МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра информационных технологий

КУРСОВАЯ РАБОТА

РАСПОЗНАВАНИЕ АВТОМОБИЛЬНЫХ НОМЕРОВ ПРИ ПОМОЩИ РЕКУРРЕНТНОЙ НЕЙРОННОЙ СЕТИ

Работу выполнил____С.О. Брезицкий
(подпись)

Направление подготовки 01.03.02 Прикладная математика и информатика

Направленность Программирование и информационные технологии

Научный руководитель, доц.
канд. техн. наук, доц.____________________________________А.А. Полупанов
(подпись)
Нормоконтролер
канд. пед. наук, доц._____________________________________А.В. Харченко
(подпись)

Краснодар
2023

РЕФЕРАТ

Курсовая работа 39 с., 20 рис., 14 источников.

СВЁРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ, РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ, РАСПОЗНАВАНИЕ, СВЁРТКА, СИСТЕМА, ОБУЧЕНИЕ, МОДЕЛЬ

Цель работы: спроектировать систему для распознавания автомобильных номеров.

В процессе работы были изучены: основные понятия рекуррентных и свёрточных нейронных сетей, средства разработки и обучения рекуррентных и свёрточных нейронных сетей, архитектура и процесс разработки рекуррентных и свёрточных нейронных сетей.

В практической части был разработана система, позволяющая распознать автомобильный номер и считать с него символы с использованием рекуррентных свёрточных нейронных сетей. Представлена программа, которая реализует систему.

Средства разработки: язык программирования Python, библиотеки OpenCV, NumPy, Scikit-image, matplotlib, TensorFlow, Keras.

СОДЕРЖАНИЕ

Введение ... 4
1 Методика распознавания объектов ... 5
1.1 Постановка задачи ... 6
2 Свёрточные нейронные сети (СНС) ... 7
2.1 Структура свёрточных нейронных сетей ... 7
2.1.1 Свёрточный слой и операция свёртки ... 7
2.1.2 Субдискретизирующий слой ... 9
2.1.3 Полносвязный слой ... 10
3 Рекуррентные нейронные сети (РНС) ... 12
3.1 Распространение ошибки и архитектуры РНС ... 12
3.1.1 LSTM ... 15
3.1.2 GRU ... 18
4 Программная реализация ... 21
4.1 Выбор языка программирования ... 21
4.2 Используемые пакеты ... 21
4.3 Распознавание номера с помощью рекуррентной сети ... 23
4.3.1 Построение сети ... 23
4.3.2 Обучение сети ... 28
4.4 Распознавание номера ... 32
4.4.1 Выбор модели для выделения номера ... 32
4.4.2 Подготовка изображения ... 33
4.4.3 Распознавание символов ... 36
Заключение ... 37
Список использованных источников ... 38

ВВЕДЕНИЕ

Проблема автоматизированного оперативного распознавания текстовой информации является актуальной задачей, связанной с широким классом практических задач. Одной из таких задач является распознавания автомобильных номеров. Создание алгоритма, распознающего автомобильные номера, позволяет:

– автоматизировать контроль въезда и перемещения транспортных средств на объектах с ограниченным доступом и закрытых территориях;

– на автомагистралях обеспечить контроль транспортных потоков и осуществлять автоматическое трассирование угнанных транспортных средств и тех, за которыми числятся правонарушения;

– автоматизировать сбор статистики для муниципальных служб;

– автоматизировать контроль выезда оплаченных или неоплаченных транспортных средств на станциях технического обслуживания и автокомбинатах, контролировать загрузку зоны обслуживания;

– отслеживать въезд и выезд на автостоянках, осуществлять автоматический подсчет стоимости предоставленных услуг, контролировать свободное место;

– отслеживать въезд, выезд и время нахождения транспортных средств на территории склада и терминала, предотвращать возможные хищения.

Проблема становится актуальнее, если учесть, что на сегодняшний день в России, как и во всём мире, существует тенденция на увеличение количества автомобилей на душу населения. Следовательно, с увеличением числа автомобилей на дорогах возрастает необходимость контроля за ними.

1 Методика распознавания объектов

Процесс распознавания автомобильного номера изображен на рисунке 1.

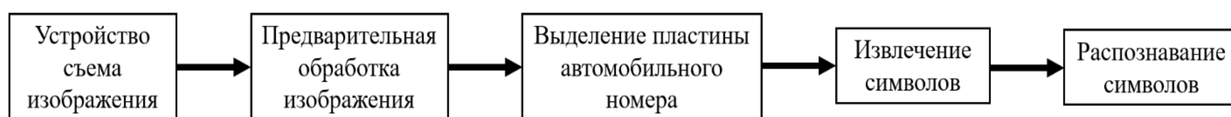

Рисунок 1 – Схема распознавания автомобильного номера

1) устройство съема изображения – видеокамера;

2) предварительная обработка полученного изображения включает следующие этапы: коррекция изображения – выравнивание, ограничение экстремальных значений яркости, устранение эффекта смазывания изображения, возникающего в связи с тем, что скорость автомобиля больше, чем скорость регистрации, устранение избыточной информации – использование бинаризации (перевод изображения из цветного в черно-белое);

3) выделение пластины номера выполняется при помощи сегментации;

4) на 4 и 5 этапах выполняется извлечение символов с изображения и их распознавание.

Одним из наиболее эффективных методов распознавания символов является рекуррентные свёрточные нейронные сети RCNN (Recurrent Convolutional Neural Network), которые являются логическим развитием идей таких архитектур нейронной сети как когнитрона и неокогнитрона.

Рекуррентные свёрточные нейронные сети используются для распознавания символов в изображениях, поскольку они могут фиксировать как пространственные, так и временные зависимости в данных. RCNN представляют собой комбинацию свёрточных нейронных сетей (CNN) и рекуррентных нейронных сетей (RNN), что позволяет им извлекать признаки из данных изображения с использованием свёрточных слоев, а также

учитывать последовательность пикселей в изображении с использованием рекуррентных слоев.

При распознавании символов RCNN могут анализировать форму и структуру символов с помощью свёрточных слоев, а затем использовать повторяющиеся слои для анализа последовательности пикселей в изображении для распознавания символа.

1.1 Постановка задачи

В данной курсовой работе исследован основной метод, который позволяет осуществить распознавание автомобильных номеров и символов, а именно рекуррентные нейронные сети. Рассмотрена структура и свойства этой сети. С помощью этого метода была спроектирована система для распознавания автомобильных номеров и символов, а также были исследованы особенности его работы при различных входных данных.

2 Свёрточные нейронные сети (СНС)

Свёрточные нейронные сети (convolutional neural networks, CNN) — это весьма широкий класс архитектур, основная идея которых состоит в том, чтобы переиспользовать одни и те же части нейронной сети для работы с разными небольшими, локальными участками входов [1].

2.1 Структура свёрточных нейронных сетей

СНС состоит из разных видов слоёв: свёрточные (convolutional) слои, субдискретизирующие (subsampling, подвыборка) слои и слои полносвязной нейронной сети – персептрона. Схема СНС представлена на рисунке 2.

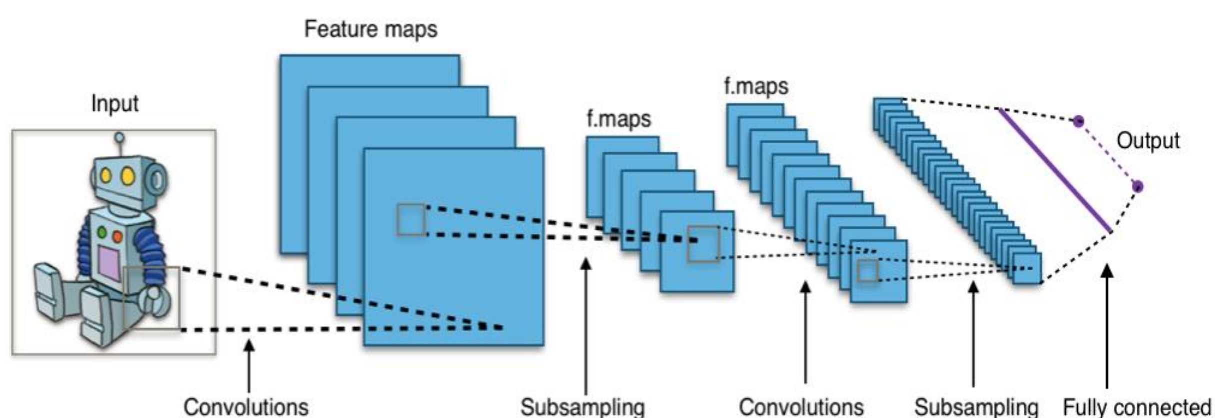

Рисунок 2 – Типовая архитектура свёрточной нейронной сети

Первые два типа слоев (convolutional, subsampling), чередуясь между собой, формируют входной вектор признаков для многослойного персептрона [2].

2.1.1 Свёрточный слой и операция свёртки

Основная идея свертечной сети состоит в том, что обработка участка изображения очень часто должна происходить независимо от конкретного расположения этого участка. Конечно, взаимное расположение объектов

играет важную роль, но сначала их нужно в любом случае распознать, и это распознавание — локально и независимо от конкретного положения участка с объектом внутри большой картинки.

