Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Метод распознавания летательных аппаратов с аэрофотоснимков с использованием нейронных сетей»

Студент	ИУ7-81Б		Мицевич М. Д.
	(Группа)	(Подпись, дата)	(И. О. Фамилия)
Руководитель ВКР			Тассов К. Л.
		(Подпись, дата)	(И. О. Фамилия)
Нормоконтролер			Мальцева Д. Ю.
		(Подпись, дата)	(И. О. Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 89 с., 24 рис., 1 табл., 29 источн., 1 прил.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

—

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие сокращения и обозначения.

—

ВВЕДЕНИЕ

Одной из основных задач, возникающих при обработке изображений, является распознавание различных объектов на снимке. В роли объектов может быть различная техника: вертолеты, самолеты, машины, корабли.

Обработка в реальном времени полезна для задач контроля движения судов, поиска объектов на местности в случае аварии и крушения, предотвращение таких ситуаций, картографии.

Распознавание объектов с аэрофотоснимков полезно во время непрерывно ведущегося наблюдения с воздуха. В этом случае можно вести наблюдение с воздуха и в автоматическом режиме выдавать информацию о происходящем на земле.

К примеру, в настоящее время один из способов разведки территорий – съемка с беспилотных летательных аппаратов. Человек в ручном режиме не всегда может обработать информацию, которая непрерывным потоком поступает с беспилотника, ведущего разведку, поэтому весь поток информации нужно обрабатывать автоматически.

В случае разведки летательной техники важным является определение моделей самолетов, стоящий в аэропорту. Такая классификации в случае военной техники поможет сделать вывод о готовящейся тактике и силах стороны, территория которой разведывается.

Целью данной работы является разработка метода распознавания летательных аппаратов с аэрофотоснимков.

Для достижения поставленной цели требуется выполнить следующие задачи:

- определить термины, связанные с предметной областью распознавания объектов;
- провести анализ методов распознавания летальной техники на аэрофотоснимках;
- определить критерии сравнения методов;
- провести их сравнительный анализ;
- спроектировать метод распознавания летальной техники на аэрофотоснимках;

- реализовать спроектированный метод;
- провести исследование точности распознавания модели на тестовой выборке при различных подходах к обучению.

1 Аналитический раздел

1.1 Анализ предметной области

1.1.1 Детерминированный подход

Детерминированный подход основан на сравнении признаков распознаваемого объекта с признаками эталона и предполагает, что в любой точке пространства признаков могут появляться реализации только одного класса объектов. Алгоритмы данного подхода должны иметь возможности к автосмещению, автомасштабированию и автоповороту.

Одним из таких является алгоритм распознавания изображений на основе градиентного совмещения объекта с эталоном, предложенный в сборнике статей [1].

Все алгоритмы детерминированного подхода можно разбить на 4 основных шага:

1. выделение контуров объектов;
2. выбор наиболее информативных узловых точек эталона и объекта, а также установление связи между ними;
3. совмещение точек, на этом шаге должна осуществляться инвариантность к смещению, повороту и масштабированию;
4. принятие решения о принадлежности объекта к тому или иному классу.

Преимуществом такого подхода является то, что для классификации не требуется никаких данных, кроме эталонных изображений.

Недостатком является сложность получения устойчивого контура на зашумленных изображениях.

1.1.2 Экспертный подход

Методы экспертного подхода основаны на разделении всего множества входных объектов по некоторым заранее заданным критериям. Одним из таких методов является дерево решений.

Дерево решений представляет собой иерархическую древовидную структуру в узлах, которой содержатся условия. Условия генерируются при со-

здании дерева, то есть в процессе обработки обучающего множества данных. Листьями дерева являются конкретные классы по которым производится классификация.

Процесс построения дерева решений предполагает рекурентное последовательное разбиение обучающего множества на подмножества с применением решающий правил в узлах дерева. Процесс разбиения продолжается до тех пор, пока во всех листах не будет содержаться конкретных класс, по которому производится классификация. Узел становится листом либо когда он содержит объект одного класса, либо когда достигнуто какое-либо из условий остановки, к примеру, достигнута максимальная глубина дерева.

При построении дерева используются жадные алгоритмы, которые выбирают оптимальные решения для конкретного шага алгоритма, а не всей системы целиком. То есть при выборе условия разбиения множества на два подмножества алгоритм будет выбирать лучшее только для этого шага и не сможет в дальнейшем вернуться и изменить его, даже если это будет оптимальнее для всей системы целиком.

На каждом этапе выбирается одно условие из всех возможных по средством максимизации функции прироста информации [2]. Для ее вычисления необходимо сначала задать функцию ошибки. К примеру, можно использовать среднюю квадратичную ошибку 1.1

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(ytrue - ypred_{\text{i}})^2, \tag{1.1}$$

где n – число элементов в подмножестве, $ytrue$ – значение атрибута в узле дерева, $ypred_{\text{i}}$ – значение атрибута в i элементе подмножества.

Тогда функция прироста информации может быть выражена зависимостью 1.2

$$IG = MSE_{\text{root}} - (\frac{n_{\text{left}}}{n}MSE_{\text{left}} + \frac{n_{\text{right}}}{n}MSE_{\text{rigth}}), \tag{1.2}$$

где MSE_{root} – значение ошибки в корневом узле, MSE_{left} – значение ошибки в левом подмножестве, MSE_{right} – в правом подмножестве.

Таким образом, на каждом шаге построения дерева из всех возможных условий выбирается то, для которого значение функции прироста информации будет наибольшим. В узлах, в которых значение функции ошибки равно 0 или остался всего один элемент, процесс построения дерева останавливается и

они считаются листами.

Недостатками дерева решений при работе с изображениями являются неустойчивость к шуму и необходимость выделения критериев разделения на этапе построения дерева.

1.1.3 Нейрокомпьютерный подход

Методы данного подхода основаны на создании и последующим обучении некоторой математической модели. Алгоритмы данного подхода можно разделить на две категории:

1. обучение с учителем;
2. обучение без учителя.

При обучении с учителем каждый прецедент представляет собой пару «объект, ответ». Требуется найти функциональную зависимость ответов от описаний объектов и построить алгоритм, принимающий на входе описание объекта и выдающий на выходе ответ.

В случае обучения без учителя ответы не задаются и требуется найти зависимость между объектами.

К задачам, которые решаются с помощью алгоритмов обучения с учителем можно отнести следующие:

- классификация – результатом является значение из конечного множества классов;
- детекция – результатом является описание положений объектов на изображении.

К задачам обучения без учителя могут быть отнесены следующие:

- задача кластеризации, целью которой является группировка объектов в кластеры на основе данных об их попарном сходстве;
- задача фильтрации выбросов, целью которой является выделение в обучающей выборке нетипичных элементов.

К алгоритмам обучения с учителем относятся нейронные сети.

1.1.4 Вывод

Методы детерминированного подхода не подходят для решения задачи распознавания летательных аппаратов на аэрофотоснимках, так как не устойчивы к шумам в входном изображении и требуют эталонных образцов, на основе которых будет проводиться определение объектов.

Дерево решений не подходит, так как оно также не устойчиво к шумам и требует выделение критериев разделения выборки на этапе построения.

Таким образом, для решения поставленной задачи нужно использовать методы нейрокомпьютерного подхода, а именно нейронные сети.

1.2 Нейронные сети

Модель нейронной сети основана на биологическом нейроне. У нейрона есть ядро, которое называется телом. В теле накапливается электрический заряд. С телом соединены отростки. Отростки, по которым сигнал поступает в тело, называются дендритами. Отросток, по которому сигнал передается другим нейронам, называется аксоном. Место, где аксон соединяется с дендритами, называется синапсом. Синапс отвечает за количество заряда, которое перейдет от аксона к дендриту. Синапс может изменяться со временем. Именно с настройкой синапса и связана тренировка биологической нейронной сети.

1.2.1 Математическая модель МакКаллока-Питтса

В математической модели МакКаллока-Питтса, тело нейрона, где накапливается заряд, заменяется на сумматор. Дендриты являются входами сумматора, а выходом – аксоном. Биологический нейрон накапливает заряд до тех пор, пока этот заряд не достигнет какого-то значения, и только после этого этот заряд уходит по аксону к другим нейронам. В математической модели к сигналу после выхода из сумматора применяется функция активации и только после этого сигнал попадает на дендрит следующего нейрона. Синапсы в математической модели заменяются на веса входов нейрона. Математическая модель нейрона выражается зависимостью 1.3

$$y = f\left(\sum_{i=1}^{n}(w_{\text{i}}x_{\text{i}}) + b\right), \tag{1.3}$$

где y – сигнал на выходе из нейрона, f – функция активации, w_i – вес i входа, x_i – сигнал этого входа, b – некоторое значение смещения, которое задается отдельно для каждого нейрона. Обучение нейронной сети происходит за счет настройки синаптических весов w_i и смещения b.

1.2.2 Функции активации

Существует много различных функций активации (фактически любая функция может быть функцией активации). Наиболее популярными считаются логистическую функцию, гиперболический тангенс, ReLU [3]. Важной особенностью функций активации является их дифференцируемость (хотя для некоторых функций это выполняется не всегда), поскольку при обратном распространении ошибки необходимо вычислять градиенты, использующие производную функции активации.

Логистическая функция преобразовывает поступающие в неё значения в вещественный диапазон [0, 1]. Это означает, что при x>0 выходное значение будет примерно равно единице, а при x<0 будет близким к нулю. Данная функция часто используется в задачах классификации [3]. Логистическая функция определяется зависимостью 1.4.

$$y = \frac{1}{1 + e^{-x}}. \tag{1.4}$$

Гиперболический тангенс схож с логистической функцией, но в отличии от нее может принимать отрицательные значения. Гиперболический тангенс определяется зависимостью 1.5.

$$y = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{1.5}$$

Функция ReLU возвращает 0, если принимает отрицательный аргумент, в случае же положительного аргумента, функция возвращает само число. Функция ReLU определяется зависимостью 1.6.

$$\text{ReLU}(x) = \begin{cases} x, & \text{если } x > 0 \\ 0, & \text{иначе} \end{cases} \tag{1.6}$$

ReLU решает проблему обнуления градиента (ситуация, при которой во

время обучения градиенты по всем весам становятся близкими или равными нулю) для положительных чисел, также она вычисляется гораздо проще, чем сигмоидальные функции (логистическая функция, гиперболический тангенс) [3].

1.2.3 Составляющие нейронной сети

При обучении нейронной сети используются две подвыборки обучающего множества. Вся обучающая выборка состоит из какого-то количества объектов, для которых известны признаки, на которые должна обучиться нейронная сеть. Первая подвыборка называется тренировочной и используется для итеративного обучения нейронной сети. Вторая называется тестовой и используется для оценки того, насколько хорошо обучена нейронная сеть.

Нейронную сеть определяют следующие параметры:

- архитектура нейронной сети – отвечает за то, как нейроны связаны между собой;
- функция потерь – определяет насколько точно работает модель [4];
- метод оптимизации – определяет способ уменьшения функции потерь на каждой итерации обучения.

Нейроны делятся на три типа: входной, скрытый и выходной. В том случае, когда нейросеть состоит из большого количества нейронов, вводят термин слоя. Соответственно, есть входной слой, который получает информацию, некоторое количество скрытых, которые ее обрабатывают и выходной слой, который выводит результат [5]. Количество скрытых слоев и число нейронов в каждом из них задают архитектуру нейронной сети.

1.2.4 Методы оптимизации

Самый используемый метод оптимизации – градиентный спуск [6]. Градиентный спуск основан на пошаговом приближении функции к локальному минимуму. На каждой итерации алгоритма новые значения получаются по формуле 1.7

$$w_1 = w_0 - \alpha \Delta f(w_0), \tag{1.7}$$

где w_1 – вектор новых значений, которые подбираются алгоритмом, w_0 – значения параметров на текущем шаге, $\Delta f(w_0)$ – вектор градиентов функции потерь по каждому из параметров на текущем шаге, α – скорость обучения.

На каждой итерации градиентного спуска требуется считать градиент функции потерь, которая зависит от функций активации каждого из нейронов сети. В связи с этим к фукнциям потерь и активации применяются требования по дифференцируемости.

В связи с тем, что градиентный спуск находит только локальный минимум, не всегда полученный результат будет оптимальным. Результат работы алгоритма зависит от изначальных настроек параметров нейронной сети.

Выделяют три основных типа градиентного спуска [6]:

- мини-пакетный градиентный спуск – в этом случае обучающий набор данных разбивается на небольшие партии, которые используются для расчета ошибки модели и обновления коэффициентов модели;
- стохастический градиентный спуск – в этом случае градиент оптимизируемой функции считается на каждом шаге не как сумма градиентов от каждого элемента выборки, а как градиент от одного, случайно выбранного элемента;
- пакетный градиентный спуск – это разновидность алгоритма градиентного спуска, который вычисляет ошибку для каждого примера в наборе обучающих данных, но обновляет модель только после того, как все обучающие примеры были оценены.

1.2.5 Функции потерь

Согласно исследованиям [7] для задачи классификации изображений самой эффективной функцией потерь являются категориальная перекрестная энтропия, которая определяется выражением 1.8

$$CM_{\text{i}} = -\sum_{i=1}^{N} t_{\text{i}} \log p_{\text{i}}, \tag{1.8}$$

где N – число классов классификации, t_{i} – 0 или 1 в зависимости от того принадлежит ли изображение на входе нейронной сети классу, за который отвечает i нейрон выходного слоя, p_{i} – результат на выходе из нейрона.

В задачах классификации используют категориальную перекрестную энтропию в качестве функции потерь. В таких случаях на выходном слое нейронной сети создается столько нейронов, сколько возможных классов может иметь объект на входе. В качестве функции активации для каждого из таких нейронов используют софт макс. Софт макс определяется выражением 1.9

$$SM_{\text{i}} = \frac{e^{y_{\text{i}}}}{\sum_{i=1}^{N} e^{y_{\text{j}}}}, \tag{1.9}$$

где y_{i} – результат на выходе из нейрона, к которому применяется функция активации, N – число нейронов в выходном слое, y_{j} – результат на выходе из j нейрона выходного слоя.

Знаменатель в выражении 1.9 отвечает за нормировку. Таким образом, каждый из нейронов выходного слоя показывает вероятность принадлежности объекта на входе нейронной сети к некоторому классу, а сумма всех этих вероятностей будет равна 1.

1.2.6 Вывод

Нейронная сеть является некоторой математической моделью, параметры которой настаиваются на этапе ее обучения.

Для создания нейронной сети требуется определить следующие параметры:

- архитектура нейронной сети – отвечает за то, как нейроны связаны между собой;
- функция потерь – определяет насколько точно работает модель;
- метод оптимизации – определяет способ уменьшения функции потерь на каждой итерации обучения.

1.3 Принципы построения нейронных сетей

1.3.1 Перцептрон

Перцептрон – математическая модель восприятия информации головным мозгом. Перцептрон состоит из трёх типов элементов, а именно: посту-

пающие от сенсоров сигналы передаются ассоциативным элементам, а затем реагирующим элементам [8].

Каждый из типов элементов относится к определенному слою в архитектуре нейронной сети. Так все сенсоры располагаются на входном слое, ассоциативные элементы находятся на одном или нескольких скрытых слоях, реагирующие элементы занимают выходной слой. Общий вид перцептрона с тремя слоями приведен на рисунке 1.1.

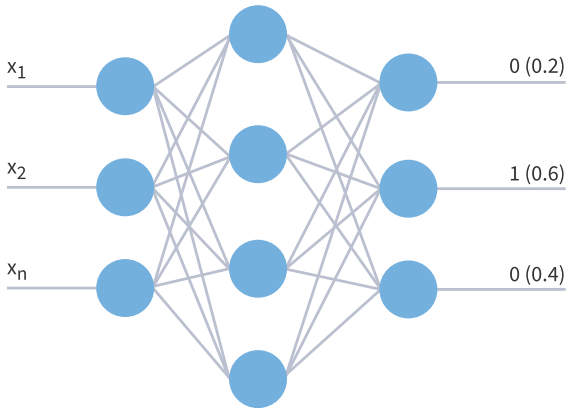

Рисунок 1.1 – Вид перцептрона с тремя слоями

На этом рисунке x_1, x_2 и x_3 обозначают входы перцептрона, которые поступают на входной слой, следующие 4 нейрона образуют слоя, их выходы поступают на 3 нейрона выходного слоя.

Увеличение числа скрытых слоев или числа нейронов на этом слое не всегда приводит к улучшению точности работы нейронной сети, поэтому данные параметры, как правило, подбираются экспериментальным путем. Число нейронов на выходном слое соответствует числу классов, по которым проводится классификация. На входном слое перцептрона число нейронов равно числу пикселей на изображении, которые подаются на вход нейронной сети.

Для нейронов скрытого слоя применяется функция активации Relu, а для нейронов выходного слоя – софт макс.

1.3.2 Сверточные нейронные сети

Свёрточная нейронная сеть — нейронная сеть, в которой присутствует слой свёртки [9]. Свертка из себя представляет некоторую маску, которая

называется ядром. Маска накладывается на пиксели исходного изображения с некоторым шагом, далее значения в маске перемножаются со значениями, которые эта маска покрыла, и результаты перемножений суммируются. Полученная сумма добавляется в результирующую матрицу сверточного слоя. Для сохранения размеров исходного изображения к нему добавляются столбцы и ряды из нулей перед началом свертки.

В случае когда на вход сверточному слою поступает трехканальное изображение, ядро свертки будет не двумерным, а трехмерным. Оно будет состоять из трех матриц – по одной для каждого канала. После применения свертки поочередно к каждому из каналов результаты суммируются и записываются в результирующую матрицу.

