

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Постановка задачи	7
2 Анализ существующих архитектурных решений	8
2.1 Задача классификации объекта на изображении	8
2.2 Задача локализации объектов на изображении	12
3 Исследование архитектур для классификации объектов на изображении	17
3.1 Сравнение архитектур	17
3.2 Маленький объём размеченных данных и бинарная классификация	20
3.3 Многоклассовая классификация	28
3.4 Результаты сравнения	30
4 Исследование архитектур для локализации объектов на изображении	32
4.1 Сравнение архитектур	32
4.2 Сравнение на наборе данных COCO 2017	35
4.3 Сравнение на произвольно выбранных изображениях	37
4.4 Результаты сравнения	41
ЗАКЛЮЧЕНИЕ	43
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	44
ПРИЛОЖЕНИЕ А Сравнение архитектур	46
ПРИЛОЖЕНИЕ Б Предобработка данных	47
ПРИЛОЖЕНИЕ В Работа с моделями для классификации	49
ПРИЛОЖЕНИЕ Г Классификация на маленьком наборе данных	55
ПРИЛОЖЕНИЕ Д Классификация на части ImageNet	57

ВВЕДЕНИЕ

Данная работа относится к области компьютерного зрения.

Компьютерное зрение — это научное направление в области искусственного интеллекта, в частности робототехники, и связанные с ним технологии получения изображений объектов реального мира, их обработки и использования полученных данных для решения разного рода прикладных задач без участия (полного или частичного) человека.

Современные задачи компьютерного зрения разделяют на четыре вида, им однозначно не сопоставлены русскоязычные термины, поэтому во избежание неточностей представим исходную классификацию на английском языке:

- Classification (далее классификация) — классификация изображения по типу объекта, которое оно содержит;
- Semantic segmentation (далее сегментация) — определение всех пикселей объектов определённого класса или фона на изображении. Если несколько объектов одного класса перекрываются, их пиксели никак не отделяются друг от друга;
- Object detection (далее локализация) — обнаружение всех объектов указанных классов и определение охватывающей рамки для каждого из них;
- Instance segmentation — определение пикселей, принадлежащих каждому объекту каждого класса по отдельности;

В данной работе исследованию подлежат задачи классификации и локализации объектов на изображении, так как они наиболее актуальны для робототехнических систем. Определение положения предметов в пространстве необходимо как для навигации мобильного робота, так и для позиционирования манипулятора на конвейере.

За последнее десятилетие в связи с наращиванием вычислительных мощностей и доступностью большого объёма данных стало активно развиваться глубокое обучение, сейчас каждый может собрать в интернете набор

данных и обучить на нём свою нейронную сеть, каждый день появляются новые модели, поэтому актуальной проблемой становится не разработка новой архитектуры, а нахождение ранее созданной модели, подходящей для поставленной задачи компьютерного зрения.

1 Постановка задачи

Целью данной работы является исследование современных популярных архитектур нейронных сетей для классификации и локализации объекта на изображении.

Необходимо проанализировать существующие решения и из них выбрать наиболее перспективные. Для выбранных решений провести сравнение – рассмотреть работу моделей на разных по типу данных (маленькое или большое количество тренировочных данных, бинарная или многоклассовая классификация) и сравнить полученные характеристики, такие как скорость обучения, число параметров, точностные метрики.

Итогом исследования являются рекомендации по использованию тех или иных архитектур, в зависимости от специфики конкретной задачи.

2 Анализ существующих архитектурных решений

В данной главе рассмотрим существующие архитектуры для выбранных задач классификации и локализации объектов, выберем по три решения для дальнейшего их исследования и сравнения.

2.1 Задача классификации объекта на изображении

Классификация объекта сводится к оценке вероятности того, что объект принадлежит к каждому из заданных классов.

2.1.1 LeNet

Архитектура, предложенная Яном ЛеКуном в 1998 г. [1] (рисунок 1) для распознавания рукописных цифр (задача MNIST). Сеть принимает на вход изображение 32×32 px. Состоит из двух свёрточных слоёв с последующим макспулингом и трёх полно связных слоёв

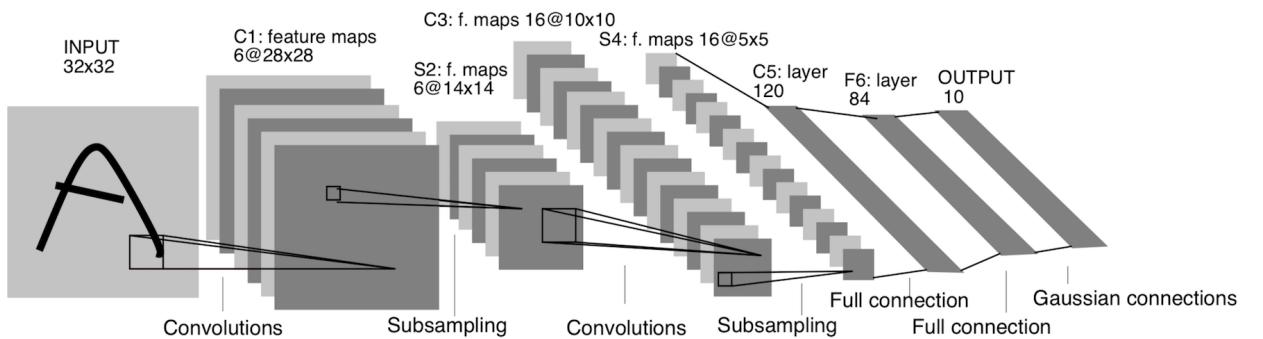


Рисунок 1 – Архитектура LeNet

2.1.2 AlexNet

Архитектура, предложенная в 2012 г. [1] (рисунок 2) для решения задачи ImageNet 1000. Сеть принимает на вход изображение $224 \times 224 \times 3$. Состоит из шести свёрточных слоёв и трёх макспулингов, завершается тремя полно связными слоями. Последовательные свёртки 3×3 использовались для экономии памяти в сравнении со свёрткой 5×5 или 7×7 .

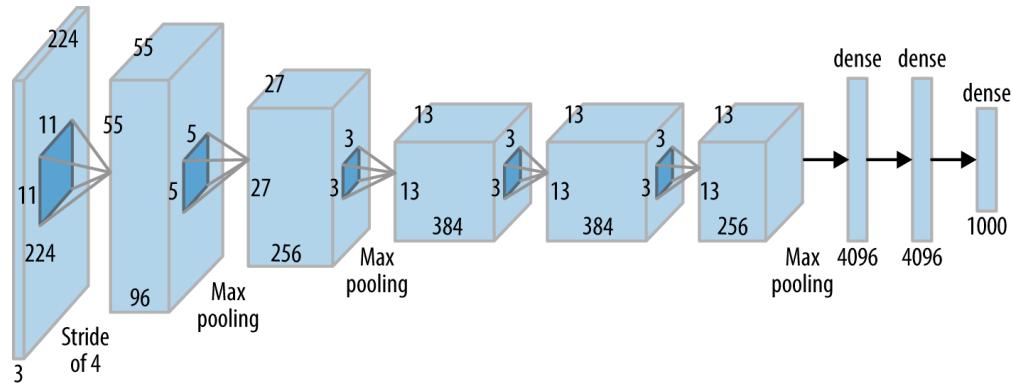


Рисунок 2 – Архитектура AlexNet

2.1.3 VGG

Архитектура, предложенная Visual Geometry Group в 2014 г. [1] (рисунок 3). Использовалась для распознавания изображений, отличается большей глубиной, что приводит к необходимости поэтапного обучения (дообучения с добавлением новых слоёв).

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Рисунок 3 – Архитектура VGG

2.1.4 GoogLeNet

Архитектура, предложенная компанией Google [1] (рисунок 4). Развивалась для распознавания изображений, отличительной особенностью является использование “inception” блоков и введение вспомогательных функций потерь.

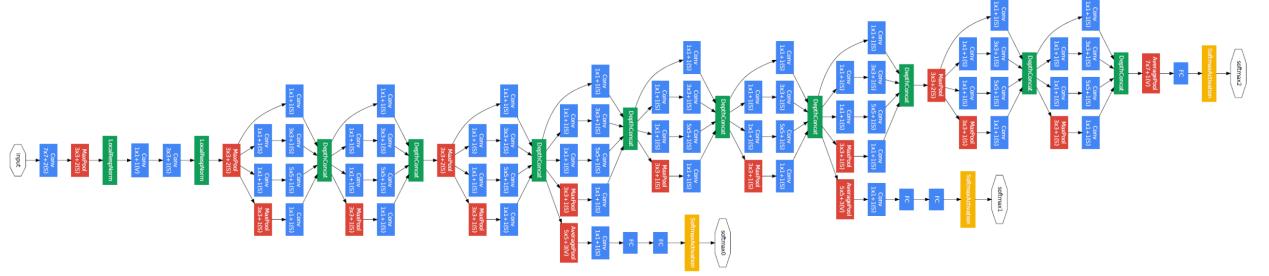


Рисунок 4 – Архитектура GoogLeNet

2.1.5 ResNet

Архитектура, предложенная компанией Microsoft [1] (рисунок 5), отличающаяся использованием “Residual” блоков и отсутствием полно связанных слоёв на выходе (кроме непосредственно одного последнего).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
average pool, 1000-d fc, softmax						
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Рисунок 5 – Архитектуры ResNet

2.1.6 ResNeXt

Отличие этих архитектур от ResNet заключается в наличии параллельных путей внутри Residual блоков (рисунок 6).

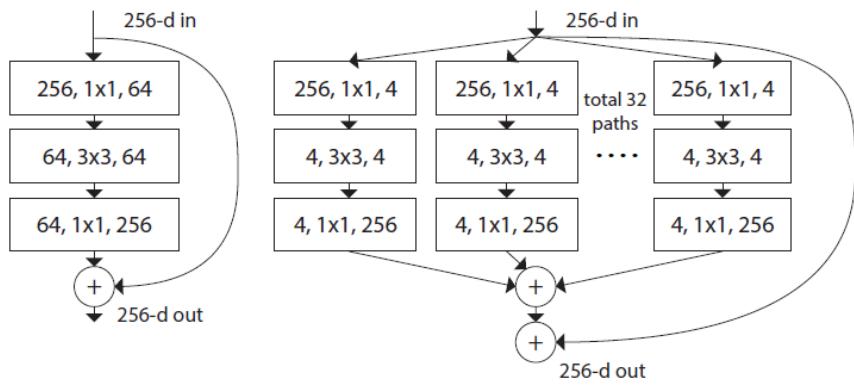


Рисунок 6 – Отличие Residual блоков архитектуры ResNet (слева) и ResNeXt (справа)

2.1.7 Выбор архитектур для дальнейшего исследования

Пользуясь проведённым сравнением в статье [2], приведём полученную пузырьковую диаграмму (рисунок 7).

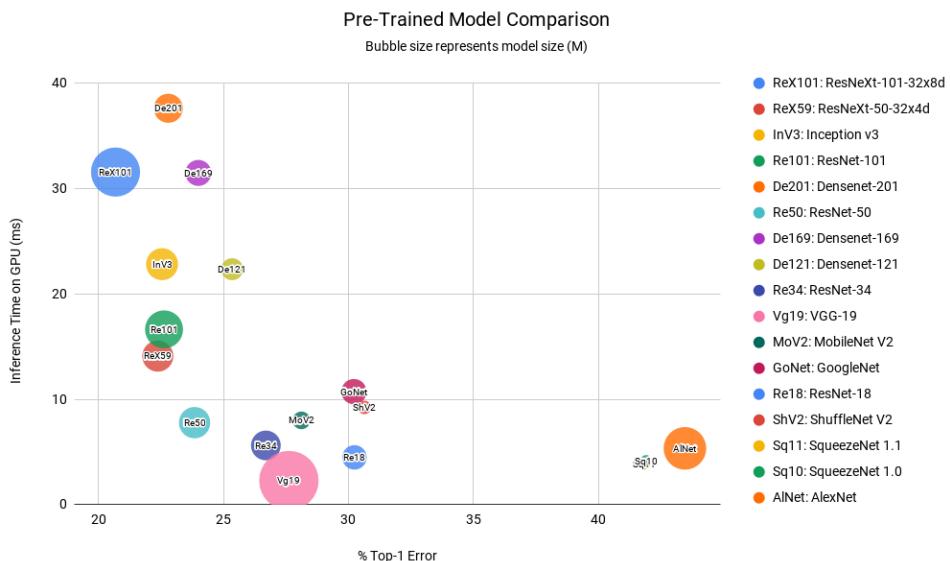


Рисунок 7 – Сравнительная диаграмма архитектур, где X-координата является ошибкой Top-1, Y-координата – это время вывода на GPU в миллисекундах, размер пузырька соответствует размеру модели

Для дальнейшего исследования выберем ResNet-50, как наиболее сбалансированную модель, ResNeXt-101-32x8d, как наиболее точную и ResNet-18, как достаточно быструю и лёгкую.

2.2 Задача локализации объектов на изображении

Задача локализации – определить наличие объекта на изображении и сформировать ограничивающую его рамку.

2.2.1 R-CNN

Архитектура сети R-CNN (Regions With CNNs) была разработана командой из UC Berkley для применения к задаче локализации [3].

Идея сводилась к предобучению свёрточной нейронной сети не на всём изображении целиком, а на предварительно выделенных другим способом регионах, на которых предположительно имеются какие-то объекты. Для выделения регионов использовали Selective Search.

В качестве свёрточной сети использовалась готовая архитектура — CaffeNet с заменой последнего классификационного слоя на слой с $N+1$ выходами (с дополнительным классом для фона).

Несмотря на то, что свёрточная сеть тренировалась на распознавание $N+1$ классов, в итоге она использовалась только для извлечения фиксированного 4096-размерного вектора признаков. Непосредственным определением объекта на изображении занимались N линейных SVM — каждый проводил бинарную классификацию по своему типу объектов, определяя есть ли такой в переданном регионе или нет.

Процедуру детектирования объектов сетью R-CNN можно разделить на следующие шаги:

1. Выделение регионов-кандидатов при помощи Selective Search.
2. Преобразование региона в размер, принимаемый CNN CaffeNet.
3. Получение при помощи CNN 4096-размерного вектора признаков.
4. Проведение N бинарных классификаций каждого вектора признаков при помощи N линейных SVM.
5. Линейная регрессия параметров рамки региона для более точного охвата объекта

Процедура иллюстрируется схемой на рисунке 8.