Поэтому свёрточная сеть попросту делает это предположение в явном виде: давайте покроем вход небольшими окнами (скажем, 5×5 пикселов) и будем выделять признаки в каждом таком окне небольшой нейронной сетью. Причем — и тут ключевое соображение — признаки будем выделять в каждом окне одни и те же, то есть маленькая нейронная сеть будет всего одна, входов у нее будет всего 5×5 = 25, а из каждой картинки для нее может получиться очень много разных входов.

Затем результаты этой нейронной сети опять можно будет представить в виде «картинки», заменяя окна 5×5 на их центральные пикселы, и на ней можно будет применить второй свёрточный слой, с уже другой маленькой нейронной сетью, и т. д.

Осталось только формально определить, что же такое свертка и как устроены слои сверточной сети. Свертка — это всего лишь линейное преобразование входных данных особого вида. Если $\boldsymbol{x}^{\boldsymbol{l}}$ – карта признаков в слое под номером l, то результат двумерной свёртки с ядром размера $2d + 1$ и матрицей весов W размера $(2d+1)\times(2d+1)$ на следующем слое будет так:

$$y_{i,j}^{l} = \sum_{-d \le a,b \le d} W_{a,b} x_{i+a,j+b}^{l} \quad (1)$$

где

$y_{i,j}^{l}$ – результат свёртки на уровне l;

$x_{i,j}^{l}$ – её вход, то есть выход всего предыдущего слоя.

Иначе говоря, чтобы получить компоненту (i, j) следующего уровня, мы применяем линейное преобразование к квадратному окну предыдущего уровня, то есть скалярно умножаем пикселы из окна на вектор свертки [3].

Свёртки определяются двумя ключевыми параметрами:

– размер шаблонов, извлекаемых из входных данных, — обычно 3×3 или 5×5;

– глубина выходной карты признаков — количество фильтров, вычисляемых сверткой [4].

В упрощённом виде свёрточный слой можно описать формулой:

$$z^l = f(y^l + b^l) \quad (2)$$

где

z^l – выход слоя l;

y^l– её вход (результат свёртки);

$f()$ – функция активации;

b^l – коэффициент сдвига слоя l [2].

2.1.2 Субдискретизирующий слой

В свёрточных нейронных сетях со свёрточными слоями чередуются субдискретизирующие слои (первым слоем обязательно должен быть свёрточный), в котором используется операция субдискретизации.

Смысл этой операции заключается в следующем: в сверточных сетях обычно исходят из предположения, что наличие или отсутствие того или иного признака гораздо важнее, чем его точные координаты. Например, при распознавании номерных пластин сверточной сетью нам гораздо важнее понять, есть ли на фотографии пластина, чем узнать, с какого конкретно пиксела она начинается и в каком заканчивается. Поэтому можно позволить себе «обобщить» выделяемые признаки, потеряв часть информации об их местоположении, но зато сократив размерность.

Обычно в качестве операции субдискретизации к каждой локальной группе нейронов применяется операция взятия максимума (max-pooling). Формально определим субдискретизацию так:

$$y_{i,j}^{l+1} = max_{-d \le a \le d, -d \le b \le d} z_{i+a,j+b}^{l} \quad (3)$$

где

$y_{i,j}^{l+1}$ – результат субдискретизации на уровне l+1;

$z_{i,j}^{l}$ – выход слоя l.

Здесь d – это размер окна субдискретизации.

Хотя в результате субдискретизации действительно теряется часть информации, сеть становится более устойчивой к небольшим трансформациям изображения вроде сдвига или поворота [3].

Субдискретизирующий слой можно описать формулой:

$$z^{l+1} = f(a^{l+1} y^{l+1} + b^{l+1}) \quad (4)$$

где

z^{l+1} – выход слоя l+1;

y^{l+1}– её вход (результат субдискретизации);

f() – функция активации;

a^{l+1}, b^{l+1} – коэффициенты сдвига слоя $l+1$.

2.1.3 Полносвязный слой

После чередования сверточных слоёв и слоёв субдискретизации в конце идёт слой обычного многослойного персептрона. Цель слоя – классификация, он моделирует сложную нелинейную функцию, оптимизируя которую, улучшается качество распознавания.

Нейроны каждой карты предыдущего субдискретизирующего слоя связаны с одним нейроном скрытого слоя. Таким образом число нейронов скрытого слоя равно числу карт субдискретизирующего слоя, но связи могут быть не обязательно такими, например, только часть нейронов какой-либо из карт субдискретизирующего слоя быть связана с первым нейроном скрытого слоя, а оставшаяся часть со вторым, либо все нейроны первой карты связаны с нейронами 1 и 2 скрытого слоя. Вычисление значений нейрона можно описать формулой (6):

$$z_j^l = f\left(\sum_i z_i^{l-1} w_{i,j}^{l-1} + b_j^{l-1}\right) \quad (5)$$

где

z_j^l – карта признаков j (выход слоя l);

$f()$ – функция активации;

b_j^l – коэффициент сдвига слоя l;

$w_{i,j}^l$ – матрица весовых коэффициентов слоя l [2].

3 Рекуррентные нейронные сети (РНС)

Рекуррентные нейронные сети (recurrent neural networks, RNN) — это вид нейронных сетей, где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки [5].

РНС обрабатывает последовательность, перебирая ее элементы и сохраняя состояние, полученное при обработке предыдущих элементов. Фактически РНС — это разновидность нейронной сети, имеющей внутренний цикл. Сеть RNN сбрасывает состояние между обработкой двух разных, независимых последовательностей, поэтому одна последовательность все еще интерпретируется как единый блок данных: единственный входной пакет. Однако теперь блок данных обрабатывается не за один шаг; сеть выполняет внутренний цикл, перебирая последовательность элементов.

3.1 Распространение ошибки и архитектуры РНС

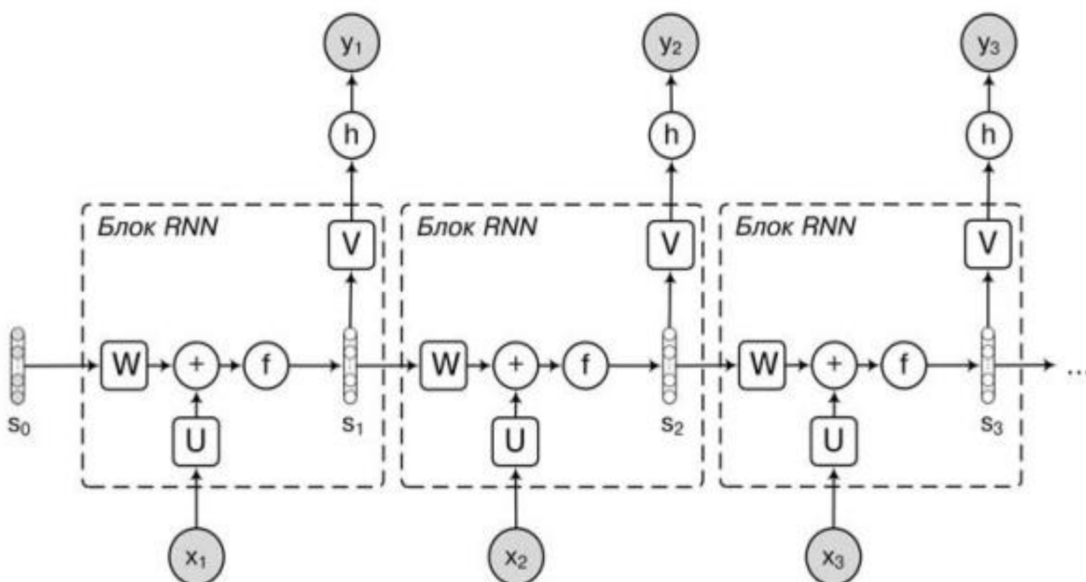

Рисунок 4 – Архитектура простой рекуррентной нейронной сети

Обозначим через W матрицу весов для перехода между скрытыми состояниями, через U матрицу весов для входов, а через V — для выходов. Архитектура простой рекуррентной нейронной сети показана на рисунке 4. Входы и скрытые состояния будем считать векторами. Будем считать, что на вход всегда подается один вектор $\boldsymbol{x}$; это не приводит ни к какой потере общности. В итоге мы получаем такое формальное задание сети — в момент времени t:

$$\boldsymbol{a}_t = \boldsymbol{b} + Ws_{t-1} + U\boldsymbol{x}_t,\ s_t = f(\boldsymbol{a}_t) \quad (6)$$

$$\boldsymbol{o}_t = \boldsymbol{c} + Vs_t,\ \boldsymbol{y}_t = h(\boldsymbol{o}_t) \quad (7)$$

где

f – это нелинейность рекуррентной сети (обычно σ, tanh или ReLU);

h – функция, с помощью которой получается ответ (например, softmax);

$\boldsymbol{b}, \boldsymbol{c}$ – свободные члены.

Собственно, блоком РНС называется часть от входов до вычисления $\boldsymbol{o}_t$; а слой РНС — это блок, развернутый во времени на входную последовательность (или несколько блоков, которые делят друг с другом веса).

Будем считать, что прошло T шагов, $t = 1..T$, и у нас задана какая-то функция ошибки $L(\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T)$. Теперь можно напрямую попытаться подсчитать частные производные, постепенно «разворачивая» внутренние состояния s_t от $t = T$ обратно к $t = 1$. Получится, что один и тот же вес из матрицы W участвует в этом процессе $T - 1$ раз, по числу переходов.

