

WYDZIAŁ INFORMATYKI STOSOWANEJ

Kierunek: INFORMATYKA

**Specjalność: Programowanie**

Bohdan Turanyi

53470

Realizacja projektu aplikacji webowej do przeprowadzenia obliczeń za pomocą sztucznych sieci neuronowych

Promotor: dr inż. Mariusz Wrzesień

PRACA DYPLOMOWA INŻYNIERSKA

Rzeszów 2019

Ja niżej podpisany/a oświadczam, że składana przeze mnie praca dyplomowa pt. „Realizacja projektu aplikacji webowej do przeprowadzenia obliczeń za pomocą sztucznych sieci neuronowych” została przygotowana samodzielnie. Oświadczam również, że praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni. Oświadczam ponadto, że niniejsza wersja pracy jest identyczna ze złożoną wersją elektroniczną.

.........................................................

data czytelny podpis autora

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego. ................................................................

data czytelny podpis promotora

Table of Contents

[1. Wstęp 4](#_Toc33649408)

[1.1. Cel pracy 5](#_Toc33649409)

[1.2. Wymagania funkcjonalne 5](#_Toc33649410)

[1.3. Wymagania niefunkcjonalne 5](#_Toc33649411)

[2. Część teoretyczna 6](#_Toc33649412)

[3. Część praktyczna 8](#_Toc33649413)

[4. Zakończenie 9](#_Toc33649414)

[5. Literatura 10](#_Toc33649415)

[Streszczenie pracy 11](#_Toc33649416)

[Załączniki 12](#_Toc33649417)

# Wstęp

Z każdym dniem uczenia maszynowego używa się coraz częściej nie tylko w badaniach akademickich, a i w rozwiązaniu codziennych problemów zwykłych użytkowników. Ilość systemów informatycznych które wykorzystują uczenie maszynowe rośnie i powodem temu jest nie tylko rozwój uczenia maszynowego pod względem ilości i efektywności jego metod. Rozwój narzędzi, które udostępniają te metody, mówiąc inaczej frameworków, pod względem wygodności korzystania z nich ma wielki wpływ na wzrost popularności całej dziedziny uczenia maszynowego.

Na dzień dzisiejszy istnieje mnóstwo frameworków dla uczenia maszynowego. Jedne skupiają się na szybkości działania implementowanych systemów co jest dużą zaletą dla systemów z wysokim obciążeniem, ale często taka szybkość jest uzyskiwana ceną przystępności. Przykładem jest biblioteka mlpack(<https://www.mlpack.org/>), która jest napisana w języku C++. Ona, chociaż i ma API w języku Python, ale korzysta pod spodem z silnika napisanego w C++ i dlatego wyróżnia się swoją składnią. Inne frameworki skupiają się na łatwości i wygodności korzystania z nich. Przykładami służą większość frameworków których „jądro” jest napisane w języku Python, wysokopoziomowy i łatwy syntaks którego sprawia, że programista traci mniej czasu na napisanie kodu, ale napisany element systemu może potrzebować więcej mocy obliczeniowej działając w środowisku produkcyjnym.

Dwa opisane powyżej czynniki, szybkość implementacji i wydajność obliczeniowa, są bardzo ze sobą powiązane: są wynikiem bilansu pomiędzy „wygodnością” dla maszyny obliczeniowej a wygodnością dla programisty, który wprowadza do maszyny rozkazy. W wysokopoziomowych językach mamy mniej możliwości do optymalizacji instrukcji przekazywanych komputerowi, dzięki czemu zyskujemy czytelność, a w niskopoziomowych mamy tych możliwości więcej, ale jesteśmy często wymuszeni do ich używania bez względu na to jaki wynik chcemy uzyskać. Z tego powodu często zyskując jedno tracimy drugie. Często ten problem w działce uczenia maszynowego rozwiązują implementując prototyp w języku, w którym się szybko pisze, a potem zaimplementowane, działające rozwiązanie przepisują w języku z wyższą wydajnością obliczeniową. Jest to rozwiązanie problemu, ale jednocześnie wymusza znajomość dwóch języków albo zatrudnienie więcej ludzi. Opisany problem jest aktualny nie tylko dla działki uczenia maszynowego, ale w niej on jest najbardziej wyraźny.