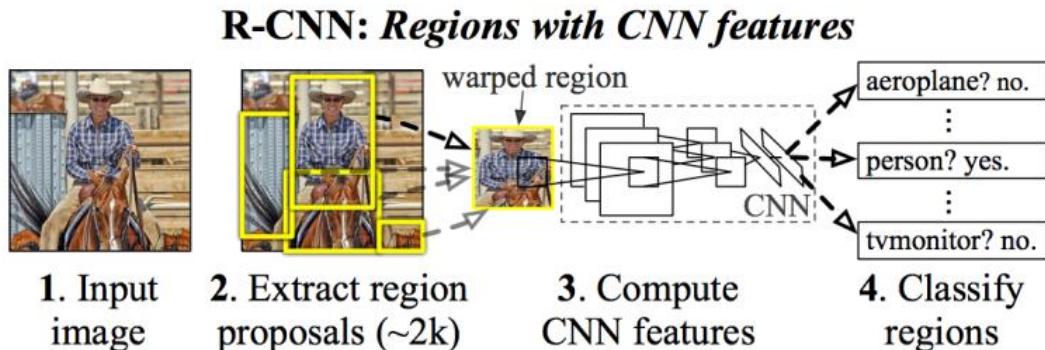


Рисунок 8 – Архитектура R-CNN

2.2.2 Fast R-CNN

Авторы Fast R-CNN предложили ускорить процесс за счёт пары модификаций [3]:

- Пропускать через CNN не 2000 регионов-кандидатов по отдельности, а всё изображение целиком. Предложенные регионы потом накладываются на полученную общую карту признаков;
- Вместо независимого обучения трёх моделей (CNN, SVM, bbox regressor) совместить все процедуры тренировки в одну.

Преобразование признаков, попавших в разные регионы, к фиксированному размеру производилось при помощи процедуры RoIPooling. Окно региона делилось на сетку, имеющую фиксированное число ячеек. По каждой ячейке проводился Max Pooling для выбора только одного значения.

Бинарные SVM не использовались, вместо этого выбранные признаки передавались на полносвязанный слой, а затем на два параллельных слоя: softmax с $N+1$ выходами (по одному на каждый класс + 1 для фона) и bounding box regressor.

Архитектура показана на рисунке 9.

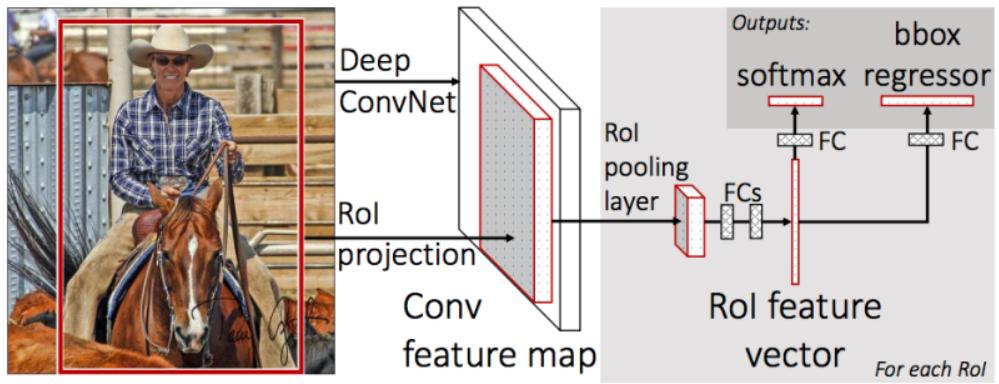


Рисунок 9 – Архитектура fast R-CNN

2.2.3 Faster R-CNN

После улучшений, сделанных в Fast R-CNN, самым узким местом нейросети оказался механизм генерации регионов-кандидатов. В 2015 команда из Microsoft Research смогла сделать этот этап значительно более быстрым [3]. Они предложили вычислять регионы не по изначальному изображению, а опять же по карте признаков, полученных из CNN. Для этого был добавлен модуль под названием Region Proposal Network (RPN). Новая архитектура показана на рисунке 10.

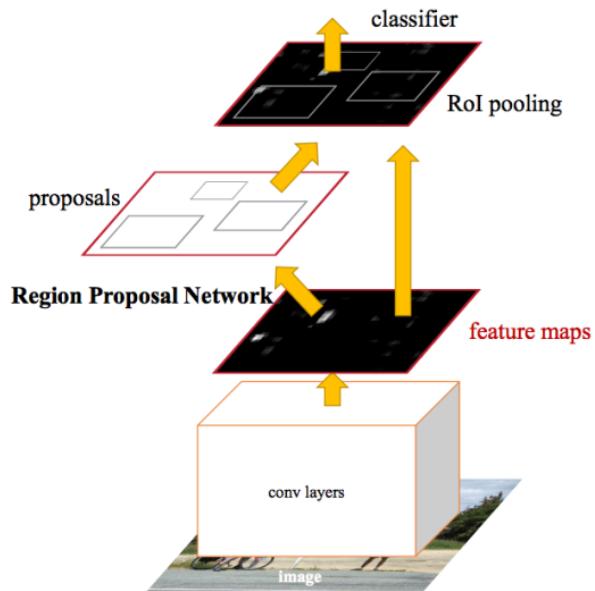


Рисунок 10 – Архитектура faster R-CNN

2.2.4 YOLO

You Only Look Once [4] (рисунок 11) — это популярная на текущий момент архитектура, которая используется для распознавания множественных объектов на изображении. Главная особенность этой архитектуры по сравнению с другими состоит в том, что большинство систем применяют CNN несколько раз к разным регионам изображения, в YOLO CNN применяется один раз ко всему изображению сразу. Сеть делит изображение на своеобразную сетку и предсказывает bounding boxes и вероятности того, что там есть искомый объект для каждого участка. Плюсы данного подхода состоят в том, что сеть смотрит на всё изображение сразу и учитывает контекст при локализации и распознавании объекта. Так же YOLO в 1000 раз быстрее чем R-CNN и около 100x быстрее чем Fast R-CNN.

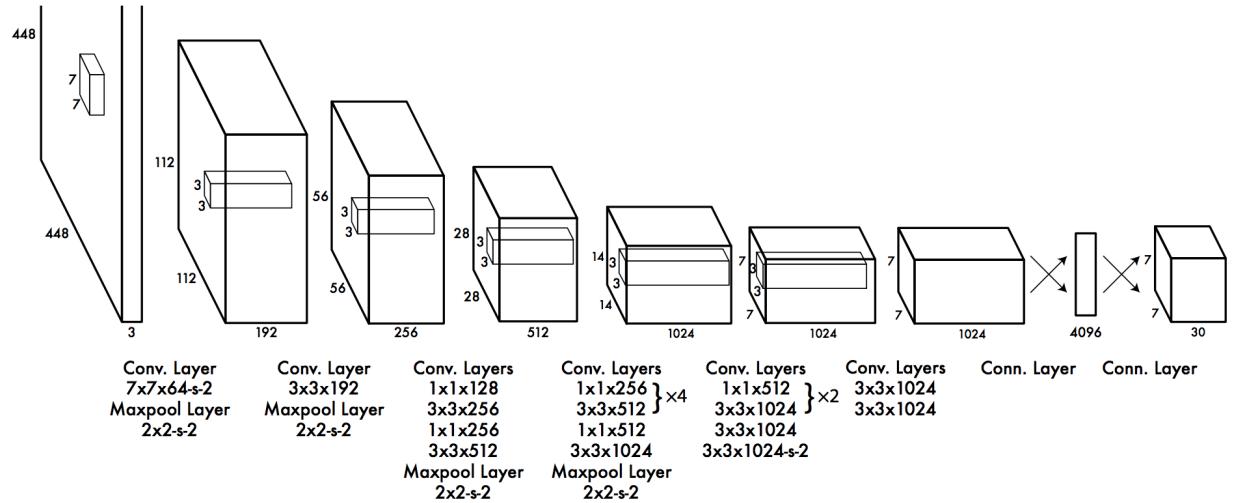


Рисунок 11 – Архитектура YOLO

2.2.5 SSD

Single Shot MultiBox Detector [5] — модель основанная на идее YOLO, то есть одноразового прохода свёртками по изображению, позволяет производить локализацию в реальном времени и по точности близка к Faster R-CNN. Особенностью этой архитектуры является отсутствие полносвязных слоёв и наличие детектирующих блоков на разных уровнях сети. Архитектура показана на рисунке 12.

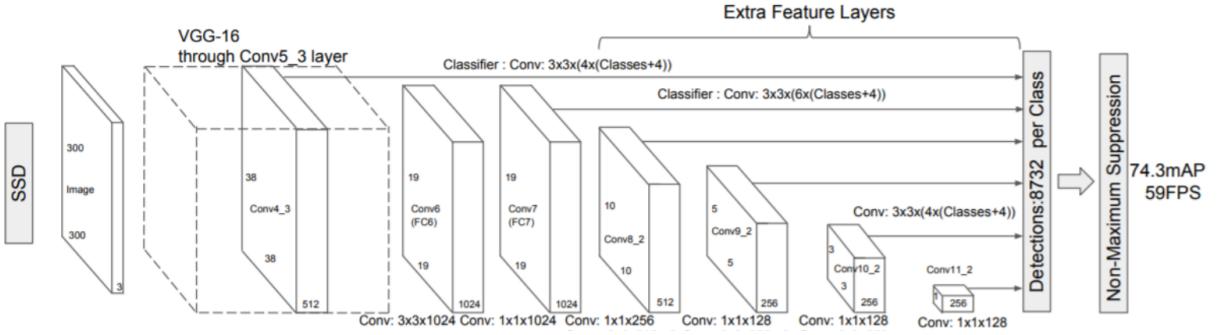


Рисунок 12 – Архитектура SSD

2.2.6 RetinaNet

Обнаружено, что в одноступенчатом детекторе существует проблема дисбаланса класса переднего плана, и считается, что это главная причина, которая делает производительность одноступенчатых детекторов ниже двухступенчатых. В RetinaNet, одноступенчатом детекторе с использованием комбинации Feature Pyramid Networks + ResNet, и благодаря специальной функции ошибки достигается более высокая точность по сравнению с Faster R-CNN [6]. Архитектура показана на рисунке 13.

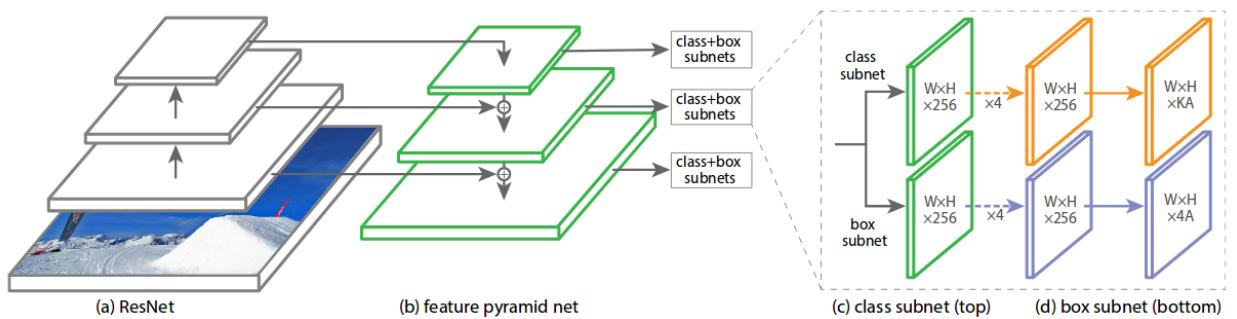


Рисунок 13 – Архитектура RetinaNet

2.2.7 Выбор архитектур для дальнейшего исследования

Для дальнейшего исследования выберем Faster R-CNN, как лучшего представителя двухступенчатого детектора, и одноступенчатую RetinaNet.

3 Исследование архитектур для классификации объектов на изображении

В данной главе представлено сравнение выбранных архитектур по числу слоёв, настраиваемых параметров, качеству работы на разных задачах.

Так как данные архитектуры уже обучены на задаче ImageNet1000 и умеют выделять признаки объектов, то для классификации на других данных они не обучаются заново, а используются предобученные модели. Все слои кроме полносвязного замораживаются для изменений, а полносвязный слой обучается с учётом необходимого количества классов.

Для исследования в данной работе используется python с библиотекой pytorch.

3.1 Сравнение архитектур

Для сравнения архитектур были загружены готовые модели, реализован подробный вывод структуры, подсчёт общего количества обучаемых параметров и количества параметров по слоям (Приложение А).

3.1.1 ResNet-18

На первом уровне ResNet-18 использует свёртку 7×7 с шагом 2, чтобы уменьшить входной сигнал до порядка, аналогично уровню пула. Затем следуют два residual блока перед повторной поникающей дискретизацией на 2. Уровень поникающей дискретизации также является слоем свёртки. Так продолжается ещё несколько раз. После применяется average pooling, который создаёт 512 признаков, усредняя данные каждой свёртки. Последний слой - полносвязный. В результате получается 1000-мерный вектор, который затем подаётся в слой Softmax.

Архитектура по слоям:

1. свёрточный слой $7 \times 7 \times 64$ + слой батч-нормализации
Max pool

2–5. свёрточные слои $\begin{bmatrix} 3 \times 3 \times 64 \\ 3 \times 3 \times 64 \end{bmatrix} \times 2 +$ слои батч-нормализации
 6–9. свёрточные слои $\begin{bmatrix} 3 \times 3 \times 128 \\ 3 \times 3 \times 128 \end{bmatrix} \times 2 +$ слои батч-нормализации
 10–13. свёрточные слои $\begin{bmatrix} 3 \times 3 \times 256 \\ 3 \times 3 \times 256 \end{bmatrix} \times 2 +$ слои батч-нормализации
 14–17. свёрточные слои $\begin{bmatrix} 3 \times 3 \times 512 \\ 3 \times 3 \times 512 \end{bmatrix} \times 2 +$ слои батч-нормализации
 Average pool
 18. полносвязный слой 512×1000

Итого 11 689 512 обучаемых параметров – 44.7 Мб.

