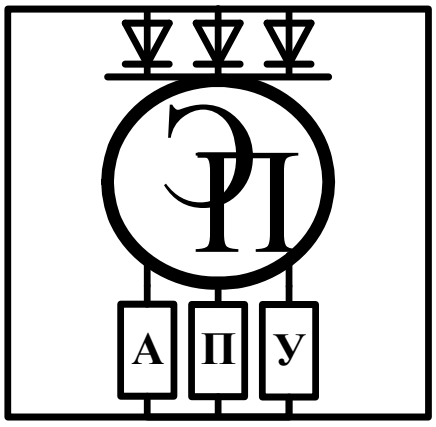
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Электропривод и автоматизация промышленных установок»

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ МЕХАТРОННЫХ И РОБОТОТЕХНИЧЕСКИХ СИСТЕМ

Методические рекомендации к практическим занятиям для студентов направления подготовки 15.03.06 «Мехатроника и робототехника» дневной формы обучения



Могилев 2023

УДК 621.865:004.45
ББК 3.816:32.9732
П75

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Электропривод и автоматизация промышленных установок» «2» мая 2023 г., протокол № 7

Составитель ст. преподаватель О. А. Капитонов

Рецензент Е. В. Ильюшина

Методические рекомендации предназначены студентов направления подготовки 15.03.06 «Мехатроника и робототехника» дневной формы обучения.

Учебное издание

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ МЕХАТРОННЫХ И РОБОТОТЕХНИЧЕСКИХ СИСТЕМ

Ответственный за выпуск	А. С. Коваль
Корректор	Т. А. Рыжикова
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 36 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение

Целью учебной дисциплины «Программное обеспечение мехатронных и робототехнических систем» является формирование специалистов, умеющих обоснованно и результативно применять существующие и осваивать новые методы разработки программного обеспечения для управления мехатронными и робототехническими системами.

Программное обеспечение современных роботов имеет, как правило, сложную многоуровневую структуру. Программное обеспечение верхнего уровня чаще всего работает под управлением операционной системы на базе ядра Linux. В настоящее время получила широкое распространение операционная система ROS, которая предоставляет возможность создания управляющих программ на языках программирования Python и C++. Программное обеспечение низкого уровня чаще всего составляется на языке программирования C и после преобразования в машинные коды запускается на микроконтроллерах, которые осуществляют сбор информации с датчиков, а также управление электроприводами и актуаторами робота.

Также в настоящее время широкое распространение в программном обеспечении роботов получили методы искусственного интеллекта, в частности, искусственные нейронные сети. Распознавание объектов на изображениях, распознавание голосовых команд и даже управление движением транспортного средства может осуществляться с помощью искусственных нейронных сетей.

В связи с вышеизложенным был разработан курс практических занятий, который помогает студентам ознакомиться с типовой структурой системы управления мобильным роботом, с программным обеспечением всех уровней, а также с особенностями применения методов искусственного интеллекта в программном обеспечении роботов, что является неотъемлемой частью процесса подготовки специалиста в области современной робототехники.

1 Общие сведения об оборудовании, используемом для проведения практических занятий

Общий вид мобильного робота, используемого в цикле практических занятий, представлен на рисунке 1.1. В данном случае это модель автомобиля с самоуправлением, которая может перемещаться в пределах выделенного трека, применяя информацию, поступающую с видеокамер, дальномеров и других датчиков.

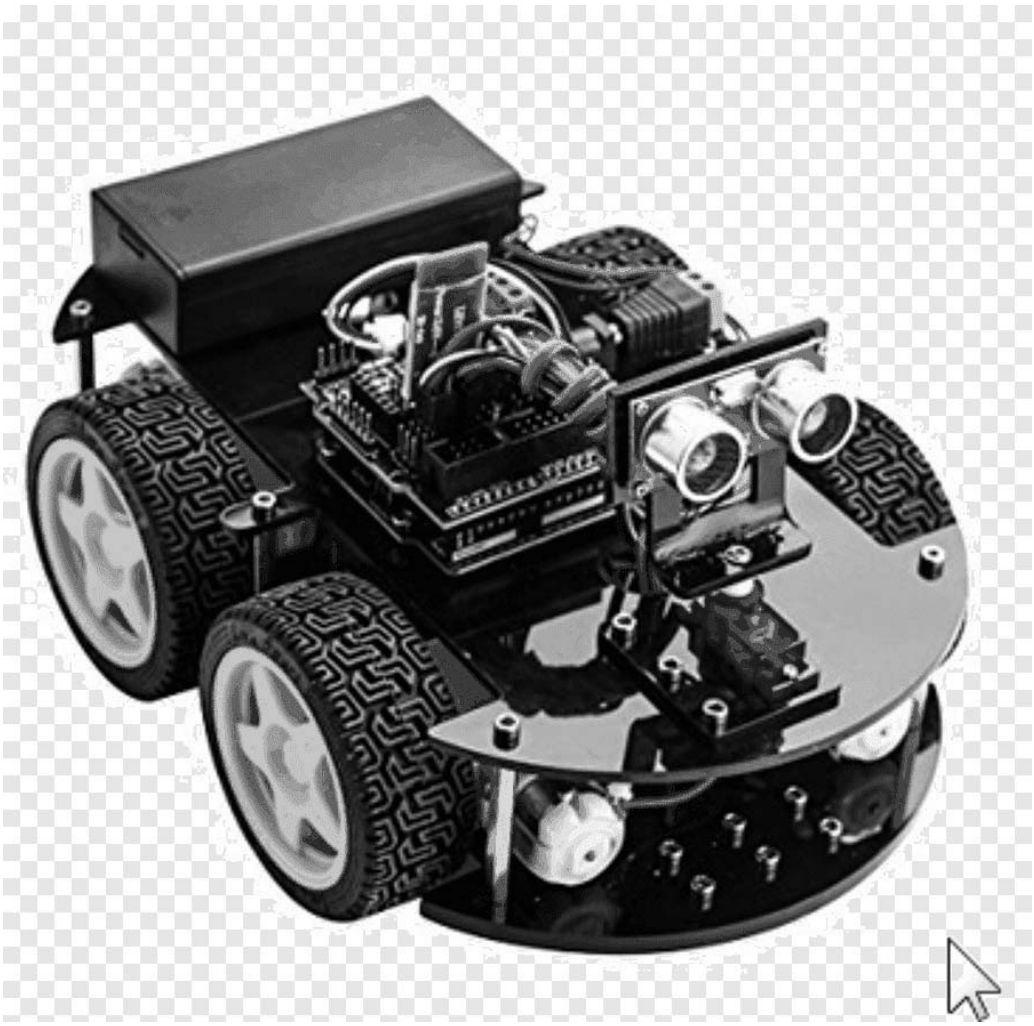


Рисунок 1.1 – Общий вид мобильного робота для практических занятий

Структурная схема системы управления мобильного робота приведена на рисунке 1.2.

Функциональная схема системы управления мобильного робота представлена на рисунке 1.3.

Рассматриваемый мобильный робот имеет привод на все колеса от электродвигателя постоянного тока, управляемого посредством широтно-импульсной модуляции. Направление движения изменяется за счет поворота передних колес, осуществляемого посредством сервопривода.

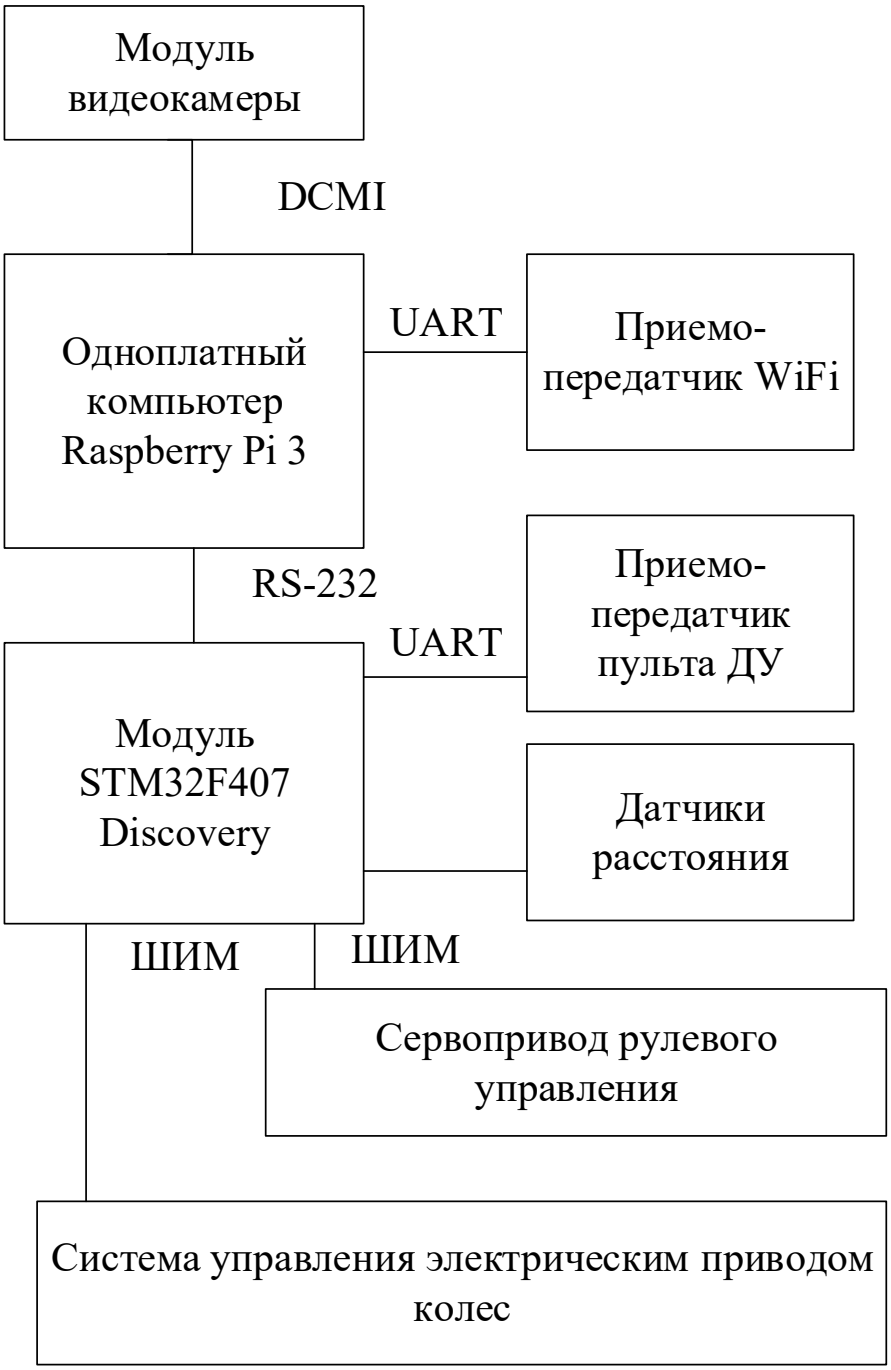


Рисунок 1.2 – Структурная схема системы управления мобильного робота

В соответствии со структурной схемой управление электроприводами робота осуществляет модуль STM32F407 Discovery, на котором реализован ПИД-регулятор скорости хода и управления поворотом в зависимости от сигнала задания. Также к указанному модулю подключен приемопередатчик пульта дистанционного управления, что позволяет управлять роботом дистанционно в режиме сбора информации или отладки программного обеспечения.