W roku 2012(<https://en.wikipedia.org/wiki/Julia_(programming_language>[)](https://en.wikipedia.org/wiki/Julia_(programming_language))) powstał język Julia, który powinien rozwiązać ten problem. Udostępniając jednocześnie syntaks wysokopoziomowy jak i duże możliwości do optymalizacji kodu, Julia może mieć duży wpływ na społeczeństwo naukowców i programistów które tworzą rozwiązania w działce obliczeń numerycznych, m. in związane z uczeniem maszynowym.

## Cel pracy

Celem niniejszej pracy jest zapoznanie się z językiem programowania Julia jak i zapoznanie się z budową systemów opartych o sztuczne sieci neuronowe (dalej „sieci neuronowe”). Cele zostaną spełnione poprzez implementację narzędzia do przeprowadzenia obliczeń za pomocą sieci neuronowych z interfejsem webowym w języku Julia. Dla implementacji zostanie użyta wersja 1.3.1 języku Julia.

## Wymagania funkcjonalne

1. Aplikacja powinna mieć interfejs w postaci strony internetowej.
2. Aplikacja powinna mieć możliwość wczytania pliku CSV jako pliku źródłowego do przeprowadzenia obliczeń.
3. Aplikacja powinna udostępniać możliwość wczytywania pliku dla predykcji na przetrenowanym modelu.
4. Aplikacja powinna mieć możliwość definicji podzbiorów pliku źródłowego jako zbiory dla uczenia i testowania sieci neuronowej. Te zbiory muszą nie mieć wspólnej części.
5. Aplikacja powinna udostępniać funkcjonalność dla definicji kolumny ze zmienną zależną w pliku źródłowym.
6. Aplikacja powinna udostępniać możliwość definicji typu problemu rozwiązywanego siecią neuronową (klasyfikacja, regresja).
7. Aplikacja powinna udostępniać możliwość definicji wymiarów i typów warstw sieci neuronowej.
8. Aplikacja powinna udostępniać możliwość ustawienia parametrów treningowych, m.in.: ilość iteracji treningowych, szybkość uczenia się, funkcję straty.
9. Aplikacja powinna udostępniać możliwość sterowaniem losowymi elementami sieci neuronowej poprzez ustawienie ziarna dla generatora liczb losowych.
10. Aplikacja powinna mieć możliwość trenowania sieci na zbiorze danych podanym przez użytkownika.
11. Aplikacja powinna wyświetlać wynik uczenia się w postaci wyniku funkcji straty na zbiorze testowym.
12. Aplikacja powinna mieć możliwość przeprowadzenia predykcji na zbiorze dla predykcji.
13. Aplikacja powinna mieć możliwość pobrania pliku CSV z wynikami predykcji.

## Wymagania niefunkcjonalne

1. Aplikacja powinna zapewniać powtarzalność wyników.
2. Stan aplikacji powinien być ograniczony do jednej sesji użytkownika.
3. Aplikacja powinna być responsywną pod czas przeprowadzenia obliczeń.

## Część teoretyczna

Sieci neuronowe składają się z neuronów. Zasadę uczenia się sieci neuronowej można wytłumaczyć na przykładzie jednego neuronu. Uczenie się neuronu jest iteratywnym procesem jeden cykl, którego składa się z trzech etapów:

1. Próba aproksymacji nieznanej funkcji.
2. Ocena wyników. Często jest to zwykle porównanie wyników aproksymacji do wyników funkcji, którą neuron aproksymuje.
3. Zmiana stanu wewnętrznego neuronu według otrzymanej oceny w celu uzyskania lepszego wyniku w dalszych próbach aproksymacji.

Mówiąc inaczej, neurony uczą się „metodą prób i błędów”.

Podczas trzeciego etapu, neuron zmienia swój stan. Stan neuronu jest reprezentowany przez, tak zwaną, wagę. Waga reguluje, jak reagować na parametry wejściowe neuronu. W nowoczesnych implementacjach waga jest liczbą, na którą mnoży się jeden parametr wejściowy. Ucząc się, neuron, uczy się reakcji na parametr wejściowy. W przypadku neuronu, który posiada więcej niż jeden parametr wejściowy, parametry, wcześniej przemnożone przez wagi, dodają się.

