Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Jakub Prądzyński
nr albumu: 286171
Informatyka

Praca licencjacka

Rozpoznawanie cyfr na obrazach przy użyciu uczenia maszynowego

Opiekun pracy dyplomowej
dr Kamila Barylska

Toruń, 2019

Spis treści

Wstęp

W swojej pracy przedstawię podstawową wiedzę z zakresu uczenia maszynowego. Rozpocznę od wprowadzenia do tematu poprzez przedstawienie historii, definicji, rodzajów oraz najbardziej popularnych algorytmów, kończąc na praktycznych zastosowaniach. W kolejnych rozdziałach wytłumaczę jak działają oraz przedstawię podstawy matematyczne Maszyny Wektorów Nośnych i Sztucznych Sieci Neuronowych jako jednych z najbardziej popularnych modeli uczenia maszynowego. Ostatni rozdział poświęcony jest aplikacji, która zostanie utworzona w ramach niniejszej pracy dyplomowej. Ma ona na celu zademonstrować w jaki sposób można utworzyć pythonowe skrypty uczące modele opisane w rozdziale drugim i trzecim oraz udostępnić, poprzez aplikację webową, prostego GUI do przetestowania tych modeli. Aplikacja wykorzysta otwartą bazę danych MNIST.

Celem tej pracy jest wprowadzenie w tematykę uczenia maszynowego agregując podstawową wiedzę oraz przedstawiając jej wykorzystanie w przykładowym programie. Dzięki temu czytelnik uzyska wiedzę umożliwiającą dalszy rozwój w tematyce sztucznej inteligencji.

Pracę dyplomową oprę głównie na najnowszych artykułach i kursach udostępnianych przez Uczelnię Stanforda, a także na portalu towardsdatascience.com będącym platformą umożliwiającą dzielenie się wiedzą z zakresu Data Science. Dodatkowo wykorzystam prezentacje udostępnione przez różne uniwersytety w Polsce.

1. Wstęp do uczenia maszynowego.

Uczenie maszynowe (ang. *machine learning*) jako dziedzina łącząca informatykę, algorytmikę, statystystykę oraz inne obszary naukowe pojawiła się już około 70 lat temu. Rozwój komputerów pod względem wydajności obliczeniowej oraz przechowywania danych sprawił, że wiedza, rozwijana już od połowy XX wieku, zaczęła być intensywniej wykorzystywana. Coraz częstsze wykorzystanie tej dziedziny w problemach biznesowych, przez popularne firmy, zwiększyło zainteresowanie nią osób nie tylko z branży IT. Punktem zwrotnym, dla wykorzystania uczenia maszynowego w celach komercyjnych, było wydanie przez firmę *Google* 9 listopada 2015r. biblioteki *TensorFlow*, stworzonej przez zespół *Google Brain*, jako produkt open-source, dzięki czemu mniejsze firmy mogły na własną rękę rozpocząć pracę z uczeniem maszynowym, rozwijając i napędzając tę dyscyplinę do dalszego rozwoju.

1.1. Historia

Początków uczenia maszynowego możemy szukać w 1943 roku, kiedy to Warren McCulloch wraz z Walterem Pitts opisali jak mogą działać neurony w ludzkim mózgu oraz zamodelowali prostą sieć neuronową z użyciem obwodów elektrycznych. W 1950 roku możliwości komputerów stały się na tyle zaawansowane, że pojawiła się szansa symulowania domniemanej sieci neuronowej. W tym samym roku Alan Turing stworzył powszechnie znany "test Turinga", za pomocą którego można określić stopień opanowania przez komputer myślenia w sposób podobny do człowieka. W 1959 roku Bernard Widrow wraz z Marcian Hoff z Uniwersytetu Stanforda zbudowali modele zwane "ADALINE" oraz "MADALINE", które odpowiednio potrafiły: rozpoznawać wzorce binarne i przewidywać następny bit (ADALINE) oraz usuwać echo na liniach telefonicznych (MADALINE).

W kolejnych latach następował rozwój uczenia maszynowego. Jednym z ważniejszych momentów dla popularności tej nauki był rok 1997, kiedy Deep Blue - komputer grający w szachy, stworzony przez IBM - wygrał z aktualnym mistrzem świata Garrim Kasparowem w meczu szachowym ze standardową kontrolą czasu. Był to początek dominacji maszyn nad ludźmi w grach strategicznych.

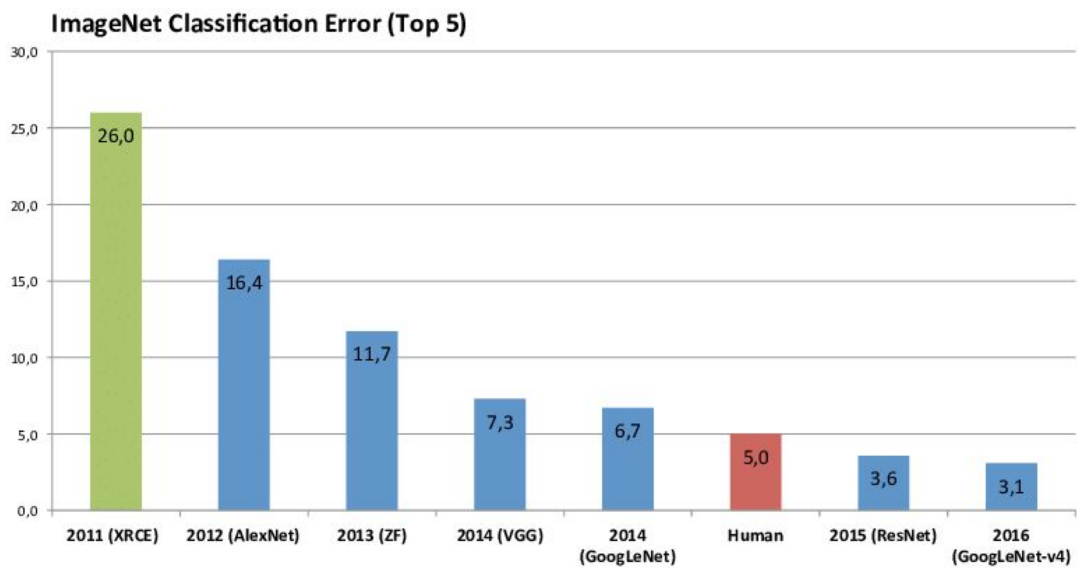
Dalsze badania i rozwijanie tej dziedziny zaczęła napędzać rywalizacja nad tworzeniem coraz lepszych algorytmów na przykład:

- 2006r. Netflix zaoferował 1 milion dolarów za pokonanie ich algorytmu przewidywania ocen filmów.

- 2011r. Superkomputer IBM - Watson - wygrał w trzydniowej rozgrywce teleturnieju Jeopardy (polski odpowiednik Va banque).

- 2012r. Google Brain zamodelował sieć neuronową rozpoznającą twarze na obrazach.

- Powstało wyzwanie ImageNet polegające na stworzenie jak najlepszego algorytmu klasyfikującego obrazy z bazy danych kilkunastu milionów egzemplarzy.



Wykres przedstawia procent błędu, osiągnięty przez kolejne zwycięskie klasyfikatory turnieju ImageNet. [Rys. 1.]

- 2014r. bot czatowy "Eugene Goostman" przeszedł test Turinga.

Kolejnym z przełomowych momentów dla uczenia maszynowego było pokonanie jednego z najlepszych zawodowych graczy Go - Lee Sedola - przez program komputerowy AlphaGo stworzony przez firmę DeepMind. Stało się to w 2016r. czyli około 10 lat wcześniej niż zakładano, że możliwa będzie wygrana z człowiekiem w grze uważanej za najtrudniejszą wśród strategicznych gier planszowych. Wydarzenie to zostało uznane 22 grudnia 2016 roku, przez czasopismo naukowe Science, jako jeden z “przełomów roku”.

1.2. Definicja uczenia maszynowego

W celu zdefiniowania uczenia maszynowego posłużę się słowami określającymi “uczenie się” oraz “uczenie maszynowe” autorstwa odpowiednio dr Susan Ambrose z Northeastern University w Bostonie oraz prof. Arthura Lee Samuela ze Stanford University:

“Uczenie się to proces, pojawiający się w wyniku pewnego doświadczenia, prowadzący do zmiany, zwiększenia potencjału poprawy wydajności oraz przyszłej nauki.”

2010 *From How Learning Works: Seven Research-Based Principles for Smart Teaching*

“Dziedzina wiedzy, która przekazuje komputerom zdolność nauki bez potrzeby zaprogramowania ich wprost.”

1959 *Some Studies in Machine Learning Using the Game of Checkers*.

Zgodnie z dwoma powyższymi definicjami, uczenie maszynowe stanowi dział nauki zajmujący się tworzeniem algorytmów, które można zaimplementować na komputerach, rozwiązujących pewne problemy, poprzez uczenie się rozwiązań w sposób podobny do człowieka, czyli wyciągania wniosków z wydarzeń wcześniejszych.

Sam etap uczenia algorytmów możemy zdefiniować jako dostarczanie mu pewnego zbioru danych, z których będzie on w stanie wywnioskować rzeczy, które pozwolą, z pewną skutecznością, znaleźć rozwiązanie dla nowych danych podanych temu algorytmowi.

1.3. Rodzaje uczenia maszynowego i najczęstsze zastosowania

Algorytmy uczenia maszynowego, w celu rozwiązania pewnego zdefiniowanego problemu, mogą “uczyć się” na 3 sposoby:

1. **Uczenie nadzorowane** - polega na tym, że dostarczane algorytmowi dane, na których ma się uczyć, są oznaczone, czyli dla każdej informacji dostarczanej algorytmowi podany jest również jej wynik.

Uczenie nadzorowane stosuje się najczęściej dla problemów klasyfikacji (rozdzielenia danych do odpowiednich zbiorów) oraz regresji (przewidywania kolejnych wyników poprzez badanie związków pomiędzy danymi).