Еще один важный вариант РНС — это двунаправленные рекуррентные сети. Дело в том, что часто бывает так, что РНС к концу последовательности уже забывают, с чего там начиналось; и вообще последние элементы последовательности, даже если не забудется начало, всегда будут гораздо важнее первых и в обычной РНС, и в сети из LSTM или GRU-ячеек.

Поэтому часто рассматривают так называемые двунаправленные рекуррентные сети (bidirectional RNN). Давайте для входной последовательности запустим РНС (обычно с разными весами) два раза: один слой будет читать последовательность слева направо, а другой — справа налево. Матрицы весов абсолютно независимы, между ними нет взаимодействия, просто для каждого элемента последовательности получатся два состояния: слева направо и справа налево. Разумеется, это работает только для последовательностей, которые даны сразу целиком (как предложения естественного языка).

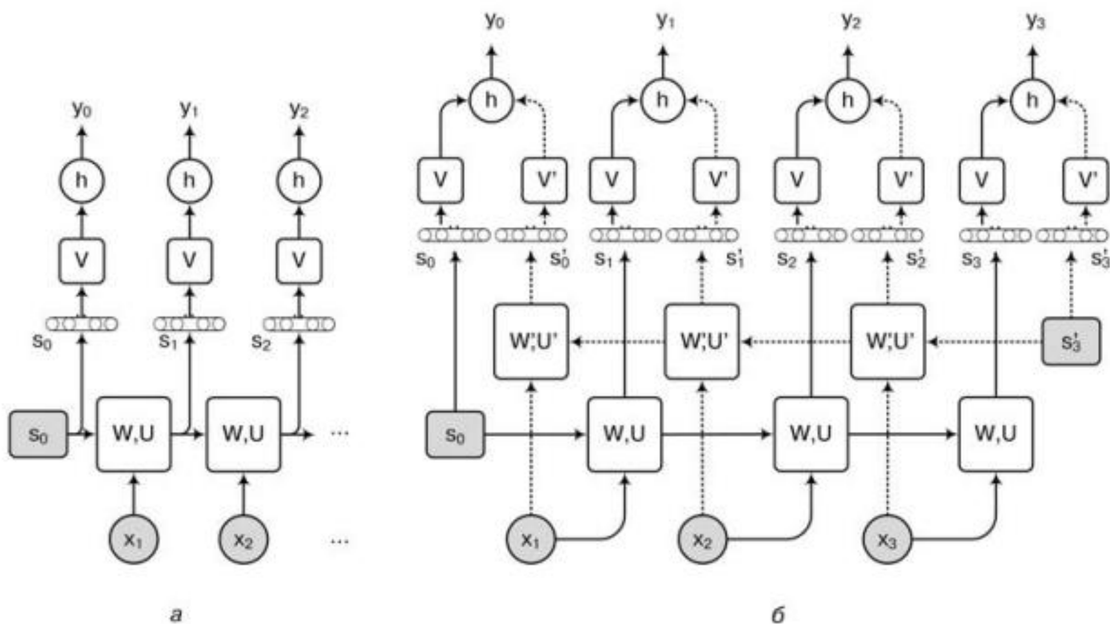

Рисунок 5 – Рекуррентные нейронные сети: *а* – обычная рекуррентная сеть; *б* – двунаправленная рекуррентная сеть; пунктиром изображены связи, относящиеся к сети, обрабатывающей входную последовательность справа налево

Всё это проиллюстрировано на рисунке 5, где слева изображена структура обычной рекуррентной нейронной сети, а справа, на рисунке 5, *б*, — двунаправленной. На этой схеме были «спрятаны» матрицы W и U в один блок и было сконцентрировано внимание на том, чтобы детально показать, что происходит с выходами; связи, относящиеся к идущей справа налево рекуррентной сети, на рисунке 5, *б* показаны пунктиром. Формально говоря, в

двунаправленной сети вычисляются состояния s_t слева направо и состояния s'_t справа налево, а затем сливаем их в один результат уже на уровне выхода; это значит, что выход вычисляется как:

$$s_t = \sigma(\boldsymbol{b} + Ws_{t-1} + U\boldsymbol{x}_t),\ s'_t = \sigma(\boldsymbol{b}' + W's'_{t-1} + U'\boldsymbol{x}_t) \tag{8}$$

$$\boldsymbol{o}_t = \boldsymbol{c} + Vs_t + V's'_t,\ \boldsymbol{y}_t = h(\boldsymbol{o}_t) \tag{9}$$

где

σ– это нелинейность рекуррентной сети;

h – функция, с помощью которой получается ответ (например, softmax);

$\boldsymbol{b}, \boldsymbol{c}$ – свободные члены.

Вместо классической рекуррентной сети из трех матриц, конечно, может стоять любая другая конструкция; на практике обычно используют двунаправленные LSTM или GRU.

3.1.1 LSTM

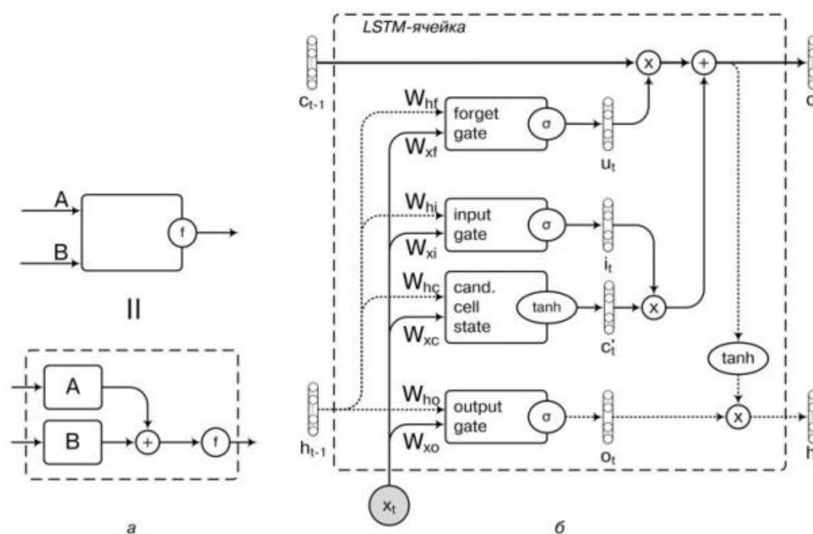

Рисунок 5 – LSTM: *а* – обозначение гейта с двумя входами; *б* – структура LSTM-ячейки

Стандартная архитектура LSTM-ячейки показана на рисунке 5[3]. В LSTM есть три основных вида узлов, которые называются гейтами: входной (input gate), забывающий (forget gate) и выходной (output gate), а также собственно рекуррентная ячейка со скрытым состоянием[6]. Кроме того, в LSTM часто добавляют ещё так называемые замочные скважины (peepholes) —дополнительные соединения, которые увеличивают связность модели.

Формально, если обозначить через $\boldsymbol{x}_t$ входной вектор во время *t*, через $\boldsymbol{h}_t$– вектор скрытого состояния во время *t*, через W_i (с разными вторыми индексами) – матрицы весов, применяющиеся ко входу, через W_h – матрицы весов в рекуррентных соединениях, а через $\boldsymbol{b}$ – векторы свободных членов, то получаем следующее формальное определение того, как работает LSTM: на очередном входе $\boldsymbol{x}_t$, имея скрытое состояние из предыдущего шага $\boldsymbol{h}_{t-1}$ и собственно состояние ячейки $\boldsymbol{c}_{t-1}$, последовательно вычисляем

$$\boldsymbol{c'}_t = tanh(W_{xc}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{c\prime}) \quad (10)$$

$$\boldsymbol{i}_t = \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \quad (11)$$

$$\boldsymbol{f}_t = \sigma(W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \quad (12)$$

$$\boldsymbol{o}_t = \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \quad (13)$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{c'}_t \quad (14)$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh{(\boldsymbol{c}_t)} \quad (15)$$

где

$\boldsymbol{c'}_t$ – кандидат в новое значение памяти (candidate cell state);

$\boldsymbol{i}_t$ – входной гейт (input gate);

$\boldsymbol{f}_t$ – забывающий гейт (forget gate);

$\boldsymbol{o}_t$ – выходной гейт (output gate);

$\boldsymbol{c}_t$ – вектор ячейки (cell state);

$\boldsymbol{h}_t$ – вектор скрытого состояния (block output).

На вход LSTM, как и в «обычной» РНС, подаются два вектора: новый вектор из входных данных $\boldsymbol{x}_t$ и вектор скрытого состояния $\boldsymbol{h}_{t-1}$, который получен из скрытого состояния этой ячейки на предыдущем шаге. Кроме того, внутри у каждого LSTM-блока есть «ячейка памяти» (cell) — вектор, который выполняет функцию памяти. Вектор ячейки на шаге *t* был обозначен выше через $\boldsymbol{c}_t$, а $\boldsymbol{c}'_t$, который получается в уравнении (10), — это вектор, полученный из входа и предыдущего скрытого состояния, который становится кандидатом на новое значение памяти. Получается он из $\boldsymbol{x}_t$ и $\boldsymbol{h}_{t-1}$ весьма обычным для нейронных сетей преобразованием: сначала линейная функция, потом гиперболический тангенс.