Модуль STM32F407 Discovery получает сигналы от ультразвуковых дальномеров. Данные сигналы могут быть использованы для обнаружения препятствий, возникающих на пути робота.

Рисунок 1.3 – Функциональная схема системы управления мобильного робота

Система управления верхнего уровня реализована на базе одноплатного компьютера RaspberryPI 3. Данный одноплатный компьютер принимает информацию с подключенной к нему видеокамеры. Затем осуществляется предварительная обработка указанной информации. Предварительная обработка заключается в выравнивании искаженного широкоугольным объективом изображения и применении различных фильтров для устранения шумов на изображении.

Полученное изображение используется для поиска препятствий, а также определения требуемых управляющих воздействий в соответствии с подходом Behavioral Cloning.

Поскольку вычислительная мощность одноплатного компьютера RaspberryPI 3 ограничена, в системе управления предусмотрена возможность

отправки по Wi-Fi изображения на более мощный персональный компьютер, а также прием результата обработки с достаточным для функционирования в режиме реального времени быстродействием.

Для вычисления управляющих воздействий используется сверточная искусственная нейронная сеть с одним нейроном в выходном слое и линейной функцией активации. Входом данной нейронной сети является изображение, поступающее с видеокамеры, а выходом – угол поворота передних колес робота. Сигнал управления скоростью робота может быть рассчитан исходя из радиуса окружности, по которой поворачивает робот.

Поиск препятствий па изображении может быть осуществлен в самом простом случае методом скользящих окон или может быть использован какой-либо из продвинутых методов SSD.

Заданием студентов на практических занятиях является сбор набора данных, а также написание и отладка программного обеспечения для обучения искусственных нейронных сетей и применения результатов данного обучения на практике.

2 Основные теоретические сведения

Для выполнения заданий практических работ студенту требуется представление о структуре и принципе функционирования сверточных искусственных нейронных сетей. Данные нейронные сети в настоящее время являются основным средством обнаружения объектов на изображениях.

Сверточная нейронная сеть – специальная архитектура искусственных нейронных сетей, предложенная Яном Лекуном в 1988 г. и нацеленная на эффективное распознавание изображений, входит в состав технологий глубокого обучения (англ. *deep learning*). Использует некоторые особенности зрительной коры, в которой были открыты так называемые простые клетки, реагирующие на прямые линии под разными углами, и сложные клетки, реакция которых связана с активацией определённого набора простых клеток. Таким образом, идея свёрточных нейронных сетей заключается в чередовании свёрточных слоёв (англ. *convolution layers*) и субдискретизирующих слоёв (англ. *subsampling layers* или *pooling layers*, слоёв подвыборки). Структура сети однонаправленная (без обратных связей), принципиально многослойная. Для обучения используются стандартные методы, чаще всего метод обратного распространения ошибки. Функция активации нейронов (передаточная функция) любая, по выбору исследователя.

Название архитектура сети получила из-за наличия операции свёртки, суть которой в том, что каждый фрагмент изображения умножается на матрицу (ядро) свёртки поэлементно, а результат суммируется и записывается в аналогичную позицию выходного изображения.

Типовая структура сверточной нейронной сети представлена на рисунке 2.1.

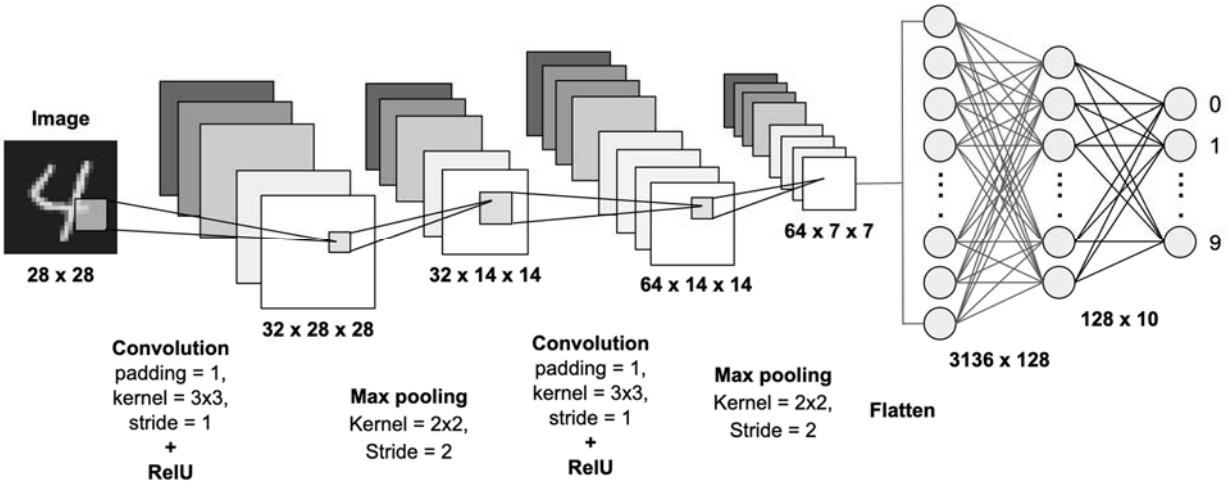


Рисунок 2.1 – Типовая структура сверточной нейронной сети

Работа свёрточной нейронной сети обычно интерпретируется как переход от конкретных особенностей изображения к более абстрактным деталям и далее к ещё более абстрактным деталям вплоть до выделения понятий высокого уровня. При этом сеть самонастраивается и вырабатывает сама необходимую иерархию абстрактных признаков (последовательности карт признаков), фильтруя маловажные детали и выделяя существенное.

Подобная интерпретация носит скорее метафорический или иллюстративный характер. Фактически «признаки», вырабатываемые сложной сетью, малопонятны и трудны для интерпретации настолько, что в практических системах не особенно рекомендуется пытаться понять содержания этих признаков или пытаться их «подправить», вместо этого рекомендуется усовершенствовать саму структуру и архитектуру сети, чтобы получить лучшие результаты Так, игнорирование системой каких-то существенных явлений может свидетельствовать о том, что либо не хватает данных для обучения, либо структура сети обладает недостатками и система не может выработать эффективных признаков для данных явлений.

В обычном перцептроне, который представляет собой полносвязную нейронную сеть, каждый нейрон связан со всеми нейронами предыдущего слоя, причём каждая связь имеет свой персональный весовой коэффициент. В свёрточной нейронной сети в *операции свёртки* используется лишь ограниченная матрица весов небольшого размера, которую «двигают» по всему обрабатываемому слою (в самом начале – непосредственно по входному изображению), формируя после каждого сдвига сигнал активации для нейрона следующего слоя с аналогичной позицией. То есть для различных нейронов выходного слоя используется одна и та же матрица весов, которую также называют *ядром свёртки.* Её интерпретируют как графическое кодирование какого-либо признака, например, наличие наклонной линии под определенным углом. Тогда следующий слой, получившийся в результате операции свёртки такой матрицей весов, показывает наличие данного признака в обрабатываемом слое и её координаты, формируя так называемую карту признаков

(англ. *featuremap)*. Естественно, в свёрточной нейронной сети набор весов не один, а целая гамма, кодирующая элементы изображения (например, линии и дуги под разными углами). При этом такие ядра свёртки не закладываются исследователем заранее, а формируются самостоятельно путём обучения сети классическим методом обратного распространения ошибки. Проход каждым набором весов формирует свой собственный экземпляр карты признаков, делая нейронную сеть многоканальной (много независимых карт признаков на одном слое). Также следует отметить, что при переборе слоя матрицей весов её передвигают обычно не на полный шаг (размер этой матрицы), а на небольшое расстояние. Так, например, при размерности матрицы весов 5 × 5 её сдвигают на один или два нейрона (пикселя) вместо пяти, чтобы не «перешагнуть» искомый признак.

Операция субдискретизации (англ. *subsampling*, англ. *pooling*, также переводимая как «операция подвыборки» или операция объединения) выполняет уменьшение размерности сформированных карт признаков. В данной архитектуре сети считается, что информация о факте наличия искомого признака важнее точного знания ею координат, поэтому из нескольких соседних нейронов карты признаков выбирается максимальный и принимается за один нейрон уплотнённой карты признаков меньшей размерности. За счёт данной операции, помимо ускорения дальнейших вычислений, сеть становится более инвариантной к масштабу входного изображения.

Рассмотрим типовую структуру свёрточной нейронной сети более подробно. Сеть состоит из большого количества слоев. После начального слоя (входного изображения) сигнал проходит серию свёрточных слоев, в которых чередуется собственно свёртка и субдискретизация (пулинг). Чередование слоев позволяет составлять «карты признаков» из карг признаков, на каждом следующем слое карта уменьшается в размере, но увеличивается количество каналов. На практике это означает способность распознавания сложных иерархий признаков. Обычно после прохождения нескольких слоев карта признаков вырождается в вектор или даже скаляр, но таких карт признаков становятся сотни. На выходе свёрточных слоёв сети дополнительно устанавливают несколько слоёв полносвязной нейронной сети (перцептрон), на вход которому подаются оконечные карты признаков.

Схема, поясняющая принцип формирования карт признаков, представлена на рисунке 2.2.

Нейроны слоя свёртки, преобразуемые по нескольким выходным каналам.

Слой свёртки (англ. *convolutional layer*) – это основной блок свёрточной нейронной сети. Слой свертки включает в себя для каждого канала свой фильтр, ядро свёртки которого обрабатывает предыдущий слой по фрагментам (суммируя результаты матричного произведения для каждого фрагмента). Весовые коэффициенты ядра свёртки (небольшой матрицы) неизвестны и устанавливаются в процессе обучения.

