

**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ
СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 14.12.2021 г.

Утвърждавам:.....

Дата на предаване: 14.03.2022 г.

/проф. д-р инж. Т. Василева/

ЗАДАНИЕ
за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ

по професия код 481020 „Системен програмист“

специалност код 4810201 „Системно програмиране“

на ученика Стефан Светославов Сотиров от 12 В клас

1. Тема: Система за засичане на замърсявания в контейнери

2. Изисквания:

Изследване, анализ и обработка на данни (снимки на силно замърсени, умерено замърсени и чисти контейнери), оценяване и сравняване на различните методи за засичане на замърсяванията, и внедряване на имплементираното решение в тестова или работна среда.

3. Съдържание

3.1 Теоретична част

3.2 Практическа част

3.3 Приложение

Дипломант :.....

/ Стефан Сотиров /

Ръководител:.....

/ Евгени Димов /

Директор:.....

/ доц. д-р инж. Ст. Стефанова /

ТЕХНОЛОГИЧНО УЧИЛИЩЕ
ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“

специалност код 4810201 „Системно програмиране“

Тема: Система за засичане на замърсявания в контейнери

Дипломант:

Стефан Светославов Сотиров

Дипломен ръководител:

Евгени Димов

СОФИЯ

2022

Мнение на научен ръководител

Дипломантът Стефан Сотиров е представил завършена дипломна работа. По време на разработването на дипломната работа са изследвани различни подходи за засичане на замърсена щайга. Всички изследвани са сравнени и е избран единствен подход за възможно най-добро представяне. Трудността на поставената дипломна задача е висока, а темата е изключително актуална. Дипломната работа в текущото състояние може да послужи като доказателство за работещта концепция, която да бъде представена на потенциални бизнес клиенти, а отделно подлежи на усъвършенстване.

Увод

През последните няколко години дълбокото обучение (deep learning) оказва огромно влияние върху различни области на бизнеса. Една от най - актуалните теми в софтуерната индустрия в последно време е computer vision, способността на компютрите да разбират изображения и видеоклипове сами.

Самоуправляващите се автомобили, биометричните данни и разпознаването на лица разчитат на компютърното зрение, за да работят. В основата на компютърното зрение е обработката на изображения.

Мачинното самообучение (Machine learning) е метод за анализ на данни, който автоматизира изграждането на аналитични модели. Това е вид artificial intelligence, базиран на идеята, че системи могат да се учат от данни, да изчислят оптимални параметри за различни видове статистически модели и да вземат решения с минимална човешка намеса.

Дълбокото обучение е вид машинно обучение, който имитира начина, по който хората получават определени видове знания. Дълбокото обучение е важен елемент от статистиката и прогнозното моделиране. Той е изключително полезен за учените и инженерите, които имат задачата да събират, анализират и интерпретират големи количества данни. Дълбокото обучение прави този процес възможен.

Разрастването на технологиите за дълбокото обучение доведе до бързото ускоряване на компютърното зрение в проекти с отворен код, което само увеличи нуждата от инструменти за обработка на изображения. Търсенето на професионалисти с ключови умения в технологиите за дълбоко обучение нараства с бързи темпове всяка година.

С този проект ще се опитам да илюстрирам горепосочените концепции в приложение и да покажа резултата от развитието в сферата на изкуствения интелект и обработването на снимки. В този процес ще разгледам няколко алгоритъма за обработка на снимки и шест модела за предсказване на база обработките.

Първа глава

Теоретична част

1.1 Алгоритми за обработка на снимки

1.1.1 Въведение

Цифровото изображение е съставено от елементи на картината, известни също като пиксели, всеки с крайни, дискретни количества на числово представяне за неговия интензитет или ниво на сиво, което е изход от неговите двуизмерни функции, подавани като вход от неговите пространствени координати, обозначени с x , y съответно по оста x и y .

Важно развитие в технологията за компресиране на цифрово изображение е дискретната косинусова трансформация (DCT), техника за компресиране със загуби. DCT компресията се използва в JPEG, който компресира изображенията до много по-малки размери на файлове и се е превърнало в най-широко използвания файлов формат на изображения в интернет пространството.

1.1.2 RGB моделът

Цветният модел RGB е адитивен цветови модел, в който червените, зелените и сините основни цветове на светлината се добавят заедно по различни начини за възпроизвеждане на широк спектър от цветове. Името на модела идва от инициалите на трите допълнителни основни цвята, червен, зелен и син.

Основната цел на цветния модел RGB е за отчитане, представяне и показване на изображения в електронни системи, като телевизори и компютри, въпреки че се използва и в конвенционалната фотография.

RGB е цветен модел, зависим от устройството: различните устройства откриват или възпроизвеждат дадена RGB стойност по различен начин, тъй като цветните елементи (като фосфор) и тяхната реакция на отделните нива на червено, зелено и синьо варират от производителя до производителя, или дори в едно и също устройство с течение на времето. По този начин RGB стойност не дефинира един и същ цвят на всички устройства без някакъв вид управление на цветовете.

Типичните RGB входни устройства са цветни телевизионни и видеокамери, скенери за изображения и цифрови фотоапарати. Типичните RGB изходни устройства са телевизори с различни технологии (CRT, LCD, плазма, OLED, квантови точки и др.), дисплеи на компютри и мобилни телефони, видеопроектори, многоцветни LED дисплеи и големи екрани като Jumbotron. Цветните принтери, от друга страна, не са RGB устройства, а устройства с изваждащи цветове, които обикновено използват цветния модел CMYK.

1.1.3 Снимката като файл

Както всяка информация в една компютърна система, снимките представляват памет, подредена по определен начин. Те са нищо повече от байтове, записани в определена последователност, на определен запамятаващ

компонент от компютърната система. Тази последователност (различна в зависимост от формата на снимката) бива интерпретирана от контролерите на графичните устройства, в следствие на което изображението излиза на показ върху екрана. В масовия случай форматът на байтовете е именно RGB (другият формат - палитрите вече е остарял).

По дефиниция, единственото, с което програмистът се занимава е управление на памет - независимост от типа и (Primary, Secondary, Registers, etc.). Това предполага, че програмистът може да управлява и паметта на снимки, както на всякакви файлове (по абсолютно идентичен начин).

В модерните формати, по които се записват и четат снимки, за всеки канал (red, green, blue (понякога и alpha - за прозрачност)) се заделя по един байт (за ефикасна обработка) и са разположени последователно в паметта. Това означава, че възможните стойности на тези байтове са от 0 до 255 (2^8 на 8-ма степен (или 256) на брой, понеже един байт се състои от 8 бита).

Ако се разгледа една съвкупност от такива байтове (RGB(A)), се получава единица, наречена пиксел (на модерен дисплей един пиксел представлява съвкупност от три светодиода). Съответно, променяйки стойностите на байтовете на един пиксел, се променят цвета и яркостта на пиксела.

Тук ще се разгледат алгоритми за обработка на снимки, които са приложени при разработката на проекта.

1.1.4 Копиране на изображения

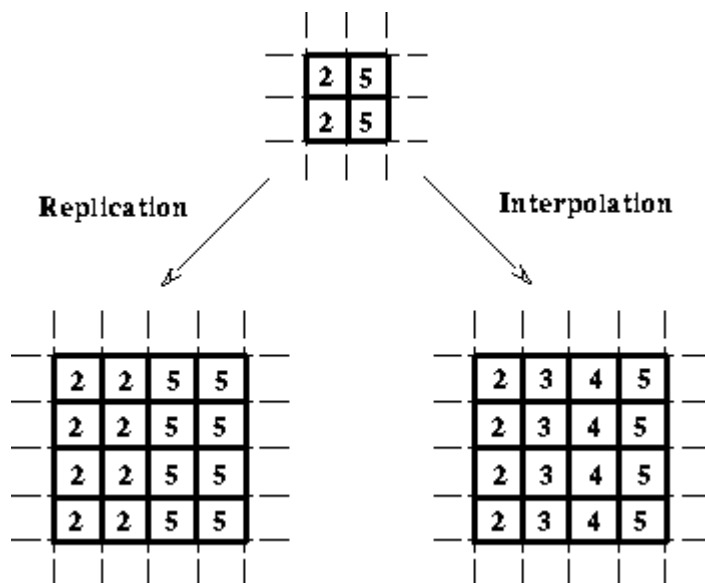
Копирането на изображения е изключително тривиална операция, ако не възможно най - простата. Единственото, което трябва да се извърши е копиране на пикселите (стойностите на RGB(A) байтовете) от един буфер в друг със същите размери (височина и широчина на снимката). В днешно време процесорите могат да изпълняват копирания от по 8 байта наведнъж, а някои процесори поддържат SIMD (Same Instruction Multiple Data) операции, които свеждат копирането до 32 или повече байта наведнъж. Операцията се изпълнява много бързо и е лесна за имплементация. Копиране винаги се случва при записване от оперативна към вторична памет.

1.1.5 Преоразмеряване (скалиране, мащабиране) на снимки

Преоразмеряването на изображението е необходимо, когато трябва да се увеличи или намали общия брой пиксели. Мащабирането на изображението може да се интерпретира като форма на повторно семплиране (интерполиране) на изображението. Интерполацията работи чрез използване на известни данни за оценка на стойности в неизвестни точки. Интерполацията на изображението се опитва да постигне най-добро приближение на интензитета на пиксела въз основа на стойностите в околните пиксели. Общите алгоритми за интерполация могат да бъдат групирани в две категории: адаптивни и не адаптивни.

Адаптивните методи се променят в зависимост от това какво интерполират, докато не адаптивните методи третират всички пиксели еднакво. Не адаптивните

алгоритми включват: най-близък съсед, билинеен, бикубичен, сплайн, sinc, lanczos и други. Адаптивните алгоритми включват много собствени алгоритми в лицензиран софтуер като: Qimage, PhotoZoom Pro и Genuine Fractals. Напоследък се откриват много нови методи за скалиране на изображения с помощта на изкуствен интелект. В по - разбираем вид, преоразмеряването представлява или интерполация, или репликация на пиксели:



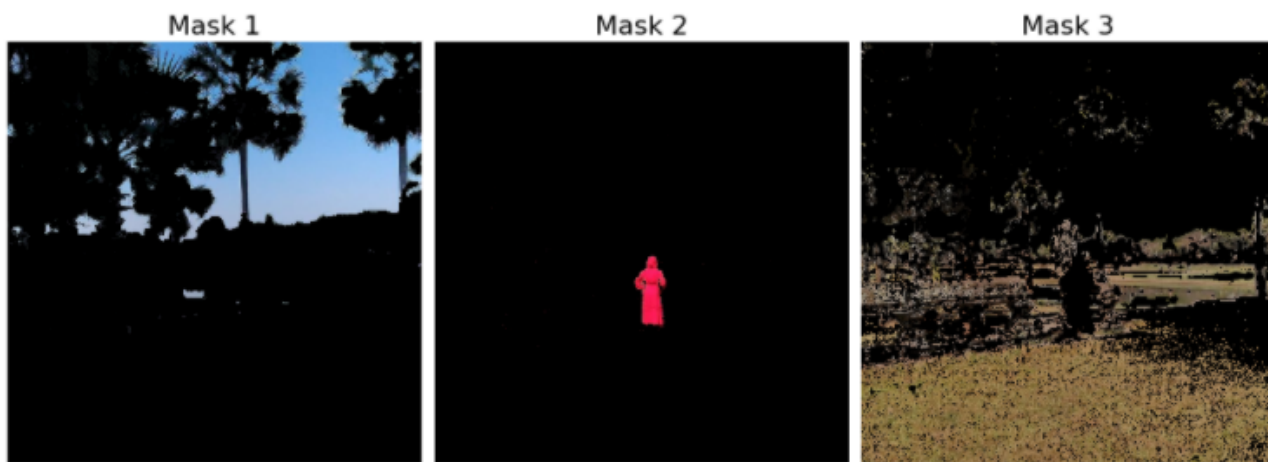
1.1.6 Маскиране на снимки

Маскирането на изображения представлява филтриране на изображения. Филтрите целят да изолират пиксели на база някакво условие. Например, може да искаме да изолираме само червените пиксели от дадено изображение. Това може да постигнем, когато извършим операцията “маскиране”, която ще ни върне изображение, което съдържа само червените пиксели от оригиналното. Това става, като се копират само байтовете на зелените пиксели на съответните

позиции в нов буфер от памет, а останалите се зануляват. Нека това е оригиналната снимка:



Така ще изглеждат различни маски, приложени върху снимката:



Маска 1 изолира сините пиксели в определен диапазон от снимката, маска 2 изолира само ярко червените, а маска три, само зелените.

1.1.7 Откриване на ръбове

Откриването на ръбове е техника за извличане на полезна структурна информация от различни визуални обекти и драстично намаляване на количеството данни, които трябва да се обработват. Широко прилагано е в различни системи за computer vision. Откриването на ръбове включва различни математически методи, които целят идентифициране на ръбове, криви в цифрово изображение, при което яркостта на изображението се променя рязко или, по-формално, има прекъсвания. То е основен инструмент в обработката на изображения, машинното и компютърното зрение, особено в областите на откриване на характеристики и извличане на характеристики.