Przykład: chcemy stworzyć model, który będzie rozpoznawał dany element na obrazach. Aby uzyskać taki efekt musimy, jako dane wejściowe, dostarczyć obrazy, na których zaznaczone będzie występowanie danego elementu (dla lepszej skuteczności algorytmu należy również dostarczać dane nieprawidłowe, czyli zdjęcia na których nie występuje dany element).

2. **Uczenie nienadzorowane** - różni się od uczenia nadzorowanego tym, że nie dostarczamy algorytmowi informacji oznaczonych. Dostaje on pewien zbiór danych, z których ma coś wywnioskować.

Sposób ten możemy stosować do grupowania danych (rozdzielenia danych na zbiory o podobnych cechach) lub do wykrywania anomalii (znajdowania elementów mocno odstających od pozostałych).

Przykład: posiadamy ogromny zbiór danych pogodowych, z których chcemy wydzielić podzbiory.

3. **Uczenie ze wzmocnieniem** - jest w pewnym sensie metodą prób i błędów. Algorytm, który rozpoczyna działanie w zupełnie nieznanym środowisku, musi realizować zadania według jakiejś strategii. Tego schematu będzie uczył się na podstawie wcześniej podjętych decyzji, które są oceniane (nagradzane lub nie). Algorytm będzie dążył do osiągnięcia pewnego stanu poprzez wykonanie zadań najbardziej opłacalnych , czyli najlepiej nagradzanych.

Uczenie ze wzmocnieniem możemy wykorzystać do znajdowania najlepszych schematów rozwiązujących dany problem.

Przykład: nauczenie komputera grania w gry planszowe opiera się na tej metodzie. Nasz komputer wykonuje ruchy w grze (przesuwa pionek) i zgodnie z zasadami zostaje nagradzany lub nie (wygrywa lub przegrywa).

1.4. Najbardziej popularne algorytmy i sposoby reprezentacji wiedzy

- **Regresja liniowa** - metoda polegająca na szukaniu funkcji liniowej, która jak najlepiej opisze zależności między x i y.

- **Regresja logistyczna** - metoda polegająca na szukaniu funkcji logistycznej, która jak najlepiej klasyfikuje binarnie zbiór danych.

- **Liniowa analiza dyskryminacyjna** - jest rozszerzeniem regresji logistycznej o klasyfikację danych na więcej niż dwa zbiory.

- **Drzewa decyzyjne** - metoda polegająca na budowaniu drzew binarnych, w których każdy węzeł reprezentuje decyzję względem pewnej wartości, natomiast liście są wynikiem dokonanych wyborów.

- **Naiwny klasyfikator bayesowski** - bazuje na teorii Bayesa i zakłada, że każda cecha w danej kategorii jest niezwiązana z żadną inną cechą. Wszystkie cechy niezależnie zwiększają prawdopodobieństwo przynależenia do danej kategorii.

- **K-najbliższych sąsiadów** - metoda, dla której modelem zostaje cały zestaw danych treningowych. Polega na znajdowaniu, dla nowej wartości, k jej najbliższych sąsiadów i określaniu jej kategorii za ich pomocą.

- **Learning Vector Quantization** - metoda próbująca udoskonalić algorytm K-najbliższych sąsiadów poprzez ograniczenie danych będących modelem (dla K-najbliższych sąsiadów jest to cały zbiór treningowy) za pomocą sztucznych sieci neuronowych.

- **Support Vector Machines** - klasyfikator binarny, którego działanie polega na wyznaczeniu hiperpłaszczyzny rozdzielającej przykłady należące do dwóch klas z pewnym marginesem.

- **Sztuczne sieci neuronowe** - jest to zbiór matematycznych modeli neuronów, które symulują działanie neuronów znajdujących się w mózgu człowieka. Ich uczenie polega na znajdowaniu wag między poszczególnymi perceptronami, dla których wynik będzie najbardziej odpowiadał rzeczywistości.

1.5. Praktyczne zastosowania

Uczenie maszynowe wykorzystywane jest w takich dziedzinach jak:

- **Bezpieczeństwo danych** - pracownicy firmy Deep Instinct twierdzą, że ich model potrafi rozpoznawać złośliwe oprogramowanie z wysoką skutecznością, przez to, że nowe wersje złośliwego oprogramowania mają tendencję do posiadania tego samego kodu co wersje starsze.

- **Finanse** - dane na giełdzie finansowej zmieniają się w bardzo szybkim tempie, które dla człowieka może być zbyt wymagające, natomiast modele sztucznej inteligencji potrafią coraz lepiej na takich danych operować.

- **Opieka medyczna** - algorytmy uczenia maszynowego potrafią przeprocesować więcej informacji i dostrzec więcej wzorców niż człowiek, dlatego są również przydatne przy wykrywaniu wczesnych stadiów chorób, dla przykładu system Computer-aided diagnosis (CAD) pomaga lekarzom w interpretacji zdjęć medycznych.

- **Personalizacja marketingu** - sztuczna inteligencja może pomóc w doborze produktów, które najbardziej będą odpowiadały klientowi.

- **Wykrywanie przestępstw** - uczenie maszynowe coraz lepiej radzi sobie z wykrywaniem potencjalnych przestępstw. Na przykład, firma PayPal wykorzystuje sztuczną inteligencję do porównywania milionów transakcji i wykrywania nieuczciwych przelewów.

- **Polecanie produktów** - serwisy internetowe już od dłuższego czasu wykorzystują uczenie maszynowe do polecenia produktów klientom. Dobrymi przykładami może tu być Netflix czy Allegro.

- **Wyszukiwanie online** - z wyszukiwarki Google'a korzysta około 95% użytkowników internetu. Wykorzystuje ona uczenie maszynowe w celu dopasowywania najlepszych wyników dla danego hasła wyszukiwania.

- **Przetwarzanie języka naturalnego** - sztuczna inteligencja pozwala na przetwarzanie języka naturalnego, co uzyskuje bardzo dobre efekty między innymi w tłumaczach czasu rzeczywistego.

- **Inteligentne samochody** - tworzenie autonomicznych pojazdów nie byłoby możliwe, gdyby nie uczenie maszynowe. Za jego pomocą powstają specjalne mapy wykorzystywane w samochodach autonomicznych. Kamery będące “oczami” pojazdów przetwarzają obraz za pomocą sztucznej inteligencji.

- **Internet of Things** - internet rzeczy powoli staje się coraz bardziej powszechny, właśnie dzięki sztucznej inteligencji. Inteligentne lodówki, pralki itp. sterowane za pomocą smartfona lub asystenta takiego jak Siri, Alexa czy asystent Google’a są już dziś dostępne.

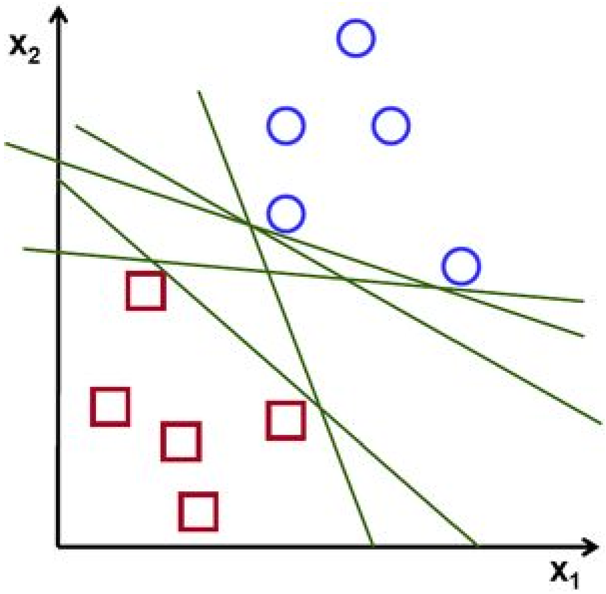
2. Maszyna wektorów nośnych - Support Vector Machine

W XXw. została opracowana teoria z dziedziny statystyki, która stała się podstawą do stworzenia nowego algorytmu uczenia maszynowego - *maszyny wektorów nośnych*. Najczęściej wymienianym współtwórcą jest prof. Vladimir Vapnik, wybitny matematyk, obecnie pracujący w "Facebook AI Research". Maszyna wektorów nośnych jest uważana za jeden z najlepszych algorytmów uczenia nadzorowanego.

2.1. Zasada działania

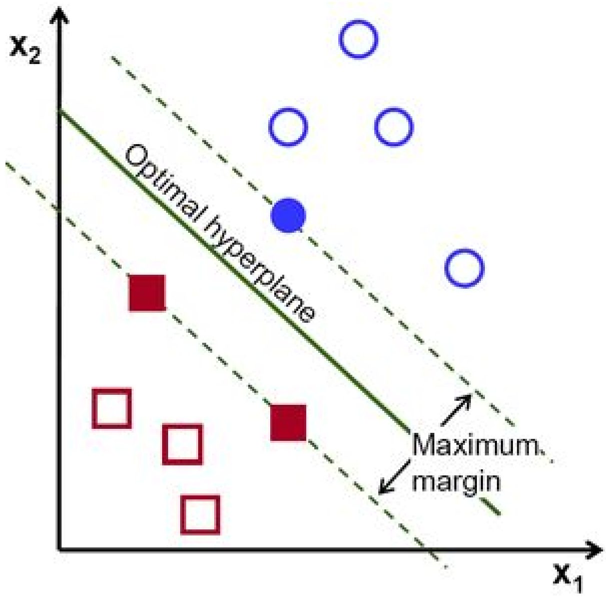
W poniższym podrozdziale opiszemy w zarysie zasadę działania maszyny wektorów nośnych. Szczegółowe definicje przedstawimy w dalszych podrozdziałach. Wymagana jest podstawowa wiedza z zakresu algebry liniowej.

Zadaniem maszyny wektorów nośnych jest znalezienie *hiperpłaszczyzny* w N wymiarowej przestrzeni (gdzie N to liczba "cech" danych wejściowych), która wyraźnie rozdziela jeden zbiór danych od pozostałych. Dla 2 wymiarowej przestrzeni hiperpłaszczyzny są określane poprzez funkcje liniowe co widać na poniższym obrazie:



Przykładowe hiperpłaszczyzny w przestrzeni dwuwymiarowej. [Rys. 2.]

