Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki,
Informatyki i Inżynierii Biomedycznej

Projekt dyplomowy

Wykorzystanie sieci neuronowych do rozpoznawania statycznych gestów dłońmi do sterowania robotem mobilnym z wykorzystaniem kamery.

Using neural networks to recognize hand static gestures to control a mobile robot using a camera.

Autor:	Sylwester Dawida
Kierunek studiów:	Automatyka i Robotyka
Opiekun pracy:	Dr hab. Adrian Horzyk, prof.AGH

Kraków, 2020

Spis treści

1. Wstęp 3

1.1 Cel i założenia projektu 4

2. Sieci neuronowe 5

2.1. Pierwowzór sztucznej sieci neuronowej 5

2.2. Model sztucznej sieci neuronowej 6

2.3. Funkcje aktywacji 8

2.4. Rodzaje sieci neuronowych 11

2.5. Trening Sieci 13

2.5.1. Funkcje błędu sieci neuronowych 14

2.5.2. Algorytmy gradientowe 15

2.5.3. Reguła delta 16

2.5.4. Algorytm wstecznej propagacji 17

2.5.5. Proces trenowania 18

2.5.6. Uczenie głębokie 20

3. Robot mobilny Arlo 21

4. Komunikacja z robotem 22

5. Implementacja projektu 25

5.1. Przegląd bibliotek użytych w projekcie 25

5.2. Implementacja komunikacji 28

5.2.1. Opis komend sterujących robotem 29

5.3. Program robota 30

5.4. Implementacja sieci neuronowej 31

5.4.1. Struktura sieci neuronowej 32

5.4.2. Trening klasyfikatora 33

5.5. Graficzny interfejs aplikacji 36

5.6. Testy sterowania robotem 37

6. Podsumowanie 39

1. Wstęp

Bardzo szybki postęp technologiczny i gwałtowny wzrost mocy obliczeniowej procesorów w ostatnich dekadach wpłynął na rozwój m.in. nauk związanych ze sztuczną inteligencją. Jedną z dziedzin tej nauki jest uczenie maszynowe (ang. *machine learning*). Pojęcie uczenia maszynowego zostało po raz pierwszy wprowadzone w 1959 r. przez Arthura Samuela w artykule naukowym pt. *"Some Studies in Machine Learning Using the Game of Checkers"*. Autor pracując w IBM zaprojektował pierwszy program potrafiący grać w szachy. Algorytmy uczenia maszynowego wykorzystują modelowanie statystyczne. W program wprowadza się dużą ilość informacji uczących, a na podstawie cech wyodrębnionych z poszczególnych danych, aproksymowany jest model, który ma odzwierciedlać dany proces. Z machine learningu wywodzi się dziedzina nauk związana ze sztucznymi sieciami neuronowymi, czerpiąca inspirację z natury. Sztuczne sieci neuronowe (w skrócie SSN) jest to struktura matematyczna posiadająca zdolność "nauki". Jest to bardzo uproszczony model mózgu.

Warto wspomnieć o tym, że bardzo duży wkład w badania nad SSN w Polsce ma profesor Ryszard Tadeusiewicz, pracownik Akademii Górniczo Hutniczej. Jego prace są często cytowane w środowisku związanym z uczeniem maszynowym. Niniejsza praca również bazowała na części publikacji profesora Tadeusiewicza [4],[5]. Sztuczne sieci neuronowe są obecnie używane na wielu płaszczyznach. W niniejszej pracy zostały użyte w roli klasyfikatora statycznych gestów dłoni. Klasyfikacja obiektów jest typowym zadaniem, przy którym wykorzystuje się sztuczne sieci neuronowe. Do analizowania obrazów najlepiej przystosowane są sieci konwolucyjne. Jest to rodzaj SSN, który specjalizuje się w rozpoznawaniu obiektów na podstawie obrazu. Rozpoznawanie następuje poprzez wyodrębnianie oraz analizowanie charakterystycznych cech poszukiwanych struktur, takich jak specyficzny kolor, kształt itp. Aplikacja ma tworzyć interfejs do sterowania robotem za pomocą gestów dłoni, które będą rozpoznawane za pomocą kamerki internetowej. Dane z kamery będą przesyłane do głównego skryptu rozpoznającego symbole, gdzie zostaną przetworzone w komendy oraz wysłane do robota mobilnego Arlo.

1.1. Cel i założenia projektu

Głównym celem pracy inżynierskiej jest napisanie oprogramowania dla robota mobilnego Arlo oraz kamery, by móc sterować jego ruchami przy pomocy statycznych gestów dłoni. Zakres prac obejmuje zbudowanie i wytrenowanie sieci neuronowych rozpoznających gesty oraz przełożenia tego na sterowanie robotem.

Prace nad projektem można podzielić na dwie główne części. Pierwszą z nich jest stworzenie aplikacji desktopowej, która będzie miała za zadanie rozpoznawać gesty sterujące robotem. Program zostanie napisany w języku skryptowym Python 3 i będzie wykorzystywać głębokie sieci neuronowe do przetwarzania sygnału wizyjnego z kamerki internetowej laptopa. Drugą częścią pracy nad projektem jest zaprogramowanie robota mobilnego. Układ sterujący robotem Arlo programuje się w języku C, przy pomocy dedykowanego środowiska pracy SimpleIde. Producent dostarcza kilkanaście bibliotek do obsługi robota, co znacząco upraszcza programowanie robota.

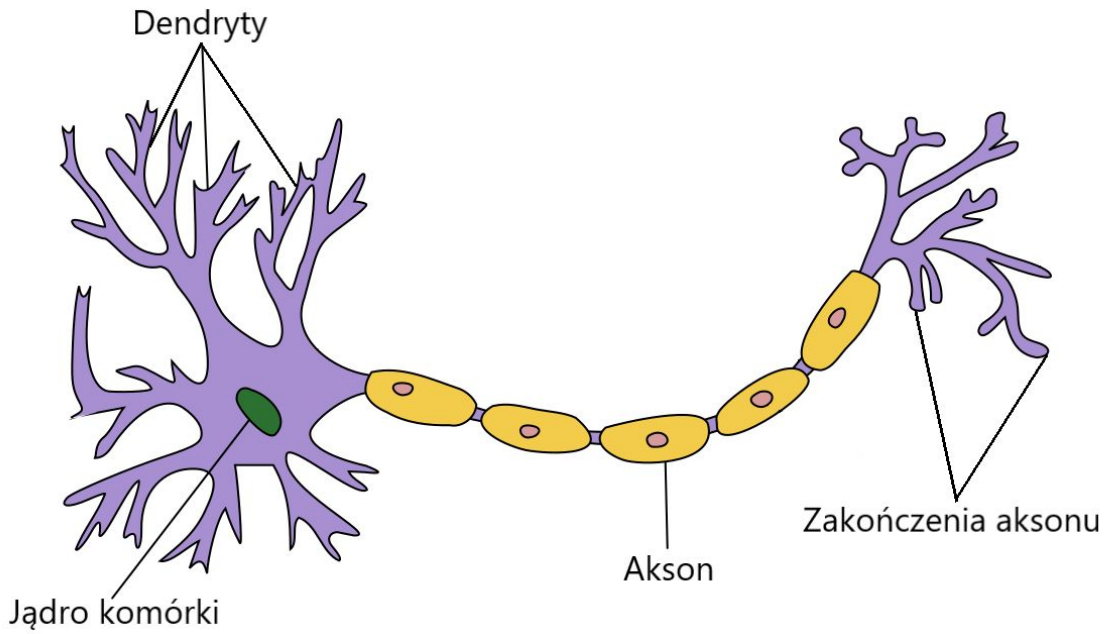
Dodatkowym zadaniem jest zapewnienie komunikacji pomiędzy robotem i laptopem, zostanie użyty do tego celu standard komunikacji bezprzewodowej Bluetooth 4.0 low energy. Robot mobilny nie posiada w podstawowej wersji możliwości komunikacji bezprzewodowej, dlatego jego peryferia zostaną poszerzone o dodatkowy moduł Bluetooth. Rozpoznane komendy będą przesyłane autorskim protokołem do mikrokontrolera robota Arlo. Dane pochodzące z kamerki internetowej mają być przetwarzane w czasie rzeczywistym - jest to główne założenie pracy. Ponadto autor będzie dążył w trakcie treningu sieci neuronowych do jak najlepszej trafności w rozpoznawaniu komend.

2. Sieci neuronowe

Sztuczne sieci neuronowe są to modele matematyczne imitujące działanie mózgu. Wielu problemów nie da się rozwiązać przy użyciu klasycznych algorytmów lub jest to zbyt trudne w implementacji. Z pomocą przychodzą SSN, które działaniem przypominają pracę mózgu. Bardzo dobrze sprawdzają się w rozwiązywaniu podobnych problemów, z którymi radzi sobie ludzki mózg: rozpoznawanie obiektów na obrazie, rozpoznawanie mowy, dźwięków, sterowanie botami w grach komputerowych czy analizowanie obrazów z tomografów komputerowych w poszukiwaniu nowotworów. Sieci neuronowe są szczególnie przydatne w zadaniach, przy których wyznaczenie modelu matematycznego procesu jest nietrywialne. Jest to jedna z największych zalet SSN. Wystarczy odpowiednie zaprojektowanie struktury oraz poprawny trening, a sieć sama będzie dążyć do idealnego odwzorowania procesu. Spektrum działania sztucznych sieci neuronowych jest szerokie i zważywszy na dynamiczny wzrost mocy obliczeniowej komputerów wciąż się poszerza [2].

2.1. Pierwowzór sztucznej sieci neuronowej

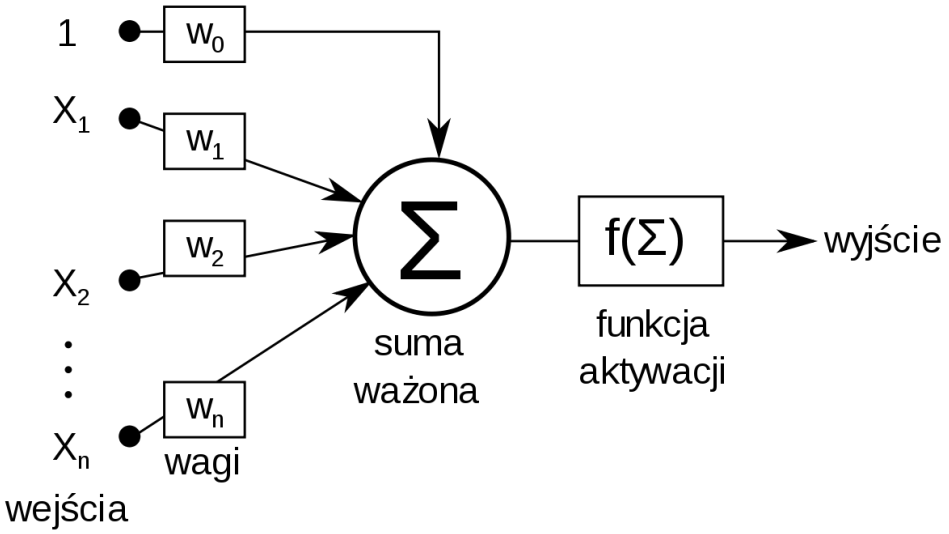
Jak wspomniano, pierwowzorem SSN jest mózg, a podstawową jednostką pozwalającą na przetwarzanie informacji jest neuron. Z perspektywy inżyniera komórka ta jest funkcją wielu zmiennych. Neuron składa się z dendrytów, jądra komórkowego, aksonu i synaps, czyli połączeń międzykomórkowych. Każdy neuron jest umieszczony w rozległej sieci neuronowej. Elektryczne sygnały pochodzące z innych komórek zostają przekształcone w sygnał chemiczny w synapsach, gdzie następuje ich wzmocnienie o odpowiednie współczynniki, które są modyfikowane w reakcji na bodźce zewnętrzne, czyli naukę. Następnie sygnały przedostają się do dendrytów, gdzie w postaci elektrycznej trafiają do wnętrza komórki (rys.2.1.). Wszystkie przeskalowane wejścia są sumowane i jeżeli suma jest większa od ustalonego progu, to neuron przesyła impuls do aksonu, gdzie następuje przekazanie sygnału do sąsiednich neuronów przez synapsy - schemat się powtarza. Sygnał rozchodzi się po sieci przechodząc przez kolejne komórki. Siła neuronów leży w ich ilości, kilka komórek jest bezwartościowych, lecz kilka milionów jest w stanie nauczyć się wielu wzorców. Szacuje się, że ludzki mózg posiada 150 miliardów neuronów oraz 100 bilionów synaps. [16][17]



Rysunek 2.1.: Komórka neuronu

2.2. Model sztucznej sieci neuronowej

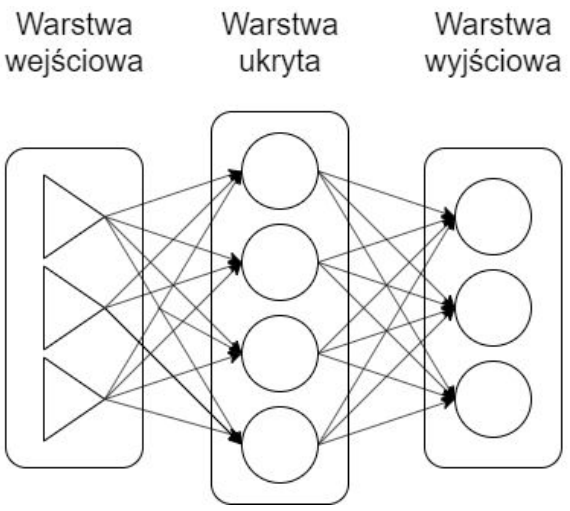
Podstawą modelu matematycznego SSN jest sztuczny neuron. Posiada on wektor wejść X_n, wyjście y, wejście specjalne zwane biasem g, wektor wag połączeń wejściowych w oraz funkcję aktywacji f [1]. Bias jest często traktowany jako element wektora wag, do którego podawana jest stała niezależna liczba jeden i ma za zadanie lekko modyfikować funkcję aktywacji (rys.2.2.). Wejścia X neuronu są mnożone przez odpowiadające im wartości z wektora wag, a następnie wynik jest sumowany, tworząc tak zwane pobudzenie neuronu. W ostatniej fazie suma jest przekazywana do funkcji aktywacji. Model neuronu można przedstawić w postaci funkcji matematycznej (2.1). Odpowiednio połączone ze sobą komórki tworzą sztuczną sieć neuronową.