Ръбовете, извлечени от двуизмерно изображение на триизмерна сцена, могат да бъдат класифицирани като зависими от гледна точка или независими от гледна точка. Независимият от гледна точка ръб обикновено отразява присъщите свойства на триизмерните обекти, като повърхностни маркировки и форма на повърхността. Зависим от гледна точка ръб може да се промени с промяната на гледната точка и обикновено отразява геометрията на сцената, като обекти, които се запущат един друг.

За да се илюстрира защо откриването на ръбове не е тривиална задача, може да се разгледа проблемът с откриването на ръбове в следния едномерен пример. Тук може интуитивно да се каже, че трябва да има граница между 4-ия и 5-ия пиксел:

5	7	6	4	152	148	149

Ако разликата в интензитета беше по-малка между 4-ия и 5-ия пиксел и ако разликите в интензитета между съседните съседни пиксели бяха по-високи, не би било толкова лесно да се каже, че трябва да има ръб в съответния регион. Освен това може да се твърди, че този случай е такъв, в който има няколко ръба.

5	7	6	41	113	148	149

Следователно не винаги е лесно да се посочи твърдо специфичен праг за това колко голяма трябва да бъде промяната на интензитета между два съседни пиксела, за да се каже, че трябва да има граница между тези пиксели. Всъщност това е една от причините, поради които откриването на ръбове може да бъде нетривиален проблем, освен ако обектите в сцената не са особено прости и условията на осветеност могат да бъдат добре контролирани.

Има много методи за откриване на ръбове, но повечето от тях могат да бъдат групирани в две категории, базирани на търсене и базирани на нула. Методите, базирани на търсене, откриват ръбове, като първо изчисляват мярка за сила на ръба, обикновено произведен израз от първи ред, като величината на наклона, и след това търсят локални насочени максимуми на величината на наклона, използвайки изчислена оценка на локалната ориентация на край, обикновено посоката на наклона. Методите, базирани на нула, търсят пресичане

на нула в произведен израз от втори ред, изчислен от изображението, за да намерят ръбове, обикновено пресичането на нулата на лапласиана или пресичането на нула на нелинеен диференциален израз. Като стъпка от предварителна обработка до откриване на ръбове, почти винаги се прилага етап на изглаждане на изображението (обикновено изглаждане на Гаус).

Методите за откриване на ръбове, които са публикувани, се различават основно по видовете изглаждащи филтри, които се прилагат, и начина, по който се изчисляват мерките за сила на ръба. Тъй като много методи за откриване на ръбове разчитат на изчисляване на наклони на изображението, те също се различават по видовете филтри, използвани за изчисляване на оценките на наклона в посоките x и y .

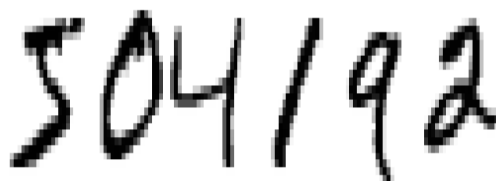
Откриването на ръбове може да се разглежда като много сложна маска (филтър) върху оригиналното изображение. Ето пример:



1.2 Принципи на машинно самообучение и невронни мрежи

1.2.1 Въведение

Човешката зрителна система е едно от чудесата на света. Илюстрирана е следната последователност от ръкописни цифри:



Повечето хора без усилие разпознават тези цифри като 504192. Тази лекота е измамна. Във всяко полукълбо на мозъка си хората имат първичен зрителен кортекс, известен също като V1, съдържащ 140 милиона неврони, с десетки милиарди връзки между тях. И все пак човешкото зрение включва не само V1, а цяла серия от зрителни кортици - V2, V3, V4 и V5 - извършващи прогресивно по-сложна обработка на изображения. Хората носят в главите си суперкомпютър, настроен от еволюцията в продължение на стотици милиони години и отлично адаптиран за разбиране на визуалния свят. Разпознаването на ръкописни цифри не е лесно. По-скоро, хората, са невероятно, удивително добри в разбирането на това, което очите им показват. Но почти цялата тази работа се извършва несъзнателно. И затова обикновено не се оценява колко труден проблем решават техните визуални системи.

Трудността на визуалното разпознаване на модели става очевидна, при опит за писане на компютърна програма за разпознаване на цифри като тези по-горе. Това, което изглежда лесно, изведнъж става изключително трудно. Простите интуиции за това как се разпознава формите - "9 има цикъл в горната част и вертикален щрих в долния десен ъгъл" - се оказва, че не е толкова лесно да се изрази алгоритмично. Опитът за направа на такива правила прецизни, бързо се губи в множество от изключения, предупреждения и специални случаи. Изглежда безнадеждно.

Невронните мрежи подхождат към проблема по различен начин. Идеята е да се вземат голям брой ръкописни цифри, известни като примери за обучение или входни данни,



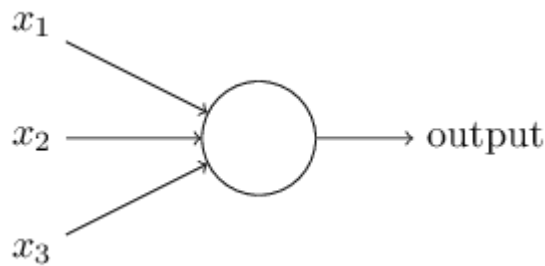
и след това да се разработи система, която може да се учи от тези примери за обучение. С други думи, невронната мрежа използва примерите за автоматично извеждане на правила за разпознаване на ръкописни цифри. Така че, докато тук са показани само 100 обучителни цифри, може да е възможно да се изгради по-добър разпознавател на ръкописен текст, като се използват хиляди, дори милиони или милиарди примери за обучение.

Фокусът е върху разпознаването на ръкописен текст, защото това е отличен прототип на проблем за изучаване на невронните мрежи, който е разгледан обилно в интернет пространството. Освен това, проблемът служи като прекрасно въведение. По пътя ще се разгледат много ключови идеи за невронни мрежи, включително два важни типа изкуствен неврон (персептрон и сигмоиден неврон) и стандартния алгоритъм за обучение за невронни мрежи, известен като градиентно спускане. Важно е обяснението защо нещата се правят така, както са, и изграждането на интуиция за невронните мрежи. Всичко това е необходимо, за да може да се разясни дълбокото обучение и защо то е толкова важно.

1.2.2 Персептрон

Персептроните са разработени през 50-те и 60-те години на миналия век от учения Франк Розенблат, вдъхновени от по-ранна работа на Уорън Маккулок и Уолтър Питс. Днес по - често се използват други модели на изкуствени неврони - сигмоидни неврони, но за разбирането защо сигмоидните неврони са дефинирани по начина, по който са, си струва да се отдели време за разглеждане първо на персептроните.

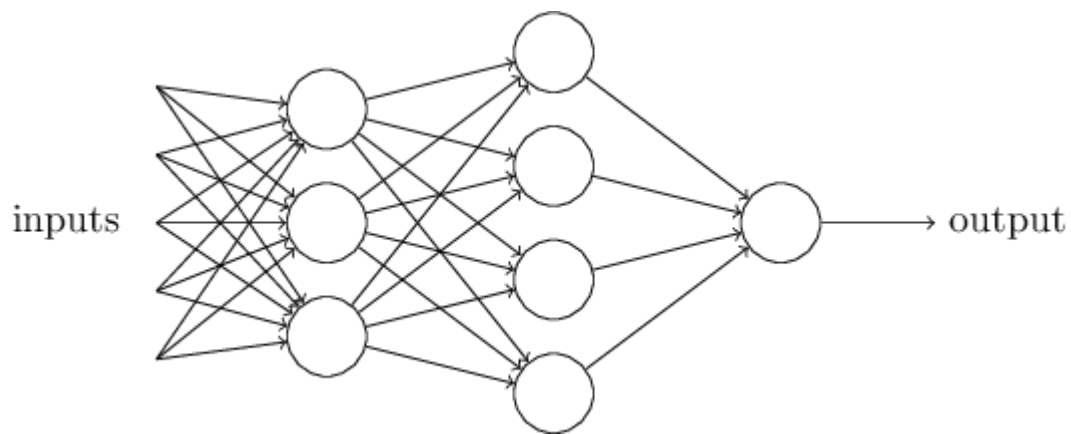
Персептронът приема няколко двоични (бинарни) входа, x_1, x_2, \dots , и произвежда единичен двоичен изход:



В показания пример персептронът има три входа, x_1, x_2, x_3 . Като цяло може да има повече или по-малко входове. Розенблат е предложил просто правило за изчисляване на изхода. Той е въвел тегла, w_1, w_2, \dots , реални числа, изразяващи важността на съответните входове за изхода. Изходът на неврона, 0 или 1, се определя от това дали претеглената сума $\sum_j w_j x_j$ е по-малка или по-голяма от някаква прагова стойност. Точно като теглата, прагът е реално число, което е параметър на неврона. Показано по-точно алгебрично:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Това е основният математически модел. Начин, по който можете да се гледа на персептрона, е, че това е устройство, което взема решения чрез претегляне на сведения.



В тази мрежа първата колона от персептрони - това, което се нарича първия слой от персептрони - взима три много прости решения чрез претегляне на входните сведения. Всеки от персептроните във втория слой взима решение, като претегля резултатите от първия слой на вземане на решения. По този начин персептрон във втория слой може да вземе решение на по-сложно от персептроните в първия слой. Още по-сложни решения могат да се вземат от персептрона в третия слой. По този начин многослойна мрежа от персептрони може да участва във вземане на сложни и абстрактни решения.

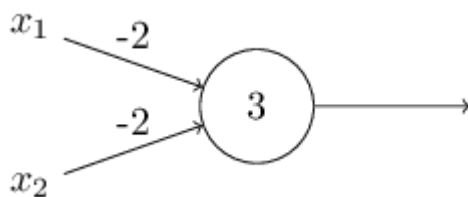
В мрежата на персептроните изглежда, сякаш те притежават множество изходи. Всъщност те имат единичен изход. Стрелките за множество изходни сигнали са просто полезен начин за индикация, че изходът от персептрон се използва като вход към няколко други персептрона. Това е по-малко удобно от начертаването на една изходна линия, която след това се разделя.

Условието $\sum_j w_j x_j > threshold$ е сложно за четене, може да се направят две промени в нотацията, за да се опрости. Първата промяна е да се замени $\sum_j w_j x_j$ със скалярно произведение, $w \cdot x \equiv \sum_j w_j x_j$, където w и x са вектори, чиито

компоненти са съответно теглата и входните данни. Втората промяна е да се премести прагът от другата страна на неравенството, което се дефинира като отклонение на персептрона, $b \equiv -\text{праг}$. Използвайки отклонението вместо прага, правилото за персептрон може да бъде пренаписано:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Персептроните могат да бъдат използвани е за изчисляване на елементарните логически функции, които обикновено смятаме за базово изчисление, функции като AND, OR и NAND. Например, да предположим, че имаме персептрон с два входа, всеки с тегло -2 и общо отклонение от 3. Ето как изглежда схематично:



Тогава може да се види, че вход 00 произвежда изход 1, тъй като $(-2) * 0 + (-2) * 0 + 3 = 3$ е положително. Подобни изчисления показват, че входовете 01 и 10 произвеждат изход 1. Но входът 11 произвежда изход 0, тъй като $(-2) * 1 + (-2) * 1 + 3 = -1$ е отрицателно. И така нашият персептрон внедрява NAND порта.

Примерът с NAND показва, че могат да се използват персептрони за изчисляване на прости логически функции. Всъщност могат да се използват мрежи от персептрони, за изчисление на всяка логическа функция. Причината е, че портата NAND е универсална за изчисление, тоест може да се изгради всяко

изчисление от съвкупност NAND порти. Изчислителната универсалност на персептроните е едновременно впечатляваща и разочароваща. Впечатляваща е, защото показва, че мрежите от персептрони могат да бъдат толкова мощни, колкото всяко друго изчислително устройство. Но също така е разочароваща, защото създава впечатлението, че персептроните са просто нов тип NAND порта. Ситуацията обаче е по - различна, отколкото предполага тази гледна точка. Оказва се, че може да се измислят алгоритми за обучение, които могат автоматично да настройват теглата и отклоненията на мрежа от изкуствени неврони. Тази настройка се случва в отговор на външни стимули, без директна намеса на програмист. Тези алгоритми за обучение позволяват използването на изкуствени неврони по начин, който е коренно различен от конвенционалните логически порти. Вместо изрично да излагат верига от NAND и други порти, невронните мрежи могат просто да се научат да решават проблеми, понякога проблеми, при които би било изключително трудно директно да се проектира конвенционална верига.