3.1.2 ResNet-50

На первом уровне ResNet-50 использует свёртку 7×7 с шагом 2, чтобы уменьшить входной сигнал до порядка, аналогично уровню пула. Затем следуют три residual блока перед повторной поникающей дискретизацией на 2. Так продолжается ещё несколько раз. После применяется average pooling, который создаёт 2048 признаков, усредняя данные каждой свёртки. Последний слой - полносвязный. В результате получается 1000-мерный вектор, который затем подаётся в слой Softmax.

Архитектура по слоям:

1. свёрточный слой $7 \times 7 \times 64 +$ слой батч-нормализации

Max pool

2–10. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 64 \\ 3 \times 3 \times 64 \\ 1 \times 1 \times 256 \end{bmatrix} \times 3 +$ слои батч-нормализации
 11–22. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 128 \\ 3 \times 3 \times 128 \\ 1 \times 1 \times 512 \end{bmatrix} \times 4 +$ слои батч-нормализации

23–40. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 256 \\ 3 \times 3 \times 256 \\ 1 \times 1 \times 1024 \\ 1 \times 1 \times 512 \\ 3 \times 3 \times 512 \\ 1 \times 1 \times 2048 \end{bmatrix} \times 3$ + слои батч-нормализации
 41–49. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 256 \\ 3 \times 3 \times 256 \\ 1 \times 1 \times 1024 \\ 1 \times 1 \times 512 \\ 3 \times 3 \times 512 \\ 1 \times 1 \times 2048 \end{bmatrix} \times 3$ + слои батч-нормализации
 Average pool
 50. полносвязный слой 2048×1000

Итого 25 557 032 обучаемых параметров – 97.8 Мб.

3.1.3 ResNeXt-101-32x8d

На первом уровне ResNeXt-101 использует свёртку 7×7 с шагом 2, чтобы уменьшить входной сигнал до порядка, аналогично уровню пула. Затем следуют residual блоки, состоящие из 32 групп параллельных друг другу свёрток. После применяется average pooling, который создаёт 2048 признаков, усредняя данные каждой свёртки. Последний слой – полносвязный. В результате получается 1000-мерный вектор, который затем подаётся в слой Softmax.

Архитектура по слоям:

1. свёрточный слой $7 \times 7 \times 64$ + слой батч-нормализации

Max pool

2–10. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 256 \\ 3 \times 3 \times 256, g = 32 \\ 1 \times 1 \times 256 \end{bmatrix} \times 3$ + слои батч-нормализации
 11–22. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 512 \\ 3 \times 3 \times 512, g = 32 \\ 1 \times 1 \times 512 \\ 1 \times 1 \times 1024 \end{bmatrix} \times 4$ + слои батч-нормализации
 23–91. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 1024 \\ 3 \times 3 \times 1024, g = 32 \\ 1 \times 1 \times 1024 \end{bmatrix} \times 3$ + слои батч-нормализации

92–100. свёрточные слои $\begin{bmatrix} 1 \times 1 \times 2048 \\ 3 \times 3 \times 2048, g = 32 \\ 1 \times 1 \times 2048 \end{bmatrix} \times 3$ + слои батч-нормализации
 Average pool
 101. полносвязный слой 2048×1000

Итого 88 791 336 обучаемых параметров – 340 Мб.

3.2 Маленький объём размеченных данных и бинарная классификация

Взят датасет с чистыми и грязными тарелками, всего 40 размеченных изображений (по 20 каждого класса) и 744 изображения в тестовой выборке [7].

Для сравнения были реализованы:

- Класс-обёртка для данных, модификация класса ImageFolder, функция записи форматированного результата в csv файл (Приложение Б).
- Класс, обобщающий работу с моделями для классификации: создание модели, определение функции потерь и оптимизатора, обучение модели, валидация модели, сохранение и загрузка модели, построение графиков достоверности и ошибки, подсчёт метрик (достоверность, точность, полнота, F-мера), построение кривой ошибок и карт распределения предсказаний (Приложение В).

Все модели обучались с одинаковыми гиперпараметрами. Обучение длилось 100 эпох, а валидация проводилась через каждые 5 эпох (Приложение Г).

3.2.1 ResNet-18

Результаты обучения на разных этапах представлены на рисунке 14.

```

Epoch 10/100:
100%[██████████| 2/2 [00:10<00:00, 5.13s/it]
train loss: 0.3843 Acc: 0.8250
100%[██████████| 38/38 [02:46<00:00, 4.37s/it]
VAL loss: 0.4654 Acc: 0.7961
saving model with name: "models/resNet18forPlates-10-0.7961.pickle"

```

a)

```

Epoch 100/100:
100%[██████████| 2/2 [00:10<00:00, 5.25s/it]
train loss: 0.1839 Acc: 0.9000
100%[██████████| 38/38 [02:41<00:00, 4.25s/it]
VAL loss: 0.5939 Acc: 0.7750
66.81 minutes

```

b)

Рисунок 14 – Результаты обучения ResNet-18: a) лучшая эпоха, b)
последняя эпоха

Время обучения на 100 эпохах - 67 минут. История изменения значения функции потерь и достоверности отражена на рисунке 15.

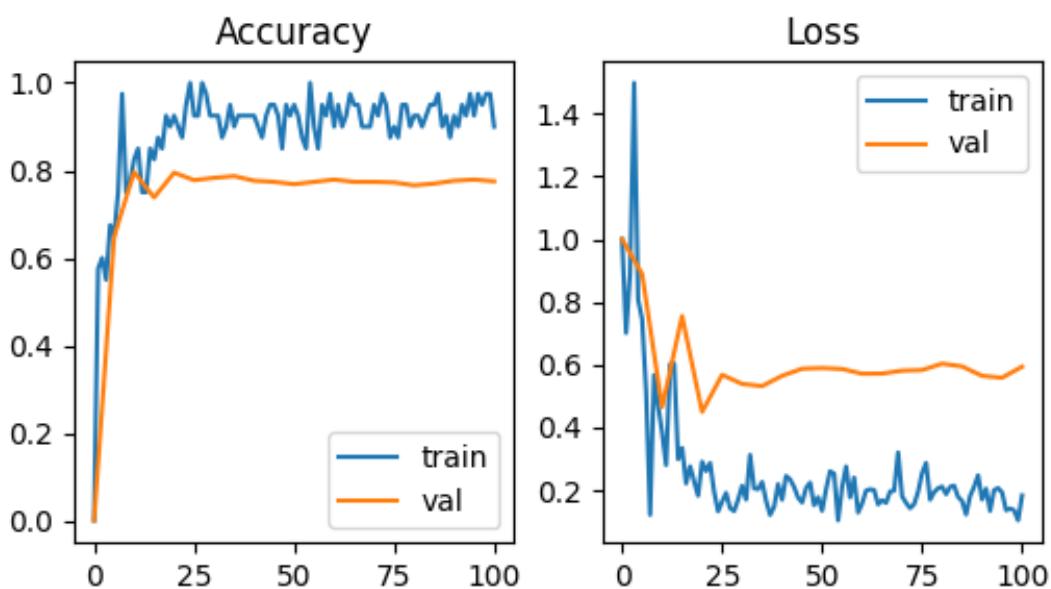


Рисунок 15 – Графики достоверности (слева) и функции потерь (справа)

Для выбора модели (последняя или лучшая по достоверности из истории обучения) построены их кривые ошибок (Рисунок 16). Кривая ошибки показывает зависимость ошибки от выбранного порогового значения, а площадь под графиком отражает качество модели независимо от выбора порога. Выбрана модель, полученная на 10 эпохе обучения.

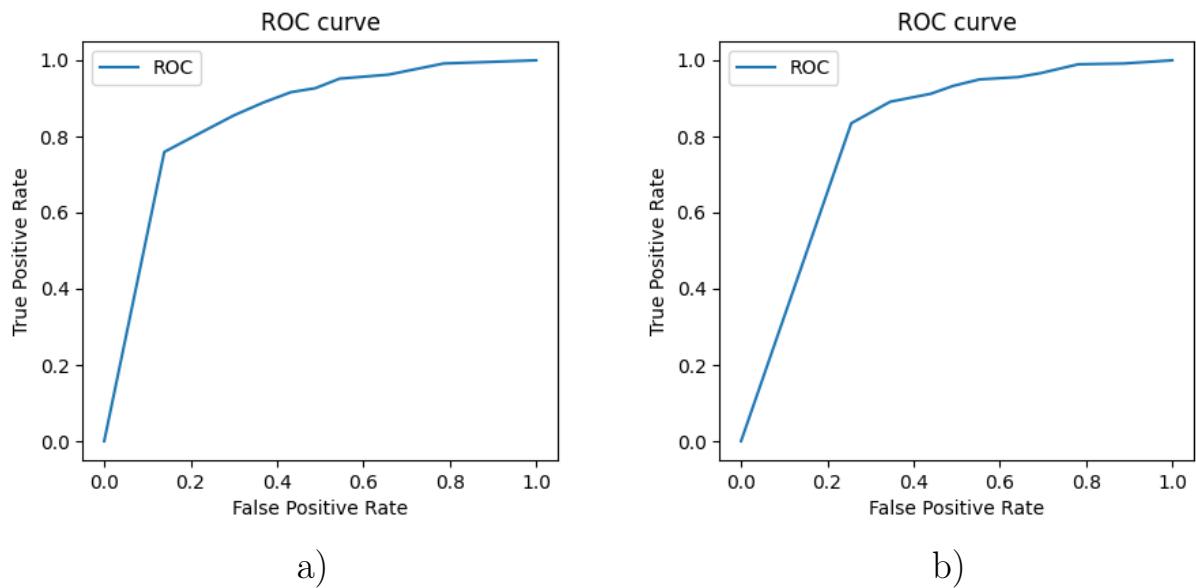


Рисунок 16 – Кривые ошибок: а) лучшая эпоха, б) последняя эпоха

Для выбора более точного порога рассчитаны метрики для разных пороговых значений (Рисунок 17) и построены карты распределений (Рисунок 18).

```
In[6]: resNet18forPlates.count_metrics_for_class(1)
Порог: 0.1, TP: 474, FP: 209, FN: 4, TN: 57, acc:0.7137, precision: 0.6940, recall: 0.9916, F: 0.8165
Порог: 0.2, TP: 460, FP: 175, FN: 18, TN: 91, acc:0.7406, precision: 0.7244, recall: 0.9623, F: 0.8266
Порог: 0.3, TP: 455, FP: 145, FN: 23, TN: 121, acc:0.7742, precision: 0.7583, recall: 0.9519, F: 0.8442
Порог: 0.4, TP: 443, FP: 130, FN: 35, TN: 136, acc:0.7782, precision: 0.7731, recall: 0.9268, F: 0.8430
Порог: 0.5, TP: 438, FP: 115, FN: 40, TN: 151, acc:0.7917, precision: 0.7920, recall: 0.9163, F: 0.8497
Порог: 0.6, TP: 425, FP: 98, FN: 53, TN: 168, acc:0.7970, precision: 0.8126, recall: 0.8891, F: 0.8492
Порог: 0.7, TP: 409, FP: 80, FN: 69, TN: 186, acc:0.7997, precision: 0.8364, recall: 0.8556, F: 0.8459
Порог: 0.8, TP: 392, FP: 64, FN: 86, TN: 202, acc:0.7984, precision: 0.8596, recall: 0.8201, F: 0.8394
Порог: 0.9, TP: 363, FP: 37, FN: 115, TN: 229, acc:0.7957, precision: 0.9075, recall: 0.7594, F: 0.8269
```

Рисунок 17 – Метрики (достоверность, точность, полнота, F-мера) для разных пороговых значений

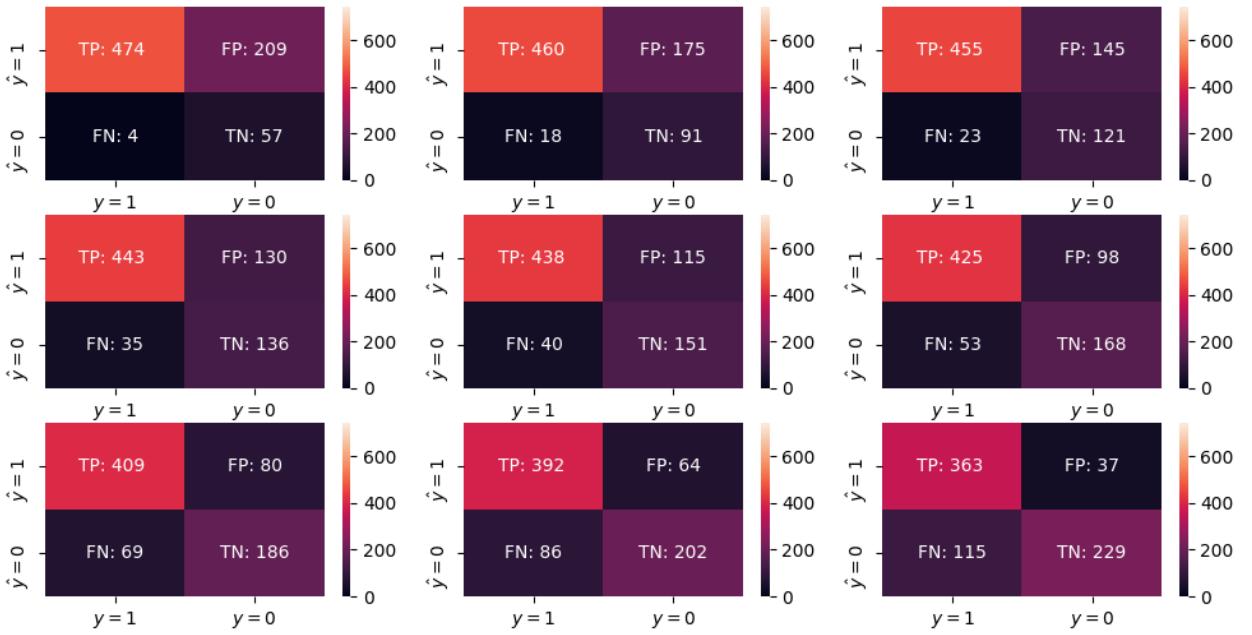
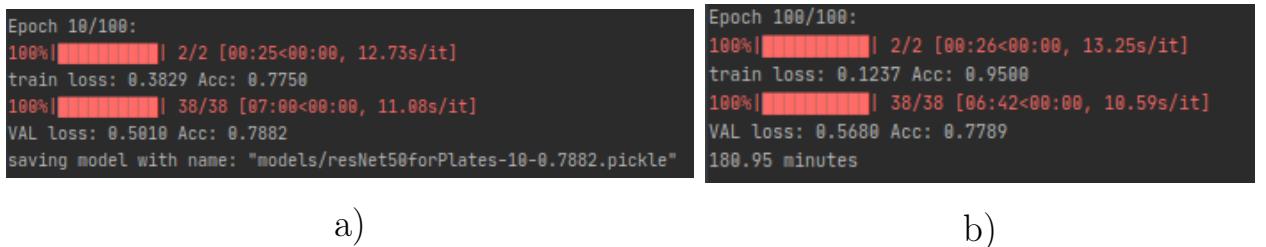


Рисунок 18 – Карты распределения предсказаний при значении порога от 0,1 до 0,9

Таким образом порог, обеспечивающий наилучшее качество равен 0.6. Среднее время для предсказания класса одного изображения занимает 0.239 секунды.

