МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Политехнический институт: Заочный
Кафедра «Системы автоматического управления»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой
__________________/ В.И. Ширяев
«_____» ________________ 2018 г.

Нейронно-сетевые алгоритмы распознавания поверхности космических тел

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ – 09.03.01.2018.751.00 ПЗ ВКР

Руководитель работы
к.т.н, доцент каф САУ____________
_________/ __В.Н. Кожеуров_____
«____» _________________ 2018 г.

Автор работы
студент группы **ПЗ-597**
_________/ __А.С. Руденко_______
«____» _________________ 2018 г.

Нормоконтролер
к.т.н, доцент каф САУ____________
_________/ __В.Н. Кожеуров_____
«____» _________________ 2018 г.

Челябинск 2018

АННОТАЦИЯ

Руденко А.С. Нейронно-сетевые алгоритмы распознавания поверхности космических тел: ЮУрГУ (НИУ), ПИ: Заочный; 2018, 78 с. 42 ил., библиогр. список – 17 наим., 12 листов слайдов презентации ф. А4.

В выпускной квалификационной работе рассмотрены основные типы нейронных сетей и нейронное сетевые алгоритмы распознавания изображений.

В работе проводится анализ построения и функционирования сверточных нейронных сетей, являющихся основным механизмом, используемым для анализа и распознавания изображений. Так же проводится анализ способов получения необходимых вычислительных мощностей, требуемых для обучения нейронных сетей.

На основании общей модели функционирования сверточной нейронной сети описаны математические модели входящих в нее слоев, а также приводятся методы оптимизации ее работы.

Представлена программа построения, обучения и тестирования сверточной нейронной сети в среде разработке Eclipse с применением программных библиотек Theano и Lasagne, а также приводятся результаты ее работы.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 7
1 ОБЗОР И АНАЛИЗ СУЩЕСТВУЮЩИХ МОДЕЛЕЙ НЕЙРОНЫХ СЕТЕЙ ПРЕДНАЗНАЧЕНЫХ ДЛЯ ОБРАБОТКИ И РАСПОЗНОВАНИЯ ИЗОБРАЖЕНИЙ 10
1.1 Общий обзор архитектур нейронных сетей 10
1.2 Выбор архитектур нейронной сети реализующих обработку изображений 13
1.3 Анализ структуры и принципов построения сверточных нейронных сетей 15
1.2.1 Слой свёртки 21
1.2.2 Субдискретизирующий слой (слой пулинга) 29
1.2.3 Полносвязный слой 31
1.2.4 Принципы построяния сверточных нейронных сетей 33
2 АНАЛИЗ МОДЕЛИ СВЕРТОЧНОЙ НЕЙРОНОЙ СЕТИ И МЕХАНИЗМОВ ЕЕ ОПТИМИЗАЦИИ. 37
2.1 Математическая модель нейронной сети. 37
2.1.1 Математическая модель сверточного слоя. 37
2.1.2 Математическая модель субдискретизирующего слоя. 38
2.1.3 Математическая модель выходного слоя. 39
2.1.4 Обучение сверточной нейронной сети. 39
2.1.5 Метод обратного распространения ошибки. 41
2.2 Механизмы настройки сверточной нейронной сети. 43
2.2.1 Выбор стратегии обучения. 43
2.2.2 Градиентные методы обучения первого порядка. 43
2.2.3 Регуляризация 48
2.2.4 Батч-нормализация. 50
2.2.5 Ранняя остановка 51
2.2.6 Перенос обучения. 52
2.2.7 Тонкая настройка сверточной нейронной сети 56

					09.03.01.2018.751.00 ПЗ	Лист
						5
Изм.	Лист	№ докум.	Подпись	Дата		

3 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 58
3.1 Подготовка данных 58
3.2 Обоснование выбора среды программной реализации проекта 60
3.3 Высокопроизводительные средства вычислений 62
3.4 Внесение изменение в выбранную архитектуру сети. 67
3.5 Обучение и тестирование сверточной нейронной сети. 69
ЗАКЛЮЧЕНИЕ 76
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 77
ПРИЛОЖЕНИЯ
ПРИЛОЖЕНИЕ А........ 79

ВВЕДЕНИЕ

Понятие нейронной сети, как одно из направлений исследования в области искусственного интеллекта, основанное на попытках воспроизвести нервную систему человека, появилось в середине двадцатого века. С тех пор этот метод машинного обучения получил огромное развитие и популярность. Нейронные сети используются во многих предметных областях: например, маркетинг и реклама, экономика и финансы, информационные технологии и защита информации.

Совершенно очевидно, что свою силу нейронные сети черпают, во-первых, из распараллеливания обработки информации и, во-вторых, из способности самообучаться, т.е. создавать обобщения. Под термином обобщение понимается способность получать обоснованный результат на основании данных, которые не встречались в процессе обучения. Эти свойства позволяют нейронным сетям решать сложные (масштабные) задачи, которые на сегодняшний день считаются трудноразрешимыми. Однако на практике при автономной работе нейронные сети не могут обеспечить готовые решения. Их необходимо интегрировать в сложные системы. В частности, комплексную задачу можно разбить на последовательность относительно простых, часть из которых может решаться нейронными сетями.

Используя способность обучения на множестве примеров, нейронная сеть способная решать задачи, в которых неизвестны закономерности развития ситуации и зависимости между входными и выходными данными. Традиционные математические методы и экспертные системы в таких случаях дают неудовлетворительны результат.

Нейронные сети обеспечивают возможность работы при наличии большого числа неинформативных, шумовых входных сигналов. Нет необходимости делать их предварительный отсев, нейронная сеть сама определит их малопригодность для решения задачи и отбросит их.

Нейронные сети обладают способностью адаптироваться к изменениям окружающей среды. В частности, нейронные сети, обученные действовать в определенной среде, могут быть легко переучены для работы в условиях незначительных колебаний параметров среды. Более того, для работы в нестационарной среде (где статистика изменяется с течением времени) могут быть созданы нейронные сети, переучивающиеся в реальном времени. Чем выше адаптивные способности системы, тем более устойчивой будет ее работа в нестационарной среде. При этом следует заметить, что адаптивность не всегда ведет к устойчивости; иногда она приводит к совершенно противоположному результату. Например, адаптивная система с параметрами, быстро изменяющимися во времени, может также быстро реагировать и на посторонние возбуждения, что

вызовет потерю производительности. Для того чтобы использовать все достоинства адаптивности, основные параметры системы должны быть достаточно стабильными, чтобы можно было не учитывать внешние помехи, и достаточно гибкими, чтобы обеспечить реакцию на существенные изменения среды.

Нейронные сети обладают потенциальным сверхвысоким быстродействием за счет использования массового параллелизма обработки информации.

Нейронные сети потенциально отказоустойчивы. Это значит, что при неблагоприятных условиях их производительность падает незначительно. Например, если поврежден какой-то нейрон или его связи, извлечение запомненной информации затрудняется. Однако, принимая в расчет распределенный характер хранения информации в нейронной сети, можно утверждать, что только серьезные повреждения структуры нейронной сети существенно повлияют на ее работоспособность.

Несмотря на широкий спектр возможностей, нейронные сети обладают также и рядом недостатков, связанных в основном со сложностью их построения, настройки и обучения. Поиск оптимального соотношения параметров нейросетевых моделей и их характеристик для каждого конкретного случая является ключевой задачей, для эффективного решения которой необходим широкий спектр методов, алгоритмов и подходов, различающихся по объему вычислений, получаемому результату, затраченному времени, способам представления данных.

Объект исследования – нейронно-сетевые алгоритмы распознавания изображений.

Предмет исследования – реализация нейронно-сетевого алгоритма распознавания изображений.

Цель работы – разработка программного обеспечения для построения, обучения и тестирования нейронной сети для распознавания изображения поверхности космических тел.

Для достижения поставленной цели необходимо решить следующие задачи:

− проанализировать существующие архитектуры нейронных сетей позволяющих распознавать и обрабатывать изображения;

− выбрать архитектуру нейронной сети;

− описать математическую модель нейронной сети;

− разработать программу создающую, обучающую и тестирующую нейронную сеть, обеспечивающую распознавание изображений поверхности космических тел;

Методы исследования − анализ, сравнение, обобщение, моделирование, формализация, разработка, практический эксперимент.

Структура работы. Дипломная работа состоит из введения, трех разделов, которые делятся на параграфы, заключения, списка использованных источников и приложений.

1 ОБЗОР И АНАЛИЗ СУЩЕСТВУЮЩИХ МОДЕЛЕЙ НЕЙРОНЫХ СЕТЕЙ ПРЕДНАЗНАЧЕНЫХ ДЛЯ ОБРАБОТКИ И РАСПОЗНОВАНИЯ ИЗОБРАЖЕНИЙ

1.1 Общий обзор архитектур нейронных сетей

Нейронные сети прямого распространения (feed forward neural networks, FF или FFNN) и перцептроны (perceptrons, P) прямолинейны, они передают информацию от входа к выходу. Нейронная сеть состоит из слоев, где каждый слой состоит из входных, скрытых или выходных нейронов. Нейроны одного слоя не связаны между собой, а соседние слои обычно полностью связаны. Самая простая нейронная сеть имеет две входных клетки и одну выходную, и может использоваться в качестве модели логических вентилей. FFNN обычно обучается по методу обратного распространения ошибки. Этот процесс называется обучением с учителем, и он отличается от обучения без учителя тем, что во втором случае множество выходных данных сеть составляет самостоятельно. Практически такие сети используются редко, но их часто комбинируют с другими типами для получения новых.

Сети радиально-базисных функций (radial basis function, RBF) — это FFNN, которая использует радиальные базисные функции как функции активации.

Нейронная сеть Хопфилда (Hopfield network, HN) — это полносвязная нейронная сеть с симметричной матрицей связей. Во время получения входных данных каждый узел является входом, в процессе обучения он становится скрытым, а затем становится выходом. Сеть обучается так: значения нейронов устанавливаются в соответствии с желаемым шаблоном, после чего вычисляются веса, которые в дальнейшем не меняются. После того, как сеть обучилась на одном или нескольких шаблонах, она всегда будет сводиться к одному из них (но не всегда — к желаемому). Она стабилизируется в зависимости от общей «энергии» и «температуры» сети. Такая сеть часто называется сетью с ассоциативной памятью; как человек, видя половину таблицы, может представить вторую половину таблицы, так и эта сеть, получая таблицу, наполовину зашумленную, восстанавливает её до полной.

Машина Больцмана (Boltzmann machine, BM) очень похожа на сеть Хопфилда, но в ней некоторые нейроны помечены как входные, а некоторые — как скрытые. Входные нейроны в дальнейшем становятся выходными. Машина Больцмана — это стохастическая сеть. Обучение проходит по методу обратного распространения ошибки или по алгоритму сравнительной расходимости. В целом процесс обучения очень похож на таковой у сети Хопфилда.

Ограниченная машина Больцмана (restricted Boltzmann machine, RBM) похожа на машину Больцмана и, следовательно, на сеть Хопфилда. Единственной

разницей является её ограниченность. В ней нейроны одного типа не связаны между собой. Ограниченную машину Больцмана можно обучать как FFNN, но с одним нюансом: вместо прямой передачи данных и обратного распространения ошибки нужно передавать данные сперва в прямом направлении, затем в обратном. После этого проходит обучение по методу прямого и обратного распространения ошибки.

Автокодировщик (autoencoder, AE) отчасти похож на FFNN, так как это скорее другой способ использования FFNN, нежели фундаментально другая архитектура. Основной идеей является автоматическое кодирование (в смысле сжатия, не шифрования) информации. В ней скрытые слои меньше входного и выходного, причём она симметрична. Сеть можно обучить методом обратного распространения ошибки, подавая входные данные и задавая ошибку равной разнице между входом и выходом.

Разреженный автокодировщик (sparse autoencoder, SAE) — в каком-то смысле противоположность обычного. Вместо того, чтобы обучать сеть отображать информацию в меньшем «объёме» узлов, мы увеличиваем их количество. Вместо того, чтобы сужаться к центру, сеть там раздувается. Сети такого типа полезны для работы с большим количеством мелких свойств набора данных. Кроме входных данных подаётся ещё и специальный фильтр разреженности, который пропускает только определённые ошибки.

Свёрточные нейронные сети (convolutional neural networks, CNN)и глубинные свёрточные нейронные сети (deep convolutional neural networks, DCNN) сильно отличаются от других видов сетей. Обычно они используются для обработки изображений, реже для аудио. Типичным способом применения CNN является классификация изображений. На практике к концу CNN прикрепляют FFNN для дальнейшей обработки данных. Такие сети называются глубинными (DCNN).

Развёртывающие нейронные сети (deconvolutional networks, DN), также называемые обратными графическими сетями, являются обратным к свёрточным нейронным сетям. DNN тоже можно объединять с FFNN. Стоит заметить, что в большинстве случаев сети передаётся не строка, а бинарный вектор.

Глубинные свёрточные обратные графические сети (deep convolutional inverse graphics networks, DCIGN) по сути являются вариационными автокодировщиками, кодирующая и декодирующая части которых представлены свёрточной и развёртывающей НС соответственно. Сети такого типа моделируют свойства в виде вероятностей. Сети такого типа обычно обучают методом обратного распространения ошибки.

Генеративные состязательные сети (generative adversarial networks, GAN) — это сети другого вида, они похожи на близнецов. Такие сети состоят из любых двух (обычно из FF и CNN), одна из которых контент генерирует, а другая — оценивает. Сеть-дискриминатор получает обучающие или созданные генератором данные. Степень угадывания дискриминатором источника данных в дальнейшем участвует в формировании ошибки. Таким образом, возникает состязание между генератором и дискриминатором, где первый учится обманывать первого, а второй — раскрывать обман. Обучать такие сети весьма тяжело, поскольку нужно не только обучить каждую из них, но и настроить баланс.

Рекуррентные нейронные сети (recurrent neural networks, RNN) — это сети типа FFNN, но с особенностью: нейроны получают информацию не только от предыдущего слоя, но и от самих себя предыдущего прохода. Это означает, что порядок, в котором вы подаёте данные и обучаете сеть, становится важным. Большой сложностью сетей RNN является проблема исчезающего (или взрывного) градиента, которая заключается в быстрой потере информации с течением времени. Обычно сети такого типа используются для автоматического дополнения информации.

Сети с долгой краткосрочной памятью (long short term memory, LSTM) стараются решить вышеупомянутую проблему потери информации, используя фильтры и явно заданную клетку памяти. У каждого нейрона есть клетка памяти и три фильтра: входной, выходной и забывающий. Целью этих фильтров является защита информации. Входной фильтр определяет, сколько информации из предыдущего слоя будет храниться в клетке. Выходной фильтр определяет, сколько информации получат следующие слои.

Управляемые рекуррентные нейроны (gated recurrent units, GRU) — это небольшая вариация предыдущей сети. У них на один фильтр меньше, и связи реализованы иначе. Фильтр обновления определяет, сколько информации останется от прошлого состояния и сколько будет взято из предыдущего слоя. Фильтр сброса работает примерно как забывающий фильтр.

Нейронные машины Тьюринга (neural Turing machines, NTM) можно рассматривать как абстрактную модель LSTM и попытку показать, что на самом деле происходит внутри нейронной сети. Ячейка памяти не помещена в нейрон, а размещена отдельно с целью объединить эффективность обычного хранилища данных и мощь нейронной сети. Собственно, поэтому такие сети и называются машинами Тьюринга — в силу способности читать и записывать данные и менять состояние в зависимости от прочитанного они являются тьюринг-полными.

Глубинные остаточные сети (deep residual networks, DRN) — это очень глубокие сети типа FFNN с дополнительными связями между отделёнными друг от

друга слоями. Такие сети можно обучать на шаблонах глубиной до 150 слоёв. Однако, было показано, что эти сети мало чем отличаются от рекуррентных, и их часто сравнивают с сетями LSTM.

Метод опорных векторов (support vector machines, SVM) находит оптимальные решения задачи оптимизации. Классическая версия способна категоризировать линейно разделяемые данные: например, различать изображения с котом Томом и с котом Гарфилдом. В процессе обучения сеть как бы размещает все данные на 2D-графике и пытается разделить данные прямой линией так, чтобы с каждой стороны были данные только одного класса и чтобы расстояние от данные до линии было максимальным. Используя трюк с ядром, можно классифицировать данные размерности n. Что характерно, этот метод не всегда рассматривается как нейронная сеть.

Сети Кохонена (Kohonen networks, KN), также известные каксамоорганизующиеся карты(self organising (feature) maps, SOM, SOFM). Эти сети используют соревновательное обучение для классификации данных без учителя. Сети подаются входные данные, после чего сеть определяет, какие из нейронов максимально совпадают с ними. После этого эти нейроны изменяются для ещё большей точности совпадения, в процессе двигая за собой соседей.

Из всех выше перечисленных сетей для анализа и обработки изображений используются сверточные нейроные сети и нейронные сети прямого распространения.

1.2 Выбор архитектур нейронной сети реализующих обработку изображений

Как известно, нейронные сети получают входные данные (один вектор), после чего трансформируют информацию, проводя ее через ряд скрытых слоев. Каждый скрытый слой состоит из множества нейронов, где всякий нейрон имеет устойчивую связь со всеми нейронами в предыдущем слое и где нейроны в функции одного слоя полностью независимы друг от друга и не имеют общих соединений. Последний полносвязный слой называется выходным слоем, и в настройках классификации он демонстрирует число классов.

Нейронные сети прямого распространения плохо масштабируются в случае с изображениями больших размеров. Так, в системе компьютерного зрения, изображения имеют размер [m×n×s] (m – ширина, n – высота, s – цветовые каналы), поэтому один полностью подключенный нейрон в первом скрытом слое обычной нейронной сети обладает количеством связей 3072 (32*32*3). Изображение с более высоким разрешением, например, [200×200×3], приведет к тому, что полностью подключенный нейрон будет иметь 120000 связей. Кроме того, мы почти наверняка

хотели бы иметь несколько таких нейронов, что привело бы к добавлению параметров. Полная связность – это огромное количество параметров, которое может быстро привести к переобучению.

Применение сверточных нейроных сетей (СНС) предоставляет более эффективный способ обработки изображения за счет самой структуру изображения: предполагается, что пиксели, находящиеся близко друг к другу, теснее “взаимодействуют” при формировании интересующего нас признака, чем пиксели, расположенные в противоположных углах. Кроме того, если в процессе классификации изображения небольшая черта считается очень важной, не будет иметь значения, на каком участке изображения эта черта обнаружена. В отличие от обычной нейронной сети, слои СНС состоят из нейронов, расположенных в 3-х измерениях: ширине, высоте и глубине, т. е. измерениях, которые формируют объем. Например, изображения на входе являются входными активационными объемами, а объем сформирован измерениями 32×32×3. Как будет показано далее, нейроны будут подключены только к небольшой области слоя перед этим участком. Кроме того, результирующий выходной слой для данной системы компьютерного зрения будет соответствовать размеру 1×1×X, поскольку к концу построения СНС будет произведено преобразование изображения в единый вектор оценок класса, расположенных по измерению глубины. Ниже приводится визуализация описанного процесса.

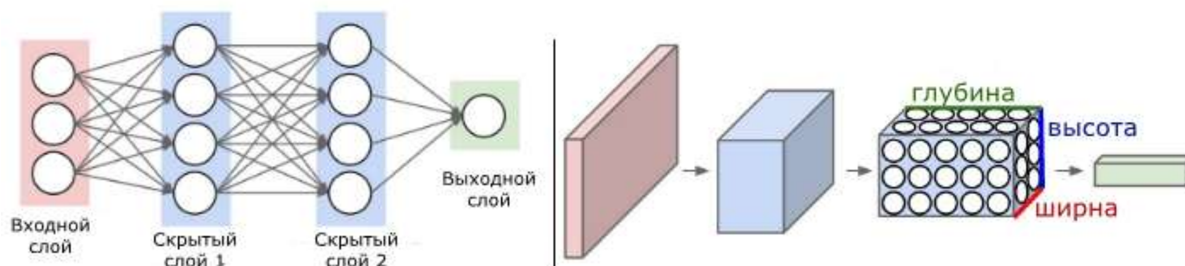


Рисунок 1.1 – Базовые структуры нейронных сетей.

В левой части изображено расположение нейронов в обычной, трехслойной нейронной сети, в правой, нейроны СНС расположенные в 3-х измерениях, представленные на одном из слоев. Каждый слой СНС преобразует входной 3-х мерный объем в выходной активационный объем нейронов, также 3-х мерный.

Следует отметить, что сверточные нейронные сети обеспечивают частичную устойчивость к изменениям масштаба, смещениям, поворотам, смене ракурса и прочим искажениям. Сверточные нейронные сети объединяют три архитектурных идеи, для обеспечения инвариантности к изменению масштаба, повороту сдвигу и пространственным искажениям:

- локальные рецепторные поля (обеспечивают локальную двумерную связность нейронов);

- общие синаптические коэффициенты (обеспечивают детектирование некоторых черт в любом месте изображения и уменьшают общее число весовых коэффициентов);
- иерархическая организация с пространственными подвыборками.

1.3 Анализ структуры и принципов построения сверточных нейронных сетей

Свёрточные нейронные сети (СНС) очень похожи на нейронные сети прямого распространения: они также построены на основе нейронов, которые обладают изменяющимся весом и смещениями. Каждый нейрон получает некоторые входные данные, выполняет скалярное произведение и в отдельных ситуациях сопровождает это нелинейностью. Как и в случае с нейронные сети прямого распространения, вся СНС выражает одну дифференцируемую функцию оценки: с одной стороны это необработанные пиксели изображения, с другой – вывод класса или группы вероятных классов, характеризующих изображение. Здесь также присутствует функция потери на последнем (полносвязном) слое, а все механизмы, разработанные и изученные для стандартных нейронных сетей, остаются справедливыми при работе с СНС.

Архитектура свёрточных нейроных сетей делает явное предположение вида «входные данные есть изображения», что позволяет закодировать определенные свойства под архитектуру. Благодаря этой особенности, удается уменьшить количество параметров в сети.

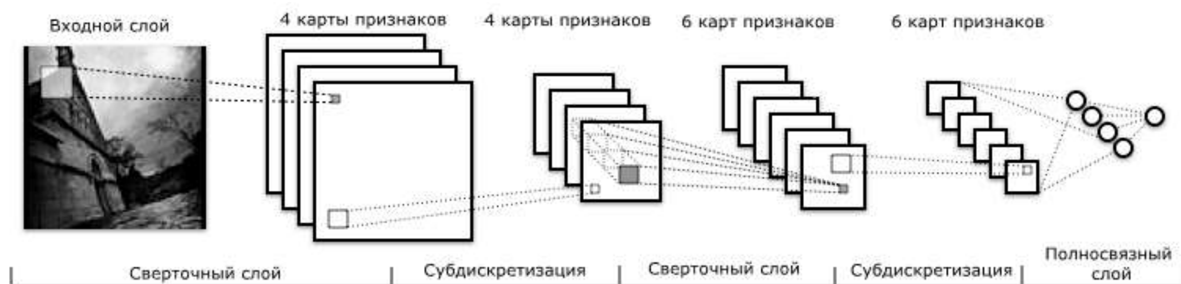


Рисунок 1.2 – Структура сверточной нейронной сети.

Как уже было сказано выше, схематично СНС – это последовательность слоев. Каждый слой преобразует один активационный объем в другой с помощью дифференцируемой функции. Для организации свёрточной нейронной сети применяется 3 основных слоя:

- сверточный слой;
- слой субдискретизации (иначе подвыборки или пулинга);
- полносвязный слой.

Эти слои используются с целью построения полной архитектуры СНС.

Далее рассмотрим организацию СНС более подробно, а простым примером свёрточной нейросети в контексте классификации системы распознавания изображений может послужить архитектура [Входной слой – Сверточный слой – Слой ректификации – Субдискретизирующий слой – Полносвязный слой].

Входной слой (INPUT) учитывает двумерную топологию изображений и состоит из нескольких карт (матриц [32×32×3], где 32 – ширина, 32 – высота, 3 – цветовые каналы R, G, B), карта может быть одна, в том случае, если изображение представлено в оттенках серого, иначе их 3, где каждая карта соответствует изображению с конкретным каналом (красным, синим и зеленым).

