Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра теоретичної механіки

ДИПЛОМНА РОБОТА

бакалавра

на тему: **«Створення клієнт-серверного додатку з використанням бібліотеки TensorFlow для розпізнавання графічного формату»**

«Creating a client-server application using the TensorFlow library for graphic format recognition»

Виконав: студент денної форми навчання
спеціальності 113 Прикладна математика
спеціалізація теоретична механіка

Очинський Микита Євгенович

Науковий керівник:

старший викладач_____Косирева Л.А.

Рецензент:

доцент_____Косой М.Б.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ р.

Завідувач кафедри

_____ Волков В. Е.
(підпис)

Захищено на засіданні ЕК №_

протокол № ____ від _____р.

Оцінка_____/_____/____
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ Волков В. Е.
(підпис)

Одеса – 2021

ЗМІСТ

ВСТУП

Актуальність теми полягає в тому, що в даний час йде розпал епідемії COVID-19 по всьому світу, і один із способів боротьби з ним - це обмеження контактів між людьми, з чим допомагає звичайна медична маска. Але є люди які не дотримуються правил і обмежень введених владою нашої країни. Для виявлення цих людей можна використовувати комп'ютери, які можуть служити автоматизованим обмежувачем наприклад відвідуваності місць скупчення людей, такі як: магазини, парки, школи, університети, лікарні, місця заходу в суспільне транспортний засіб: маршрутне таксі, метро.

У даній роботі буде використовуватися комп'ютерний зір, для виявлення та класифікації особи людини. Комп'ютерний зір працює за допомогою методів машинного навчання: нейронних мереж.

Мета: розібратися в теоретичному пристрої нейронних мереж і методів комп'ютерного зору для виявлення і класифікації об'єктів

Кінцева мета: написати програму для виявлення на зображенні осіб в масках і без.

Для написання програми необхідні знання мови програмування, теорію по нейромережах, згорткових (CNN) нейромереж та комп'ютерного зору.

РОЗДІЛ 1
НЕЙРОННІ МЕРЕЖІ

Штучні нейронні мережі, особливо, глибокі нейронні мережі (deep neural networks) по праву вважаються найбільш потужними моделями в машинному навчанні, здатними для ряду завдань досягати результатів, порівнянних з людськими. Однією з таких задач є розпізнавання образів (pattern recognition): рукописного тексту, символів на номерних знаках автомобілів, розпізнавання осіб, класифікація зображень, машинний переклад і інші. Класичні моделі ML (дерева рішень, методи найближчих сусідів, лінійні моделі), як правило, не здатні відновлювати складні нелінійні залежності (або ж сильно перенавчаються при цьому) і на подібних завданнях показують результати незрівнянно гірше, ніж з ними справляються живі люди.

Варто сказати, що ідея штучних нейронних мереж і їх використання в задачах розпізнавання образів далеко не нова і налічує вже близько 70 років. Однак успіхи в застосуванні (глибоких) нейронних мереж на практиці в основному відбулися в останні два-три десятиліття. В першу чергу це пов'язано з істотним "апгрейдом" апаратної частини комп'ютерів (CPU, GPU, TPU). Можливість навчання глибоких і складних мереж привели також до появи нових архітектур, пристосованих до вирішення конкретних завдань. Так, в кінці 20 - початку 21 століття з'явилися:

- згортованні мережі (convolutional neural networks, CNN) для розпізнавання, класифікації та сегментації зображень,
- рекурентні мережі (recurrent neural networks, RNN) для машинного перекладу і обробки сигналів,
- генеративні змагальні мережі (generative adversarial networks, GAN) для створення нових об'єктів на базі існуючих.

Біологічний нейрон має власну будову, яка уявляє собою мозок живих істот, що складається з нервових клітин - нейронів. Кожен нейрон має безліч відростків (дендритів), за якими в нього передається інформація (у вигляді електричних імпульсів), а також один аксон, який передає ці імпульси далі по нейронної мережі. Нейрони зв'язані за допомогою синапсів. Сигнали, що надходять в тіло нейрона (його називають сомой), підсумовуються, і, якщо ця сума перевищує певний поріг активації**,** нейрон активується і передає імпульси далі по мережі. Це дуже спрощене опис, однак воно допоможе зрозуміти загальне пристрій штучних нейронних мереж.

1.1. Математична модель нейрона (1943)

Вперше математична модель штучного нейрона була описана У. Маккалоком і У. Питтсом в 1943 році.

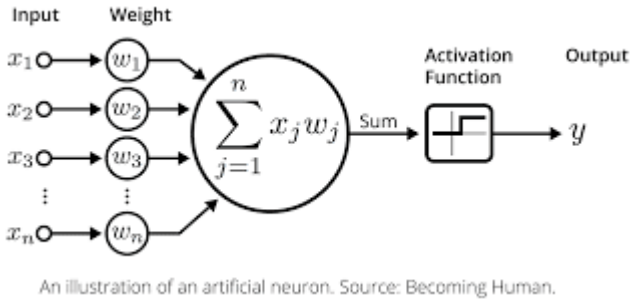


Рис.1 Математична модель штучної нейронної мережі

Ідея роботи нейрона досить проста і нагадує те, що відбувається в реальності: якщо позначити входи як $x1$, $x2$, ..., xn, то обчислення одного нейрона складаються з двох етапів:

1. обчислення зваженої суми вхідних сигналів: $w1x1 + w2x2 + ... + wnxn$
2. застосування активаційної функції: $y = g\ (\Sigma wjxj)$

Таким чином, нейрон приймає на вхід n сигналів (xj) і на виході генерує єдине значення y.

Перша штучна нейронна мережа (перцептрон) на основі цієї моделі була

створена Френк Розенблат в 1958 році. (рис.1)

1.2. Функції поширення

Розглянемо докладніше етап активації нейрона. Спочатку, за аналогією з біологічними нейронами, в мережах використовували порогову функцію активації. Суть її в наступному: якщо зважена сума сигналів на вході перевищувала наперед заданий поріг активації, то нейрон повертав значення 1 (активувався), в іншому випадку - 0. Приблизно так працюють наші нейрони в мозку, так як їх сигнали являють собою електричні імпульси.

Згодом стало зрозуміло, що для математичної моделі нейрона і нейронних мереж не обов'язково, щоб вихід нейрона був бінарним, і з'явилися інші функції активації. Зокрема, стала популярною сигмоїдальна функція поширення (Рис. 2):

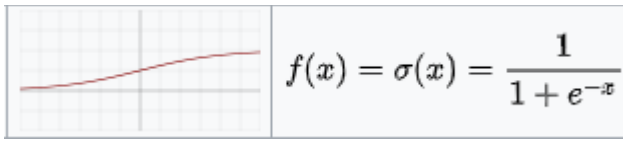


Рис. 2 Сигмоїдальна функція

Лінійна регресія як нейронна мережа

Подивимося на модель лінійної регресії. У цій моделі вихідне значення (y^) виходить як зважена сума вхідних значень (Σwjxj) плюс вільний член (w0).

Технічно лінійна регресія є "нейронну мережу" з одного нейрона з тотожною функцією поширення, т.е. G (z) = z.

Логістична регресія як нейронна мережа

Подивимося на модель логістичної регресії. Вихідне значення моделі логістичної регресії є ймовірність класу 1, т.е. P + = σ (Σwjxj).

Логістична регресія - це також "нейронна мережа" з одного нейрона з сігмоідальною функцією поширення.

Багатошарові нейронні мережі. Дві попередні розглянуті моделі включали два шари - вхідний шар (input layer), що включає в себе xj і вихідний шар (output layer) з функцією поширення. При підрахунку кількості шарів нейронної мережі вхідний шар не враховується. Таким чином, лінійна і логістична регресії є одношарові нейронні мережі.