На одном сверточном слое к входной матрице может применяться не одна свертка, а сразу несколько. В таком случае каждая свертка считается по отдельности и записывается в свою результирующую матрицу. Результатом работы такого слоя будет несколько каналов с матрицами. Число каналов равно числу фильтров.

Таким образом, сверточный слой определяется следующими величинами:

- числом нулевых строк и столбцов, которые добавляются к исходному изображению;
- шагом свертки по столбцам;
- шагом свертки по строкам;
- числом каналов на входе;
- числом каналов на выходе.

Помимо сверточных слоев в сверточной нейронной сети присутствуют слой субдискретизации и полносвязный слой. В слое субдискретизации также присутствует свертка, которая с некоторым шагом проходится по входной матрице, только вместо перемножения элементов и последующего суммирования выполняется какая-либо другая операция, к примеру, выбор наибольшего элемента. С помощью слоя субдискретизации достигается устойчивость к небольшим сдвигам входного изображения, а также уменьшается размерность

последующих слоёв. Полносвязный слой – обычный скрытый слой многослойного перцептрона, соединённый со всеми нейронами предыдущего слоя [9].

Общий вид сверточной нейронной сети приведен на рисунке 1.2.

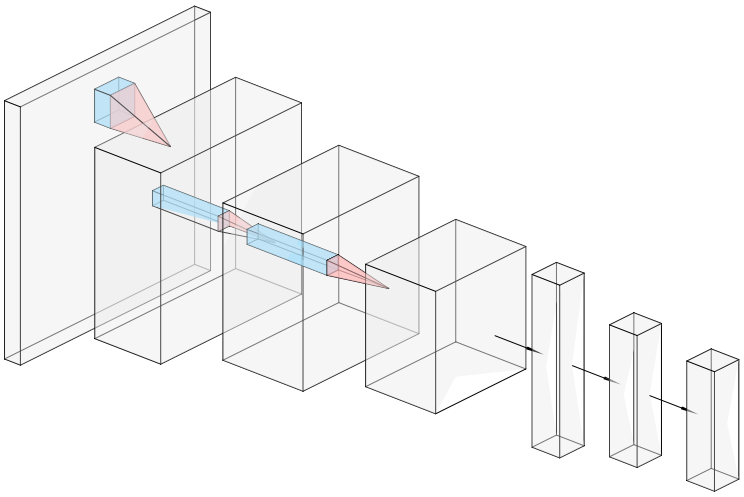

Рисунок 1.2 – Сверточная нейронная сеть с 3 сверточными слоями

На вход этой сверточной сети поступает изображение некоторой размерности, далее к нему применяется свертка с несколькими фильтрами, за счет чего происходит уменьшение размерности и увеличение числа каналов. Данная операция повторяется три раза. Далее все матрицы из всех каналов переводятся в линейный вектор и поступают вход полносвязного слоя. Дальше идет один скрытых слой и выходной аналогично тем, что были в перцептроне.

1.3.3 Капсульные нейронные сети

В капсульных нейронных сетях присутствует капсульный слой. Капсула строится на основе искусственного нейрона, но вместо скалярной расширяет его до векторной формы, что позволяет сохранять больше информации об объекте. На выходе мы получаем вектор, способный сохранять состояние объекта, например его позу [10]. Длина вектора определяет вероятность обнаружения объекта, а его положение отвечает за состояние объекта.

Функция активации в этом случае имеет вид 1.10

$$v_{\text{j}} = \frac{\|s_{\text{j}}\|^2}{1 + \|s_{\text{j}}\|^2} \frac{s_{\text{j}}}{\|s_{\text{j}}\|}, \tag{1.10}$$

где v_{j} – выходной вектор капсулы j, s_{j} – входные данные. Правая часть этого уравнения делает входной вектор единичным, а левая выполняет масштабирование таким образом, чтобы чем длиннее был входной вектор, тем ближе длина выходного была к единице, и чем меньше длина входного, тем ближе длина выходного к нулю.

Для всех капсул, кроме первого слоя, вход s_{j} является взвешанной суммой по всем векторам предсказаний u_{ji} из капсул в нижележащем слое, которые получаются путем умножения выходного u_{i} капсулы из нижележащего слоя на весовую матрицу W_{ij}. Таким образом вход в капсулу s_{j} определяется выражением 1.11

$$s_{\text{j}} = \sum_{i} c_{\text{ij}} u_{\text{ji}}, \tag{1.11}$$

где c_{ij} – коэффициент связи между капсулами i и j, который определяется выражением 1.12

$$c_{\text{ij}} = \frac{e^{b_{\text{ij}}}}{\sum\limits_{k} b_{\text{ik}}}, \tag{1.12}$$

где b_{ij} – вероятность того, что капсула i связана с капсулой j. Эти вероятности итеративно обновляются путем измерения соответствия между текущим выходом v_{j} капсулы j и предсказанием u_{ji} капсулы i. Соответствие считается как скалярное произведение v_{j} на u_{ji}.

Одной из проблем сверточных нейронных сетей является их неспособность сохранять пространственную информацию о входных данных. Например, при обработке изображений обычная нейронная сеть может утерять информацию о положении объектов на изображении. Капсульные нейронные сети решают эту проблему, сохраняя информацию о пространственной структуре входных данных.

1.3.4 Сравнение решений

В исследовании [11] приводится сравнение алгоритмов для решения задачи классификации на наборе изображений рукописных цифр MNIST [12] и наборе изображений техники 10 разных типов CIFAR-10 [13] на основе сверточной нейронной сети и капсульной нейронной сети.

Алгоритмы сравниваются по следующим критериям:

- время обучения нейронной сети;

- число параметров модели;
- точность классификации модели на тестовой выборке.

Сверточная нейронная сеть, участвующая в исследовании, имеет следующие слои:

- слой свертки с 128 фильтрами, ядром размера 5 на 5 пикселей и шагом 2;
- слой max pooling с ядром размера 3 на 3 пикселя и шагом 2;
- слой свертки с 64 фильтрами, ядром размера 5 на 5 пикселей и шагом 1;
- слой max pooling с ядром размера 3 на 3 пикселя и шагом 2;
- полносвязный слой из 1024 нейронов;
- полносвязный слой из 10 нейронов.

В качестве функции активации для всех слоев, кроме последнего, используется ReLU (1.6), на последнем слое используется софт макс (1.9).

Капсульная нейронная сеть , участвующая в исследовании, имеет следующие слои:

- слоя свертки с 256 фильтрами, ядром размера 9 на 9 пикселей и шагом 1;
- слой primary caps, состоящий из 32 капсульных слоев из капсул размерности 8;
- слой digit caps, полученный путем перемножения капсул на матрицы предсказаний размерности 8 на 16 элементов.

Обучение, описанных выше нейронный сетей, происходило на 50 эпохах.

Исходя из результатов исследования [11], время обучения на наборе данных MNIST для капсульной нейронной сети в 30 раз больше, чем в сверточной. Точность классификации для обоих видов сетей составила 99 процентов. В капсульной нейронной сети используется примерно в три раза больше обучаемых параметров, что приводит к большим затратам оперативной памяти.

На наборе данных CIFAR-10 были получены следующие результаты: время обучения капсульной нейронной сети в 10 раз больше, чем у сверточной, число параметров в капсульной приблизительно в три раза больше, чем в сверточной, точность распознавания у капсульной 53 процента, а у сверточной – 51.

Таким образом, капсульная нейронная сеть хоть и имеет немного большую точность на многоклассовой классификации, время, которое требуется на ее обучение, и число параметров, которое нужно обучить, в разы превосходит аналогичные значения для сверточной сети. Это означает, что для работы капсульной нейронной сети требуются большие вычислительные ресурсы и больший размер обучающей выборки, чем для сверточной нейронной сети.

Согласно исследованиям [14] сверточные нейронные сети имеют большую точность распознавания по сравнению с перцептроном, а также большую устойчивость к шумам и скорость обучения за счет возможности распараллеливания алгоритма.

Так как ядро свёртки для каждой карты признаков одно, это позволяет нейронной сети научиться выделять признаки вне зависимости от их расположения во входном изображении, что не возможно в перцептронах.

Согласно исследованиям [15] повысить точность работы перцептрона на задаче классификации можно за счет использования предварительной обработки изображений. Для выделения признаков из изображений к ним могут быть применены различные фильтры.

Недостатком сверточных нейронных сетей является то, что отбрасывается потенциально полезная информация, теряются пространственные связи между объектами или их частями. Помимо этого, проблема заключается в неспособности сети определять положение объекта в пространстве [10].

Капсульные нейронные сети лучше реагируют на мелкие отличия по сравнению со сверточными сетями, так как свертка является загрублением, и снижают ошибку распознавания в другом ракурсе [16].

Критерии сравнения описанных выше подходов к построению нейронных сетей и результаты сравнения представлены в таблице 1.1.

Таблица 1.1 – Сравнение видов нейронных сетей

	Перцептрон	**Сверточные сети**	**Капсульные сети**
Число обучаемых параметров	больше всего	меньше всего	среднее
Устойчивость к шумам	устойчив	устойчивы	не устойчивы
Необходимость предобработки изображений	есть	нет	нет
Скорость обучения	меньше всего	больше всего	средняя
Размер обучающей выборки	больше всего	меньше всего	больше всего
Возможность реконструкции изображения	нет	нет	есть
Устойчивость к сдвигу	нет	есть	есть

1.3.5 Вывод

Перцептрон плохо подходит для задачи распознавания объектов на изображении в силу того, что для его работы требуется предварительная обработка изображений, он не устойчив к шумам и обучается дольше, чем сверточные и капсульные нейронные сети.

Капсульные нейронные сети в отличие от перцептрона не нуждаются в предварительной обработке изображений, устойчивы к сдвигу изображений и для их обучения нужен набор данных меньшего размера. Недостатком таких сетей является их неустойчивость к шумам на изображении.

Таким образом, лучше всего для решения задачи распознавания летательных аппаратов на изображении подходят сверточные нейронные сети, которые устойчивы к сдвигу и шумам на входном снимке и требуют меньший набор данных для обучения по сравнению с капсульными сетями.

1.4 Проблема переобучения нейронной сети

Проблема переобучения в нейронных сетях заключается в том, что модель запоминает данные только из обучающей выборки, не обобщая свои знания на новые, ранее не встречавшиеся данные. Это происходит из-за того, что модель адаптируется к обучающим примерам, вместо того, чтобы учиться классифицировать новые данные [17]. Признаком переобучения модели является существенно большое значение ошибки распознавания на тренировочной выборке, нежели на тестовой. Зачастую переобучение появляется из-за использования слишком сложных моделей, либо наборов данных, в которых вхождения похожи друг на друга [17].

Недообучение - это противоположная проблема переобучения нейронных сетей. Оно характеризуется тем, что алгоритм обучения не достигает удовлетворительной точности на обучающем множестве. Это может быть связано с тем, что выбрана слишком простая модель или недостаточно обучающих примеров. В результате модель не сможет классифицировать данные в более сложных случаях. [17].

Примеры недообученной, переобученной и оптимально обученной нейронной сети приведены на рисунке 1.3.

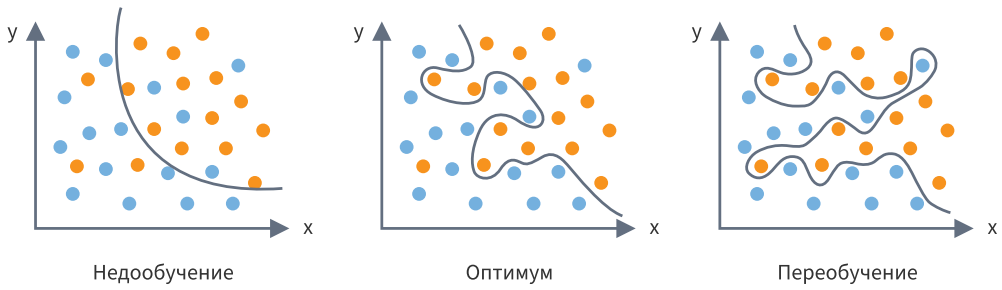

Рисунок 1.3 – Пример переобучения, недообучения и оптимального обучения

В этом примере нейронная сеть используется для разделения входного множества объектов на два класса. В первом случае нейронная сеть является недообученной, так как вероятность ошибки равна 0.22 и может быть еще уменьшена за счет использование более сложной формы зависимости.

На примере по середине нейронная сеть строит общую зависимость для данного набора данных, не подгоняя значения под аномальные элементы для

рыжего и голубого класса слева и справа соответственно.

Пример справа показывает переобученную нейронную сеть, которая строит зависимость, подгоняя параметры под аномальные параметры обучающей выборки.

Для борьбы с переобучением можно использовать следующие способы:

- аугментация обучающей выборки;
- метод раннего останова;
- регуляризация;
- батч нормализация.

1.4.1 Аугментация

Первый способ борьбы с переобучением – аугментация обучающей выборки. Аугментацией называется этап обучения нейронных сетей, состоящий в модификации обучающих изображений (поворот, масштабирование, зеркальное отражение и т. д.) по определенному правилу с целью расширить обучающую выборку и повысить ее разнообразие [18].

Существует три основных вида аугментации:

- геометрическая аугментация – изменение геометрических параметров изображения, таких как поворот, масштабирование, сдвиг и отражение;
- цветовая аугментация – изменение цветовых параметров изображения, таких как яркость, контрастность и насыщенность;
- добавление шума.

Аугментация первого типа обычно улучшает качество работы сверточных нейронных сетей, так как такие сети не инвариантны к масштабу, и изменение масштаба изображения значительно повышает разнообразие данных, позволяя сети обучаться на более разнообразных наборах данных [18]. В статье [19] описывается повышение точности распознавания нейронной сети на 10 процентов за счет использования аугментации масштаба.

Аугментации второго типа предполагают случайное изменение компонент R, G, B цвета пикселей изображения. Это один из самых эффективных

методов аугментации данных, потому что нейросети без этой аугментации имеют тенденцию к заучиванию правил вида «сумма цветов пикселей в области». Также такая аугментация может улучшить способность распознавания сети при различных условиях освещенности [18].

Аугментация добавлением шума на изображение повышает устойчивость модели к шуму на реальных изображениях.

1.4.2 Метод раннего останова

Метод раннего останова после каждой эпохи обучения проверяет точность модели на обучающей и тестовой выборках. Обучение нейронной сети начинается при случайных значениях весов, и с каждой эпохой обучения точность на обучающей выборке будет повышаться, а на тестовой точность сначала будет расти, в момент, когда сеть будет достаточно обучена, зафиксируется на некотором значении, а потом начнет падать из-за переобучения модели.

Суть метода раннего останова заключается в отслеживании точности модели на тестовой выборке и остановке обучения в момент, когда она начинает расти.

К преимуществам данного метода можно отнести:

- сокращение времени обучения, так как нейронная сеть не будет обучаться, когда в этом уже нет необходимости;
- отсутствие дополнительных затрат на дополнение обучающей выборки.

1.4.3 Регуляризация

Метод регуляризации заключается в ограничение значений весовых коэффициентов нейронной сети, что делает их распределение более равномерным. Это достигается за счет добавления некоторого штрафа за увеличение весов нейронной сети в функцию потерь.

Существует три основных вида регуляризации [20]:

- L1 регуляризация, которая также называется Лассо регуляризацией, она добавляет штраф от суммы абсолютных значений весов модели;
- L2 регуляризация, которая также называется регуляризацией Тихонова, она добавляет штраф от суммы квадратов весов модели;

- дропаут, который случайным образом удаляет связи между нейронами.

В пером виде регуляризации новая функция потерь описывается выражением 1.13

$$L_{\text{new}} = L_{\text{old}} + \lambda \sum_{i=1}^{N} |w_{\text{i}}|, \tag{1.13}$$

где L_{new} – новое значение функции потерь, полученное после регуляризации, L_{old} – значение функции потерь до проведения регуляризации, λ – коэффициент штрафования весов, N – число весов в модели, w_{i} – значение i-го веса модели.

При коэффициенте λ равном нулю никакой регуляризации не будет и модель переобучится. При повышении коэффициента штрафования модель будет приближаться к оптимальной, то есть значение ошибки на тестовой выборке будет падать, но чем больше значение этого коэффициента, тем ближе значения всех весов будут к нулю, тем дальше модель отклоняется от локального минимума функции потерь до регуляризации и тем больше растет ошибка модели не тренировочной выборке. Таким образом, значение коэффициента λ должно подбираться экспериментально во время обучения. Чем меньше ошибка на тренировочной выборке, тем лучше подобран коэффициент штрафования.

В регуляризации Тихонова штраф считается не по сумме абсолютных значений, а по сумме квадратов и выражается зависимостью 1.14

$$L_{\text{new}} = L_{\text{old}} + \lambda \sum_{i=1}^{N} w_{\text{i}}^2, \tag{1.14}$$

где все параметры аналогичны параметрам в выражении 1.13.

Главное различие между двумя методам заключается в том, что регуляризация Лассо уменьшает коэффициент менее важной характеристики до нуля, полностью удаляя ее из рассмотрения, а регуляризация Тихонова уменьшает веса, но не делает их равными нулю [20].

Еще одним видом регуляризации является дропаут. Суть метода заключается в том, что на каждой итерации обучения нейронной сети все связи между нейронами удаляются с некоторой вероятностью p. Иными словами это означает, что на каждой итерации обучения модели значение каждого веса w_{i} нейронной сети может быть на одну итерацию приравнено к нулю с

некоторой вероятностью p.

Таким образом, регуляризация борется с проблемой переобучения нейронной сети и повышает ее обобщающую способность [20]. К недостатками регуляризации можно отнести то, что:

- добавление штрафа или удаление некоторых связей может привести к ухудшению точности модели на обучающих данных;
- нельзя заранее оптимальным образом определить коэффициент штрафования весов λ и вероятность удаления связи между нейронами p.