Входные данные каждого конкретного значения пикселя нормализуются в диапазон от 0 до 1, по формуле:

$$f(p, min, max) = \frac{p - min}{max - min}, \quad (1.1)$$

где f – функция нормализации;

p – значение конкретного пиксела от 0 до 255;

min – минимальное значение пикселя;

max – максимальное значение пикселя.

Сверточный слой (CONV) представляет из себя набор карт (другое название – карты признаков, в обиходе это обычные матрицы), у каждой карты есть синаптическое ядро (в разных источниках его называют по-разному: сканирующее ядро или фильтр).

Количество карт определяется требованиями к задаче, если взять большое количество карт, то повысится качество распознавания, но увеличится вычислительная сложность. Исходя из анализа научных статей, в большинстве случаев предлагается брать соотношение один к двум, то есть каждая карта предыдущего слоя (например, у первого сверточного слоя, предыдущим является входной) связана с двумя картами сверточного слоя.

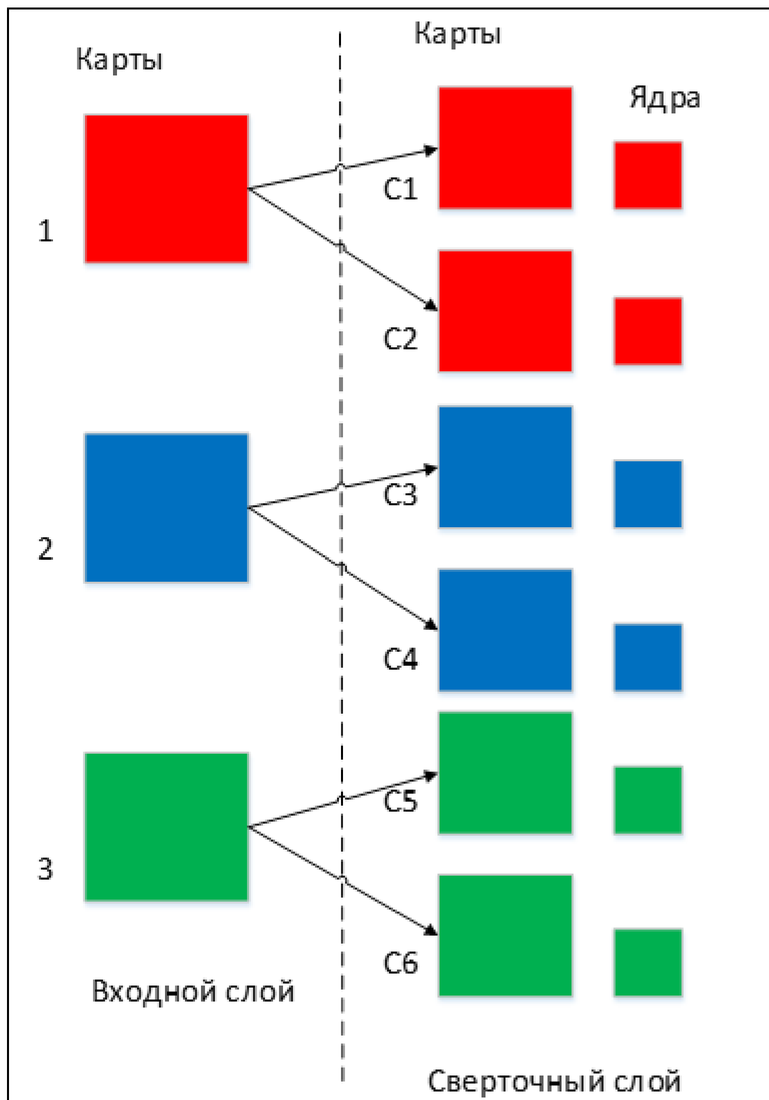


Рисунок 1.3 – Организация связей между картами сверточного слоя и предыдущего.

Ядро представляет из себя фильтр или окно, которое скользит по всей области предыдущей карты и находит определенные признаки объектов. Размер ядра обычно берут в пределах от 3х3 до 7х7. Если размер ядра маленький, то оно не сможет выделить какие-либо признаки, если слишком большое, то увеличивается количество связей между нейронами. Также размер ядра выбирается таким, чтобы размер карт сверточного слоя был четным, это позволяет не терять информацию при уменьшении размерности в подвыборочном слое, описанном ниже.

Ядро представляет собой систему разделяемых весов или синапсов, это одна из главных особенностей сверточной нейросети. В обычной многослойной сети очень много связей между нейронами, то есть синапсов, что весьма замедляет процесс детектирования. В сверточной сети – наоборот, общие веса позволяет сократить число связей и позволить находить один и тот же признак по всей области изображения.

Изначально значения каждой карты сверточного слоя равны 0. Значения весов ядер задаются случайным образом в области от -0.5 до 0.5. Ядро скользит по предыдущей карте и производит операцию свертка.

Свертка изображения

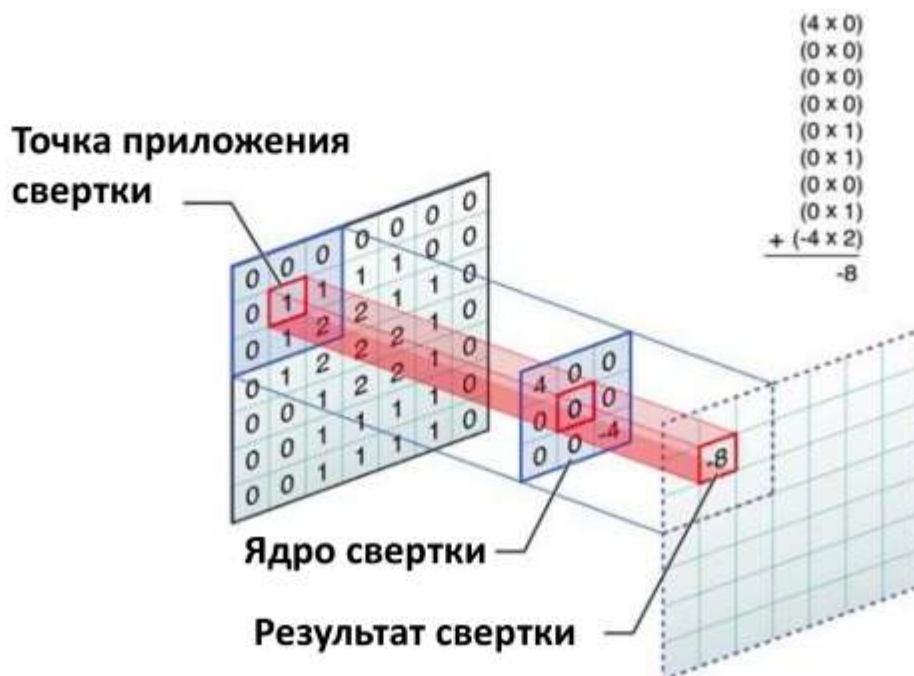


Рисунок 1.4 – Формирование карты признаков сверточного слоя.

При этом в зависимости от метода обработки краев исходной матрицы результат может быть меньше исходного изображения (valid), такого же размера (same) или большего размера (full).

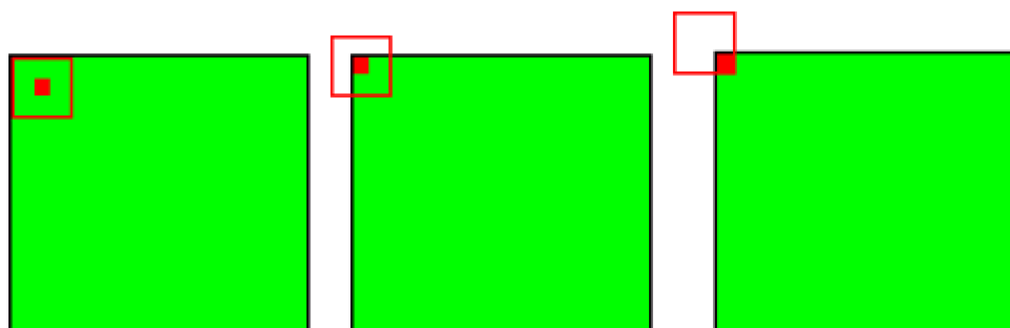


Рисунок 1.5 – Три вида свертки исходной матрицы.

Слой ректификации (RELU, блок линейной ректификации) применяет поэлементную функцию активации, устанавливая нулевой порог.

$$f(s) = \max(0, s) \tag{1.2}$$

Иными словами, RELU выполняет следующие действия: если $s > 0$, то объем остается прежним ([32×32×12]), а если $s < 0$, то осекаются ненужные детали в канале и путем замены на 0.

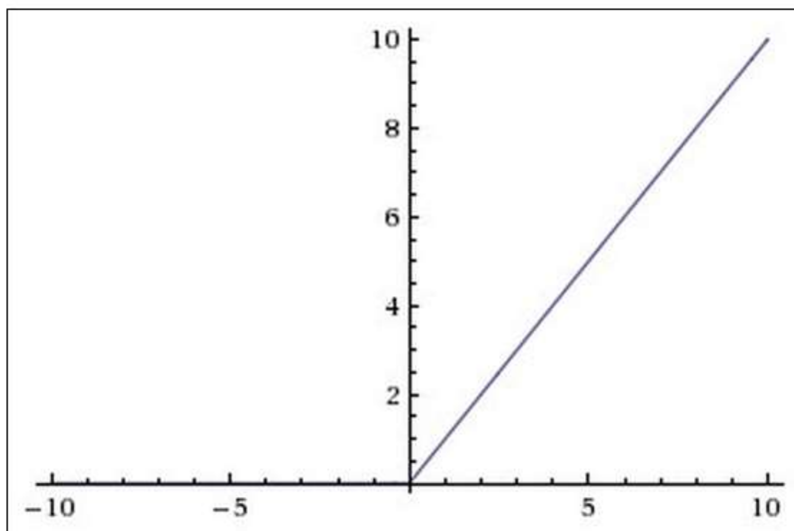


Рисунок 1.6 – График функции активации RELU.

Субдискретизирующий слой (POOL, слой подвыборки) выполняет операцию по понижающей дискретизации пространственных размеров (ширина и высота), в результате чего объем может сократиться до [16×16×12]. То есть на этом этапе выполняется нелинейное уплотнение карты признаков. Логика работы такова: если на предыдущей операции свертки уже были выявлены некоторые признаки, то для дальнейшей обработки настолько подробное изображение уже не нужно, и оно уплотняется до менее подробной картинки.

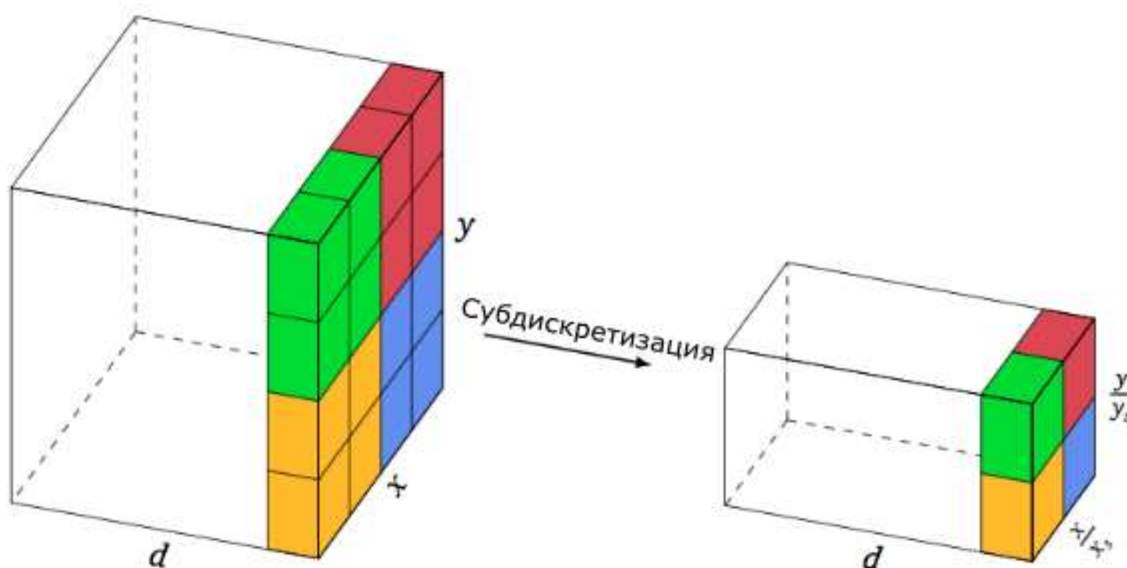


Рисунок 1.7 – Субдискритизирующий слой.

Полносвязный слой (слой FC) выводит N – мерный вектор (N – число классов). Последний из типов слоев это слой обычного многослойного персептрона. Цель слоя – классификация, моделирует сложную нелинейную функцию, оптимизируя которую, улучшается качество распознавания. Работа организуется путем обращения к выходу предыдущего слоя (карте признаков) и определения свойств, которые наиболее характерны для определенного класса.

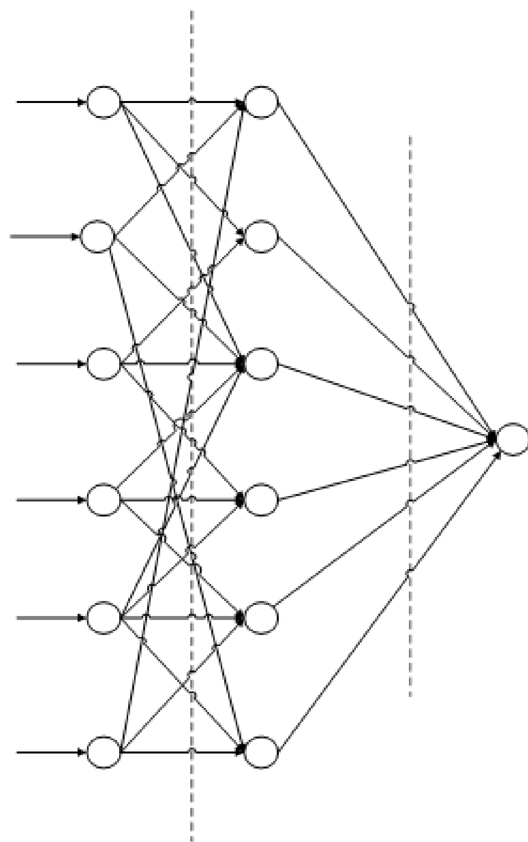


Рисунок 1.8 – Полносвязный слой.

Именно таким образом СНС слой за слоем трансформирует исходное изображение, начиная с исходных значений пикселов и заканчивая определением класса. Обратите внимание, что слои не обязательного должны содержать параметры. В частности, слой свёртки и полносвязный слой выполняют преобразования, которые являются не только функцией от входного активационного объема, но и параметров (веса и смещения нейронов). С другой стороны, блок линейной ректификации и слой пулинга реализуют фиксированную функцию. Параметры в сверточных и полносвязных слоях будут обучаться с помощью градиентного спуска, поэтому определение класса свёрточной нейросетью зависит от меток в обучающем наборе для каждого изображения.

В простейшем случае архитектура СНС – это набор слоев, которые преобразуют образ изображения в выходной образ (например, определение класса).

Каждый слой отвечает за определенный этап процесса обработки изображения (слой свёртки, линейной ректификации, подвыборки и полносвязный слой).

Каждый слой получает на входе объемную з-х мерную информацию и трансформирует с сохранением 3-х мерного объема с помощью дифференцируемой функции.

Слой может не иметь параметров (сверточные и полносвязные слои имеют, слои линейной ректификации, подвыборки – нет).

Слой может не иметь дополнительные гиперпараметры (например, сверточные и полносвязные слои и слои подвыборки имеют, слои линейной ректификации – нет).

Далее идет подробное описание каждого из слоев, изучим их связность между собой и особенности гиперпараметров.

1.2.1 Слой свёртки

Слой свёртки – это основополагающий слой СНС, который выполняет основную работу.

Параметры слоя свёртки состоят из набора обучающихся фильтров. Каждый фильтр имеет малые пространственные габариты (ширину и высоту), но проходит по всей глубине объема ввода. Например, стандартный фильтр первого слоя свёрточной нейронной сети может быть размера [5x5x3]. Во время прохода вперед, мы проскальзываем (точнее, скручиваем) каждый фильтр по ширине и высоте входных данных и вычисляем скалярное произведение между записями фильтра и входом в любое положение. По мере прохождения фильтра по ширине и высоте изображения, мы составляем 2-мерную активационную карту, которая предоставляет отклики этого фильтра на каждой пространственной позиции. Сеть выучивает фильтры, активирующиеся при обнаружении некоторой визуальной особенности. Это может быть грань некоторой направленности, пятнистость конкретного цвета на первом слое или кольцеобразные узоры на более высоких уровнях сети. Теперь мы будем работать с целым набором фильтров в каждом слое свёртки, и каждый из них будет формировать отдельную 2-мерную активационную карту. Мы будем складывать эти активационные карты вдоль измерения «глубина» и формировать выходной объем.

Если вы привыкли проводить аналогии с нейронами, тогда каждую запись в выходном объеме можно интерпретировать как выходной нейрон, который смотрит только на небольшой участок входного объема и пространственно делит

параметры со всеми нейронами слева и справа (поскольку они являются результатом применения такого же фильтра). Теперь давайте обсудим детали связей между нейронами, их расположение в пространстве и модель разделения параметров.

Локальная связность

Когда речь идет о работе с входной информацией с высокой размерностью, установка связи между нейронами и всеми нейронами с прежним объемом является нецелесообразной. Вместо этого будем подключать каждый нейрон только к локальной области входного объема. Пространственная протяженность этой связи является гиперпараметром и называется рецептивым полем (полем восприимчивости). Легко догадаться, что площадь поля восприимчивости равна площади фильтра. Пространственная протяженность вдоль оси глубины всегда эквивалента глубине входного объема. Следует еще раз подчеркнуть асимметрию в том, как мы рассматриваем пространственные измерения (ширину и высоту), а как – глубину: соединения являются локальными по ширине и высоте, но всегда проходят через всю глубину входного объема.

Пример 1. Предположим, что картинка на входе имеет размер [32×32×3] (например, RGB – изображение). Если рецептивное поле (размер фильтра) равно 5×5, тогда каждый нейрон в слое свёртки будет иметь вес в пределах [5x5x3] входного объема, что в итоге даст 5*5*3=75 (+1 параметр смещения). Заметьте, что пространственная протяженность вдоль оси глубины должна быть равна 3: тогда есть гарантия математической верности.

Пример 2. В случае, если входное изображение имеет размер [16×16×20]. Тогда, если используется поле восприимчивости 3×3, каждый нейрон в слое свёртки будет иметь в сумме 180 (3*3*20) соединений с объемом на входе. Снова обращаем ваше внимание на то, что связность является локальной по ширине и высоте (здесь – 3×3), но проходит через всю глубину ввода (20).

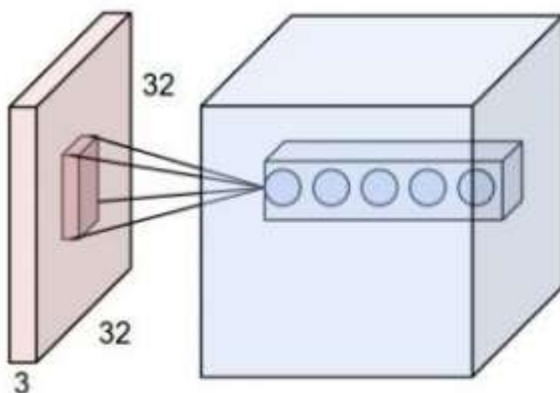


Рисунок 1.9 – Структура сверточного слоя.

Пример входного объема (красный прямоугольник, параметры [32×32×3], изображение) и предположительный объем нейронов в первом слое свёрточной

нейронной сети. Каждый нейрон в свёрточном слое соединен только с локальной входной областью, где пространственная протяженность определяется шириной и высотой, но нейрон проходит через всю глубину, т. е. охватывает все цветовые каналы. Обратите внимание, что существует несколько нейронов (здесь – 5), проходящих через всю глубину и охватывающих одну область на входе (рассмотрение глубины столбцов приводится ниже).

Пространственное расположение

Существуют 3 гиперпараметра, которые контролируют размер выходной информации: глубина, шаг и дополнение нулями.

Начнем с первого гиперпараметра выходного объема – глубины. Он эквивалентен числу фильтров, которые мы бы хотели использовать; каждый из фильтров обучается нахождению различных данных на входе. Например, первый свёрточный слой в качестве входной информации берет необработанное изображение, после различные нейроны, располагающиеся вдоль глубины, могут активироваться в случае наличия, например, сгустков определенного цвета. Будем называть множество нейронов, которые «смотрят» на один участок входного объема, столбцом глубины (некоторые предпочитают термин «волокно»).

Во-вторых, следует указать шаг, с которым фильтр выполняет проход. Когда шаг равен 1, то мы за один раз перемещаем фильтры на 1 пиксель. Когда шаг равен 2 (иногда бывает 3, но это редко используется на практике), фильтры «перепрыгивают» через 2 точки за раз при каждом обращении к ним. Это позволит вырабатывать меньшие выходные объемы пространственных измерений.

Как мы будет сказано, иногда бывает удобно окружать границу входного изображения нулями. Размер этого дополнения нулями и есть гиперпараметром. Ключевой особенностью дополнения нулями является то, что оно позволяет контролировать пространственный размер выходных объемов (чаще всего мы будем использовать данное свойство, чтобы сохранить пространственный размер входной информации, т. е. с целью сохранения входных и выходных значений высоты и ширины одинаковыми).

Пространственный размер выходного объема можно вычислить как функцию от входного объема (W), размера рецептивного поля нейронов свёрточного слоя (F), шага, с которым они перемещаются (S), и количества заполнения нулями на границе входной картинки (P). Вы можете убедиться, что формула $(W - F + 2P)/S + 1$ для подсчета количества «подходящих» нейронов является правильной. Например, для входного объема 7×7 и фильтра 3×3 с шагом 1 и дополнением 0, на выходе мы получим объем 5×5. С шагом 2 выходное значение было бы равно 3×3. Давайте посмотрим на еще один графический пример.

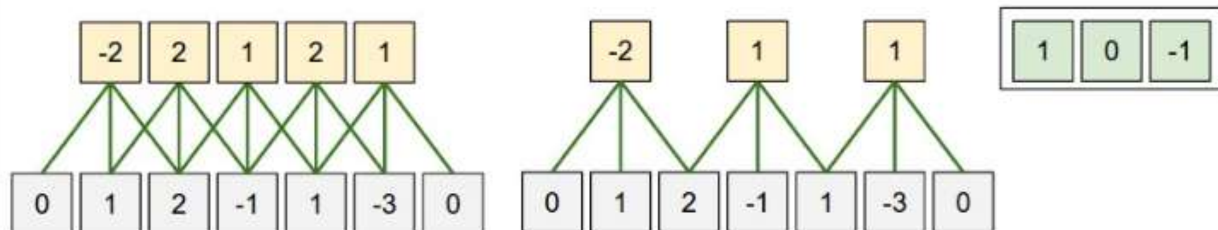


Рисунок 1.10 – Пространственной расположение нейронов.

Иллюстрация пространственного расположения нейронов. В этом примере есть только одно пространственное измерение (ось x), один нейрон с рецептивным полем размера F=3, входной размер W=5 и дополнение нулями P=1. Слева: Нейрон, протянутый вдоль входного объема с шагом S=1, выводит выходной размер (5−3+1)/1+1 = 5. Справа: Нейрон, использующий шаг S=2, обеспечивает выходной размер (5−3+1)/2+1 = 3. Обратите внимание, что шаг S=3 не может быть использован, поскольку он не сможет аккуратно покрыть весь объем. С точки зрения арифметики это определяется так: (5−3+2) = 4 не делится на 3 без остатка. Веса нейронов в данном примере – [1,0,-1] (показано справа на Рисунке 9), поэтому их смещение равно нулю. Эти веса являются общими для всех желтых нейронов (смотрите «Совместное использование параметров» ниже).