Rysunek 2.2.: Model matematyczny neuronu

$$y = f\left(\sum_{i=0}^{n} X_i w_i\right) \tag{2.1}$$

Neurony SSN najczęściej są zgrupowane w tak zwane warstwy. Z zasady neurony w warstwie nie są połączone ze sobą bezpośrednio, powinny być połączone z neuronami sąsiednich warstw. Każda sieć posiada minimum dwie warstwy: wejściową i wyjściową. Warstwy te tworzą interfejs zewnętrzny dla aplikacji, która wykorzystuje SSN. Warstwa wejściowa nie posiada neuronów, służy tylko do przekazywania lub przygotowania informacji do przetwarzania wewnątrz sieci, przykładowo skalowanie. Dodatkowe warstwy znajdują się wewnątrz sieci i są to tak zwane warstwy ukryte (rys.2.3.). Z perspektywy aplikacji nie ma do nich bezpośredniego dostępu. Sieć pogrupowana w warstwy jest łatwiejsza w opisie oraz jest lepiej zoptymalizowana pod względem szybkości działania. Obliczenie wyjścia z pojedynczej warstwy sprowadza się do przemnożenia wektora wejść z odpowiednio skonstruowaną macierzą wag danej warstwy. Wykorzystując odpowiednie biblioteki wspierające równoległe obliczenia na GPU (ang. graphics processing unit) można znacząco przyspieszyć działanie sieci neuronowej. Powyższy opis jest jedynie ogólną konwencją prostych sieci ułatwiającą jej działanie i dokumentację. Oczywiście istnieją wyjątki od powyższego opisu, na przykład neurony połączone z warstwami nie sąsiednimi, podłączenia rekurencyjne itp. Szerszy opis rodzajów struktur znajduje się w kolejnym rozdziale.



Rysunek 2.3.: Przykładowe warstwy sieci neuronowej

2.3. Funkcje aktywacji

Funkcje aktywacji decydują o ostatecznych możliwościach sieci oraz mają wpływ na szybkość treningu. Funkcje te są wybierane w zależności od problemu, jaki sztuczne sieci neuronowe mają rozwiązywać. Do najczęściej wybieranych należą takie funkcje, które posiadają możliwie jak najprostszą pochodną ze względu na obliczenia w trakcie nauki. Przykładami funkcji są (rys.2.4.):

- **Liniowa -** funkcja liniowa przekazująca bezpośrednio pobudzenie neuronu przeskalowana o pewną wartość. Obecnie używana sporadycznie ze względu na ograniczenia nakładane na sieć. Jeżeli wszystkie warstwy SSN posiadają liniowe funkcje aktywacji to całą sieć można sprowadzić do sieci jednowarstwowej. Wynika to z rachunków macierzowych [4], sieć taka ma bardzo niewielkie możliwości nauki i adaptacji (2.2).

$$f(x) = ax \tag{2.2}$$

- **Progowa -** funkcja z wyjściem dwustanowym {0, 1}, używana najczęściej w prostych sieciach neuronowych takich jak perceptron (2.3).

$$f(x) = \{0 : x < 0,\ 1 : x \geq 0\} \tag{2.3}$$

- **Softmax -** jest to funkcja wykładnicza, która dodatkowo normalizuje wyjścia wszystkich neuronów warstwy w taki sposób, aby ich suma była równa jeden. Funkcja

jest najczęściej umieszczana w ostatniej warstwie SSN. Softmax jest używana w problemach klasyfikacji do wyznaczenia prawdopodobieństw przynależności obiektu do danej klasy (2.4).

$$f(x) = \frac{e^{-x}}{\sum_{i=0}^{n} e^{-x\,i}} \tag{2.4}$$

x_i - kolejne pobudzenia neuronów w warstwie

- **Sigmoidalna -** jest ciągłym odzwierciedleniem funkcji progowej. Jej przewaga polega na łagodnym przejściu pomiędzy zerem a jedynką. Do niedawna bardzo często używana, dzisiaj jest wypierana przez inne bardziej efektywne funkcje aktywacji (2.5).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

- **Relu** (*ang. Rectified Linear Units*) **-** jest to obecnie najczęściej używana funkcja aktywacji. Została po raz pierwszy zaprezentowana w 2000 r., a w 2011 r. udowodniono, że zastosowanie relu w budowie sztucznej sieci neuronowej znacząco podnosi jej wydajność [11] (2.6). Pomimo wielu zalet funkcja posiada jedną poważną wadę, którą zwykło się nazywać martwym neuronem - jest to stan wag, w którym komórka zawsze zwraca 0. Rozwiązaniem problemu jest użycie modyfikacji relu takich jak: leaky Relu (2.7), parametric Relu (2.8), ELU (2.9), są to funkcje, które dla argumentu mniejszego od zera posiadają niezerowy "dryft". Gładką aproksymacją relu jest funkcja softplus (2.10). Porównanie wariacji relu jest przedstawione na rysunku 2.5.

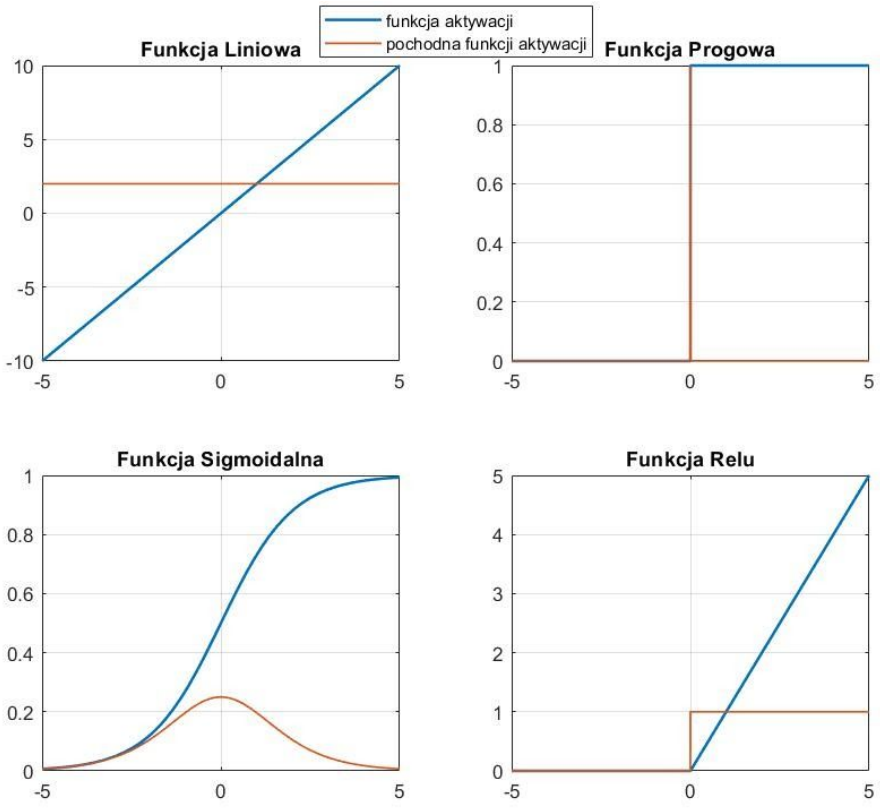
$$f(x) = max(0,\ x) \tag{2.6}$$

$$f(x) = \{0.01x\ :\ x < 0,\ x : x \geq 0\} \tag{2.7}$$

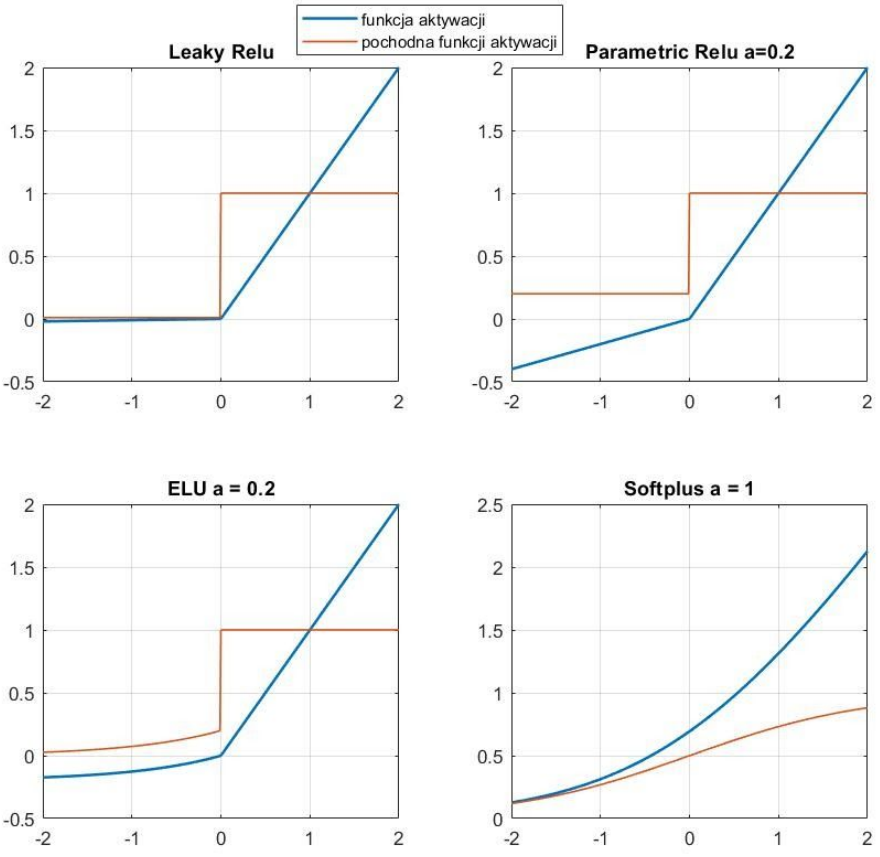
$$f(x) = \{ax\ :\ x < 0,\ x : x \geq 0\} \tag{2.8}$$

$$f(x) = \{a(e^x - 1)\ :\ x < 0,\ x : x \geq 0\} \tag{2.9}$$

$$f(x) = log(1 + e^x)a \tag{2.10}$$



Rysunek 2.4.: Wybrane funkcje aktywacji



Rysunek 2.5.: Alternatywy funkcji relu

Funkcje są najczęściej wybierane globalnie dla całej sieci, choć powszechne są również praktyki stosowania różnych funkcji dla warstw lub nawet dla poszczególnych neuronów.

2.4. Rodzaje sieci neuronowych

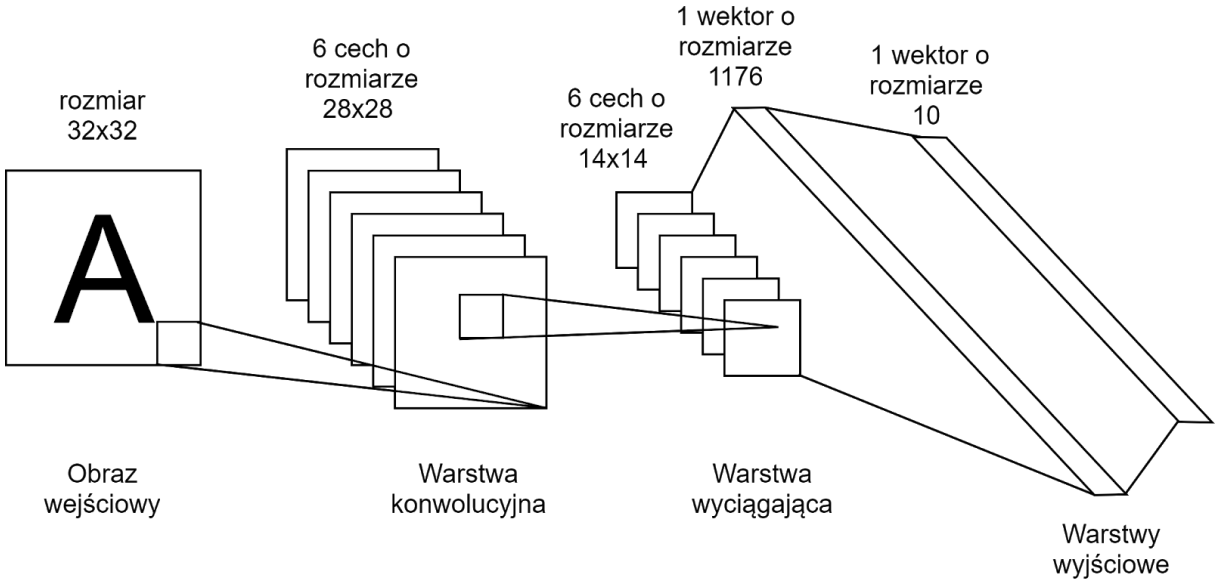
Istnieje bardzo wiele rodzajów sieci neuronowych. Dynamiczny rozwój tej dziedziny sprawia, że powstają wciąż nowe coraz bardziej skomplikowane struktury, które można usystematyzować. Podrozdział ten opisuje te najpowszechniej używane przynajmniej w roku publikacji pracy (2020 r.).