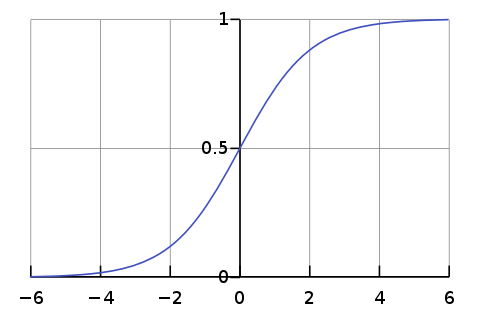
W tej postaci, neuron można porównać do funkcji:. Wspomniana funkcja zawsze będzie miała wynik równy zeru, w przypadku, kiedy wszystkie wejścia będą równe zeru, co istotnie ogranicza zakres funkcji których się neuron może nauczyć. Żeby eliminować tą wadę, możemy dodać wagę, która nie zależy od wejść i której możemy się „nauczyć”. Często taka waga jest notowana jako albo - od słowa „bias”. Z takim dodatkiem funkcja neuronu może aproksymować każdą możliwą funkcję liniową i wygląda tak: .

Na danym etapie neuron jest ograniczony liniowością aproksymowanej funkcji i, zabiegając do przodu, ze względu na to jak działa złożenie kilku funkcji liniowych, wynikiem aproksymacji całej sieci neuronowej składającej się z dowolnej ilości neuronów będzie wynik aproksymacji funkcji liniowej. Niestety, funkcje świata rzeczywistego rzadko kiedy nadają się do aproksymacji liniowej. Dla tego, żeby móc wykorzystywać sieci neuronowe do aproksymacji zjawisk naturalnych potrzebujemy pozbyć się liniowego ograniczenia neuronu. W tym celu możemy dodać do neuronu przetworzenie nieliniowe. Takie przetworzenie nieliniowe dodawane do funkcji neuronu nazywa się funkcją aktywacji. Jest niezliczone mnóstwo nieliniowych przetworzeń, które możemy zastosować na tej podstawie powstaje pytanie: jakie wybrać? Wybór zależy od rozwiązywanego problemu, od funkcji którą chcemy aproksymować i zwykle dla różnych części sieci (warstw, jeśli być bardziej precyzyjnym) wybór będzie różny.

Na dzień dzisiejszy już sformowały się standardy wykorzystywanych funkcji aktywacji dla często rozwiązywanych problemów. Opisywać poszczególne przypadki można bardzo długo, dlatego rozpatrzymy jeden, najprostszy przypadek – klasyfikację binarną. W wyniku klasyfikacji chcemy otrzymać najbardziej prawdopodobną klasę. W tym celu możemy stwierdzić, że jeśli wynik funkcji neuronu będzie większy niż przykładowe znaczenie **t**, to uznajemy, że większe prawdopodobieństwo będzie miała pierwsza kategoria, a jeśli mniej, to druga. Analogiem takiego założenia będzie funkcja progowa:

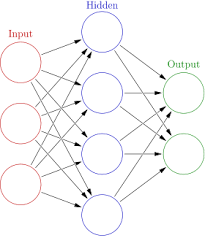
Teoretycznie **t** może być dowolnym znaczeniem ze względu na to, że neuron będzie uczyć się dostosowywać swój wynik do prawdziwego.

W przypadku klasyfikacji niebinarnej, będziemy musieli dodać więcej neuronów ze względu na to, że nie da się rozdzielić zakresu wyników możliwych funkcji neuronu na 3 albo więcej odcinków operując sensownymi dla komputera liczbami. Funkcja niebinarnej klasyfikacji zwykle jest binarną funkcją klasyfikacji dla każdej kategorii (1 neuron na kategorię). Natomiast, jeśli będziemy korzystać z połączenia kilku neuronów z progową funkcją aktywacji to pojawia się sytuacja, w której możemy dostać wynik 1 dla więcej niż jednej klasy(kategorii). Pozbyć się takiego problemu można decydując o klasie na podstawie prawdopodobieństwa, wybierając klasę z największym prawdopodobieństwem. Uzyskać prawdopodobieństwo możemy za pomocą funkcji sigmoidalnej:



Funkcja sigmoidy przetwarza dowolną liczbę racjonalną do zakresu od 0 do 1 i możemy traktować wynik jako prawdopodobieństwo.