Найпростіша багатошарова мережа складається з двох шарів (Рис. 3) (не рахуючи вхідного):

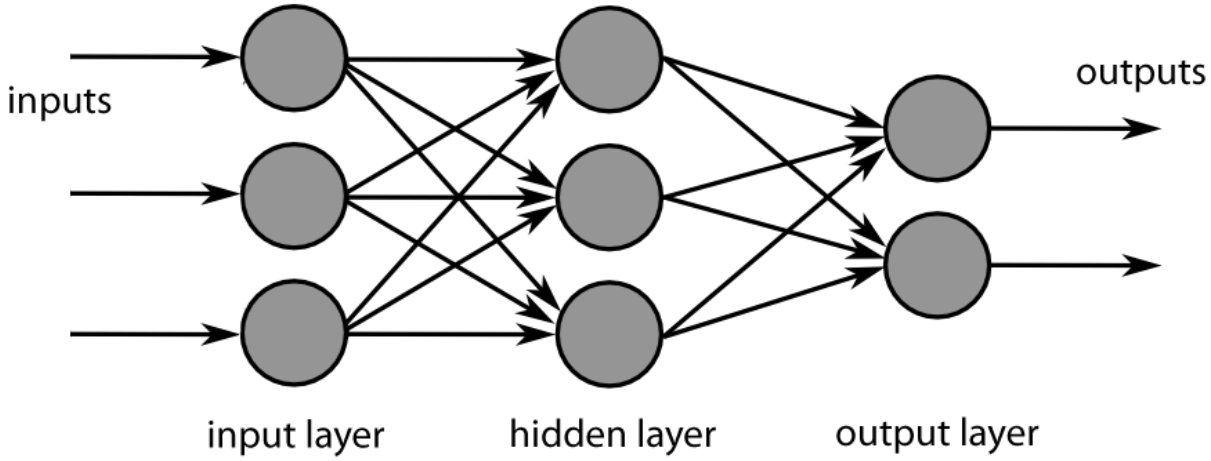


Рис.3 Двоскладова нейронна мережа

Проміжний шар називають прихованим шаром (hidden layer). Якщо проміжних шарів більше ніж один, мережа прийнято називати глибокої (deep neural network). У зв'язку з цим розділ машинного навчання, вивчає глибокі нейронні мережі, носить назву Deep Learning (глибоке, або глибинне, навчання).

Обчислення за допомогою нейронної мережі. Дивлячись на зображення вище, може здатися, що багатошарова мережа - "ускладнена версія" лінійної регресії. Адже кожен нейрон приймає в себе лінійну комбінацію своїх входів і повертає вихід, обчислений за допомогою функції поширення. Далі ці виходи надходять на вхід наступного шару, там обчислюються інші лінійні комбінації і так далі. Однак ключовим місцем відмінності нейромереж від лінійної регресії є застосування нелінійних функцій активації. Функція поширення дозволяє

мережам знаходити куди більш складні закономірності, ніж на це здатна лінійна модель. Функція поширення Rectified Linear Unit (ReLU) (Рис.4):

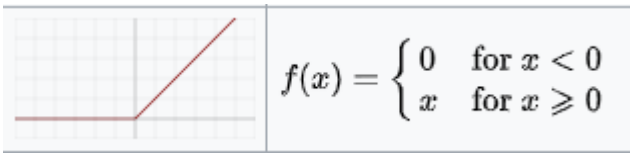


Рис. 4 Випрямляча лінійна функція

є найбільш популярною функцією поширення в прихованих шарах.

Теорема універсальності стверджує, що для будь-якої як завгодно складної функції існує нейронна мережа, яка апроксимує цю функцію.

1.3. Обчислювальний граф. Пряме поширення

Досить часто нейромережі представляють за допомогою так званих обчислювальних графів.

Знайдемо рішення трьох рівнянь:

$$e = c * d$$

$$c = a * b$$

$$d = b + 5$$

де a = 2, та b = 5

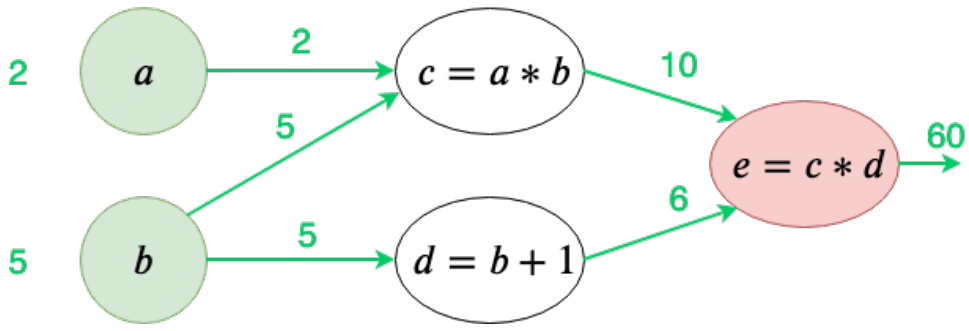


Рис. 5 Обчислювальний граф. Пряме поширення

На малюнку зображена “нейронна мережа", яка обчислює значення **e** на підставі входів a і b:

$$e = c \cdot d = (a \cdot b) \cdot (b + 1).$$

При a = 2 і b = 5 послідовно виходить значення e = 60.

Обчислення вихідного значення на основі вхідних, слідуючи логікі обчислювального графа мережі, називається прямим поширенням (forward propagation або forward prop).

Навчання мережі та зворотне поширення. Розберемо на прикладі попереднього простого графа, як же навчаються нейронні мережі. Навчання будь-якої моделі - це оптимізація її параметрів. Оптимізація зазвичай відбувається чисельно за допомогою методів градієнтного спуску. Тобто для того, щоб оновити значення вектора параметрів w, треба обчислити градієнт (частинні похідні) цільової функції J по w і оновити вектор ваг по формулі

$$w: = w - \alpha \cdot \partial J / \partial w,$$

де $\alpha > 0$ - темп навчання (learning rate).

Отже, ключовий момент в навчанні мереж - знаходити похідні за параметрами на підставі обчислювального графа.

В даному випадку "цільова функція" **e** залежить від функцій c і d, які залежать від a і b. Тобто в термінології математичного аналізу **e** є складною функцією. За правилом диференціювання складної функції похідні обчислюються починаючи з "зовнішніх" функцій і закінчуючи "внутрішніми":

$$dg (f (x)) dx = dg / df * df / dx$$

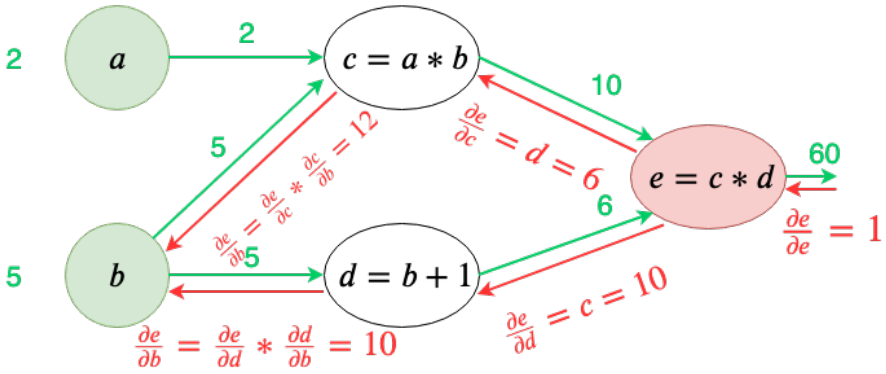


Рис. 6 Обчислювальний граф. Зворотне поширення

Отже, йдучи від вихідного шару (вузол e) до проміжного (c і d), обчислюємо:

$$\partial e\, \partial d = \partial\, (c \cdot d)\, \partial d = c,$$

$$\partial e \partial c = \partial\, (c \cdot d)\, \partial c = d.$$

Далі, при переході від проміжного шару до вхідного (вузли a і b):

$$\partial e \partial a = \partial e\, \partial c \cdot \partial c\, \partial a = d \cdot b,$$

$$\partial e\, \partial b = \partial e\, \partial c \cdot \partial c\, \partial b + \partial e\, \partial d \cdot \partial d\, \partial b = d \cdot a + c \cdot 1.$$

Такий прохід по обчислювальному графу мережі з обчисленням похідних називають зворотним поширенням (backward propagation, або backprop). (Рис.6)

1.4. Алгоритм навчання нейромереж

Розглянемо багатошарову нейронну мережу (у цьому прикладі кількість шарів 2):