Perceptron był jednym z pierwszych struktur sieci neuronowych. Można powiedzieć, że był to kamień milowy dla nauk machine learningu. Idea perceptronu została po raz pierwszy wykorzystana przez Franka Rosenblatta i Charlesa Wightmana w maszynie, która miała za zadanie rozpoznawać znaki alfanumeryczne (1957 r.). Skonstruowane urządzenie było elektromechaniczne (wartości wag odpowiadały rezystancji potencjometrów, które były regulowane silnikami elektrycznymi) [4]. Perceptron jest najprostszą siecią neuronową i składa się z warstwy wejściowej i wyjściowej. Oznacza to, że posiada tylko jedną warstwę aktywnie biorącą udział w obliczeniach. Funkcję całej sieci można sprowadzić do rachunku macierzowego (mnożenie wejść z macierzą wag warstwy wyjściowej). Klasyczny perceptron posiada z założenia progową funkcję aktywacji. Model tej struktury sztucznej sieci neuronowej aktualnie jest rzadko stosowany ze względu na prostotę, przez którą posiada niskie możliwości nauki.

Perceptron wielowarstwowy (ang. *multilayer perceptron* - MLP) aktualnie jest to najpowszechniej używana struktura sieci neuronowej, MLP jak sama nazwa wskazuje jest rozszerzeniem perceptronu, struktura ta jest rozbudowana dodatkowo o warstwy ukryte, które znacząco zwiększają możliwości sieci neuronowej. Funkcje aktywacji tej struktury są zazwyczaj nieliniowe, według badań z końca ubiegłego wieku MLP posiadający minimalnie trzy warstwy z nieliniowymi funkcjami aktywacji może aproksymować dowolne odwzorowanie [6]. Przykład perceptronu z jedną warstwą ukrytą znajduje się na rysunku 2.3.

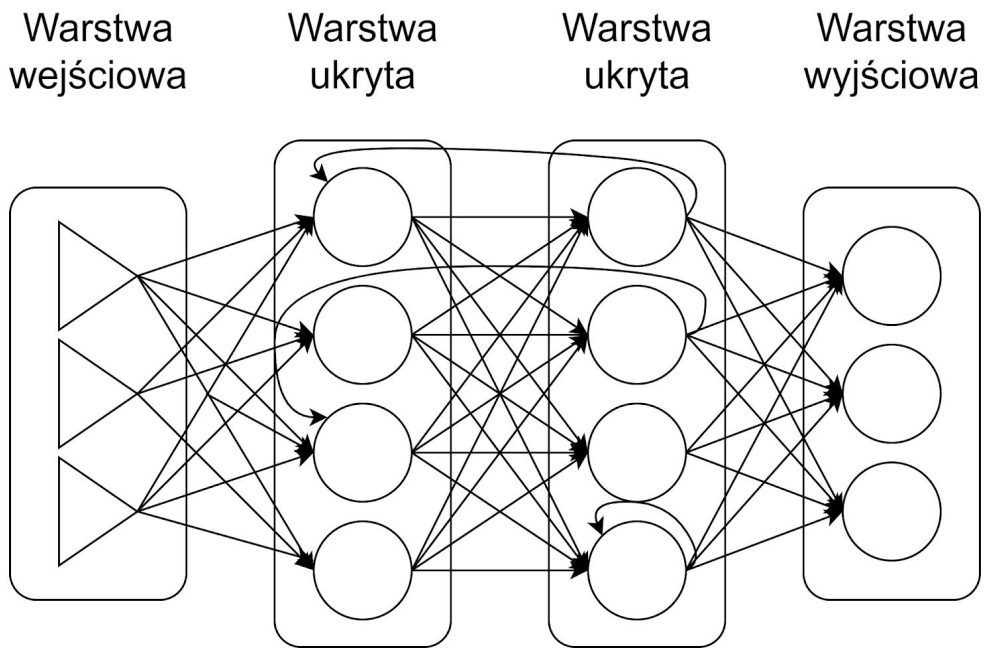
Sieć konwolucyjna (ang. *convolutional neural network*) sieć najczęściej wykorzystywana do klasyfikacji obrazów. Wykorzystuje warstwy konwolucyjne, które przeprowadzają operację splotu na obrazie [3]. Warstwa konwolucyjna zawiera kilka filtrów

zwanych maskami. Każda z nich ma za zadanie wykrywać właściwość obrazu. Współczynnikami, które się modyfikuje w trakcie nauki są tym razem poszczególne wartości liczbowe w maskach. Maski mogą być dwuwymiarowe dla obrazów w odcieniach szarości lub trójwymiarowe w przypadku obrazów RGB. Kolejną warstwą używaną w sieciach konwolucyjnych jest warstwa wyciągająca (ang. *polling layer*). Ma ona na celu zmniejszenie rozmiarów danych do dalszego przetwarzania. Warstwa wyciągająca wybiera najistotniejsze informacje z pewnych obszarów obrazów i przekazuje je do dalszej obróbki w głąb sieci. Obszarem jest najczęściej okno 2x2 a przykładowymi funkcjami "wyciągającymi" informację są: maximum i mediana. Okno 2x2 pozwala na zredukowanie rozmiaru danych czterokrotnie, co pozytywnie wpływa na szybkość działania całej sieci neuronowej. Przykładowa sieć konwolucyjna jest przedstawiona na rysunku (2.6.). Pierwsza warstwa konwolucyjna zmniejsza obraz o 4 piksele dlatego, że maska konwolucyjna ma rozmiar 5x5 i przy krawędziach obrazu z każdej strony odrzucane są po 2 piksele (brak paddingu). Następnie w warstwie wyciągającej właściwości obrazu są zmniejszane czterokrotnie, wynika to z zastosowania maski 2x2. W ostatnich liniowych warstwach następuje tak zwane "wypłaszczenie" danych - pierwsza warstwa posiada rozmiar 14*14*17 czyli 1176. Ostatnia warstwa podaje na wyjście już całej sieci 10 wartości. Może to być przykładowo rozkład prawdopodobieństwa w problemie klasyfikacji. W przypadku sieci konwolucyjnych bardzo ważne jest śledzenie wymiarów danych w poszczególnych etapach przetwarzania informacji.



Rysunek 2.6.: Przykładowa sieć konwolucyjna

Sieć rekurencyjna - jest to struktura wykorzystująca sprzężenia zwrotne, co znacząco komplikuje trenowanie takiej sieci neuronowej. Sprzężenia sprawią, że stan sieci nie zależy tylko od wejścia systemu, ale i od jego wyjścia [5]. Sieci rekurencyjne idealnie nadają się do przetwarzania sekwencji danych, przykładowo: analiza obrazu video, przetwarzanie dźwięków. Sprzężenia zwrotne sprawiają, że sieci te mają bardzo duże pojemności (pojemność (*ang. capacity*) określa potencjalne zdolności nauki, jest to cecha sieci neuronowej zależna od jej struktury, konkretnie jest to ilość informacji, jakie sieć może w sobie pomieścić, pojemność jest proporcjonalna do ilości neuronów sieci), co z kolei wpływa na duży potencjał nauki. Przykładowa sieć rekurencyjna może posiadać wiele sprzężeń zwrotnych. Mogą łączyć wyjście neuronu z wejściem tej samej komórki lub sąsiedniej pochodzącej nawet z innej warstwy (rysunek 2.7.).



Rysunek 2.7.: Przykładowa sieć rekurencyjna

2.5. Trening Sieci

Uczenie sieci neuronowych polega na iteracyjnej modyfikacji współczynników poszczególnych warstw celem lepszej aproksymacji modelu, który sieć ma odzwierciedlać. Metody uczenia sieci neuronowych można podzielić na:

Uczenie nadzorowane jest to metoda wykorzystująca fakt, że w większości przypadków modelowania dla danych wejściowych systemu znane jest pożądane wyjście. W trakcie nauki do sieci neuronowych wprowadza się dane, a następnie porównuje się odpowiedź SSN z oczekiwanym rezultatem. Posiadając odpowiedź rzeczywistą sieci oraz tą wzorcową można wyznaczyć błąd wyjścia wybraną funkcją kosztu (ang. *loss function*) i zaktualizować odpowiednim algorytmem współczynniki poszczególnych warstw sieci [8].

Uczenie nienadzorowane silnie polega na doborze odpowiedniej struktury sieci neuronowej. W tej metodzie użytkownik trenujący sieć nie posiada wiedzy na temat procesu, tzn. nie wiadomo, jakich rezultatów oczekuje się od danych wejściowych, jest to przeciwieństwo uczenia nadzorowanego. W algorytmach uczenia nienadzorowanego sieć stara się nauczyć sama modelu. Podając jej kolejne dane, polega się na tym, że sieć sama odgadnie naturę problemu, dlatego odpowiednie skonstruowanie sieci w tym przypadku jest tak ważne.

Uczenie z krytykiem, jest to rodzaj treningu, w którym analogicznie do uczenia nienadzorowanego, użytkownik trenujący sieć nie posiada wystarczającej wiedzy na temat modelowanego procesu, aby jasno określić błąd sieci. Z tą różnicą, że można jedynie odpowiedzieć na pytanie, czy wynik sieci odpowiadający poszczególnym danym wejściowym jest satysfakcjonujący, czy nie - jest to ocena binarna. Uczenie z krytykiem można umieścić pomiędzy uczeniem nienadzorowanym a nadzorowanym, gdyż istnieje jakakolwiek informacja zwrotna, co nie występuje w przypadku uczenia nienadzorowanego. Z drugiej strony ocena wyjścia jest na tyle szczątkowa, że nie możemy zakwalifikować tego uczenia do nadzorowanego.

2.5.1. Funkcje błędu sieci neuronowych

Funkcje błędu SSN (znane również jako funkcje kosztu lub celu) służą algorytmom trenującym, ocenie poprawności działania modelu. Funkcje przyjmują dwa argumenty aktualne y oraz oczekiwane z wyjście sieci [5]. Jedną z powszechniej używanych funkcji kosztu jest błąd średniokwadratowy (2.11), warto wspomnieć że istnieje wiele opracowanych funkcji błędu, które specjalizują się w konkretnych problemach, które sieci neuronowe mają rozwiązywać.

$$F(z,y) = \frac{1}{N}\sum_{i=0}^{N}(z_i - y_i)^2 \tag{2.11}$$

2.5.2. Algorytmy gradientowe

Popularnymi algorytmami używanymi w procesie nadzorowanego trenowania sieci neuronowych są między innymi algorytmy gradientowe. Bazują one na rozwinięciu pewnej funkcji błędu popełnianej przez sieć w szereg Taylora (2.12)

$$F(w^0 + \Delta w) = F(w^0) + \Delta w^T \nabla F(w^0) + O_1(w^0, \Delta w) \tag{2.12}$$

$$F(w^0 + \Delta w) \approx F(w^0) + \Delta w^T \nabla F(w^0) \tag{2.13}$$

$$F(w^0 + \Delta w) - F(w^0) \approx \Delta w^T \nabla F(w^0) \tag{2.14}$$

F - funkcja błędu sieci jest również zależna od wektora wszystkich wag sieci w,

w^0 - punkt w otoczeniu którego tworzy się rozwinięcie Taylora,

Δw - otoczenie punktu, krok algorytmu,

O_1 - reszta rozwinięcia Taylora.

Kolejne wagi wygenerowane przez algorytm muszą lepiej minimalizować funkcję błędu. Oznacza to, że wyrażenie 2.14 musi być ujemne. Dzięki jego prawej stronie można dojść do wniosku, że najlepszy krok Δw jest równoległy do gradientu $\nabla F(w^0)$, ale posiadający przeciwny zwrot (w takim przypadku wyrażenie po prawej stronie przyjmuje najniższą możliwą wartość), jest to kierunek antygradientu, czyli tak zwanego najszybszego spadku funkcji w danym punkcie. Ostateczny wzór, na którym bazują wszystkie algorytmy gradientowe, podany jest poniżej (2.15).

$$\Delta w_{ij} = -\eta \frac{dF}{dw_{ij}} \tag{2.15}$$

w_{ij} - waga wejściowa j neuronu i,

η - jest to współczynnik nauki, pozwala kontrolować proces trenowania sieci, czyli zapobiega utknięciu treningu w lokalnym minimum funkcji błędu. Jedną ze strategii doboru tego współczynnika jest rozpoczęcie szybkiej nauki z wysokim η a w kolejnych fazach stopniowe obniżanie tego współczynnika.

Wzór 2.15. stanowi podstawę wszystkich algorytmów gradientowych, takich jak metoda batch gradient descent, która aktualizuje wagi po przetworzeniu wszystkich danych treningowych. Stochastic gradient descent - ten algorytm z kolei wyznacza nowe wartości wektora w dla każdej danej treningowej z osobna. Mini batch gradient descent - natomiast ten optymalizator generuje nowe wektory dla tak zwanych batchy (batch jest złączeniem najczęściej od kilku do kilkunastu danych treningowych w jedną paczkę informacji celem przyspieszenia procesu trenowania).