1.2.3 Сигмоидни неврони

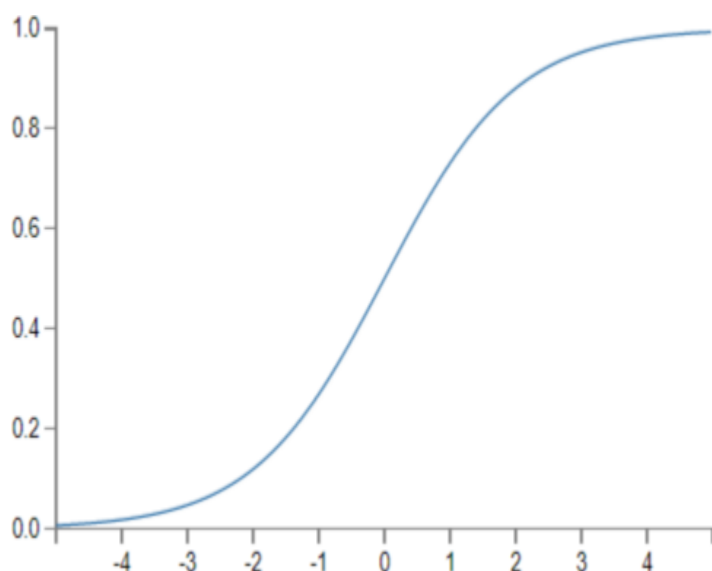
Но как се измислят такива алгоритми за невронна мрежа? Като пример, може да се вземе мрежа от персептрони, които биха се използвали, за да се научи решаването на някакъв проблем. Например, входовете към мрежата могат да бъдат необработените пикселни данни от сканирано, ръкописно изображение на цифра. И би било желано мрежата да научи тегла и отклонения, така че изходът

от мрежата да класифицира правилно цифрата. За да се наблюдава как може да работи обучението, се прави малка промяна в някакво тегло (или отклонение) в мрежата. Това, което се търси, е тази малка промяна в теглото да причини само малка съответна промяна в изхода от мрежата. Проблемът е, че това не се случва, когато мрежата съдържа персептрони. Всъщност малка промяна в теглата или отклонението на всеки един персептрон в мрежата понякога може да доведе до пълно преобръщане на изхода на този персептрон, примерно от 0 до 1. Това обръщане може след това да доведе до масивна промяна в поведението на останалата част от мрежата - напълно да се измени по много сложен начин.

Този проблем може да бъде преодолян, чрез въвеждане на нов тип изкуствен неврон, наречен сигмоиден неврон. Сигмоидните неврони са подобни на персептроните, но са модифицирани така, че малки промени в теглото и отклонението причиняват само малка промяна в техния изход. Това е решаващият факт, който ще позволи на мрежа от сигмоидни неврони да се учи. Подобно на персептрон, сигмоидният неврон има входове, x_1, x_2, \dots . Но вместо да бъдат само 0 или 1, тези входове могат също да приемат всякакви стойности. Така, например, 0,123... е валиден вход за сигмоиден неврон. Също като персептрон, сигмоидният неврон има тегла за всеки вход, w_1, w_2, \dots , и общо отклонение, b . Но изходът не е 0 или 1. Вместо това е $\sigma(w \cdot x + b)$, където σ се нарича сигмоидна функция и се дефинира по следния начин:

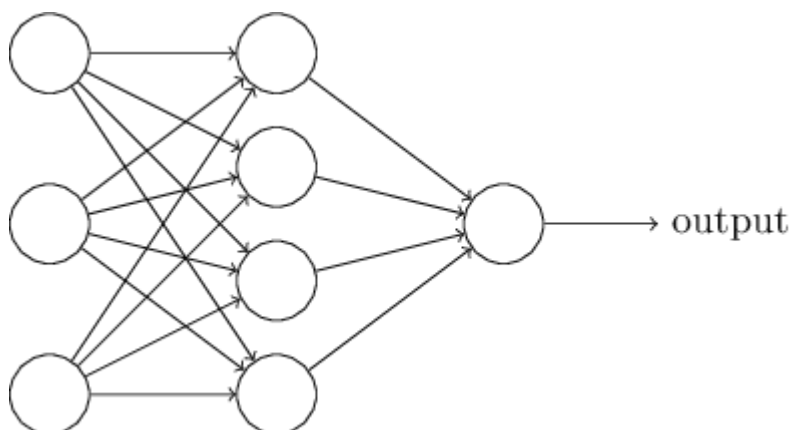
$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

Графиката изглежда по следния начин:



1.2.4 Архитектура на невронна мрежа

Нужна е проста илюстрация, за показ на основните слоеве на една невронна мрежа:



Както бе споменато по-рано, най-левият слой в тази мрежа се нарича входен слой, а невроните в слоя се наричат входни неврони. Най-десният или изходен

слой съдържа изходните неврони или, както в този случай, един единствен изходен неврон. Средният слой се нарича скрит слой, тъй като невроните в този слой не са нито входи, нито изходи. Мрежата по-горе има само един скрит слой, но някои мрежи имат няколко скрити слоя. Дизайнът на входните и изходните слоеве в мрежата обикновено е ясен и зависи от проблема налице. Дизайнът на входните слоеве се прави според данните които се използват за да се правят предсказания, а за изходния слой - според това което се предсказва. За сметка на това, дизайнът на скритите слоеве може да бъде доста сложен и непредвидим. По-точно, не е възможно да се обобщи процеса на проектиране за скритите слоеве с няколко прости правила. Вместо това специалисти са разработили много евристики за проектиране на скритите слоеве, които помагат на хората да извлекат поведението, което искат, от мрежите си. Например, такива евристики могат да се използват, за да се определи как да се компрометира броят на скрити слоеве спрямо времето, необходимо за обучение на мрежата.

Досега се разглеждаха невронни мрежи, при които изходът от един слой се използва като вход към следващия слой. Такива мрежи се наричат невронни мрежи с пренасочване (feedforward networks). Това означава, че няма цикли в мрежата - информацията винаги се подава напред, никога не се изпраща обратно. Съществуват обаче и други модели на изкуствени невронни мрежи, в които са възможни вериги за обратна връзка. Тези модели се наричат рекурентни невронни мрежи. Идеята в тези модели е да има неврони, които се задействат за известно време, преди да станат неподвижни. Това задействане може да

стимулира други неврони, които могат да се задействат малко по-късно, също за ограничено време. Това кара още повече неврони да се задействат и така с течение на времето получаваме каскада от неврони. Циклите не създават проблеми в такъв модел, тъй като изходът на неврон влияе само на входа му в някакъв по-късен момент, а не мигновено. Рекурентните невронни мрежи са по-малко използвани от мрежите с пренасочване, отчасти защото алгоритмите за обучение за рекурентни се мрежи са по - рядко приложими. Но те са изключително интересни - те са много по-близо по дизайн до начина, по който функционира човешкият мозък, отколкото мрежите за пренасочване. И е възможно повтарящите се мрежи да решават сложни проблеми, които биват решени с много голяма трудност или дори невъзможно чрез мрежи с пренасочване като например time-series predictions (прогнози за времеви серии) - предсказване на цената на тока за следващите 2 години месец по месец, предсказване на следващия артикул който клиент в онлайн магазин би си купил за recommendation engine или машинно интерпретиране на текст като поредица от думи, при които е важна поредността.

1.2.5 Обучение с градиентно спускане (спускане по наклон)

Нека x бъде входа на обучение, а $y = y(x)$ - съответния желан изход. Това, което би се искало, е алгоритъм, който позволява да се намират тегла и отклонения, така че изходът от мрежата да е приблизително $y(x)$ за всички

входове за обучение x . За да се определи количествено колко добре се постигама тази цел, се дефинира функция на разходите (cost function):

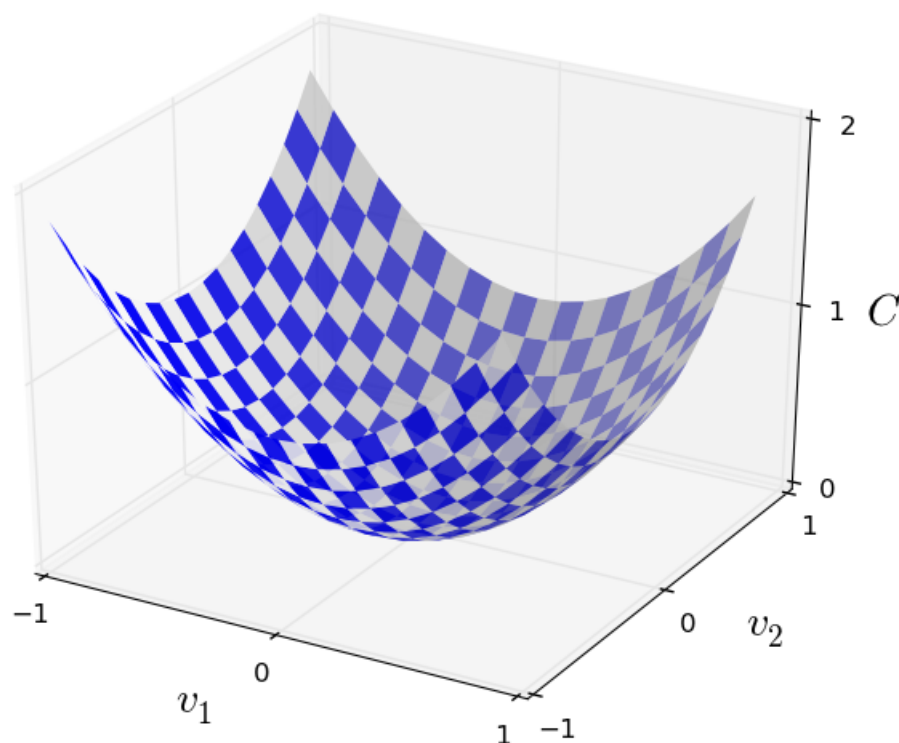
$$C(w, b) \equiv \frac{1}{2n} \sum_x (||y(x) - a||)^2$$

Тук w означава сбора на всички тегла в мрежата, b са всички отклонения, n е общият брой на входовете за обучение, a е векторът на изходите от мрежата, когато x е входен, и сумата е върху всички входове за обучение, x . Разбира се, изходът a зависи от x , w и b , но за да е проста нотацията, не е посочена изрично тази зависимост. Нотацията $||v||$ просто обозначава обичайната функция за дължина на вектор v . C е квадратична функция на разходите; понякога е известен също като средна квадратна грешка или просто MSE (mean squared error).

Проверявайки формата на квадратичната функция на разходите, се вижда, че $C(w, b)$ е неотрицателна, тъй като всеки член в сумата е неотрицателен. Освен това цената $C(w, b)$ става малка, т.е. $C(w, b) \approx 0$, точно когато $y(x)$ е приблизително равно на изхода, a , за всички входове за обучение, x . Така че алгоритъмът за обучение е свършил добра работа, ако може да намери тегла и отклонения, така че $C(w, b) \approx 0$. Обратно, не се справя толкова добре, когато $C(w, b)$ е голям - това би означавало, че $y(x)$ не е близо до изхода a за голям брой входове. Така че целта на алгоритъма за обучение ще бъде да минимизира разходите $C(w, b)$ като функция на теглата и отклоненията. С други думи, търсят се набор от тежести и отклонения, които правят разходите възможно най-ниски. Това се случва с помощта на алгоритъм, известен като градиентно спускане.

В по-голямата си част правенето на малки промени в тежестите и отклоненията няма да доведе до никаква промяна в броя на тренировъчните изображения, класифицирани правилно. Това затруднява разбирането как да се променят теглата и отклоненията, за да се постигне подобрена производителност. Ако вместо това се използва плавна функция на разходите като квадратичната цена, се оказва лесно за разбиране как се правят малки промени в теглата и отклоненията, за да се постигне подобрене в цената. Ето защо първо фокусът е върху минимизирането на квадратичната цена и едва след това се проверява точността на класификацията.

За предположение, прави се опит за минимизиране на някаква функция, $C(v)$. Това може да бъде всяка функция с реална стойност от много променливи, $v = v_1, v_2, \dots$. Нотацията w и b с v е заменена, с цел показване, че това може да бъде всяка функция - вече не се мисли конкретно в контекста на невронните мрежи. За да се минимизира $C(v)$, помага представата за C като функция само на две променливи, които може да се нарекат v_1 и v_2 :



Това, което се желае, е намирането къде C постига своя глобален минимум. Сега, разбира се, за функцията, изобразена по-горе, можем да бъде разгледана графиката и да бъде намерен минимума. Обща функция, C , може да е сложна функция от много променливи и обикновено няма да е възможно просто да се погледне графиката, за да се намери минимумът.

За щастие има добра аналогия, която предполага алгоритъм, който работи доста добре. Образно, мисли се за функцията като вид долина. И си визуализира топка, търкаляща се по склона на долината. Ежедневният опит показва, че

топката в крайна сметка ще се претърколи до дъното на долината. Ще бъде избрана произволно начална точка за (въображаема) топка и след това ще се симулира движението на топката, докато се търкаля до дъното на долината.