3.2.2 ResNet-50

Результаты обучения на разных этапах представлены на рисунке 19.



a)

b)

Рисунок 19 – Результаты обучения ResNet-50: а) лучшая эпоха, б) последняя эпоха

Время обучения на 100 эпохах - 181 минута. История изменения значения функции потерь и достоверности отражена на рисунке 20.

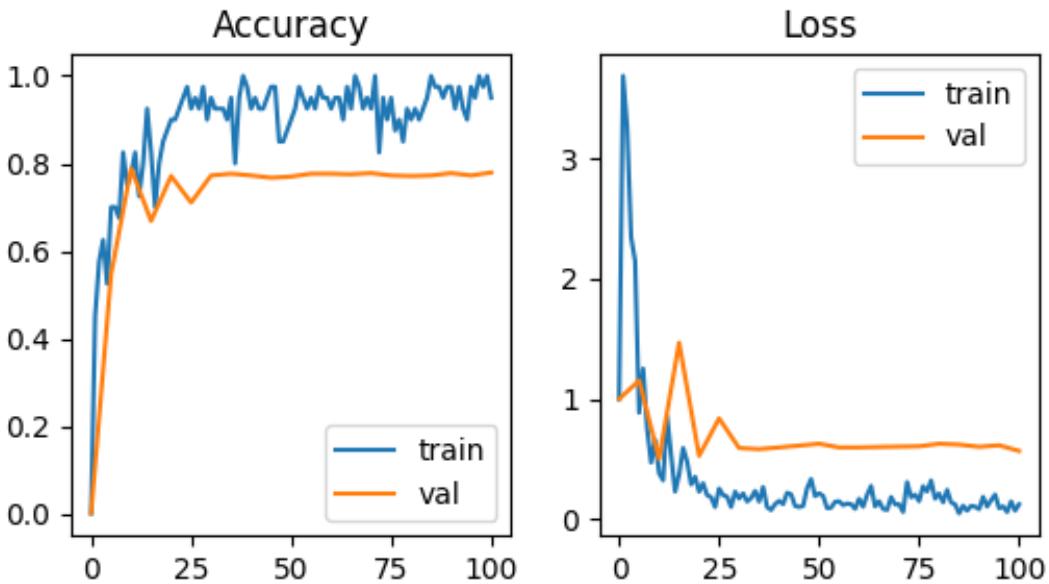


Рисунок 20 – Графики достоверности (слева) и функции потерь (справа)

Для выбора модели (последняя или лучшая по достоверности из истории обучения) построены их кривые ошибок (Рисунок 21). Кривая ошибки показывает зависимость ошибки от выбранного порогового значения, а площадь под графиком отражает качество модели независимо от выбора порога. Выбрана модель, полученная на последней эпохе обучения.

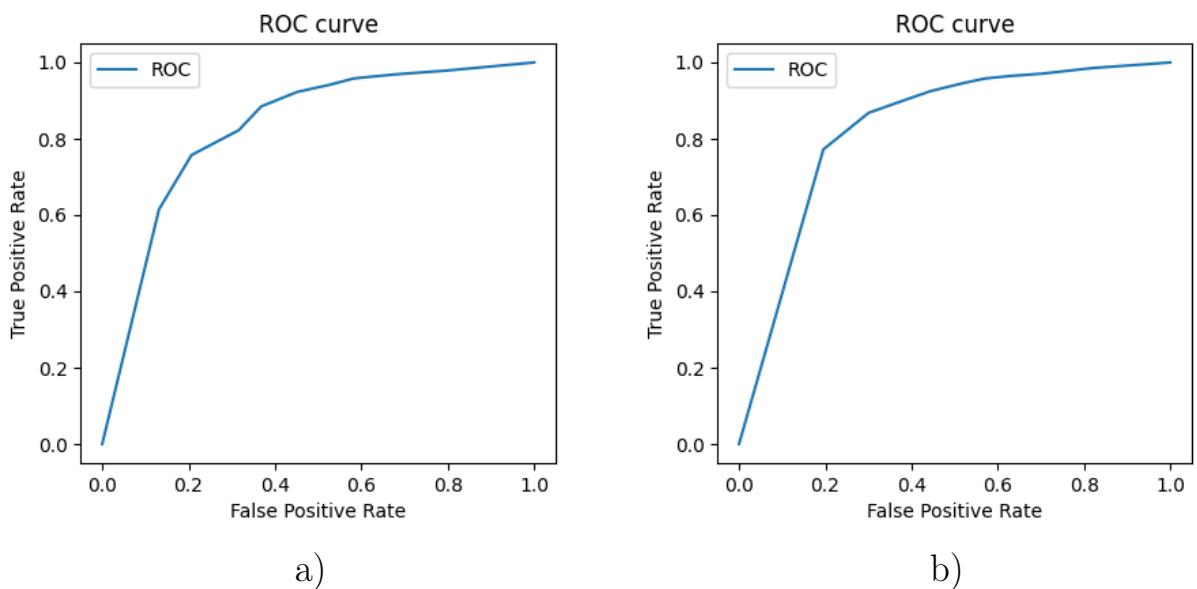


Рисунок 21 – Кривые ошибок: а) лучшая эпоха, б) последняя эпоха

Для выбора более точного порога рассчитаны метрики для разных пороговых значений (Рисунок 22) и построены карты распределений (Ри-

сунок 23).

```
In[6]: resNet50forPlates.count_metrics_for_class(1)
Порог: 0.1, TP: 471, FP: 218, FN: 7, TN: 48, acc:0.6976, precision: 0.6836, recall: 0.9854, F: 0.8072
Порог: 0.2, TP: 464, FP: 187, FN: 14, TN: 79, acc:0.7298, precision: 0.7127, recall: 0.9707, F: 0.8220
Порог: 0.3, TP: 461, FP: 167, FN: 17, TN: 99, acc:0.7527, precision: 0.7341, recall: 0.9644, F: 0.8336
Порог: 0.4, TP: 458, FP: 152, FN: 20, TN: 114, acc:0.7688, precision: 0.7508, recall: 0.9582, F: 0.8419
Порог: 0.5, TP: 452, FP: 138, FN: 26, TN: 128, acc:0.7796, precision: 0.7661, recall: 0.9456, F: 0.8464
Порог: 0.6, TP: 442, FP: 118, FN: 36, TN: 148, acc:0.7930, precision: 0.7893, recall: 0.9247, F: 0.8516
Порог: 0.7, TP: 430, FP: 101, FN: 48, TN: 165, acc:0.7997, precision: 0.8098, recall: 0.8996, F: 0.8523
Порог: 0.8, TP: 415, FP: 80, FN: 63, TN: 186, acc:0.8078, precision: 0.8384, recall: 0.8682, F: 0.8530
Порог: 0.9, TP: 369, FP: 52, FN: 109, TN: 214, acc:0.7836, precision: 0.8765, recall: 0.7720, F: 0.8209
```

Рисунок 22 – Метрики (достоверность, точность, полнота, F-мера) для разных пороговых значений

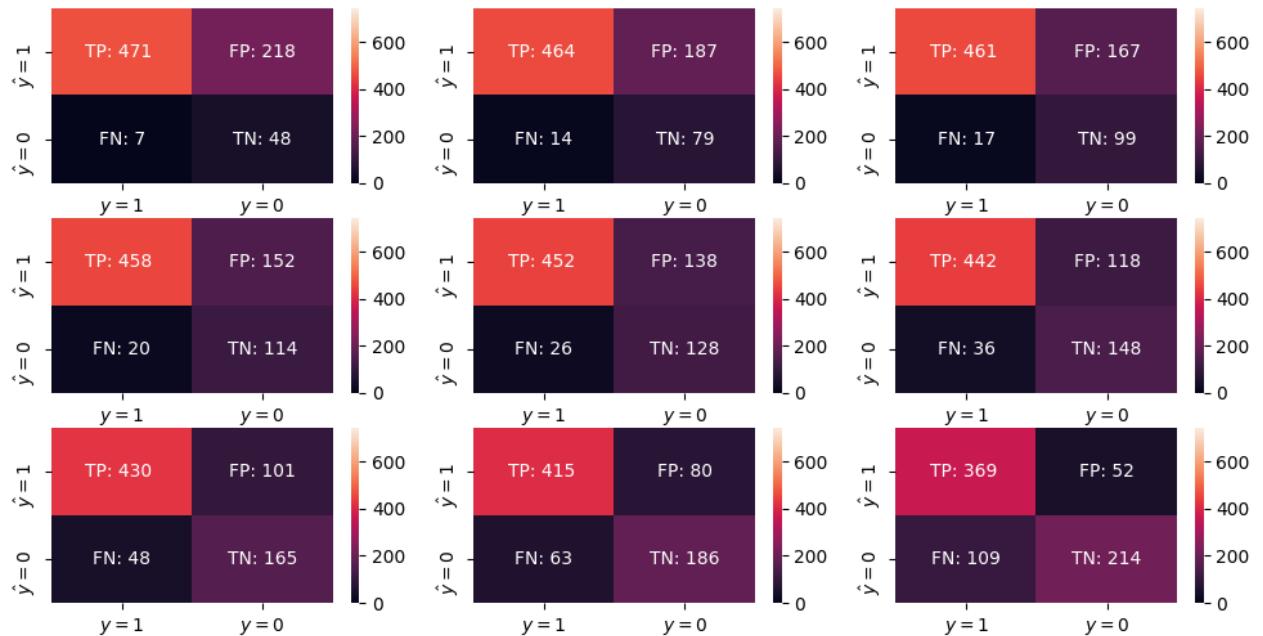


Рисунок 23 – Карты распределения предсказаний при значении порога от 0,1 до 0,9

Таким образом порог, обеспечивающий наилучшее качество равен 0.8. Среднее время для предсказания класса одного изображения занимает 0.555 секунды.

3.2.3 ResNeXt-101-32x8d

Результаты обучения на разных этапах представлены на рисунке 24.

```

Epoch 20/100:
100%[██████████] 2/2 [01:14<00:00, 37.30s/it]
train loss: 0.4971 Acc: 0.7250
100%[██████████] 38/38 [20:51<00:00, 32.94s/it]
VAL loss: 0.3763 Acc: 0.8329
saving model with name: "models/resNeXt101forPlates-20-0.8329.pickle"

Epoch 100/100:
100%[██████████] 2/2 [01:15<00:00, 37.69s/it]
train loss: 0.0687 Acc: 0.9750
100%[██████████] 38/38 [21:32<00:00, 34.01s/it]
VAL loss: 0.5085 Acc: 0.7974
542.84 minutes

```

a)

b)

Рисунок 24 – Результаты обучения ResNeXt-101-32x8d: а) лучшая эпоха, б) последняя эпоха

Время обучения на 100 эпохах - 543 минуты. История изменения значения функции потерь и достоверности отражена на рисунке 25.

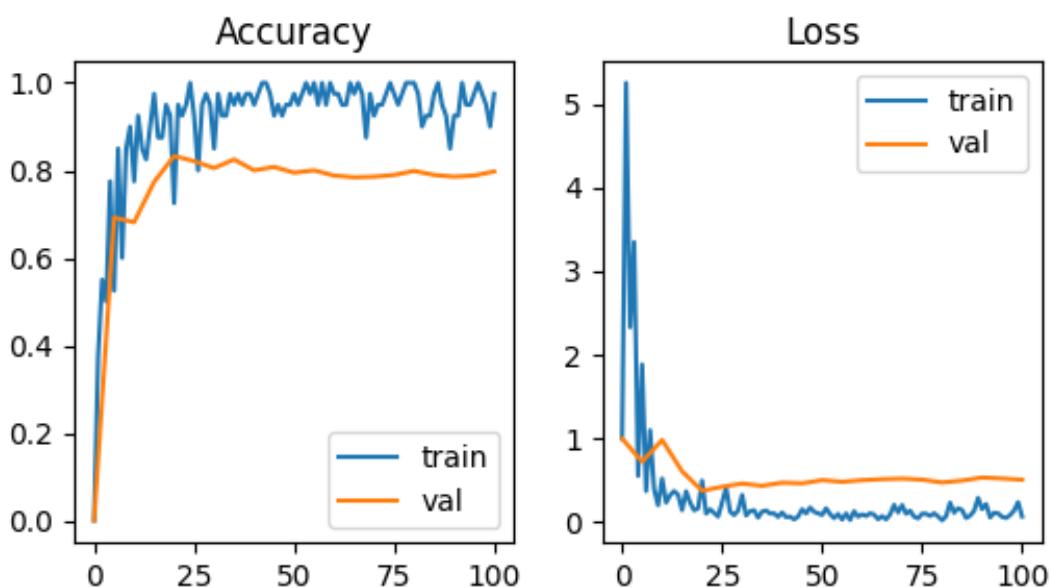


Рисунок 25 – Графики достоверности (слева) и функции потерь (справа)

Для выбора модели (последняя или лучшая по достоверности из истории обучения) построены их кривые ошибок (Рисунок 26). Кривая ошибки показывает зависимость ошибки от выбранного порогового значения, а площадь под графиком отражает качество модели независимо от выбора порога. Выбрана модель, полученная на 20 эпохе обучения.