Особенностью свёрточного слоя является сравнительно небольшое количество параметров, устанавливаемое при обучении. Так, например, если исходное изображение имеет размерность 100 × 100 пикселей по трём каналам

(это значит 30000 входных нейронов), а свёрточный слой использует фильтры с ядром 3 × 3 пикселя с выходом на шесть каналов, тогда в процессе обучения определяется только девять весов ядра, однако по всем сочетаниям каналов, т. е. 9 × 3 × 6 = 162, в таком случае данный слой требует нахождения только 162 параметров, что существенно меньше количества искомых параметров полносвязной нейронной сети.

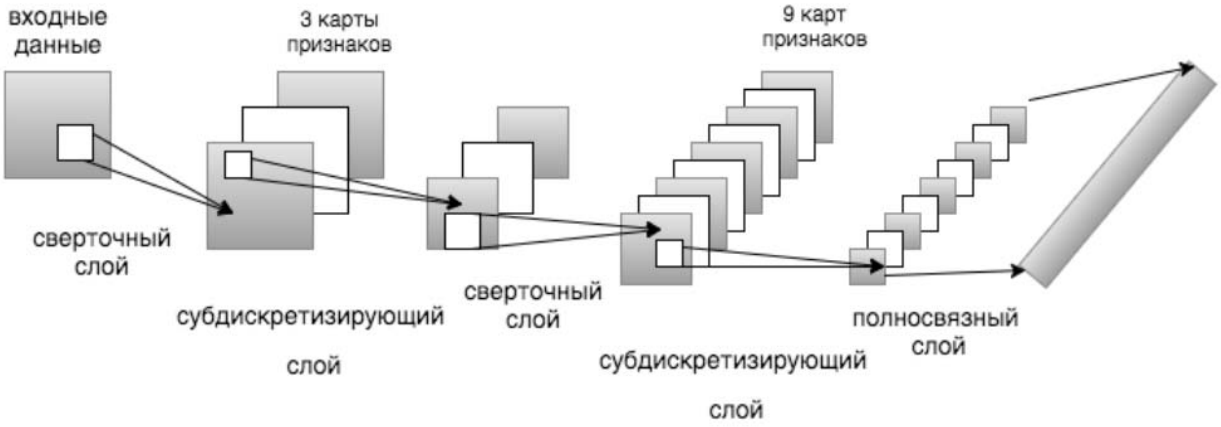


Рисунок 2.2 – Схема, поясняющая принцип формирования карт признаков

Слой активации.

Скалярный результат каждой свёртки попадает на функцию активации, которая представляет собой некую нелинейную функцию. Слой активации обычно логически объединяют со слоем свёртки (считают, что функция активации встроена в слой свёртки). Функция нелинейности может быть любой по выбору исследователя, традиционно для этого использовали функции типа гиперболического тангенса (th x,) или сигмоиды ($\sigma(x)$). Однако в 2000-х гг. была предложена и исследована новая функция активации ReLU (сокращение от англ. *rectified linear unit*), которая позволила существенно ускорить процесс обучения и одновременно упростить вычисления (за счёт простоты самой функции), что означает блок линейной ректификации, вычисляющий функцию. То есть, по сути, это операция отсечения отрицательной части скалярной величины. По состоянию на 2017 г. данная функция и её модификации (NoisyReLU, LeakyReLU и др.) являются наиболее часто используемыми функциями активации в глубоких нейросетях, в частности, в свёрточных.

Схема, поясняющая принцип субдискретизации, представлена на рисунке 2.3.

Пулинг с функцией максимума и фильтром 2 × 2 с шагом 2.

Слой пулинга (иначе подвыборки, субдискретизации) представляет собой нелинейное уплотнение карты признаков, при этом группа пикселей (обычно размером 2 х 2) уплотняется до одного пикселя, проходя нелинейное преобразование. Наиболее употребительна при этом функция максимума. Преобразования затрагивают непересекающиеся прямоугольники или квадраты, каждый из которых ужимается в один пиксель, при этом выбирается пиксель, имеющий максимальное значение. Операция пулинга позволяет существенно

уменьшить пространственный объём изображения. Пулинг интерпретируется так. Если на предыдущей операции свёртки уже были выявлены некоторые признаки, то для дальнейшей обработки настолько подробное изображение уже не нужно, и оно уплотняется до менее подробного. К тому же фильтрация уже ненужных деталей помогает не переобучаться. Слой пулинга, как правило, вставляется после слоя свёртки перед слоем следующей свёртки.

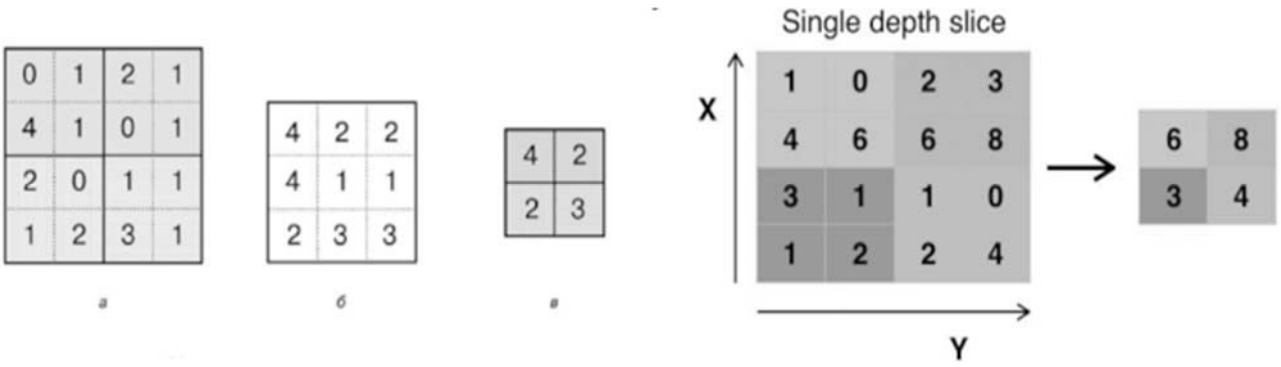


Рисунок 2.3 – Схема, поясняющая принцип субдискретизации

Кроме пулинга с функцией максимума, можно использовать и другие функции – например, среднего значения или L2-нормирования. Однако практика показала преимущества именно пулинга с функцией максимума, который включается в типовые системы.

В целях более агрессивного уменьшения размера получаемых представлений всё чаще находят распространение идеи использования меньших фильтров или полный отказ от слоев пулинга.

После нескольких прохождений свёртки изображения и уплотнения с помощью пулинга система перестраивается от конкретной сетки пикселей с высоким разрешением к более абстрактным картам признаков, как правило, на каждом следующем слое увеличивается число каналов и уменьшается размерность изображения в каждом канале. В конце концов остаётся большой набор каналов, хранящих небольшое число данных (даже один параметр), которые интерпретируются как самые абстрактные понятия, выявленные из исходного изображения.

Данные объединяются и передаются на обычную полносвязную нейронную сеть, которая тоже может состоять из нескольких слоёв. При этом полносвязные слои уже утрачивают пространственную структуру пикселей и обладают сравнительно небольшой размерностью (по отношению к количеству пикселей исходного изображения).

Наиболее простым и популярным способом обучения является метод обучения с учителем (на маркированных данных) – метод обратного распространения ошибки и его модификации. Но существует также ряд техник обучения свёрточной сети без учителя. Например, фильтры операции свёртки можно обучить отдельно и автономно, подавая на них вырезанные случайным образом кусочки исходных изображений обучающей выборки и применяя для них любой известный алгоритм обучения без учителя (например, автоассоциатор

или даже метод А-средних) – такая техника известна под названием *patch-based training*. Соответственно, следующий слой свёртки сети будет обучаться на кусочках от уже обученного первого слоя сети. Также можно скомбинировать свёрточную нейросеть с другими технологиями глубинного обучения. Например, сделать свёрточный автоассоциатор, свёрточную версию каскадных ограниченных машин Больцмана, обучающихся за счёт вероятностного математического аппарата, свёрточную версию разреженного кодирования (англ. *sparse coding*), названную *deconvolutional networks* («развертывающими» сетями).

Для улучшения работы сети, повышения её устойчивости и предотвращения переобучения применяется также исключение (дропаут) – метод тренировки подсети с выбрасыванием случайных одиночных нейронов.

Сверточные нейронные сети обладают следующими преимуществами:

– один из лучших алгоритмов по распознаванию и классификации изображений;

– по сравнению с полносвязной нейронной сетью (тина перцептрона) – гораздо меньшее количество настраиваемых весов, так как одно ядро весов используется целиком для всего изображения, вместо того, чтобы делать дня каждого пикселя входного изображения свои персональные весовые коэффициенты. Это подталкивает нейросеть при обучении к обобщению демонстрируемой информации, а не попиксельному запоминанию каждой показанной картинки в мириадах весовых коэффициентов, как это делает перцептрон;

– удобное распараллеливание вычислений, а, следовательно, возможность реализации алгоритмов работы и обучения сети на графических процессорах;

– относительная устойчивость к повороту и сдвигу распознаваемого изображения;

– обучение при помощи классического метода обратного распространения ошибки.

Недостатки свёрточных нейронных сетей заключаются в следующем.

Слишком много варьируемых параметров сети; непонятно, для какой задачи и вычислительной мощности какие нужны настройки. Так, к варьируемым параметрам можно отнести: количество слоёв; размерность ядра свёртки для каждого из слоев; количество ядер для каждого из слоёв, шаг сдвига ядра при обработке слоя; необходимость слоёв субдискретизации, степень уменьшения ими размерности; функция по уменьшению размерности (выбор максимума, среднего и т. п.); передаточная функция нейронов; наличие и параметры выходной полносвязной нейросети па выходе свёрточной. Все эти параметры значительно влияют на результат, но выбираются исследователями эмпирически. Существует несколько выверенных и прекрасно работающих конфигураций сетей, но не хватает рекомендаций, по которым нужно строить сеть для новой задачи.

3 Практическое занятие № 1. Подготовка набора данных для обучения, перекрестной проверки и тестирования искусственной нейронной сети

Задание

Собрать два набора данных для обучения искусственных нейронных сетей. Первый набор – для обнаружения определенного объекта на изображении, второй – для управления движением мобильного робота в соответствии с подходом Behavioral Cloning.

Сбор первого набора данных можно осуществлять с помощью любого мобильного устройства с цифровой камерой, однако рекомендуется использовать либо веб-камеру, либо камеру, предназначенную для одноплатного компьютера RaspberryPi.