2.5.3. Reguła delta

Reguła delta jest metodą aktualizacji wag poszczególnych komórek. Algorytm zakłada, że oczekiwana odpowiedź z_i każdego neuronu w sieci jest znana, czyli jest to algorytm uczenia nadzorowanego. Na podstawie z_i można obliczyć błąd wyjścia. Do oceny poprawności całej sieci służy funkcja kryterialna (2.16).

$$Q(\delta) = 0.5 \sum_{i=0}^{N} (z_i - y_i)^2 \tag{2.16}$$

gdzie $z_i - y_i$ traktujemy jako błąd odpowiedzi neuronu δ_i

W trakcie nauki dąży się do minimum funkcji 2.16. Błąd δ można wyrazić jako $z_i - f_i(X_i, w_i)$, wynika to wprost ze wzoru 2.1. W dalszych rozważaniach rachunki będą przeprowadzane na Q_i, czyli funkcji obliczającej błąd pojedynczego neuronu sieci, gdyż minimalizacja sumy 2.16 jest minimalizacją jej poszczególnych składników. W trakcie kolejnych iteracji treningu nowe wartości wag będą szukane na kierunku antygradientu 2.17. funkcji Q_i.

$$\Delta w_{ij} = -\eta \frac{dQ_i}{dw_{ij}} \tag{2.17}$$

w_{ij} - pojedyncza waga wejściowa neuronu i,

η - jest to współczynnik nauki.

Funkcja Q_i jest złożona więc z reguły łańcuchowej różniczkę (2.17) można rozłożyć trzy części:

$$\frac{dQ_i}{dw_{ij}} = \frac{dQ_j}{dy_i}\frac{dy_i}{dw_j} = \frac{dQ}{dy_i}\frac{dy_i}{dp_i}\frac{dp_i}{dw_{ij}}$$

$$\frac{dQ_i}{dy_i} = -(z_i - y_i) = -\delta_i$$

$$\frac{dp_i}{dw_{ij}} = X_{ij}$$

p_i - jest pobudzeniem neuronu i $(\sum_{j=0}^{N} X_{ij} w_{ij})$,

$\frac{dy_i}{dp_i}$ - jest różniczką funkcji aktywacji $\frac{df_i}{dp_i}$.

Finalny wzór aktualizacji wagi w_{ij} neuronu i regułą delta (2.18) [8]

$$w'_{ij} = w_{ij} + \eta\delta_i \frac{df_i}{dp_i} X_{ij} \tag{2.18}$$

w'_{ij} - nowa wartość wagi neuronu.

Algorytm delty w praktyce nie nadaje się do zastosowania w przypadku sieci wielowarstwowych, gdzie w warstwach ukrytych nie dysponuje się informacją o poprawnej odpowiedzi neuronów. Reguła delta ogranicza się tylko do jednowarstwowych perceptronów.

2.5.4. Algorytm wstecznej propagacji

Algorytm wstecznej propagacji jest aktualnie najczęściej używanym algorytmem aktualizacji wag w procesie uczenia z nadzorem. Wykorzystuje błąd warstwy wyjściowej wyznaczony w funkcji kosztu i jak sama nazwa wskazuje, na jego podstawie modyfikuje wagi wszystkich warstw [8]. Jest więc rozwiązaniem problemu reguły delty, gdzie nie było możliwości weryfikacji działania neuronów ukrytych. Błąd jest propagowany przeciwnie do przepływu informacji w sieci neuronowej i przechodząc przez kolejne neurony w warstwach sieci jest odpowiednio modyfikowany według wzoru (2.19). Wzór pozwala na obliczenie błędu przypadające każdemu neuronowi w SSN.

$$\delta_i = \sum_{j=1}^{n} \delta_j w_j \tag{2.19}$$

δ_i - wartość błędu przypadająca na komórkę i,

δ_j - wartość wyznaczonego już błędu bezpośrednio połączonej komórki j z neuronem i,

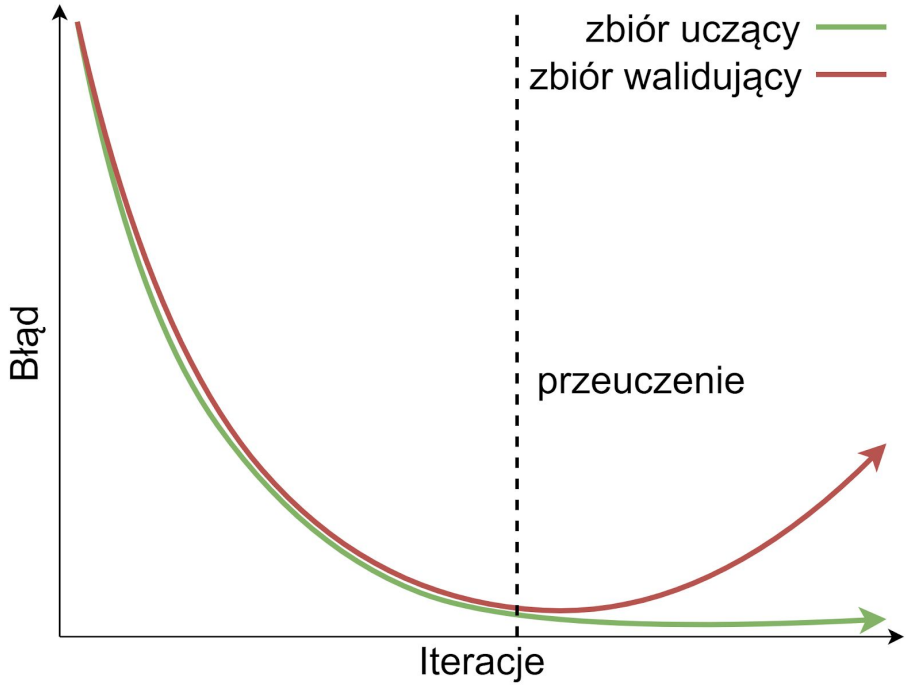
w_j - waga połączenia neuronu i z j.

Nowa wartość błędu neuronu δ_i jest obliczana przy pomocy błędów już wyznaczonych w kolejnej warstwie. Komórki j są kolejnymi bezpośrednio połączonymi neuronami z komórką i. Błędy δ_j są mnożone przez wagi odpowiadające połączeniu pomiędzy neuronami a następnie sumowane. Gdy wyznaczone zostaną wszystkie błędy w danej sieci, następuje zaktualizowanie wag neuronów według reguły delta (2.18).

2.5.5. Proces trenowania

Praca z SSN odbiega lekko od idei klasycznego programowania. Budując strukturę sieci ważne jest wyczucie, należy dostosować wielkość, konstrukcję i elementy sieci do postawionego problemu. Trening przypomina "naukę idealnego ucznia". Należy uważać na tak zwane przetrenowanie, gdzie sieć "wyucza się na pamięć" danych treningowych i błędnie reaguje na nowe. Przetrenowanie polega na zbyt dokładnym dopasowaniu modelu do danych które zostały wprowadzone do sieci. Sprawia to, że nowe dane są błędnie interpretowane, zmniejsza to zdolności generalizacji sieci [7]. Istnieje rozwiązanie tego problemu, według którego należy odpowiednio przygotować zbiór danych, czyli podzielić w sposób losowy na części: uczącą, walidującą oraz testową. W przypadku klasyfikacji ponadto należy zachować balans. Każda część powinna posiadać procentowo taką samą ilość danych z poszczególnych klas, poprawia to znacząco rezultaty nauki. Część ucząca jest używana podczas treningu, bierze aktywny udział w modyfikacji sieci, dlatego ten zbiór powinien być największy. W trakcie nauczania co kilkanaście iteracji należy sprawdzić postępy nauki zbiorem walidującym, który nie wpływa na wagi sieci neuronowej. Jest to zbiór bierny, Prawidłowy trening powinien minimalizować błąd wyznaczony dla danych uczących jak i walidujących. Gdy w trakcie trenowania błąd walidujący zatrzyma się na dłuższą metę lub zacznie wzrastać następuje przetrenowanie. Na przykładowym wykresie błędów w zależności od iteracji (rys. 2.8.) po prawej stronie linii przerywanej widoczne jest przeuczenie. Przetrenowanie świadczy o tym, że sieć posiada zbyt dużą pojemność w stosunku do problemu jaki ma rozwiązywać. W takim przypadku należy rozpocząć trening od nowa i zatrzymać się w najbardziej optymalnym miejscu lub zmniejszyć potencjalne możliwości nauki sieci, czyli zlikwidować kilka neuronów i rozpocząć ponowny trening. Istnieje również termin niedotrenowania, który

oznacza, że sieć nie jest w stanie nauczyć się w zadowalającym stopniu modelu procesu. Powodem tego stanu sieci jest zbyt niska pojemność SSN. Niedotrenowanie objawia się utknięciem błędu uczącego i walidującego na jednym poziomie pomimo dalszego treningu. Rozwiązaniem jest zwiększenie pojemności SSN przez dodanie dodatkowych neuronów. Gdy osiągniemy zadowalające rezultaty treningu, należy sprawdzić sieć przy użyciu zbioru testowego. Część testująca ma być przetwarzana przez sieć tylko raz w ostatniej fazie treningu i ma za zadanie ostateczną weryfikację procesu uczenia. Weryfikacja sieci zbiorem testowym eliminuje przypadek, w którym sieć w sposób losowy prawidłowo reagowała na zbiory uczące oraz walidujące. Sieci neuronowe mają tendencję do nauki w jak najprostszy sposób. Oznacza to, że w trakcie nauki wychwytują najprostsze cechy z danych wejściowych. Przykładem może być nauka klasyfikatora rozpoznającego psy i koty. Jeżeli zdjęcia psów są bardzo jasne a kotów ciemne sieć z pewnością wyeksponuje tą właściwość w modelu. Gdy do gotowego klasyfikatora wprowadzi się jasne zdjęcie kota, sieć błędnie rozpoznaje zwierzę, dlatego tak ważny jest dobór danych uczących. Należy zręcznie manipulować informacjami trenującymi celem jak najlepszego wykorzystania potencjału sieci neuronowych.



Rysunek 2.8.: Poglądowy wykres przedstawiający przetrenowanie SSN

2.5.6. Uczenie głębokie

Zastosowanie algorytmu wstecznej propagacji w sieciach głębokich, tzn. mających wiele warstw ukrytych, może nie przynieść zadowalających rezultatów. Jest to związane z coraz mniejszą modyfikacją wag wraz z coraz głębszą propagacją błędu w kolejnych warstwach. Zanik modyfikacji wag w początkowych warstwach sieci spowalnia lub uniemożliwia trening SSN. Rozwiązaniem tego problemu jest wprowadzenie uczenia głębokiego, które z założenia trenuje kolejno warstwy blokując uczenie pozostałych [18]. Przykładowy głęboki trening klasyfikatora rozpoczyna się od nauki tylko pierwszej warstwy. Grupa tych neuronów ma w trakcie treningu nauczyć się rozpoznawać najbardziej podstawowe właściwości obiektu. Następnie trenuje się kolejną warstwę, która ma wykrywać kolejne bardziej abstrakcyjne cechy tym razem bazujące na właściwościach z poprzedniej już wytrenowanej warstwy, trening przeprowadza się dla kolejnych warstw. Sieć wytrenowana głęboko posiada hierarchiczne warstwy, które wykrywają kolejno coraz bardziej abstrakcyjne cechy.

3. Robot mobilny Arlo

Robot Alro produkowany przez firmę Parallax (rys. 3.1.), jest w pełni funkcjonalną mobilną platformą z możliwością zamówienia na stronie producenta. Robot składa się z m.in. z układu sterującego (płytka z mikrokontrolerem i peryferiami), układu odpowiedzialnego za bezpośrednie sterowanie silnikami (mostek H), baterii 12V, czterech czujników odległości oraz ramy. Mikrokontroler sterujący robotem jest programowany w języku C przy pomocy dedykowanego środowiska pracy SimpleIde. Producent dostarcza kilka bibliotek do obsługi peryferiów robota oraz liczne poradniki, co znacząco upraszcza tworzenie oprogramowania [14]. Arlo to idealny wybór dla projektów potrzebujących gotowej mobilnej platformy z bardzo uproszczonym programowaniem. W przypadku tego projektu, autor mógł się bardziej skupić na wysokopoziomowych aspektach pracy dyplomowej, takich jak budowanie sztucznej sieci neuronowej i odpowiednie wytrenowanie, natomiast niskopoziomowe odłożyć na drugi plan.