Применение дополнения нулями. В приведенном выше примере, входная размерность была равна 5; размер на выходе остался таким же – 5. Это возможно благодаря тому, что размер рецептивных полей был равен 3, а также мы использовали дополнение нулями, равное 1. Если бы дополнение нулями отсутствовало, тогда пространственная размерность выходного объема была вы равна 3, поскольку именно 3 нейрона смогли бы идеально пройти по всему исходному объему. В общем случае, установка дополнения нулями P = (F-1)/2 при шаге S=1 гарантирует, что входной и выходной объемы будут иметь одинаковый пространственный размер. Чаще всего дополнение нулями используется как раз для сохранения исходных показателей ширины и высоты. Причины такого подхода мы рассмотрим более детально в ходе дальнейшего изучения архитектуры СНС.

Ограничения на шаги. Отметим еще раз, что пространственное расположение гиперпараметров имеет взаимные ограничения. Например, когда входной объем имеет размер W=10, дополнение нулями отсутствует (P=0), а размер фильтра F=3, тогда использование шага S=2 будет невозможно, поскольку (W-F+2P)/S+1 = (10−3+0)/2+1 = 4.5, т. е. получается равенство дробному число, указывающему, что нейроны как бы «не вписываются» и не могут расположиться симметрично вдоль входного объема. Таким образом, установка этих гиперпараметров считается неверной, и библиотека ConvNet может создать исключение, произвести дополнение нулями, обрезать входные данные или выполнить другую операцию

для подгонки корректных значений. Как будет показано в разделе «Архитектуры свёрточных нейронных сетей», определение размеров СНС и всех измерений в частности, может стать настоящей проблемой, решение которой значительно облегчается путем использования нулевых дополнений и некоторых принципов проектирования.

Практический пример. Архитектура Крижевского и других, которая выиграла соревнование ImageNet в 2012 году, принимала изображения размера [227×227×3]. На первом свёрточном слое она использовала нейроны с рецептивным полем размера F=11, шагом S=4, а дополнение нулями отсутствовало (P=0). Поскольку (227−11)/4+1 = 55 и слой свёртки имел глубину K=96, выходной объем был размера [55×55×96]. Каждый из 55*55*96 нейронов в этом объеме был соединен с областью входного объема размера [11×11×3]. Кроме того, все 96 нейронов в каждом волокне соединены с той же входной областью размера [11×11×3], но с разными весами. Интересно, что согласно официальной документации, входные изображения были размера 224×224, что, очевидно, неправильно, поскольку результат выполнения всех операций в выражении (224−11)/4+1 не даст целое число. Это смутило многих людей, которые так или иначе затрагивали СНС, и никто до сих пор не знает наверняка, что произошло. Можно предположить, что разработчик использовал дополнение нулями 3 дополнительных пикселей, о чем не упомянул в бумагах.

Совместное использование параметров

Совместное использование параметров используется в слоях свёртки с целью контроля количества параметров. Используя приведенный выше практический пример, замечаем, что в первом свёрточном слое есть 55*55*96 = 290 400 нейронов, и каждый имеет 11*11*3 = 363 весов и 1 смещение. В обще сложности это дает 290400 * 364 = 105 705 600 параметров в первом слое свёрточной нейронной сети. Очевидно, что это очень большое число.

Оказывается, можно значительно уменьшить количество параметров путем следующего предположения: если одно свойство подходит для вычислений в некотором пространственном положении (x, y), то это же свойство должно быть полезно при работе на площадке (x_2, y_2). Другими словами, обозначая один 2-мерный глубинный срез как срез глубины (например, объем размера [55×55×96] имеет 96 срезов глубины, каждый размером [55×55]), мы сдерживаем нейроны каждого среза, чтобы использовать одни и те же веса и смещения. Благодаря схеме разделения параметров, первый слой свёртки в нашем примере теперь будет иметь всего 96 уникальных наборов весов (по одному для каждого среза глубины), что в общей сложности дает 96*11*11*3 = 34 848 уникальных весов или 24 944 параметров (+96 смещений). Все 55*55 нейронов в каждом глубинном срезе теперь

будут использовать одинаковые параметры. На практике во время метода обратного распространения ошибки, каждый нейрон в объеме будет вычислять градиент для его веса, но эти градиенты будут добавлены вдоль каждого среза глубины, и обновлять будут только один набор весов на одном срезе. Обратите внимание: если все нейроны в одном глубинном срезе используют один и тот же весовой вектор, то прямой проход слоя свёртки в каждом глубине среза может быть посчитан как свёртка весов нейрона с объемом входной информации (отсюда и название — свёрточный слой). Именно поэтому он является общим для обозначения набора весов в качестве фильтра (ядра), которое скручено со входом.

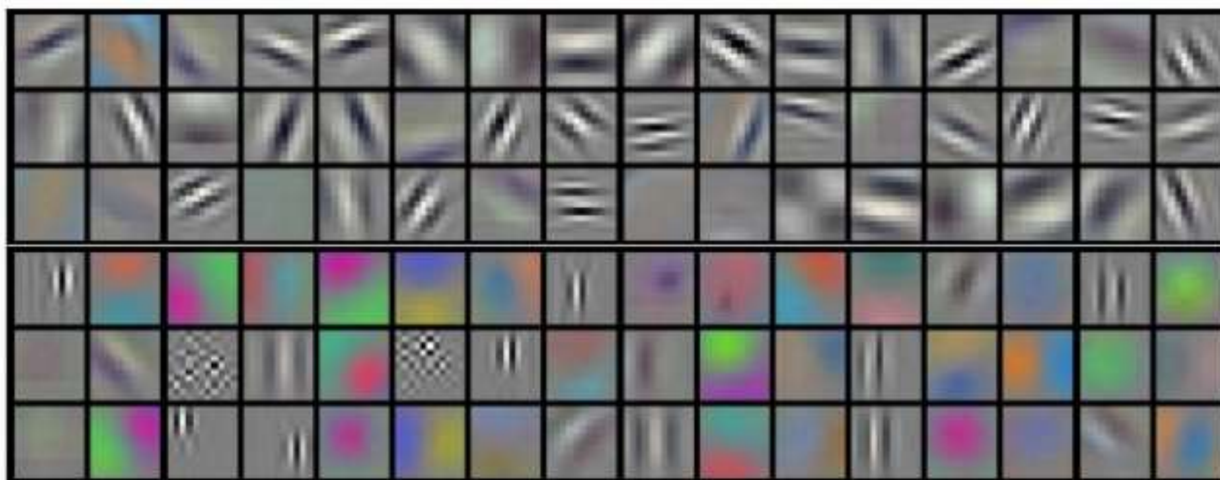


Рисунок 1.11 – Фильтры сверточных слоев

Каждый из представленных здесь 96 фильтров имеет размер [11×11×3], и каждый использует совместно 55*55 нейронов в одном срезе глубины. Заметьте, что совместное использование параметров базируется на логическом предположении: если обнаружение горизонтального ребра важно в определенном месте изображения, оно должно быть полезно и интуитивно понятно на других участках картинки в связи с поступательно-инвариантной структурой изображения. Необходимость переучиваться, чтобы обнаруживать горизонтальное ребро в каждом из 55*55 различных мест выходного объема свёрточного слоя, отсутствует.

Иногда совместное использование параметров лишено смысла. Это касается случая, когда входное изображение, подающееся в СНС, имеет специфическую центрированную структуру, в которой ожидается, к примеру, на противоположных сторонах изображения должно быть произведено изучение совершенно разных свойств. Яркий практический пример: входные данные – лица, которые были центрированы на изображении. Логично ожидать, что свойства, касающиеся глаз, и свойства, касающиеся волос, могут и должны быть изучены в разных точках пространства. В этом случае принято отходить от схемы совместного использования параметров, а вместо нее используют локально соединенный слой.

Резюме по свёрточному слою:

- Принимает данные размера W1xH1xD1
- Требует 4 гиперпараметра:
    1. количество фильтров K,
    2. их пространственную протяженность F,
    3. шаг S,
    4. количественное выражение дополнения нулями P.
- Создает объем размера W2xH2xD2, где:
    1. W2=(W1−F+2P)/S+1
    2. H2=(H1−F+2P)/S+1 (ширина и высота вычисляются в равной степени)
    3. D2=K
- С помощью совместного использования параметров, вводится F·F·D1 весов на фильтр, что в общей сложности дает (F·F·D1)·K весов и K смещений.
- В выходном объеме d-й срез глубины (размера W2xH2) есть результат выполнения корректной свёртки d-го фильтра по входному объему с шагом S с последующей поправкой d-го смещения.

Наиболее распространенные значения гиперпараметров равны: F=3, S=1, P=1. Тем не менее, существуют общие указания и эмпирические правила, которые обосновывают конкретные значения гиперпараметров. Подробнее об этом читайте в разделе «Архитектуры свёрточных нейронных сетей».

Ниже приводится работающая демо-версия сверточного слоя. Поскольку трехмерную информацию тяжело визуализировать, все объемы (входной объем (синий), веса объемов (красный), выходной объем (зеленый)) визуализированы с каждым срезом глубины, расположенным в строках. Входной объем имеет размер W1=5, H1=5, D1=3, параметры свёрточного слоя — K=2, F=3, S=2, P=1. То есть мы имеем 2 фильтра размера 3×3, и они работают с шагом 2. Отсюда пространственный размер выходного объема составляет (5−3+2)/2+1 = 3. Кроме того, дополнение нулями, равное P=1, применяется ко входному объему, делая его внешнюю границу нулевой. Приведенная ниже визуализация перебирает выходные активации (зеленый) и демонстрирует, что каждый элемент вычисляется путем поэлементного умножения выходных данных (синий) и фильтра (красный), последующего суммирования значений и дальнейшей корректировки результатов по смещению.

Реализация в виде умножения матриц

Операция свёртки в основном выполняет скалярное произведение между фильтрами и локальными областями входной информации. Общий принцип реализации свёрточного слоя состоит в том, чтобы воспользоваться указанным

выше фактом и выработать прямой проход свёрточного слоя в виде одной большой матрицы следующим образом:

Локальные области входного изображения растягиваются в столбцы в ходе процесса, который обычно называют im2col. Например, если на вход подается объем размером [227×227×3], который должен быть свёрнут с фильтрами [11×11×3] при шаге 4, необходимо взять блоки пикселов размера [11×11×3] на входе и растянуть каждый блок в вектор-столбец размером 11*11*3 = 363. Повторение этого процесса дает (227−11)/4+1 = 55 мест вдоль измерений ширины и высоты, что приводит к выводу матрицы X_col размера [363×3025], где каждый столбец представляет собой вытянутое рецептивное поле (в общей сложности есть 55*55 = 3 025 рецептивных полей).

Веса слоя свёртки аналогично растянуты в строки. Например, если есть 96 фильтров размера [11×11×3], то это даст матрицу W_row размера [96×363].

Результат выполнения свёртки теперь эквивалентен результату выполнению одной большой матрицы умножения np.dot (W_row, X_col), которая вычисляет скалярное произведение между каждым фильтром и каждым рецептивным полем. В нашем примере результатом этой операции будет [96×3025], что дает на выходе скалярное произведение каждого фильтра в каждой локации.

Результат должен быть приведен к исходному виду относительно измерений: [55×55×96].

Такой подход имеет весомый недостаток: ввиду многократного повторения некоторых значений в матрице X_col, память не всегда используется целесообразно. С другой стороны, главное преимущество заключается в том, что существует множество эффективных умножения матриц, которые можно использовать в своих интересах (практикуется в BLAS API). Также идея функционирования im2col может быть использована для реализации подвыборки, о котором речь пойдет далее.

Метод обратного распространения ошибки

Обратный проход для операции свёртки (как для данных, так и для весов) — это также свёртка, только с пространственно-перевернутыми фильтрами. Это легко выводится на 1-мерном игровом примере.

Свёртка 1x1

В качестве отступления скажем, что в ряде работ (например, Network in Network) была использована свёртка формата 1×1. Обычно сигналы имеют 2 измерения, поэтому свёртки 1×1 бессмысленны (это просто точечное масштабирование). Однако в СНС это не так, ведь нужно помнить, что работа происходит с 3-мерными данными, и что фильтры всегда проходят по всему размеру по глубине входного объема. Так, если на входе имеется изображение

[32×32×3], то свёртки 1×1 будут эффективно выполнять 3-мерное скалярное произведение (поскольку входная глубина состоит из 3 каналов).

Расширенные свёртки

Новая разработка Фишера Ю и Владлена Колтуна [16] открыла еще один гиперпараметр слоя свёртки – дилатацию (расширение). Ранее расматривались фильтры свёрточного слоя, которые являются смежными. Тем не менее, могут быть фильтры, имеющие между каждой ячейкой пробелы. Эти проблемы называются расширениями. Пример: в одном измерении фильтра w размера 3 над входом x будет проводиться операция w[0]*x[0] + w[1]*x[1] + w[2]*x[2]. Это дилатация нуля. Для дилатации 1, будут выполнены следующие вычисления: w[0]*x[0] + w[1]*x[2] + w[2]*x[4]. Другими словами, существует разрыв 1. Использования сочетания с 0-расшриненными фильтрами может быть полезно в отдельных настройках, поскольку это позволяет объединить пространственную информацию входного объема с использованием меньшего количества слоев. Например, если вы расположите два слоя свёртки размером 3×3 наверху каждого из фильтров, будет видно, что нейроны 2-го слоя являются функцией области входа 5×5 (можно сказать, что эффективное рецептивное поле этих нейронов — 5×5). Если мы будем использовать расширенные свёртки, то это эффективное рецептивное поле будет расти гораздо быстрее.

1.2.2 Субдискретизирующий слой (слой подвыборки)

В архитектуре СНС обычной практикой является вставка слоя субдискретизации между последовательностями свёрточных слоев. Его функция состоит в постепенном уменьшении пространственные габариты изображения с целью уменьшения количества параметров и вычислений в сети, а также контроля переобучения. Субдискретизирующий слой работает независимо от среза глубины входных данных и масштабирует объем пространственно, используя функцию максимума. Чаще всего используется слой с фильтрами размера 2×2 с шагом 2; подобный слой снижает дискретизацию каждого среза глубины входа в 2 раза как по ширине, так и по высоте, откидывая при этом 75% активация. Каждая операция MAX в этом случае будет выбирать максимальное значение из 4 чисел. Размер по глубине при этом остается неизменным.

Рисунок 1.12 – Субдискритизирующий слой (подвыборка).

В общем случае субдискретизирующий слой:

Получает объем на входе размером W1xH1xD1

Требует 2 гиперпараметра:

F, протяженность;

S, шаг.

Вводит нулевые параметры, поскольку производится вычисление фиксированной функции ввода.

Следует обратить внимание, что дополнение нулями не принято использовать в подвыборке. На практике можно увидеть только две вариации слоя подвыборки: слой с гиперпараметрами F=3, S=2 (также называют перекрывающей подвыборкой) и еще более распространенный слой F=2, S=2. Размеры подвыборки с большими рецептивными полями являются деструктивными.

Общиая подвыборка

В дополнение к максимальной подвыборке, субдискретизирующие слои могут выполнять другие функции, например, усредняющую подвыборку или даже L2-нормированную подвыборку. Исторически усредняющая подвыборка использовалась довольно часто, но в последнее время она отошла на второй план по сравнению с максимизационной подвыборкой, которая на практике работает лучше.

Метод обратного распространения ошибки.

Рассматривая свёрточный слой, мы уже говорили о том, что обратный проход для операции max(x, y) интерпретируется как градиент на вход, который имел наибольшее значение при прямом проходе. Следовательно, при прямом проходе слоя подвыборки общепринятым является отслеживание индекса максимальной активации (его иногда называют переключателем) так, чтобы градиент пути был эффективен во время обратного распространения.

Избавление от субдискретизации (пулинга)

Отдельные специалисты считают, что можно обойтись без операцию пулинга. Например, Д. Т. Спрингенберг, А. Досовицкий, Т. Брокс и М. Ридмиллер в своей работе [17] предложили отказаться от слоя пулинга в пользу архитектуры, которая состоит исключительно из повторяющихся слоев свёртки. Чтобы уменьшить размер представления, они предлагают использовать больший шаг в свёрточных слоях. Отказ от субдискретизации может сыграть важную роль в обучении генеративных моделей вроде вариационных автоасоциаторов (VAE) или генеративных состязательных сетей (GAN). Похоже, что уже в ближайшем будущем архитектура типовой свёрточной нейронной сети будет лишена слоев подвыборки, либо они будут представлены в очень малом количестве.

1.2.3 Полносвязный слой

Точно так же, как и в обычных нейросетях, в полносвязном слое нейроны обладают полными соединениями со всеми активациями в предыдущем слое. Их активации могут быть вычислены посредством умножения матриц, сопровождающегося смещением.

Преобразование полносвязных слоев в свёрточные

Различия между полносвязными и свёрточными слоями состоят в том, что нейроны слоя свёртки соединены только с локальной областью на входе, и что нейроны этого слоя могут совместно использовать параметры. Однако нейроны в обоих слоях, несмотря на свои особенности, подсчитывают скалярное произведение, поэтому их функциональная форма идентична. Более того, оказывается, можно выполнить конвертацию между полносвязным и свёрточным слоем:

Для любого слоя свёртки существует полносвязный слой, который реализует ту же функцию прохода. Матрица весов будет представлять собой большую матрицу, которая в основном заполнена нулями за исключением некоторых блоков (из-за локальной связности), веса большинства из которых будут равными (из-за совместного использования параметров).

И наоборот: любой полносвязный слой может быть преобразован в слой свёртки. Например, полносвязный слой с глубиной K=4 096, направленный на некоторый входной объем размера [7x7x512] может быть выражен как слой свёртки со следующими параметрами: размер фильтра F=7, дополнение нулями отсутствует (P=0), шаг S=1 и глубина фильтра K=4 096. Другими словами, мы устанавливаем размер фильтра точно такой же, как и размер входного объема. На

выходе мы получим [1x1x4096], потому что только один столбец глубины «правильно» проходит по всем входным данным, давая точно такой же результат, как и исходный полносвязный слой.

Роль преобразований

Возможность преобразовывать полносвязные слои в свёрточные и наоборот широко используется на практике. Рассмотрим свёрточную нейронную сеть, которая принимает на входе изображение [224×224×3], а затем использует ряд слоев свёртки и подвыборки, чтобы уменьшить изображение до активационного объема в размере [7x7x512]. В архитектуре AlexNet, которую мы рассмотрим позже, это делается путем использования 5 слоев подвыборки, которые понижают дискретизацию входных пространственных измерений, благодаря чему окончательный пространственный размер равен 224/2/2/2/2/2 = 7. Архитектура AlexNet использует 2 полносвязных слоя размером 4096 и еще 1 полносвязный слой с 1000 нейронов, который подсчитывает оценки классов. Мы можем преобразовать любой из этих 3 слоев в свёрточный по описанному алгоритму:

Заменить первый полносвязный слой, направленный на участок объема [7x7x512], свёрточным слоем с фильтром размера F=7, который обеспечит выходной объем [1x1x4096].

Заменить второй полносвязный слой свёрточным слоем фильтром размера F=1, который обеспечит выходной объем [1x1x4096].

Заменить последний полносвязный слой свёрточным слоем фильтром размера F=1, который обеспечит выходной объем [1x1x1000].

Каждое из этих преобразований может включать в себя определенные манипуляции (например, изменение формы) матрицы весов W в каждом полносвязном слое в фильтрах слоя свёртки. Оказывается, что эти преобразования позволяют «протаскивать» за один прямой проход исходную СНС через многие пространственные положения в более крупных изображениях.

К примеру, если картинка размера 224×224 дает объем размера [7x7x512], т. е. производит сокращение на 32, тогда пересылка изображения с размером 384×384 через преобразованную архитектуру даст эквивалентный объем в размере [12×12×512], поскольку 384/32 = 12. Вместо одного вектора оценки класса размера [1x1x1000], теперь мы получаем целый массив оценки класса 6×6 по всему изображению размера 384×384.

Вычисление исходной свёрточной нейросети (с полносвязными слоями), независимо проходящей через 224x224 изображения размером 384×384 с шагом в 32 пиксела, дает такой же результат, как и едино разовое прохождение конвертированной СНС.

Естественно, едино разовое прохождение преобразованной нейронной сети куда эффективнее, нежели многократное итерирование оригинальной СНС по всем 36 позициям. Данная особеность часто используется на практике с целью повышения производительности. Так, когда необходимо изменить размер изображения в большую сторону, используют конвертированную свёрточную нейронную сеть, чтобы дать характеристику оценке класса на многих пространственных позициях, а затем ее усреднить.

В случае если необходимо эффективно применить исходную СНС на изображении, с шагом меньшим 32 пикселей, это можно осуществить с помощью перемножения прямых проходов. Например, если мы хотим использовать шаг в 16 пикселей, то требуется объединить объемы, полученные путем прохода преобразованной сёрточной нейросети, дважды: сначала по исходному изображению, потом — также по изображению, но сдвинутому на 16 пискелей как по ширине, так и высоте.

1.2.4 Принципы построения сверточных нейронных сетей

Мы увидели, что свёрточные нейронные сети обычно состоят из 3 основных типов слоев: свёрточных, пулинга и полносвязных. Будем также явно записывать функцию активации ReLU в качестве слоя, в котором используется поэлементная нелинейность. Далее представлено, как эти слои взаимодействуют вместе, образуя целостную СНС.

Паттерны слоев

Чаще всего при построении архитектуры СНС складывают несколько слоев слоев свёртки и ректификации (ReLU), за ними следуют слои подвыборки; этот шаблон продолжает работать, пока изображение не будет объединено до малого размера. В некоторый момент осуществляется переход к полносвязным слоям. Последний полносвязный слой содержит выходную информацию, например, оценки класса. Другими словами, наиболее распространенная архитектура СНС соответствует схеме:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

Рисунок 1.13 – Группировка слоев сверточной нейронной сети.

Здесь «*» означает повторение, а POOL указывает на дополнительный слой подвыборки. Кроме того, $N >= 0$, (обычно $N <= 3$), $M >= 0, K >= 0$ (обычно $K < 3$). Ниже представлены архитектуры СНС, организованные по этому образцу:

```
INPUT -> FC, реализует линейный классификатор. Здесь N = M = K = 0.
```

```
INPUT -> CONV -> RELU -> FC
```

```
INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC. Здесь один свёрточный
слой — между каждый слоем пулинга.
```

```
INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC. Здесь 2
слоя свёртки укладываются перед каждым слоем пулинга.
```

Рисунок 1.14 – Построение сверточной нейронной сети.

Это хорошая реализация для больших и более глубоких сетей, поскольку несколько сложенных слоев свёртки могут развивать более сложные свойства входных данных перед деструктивной операцией подвыборки.

Как правило, отдают предпочтение стеку слоев с небольшими фильтрами, нежели одному большому рецептивному полю свёрточного слоя. Предположим, соблюдая нелинейность между слоями, выкладывается 3 слоя свёртки размера 3×3 друг на друга. При таком расположении, каждый нейрон на первом свёрточном слое обладает видом 3×3 входного объема. Нейроны на втором слое обладают видом 3×3 первого слоя и 5×5 входного объема. Точно так же нейроны на третьем слое обладают видом 3×3 второго слоя и 7×7 входного объема. Допустим вместо этих трех свёрточных слоев, мы хотим использовать только один CONV-слой с рецептивными полями размера 7×7. Нейроны такого слоя будут иметь рецептивное поле входного объема, аналогичное пространственной протяженности (7×7), но с некоторыми недостатками. Во-первых, нейроны будут вычислять линейную функцию входного объема, в то время как стек из 3 слоев свёртки содержит нелинейности, подчеркивающие свойства слоев. Во-вторых, если мы предположим, что весь входной объем имеет C каналов, тогда один свёрточный слой размера 7×7 будет содержать $C * (7 * 7 * C) = 49 * C^2$ параметров, в то время как стек из 3 слоев содержит лишь $3 * (C * (3 * 3 * C)) = 27 * C^2$ параметров.

Очевидно, что «укладка» нескольких CONV-слоев с небольшими фильтрами позволяет подчеркнуть более мощные особенности входной информации с меньшим количеством параметров. Недостатком является больший объем используемой памяти для сохранения всех промежуточных результатов.