В самом начале практических занятий студенты делятся на подгруппы по 3–4 человека для работы над индивидуальным заданием.

Распознаваемый объект задается преподавателем. Это может быть небольшой макет дорожного знака или просто предмет, который выделяется по цвету или каким-либо иным признакам.

Второй набор данных следует собирать на треке, границы которого определяются какими-либо визуально выделяющимися предметами также в соответствии с заданием преподавателя.

Для выполнения задания на практическом занятии № 1 студенту требуются некоторые знания библиотеки OpenCV. Библиотека является свободной библиотекой для компьютерного зрения и в данном случае может быть полезна для открытия изображений, их редактирования и сохранения с целью формирования набора данных для обучения искусственной нейронной сети. OpenCV – что мощная библиотека машинного зрения, которую часто применяют в роботах для распознавания объектов окружающего мира.

Перед тем как приступить непосредственно к функциям машинного зрения, следует ознакомиться со вспомогательными функциями, необходимыми для работы с видеопотоком. Попробуем получить кадры с видеокамеры и применим к ним различные преобразования: отражение, смена цветовой модели, размытие.

Все примеры программ написаны на языке python.

Если в ходе установки OpenCV было использовано виртуальное окружение, то перед запуском python-скриптов необходимо перейти в это окружение с помощью команд

```
$ source ~/.profile
$ workoncv
```

Получение кадров с камеры и вывод в окно осуществляется следующим образом: программа будет в бесконечном цикле получать кадры с камеры и отображать их в специальном окне.

Далее приведен пример реализации такого алгоритма:

```
import cv2
import video

if __name__ = __main__:
# создаем окно с именем result
cv2.namedWindow("result")

# создаем объект cap для захвата кадров с камеры
cap = video.create_capture(0)
while True:
# захватываем текущий кадр и кладем его в переменную img
flag, img = cap.read()
try:
# отображаем кадр в окне с именем result
cv2.imshow('result', img)
except:
cap.release()
raise

ch = cv2.waitKey(5)
if ch == 27:
break
cap.release()
cv2. destroyAllWindows()
```

Примечание – Функция create_capture имеет всего один аргумент, который отвечает за индекс камеры в данной системе. В случае, если камера всего одна, её индекс будет равен 0.

Python-скрипт запускается через консоль следующим образом:

```
$ python eye.py
```

Если в программе нет ошибок, откроется окно с видео. В предыдущей программе брали каждый кадр, полученный с камеры, и отображали его в неизменном виде. Теперь в разрыв между этими двумя операциями добавим третью – отражение кадра. За отражение кадра отвечает функция

```
flip (кадр, направление)
```

Здесь кадр – кадр видеопотока, который нужно отразить; направление – флаг, определяющий направление отражения: 0 – по горизонтали; 1 – по вертикали; –1 – по горизонтали и вертикали одновременно.

В действительности разные веб-камеры могут по-разному реагировать на вертикальное и горизонтальное отражение. Например, в используемой составителем Logitech C170 вертикаль и горизонталь перепутаны местами.

```
import cv2
import video

if __name__ == __main__:
# создаем окно с именем result
cv2.namedWindow("result")
# создаем объект cap для захвата кадров с камеры
cap = video.create_capture(0)

while True:
# захватываем текущий кадр и кладем его в переменную img
flag, img = cap.read()

try:
# отражаем кадр
img = cv2.flip(img,0)
# отображаем кадр в окне с именем result
cv2.imshow('result', img)
except:
cap.release()
raise

ch = cv2.waitKey(5)
if ch == 27:
break

cap.release()
cv2.destroyAllWindows()
```

Часто при обучении искусственных нейронных сетей полезным бывает изменить цветовую модель изображения.

Цветовая модель – это модель представления цвета каждой точки с помощью группы чисел. Некий цветовой код. Например, в RGB цвет определяется тремя компонентами: красный, зеленый и синий. Любая веб-камера передает кадры именно в такой модели. В HSV у каждой точки есть цветовой тон – H, насыщение – S и яркость – V

Некоторые функции машинного зрения проще реализовать в какой-то конкретной модели. Например, далее увидим, что распознавание цветовых пятен на картинке лучше работает в модели HSV.

Для преобразования цветовой модели используем функцию

cvtColor(кадр, модель)

где модель – код преобразования: COLOR_GRAY – в оттенки серого; COLOR_RGB2HSV – из RGB в HSV; COLOR_BGR2HSV – из BGR в HSV; COLOR_HSV2BGR – из HSV в BGR и т. д.

OpenCV поддерживает много моделей, но в рассматриваемом случае потребуются пока только эти три: RGB, HSV и GRAY.

Примечание – по умолчанию кадр с камеры приходит в модели BGR, а не RGB, поэтому для преобразования необходимо использовать флаг COLOR_BGR2HSV.

```
import cv2
import video
if __name__== __main__:
# создаем окно с именем result
cv2.namedWindow("result")
# создаем объект cap для захвата кадров с камеры
cap = video.create_capture(0)

while True:
# захватываем текущий кадр и кладем его в переменную img
flag, img = cap.read()

try:
# меняем цветовую модель на HSV
img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV )
# отображаем кадр в окне с именем result
cv2.imshow('result', img)
except:
cap.releaseO
raise

ch = cv2.waitK.ey(5)
if ch == 27:
break

cap.release()
cv2.destroyAllWindows()
```

Часто изображение с веб-камер страдает наличием большого количества шумов. Особенно это заметно, когда камере не хватает освещения. Для частичного решения данной проблемы применяют фильтр Гаусса или, другими словами, размытие по Гауссу (*Gaussian blur*). Функция имеет три аргумента:

GaussianB1ur (кадр, размер ядра, отклонение)

Размер ядра – список из двух чисел (x, y), которые задают размер ядра фильтра Гаусса по горизонтали и вертикали. Чем больше ядро, тем более размытым станет кадр; отклонение – стандартное отклонение по оси X.

```
import cv2
import video
if __name__ == __main__:
# создаем окно с именем result
cv2.namedWindow( "result")
# создаем объект cap для захвата кадров с камеры
cap = video.create_capture(0)
while True:
# захватываем текущий кадр и кладем его в переменную img
flag, img = cap.read()

try:
# размываем кадр
img = cv2.GaussianBlur(img, (5, 5), 2)
# отображаем кадр в окне с именем result
cv2.imshow('result', img)
except:
cap.relcase()
raise

ch = cv2.waitKey(5)

if ch == 27:
break
```

Полезными могут оказаться следующие функции библиотеки OpenCV:

cv2.imread(filename[, flags]). Данная функция производит считывание изображения, хранящегося в файле по заданному адресу;

cv2.imwrite(filename, img[, params]). Данная функция сохраняет изображение, хранящееся в img, в заданный файл по заданному пути.

Также следует вспомнить, что загруженные с помощью библиотеки OpenCV изображения в приложениях на python являются объектами библиотеки NumPy, что дает возможность применять к указанным изображениям множество функций библиотеки NumPy.

Подготовка набора изображений.

Для загрузки фотографий и Keras будем использовать генераторы изображений (ImageDataGenerator). Они позволяют автоматически загружать изображения с диска в память компьютера, преобразовывать их в вид, необходимый Keras, и передавать в модель для обучения. Также генераторы предоставляют возможность расширения данных (data augmentation).

Данные для генераторов изображений Keras нужно подготовить специальным образом, что делается достаточно просто: создается каталог, в котором будут подкаталоги по количеству классов объектов. В рассматриваемом случае таких классов два. Имена каталогов могут быть любыми. При обучении нейронной сети генератор изображений Keras будет читать картинки из обоих каталогов и автоматически готовить метки с правильными ответами, например, 0 – для первого класса и 1 – для второго.

Для обучения нейронной сети нужно три набора изображений:

1) набор данных для обучения;

2) набор данных для проверки (оценки качества обучения сети в процессе обучения);

3) набор данных для тестирования (оценки качества обучения сети после завершения обучения).

Каждый набор данных должен содержать подкаталоги с фотографиями изображений разных классов.

Есть три каталога: в каталоге train содержатся данные для обучения нейронной сети: в каталоге validation – данные для проверки, в каталоге test – данные для тестирования. В каждом из этих каталогов есть подкаталоги, в которых находятся фотографии каждого из классов соответственно.

Можно распределить фотографии по всем каталогом вручную, но у преподавателя можно взять скрипт в формате *.ipynb, который копирует 70 % изображений в каталог для обучения и по 15 % в каталоги для проверки или тестирования. Можно воспользоваться готовым скриптом, написать свой или распределить фотографии вручную.

Создание генераторов.

После того как три набора данных подготовлены, можно приступать к созданию генераторов в программе на Keras. Будет три генератора по количеству наборов данных: для обучения, проверки и тестирования. Полный текст программы можно найти в файле, который также имеется у преподавателя.

На первом этапе нужно создать ImageDataGenerator, который в общем виде задает работу генераторов. Это делается следующим образом:

```
datagen = ImageDataGenerator(rescale=l. / 255)
```

Здесь указываем, что генератор при загрузке изображения будет выполнять операцию rescale – делить значение каждого пиксела на 255. Таким образом, все входные данные будут в диапазоне от 0 до 1, что хорошо для обучения нейронной сети.

Теперь можем создавать три генератора. Генератор данных для обучения

```
train_generator = datagen.flow_from_directory( 'train',
target_size=( 150, 150),
batch_size=64,
class_mode='binary')
```

Метод flow_from_directory объекта datagen создает генератор, который читает данные из каталога на диске. Название каталога указывается в первом аргументе – train. Второй параметр target_size – размер, к которому преобразуются загружаемые с диска изображения. В примере все изображения будут приведены к размеру 150 × 150 пикселей. batch_size – размер мини-выборки, количество изображений, которое генератор читает с диска за один раз и передает нейронной сети для обучения.

Параметр classmode задает, каким образом генерируются правильные ответы для загруженных генератором изображений. Возможные варианты: binary, categorical и sparse. При class_mode = 'binary' используется бинарная классификация – есть всего два класса, они задаются одним битом. Именно такой тип классификации применяем, чтобы распознавать котов и собак. Если классов больше, чем два, то нужно использовать режим categorical, при котором правильные ответы задаются в формате one-hotencoding. Режим sparse тоже подходит для случая, когда классов больше двух. В этом режиме генератор формирует правильные ответы в виде целых чисел – номеров классов.

Таким образом, создали итератор traingenerator, который читает изображения из каталога train, преобразует их размер на 150 × 150 пикселей, и правильные ответы генерируются в бинарном виде. За одно обращение генератор выдает 64 изображения с правильными метками классов.