Za pomocą opisanej powyżej funkcji aktywacji dostaniemy wynik w postaci prawdopodobieństwa i dlatego możemy ją używać dla dowolnej problemu klasyfikacji. Na przykład do aproksymacji funkcji logicznych, typu „I” i „LUB”, albo do klasyfikacji na zbiorze danych „Iris”. Problemy te, natomiast, różnią się poziomem złożoności. Problem logicznej funkcji „I” może być rozwiązany nawet za pomocą funkcji liniowej, tak samo jak i funkcja „LUB”, czyli dla aproksymacji możemy użyć nawet jednego neuronu. Klasyfikacja na zbiorze danych „Iris” już nie jest tak prostym przypadkiem, niema możliwości osiągnięcia sensownych wyników za pomocą tylko jednego neuronu. W tym przypadku potrzebujemy użyć kilku. Powstaje pytanie: jak połączyć neurony?

Na dzień dzisiejszy, strukturę typowej sieci neuronowej można przedstawić w następujący sposób:

Na tym obrazie neurony są przedstawione w postaci kół, a połączeniami pomiędzy nimi, strzałkami, jest wynik funkcji jednego neuronu i parametry wejściowe innego. Jak widać z obrazku, neurony są połączone w grupy, oznaczone jednym kolorem. Takie grupy nazywają się warstwami i są standardowym sposobem grupowania neuronów w sieci neuronowej. W prostych przypadkach, jak na przykład, w przypadku który rozglądamy, warstwy są umieszczone sekwencyjnie, a pomiędzy warstwami, wszystkie wyjścia neuronów z poprzedniej warstwy są połączone ze wszystkimi wejściami neuronów następnej. W ten sposób możemy uprościć obliczenia zachodzące w sieci neuronowej za pomocą wektoryzacji wykonywanych obliczeń. Mówiąc inaczej, możemy traktować warstwy jako wektory matematyczne wykonując iloczyn skalarny pomiędzy dwoma warstwami zamiast obliczać funkcję liniową dla każdego neuronu. Z perspektywy matematycznej dwa podejścia się nie różnią, natomiast z perspektywy obliczeń komputerowych oddajemy sterowanie maszynie zamiast sterować całością procesu co pozwala na znaczącą optymalizację pośrednictwem kompilatora.

Dla tego, żeby wiedzieć w którą stronę zmieniać wagi potrzebujemy określić na ile predykcja sieci jest błędna. W tym celu można skorzystać z funkcji błędu. Istnieją różne funkcji błedu i ich użycie zależy od rozwiązywanego problemu. Można porównać dwie proste i najbardziej często używane funkcje: funkcja binarnej krosentropii

i funkcja średniokwadratowa.

Funkcja krosentropii binarnej wykorzystuje się do klasyfikacji. Wartość obserwowana jest obserwowanym prawdopodobieństwem dla danej klasy *i* i we większości przypadków jest równa 0 albo 1, dlatego że we większości przypadków jesteśmy pewni w prawdziwości tej wartości. A wartość przewidywana jest prawdopodobieństwem od 0 do 1.

Funkcja średniokwadratowa jest wykorzystywana dla problemów regresji, w danym przypadku, jednej zmiennej.

Dla tego, żeby przyśpieszyć wykonywanie obliczeń w momencie zmiany wag możemy skorzystać z wektoryzacji poprzez dodanie kolejnego wymiaru do wektorów i uśredniania wartości błędu. Kolejny wymiar będzie zawierać *b* próbek w porównaniu do poprzednio wykorzystywanej jednej. Za pomocą uśredniania błędu możemy uniknąć potrzeby aktualizacji wag dla każdej próbki, co może istotnie przyśpieszyć uczenie dużych sieci neuronowych. Taka praktyka jest wykorzystywana bardzo powszechnie, ale warto pamiętać, że duże wartości *b* mogą negatywnie wpływać na proces optymalizacji []. Wartość *b* jest często nazywana w literaturze angielskiej jako „batch size”, a uśredniony błąd jako strata (ang. „loss”).