Но $\boldsymbol{c}'_t$ — это всего лишь кандидат в новое значение памяти. Прежде чем его запишут на место $\boldsymbol{c}_{t-1}$, значение-кандидат и старое значение проходят через еще два гейта: входной гейт $\boldsymbol{i}_t$ и забывающий гейт $\boldsymbol{f}_t$. В формуле (14) новое значение получается как линейная комбинация из старого с коэффициентами из забывающего гейта $\boldsymbol{f}_t$ и нового кандидата $\boldsymbol{c}'_t$ с коэффициентами из входного гейта $\boldsymbol{i}_t$. Там, где значения вектора забывающего гейта $\boldsymbol{f}_t$ будут близки к нулю, старое значение $\boldsymbol{c}_{t-1}$ «забудется», а там, где значения $\boldsymbol{i}_t$ будут велики, новый входной вектор прибавится к тому, что было в памяти.

Покомпонентное умножение приводит к тому, что на очередном шаге может быть перезаписана только часть «памяти» LSTM-ячейки; и какая это будет часть, тоже определяет сама ячейка в зависимости от того, что получается на выходах забывающего гейта $\boldsymbol{f}_t$ и входного гейта $\boldsymbol{i}_t$. И еще более того, поскольку эта линейная комбинация «мягкая» и по дороге все пропускается через сигмоиды σ, LSTM-ячейка может не просто выбрать,

записать новое значение или выкинуть его, а еще и сохранить любую линейную комбинацию старого и нового значения, причем коэффициенты могут быть разными в разных компонентах вектора; и эти решения ячейка принимает в зависимости от конкретного входа.

3.1.2 GRU

Архитектура LSTM требует довольно значительных ресурсов. В обычном RNN каждая ячейка имела один вектор скрытого состояния $\boldsymbol{h}$, а веса были представлены тремя матрицами (плюс свободные члены): матрица весов для входов U, матрица рекуррентных весов W и матрица весов для выходов V. В LSTM-ячейке весов становится гораздо больше. Даже в базовой модели участвует сразу восемь матриц весов: W_{xc}, W_{xi}, W_{xf}, W_{xo}, W_{hc}, W_{hi}, W_{hf}, W_{ho} [3].

Оказывается, что критически важными компонентами для успешной работы LSTM выступают, по сути, только два гейта: выходной и забывающий [6]. Кроме того, понятно, что ключевым моментом является сама «память» $\boldsymbol{c}_t$ и карусель константной ошибки, которая позволяет состоянию LSTM сохраняться надолго. Все остальное можно пытаться как-то сокращать.

Наиболее перспективным оказывается вариант LSTM со связанными входным и забывающим гейтом; от него уже буквально один шаг до упрощенной модели. Эта модель получила название gated recurrent unit (GRU) [7]. В этой архитектуре используется как раз идея совмещения выходного и забывающего гейта, а скрытое состояние $\boldsymbol{h}_t$ совмещено со значением памяти $\boldsymbol{c}_t$.

Вот как работает одна GRU-ячейка:

$$\boldsymbol{u}_t = \sigma(W_{xu}\boldsymbol{x}_t + W_{hu}\boldsymbol{h}_{t-1} + \boldsymbol{b}_u) \quad (16)$$

$$\boldsymbol{r}_t = \sigma(W_{xr}\boldsymbol{x}_t + W_{hr}\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \quad (17)$$

$$\boldsymbol{h'}_t = tanh(W_{xh'}\boldsymbol{x}_t + W_{hh'}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1})) \quad (18)$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{u}_t) \odot \boldsymbol{h'}_t + \boldsymbol{u}_t \odot \boldsymbol{h}_{t-1}. \quad (18)$$

Здесь $\boldsymbol{u}_t$ — это гейт обновления (update gate), который и является комбинацией входного и забывающего гейтов. А $\boldsymbol{r}_t$ — это гейт перезагрузки (reset gate); он тоже отвечает за то, какую часть памяти нужно перенести дальше с прошлого шага, но делает это еще до применения нелинейной функции. Ячейка памяти и выход блока $\boldsymbol{h}_t$ тут, в отличие от LSTM, никак не разделяются, и следующий выход $\boldsymbol{h}_t$ получается как комбинация (задаваемая гейтом $\boldsymbol{u}_t$) предыдущего выхода $\boldsymbol{h}_{t-1}$ и текущего кандидата в выход $\boldsymbol{h'}_t$, который, в свою очередь, тоже зависит от $\boldsymbol{h}_{t-1}$, но на этот раз через гейт перезагрузки $\boldsymbol{r}_t$.

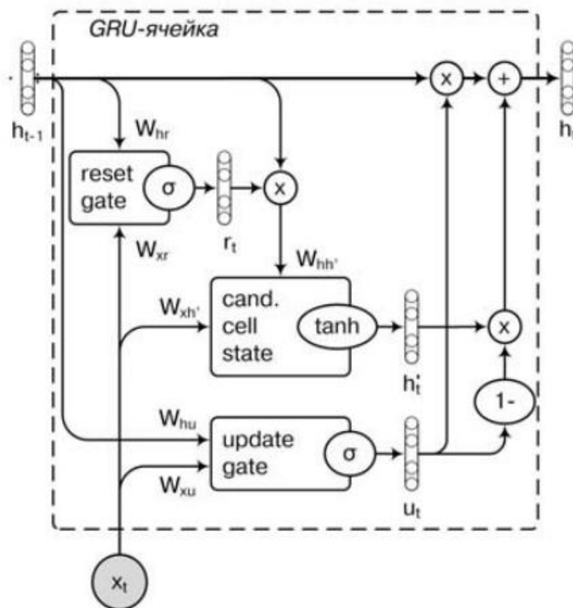

Рисунок 6 – структура GRU-ячейки

Все это проиллюстрировано на рисунке 6, где показана структура графа вычислений для одной GRU-ячейки.

Интуиция здесь в том, что t определяет, как объединить новый вход с имеющейся памятью, а $\boldsymbol{u}_t$ — какую часть имеющейся памяти оставить неизменной. Обычную РНС опять можно получить как частный случай GRU, если установить все компоненты t в единицу (всегда брать предыдущее состояние $\boldsymbol{h}_{t-1}$ целиком для вычисления вектора $\boldsymbol{h'}_t$), а все компоненты $\boldsymbol{u}_t$ в ноль (целиком заменять память новым кандидатом $\boldsymbol{h'}_t$). С точки же зрения градиентов мы снова видим, что между $\boldsymbol{h}_t$ и $\boldsymbol{h}_{t-1}$ есть линейная связь, карусель константной ошибки работает, и проблема затухающих градиентов успешно решается.

Основная разница между GRU и LSTM состоит в том, что GRU пытается сделать двумя гейтами то же самое, что LSTM делает тремя. Обязанности забывающего гейта $\boldsymbol{f}$ в LSTM здесь разделены между двумя гейтами, t и $\boldsymbol{u}_t$. Кроме того, не возникает второй нелинейности на пути от входа к выходу, как в случае LSTM. Нужно ещё заметить, что здесь опять нужно правильно проинициализировать свободные члены в гейте обновления $\boldsymbol{u}_t$: свободные члены $\boldsymbol{b}_u$ должны быть большими, иначе опять возникнет нежелательный эффект с экспоненциальным затуханием «памяти» в последовательности GRU [3].

4 Программная реализация

4.1 Выбор языка программирования

В качестве языка для моей программы был выбран высокоуровневый объектно-ориентированный язык общего назначения Python. Python – это интерпретируемый язык программирования. Исходный код преобразуется в машинный частями по мере его выполнения специальной программой, которая называется интерпретатор. Python отличается своим уникальным синтаксисом, он ориентирован на повышение читаемости кода и производительности программиста. Его отличительными особенностями являются: интроспекция, динамическая типизация, автоматическое управление памятью, механизм обработки исключений, удобные в использовании высокоуровневые структуры данных. Язык был создан в 1989—1991-х годах Гвидо Ван Россумом, и с того момента быстро стал популярен и востребован у программистов. Разработка и поддержка языка продолжается до сих пор. На данном этапе поддерживаются две ветки языка Python 3.x и Python 2.x. Этот язык создавался под влиянием других языков программирования, так, например, от языка ABC он позаимствовал отступы и высокоуровневые структуры данных, от Fortran появились срезы массивов, от C/C++ – некоторые синтаксические конструкторы и т. д. [8][9]

4.2 Используемые пакеты

— OpenCV

Это библиотека компьютерного зрения и машинного обучения с открытым исходным кодом, которая реализована на C++, однако имеет оболочку для Python, Java и других языков. Библиотека содержит более 2500 оптимизированных алгоритмов, включая полный набор как классических, так и современных алгоритмов компьютерного зрения и машинного обучения.

— NumPy

Это основной пакет, использующийся для научных вычислений при написании программ на языке Python. Он позволяет выполнять основные операции над n-мерными массивами и матрицами. Благодаря механизму векторизации NumPy повышает производительность и, соответственно, ускоряет выполнение операций.

— Scikit-image

Библиотека, предназначенная для обработки изображений.