Подобным образом создаем генератор данных для проверки:

```
val_generator = datagen.flow_from_directory( 'validation',
target_size=( 150, 150), batch_size=64, class_mode = binary')
```

Параметры этого генератора такие же, как и генератора данных для обучения, кроме каталога, из которого берутся фотографии. Используется каталог validation, в котором лежат фотографии набора данных для проверки.

Генератор тестовых данных также отличается только каталогом с фотографиями:

```
testgenerator = datagen.flow_from_direetory( 'test',
target_size=(150, 150), batch_size=64 class _mode-binary')
```

Для решения поставленной задачи будем использовать сверточную нейронную сеть, которая определяется следующим образом:

```
model = Sequential ()
model.add(Conv2D(32, (3, 3), input_shape=(150, 150, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3. 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activaiion('relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(l))
model.add(Activation('sigmoid'))
```

Сеть включает три каскада свертки и подвыборки. Размер сверточных ядер 3 × 3, размер подвыборки 2 × 2, используется выбор максимального значения (maxpooling). Сверточная часть сети предназначена для выделения характерных признаков в изображении.

После сверточной части идет полносвязная часть нейронной сети, которая отвечает за классификацию. Для этой цели используются два полносвязных слоя. На первом слое 64 нейрона, функция активации полулинейная. Затем идет слой Dropout, который используется для уменьшения переобучения. Выходной полносвязный слой включает всего один нейрон, что соответствует задаче бинарной классификации. 0 на выходе из сети означает, что на фотографии кот, а 1 – собака. Функция активации на выходном слое сигмоидальная. Эта функция плавно меняет свое значение от 0 до 1, поэтому ее удобно использовать для бинарной классификации.

Используем бэкенд TensorFlow, формат хранения изображений channels_last. Для Theano нужно поменять входную размерность нейронной сети на input_shape=(3, 150, 150) (количество каналов в изображении на первом месте, а не на последнем).

Компилируем нейронную сеть:

```
model.compile(loss='binary',
              crossentropy optimizer = 'adam',
              metrics=['accuracy'])
```

В качестве функции ошибки используем binary_crossentropy, т. к. классификация бинарная. Если классов больше двух, то нужно использовать categorical_crossentropy. Оптимизатор Adam, метрика качества обучения – точность.

Обучение нейронной сети с использованием генераторов изображений.

Обучать нейронную сеть будем с помощью метода model.fit_generator.

```
# Размер мини-выборки
batch_size = 64
# Количество изображений для обучения
nb_train_samples = 17500
# Количество изображений для проверки
nb_validation_samples = 3750
```

```
# Количество изображений для тестирования
nb_testsamples = 3750
model, fit_generator(
        traingenerator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=30,
        validation_data=val_generator,
        validation_steps=nb_validation_samples // batch size)
```

Методу fit_generator передаем два генератора: train_generator с данными для обучения и val_generator с данными для проверки. Обучение выполняется в течение 30 эпох.

Генераторы в Keras устроены так, что могут выдавать изображения бесконечно. После того как изображения в каталоге закончатся, происходит переход в начало каталога и генератор начинает работать снова. Поэтому нужно указать, сколько будет обращений к генератору на каждой эпохе обучения. Для этого используются параметры steps_per_epoch (данные для обучения) и validation_steps (данные для проверки). За одно обращение генератор выдает не одно изображение, а несколько, в соответствии с размером его мини-выборки (batch_size). Чтобы рассчитать количество обращений к генератору, при котором сможем получить все изображения из набора данных по одному разу, делим количество изображений в наборе на размер мини-выборки.

Диагностический вывод при обучении модели у метода model.fit_generator такой же, как и у model_fit.

Точность на данных для обучения обозначена acc, а точность на проверочном наборе данных – val_acc. Видно, что при обучении аккуратность увеличивается как на обучающем, так и на проверочном наборе данных.

Проверка качества обучения нейронной сети.

Оценивать качество обучения сети также будем с помощью генератора. Для этого у модели есть метод evaluate_generator:

```
scores = model.evaluate_generator(test_generator, nb_test_samples // batch_size)
print("Точность на тестовых данных: %.2i%%" % (scores[l]*100))
```

В метод model.evaluate_generator передаем генератор test_generator с изображениями, которые нейронная сеть не видела в процессе обучения. Этот генератор также может работать бесконечно, поэтому вторым аргументом указываем, сколько раз нужно обратиться к генератору (количество изображений в тестовом наборе данных, деленное на размер мини-выборки).

Точность распознавания можно еще повысить, если использовать предварительно обученные нейронные сети.

Не забудьте сохранить обученную нейронную сеть для последующего использования!

4 Практическое занятие № 2. Обучение и тестирование искусственной нейронной сети в системе управления роботом

На практическом занятии № 2 студентам необходимо использовать наборы данных, полученные при выполнении задания на практическом занятии № 1 для обучения искусственных нейронных сетей. Эти нейронные сети будут применены для управления движением мобильного робота, а также для распознавания знаков либо препятствий, возникающих перед роботом.

Рассмотрим практический пример. Обучение любой модели в машинном обучении начинается с данных. Keras содержит внутри несколько обучающих дата сетов, но они уже приведены в удобную для работы форму и не позволяют продемонстрировать большинство возможностей Keras. Для начала следует разбить имеющиеся данные па набор для обучения и набор для проверки точности работы искусственной нейронной сети.

```
newsgroups_train = fetch_20newsgroups(subset='train')
newsgroups_test = fetch_20newsgroups(subset='test')
```

Keras содержит в себе инструменты для удобного препроцессинга текстов, картинок и временных рядов, иными словами, самых распространенных типов данных. Сегодня работаем с текстами, поэтому нужно разбить их на токены и привести в матричную форму.

```
tokenizer = Tokenizer(num_words=max words)
tokenizer.fit_on_texts(newsgroups_train[ "data" ])
x_train = tokenizer.texts_to_matrix(newsgroups_train["data"J, mode-binary')
x_test = tokenizer.texts_to_matrix(newsgroups_test["data"], mode-binary')
```

На выходе получились бинарные матрицы вот таких размеров:

```
x_train_shape: (11314, 1000)
x_test_shape: (7532, 1000)
```

Первое число количество документов в выборке, а второе – размер словаря (одна тысяча в этом примере).

Еще понадобится преобразовать метки классов к матричному виду для обучения с помощью кросс-энтронии. Для этого переведем номер класса в так называемый one-hot-вектор, т. е. вектор, состоящий из нулей и одной единицы:

```
y_train = keras.utils.to_calegorical(newsgroups_train["target"]. num_classes)
v_test = keras.utils.to_categorical(newsgroups_test|"target"], num_classes)
```

На выходе получим также бинарные матрицы вот таких размеров:

y_train shape: (11314, 20)
y_test_shape: (7532, 20)

Как видим, размеры этих матриц частично совпадают с матрицами данных (по первой координате – числу документов в обучающей и тестовой выборках), а частично – нет. По второй координате стоит число классов (20, как следует из названия датасета).

Модель в Keras можно описать двумя основными способами.

Первый способ – последовательное описание модели, например, следующим образом:

```
model = Sequential()
model.add(Dense(512, mput_shape=(max_words,)))
model.add(Activation('relu'))
model.add(Dropout( 0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Также можно использовать последовательность

```
model = Sequential([
Dense(512, input_shape=(max_words,))
Activation('relu'),
Dropout(0.5),
Dense(num_classes),
Activation('softmax')
])
```

Второй способ – использовать функциональное API для создания модели:

```
a = Input(shape=(max_words,))
b = Dense(512)(a)
b = Activation('relu')(b)
b = Dropout(0.5)(b)
b = Dense(num_classes)(b)
b = Aclivation('softmax')(b)
model = Model(inputs=a, outputs=b)
```

Класс Model (и унаследованный от него Sequential) имеет удобный интерфейс, позволяющий посмотреть, какие слои входят в модель – model.layers, входы – model.inputs и выходы – model.outputs.

Также очень удобный метод отображения и сохранения модели – model.to_yaml.

Это позволяет сохранять модели в читаемом для человека виде, а также восстанавливать модели искусственных нейронных сетей из следующего описания:

```
From keras.models import model_from_yaml
yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

Важно отметить, что модель, сохраненная в текстовом виде (кстати, возможно сохранение также и в JSON), не содержит весов. Для сохранения и загрузки весов используйте функции save_weights и load_weights соответственно.

Рассмотрим визуализацию моделей искусственных нейронных сетей. Keras имеет встроенную визуализацию для моделей

```
from keras.utils import plot_model
plot_model(model, to_file = 'model.png', show_shapes=True)
```

Представленный выше код сохранит под именем model.png изображение, пример которого приведен на рисунке 4.1.

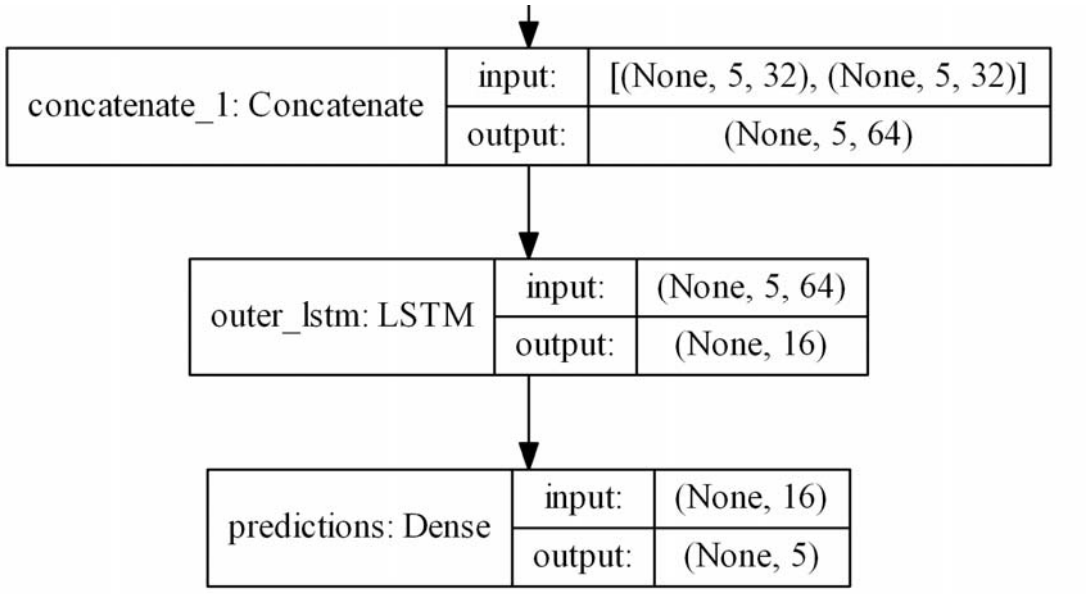


Рисунок 4.1 – Пример визуализации искусственной нейронной сети

Здесь дополнительно отображены размеры входов и выходов для слоев None, идущих первыми в кортеже размеров, – по размерности батча. Так как стоит None, то батч может быть произвольным.