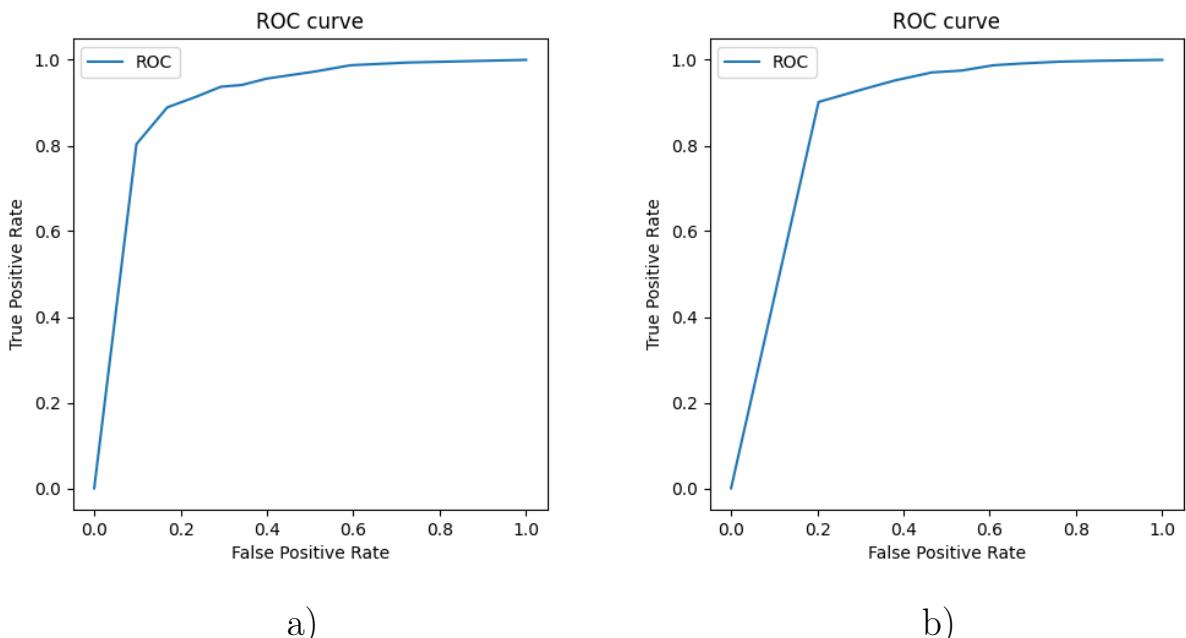


Рисунок 26 – Кривые ошибок: а) лучшая эпоха, б) последняя эпоха

Для выбора более точного порога рассчитаны метрики для разных пороговых значений (Рисунокс) и построены карты распределений (Рисунок 28).

```
In[4]: resNeXt101forPlates.count_metrics_for_class(1)
Порог: 0.1, TP: 475, FP: 193, FN: 3, TN: 73, acc:0.7366, precision: 0.7111, recall: 0.9937, F: 0.8290
Порог: 0.2, TP: 472, FP: 158, FN: 6, TN: 108, acc:0.7796, precision: 0.7492, recall: 0.9874, F: 0.8520
Порог: 0.3, TP: 465, FP: 136, FN: 13, TN: 130, acc:0.7997, precision: 0.7737, recall: 0.9728, F: 0.8619
Порог: 0.4, TP: 457, FP: 106, FN: 21, TN: 160, acc:0.8293, precision: 0.8117, recall: 0.9561, F: 0.8780
Порог: 0.5, TP: 450, FP: 91, FN: 28, TN: 175, acc:0.8401, precision: 0.8318, recall: 0.9414, F: 0.8832
Порог: 0.6, TP: 448, FP: 78, FN: 30, TN: 188, acc:0.8548, precision: 0.8517, recall: 0.9372, F: 0.8924
Порог: 0.7, TP: 437, FP: 63, FN: 41, TN: 203, acc:0.8602, precision: 0.8740, recall: 0.9142, F: 0.8937
Порог: 0.8, TP: 425, FP: 45, FN: 53, TN: 221, acc:0.8683, precision: 0.9043, recall: 0.8891, F: 0.8966
Порог: 0.9, TP: 384, FP: 26, FN: 94, TN: 240, acc:0.8387, precision: 0.9366, recall: 0.8033, F: 0.8649
```

Рисунок 27 – Метрики (достоверность, точность, полнота, F-мера) для разных пороговых значений

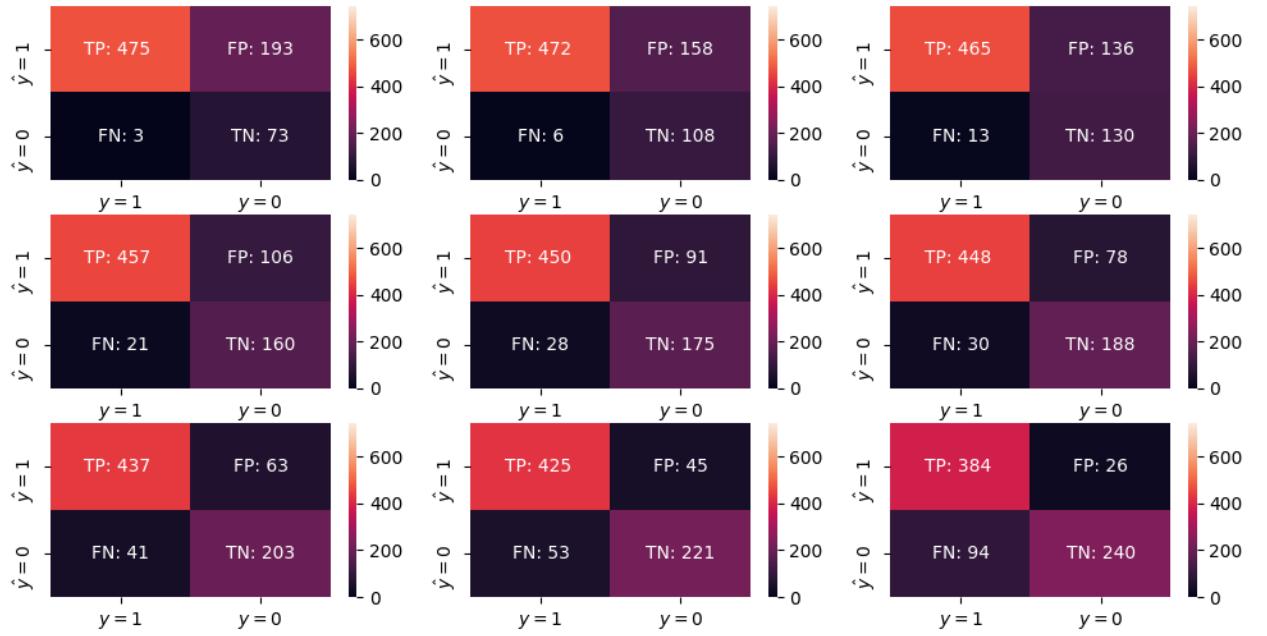


Рисунок 28 – Карты распределения предсказаний при значении порога от 0,1 до 0,9

Таким образом порог, обеспечивающий наилучшее качество равен 0.8. Среднее время для предсказания класса одного изображения занимает 1.752 секунды.

3.3 Многоклассовая классификация

Взят набор данных Stanford Dogs, который содержит 20 580 изображений 120 пород собак. Так как этот набор данных был построен с использованием изображений и аннотаций из ImageNet для задачи детальной категоризации изображений [8], то вместо дообучения производится сравнение качества готовых моделей с учётом расположения этих 120 классов среди 1000 классов ImageNet [9].

Для сравнения были реализованы:

- модификация класса ImageFolder с учётом расположения классов по род собак в наборе данных ImageNet(Приложение Б).
- Функция для оценки качества моделей: подсчёт метрик (достоверность, точность, полнота, F-мера), построение карт распределения предсказаний (Приложение Д).

3.3.1 ResNet-18

Достоверность составляет 0.8090, среднее значение F-меры – 0.8118. Карты распределения предсказаний первых 30 классов представлены на рисунке 29.

3.3.2 ResNet-50

Достоверность составляет 0.8784, среднее значение F-меры – 0.8790. Карты распределения предсказаний первых 30 классов представлены на рисунке 30.

3.3.3 ResNeXt-101-32x8d

Достоверность составляет 0.9384, среднее значение F-меры – 0.9392. Карты распределения предсказаний первых 30 классов представлены на рисунке 31.



Рисунок 29 – Карты распределения предсказаний ResNet-18 для классов с 1 по 30

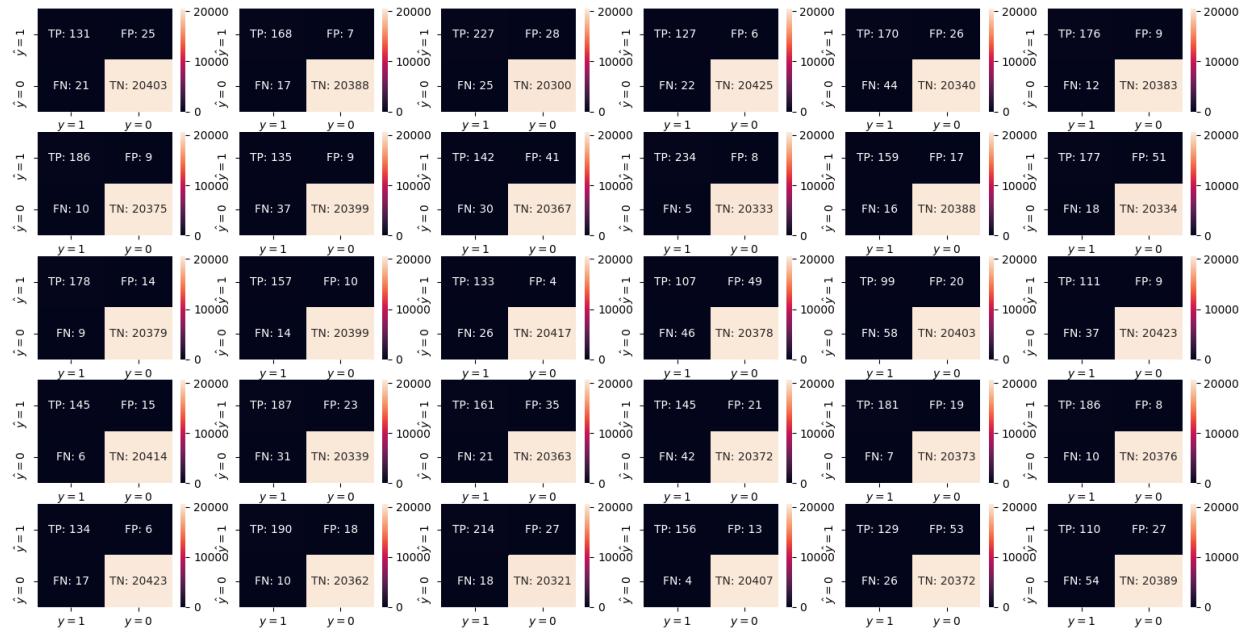


Рисунок 30 – Карты распределения предсказаний ResNet-50 для классов с 1 по 30

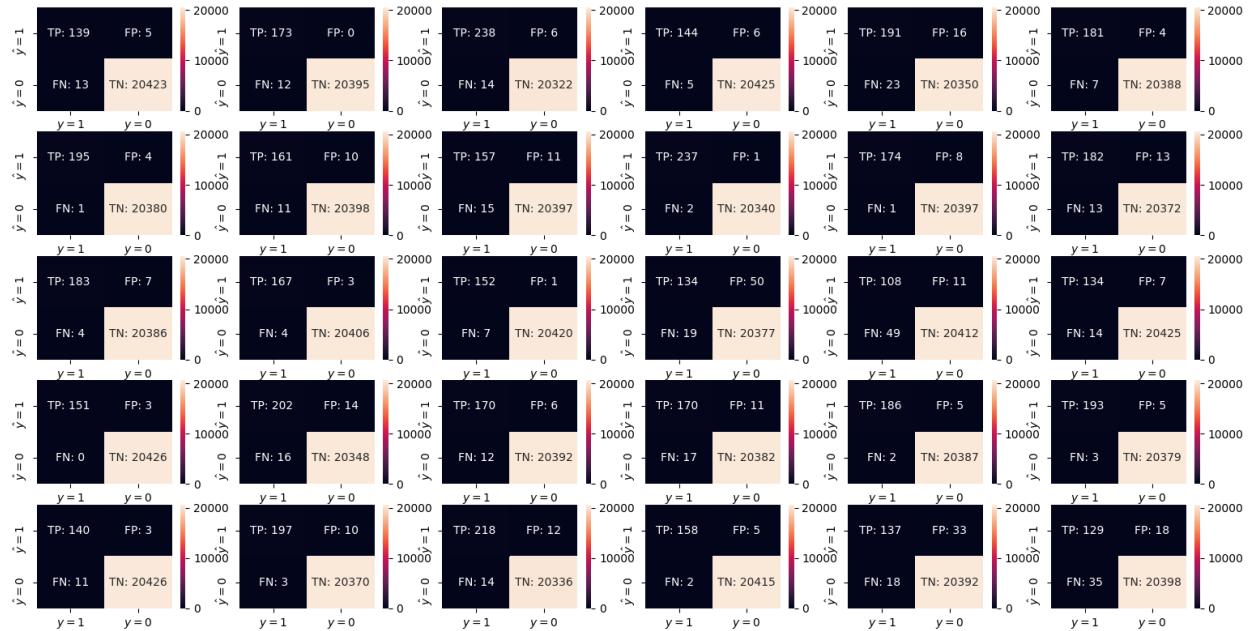


Рисунок 31 – Карты распределения предсказаний ResNeXt-101-32x8d для классов с 1 по 30

3.4 Результаты сравнения

Результаты сравнения характеристик сведены в таблицу 1.