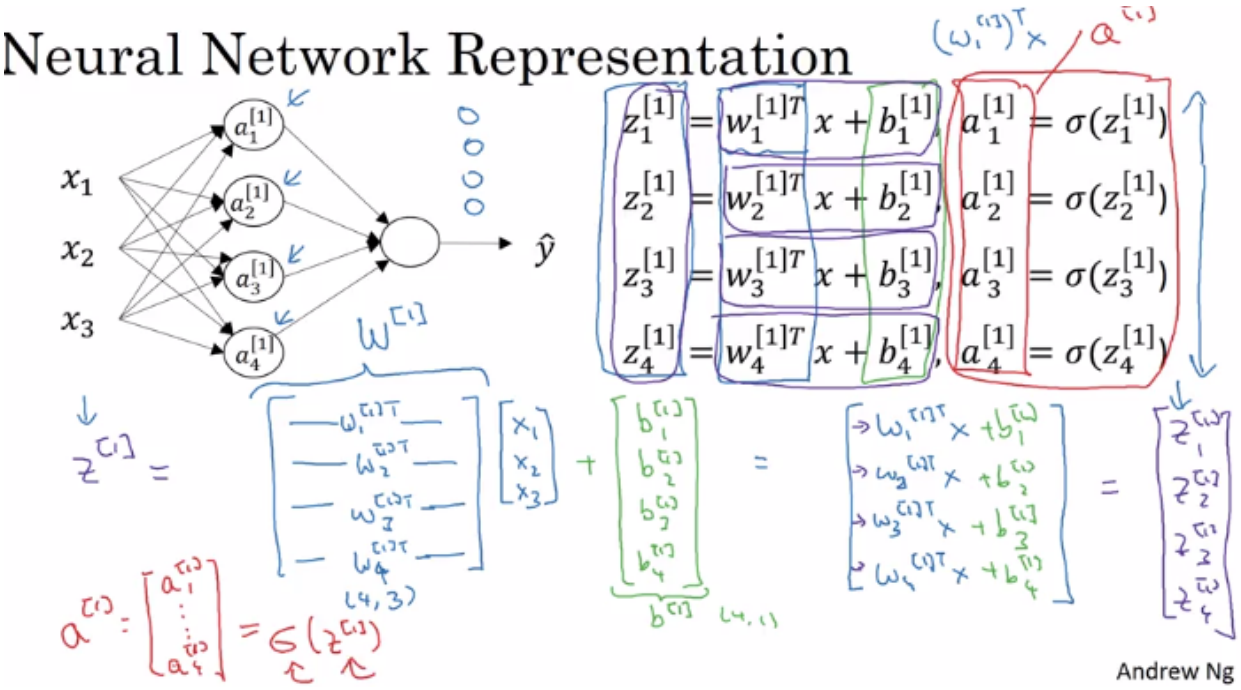


Рис. 7 Обчислювальний граф. Зворотне поширення

Позначення ak[l] варто читати так: k-й нейрон l-го шару. При цьому вхідний шар вважається "нульовим", т.е. Ak[0] = xk.

Кожен нейрон:

1. Обчислює зважену суму своїх входів
2. Застосовує до нього функцію поширення.

Позначимо зважену суму входів через zk[l], а функцію поширення g[l]. Таким чином, ak[l] = g[l] (zk [l]). Відзначимо, що активаційні функції в різних шарах можуть відрізнятися, але в межах одного шару прийнято використовувати одну і ту ж функцію.

Кожне zk першого шару в нашому прикладі є лінійною комбінацією:

$$zk = wk1x1 + wk2x2 + wk3x3 + bk \ (k = 1,4).$$

Або, в матрично-векторному вигляді

$$z = Wx + b,$$

де z, b --- вектори 4×1,

x --- вектор 3×1,

W --- матриця 4×3.

Потім до вектору z застосовується (поелементно) функція поширення:

$$a = g(z),$$

де a - вихід шару, також є вектор 4×1

У загальному випадку для шару [l] обчислення z і a виглядає так:

$$Z[l] = W[l]\, a[l-1] + b[l],$$

$$a[l] = g(z[l]).$$

Таким чином, для одного навчального прикладу (x(i)) етап forward prop полягає в послідовному обчисленні z[l] і a[l] для кожного шару від 1 до L. Останнє значення a[L] є виходом нейронної мережі: y^ = a[L]. На підставі виходу обчислюється функція втрат (функція втрат - це функція для обчислення помилки на одному навчальному прикладі).

Цей крок можна здійснити відразу для всіх навчальних прикладів, якщо в формулах замінити z[l] і a[l] на матриці: Z[l] і A[l] розмірності $n[l] \times m$, де n[l] - кількість вузлів в шарі [l], m - довжина навчальної вибірки.

Знаючи функції витрат для кожного навчального прикладу, можна обчислити функціонал якості - цільову функцію нашої мережі. Саме її треба мінімізувати за допомогою градієнтного спуску по параметрам.

На етапі backprop, відбувається зворотний прохід по мережі з метою обчислення похідних функції втрат за параметрами: ∂J∂W[l], ∂J∂b[l] для кожного шару [l] від L до 1. Нарешті, відбувається оновлення всіх параметрів мережі за методом градієнтного спуску:

$$W[l]: = W[l] - \alpha \cdot \partial J \partial W[l],$$
$$b[l]: = b[l] - \alpha \cdot \partial J \partial b[l].$$

Підсумуємо: навчання мережі відбувається в чотири етапи:

1. ініціалізація параметрів (завдання початкових значень - як правило, випадковим чином);
2. forward prop, обчислення виходу мережі і функціоналу якості;
3. backprop, обчислення похідних за параметрами;
4. оновлення параметрів.

Кроки 2-4 повторюються до тих пір, поки не буде досягнут мінімум цільової функції (критерій збіжності). З огляду на те, що досягнення мінімуму для таких складних функцій малореально, на практиці частіше за все задають фіксоване число ітерацій (епох) градієнтного спуску.

РОЗДІЛ 2
КОМП'ЮТЕРНИЙ ЗІР

Комп’ютерний зір або комп'ютерне бачення — теорія та технологія створення машин, які можуть проводити виявлення, стеження та визначення об'єктів.

2.1. Бібліотека OpenCV

Для роботи з зображеннями і знаходження особи на зображенні скористаємося бібліотекою, розробленою на мові програмування Python, OpenCV.

Для використання бібліотеки OpenCV необхідно прописати наступну команду в python IDLE

- import cv2 as cv

Для читання зображення використовуємо функцію:

- img = cv.imread(path), де
  path - це шлях до файлу зображення
  img - змінна де треба зберігати зображення

Для того, щоб вирізати потрібний нам шматок зображення, будемо використовувати наступний синтаксис:

- img[y: y+h, x: x+w], де
  (x, y) - координата точки, звідки необхідно почати вирізати зображення
  (w, h) - довжина і ширина в пікселях.

Щоб поміняти розмір зображення нам необхідна наступна функція:

- cv.resize(img, (w, h), interpolation = cv.INTER_CUBIC), де
  img - змінна зберігає зображення
  (w, h) - бажаний розмір підсумкового зображення
  interpolation = cv.INTER_CUBIC - спосіб інтерполяції зображення.

Щоб намалювати фігуру, в нашому випадку квадрат, нам необхідна наступна функція:

- cv.rectangle(img, (x, y), (x + w, y + h), (r, g, b), thickness = 2), де

img - змінна зберігає зображення

(x, y) - координата точки , звідки необхідно почати вирізати зображення

(w, h) - довжина і ширина в пікселях

(r, g, b) - трійка чисел від 0 до 255, для вибору кольору

thickness - ширина лінії квадрата в пікселях

Для збереження зображення в обрану директорію - будемо використовувати функцію:

- cv.imwrite(path, img), де
  path - це шлях до файлу і назва файлу, куди ми запишемо зображення
  img - змінна зберігає зображення.

2.2. Знаходження особи на зображенні

Будемо використовувати OpenCV, бібліотеку з відкритим вихідним кодом для комп'ютерного зору.

Каскадні класифікатори

Каскадний класифікатор, що працюють з хааропоподобнимі функціями, являє собою особливий випадок ансамблевого навчання, званий підвищенням. Як правило, спирається на AdaBoost класифікатори.

Каскадні класифікатори навчаються на кількох сотнях зображень, які містять об'єкт, який ми хочемо виявити, і інших зображеннях, які не містять цих зображень.

Для визначення особи на зображені використовується алгоритм, званий середовищем виявлення об'єктів Viola - Jones, який включає в себе всі етапи, необхідні для виявлення живого особи:

1. Вибір об'єктів Хаара, особливості, отримані з вейвлетов Хаара
2. Створити цілісне зображення
3. Adaboost Training
4. Каскадні класифікатори

Розглянемо вибір Хаара: є деякі загальні риси, які знаходяться на самих звичайних людських обличчях:

- область темних очей в порівнянні з верхніми щоками
- яскрава область перенісся в порівнянні з очима
- якесь конкретне розташування очей, рота, носа ...