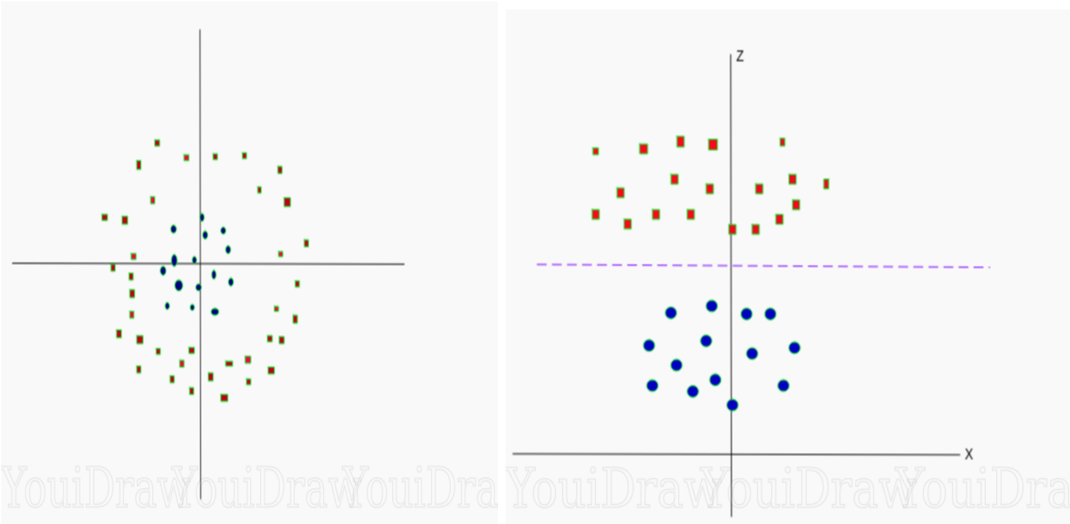
Niebieskie okręgi i czerwone kwadraty to dwa podzbiory zbioru danych wejściowych, które chcemy rozdzielić. Zielone proste to przykładowe hiperpłaszczyzny określające podział tych danych. *Maszyna wektorów nośnych* ma na celu znalezienie najbardziej *optymalnej hiperpłaszczyzny* (ang. *optimal hyperplane*), czyli takiej z *największym marginesem* (ang. *maximum margin*) - maksymalnym dystansem między punktami obu klas a hiperpłaszczyzną. Znalezienie takiej hiperpłaszczyzny daje możliwość klasyfikowania danych z największą pewnością.



Optymalna hiperpłaszczyzna w przestrzeni dwuwymiarowej. [Rys. 3.]

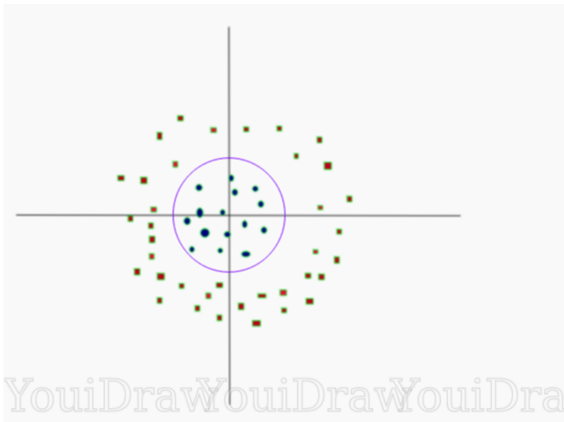
Wektorami wspierającymi są te elementy ze zbioru danych treningowych, które mają wpływ na pozycję i orientację hiperpłaszczyzny. Użycie tych wektorów pozwala na maksymalizowanie marginesu klasyfikatora, a usuwanie zmienia pozycję hiperpłaszczyzny. Działania te pozwalają budować maszynę wektorów nośnych.

Jeżeli zbiór danych o N cechach nie jest *liniowo separowalny* w N wymiarowej przestrzeni, możemy wynieść nasze punkty do wyższego wymiaru, poszukać w nim odpowiedniej hiperpłaszczyzny i przetransformować wynik do wymiaru wejściowego.



Nieliniowo separowalne dane w przestrzenii dwu [Rys. 4.] i trójwymiarowej. [Rys. 5.]

Powyższa ilustracja zawiera przykładowy zbiór danych przestrzeni dwuwymiarowej, który nie jest liniowo separowalny. Wyniesienie go do przestrzeni trójwymiarowej pozwala na liniowy podział na dwie klasy. Po transformacji wyniku do wymiaru wejściowego okazuje się, że szukana hiperpłaszczyzna jest opisywana przez równanie okręgu, co widać na poniższej grafice:



Hiperpłaszczyzna zrzutowana z przestrzenii trójwymiarowej na dwuwymiarową. [Rys. 6.]

2.2. Podstawy matematyczne

2.2.1. Problem klasyfikacji

W N wymiarowej przestrzeni danych Ω mamy próbkę uczącą U (zbiór danych treningowych) elementów zdefiniowanych jako pary (x, y), gdzie x jest pojedynczym wektorem (daną) rzędu N, a y jest elementem zbioru dwuelementowego $\{-1,\ 1\}$, które odpowiadają klasom danych treningowych: 1 - klasa wyznaczona, -1 - pozostałe klasy.

$$U = \{(x_i, y_i) | x_i \in R^N, y_i \in \{1, -1\}\}$$

Na podstawie danych wejściowych U musimy znaleźć klasyfikator - *granicę decyzyjną* $g(x)$ - która podzieli przestrzeń Ω na dwa zbiory oraz będzie z maksymalną skutecznością klasyfikować nowe obiekty x do klas.

2.2.2. Separowalność liniowa

Dwie klasy możemy nazwać liniowo separowalnymi, jeśli istnieje taka hiperpłaszczyzna H opisana wzorem:

$$g(x) = w^T x + b$$

gdzie:

w - wektor wag

x - dana wejściowa

b - polaryzacja, położenie względem początku układu współrzędnych

przyjmująca wartości:

$$\begin{cases} g(x_i) > 0 & x_i \in 1 \\ g(x_i) < 0 & x_i \in -1 \end{cases}$$

2.2.3. Równania decyzyjne

Dla próbki uczącej U przy założeniu liniowej separowalności klas y, równanie hiperpłaszczyzny separującej określone jest następująco:

$$g(x) = w^T x + b = 0$$

Zatem równaniami decyzyjnymi będą:

1. Jeżeli $w^T x + b \geq 0$ wtedy $y = 1$
2. Jeżeli $w^T x + b \leq 0$ wtedy $y = -1$,

co równoważne jest nierówności:
$g(w^T x + b) \geq 1$.

Spełnienie tej nierówności oznacza przynależenie, do wyznaczonej klasy, a zdefiniowane jest przez wektory nośne decydujące o położeniu hiperpłaszczyzny i szerokości marginesu separacji.

Należy więc wyznaczyć b oraz w, aby móc określać przynależność do klasy dla każdego elementu x z ω.

2.2.4. Przekroczenie granicy separacji

Dla problemów *niecałkowicie separowalnych liniowo* może wystąpić sytuacja, w której będziemy potrafili wyznaczyć granicę separacji poprawnie zdefiniowaną dla prawie wszystkich elementów wejściowych (przypadki leżące wewnątrz strefy marginesu separacji).

Sytuację taką możemy zapisać za pomocą nierówności:
$g_i(w^T x_i + b) \geq 1 - \delta_i$
gdzie $\delta_i \geq 0$ będzie wartością zmniejszającą margines separacji.
W takiej sytuacji, jeśli:
$0 \leq \delta_i < 1$ - wtedy para (x_i, y_i) leży po właściwej stronie hiperpłaszczyzny

$\delta_i = 1$ - wtedy para (x_i, y_i) leży na hiperpłaszczyźnie

$\delta_i > 1$ - wtedy para (x_i, y_i) leży po niewłaściwej stronie hiperpłaszczyzny

Należy więc zminimalizować wartość δ_i.

2.2.5. Szerokość marginesu separacji

Odległość wektorów nośnych od hiperpłaszczyzny określona jest następująco:

$$r(x_{sv}) = \frac{g(x_{sv})}{||w||} = \begin{cases} \frac{1}{||w||} & dla \quad g(x_{sv}) = 1 \\ \frac{-1}{||w||} & dla \quad g(x_{sv}) = -1 \end{cases}$$

A więc szerokość marginesu separacji możemy wyznaczyć ze wzoru:

$$\rho = 2 \cdot r(x_{sv}),\ bo\ \rho = (x^+ - x^-) \cdot \frac{w}{||w||} = \frac{2}{||w||}$$

Aby zmaksymalizować margines separacji $\rho = \frac{2}{||w||}$ trzeba zminimalizować $||w||$ co przy pewnych ograniczeniach liniowych, wynikających ze zdefiniowanej nierówności decyzyjnej, równoważne jest minimalizacji wyrażenia $\frac{1}{2}||w||^2$.