1.4.4 Нормализация

Обычно при обучении нейронной сети шаг градиентного спуска делается не по одному конкретному примеру, а сразу по некоторому набору обучающих примеров. Такой подход имеет следующие преимущества [20]:

- усреднение градиента по нескольким примерам представляет собой апроксимацию градиента по всему тренировочному множеству, и чем больше примеров используется в одном мини-батче, тем точнее это приближение, использование всего обучающего множества невозможно в силу ограничений вычислительных ресурсов;
- в глубоких нейронный сетях к каждому примеру в отдельности требуется применить большое число последовательных операций, в случае использования некоторого набора обучающей примеров, можно выполнять эти последовательные операции в параллельном режиме для каждого примера в отдельности.

При таком подходе к обучению и использовании глубоких нейронных сетей возникает проблема, связанная с тем, что изменение распределения активаций выходов первых слоев на очередном шаге градиентного спуска приводит к сдвигу распределения данных во всех последующих слоях, что затрудняет их обучение и может ухудшить результаты. Для борьбы с этой проблемой используется батч-нормализация, которая позволяет нормализовать выходы каждого слоя в процессе обучения. Это делает распределение данных более стабильным и уменьшает влияние сдвига распределения на

последующие слои. Такая проблема получила название внутреннего сдвига переменных.

В исследованиях, приведенных в статье [21], говорится, что процесс обучения сходится быстрее, когда входы нейронной сети нормализованы, то есть их математическое ожидание приведено к нулю, а матрица ковариаций – к единичной. Если применять нормализацию к входам каждого слоя, то удастся избежать проблемы внутреннего сдвига переменных.

Для выполнения нормализации требуется предварительно рассчитать математическое ожидание и дисперсию элементов батча, которые определяются выражениями 1.15 и 1.16 соответственно.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_{\text{i}}, \tag{1.15}$$

где μ – математическое ожидание элементов бача, N – размер бача, x_{i} – i-ый элемент бача.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{\text{i}} - \mu)^2, \tag{1.16}$$

где σ – дисперсия элементов бача, N – размер бача, x_{i} – i-ый элемент бача, mu – значение математического ожидания, посчитанное по формуле 1.15.

Тогда нормализацию входов можно проводить, используя выражение 1.17

$$\hat{x_{\text{i}}} = \frac{x_{\text{i}} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{1.17}$$

где $\hat{x_{\text{i}}}$ – нормализованное значение i-го входа, x_{i} – ненормализованное значение i-го входа, μ и σ – математическое ожидание и дисперсия, посчитанные по формулам 1.15 и 1.16 соответственно, ϵ – некоторая константа, которая нужна для предотвращения деления на ноль.

Такая нормализация имеет существенный недостаток: в случае, если в качестве функции активации слоя используется сигмоидальная функция, например логистическая 1.4, то после нормализации нелинейность, которую давала эта функция активации, пропадет, так как большинство значений будут попадать в область, где эта функция ведет себя линейно, и функция активации фактически станет линейной [22].

Для того, чтобы компенсировать этот недостаток, слой нормализации

должен быть способен в некоторых случаях практически никак не менять входные значения. Достигается это при помощи введения двух новый коэффициентов: коэффициент масштарбирования и сдвига нормализации. Итоговое выражение для слоя нормализации определяется зависимостью 1.18

$$y_{\mathrm{i}} = \gamma_{\mathrm{i}}\hat{x_{\mathrm{i}}} + \beta_{\mathrm{i}}, \tag{1.18}$$

где y_{i} – i-ый выход слоя нормализации, $\hat{x_{\mathrm{i}}}$ – величина, полученная из выражения 1.17, γ_{i} и β_{i} – коэффициенты масштабирования и сдвига, которые настраиваются во время обучения модели.

Значения математического ожидания и дисперсии во время обучения от батча к батчу будут изменяться, но на этапе тестирования модели все изменяемые параметры должны быть зафиксированы. Для того, чтобы определить значения математического ожидания и дисперсии на этапе тестирования, эти величины накапливаются во время обучения с использованием экспоненциального скользящего среднего, которое определяется зависимостью 1.19

$$EMA_{\mathrm{t}} = \alpha * x_{\mathrm{t}} + (1 - \alpha) * EMA_{\mathrm{t}-1}, \tag{1.19}$$

где EMA_{t} – значение экспоненциального скользящего среднего в точке t, $EMA_{\mathrm{t}-1}$ – значение экспоненциального скользящего среднего в точке t минус 1, причем значение экспоненциального скользящего среднего в нуле EMA_{t} равно x_0, x_{t} – значение исходной функции, в нашем случае это математическое ожидание или дисперсии, в момент времени t, α – коэффициент характеризующий скорость уменьшения весов, принимает значение от 0 и до 1, чем меньше его значение тем больше влияние предыдущих значений на текущую величину среднего.

В случае, когда входной батч описывается кортежем (N, C, H, W), где N – число элементов в батче, C – число каналов в каждом элементе, H и W – высота и ширина каждого изображения, нормализация считается по всем пикселям, всех изображений по каждому из каналов.

1.4.5 Вывод

Переобучение нейронных сетей – это проблема, когда сеть начинает запоминать обучающие данные вместо того, чтобы обобщать их. Это происхо-

дит, когда сеть становится слишком сложной и способна запомнить каждый пример из обучающего набора данных. В результате, когда сеть используется для предсказания новых данных, она может давать неверные ответы, потому что она не умеет обобщать данные и применять их к новым ситуациям.

Для решения этой проблемы можно использовать различные методы, такие как регуляризация, которая вводит дополнительные ограничения на большие веса модели, нормализация, которая приводит значения входных признаков к одному диапазону, аугментация, которая позволяет увеличить обучающий набор данных, а также метод раннего останова, который отслеживает момент, когда модель начинает переобучаться, и прекращает обучение в этот момент.

1.5 Ансамблевые методы

Ансамблевые методы классификации основаны на том, что несколько классификаторов обучаются на одном и том же наборе обучающих данных, а затем их прогнозы объединяются для классификации элементов тестового набора данных. Математическим обоснованием этой идеи служит теорема Кондорсье о жюри присяжных [23].

Классификатор называется слабым, если его ошибка на обучающей выборке менее 50 процентов, но больше нуля. Тогда, объединив предсказания нескольких таких классификаторов, можно достичь более точности классификации на элементах тестовой выборки [23].

Выделяют 4 основных метода ансамблевой классификации [23]:

- бэггинг;
- бустинг;
- стекинг.

Идея бэггинга состоит в том, что если размер обучающей выборки не велик, то можно создать много случайных выборок из исходной путем отбора некоторых элементов, и обучить слабые классификаторы на эти подвыборки. Таким образом, каждая модель имеет свой набор обучающих примеров и старается сделать предсказания на основе своего подмножества данных. Затем результаты всех моделей комбинируются для получения итоговых предсказаний.

Бустинг – это процедура последовательного построения композиции алгоритмов машинного обучения, когда каждый следующий алгоритм стремится компенсировать недостатки композиции всех предыдущих алгоритмов [23]. Таким образом, при бустинге каждая последующая модель обучается на ошибках предыдущей и старается их компенсировать, повышая точность классификации общей модели.

Идея стекинга заключается в введении некоторого алгоритма классификации и его обучении. При стекинге, в отличие от бустинга и бэггинга, классификаторы должны быть разной природы [23]. Обучение модели при стекинге можно свести к следующим трем шагам:

- обучающая выборка разбивается на две непересекающихся подвыборки;
- первая подвыборка используется для обучения классификаторов;
- вторая для обучения алгоритма, который на вход принимает выходы со всех классификаторов.

Главным недостатком стекинга является деление обучающей выборки на две части.

1.6 Существующие архитектуры

1.6.1 LeNet

LeNet является одной из первых архитектур сверточных нейронных сетей. Ее первое описание приведено в статье [24]. Модель состоит из 5 сверточных слоев, за которыми следует 2 полносвязных слоя. Общий вид модели представлен на рисунке 1.4.

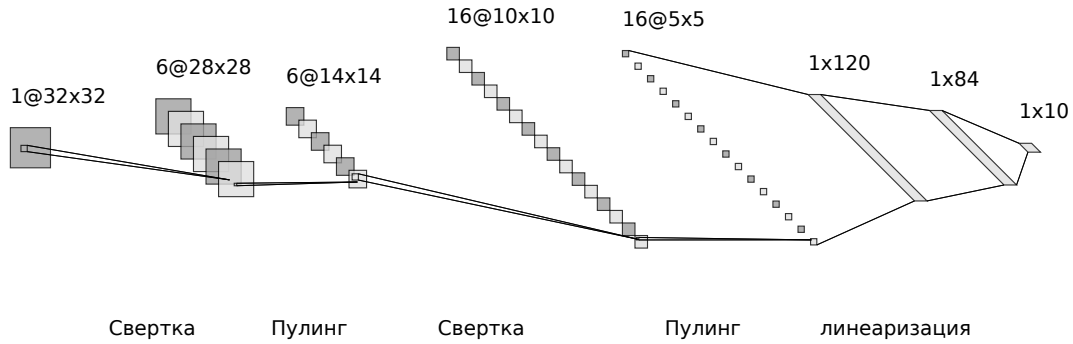

Рисунок 1.4 – Архитектура LeNet

На вход поступает одноканальное изображение размером 32 на 32 пикселей. Далее к нему применяется свертка с 6 фильтрами с ядром 5 на 5 пикселей, которая выделяет основные элементы на изображении. После свертки применяется пулинговый слой, который уменьшает размер изображения в два раза и усредняет значения пикселей, что позволяет уменьшить количество параметров.

Далее применяется свертка с 16 фильтрами и ядром 5 на 5, которая используется для выделения более сложных признаков. После к выходам этой свертки применяется пулинговый слой, аналогичный предыдущему.

После все матрицы из всех каналов пулинговго слоя переводятся в 1 вектор и поступают на вход полносвязного слоя.

Недостатком такой архитектуры является проблема затухания градиента при обучении нейронной сети [25]. Для решения этой проблемы можно использовать Max Pooling между сверточными слоями.

1.6.2 AlexNet

AlexNet по своему принципу не сильно отличается от LeNet. В AlexNet используется больше сверточных слоев, а в слоях субдисретизации используется Max Pooling. В качестве функции активации полносвязного слоя используется ReLU. Общая схема архитектуры сети приведена на рисунке 1.5.

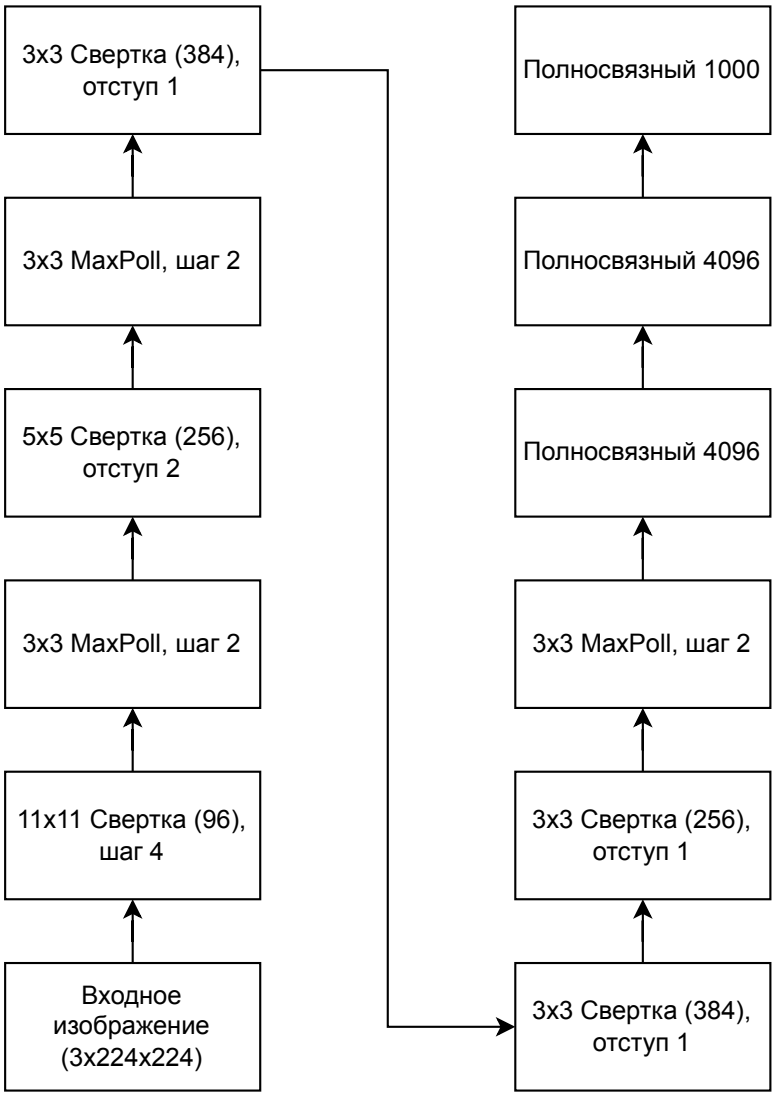

Рисунок 1.5 – Архитектура AlexNet

Преимуществом данной архитектуры является высокая точность распознавания – в 2012 AlexNet показала рекордный результат в точности распознания 1000 различных объектов в соревновании ImageNet.

Главным недостатком является большое число параметров, использующихся при обучении, – около 60 миллионов [25]. Такое количество параметров требует значительно большие вычислительные мощности и память в сравнении с LeNet. Также потребуется больше элементов в обучающей выборке, чтобы корректно настроить эти параметры.

1.6.3 GoogLeNet

В GoogLeNet было введено понятие Inception блока, формальное представление которого приведено на рисунке 1.6.

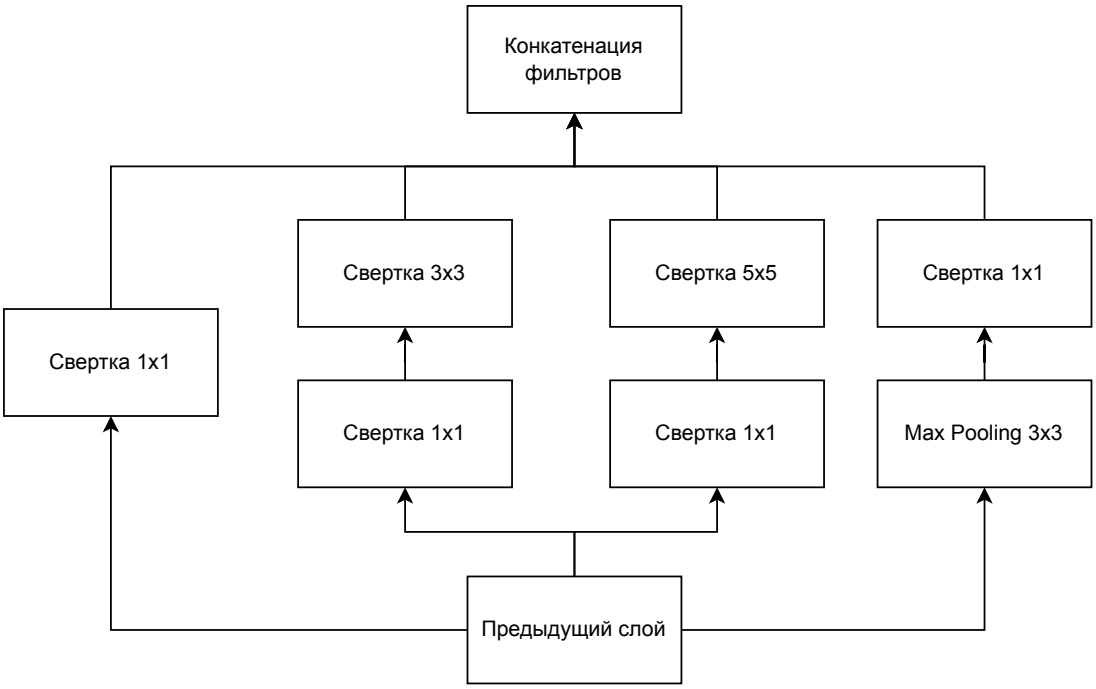

Рисунок 1.6 – Формальное представление Inception блока

В таком блоке выходы с предыдущего слоя параллельно обрабатываются сверткой 1 на 1, блоком из сверток 1 на 1 и 3 на 3, блоком из сверток 1 на 1 и 5 на 5, блоком из Max Pooling размером 3 на 3 и свертки 1 на 1. Далее выходы со всех этих блоков объединяются и передаются дальше.

Фильтр 1 на 1 используется для уменьшения количества параметров в сверточных слоях и ускорения вычислений. Он позволяет сократить количество каналов входного изображения до более низкого значения, что уменьшает количество вычислений при применении более крупных фильтров. Кроме того, фильтры 1 на 1 могут использоваться для комбинирования признаков разных каналов, что улучшает качество классификации [26].

Такой подход имеет два основных преимущества [26]:

- возможность увеличения количества блоков на каждом этапе без неконтролируемого роста вычислительной сложности;
- визуальная информация обрабатывается в различных масштабах, а затем агрегируется, чтобы на следующем этапе можно было абстрагировать признаки из разных масштабов одновременно.

Использование таких блоков в архитектуре GoogLeNet позволяет сократить число параметров примерно в 12 раз по сравнению с AlexNet, при этом точность классификации сети не падает [26].

