Projekt nr 3

Autorzy: Michał Matuszyk, Sebastian Pergała

Rozpoznawanie obrazów za pomocą komputera

Obrazy można podzielić na dwa typy:

ustrukturyzowane i nieustrukturyzowane.

Obrazy ustrukturyzowane mają budowę określoną przez pewne znane reguły. Często są wykorzystywane do identyfikacji obiektów, do których się odnoszą np. kody QR, kody kreskowe.

Obrazy nieustrukturyzowane nie mają regularnej budowy. Aby określić, co te obrazy reprezentują trzeba je najpierw sklasyfikować. Ten typ obrazów to na przykład zdjęcia otoczenia, grafika komputerowa.

Ustrukturyzowane obrazy

Jednym z ciekawych zastosowań macierzy, są kody QR, czyli Quick Response. Są to dwuwymiarowe kody (znane chyba każdemu), które zawierają informacje w postaci czarno-białych kwadratów. Wykorzystuje się je powszechnie do przechowywania danych, które mogą być szybko odczytane przez smartfony, czytniki kodów QR oraz inne urządzenia.

Celem projektu jest zgłębienie wiedzy na temat kodów QR oraz eksploracja ich potencjału w praktycznych zastosowaniach. Wygenerujmy przykładowy kod QR za pomocą strony (www.qr-code-generator.com)

Przeanalizujmy teraz jego kluczowe komponenty:

1. Znaczniki pozycji – znajdują się na każdym kodzie i pomagają znaleźć orientację kodu

2. Znaczniki orientacji – pomagają określić orientację kodu – pojawiają się w większy kodach

3. Znaczniki wielkości macierzy

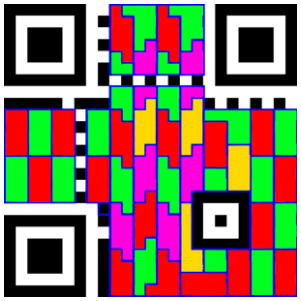
4. Informacje o wersji kodu, jest 40 możliwych, a to jeden z nich:

5. Informacje o tolerancji błędu oraz informacje o masce.

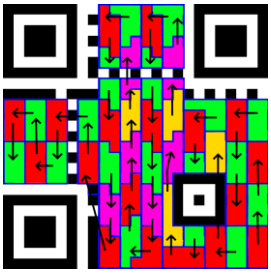
Podsumowując – dużo miejsca jest poświęcone na informacje niezbędne do poprawnego odczytania kodu. Dzięki tym znacznikom, mamy pewność, że zdjęcie będzie odpowiednio obrócone do analizy.


Przez to mamy duże szanse na poprawne odkodowanie danych.

Reszta informacji jest kodowana binarnie, w specjalnych grupach, które zależą od wersji kodu i oczywiście jego rozmiaru, dla naszego kodu grupy wyglądają następująco:


, aby lepiej zobrazować, dodamy kolorów:


Kolejność czytania bitów oraz bajtów (8 bitów) jest określona dla każdego typu i wielkości kodu, w naszym przypadku ciąg bajtów wygląda następująco.



Kolejność bitów w bajcie, jest następująca:

Zaczynamy w prawym dolnym rogu kodu. Czytamy od prawej do lewej, po dotarciu do końca „sektora”: jeśli kolejna strzałka pokazuje do góry, to po dojściu do końca poruszamy się do góry, jeśli kolejna strzałka pokazuje do dołu, to do dołu i znowu czytamy od prawej do lewej.

Z racji, że jest bardzo wiele możliwych kodów – zależy to od wielkości macierzy oraz wersji kodu - nie będziemy rozpatrywać innych kodów. W ten sposób dekodujemy wartość binarną, która może przechowywać dowolne dane. Warto jednak zaznaczyć, że w punkcie powyżej dekodujemy bajty, ale możemy tak naprawdę dekodować ciąg binarny. Trzeba natomiast pamiętać o kolejności kodowanych bitów oraz ich potencjalnym zniekształceniu. Sposobów na zrobienie tego jest wiele, kilka wartych wspomnienia to kod ASCII (UTF-8 jest słabym pomysłem, bo „marnuje" wiele bitów na zbędne informacje), który umożliwia kodowanie 128 znaków. Nic nie stoi w teorii do stworzenia własnego systemu kodowania, w takim wypadku, tych kodów będzie wiele (bardzo wiele), możemy również przechowywać dane liczbowe np. numer walizki na lotnisku, albo aktualną godzinę, żeby synchronizować kamery np. GoPro. Warto również stosować kody korekcyjne/nadmiarowe, gdyż nie zawsze uda nam się zrobić perfekcyjne zdjęcie. Jednym z często używanych kodów jest kodowanie korekcyjne Reeda-Solomona, niestety ich skomplikowana natura, uniemożliwia opisanie ich w kilku stronach, zatem nie opiszemy ich. Mamy nadzieję, że przybliżyliśmy sposób działania kodów QR i teraz, gdy je zobaczymy na ulicy, nie będą dla nas aż tak enigmatyczne.

Nieustrukturyzowane obrazy

Jak komputery rozpoznają obrazy?

W teorii można by napisać zbiór reguł, według których program byłby w stanie określić, co dany zbiór pikseli przedstawia. Takie podejście jednak nie jest realistyczne z uwagi na bardzo dużą liczbę możliwych grafik i znaczącą ilość elementów składających się na pojedynczy obraz.

Z tego względu do komputerowego określania, co znajduje się na nieustrukturyzowanym obrazie, najczęściej wykorzystuje się sztuczną inteligencję, a dokładniej uczenie głębokie. Takie podejście pozwala na stworzenie podstawy działającego programu, bez konieczności jego optymalizacji. Na podstawie dostarczonych danych program jest w stanie sam określać swoje parametry w taki sposób, aby otrzymać żądany wynik. Zanim przejdziemy do sposobu funkcjonowania uczenia głębokiego w rozpoznawaniu obrazów warto jest zrozumieć podział sztucznej inteligencji.

> **Sztuczna inteligencja** (artificial intelligence) to systemy komputerowe zdolne do wykonywania zadań, które normalnie wymagają ludzkiej inteligencji za pomocą modeli i algorytmów symulujących podejmowanie decyzji, uczenie się i rozwiązywanie problemów.
>
> **Uczenie maszynowe** (machine learning) jest jednym z poddziałów sztucznej inteligencji. To modele i algorytmy, które uczą się na podstawie danych. Program trenowany jest na zbiorze danych, aby nauczyć się rozpoznawać wzorce, tworzyć prognozy i podejmować decyzje bez konieczności jawnego programowania.
>
> **Uczenie głębokie** (deep learning) to podpodział uczenia maszynowego, który opiera się na budowie i trenowaniu sieci neuronowych. Sieci neuronowe składają się z wielu warstw neuronów, które są połączone w sposób hierarchiczny. Dzięki temu sieci neuronowe mogą uczyć się reprezentacji cech na różnych poziomach abstrakcji, co umożliwia skuteczne rozpoznawanie i przetwarzanie danych.
>
> **Sieć neuronowa** (neural network) to model matematyczny inspirowany biologicznym układem neuronów w mózgu. Składa się z połączonych ze sobą sztucznych neuronów, które przetwarzają informacje. Sieci neuronowe mogą mieć różne architektury połączeń. W procesie uczenia w sieciach neuronowych dostosowywane są wagi połączeń między neuronami na podstawie danych uczących, aby sieć mogła dokonywać odpowiednich predykcji lub klasyfikacji.

Przyjrzymy się rozpoznawaniu obrazów za pomocą **konwolucyjnych sieci neuronowych**.

Opis działania konwolucyjnej sieci neuronowej (CNN - convolutional neural network)

1) **Warstwa wejściowa - przygotowanie danych (Input layer)**

Na wejściu algorytm otrzymuje obrazy, które chcemy rozpoznać. Aby komputer mógł łatwo operować na grafikach przekształca się je w macierze. Często obrazy przerzuca się na odcienie szarości, ponieważ wtedy dla pojedynczego obrazu mamy pojedynczą macierz. Może się również zdarzyć sytuacja, gdzie chcemy operować na kolorach np. w modelu RGB. Piksel w takiej przestrzeni barw można interpretować jako trójwymiarowy wektor, gdzie pierwsza współrzędna reprezentuje ilość koloru czerwonego i ma wartość z ciała modulo 256 oraz analogicznie druga i trzecia współrzędna określają odpowiednio kolor zielony i niebieski. Każdemu obrazowi w tym modelu odpowiadają trzy macierze, każda zawierająca współrzędne powiązane z jednym z kolorów. Program może operować na bezpośrednio odczytanych wartościach pikseli z grafiki, ale częstą praktyką jest zawarcie tych wartości w przedziale od 0 do 1 lub od -1 do 1.