— matplotlib

Это библиотека для визуализации обработанных данных двумерной и трёхмерной графикой. Она является основным инструментом для визуализации данных на языке Python, поддерживается различными платформами и IDE (iPython, Jupyter и пр.).

— TensorFlow

Открытая программная библиотека для машинного обучения, разработанная компанией Google для решения задач построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов. Библиотека использует многоуровневую систему узлов для обработки большого количества данных, что расширяет сферу ее использования далеко за пределы научной области.

— Keras

Открытая библиотека, обеспечивающая взаимодействие с искусственными нейронными сетями. Она использует возможности TensorFlow в качестве компонента

— PyTesseract

Это инструмент оптического распознавания символов (OCR) для Python. Является оболочкой для Tesseract OCR – свободной компьютерной программы для распознавания текстов.

4.3 Распознавание номера с помощью рекуррентной сети

4.3.1 Построение сети

Для распознавания символов с выделенной прямоугольной облости, с помощью библиотеки Keras была разработана свёрточная рекуррентная нейронная сеть, схема которой показана на рисунке 7. А на рисунке 8 изображена схема программной реализации данной сети.

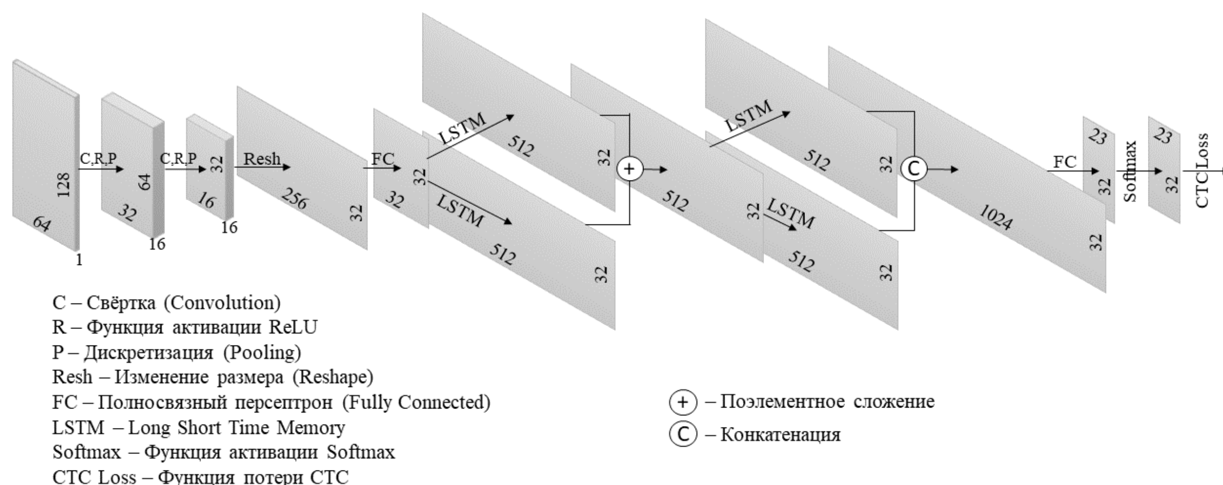

Рисунок 7 – Схема модели СРНС

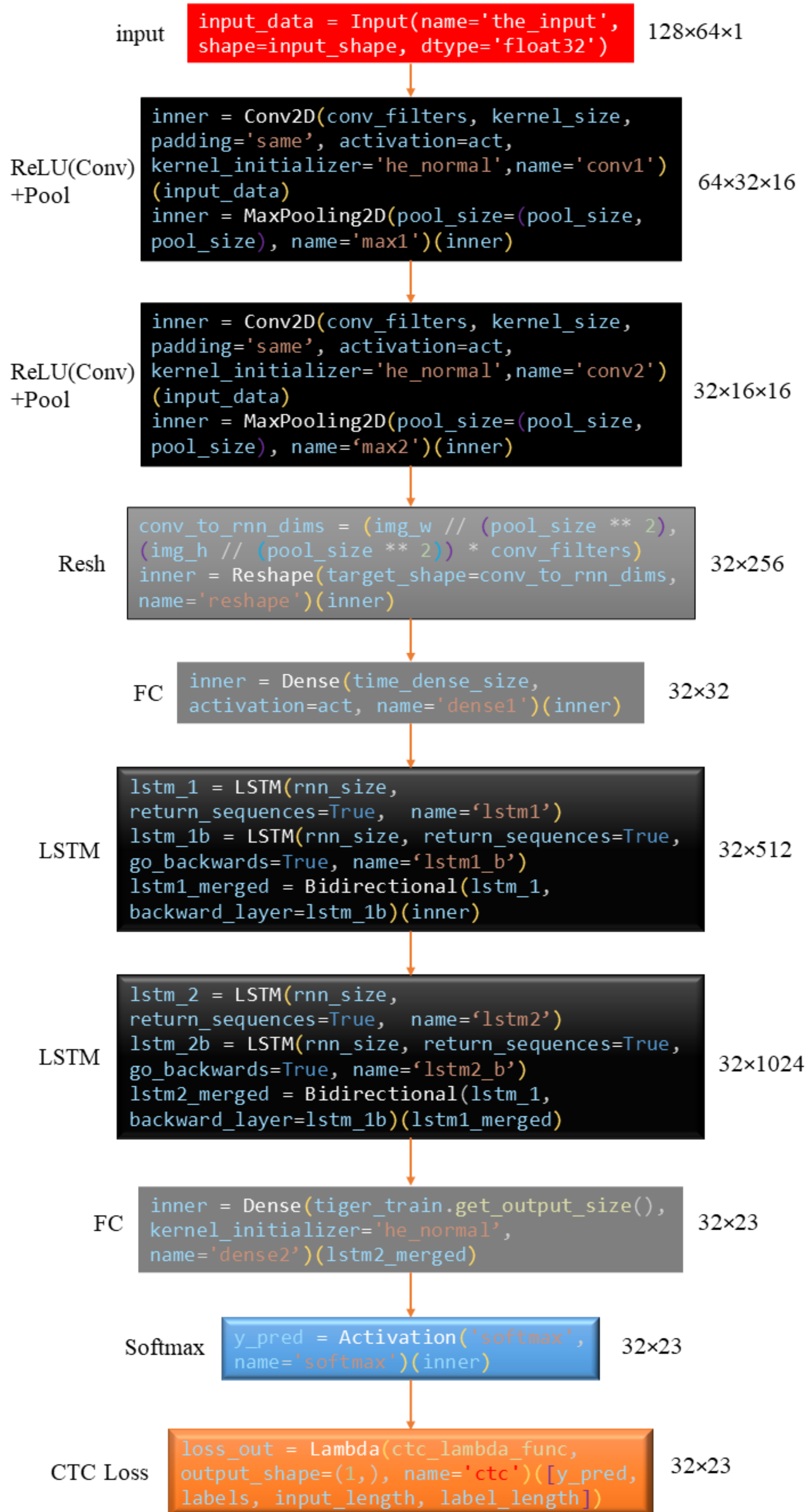

Рисунок 8 – Программная реализация СРНС

Как видно из схемы, сеть состоит из двух пар слоёв свёртки и субдискретизации, пары двунаправленных LSTM слоёв, а также пары слоёв полносвязного персептрона.

Сначала вводим входные параметры сети:

```
# Input Parameters
img_h = 64 # высота изображения
img_w = 128 # ширина изображения
# Network parameters
conv_filters = 16 # число фильтров для свёртки
kernel_size = (3, 3) # размер ядра
pool_size = 2 # размер окна субдискретизации
time_dense_size = 32 # размер тензора для Dense слоя
rnn_size = 512 # размер рекуррентного слоя
```

На вход сети подается тензор в виде двумерного массива размером 128×64 пикселов:

```
input_shape = (img_w, img_h, 1)
input_data = Input(name='the_input', shape=input_shape, dtype='float32'),
```

где `Input`– функция для создания экземпляра тензора Keras.

В третьей размерности указано, что число каналов равно 1.

Далее идёт первый свёрточный слой. Для него было выбрано ядро размером 3×3, число фильтров – 16:

```
act = 'relu'
inner = Conv2D(conv_filters, kernel_size, padding='same',
                    activation=act, kernel_initializer='he_normal',
                    name='conv1')(input_data)
```

где `'relu'` – функция активации.

Стоит заметить, что в сети используется выпрямленная линейная функция активации (Rectified Linear Unit, ReLU), так как она лишена ресурсоёмких операций, а также отсекает ненужные детали [10]. Также стоит отметить, что эта функция активации будет использоваться во всей первой половине сети.

Чтобы соблюсти стандартную архитектуру свёрточной сети, осталось только добавить слой субдискретизации с размером окна 2×2:

```
inner = MaxPooling2D(pool_size=(pool_size, pool_size), name='max1')(inner).
```

В итоге, после первой пары слоёв выходит тензор размером 64×32×16.

Следом создаётся вторая пара слоёв, в которой слой свёртки с ядром 3×3 и числом фильтров равным 16. В конце выходит тензор размером 32×16×16.

Как правило, в глубоких нейронных сетях за сверточными слоями следуют полносвязные, задача которых состоит в том, чтобы «собрать вместе» все признаки из фильтров и, собственно, перевести их в рекуррентные слои. Но для начала нам нужно из трёхмерного тензора сделать плоский. Для этого будет использована функция `Reshape()`:

```
conv_to_rnn_dims = (img_w // (pool_size ** 2), (img_h // (pool_size ** 2)) * conv_filters)
inner = Reshape(target_shape = conv_to_rnn_dims, name = 'reshape')(inner).
```

Получаем отформатированный тензор размером 32×256.

