычисления ошибки RPN модели.

В данном случае будут использоваться 2 метода ее подсчета:

1. Регрессионная ошибка (мера отклонения параметров рамки от истинных)
2. Классификационная ошибка (мера некорректного обнаружение интересующих объектов)

Так как модель всегда предсказывает некоторое фиксирование количество рамок, в качестве примера предположим, что их 100, для подсчета ошибки необходимо каждую рамку сопоставить с каждым “идеальным”, иными словами подсчитать метрику IoU.

### **Принцип работы IoU (Intersection over Union) — метрики для оценки точности детекции объектов**

**IoU (Intersection over Union)** — это метрика, которая измеряет степень пересечения между предсказанной ограничивающей рамкой (bounding box) и истинной (ground truth). Она используется в задачах детекции объектов (object detection) и сегментации (segmentation) для оценки качества работы модели.

## **Формула IoU**

IoU=

* **A** — предсказанный bounding box.
* **B** — истинный bounding box (ground truth).
* **A∩B*A*∩*B*** — площадь пересечения двух рамок.
* **A∪B*A*∪*B*** — площадь объединения (сумма площадей минус пересечение).

После вышесказанных манипуляций нужно оставить лишь самые правдоподобные рамки по следующим критериям, которые мы задаем сами.

В данной модели используются следующие критерии:

1. Если метрика для определенной рамки составляет *> 0.7*, то данный прогноз можно считать положительным результатом.
2. Если метрика для определенной рамки составляет *< 0.3*, то данных прогноз можно считать отрицательным результатом.
3. При всех остальных значениях метрики будет считаться, что модель затрудняется ответить, то есть наблюдается некий промежуточный участок.

Таким образом подсчитывается классификационная ошибка модели. Теперь модель может обучаться находить объекты, осталось лишь вычислить метрику для определения рамок.

Для регрессионной метрики было решено использовать не обыкновенную L2 метрику, а Smooth L1 loss, которая имеет меньшую чувствительность к выбросам. Данная метрика рассчитывается для всех рамок, прошедших классификацию, т.е. с максимальным пересечением истинных рамок.

Применение RoI Poolling и применение пространства признаков

После выявления рамок с необходимыми объектами для передачи данных в классификатор осталось лишь перевести вырезанные картинки из пиксельного пространства в пространство признаков, что даст нам экономию вычислительных ресурсов, поскольку в карте признаков будут находится только необходимые данные в отличие от пиксельных изображений.

Для решения данной задачи используется RoI Pooling.

**ROI Pooling (Region of Interest Pooling)** — это операция, используемая в задачах детекции объектов для преобразования признаков из произвольных областей изображения (ROI) в фиксированный размер.

### **Принцип работы ROI Pooling**

1. **Определение ROI (Region of Interest)**
   * На вход подаётся карта признаков (feature map), полученная от свёрточной нейронной сети (CNN), и набор регионов интереса (ROI) — прямоугольные области, где могут находиться объекты.
   * Каждый ROI задаётся координатами (x, y, ширина, высота) и соответствует предполагаемому объекту на изображении.
2. **Разбиение ROI на ячейки фиксированного размера**
   * Каждый ROI делится на сетку из H × W ячеек (например, 7×7).
   * Если ROI имеет размер (h, w), то каждая ячейка будет размером (h/H, w/W).
3. **Максимальное пулирование (Max Pooling) внутри каждой ячейки**
   * Для каждой ячейки вычисляется максимальное значение признаков (max pooling).
   * Это позволяет получить фиксированный выход (H × W), даже если исходные ROI имеют разный размер.

Объяснение принципа на примере:

Допустим, у нас имеется изображение, где выделены интересующие области, но появляется проблема передачи данных в классификационную модель, поскольку размерность таких областей переменна (рис. 2).

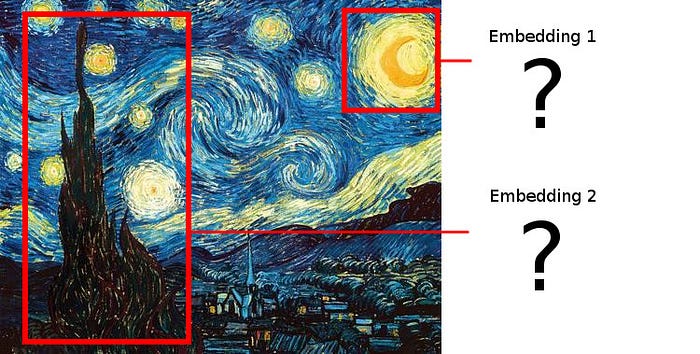


Рисунок 2 – выделение интересующих областей

Поэтому необходимо решить проблему переменной размерности изображения, сделав данные постоянной размерности, сопоставимые с количеством входных нейронов классификатора.