2) **Warstwy ukryte**

Warstwa konwolucyjna / splotowa (convolutional layer)

Przygotowaną macierz splatamy z pewnym filtrem (kernel). Splatanie polega na nałożeniu na pewien fragment macierzy filtra. Wartości w polach, które się na siebie nakładają, są ze sobą mnożone, a z wyników mnożenia jest obliczana średnia arytmetyczna. Wyniki wyżej opisanej operacji umieszcza się w wynikowej macierzy na miejscu odpowiadającemu położeniu środkowego pola filtra w danej chwili. Współczynniki filtra mogą być określone na starcie, jednak idea uczenia głębokiego sugeruje, że tego typu zmienne mogą być też dostosowywane automatycznie przez komputer na skutek trenowania modelu za pomocą na przykład algorytmu propagacji wstecznej będącej jednym z podstawowych algorytmów uczenia nadzorowanego. Co do parametrów filtra do można w nich wyróżnić:
rozmiar filtra (kernel size, oczywiście filtr powinien być mniejszy niż macierz, na której dokonujemy konwolucji), czyli wymiarów jądra, które z reguły jest kwadratowe (bo kształt nie ma większego znaczenia, a na kwadratowych macierzach się łatwiej operuje),
wypełnienie (padding) – czasem można dodać ramkę wokół obrazu, aby po konwolucji był zachowany pierwotny rozmiar,
kroki (strides) – co ile pikseli następuje splot filtra z macierzą.
Do jednego obrazu można zastosować kilka filtrów na raz i tym samym otrzymać kilka macierzy wynikowych. Czasem w grafice szuka się kilku cech charakterystycznych i wyizolowanie ich za pomocą większej liczby konwolucji pozwala na skuteczniejszą predykcję.

Funkcja aktywacyjna

W celu uzyskania nieliniowości stosuje się na wyniku funkcję aktywacyjną. Nieliniowość jest niezbędna do wytworzenia nieliniowych granic decyzyjnych, tak aby wynik nie mógł być zapisany jako liniowa kombinacja danych wejściowych. Bez tego kroku głębokie sieci CNN mogłyby się przekształcić w pojedynczą równoważną warstwę splotową tym samym obniżając celność jej przewidywań. Do wyboru jest wiele funkcji aktywacyjnych. Najczęściej stosowaną jest ReLU (Rectified Linear Activation) określoną wzorem: $ReLu(x) = \max\{0, x\}$. Jednym z atutów tej właśnie funkcji aktywacyjnej jest jej szybkość działania.

Pooling

W celu zmniejszenia rozmiaru przy zachowaniu większości informacji pozyskanych w poprzednim kroku wykonuje się pooling. Operacja ta polega na podzieleniu macierzy na mniejsze części i zastosowaniu na nich funkcji, która zbierze najważniejsze informacje ze wszystkich pikseli z rozważanego podziału i scali je w jeden piksel. Dla przykładu macierz 8x8 można podzielić na 16 macierze 2x2. Na każdej z tych 16 macierzy wywoła się pewną funkcję redukując rozmiar macierzy z 2x2 do 1x1. W rezultacie macierz 8x8 o 64 polach zmniejszyliśmy do macierzy 4x4 o 16 polach zachowując jej najważniejsze cechy. Funkcję, którą najczęściej się stosuje to scalenia mniejszych macierzy w 1 piksel jest $\max()$, ale czasem się też używa funkcji $\mathrm{avg}()$.

Warstwa konwolucyjna, funkcja aktywacyjna, pooling jako grupa mogą być zamieszczone wielokrotnie w programie. Na wynikowych macierzach otrzymanych po tych krokach można te operacje zastosować jeszcze używając tym razem innych filtrów, aby wychwycić kolejne cechy w już znalezionych w ten sposób macierzach. Zazwyczaj kilkukrotne zastosowanie tych warstw znacząco zwiększa skuteczność programu, choć koszt obliczeniowy również znacząco rośnie.

Warstwa spłaszczająca (flatten Layer)

W tej warstwie wynikowe ramki ‘spłaszcza’ się do wektora. Zazwyczaj operacja polega na kolumnowym wpisywaniu wartości z macierzy do pojedynczego długiego wektora. Robi się to w celu łatwiejszego połączenia danych z wyodrębnionymi cechami obrazu wejściowego z neuronami w celu ich klasyfikacji. Najczęściej do klasyfikacji używana jest funkcja $softmax()$, która jako argument przyjmuje wektor.

Warstwa zapobiegająca nadmiernemu dopasowaniu

Trenując program na pewnym zestawie danych można natknąć się na problem, że algorytm zamiast uczyć się radzić sobie z postawionym problemem w ogólny sposób, dostosowuje swoje zmienne tylko do danych treningowych (overfitting). Wynikiem jest program, który dobrze radzi sobie na danych, z którymi miał już do czynienia, ale jego skuteczność znacznie spada, gdy napotka coś nowego.
Jest wiele sposobów walki z tym zjawiskiem jak na przykład regularyzacja wag, metoda wczesnego zakończenia, batch normalization.
Jedną z najczęściej stosowanych metod jest wdrożenie warstwy porzucenia (dropout layer) w architekturę programu. Warstwa porzucenia usuwa połączenia losowo wybranych neuronów z określonym prawdopodobieństwem. Wynikiem tej operacji jest sieć neuronowa, która w pewnym stopniu zachowuje ulepszenia zyskane po trenowaniu, jednak musi niektórych aspektów uczyć się od nowa. Ta metoda jest skuteczna w zapobieganiu sytuacji, gdzie program zamiast rozpoznawać faktyczne cechy obrazu nauczył się klasyfikować szumy.

3) **Warstwa wyjściowa (output layer)**

W tej warstwie każda współrzędna z wektora wynikowego warstwy spłaszczającej jest połączona z możliwymi wynikami. Program porównuje, do której możliwej klasyfikacji wynik algorytmu jest najbardziej podobny na wyjściu daje właśnie taką predykcję.

Trenowanie programu

Algorytm działa według wyżej opisanych zasad, jednak nie wszystko jest w nim z góry sprecyzowane. Filtry wykorzystywane do konwolucji mogą być losowe na początku działania programu oraz wagi, czyli połączenia między kolejnymi neuronami nie są podane. Zadaniem programu jest klasyfikowanie dostarczonych danych i dokonywanie zmian w zmiennych tak, aby kolejna iteracja uzyskała lepszy wynik. Ale jak program może wiedzieć, czy coś należy zmienić? Metody uczenia algorytmu można podzielić na trzy rodzaje:

Uczenie nadzorowane (supervised learning) – polega na porównywaniu wyniku programu z poprawnym wynikiem. W razie niezgodności predykcji ze wzorcem algorytm otrzymuje sygnał, że coś należy zmienić, a gdy odpowiedź jest zgodna, program przestaje być trenowany na dotychczasowych danych, bo już je potrafi poprawnie klasyfikować.

Uczenie nienadzorowane (unsupervised learning) – program nie otrzymuje informacji zwrotnej o swojej sprawności. Samodzielnie musi znajdować regularności w danych i się do nich dostosowywać. Dwie główne metody stosowane w uczeniu nienadzorowanym to analiza składowych głównych oraz analiza skupień. Analiza składowych głównych jest wykorzystywana do zmniejszania wymiarowości danych poprzez odkrywanie i odrzucanie cech które niosą ze sobą najmniej informacji. Analiza skupień jest wykorzystywana w celu grupowania lub segmentowania zestawów danych ze wspólnymi atrybutami w celu ekstrapolacji występujących w nich zależności.

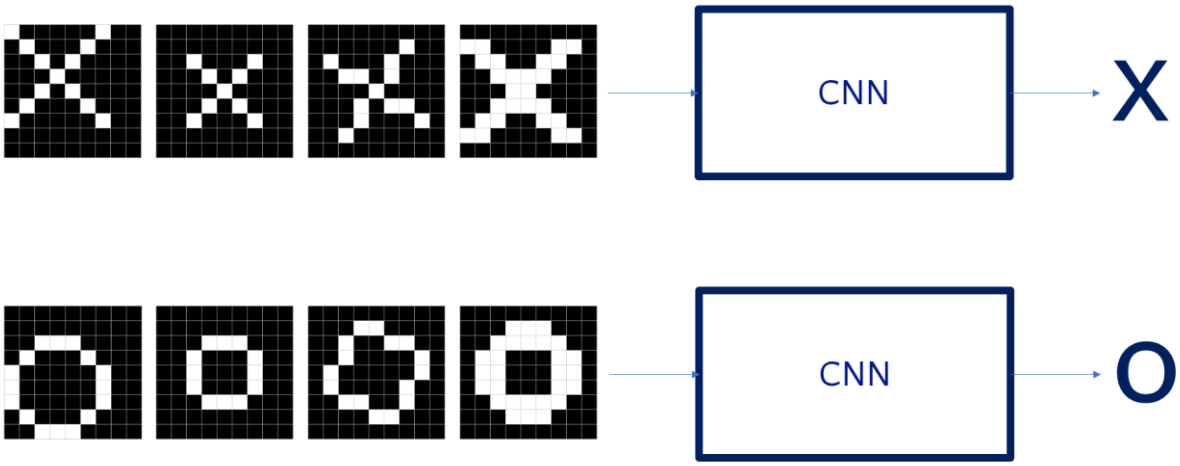
Uczenie przez wzmacnianie (reinforcement learning) - W uczeniu przez wzmacnianie nie przygotowuje się zestawu danych uczących, tylko środowisko, z którego model zbiera dane automatycznie. Jego celem jest zmaksymalizowanie zwracanej przez środowisko nagrody . Większość algorytmów uczenia przez wzmacnianie polega na przygotowaniu polityki, zebraniu za jej pomocą danych, wytrenowaniu jej na ich podstawie i powtarzania tego procesu do osiągnięcia zamierzonego skutku.