Таблица 1 – результаты сравнения архитектур для классификации

		ResNet-18	ResNet-50	ResNeXt-101
Количество слоёв		18	50	101
Количество параметров		11 689 512	22 557 032	88 791 336
Объём памяти		44.7 Мб	97.8 Мб	340 Мб
Время предсказания		0.24 с	0.56 с	1.75 с
Бинарная классификация	Достоверность	0.797	0.808	0.868
	Точность	0.813	0.838	0.904
	Полнота	0.889	0.868	0.889
	F–мера	0.849	0.853	0.897
Многоклассовая классификация	Достоверность	0.809	0.878	0.938
	F–мера	0.812	0.879	0.939

По результатам сравнения архитектура ResNeXt-101 показала лучшее качество на всех задачах, однако время предсказания на одном изображении у ResNeXt-101 примерно в 7 раз больше, чем у ResNet-18. Таким образом эта архитектура рекомендуется для задач, где необходима высокая точность, но нет требований к скорости, например, анализ медицинских снимков. Помимо высокой скорости, ResNet-18 занимает небольшой объём в памяти, что делает её пригодной для использования на мобильных устройствах, в одноплатных компьютерах и в режиме реального времени. ResNet-50 представляет собой компромисс между точностью и скоростью, подходит для широкого спектра задач.

4 Исследование архитектур для локализации объектов на изображении

В данной главе представлен обзор архитектур, сравнение по качеству работы на наборе данных COCO 2017 и нескольких произвольно выбранных изображениях.

4.1 Сравнение архитектур

Сравниваются две архитектуры Faster R-CNN и RetinaNet, основанные на уже рассмотренной архитектуре ResNeXt-101-32x8d. Изучению подлежит общая структура и характерные особенности сравниваемых решений.

4.1.1 Faster R-CNN

Схема архитектуры представлена на рисунке 32.

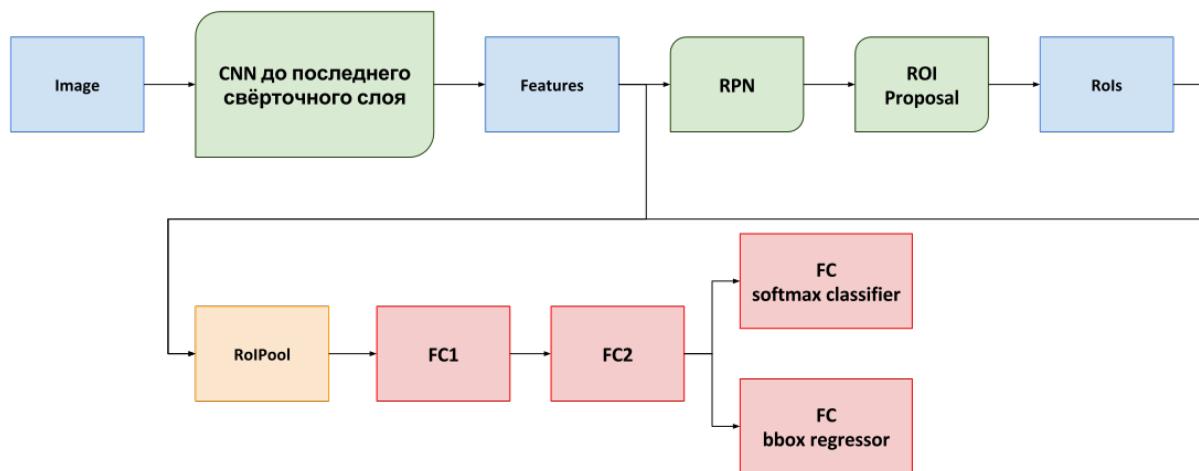


Рисунок 32 – Архитектура faster R-CNN

В качестве свёрточной сети для извлечения признаков используется ResNeXt-101-32x8d. Далее для каждой точки карты особенностей проверяется k претендентов разных размеров [10]. Для этого используется Region Proposal Network (Рисунок 33).

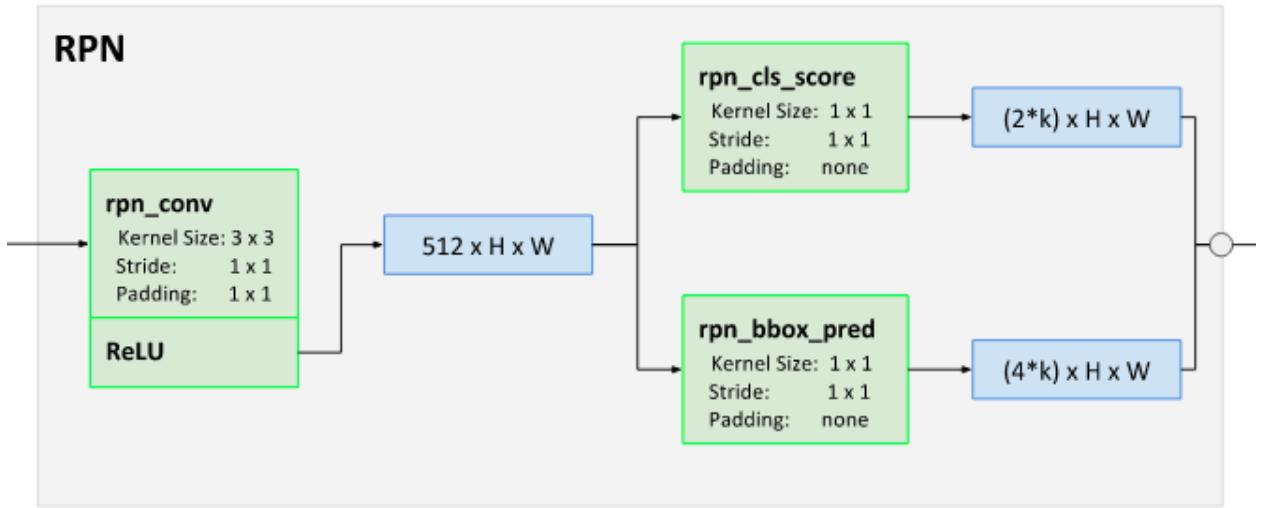


Рисунок 33 – Схема Region Proposal Network

Как видно из схемы, карта особенностей, полученная от свёрточной сети, подаётся на свёрточный слой с ядром размера 3×3 . А выход этого свёрточного слоя параллельно подаётся на два свёрточных слоя с ядром размера 1×1 . Первый слой *rpn_cls_score* выдаёт k пар - вероятности наличия или отсутствия объекта в соответствующем регионе. Слой *rpn_bbox_pred* выдаёт k четырёхок - поправки для координат центра и размеров соответствующего региона претендента.

Далее RoI pooling слой принимает на вход карту особенностей, полученную от последнего свёрточного слоя нейронной сети, и RoI претендента (в координатах изображения). RoI преобразуется из координат изображения в координаты на карте особенностей и на полученный прямоугольник накладывается сетка с наперёд заданными размерами. Делается max pooling по каждой ячейке этой сетки. Таким образом RoI pooling слой преобразует вектор особенностей произвольного прямоугольника из исходного изображения в вектор особенностей фиксированной размерности.

После RoI pooling слоя данные через два полно связных слоя подаются параллельно на softmax слой для оценки принадлежности данного претендента одному из классов объектов и слой реализующий регрессию, которая уточняет BBox объекта.

4.1.2 RetinaNet

RetinaNet (Рисунок 34) представляет собой единую унифицированную сеть, состоящую из магистральной сети и двух специализированных подсетей [12]. Магистраль отвечает за вычисление карты признаков по всему входному изображению и является собственной свёрточной сетью. Первая подсеть выполняет классификацию на выходе магистрали; вторая подсеть выполняет регрессию ограничивающего прямоугольника свёртки.

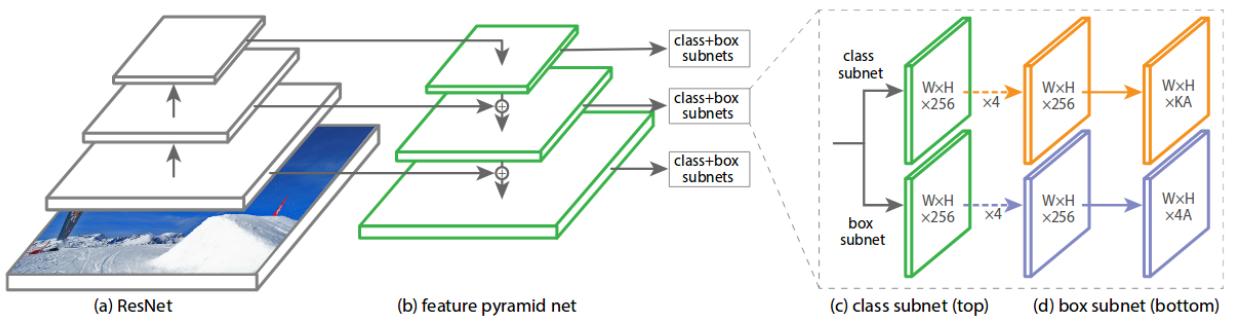


Рисунок 34 – Архитектура RetinaNet

Магистраль – Feature Pyramid Network, построенная поверх ResNet. Классификационная подсеть предсказывает вероятность присутствия объекта в каждой пространственной позиции. Фокусная потеря применяется как функция потери. Подсеть блока регрессии выводит местоположение объекта относительно якорной рамки, если объект существует.

Feature Pyramid Network (Рисунок 35) – это структура для обнаружения объектов разных масштабов. Она заменяет средство извлечения признаков таких детекторов, как Faster R-CNN, и генерирует несколько слоёв карт признаков (многомасштабные карты признаков) с более качественной информацией, чем обычная пирамида признаков для обнаружения объектов [11].

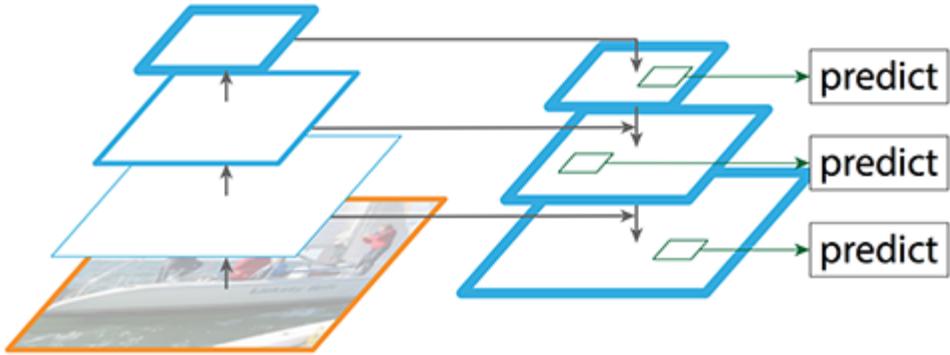


Рисунок 35 – Схема Feature Pyramid Network

FPN состоит из восходящего и нисходящего пути. Восходящий путь - это обычная свёрточная сеть для извлечения признаков. По мере подъёма пространственное разрешение уменьшается. При обнаружении большего количества высокоуровневых структур семантическое значение для каждого уровня увеличивается. FPN обеспечивает нисходящий путь для создания слоёв с более высоким разрешением из семантического обогащённого слоя. Хотя восстановленные слои семантически сильны, но расположение объектов не является точным после всех понижений и повышений. Поэтому добавлены боковые связи между реконструированными слоями и соответствующими картами признаков, чтобы помочь детектору прогнозировать местоположение лучше.

Focal Loss предназначен для решения одноэтапных проблем обнаружения объектов с дисбалансом, когда существует очень большое количество возможных классов фона и всего несколько классов переднего плана. Это приводит к тому, что обучение становится неэффективным, поскольку большинство местоположений представляют собой простые негативы, которые не дают полезного сигнала, а огромное количество этих негативных примеров подавляет обучение и снижает производительность модели. Фокусная потеря основана на перекрёстной энтропийной потере.

4.2 Сравнение на наборе данных COCO 2017

Определимся с метриками для сравнения. Оценка достоверности - это вероятность того, что якорный ящик содержит объект. Обычно это

предсказывается классификатором. Пересечение через объединение (IoU) определяется как площадь пересечения, делённая на площадь объединения предсказанного ограничивающего прямоугольника и истинного ограничивающего прямоугольника [13].

Обнаружение считается истинно положительным (TP), только если оно удовлетворяет трём условиям:

1. показатель достоверности больше порогового значения;
2. предсказанный класс соответствует истинному классу;
3. прогнозируемый ограничивающий прямоугольник имеет IoU, превышающий пороговое значение.

Нарушение любого из двух последних условий приводит к ложному положительному результату (FP). Когда показатель достоверности обнаружения, который должен обнаруживать объект, ниже порогового значения, обнаружение считается ложным отрицанием (FN). Когда показатель достоверности обнаружения, который не должен обнаруживать что-либо, ниже порогового значения, обнаружение считается истинно отрицательным (TN).

С учётом этих правил достоверность, точность и полнота считаются обычным образом.

Средняя точность (AP) основана на кривой точности-полноты. По сути, AP - это точность, усреднённая по всем уникальным уровням полноты. Расчет AP включает только один класс, однако при обнаружении объектов обычно $K > 1$ классов. Усреднённая средняя точность (mAP) определяется как среднее значение AP для всех K классов.

Задача COCO определяет несколько показателей mAP с использованием различных порогов, в том числе:

- mAP, который представляет собой mAP, усреднённый по 10 пороговым значениям IoU (т.е. 0.50, 0.55, 0.60, . . . , 0.95) и является первичной метрикой ;
- mAP50 ($\text{IoU} = 0.5$);
- mAP75 ($\text{IoU} = 0.75$);

- mAP_s, который является mAP для небольших объектов, которые охватывают площадь менее 32^2 ;
- mAP_m, который является mAP для средних объектов, которые покрывают область больше, чем 32^2 , но меньше чем 96^2 ;
- mAP_l, который является mAP для больших объектов, которые охватывают площадь больше 96^2 .

Для изучения моделей использовалась библиотека mm detection [14].

4.2.1 Faster R-CNN

Результаты валидации после 12 эпох обучения:

- $mAP = 0.412$;
- $mAP_{50} = 0.621$;
- $mAP_{75} = 0.451$;
- $mAP_s = 0.240$;
- $mAP_m = 0.455$;
- $mAP_l = 0.535$.

4.2.2 RetinaNet

Результаты валидации после 12 эпох обучения:

- $mAP = 0.399$;
- $mAP_{50} = 0.596$;
- $mAP_{75} = 0.427$;
- $mAP_s = 0.223$;
- $mAP_m = 0.442$;
- $mAP_l = 0.525$.