Для начала можно избавиться от лишних и зашумленных данных, перейдя от пиксельного пространства к более “компактному”, которое будет содержать в себе только необходимую для нейронной сети информацию. Такое пространство называется пространством признаков, которое воспринимается лишь нейронной сетью. Хоть эмбеддинг и уменьшился в размере, но главная задача, связанная с переменной размерностью, не была решена.

Для окончательного решения используется метод max pooling, который делить эмбеддинг на области одинаковой формы и находит максимальное значение в каждой области, добавляя его в новый видоизмененный эмбеддинг. Тем самым мы получается вектор из пространства признаков постоянного размера, который уже без всяких проблем будет обработан следующей моделью (рис. 3).

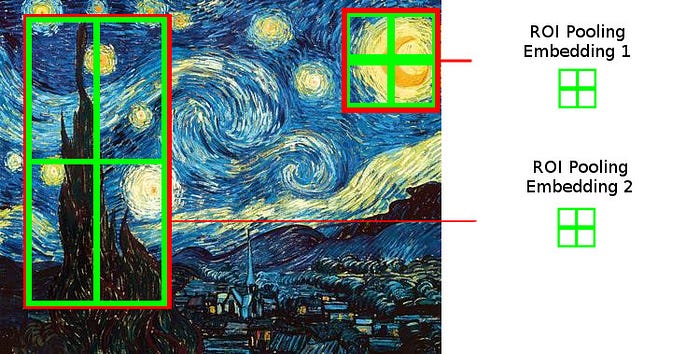


Рисунок 3 – получение конечного эмбеддинга из матрицы признаков с помощью промежуточного max pooling

После чего мы получаем эмбеддинги изображение одинакового размера, которые теперь можно передать в классификационную модель.

Архитектура нейронной сети

В качестве энкодера будет использована предобученная оптимизированная сверточная нейронная сеть MobileNetV2 с удаленными классификационными слоями (119 слоев из 154 было использовано)

(см. лис. 1).

*Листинг 1 – создание энкодер-модели на основе MobileNetV2*

base\_model = tf.keras.applications.MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

num\_layers\_to\_keep = 119

output\_tensor = base\_model.layers[num\_layers\_to\_keep - 1].output

clipped\_model = tf.keras.Model(inputs=base\_model.input, outputs=output\_tensor)

for layer in clipped\_model.layers[:80]:

layer.trainable = False

Устанавливаем для энкодера размерность входных данных как 224х224 пикселя с 3 значениями (RGB). Так же обучаемость первых 80 слоев запрещена, для того чтобы сохранить структуру предобученной модели.

Далее на основе всего вышеупомянутого строится модель с двумя выходами:

* + 1. Выходы для решения бинарной классификации
    2. Выходы для координат рамок

То есть результаты работы энкодер-модели будут подаваться на сверточные слои с двумя функциями активации соответственно: softmax и линейная (лис. 2).

Листинг 2 – реализация полноценной PRN модели

inputs = Input((224, 224 ,3))

x = clipped\_model(inputs)

box\_probabilities = Conv2D(2, kernel\_size=1, activation = 'softmax')(x)

box\_coords = Conv2D(4, kernel\_size=1, activation = 'linear')(x)

outputs = [box\_probabilities, box\_coords]

RP\_network = keras.Model(inputs, outputs)

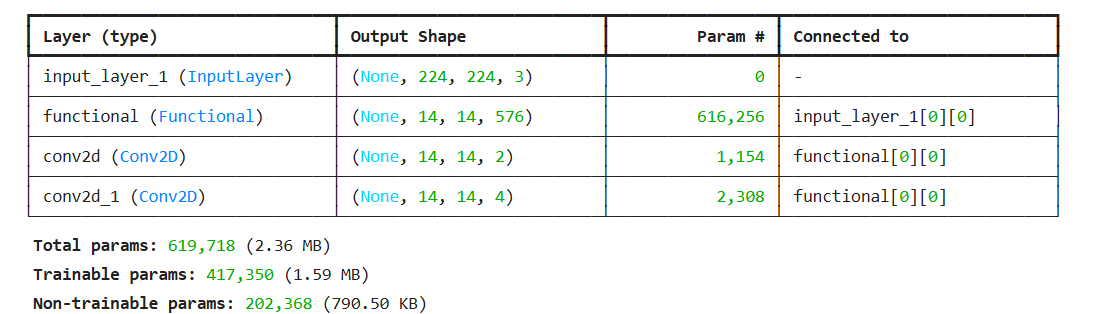


Таблица 2– схема с параметрами модели

После создания бекенда данной модели можно преступить к проектированию управляющего класса нейронной сети.