VGGNet в деталях

Рассмотрим VGGNet более детально. Вся сеть состоит из свёрточных слоев, которые выполняют свёртывание 3×3 с шагом 1 и дополнением 1, и слое подвыборки, выполняющих максимальную подвыборку 2×2 с шагом 2 без дополнения нулями. Можно выписать размер представления на каждом шаге и отслеживать как его изменение, так и изменение количества весов:

```
INPUT: [224x224x3] память: 224*224*3=150K веса: 0

CONV3-64: [224x224x64] память: 224*224*64=3.2M веса: (3*3*3)*64 = 1,728

CONV3-64: [224x224x64] память: 224*224*64=3.2M веса: (3*3*64)*64 = 36,864

POOL2: [112x112x64] память: 112*112*64=800K веса: 0

CONV3-128: [112x112x128] память: 112*112*128=1.6M веса: (3*3*64)*128 = 73,728

CONV3-128: [112x112x128] память: 112*112*128=1.6M веса: (3*3*128)*128 = 147,456

POOL2: [56x56x128] память: 56*56*128=400K веса: 0

CONV3-256: [56x56x256] память: 56*56*256=800K веса: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] память: 56*56*256=800K веса: (3*3*256)*256 = 589,824

CONV3-256: [56x56x256] память: 56*56*256=800K веса: (3*3*256)*256 = 589,824

POOL2: [28x28x256] память: 28*28*256=200K веса: 0

CONV3-512: [28x28x512] память: 28*28*512=400K веса: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] память: 28*28*512=400K веса: (3*3*512)*512 = 2,359,296

CONV3-512: [28x28x512] память: 28*28*512=400K веса: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] память: 14*14*512=100K веса: 0

CONV3-512: [14x14x512] память: 14*14*512=100K веса: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] память: 14*14*512=100K веса: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] память: 14*14*512=100K веса: (3*3*512)*512 = 2,359,296

POOL2: [7x7x512] память: 7*7*512=25K веса: 0

FC: [1x1x4096] память: 4096 веса: 7*7*512*4096 = 102,760,448
```

Рисунок 1.15 – Потребление памяти при построении нейронной сети.

Обратите внимание, что большая часть памяти используется в самых первых слоях свёртки, а максимальное количество параметров находится в последних полносвязных слоях. В данном конкретном случае первый полносвязный слой содержит 100 млн весов, в то время как общая сумма составляет 140 млн.

2 АНАЛИЗ МОДЕЛИ СВЕРТОЧНОЙ НЕЙРОНОЙ СЕТИ И СПОСОБОВ ЕЕ ОПТИМИЗАЦИИ

2.1 Математическая модель сверточной нейронной сети.

Для описания математической модели рассматриваемой нейронной сети будем использовать следующие обозначения.

Под $l \in [1; L]$будем понимать рассматриваемый в данный момент слой нейронной сети, где $L = 2a + 2, a \in \mathbb{Z}^+$- количество слоев в сети. За N^lобозначим количество карт признаков на слое l, а за $f(\cdot)$- функцию активации рассматриваемого слоя l. Также, под переменной y_n^l будем понимать n-ую карту признаков на слое l.

2.1.1 Математическая модель сверточного слоя

Введем в рассмотрение сверточный слой l. В подобной архитектуре нейронной сети I принимается нечетным числом, то есть $l = 1,3,\ldots,2a + 1$. Тогда, д.ля карты признаков n будет иметь место следующее:

- $\mathbb{w}_{m,n}^l = \{w_{m,n}^l(i,j)\}$ - свертка, применяемая к карте признаков m слоя $(l-1)$, на слое l с картой признаков n;
- b_n^l -пороговые значения, присоединяемые к карте признаков n на слое l;
- V_n^l - список всех уровней слоя $(l-1)$, которые соединяются с картой признаков n слоя l.

Таким образом, карта признаков n сверточного слоя l будет вычисляться следующим образом:

$$y_n^l = f_l(\textstyle\sum_{m \in V_n^l} y_n^{l-1} \otimes \mathbb{w}_{m,n}^l + b_n^l), \quad (2.1)$$

где под оператором $\otimes$ понимается математическая операция двумерной свертки.

Предположим, что размер входных карт признаков y_m^{l-1} равен $H^{l-1} \times W^{l-1}$, а размер применяемой к ним свертки $\mathbb{w}_{m,n}^l$ равняется $r^l \times c^l$, тогда размер выходной карты признаков y_n^l вычисляется как (рис.2.1):

$$(H^{l-1} - r^l + 1) \times (W^{l-1} - c^l + 1). \quad (2.2)$$

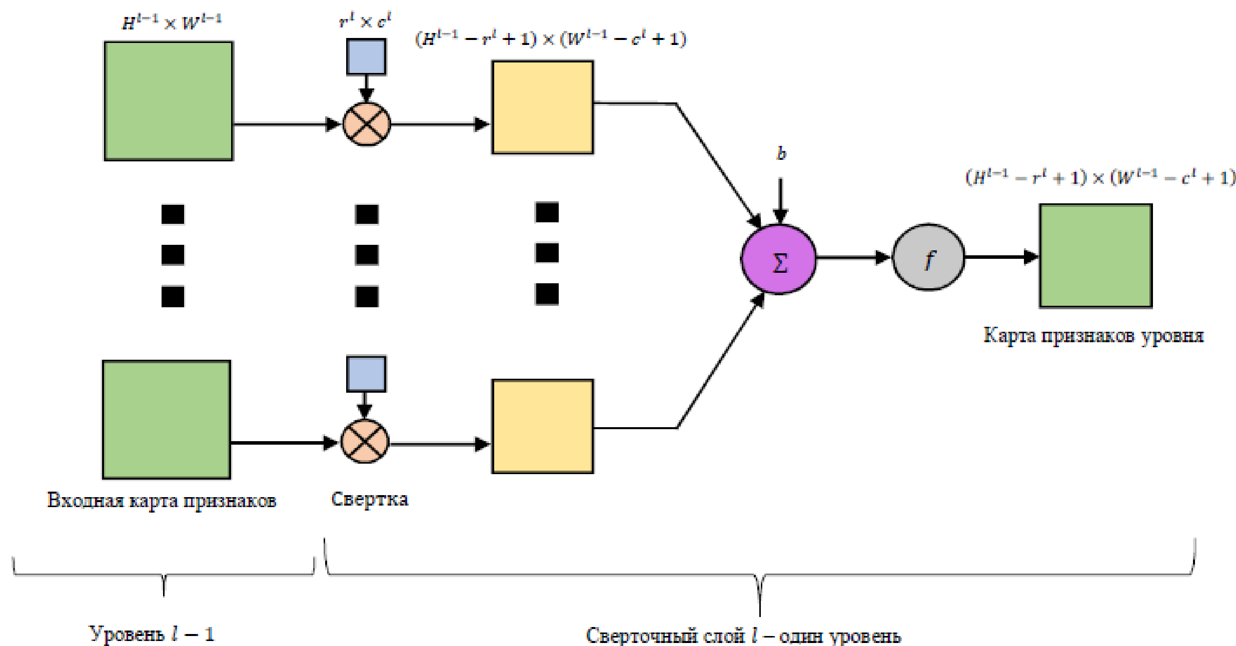


Рисунок 2.1 – Схема сверточного слоя l.

2.1.2 Математическая модель субдискретизирующего слоя

Введем в рассмотрение субднскретнзирующнй слой l. В сверточной нейронной сети l принято принимать четным числом, то есть $l = 2,4,\dots,2a$. Для карты признаков n введем следующие обозначения: $\mathbb{w}_{m,n}^{k}$- фильтр, применяемый к n на слое l и b_n^l- добавочное пороговое значение.

Далее будем действовать следующим образом: разделим карту признаков $n(l-1)$ - ого слоя на непересекающиеся блоки размером 2×2 пикселя. Затем просуммируем значения четырех пикселей в каждом блоке и в результате получим матрицу $\mathbb{z}_n^{l-1} = \{z_n^{l-1}(i,j)\}$, элементами которой будут являться соответствующие значения сумм. Таким образом, формула для вычисления значений элементов матрицы будет иметь следующий вид:

$$z_n^{l-1} = y_n^{l-1}(2i-1,2j-1) + y_n^{l-1}(2i-1,2j) + y_n^{l-1}(2i,2j-1) + y_n^{l-1}(2i,2j)$$

Карта признаков n субдискретизирующего слоя l вычисляется, как:

$$y_{1,}^{L} = f_l(\mathbb{z}_n^{l-1} \times \mathbb{w}_{m,n}^{l} \times b_n^l). \quad (2.3)$$

Благодаря представленным выше рассуждениям, становится возможным посчитать размер $H^l \times W^l$ карты признаков y_n^l субдискретизирующего слоя l (рис. 6):

$$H^l = \frac{H^{l-1}}{2}, W^l = \frac{W^{l-1}}{2}. \quad (2.4)$$

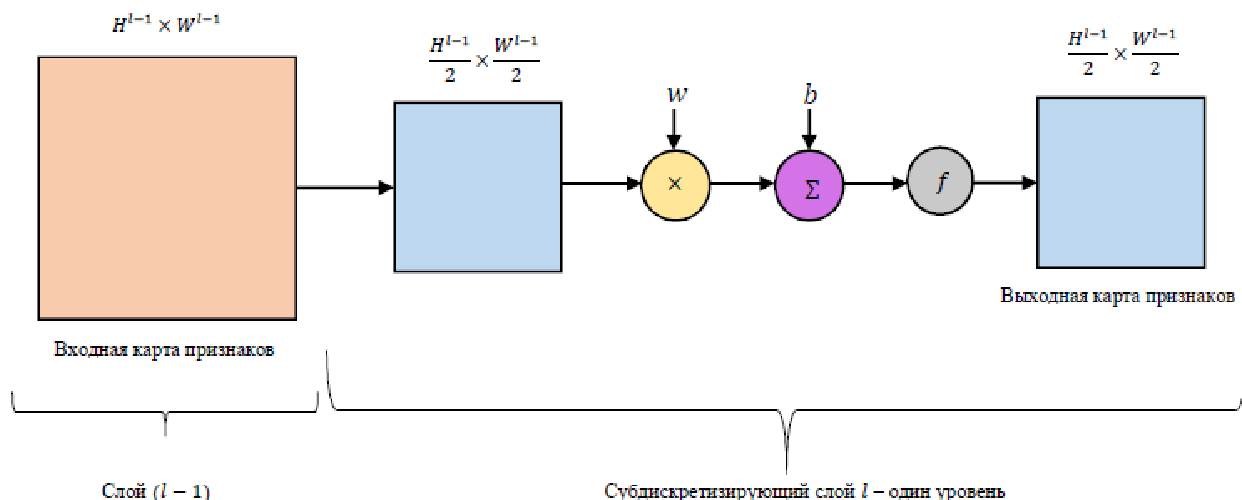


Рисунок 2.2. Схема субдискретизирующего слоя l.

2.1.3 Математическая модель выходного слоя

В данном параграфе будем рассматривать выходной слой L. состоящий из единичных нейронов. Примем за N^l - количество нейронов на данном слое. Как и при рассмотрении предыдущих слоев, обозначим за $\mathbb{w}^L_{m,n}$ - фильтр, применяемый к карте признаков m последнего сверточного слоя для получения перехода к нейрону n выходного слоя. Пусть b^L_n - пороговое значение, добавляемое к нейрону n.

Пользуясь введенными обозначениями, получаем формулу для подсчета значения выходного нейрона n:

$$y^L_n(i,j) = f_L(\sum_{m=1}^{N^{L-1}} y^{L-1}_n \mathbb{w}^L_{m,n} + b^L_n). \qquad (2.5)$$

Таким образом, выходом сверточной нейронной сети является вектор следующего вида:

$$\mathbb{y} = [y^L_1, y^L_2, \dots, y^L_{N^L,}]. \qquad (2.6)$$

2.1.4 Обучение сверточной нейронной сети

Основной целью обучения нейронных сетей, предназначенных для решения различных задач распознавания, яв.ляется минимизация функции ошибки в течение нескольких итераций (эпох).

Далее будем предполагать, что обучающее множество содержит P изображений. Обозначим за Z^p- p-ое изображение тренировочного множества, а за d^p_n- желаемый p-ый выход системы. Тогда, функция ошибки сети опреде.ляется как:

$$E(\mathbb{w}) = -d^p_n \cdot \log y^p_n \qquad (2.7)$$

где y^p_n - выход нейронной сети.

Вычисление градиента функции ошибки сети.

Для минимизации рассмотренной выше функции ошибки (2.7) необходимо вычислить ее градиент, что требует знания частных производных по основным параметрам сети.

Таким образом, для каждого нейрона (i, j) карты признаков n слоя $l, l = 1, \dots L$, частная производная по взвешенной сумме $s_n^{l,p}(i, j)$, где

$n = 1, \dots, N^l, i = 1, \dots, H^l, j = 1, \dots, W^l$ имеет вид:

$$\delta_n^{l,p}(i, j) = \frac{\partial E}{\partial s_n^{l,p}(i,j)} \quad (2.8)$$

Используя цепное правило дифференцирования, получаем возможность выразить производную *(2)* через функцию активации слоя.

Д.ля выходного слоя L *(2)* будет иметь вид:

$$\delta_n^{l,p} = \frac{1}{P \times N^l} \cdot e_n^p \cdot f'(s_n^{l,p}), \quad (2.9)$$

где $e_n^p = y_n^{L,p} - d_n^p$

Для сверточного слоя $l = 2a + 1$ *(2)* будет иметь вид:

$$\delta_n^{l,p}(i, j) = f_i'\left[s_n^{l,p}(i, j)\right] \times \delta_n^{l+1}(i, j) \times \mathbb{w}_{m,n}^{l+1,p}.$$

Для субднскретизирующего слоя *I = 2a (2)* будет иметь вид:

$$\delta_n^{l,p}(i, j) = f_i'\left[s_n^{l,p}(i, j)\right] \times \sum_{m \in U_n^l} \sum_{(i',j') \in R^l(i,j)} \delta_m^{l+1}(i', j') \times \mathbb{w}_{m,n}^{l+1,p}(i - i', j - j'), \quad (2.10)$$

где $U_n^l = \{1,2, \dots, N^{l-2}\}, R^l(i, j) = \{i = 1,2, \dots, H^l; j = 1,2, \dots, W^l\}, i' = \left[\frac{i}{2}\right], j' = \left[\frac{j}{2}\right]$

Используя вычисленные производные (2.8), (2.9), (2.10) получаем возможность вычислить градиент функции ошибки сети *(1)* для каждого слоя по параметрам $\mathbb{w}_{m,n}^l$и b_n^l.

Градиент для выходного слоя

$$\frac{\partial E}{\partial \mathbb{w}_{m,n}^l} = \sum_{p=1}^P \delta_n^{l,p} y_m^{l-1,p}; \ \frac{\partial E}{\partial b_n^l} = \sum_{p=1}^P \delta_n^{l,p}, \quad (2.11)$$

где $m = 1,2, \dots, N^{l-1}, n = 1,2, \dots, N^l$.

Градиент для сверточного слоя

$$\frac{\partial E}{\partial \mathbb{w}_{m,n}^l} = \sum_{p=1}^P \delta_n^{l,p}(i, j) \cdot y_m^{l-1,p}; \ \frac{\partial E}{\partial b_n^l} = \sum_{p=1}^P \sum_{(i,j)} \delta_n^{l,p}(i, j), \quad (2.12)$$

где $m = 1,2, \dots, N^{l-1}, n = 1,2, \dots, N^l$.

Градиент д.тя субдисретизируюшего слоя

$$\frac{\partial E}{\partial \mathbb{w}_{m,n}^l} = \sum_{p=1}^P \sum_{(i',j')} \delta_n^{l,p}(i, j) \cdot y_n^{l-1,p}(i', j'); \quad (2.13)$$

$$\frac{\partial E}{\partial b_n^l} = \sum_{p=1}^P \sum_{(i',j')} \delta_n^{l,p}(i', j'),$$

где $m = 1,2, \dots, N^{l-1}, n = 1,2, \dots, N^l, i' = \left[\frac{i}{2}\right], j' = \left[\frac{j}{2}\right]$.

2.1.5 Метод обратного распространения ошибки

В основе метода лежит стохастический градиентный спуск: настройка параметров системы для достижения минимума функции ошибки производится после одного случайно выбранного обучающего образна, предъявленного сети. То есть, обновление весов осуществляется сразу после вычисления градиента для текущего примера. Таким образом, минимизация осуществляется в направлении антиградиента:

$\Delta \mathbb{w} = -a \cdot \nabla E^{p}$,

где $a > 0$ - скорость обучения сети.

Пользуясь выведенными в пункте 3.1. значениями производных, распишем алгоритм подробнее.

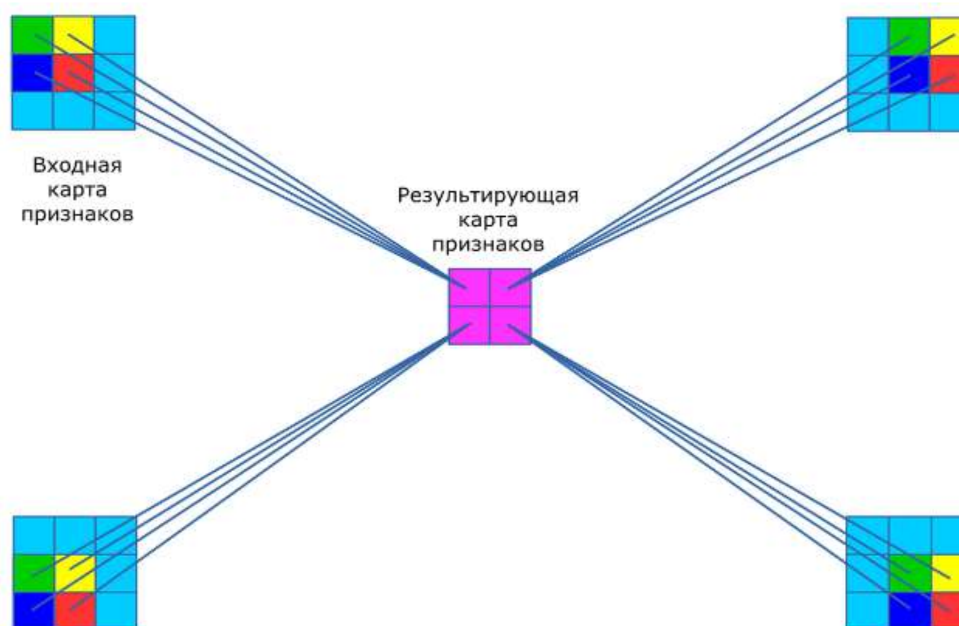


Рисунок 2.3 – Формирование карт признаков при прямом прохождении.

На первом шаге задаем произвольные значения весовых коэффициентов. Затем, передаем сигнал в сети в прямом направлении: от предыдущих слоев к последующим. На этом же шаге вычисляем значение функции ошибки для выходного слоя.

δ_{22}	δ_{21}
δ_{12}	δ_{11}

Рисунок 2.4 – Матрица ошибок.

Вторым шагом обучения сети является распространение ошибки, полученной на первом этапе, в обратном направлении, то есть от последнего слоя к предыдущим. При этом происходит корректировка весов и пороговых значений каждого входа нейрона с использованием градиентного спуска. То есть, вычисляем новые значение весов и порогов:

$$\mathbb{w}' = \mathbb{w} + a \cdot \frac{\partial E}{\partial \mathbb{w}};\ b' = b + a \cdot \frac{\partial E}{\partial b}, \tag{2.14}$$

где $\frac{\partial E}{\partial \mathbb{w}}, \frac{\partial E}{\partial b}$ вычесляются по формулам (2.11),(2.12),(2.13)

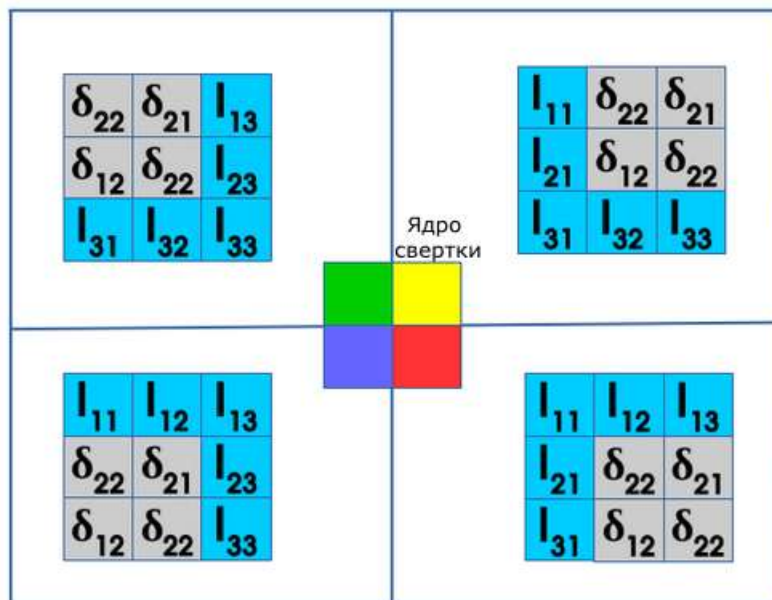


Рисунок 2.5 – Обратное распространение ошибки.

2.2 Механизмы настройки сверточной нейронной сети

2.2.1 Выбор стратегии обучения

Обучение СНС можно производить тремя методами: стохастический метод (stochastic), пакетный метод (batch) и мини-пакетный метод (mini-batch).

Стохастический (его еще иногда называют онлайн) метод работает по следующему принципу - нашел ΔW, сразу обновляется соответствующий вес.

Пакетный метод работает несколько иначе. Суммируется ΔW всех весов на текущей итерации и только потом обновляем все веса используя эту сумму. Один из самых важных плюсов такого подхода — это значительная экономия времени на вычисление, точность же в таком случае может сильно пострадать.

Мини-пакетный метод является наиболее оптимальным и пытается совместить в себе плюсы обоих методов. Используется следующий принцип: в свободном порядке распределяются веса по группам и меняются на сумму Δw всех весов в той или иной группе.

О разнице, плюсах и минусах данных подходов написано достаточно много, вкратце можно выделить следующие:

— Пакетный спуск хорош для строго выпуклых функций, потому что уверенно стремится к минимуму глобальному или локальному.

— Стохастический в свою очередь лучше работает на функциях с большим количеством локальных минимумов — каждый шаг есть шанс, что очередное значение «выбьет» из локальной ямы и конечное решение будет более оптимальным, нежели для пакетного спуска.

— Стохастический вычисляется быстрее — на каждом шаге нужны не все элементы из выборки. Вся выборка целиком может не влезть в память. Но требуется больше шагов.

— Для стохастического легко добавить новые элементы во время работы («онлайн» обучение).

— В случае mini-batch, можно также векторизовать код, что значительно ускорит его выполнение.

2.2.2 Градиентные методы обучения первого порядка

Обучение нейронной сети, это настройка весов W в соответствии с учебным множеством *(X,C)* и важным элементом этой процедуры является способ оценки работы сети или функция потери (loss function) E.

$$h: X \times W \to Y$$
$$E: Y \times C \to R, \quad (2.15)$$

где h – классификатор;

W - веса сети:

$X \ni (x_1 \dots, x_n)$ - пространство признаков размерности n;

E - функция потери;

Y - выход классификатора;

C - множество правильных ответов (номеров классов).

Введя функцию потери E, можно формально поставить задачу обучения классификатора h следующим образом - процедура обучения нейронной сети - это минимизация функции потери в пространстве весов.

$$\min_W (h(X, W), C) \quad (2.16)$$

Для решения задачи (1), будем использовать градиентные методы оптимизации (первого порядка). Градиент функции $\nabla E(W)$ в точке W это направление наискорейшего её возрастания, соответственно для минимизации функции необходимо изменять параметры в сторону противоположную градиенту. Этот подход называется методом градиентного спуска. Общая схема такого обучения выглядит следующим образом:

- инициализировать веса W (случайными малыми значениями)
- вычисление ошибку $E(\mathrm{h(X, W), C})$
- если результат удовлетворительный то конец работы
- вычисляем значение градиента функции потери: $\nabla E(h(X, W), C)$
- вычисляем изменение параметров: $\Delta W = \eta \cdot \nabla E$
- корректировка параметров: $W \coloneqq W - \Delta W$
- переход на п.2

Параметр η называют скоростью обучения, он определяет величину шага процесса оптимизации и его выбор это отдельная задача, в простейшем случае его задают как константу $0 < \eta < 1$.