Mając miernik trafności (albo błędności) predykcji możemy rozpocząć proces optymalizacji wag. Pierwszy problem, który napotkamy, jest to, że obliczony błąd mamy w postaci jednej wartości, a ilość wag, które chcemy optymalizować, jest większa. Wagi, w pewnym sensie, są wartościami wejściowymi do funkcji warstwy, od nich zależy wynik funkcji warstwy. Dla tego, żeby wiedzieć na ile zmieniać każdą wagę w każdej warstwie trzeba wiedzieć na ile ta waga wpłynęła na wynik predykcji. W tym celu można skorzystać z pochodnej częściowej, obliczenie której dla danych wejściowych, daje nachylenie funkcji, które wskazuje w jakim kierunku trzeba zmieniać zmienną dla najszybszego zwiększenia wartości funkcji. Jeśli zmieniać wagi na pomnożony na minus jeden wynik pochodnej to wynik funkcji będzie się zmieniać w stronę lokalnego minimum. Za jej pomocą jesteśmy w stanie minimalizować błąd, który jest wynikiem funkcji sieci neuronowej, i tym samym uczyć model.

Żeby móc wykorzystać pochodną trzeba przedstawić sieć neuronową w postaci funkcji. Liniową część warstwy można zapisać jak:

Funkcja aktywacji zależy od sytuacji i będzie zapisana jako funkcja . Wtedy funkcja warstwy będzie wyglądać następująco:

W przypadku sieci z liczbą warstw większą od 1 trzeba uwzględnić każdą warstwę. Biorąc pod uwagę strukturę sieci wspomnianą wyżej funkcja dla dowolnej liczby warstw jest kompozycją funkcji warstw. Oprócz funkcji warstw trzeba uwzględnić funkcję straty.

We większości przypadków sieć będzie zawierała więcej niż jedną wagę. Będziemy obliczać pochodną częściową dla każdej wagi i możemy te pochodne umieścić w wektorze. Taki wektor pochodnych nazywa się gradientem funkcji.

Obliczając gradient wiemy na ile zmienić wagi dla podanych parametrów niezależnych, czyli, danych wejściowych do modelu. Dla jednego przykładu jest łatwo zoptymalizować wagi, ale niestety, celem nie jest optymalizacja sieci dla jednego przypadku, tylko dla całego zbiory danych. Ze zwiększeniem rozmiaru zbioru danych rośnie i złożoność funkcji, którą aproksymuje sieć; zwiększa się liczba lokalnych minimum i jest trudniej zoptymalizować wagi, natomiast zwiększa się ilość informacji, której może się nauczyć sieć.

Jeśli zmieniać wagi sieci na całość gradientu to

Zbiór danych często może się nie mieścić w pamięci RAM. W takim przypadku można uczyć sieć iteracjami, podając dane częściami. Taka część danych w literaturze angielskiej nazywają się „batch”. Oprócz tego, że dane mogą nie mieścić się w pamięci RAM, podawać dane częściami zaleca się w celu usprawnienia działania modelu, polepszenia generalizacji [].

Todo: add julia optimizations description

# Część praktyczna

# Projektowanie

Ważną częścią projektowania systemy informatycznego jest analiza przypadków użycia narzędzia. Celem aplikacji jest udostępnianie możliwości dokonania predykcji na danych użytkownika za pomocą sieci neuronowej. Dla osiągnięcia tego celu są niezbędne takie funkcjonalności jak wczytanie danych i pobranie predykcji. Oprócz tego aplikacja musi zawierać możliwość uczenia sieci neuronowej, ponieważ implementowany serwis nie będzie zawierać możliwości wczytania własnego modelu albo skorzystania z już zaimplementowanych. Z tego wynika, że w implementowanym systemie jedynym przypadkiem użycia będzie „Przeprowadzenie predykcji” poszerzony o „Wczytanie danych”, „Uczenie modelu” i zawierający „Pobranie predykcji”.