Zaprezentowanie działania CNN na przykładzie.

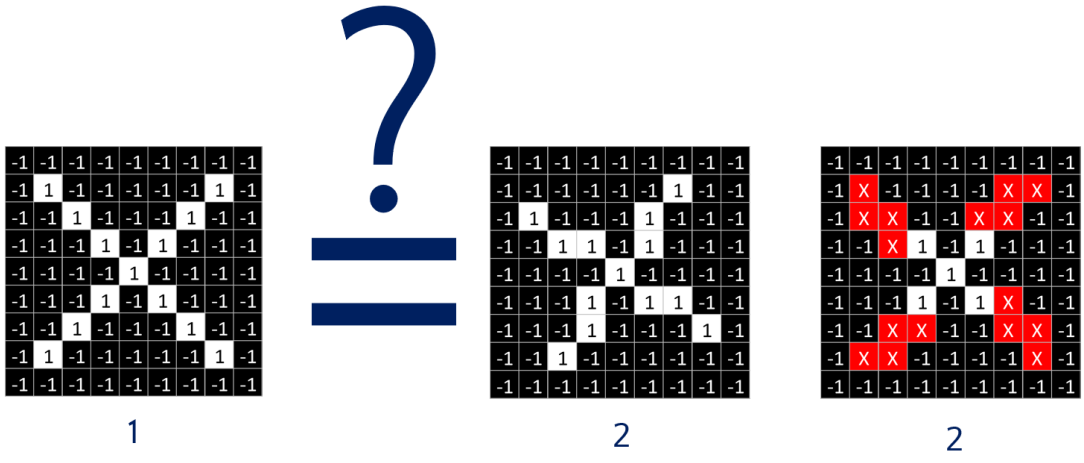
1) **Warstwa wejściowa**

Od programu jest wymagane rozpoznawanie X i O w otrzymanej grafice.

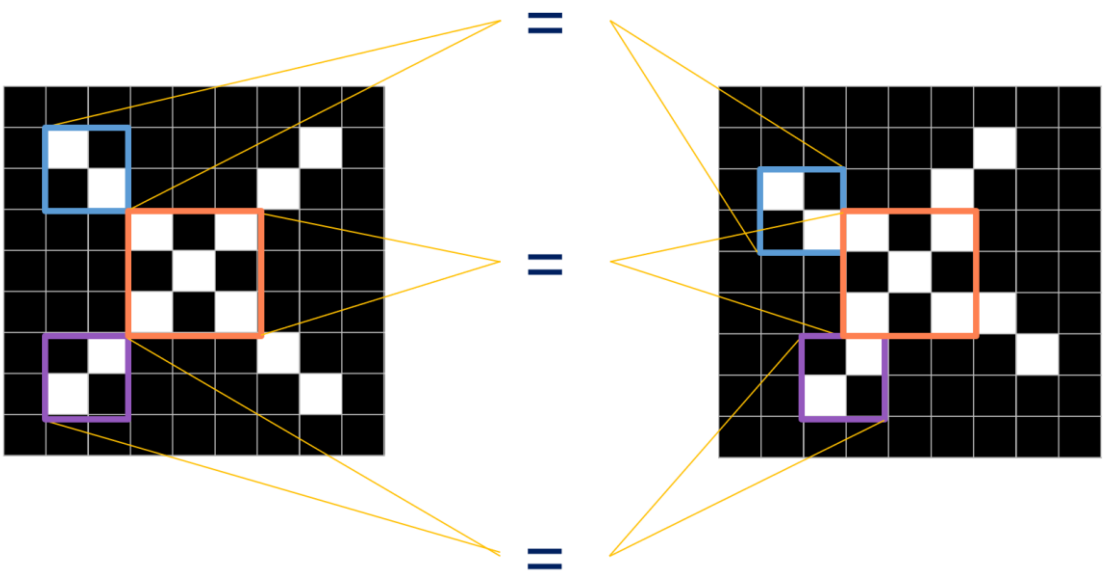
Dane są różne i program musi je poprawnie sklasyfikować znajdując w nich podobieństwa.



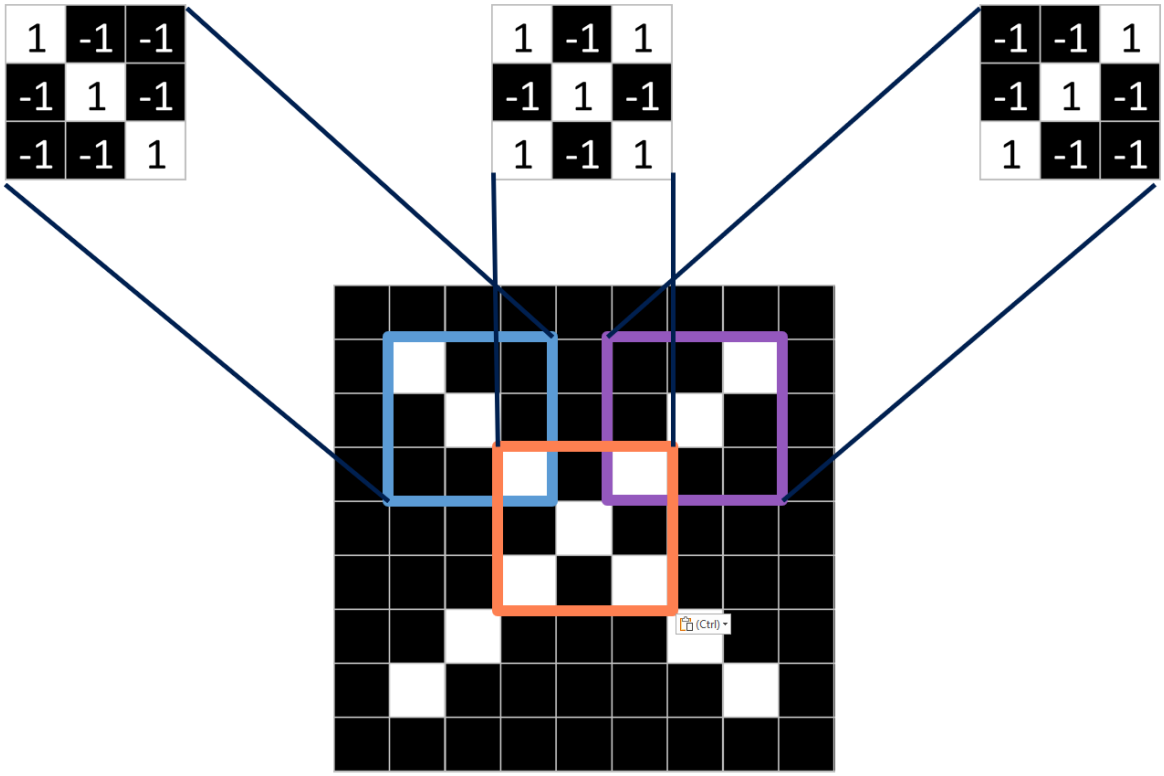
Na obu poniższych grafikach jest symbol X, jednak te obrazy nie są identyczne. (na czerwonym tle są zaznaczone piksele, które w obrazie nr 2 są inne niż w nr 1).



Patrząc na obraz nr 1 i 2 całościowo widać, że różnice są znaczne. Można jednak szukać w grafice cech podobnych i na ich podstawie klasyfikować dane. Każdy symbol „X” ma dwie linie ukośne przecinające się w jednym miejscu, zatem w celu rozpoznania symbolu „X” trzeba znaleźć w grafice te właśnie elementy.



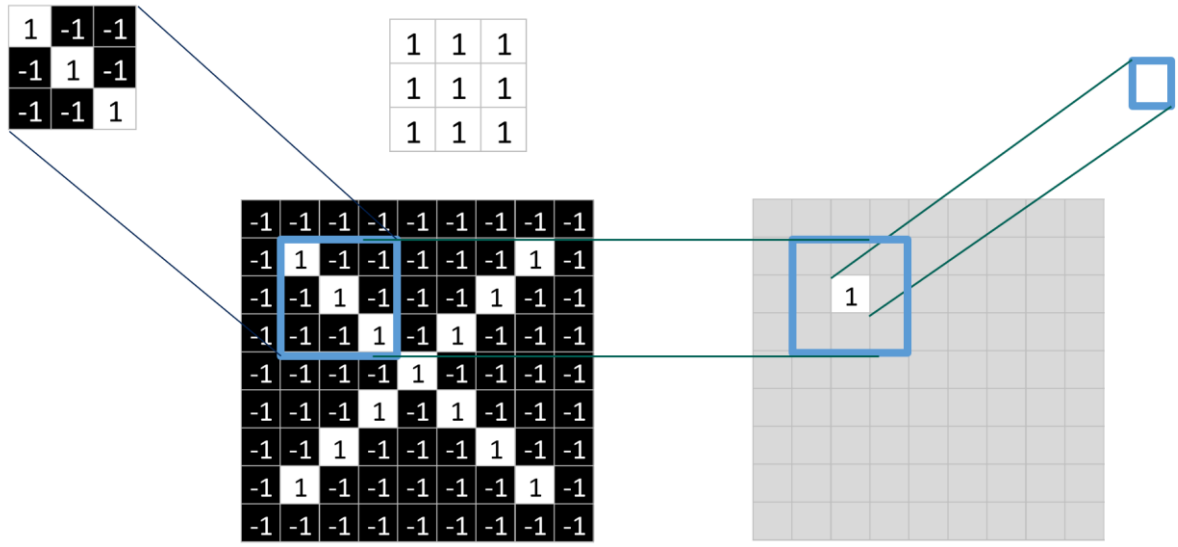
Szukać wcześniej wspomnianych elementów można za pomocą odpowiednich filtrów. Używając konwolucji otrzymamy obrazy, w których szukane cechy będą wyróżnione.