2.2.6. Minimalizacja przy zastosowaniu mnożnika Lagrange'a

Dla wyrażenia $\frac{1}{2}||w||^2$ możemy określić problem optymalizacji:

$$min_w \frac{1}{2}||w||^2$$

przy zdefiniowanych ograniczeniach:

$g_i(w^T x_i + b) \geq 1$

Otrzymujemy następującą *funkcję Lagrange'a*:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_i a_i(g_i(w^T x_i + b) - 1)$$

gdzie:

a_i - wektor mnożników Lagrange'a o wartościach nieujemnych

Aby wyznaczyć ekstrema tej funkcji, musimy znaleźć miejsca zerowe jej pochodnej:

$$\frac{\partial L}{\partial w_i} = 0; \ \frac{\partial L}{\partial a_i} = 0$$

Rozwiązując równanie pierwsze otrzymujemy:

$$\frac{\partial L}{\partial w} = \overline{w} - \sum_i a_i g_i \overline{x}_i = 0 \Rightarrow \overline{w} = \sum_i a_i g_i \overline{x}_i = 0$$

Dla drugiego mamy:

$$\frac{\partial L}{\partial b} = -\sum_i a_i g_i = 0 \Rightarrow \sum_i a_i g_i = 0$$

Podstawiając wyliczone $\overline{w}$ oraz wynik drugiego równania do funkcji Lagrange'a otrzymujemy:

$$L = \frac{1}{2}\overline{w} \cdot \overline{w} - \sum_i \alpha_i(g_i({}^T w \cdot x_i + b) - 1) = \frac{1}{2}(\sum_i \alpha_i g_i \overline{x}_i) \cdot (\sum_j \alpha_j g_j \overline{x}_j) -$$

$$(\sum_i \alpha_i g_i \overline{x}_i) * (\sum_j \alpha_j g_j \overline{x}_j) - \sum_i \alpha_i g_i b + \sum_i \alpha_i = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j g_i g_j (\overline{x}_i \cdot \overline{x}_j)$$

2.2.7. Problem dualny

Zgodnie z wykładem dr Andrew Ng ([Andrew Ng. CS229 Lecture Notes, Part 5 (pdf)]) *problemem dualnym* do powyższego jest:

$$max_\alpha W(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j g_i g_j(\overline{x}_i \cdot \overline{x}_j)$$

Przy warunkach:

$$\alpha_i \geq 0 \ i \ \sum_i \alpha_i g_i = 0$$

Maksymalizacja odbywa się tylko po wartościach α

2.2.8. Reguła decyzyjna

Mając do dyspozycji wyliczone α możemy wyliczyć:

$$b = -\frac{1}{2}(max_{i:y_i=-1}\overline{w} \cdot \overline{x}_i + min_{i:y_i=1}\overline{w} \cdot \overline{x}_i)$$

(powyższe funkcje min i max posłużą znalezieniu wektorów nośnych z granicy marginesu)

$$\overline{w} = \sum_i \alpha_i g_i \overline{x}_i$$

Podstawiając wyliczone $\overline{w}$ do wyrażenia $\overline{w} \cdot \overline{u} + b \geq 0$

$$\overline{w} \cdot \overline{u} + b = \sum_i \alpha_i g_i \overline{x}_i \cdot \overline{u} + b \geq 0$$

uniezależniliśmy się od $\overline{w}$ - musimy znaleźć tylko mnożniki Lagrange’a α_i.

Niezerowe α_i wyznaczą wektory nośne.

3. Sztuczne sieci neuronowe - Artificial Neural Networks

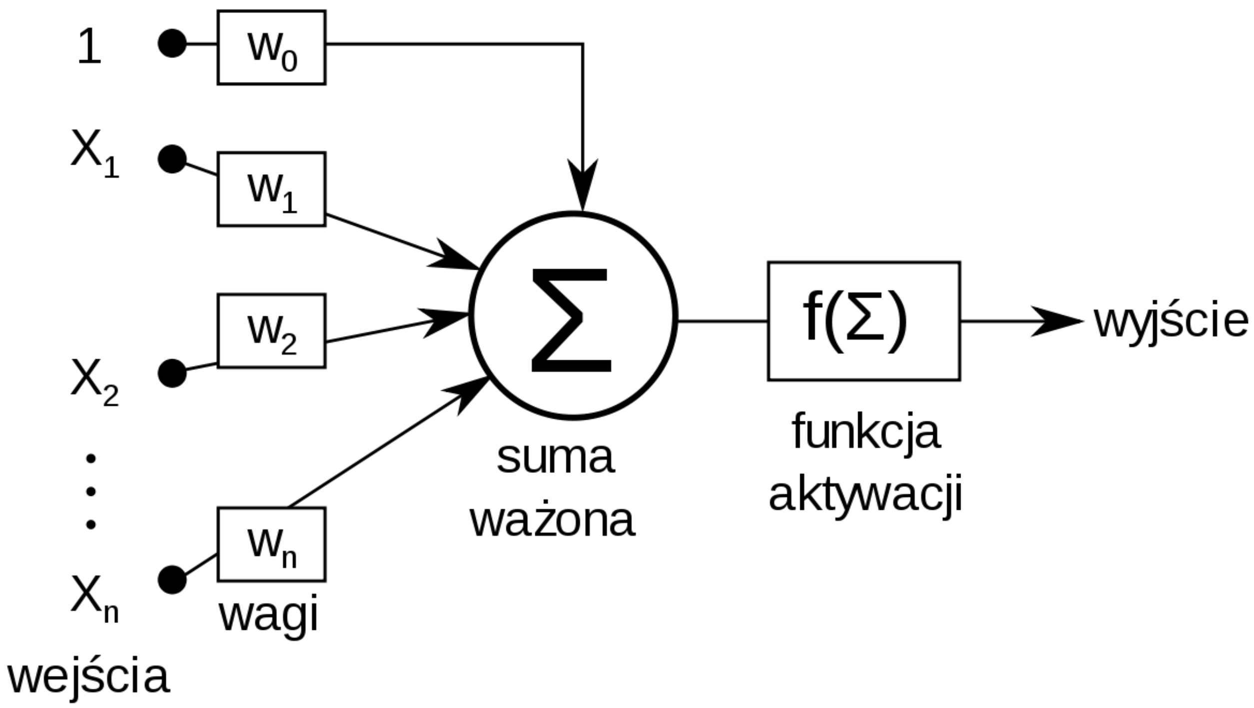
Sztuczne sieci neuronowe stanowią symulację funkcjonowania ludzkiego mózgu w problemach uczenia maszynowego. Ich historia zaczyna się w 1943r., kiedy to Warren McCulloch wraz z Walterem Pitts zamodelowali pierwszą sieć neuronową. Sztuczne sieci neuronowe są często uważane jako rozwiązanie na dowolny zadany problem, jako że dostarczają zazwyczaj lepszych rezultatów niż inne algorytmy uczenia maszynowego. W niektórych wypadkach pozostałe techniki, takie jak maszyna wektorów nośnych lub k-najbliższych sąsiadów, mogą dać lepszy rezultat, a użycie złożonych sieci neuronowych bywa nadmiarowe.

3.1. Zasada działania

W poniższym podrozdziale opiszemy w zarysie zasadę działania sztucznych sieci neuronowych. Szczegółowe definicje przedstawimy w dalszych podrozdziałach.

Mózg ludzki zbudowany jest z 86 miliardów komórek nerwowych zwanych neuronami. Każdy z nich połączony jest z tysiącami innych poprzez akson. Bodźce zewnętrzne docierają poprzez dendryty (wypustki komórki nerwowej przewodzące impulsy nerwowe). Wejścia te wytwarzają impulsy elektryczne, które neuron może przekazać dalej lub stłumić. Wszystkie te ładunki elektryczne bardzo szybko przemieszczają się po sieci neuronowej, wzmacniając się lub tłumiąc, by wykonać odpowiednią reakcję na zadany bodziec.

Wzorując się na ludzkim mózgu stworzono model sztucznego neuronu McCullocha-Pittsa, będący układem posiadającym pewną ilość argumentów wejściowych plus argument zerowy równy 1 nazywany *biasem*, taką samą liczbę wag stowarzyszonych z tymi argumentami oraz funkcji aktywacyjnej. Sieć zbudowaną za pomocą pojedynczego neuronu nazywamy *perceptronem*.



Model sztucznego neuronu McCullocha-Pittsa. ***[Rys. 7.]***

Działanie tego neuronu możemy opisać za pomocą wzoru:

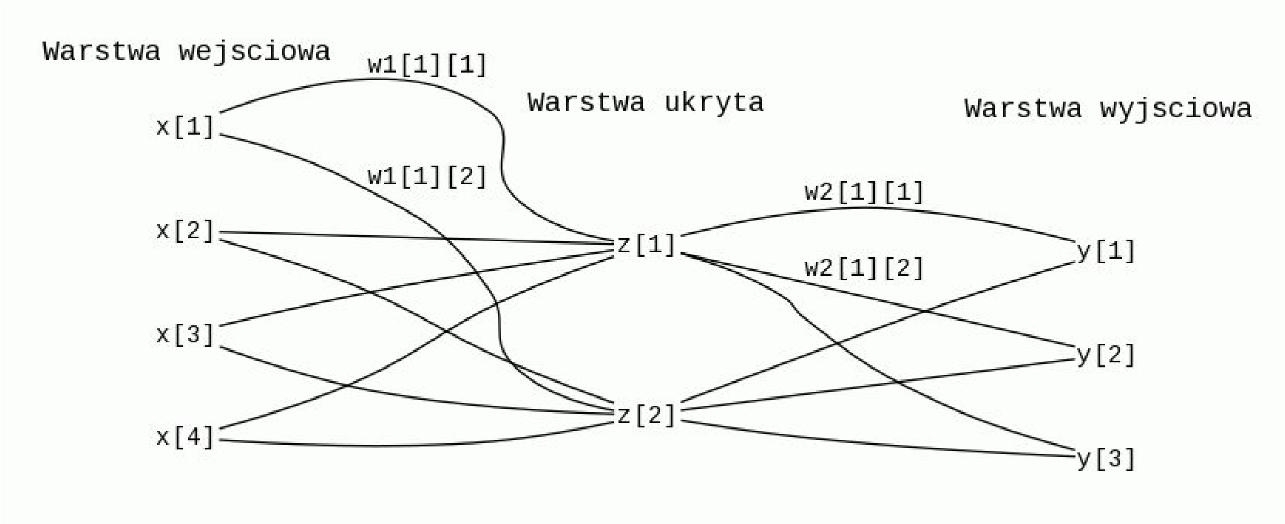
$$O(x_1, ..., x_n) = f(\sum_{i=1}^{n} w_i x_i) = f(w^t \cdot x)$$

gdzie $f : R \to R$ jest pewną *funkcją aktywacyjną*, w jest wektorem wag, a x wektorem wejściowym.