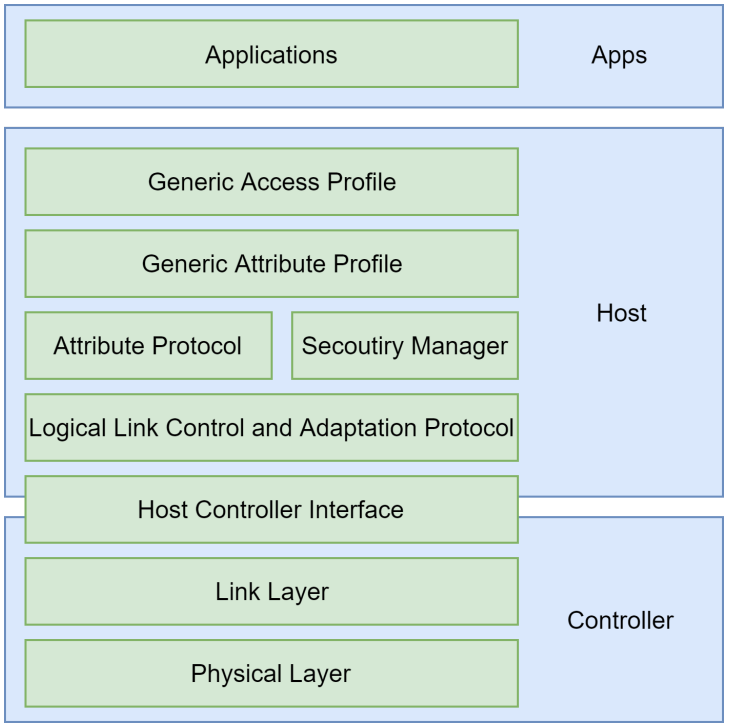
Rysunek 3.1 Robot mobilny Arlo

4. Komunikacja z robotem

Aplikacja desktopowa będzie komunikowała się z robotem mobilnym poprzez standard Bluetooth 4.0 Low Energy (BLE). Charakteryzuje się on bardzo niskim poborem mocy, co czyni je idealnym standardem komunikacji dla urządzeń “Internet of things”, często zasilanymi przez baterie. Wysoka energooszczędność jest zapewniona kosztem transferu, który jest mniejszy niż w wersji 3.0 i wynosi około 1 Mbit/s, Bluetooth Low Energy jest przystosowane do częstego przesyłania niewielkich ilości danych [10]. Ogólne porównanie standardów znajduje się w tabeli 4.1. Stos protokołu Bluetooth Low Energy można podzielić na trzy główne części: *application*, *host* i *controller* (rys. 4.1) [9]:

	Bluetooth	Bluetooth Low Energy
Szerokość kanału	1 MHz	2 MHz
Nominalna prędkość transmisji	1-3 Mbit/s	1 Mbit/s
Liczba węzłów	7	nielimitowana
Zabezpieczenie	56-128 bitowe	AES 128 bitowe

Tabela 4.1.: Porównanie standardów Bluetooth



Rysunek 4.1.: Stos protokołu Bluetooth low energy

Application, jest to warstwa położona najwyżej w hierarchii i odpowiada za interfejs użytkownika oraz obsługę danych.

Host zawiera warstwy odpowiadające m.in. za bezpieczeństwo transmisji, wyszukiwanie urządzeń, nawiązanie połączeń oraz ich konfigurację.

- Generic Access Profile odpowiada za obsługę stanów, w jakich znajduje się moduł komunikujący, stany są zależne od konfiguracji urządzenia (master lub slave) i można je podzielić na:
    - standby, jest to stan domyślny, ustalany po restarcie urządzenia,
    - advertising, stan w trybie slave, urządzenie oczekuje połączenia oraz udostępnia pewne informacje dotyczące modułu,
    - scanning, stan w trybie master, moduł wyszukuje pobliskie urządzenia, które w trybie advertising udostępniają mu takie informacje jak nazwa urządzenia itp. "przeskanowane" urządzenia są gotowe na stałe połączenie,
    - initiating, stan trybu master inicjujący połączenie z wcześniej zeskanowanym urządzeniem,
    - connected, docelowy stan połączenia urządzeń.
- Attribute Protocol (ATT) pozwala na udostępnianie pewnych struktur danych przechowywanych w urządzeniu, tak zwanych atrybutów urządzenia,
- Generic Attribute Profile (GATT) jest warstwą udostępniającą interfejs komunikacyjny pomiędzy połączonymi urządzeniami. Warstwa ta korzysta z atrybutów protokołu ATT. Moduły przyjmują jedną z dwóch funkcji GATT serwer lub klient zależnie od trybu, w jakim pracują. W trybie slave urządzenie staje się serwerem i udostępnia tak zwane charakterystyki klientowi, który jest w trybie master. W uproszczeniu charakterystyka jest interfejsem danych z warstwy ATT. Atrybuty znajdują się w serwerze, które klient może odczytać lub nadpisać przez warstwę GATT, charakterystyki identyfikuje się przez 128 bitowy numer UUID. Część numerów identyfikacyjnych posiada ustandaryzowany początek (16 bitów), przykładowo: Device name - 0x2A00, Heart Rate Measurement - 0x2A37. Standaryzacja ma na celu szybką interpretację charakterystyk przez aplikację klienta. Dodatkowo serwer może kontrolować dostęp do charakterystyk. Każda zmienna może być udostępniana w trybach tylko do odczytu, tylko do nadpisania lub do odczytu i zapisu.

- Logical link control and adaptation protocol odpowiada za enkapsulację i dekapsulację przesyłanych danych.
- Security Manager udostępnia metody parowania i ogólnego zabezpieczenia wymiany danych pomiędzy urządzeniami.

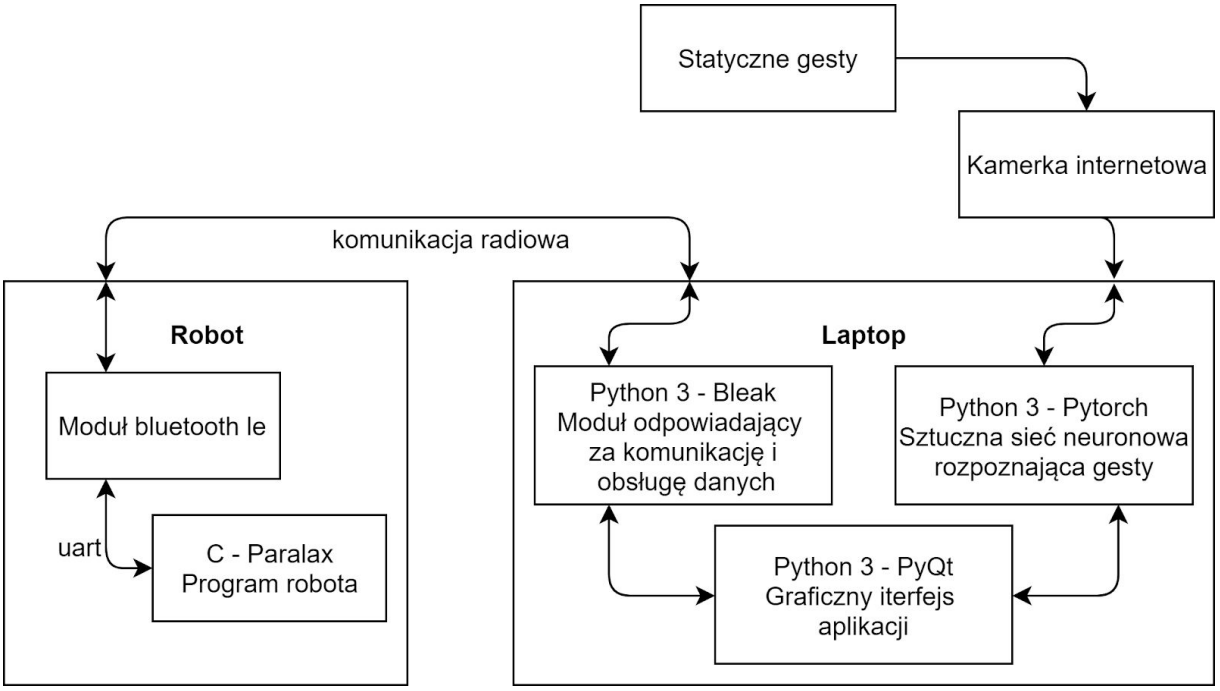
Controller to grupa najniższych warstw odpowiadająca między innymi za komunikację radiową.

- Host Controller Interface udostępnia komunikację pomiędzy hostem a controllerem. Jest to najczęściej zrealizowane przy pomocy programowej komunikacji wewnętrznej lub zewnętrznej zbudowanej na bazie protokołów komunikacyjnych SPI, UART itp.
- Link Layer jest bezpośrednim połączeniem pomiędzy ostatnią warstwą (physical layer) i ma za zadanie przetransformować dane cyfrowe z wyższych warstw w odpowiednie sygnały przekazywane do warstwy fizycznej.
- Physical Layer odpowiada za fizyczną komunikację radiową modułu. Wykorzystuje 40 kanałów w zakresie 2.4000 - 2.4835 GHz.

Moduł Bluetooth robota zostanie połączony z układem sterującym Arlo poprzez UART (ang. *universal asynchronous receiver-transmitter*). Laptop z kolei posiada wbudowaną kartę sieciową, która udostępnia możliwość komunikacji w standardzie Bluetooth 4.0 Low Energy. Do obsługi wymiany informacji zostanie napisany skrypt w Pythonie 3. Program będzie miał za zadania: zabezpieczyć połączenie, przesyłać informacje pomiędzy aplikacją z SSN a modułem Bluetooth robota oraz odpowiednio obsługiwać przesyłane dane. Kamerka internetowa w czasie rzeczywistym będzie przekazywać dane do aplikacji wykorzystującej sztuczne sieci neuronowe w roli klasyfikatora statycznych gestów. Gdy gest zostanie wykryty, odpowiadająca mu komenda zostanie przesłana przez aplikację pośredniczącą w komunikacji do robota. Lista komend wraz z opisami znajduje się w kolejnych rozdziałach poświęconych praktycznej części pracy inżynierskiej.

5. Implementacja projektu

Rozdział omawia praktyczną część pracy inżynierskiej. Rysunek 5.1. przedstawia schemat ideowy całego projektu. Aplikacja desktopowa została napisana dla systemu operacyjnego windows, przy użyciu środowiska programistycznego pythona pyCharm. Pozwala ono na wygodne importowanie bibliotek oraz udostępnia mechanizm debugowania programu. Aplikację można podzielić na trzy moduły: graficznego interfejsu (ang. *graphical user interface* - GUI), sieci neuronowej oraz komunikacyjny. Główną część stanowi GUI. Pozostałe moduły są uruchamiane w osobnych wątkach tego procesu. Program robota został napisany w języku C przy pomocy bibliotek firmy Parallax.



Rysunek 5.1.: Schemat projektu

5.1. Przegląd bibliotek użytych w projekcie

PyTorch jest to jedna z najszybciej rozwijających się bibliotek Pythona wspierających sztuczne sieci neuronowe [2]. Framework został opracowany przez firmę Facebook w roku 2017. Największą zaletą PyTorche'a jest prostota, sieci buduje się szybko dzięki rozbudowanym narzędziom. Dodatkowym atutem jest akceleracja obliczeń przy użyciu GPU poprzez wykorzystanie platformy CUDA (ang. *Compute Unified Device Architecture*), jest to architektura opracowana przez firmę Nvidia pozwalająca na wykorzystanie potencjału kart

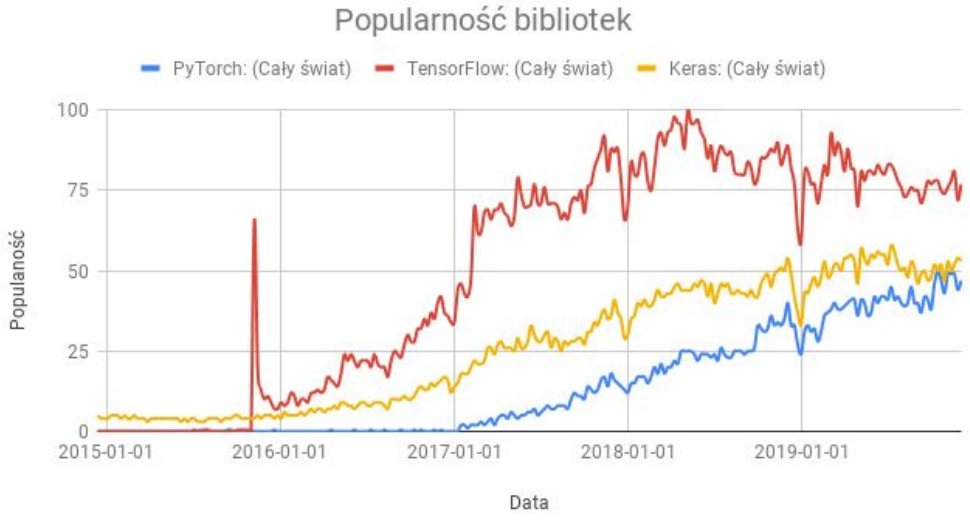
graficznych dzięki językowi programowania bazującego na składni C/C++. Framework jest również dostępny jako biblioteka C++. PyTorch zawiera kilkanaście modułów i funkcjonalności, najważniejszymi są m.in.:

- **Tensor** reprezentuje dane przetwarzane przez sieć. Moduł ten jest rozszerzeniem biblioteki NumPy (jest to popularna biblioteka Pythona wspierającą operacje na wielowymiarowych wektorach, jest znacznie lepiej zoptymalizowana pod względem szybkości niż tradycyjne "pythonowe" listy). Tensory są najczęściej wielowymiarowymi macierzami i w przeciwieństwie do zwykłych kontenerów danych są przystosowane do przechowywania informacji w sztucznych sieciach neuronowych, tzn. przechowują informacje potrzebne do wyznaczenia gradientu. Mogą być przechowywane i równolegle przetwarzane w karcie graficznej.
- **Autograd** jest to moduł ściśle powiązany z tensorami. Pozwala na śledzenie wszystkich operacji w sieci. Każdy tensor w trybie autograd posiada dwie dodatkowe zmienne, wartość gradientu oraz odnośnik do funkcji, która służyła do obliczenia nowej wartości tensora. W trakcie przetwarzania informacji przez sieć czyli forward pass, dana wejściowa jest przetwarzana przez kolejne warstwy w sieci. Obliczane są nowe wartości tensorów i ustawiany jest odnośnik do funkcji tworzącej tensor. Do wyznaczenia gradientów poszczególnych neuronów służy funkcja backward(). Wywołanie jej dla konkretnego tensora pozwala obliczyć gradient każdego poprzedzającego go w obliczeniach tensora przy użyciu zapisanych odnośników do funkcji. Mechanizm automatycznego obliczania gradientów wykorzystują algorytmy treningu sieci. Powyższy opis modułu Autograd jest bardzo ogólny, w rzeczywistości do automatycznego wyznaczania gradientu tworzona jest skomplikowana sieć, która przechowuje informacje o funkcjach użytych w sieci neuronowej, połączeniach między neuronami itp. Mechanizm ten jest bardzo złożony, więcej informacji należy szukać w oficjalnej dokumentacji PyTorch.
- **Optim** moduł przechowuje optymalizatory SSN, czyli algorytmy treningu sieci. Pozwala ustalić parametry algorytmu, takie jak: współczynnik nauki czy pęd nauki. Optymalizatory w module Optim są rozbudowanymi algorytmami gradientowymi lub wstecznej propagacji.
- **Utils** przechowuje dodatkowe funkcjonalności PyTorche'a, takie jak dataset, czyli klasę bazową dla danych używanych w trakcie nauki. Funkcjonalność ta pozwala na

iterację po danych. Kolejną przykładową funkcjonalnością Utils jest dataloader. Jest to klasa pośrednicząca w treningu łącząca dane w batche. Udostępnia losowe iterowanie po danych oraz pozwala na równoległe wczytywanie informacji.