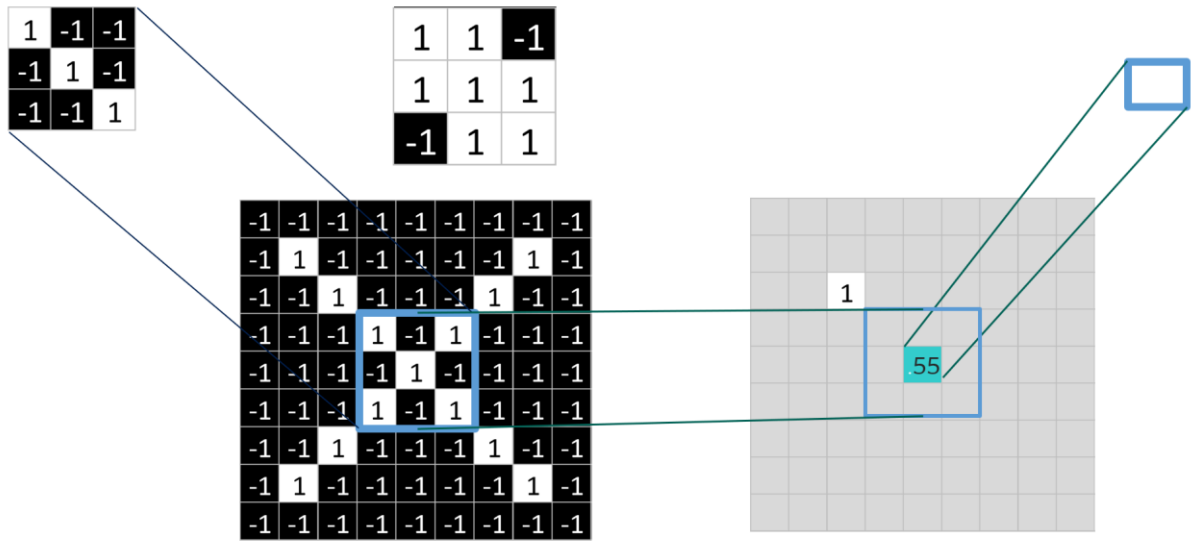


2) Warstwy ukryte

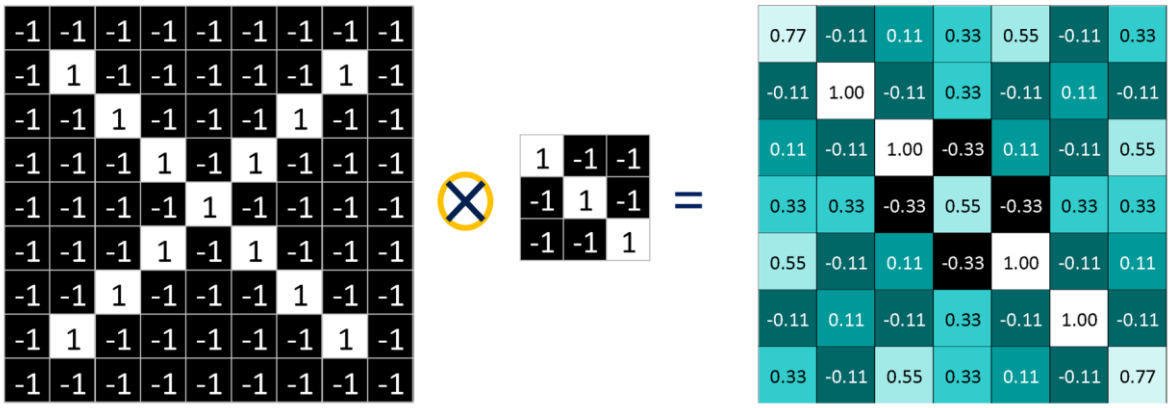
Warstwa konwolucyjna / splotowa

Filtr przedstawiony poniżej wyróżnia linie ukośne biegnące od lewej do prawej w dół. Splatanie polega na nałożeniu na pewien fragment macierzy filtra i wartości w polach, które się na siebie nakładają, są ze sobą mnożone. Obliczając średnią arytmetyczną z wyników mnożenia dostajemy wartość, którą umiejscawiamy w macierzy wynikowej.

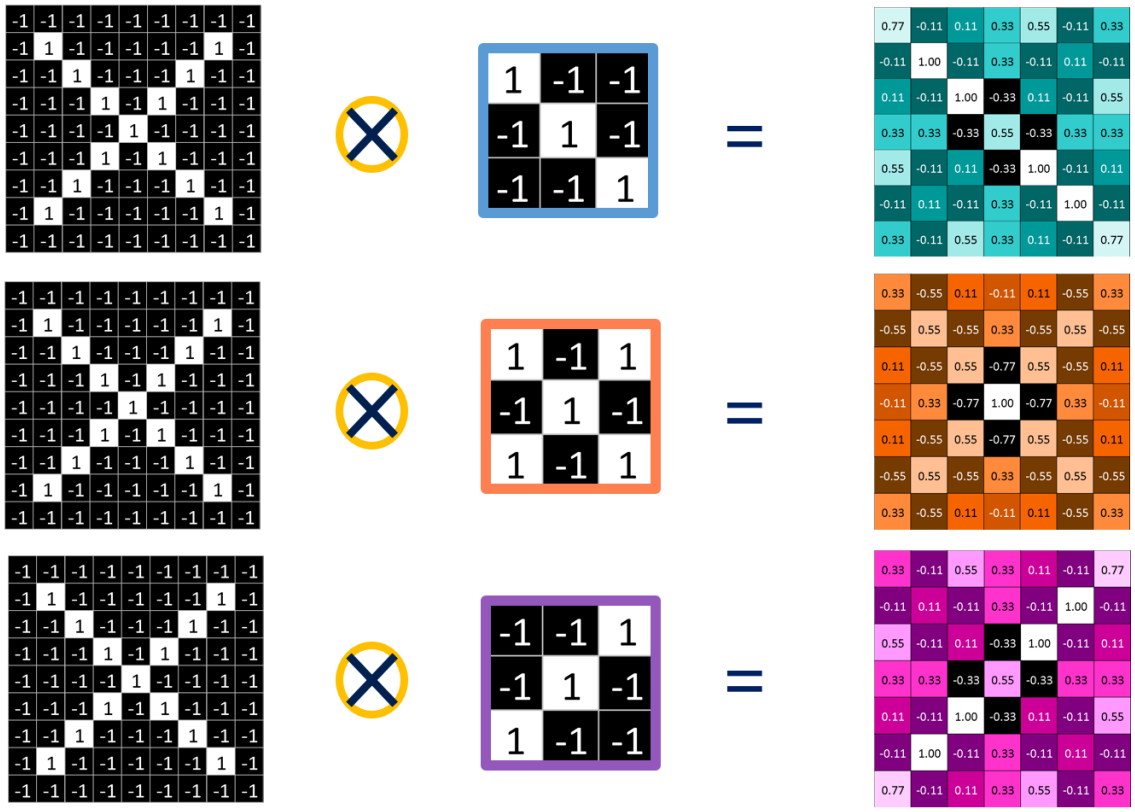




Rozmieszczanie wyników wyżej opisanej operacji odpowiada położeniu środkowego pola filtra w danej chwili. Po przeprowadzeniu konwolucji na całej macierzy otrzymany wynik jest macierzą o wymiarach mniejszych o 1 pole wzdłuż każdej ścianki obrazu.

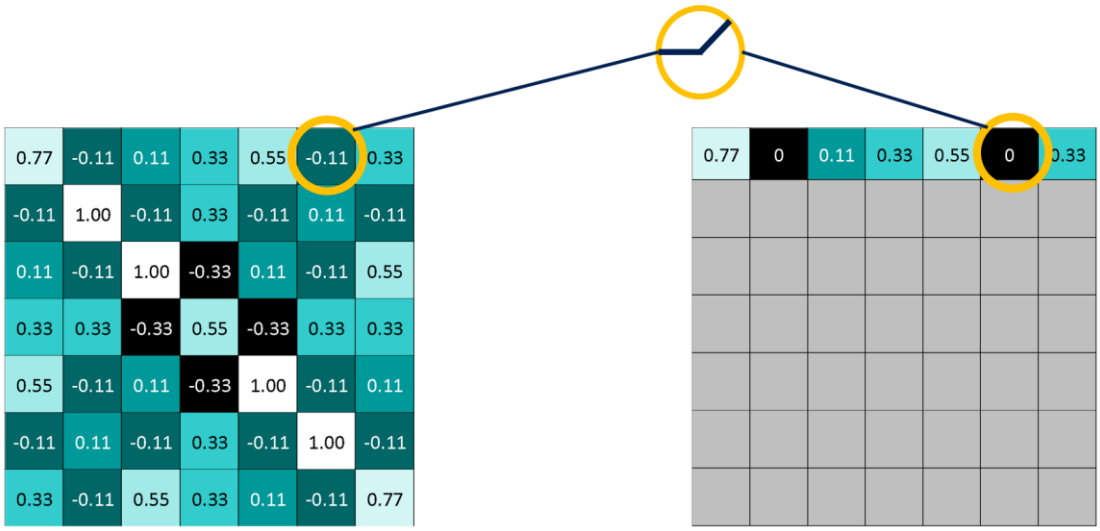


Stosując konwolucję na danej grafice za pomocą różnych filtrów otrzymamy różne macierze wynikowe. Każdy wynik to obraz wyróżniający pewną cechę pierwotnego obrazu.