Wyróżniamy kilka podstawowych funkcji aktywacyjnych:

- progowa
- znakowa
- bipolarna (binarna)
- sigmoida
- tangens hiperboliczny (symetryczna sigmoida)
- identycznościowa
- afiniczna

Najbardziej rozpowszechnionym przykładem sieci neuronowych jest *perceptron wielowarstwowy*. Składa się on z uporządkowanych i rozłącznych klas elementów, nazywanych *warstwami*. Wśród nich można wyróżnić *warstwę wejściową* oraz *wyjściową*. Pozostałe warstwy nazywamy *ukrytymi*. Połączenia warstw są asymetryczne i skierowane zgodnie z ich uporządkowaniem - od warstwy wejściowej do wyjściowej. Neurony należące do tej samej klasy nie zawierają połączeń między sobą.



Perceptron wielowarstwowy. [Rys. 8.]

Na powyższej grafice widzimy schemat budowy perceptronu wielowarstwowego. Wektor x jest wektorem danych wejściowych, wektory $w1$, $w2$ zawierają wagi pomiędzy warstwami, natomiast y jest wektorem wynikowym.

W uczeniu nadzorowanym, mając do dyspozycji dane wejściowe oraz wyjściowe, problem wytrenowania sieci neuronowej jest równoważny znalezieniu wag między poszczególnymi neuronami. Zaletą sieci neuronowych jest to, że nie musimy szukać wag ręcznie. Dzięki metodzie obliczeniowej zwanej *wsteczną propagacją błędu*, potrafimy wytrenować naszą sieć, aby otrzymać w przybliżeniu optymalny zestaw wag.

Ogólny schemat procesu trenowania sieci wygląda następująco:

1. Ustalamy topologię sieci, czyli liczbę warstw oraz liczbę neuronów w warstwach.

2. Dobieramy losowe wagi o małych wartościach.
3. Dla ustalonej sieci, wag i wektora wejściowego obliczamy wynik warstwa po warstwie.
4. Neurony wyjściowe obliczają błąd (różnicę pomiędzy obliczoną wartością, a tą otrzymaną jako dane treningowe).
5. Błędy propagowane są do wcześniejszych warstw.
6. Każdy neuron modyfikuje swoje wagi na podstawie wartości obliczonego błędu.
7. Powtarzamy od punktu 3. dla następnych danych wejściowych, dopóki średni błąd nie przestaje maleć.

3.2. Podstawy matematyczne

3.2.1. Działanie pojedynczego neuronu

Dla wektora danych wejściowych $x_1, ..., x_j$ (wraz z dodatkowym wejściem $x_0 = 1$) i wektora wag $w_0, ..., w_j$ stowarzyszonych z tymi argumentami możemy obliczyć wartość h_{i_i}, zwaną *pobudzeniem*, korzystając z poniższego wzoru:

$$h_i = \sum_{j=0..n} w_{ij} x_j$$

Wynik ten przekazujemy do funkcji aktywacyjnej $g : R \to R$:

$$y_i = g(h_i)$$

Otrzymujemy w ten sposób wynik działania pojedynczego neuronu.

3.2.2. Podstawowe funkcje aktywacyjne

- Funkcja progowa

$$f(x) = \begin{cases} -1 & x < \Theta \\ +1 & x \geq \Theta \end{cases}$$

- Funkcja znakowa

$$f(x) = \begin{cases} -1 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

- Funkcja bipolarna (binarna)

$$f(x) = \begin{cases} 0 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

- Sigmoida

$$f(x) = \sigma(x) = \frac{1}{1 + exp(-\beta x)}$$

- Tangens hiperboliczny (symetryczna sigmoida)

$$f(x) = tanh(\frac{1}{2}\beta x) = \frac{1 - exp(-\beta x)}{1 + exp(-\beta x)}$$

- Funkcja identycznościowa

$$f(x) = x$$

- Funkcja afiniczna

$$f(x) = ax + b$$

3.2.3. Działanie perceptronu wielowarstwowego

Wynik sieci neuronowej można opisać za pomocą pewnego "wzoru":

$$y_i = \sum_{i,j} w_{ij} g(\sum_k w_{jk} g(...(\sum_t w_{st} x_t)))$$

gdzie wielokropek oznacza liczbę warstw neuronów.

Zatem wynik zwrócony przez sieć jest wynikiem pewnej funkcji bazującej na danych wejściowych oraz wagach w tych danych.

3.2.4. Algorytm wstecznej propagacji błędu

Losowo dobrany wektor wag nie zapewni najbardziej optymalnego modelu sieci. W celu uczenia sieci, czyli znajdowania najlepszego zestawu wag, stosuje się *algorytm wstecznej propagacji błędu.*

Dla wyliczonej przez sieć wartości wyjściowej t, wyznaczamy *średniokwadratową funkcję błędu* a następnie dążymy do znalezienia minimum tej funkcji względem wektora w.

Funkcja błędu wyraża się wzorem:

$$err(t) = \frac{1}{2}(t - y)^2$$

Przy wstecznej propagacji błędu ważną rolę odgrywa *algorytm spadku gradientowego*, dzięki któremu potrafimy znaleźć "kierunek", w którym należy podążyć, aby znaleźć minimum funkcji.

Zakładając, że mamy funkcję $f : R^d \to R$, która jest ciągła i różniczkowalna, czyli dla której możemy wyznaczyć pochodne cząstkowe $\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_d}$, oraz mamy ustalony pewien punkt startowy $a^{(0)} \in R$, możemy znaleźć kierunek w którym należy podążyć, aby znaleźć minimum.

Należy wyliczyć pochodne cząstkowe $\frac{\partial f}{\partial x_1}(a^{(0)}), \dots, \frac{\partial f}{\partial x_d}(a^{(0)})$. Każda z nich wyznaczać będzie kierunek, w którym nasza funkcja f rośnie, przy ustalonych pozostałych zmiennych. Aby znaleźć minimum tej funkcji należy wykonać krok w przeciwnym kierunku.

Aby zminimalizować funkcję błędu err za pomocą algorytmu spadku gradientowego, funkcja ta musi być również ciągła i różniczkowalna. Możemy to uzyskać poprzez wybranie funkcji aktywacyjnych poszczególnych neuronów, które będą ciągłe i różniczkowalne, np. sigmoida.

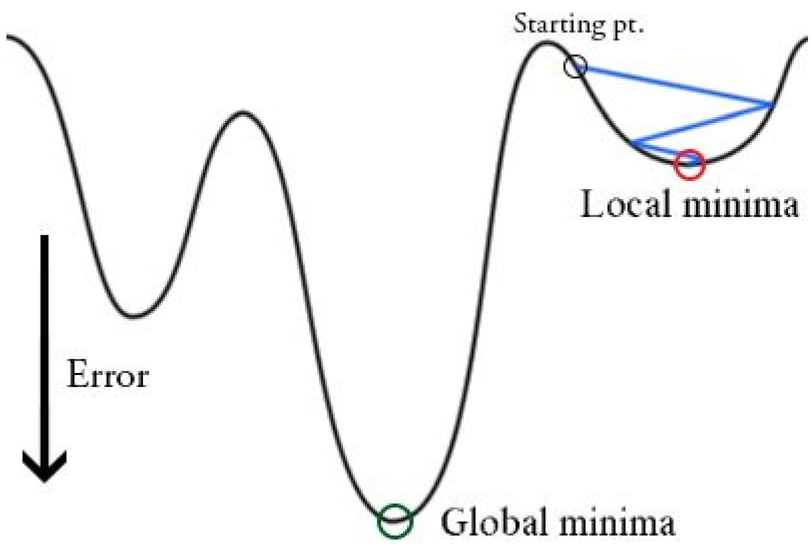
Algorytm wstecznej propagacji błędu polega na tym, że błąd wyliczony dla ostatniej warstwy wyjściowej propagowany do wcześniejszych warstw, mnożąc przez wagi na łączeniach neuronów oraz sumując dla kolejnych warstw:

$\delta_i = \sum_{j-1}^{k} w_{ij}\delta_j$, gdzie δ oznacza wynik funkcji błędu.

Korektę wag możemy dokonywać:

- on-line training - od razu dla każdego neuronu
- off-line training / batch training - po zakończeniu propagacji błędu przez całą sieć, na końcu wyliczając średnią

Problem na jaki możemy natrafić to utknięcie w *lokalnych minimach* (ang. *local minima*), zamiast dotarcia do *globalnego minimum* (ang. *global minima*) funkcji, stosując algorytm spadku gradientowego funkcji błędu. Szukając minimum i zaczynając z różnych miejsc, możemy otrzymać sytuację, w której przestaniemy minimalizować funkcję błędu znajdując lokalne minimum. Taką sytuację przedstawia poniższa grafika:

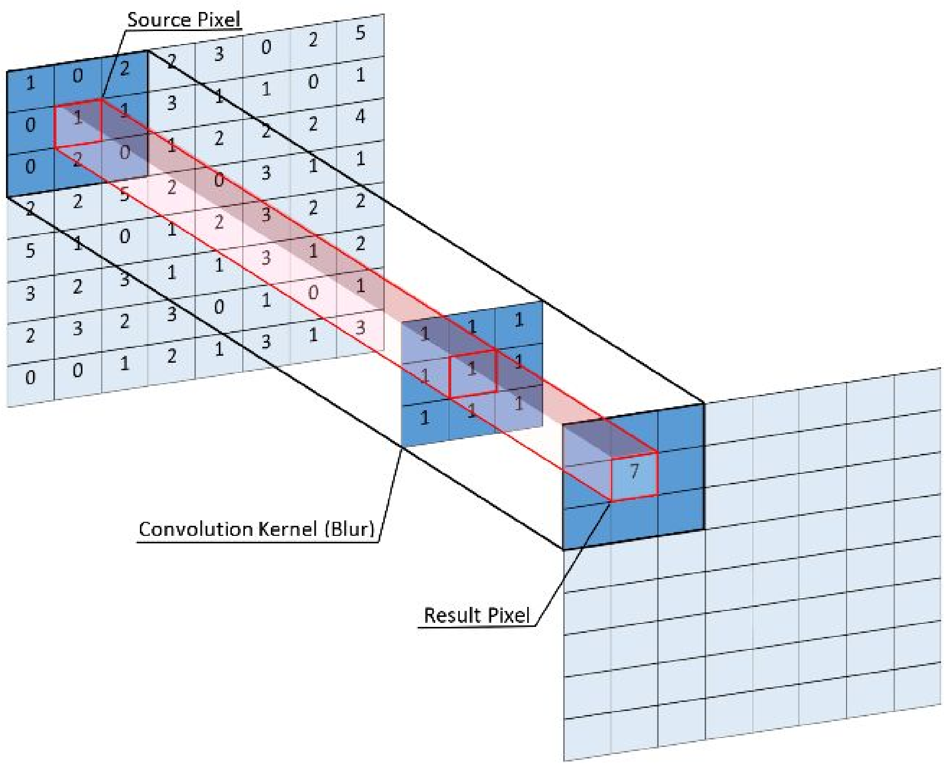


Problem gradientu prostego. [Rys. 9.]