Може да се направи тази симулация просто като изчислят производни (и може би някои втори производни) на C – тези производни биха казали всичко, което трябва да се знае за локалната „форма“ на долината и следователно как трябва да се търкаля нашата топка. За да бъде този въпрос по-точен, трябва да се помисли какво се случва, когато топката бива преместена малко количество Δv_1 в посока v_1 и малко количество Δv_2 в посока v_2 . Изчислението казва, че C се променя, както следва:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2$$

Ще се намери начин да се избере Δv_1 и Δv_2 , така че да направим ΔC отрицателен; т.е. ще се изберат, така че топката да се търкаля надолу в долината. За яснота как се прави такъв избор, помага да се дефинира Δv като вектор на промените във v , $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, където T е операцията за транспониране, превръщаща векторите-редове в вектори-столби. Ще се дефинира също градиента на C като вектор на частични производни, $\nabla C \equiv (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$. С тези дефиниции изразът за ΔC може да бъде пренаписан по следния начин:

$$\Delta C \approx \nabla C \cdot \Delta v$$

Това уравнение помага да се обясни защо ∇C се нарича градиентен вектор: ∇C свързва промените във v с промените в C , точно както би се очаквало да направи

нещо, наречено градиент. Но това, което е наистина вълнуващо в уравнението, е, че позволява да се види как се изберем Δv , така че да се направи ΔC отрицателен.

В частност, нека $\Delta v = -\eta \nabla C$, където η е малък, положителен параметър

(известен като скорост на обучение). Тогава $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta (\|\nabla C\|)^2$.

Тъй като $(\|\nabla C\|)^2 \geq 0$, това гарантира, че $\Delta C \leq 0$, т.е., C винаги ще намалява,

никога няма да се увеличава, при промяна на v . Точно това е свойството, което се

търси. Това означава, че ще се използва $\Delta v = -\eta \nabla C$, за да се изчисли стойност

за Δv , след което ще се премести позицията на топката v с това количество:

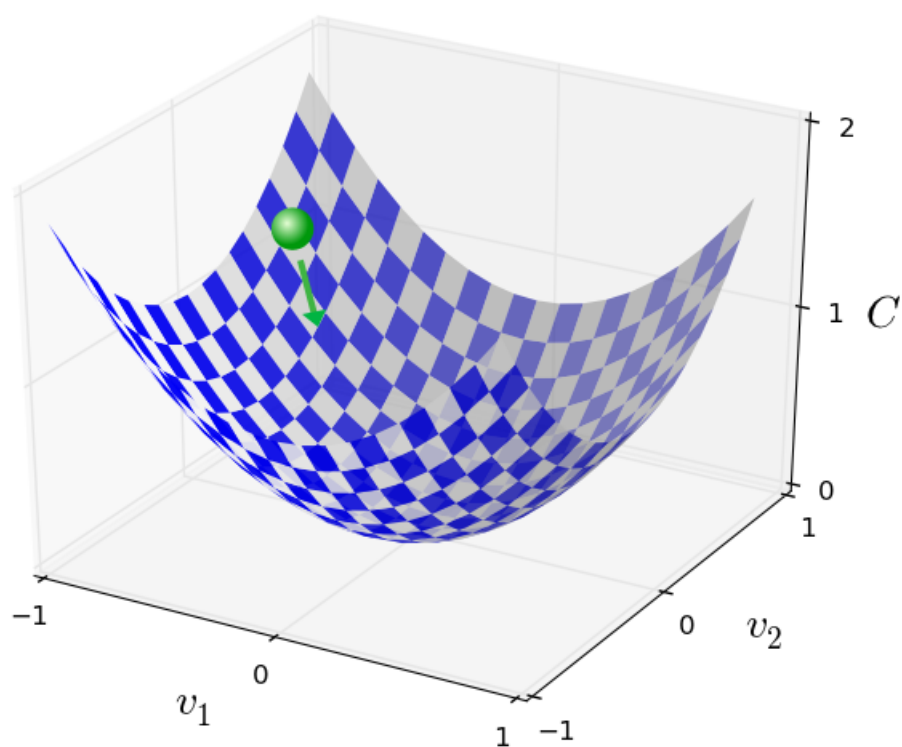
$$v \rightarrow v' = v - \eta \nabla C$$

След това отново ще се използва това правило за актуализиране, за да се направи

още един ход. Ако продължи да се случва това, отново и отново, ще продължи

намаляването на C , докато - с късмет - не се достигне глобален минимум.

Обобщавайки, начинът, по който работи алгоритъмът за градиентно спускане, е многократно да се изчисли градиента ∇C и след това да се движи в обратна посока, „падайки надолу“ по склона на долината. Можем да се визуализира така:



Ясно е градиентното спускане, когато C е функция само на две променливи. Но всъщност всичко работи еднакво добре, дори когато C е функция на много повече променливи.

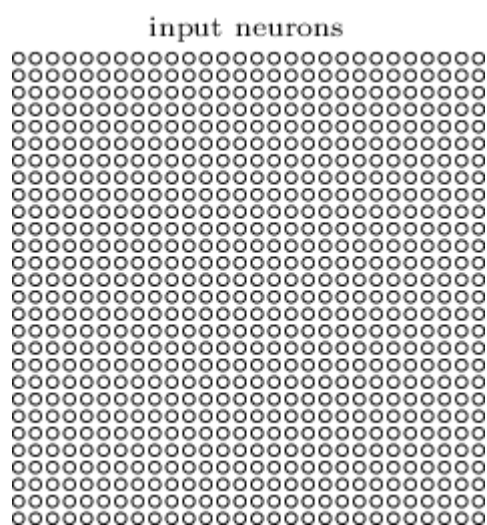
1.2.6 Конволюционни мрежи

Тези мрежи използват специална архитектура, която е особено добре адаптирана за класифициране на изображения. Използването на тази архитектура прави конволюционните мрежи бързи за обучение. Това от своя страна помага с обучаването на дълбоки, многослойни мрежи, които са много добри в класифицирането на изображения. Днес дълбоки конволюционни мрежи

или някакъв близък вариант се използват в повечето невронни мрежи за разпознаване на изображения.

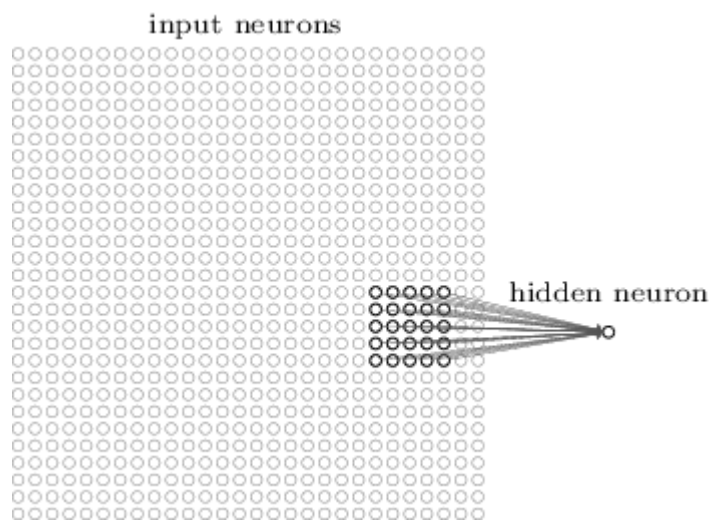
Конволюционните невронни мрежи използват три основни идеи: локални рецептивни полета (local receptive fields), споделени тегла (shared weights) и обединяване (pooling). Ще бъдат разгледани всяка една от тези идеи на свой ред.

Локални рецептивни полета: В напълно свързаните слоеве, показани по-рано, входовете бяха изобразени като вертикална линия от неврони. В конволюционна мрежа е по - удобно да се разгледажат входовете като $n \times m$ правоъгълник от неврони, чиито стойности съответстват на $n \times m$ пиксела, който се използва като входни данни:



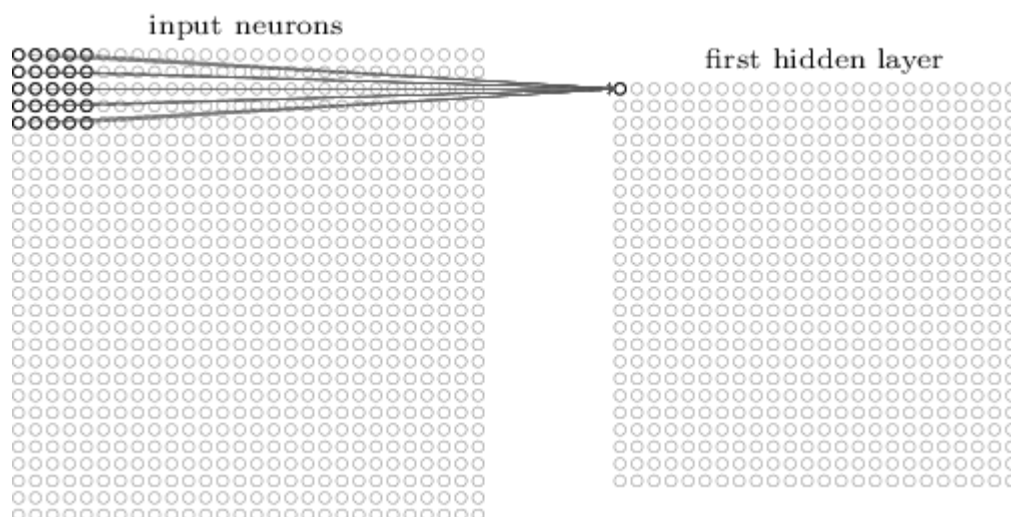
Както обикновено, ще се свържат входните пиксели към слой от скрити неврони. Но няма да се свързва всеки входен пиксел към всеки скрит неврон. Вместо това, ще се правят връзки само в малки, локализирани области на входното изображение. По - точно, всеки неврон в първия скрит слой ще бъде свързан с малка област от входните неврони, например, област 5×5 , съответстваща на 25

входни пиксела. Така че, за конкретен скрит неврон, може да има връзки, които изглеждат така:

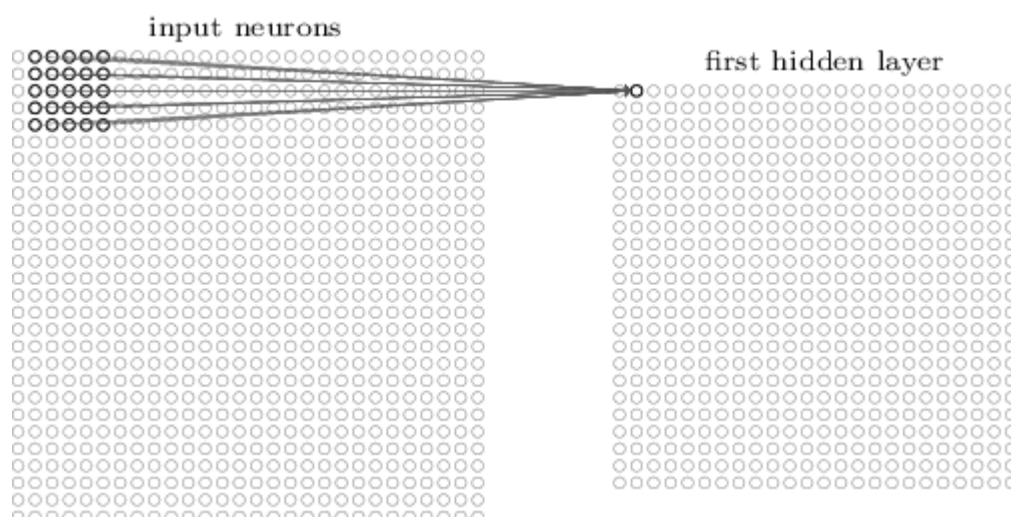


Тази област във входното изображение се нарича локално рецептивно поле за скрития неврон. Това е малък прозорец на входните пиксели. Всяка връзка научава тегло. И скритият неврон научава и цялостно пристрастие. Може да се мисли за този конкретен скрит неврон като учещ се да анализира своето конкретно локално рецептивно поле.

След това се плъзга локалното рецептивно поле през цялото входно изображение. За всяко локално рецептивно поле има различен скрит неврон в първия скрит слой. За да се илюстрира това конкретно, може да се започне с локално приемливо поле в горния ляв ъгъл:



След това се плъзга локалното рецептивно поле с един пиксел надясно (т.е. с един неврон), за да се свърже с втори скрит неврон:



И така нататък, изгражда се първия скрит слой.

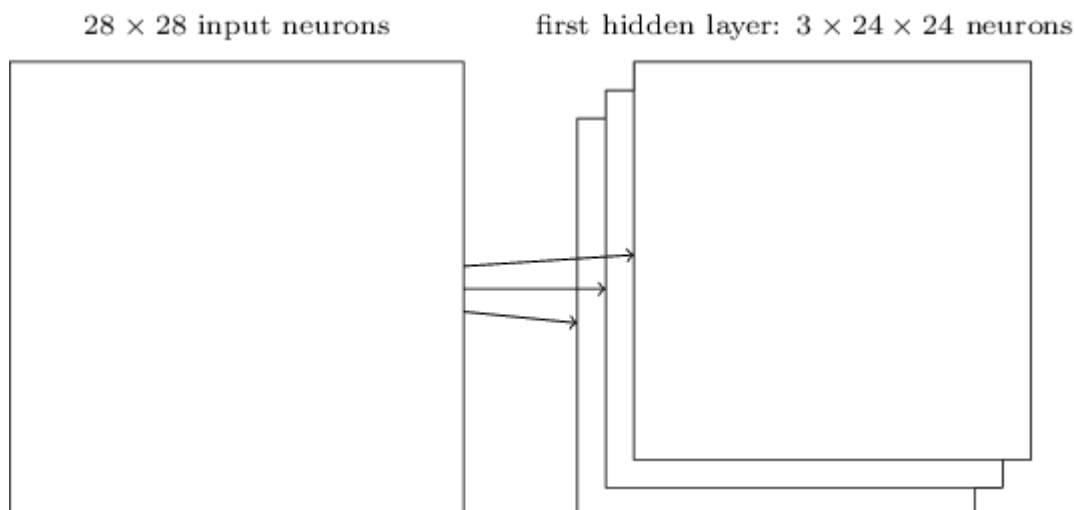
Понякога се използва различна дължина на крачката (stride length).