Фронтенд модели

После создание фундаменты перед обучением модели осталось лишь создать класс управления, где самым сложным методом будет непосредственно с метод вычисления ошибки.

Сам метод стоит разобрать подробнее с математической точки зрения:

Допустим, мы имеем два предсказанных тензора:

* + 1. Классификационные (вероятность принадлежности к классу 1 – интересующий класс и вероятность принадлежности к классу 0 – не интересующий)
    2. Тензор с координатами и параметрами рамок соответственно.

Необходимо эти данные сопоставить с истинными параметрами рамок (см. лис. 5).

Листинг 5 – Примерные тензоры для вычисления ошибки

tf.Tensor(

[[0.4 0.6 ] (тензор вероятностей)

[0.8 0.2 ]

[0.45 0.55 ]

[0.1 0.9 ]], shape=(4, 2), dtype=float32)

tf.Tensor(

[[ 1. 1. 5. 5. ]

[ 7. 7. 9. 9. ] (тензор предсказанных рамок)

[ 1. 2. 3. 4. ]

[50. 60. 70. 80.]], shape=(4, 4), dtype=float32)

tf.Tensor(

[[ 1. 7. 9. 8. ] (тензор истинных рамок)

[54. 65. 72. 81.]], shape=(2, 4), dtype=float32)

Далее находим количество рамок предсказанных и истинных и проводим над рамочными тензорами манипуляции (см. лис. 6)

Листинг 6 – видоизменение тензора предсказанных рамок

r\_pred = tf.repeat(tf.expand\_dims(pred\_boxes, axis = 0), num\_gt, axis = 0)

r\_pred = tf.reshape(r\_pred, [-1, 4])

tf.Tensor(

[[ 1. 1. 5. 5.]

[ 7. 7. 9. 9.]

[ 1. 2. 3. 4.]

[50. 60. 70. 80.]

[ 1. 1. 5. 5.]

[ 7. 7. 9. 9.]

[ 1. 2. 3. 4.]

[50. 60. 70. 80.]], shape=(8, 4), dtype=float32)

Мы изменяем размерность, а затем повторяем параметры рамок столько раз, сколько существует истинных рамок. Это делается для того, чтобы в дальнейшем можно было легко, используя лишь тензорные операции, посчитать метрику IoU. После чего проделываем туже операцию для параметров истинных рамок только создаем столько копий, чтобы количество каждого параметра в тензоре было одинаково и размерность массива данных тоже совпадала с размеров предсказанного массива.

Листинг 7 – посчитанный тензор с помощью метрики IoU

tf.Tensor(

[0. 0.2857143 0. 0. 0. 0.

0. 0.55284554], shape=(8,), dtype=float32)

Далее после видоизменений можно сопоставить эти значения с каждой предсказанной рамкой (см. лис. 8).

Листинг 8 – сопоставление IoU метрик с рамками

iou = tf.reshape(iou, [num\_gt, num\_pred])

tf.Tensor(

[[0. 0.2857143 0. 0. ]

[0. 0. 0. 0.55284554]], shape=(2, 4), dtype=float32)

max\_iou\_arg = tf.argmax(iou, axis = 0)

corresponding\_gt = tf.gather(gt, max\_iou\_arg)

iou\_rates = tf.reduce\_max(iou, axis = 0)

Далее сортируем наши метрики (см. лис. 9)

Листинг 9 – сортировка метрик с помощью функции argsort и сортировка остальных данных

indexes = tf.argsort(iou\_rates, direction='ASCENDING')

corresponding\_pred = tf.gather(corresponding\_pred, indexes)

corresponding\_prob = tf.gather(corresponding\_prob, indexes)

corresponding\_gt = tf.gather(corresponding\_gt, indexes)

iou\_rates = tf.gather(iou\_rates, indexes)

Далее займемся классификационной ошибкой. Сначала необходимо создать маски для двух возможных классов (см. 10 лис.).

Листинг 10 – создание классификационных масок

tf.Tensor(

[[0. 1.]

[0. 1.]

[0. 1.]

[0. 1.]], shape=(4, 2), dtype=float32)

tf.Tensor(

[[1. 0.]

[1. 0.]

[1. 0.]

[1. 0.]], shape=(4, 2), dtype=float32)

positive\_true = tf.repeat(tf.expand\_dims(tf.one\_hot(1, 2), axis = 0), num\_pred, axis = 0)

negative\_true = tf.repeat(tf.expand\_dims(tf.one\_hot(0, 2), axis = 0), num\_pred, axis = 0)

regression\_loss = self.hubert(corresponding\_pred, corresponding\_gt)

tf.Tensor([2.875 3.375 1.5 2.5 ], shape=(4,), dtype=float32)

Вычисляем регрессионную ошибку для сдвига рамок с помощью smooth L1 (Hubert) (см. лис. 10).