3.3. Konwolucyjna sieć neuronowa

Rozważając wykorzystanie sztucznych sieci neuronowej do rozpoznawania elementów na obrazach również możemy traktować dane wejściowe (w tym wypadku grafiki) jako wektor liczb. Dla przykładu: pojedyncza grafika w formacie 28x28 ze zbioru MNIST zawierająca dowolną cyfrę w skali szarości może być traktowana jako wektor liczb o rozmiarze 784, w którym każda liczba znajduje się w przedziale od 0 do 255. Takie podejście powoduje, że jakakolwiek zmiana położenia cyfry powoduje traktowanie jej, przez sieć, jako kompletnie innego obiektu. A zatem nasz model musi mieć możliwość uczenia się każdej możliwej pozycji na zdjęciu. Aby tego uniknąć wykorzystuje się proces konwolucji, który powoduje, że dana cyfra pozostanie tą samą cyfrą niezależnie od pozycji na obrazie.

Działanie konwolucji, dla uproszczenia, przedstawię opisując poniższą grafikę:

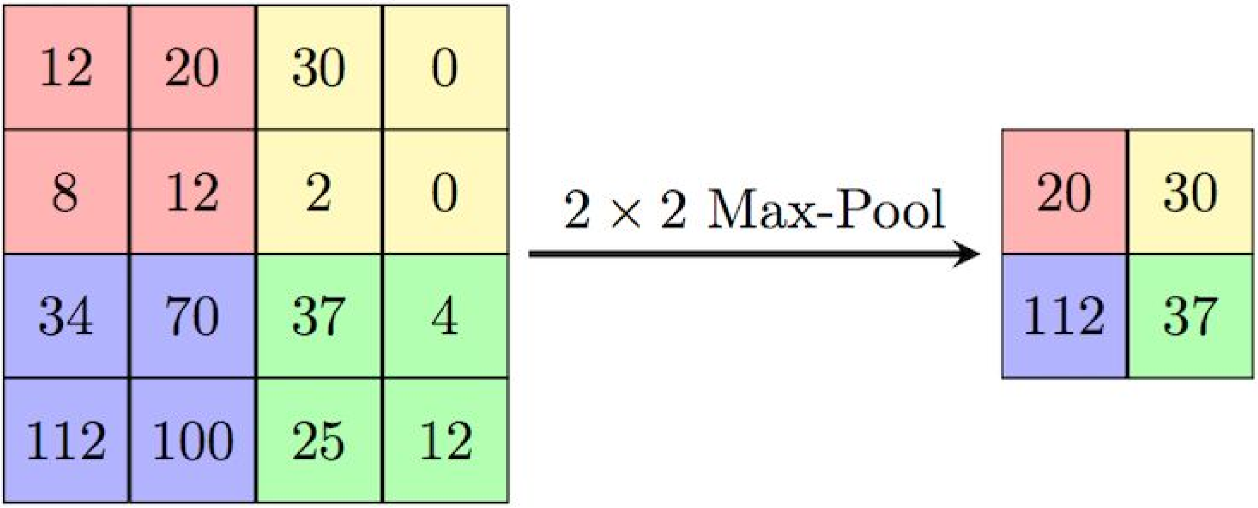


Zasada działania konwolucji. [Rys. 10.]

Mamy daną wejściową (grafikę), która jest tablicą składającą się z pojedynczych pixeli. Nakładamy na nią filtr nazywany jądrem (kernel) konwolucji. Może on być w rozmiarze np. 3x3, 5x5 lub 7x7. Zaczynamy od lewego górnego rogu tablicy, wybierając macierz rozmiaru identycznego jak jądro konwolucji. Następnie wykonujemy operację konwolucji, czyli mnożymy odpowiadające pixele przez siebie, a następnie je sumujemy (na powyższej grafice operacja ta wygląda następująco: $1*1+0*1+2*1+0*1+1*1+1*1+0*1+2*1+0*1=7$). Wynik tej operacji zapisujemy na obrazie wyjściowym w miejscu odpowiadającym centrum macierzy. Kolejnym krokiem jest przesunięcie filtru o jeden pixel w prawo. W momencie dojścia jądra do krawędzi obrazu filtr zostaje przeniesiony rząd niżej i rozpoczyna dalsze wykonywanie operacji konwolucji od lewej strony obrazu.

Warto zauważyć, że wynikowy obraz jest mniejszy od wejściowego. Pixele znajdujące się na krawędzi nigdy nie znajdą w centrum jądra konwolucji.

Wraz z operacją konwolucji często stosuje operację poolingu, czyli redukowania rozmiaru danych. Wykonuję się ją by ograniczyć ilość operacji na danych, a co za tym idzie, zmniejszyć zapotrzebowanie na moc obliczeniową. Jest ona bardzo podobna do operacji konwolucji. Tak samo na kolejne mniejsze macierze danych wejściowych nakłada się filtr, najczęściej rozmiaru 2x2, ale może być również 3x3, 5x5 itd.. Jednak tym razem nie mnoży się wartości (filr w poolingu nie posiada żadnych wartości), a wybiera pixel o największej wartości (MaxPooling) lub wylicza wartość średnią pixeli wybranej macierzy (AveragePooling). Wartość wynikową umieszcza się na obrazie wynikowym w miejscu odpowiadającym centrum filtru. Przykład działania MaxPoolingu 2x2 przedstawia poniższa grafika:

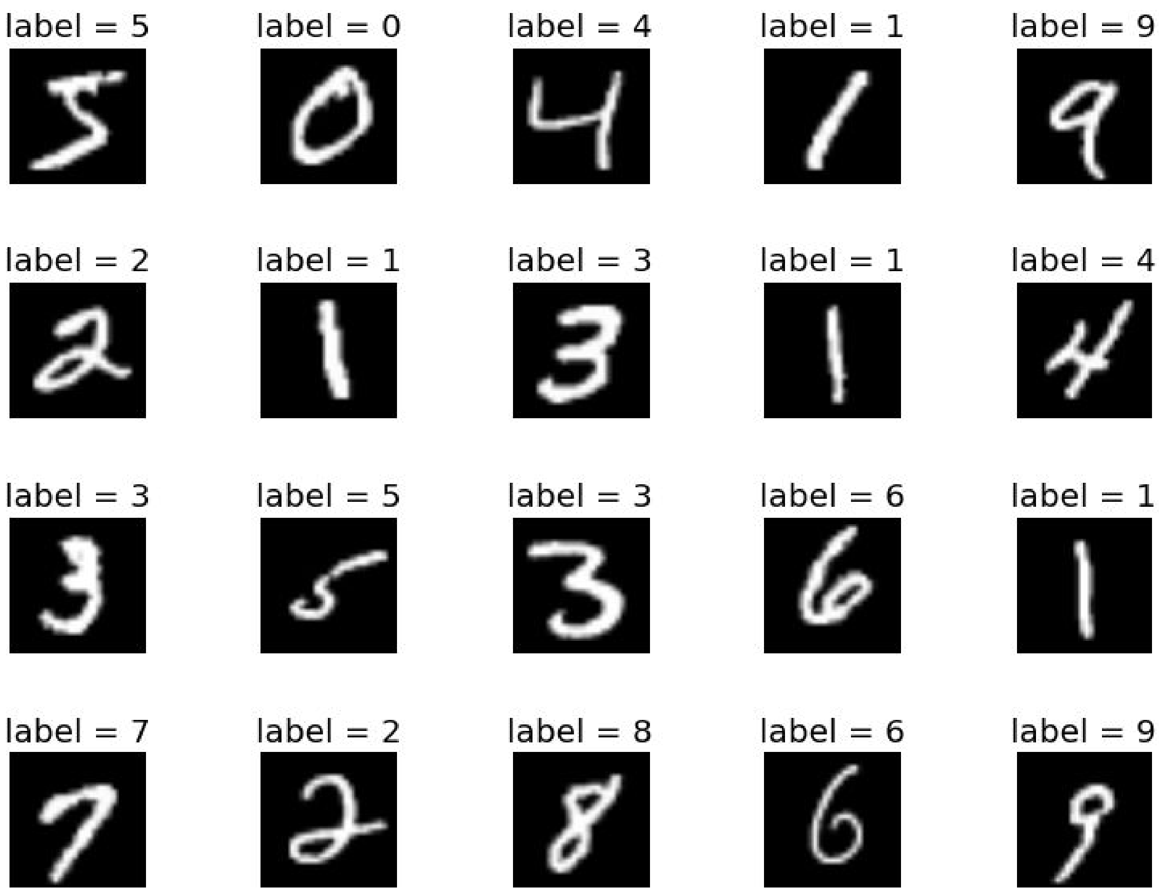


Zasada działania MaxPoolingu. [***Rys.11.***]

4. Rozpoznawanie odręcznie pisanych cyfr - aplikacja

Poniższy fragment pracy dyplomowej został poświęcony aplikacji, dzięki której można zobaczyć działanie modeli opisywanych w powyższych rozdziałach. Zawiera skrypty napisane w języku Python pozwalające na wytrenowanie modeli: sieci neuronowej oraz maszyny wektorów nośnych. Dane treningowe zostały pobrane z publicznie dostępnej bazy danych MNIST, zawierającej 70 tysięcy ręcznie pisanych cyfr w grafikach formatu 28x28px.

Repozytorium aplikacji znajduje się na GitHub:
https://github.com/jakubpradzynski/digits-predictor



Przykładowe cyfry ze zbioru MNIST. [Rys. 12.]