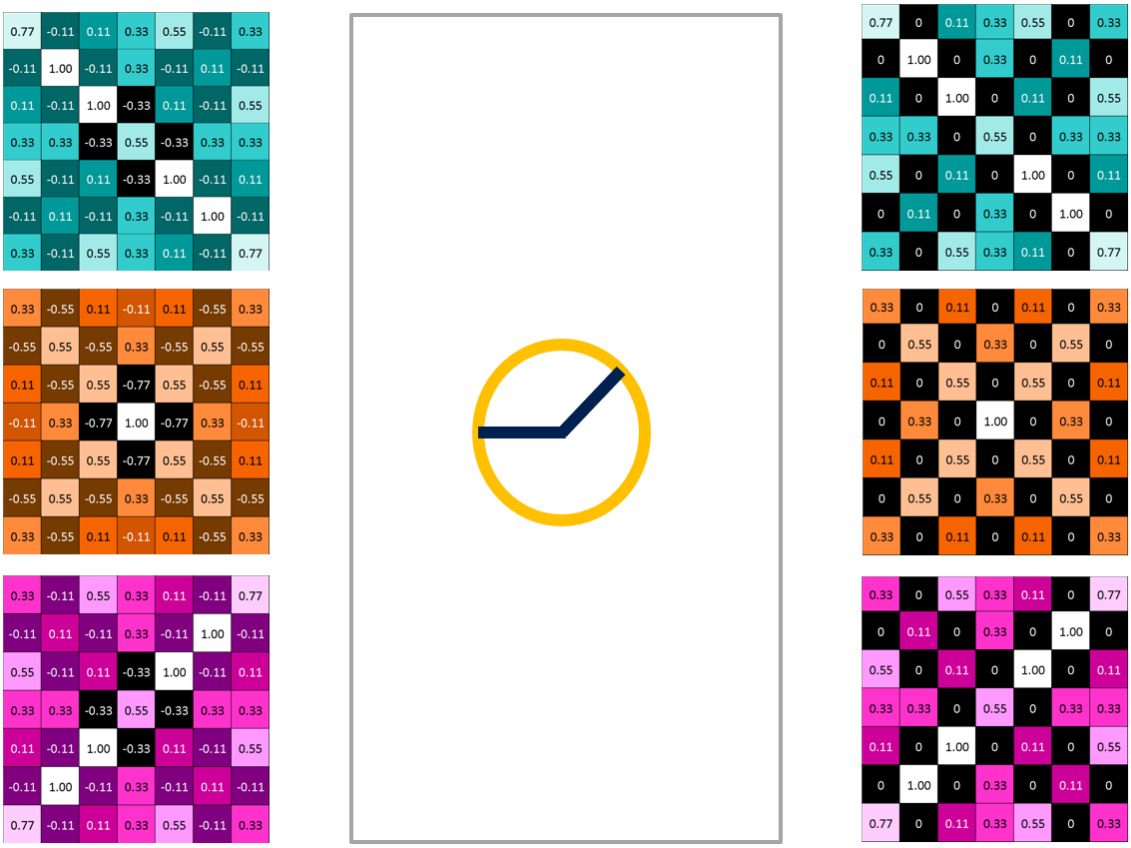


Funkcja aktywacyjna

Na macierzach stosowana jest funkcja aktywacyjna eliminująca linowość z otrzymanych wyników. W tym wypadku używane jest funkcja: $ReLu(x) = \max\{0, x\}$, czyli każda wartość ujemna jest zmieniana na 0.

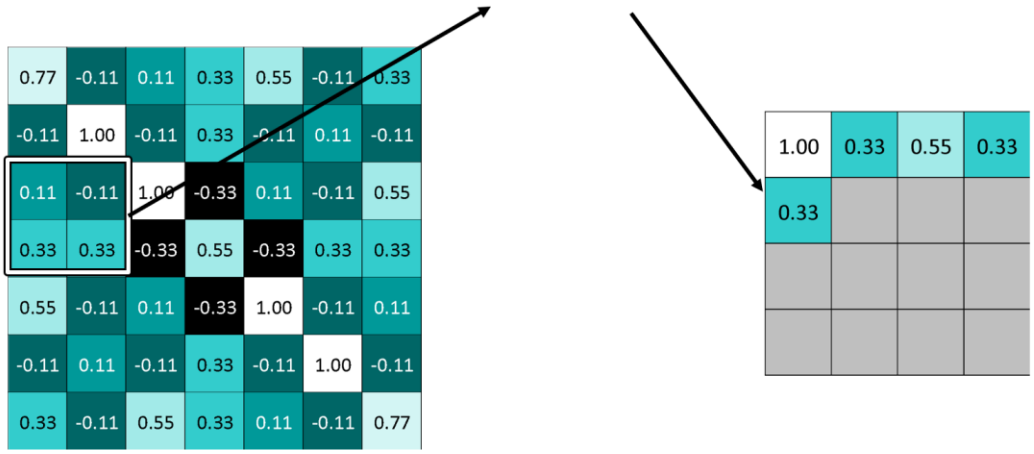


Funkcja aktywacyjna jest stosowana dla wszystkich wyników.