Добавляем полносвязный слой из 32 нейронов:

```
inner = Dense(time_dense_size, activation=act, name='dense1')(inner).
```

На выходе получается тензор размером 3232.

Свёрточная часть нейросети закончилась, далее идёт рекуррентная часть, состоящая из двух слоёв двунаправленных LSTM, которые добавляют поддержку переноса информации через многие интервалы времени.

Получившийся тензор признаков подается в первый двунаправленный LSTM слой:

```
lstm_1 = LSTM(rnn_size, return_sequences=True,  name='lstm1')
lstm_1b = LSTM(rnn_size, return_sequences=True, go_backwards=True, name='lstm1_b')
lstm1_merged = Bidirectional(lstm_1, backward_layer= lstm_1b)(inner),
```

где `lstm_1` обрабатывает входную последовательность в прямом направлении, `lstm_1b` – в обратном (`go_backwards=True`), а `lstm1_merged` объединяет полученные представления. Слой `Bidirectional` создает второй, отдельный экземпляр этого рекуррентного слоя и использует один экземпляр для обработки входных последовательностей в прямом порядке, а другой – в обратном. Второй двунаправленный слой задается аналогично.

После прохождения рекуррентных слоёв получается тензор размером 32×1024 (так как каждый слой содержит 512*2 LSTM ячеек), который, наконец, подаётся в последний полносвязный слой из 23 нейронов:

```
inner = Dense(tiger_train.get_output_size(), kernel_initializer='he_normal', name='dense2')(gru2_merged)
y_pred = Activation('softmax', name='softmax')(inner).
```

Было принято решение использовать функцию активации `softmax`, благодаря которой сеть будет выводить распределение вероятностей по 23 разным классам. Это значит, что для каждого из 32 входных образцов сеть будет возвращать 23-мерный вектор. Вывод сети будет декодирован в последовательность символов с помощью функции, представленной на рисунке 9.

```
def decode_batch(out):
    ret = []
    for j in range(out.shape[0]):
        out_best = list(np.argmax(out[j, 2:], 1))
        out_best = [k for k, g in itertools.groupby(out_best)]
        outstr = ''
        for c in out_best:
            if c < len(letters):
                outstr += letters[c]
        ret.append(outstr)
    return ret
letters=['0' ,'1' ,'2','3' ,'4' ,'5', '6', '7', '8','9', 'A', 'B', 'C', 'E' ,'H','K', 'M','O', 'P', 'T', 'X', 'Y']
```

Рисунок 9 – Функция декодирования вывода сети

В итоге эта сеть должна предсказать вероятность появления каждого из 23 заданных символов.

4.3.2 Обучение сети

Обучение сети происходило с помощью набора данных anpr_ocr, составленного сервисом Supervisely [11]. Этот набор составлен из искусственно созданных чёрно-белых случайных изображений номеров. Каждое изображение имеет размер 128×64 пикселов. Для данного набора была создана специальная функция-генератор данных, представленная на рисунке 10, которая преобразует изображения и кодирует последовательность символов на ней в данные, которые будет получать сеть.

```python
class TextImageGenerator:

    def __init__(self,
                 dirpath,
                 img_w, img_h,
                 batch_size,
                 downsample_factor,
                 max_text_len=9):

        self.img_h = img_h
        self.img_w = img_w
        self.batch_size = batch_size
        self.max_text_len = max_text_len
        self.downsample_factor = downsample_factor

        img_dirpath = join(dirpath, 'img')
        ann_dirpath = join(dirpath, 'ann')
        self.samples = []
        for filename in os.listdir(img_dirpath):
            name, ext = os.path.splitext(filename)
            if ext == '.png':
                img_filepath = join(img_dirpath, filename)
                json_filepath = join(ann_dirpath, name + '.json')
                description = json.load(open(json_filepath, 'r'))['description']
                if is_valid_str(description) and len(description)>0 and len(description)<10:

                    self.samples.append([img_filepath, description])

        self.n = len(self.samples)
        self.indexes = list(range(self.n))
        self.cur_index = 0

    def build_data(self):
        self.imgs = np.zeros((self.n, self.img_h, self.img_w))
        self.texts = []
        for i, (img_filepath, text) in enumerate(self.samples):
            img = cv2.imread(img_filepath)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            img = cv2.resize(img, (self.img_w, self.img_h))
            img = img.astype(np.float32)
            img /= 255
            # width and height are backwards from typical Keras convention
            # because width is the time dimension when it gets fed into the RNN
            self.imgs[i, :, :] = img
            self.texts.append(text)

    def get_output_size(self):
        return len(letters) + 1

    def next_sample(self):
        self.cur_index += 1
        if self.cur_index >= self.n:
            self.cur_index = 0
            random.shuffle(self.indexes)
        return self.imgs[self.indexes[self.cur_index]], self.texts[self.indexes[self.cur_index]]

    def next_batch(self):
        while True:
            # width and height are backwards from typical Keras convention
            # because width is the time dimension when it gets fed into the RNN
            if K.image_data_format() == 'channels_first':
                X_data = np.ones([self.batch_size, 1, self.img_w, self.img_h])
            else:
                X_data = np.ones([self.batch_size, self.img_w, self.img_h, 1])
            Y_data = np.ones([self.batch_size, self.max_text_len])*22
            input_length = np.ones((self.batch_size, 1)) * (self.img_w // self.downsample_factor - 2)
            label_length = np.zeros((self.batch_size, 1))
            source_str = []

            for i in range(self.batch_size):
                img, text = self.next_sample()
                img = img.T
                if K.image_data_format() == 'channels_first':
                    img = np.expand_dims(img, 0)
                else:
                    img = np.expand_dims(img, -1)
                X_data[i] = img
                aaaa=text_to_labels(text)
                laaaa=len(aaaa)
                Y_data[i,:laaaa] = aaaa
                source_str.append(text)
                label_length[i] = len(text)

            inputs = {
                'the_input': X_data,
                'the_labels': Y_data,
                'input_length': input_length,
                'label_length': label_length,
                #'source_str': source_str
            }
            outputs = {'ctc': np.zeros([self.batch_size])}
            yield (inputs, outputs)
```

Рисунок 10 – Функция-генератор данных

Результат работы функции-генератора представлен на рисунке 11. На нём изображено в каком виде данные будут поданы в сеть, а именно: сам вид изображения, как будут закодированы символы (метки), размер входных данных и длина последовательности символов.

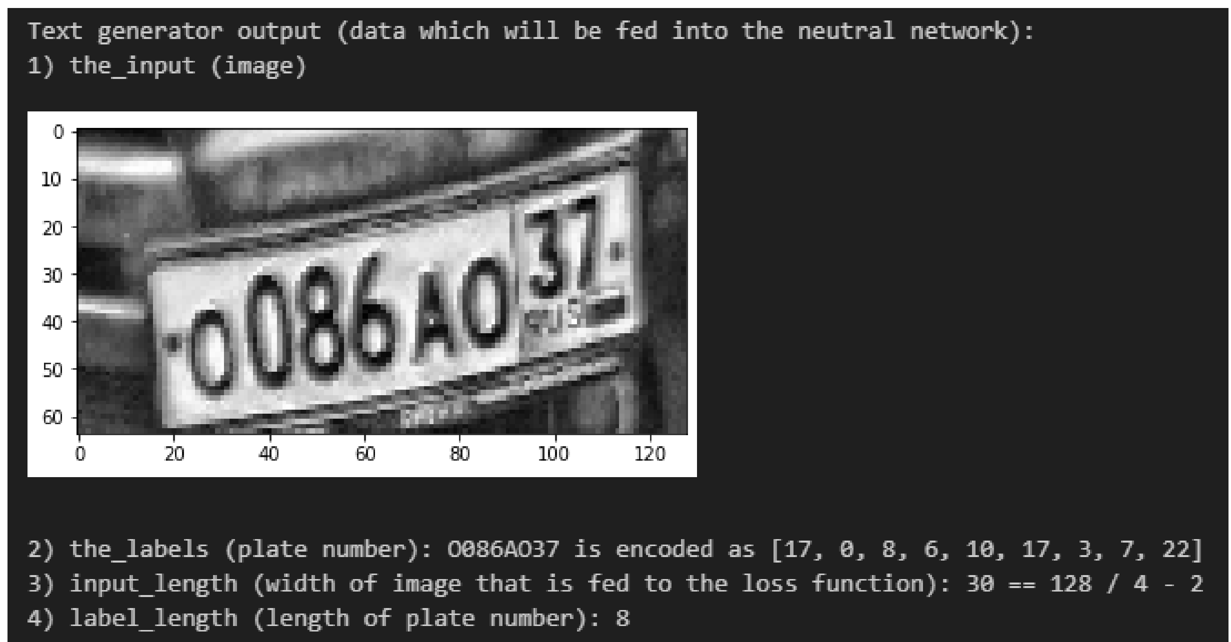

Рисунок 11 – Вывод функции-генератора

Для того чтобы обучить модель, нужно также зафиксировать некий способ оценки качества предсказаний (именно эту оценку мы и будем в конечном счете оптимизировать). Опишем в терминах Keras функцию потерь:

```
loss_out = Lambda(ctc_lambda_func, output_shape=(1,), name='ctc')([y_pred, labels, input_length, label_length]),
```

где `ctc_lambda_func` – функция потери:

```
def ctc_lambda_func(args):
    y_pred, labels, input_length, label_length = args
    y_pred = y_pred[:, 2:, :]
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length).
```

В качестве функции потери была использована ассоциативная временная классификация (Connectionist Temporal Classification, CTC) [12], которая вычисляется по формуле:

$$P_{CTC}(Y|X) = \sum_{A \in A_{X,Y}} \prod_{t=1}^{T} p_t(a_t|X) \quad (19)$$

где

t – временной шаг;

X – входная последовательность;

Y – выходная последовательность;

p – вероятность.