Характеристики називаються Haar Features. процес вилучення функції буде виглядати так (Рис. 8):

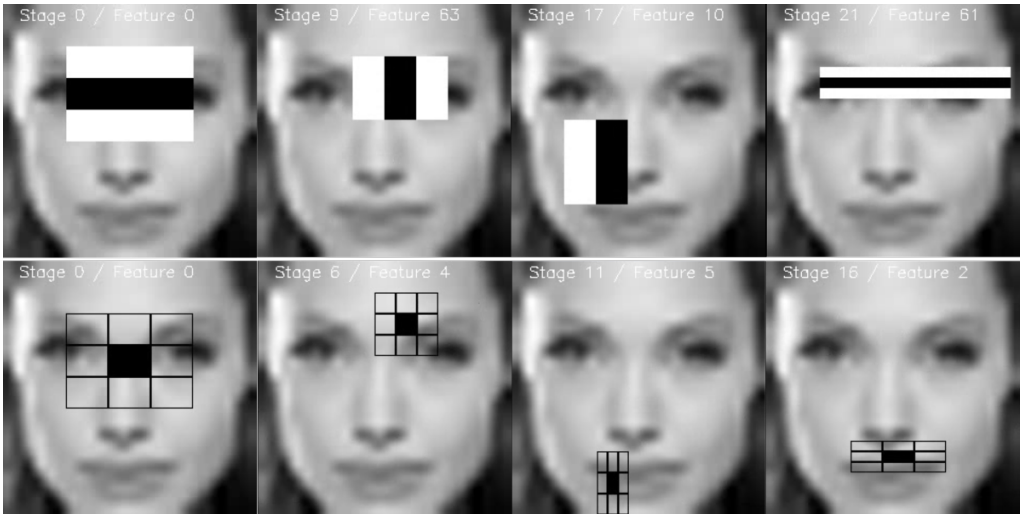


Рис. 8 Процес вилучення об'єктів Хаара

Дивлячись на особливості Haar Features (Хаара), можна зробити висновок, що у цьому прикладі перша ознака вимірює різницю в інтенсивності між областю очей і областю через верхні щоки. Значення об'єкта просто обчислюється шляхом підсумовування пікселів в чорній області і віднімання пікселів в білій області.

$$RectangleFeature = \sum(pixels_{blackarea}) - \sum(pixels_{whitearea})$$

Потім застосовуємо цей прямокутник як згорткове ядро по всьому зображенню. Щоб бути вичерпним, треба застосувати всі можливі розміри і положення кожного ядра. Прості 24 * 24 зображення зазвичай дають більше

160 000 об'єктів, кожне з яких складається з суми / віднімання значень пікселів. В обчислювальному відношенні це було б неможливо для живого виявлення особи. Отже, можна прискорити цей процес:

- як тільки гарна область була ідентифікована прямокутником, марно запускати вікно по абсолютно іншій області зображення. Це може бути досягнуто Adaboost.
- обчислити прямокутні елементи, використовуючи принцип інтегрального зображення, який набагато швидше. Розглянемо це в наступному розділі.

Існує кілька типів прямокутників, які можна застосовувати для вилучення об'єктів Haar. Згідно оригінальної статті:

- двох вугільний ознака - це різниця між сумою пікселів в двох прямокутних областях, використовувана в основному для виявлення країв (a, b).
- функція чотирикутника обчислює різницю між діагональними парами прямокутника (e).

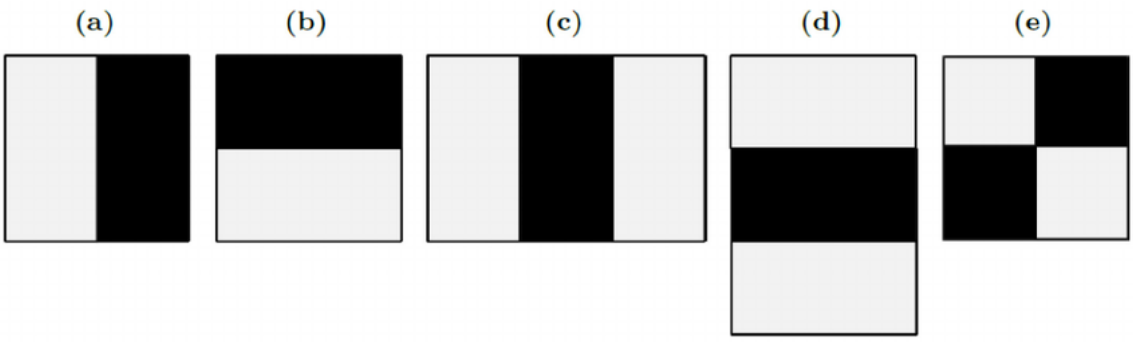


Рис. 9 Прямокутники, для вилучення об'єктів Haar

Розглядаючи прямокутники Хаара, коли функції обрані, застосовуємо їх до набору навчальних зображень, використовуючи класифікацію Adaboost, яка об'єднує набір слабких класифікаторів для створення точної моделі ансамблю. Завдяки 200 функцій (замість 160 000 спочатку) досягається точність 95%. Автори статті вибрали 6 000 функцій.

Цілісне зображення.Обчислення елементів прямокутника в стилі згорткового ядра може бути довгим. З цієї причини автори, Віола і Джонс, запропонували проміжне представлення для зображення: цілісне зображення. Роль інтегрального зображення полягає в тому, щоб просто обчислити будь-яку прямокутну суму, використовуючи тільки чотири значення.

1. Визначається об'єкти прямокутника в даному пікселі з координатами (x,y).
2. Виконується інтегральне зображення пікселя в сумі пікселів вище і зліва від даного пікселя.

$$ii(x,y) = \sum_{x' \le x, y' \le y} i(x', y')$$

де ii(x, y) - інтегральне зображення, а i(x, y) - вихідне зображення. Обчислюючи ціле цілісне зображення, виникає рецидив форми, який вимагає тільки одного проходу вихідного зображення. Дійсно, можна визначити наступну пару повторень:

$$s(x,y) = s(x,y-1) + i(x,y)$$
$$ii(x,y) = ii(x-1,y) + s(x,y)$$

де s(x, y) - сукупна сума рядків, а s(xâ1 ') = 0, ii(â"1, y) = 0.

Чим це може бути корисно? Добре, розглянемо область D, щоб оцінити суму пікселів. Визначимо 3 інших регіону: A, B і C.

- Значення інтегрального зображення в точці 1 є сумою пікселів в прямокутнику A.
- Значення в точці 2 A + B
- Значення в точці 3 A + C
- Значення в точці 4 - A + B + C + D.

Отже, сума пікселів в області D може бути просто обчислена як: 4+1(2+3).

І за один прохід підрахували значення усередині прямокутника, використовуючи тільки 4 посилання на масив.

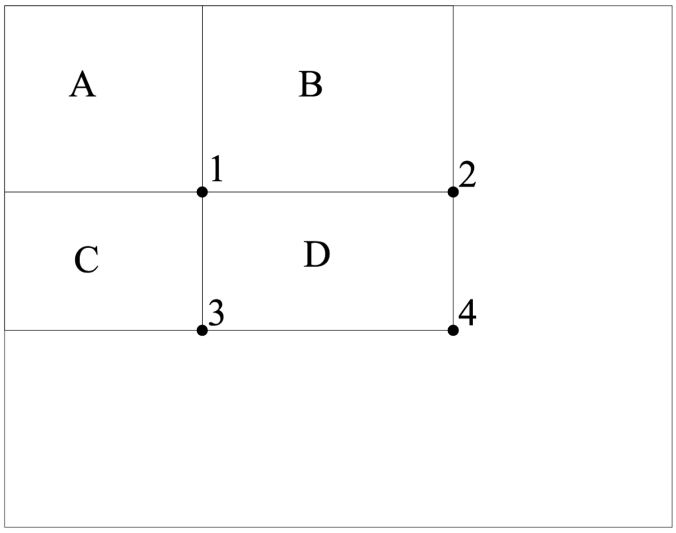


Рис. 10 Цілісне зображення

Потрібно просто пам'ятати, що прямокутники - досить прості функції на практиці, але достатні для виявлення особи. Керовані фільтри мають тенденцію бути більш гнучкими, коли мова йде про складні проблеми

3. Вивчення функції класифікації з Adaboost.

З огляду на набір помічених тренувальних образів (позитивних або негативних), Adaboost використовується для:

- обрати невеликий набір функцій
- і навчити класифікатор

Оскільки передбачається, що більшість функцій з 160 000 не мають ніякого значення, слабкий алгоритм навчання, на основі якого поліпшується модель, призначений для вибору одного прямокутника, який розділяє кращі негативні і позитивні приклади.

4. Каскадний класифікатор.

Хоча процес, описаний вище, досить ефективний, основна проблема залишається. Велика частина зображення - це область без обличчя. Надавати однакову важливість кожної області зображення не має сенсу, оскільки треба зосередитися в основному на областях, які, швидше за все, містять

зображення. Віола і Джонс досягли підвищеній швидкості виявлення, скоротивши при цьому час обчислень за допомогою каскадних класифікаторів.

Ключова ідея полягає в тому, щоб відхилити області, які не містять граней, при визначенні областей, які мають. Оскільки завдання полягає в тому, щоб правильно ідентифікувати особу, ми хочемо мінімізувати кількість помилкових негативних результатів, тобто області, які містять обличчя і не були ідентифіковані як такі.