Для борьбы с затуханием градиента в GoogLeNet используется следующий прием: помимо классификатора в конце нейронной сети добавляется еще один после третьего Inception блока и еще один после 6. Во время обучения функция потерь считается не только по значениям из последнего классификатора, но и по 2 добавленным, домжноженным на некоторый коэффициент. Итоговое выражение для подсчета функции потерь определяется следующей зависимостью 1.20

$$TL = l + k * (l_1 + l_2), \tag{1.20}$$

где TL – общее значение функции потерь, которое нужное уменьшать во время обучения, l – значение функции потерь, посчитанное по выходам из последнего классификатора, k – некоторый коэффициент значимости, на который домножаются значения функции потерь, посчитанные по выходам двух других классификаторов, в GoogLeNet этот коэффициент равен 0.3, l_1 и l_2 – это значения функции потерь посчитанные по выходам классификаторов, добавленный в начале и в середине соответственно.

1.6.4 CapsNet

CapsNet является капсульной нейронной сетью. В статье [27] описывается работа этой сети, которую можно разделить на 4 основных шага

- слой свертки;
- слой primary caps;
- направления по соглашению;
- слой digit caps.

Общий вид архитектуры нейронной сети CapsNet представлен на рисунке 1.7.

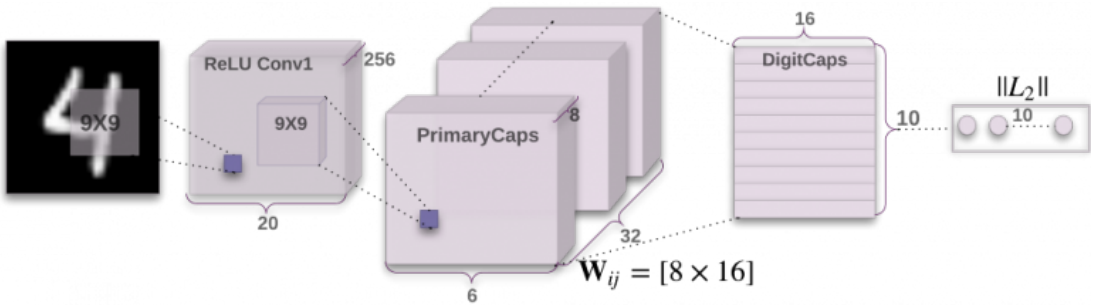

Рисунок 1.7 – Архитектура CapsNet

На вход нейронной сети поступает изображение размером 28 на 28 пикселей. В данном примере изображение имеет только 1 канал, но в реальности может быть и 3 канала для цветного изображения, и 4, если используется дополнительный альфа канал, отвечающий за прозрачность. Число каналов на входе влияет только на размерность ядра свертки на первом шаге.

На первом шаге к входному изображению применяется обычный сверточный слой с ядром 9 на 9 и 256 фильтрами. В качестве функции активации слоя используется ReLU. На выходе из слоя получается 256 каналов размером 20 на 20 пикселей. За счет этого слоя из изображения выделяются базовые признаки, такие как грани объектов.

Начало работы слоя primary caps похоже на обычный сверточный слой, только ядро свертки применяется не к каждому каналу по отдельности, а ко всем сразу, то есть размерность ядра свертки на этом слое 9x9x256. Шаг ядра свертки на этом слое равен двум. Число фильтров также равно 256, поэтому на выходе из сверточного этапа слоя primary caps получается 256 каналов размером 6 на 6 пикселей. На этом этапе выделяются более сложные формы из уже найденных граней.

Далее выходы с предыдущего этапа разбиваются на 32 части по 8 элементов в каждой. Каждый из таких кусков имеет 8 каналов размером 6 на 6 пикселей и называется капсульным слоем. Каждый капсульный слой состоит из 36 капсул, состоящих из 8 элементов. Далее к каждой капсуле применяется функция активации, описывающаяся зависимостью (1.10), которая изменяет длину входного вектора и делает ее значение в диапазоне от 0 до 1, при этом угол поворота вектора не изменяется.

На шаге направления по соглашению определяется, какие капсулы несут полезную нагрузку и должны быть переданы на следующий слой, а какие должны быть отброшены. Каждая капсула, основываясь на самой себе, пытается предсказать активацию следующего слоя, далее из всех предсказаний капсул выбирается самое частое, и капсулы, которые сделали это предсказание, передаются дальше.

К примеру, нейронная сеть должна определять 1 из 10 различных классов. Предсказание капсулы для каждого из классов создается путем перемножения соответствующего вектора на матрицу весов для каждого класса. Общее число предсказаний равно количеству капсул, умноженному на число

классов. В архитектуре CaspNet это значение равно 11520.

Далее определяется, какие из предсказаний лучше всего соотносятся друг с другом и именно они будут переданы на следующий слой. Так как каждое из предсказаний является вектором, определить лучше всего соотносящиеся можно при помощи следующего алгоритма: сначала считается средний вектор по всем предсказанием при этом каждое предсказание имеет одинаковый вес, дальше считается среднее по всем векторам, но важность каждого предсказания тем меньше, чем больше его расстояние от среднего. Эта операция повторяется еще несколько раз, и наиболее согласованными предсказаниями являются те, которые ближе всего находятся к полученному среднему вектору. Предсказания с наибольшей согласованностью отправляются в следующий слой.

На вход слоя digit caps поступает по одному предсказанию на каждый класс, который должна определять нейронная сеть. Длинна вектора каждого из предсказаний определяет вероятность того, что на вход было подано изображение с соответствующим классом.

Способ тренировки такой сети отличается от обычной сверточной. В сверточных сетях обучение нацелено лишь только на правильное определение класса объекта. В капсульных сетях помимо определения класса сеть обучается на правильную реконструкцию входного изображения по выходному вектору.

1.6.5 SimpLeNet

Архитектура нейронной сети SimpLeNet состоит из сверточных слоев только размером 3 на 3 пикселя. Для уменьшения размеров изображения используются слои пуллинга с ядром размера 2 на 2 пикселя и шагом два пикселя. После каждого из сверточных слоев применяется слой нормализации и функция активации ReLU.

Благодаря использованию сверток только размером 3 на 3, в такой нейронной сети меньше обучаемых параметров по сравнению с сетями, где используются свертки больше размеров. Согласно исследованиям [28] в такой сети используется в два с половиной раза меньше вычислительных операций по сравнению с GoogLeNet и в 15 раз меньше обучаемых параметров по сравнению с AlexNet.

Использование меньшего числа параметров приводит к уменьшению

скорости обучения нейронной сети, а также к меньшим ограничениям на вычислительные ресурсы при использовании модели, что важно, так как все изображения, которые поступают на вход модели, сделаны с беспилотных летательных аппаратов и могут на них же обрабатываться.

1.6.6 Yolo

Нейронная сеть Yolo предназначена для детектирования объектов на изображении. Результатом работы такой сети являются граничные области найденных объектов. В отличие от других сетей Yolo способна детектировать сразу несколько объектов на изображении.

Алгоритм работы yolo состоит из двух частей: сверточной нейронной сети и последующей обработки ее результатов. Сверточная нейронная сеть должна выделить ограничивающие рамки объектов на изображении. Задача последующей обработки заключается в том, чтобы провести пороговую фильтрацию полученных из сверточной сети ограничивающий рамок на основе вероятности нахождения в них объекта, а также исключить из рассмотрения схожие рамки.

Сверточная нейронная сеть разбивает входное изображения на ячейки некоторого размера и для каждой из них предсказывает следующий набор параметров:

- p – вероятность нахождения центра какого-нибудь объекта в этой ячейке, принимает значения в диапазоне от нуля до одного;
- x – центр объекта по оси абсцисс внутри этой ячейки, левый верхний угол имеет координаты 0 и 0, правый нижний – 1 и 1, соответственно значение x находится в диапазоне от нуля до одного;
- y – центр объекта по оси ординат;
- w – ширина ограничительной рамки, поделенная на ширину ячейки, принимает значения больше нуля;
- h – высота ограничительной рамки, поделенная на высоту ячейки, принимает значения больше нуля;
- c_i – вероятность того, что в рамке находится объект i-го класса.

Таким образом, выход из сверточной нейронной сети имеет размерность CxCx$(5 + \text{classes})$, где C – число размер сетки по высоте и ширине, $classes$ – число классов, которые распознает нейронная сеть.

Координаты центра по оси абсцисс и ординат ограничительной рамки получаются из формул 1.21 и 1.22 соответственно

$$x = \sigma(t_{\text{x}}) + c_{\text{x}}, \tag{1.21}$$

$$y = \sigma(t_{\text{y}}) + c_{\text{y}}, \tag{1.22}$$

где x и y – координаты центра рамка, c_{x} и c_{y} – координаты левого верхнего угла ячейки, в которой находится центр рамки, t_{x} и t_{y} – выходы из нейронной сети.

Высота и ширина ограничительной рамки определяются по формулам 1.23 и 1.24 соответственно

$$h = p_{\text{h}} e^{t_{\text{h}}}, \tag{1.23}$$

$$w = p_{\text{w}} e^{t_{\text{w}}}, \tag{1.24}$$

где t_{h} и t_{w} – выходы из нейронной сети, p_{h} и p_{w} – высота и ширина якоря, который используются для определения, какие ограничивающие рамки должны быть выданы алгоритмом для конкретных объектов на изображении.

В нейронной сети Yolo v3 входное изображение разделяется на сетка трех разных масштабов и для каждой ячейки используется по три якоря.

Для удаления схожих рамок на этапе обработки выходов сверточной сети используется алгоритм non maximum supperssion, суть работы которого заключается в следующем:

- ограничивающие рамки сортируются по вероятности нахождения в них объекта;
- выбирается рамка с самой высокой вероятностью и она сохраняется в окончательном списке ограничивающих рамок;
- после этого отбрасываются все рамки, которые нашли объект того же класса и имеют значение перекрытия с выбранной на предыдущем шаге рамкой больше порогового;
- процесс продолжается, пока не будут обработаны все оставшиеся рамки.

Перекрытие двух рамок высчитывается делением площади их пересечения на площадь их объединения.

1.6.7 Вывод

Архитектура CapsNet плохо подходит для решения задачи распознавания летательных аппаратов с аэрофотоснимков, так как не устойчива к шумам во входном изображении.

Архитектура LeNet менее глубокая по сравнению с другими архитектурами, что ограничивает ее способность к достижению большей точности классификации.

Нейронные сети AlexNet и GoogLeNet плохо подходят для решения поставленной задачи в силу большего числа обучающих параметров по сравнению с другими архитектурами, так как большее число параметров приводит к большим ограничениям на вычислительные ресурсы, увеличению времени обучения и требуемого размера обучающей выборки.

Таким образом, для решения задачи классификации лучше всего подходит архитектура SimpLeNet за счет использования меньшего числа параметров по сравнению с другими архитектурами, а также за счет использования слоев нормализации, которые ускоряют процесс обучения и повышают точность модели, а для решения задачи детектирования – Yolo.

1.7 Формализованная постановка задачи

Цель работы – разработать метод распознавания летательных аппаратов с аэрофотоснимков.

Для достижения поставленной цели требуется выполнить следующие задачи:

- провести анализ существующих программных подходов распознавания летательных аппаратов с аэрофотоснимков (проведено выше);
- разработать метод распознавания летальной техники на аэрофотоснимках;
- реализовать спроектированный метод;
- провести исследование точности распознавания модели на тестовой

выборке при различных подходах к обучению.

На вход методу подается изображение со снимком аэропорта, где находится летательные аппараты одного из 20 классов. Результатом работы метода является распознанные на изображении самолеты и их модели. Требуемая точность работы метода на тестовом наборе данных должна составлять не менее 75 процентов.

На входное изображение накладываются следующие ограничения:

- изображение сделано в дневное время суток;
- размер изображения 800 на 800 пикселей;
- размер самолетов на изображении более 70 пикселей, но менее 96.

На рисунке 1.8 приведена IDEF-0 диаграмма уровня A0 метода распознавания летательных аппаратов с аэрофотоснимков с использованием нейронных сетей.

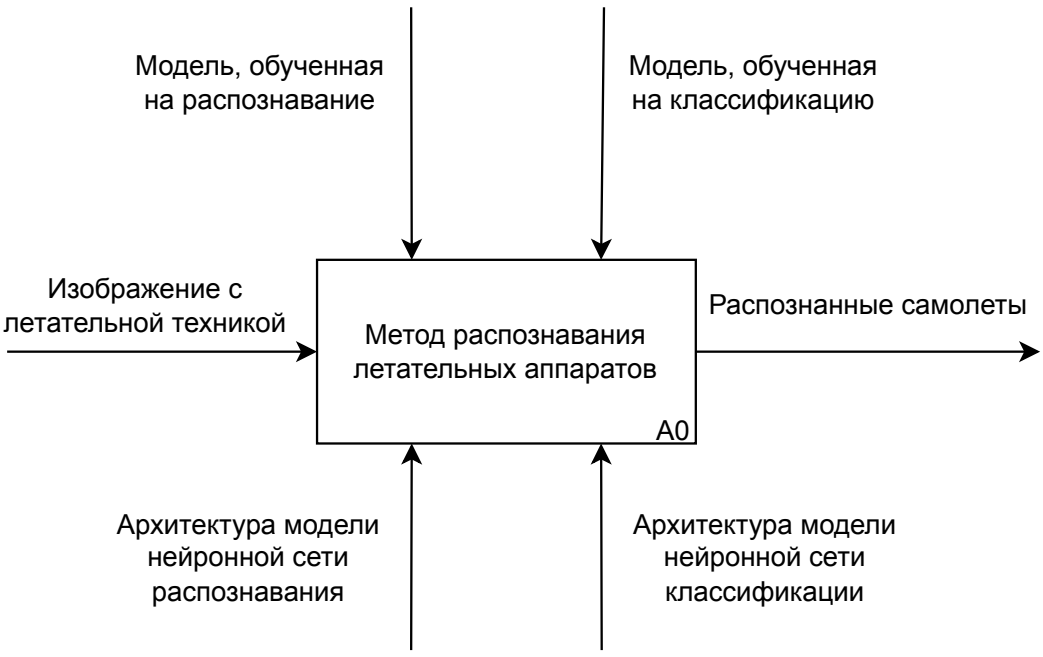

Рисунок 1.8 – IDEF-0 диаграмма метод распознавания летательных аппаратов с аэрофотоснимков

1.8 Вывод

Методы детерминированного подхода плохо подходят для задачи классификации изображений с аэрофотоснимков в силу того, что не устойчивы к шуму.

Дерево решений не применимо для поставленной задачи, так как на этапе построения дерева надо выделять критерии, по которым нужно классифицировать входную информацию.

Среди нейрокомпьютерных алгоритмов выделяют алгоритмы обучения с учителем и без. Задача распознавания объектов на изображении относится к алгоритмам обучения с учителем и может быть решена с использованием нейронных сетей.

Среди основный принципов построения нейронных сетей: перцептрон, сверточные сети и капсульные сети. Большую точность классификации показывают сверточные и капсульные сети. Сверточные нейронные сети обучаются быстрее и требуют меньше обучаемых данных, чем капсульные, а также устойчивы к шумам.

Таким образом, для решения задачи распознавания летательных аппаратов с аэрофотоснимков, лучше всего подходят сверточные нейронные сети за счет большей точности распознавания в сравнении с перцептроном, а также за счет устойчивости к шуму и большей скорости обучении по сравнению с капсульными сетями. При обучении сверточной нейронной сети нужно использовать регуляризацию и нормализацию для борьбы с проблемой переобучения, а также выполнять аугментацию обучающей выборки для повышения обобщающей способности модели. В случае слабых классификаторов, нужно использовать методы ансамблевой обработки: бустинг, бэггинг и стэкинг.

Среди сверточных нейронных сетей для решения задачи классификации лучше всего подходит SimpLeNet за счет меньших требований к вычислительным ресурсам, большей скорости обучения и меньшим объемом требуемых обучающих данных по сравнению с другими сетями, а для решения задачи детектирования объектов – Yolo.

2 Конструкторский раздел

2.1 Требования к предъявляемые к ПО

На вход метода поступает изображение, на которое накладываются следующие ограничения:

- изображение в формате PNG или JPG;
- размер изображения 800 на 800 пикселей;
- размер самолетов на изображении больше 70 пикселей, но меньше 96;
- изображение сделано под углом 90 градусов к поверхности Земли.

Результатом работы метода являются распознанные самолеты, а именно их количество, расположение на входном изображении и их модели.

Программно обеспечение должно предоставлять интерфейс для разработанного метода со следующими функциями:

- возможность выбора и использования одной из заранее обученных моделей;
- возможность выбора и настройки метода обучения модели;
- возможность загрузки аэрофотоснимка с летательной техникой и получение информации о распознанных на нем летательных объектах, а именно их количество, местоположение и модель;
- возможность загрузки изображения с одним самолетом и получение результатов о классе летательной техники на нем.

2.2 Проектирование метода

IDEF-0 диаграмма метода распознавания летательной техники с аэрофотоснимков уровня A1 приведена на рисунке 2.1.