4.3 Сравнение на произвольно выбранных изображениях

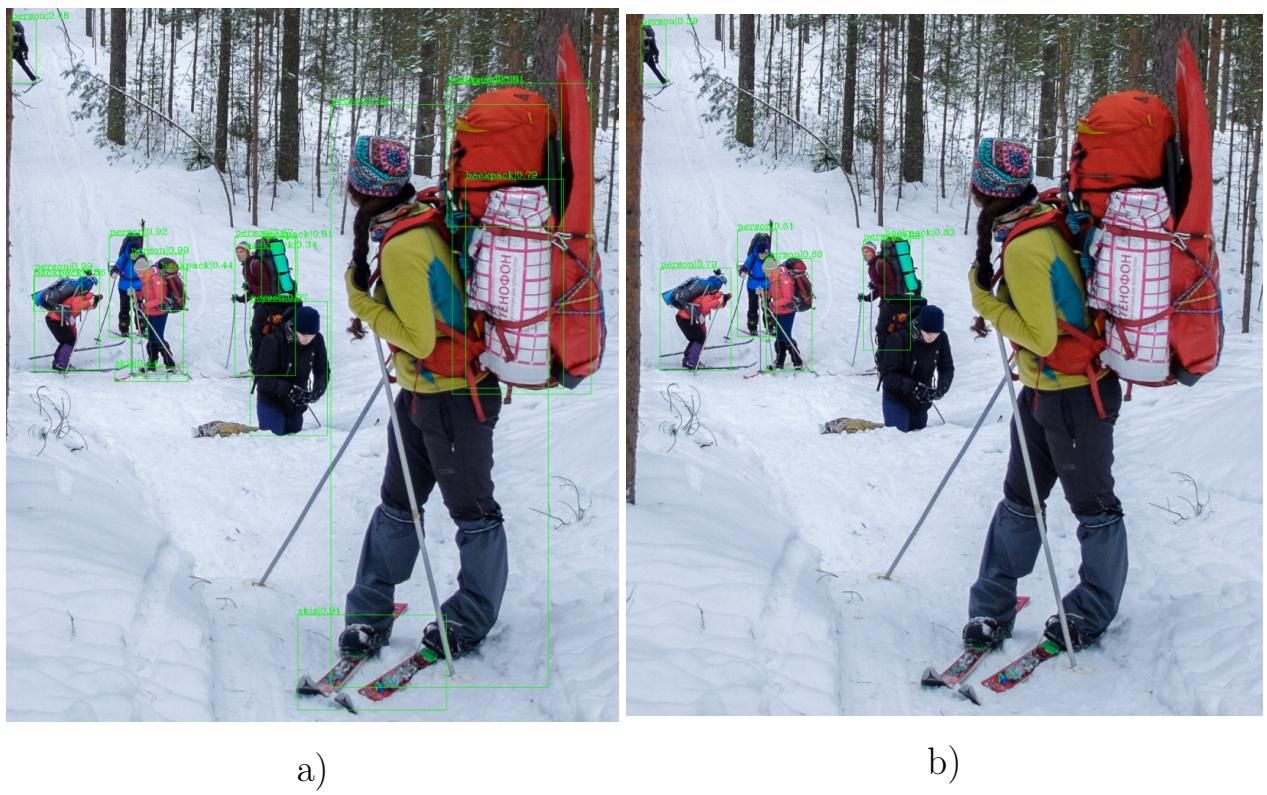
Результаты сравнения на выбранных изображениях представлены на рисунках 36–44.



Рисунок 36 – Предсказание Faster R-CNN на изображении 1



Рисунок 37 – Предсказание RetinaNet на изображении 1



a)

b)

Рисунок 38 – Предсказания на изображении 2: a) Faster R-CNN, b)
RetinaNet



Рисунок 39 – Предсказание Faster R-CNN на изображении 3



Рисунок 40 – Предсказание RetinaNet на изображении 3

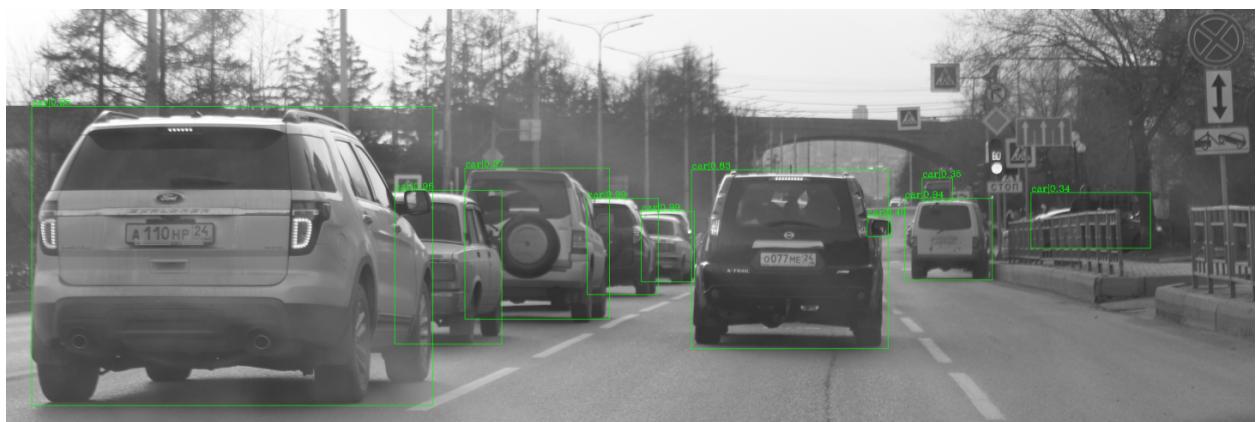


Рисунок 41 – Предсказание Faster R-CNN на изображении 4



Рисунок 42 – Предсказание RetinaNet на изображении 4



Рисунок 43 – Предсказание Faster R-CNN на изображении 5



Рисунок 44 – Предсказание RetinaNet на изображении 5

4.4 Результаты сравнения

В ходе сравнения обнаружено, что RetinaNet плохо обнаруживает крупные объекты, а также хуже обнаруживает разнообразные классы, но неплохо справляется с задачами обнаружения маленьких машин и людей. Время распознавания у Faster R-CNN незначительно меньше (на 2-3 секунды, при времени распознавания одного изображения около 30 секунд), а размер обученной модели незначительно больше (на 15 Мб, при среднем

весе модели в 235 Мб). Итого экспериментальное сравнение на произвольной выборке подтверждает преимущество Faster R-CNN, полученное при валидации на наборе данных COCO. Таким образом из архитектур Faster R-CNN и RetinaNet, основанных на ResNeXt-101-32x8d и обученных на наборе данных COCO, для большинства задач рекомендуется использовать Faster R-CNN, исключением могут быть такие специфические задачи, как распознавание мелких объектов с воздуха.

ЗАКЛЮЧЕНИЕ

В ходе работы было проведено исследование архитектур нейронных сетей для классификации и локализации объектов на изображении:

1. Рассмотрены существующие решения для классификации объектов, среди которых для дальнейшего сравнения были выбраны: ResNet-18, ResNet-50, ResNeXt-101-32x8d.
2. Рассмотрены существующие решения для локализации объектов, среди которых для дальнейшего сравнения были выбраны Faster R-CNN и RetinaNet.
3. Произведено изучение принципов работы выбранных архитектур.
4. Произведено экспериментальное сравнение выбранных архитектур.

В результате сравнения моделей для классификации получены следующие рекомендации для использования: использовать ResNet-18 в задачах, требующих минимальной временной задержки или в условиях минимизации занимаемой программой памяти; использовать ResNeXt-101-32x8d для достижения максимальной точности в задачах, не требовательных к скорости распознавания, в остальных случаях рекомендуется использовать ResNet-50, как компромиссный вариант, обеспечивающий достаточно высокую точность и скорость, приемлемую для распознавания в реальном времени.

Сравнение Faster R-CNN и RetinaNet показало, что первая архитектура превосходит вторую как по качеству работы, так и по скорости распознавания. Наиболее подходящая область для использования RetinaNet, учитывая особенность архитектуры, которая позволяет получать хорошо обобщённые признаки для маленьких объектов, – это распознавание на изображениях, снятых с воздуха.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Нейронные сети и компьютерное зрение [Электронный ресурс]: онлайн курс – URL: <https://stepik.org/course/50352> (дата обращения: 18.02.2020).
2. PyTorch для начинающих: классификация изображений с использованием предварительно подготовленных моделей [Электронный ресурс]: статья – URL: https://ai-news.ru/2019/10/pytorch_dlya_nachinaushih_klassifikaciya_izobrazhenij_s_ispolzovaniem.html (дата обращения: 02.03.2020).
3. Mask R-CNN: архитектура современной нейронной сети для сегментации объектов на изображениях [Электронный ресурс]: статья – URL: <https://habr.com/ru/post/421299/> (дата обращения: 06.03.2020).
4. Распознавание объектов в режиме реального времени на iOS с помощью YOLOv3 [Электронный ресурс]: статья – URL: <https://habr.com/ru/post/460869/> (дата обращения: 16.03.2020).
5. SSD: Single Shot MultiBox Detector (обзор архитектуры) [Электронный ресурс]: видео – URL: <https://medium.com/deepsystems-ru/ssd-single-shot-multibox-detector-%D0%BE%D0%B1%D0%B7%D0%BE%D1%80%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D1%8B-80730ff5735f> (дата обращения: 07.03.2020).
6. Обнаружение объектов на аэрофотоснимках с использованием RetinaNet [Электронный ресурс]: статья – URL: <https://www.machinelearningmastery.ru/object-detection-on-aerial-imagery-using-retinanet-626130ba2203/> (дата обращения: 07.03.2020).
7. Cleaned vs Dirty V2: Classify if a plate is cleaned or dirty? [Электронный ресурс]: набор данных – URL: <https://www.kaggle.com/c/platesv2> (дата обращения: 24.03.2020).
8. Stanford Dogs Dataset [Электронный ресурс]: набор данных –

URL: <http://vision.stanford.edu/aditya86/ImageNetDogs/> (дата обращения: 30.04.2020).

9. imagenet1000_clsidx_to_labels.txt [Электронный ресурс]: текстовый файл – URL: <https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a> (дата обращения: 02.05.2020).
10. R-CNN, Fast R-CNN, Faster R-CNN etc [Электронный ресурс]: статья – URL: https://vbystricky.github.io/2017/06/rcnn_etc.html (дата обращения: 12.05.2020).
11. Understanding Feature Pyramid Networks for object detection (FPN) [Электронный ресурс]: статья – URL: https://medium.com/@jonathan_hui/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c (дата обращения: 15.05.2020).
12. The intuition behind RetinaNet [Электронный ресурс]: статья – URL: <https://medium.com/@14prakash/the-intuition-behind-retinanet-eb636755607d> (дата обращения: 15.05.2020).
13. An Introduction to Evaluation Metrics for Object Detection [Электронный ресурс]: статья – URL: <https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/> (дата обращения: 17.05.2020).
14. mmdetection [Электронный ресурс]: библиотека – URL: <https://github.com/open-mmlab/mmdetection> (дата обращения: 18.05.2020).

ПРИЛОЖЕНИЕ А

Сравнение архитектур

Импорт готовых моделей, загрузка моделей и печать параметров:

```
from torchvision import models

def count_parameters(model):
    print(sum(p.numel() for p in model.parameters() if p.requires_grad))
    for name, param in model.named_parameters():
        if param.requires_grad:
            print(name, param.data.shape)

resnet18 = models.resnet18(pretrained=True)
print(resnet18)
count_parameters(resnet18)

resnet50 = models.resnet50(pretrained=True)
print(resnet50)
count_parameters(resnet50)

resnext101 = models.resnext101_32x8d(pretrained=True)
print(resnext101)
count_parameters(resnext101)
```

ПРИЛОЖЕНИЕ Б

Предобработка данных

```
import torchvision
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class ImageFolderWithNames(torchvision.datasets.ImageFolder):
    def __getitem__(self, index):
        original_tuple = super(ImageFolderWithNames, self).__getitem__(index)
        path = self.imgs[index][0]
        return original_tuple[0], path[path.rfind('/') + 1:path.rfind('.')] 

class ImageFolderForDogs(torchvision.datasets.ImageFolder):
    def __getitem__(self, index):
        original_tuple = super(ImageFolderForDogs, self).__getitem__(index)
        if original_tuple[1] < 100:
            target = original_tuple[1]+151
        elif original_tuple[1] < 117:
            target = original_tuple[1] + 152
        else:
            target = original_tuple[1] + 156
        return original_tuple[0], target

def write_plates_result(data, p):
    submission_df = pd.DataFrame.from_dict({'id': data[0], 'label': data[1]})
    submission_df['label'] = submission_df['label'].map(lambda pred: 'dirty'
        if pred >= p else 'cleaned')
    submission_df.set_index('id', inplace=True)
    submission_df.to_csv('submission.csv')

class PreprocessData:
    def __init__(self, datasetdir):
        self.img_size = 224
        self.mean = np.array([0.485, 0.456, 0.406])
        self.std = np.array([0.229, 0.224, 0.225])
        self.datasetdir = datasetdir
```

```

        self.train_transforms = torchvision.transforms.Compose([
            torchvision.transforms.Resize((224, 224)),
            torchvision.transforms.RandomHorizontalFlip(),
            torchvision.transforms.RandomVerticalFlip(),
            torchvision.transforms.ToTensor(),
            torchvision.transforms.Normalize(self.mean, self.std)
        ])

        self.test_transforms = torchvision.transforms.Compose([
            torchvision.transforms.Resize((224, 224)),
            torchvision.transforms.ToTensor(),
            torchvision.transforms.Normalize(self.mean, self.std)
        ])

    self.train_dataset = torchvision.datasets.ImageFolder(
        self.datasetdir + 'train', self.train_transforms)
    self.val_dataset = torchvision.datasets.ImageFolder(
        self.datasetdir + 'val', self.test_transforms)
    self.test_dataset = ImageFolderWithNames(datasetdir + 'test',
        self.test_transforms)

def train_with_random_resize(self):
    self.train_transforms = torchvision.transforms.Compose([
        torchvision.transforms.RandomResizedCrop(224),
        torchvision.transforms.RandomHorizontalFlip(),
        torchvision.transforms.RandomVerticalFlip(),
        torchvision.transforms.ColorJitter(),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(self.mean, self.std)
    ])

    self.train_dataset = torchvision.datasets.ImageFolder(
        self.datasetdir + 'train', self.train_transforms)

def show_input(self, input_tensor, title):
    image = input_tensor.permute(1, 2, 0).numpy()
    image = self.std * image + self.mean
    plt.imshow(image.clip(0, 1))
    plt.title(title)
    plt.show()
    plt.pause(0.001)