Ряд класифікаторів застосовується до кожної області. Ці класифікатори є простими деревами рішень:

- якщо перший класифікатор позитивний, переходимо до другого
- якщо другий класифікатор позитивний, переходимо до третього
- â € |

Будь-негативний результат в деякій точці призводить до відхилення області як потенційно містячий особу. Початковий класифікатор виключає більшість негативних прикладів при низьких обчислювальних витратах, а наступні класифікатори усувають додаткові негативні приклади, але вимагають великих обчислювальних зусиль.

Класифікатори навчаються з використанням Adaboost і налаштуванням порога, щоб мінімізувати неправдиву оцінку. При навчанні такої моделі змінними є такі:

- кількість ступенів класифікатора
- кількість функцій на кожному етапі
- поріг кожного етапу

В OpenCV вся ця модель уже пройшла навчання по розпізнаванню осіб.

Імпорт. Наступний крок - знайти попередньо підготовлені ваги. Тому потрібно використовувати попередньо навчені моделі за замовчуванням для визначення особи.

Візьмемо стандартну модель для визначення особи haarcascade_frontalface_default.xml

Для завантаження моделі використовуємо таку функцію:

cv.CascadeClassifier ("haarcascade_frontalface_default.xml")

Виявляємо обличчя і додаємо навколо нього прямокутник:

```
# Detect faces
faces = faceCascade.detectMultiScale(gray,
                                     scaleFactor = 1.1,
                                     minNeighbors = 5,
                                     flags = cv2.CASCADE_SCALE_IMAGE
                                     )
# For each face
for (x, y, w, h) in faces:
   # Draw rectangle around the face
   cv2.rectangle (gray, (x, y), (x + w, y + h), (255, 255, 255), 3)
```

Список найбільш поширених параметрів detectMultiScale функція:

- scaleFactor: параметр, який вказує, на скільки зменшується розмір зображення при кожному масштабі зображення.
- minNeighbors: параметр, який вказує, скільки сусідів повинен мати кожен прямокутник-кандидат для його збереження.
- minSize: мінімально можливий розмір об'єкта. Об'єкти меншого розміру ігноруються.
- maxSize: максимально можливий розмір об'єкта. Об'єкти більшого розміру ігноруються.

2.3. Побудова моделі класифікації за допомогою бібліотеки TensorFlow. Згорткові нейронні мережі

Для класифікації об'єктів потрібно вивчити базову теорію нейромереж і

згортальних мереж.

Найкращі результати в області розпізнавання осіб показала Convolutional Neural Network або згорткова нейронна мережа (далі - ЗНМ)

Згорткові нейронні мережі забезпечують часткову стійкість до змін масштабу, зсувів, поворотам, зміні ракурсу і іншим спотворень. Згорткові нейронні мережі об'єднують три архітектурних ідеї, для забезпечення інваріантності до зміни масштабу, повороту зрушення і просторовим спотворень:

- локальні рецепторні поля (забезпечують локальну двовимірну зв'язність нейронів);
- загальні вагові коефіцієнти синапсів (забезпечують детектування деяких рис в будь-якому місці зображення і зменшують загальне число вагових коефіцієнтів);
- ієрархічна організація з просторовими підвибірки.

Розглядаючи структуру згорткової нейронної мережі слід зазначити, що згортка, використовувана для класифікації зображень, складається з двох частин: згорткової основи і щільною головки.

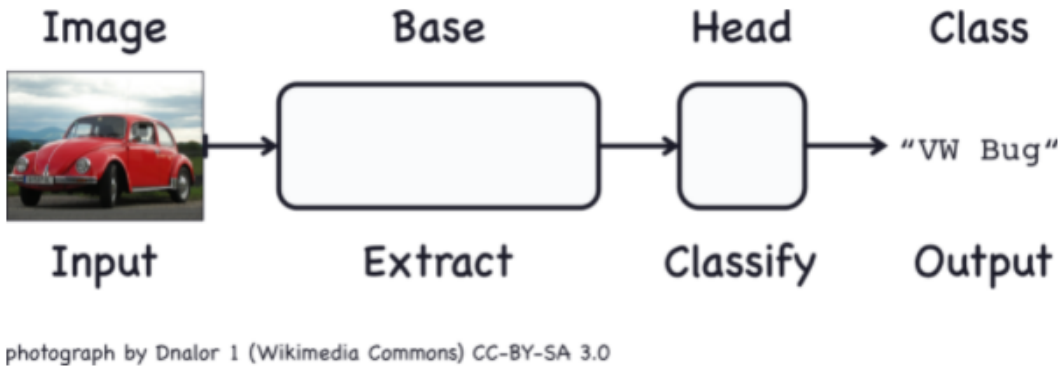


Рис. 11 Структура ЗНМ (згорткової нейронної мережі)

База (Base) використовується для вилучення особливостей, ознак (features) з зображення. Вона складається в основному з шарів, що виконують операцію згортки, але часто включає і інші типи шарів.

Голова (Head) використовується для визначення класу зображення. Вона складається в основному з щільних шарів, але може включати і інші шари, такі як dropout.

Витяг складається з трьох основних операцій:

1. Фільтрація зображення за певною ознакою (згортка/Convolution)
2. Виявити цю функцію в відфільтрованому зображенні (ReLU)
3. Стиснути зображення, щоб розширити можливості (максимальне об'єднання / Maximum Pooling)

Наступний рисунок ілюструє цей процес. Можно бачити, як ці три операції можуть виділити деякі особливості вихідного зображення (в даному випадку горизонтальні лінії).

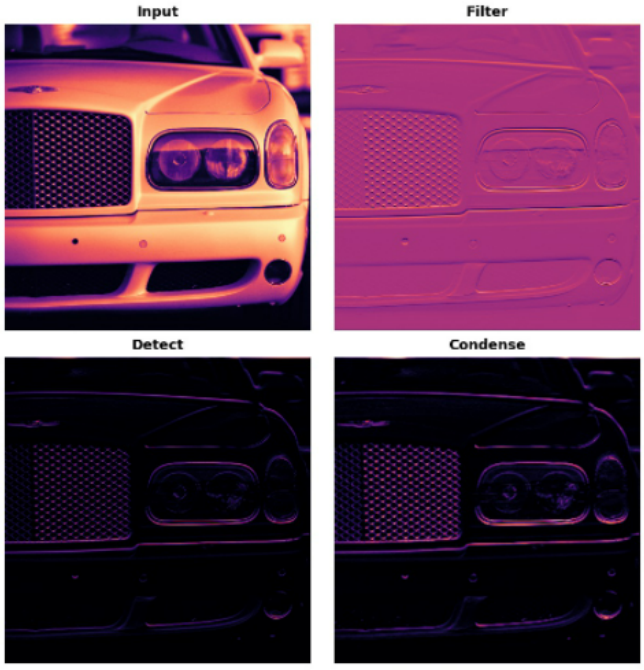


Рис. 12 Основні операцій для витягу ознак

Згорнутий шар виконує етап фільтрації.

Згортка (Convolution)

```
from tensorflow import keras
from tensorflow.keras import layers
```

```
model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3), # activation is None
    # More layers follow
])
```

Можна зрозуміти ці параметри, розглянувши їх зв'язок із вагами та активацією шару.

Ваги (Weight). Ваги, які вивчає convnet (згортка) під час тренування, в основному містяться в його згорткових шарах. Ці ваги називаються ядрами (Kernel). Можно представити їх як невеликі масиви:

Рис. 13 Приклад ядра в згортковому шарі

Ядро функціонує шляхом сканування зображення та отримання зваженої суми значень пікселів. Таким чином, ядро діятиме подібно до поляризованої лінзи, підкреслюючи або розголошуючи певні шаблони інформації.

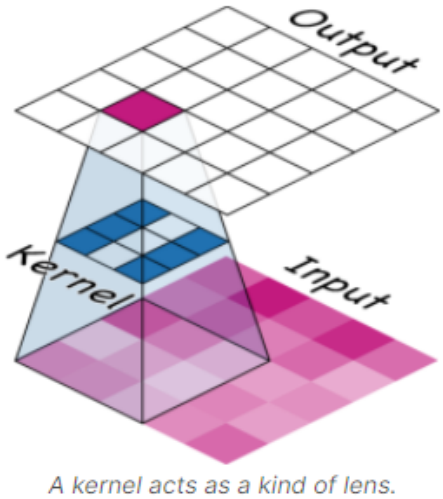


Рис. 14 Ілюстрація сканування зображення ядром

Ядра визначають, як згортковий шар підключений до шару, що йде далі. Ядро вище зв’яже кожен нейрон на виході з дев’ятьма нейронами на вході. Встановивши розміри ядер за допомогою kernel_size, ви розповідаєте **c**onvnet, як формувати ці з'єднання. Найчастіше ядро буде мати непарні розміри - наприклад, kernel_size = (3, 3) або (5, 5) - так що один піксель знаходиться в центрі, але це не є вимогою.