- **NN** jest to zestaw narzędzi do budowania sieci neuronowej. Zawiera klasę bazową sieci, funkcje aktywacji, rodzaje warstw, funkcje kosztu sieci itp.

Alternatywnymi frameworkami Pythona są m.in. TensorFlow (TF) oraz Keras. TF jest obecnie najpopularniejszą biblioteką wspierającą uczenie maszynowe. Została opracowana przez Google w 2015 roku. Tworzenie sieci przy użyciu TensorFlow jest znacznie trudniejsze niż przy zastosowaniu PyTorche'a. Wynika to z budowy i struktury biblioteki, TF jest znacznie mniej intuicyjny i bardziej złożony. Graf sieci w TensorFlow jest statyczny. Oznacza to, że nie mamy możliwości zmiany struktury SSN w trakcie jej działania. Pytorch natomiast udostępnia tę dynamikę. Keras z kolei jest frameworkiem opracowanym przez François Cholleta. Jest to biblioteka zawierająca w sobie TensorFlow tworzący prostszy i bardziej przejrzysty interfejs pomiędzy użytkownikiem a wewnętrznym TF. Keras wprowadza pewną warstwę abstrakcji do TensorFlowa. Pytorch został zaprojektowany później niż pozostałe biblioteki, dlatego projektanci mogli zwrócić uwagę na wady już istniejących frameworków i wyciągnąć na ich podstawie wnioski. Popularność bibliotek przedstawioną na przestrzeni kilku lat przedstawia wykres 5.2. Warto również wspomnieć o tym, że Google kilka miesięcy przed publikacją tej pracy (30.09.2019) opracował TensorFlow 2.0, rozszerzenie obecnego TF. Wprowadza szereg usprawnień i upraszcza proces tworzenia SSN. Być może nowy framework zdeklasuje pozostałe w najbliższej przyszłości.



Rysunek 5.2 Popularność wybranych bibliotek wspierających SSN

Bleak jest aktualnie jedyną biblioteką Pythona 3 obsługującą komunikację w standardzie Bluetooth 4.0 Low Energy dla systemu operacyjnego windows [12]. Biblioteka jest relatywnie nowa i dostarcza bardzo uproszczone funkcjonalności. Przykładowo brakuje obsługi ustalania połączenia dla modułów z ustawionym kodem pin. Framework jest dostępny dla systemów operacyjnych linux, windows oraz macOS. W przypadku windowsa biblioteka korzysta z platformy .NET ściśle związaną z systemem operacyjnym.

PyQt jest frameworkiem Pythona 3 dostarczającym narzędzia do budowy graficznego interfejsu użytkownika [13]. PyQt jest nadbudową bardzo popularnej biblioteki Qt języka C++, udostępniającą duże możliwości Qt w pythonie. Framework pozwala na bardzo szybkie budowanie GUI, szczególnie dzięki takim narzędziom jak Qt Designer. Jest to program pozwalający budować graficzny interfejs przy pomocy gotowych elementów, które można umieścić w głównej aplikacji techniką drag and drop. Największą zaletą Qt Designera jest możliwość podglądu efektów w czasie rzeczywistym. Gotową aplikację zapisuje się w pliku z rozszerzeniem .ui a następnie konwertuje się do kodu Pythona dzięki programowi pyuic5.

5.2. Implementacja komunikacji

Moduł komunikacyjny ma za zadanie zabezpieczyć połączenie. Niestety biblioteka bleak nie dostarcza narzędzi do obsługi pinu bezpieczeństwa modułu BLE, dlatego zbudowano zabezpieczenia programowe. W celu podniesienia bezpieczeństwa każde skomunikowane z Arlo urządzenie ma podać w ciągu 3 sekund 20 znakowy kod zabezpieczający. W przeciwnym wypadku robot zostanie zablokowany. Ponadto moduł ma nadzorować wykonywanie komend. Każde polecenie posiada trzy stany: inicjacji, wykonywania oraz zakończenia, każde przejście między stanami ma być potwierdzone przez robota. Nakłada się również ograniczenia czasowe na wykonywanie kolejnych etapów komendy. Moduł w celach bezpieczeństwa ma dodatkowo obsługiwać specjalną komendę emergency stop. Ma ona najwyższy priorytet, polecenie to ma natychmiast anulować wykonywanie wszystkich komend i zatrzymać robota.

Protokół komunikacji w warstwie aplikacji będzie bardzo uproszczony ze względu na nieskomplikowane komendy wysyłane do robota. Ramka danych będzie się składała z 5 bajtów, w których można zapisać ośmiobitową liczbę bez znaku w zakresie 0-255 (rys.5.3.), (tab.5.1.).

bajt 1: identyfikator urządzenia	bajt 2: komenda	bajt 3: identyfikator komendy	bajt 4: identyfikator wiadomości	bajt 5: dodatkowe informacje

Rysunek 5.3.: Ramka danych w protokole komunikacji

Bajt	**Możliwe wartości bajtu**
bajt 1: identyfikator urządzenia	1 - robot mobilny 2 - aplikacja desktopowa
bajt 2: komenda	1-12 - poszczególne komendy wysyłane do robota 13 - awaryjny stop
bajt 3: identyfikator komendy	1-255 identyfikator polecenia/komendy
bajt 4: identyfikator wiadomości	1-255 numer wiadomości w danej sesji
bajt 5: dodatkowe informacje	1 - pusta informacja 2 - potwierdzenie odbioru komendy przez robota 3 - potwierdzenie zakończenia egzekucji komendy 4 - błąd komendy 5 - błąd działania robota 6 - gotowość robota (zaakceptowanie kodu bezpieczeństwa) 7 - odmowa dostępu (odrzucenie kodu bezpieczeństwa lub przekroczenie czasu oczekiwania na wiadomość)

Tabela 5.1.: wartości i znaczenie poszczególnych bajtów w protokole

5.2.1. Opis komend sterujących robotem

Komendy wysyłane do robota można podzielić na dwie główne grupy: sterujące oraz nastawcze. Pierwsze z nich służą do poruszania robotem, drugie z kolei pozwalają na zmianę parametrów wykonywania komend sterujących. Poniżej znajduje się opis poleceń (numer odpowiada id komendy).

Komendy sterujące:

- **Default 1** - komenda terminująca rozpoznawanie komend, nie wpływa na działanie robota i służy jedynie do blokowania wysyłania kolejnych komend w trakcie wykonywania już innego polecenia przez robota,

- **Turn left 2** - komenda wymuszająca obrót robota w lewo o modyfikowalny kąt alfa który mieści się w zakresie (10 - 180 stopni),
- **Turn right 3** - lustrzane polecenia komendy turn left wymuszająca obrót robota w prawo o alfa,
- **Turn around 4** - polecenie obrotu robota o 180 stopni,
- **Forward 5** - komenda nakazująca robotowi jazdę do przodu, pokonując drogę o modyfikowanej długości beta w zakresie (20 - 300 jednostek),
- **Backward 6** - komenda nakazująca robotowi jazdę do tyłu, pokonując drogę o długości beta.

Komendy nastawcze:

- **Speed up 7 -** zwiększa współczynnik szybkości wykonywania komend sterujących, możliwe wartości współczynnika wyraża się w procentach maksymalnej szybkości robota (10% - 100%), użycie komendy zwiększa współczynnik o 10 punktów procentowych,
- **Slow down 8** - polecenie zmniejsza współczynnik szybkości o 10 punktów procentowych,
- **Bigger turn angle 9** - zwiększa kąt obrotu robota alfa o 18 stopni,
- **Smaller turn angle 10 -** zmniejsza kąt obrotu robota alfa o 18 stopni,
- **Bigger step 11** - zwiększa drogę beta używaną w komendach forward i backward o 20 jednostek,
- **Smaller step 12 -** zmniejsza drogę beta o 20 jednostek.

Istnieje dodatkowo komenda emergency stop o id 13 zatrzymująca robota i anulująca aktualnie wykonywaną komendę sterującą.

5.3. Program robota

Układ sterujący robotem znajduje się na płytce propeller. Zawiera ona wszystkie potrzebne peryferia do poprawnego funkcjonowania Arlo, takie jak: interfejs komunikacyjny USB, slot na kartę SD, mikrokontroler, wyjścia pinów, małą płytkę stykową itp. Programy napisane w języku C kompiluje się i wgrywa robotowi dzięki środowisku programistycznym dedykowanym do produktów firmy Parallax SimpleIde. Najważniejszą częścią programu robota jest możliwość komunikowania się z laptopem. Jak wspomniano, do komunikacji

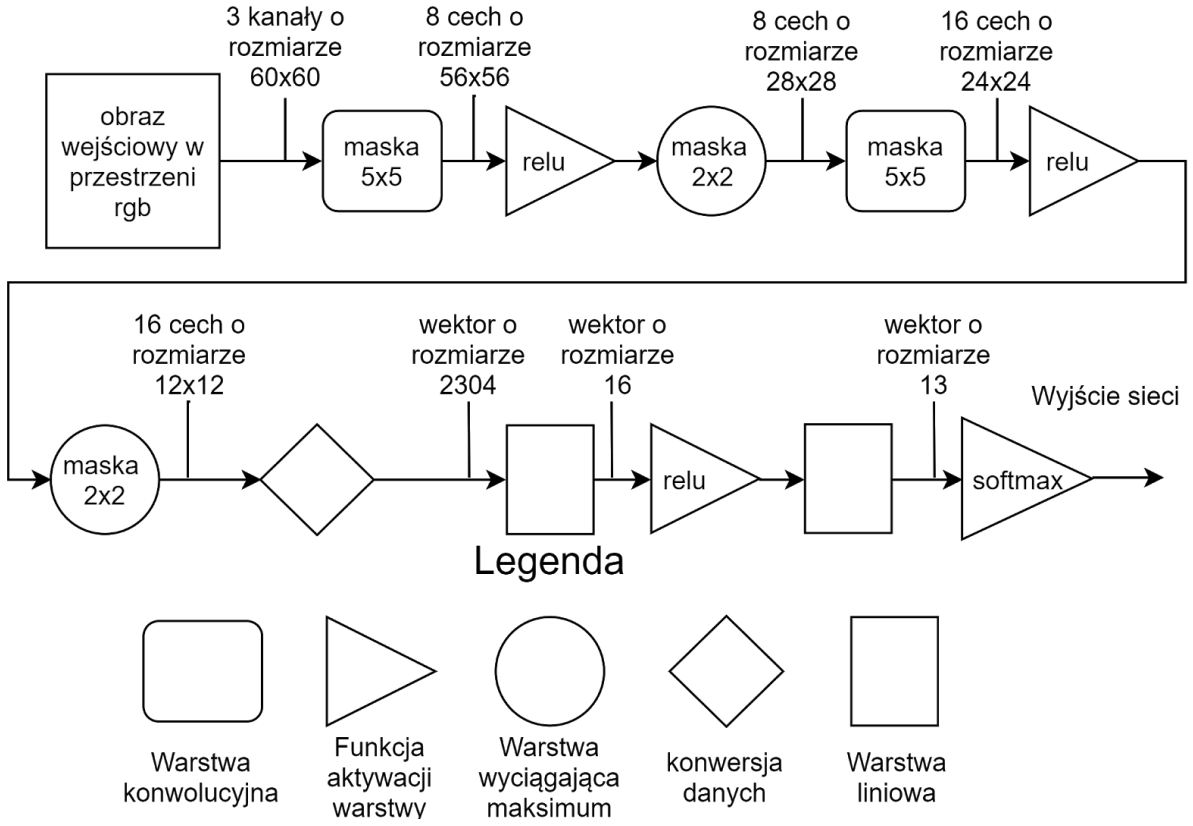
bezprzewodowej użyto moduł Bluetooth 4.0 Low Energy, moduł odbiera ramki danych i przekazuje je przez interfejs UART do mikrokontrolera robota. Dzięki bibliotekom dostarczonym przez firmę Parallax część z dwustanowych pinów płytki można zamienić na interfejs szeregowy do odbioru danych z modułu BLE. Moduł zasilany jest napięciem 5V, natomiast pozostałe piny pracują w logice 3.3V. Oznacza to, że komunikacja UART również musi się odbywać w logice 3.3V. Mikrokontroler sterujący robotem wspiera stany napięciowe 0 - 3.3V, natomiast Arduino (otwarta platforma programistyczna systemów wbudowanych, płytki Arduino programuje się w środowisku Arduino IDE przy użyciu języka C/C++), które było używane w testowaniu komunikacji pracuje tylko w logice 5V. Chcąc, aby komunikacja była bezproblemowa należało stworzyć odpowiednio obniżający napięcia dzielnik. Jak wspomniano w poprzednim rozdziale komunikacja jest zabezpieczona kodem dostępu do robota. W przypadku błędnego kodu włączana jest czerwona dioda elektroluminescencyjna (ang. *led*) sygnalizująca zablokowanie urządzenia, w takim przypadku należy ponownie połączyć się z robotem i podać prawidłowy kod. Płytka propeller posiada dodatkową zieloną diodę (led) informującą o statusie połączenia radiowego modułu BLE. Komendy sterujące zrealizowane są przy użyciu funkcji ustawiającej prędkości dwóch kół robota, następnie w głównej pętli programu sprawdzane są odczyty danych z enkoderów wyznaczających obroty kół robota. Arlo jest zatrzymywany po przekroczeniu pewnych obrotów kół zależnych od komend. Natomiast komendy nastawcze zmieniają współczynniki oraz progi zapisane w pamięci mikrokontrolera potrzebne do realizacji poleceń sterujących. Każda komenda odebrana przez robota wymaga odesłania potwierdzenia. W przypadku poleceń sterujących mikrokontroler wysyła potwierdzenie odbioru komendy oraz zakończenia wykonywania zadania. Natomiast w przypadku poleceń nastawczych robot odsyła tylko informację o zakończeniu realizacji komendy.