Для обучения модели использован алгоритм оптимизации Adam – адаптивный вариант градиентного спуска, являющийся модификацией Adagrad, но использует сглаженные версии среднего и среднеквадратичного градиентов [13]:

$$m_t = \beta_1 m + (1 - \beta_1) g_t \quad (20)$$

$$v_t = \beta_2 m + (1 - \beta_1) g_t^2 \quad (21)$$

$$u_t = \frac{\eta}{\sqrt{v + \varepsilon}} \quad (22)$$

```
sgd=Adam()
model.compile(loss={'ctc': lambda y_true, y_pred: y_pred}, optimizer=sgd).
```

```
Model: "model_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       [(None, 128, 64, 1)]      0
_________________________________________________________________
conv1 (Conv2D)               (None, 128, 64, 16)       160
_________________________________________________________________
max1 (MaxPooling2D)          (None, 64, 32, 16)        0
_________________________________________________________________
conv2 (Conv2D)               (None, 64, 32, 16)        2320
_________________________________________________________________
max2 (MaxPooling2D)          (None, 32, 16, 16)        0
_________________________________________________________________
reshape (Reshape)            (None, 32, 256)           0
_________________________________________________________________
dense1 (Dense)               (None, 32, 32)            8224
_________________________________________________________________
bidirectional_6 (Bidirection (None, 32, 1024)          2232320
_________________________________________________________________
bidirectional_7 (Bidirection (None, 32, 1024)          6295552
_________________________________________________________________
dense2 (Dense)               (None, 32, 23)            23575
_________________________________________________________________
softmax (Activation)         (None, 32, 23)            0
...
Trainable params: 8,562,151
Non-trainable params: 0
__________________________________________________________________________________________
Epoch 1/1
10298/10298 [==============================] - 1219s 118ms/step - loss: 0.3557 - val_loss: 1.7434e-04
```

Рисунок 12 – Обучение модели

Осталось только запустить обучение и дождаться результата. Ход обучения представлен на рисунке 12.

Всего была задана 1 эпоха. Обучение прошло всего за 1219 секунд и общие потери составляли около 0.00017.

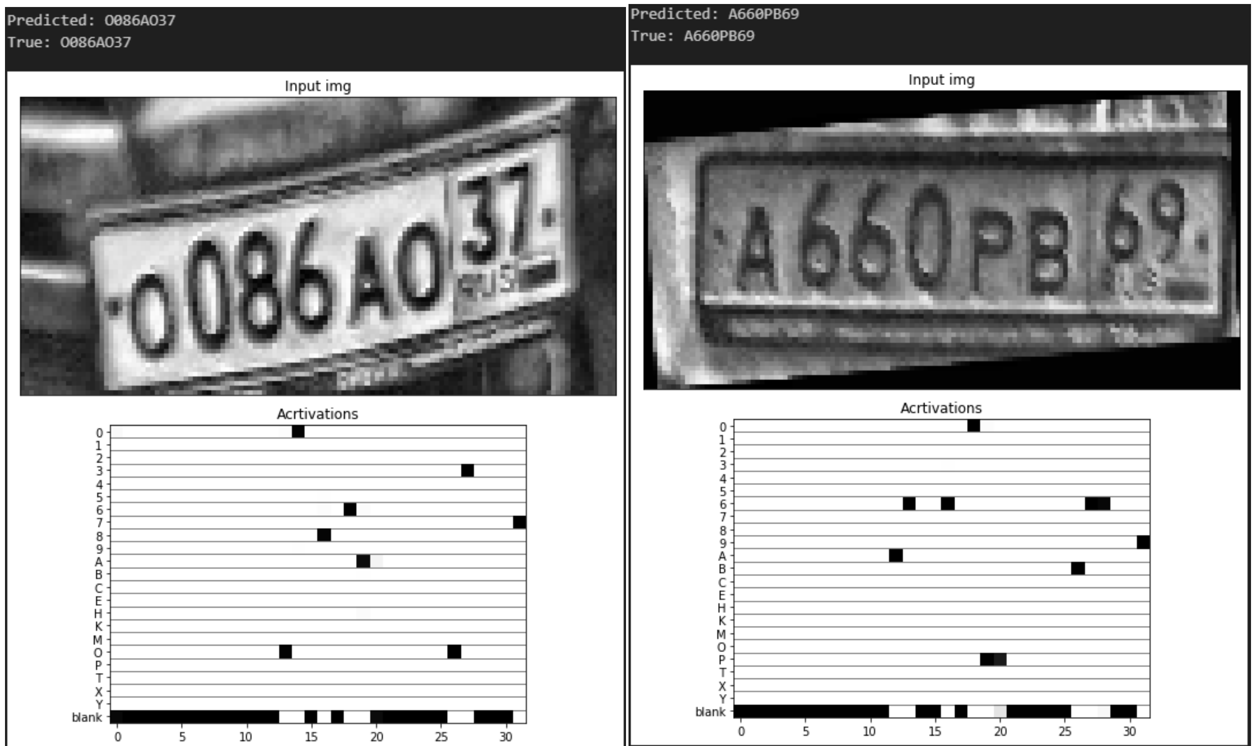

Рисунок 13 – Результат тестирования модели

После обучения модель можно проверить на тестовой выборке. Результат работы показан на рисунке 13.

Из рисунка видно, что модель достаточно чётко определила последовательность символов на номерах. Также на рисунке продемонстрировано поведение сети: строки матрицы соответствуют всем символам, которые используются в номерах, а столбцы соответствуют шагам сети.

4.4 Распознавание номера

4.4.1 Выбор модели для выделения номера

Для системы распознавания номеров будет использована уже готовая модель SSD ResNet50 V1 FPN 1024x1024. Модель была дотренирована с помощью набора данных о парковке в китайском городе CCPD [14]. Результат работы модели продемонстрирован на рисунке 14.

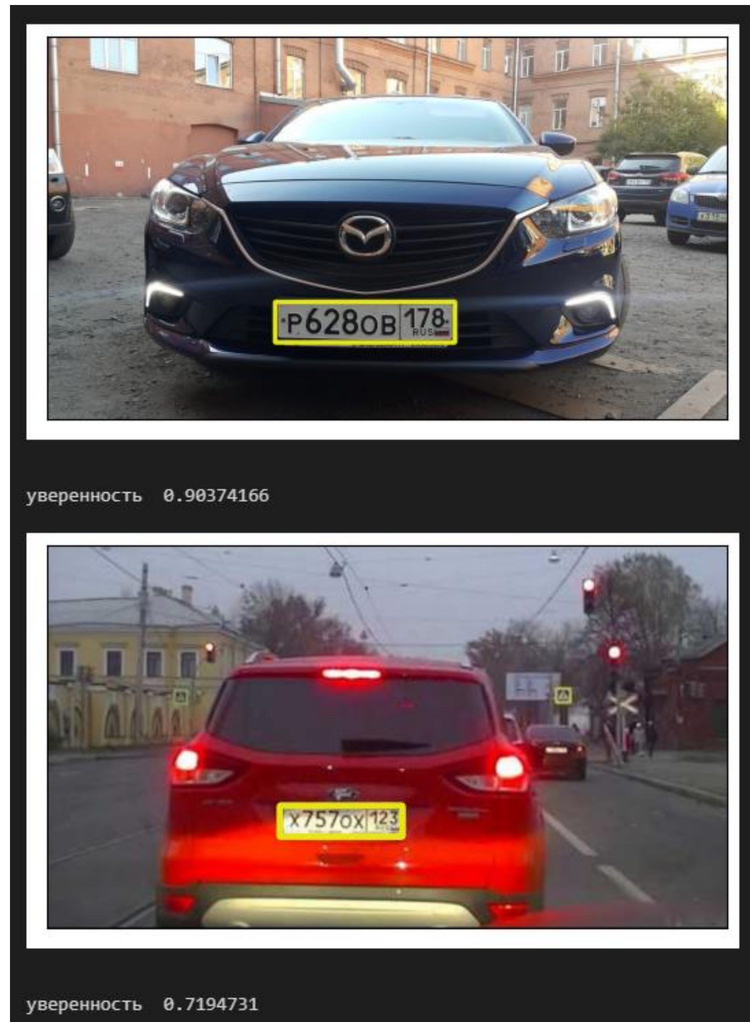

Рисунок 14 – Результат тестового запуска

4.4.2 Подготовка изображения

Прежде чем распознавать символы из выделенной области, нужно предварительно обработать изображение для более точного результата. Обработка изображений будет производиться с помощью библиотеки OpenCV и Scikit-Image.