Если захотите отобразить ее в jupyter-ноутбуке, нужен немного другой код:

```
from Python.display import SVG
from keras.utils.vis_utils import model_to_dot
```

```
SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```

Важно отметить, что для визуализации нужен пакет graphviz, а также питоновский пакет pydot. Есть тонкий момент, что для корректной работы визуализации пакет pydot из репозитория не пойдет, нужно взять его обновленную версию pydot-ng.

Пакет graphviz в Ubuntu ставится так (в других дистрибутивах Linux аналогично):

```
apt install graphviz
```

Итак, модель сформирована. Теперь нужно подготовить ее к работе:

```
model. compile(loss='categorical_crossentropy',
optimizer='adam',
metrics=['accuracy'])
```

Что означают параметры функции compile? Loss – это функция ошибки, в данном случае – это перекрестная энтропия, именно для нее подготавливали метки в виде матриц; optimizer – используемый оптимизатор, здесь мог бы быть обычный стохастический градиентный спуск, но Adam показывает лучшую сходимость на этой задаче; metrics – метрики, по которым считается качество модели, в рассматриваемом случае – это точность (accuracy), т. е. доля верно угаданных ответов.

Несмотря на то, что Keras содержит большинство популярных функций ошибки, для поставленной задачи может потребоваться что-то уникальное. Чтобы сделать свой собственный loss, нужно немного: просто определить функцию, принимающую векторы правильных и предсказанных ответов и выдающую одно число на выход. Для тренировки сделаем свою функцию расчета перекрестной энтропии. Чтобы она чем-то отличалась, введем так называемый clipping – обрезание значений вектора сверху и снизу. Еще важное замечание: нестандартный loss может быть необходимо описывать в терминах нижележащего фреймворка, но в данном случае можем обойтись средствами Keras.

```
from keras import backend as K
epsilon= 1.0e-9
def custom_objective(y_true, v_pred):
 '"Yet another cross-entropv'"
y_pred = K.clip(y_pred, epsilon, 1.0 - epsilon)
y_pred = K.sum(y_pred.axis – 1, keepdims=True)
cce = categorical_crossentropy(y_pred, y_true)
return cce
```

Здесь y_true и y_pred – тензоры из Tensorflow, поэтому для их обработки используются функции Tensorflow.

Для использования другой функции потерь достаточно изменить значения параметра loss функции compile, передав туда объект функции потерь (в питоне функции – тоже объекты):

```
model. compile(loss=custom_objective, optimizer='adam', metrics=['accuracy'])
```

Наконец, пришло время для обучения модели. Проинициализировать процесс обучения можно с помощью нижеприведенной последовательности команд:

```
history = model.fit(x_train, y train,
batch_size=batch size,
epochs=epochs,
verbose=l,
validation_split=0.1)
```

Метод fit() непосредственно производит обучение нейронной сети. Он принимает на вход обучающую выборку вместе с метками – x_train и y_train, размером батча batch size, который ограничивает количество примеров, подаваемых за раз, количеством эпох дня обучения epochs (одна эпоха – это один раз полностью пройденная моделью обучающая выборка), а также тем, какую долю обучающей выборки отдать под перекрестную проверку – validation_split.

Метод fit() возвращает структуру history – это история ошибок на каждом шаге обучения.

И, наконец, тестирование. Метод cvaluate() получает на вход тестовую выборку вместе с метками для нее. Метрика была задана еще при подготовке к работе, так что больше ничего не нужно (однако укажем еще размер батча).

```
score = model.evaluate(x_test, y_test, batch_size-batch_size)
```

Следует также сказать несколько слов о такой важной особенности Keras. как «колбеки». Через них реализовано много полезной функциональности. Например, если тренируете сеть в течение очень долгого времени, нужно понять, когда пора остановиться, если ошибка на наборе данных перестала уменьшаться. По-английски описываемая функциональность называется «early stopping» («ранняя остановка»). Посмотрим, как можно применить ее при обучении сети:

```
from keras.callbacks import EarlyStopping
early_stopping=EarlyStopping(monitor='value_loss')
history = model.fit(x_trairi, y train,
batch_size=batch_size,
epochs-epochs,
verbose= 1,
validation_spiit=0.1,
```

```
callbacks=[ early_stopping])
```

Проведите эксперимент и проверьте, как быстро сработает earlvstopping в данном примере.

Еще в качестве колбека можно использовать сохранение логов в формате, удобном для Tensorboard (Tensorboard – это специальная утилита для обработки и визуализации информации из логов Tensorflow).

```
from keras.callbacks import TensorBoard
tensorboard=TensorBoard(log_dir='./logs', write_graph=True)
history = model. fit(x_train, y_train.
batch_size=batch_size,
epochs=epochs,
verbose= 1,
validation_split=0.1,
callbacks=[ tensorboard])
```

После того как обучение закончится (или даже в процессе!), можно запустить Tensorboard, указав абсолютный путь к директории с логами:

```
tensorboard --logdir=/path/to/logs
```

Теперь рассмотрим построение чуть более сложного графа вычислений. У нейросети может быть множество входов и выходов, входные данные могут преобразовываться разнообразными отображениями. Для переиспользования частей сложных графов (в частности, для transfer learning) имеет смысл описывать модель в модульном стиле, позволяющем удобным образом извлекать, сохранять и применять к новым входным данным куски модели.

Наиболее удобно описывать модель, смешивая оба способа – Functional API и Sequential API.

Рассмотрим этот подход на примере модели SiameseNetwork. Схожие модели активно применяются на практике для получения векторных представлений, обладающих полезными свойствами. Например, подобная модель может быть использована для того, чтобы выучить такое отображение фотографий лиц в вектор, что вектора для похожих лиц будут близко друг к другу. В частности, этим пользуются приложения поиска по изображениям, такие как FindFace.

Иллюстрацию модели можно видеть на рисунке 4.2.

Здесь функция f превращает входную картинку в вектор, после чего вычисляется расстояние между векторами для пары картинок. Если картинки из одного класса, расстояние нужно минимизировать, если из разных – максимизировать.

После того как такая нейросеть будет обучена, можно представить произвольную картинку в виде вектора f(x) и использовать это представление либо для

поиска ближайших изображений, либо как вектор признаков для других алгоритмов машинного обучения.

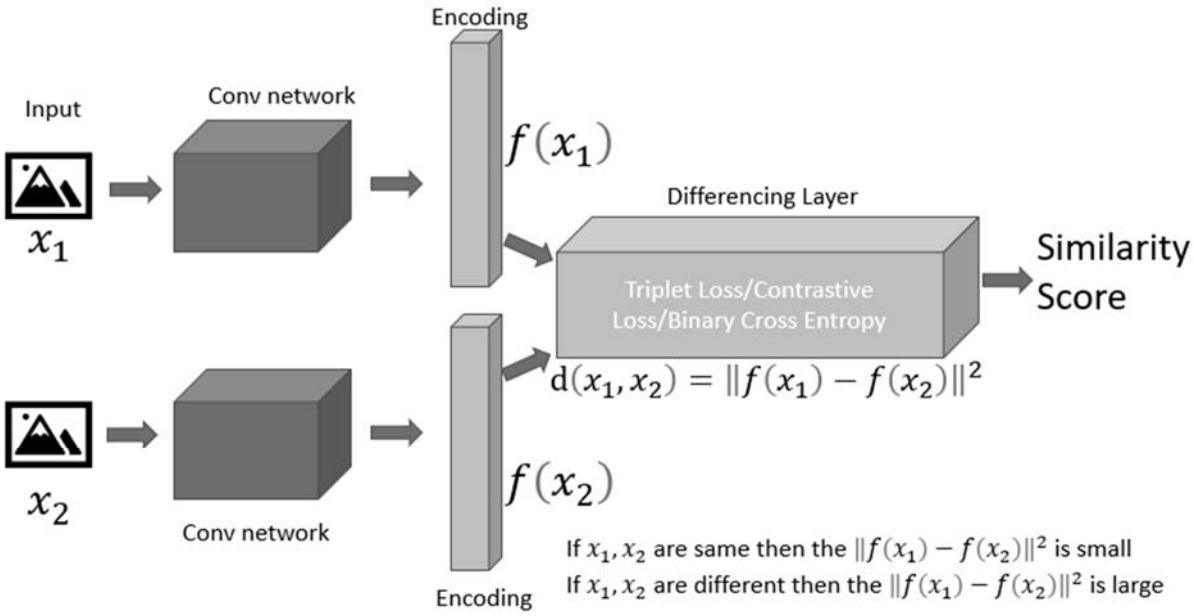


Рисунок 4.2 – Иллюстрация модели SiameseNetwork

Будем описывать модель в коде соответствующим образом, максимально упростив извлечение и переиспользование частей нейросети.

Сначала определим с помощью Keras функцию, отображающую входной вектор.

```
def create_base_network(input_dim):
seq = Sequential
seq.add(Dense(128, input_shape=(input_dim,), activation='relu'))
seq.add(Dropout(0.1))
seq.add(Dense( 128, aclivation='relu'))
seq.add(Dropout(0.1))
seq.add(Dense( 128, activation—relit1))
return seq
```

Обратите внимание: была описана модель с помощью SequentialAPI, однако ее создание обернули в функцию. Теперь можно создать такую модель, вызвав эту функцию, и применить ее с помощью FunctionalAPI ко входным данным:

```
base network = create_base_network(input_dim)
input_a = Input(shape=( input dim,))
inputb = Input(shape=(input_dim,))
processed_a = base_network(input_a)
processed_b = base_network(input_b)
```

В переменных processed_a и processed_b лежат векторные представления, полученные путем применения сети, определенной ранее, ко входным данным.

Нужно посчитать между ними расстояния. Для этого в Keras предусмотрена функция-обертка Lambda, представляющая любое выражение как слой (Layer). Не забудьте, что обрабатываем данные в батчах, так что у всех тензоров всегда есть дополнительная размерность, отвечающая за размер батча:

```
from keras import backend as К
def euclidean_distance(vects):
x, y = vects
return K.sqrt(K.sum(K.square(x - y), axis=l, keepdims=True))

distance = Lambda(euclidean_distance)([processed_a, processed_b])
```

Теперь получили расстояние между внутренними представлениями. Осталось собрать входы и расстояние в одну модель.