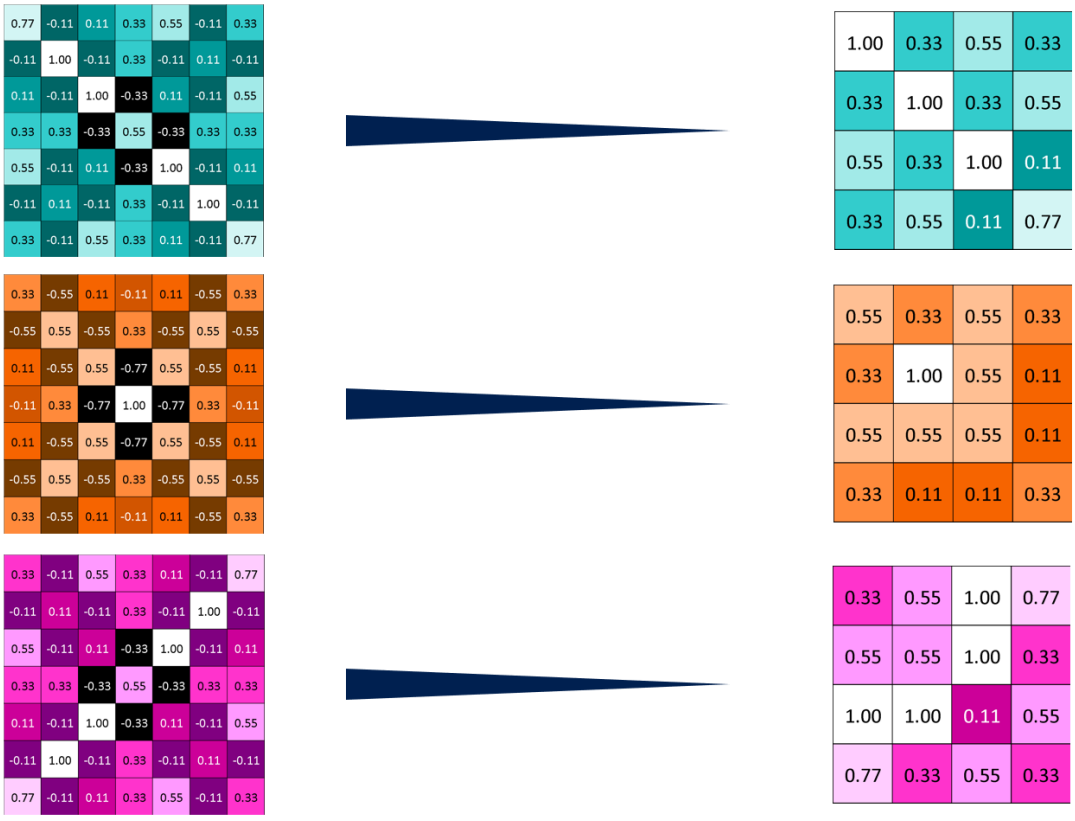


Pooling

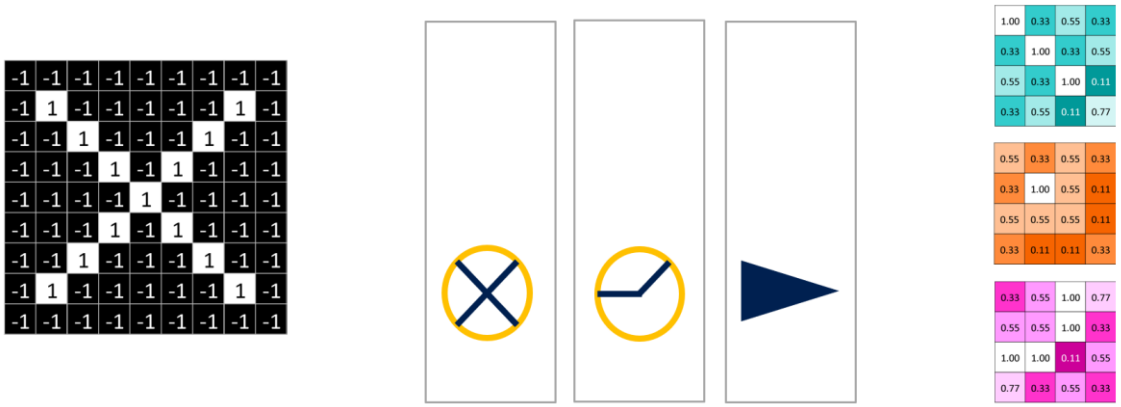
Następnie stosujemy max-pooling dzieląc macierz na segmenty 2x2. Tam, gdzie taki podział jest niemożliwy (ze względu na nieparzyste wymiary macierzy), bierze się największą wartość z obecnych pól. Na przykład w lewym dolnym rogu jest wartość 0.77, a dzieląc macierz na fragmenty 2x2 zaczynając od lewego górnego rogu i idąc po kolei w poziomie rozpatrywane pole będzie jako jedyne należało do swojej grupy ze względu na podział. W takim wypadku funkcja $\max()$ zwróci wartość 0.77.



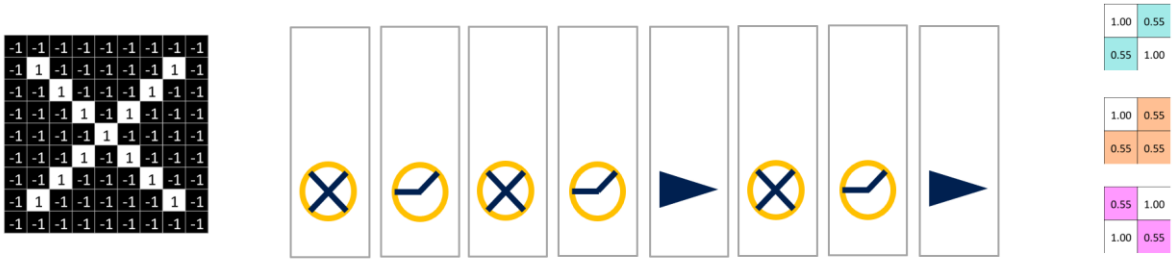
Stosując wyżej opisaną metodę do pozostałych macierzy otrzymujemy macierze mniejszych rozmiarów, w których najbardziej znaczące elementy zostały zachowane.



Warstwa konwolucyjna, funkcja aktywacyjna i pooling mogą być wykorzystane wielokrotnie w programie.

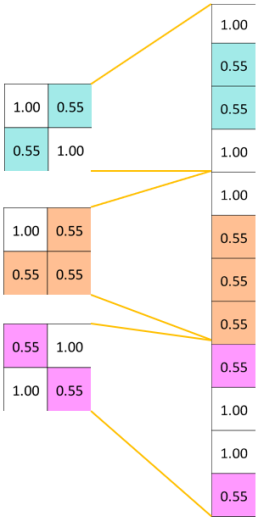


Jako wynik wielokrotnego zastosowania wymienionych warstw wynikowe macierze są małych rozmiarów, a cechy charakterystyczne są w nich bardzo wyróżnione.



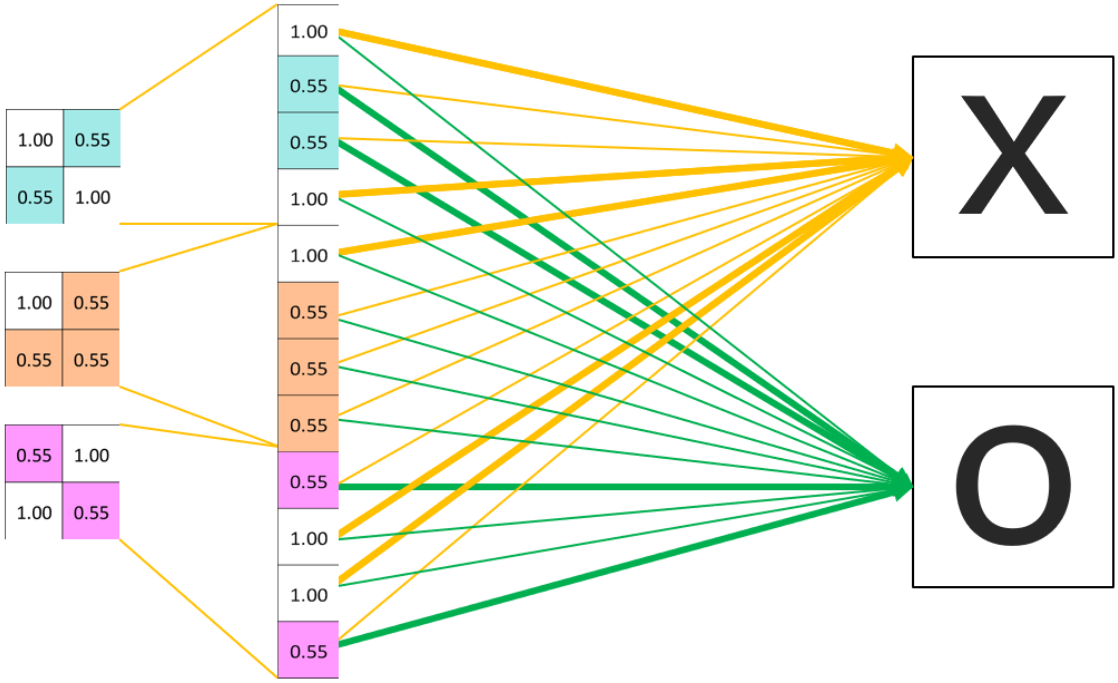
Warstwa spłaszczająca

Kolumny macierzy są przepisywane do pojedynczego wektora.

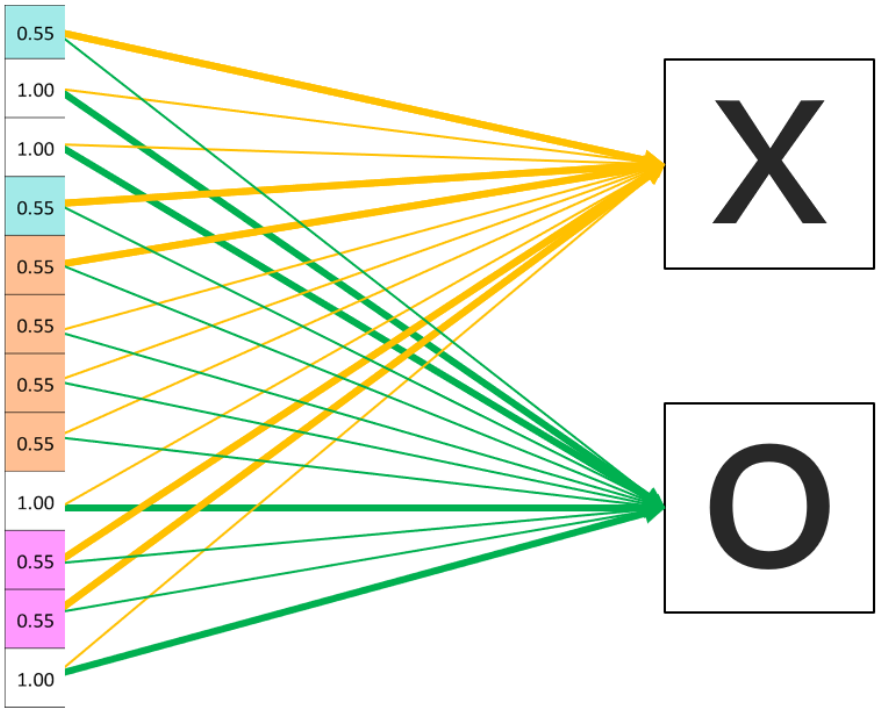


3) Warstwa wyjściowa

Współrzędne wektora wynikowego z warstwy spłaszczającej są łączone z możliwymi wynikami. Program porównuje, do której możliwej klasyfikacji wynik algorytmu jest najbardziej podobny i podaje predykcję. Grubość linii oznacza wartość wag połączeń. Program klasyfikuje obraz jako „X”, ponieważ największe wagi odpowiadają największym wartościom z wektora.



Tutaj dla wcześniej ustalonych wag jest przykład klasyfikacji grafiki jako „O” na podstawie innego wektora z warstwy spłaszczającej.



Czy uczenie głębokie to jedyny sposób?