Таким образом, следующая задача, которую необходимо решить для реализации обучения нейронной сети, это найти способ вычисления градиента функции потери $\nabla E(h(X, W), C)$.

Метод градиентного спуска в "чистом" виде может "застревать" в локальных минимумах функции потери E, для борьбы с этим можно использовать несколько дополнений для этого метода. Первое дополнение -это применение уже описанной выше стратегии mini-batch, а второе это так называемые "моменты".

Метод SGD Moment

Метод моментов можно сравнить с поведением тяжёлого шарика, который скатываясь по склону в ближайшую низину некоторое расстояние способен двигаться вверх по инерции, выбираясь таким образом из локальных минимумов. Формально это выглядит как добавка для изменения весов сети.

$$\Delta W_t := \eta \cdot \nabla E + \mu \cdot \Delta W_{t-1}, \quad (2.17)$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации.

Ещё одним усовершенствованием метода оптимизации будет применение регуляризации. Регуляризация "накладывает штраф" на чрезмерный рост значений весов это помогает бороться с переобучением. Формально это выглядит ещё одна добавка для изменения весов сети.

$$\Delta Wt := \eta \cdot (\nabla E + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}, \quad (2.18)$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации;

ρ - коэффициент регуляризации;

W_{t-1} - значения весов на предыдущей итерации.

Так же будем изменять коэффициент скорости обучения η на каждой итерации t в зависимости от изменения ошибки E следующим образом.

$$f(x) = \begin{cases} \alpha \cdot \eta_{t-1}, & \Delta E > 0 \\ \beta \cdot \eta_{t-1}, & - \end{cases} \quad (2.19)$$

где $\eta_0 = 0{,}01$- начальное значение скорости обучения;

$\Delta E = E_t - \gamma \cdot E_{t-1}$ - изменение ошибки;

$\alpha = 0.99, \beta = 1.01, \gamma = 1.01$ - константы

Таким образом, при существенном росте ошибки E шаг изменения параметров η, уменьшается, в противном случае - шаг η увеличивается. Это дополнение может увеличить скорость сходимости алгоритма.

Собрав всё вместе, алгоритм приобретает следующий вид.

инициализировать веса W (случайными малыми значениями)

инициализировать нулями начальные значение изменения весов ΔW

вычисляем ошибку $E(h(X, W), C, W)$ и её изменение ΔE на контрольном наборе

если результат удовлетворительный то то выполняем итоговый тест и конец работы

случайным образом выбрать подмножество $(X, C)_L$ из учебного набора

вычисляем значение градиента функции потери ∇E на выбранном подмножестве

вычисляем изменение параметров: $\Delta W \coloneqq \eta \cdot (\nabla E + \rho \cdot W) + \mu \cdot \Delta W$

корректируем параметры: $W \coloneqq W - \Delta W$

корректируем скорость обучения η

переход на п.3

Метод rProp.

В этом разделе рассмотрим модификацию описанного в предыдущем разделе метода градиентного спуска, которая называется rProp (resilient back propagation). В случае rProp моменты и регуляризация не используются, применяется простая стратегия full-batch. Ключевую роль играет параметр скорости обучения η, он рассчитывается для каждого веса индивидуально.

$$f(x) = \begin{cases} min(\eta_{max}, a \cdot \eta(t-1)), & S > 0 \\ max(\eta_{min}, b \cdot \eta(t-1)), & S < 0 \\ \eta(t-1) \qquad , & S = 0 \end{cases} \tag{2.20}$$

Где $S = \nabla E(t-1) \cdot \nabla E(t)$ - произведения значений градиента на этом и предыдущем шаге, $\eta_{max} = 50$, $\eta_{min} = 10^{-6}$, $a = 1.2$, $b = 0.5$ – константы.

Изменение параметров выглядит следующим образом.

$$\Delta Wt := \eta \cdot (sign(\nabla E) + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1},$$

где

$$sign(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases} \tag{2.21}$$

Метод NAG (Nesterov’s Accelerated Gradient)

В этом разделе рассмотрим модификацию метода градиентного спуска NAG, здесь градиент вычисляется относительно сдвинутых на значение момента весов.

$$\Delta W_t := \eta \cdot (\nabla E(W_{t-1} + \mu \cdot \Delta W_{t-1}) + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1} \qquad (2.22),$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации;

ρ - коэффициент регуляризации;

W_{t-1} - значения весов на предыдущей итерации.

Метод AdaGrad (Adaptive Gradient)

Метод AdaGrad учитывает историю значений градиента следующим образом.

$$g_t \coloneqq \frac{\nabla E^2}{\sqrt{\sum_{t=1}^{t} \nabla E_i^2}} \quad (2.23)$$

$$\Delta W_t := \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}, \quad (2.24)$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации;

ρ - коэффициент регуляризации;

W_{t-1} - значения весов на предыдущей итерации.

Метод AdaDelta

Метод AdaDelta учитывает историю значений градиента и историю изменения весов следующим образом.

$$S_t := \alpha \cdot S_t - 1 + (1 - \alpha) \cdot \nabla E_t^2\ ;\ S_0 := 0 \quad (2.25)$$

$$D_t := \beta \cdot D_t - 1 + (1 - \beta) \cdot \Delta W_{t-1}^2\ ;\ D_0 := 0 \quad (2.26)$$

$$g_t \coloneqq \frac{\sqrt{D_t}}{\sqrt{S_t}} \cdot \nabla E_t \quad (2.27)$$

$$\Delta W_t := \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}, \quad (2.28)$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации;

ρ - коэффициент регуляризации;

W_{t-1} - значения весов на предыдущей итерации, $\alpha = \beta = 0.9$

Метод Adam

Метод Adam преобразует градиент следующим образом.

$$S_t \coloneqq \alpha \cdot S_{t-1} + (1 - \alpha) \cdot \nabla E_t^2\ ;\ S_0 \coloneqq 0 \quad (2.29)$$

$$D_t \coloneqq \beta \cdot D_{t-1} + (1 - \beta) \cdot \nabla E_t\ ;\ D_0 \coloneqq 0 \quad (2.30)$$

$$g_t \coloneqq \frac{D_t}{1 - \beta} \cdot \sqrt{\frac{1 - \alpha}{S_t}} \quad (2.31)$$

$$\Delta W_t := \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}, \quad (2.32)$$

где η - коэффициент скорости обучения;

∇E - градиент функции потери;

μ - коэффициент момента;

ΔW_{t-1} - изменение весов на предыдущей итерации;

ρ - коэффициент регуляризации;

W_{t-1} - значения весов на предыдущей итерации, $\alpha = 0.999, \beta = 0.9$

Вычислительной математике так же известны алгоритмы второго порядка, которым под силу найти хороший минимум и на сложном ландшафте. Чтобы воспользоваться честным методом второго порядка, придётся посчитать гессиан $J(\Theta)$ - матрицу производных по каждой паре параметров пары параметров, а в случае метода Ньютона, еще и обратную к ней. Приходится применять различные подходы, чтобы справиться с проблемами, оставляя задачу вычислительно подъёмной.

2.2.3 Регуляризация

Существует несколько методов контроля емкости нейронной сети, позволяющих предотвратить переобучение.

L2-регуляризация

L2-регуляризация, вероятно, является наиболее распространенным методом регуляризации. Данный метод штрафует модель с помощью квадратов весов:

$$J = J_0 + \frac{\lambda}{2n}\sum_{w} W^2, \quad (2.33)$$

где λ - коэффициент регуляризации.

Множитель 1/2 используется для того, чтобы градиент этого слагаемого по параметру *w* равнялся λw, а не $2\lambda w$. Интуитивная интерпретация L2-регуляризации заключается в том, что она сильно штрафует векторы весов с большими значениями, и слабо затрагивает векторы с умеренными значениями.

L1-регуляризация является еще одним распространенным методом регуляризации. В рамках этого метода для каждого веса ω мы прибавляем к целевой функции слагаемое $\lambda|w|$:

$$J = J_0 + \frac{\lambda}{n}\sum_{w} |W| \quad (2.34)$$

Применяется также комбинация L1- и L2-регуляризации: $\lambda_1|\omega| + \lambda_2 w^2$. Этот метод имеет название эластичная сеть (elastic net).

L1-регуляризация имеет интересное свойство, заключающееся в том, что в ее результате векторы весов становятся разреженными (т.е. очень близкими к нулю). Другими словами, нейроны с L1-регуляризацией в итоге используют только

небольшое подмножество наиболее важных входов и, соответственно, почти не подвержены влиянию «шумных» входов.

На практике, если нет необходимости в непосредственном отборе признаков, L2-регуляризация обеспечит лучший результат по сравнению с L1-регуляризацией.

Ограничение нормы вектора весов

Еще одним методом регуляризации является метод ограничения нормы вектора весов (max norm constraint). В рамках данного метода задается абсолютный верхний предел для нормы вектора весов каждого нейрона. Соблюдение ограничения обеспечивается с помощью проецируемого градиентного спуска (projected gradient descent). На практике это реализуется следующим образом: обновление весов выполняется как обычно, а затем вектор весов w каждого нейрона ограничивается так, чтобы выполнялось условие $||w||2 < \beta$. Обычно значение β составляет порядка 3 или 4. Некоторые исследователи сообщают о положительном эффекте при использовании данного метода регуляризации. Одно из полезных свойств этого метода заключается в том, что он позволяет предотвратить «взрывной» рост весов даже при слишком большой скорости обучения, потому что обновления весов всегда ограничены.

Дропаут (dropout) - простой и очень эффективный метод регуляризации, дополняющий вышеназванные методы. Он был предложен в работе [5]. Суть метода состоит в том, что в процессе обучения из общей сети случайным образом многократно выделяется подсеть, и обновление весов выполняется только в рамках этой подсети. Нейроны попадают в подсеть с вероятностью p, которая называется коэффициентом дропаута. Во время тестирования дропаут не применяется, вместо этого веса умножаются на коэффициент дропаута, в результате чего можно получить усредненную оценку для ансамбля всех подсетей. На практике коэффициент дропаута p обычно выбирают равным 0,5, но его можно подобрать с помощью валидационного набора данных.

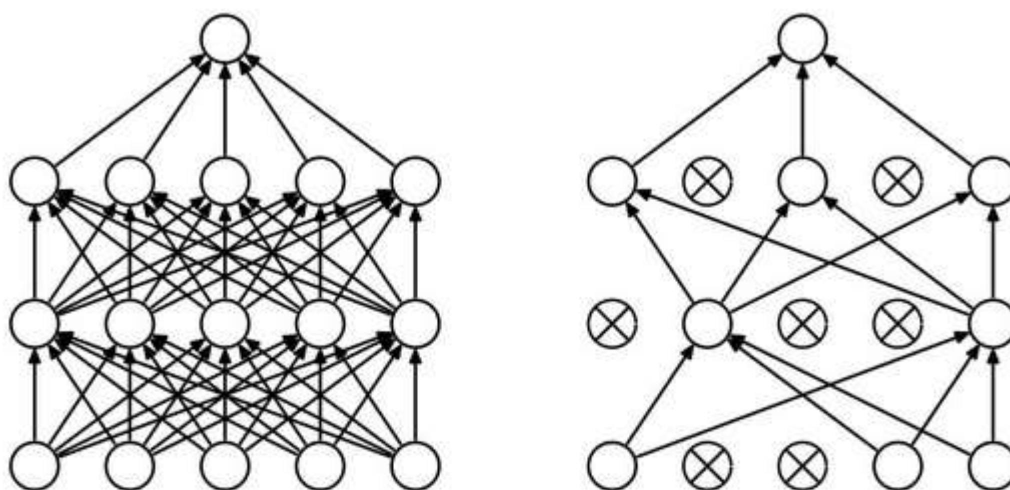


Рисунок 2.5 – Регулиризация - дропаут

.

2.2.4 Батч-нормализация

Батч-нормализация (batch normalization) - метод ускорения глубокого обучения, предложенный Ioffe и Szegedy в начале 2015 года. Метод решает следующую проблему, препятствующую эффективному обучению нейронных сетей: по мере распространения сигнала по сети, даже если его нормализовали на входе, пройдя через внутренние слои, он может сильно исказиться как по матожиднию, так и по дисперсии (данное явление называется внутренним ковариационным сдвигом), что чревато серьезными несоответствиями между градиентами на различных уровнях. Поэтому приходится использовать более сильные регуляризаторы, замедляя тем самым темп обучения.

Батч-нормализация предлагает весьма простое решение данной проблемы: нормализовать входные данные таким образом, чтобы получить нулевое матожидание и единичную дисперсию Нормализация выполняется перед входом в каждый слой. Это значит, что во время обучения нормализуется batch_size примеров, а во время тестирования нормализует статистику, полученную на основе всего обучающего множества, так как увидеть заранее тестовые данные не можем. А именно, вычисляем матожидание и дисперсию для определенного батча (пакета) $\beta = x_1, \dots, x_m$ следующим образом:

$$\mu\beta = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.35}$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu\beta)^2 \tag{2.36}$$

С помощью этих статистических характеристик преобразуем функцию активации таким образом, чтобы она имела нулевое матожидание и единичную дисперсию на всем батче:

$$\widehat{x_\iota} = \frac{x_i - \mu\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{2.37}$$

где $\varepsilon > 0$ - параметр, защищающий от деления на 0 (в случае, если среднеквадратичное отклонение батча очень мало или даже равно нулю). Наконец, чтобы получить окончательную функцию активации y, надо убедиться, что во время нормализации не потеряли способности к обобщению, и так как к исходным данным применили операции масштабирования и сдвига, можем позволить

произвольные масштабирование и сдвиг нормализованных значений, получив окончательную функцию активации:

$$y_i = \gamma \hat{x_i} + \beta \quad (2.38)$$

Где β и γ - параметры батч-нормализации, которым системы можно обучить (их можно оптимизировать методом градиентного спуска на обучающих данных). Это обобщение также означает, что батч-нормализацию может быть полезно применять непосредственно к входным данным нейронной сети.

Этот метод в применении к глубоким сверточным сетям почти всегда успешно достигает своей цели - ускорить обучение. Более того, он может случить отличным регуляризатором, позволяя не так осмотрительно выбирать темп обучения, мощность L2-регуляризатора и dropout (иногда необходимость в них совсем отпадает). Регуляризация здесь - следствие того факта, что результат работы сети для определенного примера больше не детерминировано (он зависит от всего батча, в рамках которого данный результат получен), что упрощает обобщение.

Хотя авторы метода рекомендуют применять батч-нормализацию до функции активации нейрона, последние исследования показывают, что если не полезнее, то по крайней мере так же выгодно использовать ее после активации.

2.2.5 Ранняя остановка

Оптимальное число скрытых элементов - специфическая проблема, решаемая опытным путем. Но общее правило: чем больше скрытых нейронов - тем выше риск переобучения. В этом случае система не изучает возможности данных, а как бы запоминает сами паттерны и любой содержащийся в них шум. Такая сеть отлично работает на выборке и плохо за пределами выборки. Есть два основных метода для решения данной проблемы: ранняя остановка (early stopping) и регуляризация (рассмотренная ранее).

Ранняя остановка предполагает разделение процесса обучения на этапы самого обучения и валидации результатов. Вместо того чтобы обучать сеть на ограниченном числе итераций, необходимо обучаете ее пока производительность сети на этапе подтверждения не начинает падать. По-существу, это не дает сети использовать все доступные параметры и ограничивает способности к простому запоминанию паттернов. Ниже показаны две возможные точки остановки:

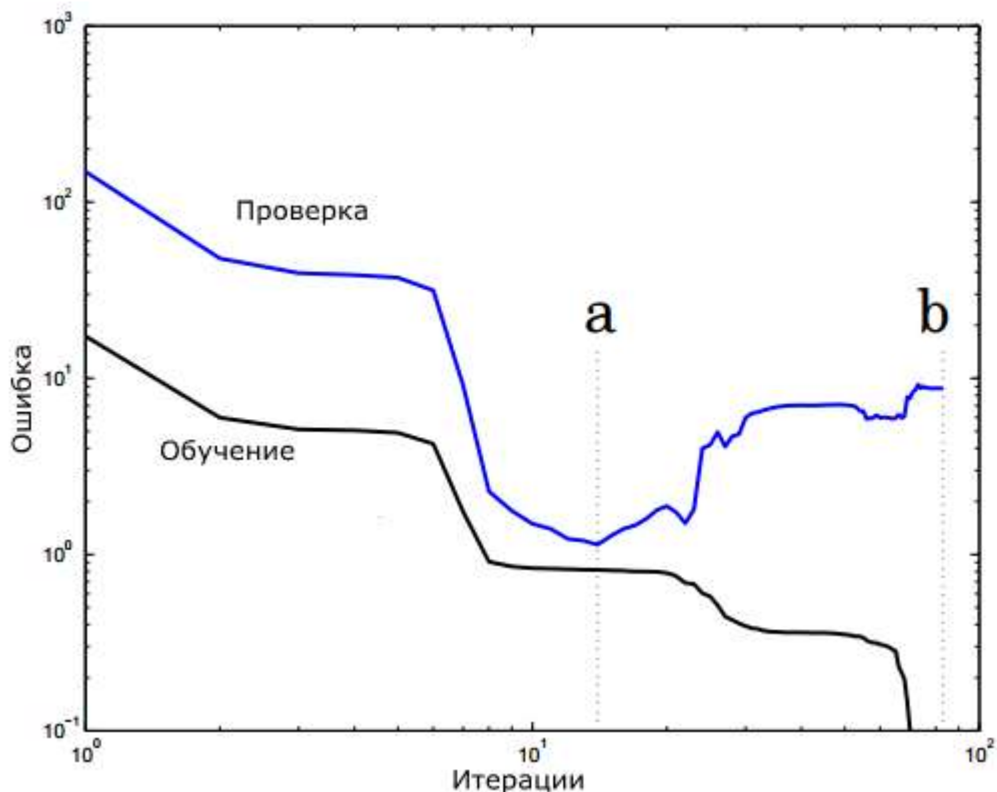


Рисунок 2.6 – Схема сверточного слоя

Еще одна картинка показывает производительность и степень переобучение сети при остановке в этих точках a и b:

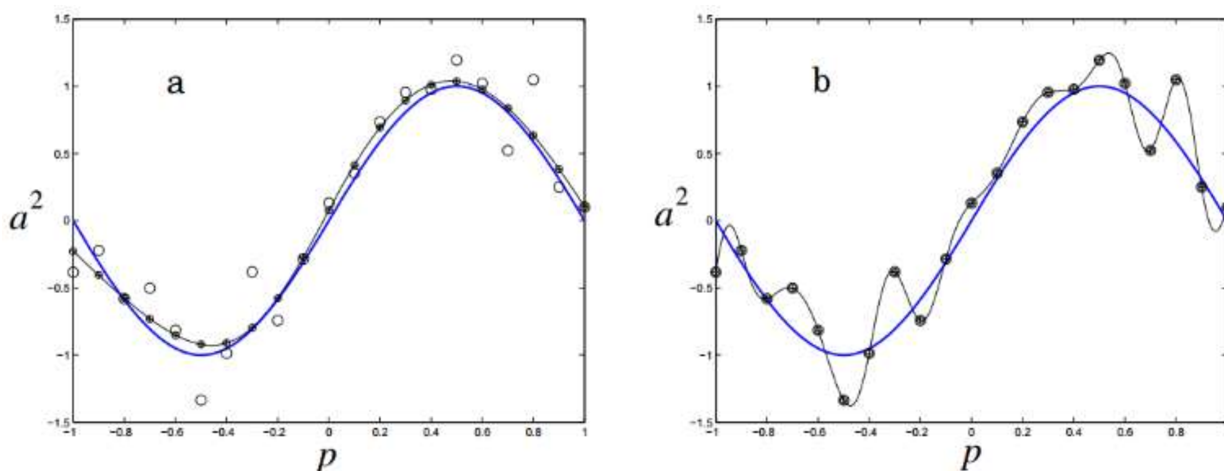


Рисунок 2.7 – Схема сверточного слоя

2.2.6 Перенос обучения

На практике обучение целых СНС обычно не производится с нуля, с произвольной инициализацией. Причина состоит в том, что обычно не удается найти набор данных достаточного размера, требуемого для сети нужной глубины. Вместо этого чаще всего происходит предварительное обучение СНС на очень крупном наборе данных, а затем использование весов обученной СНС либо в

качестве инициализации, либо в качестве выделения отличительных признаков для определенной задачи.

Стратегии переноса обучения зависят от разных факторов, но наиболее важными являются два: размер нового набора данных и его схожесть с исходным набором данных. Если учесть, что характер работы СНС более универсален на ранних слоях и становится более тесно связанным с конкретным набором данных на последующих слоях, можно выделить четыре основных сценария:

-Новый набор данных меньше по размеру и аналогичен по содержанию исходному набору данных. Если объем данных невелик, то нет смысла проводить тонкую настройку СНС из-за чрезмерной подгонки. Поскольку данные схожи с изначальными, можно предполагать, что отличительные черты в СНС будут релевантны и для этого набора данных. Поэтому оптимальным решением является обучение линейного классификатора отличительным признаком СНС.

-Новый набор данных относительно крупный и аналогичен по содержанию исходному набору данных. Поскольку у нас больше данных, можно не беспокоиться о чрезмерной подгонке, если мы попытаемся провести тонкую настройку всей сети.

-Новый набор данных меньше по размеру и существенно отличается по содержанию от исходного набора данных. Поскольку объем данных невелик, будет вполне достаточно только линейного классификатора. Так как данные существенно отличаются, лучше обучать классификатор не с вершины сети, где содержатся более конкретные данные. Вместо этого лучше обучить классификатор, активировав его на более ранних слоях сети.

-Новый набор данных относительно крупный и существенно отличается по содержанию от исходного набора данных. Поскольку набор данных очень крупный, можно позволить себе обучить всю СНС с нуля. Тем не менее на практике зачастую все равно оказывается выгоднее использовать для инициализации весов из заранее обученной модели. В этом случае мы будем располагать достаточным объемом данных для тонкой настройки всей сети.

Решая вопрос определения подходящей площадки для приземления, предлагается действовать по сценарию IV. Провести тонкую настройку весов заранее обученной СНС, продолжая обратное распространение. Можно либо провести тонкую настройку всех слоев СНС, либо оставить некоторые из ранних слоев неизменными (во избежание чрезмерной подгонки) и настроить только высокоуровневую часть сети. Это обусловлено тем, что на ранних слоях СНС содержатся более универсальные функции (например, определение краев или цветов), полезные для множества задач, а более поздние слои СНС уже

ориентированы на классы набора данных возможных окрестностей для приземления.

Технология переноса обучения (transfer learning) позволяет использовать готовые нейронные сети для решения задач нового типа, не тех, для которых сети предварительно обучались.

Многие нейронные сети для задач классификации объектов на изображениях обучены на наборе данных ImageNet. Этот набор данных включает 14 миллионов изображений, относящихся к 21 тысяче 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 нашей новой задачи. Измененная сеть затем обучается на новом наборе данных.

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

Сверточная часть используется для выделения характерных признаков из изображения.

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, то мы можем взять их и применить для решения другой задачи классификации. Таким образом, мы переносим обучение сверточной части сети на новую задачу.

Для реализации переноса обучения нам нужно заменить классификатор в предварительно обученной нейронной сети. Дальнейшее изложение будет применительно к сети VGG16.

Рисунок 2.8 – Схема сверточного слоя

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 (max pooling). На трех следующих каскадах по три слоя свертки и один слой подвыборки. Размер ядер во всех слоях свертки 3x3.