Далее используем метод count\_in\_tensor, чтобы вычислить количество рамок, прошедших классификацию (см. лис. 11).

Листинг 11 – метод подсчета количества рамок, прошедших бинарную классификацию, и его использование

def count\_in\_tensor(self, tensor, th, more = False):

if more:

true\_positions = tf.cast(tensor > th, dtype = tf.float32)

else:

true\_positions = tf.cast(tensor < th, dtype = tf.float32)

return tf.reduce\_sum(true\_positions)

num\_less\_than\_low\_th = self.count\_in\_tensor(iou\_rates, low\_th, more = False)

num\_more\_than\_high\_th =self.count\_in\_tensor(iou\_rates, high\_th, more= True)

Потом находим размерность тензора рамок, который равен либо количеству прошедших классификацию рамок, либо размеру минибатча (см. лис. 12).

Листинг 12 – нахождение размерности тензоров

num\_more\_than\_high\_th = tf.maximum(num\_more\_than\_high\_th, minibatch\_size//2)

num\_less\_than\_low\_th = tf.maximum(num\_less\_than\_low\_th, minibatch\_size//2)

Создаем итоговые классификационные маски, то есть тензоры, состоящие только из нулей и единиц (см. лис. 13).

Листинг 13 – создание итоговых классификационных масок

positive\_sample\_mask = tf.concat([tf.zeros([num\_pred - num\_more\_than\_high\_th], dtype = tf.float32),

self.random\_sample(num\_more\_than\_high\_th, minibatch\_size//2)], axis = 0)

negative\_sample\_mask = tf.concat([self.random\_sample(num\_less\_than\_low\_th, minibatch\_size//2),

tf.zeros([num\_pred - num\_less\_than\_low\_th], dtype = tf.float32)], axis = 0)

tf.Tensor([0. 0. 0. 1.], shape=(4,), dtype=float32)

tf.Tensor([0. 1. 0. 0.], shape=(4,), dtype=float32)

Пример итоговых классификационных масок

В создание вышеуказанного тензора используется метод random\_sample,

которые генерируют фиксированное количество единичек на случайных

позициях тензора (см. лис. 14).

Листинг 14 – генератор тензоров с случайно разбросанными единичкам

def random\_sample(self, length, num\_ones):

indices = tf.range(length)

shuffled\_indices = tf.random.shuffle(indices)

return tf.cast((shuffled\_indices<num\_ones), dtype = tf.float32)

Далее вычисляем функцию потерь для рамок с отсутствием объектов и с наличием, а затем, суммируя, находим общую ошибку по всем элементам изображения (см. лис. 15).

Листинг 15 – подсчет итоговой (общей ошибки)

positive\_loss = (self.crossentropy(positive\_true, corresponding\_prob) + regression\_loss \* alpha) \* positive\_sample\_mask

negative\_loss = (self.crossentropy(negative\_true, corresponding\_prob)) \* negative\_sample\_mask

loss = tf.reduce\_sum(negative\_loss + positive\_loss)

Стоит отметить, что такая итоговая метрика не имеет конкретного ограничения сверху, поскольку описывает общее отклонение от истины по всем объектам изображения.

Благодаря тому, что все вычислительные операции происходили с помощью встроенных тензорных методов tensorflow, можно построить некий вычислительных граф (устойчивое математическое выражение), обработку которого можно перенести на видеочип, тем самым увеличив быстродействие в разы.

Обучение RPN модели

После полноценного создания нейронной сети следует перейти к ее непосредственному обучению. Для этого создаем генератор датасета и тривиальный алгоритм обработки эпох обучения с выводом графики зависимости ошибки от количества итераций и сохранением весов лучшей модели (см. лис. 16 и лис. 17 соответственно).

Листинг 16 – генератор датасета для RPN модели

def generator():

path = 'C:\\Users\\Ильяс\\.cache\\roboflow\\street\_work\\update\_\_annotations.txt'

data = open(path, 'r').readlines()

for el in data:

path\_image = el.replace('\n','')

path\_label = 'C:\\Users\\Ильяс\\.cache\\roboflow\\street\_work\\train\\labels\\' +\

path\_image.split('\\')[-1].replace('\n','') + '.txt'

label = open(path\_label, 'r').readlines()

bbox = []

for e in label:

if len(e) <= 1:

break

li = e.split()[1:]

x,y = float(li[0]) \* 224, float(li[1]) \* 224

w, h = float(li[2]) \* 224, float(li[3]) \* 224

bbox.append([x - w / 2, y - h / 2, x + w / 2, y + h / 2])

if len(bbox) == 50:

break

else:

image = tf.image.resize(mpimg.imread(path\_image),(224, 224)) / 255

yield (image,bbox)