Uczenie głębokie jest najczęściej stosowanym modelem do rozpoznawania grafiki. Jednak przed wynalezieniem tego typu samouczących się programów też istniały algorytmy pozwalające rozpoznawać obrazy. Aby taki program stworzyć ludzie musieli ręcznie dobierać odpowiednie filtry i ustawiać zmienne. Takie podejście nie jest realistyczne, gdy ma się na celu rozpoznawanie dużej liczby różnych obrazów, ale zastosowane w celu rozróżniania mniejszej grupy obiektów może być skuteczne.

W dalszej części przedstawię program, który w założeniu ma rozpoznawać pisane ręcznie cyfry.

Program rozpoznający cyfry bez uczenia głębokiego

Założenia

Program powinien przyjmować grafikę w formacie .png reprezentującą cyfrę i zwracać cyfrę jej odpowiadającą. Zakładamy, że dane są poprawne, a rozmiar obrazów jest 28x28, cyfry są czarne a białym tle.

Wczytanie danych

Aby łatwo operować na grafice wczytane obrazy będą przekształcone w macierze o wartościach od 0 do 1, gdzie 0 oznacza kolor czarny, a 1 biały. Obrazy są wczytywane za pomocą funkcji:

```
# Wczytanie obrazu
def png_read(filepath):
    # operuje na typie PIL.Image.image
    # zwraca macierz
    img = Image.open(filepath).convert('L')
    return (numpy.array(img)/255).reshape(img.size[1],img.size[0]).tolist()
```

Algorytm ma rozpoznawać czarne pismo na białym tle, ale programy graficzne często zmieniają kolor pikseli na granicy czarnej cyfry z białym tłem w miejscach zagięć cyfr, aby zasymulować gładkie krawędzie. Sposobem na obejście tego problemu jest zastosowanie funkcji, która wartości wszystkich pikseli o wartości większej od danej zamieni na 1, a pozostałe na 0.

```
# zmienia obraz na czarno-biały, piksele o intensywności > p zamienia na 1,
# a resztę na 0
def czarno_bialy(A, p):
    n = len(A)
    m = len(A[0])
    wynik = [[0] * m for i in range(n)]
    for i in range(n):
        for j in range(m):
            if A[i][j] > p:
                wynik[i][j] = 1
    return wynik
```

Jako ulepszenie powyższej funkcji można zaimplementować rozwiązanie, aby parametr p był określany automatycznie ze względu na uśrednioną jasność najjaśniejszego i najciemniejszego piksela. W ten sposób program będzie obsługiwał nie tylko kolory czarny i biały, ale też inne pod warunkiem, że cyfra będzie ciemniejsza od tła. Poprawiona funkcja:

```
# zwraca listę z najmniejszą i największą wartością z macierzy
def min_max(A):
    minimum = A[0][0]
    maksimum = A[0][0]
    for i in range(len(A)):
        for j in range(len(A[0])):
            if A[i][j] > maksimum:
                maksimum = A[i][j]
            if A[i][j] < minimum:
                minimum = A[i][j]
    return [minimum, maksimum]
# tak, jak wcześniej, tylko nie trzeba przekazywać p z zewnątrz
def czarno_bialy(A):
    n = len(A)
    m = len(A[0])
    minimum_maksimum = min_max(A)
    p = (minimum_maksimum[0] + minimum_maksimum[1]) / 2
    wynik = [[0] * m for i in range(n)]
    for i in range(n):
        for j in range(m):
            if A[i][j] > p:
                wynik[i][j] = 1
    return wynik
```

Po przygotowaniu danych należy opracować metodę klasyfikacji danej grafiki jako cyfry. Program będzie wykonywał to zadanie na zasadzie porównywania danego obrazu ze wzorcami. Wzorce mają znaną interpretację, więc algorytm po ustaleniu, do jakiego wzorca grafika jest najbardziej podobna, będzie mógł od razu sklasyfikować grafikę.
Metoda obliczania podobieństwa obrazu do wzorca:
- przechodzimy po każdym pikselu z obrazu,
- dla każdego czarnego piksela z obrazu (czyli stanowiącego część cyfry) liczymy najmniejszą odległość od czarnego piksela ze wzorca w następujący sposób:
    - przechodzimy po każdym pikselu ze wzorca,
    - jeśli piksel jest czarny, to liczymy odległość,
    - zapamiętujemy minimalną odległość,
- mając minimalne odległości od czarnych pikseli ze wzorca dla każdego czarnego piksela z obrazu liczymy średnią arytmetyczną tych odległości,
- powtarzamy tę czynność dla każdego wzorca,
- uznajemy, że obraz odzwierciedla wzorzec, dla którego średnia była najmniejsza.

W powyższej metodzie możemy zastosować różne metryki do obliczania odległości. Dwie najpopularniejsze to euklidesowa i miejska. Po testowaniach doszedłem do wniosku, że poprawność programu jest taka sama dla obu metryk, jednak metryka euklidesowa wymaga większych zasobów obliczeniowych. Ze względu na szybkość działania wybrałem metrykę miejską.

```
def srednia_odleglosc_od_wzorca(pismo, wzorzec):
    rozmiar = len(wzorzec)
    liczba_pikseli = 0
    lista_minimalnych_odleglosci = [0 for i in
        range(len(wzorzec)*len(wzorzec))]
    # biorę każdy piksel z danej macierzy
    for i in range(rozmiar):
        for j in range(rozmiar):
            # porównuję go z każdym pikselem ze wzorca i liczę odległość
            if pismo[i][j] == 0:
                min_odleglosc = 2 * rozmiar
                for i2 in range(rozmiar):
                    for j2 in range(rozmiar):
                        if wzorzec[i2][j2] == 0:
                            odleglosc = abs(i - i2) + abs(j - j2)
                            if min_odleglosc > odleglosc:
                                min_odleglosc = odleglosc
                                if min_odleglosc == 0:
                                    break
                    if min_odleglosc == 0:
                        break
                lista_minimalnych_odleglosci[piksel_z_kolei]=min_odleglosc
                liczba_pikseli += 1
    usredniona_odleglosc = 0
    for i in range(len(lista_minimalnych_odleglosci)):
        usredniona_odleglosc += lista_minimalnych_odleglosci[i]
    return usredniona_odleglosc / liczba_pikseli
```

Program poprawnie odgadywał większość napisanych ręcznie cyfr, jednak cyfry pisane w małym rozmiarze albo przesunięte do rogu często były mylone. Rozwiązaniem tego jest przeskalowanie i umieszczenie cyfry z obrazu na środku grafiki tak, aby cyfra zajmowała maksymalnie dużo miejsca. Nazwijmy ten proces normalizacją. Znormalizowane obrazy i wzorce w ten sposób są rozmieszczone w ramce na takich samych zasadach, a rozciąganie cyfr w osi y pozwala na lepsze dopasowanie grafiki do wzorca. Należy jednak zwrócić uwagę na to, że przy rozciąganiu cyfry „1" w kierunku osi x może dojść niezgodności ze wzorcem. Powodem jest to, że „1" pisane pionowo po rozciągnięciu wypełnia większość ramki, a rozciąganie „1" pisanego skośnie po rozciągnięciu daje bardzo różniący się wynik. Uwzględniając te uwagi funkcja normalizująca obraz wygląda następująco:

```
def wycentrowanie_obrazu(oryginal_img):
    # otrzymuje typ image
    # usuwa z obrazu brzegi, gdzie nie ma cyfry (w osi y)
    # przywraca pierwotny rozmiar obrazu rozciągając go, aby cyfra dotykała
    # ścianek (górnej i dolnej)
    # zwraca typ image
    img = czarno_bialy((numpy.array(oryginal_img) /
        255).reshape(oryginal_img.size[1], oryginal_img.size[0]).tolist())
    rozmiar = len(img)
    pusta_przestrzen_lewo_x = 0
    pusta_przestrzen_prawo_x = 0
    pusta_przestrzen_dol_y = 0
    pusta_przestrzen_gora_y = 0
    for i in range(rozmiar):
        for j in range(rozmiar):
            if img[i][j] == 0:
                if pusta_przestrzen_gora_y == 0:
                    pusta_przestrzen_gora_y = i
                pusta_przestrzen_dol_y = rozmiar - i - 1
                if pusta_przestrzen_lewo_x > j or
                   pusta_przestrzen_lewo_x == 0:
                    pusta_przestrzen_lewo_x = j
                if pusta_przestrzen_prawo_x < j or
                   pusta_przestrzen_prawo_x == 0:
                    pusta_przestrzen_prawo_x = j
    pusta_przestrzen_prawo_x = rozmiar - pusta_przestrzen_prawo_x - 1
    oryginal_img = oryginal_img.crop((0, pusta_przestrzen_gora_y,
                   rozmiar, rozmiar - pusta_przestrzen_dol_y))
    return oryginal_img.resize((rozmiar, rozmiar))
```

Aby łatwo wczytywać obrazy w uwzględnieniem wszystkich powyższych funkcji tak wygląda nowa funkcja wczytująca dane:

```
def png_read_normalised(filepath):
    # operuje na typie PIL.Image.image
    # wyśrodkowuje, przerzuca kolory na czarny i biały
    # zwraca znormalizowaną macierz
    img = Image.open(filepath).convert('L')
    img = wycentrowanie_obrazu(img)
    return czarno_bialy((numpy.array(img) / 255).reshape(img.size[1],
           img.size[0]).tolist())
```