Полносвязная часть сети VGG16 включает три уровня. На выходном уровне 1000 нейронов по количеству классов объектов. Используется формат one-hot encoding: 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:

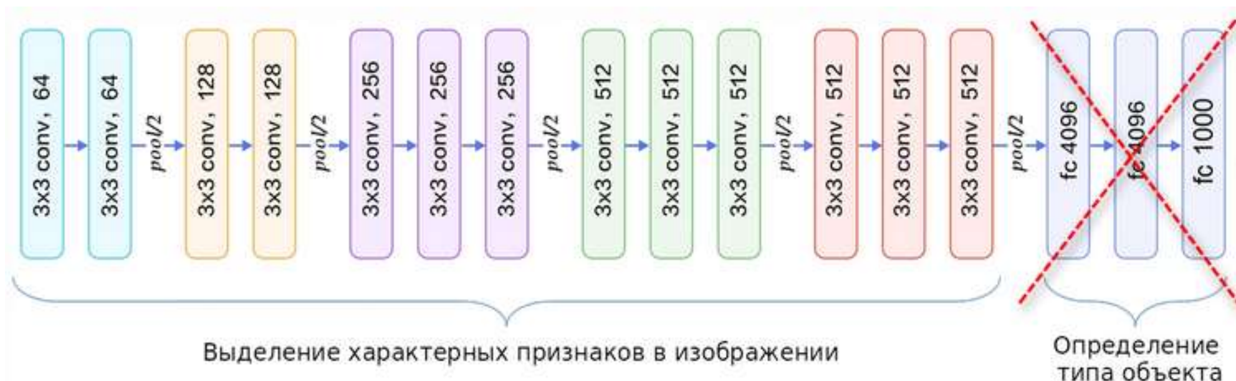


Рисунок 2.9 – Схема сверточного слоя

Второй этап: к сверточной части сети VGG16 добавляем новый классификатор для распознавания холмов и равнин:

Рисунок 2.10 – Схема сверточного слоя

Новый классификатор устроен гораздо проще полносвязной части сети VGG16, т.к. нам 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 равнина, а единица - холм.

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 этапе измененную сеть нужно обучить на новом наборе данных с фотографиями холмов и равнин.

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

Часть сети уже предварительно обучена, поэтому для новой части нужно использовать небольшую скорость обучения, иначе алгоритм обучения может не сойтись.

2.2.7 Тонкая настройка

Тонкая настройка сети (fine tuning) позволяет пойти дальше и еще больше увеличить качество работы предварительно обученной сети на новой задаче. Для этого обучается не только новый классификатор, который был добавлен в сеть, но и некоторые слои предварительно обученной нейронной сети. Это особенно эффективно, когда новый набор данных достаточно сильно отличается от исходного набора, на котором обучалась сеть.

Алгоритм тонкой настройки предварительно обученной нейронной сети

Для тонкой настройки сети необходимо выполнить следующие действия:

Заменить классификатор предварительно обученной нейронной сети новым классификатором, подходящим под нашу задачу.

“Заморозить” сверточные слои предварительно обученной нейронной сети. В результате эти слои не будут обучаться.

Провести обучение составной сети с новым классификатором на новом наборе данных.

“Разморозить” несколько слоев сверточной части предварительно обученной нейронной сети.

Дообучить сеть с размороженными сверточными слоями на новом наборе данных. Именно этот этап и называется тонкой настройкой (fine tuning).

Рисунок 2.11 – Схема сверточного слоя

Тонкую настройку можно проводить только после того, как обучен новый классификатор. В классификаторе веса назначаются случайным образом, поэтому на первых этапах обучения сигнал об ошибке будет очень большой. Если этот сигнал распространится в сверточную часть сети, то результат предварительного обучения может быть утерян, т.к. веса нейронов будут сильно меняться. Когда же обучение классификатора на новом наборе данных завершено, то таких сильных изменений весов уже не будет, и можно переходить к обучению сверточной части.

Первый этап тонкой настройки – разморозить несколько сверточных слоев в сети. VGG16 состоит из нескольких блоков, в каждом из которых несколько сверточных слоев и слой подвыборки. Разморозим последний блок с номером 5. В этом блоке четыре слоя, их:

block5_conv1 - первый сверточный слой;

block5_conv2 - второй сверточный слой;

block5_conv3 - третий сверточный слой;

block5_pool - слой подвыборки.

Тонкая настройка (fine tuning) нейронной сети используется совместно с переносом обучения (transfer learning), когда необходимо применить предварительно обученную нейронную сеть для решения другой задачи. При тонкой настройке обучается не только новый классификатор, но и несколько слоев предварительно обученной части сети.

Тонкая настройка наиболее эффективна, когда новый набор данных значительно отличается от того, на котором выполнялось предварительное обучение сети. Если же наборы данных похожи, то эффект от тонкой настройки будет небольшим.

3 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1 Подготовка данных

В связи со схожестью поверхности космических тел и рельефа пустынь было принято решение проводить предварительное обучения сети с использованием спутниковых снимков поверхности пустынь, полученных с помощью ПО Google Earth.

Google Планета Земля (англ. Google Earth) — проект компании Google, в рамках которого в сети Интернет были размещены спутниковые (или в некоторых точках аэрофото-) изображения всей земной поверхности. Фотографии некоторых регионов имеют беспрецедентно высокое разрешение.

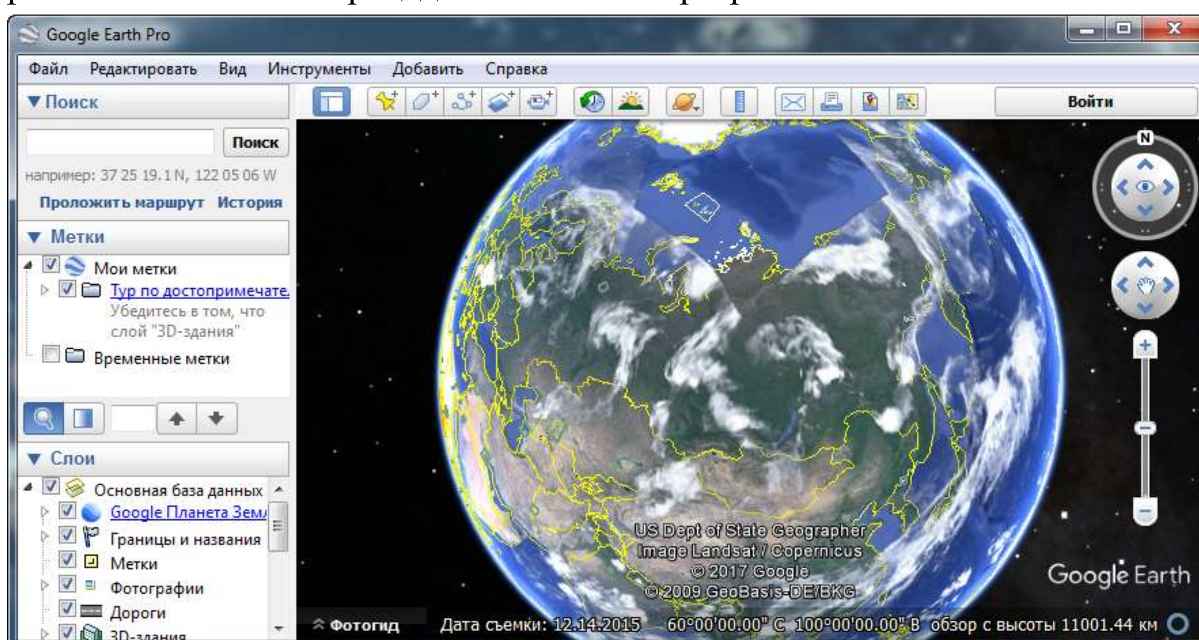


Рисунок 3.1 – Интерфейс ПО Google Earth.

В отличие от других аналогичных сервисов, показывающих спутниковые снимки в обычном браузере (например, Google Maps), в данном сервисе используется специальная, загружаемая на компьютер пользователя клиентская программа Google Earth. Такой подход хотя и требует закачивания и установки программы, но зато в дальнейшем обеспечивает дополнительные возможности, трудно реализуемые с помощью веб-интерфейса.

Эта программа изначально была выпущена компанией Keyhole, а затем куплена компанией Google, которая в 2005 году сделала программу общедоступной (сначала только для территории США, затем Европы и всего мира). Существуют также платная версия Google Earth Plus и бесплатная версия Google Earth Pro, отличающиеся поддержкой GPS навигации, средств презентаций и повышенным разрешением распечатки.

Возможности программы:

- Google Earth автоматически загружает из интернета необходимые пользователю изображения и другие данные, сохраняет их в памяти компьютера и на жёстком диске для дальнейшего использования. Скачанные данные сохраняются на диске, и при последующих запусках программы закачиваются только новые данные, что позволяет существенно экономить трафик.
- Для визуализации изображения используется трёхмерная модель всего земного шара (с учётом высоты над уровнем моря), которая отображается на экране при помощи интерфейсов DirectX или OpenGL. Именно в трёхмерности ландшафтов поверхности Земли и состоит главное отличие программы Google Earth от её предшественника Google Maps. Пользователь может легко перемещаться в любую точку планеты, управляя положением «виртуальной камеры».
- Практически вся поверхность суши покрыта изображениями, полученными от компании DigitalGlobe и имеющими разрешение 15 м на пиксель. Данные высот рельефа имеют разрешение порядка 30 метров по горизонтали на территории США, порядка 90 метров на остальной территории и точность по вертикали вплоть до одного метра.

Рисунок 3.2 – Изображение полученное с помощью ПО Google Earth.

В связи с необходимостью достаточно большого объема данных для тренировки нейронной сети, над полученными данными в дальнейшем будут проводится различного рода преобразования (внесение искажений). В ходе этих преобразований будет получено достаточное количество изображений необходимое для обучения сети.

3.2 Обоснование выбора среды программной реализации проекта

Наиболее распространенным языком программирования для создания и исследования нейронных сетей является Python.

Python — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций.

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных.

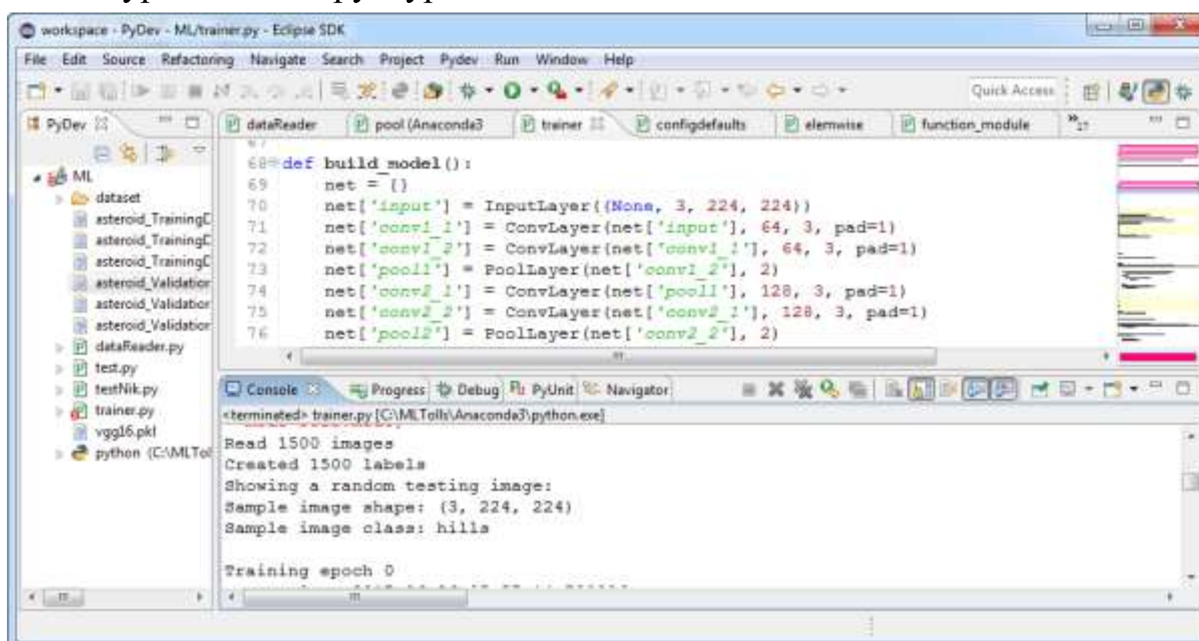


Рисунок 3.3 – Интерфейс среды разработки Eclipse.

Eclipse — свободная интегрированная среда разработки модульных кроссплатформенных приложений. Развивается и поддерживается Eclipse Foundation.

Наиболее известные приложения на основе Eclipse Platform — различные «Eclipse IDE» для разработки ПО на множестве языков (например, наиболее популярный «PyDev IDE, Java IDE», поддерживавшийся изначально, не полагается на какие-либо закрытые расширения, использует стандартный открытый API для доступа к Eclipse Platform).

Для возможности использования языка Python в среде разработки Eclipse необходимо установка модуля PyDev.

Библиотеки Theano/Lasagne

Theano — это расширение языка Python, позволяющее эффективно вычислять математические выражения, содержащие многомерные массивы. Библиотека получила свое название в честь имени жены древнегреческого философа и математика Пифагора — Феано (или Теано). Theano разработана в лаборатории LISA для поддержки быстрой разработки алгоритмов машинного обучения.

Библиотека реализована на языке Python, поддерживается на операционных системах Windows, Linux и Mac OS. В состав Theano входит компилятор, который переводит математические выражения, написанные на языке Python в эффективный код на C или CUDA.

Theano предоставляет базовый набор инструментов для конфигурации нейросетей и их обучения. Возможна реализация многослойных полностью связанных сетей (Multi-Layer Perceptron), сверточных нейросетей (CHC), рекуррентных нейронных сетей (Recurrent Neural Networks, RNN), автокодировщиков и ограниченных машин Больцмана. Также предусмотрены различные функции активации, в частности, сигмоидальная, softmax-функция, кросс-энтропия. В ходе обучения используется пакетный градиентный спуск (Batch SGD).

Установка

Для установки нам понадобятся: python версии старше 2.6 или 3.3 (лучше dev-версию), компилятор C++ (g++ для Linux или Windows, clang для MacOS), библиотека примитивов линейной алгебры (например ATLAS, OpenBLAS, Intel MKL), NumPy и SciPy.

Для выполнения вычислений на GPU понадобится CUDA, а ряд операций, встречающихся в нейронных сетях, можно ускорить с помощью cuDNN. Начиная с версии 0.8.0, разработчики Theano рекомендуют использовать libgpuarray, что также даёт возможность использовать несколько GPU.

Настройка

Theano можно настроить тремя способами:

- Выставив атрибуты объекта theano.config в нужное значение;
- Через переменную окружения THEANO_FLAGS;
- Через конфигурационный файл $HOME/.theanorc (или $HOME/.theanorc.txt под Windows).

Пример конфигурационного файла:

```
[global]
    device = gpu # выбирает устройство, на котором будет
выполняться наш код - GPU или CPU
    floatX = float32
```

```
    optimizer_including=cudnn
    allow_gc = False # быстрее, но использует больше памяти
    #exception_verbosity=high
    #optimizer = None
    #profile = True
    #profile_memory = True
     config.dnn.conv.algo_fwd = time_once # эти две опции зачастую приводят к ускорению свёрток
     config.dnn.conv.algo_bwd = time_once
[lib]
     Cnmem = 0.95 # позволяет включить CNMeM (https://github.com/NVIDIA/cnmem) - менеджер CUDA-памяти
```

Lasagne - библиотека для нейронных сетей, работающей поверх Theano. Lasagne предоставляет набор готовых компонентов: слоёв, алгоритмов оптимизации, функций потерь, инициализаций параметров и т.д., при этом не скрывает Theano за многочисленными слоями абстракций.

Его основными функциями являются:

- Поддерживает нейронные сети, такие как сверточные нейронные сети (СНС), рекурентные сети, включая Long Short-Term Memory (LSTM) и любую их комбинацию;
- Позволяет создавать архитектуры нескольких входов и нескольких выходов, включая вспомогательные классификаторы;
- Реализует многие методы оптимизации, включая импульс Нестерова, RMSprop и ADAM;
- Прозрачная поддержка CPU и GPU компилятору выражений Theano.

В связи с необходимостью использования больших вычислительных ресурсов для создания и обучения сверточной нейронной сети было принято решение использовать вычислительные ресурсы специализированных сервисов.

3.3 Высокопроизводительные средства вычислений

Сервис Amazon EC2 предоставляет широкий выбор типов инстансов (выделенная вычислительная машина, ВМ), оптимизированных для различных вариантов использования. Типы ВМ включают различные комбинации таких компонентов, как CPU, память, хранилище и сетевые возможности, и позволяют выбрать соответствующий набор ресурсов для приложений. Каждый тип инстанса включает в себя один или несколько размеров инстансов, что позволяет масштабировать ресурсы в соответствии с требованиями целевой рабочей нагрузки.

ВМ для ускоренных вычислений P2

ВМ P2 предназначены для высоконагруженных вычислительных приложений, требующих наличия графических процессоров GPU.

Возможности:

- Высокочастотные процессоры Intel Xeon E5-2686 v4 (Broadwell)
- Высокопроизводительные графические процессоры NVIDIA K80 с 2496 параллельно работающими ядрами каждый и графической памятью объемом 12 ГиБ.
- Поддержка GPUDirect™ (прямое взаимодействие между графическими процессорами).
- Предоставление улучшенной сетевой конфигурации с помощью Amazon EC2 Elastic Network Adaptor с суммарной пропускной способностью сети 20 Гбит/с в рамках группы размещения.

Модель	**GPU**	**vCPU**	**Память (ГиБ)**	**Графическая память (ГиБ)**
p2.xlarge	1	4	61	12
p2.8xlarge	8	32	488	96
p2.16xlarge	16	64	732	192

Таблица 3.1 – Параметры ВМ P2

Примеры использования

Нагрузки, связанные с машинным обучением, высокопроизводительными базами данных, расчетной гидродинамикой, финансовой инженерией, сейсмическим анализом, молекулярным моделированием, геномикой, рендерингом, а также прочие вычислительные нагрузки на стороне сервера, требующие наличия графического процессора.

Модель	**Программное обеспечение, час**	**ВМ, час**	**Итого, час**
p2.xlarge	$0.09	$0.90	$0.99
p2.8xlarge	$0.297	$7.20	$7.497
p2.16xlarge	$0.297	$14.40	$14.697

Таблица 3.2 – Стоимость аренды ВМ P2

Характерной особенностью ВМ P2 является использования графических адаптеров Tesla K80, являющихся на сегодняшней день наиболее передовыми графическими картами используемыми в вычислительных системах.

Рисунок 3.4 – Внешний вид графического адаптера Tesla K80.

Характеристики графического адаптера Tesla K80:

- 4992 ядра CUDA на карте с двумя GPU для максимальной производительности приложений;
- Производительность в операциях с двойной точностью до 2,91 терафлопс благодаря технологии NVIDIA GPU Boost;
- Производительность в операциях с одинарной точностью до 8,73 терафлопс благодаря технологии NVIDIA GPU Boost;
- 24 ГБ памяти GDDR5;
- Средняя пропускная способность памяти 480 ГБ/с;
- ECC защита для 100% надежности данных;
- Оптимизация для серверов для обеспечения лучшей производительности в дата-центре;

Средняя стоимость графического адаптера Tesla K80 составляет порядка 290 тыс. руб.

На арендованной ВМ P2 необходимо развернуть образ операционной системы. Для данной дипломной работы использовался специализированый образ Bitfusion Ubuntu.

Описание продукта AMI Bitfusion Ubuntu 14 Theano

Ubuntu 14 AMI с предварительно установленными драйверами Nvidia, дающая возможность использовать графический процессор GPU, а также

экземпляры CPU. Предназначен для разработчиков, а также для тех, кто хочет начать работу с математическими вычислениями, машинным обучением и глубоким обучением нейронных сетей.

Текущая версия: 2016.03

Операционная система: Linux/Unix, Ubuntu 14.04

Предоставление: 64-bit Amazon Machine Image (AMI)

Требуемый сервис AWS: EC2, EBS

Особенности

- Проверенная установка совместимой ОС, ядра, драйверов, инструментария cuda, cuDNN 5, Theano, Keras, Lasagne, Python 2 и Python 3, PyCuda, Scikit-Learn, Pandas, Enum34, iPython 5 и Jupyter.
- Образ оптимизирован для работы на процессоре с экземплярами GPU.
- Быстрое начало работы не требующие установки и настройку драйверов и инструментов.

Настройка и последующий запуск ВМ Р2 производится с помощью консоли управления EC2 представленной на рис. 5

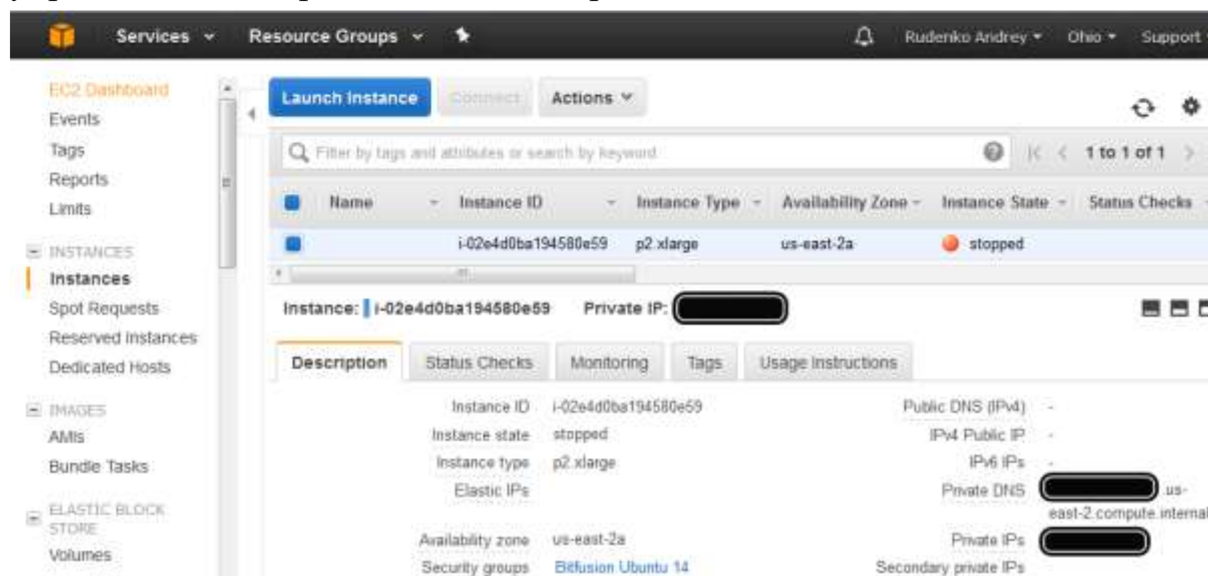


Рисунок 3.5 – Окно консоли управления EC2.

Для передачи данных (набор изображений) для обучения нейронной сети и скрипта (программа) Python на арендованную ВМ Р2 используется защищённый протокол SSH посредством использования специализированного ПО PuTTY, либо терминала UNIX-подобной операционной системы.

PuTTY — клиентская программа для работы с сетевыми протоколами SSH, Telnet, SCP, SFTP, для подключения по COM-порту и ZModem, утилита для генерации RSA и DSA цифровых SSH-ключей.

Наиболее популярные способы использования PuTTY — это удалённое администрирование Linux, подключение к виртуальным серверам VDS/VPS по

протоколу SSH, настройка сетевых маршрутизаторов через последовательный порт, соединение с удалёнными Telnet-терминалами.

PuTTY работает как под Windows, так и под Linux. А в списке сторонних модификаций вы найдёте версии SSH-клиента для Mac OS X, iPhone, Android, Windows Mobile, Symbian.