Листинг 17 – алгоритм реализации обучения

best\_loss = 100000

from IPython.display import clear\_output

for epoch in range(epochs):

midloss = 0

for step, x in enumerate(dataset):

image, bounding\_boxes = x

loss, test1, test2 = model.train\_step(image, bounding\_boxes)

midloss += loss

if(step % 100 == 99):

clear\_output(wait=True)

print('эпоха ' + str(epoch))

print('ошибка: ' + str(float(midloss/100)))

print(test1, test2)

hist = np.append(hist, float(midloss/100))

plt.plot(np.arange(0,len(hist)), hist)

plt.show()

if (midloss/100 < best\_loss):

best\_loss = midloss/100

model.network.save('best.h5')

midloss = 0

После выполнения 500 эпох получается следующий график обучения

(см. рис. 4)

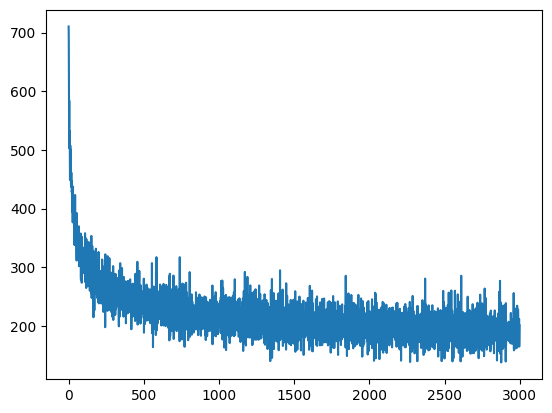


Рисунок 3 – график обучения RPN модели

Стоит ещё раз упомянуть, что в связи с специфичностью подсчета метрики ошибки, поэтому значение 200 вполне приемлемое. Самые главные факторы адекватности процесса обучения для данного случая являются:

* 1. Сам факт убывания ошибки
  2. Стремление ошибки к нулю
  3. Характер убывания функции ошибки (экспоненциальный)

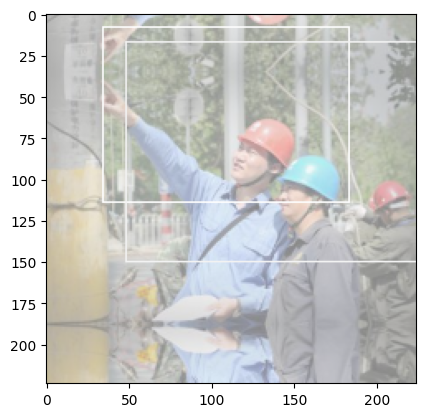


Рисунок 4 – пример работы RPN модели

Алгоритм RoI Pooling для удобства обучения и тестирования было решено реализовать в начале классификатора.

Так же в нейронной сети возникает проблема такая, как генерация большого количества рамок, несущих одинаковую информацию, что неприемлемо для дальнейшей обработки. Для решения этого недостатка было решено использовать алгоритм Non maximum suppression, который, удаляя все лишние рамки, оставляет только те, которые имеют наибольшую правдоподобность (см. лис. 18).

*Листинг 18 – метод non maximum suppression*

def non\_maximum\_supression(self, boxes, scores, th = 0.6):

num\_boxes = boxes.shape[0]

axis\_A = tf.repeat(tf.expand\_dims(boxes, axis = 0), num\_boxes, axis = 0)

axis\_B = tf.tile(tf.expand\_dims(boxes, axis=1), multiples=[1, num\_boxes, 1])

axis\_A = tf.reshape(axis\_A, [-1, 4])

axis\_B = tf.reshape(axis\_B, [-1, 4])

cross\_iou = self.IoU(axis\_A, axis\_B)

resulted\_boxes = []

resulted\_scores = []

for i in range(num\_boxes):

discard = False

for j in range(num\_boxes):

if (cross\_iou[i \* num\_boxes + j] > th):

if(scores[j, 1] > scores[i, 1]):

discard = True

break

if not discard:

resulted\_boxes.append(boxes[i])

resulted\_scores.append(scores[i])

resulted\_boxes = tf.cast(resulted\_boxes, dtype = tf.float32)

resulted\_scores = tf.cast(resulted\_scores, dtype = tf.float32)

return (resulted\_boxes, resulted\_scores)

Классификатор общий модели (решение задачи классификации)

После разработки фундаментальной составляющей модели, благодаря которой осуществляется выделения самой необходимой информации из пиксельного изображения, осталось лишь сопоставить фрагменты в пространстве признаков с классами, интересующих объектов, добавив класс 0, который будет описывать класс пустоты (void-объект).