4.1. Sprzęt i technologie

Istotną kwestią mającą wpływ na rozwój uczenia maszynowego jest rosnąca moc obliczeniowa komputerów. Wielu dostawców platform chmurowych (między innymi Google z Google Cloud Platform oraz Microsoft z Microsoft Azure) również udostępnia produkty wyspecjalizowane do rozwiązywania problemów przy pomocy sztucznej inteligencji.

Klasyfikacja odręcznie pisanych cyfr ze zbioru MNIST nie wymaga bardzo dużych zasobów obliczeniowych. Problem ten można z sukcesem rozwiązać na standardowym komputerze czy laptopie. Modele niezbędne do działania aplikacji, napisanej w ramach pracy dyplomowej, zostały wytrenowane na laptopie o następujących parametrach technicznych:

- Procesor: Intel(R) Core(™) i7-5500U CPU @ 2.40GHz (4 CPUs)
- Ram: 16384 MB
- Typ systemu: 64-bitowy system operacyjny Windows 10 Pro
- Pamięć:
- Karta graficzna: Intel(R) HD Graphics 5000
- Karta graficzna do renderowania: AMD Radeon (™) R9 M375.

Językiem programowania, uważanym za najbardziej popularny wśród specjalistów Data Science jest Python. Niski próg wejścia, prostota oraz duży zasób gotowych rozwiązań sprawia, że jest pierwszym wyborem osób rozpoczynających karierę w tym zawodzie. Dlatego poniżej przedstawiona aplikacja również zawiera skrypty napisane w języku Python w wersji 3.7.2. Te zaś wymagają następujących bibliotek:

- Keras - wersja 2.2.4 (z TensorFlow w wersji 1.13.0rc1)
- Scikit-learn - wersja 0.20.2
- Matplotlib - wersja 3.0.2
- NumPy - wersja 1.16.1
- Pillow - wersja 5.4.1

Powyższe biblioteki można zainstalować wykonując komendę "*pip3 install keras tensorflow scikit-learn matplotlib numpy pillow*".

Aplikacja pozwalająca na zademonstrowanie wykorzystania utworzonych modeli została napisana w języku Java (wersja 8) oraz wykorzystuje Maven jako narzędzie do automatycznego budowania.

4.2. Skrypt uczący model Maszyny Wektorów Nośnych

Skrypt znajduje się w pliku o nazwie "*svm-train.py*".

Działanie procesu rozpoczyna się od wczytania danych treningowych z bazy danych MNIST:

```
mnist_data_set = fetch_openml('mnist_784', version=1, cache=True)
```

Otrzymany zbiór należy rozdzielić na grafiki zawierające odręcznie pisane cyfry oraz oznaczenie, który z symboli przedstawia:

```
digits_images = mnist_data_set.data
digits_labels = mnist_data_set.target
```

Obrazy to tablica liczb o długości 784, każdy z nich zawiera liczbę z przedziału od 0 do 255. W celu zmniejszyć rozmiar obliczeń należy je znormalizować, czyli każdą z wartości podzielić przez 255.

```
digits_images = digits_images / 255.0
```

Dzięki funkcji train_test_split z biblioteki scikit-learn dzielimy posiadane dane na zestaw treningowy i zestaw testowy w stosunku 85% do 15% z ziarnem inicjującym generator pseudo losowych liczb o wartości 42:

```
train_images, test_images, train_labels, test_labels =
train_test_split(digits_images, digits_labels, test_size=0.15,
random_state=42)
```

Następnie, za pomocą metody SVC z pakietu svm dostępnego w bibliotece scikit-learn, tworzony jest klasyfikator. Przy inicjowaniu podajemy następujące parametry: C - margines błędu, gamma - współczynnik jądra, probability - włączenie szacunków prawdopodobieństwa, verbose - włączenie logów podczas uczenia modelu:

```
param_C = 5
param_gamma = 0.05
classifier = svm.SVC(C=param_C, gamma=param_gamma, probability=True,
verbose=True)
```

Wywołanie metody fit na wcześniej utworzonym klasyfikatorze rozpoczyna trenowanie modelu Maszyny Wektorów Nośnych. Metoda jako parametry przyjmuje dane treningowe (w tym wypadku grafiki zawierające cyfry) oraz zestaw ich oznaczeń (informacje jakie symbole zawiera poszczególny obrazek)

```
classifier.fit(train_images, train_labels)
```

Po zakończeniu trenowania, obiekt klasyfikatora zawiera niezbędne dane do przewidywania kolejnych cyfr z dostarczanych grafik. W celu sprawdzenia skuteczności modelu należy wywołać metodę predict na klasyfikatorze, dostarczając jej wcześniej wydzielonych danych testowych. Za pomocą metody accuracy_score z pakietu metrics z biblioteki scikit-learn, możemy otrzymać jakość modelu:

```
predict_result_on_test_images = classifier.predict(test_images)
print("Model accuracy =
{}".format(metrics.accuracy_score(test_labels,
predict_result_on_test_images)))
```

Dzięki funkcji dump z biblioteki pickle możemy zapisać uzyskany model do pliku, aby móc z niego ponownie skorzystać w innych programach:

```
model_filename = "svm_model.sav"
pickle.dump(classifier, open(model_filename, "wb"))
```

Skrypt można uruchomić za pomocą komendy “*python3 svm-train.py*”.

4.2. Skrypt uczący model Sztucznej Sieci Neuronowej

Skrypt znajduje się w pliku o nazwie “*ann-train.py*”.

Punktem startowym programu jest wczytanie zestawu danych z bazy danych MNIST dzięki funkcji load_data z biblioteki keras. Pozwala ona na odczyt z jednoczesnym podziałem danych na treningowe i testowe, oraz grafiki i ich oznaczenia:

```
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
```

Następnie należy zmienić format wczytanych grafik na odpowiadający modelowi Sieci Neuronowej:

```
train_images = train_images.reshape(train_images.shape[0],
train_images.shape[1], train_images.shape[2], 1).astype("float32")
test_images = test_images.reshape(test_images.shape[0],
test_images.shape[1], test_images.shape[2], 1).astype("float32")
```

Tak jak przy trenowaniu Maszyny Wektorów Nośnych, w celu ułatwienia obliczeń, należy znormalizować wartości pojedynczych elementów, składających się na tablicę opisującą pojedynczą grafikę zawierającą odręcznie napisaną cyfrę.

```
train_images /= 255
test_images /= 255
```

Do oznaczenia kategorii często używa się kodu 1 z n. Aby otrzymać taką kategoryzację, wykorzystujemy funkcję to_categorical z pakietu utils dostępnego w bibliotece keras, która zamieni wektor liczbowy klas na macierz klas binarnych. Wymaga ona dostarczenia oznaczeń grafik oraz liczbę klas (w tym przypadku liczbę cyfr w systemie dziesiętnym):

```
number_of_classes = 10
```

```
train_labels = keras.utils.to_categorical(train_labels,
number_of_classes)
test_labels = keras.utils.to_categorical(test_labels,
number_of_classes)
```

Kolejnym etapem jest utworzenie modelu Sieci Neuronowej, który później będzie trenowany na wcześniej zebranych danych. Instancję modelu możemy zainicjować metodą Sequential, odpowiadającą za utworzenie liniowego stosu warstw. Kolejne warstwy możemy dodawać za pomocą metody add wywołaną na modelu:

```
model = Sequential()
model.add(Conv2D(32, (5, 5), input_shape=(train_images.shape[1],
train_images.shape[2], 1), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(number_of_classes, activation="softmax"))
```

Przed rozpoczęciem trenowania modelu należy skonfigurować proces uczenia metodą compile. Jako parametry przyjmuje ona nazwę funkcji straty, funkcję optymalizującą oraz listę metryk jakie chcemy otrzymać:

```
model.compile(loss="categorical_crossentropy", optimizer=Adam(),
metrics=["accuracy"])
```

Po skompilowaniu możemy rozpocząć trenowanie Sieci Neuronowej metodą fit dostarczając jej niezbędnych danych, takich jak zestaw treningowy, zestaw walidacyjny, liczbę iteracji oraz jednorazowy rozmiar próby:

```
model.fit(train_images, train_labels, validation_data=(test_images,
test_labels), epochs=7, batch_size=200)
```

Otrzymany model możemy przetestować za pomocą metody evaluate dostarczając jej danych testowych pozyskując dane o skuteczności:

```
score = model.evaluate(test_images, test_labels, verbose=0)
print("Model loss = {}".format(score[0]))
print("Model accuracy = {}".format(score[1]))
```

Za pomocą metody save zapisujemy model do pliku, aby można było z niego skorzystać w innym programie:

```
model_filename = "ann_model.h5"
model.save(model_filename)
```

Skrypt można uruchomić za pomocą komendy "*python ann-train.py*".

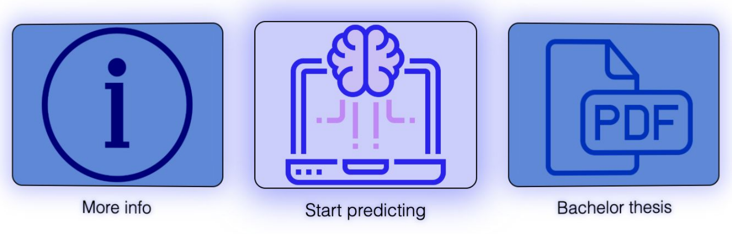
4.3. Działanie aplikacji do przewidywania cyfr

Główna aplikacja została napisana w Javie wykorzystując Spring Boot. Może zostać uruchomiona za pomocą komendy "mvn spring-boot:run". Aby aplikacja działała poprawnie, należy w głównym katalogu projektu posiadać pliki "ann_model.h5" oraz "svm_model.sav", które zostają utworzone po wytrenowaniu modeli Sieci Neuronowej oraz Maszyny Wektorów Nośnych poprzez skrypty przedstawione w dwóch poprzednich podrozdziałach.