Opis działania

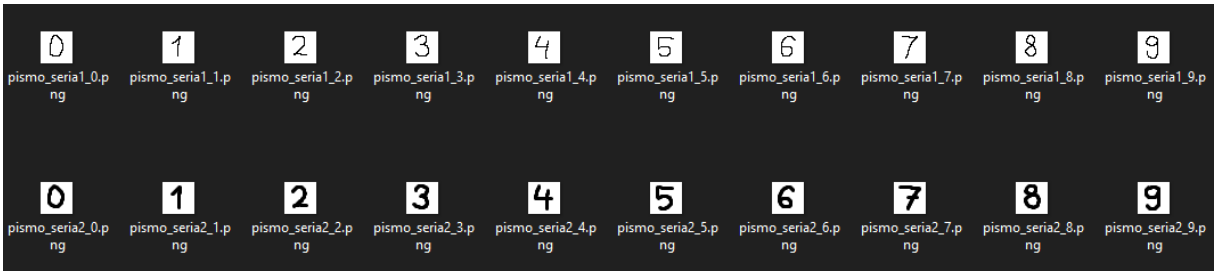
Do listy 1. są wczytywane obrazy z odręcznym pismem, a do 2. wzorce. Następnie każdy obraz z listy 1. jest porównywany z każdym wzorcem według wyżej opisanego algorytmu. Wyniki tych porównań są zapisywane do listy 3., w której na wcześniejszych pozycjach stoją cyfry odpowiadające wzorcom bliższym danemu obrazowi. Następnie obraz jest klasyfikowany do 1 z 10 możliwych grup cyfr według algorytmu K najbliższych sąsiadów (KNN - k nearest neighbours). Klasyfikacja polega na przypisaniu obrazu do określonej grupy na podstawie k pierwszych wyrazów w liście 3.; jeśli wśród tych k pierwszych wyrazów liczność pewnej grupy przewyższa liczność pozostałych, to obraz jest zaliczany do tej grupy, a w przeciwnym przypadku obraz jest zaliczany do grupy na pozycji pierwszej w liście 3. Na koniec program liczy ile predykcji było prawidłowych.

```
# lista zawiera pola, ktore są postaci [obraz, indeks serii, indeks cyfry]
znormalizowane_obrazy_do_testowania =
        [[png_read_normalised(f'./pismo/pismo_seria{j}_{i}.png'), j, i]
        for j in range(1, 3) for i in range(0, 10)]
znormalizowane_wzorce =
        [[png_read_normalised(f'./wzorce/wzorzec_seria{j}_{i}.png'), j, i]
        for j in range(1, 4) for i in range(0, 10)]

dobrze_odgadniete = 0
zle_odgadniete = 0
for pismo in znormalizowane_obrazy_do_testowania:
    predykcje = [] # lista zawierająca [cyfra ze wzorca, odległość od tego
        # wzorca, zgodność ze wzorcem w %]
    minimalna_odleglosc_od_wzorca = float('inf')
    for wzorzec in znormalizowane_wzorce:
        temp = srednia_odleglosc_od_wzorca(pismo[0], wzorzec[0])
        zgodność = round(100 - temp * 100 / 27)
        predykcje.append([wzorzec[2], temp, zgodność])
        if minimalna_odleglosc_od_wzorca > temp:
            minimalna_odleglosc_od_wzorca = temp
            odczytana_cyfra = wzorzec[2]
    predykcje.sort(key=lambda x: x[1])
    # KNN
    k = 3
    # glosowanie = [cyfra, ilość głosów]
    glosowanie = [[i, 0] for i in range(10)]
    for i in range(k):
        glosowanie[predykcje[i][0]][1] += 1
    glosowanie.sort(reverse=True, key=lambda x: x[1])
    # klasyfikacja
    if (glosowanie[0][1] == glosowanie[1][1]):
        klasyfikacja = predykcje[0][0]
    else:
        klasyfikacja = glosowanie[0][0]
    print(f"Obraz z cyfrą {pismo[2]} odczytano jako cyfra
        {klasyfikacja}.\t", end = '')
    if pismo[2] == klasyfikacja:
        dobrze_odgadniete += 1
        print("DOBRZE")
    else:
        zle_odgadniete += 1
        print("ŹLE")
print(f"Odczytano poprawnie {dobrze_odgadniete} z {dobrze_odgadniete +
        zle_odgadniete}, czyli
        {round((dobrze_odgadniete*100)/(dobrze_odgadniete +
        zle_odgadniete))}%")
```

Wynik działania programu dla następujących danych:

Obrazy z pismem odręcznym:



Wzorce:

Wynik dla k = 1:

Obraz z cyfrą 0 odczytano jako cyfra 0. *DOBRZE*
Obraz z cyfrą 1 odczytano jako cyfra 4. *ŹLE*
Obraz z cyfrą 2 odczytano jako cyfra 2. *DOBRZE*
Obraz z cyfrą 3 odczytano jako cyfra 3. *DOBRZE*
Obraz z cyfrą 4 odczytano jako cyfra 9. *ŹLE*
-Obraz z cyfrą 5 odczytano jako cyfra 5. *DOBRZE*
Obraz z cyfrą 6 odczytano jako cyfra 6. *DOBRZE*
Obraz z cyfrą 7 odczytano jako cyfra 7. *DOBRZE*
Obraz z cyfrą 8 odczytano jako cyfra 8. *DOBRZE*
Obraz z cyfrą 9 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 0 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 1 odczytano jako cyfra 4. *ŹLE*
Obraz z cyfrą 2 odczytano jako cyfra 2. *DOBRZE*
Obraz z cyfrą 3 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 4 odczytano jako cyfra 9. *ŹLE*
Obraz z cyfrą 5 odczytano jako cyfra 5. *DOBRZE*
Obraz z cyfrą 6 odczytano jako cyfra 6. *DOBRZE*
Obraz z cyfrą 7 odczytano jako cyfra 7. *DOBRZE*
Obraz z cyfrą 8 odczytano jako cyfra 8. *DOBRZE*
Obraz z cyfrą 9 odczytano jako cyfra 9. *DOBRZE*

Odczytano poprawnie 13 z 20, czyli 65%

Wynik dla k = 3:

Obraz z cyfrą 0 odczytano jako cyfra 0. *DOBRZE*
Obraz z cyfrą 1 odczytano jako cyfra 1. *DOBRZE*
Obraz z cyfrą 2 odczytano jako cyfra 2. *DOBRZE*
Obraz z cyfrą 3 odczytano jako cyfra 3. *DOBRZE*
Obraz z cyfrą 4 odczytano jako cyfra 9. *ŹLE*
Obraz z cyfrą 5 odczytano jako cyfra 5. *DOBRZE*
Obraz z cyfrą 6 odczytano jako cyfra 6. *DOBRZE*
Obraz z cyfrą 7 odczytano jako cyfra 7. *DOBRZE*
Obraz z cyfrą 8 odczytano jako cyfra 8. *DOBRZE*
Obraz z cyfrą 9 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 0 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 1 odczytano jako cyfra 4. *ŹLE*
Obraz z cyfrą 2 odczytano jako cyfra 2. *DOBRZE*
Obraz z cyfrą 3 odczytano jako cyfra 8. *ŹLE*
Obraz z cyfrą 4 odczytano jako cyfra 9. *ŹLE*
Obraz z cyfrą 5 odczytano jako cyfra 5. *DOBRZE*
Obraz z cyfrą 6 odczytano jako cyfra 6. *DOBRZE*
Obraz z cyfrą 7 odczytano jako cyfra 7. *DOBRZE*
Obraz z cyfrą 8 odczytano jako cyfra 8. *DOBRZE*
Obraz z cyfrą 9 odczytano jako cyfra 9. *DOBRZE*
Odczytano poprawnie 14 z 20, czyli 70%

Jak widać skuteczność jest >50%, co oznacza, że program działa. Dla k = 3 skuteczność algorytmu jest większa, niż dla k = 1. Program ma możliwość bycia ulepszanym poprzez większy zbiór wzorców. Stuprocentową skuteczność opisany algorytm ma możliwość osiągnąć w teorii, gdy we wzorcach znajdą się wszystkie możliwe reprezentacje wszystkich cyfr. Takie podejście jest jednak wysoce nierealistyczne.

Źródła

https://www.qr-code-generator.com/blog/how-does-a-qr-code-encode-data/

https://www.ams.org/publicoutreach/feature-column/fc-2013-02

https://www.makeuseof.com/how-to-create-and-decode-a-qr-code-using-python/

https://www.qrcode.com/en/about/error_correction.html

https://pl.wikipedia.org/wiki/Kodowanie_korekcyjne_Reeda-Solomona

https://gopro.github.io/labs/control/precisiontime/

https://www.youtube.com/watch?v=YRhxdVk_sIs

https://www.youtube.com/watch?v=pj9-rr1wDhM

https://www.scs.org.sg/articles/machine-learning-vs-deep-learning

https://docs.google.com/presentation/d/1R-DnrghbU36jO8X4scbrrlx6gFyJHgSL3bD274sutng/edit

https://www.youtube.com/watch?v=JB8T_zN7ZC0

https://miroslawmamczur.pl/jak-dzialaja-konwolucyjne-sieci-neuronowe-cnn/

https://www.youtube.com/watch?v=JboZfxUjLSk

https://www.kaggle.com/general/206846