Ядра в згортковому шарі визначають, які типи функцій він створює. Під час навчання учасник намагається дізнатися, які особливості йому потрібні для вирішення проблеми класифікації. Це означає знайти найкращі значення для його ядер.

Активації в мережі називаються функціональними картами. Вони є результатом, коли застосовується фільтр до зображення; вони містять візуальні особливості, які виділяє ядро. Ось декілька ядер, зображених на картах функцій, які вони створили.

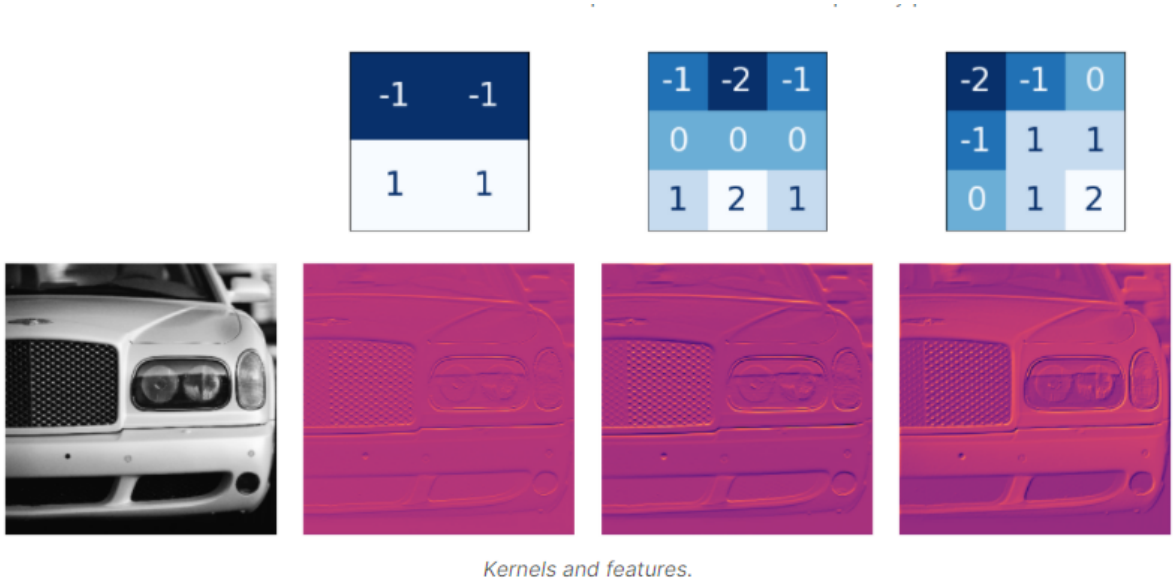


Рис. 15 Ілюстрація фільтрації зображення різними ядрами

За зразком чисел у ядрі можна визначити, які типи функціональних карт створюються. Як правило, те, що підкреслює згортка у своїх вихідних даних, буде відповідати формі позитивних чисел у ядрі. Ліве та середнє ядра вгорі

фільтрують горизонтальні фігури.

За допомогою параметра фільтри ви повідомляєте згортковому шару, скільки карт функцій ви хочете, щоб він створив як вихідні дані.

Виявляємо за допомогою ReLU

Після фільтрування карти функцій проходять через функцію поширення. Наприклад випрямляча лінійна функція (Рис. 4)

Нейрон з приєднаним випрямлячем називається випрямленою лінійною одиницею.

Активацію ReLU можна визначити у власному рівні активації, але найчастіше просто включаєте її як функцію активації Conv2D.

```
model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3, activation='relu')
    # More layers follow
])
```

Можна подумати про функцію активації як оцінку значень пікселів відповідно до певної міри важливості. Активація ReLU говорить, що від'ємні значення не важливі, і тому встановлює їх на 0. ("Все, що не важливо, однаково не важливо".)

Ось ReLU застосував карти функцій вище. Зверніть увагу, як йому вдається відокремити особливості.

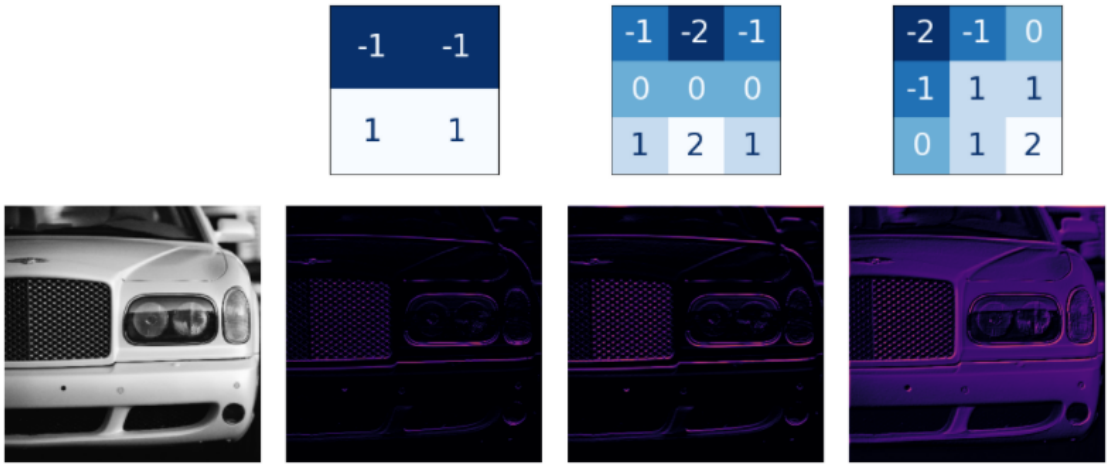


Рис. 16 Ілюстрація результату роботи функції ReLU

Як і інші функції активації, функція ReLU є нелінійною. По суті, це означає, що загальний ефект усіх шарів у мережі стає іншим, ніж той, можна отримати, просто додавши ефекти разом - що буде однаковим, якби ви могли досягти лише одним шаром. Не лінійність гарантує, що функції будуть поєднуватися цікавими способами, коли вони заглиблюються в мережу.

Condense with Maximum Pooling

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3), # activation is None
    layers.MaxPool2D(pool_size=2),
    # More layers follow
])
```

Шар MaxPool2D дуже схожий на рівень Conv2D, за винятком того, що він використовує просту максимальну функцію замість ядра, а параметр pool_size аналогічний kernel_size. Однак шар MaxPool2D немає тренувальних ваг, як згортковий шар у своєму ядрі.

Якщо ще раз розглянути Рис.12, то можна побачити, що після застосування функції ReLU (Виявлення/поширення) карта об'єктів закінчується великою кількістю "мертвого простору", тобто великими областями, що містять лише 0 (чорні області на зображенні). Необхідність проведення цих 0 активацій по всій мережі збільшила б розмір моделі без додавання багато корисної інформації. Натомість потрібно стиснути карту об’єктів, щоб зберегти лише найкориснішу частину - саму функцію.

Насправді це те, що робить Maximum Pooling. Максимальне об’єднання приймає патч активацій на оригінальній карті об’єктів і замінює їх максимальним активуванням у цьому патчі.

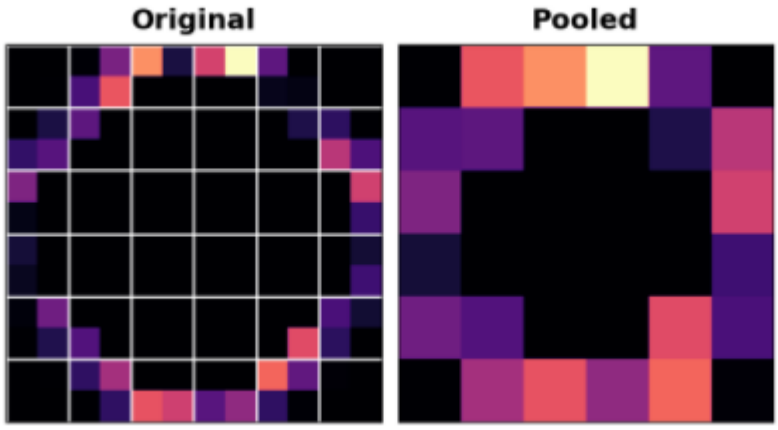


Рис. 17 Ілюстрація роботи об’єднання (Condense with Maximum Pooling)

Застосовуваний після активації ReLU, він має ефект "посилення" ознак (features). Крок об'єднання збільшує частку активних пікселів до нуля пікселів. Наступний крок - треба прикріпити голову.

Додавання голови (Attach Head). Далі треба прикріпити голову класифікатора. У цьому прикладі будемо використовувати шар прихованих одиниць (перший щільний шар), за яким слідує шар, щоб перетворити результати на оцінку ймовірності для класу 1. Шар Flatten перетворює двовимірні виходи основи в одновимірні входи, необхідні голові.

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    pretrained_base,
    layers.Flatten(),
    layers.Dense(6, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])
```

Pretrained_base - це натренирована база (base)

Нарешті, навчимо модель. Для бінарної класифікації будемо використовувати бінарну версію функцію витрат crossentropy та точності. Оптимізатор adam, як правило, працює добре, тому ми також його виберемо.