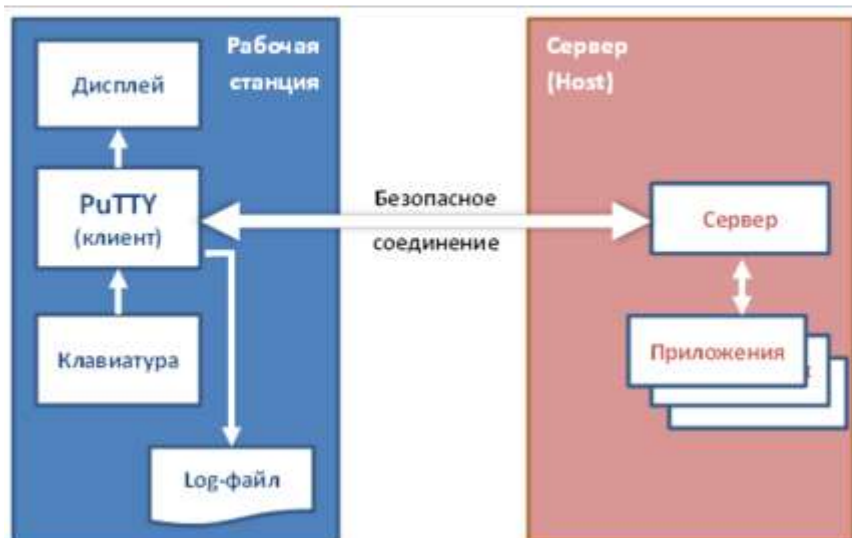


Рисунок 3.6 – Структура взаимодействия PuTTY с удаленной рабочей станцией.

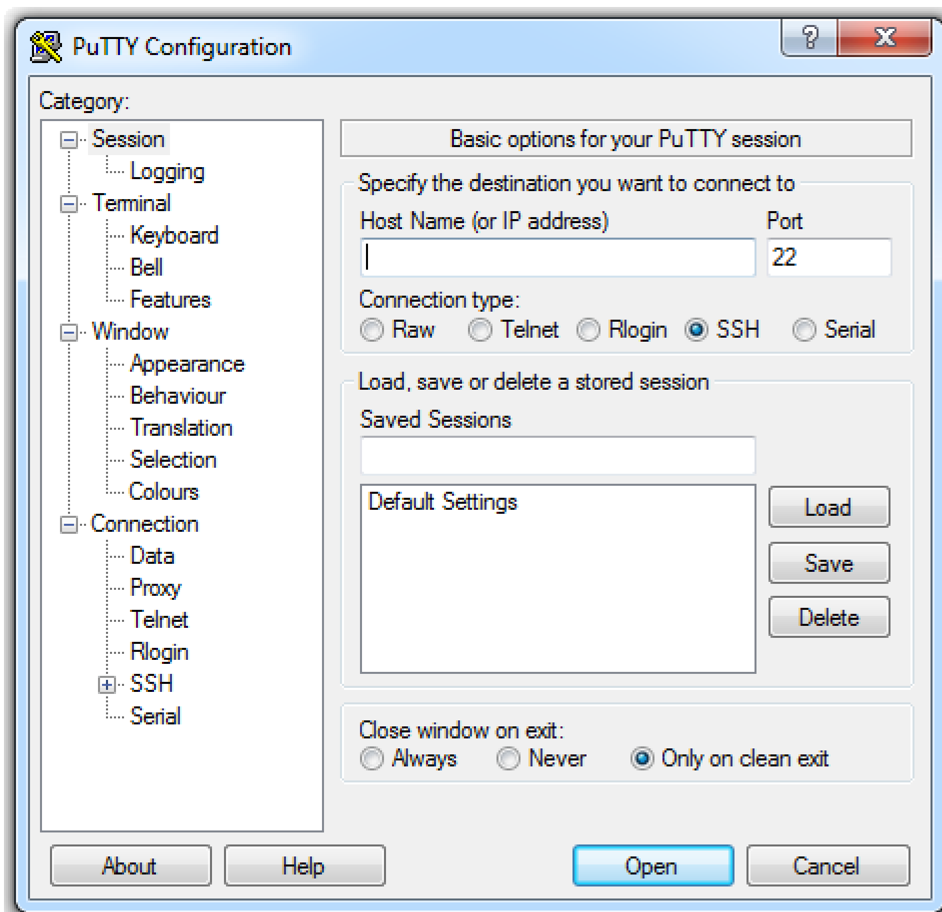


Рисунок 3.7 – Окно настройки параметров подключения PuTTY.

```
andy@ubuntu:~$ ssh ubuntu@52.14.110.61
ssh: connect to host 52.14.110.61 port 22: Connection refused
andy@ubuntu:~$ ssh ubuntu@52.14.110.61
ssh: connect to host 52.14.110.61 port 22: Connection refused
andy@ubuntu:~$ ssh ubuntu@13.59.174.150
The authenticity of host '13.59.174.150 (13.59.174.150)' can't be established.
ECDSA key fingerprint is SHA256:[redacted].
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added '13.59.174.150' (ECDSA) to the list of known hosts.

1 package can be updated.
0 updates are security updates.


################################################################################
########################################
################################################################################
########################################
```

Рисунок 3.8 – Подключение по SSH через терминал ОС Ubuntu

3.4 Внесение изменение в выбранную архитектуру сети

В данной дипломной работе используется архитектура сети аналогичная сети VGG16.

VGG16 - это 16-слойная сеть, используемая Группой визуальных геометрий в Оксфордском университете для участия в конкурсе ILSVRC-2014, где она заняла одно из призовых мест. Главной особенностью этой архитектуры была увеличенная глубина сети. В VGG16 изображения 224x224 RGB передаются через 5 блоков сверточных слоев, где каждый блок состоит из определенного числа карт признаков. Блоки сверточных слоев разделены слоями maxPool. После 5 блоков сверточных слоев следуют три полно-связанных (FC) слоя. Последний слой - это слой с Softmax, который выводит вероятности классов. Полная модель показана на рисунке 1.

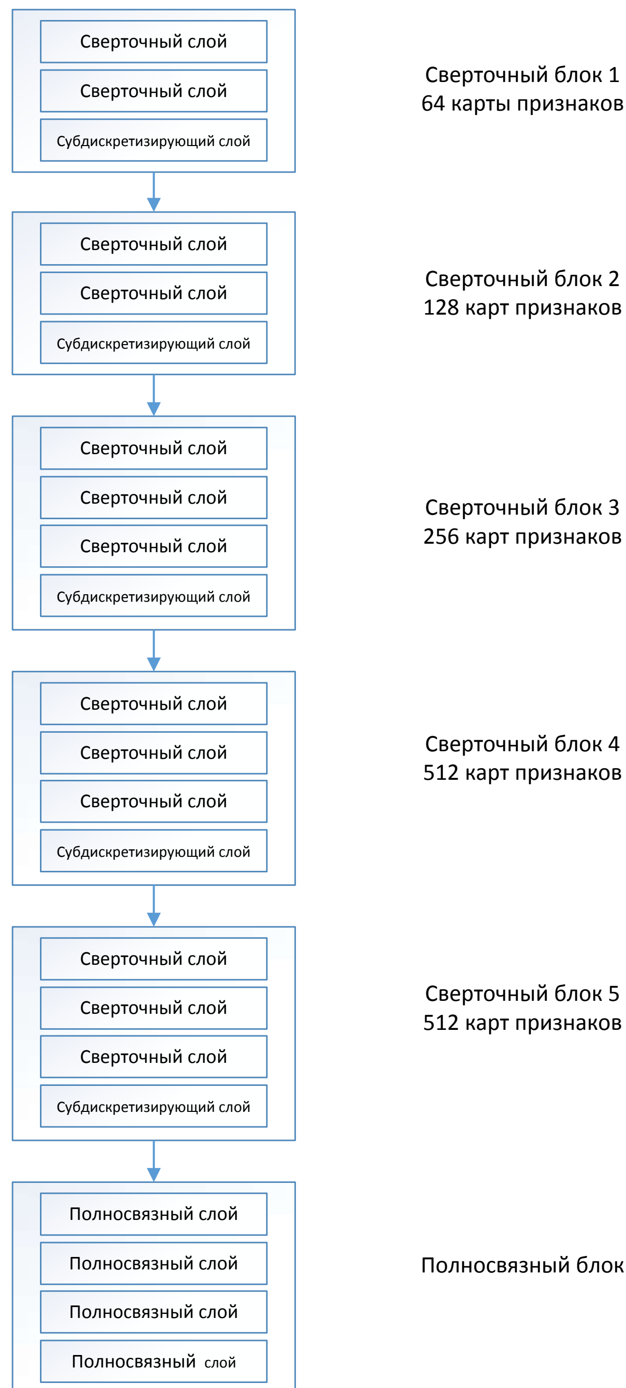


Рисунок 3.9 – Архитектура сети VGG16.

Для реализация поставленной задачи выходной слой стандартной сверточной сети, состоящий из 1000 нейронов, был замен на слой содержащий 2 нейрона, т.к. для нашей задачи требуется определить пригоден или нет выбранный периметр для посадки, что предполагает только два возможных варианта. Так же для ускорения обучения был применен принцип TransferLearning (передача обучения), для реализации которого была произведена соответствующая настройка сети: было произведено переобучение на новом наборе данных, слоев только двух последних блоков. Что позволило существенно сократить время обучения сети.

Рисунок 3.10 – Скорректированная архитектура сети VGG16.

3.5 Обучение и тестирование сверточной нейронной сети

После передачи данных (набор изображений) для обучения нейронной сети, файлов содержащих веса предварительно обученных моделей сети VGG16, Densnet, MobileNet и скрипта (программа) Python реализующего процедуру обучения и проверки нейронной сети на арендованную ВМ P2 используя защищённый протокол SSH, производим его запуск:

```
andy@ubuntu: ~
Loading VGGN and its metadata
Read 1500 images
Created 1500 labels
Showing a random testing image:
Sample image shape: (3, 224, 224)
Sample image class: hills

Training epoch 0
Start time: 2017-06-23 04:43:43.801287
Training 0 epoch over dataset 0
0
loss: 0.697457492351532
1
2
3
4
5
6
7
```

Рисунок 3.11 – Запуск процедуры обучения нейронной сети

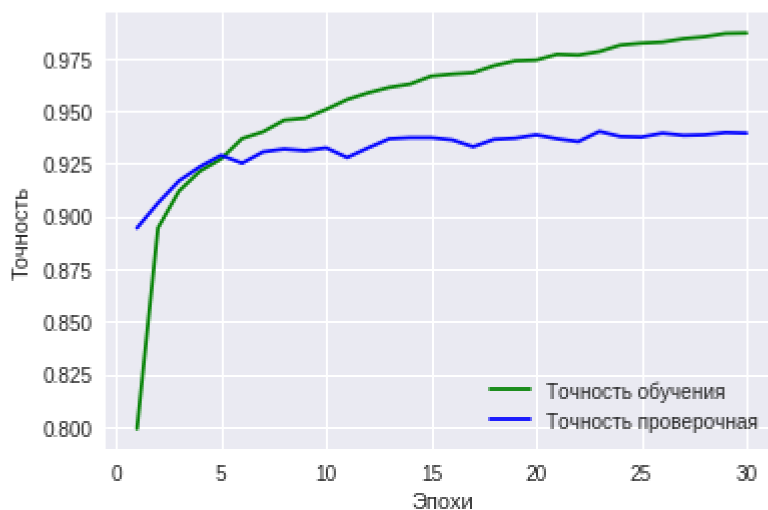
Ниже приведены данные обучения и работы сети. До бучение проводилось в течении 10 эпох, для сетей DenseNet и ResNet, для сети VGG16 потребовалось 30 эпох. Скорость работы проверялась на распознавании 2000 тестовых образцов изображений:

Сеть	Ошибка обучения	Точность обучения	Ошибка проверки	Точность тест, %	Время до обучения, мин	Время обучения эпохи, сек	Скорость работы, сек
VGG16	0,015	0,9868	0,1601	92,86	91	182	24,86
DenseNet	0,1305	0,9508	0,1669	93,16	32	189	24,51
ResNet	0,0898	0,9665	0,93	50,19	29	168	38,14

Таблица 3.1 – Сводная таблица результатов обучения нейронной сети

Как видно из результатов проверки работы, сеть ResNet переобучилась.

```
Epoch 17/30
 - 181s - loss: 0.0928 - acc: 0.9681 - val_loss: 0.1604 - val_acc: 0.9329
Epoch 18/30
 - 183s - loss: 0.0884 - acc: 0.9714 - val_loss: 0.1560 - val_acc: 0.9364
Epoch 19/30
 - 181s - loss: 0.0850 - acc: 0.9737 - val_loss: 0.1559 - val_acc: 0.9370
Epoch 20/30
 - 183s - loss: 0.0813 - acc: 0.9740 - val_loss: 0.1555 - val_acc: 0.9386
Epoch 21/30
 - 181s - loss: 0.0772 - acc: 0.9767 - val_loss: 0.1554 - val_acc: 0.9367
Epoch 22/30
 - 182s - loss: 0.0739 - acc: 0.9764 - val_loss: 0.1565 - val_acc: 0.9353
Epoch 23/30
 - 182s - loss: 0.0716 - acc: 0.9781 - val_loss: 0.1583 - val_acc: 0.9402
Epoch 24/30
 - 181s - loss: 0.0676 - acc: 0.9811 - val_loss: 0.1559 - val_acc: 0.9378
Epoch 25/30
 - 183s - loss: 0.0647 - acc: 0.9821 - val_loss: 0.1554 - val_acc: 0.9375
Epoch 26/30
 - 181s - loss: 0.0628 - acc: 0.9826 - val_loss: 0.1547 - val_acc: 0.9394
Epoch 27/30
 - 182s - loss: 0.0589 - acc: 0.9842 - val_loss: 0.1555 - val_acc: 0.9383
Epoch 28/30
 - 182s - loss: 0.0566 - acc: 0.9851 - val_loss: 0.1591 - val_acc: 0.9386
Epoch 29/30
 - 182s - loss: 0.0543 - acc: 0.9867 - val_loss: 0.1562 - val_acc: 0.9397
Epoch 30/30
 - 182s - loss: 0.0515 - acc: 0.9868 - val_loss: 0.1601 - val_acc: 0.9394
24.861979000001156 seconds
Точность на тестовых данных: 92.86%
```




```
CPU times: user 1h 7min 7s, sys: 7min 24s, total: 1h 14min 31s
Wall time: 1h 31min 58s
```

Рисунок 3.12 – Вывод отладочной информации обучения сети VGG16

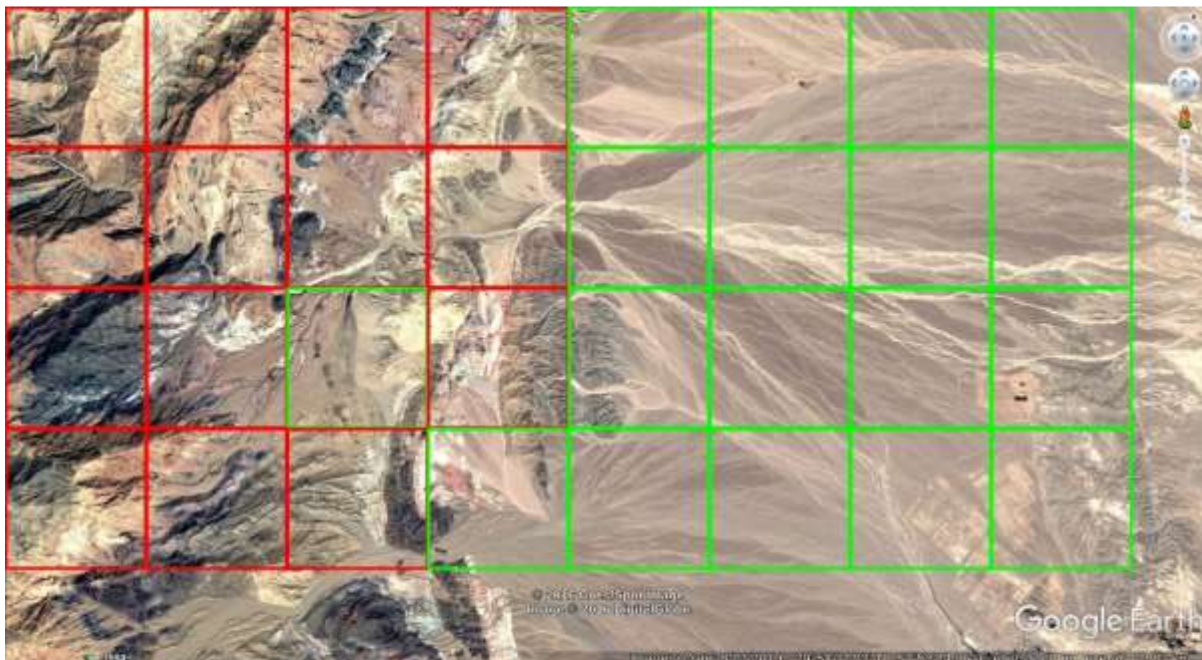
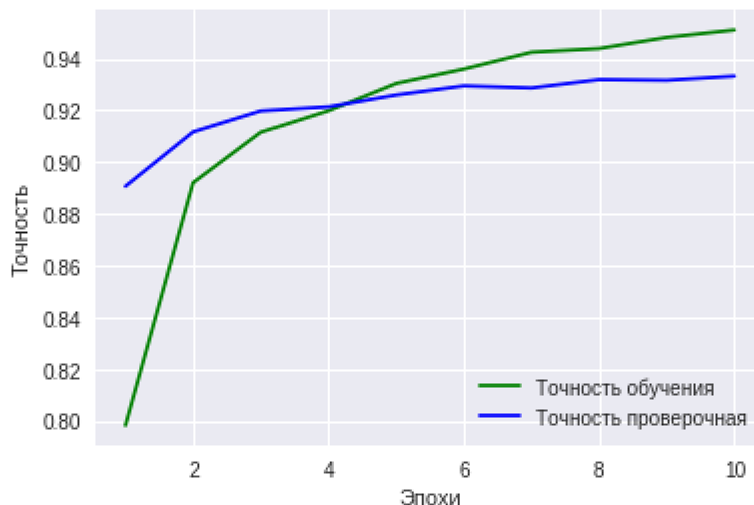


Рисунок 3.13 – Результат работы обученной нейронной сети VGG16

```
================================================================
Total params: 21,137,729
Trainable params: 6,423,041
Non-trainable params: 14,714,688
________________________________________________________________
Epoch 1/10
 - 200s - loss: 0.4252 - acc: 0.7982 - val_loss: 0.2802 - val_acc: 0.8906
Epoch 2/10
 - 188s - loss: 0.2661 - acc: 0.8922 - val_loss: 0.2311 - val_acc: 0.9116
Epoch 3/10
 - 188s - loss: 0.2197 - acc: 0.9117 - val_loss: 0.2097 - val_acc: 0.9197
Epoch 4/10
 - 188s - loss: 0.2016 - acc: 0.9197 - val_loss: 0.2012 - val_acc: 0.9213
Epoch 5/10
 - 188s - loss: 0.1789 - acc: 0.9304 - val_loss: 0.1859 - val_acc: 0.9259
Epoch 6/10
 - 188s - loss: 0.1663 - acc: 0.9358 - val_loss: 0.1806 - val_acc: 0.9294
Epoch 7/10
 - 188s - loss: 0.1553 - acc: 0.9425 - val_loss: 0.1748 - val_acc: 0.9286
Epoch 8/10
 - 188s - loss: 0.1464 - acc: 0.9439 - val_loss: 0.1730 - val_acc: 0.9318
Epoch 9/10
 - 188s - loss: 0.1374 - acc: 0.9483 - val_loss: 0.1697 - val_acc: 0.9316
Epoch 10/10
 - 188s - loss: 0.1305 - acc: 0.9508 - val_loss: 0.1669 - val_acc: 0.9332
24.513177999999243 seconds
Аккуратность на тестовых данных: 93.16%
```




```
CPU times: user 22min 31s, sys: 2min 40s, total: 25min 12s
Wall time: 32min 12s
```

Рисунок 3.14 – Вывод отладочной информации обучения сети DenseNet

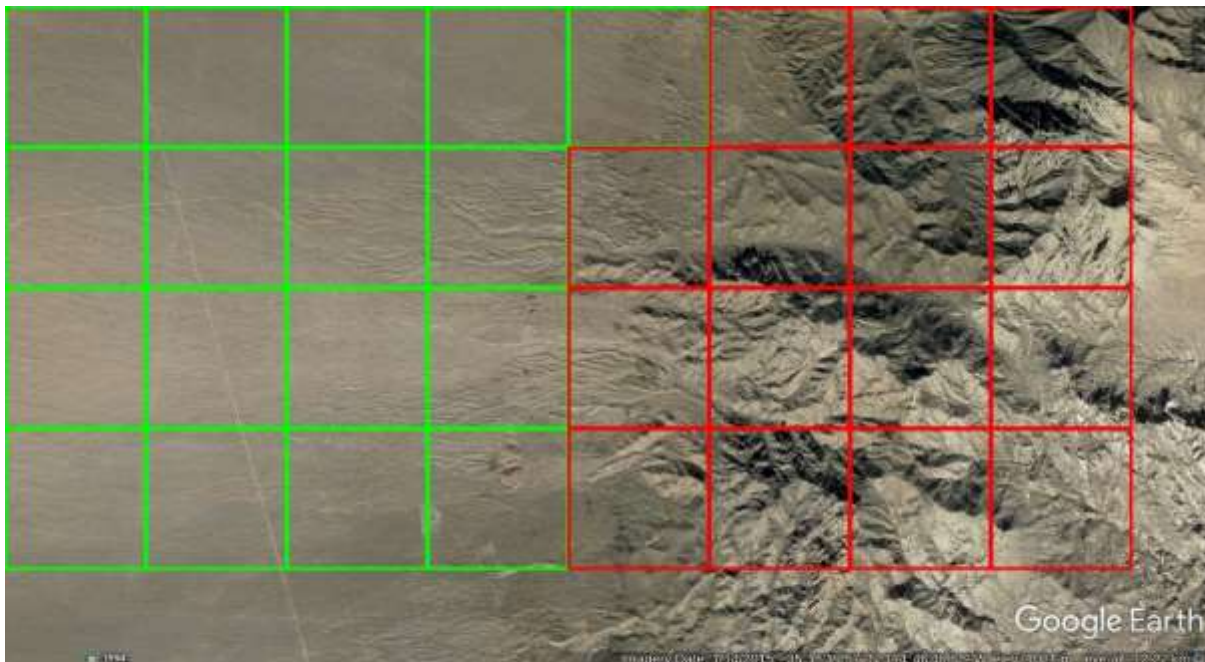
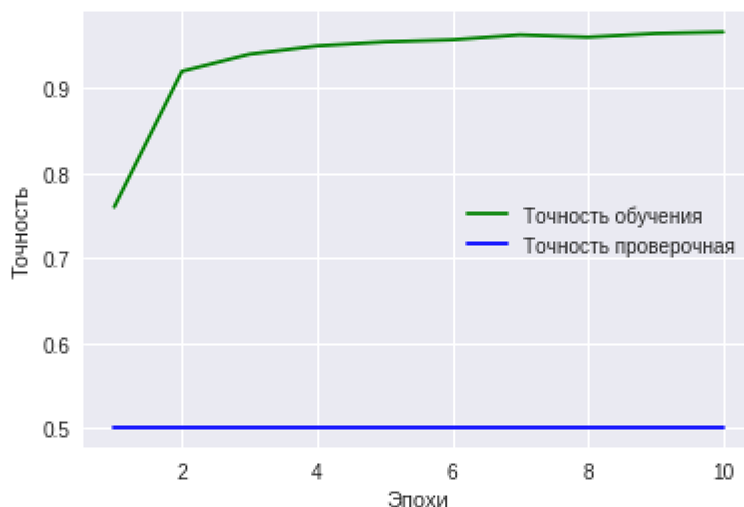


Рисунок 3.15 – Результат работы обученной нейронной сети DenseNet

```
================================================================
Total params: 24,112,513
Trainable params: 524,801
Non-trainable params: 23,587,712
________________________________________________________________
Epoch 1/10
 - 176s - loss: 0.4877 - acc: 0.7605 - val_loss: 0.7161 - val_acc: 0.5011
Epoch 2/10
 - 168s - loss: 0.2305 - acc: 0.9201 - val_loss: 0.7604 - val_acc: 0.5011
Epoch 3/10
 - 168s - loss: 0.1661 - acc: 0.9402 - val_loss: 0.7893 - val_acc: 0.5011
Epoch 4/10
 - 168s - loss: 0.1385 - acc: 0.9500 - val_loss: 0.8381 - val_acc: 0.5011
Epoch 5/10
 - 168s - loss: 0.1230 - acc: 0.9547 - val_loss: 0.8529 - val_acc: 0.5011
Epoch 6/10
 - 168s - loss: 0.1126 - acc: 0.9572 - val_loss: 0.8577 - val_acc: 0.5011
Epoch 7/10
 - 168s - loss: 0.1000 - acc: 0.9631 - val_loss: 0.8800 - val_acc: 0.5011
Epoch 8/10
 - 168s - loss: 0.1010 - acc: 0.9604 - val_loss: 0.8974 - val_acc: 0.5011
Epoch 9/10
 - 167s - loss: 0.0929 - acc: 0.9645 - val_loss: 0.8985 - val_acc: 0.5011
Epoch 10/10
 - 168s - loss: 0.0898 - acc: 0.9665 - val_loss: 0.9300 - val_acc: 0.5011
38.139455 seconds
Аккуратность на тестовых данных: 50.19%
```




```
CPU times: user 26min 2s, sys: 2min 26s, total: 28min 29s
Wall time: 29min 10s
```

Рисунок 3.16 – Вывод отладочной информации обучения сети Resnet

ЗАКЛЮЧЕНИЕ

Результаты работы носят как теоретический, так и практический характер. В теоретической части работы были рассмотрены основные архитектуры построения нейронных сетей позволяющих распознавать и обрабатывать изображения.