Например, може да се премести локалното приемливо поле с 2 пиксела надясно (или надолу), в който случай би могло да се каже, че се използва дължина на крачката 2. Този текст се фокусира предимно към дължината на крачката 1, но си струва се знае, че хората понякога експериментират с различни дължини на крачката.

Споделени тегла и отклонения: Ясно е, че всеки скрит неврон има отклонение и 5×5 тегла, свързани с неговото локално рецептивно поле. Това, което досега не беше ясно, е, че ще се използват еднакви тегла и отклонения за всеки от 24×24 скрити неврони. Това означава, че всички неврони в първия скрит слой откриват точно една и съща характеристика. За да стане ясно защо това има смисъл, нека се предположи, че теглата и отклоненията са такива, че скритият неврон може да избере, например, вертикален ръб в определено локално рецептивно поле. Тази способност вероятно ще бъде полезна и на други места в изображението. И затова е полезно да се приложи един и същ детектор на функции навсякъде в изображението. Казано с малко по-абстрактни термини, конволюционните мрежи са добре адаптирани към транслационната инвариантност на изображенията: премества се снимка на котка (като пример) малко и тя все още е изображение на котка.

Поради тази причина понякога се нарича картата от входния слой към скрития слой карта на характеристиките (feature map). Наричат се още теглата, определящи картата на характеристиките, споделени тегла (shared weights). И отклонението се нарича, дефиниращо картата на характеристиките по този начин, споделено отклонение (shared weight). Често се казва, че споделените тегла и пристрастия определят ядро (kernel) или филтър (filter). В литературата хората понякога използват тези термини по малко по-различен начин.

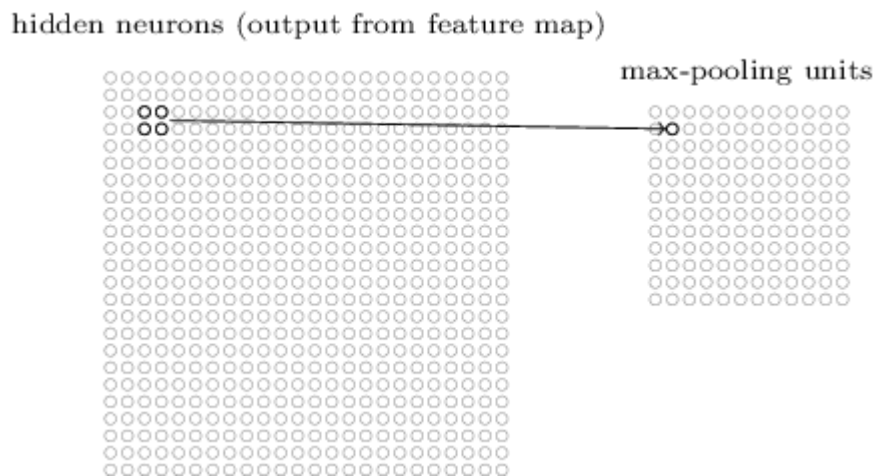
Пълният конволюционен слой се състои от няколко различни карти на характеристиките:



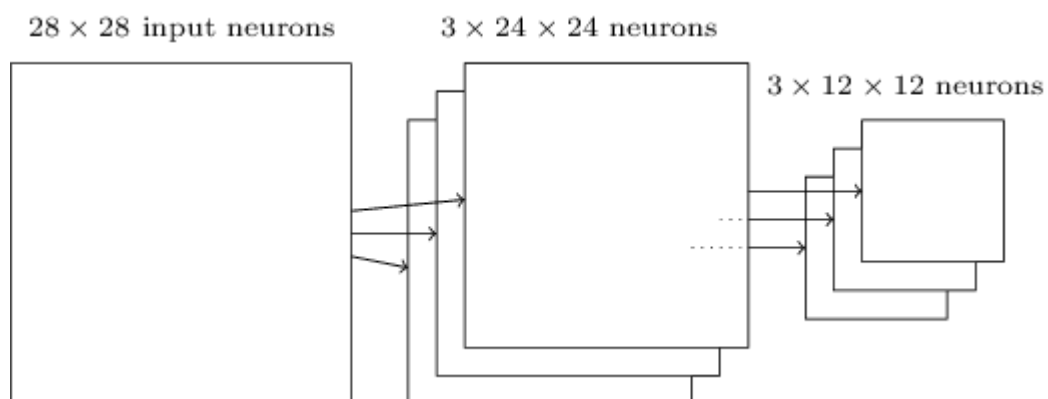
В показания пример има 3 характеристики на карти. Всяка карта на характеристиките се дефинира от набор от 5×5 споделени тегла и едно споделено отклонение. Резултатът е, че мрежата може да открие 3 различни вида характеристики, като всяка може да бъде открита в цялото изображение. Показани са само 3 карти на характеристиките, за да е проста диаграмата по-горе. Въпреки това, на практика конволюционните мрежи могат да използват повече (и може би много повече) карти на характеристиките.

Обедняващи слоеве: В допълнение към току-що описаните конволюционни слоеве, конволюционните невронни мрежи също съдържат обедняващи слоеве. Обедняващите слоеве обикновено се използват веднага след конволюционните слоеве. Това, което правят, е да опростят информацията в изхода от конволюционния слой. По - точно, обедняващият слой взема всяка карта на характеристиките и създава кондензирана карта. Например, всяка единица в обединителния слой може да обобщава регион от (например 2×2) неврони в предишния слой. Като конкретен пример, една обща процедура за

обединяване е известна като max-pooling. При max-pooling единицата за обединение просто извежда максималното активиране във входната област 2×2 , както е показано на следната диаграма:

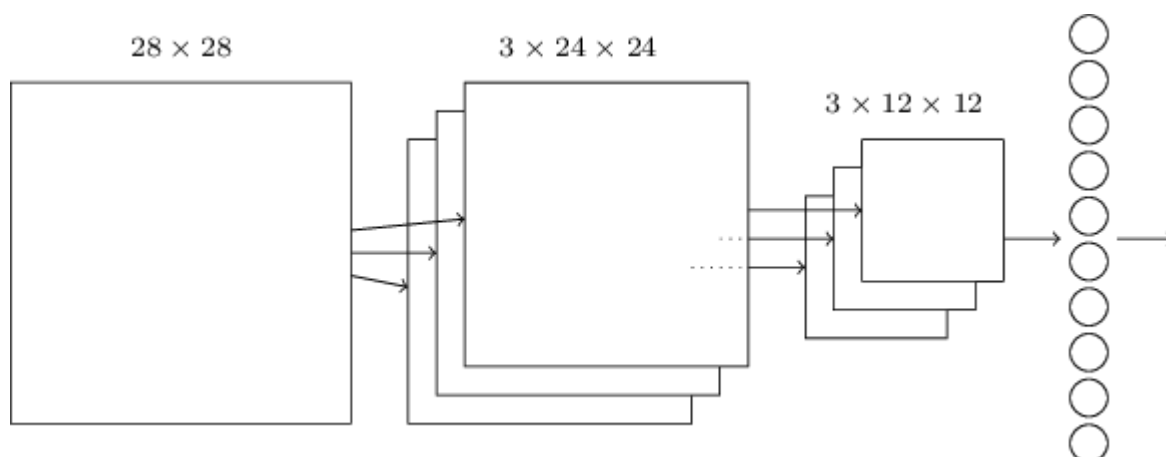


Както бе споменато по-горе, конволюционният слой обикновено включва повече от една карта на характеристиките. Прилага се максимално обединяване към всяка карта на характеристиките поотделно. Така че, ако има три карти на характеристиките, комбинираните конволюционни и max-pooling слоеве биха изглеждали така:



Може да се мисли за max-pooling като начин мрежата да попита дали дадена характеристика се намира някъде в регион на изображението.

Вече може да се обединят всички тези идеи, за да се образува цялостна конволюционна невронна мрежа. Изглежда подобно на архитектурата, която току-що бе разгледана, но има добавяне на слой от изходни неврони:



1.3 Технологии и практики в разработването на софтуер, занимаващ се с изображения

Всичко, разгледано до момента, може да се имплементира, използвайки всякакви технологии и платформи. До момента има голямо количество библиотеки (libraries), фреймуърци (frameworks) и среди, които целят да помогнат при решаването на проблеми, свързани със обработка на снимки, невронни мрежи и статистически задачи като цяло. Най - разпространеният програмен език в сферата е Python. Практиката показва, че скриптовите езици (като Python, Bash и т.н.) са по - лесни за използване от хора, които не са програмисти. Това е поради причината, че скриптовите езици боравят с програми, те управляват входа и изхода на вече написани програми.

Предимството и слабостта им е липсата на контрол, който те могат да упражняват върху компютърната система. Масово с Python се използват модулите OpenCV, Pillow и Numpy при обработка на снимки. Изкуственият интелект се изгражда практически винаги върху две платформи - Tensorflow и Pytorch. Те позволяват сравнително лесна изработка на скриптове за контрол на невронни мрежи, но отнемат голямо количество контрол от потребителя. Друга среда за разработка на такива приложения е MATLAB. Там са вградени няколко възможности за програмни езици, включително езикът, разработен заедно с платформата.

Втора Глава

Практическа част

2.1 Проблемът

Проблемът, върху който се фокусира проектът, е разпознаването на замърсявания на контейнери (щайги) от снимки. Ocado Technologies се опитват да автоматизират склад за хранителни продукти. Част от процеса е разпознаването на замърсени контейнери и тяхното почистване (неприемливо е да се пълнят с хранителна стока, докато са мръсни). Дипломната работа цели да служи като прототипна версия на тази задача, като удовлетворява функционалните изисквания, които предварително са зададени. Пример за чист контейнер:



Пример за мръсен контейнер:



Снимката бива уловена от камера, прикачена към контролер, който на свой ред изпраща информацията от направената снимка на машина, която има капацитета да извърши обработки и анализ върху снимката и да установи дали щайгата е замърсена.

Недопустима е грешната класификация на мръсен контейнер, тъй като не е редно да се поставят хранителни продукти в замърсено пространство. По - малката грешка е неправилното класифициране на чист контейнер, но все пак е нежелателно, поради причината, че измиването на контейнерите коства разходи.

2.2 Решението

Имайки предвид характера на проблема, единственият начин за изграждане на подобаващо решение е съставяне на комуникация между клиент (Платката, която праща снимките) и сървър (машината, която ги обработва). След получаването на снимката, сървърът извършва необходимите обработки върху нея и прогнозира дали щайгата е замърсена. Това, от своя страна, става чрез

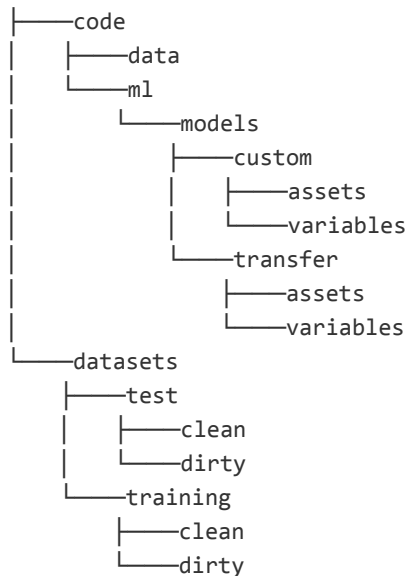
подаване на снимката на трениран модел (невронна мрежа), изходът на който определя прогнозата.

Разбира се, предварително трябва да бъде трениран такъв модел. Това обаче предполага наличието на входни данни, върху които да бъде трениран. А в зависимост от избраните обработки върху снимката, нужно е да се извършат същите обработки и върху входните данни.

В този проект са разработени и сравнени три системи за обработка на снимки, следващи този модел на функционалност, при които единствената разлика е начинът и степента на обработка върху снимките, след което е избрано най - ефективната и е внедрена в тестовата среда (клиент - сървър архитектурата). И в трите системи снимката бива преоразмерена в 224x224, поради причините, че оригиналните размери са прекалено големи за бърза обработка и за вход на един от моделите (тренираните инстанции на невронни мрежи). За всяка система се тренират по два модела - един , който е ръчно изграден, и един, чиято архитектура имитира успешна архитектура изградена от Google и е напаснат към конкретната задача (transfer learning). Това означава, че архитектурите на моделите си остават същите за всички системи, само теглата и отклоненията се променят в процес на трениране. И двете архитектури са изградени на база проба и грешка, тъй като няма открит конкретен метод за създаване на такива архитектури. Разликата е, че Google са вложили много повече време и ресурси в своята мрежа. Заетата мрежа се нарича MobileNetV2.