```
model = Model([input_a, input_b], distance )
```

Благодаря модульной структуре можно использовать base_network отдельно, что особенно полезно после обучения модели. Как это сделать? Посмотрим на слои модели:

```
>>>model. layers
[<keras.engine.topology.InputI,ayer object at 0x7f238fdacb38>, <keras.engine.topology.InputLayer object at 0x7f238fdc34a8>, <keras.models. Sequential object at 0x7f239127c3c8>, <keras.layers.core.Lambda object at 0x7f238fddc4a8>]
```

Видим третий объект в списке типа models.Sequential. Это и есть модель, отображающая входную картинку и вектор. Чтобы ее извлечь и использовать как полноценную модель (можно дообучать, валидировать, встраивать в другой граф), достаточно всего лишь вытащить ее из списка слоев:

```
>>>embedding_model = model. layers[2]
>>>embedding_model. layers [<keraslayers.eore.Dense object at Ox7f23c4e557fO>, <keras.layers.core.Dropout object at 0x7f238fe97908>, <keras.layers.core.Dense object at 0x7f238fe44898>, <keras.layers.core Dropout object at 0x71238fe449e8> <keras. layers.core.Dense object at 0x7f238fe01f60>]
```

Например, для уже обученной на данных MNIST сиамской сети с размерностью на выходе, равной двум, можно визуализировать векторные представления следующим образом.

Загрузим данные и приведем картинки размером 28 × 28 к плоским векторам.

```
(x_train, y_train), (x_test, y_test) = mnist.loaddata()
xjest = x_test.reshapc( 10000, 784)
```

Отобразим картинки с помощью извлеченной ранее модели:

```
embeddings = embedding_model.predict(x_test)
```

Теперь в embeddings лежат двумерные векторы, их можно изобразить на плоскости.

Недостатки Keras. К сожалению, идея Keras о универсальности кода выполняется не всегда: Keras 2.0 не имеет совместимости с первой версией, некоторые функции стали называться по-другому, некоторые были перемещены, в общем, история похожа на второй и третий python. Отличием является то, что в случае Keras была выбрана только вторая версия для развития. Также код Keras работает на Tensorflow пока медленнее, чем на Theano.

В целом можно порекомендовать Keras к использованию, когда нужно быстро составить и протестировать сеть для решения конкретной задачи. Но если необходимы какие-то сложные вещи, вроде нестандартного слоя или распараллеливания кода на несколько GPU, то лучше (а подчас просто неизбежно) использовать нижележащий фреймворк, т. е. Tensorflow.

Для уменьшения времени обучения и получения большей точности при распознавании полезно использовать предварительно обученные искусственные нейронные сети.

Предварительно обученные нейронные сети позволяют решать задачи компьютерного зрения, не тратя значительного времени на обучение сети. Такие сети создаются крупными компаниями (Google, Microsoft и т. п.), включают большое количество слоев, обладают высокой точностью и обучаются на больших вычислительных кластерах с GPU.

Технология переноса обучения (transfer learmng) позволяет использовать готовые нейронные сети для решения задач нового типа, не тех, для которых сети предварительно обучались. Рассмотрим, как использовать перенос обучения для задач компьютерного зрения в Keras, а также как с помощью сети VGG16 распознавать различные объекты на изображениях.

Keras включает набор предварительно обученных нейронных сетей в модуле Keras Applications. Среди наиболее популярных – сети VGG16 и VGG19, разработанные в Oxford Visual Geometry Group, InceptionV3 компании Google и ResNet50 компании Microsoft. В каждой новой версии Keras появляется все больше предварительно обученных сетей.

Многие нейронные сети для задач классификации объектов на изображениях обучены на наборе данных ImageNet. Этот набор данных включает 14 млн изображений, относящихся к 21 тыс. классов. Однако нейронные сети обучают не на всем наборе ImageNet, а на его части из 1000 классов объектов. Ежегодно проводятся соревнования Large Scale Visual Recognition Challenge по распознаванию именно этих 1000 классов из набора данных ImageNet.

С помощью технологии переноса обучения можно изменить архитектуру предварительно обученной сети таким образом, чтобы она подходила для решения новой задачи. Измененная сеть затем обучается на новом наборе данных.

Нейронные сети, обученные для решения задач классификации изображений, состоят из двух частей:

1) сверточная часть используется для выделения характерных признаков из изображения;

2) полносвязная часть реализует классификацию – определяет, что за объект находится на изображении на основе признаков, которые извлекла сверточная часть.

Идея переноса обучения заключается в следующем. Сверточная часть сети во время обучения учится выделять характерные признаки на изображениях. Если признаки получились достаточно общими, можно взять их и применить для решения другой задачи классификации. Таким образом, переносим обучение сверточной части сети на новую задачу.

Для реализации переноса обучения нужно заменить классификатор в предварительно обученной нейронной сети. Рассмотрим, как это сделать на примере сети VGG16. Данная сеть достаточно просто устроена и ее легко понять, но в то же время качество работы сети довольно высокое. Архитектура сети VGG16 показана на рисунке 4.3.

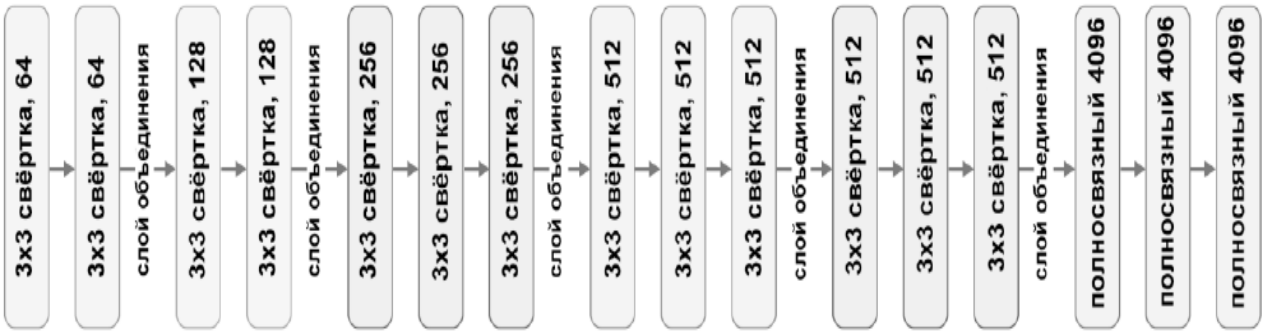


Рисунок 4.3 – Архитектура искусственной нейронной сети VGG16

Сверточная часть сети VGG16 состоит из пяти каскадов свертки и подвыборки. В первых двух каскадах используются по два слоя свертки и слой подвыборки с выбором максимального значения (maxpooling). На трех следующих каскадах по три слоя свертки и один слой подвыборки. Размер ядер во всех слоях свертки 3 × 3.

Полносвязная часть сети VGG16 включает три уровня. На выходном уровне 1000 нейронов по количеству классов объектов. Используется формат one-hotencoding: значение только одного выходного нейрона должно быть близко к единице, остальные близки к нулю. Класс объекта на картинке соответствует нейрону, значение которого близко к единице. Перед выходным слоем в сети VGG16 еще два полносвязных слоя по 4096 нейронов.

На первом этапе необходимо убрать полносвязную часть из сети VGG16. Получится следующая сеть, архитектура которой представлена на рисунке 4.4.

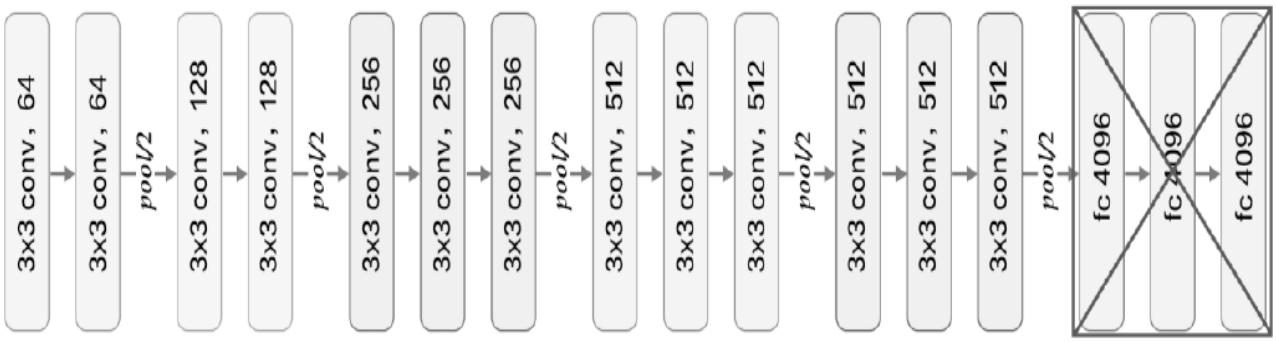


Рисунок 4.4 – Архитектура искусственной нейронной сети VGG16, лишенная полносвязной части

Второй этап: к сверточной части сети VGG16 добавляем новый классификатор для распознавания собственных классов, как показано на рисунке 4.5.

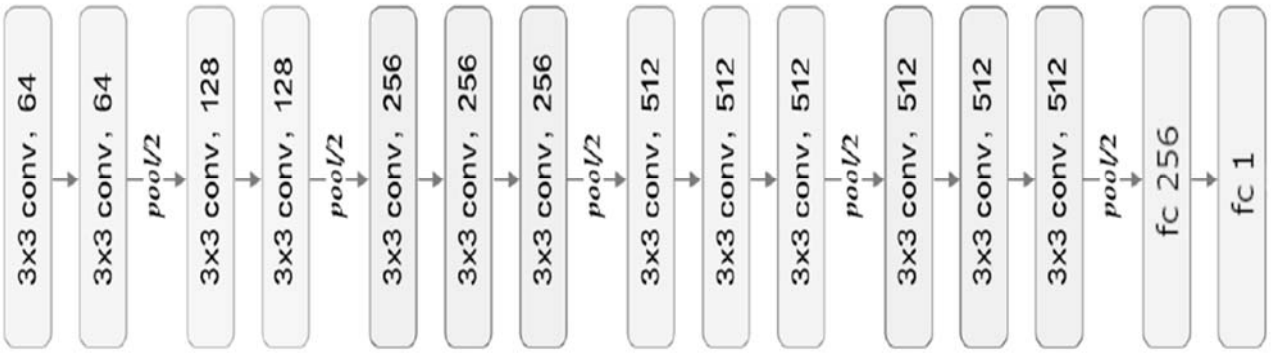


Рисунок 4.5 – Добавление новых классов к структуре VGG16

Определение типа объекта.

Новый классификатор устроен гораздо проще полносвязной части сети VGG16, т. к. нужно распознавать всего два класса объектов, а не 1000. На выходном слое один нейрон, что соответствует задаче бинарной классификации. Ноль на выходе из сети означает, что на фотографии отсутствует объект, а единица – присутствует. Перед выходным слоем находится еще один полносвязный слой, в котором 256 нейронов. На вход этого слоя поступают данные из сверточной части сети VGG16.

На третьем этане измененную сеть нужно обучить на новом наборе данных с фотографиями котов и собак.

Реализация переноса обучения на Keras.

Рассмотрим, как реализовать перенос обучения для распознавания объектов в Keras.

Для начала подключаем необходимые модули Keras. Для TensorFlow версии 1.4 и выше, которая уже содержит Keras:

```
# Определение типа объекта
from tensorflow.python.keras.preprocessing_image import ImageDataGenerator
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Activation, Dropout, Flatten, Dense
from tensorflow.python.keras.applications import VGG16
from tensorflow.python.keras.optimizers import Adam
```

Если Keras установлен отдельно от TensorFlow, в том числе с другим бэкендом:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Activation, Dropout, Flatten, Dense
from keras.applications import VGG16
from keras.optimizers import Adam
```

Задаем константы:

```
# Каталог с данными для обучения
train_dir = 'train'
# Каталог с данными для проверки
val_dir = 'val'
# Каталог с данными для тестирования
test_dir = 'test'
# Размеры изображения
img_width, img_lieight = 150, 150
#Размерность тензора на основе изображения для входных данных в нейронную сеть
backendTensorflow, channelsjast input_shape = (imgwidth, img_height, 3)
# Размер мини-выборки
batchsize = 64
# Количество изображений для обучения
nb_train_samples = 17500
# Количество изображений для проверки
nb_validation_samples = 3750
#Количество изображений для тестирования
nb_test_samples = 3750
```

Фотографии должны быть разделены на три набора данных: для обучения, проверки и тестирования, которые находятся в трех каталогах.

Загружаем предварительно обученную нейронную сеть. Для загрузки сети VGG16 используем следующую команду:

```
vggl6_net = VGG16(weights- lmagenet',
include_top=False,
input_shape=( 150, 150, 3))
```

Обратите внимание, что используем параметр include_top = False, который говорит Keras, что не нужно загружать часть сети VGG16, отвечающую за классификацию. Будет загружена только сверточная часть сети.

Необходимо запретить обучать сеть VGG16, в противном случае веса в сети могут испортиться в процессе обучения с новым классификатором. В классификаторе, который добавим к сети, веса нейронов будут инициализированы случайными числами. Поэтому на первых этапах обучения значения ошибки на выходе из сети будут очень большими. По алгоритму обратного распространения ошибки сигнал об ошибке будет передаваться и в сверточную часть сети VGG16, из-за чего веса в ней могут испортиться.

Для запрещения обучения сети VGG16 устанавливаем значение ноля trainable в False.

После этого можно создавать новую искусственную нейронную сеть. Новая сеть будет включать сверточную часть VGG16 и новый классификатор для распознавания объектов. Для создания такой составной сети будем использовать последовательную (Sequential) модель Keras. Модель задается следующим образом:

```
model = Sequential()
model.add(vggl6_net)
model.add(Flatten())
model.add(Dense(256))
model.add(Activation('relu'))
model.add( Dropout(0.5))
model.add(Dense( 1))
model. add(Activation('sigmoid'))
```

Особенность создания такой сети заключается в том, что на первом этапе добавляем в сеть не отдельный слой, а целую предварительно обученную модель:

```
model.add( vgg16_net)
```

После этого добавляем в модель слой Flatten, который преобразует двумерные вектора признаков, полученные от свёрточной части сети VGG16, в одномерный вектор. Полносвязные слои, используемые для классификации, не могут работать с двумерными данными на входе, им нужен одномерный вектор. Первый полносвязный (Dense) слой содержит 256 нейронов, функция активации полулинейная. Затем идет слой Dropout, который используется для снижения переобучения.

На выходном полносвязном слое один нейрон. Имеем два возможных класса объектов: кот и собака. Им соответствуют значения выходного нейрона ноль и единица. Функция активации выходного нейрона сигмоидальная. Эта функция плавно меняет свое значение от нуля до единицы, поэтому она хорошо подходит для бинарной классификации.

Видно, что первый слой в сети – это отдельная модель. Всего в сети 16,812,353 параметров, из них обучаемых – 2,097,665. Все обучаемые параметры относятся к полносвязным слоям (2,097,408 параметров – на слое dense_l и 257 – на выходном слое dense_2).

После этого обучаем нейронную сеть на новом наборе данных. Перед обучением сеть необходимо скомпилировать:

```
model.compile(loss='binary_crossentropy',
optimizer=Adam(lr= 1 e-5),
metrics=[ 'accuracy'])
```

В качестве функции ошибки указываем binary_crossentropy, потому что имеем задачу бинарной классификации. Если классов больше, чем два, то нужно использовать categorical_crossentropv.

Часть сети уже предварительно обучена, поэтому для новой части нужно использовать небольшую скорость обучения, иначе алгоритм обучения может не сойтись. Параметр скорости обучения указываем при создании оптимизатора Adam в параметре lr (сокращение от learning rate, скорость обучения). Значение параметра lr в примере единица на десять в минус пятой степени (для оптимизатора Adam по умолчанию lr = 0,001, на два порядка больше).

Обучать сеть будем при помощи генераторов изображений Keras. Создаем генераторы для обучающего, проверочного и тестового наборов данных:

```
datagen = ImageDataGenerator(rescale=l. / 255)
train_generator = datagen.flow_from directory(
train_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode='binary')

valgenerator = datagen.flow_from_directory( val_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode='binary')

test_generator = datagen.flow_from_directorv( tesl_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode='binary')
```

Обучаем сеть с использованием генераторов:

```
model.fit_generator(
train_generator,
steps_per_epoch=nb_train_samples // batch_size, epochs= 10,
validation_data=val_generator,
validation_steps=nb_validation_samples // batch size)
```

Сеть обучается достаточно долго. На GPU NVIDIA GTX 1050Ti одна эпоха обучения занимает примерно 450 с, общее время обучения составляет примерно 75 мин. На центральном процессоре время обучения будет значительно больше. Поэтому не рекомендуется обучать такую сеть без GPU.

После завершения обучения сети необходимо обязательно проверить качество ее работы на данных, которые сеть не видела в процессе обучения.

Для проверки будем использовать подготовленный ранее генератор тестовых данных test generator:

```
scores = model.evaluate_generator(test_generator, nb_test_ samples // batch_size)
print("Аккуратность на тестовых данных: %.2t%%" % (scores[ 1]* 100))
```

Если видим, что точность на тестовом наборе данных незначительно меньше, чем на проверочном и обучающем, то это свидетельствует об отсутствии переобучения.

С другой стороны, при обучении сверточной нейронной сети распознавались объекты с нуля, аккуратность на тестовом наборе данных была 84,3 %. Использование предварительно обученной сверточиой нейронной сети VGG16 позволило поднять точность работы более чем на 10 %.

В ходе работы были получены навыки использования технологии переноса обучения (transfer learning) для применения предварительно обученных сетей к новым типам задач. Далее приведена краткая схема переноса обучения для задач компьютерного зрения.

1 Выбрать предварительно обученную нейронную сеть и взять от нее только сверточную часть, удалив полносвязную. В Keras нужно указать параметр include_top=False при создании сети.

2 Создать составную сеть на основе предварительно обученной сети и нового классификатора, который создается для поставленной задачи. Предварительно обученную сеть можно добавить в составную сеть таким же образом, как добавляем отдельные слои.

3 Обучить составную сеть на новом наборе данных. Обучать нужно только веса добавленного в есть классификатора, предварительно обученная сверточная часть должна быть «заморожена».

В качестве примера было рассмотрено применение сети VGG16 для распознавания различных объектов на изображениях. Keras включает другие предварительно обученные сети, можно попробовать использовать их для этой

же задачи. Особенно рекомендуется протестировать сети InceptionV3 и ResNet5(), они были разработаны позже сети VGG16, имеют более сложную структуру и обеспечивают более высокую точность.

Также можно попробовать применить описанный подход для классификации изображений из другого набора данных, в том числе с количеством классов больше двух.

После успешного обучения искусственных нейронных сетей нужно разработать программное обеспечение для управления мобильным роботом. Основная система управления роботом запрограммирована заранее таким образом, чтобы отправлять по Wi-Fi через сокет-изображение, получаемое с видеокамеры, и принимать управляющее воздействие. Формат строки с управляющими воздействиями нужно уточнить у преподавателя, т. к. управляющие программы, запускаемые на RaspberryPi, могут совершенствоваться и формат данных может быть изменен. Вспомогательные системы мобильного робота также запрограммированы заранее.

Задачей студентов является разработка управляющего скрипта на языке программирования Python. Данный скрипт должен быть запущен на персональном компьютере, подключенном к той же сети WiFi, что и робот.

Пример алгоритма приема изображения через сокет можно найти на официальном сайте RaspberryPi разделе Basic Recipes. В управляющем скрипте должны быть проинициализированы и запущены модели искусственных нейронных сетей для управления движением и распознавания объектов. При обнаружении препятствия должен быть выработан сигнал останова и отправлен роботу вместе со строкой, содержащей управляющие воздействия.

Список литературы

1 **Гуров, В. В.** Микропроцессорные системы : учебное пособие / В. В. Гуров. – Москва : ИНФРА-М, 2022. – 336 с.

2 **Жежера, Н. И.** Микропроцессорные системы автоматизации технологических процессов : учебное пособие / Н. И. Жежера. – 2-е изд. – Москва; Вологда: Инфра-Инженерия, 2020. – 240 с. : ил.

3 **Овсянников, Е. М.** Электрический привод : учебник / Е. М. Овсянников. – Москва : ФОРУМ, 2011. – 224 с.

4 **Хартов, В. Я.** Микропроцессорные системы : учебное пособие / В. Я. Хартов. – Москва : Академия, 2010. – 352 с.