Проведенный анализ показал, что наиболее эффективной архитектурой неройной сети является сверточная нейронная сеть, которая обеспечивает необходимую точность распознавания изображений в режиме реального времени.

Преимуществом разработки является использование принципа передачи обучения, обеспечивающего сокращение времени необходимого на обучение нейроной сети на новом наборе данных, а так же позволяющего использовать небольшой объем данных для обучения.

В ходе работы было осуществлено привлечение сторонних вычислительных мощностей для обучения сверточной нейронной сети, результатом чего стало значительно сокращение стоимость разработки.

В ходе тестировании реализованных структур, сверточных нейронных сетей была достигнута точность равная 95,08 % на обучающем наборе данных и 93,16 % на тестовом наборе данных. Разница в процентах точности работы модели сверточной нейронной сети свидетельствует о присутствии небольшого переобучения сети, характерного для всех нейронных сетей. Полученная процент точности может быть увеличен за счет более точного выбора гиперпараметров нейронной сети и увеличения обучающего набора данных.

Получение результаты говорят возможности использования данной разработки в системах посадки на поверхность космических тел.

Разработка после соответствующей адаптации может быть применена в системах посадки на земную поверхность.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Yann LeCun, Gradient-Based Learning Applied to Document Recognition. // Proc. of the IEEE, November 1998.
2. Alex Smola, S.V.N.Vishwanatham. Introduction to Machine Learning // Cambridge University Press, 2008. P.20-32.
3. Сергея Николенко. Глубокое обучение. Погружение в мир нейронных сетей // Питер, 2018
4. An introduction to Neural Networks. http://www.explainthatstuff.com/introduction-to-neural-networks.html
5. Ben Krose, Patrick van der Smagt. An Introduction to Neural Networks. Amsterdam, November, 1996. 135 P.
6. David Kriesel. A Brief Introduction to Neural Networks. http://www.dkriesel.com/en/science/neural_networks
7. David Stutz. Understanding Convolutional Neural Networks // Seminar Report, Fakultät für Mathematik, Informatik und Naturwissenschaften. August, 2014.
8. Ganesh K. Venayagamoorthy, Teaching Neural Networks Concepts and Thir Learning Techniques // Proceedings of the 2004 American Society for Engineering Education Midwest Section Conference.
9. Ing.H.Ney, B.Leibe. Matching Algorithms for Image Recognition // RWTH Aachen University, Januar, 2010.
10. Leon Bottou. Stochastic Gradient Descent Tricks // Neural Networks: Tricks of the Trade. Volume 7700 2012 of the series Lecture Notes in Computer Science. P. 421-436.
11. Line Eikvil. Optical Character Recognition. December, 1993.
12. Matthew D.Zeiler, Dilip Krishnan, Graham W. Taylor, Rob Fergus. Deconvolutional Networks // Computer Vision and Pattern Recognition, June 13-18, 2010.
13. Vivek Shrivastava, Navdeep Sharma. Artificial Neural Networks Based Optical Character Recognition // Signal & Image Processing: An International Journal (SIPIJ) Vol.3, No.5, October 2012.
14. Yangwei Wu, Haouhua Zhao, Liqing Zhang. Image Denoising with Rectified Linear Units // Neural Information Processing. 21st International Conference, ICONIP 2014 / Eds. Chu Kiong Loo, Keem Siah Yap, Kok Wai Wong, Andrew Teoh, Kaizhu Huang. Kuching, Malaysia, November 3-6, 2014. P. 142-149.

15. Alex Graves, Jurgen Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks // NIPS 2008, Vancouver, Canada. P.545-552

16. Fisher Yu, Vladlen Koltun. Multi-Scale Context Aggregation by Dilated Convolutions, Submitted on 23 Nov 2015

17. Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller. Striving for Simplicity: The All Convolutional Net,Submitted on 21 Dec 2014

ПРИЛОЖЕНИЕ А

Листинг программы.

trainer.py

```
# -*- coding: utf-8 -*-
#%%
import numpy as np
import theano
import theano.tensor as T
import lasagne

#matplotlib inline
import matplotlib.pyplot as plt

import skimage.transform
import sklearn.cross_validation
import pickle
import tables
import os
import datetime

def LoadDatasetFromH5(filePath):
    h5file = tables.open_file(filePath, mode='r')
    X = h5file.root.data.read();
    y = h5file.root.labels.read();
    h5file.close()
    return X,y

def deprocessImage(im):
    im = im[::-1, :, :]
    im = np.swapaxes(np.swapaxes(im, 0, 1), 1, 2)
    im = (im - im.min())
    im = im / im.max()
    return im

def ShowRandomImageFromDataset(X, y) :
    randomIndex = np.random.randint(0, len(X))
    # de-normalize by adding VGGN image mean back (both are in
float64 here)
    sampleImage = X[randomIndex] + IMAGE_MEAN
```

```
    # convert to RGB
    sampleImage = sampleImage[::-1, :, :]
    # swap axes so channels are last
    sampleImage = np.swapaxes(np.swapaxes(sampleImage, 1, 0),
1, 2)

    print ("Sample image shape: " + str(sampleImage.shape))
    print ("Sample image class: " +
str(CLASSES[y[randomIndex]]))
    sampleImage = np.copy(sampleImage).astype('uint8')
    plt.imshow(sampleImage)
    return

# Seed for reproducibility
np.random.seed(42)

#%%

# Model definition for VGG-16, 16-layer model from the paper:
# "Very Deep Convolutional Networks for Large-Scale Image
Recognition"
# Original source:
https://gist.github.com/ksimonyan/211839e770f7b538e2d8
# More pretrained models are available from
# https://github.com/Lasagne/Recipes/blob/master/modelzoo/
from lasagne.layers import InputLayer, DenseLayer,
NonlinearityLayer
from lasagne.layers import Conv2DLayer as ConvLayer
from lasagne.layers import Pool2DLayer as PoolLayer
from lasagne.nonlinearities import softmax
from lasagne.utils import floatX

def build_model():
    net = {}
    net['input'] = InputLayer((None, 3, 224, 224))
    net['conv1_1'] = ConvLayer(net['input'], 64, 3, pad=1)
    net['conv1_2'] = ConvLayer(net['conv1_1'], 64, 3, pad=1)
    net['pool1'] = PoolLayer(net['conv1_2'], 2)
    net['conv2_1'] = ConvLayer(net['pool1'], 128, 3, pad=1)
    net['conv2_2'] = ConvLayer(net['conv2_1'], 128, 3, pad=1)
```

```
    net['pool2'] = PoolLayer(net['conv2_2'], 2)
    net['conv3_1'] = ConvLayer(net['pool2'], 256, 3, pad=1)
    net['conv3_2'] = ConvLayer(net['conv3_1'], 256, 3, pad=1)
    net['conv3_3'] = ConvLayer(net['conv3_2'], 256, 3, pad=1)
    net['pool3'] = PoolLayer(net['conv3_3'], 2)
    net['conv4_1'] = ConvLayer(net['pool3'], 512, 3, pad=1)
    net['conv4_2'] = ConvLayer(net['conv4_1'], 512, 3, pad=1)
    net['conv4_3'] = ConvLayer(net['conv4_2'], 512, 3, pad=1)
    net['pool4'] = PoolLayer(net['conv4_3'], 2)
    net['conv5_1'] = ConvLayer(net['pool4'], 512, 3, pad=1)
    net['conv5_2'] = ConvLayer(net['conv5_1'], 512, 3, pad=1)
    net['conv5_3'] = ConvLayer(net['conv5_2'], 512, 3, pad=1)
    net['pool5'] = PoolLayer(net['conv5_3'], 2)
    net['fc6'] = DenseLayer(net['pool5'], num_units=4096)
    net['fc7'] = DenseLayer(net['fc6'], num_units=2)
    net['fc8'] = DenseLayer(net['fc7'], num_units=1000,
nonlinearity=None)
    net['prob'] = NonlinearityLayer(net['fc8'], softmax)

    return net

# Load VGGN model weights and metadata
print ("Loading VGGN and its metadata")
# d = pickle.load(open('vgg16.pkl', 'rb'))
#d = pickle.load(open('vgg16.pkl', 'rb'), encoding='latin1')

# Build the network and fill with pretrained weights
net = build_model()
# lasagne.layers.set_all_param_values(net['prob'], d['param
values'])
# IMAGE_MEAN = d['mean value'][:, np.newaxis, np.newaxis]



#%%
# Define our own deep net and its training functions

CLASSES = ['plains', 'hills']
LABELS = {cls: i for i, cls in enumerate(CLASSES)}
```

```
# We need a fairly small batch size to fit a large network like
this in GPU memory
BATCH_SIZE = 25

# We'll connect our output classifier to the last fully
connected layer of the network
# output_layer = DenseLayer(net['fc7'], num_units=len(CLASSES),
nonlinearity=softmax)
output_layer = DenseLayer(net['fc7'], num_units=len(CLASSES),
nonlinearity=softmax)

# Define loss function and metrics, and get an updates
dictionary
X_sym = T.tensor4()
y_sym = T.ivector()

prediction = lasagne.layers.get_output(output_layer, X_sym)
loss = lasagne.objectives.categorical_crossentropy(prediction,
y_sym)
loss = loss.mean()

acc = T.mean(T.eq(T.argmax(prediction, axis=1), y_sym),
                       dtype=theano.config.floatX)

params = lasagne.layers.get_all_params(output_layer,
trainable=True)
updates = lasagne.updates.nesterov_momentum(
        loss, params, learning_rate=1, momentum=0.9)

# Compile functions for training, validation and prediction
train_fn = theano.function([X_sym, y_sym], loss,
updates=updates)
val_fn = theano.function([X_sym, y_sym], [loss, acc])
pred_fn = theano.function([X_sym], prediction)
y_tr = 1

# generator splitting an iterable into chunks of maximum length
N
def batches(iterable, N):
    chunk = []
```

```
    for item in iterable:
        chunk.append(item)
        if len(chunk) == N:
            yield chunk
            chunk = []
    if chunk:
        yield chunk

#

# Test dataset loading
X,y = LoadDatasetFromH5('asteroid_TrainingDataset.0.h5')
print ("Read {} images".format(len(X)))
print ("Created {} labels".format(len(y)))

print ("Showing a random testing image:")
rndIndex = np.random.randint(0, len(X))
plt.imshow(deprocessImage(X[rndIndex]))
print ("Sample image shape: " + str(X[rndIndex].shape))
print ("Sample image class: " + str(CLASSES[y[rndIndex]]))


#%%
# Training

X = []
y = []

trainingSets = ['0', '1', '2']
validationDatasetPath = 'asteroid_ValidationDataset.0.h5'
#logFile = open('lossLog.txt', 'w')

for epoch in range(0, 20):
    print ("\nTraining epoch {}".format(epoch))
    print ("Start time: " + str(datetime.datetime.now()))

    # Shuffle datasets and train on each
    datasetIndex = list(range(len(trainingSets)))
    np.random.shuffle(datasetIndex)
    for ids in datasetIndex:
```

```
        datasetName = trainingSets[ids]
        print ("Training {} epoch over dataset
{}".format(epoch, datasetName))
        X, y =
LoadDatasetFromH5('asteroid_TrainingDataset.{}.h5'.format(datas
etName))
        ix = list(range(len(y)))
        np.random.shuffle(ix)
        ix = ix[:len(y)//2] # use half of each dataset for
training (randomly shuffled)
        iterNum = 0
        for chunk in batches(ix, BATCH_SIZE):
            print(iterNum)
            loss = train_fn(X[chunk], y[chunk])   # returns
mean loss over the batch
            if iterNum % 40 == 0 :
                print ("loss: {}".format(loss))
                #logFile.write("{}\n".format(loss))
            iterNum += 1

    print ("Validating {} epoch over dataset {}".format(epoch,
validationDatasetPath))
    X, y = LoadDatasetFromH5(validationDatasetPath)
    ix = list(range(len(y)))
    loss_tot = 0.
    acc_tot = 0.
    for chunk in batches(ix, BATCH_SIZE):
        loss, acc = val_fn(X[chunk], y[chunk])
        loss_tot += loss * len(chunk)
        acc_tot += acc * len(chunk)

    loss_tot /= len(ix)
    acc_tot /= len(ix)

    print ("End time:
{}\n".format(str(datetime.datetime.now())))
    print ("Epoch={}, loss={}, accuracy={}".format(epoch,
loss_tot, acc_tot*100))

    print ("Saving model after epoch {}".format(epoch))
```

```
    X = []
    y = []
    pmodel = lasagne.layers.get_all_param_values(output_layer)
    pickle.dump(pmodel,
open('asteroidModel_{}_{}.pkl'.format(epoch, acc_tot*100),
'wb'), -1)

#logFile.close()

#%%
# Save trained model and test set
pmodel = lasagne.layers.get_all_param_values(output_layer)
pickle.dump(pmodel, open('asteroidModel.pkl', 'wb'), -1)

#%%

# Load model weights and metadata
d = pickle.load(open('asteroidModel.pkl', 'rb'))
dVGGN = pickle.load(open('vgg16.pkl', 'rb'))
lasagne.layers.set_all_param_values(output_layer, d)
IMAGE_MEAN = dVGGN['mean value'][:, np.newaxis, np.newaxis]

#%%
# Load Test Datset 000

# Release train set memory for loading test set
X = []
y = []
X_test,y_test =
LoadDatasetFromH5('asteroid_ValidationDataset.0.h5')
print ("Read images")
print (len(X_test))
print ("Created labels")
print (len(y_test))

#%%

print ("Showing a random testing image:")
rndIndex = np.random.randint(0, len(X_test))
plt.imshow(deprocessImage(X_test[rndIndex]))
```

```
print ("Sample image shape: " + str(X_test[rndIndex].shape))
print ("Sample image class: " + str(CLASSES[y_test[rndIndex]]))

#%%

ix_test = range(len(y_test))
loss_test_total = 0.
accuracy_test_total = 0.
for chunk in batches(ix_test, BATCH_SIZE):
    loss, acc = val_fn(X_test[chunk], y_test[chunk])
    loss_test_total += loss * len(chunk)
    accuracy_test_total += acc * len(chunk)

loss_test_total /= len(ix_test)
accuracy_test_total /= len(ix_test)
print ("Testing results: loss={}, 
accuracy={}".format(loss_test_total, accuracy_test_total * 
100))

#%%

# Plot some results from the testing set
rndIndices = np.random.randint(0, len(X_test)-1, 100)
p_y = pred_fn(X_test[rndIndices]).argmax(-1)

plt.figure(figsize=(24, 24))
for i in range(0, 100):
    plt.subplot(10, 10, i+1)
    plt.imshow(deprocessImage(X_test[rndIndices[i]]))
    true = y_test[rndIndices[i]]
    pred = p_y[i]
    color = 'green' if true == pred else 'red'
    plt.text(0, 0, true, color='black', 
bbox=dict(facecolor='white', alpha=1))
    plt.text(0, 32, pred, color=color, 
bbox=dict(facecolor='white', alpha=1))

    plt.axis('off')
```

testing.py

```
# -*- coding: utf-8 -*-

#%%
import numpy as np
import theano
import theano.tensor as T
import lasagne

#matplotlib inline
import matplotlib.pyplot as plt

import cv2
print cv2.__version__

import skimage.transform
import sklearn.cross_validation
import pickle
import os
import glob

#%%

# Model definition for VGG-16, 16-layer model from the paper:
# "Very Deep Convolutional Networks for Large-Scale Image
Recognition"
# Original source:
https://gist.github.com/ksimonyan/211839e770f7b538e2d8
# More pretrained models are available from
# https://github.com/Lasagne/Recipes/blob/master/modelzoo/
from lasagne.layers import InputLayer, DenseLayer,
NonlinearityLayer
from lasagne.layers import Conv2DLayer as ConvLayer
from lasagne.layers import Pool2DLayer as PoolLayer
from lasagne.nonlinearities import softmax
from lasagne.utils import floatX

# Seed for reproducibility
np.random.seed(42)
```

```
def deprocessImage(imageSrc):
    imageDst = imageSrc[::-1, :, :]
    imageDst = np.swapaxes(np.swapaxes(imageDst, 0, 1), 1, 2)
    imageDst = (imageDst - imageDst.min())
    imageDst = imageDst / imageDst.max()
    return imageDst

def prepareImage(image, imageMean, targetWidth, targetHeight) :
    h, w, _ = image.shape

    # Central crop to target size
    im = image[h/2-targetHeight/2 : h/2+targetHeight/2, w/2-
targetWidth/2 : w/2+targetWidth/2]

    rawim = np.copy(im).astype('uint8')

    # Shuffle axes to c01
    im = np.swapaxes(np.swapaxes(im, 1, 2), 0, 1)

    # discard alpha channel if present
    im = im[:3]

    # Convert to BGR
    im = im[::-1, :, :]

    im = im - imageMean

    return rawim, floatX(im[np.newaxis])

#%%
# Define DNN

CLASSES = ['plains', 'hills']
LABELS = {cls: i for i, cls in enumerate(CLASSES)}

TESTSET_PATH = "dataset/test"

def build_model():
    net = {}
    net['input'] = InputLayer((None, 3, 224, 224))
```

```
    net['conv1_1'] = ConvLayer(net['input'], 64, 3, pad=1)
    net['conv1_2'] = ConvLayer(net['conv1_1'], 64, 3, pad=1)
    net['pool1'] = PoolLayer(net['conv1_2'], 2)
    net['conv2_1'] = ConvLayer(net['pool1'], 128, 3, pad=1)
    net['conv2_2'] = ConvLayer(net['conv2_1'], 128, 3, pad=1)
    net['pool2'] = PoolLayer(net['conv2_2'], 2)
    net['conv3_1'] = ConvLayer(net['pool2'], 256, 3, pad=1)
    net['conv3_2'] = ConvLayer(net['conv3_1'], 256, 3, pad=1)
    net['conv3_3'] = ConvLayer(net['conv3_2'], 256, 3, pad=1)
    net['pool3'] = PoolLayer(net['conv3_3'], 2)
    net['conv4_1'] = ConvLayer(net['pool3'], 512, 3, pad=1)
    net['conv4_2'] = ConvLayer(net['conv4_1'], 512, 3, pad=1)
    net['conv4_3'] = ConvLayer(net['conv4_2'], 512, 3, pad=1)
    net['pool4'] = PoolLayer(net['conv4_3'], 2)
    net['conv5_1'] = ConvLayer(net['pool4'], 512, 3, pad=1)
    net['conv5_2'] = ConvLayer(net['conv5_1'], 512, 3, pad=1)
    net['conv5_3'] = ConvLayer(net['conv5_2'], 512, 3, pad=1)
    net['pool5'] = PoolLayer(net['conv5_3'], 2)
    net['fc6'] = DenseLayer(net['pool5'], num_units=4096)
    net['fc7'] = DenseLayer(net['fc6'], num_units=4096)
    net['fc8'] = DenseLayer(net['fc7'], num_units=1000,
nonlinearity=None)
    net['prob'] = NonlinearityLayer(net['fc8'], softmax)

    return net

# Build the VGGN network
net = build_model()

# We'll connect our output classifier to the last fully
connected layer of the network
output_layer = DenseLayer(net['fc7'], num_units=len(CLASSES),
nonlinearity=softmax)

# Load model weights and metadata
d = pickle.load(open('asteroidModel.pkl', 'rb'))
dVGGN = pickle.load(open('vgg16.pkl', 'rb'))
lasagne.layers.set_all_param_values(output_layer, d)
IMAGE_MEAN = dVGGN['mean value'][:, np.newaxis, np.newaxis]
IMAGE_W = IMAGE_H = 224
```

```
# Define loss function and metrics, and get an updates
dictionary
X_sym = T.tensor4()
y_sym = T.ivector()

prediction = lasagne.layers.get_output(output_layer, X_sym)
loss = lasagne.objectives.categorical_crossentropy(prediction,
y_sym)
loss = loss.mean()

acc = T.mean(T.eq(T.argmax(prediction, axis=1), y_sym),
                      dtype=theano.config.floatX)

params = lasagne.layers.get_all_params(output_layer,
trainable=True)
updates = lasagne.updates.nesterov_momentum(
        loss, params, learning_rate=0.0001, momentum=0.9)

# Compile functions for training, validation and prediction
train_fn = theano.function([X_sym, y_sym], loss,
updates=updates)
val_fn = theano.function([X_sym, y_sym], [loss, acc])
pred_fn = theano.function([X_sym], prediction)



#%%
# Read a few images and display
print "Reading test image and showing"
testImage = plt.imread("{}/001.jpg".format(TESTSET_PATH))
print "testImage shape:", testImage.shape
plt.figure(0)
plt.imshow(testImage)

# Test preprocesing and show the cropped input
print "Testing image preprocessing"
rawimage, image = prepareImage(testImage, IMAGE_MEAN, IMAGE_W,
IMAGE_H)
print "rawimage shape: ", rawimage.shape
print "image shape: ", image.shape
plt.figure(1)
```

```
plt.imshow(rawimage)
plt.figure(2)
plt.imshow(deprocessImage(image[0]))


#%%

imageMean = IMAGE_MEAN
tileWidth = IMAGE_W
tileHeight = IMAGE_H
classList = CLASSES
labelList = LABELS
colors = [(0,255,0), (0,0,255)]

fileSearchPattern = "{}/*.jpg".format(TESTSET_PATH)
for filePath in glob.glob(fileSearchPattern) :
    testImage = cv2.imread(filePath)
    _, filename = os.path.split(filePath)
    outImage = testImage
    height, width, _ = testImage.shape
    print "Processing {} image with width={} and 
height={}".format(filePath, width, height)
    count = 0
    for i in range(0, height/tileHeight) :
        for j in range(0, width/tileWidth) :
            x = j*tileWidth;
            y = i*tileHeight;
            tile = testImage[y:y+tileHeight, x:x+tileWidth]
            rawTile, tileForDNN = prepareImage(tile, imageMean, 
tileWidth, tileHeight)
            X = []
            X.append(tileForDNN)
            X = np.concatenate(X)
            #cv2.imwrite("{}_{}_{}".format(x,y,filename), 
deprocessImage(tileForDNN[0]))

            prediction = pred_fn(X)[0]
            #label = pred_fn(X).argmax(-1)
            #color = colors[label[0]]
            print prediction[labelList["plains"]]
```

```
            if prediction[labelList["plains"]]>0.99 :
                color = colors[labelList["plains"]]
            else :
                color = colors[labelList["hills"]]

            cv2.rectangle(outImage,(x+1,y+1),(x+tileWidth-
1,y+tileHeight-1),color,3)

    outFilename = "result_{}".format(filename)
    print "Writing out {} file".format(outFilename)
    cv2.imwrite(outFilename, outImage)
```