Для начала стоит реализовать нейронную сеть-энкодер для перехода из пиксельного пространства в признаковое. Для этого воспользуемся вышеупомянутой архитектурой MobileNetV2 с 154 слоями (см. лис 19).

*Листинг 19 – реализация нейронной сети – энкодер*

base\_model = tf.keras.applications.MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

num\_layers\_to\_keep = 154

output\_tensor = base\_model.layers[num\_layers\_to\_keep - 1].output

clipped\_model = tf.keras.Model(inputs=base\_model.input, outputs=output\_tensor)

for layer in clipped\_model.layers[:50]:

layer.trainable = False

Энкодер принимает на вход пиксельное изображение 224х224x3 пикселя и выдает тензор в пространстве признаков 7x7x1280. Далее с помощью алгоритма RoI Pooling данный массив будет разбит на элементарные. Стоит упомянуть, что первым 50 слоям запрещается возможность обучения для сохранение общей структуры нейронной сети (см таб. 2).

После чего необходимо сопоставить фрагменты в пиксельном пространстве с фрагментами признакового пространства. На данный момент имеется лишь признаковая сетка всего изображения и пиксельный фрагмент. Первым делом нужно нормализовать параметры рамки, то есть перевести их из диапазона (0, 224) в (0, 1), далее создаем одномерный массив, содержащий индексы боксов. Затем применяем встроенную в tensorflow операцию crop\_and\_resize, которая, получая все вышеупомянутые параметры, находит соответствующие фрагменты в признаковой сетке, вырезает их и, находя в каждой ячейка максимальное значение, записывает его в новый тензор, тем самым получая объект данных размерностью 3x3x1280, соответствующих конкретной рамке объекта (см. лис 20).

Главным преимуществом данного метода является то, что в конечном итоге получается тензор с постоянной размерностью, так как объекты на изображениях могут быть различного размера, поэтому задача классификации достаточно сильно упрощается.

*Листинг 20 – реализация метода roi pooling*

def roi\_pool(self, boxes, feature\_map, pool\_size=3, image\_size=224):

# Нормализуем координаты боксов к [0,1]

boxes\_normalized = boxes / image\_size

box\_indices = tf.zeros([tf.shape(boxes)[0]], dtype=tf.int32)

# Добавляем размерность батча к feature map

batched\_feature\_map = tf.expand\_dims(feature\_map, axis=0)

# Применяем операцию

pooled\_regions = tf.image.crop\_and\_resize(

batched\_feature\_map,

boxes\_normalized,

box\_indices,

[pool\_size, pool\_size] )

return pooled\_regions

После чего необходимо создать генератор датасета, который по одному пиксельному изображению будет подавать в энкодер (см. лис. 20)

*Листинг 21 – генератор датасета для классификационной модели*

def generator():

# Путь к файлу с изображениями

annotations\_path ='C:\\Users\\Ильяс\\.cache\\roboflow\\street\_work\\update\_\_annotations.txt'

# Читаем список изображений

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

image\_paths = [line.strip() for line in f.readlines() if line.strip()]

for image\_path in image\_paths:

# Полный путь к изображению

full\_image\_path = image\_path

# Соответствующий файл с разметкой (YOLO формат)

label\_path = os.path.join(

'C:\\Users\\Ильяс\\.cache\\roboflow\\street\_work\\train\\labels',

os.path.basename(image\_path) + '.txt'

)

# Загружаем изображение

try:

image = preprocess\_image(full\_image\_path)

image = tf.image.resize(image, (224, 224))

except:

continue # Пропускаем битые изображения

# Читаем разметку

boxes = []

classes = []

if os.path.exists(label\_path):

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

for line in f:

line = line.strip()

if not line:

continue

# Парсим YOLO формат: class\_id x\_center y\_center width height

parts = line.split()

class\_id = int(parts[0])

x\_center, y\_center = float(parts[1]), float(parts[2])

width, height = float(parts[3]), float(parts[4])

# Конвертируем в формат [x\_min, y\_min, x\_max, y\_max]

x\_min = (x\_center - width/2) \* 224

y\_min = (y\_center - height/2) \* 224

x\_max = (x\_center + width/2) \* 224

y\_max = (y\_center + height/2) \* 224

boxes.append([x\_min, y\_min, x\_max, y\_max])

classes.append([class\_id])