Po uruchomieniu aplikacja dostępna jest na porcie 8080. Wystarczy w przeglądarce wpisać http://localhost:8080. Po wejściu na stronę ukaże się strona z przyciskiem:

Przycisk startowy aplikacji. *[Rys. 13.]*

Po najechaniu na przycisk rozszerzy się on do 3 pól:



Menu aplikacji otwierane po najechaniu na przycisk Start. *[Rys. 14.]*

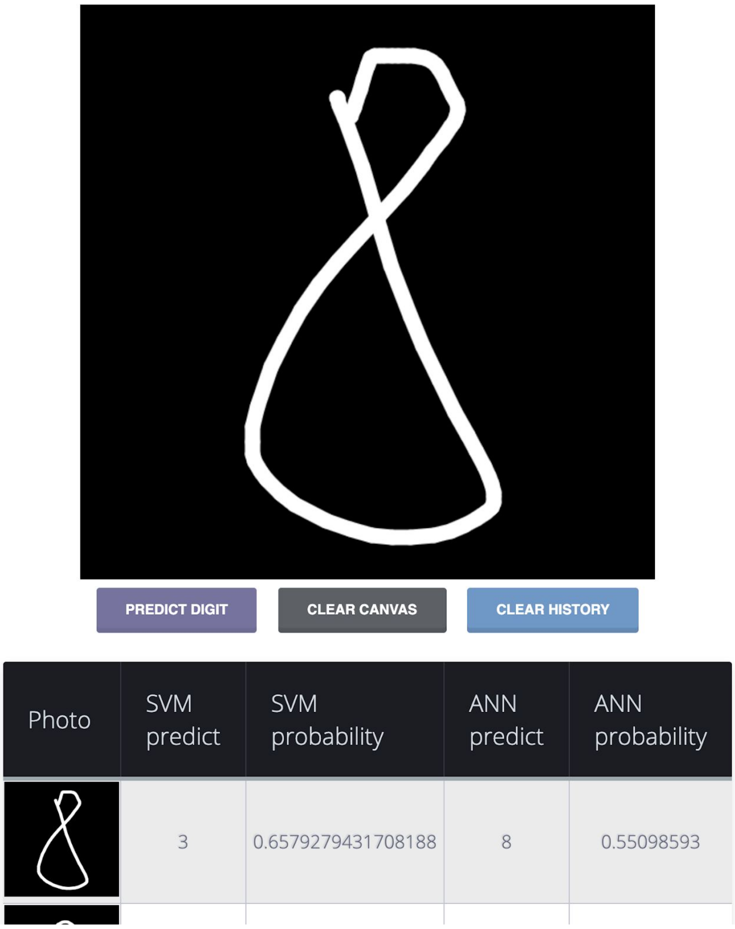
“More info” przekierowuje do pliku README.md znadującego się w repozytorium projektu na GitHubie. Zawiera on informacje o aplikacji. Przycisk “Bachelor thesis” pozwala na pobranie niniejszej pracy licencjackiej w formacie pdf. “Start predicting” przekierowuje do kolejnej podstrony programu, która zawiera główną część aplikacji.

Na czarnym tle za pomocą myszki lub touchpada możemy narysować dowolną cyfrę. Przycisk “*Clear canvas*” spowoduje wyczyszczenie płótna. Za pomocą przycisku “*Predict digit*” wywołamy zapisanie płótna do zdjęcia w formacie png oraz uruchomienie skryptu “predictor.py”. Wczytuje on wcześniej zapisaną grafikę, przygotowuje dane dla wcześniej utworzonych modeli Sieci Neuronowej oraz Maszyny Wektorów Nośnych oraz uruchamia je w celu otrzymania predykcji. Po chwili, pod płótnem, ukaże się tabela zawierająca wyniki predykcji. Pokazuje ona jaką cyfrę wyróżniły poszczególne modele oraz z jakim prawdopodobieństwem. Każde kolejna predykcja zostanie umieszczona na górze tabeli. Przycisk “Clear history” czyści tabelę z poprzednimi wywołaniami. Poprawną predykcję przedstawia poniższa grafika:

Photo	SVM predict	SVM probability	ANN predict	ANN probability
	1	0.8280776982292735	1	0.6892023

Przykład poprawnej predykcji w aplikacji. [Rys. 15.]

Czasami, dla mniej dokładnie narysowanych cyfr, modele mogą zwrócić błędny wynik:



Przykład błędnej predykcji w aplikacji przez model SVM. [Rys. 16.]

Spowodowane jest to tym, że modele analizują obrazy w dużo prostszy sposób niż ludzki mózg. Każde odchylenie, zmiana pozycji czy rozmiaru wpływają na wynik zwrócony przez Maszynę Wektorów Nośnych czy też Sztuczną Sieć Neuronową.

Podsumowanie

Problem rozpoznawania odręcznie pisanych cyfr, zawartych w bazie danych MNIST jest często nazywany Hello Worldem przetwarzania obrazów za pomocą uczenia maszynowego. Na konferencjach takich jak CVPR (Conference on Computer Vision and Pattern Recognition) były organizowane zawody polegające na utworzeniu modelu o jak najmniejszym współczynniku błędu rozwiązujący ten problem. Modele stworzone w ramach pracy dyplomowej osiągnęły skuteczność na poziomie około 0.985 (Maszyna Wektorów Nośnych) przy czasie uczenia równym 23 minuty i 12 sekund oraz 0.99 (Sieć Neuronowa) przy czasie uczenia równym 6 minut i 47 sekund.

W powyżej opisywanej aplikacji chciałem zademonstrować jak bardzo rozwój sztucznej inteligencji oraz zainteresowanie tym tematem wpłynęło na prostotę korzystania z tak zaawansowanego zagadnienia jakim jest uczenie maszynowe. Aby samemu zaimplementować i skonfigurować proces uczenia wybranego modelu wystarczy zapoznać się z biblioteką (np. Keras) dostarczającą gotowych rozwiązań. Dla osób preferujący operowanie na interfejsie graficznym powstały platformy takie jak Google App Engine czy Microsoft Azure, gdzie zamiast pisania własnych skryptów można wyklikać to co jest nam potrzebne do rozwiązania problemu biznesowego.

Dodatkowo program główny pozwala zobaczyć jak w prosty sposób można wykorzystać uczenie maszynowe, aby otrzymać rozwiązanie na zadany problem oraz użyć go w swojej aplikacji. W tym przypadku możemy zobaczyć czy narysowana liczba jest na tyle kształtna, aby komputer ją rozpoznał i zakwalifikował do odpowiedniej klasy. Rozwiązanie to można by przenieść na przykład do aplikacji mobilnej, która służyłaby do nauki pisania. Jeżeli stworzyłoby się dodatkowy model do rozpoznawania liter danego alfabetu, można by użyć prawdopodobieństwa rozpoznania danej litery lub cyfry przez komputer jako wyznacznika jakości narysowanej cyfry. Im bardziej model jest pewny, że narysowany znak jest poprawny, tym pismo jest lepsze. Taką aplikację mogliby wykorzystywać rodzice, których dzieci od najmłodszych lat interesują się telefonami czy tabletami lub w szkole podstawowej jako dodatkowa metoda nauki pisania.

Bibliografia

Kursy

1. dr Andrew Ng, *Machine Learning*, Coursera 2017
2. dr Geoffrey Hinton, *Neural Networks*, Coursera 2017
3. dr Andrew Ng, *Neural Networks and Deep Learning*, Coursera 2018
4. *Artificial Intelligence - Neural Networks*, Tutorials Point

Artykuły

1. *Neural Networks History: The 1940's to the 1970's*, Stanford University
2. *Neural Networks History: The 1980's to the present*, Stanford University
3. Andrew Ng, *CS229 Lecture notes Part V Support Vector Machines*, Stanford University
4. Andrew Ng, Kian Katanforoosh, *CS229 Lecture Notes Deep Learning*, Stanford University
5. *CS231n Convolutional Neural Networks for Visual Recognition*, Stanford University
6. *Profesor Vladimir Vapnik*, Columbia University in the city of New York
7. *A history of machine learning*, Google
8. David Fumo, *Types of Machine Learning Algorithms You Should Know*, Towards Data Science 2017
9. Oleksii Kharkovyna, *Top 10 Machine Learning Algorithms for Data Science*, Towards Data Science 2018
10. Rohith Gandhi, *Support Vector Machine — Introduction to Machine Learning Algorithms*, Towards Data Science 2018
11. Rushikesh Pupale, *Support Vector Machines(SVM) — An Overview*, Towards Data Science 2018
12. Anuradha Wickramarachchi, *Introduction to Neural Networks*, Towards Data Science 2017

13. Sumit Saha, *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*, Towards Data Science 2018
14. Christina Voskoglou, *What is the best programming language for Machine Learning?*, Towards Data Science 2017
15. *Sztuczna inteligencja/SI moduł 12*, Wazniak Mimuw Edu
16. Bernard Marr, *The Top 10 AI And Machine Learning Use Cases Everyone Should Know About*, Forbes 2016
17. Yann LeCun, Corinna Cortes, Christopher J.C. Burges, *The MNIST database of handwritten digits*
18. *IEEE Conference on Computer Vision and Pattern Recognition*

Prezentacje

1. Adrian Horzyk, *Maszyna Wektorów Nośnych*, Akademia Górniczo-Hutnicza
2. Adrian Horzyk, *Biocybernetyka Sieci Neuronowe*, Akademia Górniczo-Hutnicza
3. A. Rutkowski, M. Czoków, J. Piersa, *Support Vector Machine*, Uniwersytet Mikołaja Kopernika 2017
4. A. Rutkowski, M. Czoków, J. Piersa, *Neuron biologiczny. Model perceptronu prostego.*, Uniwersytet Mikołaja Kopernika 2018
5. A. Rutkowski, M. Czoków, J. Piersa, *Neuron biologiczny. Model perceptronu prostego.*, Uniwersytet Mikołaja Kopernika 2018
6. M. Czoków, J. Piersa, *Algorytm wstecznej propagacji błędu*, Uniwersytet Mikołaja Kopernika 2011
7. *Sztuczna Sieć Neuronowa*, Politechnika Poznańska

Grafiki