Тя е избрана, поради причината, че постига много добри резултати и изисква сравнително ниско количество ресурси от машината.

Поради приликите в естеството на системите, те имат абсолютно идентична файлова структура:



2.2.1 Система 1 - без обработка

Тук целта е моделите да се тренират на пълната снимка след преоразмеряването - няма загуба на информация от обработки върху снимката. Това предполага най - висока скорост от системите.

2.2.2 Система 2 - изолиране на контейнер

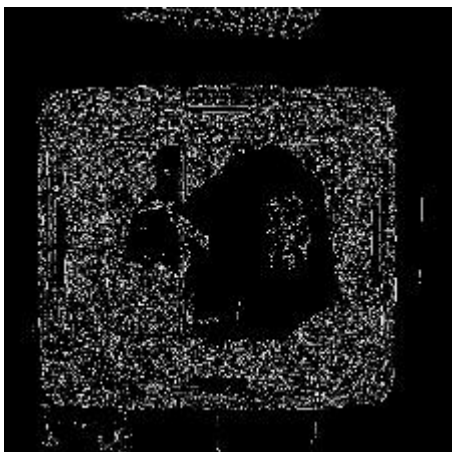
Същото като система 1, само че тук се добавя обработка на снимките с маски, за да се изолира щайгата от останалата част от снимката (има малко загуби от самата щайга). Целта е да се даде на невронната мрежа само

информацията която и е нужна. С други думи е опит за ръчна оптимизация. Ето как изглеждат снимки на чиста и мръсна щайга съответно:



2.2.3 Система 3 - изолиране на ръбове

Тази система включва всичко от система 2, но добавя и засичане на ръбове, като добавя редукция на шума в снимката и засичане на ръбове, всичко с функционалности на готови библиотеки:



2.2.4 Входните данни

Входните данни се разделят на два вида - тренировъчни (training) и тестови (test). Когато една невронна мрежа се тренира, е важно да има (поне приблизително) еднакъв брой входни данни от всеки клас (в този случай са снимки на чисти (clean) и мръсни (dirty) контейнери). Причината за това е, че ако това условие не е покрито, е много вероятно мрежата да започне да се фокусира прекалено много в детайлите на преобладаващия клас от данни, което може да доведе до огромни грешки при класифицирането на данни от другите класове. Този проблем се нарича свръх напасване (overfitting). Входните данни в training/директориите са изравнени като бройка с цел избягване точно на този проблем. В

работната среда обаче рядко се случва мрежата да се налага да класифицира по еднакъв брой данни от всеки клас (в конкретния случай мрежата ще се налага да класифицира много повече чисти, отколкото мръсни контейнери - пропорцията е приблизително 13 : 1). Преди да се вкара в действие в работна среда, редно е една невронна мрежа да бъде изтествана и оценена с входни данни, наподобяващи тези в работната среда - това дава реалистична представа за ефективността на мрежата. Входните данни в test/ директориите са акумулирани в горепосочената пропорция (13:1) и с тях ще бъдат изтествани моделите.

2.2.5 Архитектури на невронните мрежи

Изградени са две архитектури на невронни мрежи.

Първата е ръчно изработена. Тя започва с входен слой (224x224x3) и уголемяващ слой (augmentation layer), който прави случайни завъртания (random rotations), случайни наклонения (random flips) и случайни оразмерявания (random zooms) върху входните данни. Това гарантира, че моделът ще може да разпознава правилните характеристики на снимки след приключване на обучението си. Следва съвкупност от конволюционни слоеве и обединяващи слоеве. След тях е поставен отпадащ слой. Това е маска, която анулира приноса на някои неврони към следващия слой и оставя непроменени всички други. Ако този слой не присъства, първите входни данни ще влияят върху ученето по непропорционално висок начин - това от своя страна би попречило на изучаването на характеристики, които се появяват само в по-късни данни. Предпоследният слой

се нарича изравняващ слой. Той превръща данните от миналия слой в едноизмерен вектор, защото не е възможно квадратни или кубични форми да се свържат с плътен слой (най - простият слой, разглеждан досега - слой от неврони, в който всеки неврон получава вход от всички неврони на предишния слой). Последният слой представлява сигмоиден неврон - изходът на мрежата. Неговата стойност е разбираемо представяне на класификацията на модела.

Втората архитектура имитира успешна архитектура (MobileNetV2), изградена от Google, и я напасва към конкретния проблем (разпознаване на замърсявания в контейнери). Този процес се нарича трансферно обучение (transfer learning). Първото, което е нужно да се направи е представянето на входните данни във вид, приемлив за изградения модел. Напасването към конкретния проблем представлява претрениране на някои от слоевете на имитирания модел (в текущия случай са от 125-ти слой до последния - числото е избрано на база препоръки от екипът на Google). Първоначално, моделът е “замразен” със собствените си предефинирани параметри. За да се претренират слоеве, трябва първо да се “отмразят” - трябва да се конфигурира мрежата, така че потребителят да има възможността да променя параметрите. След претрениране на мрежата, са добавени слоеве, които обработват изхода на имитирания модел (7x7 снимка) по начин, който е удобен за класификация в текущия случай. Тъй като изходът от базовия модел (имитирания модел) е в 2 измерения, обединяващият слой след него дава добра репрезентация на данните и може директно да се обвърже към сигмоиден неврон. Последният слой е

сигмоиден неврон, който показва класификацията както при първата архитектура.

2.2.6 Изводи

Входните данни се подават на невронните мрежи под формата на партии (batches): групи от снимки - в този случай по 32 на група. Това се случва поради две причини: по - ефективно кеширане на данните в оперативната памет (намаляването на входно - изходните операции) и възможността да се дават измервания за производителността и ефективността на трениращият се модел. Такива измервания се правят след всяко обработване на партида - това се нарича епоха (epoch). Следните метрики са следени по време на трениране на моделите: точност (accuracy), загуба (loss), фалшиви положителни (false positives), фалшиви отрицателни (false negatives), истински положителни (true positives) и истински отрицателни (true negatives). Всички останали метрики могат да се извлекат от посочените.

Точност - показва колко процента от класификациите са коректни

Загуба - показва колко влиятелни са грешките от класификациите спрямо верните резултати

Фалшиви положителни - брой на грешно класифицираните положителни екземпляри (в случая са мръсни щайги)

Фалшиви отрицателни - брой на грешно класифицираните отрицателни екземпляри (в случая са чисти щайги)

Истински положителни - брой на правилно класифицираните

положителни екземпляри

Истински отрицателни - брой на правилно класифицираните

отрицателни екземпляри

Всички невронни мрежи имат определен брой епохи на обучение преди да започнат да извършват свръх напасване. За щастие, измерванията, които се случват след всяка епоха могат да помогнат при минимизирането на този проблем. В случая се следи загубата, защото тя винаги става по - малка при обучение и винаги става по - голяма при свръх напасване. Зададен е толеранс от 5 епохи, в които грешката се увеличава, преди обучението да приключи. Това число дава оптимален резултат - това е установено от множество опити с различни стойности.

На база горните метрики се оценява избраният модел за клиент - сървърната архитектура. Всички модели се тестват върху собствени тестови входни данни, като за всички системи са трениране по 2 модела - за всяка архитектура:

Система 1 - Архитектура 1

True Positives: 385	False Negatives: 27
False Positives: 198	True Negatives: 1600

Система 1 - Архитектура 2

True Positives: 402	False Negatives: 33
False Positives: 292	True Negatives: 1686

Система 2 - Архитектура 1

True Positives: 379	False Negatives: 10
False Positives: 152	True Negatives: 1546

Система 2 - Архитектура 2

True Positives: 403	False Negatives: 9
False Positives: 375	True Negatives: 1463

Система 3 - Архитектура 1

True Positives: 341	False Negatives: 71
False Positives: 150	True Negatives: 1688

Система 3 - Архитектура 2

True Positives: 366	False Negatives: 46
False Positives: 315	True Negatives: 1523

На база горепосочената информация, могат да се съревновават само “Система 2 - Архитектура 1” и “Система 3 - Архитектура 1”. Наблюдавайки таблиците, може да се забележи, че разликата между фалшивите положителни е само 2 в полза на система 3, докато разликата между фалшивите отрицателни е цели 62. Поради тази причина е разумно да се избере “Система 2 - Архитектура 1” като подобаващ модел за клиент - съвърнатата архитектура.

2.2.7 Тестовата среда: клиент - сървърната архитектура

Комуникацията е осъществена чрез скриптове, които позволяват на клиент и сървър да комуникират през сокети. Очевидна е необходимостта на физическа връзка между мрежовите карти на двете устройства (независимо дали е интернет или интранет). Избраният мрежови протокол е TCP, защото е от критична важност данните да пристигат в целостта си до сървъра. Клиентът изпраща снимки на сървърът, след което сървърът ги класифицира и изпечатва резултата на стандартния изход (stdout) - в случая е терминал. Сървърът е отворен на порт 1024.

Ето резултатите от направени класификации на сървъра след получаване на 3 снимки от клиента и обработка:

Снимка 1 - чист контейнер

```
tf.Tensor([[0.06666712]], shape=(1, 1), dtype=float32)
```

Снимка 2 - мръсен контейнер

```
tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
```

Снимка 3 - мръсен контейнер

```
tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
```

Резултатите от класификацията са много задоволителни.

Трета глава

Програмна реализация на система за засичане на замърсявания в контейнери

Целият проект се съдържа в гитхъб хранилище на следния адрес:

<https://github.com/stefan-sotirov-elsys/Thesis/>. Има ограничение на количеството

памет, която може да се заема в едно хранилище - 2 гигабайта. Поради тази

причина входните данни не са качени, даден е само по един пример. В

Readme.md файла е оказано какво се прави в случай, че има нужда от тях.

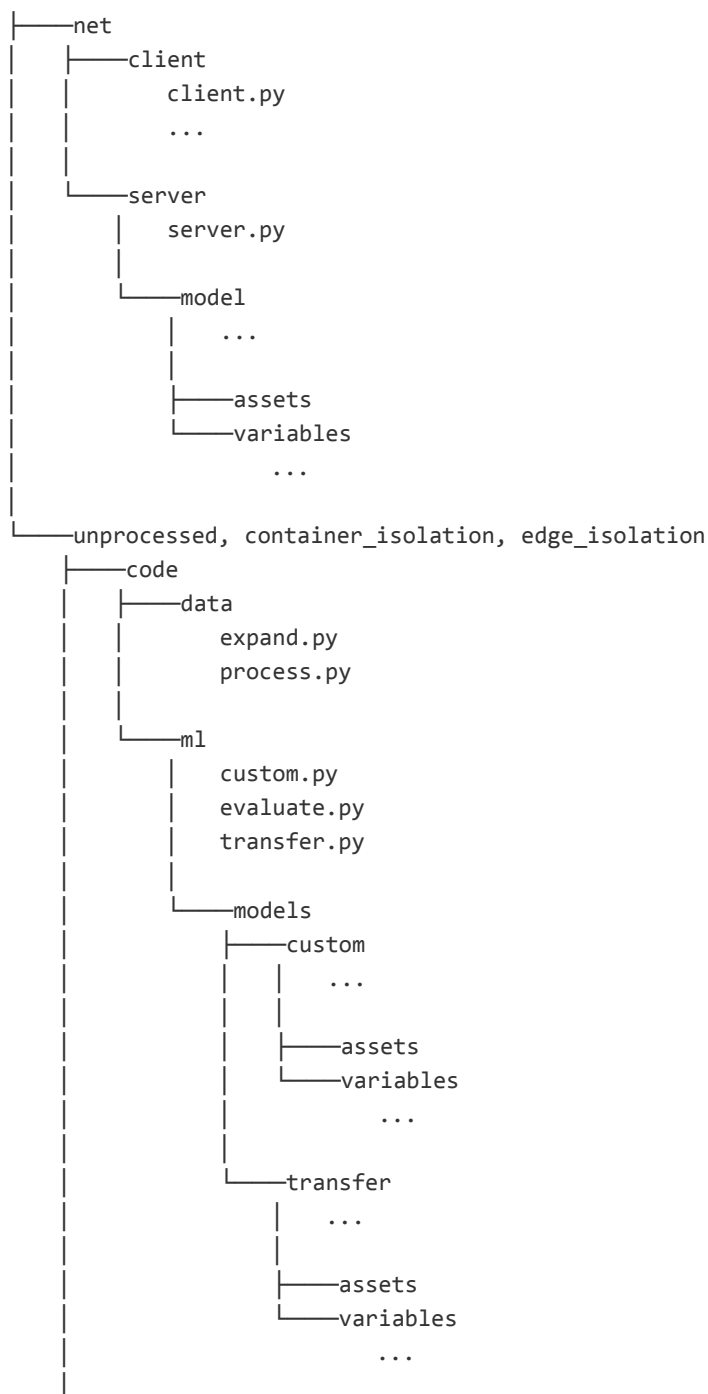
3.1 Използвани технологии

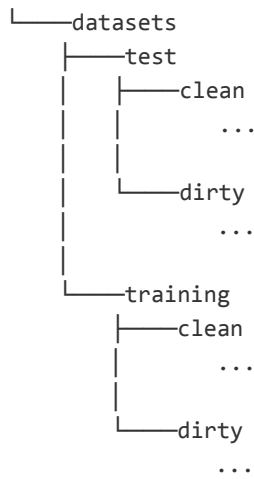
Проектът е изграден изцяло с употребата на скриптовия език Python.