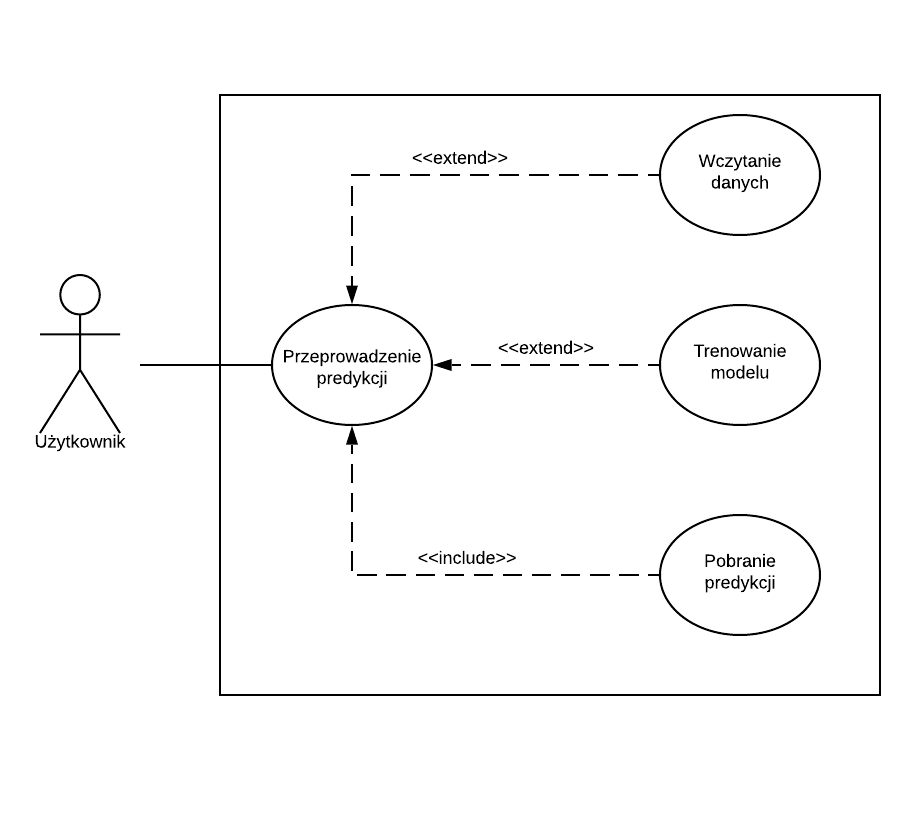
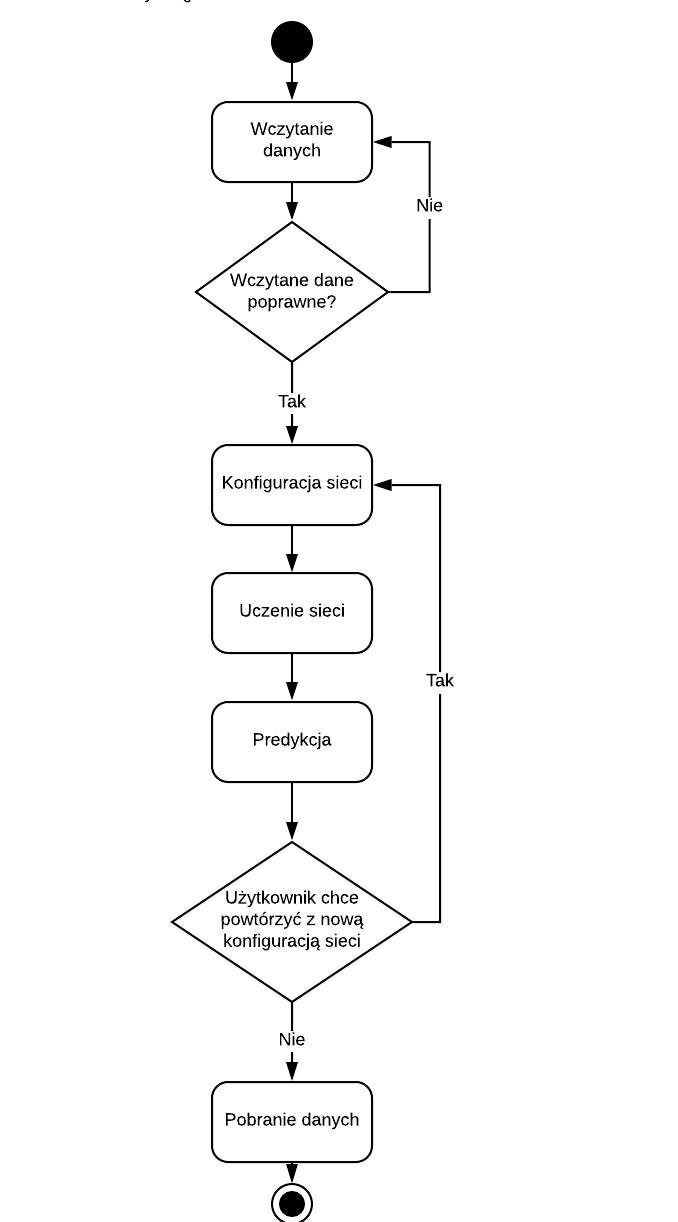