# Добавляем void-боксы, если нужно

if boxes:

boxes\_tensor = tf.convert\_to\_tensor(boxes, dtype=tf.float32)

void\_boxes = generate\_void\_boxes(boxes\_tensor, n\_void=2)

if void\_boxes is not None and void\_boxes.shape[0] > 0:

boxes.extend(void\_boxes.numpy().tolist())

classes.extend([[0]] \* void\_boxes.shape[0]) # Класс 0 для void-боксов

# Приводим к фиксированному количеству боксов

if len(boxes) > fixed\_num\_boxes:

boxes = boxes[:fixed\_num\_boxes]

classes = classes[:fixed\_num\_boxes]

elif len(boxes) < fixed\_num\_boxes:

# Дополняем нулевыми боксами

pad\_size = fixed\_num\_boxes - len(boxes)

boxes.extend([[0, 0, 0, 0]] \* pad\_size)

classes.extend([[0]] \* pad\_size)

# Конвертируем в тензоры

boxes\_tensor = tf.convert\_to\_tensor(boxes, dtype=tf.float32)

classes\_tensor = tf.convert\_to\_tensor(classes, dtype=tf.int32)

yield image, boxes\_tensor, classes\_tensor

После создания всех подготовительных модели и функций следует преступить к созданию самого классификатора. На вход он будет получать тензор размерностью 3х3х1280 (после обработки методом RoI Pooling) и иметь слой Flatten и 2 слой Dense. Было решено вместо слоя свертки использовать слой Dense на первом месте, что дает значительный пророст в скорости обучения.

Связь между двумя слоями Dense осуществляется с помощью функции активации Relu, а на выходном слое используется функция softmax (см. лис. 21).

*Листинг 22 – создание структуры классификатора*

inputs = Input((3, 3, 1280))

x = inputs

x = Flatten()(x)

x = Dense(128, activation = 'relu')(x)

outputs = Dense(9, activation = 'softmax')(x)

classifier\_network = keras.Model(inputs, outputs)

Несмотря на то, что по сравнению RPN модель и энкодером данная модель достаточно мала в объеме, но благодаря двух слоям Dense имеет большое количество весов (см. таб. 2).

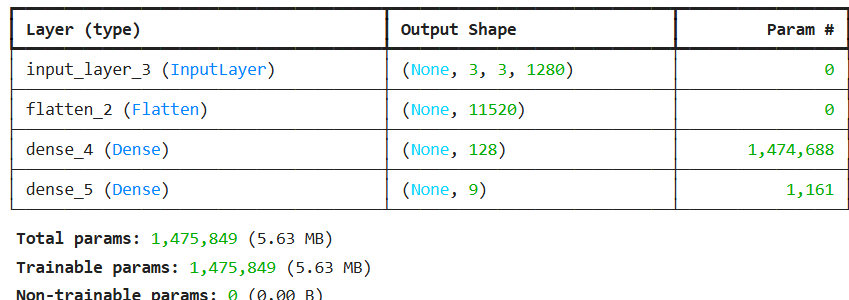


Таблица 2 – структура классификационной модели

Для подсчета метрики ошибки будет использоваться стандартная категориальная кросс энтропия (см. лис. 23).

*Листинг 23 – атрибуты класса управления классификатором*

self.network = network

self.base\_network = base\_network

self.lr = 1e-5

self.step\_counter = 0

self.colors = tf.concat([[[0,0,0]], tf.random.uniform([9, 3], 0, 1)], axis=0)

self.optimizer = tf.keras.optimizers.Adam(self.lr)

self.base\_optimizer = tf.keras.optimizers.Adam(1e-5)

self.crossentropy = tf.keras.losses.CategoricalCrossentropy()

Рассмотрим атрибуты подробно: в network передается сама классифицирующая нейронная сеть; base\_network предназначена для энкодера оригинального изображения; lr – коэффициент обучения нейронной сети; step\_counter – счетчик итераций обучения; colors – тензор, который содержит цвета, соответствующие для каждой рамки класса; base\_optimizer содержит в себе экземпляр оптимизатора по типу “Адам”; crossentropy предназначен для экземпляра, подсчитывающего ошибку при обучении по типу “категориальной кросс энтропии”.

После объявления атрибутов остается реализовать основные вспомогательные методы. В первую очередь был реализован упрощенный метод RoI Polling, который описан ранее (см. лис. 24).

*Листинг 24 – реализация метода roi pooling*

def roi\_pool(self, boxes, feature\_map, pool\_size=3, image\_size=224):

# Нормализуем координаты боксов к [0,1]

boxes\_normalized = boxes / image\_size

# Создаем индексы для каждого бокса (все относятся к 0-му изображению в батче)

box\_indices = tf.zeros([tf.shape(boxes)[0]], dtype=tf.int32)

# Добавляем размерность батча к feature map

batched\_feature\_map = tf.expand\_dims(feature\_map, axis=0)

# Применяем операцию

pooled\_regions = tf.image.crop\_and\_resize(

batched\_feature\_map,

boxes\_normalized,

box\_indices,

[pool\_size, pool\_size]

)

return pooled\_regions

Далее создадим метод, рисующий рамки на исходном изображении (см. лис. 25).