Обработку изображения можно представить следующей последовательностью этапов:

1) с помощью модели, описанной выше, оставляется только номер. Результат этого этапа продемонстрирован на рисунке 15;

2) изображение выравнивается с помощью преобразования Хафа. Результат этого этапа продемонстрирован на рисунке 16;

3) увеличивается контрастность изображения. Результат этого этапа продемонстрирован на рисунке 17;

4) изображение переводится из цветного в чёрно-белое. Результат этого этапа продемонстрирован на рисунке 18;

5) с помощью двустороннего фильтра убирается лишний шум. Результат этого этапа продемонстрирован на рисунке 19.

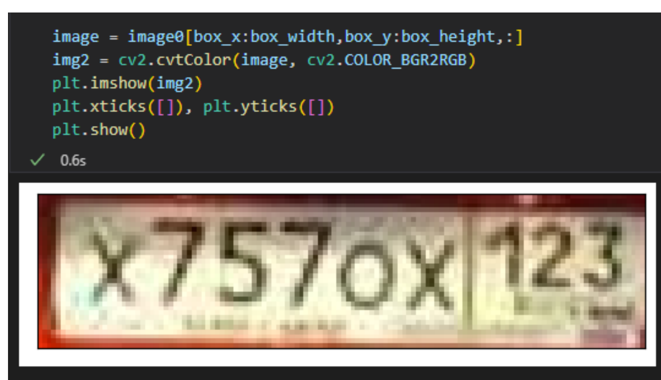

Рисунок 15 – Обрезка номерной пластины

Рисунок 16 – Выравнивание изображения

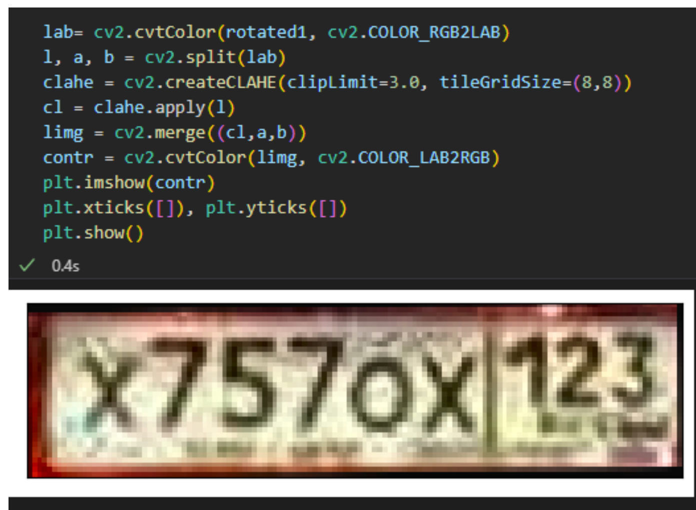

Рисунок 17 – Увеличение контрастности

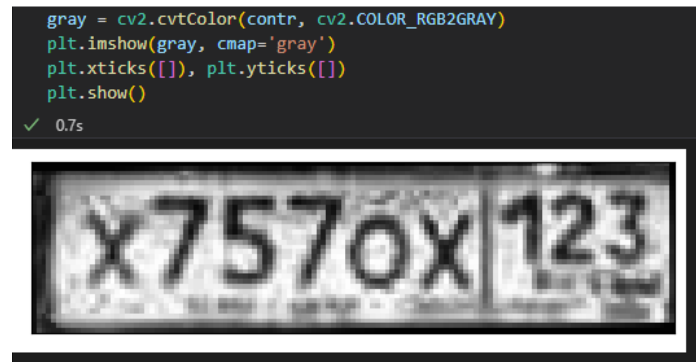

Рисунок 18 – Перевод из цветного в чёрно-белое

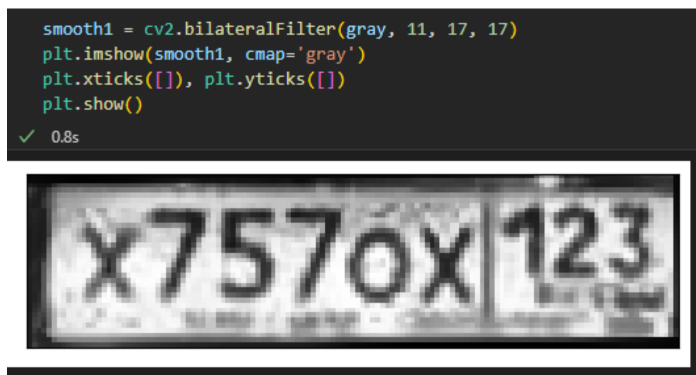

Рисунок 19 – Избавление от лишнего шума

4.4.3 Распознавание символов

Последним этапом в распознавании регистрационного номера – это фактическое считывание информации о номере с сегментированного изображения. Для распознавания символов из изображения использовалась полученная в ходе работы RCNN

Результат выполнения представлен на рисунке 20.

```
paths='./model1_nomer.tflite'
interpreter = tf.lite.Interpreter(model_path=paths)
interpreter.allocate_tensors()
# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
img = smooth1
img = cv2.resize(img, (128,64))
img = img.astype(np.float32)
img /= 255

img1=img.T
img1.shape
X_data1=np.float32(img1.reshape(1,128, 64,1))
input_index = (interpreter.get_input_details()[0]['index'])
interpreter.set_tensor(input_details[0]['index'], X_data1)

interpreter.invoke()

net_out_value = interpreter.get_tensor(output_details[0]['index'])
pred_texts = decode_batch(net_out_value)
print('Номер автомобиля:', pred_texts[0])
✓ 0.0s

Номер автомобиля: X757OX123
```

Рисунок 20 – Результат работы программы

По итогу данная программа была протестирована на 10 изображениях, вероятность обнаружения автомобильного номера составила 100%, а точность распознавания символов так же составила 100%

ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы был исследован метод, позволяющий выполнить распознавание регистрационных автомобильных номеров на изображении. Рассмотрены библиотеки, которые делают это возможным, а также на основе полученной информации реализован подход к распознаванию номеров и символов (рекуррентная свёрточная нейронная сеть). Были выявлены преимущества и недостатки данного алгоритма, а также проведено тестирование из данных, полученных в условиях реального транспортного движения.

Несмотря на то, что на сегодняшний день существует достаточно большое количество систем, основанных на свёрточных и рекуррентных нейронных сетях, данная область не прекращает развиваться и охватывает всё больше сфер человеческой деятельности, что позволяет ей быть востребованной в течение долгого периода времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Convolutional neural network // Wikipedia, the free encyclopedia: [сайт]. – URL: https://en.wikipedia.org/wiki/Convolutional_neural_network (дата обращения: 22.04.2023)

2. HybridTech: Свёрточная нейронная сеть, часть 1: структура, топология, функции активации и обучающее множество // Habr: [сайт]. – URL: https://habr.com/ru/post/348000/ (дата обращения: 22.04.2023)

3. Николенко, С. Глубокое обучение / С. Николенко, А. Кадурин, Е. Архангельская. – Санкт-Петербург: Питер, 2018. – 480 с. – ISBN 978-5-496-02536-2.

4. Chollet, F. Deep Learning with Python / F. Chollet. – New York: Manning Publications, 2017. – 384 с.

5. Recurrent neural network // Wikipedia, the free encyclopedia: [сайт]. – URL: https://en.wikipedia.org/wiki/Recurrent_neural_network (дата обращения: 22.04.2023)

6. Greff K. LSTM: A Search Space Odyssey / Greff K., Srivastava R. K., Koutnik J., Steunebrink B.R., Schmidhuber J. // arXiv, 2015. http://arxiv.org/abs/1503.04069

7. Cho K. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches / Cho K., van Merrienboen B., Bahdanau D., Bengio Y. // arXiv, 2014. http://arxiv.org/abs/1409.1259

8. Шакирьянов, Э. Д. Компьютерное зрение на Python. Первые шаги / Э. Д. Шакирьянов. – Москва: Лаборатория знаний, 2021. – 163 с. – ISBN 978-5-00101-944-2.

9. Постолит, А. В. Основы искусственного интеллекта в примерах на Python. Самоучитель / А. В. Постолит. – Санкт-Петербург: БХВ-Петербург, 2021. – 448 с. – ISBN 978-5-9775-6765-7

10. Koul, A. Practical Deep Learning for Cloud, Mobile and Edge / A. Koul, S. Ganju, M. Kasam. – Sebastopol: O`Reilly Media, Inc., 2020. – 958 с.

11. Supervisely: unified OS/Platform for computer vision: сайт – Tallinn. – URL: https://supervise.ly/ (дата обращения: 15.04.2023)

12. Sequence Modeling With CTC // Distill: [сайт]. – URL: https://distill.pub/2017/ctc/ (дата обращения: 05.05.2023)

13. Kingma, D. Adam: A Method for Stochastic Optimization / D. Kingma, J. Ba // 3rd International Conference for Learning Representations (San Diego; 2014). – San Diego, 2015 – 15 с.

14. CCPD (Chinese City Parking Dataset, ECCV) // Github: [сайт]. – URL: https://github.com/detectRecog/CCPD (дата обращения: 20.04.2023)