5.4. Implementacja sieci neuronowej

Moduł sieci neuronowej ma za zadanie pobrać obraz z kamerki internetowej, przygotować dane do obróbki i poprzez sieć neuronową rozpoznać gest użytkownika aplikacji sterujący robotem.

5.4.1. Struktura sieci neuronowej

Do rozpoznawania statycznych gestów na obrazie została zastosowana sieć konwolucyjna. Strukturę sieci przedstawia rysunek 5.4. Sieć ma za zadanie wykrywać trzynaście różnych gestów.



Rysunek 5.4.: struktura konwolucyjnej sieci neuronowej

Natywny obraz otrzymywany z kamerki internetowej jest w przestrzeni RGB i posiada rozdzielczość 640x480 pixeli. Biblioteka OpenCV użyta w projekcie do obsługi pobierania i wyświetlania obrazów przechowuje ramki w macierzach NumPy. Przed przetwarzaniem danych w sieci należy je odpowiednio przetransformować. W tym celu została użyta biblioteka autorstwa Jima Bohuslava [14] udostępniająca szereg transformat dla obrazów OpenCV. Wykorzystano pięć operacji: przycięcie obrazu do rozmiaru 480x480 pixeli, przeskalowania do rozmiaru 60x60, odwrócenia wertykalnego obrazu (uzyskanie efektu lustra), transformacji danych w tensor (dodatkowo konwersja w tensor skaluje poszczególne pixele w zakresie 0:1) oraz normalizacji danych w zakres -1:1 dla lepszych efektów treningowych sieci. Wstępnie przetworzony obraz zostaje przekazany do sieci neuronowej (rys. 5.4.), gdzie przechodzi przez kolejne warstwy, początkowo jest to konwolucja z funkcją

relu oraz wyciąganie danych. Sekwencja powtarza się dwukrotnie, na wyjściu obróbki konwolucyjnej otrzymujemy 16 cech o rozmiarze 12x12 pikseli. W tym miejscu następuje konwersja wszystkich cech w jeden długi wektor (wypłaszczenie). Jest to przygotowanie informacji do przetwarzania w warstwach liniowych, czyli “zwykłych neuronach”. Pierwsza z nich posiada funkcję aktywacji relu, natomiast ostatnia służy do ostatecznej klasyfikacji i użyto w tym przypadku softmax.

5.4.2. Trening klasyfikatora

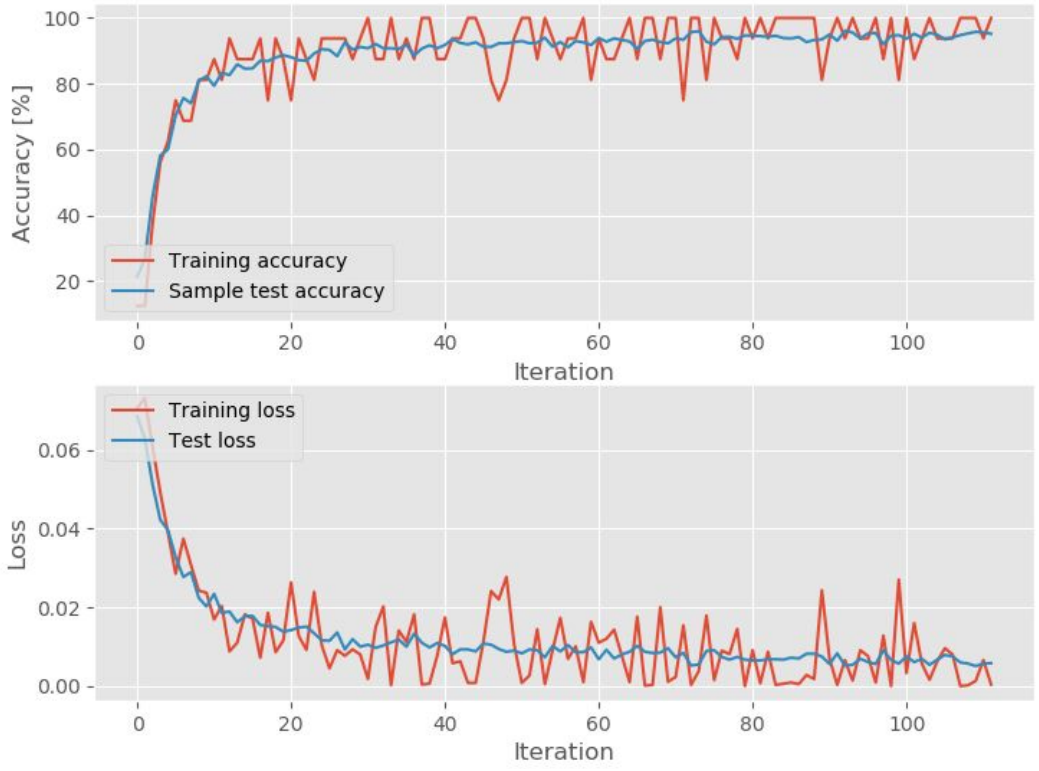
Do treningu sieci neuronowej użyto 33800 zdjęć. Oznacza to, że na pojedynczy gest przypada 2600 obrazów. Są to pliki przygotowane przez autora projektu, przykładowe fragmenty zdjęć przedstawiające gesty oraz odpowiadające im komendy przedstawiają rysunki 5.5. oraz 5.6.

Rysunek 5.5.: Gesty sterujące robotem, numer odpowiada id komendy

Rysunek 5.6.: Gest awaryjnego stopu

Ze względu na właściwości sieci (tendencja do wykrywania najprostszych cech) i skuteczniejszą naukę, zdjęcia gestów były robione w różnych miejscach, tzn. posiadają zróżnicowane tła. Ponadto aby SSN mogły lepiej generalizować wykrywane komendy, dane przedstawiają gesty kilku osób. Zdjęcia zostały podzielone na trzy grupy: walidującą 10%, testową 10% oraz treningową 80%. W trakcie treningu pliki uczące były wczytywane losowo, były łączone w 16 zdjęciowe batche celem przyspieszenia treningu oraz zastosowano technikę augmentacji danych. Metoda ta lekko modyfikuje dane wejściowe celem sztucznego powiększenia puli treningowej. W projekcie użyto 3 losowo modyfikujące funkcje: wycinania okna 400x400 pikseli przed skalowaniem danych do rozmiaru 60x60, zmiany jasności oraz kontrastu zdjęć. Jak się okazało, w trakcie treningów augmentacja znacząco poprawiła działanie sieci neuronowej. Trening klasyfikatora przeprowadzono przy użyciu karty graficznej, która przyspieszyła proces uczenia o około 30%. Ostatecznie zadowalające wyniki udało się uzyskać po 13 epokach (jest to określenie powszechnie używane w środowisku machine learningu oznaczające liczbę cykli przetworzeń danych uczących) - trwało to około 1.5h. Test walidacyjny przeprowadzano co 200 batchy uczących. Ocena była przeprowadzana

dla 64 losowych batchy pochodzących ze zbioru walidacyjnego, natomiast wynikami były średnia skuteczność oraz średnia wartość funkcji błędu. Przebieg treningu przedstawia rysunek 5.7. Pierwszy wykres przedstawia skuteczność sieci drugi natomiast prezentuje wartości funkcji kosztu. Niebieskie przebiegi odnoszą się do danych walidacyjnych, są one znacznie gładsze niż czerwone, spowodowane jest to uśrednieniem wyników z testu walidacyjnego. Z wykresu można wywnioskować, że w końcowych etapach treningu dla danych walidacyjnych prawidłowość klasyfikacji wynosiła około 95%. Ostatecznie potwierdzenie poprawności działania sieci przeprowadzono dla danych testowych, w tym przypadku średnia skuteczność wynosiła 97.72 %.



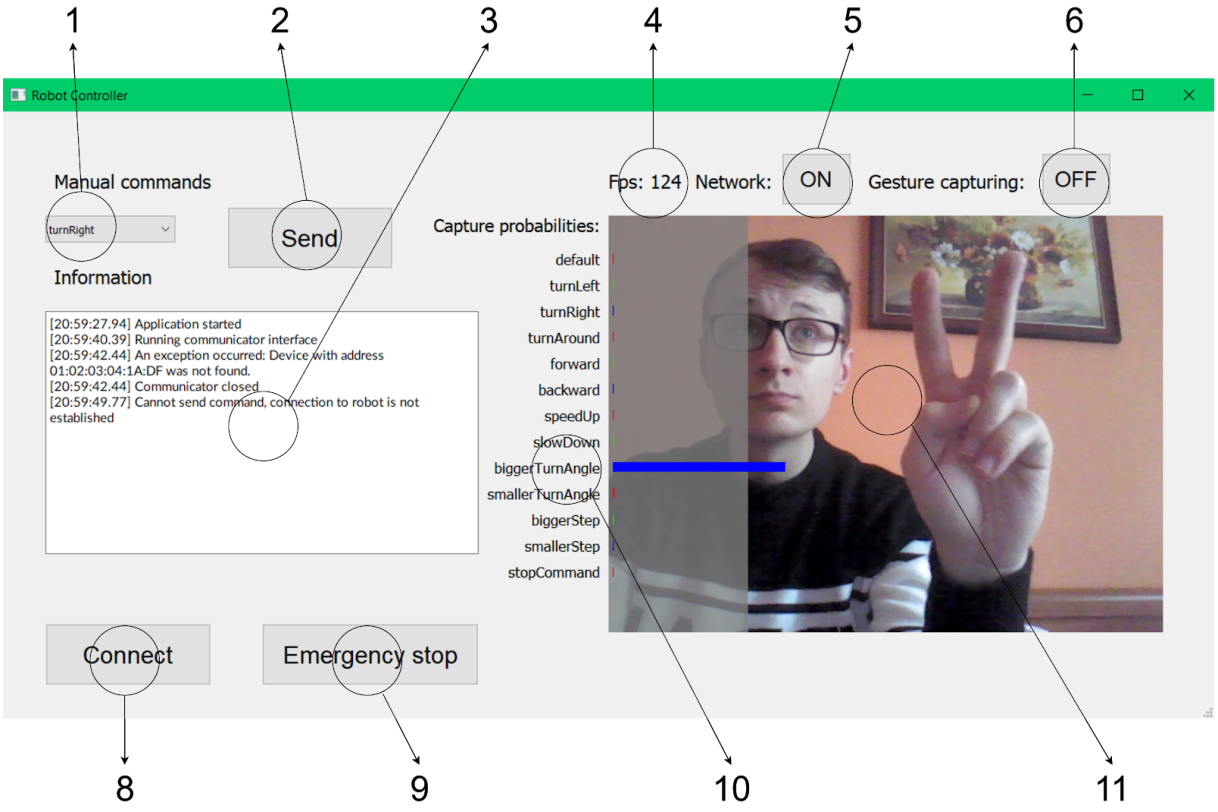
Rysunek 5.7.: Wykresy przedstawiające przebieg treningu

Parametry poszczególnych elementów warstw sieci zostały dobrane w drodze eksperymentalnej, z uwzględnieniem stabilności pracy systemu w czasie rzeczywistym oraz maksymalizacji możliwości adaptacyjnych. Początkowo zbudowano bardzo pojemną sieć, którą poprzez analizę wykresów treningowych kolejno redukowano. Ostatecznie uzyskano możliwie najmniejszą pojemność sieci, która była w stanie poprawnie funkcjonować. Zabieg

ten optymalizuje szybkość działania systemu oraz minimalizuje zasoby pochłaniane przez program.