*Листинг 25 – реализация draw\_box*

def draw\_box(self, image, box, color=(255, 0, 0)):

image = cv2.rectangle(image,

(int(tf.clip\_by\_value(box[0], 0, 244)), int(tf.clip\_by\_value(box[1], 0, 244))),

(int(tf.clip\_by\_value(box[2], 0, 244)), int(tf.clip\_by\_value(box[3], 0, 244))),

color, 1)

return image

В конце реализации управляющего класса создадим методы тестирования и обучения, которые оперируют с ранее созданными методами (см. лис. 26 и 27).

*Листинг 26 – реализация метода тестирования классификатора*

def testing(self):

for data in dataset.take(1):

image, bounding\_boxes, classes = data

image = tf.expand\_dims(image, axis=0)

feature\_map = self.base\_network(image)

rois = self.roi\_pool(bounding\_boxes, feature\_map[0])

pred\_probs = self.network(rois)

print(pred\_probs)

pred\_classes = tf.argmax(pred\_probs, axis=1)

print(classes[:, 0])

print(pred\_classes)

image = ((image+1)/2).numpy()[0]

for box\_n in range(bounding\_boxes.shape[0]):

r = float(self.colors[pred\_classes[box\_n]].numpy()[0])

g = float(self.colors[pred\_classes[box\_n]].numpy()[1])

b = float(self.colors[pred\_classes[box\_n]].numpy()[2])

image = self.draw\_box(image, bounding\_boxes[box\_n], color=(r, g, b))

plt.imshow(image)

*Листинг 27 – метод реализации одной итерации обучения классификатора*

@tf.function

def train\_step(self, image, bounding\_boxes, classes):

image = tf.expand\_dims(image, axis=0)

classes = classes[:, 0]

with tf.GradientTape() as tape, tf.GradientTape() as base\_tape:

feature\_map = self.base\_network(image, training=True)

rois = self.roi\_pool(bounding\_boxes, feature\_map[0])

pred\_probs = self.network(rois, training=True)

gt\_probs = tf.one\_hot(classes, 9)

loss = self.crossentropy(gt\_probs, pred\_probs)

gradients = tape.gradient(loss, self.network.trainable\_variables)

base\_gradients = base\_tape.gradient(loss, self.base\_network.trainable\_variables)

self.optimizer.apply\_gradients(zip(gradients, self.network.trainable\_variables))

self.base\_optimizer.apply\_gradients(zip(base\_gradients, self.base\_network.trainable\_variables))

return loss

Далее создаем алгоритм обучения (см. лис. 28)

*Листинг 28 – алгоритм обучения классификатора*

epochs = 500

hist = np.array(np.empty([0]))

best\_loss = 100000

from IPython.display import clear\_output

for epoch in range(epochs):

midloss = 0

for step, x in enumerate(dataset):

image, bounding\_boxes, classes = x

loss = model.train\_step(image, bounding\_boxes, classes)

midloss += loss

if(step % 100 == 99):

clear\_output(wait=True)

print('эпоха ' + str(epoch))

print('ошибка: ' + str(float(midloss/100)))

hist = np.append(hist, float(midloss/100))

plt.plot(np.arange(0,len(hist)), hist)

plt.show()

if (midloss/100 < best\_loss):

best\_loss = midloss/100

model.network.save('b\_classifier.h5')

model.base\_network.save('b\_base\_classifier.h5')

midloss = 0

В результате обучения получается следующий график (см. рис. 5).

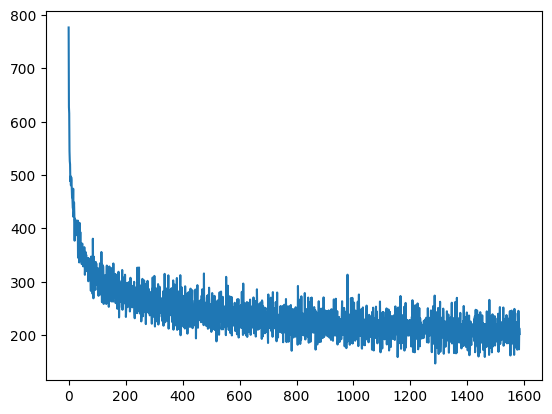


Рисунок 5 – график обучения классификатора

Тестирование алгоритма

После создания