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)

history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=30,
    verbose=0,
)
```

ds_train - загружений набір даних

РОЗДІЛ 3

Програмна реалізація завдання

3.1. Етапи роботи програми

До них відносять:

1. Читання зображення
2. Знаходження всіх осіб на зображення
3. Обрізання кожної особи
4. Зміна розміру зображення для стандартизації розміру для моделі класифікації зображення
5. Класифікація зображення (й)
6. Обрамлення особи в квадрат залежно від умови
7. Збереження зображення.

Перед написання програми необхідно було написати модель класифікації особи: в масці людина, чи ні.

Для початку потрібно знайти набір даних на якому буде будуватися модель. Було знайдено 12 тис розмічених прикладів на ресурсі kaggle: kaggle.com/ashishjangra27/face-mask-12k-images-dataset

Імпорт необхідних бібліотек

```
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.layers.experimental.preprocessing as preprocessing

# Imports
import os, warnings
import matplotlib.pyplot as plt
from matplotlib import gridspec

import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
```

Налаштування відтворюваності моделі і бібліотеки для відображення графіків:

```
# Reproducibility
def set_seed(seed=31415):
    np.random.seed(seed)
```

```
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
set_seed()

# Set Matplotlib defaults
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)
plt.rc('image', cmap='magma')
warnings.filterwarnings("ignore") # to clean up output cells
```

Читання розмічених даних:

```
from tensorflow.keras import layers, callbacks

early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimum amount of change to count as an improvement
    patience=5, # how many epochs to wait before stopping
    restore_best_weightsa=True,
)
optimizer = tf.keras.optimizers.Adam(epsilon=0.01)
model.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)
history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=15,
)
# Plot learning curves
import pandas as pd
history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot()
history_frame.loc[:, ['binary_accuracy', 'val_binary_accuracy']].plot();
```

Створюємо послідовність шарів:

```
from tensorflow import keras
from tensorflow.keras import layers

pretrained_base = tf.keras.models.load_model(
    '../input/cv-course-models/cv-course-models/vgg16-pretrained-base',
)
pretrained_base.trainable = False

model = keras.Sequential([
    layers.InputLayer(input_shape=[128, 128, 3]),

#     # Data Augmentation
#     preprocessing.RandomContrast(factor=0.10),
```

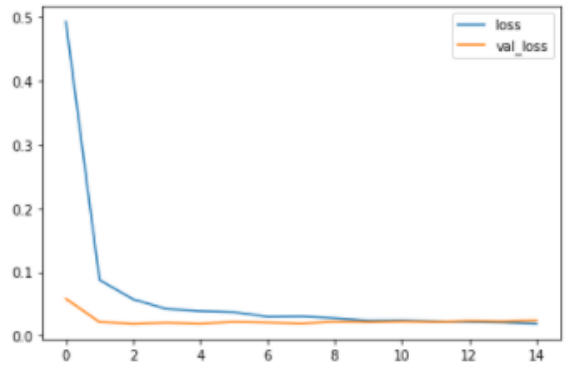
```
#     preprocessing.RandomFlip(mode='horizontal'),
#     preprocessing.RandomRotation(factor=0.10),

    # Base
    pretrained_base,

    # Head
    layers.BatchNormalization(renorm=True),
    layers.Flatten(),
    layers.Dense(8, activation='relu'),
    layers.Dropout(rate=0.3),
    layers.Dense(1, activation='sigmoid'),
])
```

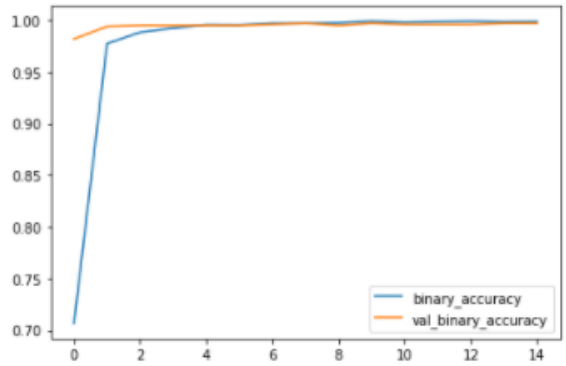
Задаємо функцію втрат, оптимізатор і тренуємо модель:

```
from tensorflow.keras import layers, callbacks

early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimum amount of change to count as an improvement
    patience=5, # how many epochs to wait before stopping
    restore_best_weights=True,
)
optimizer = tf.keras.optimizers.Adam(epsilon=0.01)
model.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)
history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=15,
)
# Plot learning curves
import pandas as pd
history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot()
history_frame.loc[:, ['binary_accuracy', 'val_binary_accuracy']].plot();
```

Процес навчання:

```
Epoch 1/15
157/157 [==============================] - 100s 581ms/step - loss: 0.6757 - binary_accuracy:
0.5867 - val_loss: 0.0582 - val_binary_accuracy: 0.9819
Epoch 2/15
157/157 [==============================] - 18s 112ms/step - loss: 0.1146 - binary_accuracy:
0.9698 - val_loss: 0.0216 - val_binary_accuracy: 0.9940
Epoch 3/15
157/157 [==============================] - 17s 111ms/step - loss: 0.0656 - binary_accuracy:
0.9858 - val_loss: 0.0188 - val_binary_accuracy: 0.9950
Epoch 4/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0486 - binary_accuracy:
0.9903 - val_loss: 0.0204 - val_binary_accuracy: 0.9950
Epoch 5/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0385 - binary_accuracy:
0.9950 - val_loss: 0.0190 - val_binary_accuracy: 0.9950
Epoch 6/15
157/157 [==============================] - 17s 111ms/step - loss: 0.0387 - binary_accuracy:
0.9960 - val_loss: 0.0218 - val_binary_accuracy: 0.9950
Epoch 7/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0313 - binary_accuracy:
0.9964 - val_loss: 0.0207 - val_binary_accuracy: 0.9960
Epoch 8/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0312 - binary_accuracy:
0.9964 - val_loss: 0.0192 - val_binary_accuracy: 0.9970
Epoch 9/15
157/157 [==============================] - 17s 111ms/step - loss: 0.0287 - binary_accuracy:
0.9976 - val_loss: 0.0221 - val_binary_accuracy: 0.9950
Epoch 10/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0227 - binary_accuracy:
0.9993 - val_loss: 0.0213 - val_binary_accuracy: 0.9970
Epoch 11/15
157/157 [==============================] - 17s 111ms/step - loss: 0.0264 - binary_accuracy:
0.9983 - val_loss: 0.0225 - val_binary_accuracy: 0.9960
Epoch 12/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0236 - binary_accuracy:
0.9986 - val_loss: 0.0218 - val_binary_accuracy: 0.9960
Epoch 13/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0226 - binary_accuracy:
0.9990 - val_loss: 0.0235 - val_binary_accuracy: 0.9960
Epoch 14/15
157/157 [==============================] - 17s 111ms/step - loss: 0.0227 - binary_accuracy:
0.9982 - val_loss: 0.0227 - val_binary_accuracy: 0.9970
Epoch 15/15
157/157 [==============================] - 18s 112ms/step - loss: 0.0185 - binary_accuracy:
0.9992 - val_loss: 0.0243 - val_binary_accuracy: 0.9970
```





Зберігаємо модель:

```
model.save('./kaggle/output/model2')
```

Результатом роботи вважається те, що модель досить швидко навчилася класифікувати особи на: "особа в масці" або "особа без маски".

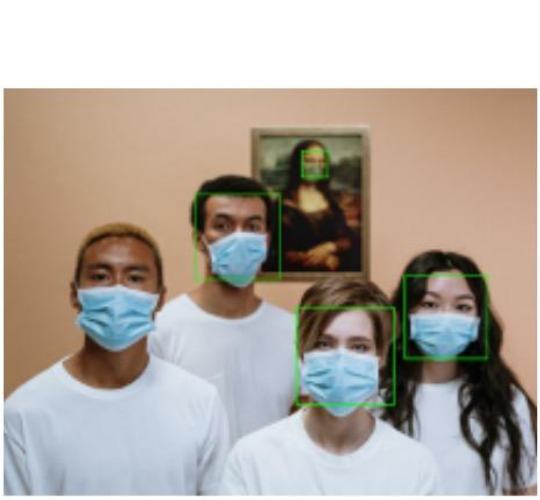
Тепер пишемо програму, яка використовуючи модель розпізнавання особи на зображенні і модель класифікації особи - завантажує зображення і малює над особами квадрат залежно від того "в масці" воно або "без".

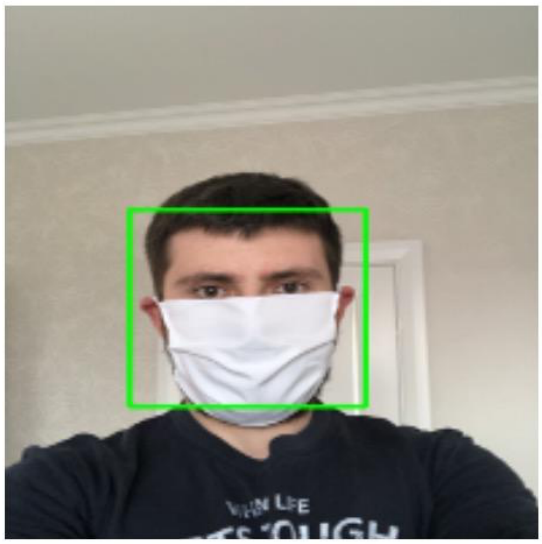
=

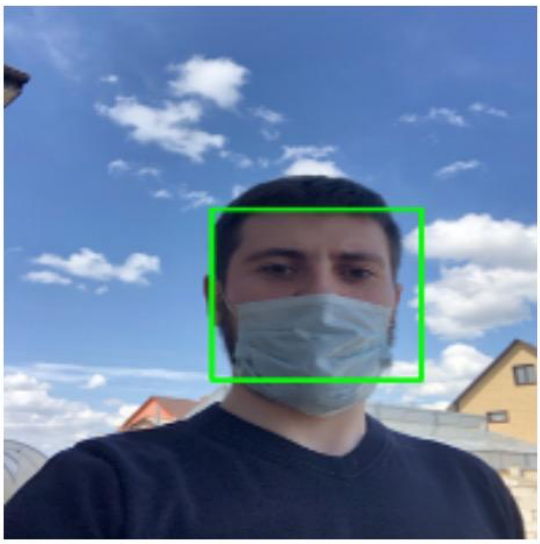
```
import cv2 as cv
import numpy as np
import os
import tensorflow as tf
from tensorflow import keras

# Читаем созданную ранее модель классификации лица
model = keras.models.load_model("model2")

# Читаем взятую из открытых источников OpenCV модель нахождения лица
haar_cascade = cv.CascadeClassifier('haar_face.xml')

# Чтение изображения
img = cv.imread('photo/cam10.png')
# Переведем изображение в серый оттенок
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Находим все лица на изображении
faces_rect = haar_cascade.detectMultiScale(img, scaleFactor=1.1, minNeighbors=5)
cropped = []
for (x,y,w,h) in faces_rect:
    # Обрезаем найденное лицо
    crop = img[int(y*1.05):int((y+h) * 1), int(x*1.05):int((x+w) * 1)]
    # Меняем размер лица под нашу модель классификации
    resized = cv.resize(crop, (128,128), interpolation=cv.INTER_CUBIC)
    # Меняем тип данных у каждого пикселя на Float32
    converted = tf.image.convert_image_dtype(resized, dtype=tf.float32)
    # Делаем предсказание с помощью нашей модели классификации
    predicted = model.predict(np.array([converted]))
    # В зависимости от результата рисуем красный или зеленый кружочек вокруг лиц
    if predicted[0] > 0.5:
        cv.rectangle(img, (x,y), (x+w, y+h), (0,0,255), thickness=2)
    else:
        cv.rectangle(img, (x,y), (x+w, y+h), (0,255,0), thickness=2)

# Выводим и сохраняем новое изображение
cv.imshow('Image with detected Faces', img)
cv.imwrite('Image with detected Faces10.png', img)

cv.waitKey(0)
```

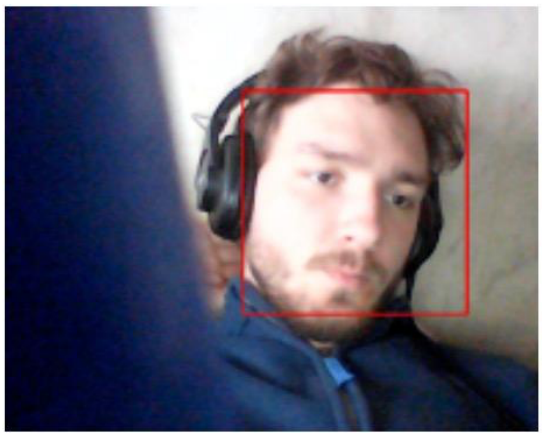
3.2. Демонстрація роботи додатку

Пряме поширення сигналу від вхідного зображення розміром 90x90 пікселів займає 20 мс (на ПК), 3000 мс у мобільному додатку. При детектуванні особи в відеопотоці з роздільною здатністю 640x480 пікселів, можливо детектувати 50 не перекрито областей розміром 90x90 пікселів. Отримані результати з обраної топологією мережі гірше в порівнянні з алгоритмом Віоли-Джонса.









Існують випадки коли спрацьовує невірно одна з моделей:

Отже, модель найкраще працює, коли особа на фотографії велике - одне - і дивиться прямо в камеру.

Згорткові нейронні мережі забезпечують часткову стійкість до змін масштабу, зсувів, поворотам, зміні ракурсу і іншим спотворень.

Ядро - вдає із себе фільтр, який ковзає по всьому зображенню і знаходить ознаки особи в будь-якому його місці (інваріантність до зсувів).

Подвиборочний шар дає:

- збільшення швидкості обчислень (мінімум в 2 рази), за рахунок

зменшення розмірності карт попереднього шару;

- фільтрація вже непотрібних деталей;
- пошук ознак більш високого рівня (для наступного згорткового шару).

Останні шари - шари звичайного багатошарового персептрона. Два повнозв’язних і один вихідний. Цей шар відповідає за класифікацію, з математичної точки зору моделює складну нелінійну функцію, оптимізуючі яку поліпшується якість розпізнавання. Число нейронів в шарі 6 по числу карт ознак подвиборочного шару.

ВИСНОВКИ

Для реалізації поставленої мети, була використана платформа “Kaggle” як середовище, у якому виконувалось розробка моделі. При дослідженні були використані такі бібліотеки як OpenCV та TensorFlow. В даній роботі була створена модель класифікації людського обличчя на предмет присутності на ньому маски. Модель показала результати майже близьку до 100% точності розпізнавання маски на обличчі.

Створений додаток, який використовує цю модель: компьютер знаходить особи на зображенні і після кожне класифікує.

Випадки неправильного спрацьовування можуть бути наступні:

- Програма може знайти особу, але неправильно його класифікувати
- Програма може неправильно знайти обличчя і тоді класифікація буде безглуздою.

Для поліпшення моделі можна додати методи Data Augmentation, які допомагають створити більш різноманітний набір даних. Так як можна помітити неправильному спрацьовуванню моделі у випадках, коли особа в поганому фокусі.

Додаток служе як пример роботи моделі, яку може використовувати для різноманітних задач, таких як:

- Контроль пропуску людей до супермаркету або міського транспорту
- Виявлення осіб без масок у громадських місцях та збереження їх для подальшого опізнання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Intro to Deep Learning - Курс на платформі “Kaggle” [Електронний ресурс].
   – Режим доступу: https://www.kaggle.com/learn/intro-to-deep-learning
2. Computer Vision - Курс на платформі “Kaggle” [Електронний ресурс].
   – Режим доступу: https://www.kaggle.com/learn/computer-vision
3. Курс по машинному навчанню Е.М. Страхова[Електронний ресурс].
   – Режим доступу: https://emstrakhov.com/data_science/ml/
4. Курс по машинному навчанню Andrew Ng [Електронний ресурс].
   – Режим доступу: COURSERA.ORG/LEARN/NEURAL-NETWORKS-DEEP-LEARNING
5. Стаття на інтернет-порталі “Habr” [Електронний ресурс].– Режим доступу: habr.com/ru/post/312450/
6. Стаття про комп'ютерний зір [Електронний ресурс]. – Режим доступу: uk.wikipedia.org/wiki/Комп%27ютерний_зір
7. Стаття на інтернет-порталі “Habr” [Електронний ресурс]. – Режим доступу: https://habr.com/ru/post/461365/