5.5. Graficzny interfejs aplikacji

Graficzny interfejs aplikacji umożliwia łatwe sterowania robotem. Użytkownik ma możliwość połączenia się z robotem, manualnego wysyłania komend oraz wykorzystania automatycznego rozpoznawania poleceń dzięki modułowi sieci neuronowej. GUI posiada ponadto podgląd przetwarzanych obrazów z kamery oraz prawdopodobieństw wystąpień poleceń w danej klatce wideo. Aplikacja działa w czasie rzeczywistym z szybkością przetwarzania ramek wideo na sekundę (ang. *frames per second*) wynoszącą około 15-30. Wygląd GUI wraz z opisem funkcjonalności znajduje się na rysunku 5.8.



Rysunek 5.8. Graficzny interfejs aplikacji

Opis rysunku 5.8.:

1. Manualny wybór komendy,
2. Manualne wysłanie komendy,
3. Konsola wyświetlająca informacje kontrolne, takie jak: stasus połączanie aplikacji z robotem, status wysłanych komend, wykrytą komendę przez sieć neuronową itp.,
4. Licznik przetwarzanych klatek na sekundę (ang. *frames per second*),
5. Włączenie lub wyłączenie przetwarzania obrazu przez sieć neuronową,
6. Włączenie lub wyłączenie automatycznego wysyłania wykrytych komend do robota. Jeżeli wykryto komendę dłużej niż 2 sekundy, jest ona automatycznie przesyłana do Arlo. Przycisk jest aktywowany również przez wciśnięcie klawisza control,
7. Ustalenie lub zerwanie połączenia z robotem,
8. Awaryjny stop, jest to specjalna komenda anulująca aktualnie wykonywaną komendę. Przycisk jest aktywowany również przez wciśnięcie klawisza Shift,
9. Kolejne komendy oraz prawdopodobieństwo wystąpienia na aktualnie przetwarzanym obrazie przedstawione jako kolorowe paski po prawej stronie (im dłuższy pasek tym wyższe prawdopodobieństwo),
10. Podgląd przetwarzanych klatek z kamerki internetowej w rozdzielczości 640x480 pixeli, szary obszar obrazu jest ignorowany w operacji wykrywania komend.

5.6. Testy sterowania robotem

Przed przystąpieniem do finalnego testowania sterowania robotem, należało sprawdzić funkcjonalność automatycznego wykrywania komend w praktyce (bez udziału robota). Aplikacja w czasie rzeczywistym, dla każdego obrazu pochodzącego z kamery internetowej wyliczała prawdopodobieństwa wykrycia poleceń (komenda o największym prawdopodobieństwie staje się wykrytym poleceniem danej ramki), gdy komenda jest wykrywana dłużej niż dwie sekundy jest automatycznie wysyłana do robota. Ten etap można nazwać właściwym wykryciem polecenia sterującego Arlo. Udostępnienie podglądu prawdopodobieństw (rozdział 5.5. opis rysunku 5.8) oraz wprowadzenie ram czasowych sprawia, że użytkownik aplikacji może korygować błędy klasyfikacji w czasie rzeczywistym (poprzez zmianę położenia dłoni, oddalenie ręki od kamery itp.). Doprowadzając do przesłania poprawnego polecenia robotowi. W przypadku wysłania błędnej komendy można

skorzystać z funkcjonalności awaryjnego stopu. Wykrywanie komend było testowane przez grupę pięciu osób. Wyniki były zadowalające, sieć neuronowa poprawnie wykryła wszystkie komendy (mowa o właściwym wykryciu, z uwzględnieniem korekcji). Analizując klasyfikację poleceń poszczególnych klatek wideo zauważono, że sieć świetnie radziła sobie z gestami, które pojawiały się w środkowej części kadru kamerki internetowej. W pozostałych miejscach pojawiały się błędy, co spowodowane jest niedoskonałością zbioru uczącego, który zawierał w znacznej większości gesty znajdujące się w centralnej części obrazów.

Niestety okazało się, że robot mobilny Arlo posiadał mechanicznie uszkodzony lewy enkoder, który był kluczowy w realizacji tego projektu. W przypadku tak niefortunnej awarii autor zdecydował się na sterowanie robotem z wykorzystaniem tylko jednego sprawnego czujnika. Komendy wykonywane przez robota są symetryczne, dlatego istnieje taka możliwość. Komunikacja Bluetooth po stronie aplikacji desktopowej była niezawodna, natomiast moduł Bluetooth odbierający komendy okazał się częściowo wadliwy. Pomimo ustawienia bardzo niskiej szybkości transmisji, która wynosiła 2400 bod (ang. *baud*), niekiedy pojawiały się błędy w przesyłaniu informacji, przykładowo w kodzie zabezpieczającym połączenie brakowało środkowej wartości. Po dłuższej eksploatacji w trakcie właściwych testów moduł ostatecznie uległ trwałemu uszkodzeniu, co zablokowało możliwość bezprzewodowej komunikacji. Niestety w takim wypadku autor tymczasowo zrezygnował ze standardu Bluetooth Low Energy, na rzecz komunikacji szeregowej, co wymagało wprowadzenia niewielkich zmian kodu aplikacji użytkownika (zmodyfikowany został moduł komunikacji). Natomiast kod robota mobilnego pozostał praktycznie w niezmienionej formie. Dalsze testy były przeprowadzane przy użyciu komunikacji przewodowej, która okazała się niezawodna. Niestety długość przewodu znacząco ograniczała ruchy robota. Arlo został przetestowany pod względem poprawności wykonywania komend, ponadto w trakcie testów szczególnie zwrócono uwagę na niezawodność komendy awaryjnego stopu. Wyniki praktycznych testów były zadowalające, robot poprawnie reagował na polecenia.

6. Podsumowanie

Celem niniejszej pracy było zaprojektowanie systemu pozwalającego na sterowanie robotem za pomocą statycznych gestów dłoni. Praca nad projektem została podzielona na trzy części. Pierwszą z nich stanowiło zbudowanie oraz wytrenowanie sieci neuronowej do rozpoznawania statycznych gestów dłoni, oraz umożliwienie wysyłania wykrytych komend. Interfejs aplikacji udostępnia funkcjonalności: nawiązywania połączenia z robotem, manualnego wysyłania komend, automatycznego wykrywania komend przy użyciu SSN, podglądu prawdopodobieństw wykrytych poleceń oraz awaryjnego zatrzymania robota. Dzięki wykorzystaniu biblioteki PyQt graficzny interfejs użytkownika jest intuicyjny oraz przejrzysty, sprawia to, że aplikacja jest bardzo prosta w obsłudze. Kolejnym zadaniem było napisanie programu robota, który miał umożliwiać wykonywanie komend wysyłanych z głównej aplikacji. Dzięki bibliotekom udostępnionym przez producenta, zadanie zostało zrealizowane bez przeszkód. Niestety trzecia część projektu, czyli zapewnienie komunikacji bezprzewodowej pomiędzy programami w standardzie Bluetooth Low Energy, nie została w pełni zrealizowana. Spowodowane to było wadliwym modułem BLE, który w trakcie testów uległ uszkodzeniu. Standard Bluetooth został zastąpiony komunikacją przewodową (szeregową), która okazała się niezawodna. Nie zmienia to głównego założenia projektu, lecz znacząco ogranicza mobilność Arlo.

Robot posiada wbudowane czujniki odległości, które można w przyszłości wykorzystać w roli dodatkowego zabezpieczenia (automatyczne zatrzymanie robota, gdy napotka przeszkodę). Ponadto rozpoznawanie statycznych gestów można poszerzyć o dynamikę poprzez zastosowanie rekurencji w konwolucyjnej sieci neuronowej. Znacząco podnosi to poziom skomplikowania projektu, ale dynamiczne gesty są znacznie bardziej intuicyjne.

Badania w tej pracy dyplomowej zostały wsparte grantem Narodowego Centrum Nauki, Polska DEC2016/21/B/ST7/02220.

Wykaz Rysunków

Rysunek 2.1: Quasar Jarosz, Komórka neuronu, url:
https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg

Rysunek 2.2: użytkownik polskiego projektu wikipedii: Andrut, url:

https://commons.wikimedia.org/wiki/File:Neuron_McCullocha-Pittsa.svg

Rysunek 2.3: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 2.4: opracowanie własne, stworzone przy użyciu programu Matlab

Rysunek 2.5: opracowanie własne, stworzone przy użyciu programu Matlab

Rysunek 2.6: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 2.7: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 2.8: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 3.1: Parallax, url:
https://www.google.com/search?biw=1536&bih=722&tbm=isch&sxsrf=ACYBGNStdzNVpJne5x4Z6zoCZMMXOuOrDA%3A1575495172522&sa=1&ei=BCboXYy8H4zRrgTLlp5A&q=Arlo+parallax&oq=Arlo+para&gs_l=img.3.0.35i39j0i30l4j0i5i30l5.1286.2818..5400...3.0..1.157.843.7j3.....0....1..gws-wiz-img.....0j0i19j0i67.CFv5PKkxLPA#imgrc=fiil-14RyolOSM:

Rysunek 4.1: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 5.1: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 5.2: Wykres stworzony w programie ms Excel przy użyciu danych z Google Trends, url:
https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11gd3905v1,%2Fg%2F11bwp1s2k3,%2Fg%2F11c1r2rvnp

Rysunek 5.3: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 5.4: opracowanie własne, stworzone przy użyciu aplikacji DrawIo, url
https://www.draw.io

Rysunek 5.5: opracowanie własne, zdjęcia pochodzą ze zbiorów uczących sieć neuronową

Rysunek 5.6: opracowanie własne, zdjęcie pochodzi ze zbiorów uczących sieć neuronową

Rysunek 5.7: Wykres stworzony przy użyciu biblioteki pythona matlibplot

Rysunek 5.8: opracowanie własne

Bibliografia

[1] Horzyk A., *Sztuczne sieci neuronowe*, Prezentacja, dostęp 10.10.2019, url: http://home.agh.edu.pl/~horzyk/ahdydaktyka.php

[2] Oficjalna strona internetowa biblioteki PyTorch, dostęp 2.10.2019, url: https://pytorch.org/tutorials/

[3] Yegnanarayana B., *Artificial neural networks*, New Delhi Wyd. Prentice Hall, 2006, dostęp 15.10.2019 url: https://books.google.pl/books?id=RTtvUVU_xL4C&printsec=frontcover&dq=neural+network&hl=pl&sa=X&ved=0ahUKEwiytpiFjYHmAhWns4sKHZAqBrsQ6AEIKTAA#v=onepage&q=convolutional&f=false , 2006.

[4] Tadeusiewicz R, *sieci neuronowe,* 1996 dostęp 13.10.2019

[5] Tadeusiewicz R & Szalaniec M., *Leksykon sieci neuronowych*, Wrocław, Wyd. Projekt Nauka 2015, dostęp 11.10.2019

[6] Cybenko G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems,* 1989, dostęp 23.10.2019, url: https://pdfs.semanticscholar.org/05ce/b32839c26c8d2cb38d5529cf7720a68c3fab.pdf

[7] StatSoft, electronic statistics textbook, dostęp 20.11.2019, url: https://www.statsoft.pl/textbook/stathome_stat.html?https%3A%2F%2Fwww.statsoft.pl%2Ftextbook%2Fstneunet.html

[8] Arabas J. & Cichosz P., Skrypt 2 *algorytmy uczenia sieci neuronowych*, dostęp 18.11.2019, url: http://www.neurosoft.edu.pl/media/pdf/tkwater/sztuczna_inteligencja/2_alg_ucz_ssn.pdf

[9] Heydon R., *Bluetooth low energy,* Prezentacja, dostęp 23.10.2019, url: https://www.cl.cam.ac.uk/teaching/1819/MobSensSys/mobile-10.pdf

[10] Stępień J. & Kołodziej J. & Machowski W., *Niskoenergetyczne bezprzewodowe personalne sieci sensorowe,* Artykuł Naukowy 2017, dostęp 7.11.2019, url: http://pe.org.pl/articles/2017/2/43.pdf

[11] Glorot X. & Bordes A. & Bengio Y, *Deep sparse rectifier neural networks,* Artykuł Naukowy 2011, dostęp 5.11.2019.

[12] Oficjalna strona internetowa biblioteki Bleak, dostęp 11.12.2019, url: https://github.com/hbldh/bleak

[13] Oficjalna strona internetowa biblioteki PyQt, dostęp 17.12.2019, url: https://doc.qt.io/qtforpython/#documentation

[14] Bohnslav J., biblioteka transformat obrazów OpenCV kompatybilna z pytorch, dostęp 15.12.2019, url: https://github.com/jbohnslav/opencv_transforms

[15] Strona internetowa producenta robota Arlo, dostęp 14.10.2019, url: https://www.parallax.com/product/Arlo-robotic-platform-system

[16] Longstaff A., *Neurobiologia*, Warszawa Wyd. PWN, 2009, dostęp 21.10.2019.

[17] Williams RW. & Herrup K., *The control of neuron number,* Annu Rev Neurosci 1988, dostęp 21.10.2019.

[18] Horzyk A,, *Uczenie głębokich sieci neuronowych* Prezentacja, dostęp 10.10.2019, url: http://home.agh.edu.pl/~horzyk/ahdydaktyka.php,