Използвани са външните модули OpenCV, NumPy, TensorFlow. OpenCV е един от най - разпространените модули за обработка на снимки, който предоставя много алгоритми за улеснение на потребителя. TensorFlow е най - често използваният модул за разработка на невронни мрежи, предоставя широк спектър от възможности и е изключително добре документиран. Причината за избора на тези конкретни технологии лежи във факта, че проектът цели максимално да се

доближи до реалните стандарти на индустрията. NumPy носи изчислителната мощ на езици като C и Fortran, като я внедрява в Python. Модулът добавя поддръжка за големи, многомерни масиви и матрици, заедно с голяма колекция от математически функции за работа с тези масиви.

3.2 Въведение в структурата на проекта





Всичко, означено с триточие са или снимки, или конфигурационни файлове, генерирани от TensorFlow модула по време на трениране.

3.2.1 net/

Това е имплементацията на клиент - сървър архитектурата. В client/ се намира клиентският скрипт, а в server/ се намират сървърният скрипт и компилираният модел на невронна мрежа.

3.2.2 unprocessed/, container_isolation/, edge_islation/

Това са имплементациите на трите системи. Поддиректориите се разделят на code/ и datasets/. code/ съдържа кода и моделите, а datasets/ съдържа входните данни. code/ от своя страна съдържа data/ и ml/, където в data/ е кодът, който обработва входните данни, а в ml/ е този, който създава и тренира моделите, и метаданните на моделите. В datasets/ се намират двете групи входни данни под формата на директории - training/ и test/. Тези директории от своя страна

съдържат входните данни, които са разделени на класове - clean и dirty, също под формата на директории. В тях се съдържат данните - снимките на контейнери.

3.3 Обработка на входните данни

3.3.1 process.py

Целта на скрипта е да обработи входните данни преди да ги изравни - причината за тази последователност е съображението, че обработката е много по - бавна от копирането. Степента на обработка е различна при трите системи, всичко останало е идентично:

```
import os
import cv2

directories_paths = ["../datasets/training/clean/", "../datasets/training/dirty/"]

directories_count = 2

directories_contents = []

directories_sizes = []

total = 0

for dir_path in directories_paths:

    directories_contents.append(os.listdir(dir_path))

    directories_sizes.append(len(directories_contents[-1]))

    total += directories_sizes[-1]

width = 224

height = 224

processed = 1
```

При първата система единственото обработване е преоразмеряване:

```
for i in range(0, directories_count):
```

```

for file_name in directories_contents[i]:

    img_path = directories_paths[i] + file_name

    print(img_path) # debug

    img = cv2.imread(img_path)

    if img.shape[0] == width:

        # the image has been processed

        print(str(processed) + " / " + str(total)) # debug

        processed += 1

        continue

    # resize and save

    img = cv2.resize(img, (width, height))

    cv2.imwrite(img_path, img)

    print(str(processed) + " / " + str(total)) # debug

    processed += 1

```

Във втората система има и изолация на контейнерите. Първо пикселите на снимка се превеждат в HSV (Hue, Saturation, Value) формат. Това позволява изолирането на пикселите на база техния цветен оттенък, насищането на светлината върху тях и яркостта на оттенъка. За всеки канал на HSV формата се прави отделна маска, след което трите маски се умножават (извършва се AND операция върху съответстващите пиксели) - по този начин маските се комбинират. Ето как изглежда програмната реализация:

```

for i in range(0, directories_count):

    for file_name in directories_contents[i]:

        img_path = directories_paths[i] + file_name

        print(img_path) # debug

```

```

img = cv2.imread(img_path)

if img.shape[0] == width:

    # the image has been processed

    print(str(processed) + " / " + str(total)) # debug

    processed += 1

    continue

# isolate the container

img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hue = (img_hsv[:, :, 0] < 100) * (img_hsv[:, :, 0] > 3)

saturation = (img_hsv[:, :, 1] > 70)

value = (img_hsv[:, :, 2] > 120)

mask = saturation * value * hue

img_hsv[:, :, 0] *= mask

img_hsv[:, :, 1] *= mask

img_hsv[:, :, 2] *= mask

img = cv2.cvtColor(img_hsv, cv2.COLOR_HSV2BGR)

# resize and save

img = cv2.resize(img, (width, height))

cv2.imwrite(img_path, img)

print(str(processed) + " / " + str(total)) # debug

processed += 1

```

Третата система включва горните обработки, но в нея се намира и изолирането на ръбовете на контейнерите. Това е реализирано, като преди изолирането на контейнера, се премахва количество шум от снимката - това може да се постигне по много начини, в случая е избран този, който е вграден в OpenCV модула. След изолирането на контейнерите, чрез друг вграден метод на

OpenCV, се изолират и ръбовете. Този алгоритъм (алгоритъм на Canny) намира контури на база интензитет на пикселите (затова беше важно да се премахне шумът). Ето как изглежда кодът:

```
import cv2
import os
import numpy

directories_paths = ["../..../datasets/training/clean/", "../..../datasets/training/dirty/"]

directories_count = 2

directories_contents = []

directories_sizes = []

total = 0

for dir_path in directories_paths:

    directories_contents.append(os.listdir(dir_path))

    directories_sizes.append(len(directories_contents[-1]))

    total += directories_sizes[-1]

width = 224

height = 224

processed = 1

for i in range(0, directories_count):

    for file_name in directories_contents[i]:

        img_path = directories_paths[i] + file_name

        print(img_path) # debug

        # isolate the container

        img = cv2.imread(img_path)

        if img.shape[0] == width:

            # the image has been processed

            print(str(processed) + " / " + str(total)) # debug

            processed += 1

            continue
```

```

img = cv2.fastNlMeansDenoisingColored(img, None, 10, 10, 7, 21)

img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hue = (img_hsv[:, :, 0] < 100) * (img_hsv[:, :, 0] > 3)

saturation = (img_hsv[:, :, 1] > 70)

value = (img_hsv[:, :, 2] > 120)

mask = saturation * value * hue

img_hsv[:, :, 0] *= mask

img_hsv[:, :, 1] *= mask

img_hsv[:, :, 2] *= mask

img = cv2.cvtColor(img_hsv, cv2.COLOR_HSV2BGR)

# isolate the edges

img = cv2.GaussianBlur(cv2.cvtColor(img, cv2.COLOR_RGB2GRAY), (3, 3), 0)

median = numpy.median(img)

lower = int(max(0, 0.66 * median))

upper = int(min(255, 1.33 * median))

img = cv2.Canny(image = img, threshold1 = lower, threshold2 = upper)

# resize and save

img = cv2.resize(img, (width, height))

cv2.imwrite(img_path, img)

print(str(processed) + " / " + str(total)) # debug

processed += 1

```

3.3.2 expand.py

Скриптът е идентичен и в трите системи. Целта му е да изравни пропорцията на входните данни - еднакъв брой снимки с мръсни и чисти щайки. Това гарантира, че невронните мрежи няма да тренират прекалено много с един

тип данни, което би довело до лошо научаване. Това се постига чрез копиране на вече съществуващите снимки, докато се изравни бройката.

```
import os
import shutil

directories_count = 2

directories_paths = ["../..../datasets/training/clean/", "../..../datasets/training/dirty/"]

directories_contents = []

directories_sizes = []

largest = 0

for path in directories_paths:

    directories_contents.append(os.listdir(path))

    print(path) # debug

    directories_sizes.append(len(directories_contents[-1]))

    print(directories_sizes[-1]) # debug

    if directories_sizes[-1] > largest:

        largest = directories_sizes[-1]

print(largest) # debug

for i in range(0, directories_count):

    size_difference = largest - directories_sizes[i]

    j = 0

    k = 0

    substring = "_"

    while size_difference > 0:

        if j == directories_sizes[i]:

            j = 0

            k += 1

            for l in range(0, k):
```

```
        substring += " "

        shutil.copy(directories_paths[i] + directories_contents[i][j], directories_paths[i] +
substring + directories_contents[i][j])

        print(directories_paths[i] + directories_contents[i][j]) # debug

        print(size_difference)

        j += 1

        size_difference -= 1
```

3.4 Машинно самообучение

Скриптовете са идентични и в трите системи

3.4.1 custom.py

Скриптът е реализация на персонализирания модел на невронна мрежа. В него входните данни се разделят на групи - тренировъчни, валидационни и тестови, всяка от която служи по различен начин. Тренировъчният се използва по време на тренирането. Валидационният и тестовият имат една и съща функционалност, но се използват в различни етапи от обучението. Валидационният се използва, докато мрежата се учи, и цели да даде приблизителна преценка на модела по време на обучение. Тестовият прави същото, но след като мрежата приключи обучението си. След това се извършват две конфигурации, които правят обучението по - бързо. Едната кешира снимки в RAM, за да намали I/O латенцията, а другата извлича снимки по време на обработка на други снимки. Моделът се компилира, като му се подават метрики

и оптимизатор (начин за намиране на функцията на разходите (loss function)).

Последните стъпки са тренирането, оценяването и запазването на модела.

Скриптът изглежда по следния начин:

```
import tensorflow as tf

# prepare the dataset

data_dir = "../..../datasets/test/"

width = 224

height = 224

batch_size = 32

training_dataset = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    label_mode = "binary",
    class_names = ["clean", "dirty"],
    validation_split = 0.2,
    subset = "validation",
    seed = 123,
    image_size = (width, height),
    batch_size = batch_size
)

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    label_mode = "binary",
    class_names = ["clean", "dirty"],
    validation_split = 0.2,
    subset = "validation",
    seed = 123,
    image_size = (width, height),
    batch_size = batch_size
)

test_dataset = validation_dataset.take(
    tf.data.experimental.cardinality(validation_dataset)
)

class_names = training_dataset.class_names

# print(class_names) # debug

# optimise the dataset

training_dataset = training_dataset.cache().shuffle(1000).prefetch(buffer_size =
tf.data.AUTOTUNE)

validation_dataset = validation_dataset.cache().prefetch(buffer_size = tf.data.AUTOTUNE)

test_dataset = test_dataset.prefetch(buffer_size = tf.data.AUTOTUNE)
```



```

data_augmentation = tf.keras.models.Sequential([
    tf.keras.layers.RandomFlip("horizontal",
                               input_shape = (width, height, 3)),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
])

# create the model

model = tf.keras.models.Sequential([
    data_augmentation,
    tf.keras.layers.experimental.preprocessing.Rescaling(1 / 255),
    tf.keras.layers.Conv2D(16, 3, padding = "same", activation = "leaky_relu"),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(32, 3, padding = "same", activation = "leaky_relu"),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(64, 3, padding = "same", activation = "leaky_relu"),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    # tf.keras.layers.Dense(128, activation = 'leaky_relu'),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])

early_stop_callback = callback = tf.keras.callbacks.EarlyStopping(monitor = "loss", patience =
3)

# compile the model

model.compile(
    optimizer = "adam",
    loss = tf.keras.losses.BinaryCrossentropy(from_logits = False),
    metrics = [
        tf.keras.metrics.FalseNegatives(),
        tf.keras.metrics.FalsePositives(),
        tf.keras.metrics.TrueNegatives(),
        tf.keras.metrics.TruePositives(),
        "accuracy"
    ]
)

# model.summary() # debug

# train the model

history = model.fit(
    training_dataset,
    validation_data = validation_dataset,
    epochs = 100000,
    callbacks = [early_stop_callback]
)

metrics = model.evaluate(test_dataset)

model.save("models/custom")

```

3.4.2 transfer.py

Концепциите и техниките, приложени тук са обяснени в миналата глава.