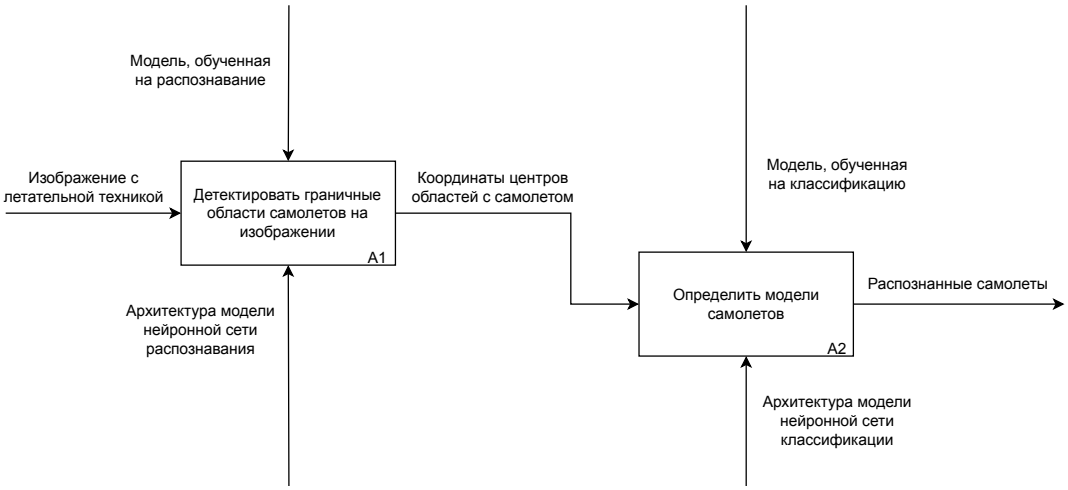

Рисунок 2.1 – IDEF-0 диаграмма уровня A1

Решение задачи распознавания самолетов на аэрофотоснимках было поделено на две подзадачи: детектирование объектов на изображении и их классификация.

В качестве модели классификации была выбрана сверточная нейронная сеть из 12 сверточных слоев, выходы которой соединены с двухслойным перцептроном.

Обучаемая модель имеет следующую структуру:

- слой свертки с 64 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;
- три аналогичных слоя с 128 фильтрами;
- слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 48 на 48;
- два слоя с 128 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;
- аналогичный слой с 256 фильтрами;
- слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 24 на 24;
- слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

- слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 12 на 12;
- слой свертки с 512 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;
- слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 6 на 6;
- слой свертки с 2048 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;
- слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;
- слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 3 на 3;
- слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом один пиксель;
- входной слой перцептрона, состоящий из 256 нейронов;
- выходной слой перцептрона, состоящий из 20 нейронов.

После каждого сверточного слоя применяется слой нормализации, описанный в главе 1.4.4 и функция активации ReLU. Такая архитектура нейронной сети имеет следующие преимущества:

- использование сверточных слоев позволяет выделять различные признаки, такие как границы, формы, текстуры, вне зависимости от их расположения в входном изображении;
- использование пуллинговых слоев позволяет уменьшить размерность изображения с сохранением отличительных признаков;
- несмотря на количество слоев, такая архитектура за счет использования пуллинговых слоев и сверток с ядром 3 на 3 вместо 5 на 5 и больших имеет относительно немного обучаемых параметров: 4910356 в сравнении с шестьюдесятью миллионами в архитектуре AlexNet, которая была описана в главе 1.6.2.

Использование меньшего числа параметров приводит к уменьшению скорости обучения нейронной сети, а также к меньшим ограничениям на вычислительные ресурсы при использовании модели, что важно, так как все изображения, которые поступают на вход модели, сделаны с беспилотных летательных аппаратов и могут на них же обрабатываться.

Для решения задачи детектирования объектов была выбрана нейронная сеть Yolo v3, состоящая из 106 сверточных слоев, среди которых сверточный слои с размером ядра 3 на 3 и 1 на 1 и шагом свертки 1 или 2 пикселя, а также блоки, в которых используется техника пропуска соединений.

Всего такая сеть выдает три матрица предсказаний для размеров ячеек 25, 50 и 100 пикселей.

Схемы алгоритмов обучения нейронных сетей и прохождения одной эпохи приведены на рисунках 2.2 и 2.3 соответственно.

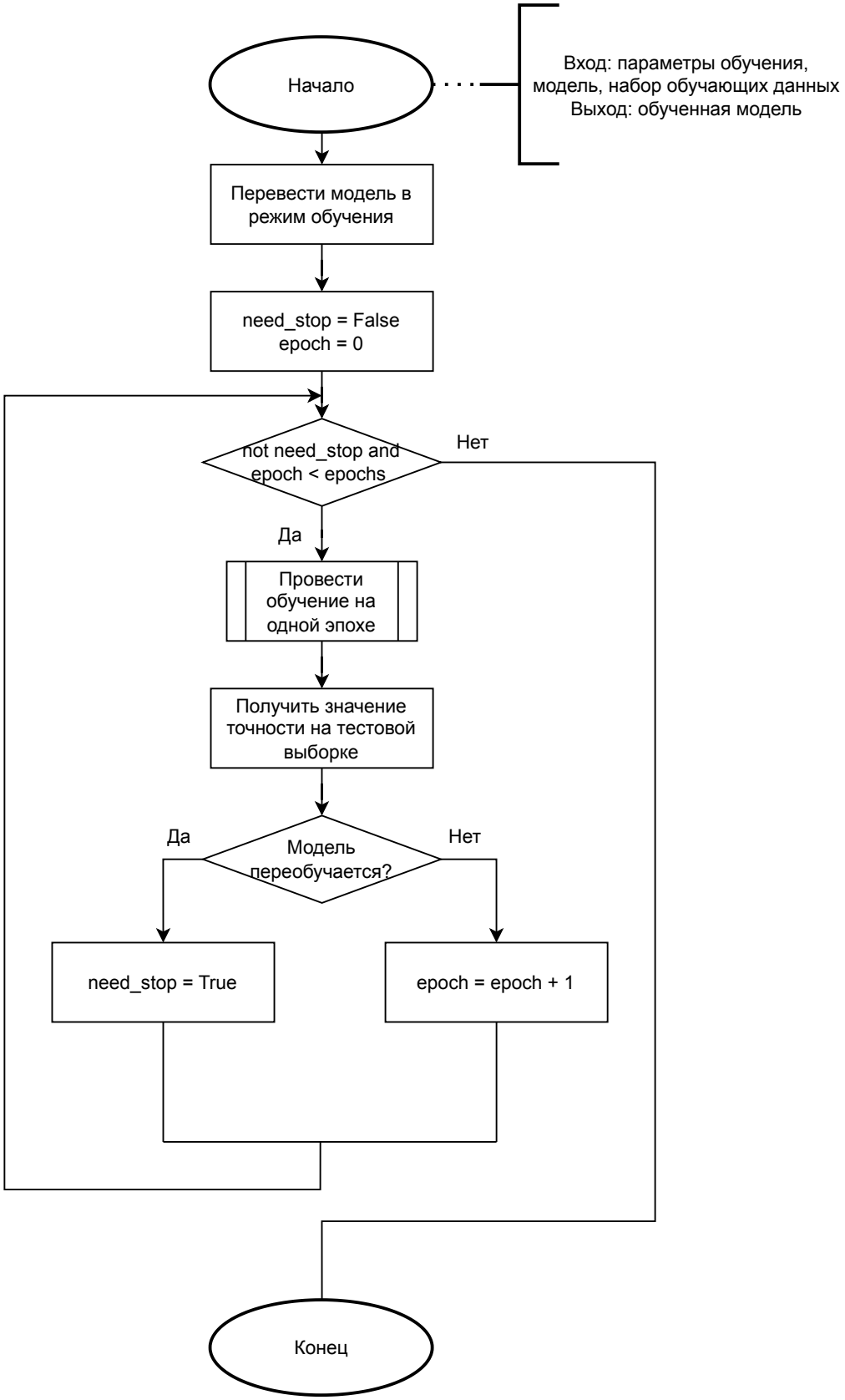

Рисунок 2.2 – Схема алгоритма обучения нейронной сети

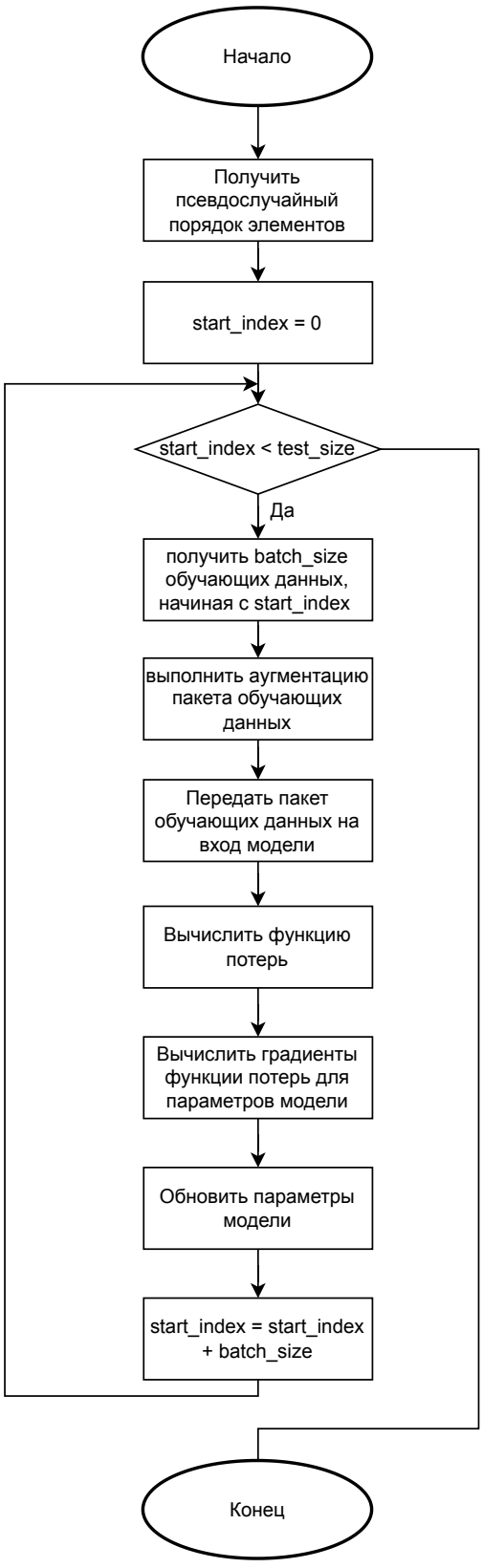

Рисунок 2.3 – Схема алгоритма прохождения одной эпохи

При подсчете точности модели на тестовой выборке важно учитывать ее размер и загружать в оперативную память по частям, каждый раз освобождая выделенные ресурсы.

2.3 Структура программного обеспечения

Программное обеспечение состоит из четырех модулей:

- модуль, реализующий модель нейронной сети распознавания летальной техники;
- модуль, реализующий модель нейронной сети классификации летальной техники;
- модуль, реализующий интерфейс взаимодействия с пользователем;
- модуль пользовательского интерфейса.

Структурная схема взаимодействия модулей разрабатываемого программного обеспечения представлена на рисунке 2.4.

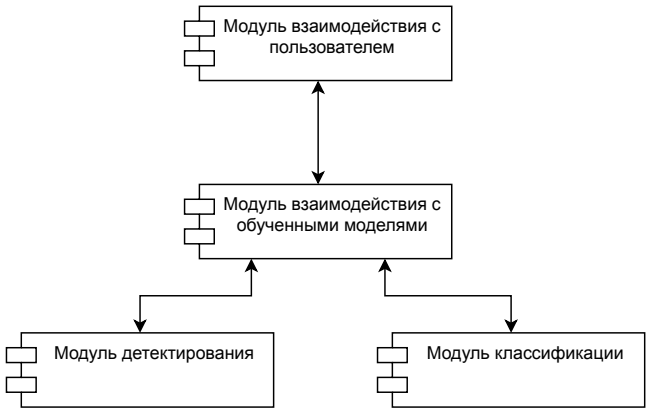

Рисунок 2.4 – Структурная схема взаимодействия модулей

Модули детектирования и классификации используются для обучения моделей и взаимодействия с уже обученными. После завершения обучения параметры модели должны быть сохранены в файл для обеспечения в дальнейшем загрузки модели без обучения.

Модуль взаимодействия с пользователем должен обеспечивать следующие возможности:

- загрузка одной из заранее обученных моделей;
- запуск обучения модели с выбранными параметрами и сохранение его результаты;

- загрузка изображения с летательной техникой и определение ее класса с помощью загруженной или обученной модели;
- загрузка аэрофотоснимка с различными классами летательной техники и определение их классов.

2.4 Набор обучающих данных

Для обучения нейронной сети был выбран набор данных из источника [29], состоящий из 3842 снимков аэропортов, сделанных с беспилотных летательных аппаратов. На этих снимках находится 22341 самолет 20 различных типов.

Пример элемента обучающей выборки приведен на рисунке 2.5.

Рисунок 2.5 – Пример элемента обучающей выборки

Вся выборка была поделена на обучающую и тестовую. Обучающая выборка содержит 20344 самолета, а тестовая – 1997.

2.5 Вывод

В данном разделе были определены ограничения, которые накладываются на входное изображение, и требования, которые предъявляются к разрабатываемому программному обеспечению.

Была детализирована IDEF-0 диаграмма уровня A0, описанная в разделе формализованной постановки задачи, а также было проведено разбиение программного обеспечения на модули и описание функциональных требований, которые они должны обеспечивать. Задача распознавания самолетов была разбита на две подзадачи: детектирование объектов на изображении и их классификация.

Были определены архитектуры нейронных сетей классификации и детектирования, а также преимущества использования именно таких архитектур. Была определена схема алгоритма обучения составленных нейронных сетей.

Была найдена выборка данных, которые будут использоваться для обучения моделей, и было произведено ее деление на две части: обучающую и тестовую.

3 Технологический раздел

3.1 Средства реализации программного обеспечения

В качестве языка программирования был выбран Python. Данный выбор обусловлен тем, что Python имеет множество библиотек, таких как TensorFlow, Keras, PyTorch и Theano, которые предоставляют множество инструментов для создания и обучения нейронных сетей.

В качестве библиотеки для создания нейронной сети был выбрана библиотека PyTorch версии 2.0.0, так как она имеет следующие возможности и инструменты:

- динамический граф вычислений, использование которого облегчает отладку моделей;
- возможность переноса вычислений на GPU;
- набор инструментов для создания различных слоев, из которых складывается архитектура нейронной сети;
- имеет API на языке C++, что позволяет обучить модель с использованием интерпретируемого языка Python, а использовать ее на компилируемом языке C++.

Для работы с большими данными была выбрана библиотека numpy версии 1.21.0, так как она использует оптимизированный код на C, что позволяет выполнять вычисления быстрее, чем с использованием чистого Python, а также потому что классы этой библиотеки интегрируются с библиотекой PyTorch, которая используется для создания нейронной сети.

Для работы с изображениями была выбрана библиотека Pillow версии 8.2.0, так как она позволяет загружать и трансформировать изображения разных форматов, таких как PNG, JPEG, BMP, а также переводить изображения в объекты, которые передаются на вход нейронной сети.

Для создания графического интерфейса использовалась библиотека PyQt версии 5.0, так как она является кроссплатформенной, имеет базовые виджеты, на которых может быть построен интерфейс, а также предоставляет возможность подписки на события, которые генерируются виджетами.

3.2 Реализация программного комплекса

В качестве модели классификации используется сверточная нейронная сеть с 12 слоями свертки и 5 слоями пуллинга, которая соединяется с двухслойным перцептроном. После каждого сверточного слоя за исключением последнего используется слой нормализации и функция активации ReLU. Реализация модели приведена в листинге А.1 (приложение А).

Перед обучением модели выполняется предобработка всего обучающего набора данных, во время которой нужно разбить все входные изображения по классам, привести все изображения к размеру 96 на 96 пикселей, так как только такое изображение обрабатывается моделью, а также поделить весь набор данных на обучающую и тестовую выборки.

После выполнения предобработки нужно произвести аугментацию для расширения обучающей выборки. К каждому изображения в обучающем множестве применяются следующие преобразования: поворот на 30 и 330 градусов, увеличение яркости в полтора раза, а также гауссово размытие, которое создает фильтр Гаусса с заданным размером и силой размытия, и применяют его к изображению, используя операцию свертки. Коэффициенты в фильтре Гаусса вычисляются по формуле 3.1

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{3.1}$$

где x и y – расстояния от центра ядра по горизонтали и вертикали соответственно, σ – сила размытия.

Исходное изображения и изображения после аугментации приведены на рисунках 3.1-3.5.

Рисунок 3.1 – Изображение до аугментации

Рисунок 3.2 – Изображение после поворота на 30 градусов

Рисунок 3.3 – Изображение после поворота на 330 градусов

Рисунок 3.4 – Изображение после увеличения яркости

Рисунок 3.5 – Изображение после применения гауссова размытия

Обучение модели проводилось на машине с процессором Intel Core i9-10900, 64 гигабайтами оперативной памяти и графической картой NVIDIA GeForce RTX 3080 с 16 гигабайтами памяти типа GDDR6. Время прохождения одной эпохи в среднем занимает 2 минуты 41 секунду.

В качестве оптимизатора функции потерь был выбран Adam, так как он автоматически адаптирует скорость обучения для каждого параметра в

зависимости от его градиента, что позволяет более эффективно использовать скорость обучения и ускоряет сходимость.

Обучение модели останавливается, когда точность распознавания на тестовой выборке после прохождения очередной эпохи становится меньше, чем на двух предыдущих, так как это свидетельствует о том, что сеть начала переобучаться.

Код обработки и загрузки обучающего набора данных представлен в листинге А.3. Реализация алгоритмов обучения и прохождения одной эпохи представлены на листингах А.4 и А.5 соответственно. Все листинги находятся в приложении А.

Графики зависимостей точности распознавания на тестовой и обучающей выборках от номера эпохи обучения приведены на рисунке 3.6.