Diagram aktywności dla systemu wygląda następująco:

Rysunek 1

Pierwszym krokiem jest wczytanie danych, do uczenia sieci i na których będzie wykonana predykcja. Ponieważ wszystkie funkcjonalności udostępniane poprzez aplikację są wykonywalne z użyciem najprostszego możliwego formatu danych i ponieważ poprawne względem aplikacji dane jest łatwo przeformatować do formatu CSV, aplikacja będzie wspierała tylko CSV jako format danych wejściowych. Ważnym elementem jest walidacja wczytanych danych polegająca na następujących czynnościach:

* Walidacja poprawności według formatu CSV
* Walidacja typów danych w kolumnach
* Wizualna walidacja wczytanych danych poprzez użytkownika

Konfiguracja sieci jest niezbędną częścią dla uzyskania jak najlepszych wyników predykcji. W sieci neuronowej jest mnóstwo konfigurowalnych elementów Wszystkie najbardziej powszechne były opisane w części teoretycznej i będą dostępne w aplikacji. Konfiguracja sieci będzie zawierała następujące konfigurowalne elementy:

* Ilość warstw
* Typy warstw
* Liczba neuronów w każdej warstwie
* Wskaźnik uczenia się
* Funkcja straty
* Liczba próbek w „batchu”
* Ilość iteracji

Po próbie nauczenia sieci i sprawdzenia wyników predykcji w przypadku, jeśli wyniki będą niezadowalające użytkownik będzie chciał nauczyć sieć ponownie z innymi parametrami. Dla tego będzie udostępniona możliwość nauczenia sieci ponownie w innej konfiguracji na wczytanych wcześniej danych.

Predykcja będzie wykonana na danych wczytanych przez użytkownika. Dla zapisu predykcji, jak i do wczytania danych wejściowych, można użyć najprostszego formatu, ze względu na to do eksportu będzie użyty format CSV.

## Zakończenie

## Literatura

<https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>

<https://arxiv.org/abs/1209.5145>

<https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>

<https://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>

<https://arxiv.org/pdf/1811.01457.pdf>

<https://arxiv.org/pdf/1902.02376.pdf>

<https://julialang.org/blog/2017/12/mlpl/>

<https://github.com/mila-iqia/myia>

<https://github.com/tensorflow/swift/blob/master/docs/WhySwiftForTensorFlow.md>

<https://github.com/PaddlePaddle/Paddle/>

<http://galaxy.agh.edu.pl/~vlsi/AI/bias/bias_eng.html>

<https://wsiz.rzeszow.pl/wp-content/uploads/2019/10/2019_45_Z-PRACE-DYPLOMOWE_2019-20.pdf>

<https://arxiv.org/abs/1609.04836>

Spis rysunków

# Streszczenie pracy

**Wyższa Szkoła Informatyki i Zarządzania z siedzibą w Rzeszowie**

**Kolegium Informatyki Stosowanej**

**Streszczenie pracy dyplomowej**

Realizacja projektu aplikacji webowej do przeprowa-dzenia obliczeń za pomocą sztucznych sieci neuronowych

**Autor:** Bohdan Turanyi

**Promotor:** dr. Inż. Mariusz Wrzesień

**Słowa kluczowe:**

Praca dyplomowa jest

# Załączniki

Płyta CD