```

ПРИЛОЖЕНИЕ В

Работа с моделями для классификации

```
import torch
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from tqdm import tqdm
import pickle

def load_model(name):
    with open('models/' + name + '.pickle', 'rb') as f:
        model = pickle.load(f)
    return model

class ClassificationModel:
    def __init__(self, model, num_class, name):
        self.model = model
        self.name = name
        self.num_class = num_class
        for param in self.model.parameters():
            param.requires_grad = False

        self.model.fc = torch.nn.Linear(model.fc.in_features, num_class)
        self.device = torch.device("cuda:0" if torch.cuda.is_available()
                                  else "cpu")
        print("device: ", self.device)
        self.model = model.to(self.device)

        self.loss = torch.nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(model.parameters(), lr=5.0e-2)
        self.scheduler = torch.optim.lr_scheduler.StepLR(self.optimizer,
                                                       step_size=4, gamma=0.5)

        self.train_loss_hist = [1.]
        self.train_acc_hist = [0.]
        self.val_loss_hist = [1.]
        self.val_acc_hist = [0.]
        self.epochs = 0
```

```

        self.val_predictions = []
        self.val_targets = []
        self.best_val_acc = 0

def train_model(self, data, num_epochs, n_val, batch_size):
    for epoch in range(1, num_epochs + 1):
        if epoch == 10:
            data.train_with_random_resize()
        self.epochs += 1
        print('Epoch {}/{}:'.format(epoch, num_epochs), flush=True)
        dataloader = torch.utils.data.DataLoader(data.train_dataset,
                                                batch_size=batch_size, shuffle=True, num_workers=batch_size,
                                                drop_last=True)
        self.model.train()
        running_loss = 0.
        running_acc = 0.
        for inputs, labels in tqdm(dataloader):
            inputs = inputs.to(self.device)
            labels = labels.to(self.device)
            self.optimizer.zero_grad()
            with torch.set_grad_enabled(True):
                preds = self.model(inputs)
                loss_value = self.loss(preds, labels)
                preds_class = preds.argmax(dim=1)
                loss_value.backward()
                self.optimizer.step()
                self.scheduler.step()
                running_loss += loss_value.item()
                running_acc += (preds_class == labels.data).float().mean()
            epoch_loss = running_loss / len(dataloader)
            epoch_acc = running_acc / len(dataloader)
            self.train_loss_hist.append(epoch_loss)
            self.train_acc_hist.append(epoch_acc)
            print('train loss: {:.4f} Acc: {:.4f}'.format(epoch_loss,
                                                          epoch_acc), flush=True)
            if epoch % n_val == 0:
                self.val_model(data, batch_size)

def val_model(self, data, batch_size):
    val_predictions = []

```

```

val_targets = []
dataloader = torch.utils.data.DataLoader(data.val_dataset,
                                         batch_size=batch_size, shuffle=False, num_workers=batch_size)
self.model.eval() # Set model to evaluate mode
running_loss = 0.
running_acc = 0.
for inputs, labels in tqdm(dataloader):
    inputs = inputs.to(self.device)
    labels = labels.to(self.device)
    with torch.set_grad_enabled(False):
        preds = self.model(inputs)
        loss_value = self.loss(preds, labels)
        preds_class = preds.argmax(dim=1)
        running_loss += loss_value.item()
        running_acc += (preds_class == labels.data).float().mean()
        val_targets.extend(labels.data.cpu().numpy())
        val_predictions.extend(torch.nn.functional.softmax(preds, dim=1)
                               .data.cpu().numpy())
epoch_loss = running_loss / len(dataloader)
epoch_acc = running_acc / len(dataloader)
self.val_loss_hist.append(epoch_loss)
self.val_acc_hist.append(epoch_acc)
print('VAL loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc),
      flush=True)
self.val_predictions = np.array(val_predictions)
self.val_targets = np.array(val_targets)
if epoch_acc > self.best_val_acc:
    self.best_val_acc = epoch_acc
    self.save_model()

def count_metrics_for_class(self, class_id):
    class1 = self.val_predictions[:, class_id]
    labels = self.val_targets
    tpr = []
    fpr = []
    plt.figure(1, figsize=(12, 6))
    for i in range(1, 10):
        p = float(i) / 10
        plt.subplot(3, 3, i)
        tp = (labels[class1 >= p] == class_id).sum()

```

```

fp = (labels[class1 >= p] != class_id).sum()
tn = (labels[class1 < p] != class_id).sum()
fn = (labels[class1 < p] == class_id).sum()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
print('Порог: {:.1f}, TP: {}, FP: {}, FN: {}, TN: {}, acc:{:.4f},'
      'precision: {:.4f}, recall: {:.4f}, F: {:.4f}'
      .format(p, tp, fp, fn, tn, (tp + tn) / (tp + tn + fp + fn),
              precision, recall, 2 * precision * recall /
              (precision + recall)), flush=True)
names = [['TP: %i' % tp, 'FP: %i' % fp],
          ['FN: %i' % fn, 'TN: %i' % tn]]
sns.heatmap([[tp, fp], [fn, tn]], vmin=0, vmax=len(self.val_targets),
            annot=names, fmt = '', xticklabels=['$y=1$', '$y=0$'],
            yticklabels=['$\hat{y}=1$', '$\hat{y}=0$'])
tpr.append(recall)
fpr.append(fp / (fp + tn))
plt.figure(2, figsize=(3, 3))
plt.plot(fpr, tpr, label='ROC')
plt.title('ROC curve')
plt.legend()
plt.show()
plt.pause(0.001)

def count_metrics_for_model(self):
    f_mera = 0
    fig = 0
    pos = 0
    for class_id in range(self.num_class):
        preds_class = self.val_predictions.argmax(axis=1)
        labels = self.val_targets
        if class_id % 30 == 0:
            fig += 1
            plt.figure(fig, figsize=(18, 15))
            pos = 0
        pos += 1
        plt.subplot(5, 6, pos)
        tp = (labels[preds_class == class_id] == class_id).sum()
        fp = (labels[preds_class == class_id] != class_id).sum()
        tn = (labels[preds_class != class_id] != class_id).sum()

```

```

fn = (labels[preds_class != class_id] == class_id).sum()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f_mera += 2 * precision * recall / (precision + recall)
print('Класс: {}, TP: {}, FP: {}, FN: {}, TN: {}, acc:{:.4f},'
      'precision: {:.4f}, recall: {:.4f}, F: {:.4f},'
      .format(class_id, tp, fp, fn, tn, (tp + tn) /
              (tp + tn + fp + fn), precision, recall,
              2 * precision * recall / (precision + recall)), flush=True)
names = [['TP: %i' % tp, 'FP: %i' % fp],
          ['FN: %i' % fn, 'TN: %i' % tn]]
sns.heatmap([[tp, fp], [fn, tn]], vmin=0, vmax=len(self.val_targets),
            annot=names, fmt='', xticklabels=['$y=1$', '$y=0$'],
            yticklabels=['$\hat{y}=1$', '$\hat{y}=0$'])
plt.show()
plt.pause(0.001)
print('Average F: {:.4f}'.format(f_mera/num_class), flush=True)

def plot_metrics(self, num_val):
    plt.figure(3, figsize=(6, 3))
    plt.subplot(1, 2, 1)
    plt.plot(np.arange(self.epochs + 1), self.train_acc_hist, label='train')
    plt.plot(np.arange(0, self.epochs + 1, num_val), self.val_acc_hist,
             label='val')
    plt.title('Accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(np.arange(self.epochs + 1), self.train_loss_hist, label='train')
    plt.plot(np.arange(0, self.epochs + 1, num_val), self.val_loss_hist,
             label='val')
    plt.title('Loss')
    plt.legend()
    plt.show()
    plt.pause(0.001)

def predict(self, data, batch_size):
    self.model.eval()
    test_predictions = []
    test_img_idx = []
    dataloader = torch.utils.data.DataLoader(data.test_dataset,

```

```
batch_size= batch_size, shuffle=False, num_workers=batch_size)
for inputs, idx in tqdm(dataloader):
    inputs = inputs.to(self.device)
    with torch.set_grad_enabled(False):
        preds = self.model(inputs)
    test_predictions.extend(
        torch.nn.functional.softmax(preds, dim=1)[:, 1].data.cpu().numpy())
    test_img_idx.extend(idx)
return test_img_idx, test_predictions

def save_model(self):
    file_name = 'models/{}-{}-{:.4f}.pickle'.format(self.name,
                                                    self.epochs, self.best_val_acc)
    print('saving model with name: "{}"'.format(file_name))
    with open(file_name, 'wb') as f:
        pickle.dump(self, f)
```

ПРИЛОЖЕНИЕ Г

Классификация на маленьком наборе данных

```
num_epoch = 100
num_val = 5
batch_size = 20
plates = PreprocessData('./datasets/plates/')

resNet18forPlates = ClassificationModel(models.resnet18(pretrained=True),
                                         2, 'resNet18forPlates')
start_time = time.time()
resNet18forPlates.train_model(plates, num_epoch, num_val, batch_size)
print("{:.2f} minutes".format((time.time() - start_time)/60))

resNet50forPlates = ClassificationModel(models.resnet50(pretrained=True),
                                         2, 'resNet50forPlates')
start_time = time.time()
resNet50forPlates.train_model(plates, num_epoch, num_val, batch_size)
print("{:.2f} minutes".format((time.time() - start_time)/60))

resNeXt101forPlates = ClassificationModel(models.resnext101_32x8d(
                                             pretrained=True), 2, 'resNeXt101forPlates')
start_time = time.time()
resNeXt101forPlates.train_model(plates, num_epoch, num_val, batch_size)
print("{:.2f} minutes".format((time.time() - start_time)/60))

resNet18forPlates.plot_metrics(num_val)
resNet18forPlates.count_metrics_for_class(1)

resNet50forPlates.plot_metrics(num_val)
resNet50forPlates.count_metrics_for_class(1)

resNeXt101forPlates.plot_metrics(num_val)
resNeXt101forPlates.count_metrics_for_class(1)

start_time = time.time()
test_img_idx, test_predictions = resNet18forPlates.predict(plates, batch_size)
print("{:.3f} seconds".format((time.time() - start_time)/744))
```

```
start_time = time.time()
test_img_idx, test_predictions = resNet50forPlates.predict(plates, batch_size)
print("{:.3f} seconds".format((time.time() - start_time)/744))

start_time = time.time()
test_img_idx, test_predictions = resNeXt101forPlates.predict(plates, batch_size)
print("{:.3f} seconds".format((time.time() - start_time)/744))

write_plates_result((test_img_idx, test_predictions), 0.5)
```

ПРИЛОЖЕНИЕ Д

Классификация на части ImageNet

```
from torchvision import models
from preprocessdata import *
from tqdm import tqdm
import seaborn as sns
import torchvision
import torch

def pred(model, dataset):
    loss = torch.nn.CrossEntropyLoss()
    val_predictions = []
    val_targets = []
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=20,
                                              shuffle=False, num_workers=20)
    model.eval() # Set model to evaluate mode
    running_loss = 0.
    running_acc = 0.
    for inputs, labels in tqdm(dataloader):
        with torch.set_grad_enabled(False):
            preds = model(inputs)
            loss_value = loss(preds, labels)
            preds_class = preds.argmax(dim=1)
            running_loss += loss_value.item()
            running_acc += (preds_class == labels.data).float().mean()
            val_targets.extend(labels.data.numpy())
            val_predictions.extend(torch.nn.functional.softmax(preds, dim=1)
                                  .data.numpy())
    epoch_loss = running_loss / len(dataloader)
    epoch_acc = running_acc / len(dataloader)
    print('VAL loss: {:.4f} Acc: {:.4f}'.format(epoch_loss, epoch_acc),
          flush=True)
    return np.array(val_predictions), np.array(val_targets)

def pred_maps(val_predictions, labels):
    f_mera = 0
    fig = 0
    pos = 0
```

```

class_ids = [i for i in range(151, 276)]
class_ids.remove(251)
class_ids.remove(269)
class_ids.remove(270)
class_ids.remove(271)
class_ids.remove(272)
for class_id in class_ids:
    preds_class = val_predictions.argmax(axis=1)
    if pos % 30 == 0:
        fig += 1
        plt.figure(fig, figsize=(18, 15))
        pos = 0
    pos += 1
    plt.subplot(5, 6, pos)
    tp = (labels[preds_class == class_id] == class_id).sum()
    fp = (labels[preds_class == class_id] != class_id).sum()
    tn = (labels[preds_class != class_id] != class_id).sum()
    fn = (labels[preds_class != class_id] == class_id).sum()
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f_mera += 2 * precision * recall / (precision + recall)
    print('Класс: {}, TP: {}, FP: {}, FN: {}, TN: {}, acc:{:.4f}, '
          'precision: {:.4f}, recall: {:.4f}, F: {:.4f} ' +
          .format(class_id, tp, fp, fn, tn, (tp + tn) /
                  (tp + tn + fp + fn), precision, recall,
                  2 * precision * recall / (precision + recall)), flush=True)
names = [['TP: %i' % tp, 'FP: %i' % fp,
          'FN: %i' % fn, 'TN: %i' % tn]]
sns.heatmap([[tp, fp], [fn, tn]], vmin=0, vmax=len(labels),
            annot=names, fmt='', xticklabels=['$y=1$', '$y=0$'],
            yticklabels ['$\hat{y}=1$', '$\hat{y}=0$'])
plt.show()
plt.pause(0.001)
print('Average F: {:.4f}'.format(f_mera / 120), flush=True)

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(np.array([0.485, 0.456, 0.406]),
                                   np.array([0.229, 0.224, 0.225]))])

```

```
])

dogs = ImageFolderForDogs('./datasets/dogs/test', test_transforms)

model = models.resnet18(pretrained=True)
pred_maps(pred(model, dogs))

model = models.resnet50(pretrained=True)
pred_maps(pred(model, dogs))

model = models.resnext101_32x8d(pretrained=True)
pred_maps(pred(model, dogs))
```