Много от кода се повтаря с предишния, затова тук са илюстрирани различните части:

```
import tensorflow as tf

# prepare the dataset

data_dir = "../../datasets/test/"

width = 224

height = 224

batch_size = 32

training_dataset = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    label_mode = "binary",
    class_names = ["clean", "dirty"],
    validation_split = 0.2,
    subset = "validation",
    seed = 123,
    image_size = (width, height),
    batch_size = batch_size
)

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    label_mode = "binary",
    class_names = ["clean", "dirty"],
    validation_split = 0.2,
    subset = "validation",
    seed = 123,
    image_size = (width, height),
    batch_size = batch_size
)

test_dataset = validation_dataset.take(
    tf.data.experimental.cardinality(validation_dataset)
)

class_names = training_dataset.class_names

# optimise the dataset

training_dataset = training_dataset.cache().shuffle(1000).prefetch(buffer_size =
tf.data.AUTOTUNE)
```

```

validation_dataset = validation_dataset.cache().prefetch(buffer_size = tf.data.AUTOTUNE)

test_dataset = test_dataset.prefetch(buffer_size = tf.data.AUTOTUNE)

# create the base model

base_model = tf.keras.applications.MobileNetV2(
    input_shape = (width, height, 3),
    include_top = False,
    weights = "imagenet"
)

# freeze the base model / unfreeze it

base_model.trainable = True

# Fine-tune from this layer onwards

fine_tune_start = 125

# Freeze all the layers before the fine_tune_start layer

for layer in base_model.layers[:fine_tune_start]:

    layer.trainable = False

print(len(base_model.layers))

# add the classification head

data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.2),
    tf.keras.layers.RandomZoom(0.1),
])

global_avg_layer = tf.keras.layers.GlobalAveragePooling2D()

prediction_layer = tf.keras.layers.Dense(1, activation = "sigmoid")

input = tf.keras.Input(shape = (width, height, 3))

augmented_input = data_augmentation(input)

preprocessed_input = tf.keras.applications.mobilenet_v2.preprocess_input(augmented_input)

base_model_output = base_model(preprocessed_input, training = True)

avg_layer_output = global_avg_layer(base_model_output)

avg_layer_output_dropout = tf.keras.layers.Dropout(0.2)(avg_layer_output)

output = prediction_layer(avg_layer_output_dropout)

model = tf.keras.Model(input, output)

# compile the model

```

```

early_stop_callback = tf.keras.callbacks.EarlyStopping(monitor = "loss", patience = 5)

model.compile(
    loss = tf.keras.losses.BinaryCrossentropy(from_logits = False),
    optimizer = "adam",
    metrics = [
        tf.keras.metrics.FalseNegatives(),
        tf.keras.metrics.FalsePositives(),
        tf.keras.metrics.TrueNegatives(),
        tf.keras.metrics.TruePositives(),
        "accuracy"
    ]
)

# train the model

history = model.fit(
    training_dataset,
    epochs = 100000,
    validation_data = validation_dataset,
    callbacks = [early_stop_callback]
)

#metrics = model.evaluate(test_dataset)

model.save("models/transfer")

```

3.4.3 evaluate.py

Скриптът е идентичен в трите системи. Целта му е да даде оценка на тренирания модел. Това е базата за сравнение между моделите. Ето как изглежда КОДЪТ:

```

import os
import tensorflow as tf

os.environ["TF_CPP_MIN_LOG_LEVEL"] = '3'

data_dir = "../../datasets/test/"

width = 224

height = 224

batch_size = 32

dataset = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    label_mode = "binary",
    class_names = ["clean", "dirty"],
    validation_split = 0.2,

```

```

        subset = "validation",
        seed = 123,
        image_size = (width, height),
        batch_size = batch_size
    )

# print(str(dataset)) # debug

custom = tf.keras.models.load_model("models/custom/")

custom.evaluate(dataset)

transfer = tf.keras.models.load_model("models/transfer/")

transfer.evaluate(dataset)

```

3.5 Клиент - сървър архитектурата

3.5.1 client.py

Клиентът установява връзка със сървъра и започва да изпраща снимки, като преди това ги кодира в UTF-16 формат (това е изискване на Python), когато потребител въведе път до снимките и натисне Enter клавиша на клавиатурата си. Програмата приключва след въвеждане на следната последователност от символи “\n\0” символ (натискане на Enter клавиш без въвеждане на път) или при невъзможност за изпращане на снимки (прекъсната връзка или неосъществена първоначално). Кодът изглежда по следния начин:

```

import socket
import cv2
import numpy

def program_exit():

    connection.close()

    print("exiting...")

    quit()

connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

connection.bind((socket.gethostname(), 0))

```

```

try:

    connection.connect((socket.gethostname(socket.gethostname()), 1024))

except:

    print("connection could not be established")

    program_exit()

print("connection has been established")

while True:

    path = input("path to file: ")

    if path == '':

        program_exit()

    try:

        img = cv2.imread(path)

        success, img_bytes = cv2.imencode(".jpg", img)

    except:

        print("image could not be loaded")

        continue

    if success:

        try:

            connection.sendall(img_bytes)

        except:

            print("connection to the server is lost")

            program_exit()

        print("sent " + path)

    else:

        print("image could not be loaded")

```

3.5.2 server.py

Сървърът отваря връзка и получава снимки. При получаване на снимка, сървърът я подава на трениран модел (невронна мрежа) и извежда резултата от класификацията (1 == мръсна щайга; 0 == чиста щайга). Резултатът е сигурността, с която мрежата прави класификацията. Сървърът работи на една нишка и използва `select()` функцията - имплементация на `select()` системна функция в Linux. Въпреки, че системната функция не би трябвало да работи на други операционни системи, Python имплементацията е кросплатформна (работи на всички операционни системи, които поддържат Python). T1 следи сокетите, отворените файлове и каналите (всичко с метод `fileno()`, който връща валиден файлов дескриптор), докато станат четими или записваеми, или възникне грешка в комуникацията. Ето как е реализиран скриптът:

```
import os
import tensorflow as tf
import socket
import select
import numpy
import cv2

os.environ["TF_CPP_MIN_LOG_LEVEL"] = '3'

model = tf.keras.models.load_model("model/")

listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

listener.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

listener.bind((socket.gethostname(socket.gethostname()), 1024))

listener.listen()

print("listening...")

sockets = [listener]

while True:

    ready = select.select(sockets, [], [], 10000000)[0]
```

```

if ready:

    try:

        listener.settimeout(0.1)

        new_sock = (listener.accept()[0])

        sockets.append(new_sock)

        listener.settimeout(None)

        print("client has connected: " + str(new_sock))

    except:

        listener.settimeout(None)

sockets_count = len(sockets)

if sockets_count > 1:

    for cur in range(1, sockets_count):

        try:

            sockets[cur].send('0'.encode())

            img_bytes = sockets[cur].recv(5000000)

        except Exception:

            print("client has disconnected: " + str(sockets[cur]))

            sockets.remove(sockets[cur])

            continue

        if not img_bytes:

            continue

        img_arr = numpy.frombuffer(img_bytes, numpy.uint8)

        img = cv2.imdecode(img_arr, cv2.IMREAD_COLOR)

        img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

        hue = (img_hsv[:, :, 0] < 100) * (img_hsv[:, :, 0] > 3)

        saturation = (img_hsv[:, :, 1] > 70)

        value = (img_hsv[:, :, 2] > 120)

        mask = saturation * value * hue

        img_hsv[:, :, 0] *= mask

```



```
img_hsv[:, :, 1] *= mask

img_hsv[:, :, 2] *= mask

img = cv2.cvtColor(img_hsv, cv2.COLOR_HSV2RGB)

img = cv2.resize(img, (224, 224))

img = numpy.expand_dims(img, axis = 0)

print(model(img))
```

Четвърта глава

Ръководство на потребителя

4.1 Хардуерни и софтуерни изисквания

Програмният продукт поддържа всички машини, които могат да поддържат Python (няма значение коя версия). За максимална ефикасност се препоръчва наличието на графична карта, която е съвместима с модула TensorFlow. Продуктът е тестван на 2 различни софтуерни платформи - Windows 10 и Linux 5.17. Теоретично, не би трябвало да е проблем да работи и на BSD.

4.2 Инсталация

Първо, кодът трябва да се изтегли от официалния източник (Github: <https://github.com/stefan-sotirov-elsys/Thesis/>). След което е нужна инсталацията на модулите OpenCV, NumPy и Tensorflow. Препоръчително е използването на Anaconda за целта, но не е задължително. Ако трябва да се работи с входни данни, първо трябва да се получат при иск на следния имейл адрес: stefan.s.sotirov.2017@elsys-bg.org.

4.3 Управление на продукта

Скриптовете се стартират с Python интерпретатор.

4.3.1 Входни данни

Входните данни се намират в `datasets/` директориите. Снимки могат да се добавят и изваждат свободно, ако е нужно изравняване на данните, трябва да се стартира скрипта `expand.py` в `code/data/`.

4.3.2 Обработка на данни и машинно самообучение

Нужно е стартирането на избран скрипт и изчакването му да приключи изпълнение. По време на изпълнение, програмата извежда прогреса си визуално. Рисково е стартирането на `process.py` върху обработени данни, защото алгоритмите очакват необработени снимки - може да доведе до нежелани промени от типа напълно черно изображение.

4.3.3 Клиент - сървър

Нужно е стартирането на скриптовете `server.py` и `client.py`. Клиентският скрипт ще приключи изпълнение, ако сървърът не е отворил връзка:

```
connection could not be established  
exiting...
```

Ако сървърът затвори връзката, след като тя е установена между устройствата, клиентският скрипт ще приключи изпълнение след нов опит за изпращане на снимка:

```
connection to the server is lost  
exiting...
```

Ако клиентът не въведе нищо освен Enter клавиш, скриптът също ще приключи изпълнение:

```
path to file:  
exiting...
```

При успешно изпращане на снимка се извежда съобщение от вида:

```
sent clean.jpg
```

При неуспешно изпращане на снимка се извежда съобщение от вида:

```
[ WARN:0@55.606] global  
D:\a\opencv-python\opencv-python\opencv\modules\imgcodecs\src\loadsave.cpp (239) cv::findDecoder  
imread_('incorrect.jpg'): can't open/read file: check file path/integrity  
image could not be loaded
```

При успешно стартиране на сървър скрипта се извежда следното съобщение:

```
listening...
```

При установяване на връзка сървърът извежда информация за клиента от типа:

```
client has connected: <socket.socket fd=3124, family=AddressFamily.AF_INET,  
type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.1.44', 1024), raddr=('192.168.1.44',  
63656)>
```

При успешно получаване на снимка се извежда резултат от вида:

```
tf.Tensor([[0.06666712]], shape=(1, 1), dtype=float32)
```

При прекъсване на връзка от страна на клиента се извежда съобщение от вида:

```
client has disconnected: <socket.socket fd=3128, family=AddressFamily.AF_INET,  
type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.1.44', 1024), raddr=('192.168.1.44',  
63676)>
```

Заклучение

Настоящата дипломна работа има завършен облик и е пример за система за засична на замърсявания в контейнер, създадена чрез скриптовия език Python.

Проектът спазва всички функционални изисквания, описани в началото на заданието и е реализирана с внимание към детайлите. Употребената клиент - сървър архитектура за реализиране на решението осигурява бърз и надежден синхрон между потребителите и сървъра.

Продуктът има широка гама от възможности за бъдещо развитие - внедряване на http архитектура, подобряване на моделите на невронни мрежи и едновременна обработка върху данни на различни машини.

Исползвана литература

1. <https://realpython.com/python-opencv-color-spaces/>
2. <https://www.youtube.com/watch?v=aircAruvnKk>
3. <https://machinelearningmastery.com/>
4. <https://alex.smola.org/drafts/thebook.pdf>
5. https://www.deeplearningbook.org/contents/linear_algebra.html
6. <https://www.deeplearningbook.org/contents/ml.html>
7. <https://www.deeplearningbook.org/contents/mlp.html>
8. <https://www.cin.ufpe.br/~cavmj/Machine%20-%20Learning%20-%20Tom%20Mitchell.pdf>
9. <https://www.geeksforgeeks.org/machine-learning/>

Съдържание

Мнение на научен ръководител	6
Увод	7
Първа глава	9
Теоретична част	9
1.1 Алгоритми за обработка на снимки	9
1.1.1 Въведение	9
1.1.2 RGB моделът	9
1.1.3 Снимката като файл	10
1.1.4 Копиране на изображения	12
1.1.5 Преоразмеряване (скалиране, мащабиране) на снимки	12
1.1.6 Маскиране на снимки	13
1.1.7 Откриване на ръбове	15
1.2 Принципи на машинно самообучение и невронни мрежи	18
1.2.1 Въведение	18
1.2.2 Персептрон	20
1.2.3 Сигмоидни неврони	24
1.2.4 Архитектура на невронна мрежа	26
1.2.5 Обучение с градиентно спускане (спускане по наклон)	28
1.2.6 Конволюционни мрежи	34
1.3 Технологии и практики в разработването на софтуер, занимаващ се с изображения	41
Втора Глава	43
Практическа част	43
2.1 Проблемът	43
2.2 Решението	44
2.2.1 Система 1 - без обработка	46
2.2.2 Система 2 - изолиране на контейнер	46
2.2.3 Система 3 - изолиране на ръбове	47
2.2.4 Входните данни	48
2.2.5 Архитектури на невронните мрежи	49
2.2.6 Изводи	51
2.2.7 Тестовата среда: клиент - сървърната архитектура	54
Трета глава	55
Програмна реализация на система за засичане на замърсявания в контейнери	55
3.1 Използвани технологии	55
	80

3.2 Въведение в структурата на проекта	56
3.2.1 net/	57
3.2.2 unprocessed/, container_isolation/, edge_islation/	57
3.3 Обработка на входните данни	58
3.3.1 process.py	58
3.3.2 expand.py	62
3.4 Машинно самообучение	64
3.4.1 custom.py	64
3.4.2 transfer.py	67
3.4.3 evaluate.py	69
3.5 Клиент - сървър архитектурата	70
3.5.1 client.py	70
3.5.2 server.py	72
Четвърта глава	75
Ръководство на потребителя	75
4.1 Хардуерни и софтуерни изисквания	75
4.2 Инсталация	75
4.3 Управление на продукта	76
4.3.1 Входни данни	76
4.3.2 Обработка на данни и машинно самообучение	76
4.3.3 Клиент - сървър	76
Заклучение	79
Използвана литература	80
Съдържание	81