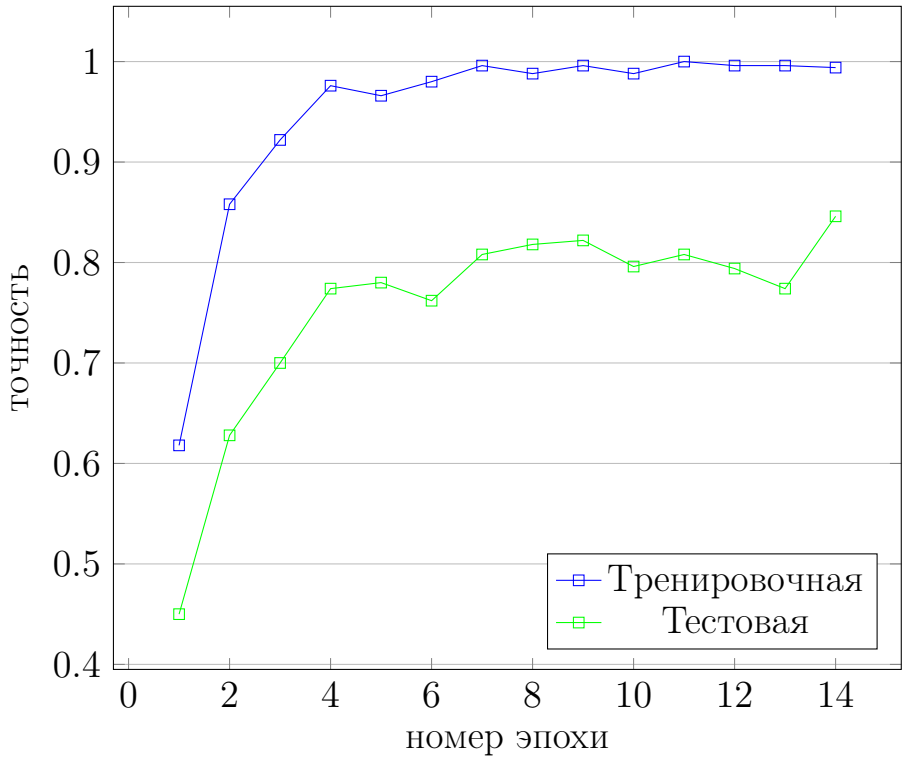

Рисунок 3.6 – Обучение модели

Из графиков видно, что после 7 эпохи модель начинает переобучаться и требуется остановить обучение. Итоговая точность полученной модели на тестовой выборке составила 84 процента.

В качестве модели детектирования была реализована сверточная нейронная сеть, состоящая из 106 слоев. В качестве функции активации была выбрана функция ReLU. Разработанная модель имеет три выходных слоя,

каждый из которых имеет размерность (3xSxSx5), где 3 – число якорей, S – размер ячейки, для 3 выходов размеры ячеей будут 25, 50 и 100 соответственно, вектор из пяти элементов состоит из вероятности нахождения центра объекта внутри ячейки, а также координат его центра, высоты и ширины рамки. Реализация модели приведена в листинге А.2 (приложение А).

3.3 Взаимодействие с разработанным ПО

Взаимодействие с разработанным программным обеспечение осуществляется через графический пользовательский интерфейс. Интерфейс приложения представлен на рисунке 3.7.

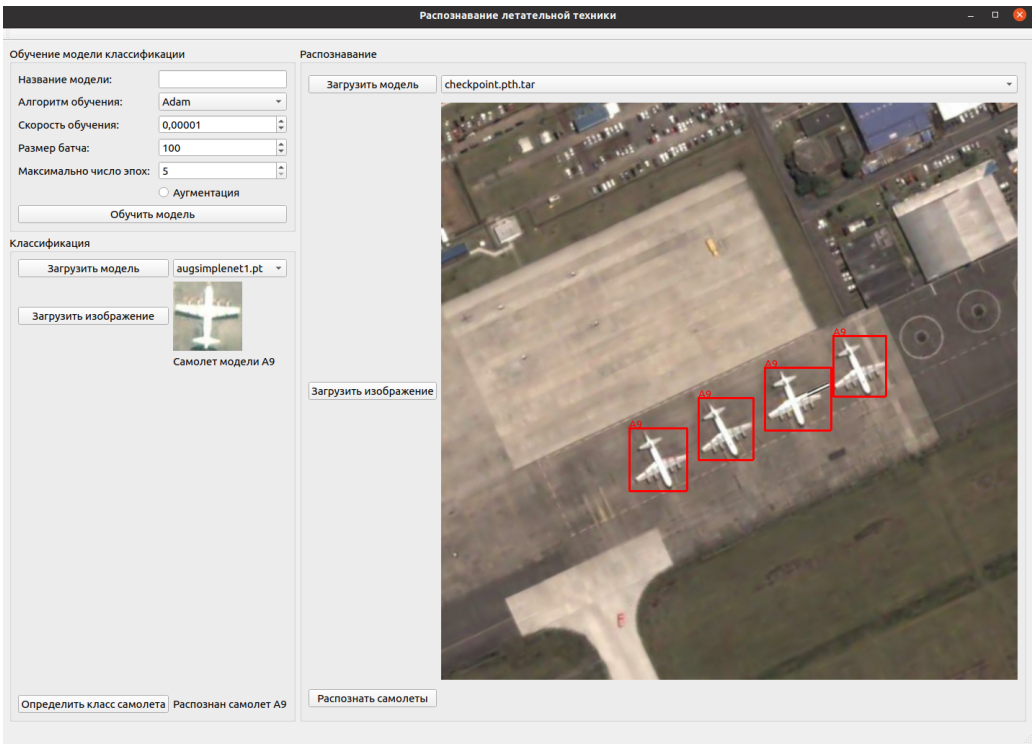

Рисунок 3.7 – Интерфейс приложения

Графический интерфейс поделен на две части. Первая часть позволяет выбрать и загрузить модель обученную на классификацию и проверить ее работу. Вторая позволяет выбрать и загрузить модель, обученную на детектирование, а также проверить ее работу вместе с моделью классификации.

3.4 Вывод

В рамках данного раздела были выбраны и реализованы средства распознавания летательных аппаратов. Для этого были выбраны методы ауг-

ментации обучающих данных, направленные на увеличение количества и разнообразия данных для обучения модели.

Были созданы модели для детектирования и классификации летательных аппаратов и реализован алгоритм их обучения. После этого был проведен ряд тестов, чтобы оценить качество модели. Итоговая точность модели классификации на тестовой выборке составила 84 процента. Точность детектирования на тестовой выборке составила 90 процентов.

Кроме того, был создан графический интерфейс, который позволяет пользователям использовать обученную модель для распознавания типов летательных аппаратов. После этого было проведено ручное тестирование обученных моделей, чтобы убедиться в правильности их работы.

4 Исследовательский раздел

4.1 Сравнение различных методов оптимизации

Одной из проблем стохастического градиентного спуска является неизменяемая во время обучения скорость обучения. Постоянная скорость обучения может привести к следующим проблемам: если ее значение будет выбрано слишком низким, то модель будет дольше сходиться и требовать большего числа итераций для достижения оптимального решения, если ее значение будет слишком большим, то модель может расходиться и на каждой итерации проходить мимо глобального минимума.

Для решения этой проблемы нужно использовать адаптивно настраивающуюся скорость обучения. Значение скорости обучения для каждого параметра должно настраиваться адаптивно, исходя из правила, что чем больше значение ошибки, тем больше должна быть скорость обучения. Увеличение скорости обучения при больших значениях ошибки дает возможность перескочить через локальные минимумы, а ее уменьшение при малых значениях не дает модели на каждой итерации перескакивать через глобальный минимум.

Для адаптивного изменения весов модели можно использовать алгоритм RMSProps. Этот алгоритм работает по следующим правилам:

- на каждой итерации для каждого параметра считается экспоненциальное скользящее среднее градиента с учетом всей истории обучения;
- при помощи полученных значений для каждого параметра вычисляется скорость обучения и производится обновление весов модели.

Экспоненциальное скользящее среднее на очередной итерации высчитывается по формуле 4.1

$$E_{\mathrm{t}} = \beta * g_{\mathrm{t}}^2 + (1 - \beta) * E_{\mathrm{t}-1}, \tag{4.1}$$

где E_{t} – новое полученное значение экспоненциального скользящего среднего, $E_{\mathrm{t}-1}$ – значение, полученное на предыдущей итерации, β – настраиваемый коэффициент, g_{t} – градиент функции потерь по соответствующему параметру.

После этого веса обновляются с использованием соотношения 4.2

$$w_{\mathrm{t}} = w_{\mathrm{t}-1} + \frac{\eta}{\sqrt{E_{\mathrm{t}}}} g, \tag{4.2}$$

где w_{t} – новое полученное значение параметра модели, $w_{\mathrm{t}-1}$ – предыдущее значение этого параметра, η – настраиваемый коэффициент, E_{t} – значение экспоненциального скользящего среднего для этого параметра.

На рисунках 4.1 и 4.2 представлены графики сравнения точность моделей, обученных с помощью стохастического градиентного спуска и RMSProps, на тренировочной и тестовой выборках соответственно.

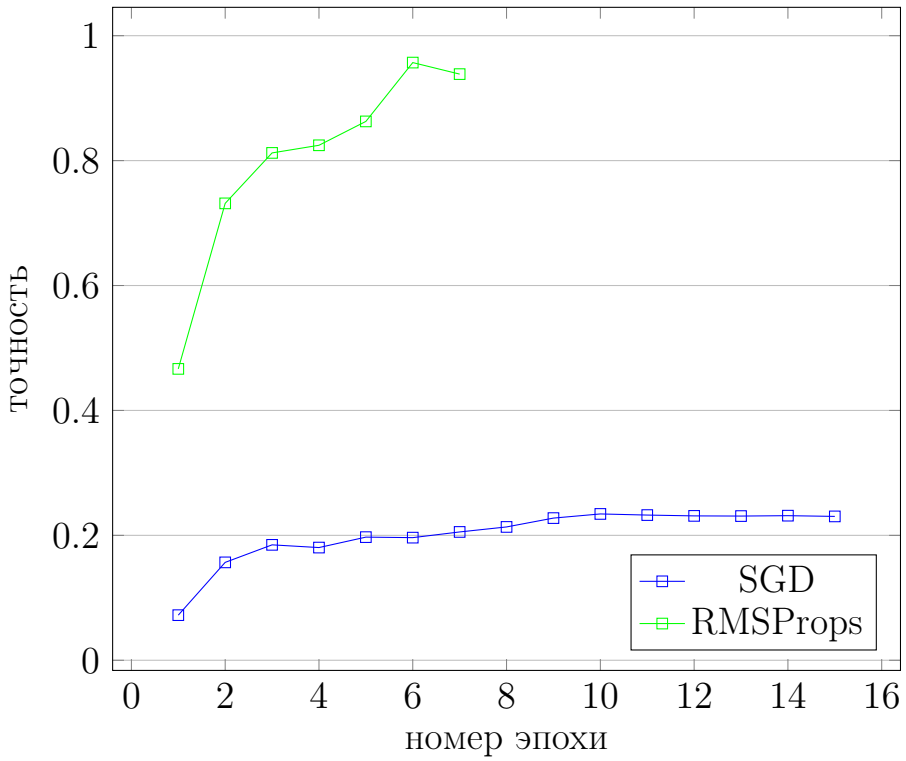

Рисунок 4.1 – Сравнение моделей, обученных на RMPSProp и SGD, на тренировочной выборке

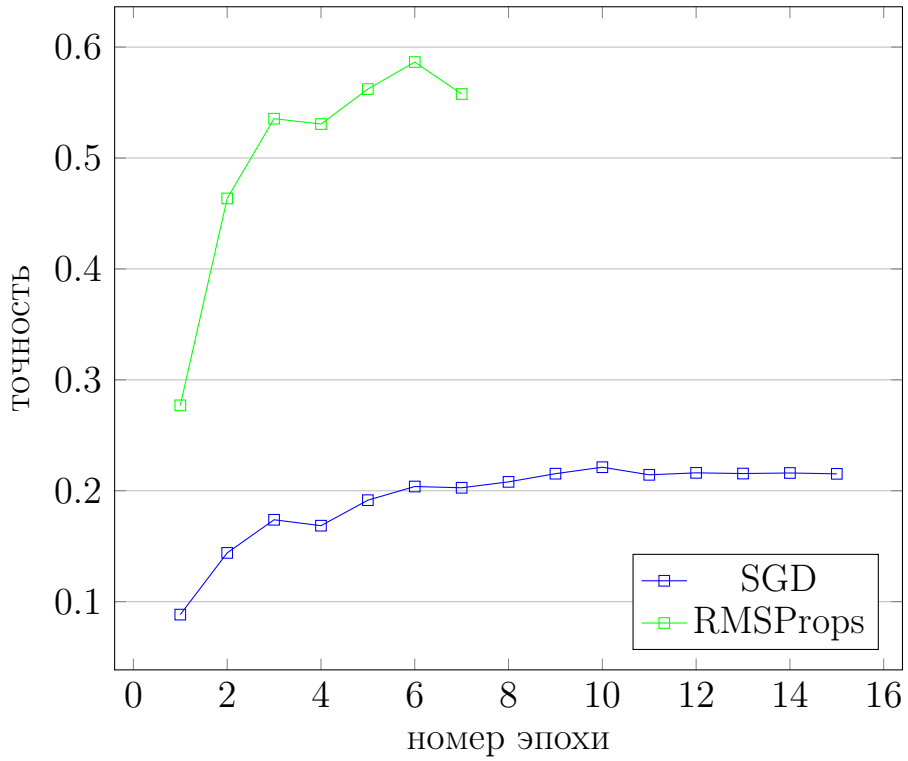

Рисунок 4.2 – Сравнение моделей, обученных на RMPSProp и SGD, на тестовой выборке

Из полученных результатов можно сделать вывод о том, что модель, которая обучалась с использованием градиентного спуска попала в локальный минимум функции потерь и перестала обучаться.

Модель, которая обучалась с помощью RMSProps показывает лучшую сходимость и большую точность в сравнении с градиентным спуском за счет адаптивного вычисления скорости обучения на каждой итерации.

Для улучшения сходимости модели при адаптивном обновлении скорости обучения можно считать экспоненциальное скользящее среднее не только по квадрату градиента, но и по самому значению и использовать оба полученных значения при подсчете новой скорости обучения на каждой итерации. Такой подход реализован в алгоритме Adam.

Первый и второй моменты высчитываются по формулам 4.3 и 4.4 соответственно

$$m_t = \beta_1 * g_t + (1 - \beta_1) * m_{t-1}, \tag{4.3}$$

$$v_t = \beta_2 * g_t^2 + (1 - \beta_2) * v_{t-1}, \tag{4.4}$$

где m_t и v_t – первый и второй моменты в соответствующий момент времени, β_1 и β_2 – настраиваемые коэффициента, g – градиент функции потерь по соответствующему параметру.

Для учета начальных значений скользящий средний к ним применяется корректировка по формулам 4.5 и 4.6.

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}, \tag{4.5}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t}, \tag{4.6}$$

Итоговое обновление весов осуществляется по формуле 4.7

$$w_t = w_{t-1} + \alpha \frac{\hat{m_t}}{\sqrt{\hat{v_t} + \epsilon}}, \tag{4.7}$$

где w_t – новое полученной значение параметра модели, w_{t-1} – предыдущее значение этого параметра, α – скорость обучения, ϵ – поправка, защищающая от деления на ноль.

На рисунках 4.3 и 4.4 представлены графики сравнения точность моделей, обученных с помощью Adam и RMSProps, на тренировочной и тестовой выборках соответственно.

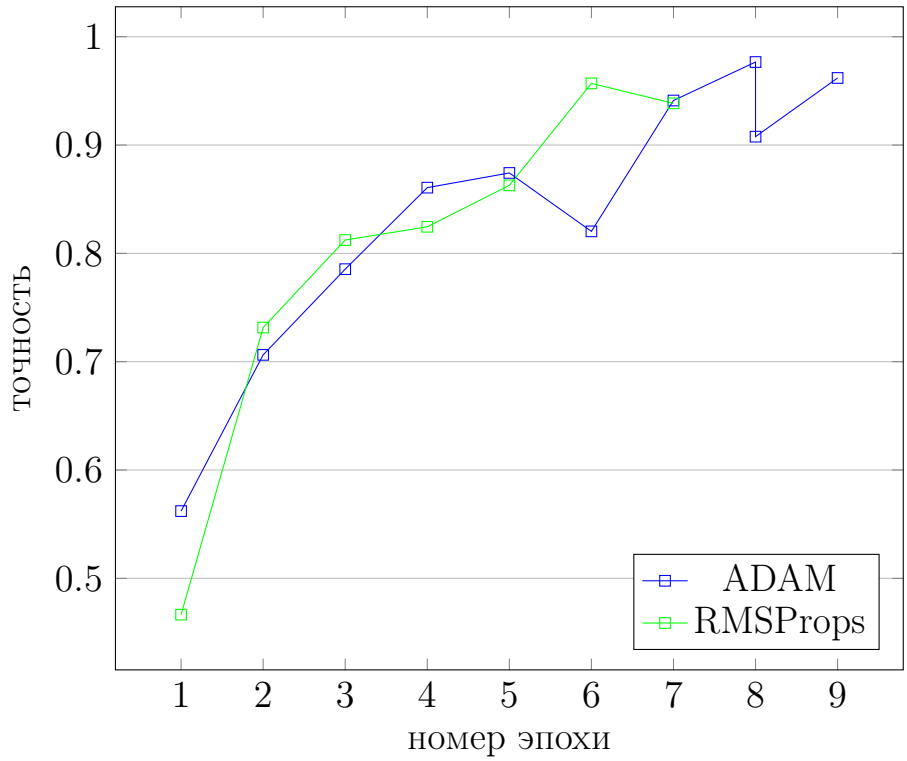

Рисунок 4.3 – Сравнение моделей, обученных на RMPSProp и ADAM, на тренировочной выборке

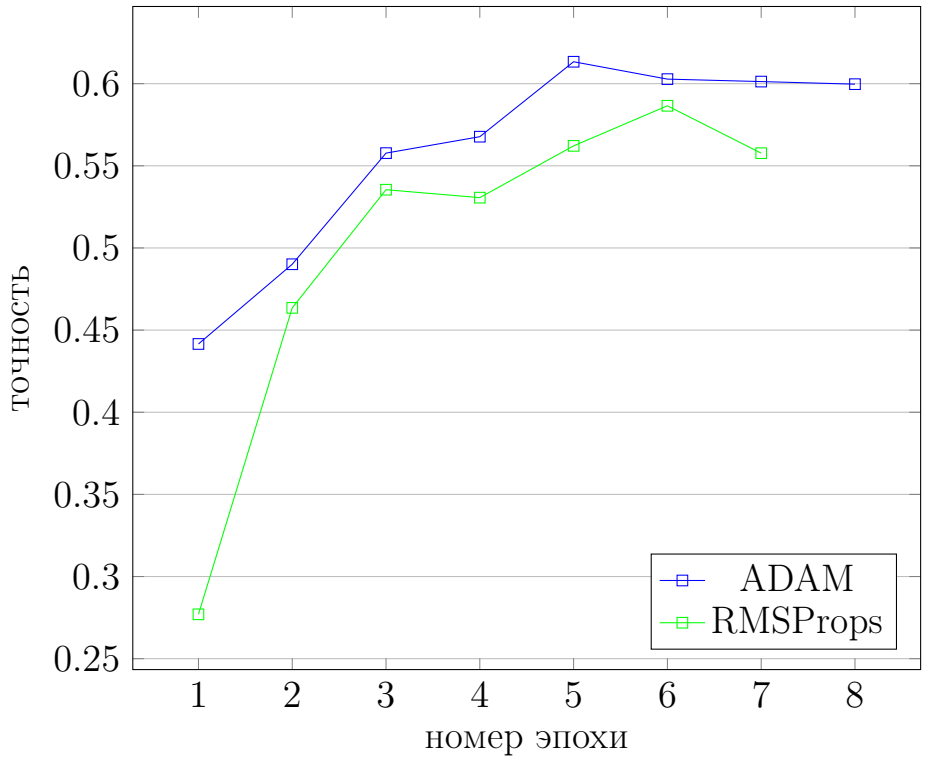

Рисунок 4.4 – Сравнение моделей, обученных на RMPSProp и ADAM, на тестовой выборке

Из графиков видно, что Adam сходится быстрее, а модель, обученная с использованием этого оптимизатора, имеет большую точность распознавания на тестовой выборке.

4.2 Вывод

При сравнении различных оптимизаторов было выяснено, что скорость обучения сети должна адаптивно настраиваться в зависимости от текущего значения ошибки по правилу: чем больше ошибка, тем больше скорость обучения. В ином случае модель может дольше сходиться при малом значении скорости обучения, либо расходиться при больших значениях этого параметра.

Алгоритмы Adam и RMSProps используются для адаптивной настройки скорости обучения. В RMSprops учитывается экспоненциальное скользящее среднее по квадрату градиента для учета истории обучения при обновлении весов модели. В алгоритме Adam также учитывается экспоненциальное скользящее среднее и по первому моменту, что позволяет ему достичь большей сходимости при обучении модели.

Таким образом, при обучении модели для решения поставленной задачи лучше всего подходит оптимизатор Adam за счет большей сходимости и большей точности итоговой обученной модели.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был спроектирован и разработан метод распознавания летательных аппаратов с аэрофотоснимков с использование нейронных сетей. В ходе выполнения работы были выполнены следующие задачи:

- определены термины, связанные с предметной областью распознавания объектов;
- проведен анализ методов распознавания летальной техники на аэрофотоснимках;
- определены критерии сравнения методов;
- проведен их сравнительный анализ;
- спроектирован метод распознавания летальной техники на аэрофотоснимках;
- реализован спроектированный метод;
- проведено исследование точности распознавания модели на тестовой выборке при различных подходах к обучению.

Цель работы достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Самойлин Е.* АЛГОРИТМ РАСПОЗНАВАНИЯ ИЗОБРАЖЕНИЙ НА ОСНОВЕ ГРАДИЕНТНОГО СОВМЕЩЕНИЯ ОБЪЕКТА С ЭТАЛОНОМ. — Сборник трудов XXV Международной научно-технической конференции, 2019. — с. 150.
2. *Мифтахова А. А.* ПРИМЕНЕНИЕ МЕТОДА ДЕРЕВА РЕШЕНИЙ В ЗАДАЧАХ КЛАССИФИКАЦИИ И ПРОГНОЗИРОВАНИЯ. — Поволжский государственный университет телекоммуникаций и информатики, 2016. — с. 7.
3. *Бабушкина Н. Е.* ВЫБОР ФУНКЦИИ АКТИВАЦИИ НЕЙРОННОЙ СЕТИ В ЗАВИСИМОСТИ ОТ УСЛОВИЙ ЗАДАЧИ. — Донской государственный технический университет, 2022. — с. 4.
4. *Антонов Г. В.* ПРОСТАЯ НЕЙРОННАЯ СЕТЬ И ЕЕ ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ. — ФГБОУ ВО Великолукская государственная сельскохозяйственная академия, 2021. — с. 11.
5. *Левченко К. М.* Нейронные сети. — Белорусский государственный университет информатики и радиоэлектроники, 2022. — с. 5.
6. *Барвинский Д. А.* Применение метода градиентного спуска в решении задач оптимизации. — Тенденции развития науки и образования, 2021.
7. *Апарнев А. Н.* Анализ функций потерь при обучении сверточных нейронных сетей с оптимизатором Adam для классификации изображений. — ВЕСТНИК МОСКОВСКОГО ЭНЕРГЕТИЧЕСКОГО ИНСТИТУТА. ВЕСТНИК МЭИ, 2020.
8. *Митина О.* ПЕРЦЕПТРОН В ЗАДАЧАХ БИНАРНОЙ КЛАССИФИКАЦИИ. — Национальная ассоциация ученых, 2021. — с. 6.
9. *Сикорский О. С.* Обзор свёрточных нейронных сетей для задачи классификации изображений. — Новые информационные технологии в автоматизированных системах, 2017. — с. 8.
10. *Алексеев И. П.* ПЕРСПЕКТИВЫ ПРИМЕНЕНИЯ КАПСУЛЬНЫХ НЕЙРОННЫХ СЕТЕЙ В РАСПОЗНАВАНИИ ОБЪЕКТОВ НА ИЗОБРАЖЕНИЯХ. — ТИНЧУРИНСКИЕ ЧТЕНИЯ - 2021 «ЭНЕРГЕТИКА И ЦИФРОВАЯ ТРАНСФОРМАЦИЯ», 2021. — с. 4.

11. *J. X.* CapsNet, CNN, FCN: Comparative Performance Evaluation for Image Classification // International Journal of Machine Learning and Computing. — 2019. — т. 9, № 6. — с. 9.

12. MNIST dataset. — Дата обращения: 28.04.2023. Режим доступа: `https://www.tensorflow.org/datasets/catalog/mnist`.

13. CIFAR-10 dataset. — Дата обращения: 28.04.2023. Режим доступа: `https://www.tensorflow.org/datasets/catalog/cifar10`.

14. *Паршин С. Е.* Исследование параметров алгоритмов распознавания лиц. — Сборник научных трудов Новосибирского государственного технического университета, 2019. — с. 6.

15. *Э. Я. Р.* Разработка программного средства для идентификации номерных знаков транспортных средств на основе методов компьютерного зрения // Journal of new century innovations. — 2022. — т. 15, № 1. — с. 81—93.

16. *Береснев Д. В.* КЛАССИФИКАЦИЯ ГАЛАКТИК С ПОМОЩЬЮ КАПСУЛЬНЫХ НЕЙРОННЫХ СЕТЕЙ. — БГУИР, 2019. — с. 4.

17. *Воронецкий Ю. О.*, *Жданов Н. А.* Методы борьбы с переобучением искусственных нейронных сетей // Научный аспект. — 2019. — т. 13, № 2. — с. 1639—1647.

18. *В. П. А.* Формирование обучающей выборки в задачах машинного обучения. Обзор // Информационно-управляющие системы. — 2021. — 4 (113). — с. 61—70.

19. *Z. C.* Cascade R-CNN: High Quality Object Detection and Instance Segmentation // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2021. — т. 43, № 5. — с. 1483—1498.

20. *О.А. П.* МЕТОДЫ И ПРОБЛЕМЫ ПЕРЕОБУЧЕНИЯ МНОГОСЛОЙНОЙ НЕЙРОННОЙ СЕТИ // ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ В СТРОИТЕЛЬНЫХ, СОЦИАЛЬНЫХ И ЭКОНОМИЧЕСКИХ СИСТЕМАХ. — 2020. — с. 101—103.

21. *Lecun Y.* Efficient Backprop // Neural Networks: Tricks of the Trade. — 2019. — с. 99—48.

22. *Николенко С.* Глубокое обучение. Погружение в мир нейронных сетей. — Издательство Питер, 2018. — с. 477.

23. *С. К. Ю.* Ансамблевый метод машинного обучения, основанный на рекомендации классификаторов // Интеллектуальные системы. Теория и приложения. — 2015. — т. 19, № 4. — с. 37—55.

24. *Y. LeCun L. Bottou Y. B., Haffner P.* Gradient-Based Learning Applied to Document Recognition. — Proceedings of the IEEE, 1998. — с. 46.

25. Different types of CNN Architectures. — Дата обращения: 28.04.2023. Режим доступа: `https://vitalflux.com/different-types-of-cnn-architectures-explained-examples/`.

26. *C. Szegedy W. Liu Y. J.* Going Deeper With Convolutions. — Proceedings of the IEEE Conference on Computer Vision, Pattern Recognition (CVPR), 2015. — с. 12.

27. Understanding Capsule Networks. — Дата обращения: 28.04.2023. Режим доступа: `https://www.freecodecamp.org/news/understanding-capsule-networks-ais-allurin-new-architecture-bdb228173ddc`.

28. *Nitin R. Y. A.* MobileNets for flower classification using TensorFlow // 2017 international conference on big data, IoT and data science (BID). — IEEE. 2017. — с. 154—158.

29. Military Aircraft Recognition dataset. — Дата обращения: 28.02.2023. Режим доступа: `https://www.kaggle.com/datasets/khlaifiabilel/military-aircraft-recognition-dataset`.

ПРИЛОЖЕНИЕ А
Модуль модели

Листинг А.1 – Модель для классификации самолетов

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F

class simplenet(nn.Module):
    def __init__(self, classes=20, simpnet_name='simplenet'):
        super(simplenet, self).__init__()
        #print(simpnet_name)
        self.features = self._make_layers()
           #self._make_layers(cfg[simpnet_name])
        self.classifier = nn.Linear(256, classes)
        self.drp = nn.Dropout(0.1)

    def load_my_state_dict(self, state_dict):

        own_state = self.state_dict()

        # print(own_state.keys())
        # for name, val in own_state:
        # print(name)
        for name, param in state_dict.items():
            name = name.replace('module.', '')
            if name not in own_state:
                # print(name)
                continue
            if isinstance(param, Parameter):
                # backwards compatibility for serialized
                   parameters
                param = param.data
            print("STATE_DICT: {}".format(name))
            try:
                own_state[name].copy_(param)
            except:
                print('While copying the parameter named {},
                   whose dimensions in the model are'
```

```
                    ' {} and whose dimensions in the
                       checkpoint are {}, ... Using Initial
                       Params'.format(
                  name, own_state[name].size(), param.size()))

    def forward(self, x):
        out = self.features(x)

        # print(out.shape)
        #Global Max Pooling
        out = F.max_pool2d(out, kernel_size=out.size()[2:])
        # out = F.dropout2d(out, 0.1, training=True)
        out = self.drp(out)

        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def _make_layers(self):

        model = nn.Sequential(
                              nn.Conv2d(3, 64, kernel_size=[3,
                                 3], stride=(1, 1), padding=(1,
                                 1)),
                              nn.BatchNorm2d(64, eps=1e-05,
                                 momentum=0.05, affine=True),
                              nn.ReLU(inplace=True),

                              nn.Conv2d(64, 128, kernel_size=[3,
                                 3], stride=(1, 1), padding=(1,
                                 1)),
                              nn.BatchNorm2d(128, eps=1e-05,
                                 momentum=0.05, affine=True),
                              nn.ReLU(inplace=True),

                              nn.Conv2d(128, 128, kernel_size=[3,
                                 3], stride=(1, 1), padding=(1,
                                 1)),
                              nn.BatchNorm2d(128, eps=1e-05,
                                 momentum=0.05, affine=True),
                              nn.ReLU(inplace=True),
```

```
nn.Conv2d(128, 128, kernel_size=[3,
    3], stride=(1, 1), padding=(1,
    1)),
nn.BatchNorm2d(128, eps=1e-05,
    momentum=0.05, affine=True),
nn.ReLU(inplace=True),


nn.MaxPool2d(kernel_size=(2, 2),
    stride=(2, 2), dilation=(1, 1),
    ceil_mode=False),
nn.Dropout2d(p=0.1),


nn.Conv2d(128, 128, kernel_size=[3,
    3], stride=(1, 1), padding=(1,
    1)),
nn.BatchNorm2d(128, eps=1e-05,
    momentum=0.05, affine=True),
nn.ReLU(inplace=True),

nn.Conv2d(128, 128, kernel_size=[3,
    3], stride=(1, 1), padding=(1,
    1)),
nn.BatchNorm2d(128, eps=1e-05,
    momentum=0.05, affine=True),
nn.ReLU(inplace=True),

nn.Conv2d(128, 256, kernel_size=[3,
    3], stride=(1, 1), padding=(1,
    1)),
nn.BatchNorm2d(256, eps=1e-05,
    momentum=0.05, affine=True),
nn.ReLU(inplace=True),



nn.MaxPool2d(kernel_size=(2, 2),
    stride=(2, 2), dilation=(1, 1),
    ceil_mode=False),
```

```
            nn.Dropout2d(p=0.1),


            nn.Conv2d(256, 256, kernel_size=[3,
               3], stride=(1, 1), padding=(1,
               1)),
            nn.BatchNorm2d(256, eps=1e-05,
               momentum=0.05, affine=True),
            nn.ReLU(inplace=True),


            # nn.Conv2d(256, 256,
               kernel_size=[3, 3], stride=(1,
               1), padding=(1, 1)),
            # nn.BatchNorm2d(256, eps=1e-05,
               momentum=0.05, affine=True),
            # nn.ReLU(inplace=True),



            nn.MaxPool2d(kernel_size=(2, 2),
               stride=(2, 2), dilation=(1, 1),
               ceil_mode=False),
            nn.Dropout2d(p=0.1),



            nn.Conv2d(256, 512, kernel_size=[3,
               3], stride=(1, 1), padding=(1,
               1)),
            nn.BatchNorm2d(512, eps=1e-05,
               momentum=0.05, affine=True),
            nn.ReLU(inplace=True),



            nn.MaxPool2d(kernel_size=(2, 2),
               stride=(2, 2), dilation=(1, 1),
               ceil_mode=False),
            nn.Dropout2d(p=0.1),
```

```
                        # TODO 256 -> 2048
                        nn.Conv2d(512, 2048,
                           kernel_size=[1, 1], stride=(1,
                           1), padding=(0, 0)),
                        nn.BatchNorm2d(2048, eps=1e-05,
                           momentum=0.05, affine=True),
                        nn.ReLU(inplace=True),



                        nn.Conv2d(2048, 256,
                           kernel_size=[1, 1], stride=(1,
                           1), padding=(0, 0)),
                        nn.BatchNorm2d(256, eps=1e-05,
                           momentum=0.05, affine=True),
                        nn.ReLU(inplace=True),


                        nn.MaxPool2d(kernel_size=(2, 2),
                           stride=(2, 2), dilation=(1, 1),
                           ceil_mode=False),
                        nn.Dropout2d(p=0.1),


                        nn.Conv2d(256, 256, kernel_size=[3,
                           3], stride=(1, 1)), # ,
                           padding=(1, 1)),
                        nn.BatchNorm2d(256, eps=1e-05,
                           momentum=0.05, affine=True),
                        nn.ReLU(inplace=True),

                       )

    for m in model.modules():
      if isinstance(m, nn.Conv2d):
        nn.init.xavier_uniform_(m.weight.data,
           gain=nn.init.calculate_gain('relu'))

    return model
```

Листинг А.2 – Модель для детектирования самолетов

```
config = [
    (32, 3, 1),
    (64, 3, 2),
    ["B", 1],
    (128, 3, 2),
    ["B", 2],
    (256, 3, 2),
    ["B", 8],
    (512, 3, 2),
    ["B", 8],
    (1024, 3, 2),
    ["B", 4],  # To this point is Darknet-53
    (512, 1, 1),
    (1024, 3, 1),
    "S",
    (256, 1, 1),
    "U",
    (256, 1, 1),
    (512, 3, 1),
    "S",
    (128, 1, 1),
    "U",
    (128, 1, 1),
    (256, 3, 1),
    "S",
]


class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, bn_act=True,
       **kwargs):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
           bias=not bn_act, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels)
        self.leaky = nn.LeakyReLU(0.1)
        self.use_bn_act = bn_act

    def forward(self, x):
        if self.use_bn_act:
            return self.leaky(self.bn(self.conv(x)))
```

```
        else:
            return self.conv(x)


class ResidualBlock(nn.Module):
    def __init__(self, channels, use_residual=True,
       num_repeats=1):
        super().__init__()
        self.layers = nn.ModuleList()
        for repeat in range(num_repeats):
            self.layers += [
                nn.Sequential(
                    CNNBlock(channels, channels // 2,
                       kernel_size=1),
                    CNNBlock(channels // 2, channels,
                       kernel_size=3, padding=1),
                )
            ]

        self.use_residual = use_residual
        self.num_repeats = num_repeats

    def forward(self, x):
        for layer in self.layers:
            if self.use_residual:
                x = x + layer(x)
            else:
                x = layer(x)

        return x


class ScalePrediction(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.pred = nn.Sequential(
            CNNBlock(in_channels, 2 * in_channels,
               kernel_size=3, padding=1),
            CNNBlock(
                2 * in_channels, (num_classes + 5) * 3,
                   bn_act=False, kernel_size=1
```

```
            ),
        )
        self.num_classes = num_classes

    def forward(self, x):
        return (
            self.pred(x)
            .reshape(x.shape[0], 3, self.num_classes + 5,
               x.shape[2], x.shape[3])
            .permute(0, 1, 3, 4, 2)
        )


class YOLOv3(nn.Module):
    def __init__(self, in_channels=3, num_classes=80):
        super().__init__()
        self.num_classes = num_classes
        self.in_channels = in_channels
        self.layers = self._create_conv_layers()

    def forward(self, x):
        outputs = []  # for each scale
        route_connections = []
        for layer in self.layers:
            if isinstance(layer, ScalePrediction):
                outputs.append(layer(x))
                continue

            x = layer(x)

            if isinstance(layer, ResidualBlock) and
               layer.num_repeats == 8:
                route_connections.append(x)

            elif isinstance(layer, nn.Upsample):
                x = torch.cat([x, route_connections[-1]], dim=1)
                route_connections.pop()

        return outputs

    def _create_conv_layers(self):
```

```
        layers = nn.ModuleList()
        in_channels = self.in_channels

        for module in config:
            if isinstance(module, tuple):
                out_channels, kernel_size, stride = module
                layers.append(
                    CNNBlock(
                        in_channels,
                        out_channels,
                        kernel_size=kernel_size,
                        stride=stride,
                        padding=1 if kernel_size == 3 else 0,
                    )
                )
                in_channels = out_channels

            elif isinstance(module, list):
                num_repeats = module[1]
                layers.append(ResidualBlock(in_channels,
                   num_repeats=num_repeats,))

            elif isinstance(module, str):
                if module == "S":
                    layers += [
                        ResidualBlock(in_channels,
                           use_residual=False, num_repeats=1),
                        CNNBlock(in_channels, in_channels // 2,
                           kernel_size=1),
                        ScalePrediction(in_channels // 2,
                           num_classes=self.num_classes),
                    ]
                    in_channels = in_channels // 2

                elif module == "U":
                    layers.append(nn.Upsample(scale_factor=2),)
                    in_channels = in_channels * 3

        return layers
```

```
if __name__ == "__main__":
    num_classes = 1
    IMAGE_SIZE = 800
    model = YOLOv3(num_classes=num_classes)
    x = torch.randn((2, 3, IMAGE_SIZE, IMAGE_SIZE))
    out = model(x)
    assert model(x)[0].shape == (2, 3, IMAGE_SIZE//32,
       IMAGE_SIZE//32, num_classes + 5)
    assert model(x)[1].shape == (2, 3, IMAGE_SIZE//16,
       IMAGE_SIZE//16, num_classes + 5)
    assert model(x)[2].shape == (2, 3, IMAGE_SIZE//8,
       IMAGE_SIZE//8, num_classes + 5)
    print("Success!")
```

Листинг А.3 – Класс для взаимодействия с обучаемым данными

```
from os import walk, path, listdir
import fnmatch
from torchvision import transforms
from torchvision.utils import save_image
from PIL import Image
import torch
import numpy as np
import matplotlib.pyplot as plt

DATASET_PATH: str = "../planes_dataset"
# TODO add base path
TRAIN_TENSORS_PATH: str = "./train_tensors"
TEST_TENSORS_PATH: str = "./test_tensors"

class DataSetHandler:
    _AUG_ROTATE_ANGLE = 30
    def TrainSize(self):
        files = listdir(TRAIN_TENSORS_PATH)
        return len(files) - 1 #file with y results

    def TestSize(self):
        files = listdir(TEST_TENSORS_PATH)
        return len(files) - 1 #file with y results

    def GetTrainingBatch(self, batchIndexes, needAug=False):
        xBatch, yBatch = self._GetBatch(batchIndexes,
           f"{TRAIN_TENSORS_PATH}/train")
```

```
        if needAug:
            xBatch, yBatch = self._AugmentateBatches(xBatch,
               yBatch)

        return xBatch, yBatch


    def GetTestBatch(self, batchIndexes):
        return self._GetBatch(batchIndexes,
           f"{TEST_TENSORS_PATH}/test")

    def UpdateData(self):
        self._UpdateTrainData()
        self._UpdateTestData()

    def _GetBatch(self, batchIndexes, pathPrefix):
        y = []
        with open(f"{pathPrefix}_results.txt") as classesFile:
            classes = classesFile.read().split('\n')
            for i in batchIndexes:
                y.append(int(classes[i]))

        x = []
        for i in batchIndexes:
            x.append(torch.load(f"{pathPrefix}_{i}.pt"))

        return torch.stack(x), torch.Tensor(y).to(torch.long)

    def _UpdateTrainData(self):
        xTrain = []
        yTrain = []
        with open(f"{DATASET_PATH}/ImageSets/Main/train.txt") as
           trainImagesFile:
            for imageNumber in trainImagesFile:
                images, classes =
                   self._FindAllImages(int(imageNumber))
                xTrain = xTrain + images
                yTrain = yTrain + classes

        for i in range(len(xTrain)):
            tensor: torch.Tensor =
```

```
            self._ConverToTensor(xTrain[i])
        torch.save(tensor,
            f"{TRAIN_TENSORS_PATH}/train_{i}.pt")

    with open(f"{TRAIN_TENSORS_PATH}/train_results.txt",
        "w") as f:
        for i in range(len(yTrain)):
            f.write(f"{yTrain[i]}\n")

def _UpdateTestData(self):
    xTest = []
    yTest = []
    with open(f"{DATASET_PATH}/ImageSets/Main/test.txt") as
        testImagesFile:
        for imageNumber in testImagesFile:
            images, classes =
                self._FindAllImages(int(imageNumber))
            xTest = xTest + images
            yTest = yTest + classes

    for i in range(len(xTest)):
        tensor: torch.Tensor = self._ConverToTensor(xTest[i])
        torch.save(tensor,
            f"{TEST_TENSORS_PATH}/test_{i}.pt")

    with open(f"{TEST_TENSORS_PATH}/test_results.txt", "w")
        as f:
        for i in range(len(yTest)):
            f.write(f"{yTest[i]}\n")


    # batchSize = 50
    # xTrainTensors = torch.stack(xTrain[:batchSize])
    # del(xTrain[:batchSize])
    # while len(xTrain) > 0:
    #     ram1 = psutil.virtual_memory().percent
    #     xTrainBatch = torch.stack(xTrain[:batchSize])
    #     xTrainTensors = torch.cat([xTrainTensors,
        xTrainBatch], dim=0)
    #     ram2 = psutil.virtual_memory().percent
    #     del(xTrain[:batchSize])
```

```
        #     # gc.collect()
        #     ram3 = psutil.virtual_memory().percent
        #     print(ram1, ram2, ram3)

        # return xTrainTensors, yTrain

    def _FindAllImages(self, imageNumber: int):
        images = []
        classes = []
        for root, dirnames, filenames in
           walk(f"{DATASET_PATH}/Parsed/"):
            for filename in fnmatch.filter(filenames,
               f"{imageNumber}_*.png"):
                planeImage = path.join(root, filename)
                images.append(planeImage)
                classNameIndex =
                   path.dirname(planeImage).rfind("A")
                classNumber =
                   int(path.dirname(planeImage)[classNameIndex+1:])
                   - 1
                classes.append(classNumber)

        return (images, classes)

    def _ConverToTensor(self, imagePath: str) -> torch.Tensor:
        image: Image.Image = Image.open(imagePath)
        transform = transforms.ToTensor()
        tensor: torch.Tensor = transform(image)
        # remove alpha channel
        if (tensor.size(0) == 4):
            tensor = tensor[:-1]

        return tensor

    def _AugmentateBatches(self, xBatch: torch.Tensor, yBatch:
       torch.Tensor):
        xBatchAug = []
        yBatchAug = []

        for i, tensor in enumerate(xBatch):
            augTensor1 = transforms.functional.rotate(tensor,
```

```
                self._AUG_ROTATE_ANGLE)
            augTensor2 = transforms.functional.rotate(tensor,
                -self._AUG_ROTATE_ANGLE)
            augTensor3 =
                transforms.functional.adjust_brightness(tensor,
                1.5)
            augTensor4 =
                transforms.functional.gaussian_blur(tensor,
                kernel_size=(5,9), sigma=3)
            # save_image(tensor, "tensor.png")
            # save_image(augTensor1, "augTensor1.png")
            # save_image(augTensor2, "augTensor2.png")
            # save_image(augTensor3, "augTensor3.png")
            # save_image(augTensor4, "augTensor4.png")
            # exit()

            xBatchAug += [tensor, augTensor1, augTensor2,
                augTensor3, augTensor4]
            yBatchAug += [yBatch[i].item()] * 5

        order = np.random.permutation(len(xBatchAug))
        xBatchAug = np.array(xBatchAug)[order]
        yBatchAug = np.array(yBatchAug)[order]

        return torch.stack(list(xBatchAug)),
            torch.Tensor(list(yBatchAug)).to(torch.long)



if __name__ == "__main__":
    DataSetHandler().UpdateData()
    print(DataSetHandler().TrainSize())
    print(DataSetHandler().TestSize())
    xTrain, yTrain =
        DataSetHandler().GetTrainingBatch(range(100))
    print(xTrain[0], yTrain[0])
    print(xTrain.size(0), xTrain.size(1), xTrain.size(2),
        xTrain.size(3))

    im2display = np.transpose(xTrain[0], (1,2,0))
    plt.imshow(im2display)
```

```
    plt.show()
```

Листинг А.4 – Алгоритм прохода одной эпохи

```
from abc import ABC, abstractmethod
import numpy as np
from dataset_handler.dataset_handler import DataSetHandler
import gc

class INetworkController(ABC):
    @abstractmethod
    def TrainPrepare(self):
        pass

    @abstractmethod
    def TrainEpoch(self):
        pass

    @abstractmethod
    def GetResults(self, xBatch):
        pass

    @abstractmethod
    def GetResult(self, imagePath):
        pass

    @abstractmethod
    def SaveModel(self, modelPath):
        pass

    @abstractmethod
    def LoadModel(self, modelPath):
        pass

    @abstractmethod
    def GetAllModels(self):
        pass

    def TrainNetwork(self, epochs: int):
        self.TrainPrepare()

        trainAccs, testAccs = [], []
        for epoch in range(epochs):
```

```
            self.TrainEpoch()
            print(f"================================== {epoch}
               ==============================")
            trainAcc, testAcc = self.LogTraining()
            trainAccs.append(trainAcc)
            testAccs.append(testAcc)
            if len(testAccs) >= 3:
                # testAccsVar = np.var(testAccs[-3:])
                # print(f"testAccsVar == {testAccsVar}")
                # if testAccsVar < 0.05:
                if testAccs[-1] < testAccs[-2] and testAccs[-1]
                   < testAccs[-3]:
                    break

    def LogTraining(self):
        datasetHandler = DataSetHandler()

        predsTest = []
        yBatchTest = []
        for startTestBatch in range(0,
           datasetHandler.TestSize(), 500):
            # testOrder =
               np.random.permutation(datasetHandler.TestSize())[:500]
            xBatch, yBatch =
               datasetHandler.GetTestBatch(list(range(startTestBatch,
               min(startTestBatch + 500,
               datasetHandler.TestSize()))))
            preds = self.GetResults(xBatch)
            predsTest = predsTest + preds.tolist()
            yBatchTest = yBatchTest + yBatch.tolist()
            gc.collect()

        predsTrain = []
        yBatchTrain = []
        for startTrainBatch in range(0,
           datasetHandler.TrainSize(), 500):
            # xBatchTrain, yBatchTrain =
               datasetHandler.GetTrainingBatch(np.random.permutation(
            xBatch, yBatch =
               datasetHandler.GetTrainingBatch(list(range(startTrainBa
               min(startTrainBatch + 500,
```

```
        datasetHandler.TrainSize())))
    preds = self.GetResults(xBatch)
    predsTrain = predsTrain + preds.tolist()
    yBatchTrain = yBatchTrain + yBatch.tolist()
    gc.collect()

print("====================================")
testMisses = [0] * 20
testClasses = [0] * 20
recognizedTest = 0
for i in range(len(predsTest)):
    testClasses[yBatchTest[i]] += 1
    if predsTest[i] == yBatchTest[i]:
        recognizedTest += 1
    else:
        testMisses[yBatchTest[i]] += 1


trainMisses = [0] * 20
trainClasses = [0] * 20
recognizedTrain = 0
for i in range(len(predsTrain)):
    trainClasses[yBatchTrain[i]] += 1
    if predsTrain[i] == yBatchTrain[i]:
        recognizedTrain += 1
    else:
        trainMisses[yBatchTrain[i]] += 1


print("".join(map(lambda x: "{:6}".format(x),
   list(range(1,21)))))
print("".join(map(lambda x: "{:6}".format(x),
   testClasses)))
print("".join(map(lambda x: "{:6}".format(x),
   testMisses)))
print("".join(map(lambda x: "{:6}".format(x),
   trainClasses)))
print("".join(map(lambda x: "{:6}".format(x),
   trainMisses)))

testAcc = float(recognizedTest) / len(predsTest)
```

```
        trainAcc = float(recognizedTrain) / len(predsTrain)
        print(float(recognizedTest) / len(predsTest))
        print(float(recognizedTrain) / len(predsTrain))
        print("====================================")

        return trainAcc, testAcc
```

Листинг А.5 – Класс для взаимодействия с моделью

```
import torch
from PIL import Image
from torchvision import transforms
import numpy as np
import matplotlib.pyplot as plt
import os

from models.conv_network import PlanesNetwork
from models.simple_net import simplenet
from dataset_handler.dataset_handler import DataSetHandler

from network_controllers import INetworkController

class NetworkController(INetworkController):
    _TRAINED_MODELS_PATH = "trained_models/"

    def __init__(self, batchSize, learningRate, needAug=True):
        # self.m_planesNetwork = PlanesNetwork(20)
        # self.m_device = torch.device("cuda" if
           torch.cuda.is_available() else "cpu")
        # TODO
        self.m_device = "cpu"
        print(self.m_device)
        self.m_planesNetwork = simplenet(20).to(self.m_device)
        # print(sum(p.numel() for p in
           self.m_planesNetwork.parameters() if p.requires_grad))
        self.m_datasetHandler = DataSetHandler()

        self.m_batchSize = batchSize
        self.m_learningRate = learningRate
        self.m_needAug = needAug

        self.m_loss = torch.nn.CrossEntropyLoss()
        # TODO change on ADAM
```

```
        self.m_optimizer =
           torch.optim.Adam(self.m_planesNetwork.parameters(),
           lr=self.m_learningRate)
        self.m_datasetLen = self.m_datasetHandler.TrainSize()

    def TrainPrepare(self):
        pass

    def TrainEpoch(self):
        self.m_planesNetwork.train()
        order = np.random.permutation(self.m_datasetLen)
        for startIndex in range(0, self.m_datasetLen,
           self.m_batchSize):
            self.m_optimizer.zero_grad()

            xBatch, yBatch =
               self.m_datasetHandler.GetTrainingBatch(order[startIndex
               needAug=self.m_needAug)
            xBatch = xBatch.to(self.m_device)
            yBatch = yBatch.to(self.m_device)

            preds = self.m_planesNetwork.forward(xBatch)
            lossValue = self.m_loss(preds, yBatch)

            # print(preds.argmax(dim=1))
            # print(lossValue)
            lossValue.backward()

            self.m_optimizer.step()

    def GetResults(self, xBatch):
        self.m_planesNetwork.eval()
        results =
           self.m_planesNetwork.forward(xBatch).argmax(dim=1)
        self.m_planesNetwork.train()

        return results

    def GetResult(self, imagePath):
        image: Image.Image = Image.open(imagePath)
        transform = transforms.ToTensor()
```

```
        tensor: torch.Tensor = transform(image)
        # remove alpha channel
        if (tensor.size(0) == 4):
            tensor = tensor[:-1]

        # # TODO constant
        # stride = 5
        # i = 0
        # while i < len(tensor[0]) + 96:
        #     j = 0
        #     while j < len(tensor[0][0]) + 96:
        #         sector = tensor[:, i:i+96, j:j+96]

        #         im2display = np.transpose(sector, (1,2,0))
        #         plt.imshow(im2display)
        #         plt.show()

        #         t = torch.stack([sector])
        #         self.GetResults(t)


        #         j += stride
        #     i += stride

        t = torch.stack([tensor])
        return self.GetResults(t)[0]


    def SaveModel(self, modelPath):
        torch.save(self.m_planesNetwork,
           f"{self._TRAINED_MODELS_PATH}{modelPath}")

    def LoadModel(self, modelPath):
        self.m_planesNetwork =
           torch.load(f"{self._TRAINED_MODELS_PATH}{modelPath}")
        self.m_planesNetwork.to(self.m_device)

    def GetAllModels(self):
        models = []
        for _, _, filenames in
           os.walk(self._TRAINED_MODELS_PATH):
```

```
            for model in filenames:
                models.append(model)

        return models



if __name__ == "__main__":
    NetworkController().TrainNetwork(15, 100, 1e-3)
```